Loops & Complexity
in
DIGITAL SYSTEMS

*Lecture Notes on Digital Design
in
Giga-Gate/Chip Era
(work in endless progress)

Gheorghe M. Ștefan

– 2019 version –
Introduction

... theories become clear and ‘reasonable’ only after incoherent parts of them have been used for a long time.

Paul Feyerabend

The price for the clarity and simplicity of a ‘reasonable’ approach is its incompleteness.

Few legitimate questions about how to teach digital systems in Giga-Gate Per Chip Era (G2CE) are waiting for an answer.

1. What means a complex digital system? How complex systems are designed using small and simple circuits?

2. How a digital system expands its size, increasing in the same time its speed? Are there simple mechanisms to be emphasized?

3. Is there a special mechanism allowing a “hierarchical growing” in a digital system? Or, how new features can be added in a digital system?

The first question occurs because already exist many different big systems which seem to have different degree of complexity. For example: big memory circuits and big processors. Both are implemented using a huge number of circuits, but the processors seem to be more “complicated” than the memories. In almost all text books complexity is related only with the dimension of the system. Complexity means currently only size, the concept being unable to make necessary distinctions in G2CE. The last improvements of the microelectronic technologies allow us to put on a Silicon die around a billion of gates, but the design tools are faced with more than the size of the system to be realized in this way. The size and the complexity of a digital system must be distinctly and carefully defined in order to have a more flexible conceptual environment for designing, implementing and testing systems in G2CE.

The second question rises in the same context of the big and the complex systems. Growing a digital system means both increasing its size and its complexity. How are correlated these two growing processes? The dynamic of adding circuits and of adding adding features seems to be very different and governed by distinct mechanisms.

1Paul Feyerabend (b.1924, d.1994), having studied science at the University of Vienna, moved into philosophy for his doctoral thesis. He became a critic of philosophy of science itself, particularly of “rationalist” attempts to lay down or discover rules of scientific method. His first book, Against Method (1975), sets out “epistemological anarchism”, whose main thesis was that there is no such thing as the scientific method.
The third question occurs in the hierarchical contexts in which the computation is defined. For example, Kleene’s functional hierarchy or Chomsky’s grammatical hierarchy are defined to explain how computation or formal languages used in computation evolve from simple to complex. Is this hierarchy reflected in a corresponding hierarchical organization of digital circuits? It is obvious that a sort of similar hierarchy must be hidden in the multitude of features already emphasized in the world of digital circuits. Let be the following list of usual terms: boolean functions, storing elements, automata circuits, finite automata, memory functions, processing functions, …, self-organizing processes, …. Is it possible to disclose in this list a hierarchy, and more, is it possible to find similarities with previously exemplified hierarchies?

The first answer will be derived from the Kolmogorov-Chaitin algorithmic complexity: the complexity of a circuit is related with the dimension of its shortest formal description. A big circuit (a circuit built using a big number of gates) can be simple or complex depending on the possibility to emphasize repetitive patterns in its structure. A no pattern circuit is a complex one because its description has the dimension proportional with its size. Indeed, for a complex, no pattern circuit each gate must be explicitly specified.

The second answer associate the composition with sizing and the loop with featuring. Composing circuits results biggest structures with the same kind of functionality, while closing loops in a circuit new kind of behaviors are induced. Each new loop adds more autonomy to the system, because increases the dependency of the output signals in the detriment of the input signals. Shortly, appropriate loops means more autonomy that is equivalent sometimes with a new level of functionality.

The third answer is given by proposing a taxonomy for digital systems based on the maximum number of included loops closed in a certain digital system. The old distinction between combinational and sequential, applied only to circuits, is complemented with a classification taking into the account the functional and structural diversity of the digital systems used in the contemporary designs. More, the resulting classification provides classes of circuits having direct correspondence with the levels belonging to Kleene’s and Chomsky’s hierarchies.

The first part of the book – Digital Systems: a Bird’s-Eye View – is a general introduction in digital systems framing the digital domain in the larger context of the computational sciences, introducing the main formal tool for describing, simulating and synthesizing digital systems, and presenting the main mechanisms used to structure digital systems. The second part of the book – Looping in Digital Systems – deals with the main effects of the loop: more autonomy and segregation between the simple parts and the complex parts in digital systems. Both, autonomy and segregation, are used to minimize size and complexity. The third part of the book – Loop Based Morphisms – contains three attempts to make meaningful connections between the domain of the digital systems, and the fields of recursive functions, of formal languages and of information theories. The last chapter sums up the main ideas of the book making also some new correlations permitted by its own final position. The book ends with a lot of annexes containing short reviews of the prerequisite knowledge (binary arithmetic, Boolean functions, elementary digital circuits, automata theory), compact presentations of the formal tools used (pseudocode language, Verilog HDL), examples, useful data about real products (standard cell libraries).

PART I: Digital Systems: a Bird’s-Eye View

The first chapter: What’s a Digital System? Few general questions are answered in this chapter. One refers to the position of digital system domain in the larger class of the sciences of computation. Another
asks for presenting the ways we have to implement actual digital systems. The importance is also to present the correlated techniques allowing to finalize a digital product.

The second chapter: *Let’s Talk Digital Circuits in Verilog*  The first step in approaching the digital domain is to become familiar with a Hardware Description Language (HDL) as the main tool for mastering digital circuits and systems. The Verilog HDL is introduced and in the same time used to present simple digital circuits. The distinction between behavioral descriptions and structural descriptions is made when Verilog is used to describe and simulate combinational and sequential circuits. The temporal behaviors are described, along with solutions to control them.

The third chapter: *Scaling & Speeding & Featuring*  The architecture and the organization of a digital system are complex objectives. We can not be successful in designing big performance machine without strong tools helping us to design the architecture and the high level organization of a desired complex system. These mechanisms are three. One helps us to increase the brute force performance of the system. It is composition. The second is used to compensate the slow-down of the system due to excessive serial composition. It is pipelining. The last is used to add new features when they are asked by the application. It is about closing loops inside the system in order to improve the autonomous behaviors.

The fourth chapter: *The Taxonomy of Digital Systems*  A loop based taxonomy for digital systems is proposed. It classifies digital systems in orders, as follows:

- **0-OS**: zero-order systems - no-loop circuits - containing the combinational circuits;

- **1-OS**: 1-order systems - one-loop circuits - the memory circuits, with the autonomy of the internal state; they are used mainly for storing

- **2-OS**: 2-order systems - two-loop circuits - the automata, with the behavioral autonomy in their own state space, performing mainly the function of sequencing

- **3-OS**: 3-order systems - three-loop circuits - the processors, with the autonomy in interpreting their own internal states; they perform the function of controlling

- **4-OS**: 4-order systems - four-loop circuits - the computers, which interpret autonomously the programs according to the internal data

- ...  

- **n-OS**: n-order systems - n-loop circuits - systems in which the information is interpenetrated with the physical structures involved in processing it; the distinction between data and programs is surpassed and the main novelty is the self-organizing behavior.

The fifth chapter: *Our Final Target*  A small and simple programmable machine, called toyMachine is defined using a behavioral description. In the last chapter of the second part a structural design of this machine will be provided using the main digital structure introduced meantime.

PART II: Looping in Digital Domain
The sixth chapter: Gates The combinational circuits (0-OS) are introduced using a functional approach. We start with the simplest functions and, using different compositions, the basic simple functional modules are introduced. The distinction between simple and complex combinational circuits is emphasized, presenting specific technics to deal with complexity.

The seventh chapter: Memories There are two ways to close a loop over the simplest functional combinational circuit: the one-input decoder. One of them offers the stable structure on which we ground the class of memory circuits (1-OS) containing: the elementary latches, the master-slave structures (the serial composition), the random access memory (the parallel composition) and the register (the serial-parallel composition). Few applications of storing circuits (pipeline connection, register file, content addressable memory, associative memory) are described.

The eight chapter: Automata Automata (2-OS) are presented in the fourth chapter. Due to the second loop the circuit is able to evolve, more or less, autonomously in its own state space. This chapter begins presenting the simplest automata: the T flip-flop and the JK flip-flop. Continues with composed configurations of these simple structures: counters and related structures. Further, our approach makes distinction between the big sized, but simple functional automata (with the loop closed through a simple, recursive defined combinational circuit that can have any size) and the random, complex finite automata (with the loop closed through a random combinational circuit having the size in the same order with the size of its definition). The autonomy offered by the second loop is mainly used to generate or to recognize specific sequences of binary configurations.

The ninth chapter: Processors The circuits having three loops (3-OS) are introduced. The third loop may be closed in three ways: through a 0-OS, through an 1-OS or through a 2-OS, each of them being meaningful in digital design. The first, because of the segregation process involved in designing automata using JK flip-flops or counters as state register. The size of the random combinational circuits that compute the state transition function is reduced, in the most of case, due to the increased autonomy of the device playing the role of the register. The second type of loop, through a memory circuit, is also useful because it increases the autonomy of the circuit so that the control exerted on it may be reduced (the circuit “knows more about itself”). The third type of loop, that interconnects two automata (an functional automaton and a control finite automaton), generates the most important digital circuits: the processor.

The tenth chapter: Computers The effects of the fourth loop are shortly enumerated in the sixth chapter. The computer is the typical structure in 4-OS. It is also the support of the strongest segregation between the simple physical structure of the machine and the complex structure of the program (a symbolic structure). Starting from the fourth order the main functional up-dates are made structuring the symbolic structures instead of restructuring circuits. Few new loops are added in actual designs only for improving time or size performances, but not for adding new basic functional capabilities. For this reason our systematic investigation concerning the loop induced hierarchy stops with the fourth loop. The toyMachine behavioral description is revisited and substituted with a pure structural description.

The eleventh chapter: Self-Organizing Structures ends the first part of the book with some special circuits which belongs to n-OSs. The cellular automata, the connex memory and the eco-chip are n-loop structures that destroy the usual architectural thinking based on the distinction between the physical
support for symbolic structures and the circuits used for processing them. Each bit/byte has its own processing element in a system which performs the finest grained parallelism.

**The twelfth chapter: Global-Loop Systems**  Why not a hierarchy of hierarchies of loops? Having an $n$-order system how new features can be added? A possible answer: adding a global loop. Thus, a new hierarchy of super-loops starts. It is not about science fiction. ConnexArray™ is an example. It is described, evaluated and some possible applications are presented.

The main stream of this book deals with the *simple* and the *complex* in digital systems, emphasizing them in the *segregation* process that opposes simple structures of circuits to the complex structures of symbols. The *functional information* offers the environment for segregating the simple circuits from the complex binary configurations.

When the simple is mixed up with the complex, the *apparent complexity* of the system increases over its *actual complexity*. We promote design methods which reduce the apparent complexity by segregating the simple from the complex. The best way to substitute the apparent complexity with the actual complexity is to drain out the chaos from order. One of the most important conclusions of this book is that the main role of the *loop* in digital systems is to segregate the *simple* from the *complex*, thus emphasizing and using the hidden resources of *autonomy*.

In the *digital systems domain* prevails the *art of disclosing the simplicity* because there exists the symbolic domain of functional information in which we may ostracize the complexity. But, the complexity of the process of disclosing the simplicity exhausts huge resources of imagination. This book offers only the starting point for the *architectural thinking*: the art of finding the right place of the interface between simple and complex in computing systems.

**Acknowledgments**
Contents

PART I: Digital Systems: a Bird’s-Eye View ........................................... 4
The first chapter: What’s a Digital System? ........................................ 4
The second chapter: Let’s Talk Digital Circuits in Verilog ................... 5
The third chapter: Scaling & Speeding & Featuring .............................. 5
The fourth chapter: The Taxonomy of Digital Systems ......................... 5
The fifth chapter: Our Final Target .................................................. 5
PART II: Looping in Digital Domain .................................................. 5
The sixth chapter: Gates .................................................................... 6
The seventh chapter: Memories ....................................................... 6
The eighth chapter: Automata .......................................................... 6
The ninth chapter: Processors .......................................................... 6
The tenth chapter: Computers .......................................................... 6
The eleventh chapter: Self-Organizing Structures ............................... 6
The twelfth chapter: Global-Loop Systems ........................................ 7

1 A BIRD’S-EYE VIEW ON DIGITAL SYSTEMS ...................................... 1

1 WHAT’S A DIGITAL SYSTEM? ........................................................ 3
  1.1 Framing the digital design domain ............................................ 4
    1.1.1 Digital domain as part of electronics .............................. 4
    1.1.2 Modules in Verilog vs. Classes in Object Oriented Languages 12
    1.1.3 Digital domain as part of computer science ..................... 15
  1.2 Defining a digital system ....................................................... 18
  1.3 Our first target ................................................................. 24
  1.4 Problems ............................................................................ 29

2 DIGITAL CIRCUITS ........................................................................ 31
  2.1 Combinational circuits ......................................................... 32
    2.1.1 Zero circuit ............................................................... 33
    2.1.2 Selection ................................................................. 34
    2.1.3 Adder .................................................................. 36
    2.1.4 Divider ................................................................. 38
  2.2 Sequential circuits ............................................................... 38
## 2.2.1 Elementary Latches
- The reset-only latch .......................... 39
- The set-only latch ............................ 39
- The heterogenous set-reset latch ........... 39
- The symmetric set-reset latches .......... 39
- The first latch problem .................... 41
- The second latch problem ................. 41
- Application: de-bouncing circuit .......... 41

## 2.2.2 Elementary Clocked Latches .................. 42

## 2.2.3 Data Latch ............................... 43

## 2.2.4 Master-Slave Principle .................. 46

## 2.2.5 D Flip-Flop .............................. 48

## 2.2.6 Register ................................. 49
- Storing ......................................... 50
- Buffering ....................................... 50
- Synchronizing .................................. 50
- Delaying ....................................... 51
- Looping ....................................... 51

## 2.2.7 Shift register ............................ 51

## 2.2.8 Counter ................................. 52

## 2.3 Putting all together .......................... 53

## 2.4 Concluding about this short introduction in digital circuits .......... 54
- A digital circuit is build of combinational circuits and storage registers 54
- Combinational logic can do both, control and arithmetic .................. 54
- Logic circuits, with appropriate loops, can memorize ..................... 54
- HDL, as Verilog or VHDL, must be used to describe digital circuits ...... 54
- Growing, speeding and featuring digital circuits digital systems are obtained 54

## 2.5 Problems .................................... 54

## 3 GROWING & SPEEDING & FEATURING 59

### 3.1 Size vs. Complexity ........................ 62

### 3.2 Time restrictions in digital systems .... 64
- Pipelined connections ..................... 67
- Fully buffered connections ............... 70

### 3.3 Growing the size by composition ........ 71

### 3.4 Speeding by pipelining .................. 77
- Register transfer level .................... 77
- Pipeline structures ......................... 78
- Data parallelism vs. time parallelism .... 80

### 3.5 Featuring by closing new loops ........ 82
- Data dependency ............................. 84
- Speculating to avoid limitations imposed by data dependency ....... 87

### 3.6 Concluding about composing & pipelining & looping .......... 88

### 3.7 Problems .................................. 90

### 3.8 Projects .................................. 92
## 4 THE TAXONOMY OF DIGITAL SYSTEMS

4.1 Loops & Autonomy ................................. 94
4.2 Classifying Digital Systems ......................... 97
4.3 # Digital Super-Systems ........................... 100
4.4 Preliminary Remarks On Digital Systems .......... 100

- Combinational circuits vs. sequential circuits
- Composing circuits & closing loops
- Composition allows data parallelism and time parallelism
- Closing loops disturbs time parallelism
- Speculation can restore time parallelism
- Closed loops increase system autonomy
- Closing loops induces a functional hierarchy in digital systems

4.5 Problems ........................................... 102
4.6 Projects ............................................ 102

## 5 OUR FINAL TARGET

5.1 *toyMachine*: a small & simple computing machine .......................... 104
5.2 How *toyMachine* works ................................ 113

- 5.2.1 The Code Generator ................................. 114
- 5.2.2 The Simulation Module ............................. 118
- 5.2.3 Programming *toyMachine* ........................ 120

- The pseudo-macro *input*: ............................. 121
- The pseudo-macro *output*: ............................. 121
- The pseudo-macro *compute*: ........................... 121

5.3 Concluding about *toyMachine* ......................... 125
- Our final target ........................................ 125
- The behavioral description of *toyMachine* is syntheisable ........ 125
- Pros & cons for *programmed logic* ..................... 125

5.4 Problems ........................................... 125
5.5 Projects ............................................ 125

## II LOOPING IN THE DIGITAL DOMAIN

### 6 GATES:

**Zero order, no-loop digital systems** ........................................ 129

6.1 Simple, Recursive Defined Circuits ....................... 130

- 6.1.1 Decoders ........................................ 131

- Informal definition ..................................... 131
- Formal definition ...................................... 131
- Recursive definition ................................... 132
- Non-recursive description ............................... 133
- Arithmetic interpretation .............................. 134
7.2 The Serial Composition: the Edge Triggered Flip-Flop ........................................... 192
  7.2.1 The Serial Register ......................................................................................... 193
7.3 The Parallel Composition: the Random Access Memory ........................................ 194
  7.3.1 The $n$-Bit Latch ......................................................................................... 194
  7.3.2 Asynchronous Random Access Memory ......................................................... 195
      Expanding the number of bits per word ............................................................. 197
      Expanding the number of words by two dimension addressing ......................... 197
7.4 Applications ........................................................................................................... 200
  7.4.1 Synchronous RAM ......................................................................................... 200
  7.4.2 Register File ................................................................................................... 201
  7.4.3 Field Programmable Gate Array – FPGA .......................................................... 202
      The system level organization of an FPGA ...................................................... 203
      The IO interface ................................................................................................ 204
      The switch node ............................................................................................... 204
      The basic building block .................................................................................. 205
      The configurable logic block .......................................................................... 206
  7.4.4 * Content Addressable Memory ...................................................................... 206
  7.4.5 * An Associative Memory ............................................................................... 208
  7.4.6 * Beneš-Waxman Permutation Network ......................................................... 210
  7.4.7 * First-Order Systolic Systems ...................................................................... 214
7.5 Concluding About Memory Circuits ...................................................................... 216
      The first closed loop in digital circuits latches events ..................................... 217
      Meaningful circuits occur by composing latches ............................................. 217
      Distinguishing between “how?” and “when?” ............................................... 217
      Registers and RAMs are basic structures ....................................................... 217
      RAM is not a memory, it is only a physical support ....................................... 217
      Memorizing means to associate .................................................................... 217
      To solve ambiguities a new loop is needed .................................................... 217
7.6 Problems ............................................................................................................... 218
7.7 Projects .................................................................................................................. 223
8 AUTOMATA: Second order, 2-loop digital systems .................................................. 225
  8.1 Basic definitions in automata theory ................................................................... 227
8.2 Two States Automata ............................................................................................. 230
  8.2.1 Optimizing DFF with an asynchronous automaton ....................................... 230
  8.2.2 The Smallest Automaton: the T Flip-Flop .................................................... 232
  8.2.3 The JK Automaton: the Greatest Flip-Flop .................................................. 233
  8.2.4 * Serial Arithmetic ....................................................................................... 233
  8.2.5 * Hillis Cell: the Universal 2-Input, 1-Output and 2-State Automaton .......... 234
8.3 Functional Automata: the Simple Automata .......................................................... 235
  8.3.1 Counters ........................................................................................................ 235
  8.3.2 Linear Feedback Shift Registers .................................................................... 237
  8.3.3 RALU: Registers with Arithmetic-Logic Unit ............................................. 240
      Structured State Space Automaton($S^3$A) ...................................................... 240
Multi-port S$^3$A ....................................................... 241
8.3.4  * Accumulator Automaton ........................................ 243
8.3.5  * Sequential multiplication ..................................... 243
     * Radix-2 multiplication ....................................... 243
     * Radix-4 multiplication ....................................... 245
8.3.6  * “Bit-eater” automaton ........................................ 249
8.3.7  * Sequential divider ........................................... 250
8.4  * Composing with simple automata ................................ 251
     8.4.1  * LIFO memory ............................................. 251
     8.4.2  * FIFO memory ............................................. 253
     8.4.3  * The Multiply-Accumulate Circuit ......................... 255
     8.4.4  * Weighted Pseudo-Random Generator ....................... 257
8.5  Finite Automata: the Complex Automata ........................... 258
     8.5.1  Representing finite automata ................................ 258
           Flow-charts ............................................... 258
           The flow-chart for a half-automaton ....................... 258
           The flow-chart for a Moore automaton ..................... 259
           The flow-chart for a Mealy automaton ..................... 260
           Transition diagrams ....................................... 261
           Transition diagrams for half-automata ..................... 261
           Transition diagrams Moore automata ....................... 263
           Transition diagrams Mealy automata ....................... 264
           Procedures ............................................... 265
           HDL representations for Moore automata .................... 265
           HDL representations for Mealy automata .................... 267
     8.5.2  Designing Finite Automata .................................... 268
     Preliminary Examples ............................................ 268
     State Coding .................................................... 283
     Minimal variation encoding ..................................... 284
     Reduced dependency encoding ................................... 284
     Incremental coding ............................................. 286
     One-hot state encoding ........................................ 286
     Minimizing finite automata ...................................... 287
     Minimizing the size by an appropriate state coding .......... 287
     Minimizing the complexity by one-hot encoding ............... 289
     Version 1: with ”one-hot” encoding ............................. 290
     Version 2: compact binary coding ................................ 290
     Asynchronous inputs .............................................. 290
     The case of one asynchronous input ............................. 291
     The case of more than one asynchronous inputs ............... 292
     The Hazard ....................................................... 294
     Hazard generated by asynchronous inputs ...................... 294
     Propagation hazard .............................................. 296
     Dynamic hazard ................................................ 298
     Fundamental limits in implementing automata ................. 298
     8.5.3  * Control Automata: the First “Turning Point” .............. 299
CONTENTS

Very important comment! ........................................... 301
Verilog descriptions for CROM ................................. 304
Binary code generator ........................................... 306
8.6 * Automata vs. Combinational Circuits ................. 308
8.7 * The Circuit Complexity of a Binary String .......... 311
8.8 Concluding about automata ................................. 313
    Synchronous automata need non-transparent state registers 313
    The second loop means the behavior’s autonomy .......... 314
    Simple automata can have $n$ states ..................... 314
    Complex automata have only finite number of states .... 314
    Control automata suggest the third loop ................ 314
8.9 Problems .................................................... 314
8.10 Projects ................................................... 320

9 PROCESSORS:
Third order, 3-loop digital systems .......................... 323
    9.1 Implementing finite automata with “intelligent registers” 325
    9.1.1 Automata with JK “registers” ......................... 325
    9.1.2 * Automata using counters as registers ............ 328
    9.2 Loops closed through memories .......................... 330
        Version 1: the controlled Arithmetic & Logic Automaton 331
        Version 2: the commanded Arithmetic & Logic Automaton 331
    9.3 Loop coupled automata: the second “turning point” .... 333
        9.3.1 Counter extended automata (CEA) ................. 333
        9.3.2 The elementary processor ......................... 333
        9.3.3 Executing instructions vs. interpreting instructions 337
            Von Neumann architecture / Harvard architecture .... 339
        9.3.4 An executing processor ......................... 340
            The organization .................................. 340
            RALU ............................................ 340
            The instruction set architecture .................. 341
            Implementing toyRISC ................................ 344
            The time performance ............................. 344
        9.3.5 * An interpreting processor ....................... 347
            The organization ................................. 347
            Microarchitecture ............................... 352
            Instruction set architecture (ISA) ................ 352
            Implementing ISA ................................ 353
            Time performance ................................ 359
            Concluding about our CISC processor ............... 360
    9.4 * The assembly language: the lowest programming level 360
    9.5 Concluding about the third loop ....................... 360
        The third loop is closed through simple automata .... 360
        “Intelligent registers” ask less structural control .... 360
        The loop through a storage element ask less symbolic control 360
Looping through a memory circuit allows a more complex “understanding”.
Looping through an automaton allows any effective computation.
The third loop allows the symbolic functional control.
Real processors use circuit level parallelism.

9.6 Problems ................................................................. 361
9.7 Projects ................................................................. 361

10 COMPUTING MACHINES:

4-loop digital systems .......................... 367
10.1 Types of fourth order systems ............... 368
  10.1.1 Counter extended CEA ....................... 370
  10.1.2 Push-down automata ......................... 370
  10.1.3 The computer – support for the strongest segregation ........... 372
10.2 Stack processor – a processor as 4-OS ....... 373
  10.2.1 The organization ...................... 373
  10.2.2 The micro-architecture ..................... 376
  10.2.3 The instruction set architecture ............. 379
  10.2.4 Implementation: from micro-architecture to architecture ....... 380
    Instruction nop .................................. 380
    Instruction add .................................. 380
    Instruction load .................................. 380
    Instruction inc .................................. 382
    Instruction store .................................. 382
    Instruction push .................................. 382
    Instruction dup .................................. 382
    Instruction over .................................. 382
    Instruction zero .................................. 383
    Instruction jmp .................................. 383
    Instruction call .................................. 383
    Instruction jmpz .................................. 384
    Instruction ret .................................. 384
  10.2.5 Time performances ......................... 384
  10.2.6 Concluding about our Stack Processor .... 385
10.3 Embedded computation ....................... 385
  10.3.1 The structural description of toyMachine ...... 385
    The top module .................................. 386
    The interrupt .................................. 387
    The control section .......................... 387
    The data section .............................. 391
    Multiplexors .................................. 393
    Concluding about toyMachine .............. 393
  10.3.2 Interrupt automaton: the asynchronous version ...... 399
10.4 Problems ................................................................. 402
11 # * SELF-ORGANIZING STRUCTURES:  
N-th order digital systems 405  
11.1 N-counter extended finite automata 406  
11.2 Push-Down Stack as n-OS 406  
11.3 Cellular automata 407  
11.3.1 General definitions 407  
The linear cellular automaton 407  
The two-dimension cellular automaton 412  
11.3.2 Applications 414  
11.4 Systolic systems 415  
11.5 Interconnection issues 418  
11.5.1 Local vs. global connections 418  
Memory wall 418  
11.5.2 Localizing with cellular automata 418  
11.5.3 Many clock domains & asynchronous connections 418  
11.6 Neural networks 418  
11.6.1 The neuron 418  
11.6.2 The feedforward neural network 419  
11.6.3 The feedback neural network 421  
11.6.4 The learning process 423  
Unsupervised learning: Hebbian rule 424  
Supervised learning: perceptron rule 425  
11.6.5 Neural processing 426  
11.7 Problems 426  

12 # * GLOBAL-LOOP SYSTEMS 427  
12.1 Global loops in cellular automata 428  
12.2 The Generic ConnexArray\textsuperscript{TM}: the first global loop 429  
12.2.1 The Generic ConnexArray\textsuperscript{TM} 429  
The Instruction Set Architecture 432  
The operand code 432  
The conditions 432  
The opCode code 433  
Program Control Section 433  
The Execution in Controller and in Cells 434  
Operand selection in controller 434  
Operand selection in the array’s cells 434  
Data operations in controller 434  
Data operations in the array’s cells 435  
Spatial selection 436  
Vector transfer 436  
12.2.2 Assembler Programming the Generic ConnexArray\textsuperscript{TM} 436  
12.3 Search Oriented ConnexArray\textsuperscript{TM}: the Second Global Loop 438  
12.3.1 List processing 439
12.3.2 Sparse matrix operations ........................................ 439
12.4 High Level Architecture for ConnexArray™ ....................... 439
  12.4.1 System initialization ........................................... 440
    ResetSystem ..................................................... 440
    pp .............................................................. 440
    rs .............................................................. 440
    test ........................................................... 441
    Index .......................................................... 441
    MakeAll ......................................................... 441
  12.4.2 Addressing functions .......................................... 442
    Vec ............................................................. 442
    Mem ............................................................. 442
  12.4.3 Setting functions .............................................. 442
    Copy ............................................................ 442
    SetVector ....................................................... 442
    SetAll ........................................................ 442
    SetStream ...................................................... 443
  12.4.4 Transfer functions ............................................ 443
    Basic transfer functions ....................................... 443
      StoreVector .................................................. 443
      LoadVector ................................................... 443
      StoreVectorStrided ......................................... 443
      LoadVectorStrided .......................................... 444
  12.4.5 Arithmetic functions ......................................... 444
    Abs .............................................................. 444
    Inc .............................................................. 445
    Dec .............................................................. 445
    Add .............................................................. 445
    Sub .............................................................. 445
    Mult ............................................................ 445
    Div .............................................................. 446
    Quotient ........................................................ 446
    Rem .............................................................. 446
    Not ............................................................... 446
    And .............................................................. 446
    Or ................................................................. 446
    Xor .............................................................. 446
  12.4.6 Reduction functions ............................................ 446
    RedAdd .......................................................... 447
    RedAddAll ...................................................... 447
    RedMax ........................................................ 447
    RedMaxAll ...................................................... 447
    Select .......................................................... 447
    SelectFromAll ................................................. 447
    AnyActive ....................................................... 447
  12.4.7 Test functions ................................................. 448
CONTENTS

19

Lt ................................................................. 448
Gt ................................................................. 448
Leq .............................................................. 448
Geq .............................................................. 448
Eq ............................................................... 449
Even? .............................................................. 449

12.4.8 Control functions ........................................ 449
ResetActive ..................................................... 449
SetActive ....................................................... 449
SaveActive ....................................................... 449
Where ............................................................. 449
EndWhere ........................................................ 449
ElseWhere ....................................................... 449
First .............................................................. 449
FirstIndex ....................................................... 449

12.4.9 Stream functions ........................................ 450
Permute .......................................................... 450
ShiftLeft ........................................................ 450
ShiftLeftEl ...................................................... 450
ShiftRight ...................................................... 450
ShiftRightEl .................................................... 450
RotateLeft ...................................................... 450
RotateRight .................................................... 451
Search .......................................................... 451
SearchCond ..................................................... 451
InsertPos ....................................................... 451
DeletePos ....................................................... 451

12.5 Index ........................................................ 452

12.6 Problems .................................................... 454

III ANNEXES ...................................................... 455

A Boolean functions ........................................... 457

A.1 Short History ............................................... 457
Aristotle of Stagira ............................................. 457
George Boole ................................................... 457
Claude Elwood Shannon ..................................... 457

A.2 Elementary circuits: gates .................................. 457
A.2.1 Zero-input logic circuits ................................ 458
A.2.2 One input logic circuits ................................ 458
A.2.3 Two inputs logic circuits ............................... 458
A.2.4 Many input logic circuits ............................... 459

A.3 How to Deal with Logic Functions ....................... 459
Identity principle .............................................. 459
Double negation principle ................................... 459
CONTENTS

Associativity .............................................. 460
Commutativity ............................................. 460
Distributivity ............................................. 460
Absorption .................................................. 460
Half-absorption .......................................... 460
Substitution ................................................ 461
Exclusion ................................................... 461
De Morgan laws ............................................. 461

A.4 Minimizing Boolean functions ....................... 461
A.4.1 Canonical forms .................................... 461
A.4.2 Algebraic minimization ............................... 462
Minimal depth minimization ......................... 462
Multi-level minimization .............................. 463
Many output circuit minimization ................. 464
A.4.3 Veitch-Karnaugh diagrams ......................... 465
Minimizing with V-K diagrams .................... 466
Minimizing incomplete defined functions ........ 467
V-K diagrams with included functions ........... 468

A.5 Problems ............................................... 469

B Basic circuits ........................................... 471
B.1 Actual digital signals ................................ 471
B.2 CMOS switches ...................................... 473
B.3 The Inverter .......................................... 474
B.3.1 The static behavior ................................. 474
B.3.2 Dynamic behavior ................................. 475
B.3.3 Buffering ........................................... 477
B.3.4 Power dissipation ................................. 478
Switching power ........................................ 478
Short-circuit power ..................................... 479
Leakage power ........................................... 480

B.4 Gates .................................................. 480
B.4.1 NAND & NOR gates ................................. 480
The static behavior of gates ....................... 480
Propagation time ........................................ 481
Propagation time for NAND gate .................. 481
Propagation time for NOR gate .................... 481
Power consumption & switching activity ......... 482
Switching activity for 2-input AND ............... 482
Switching activity for 3-input AND ............... 483
Switching activity for n-input AND ............... 483
Power consumption & glitching .................... 483

B.4.2 Many-Input Gates ................................ 484
B.4.3 AND-NOR gates ................................ 485

B.5 The Tristate Buffers ................................ 486

B.6 The Transmission Gate .............................. 486
B.7 Memory Circuits ............................................. 488
  B.7.1 Flip-flops ............................................. 488
    Data latches and their transparency ..................... 488
    Master-slave DF-F ..................................... 488
    Resetable DF-F ......................................... 488
  B.7.2 # Static memory cell ................................. 488
  B.7.3 # Array of cells .................................... 488
  B.7.4 # Dynamic memory cell .............................. 488
B.8 Problems .................................................. 488

C Standard cell libraries ...................................... 491
  C.1 Main parameters ........................................ 491
  C.2 Basic cells ............................................. 492
    C.2.1 Gates ............................................... 492
      Two input AND gate: AND2 ............................... 492
      Implementation ....................................... 492
      Two input OR gate: OR2 ................................ 493
      Implementation ....................................... 493
      Three input AND gate: AND3 ............................. 493
      Implementation ....................................... 493
      Three input OR gate: OR3 ............................... 493
      Implementation ....................................... 493
      Four input AND gate: AND4 ............................. 494
      Implementation ....................................... 494
      Four input OR gate: OR4 ............................... 494
      Implementation ....................................... 494
      Four input AND-OR gate: ANDOR22 ...................... 494
      Implementation ....................................... 494
      Invertor: NOT .......................................... 494
      Implementation ....................................... 494
      Two input NAND gate: NAND2 ............................. 494
      Implementation ....................................... 494
      Two input NOR gate: NOR2 ............................... 494
      Implementation ....................................... 494
      Three input NAND gate: NAND3 ........................... 495
      Implementation ....................................... 495
      Three input NOR gate: NOR3 ............................. 495
      Implementation ....................................... 495
      Four input NAND gate: NAND4 ............................ 495
      Implementation ....................................... 495
      Four input NOR gate: NOR4 ............................... 495
      Implementation ....................................... 495
      Two input multiplexer: MUX2 ............................ 495
      Implementation ....................................... 495
      Two input inverting multiplexer: NMUX2 ................. 495
      Implementation ....................................... 495
Part I

A BIRD’S-EYE VIEW ON DIGITAL SYSTEMS
Chapter 1

WHAT’S A DIGITAL SYSTEM?

In the previous chapter
we can not find anything because it does not exist, but we suppose the reader is familiar with:

- fundamentals about what means computation
- basics about Boolean algebra and basic digital circuits (see Annexes Boolean Functions and Basic circuits for a short refresh)
- the usual functions supposed to be implemented by digital sub-systems in the current audio, video, communication, gaming, ... market products

In this chapter
general definitions related with the digital domain are used to reach the following targets:

- to frame the digital system domain in the larger area of the information technologies
- to present different ways the digital approach is involved in the design of the real market products
- to enlist and shortly present the related domains, in order to integrate better the knowledge and skills acquired by studying the digital system design domain

In the next chapter
is a friendly introduction in both, digital systems and a HDLs (Hardware Description Languages) used to describe, simulate, and synthesized them. The HDL selected for this book is called Verilog. The main topics are:

- the distinction between combinational and sequential circuits
- the two ways to describe a circuit: behavioral or structural
- how digital circuits behave in time.
Talking about Apple, Steve said, “The system is there is no system.” Then he added, “that doesn’t mean we don’t have a process.” Making the distinction between process and system allows for a certain amount of fluidity, spontaneity, and risk, while in the same time it acknowledges the importance of defined roles and discipline.

J. Young & W. Simon

A process is a strange mixture of rationally established rules, of imaginatively driven chaos, and of integrative mystery.

A possible good start in teaching about a complex domain is an informal one. The main problems are introduced friendly, using an easy approach. Then, little by little, a more rigorous style will be able to consolidate the knowledge and to offer formally grounded techniques. The digital domain will be disclosed here alternating informal “bird’s-eye views” with simple, formalized real stuff. Rather than imperatively presenting the digital domain we intend to disclose it in small steps using a project oriented approach.

1.1 Framing the digital design domain

Digital domain can be defined starting from two different, but complementary view points: the structural view point or the functional view point. The first version presents the digital domain as part of electronics, while the second version sees the digital domain as part of computer science.

1.1.1 Digital domain as part of electronics

Electronics started as a technical domain involved in processing continuously variable signals. Now the domain of electronics is divided in two sub-domains: analogue electronics, dealing with continuously variable signals and digital electronics based on elementary signals, called bits, which take only two different levels 0 and 1, but can be used to compose any complex signals. Indeed, a sequence of $n$ bits is used to represent any number between 0 and $2^n - 1$, while a sequence of numbers can be used to approximate a continuously variable signal.

Let be, for example, the analogue, continuously variable, signal in Figure 1.1. It can be approximated by values sampled in discrete moments of time determined by the positive transitions of a square wave periodic signal called clock. It switches with a frequency of $1/T$. The value of the signal is measured in units $u$ (for example, $u = 100mV$ or $u = 10\mu A$). The operation is called analog to digital conversion, and it is performed by an analog to digital converter – ADC. Results the following sequence of numbers:

\[
\begin{align*}
    s(0 \times T) &= 1 \text{ units} \Rightarrow 001, \\
    s(1 \times T) &= 4 \text{ units} \Rightarrow 100, \\
    s(2 \times T) &= 5 \text{ units} \Rightarrow 101,
\end{align*}
\]

1They co-authored iCon. Steve Jobs. The Greatest Second Act in the History of Business, an unauthorized portrait of the co-founder of Apple.
1.1. FRAMING THE DIGITAL DESIGN DOMAIN

\[
s(3 \times T) = 6 \text{units} \Rightarrow 110, \\
s(4 \times T) = 6 \text{units} \Rightarrow 110, \\
s(5 \times T) = 6 \text{units} \Rightarrow 110, \\
s(6 \times T) = 6 \text{units} \Rightarrow 110, \\
s(7 \times T) = 6 \text{units} \Rightarrow 110, \\
s(8 \times T) = 5 \text{units} \Rightarrow 101, \\
s(9 \times T) = 4 \text{units} \Rightarrow 100, \\
s(10 \times T) = 2 \text{units} \Rightarrow 010, \\
s(11 \times T) = 1 \text{unit} \Rightarrow 001, \\
s(12 \times T) = 1 \text{unit} \Rightarrow 001, \\
s(13 \times T) = 1 \text{unit} \Rightarrow 001, \\
s(14 \times T) = 1 \text{unit} \Rightarrow 001, \\
s(15 \times T) = 2 \text{units} \Rightarrow 010, \\
s(16 \times T) = 3 \text{units} \Rightarrow 011, \\
s(17 \times T) = 4 \text{units} \Rightarrow 100, \\
s(18 \times T) = 5 \text{units} \Rightarrow 101, \\
s(19 \times T) = 5 \text{units} \Rightarrow 101, \\
s(20 \times T) = 5 \text{units} \Rightarrow 101, \\
\ldots
\]

If a more accurate representation is requested, then both, the sampling period, \(T\) and the measure units \(u\) must be reduced. For example, in Figure 1.2 both, \(T\) and \(u\) are halved. A better approximation is obtained with the price of increasing the number of bits used for representation. Each sample is represented on 4 bits instead of 3, and the number of samples is doubled. This second, more accurate, conversion provides the following stream of binary data:

\(<0011, 0110, 1000, 1001, 1010, 1011, 1100, 1100, 1100, 1100, 1100, 1100, 1100, 1100, 1100, 1101, 1010, 1010, 1011, 0011, 0010, 0001, 0001, 0001, 0001, 0001, 0010, 0011, 0011, 0101, 0110, 0111, 1000, 1001, 1001, 1001, 1010, 1010, 1010, \ldots>\)

An ADC is characterized by two main parameters:

- the sampling rate: expressed in samples per second – SPS – or by the sampling frequency – \(1/T\)
- the resolution: the number of bits used to represent the value of a sample

Commercial ADC are provided with resolution in the range of 6 to 24 bits, and the sample rate exceeding 3 GSPS (giga SPS). At the highest sample rate the resolution is limited to 12 bits.

The generic digital electronic system is represented in Figure 1.3, where:

- \(\text{analogInput}_i\), for \(i = 1, \ldots M\), provided by various sensors (microphones, ...), are sent to the input of M ADCs
- \(\text{ADC}_i\) converts \(\text{analogInput}_j\) in a stream of binary coded numbers, using an appropriate sampling interval and an appropriate number of bits for approximating the level of the input signal
- DIGITAL SYSTEM processes the M input streams of data providing on its outputs N streams of data applied on the input of N Digital-to-Analog Converters (DAC)
CHAPTER 1. WHAT’S A DIGITAL SYSTEM?

Figure 1.1: Analogue to digital conversion. The analog signal, $s(t)$, is sampled at each $T$ using the unit measure $u$, and results the three-bit digital signal $S[2:0]$. **A first application:** the one-bit digital signal $\hat{W} = \{1 < s < 5\}$ indicates, by its active value 1, the time interval when the digital signal is strictly included between $1u$ and $5u$. The three-bit result of conversion is $S[2:0]$.
1.1. FRAMING THE DIGITAL DESIGN DOMAIN

Figure 1.2: More accurate analogue to digital. The analogous signal is sampled at each \( T/2 \) using the unit measure \( u/2 \).

- **DAC**\(_j\) converts its input binary stream to **analogOutput**\(_j\)

- **analogOutput**\(_j\), for \( j = 1, \ldots, N \), are the outputs of the electronic system used to drive various actuators (loudspeakers, ...)

- **clock** is the synchronizing signal applied to all the components of the system; it is used to trigger the moments when the signals are ready to be used and the subsystems are ready to use the signals.

While loosing something at conversion, we are able to gain at the level of processing. The good news is that the loosing process is under control, because both, the accuracy of conversion and of digital processing are highly controllable.

In this stage we are able to understand that the internal structure of DIGITAL SYSTEM from Figure 1.3 must have the possibility to do deal with **binary signals** which must be **stored & processed**. The signals are stored synchronized with the active edge of the **clock** signal, while for processing are used circuits dealing with two distinct values: 0 and 1. Usually, the value 0 is represented by the low voltage, currently 0, while the value 1 by high voltage, currently \( \sim 1V \). Consequently, two distinct kinds of circuits can be emphasized in this stage:

- **registers**: used to register, synchronously with the active edge of the clock signal, the **n**-bit binary configuration applied on its inputs

- **logic circuits**: used to implement a correspondence between all the possible combinations of 0s and 1s applied on its **m**-bit input and the binary configurations generated on its **n**-bit output.

**Example 1.1** Let us consider a system with one analog input digitized with a low accuracy converter which provides only three bits (like in the example presented in Figure 1.1). The one-bit output, \( v \), of the Boolean (logic) circuit\(^2\) to be designed, let’s call it \( \text{window} \), must be active (on 1) each time when the result of conversion is less than 5 and greater than 1. In Figure 1.1 the wave form represents the signal \( v \) for the particular signal represented in the first wave form. The transfer function of the circuit is represented in the table from Figure 1.4a, where: for three binary input configurations, \( S[2:0] = \)

\(^2\)See details about Boolean logic in the appendix **Boolean Functions**.
\{C, B, A\} = 010 | 011 | 100, the output must take the value 1, while for the rest the output must be 0. Pseudo-formally, we write:

\[ W = 1 \text{ when } \left( (\text{not } C = 1) \text{ and } (B = 1) \text{ and } (\text{not } A = 1) \right) \text{ or } \left( (\text{not } C = 1) \text{ and } (B = 1) \text{ and } (A = 1) \right) \text{ or } \left( (C = 1) \text{ and } (\text{not } B = 1) \text{ and } (\text{not } A = 1) \right) \]

Using the Boolean logic notation:

\[ W = C' \cdot B \cdot A' + C' \cdot B \cdot A + C \cdot B' \cdot A' = C'B(A' + A) + CB'A' = C'B + CB'A' \]

The resulting logic circuit is represented in Figure 1.4b, where:

- three NOT circuits are used for generating the negated values of the three input variables: C, B, A
- one 2-input AND circuit computes C'B
- one 3-input AND circuit computes CB'A'
- one 2-input OR circuit computes the final OR between the previous two functions.

The circuit is simulated and synthesized using its description in the hardware description language (HDL) Verilog, as follows:

```verilog
module window( output W, input C, B, A);

wire w1, w2, w3, w4, w5; // wires for internal connections

not notc(w1, C), // the instance 'notc' of the generic 'not'
notb(w2, B), // the instance 'notb' of the generic 'not'
```
In Verilog, the entire circuit is considered a module, whose description starts with the keyword `module` and ends with the keyword `endmodule`, which contains:

- the declarations of two kinds of connections:
  - external connections associated to the name of the module as a list containing:
    * the output connections (only one, W, in our example)
    * the input connections (C, B and A)
  - internal connections declared as wire, w1, w2, ... w5, used to interconnect the output of the internal circuits to the input of the internal circuits

- the instantiation of previously defined modules; in our example these are generic logic circuits expressed by keywords of the language, as follows:
  - circuits not, instantiated as nota, notb, notc; the first connection in the list of connections is the output, while the second is the input
  - circuits and, instantiated as and1, and2; the first connection in the list of connections is the output, while the next are the inputs
  - circuit or, instantiated as outOr; the first connection in the list of connections is the output, while the next are the inputs

Figure 1.4: The circuit window. a. The truth table represents the behavior of the output for all binary configurations on the input. b. The circuit implementation.
The Verilog description is used for simulating and for synthesizing the circuit.

The simulation is done by instantiating the circuit window inside the simulation module simWindow:

```verilog
module simWindow;
    reg    A, B, C ;
    wire   W    ;

    initial begin
        {C, B, A} = 3'b000 ;
        #1 {C, B, A} = 3'b001 ;
        #1 {C, B, A} = 3'b010 ;
        #1 {C, B, A} = 3'b011 ;
        #1 {C, B, A} = 3'b100 ;
        #1 {C, B, A} = 3'b101 ;
        #1 {C, B, A} = 3'b110 ;
        #1 {C, B, A} = 3'b111 ;
        #1 $stop    ;
    end

    window dut( W, C, B, A);

    initial $monitor( "S=%b W=%b" ,
        {C, B, A}, W);
endmodule
```

Example 1.2 The problem to be solved is to measure the length of objects on a transportation band which moves with a constant speed. A photo-sensor is used to detect the object. It generates 1 during the displacement of the object in front of the sensor. The occurrence of the signal must start the process of measurement, while the end of the signal must stop the process. Therefore, at every ends of the signal a short impulse, of one clock cycle long, must be generated.

The problem is solved in the following steps:

1. the asynchronous signal pulse, generated by the sensor, is synchronized with the system clock; now the actual signal is approximated with a reasonable error by the signal syncPulse
2. the synchronized pulse is delayed one clock cycle and results delPulse
3. the relation between syncPulse and syncPulse is used to identify the beginning and the end of the pulse with an accuracy given by the frequency of the clock signal (the higher the frequency the higher the accuracy):
   - only in the first clock cycle after the beginning of syncPulse the signal delPulse is 0; then
     \[ \text{start} = \text{syncPulse} \cdot \text{depPulse} \]
   - only in the first clock cycle after the end of syncPulse the signal delPulse is 1; then
     \[ \text{stop} = \text{syncPulse}' \cdot \text{depPulse} \]
1.1. Framing the Digital Design Domain

Figure 1.5: The waveforms defining the start/stop circuit. The pulse signal is asynchronously provided by a sensor. The signal syncPulse captures synchronously the signal to be processed. The signal delPulse is syncPulse delayed one clock cycle using a second one-bit register.

The circuit (see Figure 1.6) used to perform the previous steps contains:

- the one-bit register R1 which synchronizes the one-bit digital signal pulse
- the one bit register R2 which delays with one clock cycle the synchronized signal
- the combinational circuit which computes the two-output logic function

The Verilog description of the circuit is:

```verilog
module ends(output start ,
            output stop ,
            input pulse ,
            input clock );

    reg syncPulse ;
    reg delPulse ;
```

CHAPTER 1. WHAT'S A DIGITAL SYSTEM?

12

Figure 1.6: The ends circuit. The one-bit register R1 synchronises the raw signal pulse. The one-bit register R2 delays the synchronized signal to provide the possibility to emphasize the two ends of the synchronized pulse. The combinatorial circuit detects the two ends of the pulse signal approximated by the syncPulse signal.

wire w1, w2;

always @(posedge clock) begin syncPulse <= pulse;
        delPulse <= syncPulse;
        not not1(w1, syncPulse);
        not not2(w2, delPulse);
        and startAnd(start, syncPulse, w2);
        and stopAnd(stop, w1, delPulse);
    endmodule

Besides wire and gates, we have to declare now registers and we must show how their content change with the active edge of clock.

1.1.2 Modules in Verilog vs. Classes in Object Oriented Languages

What kind of language is the Verilog HDL? We will show it is a sort of Object Oriented Language. Let us design in Verilog a four-input adder modulo $2^8$.

/************************************************
File: adder2.v
Describes: two-input mod256 adder
************************************************/
module adder2( output [7:0] out,
        input [7:0] in0, in1);

        assign out = in0 + in1;
endmodule

/****************************************************************************
/****************************************************************************
File: adder4.v

Describes: four-input mod256 adder
****************************************************************************/
module adder4( output [7:0] out,
              input [7:0] in0, in1, in2,in3);
    wire [7:0] sum1, sum2;
    adder2 add1(sum1, in0, in1),
              add1(sum2, in2, in3),
              add1(out, sum1, sum2);
endmodule
//******************************************************************************
In C++ programming language the programm for adding four numbers can be write using, instead of two modules, two classes, as follow:

```cpp
class adder2{public:
  int in1, in2, out;
  // Constructor
  adder2(int a, int b){
    in1 = a;
    in2 = b;
    out = in1 + in2;
  }
  // Method
  void displayAdd2(){
    cout << in1 << in2 << out << endl;
  }
};
```

File: adder2.cpp

Describes:
- Constructor: describes a two-input integer adder
- Methods: displays the behavior of adder2 for test

/***************************************************************
File: adder2.cpp
Describes:
- Constructor: describes a two-input integer adder
- Methods: displays the behavior of adder2 for test
***************************************************************
class adder2{public:
  int in1, in2, out;
  // Constructor
  adder2(int a, int b){
    in1 = a;
    in2 = b;
    out = in1 + in2;
  }
  // Method
  void displayAdd2(){
    cout << in1 << in2 << out << endl;
  }
};
/***************************************************************
1.1. FRAMING THE DIGITAL DESIGN DOMAIN

/* File: adder4.cpp

Describes:
- Constructor: describes a four-input integer adder
  + uses three instances of adder2: S1, S2, S3
- Methods: displays the behavior of adder4 for test
*/

class adder4
{
  public:
    int in1, in2, in3, in4, out;
    // Constructor
    adder4(int a, int b, int c, int d)
    { in1 = a;
      in2 = b;
      in3 = c;
      in4 = d;
      adder2 S1(a, b);
      adder2 S2(c, d);
      adder2 S3(S1.out, S2.out);
      out = S3.out;
    }
    // Method
    void displayAdd4()
    { cout << in1 << in2 << in3 << in4 << out << endl; }
};

The class adder2 describe the two-input adder used to build, three times instantiated in class adder4, a four input adder.

A class is more complex than a module because it can contain, as a method, the way the class is tested. In Verilog we have to define a distinct module, testAdde2 or testAdder4, for simulation.

1.1.3 Digital domain as part of computer science

The domain of digital systems is considered, form the functional view point, as part of computing science. This, possible view point presents the digital systems as systems which compute their associated transfer functions. A digital system is seen as a sort of electronic system because of the technology used now to implement it. But, from a functional view point it is simply a computational system, because future technologies will impose maybe different physical ways to implement it (using, for example, different kinds of nano-technologies, bio-technologies, photon-based devices, ...). Therefore, we decided to start our approach using a functionally oriented introduction in digital systems, considered as a sub-domain of computing science. Technology dependent knowledge is always presented only as a supporting background for various design options.

Where can be framed the domain of digital systems in the larger context of computing science? A simple, informal definition of computing science offers the appropriate context for introducing digital systems.
Definition 1.1 Computer science (see also Figure 1.7) means to study:

- algorithms.
- their hardware embodiment
- and their linguistic expression

with extensions toward

- hardware technologies
- and real applications.

The initial and the most abstract level of computation is represented by the algorithmic level. Algorithms specify what are the steps to be executed in order to perform a computation. The most actual level consists in two realms: (1) the huge and complex domain of the application software and (2) the very tangible domain of the real machines implemented in a certain technology. Both contribute to implement real functions (asked, or aggressively imposed, my the so called free market). An intermediate level provides the means to be used for allowing an algorithm to be embodied in a physical structure of a machine or in an informational structure of a program. It is about (1) the domain of the formal programming languages, and (2) the domain of hardware architecture. Both of them are described using specific and rigorous formal tools.

The hardware embodiment of computations is done in digital systems. What kind of formal tools are used to describe, in the most flexible and efficient way, a complex digital system? Figure 1.8 presents the formal context in which the description tools are considered. Pseudo-code language is an easy to understand and easy to use way to express algorithms. Anything about computation can be expressed using this kind of languages. By the rule, in a pseudo-code language we express, for our (human) mind, preliminary, not very well formally expressed, ideas about an algorithm. The “main user” of this kind of language is only the human mind. But, for building complex applications or for accessing advanced technologies involved in building big digital systems, we need refined, rigorous formal languages and specific styles to express computation. More, for a rigorous formal language we must take into account
1.1. FRAMING THE DIGITAL DESIGN DOMAIN

that the “main user” is a merciless machine, instead of a tolerant human mind. Elaborated programming languages (such as C++, Java, Prolog, Lisp) are needed for developing complex contexts for computation and to write using them real applications. Also, for complex hardware embodiments specific hardware description languages, HDL, (such as Verilog, VHDL, SystemC) are proposed.

Figure 1.8: **The linguistic context in computer science.** Human mind uses pseudo-code languages to express informally a computation. To describe the circuit associated with the computation a rigorous HDL (hardware description language) is needed, and to describe the program executing the computation rigorous programming languages are used.

Both, general purpose programming languages and HDLs are designed to describe something for another program, mainly for a compiler. Therefore, they are more complex and rigorous than a simple pseudo-code language.

The starting point in designing a digital system is to describe it using what we call a specification, shortly, a spec. There are many ways to specify a digital system. In real life a hierarchy of specs are used, starting from high-level informal specs, and going down until the most detailed structural description is provided. In fact, the design process can be seen as a stream of descriptions which starts from an idea about how the new object to be designed behaves, and continues with more detailed descriptions, in each stage more behavioral descriptions being converted in structural descriptions. At the end of the process a full structural description is provided. The design process is the long way from a spec about what we intend to do to another spec describing how our intention can be fulfilled.

At one end of this process there are innovative minds driven by the will to change the world. In these imaginative minds there is no knowledge about “how”, there is only willingness about “what”. At the other end of this process there are very skilled entities “knowing” how to do very efficiently what the last description provides. They do not care to much about the functionality they implement. Usually, they are machines driven by complex programs.

In between we need a mixture of skills provided by very well instructed and trained people. The role of the imagination and of the very specific knowledge are equally important.

How can be organized optimally a designing system to manage the huge complexity of this big chain, leading from an idea to a product? There is no system able to manage such a complex process. No one can teach us about how to organize a company to be successful in introducing, for example, a new processor on the real market. The real process of designing and imposing a new product is trans-systemic. It is a rationally adjusted chaotic process for which no formal rules can ever provided.

Designing a digital system means to be involved in the middle of this complex process, usually far
CHAPTER 1. WHAT'S A DIGITAL SYSTEM?

away from its ends. A digital system designer starts his involvement when the specs start to be almost rigorously defined, and ends its contribution before the technological borders are reached.

However, a digital designer is faced in his work with few level of descriptions during the execution of a project. More, the number of descriptions increases with the complexity of the project. For a very simple project, it is enough to start from a spec and the structural description of the circuit can be immediately provided. But for a very complex project, the spec must be split in specs for sub-systems, each sub-system must be described first by its behavior. The process continue until enough simple sub-systems are defined. For them structural descriptions can be provided. The entire system is simulated and tested. If it works synthesisable descriptions are provided for each sub-system.

A good digital designer must be well trained in providing various description using an HDL. She/he must have the ability to make, both behavioral and structural descriptions for circuits having any level of complexity. Playing with inspired partitioning of the system, a skilled designer is one who is able to use appropriate descriptions to manage the complexity of the design.

1.2 Defining a digital system

Digital systems belong to the wider class of the discrete systems (systems having a countable number of states). Therefore, a general definition for digital system can be done as a special case of discrete system.

Definition 1.2 A digital system, DS, in its most general form is defined by specifying the five components of the following quintuple:

\[ DS = (X, Y, S, f, g) \]

where: \( X \subseteq \{0, 1\}^n \) is the input set of \( n \)-bit binary configurations, \( Y \subseteq \{0, 1\}^m \) is the output set of \( m \)-bit binary configurations, \( S \subseteq \{0, 1\}^q \) is the set of internal states of \( q \)-bit binary configurations,

\[ f : (X \times S) \rightarrow S \]

is the state transition function, and

\[ g : (X \times S) \rightarrow Y \]

is the output transition function.

A digital system (see Figure 1.9) has two simultaneous evolutions:

- the evolution of its internal state which takes into account the current internal state and the current input, generating the next state of the system

- the evolution of its output, which takes into account the current internal state and the current input generating the current output.

The internal state of the system determines the partial autonomy of the system. The system behaves on its outputs taking into account both, the current input and the current internal state. Because all the sets involved in the previous definition have the form \( \{0, 1\}^b \), each of the \( b \) one-bit input, output, or state evolves in time switching between two values: 0 and 1. The previous definition specifies a system having a \( n \)-bit input, an \( m \)-bit output and a \( q \)-bit internal state. If \( x_i \in X = \{0, 1\}^n \),
1.2. DEFINING A DIGITAL SYSTEM

State transition function $f: (X \times S) \rightarrow S$

Output transition function $g: (X \times S) \rightarrow Y$

Figure 1.9: Digital system.

$y_t \in Y = \{0,1\}^m, s_t \in S = \{0,1\}^q$ are values on input, output, and of state at the discrete moment of time $t$, then the behavior of the system is described by:

$$s_t = f(x_{t-1}, s_{t-1})$$

$$y_t = g(x_t, s_t)$$

While the current output is computed from the current input and the current state, the current state was computed using the previous input and the previous state. The two functions describing a discrete system belong to two distinct class of functions:

**sequential functions**: used to generate a sequence of values each of them iterated from its predecessor (an initial value is always provided, and the $i$-th value cannot be computed without computing all the previous $i - 1$ values); it is about functions such as $s_t = f(x_{t-1}, s_{t-1})$

**non-sequential functions**: used to compute an output value starting only from the current values applied on its inputs; it is about functions such as $y_t = g(x_t, s_t)$.

Depending on how the functions $f$ and $g$ are defined results a hierarchy of digital systems. More on this in the next chapters.

The variable time is essential for the formal definition of the sequential functions, but for the formal definition of the non-sequential ones it is meaningless. But, for the actual design of both, sequential and non-sequential function the time is a very important parameter.
CHAPTER 1. WHAT’S A DIGITAL SYSTEM?

Results the following requests for the simplest embodiment of an actual digital systems:

- the elements of the sets $X$, $Y$ and $S$ are binary codes of $n$, $m$ and $q$ bits – 0s and 1s – which are be
coded by two electric levels; the current technologies work with 0 Volts for the value 0, and with
a tension level in the range of 1-2 Volts for the value 1; thus, the system receives on its inputs:

$$X_{n-1}, X_{n-2}, \ldots, X_0$$

stores the internal state of form:

$$S_{q-1}, S_{q-2}, \ldots, S_0$$

and generate on its outputs:

$$Y_{m-1}, Y_{m-2}, \ldots, Y_0$$

where: $X_i, S_j, Y_k \in \{0, 1\}$.

- physical modules (see Figure 1.10), called combinational logic circuits – CLC –, to compute
functions like $f(x_t, s_t)$ or $g(x_t, s_t)$, which continuously follow, by the evolution of their output
values delayed with the propagation time $t_p$, any change on the inputs $x_t$ and $s_t$ (the shaded time
interval on the wave out represent the transient value of the output)

Figure 1.10: The module for non-sequential functions. a. The table used to define the function as a
correspondence between all input binary configurations in and binary configurations out. b. The logic symbol for
the combinational logic circuit – CLC – which computes out = $F(in)$. c. The wave forms describing the time
behaviour of the circuit.

- a “master of the discrete time” must be provided, in order to make consistent suggestions for the
simple ideas as “previous”, “now”, “next”; it is about the special signal, already introduced, having
form of a square wave periodic signal, with the period $T$ which swings between the logic level 0
and the logic level 1; it is called clock, and is used to “tick” the discrete time with its active edge
(see Figure 1.11 where a clock signal, active on its positive edge, is shown)

- a storing support to memorize the state between two successive discrete moments of time is re-
quired; it is the register used to register, synchronized with the active edge of the clock signal, the
state computed at the moment $t$ in order to be used at the next moment, $t$, to compute a new
state and a new output; the input must be stable a time interval $t_{su}$ (set-up time) before the active
1.2. DEFINING A DIGITAL SYSTEM

Figure 1.11: The clock. This clock signal is active on its positive edge (negative edge as active edge is also possible). The time interval between two positive transitions is the period $T_{\text{clock}}$ of the clock signal. Each positive transition marks a discrete moment of time.

edge of clock, and must stay unchanged $t_h$ (hold time) after; the propagation time after the clock is $t_p$.

(More complex embodiment are introduced later in this text book. Then, the state will have a structure and the functional modules will result as multiple applications of this simple definition.)

The most complex part of defining a digital system is the description of the two functions $f$ and $g$. The complexity of defining how the system behaves is managed by using various Hardware Description Languages – HDLs. The formal tool used in this text book is the Verilog HDL. The algebraic description of a digital system provided in Definition 1.2 will be expressed as the Verilog definition.

**Definition 1.3** A digital system is defined by the Verilog module `digitalSystem`, an object which consists of:

- **external connections**: lists the type, the size and the name of each connection
- **internal resources**: of two types, as follows
  - **storage resources**: one or more registers used to store (to `register`) the internal state of the system
  - **functional resources**: of two types, computing the transition functions for
    - `state`: generating the `nextState` value from the current state and the current input
    - `output`: generating the current output value from the current state and the current input

The simplest Verilog definition of a digital system follows (see Figure 1.13). It is simple because the state is defined only by the content of a single $q$-bit register (the state has no structure) and the functions are computed by combinational circuits.

There are few keywords which any text editor emphasize using bolded and colored letters:

- **module** and **endmodule** are used to delimit the definition of an entity called module which is an object with inputs and outputs
- **input** denotes an input connection whose dimension, in number of bits, is specified in the associated square brackets as follows: `[n-1:0]` which means the bits are indexed from $n-1$ to $0$ from left to right
- **output** denotes an output connection whose dimension, in number of bits, is specified in the associated square brackets as follows: `[n-1:0]` which means the bits are indexed from $n-1$ to $0$ from left to right
CHAPTER 1. WHAT'S A DIGITAL SYSTEM?

Figure 1.12: The register. a. The wave forms describing timing details about how the register switches around the active edge of clock. b. The logic symbol used to define the static behaviour of the register when both, inputs and outputs are stable between two active edges of the clock signal.

- **reg [n-1:0]** defines a storage element able to store n bits synchronized with the active edge of the clock signal.

- **wire [n-1:0]** defines a n-bit internal connection used to interconnect two subsystems in the module.

- **always @ (event) action** specifies the action action triggered by the event event; in our first example the event is the positive edge of clock (posedge clock) and the action is: the state register is loaded with the new state stateRegister <= nextState.

- `include` is the command used to include the content of another file.

- "fileName.v" specifies the name of a Verilog file.

The following two dummy modules are used to synthesize the top level of the system; their content is not specified, because we do not define a specific system; only the frame of a possible definition is provided.

```verilog
module stateTransition #( `include "0_parameter.v")
(output [q-1:0] next ,
 input [q-1:0] state,
 input [n-1:0] in );
// describe here the state transition function
endmodule
```

```verilog
module outputTransition #( `include "0_parameter.v")
(output [m-1:0] out ,
 input [q-1:0] state,
```

```verilog```
1.2. DEFINING A DIGITAL SYSTEM

```verilog
module digitalSystem #(include "0_parameter.v")
    (output [m-1:0] out , // generates elements from the set Y
     input [n-1:0] in , // receives elements from the set X
     input
     clock);

    reg [q-1:0] state; // stores elements from the set S
    wire [q-1:0] next; // the value of the next state

    stateTransition stateTrans(state,
                                   state,
     in ); // f: (X x S) -> S

    always @(posedge clock) state <= next; // state update

    outputTransition outTrans(out ,
                                state,
     in ); // g: (X x S) -> Y

endmodule
```

Figure 1.13: The top module for the general form of a digital system.

```verilog
input [m-1:0] in );
     // describe here the output transition function
endmodule
```

where the content of the file 0_parameter.v is:

```verilog
parameter n = 8, // the input is coded on 8 bits
     m = 8, // the output is coded on 8 bits
     q = 4 // the state is coded on 4 bits
```

It must be actually defined for synthesis reasons. The synthesis tool must "know" the size of the internal and external connections, even if the actual content of the internal modules is not yet specified.

The synthesis of the generic structure, just defined, is represented in Figure 1.14, where there are represented three (sub-)modules:

- the module `fd` which is a 4-bit state register whose output is called `state[3:0]`; it stores the internal state of the system
- the module `stateTransition` instantiated as `stateTrans`; it computes the value of the state to be loaded in the state register in triggered by the next active (positive, in our example) edge of clock; **this module closes a loop over the state register**
- the module `outputTransition` instantiated as `outTrans`; it computes the output value from the current states and the current input (for some applications the current input is not used directly to generate the output, its contribution to the output being delayed through the state register).
The internal modules are interconnected using also the wire called next. The clock signal is applied only to the register. The register module and the module stateTransition compute a sequential function, while the outputTransition module computes a non-sequential, combinational function.

1.3 Our first target

We will pass twice through the matter of digital systems. Every time we have a specific target. In this section the first target is presented. It consists of a simple system used to introduce the basic knowledge about simple and small digital circuits. Our target has the form of a simple specific circuit. It is about a digital pixel corrector.

A video sensor is a circuit built as a big array of cells which provides the stream of binary numbers used to represent a picture or a frame in a movie. To manufacture such a big circuit without any broken cell is very costly. Therefore, circuits with a small number of isolated wrong cells, providing the erroneous signal zero, are accepted, because it is easy to make few corrections on an image containing millions of pixels. The error manifests by providing a zero value for the light intensity. The stream of numbers generated by Video Sensor is applied to the input of Digital Pixel Corrector (see Figure 1.15) which performs the correction. It consists of detecting the zeroes in the digital video stream ($s(t) = 0$) and of replacing them with the corrected value $s'(t)$ obtained by interpolation. The simplest interpolation uses $s(t-1)$ and $s(t+1)$ as follows:

$$s'(t) = \begin{cases} \frac{s(t-1) + s(t+1)}{2} & \text{if } (s(t) = 0) \\ s(t) & \text{else} \end{cases}$$

The circuits checks if the input is zero (if $s(t) = 0$). If not, $s(t)$ goes through. If the input is wrong the circuit provides the arithmetic mean computed in the smallest neighborhood, $s(t-1)$ and $s(t+1)$.

The previous simple computation is made by the circuit Digital Pixel Corrector. In Figure 1.16 the wave form $s(t)$ represents the light intensity, while $s(t)$ represents the discrete stream of samples to be converted in numbers. The stream $s(t)$ follows the wave $s(t)$, excepting in one point where the value provided by the conversion circuit is, by error, zero.

In our simple example, the circuit Digital Pixel Corrector receives from the convertor a stream of 4-bit numbers. The data input of the circuit is in [3:0]. It receives also the clock signal. The active edge of clock is, in this example, the positive transition.
Figure 1.15: **Pixel correction system.** The digital stream of pixels is corrected by substituting the wrong values by interpolation.

The stream of numbers received by the circuit is: 7, 9, 10, 11, 12, 11, 10, 8, 5, 3, 2, 2, 0, 5, 7, 9, 11, 12, 12, 13, ... On the 13-th position the wrong value, 0, is received. It will be substituted, in the output sequence, with the integer part of \((2 + 5)/2\).

In the first clock cycle, \(\text{in}[3:0]\) takes the value 0111, i.e., \(\text{in}[3] = 0, \text{in}[2] = 1, \text{in}[1] = 1, \text{in}[0] = 1\). In the second clock cycle \(\text{in}[3:0] = 1010\), and so on. Thus, on each binary input, \(\text{in}[3]\), ... \(\text{in}[0]\), a specific square wave form is applied. They consists of transitions between 0 and 1. In real circuits, 0 is represented by the voltage 0, while 1 by a positive voltage \(D_{DD} = 1\ldots 2V\).

A compact representation of the four wave form, \(\text{in}[3], \text{in}[2], \text{in}[1], \text{in}[0]\), is shown in the synthetic wave form \(\text{in}[3:0]\). It is a conventional representation. The two overlapped wave suggest the transition of the input value \(\text{in}[3:0]\), while “inside” each delimited time interval the decimal value of the input is inserted. The simultaneous transitions, used to delimit a time interval, signify the fact that some bits could switch form 0 to 1, while others from 1 to 0.

The output behaviour is represented in the same compact way. Our circuit transmits the input stream to the output of the circuit with a delay of two clock cycles! Why? Because, to compute the current output value \(\text{out}(t)\) the circuit needs the previous input value and the next input value. Any value must "wait" the next one to be "loaded" in the correction circuit, while the previous is already memorized. "Waiting" and “already memorized” means to be stored in the internal state of the system. Thus, the internal state consists of three sub-states: \(\text{ss1}[3:0] = \text{in}(t-1), \text{ss2}[3:0] = \text{in}(t-2)\) and \(\text{ss3}[3:0] = \text{in}(t-3)\), i.e., state is the concatenation of the three sub-states:

\[
\text{state} = \{\text{ss3}[3:0], \text{ss2}[3:0], \text{ss1}[3:0]\}
\]

In each clock cycle the state is updated with the current input, as follows:

\[
\text{if state}(t) = \{\text{ss3}, \text{ss2}, \text{ss1}\} \text{ then state}(t+1) = \{\text{ss2}, \text{ss1, in}\}
\]

Thus, the circuit takes into account simultaneously three successive values from the input stream, all stored, concatenated, in the internal state register. The previously stated interpolation relation is now reconsidered, for an actual implementation using a digital system, as follows:

\[
\text{out}(t) = \begin{cases} 
\text{if } (\text{ss2} = 0) \\
\text{then } (\text{ss1} + \text{ss3})/2 \\
\text{else } \text{ss2}
\end{cases}
\]

If no wrong value on the stream, then the current output takes the value of the input received two cycles before: one to load it as \(\text{ss1}\) and another to move it in \(\text{ss2}\)(two clock cycles delay, or two-clock latency). Else, the current output value is (partially) “restored” from the other two sub-states of the system \(\text{ss1}\).
and ss3, first just received triggered by the last active edge of the clock, and the second loaded two cycles before.

Now, let us take the general definition of a digital system and adapt it for designing the Digital Pixel Corrector circuit. First, the file used to define the parameters – 0\_parameter\_v – is modified according to the size of the external connections and of state. For our application, the file takes the name 0\_paramPixelCor\_v, having the following content:

```plaintext
parameter n = 4, // the input is coded on 4 bits
    m = 4, // the output is coded on 4 bits
    q = 12 // the state is coded on 12 bits (to store three 4-bit values)
```

In the top module digitalSystem little changes are needed. The module’s name is changed to pixelCorrector, the included parameter file is substituted, and the module outputTransition is simplified, because the output value does not depend directly by the input value. The resulting top
1.3. OUR FIRST TARGET

module is:

module pixelCorrector #(`include "0_paramPixelCor.v")
(output [m-1:0] out ,
 input [n-1:0] in ,
 input clock);
reg [q-1:0] state;
wire [q-1:0] next ;
stateTransition stateTrans(next ,
 state, 
in );
always @(posedge clock) state <= next;
outputTransition outTrans(out ,
 state);
endmodule

Now, the two modules defining the transition functions must be defined according to the functionality desired for the system. State transition means to shift left the content of the state register n positions and on the freed position to put the input value. The output transition is a conditioned computation. Therefore, for our pixelCorrector module the combinational modules have the following form:

module stateTransition #(`include "0_paramPixelCor.v")
(output [q-1:0] next ,
 input [q-1:0] state,
 input [n-1:0] in );
//state[2*n-1:0] is {in(t-2), in(t-1)}
assign next = {state[2*n-1:0], in};
endmodule

module outputTransition #(`include "0_paramPixelCor.v")
(output reg [m-1:0] out ,
 input [q-1:0] state);
//state[n-1:0] is in(t-1)
//state[2*n-1:n] is in(t-2)
//state[q-1:2*n] is in(t-3)
always @(state) // simpler: "always @(*)"
if (state[2*n-1:n] == 0) out = (state[n-1:0] + state[q-1:2*n])/2;
else out = state[2*n-1:n];
endmodule

In order to verify the correctness of our design, a simulation module is designed. The clock signal and the input stream are generated and applied to the input of the top module, instantiated under the name dut (devices under test). A monitor is used to access the behavior of the circuit.

module testPixelCorrector #(`include "0_paramPixelCor.v");
 reg clock;
 reg [3:0] in ;
 wire [3:0] out ;
initial begin
clock = 0;
forever #1 clock = ~clock;
end

initial begin
in = 4'b0111;
#2 in = 4'b1001;
#2 in = 4'b1010;
#2 in = 4'b1011;
#2 in = 4'b1100;
#2 in = 4'b1010;
#2 in = 4'b1000;
#2 in = 4'b0101;
#2 in = 4'b0011;
#2 in = 4'b0010;
#2 in = 4'b0000; // the error
#2 in = 4'b0101;
#2 in = 4'b0111;
#2 in = 4'b1001;
#2 in = 4'b1011;
#2 in = 4'b1100;
#2 in = 4'b1100;
#2 in = 4'b1101;
#2 $stop;
end

pixelCorrector dut(out ,
    in ,
clock);

initial $monitor("time = %d state = %b_%b_%b out = %b",
    $time, dut.state[q-1:2*n], dut.state[2*n-1:n],dut.state[n-1:0], out);
endmodule

The monitor provides the following stream of data:

<table>
<thead>
<tr>
<th>posedge clock</th>
<th>ss3</th>
<th>ss2</th>
<th>ss1</th>
<th>out = if (ss2=0) (ss3+ss1)/2 else ss2</th>
</tr>
</thead>
<tbody>
<tr>
<td># time = 0</td>
<td>state = xxxx_xxxx_xxxx out = xxxx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># time = 1</td>
<td>state = xxxx_xxxx_0111 out = xxxx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># time = 5</td>
<td>state = 0111_1001_1010 out = 1001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># time = 7</td>
<td>state = 1001_1010_1011 out = 1010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># time = 9</td>
<td>state = 1010_1011_1100 out = 1011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># time = 11</td>
<td>state = 1011_1100_1011 out = 1100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># time = 13</td>
<td>state = 1100_1011_1010 out = 1011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># time = 15</td>
<td>state = 1011_1010_1000 out = 1010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># time = 17</td>
<td>state = 1010_1000_0101 out = 1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># time = 19</td>
<td>state = 1000_0101_0011 out = 0101</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1.4. PROBLEMS

while the wave form of the same simulation are presented in Figure 1.17. The output values correspond to the input values with a two clock cycle latency. Each nonzero input goes through the output in two cycles, while the wrong, zero inputs generate the interpolated values in the same two cycles (in our example 0 generates 3, as the integer mean value of 2 and 5). The functions involved in solving the pixel correction are:

- the predicate function, \( \text{state}[2*n-1:n] == 0 \), used to detect the wrong value zero
- the addition, used to compute the mean value of two numbers
- the division, used to compute the mean value of two numbers
- the selection function, which according to the a predicate sends to the output one value or another value
- storage function, triggered by the active edge of the clock signal

Our first target is to provide the knowledge about the actual circuits used in the previous simple application. The previous description is a behavioral description. We must acquire the ability to provide the corresponding structural description. How to add, how to compare, how to select, how to store, and few other similar function will be investigated before going to approach the final, more complex target.

1.4 Problems

Problem 1.1 How behaves the pixelCorrector circuit if the very first value received is zero? How can be improved the circuit to provide a better response?

Problem 1.2 How behaves the pixelCorrector circuit if if two successive wrong zeroes are received to the input? Provide an improvement for this situation.
Problem 1.3 What is the effect of the correction circuit when the zero input comes from an actual zero light intensity?

Problem 1.4 Synthesize the module pixelCorrector and identify in the RTL Schematic provided by the synthesis tool the functional components of the design. Explain the absences, if any.

Problem 1.5 Design a more accurate version of the pixel correction circuit using a more complex interpolation rule, which takes into account an extended neighborhood. For example, apply the following interpolation:

\[ s'(t) = 0.2s(t-2) + 0.3s(t-1) + 0.3s(t+1) + 0.2s(t+2) \]
Chapter 2

DIGITAL CIRCUITS

In the previous chapter
the concept of digital system was introduced by:

- differentiating it from analog system
- but integrating it, in the same time, in a hybrid electronic system
- defining formally what means a digital system
- and by stating the first target of this text book: the introducing the basic small and simple digital circuits

In this chapter
general definitions related with the digital domain are used to reach the following targets:

- to frame the digital system domain in the larger area of the information technologies
- to present different ways the digital approach is involved in the design of the real market products
- to enlist and shortly present the related domains, in order to integrate better the knowledge and skills acquired by studying the digital system design domain

In the next chapter
is a friendly introduction in both, digital systems and a HDLs (Hardware Description Languages) used to describe, simulate, and synthesized them. The HDL selected for this book is called Verilog. The main topics are:

- the distinction between combinational and sequential circuits
- the two ways to describe a circuit: behavioral or structural
- how digital circuits behave in time.
In the previous chapter we learned, from an example, that a simple digital system, assimilated with a digital circuit, is built using two kinds of circuits:

- non-sequential circuits, whose outputs follow continuously, with a specific delay, the evolution of input variable, providing a “combination” of input bits as the output value

- sequential circuits, whose output evolve triggered by the active edge of the special signal called *clock* which is used to determine the “moment” when the a storage element changes its content.

Consequently, in this chapter are introduced, by simple examples and simple constructs, the two basic types of digital circuits:

- **combinational circuits**, used to compute $f_{comb} : X \rightarrow Y$, defined in $X = \{0, 1\}^n$ with values in $Y = \{0, 1\}^m$, where $f_{comb}(x(t)) = y(t)$, with $x(t) \in X$, $y(t) \in Y$ representing two values generated in the same *discrete unit* of time $t$ (discrete time is “ticked” by the active edge of clock)

- **storage circuits**, used to design sequential circuits, whose outputs follow the input values with the delay of one clock cycle; $f_{store} : X \rightarrow X$, defined in $X = \{0, 1\}^n$ with values in $X = \{0, 1\}^n$, where $f_{store}(x(t)) = x(t-1)$, with $x(t), x(t-1) \in X$, representing the same value considered in two successive units of time, $t-1$ and $t$.

While a combinational circuit computes continuously its outputs according to each input change, the output of the storage circuit changes only triggered by the active edge of clock.

In this chapter, the first section is for combinational circuits which are introduced by examples, while, in the second section, the storage circuit called *register* is generated step by step starting from the simplest combinational circuits.

### 2.1 Combinational circuits

Revisiting the *Digital Pixel Corrector* circuit, lets take the functional description of the output function:

$$\begin{align*}
\text{if } (\text{state}[2*n-1:n] == 0) \quad \text{out} &= (\text{state}[n-1:0] + \text{state}[q-1:2*n])/2; \\
\text{else} \quad \text{out} &= \text{state}[2*n-1:n];
\end{align*}$$

The previous form contains the following elementary functions:

- **test function**: state[2*n-1:n] == 0, defined in $\{0, 1\}^n$ with value in $\{0, 1\}$

- **selection function**:

  $$\begin{align*}
  \text{if } (\text{test}) \quad \text{out} &= \text{action1}; \\
  \text{else} \quad \text{out} &= \text{action2};
  \end{align*}$$

  defined in the Cartesian product $\{(0, 1) \times \{0, 1\}^n \times \{0, 1\}^n\}$ with values in $\{0, 1\}^n$
2.1. COMBINATIONAL CIRCUITS

- **Add** function: state[n−1:0] + state[q−1:2n], defined in \((\{0,1\}^n \times \{0,1\}^n)\) with value in \(\{0,1\}^n\)

- **Divide by 2** function: defined in \(\{0,1\}^n\) with value in \(\{0,1\}^n\).

In Appendix D, section **Elementary circuits: gates** basic knowledge about Boolean logic and the associated logic circuits are introduced. We use simple functions and circuits, like AND, OR, NOT, XOR, ..., to design the previously emphasized combinational functions.

2.1.1 Zero circuit

The simplest test function tests if a \(n\)-bit binary configuration represents the number 0. The function OR provides 1 if at least one of its inputs is 1, which means it provides 0 if all its inputs are 0. Then, inverting – negating – the output of a \(n\)-input OR we obtain a circuit NOR – not OR – whose output is 1 only when all its inputs are 0.

**Definition 2.1** *The \(n\)-input Zero circuit is a \(n\)-input NOR.*

The Figure 2.1 represents few embodiment of the Zero circuit. The elementary, 2-input, Zero circuit is represented in Figure 2.1a as a two-input NOR. For the \(n\)-input Zero circuit a \(n\)-input NOR is requested (see Figure 2.1b) which can be implemented in two different ways (see Figure 2.1c and Figure 2.1d). One level NOR (see Figure 2.1b) with more than 4 inputs are impractical (for reasons disclosed when we will enter in the physical details of the actual implementations).

**Figure 2.1:** The Zero circuit. a. The 2-input Zero circuit is a 2-input NOR. b. The \(n\)-input Zero circuit is a \(n\)-input NOR. c. The 8-input Zero circuit as a degenerated tree of 2-input ORs. d. The 8-input Zero circuit as a balanced tree of 2-input ORs. e. The logic symbol for the Zero circuit.

The two solution for the \(n\)-input NOR come from the two ways to expand an associative logic function. It is about how the parenthesis are used. The first form (see Figure 2.1c) comes from:

\[(a + b + c + d + e + f + g + h)’ = (((((a + b) + c) + d) + e) + f) + g) + h)’\]
CHAPTER 2. DIGITAL CIRCUITS

generating a 7 level circuit (7 included parenthesis), while, the second form (see Figure 2.1d) comes from:

\[(a+b+c+d+e+f+g+h)' = ((a+b)+(c+d)+(e+f)+(g+h))' = (((a+b)+(c+d))+(e+f)+(g+h))'\]

providing a 3 level circuit (3 included parenthesis). The number of gates used is the same for the two solution. We expect that the second solution provide a faster circuit.

2.1.2 Selection

The selection circuit, called also multiplexer, is a three input circuit: a one-bit selection input – \(sel\) –, and two selected inputs, one – \(in0\) – selected to the output when \(sel = 0\) and another – \(in1\) – selected for \(sel = 1\). Let us take first the simplest case when both selected inputs are of 1 bit. This is the case for the elementary multiplexer, EMUX. If the input are: \(sel, in0, in1\), then the logic equation describing the logic circuit is:

\[out = sel' \cdot in0 + sel \cdot in1\]

Then the circuit consists of one NOT, two ANDs and one OR as it is shown in Figure 2.2. The AND gates are opened by selection signal, \(sel\), allowing to send out the value applied on the input \(in1\), and by the negation of the selection signal signal, \(sel'\), allowing to send out the value applied on the input \(in0\). The OR circuit “sum up” the outputs of the two ANDs, because only one is “open” at a time.

Figure 2.2: The selection circuit. a. The logic schematic for the elementary selector, EMUX (elementary multiplexer). b. The logic symbol for EMUX. c. The selector (multiplexor) for \(n\)-bit words, \(MUX_n\). d. The logic symbol for \(MUX_n\).

The selection circuit for two \(n\)-bit inputs is functionally (behaviorally) described by the following Verilog module:
module ifThenElse #(parameter n = 4)
  (output [n-1:0] out,
   input  sel,
   input [n-1:0] in1, in0);

  assign out = sel ? in1 : in0;
endmodule

In the previous code we decided to design a circuit for 4-bit data. Therefore, the parameter \(n\) is set to 4 only in the header of the module.

The structural description is much more complex because it specifies all the details until the level of elementary gates. The description has two modules: the top module – ifThenElse – and the module describing the simplest select circuit – eMux.

module eMux(output out,
            input  sel, in1, in0);

  wire           invSel;
  not inverter(invSel, sel);
  and and1(out1, sel, in1),
               and0(out0, invSel, in0);
  or  outGate(out, out1, out0);
endmodule

module ifThenElse #(parameter n = 4)
  (output [n-1:0] out,
   input  sel,
   input [n-1:0] in1, in0);

  genvar i ;
  generate for (i=0; i<n; i=i+1)
  begin: eMUX
    eMux selector(.out(out[i]),
                   .sel(sel ),
                   .in1(in1[i]),
                   .in0(in0[i]));
  end
  endgenerate
endmodule

The repetitive structure of the circuit is described using the generate form.

To verify the design a test module is designed. This module generate stimuli for the input of the device under test (dut), and monitors the inputs and the outputs of the circuit.

module testIfThenElse #(parameter n = 4);
  reg  [n-1:0] in1, in0;
  reg       sel;
  wire [n-1:0] out;

  initial begin
    in1 = 4’b0101;
    in0 = 4’b1011;
    sel = 1’b0;
  end
endmodule
### 2.1.3 Adder

A \( n \)-bit adder is defined as follows, using a Verilog behavioral description:

```verilog
module adder #(parameter n = 4) // defines a n-bit adder
     (output [n-1:0] sum, // the n-bit result
       output carry, // carry output
       input c, // carry input
       input [n-1:0] a, b); // the two n-bit numbers
```

The result of synthesis is represented in Figure 2.3.

Figure 2.3: The result of the synthesis for the module ifThenElse.
assign {carry, sum} = a + b + c;
endmodule

Fortunately, the previous module is synthesisable by the currently used synthesis tools. But, in this stage, it is important for us to define the actual internal structure of an adder. We start from a 1-bit adder, whose output are described by the following Boolean equations:

\[
\text{sum} = a \oplus b \oplus c
\]

\[
\text{carry} = a \cdot b + a \cdot c + b \cdot c
\]

where, \(a, b\) and \(c\) are one bit Boolean variables. Indeed, the \text{sum} output results as the sum of three bits: the two numbers, \(a\) and \(b\), and the carry bit, \(c\), coming from the previous binary range. As we know, the modulo 2 sum is performed by a XOR circuit. Then, \(a \oplus b\) is the sum of the two one-bit numbers. The result must be added with \(c - (a \oplus b) \oplus c\) – using another XOR circuit. The \text{carry} signal is used by the next binary stage. The expression for \text{carry} is written taking into account that the carry signal is one if at least two of the input bits are one: \text{carry} is 1 if \(a\ \text{and} \ b\) or \(a\ \text{and} \ c\) or \(b\ \text{and} \ c\) (the function is the \textit{majority function}). Its expression is embodied also in logic circuits, but not before optimizing its form as follows:

\[
\text{carry} = a \cdot b + a \cdot c + b \cdot c = a \cdot b + c \cdot (a + b) = a \cdot b + c \cdot (a \oplus b)
\]

Because \((a \oplus b)\) is already computed for \text{sum}, the circuit for \text{carry} requests only two ANDs and an OR. In Figure 2.4a the external connections of the 1-bit adder are represented. The input \(c\) receives the carry signal from the previous binary range. The output \text{carry} generate the carry signal for the next binary or range.

Figure 2.4: The adder circuit. a. The logic symbol for one-bit adder. b. The logic schematic for the one-bit adder. c. The block schematic for the \(n\)-bit adder.

The functions for the one bit adder are obtained formally, without any trick, starting from the truth table defining the operation (see Figure 2.5).

The two expressions are extracted from the truth table as “sum” of “products”. Only the “products” generating 1 to output are “summed”. Results:

\[
\text{sum} = a'bc' + a'bc + ab'c' + abc
\]
The truth table for the adder circuit. The first three columns contain all the three-bit binary configuration the circuit could receive. The two last columns describe the behavior of the sum and carry output.

\[
\begin{array}{cccc}
 a & b & c & \text{sum} & \text{carry} \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 1 & 1 & 0 \\
 0 & 1 & 0 & 1 & 0 \\
 0 & 1 & 1 & 0 & 1 \\
 1 & 0 & 0 & 1 & 0 \\
 1 & 0 & 1 & 0 & 1 \\
 1 & 1 & 0 & 0 & 1 \\
 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Figure 2.5: The truth table for the adder circuit. The first three columns contain all the three-bit binary configuration the circuit could receive. The two last columns describe the behavior of the sum and carry output.

\[
carry = a'bc + ab'c + abc' + abc
\]

and, using the Boolean algebra rules the form are reduced to the previously written expressions.

For a circuit with more than one output, minimizing it means to minimize the overall design, not only each expressions associated to its outputs. For our design – the one bit adder – the expression used to implement the sum output is not minimal. It is more complex that the minimal form (instead of an OR gate we used the more complicated gate XOR), but it contains a sub-circuit shared with the circuit associated to carry output. It is about the first XOR circuit (see Figure 2.4b).

2.1.4 Divider

The divide operation – \(a/b\) – is, in the general case, a complex operation. But, in our application – Digital Pixel Correction – it is about dividing by 2 a binary represented number. It is performed, without any circuit, simply by shifting the bits of the binary number one position to right. The number \(\text{number}[n-1:0]\) divided by two become \(\{1'b0, \text{number}[n-1:1]\}\).

2.2 Sequential circuits

In this section we intend to introduce the basic circuits used to build the sequential parts of a digital system. It is about the sequential digital circuits. These circuits are mainly used to build the storing subsystems in a digital system. To store in a digital circuit means to maintain the value of a signal applied on the input of the circuit. Simply speaking, the effect of the signal to be stored must be “re-applied” on another input of the circuit, so as the effect of the input signal to be memorized is substituted. Namely, the circuit must have a loop closed form one of its output to one of its input. The resulting circuit, instead of providing the computation it performs without loop, it will provide a new kind of functionality: the function of memorizing. Besides the function of memorizing, sequential circuits are used to design simple or complex automata (in this section we provide only examples of simple automata). The register, the typical sequential circuit, is used also in designing complex systems allowing efficient interconnections between various sub-systems.

2.2.1 Elementary Latches

This subsection is devoted to introduce the elementary structures whose internal loop allow the simplest storing function: latching an event.
The reset-only latch is the AND loop circuit represented in Figure 2.6a. The passive input value for AND loop is 1 \((R_{\text{eset}})' = 1\), while the active input value is 0 \((R_{\text{eset}})' = 0\). If the passive input value is applied, then the output of the circuits is not affected (the output depends only by the other input of the AND circuit). It can be 0 or 1, depending by the previous values applied on the input. When the active value is temporary applied, then the state of the circuit (the value of its output) switches in 0, with a delay of \(t_{\text{pHL}}\) (propagation time from high to low) and remains forever in this state, independent on the following the input value. We conclude that the circuit is sensitive to the signal 0 temporarily applied on its input, i.e., it is able to memorize forever the event 0. The circuit “catches” and “latches” the input value only if the input in maintained on 0 until the second input of the AND circuit receives the value 0, with a delay time \(t_{\text{pHL}}\). If the temporary input transition in 0 is too short the loop is unable to latch the event.

The set-only latch is the OR loop circuit represented in Figure 2.6b. The passive value for OR loop is 0 \((\text{Set} = 0)\) while the active input value is 1 \((\text{Set} = 1)\). If the passive input value is applied, then the output of the circuits is not affected (the output depends only by the other input of the OR circuit). It can be 0 or 1, depending by the previous values applied on the input. When the active value is temporary applied, then the state of the circuit (the value of its output) switches in 1 and remains forever in this state, independent on the input value. We conclude that the circuit is sensitive to the signal 1 temporarily applied on its input, i.e., it is able to memorize forever the event 1. The only condition, similar to that applied for AND loop, is to have an enough long duration of temporary input transition in 1.

The heterogenous set-reset latch results by combining the previous two latches (see Figure 2.6c). The circuit has two inputs: one active-low (active on 0) input, \(R'\), to reset the circuit \((\text{out} = 0)\), and another active-high (active on 1) input, \(S\), to set the circuit \((\text{out} = 0)\). The value 0 must remain to the input \(R'\) at least \(2t_{\text{pHL}}\) for a stable switching of the circuit into the state 0, because the loop depth in the state 1 is given by the propagation time through both gates that switch from high to low. For a similar reason, the value 1 must remain to the input \(S\) at least \(2t_{\text{pHL}}\) when the circuit must switch in 1. However, the output of the circuit reacts faster to the set signal, because from the input set to the output of the circuit there is only one gate, while from the other input to output the depth of the circuit is doubled.

The symmetric set-reset latches are obtained by applying De Morgan’s law to the heterogenous elementary latch. In the first version, the OR circuit is transformed by De Morgan’s law (the form \(a + b = (a' b')'\) is used) resulting the circuit from Figure 2.7a. The second version (see Figure 2.7b) is obtained applying the other form of the same law to the AND circuit \((ab = (a' + b')')\). The passive input value for the NAND elementary latch is 1, while for the NOR elementary latch it is 0. The active input value for the NAND elementary latch is 0, while for the NOR elementary latch it is 1. The symmetric structure of these latches have two outputs, \(Q\) and \(Q'\).

Although, the structural description in an actual design does not go until such detail, it is useful to use a simulation for understand how this small, simple, but fundamental circuit works. For the sake of simulation only, the description of the circuit contains time assignment. If the module is supposed to by eventually synthesised, then the time assignment must be removed.

\textbf{VeriSim 2.1} The Verilog structural description of NAND latch is:

```
module elementary_latch(output out, not_out,
                         input not_set, not_reset);
```
Figure 2.6: The elementary latches. Using the loop, closed from the output to one input, elementary storage elements are built. a. AND loop provides a reset-only latch. b. OR loop provides the set-only version of a storage element. c. The heterogeneous elementary set-reset latch results combining the reset-only latch with the set-only latch. d. The wave forms describing the behavior of the previous three latch circuits.

nand #2 nand0(out, not_out, not_set);
nand #2 nand1(not_out, out, not_reset);
endmodule

The two NAND gates considered in this simulation have the propagation time equal with 2 unit times – #2.

For testing the behavior of the NAND latch just described, the following module is used:

module test_shortest_input;
reg not_set, not_reset;
initial begin not_set = 1;
not_reset = 1;
#10 not_reset = 0; // reset
#10 not_reset = 1;
#10 not_set = 0; // set
#10 not_set = 1; // 1-st experiment
//#1 not_set = 1; // 2-nd experiment
//#2 not_set = 1; // 3-rd experiment
//#3 not_set = 1; // 4-th experiment
#10 not_set = 0; // another set
2.2. SEQUENTIAL CIRCUITS

Figure 2.7: Symmetric elementary latches. a. Symmetric elementary NAND latch with low-active commands $S'$ and $R'$. b. Symmetric elementary NOR latch with high-active commands $S$ and $R$.

```verilog
#10 not_set = 1;
#10 not_reset = 0; // reset
#10 not_reset = 1;
#10 $stop;
end

elementary_latch dut(out, not_out, not_set, not_reset);
endmodule
```

In the first experiment the set signal is activated on 0 during 10ut (ut stands for unit time). In the second experiment (comment the line 9 and de-comment the line 10 of the test module), a set signal of 1ut is unable to switch the circuit. The third experiment, with 2ut set signal, generate an unstable simulated, but non-actual, behavior (to be explained by the reader). The fourth experiment, with 3ut set signal, determines the shortest set signal able to switch the latch (to be explained by the reader).

In order to use these latches in more complex applications we must solve two problems.

The first latch problem: the inputs for indicating how the latch switches are the same as the inputs for indicating when the latch switches; we must find a solution for declutching the two actions building a version with distinct inputs for specifying “how” and “when”.

The second latch problem: if we apply synchronously $S'$=0 and $R'$=0 on the inputs of NAND latch (or $S$=1 and $R$=1 on the inputs of OR latch), i.e., the latch is commanded “to switch in both states simultaneously”, then we can not predict what is the state of the latch after the ending of these two active signals.

The first latch problem will be partially solved in the next subsection, introducing the clocked latch, but the problem will be completely solved only by introducing the master-slave structure. The second latch problem will be solved, only in one of the chapter that follow, with the JK flip-flop, because the circuit needs more autonomy to “solve” the contradictory command that “says him” to switch in both states simultaneously.

Application: de-bouncing circuit Interfacing digital systems with the real world involves sometimes the use of mechanical switching contacts. The bad news is that this kind of contact does not provide an
accurate transition. Usually when it closes, a lot of parasitic bounces come with the main transition (see wave forms $S'$ and $R'$ in Figure 2.8).

![De-bouncing circuit](image)

Figure 2.8: The de-bouncing circuit.

The debouncing circuit provide clean transitions when digital signals must generated by electro-mechanical switches. In Figure 2.8 an RS latch is used to clear up the bounces generated by a two-position electro-mechanical switch. The elementary latch latches the first transition from $V_{DD}$ to 0. The bounces that follow have no effect on the output $Q$ because the latch is already switched, by the first transition, in the state they intend to lead the circuit.

2.2.2 Elementary Clocked Latches

In order to start solving the first latch problem the elementary latch is supplemented with two gates used to validate the data inputs only during the active level of clock. Thus the clocked elementary latch is provided.

![Elementary clocked latch](image)

Figure 2.9: Elementary clocked latch. The transparent RS clocked latch is sensitive (transparent) to the input signals during the active level of the clock (the high level in this example). a. The internal structure. b. The logic symbol.

The NAND latch is used to exemplify (see Figure 2.9a) the partial separation between how and when.
The signals R’ and S’ for the NAND latch are generated using two 2-input NAND gates. If the latch must be set, then on the input S we apply 1, R is maintained in 0 and, only after that, the clock is applied, i.e., the clock input CK switches temporary in 1. In this case the active level of the clock is the high level. For reset, the procedure is similar: the input R is activated, the input S is inactivated, and then the clock is applied.

We said that this approach allows only a partial declutching of how by when because on the active level of CK the latch is transparent, i.e., any change on the inputs S and R can modify the state of the circuit. Indeed, if CK = 1 and S or R is activated the latch is set or reset, and in this case how and when are given only by the transition of these two signals, S for set or R for reset. The transparency will be avoided only when, in the next subsection, the transition of the output will be triggered by the active edge of clock.

The clocked latch does not solve the second latch problem, because for R = S = 1 the end of the active level of CK switches the latch in an unpredictable state.

**VeriSim 2.2** The following Verilog code can be used to understand how the elementary clocked latch works.

```verilog
module clocked_nand_latch(output out, not_out, 
                          input set, reset, clock);

  elementary_latch the_latch(out, not_out, not_set, not_reset);
  nand #2 nand2(not_set, set, clock);
  nand #2 nand3(not_reset, reset, clock);
endmodule
```

2.2.3 Data Latch

The second latch problem can be only avoided, not removed in this stage of our approach, by introducing a restriction on the inputs of the clocked latch. Indeed, introducing an inverter circuit between the inputs of the RS clocked latch, as is shown in Figure 2.10a, the ambiguous command (simultaneous set and reset) cannot be applied. Now, the situation R = S = 1 becomes impossible. The output is synchronized with the clock only if on the active level of CK the input D is stable.

We call the resulting one input with D (from Data). The circuit is called Data Latch, or simple D-latch.

![Figure 2.10: The data latch. Imposing the restriction R = S’ to an RS latch results the D latch without non-predictable transitions (R = S = 1 is not anymore possible). a. The structure. b. The logic symbol. c. An improved version for the data latch internal structure.](image-url)
The output of this new circuit follows continuously the input D during the active level of clock. Therefore, the autonomy of this circuit is questionable because act only in the time when the clock is inactive (on the inactive level of the clock). We say D latch is transparent on the active level of the clock signal, i.e, the output is sensitive, to any input change, during the active level of clock.

**VeriSim 2.3** The following Verilog code can be used to describe the behavior of a D latch.

```verilog
module data_latch( output reg out,
                  output not_out,
                  input data, clock);
    always @(*) if (clock) out = data;
    assign not_out = ~out;
endmodule
```

The main problem when data input D is separated by the timing input CK is the correlation between them. When this two inputs change in the same time, or, more precisely, during the same small time interval, some behavioral problems occur. In order to obtain a predictable behavior we must obey two important time restrictions: the set-up time and the hold time.

In Figure 2.10c an improved version of the circuit is presented. The number of components are minimized, the maximum depth of the circuit is maintained and the input load due to the input D is reduced from 2 to 1, i.e., the circuit generating the signal D is loaded with one input instead of 2, in the original circuit.

**VeriSim 2.4** The following Verilog code can be used to understand how a D latch works.

```verilog
module test_data_latch;
    reg data, clock;
    initial begin clock = 0;
        forever #10 clock = ~clock;
    end
    initial begin data = 0;
        #25 data = 1;
        #10 data = 0;
        #20 $stop;
    end
    data_latch dut(out, not_out, data, clock);
endmodule
```

The second initial construct from test_data_latch module can be used to apply data in different relation with the clock.
2.2. SEQUENTIAL CIRCUITS

The internal structure of the data latch (4 2-input NANDs and an inverter in Figure 2.10a) can be minimized opening the loop by disconnecting the output \( Q \) from the input of the gate generating \( Q' \), and renaming it \( C \). The resulting circuit is described by the following equation:

\[
Q = ((D \cdot CK)' \cdot (C(D' \cdot CK)'))'
\]

which can be successively transformed as follows:

\[
Q = ((D \cdot CK) + (C(D' \cdot CK)'))
\]

\[
Q = ((D \cdot CK) + (C(D + CK')))
\]

\[
Q = D \cdot CK + C \cdot D + C \cdot CK'(\text{anti – hasard redundancy}^1)
\]

\[
Q = D \cdot CK + C \cdot CK'
\]

The resulting circuit is an elementary multiplexor (the selection input is \( CK \) and the selected inputs are \( D \), by \( CK = 1 \), and \( C \), by \( CK = 0 \). Closing back the loop, by connecting \( Q \) to \( C \), results the circuit represented in Figure 2.11a. The actual circuit has also the inverted output \( Q' \) and is implemented using only inverted gates as in Figure 2.11b. The circuit from Figure 2.10a (using the RSL circuit from Figure 2.9a) is implemented with 18 transistors, instead of 12 transistors supposed by the minimized form Figure 2.11b.

---

**Figure 2.11: The optimized data latch.** An optimized version is implemented closing the loop over an elementary multiplexer, EMUX. **a.** The resulting minimized structure for the circuit represented in Figure 2.10a. **b.** Implementing the minimized form using only inverting circuits.

---

\[^1\text{Anti-hasard redundancy equivalence: } f(a,b,c) = ab + ac + bc' = ac + bc'\]

**Proof:**

\[
f(a,b,c) = ab + ac + bc' + cc', cc' \text{ is ORed because } xx' = 0 \text{ and } x = x + 0
\]

\[
f(a,b,c) = a(b + c) + c'(b + c) = (b + c)(a + c')
\]

\[
f(a,b,c) = (b + c)' + (a + c')', \text{ applying De Morgan law}
\]

\[
f(a,b,c) = (b'c' + a'c'), \text{ applying again De Morgan law}
\]

\[
f(a,b,c) = (ab'c' + a'b'c' + a'bc + a'b'c)' = (m4 + m0 + m3 + m1)', \text{ expanding to the disjunctive normal form}
\]

\[
f(a,b,c) = m2 + m5 + m6 + m7 = a'bc' + ab'c + abc' + abc, \text{ using the "complementary" minterms}
\]

\[
f(a,b,c) = bc'(a + a') + ac(b + b') = ac + bc', \text{ q.e.d.}
\]
VeriSim 2.5 The following Verilog code can be used as one of the shortest description for a D latch represented in Figure 2.11a.

```
module mux_latch( output q ,
                input d, ck );

assign q = ck ? d : q;
endmodule
```

In the previous module the assign statement, describing an elementary multiplexer, contains the loop. The variable q depends by itself. The code is synthesisable.

We ended using the elementary multiplexer to describe the most complex latch. This latch is used in structuring almost any storage sub-system in a digital system. Thus, one of the basic combinational function, associated to the main control function if-then-else, is proved to be the basic circuit in designing storage elements.

2.2.4 Master-Slave Principle

In order to remove the transparency of the clocked latches, disconnecting completely the how from the when, the master-slave principle was introduced. This principle allows us to build a two state circuit named flip-flop that switches synchronized with the rising or falling edge of the clock signal.

![Figure 2.12: The master-slave principle. Serially connecting two RS latches, activated with different levels of the clock signal, results a non-transparent storage element.](image)

The principle consists in serially connecting two clocked latches and in applying the clock signal in opposite on the two latches (see Figure 2.12a). In the exemplified embodiment the first latch is transparent on the high level of clock and the second latch is transparent on the low level of clock. (The symmetric situation is also possible: the first latch is transparent of the low level value of clock and the second no the high value of clock.) Therefore, there is no time interval in which the entire structure is transparent. In the first phase, \( CK = 1 \), the first latch is transparent - we call it the master latch - and it switches according to the inputs S and R. In the second phase \( CK = 0 \) the second latch - the slave latch - is transparent and it switches copying the state of the master latch. Thus the output of the entire structure is modified only synchronized with the negative transition of CK, i.e., only at the transition from 1 to 0.
of the clock, because the state of the master latch freezes until the clock switches back to 1. We say the
RS master-slave flip-flop switches always at (always @ expressed in Verilog) the falling (negative) edge
of the clock. (The version triggered by the positive edge of clock is also possible.)

The switching moment of a master-slave structure is determined exclusively by the active edge
of clock signal. Unlike the RS latch or data latch, which can sometimes be triggered (in the trans-
pparency time interval) by the transitions of the input data (R, S or D), the master-slave flip-flop flips
only at the positive edge of clock (always @(posedge clock)) or at the negative edge of clock (always
@(negedge clock)) edge of clock, according with the values applied on the inputs R and S. The how
is now completely separated from the when. The first latch problem is finally solved.

VeriSim 2.6 The following Verilog code can be used to understand how a master-slave flip-flop works.

module master_slave(output out, not_out, input set, reset, clock);

wire master_out, not_master_out;

clocked_nand_latch master_latch( .out (master_out ),
     .not_out(not_master_out ),
     .set (set ),
     .reset (reset ),
     .clock (clock ));

slave_latch( .out (out ),
     .not_out(not_out ),
     .set (master_out ),
     .reset (not_master_out ),
     .clock (~clock ));

endmodule

There are some other embodiments of the master-slave principle, but all suppose to connect latches
serially.

Three very important time intervals (see Figure 2.13) must catch our attention in designing digital
systems with edge triggered flip-flops:

set-up time – \( t_{SU} \) – the time interval before the active edge of clock in which the inputs R and S must
stay unmodified allowing the correct switch of the flip-flop

edge transition time – \( t_\pm \) – the positive or negative time transition of the clock signal

hold time – \( t_H \) – the time interval after the active edge of CK in which the inputs R and S must be
stable (even if this time is zero or negative).

In the switching “moment”, that is approximated by the time interval \( t_{SU} + t_+ + t_H \) or \( t_{SU} + t_- + t_H \)
“centered” on the active edge (+ or −), the data inputs must evidently be stable, because otherwise the
flip-flop “does not know” what is the state in which it must switch.

Now, the problem of decoupling the how by the when is better solved. Although, this solution is not
perfect, because the “moment” of the switch is approximated by the short time interval \( t_{SU} + t_{+/-} + t_H \).
But the "moment" does not exist for a digital designer. Always it must be a time interval, enough over-
estimated for an accurate work of the designed machine.
CHAPTER 2. DIGITAL CIRCUITS

Figure 2.13: **Magnifying the transition of the active edge of the clock signal.** The input data must be stable around the active transition of the clock $t_{su}$ (set-up time) before the beginning of the clock transition, during the transition of the clock, $t_a$ (active transition time), and $t_h$ (hold time) after the end of the active edge.

### 2.2.5 D Flip-Flop

Another tentative to remove the second latch problem leads to a solution that again avoids only the problem. Now the RS master-slave flip-flop is restricted to $R = S'$ (see Figure 2.14a). The new input is named also D, but now D means delay. Indeed, the flip-flop resulting by this restriction, besides avoiding the unforeseeable transition of the flip-flop, gains a very useful function: the output of the D flip-flop follows the D input with a delay of one clock cycle. Figure 2.14c illustrates the delay effect of this kind of flip-flop.

---

**Warning!** *D latch* is a transparent circuit during the active level of the clock, unlike the *D flip-flop* which is no time transparent and switches only on the active edge of the clock.

**VeriSim 2.7** *The structural Verilog description of a D flip-flop, provided only for simulation purpose, follows.*
2.2. SEQUENTIAL CIRCUITS

module dff(output out, not_out,
            input d, clock);
    wire not_d;
    not #2 data_inverter(not_d, d);
    master_slave rs_ff(out, not_out, d, not_d, clock);
endmodule

The functional description currently used for a D flip-flop active on the negative edge of clock is:

module dff(output reg out ,
            input d, clock);
    always @(negedge clock) out <= d;
endmodule

The main difference between latches and flip-flops is that over the D flip-flop we can close a new loop in a very controllable fashion, unlike the D latch which allows a new loop, but the resulting behavior is not so controllable because of its transparency. Closing loops over D flip-flops result in synchronous systems. Closing loops over D latches result in asynchronous systems. Both are useful, but in the first kind of systems the complexity is easiest manageable.

2.2.6 Register

One of the most representative and useful storage circuit is the register. The main application of register is to support the synchronous sequential processes in a digital system. There are two typical use of the register:

- provides a delayed connection between sub-systems
- stores the internal state of a system (see section 1.2); the register is used to close of the internal loop in a digital system.

The register circuit store synchronously the value applied on its inputs. Register is used mainly to support the design of control structures in a digital system.

The skeleton of any contemporary digital design is based on registers, used to store, synchronously with the system clock, the overall state of the system. The Verilog (or VHDL) description of a structured digital design starts by defining the registers, and provides, usually, an Register Transfer Logic (RTL) description. An RTL code describe a set of registers interconnected through more or less complex combinational blocks. For a register is a non-transparent structure any loop configurations are supported. Therefore, the design is freed by the care of the uncontrollable loops.

Definition 2.2 An n-bit register, \( R_n \), is made by parallel connecting a \( R_{n-1} \) with a D (master-slave) flip-flop (see Figure 2.15). \( R_1 \) is a D flip-flop.

VeriSim 2.8 An 8-bit enabled and resetable register with 2 unit time delay is described by the following Verilog module:
module register #(parameter n = 8)(output reg [n-1:0] out ,
    input [n-1:0] in ,
    input reset, enable, clock);
always @(posedge clock) #2 if (reset) out <= 0 ;
else if (enable) out <= in ;
else out <= out ;
endmodule

The time behavior specified by #2 is added only for simulation purpose. The synthesizable version must avoid this non-synthetisable representation.

⋄

The main feature of the register assures its non-transparency, excepting an "undecided transparency" during a short time interval, $t_{SU} + t_{edge} + t_{H}$, centered on the active edge of the clock signal. Thus, a new loop can be closed carelessly over a structure containing a register. Due to its non-transparency the register will be properly loaded with any value, even with a value depending on its own current content. This last feature is the main condition to close the loop of a synchronous automata - the structure presented in the next chapter.

The register is used at least for the following purposes: to store, to buffer, to synchronize, to delay, to loop, . . .

**Storing** The enable input allows us to determine when (i.e., in what clock cycle) the input is loaded into a register. If enable = 0, the registers stores the data loaded in the last clock cycle when the condition enable = 1 was fulfilled. This means we can keep the content once stored into the register as much time as it is needed.

**Buffering** The registers can be used to buffer (to isolate, to separate) two distinct blocks so as some behaviors are not transmitted through the register. For example, in Figure 2.16 the transitions from c to d and from d to e at the input of the register are not transmitted to the output.

**Synchronizing** For various reasons the digital signals are generated “unaligned in time” to the inputs of a system, but they are needed to be received very well controlled in time. We say usually, the signals are applied asynchronously but they must be received synchronously. For example, in Figure 2.16 the input of the register changes somehow chaotically related to the active edge of the clock, but the output
2.2. SEQUENTIAL CIRCUITS

Figure 2.16: Register at work. At each active edge of clock (in this example it is the positive edge) the register’s output takes the value applied on its inputs if reset = 0 and enable = 1.

of the register switches with a constant delay after the positive edge of clock. We say the inputs are synchronized to the output of the register. Their behavior is “time tempered”.

**Delaying** The input value applied in the clock cycle n to the input of a register is generated to the output of the register in the clock cycle n+1. In other words, the input of a register is delayed one clock cycle to its output. See in Figure 2.16 how the occurrence of a value in one clock cycle to the register’s input is followed in the next clock cycle by the occurrence of the same value to the register’s output.

**Looping** Structuring a digital system means to make different kind of connections. One of the most special, as we see in what follows, is a connection from some outputs to certain inputs in a digital subsystem. This kind of connections are called loops. The register is an important structural element in closing controllable loops inside a complex system.

2.2.7 **Shift register**

One of the simplest application of register is to perform shift operations. The numerical interpretation of a shift is the multiplication by the power of 2, for left shift, or division with the of 2, for right shift. A register used as shifter must be featured with four operation modes:

00: **nop** – no operation mode; it is mandatory for any set of function associated with a circuit (the circuit must be “able to stay doing nothing”)

01: **load** – the register’s state is initialized to the value applied on its inputs

10: **leftShift** – shift left with one binary position; if the register’s state is interpreted as a binary number, then the operation performed is a multiplication by 2

11: **rightShift** – shift right with one binary position; if the register’s state is interpreted as a binary number, then the operation performed is a division by 2
A synthesisable Verilog description of the circuit is:

```verilog
module shiftRegister(output reg [15:0] out ,
    input [15:0] in ,
    input [1:0] mode ,
    input clock);
always @(posedge clock)
    case(mode)
        2'b00: out <= out;
        2'b01: out <= in ;
        2'b10: out <= out << 1;
        2'b11: out <= out >> 1; // for positive integers
        //2'b11: out <= {out[15], out[15:1]}; // for signed integers
    endcase
endmodule
```

The case construct describes a 4-input multiplexor, \( MUX_4 \). Two versions are provided in the previous code, one for positive integer numbers and another for signed integers. The second is “commented”.

### 2.2.8 Counter

Let be the following circuit: its output is identical with its internal state, its state can take the value received on its data input, its internal state can be modified incrementing the number which represents its state or can stay unchanged. Let us call this circuit: presetable counter. Its Verilog behavioral description, for an 8-bit state, is:

```verilog
module counter( output reg [7:0] out ,
    input [7:0] in ,
    input init , // initialize with in
    input count , // increment state
    input clock);
always @(posedge clock) // always at the positive edge of clock
    if (init) out <= in;
    else if (count) out <= out + 1;
endmodule
```

The init input has priority to the input count, if it is active (\( init = 1 \)) the value of count is ignored and the value of state is initialized to in. If init in not activated, then if count = 1 then the value of counter is incremented modulo 256.

The actual structure of the circuit results (easy) from the previous Verilog description. Indeed, the structure

```verilog
if (init) ...
else ...
```

suggests a selector (a multiplexor), while

```verilog
out <= out + 1;
```

imposes an increment circuit. Thus, the schematic represented in Figure 2.17 pops up in our mind.

The circuit \( INC_8 \) in Figure 2.17 represents an increment circuit which outputs the input in incremented when the input inc_en (increment enable) is activated.
2.3 Putting all together

Now, going back to our first target enounced in section 1.3, let us put together what we learned about digital circuits in this section. The RTL code for the Digital Pixel Corrector circuit can be written now "more directly" as follows:

```verilog
module pixelCorrector #(`include "0_paramPixelCor.v")
    (output [m-1:0] out,
     input [n-1:0] in,
     input clock);

    reg [q-1:0] state; // the state register

    always @(posedge clock) state <= {state[7:0], in}; // state transition

    assign out = (state[7:4] == 0) ? ({1'b0, state[3:0]} + state[11:8]) >> 1 : state[7:4];

endmodule
```

The schematic we have in mind while writing the previous code is represented in Figure 2.18, where:

- the state register, state, has three sections of 4 bits each; in each cycle the positive edge of clock shifts left the content of state 4 binary positions, and in the freed locations loads the input value in
- the middle section is continuously tested, by the module Zero, if its value is zero
- the first and the last sections of the state are continuously added and the result is divided by 2 (shifted one position right) and applied to the input 1 of the selector circuit

Figure 2.17: The internal structure of a counter. If \( init = 1 \), then the value of the register is initialized to \( in \), else if \( count = 1 \) each active edge of clock loads in register its incremented value.
• the selector circuit, ifThenElse (the multiplexer), selects to the output, according to the test performed by the module Zero, the middle section of state or the shifted output of the adder.

![Diagram of the pixelCorrector circuit]

Figure 2.18: The structure of the pixelCorrector circuit.

Each received value is loaded first as state[3:0], then it moves in the next section. Thus, an input value comes in the position to be sent out only with a delay of two clock cycles. This two-cycle latency is imposed by the interpolation algorithm which must wait for the next input value to be loaded as a stable value.

2.4 Concluding about this short introduction in digital circuits

A digital circuit is build of combinational circuits and storage registers

Combinational logic can do both, control and arithmetic

Logic circuits, with appropriate loops, can memorize

HDL, as Verilog or VHDL, must be used to describe digital circuits

Growing, speeding and featuring digital circuits digital systems are obtained

2.5 Problems

Combinational circuits

Problem 2.1 Design the n-input Equal circuit which provides 1 on its output only when its two inputs, of n-bits each, are equal.
2.5. PROBLEMS

Problem 2.2 Design a 4-input selector. The selection code is sel[1:0] and the inputs are of 1 bit each: in3, in2, in1, in0.

Problem 2.3 Provide the proof for the following Boolean equivalence:

\[ a \cdot b + c \cdot (a + b) = a \cdot b + c \cdot (a \oplus b) \]

Problem 2.4 Provide the Verilog structural description of the adder module for \( n = 8 \). Synthesise the design and simulate it.

Problem 2.5 Draw the logic schematic for the XOR circuit.

Problem 2.6 Provide the proof that: \( a \oplus b = (a' \oplus b)' = (a \oplus b')' \).

Problem 2.7 Define the truth table for the one-bit subtractor and extract the two expressions describing the associated circuit.

Problem 2.8 Design the structural description of a \( n \)-bit subtractor. Synthesise and simulate it.

Problem 2.9 Provide the structural Verilog description of the adder/subtractor circuit behaviorally defined as follows:

```verilog
module addSub #(parameter n = 4)// defines a n-bit adder
(output [n-1:0] sum, // the n-bit result
     output carry, // carry output (borrow output for subtract)
     input sub, // sub=1 ? sub : add
     input c, // carry input (borrow input for subtract)
     input [n-1:0] a, b); // the two n-bit numbers

assign {carry, sum} = sub ? a - b - c : a + b + c;
endmodule
```

Simulate and synthesise the resulting design.

Flip-flops

Problem 2.10 Why, in Figure 2.6, we did not use a XOR gate to close a latching loop?

Problem 2.11 Design the structural description, in Verilog, for a NOR elementary latch. Simulate the circuit in order to determine the shortest signal which is able to provide a stable transition of the circuit. Try to use NOR gates with different propagation time.

Problem 2.12 When it is necessary to use a NOR elementary latch for a de-bouncing circuit?

Problem 2.13 Try to use the structural simulation of an elementary latch to see how behaves the circuit when the two inputs of the circuit are activated simultaneously (the second latch problem). If you are sure the simulation is correct, but it goes spooky, then go to office ours to discuss with your teacher.

Problem 2.14 What if the NAND gates from the circuit represented in Figure 2.9 are substituted with NOR gates?
Problem 2.15  Design and simulate structurally an elementary clocked latch, using only 4 gates, which is transparent on the level 0 of the clock signal.

Problem 2.16  Provide the test module for the module data_latch (see subsection 2.2.3) in order to verify the design.

Problem 2.17  Draw, at the gate level, the internal structure of a master-slave RS flip-flop using

- NAND gates and an inverter
- NOR gates and an inverter
- NAND and NOR gates, for two versions:
  - triggered by the positive edge of clock
  - triggered by the negative edge of clock.

Applications

Problem 2.18  Draw the block schematic for the circuit performing pixel correction according to the following interpolation rule:

\[ s'(t) = \frac{(2 \times s(t-2) + 6 \times s(t-1) + 6 \times s(t+1) + 6 \times s(t+2))}{16} \]

Using the schematic, write the Verilog code describing the circuit. Simulate and synthesise it.

Problem 2.19  Design a circuit which receives a stream of 8-bit numbers and sends, with a minimal latency, instead of each received number the mean value of the last three received numbers.

Problem 2.20  Design a circuit which receives a stream of 8-bit signed numbers and sends, with one clock cycle latency, instead of each received number its absolute value.

Problem 2.21  Draw the block schematic for the module shiftRegister, described in subsection 2.2.7, using a register and two input multiplexers, MUX2. Provide a Verilog structural description for the resulting circuit. Simulate and synthesise it.

Problem 2.22  Define a two-input DF-F using a DF-F and an EMUX. Use the new structure to describe structurally a presetable shift right register. Add the possibility to perform logic or arithmetic shift.

Problem 2.23  Write the structural description for the increment circuit INC_8 introduces in subsection 2.2.8.

Problem 2.24  Write the structural description for the module counter defined in subsection 2.2.8. Simulate and synthesise it.

Problem 2.25  Define and design a reversible counter able to count-up and count-down. Simulate and synthesise it.

Problem 2.26  Design the accumulator circuit able to add sequentially a clock synchronized sequence of up to 256 16-bit signed integers. The connections of the circuit are
2.5. PROBLEMS

module accumulator
  (output reg [?:0] acc , // the output register used to accumulate
   input [15:0] number, // the input receiving the stream of numbers
   input [1:0] com , // the command: 00=nop, 01=init, 10=accumulate
   input clock );

... 

endmodule

The init command initializes the state, by clearing the register, in order to start a new accumulation process.

**Problem 2.27** Design the two \( n \)-bit inputs combinational circuit which computes the absolute difference of two numbers.

**Problem 2.28** Define and design a circuit which receives a one-bit waveform and shows on its three one-bit outputs, by one clock cycle long positive impulses, the following events:

- any positive transition of the input signal
- any negative transition of the input signal
- any transition of the input signal.

**Problem 2.29** Design the combinational circuit which compute the absolute value of a signed number.
Chapter 3

GROWING & SPEEDING & FEATURING

In the previous chapter starting from simple algorithms small combinational and sequential circuits were designed, using the Verilog HDL as tool to describe and simulate. From the first chapter is good to remember:

- Verilog can be used for both behavioral (what does the circuit?) and structural (how looks the circuit?) descriptions
- the outputs of a combinational circuits follow continuously (with as small as possible delay) any change of its inputs, while a sequential one takes into account a shorter or a longer history of the input behavior
- the external time dependencies must be minimized if not avoided; each circuit must have its own and independent time behavior in order to allow global optimizations

In this chapter the three main mechanisms used to generate a digital system are introduced:

- composition: the mechanism allowing a digital circuit to increase its size and its computational power
- pipeline: is the way of interconnecting circuits to avoid the increase of the delays generated by too many serial compositions
- loop: is a kind of connection responsible for adding new type of behaviors, mainly by increasing the autonomy of the system

In the next chapter a taxonomy based on the number of loops closed inside a digital system is proposed. Each digital order, starting from 0, is characterized by the degree of the autonomy its behavior develops. While digital circuits are combinational or sequential, digital systems will be:

- 0 order, no-loop circuits (the combinational circuits)
- first order, 1-loop circuits (simple flip-flops, ...)
- second order, 2-loop circuits (finite automata, ...)
- third order, 3-loop circuits (processors, ...)
• ...
... there is no scientific theory about what can and can’t be built.

David Deutsch

Engineering only uses theories, but it is art.

In this section we talk about simple things which have multiple, sometime spectacular, followings. What can be more obvious than that a system is composed by many subsystems and some special behaviors are reached only using appropriate connections.

Starting from the ways of composing big and complex digital systems by appropriately interconnecting simple and small digital circuits, this book introduces a more detailed classification of digital systems. The new taxonomy classifies digital systems in orders, based on the maximum number of included loops closed inside each digital system. We start from the basic idea that a new loop closed in a digital system adds new functional features in it. By composition, the system grows only by forwarded connections, but by appropriately closed backward connections it gains new functional capabilities. Therefore, we will be able to define many functional levels, starting with time independent combinational functions and continuing with memory functions, sequencing functions, control functions and interpreting functions. Basically, each new loop manifests itself by increasing the degree of autonomy of the system.

Therefore, the main goal of this section is to emphasize the fundamental developing mechanisms in digital systems which consist in compositions & loops by which digital systems gain in size and in functional complexity.

In order to better understand the correlation between functional aspects and structural aspect in digital systems we need a suggestive image about how these systems grow in size and how they gain new functional capabilities. The oldest distinction between combinational circuits and sequential circuits is now obsolete because of the diversity of circuits and the diversity of their applications. In this section we present a new idea about a mechanism which emphasizes a hierarchy in the world of digital system. This world will be hierarchically organized in orders counted from 0 to n. At each new level a functional gain is obtained as a consequence of the increased autonomy of the system.

Two are the mechanisms involved in the process of building digital systems. The first allows of system to grow in size. It is the composition, which help us to put together, using only forward connections, many subsystems in order to have a bigger system. The second mechanism is a special connection that provides new functional features. It is the loop connection, simply the loop. Where a new loop is closed, a new kind of behavior is expected. To behave means, mainly, to have autonomy. If a system use a part of own outputs to drive some of its inputs, then “he drives himself” and an outsider receives this fact as an autonomous process.

Let us present in a systematic way, in the following subsections, the two mechanisms. Both are very simple, but our goal is to emphasize, in the same time, some specific side effects as consequences of composing & looping, like the pipeline connection – used to accelerate the speed of the too deep circuits – or the speculative mechanisms – used to allow loops to be closed in pipelined structures.

Building a real circuit means mainly to interconnect simple and small components in order to grow an enough fast system appropriately featured. But, growing is a concept with no precise meaning. Many people do not make distinction between “growing the size” and “growing the complexity” of a system,

1David Deutsch’s work on quantum computation laid the foundation for that field, grounding new approaches in both physics and the theory of computation. He is the author of The Fabric of Reality.
for example. We will start making the necessary distinctions between “size” and “complexity” in the
process of growing a digital system.

## 3.1 Size vs. Complexity

The huge size of the actual circuits implemented on a single chip imposes a more precise distinction
between simple circuits and complex circuits. When we can integrated on a single chip more than $10^9$
components, the size of the circuits becomes less important than their complexity. Unfortunately we
don’t make a clear distinction between size and complexity. We say usually: “the complexity of a com-
putation is given by the size of memory and by the CPU time”. But, if we have to design a circuit of
100 million transistors it is very important to distinguish between a circuit having an uniform structure
and a randomly structured ones. In the first case the circuit can be easy specified, easy described in an
HDL, easy tested and so on. Otherwise, if the structure is completely random, without any repetitive
substructure inside, it can be described using only a description having a similar dimension with the
circuit size. When the circuit is small, it is not a problem, but for million of components the problem
has no solution. Therefore, if the circuit is very big, it is not enough to deal only with its size, the most
important becomes also the degree of uniformity of the circuit. This degree of uniformity, the degree of
order inside the circuit can be specified by its complexity.

As a consequence we must distinguish more carefully the concept of size by the concept of complexity.
Follow the definitions of these terms with the meanings we will use in this book.

**Definition 3.1** The size of a digital circuit, $S_{\text{digital circuit}}$, is given by the dimension of the physical re-
sources used to implement it. *

In order to provide a numerical expression for size we need a more detailed definition which takes
into account technological aspects. In the ’40s we counted electronic bulbs, in the ’50s we counted
transistors, in the ’60s we counted SSI$^2$ and MSI$^3$ packages. In the ’70s we started to use two measures:
sometimes the number of transistors or the number of 2-input gates on the Silicon die and other times
the Silicon die area. Thus, we propose two numerical measures for the size.

**Definition 3.2** The gate size of a digital circuit, $GS_{\text{digital circuit}}$, is given by the total number of CMOS
pairs of transistors used for building the gates (see the appendix Basic circuits) used to implement it$^4$. *

This definition of size offers an almost accurate image about the Silicon area used to implement the
circuit, but the effects of lay-out, of fan-out and of speed are not catched by this definition.

**Definition 3.3** The area size of a digital circuit, $AS_{\text{digital circuit}}$, is given by the dimension of the area on
Silicon used to implement it. *

The area size is useful to compute the price of the implementation because when a circuit is produced
we pay for the number of wafers. If the circuit has a big area, the number of the circuits per wafer is
small and the yield is low$^5$.

$^2$Small Size Integrated circuits  
$^3$Medium Size Integrated circuits  
$^4$Sometimes gate size is expressed in the total number of 2-input gates necessary to implement the circuit. We prefer to
count CMOS pairs of transistors (almost identical with the number of inputs) instead of equivalent 2-input gates because is
simplest. Anyway, both ways are partially inaccurate because, for various reasons, the transistors used in implementing a gate
have different areas.  
$^5$The same number of errors make useless a bigger area of the wafer containing large circuits.
3.1. SIZE VS. COMPLEXITY

Definition 3.4 The algorithmic complexity of a digital circuit, simply the complexity, $C_{\text{digital circuit}}$, has the magnitude order given by the minimal number of symbols needed to express its definition. ⊖

Definition 2.2 is inspired by Gregory Chaitin’s definition for the algorithmic complexity of a string of symbols [Chaitin ’77]. The algorithmic complexity of a string is related to the dimension of the smallest program that generates it. The program is interpreted by a machine (more in Chapter 12). Our $C_{\text{digital circuit}}$ can be associated to the shortest unambiguous circuit description in a certain HDL (in the most of cases it is about a behavioral description).

Definition 3.5 A simple circuit is a circuit having the complexity much smaller than its size:

$$C_{\text{simple circuit}} << S_{\text{simple circuit}}.$$  

Usually the complexity of a simple circuit is constant: $C_{\text{simple circuit}} \in O(1)$. ⊖

Definition 3.6 A complex circuit is a circuit having the complexity in the same magnitude order with its size:

$$C_{\text{complex circuit}} \sim S_{\text{complex circuit}}.$$ ⊖

Example 3.1 The following Verilog program describes a complex circuit, because the size of its definition (the program) is

$$S_{\text{def. of random circ}} = k_1 + k_2 \times S_{\text{random circ}} \in O(S_{\text{random circ}}).$$

```
module random_circ(output f, g, input a, b, c, d, e);
    wire w1, w2;
    and and1(w1, a, b),
        and2(w2, c, d);
    or  or1(f, w1, c),
        or2(g, e, w2);
endmodule
```

Example 3.2 The following Verilog program describes a simple circuit, because the program that define completely the circuit is the same for any value of n.

```
module or_prefixes #(parameter n = 256)(output reg [0:n-1] out, input [0:n-1] in);
    integer k;
    always @(in) begin out[0] = in[0];
        for (k=1; k<n; k=k+1) out[k] = in[k] | out[k-1];
    end
endmodule
```

The prefixes of OR circuit consists in n OR2 gates connected in a very regular form. The definition is the same for any value of n.6 ⊖

Composing circuits generate not only biggest structures, but also deepest ones. The depth of the circuit is related with the associated propagation time.

---

6A short discussion occurs when the dimension of the input is specified. To be extremely rigorous, the parameter $n$ is expressed using a string o symbols in $O(\log n)$. But usually this aspect can be ignored.
Definition 3.7 The depth of a combinational circuit is equal with the total number of serially connected constant input gates (usually 2-input gates) on the longest path from inputs to the outputs of the circuit.

The previous definition offers also only an approximate image about the propagation time through a combinational circuit. Inspecting the parameters of the gates listed in Appendix Standard cell libraries you will see more complex dependence contributing to the delay introduced by a certain circuit. Also, the contribution of the interconnecting wires must be considered when the actual propagation time in a combinational circuit is evaluated.

Some digital functions can be described starting from the elementary circuit which performs them, adding a recursive rule for building a circuit that executes the same function for any size of the input. For the rest of the circuits, which don’t have such type of definitions, we must use a definition that describes in detail the entire circuit. This description will be non-recursive and thus complex, because its dimension is proportional with the size of circuit (each part of the circuit must be explicitly specified in this kind of definition). We shall call random circuit a complex circuit, because there is no (simple) rule for describing it.

The first type of circuits, having recursive definitions, are simple circuits. Indeed, the elementary circuit has a constant (usually small) size and the recursive rule can be expressed using a constant number of signs (symbolic expressions or drawings). Therefore, the dimension of the definition remains constant, independent by \( n \), for this kind of circuits. In this book, this distinction, between simple and complex, will be exemplified and will be used to promote useful distinctions between different solutions.

At the current technological level the size becomes less important than the complexity, because we can produce circuits having an increasing number of components, but we can describe only circuits having the range of complexity limited by our mental capacity to deal efficiently with complex representations. The first step to have a circuit is to express what it must do in a behavioral description written in a certain HDL. If this "definition" is too large, having the magnitude order of a huge multi-billion-transistor circuit, we don’t have the possibility to write the program expressing our desire.

In the domain of circuit design we passed long ago beyond the stage of minimizing the number of gates in a few gates circuit. Now, the most important thing, in the multi-billion-transistor circuit era, is the ability to describe, by recursive definitions, simple (because we can’t write huge programs), big (because we can produce more circuits on the same area) sized circuits. We must take into consideration that the Moore’s Law applies to size not to complexity.

3.2 Time restrictions in digital systems

The most general form of a digital circuit (see Figure 3.1) includes both combinational and sequential behaviors. It includes two combinational circuits – (comb_circ_1 and comb_circ_2) – and register. There are four critical propagation paths in this digital circuit:

1. form input to register through comb_circ_1, which determines minimum input arrival time before clock: \( t_{\text{in-reg}} \)

2. from register to register through comb_circ_1, which determines minimum period of clock: \( t_{\text{reg-reg}} = T_{\text{min}} \), or maximum frequency of clock: \( f_{\text{max}} = 1/T \)

3. from input to output through comb_circ_2, which determines maximum combinational path delay: \( t_{\text{in-out}} \)
3.2. Time Restrictions in Digital Systems

Figure 3.1: The four critical propagation paths in digital circuits. Input-to-register time \( t_{in\_reg} \) is recommended to be as small as possible in order to reduce the time dependency from the previous sub-system. Register-to-register time \( T_{\text{min}} \) must be minimal to allow a high frequency for the clock signal. Input-to-output time \( t_{in\_out} \) is good to be undefined to avoid hard to manage sub-systems interdependencies. Register-to-output time \( t_{reg\_out} \) must be minimal to reduce the time dependency for the next sub-system.

4. from register to output through \texttt{comb\_circ\_2}, which determines maximum output required time after clock: \( t_{reg\_out} \).

If the active transition of clock takes place at \( t_0 \) and the input signal changes after \( t_0 - t_{in\_reg} \), then the effect of the input change will be not registered correctly at \( t_0 \) in register. The input must be stable in the time interval from \( t_0 - t_{in\_reg} \) to \( t_0 \) in order to have a predictable behavior of the circuit.

The loop is properly closed only if \( T_{\text{min}} > t_{reg} + t_{cc2} + t_{su} \) and \( t_h < t_{reg} + t_{cc2} \), where: \( t_{reg} \) is the propagation time through register from active edge of clock to output, and \( t_{cc2} \) is the propagation time through \texttt{comb\_circ\_1} on the path 2.

If the system works with the same clock, then \( t_{in\_out} < T_{\text{min}} \), preferably \( t_{in\_out} << T_{\text{min}} \). Similar conditions are imposed for \( t_{in\_reg} \) and \( t_{reg\_out} \), because we suppose there are additional combinational delays in the circuits connected to the inputs and to the outputs of this circuit, or at least a propagation time through a register or set-up time to the input of a register.

**Example 3.3** Let us compute the propagation times for the four critical propagation paths of the counter.
circuit represented in Figure 2.17. If we consider \( \#1 = 100 \text{ps} \) results:

- \( t_{\text{in-reg}} = t_p(\text{mux2-8}) = 0.1 \text{ns} \)
  (the set-up time for the register is considered too small to be considered)
- \( f_{\text{max}} = 1/T = 1/(t_p(\text{reg}) + t_p(\text{inc}) + t_p(\text{mux2-8})) = 1/(0.2 + 0.1 + 0.1) \text{ns} = 2.5 \text{ GHz} \)
- \( t_{\text{in-reg}} \) is not defined
- \( t_{\text{reg-out}} = t_p(\text{reg}) = 0.2 \text{ns} \)

**Example 3.4** Let be the circuit from Figure 3.2, where:

- **register** is characterized by: \( t_p(\text{register}) = 150 \text{ps}, t_{\text{su}}(\text{register}) = 35 \text{ps}, t_h = 27 \text{ps} \)
- **adder** with \( t_p(\text{adder}) = 550 \text{ps} \)
- **selector** with \( t_p(\text{selector}) = 85 \text{ps} \)
- **comparator** with \( t_p(\text{comparator}) = 300 \text{ps} \)

The circuit is used to accumulate a stream of numbers applied on the input **data**, and to compare it against a threshold applied on the input **thr**. The accumulation process is initialized by the signal **reset**, and is controlled by the signal **acc**.

![Accumulate & compare circuit](image)

Figure 3.2: **Accumulate & compare circuit.** In the left-down corner of each rectangle is written the propagation time of each module. If \( \text{acc} = 1 \) the circuit accumulates, else the content of **register** does not change.

The propagation time for the four critical propagation path of this circuit are:
3.2. TIME RESTRICTIONS IN DIGITAL SYSTEMS

- \( t_{\text{in-reg}} = t_p(\text{adder}) + t_p(\text{selector}) + t_s(\text{register}) = (550 + 85 + 35)ps = 670ps \)
- \( f_{\text{max}} = 1/T = 1/(t_p(\text{register}) + t_p(\text{adder}) + t_p(\text{selector}) + t_s(\text{register})) = 1/(150 + 550 + 85 + 35)ps = 1.21GHz \)
- \( t_{\text{in-out}} = t_p(\text{comparator}) = 300ps \)
- \( t_{\text{reg-out}} = t_p(\text{register}) + t_p(\text{comparator}) = 450ps \)

While at the level of small and simple circuits no additional restriction are imposed, for complex digital systems there are mandatory rules to be followed for an accurate design. Two main restrictions occur:

1. the combinational path through the entire system must be completely avoided,
2. the combinational, usually called asynchronous, input and output path must be avoided as much as possible if not completely omitted.

Combinational paths belonging to distinct modules are thus avoided. The main advantage is given by the fact that design restrictions imposed in one module do not affect time restriction imposed in another module. There are two ways to consider these restrictions, a weak one and a strong one. The first refers to the pipeline connections, while the second to the fully buffered connections.

3.2.1 Pipelined connections

For the pipelined connection between two complex modules the timing restrictions are the following:

1. from input to output through: it is not defined
2. from register to output through: \( t_{\text{reg-out}} = t_{\text{reg}} – \text{it does not depend by the internal combinational structure of the module, i.e., the outputs are synchronous, because they are generated directly from registers.} \)

Only two combinational paths are accepted: (1) from register to register, and (2) form input to register. In Figure 3.3 a generic configuration is presented. It is about two systems, sys1 and sys2, pipeline connected using the output pipeline registers (pr1 between sys1 and sys2, and pr2 between sys2 and an external system). For the internal state are used the state registers sr1 and sr2. The timing restrictions for the two combinational circuits comb1 and comb2 are not correlated. The maximum clock speed for each system does not depend by the design restrictions imposed for the other system.

The pipeline connection works well only if the two systems are interconnected with short wires, i.e., the two systems are implemented on adjacent areas on the silicon die. No additional time must be considered on connections because they a very short.

The system from Figure 3.3 is descried by the following code.

```verilog
module pipelineConnection( output [15:0] out ,
                          input [15:0] in ,
                          input clock );
wire [15:0] pipeConnect ;
sys sys1( .pr (pipeConnect),
```

```verilog```

```verilog```

```verilog```

```verilog```

```verilog```

```verilog```

```verilog```

```verilog```

```verilog```

```verilog```

```verilog```

```verilog```

```verilog```

```verilog```
Something very important is introduced by the last Verilog code: the distinction between blocking and non-blocking assignment:

- the **blocking assignment**, =: the whole statement is done before control passes to the next
3.2. TIME RESTRICTIONS IN DIGITAL SYSTEMS

- the **non-blocking assignment**, $<=$: evaluate all the right-hand sides in the project for the current time unit and assign the left-hand sides only at the end of the time unit.

Let us use the following simulation to explain the very important difference between the two kinds of assignment.

**VeriSim 3.1** The following simulation used 6 clocked registers. All of them switch on the positive edge. But, the code is written for three of them using the **blocking assignment**, while for the other three using the **non-blocking assignment**. The resulting behavior show us the difference between the two clock triggered assignment. The blocking assignment seems to be useless, because propagates the input through all the three registers in one clock cycle. The non-blocking assignment shifts the input along the three serially connected registers clock by clock. This second behavior can be used in real application to obtain clock controlled delays.

```verilog
module blockingNonBlocking(output reg [1:0] blockingOut ,
                           output reg [1:0] nonBlockingOut ,
                           input [1:0] in ,
                           input clock);
    reg [1:0] reg1, reg2, reg3, reg4;
    always @(posedge clock) begin reg1 = in ;
        reg2 = reg1 ;
        blockingOut = reg2 ; end
    always @(posedge clock) begin reg3 <= in ;
        reg4 <= reg3 ;
        nonBlockingOut <= reg4 ; end
endmodule

module blockingNonBlockingSimulation;
    reg clock ;
    reg [1:0] in ;
    wire [1:0] blockingOut, nonBlockingOut ;
    initial begin clock = 0 ;
        forever #1 clock =~clock ; end
    initial begin in = 2'b01 ;
        #2 in = 2'b10 ;
        #2 in = 2'b11 ;
        #2 in = 2'b00 ;
        #7 $stop ; end
    blockingNonBlocking dut(blockingOut ,
                           nonBlockingOut ,
                           in ,
                           clock );
endmodule
/*

module blockingNonBlocking(output reg [1:0] blockingOut ,
                           output reg [1:0] nonBlockingOut ,
                           input [1:0] in ,
                           input clock);
    reg [1:0] reg1, reg2, reg3, reg4;
    always @(posedge clock) begin reg1 = in ;
        reg2 = reg1 ;
        blockingOut = reg2 ; end
    always @(posedge clock) begin reg3 <= in ;
        reg4 <= reg3 ;
        nonBlockingOut <= reg4 ; end
endmodule

module blockingNonBlockingSimulation;
    reg clock ;
    reg [1:0] in ;
    wire [1:0] blockingOut, nonBlockingOut ;
    initial begin clock = 0 ;
        forever #1 clock =~clock ; end
    initial begin in = 2'b01 ;
        #2 in = 2'b10 ;
        #2 in = 2'b11 ;
        #2 in = 2'b00 ;
        #7 $stop ; end
    blockingNonBlocking dut(blockingOut ,
                           nonBlockingOut ,
                           in ,
                           clock );
endmodule
/*
It is obvious that the registers reg1 and reg2 are useless because they are somehow “transparent”.

The non-blocking version of assigning the content of a register will provide a clock controlled delay. Anytime in a design there are more than one registers the non-blocking assignment must be used.

VerilogSummary 1 :

= : **blocking assignment** the whole statement is done before control passes to the next

<= : **non-blocking assignment** evaluate all the right-hand sides in the project for the current time unit and assign the left-hand sides only at the end of the time unit.

### 3.2.2 Fully buffered connections

The most safe approach, the synchronous one, supposes fully registered inputs and outputs (see Figure 3.4 where the functionality is implemented using combinatorial circuits and registers and the interface with the rest of the system is implemented using only input register and output register).

The modular synchronous design of a big and complex system is the best approach for a robust design, and the maximum modularity is achieved removing all possible time dependency between the modules. Then, take care about the module partitioning in a complex system design!

Two fully buffered modules can be placed on the silicon die with less restrictions, because even if the resulting wires are long the signals have time to propagate because no gates are connected between the output register of the sender system and the input register of the receiver system.

For the synchronously interfaced module represented in Figure 3.4 the timing restrictions are the following:

1. form input to register: $t_{in_{-}reg} = t_{iu} - it
does
not
depend
by
the
internal
structure
of
the
module$

2. from register to register: $T_{min}, and f_{max} = 1/T - it
is
a
system
parameter$

3. from input to output through: **it is not defined**
3.3. GROWING THE SIZE BY COMPOSITION

The mechanism of composition is well known to everyone who worked at least a little in mathematics. We use forms like:

$$f(x) = g(h_1(x), \ldots, h_m(x))$$

to express the fact that computing the function $f$ requests to compute first all the functions $h_i(x)$ and after that the $m$-variable function $g$. We say: the function $g$ is composed with the functions $h_i$ in order to have computed the function $f$. In the domain digital systems a similar formalism is used to “compose” big circuits from many smaller ones. We will define the composition mechanism in digital domain using a Verilog-like formalism.

**Definition 3.8** The composition (see Figure 3.5) is a two level construct, which performs the function $f$ using on the second level the $m$-ary function $g$ and on the first level the functions $h_1, h_2, \ldots, h_m$, described by the following, incompletely defined, but synthesisable, Verilog modules.
Figure 3.5: **The circuit performing composition.** The function \( g \) is composed with the functions \( h_1, \ldots, h_m \) using a two level circuit. The first level contains \( m \) circuits computing \textit{in parallel} the functions \( h_i \), and on the second level there is the circuit computing the \textit{reduction}-like function \( g \).

```verilog
module f #(include "parameters.v")(output [sizeOut-1:0] out, input [sizeIn-1:0] in);
    wire [size_1-1:0] out_1;
    wire [size_2-1:0] out_2;
    // ...
    wire [size_m-1:0] out_g
    g second_level( .out (out ),
                       .in_1 (out_1 ),
                       .in_2 (out_2 ),
                        // ...
                       .in_m (out_m ));
    h_1 first_level_1(.out(out_1), .in(in));
    h_2 first_level_2(.out(out_2), .in(in));
    // ...
    h_m first_level_m(.out(out_m), .in(in));
endmodule

module g #(include "parameters.v")(output [sizeOut-1:0] out, input [size_1-1:0] in_1,
                                                     input [size_2-1:0] in_2,
                                                     // ...
                                                     input [size_m-1:0] in_m);
    // ...
endmodule

module h_1 #(include "parameters.v")(output [size_1-1:0] out ,
                                                     input [sizeIn-1:0] in );
    // ...
endmodule

module h_2 #(include "parameters.v")(output [size_2-1:0] out ,
                                                     input [sizeIn-1:0] in );
```
The content of the file parameters.v is:

```verbatim
parameter sizeOut = 32,
    sizeIn = 8,
    size_1 = 12,
    size_2 = 16,
    // ...
    size_m = 8
```

The general form of the composition, previously defined, can be called the *serial-parallel* composition, because the modules $h_1, \ldots, h_m$ compute in parallel $m$ functions, and all are serial connected with the module $g$ (we can call it *reduction type function*, because it reduces the vector generated by the previous level to a value). There are two limit cases. One is the *serial composition* and another is the *parallel composition*. Both are structurally trivial, but represent essential limit aspects regarding the resources of parallelism in a digital system.

**Definition 3.9** The *serial composition* (see Figure 3.6a) is the composition with $m = 1$. Results the Verilog description:

```verbatim
module f #(`include "parameters.v")( output [sizeOut-1:0] out ,
    input [sizeIn-1:0] in );
    wire [size_1-1:0] out_1;
    g second_level( .out (out ),
        .in_1 (out_1 ));
    h_1 first_level_1(.out(out_1), .in(in));
endmodule

module g #(`include "parameters.v")( output [sizeOut-1:0] out ,
    input [size_1-1:0] in_1);
    // ...
endmodule

module h_1 #(`include "parameters.v")( output [size_1-1:0] out ,
    input [sizeIn-1:0] in );
    // ...
endmodule
```

⋄
CHAPTER 3. GROWING & SPEEDING & FEATURING

Figure 3.6: The two limit forms of composition. a. The serial composition, for \( m = 1 \), imposing an inherent sequential computation. b. The parallel composition, with no reduction-like function, performing data parallel computation.

Definition 3.10 The parallel composition (see Figure 3.6b) is the composition in the particular case when \( g \) is the identity function. Results the following Verilog description:

```verilog
module f #(`include "parameters.v")( output [sizeOut-1:0] out ,
    input [sizeIn-1:0] in );
    wire [size_1-1:0] out_1;
    wire [size_2-1:0] out_2;
    // ...
    wire [size_m-1:0] out_m;
    assign out = { out_m,
        // ...
        out_2,
        out_1}; // g is identity function

    h_1 first_level_1(.out(out_1), .in(in));
    h_2 first_level_2(.out(out_2), .in(in));
    // ...
    h_m first_level_m(.out(out_m), .in(in));
endmodule

module h_1 #(`include "parameters.v")( output [size_1-1:0] out ,
    input [sizeIn-1:0] in );
    // ...
endmodule

module h_2 #(`include "parameters.v")( output [size_2-1:0] out ,
    input [sizeIn-1:0] in );
    // ...
endmodule

// ...

module h_m #(`include "parameters.v")( output [size_m-1:0] out ,
    input [sizeIn-1:0] in );
    // ...
```


3.3. GROWING THE SIZE BY COMPOSITION

The content of the file parameters.v is now:

```verilog
parameter sizeIn = 8 ,
size_1 = 12 ,
size_2 = 16 ,
// ...
size_m = 8 ,
sizeOut = size_1 +
size_2 +
// ...
size_m
```

Example 3.5 Using the mechanism described in Definition 1.3 the circuit computing the scalar product between two 4-component vectors will be defined, now in true Verilog. The test module for n = 8 is also defined allowing to test the design.

```verilog
module inner_prod #(`include "parameter.v")
    (output [((2*n+2)-1):0] out, input [n-1:0] a3, a2, a1, a0, b3, b2, b1, b0);
wire[2*n-1:0] p3, p2, p1, p0;
mult m3(p3, a3, b3),
m2(p2, a2, b2),
m1(p1, a1, b1),
m0(p0, a0, b0);
add4 add(out, p3, p2, p1, p0);
endmodule

module mult #(`include "parameter.v")
    (output [((2*n-1):0] out, input [n-1:0] m1, m0);
    assign out = m1 * m0;
endmodule

module add4 #(`include "parameter.v")
    (output [((2*n+2)-1):0] out, input [(2*n-1):0] t3, t2, t1, t0);
    assign out = t3 + t2 + t1 + t0;
endmodule
```

The content of the file parameter.v is:

```verilog
parameter n = 8
```

The simulation is done by running the module:

```verilog
module test_inner_prod;
reg[7:0] a3, a2, a1, a0, b3, b2, b1, b0;
wire[17:0] out;

initial begin
    {a3, a2, a1, a0} = {8'd1, 8'd2, 8'd3, 8'd4};
    {b3, b2, b1, b0} = {8'd5, 8'd6, 8'd7, 8'd8};
endmodule
```
CHAPTER 3. GROWING & SPEEDING & FEATURING

Figure 3.7: An example of composition. The circuit performs the scalar vector product for 4-element vectors. The first level compute in parallel 4 multiplications generating the vectorial product, and the second level reduces the resulting vector of products to a scalar.

```verilog
inner_prod dut(out, a3, a2, a1, a0, b3, b2, b1, b0);
initial $monitor("out=%0d", out);
endmodule
```

The test outputs: out = 70

The description is structural at the top level and behavioral for the internal sub-modules (corresponding to our level of understanding digital systems). The resulting circuit is represented in Figure 3.7.

**VerilogSummary 2:**

- The directive `include` is used to add in any place inside a module the content of the file `xxx.v` writing: `include "xxx.v"
- We just learned how to concatenate many variables to obtain a bigger one (in the definition of the parallel composition the output of the system results as a concatenation of the outputs of the sub-systems it contains)
- Is good to know there is also a risky way to specify the connections when a module is instantiated into another: to put the name of connections in the appropriate positions in the connection list (in the last example)

By composition we add new modules in the system, but we don’t change the class to which the system belongs. The system gains the behaviors of the added modules but nothing more. By composition we sum behaviors only, but we can not introduce in this way a new kind of behavior in the world of digital machines. What we can’t do using new modules we can do with an appropriate connection: the loop.
3.4 Speeding by pipelining

One of the main limitations in applying the composition is due to the increased propagation time associated to the serially connected circuits. Indeed, the time for computing the function $f$ is:

$$t_f = \max(t_{h,1}, \ldots, t_{h,m}) + t_g$$

In the last example, the inner product is computed in:

$$t_{\text{inner product}} = t_{\text{multiplication}} + t_{\text{4-number add}}$$

If the 4-number add is also composed using 2-number add (as in usual systems) results:

$$t_{\text{inner product}} = t_{\text{multiplication}} + 2 \times t_{\text{addition}}$$

For the general case of $n$-components vectors the inner product will be computed, using a similar approach, in:

$$t_{\text{inner product}}(n) = t_{\text{multiplication}} + t_{\text{addition}} \times \log_2 n \in O(\log n)$$

For this simple example, of computing the inner product of two vectors, results for $n \geq n_0$ a computational time bigger than can be accepted in some applications. Having enough multipliers, the multiplication will not limit the speed of computation, but even if we have infinite 2-input adders the computing time will remain dependent by $n$.

The typical case is given by the serial composition (see Figure 3.6a), where the function $out = f(in) = g(h_1(in))$ must be computed using 2 serial connected circuits, $h_1(in)$ and $g(\text{int_out})$, in time:

$$t_f = t_{h_1} + t_g.$$

A solution must be found to deal with the too deep circuits resulting from composing to many or to "lazy" circuits.

First of all we must state that fast circuits are needed only when a lot of data is waiting to be computed. If the function $f(in)$ is rarely computed, then we do not care to much about the speed of the associated circuit. But, if there is an application supposing a huge stream of data to be successively submitted to the input of the circuit $f$, then it is very important to design a fast version of it.

Golden rule: only what is frequently computed must be accelerated!

3.4.1 Register transfer level

The good practice in a digital system is: any stream of data is received synchronously and it is sent out synchronously. Any digital system can be reduced to a synchronous machine receiving a stream of input data and generating another stream of output results. As we already stated, a “robust” digital design is a fully buffered one because it provides a system interfaced to the external “world” with registers.

The general structure of a system performing the function $f(x)$ is shown in Figure 3.8a, where it is presented in the fully buffered version. This kind of approach is called register transfer level (RTL) because data is transferred, modified by the function $f$, from a register, input_reg, to another register, output_reg. If $f = g(h_1(x))$, then the clock frequency is limited to:

$$f_{\text{clock max}} = \frac{1}{t_{\text{reg}} + t_f + t_{\text{su}}} = \frac{1}{t_{\text{reg}} + t_{h_1} + t_g + t_{\text{su}}}$$

The serial connection of the module computing $h_1$ and $g$ is a fundamental limit. If $f$ computation is not critical for the system including the module $f$, then this solution is very good, else you must read and assimilate the next, very important, paragraph.
3.4.2 Pipeline structures

To increase the processing speed of a long stream of data the clock frequency must be increased. If the stream has the length \( n \), then the processing time is:

\[
T_{\text{stream}}(n) = \frac{1}{f_{\text{clock}}} \times (n + 2) = (t_{\text{reg}} + t_{h_1} + t_g + t_{su}) \times (n + 2)
\]

![Diagram of pipeline computation](image)

**Figure 3.8: Pipelined computation.**

a. A typical Register Transfer Logic (RTL) configuration. Usually it is supposed a “deep” combinational circuit computes \( f(x) \).

b. The pipeline structure splits the combinational circuit associated with function \( f(x) \) in two less “deep” circuits and inserts the pipeline register in between.

The only way to increase the clock rate is to divide the circuit designed for \( f \) in two serially connected circuits, one for \( h_1 \) and another for \( g \), and to introduce between them a new register. Results the system represented in Figure 3.8b. Its clock frequency is:

\[
f_{\text{clock, max}} = \frac{1}{\max(t_{h_1}, t_g) + t_{\text{reg}} + t_{\text{su}}}
\]

and the processing time for the same string is:

\[
T_{\text{stream}}(n) = (\max(t_{h_1}, t_g) + t_{\text{reg}} + t_{\text{su}}) \times (n + 3)
\]

The two systems represented in Figure 3.8 are equivalent. The only difference between them is that the second performs the processing in \( n + 3 \) clock cycles instead of \( n + 2 \) clock cycles for the first version. For big \( n \), the current case, this difference is a negligible quantity. We call latency the number of the additional clock cycle. In this first example latency is: \( \lambda = 1 \).

This procedure can be applied many times, resulting a processing “pipe” with a latency equal with the number of the inserted register added to the initial system. The resulting system is called a pipelined system. The additional registers are called pipeline registers.
3.4. SPEEDING BY PIPELINING

The maximum efficiency of a pipeline system is obtained in the ideal case when, for an \((m+1)\)-stage pipeline, realized inserting \(m\) pipeline registers:

\[
\max(t_{\text{stage}_0}, t_{\text{stage}_1}, \ldots, t_{\text{stage}_m}) = \frac{t_{\text{stage}_0} + t_{\text{stage}_1} + \ldots + t_{\text{stage}_m}}{m+1}
\]

\[
\max(t_{\text{stage}_0}, t_{\text{stage}_1}, \ldots, t_{\text{stage}_m}) > t_{\text{reg}} + t_{\text{su}}
\]

\[
\lambda = m << n
\]

In this ideal case the speed is increased almost \(m\) times. Obviously, no one of these conditions can be fully accomplished, but there are a lot of real applications in which adding an appropriate number of pipeline stages allows to reach the desired speed performance.

**Example 3.6** The pseudo-Verilog code for the 2-stage pipeline system represented in Figure 3.8 is:

```verilog
module pipelined_f( output reg [size_in-1:0] sync_out,
input [size_out-1:0] in);
reg [size_in-1:0] sync_in;
wire [size_int-1:0] int_out,
reg [size_int-1:0] sync_int_out;
wire [size_out-1:0] out;
h_1 this_h_1( .out (int_out),
.in (sync_in));
g this_g( .out (out),
.in (sync_int_out));
always @(posedge clock) begin sync_in <= #2 in;
sync_int_out <= #2 int_out;
sync_out <= #2 out;
end
endmodule

module h_1( output [size_int-1:0] out,
input [size_in-1:0] in);
assign #15 out = \ldots;
endmodule

module g( output [size_out-1:0] out,
input [size_int-1:0] in);
assign #18 out = \ldots;
endmodule
```

Suppose, the unit time is \(1\) ns. The maximum clock frequency for the pipeline version is:

\[
f_{\text{clock}} = \frac{1}{\max(15, 18)+2} \text{GHz} = 50 \text{MHz}
\]

This value must be compared with the frequency of the non-pipelined version, which is:

\[
f_{\text{clock}} = \frac{1}{15+18+2} \text{GHz} = 28.57 \text{MHz}
\]

Adding only a simple register and accepting a minimal latency \((\lambda = 1)\), the speed of the system increased with 75%. ◦
3.4.3 Data parallelism vs. time parallelism

The two limit cases of composition correspond to the two extreme cases of parallelism in digital systems:

- the serial composition will allow the pipeline mechanism which is a sort of parallelism which could be called diachronic parallelism or time parallelism
- the parallel composition is an obvious form of parallelism, which could be called synchronic parallelism or data parallelism.

The data parallelism is more obvious: \( m \) functions, \( h_1, \ldots, h_m \), are performed in parallel by \( m \) circuits (see Figure 3.6b). But, time parallelism is not so obvious. It acts only in a pipelined serial composition, where the first stage is involved in computing the most recently received data, the second stage is involved in computing the previously received data, and so on. In an \((m+1)\)-stage pipeline structure \( m+1 \) elements of the input stream are in different stages of computation, and at each clock cycle one result is provided. We can claim that in such a pipeline structure \( m+1 \) computations are done in parallel with the price of a latency \( \lambda = m \).

![Diagram of pipelined inner product circuit for 4-component vectors](image)

Figure 3.9: The pipelined inner product circuit for 4-component vectors. Each multiplier and each adder send its result in a pipeline register. For this application results a three level pipeline structure with different degree of parallelism. The two kind of parallelism are exemplified. Data parallel has the maximum degree on the first level. The degree of time parallelism is three: in each clock cycle three pairs of 4-element vectors are processed. One pair in the first stage of multiplications, another pair is the second stage of performing two additions, and one in the final stage of making the last addition.

The previous example of a 2-stage pipeline accelerated the computation because of the time parallelism which allows to work simultaneously on two input data, on one applying the function \( h_1 \) and in another applying the function \( g \). Both being simpler than the global function \( f \), the increase of clock frequency is possible allowing the system to deliver results at a higher rate.
Computer scientists stress on both type of parallelism, each having its own fundamental limitations. More, each form of parallelism bounds the possibility of the other, so as the parallel processes are strictly limited in now a day computation. But, for us it is very important to emphasize in this stage of the approach that:

circuits are essentially parallel structures with both the possibilities and the limits given by the mechanism of composition.

The parallel resources of circuits will be limited also, as we will see, in the process of closing loops one after another with the hope to deal better with complexity.

**Example 3.7** Let us revisit the problem of computing the scalar product. We redesign the circuit in a pipelined version using only binary functions.

```verilog
module pipelined_inner_prod(
  output [17:0] out,
  input [7:0] a3, a2, a1, a0, b3, b2, b1, b0,
  input clock);
wire[15:0] p3, p2, p1, p0;
wire[17:0] s1, s0;
mult mult3(p3, a3, b3, clock),
mult2(p2, a2, b2, clock),
mult1(p1, a1, b1, clock),
mult0(p0, a0, b0, clock);
add add11(s1, {1'b0, p3}, {1'b0, p2}, clock),
add10(s0, {1'b0, p1}, {1'b0, p0}, clock),
add0(out, s1[16:0], s0[16:0], clock);
endmodule

module mult(
  output reg [15:0] out,
  input [7:0] m1, m0,
  input clock);
always @(posedge clock) out <= m1 * m0;
endmodule

module add(output reg [17:0] out,
  input [16:0] t1, t0,
  input clock);
always @(posedge clock) out <= t1 + t0;
endmodule

⋄
```

The structure of the pipelined *inner product* (dot product) circuit is represented in Figure 3.9. It shows us the two dimensions of the parallel computation. The horizontal dimension is associated with *data parallelism*, the vertical dimension is associated with *time parallelism*. The first stage allows 4 parallel computation, the second allows 2 parallel computation, and the last consists only in a single addition. The mean value of the *degree of data parallelism* is 2.33. The system has latency 2, allowing 7 computations in parallel. The *peak performance* of this system is the *whole degree of parallelism* which is 7. The peak performance is the performance obtained if the input stream of data is uninterrupted. If it is interrupted because of the lack of data, or for another reason, the latency will act reducing the peak performance, because some or all pipeline stages will be inactive.
3.5 Featuring by closing new loops

A loop connection is a very simple thing, but the effects introduced in the system in which it is closed are sometimes surprising. All the time are beyond the evolutionary facts. The reason for these facts is the spectacular effect of the autonomy whenever it manifests. The output of the system starts to behave less conditioned by the evolution of inputs. The external behavior of the system starts to depend more and more by something like an “internal state” continuing with a dependency by an “internal behavior”. In the system starts to manifest internal processes seem to be only partially under the external control. Because the loop allows of system to act on itself, the autonomy is the first and the main effect of the mechanism of closing loops. But, the autonomy is only a first and most obvious effect. There are others, more subtle and hidden consequences of this apparent simple and silent mechanism. This book is devoted to emphasize deep but not so obvious correlations between loops and complexity. Let’s start with the definition and a simple example.

Figure 3.10: The loop closed into a digital system. The initial system has two inputs, in1 and in0, and two outputs, out1 and out0. Connecting out0 to in0 results a new system with in and out only.

Definition 3.11 The loop consists in connecting some outputs of a system to some of its inputs (see Figure 3.10), as in the pseudo-Verilog description that follows:

```verilog
define loop_system #('include "parameters.v")( output [out_dim-1:0] out ,
            input [in_dim-1:0] in );
    wire [loop_dim-1:0] the_loop;
    no_loop_system our_module( .out1 (out) ,
            .out0 (the_loop) ,
            .in1 (in) ,
            .in0 (the_loop) );
endmodule

module no_loop_system #('include "parameters.v")( output [out_dim-1:0] out1 ,
            output [loop_dim-1:0] out0 ,
            input [in_dim-1:0] in1 ,
            input [loop_dim-1:0] in0 );
/* The description of 'no_loop_system' module */
endmodule
```

The most interesting thing in the previous definition is a “hidden variable” occurred in module loop_system(). The wire called the_loop carries the non-apparent values of a variable evolving
inside the system. This is the variable which evolves only internally, generating the autonomous behavior of the system. The explicit aspect of this behavior is hidden, justifying the generic name of the “internal state evolution”.

The previous definition don’t introduce any restriction about how the loop must be closed. In order to obtain desired effects the loop will be closed keeping into account restrictions depending by each actual situation. There also are many technological restrictions that impose specific modalities to close loops at different level in a complex digital system. Most of them will be discussed later in the next chapters.

**Example 3.8** *Let be a synchronous adder. It has the outputs synchronized with an positive edge clocked register (see Figure 3.11a). If the output is connected back to one of its input, then results the structure of an accumulator (see Figure 3.11b). The Verilog description follows.*

```verilog
module acc(output[19:0] out, input[15:0] in, input clock, reset);
    sync_add our_add(out, in, out, clock, reset);
endmodule

module sync_add( output reg [19:0] out ,
                 input [15:0] in1 ,
                 input [19:0] in2 ,
                 input clock , reset );
    always @(posedge clock) if (reset) out = 0;
    else out = in1 + in2;
endmodule
```

Figure 3.11: *Example of loop closed over an adder with synchronized output.* If the output becomes one of the inputs, results a circuit that accumulates at each clock cycle. **a.** The initial circuit: the synchronized adder. **b.** The resulting circuit: the accumulator.

_In order to make a simulation the next test_acc module is written:_

```verilog
module test_acc;
    reg clock, reset;
    reg[15:0] in;
    wire[19:0] out;
    initial begin clock = 0;
        forever #1 clock = ~clock;
    end // the clock
    initial begin reset = 1;
```
By simulation results the following behavior:

<table>
<thead>
<tr>
<th>time</th>
<th>clock</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>5</td>
<td>15</td>
</tr>
</tbody>
</table>

The adder becomes an accumulator. What is spectacular in this fact? The step made by closing the loop is important because an “obedient” circuit, whose outputs followed strictly the evolution of its inputs, becomes a circuit with the output depending only partially by the evolution of its inputs. Indeed, the output of the circuit depends by the current input but, in the same time, depends by the content of the register, i.e., by the “history accumulated” in it. The output of adder can be predicted starting from the current inputs, but the output of the accumulator supplementary depends by the state of circuit (the content of the register). It was only a simple example, but I hope, useful to pay more attention to loop.

3.5.1 * Data dependency

The good news about loop is its “ability” to add new features. But any good news is accompanied by its own bad news. In this case is about the limiting of the degree of parallelism allowed in a system with a just added loop. It is mainly about the necessity to stop sometimes the input stream of data in order to decide, inspecting an output, how to continue the computation. The input data waits for data arriving from an output a number of clock cycles related with the system latency. To do something special the system must be allowed to accomplish certain internal processes.

Both, data parallelism and time parallelism are possible because when the data arrive the system “knows” what to do with them. But sometimes the function to be applied on certain input data is decided by processing previously received data. If the decision process is to complex, then new data can not be processed even if the circuits to do it are there.

Example 3.9 Let be the system performing the following function:
3.5. FEATURING BY CLOSING NEW LOOPS

procedure cond_acc(a, b, cond);
    out = 0;
    end = 0;
    loop
        if (cond = 1) out = out + (a + b);
        else out = out + (a - b);
    until (end = 1) // the loop is unending
end

For each pair of input data the function is decided according to a condition input.
The Verilog code describing an associated circuit is:

module cond_acc0( output reg [15:0] out,
    input [15:0] a, b,
    input cond, reset, clock);
always @(posedge clock)
    if (reset) out <= 0;
    else if (cond) out <= out + (a + b);
    else out <= out + (a - b);
endmodule

In order to increase the speed of the circuit a pipeline register is added with the penalty of \(\lambda = 1\). Results:

module cond_acc1( output reg [15:0] out,
    input [15:0] a, b,
    input cond, reset, clock);
reg[15:0] pipe;
always @(posedge clock)
    if (reset) begin out <= 0;
        pipe <= 0;
    end
    else begin if (cond) pipe <= a + b;
        else pipe <= a - b;
        out <= out + pipe;
    end
endmodule

Now let us close a loop in the first version of the system (without pipeline register). The condition input takes the value of the sign of the output. The loop is: \(\text{cond} = \text{out}[15]\). The function performed on each pair of input data in each clock cycle is determined by the sign of the output resulted from the computation performed with the previously received pairs of data. The resulting system is called addapt_acc.

module addapt_acc0(output [15:0] out,
    input [15:0] a, b,
    input reset, clock);
cond_acc0 cont_acc0( .out (out),
    .a (a),
    .b (b),
    .cond (out[15]), // the loop
    .reset (reset),
    .clock (clock));
endmodule
Figure 3.12: Data dependency when a loop is closed in a pipelined structure. a. The non-pipelined version. b. The pipelined version. c. Adding a loop to the non-pipelined version. d. To the pipelined version the loop can not be added without supplementary precautions because data dependency change the overall behavior. The selection between add and sub, performed by the looped signal comes too late.

Figure 3.12a represents the first implementation of the cond_acc circuit, characterized by a low clock frequency because both the adder and the adder/subtractor contribute to limiting the clock frequency:

\[ f_{\text{clock}} = \frac{1}{t_{+/-} + t_c + t_{\text{reg}}} \]

Figure 3.12b represents the pipelined version of the same circuit working faster because only one from adder and the adder/subtractor contributes to limiting the clock frequency:

\[ f_{\text{clock}} = \frac{1}{\max(t_{+/-}, t_c) + t_{\text{reg}}} \]

A small price is paid by \( \lambda = 1 \).

The 1-bit loop closed from the output out[15] to cond input (see Figure 3.12c) allows the circuit to decide itself if the sum or the difference is accumulated. Its speed is identical with the initial, no-loop, circuit.

Figure 3.12d warns us against the expected damages of closing a loop in a pipelined system. Because of the latency the “decision comes” to late and the functionality is altered.
3.5. FEATURING BY CLOSING NEW LOOPS

In the system from Figure 3.12a the degree of parallelism is 1, and in Figure 3.12b the system has the degree of parallelism 2, because of the pipeline execution. When we closed the loop we where obliged to renounce to the bigger degree of parallelism because of the latency associated with the pipe. We have a new functionality – the circuit decides itself regarding the function executed in each clock cycle – but we must pay the price of reducing the speed of the system.

According to the algorithm the function performed by the block $+/-$ depends on data received in the previous clock cycles. Indeed, the sign of the number stored in the output register depends on the data stream applied on the inputs of the system. We call this effect data dependency. It is responsible for limiting the degree of parallelism in digital circuits.

The circuit from Figure 3.12d is not a solution for our problem because the condition cond comes to late. It corresponds to the operation executed on the input stream excepting the most recently received pair of data. The condition comes too late, with a delay equal with the latency introduced by the pipeline execution.

3.5.2 Speculating to avoid limitations imposed by data dependency

How can we avoid the speed limitation imposed by a new loop introduced in a pipelined execution? It is possible, but we must pay a price enlarging the structure of the circuit.

If the circuit does not know what to do, addition or subtract in our previous example, then in it will be compute both in the first stage of pipeline and will delay the decision for the next stage so compensating the latency. We use the same example to be more clear.

Example 3.10 The pipelined version of the circuit addapt_acc is provided by the following Verilog code:

```verilog
module addapt_acc1(output reg [15:0] out,
                     input [15:0] a, b,
                     input reset, clock);
```

Figure 3.13: The speculating solution to avoid data dependency. In order to delay the moment of decision both addition and subtract are computed on the first stage of pipeline. Speculating means instead to decide what to do, addition or subtract, we decide what to consider after doing both.
reg [15:0] pipe1, pipe0;
always @(posedge clock)
  if (reset) begin
    out <= 0;
    pipe1 <= 0;
    pipe0 <= 0;
  end
  else begin
    pipe1 <= a + b;
    pipe0 <= a - b;
    if (out[15]) out <= out + pipe1;
    else out <= out + pipe0;
  end
endmodule

The execution time for this circuit is limited by the following clock frequency:

\[ f_{\text{clock}} = \frac{1}{\max(t_+, t_-, (t_+ + t_{\text{max}})) + t_{\text{reg}}} \approx \frac{1}{t_- + t_{\text{reg}}} \]

The resulting frequency is very near to the frequency for the pipeline version of the circuit designed in the previous example.

Roughly speaking, the price for the speed is: an adder & two registers & a multiplexer (see for comparing Figure 3.12c and Figure 3.13). Sometimes it deserves! ♦

The procedure applied to design \texttt{addapr\_acc1} involves the multiplication of the physical resources. We \textit{speculated}, computing on the first level of pipe both the sum and the difference of the input values. On the second state of pipe the multiplexer is used to select the appropriate value to be added to \texttt{out}.

We call this kind of computation \textit{speculative evaluation} or simply \textit{speculation}. It is used to accelerate complex (i.e., "under the sign" of a loop) computation. The price to be paid is an increased dimension of the circuit.

### 3.6 Concluding about composing & pipelining & looping

The basic ideas exposed in this section are:

- a digital system develops applying two mechanisms: \textbf{composing} functional modules and closing new \textbf{loops}
- by composing the system \textbf{grows} in size improving its functionality with the composed functions
- by closing loops the system gains \textbf{new features} which are different from the previous functionality
- the composition generates the conditions for two kinds of \textbf{parallel computation}:
  - \textbf{data parallel} computation
  - \textbf{time parallel} computation (in \textbf{pipeline} structures)
- the loop limits the possibility to use the time parallel resources of a system because of \textbf{data dependency}
- a \textbf{speculative} approach can be used to accelerate data dependent computation in pipeline systems; it means the execution of operation whose result may not actually be needed; it is an useful optimization when early execution accelerates computation justifying for the wasted effort of computing a value which is never used
3.6. CONCLUDING ABOUT COMPOSING & PIPELINING & LOOPING

- Circuits are mainly parallel systems because of composition (some restriction may apply because of loops).

Related with the computing machines Flynn [Flynn '72] introduced three kind of parallel machines:

- MIMD (multiple-instructions-multiple data), which means mainly having different programs working on different data

- SIMD (single-instructions-multiple-data), which means having one program working on different data,

- MISD (multiple-instructions-single-data), which means having different programs working on the same data.

Related with the computing a certain function also three kind of almost the same parallelism can be emphasized:

- Time parallelism, which is somehow related with MIMD execution, because in each temporal stage a different operation (instruction) can be performed

- Data parallelism, which is identic with SIMD execution

- Speculative parallelism, which is a sort of MISD execution.

Thus, the germs of parallel processes, developed at the computing machine level, occur, at an early stage, at the circuit level.
3.7 Problems

Problem 3.1 Let be the design below. The modules instantiated in topModule are defined only by their time behavior only.

1. Synthesise the circuit.

2. Compute the maximum click frequency.
Speculative circuits

Problem 3.2 Let be the circuit described by the following Verilog module:

```verilog
module xxx( output reg [31:0] a ,
            input [31:0] x1 ,
            input [31:0] x2 ,
            input [31:0] x3 ,
            input clock ,
            input reset );

    always @(posedge clock) if (reset) a <= 0;
    else if (a > x3) a <= a + (x1 + x2);
    else a <= a + (x1 - x2);
endmodule
```

The maximum frequency of clock is limited by the propagation time through the internal loop (\(t_{\text{reg-reg}}\)) or by \(t_{\text{in-reg}}\). To maximize the frequency a speculative solution is asked.

Problem 3.3 Provide the speculative solution for the next circuit.

```verilog
module yyy( output reg [31:0] a ,
            output reg [31:0] b ,
            input [31:0] x1 ,
            input [31:0] x2 ,
            input clock ,
            input reset );

    always @(posedge clock)
    if (reset) begin a <= 0;
      b <= 0;
    end
    else case({a + b > 8'b101, a - b < 8'b111})
      2'b00: {a,b} <= {a + (x1-x2),
                 b + (x1+x2) };
      2'b01: {a,b} <= {a + (8'b10 * x1+x2),
                    b + (x1+8'b10 * x2) };
      2'b10: {a,b} <= {a + (8'b100 * x1-x2),
                       b + (8'b100 * x2) };
      2'b11: {a,b} <= {a + (x2-x1),
                       b + (8'b100 * x1+x2)};
    endcase
endmodule
```

Problem 3.4 The following circuit has two included loops. The speculation will increase the dimension of the circuit accordingly. Provide the speculative version of the circuit.

```verilog
module zzz( output reg [31:0] out,
            input [15:0] x1,
            input [15:0] x2,
            input [15:0] x3,
            input clock,
            input reset);

    reg [15:0] acc;

    always @(posedge clock)
```

```
if (reset) begin
  out <= 0;
  acc <= 0;
end
else begin
  out <= acc * x3;
  if (out[15]) acc <= acc + x1 + x2 + out[31:0];
  else acc <= acc + x1 - x2 + out[15:0];
end
endmodule

3.8 Projects

Use Appendix How to make a project to learn how to proceed in implementing a project.

Project 3.1
Chapter 4

THE TAXONOMY OF DIGITAL SYSTEMS

In the previous chapter
the basic mechanisms involved in defining the architecture of a digital system were introduced:

- the parallel composing and the serial composing are the mechanism allowing two kind of parallelism in digital systems – data parallelism & time parallelism – both involved in increasing the “brute force” of a computing machine
- the pipeline connection supports the time parallelism, accelerating the inherent serial computation
- closing loops new kinds of functionality are allowed (storing, behaving, interpreting, ... self-organizing)
- speculating is the third type of parallelism introduced to compensate the limitations generated by loops closed in pipelined systems

In this chapter
loops are used to classify digital systems in orders, taking into account the increased degree of autonomy generated by each new added loop. The main topics are:

- the autonomy of a digital system depends on the number of embedded loops closed inside
- the loop based taxonomy of digital systems developed to match the huge diversity of the systems currently developed
- some preliminary remarks before starting to describe in detail digital circuits and how they can be used to design digital systems

In the next chapter
the final target of our lessons on digital design is defined as the structure of the simplest machine able to process a stream of input data providing another stream of output data. The functional description of the machine is provided emphasizing:

- the external connections and how they are managed
- the internal control functions of the machine
- the internal operations performed on the received and internally stored data.
A theory is a compression of data; comprehension is compression.

Gregory Chaitin

Any taxonomy is a compressed theory, i.e., a compression of a compression. It contains, thus, illuminating beauties and dangerous insights for our way to comprehend a technical domain. How can we escape from this attractive trap? Trying to comprehend beyond what the compressed data offers.

4.1 Loops & Autonomy

The main and the obvious effect of the loop is the autonomy it can generate in a digital system. Indeed, the first things we observe in a circuit in which a new loop is introduced are new and independent behaviors. Starting with a simple example the things will become more clear in an easy way. We use an example with a system initially defined by a transition table. Each output corresponds to an input with a certain delay (one time unit, #1, in our example). After closing the loop, starts a sequential process, each sequence taking time corresponding with the delay introduced by the initial system.

Example 4.1 Let be the digital system initSyst from Figure 4.1a, with two inputs, \( \text{in} \), \( \text{lp} \), and one output, \( \text{out} \). What happens when is closed the loop from the output \( \text{out} \) to the input \( \text{lp} \)? Let’s make it. The following Verilog modules describe the behavior of the resulting circuit.

```verilog
module loopSyst( output [1:0] out, input in);
    initSyst noLoopsyst(.out(out), .in(in), .loop(out));
endmodule

module initSyst( output reg [1:0] out , input in , input [1:0] loop);

initial out = 2'b11; // only for simulation purpose
always @(in or loop) #1 case ({in, loop})
    3'b000: out = 2'b01;
    3'b001: out = 2'b00;
    3'b010: out = 2'b00;
    3'b011: out = 2'b10;
    3'b100: out = 2'b01;
    3'b101: out = 2'b10;
    3'b110: out = 2'b11;
    3'b111: out = 2'b01;
```

1From [Chaitin '06]
In order to see how behave loopSyst we will use the following test module which initialize (for this example in a non-orthodox fashion because we don’t know nothing about the internal structure of initSyst) the output of initSyst in 11 and put on the input in for 10 unit time the value 0 and for the next 10 unit time the value 1.

```verilog
module test;
    reg in;
    wire[1:0] out;
    initial begin
        in = 0;
        #10 in = 1;
        #10 $stop;
    end
    loopSyst dut(out, in);
    initial $monitor( "time=%0d in=%b out=%b", $time, in, dut.out);
endmodule
```

The simulation offers us the following behavior:

```
# time=0  in=0 out=11
# time=1  in=0 out=10
# time=2  in=0 out=00
# time=3  in=0 out=01
# time=4  in=0 out=00
# time=5  in=0 out=01
# time=6  in=0 out=00
# time=7  in=0 out=01
```

Figure 4.1: Example illustrating the autonomy. a. A system obtained from an initial system in which a loop is closed from output to one of its input. b. The transition table of the initial system where each output strict corresponds to the input value. c. The output evolution for constant input: in = 0. d. The output evolution for a different constant input: in = 1.
The main effect we want to emphasize is the evolution of the output under no variation of the input in. The initial system, defined in the previous case, has an output that switches only responding to the input changing (see also the table from Figure 4.1b). The system which results closing the loop has its own behavior. This behavior depends by the input value, but is triggered by the events coming through the loop. Figure 4.1c shows the output evolution for $\text{in} = 0$ and Figure 4.1d represents the evolution for $\text{in} = 1$.

Verilog Summary 3:

- the register $\text{reg}[1:0]$ out defined in the module initSyst is not a register, it is a Verilog variable, whose value is computed by a case procedure anytime at least one of the two inputs change ($\text{always @(in or lp)}$)
- a register which changes its state “ignoring” a clock edge is not a register, it is a variable evolving like the output of a combinational circuit
- what is the difference between an assign and an always ($a \text{ or } b \text{ or } ...$)? The body of assign is continuously evaluated, rather than the body of always which is evaluated only if at least an element of the list of sensitivity ($\text{(a or b or ...)}$) changes
- in running a simulation an assign is more computationally costly in time than an always which is more costly in memory resources.

Until now we used in a non-rigorous manner the concept of autonomy. It is necessary for our next step to define more clearly this concept in the digital system domain.

**Definition 4.1** In a digital system a behavior is called autonomous iff for the same input dynamic there are defined more than one distinct output transitions, which manifest in distinct moments.

If we take again the previous example we can see in the result of the simulation that in the moment $\text{time} = 2$ the input switches from 0 to 0 and the output from 10 to 00. In the next moment input switches the same, but output switches from 00 to 10. The input of the system remains the same, but the output behaves distinctly. The explanations is for us obvious because we have access to the definition of the initial system and in the transition table we look for the first transition in the line 010 and we find the
output 00 and for the second in the line 000 finding there 00. The input of the initial system is changed because of the loop that generates a distinct response.

In our example the input dynamic is null for a certain output dynamic. There are example when the output dynamic is null for some input transitions (will be found such examples when we talk about memories).

**Theorem 4.1** In the respect of the previous definition for autonomy, closing an internal loop generates autonomous behaviors.

**Proof**  Let be the description of ’no_loop_system’ module from Definition 3.4 described, in the general form, by the following pseudo-Verilog construct:

```verilog
class (in1)
  ... : case (in0) 
    ... : {out1, out0} = f_00(in1, in0);
    ... : {out1, out0} = f_0p(in1, in0);
    endcase 
    ...
    ... : case (in0)
    ... : {out1, out0} = f_q0(in1, in0);
    ... : {out1, out0} = f_qp(in1, in0);
    endcase 
  endcase
endclass
```

The various occurrences of \{out1, out2\} are given by the functions f_ij(in1, in2) defined in Verilog.

When the loop is closed, in0 = out0 = state, the in1 remains the single input of the resulting system, but the internal structure of the system continue to receive both variable, in1 and in0. Thus, for a certain value of in1 there are more Verilog functions describing the next value of \{out1, out0\}. If in1 = const, then the previous description is reduced to:

```verilog
always @(state) #1 case (state)
  ... : {out1, state} = f_i0(const, state);
  ... : {out1, state} = f_ip(const, state);
endcase
```

The output of the system, out1, will be computed for each change of the variable state, using the function f_ji selected by the new value of state, which function depends by state. For each constant value of in1 another set of functions is selected. In the two-level case, which describe no_loop_system, this second level is responsible for the autonomous behavior.

□

### 4.2 Classifying Digital Systems

The two mechanisms, of composing and of ”looping”, give us a very good instrument for a new classification of digital systems. If the system grows by different compositions, then it allows various kinds of
connections. In this context the loops are difficult to be avoided. They occur sometimes in large systems without the explicit knowledge of the designer, disturbing the design process. But, usually we design being aware of the effect introduced by this special connection – the loop. This mechanism leads us to design a complex network of loops which include each other. Thus, in order to avoid ambiguities in using the loops we must define what means “included loop”. We shall use frequently in the next pages this expression for describing how digital systems are built.

**Definition 4.2** A loop includes another loop only when it is closed over a serial or a serial-parallel composition which have at least one subsystem containing an internal loop, called an included loop.

**Attention!** In a parallel composition a loop going through one of the parallel connected subsystem does not include a loop closed over another parallel connected subsystem. A new loop of the kind “grows” only a certain previously closed loop, but does not add a new one.

**Example 4.2** In Figure 4.2 the loop (1) is included by the loop (2). In a serial composition built with $S_1$ and $S_2$ interconnected by (3), we use the connection (2) to add a new loop.

```
X
(1) the loop which includes
(2) the included loop
(3) the serial connection
```

![Figure 4.2: Included loops. The loop (2) includes loop (1), closed over the subsystem $S_2$, because $S_2$ is serially connected with the subsystem $S_1$ and loop (2) includes both $S_1$ and $S_2$.](image)

Now we can use the next recursive definition for a new classification of digital systems. The classification contains orders, from 0 to $n$.

**Definition 4.3** Let be a $n$-order system, $n$-OS. A $(n+1)$-OS can be built only adding a new loop which includes the first $n$ loops. The 0-OS contains only combinational circuits (the loop-less circuits).

This classification in orders is very consistent with the nowadays technological reality for $n < 5$. Over this order the functions of digital systems are imposed mainly by information, this strange ingredient who blinks in 2-OS, is born in 3-OS and grows in 4-OS monopolizing the functional control in digital systems (see Chapter 16 in this book). But obviously, a function of a circuit belonging of certain order can be performed also by circuits from any higher ones. For this reason we use currently circuits with more than 4 loops only for they allow us to apply different kind of optimizations. Even if a new loop is not imposed by the desired functionality, we will use it sometimes because of its effect on the system complexity. As will be exemplified, a good fitted loop allows the segregation of the simple part from an apparent complex system, having as main effect a reduced complexity.

Our intention in the **second part** of this book is to propose and to show how works the following classification:
4.2. CLASSIFYING DIGITAL SYSTEMS

0-OS - combinational circuits, with no autonomy
1-OS - memories, having the autonomy of internal state
2-OS - automata, with the autonomy to sequence
3-OS - processors, with the autonomy to control
4-OS - computers, with the autonomy to interpret
... 
n-OS - systems with the highest autonomy: to self-organize.

Figure 4.3: Examples of circuits belonging to different orders. A combinational circuit is in 0-OS class because has no loops. A memory circuit contains one-loop circuits and therefore it is in 1-OS class. Because the register belongs to 1-OS class, closing a loop containing a register and a combinational circuit (which is in 0-OS class) results an automaton: a circuit in 2-OS class. Two loop connected automata – a circuit in 3-OS class – can work as a processor. An example of 4-OS is a simple computer obtained loop connecting a processor with a memory. Cellular automata contains a number of loops related with the number of automata it contains.

This new classification can be exemplified\(^2\) (see also Figure 4.3) as follows:

- 0-OS: gate, elementary decoder (as the simplest parallel composition), buffered elementary decoder (the simplest serial-parallel composition), multiplexer, adder, priority encoder, ...

\(^2\)For almost all the readers the following enumeration is now meaningless. They are kindly invited to revisit this end of chapter after assimilating the first 7 chapter of this book.
1-OS: elementary latch, master-slave flip-flop (serial composition), random access memory (parallel composition), register (serial-parallel composition), ...

2-OS: T flip-flop (the simplest two states automaton), J-K flip-flop (the simplest two input automaton), counters, automata, finite automata, ...

3-OS: automaton using loop closed through K-J flip-flops or counters, stack-automata, elementary processors, ...

4-OS: micro-controller, computer (as Processor & RAM loop connected), stack processor, co-processor

n-OS: cellular automaton.

The second part of this book is devoted to sketch a digital system theory based on these two-mechanism principle of evolving in digital circuits: composing & looping. Starting with combinational, loop-less circuits with no autonomy, the theory can be developed following the idea of the increasing system autonomy with each additional loop. Our approach will be a functional one. We will start with simple functions and we will end with complex structures with emphasis on the relation between loops and complexity.

4.3 # Digital Super-Systems

When a global loop is introduced in an $n$-order system results a digital super-system (DSS).

4.4 Preliminary Remarks On Digital Systems

The purpose of this first part of the book is to run over the general characteristics of digital systems using an informal high level approach. If the reader become accustomed with the basic mechanisms already described, then in the second part of this book he will find the necessary details to make useful the just acquired knowledge. In the following paragraphs the governing ideas about digital systems are summed up.

**Combinational circuits vs. sequential circuits** Digital systems receive symbols or stream of symbols on their inputs and generate other symbols or stream of symbols on their outputs by computation. For combinational systems each generated symbol depends only by the last recently received symbol. For sequential systems at least certain output symbols are generated taking into account, instead of only one input symbol, a stream of more than one input symbols. Thus, a sequential system is history sensitive, memorizing the meaningful events for its own evolution in special circuits – called registers – using a special synchronization signal – the clock.

**Composing circuits & closing loops** A big circuit results composing many small ones. A new kind of feature can be added only closing a new loop. The structural composing corresponds the the mathematical concept of composition. The loop corresponds somehow to the formal mechanism of recursion. Composing is an “additive” process which means to put together different simple function to obtain a
bigger or a more complex one. Closing a loop new behaviors occur. Indeed, when a snake eats a mouse nothing special happens, but if the Orouboros\textsuperscript{3} serpent bits its own tail something very special must be expected.

**Composition allows data parallelism and time parallelism**  Digital systems perform in a “natural” way parallel computation. The composition mechanism generate the context for the most frequent forms of parallelism: *data parallelism* (in parallel composition) and *time parallelism* (in serial composition). Time parallel computation is performed in pipeline systems, where the only limitation is the *latency*, which means we must avoid to stop the flow of data through the “pipe”. The simplest data parallel systems *can* be implemented as combinational circuits. The simplest time parallel systems *must* be implemented as sequential circuits.

**Closing loops disturbs time parallelism**  The price we pay for the additional features we get when a new loop is closed is, sometimes, the necessity to stop the data flow through the pipelined circuits. The stop is imposed by the latency and the effect can be loosing, totally or partially, the benefit of the existing time parallelism. *Pipelines & loops* is a bad mixture, because the pipe delays the data coming back from the output of the system to its own input.

**Speculation can restore time parallelism**  If the data used to decide comes back to late, the only solution is to delay also the decision. Follows, instead of *selecting what to do*, the need to perform *all* the computations envisaged by the decision and to *select later only the desired result* according to the decision. To do all the computations means to perform *speculative parallel computation*. The structure imposed for this mechanism is a MISD (multiple instruction single data) parallel computation on certain pipeline stage(s). Concluding, *three kind of parallel processes* can be stated in a digital system: data parallelism, time parallelism and speculative parallelism.

**Closed loops increase system autonomy**  The features added by a loop closed in a digital system refer mainly to different kinds of autonomy. The loop uses the just computed data to determine how the computation must be continued. It is like an internal decision is partially driven by the system behavior. Not all sort of autonomy is useful. Some times the increased autonomy makes the system too “stubborn”, unable to react to external control signals. For this reason, only an *appropriately closed loop* generates an useful autonomy, that autonomy which can be used to minimize the externally exercised control. More about how to close proper loops in the next chapters.

**Closing loops induces a functional hierarchy in digital systems**  The degree of autonomy is a good criteria to classify digital systems. The proposed taxonomy establishes the degree of autonomy counting the number of the included loops closed inside a system. Digital system are classified in *orders*: the 0-order systems contain no loop circuits, and *n*-order systems contain at least one circuit with *n* included loops. This taxonomy corresponds with the structural and functional diversity of the circuits used in the actual digital systems.

\textsuperscript{3}This symbol appears usually among the Gnostics and is depicted as a dragon, snake or serpent biting its own tail. In the broadest sense, it is symbolic of time and the continuity of life. The Orouboros biting its own tail is symbolic of self-fecundation, or the “primitive” idea of a self-sufficient Nature - a Nature, that is continually returning, within a cyclic pattern, to its own beginning.
The top view of the digital circuits domain is almost completely characterized by the previous features. Almost all of them are not technology dependent. In the following, the physical embodiment of these concepts will be done using CMOS technology. The main assumptions grounding this approach may change in time, but now they are enough robust and are simply stated as follows: computation is an effective formally defined process, specified using finite descriptions, i.e., the length of the description is not related with the dimension of the processed data, with the amount of time and of physical resources involved.

**Important question:** What are the rules for using composition and looping? No rules restrict us to compose or to loop. The only restrictions come from our limited imagination.

### 4.5 Problems

**Autonomous circuits**

**Problem 4.1** Prove the reciprocal of Theorem 1.1.

**Problem 4.2** Let be the circuit from Problem 1.25. Use the Verilog simulator to prove its autonomous behavior. After a starting sequence applied on its inputs, keep a constant set of values on the input and see if the output is evolving.

Can be defined an input sequence which brings the circuit in a state from which the autonomous behavior is the longest (maybe unending)? Find it if it exists.

**Problem 4.3** Design a circuit which after the reset generates in each clock cycle the next Fibonacci number starting from zero, until the biggest Fibonacci number smaller than $2^{32}$. When the biggest number is generated the machine will start in the next clock cycle from the beginning with 0. It is supposed the biggest Fibonacci number smaller than $2^{32}$ in unknown at the design time.

**Problem 4.4** To the previously designed machine add a new feature: an additional output generating the index of the current Fibonacci number.

### 4.6 Projects

Use Appendix *How to make a project* to learn how to proceed in implementing a project.

**Project 4.1**
Chapter 5

OUR FINAL TARGET

In the previous chapter

a new, loop based taxonomy was introduced. Because each newly added loop increases the autonomy of the system, results a functional circuit hierarchy:

- **history free, no-loop**, combinational circuits performing logic and arithmetic functions (decoders, multiplexors, adders, comparators, ...)
- **one-loop** circuits used mainly as storage support (registers, random access memories, register files, shift registers, ...)
- **two-loop**, automata circuits used for recognition, generation, control, in simple (counters, ...) or complex (finite automata) embodiments
- **three-loop**, processors systems: the simplest information & circuit entanglement used to perform complex functions
- **four-loop**, computing machines: the simplest digital systems able to perform complex programmable functions, because of the segregation between the simple structure of the circuit and the complex content of the program memory
- ...

In this chapter

a very simple programmable circuit, called *toyMachine*, is described using the shortest Verilog description which can be synthesized using the current tools. It is used to delimit the list of circuits that must be taught for undergraduates students. This version of a programmable circuit is selected because:

- its physical implementation contains only the basic structures involved in defining a digital system
- it is a very small & simple entangled structure of circuits & information used for defining, designing and building a digital system with a given transfer function
- it has a well weighted complexity so as, after describing all the basic circuits, an enough meaningful structure can be synthesized.

In the next chapter

starts the second part of this book which describes digital circuits closing a new loop after each chapter. It starts with the chapter about no-loop digital circuits, discussing about:

- simple (and large sized) uniform combinational circuits, easy to be described using a recursive pattern
- complex and size limited random combinational circuits, whose description’s size is in the same range with their size
We must do away with all explanation, and description alone must take its place. … The problems are solved, not by giving new information, but by arranging what we have always known.

Ludwig Wittgenstein

Before proceeding to accomplish our targeted project we must describe it using what we have always known.

Our final target, for these lessons on Digital Design, is described in this chapter as an architecture. The term is borrowed from builders. They use it to define the external view and the functionality of a building. Similarly, in computer science the term of architecture denotes the external connections and the functions performed by a computing machine. The architecture does not tell anything about how the defined functionality is actually implemented inside the system. Usually there are multiple possible solutions for a given architecture.

The way from “what” to “how” is the content of the next part of this book. The architecture we will describe here states what we intend to do, while for learning how to do, we must know a lot about simple circuits and the way they can be put together in order to obtain more complex functions.

5.1 toyMachine: a small & simple computing machine

The architecture of one of the simplest meaningful machine will be defined by (1) its external connections, (2) its internal state and (3) its transition functions. The transition functions refer to how both, the internal state (the function \( f \) from the general definition) and the outputs (the function \( g \) from the general definition) switch.

Let us call the proposed system toyMachine. It is almost the simplest circuit whose functionality can be defined by a program. Thus, our target is to provide the knowledge for building a simple programmable circuit in which both, the physical structure of the circuit and the informational structure of the program contribute to the definition of a certain function.

The use of such a programmable circuit is presented in Figure 5.1, where inputStream\([15:0]\) represents the stream of data which is received by the toyMachine processor, it is processed according to the program stored in programMemory, while, if needed, dataMemory stores intermediate data or support data. The result is issued as the data stream outputStream\([15:0]\).

For the purpose of this chapter, the internal structure of toyMachine is presented in Figure 5.2. The internal state of toyMachine is stored in:

\[^1\]From Wittgenstein’s *Philosophical Investigation* (#109). His own very original approach looked for an alternative way to the two main streams of the 20th Century philosophy: one originated in Frege’s formal positivism, and another in Husserl’s phenomenology. Wittgenstein can be considered as a forerunner of the architectural approach, his vision being far beyond his contemporary fellows were able to understand.
programCounter: is a 32-bit register which stores the current address in the program memory; it points in the program memory to the currently executed instruction; the reset signal sets its value to zero; during the execution of each instruction its content is modified in order to read the next instruction.

intEnable: is a 1-bit state register which enable the action of the input int; the reset signal sets it to 0, thus disabling the interrupt signal.

regFile: the register file is a collection of 32 32-bit registers organized as a three port small memory (array of storage elements):

- one port for write to the address destAddr
- one port for read the left operand from the address leftAddr
- one port for read the right operand from the address rightAddr

used to store the most frequently used variables involved in each stage of the computation.

carry: is a 1-bit register to store the value of the carry signal when an arithmetic operation is performed; the value can be used for one of the next arithmetic operation.

inRegister: is a 16-bit input register used as buffer.

outRegister: is a 16-bit output register used as buffer.

The external connections are of two types:

- data connections:
  - inputStream: the input stream of data
  - outputStream: the output stream of data
  - progAddr: the address for the program memory
  - instruction: the instruction received from the program memory read using progAddr
  - dataAddr: the address for data memory
  - dataOut: the data sent to the data memory
dataIn : the data received back from the data memory

- control connections:
  empty : the input data on inStream is not valid, i.e., the system which provides the input stream of data has noting to send for toyMachine
  read : loads in inRegister the data provided by the sender only if empty = 0, else nothing happens in toyMachine or in the sender
  full : the receiver of the data is unable to receive data sent by toyMachine, i.e., the receiver
  write : send the date to the receiver of outStream\(^2\)
  int : interrupt signal is an “intrusive” signal used to trigger a “special event”; the signal int acts, only if the interrupt is enabled (intEnable = 1), as follows:

\(^2\)The previously described four signals define one of the most frequently used interconnection device: the First-In-First-Out buffer (FIFO). A FIFO (called also queue) is defined by the following data connections
  - data input
  - data output
and the following control connections:
  - empty: the queue is empty, nothing to be read
  - read: the read signal used to extract the last recently stored data
  - full: the queue is full, no place to add new data
  - write: add in queue the data input value

In our design, the sender’s output is the output of a FIFO, and the receiver’s input is the input of another FIFO, let us call them outFIFO and inFIFO. The signals inStream, empty and read belong to the outFIFO of the sender, while outStream, full and write belong to the inFIFO of the receiver.
begin    regFile[30] <= programCounter ; // one-level stack
   programCounter <= regFile[31] ;
end

The location regFile[30] is loaded with the current value of programCounter when the interrupt is acknowledged, and the content of regFile[31] is loaded as the next program counter, i.e., the register 31 contains the address of the routine started by the occurrence of the interrupt when it is acknowledged. The content of regFile[30] will be used to restore the state of the machine when the program started by the acknowledged signal int ends.

inta : interrupt acknowledge
store : the write signal for the data memory; is is used to write dataOut at dataAddr in the external data memory
reset : the synchronous reset signal is activated to initialize the system
clock : the clock signal

For the interconnections between the buffer registers, internal registers and the external signals area is responsible the unspecified bloc Combinatorial logic.

The transition function is given by the program stored in an external memory called Program Memory (reg[31:0] programMemory[0:1023], for example). The program “decides” (1) when a new value of the inStream is received, (2) when and how a new state of the machine is computed and (3) when the output outStream is actualized. Thus, the output outStream evolves according to the inputs of the machine and according to the history stored in its internal state.

The internal state of the above described engine is processed using combinational circuits, whose functionality will be specified in this section using a Verilog behavioral description. At the end of the next part of this book we will be able to synthesize the overall system using a Verilog structural description.

The toyMachine’s instruction set architecture (ISA) is a very small subset of any 32-bit processor (for example, the MicroBlaze processor [MicroBlaze]).

Each location in the program memory contains one 32-bit instruction organized in two formats, as follows:

\[
\text{instruction} = \{\text{opCode}[5:0], \text{destAddr}[4:0], \text{leftAddr}[4:0], \text{rightAddr}[4:0], 11\text{’}b0\} \ |
\{\text{opCode}[5:0], \text{destAddr}[4:0], \text{leftAddr}[4:0], \text{immValue}[15:0]\};
\]

where:
- opCode[5:0] : operation code
- destAddr[4:0] : selects the destination in the register file
- leftAddr[4:0] : selects the left operand from the register file
- rightAddr[4:0]: selects the right operand from the register file
- immValue[15:0]: immediate value

The actual content of the first field – opCode[5:0] – determines how the rest of the instruction is interpreted, i.e., what kind of instruction format has the current instruction. The first format applies the operation coded by opCode to the values selected by leftAddr and rightAddr from the register file; the result is stored in register file to the location selected by destAddr. The second format uses immValue extended with sign as a 32-bit value to be stored in register file at destAddr or as a relative address for jump instructions.

The Instruction Set Architecture (ISA) of toyMachine is described in Figure 5.3, where are listed two subset of instructions:
 control instructions: used to control the program flow by different kinds of jumps performed conditioned, unconditioned or triggered by the acknowledged interrupt interrupt

data instructions: used to modify the content of the file register, or to exchange data with the external systems (each execution is accompanied with programCounter <= programCounter + 1).

The file is used to specify opCode, the binary codes associated to each instruction.

The detailed description of each instruction is given by the Verilog behavioral descriptions included in the module toyMachine (see Figure 5.4).

The first 'include includes the binary codes defined in 0_toyMachineArchitecture.v (see Figure 5.3) for each instruction executed by our simple machine.

The second 'include includes the file used to describe how the instruction fields are structured.

The last two 'include lines include the behavioral description for the two subset of instructions performed by our simple machine. These last two files reflect the ignorance of the reader in the domain of digital circuits. They are designed to express only what the designer intent to build, but she/he doesn’t know yet how to do what must be done. The good news: the resulting description can be synthesized.
module toyMachine(
    input [15:0] inStream , // input stream of data
    input empty , // inStream has no meaning
    output read , // read from the sender
    output [15:0] outStream , // output stream of data
    input full , // the receiver is not able to receive
    output write , // write the element form outRegister
    input interrupt , // interrupt input
    output inta , // interrupt acknowledge
    output [31:0] dataAddr , // address for data memory
    output [31:0] dataOut , // data of be stored in data memory
    input store , // store dataOut at dataAddr in data memory
    input [31:0] dataIn , // data from data memory
    output reg [31:0] programCounter, // address for program memory
    input [31:0] instruction , // instruction form the program memory
    input reset , // reset input
    input clock );// clock input

// INTERNAL STATE
reg [15:0] inRegister ;
reg [15:0] outRegister ;
reg [31:0] regFile[0:31] ;
reg carry ;
reg intEnable ;

`include "0_toyMachineArchitecture.v"
`include "instructionStructure.v"
`include "controlFunction.v"
`include "dataFunction.v"
endmodule

Figure 5.4: The file toyMachine.v containing the toyMachine’s behavioral description.

The bad news: the resulting structure is very big (far from optimal) and has a very complex form, i.e.,
no pattern can be emphasized. In order to provide a small & simple circuit, in the next part of this book
we will learn how to segregate the simple part from the complex part of the circuits used to provide
an optimal actual structure. Then, we will learn how to optimize both, the simple, pattern-dominated
circuits and the complex, pattern-less ones.

The file instructionStructure.v (see Figure 5.5) defines the fields of the instruction. For the
two forms of the instruction appropriate fields are provided, i.e., the instruction content is divided in
many forms, thus allowing different interpretation of it. The bits instruction[15:0] are used in two
ways according to the opCode. If the instruction uses two operands, and both are supplied by the content
of the register file, then instruction[15:0] = rightAddr, else the same bits are the most significant
5 bits of the 16-bit immediate value provided to be used as signed operand or as a relative jump address.

The file controlFunction.v (see Figure 5.6) describes the behavior of the control instructions.
The control of toyMachine refers to both, interrupt mechanism and the program flow mechanism.

The interrupt signal int is acknowledged, activating the signal inta only if intEnable = 1 (see
the assign on the first line in Figure 5.6). Initially, the interrupt is not allowed to act: reset signal
forces intEnable = 0. The program decides when the system is “prepared” to accept interrupts. Then,
the execution of the instruction ei (enable interrupt) determines intEnable = 1. When an interrupt is
CHAPTER 5. OUR FINAL TARGET

wire [5:0] opCode ;
wire [4:0] destAddr ;
wire [4:0] leftAddr ;
wire [4:0] rightAddr ;
wire [31:0] immValue ;

assign opCode = instruction[31:26] ;
assign destAddr = instruction[25:21] ;
assign leftAddr = instruction[20:16] ;
assign rightAddr = instruction[15:11] ;
assign immValue = {{16{instruction[15]}}, instruction[15:0]};

Figure 5.5: The file instructionStructure.v.

acknowledged, the interrupt is disabled, letting the program decide when another interrupt is welcomed. The interrupt is disabled by executing the instruction \texttt{di} – disable interrupt.

The program flow is controlled by unconditioned and conditioned jump instructions. But, the \texttt{inta} signal once activated, has priority, allowing the load of the program counter with the value stored in \texttt{regFile[31]} which was loaded, by the initialization program of the system, with the address of the subroutine associated to the interrupt signal.

The value of the program counter, \texttt{programCounter}, is by default incremented with 1, but when a control instruction is executed its value can be incremented with the signed integer \texttt{instruction[15:0]} or set to the value of a register contained in the register file. The program control instructions are:

\begin{itemize}
  \item \textbf{jmp}: absolute jump with the value selected from the register file by the field \texttt{leftAddr}; the register \texttt{programCounter} takes the value contained in the selected register
  \item \textbf{zjmp}: relative jump with the signed value \texttt{immValue} if the content of the register selected by \texttt{leftAddr} from the register file is 0, else \texttt{programCounter = programCounter + 1}
  \item \textbf{nzjmp}: relative jump with the signed value \texttt{immValue} if the content of the register selected by \texttt{leftAddr} from the register file is not 0, else \texttt{programCounter = programCounter + 1}
  \item \textbf{receive}: relative jump with the signed value \texttt{immValue} if \texttt{readyIn} is 1, else \texttt{programCounter = programCounter + 1}
  \item \textbf{issue}: relative jump with the signed value \texttt{immValue} if \texttt{readyOut} is 1, else \texttt{programCounter = programCounter + 1}
  \item \textbf{halt}: the program execution halts, \texttt{programCounter = programCounter} (it is a sort of \texttt{nop} instruction without incrementing the register \texttt{programCounter = programCounter}).
\end{itemize}

\textbf{Warning!} If \texttt{intEnable = 0} when the instruction \texttt{halt} is executed, then the overall system is blocked. The only way to turn it back to life is to activate the \texttt{reset} signal.

The file \texttt{dataFunction.v} (see Figure 5.7) describes the behavior of the data instructions. The signal \texttt{inta} has the highest priority. It forces the register 30 of the register file to store the current state of the register \texttt{programCounter}. It will be used to continue the program, interrupted by the acknowledged interrupt signal \texttt{int}, by executing a \texttt{jmp} instruction with the content of \texttt{regFile[30]}. The following data instructions are described in this file:
5.1. TOYMACHINE: A SMALL & SIMPLE COMPUTING MACHINE

assign inta = intEnable & interrupt;
always @(posedge clock) if (reset) intEnable <= 0 ;
    else if (inta) intEnable <= 0 ;
    else if (opCode == ei) intEnable <= 1 ;
    else (opCode == di) intEnable <= 0 ;
always @(posedge clock)
    if (reset) programCounter <= 0 ;
    else if (inta) programCounter <= regFile[31] ;
    else case(opCode)
        jmp : programCounter <= regFile[leftAddr] ;
        zjmp : if (regFile[leftAddr] == 0)
            programCounter <= programCounter + immValue ;
            else programCounter <= programCounter + 1 ;
        nzjmp : if (regFile[leftAddr] !== 0)
            programCounter <= programCounter + immValue ;
            else programCounter <= programCounter + 1 ;
        rjmp : programCounter <= programCounter + immValue ;
        receive : if (!empty)
            programCounter <= programCounter + 1 ;
            else programCounter <= programCounter ;
        issue : if (!full)
            programCounter <= programCounter + 1 ;
            else programCounter <= programCounter ;
        halt :
            programCounter <= programCounter ;
        default programCounter <= programCounter + 1 ;
endcase

Figure 5.6: The file controlFunction.v.

add : the content of the registers selected by leftAddr and rightAddr are added and the result is stored in the register selected by destAddr; the value of the resulted carry is stored in the carry one-bit register

sub : the content of the register selected by rightAddr is subtracted form the content of the register selected by leftAddr; the result is stored in the register selected by destAddr; the value of the resulted borrow is stored in the carry one-bit register

adde : add with carry - the content of the registers selected by leftAddr and rightAddr and the content of the register carry are added and the result is stored in the register selected by destAddr; the value of the resulted carry is stored in the carry one-bit register

subc : subtract with carry - the content of the register selected by rightAddr and the content of carry are subtracted form the content of the register selected by leftAddr, the result is stored in the register selected by destAddr; the value of the resulted borrow is stored in the carry one-bit register

ashr : the content of the register selected by leftAddr is arithmetically shifted right one position and stored in the register selected by destAddr

gen : every bit contained in the register selected by leftAddr are inverted and the result is stored in the register selected by destAddr
always @posedge clock
if (inta)   regFile[30] <= programCounter ;
else
    case(opCode)
        add : {carry, regFile[destAddr]}
            <= regFile[leftAddr] + regFile[rightAddr] ;
        sub : {carry, regFile[destAddr]}
        addc : {carry, regFile[destAddr]}
        subc : {carry, regFile[destAddr]}
        ashr : regFile[destAddr]
            <= {regFile[leftAddr][31], regFile[leftAddr][31:1]} ;
        val : regFile[destAddr] <= immValue ;
        hval : regFile[destAddr] <= {immValue[15:0], regFile[leftAddr][15:0]};
        get : regFile[destAddr] <= inRegister ;
        send : outRegister <= regFile[leftAddr][15:0] ;
        receive : if (!empty)
            inRegister <= inStream ;
        datard : regFile[destAddr] <= dataIn ;
        default regFile[0] <= regFile[0] ;
    endcase

assign read = (opCode == receive) & (!empty);
assign write = (opCode == issue) & (!full) ;
assign store = (opCode == datawr) ;
assign dataAddr = regFile[leftAddr] ;
assign dataOut = regFile[rightAddr] ;
assign outStream = outRegister ;

Figure 5.7: The file dataFunction.v.
5.2. HOW TOYMACHINE WORKS

bwand : the content of the register selected by leftAddr is **AND-ed** bit-by-bit with the content of the register selected by rightAddr and the result is stored in the register selected by destAddr

bwor : the content of the register selected by leftAddr is **OR-ed** bit-by-bit with the content of the register selected by rightAddr and the result is stored in the register selected by destAddr

bwxor : the content of the register selected by leftAddr is **XOR-ed** bit-by-bit with the content of the register selected by rightAddr and the result is stored in the register selected by destAddr

val : the register selected by destAddr is loaded with the signed integer immValue

hval : is used to construct a 32-bit value placing instruction[15:0] on the 16 highest binary position in the content of the register selected by leftAddr; the result is stored at destAddr in the register file

get : the register selected by destAddr are loaded with the content of inRegister

send : the outRegister register is loaded with the least 15 significant bits of the register selected by leftAddr

receive : if readyIn = 1, then the inRegister is loaded with the current value applied on the input inStream and the readIn signal is activated for the sender to “know” that the current value was received

datard : the data accessed at the address dataAddr = leftOp = regFile[leftAddr] is loaded in register file at the location destAddr

issue : generate, only when readyOut = 1, the signal writeOut used by the receiver to take the value from the outRegister register

datawr : generate the signal write used by the data memory to write at the address regFile[leftAddr] the data stored in regFile[rightAddr]

5.2 How toyMachine works

The simplest, but not the easiest way to use toyMachine is to program it in **machine language**\(^3\), i.e., to write programs as sequence of binary coded instructions stored in programMemory starting from the address 0.

The general way to solve digital problems using toyMachine, or a similar device, is (1) to define the input stream, (2) to specify the output stream, and (3) to write the program which transforms the input stream into the corresponding output stream. Usually, we suppose an input signal which is sampled at a program controlled rate, and the results is an output stream of samples which is interpreted as the output signal. The transfer function of the system is programmed in the binary sequence of instructions stored in the program memory.

The above described method to implement a digital system is called **programmed logic**, because a general purpose programmable machine is used to implement a certain function which generate an output stream of data starting from an input stream of data. The main advantage of this method is its

\(^3\)The next levels are to use an **assembly language** or a **high level language** (for example: C), but these approaches are beyond our goal in this text book.
flexibility, while the main disadvantages are the reduced speed and the increased size of the circuit. If the complexity, price and time to market issues are important, then it can be the best solution.

At http://arh.pub.ro/gstefan/toyMachine.zip you can find the files used to simulate a toy-Machine system.

### 5.2.1 The Code Generator

The file toyMachineCodeGenerator.v contains the description of the engine used to fill up the program memory – progMem – containing the “executable” code, i.e., the binary form of the program to be executed.

For the instructions we use capital letters with parameters in parenthesis, if needed. For example: ADD(4, 3, 17) which stands for add in the register 4 the content of the register 3 with the content of the register 17. To specify the jump addresses are used labels of form LB(2). When the instruction ZJMP(2) is executed, the jump address is calculated using the address labeled with LB(2).

The full form of the file toyMachineCodeGenerator.v is in the folder pointed by http://arh.pub.ro/gstefan/toyMachine.zip, while, in the following, a shorted form is presented in order to explain only the way the code is generated and stored in the program memory.

```verilog
// CODE GENERATOR
reg [5:0] opCode ;
reg [4:0] destAddr ;
reg [4:0] leftAddr ;
reg [4:0] rightAddr ;
reg [10:0] value ;
reg [5:0] addrCounter ;
reg [5:0] labelTab[0:63] ;

#include "0_toyMachineArchitecture.v"

task endLine;
begin
  dut.progMem[addrCounter] = {opCode ,
    destAddr ,
    leftAddr ,
    rightAddr ,
    value } ;
  addrCounter = addrCounter + 1 ;
end
endtask

// sets labelTab in the first pass associating 'counter' with 'labelIndex'
task LB ;
  input [4:0] labelIndex;
  labelTab[labelIndex] = addrCounter;
endtask

// uses the content of labelTab in the second pass
task ULB;
  input [4:0] labelIndex;
  {rightAddr, value} = labelTab[labelIndex] - addrCounter;
endtask

task NOP;
```
5.2. HOW TOYMACHINE WORKS

begin  
opCode = nop ;
destAddr = 5'b0 ;
leftAddr = 5'b0 ;
{rightAddr, value} = 16'b0 ;
endLine
end

task JMP;
  input  [4:0] left;
  begin  
opCode = jmp ;
destAddr = 5'b0 ;
leftAddr = left ;
{rightAddr, value} = 16'b0 ;
endLine
end
endtask

task ZJMP;
  input  [4:0] left;
  input  [5:0] label;
  begin  
opCode = zjmp ;
destAddr = 5'b0 ;
leftAddr = left ;
ULB(label) ;
endLine
end
endtask

// ...

task RJMP;
  input  [5:0] label;
  begin  
opCode = rjmp ;
destAddr = 5'b0 ;
leftAddr = 5'b0 ;
ULB(label) ;
endLine
end
endtask

task EI;
  begin  
opCode = ei ;
destAddr = 5'b0 ;
leftAddr = 5'b0 ;
{rightAddr, value} = 16'b0 ;
endLine
end
endtask

// ...

task AND;
  input  [4:0] dest ;
  input  [4:0] left ;
  input  [4:0] right ;
begin opCode = bwand ;
destAddr = dest ;
leftAddr = left ;
{rightAddr, value} = {right, 11'b0};
endLine ;
end
task ADD;
// ...
begin opCode = add ;
destAddr = dest ;
leftAddr = left ;
{rightAddr, value} = {right, 11'b0};
endLine ;
end
endtask
// ...
task ADDC;
// ...
begin opCode = addc ;
destAddr = dest ;
leftAddr = left ;
{rightAddr, value} = {right, 11'b0};
endLine ;
end
endtask
// ...
task VAL;
// ...
begin opCode = val ;
destAddr = dest ;
leftAddr = 5'B0 ;
{rightAddr, value} = immVal;
endLine ;
end
endtask
// ...
task RECEIVE;
begin opCode = receive ;
destAddr = 5'b0 ;
leftAddr = 5'b0 ;
{rightAddr, value} = 16'b0 ;
endLine ;
end
endtask
// ...
5.2. HOW TOYMACHINE WORKS

```verbatim

```verbatim

The code generator program is a two-pass generator (see RUNNING ... in the above code) which uses, besides the program memory progMem, a counter, addrCounter, and a memory, called labelTab, for storing the address labeled by LB(n). In the first pass, in labelTab are stored the addresses counted by addrCounter (see task LB), while dummy jump addresses are computed using the not up-dated content of the labelTab memory. In the second pass, the content of the labelTab memory is used by task ULB to compute the correct jump addresses.

The main tasks involved are of two types:

- additional tasks used for generating the binary code

  - `endLine`: once a line of code is filled up by an instruction task (such as NOP, AND, OR, ..., JMP, HALT, ...), the resulting binary code is loaded in the program memory at the address given by addrCounter, and the counter addrCounter is incremented.
  - `LB`: the label's argument is loaded in the labelTab memory at the address addrCounter; the action make sense only at the first pass
  - `ULB`: uses, at the second pass, the content of the labelTab memory to compute the actual value of the jump address; it is pointless at the firts pass

- instruction generating tasks, of type:

  - `NOP`: no operand instructions
  - `JMP(n)`: control instruction with one parameter, n, which is a number indicating the register to be used
  - `ZJMP(n,m)`: control instruction with two parameters, n and m, indicating a register and a label to be used for a conditioned jump
  - `ADD(d,l,r)`: three-parameter instruction indicating destination register, d, left operand register, l, and right operand register, r

The program is sequence of tasks which is translated in binary code by the 0_toyMachineCodeGenerator.v program.

Example 5.1 The following simple program:
VAL(1, 5);
VAL(2, 6);
ADD(3, 2, 1);

adds in the register 3 the numbers loaded in the registers 1 and 2. The code generator sees this program as a sequence of three tasks and generates three 32-bit words in the program memory starting with the address 0.

5.2.2 The Simulation Module

The simulation module (stored in the file 0_toyMachineSimulator.v under the name toyMachineSimulator) is used for two purposes:

- as the verification environment for the correctness of the design
- as the verification environment for the correctness of the programs written for the toyMachine simple processor.

The full form of the file toyMachineSimulator.v is in the folder pointed by http://arh.pub.ro/gstefan/toyMachine.zip, while, in the following, a little edited form is presented.

```vhdl
module toyMachineSimulator;
  reg [15:0] inStream; // input stream of data
  reg empty; // input stream is ready
  wire read; // read one element from the input stream
  wire [15:0] outStream; // output stream of data
  reg full; // ready to receive an element of the output stream
  wire write; // write the element form outRegister
  reg interrupt; // interrupt input
  wire inta; // interrupt acknowledge
  reg reset; // reset input
  reg clock; // clock input

  integer i;

  initial begin
clock = 0;
    forever #1 clock = ~clock;
  end

  `include "0_toyMachineCodeGenerator.v"

  initial for (i=0; i<32; i=i+1)
    $display("progMem[%0d] = %b", i, dut.progMem[i]);

  initial begin
    reset = 1;
    inStream = 0;
    empty = 0;
    full = 0;
    #3 reset = 0;
    #380 $stop;
```

5.2. HOW TOYMACHINE WORKS

end

always @(posedge clock)
if (reset) inStream[2:0] <= 0 ;

toySystem dut( inStream ,
empty ,
read ,
outStream ,
full ,
write ,
interrupt ,
inta ,
reset ,
clock );

initial $monitor("time=%0d \t rst=%b pc=%0d ..."); // !!! here the code is edited
$time,
reset,
dut.programCounter,
dut.tM.carry,
dut.tM.regFile[0],
dut.tM.regFile[1],
dut.tM.regFile[2],
dut.tM.regFile[3],
dut.tM.regFile[4],
dut.tM.regFile[5],
dut.tM.regFile[6],
dut.tM.regFile[7],
dut.tM.read,
dut.tM.inRegister,
dut.tM.write,
dut.tM.outRegister);
endmodule

The toySystem module instantiated in the simulator contains, besides the processor, two memories: one for data and another for programs (see Figure 5.1. The module toySystem described in a file contained in the folder pointed by http://arh.pub.ro/gstefan/toyMachine.zip. It has the following form:

module toySystem
  (input [15:0] inStream , // input stream of data
  input empty , // input stream is ready
  output read , // read one element from the input stream
  output [15:0] outStream , // output stream of data
  input full , // ready to receive an element of the output stream
  output write , // write the element from outRegister
  input interrupt , // interrupt input
  output inta , // interrupt acknowledge
  input reset , // reset input
  input clock ); // clock input
CHAPTER 5. OUR FINAL TARGET

```verilog
reg [31:0] progMem[0:1023]; // is a read only memory
reg [31:0] dataMem[0:1023];
wire [31:0] dataAddr; // address for data memory
wire [31:0] dataOut; // data of be stored in data memory
wire store; // write in data memory
wire [31:0] dataIn; // data from data memory
wire [31:0] programCounter; // address for program memory
wire [31:0] instruction; // instruction form the program memory

toyMachine tM( inStream ,
               empty ,
               read ,
               outStream ,
               full ,
               write ,
               interrupt ,
               inta ,
               dataAddr ,
               dataOut ,
               store ,
               dataIn ,
               programCounter ,
               instruction ,
               reset ,
               clock );

always @(posedge clock) dataMem[dataAddr[9:0]] <= dataOut;
assign dataIn = dataMem[dataAddr[9:0]];
assign instruction = progMem[programCounter[9:0]];
endmodule
```

5.2.3 Programming toyMachine

A program, 0_theProgram.v, written by one user is an input for the code generator, 0_toyMachineCodGenerator.v, which at its turn uses the architecture, described in 0_toyMachineArchitecture.v, and is included in the simulator, 0_toyMachineSimulator.v.

Each line in the file 0_theProgram.v contains one instruction or a label followed by an instruction. Each instruction or label represents a task for the cod generator program. The simulator starts by printing the binary form of the program and ends by printing the behavior of the system.

Example 5.2 Let us revisit the pixel correction problem whose solution as circuit was presented in Chapter 1. Now we consider a more elaborated environment (see Figure 5.8). The main difference is that the transfers of the streams of data are now conditioned by specific dialog signals. The subSystem generating pixels is interrogated, by empty, before its output is loaded in inRegister; receiving the data is notified back by the signal read. Similarly, there is a dialog with the subSystem using pixels. It is interrogated by full and notified by write.

In this application the data memory is not needed and the int signal is not used. The corresponding signals are omitted or connected to fix values in Figure 5.8.

The program (see Figure 5.9) is structured to use three sequences of instructions called pseudo-
5.2. HOW TOYMACHINE WORKS

Figure 5.8: Programmed logic implementation for the interpol circuit.

macros$^4$.

- The first, called input, reads the input dealing with the dialog signals, empty and read.
- The second, called output, controls the output stream dealing with the signals full and write.
- The third pseudo-macro tests if the correction is needed, and apply it if necessary.

The pseudo-macro input: The registers 0, 1, and 2 from regFile are used to store three successive values of pixels from the input stream. Then, before receiving a new value, the content of register 1 is moved in the register 2 and the content of register 0 is moved in register 1 (see the first two line in the code printed in Figure 5.9 in the section called "input" pseudo-macro). Now the register 0 form the register file is ready to receive a new value. The next instruction loads in inRegister the value of the input stream when it is valid. The instruction receive loops with the same value in programCounter until the empty signal becomes 0. The input value, once buffered in the inRegister, could be loaded in regFile[0].

The sequence of instructions just described performs the shift operation defined in the module stateTransition which is instantiated in the module pixelCorrector used as example in our introductory first chapter.

The pseudo-macro output: See the code printed in Figure 5.9 in the section called "output" pseudo-macro. The value to be sent out as the next pixel is stored in the register 1. Then, outRegister register must be loaded with the content of regFile[1] (SEND(1)) and the signal write must be kept active until full becomes 0, i.e., the instruction ISSUE loops with the same value in programCounter until full = 0.

The pseudo-macro compute: This pseudo-macro first perform the test on the content of the register 1 (NZJMP(1,2)), and, if necessary, makes the correction adding in the register 1 the content of the registers 0 and 1 (ADD(1,0,2)), and then dividing the result by two performing an arithmetic shift right (ASHR(1,1)).

$^4$The true macros are used in assembly languages. For this level of the machine language a more rudimentary form of macro is used.
CHAPTER 5. OUR FINAL TARGET

RECEIVE; // inRegister is loaded if (empty = 0), else wait
GET(0); // regFile[0] <= inRegister
RECEIVE; // inRegister is loaded if (empty = 0), else wait
GET(1); // regFile[1] <= inRegister

"input" pseudo-macro

LB(1); MOVE(2,1); // regFile[2] <= regFile[1]
MOVE(1,0); // regFile[1] <= regFile[0]
RECEIVE; // inRegister is loaded if (empty = 0), else wait
GET(0); // regFile[0] <= inRegister

"compute" pseudo-macro

NZJMP(1, 2); // if (regFile[1] !== 0) then jump to "output"
ADD(1, 0, 2); // regFile[1] = regFile[0] + regFile[2]
ASHR(1, 1); // regFile[1] = regFile[1]/2

"output" pseudo-macro

LB(2); SEND(1); // outRegister = regFile[1]
ISSUE; // data is issued if (full = 0), else wait
RJMP(1); // unconditioned jump to LB(1)

Figure 5.9: Machine language program for interpol.

The actual program, stored in the internal program memory (see Figure 5.9), has a starting part receiving two input values, followed by an unending loop which receives a new value, computes the value to be sent out and sends it.

The behavior of the system, provided by the simulator, is:

time=0 rst=1 pc=x cr=x rf[0]=x rf[1]=x rf[2]=x read=x inReg=x write=x outReg=x
time=1 rst=1 pc=0 cr=x rf[0]=x rf[1]=x rf[2]=x read=1 inReg=x write=0 outReg=x
time=3 rst=0 pc=1 cr=x rf[0]=x rf[1]=x rf[2]=x read=0 inReg=0 write=0 outReg=x
time=5 rst=0 pc=2 cr=x rf[0]=0 rf[1]=x rf[2]=x read=1 inReg=0 write=0 outReg=x
time=7 rst=0 pc=3 cr=x rf[0]=0 rf[1]=x rf[2]=x read=0 inReg=1 write=0 outReg=x
time=9 rst=0 pc=4 cr=x rf[0]=0 rf[1]=1 rf[2]=x read=0 inReg=1 write=0 outReg=x
time=11 rst=0 pc=5 cr=x rf[0]=0 rf[1]=1 rf[2]=1 read=0 inReg=1 write=0 outReg=x
time=13 rst=0 pc=6 cr=x rf[0]=0 rf[1]=0 rf[2]=1 read=1 inReg=1 write=0 outReg=x
time=15 rst=0 pc=7 cr=x rf[0]=0 rf[1]=0 rf[2]=1 read=0 inReg=2 write=0 outReg=x
time=17 rst=0 pc=8 cr=x rf[0]=0 rf[1]=0 rf[2]=1 read=0 inReg=2 write=0 outReg=x
time=19 rst=0 pc=9 cr=x rf[0]=0 rf[1]=0 rf[2]=1 read=0 inReg=2 write=0 outReg=x
time=21 rst=0 pc=10 cr=x rf[0]=0 rf[1]=3 rf[2]=1 read=0 inReg=2 write=0 outReg=x
time=23 rst=0 pc=11 cr=x rf[0]=2 rf[1]=1 rf[2]=1 read=0 inReg=2 write=0 outReg=x
time=25 rst=0 pc=12 cr=x rf[0]=2 rf[1]=1 rf[2]=1 read=0 inReg=2 write=1 outReg=1
time=27 rst=0 pc=13 cr=x rf[0]=2 rf[1]=1 rf[2]=1 read=0 inReg=2 write=0 outReg=1
time=29 rst=0 pc=4 cr=x rf[0]=2 rf[1]=1 rf[2]=1 read=0 inReg=2 write=0 outReg=1
time=31 rst=0 pc=5 cr=x rf[0]=2 rf[1]=1 rf[2]=1 read=0 inReg=2 write=0 outReg=1
time=33 rst=0 pc=6 cr=x rf[0]=2 rf[1]=2 rf[2]=1 read=1 inReg=2 write=0 outReg=1
time=35 rst=0 pc=7 cr=x rf[0]=2 rf[1]=2 rf[2]=1 read=1 inReg=2 write=0 outReg=1
time=37 rst=0 pc=8 cr=x rf[0]=3 rf[1]=2 rf[2]=1 read=0 inReg=3 write=0 outReg=1
time=39 rst=0 pc=11 cr=x rf[0]=3 rf[1]=2 rf[2]=1 read=0 inReg=3 write=0 outReg=1
time=41 rst=0 pc=12 cr=x rf[0]=3 rf[1]=2 rf[2]=1 read=0 inReg=3 write=1 outReg=2
time=43 rst=0 pc=13 cr=x rf[0]=3 rf[1]=2 rf[2]=1 read=0 inReg=3 write=0 outReg=2
time=45 rst=0 pc=4 cr=x rf[0]=3 rf[1]=2 rf[2]=1 read=0 inReg=3 write=0 outReg=2
time=47 rst=0 pc=5 cr=x rf[0]=3 rf[1]=2 rf[2]=2 read=0 inReg=3 write=0 outReg=2
5.2. **HOW TOYMACHINE WORKS**

```plaintext
progMem[0] = 10000000000000000000000000000000
progMem[1] = 10001000000000000000000000000000
progMem[2] = 10000000000000000000000000000000
progMem[3] = 10001000000000000000000000000000
progMem[4] = 01100000100001000000000000000000
progMem[5] = 01100000100001000000000000000000
progMem[6] = 10000000000000000000000000000000
progMem[7] = 10001000000000000000000000000000
progMem[8] = 00010000000010000000000000000000
progMem[9] = 00000000000000000000000000000000
progMem[10] = 00000000000000000000000000000000
progMem[12] = 10000100000000000000000000000000
progMem[13] = 00010000000000001111111111111111
progMem[14] = 01010000000000000000000000000000
progMem[15] = 01100100000000000000000000000000
progMem[16] = 10001100000000010000000000000000
progMem[17] = 10000100000000000000000000000000
progMem[18] = 00010000000000001111111111111111
```

Figure 5.10: **The binary form of the program for** interpol.

```plaintext
time=49  rst=0  pc=6  cr=0  rf[0]=3  rf[1]=3  rf[2]=2  read=1  inReg=3  write=0  outReg=2
time=51  rst=0  pc=7  cr=0  rf[0]=3  rf[1]=3  rf[2]=2  read=0  inReg=4  write=0  outReg=2
time=53  rst=0  pc=8  cr=0  rf[0]=4  rf[1]=3  rf[2]=2  read=0  inReg=4  write=0  outReg=2
time=55  rst=0  pc=11  cr=0  rf[0]=4  rf[1]=3  rf[2]=2  read=0  inReg=4  write=0  outReg=2
time=57  rst=0  pc=12  cr=0  rf[0]=4  rf[1]=3  rf[2]=2  read=0  inReg=4  write=0  outReg=2
time=59  rst=0  pc=13  cr=0  rf[0]=4  rf[1]=3  rf[2]=2  read=0  inReg=4  write=0  outReg=2
time=61  rst=0  pc=4  cr=0  rf[0]=4  rf[1]=3  rf[2]=2  read=0  inReg=4  write=0  outReg=2
time=63  rst=0  pc=5  cr=0  rf[0]=4  rf[1]=3  rf[2]=2  read=0  inReg=4  write=0  outReg=2
time=65  rst=0  pc=6  cr=0  rf[0]=4  rf[1]=3  rf[2]=2  read=0  inReg=4  write=0  outReg=2
time=67  rst=0  pc=7  cr=0  rf[0]=4  rf[1]=4  rf[2]=3  read=0  inReg=5  write=0  outReg=3
time=69  rst=0  pc=8  cr=0  rf[0]=4  rf[1]=4  rf[2]=3  read=0  inReg=5  write=0  outReg=3
time=71  rst=0  pc=11  cr=0  rf[0]=5  rf[1]=4  rf[2]=3  read=0  inReg=5  write=0  outReg=3
time=73  rst=0  pc=12  cr=0  rf[0]=5  rf[1]=4  rf[2]=3  read=0  inReg=5  write=1  outReg=4
time=75  rst=0  pc=13  cr=0  rf[0]=5  rf[1]=4  rf[2]=3  read=0  inReg=5  write=0  outReg=4
time=77  rst=0  pc=4  cr=0  rf[0]=5  rf[1]=4  rf[2]=3  read=0  inReg=5  write=0  outReg=4
time=79  rst=0  pc=5  cr=0  rf[0]=5  rf[1]=4  rf[2]=3  read=0  inReg=5  write=0  outReg=4
time=81  rst=0  pc=6  cr=0  rf[0]=5  rf[1]=5  rf[2]=4  read=1  inReg=5  write=0  outReg=4
time=83  rst=0  pc=7  cr=0  rf[0]=5  rf[1]=5  rf[2]=4  read=0  inReg=6  write=0  outReg=4
time=85  rst=0  pc=8  cr=0  rf[0]=6  rf[1]=5  rf[2]=4  read=0  inReg=6  write=0  outReg=4
time=87  rst=0  pc=11  cr=0  rf[0]=6  rf[1]=5  rf[2]=4  read=0  inReg=6  write=0  outReg=4
time=89  rst=0  pc=12  cr=0  rf[0]=6  rf[1]=5  rf[2]=4  read=0  inReg=6  write=1  outReg=5
time=91  rst=0  pc=13  cr=0  rf[0]=6  rf[1]=5  rf[2]=4  read=0  inReg=6  write=0  outReg=5
time=93  rst=0  pc=4  cr=0  rf[0]=6  rf[1]=5  rf[2]=4  read=0  inReg=6  write=0  outReg=5
time=95  rst=0  pc=5  cr=0  rf[0]=6  rf[1]=5  rf[2]=5  read=0  inReg=6  write=0  outReg=5
time=97  rst=0  pc=6  cr=0  rf[0]=6  rf[1]=6  rf[2]=5  read=1  inReg=6  write=0  outReg=5
time=99  rst=0  pc=7  cr=0  rf[0]=6  rf[1]=6  rf[2]=5  read=0  inReg=7  write=0  outReg=5
time=101  rst=0  pc=8  cr=0  rf[0]=7  rf[1]=6  rf[2]=5  read=0  inReg=7  write=0  outReg=5
time=103  rst=0  pc=11  cr=0  rf[0]=7  rf[1]=6  rf[2]=5  read=0  inReg=7  write=0  outReg=5
time=105  rst=0  pc=12  cr=0  rf[0]=7  rf[1]=6  rf[2]=5  read=0  inReg=7  write=1  outReg=6
time=107  rst=0  pc=13  cr=0  rf[0]=7  rf[1]=6  rf[2]=5  read=0  inReg=7  write=0  outReg=6
```
CHAPTEER 5. OUR FINAL TARGET
5.3 Concluding about toyMachine

Our final target is to be able to describe the actual structure of toyMachine using as much as possible simple circuits. Maybe we just caught a glimpse about the circuits we must learn how to design. It is almost obvious that the following circuits are useful for building toyMachine: adders, subtractors, increment circuits, selection circuits, various logic circuits, registers, file-registers, memories, read-only memories (for fix program memory). The next chapters present detailed descriptions of all above circuits, and a little more.

The behavioral description of toyMachine is synthesisable, but the resulting structure is too big and has a completely unstructured shape. The size increases the price, and the lack of structure make impossible any optimization of area, of speed or of the energy consumption.

The main advantages of the just presented behavioral description is its simplicity, and the possibility to use it as a more credible “witness” when the structural description will be verified.

Pros & cons for programmed logic:

- it is a very flexible tool, but can not provide hi-performance solutions
- very good time-to-market, but not for mass production
- good for simple one chip solution, but not to be integrated as an IP on a complex SoC solution.

5.4 Problems

Problem 5.1 Write for toyMachine the program which follows-up as fast as possible by the value on outStream the number of 1s on the inputs inStream.

Problem 5.2 Redesign the interpol program for a more accurate interpolation rule:

\[ p_i = 0.2 \times p_{i-2} + 0.3 \times p_{i-1} + 0.3 \times p_{i+1} + 0.2 \times p_{i+2} \]

Problem 5.3

5.5 Projects

Project 5.1 Design the test environment for toyMachine, and use it to test the example from this chapter.
Part II

LOOPING IN THE DIGITAL DOMAIN
Chapter 6

GATES:
Zero order, no-loop digital systems

In the previous chapter ended the first part of this book, where we learned to “talk” in the Verilog HDL about how to build big systems composing circuits and smaller systems, how to accelerate the computation in a lazy system, and how to increase the autonomy of a system closing appropriate loops. Where introduced the following basic concepts:

- serial, parallel, and serial-parallel compositions used to increase the size of a digital system, maintaining the functional capabilities at the same level
- data (synchronous) parallelism and time (diachronic) parallelism (the pipeline connection) as the basic mechanism to improve the speed of processing in digital systems
- included loops, whose effect of limiting the time parallelism is avoided by speculating – the third form of parallelism, usually ignored in the development of the parallel architectures
- classifying digital circuits in orders, the $n$-th order containing circuits with $n$ levels of embedded loops

The last chapter of the first part defines the architecture of the machine whose components will be described in the second part of this book.

In this chapter the zero order, no-loop circuits are presented with emphasis on:

- how to expand the size of a basic combinational circuit
- the distinction between simple and complex combinatorial circuits
- how to deal with the complexity of combinatorial circuits using “programmable” devices

In the next chapter the first order, memory circuits are introduced presenting

- how a simple loop allows the occurrence of the memory function
- the basic memory circuits: elementary lathes, clocked latches, master-slave flip-flops
- memories and registers as basic systems composed using the basic memory circuits
Belief #5: That qualitative as well as quantitative aspects of information systems will be accelerated by Moore’s Law. ... In the minds of some of my colleagues, all you have to do is identify one layer in a cybernetic system that’s capable of fast change and then wait for Moore’s Law to work its magic.

Jaron Lanier

The Moore’s Law applies to size not to complexity.

In this chapter we will forget for the moment about loops. Composition is the only mechanism involved in building a combinational digital system. No-loop circuits generate the class of history free digital systems whose outputs depend only by the current input variables, and are reassigned “continuously” at each change of inputs. Anytime the output results as a specific “combination” of inputs. No autonomy in combinational circuits, whose outputs obey “not to say a word” to inputs.

The combinational functions with \( n \) 1-bit inputs and \( m \) 1-bit outputs are called Boolean function and they have the following form:

\[
f : \{0,1\}^n \to \{0,1\}^m.
\]

For \( n = 1 \) only the NOT function is meaningful in the set of the 4 one-input Boolean functions. For \( n = 2 \) from the set of 16 different functions only few functions are currently used: AND, OR, XOR, NAND, NOR, NXOR. Starting with \( n = 3 \) the functions are defined only by composing 2-input functions. (For a short refresh see Appendix Boolean functions.)

Composing small gates results big systems. The growing process was governed in the last 40 years by Moore’s Law\(^2\). For a few more decades maybe the same growing law will act. But, starting from millions of gates per chip, it is very important what kind of circuits grow exponentially!

Composing gates results two kinds of big circuits. Some of them are structured following some repetitive patterns, thus providing simple circuits. Others grow patternless, providing complex circuits.

### 6.1 Simple, Recursive Defined Circuits

The first circuits used by designers were small and simple. When they were grew a little they were called big or complex. But, now when they are huge we must talk, more carefully, about big sized simple circuits or about big sized complex circuits. In this section we will talk about simple circuits which can be actualized at any size, i.e., their definitions don’t depend by the number, \( n \), of their inputs.

In the class of \( n \)-inputs circuits there are \( 2^{2^n} \) distinct circuits. From this tremendous huge number of logical function we use currently an insignificant small number of simple functions. What is strange is that these functions are sufficient for almost all the problem which we are confronted (or we are limited to be confronted).

---

1Jaron Lanier coined the term virtual reality. He is a computer scientist and a musician.

2The Moore’s Law says the physical performances in microelectronics improve exponentially in time.
One fact is clear: we cannot design very big complex circuits because we cannot specify them. The complexity must get away in another place (we will see that this place is the world of symbols). If we need big circuit they must remain simple.

In this section we deal with simple, if needed big, circuits and in the next with the complex circuits, but only with ones having small size.

From the class of the simple circuits we will present only some very usual such as decoders, demultiplexors, multiplexors, adders and arithmetic-logic units. There are many other interesting and useful functions. Many of them are proposed as problems at the end of this chapter.

### 6.1.1 Decoders

The simplest problem to be solved with a combinational logic circuit (CLC) is to answer the question: “what is the value applied to the input of this one-input circuit?” The circuit which solves this problem is an elementary decoder (EDCD). It is a decoder because decodes its one-bit input value by activating distinct outputs for the two possible input values. It is elementary because does this for the smallest input word: the one-bit word. By decoding, the value applied to the input of the circuit is emphasized activating distinct signals (like lighting only one of 7 bulbs). This is one of the main functions in a digital system. Before generating an answer to the applied signal, the circuit must “know” what signal arrived on its inputs.

**Informal definition**

The \( n \)-input decoder circuit – \( DCD_n \) – (see Figure 6.1) performs one of the basic function in digital systems: with one of its \( m \) one-bit outputs specifies the binary configuration applied on its inputs. The binary number applied on the inputs of \( DCD_n \) takes values in the set \( X = \{0, 1, ... 2^n - 1\} \). For each of these values there is one output \( y_0, y_1, ... y_{m-1} \) – which is activated on 1 if its index corresponds with the current input value. If, for example, the input of a \( DCD_4 \) takes value 1010, then \( y_{10} = 1 \) and the rest 15 one-bit outputs take the value 0.

![Figure 6.1: The \( n \)-input decoder (\( DCD_n \)).](image)

**Formal definition**

In order to rigorously describe and to synthesize a decoder circuit a formal definition is requested. Using Verilog HDL, such a definition is very compact certifying the non-complexity of this circuit.

**Definition 6.1** \( DCD_n \) is a combinational circuit with the \( n \)-bit input \( X, x_{n-1}, ... , x_0 \), and the \( m \)-bit output \( Y, y_{m-1}, ... , y_0 \), where: \( m = 2^n \), with the behavioral Verilog description:
module dec #(parameter inDim = n)(input [inDim - 1:0] sel, 
             output [(1 << inDim) - 1:0] out);
    assign out = 1 << sel;
endmodule

The previous Verilog description is synthesizable by the current software tools which provide an efficient solution. It happens because this function is simple and it is frequently used in designing digital systems.

Recursive definition
The decoder circuit \( DCD_n \) for any \( n \) can be defined recursively in two steps:

- defining the elementary decoder circuit \( EDCD = DCD_1 \) as the smallest circuit performing the decode function
- applying the divide & impera rule in order to provide the \( DCD_n \) circuit using \( DCD_{n/2} \) circuits.

For the first step \( EDCD \) is defined as one of the simplest and smallest logical circuits. Two one-input logical function are used to perform the decoding. Indeed, parallel composing (see Figure 6.2a) the circuits performing the simplest functions: \( f_2(x_0) = y_1 = x_0 \) (identity function) and \( f_1(x_0) = y_0 = x'_0 \) (NOT function), we obtain an \( EDCD \). If the output \( y_0 \) is active, it means the input is zero. If the output \( y_1 \) is active, then the input has the value 1.

\[ \text{Figure 6.2: The elementary decoder (EDCD). a. The basic circuit. b. Buffered EDCD, a serial-parallel composition.} \]

In order to isolate the output from the input the buffered \( EDCD \) version is considered serial composing an additional inverter with the previous circuit (see Figure 6.2b). Hence, the fan-out of \( EDCD \) does not depend on the fan-out of the circuit that drives the input.

The second step is to answer the question about how can be build a \( (DCD_n) \) for decoding an \( n \)-bit input word.

**Definition 6.2** The structure of \( DCD_n \) is recursive defined by the rule represented in Figure 6.3. The \( DCD_1 \) is an \( EDCD \) (see Figure 6.2b).

The previous definition is a constructive one, because provide an algorithm to construct a decoder for any \( n \). It falls into the class of the “divide & impera” algorithms which reduce the solution of the problem for \( n \) to the solution of the same problem for \( n/2 \).

The quantitative evaluation of \( DCD_n \) offers the following results:

**Size:**

\[ GS_{DCD}(n) = 2^n GS_{AND}(2) + 2 GS_{DCD}(n/2) = 2(2^n + GS_{DCD}(n/2)) \]
\[ GS_{DCD}(1) = GS_{EDCD} = 2 \]
\[ GS_{DCD}(n) \in O(2^n) \]
6.1. SIMPLE, RECURSIVE DEFINED CIRCUITS

Figure 6.3: The recursive definition of n-inputs decoder \( (DCD_n) \). Two \( DCD_{n/2} \) are used to drive a two dimension array of \( AND_2 \) gates. The same rule is applied for the two \( DCD_{n/2} \) and so on until \( DCD_1 = EDCD \) is needed.

**Depth:**
\[
D_{DCD}(n) = D_{AND}(2) + D_{DCD}(n/2) = 1 + D_{DCD}(n/2) = O(\log n)
\]
\[
D_{DCD}(1) = D_{EDCD} = 2
\]

**Complexity:** \( C_{DCD} \in O(1) \) because the definition occupies a constant drawn area (Figure 6.3) or a constant number of symbols in the Verilog description for any \( n \).

The size, the complexity and the depth of this version of decoder is out of discussion because the order of the size can not be reduced under the number of outputs \( (m = 2^n) \), for complexity \( O(1) \) is the minimal order of magnitude, and for depth \( O(\log n) \) is optimal taking into account we applied the "divide & impera" rule to build the structure of the decoder.

**Non-recursive description**

An iterative structural version of the previous recursive constructive definition is possible, because the outputs of the two \( DCD_{n/2} \) from Figure 6.3 are also 2-input AND circuits, the same as the circuits on the output level. In this case we can apply the associative rule, implementing the last two levels by only one level of 4-input ANDs. And so on, until the output level of the \( 2^n \) n-input ANDs is driven by \( n \) EDCDs. Now we have the decoder represented in Figure 6.4. Apparently it is a constant depth circuit, but if we take into account that the number of inputs in the AND gates is not constant, then the depth is given by the depth of an n-input gate which is in \( O(\log n) \). Indeed, an n-input AND has an efficient implementation as a binary tree of 2-input ANDs.

This "constant depth" DCD version – CDDCD – is faster than the previous for small values of \( n \) (usually for \( n < 6 \); for more details see Appendix Basic circuits), but the size becomes \( S_{CDDCD}(n) = n \times 2^n + 2n \in O(n2^n) \). The price is over-dimensioned related to the gain, but for small circuits sometimes it can be accepted.

The pure structural description for \( DCD_3 \) is:

```verilog
module dec3(output [7:0] out,
            input [2:0] in );
```
Figure 6.4: “Constant depth” DCD Applying the associative rule into the hierarchical network of $AND_2$ gates results the one level $AND_n$ gates circuit driven by $n$ EDCDs.

```
// internal connections
wire in0, nin0, in1, nin1, in2, nin2;
// EDCD for in[0]
not not00(nin0, in[0]);
not not01(in0, nin0);
// EDCD for in[1]
not not10(nin1, in[1]);
not not11(in1, nin1);
// EDCD for in[2]
not not20(nin2, in[2]);
not not21(in2, nin2);
// the second level
and and0(out[0], nin2, nin1, nin0); // output 0
and and1(out[1], nin2, nin1, in0); // output 1
and and2(out[2], nin2, in1, nin0); // output 2
and and3(out[3], nin2, in1, in0); // output 3
and and4(out[4], in2, nin1, nin0); // output 4
and and5(out[5], in2, nin1, in0); // output 5
and and6(out[6], in2, in1, nin0); // output 6
and and7(out[7], in2, in1, in0); // output 7
endmodule
```

For $n = 3$ the size of this iterative version is identical with the size which results from the recursive definition. There are meaningful differences only for big $n$. In real designs we do not need this kind of pure structural descriptions because the current synthesis tools manage very well even pure behavioral descriptions such that from the formal definition of the decoder.

**Arithmetic interpretation**

The decoder circuit is also an arithmetic circuit. It computes the numerical function of exponentiation: $Y = 2^X$. Indeed, for $n = i$ only the output $y_i$ takes the value 1 and the rest of the outputs take the value 0. Then, the number represented by the binary configuration $Y$ is $2^i$.

**Application**

Because the expressions describing the $m$ outputs of $DCD_n$ are:
6.1. SIMPLE, RECURSIVE DEFINED CIRCUITS

\[ y_0 = x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_1 \cdot x_0 \]
\[ y_1 = x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_2 \cdot x_1 \cdot x_0 \]
\[ y_2 = x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_3 \cdot x_2 \cdot x_1 \cdot x_0 \]

\[ \vdots \]
\[ y_{m-2} = x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_{m-2} \cdot x_{m-1} \cdot x_0 \]
\[ y_{m-1} = x_{n-1} \cdot x_{n-2} \cdot \ldots \cdot x_{m-1} \cdot x_0 \]

the logic interpretation of these outputs is that they represent all the min-terms for an \( n \)-input function. Therefore, any \( n \)-input logic function can be implemented using a \( DCD_n \) and an \( OR \) with maximum \( m - 1 \) inputs.

**Example 6.1** Let be the 3-input 2-output function defined in the table from Figure 6.5. A \( DCD_3 \) is used to compute all the min-terms of the 3 variables \( a \), \( b \), and \( c \). A 3-input OR is used to “add” the min-terms for the function \( X \), and a 4-input OR is used to “add” the min-terms for the function \( Y \).

\[
\begin{array}{ccc|cc}
 a & b & c & X & Y \\
 0 & 0 & 0 & 0 & 1 \\
 0 & 1 & 1 & 1 & 1 \\
 0 & 0 & 1 & 1 & 0 \\
 0 & 1 & 0 & 0 & 0 \\
 0 & 1 & 1 & 0 & 1 \\
 1 & 0 & 1 & 0 & 0 \\
 1 & 0 & 1 & 1 & 0 \\
 1 & 1 & 0 & 1 & 1 \\
 1 & 1 & 0 & 0 & 1 \\
\end{array}
\]

![Figure 6.5:](image)

Each min-term is computed only once, but it can be used as many times as the implemented functions suppose.

\[ \diamond \]

6.1.2 Demultiplexors

The structure of the decoder is included in the structure of the other usual circuits. Two of them are the demultiplexer circuit and the multiplexer circuit. These complementary functions are very important in digital systems because of their ability to perform “communication” functions. Indeed, demultiplexing means to spread a signal from a source to many destinations, selected by a binary code and multiplexing means the reverse operation to catch signals from distinct sources also selected using a selection code. Inside of both circuits there is a decoder used to identify the source of the signal or the destination of the signal by decoding the selection code.

**Informal definition**

The first informally described solution for implementing the function of an \( n \)-input demultiplexer is to use a decoder with the same number of inputs and \( m \) 2-input AND connected as in Figure 6.6. The value of the input \( enable \) is generated to the output of the gate opened by the activated output of the decoder \( DCD_n \). It is obvious that a \( DCD_n \) is a \( DMUX_n \) with \( enable = 1 \). Therefore, the size, depth of DMUXs are the same as for DCDs, because the depth is incremented by 1 and to the size is added a value which is in \( O(2^n) \).

For example, if on the selection input \( X = s \), then the outputs \( y_i \) take the value 0 for \( i \neq s \) and \( y_s = enable \). The inactive value on the outputs of this DMEX is 0.
CHAPTER 6. GATES: ZERO ORDER, NO-LOOP DIGITAL SYSTEMS

Figure 6.6: Demultiplexor. The \( n \)-input demultiplexor (DMUX\(_n\)) includes a \( DCD_n \) and \( 2^n \) AND\(_2\) gates used to distribute the input \( enable \) in \( 2^n \) different places according to the \( n \)-bit selection code.

**Formal definition**

**Definition 6.3** The \( n \)-input demultiplexor – DMUX\(_n\) – is a combinational circuit which transfers the 1-bit signal from the input \( enable \) to the one of the outputs \( y_{m-1}, \ldots, y_0 \) selected by the \( n \)-bit selection code \( X = x_n-1, \ldots, x_0 \), where \( m = 2^n \). It has the following behavioral Verilog description:

```verilog
module dmux #(parameter inDim = n)(input [inDim - 1:0] sel ,
                       input enable,
                       output [(1 << inDim) - 1:0] out );
assign out = enable << sel;
endmodule
```

**Recursive definition**

The DMUX circuit has also a recursive definition. The smallest DMUX, the elementary DMUX – EDMUX –, is a 2-output one, with a one-bit selection input. EDMUX is represented in Figure 6.7. It consists of an EDCD used to select, with its two outputs, the way for the signal \( enable \). Thus, the EDMUX is a circuit that offers the possibility to transfer the same signal (\( enable \)) in two places (\( y_0 \) and \( y_1 \)), according with the selection input (\( x_0 \)) (see Figure 6.7).

Figure 6.7: The elementary demultiplexor. a. The internal structure of an elementary demultiplexor (EDMUX) consists in an elementary decoder, 2 AND\(_2\) gates, and an inverter circuit as input buffer. b. The logic symbol.

The same rule – divide & impera – is used to define an \( n \)-input demultiplexor, as follows:
6.1. SIMPLE, RECURSIVE DEFINED CIRCUITS

Figure 6.8: **The recursive definition of DMUX\(_n\).** Applying the same rule for the two DMUX\(_{n-1}\) a new level of 2 EDMUXs is added, and the output level is implemented using 4 DMUX\(_{n-2}\). And so on until the output level is implemented using \(2^{n-1}\) EDMUXs. The resulting circuit contains \(2^n - 1\) EDMUXs.

**Definition 6.4** DMUX\(_n\) is defined as the structure represented in Figure 6.8, where the two DMUX\(_{n-1}\) are used to select the outputs of an EDMUX. ♦

If the recursive rule is applied until the end the resulting circuit is a binary tree of EDMUXs. It has \(S_{DMUX}(N) \in O(2^n)\) and \(D_{DMUX}(n) \in O(n)\). If this depth is considered too big for the current application, the recursive process can be stopped at a convenient level and that level is implemented with a “constant depth” DMUXs made using “constant depth” DCDs. The mixed procedures are always the best. The previous definition is a suggestion for how to use small DMUXs to build bigger ones.

6.1.3 Multiplexors

Now about the inverse function of demultiplexing: the multiplexing, i.e., to take a bit of information from a selected place and to send in one place. Instead of spreading by demultiplexing, now the multiplexing function gathers from many places in one place. Therefore, this function is also a communication function, allowing the interconnecting between distinct places in a digital system. In the same time, this circuit is very useful for implementing random, i.e. complex, logical functions, as we will see at the end of this chapter. More, in the next chapter we will see that the smallest multiplexor is used to build the basic memory circuits. Looks like this circuit is one of the most important basic circuit, and we must pay a lot of attention to it.

**Informal definition**

The direct intuitive implementation of a multiplexor with \(n\) selection bits – MUX\(_n\) – starts also from a DCD\(_n\) which is now serially connected with an AND-OR structure (see Figure 6.9). The outputs of the decoder open, for a given input code, only one AND gate that transfers to the output the corresponding selected input which, by turn, is OR-ed to the output \(y\).

Applying in this structure the associativity rule, for the AND gates to the output of the decoder and the supplementary added ANDs, results the actual structure of MUX. The structure AND-OR maintains the size and the depth of MUX in the same orders as for DCD.
**Figure 6.9: Multiplexer.** The $n$ selection inputs multiplexer $MUX_n$ is made serial connecting a $DCD_n$ with an AND-OR structure.

**Formal definition**

As for the previous two circuits – DCD and DMUX –, we can define the multiplexer using a behavioral (functional) description.

**Definition 6.5** A multiplexer $MUX_n$ is a combinational circuit having $n$ selection inputs $x_{n-1}, \ldots, x_0$ that selects to the output $y$ one input from the $m = 2^n$ selectable inputs, $i_{m-1}, \ldots, i_0$. The Verilog description is:

```verilog
module mux #(parameter inDim = n)(input [inDim-1:0] sel, // selection inputs
        input [(1<<inDim)-1:0] in , // selected inputs
        output out);
    assign out = in[sel];
endmodule
```

The MUX is obviously a simple function. Its formal description, for any number of inputs has a constant size. The previous behavioral description is synthesisable efficiently by the current software tools.

**Recursive definition**

There is also a rule for composing large MUXs from the smaller ones. As usual, we start from an elementary structure. The elementary MUX – EMUX – is a selector that connects the signal $i_1$ or $i_0$ in $y$ according to the value of the selection signal $x_0$. The circuit is presented in Figure 6.10a, where an EDCD with the input $x_0$ opens only one of the two ANDs "added" by the OR circuit in $y$. Another version for EMUX uses tristate inverting drivers (see Figure 6.10c).

The definition of $MUX_n$ starts from EMUX, in a recursive manner. This definition will show us that MUX is also a simple circuit ($C_{MUX}(n) \in O(1)$). In the same time this recursive definition will be a suggestion for the rule that composes big MUXs from the smaller ones.
6.1. SIMPLE, RECURSIVE DEFINED CIRCUITS

Figure 6.10: The elementary multiplexer (EMUX). a. The structure of EMUX containing an EDCD and the smallest AND-OR structure. b. The logic symbol of EMUX. c. A version of EMUX using transmission gates (see section Basic circuits).

Definition 6.6 $MUX_n$ can be made by serial connecting two parallel connected $MUX_{n/2}$ with an EMUX (see Figure 6.11 that is part of the definition), and $MUX_1 = EMUX$. ◊

Figure 6.11: The recursive definition of $MUX_n$. Each $MUX_{n-1}$ has a similar definition (two $MUX_{n-2}$ and one EMUX), until the entire structure contains EMUXs. The resulting circuit is a binary tree of $2^n - 1$ EMUXs.

Structural aspects

This definition leads us to a circuit having the size in $O(2^n)$ (very good, because we have $m = 2^n$ inputs to be selected in $y$) and the depth in $O(n)$. In order to reduce the depth we can apply step by step the next procedure: for the first two levels in the tree of EMUXs we can write the equation

$$y = x_1(x_0i_3 + x'_0i_2) + x'_1(x_0i_1 + x'_0i_0)$$

that becomes

$$y = x_1x_0i_3 + x_1x'_0i_2 + x'_1x_0i_1 + x'_1x'_0i_0.$$
CHAPTER 6. GATES: ZERO ORDER, NO-LOOP DIGITAL SYSTEMS

Using this procedure two or more levels (but not too many) of gates can be reduced to one. Carefully applied this procedure accelerate the speed of the circuit.

Application

Because the logic expression of a \( n \) selection inputs multiplexor is:

\[
y = x_{n-1} \cdots x_1 x_0 \bar{m}_{i-1} + \cdots + x'_{n-1} \cdots x'_1 x_0 \bar{i}_1 + x'_{n-1} \cdots x'_1 x'_0 \bar{i}_0
\]

any \( n \)-input logic function is specified by the binary vector \( \{i_{m-1}, \ldots, i_1, i_0\} \). Thus any \( n \) input logic function can be implemented with a \( MUX_n \) having on its selected inputs the binary vector defining it.

Example 6.2 Let be function \( X \) defined in Figure 6.12 by its truth table. The implementation with a \( MUX_3 \) means to use the right side of the table as the defining binary vector.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 6.12:

\[x_0 \quad x_1 \quad x_2 \quad MUX_3 \]

\[x_0 \quad x_1 \quad x_2 \quad y \]

\[c \quad i_0 \quad i_1 \quad i_2 \quad i_3 \quad i_4 \quad i_5 \quad i_6 \quad i_7 \]

6.1.4 * Shifters

One of the simplest arithmetic circuit is a circuit able to multiply or to divided with a number equal with a power of 2. The circuit is called also shifter because these operations do not change the relations between the bits of the number, they change only the position of the bits. The bits are shifted a number of positions to the left, for multiplication, or to the right for division.

The circuit used for implementing a shifter for \( n \)-bit numbers with \( m-1 \) positions is an \( m \)-input multiplexor having the selected inputs defined on \( n \) bits.

In the previous subsection were defined multiplexors having 1-bit selected inputs. How can be expanded the number of bits of the selected inputs? An elementary multiplexor for \( p \)-bit words, \( pEMUX \), is made using \( p \) \( EMUX \)s connected in parallel. If the two words to be multiplexed are \( a_{p-1}, \ldots, a_0 \) and \( b_{p-1}, \ldots, b_0 \), then each \( EMUX \) is used to multiplex a pair of bits \( (a_i, b_i) \). The one-bit selection signal is shared by the \( p \) \( EMUX \)s. \( nEMUX \) is a parallel extension of \( EMUX \).

Using \( pEMUX \)s an \( pMUX_n \) can be designed using the same recursive procedure as for designing \( MUX_n \) starting from \( EMUX \)s.

An \( 2^n - 1 \) positions left shifter for \( p \)-bit numbers, \( pLSHIFT_n \), is designed connecting the selected inputs of an \( pEMUX, i_0, \ldots, i_{m-1} \) where \( m = 2^n \), to the number to be shifted \( N = \{a_{n-1}, a_{n-2}, \ldots, a_0\} \) (\( a_i \in \{0,1\} \) for \( i = 0,1,\ldots,(m-1) \)) in \( 2^n - 1 \) ways, according to the following rule:

\[
i_j = \{\{a_{n-j-1}, a_{n-j-2}, \ldots, a_0\}, \{j[0]\}\}
\]

for: \( j = 0,1,\ldots,(m-1) \).
6.1. SIMPLE, RECURSIVE DEFINED CIRCUITS

Example 6.3 For $4LSHIFT_2$ an 4MUX_2 is used. The binary code specifying the shift dimension is shiftDim[1:0], the number to be shifted is in[3:0], and the ways the selected inputs, in0, in1, in2, in3, are connected to the number to be shifted are:

- $in0 = \{in[3], in[2], in[1], in[0]\}$
- $in1 = \{in[2], in[1], in[0], 0 \}$
- $in2 = \{in[1], in[0], 0, 0 \}$
- $in3 = \{in[0], 0, 0, 0 \}$

Figure 6.14 represents the circuit.

The Verilog description of the shifter is done by instantiating a 4-way 4-bit multiplexor with its inputs connected according to the previously described rule.

```verilog
module leftShifter(output [3:0] out ,
    input [3:0] in ,
    input [1:0] shiftDim);
    mux4_4 shiftMux(.out(out ),
        .in0(in ),
        .in1(in[2:0], 1'b0),
        .in2(in[1:0], 2'b0),
        .in3(in[0], 3'b0),
        .sel(shiftDim ));
endmodule
```

The multiplexor used in the previous module is built using 3 instantiations of an elementary 4-bit multiplexors. Results the two level tree of elementary multiplexors interconnected as the following Verilog code describes.

```verilog
module mux4_4(output [3:0] out ,
    input [3:0] in0, in1, in2, in3,
    input [1:0] sel); // 4-way 4-bit multiplexor (4MUX_2)
    wire[3:0] out1, out0; // internal connections between the two levels
    mux2_4 mux(out, out0, out1, sel[1]), // output multiplexor
    mux1(out1, in2, in3, sel[0]), // input multiplexor for in3 and in2
    mux0(out0, in0, in1, sel[0]); // input multiplexor for in1 and in0
endmodule
```

Any n-bit elementary multiplexor is described by the following parameterized module:
module mux2_4 #(parameter n = 4)(output [n-1:0] out,  
    input [n-1:0] in0, in1,  
    input sel); // 2-way 4-bit mux: 4EMUX  
    assign out = sel ? in1 : in0; // if (sel) then in1, else in0  
endmodule

The same idea helps us to design a special kind of shifter, called \textit{barrel shifter} which performs a \textit{rotate} operation described by the following rule: if the input number is $N = \{a_n-1, a_{n-2}, \ldots a_0\}$ ($a_i \in \{0, 1\}$ for $i = 0, 1, \ldots (m - 1)$), then rotating it with $i$ positions will provide:

$$i_i = \{a_{n-i-1}, a_{n-i-2}, \ldots a_0, a_{n-1}, a_{n-2}, \ldots a_{n-i}\}$$

for $i = 0, 1, \ldots (m - 1)$. This \textit{first solution} for the rotate circuit is very similar with the shift circuit. The only difference is: all the inputs of the multiplexor are connected to an input value. No 0s on any inputs of the multiplexor.

A \textit{second solution} uses only elementary multiplexors. A version for 8-bit numbers is presented in the following Verilog code.

module leftRotate(output [7:0] out ,  
    input [7:0] in ,  
    input [2:0] rotateDim);  
    wire [7:0] out0, out1;  
    mux2_8  
        level0(.out(out0), .in0(in ), .ini({in[6:0], in[7]} ), .sel(rotateDim[0])),  
        level1(.out(out1), .in0(out0), .ini({out0[5:0], out0[7:6]}), .sel(rotateDim[1])),  
        level2(.out(out0), .in0(out1), .ini({out1[3:0], out1[7:4]}), .sel(rotateDim[2]));  
endmodule

module mux2_8(output [7:0] out,  
    input [7:0] in0, in1,  
    input sel);  
    assign out = sel ? in1 : in0;  
endmodule

While the \textit{first solution} uses for $n$ bit numbers $n \text{MUX}_{\log_2(\text{rotateDim})}$, the \textit{second solution} uses $\log_2(\text{rotateDim}) \ n\text{EMUX}$s. Results:

$$S_{\text{firstSolutionOfLeftRotate}} = (n \times (\text{rotateDim} - 1)) \times S_{\text{EMUX}}$$

$$S_{\text{secondSolutionOfLeftRotate}} = (n \times \log_2(\text{rotateDim})) \times S_{\text{EMUX}}$$
6.1.5 *Priority encoder*

An encoder is a circuit which connected to the outputs of a decoder provides the value applied on the input of the decoder. As we know only one output of a decoder is active at a time. Therefore, the encoder compute the index of the activated output. But, a real application of an encoder is to encode binary configurations provided by any kind of circuits. In this case, more than one input can be active and the encoder must have a well defined behavior. One of this behavior is to encode the most significant bit and to ignore the rest of bits. For this reason the encoder is a priority encoder.

The \( n \)-bit input, enabled priority encoder circuit, \( PE(n) \), receives \( x_{n-1}, x_{n-2}, \ldots, x_0 \) and, if the enable input is activated, \( en = 1 \), it generates the number \( Y = y_{m-1}, y_{m-2}, \ldots, y_0 \), with \( n = 2^m \), where \( Y \) is the biggest index associated with \( x_i = 1 \) if any, else zero output is activated. (For example: if \( en = 1 \), for \( n = 8 \), and \( x_7, x_6, \ldots, x_0 = 00110001 \), then \( y_2, y_1, y_0 = 101 \) and \( zero = 0 \)) The following Verilog code describe the behavior of \( PE(n) \).

```verilog
module priority_encoder #(parameter m = 3)(input [(1'b1<<m)-1:0] in ,
       input enable ,
       output reg [m-1:0] out ,
       output reg zero );

    integer i;
    always @(*) if (enable) begin out = 0;
          for(i=(1'b1 << m)-1; i>=0; i=i-1)
            if ((out == 0) && in[i]) out = i;
          if (in == 0) zero = 1;
          else zero = 0;
    end

    else begin out = 0;
          zero = 1;
    end

endmodule
```

For testing the previous description the following test module is used:

```verilog
module test_priority_encoder #(parameter m = 3);
    reg [(1'b1<<m)-1:0] in ;
    reg enable ;
    wire [m-1:0] out ;
    wire zero ;
    initial begin enable = 0;
          in = 8'b11111111;
          #1 enable = 1;
          #1 in = 8'b00000001;
          #1 in = 8'b0000001x;
          #1 in = 8'b000001xxx;
          #1 in = 8'b0001xxxxx;
          #1 in = 8'b01xxxxxx;
          #1 in = 8'b1xxxxxxxx;
          #1 in = 8'b110;
          #1 $stop;
    end
    priority_encoder dut(in ,
                       enable ,
```
CHAPTER 6. GATES: ZERO ORDER, NO-LOOP DIGITAL SYSTEMS

out ,
zero );
initial $monitor ($time, "enable=%b in=%b out=%b zero=%b",
           enable, in, out, zero);
endmodule

Running the previous code the simulation provides the following result:

<table>
<thead>
<tr>
<th>Time</th>
<th>Enable</th>
<th>Input</th>
<th>Output</th>
<th>Zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11111111</td>
<td>000</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111111</td>
<td>111</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>00000001</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0000001x</td>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>000001xx</td>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0001xxxx</td>
<td>011</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>001xxxxx</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>01xxxxxx</td>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1xxxxxxx</td>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0xxxxxxx</td>
<td>111</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00000110</td>
<td>010</td>
<td>0</td>
</tr>
</tbody>
</table>

It is obvious that this circuit computes the integer part of the base 2 logarithm. The output zero is used to notify that the input value is inappropriate for computing the logarithm, and “prevent” us from taking into account the output value.

6.1.6 Prefix computation network

There is a class of circuits, called prefix computation networks, $PCN_{\text{func}}(n)$, defined for $n$ inputs and having the characteristic function $\text{func}$. If $\text{func}$ is expressed using the operation $\circ$, then the function of $PCN_{\text{func}}(n)$ is performed by a circuit having the inputs $x_0, \ldots, x_{n-1}$ and the outputs $y_0, \ldots, y_{n-1}$ related as follows:

$$
\begin{align*}
    y_0 &= x_0 \\
    y_1 &= x_0 \circ x_1 \\
    y_2 &= x_0 \circ x_1 \circ x_2 \\
    \vdots \\
    y_{n-1} &= x_0 \circ x_1 \circ \ldots \circ x_{n-1}
\end{align*}
$$

where the operation "$\circ$" is an associative and commutative operation. For example, $\circ$ can be the arithmetic operation $\text{add}$, or the logic operation $\text{AND}$. In the first case $x_i$ is an $m$-bit binary number, and in the second case it is a 1-bit Boolean variable.

Example 6.4 If $\circ$ is the Boolean function $\text{AND}$, then $PCN_{\text{AND}}(n)$ is described by the following behavioral description:

```verilog
module and_prefixes #(parameter n = 64)(input [0:n-1] in ,
                        output reg [0:n-1] out);
    integer k;
    always @(in) begin out[0] = in[0];
              for (k=1; k<n; k=k+1) out[k] = in[k] & out[k-1];
    end
endmodule
```
There are many solutions for implementing $PCN_{\text{AND}}(n)$. If we use AND gates with up to $n$ inputs, then there is a first direct solution for $PCN_{\text{AND}}$ starting from the defining equations (it consists in one 2-input gate, plus one 3-input gate, ... plus one $(n-1)$-input gate). A very large high-speed circuit is obtained. Indeed, this direct solution offers a circuit with the size $S(n) \in O(n^2)$ and the depth $D(n) \in O(1)$. We are very happy about the speed (depth), but the price paid for this is too high: the squared size. In the same time our design experience tells us that this speed is not useful in current applications because of the time correlations with other subsystems. (There is also a discussion about gate having $n$ inputs. These kind of gates are not realistic.)

A second solution offers a very good size but a too slow circuit. If we use only 2-input AND gates, then the definition becomes:

$$
\begin{align*}
  y_0 &= x_0 \\
  y_1 &= y_0 \& x_1 \\
  y_2 &= y_1 \& x_2 \\
  &\vdots \\
  y_{n-1} &= y_{n-2} \& x_{n-1}
\end{align*}
$$

A direct solution starting from this new form of the equations (as a degenerated binary tree of ANDs) has $S(n) \in O(n)$ and $D(n) \in O(n)$. This second solution is also very inefficient, now because of the speed which is too low.

The third implementation is a optimal one. For $PCN_{\text{AND}}(n)$ is used the recursive defined network represented in Figure 6.15 [Ladner '80], where in each node, for our application, there is a 2-inputs AND gate. If $PCN_{\text{AND}}(n/2)$ is a well-functioning prefix network, then all the structure works as a prefix network. Indeed, $PCN_{\text{AND}}(n/2)$ computes all even outputs because of the $n/2$ input circuits that perform
the \( y_2 \) function between successive pairs of inputs. On the output level the odd outputs are computed using even outputs and odd inputs. The \( PCN_{\text{AND}}(n/2) \) structure is built upon the same rule and so on until \( PCN_{\text{AND}}(1) \) that is a system without any circuit. The previous recursive definition is applied for \( n = 16 \) in Figure 6.16.

The size of \( PCN_{\text{AND}}(n) \), \( S(n) \) (with \( n \) a power of 2), is evaluated starting from: \( S(1) = 0, S(n) = S(n/2) + (n - 1)S_{y_1} \) where \( S_{y_1} \) is the size of the elementary circuit that defines the network (in our case is a 2-inputs AND gate). The next steps leads to:

\[
S(n) = S(n/2^i) + (n/2^{i-1} + \ldots + n/2^0 - i)S_{y_1}
\]

and ends, for \( i = \log_2 n \) (the rule is recursively applied \( \log n \) times), with:

\[
S(n) = (2n - 2 - \log_2 n)S_{y_1} \in O(n).
\]

(The network consists in two binary trees of elementary circuits. The first with the bottom root having \( n/2 \) leaves on the first level. The second with the top root having \( n/2 - 1 \) leaves on the first level. Therefore, the first tree has \( n - 1 \) elementary circuits and the second tree has \( n - 1 - \log n \) elementary circuits.) The depth is \( D(n) = 2D_{y_1}\log_2 n \in O(\log n) \) because \( D(n) = D(n/2) + 2 \) (at each step two levels are added to the system).

---

**Figure 6.16: PCN\textsubscript{AND}(16)**
6.1.7 Increment circuit

The simplest arithmetic operation is the increment. The combinational circuit performing this function receives an \( n \)-bit number, \( x_{n-1}, \ldots, x_0 \), and a one-bit command, \( \text{inc} \), enabling the operation. The outputs, \( y_{n-1}, \ldots, y_0 \), and \( cr_{n-1} \) behaves according to the value of the command:

- If \( \text{inc} = 1 \), then
  \[
  \{ cr_{n-1}, y_{n-1}, \ldots, y_0 \} = \{ x_{n-1}, \ldots, x_0 \} + 1
  \]
- else
  \[
  \{ cr_{n-1}, y_{n-1}, \ldots, y_0 \} = \{ 0, x_{n-1}, \ldots, x_0 \}.
  \]

Figure 6.17: Increment circuit. a. The elementary increment circuit (called also half adder). b. The recursive definition for an \( n \)-bit increment circuit.

The increment circuit is built using as “brick” the elementary increment circuit, \( \text{EINC} \), represented in Figure 6.17a, where the XOR circuit generate the increment of the input if \( \text{inc} = 1 \) (the current bit is complemented), and the circuit AND generate the carry for the the next binary order (if the current bit is incremented and it has the value 1). An \( n \)-bit increment circuit, \( \text{INC}_n \), is recursively defined in Figure 6.17b: \( \text{INC}_n \) is composed using an \( \text{INC}_{n-1} \) serially connected with an \( \text{EINC} \), where \( \text{INC}_0 = \text{EINC} \).

6.1.8 Adders

Another usual digital functions is the sum. The circuit associated to this function can be also made starting from a small elementary circuits, which adds two one-bit numbers, and looking for a simple recursive definitions for \( n \)-bit numbers.

The elementary structure is the well known full adder which consists in two half adders and an OR. An \( n \)-bit adder could be done in a recursive manner as the following definition says.

**Definition 6.7** The full adder, \( \text{FA} \), is a circuit which adds three 1-bit numbers generating a 2-bit result:

\[
\text{FA}(\text{in}1, \text{in}2, \text{in}3) = \{ \text{out}1, \text{out}0 \}
\]

\( \text{FA} \) is used to build \( n \)-bit adders. For this purpose its connections are interpreted as follows:

- \( \text{in}1, \text{in}2 \) represent the \( i \)-th bits if two numbers
• \( \text{in}3 \) represents the carry signal generated by the \( i - 1 \) stage of the addition process

• \( \text{out}0 \) represents the \( i \)-th bit of the result

• \( \text{out}1 \) represents the carry generated for the \( i + 1 \)-th stage of the addition process

Follows the Verilog description:

```verilog
module full_adder(output sum, carry_out, input in1, in2, carry_in);
    half_adder ha1(sum1, carry1, in1, in2),
        ha2(sum, carry2, sum1, carry_in);
    assign carry_out = carry1 | carry2;
endmodule

module half_adder(output sum, carry, input in1, in2);
    assign sum = in1 ^ in2,
        carry = in1 & in2;
endmodule

⋄

Note: The half adder circuit is also an elementary increment circuit (see Figure 6.17a).

**Definition 6.8** The \( n \)-bits ripple carry adder, \( \text{ADD}_n \), is made by serial connecting on the carry chain an \( \text{ADD}_{n-1} \) with a FA (see Figure 6.18). \( \text{ADD}_1 \) is a full adder.

```verilog
module adder #(parameter n = 4)(output [n-1:0] out ,
    output cry ,
    input [n-1:0] in1 ,
    input [n-1:0] in2 ,
    input cin);
    wire [n:1] carry ;
    assign cry = carry[n] ;
    generate
        if (n == 1) fullAdder firstAder(.out(out[0] ),
            .cry(carry[1] ),
            .in1(in1[0] ),
            .in2(in2[0] ),
            .cin(cin ));
        else begin adder #(n(n-1)) partAdder( .out(out[n-2:0] ),
            .cry(carry[n-1] ),
            .in1(in1[n-2:0] ),
            .in2(in2[n-2:0] ),
            .cin(cin ));
            fullAdder lastAdder(.out(out[n-1] ),
                .cry(carry[n] ),
                .in1(in1[n-1] ),
                .in2(in2[n-1] ),
                .cin(carry[n-1] ));
        end
endgenerate
```
module fullAdder( output out ,
output cry ,
input  in1 ,
input  in2 ,
input  cin);
assign cry = in1 & in2 | (in1 ^ in2) & cin ;
assign out = in1 ^ in2 ^ cin ;
endmodule

Figure 6.18: The recursive defined \(n\)-bit ripple-carry adder (\(ADD_n\)). \(ADD_n\) is simply designed adding to an \(ADD_{n-1}\) a full adder (FA), so as the carry signal ripples from one FA to the next.

The previous definition used the conditioned generation block.\(^3\) The Verilog code from the previous recursive definition can be used to simulate and to synthesize the adder circuit. For this simple circuit this definition is too sophisticated. It is presented here only to provide a simple example of how a recursive definition is generated.

A simpler way to define an adder is provided in the next example where a generate block is used.

**Example 6.5 Generated \(n\)-bit adder:**

```verilog
module add #(parameter n=8)( input [n-1:0] in1, in2, 
input cIn ,
output [n-1:0] out ,
output cOut );
wire [n:0] cr ;
assign cr[0] = cIn ;
assign cOut = cr[n] ;
genvar i ;
generate for (i=0; i<n; i=i+1) begin: S
  fa adder( .in1 (in1[i] ),
           .in2 (in2[i] ),
           .cIn (cr[i] ),
           .out (out[i] ),
           .cOut (cr[i+1]));
endgenerate
```

\(^3\)The use of the conditioned generation block for recursive definition was suggested to me by my colleague Radu Hobincu.
end
dendgenerate
dendmodule

module fa( input in1, in2, cIn ,
              output out, cOut );
    wire xr ;
    assign xr = in1 ^ in2 ;
    assign out = xr ^ cIn ;
    assign cOut = in1 & in2 | cIn & xr ;
endmodule

Because the add function is very frequently used, the synthesis and simulation tools are able to "understand" the simplest one-line behavioral description used in the following module:

module add #(parameter n=8)( input [n-1:0] in1, in2,
                                  input cIn ,
                                  output [n-1:0] out ,
                                  output cOut );
    assign {cOut, out} = in1 + in2 + cIn ;
endmodule

Carry-Look-Ahead Adder

The size of $ADD_n$ is in $O(n)$ and the depth is unfortunately in the same order of magnitude. For improving the speed of this very important circuit there was found a way for accelerating the computation of the carry: the carry-look-ahead adder ($CLA_n$). The fast carry-look-ahead adder can be made using a carry-look-ahead (CL) circuit for fast computing all the carry signals $C_i$ and for each bit an half adder and a XOR (the modulo two adder)(see Figure 6.19). The half adder has two roles in the structure:

![Carry-Lookahead Circuit](image)

Figure 6.19: The fast $n$-bit adder. The $n$-bit Carry-Lookahead Adder ($CLA_n$) consists in $n$ HAs, $n$ 2-input XORs and the Carry-Lookahead Circuit used to compute faster the $n$ $C_i$, for $i = 1, 2, \ldots n$.

- sums the bits $A_i$ and $B_i$ on the output $S$
computes the signals \( G_i \) (that \textit{generates} carry as a local effect) and \( P_i \) (that allows the \textit{propagation} of the carry signal through the binary level \( i \)) on the outputs \( CR \) and \( P \).

The XOR gate adds modulo 2 the value of the carry signal \( C_i \) to the sum \( S \).

In order to compute the carry input for each binary order an additional fast circuit must be build: the \textit{carry-look-ahead circuit}. The equations describing it start from the next rule: \textit{the carry toward the level \((i+1)\) is \textbf{generated} if both \( A_i \) and \( B_i \) inputs are \textbf{1} or is \textbf{propagated} from the previous level if only one of \( A_i \) or \( B_i \) are \textbf{1}}. Results:

\[
C_{i+1} = A_i B_i + (A_i + B_i) C_i = A_i B_i + (A_i \oplus B_i) C_i = G_i + P_i C_i.
\]

Applying the previous rule we obtain the general form of \( C_{i+1} \):

\[
C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + P_i P_{i-1} P_{i-2} G_{i-3} + \ldots + P_i P_{i-1} \ldots P_1 P_0 C_0
\]

for \( i = 0, \ldots, n \).

Computing the size of the carry-look-ahead circuit results \( S_{\text{CLA}}(n) \in O(n^3) \), and the theoretical depth is only 2. But, for real circuits an \( n \)-input gates can not be considered as a one-level circuit. In \textit{Basic circuits} appendix (see section \textit{Many-Input Gates}) is shown that an optimal implementation of an \( n \)-input simple gate is realized as a binary tree of \( 2 \)-input gates having the depth in \( O(\log n) \). Therefore, in a real implementation the depth of a carry-look ahead circuit has \( D_{\text{CLA}} \in O(\log n) \).

For small \( n \) the solution with carry-look-ahead circuit works very good. But for larger \( n \) the two solutions, without carry-look-ahead circuit and with carry-look-ahead circuit, must be combined in many fashions in order to obtain a good price/performance ratio. For example, the ripple carry version of \( \text{ADD}_n \) is divided in two equal sections and two carry look-ahead circuits are built for each, resulting two serial connected \( \text{CLA}_{n/2} \). The state of the art in this domain is presented in [Omondi ’94].

It is obvious that the adder is a simple circuit. There exist constant sized definition for all the variants of adders.

\* \textbf{Prefix-Based Carry-Look-Ahead Adder}

Let us consider the expressions for \( C_1, C_2, \ldots C_i, \ldots \). It looks like each product from \( C_{i-1} \) is a prefix of a product from \( C_i \). This suggests a way to reduce the too big size of the carry-look-ahead circuit from the previous paragraph. Following [Cormen ’90] (see section 29.2.2), an optimal carry-look-ahead circuit (with asymptotically linear size and \( \log \) depth) is described in this paragraph. It is known as Brent-Kung adder.

Let be \( x_i \), the \textit{carry state}, the information used in the stage \( i \) to determine the value of the carry. The carry state takes three values, according to the table represented in Figure 6.20, where \( A_i \) and \( B_i \) are the \( i \)-th bits of the numbers to be added. If \( A_i = B_i = 0 \), then the carry bit is 0 (it is \textit{killed}). If \( A_i = B_i = 1 \), then the carry bit is 1 (it is \textit{generated}). If \( A_i = 0 \), and \( B_i = 1 \) or \( A_i = 1 \) and \( B_i = 0 \), then the carry bit is equal with the carry bit generated in the previous stage, \( C_{i-1} \) (the carry from the previous range \textit{propagates}). Therefore, in each binary stage the carry state has three values: \( k, p, g \).

We define the function \( \odot \) which composes the carry states of the two binary ranges. In Figure 6.21 the states \( x_i \) and \( x_{i-1} \) are composed generating the carry state of two successive bits, \( i - 1 \) an \( i \). If \( x_i = k \), then the resulting composed state is independent by \( x_{i-1} \) and it takes the value \( k \). If \( x_i = g \), then the resulting composed state is independent by \( x_{i-1} \) and it takes the value \( g \). If \( x_i = p \), then the resulting composed state \textit{propagates} and it is \( x_{i-1} \).

The carry state for the binary range \( i \) is determined using the expression:

\[
y_i = y_{i-1} \odot x_i = x_0 \odot x_1 \odot \ldots x_i
\]

starting with:

\[
y_0 = x_0 = \{k, g\}
\]
which is associated with the carry state used to specify the carry input \( C_0 \) (for \( k \) is no carry, for \( g \) is carry, while \( p \) has no meaning for the input carry state).

Thus, carry states are computed as follows:

\[
\begin{align*}
y_0 &= x_0 \\
y_1 &= x_0 \otimes x_1 \\
y_2 &= x_0 \otimes x_1 \otimes x_2 \\
\vdots \\
y_i &= x_0 \otimes x_1 \otimes x_2 \ldots x_i
\end{align*}
\]

It is obvious that we have a prefix computation. Let us call the function \( \otimes \) eCLA (elementary Carry-Look-Ahead). Then, the circuit used to compute the carry states \( y_0, y_1, y_2, \ldots, y_i \) is \( \text{prefixCLA} \).

**Theorem 6.1** If \( x_0 = \{k, g\} \), then \( y_1 = \{k, g\} \) for \( i = 1, 2, \ldots, i \) and the value of \( C_i \) is set to 0 by \( k \) and to 1 by \( g \).

**Proof:** in the table from Figure 6.21 if the line \( p \) is not selected, then the value \( p \) does not occur in any carry state. Because, \( x_0 = \{k, g\} \) the previous condition is fulfilled.

An appropriate coding of the three values of \( x \) will provide a simple implementation. We propose the coding represented in the table from Figure 6.22. The two bits used to code the state \( x \) are \( P \) and \( G \) (with the meaning used in the previous paragraph). Then, \( y_i[0] = C_i \).

\[
\begin{array}{c|cc}
  x_i & P & G \\
  \hline
  k & 0 & 0 \\
  p & 1 & 0 \\
  g & 0 & 1 \\
\end{array}
\]

**Figure 6.22:** Coding the carry state.

The first, direct form of the adder circuit, for \( n = 7 \), is represented in Figure 6.23, where there are three levels:
6.1. SIMPLE, RECURSIVE DEFINED CIRCUITS

- the input level of half-adders which compute \( x_i = \{ P_i, G_i \} \) for \( i = 1, 2, \ldots, 6 \), where:

  \[
P_i = A_{i-1} \oplus B_{i-1}
  
  G_i = A_{i-1} \cdot B_{i-1}
  \]

- the intermediate level of prefixCLA which computes the carry states \( y_1, y_2, \ldots, y_7 \) with a chain of eCLA circuits

- the output level of XORs used to compute the sum \( S_i \) starting from \( P_{i+1} \) and \( C_i \).

The size of the resulting circuit is in \( O(n) \), while the depth is in the same order of magnitude. The next step is to reduce the depth of the circuit to \( O(\log n) \).

The expressions for the two outputs of the eCLA circuit are:

\[
P_{\text{out}} = P_i \cdot P_{i-1}

G_{\text{out}} = G - i + P - i \cdot G_{i-1}
\]

For the pure serial implementation of the prefixCLA circuit from Figure 6.23, because \( x_0 = \{ 0, C_0 \} \), in each eCLA circuit \( P_{\text{out}} = 0 \). The full circuit will be used in the log-depth implementation of the prefixCLA circuit. Following the principle exposed in the paragraph Prefix Computation network the prefixCLA circuit is redesigned optimally for the adder represented in Figure 6.25. If we take from Figure 6.16 the frame labeled \( PN_{\text{out}}(8) \) and the 2-input AND circuits are substituted with eCLA circuits, then results the prefixCLA circuit from Figure 6.25.

**Important notice:** the eCLA circuit is not a symmetric one. The most significant inputs, \( P_i \) and \( G_i \) correspond to the \( i \)-th binary range, while the other two correspond to the previous binary range. This order should be taken into consideration when the log-depth prefixCLA (see Figure 6.25) circuit is organized.

While in the prefixCLA circuit from Figure 6.23 all eCLA circuits have the output \( P_{\text{out}} = 0 \), in the log-depth prefixCLA from Figure 6.25 some of them are fully implemented. Because \( x_0 = \{ 0, C_0 \} \), eCLA indexed with 0 and all the eCLAs enchain after it, indexed with 4, 6, 7, 8, 9, and 10, are of the simple form with \( P_{\text{out}} = 0 \). Only the eCLA indexed with 1, 2, 3, and 5 are fully implemented, because their inputs are not restricted.

This version of \( n \)-bit adder is asymptotically optimal, because the size of the circuit is in \( O(n) \) and the depth is in \( O(\log n) \). Various versions of this circuit are presented in [webRef_3].
CHAPTER 6. GATES: ZERO ORDER, NO-LOOP DIGITAL SYSTEMS

<table>
<thead>
<tr>
<th>$P_i$</th>
<th>$G_i$</th>
<th>$P_{i-1}$</th>
<th>$G_{i-1}$</th>
<th>$P_{out}$</th>
<th>$G_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 6.24: The logic table for eCLA.

* Carry-Save Adder

For adding $m$ $n$-bit numbers there is a faster solution than the one which supposes to use the direct circuit build as a tree of $m - 1$ 2-number adders. The depth and the size of the circuit is reduced using a carry-save adder circuit.

The carry save adder receives three $n$-bit numbers:

- $x = \{x_{n-1}, x_{n-2}, \ldots, x_1, x_0\}$
- $y = \{y_{n-1}, y_{n-2}, \ldots, y_1, y_0\}$
- $z = \{z_{n-1}, z_{n-2}, \ldots, z_1, z_0\}$

and generate two $(n + 1)$-bit numbers:

- $c = \{c_{n-1}, c_{n-2}, \ldots, c_1, c_0, 0\}$
- $s = \{0, s_{n-1}, s_{n-2}, \ldots, s_1, s_0\}$

where:

$$c + s = x + y + z.$$

The function of the circuit is described by the following transfer function applied for $i = 0, \ldots, n - 1$:

$$x_i + y_i + z_i = \{c_i, s_i\}.$$

The internal structure of the carry-save adders contains $n$ circuits performing the previous function which is the function of a full adder. Indeed, the binary variables $x_i$ and $y_i$ are applied on the two inputs of the FA, $z_i$ is applied on the carry input of the same FA, and the two inputs $c_i$, $s_i$ are the carry-out and the sum outputs. Therefore, an elementary carry-save adder, ECSA, has the structure of a FA (see Figure 6.26a).

To prove for the functionality of CSA we write for each ECSA:

$$\{c_{n-1}, s_{n-1}\} = x_{n-1} + y_{n-1} + z_{n-1}$$

$$\{c_{n-2}, s_{n-2}\} = x_{n-2} + y_{n-2} + z_{n-2}$$

...
6.1. SIMPLE, RECURSIVE DEFINED CIRCUITS

Figure 6.25: The adder with the prefix based carry-look-ahead circuit.

\[
\{c_1, s_1\} = x_1 + y_1 + z_1
\]

from which results that:

\[
x + y + z = \{c_{n-1}, s_{n-1}\} \times 2^{n-1} + \{c_{n-2}, s_{n-2}\} \times 2^{n-2} + \ldots + \{c_1, s_1\} \times 2^1 + \{c_0, s_0\} \times 2^0 = \\
\{c_{n-1}, c_{n-2}, \ldots, c_1, c_0\} \times 2 + \{s_{n-1}, s_{n-2}, \ldots, s_1, s_0\} = \{c_{n-1}, c_{n-2}, \ldots, c_1, c_0, 0\} + \{0, s_{n-1}, s_{n-2}, \ldots, s_1, s_0\}.
\]

Figure 6.26c shows a 4-number adder for \(n\)-bit numbers. Instead of 3 adders, 2 3CSA\(_n\) circuits and a \((n + 1)\)-bit adder are used. The logic symbol for the resulting 4-input reduction adder – 4REDA\(_n\) – is represented in Figure 6.26d. The depth of the circuit is of 2 FAs and an \((n + 1)\)-bit adder, instead of the depth associated with one \(n\)-bit adder and an \((n + 1)\)-bit adder. The size is also minimized if in the standard solution carry-look-ahead adders are used.

Example 6.6 The module 3CSA\(_m\) is generated using the following template:

```vhdl
module csa3_m #(parameter n=m)(input [n-1:0] in0, in1, in2 , output [n:0] sOut, cOut );
wire [n-1:0] out ;
```


Figure 6.26: Carry-Save Adder. a. The elementary carry-save adder. b. CSA for 3 \( n \)-bit numbers: \( 3\text{CSA}_n \). c. How to use carry-save adders to add 4 \( n \)-bit numbers. d. The logic symbol for a 4 \( n \)-bit inputs reduction adder (4REDA\(_n\)) implemented with two \( 3\text{CSA}_n \) and a \( (n+1) \)-bit adder.

```verilog
wire [n-1:0] cr;

genvar i;
generate for (i=0; i<n; i=i+1) begin: S
    fa adder(in0[i], in1[i], out[i], cr[i]);
end
degenerate

assign sOut = {1'b0, out} ;
assign cOut = {cr, 1'b0} ;
endmodule
```

where: (1) an actual value for \( m \) must be provided and (2) the module \( fa \) is defined in the previous example. ⚫

**Example 6.7** For the design of a 8 1-bit input adder (8REDA\(_1\)) the following modules are used:

```verilog
module reda8_1(output [3:0] out,
```

![Diagrams](image-url)
6.1. SIMPLE, RECURSIVE DEFINED CIRCUITS

```markdown
input [7:0] in);
wire [2:0] sout, cout;
csa8_1 csa( in[0],in[1],in[2],in[3],in[4],in[5],in[6],in[7],cout,sout);
assign out = cout + sout;
endmodule

module csa8_1( input in0, in1, in2, in3, in4, in5, in6, in7,
output [2:0] cout, sout );
wire [1:0] sout0, cout0, sout1, cout1;
csa4_1 csa0(in0, in1, in2, in3, sout0, cout0),
csa1(in4, in5, in6, in7, sout1, cout1);
csa4_2 csa2(sout0, cout0, sout1, cout1, cout, sout);
endmodule
```

where csa4_1 and csa4_2 are instances of the following generic module:

```markdown
module csa4_p( input [1:0] in0, in1, in2, in3 ,
output [2:0] sout, cout );
wire [2:0] s1, c1 ;
wire [3:0] s2, c2 ;
csa3_p csa0(in0, in1, in2, s1, c1);
csa3_q csa1(s1, c1, {1'b0, in3}, s2, c2);
assign sout = s2[2:0], cout = c2[2:0];
endmodule
```

for \( p = 1 \) and \( p = 2 \), while \( q = p + 1 \). The module csa3_m is defined in the previous example.

The efficiency of this method to add many numbers increases, compared to the standard solution, with the number of operands.

6.1.9 * Combinational Multiplier

One of the most important application of the CSAs circuits is the efficient implementation of a combinational multiplier. Because multiplying \( n \)-bit numbers means to add \( n \frac{2n - 1}{2} \)-bit partial products, a \( n\text{CSA}_{2n-1} \) circuit provides the best solution. Adding \( n \)-bit numbers using standard carry-adders is done, in the best case, in time belonging to \( O(n \times \log n) \) on a huge area we can not afford. The proposed solution provides a linear time.

In Figure 6.27 is an example for how for the binary multiplication for \( n = 4 \) works. Thus, the combinational multiplication is done in the following three stages:

```
<table>
<thead>
<tr>
<th></th>
<th>a3</th>
<th>a2</th>
<th>a1</th>
<th>a0</th>
<th></th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
<th></th>
<th>multiplicand: a</th>
<th>1 0 1 1</th>
<th></th>
<th>multiplier: b</th>
<th>1 1 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p03</td>
<td>p02</td>
<td>p01</td>
<td>p0</td>
<td></td>
<td>partial product: pp0</td>
<td>1 0 1 1</td>
<td></td>
<td>partial product: pp1</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P13</td>
<td>P12</td>
<td>P11</td>
<td>P10</td>
<td></td>
<td>partial product: pp2</td>
<td>1 0 1 1</td>
<td></td>
<td>partial product: pp3</td>
<td>1 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>p23</td>
<td>p22</td>
<td>P21</td>
<td>P20</td>
<td></td>
<td>final product: p</td>
<td>1 0 0 0 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P33</td>
<td>P32</td>
<td>P31</td>
<td>P30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 6.27: Multiplying 4-bit numbers.
1. compute $n$ partial products – $pp_0, \ldots, pp_{n-1}$ – using a two-dimension array of $n \times n$ AND$_2$ circuits (see in Figure 6.28 the “partial products computation” dashed box); for $n = 4$ the following relations describe this stage:

$$pp_0 = \{a_3 \cdot b_0, a_2 \cdot b_0, a_1 \cdot b_0, a_0 \cdot b_0\}$$

$$pp_1 = \{a_3 \cdot b_1, a_2 \cdot b_1, a_1 \cdot b_1, a_0 \cdot b_0\}$$

$$pp_2 = \{a_3 \cdot b_2, a_2 \cdot b_2, a_1 \cdot b_2, a_0 \cdot b_0\}$$

$$pp_3 = \{a_3 \cdot b_3, a_2 \cdot b_3, a_1 \cdot b_3, a_0 \cdot b_0\}$$

2. shift each partial product $pp_i$ $i$ binary positions left; results $n$ $(2n - 1)$-bit numbers; for $n = 4$:

$$n0 = pp_0 << 0 = \{0, 0, 0, pp_0\}$$

$$n1 = pp_1 << 1 = \{0, 0, pp_1, 0\}$$

$$n2 = pp_2 << 2 = \{0, pp_2, 0, 0\}$$

$$n3 = pp_3 << 3 = \{pp_3, 0, 0, 0\}$$

easy to be done, with no circuits, by an appropriate connection of the AND array outputs to the next circuit level (see in Figure 6.28 the “hardware-less shifter” dashed box)

3. add the resulting $n$ numbers using a $n$CSA$_{2^{n-1}}$; for $n = 4$:

$$p = \{0, 0, 0, pp_0\} plus \{0, 0, pp_1, 0\} plus \{0, pp_2, 0, 0\} plus \{pp_3, 0, 0, 0\}$$

The combinational multiplier circuit is presented in Figure 6.28 for the small case of $n = 4$. The first stage – partial products computation – generate the partial products $pp_i$ using 2-input ANDs as one bit multipliers. The second stage of the computation request a hardware-less shifter circuit, because the multiplying $n$-bit numbers with a power of 2 no bigger than $n - 1$ is done by an appropriate connection of each $pp_i$ to the $(2n - 1)$-bit inputs of the

Figure 6.28: 4-bit combinational multiplier.
6.1. SIMPLE, RECURSIVE DEFINED CIRCUITS

next stage, filling up the unused positions with zeroes. The third stage consists of a reduction carry-save adder – nREDA_{2n−1} – which receives, as 2n − 1-bit numbers, the partial products pp_i, each multiplied with 2^i.

The circuit represented in Figure 6.28 for n = 4, has in the general case the size in O(n^2) and the depth in O(n). But, the actual size and depth of the circuit is established by the 0s applied to the input of the nREDA_{2n−1} circuit, because some full-adders are removed from design and others are reduced to half-adders. The actual size of nREDA_{2n−1} results to be very well approximated with the actual size of a nREDA_{n}. The actual depth of the combinational multiplier is well approximated with the depth of a 2.5n-bit adder.

The decision to use a combinational multiplier must be done taking into account (1) its area, (2) the acceleration provided for the function it performs and (3) the frequency of the multiplication operation in the application.

6.1.10 Arithmetic and Logic Unit

All the before presented circuits have had associated only one logic or one arithmetic function. Now is the time to design the internal structure of a previously defined circuit having many functions, which can be selected using a selection code: the arithmetic and logic unit – ALU. ALU is the main circuit in any computational device, such as processors, controllers or embedded computation structures.

A generic version of a simple ALU is presented in the following example.

Example 6.8 The 8-function ALU working on 32-bit numbers is described by the following Verilog module:

```verilog
module ALU(input carryIn, input [2:0] func, input [31:0] left, right, output reg carryOut, output reg [31:0] out);
always @(*)
  case (func)
    3’b000: {carryOut, out} = left + right + carryIn; // add
    3’b001: {carryOut, out} = left − right − carryIn; // sub
    3’b010: {carryOut, out} = {1’b0, left & right}; // and
    3’b011: {carryOut, out} = {1’b0, left | right}; // or
    3’b100: {carryOut, out} = {1’b0, left ^ right}; // xor
    3’b101: {carryOut, out} = {1’b0, ~left}; // not
    3’b110: {carryOut, out} = {1’b0, left}; // left
    3’b111: {carryOut, out} = {1’b0, left >> 1}; // shr
  default {carryOut, out} = 33’b0 − 1’b1;
endcase
endmodule
```

*File name: alu.v
Circuit name: arithmetic and logic unit
Description: the circuit selects, using the selection code 'func', one of the 8 functions*

---

159
The ALU circuit can be implemented in many forms. One of them is the speculative version (see Figure 6.29) described by the Verilog module from Example 6.8, where the case structure describes, in fact, an 8-input multiplexor for 33-bit words. We call this version speculative because all the possible functions are computed in order to be all available to be select when the function code arrives to the func input of ALU. This approach is efficient when the operands are available quickly and the function to be performed “arrives” lately (because it is usually decoded from the instruction fetched from a program memory). The circuit “speculates” computing all the defined functions offering 8 results from which the func code selects one. (This approach will be useful for the ALU designed for the stack processor described in Chapter 10.)

Figure 6.29: The internal structure of the speculative version of an arithmetic and logic unit. Each function is performed by a specific circuit and the output multiplexer selects the desired result.

The speculative version provides a fast version in some specific designs. The price is the big size of the resulting circuit (mainly because the arithmetic section contains an adder and a subtractor, instead a smaller circuit performing add or subtract according to a bit used to complement the right operand and the carryIn signal).

An area optimized solution is provided in the next example.

**Example 6.9** Let be the 32-bit ALU with 8 functions described in Example 2.8. The implementation will be done using an adder-subtractor circuit and a 1-bit slice for the logic functions. Results the following Verilog description:

```verilog
classicalAlu(output [31:0] out ,
              output carryOut,
              input carryIn ,
              input [31:0] left , right ,
              input [2:0] func );
wire [31:0] shift, add_sub, arith, logic;
addSub addSub(.out (add_sub ),
               .cout (carryOut),
               .left (left ),
               .right (right ),
```

...
The resulting circuit is represented in Figure 6.30. This version can be synthesized on a smaller area, because the number of EMUXs is smaller, instead of an adder and a subtractor an adder/subtractor is
used. The price for this improvement is a smaller speed. Indeed, the add_sub module “starts” to compute the addition or the subtract only when the signal \( \text{sub} = \text{func}[0] \) is received. Usually, the code \( \text{func} \) results from the decoding of the current operation to be performed, and, consequently, comes later.

We just learned a new feature of the Verilog language: how to use a \texttt{task} to describe a circuit used many times in implementing a simple, repetitive structure.

The internal structure of \( \text{ALU} \) consists mainly in \( n \) slices, one for each input pair \( \text{left}[i] \), \( \text{right}[i] \) and a carry-look-ahead circuit(s) used for the arithmetic section. It is obvious that \( \text{ALU} \) is also a simple circuit. The magnitude order of the size of \( \text{ALU} \) is given by the size of the carry-look-ahead circuit because each slice has only a constant dimension and a constant depth. Therefore, the \textit{fastest} version implies a size in \( O(n^3) \) because of the carry-look-ahead circuit. But, let’s remind: the price for the fastest solution is always too big! For optimal solutions see [Omondi ’94].

### 6.1.11 Comparator

Comparing functions are used in decisions. Numbers are compared to decide if they are equal or to indicate the biggest one. The \( n \)-bit comparator, \( \text{COMP}_n \), is represented in Figure 6.31a. The numbers to be compared are the \( n \)-bit positive integers \( a \) and \( b \). Three are the outputs of the circuit: \( \text{lt\_out} \), indicating by 1 that \( a < b \), \( \text{eq\_out} \), indicating by 1 that \( a = b \), and \( \text{gt\_out} \), indicating by 1 that \( a > b \). Three additional inputs are used as expanding connections. On these inputs is provided information about the comparison done on the higher range, if needed. If no higher ranges of the number under comparison, then these three inputs must be connected as follows: \( \text{lt\_in} = 0, \text{eq\_in} = 1, \text{gt\_in} = 0 \).

The comparison is a numerical operation which starts inspecting the most significant bits of the numbers to be compared. If \( a[n-1] = b[n-2] \), then the result of the comparison is given by comparing \( a[n-2:0] \) with \( b[n-1:0] \), else, the decision can be done comparing only \( a[n-1] \) with \( b[n-1] \) (using an elementary comparator, \( \text{ECOMP} = \text{COMP}_1 \) (see Figure 6.31b)), ignoring \( a[n-2:0] \) and \( b[n-2:0] \). Results a recursive definition for the comparator circuit.
6.1. SIMPLE, RECURSIVE DEFINED CIRCUITS

Definition 6.9 An n-bit comparator, COMP_n, is obtained serially connecting an COMP_1 with a COMP_{n−1}. The Verilog code describing COMP_1 (ECOMP) follows:

```verilog
module e_comp( input a , b ,
               lt_in , // the previous e_comp decided lt
eq_in , // the previous e_comp decided eq
gt_in , // the previous e_comp decided gt
          output lt_out , // a < b
eq_out , // a = b
gt_out ); // a > b);
assign lt_out = lt_in | eq_in & ~a & b,
        eq_out = eq_in & ~(a ^ b),
gt_out = gt_in | eq_in & a & ~b;
endmodule
```

⋄

The size and the depth of the circuit resulting from the previous definition are in \( O(n) \). The size is very good, but the depth is too big for a high speed application.

An optimal comparator is defined using another recursive definition based on the divide et impera principle.

Definition 6.10 An n-bit comparator, COMP_n, is obtained using two COMP_{n/2} to compare the higher and the lower half of the numbers (resulting \{lt_out_high, eq_out_high, gt_out_high\} and \{lt_out_low, eq_out_low, gt_out_low\}), and a COMP_1 to compare gt_out_low with lt_out_low in the context of \{lt_out_high, eq_out_high, gt_out_high\}. The resulting circuit is represented in Figure 6.32. ⋄

Figure 6.31: The n-bit comparator, COMP_n. a. The n-bit comparator. b. The elementary comparator. c. A recursive rule to build an COMP_n, serially connecting an ECOMP with a COMP_{n−1}
The resulting circuit is a log-level binary tree of ECOMPs. The size remains in the same order\(^4\), but now the depth is in \(O(\log n)\).

The bad news is: the HDL languages we have are unable to handle safely recursive definitions. The good news is: the synthesis tools provide good solutions for the comparison functions starting from a very simple behavioral description.

### 6.1.12 Sorting network

In the most of the cases numbers are compared in order to be sorted. There are a lot of algorithms for sorting numbers. They are currently used to write programs for computers. But, in G2CE the sorting function will migrate into circuits, providing specific accelerators for general purpose computing machines.

To solve in circuits the problem of sorting numbers we start again from an elementary module: the **elementary sorter** (ESORT).

**Definition 6.11** An elementary sorter (ESORT) is a combinational circuit which receives two \(n\)-bit integers, \(a\) and \(b\) and generate outputs them as \(\min(a, b)\) and \(\max(a, b)\). The logic symbol of ESORT is represented in Figure 6.33b.

The internal organization of an ESORT is represented in Figure 6.33a. If \(\text{COMP}_n\) is implemented in an optimal version, then this circuit is optimal because its size is linear and its depth is logarithmic.

The circuit for sorting a vector of \(n\) numbers is build by ESORTs organized on many stages. The resulting combinational circuit receives the input vector \((x_1, x_2, x_3, \ldots, x_n)\) and generates the sorted version of it. In Figure 6.34 is represented a small network of ESORTs able to sort the vector of integers \((a, b, c, d)\). The sorted is organized on three stages. On the first stage two ESORTs are used sort separately the sub-vectors \((a, b)\) and \((c, d)\). On the second stage, the minimal values and the maximal values obtained from the previous stage are sorted, resulting the the smallest value (the minimal of the minimals), the biggest value (the maximal of the maximal) and the two intermediate values. For the last two the third level contains the last ESORT which sorts the middle values.

The resulting 4-input sorting circuit has the depth \(D_{\text{sort}}(4) = 3 \times D_{\text{esort}}(n)\) and the size \(S_{\text{sort}}(4) = 5 \times S_{\text{esort}}(n)\), where \(n\) is the number of bits used to represent the sorted integers.

\(^4\) The actual size of the circuit can be minimized taking into account that: (1) the compared input of ECOMP cannot be both 1, (2) the output \(\text{eq}_\text{out}\) of one \(\text{COMP}_{\text{n} / 2}\) is unused, and (3) the expansion inputs of both \(\text{COMP}_{\text{n} / 2}\) are all connected to fix values.
6.1. SIMPLE, RECURSIVE DEFINED CIRCUITS

Figure 6.33: **The elementary sorter.** a. The internal structure of an elementary sorter. The output \( \text{lt}_\text{out} \) of the comparator is used to select the input values to output in the received order (if \( \text{lt}_\text{out} = 1 \)) or in the crossed order (if \( \text{lt}_\text{out} = 0 \)). b. The logic symbol of an elementary sorter.

![Diagram of elementary sorter]

**Figure 6.34: The 4-input sorter.** The 4-input sorter is a three-stage combinational circuit built by 5 elementary sorters.

**Batcher’s sorter**

What is the rule to design a sorter for a \( n \)-number sequence? This topic will be presented using [Batcher ’68], [Knuth ’73] or [Parberry ’87].

The \( n \)-number sequence sorter circuit, \( S_n \), is presented in Figure 6.35. It is a double-recursive construct containing two \( S_{n/2} \) modules and the merge module \( M_n \), which has also a recursive definition, because it contains two \( M_{n/2} \) modules and \( n/2 - 1 \) elementary sorters, \( S_2 \). The \( M_n \) module is defined as a sorter which sorts the input sequence of \( n \) numbers only if it is formed by two \( n/2 \)-number sorted sequences.

In order to prove that Batcher’s sorter represented in Figure 6.35 sorts the input sequence we need the following theorem.

**Theorem 6.2** An \( n \)-input comparator network is a sorting network if and only if it works as sorter for all sequences of \( n \) symbols of zeroes and ones. □

The previous theorem is known as Zero-One Principle.

We must prove that, if \( M_{n/2} \) are merge circuits, then \( M_n \) is a merge circuit. The circuit \( M_2 \) is an elementary sorter, \( S_2 \).

If \( \{x_0, \ldots, x_{n-1}\} \) is a sequence of 0 and 1, and \( \{a_0, \ldots, a_{n-1}\} \) is a sorted sequence with \( g \) zeroes, while \( \{b_0, \ldots, b_{n-1}\} \) is a sorted sequence with \( h \) zeroes, then the left \( M_{n/2} \) circuit receives \( \lceil g/2 \rceil + \lceil h/2 \rceil \) zeroes, and

\[ [a] \text{ means rounded up integer part of the number } a. \]
Figure 6.35: **Batcher’s sorter.** The \( n \)-input sorter, \( S_n \), is defined by a double-recursive construct: “\( S_n = 2 \times S_{n/2} + M_n \)”, where the merger \( M_n \) consists of “\( M_n = 2 \times M_{n/2} + (n/2 - 1)S_2 \)”. 

\[
S_n = S_{n/2} = 2 \times S_{n/4} + M_{n/2} \\
M_n = M_{n/2} = 2 \times M_{n/4} + (n/4 - 1)S_2 \\
S_2 = 2 \\
M_2 = M_1 = 2 \\
\]

\[
M_{n/2} = M_n = 2 \times M_{n/4} + (n/4 - 1)S_2 \\
M_1 = M_{n/2} = 2 \times M_{n/4} + (n/4 - 1)S_2 \\
S_4 = 2 \times S_2 + M_2 \\
M_4 = M_2 = 2 \times M_1 + (n/2 - 1)S_2 \\
S_8 = 2 \times S_4 + M_4 \\
M_8 = M_4 = 2 \times M_2 + (n/2 - 1)S_2 \\
S_{16} = 2 \times S_8 + M_8 \\
M_{16} = M_8 = 2 \times M_4 + (n/4 - 1)S_2 \\
S_{32} = 2 \times S_{16} + M_{16} \\
M_{32} = M_{16} = 2 \times M_8 + (n/4 - 1)S_2 \\
S_{64} = 2 \times S_{32} + M_{32} \\
M_{64} = M_{32} = 2 \times M_{16} + (n/8 - 1)S_2 \\
\]

\[
S_n = 2 \times S_{n/2} + M_n \\
M_n = 2 \times M_{n/2} + (n/2 - 1)S_2 \\
S_2 = 2 \\
M_2 = M_1 = 2 \\
\]
6.1. SIMPLE, RECURSIVE DEFINED CIRCUITS

Figure 6.36: 16-input Batcher's sorter.

Figure 6.37: 32-input Batcher's merge circuit.
the right $M_{n/2}$ circuit receives $\lfloor g/2 \rfloor + \lfloor h/2 \rfloor$ zeroes\(^6\). The value:

$$Z = \lfloor g/2 \rfloor + \lfloor h/2 \rfloor - (\lfloor g/2 \rfloor + \lfloor h/2 \rfloor)$$

takes only three values with the following output behavior for $M_n$:

- **Z=0**: at most one $S_2$ receives on its inputs the unsorted sequence $\{1,0\}$ and does it work, while all the above receive $\{0,0\}$ and the bellow receive $\{1,1\}$
- **Z=1**: $y_0 = 0$, follows a number of elementary sorters receiving $\{0,0\}$, while the rest receive $\{1,1\}$ and the last output is $y_{n-1} = 1$
- **Z=2**: $y_0 = 0$, follows a number of elementary sorters receiving $\{0,0\}$, then one sorter with $\{0,1\}$ on its inputs, while the rest receive $\{1,1\}$ and the last output is $y_{n-1} = 1$

Thus, no more than one elementary sorter is used to reverse the order in the received sequence.

The size and depth of the circuit is computed in two stages, corresponding to the two recursive levels of the definition. The size of the $n$-input merge circuit, $S_M(n)$, is iteratively computed starting with:

$$S_M(n) = 2S_M(n/2) + (n/2 - 1)S_S(2)$$

$$S_M(2) = S_S(2)$$

Once the size of the merge circuit is obtained, the size of the $n$-input sorter, $S_S(n)$, is computed using:

$$S_S(n) = 2S_S(n/2) + S_M(n)$$

Results:

$$S_S(n) = (n\log_2 n)(-1 + \log_2 n)/4 + n - 1)S_S(2)$$

A similar approach is used for the computation of the depth. The depth of the $n$-input merge circuit, $D_M(n)$, is iteratively computed using:

$$D_M(n) = D_M(n/2) + D_S(2)$$

$$D_M(2) = D_S(2)$$

while the depth of the $n$-input sorter, $D_S(n)$, is computed with:

$$D_S(n) = D_S(n/2) + D_M(n)$$

Results:

$$D_S(n) = (\log_2 n)(\log_2 n + 1)/2$$

We conclude that: $S_S(n) \in O(n \times \log^2 n)$ and $D_S(n) \in O(\log^2 n)$.

In Figure 6.36 a 16-input sorter designed according to the recursive rule is shown, while in Figure 6.37 a 32-input merge circuit is detailed. With 2 16-input sorters and a 32-input merger a 32-input sorted can be build.

In [Ajtai '83] is a presented theoretically improved algorithm, with $S_S(n) \in O(n \times \log n)$ and $D_S(n) \in O(\log n)$. But, the constants associated to the magnitude orders are too big to provide an optimal solution for currently realistic circuits characterized by $n < 10^9$.

---

\(^6\) $\lfloor a \rfloor$ means rounded down integer part of the number $a$. 
6.1. SIMPLE, RECURSIVE DEFINED CIRCUITS

The recursive Verilog description is very useful for this circuit because of the difficulty to describe in a HDL a double recursive circuit.

The top module describes the first level of the recursive definition: a \(n\)-input sorter is built using two \(n/2\)-input sorters and an \(n\)-input merger, and the 2-input sorter is the elementary sorter. Results the following description in Verilog:

```verilog
module sorter #(`include "0_parameters.v")
    ( output [m*n-1:0] out ,
      input [m*n-1:0] in );

    wire [m*n/2-1:0] out0;
    wire [m*n/2-1:0] out1;

    generate
        if (n == 2)
            eSorter eSorter(.out0 (out[m-1:0] ),
                           .out1 (out[2*m-1:m] ),
                           .in0 (in[m-1:0] ),
                           .in1 (in[2*m-1:m] ));
        else begin
            sorter #(.n(n/2)) sorter0(.out(out0 ),
                                       .in (in[m*n/2-1:0] )�
                                 sorter1(.out(out1 ),
                                       .in (in[m*n-1:m*n/2]));
            merger #(.n(n)) merger(.out(out ),
                                      .in ({out1, out0} )�
        end
    endgenerate
endmodule
```

For the elementary sorter, eSorter, we have the following description:

```verilog
module eSorter #(`include "0_parameters.v")
    ( output [m-1:0] out0,
      output [m-1:0] out1,
      input [m-1:0] in0 ,
      input [m-1:0] in1 );

    assign out0 = (in0 > in1) ? in1 : in0 ;
    assign out1 = (in0 > in1) ? in0 : in1 ;
endmodule
```

The \(n\)-input merge circuit is described recursively using two \(n/2\)-input merge circuits and the elementary sorters as follows:

```verilog
module merger #(`include "0_parameters.v")
    ( output [m*n-1:0] out ,
      input [m*n-1:0] in );

    wire [m*n/4-1:0] even0 ;
    wire [m*n/4-1:0] odd0 ;
    wire [m*n/4-1:0] even1 ;
    wire [m*n/4-1:0] odd1 ;
    wire [m*n/2-1:0] out0 ;
    wire [m*n/2-1:0] out1 ;
```
CHAPTER 6. GATES: ZERO ORDER, NO-LOOP DIGITAL SYSTEMS

```verbatim
genvar i;
generate
if (n == 2) eSorter eSorter(.out0 (out[m-1:0] ),
  .out1 (out[2*m-1:m] ),
  .in0 (in[m-1:0] ),
  .in1 (in[2*m-1:m] ));
else begin
  for (i=0; i<n/4; i=i+1) begin : oddEven
    assign even0[(i+1)*m-1:i*m] = in[2*i*m+m-1:2*i*m] ;
    assign even1[(i+1)*m-1:i*m] = in[m*n/2+2*i*m+m-1:m*n/2+2*i*m] ;
    assign odd0[(i+1)*m-1:i*m] = in[2*i*m+2*m-1:2*i*m+m] ;
    assign odd1[(i+1)*m-1:i*m] = in[m*n/2+2*i*m+2*m-1:m*n/2+2*i*m+m] ;
  end
  merger #(n(n/2)) merger0(.out(out0 ),
    .in ({even1, even0} )),
  merger1(.out(out1 ),
    .in ({odd1, odd0} ));
  for (i=1; i<n/2; i=i+1) begin : elSort
    eSorter eSorter(.out0 (out[(2*i-1)*m+m-1:(2*i-1)*m] ),
      .out1 (out[2*i*m+m-1:2*i*m] ),
      .in0 (out[(i+1)*m-1:i*m] ),
      .in1 (out[(i-1)*m+:m-1]));
  end
  assign out[m-1:0] = out0[m-1:0] ;
  assign out[m*n-1:m*(n-1)] = out1[m*n/2-1:m*(n/2-1)] ;
end
endgenerate
endmodule
```

The parameters of the sorter are defined in the file zero_parameters.v:

```verbatim
parameter n = 16, // number of inputs
    m = 8  // number of bits per input
```

6.1.13 * First detection network

Let be the vector of Boolean variable: inVector = \{x_0, x_1, \ldots x_(n-1)\}. The function firstDetect outputs three vectors of the same size:

- first = \{0, 0, \ldots 0, 1, 0, 0, \ldots 0\}
- beforeFirst = \{1, 1, \ldots 1, 0, 0, 0, \ldots 0\}
- afterFirst = \{0, 0, \ldots 0, 0, 1, 1, \ldots 1\}

indicating, by turn, the position of the first 1 in inVector, all the positions before the first 1, and all the positions after the first 1.

**Example 6.10** Let be a 16-bit input circuit performing the function firstDetect.

- inVector = \{0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1\}
- first = \{0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}
- beforeFirst = \{1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}
- afterFirst = \{0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1\}
The circuit performing the function firstDetect is described by the following Verilog program:

```verilog
module firstDetect #(parameter n =4)( input [0:n-1] in ,
               output [0:n-1] first ,
               output [0:n-1] beforeFirst ,
               output [0:n-1] afterFirst );
    wire [0:n-1] orPrefixes;
    or_prefixes prefixNetwork(.in (in ),
                                .out(orPrefixes));

    assign first = orPrefixes & ~(orPrefixes >> 1),
    beforeFirst = ~orPrefixes ,
    afterFirst = (orPrefixes >> 1) ;
endmodule

module or_prefixes #(parameter n =4)(input [0:n-1] in ,
                                    output reg [0:n-1] out);
    integer k;
    always @(in) begin out[0] = in[0];
        for (k=1; k<n; k=k+1) out[k] = in[k] | out[k-1];
    end
endmodule
```

The function firstDetect classifies each component of a Boolean vector related to the first occurrence of the value 1.

6.1.14 *Spira’s theorem*

6.2 Complex, Randomly Defined Circuits

6.2.1 An Universal circuit

Besides the simple, recursively defined circuits there is the huge class of the complex or random circuits. Is there a general solution for these category of circuits? A general solution asks a general circuit and this circuit surprisingly exists. Now rises the problem of how to catch the huge diversity of random in this approach. The following theorem will be the first step in solving the problem.

**Theorem 6.3** For any \( n \), all the functions of \( n \) binary-variables have a solution with a combinational logic circuit.  

**Proof** Any Boolean function of \( n \) variables can be written as:

\[
f(x_{n-1}, \ldots, x_0) = x_{n-1}^j g(x_{n-2}, \ldots, x_0) + x_{n-1} h(x_{n-2}, \ldots, x_0),
\]

where:

\[
g(x_{n-2}, \ldots, x_0) = f(0, x_{n-2}, \ldots, x_0)
\]

\[
h(x_{n-2}, \ldots, x_0) = f(1, x_{n-2}, \ldots, x_0)
\]
Therefore, the computation of any $n$-variable function can be reduced to the computation of two other $(n - 1)$-variables functions and an EMUX. The circuit, and in the same time the algorithm, is represented in Figure 6.40. For the functions $g$ and $h$ the same rule may applies. And so on until the two zero-variable functions: the value 0 and the value 1. The solution is an $n$-level binary tree of EMUXs having applied to the last level zero-variable functions. Therefore, solution is a $MUX_n$ and a binary string applied on the $2^n$ selected inputs. The binary string has the length $2^n$. Thus, for each of the $2^{2^n}$ functions there is a distinct string defining it.

The universal circuit is indeed the best example of a big simple circuit, because it is described by the following code:

```verilog
module nU_circuit #('include "parameter.v")
  ( output out ,
    input [(1'b1 << n)-1:0] program ,
    input [n-1:0] data );

  assign out = program[data];
endmodule
```

The file parameter.v contains the value for $n$. But, attention! The size of the circuit is: $s_{nU\_circuit}(n) \in O(2^n)$.

Thus, circuits are more powerful than Turing Machine (TM), because TM solve only problem having the solution algorithmically expressed with a sequence of symbols that does not depend by $n$. Beyond

\footnote{The function firstDetect becomes very meaningful related to the minimalization rule in Kleene’s computational model [Kleene '36].}
The solution imposed by the previous theorem is an universal circuit for computing the \(n\)-binary variable functions. Let us call it \(n\text{-U-circuit}\) (see Figure 6.41). The size of this circuit is \(S_{\text{universal}}(n) \in O(2^n)\) and its complexity is \(C_{\text{universal}}(n) \in O(1)\). The functions is specified by the “program” \(P = m_{p-1}, m_{p-2}, \ldots, m_0\) which is applied on the selected inputs of the \(n\)-input multiplexer \(MUX_n\). It is about a huge simple circuit. The functional complexity is associated with the “program” \(P\), which is a binary string.

This universal solution represents the strongest segregation between a simple physical structure - the \(n\)-input MUX - and a complex symbolic structure - the string of \(2^n\) bits applied on the selected inputs which works like a “program”. Therefore, this is THE SOLUTION, MUX is THE CIRCUIT and we can stop here our discussion about digital circuits!? ... **But, no!** There are obvious reasons to continue our walk in the world of digital circuits:

- first: the exponential size of the resulting physical structure
CHAPTER 6. GATES: ZERO ORDER, NO-LOOP DIGITAL SYSTEMS

Figure 6.40: The universal circuit. For any $CLC_f(n)$, where $f(x_{n-1}, \ldots, x_0)$ this recursively defined structure is a solution. EMUX behaves as an elementary universal circuit.

- second: the huge size of the “programs” which are in a tremendous majority represented by random, uncompressible, strings (hard or impossible to be specified).

The strongest segregation between simple and complex is not productive in no-loop circuits. Both resulting structure, the simple circuit and the complex binary string, grow exponentially.

6.2.2 Using the Universal circuit

We have a chance to use $MUX_n$ to implement $f(x_{n-1}, \ldots, x_0)$ only if one of the following conditions is fulfilled:

1. $n$ is small enough resulting realizable and efficient circuits

2. the “program” is a string with useful regularities (patterns) allowing strong minimization of the resulting circuit

Follows few well selected examples. First is about an application with $n$ enough small to provide an useful circuit (it is used in Connection Machine as an “universal” circuit performing anyone of the 3-input logic function [Hillis '85]).

**Example 6.11** Let be the following Verilog code:

```verilog
module three_input_functions(output out, input [7:0] func, input in0, in1, in2);
assign out = func[{in0, in1, in2}];
endmodule
```

The circuit three_input_functions can be programmed, using the 8-bit string func as “program”, to perform anyone 3-input Boolean function. It is obviously a $MUX_3$ performing

$$out = f(in0,in1,in2)$$

where the function $f$ is specified (“programmed”) by an 8-bit word (“program”). ⊥
6.2. COMPLEX, RANDOMLY DEFINED CIRCUITS

The “programmable” circuit for any 4-input Boolean function is obviously MUX4:

\[ \text{out} = f(\text{in}_0, \text{in}_1, \text{in}_2, \text{in}_3) \]

where \( f \) is “programmed” by a 16-bit word applied on the selected inputs of the multiplexer.

The bad news is: we can not go too far on this way because the size of the resulting universal circuits increases exponentially. The good news is: usually we do not need universal but particular solution. The circuits are, in most of cases, specific not universal. They “execute” a specific “program”. But, when a specific binary word is applied on the selected inputs of a multiplexer, the actual circuit is minimized using the following removing rules and reduction rules.

An EMUX defined by:

\[ \text{out} = x \ ? \ \text{in}_1 : \text{in}_0; \]

can be removed, if on its selected inputs specific 2-bit binary words are applied, according to the following rules:

- if \( \{\text{in}_1, \text{in}_0\} = 00 \) then \( \text{out} = 0 \)
- if \( \{\text{in}_1, \text{in}_0\} = 01 \) then \( \text{out} = x' \)
- if \( \{\text{in}_1, \text{in}_0\} = 10 \) then \( \text{out} = x \)
- if \( \{\text{in}_1, \text{in}_0\} = 11 \) then \( \text{out} = 1 \)

or, if the same variable, \( y \), is applied on both selected inputs:

- if \( \{\text{in}_1, \text{in}_0\} = yy \) then \( \text{out} = y \)
An EMUX can be reduced, if on one of its selected inputs a 1-bit binary word are applied, being substituted with a simpler circuit according to the following rules:

- if \( f_{in} = 1; in = 0 \), then \( out = y \)
- if \( f_{in} = 1; in = 0 \), then \( out = y + x' \)
- if \( f_{in} = 0 \), then \( out = x'y \)
- if \( f_{in} = 1 \), then \( out = x + y \)

Results: the first level of \( 2^{n-1} \) EMUXs of a \( MUX_n \) is reduced, and on the inputs of the second level (of \( n \) EMUXs) is applied a word containing binary values (0s and 1s) and binary variables (\( x_0 \)'s and \( x_0' \)'s). For the next levels the removing rules or the reducing rules are applied.

![Figure 6.42: The majority function.](image)
The majority function for 3 binary variables is solved by a 3-level binary tree of EMUXs. The actual “program” applied on the “leafs” will allow to minimize the tree.

**Example 6.12** Let us solve the problem of majority function for three Boolean variable. The function \( \text{maj}(x_2, x_1, x_0) \) returns 1 if the majority of inputs are 1, and 0 if not. In Figure 6.42 a “programmable” circuit is used to solve the problem.

Because we intend to use the circuit only for the function \( \text{maj}(x_2, x_1, x_0) \) the first layer of EMUXs can be removed resulting the circuit represented in Figure 6.43a.

On the resulting circuit the reduction rules are applied. The result is presented in Figure 6.43b.

The next examples refer to big \( n \), but “program” containing repetitive patterns.

**Example 6.13** If the “program” is 128-bit string \( i_{127} \ldots i_0 = (10)^{64} \), it corresponds to a function of form:

\[
    f(x_6, \ldots, x_0)
\]

where: the first bit, \( i_0 \) is the value associated to the input configuration \( x_6, \ldots, x_0 = 0000000 \) and the last bit \( i_{127} \) is the value associated to input configuration \( x_6, \ldots, x_0 = 1111111 \) (according with the representation from Figure 6.11 which is equivalent with Figure 6.40).
6.2. COMPLEX, RANDOMLY DEFINED CIRCUITS

The reduction process. a. For any function the first level of EMUSs is reduced to a binary vector (\((1, 0)\) in this example). b. For the actual “program” of the 3-input majority function the second level is supplementary reduced to simple gates (an AND\(_2\) and an OR\(_2\)).

The obvious regularities of the “program” leads our mind to see what happened with the resulting tree of EMUXs. Indeed, the structure collapses under this specific “program”. The upper layer of 64 EMUXs are selected by \(x_0\) and each have on their inputs \(i_0 = 1\) and \(i_1 = 0\), generating \(x'_0\) on their outputs. Therefore, the second layer of EMUXs receive on all selected inputs the value \(x'_0\), and so on until the output generates \(x'_0\). Therefore, the circuit performs the function \(f(x_0) = x'_0\) and the structure is reduced to a simple inverter.

In the same way the “program” \((0011)^{32}\) programs the 7-input MUX to perform the function \(f(x_1) = x_1\) and the structure of EMUXs disappears.

For the function \(f(x_1, x_0) = x_1x'_0\) the “program” is \((0010)^{32}\).

For a 7-input AND the “program” is \(01^{127}\), and the tree of MUXs degenerates in 7 EMUXs serially connected each having the input \(i_0\) connected to 0. Therefore, each EMUX become an AND\(_2\) and applying the associativity principle results an AND\(_7\).

In a similar way, the same structure become an OR\(_7\) if it is “programmed” with \(01^{127}\).

It is obvious that if the “program” has some uniformities, these uniformities allow to minimize the size of the circuit in polynomial limits using removing and reduction rules. The simple “programs” lead toward small circuits.

6.2.3 The many-output random circuit: Read Only Memory

The simple solution for the following many-output random circuits having the same inputs:

\[
\begin{align*}
f(x_{n-1}, \ldots, x_0) \\
g(x_{n-1}, \ldots, x_0) \\
\vdots \\
s(x_{n-1}, \ldots, x_0)
\end{align*}
\]
is to connect in parallel many one-output circuits. The inefficiency of the solution becomes obvious when
the structure of the MUX presented in Figure 6.9 is considered. Indeed, if we implement many MUXs
with the same selection inputs, then the decoder $DCD_n$ is replicated many times. One DCD is enough for
many MUXs if the structure from Figure 6.44a is adopted. The DCD circuit is shared for implementing
the functions $f, g, \ldots, s$. The shared DCD is used to compute all possible minterms (see Appendix C.4)
needed to compute an $n$-variable Boolean function.

Figure 6.44: Many-output random circuit. a. One DCD and many AND-OR circuits. b. An example. c.
The version using programmable connections.

Figure 6.44b is an example of using the generic structure from Figure 6.44a to implement a specific
many-output function. Each output is defined by a different binary string. A 0 removes the associated
AND, connecting the corresponding OR input to 0, and an 1 connects to the corresponding $i$-th input of
6.2. COMPLEX, RANDOMLY DEFINED CIRCUITS

Each OR to the \( i \)-th DCD output. The equivalent resulting circuit is represented in Figure 6.44c, where some OR inputs are connected to \textit{ground} and other directly to the DCD’s output. Therefore, we use a technology allowing us to make “programmable” connections of some wires to other (each vertical line must be connected to one horizontal line). The \textit{uniform} structure is “programmed” with a more or less \textit{random} distribution of connections.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6.45.png}
\caption{The internal structure of a Read Only Memory used as trans-coder. \textbf{a.} The internal structure. \textbf{b.} The simplified logic symbol where a thick vertical line is used to represent an \( m \)-input NAND gate.}
\end{figure}

If De Morgan transformation is applied, the circuit from Figure 6.44c is transformed in the circuit represented in Figure 6.45a, where instead of an active high outputs DCD an active low outputs DCD is considered and the OR gates are substituted with NAND gates. The DCD’s outputs are generated using NAND gates to \textit{decode} the input binary word, the same as the gates used to \textit{encode} the output binary word. Thus, a multi-output Boolean function works like a \textit{trans-coder}. A trans-coder works translating all the binary input words into output binary words. The list of input words can by represented as an ordered list of sorted binary numbers starting with 0 and ending with \( 2^n - 1 \). The table from Figure 6.46 represents the \textbf{truth table} for the multi-output function used to exemplify our approach. The left column contains all binary numbers from 0 (on the first line) until \( 2^n - 1 = 11\ldots1 \) (on the last line). In the right column the desired function is defined associating to each input an output. If the left column is an ordered list, the right column has a more or less random content (preferably more random for this type of solution).

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Input & Output \\
\hline
00 \ldots 00 & 11 \ldots 0 \\
\ldots & \ldots \\
11 \ldots 10 & 10 \ldots 0 \\
11 \ldots 11 & 01 \ldots 1 \\
\hline
\end{tabular}
\caption{The truth table for a multi-output Boolean function. The input columns can be seen as addresses, from 00...0 to 11...1 and the output columns as the content stored at the corresponding addresses.}
\end{table}

The trans-coder circuit can be interpreted as a \textit{fix content memory}. Indeed, it works like a memory containing at the location 00...00 the word 11...0, \ldots at the location 11...10 the word 10...0, and at the last location the word 01...1. The name of this kind of programmable device is \textit{read only memory}, ROM.
Example 6.14 The trans-coder from the binary coded decimal numbers to 7 segments display is a combinational circuit with 4 inputs, $a, b, c, d$, and 7 outputs $A, B, C, D, E, F, G$, each associated to one of the seven segments. Therefore we have to solve 7 functions of 4 variables (see the truth table from Figure 6.48).

The Verilog code describing the circuit is:

```verilog
module seven_segments( output reg [6:0] out ,
                        input [3:0] in );

always @(in) case(in)
    4'b0000: out = 7'b0000001;
    4'b0001: out = 7'b1001111;
    4'b0010: out = 7'b0010010;
    4'b0011: out = 7'b0000110;
    4'b0100: out = 7'b1001100;
    4'b0101: out = 7'b0100100;
    4'b0110: out = 7'b0100000;
    4'b0111: out = 7'b0001111;
    4'b1000: out = 7'b0000000;
    4'b1001: out = 7'b0000100;
    default out = 7'bxxxxxx;
endcase
endmodule
```

The first solution is a 16-location of 7-bit words ROM (see Figure 6.47a. If inverted outputs are needed results the circuit from Figure 6.47b.

Programmable Logic Array

In the previous example each output of DCD compute the inverted value of a minterm. But our applications do not need all the possible minterms, for two reasons:

* Programmable Logic Array
6.2. COMPLEX, RANDOMLY DEFINED CIRCUITS

<table>
<thead>
<tr>
<th>abcd</th>
<th>ABCDEFG</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1111110</td>
</tr>
<tr>
<td>0001</td>
<td>0110000</td>
</tr>
<tr>
<td>0010</td>
<td>1101101</td>
</tr>
<tr>
<td>0011</td>
<td>1111001</td>
</tr>
<tr>
<td>0100</td>
<td>0110011</td>
</tr>
<tr>
<td>0101</td>
<td>1011011</td>
</tr>
<tr>
<td>0110</td>
<td>1011111</td>
</tr>
<tr>
<td>0111</td>
<td>1110000</td>
</tr>
<tr>
<td>1000</td>
<td>1111111</td>
</tr>
<tr>
<td>1001</td>
<td>1111011</td>
</tr>
<tr>
<td>1010</td>
<td>-------</td>
</tr>
<tr>
<td>......</td>
<td>.......</td>
</tr>
<tr>
<td>1111</td>
<td>-------</td>
</tr>
</tbody>
</table>

Figure 6.48: The truth table for the 7 segment trans-coder. Each binary represented decimal (in the left columns of inputs) has associated a 7-bit command (in the right columns of outputs) for the segments used for display. For unused input codes the output is “don’t care”.

- the function is not defined for all possible input binary configurations (only the input codes representing numbers from 0 to 9 define the output behavior of the circuit)
- in the version with inverted outputs the minterm corresponding for the input 1000 (the number 8) is not used.

A more flexible solution is needed. ROM consists in two arrays of NANDs, one fix and another configurable (programmable). What if also the first array is configurable, i.e., the DCD circuit is programmed to compute only some minterms? More, what if instead of computing only minterms (logic products containing all the input variable) we compute also some, or only, terms (logic products containing only a part of input variable)? As we know a term corresponds to a logic sum of minterms. Computing a term in a programmable array of NANDs two or more NANDs with \( n \) inputs are substituted with a NAND with \( n-1 \) or less inputs. Applying these ideas results in another frequently used programmable circuit: the famous PLA (programmable logic array).

Example 6.15 Let’s revisit the problem of 7 segment trans-coder. The solution is to use a PLA. Because now the minimal form of equations is important the version with inverted outputs is considered. Results the circuit represented in Figure 6.49, where a similar convention for representing NANDs as a thick line is used.

When PLA are used as hardware support the minimized form of Boolean functions (see Appendix C.4 for a short refresh) are needed. In the previous example for each inverted output its minimized Boolean expression was computed.

The main effect of substituting, whenever is possible, ROMs with PLAs are:
- the number of decoder outputs decreases
- the size of circuits that implements the decoder decreases (some or all minterms are substituted with less terms)
- the number of inputs of the NANDs on the output level also decreases.

There are applications supposing ROMs with a very random content, so as the equivalent PLA has the same ar almost the same size and the effort to translate the ROM into a PLA does not deserve. A typical case is when we “store” into a ROM a program executed by a computing machine. No regularities are expected in such an applications.
It is also surprising the efficiency of a PLA in solving pure Boolean problems which occur in current digital designs. A standard PLA circuit, with 16 inputs, 8 outputs and only 48 programmable (min)terms on the first decoding level, is able to solve a huge amount of pure logic (non-arithmetic) applications. A full decoder in a ROM circuit computes 66536 minterms, and the previous PLA is designed to support no more than 48 terms!

Warning! For arithmetic applications PLA are extremely inefficient. Short explanation: a 16-input XOR supposes 32768 minterms to be implemented, but a 16-input AND can be computed using one minterm. The behavioral diversity to the output of an adder is similar with the behavioral diversity on its inputs. But the diversity on the output of 8-input NAND is almost null. The probability of 1 on the output of 16-input AND is $\frac{1}{2^{16}} = 0.000015$.

6.3 Concluding about combinational circuits

The goal of this chapter was to introduce the main type of combinational circuits. Each presented circuit is important first, for its specific function and second, as a suggestion for how to build similar ones. There are a lot of important circuits undiscussed in this chapter. Some of them are introduced as problems at the end of this chapter.

Simple circuits vs. complex circuits Two very distinct class of combinational circuits are emphasized. The first contains simple circuits, the second contains complex circuits. The complexity of a circuit is distinct from the size of a circuit. Complexity of a circuit is given by the size of the definition used to specify that circuit. Simple circuits can achieve big sizes because they are defined using a repetitive pattern. A complex circuit can not be very big because its definition is dimensioned related with its size.

Simple circuits have recursive definitions Each simple circuit is defined initially as an elementary module performing the needed function on the smallest input. Follows a recursive definition about how can be used the elementary circuit to define a circuit working for any input dimension. Therefore, any big simple circuit is a network of elementary modules which expands according to a specific rule. Unfortunately, the actual HDL, Verilog included, are not able to manage without (strong) restrictions recursive definitions neither in simulation nor in synthesis. The recursiveness is a property of simple circuits to be fully used only for our mental experiences.
6.4. PROBLEMS

**Speeding circuits means increase their size** Depth and size evolve in opposite directions. If the speed increases, the pay is done in size, which also increases. We agree to pay, but in digital systems the pay is not fair. We conjecture the bigger is performance the bigger is the unit price. Therefore, the pay increases more than the units we buy. It is like paying urgency tax. If the speed increases \( n \) times, then the size of the circuit increases more than \( n \) times, which is not fair but it is real life and we must obey.

**Big sized complex circuits require programmable circuits** There are software tools for simulating and synthesizing complex circuits, but the control on what they generate is very low. A higher level of control we have using programmable circuits such as ROMs or PLA. PLA are efficient only if non-arithmetic functions are implemented. For arithmetic functions there are a lot of simple circuits to be used. ROM are efficient only if the randomness of the function is very high.

**Circuits represent a strong but ineffective computational model** Combinational circuits represent a theoretical solution for any Boolean function, but not an effective one. Circuits can do more than algorithms can describe. The price for their universal completeness is their ineffectiveness. In the general case, both the needed physical structure (a tree of EMUXs) and the symbolic specification (a binary string) increase exponentially with \( n \) (the number of binary input variables). More, in the general case only a family of circuits represents the solution.

To provide an effective computational tool new features must be added to a digital machine and some restrictions must be imposed on what is to be computable. The next chapters will propose improvements induced by successively closing appropriate loops inside the digital systems.

### 6.4 Problems

**Gates**

**Problem 6.1** Determine the relation between the total number, \( N \), of \( n \)-input \( m \)-output Boolean functions \( (f : \{0,1\}^n \rightarrow \{0,1\}^m) \) and the numbers \( n \) and \( m \).

**Problem 6.2** Let be a circuit implemented using 32 3-input AND gates. Using the appendix evaluate the area if 3-input gates are used and compare with a solution using 2-input gates. Analyze two cases: (1) the fan-out of each gate is 1, (2) the fan-out of each gate is 4.

**Decoders**

**Problem 6.3** Draw \( \text{DCD}_4 \) according to Definition 2.9. Evaluate the area of the circuit, using the cell library from Appendix E, with the placement efficiency\(^8\) 70%. Estimate the maximum propagation time. The wires are considered enough short to be ignored their contribution in delaying signals.

**Problem 6.4** Design a constant depth \( \text{DCD}_4 \). Draw it. Evaluate the area and the maximum propagation time using the cell library from Appendix E. Compare the results with the results of the previous problem.

**Problem 6.5** Propose a recursive definition for \( \text{DCD}_n \) using EDMUXs. Evaluate the size and the depth of the resulting structure.

\(^8\)For various reason the area used to place gates on Silicon can not completely used. Some unused spaces remain between gates. Area efficiency measures the degree of area use.
Multiplexors

Problem 6.6 Draw MUX₄ using EMUX. Make the structural Verilog design for the resulting circuit. Organize the Verilog modules as hierarchical as possible. Design a tester and use it to test the circuit.

Problem 6.7 Define the 2-input XOR circuit using an EDCD and an EMUX.

Problem 6.8 Make the Verilog behavioral description for a constant depth left shifter by maximum \( m - 1 \) positions for \( m \)-bit numbers, where \( m = 2^n \). The “header” of the project is:

```
module left_shift(output [2m-2:0] out ,
               input [m-1:0] in ,
               input [n-1:0] shift );
...
endmodule
```

Problem 6.9 Make the Verilog structural description of a log-depth (the depth is \( \log_2 16 = 4 \)) left shifter by 16 positions for 16-bit numbers. Draw the resulting circuit. Estimate the size and the depth comparing the results with a similar shifter designed using the solution of the previous problem.

Problem 6.10 Draw the circuit described by the Verilog module leftRotate in the subsection Shifters.

Problem 6.11 A barrel shifter for \( m \)-bit numbers is a circuit which rotate the bits the input word a number of positions indicated by the shift code. The “header” of the project is:

```
module barrel_shift(output [m-1:0] out ,
                   input [m-1:0] in ,
                   input [n-1:0] shift );
...
endmodule
```

Write a behavioral code and a minimal structural version in Verilog.

Prefix network

Problem 6.12 A prefix network for a certain associative function \( f \),

\[
P_f(x_0, x_1, \ldots, x_{n-1}) = \{ y_0, y_1, \ldots, y_{n-1} \}
\]

receives \( n \) inputs and generate \( n \) outputs defined as follows:

\[
\begin{align*}
y_0 &= f(x_0) \\
y_1 &= f(x_0, x_1) \\
y_2 &= f(x_0, x_1, x_2) \\
&\quad \quad \ldots \\
y_{n-1} &= f(x_0, x_1, \ldots, x_{n-1}).
\end{align*}
\]

Design the circuit \( P_{OR}(n) \) for \( n = 16 \) in two versions: (1) with the smallest size, (2) with the smallest depth.

Problem 6.13 Design \( P_{OR}(n) \) for \( n = 8 \) and the best product size \( \times \) depth.

Problem 6.14 Design \( P_{addition}(n) \) for \( n = 4 \). The inputs are 8-bit numbers. The addition is a mod256 addition.
6.4. PROBLEMS

Recursive circuits

Problem 6.15 A comparator is a circuit designed to compare two $n$-bit positive integers. Its definition is:

```verilog
module comparator( input [n-1:0] in1 , // first operand
                     input [n-1:0] in2 , // second operand
                     output eq , // in1 = in2
                     output lt , // in1 < in2
                     output gt ); // in1 > in2

... endmodule
```

1. write the behavioral description in Verilog
2. write a structural description optimized for size
3. design a tester which compare the results of the simulations of the two descriptions: the behavioral description and the structural description
4. design a version optimized for depth
5. define an expandable structure to be used in designing comparators for bigger numbers in two versions: (1) optimized for depth, (2) optimized for size.

Problem 6.16 Design a comparator for signed integers in two versions: (1) for negative numbers represented in 2s complement, (2) for negative numbers represented a sign and number.

Problem 6.17 Design an expandable priority encoder with minimal size starting from an elementary priority encoder, EPE, defined for $n = 2$. Evaluate its depth.

Problem 6.18 Design an expandable priority encoder, PE($n$), with minimal depth.

Problem 6.19 What is the numerical function executed by a priority encoder circuit if the input is interpreted as an $n$-bit integer, the output is an $m$-bit integer and $n_{ones}$ is a specific warning signal?

Problem 6.20 Design the Verilog structural descriptions for an 8-input adder in two versions: (1) using 8 FAs and a ripple carry connection, (2) using 8 HAs and a carry look ahead circuit. Evaluate both solutions using the cell library from Appendix E.

Problem 6.21 Design an expandable carry look-ahead adder starting from an elementary circuit.

Problem 6.22 Design an enabled incrementer/decrementer circuit for $n$-bit numbers. If $en = 1$, then the circuit increments the input value if $inc = 1$ or decrements the input value if $inc = 0$, else, if $en = 0$, the output value is equal with the input value.

Problem 6.23 Design an expandable adder/subtractor circuit for 16-bit numbers. The circuit has a carry input and a carry output to allow expandability. The 1-bit command input is $sub$. For $sub = 0$ the circuit performs addition, else it subtracts. Evaluate the area and the propagation time of the resulting circuit using the cell library from Appendix E.

Problem 6.24 Provide a “divide et impera” solution for the circuit performing firstDetect function.
Random circuits

**Problem 6.25** The Gray counting means to count, starting from 0, so as at each step only one bit is changed. Example: the three-bit counting means 000, 001, 011, 010, 110, 111, 101, 100, 000, ... Design a circuit to convert the binary counting into the Gray counting for 8-bit numbers.

**Problem 6.26** Design a converter from Gray counting to binary counting for \( n \)-bit numbers.

**Problem 6.27** Write a Verilog structural description for ALU described in Example 2.3. Identify the longest path in the resulting circuit. Draw the circuit for \( n = 8 \).

**Problem 6.28** Design a 8-bit combinational multiplier for \( a_7 \ldots a_0 \) and \( b_7 \ldots b_0 \), using as basic “brick” the following elementary multiplier, containing a FA and an AND:

```verilog
module em(carry_out, sum_out, a, b, carry, sum);
  input a, b, carry, sum;
  output carry_out, sum_out;
  assign {carry_out, sum_out} = (a & b) + sum + carry;
endmodule
```

**Problem 6.29** Design an adder for 32 1-bit numbers using the carry save adder approach.

**Hint:** instead of using the direct solution of a binary tree of adders a more efficient way (from the point of view of both size and depth) is to use circuits to “compact” the numbers. The first step is presented in Figure 6.50, where 4 1-bit numbers are transformed in two numbers, a 1-bit number and a 2-bit number. The process is similar in Figure 6.51 where 4 numbers, 2 1-bit numbers and 2 2-bit numbers are compacted as 2 numbers, one 2-bit number and one 3-bit number. The result is a smaller and a faster circuit than a circuit realized using adders.

![Figure 6.50: 4-bit compacter.](image)

Compare the size and depth of the resulting circuit with a version using adders.

**Problem 6.30** Design in Verilog the behavioral and the structural description of a multiply and accumulate circuit, MACC, performing the function: \((a \times b) + c\), where \( a \) and \( b \) are 16-bit numbers and \( c \) is a 24-bit number.

**Problem 6.31** Design the combinational circuit for computing

\[
c = \sum_{i=0}^{7} a_i \times b_i
\]
where: $a_i, b_i$ are 16-bit numbers. Optimize the size and the depth of the 8-number adder using a technique learned in one of the previous problem.

**Problem 6.32** Exemplify the serial composition, the parallel composition and the serial-parallel composition in 0 order systems.

**Problem 6.33** Write the logic equations for the BCD to 7-segment trans-coder circuit in both high active outputs version and low active outputs version. Minimize each of them individually. Minimize all of them globally.

**Problem 6.34** Applying removing rules and reduction rules find the functions performed by 5-level universal circuit programmed by the following binary strings:

1. $(0100)^8$
2. $(01000010)^4$
3. $(0100001011001010)^2$
4. $0^{24}(01000010)$
5. $00000001001001001111000011000011$

**Problem 6.35** Compute the biggest size and the biggest depth of an n-input, 1-output circuit implemented using the universal circuit.

**Problem 6.36** Provide the proof for Zero-One Principle.

### 6.5 Projects

**Project 6.1** Finalize Project 1.1 using the knowledge acquired about the combinational structures in this chapter.
Project 6.2  Design a combinational floating point single precision (32 bit) multiplier according to the ANSI/IEEE Standard 754-1985, Standard for Binary Floating Point Arithmetic.
Chapter 7

MEMORIES:
First order, 1-loop digital systems

In the previous chapter
the main combinational, no-loop circuits were presented with emphasis on

- the simple, pattern-based basic combinational circuits performing functions like: decode using demultiplexors, selection using multiplexors, increment, add, various selectable functions using arithmetic and logic units, compare, shift, priority encoding, ...
- the difference between the simple circuits, which grow according to recursive rules, and the complex, pattern-less circuits whose complexity must be kept at lowest possible level
- the compromise between area and speed, i.e., how to save area accepting to give up the speed, or how can be increased the speed accepting to enlarge the circuit’s area.

In this chapter
the first order, one-loop circuits are introduced studying

- how to close the first loop inside a combinational circuit in order to obtain a stable and useful behavior
- the elementary storage support – the latch – and the way to expand it using the serial, parallel, and serial-parallel compositions leading to the basic memory circuits, such as: the master-slave flip-flop, the random access memory and the register
- how to use first order circuits to design basic circuits for real applications, such as register files, content addressable memories, associative memories or systolic systems.

In the next chapter
the second order, automata circuits are described. While the first order circuits have the smallest degree of autonomy – they are able only to maintain a certain state – the second order circuits have an autonomous behavior induced by the loop just added. The following circuits will be described:

- the simplest and smallest elementary, two-state automata: the T flip-flop and JK flip-flop, which besides the storing function allow an autonomous behavior under a less restrictive external command
- simple automata performing recursive functions, generated by expanding the function of the simplest two-state automata (example: n-bit counters)
- the complex, finite automata used for control or for recognition and generation of regular streams of symbols.
The magic images were placed on the wheel of the memory system to which correspondent other wheels on which were remembered all the physical contents of the terrestrial world – elements, stones, metals, herbs, and plants, animals, birds, and so on – and the whole sum of the human knowledge accumulated through the centuries through the images of one hundred and fifty great men and inventors. The possessor of this system thus rose above time and reflected the whole universe of nature and of man in his mind.

Frances A. Yates

A true memory is an associative one. Please do not confuse the physical support – the random access memory – with the function – the associative memory.

According to the mechanisms described in Chapter 3 of this book, the step toward a new class of circuits means to close a new loop. This will be the first loop which closed over the combinational circuits already presented. Thus, a first degree of autonomy will be reached in digital systems: the autonomy of the state of the circuit. Indeed, the state of the circuit will be partially independent by the input signals, i.e., the output of the circuits do not depend on or not respond to certain input switching.

In this chapter we introduce some of the most important circuits used for building digital systems. The basic function in which they are involved is the memory function. Some events on the input of a memory circuit are significant for the state of the circuits and some are not. Thus, the circuit “memorizes”, by the state it reaches, the significant events and “ignores” the rest. The possibility to have an “attitude” against the input signals is given to the circuit by the autonomy induced by its internal loop. In fact, this first loop closed over a simple combinational circuit makes insignificant some input signals because the circuit is able to compensate their effect using the signals received back from its output.

The main circuits with one internal loop are:

- **the elementary latch** - the basic circuit in 1-OS, containing two appropriately loop-coupled gates; the circuit has two stable states being able to store 1 bit of information

- **the clocked latch** - the first digital circuit which accepts the clock signal as an input distinct from data inputs; the clock signal determines by its active level when the latch is triggered, while the data input determines how the latch switches

- **the master-slave flip-flop** - the serial composition in 1-OS, built by two clocked latches serially connected; results a circuit triggered by the active transition of clock

- **the random access memory (RAM)** - the parallel composition in 1-OS, containing a set of $n$ clocked elementary latches accessed with a $DMUX_{\log_2 n}$ and a $MUX_{\log_2 n}$

- **the register** - the serial-parallel composition in 1-OS, made by parallel connecting master-slave flip-flops.

---

1She was Reader in the History of the Renaissance at the University of London. The quote is from Giordano Bruno and the Hermetic Tradition. Her other books include The Art of Memory.
These first order circuits don’t have a direct computational functionality, but are involved in supporting the following main processes in a computational machine:

- offer the storage support for implementing various memory functions (register files, stacks, queues, content addressable memories, associative memories, ...)
- are used for synchronizing different subsystems in a complex system (supports the pipeline mechanism, implements delay lines, stores the state of automata circuits).

### 7.1 Stable/Unstable Loops

There are two main types of loops closed over a combinational logic circuit: loops generating a stable behavior and loops generating an unstable behavior. We are interested in the first kind of loop that generates a *stable state* inside the circuit. The other loop cannot be used to build anything useful for computational purposes, except some low performance signal generators.

The distinction between the two types of loops is easy exemplified closing loops over the simplest circuit presented in the previous chapter, the elementary decoder (see Figure 7.1a).

The **unstable loop** is closed connecting the output $y_0$ of the elementary decoder to its input $x_0$ (see Figure 7.1b). Suppose that $y_0 = 0 = x_0$. After the time interval equal with $t_{pLH}$ the output $y_0$ becomes 1. After another time interval equal with $t_{pHL}$ the output $y_0$ becomes again 0. And so on, the two outputs of the decoder are unstable oscillating between 0 and 1 with a period of time $T_{osc} = t_{pLH} + t_{pHL}$, or the frequency $f_{osc} = 1/(t_{pLH} + t_{pHL})$.

The **stable loop** is obtained connecting the output $y_1$ of the elementary decoder to the input $x_0$ (see Figure 7.1c). If $y_1 = 0 = x_0$, then $y_0 = 1$ fixing again the value 0 to the output $y_1$. If $y_1 = 1 = x_0$, then $y_0 = 0$ fixing again the value 1 to the output $y_1$. Therefore, the circuit has two stable states. (For the moment we don’t know how to switch from one state to another state, because the circuit has no input to command the switching from 0 to 1 or conversely. The solution comes soon.)

---

Figure 7.1: **The two loops closed over an elementary decoder.**

a. The simplest combinational circuit: the one-input, elementary decoder.  
b. The unstable, inverting loop containing one (odd) inverting logic level(s).  
c. The stable, non-inverting loop containing two (even) inverting levels.
What is the main structural distinction between the two loops?

- The unstable loop has an odd number of inverting levels, thus the signal comes back to the output having the complementary value.
- The stable loop has an even number of inverting levels, thus the signal comes back to the output having the same value.

**Example 7.1** Let be the circuit from Figure 7.2a, with 3 inverting levels on its internal loop. If the command input C is 0, then the loop is “opened”, i.e., the flow of the signal through the circular way is interrupted. If C switches in 1, then the behavior of the circuit is described by the wave forms represented in Figure 7.2b. The circuit generates a periodic signal with the period \( T_{osc} = 3(t_{pLH} + t_{pHL}) \) and frequency \( f_{osc} = 1/(3(t_{pLH} + t_{pHL})) \). (To keep the example simple we consider that \( t_{pLH} \) and \( t_{pHL} \) have the same value for the three circuits.)

![Figure 7.2: The unstable loop. The circuit version used for a low-cost and low-performance clock generator.](image)

a. The circuit with a three (odd) inverting circuits loop coupled. b. The wave forms drawn taking into account the propagation times associated to the low-high transitions (\( t_{pLH} \)) and to the high-low transitions (\( t_{pHL} \)).

In order to be useful in digital applications, a loop closed over a combinational logic circuit must contain an even number of inverting levels for all binary combinations applied to its inputs. Else, for certain or for all input binary configurations, the circuit becomes unstable, useless for implementing computational functions. In the following, only even (in most of cases two) number of inverting levels are used for building the circuits belonging to 1-OS.

### 7.2 The Serial Composition: the Edge Triggered Flip-Flop

The first composition in 1-order systems is the serial composition, represented mainly by:

- the master-slave structure as the main mechanism that avoids the transparency of the storage structures
• the delay flip-flop, the basic storage circuit that allows to close the second loop in the synchronous digital systems

• the serial register, the first big and simple memory circuit having a recursive definition.

This class of circuits allows us to design synchronous digital systems. Starting from this point the inputs in a digital system are divided in two categories:

• clock inputs for synchronizing different parts of a digital system

• data and control inputs that receive the “informational” flow inside a digital system.

7.2.1 The Serial Register

Starting from the delay function of the last presented circuit (see Figure 2.14) a very important function and the associated structure can be defined: the serial register. It is very easy to give a recursive definition to this simple circuit.

**Definition 7.1** An n-bit serial register, $SR_n$, is made by serially connecting a D flip-flop with an $SR_{n-1}$. $SR_1$ is a D flip-flop. ◻

In Figure 7.3 is shown a $SR_n$. It is obvious that $SR_n$ introduces a $n$ clock cycle delay between its input and its output. The current application is for building digital controlled “delay lines”.

![Figure 7.3: The n-bit serial register ($SR_n$).](image)

We hope that now it is very clear what is the role of the master-slave structure. Let us imagine a “serial register built with D latches”! The transparency of each element generates the strange situation in which at each clock cycle the input is loaded in a number of latches that depends by the length of the active level of the clock signal and by the propagation time through each latch. Results an uncontrolled system, useless for any application. Therefore, for controlling the propagation with the clock signal we must use the master-slave, non-transparent structure of D flip-flop that switches on the positive or negative edge of clock.

**VeriSim 7.1** The functional description currently used for an n-bit serial register active on the positive edge of clock is:

```verilog
module serial_register #(parameter n = 1024)(output out ,
               input in, enable, clock);

reg[0:n-1] serial_reg;

assign out = serial_reg[n-1];
```
always @(posedge clock)
  if (enable) serial_reg = {in, serial_reg[0:n-2]};
endmodule

7.3 The Parallel Composition: the Random Access Memory

The parallel composition in 1-OS provides the random access memory (RAM), which is the main storage support in digital systems. Both, data and programs are stored on this physical support in different forms. Usually we call these circuits improperly memories, even if the memory function is something more complex, which suppose besides a storage device a specific access mechanism for the stored information. A true memory is, for example, an associative memory (see the next subchapters about applications), or a stack memory (see next chapter).

This subchapter introduces two structures:

- a trivial composition, but a very useful circuit: the \textit{n-bit latch}
- the asynchronous \textit{random access memory} (RAM),

both involved in building big but simple recursive structures.

7.3.1 The \textit{n}-Bit Latch

The \textit{n}-bit latch, \(L_n\), is made by parallel connecting \(n\) data latches clocked by the same CK. The system has \(n\) inputs and \(n\) outputs and stores an \(n\)-bit word. \(L_n\) is a transparent structure on the active level of the CK signal. The \(n\)-bit latch must be distinguished by the \(n\)-bit register (see the next section) that switches on the edge of the clock. In a synchronous digital system is forbidden to close a combinational loop over \(L_n\).

\textbf{VeriSim 7.2} A 16-bit latch is described in Verilog as follows:

```verilog
module n_latch #(parameter n = 16)(output reg [n-1:0] out ,
  input [n-1:0] in ,
  input clock);
  always @(in or clock)
    if (clock == 1) // the active-high clock version
      //if (clock == 0) // the active-low clock version
      out = in;
  endmodule
```

The \(n\)-bit latch works like a memory, storing \(n\) bits. The only deficiency of this circuit is due to the access mechanism. We must control the value applied on all \(n\) inputs when the latch changes its content. More, we can not use selectively the content of the latch. The two problems are solved adding some combinational circuits to limit both the changes and the use of the stored bits.
7.3. THE PARALLEL COMPOSITION: THE RANDOM ACCESS MEMORY

7.3.2 Asynchronous Random Access Memory

Adding combinational circuits for accessing in a more flexible way an \( m \)-bit latch for write and read operations, results one of the most important circuits in digital systems: the random access memory. This circuit is the biggest and simplest digital circuit. And we can say it can be the biggest because it is the simplest.

![The principle of the random access memory (RAM)](image)

**Definition 7.2** The \( m \)-bit random access memory, \( \text{RAM}_m \), is a linear collection of \( m \) \( D \) (data) latches parallel connected, with the 1-bit common data inputs, DIN. Each latch receives the clock signal distributed by a DMUX\( \log_2 m \). Each latch is accessed for reading through a MUX\( \log_2 m \). The selection code is common for DMUX and MUX and is represented by the \( p \)-bit address code: \( A_p-1; \ldots; A_0 \), where \( p = \log_2 m \).

The logic diagram associated with the previous definition is shown in Figure 7.4. Because no one of the input signal is clock related, this version of RAM is considered an asynchronous one. The signal \( WE' \) is the low-active write enable signal. For \( WE' = 0 \) the write operation is performed in the memory cell selected by the address \( A_{n-1}; \ldots; A_0 \). The wave forme describing the relation between the input and output signals of a RAM are represented in Figure 7.5, where the main time restrictions are the followings:

- \( t_{\text{ACC}} \): access time - the propagation time from address input to data output when the read operation is performed; it is defined as a minimal value
- \( t_w \): write signal width - the length of active level of the write enable signal; it is defined as the shortest time interval for a secure writing

---

\(^3\)The actual implementation of this system uses optimized circuits for each 1-bit storage element and for the access circuits. See Appendix C for more details.)
Figure 7.5: **Read and write cycles for an asynchronous RAM.** Reading is a combinational process of selecting. The access time, $t_{\text{ACC}}$, is given by the propagation through a big MUX. The write enable signal must be strictly included in the time interval when the address is stable (see $t_{\text{ASU}}$ and $t_{\text{AH}}$). Data must be stable related to the positive transition of $WE'$ (see $t_{\text{DSU}}$ and $t_{\text{DH}}$).

- $t_{\text{ASU}}$: address set-up time related to the occurrence of the write enable signal; it is defined as a minimal value for avoiding to disturb the content of other than the storing cell selected by the current address applied on the address inputs
- $t_{\text{AH}}$: address hold time related to the end transition of the write enable signal; it is defined as a minimal value for similar reasons
- $t_{\text{DSU}}$: data set-up time related to the end transition of the write enable signal; it is defined as a minimal value that ensure a proper writing
- $t_{\text{DH}}$: data hold time related to the end transition of the write enable signal; it is defined as a minimal value for similar reasons.

The just described version of a RAM represents only the asynchronous core of a memory subsystem, which must have a synchronous behavior in order to be easy integrated in a robust design. In Figure 7.4 there is no clock signal applied to the inputs of the RAM. In order to synchronize the behavior of this circuit with the external world, additional circuits must be added (see the first application in the next subchapter: **Synchronous RAM**).

The actual organization of an asynchronous RAM is more elaborated in order to provide the storage support for a big number of $m$-bit words.

**VeriSim 7.3** The functional description of a asynchronous $n = 2^p$ $m$-bit words RAM follows:

```verilog
module ram(input [m-1:0] din , // data input
           input [p-1:0] addr, // address
           input we , // write enable
           output [m-1:0] dout); // data out
reg [m-1:0] mem[(1'b1<<p)-1:0]; // the memory
```
assign dout = mem[addr]; // reading memory
always @(din or addr or we) if (we) mem[addr] = din; // writing memory
endmodule

The real structural version of the storage array will be presented in two stages. First the number of bits per word will be expanded, then the e solution for a big number of words number of words will be presented.

Expanding the number of bits per word

The pure logic description offered in Figure 7.4 must be reconsidered in order (1) to optimize it and (2) to show how the principle it describe can be used for designing a many-bit word RAM. The circuit structure from Figure 7.6 represents the $m$-bit word RAM. The circuit is organized in $m$ columns, one for each bit of the $m$-bit word. The DMUX structure is shared all by the $m$ columns, while each column has its own MUX structure. Let us remember that both, the DMUX and MUX circuits are structured around a DCD. See Figure 6.6 and 6.9, where the first level in both circuits is a decoder, followed by a linear network of 2-input ANDs for DMUX, and by an AND-OR circuit for MUX. Then, only one decoder, DCD$_p$, must be provided for the entire memory. It is shared by the demultiplexing function and by the $m$ multiplexors. Indeed, the outputs of the decoder, LINE$_{p-1}$, ... LINE$_1$, LINE$_0$, are used to drive:

- one AND$_2$ gate associate cu each line in the array, whose output clocks the DL latches associated to one word; with these gates the decoder forms the demultiplexing circuit used to clock, when WE = 1, the latches selected (addressed) by the current value of the address: $A_{p-1}, ..., A_0$

- $m$ AND$_2$ gates, one in each column, selecting the read word to be ORed to the outputs DOUT$_{m-1}$, DOUT$_{m-2}$, ... DOUT$_0$; with the AND-OR circuit from each COLUMN the decoder forms the multiplexor circuit associated to each output bit of the memory.

The array of lathes is organized in $n$ and $m$ columns. Each line is driven for write by the output of a demultiplexer, while for the read function the addressed line (word) is selected by the output of a decoder. The output value is gathered from the array using $m$ multiplexors.

The reading process is a pure combinational one, while the writing mechanism is an asynchronous sequential one. The relation between the WE signal and the address bits is very sensitive. Due to the combinational hazard to the output of DCD, the WE’ signal must be activated only when the DCD’s outputs are stabilized to the final value, i.e., $t_{ASU}$ before the fall edge of WE’ or $t_H$ after the rise edge of WE’.

Expanding the number of words by two dimension addressing

The factor form on silicon of the memory described in Figure 7.6 is very unbalanced for $n >> m$. Expanding the number of words for the a RAM in the previous, one block version is not efficient because request a complex lay-out involving very long wires. We are looking for a more “squarish” version of the lay-out for a big memory. The solution is to connect in parallel many $m$-column blocks, thus defining a many-word from which to select one word using another level of multiplexing. The reading process selects the many-word containing the requested word from which the requested word is selected.
The internal organization of memory is now a two dimension array of rows and columns. Each row contains a many-word of $2^q$ words. Each column contains a number of $2^r$ words. The memory is addressed using the $(p = r + q)$-bit address:

$$\text{addr}[p-1:0] = \{\text{rowAddr}[r-1:0], \text{colAddr}[q-1:0]\}$$

The row address $\text{rowAddr}[r-1:0]$ selects a many-word, while from the selected many-word, the column address $\text{colAddr}[q-1:0]$ selects the word addressed by the address $\text{addr}[p-1:0]$. Playing with the values of $r$ and $q$ an appropriate lay-out of the memory array can be designed.

In Figure 7.7 the block schematic for the resulting memory is presented. The second decoder – COLUMN DECODE – selects from the $s$ $m$-bit words provided by the $s$ COLUMN BLOCKs the word addressed by $\text{addr}[p-1:0]$.

While the size decoder for a one block memory version is in the same order with the number of words $(S_{\text{CD}} \leq 2^p)$, the sum of the sizes of the two decoders in the two dimension version is much smaller, because usually $2^p >> 2^r + 2^q$, for $p = r + q$. Thus, the area of the memory circuit is dominated only by the storage elements.
7.3. THE PARALLEL COMPOSITION: THE RANDOM ACCESS MEMORY

Figure 7.7: **RAM version with two dimension storage array.** A number of m-bit blocks are parallel connected and driven by the same row decoder. The column decoder selects to output an m-bit word from the (s \times m)-bit row.
The second level of selection is based also on a shared decoder – COLUMN DECODER. It forms, with the s two-input ANDs a $DMUX_q$ – the q-input DMUX in Figure 7.7 – which distributes the write enable signals, we, to the selected m-column block. The same decoder is shared by the $m s$-input MUXs used to select the output word from the many-word selected by ROW DECODE.

The well known principle of ”divide et impera” (divide and conquer) is applied when the address is divided in two parts, one for selecting a row and another for selecting a column. The access circuits is thus minimized.

Unfortunately, RAM has not the function of memorizing. It is only a storage support. Indeed, if we want to “memorize” the number 13, for example, we must store it to the address 131313, for example, and to keep in mind (to memorize) the value 131313, the place where the number is stored. And than, what’s the help provided us by a the famous RAM memory? No one. Because RAM is not a memory, it becomes a memory only if the associated processor runs an appropriate procedure which allows us to forget about the address 131313. Another solution is provided by additional circuits used to improve the functionality (see the subsection about Associative Memories.)

7.4 Applications

Composing basic memory circuits with combinational structures result typical system configurations or typical functions to be used in structuring digital machines. The pipeline connection, for example, is a system configuration for speeding up a digital system using a sort of parallelism. This mechanism is already described in the subsections 2.5.1 Pipelined connections, and 3.3.2 Pipeline structures. Few other applications of the circuits belonging to 1-OS are described in this section. The first is a frequent application of 1-OS: the synchronous memory, obtained adding clock triggered structures to an asynchronous memory. The next is the file register – a typical storage subsystem used in the kernel of the almost all computational structures. The basic building block in one of the most popular digital device, the Field Programmable Gate Array, is also SRAM based structure. Follows the content addressable memory which is a hardware mechanism useful in controlling complex digital systems or for designing genuine memory structures: the associative memories.

7.4.1 Synchronous RAM

It is very hard to consider the time restriction imposed by the wave forms presented in Figure 7.5 when the system is requested to work at high speed. The system designer will be more comfortable with a memory circuit having all the time restrictions defined related only to the active edge of the system clock. The synchronous RAM (SRAM) is conceived to have all time relations defined related to the active edge of the clock signal. SRAM is the preferred embodiment of a storage circuit in the contemporary designs. It performs write and read operations synchronized with the active edge of the clock signal (see Figure 7.8).

VeriSim 7.4 The functional description of a synchronous RAM (0.5K of 64-bit words) follows:

```vhdl
module sram(
    input [63:0] din ,  // data input
    input [8:0] addr ,  // address
    input we, clk ,  // write enable & clock signal
    output reg [63:0] dout );  // data out
```
7.4. APPLICATIONS

Figure 7.8: Read and write cycles for SRAM. For the flow-through version of a SRAM the time behavior is similar to a register. The set-up and hold time are defined related to the active edge of clock for all the input connections: data, write-enable, and address. The data output is also related to the same edge.

```verilog
reg [63:0] mem[511:0];
always @(posedge clk) if (we) dout <= din ;
    else dout <= mem[addr] ;
always @(posedge clk) if (we) mem[addr] <= din ;
endmodule
```

The previously described SRAM is the flow-through version of a SRAM. A pipelined version is also possible. It introduces another clock cycle delay for the output data.

7.4.2 Register File

The most accessible data in a computational system is stored in a small and fast memory whose locations are usually called machine registers or simply registers. In most usual embodiment they have actually the physical structure of a register. The machine registers of a computational (processing) element are organized in what is called register file. Because computation supposes two operands and one result in most of cases, two read ports and one write port are currently provided to the small memory used as register file (see Figure 7.9).

VeriSim 7.5 Follows the Verilog description of a register file containing 32 32-bit registers. In each clock cycle any two pair of registers can be accessed to be used as operands and a result can be stored in any one register.
Figure 7.9: Register file. In this example it contains $2^n$ $m$-bit registers. In each clock cycle any two registers can be read and writing can be performed in anyone.

```verilog
module register_file( 
    output [31:0] left_operand, 
    output [31:0] right_operand, 
    input [31:0] result, 
    input [4:0] left_addr, 
    input [4:0] right_addr, 
    input [4:0] dest_addr, 
    input write_enable, 
    input clock );

    reg [31:0] file[0:31];

    assign left_operand = file[left_addr],
    right_operand = file[right_addr];

    always @(posedge clock) if (write_enable) file[dest_addr] <= result;
endmodule
```

The internal structure of a register file can be optimized using $m \times 2^n$ 1-bit clocked latches to store data and $2m$-bit clocked latches to implement the master-slave mechanism.

### 7.4.3 Field Programmable Gate Array – FPGA

Few decades ago the prototype of a digital system was realized in a technology very similar with the one used for the final form of the product. Different types of standard integrated circuits where connected according to the design on boards using a more or less flexible interconnection technique. Now we do not have anymore standard integrated circuits, and making an Application Specific Integrated Circuit (ASIC) is a very expensive adventure. Fortunately, now there is a wonderful technology for prototyping (which can be used also for small production chains). It is based on a one-chip system called Field Programmable Gate Array – FPGA. The name comes from its flexibility to be configured by the user after manufacturing, i.e., “in the field”. This generic circuit can be programmed to perform any digital function.

In this subsection the basic configuration of an FPGA circuit will be described. The internal cellular...
structure of the system is described for the simplest implementation, letting aside details and improvements used by different producer on this very diverse market (each new generation of FPGA integrates different usual digital blocks in order to help efficient implementations; for example: multipliers, block RAMs, ...; learn more about this from the on-line documentation provided by the FPGA producers).

![Diagram of FPGA organization]

Figure 7.10: Top level organization of FPGA.

The system level organization of an FPGA

The FPGA chip has a cellular structure with three main programmable components, whose function is defined by setting on 0 or on 1 control bits stored in memory elements. An FPGA can be seen as a big structured memory containing million of bits used to control the state of million of switches. The main type of cells are:

- **Configurable Logic Blocks** (CLB) used to perform a programmable combinational and/or sequential function

- **Switch Nodes** which interconnect in the most flexible way the CLB modules and some of them to the IO pins, using a matrix of programmed switches

- **Input-Output Interfaces** are two-direction programmable interfaces, each one associated with an IO pin.
Figure 7.10 provides a simplified representation of the internal structure of an FPGA at the top level. The area of the chip is filled up with two interleaved arrays. One of the CLBs and another of the Switch Nodes. The chip is boarded by IO interfaces.

The entire functionality of the system can be programmed by an appropriate binary configuration distributed in all the cells. For each IO pin is enough one bit to define if the pin is an input or an output. For a Switch Node more bits are needed because each switch asks for 6 bits to be configured. But, most of bits (in some implementations more than 100 per CLB) are used to program the functions of the combinational and sequential circuits in each node containing a CLB.

**The IO interface**

Each signal pin of the FPGA chip can be assigned to be an input or an output. The simplest form of the interface associated to each IO pin is presented in Figure 7.11, where:

- **D-FF0**: is the D master-slave flip-flop which synchronously receives the value of the I/O pin through the associated input non-inverting buffer
- **m**: the storage element which contains the 1-bit program for the input interface used to command the tristate buffer; if \( m = 1 \) then the tristate buffer is enabled and interface is in the output mode, else the tristate buffer is disabled and interface is in the input mode
- **D-FF1**: is the flip-flop loaded synchronously with the output bit to be sent to the I/O pin if \( m = 1 \).

![Input-Output interface](image)

Figure 7.11: **Input-Output interface.**

The storage element \( m \) is part of the big distributed RAM containing all the storage elements used to program the FPGA.

**The switch node**

The switch node (Figure 7.12a) consists of a number of programmable switches (4 in our description). Each switch (Figure 7.12b) manages 4 wires, connection them in different configurations using 6 nMOS transistors, each commanded by the state of 1-bit memory (Figure 7.12c). If \( m_i = 1 \) then the associated
nMOS transistor is on and between its drain end source the resistor has a small value. If \( m_1 = 0 \) then the associated nMOS transistor is off and the two ends of the switch are not connected.

For example, the configuration shown in Figure 7.12d is programmed as follows:

\[
\begin{align*}
\text{switch 0} &: \{m_0, m_1, m_2, m_3, m_4, m_5\} = 011010; \\
\text{switch 1} &: \{m_0, m_1, m_2, m_3, m_4, m_5\} = 101000; \\
\text{switch 2} &: \{m_0, m_1, m_2, m_3, m_4, m_5\} = 000001; \\
\text{switch 3} &: \{m_0, m_1, m_2, m_3, m_4, m_5\} = 010000;
\end{align*}
\]

Any connection is a two-direction connection.

**The basic building block**

Because any digital circuit can be composed by properly interconnected gates and flip-flops, each CLB contains a number of basic building blocks, called bit slices (BSs), each able to provide at least an \( n \)-input, 1-output programmable combinational circuit and an 1-bit register.

In the previous chapter was presented an Universal combinational circuit: the \( n \)-input multiplexer able to perform any \( n \)-variable Boolean function. It was programmed applying on its selected inputs an \( m \)-bit binary configuration (where \( m = 2^n \)). Thereby, an \( MUX_n \) and a memory for storing the \( m \)-bit program provide the structure able to be programmed to perform any \( n \)-input 1-output combinational circuit. In Figure 7.13 it is represented, for \( n = 4 \), by the multiplexer \( MUX \) and the 16 memory elements \( m_0, m_1, \ldots, m_{15} \). The entire sub-module is called LUT (from look-up table). The memory elements \( m_0, m_1, \ldots, m_{15} \), being part of the big distributed RAM of the FPGA chip, can be loaded with any out of 65536 binary configuration used to define the same number of 4-input Boolean function.

Because the arithmetic operations are very frequently used the BS contains a specific circuit for any arithmetic operation: the circuit computing the value of the carry signal. The module carry Figure 7.13 has also its specific propagation path defined by a specific input, carryIn, and a specific output carryOut.

The BS module contains also the one-bit register D-FF. Its contribution can be considered in the current design if the memory element \( m_d \) is programmed appropriately. Indeed, if \( m_d = 1 \), then the
output of the BS comes from the output of D-FF, else the output of the BS is a combinational one, the flip-flop being shortcut.

The memory element \(mc\) is used to program the selection of the LUT output or of the Carry output to be considered as the programmable combinational function of this BS.

The total number of bits used to program the function of the BS previously described is 18. Real FPGA circuits are now featured with much more complex BSs (please search on their web pages for details).

There are two kinds of BS: logic type and memory type. The logic type uses LUT to implement combinational functions. The memory type uses LUT for implementing both, combinatorial functions and memory function (RAM or serial shift register).

The configurable logic block

The main cell used to build an FPGA, CLB (see Figure 7.10) contains many BSs organized in slices. The most frequent organization is of 2 slices, each having 4 BSs (see Figure 7.14). There are slices containing logic type BSs (usually called SLICEL), or slices containing memory type BSs (usually called SLICEM). Some CLBs are composed by two SLICEL, others are composed by one SLICEL and one SLICEM.

A slice has some fix connections between its BSs. In our simple description, the fix connections refers to the carry chain connections. Obviously, we can afford to make fix connections for circuits having specific function.

7.4.4 * Content Addressable Memory

A normal way to “question” a memory circuit is to ask for:

\[ Q_1: \text{the value of the property } A \text{ of the object } B \]
Figure 7.14: Configurable logic block.

For example: How old is George? The age is the property and the object is George. The first step to design an appropriate device to be questioned as previously is exemplified is to define a circuit able to answer the question:

Q2: where is the object B?

with two possible answers:

1. the object B is not in the searched space
2. the object B is stored in the cell indexed by X.

The circuit for answering Q2-type questions is called Content Addressable Memory, shortly: CAM. (About the question Q1 in the next subsection.)

The basic cell of a CAM is consists of:

- the storage elements for binary objects
- the “questioning” circuits for searching the value applied to the input of the cell.

In Figure 7.15 there are 4 D latches as storage elements and four XORs connected to a 4-input NAND used as comparator. The cell has two functions:

- to store: the active level of the clock modify the content of the cell storing the 4-bit input data into the four D latches
- to search: the input data is continuously compared with the content of the cell generating the signal AO′ = 0 if the input matches the content.

The cell is written as an m-bit latch and is continuously interrogated using a combinational circuit as comparator. The resulting circuit is an 1-OS because results serially connecting a memory, one-loop circuit with a combinational, no-loop circuit. No additional loop is involved.

An n-word CAM contains n CAM cells and some additional combinational circuits for distributing the clock to the selected cell and for generating the global signal M, activated for signaling a successful match between the input value and one or more cell contents. In Figure 7.16a a 4-word of 4 bits each is represented. The write enable, WE, signal is demultiplexed as clock to the appropriate cell, according to the address codded by A1A0. The 4-input NAND generate the signal M. If, at least one address output, AO′ is zero, indicating match in the corresponding cell, then M = 1 indicating a successful search.

The input address A_{p-1},...,A_0 is binary coddod on p = \log_2 n bits. The output address AO_{n-1},...,AO_0 is an unary code indicating the place or the places where the data input D_{m-1},...,D_0 matches the content of the cell. The output address must be unary coddod because there is the possibility of match in more than one cell.
Figure 7.15: **The Content Addressable Cell.**  

**a.** The structure: data latches whose content is compared against the input data using 4 XORs and one NAND. Write is performed applying the clock with stable data input.  

**b.** The logic symbol.

Figure 7.16b represents the logic symbol for a CAM with \( n \) \( m \)-bit words. The input \( WE \) indicate the function performed by CAM. Be very careful with the set-up time and hold time of data related to the \( WE \) signal!

The CAM device is used to locate an object (to answer the question \( Q2 \)). Dealing with the properties of an object (answering \( Q1 \)-type questions) means to use o more complex devices which associate one or more properties to an object. Thus, the **associative memory** will be introduced adding some circuits to CAM.

### 7.4.5 * An Associative Memory *

A partially used RAM can be an associative memory, but a very inefficient one. Indeed, let be a RAM addressed by \( A_{n-1} \ldots A_0 \) containing 2-field words \( \{ V, D_{m-1} \ldots D_0 \} \). The objects are coded using the address, the values of the unique property \( P \) are coded by the data field \( D_{m-1} \ldots D_0 \). The one-bit field \( V \) is used as a validation flag. If \( V = 1 \) in a certain location, then there is a match between the object designated by the corresponding address and the value of property \( P \) designated by the associated data field.

**Example 7.2** Let be the 1Mword RAM addressed by \( A_{19} \ldots A_0 \) containing 2-field 17-bit words \( \{ V, D_{15} \ldots D_0 \} \). The set of objects, \( OBJ \), are coded using 20-bit words, the property \( P \) associated to \( OBJ \) is coded using 16-bit words. If

\[
\text{RAM}\left[11110000111100001111\right] = 1_0011001111110000 \\
\text{RAM}\left[11110000111100001010\right] = 0_0011001111110000
\]

then:

- for the object \( 11110000111100001111 \) the property \( P \) is defined (\( V = 1 \)) and has the value 0011001111110000

- for the object \( 11110000111100001010 \) the property \( P \) is not defined (\( V = 0 \)) and the data field is meaningless.

Now, let us consider the 20-bit address codes four-letter names using for each letter a 5-bit code. How many locations in this memory will contain the field \( V \) instantiated to 1? Unfortunately, only extremely few of them, because:
7.4. APPLICATIONS

a. A 4-word CAM is built using 4 content addressable cells, a demultiplexor to distribute the write enable (WE) signal, and a NAND4 to generate the match signal (M).

b. The logic symbol.

- only 24 from 32 binary configurations of 5 bits will be used to code the 24 letters of Latin alphabet ($24^4 < 2^{20}$)
- but more important: how many different name expressed by 4 letters can be involved in a real application? Usually no more than few hundred, meaning almost nothing related to $2^{20}$.

The previous example teaches us that a RAM used as associative memory is a very inefficient solution. In real applications are used names coded very inefficiently:

$$\text{number of possible names} >> \text{number of actual names}.$$ 

In fact, the natural memory function means almost the same: to remember about something immersed in a huge set of possibilities.

One way to implement an efficient associative memory is to take a CAM and to use it as a programmable decoder for a RAM. The (extremely) limited subset of the actual objects are stored into a CAM, and the address outputs of the CAM are used instead of the output of a combinational decoder to select the accessed location of a RAM containing the value of the property $P$. In Figure 7.17 this version of an associative memory is presented. $\text{CAM}_{m \times n}$ is usually dimensioned with $2^m >> n$ working as a decoder programmed to decode any very small subset of $n$ addresses expressed by $m$ bits.

Here are the three working mode of the previously described associative memory:

**define object**: write the name of an object to the selected location in CAM

- $wa' = 0$, $address = \text{name of object}$, $sel = \text{cam address}$
- $wd' = 1$, $din = \text{don't care}$

**associate value**: write the associated value in the randomly accessed array to the location selected by the active address output of CAM

- $wa' = 1$, $address = \text{name of object}$, $sel = \text{don't care}$
- $wd' = 0$, $din = \text{value}$

---

Figure 7.16: The Content Addressable Memory (CAM). a. A 4-word CAM is built using 4 content addressable cells, a demultiplexor to distribute the write enable (WE) signal, and a NAND4 to generate the match signal (M). b. The logic symbol.
Figure 7.17: An associative memory (AM). The structure of an AM can be seen as a RAM with a programmable decoder implemented with a CAM. The decoder is programmed loading CAM with the considered addresses.

**search**: search for the value associated with the name of the object applied to the address input

- wa' = 1, address = name_of_object, sel = don't care
- wd' = 1, din = don't care
- dout is valid only if valide_dout = 1.

This associative memory will be dimensioned according to the dimension of the actual subset of names, which is significantly smaller than the virtual set of the possible names ($2^p << 2^m$). Thus, for a searching space with the size in $O(2^m)$ a device having the size in $O(2^p)$ is used.

### 7.4.6 Beneš-Waxman Permutation Network

In 1968, even though it was defined by others before, Václav E. Beneš promoted a permutation network [Benes '68] and Abraham Waxman published an optimized version [Waksman '68] of it.

A permutation circuit is a network of programmable switching circuits which receives the sequence $x_1; \ldots; x_n$ and can be programmed to provide on its outputs any of the $n!$ possible permutations.

The two-input programmable switching circuit is represented in Figure 7.18 a. It consists of a D-FF to store the programming bit and two 2-input multiplexors to perform the programmed switch. The circuit works as follows:

- **D-FF = 0**: $x'_1 = x_1$ and $x'_2 = x_2$
- **D-FF = 1**: $x'_1 = x_2$ and $x'_2 = x_1$

The input enable allows to load D-FF with the programming bit. The storage element is a master-slave structure because in a complex network the D flip-flops are chained because they are loaded with the programming bits by shifting. The logic symbol for this elementary circuit is represented in Figure 7.18 b (the clock input and enable input are omitted for simplicity).

Beneš-Waxman permutation network (we must recognize credit for this circuit, at least, for both, Beneš and Waxman) with $n$ inputs has the recursive definition presented in Figure 7.18 c. (The only difference between the definition provided by Beneš and the optimization done by Waxman refers to the number of switches on the output layer: in Waxman’s approach there are only $n/2 - 1$ switches, instead of $n$ for the version presented by Beneš.)

**Theorem 7.1** The switches of Beneš-Waxman permutation network can be set to realize any permutation.

**Proof.** For $n = 2$ the permutation network is a programmable switch circuit. For $n > 2$ we consider, for simplicity, $n$ as a power of 2.

If the two networks $Left_{P_{n/2}}$ and $Right_{P_{n/2}}$ are permutation networks with $n/2$ inputs, then we will prove that it is possible to program the input switches so as each sequence of two successive value on the outputs contains...
values that reach the output switch going through different permutation network. If the “local” order, on the output pair, is not the desired one, then the output switch is used to fix the problem by an appropriate programming.

The following steps can be followed to establish the programming Boolean sequence – \( q_{ln1,\ldots, q_{lnn/2}} \) – for the \( n/2 \) input switches and the programming Boolean sequence – \( q_{Out1,\ldots, q_{Outn/2}} \) – for the \( n/2 - 1 \) output switches:

1. because \( y_1 \leftarrow x_i \), the input switch where \( x_i \) is connected is programmed to \( q_{lni/2} = i + 1 - 2 \times [i/2] \) in order to let \( x_i \) to be applied on the LeftP\( n/2 \) permutation network (if \( i \) is an odd number, then \( q_{lni/2} = 0 \), else \( q_{lni/2} = 1 \)); the \( x_{-(i-1)/2} \) input is the “companion” of \( x_i \) on the same switch; it is consequently routed to the input of RightP\( n/2 \)

2. the output switch \( [j/2] \) is identified using the correspondence \( y_j \leftarrow x_{-(i-1)/2} \); the state of the switch is set to \( s_{Out[j/2]} = j - 2 \times [j/2] \) (if \( j \) is an odd number, then \( q_{Out[j/2]} = 1 \), else \( q_{Out[j/2]} = 0 \))

3. for \( y_{j-(i-1)/2} \leftarrow x_i \), the “companion” of \( y_j \) on the \( j/2 \)-th output switch, we go back to the step 1 until in the step 2 we reach the second output, \( y_2 \); for the first two outputs there is no need of a switch because a partial or the total programming cycle ends when \( y_2 \) receives its value from the output of the RightP\( n/2 \) permutation network

4. if all the output switches are programmed the process stops, else, we start again from the left output of an un-programmed output switch.

Any step 1, let us call it up-step, programs an input switch, while any step 2, let us call it down-step, programs an output switch. Any up-step, which solves the connection \( y_i \leftarrow x_j \), is feasible because the source \( x_j \) is always connected to a still un-programmed switch. Similarly, any down-step is also feasible.

The size and depth of the permutation network \( P_i \) is computed using the relations:

\[
S_P(n) = (2 \times S_P(n/2) + n - 1) \times S_P(2)
\]

\[
S_P(2) \in O(1)
\]

\[
D_P(n) = D_P(n) + 2 \times D_P(2)
\]

\[
D_P(2) \in O(1)
\]

Results:

\[
S_P(n) = (n \times \log_2 n - n + 1) \times S_P(2) \in O(n \log n)
\]

\[
D_P(n) = -1 + 2 \times \log_2 n \in O(n \log n)
\]

**Example 7.3** Let be an 8-input permutation network which must be programmed to perform the following permutation:

\[
\{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\} \rightarrow \{x_8, x_6, x_3, x_1, x_4, x_7, x_5, x_2\} = \{y_1, \ldots, y_8\}
\]

The permutation network with 8 inputs is represented in Figure 7.19. It is designed starting from the recursive definition.

The programming bits for each switch are established according to the algorithm described in the previous proof. In the first stage the programming bits for the first layer – \( \{p_{11}, p_{12}, p_{13}, p_{14}\} \) – and last layer – \( \{p_{52}, p_{53}, p_{54}\} \) – are established, as follows:

1. \( y_1 = x_8 \Rightarrow (x_8 \rightarrow leftP_4) \Rightarrow (p_{14} = 1) \)

2. \( p_{14} = 1 \Rightarrow (x_7 \rightarrow rightP_4) \Rightarrow (p_{53} = 0) \)

3. \( y_5 = x_4 \Rightarrow (x_4 \rightarrow leftP_4) \Rightarrow (p_{12} = 1) \)

4. \( p_{12} = 1 \Rightarrow (x_3 \rightarrow rightP_4) \Rightarrow (p_{52} = 1) \)

5. \( y_4 = x_1 \Rightarrow (x_1 \rightarrow leftP_4) \Rightarrow (p_{11} = 0) \)
Figure 7.18: Beneš-Waxman permutation network. a. The programmable switching circuit. b. The logic symbol for the programmable switching circuit. c. The recursive definition of a \( n \)-input Beneš-Waxman permutation network, \( P_n \).
6. \((p_{11} = 0) \Rightarrow (x_2 \rightarrow \text{right}P_4) \Rightarrow (p_{54} = 0)\)

7. \((y_7 = x_5) \Rightarrow (x_5 \rightarrow \text{left}P_4) \Rightarrow (p_{13} = 0)\)

8. \((p_{13} = 0) \Rightarrow (x_6 \rightarrow \text{right}P_4) \Rightarrow \text{the first stage of programming closes successfully.}\)

In the second stage there are two \(P_4\) permutation networks to be programmed: \(\text{left}P_4\) and \(\text{right}P_4\). From the first stage of programming resulted the following permutations to be performed:

for \(\text{left}P_4\): \(\{x_1,x_4,x_5,x_8\} \rightarrow \{x_8,x_1,x_4,x_5\}\)

for \(\text{right}P_4\): \(\{x_2,x_3,x_6,x_7\} \rightarrow \{x_6,x_3,x_7,x_2\}\)

The same procedure is applied now twice providing the programming bits for the second and the fourth layers of switches.

The last step generate the programming bits for the third layer of switches.

The programming sequences for the five layers of switches are:

\[
\begin{align*}
\text{prog1} &= \{0 \ 1 \ 0 \ 1\} \\
\text{prog2} &= \{1 \ 1 \ 0 \ 0\} \\
\text{prog3} &= \{1 \ 0 \ 1 \ 0\} \\
\text{prog4} &= \{- \ 0 \ - \ 1\} \\
\text{prog5} &= \{- \ 1 \ 0 \ 0\}
\end{align*}
\]
To insert in five clock cycles the programming sequences into the permutation network on the inputs \( \{ p_1, p_2, p_3, p_4 \} \) (see Figure 7.19) are successively applied the following 4-bit words: 0100, 0001, 1010, 1100, 0101. During the insertion the input enable on each switch is activated.

\[ a_{11} a_{12} 0 0 \cdots \]
\[ a_{21} a_{22} a_{23} 0 0 \cdots \]
\[ a_{31} a_{32} a_{33} a_{34} 0 \cdots \]
\[ 0 a_{42} a_{43} a_{44} a_{45} \cdots \]
\[ 0 0 \cdots \cdots \cdots \]
\[ 0 0 0 \cdots \cdots \cdots \]

The main operation executed for matrix-vector operations is multiply and accumulate (MACC):

\[
Z = A \times B + C
\]

for which a specific combinational module is designed. Interleaving MACCs with memory circuits is provided a structure able to compute and to control the flow of data in the same time. The systolic vector-matrix multiplier is represented in Figure 7.20.

The systolic module is represented in Figure 7.20a, where a combinational multiplier \( (M = A \times B) \) is serially connected with an combinational adder \( (M + C) \). The result of MACC operation is latched in the output latch which latches besides the result of the computation, the two input value A and B. The latch is transparent on the high level of the clock. It is used to buffer intermediary results and to control the data propagation through the system.

The system is configured using pairs of modules to generate a master-slave structures, where one module receives \( ck \) and another \( ck' \). The resulting structure is a non-transparent one ready to be used in a pipelined connection.

For a band matrix having the width 4, two non-transparent structures are used (see Figure 7.20c). Data is inserted in each phase of the clock (correlate data insertion with the phase of clock represented in Figure 7.20b) as follows:

The result of the computation is generated sequentially to the output \( y_i \) of the circuit from Figure 7.20b, as follows:

\[
y_1 = a_{11} x_1 + a_{12} x_2 \\
y_2 = a_{21} x_1 + a_{22} x_2 + a_{23} x_3 \\
y_3 = a_{31} x_1 + a_{32} x_2 + a_{33} x_3 + a_{34} x_4 \\
y_4 = a_{42} x_2 + a_{43} x_3 + a_{44} x_4 + a_{45} x_5 \\
y_5 = \ldots
\]

The output X of the module is not used in this application (it is considered for matrix matrix multiplication only). The state of the system in each phase of the clock (see Figure 7.20b) is represented by two quadruples:

\( (Y_1, Y_2, Y_3, Y_4) \)
Figure 7.20: **Systolic vector-matrix multiplier.** a. The module. b. The clock signal with indexed half periods. c. How the modular structure is fed with the data in each half period of the clock signal.
If the initial state of the system is unknown,
\( (Y_1, Y_2, Y_3, Y_4) = (-, -, -) \)
\( (Z_1, Z_2, Z_3, Z_4) = (-, -, -) \)
then the state of the system in the first 10 phases of the clock, numbered in Figure 7.20c, are the following:

Phase: (1)
\( (Y_1, Y_2, Y_3, Y_4) = (-, -, -, -) \)
\( (Z_1, Z_2, Z_3, Z_4) = (-, -, -, -) \)

Phase: (2)
\( (Y_1, Y_2, Y_3, Y_4) = (0, -, -, -) \)
\( (Z_1, Z_2, Z_3, Z_4) = (-, -0, -) \)

Phase: (3)
\( (Y_1, Y_2, Y_3, Y_4) = (0, 0, -, -) \)
\( (Z_1, Z_2, Z_3, Z_4) = (-, 0, 0, 0) \)

Phase: (4)
\( (Y_1, Y_2, Y_3, Y_4) = (x_1, 0, 0, -) \)
\( (Z_1, Z_2, Z_3, Z_4) = (0, 0, 0, 0) \)

(5)
\( (Y_1, Y_2, Y_3, Y_4) = (x_1, x_1, 0, 0) \)
\( (Z_1, Z_2, Z_3, Z_4) = (0, a_{11}x_1, 0, 0) \)

(6)
\( (Y_1, Y_2, Y_3, Y_4) = (x_2, x_1, x_1, 0) \)
\( (Z_1, Z_2, Z_3, Z_4) = (a_{11}x_1 + a_{12}x_2, a_{11}x_1, a_{21}x_1, 0) \)

(7)
\( (Y_1, Y_2, Y_3, Y_4) = (x_2, x_2, x_1, x_1) \)
\( (Z_1, Z_2, Z_3, Z_4) = (y_1, a_{21}x_1 + a_{22}x_2, a_{21}x_1, a_{31}x_1) \)

(8)
\( (Y_1, Y_2, Y_3, Y_4) = (x_3, x_2, x_2, x_1) \)
\( (Z_1, Z_2, Z_3, Z_4) = (a_{21}x_1 + a_{22}x_2 + a_{23}x_3, a_{21}x_1 + a_{22}x_2, a_{31}x_1 + a_{32}x_2, a_{31}x_1) \)

(9)
\( (Y_1, Y_2, Y_3, Y_4) = (x_3, x_3, x_2, x_2) \)
\( (Z_1, Z_2, Z_3, Z_4) = (y_2, a_{31}x_1 + a_{32}x_2 + a_{33}x_3, a_{31}x_1 + a_{32}x_2, a_{42}x_2) \)

(10)
\( (Y_1, Y_2, Y_3, Y_4) = (x_4, x_3, x_2, x_2) \)
\( (Z_1, Z_2, Z_3, Z_4) = (a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4, a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{42}x_2 + a_{43}x_3, a_{42}x_2) \)

(11)
\( (Y_1, Y_2, Y_3, Y_4) = (x_4, x_4, x_3, x_3) \)
\( (Z_1, Z_2, Z_3, Z_4) = (y_3, \ldots) \)

In each clock cycle 4 multiplications and 4 additions are performed. The pipeline connections allow the synchronous insertion and extraction of data. The maximum width of the matrix band determines the number of modules used to design the systolic system.

### 7.5 Concluding About Memory Circuits

For the first time, in this chapter, both composition and loop are used to construct digital systems. The loop adds a new feature and the composition expands it. The chapter introduced only the basic concepts and the main ways to use them in implementing actual digital systems.
The first closed loop in digital circuits latches events. Closing properly simple loops in small combinational circuits very useful effects are obtained. The most useful is the “latch effect” allowing to store certain temporal events. An internal loop is able to determine an internal state of the circuit which is independent in some extent from the input signals (the circuit controls a part of its inputs using its own outputs). Associating different internal states to different input events the circuit is able to store the input event in its internal states. The first loop introduces the first degree of autonomy in a digital system: the autonomy of the internal state. The resulting basic circuit for building memory systems is the elementary latch.

Meaningful circuits occur by composing latches. The elementary latches are composed in different modes to obtain the main memory systems. The serial composition generates the master-slave flip-flop which is triggered by the active edge of the clock signal. The parallel composition introduces the concept of random access memory. The serial-parallel composition defines the concept of register.

Distinguishing between “how?” and “when?” At the level of the first order systems occurs a very special signal called clock. The clock signal becomes responsible for the history sensitive processes in a digital system. Each “clocked” system has inputs receiving information about “how” to switch and another special input – the clock input acting on one of its edge called the active edge of clock – and another special input indicating “when” the system switches. We call this kind of digital systems synchronous systems, because any change inside the system is triggered synchronously by the same edge (positive or negative) of the clock signal.

Registers and RAMs are basic structures. First order systems provide few of the most important type of digital circuits used to support the future developments when new loops will be closed. The register is a synchronous subsystem which, because of its non-transparency, allows closing the next loop leading to the second order digital systems. Registers are used also for accelerating the processing by designing pipelined systems. The random access memory will be used as storage element in developing systems for processing a big amount of data or systems performing very complex computations. Both, data and programs are stored in RAMs.

RAM is not a memory, it is only a physical support. Unfortunately RAM has not the function of memorizing. It is only a storage element. Indeed, when the word W is stored at the address A we must memorize the address A in order to be able to retrieve the word W. Thus, instead of memorizing W we must memorize A, or, as usual, we must have a mechanism to regenerate the address A. In conjunction with other circuits RAM can be used to build systems having the function of memorizing. Any memory system contains a RAM but not only a RAM, because memorizing means more than storing.

Memorizing means to associate. Memorizing means both to store data and to retrieve it. The most “natural” way to design a memory system is to provide a mechanism able to associate the stored data with its location. In an associative memory to read means to find, and to write means to find a free location. The associative memory is the most perfect way of designing a memory, even if it is not always the most optimal as area (price), time and power.

To solve ambiguities a new loop is needed. At the level of the first order systems the second latch problem can not be solved. The system must be more “intelligent” to solve the ambiguity of receiving
synchronously contradictory commands. The system must know more about itself in order to be “able” to behave under ambiguous circumstances. Only a new loop will help the system to behave coherently. The next chapter, dealing with the second level of loops, will offer a robust solution to the second latch problem.

The storing and memory functions, typical for the first order systems, are not true computational features. We will see that they are only useful ingredients allowing to make digital computational systems efficient.

7.6 Problems

Stable/unstable loops

Problem 7.1 Simulate in Verilog the unstable circuit described in Example 3.1. Use 2 unit time (#2) delay for each circuit and measure the frequency of the output signal.

Problem 7.2 Draw the circuits described by the following expressions and analyze their stability taking into account all the possible combinations applied on their inputs:

\[ d = b(ad)' + c \]

\[ d = (b(ad)' + c)' \]

\[ c = (ac' + bc)' \]

\[ c = (a \oplus c) \oplus b. \]

Simple latches

Problem 7.3 Illustrate the second latch problem with a Verilog simulation. Use also versions of the elementary latch with the two gates having distinct propagation times.

Problem 7.4 Design and simulate an elementary clocked latch using a NOR latch as elementary latch.

Problem 7.5 Let be the circuit from Figure 7.21. Indicate the functionality and explain it.
Hint: emphasize the structure of an elementary multiplexer.

Problem 7.6 Explain how it works and find an application for the circuit represented in Figure 7.22.
Hint: Imagine the tristate drivers are parts of two big multiplexors.

Master-slave flip-flops

Problem 7.7 Design an asynchronously presetable master-slave flip-flop.
Hint: to the slave latch must be added asynchronous set and reset inputs (S’ and R’ in the NAND latch version, or S and R in the NOR latch version).

Problem 7.8 Design and simulate in Verilog a positive edge triggered master-slave structure.
Problem 7.9 Design a positive edge triggered master slave structure without the clock inverter.
*Hint:* use an appropriate combination of latches, one transparent on the low level of the clock and another transparent on the high level of the clock.

Problem 7.10 Design the simulation environment for illustrating the master-slave principle with emphasis on the set-up time and the hold time.

Problem 7.11 Let be the circuit from Figure 7.23. Indicate the functionality and explain it. Modify the circuit to be triggered by the other edge of the clock.
*Hint:* emphasize the structures of two clocked latches and explain how they interact.
Problem 7.12 Let be the circuit from Figure 7.24. Indicate the functionality and explain it. Assign a name for the questioned input. What happens if the NANDs are substituted with NORs. Rename the questioned input. Combine both functionality designing a more complex structure. 

**Hint:** go back to Figure 2.6c.

---

**Enabled circuits**

Problem 7.13 An n-bit latch stores the n-bit value applied on its inputs. It is transparent on the low level of the clock. Design an enabled n-bit latch which stores only in the clock cycle in which the enable input, \( en \), take the value 1 synchronized with the positive edge of the clock. Define the set-up time and the hold time related to the appropriate clock edge for data input and for the enable signal.

Problem 7.14 Provide a recursive Verilog description for an n-bit enabled latch.

**RAMs**

Problem 7.15 Explain the reason for \( t_{ASU} \) and for \( t_{AH} \) in terms of the combinational hazard.

Problem 7.16 Explain the reason for \( t_{DSU} \) and for \( t_{DH} \).

Problem 7.17 Provide a structural description of the RAM circuit represented in Figure 7.4 for \( m = 256 \). Compute the size of the circuit emphasizing both the weight of storing circuits and the weight of the access circuits.

Problem 7.18 Design a 256-bit RAM using a two-dimensional array of \( 16 \times 16 \) latches in order to balance the weight of the storing circuits with the weight of the accessing circuits.

Problem 7.19 Design the flow-through version of SRAM defined in Figure 7.8. 
**Hint:** use additional storage circuits for address and input data, and relate the \( WE' \) signal with the clock signal.

Problem 7.20 Design the register to latch version of SRAM defined in Figure 7.25. 
**Hint:** the write process is identical with the flow-through version.
7.6. PROBLEMS

Problem 7.21 Design the pipeline version of SRAM defined in Figure 7.25.
Hint: only the output storage device must be adapted.

Registers

Problem 7.22 Provide a recursive description of an n-bit register. Prove that the (algorithmic) complexity of the concept of register is in $O(n)$ and the complexity of a certain register is in $O(\log n)$.

Problem 7.23 Draw the schematic for an 8-bit enabled and resetable register. Provide the Verilog environment for testing the resulting circuit. Main restriction: the clock signal must be applied only directly to each D flip-flop.
Hint: an enabled device performs its function only if the enable signal is active; to reset a register means to load it with the value 0.

Problem 7.24 Add to the register designed in the previous problem the following feature: the content of the register is shifted one binary position right (the content is divided by two neglecting the reminder) and on most significant bit (MSB) position is loaded the value of the one input bit called SI (serial input). The resulting circuit will be commanded with a 2-bit code having the following meanings:

- **nop**: the content of the register remains unchanged (the circuit is disabled)
- **reset**: the content of the register becomes zero
- **load**: the register takes the value applied on its data inputs
- **shift**: the content of the register is shifted.

Problem 7.25 Design a serial-parallel register which shifts 16 16-bit numbers.
Definition 7.3 The serial-parallel register, $SPR_{n \times m}$, is made by a $SPR_{(n-1) \times m}$ serial connected with a $R_m$. The $SPR_{1 \times m}$ is $R_m$. \( \diamond \)

**Hint:** the serial-parallel register, $SPR_{n \times m}$ can be seen in two manners. $SPR_{n \times m}$ consists in $m$ parallel connected serial registers $SR_n$, or $SPR_{n \times m}$ consists in $n$ serially connected registers $R_m$. We prefer usually the second approach. In Figure 7.26 is shown the serial-parallel $SPR_{n \times m}$.

![Figure 7.26: The serial-parallel register. a. The structure. b. The logic symbol.](image)

**Problem 7.26** Let be $t_{SU}$, $t_{H}$, $t_p$, for a register and $t_{pCLC}$ the propagation time associated with the CLC loop connected with the register. The maximal and minimal value of each is provided. Write the relations governing these time intervals which must be fulfilled for a proper functioning of the loop.

**Pipeline systems**

**Problem 7.27** Explain what is wrong in the following `always` construct used to describe a pipelined system.

```verilog
module pipeline #(parameter n = 8, m = 16, p = 20)
(output reg[m-1:0] output_reg,
 input wire[n-1:0] in,
 clock);

reg[n-1:0] input_reg;
reg[p-1:0] pipeline_reg;
wire[p-1:0] out1;
wire[m-1:0] out2;
clc1 first_clc(out1, input_reg);
clc2 second_clc(out2, pipeline_reg);

always @(posedge clock) begin
  input_reg = in;
  pipeline_reg = out1;
  output_reg = out2;
end
endmodule
module clc1(out1, in1);
  // ...
endmodule
module clc2(out2, in2);
```
7.7. PROJECTS

// ...
endmodule

Hint: revisit the explanation about blocking and nonblocking evaluation in Verilog.

Register file

Problem 7.28 Draw register file 16_4 at the level of registers, multiplexors and decoders.

Problem 7.29 Evaluate for register file 32_5 minimum input arrival time before clock (t_{in,reg}), minimum period of clock (T_{min}), maximum combinational path delay (t_{in,\text{out}}) and maximum output required time after clock (t_{reg,\text{out}}) using circuit timing from Appendix Standard cell libraries.

CAMs

Problem 7.30 Design a CAM with binary coddled output address, which provides as output address the first location containing the searched binary configuration, if any.

Problem 7.31 Design an associative memory, AM, implemented as a maskable and readable CAM. A CAM is maskable if any of the m input bits can be masked using an m-bit mask word. The masked bit is ignored during the comparison process. A CAM is readable if the full content of the first matched location in sent to the data output.

Problem 7.32 Find examples for the inequality

\[
\text{number of possible names} >> \text{number of actual names}
\]

which justify the use of the associative memory concept in digital systems.

7.7 Projects

Project 7.1 Let be the module system containing system1 and system2 interconnected through the two-direction memory buffer module bufferMemory. The signal mode controls the sense of the transfer: for mode = 0 system1 is in read mode and system2 in write mode, while for mode = 1 system2 is in read mode and system1 in write mode. The module library provide the memory block described by the module memory.

module system( input [m-1:0] in1 ,
input [n-1:0] in2 ,
output [p-1:0] out1 ,
output [q-1:0] out2 ,
input clock );
wire [63:0] memOut1 ;
wire [63:0] memIn1 ;
wire [13:0] addr1 ;
wire we1 ;
wire [255:0] memOut2 ;
wire [255:0] memIn2 ;
wire [11:0] addr2 ;
wire we2 ;
Design the module bufferMemory.

**Project 7.2** Design a systolic system for multiplying a band matrix of maximum width 16 with a vector. The operands are stored in serial registers.
Chapter 8

AUTOMATA:
Second order, 2-loop digital systems

In the previous chapter
the memory circuit were described discussing about

- how is built an elementary memory cell
- how applying all type of compositions the basic memory structures (flip-flops, registers, RAMs) can be obtained
- how the basic memory structures are in used real applications

In this chapter
the second order, two-loop circuits are presented with emphasis on

- defining what is an automaton
- the smallest 2-state automata, such as T flip-flop and JK flip-flop
- big and simple automata exemplified by the binary counters
- small and complex finite automata exemplified by the control automata

In the next chapter
the third order, three-loop systems are described taking into account the type of system through which the third loop is closed:

- combinational circuit - resulting optimized design procedures for automata
- memory systems - supposing simplified control
- automata - with the processor as typical structure.
The Tao of heaven is impartial.
If you perpetuate it, it perpetuates you.

Lao Tzu

Perpetuating the inner behavior is the magic of the second loop.

The next step in building digital systems is to add a new loop over systems containing 1-OS. This new loop must be introduced carefully so as the system remains stable and controllable. One of the most reliable ways is to build synchronous structures, that means to close the loop through a way containing a register. The non-transparency of registers allows us to separate with great accuracy the current state of the machine from the next state of the same machine.

This second loop increases the autonomous behavior of the system including it. As we shall see, in 2-OS each system has the autonomy of evolving in the state space, partially independent from the input dynamics, rather than in 1-OS in which the system has only the autonomy of preserving a certain state.

The basic structure in 2-OS is the automaton, a digital system with outputs evolving according to two variables: the input variable and a “hidden” internal variable named the internal state variable, simply the em state. The autonomy is given by the internal effect of the state. The behavior of the circuit output can not be explained only by the evolution of the input, the circuit has an internal autonomous evolution that “memorizes” previous events. Thus the response of the circuit to the actual input takes into account the more or less recent history. The state space is the space of the internal state and its dimension is responsible for the behavioral complexity. Thus, the degree of autonomy depends on the dimension of the state space.

Figure 8.1: The two type of 2-OS. a. The asynchronous automata with a hazardous loop over a transparent latch. b. The synchronous automata with a edge clock controlled loop closed over a non-transparent register.

An automaton is built closing a loop over a 1-OS represented by a collection of latches. The loop can be structured using the previous two type of systems. Thus, there are two type of automata:

---

1Quote from Tao Te King of Lao Tzu translated by Brian Browne Walker.
• *asynchronous automata*, for which the loop is closed over *unclocked latches*, through combina-
tional circuit and/or *unclocked* latches as in Figure 8.1a

• *synchronous automata*, having the loop closed through an 1-OS and all latches are clocked latches
connected on the loop in master-slave configurations (see Figure 8.1b).

Our approach will be focused on the synchronous automata, after considering only in the first subchapter
an asynchronous automaton used to optimize the internal structure of the widely used flip-flop: DFF.

8.1 Basic definitions in automata theory

**Definition 8.1** An automaton, $A$, is defined by the following 5-uple:

$$A = (X, Y, Q, f, g)$$

where:

$X$ : the finite set of input variables

$Y$ : the finite set of output variables

$Q$ : the set of state variables

$f$ : the state transition function, described by $f : X \times Q \rightarrow Q$

$g$ : the output transition function, with one of the following definitions:

- $g : X \times Q \rightarrow Y$ for Mealy type automaton
- $g : Q \rightarrow Y$ for Moore type automaton
- $g(q) = q$ for $Y \equiv Q$, where $q \in Q$ for half-automaton, symbolized with $A_{1/2}$.

At each clock cycle the state of the automaton switches and the output takes the value according to the
new state (and the current input, in Mealy’s approach).

**Definition 8.2** A finite automaton, $FA$, is an automaton with $Q$ a finite set.

$FA$ is a complex circuit because the size of its definition depends by $|Q|$.

**Definition 8.3** A recursively defined $n$-state automaton, $n$-SA, is an automaton with $|Q| \in O(f(n))$.

An $n$-SA has a finite (usually short) definition depending by one or many parameters. Its size will
depend by parameters. Therefore, it is a simple circuit.

**Definition 8.4** An initial state is a state having no predecessor state.

**Definition 8.5** An initial automaton is an automaton having a set of initial states, $Q'$, which is a subset
of $Q$, $Q' \subset Q$.

**Definition 8.6** A strict initial automaton is an automaton having only one initial state, $Q' = \{q_0\}$. 
A strict initial automaton is defined by:

\[ A = (X, Y, Q, f, g; q_0) \]

and has a special input, called reset, used to led the automaton in the initial state \( q_0 \). If the automaton is initial only, the input reset switches the automaton in one, specially selected, initial state.

**Definition 8.7** The delayed (Mealy or Moore) automaton is an automaton with the output values generated through a (delay) register, thus the current output value corresponds to the previous internal state of the automaton, instead of the current value of the state, as in non-delayed version.

The half automaton is an automaton with identity function as the output function (see Figure 8.2a,b) defined for two reasons:

- many optimization techniques are related only with the loop circuits of the automaton. The main feature of an automaton is the autonomy and the associated half-automaton, concept which describes especially this type of behavior
- there are applications that use directly the state as outputs.

All kind of automata can be described starting from a half-automaton, adding only combinational (no loops) circuits and/or memory (one loop) circuits. In Figure 8.2 are presented all the four types of automata:

**Mealy automaton**: results connecting to the “output” of an \( A_{1/2} \) the output CLC that receives also the input \( X \) (Figure 8.2c) and computes the output function \( g \); a combinational way occurs between the input and the output of this automaton allowing a fast response, in the same clock cycle, to the input variation

**Moore automaton**: results connecting to the “output” of an \( A_{1/2} \) the output CLC (Figure 8.2d) that computes the output function \( g \); this automaton reacts to the input signal in the next clock cycle

**delayed Mealy automaton**: results serially connecting a register, \( R \), to the output of the Mealy automaton (Figure 8.2e); this automaton reacts also to the input signal in the next clock cycle, but the output is hazard free because it is registered

**delayed Moore automaton**: results serially connecting a register, \( R \), to the output of the Moore automaton (Figure 8.2f); this automaton reacts to the input signal with a two clock cycles delay.

Real applications use all the previous type of automata, because they react with different delay to the input change. The registered outputs are preferred if possible.

**Theorem 8.1** The time relation between the input value and the output value is the following for the four types of automata:

1. for Mealy automaton the output to the moment \( t \), \( y(t) \in Y \) depends on the current input value, \( x(t) \in X \), and by the current state, \( q(t) \in Q \), i.e., \( y(t) = g(x(t), q(t)) \)

2. for delayed Mealy automaton and Moore automaton the output corresponds with the input value from the previous clock cycle:
8.1. BASIC DEFINITIONS IN AUTOMATA THEORY

Figure 8.2: Automata types. a. The structure of the half-automaton (A₁/₂), the no-output automaton: the state is generated by the previous state and the previous input. b. The logic symbol of half-automaton. c. Immediate Mealy automaton: the output is generated by the current state and the current input. d. Immediate Moore automaton: the output is generated by the current state. e. Delayed Mealy automaton: the output is generated by the previous state and the previous input. f. Delayed Moore automaton: the output is generated by the previous state.

- \( y(t) = g(x(t-1),q(t-1)) \) for Mealy delayed automaton
- \( y(t) = g(q(t)) = g(f(x(t-1),q(t-1)) \) for Moore automaton

3. for delayed Moore automaton the input transition acts on the output transition delayed with two clock cycles:

\[ y(t) = g(q(t-1)) = g(f(x(t-2),q(t-2)) \]

Proof The proof is evident starting from the previous two definitions. ◇

The possibility emphasized by this theorem is that we dispose of automata with different time reaction to the input variations. The Mealy automaton follows immediate the input transitions, delayed Mealy and Moore automata react with one clock cycle delay to the input transitions and delayed Moore automaton delays with two cycles the response to the input.
The symbols from the sets \( X, Y, \) and \( Q \) are binary coded using bits specified by \( X_0, X_1, \ldots \) for \( X, Y_0, Y_1, \ldots \) for \( Y, Q_0, Q_1, \ldots \) for \( Q \).

Actually, all implementable automata are finite. Traditionally, the term *finite automaton* is used to distinguish a subset of automata whose behavior is described using a constant number of states. Even if the input string is *infinite*, the behavior of the automaton is limited to a trajectory traversing a constant (*finite*) number of states. A finite automaton will be an automaton having a random combinational function for its transition functions \( f \) and \( g \). Therefore, a finite automaton is a complex structure.

A “non-finite” automaton that is an automaton designed to evolve in a state space proportional with the length of the input string. Now, if the input string is *infinite* the number of states must be also *infinite*. Such an automaton can be defined only if its transition function is simple. Its combinational loop is a simple circuit even if it can be a big one. The “non-finite” automaton has a number of states that does not affect the definition (see the following examples of counters, for sum prefix automaton, ...). We classify the automata in two categories:

- “non-finite”, recursive defined, simple automata, called *functional automata*, or simply *automata*
- non-recursive defined, complex automata, called *finite automata*.

We continue this chapter with an example of asynchronous circuit, because of its utility and because we intend to show how complex is the management of its behavior. We will continue presenting only synchronous automata, starting with small automata having only two states (the smallest state space). We will continue with simple, recursive defined automata and we will end with finite automata, that are the most *complex* automata.

### 8.2 Two States Automata

The smallest two-state half-automata can be explored almost systematically. Indeed, there are only 16 one-input two-state half-automata and 256 with two inputs. We choose only two of them: the *T flip-flop*, the *JK flip-flop*, which are automata with \( Q = Y \) and \( f = g \). For simple 2-operand computations 2-input automata can be used. One of them is the *adder automaton*. This section ends with a small and simple universal automaton having 2 inputs and 2 states.

#### 8.2.1 Optimizing DFF with an asynchronous automaton

The very important feature added by the master-slave configuration – that of edge triggering the flip-flop – was paid by increasing two times the size of the structure. An improvement is possible for DFF (the master-slave D flip-flop) using the structure presented in Figure 8.3, where instead of 8 2-input NANDs and 2 invertors only 6 2-input gates are used. The circuit contains three elementary unclocked latches: the output latch, with the inputs \( R' \) and \( S' \) commanded by the outputs of the other two latches, \( L1 \) and \( L2 \). \( L1 \) and \( L2 \) are loop connected building up a very simple asynchronous automaton with two inputs – \( D \) and \( CK \) – and two outputs – \( R' \) and \( S' \).

The explanation of how this DFF, designed as a 2-OS, works uses the static values on the inputs of the latches. For describing the process of switching in 1 the triplets \( (x,y,z) \) are used, while for switching in 0 are used \( [x,y,z] \), where:

- \( x \): is the stable value in the *set-up time interval* (in a time interval, equal with \( t_{su} \), before the positive transition of \( CK \))
8.2. TWO STATES AUTOMATA

\( y \) : is the stable value in the **hold time interval** (in a time interval of \( t_h \), after the positive transition of CK; the transition time, \( t_+ \), is considered very small and is neglected)

\( z \) : is a possible value **after the hold time interval** (after \( t_h \) measured from the positive transition of CK)

For the process of transition in 1 we follow the triplets \((x,y,z)\) in Figure 8.3:

**in set-up time interval** : CK = 0 forces the values \( R' \) and \( S' \) to 1, does not matter what is the value on D. Thus, the output latch receives passive values on both of its inputs.

**in hold time interval** : CK = 1 frees L1 and L2 to follow the signals they receive on their inputs. The first order and the second order loops are now closed. L2 switches to \( S' = 0 \), because of the 0 received from L1, which maintains its state because D and CK have passive values and the output \( S' \) of L2 reinforces its state to \( R' = 1 \). The output latch is then set because of \( S' = 0 \).

**after the hold time interval** : the possible transition in 0 of D after the hold time does not affect the output of the circuit, because the second loop, from L2 to L1, forces the output \( R' \) to 1, while L2 is not affected by the transition of its input to the passive value because of \( D = 0 \). Now, the second loop allow the system to “ignore” the switch of D after the hold time.

![Figure 8.3: The D flip-flop implemented as a 2-OS system.](image)

The asynchronous automaton built up loop connecting two unclocked latches allows to trigger the output latch according to the input data value available at the positive transition of clock.

For the process of transition in 0 we follow the triplets \([x,y,z]\) in Figure 8.3:

**in set-up time interval** : CK = 0 forces the values \( R' \) and \( S' \) to 1, does not matter what is the value on D. Thus, the output latch receives passive values on both of its inputs. The output of L1 applied to L2 is also forced to 1, because of the input \( D = 0 \).
in hold time interval: CK = 1 frees L1 and L2 to follow the signals they receive on their inputs. The first order and the second order loops are now closed. L1 switches to R' = 0, because of the 0 maintained on D. L2 does not change its state because the input received from L1 has the passive value and the CK input switches also in the passive value. The output latch is then reset because of R' = 0.

after the hold time interval: the possible transition in 1 of D after the hold time does not affect the state of the circuit, because 1 is a passive value for a NAND elementary latch.

The effect of the second order loop is to “inform” the circuit that the set signal was, and still is, activated by the positive transition of CK and any possible transition on the input D must be ignored. The asynchronous automaton L1 & L2 behaves as an autonomous agent who “knows” what to do in the critical situation when the input D takes an active value in an unappropriate time interval.

8.2.2 The Smallest Automaton: the T Flip-Flop

The size and the complexity of an automaton depends at least on the dimension of the sets defining it. Thus, the smallest (and also the simplest) automaton has two states, Q = {0, 1} (represented with one bit), one-bit input, T = {0, 1}, and Q = Y. The associated structure is represented in Figure 8.4, where is represented a circuit with one-bit input, T, having a one-bit register, a D flip-flop, for storing the 1-bit coded state, and a combinational logic circuit, CLC, for computing the function f.

What can be the meaning of an one-bit “message”, received on the input T, by a machine having only two states? We can “express” with the two values of T only the following things:

no op: T = 0 - the state of the automaton remains the same

switch: T = 1 - the state of the automaton switches.

![Figure 8.4: The T flip-flop.](image)

The resulting automaton is the well known T flip-flop. The actual structure of a T flip-flop is obtained connecting on the loop a commanded inverter, i.e., a XOR gate (see Figure 8.4b). The command input is T and the value to be inverted is Q, the state and the output of the circuit.

This small and simple circuit can be seen as a 2-modulo counter because for T = 1 the output “says”: 01010101... Another interpretation of this circuit is: the T flip-flop is a frequency divider. Indeed, if the
8.2. TWO STATES AUTOMATA

clock frequency is \( f_{\text{CK}} \), then the frequency of the signal received to the output \( Q \) is \( f_{\text{CK}}/2 \) (after each clock cycle the circuit comes back in the same state).

8.2.3 The JK Automaton: the Greatest Flip-Flop

The “next” automaton in an imaginary hierarchy is one having two inputs. Let’s call them \( J \) and \( K \). Thus, we can define the famous \( JK \) flip-flop. Also, the function of this automaton results univocally. For an automaton having only two states the four input messages coded with \( J \) and \( K \) will be compulsory:

**no op** : \( J = K = 0 \) - the flip-flop output does not change (the same as \( T = 0 \) for \( T \) flip-flop)

**reset** : \( J = 0, K = 1 \) - the flip-flop output takes the value 0 (specific for \( D \) flip-flop)

**set** : \( J = 1, K = 0 \) - the flip-flop output takes the value 1 (specific for \( D \) flip-flop)

**switch** : \( J = K = 1 \) - the flip-flop output switches in the complementary state (the same as \( T = 1 \) for \( T \) flip-flop)

Only for the last function the loop acts specific for a second order circuit. The flip-flop must “tell to itself” what is its own state in order “to knows” how to switch in the other state. Executing this command the circuit asserts its own autonomy. The vagueness of the command “switch” imposes a sort of autonomy to determine a precise behavior. The loop that assures this needed autonomy is closed through two AND gates (see Figure 8.5a).

![Figure 8.5: The JK flip-flop. It is the simplest two-input automaton. a. The structure: the loop is closed over a master-slave RSF-F using only two AND gates. b. The logic symbol.](image)

Finally, we solved the second latch problem. We have a two state machine with two command inputs and for each input configuration the circuit has a predictable behavior. The JK flip-flop is the best flip-flop ever defined. All the previous ones can be reduced to this circuit with minimal modifications (\( J = K = T \) for \( T \) flip-flop or \( K' = J = D \) for \( D \) flip-flop).

8.2.4 Serial Arithmetic

As we know the ripple carry adder has the size in \( O(n) \) and the depth also in \( O(n) \) (remember Figure 6.18). If we agree with the time in this magnitude order, then there is a better solution where a second order circuit is used.

The best solution for the \( n \)-bit adder is a solution involving a small and simple automaton. Instead of storing the two numbers to be added in (parallel) registers, as in the pure combinational solution, the sequential solutions
needs serial registers for storing the operands. The system is presented in Figure 8.6, containing three serial registers (two for the operands and one for the result) and the adder automaton.

The adder automaton is a two states automaton having in the loop the carry circuit of a full adder (FA). The one-bit state register contains the carry bit from the previous cycle. The inputs A and B of FA receive synchronously, at each clock cycle, bits having the same binary range from the serial registers. First, LSBs are read from the serial registers. Initially, the automaton is in the state 0, that means CR = 0. The output S is stored bit by bit in the third serial register during n clock cycles. The final (n + 1)-bit result is contained in the output serial register and in the state register.

The operation time remains in the order of $O(n)$, but the structure involved in computation becomes the constant structure of the adder automaton. The product of the size, $S_{ADD}(n)$, into the time, $T_{ADD}(n)$ is in $O(n)$ for this sequential solution. Again, Conjecture 2.1 acts emphasizing the slowest solution as optimal. Let us remember that for a carry-look-ahead adder, the fastest $O(1)$ variant, the same product was in $O(n^3)$. The price for the constant execution time is, in this example, in $O(n^2)$. I believe it is too much. We will prefer architectural solutions which allow us to avoid the structural necessity to perform the addition in constant time.

### 8.2.5 Hillis Cell: the Universal 2-Input, 1-Output and 2-State Automaton

Any binary (two-operando) simple operation on n-bit operands can be performed serially using a 2-state automaton. The internal state of the automaton stores the “carry” information from one stage of processing to another. In the adder automaton, just presented, the internal state is used to store the carry bit generated adding the $i$-th bits of a number. It is used in the next stage for adding the $(i+1)$-th bits. This mechanism can be generalized, resulting an universal 2-input (for binary operation), one-output and 2-state (for “carry” bit) automaton.

**Definition 8.8** An Universal 2-input ($in_1$, $in_2$),one-output, 2-state (codded by state[0]) automaton is a programmable structure using a 16-bit program word, \{next\_state\_func[7:0], out\_func[7:0]\}. It is defined by the following Verilog code:

```verilog
module univAut( output out , // output
                input in1, in2 , // operands
                input [7:0] nextStateFunc, outFunc , // loop program, output program
                input reset, clock );

reg state;
assign out = outFunc[{state, in2, in1}];
always @(posedge clock) state <= reset ? 1'b0 : nextStateFunc[{state, in2, in1}];
```

Figure 8.6: **Serial $n$-bit adder.** The state of the adder automaton has the value of the carry generated adding the previous 2 bits received from the output of the two serial registers containing the operands.
8.3 Functional Automata: the Simple Automata

The smallest automata before presented are used in recursively extended configuration to perform similar functions for any \( n \). From this category of circuits we will present in this section only the binary counters. The next circuit will be also a simple one, having the definition independent by size. It is a sum-prefix automaton. The last subject will be a multiply-accumulate circuit built with two simple automata serially connected.

8.3.1 Counters

The first simple automaton is a composition starting from one of the function of T flip-flop: the counting. If one T flip-flop counts modulo-\( 2^1 \), maybe two T flip-flops will count modulo-\( 2^2 \) and so on. Seems to be right, but we must find the way for connecting many T flip-flops to perform the counter function.

For the synchronous counter\(^2\) built with \( n \) T flip-flops, \( T_{n-1}, \ldots, T_0 \), the formal rule is very simple: if \( \text{INC}_0 \), then the first flip-flop, \( T_0 \), switches, and the \( i \)-th flip-flop, for \( i = 1, \ldots, n-1 \), switches only if all the previous flip-flops are in the state 1. Therefore, in order to detect the switch condition for \( i \)-th flip-flop an \( \text{AND}_{i+1} \) must be used.

**Definition 8.9** The \( n \)-bit synchronous counter, \( \text{COUNT}_n \), has a clock input, \( CK \), a command input, \( \text{INC}_0 \), an \( n \)-bit data output, \( Q_{n-1}, \ldots, Q_0 \), and an expansion output, \( \text{INC}_n \). If \( \text{INC}_0 = 1 \), the active edge of clock increments the value on the data output (see Figure 8.8).

There is also a recursive, constructive, definition for \( \text{COUNT}_n \).

\(^2\)There exist also asynchronous counters. They are simpler but less performant.
Definition 8.10 An $n$-bit synchronous counter, $COUNT_n$ is made by expanding a $COUNT_{n-1}$ with a T flip-flop with the output $Q_{n-1}$, and an AND$_{n+1}$, with the inputs $INC_0$, $Q_{n-1}, \ldots, Q_0$, which computes $INC_n$ (see Figure 8.8). $COUNT_1$ is a T flip-flop and an AND$_2$ with the inputs $Q_0$ and $INC_0$ which generates $INC_1$. ♦

![Figure 8.8: The synchronous counter.](image)

The recursive definition of a synchronous counter has $S_{COUNT}(n) \in O(n^2)$ and $T_{COUNT}(n) \in O(\log n)$, because for the $i$-th range one TF-F and one AND$_i$ are added.

Example 8.1 *The Verilog description of a synchronous counter follows:

```verilog
module sync_counter #(parameter n = 8)(output [n-1:0] out ,
    output inc_n ,
    input inc_0 ,
    reset ,
    clock);

    t_reg t_reg( .out (out) ,
        .in (prefix_out[n-1:0]) ,
        .reset (reset) ,
        .clock (clock) );

    and_prefix and_prefix( .out (prefix_out) ,
        .in ({out, inc_0}) );

    assign inc_n = prefix_out[n];
endmodule

module t_reg #(parameter n = 8)( output reg [n-1:0] out ,
    input [n-1:0] in ,
    input reset ,
    clock );

    always @(posedge clock) if (reset) out <= 0;
    else out <= out ^ in;
endmodule
```

The reset input is added because it is used in real applications. Also, a reset input is good in simulation because makes the simulation possible allowing an initial value for the flip-flops (reg[n-1:0] out in module t_reg) used in design. ♦
It is obvious that $C_{\text{COUNT}}(n) \in O(1)$ because the definition for any $n$ has the same, constant size (in number of symbols used to write the Verilog description for it or in the area occupied by the drawing of $\text{COUNT}_n$). The size of $\text{COUNT}_n$, according to the Definition 4.4, can be computed starting from the following iterative form:

$$S_{\text{COUNT}}(n) = S_{\text{COUNT}}(n - 1) + (n + 1) + S_T$$

and results:

$$S_{\text{COUNT}}(n) \in O(n^2)$$

because of the AND gates network used to command the T flip-flop. The counting time is the clock period. The minimal clock period is limited by the propagation time inside the structure. It is computed as follows:

$$T_{\text{COUNT}}(n) = t_{\text{propT}} + t_{\text{propAND}_n} + t_{\text{SU}} \in O(\log n)$$

where: $t_{\text{propT}} \in O(1)$ is the propagation time through the T flip-flop, $t_{\text{propAND}_n} \in O(\log n)$ is the propagation time through the AND$_n$ (in the fastest version it is implemented using a tree of AND$_2$ gates) gate and $t_{\text{SU}} \in O(1)$ is the set-up time at the input of T flip-flop.

In order to reduce the size of the counter we must find another way to solve the function performed by the network of ANDs. Obviously, the network of ANDs is an AND prefix-network. Thus, the problem could be reduced to the problem of the general form of prefix-network. The optimal solution exists and has the size in $O(n)$ and the time in $O(\log n)$ (see in this respect the section 8.2).

Finishing this short discussion about counters must be emphasized the autonomy of this circuit which consists in switching in the next state according to the current state. We “tell” simply to the circuit “please count”, and the circuit know what to do. The loop allow “him to know” how to behave.

Real applications uses more complex counters able to be initialized in any states or the count in both ways, up and down. Such a counter is described by the following code:

```verilog
module full_counter #(parameter n = 4)(output reg [n-1:0] out ,
                                   input [n-1:0] in ,
                                   input reset ,
                                   load ,
                                   down ,
                                   count ,
                                   clock );

always @(posedge clock)
  if (reset) out <= 0 ;
  else if (load) out <= in ;
  else if (count) if (down) out <= out - 1 ;
  else if (count) if (down) out <= out + 1 ;
  else out <= out ;
endmodule
```

The reset operation has the highest priority, and the counting operations have the lowest priority.

### 8.3.2 Linear Feedback Shift Registers

Linear feedback shift registers (LFSR) provide a simple way for generating non-sequential lists of numbers which behaves as a random sequence of numbers. For this reason LFSR are called also pseudo-random number generators. Thus, generating a sequence of pseudo-random numbers only requires a
shift register and a number of XORs. The mathematical description of these circuits is based on the Galois Field theory\(^3\).

The structure is very simple: a \(n\)-bit left shift serial register whose serial input is fed with the output of a XOR circuit. The inputs of the XOR circuit are selected from the \(n\) outputs of the register. A loop of 2-input XORs is closed as the second loop. The LFSR is, thus, a simple automaton.

**Example 8.2** Let’s consider a 4-bit left shift register, \(s\text{Reg}[3:0]\), and the loop closed through a 2-input XOR in two versions:

- *from* \(s\text{Reg}[3] and s\text{Reg}[0]\) (see Figure 8.9a)
- *from* \(s\text{Reg}[3] and s\text{Reg}[1]\) (see Figure 8.9b)

Let’s consider the initial state in both cases: \(s\text{Reg} = 4\text{'b}0001\). The circuit from Figure 8.9a generates the following periodic sequence starting from the initial state:

\[
\begin{align*}
0001 \\
0011 \\
0111 \\
1111 \\
1110 \\
1101 \\
1010 \\
0101 \\
1011 \\
0110 \\
1100 \\
1001 \\
0010 \\
0100 \\
1000 \\
0001 \\
\ldots
\end{align*}
\]

while the circuit from Figure 8.9b generate a shorter periodic sequence, as follow:

\[
\begin{align*}
0001 \\
0010 \\
0101 \\
1010 \\
0100 \\
1000 \\
0001 \\
\ldots
\end{align*}
\]

**Important note**: the initial state \(s\text{Reg} = 4\text{'b}0000\) must be avoided, because from this state there is no evolution.

\(^3\)See a tutorial at: http://homepages.cae.wisc.edu/~ece553/handouts/LFSR-notes.PDF
8.3. FUNCTIONAL AUTOMATA: THE SIMPLE AUTOMATA

The LFSR of interest are mainly those who generate the longest periodic sequence. Because form 00...0 there is no evolution, the longest sequence generated by a $n$-bit LFSR is of $2^n - 1$ numbers. The length of the period depends of the loop. More specific, it is about what outputs of the register are XORed. If the XORs considered have at least 2 inputs, then there are $2^n - (n + 1)$ version of LFSRs. They are specified by the binary sequence used to select the outputs. For example: if the register is $sReg[7:0]$ and the loop is selected by A2, then it corresponds to the loop having the logic function:


the 3 inputs to the 3-input XOR being selected by the 1s of the selection code A2 = 1010_0010. For $n = 8$ the following selection codes: 8E 95 96 A6 AF B1 B2 B4 B8 C3 C6 D4 E1 E7 F3 FA correspond to the LFSRs with cycles of 255 numbers, which are maximal\(^4\). Therefore, LFSR(C3) stands for the LFSR with the loop characterized by the 8-bit number C3.

Experiments with LFSRs suppose:

- to initialize the register is a certain state
- to select the loop configuration, i.e., select the output of the register to be XORed to the serial input

The following circuit can be used to simulate the 8-bit LFSRs:

```vhdl
BEGIN
  PROCESS (rst)
  BEGIN
    IF rst = '1' THEN
      sReg <= (OTHERS => '0');
    END IF;
  END PROCESS;

  PROCESS (prog)
  BEGIN
    sReg <= (1111, 0000) WHEN prog = '0' OR prog = '1' OR prog = '2' OR prog = '3' OR prog = '4' OR prog = '5' OR prog = '6' OR prog = '7' OR prog = '8' OR prog = '9' OR prog = 'A' OR prog = 'B' OR prog = 'C' OR prog = 'D' OR prog = 'E' OR prog = 'F' ELSE
      sReg <= sReg XOR (sReg[7] \&\& sReg[5] \&\& sReg[1]);
    END IF;
  END PROCESS;

END;
```

File: progLFSR.v

Programmable Linear Feedback Shift Register

Used as pseudo-random numbers generator.

The input `prog` is used for:

- set the initial state of the register when `rst = 1`; it must be different from 0000_0000
- "programming", i.e., select the outputs to be XORed to the input of the register; when `rst = 0

The 16 "programs" for the longest cycle (sequence of 255 8-bit numbers):

8E 95 96 A6 AF B1 B2 B4 B8 C3 C6 D4 E1 E7 F3 FA

\(^4\)https://users.ece.cmu.edu/~koopman/lfsr/index.html
CHAPTER 8. AUTOMATA: SECOND ORDER, 2-LOOP DIGITAL SYSTEMS

module progLFSR(output reg [7:0] out ,
    input [7:0] prog ,
    input rst ,
    input clk);

    always @(posedge clk) if (rst) out <= prog ;
    else out <= {out[6:0], ^(out & prog)};
endmodule

In order to use LFSR(AF) initialized at 0000_0011, during at least one clock cycle apply prog = 8’b0000_0011 with rst = 1, then switch to rst = 0 with prog = 8’b1010_1111.

8.3.3 RALU: Registers with Arithmetic-Logic Unit

For very big sized state space the associated combinational circuits used to compute the next state and the output become too big to be efficiently implemented. Therefore, a possible solution is to structure the state so as in each cycle only a part of the state will be affected by the transition. Thus the the time for provide a transition of the entire state will increase linearly, but the size of the circuits associated to the functions \( f \) and \( g \) will decrease exponentially.

Structured State Space Automaton\((S^3A)\)

**Definition 8.11**  The function:

\[
P(i, n, x_0, x_1, \ldots, x_{n-1}) = x_i
\]

is the projection (selection) function which returns the \( i \)-th element from a set of \( n \) elements.

\( \Diamond \)

**Definition 8.12**  A Structured State Space Automaton\((S^3A)\) is defined by the following quintuple:

\[
S^3A = (X \times A, Y, Q, f, g)
\]

where:

- \( Q = (Q_0 \times Q_1 \times \ldots \times Q_{s-1}) \) : is the structured state space described as a Cartesian set of elements binary represented on \( m \) bits
- \( X \) : the finite set of inputs binary represented on \( p \) bits
- \( A \) : the finite set of codes used to select an element of the set \( Q \) involved in the current state transition
- \( Y \) : the finite set of outputs
- \( f : (X \times A \times Q) = (X \times P(i,s,Q)) = (X \times Q_i) \to Q_i \) is the state transition function where \( i \in A \)
- \( g : (X \times A \times Q) = (X \times P(i,s,Q)) = (X \times Q_i) \to Y \) is the output transition function where \( i \in A \)

\( \Diamond \)
An $S^3A$ is implemented using a synchronous RAM to store the state. The input $A$ is the address which selects the element of the Cartesian product stored in the $s$ locations of the RAM. The efficiency of this approach could be evaluated as follows. The execution time for a full transition of $S^3A$ is $s$ times bigger than for the equivalent standard automaton, because only one element of the Cartesian product can be computed in one cycle. Therefore the time performance is $1/s$. The size of the combinational circuit for $f$ belongs, in the worst case, to $O(2^{p+m})$, while for the standard automaton it belongs, in the worst case, to $O^{p+s+m}$. Results a decrease in size belonging to $O(2^{m(s-1)})$. The time performance decreases linearly with $s$, while the size decreases exponentially with $s$. There is no room for debate: when possible, the $S^3A$ is the solution.

Multi-port $S^3A$

Because the binary functions dominate the class of arithmetic and logic functions, multi-port $S^3A$s are used in designing the executing core of any processing element. The most frequently used multi-port $S^3A$ is a 3-port $S^3A$. Two ports are used to fetch the operands and the third for selecting the destination of the result. The following definition refers only the the half-automaton, because only the way the loop is closed in important. We can get the output of the system in various ways, depending on the application.

**Definition 8.13** A 3-port Structured State Space Half-Automaton, $S^3HA$ is defined as following:

$$S^3HA = (X \times DA \times LA \times RA, Q, f)$$

where:

- $Q = (Q_0 \times Q_1 \times \ldots \times Q_{s-1})$: is the structured state space described as a Cartesian set of elements binary represented on $m$ bits
- $X$: the finite set of inputs binary represented on $p$ bits
- $DA$: the finite set of codes used to select the element of the set $Q$ to be modified (is the destination of the change) in the current state transition
- $LA$: the finite set of codes used to select the element of the set $Q$ to be used as left operand in the current state transition
- $RA$: the finite set of codes used to select the element of the set $Q$ to be used as right operand in the current state transition
- $f: (X \times LA \times RA \times Q) = (X \times P(i, s, Q) \times P(j, s, Q)) = (X \times Q_i \times Q_j) \rightarrow P(k, s, Q) = Q_k$ is the state transition function where $i \in LA$, $j \in RA$, $k \in DA$

**Example 8.3** Let be a RALU designed with two modules already presented in the previous sections: the ALU exemplified in Example 6.8 and the register file presented in Simulation 7.5. In Figure 8.10 is represented the schematic of a 32-bit RALU.
Figure 8.10: 32-bit RALU.

```vhdl
module RALU(
    output [31:0] left_out,
    output [31:0] right_out,
    output carryOut,
    input load,
    input [3:0] left_addr,
    input [3:0] right_addr,
    input [3:0] dest_addr,
    input [3:0] write_enable,
    input [31:0] in,
    input carryIn,
    input [2:0] func,
    input clock
);

wire [31:0] out;

register_file rf(
    .left_operand (left_out ),
    .right_operand (right_out ),
    .result (out ),
    .left_addr (left_addr ),
    .right_addr (right_addr ),
    .dest_addr (dest_addr ),
    .write_enable (write_enable ),

    .left_out (left_out ),
    .right_out (right_out ),
    .out (out ),
    .left_addr (left_addr ),
    .right_addr (right_addr ),
    .dest_addr (dest_addr ),
    .write_enable (write_enable )
);
```

---

Name: RALU: Register file with Arithmetic and Logic Unit
File: RALU.v

Description: register file with 16 32-bit register and an ALU with 8 generic arithmetic and logic functions.
8.3. FUNCTIONAL AUTOMATA: THE SIMPLE AUTOMATA

8.3.4 * Accumulator Automaton

The accumulator automaton is a generalization of the counter automaton. A counter can add 1 to the value of its state in each clock cycle. An accumulator automaton can add in each clock cycle any value applied on its inputs.

Many applications require the accumulator function performed by a system which adds a string of numbers returning the final sum and all partial results – the prefixes. Let be \( p \) numbers \( x_1, \ldots, x_p \). The sum-prefixes are:

\[
\begin{align*}
y_1 &= x_1 \\
y_2 &= x_1 + x_2 \\
y_3 &= x_1 + x_2 + x_3 \\
&\quad \vdots \\
y_p &= x_1 + x_2 + \ldots + x_p.
\end{align*}
\]

This example of arithmetic automata generates at each clock cycle one prefix starting with \( y_1 \). The initial value in the register \( R_{m+n} \) is zero. The structure is presented in Figure 8.11 and consists in an adder, \( ADD_{m+n} \), two multiplexors and a state register, \( R_{m+n} \). This automaton has \( 2^{m+n} \) states and computes sum prefixes for \( p = 2^m \) numbers, each represented with \( n \) bits. The supplementary \( m \) bits are needed because in the worst case adding two numbers of \( n \) bits results a number of \( n + 1 \) bits, and so on, ... adding \( 2^m \) \( n \)-bit numbers results, in the worst case, a \( n + m \)-bit number. The automaton must be dimensioned such as in the worst case the resulting prefix can be stored in the state register. The two multiplexors are used to initialize the system clearing the register (for \( acc = 0 \) and \( clear = 1 \)), to maintain unchanged the accumulated value (for \( acc = 0 \) and \( clear = 0 \)), or to accumulate the \( n \)-bit input value (for \( acc = 1 \) and \( clear = 0 \)). It is obvious the accumulate function has priority: for \( acc = 1 \) and \( clear = 1 \) the automaton accumulates ignoring the clear command.

The size of the systems depends on the speed of adder and can be found between \( O(m + n) \) (for ripple carry adder) and \( O((m + n)^3) \) (for carry-look-ahead adder).

It is evident that this automaton is a simple one, having a constant sized definition. The four components are all simple recursive defined circuits. This automaton can be built for any number of states using the same definition. In this respect this automaton is a "non-finite", functional automaton.

8.3.5 * Sequential multiplication

Multiplication is performed sequentially by repeated additions and shifts. The \( n \)-bit multiplier is inspected and the multiplicand is accumulated shifted according to the position of the inspected bit or bits. If in each cycle one bit is inspected (radix-2 multiplication), then the multiplication is is performed in \( n \) cycles. If 2 bits are inspected (radix-2 multiplication) in each cycle, then the operation is performed in \( n/2 \) cycles, and so on.

* Radix-2 multiplication

The generic three-register structure for radix-2 multiplication is presented in the following Verilog module.
Figure 8.11: Accumulator automaton. It can be used as sum prefix automaton because in each clock cycle outputs a new value as a result of a sequential addition of a stream of signed integers.

Figure 8.12: Radix-2 sequential multiplier.

```
module rad2mult #(parameter n = 8)(output [2*n-1:0] product , // result output
                   input [n-1:0] op , // operands input
                   input [2:0] com , // command input
                   input clock);
reg [n-1:0] op2, op1, prod ; // multiplicand, multiplier, upper result
wire [n:0] sum ;
assign sum = prod + op2;
always @(posedge clock) if (com[2])
    case(com[1:0])
      2'b00: prod <= 0 ; // clear prod
      2'b01: op1 <= op ; // load multiplier
      2'b10: op2 <= op ; // load multiplicand
      2'b11: {prod, op1} <= (op1[0] == 1) ? {sum, op1[n-1:1]} : {prod, op1} >> 1 ; // multiplication step
    endcase
```
8.3. FUNCTIONAL AUTOMATA: THE SIMPLE AUTOMATA

```verilog
assign product = {prod, op1};
endmodule
```

The sequence of commands applied to the previous module is:

```verilog
initial begin
    com = 3'b100;
#2 com = 3'b101;
    op = 8'b0000_1001;
#2 com = 3'b110;
    op = 8'b0000_1100;
#2 com = 3'b111;
#2 com = 3'b111;
#2 com = 3'b111;
#2 com = 3'b111;
#2 com = 3'b111;
#2 com = 3'b111;
#2 com = 3'b111;
#2 com = 3'b111;
#2 com = 3'b000;
#2 $stop;
end
```

In real application the first three steps can be merged in one, depending on the way the multiplier is connected. The effective number of clock cycles for multiplication is $n$.

* Radix-4 multiplication

The time performance for the sequential multiplication is improved if in each clock cycle 2 bits are considered instead of one. In order to keep simple the operation performed in each cycle a small and simple two-state automaton is included in design.

![Radix-4 sequential multiplier](image)

Figure 8.13: Radix-4 sequential multiplier.

Let us consider positive integer multiplication. The design inspects by turn each 2-bit group of the multiplier, $m[i+1:i]$ for $i = 0, 2, \ldots, n-2$, doing the following simple actions:

- $m[i+1:i] = 00$: adds 0 to the result and multiply by 4 the multiplicand
- $m[i+1:i] = 01$: adds the multiplicand to the result and multiply by 4 the multiplicand
- $m[i+1:i] = 10$: adds twice the multiplicand to the result and multiply by 4 the multiplicand
- $m[i+1:i] = 11$: subtract the multiplicand from the results, multiply by 4 the multiplicand, and sends to the next cycle the information that the current value of multiplicand must be added 4 times to the result
The inter-cycle message is stored in the state of the automaton. In the initial cycle the state of the automaton is ignored, while in the next stages it is added to the value of \( m[i+1] \).

```
module rad4mult #(parameter n = 8)(output [2*n-1:0] product, // result output
  input [n-1:0] op, // operands input
  input [2:0] com, // command input
  input clock);

reg [n-1:0] op2, op1; // multiplicand, mutiplier, upper part of result
reg [n+1:0] prod; // upper part of result
reg state; // the state register of a two-state automaton
reg [n+1:0] nextProd;
reg nextState;
wire [2*n+1:0] next;
/*
com = 3'b000 // nop
com = 3'b001 // clear prod register & initialize automaton
com = 3'b010 // load op1
com = 3'b011 // load op2
com = 3'b101 // mult
com = 3'b110 // lastStep
*/
always @(posedge clock)
case(com)
  3'b001: begin prod <= 0;
      state <= 0;
      end
  3'b010: op1 <= op;
  3'b011: op2 <= op;
  3'b101: begin {prod, op1} <= next;
      state <= nextState;
      end
  3'b110: if (state) prod <= prod + op2;
default prod <= prod;
endcase
assign next = {{2{nextProd[n+1]}}, nextProd, op1[n-1:2]};
// begin algorithm
always @(*) if (state) case(op1[1:0])
  2'b00: begin nextProd = prod + op2;
      nextState = 0;
      end
  2'b01: begin nextProd = prod + (op2 << 1);
      nextState = 0;
      end
  2'b10: begin nextProd = prod - op2;
      nextState = 1;
      end
  2'b11: begin nextProd = prod;
      nextState = 1;
      end
endcase
else case(op1[1:0])
  2'b00: begin nextProd = prod;
      nextState = 0;
      end
```
endcase

// end algorithm
assign product = {prod[n-1:0], op1};
endmodule

The sequence of commands for \( n = 8 \) is:

```verbatim
initial begin
  com = 3’b001 ;
  #2 com = 3’b100 ;
  op = 8’b1100_1111 ;
  #2 com = 3’b011 ;
  op = 8’b1111_1111 ;
  #2 com = 3’b101 ;
  #2 com = 3’b101 ;
  #2 com = 3’b101 ;
  #2 com = 3’b101 ;
  #2 com = 3’b101 ;
  #2 com = 3’b100 ;
  #2 $stop
end
```

The effective number of clock cycles for positive integer radix-4 multiplication is \( n/2 + 1 \).

Let’s now solve the problem multiplying signed integers. The Verilog description of the circuit is:

```verilog
module signedRad4mult #(parameter n = 8)( output [2*n-1:0] product, // result output 
  input [n-1:0] op, // operands input 
  input [2:0] com, // command input 
  input clock );

reg [n-1:0] op1 ; // signed mutiplier
reg [n:0] op2 ; // signed multiplicand
reg [n:0] prod ; // upper part of the signed result
reg state ; // the state register of a two-state automaton

reg [n:0] nextProd ;
reg nextState ;

wire [2*n:0] next ;
/*
 com = 3’b000 // nop
 com = 3’b001 // clear prod register & initialize automaton
 com = 3’b010 // load op1
 com = 3’b011 // load op2 with one bit sign expansion
 com = 3’b100 // mult
*/
always @(posedge clock)
```
CHAPTER 8. AUTOMATA: SECOND ORDER, 2-LOOP DIGITAL SYSTEMS

case(com)
    3'b001: begin prod <= 0 ;
             state <= 0 ;
         end
    3'b010:  op1 <= op ;
    3'b011:  op2 <= {op[n-1], op}; // to allow left shift
    3'b100: begin {prod, op1} <= next ;
             state <= nextState ;
         end
    default prod <= prod ;
endcase

assign next = {{2{nextProd[n]}}, nextProd, op1[n-1:2]};
// begin algorithm
always @(*) if (state) case(op1[1:0])
    2'b00: begin nextProd = prod + op2 ;
             nextState = 0 ;
         end
    2'b01: begin nextProd = prod + (op2 << 1) ;
             nextState = 0 ;
         end
    2'b10: begin nextProd = prod - op2 ;
             nextState = 1 ;
         end
    2'b11: begin nextProd = prod ;
             nextState = 1 ;
         end
else case(op1[1:0])
    2'b00: begin nextProd = prod ;
             nextState = 0 ;
         end
    2'b01: begin nextProd = prod + op2 ;
             nextState = 0 ;
         end
    2'b10: begin nextProd = prod - (op2 << 1) ;
             nextState = 1 ;
         end
    2'b11: begin nextProd = prod - op2 ;
             nextState = 1 ;
         end
endcase
// end algorithm
assign product = {prod[n-1:0], op1};
endmodule

The sequence of commands for n = 8 is:

initial begin com = 3'b001 ;
    #2 com = 3'b010 ;
    op = 8'b1111_0001 ;
    #2 com = 3'b011 ;
    op = 8'b1111_0001 ;
    #2 com = 3'b100 ;
    #2 com = 3'b100 ;
    #2 com = 3'b100 ;
end
8.3.6 * “Bit-eater” automaton

A very useful function is to search the bits of a binary word in order to find the positions occupied by the 1s. For example, inspecting the number 00100100 we find in 2 steps a 1 on the 5-th position and another on the 2-nd position. A simple automaton does this operation in a number of clock cycles equal with the number of 1s contained in its initial state. In Figure 8.14 is represented The “bit-eater” automaton which is a simple machine containing:

- an n-bit state register
- a multiplexer used to initialize the automaton with the number to be inspected, if \( \text{load} = 1 \), then the register takes the input value, else the automaton’s loop is closed
- a priority encoder circuit which computes the index of the most significant bit of the state and activates the demultiplexer \( (E' = 0) \), if its enable input is activated, \( (\text{eat} = 1) \)
- an enabled decoder (or a demultiplexor) which decodes the value generated by the priority encoder applying 1 only to the input of one XOR circuit if \( \text{zero} = 0 \) indicating at least one bit of the state word is 1
- \( n \) 2-input XORs used to complement the most significant 1 of the state.

In each clock cycle after the initialization cycle the output takes the value of the index of the most significant bit of state having the value 1, and the next state is computed clearing the pointed bit.

Another, numerical interpretation is: while \( \text{state} \neq 0 \), the output of the automaton takes the integer value of the base 2 logarithm of the state value, \( \lfloor \log_2(\text{state}) \rfloor \), and the next state will be \( \text{next state} = \text{state} - \lfloor \log_2(\text{state}) \rfloor \). If \( \text{state} = 0 \), then \( n\text{ bit} = 1 \), the output takes the value 0, and the state remains in 0.
### 8.3.7 Sequential divider

The sequential divisor circuit receives two $n$-bit positive integers, the dividend and the divisor, and returns other $n$-bit positive integers: the quotient and the remainder. The sequential algorithm to compute:

$$\text{dividend} / \text{divisor} = \text{quotient} + \text{remainder}$$

a sort of “trial & error” algorithm which computes the $n$ bits of the quotient starting with $\text{quotient}[n-1]$. Then in the first step $\text{remainder} = \text{dividend} - (\text{divisor} \ll (n-1))$ is computed and if the result is a positive number the most significant bit of the quotient is 1 and the operation is validated as the new state of the circuit, else the most significant bit of the quotient is 0. Next step we try with $\text{divisor} \ll (n-2)$ computing $\text{quotient}[n-2]$, and so on until the $\text{quotient}[0]$ bit is determined. The structure of the circuit is represented in figure 8.15.

The Verilod description of the circuit is:

```verilog
module divisor #(parameter n = 8)(
    output reg [n-1:0] quotient ,
    output reg [n-1:0] remainder ,
    output error ,
    input [n-1:0] dividend ,
    input [n-1:0] divisor ,
    input [1:0] com ,
    input clk);

parameter nop = 2'b00,
    ld = 2'b01,
    div = 2'b10;

wire [n:0] sub;

assign error = (divisor == 0) & (com == div);
assign sub = {remainder, quotient[n-1]} - {1'b0, divisor} ;
```

![Sequential divisor](image-url)
8.4. *COMPOSING WITH SIMPLE AUTOMATA*

always @(posedge clk)
if (com == ld) begin
  quotient <= dividend;
  remainder <= 0;
end
else if (com == div) begin
  quotient <= {quotient[n-2:0], ~sub[n]};
  remainder <= sub[n] ?
    {remainder[n-2:0], quotient[n-1]} :
    sub[n-1:0];
end
endmodule

8.4  *Composing with simple automata*

Using previously defined simple automata some very useful subsystem can be designed. In this section are
presented some subsystems currently used to provide solutions for real applications: Last-In First-Out memory
(LIFO), First-In-First-Out memory (FIFO), and a version of the multiply accumulate circuit (MACC). All are simple
circuits because result as simple compositions of simple circuits, and all are expandable for any $n ... m$, where
$n ... m$ are a parameters defining different part of the circuit. For example, the memory size and the word size are
independent parameters is a FIFO implementation.

8.4.1  *LIFO memory*

The LIFO memory or the stack memory has many applications in structuring the processing systems. It is used
both for building the control part of the system, or for designing the data section of a processing system.

**Definition 8.14** *LIFO memory implements a data structure which consists of a string $S = <s_0, s_1, s_2, ...>$ of maximum $2^m$ n-bit recordings accessed for write, called push, and read, called pop, at the same end, $s_0$, called top of stack (TOS).* ◦

**Example 8.4** *Let be the stack $S = <s_0, s_1, s_2, ...>$. It evolve as follows under the sequence of five commands:*

- push a --> $S = <a, s_0, s_1, s_2, ...>
- push b --> $S = <b, a, s_0, s_1, s_2, ...>
- pop --> $S = <a, s_0, s_1, s_2, ...>
- pop --> $S = <s_0, s_1, s_2, ...>
- pop --> $S = <s_1, s_2, ...>

◦

Real applications request additional functions for a LIFO used for expression evaluation. An minimally ex-
panded set of functions for the LIFO $S = <s_0, s_1, s_2, ...> contains the following operations:

- nop: no operation; the contents of $S$ in untouched
- write a: write a in TOS
  $<s_0, s_1, s_2, ...> --> <a, s_1, s_2, ...>
used for unary operations; for example:
  a = s_0 + 1
  the TOS is incremented and write back in TOS (pop, push = write)
- pop:
  $<s_0, s_1, s_2, ...> --> <s_1, s_2, ...>
• popwr: a: pop & write
  \(<s_0, s_1, s_2, ...> \rightarrow <a, s_2, ...>\)
  used for binary operations; for example:
  \(a = s_0 + s_1\)
  the first two positions in LIFO are popped, added and the result is pushed back into the LIFO memory (pop, pop, push = pop, write)

• push a:
  \(<s_0, s_1, s_2, ...> \rightarrow <a, s_0, s_1, s_2, ...>\)

A possible implementation of such a LIFO is presented in Figure 11.1, where:

• **Register File** is organized, using the logic surrounding it, as a \(2^m\) \(n\)-bit stream of words accessed at TOS

• **LeftAddrReg** is a \(m\)-bit register containing the pointer to TOS = \(s_0\)

• **Dec** is the decrement circuit pointing to \(s_1\)

• **IncDec** is the circuit which increment, decrement or do not touch the content of the register **LeftAddrReg** as follows:
  - increment for push
  - decrement for pop or popwr
  - keeps unchanged for nop or write

Its output is used to select the destination in **Register File** and to update, in each clock cycle, the content of the register **LeftAddrReg**.

![LIFO memory diagram](image)

Figure 8.16: **LIFO memory**. The **LeftAddrReg** register is, in conjunction with **IncDec** combinational circuit, an up/down counter used as stack pointer to organize in **Register File** an expression evaluation stack.

A Verilog description of the previously defined LIFO (stack) is:

```verilog
module lifo #(\'include "0_parameters.v")
  ( output [m-1:0] stack0, stack1,
    input [m-1:0] in ,
    input [2:0] com ,
    input reset , clock );
```
8.4. *COMPOSING WITH SIMPLE AUTOMATA*

```markdown
/* The command codes
nop = 3'b000, //
write = 3'b001, // we
pop = 3'b010, // dec
popwr = 3'b011, // dec, we
push = 3'b101; // inc, we */

reg [n-1:0] leftAddr; // the main pointer
wire [n-1:0] nextAddr;
// The increment/decrement circuit
assign nextAddr = com[2] ? (leftAddr + 1'b1) : (com[1] ? (leftAddr - 1'b1) : leftAddr);
// The address register for TOS
always @posedge clock if (reset) leftAddr <= 0;
else leftAddr <= nextAddr;
// The register file
reg [m-1:0] file[0:(1'b1 << n)-1];
assign stack0 = file[leftAddr],
stack1 = file[leftAddr - 1'b1];
always @posedge clock if (com[0]) file[nextAddr] <= in;
endmodule
```

Faster implementations can be done using registers instead of different kind of RAMs and counters (see the chapter *SELF-ORGANIZING STRUCTURES: N-th order digital systems*). For big stacks, optimized solutions are obtained combining a small register implemented stack with a big RAM based implementation.

### 8.4.2 *FIFO memory*

The FIFO memory, or the *queue memory* is used to interconnect subsystems working logical, or both logical and electrical, asynchronously.

**Definition 8.15** FIFO memory implements a data structure which consists in a string of maximum $2^n$ n-bit recordings accessed for write and read, at its two ends. Full and empty signals are provided indicating the write operation or the read operation are not allowed.

A FIFO is considered synchronous if both read and write signals are synchronized with the same clock signal. If the two commands, read and write, are synchronized with different clock signals, then the FIFO memory is called asynchronous.

In Figure 8.17 is presented a solution for the synchronous version, where:

- **RAM** is a $2^n$ m-bit words two-port asynchronous random access memory, one port for write to the address \( w_{addr} \) and another for read form the address \( r_{addr} \)
- **write counter** is an \((n+1)\)-bit reselectable counter incremented each time a write is executed; its output is \( w_{addr}[n:0] \), initially it is reset
- **read counter** is an \((n+1)\)-bit reselectable counter incremented each time a read is executed; its output is \( r_{addr}[n:0] \), initially it is reset
- **eq** is a comparator activating its output when the least significant \( n \) bits of the two counters are identical.

FIFO works like a circular memory addressed by two pointers \( w_{addr}[n-1:0] \) and \( r_{addr}[n-1:0] \) running on the same direction. If the write pointer after a write operation becomes equal with the read pointer, then the memory is full and the full signal is 1. If the read pointer after a read operation becomes equal with the write pointer, then the memory is empty and the empty signal is 1. The \( n+1 \)-th bit in each counter is used to differentiate between empty and full when \( w_{addr}[n-1:0] \) and \( r_{addr}[n-1:0] \) are the same. If \( w_{addr}[n] \) and \( r_{addr}[n] \) are different, then \( w_{addr}[n-1:0] = r_{addr}[n-1:0] \) means full, else it means empty.
Figure 8.17: **FIFO memory.** Two pointers, evolving in the same direction, and a two-port RAM implement a LIFO (queue) memory. The limit flags are computed combinational from the addresses used to write and to read the memory.

The circuit used to compare the two addresses is a combinational one. Therefore, its output has a hazardous behavior which affects the outputs `full` and `empty`. These two outputs must be used carefully in designing the system which includes this FIFO memory. The problem can be managed because the system works in the same clock domain (clock is the same for both ends of FIFO and for the entire system). We call this kind of FIFO **synchronous FIFO**.

**VeriSim 8.1** A Verilog synthesisable description of a synchronous FIFO follows:

```verilog
module simple_fifo(output [31:0] out ,
                   output empty ,
                   output full ,
                   input [31:0] in ,
                   input write ,
                   input read ,
                   input reset ,
                   input clock );
wire [9:0] write_addr, read_addr;

counter write_counter( .out (write_addr ),
                       .reset (reset ),
                       .count_up (write ),
                       .clock (clock ));
read_counter( .out (read_addr ),
              .reset (reset ),
              .count_up (read ),
              .clock (clock ));
dual_ram memory(.out (out ),
                 .in (in ),
                 .clock (clock ));
```

---

The circuit used to compare the two addresses is a combinational one. Therefore, its output has a hazardous behavior which affects the outputs `full` and `empty`. These two outputs must be used carefully in designing the system which includes this FIFO memory. The problem can be managed because the system works in the same clock domain (clock is the same for both ends of FIFO and for the entire system). We call this kind of FIFO **synchronous FIFO**.

**VeriSim 8.1** A Verilog synthesisable description of a synchronous FIFO follows:

```verilog
module simple_fifo(output [31:0] out ,
                   output empty ,
                   output full ,
                   input [31:0] in ,
                   input write ,
                   input read ,
                   input reset ,
                   input clock );
wire [9:0] write_addr, read_addr;

counter write_counter( .out (write_addr ),
                       .reset (reset ),
                       .count_up (write ),
                       .clock (clock ));
read_counter( .out (read_addr ),
              .reset (reset ),
              .count_up (read ),
              .clock (clock ));
dual_ram memory(.out (out ),
                 .in (in ),
                 .clock (clock ));
```
8.4. *COMPOSING WITH SIMPLE AUTOMATA*

![Diagram](image)

---

An asynchronous FIFO uses two independent clocks, one for write counter and another for read counter. This type of FIFO is used to interconnect subsystems working in different clock domains. The previously described circuit is unable to work as an asynchronous FIFO. The signals `empty` and `full` are meaningless, being generated in two clock domains. Indeed, `write counter` and `read counter` are triggered by different clocks generating the signal `eq` with hazardous transitions related to two different clocks: write clock and read clock. This signal can not be used neither in the system working with write clock nor in the system working with read clock. Read clock is unable to avoid the hazard generated by write clock, and write clock is unable to avoid the hazard generated by read clock. Special tricks must be used.

### 8.4.3 *The Multiply-Accumulate Circuit*

The functional automata can be composed in order to perform useful functions in a digital system. Otherwise, we can say that a function can be decomposed in many functional units, some of them being functional automata, in order to implement it efficiently. Let’s take the example of the Multiply-Accumulate Circuit (MACC) and implement it in few versions. It is mainly used to implement one of the most important numerical functions performed in our digital machines, the scalar product of two vectors: \(a_1 \times b_1 + \ldots + a_n \times b_n\).

We will offer in the following a solution involving two serially connected functional automata: an *accumulator automaton* and “bits eater” automaton.

The starting idea is that the multiplication is also an accumulation. Thus we use an accumulator automaton for implementing both operations, the multiplication and the sum of products, without any loss in the execution time.

The structure of the multiply-accumulate circuit is presented in Figure 8.19 and consists in:
“bits eater” automaton – used to indicates successively the positions of the bits having the value 1 from the first operand $a_i$; it also points out the end of the multiplication (see Figure 8.14)

combinational shifter – shifts the second operand, $b_i$, with a number of positions indicated by the previous automaton

accumulate automaton – performs the partial sums for each multiplication step and accumulate the sum of products, if it is not cleared after each multiplication (see Figure 8.11).

In order to execute a multiplication only we must execute the following steps:

- load the “beat-eater” automaton with the first operand and clear the content of the output register in accumulator automaton
- select to the input the second operand which remains applied to the input of the shifter circuit during the operation
- wait for the end of operation indicated by the done output.

The operation is performed in a number of clock cycles equal with the number of 1s of the first operand. Thus, the mean execution time is proportional with $n/2$. To understand better how this machine works the next example will be an automaton which controls it.

If a MACC function is performed the clear of the state register of the accumulator automaton is avoided after each multiplication. Thus, the register accumulates the results of the successive multiplications.
8.4. COMPOSING WITH SIMPLE AUTOMATA

8.4.4 Weighted Pseudo-Random Generator

A Pseudo-Random Generator, PRG, generates a balanced sequence of 0s and 1s. There are applications for PRG with programmable weight of 1's. The solution\(^5\) starts with an \(n\)-bit LFSR which provides a pseudo-random sequence of \(2^n-1\) \(n\)-bit numbers (revisit 8.3.2). A priority encoder is used to generate the selection bits for an \(n\)-input one-bit multiplexer. An example, for \(n=8\), is represented in Figure 8.20. During a cycle of 255 states:

- the input \(\text{in}[7]\) will be selected 128 times
- the input \(\text{in}[6]\) will be selected 64 times
- ...
- the input \(\text{in}[0]\) will be selected once

Therefore, the binary sequence on the output \(\text{wprs}\) will be on 1 a time interval proportional with the number \(w[7:0]\) applied on the multiplexer’s inputs. This means, if the output is measured randomly, then the probability to find 1 is \(w/255\), where \(w\) is the integer value coded by the input \(w\). There are various applications for this circuit.

One application of this circuit is in designing a Digital-to-Analog Converter, DAC. If the output is mediated by an integrator, then the resulting DC level is proportional with the value \(w\) applied on the multiplexer’s input.

Another application is in stochastic computing. Let be two Weighted Pseudo-Random Generator with the associated outputs, \(\text{wprs}(w_1)\) and \(\text{wprs}(w_2)\). If the sequence are independent, then ANDing them results a binary sequence with the probability of 1 equal with \((w_1 \times w_2)/255^2\). The 2-input AND performs the multiplication of the two probabilities.

In the application note from [Alfke ’73], instead of the LFSR is used a 8-bit binary counter. More, it is connected to the input of the priority encoder in two ways: with the output bits in normal order and with the output bits in reverse order. Please compare these versions with the version just presented.

\(^5\)The suggestion for this circuit comes from [Alfke ’73], p. 4-12.
8.5 Finite Automata: the Complex Automata

After presenting the elementary small automata and the large and simple functional automata it is the time to discuss about the complex automata. The main property of these automata is to use a random combinational circuit, CLC, for computing the state transition function and the output transition function. Designing a finite automaton means mainly to design two CLC: the loop CLC (associated to the state transition function $f$) and the output CLC (associated to the output transition function $g$).

8.5.1 Representing finite automata

A finite automaton is represented by defining its transition functions $f$, the state transition function, and $g$, the output transition function. For a half-automaton only the function $f$ defined.

Flow-charts

A flow-chart contains for each state a circle and for each type of transition an arrow. In each clock cycle the automaton “races” on an arrow going from the current state to the next state. In our simple model the “race” on arrow is done in the moment of the active edge of the clock.

The flow-chart for a half-automaton  The first version is a pure symbolic representation, where the flow chart is marked on each circle with the name of the state, and on each arrow with the transition condition, if any. The initial states can be additionally marked with the minus sign (−), and the final states can be additionally marked with the plus sign (+).

The second version is used when the input are considered in the binary form. Instead of arches are used rhombuses containing the symbol denoting a binary variable.

Example 8.5 Let be a finite half-automaton that receives on its input strings containing symbols from the alphabet $X = \{a, b\}$. The machine stops in the final state when the first sequence $bb$ is received. The first version of the associated flow-chart is in Figure 8.21a. Here is how the machine works:

- the initial state is $q_0$; if $a$ is received the machine remains in the same state, else, if $b$ is received, then the machine switch in the state $q_1$.
8.5. **FINITE AUTOMATA: THE COMPLEX AUTOMATA**

Figure 8.21: **Example of flow-chart for a half-automaton.** The machine is a “double b detector”. It stops when the first bb occurs.

- in the state $q_1$ the machine “knows” that one b was just received; if a is received the half-automaton switch back in $q_0$, else, if b is received, then the machine switch in $q_2$
- $q_2$ is the final state; the next state is unconditionally $q_2$.

The second version uses tests represented by a rhombus containing the tested binary input variable (see Figure 8.21b). The input $I$ takes the binary value 0 for the the symbol a and the binary value 1 for the symbol b.

The second version is used mainly when a circuit implementation is envisaged.

**The flow-chart for a Moore automaton** When an automaton is represented the output behavior must be also included.

The first, pure symbolic version contains in each circle besides, the name of the state, the value of the output in that state. The output of the automaton shows something which is meaningful for the user. Each state generates an output value that can be different from the state’s name. The output set of value are used to classify the state set. The input events are mapped into the state set, and the state set is mapped into the output set.

The second uses for each pair state/output one rectangle. Inside of the rectangle is the value of the output and near to it is marked the state (by its name, by its binary code,, or both).

**Example 8.6** The problem solved in the previous example is revisited using an automaton. The output set is $Y = \{0, 1\}$. If the output takes the value 1, then we learn that a double b was already received. The state set $Q = \{q_0, q_1, q_2\}$ is divided in two classes: $Q^0 = \{q_0, q_1\}$ and $Q^1 = \{q_2\}$. If the automaton stays in $Q^0$ with $out = 1$, then it is looking for bb. If the automaton stays in $Q^1$ with $out = 1$, then it stopped investigating the input because a double b was already received.
Figure 8.22: Example of flow-chart for a Moore automaton. The output of this automaton tells us: “bb was already detected”.

The associated flow-chart is in, in the first version represented by Figure 8.22a. The states $q_0$ and $q_1$ belong to $Q^0$ because in the corresponding circles we have $q_0/0$ and $q_1/0$. The state $q_2$ belongs to $Q^1$ because in the corresponding circle we have $q_2/1$. Because the evolution from $q_2$ does not depend by input, the arrow emerging from the corresponding circle is not labelled.

The second version (see Figure 8.22b) uses three rectangles, one for each state. ⋄

A meaningful event on the input of a Moore automaton is shown on the output with a delay of a clock cycle. All goes through the state set. In the previous example, if the second $b$ from $bb$ is applied on the input in the period $T_i$ of the clock cycle, then the automaton points out the event in the period $T_{i+1}$ of the clock cycle.

The flow-chart for a Mealy automaton The first, pure symbolic version contains on each arrow besides, the name of the condition, the value of the output generated in the state where the arrow starts with the input specified on the arrow.

The Mealy automaton reacts on its outputs more promptly to a meaningful input event. The output value depends on the input value from the same clock cycle.

The second, implementation oriented version uses rectangles to specify the output’s behavior.

Example 8.7 Let us solve again the same problem of $bb$ detection using a Mealy automaton. The resulting flow-chart is in Figure 8.23a. Now the output is activated (out = 1) when the automaton is in the state $q_1$ (one $b$ was detected in the previous cycle) and the input takes the value $b$. The same condition triggers the switch in the state $q_2$. In the final state $q_2$ the output is unconditionally 1. In the notation $-/-1$ the sign $-$ stands for “don’t care”.

Figure 8.23b represents the second representation. ⋄

We can say the Mealy automaton is a “transparent” automaton, because a meaningful change on its inputs goes directly to its output.
Figure 8.23: **Example of flow-chart for a Mealy automaton.** The occurrence of the second $b$ from $bb$ is detected as fast as possible.

**Transition diagrams**

Flow-charts are very good to offer an intuitive image about how automata behave. The concept is very well represented. But, automata are also actual machines. In order to help us to provide the real design we need different representation. Transition diagrams are less intuitive, but they work better for helping us to provide the image of the circuit performing the function of a certain automaton.

Transition diagrams uses Vetch-Karnaugh diagrams, VKD, for representing the transition functions. The representation maps the VKD describing the state set of the automaton into the VKDs defining the function $f$ and the function $g$.

Transition diagrams are about real stuff. Therefore, the symbols like $a, b, q_0, \ldots$ must be coded binary, because a real machine work with bits, 0 and 1, not with symbols.

The output is already coded binary. For the input symbols the code is established by “the user” of the machine (similarly the output codes have been established by “the user”). Let say, for the input variable, $X_0$, was decided the following codification: $a \rightarrow X_0 = 0$ and $b \rightarrow X_0 = 1$.

Because the actual value of the state is “hidden” from the user, the designer has the freedom to assign the binary values according to its own (engineering) criteria. Because the present approach is a
theoretical one, we do not have engineering criteria. Therefore, we are completely free to assign the binary codes. Two option are presented:

**option 1:** \( q_0 = 00, q_1 = 01, q_2 = 10 \)

**option 2:** \( q_0 = 00, q_1 = 10, q_2 = 11 \)

For both the external behavior of the automaton must be the same.

**Transition diagrams for half-automata**  The transition diagram maps the reference VKD into the next state VKD, thus defining the state transition function. Results a representation ready to be used to design and to optimize the physical structure of a finite half-automaton.

**Example 8.8**  The flow-chart from Figure 8.21 has two different correspondent representations as transition diagrams in Figure 8.24, one for the option 1 of coding (Figure 8.24a), and another for the option 2 (Figure 8.24b).

![Transition Diagrams](image)

Figure 8.24: Example of transition diagram for a half-automaton.  a. For the option 1 of coding.  b. For the option 2 of coding.

In VKD \( S_1, S_0 \) each box contains a 2-bit code. Three of them are used to code the states, and one will be ignored. VKD \( S_1^+, S_0^+ \) represents the transition from the corresponding states. Thus, for the first coding option:

- from the state codded 00 the automaton switch in the state 0x, that is to say:
  - if \( X_0 = 0 \) then the next state is 00 \((q_0)\)
  - if \( X_0 = 1 \) then the next state is 01 \((q_1)\)

- from the state codded 01 the automaton switch in the state x0, that is to say:
  - if \( X_0 = 0 \) then the next state is 00 \((q_0)\)
  - if \( X_0 = 1 \) then the next state is 10 \((q_2)\)
8.5. FINITE AUTOMATA: THE COMPLEX AUTOMATA

- from the state coded 10 the automaton switch in the same state, 10 that is the final state
- the transition from 11 is not defined.

If in the clock cycle $T_i$ the state of the automaton is $S_{1_i}S_{0_i}$ (defined in the reference VKD), then in the next clock cycle, $T_{i+1}$, the automaton switches in the state $S_{1_{i+1}}S_{0_{i+1}}$ (defined in the next state VKD).

For the second coding option:
- from the state coded 00 the automaton switch in the state $X_{0_{0}}$, that is to say:
  - if $X_{0} = 0$ then the next state is 00 ($q_0$)
  - if $X_{0} = 1$ then the next state is 10 ($q_1$)
- from the state coded 10 the automaton switch in the state $X_{0_{1}}X_{0_{0}}$, that is to say:
  - if $X_{0} = 0$ then the next state is 00 ($q_0$)
  - if $X_{0} = 1$ then the next state is 11 ($q_2$)
- from the state coded 11 the automaton switch in the same state, 11 that is the final state
- the transition from 01 is not defined.

⋄ The transition diagram can be used to extract the Boolean functions of the loop of the half-automaton.

**Example 8.9** The Boolean function of the half-automaton working as “double b detector” can be extracted from the transition diagram represented in Figure 8.24a (for the first coding option). Results:

\[
S_{1_i}^+ = S_{1_i} + X_{0_i}S_{0_i}
\]
\[
S_{0_i}^+ = X_{0_i}S_{1_i}S'_{0_i}
\]

⋄

**Transition diagrams Moore automata** The transition diagrams define the two transition functions of a finite automaton. To the VKDs describing the associated half-automaton is added another VKD describing the output’s behavior.

**Example 8.10** The flow-chart from Figure 8.22 have a correspondent representation in the transition diagrams from Figure 8.25a or Figure 8.25b. Besides the transition diagram for the state, the output transition diagrams are presented for the two coding options.

For the first coding option:
- for the states coded with 00 and 01 the output has the value 0
- for the state coded with 10 the output has the value 1
- we do not care about how works the function $g$ for the state coded with 11 because this code is not used in defining our automaton (the output value can 0 or 1 with no consequences on the automaton’s behavior).
CHAPTER 8. AUTOMATA: SECOND ORDER, 2-LOOP DIGITAL SYSTEMS

Example 8.11 The resulting output function is:

\[ \text{out} = S_1. \]

Now the resulting automaton circuit can be physically implemented, in the version resulting from the first coding option, as a system containing a 2-bit register and few gates. Results the circuit in Figure 8.26, where:

- the 2-bit register is implemented using two resetable D flip-flops
- the combinational loop for state transition function consists in few simple gates
- the output transition function is so simple as no circuit are needed to implement it.

When \( \text{reset} = 1 \) the two flip-flops switch in 0. When \( \text{reset} = 0 \) the circuit starts to analyze the stream received on input symbol by symbol. In each clock cycle a new symbol is received and the automaton switches according to the new state computed by three gates.

Transition diagrams Mealy automata The transition diagrams for a Mealy automaton are a little different from those of Moore, because the output transition function depends also by the input variable. Therefore the VKD defining \( g \) contains, besides 0s and 1s, the input variable.

Example 8.12 Revisiting the same problem result, in Figure 8.27 the transition diagrams associated to the flow-chart from Figure 8.23.

The two functions \( f \) are the same. The function \( g \) is defined for the first coding option (Figure 8.27a) as follows:

- in the state coded by 00 (\( q_0 \)) the output takes value 0
8.5. **FINITE AUTOMATA: THE COMPLEX AUTOMATA**

- in the state coded by 01 \((q_1)\) the output takes value 0
- in the state coded by 10 \((q_2)\) the output takes value 1
- in the state coded by 11 (unused) the output takes the “don’t care” value

Extracting the function \(\text{out}\) results:

\[
\text{out} = S_1 + X_0 S_0
\]

a more complex from compared with the Moore version. (But fortunately \(\text{out} = S_1^+,\) and the same circuits can be used to compute both functions. Please ignore. Engineering stuff.)

**Procedures**

The examples used to explain how the finite automata can be represented are simple because of obvious reasons. The real life is much more complex and we need tools to face its real challenges. For real problems software tools are used to provide actual machines. Therefore, software oriented representation must be provided for representing automata. The so called Hardware Description Languages, HDLs, are widely used to manage complex applications. (The Verilog HDL is used to exemplify the procedural way to specify a finite automaton.)

**HDL representations for Moore automata**

A HDL (Verilog, in our example) representation consists in a program module describing the connections and the behavior of the automaton.

**Example 8.13** The same “bb detector” is used to exemplify the procedures used for the Moore automaton representation.

```verilog
module moore_automaton(out, in, reset, clock);
// input codes
```

Figure 8.26: The Moore version of “bb detector” automaton.
Figure 8.27: Example of transition diagram for a Mealy automaton.

parameter  a = 1’b0,
b = 1’b1;
// state codes
parameter  init_state = 2’b00,   // the initial state
          one_b_state = 2’b01,  // the state for one b received
          final_state = 2’b10;  // the final state
// output codes
parameter  no = 1’b0,   // no bb yet received
          yes = 1’b1;   // two successive b have been received
// external connections
input   in, reset, clock;
output  out;

reg   [1:0] state;  // state register
reg    out;  // output variable
// f: the state sequential transition function
always @(posedge clock)
if (reset) state <= init_state;
else  case(state)
    init_state : if (in == b) state <= one_b_state;
                else state <= init_state;
    one_b_state : if (in == b) state <= final_state;
                  else state <= init_state;
    final_state : state <= final_state;
    default    : state <= 2’bx;
endcase
// g: the output combinational transition function
always @(state) case(state)
8.5. **FINITE AUTOMATA: THE COMPLEX AUTOMATA**

```
init_state : out = no;
one_b_state : out = no;
final_state : out = yes;
default out = 1’bx;
endcase
endmodule
```

**HDL representations for Mealy automata** A Verilog description consists in a program module describing the connections and the behavior of the automaton.

**Example 8.14** *The same “bb detector” is used to exemplify the procedures used for the Mealy automaton representation.*

```
module mealy_automaton(out, in, reset, clock);
  parameter a = 1’b0,
             b = 1’b1;
  parameter init_state = 2’b00, // the initial state
                   one_b_state = 2’b01, // the state for one b received
                      final_state = 2’b10; // the final state
  parameter no = 1’b0, // no bb yet received
               yes = 1’b1; // two successive b have been received
  input in, reset, clock;
  output out;
  reg [1:0] state;
  reg out;
  always @(posedge clock)
    if (reset) state <= init_state;
    else case(state)
      init_state : if (in == b) state <= one_b_state;
                   else state <= init_state;
      one_b_state : if (in == b) state <= final_state;
                    else state <= init_state;
      final_state : state <= final_state;
      default state <= 2’bx;
    endcase
  always @(state or in) case(state)
    init_state : out = no;
    one_b_state : if (in == b) out = yes;
                  else out = no;
    final_state : out = yes;
                 default out = 1’bx;
  endcase
endmodule
```

The procedural representations are used as inputs for automatic design tools.
8.5.2 Designing Finite Automata

Preliminary Examples

The behavior of a finite automaton can be defined in many ways. Graphs, transition tables, flow-charts, transition V/K diagrams or HDL description are very good for defining the transition functions \( f \) and \( g \). All these forms provide non-recursive definitions. Thus, the resulting automata has the size of the definition in the same order with the size of the structure. Therefore, the finite automata are complex structures even when they have small size.

In order to exemplify the design procedure for a finite automaton let be two examples, one dealing with a 1-bit input string and another related with a system built around the multiply-accumulate circuit (MAC) previously described.

Example 8.15 The binary strings \( 1^n0^m \), for \( n \geq 1 \) and \( m \geq 1 \), are recognized by a finite half-automaton by its internal states. Let’s define and design it. The transition diagram defining the behavior of the half-automaton is presented in Figure 8.28, where:

![Transition diagram](image)

- \( q_0 \) - is the initial state in which 1 must be received, if not the the half-automaton switches in \( q_3 \), the error state
- \( q_1 \) - in this state at least one 1 was received and the first 0 will switch the machine in \( q_2 \)
- \( q_2 \) - this state acknowledges a well formed string: one or more 1s and at least one 0 are already received
- \( q_3 \) - the error state: an incorrect string was received.
The first step in implementing the structure of the just defined half-automaton is to **assign binary codes to each state**.

In this stage we have the absolute freedom. Any assignment can be used. The only difference will be in the resulting structure but not in the resulting behavior.

For a first version let be the codes assigned in square brackets in Figure 8.28. Results the transition diagram presented in Figure 8.29. The resulting transition functions are:

\[ Q_1^+ = Q_1 \cdot X_0 = \left( (Q_1 \cdot X_0)' \right)' \]

\[ Q_0^+ = Q_1 \cdot X_0 + Q_0 \cdot X_0' = \left( (Q_1 \cdot X_0)' \cdot (Q_0 \cdot X_0') \right)' \]

(The 1 from \( q_0^+ \) map is double covered. Therefore, it is taken into consideration as a “don’t care”.) The circuit is represented in Figure 8.68 in a version using inverted gated only. The 2-bit state register is designed by 2 D flip-flops. The **reset** input is applied on the set input of D-FF1 and on the reset input of D-FF0.

The Verilog behavioral description of the automaton is:
module rec_aut( output reg [1:0] state ,
               input  in ,
               input  reset ,
               input  clock );
always @(posedge clock)
  if (reset) state <= 2'b10;
  else
    case(state)
      2'b00: state <= 2'b00 ;
      2'b01: state <= {1'b0, ~in} ;
      2'b10: state <= {in, in} ;
      2'b11: state <= {in, 1'b1} ;
    endcase
endmodule

Example 8.16  Let us revisit the previous example in a more accurate implementation. Now a stream of characters to be recognized is delimited by the empty character \( e \). Therefore an actual stream to be recognized has the form:

\[ \ldots e e a \ldots a b b \ldots b e e \ldots \]

The stream is considered recognized only when it ends. The graph describing the automaton has one state more compared with the previous approach, without the delimiting symbol \( e \). It is represented in Figure 8.31. The automaton has the following 5 states:

- \( q_0 \): the initial state in which the automaton goes by reset, and if
  \[ \text{in} = a \]  the automaton switches in \( q_1 \) signaling that it entered in the search state
in = b  the automaton switches in q_3 signaling that the stream started wrong and the search process failed

in = e  the automaton remains in q_0 waiting the start of an input stream of as and bs

q_1 : the state waiting the flow of as

q_2 : the state waiting the flow of bs

q_3 : the state indicating that the string does not belong to the set \( 1^n0^m | n, m \geq 1 \)

q_4 : the state indicating that the string belongs to the set \( 1^n0^m | n, m \geq 1 \)

The symbols used to describe the automaton are binary codd as follows:

\[
X = \{a, b, e\} = \{01, 10, 00\}
\]

\[
Y = \{\text{wait, search, not, yes}\} = \{00, 11, 01, 10\}
\]

\[
Q = \{q_0, q_1, q_2, q_3, q_4\} = \{000, 001, 010, 011, 100\}
\]

The sets \( X \) and \( Y \) are defined by the user (the one who proposed the design), while the state coding is at the discretion of the designer. Then, the Table 8.1 describing the state transition function and the output transition function.

<table>
<thead>
<tr>
<th>q_2</th>
<th>q_1</th>
<th>q_0</th>
<th>x_1</th>
<th>x_0</th>
<th>q_{2+}</th>
<th>q_{1+}</th>
<th>q_{0+}</th>
<th>y_1</th>
<th>y_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 8.1: The truth table for the transition functions.
We have to solve 5 functions of 5 variables. Let us use V-K diagrams for 4 variables \( q_2, q_1, q_0, x_1 \) and the 5th variable, \( x_0 \), will be used to define the value of some boxes belonging to the diagrams. In Figure 8.32, we represented first the reference diagram to help us in defining the diagrams for \( f \) and \( g \). We will explain at length how the diagram for the function \( q_2+ \) is built:

- in the box 0 is filled with 0, because for \( \{q_2, q_1, q_0, x_1\} = \{0 0 0 0\} \) the output \( q_2+ \) does not depend on \( x_0 \) and takes the value 0
- in the box 1 in filled with 0, because for \( \{q_2, q_1, q_0, x_1\} = \{0 0 0 1\} \) the output \( q_2+ \) could be considered 0 if we decide to select for the don’t care value the value 0
- in the box 2 we fill up as in the box 0
- in the box 3 we fill up as in the box 1
- in the box 4 is filled with \( x_0' \), because for \( \{q_2, q_1, q_0, x_1\} = \{0 1 0 0\} \) the output \( q_2+ \) takes the value 1, if \( x_0 = 0 \) and the value 0 if \( x_0 = 1 \)
- in the boxes 5 and 7 we do as for the box 1
- in the box 6 we do as for the box 0
- in the box 8 in the box 1, because for \( \{q_2, q_1, q_0, x_1\} = \{1 0 0 0\} \) the output \( q_2+ \) does not depend on \( x_0 \) and takes the value 1
- in the box 9 in filled with 1, because for \( \{q_2, q_1, q_0, x_1\} = \{1 0 0 1\} \) the output \( q_2+ \) could be considered 1 if we decide to select for the don’t care value the value 1
- in the boxes 10 to 15 we fill up with don’t cares
The 5 function are extracted from the V-K diagrams in two stages. The first stage (which consider only the 1s from the diagram) is represented in Figure 8.33. The second stage (which considers the 1s as “don’t care”s) is represented in Figure 8.34. The resulting expressions are the following:

\[
\begin{align*}
q_2^+ &= q_2 + q_1 q_0' x_1' x_0' \\
q_1^+ &= q_2' x_1 + q_1 q_0 + q_0 x_0' + q_1 x_0 \\
q_0^+ &= q_1 q_0 + q_0 x_1' + q_2' q_1' q_0' x_1 + q_2' x_1' x_0 \\
y_1 &= q_2 + q_1' q_0 x_1 + q_1 q_0' x_1 + q_1 q_0' x_0' + q_1' x_1' x_0 \\
y_0 &= q_0 + q_2' x_1 + q_2' x_0
\end{align*}
\]

Figure 8.33: The first stage in the extracting algebraic expressions from V-K diagrams: the functions included in diagrams are ignored.

Figure 8.34: The second stage in the extracting algebraic expressions from V-K diagrams: the 1s are considered “don’t care”s.
Until now we minimized each of the 5 functions independently. Each function is minimal, but what about the whole circuit? The global minimization supposes the maximization of the number of gates shared in the implementation of the 5 functions. Therefore, we must try to define the surfaces in the V-K diagram so as to maximize the number of identical surfaces, even if we will be pushed to avoid the minimal form for some functions.

Figure 8.35: The first stage in the extracting algebraic expressions from V-K diagrams:

In Figure 8.35 the diagram for \( y_0 \) is modified: instead of the surface \( q_0 \), emphasize in Figure 8.33, here we have a smaller one, \( q_0 x_1' \), because this surface is selected also in the diagram for \( q_0' \). The impact on the final circuit is minimal: the fan-out of the D-FF0 is reduced.

Figure 8.36: The second stage in the extracting algebraic expressions from V-K diagrams:

The impact of this approach in the second stage is more important: the NAND circuit for \( q_2' q_1' \).
Figure 8.37:
$x_0$ is shared for the implementation of $q_0^+$ and $y_0$, and the NAND circuit for $q_1$ $q_0'$ $x_1'$ $x_0'$ is shared for the implementation of $q_2^+$ and $y_1$.

The resulting expressions are (with various brackets are emphasized the shared logic products):

- $q_2^+ = q_2 + [q_1 q_0' x_1' x_0']$
- $q_1^+ = q_2' x_1 + <q_1 q_0> + q_0 x_0' + q_1 x_0$
- $q_0^+ = <q_1 q_0> + (q_0 x_1') + q_2' q_1' q_0' x_1 + \{q_2' x_1' x_0\}$
- $y_1 = q_2 + q_1' q_0 x_1 + q_1 q_0' x_1 + [q_1 q_0' x_1' x_0'] + q_1' x_1' x_0$
- $y_0 = (q_0 x_1') + q_2' x_1 + \{q_2' x_1' x_0\}$

In Figure 8.37 is represented the resulting circuit, where the state register is implemented using 3 delay-flip-flops (D-FF) with their pair of outputs, one for $Q$ and another for $Q'$. Thus, we do not need inverters for the bits coding the state. The circuit is implemented using NAND gates by applying the de Morgan law which transforms the AND-OR structure in a NAND-NAND configuration.

Example 8.17 Let us revisit the previous example using another state coding:

$Q = \{q_0, q_1, q_2, q_3, q_4\} = \{000, 001, 111, 011, 010\}$

Then, the Table 8.2 describes the state transition function and the output transition function for the new coding. The transition functions are represented with 3-variable V-K diagrams in Figure 8.38

From V-K diagrams result the following expressions:

- $q_2^+ = q_1' q_0 x_1$
- $q_1^+ = q_2 + q_1 + q_0 x_0' + q_0' x_1$
- $q_0^+ = q_2' q_0 + q_1 (x_1 + x_0)$
- $y_1 = q_1 q_0' + q_2 x_0' + q_0' x_0 + q_2' q_1' q_0 (x_1 + x_0)$
- $y_0 = q_2' q_0 + q_1' (x_1 + x_0) = q_0^+$

The resulting circuit is represented in Figure 8.39

The size of the combinational circuits is only 70% from the previous solution. This reduction was obtained only by changing the state coding.

Example 8.18 The execution time of the MAC circuit is data dependent, depends on how many 1s contains the multiplicand. Therefore, the data flow through it has no a fix rate. The best way to interconnect this version of MAC
Figure 8.39: The circuit for the coding dominated by the reduce dependency coding style.
circuit supposes two FIFOs, one to its input and another to its output. Thus, a flexible buffered way to interconnect MAC is provided.

A complex finite automaton must be added to manage the signals and the commands associated with the three simple subsystems: IN FIFO, OUT FIFO, and MAC (see Figure 8.40). The flow-chart describing the version for performing multiplications is presented in Figure 8.41, where:

$q_0$ : *wait first* state – the system waits to have at least one operand in IN FIFO, clearing in the same time the output register of the accumulator automaton, when empty = 0 reads the first operand from IN FIFO and loads it in MAC

$q_1$ : *wait second* state – if IN FIFO is empty, the system waits for the second operand

$q_2$ : *multiply* state – the system perform multiplication while done = 0

$q_3$ : *write* state – the system writes the result in OUT FIFO and read the second operand from IN FIFO if full = 0 to access the first operand for the next operation, else waits while full = 1.

The flow chart can be translated into VK transition maps (see Figure 8.42) or in a Verilog description. From the VK transition maps result the following equations describing the combinational circuits for the loop \((q_1^+, q_0^+))\) and for the outputs.

\[
Q_1^+ = Q_1 \cdot Q_0 + Q_0 \cdot \text{empty}' + Q_1 \cdot \text{full}
\]

\[
Q_0^+ = Q_1' \cdot Q_0 + Q_0' \cdot \text{done}' + Q_1' \cdot \text{empty}'
\]
finite automata: the complex automata

The system consists in a multiply-accumulate circuit (MAC), two FIFOs and a finite automaton (FA) controlling all of them.

\[
clear = Q'_1 \cdot Q'_0
\]

\[
nop = Q'_1 \cdot Q'_0 + Q'_1 \cdot \text{empty}'
\]

\[
load = Q'_1 \cdot Q'_0 \cdot \text{empty}'
\]

\[
read = Q'_1 \cdot Q'_0 \cdot \text{full}' + Q'_1 \cdot Q'_0 \cdot \text{empty}'
\]

\[
write = Q'_1 \cdot Q'_0 \cdot \text{full}'
\]

The resulting circuit is represented in Figure 8.43, where the state register is implemented using 2 D flip-flops and the combinational circuits are implemented using a PLA.

If we intend to use a software tool to implement the circuit the following Verilog description is a must.

```verilog
module macc_control(read, write, load, clear, nop, empty, full, done, reset, clock);

input empty, full, done, reset, clock;
output read, write, load, clear, nop;

reg [1:0] state;
reg read, write, load, clear, nop; // as variables

parameter wait_first = 2’b00,
wait_second = 2’b01;
```
Figure 8.41: **Flow chart describing a Mealy finite automaton.** The flow-chart describes the finite automaton FA from Figure 8.40, which controls MAC and the two FIFOs in MAC system. (The state coding shown in parenthesis will be used in the next chapter.)

```plaintext
multiply = 2'b11,
write_result = 2'b10;

// THE STATE TRANSITION FUNCTION
always @(posedge clock)
  if (reset) state <= wait_first;
  else
    case(state)
      wait_first : if (empty) state <= wait_first;
                  else state <= wait_second;
      wait_second : if (empty) state <= wait_second;
                   else state <= multiply;
      multiply : if (done) state <= write_result;
                 else state <= multiply;
      write_result: if (full) state <= write_result;
                  else state <= wait_first;
    endcase

// THE OUTPUT TRANSITION FUNCTION (MEALY IMMEDIATE)
always @(state or empty or full or done)
case(state)
  wait_first : if (empty)
```
The resulting circuit will depend by the synthesis tool used because the previous description is “too” behavioral. There are tools which will synthesize the circuit coding the four states using four bits ....!!!!!. If we intend to impose a certain solution, then a more structural description is needed. For example, the following “very” structural code which translate directly the transition equations extracted from VK transition maps.

```verilog
module macc_control(read , // read from IN FIFO
        write , // write in OUT FIFO
        load , // load the multiplier in MAC
        clear , // reset the output of MAC
        nop , // stops the multiplication
        empty , // IN FIFO is empty
        full , // OUT FIFO is full
        done , // the multiplication is concluded
        reset
        ,
        clock );

        input empty, full, done, reset, clock;
        output read, write, load, clear, nop;
```

The transition VK diagrams for FA (see Figure 8.40). The reference diagram has a box for each state. The state transition diagram, $Q'_0 Q'_1$, contains in the same positions the description of the next state. For each output a diagram describe the output’s behavior in the corresponding state.

---

**Veitch-Karnaugh transition diagrams.** The transition VK diagrams for FA (see Figure 8.40). The reference diagram has a box for each state. The state transition diagram, $Q'_0 Q'_1$, contains in the same positions the description of the next state. For each output a diagram describe the output’s behavior in the corresponding state.

{read, write, load, clear, nop} = 5’b00011;
else
{read, write, load, clear, nop} = 5’b10111;

wait_second : if (empty)
{read, write, load, clear, nop} = 5’b00001;
else
{read, write, load, clear, nop} = 5’b00000;

multiply : {read, write, load, clear, nop} = 5’b00000;

write_result: if (full)
{read, write, load, clear, nop} = 5’b00000;
else
{read, write, load, clear, nop} = 5’b11000;
endcase
demodule
CHAPTER 8. AUTOMATA: SECOND ORDER, 2-LOOP DIGITAL SYSTEMS

Figure 8.43: FA's structure. The FA is implemented with a two-bit register and a PLA with 5 input variables (2 for state bits, and 3 for the input signals), 7 outputs and 10 products.

```verilog
reg [1:0] state; // as an actual register

// THE STATE TRANSITION FUNCTION
always @(posedge clock)
if (reset) state <= 2'b00;
else state <= {
    (state[1] & state[0] | state[0] & ~full),
    (state[1] & state[0] | state[0] & ~full),
    ~state[1] & ~state[0] & ~empty |
    state[1] & ~state[0] & ~full,
    state[1] & ~state[0] & ~full,
    state[1] & ~state[0] & ~full,
    ~state[1] & ~state[0],

assign read = ~state[1] & ~state[0] & ~empty |
    state[1] & ~state[0] & ~full,
write = state[1] & ~state[0] & ~full,
load = ~state[1] & ~state[0] & ~empty,
clear = ~state[1] & ~state[0],
endmodule
```

The resulting circuit will be eventually an optimized form of the version represented in Figure 8.43 because instead a PLA, the current tools use an minimized network of gates.

The finite automaton has two distinct parts:

- the simple, recursive defined part, that consists in the state register; it can be minimized only by minimizing the definition of the automaton
• the complex part, that consists in the PLA that computes functions \( f \) and \( g \) and this is the part submitted to the main minimization process.

Our main goal in designing finite automaton is to reduce the random part of the automaton, even if the price is to enlarge the recursive defined part. In the current VLSI technologies we prefer big size instead of big complexity. A big sized circuit has now a technological solution, but for describing very complex circuits we have not yet efficient solutions (maybe never).

State Coding

The function performed by an automaton does not depend by the way its states are encoded, because the value of the state is a “hidden variable”. But, the actual structure of a finite automaton and its proper functioning are very sensitive to the state encoding.

The designer uses the freedom to code in different way the internal state of a finite automaton for its own purposes. A finite automaton is a concept embodied in physical structures. The transition from concept to an actual structure is a process with many traps and corner cases. Many of them are avoided using an appropriate codding style.

Example 8.19 Let be a first example showing the structural dependency by the state encoding. The automaton described in Figure 8.44a has three state. The first codding version for this automaton is: \( q_0 = 00, q_1 = 01, q_2 = 10 \). We compute the next state \( Q_1^+ \), \( Q_0^+ \), and the output \( Y_1, Y_0 \) using the first two VK transition diagrams from Figure 8.44b:

\[
Q_1^+ = Q_0 + X_0Q_1', \\
Q_0^+ = Q_1'Q_0'x_0' \\
Y_1 = Q_0 + X_0Q_1' \\
Y_0 = Q_1'Q_0'.
\]

The second codding version for the same automaton is: \( q_0 = 00, q_1 = 01, q_2 = 11 \). Only the code for \( q_2 \) is different. Results, using the last two VK transition diagrams from Figure 8.44b:

\[
Q_1^+ = Q_1'Q_0 + X_0Q_1' = (Q_1 + (Q_0 + X_0))' \\
Q_0^+ = Q_1' \\
Y_1 = Q_1'Q_0 + X_0Q_1' = (Q_1 + (Q_0 + X_0))' \\
Y_0 = Q_0'.
\]

Obviously the second codding version provides a simpler and smaller combinational circuit associated to the same external behavior. In Figure 8.45 the resulting circuit is represented. \( \diamond \)
Minimal variation encoding  Minimal variation state assignment (or encoding) refers to the codes assigned to successive states.

**Definition 8.16** Codding with minimal variation means successive state are coded with minimal Hamming distance. ♦

**Example 8.20** Let be the fragment of a flow chart represented in Figure 8.46a. The state $q_i$ is followed by the state $q_j$ and the assigned codes differ only by the least significant bit. The same for $q_k$ and $q_l$ which both follow the state $q_j$. ♦

**Example 8.21** Some times the minimal variation encoding is not possible. An example is presented in Figure 8.46b, where $q_k$ can not be coded with minimal variation. ♦

The minimal variation coding generates a minimal difference between the reference VK diagram and the state transition diagram. Therefore, the state transition logical function extracted form the VK diagram can be minimal.

Reduced dependency encoding  Reduced dependency encoding refers to states which conditionally follow the same state. The reduced dependency is related to the condition tested.
8.5. **FINITE AUTOMATA: THE COMPLEX AUTOMATA**

---

**Figure 8.45: The resulting circuit** It is done for the second state assignment of the automaton defined in Figure 8.44a.

---

**Figure 8.46: Minimal variation encoding.** a. An example. b. An example where the minimal variation encoding is not possible.

---

**Definition 8.17** Reduced dependency encoding means the states which conditionally follow a certain state to be coded with binary configurations which differs minimal (have the Hamming distance minimal). ♦

---

**Example 8.22** In Figure 8.47a the states \( q_j \) and \( q_k \) follow, conditioned by the value of 1-bit variable \( X_0 \), the state \( q_i \). The assigned codes for the first two differ only in the most significant bit, and they are not related with the code of their predecessor. The most significant bit used to code the successors of \( q_i \) depends by \( X_0 \), and it is \( X'_0 \). We say: the next states of \( q_i \) are \( X'_0 11 \), for \( X_0 = 0 \) the next state is 111, and for \( X_0 = 1 \) it is 011. Reduced dependency means: only one bit of the codes associated with the successors of \( q_i \) depends by \( X_0 \), the variable tested in \( q_i \). ♦

---

**Example 8.23** In Figure 8.47b the transition from the state \( q_i \) depends by two 1-bit variable, \( X_0 \) and \( X_1 \). A reduced dependency coding is possible by only one of them. Without parenthesis is a reduced dependency coding by the variable \( X_1 \). With parenthesis is a reduced dependency coding by \( X_0 \). ♦

The reader is invited to provide the proof for the following theorem.
Figure 8.47: **Examples of reduced dependency encoding.**

- **a.** The transition from the state is conditioned by the value of a single 1-bit variable.
- **b.** The transition from the state is conditioned by two 1-bit variables.

**Theorem 8.2** If the transition from a certain state depends by more than one 1-bit variable, the reduced dependency encoding can not be provided for more than one of them.

The reduced dependency encoding is used to minimize the transition function because it allows to minimize the number of included variables in the VK state transition diagrams. Also, we will learn soon that this encoding style is very helpful in dealing with asynchronous input variables.

**Incremental coding** The incremental encoding provides an efficient encoding when we are able to use simple circuits to compute the value of the next state. An incrementer is the simple circuit used to design the simple automaton called counter. The incremental encoding allows sometimes to center the implementation of a big half-automaton on a presetable counter.

**Definition 8.18** Incremental encoding means to assign, whenever it is possible, for a state following \( q_i \), a code determined by incrementing the code of \( q_i \).

Incremental encoding can be useful for reducing the complexity of a big automaton, even if sometimes the price will be to increase the size. But, as we more frequently learn, bigger size is a good price for reducing complexity.

**One-hot state encoding** The register is the simple part of an automaton and the combinational circuits computing the state transition function and the output function represent the complex part of the automaton. More, the speed of the automaton is limited mainly by the size and depth of the associated combinational circuits. Therefore, in order to increase the simplicity and the speed of an automaton we can use a coding style which increase the dimension of the register reducing in the same time the size and the depth of the combinational circuits. Many times a good balance can be established using the one-hot state encoding.

**Definition 8.19** The one-hot state encoding associates to each state a bit, and consequently the state register has a number of flip-flops equal with the number of states.

All previous state encodings used a log-number of bits to encode the state. The size of the state register will grow, using one-hot encoding, from \( O(\log n) \) to \( O(n) \) for an \( n \)-state finite automaton. Deserves to pay sometimes this price for various reasons, such as speed, signal accuracy, simplicity, ….
Minimizing finite automata

There are formal procedures to minimize an automaton by minimizing the number of internal states. All these procedures refer to the concept. When the conceptual aspects are solved remain the problems related with the minimal physical implementation. Follow a short discussion about minimizing the size and about minimizing the complexity.

Minimizing the size by an appropriate state coding There are some simple rules to be applied in order to generate the possibility to reach a minimal implementation. Applying all of these rules is not always possible or an easy task and the result is not always guarantee. But it is good to try to apply them as much as possible.

A secure and simple way to optimize the state assignment process is to evaluate all possible coding versions and to choose the one which provide a minimal implementation. But this is not an effective way to solve the problem because the number of different versions is in $O(n!)$. For this reason are very useful some simple rules able to provide a good solution instead of an optimal one.

A lucky, inspired, or trained designer will discover an almost optimal solution applying the following rule in the order they are enounced.

**Rule 1** : apply the reduced dependency coding style whenever it is possible. This rule allows a minimal occurrence of the input variable in the VK state transition diagrams. Almost all the time this minimal occurrence has as the main effect reducing the size of the state transition combinational circuits.

**Rule 2** : the states having the same successor with identical test conditions (if it is the case) will have assigned adjacent codes (with the Hamming distance 1). It is useful because brings in adjacent locations of a VK diagrams identical codes, thus generating the conditions to maximize the arrays defined in the minimizing process.

**Rule 3** : apply minimal variation for unconditioned transitions. This rule generates the conditions in which the VK transition diagram differs minimally from the reference diagram, thus increasing the chance to find bigger surfaces in the minimizing process.

**Rule 4** : the states with identical outputs are coded with minimal Hamming distance (1 if possible). Generates similar effects as Rule 2.

To see at work these rules let’s take an example.

**Example 8.24** Let be the finite automaton described by the flow-chart from Figure 8.48. Are proposed two coding versions, a good one (the first), using the coding rules previously listed, and a bad one (the second with the codes written in parenthesis), ignoring the rules.

For the first coding version results the expressions:

$$Q_2^+ = Q_2Q_0 + Q_1Q_1$$
$$Q_1^+ = Q_1Q_0 + Q_2Q_0 + Q_2Q_0X_0$$
$$Q_0^+ = Q_0 + Q_2Q_1X_0'$$
$$Y_2 = Q_2 + Q_1Q_0$$
Figure 8.48: **Minimizing the structure of a finite automaton.** Applying appropriate coding rules the occurrence of the input variable $X_0$ in the transition diagrams can be minimized, thus resulting smaller Boolean forms.
8.5. FINITE AUTOMATA: THE COMPLEX AUTOMATA

\[ Y_1 = Q_2 Q_1 Q'_0 + Q'_2 Q'_1 \]
\[ Y_0 = Q_2 + Q'_1 + Q'_0 \]

the resulting circuit having the size \( S_{\text{CLCver1}} = 37 \).

For the second coding version results the expressions:

\[ Q'_2 = Q_2 Q_1 Q'_0 + Q'_2 Q_0 + Q'_2 Q_0 X_0 + Q_1 Q'_0 X'_0 \]
\[ Q'_1 = Q'_1 Q_0 + Q'_2 Q'_1 + Q'_2 X'_0 \]
\[ Q'_0 = Q'_1 Q_0 + Q'_2 Q'_1 + Q'_2 X_0 \]
\[ Y_2 = Q_2 Q'_0 + Q_2 Q_1 + Q'_2 Q'_1 Q_0 + Q_1 Q'_0 \]
\[ Y_1 = Q'_2 Q_0 + Q'_2 Q'_1 \]
\[ Y_0 = Q_2 + Q'_1 + Q'_0 \]

the resulting circuit having the size \( S_{\text{CLCver2}} = 50 \).

Minimizing the complexity by one-hot encoding Implementing an automaton with one-hot encoded states means increasing the simple part of the structure, the state register. It is expected at least a part of this additional structure to be compensated by a reduced combinational circuit used to compute the transition functions. But, for sure the entire complexity is reduced because of a simpler combinational circuit.

Example 8.25 Let be the automaton described by the flow-chart from Figure 8.49, for which two coding version are proposed: a one-hot encoding using 6 bits (\( Q_6 \ldots Q_1 \)), and a compact binary encoding using only 3 bits (\( Q_2 Q_1 Q_0 \)).

Figure 8.49: Minimizing the complexity using one-hot encoding.

The outputs are \( Y_6, \ldots, Y_1 \) each active in a distinct state.
Version 1: with "one-hot" encoding  The state transition functions, \( Q^+_i, i = 1, \ldots, 6 \), can be written directly inspecting the definition. Results:

\[
\begin{align*}
Q^+_1 &= Q_4 + Q_5 + Q_6 \\
Q^+_2 &= Q_1X_0 \\
Q^+_3 &= Q_1X'_0 \\
Q^+_4 &= Q_2X'_0 \\
Q^+_5 &= Q_2X_0 + Q_3X'_0 \\
Q^+_6 &= Q_3X_0
\end{align*}
\]

Because in each state only one output bit is active, results:

\[ Y_i = Q_i, \text{ pentru } i = 1, \ldots, 6. \]

The combinational circuit associated with the state transition function is very simple, and for outputs no circuits are needed. The size of the entire combinational circuit is \( S_{\text{CLC, var}} = 18 \), with the big advantage that the outputs come directly from a flip-flop without additional unbalanced delays or other parasitic effects (like different kinds of hazards).

Version 2: compact binary coding  The state transition functions for this c od ding version (see Figure 8.49 for the actual binary codes) are:

\[
\begin{align*}
Q^+_2 &= Q_2Q_0 + Q_0X_0 + Q'_2Q'_1X_0 \\
Q^+_1 &= Q'_2Q_0 + Q'_2Q'_1 + Q_0X'_0 \\
Q^+_0 &= Q'_2Q'_1
\end{align*}
\]

For the output transition function an additional decoder, \( DCD_3 \), is needed. The resulting combinational circuit has the size \( S_{\text{CLC, var}} = 44 \), with the additional disadvantage of generating the outputs signal using a combinational circuit, the decoder.

Asynchronous inputs

A real automaton is connected to the “external world” from which it receives of where it sends signals only partially are controlled. This happens mainly when the connection is not sequential, mediated by a synchronous register, because sometimes this is not possible. The designer controls very well the signals on the loop. But, the uncontrolled arriving signals can by very dangerous for the proper functioning of an automaton. Similarly, an uncontrolled output signal can have “hazardous” behaviors.

An automaton is implemented as a synchronous circuit changing its internal states at each active (positive or negative) edge of clock. Let us remember the main restrictions imposed by the set-up time and hold time related to the active edge of a clock applied to a flip-flop. No input signal can change in the time interval beginning \( t_{SU} \) before the clock transition and ending \( t_{HF} \) after the clock transition. Call it the prohibited time interval. But, if at least one input of a certain finite automaton determines a switch on at least one input of the state register, then no one can guarantee a proper functioning of that automaton.

Let be a finite automaton with one input, \( A \), changing unrelated with the system clock. Its transition can determine a transition on the input of a state flip-flop in the prohibited time interval. We call this
kind of variable asynchronous input variable or simply asynchronous variable, and we use for it the notation $A^*$. If, in a certain state the automaton test $A^*$ and switches in $1A00$ (which means in 1000 if $A^* = 0$, or 1110 is $A^* = 1$), then we are in trouble. The actual behavior of the automaton will allow also the transition in 1010 and in 1100, which means the actual transition of the automaton will be in fact in $1xx0$, where $x \in \{0, 1\}$. Indeed, if $A^*$ determine the transition of two state flip-flops in the prohibited time interval, any binary configuration can be loaded in that flip-flops, not only 11 or 00.

The case of one asynchronous input  What is the solution for this pathological behavior induced by one asynchronous variable? To use reduced dependency coding for the transition from the state in which $X_0^*$ is tested. If the state assignment will allow, for example, a transition to $1110$, then the behavior of the automaton becomes coherent. Indeed, if $X_0^*$ determine a transition in the prohibited time interval on only one state flip-flop, then the next state will be only 1110 or 1100. In both cases the automaton behaves according to its definition. If the transition of $X_0^*$ is considered then the behavior of the automaton is correct, but even if the transition is not catched it will be considered at the net clock cycle.

Example 8.26 In Figure 8.50 is defined a 3-state automaton with the asynchronous input variable $X_0^*$. Two code assignment are proposed. The first one uses the minimal variation kind of codding, and the second uses for the transition from the state $q_0$ a reduced dependency codding.

The first codding is: $q_0 = 01$, $q_1 = 00$, $q_2 = 10$, $q_3 = 11$.

Results the following circuits for state transition:

$$Q_1^+ = Q_0' + Q_0'X_0$$
$$Q_0^+ = Q_1 + Q_0X_0.$$  

The transition from the state $Q_1Q_0 = 01$ is dangerous for the proper functioning of the finite automaton. Indeed, from $q_0$ the transition is defined by:

$$Q_1^+ = X_0, \ Q_0^+ = X_0$$

and the transition of $X_0$ can generate changing signals on the state flip-flops in the prohibited time interval. Therefore, the state $q_0$ can be followed by any state.

The second codding, with reduced (minimal) dependency, is: $q_0 = 01$, $q_1 = 00$, $q_2 = 11$, $q_3 = 10$.

Results the following equations describing the loop circuits:

$$Q_1^+ = Q_1Q_0 + Q_1'Q_0' + Q_0X_0$$
$$Q_0^+ = Q_0.$$  

The transition from the critical state, $q_0$, is

$$Q_1^+ = A, \ Q_0^+ = Q_0.$$  

Only $Q_1^+$ depends by the asynchronous input.

The size, the depth and the complexity of the resulting circuit is similar, but the behavior is correct only for the second version. The correctness is achieved only by a proper encoding. ◦
Obviously, transition determined by more than one asynchronous variable must be avoided, because, as we already know, the reduced dependency coding can be done only for one asynchronous variable in each state. But, what is the solution for more than one asynchronous input variable? Introducing new states in the definition of the automaton, so as in each state no more than one asynchronous variable will be tested.

The case of more than one asynchronous inputs If there are more than one asynchronous inputs the danger occurs when more than one of such variables are tested in the same state. Let be these asynchronous variables $A^*$ and $B^*$, and, for example, there are a transition in $1A^*B^*0$. Then, this transition become equivalent with the transition in $1xx0$. According to Theorem 8.2, the reduce dependency encoding, at the transition from each state, is possible only for one asynchronous variable. What can be done in this case?

We can try to synchronize the two variable, $A^*$ and $B^*$, using a register. This attempt is represented in
Figure 8.51. But, unfortunately, this solution doesn’t work. Because, the two asynchronous variables can switch in the prohibited time interval, the active edge of clock in the $t_2$ moment can load in the register any 2-bit binary configuration. Thus, in the flow of input data could be inserted parasitic configurations such as $10 \rightarrow 00 \rightarrow 01$ or $10 \rightarrow 11 \rightarrow 01$ instead of the correct flow of data represented by $10 \rightarrow 10 \rightarrow 01$ or by $10 \rightarrow 01 \rightarrow 01$.

We must conclude that synchronizing binary configurations consisting in more than on bit is not possible in a digital system.

For our 2-input asynchronous input we must propose the following solution: in the flowchart a supplementary state must be introduces so as in each state no more than one asynchronous input variable is tested.

Example 8.27 Let be the fragment of flowchart from Figure 8.52a, where two asynchronous variable,
\(A^*\) and \(B^*\), are tested. An additional state is added in Figure 8.52b. In this new state the output is the same with the first state.

![Diagram](image)

**Figure 8.52:** Reduce dependency coding. 

**a.** The coding is possible only related to one asynchronous variable. The first version is according to \(B^*\), while the second (in in square brackets) is according to \(A^*\).  

**b.** The second version added a new state, doubling the first state.

\[\text{\ding{277}}\]

### The Hazard

Some additional problems must be solved to provide accurate signals to the outputs of the immediate finite automata. The output combinational circuit introduces, besides a delay due to the propagation time through the gate used to build it, some parasitic effects due to a kind of “indecision” in establishing the output value. Each bit on the output is computed using a different network of gates and the effect of an input switch reaches the output going through different logic paths. The propagation through these various circuits can provide hazardous transient behaviors on certain outputs.

**Hazard generated by asynchronous inputs**  A first form of hazardous behavior, or simply hazard, is generated by the impossibility to have synchronous transitions to the input of a combinational circuit.

Let be circuit from Figure 8.53a representing the typical gates receiving the signal \(A\) and \(B\), ideally represented in Figure 8.53b. Ideally means the two signals switch synchronously. They are considered ideal because no synchronous signal can be actually generated. In Figure 8.53c and Figure 8.53d two actual relations between the signals \(A\) and \(B\) are represented (other two are possible, but our purpose this two cases will allow to emphasize the main effects of the actual asynchronicity).

Ideally, the AND gate must have the output continuously on 0, and the OR and XOR gates on 1. Because of the inherent asynchronicity between the input signals some parasitic transitions occur to the outputs of the three gates (see Figure 8.53c and Figure 8.53d). Ideally, to the inputs of the three gates are applied only two binary configurations: \(AB = 10\) and \(AB = 01\). But, because of the asynchronicity between the two inputs, all possible binary configurations are applied, two of them for long time (\(AB = 10\) and \(AB = 01\)) and the other two (\(AB = 00\) and \(AB = 11\)) only for short (transitory) time. Consequently, transitory effects are generated, by hazard, on the outputs of the three circuits.

Some times the transitory unexpected effects can be ignored including them into the transition time of the circuit. But, there are applications where they can generate big disfunctionalities. For example,
when one of the hazardous output is applied on a set or reset input of a latch.

![Diagram of asynchronous inputs generating hazard](image)

Figure 8.53: How the asynchronous inputs generate hazard.

In order to offer an additional explanation for this kind of hazard VK diagrams are used in Figure 8.54, where in the first column of diagrams the ideal case is presented (the input switches directly to the desired value). In the next two column the input reach the final value through an intermediary value. Some times the intermediary value is associated with a parasitic transition of the output.

When between two subsystems multi-bit binary configurations are transferred, parasitic configuration must be considered because of the asynchronicity. The hazardous effects can be “healed” being “patient” waiting for the hazardous transition to disappear. But, we can wait only if we know when the transition occurred, i.e., the hazard is easy to be avoided in synchronous systems.

Simply, when more than one input of a combinational is changing we must expect hazardous transitions at least on some outputs.
Figure 8.54: VK diagrams explaining the hazard due to the asynchronous inputs. "A < B" means the input A switch before the input B, and "A > B" means the input B switch before the input A.

**Propagation hazard** Besides the hazard generated by the two or many switching inputs there exists hazard due to the transition of only one input. In this case the internal propagations inside of the combinational circuit generate the hazard. It could by a sort of asynchronicity generated by the different propagation paths inside the circuit.

Let be a simple example of the circuit represented in Figure 8.55a, where two input are stable \(A = C = 1\) and only one input switches. The problem of asynchronous inputs is not an issue because only one input is in transition. In Figure 8.55b the detailed wave forms allow us to emphasize a parasitic transition on the output D. For \(A = C = 1\) the output must stay on 1 independent by the value applied on B. The actual behavior of the circuit introduces a parasitic (hazardous) transition in 0 due to the switch of B from 1 to 0. An ideal circuit with zero propagation times should maintain its output on 1.

A simple way to explain this kind of hazard is to say that in the VK diagram of the circuit (see Figure 8.55c) when the input “flies” from one surface of 1s to another it goes through the 0 surface generating a temporary transition to 0. In order to avoid this transitory journey through the 0 surface an additional surface (see Figure 8.55d) is added to transform the VK diagram in a surface containing two contiguous surfaces, one for 0s and one for 1s. The resulting equation of the circuit has an additional term: \(AC\). The circuit with the same logic behavior, but without hazardous transitions is represented in Figure 8.55e.

**Example 8.28** Let be the function presented in VK diagram from Figure 8.56a. An immediate solution is shown in Figure 8.56b, where a square surface is added in the middle. But this solution is partial because ignores the fact that the VK diagram is defined as a thor, with a three-dimensional adjacency. Consequently the surfaces \(A'B'CD'\) and \(A'B'CD\) are adjacent, and the same for \(AB'CD\) and \(A'B'CD\).
8.5. FINITE AUTOMATA: THE COMPLEX AUTOMATA

Figure 8.55: The typical example of propagation hazard. a. The circuit. b. The wave forms. c. The VK diagram of the function executed by the circuit. d. The added surface allowing the behavior of the circuit to have a continuous 1 surface. e. The equivalent circuit without hazard.

Therefore, the solution to completely avoid the hazard is presented in Figure 8.56c, where two additional surfaces are added.

Theorem 8.3 If the expression of the Boolean function

\[ f(x_{n-1}, \ldots, x_0) \]

takes the form

\[ x_i + x'_i \]

for at least one combination of the other variables than \( x_i \), then the actual associated circuit generates hazard when \( x_i \) switches. (The theorem of hazard)

Example 8.29 The function \( f(A, B, C) = AB' + BC \), is hazardous because: \( f(1, B, 1) = B' + B \).

The function \( g(A, B, C, D) = AD + BC + A'B' \) is hazardous because: \( g(A, 0, -, 1) = A + A' \), and \( g(0, B, 1, -) = B + B' \). Therefore, there are 4 input binary configuration generating hazardous conditions.
Dynamic hazard The hazard generated by asynchronous inputs occurs in circuits after a first level of gates. The propagation hazard needs a logic sum of products (2 or 3 levels of gates). The dynamic hazard is generated by similar causes but manifests in circuits having more than 3 layers of gates. In Figure 8.57 few simple dynamic hazards are shown.

There are complex and not very efficient techniques to avoid dynamic hazard. Usually it is preferred to transform the logic in sums of products (enlarging the circuit) and to apply procedures used to remove propagation hazard (enlarging again the circuit).

Fundamental limits in implementing automata

Because of the problems generated in the real world by the hazardous behaviors some fundamental limitations are applied when an actual automaton works.

1. **The asynchronous input bits can be interpreted only independently in distinct states.** In each clock cycle the automaton interprets the bits used to determine the transition from the current state. If more than one of these bits are asynchronous the reduced dependency coding style must be applied for all of them. But, as we know, this is impossible, only one bit can be considered with reduced dependency. Therefore, in each state no more than one tested bit can be asynchronous. If more than one is asynchronous, then the definition of the automaton must be modified introducing additional states.
2. **Immediate Mealy automaton with asynchronous inputs has no actual implementation** The outputs of an immediate Mealy automaton are combinational conditioned by inputs. Therefore, an asynchronous input will determine **untolerable** asynchronous transitions on some or on all of the outputs.

3. **Delayed Mealy automaton can not be implemented with asynchronous input variables** Even if all the asynchronous inputs are taken into consideration properly when the state code are assigned, the assemble formed by the state register plus the output register works wrong. Indeed, if at least one state bit and one output bit change triggered by an asynchronous input there is the risk that the output register to be loaded with a value unrelated with the value loaded into the state register.

4. **Hazard free Moore automaton with asynchronous inputs has no actual implementation** Asynchronous inputs involve coding with reduced dependency encoding. Hazard free outputs ask coding with minimal variation. But, these two coding styles are incompatible.

### 8.5.3 * Control Automata: the First “Turning Point” *

A very important class of finite automata is the class of **control automata**. A control automaton is embedded in a system using three main connections (see Figure 8.58):

- the $p$-bit input operation \[p-1:0\] selects the control sequence to be executed by the control automaton (it receives the information about “what to do”); it is used to part the ROM in $2^p$ parts, each having the same dimension; in each part a sequence of maximum $2^n$ operation can be “stored” for execution
- the $m$-bit command output, command\[m-1:0\], the control automaton uses to generate “the command” toward the controlled subsystem
- the $n$-bit input flags\[q-1:0\] the control automaton uses to receive information, represented by some independent bits, about “what happens” in the controlled subsystems commanded by the output command\[m-1:0\].

![Control Automaton Diagram](image)

Figure 8.58: **Control Automaton**. The functional definition of control automaton. Control means to issue commands and to receive back signals (flags) characterizing the effect of the command.

The size and the complexity of the control sequence asks the replacement of the PLA with a ROM, at least for the designing and testing stages in implementing the application. The size of the ROM has the magnitude order:

$$S_{\text{ROM}}(n, p, q) \in O(2^{n+p+q}).$$

In order to reduce the ROM's size we start from the actual applications which emphasize two very important facts:
1. The automaton can “store” the information about “what to do” in the state space, i.e., each current state belongs to a path through the state space, started in one initial state given by the code used to specify the operation.

2. In most of the states the automaton tests only one bit from the \( \text{flags}[q-1:0] \) input and if not, a few additional states in the flow-chart solve the problem in most of the cases.

[Diagram of control automaton with labels and connections]

Figure 8.59: **Optimized version of control automata.** The flags received from the controlled system have independent meaning considered in distinct cycles. The flag selected by the code TEST, T, decides from what half of ROM the next state and output will be read.

Starting from these remarks the structure of the control automaton can be modified (see Figure 8.59). Because the sequence is only initialized using the code \( \text{operation}[n-1:0] \), this code is used only for addressing the first command line from ROM in a single state in which \( \text{MOD} = 1 \). For this feature we must add \( n \) EMUXs and a new output to the ROM to generate the signal \( \text{MOD} \). This change allows us to use in a more flexible way the “storing space” of ROM. Because a control sequence can have the dimension very different from the dimension of the other control sequence it is not efficient to allocate fix size part of ROM for each sequence as in we did in the initial solution. The version presented in Figure 8.59 uses for each control sequence only as much of space as needed to store all lines of command.

The second modification refers to the input \( \text{flags}[q-1:0] \). Because the bits associated with this input are tested in different states, \( \text{MUX}_q \) selects in each state the appropriate bit using the \( t \)-bit field TEST. Thus, the \( q - 1 \) bits associated to the input \( \text{flags}[q-1:0] \) are removed from the input of the ROM, adding only \( t \) output bits to ROM. Instead of around \( q \) bits we connect only one, T, to the input of ROM.

This new structure works almost the same as the initial structure but the size of ROM is very strongly minimized. Now the size of the ROM is estimated as being:

\[
S_{\text{ROM}}(2^n).
\]

Working with the control automaton in this new version we will make another remark: *the most part of the sequence generated is organized in a linear sequence*. Therefore, the commands associated to the linear sequences can be stored in ROM at the successive addresses, i.e., the next address for ROM can be obtained incrementing the current address stored in the register \( R \). The structure represented in Figure 8.60 results. What is new in this structure is an increment circuit connected to the output of the state register and a small combinational circuit that transcodes the bits \( M_1, M_0, T \) into \( S_1 \) and \( S_0 \). There are 4 transition modes coded by \( M_1, M_0 \):
8.5. **FINITE AUTOMATA: THE COMPLEX AUTOMATA**

---

**Figure 8.60:** The **simplest Controller with ROM (CROM)**. The Moore form of control automaton is optimized using an incremented circuit (INC) to compute the most frequent next address for ROM.

- **inc**, codd ed by $M_1M_0 = 00$: the next address for ROM results by incrementing the current address; the selection code must be $S_1S_0 = 00$
- **jmp**, codd ed by $M_1M_0 = 01$: the next address for ROM is given by the content of the one field to the output of ROM; the selection code must be $S_1S_0 = 01$
- **cjmp**, codd ed by $M_1M_0 = 10$: if the value of the selected flag, $T$, is 1, then the next address for ROM is given by the content of the one field to the output of ROM, else the next address for ROM results by incrementing the current address; the selection code must be $S_1S_0 = 0T$
- **init**, codd ed by $M_1M_0 = 11$: the next address for ROM is selected by $nMUX_4$ from the initialization input operation; the selection code must be $S_1S_0 = 1$

Results the following logic functions for the transcoder TC: $S_1 = M_1M_0$, $S_0 = M_1T + M_0$.

The output of ROM can be seen as a **microinstruction** defined as follows:

- `<microinstruction>::= <setLabel> <Command> <Mod> <Test> <useLabel>;
- `<command>::= <to be defined when use>;
- `<mod>::= jmp | cjmp | init | inc ;
- `<test>::= <to be defined when use>;
- `<setLabel>::= setLabel(<number>);
- `<useLabel>::= useLabel(<number>);
- `<number>::= 0 | 1 | ... | 9 | <number><number>;

This last version will be called CROM (Controller with ROM) and will be considered, in the present approach, to have enough functional features to be used as controller for the most complex structures described in this book.

**Very important comment!** The previous version of the control automaton’s structure is characterized by two processes:

- the first is the increasing of the structural complexity.
- the second is the decreasing of the dimension and of the complexity of the binary configuration “stored” in ROM.
In this third step both, the size and the complexity of the system grows without any functional improvement. The only effect is reducing the (algorithmic) complexity of ROM’s content.

We are in a very important moment of digital system development, in which the physical complexity starts to compensate the “symbolic” complexity of ROM’s content. Both, circuits and symbols, are structures but there is a big difference between them. The physical structures have simple recursive definitions. The symbolic content of ROM is (almost) random and has no simple definition.

We agree to grow a little the complexity of the physical structure, even the size, in order to create the condition to reduce the effort to set up the complex symbolic content of ROM.

This is the first main “turning point” in the development of digital systems. We have here the first sign about the higher complexity of symbolic structures. Using recursive defined objects the physical structures are maintained at smaller complexity, rather than the symbolic structures, that must assume the complexity of the actual problems to be solved with the digital machines. The previous defined CROM structure is so thought as the content of ROM to be easy designed, easy tested and easy maintained because it is complex. This is the first moment, in our approach, when the symbolic structure has more importance than the physical structure of a digital machine.

**Example 8.30** Let’s revisit the automaton used to control the MAC system. Now, because a more powerful tool is available, the control automaton will perform three functions, multiply, multiply and accumulate, no operation, coded as follows:
mult: op = 01,
macc: op = 11,
noop: op = 00.

![Diagram of CROM](image)

**Figure 8.61:** Using a CROM. A more complex control can be done for Multiply Accumulate System using a CROM instead of a standard finite automaton.

*The CROM circuit is actualized in Figure 8.61 with the word of ROM organized as follows:*
The fields <c1> ... <c5> are one-bit fields taking the value 0 for ".". When nothing is specified, then in the corresponding position is 0. The bit end is used to end the accumulation. If stop = 0 the macc operation does not end, the system waits for a new pairs of numbers to be multiplied and accumulated. The result is sent out only when stop = 1.

Figure 8.62: Control flowchart. The control flowchart for the function macc.
The function \texttt{mult} is defined in the flowchart from Figure 8.43 as a Mealy automaton. Because the CROM automaton is defined as a Moore automaton the code sequence will be defined taking into account the Moore version of the control multiply-accumulate automaton. The function \texttt{macc} is defined in Figure 8.62 as a Moore automaton. The function \texttt{nop} consist in looping in the reset state waiting for a command different from \texttt{nop}. The content of ROM has the following symbolic definition:

\begin{verbatim}
// no operation
setLabel(0) init; // 00000
// multiplication
setLabel(1) nop clear jmp empty setLabel(1);
nop load read inc;
setLabel(2) nop jmp empty setLabel(2);
setLabel(3) jmp done setLabel(3);
setLabel(4) jmp full setLabel(4);
read write jmp setLabel(0);
// multiply and accumulate
setLabel(5) nop clear jmp empty setLabel(5); // q0
setLabel(8) nop load read inc; // q1
setLabel(6) jmp done jmp empty setLabel(6); // q2
setLabel(7) jmp n_done setLabel(7); // q3
read inc; // q4
jmp n_empty setLabel(8); // q5
cjmp stop setLabel(10); // q6
setLabel(9) jmp empty setLabel(9); // q9
jmp loop8; // q10!
setLabel(10) jmp full setLabel(10); // q7
write jmp setLabel(0); // q8
\end{verbatim}

The binary sequence is stored in ROM starting from the address zero with the line labelled as \texttt{setLabel(0)}. The sequence associated to the function \texttt{mult} has 6 lines because a Moore automatons has usually more states when the equivalent Mealy version. For \texttt{macc} function the correspondence with the state are included in commentaries on each line. An additional state (\texttt{q10}) occurs also here, because this version of CROM can not consider jump addresses depending on the tested bits; only one jump address per line is available.

The binary image of the previous code asks codes for the fields acting on the loop. ◊

\textbf{Verilog descriptions for CROM}

The most complex part in defining a CROM unit is the specification of the ROM’s content. There are few versions to be used. One is to provide the bits using a binary file, another is to generate the bits using a Verilog program. Let us start with the first version.

The description of the unit CROM implies the specification of the following parameters used for dimensioning the ROM:

- \texttt{comDim}: the number of bits used to encode the command(s) generated by the control automaton; it depends by the system under control
- \texttt{adrDim}: the number of bits used to encode the address for ROM; it depends on the number of state of the control automaton
- \texttt{testDim}: the number of bits used to select one of the flags coming back from the controlled system; it depends by the functionality performed by the entire system.

The following description refers to the CROM represented in Figure 8.60. It is dimensioned to generate a 5-bit command, to have maximum 32 internal states, and to evolve according to maximum 8 flags (the dimensioning fits with the simple application presented in the previous example). Adjusting these parameters, the same design
can be reused in different projects. Depending on the resulting size of the ROM, its content is specified in various ways. For small sizes the ROM content can be specified by a hand written file of bits, while for big sizes it must be generated automatically starting from a "friendly" definition.

A generic Verilog description of the simple CROM already introduced follows:

```verilog
module crom #(`include "0_parameter.v")(output [comDim - 1:0] command ,
input [addrDim - 1:0] operation,
input [(1 << testDim) - 1:0] flags ,
input reset ,
input clock );

reg [addrDim - 1:0] stateRegister;
wire [comDim + testDim + addrDim + 1:0] romOut ;
wire flag ;
wire [testDim - 1:0] test ;
wire [1:0] mode ;
wire [addrDim - 1:0] nextAddr;
wire [addrDim - 1:0] jumpAddr;

rom rom(.address (stateRegister),
.data (romOut ));

assign {command, test, jumpAddr, mode} = romOut,
flag = flags[test];

mux4 addrSelMux(.out(nextAddr ),
.in0(stateRegister + 1 ),
.in1(jumpAddr ),
.in2(operation ),
.in3(operation ),
.sel({&mode, (mode[1] & flag | mode[0])}));

always @(posedge clock) if (reset) stateRegister <= 0 ;
else stateRegister <= nextAddr;
endmodule
```

The simple uniform part of the previous module consists in two multiplexer, an increment circuit, and a register. The complex part of the module is formed by a very small one (the transcoder) and a big one the ROM.

From the simple only the addrSelMux multiplexor asks for a distinct module. It follows:

```verilog
module mux4 #(`include "0_parameter.v")(out, in0, in1, in2, in3, sel);
input [1:0] sel ;
input [addrDim - 1:0] in0, in1, in2, in3;
output [addrDim - 1:0] out ;
reg [addrDim - 1:0] out ;
always @(in0 or in1 or in2 or in3 or sel)
case(sel)
  2'b00: out = in0;
  2'b01: out = in1;
  2'b10: out = in2;
  2'b11: out = in3;
endcase
```
endmodule

The big complex part has a first version described by the following Verilog module:

```verilog
module rom #(`include "0_parameter.v")

(input [addrDim - 1:0] address,
output [comDim + testDim + addrDim + 1:0] data );
reg [comDim + testDim + addrDim + 1:0] mem [0:(1'<< addrDim) - 1];

initial $readmemb("0_romContent.v", mem); // the fix content of the memory

assign data = mem[address]; // it is a read only memory
endmodule
```

The file `0_parameter.v` defines the dimensions used in the project crom. It must be placed in the same folder with the rest of the files defining the project. For our example its content is:

```plaintext
parameter comDim = 5,
addrDim = 5,
testDim = 3
```

The initial line loads in background, in a transparent mode, the memory module mem. The module rom does not have explicit writing capabilities, behaving like a “read only” device. The synthesis tools are able to infer from the previous description that it is about a ROM combinational circuit.

The content of the file `0_romContent.v` is filled up according to the micro-code generated in Example 7.6. Obviously, after the first 4 line the our drive to continue is completely lost.

```
/* 00 */ 00000_000_00000_11
/* 01 */ 11000_011_00001_10
/* 02 */ 10100_000_00000_00
/* 03 */ 10000_011_00011_10
//
/* 30 */ 00000_000_00000_00
/* 31 */ 00000_000_00000_00
```

Obviously, after filling up the first 4 lines our internal drive to continue is completely lost. The full solution asks for 270 bits free of error bits. Another way to generate them must be found!

**Binary code generator**

Instead of defining and writing bit by bit the content of the ROM, using a hand written file (in our example `0_romContent.v`), is easiest to design a Verilog description for a “machine” which takes a file containing lines of microinstructions and translate it into the corresponding binary representation. Then, in the module rom the line initial ...; must be substituted with the following line:

```verilog
`include "codeGenerator.v" // generates ROM's content using to 'theDefinition.v'
```

which will act by including the the description of a loading mechanism for the memory mem. The code generating machine is a program which has as input a file describing the behavior of the automaton. Considering the same example of the control automaton for MAC system, the file `theDefinition` is a possible input for our code generator. It has the following form:

```plaintext
// MAC control automaton
setLabel(0); init;
// multiplication
```
8.5. FINITE AUTOMATA: THE COMPLEX AUTOMATA

setLabel(1); nop; clear; cjmp; empty; useLabel(1);
setLabel(2); nop; load; read; inc; useLabel(2);
setLabel(3); cjmp; done; useLabel(3);
setLabel(4); cjmp; full; useLabel(4);
  read; write; jmp; useLabel(0);
// multiply & accumulate
setLabel(5); nop; clear; cjmp; empty; useLabel(5);
setLabel(8); nop; load; read; inc;
setLabel(7); cjmp; notDone; useLabel(7);
  read; inc;
  cjmp; notEmpty; useLabel(8);
  cjmp; stop; useLabel(10);
setLabel(9); cjmp; empty; useLabel(9);
  jmp; useLabel(8);
setLabel(10); cjmp; full; useLabel(10);
  write; jmp; useLabel(0);

The theDefinition file consist in a stream of Verilog tasks. The execution of these tasks generate the ROM’s content.

The file codeGenerator.v “understand” and use the file theDefinition, whose content follows:

// Generate the binary content of the ROM
reg nopReg ;
reg clearReg ;
reg loadReg ;
reg readReg ;
reg writeReg ;
reg [1:0] mode ;
reg [2:0] test ;
reg [4:0] address ;
reg [4:0] counter ;
reg [4:0] labelTab[0:31];

task endLine;
begin
  mem[counter] = 
    {nopReg, clearReg, loadReg, readReg, writeReg, mode, test, address};
  nopReg = 1’b0;
  clearReg = 1’b0;
  loadReg = 1’b0;
  readReg = 1’b0;
  writeReg = 1’b0;
  counter = counter + 1;
end
endtask

// sets labelTab in the first pass associating 'counter' with 'labelIndex'
// task setLabel; input [4:0] labelIndex; labelTab[labelIndex] = counter; endtask
// uses the content of labelTab in the second pass
// task useLabel; input [4:0] labelIndex; begin address = labelTab[labelIndex];
//                  endLine;
end
```verilog
// external commands
task nop ; nopReg = 1'b1; endtask
task clear; clearReg = 1'b1; endtask
task load ; loadReg = 1'b1; endtask
task read ; readReg = 1'b1; endtask
task write; writeReg = 1'b1; endtask

// transition mode
task inc ; begin mode = 2'b00; endLine; end endtask
task jmp ; mode = 2'b01; endtask
task cjmp; mode = 2'b10; endtask
task init; begin mode = 2'b11; endLine; end endtask

// flag selection
task empty ; test = 3'b000; endtask
task full ; test = 3'b001; endtask
task done ; test = 3'b010; endtask
task stop ; test = 3'b011; endtask
task notDone; test = 3'b100; endtask
task notEmpt; test = 3'b101; endtask

initial begin counter = 0;
  nopReg = 0;
  clearReg = 0;
  loadReg = 0;
  readReg = 0;
  writeReg = 0;
  `include "theDefinition.v"; // first pass
  `include "theDefinition.v"; // second pass
end
```

The file `theDefinition` is included twice because if a label is used before it is defined, only at the second pass in the memory `labelTab` the right value of a label will be found when the task `useLabel` is executed.

### 8.6 Automata vs. Combinational Circuits

As we saw, both combinational circuits (0-OS) and automata (2-OS) execute digital functions. Indeed, there are combinational circuits performing addition or multiplication, but there are also sequential circuits performing the same functions. What is the correlation between a gates network and an automaton executing the same function? What are the conditions in which we can transform a combinational circuit in an automaton or conversely? The answer to this question will be given in this last section.

Let be a Mealy automaton, his two CLCs (LOOP CLC and OUT CLC), the initial state of the automaton, \( q(t_0) \) and the input sequence for the first \( n \) clock cycle: \( x(t_0), \ldots, x(t_{n-1}) \). The combinational circuit that generates the corresponding output sequence \( y(t_0), \ldots, y(t_{n-1}) \) is represented in Figure 8.63. Indeed, the first pair LOOP CLC, OUT CLC computes the first output, \( y(t_0) \), and the next state, \( q(t_1) \) to be used by the second pair of CLCs to compute the second output and the next state, and so on.

**Example 8.31** The ripple carry adder (Figure 6.18) has as correspondent automaton the adder automaton from the serial adder (Figure 8.6).  

Should be very interesting to see how a complex problem having associated a finite automaton can be solved starting from a combinational circuit and reducing it to a finite automaton. Let us revisit in the next example the problem of recognizing strings from the set \( 1^a 0^b \), for \( a, b > 0 \).
8.6. AUTOMATA VS. COMBINATIONAL CIRCUITS

Figure 8.63: Converting an automata into a combinational circuit. The conversion rule from the finite (Mealy) automaton into a combinational logic circuit means to use a pair of circuits (LOOP CLC, OUTPUT CLC) for each clock cycle. The time dimension is transformed in space dimension.

Example 8.32 The universal combinational circuit (see 2.3.1) is used to recognize all the strings having the form:

\[ x_0, x_1, \ldots, x_i, \ldots, x_{n-1} \in 1^a0^b \]

for \( a, b > 0 \), and \( a + b = n \). The function performed by the circuit will be:

\[ f(x_{n-1}, \ldots, x_0) \]

which takes value 1 for the following inputs:

\[
\begin{align*}
  x_{n-1}, \ldots, x_0 &= 0000 \ldots 01 \\
  x_{n-1}, \ldots, x_0 &= 000 \ldots 011 \\
  x_{n-1}, \ldots, x_0 &= 00 \ldots 0111 \\
  \cdots \\
  x_{n-1}, \ldots, x_0 &= 00011 \ldots 1 \\
  x_{n-1}, \ldots, x_0 &= 0011 \ldots 11 \\
  x_{n-1}, \ldots, x_0 &= 011 \ldots 111
\end{align*}
\]

Any function \( f(x_{n-1}, \ldots, x_0) \) of \( n \) variables can be expressed using certain minterms from the set of \( 2^n \) minterms of \( n \) variables. Our functions uses only \( n - 2 \) minterms from the total number of \( 2^n \). They are:

\[ m_{2^i-1} \]

for \( i = 1, \ldots (n - 1) \), i.e., the functions takes the value 1 for \( m_1 \) or \( m_3 \) or \( m_7 \) or \( m_{15} \) or \( \ldots \)

Figure 8.64 represents the universal circuits receiving as "program" the string:

\[ \ldots 00100000010001010 \]
Figure 8.64: The Universal Circuit “programmed” to recognize $1^a0^b$. A full tree of $2^n$ EMUXs are used to recognize the strings belonging to $1^a0^b$. The “program” is applied on the selected inputs of the first level of EMUXs.

Figure 8.65: Minimizing the Universal Circuit “programmed to recognize $1^a0^b$. a. The first step of minimizing the full tree of $2^n - 1$ EMUXs to a tree containing $0.5n(n+1)$ EMUXs. Each EMUX selecting between 0 and 0 is substituted with a connection to 0. b. The minimal combinational network of EMUXs obtained removing the duplicated circuits. The resulting network is a linear stream of identical CLCs.
where 1s corresponds to minterms having the value 1, and 0s to the minterms having the value 0.

Initially the size of the resulting circuit is too big. For an n-bit input string from $x_0$ to $x_{n-1}$ the circuits contains $2^n - 1$ elementary multiplexors. But, a lot of EMUXs have applied 0 on both selected inputs. They will generate 0 on their outputs. If the multiplexors generating 0 are removed and substituted with connections to 0, then the resulting circuit containing only $n(n - 1)/2$ EMUXs is represented in Figure 8.65a.

The circuit can be more reduced if we take into account that some of them are identical. Indeed, on the first line all EMUXs are identical an the third (from left to right) can do the “job” of the first tree circuits. Therefore, the output of the third circuit from the first line will be connected to the input of all the circuits from the second line. Similarly, on the second line we will maintain only two EMUXs, and so on on each line. Results the circuit from Figure 8.65b containing $(2n - 1)$ EMUXs.

This last form consists in a serial composition made using the same combinational circuit: an EMUX and an 2-input AND (the EMUX with the input 0 connected to 0). Each stage of the circuit receives one input value starting with $x_0$. The initial circuit receives on the selected inputs a fix binary configuration (see Figure 8.65b). It can be considered as the initial state of the automaton. Now we are in the position to transform the circuit in a finite half-automaton connecting the emphasized module in the loop with a 2-bit state register (see Figure 8.66a).

The resulting half-automaton can be compared with the half-automaton from Figure 8.68, reproduced in Figure 8.66b. Not-surprisingly they are identical. ○

![Figure 8.66: From a big and simple CLC to a small and complex finite automata. a. The resulting half-automaton obtained collapsing the stream of identical circuits. b. Minimizing the structure of the two EMUXs results a circuit identical with the solution provided in Figure 8.68 for the same problem.](image)

To transform a combinational circuit in a (finite) automaton the associated tree (or trees) of EMUXs must degenerate into a linear graph of identical modules. An interesting problem is: how many of “programs”, $P = m_{p-1}, m_{p-2}, \ldots, m_0$, applied as “leaves” of Universal Circuit allows the tree of EMUXs to be reduced to a linear graph of identical modules?

### 8.7 The Circuit Complexity of a Binary String

Greg Chaitin taught us that simplicity means the possibility to compress. He expressed the complexity of a binary string as being the length of the shortest program used to generate that string. An alternative form to express the complexity of a binary string is to use the size of the smallest circuit used to generate it.

**Definition 8.20** The circuit complexity of a binary string $P$ of length $p$, $CC_P(p)$, is the size of the minimized circuit used to generate it. ○
**Definition 8.21** The universal circuit used to generate any \( p \)-bit string, \( PU-Generator \), consists in a \( nU-Circuit \) programmed with the string to be generated and triggered by a resetable counter (see Figure 8.67).

The counter, starting from zero, selects to the output the bits of the “program” one by one starting with \( m_0 \).

According to the actual content of the “program” \( P = m_{p-1} \ldots m_0 \) the \( nU-Circuit \) can be reduced to a minimal size using techniques previously described in the section 2.3. The minimal size of the counter is in \( O(\log p) \) (the “first” proposal for an actual value is \( 11(1 + \log_2 p) + 5 \)). Therefore, the minimal size of \( PU-Generator \), used to generate an actual string of \( p \) bits is the very precisely defined number \( CC_P(p) \).

**Example 8.33** Let us compute the circuit size of the following 16-bit strings:

\[
P_1 = 0000000000000000 \quad P_2 = 1111111111111111 \quad P_3 = 0101000001010000 \quad P_4 = 0110100110110001
\]

For both, \( P_1 \) and \( P_2 \) the \( nU-Circuit \) is reduced to circuits containing no gates. Therefore, \( CC(P_1) = CC(P_2) = 11(1 + \log_2 16) + 5 + 0 = 60 \).

For \( P_3 \), applying the removing rules the first level of EMUXs in \( nU-Circuit \)s is removed and to the inputs of the second level the following string is applied:

\[
x_0',x_0,0,0,x_0',x_0,0,0
\]

We continue applying removing and reducing rules. Results the inputs of the third level:

\[
x_0',x_0,0,0,0
\]

The last level is removed because its inputs are identic. The resulting circuit is: \( x_0'x_2 \). It has the size 3. Therefore \( CC(P_3) = 60 + 3 = 63 \).

For \( P_4 \), applying the removing rules results the following string for the second level of EMUXs:

\[
x_0',x_0,x_0',x_0,1,0,x_0'
\]

No removing or reducing rule apply for the next level. Therefore, the size of the resulting circuit is: \( CC(P_4) = 1 + 7S_{EMUX} + 88 = 103 \).
8.8. CONCLUDING ABOUT AUTOMATA

The main problem in computing the circuit complexity of a string is to find the minimal form of a Boolean function. Fortunately, there are rigorous formal procedures to minimize logic functions (see Appendix C.4 for some of them). (Important note: the entire structure of pU-Generator can be designed composing and closing loops in a structure containing only elementary multiplexors and inverters. In the language of the partial recursive functions these circuits perform the elementary selection and the elementary increment. “Programming” uses only the function zero and the elementary increment. No restrictions imposed by primitive recursiveness or minimalization are applied!)

An important problem rises: how many of the n-bit variable function are simple? The answer comes from the next theorem.

Theorem 8.4 The weight, w, of Turing-computable functions, of n binary variables, in the set of the formal functions decreases twice exponentially with n.

Proof Let be a given n. The number of formal n-input function is \( N = 2^{2^n} \), because the definition are expressed with \( 2^n \) bits. Some of this functions are Turing-computable. Let be these functions defined by the compressed m-bit strings. The value of m depends on the actual function, but is realized the condition that \( \max(m) < 2^n \) and m does not depends by n. Each compressed form of m bits corresponds only to one \( 2^n \)-bit uncompressed form. Thus, the ratio between the Turing-computable function of and the formal function, both of n variables, is smaller than

\[
\max(w) = 2^{-\left(2^n - \max(m)\right)}.
\]

And, because \( \max(m) \) does not depends by n, the ratio has the same form for no matter how big becomes n. Results:

\[
\max(w) = \text{const}/2^n.
\]

A big question arises: how could be combinational circuits useful with this huge ratio between complex circuits and simple circuits? An answer could be: potentially this ratio is very high, but actually, in the real world of problems this ratio is very small. It is small because we do not need to compute too many complex functions. Our mind is usually attracted by simple functions in a strange manner for which we do not have (yet?) a simple explanation.

The Turing machine is limited to perform only partial recursive functions (see Chapter 9 in this book). The halting problem is an example of a problem that has no solutions on a Turing machine (see subsection 9.3.5?? in this book). Circuits are more powerful but they are not so easy “programmed” as the Turing Machine, and the related systems. We are in a paradoxical situation: the circuit does not need algorithms and Turing Machine is limited only to the problems that have an algorithm. But without algorithms many solutions exist and we do not know the way to find them. The complexity of the way to find of a solution becomes more and more important.

The working hypothesis will be that at the level of combinational (without autonomy) circuits the segregation between simple circuits and complex programs is not productive. In most of cases the digital system grows toward higher orders where the autonomy of the structures allow an efficient segregation between simple and complex.

8.8 Concluding about automata

A new step is made in this chapter in order to increase the autonomous behavior of digital systems. The second loop looks justified by new useful behaviors.

Synchronous automata need non-transparent state registers The first loop, closed for gain the storing function, is applied carefully to obtain stable circuits. Tough restrictions can be applied (even number of inverting levels on the loop) because of the functional simplicity. The functional complexity of automata rejects any functional restrictions applied for the transfer function associated to loop circuits. The
unstable behavior is avoided using non-transparent memories (registers) to store the state. Thus, the state switches synchronized by clock. The output switches synchronously for delayed version of the implementation. The output is asynchronous for the immediate versions.

The second loop means the behavior’s autonomy  Using the first loop to store the state and the second to compute any transition function, a half-automaton is able to evolve in the state space. The evolution depends by state and by input. The state dependence allows an evolution even if the input is constant. Therefore, the automaton manifests its autonomy being able to behave, evolving in the state space, under constant input. An automaton can be used as “pure” generator of more or less complex sequence of binary configuration. the complexity of the sequence depends by the complexity of the state transition function. A simple function on the second loop determine a simple behavior (a simple increment circuit on the second loop transforms a register in a counter which generate the simple sequence of numbers in the strict increasing order).

Simple automata can have \( n \) states  When we say \( n \) states, this means \( n \) can be very big, is not limited by our ability to define the automaton, it is limited only by the possibility to implement it using the accessible technologies. A simple automata can have \( n \) states because the state register contains \( \log n \) flip-flops, and its second loop contains a simple (constant defined) circuit having the size in \( O(f(\log n)) \). The simple automata can be big because they can be specified easy, and they can be generated automatically using the current software tools.

Complex automata have only finite number of states  Finite number of states means: a number of states unrelated with the length (theoretically accepted as infinite) of the input sequence, i.e., the number of states is constant. The definition must describe the specific behavior of the automaton in each state. Therefore, the definition is complex having the size (at least) linearly related with the number of states. Complex automata must be small because they suppose combinational loops closed through complex circuits having the description in the same magnitude order with their size.

Control automata suggest the third loop  Control automata evolve according to their state and they take into account the signals received from the controlled system. Because the controlled system receives commands from the same control automaton a third loop prefigures. Usually finite automata are used as control automata. Only the simple automata are involved directly in processing data.

An important final question: adding new loops the functional power of digital systems is expanded or only helpful features are added? And, if indeed new helpful features occur, who is helped by these additional features?

8.9 Problems

Problem 8.1 Draw the JK flip-flop structure (see Figure 8.5) at the gate level. Analyze the set-up time related to both edges of the clock.

---

Asynchronous automata are possible but their design is restricted by to complex additional criteria. Therefore, asynchronous design is avoided until stronger reason will force us to use it.
Problem 8.2 Design a JK FF using a D flip-flop by closing the appropriate combinational loop. Compare the set-up time of this implementation with the set-up time of the version resulting in the previous problem.

Problem 8.3 Design the sequential version for the circuit which computes the n-bit AND prefixes. Follow the approach used to design the serial n-bit adder (see Figure 8.6).

Problem 8.4 Write the Verilog structural description for the universal 2-input, 2-state programmable automaton.

Problem 8.5 Draw at the gate level the universal 2-input, 2-state programmable automaton.

Problem 8.6 Use the universal 2-input, 2-state automaton to implement the following circuits:

- n-bit serial adder
- n-bit serial subtractor
- n-bit serial comparator for equality
- n-bit serial comparator for inequality
- n-bit serial parity generator (returns 1 if odd)

Problem 8.7 Define the synchronous n-bit counter as a simple n-bit Increment Automaton.

Problem 8.8 Design a Verilog tester for the resetable synchronous counter from Example 4.1.

Problem 8.9 Evaluate the size and the speed of the counter defined in Example 4.1.

Problem 8.10 Improve the speed of the counter designed in Example 4.1 designing an improved version for the module and_prefix.

Problem 8.11 Design a reversible counter defined as follows:

```verilog
module smartest_counter #(parameter n = 16)
    ( output [n-1:0] out ,
      input [n-1:0] in , // preset value
      input reset , // reset counter to zero
      input load , // load counter with ‘in’
      input down , // counts down if (count)
      input count , // counts up or down
      input clock );
endmodule
```

Problem 8.12 Simulate a 3-bit counter with different delay on its outputs. It is the case in real world because the flop-flops can not be identical and their load could be different. Use it as input for a three input decoder implemented in two versions. One without delays and another assigning delays to the inverters and the the gates used to implement the decoder. Visualize the outputs of the decoder in both cases and interpret what you will find.

Solution:
module dec_spyke;
    reg    clock,
           enable;
    reg [2:0] counter;
    wire    out0, out1, out2, out3, out4, out5, out6, out7;

    initial begin    clock = 0;
                      enable = 1;
                      counter = 0;
                      forever #20 clock = ~clock;
    end

    initial #400 $stop;

    always @(posedge clock)
    begin
        counter[0] <= #3 ~counter[0];
        if (counter[0])     counter[1] <= #4 ~counter[1];
        if (&counter[1:0])  counter[2] <= #5 ~counter[2];
    end

    dmux dmux( .out0 (out0) ,
               .out1 (out1) ,
               .out2 (out2) ,
               .out3 (out3) ,
               .out4 (out4) ,
               .out5 (out5) ,
               .out6 (out6) ,
               .out7 (out7) ,
               .in (counter) ,
               .enable (enable) );

    initial $vw_dumpvars;
endmodule

module dmux(out0, out1, out2, out3, out4, out5, out6, out7, in, enable);

    input enable;
    input [2:0] in;
    output out0, out1, out2, out3, out4, out5, out6, out7;

    assign {out0, out1, out2, out3, out4, out5, out6, out7} = 1'b1 << in;
    /*
    not    #1 not0(nin2, in[2]);
    not    #1 not1(nin1, in[1]);
    not    #1 not2(nin0, in[0]);
    */
8.9. PROBLEMS

Problem 8.13 Justify the reason for which the LIFO circuit works properly without a reset input, i.e., the initial state of the address counter does not matter.

Problem 8.14 How behaves simple_stack.

Problem 8.15 Design a LIFO memory using a synchronous RAM (SRAM) instead of an asynchronous one as in the embodiment represented in Figure 11.1.

Problem 8.16 Some applications ask the access to the last two data stored into the LIFO. Call them tos, for the last pushed data, and prev_tos for the previously pushed data. Both accessed data can be popped from stack. Double push is allowed. The accessed data can be rearranged swapping their position. Both, tos and prev_tos can be pushed again in the top of stack. Design such a LIFO defined as follows:

```verilog
module two_head_lifo( output [31:0] tos ,
                     output [31:0] prev_tos ,
                     input [31:0] in ,
                     input [31:0] second_in ,
                     input [2:0] com , // the operation
                     input clock );
  // the semantics of 'com'
  parameter nop = 3'b000, // no operation
              swap = 3'b001, // swap the first two
             pop  = 3'b010, // pop tos
            pop2  = 3'b011, // pop tos and prev_tos
            push  = 3'b100, // push in as new tos
           push2  = 3'b101, // push 'in' and 'second_in'
          push_tos  = 3'b110, // push 'tos' (double tos)
         push_prev  = 3'b111; // push 'prev_tos'
endmodule
```

Problem 8.17 Write the Verilog description of the FIFO memory represented in Figure 8.17.

Problem 8.18 Redesign the FIFO memory represented in Figure 8.17 using a synchronous RAM (SRAM) instead of the asynchronous RAM.
Problem 8.19  There are application asking for a warning signal before the FIFO memory is full or empty. Sometimes full and empty come to late for the system using the FIFO memory. For example, no more than 3 write operation are allowed, or no more than 7 read operation are allowed are very useful in systems designed using pipeline techniques. The threshold for this warning signals is good to be programmable. Design a 256 8-bit entries FIFO with warnings activated using a programmable threshold. The interconnection of this design are:

```verilog
module th_fifo(output [7:0] out ,
    input [7:0] in ,
    input [3:0] write_th, // write threshold
    input [3:0] read_th , // read threshold
    input write ,
    input read ,
    output w_warn , // write warning
    output r_warn , // read warning
    output full ,
    output empty ,
    input reset ,
    input clock );
endmodule
```

Problem 8.20  A synchronous FIFO memory is written or read using the same clock signal. There are many applications which use a FIFO to interconnect two subsystems working with different clock signals. In this cases the FIFO memory has an additional role: to cross from the clock domain clock_in into another clock domain, clock_out. Design an asynchronous FIFO using a synchronous RAM.

Problem 8.21  A serial memory implements the data structure of a fix length circular list. The first location is accessed, for write or read operation, activating the input init. Each read or write operation move the access point one position right. Design an 8-bit word serial memory using a synchronous RAM as follows:

```verilog
module serial_memory( output [7:0] out ,
    input [7:0] in ,
    input init ,
    input write ,
    input read ,
    input clock );
endmodule
```

Problem 8.22  A list memory is a circuit in which a list can be constructed by insert, can be accessed by read_forward, read_back, and modified by insert, delete. Design such a circuit using two LIFOs.

Problem 8.23  Design a sequential multiplier using as combinational resources only an adder, a multiplexors.

Problem 8.24  Write the behavioral and the structural Verilog description for the MAC circuit represented in Figure 8.19. Test it using a special test module.

Problem 8.25  Redesign the MAC circuit represented in Figure 8.19 adding pipeline register(s) to improve the execution time. Evaluate the resulting speed performance using the parameters form Appendix E.
Problem 8.26  How many 2-bit code assignment for the half-automaton from Example 4.2 exist? Revisit the implementation of the half-automaton for four of them different from the one already used. Compare the resulting circuits and try to explain the differences.

Problem 8.27  Ad to the definition of the half-automaton from Example 4.2 the output circuits for: (1) error, a bit indicating the detection of an incorrectly formed string, (2) ack, another bit indicating the acknowledge of a well formed string.

Problem 8.28  Multiplier control automaton can be defined testing more than one input variable in some states. The number of states will be reduced and the behavior of the entire system will change. Design this version of the multiply automaton and compare it with the circuit resulted in Example 4.3. Reevaluate also the execution time for the multiply operation.

Problem 8.29  Revisit the system described in Example 4.3 and design the finite automaton for multiply and accumulate (MACC) function. The system perform MACC until the input FIFO is empty and end = 1.

Problem 8.30  Design the structure of TC in the CROM defined in 4.4.3 (see Figure 8.60). Define the codes associated to the four modes of transition (jmp, cjmp, init, inc) so as to minimize the number of gates.

Problem 8.31  Design an easy to actualize Verilog description for the CROM unit represented in Figure 8.60.

Problem 8.32  Generate the binary code for the ROM described using the symbolic definition in Example 4.4.

Problem 8.33  Design a fast multiplier converting a sequential multiplier into a combinational circuit.

Problem 8.34  Let be the finite automaton defined in Figure 8.68. Do the following:

Figure 8.68:
1. assign the state codes in two versions:
   (a) according priority to the reduce dependency coding style
   (b) according priority to the minimal variation coding style
2. implement the finite automaton in the resulting two versions by:
   - drawing the transition VK diagrams
   - extracting the logic functions for $Q_2^+, Q_1^+, Q_0^+, Y_2, Y_1, Y_0$
   - drawing the logic schematic of the resulting automaton

Problem 8.35 Describe in Verilog the automaton defined in Problem 8.34 and simulate it.

8.10 Projects

Project 8.1 Finalize Project 1.2 using the knowledge acquired about the combinational and sequential structures in this chapter and in the previous two.

Project 8.2 The idea of simple FIFO presented in this chapter can be used to design an actual block having the following additional features:

- fully buffered inputs and outputs
- programmable thresholds for generating the empty and full signals
- asynchronous clock signals for input and for output (the design must take into consideration that the two clocks – clockIn, clockOut – are considered completely asynchronous)
- the read or write commands are executed only if the it is possible (reads only if not-empty, or writes only if not-full).

The module header is the following:

```verilog
module asyncFIFO #(`include "fifoParameters.v")
(  output reg [n-1:0] out ,
  output reg   empty ,
  output reg   full ,
  input       [n-1:0] in  ,
  input       write  ,
  input       read   ,
  input       [m-1:0] inTh  , // input threshold  
  input       [m-1:0] outTh , // output threshold
  input       reset   ,
  input       clockIn ,
  input       clockOut);
  // ...
endmodule
```

The file fifoParameters.v has the content:
Project 8.3 Design a stack execution unit with a 32-bit ALU. The stack is 16-level depth (stack0, stack1, ... stack15) with stack0 assigned as the top of stack. ALU has the following functions:

- **add:** addition
  \[
  \{\text{stack0, stack1, stack2, ...}\} \leftarrow \{(\text{stack0} + \text{stack1}), \text{stack2}, \text{stack3}, \ldots\}
  \]

- **sub:** subtract
  \[
  \{\text{stack0, stack1, stack2, ...}\} \leftarrow \{(\text{stack0} - \text{stack1}), \text{stack2}, \text{stack3}, \ldots\}
  \]

- **inc:** increment
  \[
  \{\text{stack0, stack1, stack2, ...}\} \leftarrow \{(\text{stack0} + 1), \text{stack1}, \text{stack2}, \ldots\}
  \]

- **dec:** decrement
  \[
  \{\text{stack0, stack1, stack2, ...}\} \leftarrow \{(\text{stack0} - 1), \text{stack1}, \text{stack2}, \ldots\}
  \]

- **and:** bitwise AND
  \[
  \{\text{stack0, stack1, stack2, ...}\} \leftarrow \{(\text{stack0} \& \text{stack1}), \text{stack2}, \text{stack3}, \ldots\}
  \]

- **or:** bitwise OR
  \[
  \{\text{stack0, stack1, stack2, ...}\} \leftarrow \{(\text{stack0} | \text{stack1}), \text{stack2}, \text{stack3}, \ldots\}
  \]

- **xor:** bitwise XOR
  \[
  \{\text{stack0, stack1, stack2, ...}\} \leftarrow \{(\text{stack0} \oplus \text{stack1}), \text{stack2}, \text{stack3}, \ldots\}
  \]

- **not:** bitwise NOT
  \[
  \{\text{stack0, stack1, stack2, ...}\} \leftarrow \{(\neg \text{stack0}), \text{stack1}, \text{stack2}, \ldots\}
  \]

- **over:**
  \[
  \{\text{stack0, stack1, stack2, ...}\} \leftarrow \{\text{stack1, stack0, stack1, stack2, ...}\}
  \]

- **dup:** duplicate
  \[
  \{\text{stack0, stack1, stack2, ...}\} \leftarrow \{\text{stack0, stack0, stack1, stack2, ...}\}
  \]

- **rightShift:** right shift one position (integer division)
  \[
  \{\text{stack0, stack1, ...}\} \leftarrow \{\{1'b0, \text{stack0}[31:1]\}\}, \text{stack1, ...}\}
  \]

- **arithShift:** arithmetic right shift one position
  \[
  \{\text{stack0, stack1, ...}\} \leftarrow \{\{\text{stack0}[31], \text{stack0}[31:1]\}\}, \text{stack1, ...}\}
  \]

- **get:** push dataIn in top of stack
  \[
  \{\text{stack0, stack1, stack2, ...}\} \leftarrow \{\text{dataIn, stack0, stack1, ...}\}.
  \]

- **acc:** accumulate dataIn
  \[
  \{\text{stack0, stack1, stack2, ...}\} \leftarrow \{(\text{stack0} + \text{dataIn}), \text{stack1}, \text{stack2}, \ldots\}.
  \]

- **swp:** swap the last two recordings in stack
  \[
  \{\text{stack0, stack1, stack2, ...}\} \leftarrow \{\text{stack1}, \text{stack0, stack2, ...}\}
  \]
• **nop**: no operation
  
  \{\text{stack0}, \text{stack1}, \text{stack2}, \ldots\} \leftarrow \{\text{stack0}, \text{stack1}, \text{stack2}, \ldots\}.

*All the register buffered external connections are the following:*

- **dataIn[31:0]**: data input provided by the external subsystem
- **dataOut[31:0]**: data output sent from the top of stack to the external subsystem
- **aluCom[3:0]**: command code executed by the unit
- **carryIn**: carry input
- **carryOut**: carry output
- **eqFlag**: is one if \((\text{stack0} == \text{stack1})\)
- **ltFlag**: is one if \((\text{stack0} < \text{stack1})\)
- **zeroFlag**: is one if \((\text{stack0} == 0)\)

*Project 8.4*
Chapter 9

PROCESSORS:
Third order, 3-loop digital systems

In the previous chapter
the circuits having an autonomous behavior were introduced pointing on

- how the increased autonomy adds new functional features in digital systems
- the distinction between finite automata and uniform automata
- the segregation mechanism used to reduce the complexity

In this chapter
the third order, three-loop systems are studied presenting

- how a “smart register” can reduce the complexity of a finite automaton
- how an additional memory helps for designing easy controllable systems
- how the general processing functions can be performed loop connecting two appropriate automata forming a processor

In the next chapter
the fourth order, four-loop systems are suggested with emphasis on

- the four types of loops used for generating different kind of computational structures
- the strongest segregation which occurs between the simple circuits and the complex programs
The soft overcomes the hard in the world as a gentle rider controls a galloping horse.

Lao Tzu

The third loop allows the softness of symbols to act imposing the system’s function.

In order to add more autonomy in digital systems the third loop must be closed. Thus, new effects of the autonomy are used in order to reduce the complexity of the system. One of them will allow us to reduce the apparent complexity of an automaton, another, to reduce the complexity of the sequence of commands, but, the main form of manifesting of this third loop will be the control process.

![Diagram of 3-OS machines]

Figure 9.1: The three types of 3-OS machines. a. The third loop is closed through a combinational circuit resulting less complex, sometimes smaller, finite automaton. b. The third loop is closed through memories allowing a simplest control. c. The third loop is closed through another automaton resulting the Processor: the most complex and powerful circuit.

The third loop can be closed in three manners, using the three types of circuits presented in the previous chapters.

- The first 3-OS type system is a system having the third loop closed through a combinational circuit, i.e., over an automaton or a network of automata the loop is closed through a 0-OS (see Figure 9.1a).

- The second type (see Figure 9.1b) has on the loop a memory circuit (1-OS).

---

1Quote from *Tao Te King* of Lao Tzu translated by Brian Browne Walker.
9.1 Implementing finite automata with "intelligent registers"

The automaton function rises at the second order level, but this function can be better implemented using the facilities offered by the systems having a higher order. Thus, in this section we resume a previous example using the feature offered by 3-OS. The main effect of these new approaches: the ratio between the simple circuits and the complex circuits grows, without spectacular changes in the size of circuits. The main conclusion of this section: more autonomy means less complexity.

9.1.1 Automata with JK “registers”

In the first example we will substitute the state register with a more autonomous device: a “register” made by JK flip-flops. The “JK register” is not a register, it is a network of parallel connected simple automata. We shall prove that, using this more complicated flip-flop, the random part of the system will be reduced and in most of big sized cases the entire size of the system could be also reduced. Thus, both the size and the complexity diminishes when we work with autonomous (“smart”) components.

But let’s start to disclose the promised magic method which, using flip-flops having two inputs instead of one, offers a minimized solution for the combinational circuit performing the loop’s function $f$. The main step is to offer a simple rule to substitute a D flip-flop with a JK flip-flop in the structure of the automaton.

The JK flip-flop has more autonomy than the D flip-flop. The first is an automaton and the second is only a storage element used to delay. The JK flip-flop has one more loop than the D flip-flop. Therefore, for switching from a state to another the input signals of a JK flip-flop accepts more “ambiguity” than the signal to the input of a D flip-flop. The JK flip-flop transition can be commanded as follows:

- for $0 \rightarrow 0$ transition, JK can be $00$ or $01$, i.e., JK=$0–$ ("–" means “don’t care” value)
- for $0 \rightarrow 1$ transition, JK can be $11$ or $10$, i.e., JK=$1–$
- for $1 \rightarrow 0$ transition, JK can be $11$ or $01$, i.e., JK=$–1$
- for $1 \rightarrow 1$ transition, JK can be $00$ or $10$, i.e., JK=$–0$

From the previous rule results the following rule:

- for $0 \rightarrow A$, JK=$A–$
- for $1 \rightarrow A$, JK=$–A'$.

Using these rules, each transition diagram for $Q^+_i$ can be translated in two transition diagrams for $J_i$ and $K_i$. Results: twice numbers of equations. But surprisingly, the entire size of the random circuit which computes the state transition will diminish.
Figure 9.2: Translating D transition diagrams in the corresponding JK transition diagrams. The transition VK diagrams for the JK implementation of the finite half-automaton used to recognize binary string belonging to the $1^n0^m$ set of strings.

**Example 9.1** The half-automaton designed in Example 8.4 is reconsidered in order to be designed using JK flip-flops instead of D flip-flops. The transition map from Figure 8.29 (reproduces in Figure 9.2a) is translated in JK transition maps in Figure 9.2b. The resulting circuit is represented in Figure 9.2c.

The size of the random circuit which computes the state transition function is now smaller (from the size 8 for D–FF to size 5 for JK–FF). The increased autonomy of the now used flip-flops allows a smaller "effort" for the same functionality.

**Example 9.2** Let’s revisit also Example 8.5. Applying the transformation rules results the VK diagrams from Figure 9.3 from which we extract:

Figure 9.3: Translating D transition diagrams in the corresponding JK transition diagrams. The transition VK diagrams for the JK implementation of the finite automaton used to control MAC circuit (see Example 4.3).
9.1. IMPLEMENTING FINITE AUTOMATA WITH "INTELLIGENT REGISTERS"

\[ J_1 = Q_0 \cdot \text{empty}' \]
\[ K_1 = Q_0 \cdot \text{full}' \]
\[ J_0 = Q_1 \cdot \text{empty}' \]
\[ K_0 = Q_1 \cdot \text{done} \]

If we compare with the previous D flip-flop solution where the loop circuit is defined by

\[ Q_1^+ = Q_1 \cdot Q_0 + Q_0 \cdot J_0 \cdot \text{empty}' + Q_1 \cdot K_0 \cdot \text{full} \]
\[ Q_0^+ = Q_1' \cdot Q_0 + Q_0 \cdot J_0' \cdot \text{done}' + Q_1' \cdot K_0 \cdot \text{empty}' \]

does a big reduction of complexity.

Figure 9.4: Finite automaton with smart “JK register”. The new implementation of FA from Figure 8.40 using a "JK register" as a state register. The associated half-automaton is simpler (the corresponding PLA is smaller).

In this new approach, using a “smart register”, a part of loopCLC from the automaton built with a true register was segregated in the uniform structure of the “JK register”. Indeed, the size of loopCLC decreases, but the size of each flip-flop increases with 3 units (instead of an inverter between S and R in D flip-flop, there are two AND2 in JK flip-flop). Thus, in this new variant the size of loopCLC decreases on the account of the size of the “JK register”.

This method acts as a mechanism that emphasizes more uniformities in the designing process and allows to build for the same function a less complex and, only sometimes, a smaller circuit. The efficiency of this method increases with the complexity and the size of the system.

We can say that loopCLC of the first versions has only an apparent complexity, because of a certain quantity of “order” distributed, maybe hidden, among the effective random parts of it. Because the
“order” sunk in “disorder” can not be easy recognized we say that “disorder + order” means “disorder”. In this respect, the apparent complexity must be defined. The apparent complexity of a circuit is reduced segregating the “hidden order”, until the circuit remains really random. The first step is done. The next step, in the following subsection.

What is the explanation for this segregation that implies the above presented minimization in the random part of the system? Shortly: because the “JK register” is a “smart register” having more autonomy than the true register built by D flip-flops. A D flip-flop has only the partial autonomy of staying in a certain state, instead of the JK flip-flop that has the autonomy to evolve in the state space. Indeed, for a D flip-flop we must all the time “say” on the input what will be the next state, 0 or 1, but for a JK flip-flop we have the vague, almost “evasive”, command \( J = K = 1 \) that says: “switch in the other state”, without indicating precisely, as for D flip-flop, the next state, because the JK “knows”, aided by the second loop, what is its present state.

Because of the second loop, that informs the JK flip-flop about its own state, the expressions for \( J_i \) and \( K_i \) do not depend by \( Q_i \), rather than \( Q_i^+ \) that depends on \( Q_i \). Thus, \( J_i \) and \( K_i \) are simplified. More autonomy means less control. For this reason the PLA that closes the third loop over a “JK register” is smaller than a PLA that closes the second loop over a true register.

9.1.2 * Automata using counters as registers

Are there ways to “extract” more “simplicity” by segregation from the PLA associated to an automaton? For some particular problems there is at least one more solution: to use a synchronous setable counter, \( SCOUNT_n \). The synchronous setable counter is a circuit that combines two functions, it is a register (loaded on the command L) and in the same time it is a counter (counting up under the command U). The load has priority before the count.

Instead of using few one-bit counters, i.e. JK flip-flops, one few-bit counter is used to store the state and to simplify, if possible, the control of the state transition. The coding style used is the incremental encoding (see E.4.3), which provides the possibility that some state transitions to be performed by counting (increment).

**Warning:** using setable counters is not always an efficient solution!

Follows two example. One is extremely encouraging, and another is more realistic.

**Example 9.3** The half-automaton associated to the codes assignment written in parenthesis in Figure 8.41 is implemented using an \( SCOUNT_n \) with \( n = 2 \). Because the states are coded using increment encoding, the state transitions in the flow-chart can be interpreted as follows:

- in the state \( q_0 \) if \( \text{empty} = 0 \), then the state code is incremented, else it remains the same
- in the state \( q_1 \) if \( \text{empty} = 0 \), then the state code is incremented, else it remains the same
- in the state \( q_2 \) if \( \text{done} = 1 \), then the state code is incremented, else it remains the same
- in the state \( q_3 \) if \( \text{full} = 0 \), then the state code is incremented, else it remains the same

Results the very simple (not necessarily very small) implementation represented in Figure 9.5. where a 4-input multiplexer selects according to the state the way the counter switches: by increment (up \( = 1 \)) or by loading (load \( = 1 \)). Comparing with the half-automaton part in the circuit represented in Figure 9.4, the version with counter is simpler, eventually smaller. But, the most important effect is the reducing complexity.

**Example 9.4** This example is also a remake. The half-automaton of the automaton which controls the operation \( \text{macc} \) in Example 4.6 will be implemented using a presetable counter as register. See Figure 8.62 for the state encoding. The idea is to have in the flow-chart as many as possible transitions by incrementing.

Building the solution starts from a \( SCOUNT_4 \) and a \( MUX_4 \) connected as in Figure 9.6. The multiplexer selects the counter’s operation (load or up-increment) in each state according to the flow-chart description. For example
Figure 9.5: Finite half-automaton implemented with a setable counter. The last implementation of the half-automaton associated with FA from Figure 8.40 (with the function defined in Figure 8.41 where the states coded in parenthesis). A synchronous two-bit counter is used as state register. The simple four-input MUX commands the counter.

in the state 0000 the transition is made by counting if empty = 0, else the state remains the same. Therefore, the multiplexer selects the value of empty′ to the input U of the counter.

The main idea is that the loading inputs I3, I2, I1 and I0 must have correct values only if in the current state the transition can be made by loading a certain value in the counter. Thus, in the definition of the logical functions associated with these inputs we have many "don't care"s. Results the circuit represented in Figure 9.6. The random part of the circuit is designed using the transition diagrams from Figure 9.7.

The resulting structure has a minimized random part. We assumed even the risk of increasing the recursive defined part of the circuit in order to reduce the random part of it.

Now, the autonomous device that allows reducing the randomness is the counter used as state register. An adequate state assignment implies many transitions by incrementing the state code. Thus, the basic function of the counter is many times involved in the state transition. Therefore, the second loop of the system, the simple defined “loop that counts”, is frequently used by the third loop, the random loop. The simple command UP, on the third loop, is like a complex “macro” executed by the second loop using simple circuits. This hierarchy of autonomies simplifies the system, because at the higher level the loop uses simple commands for complex actions. Let us remember:

- the loop over a true register (in 2-OS) uses the simple commands for the simplest actions: load 0 in D flip-flop and load 1 in D flip-flop
- the loop over a “JK register” (in 3-OS) uses beside the previous commands the following: no op (remain in the same state!) and switch (switch in the complementary state!)
- the loop over a SCOUNTn substitutes the command switch with the same simple expressed, but more powerful, command increment.

The “architecture” used on the third loop is more powerful than the two previous. Therefore, the effort of this loop to implement the same function is smaller, having the simpler expression: a reduced random circuit.

The segregation process is more deep, thus we imply in the designing process more simple, recursive defined, circuits. The apparent complexity of the previous solution is reduced towards, maybe on, the actual complexity. The complexity of the simple part is a little increased in order to “pay the price” for a strong minimization of the random part of the system. The quantitative aspects of our small example are not very significant. Only the design of the actual large systems offers a meaningful example concerning the quantitative effects.
9.2 Loops closed through memories

Because the storage elements do not perform logical or arithmetical functions - they only store - a loop closed through the 1-OS seems to be unuseful or at least strange. But a selective memorizing action is used sometimes to optimize the computational process! The key is to know what can be useful in the next steps.

The previous two examples of the third order systems belong to the subclass having a combinational loop. The function performed remains the same, only the efficiency is affected. In this section, because automata having the loop closed through a memory is presented, we expect the occurrence of some supplementary effects.

In order to exemplify how a trough memory loop works an Arithmetic & Logic Automaton – ALA – will be used (see Figure 9.8a). This circuit performs logic and arithmetic functions on data stored in its own state register called accumulator – ACC –, used as left operand and on the data received on its input in, used as right operand. A first version uses a control automaton to send commands to ALA, receiving back one flag: crout.

A second version of the system contains an additional D flip-flop used to store the value of the CRout signal, in each clock cycle when it is enabled (E = 1), in order to be applied on the CRin input of ALU. The control automaton is now substituted with a command automaton, used only to issue commands, without receiving back any flag.

Follow two example of using this ALA, one without an additional loop and another with the third loop closed trough a simple D flip-flop.
9.2. LOOPS CLOSED THROUGH MEMORIES

Figure 9.7: Transition diagrams for the presetable counter used as state register. The complex (random) part of the automaton is represented by the loop closed to the load input of the presetable counter.

Version 1: the controlled Arithmetic & Logic Automaton

In the first case ALA is **controlled** (see Figure 9.8a) using the following definition for the undefined fields of `< microinstruction>` specified in 8.4.3:

```plaintext
<command> ::= <func> <carry>
<func> ::= and | or | xor | add | sub | inc | shl | right;
<test> ::= crout | -;
```

Let be the sequence of commands that controls the increment of a double-length number:

```plaintext
inc jmp crout bubu // ACC = in + 1
right jmp cucu // ACC = in
bubu inc // ACC = in + 1
cucu ...
```

The first increment command is followed by different operation according to the value of `crout`. If `crout = 1` then the next command is an increment, else the next command is a simple load of the upper bits of the double-length operand into the accumulator. The control automaton decides according to the result of the first increment and behaves accordingly.

Version 2: the commanded Arithmetic & Logic Automaton

The second version of *Arithmetic & Logic Automaton* is a 3-OS because of the additional loop closed through the D flip-flop. The role of this new loop is to reduce, to simplify and to speed up the routine that performs the same operation. Now the microinstruction is actualized differently:

```plaintext
<command> ::= <func>;
<func> ::= right | and | or | xor | add |
          sub | inc | shl | addcr | subcr | incrcr | shlcr;
<test> ::= -;
```

The field `<test>` is not used, and the control automaton can be substituted by a command automaton. The field `<func>` is coded so as one of its bit is 1 for all arithmetic functions. This bit is used to enable the switch of D-FF. New functions are added: `addcr`, `subcr`, `incrcr`, `shlcr`. The instructions `xxxcr`
CHAPTER 9. PROCESSORS: THIRD ORDER, 3-LOOP DIGITAL SYSTEMS

Figure 9.8: The third loop closed over an arithmetic and logic automaton. a. The basic structure: a simple automaton (its loop is closed through a simple combinational circuit: ALU) working under the supervision of a control automaton. b. The improved version, with an additional 1-bit state register to store the carry signal. The control is simpler if the third loop “tells” back to the arithmetic automaton the value of the carry signal in the previous cycle.

operates with the value of carry F-F. The set of operations are defined now on in, ACC, carry with values in carry, ACC, as follows:

right: \{carry, ACC\} <= \{carry, in\}
and: \{carry, ACC\} <= \{carry, ACC & in\}
or: \{carry, ACC\} <= \{carry, ACC | in\}
xor: \{carry, ACC\} <= \{carry, ACC ^ in\}
add: \{carry, ACC\} <= ACC + in
sub: \{carry, ACC\} <= ACC - in
inc: \{carry, ACC\} <= in + 1
shl: \{carry, ACC\} <= \{in, 0\}
adccr: \{carry, ACC\} <= ACC + in + carry
subcr: \{carry, ACC\} <= ACC - in - carry
inccr: \{carry, ACC\} <= in + carry
shlcr: \{carry, ACC\} <= \{in, carry\}

The resulting difference in how the system works is that in each clock cycle \(CR_{in}\) is given by the content of the D flip-flop. Thus, the sequence of commands that performs the same action becomes:

inc // ACC = in + 1
inccr // ACC = in + Q
In the two previous use of the arithmetic and logic automaton the execution time remains the same, but the expression used to command the structure in the second version is shorter and simpler. The explanation for this effect is the improved autonomy of the second version of the ALA. The first version was a 2-OS but the second version is a 3-OS. A significant part of the random content of the ROM from CROM can be removed by this simple new loop. Again, more autonomy means less control. A small circuit added as a new loop can save much from the random part of the structure. Therefore, this kind of loop acts as a segregation method.

Specific for this type of loop is that adding simple circuits we save random, i.e., complex, structured symbolic structures. The circuits grow by simple physical structure and the complex symbolic structures are partially avoided.

In the first version the sequence of commands are executed by the automaton all the time in the same manner. In the second version, a simpler sequence of commands are executed different, according to the processed data that impose different values in the carry flop-flop. This “different execution” can be thought as an “interpretation”.

In fact, the execution is substituted by the interpretation, so as the apparent complexity of the symbolic structure is reduced based on the additional autonomy due to the third structural loop. The autonomy introduced by the new loop through the D flip-flop allowed the interpretation of the commands received from the sequencer, according to the value of CR.

The third loop allows the simplest form of interpretation, we will call it static interpretation. The fourth loop allows a dynamic interpretation, as we will see in the next chapter.

### 9.3 Loop coupled automata: the second ”turning point”

This last step in building 3-OS stresses specifically on the maximal segregation between the simple physical structure and the complex symbolic structures. The third loop allows us to make a deeper segregation between simple and complex.

We are in the point where the process of segregation between simple and complex physical structures ends. The physical structures reach the stage from which the evolution can be done only coupled with the symbolic structures. From this point a machine means: circuits that execute or interpret bit configurations structured under restrictions imposed by the formal languages used to describe the functionality to be performed.

#### 9.3.1 Counter extended automata (CEA)

Let us revisit Example 8.15 and try to solve the problem for $m = n$.

#### 9.3.2 The elementary processor

The most representative circuit in the class of 3-OS is the processor. The processor is maybe the most important digital circuit because of its flexibility to compute any computable function.

**Definition 9.1** The processor, $P$, is a circuit realized loop connecting a functional automaton with a finite (control) automaton. ◦

The function of a processor $P$ is specified by the sequences of commands “stored” in the loopCLC of the finite automaton used for control. (In a microprogrammed processor each sequence represents a mi-
croprogram. A microprogram consists in a sequence of microinstructions each containing the commands executed by the functional automaton and fields that allow to select the next microinstruction.)

In order to understand the main mechanisms involved by the third loop closed in digital systems we will present initially only how an elementary processor works.

**Definition 9.2** The elementary processor, EP, is a processor executing only one control sequence, i.e., the associated finite automaton is a strict initial automaton. ♦

An EP performs only one function. It is a structure having a fix, nonprogrammable function. The two parts of an EP are very different. One, the control automaton, is a complex structure, while another, the functional automaton, is a simple circuit assembled from few recursively defined circuits (registers, ALU, file registers, multiplexors, and the kind). This strong segregation between the simple part and the complex part of a circuit is the key idea on which the efficiency of this approach is based.

Even on this basic level the main aspect of computation manifest. It is about control and execution. The finite automaton performs the control, while the functional automaton executes the logic or arithmetical operations on data. The control depends on the function to be computed (the 2nd level loop at the level of the automaton) and on the actual data received by the system (the 3rd level loop at the system level).

**Example 9.5** Let’s revisit Example 5.2 in order to implement the function **interpol** using an EP. The organization of the EP **interpolEP** is presented in Figure 9.9.

![Diagram of interpolEP](image)

**Figure 9.9:** The elementary processor **interpolEP**.

The functional automaton consists of a register file, an Arithmetic and Logic Unit and a 2-way multiplexer. Such a simple functional automaton can be called **RALU** (Registers & ALU). In each clock
cycle two operands are read from the register file, they are operated in ALU, and the result is stored back at destination register in the register file. The multiplexer is used to load the register file with data. The loop closed from the ALU’s output to the MUX’s input is a 2nd level loop, because each register in the file register contains a first level loop.

The system has fully buffered connections. Synchronization signals (send, get, sendAck, getAck) are connected through D–FFs (one-bit registers) and data through two 8-bit registers: \( \text{inR} \) and \( \text{outR} \).

The control of the system is performed by the finite automaton FA. It is initialized by the reset signal, and evolve by testing three independent 1-bit signals: send (the sending external subsystem provides a new input byte), get (the receiving external subsystem is getting the data provided by the EP), zero (means the current output of ALU has the value 0). The last 1-bit signal closes the third loop of the system. The transition function is described in the following lines:

<table>
<thead>
<tr>
<th>STATE</th>
<th>FUNCTION</th>
<th>TEST</th>
<th>EXT. SIGNAL</th>
<th>NEXT STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>waitSend</td>
<td>reg0 &lt;= ( \text{inReg} ), if (send) sendAck, else next = test;</td>
<td></td>
<td></td>
<td>next = test;</td>
</tr>
<tr>
<td>test</td>
<td>reg1 &lt;= reg1, if (zero) next = add; else next = waitGet;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>waitGet</td>
<td>outReg &lt;= reg1, if (get) getAck, else next = move1;</td>
<td></td>
<td></td>
<td>next = move1;</td>
</tr>
<tr>
<td>move1</td>
<td>reg2 &lt;= reg1, next = move0;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>move0</td>
<td>reg1 &lt;= reg0, next = waitSend;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>reg1 &lt;= reg0 + reg2, next = divide;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>divide</td>
<td>reg1 &lt;= reg1 &gt;&gt; 1, next = waitGet;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The outputs of the automaton provide the command for the acknowledge signals for the external subsystems, and the internal command signals for RALU and output register \( \text{outR} \).

**Example 9.6** The EP structure is exemplified framed inside the simple system represented in Figure 9.10, where:

- \( \text{inFIFO} \) provides the input data for EP when \( \text{read} = 1 \) if \( \text{empty} = 0 \)
- \( \text{outFIFO} \) receives the output data generated by EP when \( \text{write} = 1 \) if \( \text{full} = 0 \)
- \( \text{LIFO} \) stores intermediary data for EP if \( \text{push} = 1 \) and send back the last sent data if \( \text{pop} \)

**Elementary Processor** is one of the simplest embodiment of an EP containing:

- **Control Automaton**: a strict initial control automaton (see CROM from Figure 8.60)
- **alu**: an Arithmetic & Logic Unit
- **acc_reg**: an accumulator register, used as state register for Arithmetic & Logic Automaton which is a functional automaton
- **mux**: is the multiplexer for select the left operand from \( \text{inFIFO} \) or from \( \text{LIFO} \).

The control automaton is a one function CROM that commands the functional automaton, receiving from it only the carry output, \( \text{cr} \), of the adder embedded in ALU.

The description of PE must be supplemented with the associated microprogramming language, as follows:

\[
\text{<microinstruction>} ::= \text{<label>} \text{<command>} \text{<mod>} \text{<test>} \text{<next>};
\text{<label>} ::= \text{<any string having maximum 6 symbols>};
\text{<command>} ::= \text{<func>} \text{<inout>};
\text{<mod>} ::= \text{jmp} | \text{cjmp} | -;
\]
Figure 9.10: **An example of elementary processor (EP)**. The third loop is closed between a simple execution automaton (alu & acc_reg) and a complex control automaton used to generate the sequence of operations to be performed by alu and to control the data flow between EP and the associated memory resources: LIFO, inFIFO, outFIFO.

\[
\begin{align*}
\text{<test>} &::= \text{zero} \mid \text{notzero} \mid \text{cr} \mid \text{notcr} \mid \text{empty} \mid \text{nempty} \mid \text{full} \mid \text{nfull};
\text{<next>} &::= \text{<label>};
\text{<func>} &::= \text{left} \mid \text{add} \mid \text{half0} \mid \text{half1} \mid - ;
\text{<inout>} &::= \text{read} \mid \text{write} \mid \text{push} \mid \text{pop} ;
\end{align*}
\]

where:

| notcr | inverted cr |
|------------------|
| nempty | inverted empty |
| nfull | inverted full |
| left | acc_reg <= left |
| add | acc_reg <= left + acc_reg |
| half0 | acc_reg <= \{0, acc_reg[n-1:1]\} |
| half1 | acc_reg <= \{1, acc_reg[n-1:1]\} |
| left = read ? out(inFIFO) : out(LIFO) |

and by default command are:

\[
\begin{align*}
\text{inc for <mode>}
\text{right: acc_reg <= acc_reg}
\end{align*}
\]

The only microprogram executed by the previous described EP receives a string of numbers and generates another string of numbers representing the mean values of the successive two received numbers. The numbers are positive integers. Using the previous defined microprogramming language results the following microprogram:

\[
\begin{align*}
\text{microprogram mean;}
\text{bubu read, cjmp, empty, bubu, left;}
\text{cucu cjmp, empty, cucu;}
\text{read, add, cjmp, cr, one;}
\text{half0;}
\end{align*}
\]
9.3. LOOP COUPLED AUTOMATA: THE SECOND "TURNING POINT"

```
out write, cjmp, full, out;
jmp, bubu;
one half1, jmp, out;
endmicroprogram
```

On the first line PE waits for non-empty inFIFO; when empty becomes inactive the last left command puts in the accumulator register the correct value. The second microinstruction PE waits for the second number, when the number arrives the microprogram goes to the next line. The third line adds the content of the register with the just read number from inFIFO. If \( c_\tau = 1 \), the next microinstruction will be one, else the next will be the following microinstruction. The fourth and the last microinstructions performs the right shift setting the most significant bit on 0, i.e., the division for finishing to compute the mean between the two received numbers. The line out sends out the result when full = 0. The jump to bubu restart again the procedure, and so on unending. The line one performs a right shift setting the most significant bit on 1.

The entire physical structure of EP is not relevant for the actual function it performs. The function is defined only by the loopCLC of the finite automaton. The control performed by the finite automaton combines the simple functional facilities of the functional automaton that is a simple logic-arithmetic automaton. The randomness is now concentrated in the structure of loopCLC which is the single complex structure in the system. If loopCLC is implemented as a ROM, then its internal structure is a symbolic one. As we said at the beginning of this section, at the level of 3-OS the complexity is segregated in the symbolic domain. The complexity is driven away from the circuits being lodged inside the symbolic structures supported by ROM. The complexity can not be avoided, it can be only transferred in the more controllable space of the symbolic structures.

9.3.3 Executing instructions vs. interpreting instructions

A processor is a machine which composes & loops functions performed by elementary processors. Let us call them elementary computations or, simply, instructions. But now it is not about composing circuits. The big difference from a physical composition or a physical looping, already discussed, is that now the composition and looping are done "in the symbolic domain".

As we know, an EP computes a function of variables received from an external sub-system (in the previous example from inFIFO), and sends the result to an external sub-system (in the previous example to outFIFO). Besides input variables a processor receives also functions. The results are stored sometimes internally or in specific external resources (for example a LIFO memory), and only at the end of a complex computation a result or a partial result is outputed.

The "symbolic composition" is performed applying the computation \( g \) on the results of computations \( h_m, \ldots h_0 \). Let's call now \( g, h_i \), or other similar simple computations, instructions.

The "symbolic looping" means to apply the same string of instructions to the same variables as many time as needed.

Any processor is characterized by its instruction set architecture (ISA). As we mentioned, an instruction is equivalent with an elementary computation performed by an EP, and its code is used to specify:

- the operation to be performed (<op_code>),
- sometimes an immediate operand, i.e., a value known at the moment the computation is defined (<value>),

therefore, in the simplest cases instruction :\( := \langle \text{op\_code} \rangle \langle \text{value} \rangle \)
A program is a sequence of instructions allowing to compose and to loop more or less complex computations.

There are two ways to perform an instruction:

- to execute it: to transcode op_code in one or many elementary operations executed in one clock cycle
- to interpret it: to expand op_code is a sequence of operations performed in many clock cycles.

Accordingly, two kind of processors are defined:

- executing processors
- interpreting processors.

Figure 9.11: The processor (P) in its environment. P works loop connected with an external memory containing data and programs. Inside P elementary function, applied to a small set of very accessible variables, are composed in linear or looped sequences. The instructions read from the external memory are executed in one (constant) clock cycle(s) or they are interpreted by a sequence of elementary functions.

In Figure 9.11 the processing module is framed in a typical context. The data to be computed and the instructions to be used perform the computation are stored in a RAM module (see in Figure 9.11 DATA & PROGRAMS). PROCESSOR is a separate unit used to compose and to loop strings of instructions. The internal resources of a processor consists, usually, in:

- a block to perform elementary computations, containing:
  - an ALU performing at least simple arithmetic operations and the basic logic operations
  - a memory support for storing the most used variable
- the block used to transform each instruction in an executable internal mico-code, with two possible versions:
9.3. LOOP COUPLED AUTOMATA: THE SECOND “TURNING POINT”

- a simple decoder allowing the execution of each instruction in one clock cycle
- a microprogrammed unit used to “expand” each instruction in a microprogram, thus allowing the interpretation of each instruction in a sequence of actions

- the block used to compose and to loop by:
  - reading the successive instructions organized as a program (by incrementing the PROGRAM COUNTER register) from the external memory devices, here grouped under the name DATA & PROGRAMS
  - jumping in the program space (by adding signed value to PROGRAM COUNTER)

In this section we introduce only the executing processors (in Chapter 11 the interpreting processor will be used to exemplify how the functional information works).

Informally, the processor architecture consists in two main components:

- the internal organization of the processor at the top level used to specify:
  - how are interconnected the top levels blocks of processor
  - the micro-architecture: the set of operations performed by each top level block

- the instruction set architecture (ISA) associated to the top level internal organization.

Von Neumann architecture / Harvard architecture

When the instruction must be executed (in one clock cycle) two distinct memories are mandatory, one for programs and one for data, because in each cycle a new instruction must be fetched and sometimes data must be exchanged between the external memory and the processor. But, when an instructions is interpreted in many clock cycles it is possible to have only one external memory, because, if a data transfer is needed, then it can be performed adding one or few extra cycles to the process of interpretation.

Two kind of computer architecture where imposed from the beginning of the history of computers:

- Harvard architecture with two external memories, one for data and another for programs (see Figure 10.8a)
• **von Neumann architecture** with only one external memory used for storing both data and programs (see Figure 10.8b).

The preferred embodiment for an executing processor is a Hardware architecture, and the preferred embodiment for an interpreting processor is a von Neumann architecture. For technological reasons in the first few decades of development of computing the von Neumann architecture was more taken into account. Now the technology being freed by a lot of restriction, we pay attention to both kind of architectures.

In the next two subsections both, executing processor (commercially called **Reduced Instruction Set Computer** – RISC – processors) and interpreting processor (commercially called **Complex Instruction Set Computer** – CISC – processors) are exemplified by implementing very simple versions.

### 9.3.4 An executing processor

The executing processor is simpler than an interpreting processor. The complexity of computation moves almost completely from the physical structure of the processor into the programs executed by the processor, because a RISC processor has an organization containing mainly simple, recursively defined circuits.

**The organization**

The Harvard architecture of a RISC executing machine (see Figure 10.8a) determine the internal structure of the processor to have mechanisms allowing in each clock cycle to address both, the program memory and the data memory. Thus, the RALU-type functional automaton, directly interfaced with the data memory, is loop-connected with a control automaton designed to fetch in each clock cycle a new instruction from the program memory. The control automaton does not “know” the function to be performed, as it does for the elementary processor, rather he “knows” how to “fetch the function” from an external storage support, the program memory\(^2\).

The organization of the simple executive processor **toyRISC** is given in Figure 9.13, where the **RALU** subsystem is connected with the **Control** subsystem, thus closing a 3rd loop.

**Control** section is simple functional automaton whose state, stored in the register called Program Counter (PC), is used to compute in each clock cycle the address from where the next instruction is fetched. There are two modes to compute the next address: incrementing, with 1 or signed number the current address. The next address can be set, independently from the current value of PC, using a value fetched from an internal register or a value generated by the currently executed instruction. The way the address is computed can be determined by the value, 0 or different from 0, of a selected register. More, the current \(pc+1\) can be stored in an internal register when the control of the program call a new function and a return is needed. For all the previously described behaviors the combinational circuit **NextPC** is designed. It contains **outCLC** and **loopCLC** of the automaton whose state is stored in **PC**.

**RALU** section accepts data coming form data memory, from the currently executed instruction, or from the **Control** automaton, thus closing the 3rd loop.

Both, the **Control** automaton and the **RALU** automaton are simple, recursively defined automata. The computational complexity is completely moved in the code stored inside the program memory.

\(^2\)The relation between an elementary processor and a processor is somehow similar with the relation between a Turing Machine and an Universal Turing Machine.
The instruction set architecture

The architecture of toyRISC processor is described in Figure 9.14.

The 32-bit instruction has two forms: (1) control form, and (2) arithmetic-logic & memory form. The first field, opCode, is used to determine what is the form of the current instruction. Each instruction is executed in one clock cycle.
INSTRUCTION SET ARCHITECTURE

reg [15:0] pc; // program counter
reg [31:0] programMemory[0:65535];
reg [31:0] dataMemory[0:n-1];
instruction[31:0] = {opCode[5:0], dest[4:0], left[4:0], value[15:0]} |
{opCode[5:0], dest[4:0], left[4:0], right[4:0], noUse[10:0]};  

parameter // CONTROL
nop = 6'b00_0000, // no operation: pc = pc+1;
rjmp = 6'b00_0001, // relative jump: pc = pc + value;
zjpm = 6'b00_0010, // cond jump: pc = (rf[left] = 0) ? pc + value : pc+1
nzjmp = 6'b00_0011, // !cond jump: pc = !(rf[left] = 0) ? pc + value : pc+1
ret = 6'b00_0101, // return from subroutine: pc = rf[left][15:0];
ajmp = 6'b00_0110, // absolute jump: pc = value;
call = 6'b00_0111, // subroutine call: pc = value; rf[dest] = pc+1;

parameter // ARITHMETIC & LOGIC
inc = 6'b11_0000, // rf[dest] = rf[left] + 1; pc = pc+1;
dec = 6'b11_0001, // rf[dest] = rf[left] - 1; pc = pc+1;
add = 6'b11_0010, // rf[dest] = rf[left] + rf[right]; pc = pc+1;
sub = 6'b11_0011, // rf[dest] = rf[left] - rf[right]; pc = pc+1;
inccr = 6'b11_0100, // rf[dest] = (rf[left] + 1)[32]; pc = pc+1;
decr = 6'b11_0101, // rf[dest] = (rf[left] - 1)[32]; pc = pc+1;
addcr = 6'b11_0110, // rf[dest] = (rf[left] + rf[right])[32]; pc = pc+1;
subcr = 6'b11_0111, // rf[dest] = (rf[left] - rf[right])[32]; pc = pc+1;
lsh = 6'b11_1000, // rf[dest] = rf[left] >> 1; pc = pc+1;
ash = 6'b11_1001, // rf[dest] = {rf[left][31], rf[left][31:1]}; pc = pc+1;
move = 6'b11_1010, // rf[dest] = rf[left]; pc = pc+1;
swap = 6'b11_1011, // rf[dest] = {rf[left][15:0], rf[left][31:16]}; pc = pc+1;
neg = 6'b11_1100, // rf[dest] = ~rf[left]; pc = pc+1;
bwand = 6'b11_1101, // rf[dest] = rf[left] & rf[right]; pc = pc+1;
bwor = 6'b11_1110, // rf[dest] = rf[left] | rf[right]; pc = pc+1;
bxor = 6'b11_1111, // rf[dest] = rf[left] ^ rf[right]; pc = pc+1;

parameter // MEMORY
read = 6'b10_0000, // read from dataMemory[rf[right]]; pc = pc+1;
load = 6'b10_0111, // rf[dest] = dataOut; pc = pc+1;
store = 6'b10_1000, // dataMemory[rf[right]] = rf[left]; pc = pc+1;
val = 6'b01_0111; // rf[dest] = {{16*{value[15]}}, value}; pc = pc+1;

Figure 9.14: The architecture of toyRISC processor.
module toyRISC(output [15:0] instrAddr , // program memory address
    input [31:0] instruction , // instruction from program memory
    output [31:0] dataAddr , // data memory address
    output [31:0] dataOut , // data send to data memory
    input [31:0] dataIn , // data received from data memory
    output we , // write enable for data memory
    input reset ,
    input clock );

wire writeEnable ;
wire [15:0] incPc ;
wire [31:0] leftOp ;

Decode Decode( .we (we ),
    .writeEnable(writeEnable ),
    .opCode (instruction[31:26] ));

Control Control(instrAddr ,
    instruction ,
    incPc ,
    leftOp ,
    reset ,
    clock );

RALU RALU( instruction ,
    dataAddr ,
    dataOut ,
    dataIn ,
    incPc ,
    leftOp ,
    writeEnable ,
    clock );
endmodule

Figure 9.15: The top module of toyRISC processor. The modules Control and RALU of the design are simple circuits, while the module Decode is a small complex module.

module Decode( output we ,
    output writeEnable ,
    input [5:0] opCode );

assign we = opCode == 6'b101000 ;
endmodule

Figure 9.16: The module Decode of the toyRISC processor.
module Control(output [15:0] instrAddr ,
    input [31:0] instruction ,
    output [15:0] incPc ,
    input [31:0] leftOp ,
    input reset ,
    input clock );

reg [15:0] pc ;

always @(posedge clock) if (reset) pc <=0 ;
else pc <= instrAddr ;

nextPc nextPc( .addr (instrAddr ),
    .incPc (incPc ),
    .pc (pc ),
    .jmpVal (instruction[15:0] ),
    .leftOp (leftOp ),
    .opCode (instruction[31:26] ));
endmodule

Figure 9.17: The module Control of the toyRISC processor.

Implementing toyRISC

The structure of toyRISC will be implemented as part of a bigger project realized for a SoC, where the program memory and data memory are on the same chip, tightly coupled with our design. Therefore, the connections of the module are not very rigorously buffered.

The Figure 9.15 describe the structure of the top level of our design, which is composed by two simple modules and a small and complex one.

The time performance

The longest combinational path in a system using our toyRISC, which imposes the minimum clock period, is:

\[ T_{clock} = t_{clock\to\text{instruction}} + t_{left\text{Addr}\to left\text{Op}} + t_{through\text{ALU}} + t_{through\text{MUX}} + t_{file\text{RegSU}} \]

Because the system is not buffered the clock frequency depends also by the time behavior of the system directly connected with toyRISC. In this case \( t_{clock\to\text{instruction}} \) – the access time of the program memory, related to the active edge of the clock – is an extra-system parameter limiting the speed of our design. The internal propagation time to be considered are: the read time from the file register (\( t_{left\text{Addr}\to left\text{Op}} \) or \( t_{right\text{Addr}\to right\text{Op}} \)), the maximum propagation time through ALU (dominated by the time for an 32-bit arithmetic operation), the propagation time through a 4-way 32-bit multiplexer, and the set-up time on the file register’s data inputs. The way from the output of the file register through Next PC circuit is “shorter” because it contains a 16-bit adder, comparing with the 32-bit one of the ALU.
module RALU( input [31:0] instruction ,
            output [31:0] dataAddr ,
            output [31:0] dataOut ,
            input [31:0] dataIn ,
            input [15:0] incPc ,
            output [31:0] leftOp ,
            input writeEnable ,
            input clock );

wire [31:0] aluOut ;
wire [31:0] rightOp ;
wire [31:0] regFileIn ;

assign dataAddr = rightOp ;
assign dataOut = leftOp ;

fileReg fileReg(.leftOut (leftOp ),
                .rightOut (rightOp ),
                .in (regFileIn ),
                .leftAddr (instruction[15:11] ),
                .rightAddr (instruction[20:16] ),
                .destAddr (instruction[25:21] ),
                .writeEnable (writeEnable ),
                .clock (clock ));

mux4_32 mux(.out(regFileIn ),
            .in0({16'b0, incPc} ),
            .in1({{16{instruction[15]}}, instruction[15:0] } ),
            .in2(dataIn ),
            .in3(aluOut ),
            .sel(instruction[31:30] ));

alu alu(.out (aluOut ),
        .leftIn (leftOp ),
        .rightIn (rightOp ),
        .func (instruction[29:26] ),
        .clock (clock ));

endmodule

Figure 9.18: The module RALU of the toyRISC processor.
module arithmetic( output reg [31:0] arithOut, 
    input [31:0] leftIn ,
    input [31:0] rightIn ,
    input [2:0] func ,
    input         clock );

    reg carry        ;
    reg nextCarry   ;

    always @(posedge clock) carry <= nextCarry ;

    always @(*)
        case(func)
            3'd000: {nextCarry, arithOut} = leftIn + 1'b1 ; // inc
            3'd001: {nextCarry, arithOut} = leftIn - 1'b1 ; // dec
            3'd010: {nextCarry, arithOut} = leftIn + rightIn ; // add
            3'd011: {nextCarry, arithOut} = leftIn - rightIn ; // sub
            3'd100: {nextCarry, arithOut} = leftIn + carry ; // inccr
            3'd101: {nextCarry, arithOut} = leftIn - carry ; // deccr
            3'd110: {nextCarry, arithOut} = leftIn + rightIn + carry; // addcr
            3'd111: {nextCarry, arithOut} = leftIn - rightIn - carry; // subcr
        endcase
    
endmodule

Figure 9.19: **The version 1 of the module alu of the toyRISC processor.**

module arithmetic( output [31:0] arithOut, 
    input [31:0] leftIn ,
    input [31:0] rightIn ,
    input [2:0] func ,
    input         clock );

    reg    carry  ;
    wire   nextCarry  ;

    always @(posedge clock) carry <= nextCarry ;

    assign {nextCarry, arithOut} =
        leftIn +
        {32{func[0]}} ^ (func[1] ? rightIn : {{31{1'b0}}, ~func[2]}) +
        func[0] ^ (func[2] ? carry : 1'b0)

endmodule

Figure 9.20: **The version 2 of the module alu of the toyRISC processor.**
module arithmetic(output [31:0] arithOut,
input [31:0] leftIn ,
input [31:0] rightIn ,
input [2:0] func ,
input clock );

reg carry ;
wire nextCarry ;
wire [31:0] rightOp ;
wire cr ;
always @(posedge clock) carry <= nextCarry ;
assign rightOp = {32{func[0]}} ^ (func[1] ? rightIn : {{31{1'b0}}, ~func[2]});
assign cr = func[0] ^ (func[2] ? carry : 1'b0) ;
assign {nextCarry, arithOut} = leftIn + rightOp + cr ;
endmodule

Figure 9.21: The version 3 of the module alu of the toyRISC processor.

9.3.5 * An interpreting processor

The interpreting processor are known also as processors having a Complex Instruction Set Computer (CISC)
architecture, or simply as CISC Processors. The interpreting approach allows us to design complex instructions
which are transformed at the hardware level in a sequence of operations. Lets remember that an executing (RISC)
processor has almost all instructions implemented in one clock cycle. It is not decided what style of designing an
architecture is the best. Depending on the application sometimes a RISC approach is mode efficient, sometimes a
CISC approach is preferred.

The organization

Our CISC Processor is a machine characterized by using a register file to store the internal (the most frequently
used) variables. The top level view of this version of processor is represented in Figure 9.22. It contains the
following blocks:

- REGISTER & ALU – RALU – 32 32-bit registers organized in a register file, and an ALU; the registers are
  used also for control purposes (program counter, return address, stack pointer in the external memory, ...)
- INPUT & OUTPUT BUFFER REGISTERS used to provide full synchronous connections with the exter-
  nal “world”, minimizing $t_{in\_reg}$, $t_{reg\_out}$, maximizing $f_{\text{max}}$, and avoiding $t_{in\_out}$ (see subsection 1.1.5); the
  registers are the following:
  - COM REG : sends out the 2-bit read or write command for the external data & program memory
  - ADDR REG : sends out the 32-bit address for the external data & program memory
  - OUT REG : sends out the 32-bit data for the external memory
  - DATA REG : receives back, with one clock cycle delay related to the command loaded in COM REG,
    32-bit data from the external data & program memory
  - INST REG : receives back, with one clock cycle delay related to the command loaded in COM REG,
    32-bit instruction from the external data & program memory
- CONTROL AUTOMATON used to control the fetch and the interpretation of the instructions stored in
  the external memory; it is an initial automaton initialized, for each new instruction, by the operation code
  ($\text{inst}[31:26]$ received from INST REG)
Figure 9.22: An interpreting processor. The organization is simpler because only one external memory is used.

The instruction of our CISC Processor has two formats. The first format is:

\[
\{ \text{opcode}[5:0] , \quad \text{// operation code} \\
\text{dest_addr}[4:0] , \quad \text{// selects the destination} \\
\text{left_addr}[4:0] , \quad \text{// selects the left operand} \\
\text{right_addr}[4:0] , \quad \text{// selects the right operand} \\
\text{rel_addr}[10:0] \} \quad \text{// small signed jump for program address} \\
= \text{instr}[31:0];
\]

The relative address allows a positive or a negative jump of 1023 instructions in the program space. It is sufficient for almost all jumps in a program. If not, special absolute jump instruction can solve this very rare cases.

The second format is used when the right operand is a constant value generated at the compiling time in the instruction body. It is:

\[
\{ \text{opcode}[5:0] , \quad \text{// operation code} \\
\text{dest_addr}[4:0] ,
\]
left_addr[4:0],
value[15:0]  }  // signed integer
= instr[31:0];

When the instruction is fetched from the external memory it is memorized in INST REG because its content will be used in different stages of the interpretation, as follows:

- \( \text{inst}[31:26] = \text{opcode}[5:0] \) to initialize CONTROL AUTOMATON in the state from which flows the sequence of commands used to interpret the current instruction
- \( \text{inst}[29:26] = \text{opcode}[3:0] \) to command the function performed by ALU in the step associated to perform the main operation associated with the current instruction (for example, if the instruction is add 12, 3, 7, then the bits \( \text{opcode}[3:0] \) are used to command the ALU to do the addition of registers 3 and 7 in the appropriate step of interpretation)
- \( \text{inst}[25:11] = \{\text{dest_addr}, \text{left_addr}, \text{right_addr}\} \) is used to address the REGISTER FILE unit when the main operation associated with the current instruction is performed
- \( \text{inst}[15:0] = \text{value} \) is selected to form the right operand when an instruction operating with immediate value is interpreted
- \( \text{inst}[10:0] = \text{rel_addr} \) is used in jump instructions, in the appropriate clock cycle, to compute the next program address.

The REGISTER FILE unit contains 32 32-bit registers. In each clock cycle, any ordered pair of registers can be selected as operands, and the result can be stored back in any of them. They have the following use:

- \( r0, r1, \ldots r29 \) are general purpose registers;
- \( r31 \) is used as Program Counter (PC);
- \( r30 \) is used to store the Return Address (RA) when the call instruction is interpreted (no embedded calls are allowed for this simple processor\(^3\)).

CONTROL AUTOMATON has the structure presented in Figure 9.23. In the fetch cycle \( \text{init} = 1 \) allows the automaton to jump into the state coded by \( \text{opcode} \), from which a sequence of operations flows with \( \text{init} = 0 \), ignoring the initialization input. This is the simplest way to associate for each instruction the interpreting sequence of elementary operation.

The output of CONTROL AUTOMATON commands all the top level blocks of our CISC Processor using the following fields:

\[
\begin{align*}
\{ \text{en}_\text{inst} & , \ // \text{write enable for the instruction register} \\
\text{write}_\text{enable} & , \ // \text{write back enable for the register file} \\
\text{dest}_\text{addr}[4:0] & , \ // \text{destination address in file register} \\
\text{left}_\text{addr}[4:0] & , \ // \text{left operand address in file register} \\
\text{alu}_\text{com}[3:0] & , \ // \text{alu functions} \\
\text{right}_\text{addr}[4:0] & , \ // \text{right operand address in file register} \\
\text{left}_\text{sel} & , \ // \text{left operand selection} \\
\text{right}_\text{sel}[1:0] & , \ // \text{right operand selection} \\
\text{mem}_\text{com}[1:0] & } \ // \text{memory command} \\
= \text{command}
\end{align*}
\]

\(^3\)If embedded calls are needed, then this register contains the stack pointer into a stack organized in the external memory. We are not interested in adding the feature of embedded calls, because in this digital system lessons we intend to keep the examples small and simple.
Figure 9.23: **The control automaton for our CISC Processor.** It is a more compact version of CROM (see Figure 8.60). Instead of a CLC used for the complex part of executing processor, for an interpreting processor a sequential machine is used to solve the problem of complexity.

The fields `dest_addr`, `left_addr`, `right_addr`, `alu_com` are sometimes selected from INST REG (see ADDR MUX and FUNC MUX in Figure 9.23) and sometimes their value is generated by CONTROL AUTOMATON according to the operation to be executed in the current clock cycle. The other command fields are generated by CONTROL AUTOMATON in each clock cycle.

CONTROL AUTOMATON receives back from ALU only one flag: the least significant bit of ALU, `alu_out[0]`; thus closing the third loop\(^4\).

In each clock cycle the content of two registers can be operated in ALU and the result stored in a third register.

The left operand can be sometimes `data_in` if `left_sel = 1`. It must be addressed two clock cycles before use, because the external memory is supposed to be a synchronous one, and the input register introduces another one cycle delay. The sequence generated by CONTROL AUTOMATON takes care by this synchronization.

The right operand can be sometimes `value = instr_reg[15:0]` if `right_sel = 2'b1x`. If `right_sel = 2'b01` the right operand is the 11-bit signed integer `rel_addr = instr_reg[10:0]`.

The external memory is addressed with a delay of one clock cycle using the value of `left_out`. We are not very happy about this additional delay, but this is the price for a robust design. What we loose in number of clock cycles used to perform some instructions is, at least partially, recuperated by the possibility to increase the frequency of the system clock.

Data to be written in the external memory is loaded into OUT REG from the right output of FILE REG. It is synchronous with the address.

The command for the external memory is also delayed one cycle by the synchronization register COM REG. It is generated by CONTROL AUTOMATON.

Data and instructions are received back from the external memory with two clock cycle delay, one because

\(^4\)The second loop is closed once in the big & simple automaton RALU, and another in the complex finite automaton CONTROL AUTOMATON. The first loop is closed in each flip-flop used to build the registers.
module cisc_processor(input  clock ,
input  reset ,
output  reg [31:0] addr_reg, // memory address
output  reg [1:0] com_reg , // memory command
output  reg [31:0] out_reg , // data output
input  [31:0] in ); // data/inst input

// INTERNAL CONNECTIONS
wire [25:0] command;
wire flag;
wire [31:0] alu_out, left, right, left_out, right_out;

// INPUT & OUTPUT BUFFER REGISTERS
reg [31:0] data_reg, inst_reg;
always @(posedge clock) begin
  if (command[25]) inst_reg <= in ;
  data_reg <= in ;
  addr_reg <= left_out ;
  out_reg <= right_out ;
  com_reg <= command[1:0] ;
end

// CONTROL AUTOMATON
control_automaton control_automaton(.clock (clock ),
  .reset (reset ),
  .inst (inst_reg[31:11]),
  .command(command ),
  .flag (alu_out[0] ));

// REGISTER FILE
register_file register_file(.left_out (left_out ),
  .right_out (right_out ),
  .result (alu_out ),
  .left_addr (command[18:14] ),
  .right_addr (command[9:5] ),
  .dest_addr (command[23:19] ),
  .write_enable (command[24] ),
  .clock (clock ) );

// MULTIPLEXERS
mux2 left_mux(.out(left ),
  .in0(left_out ),
  .in1(data_reg ),
  .sel(command[4]));
mux4 right_mux(.out(right ),
  .in0(right_out ),
  .in1({{21{inst_reg[10]}}, inst_reg[10:0] } ),
  .in2({16'b0, inst_reg[15:0] } ),
  .in3({inst_reg[15:0], 16'b0} ),
  .sel(command[3:2] ));

// ARITHMETIC & LOGIC UNIT
cisc_alu alu(.alu_out(alu_out ),
  .left (left ),
  .right (right ),
  .alu_com(command[13:10] ));
endmodule

Figure 9.24: The top module of our CISC Processor.
of we have an external synchronous memory, and another because of the input re-synchronization done by DATA REG and INST REG.

The structural Verilog description of the top level of our CISC Processor is in Figure 9.24.

Microarchitecture

The complex part of our CISC Processor is located in the block called CONTROL AUTOMATON. More precisely, the only complex circuit in this design is the loop of the automaton called “RANDOM CLC” (see Figure 9.23). The Verilog module describing CONTROL AUTOMATON is represented in Figure 9.25.

The micro-architecture defines all the fields used to control the simple parts of this processor. Some of them are used inside the control_automaton module, while others command the top modules of the processor.

The inside used fields command are the following:

- **init**: allows the jump of the automaton into the initial state associated with each instruction when init = new_seq
- **addr_sel**: the three 5-bit addresses for FILE REGISTER are considered only if the field addr_sel takes the value from_inst, else three special combinations of addresses are generated by the control automaton
- **func_sel**: the field alu_com is considered only if the field func_sel takes the value from_out, else the code opcode[3:0] selects the ALU’s function

The rest of fields command the function performed in each clock cycle by the top modules of our CISC Processor. They are:

- **en_inst**: enables the load of data received from the external memory only when it represents the next instruction to be interpreted
- **write_enable**: enables write back into FILE REGISTER the result from the output of ALU
- **alu_com**: is a 4-bit field used to command ALU’s function for the specific purpose of the interpretation process (it is considered only if func_sel = from_aut)
- **left_sel**: is the selection code for LEFT MUX (see Figure 9.22)
- **right_sel**: is the selection code for RIGHT MUX (see Figure 9.22)
- **mem_com**: generated the commands for the external memory containing both data and programs.

The micro-architecture (see Figure 9.26) is subject of possible changes during the definition of the transition function of CONTROL AUTOMATON.

Instruction set architecture (ISA)

There is a big flexibility in defining the ISA for a CISC machine, because we accepted to interpret each instruction using a sequence of micro-operations. The control automaton is used as sequencer for implementing instructions beyond what can be simply envisaged inspecting the organization of our simple CISC processor.

An executing (RISC) processor displays its architecture in its organization, because the control is very simple (the decoder is a combinational circuit used to trans-code only). The complexity of the control of an interpreting processor hides the architecture in the complex definition of the control automaton (which can have a strong generative power).

In Figure 9.27 is sketched a possible instruction set for our CISC processor. There are at least the following classes of instructions:

- **Arithmetic & Logic Instructions**: the destination register takes the value resulted from operating any two registers (unary operations, such as increment, are also allowed)
- **Data Move Instructions**: data exchange between the external memory and the file register are performed
9.3. LOOP COUPLED AUTOMATA: THE SECOND "TURNING POINT"

- Control Instructions: the flow of instruction is controlled according to the fix or data dependent patterns
- ...

We limit our discussion to few and small classes of instructions because our goal is to offer only a structural image about what an interpretative processor is. An exhaustive approach is an architectural one, which is far beyond our intention in these lessons about digital systems, not about computational systems.

Implementing ISA

Implementing a certain Instruction Set Architecture means to define the transition functions of the control automaton:

- the output transition function, in our case to specify for each state the value of the command code (see Figure 9.25)
- the state transition function, which specifies the value of next_state

The content of the file the_control_automaton's_loop.v contains the description of the combinational circuit associated to the control automaton. It generates both the 26-bit command code and the 6-bit next_state code. The following Verilog code is the most compact way to explain how the control automaton works. Please read the next "always" as the single way to explain rigorously how our CISC machine works.

```verilog
// THE CONTROL AUTOMATON'S LOOP
always @(state_reg or opcode or dest or left_op or right_op or flag)
begin
  en_inst = 1'bx;
  write_enable = 1'bx;
  dest_addr = 5'bxxxxx;
  left_addr = 5'bxxxxx;
  alu_com = 4'bxxxx;
  right_addr = 5'bxxxxx;
  left_sel = 1'bx;
  right_sel = 2'bxx;
  mem_com = 2'bxx;
  next_state = 6'bxxxxxx;
// INITIALIZE THE PROCESSOR
if (state_reg == 6'b00_0000)
  // pc = 0
  begin
    en_inst = no_load;
    write_enable = write_back;
    dest_addr = 5'b11111;
    left_addr = 5'b11111;
    alu_com = alu_xor;
    right_addr = 5'b11111;
    left_sel = left_out;
    right_sel = right_out;
    mem_com = mem_nop;
    next_state = state_reg + 1;
  end
// INSTRUCTION FETCH
if (state_reg == 6'b00_0001)
  // request for a new instruction & increment pc
  begin
```
CHAPTER 9. PROCESSORS: THIRD ORDER, 3-LOOP DIGITAL SYSTEMS

en_inst = no_load;
write_enable = write_back;
dest_addr = 5'b11111;
left_addr = 5'b11111;
alu_com = alu_inc;
right_addr = 5'bxxxxx;
left_sel = left_out;
right_sel = 2'bxx;
mem_com = mem_read;
next_state = state_reg + 1;
end

if (state_reg == 6'b00_0010)
// wait for memory to read doing nothing (synchronous memory)
begin
  en_inst = no_load;
  write_enable = no_write;
dest_addr = 5'bxxxxx;
left_addr = 5'bxxxxx;
alu_com = 4'bxxxx;
right_addr = 5'bxxxxx;
left_sel = 1'bx;
right_sel = 2'bxx;
mem_com = mem_nop;
next_state = state_reg + 1;
end

if (state_reg == 6'b00_0011)
// load the new instruction in instr_reg
begin
  en_inst = load_inst;
  write_enable = no_write;
dest_addr = 5'bxxxxx;
left_addr = 5'bxxxxx;
alu_com = 4'bxxxx;
right_addr = 5'bxxxxx;
left_sel = 1'bx;
right_sel = 2'bxx;
mem_com = mem_nop;
next_state = state_reg + 1;
end

if (state_reg == 6'b00_0100)
// initialize the control automaton
begin
  en_inst = no_load;
  write_enable = no_write;
dest_addr = 5'bxxxxx;
left_addr = 5'bxxxxx;
alu_com = 4'bxxxx;
right_addr = 5'bxxxxx;
left_sel = 1'bx;
right_sel = 2'bxx;
mem_com = mem_nop;
next_state = opcode[5:0];
9.3. LOOP COUPLED AUTOMATA: THE SECOND "TURNING POINT"

end

// EXECUTE THE ONE CYCLE FUNCTIONAL INSTRUCTIONS
if (state_reg[5:4] == 2’b10)
    // dest = left_op OPERATION right_op
    begin
        en_inst = no_load;
        write_enable = write_back;
        dest_addr = dest;
        left_addr = left_op;
        alu_com = opcode[3:0];
        right_addr = right_op;
        left_sel = left_out;
        right_sel = right_out;
        mem_com = mem_nop;
        next_state = 6’b00_0001;
    end

// EXECUTE MEMORY READ INSTRUCTIONS
if (state_reg == 6’b01_0000)
    // read from left_reg in dest_reg
    begin
        en_inst = no_load;
        write_enable = no_write;
        dest_addr = 5’bxxxxx;
        left_addr = left_op;
        alu_com = alu_left;
        right_addr = 5’bxxxxx;
        left_sel = left_out;
        right_sel = 2’bxx;
        mem_com = mem_read;
        next_state = 6’b01_0010;
    end

if (state_reg == 6’b01_0001)
    // read from left_reg + <value> in dest_reg
    begin
        en_inst = no_load;
        write_enable = no_write;
        dest_addr = 5’bxxxxx;
        left_addr = left_op;
        alu_com = alu_add;
        right_addr = 5’bxxxxx;
        left_sel = left_out;
        right_sel = low_value;
        mem_com = mem_read;
        next_state = 6’b01_0010;
    end

if (state_reg == 6’b01_0010)
    // wait for memory to read doing nothing
    begin
        en_inst = no_load;
        write_enable = no_write;
        dest_addr = 5’bxxxxx;
        left_addr = 5’bxxxxx;
    end
CHAPTER 9. PROCESSORS: THIRD ORDER, 3-LOOP DIGITAL SYSTEMS

alu_com = 4'bxxxx ;
right_addr = 5'bxxxxxx ;
left_sel = 1'bx ;
right_sel = 2'bxx ;
mem_com = mem_nop ;
next_state = state_reg + 1;

end
if (state_reg == 6'b01_0011)
// the data from memory is loaded in data_reg
begin
    en_inst = no_load ;
    write_enable = no_write ;
    dest_addr = 5'bxxxxxx ;
    left_addr = 5'bxxxxxx ;
    alu_com = 4'bxxxx ;
    right_addr = 5'bxxxxxx ;
    left_sel = 1'bx ;
    right_sel = 2'bxx ;
    mem_com = mem_nop ;
    next_state = state_reg + 1;
end

if (state_reg == 6'b01_0100)
// data_reg is loaded in dest_reg & go to fetch
begin
    en_inst = no_load ;
    write_enable = write_back ;
    dest_addr = dest ;
    left_addr = 5'bxxxxxx ;
    alu_com = alu_left ;
    right_addr = 5'bxxxxxx ;
    left_sel = from_mem ;
    right_sel = 2'bxx ;
    mem_com = mem_nop ;
    next_state = 6'b00_0001 ;
end

// EXECUTE MEMORY WRITE INSTRUCTIONS
if (state_reg == 6'b01_1000)
// write right_op to left_op & go to fetch
begin
    en_inst = no_load ;
    write_enable = no_write ;
    dest_addr = 5'bxxxxxx ;
    left_addr = left_op ;
    alu_com = alu_left ;
    right_addr = right_op ;
    left_sel = left_out ;
    right_sel = 2'bxx ;
    mem_com = mem_write ;
    next_state = 6'b00_0001 ;
end

if (state_reg == 6'b01_1000)
// write right_op to left_op + <value> & go to fetch
begin
    en_inst = no_load ;
    write_enable = no_write ;
    dest_addr = 5'bxxxxx ;
    left_addr = left_op ;
    alu_com = alu_add ;
    right_addr = right_op ;
    left_sel = left_out ;
    right_sel = low_value ;
    mem_com = mem_write ;
    next_state = 6'b00_0001 ;
end

// CONTROL INSTRUCTIONS
if (state_reg == 6'b11_0000)
// no operation & go to fetch
begin
    en_inst = no_load ;
    write_enable = no_write ;
    dest_addr = 5'bxxxxx ;
    left_addr = 5'bxxxxx ;
    alu_com = 4'bxxxx ;
    right_addr = 5'bxxxxx ;
    left_sel = 1'bx ;
    right_sel = 2'bxx ;
    mem_com = mem_nop ;
    next_state = 6'b00_0001 ;
end
if (state_reg == 6'b11_0001)
// jump to (pc + <value>) & go to fetch
begin
    en_inst = no_load ;
    write_enable = write_back ;
    dest_addr = 5'b11111 ;
    left_addr = 5'b11111 ;
    alu_com = alu_add ;
    right_addr = 5'bxxxxx ;
    left_sel = left_out ;
    right_sel = low_value ;
    mem_com = mem_nop ;
    next_state = 6'b00_0001 ;
end
if (state_reg == 6'b11_0010)
// call: first step: ra = pc + 1
begin
    en_inst = no_load ;
    write_enable = write_back ;
    dest_addr = 5'b11110 ;
    left_addr = 5'b11111 ;
    alu_com = alu_left ;
    right_addr = 5'bxxxxx ;
    left_sel = left_out ;
    right_sel = 2'bxx ;
mem_com = mem_nop;
next_state = 6'b11_0110;
end
if (state_reg == 8'b0011_0110)
// call: second step: pc = value
begin
  en_inst = no_load;
  write_enable = write_back;
  dest_addr = 5'b111111;
  left_addr = 5'bxxxxxx;
  alu_com = alu_right;
  right_addr = 5'bxxxxxx;
  left_sel = 1'bx;
  right_sel = jmp_addr;
  mem_com = mem_nop;
  next_state = 6'b00_0001;
end
if (state_reg == 6'b11_0011)
// ret: pc = ra
begin
  en_inst = no_load;
  write_enable = write_back;
  dest_addr = 5'b11111;
  left_addr = 5'b11110;
  alu_com = alu_left;
  right_addr = 5'bxxxxxx;
  left_sel = left_out;
  right_sel = 2'bxx;
  mem_com = mem_nop;
  next_state = 6'b00_0001;
end
if ((state_reg == 6'b11_0100) && flag)
// jzero: if (left_out = 0) pc = pc + value;
begin
  en_inst = no_load;
  write_enable = write_back;
  dest_addr = 5'b11111;
  left_addr = 5'b11111;
  alu_com = alu_add;
  right_addr = 5'bxxxxxx;
  left_sel = left_out;
  right_sel = low_value;
  mem_com = mem_nop;
  next_state = 6'b00_0001;
end
if ((state_reg == 6'b11_0100) && ~flag)
// jzero: if (left_out = 1) pc = pc + 1;
begin
  en_inst = no_load;
  write_enable = no_write;
  dest_addr = 5'bxxxxxx;
  left_addr = 5'bxxxxxx;
end
9.3. LOOP COUPLED AUTOMATA: THE SECOND "TURNING POINT"

alu_com = 4'bxxxx ;
right_addr = 5'bxxxxx ;
left_sel = 1'bx ;
right_sel = 2'bxx ;
mem_com = mem_nop ;
next_state = 6'b00_0001 ;
end

if ((state_reg == 6'b11_0100) && ~flag)
// jnzero: if (left_out = 1) pc = pc + value;
begin
    en_inst = no_load ;
    write_enable = write_back ;
    dest_addr = 5'b11111 ;
    left_addr = 5'b11111 ;
    alu_com = alu_add ;
    right_addr = 5'bxxxxx ;
    left_sel = left_out ;
    right_sel = low_value ;
    mem_com = mem_nop ;
    next_state = 6'b00_0001 ;
end

if ((state_reg == 6'b11_0100) && flag)
// jnzero: if (left_out = 0) pc = pc + 1;
begin
    en_inst = no_load ;
    write_enable = no_write ;
    dest_addr = 5'bxxxxx ;
    left_addr = 5'bxxxxx ;
    alu_com = 4'bxxxx ;
    right_addr = 5'bxxxxx ;
    left_sel = 1'bx ;
    right_sel = 2'bxx ;
    mem_com = mem_nop ;
    next_state = 6'b00_0001 ;
end

end

The automaton described by the previous code has 36 states for the 25 instructions implemented (see Figure 9.27). More instructions can added if new state are described in the previous “always”. Obviously, the most complex part of the processor is this combinational circuit associated to the control automaton.

**Time performance**

The representation from Figure 9.28 is used to evaluate the time restrictions imposed by our CISC processor.

The full registered external connections of the circuit allows us to provide the smallest possible values for minimum input arrival time before clock, $t_{in_{reg}}$, maximum output required time after clock, $t_{reg_{out}}$, and no path for maximum combinational path delay, $t_{in_{out}}$. The maximum clock frequency is fully determined by the internal structure of the processor, by the path on the loop closed inside RALU or between RALU and CONTROL AUTOMATON. The actual time characterization is:

- $t_{in_{reg}} = t_{su}$ – the set-up time for the input registers
- $t_{reg_{out}} = t_{reg}$ – the propagation time for the output registers
CHAPTER 9. PROCESSORS: THIRD ORDER, 3-LOOP DIGITAL SYSTEMS

- $T_{min} = \max(t_{RALU\_loop}, t_{processor\_loop})$, where:

$$t_{RALU\_loop} = t_{state\_reg} + t_{aut\_out\_cl} + t_{reg\_file} + t_{max} + t_{alu} + t_{reg\_file\_su}$$

$$t_{processor\_loop} = t_{state\_reg} + t_{aut\_out\_cl} + t_{reg\_file} + t_{max} + t_{alu\_flag} + t_{aut\_in\_cl} + t_{state\_reg\_su}$$

This well packed version of a simple processor is very well characterized as time behavior. The price for this is the increasing number of clock cycle used for executing an instruction. The effect of the increased number of clock cycles is sometimes compensated by the possibility to use a higher clock frequency. But, all the time the modularity is the main benefit.

Concluding about our CISC processor

A CISC processor is more complex than a stack processor because for each instruction the operands must be selected from the file register. The architecture is more flexible, but the loop closed in RALU is longer than the loop closed in SALU.

A CISC approach allows more complex operations performed during an instruction because it is interpreted, not simply executed in one clock cycle.

Interpretation allows a single memory for both data and programs with all the resulting advantages and disadvantages.

An interpreting processor contains a simple automaton – RALU – and a complex one – Control Automaton – because its complex behavior.

Both, Stack Processor and CISC Processor are only simple exercises designed for presenting the circuit aspects of the closing of the third loop. The real and complex architectural aspects are minimally presented because this textbook is about circuits not about computation.

9.4 The assembly language: the lowest programming level

The instruction set represent the machine language: the lowest programming level in a computation machine. The programming is very difficult at this level because of the concreteness of the process. Too many details must be known by the programmer. The main improvement added by a higher level language is the level of abstraction used to present the computational resources. Writing a program in machine language we must have in mind a lot of physical details of the machine. Therefore, a real application must be developed in a higher level language.

The machine language can be used only for some very critical section of the algorithms. The automatic translation done by a compiler from a high level language into the machine language is sometimes unsatisfactory for high performance application. Only in this cases small part of the code must be generated “manually” using the machine language.

9.5 Concluding about the third loop

The third loop is closed through simple automata avoiding the fast increasing of the complexity in digital circuit domain. It allows the autonomy of the control mechanism.

"Intelligent registers” ask less structural control maintaining the complexity of a finite automaton at the smallest possible level. Intelligent, loop driven circuits can be controlled using smaller complex circuits.

The loop through a storage element ask less symbolic control at the micro-architectural level. Less symbols are used to determine the same behavior because the local loop through a memory element generates additional information about the recent history.
Looping through a memory circuit allows a more complex “understanding” because the controlled circuits “knows” more about its behavior in the previous clock cycle. The circuit is somehow “conscious” about what it did before, thus being more “responsible” for the operation it performs now.

Looping through an automaton allows any effective computation. Using the theory of computation (see chapter Recursive Functions & Loops in this book) can be proved that any effective computation can be done using a three loop digital system. More than three loops are needed only for improving the efficiency of the computational structures.

The third loop allows the symbolic functional control using the arbitrary meaning associated to the binary codes embodied in instructions or micro-instructions. Both, the coding and the decoding process being controlled at the design level, the binary symbols act actualizing the potential structure of a programmable machine.

Real processors use circuit level parallelism discussed in the first chapter of this book. They are: data parallelism, time parallelism and speculative parallelism. How all these kind of parallelism are used is a computer architecture topic, beyond the goal of these lecture notes.

9.6 Problems

Problem 9.1 Interrupt automaton with asynchronous input.

Problem 9.2 Solving the second degree equations with an elementary processor.

Problem 9.3 Compute $y$ if $x$, $m$ and $n$ is given with an elementary processor.

Problem 9.4 Modify the unending loop of the processor to avoid spending time in testing if a new instruction is in inFIFO when it is there.

Problem 9.5 Define an instruction set for the processor described in this chapter using its microarchitecture.

Problem 9.6 Is it closed another loop in our Stack Processor connecting $tos$ to the input of DECODE unit?

Problem 9.7 Our CISC Processor: how must be coded the instruction set to avoid FUNC MUX?

9.7 Projects

Project 9.1 Design a specialized elementary processor for rasterization function.

Project 9.2 Design a system integrating in a parallel computational structure 8 rasterization processors designed in the previous project.

Project 9.3 Design a floating point arithmetic coprocessor.
Project 9.4 Design the RISC processor defined by the following Verilog behavioral description:

```verilog
module risc_processor();
endmodule
```

Project 9.5 Design a version of Stack Processor modifying SALU as follows: move MUX4 to the output of ALU and the input of STACK.
module control_automaton(
    input clock ,
    input reset ,
    input [20:0] inst ,
    output [25:0] command ,
    input [3:0] flags );

    `include "micro_architecture.v"
    `include "instruction_set_architecture.v"

    // THE STRUCTURE OF 'inst'
    wire [5:0] opcode ; // operation code
    wire [4:0] dest , // selects destination register
        left_op , // selects left operand register
    wire right_op; // selects right operand register
    assign {opcode, dest, left_op, right_op} = inst;

    // THE STRUCTURE OF 'command'
    reg en_inst ; // enable load a new instruction in inst_reg
    reg write_enable; // writes the output of alu at dest_addr
    reg [4:0] dest_addr ; // selects the destination register
    reg [4:0] left_addr ; // selects the left operand in file register
    reg [3:0] alu_com ; // selects the operation performed by the alu
    reg [4:0] right_addr ; // selects the right operand in file register
    reg left_sel ; // selects the source of the left operand
    reg [1:0] right_sel ; // selects the source of the right operand
    reg [1:0] mem_com ; // generates the command for memory
    assign command = {en_inst, write_enable, dest_addr, left_addr, alu_com,
        right_addr, left_sel, right_sel, mem_com};

    // THE STATE REGISTER
    reg [5:0] state_reg ; // the state register
    reg [5:0] next_state ; // a "register" used as variable
    always @(posedge clock) if (reset) state_reg <= 0 ;
    else state_reg <= next_state ;

    `include "the_control_automaton's_loop.v"
endmodule

Figure 9.25: Verilog code for control_automaton.
// MICRO-ARCHITECTURE
// en_inst
parameter no_load = 1'b0, // disable instruction register
load_inst = 1'b1; // enable instruction register

// write_enable
parameter no_write = 1'b0,
write_back = 1'b1; // write back the current ALU output

// alu_func
parameter alu_left = 4'b0000, // alu_out = left
alu_right = 4'b0001, // alu_out = right
alu_inc = 4'b0010, // alu_out = left + 1
alu_dec = 4'b0011, // alu_out = left - 1
alu_add = 4'b0100, // alu_out = left + right
alu_sub = 4'b0101, // alu_out = left - right
alu_shl = 4'b0110, // alu_out = {1'b0, left[31:1]}
alu_half = 4'b0111, // alu_out = {left[31], left[31:1]}
alu_zero = 4'b1000, // alu_out = {31'b0, (left == 0)}
alu_equal = 4'b1001, // alu_out = {31'b0, (left == right)}
alu_less = 4'b1010, // alu_out = {31'b0, (left < right)}
alu_carry = 4'b1011, // alu_out = {31'b0, add[32]}
alu_borrow = 4'b1100, // alu_out = {31'b0, sub[32]}
alu_and = 4'b1101, // alu_out = left & right
alu_or = 4'b1110, // alu_out = left | right
alu_xor = 4'b1111, // alu_out = left ^ right

// left_sel
parameter left_out = 1'b0, // left out of the reg file as left op
from_mem = 1'b1; // data from memory as left op

// right_sel
parameter right_out = 2'b00, // right out of the reg file as right op
jmp_addr = 2'b01, // right op = {{22{inst[10]}}, inst[9:0]}
low_value = 2'b10, // right op = {{16{inst[15]}}, inst[15:0]}
high_value = 2'b11; // right op = {inst[15:0], 16'b0}

// mem_com
parameter mem_nop = 2'b00,
read = 2'b10, // read from memory
write = 2'b11; // write to memory

Figure 9.26: The micro-architecture of our CISC Processor.
// INSTRUCTION SET ARCHITECTURE (only samples)
// arithmetic & logic instructions & pc = pc + 1
parameter
move   = 6'b10_0000, // dest_reg = left_out
inc    = 6'b10_0010, // dest_reg = left_out + 1
dec    = 6'b10_0011, // dest_reg = left_out - 1
add    = 6'b10_0100, // dest_reg = left_out + right_out
sub    = 6'b10_0101, // dest_reg = left_out - right_out
bwxor  = 6'b10_1111; // dest_reg = left_out ^ right_out
// ...
// data move instructions & pc = pc + 1
parameter
read   = 6'b01_0000, // dest_reg = mem(left_out)
rdinc  = 6'b01_0001, // dest_reg = mem(left_out + value)
write  = 6'b01_1000, // mem(left_out) = right_out
wrinc  = 6'b01_1001; // mem(left_out + value) = right_out
// ...
// control instructions
parameter
nop    = 6'b11_0000, // pc = pc + 1
jmp    = 6'b11_0001, // pc = pc + value
call   = 6'b11_0010, // pc = value, ra = pc + 1
ret    = 6'b11_0011, // pc = ra
jzero  = 6'b11_0100, // if (left_out = 0) pc = pc + value;
               // else pc = pc + 1
jnzero = 6'b11_0101; // if (left_out != 0) pc = pc + value;
               // else pc = pc + 1
// ...

Figure 9.27: The instruction set architecture of our CISC Processor. The partial definition of the file instruction_set_architecture.v included in the control_automaton.v file.
Figure 9.28: The simple block diagram of our CISC processor. The fully buffered solution imposed for designing this interpretative processor minimizes the depth of signal path entering and emerging in/from the circuit, and avoid a going through combinational path.
Chapter 10

COMPUTING MACHINES: 
≥4–loop digital systems

In the previous chapter
was introduced the main digital system - the processor - and we discussed how works the third loop in a digital system emphasizing

- effects on the size of digital circuits
- effects on the complexity of digital systems
- how the apparent complexity can be reduced to the actual complexity in a digital system

In this chapter
a very short introduction in the systems having more than three internal loops is provided, talking about

- how are defined the basic computational structures: microcontrollers, computers, stack machines, co-processors
- how the classification in orders starts to become obsolete with the fourth order systems
- the concept of embedded computation

In the next chapter
some futuristic systems are described as N-th order systems having the following features:

- they can behave as self-organizing systems
- they are cellular systems easy to be expanded in very large and simple powerful computational systems
Software is getting slower more rapidly than hardware becomes faster.

Wirth’s law

To compensate the effects of the bad behavior of software guys, besides the job done by the Moore law a lot of architectural work must be added.

The last examples of the previous chapter emphasized a process that appears as a "turning point" in 3-OS: the function of the system becomes lesser and lesser dependent on the physical structure and the function is more and more assumed by a symbolic structure (the program or the microprogram). The physical structure (the circuit) remains simple, rather than the symbolic structure, “stored” in program memory of in a ROM, that establishes the functional complexity. The fourth loop creates the condition for a total functional dependence on the symbolic structure. By the rule, at this level an universal circuit - the processor - executes (in RISC machines) or interprets (in CISC machines) symbolic structures stored in an additional device: the program memory.

10.1 Types of fourth order systems

There are four main types of fourth order systems (see Figure 10.1) depending on the order of the system through which the loop is closed:

1. **P & ROM** is a 4-OS with loop closed through a 0-OS - in Figure 10.1a the combinational circuit is a ROM containing only the programs executed or interpreted by the processor

2. **P & RAM** is a 4-OS with loop closed through a 1-OS - is the computer, the most representative structure in this order, having on the loop a RAM (see Figure 10.1b) that stores both data and programs

3. **P & LIFO** is a 4-OS with loop closed through a 2-OS - in Figure 10.1c the automaton is represented by a push-down stack containing, by the rule, data (or sequences in which the distinction between data and programs does not make sense, as in the Lisp programming language, for example)

4. **P & CO-P** is a 4-OS with loop closed through a 3-OS - in Figure 10.1d COPROCESSOR is also a processor but a specialized one executing efficiently critical functions in the system (in most of cases the coprocessor is a floating point arithmetic processor).

The representative system in the class of P & ROM is the microcontroller the most successful circuit in 4-OS. The microcontroller is a “best seller” circuit realized as a one-chip computer. The core of a microcontroller is a processor executing/interpreting the programs stored in a ROM.

---

1Niklaus Wirth is an already legendary Swiss born computer scientist with many contributions in developing various programming languages. The best known is Pascal. Wirth’s law is a sentence which Wirth made popular, but he attributed it to Martin Reiser.
10.1. TYPES OF FOURTH ORDER SYSTEMS

Figure 10.1: The four types of 4-OS machines. a. Fix program computers usual in embedded computation. b. General purpose computer. c. Specialized computer working working on a restricted data structure. d. Accelerated computation supported by a specialized co-processor.
The representative structure in the class of P & RAM is the computer. More precisely, the structure Processor - Channel - Memory represents the physical support for the well known von Neumann architecture. Almost all present-day computers are based on this architecture.

The third type of system seems to be strange, but a recent developed architecture is a stack oriented architecture defined for the successful Java language. Naturally, a real Java machine is endowed also with the program memory.

The third and the fourth types are machines in which the segregation process emphasized physical structures, a stack or a coprocessor. In both cases the segregated structures are also simple. The consequence is that the whole system is also a simple system. But, the first two systems are very complex systems in which the simple is net segregated by the random. The support of the random part is the ROM physical structure in the first case and the symbolic content of the RAM memory in the second.

The actual computing machines have currently more than order 4, because the processors involved in the applications have additional features. Many of these features are introduced by new loops that increase the autonomy of certain subsystems. But theoretically, the computer function asks at least four loops.

10.1.1 Counter extended CEA

Let us revisit again Example 8.15 and try to expand the problem to the process of recognizing $1^n0^m1^p$, for $n = m = p \geq 1$.

10.1.2 * Push-down automata

The first example of loop coupled automata uses a finite automaton and a functional automaton: the stack (LIFO memory). A finite complex structure is interconnected with an “infinite” but simple structure. The simple and the complex are thus perfectly segregated. This approach has the role of minimizing the size of the random part. More, this loop affects the magnitude order of the randomness, instead of the previous examples (Arithmetic & Logic Automaton) in which the size of randomness is reduced only by a constant. The proposed structure is a well known system having many theoretical and practical applications: the push-down automaton.

Figure 10.2: The push-down automaton (PDA). A finite (random) automaton loop-coupled with an “infinite” stack (a simple automaton) is an enhanced toll for dealing with formal languages.

Definition 10.1 The push-down automaton, $PDA$, (see Figure 10.2) built by a finite automaton loop connected with a push-down stack (LIFO), is defined by the six-tuple:

$$PDA = (X \times X', Y \times Y', X, Q, f, g, z_0)$$

where:
10.1. TYPES OF FOURTH ORDER SYSTEMS

\( \mathbf{X} \) : is the finite alphabet of the machine; the input string is in \( X^* \)

\( \mathbf{X}' \) : is the finite alphabet of the stack, \( X' = X' \cup \{ z_0 \} \)

\( \mathbf{Y} \) : is the finite output set of the machine

\( \mathbf{Y}' \) : is the set of commands issued by the finite automaton toward LIFO, \( \{ \text{PUSH, POP, } - \} \)

\( \mathbf{Q} \) : is the finite set of the automaton states (i.e., \( |Q| \neq h(\max I(s)) \), where \( s \in X^* \) is received on the input of the machine)

\( f \) : is the state transition function of the machine

\[
f : X \times X' \times Q \rightarrow Q \times X \times Y'
\]

(i.e., depending on the received symbol, by the value of the top of stack (TOS) and by the automaton’s state, the automaton switches in a new state, a new value can be sent to the stack and the stack receives a new command (PUSH, POP or NOP))

\( g \) : is the output transition function of the automaton - \( g : Q \rightarrow Y \)

\( z_0 \) : is the initial value of TOS.

\[ q_0 \]
\[ 0 \]
\[ X = \$ \]
\[ 1 \]
\[ q_0 \]
\[ 0 \]
\[ X = \& \]
\[ 1 \]
\[ PUSH X \]
\[ 0 \]
\[ X = S \]
\[ 1 \]
\[ POP \]
\[ 0 \]
\[ X = TOS \]
\[ 1 \]
\[ TOS = z_0 \]
\[ 0 \]
\[ q_4 \]
\[ \text{WRONG} \]
\[ 1 \]
\[ OK \]

Figure 10.3: Defining the behavior of a PDA. The algorithm detecting the antisymmetrical sequences of symbols.

**Example 10.1** The problem to be solved is designing a machine that recognizes strings having the form \( $x\&y$ \), where \( $, \& \in X \) and \( x, y \in X^* \), \( X \) being a finite alphabet and \( y \) is the antisymmetric version of \( x \).

The solution is to use a PDA with \( f \) and \( g \) described by the flow-chart given in Figure 10.3. Results a five state, initial (in \( q_0 \)) automaton, each state having the following meaning and role:
CHAPTER 10. COMPUTING MACHINES: ≥4–LOOP DIGITAL SYSTEMS

$q_0$: is the initial state in which the machine is waiting for the first dollar

$q_1$: in this state the received symbols are pushed into the stack, excepting & that switches the automaton in the next state

$q_2$: in this state, each received symbol is compared with TOS, that is popped on, while the received symbol is not a dollar; when the input is a dollar and TOS = $0 the automaton switches in $q_3$, else, if the received symbols do not correspond with the successive value of the TOS or the final value of TOS differs from $z_0$, the automaton switches in $q_4$

$q_3$: if the automaton is in this state the received string was recognized as a well formed string

$q_4$: if the automaton is in this state the received string was wrong.

The reader can try to solve the problem using only an automaton. For a given $X$ set, especially for a small set, the solution is possible and small, but the LOOP PLA of the resulting automaton will be a circuit with the size and the form depending by the dimension and by the content of the set $X$. If only one symbol is added or at least is changed, then the entire design process must be restarted from scratch. The automaton imposes a solution in which the simple, recursive part of the solution is mixed up with the random part, thus all the system has a very large apparent complexity. The automaton must store in the state space what PDA stores in stack. You imagine how huge become the state set in a such crazy solution. Both, the size and the complexity of the solution become unacceptable.

The solution with PDA, just presented, does not depend by the content and by the dimension of the set $X$. In this solution the simple is well segregated from the complex. The simple part is the “infinite” stack and the complex part is a small, five-state finite automaton.

10.1.3 The computer – support for the strongest segregation

The ROM content is defined symbolically and after that it is converted in the actual physical structure of ROM. Instead, the RAM content remains in symbolic form and has, in consequence, more flexibility. This is the main reason for considering the PROCESSOR & RAM = COMPUTER as the most representative in 4-OS.

The computer is not a circuit. It is a new entity with a special functional definition, currently called computer architecture. Mainly, the computer architecture is given by the machine language. A program written in this language is interpreted or executed by the processor. The program is stored in the RAM memory. In the same subsystem are stored data on which the program “acts”. Each architecture can have many associated computer structures (organizations).

Starting from the level of four order systems the behavior of the system is controlled mainly by the symbolic structure of programs. The architectural approach settles the distinction between the physical structures and the symbolic structures. Therefore, any computing machine supposes the following triadic definition (suggested by ["Milutinovic" '89]):

- the machine language (usually called architecture)
- the storage containing programs written in the machine language
- the machine that interprets the programs, containing:
  - the machine language ...
  - the storage ...
  - the machine ... containing:
    * ...

...
and so on until the machine executes the programs.

Does it make any sense to add new loops? Yes, but not too much! It can be justified to add loops inside the processor structure to improve its capacity to interpret fast the machine language by using simple circuits. Another way is to see PROCESSOR & COPROCESSOR or PROCESSOR & LIFO as performant processors and to add over them the loop through RAM. But, mainly these machines remain structures having the computer function. The computer needs at least four loops to be competent, but currently it is implemented on system having more loops in order to become performant.

10.2 * The stack processor – a processor as 4-OS

The best way to explain how to use the concept of architecture to design an executive processor is to use an example having an appropriate complexity. One of the simplest model of computing machine is the stack machine. A stack machine finds always its operands in the first two stages of a stack (LIFO) memory. The last two pushed data are the operands involved in the current operation. The computation must be managed to have accessible the current operand(s) in the data stack. The stack used in a stack processor have some additional features allowing an efficient data management. For example: double pop, swap, ... .

The high level description of a stack processor follows. The purpose of this description is to offer an example of how starts the design of a processor. Once the functionality of the machine is established at the higher level of the architecture, there are many ways to implement it.

10.2.1 * The organization

Our Stack Processor is a sort of simple processing element characterized by using a stack memory (LIFO) for storing the internal variables. The top level internal organization of a version of Stack Processor (see Figure 10.4) contains the following blocks:

- STACK & ALU – SALU – is the unit performing the elementary computations; it contains:
  - a two-output stack; the top of stack (stack0 or tos) and the previous recording (stack1) are accessible
  - an ALU with the operands from the top of stack (left_op = stack0 and right_io = stack1)
  - a selector for the input of stack grabbing data from: (0) the output of ALU, (1) external data memory, (2) the value provided by the instruction, or (3) the value of pc +1 to be used as return address

- PROGRAM FETCH – a unit used to generate in each clock cycle a new address for fetching from the external program memory the next instruction to be executed

- DECODER – is a combinational circuit used to trans-code the operation code – op_code – into commands executed by each internal block or sub-block.

Figure 10.5 represents the Verilog top module for our Stack Processor (stack_processor).

The two loop connected automata are SALU and PROGRAM FETCH. Both are simple, recursive defined structures. The complexity of the Stack Processor is given by the DECODE unit: a combinational circuit used to trans-code op_code providing 5 small command words to specify how behaves each component of the system. The Verilog decode module uses test_in = tos and mem_ready to make decisions. The value of tos can be tested (if it is zero or not, for example) to decide a conditional jump in program (on this way only PROGRAM FETCH module is affected). The mem_ready input received from data memory allows the processor to adapt itself to external memories having different access time.

The external data and program memories are both synchronous: the content addressed in the current clock cycle is received back in the next clock cycle. Therefore, instruction received in each clock cycle corresponds to instr_addr generated in the previous cycle. Thus, the fetch mechanism fits perfect with the behavior of the
Figure 10.4: An executing Stack Processor. Elementary functions are performed by ALU on variables stored in a stack (LIFO) memory. The decoder supports the one-cycle execution of the instructions fetched from the external memory.

synchronous memory. For data memory mem_ready flag is used to “inform” the decoder module to delay one clock cycle the use of the data received from the external data memory.

In each clock cycle ALU unit from SALU receives on its data inputs the two outputs of the stack, and generates the result of the operation selected by the alu_com code. If MUX4 has the input 0 selected by the data_sel code, then the result is applied to the input of stack. The result is written back in tos if a unary operation (increment, for example) is performed (write the result of increment in tos is equivalent with the sequence pop, increment & push). If a binary operation (addition, for example) is performed, then the first operand is popped from stack and the result is written back over the new tos (double pop & push involved in a binary operation is equivalent with pop & write).

MUX4 selects for the stack input, according to the command data_sel, besides the output of ALU, data received back from the external data memory, the value carried by the currently executed instruction, or the value pc+1 (to be used as return address).

The unit PC generates in each clock cycle the address for program memory. It uses mainly the value from the register PC, which contains the last used address, to fetch an instruction. The content of tos or the value contained in the current instruction are also used to compute different conditioned or unconditioned jumps.

To keep this example simple, the program memory is a synchronous one and it contains anytime the addressed instruction (no misses in this memory).

Because our Stack Processor is designed to be an executing machine, besides the block associated with the
module stack_processor(input clock ,
  input reset ,
  output [31:0] instr_addr , // instruction address
  output [31:0] data_addr , // data address
  output [1:0] data_mem_com, // data memory command
  output [31:0] data_out , // data output
  input [23:0] instruction , // instruction input
  input [31:0] data_in , // data input
  input mem_ready ); // data memory is ready

// INTERNAL CONNECTIONS
wire [2:0] stack_com ; // stack command
wire [3:0] alu_com ; // alu command
wire [1:0] data_sel ; // data selection for SALU
wire [2:0] pc_com ; // program counter command
wire [31:0] tos , // top of stack
  ret_addr ; // return from subroutine address

// INSTRUCTION DECODER UNIT
decode decode( .op_code (instruction[23:16]),
  .test_in (tos),
  .mem_ready (mem_ready),
  .stack_com (stack_com),
  .alu_com (alu_com),
  .data_sel (data_sel),
  .pc_com (pc_com),
  .data_mem_com (data_mem_com) );

// SALU: STACK WITH ARITHMETIC & LOGIC UNIT
salu salu( .stack0 (tos),
  .stack1 (data_out),
  .in1 (data_in),
  .in2 ({16'b0, instruction[15:0]}),
  .in3 (ret_addr),
  .s_com (stack_com),
  .data_sel (data_sel),
  .alu_com (alu_com),
  .reset (reset),
  .clock (clock) );
assign data_addr = tos;

// PROGRAM COUNTER UNIT
program_counter pc( .clock (clock),
  .reset (reset),
  .addr (instr_addr),
  .inc_pc (ret_addr),
  .value (instruction[15:0]),
  .tos (tos),
  .pc_com (pc_com) );
endmodule

Figure 10.5: The top level structural description of a Stack Processor. The Verilog code associated to the circuit represented in Figure 10.4.
module decode( input [7:0] op_code ,
input [31:0] test_in ,
input mem_ready ,
output [2:0] stack_com ,
output [3:0] alu_com ,
output [1:0] data_sel ,
output [2:0] pc_com ,
output [1:0] data_mem_com);

`include "micro_architecture.v"
`include "instruction_set_architecture.v"
`include "decoder_implementation.v"
endmodule

Figure 10.6: The decode module. It contains the three complex components of the description of Stack Processor.

elementary functions (SALU) and the block used to compose & and loop them (PC) there is only a decoder used as execution unit (see Figure 9.11. The decoder module – decode – is the most complex module of Stack Processor (see Figure 10.6). It contains three sections:

- micro-architecture: it describes the micro-operations performed by each top level block listing the meaning of all binary codes used to command them
- instruction set architecture: describe each instruction performed by Stack Processor
- decoder implementation: describe how the micro-architecture is used to implement the instruction set architecture.

10.2.2 The micro-architecture

Any architecture can be implemented using various micro-architectures. For our Stack Processor one of them is presented in Figure 10.7. The decoder unit generates in each clock cycle a command word having the following 5-field structure:

\[
\{alu\_com, data\_sel, stack\_com, data\_mem\_com, pc\_com\} = \text{command}
\]

where:

- **alu\_com**: is a 4-bit code used to select the arithmetic or logic operation performed by ALU in the current cycle; it specifies:
  - well known binary operations such as: add, subtract, and, or, xor
  - usual unary operations such as: increment, shifts
  - test operations indicating by alu_out[0] the result of testing, for example: if an input is zero or if an input is less than another input

- **data\_sel**: is a 2-bit code used to select the value applied on the input of the stack for the current cycle as one from the following:
  - the output of ALU
  - data received from data memory addressed by tos (with a delay of one clock cycle controlled by mem_ready signal because the external data memory is synchronous)
  - the 16-bit integer selected from the current instruction
10.2. *THE STACK PROCESSOR – A PROCESSOR AS 4-OS*

// MICROARCHITECTURE
// pc_com
parameter
  stop = 3'b000, // pc = pc
  next = 3'b001, // pc = pc + 1
  small_jmp = 3'b010, // pc = pc + value
  big_jmp = 3'b011, // pc = pc + tos
  abs_jmp = 3'b100, // pc = value
  ret_jmp = 3'b101; // pc = tos

// alu_com
parameter
  alu_left = 4'b0000, // alu_out = left
  alu_right = 4'b0001, // alu_out = right
  alu_inc = 4'b0010, // alu_out = left + 1
  alu_dec = 4'b0011, // alu_out = left - 1
  alu_add = 4'b0100, // alu_out = left + right = add[32:0]
  alu_sub = 4'b0101, // alu_out = left - right = sub[32:0]
  alu_shl = 4'b0110, // alu_out = {1'b0, left[31:1]}
  alu_half = 4'b0111, // alu_out = {left[31], left[31:1]}
  alu_zero = 4'b1000, // alu_out = {31'b0, (left == 0)}
  alu_equal = 4'b1001, // alu_out = {31'b0, (left == right)}
  alu_less = 4'b1010, // alu_out = {31'b0, (left < right)}
  alu_carry = 4'b1011, // alu_out = {31'b0, add[32]}
  alu_borrow = 4'b1100, // alu_out = {31'b0, sub[32]}
  alu_and = 4'b1101, // alu_out = left & right
  alu_or = 4'b1110, // alu_out = left | right
  alu_xor = 4'b1111; // alu_out = left ^ right

// data_sel
parameter
  alu = 2'b00, // stack_input = alu_out
  mem = 2'b01, // stack_input = data_in
  val = 2'b10, // stack_input = value
  return = 2'b11; // stack_input = ret_addr

// stack_com
parameter
  s_nop = 3'b000, // no operation
  s_swap = 3'b001, // swap the content of the first two
  s_push = 3'b010, // push
  s_write = 3'b100, // write in tos
  s_pop = 3'b101, // pop
  s_popwr = 3'b110, // pop2 & push
  s_pop2 = 3'b111; // pops two values

// data_mem_com
parameter
  mem_nop = 2'b00, // no data memory command
  read = 2'b01, // read from data memory
  write = 2'b10; // write to data memory

Figure 10.7: The micro-architecture of our Stack Processor. The content of file micro_architecture.v defines each command word generated by the decoder describing the associated micro-commands and their binary codes.
– pc+1, generated by the PROGRAM FETCH module, to be pushed in stack when the a call instruction is executed

• stack_com: is a 3-bit code used to select the operation performed by the stack unit in the current cycle (it is correlated with the ALU operation selected by alu_com); the following micro-operations are coded:

– push: it is the well known standard writing operation into a stack memory
– pop: it is the well known standard reading operation into a stack memory
– write: it writes in top of stack, which is equivalent with popping an operand and pushing back the result of operation performed on it (used mainly in performing unary operations)
– pop & write: it is equivalent with popping two operands from stack and pushing back the result of operation performed on them (used mainly in performing binary operations)
– double pop: it is equivalent with two successive pops, but is performed in one clock cycle; some instructions need to remove both the content of stack0 and of stack1 (for example, after a data write into the external data memory)
– swap: it exchange the content of stack0 and of stack1; it is useful, for example to make a subtract in the desired order.

• data_mem_com: is a 2-bit command for the external data memory; it has three instantiations:

– memory nop: keep memory doing nothing is a very important command
– read: commands the read operation from data memory with the address from tos; the data will be returned in the next clock cycle; in the current cycle mem_read is activated to allow stoping the processor one clock cycle (the associated read instruction will be executed in two clock cycles)
– write: the data contained in stack1 is written to the address contained in stack0 (both, address and data will be popped from stack)

• pc_com: is a 3-bit code used to command how is computed the address for the fetching of the next instruction; 6 modes are used:

– stop: program counter is not incremented (the processor halts or is waiting for a condition to be fulfilled)
– next: it is the most frequent mode to compute the program counter by incrementing it
– small jump: compute the next program counter adding to it the value contained in the current instruction (instruction[15:0]) interpreted as a 16-bit signed integer; a relative jump in program is performed
– big jump: compute the next program counter adding to it the value contained in tos interpreted as a 32-bit signed integer; a relative big jump in program is performed
– absolute jump: the program counter takes the value of instruction[15:0]; the processor performs an absolute jump in program
– return jump: is an absolute jump performed using the content of tos (usually performs a return from a subroutine, or is used to call a subroutine in a big addressing space)

The 5-field just explained can not be filled up without inter-restrictions imposed by the meaning of the micro-operations. There exist inter-correlations between the micro-operations assembled in a command. For example, if ALU performs an addition, then the stack must perform mandatory pop & poop & push == pop_write. If the ALU operation is increment, then the stack must perform write. Some fields are sometimes meaningless. For example, when an unconditioned small jump is performed the fields alu_com and data_sel can take don’t care values. But, for obvious reasons, no times stack_com and data_mem_com can take don’t care values.

Each unconditioned instruction has associated one 5-field commands, and each conditioned instructions is defined using two 5-field commands.
10.2.3 * The instruction set architecture*

Instruction set architecture is the *interface* between the hardware and the software part of a computing machine. It grounds the definition of the lowest level programming language: the *assembly language*. It is an interface because allows the parallel work of two teams once its definitions is frozen. One is the hardware team which starts to design the physical structure, and the other is the software team which starts to grow the symbolic structure of the hierarchy of programs. Each architecture can be embodied in many forms according to the technological restrictions or to the imposed performances. The main benefit of this concept is the possibility to change the hardware without throwing out the work done by the software team.

// INSTRUCTION SET ARCHITECTURE
// arithmetic & logic instructions (pc <= pc + 1)
parameter	nop = 8'b0000_0000, // s0, s1, s2 ... <= s0, s1, s2, ...
	add = 8'b0000_0001, // s0, s1, s2 ... <= s0 + s1, s2, ...
	inc = 8'b0000_0010, // s0, s1, s2 ... <= s0 + 1, s1, s2, ...
	half = 8'b0000_0011; // s0, s1, s2 ... <= s0/2, s1, s2, ...
	...
// input output instructions (pc <= pc + 1)
parameter
tload = 8'b0001_0000, // s0, s1, s2 ... <= data_mem[s0], s1, s2, ...
	store = 8'b0001_0001; // s0, s1, s2 ... <= s2, s3, ...; data_mem[s0] = s1
// stack instructions (pc <= pc + 1)
parameter
tpush = 8'b0010_0000, // s0, s1, s2 ... <= value, s0, s1, ...
	pop = 8'b0010_0010, // s0, s1, s2 ... <= s1, s2, ...
	dup = 8'b0010_0011, // s0, s1, s2 ... <= s0, s0, s1, s2, ...
	swap = 8'b0010_0100, // s0, s1, s2 ... <= s1, s0, s2, ...
	over = 8'b0010_0101; // s0, s1, s2 ... <= s1, s0, s1, s2, ...
	...
// test instructions (pc <= pc + 1)
parameter
tzero = 8'b0100_0000, // s0, s1, s2 ... <= (s0 == 0), s1, s2, ...
	eq = 8'b0100_0001; // s0, s1, s2 ... <= (s0 == s1), s2, ...
	...
// control instructions
parameter
tjmp = 8'b0111_0000, // pc = pc + value
tcall = 8'b0111_0001, // pc = s0; s0, s1, ... <= pc + 1, s1, ...

tcjmpz = 8'b0111_0010, // if (s0 == 0) pc <= pc + value, else pc <= pc + 1
tcjmpnz = 8'b0111_0011, // if (s0 == 0) pc <= pc + 1, else pc <= pc + value
	ret = 8'b0111_0111; // pc = s0; s0, s1, ... <= s1, s2, ...
	...

Figure 10.8: *Instruction set architecture of our Stack Processor.* From each subset few typical example are shown. The content of data stack is represented by: s0, s1, s2, ...

The implementation of our Stack Processor has, as the majority of the currently produced processors, an instruction set architecture containing the following class of instructions:

**arithmetic and logic instructions** having the form:

- \([\text{stack0}, \text{stack1}, s2, \ldots] = [\text{op}(\text{stack0}, \text{stack1}), s2, \ldots]\)

where: stack0 is the *top of stack*, stack1 is the *next recording* in stack, and \(\text{op}\) is an arithmetic or
logic binary operation

- \([\text{stack}_0, \text{stack}_1, \text{s}_2, \ldots] = [(\text{op}(\text{stack}_0), \text{stack}_1, \text{s}_2, \ldots]\)
  
if the operation \(\text{op}\) is unary

input-output instructions which uses stack0 as data_addr and stack1 as data_out

stack instructions (only for stack processors) used to immediate load the stack or to change the content in the first two recordings (stack0 and stack1)

test instructions used to test the content of stack putting the result of the test back into the stack

control instructions used to execute unconditioned or conditioned jumps in the instruction stream by modifying the variable program_counter used to address in the program space.

The instruction set architecture is given as part of the Verilog code describing the module decode: the content of the file instruction_set_architecture.v (a more complete stage of this module in Appendix: Designing a Stack processor). Figure 10.8 contains an incipient form of file instruction_set_architecture.v. From each class of instructions only few examples are shown. Each instruction is performed in one clock cycle, except \text{load} whose execution can be delayed if \text{data_ready} = 0.

10.2.4 * Implementation: from micro-architecture to architecture

Designing a processor (in our case designing the Stack Processor) means to use the micro-architecture to implement the instruction set architecture. For an executing processor the "connection" between micro-architecture and architecture is done by the decoder which is a combinational structure.

The main body of the decode module – decoder_implementation.v – contains the description of the Stack Processor's instruction set architecture in term of micro-architecture.

The structure of the file decoder_implementation.v is suggested in Figure 10.9, where the output variables are the 5 command fields (declared as registers) and the input variables are: the operation code from instruction, the value of tos received as test_in and the flag received from the external memory: \text{mem_ready}.

The main body of this file consists in a big case structure with an entry for each instruction. In Figure 10.9 only few instructions are implemented (nop, add, load) to show how an unconditioned instruction nop, add or a conditioned instruction load is executed.

Instruction \text{nop} does not affect the state of stack and PC is incremented. We must take care only about three command fields. PC must be incremented (next, and the fields commanding memory resources (stack, external data memory) must be set on "no operation" (\text{s\_nop}, \text{mem\_nop}. The operation performed by ALU and data selected as right operand have no meaning for this instruction.

Instruction \text{add} pops the two last recordings in stack, adds them, pushes back the result in tos, and increments PC. Meantime the data memory receives no active command.

Instruction \text{load} is executed in two clock cycles. In the first cycle, when \text{mem_ready} = 0, the command \text{read} is sent to the external data memory, and the PC is maintained unchanged. The operation performed by ALU does not matter. The selection code for MUX4 does not matter. In the next clock cycle data memory sets it flag on 1 (\text{mem_ready} = 1 means the requested data is available), data selected is from memory \text{mem}, and the output of MUX4 is pushed in stack (\text{s\_push}).

By default the decoder generates “don’t care” commands. Another possibility is to have \text{nop} instruction the “by default” instruction. Or by default to have a halt instruction which stops the processor. The first version is good as a final solution because generates a minimal solution. The last version is preferred in the initial stage of development because provides an easy testing and debugging solution.
// THE IMPLEMENTATION

reg [3:0] alu_com ;
reg [2:0] pc_com, stack_com ;
reg [1:0] data_sel, data_mem_com ;

always @(op_code or test_in or mem_ready )
case(op_code)
  // arithmetic & logic instructions
  nop : begin pc_com = next ;
        alu_com = 4’bx ;
        data_sel = 2’bx ;
        stack_com = s_nop ;
        data_mem_com = mem_nop ;
      end
  add : begin pc_com = next ;
        alu_com = alu_add ;
        data_sel = alu ;
        stack_com = s_popwr ;
        data_mem_com = mem_nop ;
      end
  // ...
  // input output instructions
  load : if (mem_ready)
    begin pc_com = next ;
        alu_com = 4’bx ;
        data_sel = mem ;
        stack_com = s_write ;
        data_mem_com = mem_nop ;
      end
    else
    begin pc_com = stop ;
        alu_com = 4’bx ;
        data_sel = 2’bx ;
        stack_com = s_nop ;
        data_mem_com = read ;
      end
  // ...
  default begin pc_com = 3’bx ;
          alu_com = 4’bx ;
          data_sel = 2’bx ;
          stack_com = 3’bx ;
          data_mem_com = 2’bx ;
    end
endcase

Figure 10.9: Sample from the file decoder.implementation.v. Implementation consists in a big case form with an entry for each instruction.
Follows the description of some typical instructions from a possible instruction set executed by our Stack Processor.

**Instruction** *inc*  increments the top of stack, and increments also PC. The right operand of ALU does not matter. The code describing this instruction, to be inserted into the big case sketched in Figure 10.9, is the following:

```plaintext
inc : begin
    pc_com = next ;
    alu_com = alu_inc ;
    data_sel = alu ;
    stack_com = s_write ;
    data_mem = m_nop ;
end
```

**Instruction** *store*  stores the value contained in stack1 at the address from stack0 in external data memory. Both, data and address are popped from stack. The associated code is:

```plaintext
store : begin
    pc_com = next ;
    alu_com = 4'bx ;
    data_sel = 2'bx ;
    stack_com = s_pop2 ;
    data_mem = write ;
end
```

**Instruction** *push*  pushes `{16'b0, instruction[15:0]}` in in stack. The code is:

```plaintext
push : begin
    pc_com = next ;
    alu_com = 4'bx ;
    data_sel = val ;
    stack_com = s_push ;
    data_mem = m_nop ;
end
```

**Instruction** *dup*  pushes in stack the top of stack, thus duplicating it. ALU performs alu_left, the right operand does not matter, and in the stack is pusher the output of ALU. The code is:

```plaintext
dup : begin
    pc_com = next ;
    alu_com = alu_left ;
    data_sel = alu ;
    stack_com = s_push ;
    data_mem = m_nop ;
end
```

**Instruction** *over*  pushes stack1 in stack, thus duplicating the second stage of stack. ALU performs alu_right, and in the stack is pusher the output of ALU.

```plaintext
over : begin
    pc_com = next ;
    alu_com = alu_right ;
    data_sel = alu ;
    stack_com = s_push ;
    data_mem = m_nop ;
end
```
The sequence of instructions:

```plaintext
over;
over;
```
duplicates the first two recordings in stack to be reused later in another stage of computation.

**Instruction zero** substitute the top of stack with 1, if its content is 0, or with 0 if the content is different from 0.

```plaintext
zero : begin pc_com = next ;
alu_com = alu_zero ;
data_sel = alu ;
stack_com = s_write ;
data_mem = m_nop ;
end
```

This instruction is used in conjunction with a conditioned jump (cjmpz or cjmpnz) to decide according to the value of stack0.

**Instruction jmp** adds to PS the signed value instruction[15:0].

```plaintext
jmp : begin pc_com = rel_jmp ;
alu_com = 4 ’bx ;
data_sel = 2 ’bx ;
stack_com = s_nop ;
data_mem = m_nop ;
end
```

This instruction is expressed as follows:

```plaintext
jmp <value>
```

where, `<value>` is expressed sometimes as an explicit signed integer, but usually as a label which takes a numerical value only when the program is assembled. For example:

```plaintext
jmp loop1;
```

**Instruction call** performs an absolute jump to the subroutine placed at the address instruction[15:0], and saves in tos the return address (ret_addr) which is pc + 1. The address saved in stack will be used by ret instruction to return the processor from the subroutine into the main program.

```plaintext
call : begin pc_com = abs_jmp ;
alu_com = 4 ’bx ;
data_sel = return ;
stack_com = s_write ;
data_mem = m_nop ;
end
```

The instruction is used, for example, as follows:

```plaintext
jmp subrt5;
```

where `subrt5` is the label of a certain subroutine.
**Instruction** cjmpz performs a relative jump if the content of tos is zero; else PC is incremented. The content of stack is unchanged. (A possible version of this instruction pops the tested value from the stack.)

\[
cjmpz : \text{if (test\_in == 32'b0)}
\begin{align*}
\text{begin} & \quad \text{pc\_com} = \text{small\_jmp} ; \\
& \quad \text{alu\_com} = 4\'bx ; \\
& \quad \text{data\_sel} = 2\'bx ; \\
& \quad \text{stack\_com} = \text{s\_nop} ; \\
& \quad \text{data\_mem} = \text{m\_nop} ; \\
\end{align*}
\begin{align*}
\text{end} & \quad \text{else} \\
\text{begin} & \quad \text{pc\_com} = \text{next} ; \\
& \quad \text{alu\_com} = 4\'bx ; \\
& \quad \text{data\_sel} = 2\'bx ; \\
& \quad \text{stack\_com} = \text{s\_nop} ; \\
& \quad \text{data\_mem} = \text{m\_nop} ; \\
\end{align*}
\]

The instruction is used, for example, as follows:

\[\text{jmp george;}\]

where `george` is a label to be converted in a signed 16-bit integer in the assembly process.

**Instruction** ret performs a jump from subroutine back into the main program using the address popped from tos.

\[
\text{ret} : \text{begin} \quad \text{pc\_com} = \text{ret\_jmp} ; \\
& \quad \text{alu\_com} = 4\'bx ; \\
& \quad \text{data\_sel} = 2\'bx ; \\
& \quad \text{stack\_com} = \text{s\_pop} ; \\
& \quad \text{data\_mem} = \text{m\_nop} ; \\
\end{align*}

The hardware resources of this Stack Processor permits up to 256 instructions to be defined. For this simple machine we do not need to define too many instructions. Therefore, a “smart” coding of instructions will allow minimizing the size of decoder. More, for some critical paths the depth of decoder can be also minimized, eventually reduced to zero. For example, maybe it is possible to set

\[
\begin{align*}
\text{alu\_com} &= \text{instruction}[19:16] \\
\text{data\_sel} &= \text{instruction}[21:20]
\end{align*}
\]

allowing the critical loop to be closed faster.

**10.2.5 Time performances**

Evaluating the time behavior of the just designed machine does not make us too happy. The main reason is provided by the fact that all the external connections are unbuffered.

All the three inputs, `instruction`, `data\_in`, `mem\_ready` must be received long time before the active edge of clock because their combinational path inside the Stack Processor are too deep. More, these paths are shared partially with the internal loops responsible for the maximum clock frequency. Therefore, optimizing the clock interferes with optimizing `t_{in\_req}`.

Similar comments apply to the output combinational paths.
The most disturbing propagation path is the combinational path going from inputs to outputs (for example: from instruction to data_mem_com). The impossibility to avoid $t_{in-out}$ make this design very unfriendly at the system level. Connecting this module together with a data memory a program memory and some input-output circuits will generate too many (restrictive) time dependencies.

This kind of approach can be useful only if it is strongly integrated with the design of the associated memories and interfaces in a module having all inputs and outputs strictly registered.

The previously described Stack Processor remains to be a very good bad example of a pure functionally centered design which ignores the basic electrical restrictions.

10.2.6 Concluding about our Stack Processor

The simple processor exemplified by Stack Processor is typical for a computational engine: it contains an simple working 3loop system – SALU – and another simple automaton – Program Fetch – both driven by a decoder to execute what is codded in each fetched instruction. Therefore, the resulting system is a 4th order one. This is not the solution! A lot of improvement are possible, and a lot of new features can be added. It is very useful to exemplify one of the main virtue of the fourth loop: the 4-loop processing. A processor with more than the minimal 3 loops is easiest to be controlled. In our cases the operands are automatically selected by the stack mechanism. Results a lot of advantages in control and some performance loss. But, the analysis of pros & cons is not a circuit design problem. It is a topics to be investigated in the computer architecture domain.

The main advantages of a stack machine is its simplicity. The operands are in each cycle already selected, because they are the first to recording in the top of stack. Results a simple instruction containing only two fields: op_code[7:0] and value[15:0].

The loop inside SALU is very short allowing a high clock frequency (if other loop do not impose a smaller one).

The main disadvantage of a stack machine is the stack discipline which sometimes adds new instructions in the code generated by the compiler.

Writing a compiler for this kind of machine is simple because the discipline in selecting the operands is high. The efficiency of the resulting code is debatable. Sometimes a longer sequence of operation is compensated by the higher frequency allowed by a stack architecture.

A real machine can adopt a more sophisticated stack in order to remove some limitation imposed by the restricted access imposed by the discipline.

10.3 Embedded computation

Now we are prepared to revisit the Chapter OUR FINAL TARGET in order to offer an optimal implementation for the small & simple system toyMachine. The main application for such a machine is in the domain of the embedded computation. The technology of embedded computation uses programmable machines of various complexity to implement by programming functions formerly implemented by big & complex circuits.

Instead of the behavioral description by the module toyMachine (see Figure 5.4) we are able to provide now a structural description. Even if the behavioral description offered by the module toyMachine is synthesisable will we see that the following structural version provides a half sized circuit.

10.3.1 The structural description of toyMachine

A structural description is supposed to be a detailed description which provide a hierarchical description of the design using on the “leafs of the tree” simple and optimal circuits. A structural description answers the question of “how”.
The top module

In the top module of the design – toyMachineStructure – there are two structures (see Figure 10.10):

**controlSection**: manages the instruction flow read from the program memory and executed, one per clock cycle, by the entire system; the specific control instructions are executed by this module using data, when needed, provided by the other modules ("dialog" bits for the stream flow, values from controlSection); the asynchronous inta signal constrains the specific action of jumping to the instruction addressed with the content of refFile[31]

**dataSection**: performs the functional aspect of computation, operating on data internally stored by the register file, or received from the external world; it generate also the output signals loading the output register outRegister with the results of the internal computation.

The block dataSection is a third order (3-loop) digital system having the third loop closed over the regFile through alu with carry (see Figure 10.11). The second loop is closed over alu and the carry flip-flop. The first loop in this section is closed in the latches used to build the module regFile and the flip-flop carry.

The block controlSection is a second order (2-loop) digital system, because the first loop is closed in the master-slave flip-flops of the (programCounter module, the second loop is closed over programCounter through pcMux inc and add (see Figure 10.11).
Thus, the *toyMachine* system is a fourth order digital system, the last loop being closed through *dataSection* and *controlSection*. The module *controlSection* sends to the *dataSection* the value *progAddr* as the return address from the sub-routine associated to the interrupt. The module *dataSection* sends back to the module *controlSection* the absolute jump address.

See in Figure 10.12 the code describing the top module. Unlike the module *toyMachine* (see Chapter *OUR FINAL TARGET*), which describe on one level design the behavior of *toyMachine*, the module *toyMachineStructure* is a pure structural description providing only the top level description of the same digital system. It contains two modules, one for each main sub-system of or design.

### The interrupt

There are many ways to solve the problem of the interrupt signal in a computing machine. The solutions are different depending on the way the signals *int* and *inta* are connected to the external systems. The solution provided here is the simplest one. It is supposed that both signals are synchronous with the *toyMachine* structure. This simple solution consists of a 2-state half-automaton (the one-bit register *intEnable* and the multiplexer *ieMux*).

Because the input *int* is considered synchronously generated with the system clock, the signal *inta* is combinational generated.

The next subsection provides an enhanced version of this module which is able to manage asynchronous *int* signal.

### The control section

This unit fetches in each clock cycle a new instruction from the program memory. The instruction is decoded locally for its use and is also sent for the use of the data unit. For each instruction there is a specific way to use the content of the program counter in order to compute the address of the next instruction. For data and interrupt instructions (see Figure 5.3) the next instruction is always fetched form the address *programCounter + 1*. For the control instructions (see Figure 5.3) there are different modes for each instruction. The internal structure of the module *controlSection* is designed to provide the specific modes of computing the next value for the program counter.

The multiplexers *pcMux* from the control section (see Figure 10.11) is used to select the next value of the program counter, providing the value of *progAddr*, as follows:

- program counter keep its own value for the *halt* instruction or in the wait instructions for input or output to become ready
- program counter is incremented for the linear part of the program
- program counter is added to the *immValue* provided by the current instruction
- program counter is set to the value provided by *regFile[leftAddr]* for unconditioned jump

The selection bits for *pcMux* are generated by the *contrDecode*. It uses for generating the selections for the multiplexers *opCode* from *instruction*, *asyncInta*, the content of *regFile[leftAddr]* and the input signals *readyIn*, *readyOut*.

The only complex module in the control section is the combinational circuit described in *contrDecode* (see Figure 10.14). The type *reg* in this description must be understood as a variable. The actual structure of a register is not generated.
Figure 10.11:
module toyMachineStructure
  ( input [15:0] inStream ,
    input  readyIn   ,
    output readIn   ,
    output [15:0] outStream ,
    input  readyOut  ,
    output writeOut ,
    input  int       ,
    output inta     ,
    output [31:0] dataAddr, dataOut, progAddr ,
    output write ,
    input [31:0] dataIn, instruction ,
    input  reset, clock         ); // 321 LUTs, 205 MHz
wire [31:0] leftOp, immValue, programCounter;
wire [5:0] opCode ;
wire [4:0] destAddr, leftAddr, rightAddr ;
assign opCode     = instruction[31:26] ;
assign destAddr   = instruction[25:21] ;
assign leftAddr   = instruction[20:16] ;
assign rightAddr  = instruction[15:11] ;
assign immValue   = {{16{instruction[15]}}, instruction[15:0]};
dataSection dataSection(inStream ,
  readyIn   ,
  readIn   ,
  outStream ,
  readyOut  ,
  writeOut ,
  dataAddr ,
  dataOut ,
  write ,
  dataIn ,
  programCounter ,
  opCode ,
  destAddr ,
  leftAddr ,
  rightAddr ,
  immValue ,
  inta ,
  clock ) ;
controlSection controlSection( int    ,
  inta    ,
  progAddr ,
  programCounter ,
  dataAddr ,
  opCode ,
  immValue ,
  readyIn ,
  readyOut ,
  reset ,
  clock ) ;
endmodule

Figure 10.12: The top module toyMachineStructure.
module controlSection( input int ,
output inta ,
output [31:0] progAddr ,
output [31:0] programCounter ,
input [31:0] dataAddr ,
input [5:0] opCode ,
input [31:0] immValue ,
input readyIn ,
input readyOut ,
input reset ,
input clock );

reg [31:0] programCounter ;
reg intEnable ;
wire [1:0] nextPcSel, nextIESel;
wire nextIE ;
assign inta = intEnable & int;

contrDecode
  contrDecode(opCode ,
dataAddr ,
inta ,
readyIn ,
readyOut ,
nextPcSel ,
nextIESel );

always @(posedge clock) if (reset) begin
  programCounter <= 32'b0;
  intEnable <= 1'b0 ;
end
else begin
  programCounter <= progAddr ;
  intEnable <= nextIE ;
end

mux4_32 pcMux( .out(progAddr ),
  .in0(programCounter ),
  .in1(programCounter + 1 ),
  .in2(programCounter + immValue ),
  .in3(dataAddr ),
  .sel(nextPcSel ));

mux4_1 ieMux( .out(nextIE ),
  .in0(intEnable ),
  .in1(1'b0 ),
  .in2(1'b1 ),
  .in3(1'b0 ),
  .sel(nextIESel ));

endmodule

Figure 10.13: The module controlSection.
module contrDecode( 
    input [5:0] opCode ,
    input [31:0] dataAddr ,
    input inta ,
    input readyIn ,
    input readyOut ,
    output reg [1:0] nextPcSel ,
    output reg [1:0] nextIESel);

`include "0_toyMachineArchitecture.v"

always @(*) if (inta) nextIESel = 2'b10 ;
else if (opCode == ei) nextIESel = 2'b01 ;
else if (opCode == di) nextIESel = 2'b10 ;
else nextIESel = 2'b00 ;

always @(*) if (inta)
    begin
        endcase
    endmodule

Figure 10.14: The module contrDecode.

The data section

The module dataSection includes mainly the data storage resources and the combinational circuits allowing the execution of each data instruction in one clock cycle.

Data is stored in the register file, regFile, which allows to read two variable as operands for the current instruction, selected by leftAddr and rightAddr, and to store the result of the current instruction to the location selected by desrAddr (except the case when inta = 1 forces reading leftOp form the location 30, to be used as absolute jump address, and loading the location 31 with the current value of programCounter).

The arithmetic-logic unit, alu, operate in each clock cycle on the operands received from the two outputs of the register file: leftOp and rightOp. The operation code is given directly from the output of the dataDecode block described by the module dataDecode (see Figure 10.20).

The input of the register file is provided from the alu output and from other four sources:

- inRegister: because the input bits can be submitted to arithmetic and logic processing only if they are stored in the register file first
- immValue: is used to generate immediate values for the purpose of the program
- dataIn: data provided by the external data memory addressed by leftOp
module dataSection( input [15:0] inStream ,
   input  readyIn ,
   output readIn ,
   output [15:0] outStream ,
   input  readyOut ,
   output writeOut ,
   output [31:0] dataAddr ,
   output [31:0] dataOut ,
   output write ,
   input [31:0] dataIn ,
   input [31:0] programCounter ,
   input [5:0] opCode ,
   input [4:0] destAddr ,
   input [4:0] leftAddr ,
   input [4:0] rightAddr ,
   input [31:0] immValue ,
   input  inta ,
   input  clock );

reg  [15:0] inRegister ;
reg  [15:0] outRegister ;
reg   carry ;
wire  [31:0] result ;
wire   carryOut ;
wire  [31:0] leftOp ;
wire  [31:0] rightOp ;
wire   [1:0] arithLogOp ;
wire   [2:0] resultSel ;
wire   writeBack ;
wire   inRegEnable ;
wire   outRegEnable ;
wire   carryEnable ;
assign dataAddr  = leftOp ;
assign dataOut  = leftOp ;
assign outStream  = outRegister ;
dataDecode
   dataDecode
   (arithLogOp, resultSel, writeBack, inRegEnable, outRegEnable,
   carryEnable, readIn, writeOut, write, opCode, readyIn, readyOut, inta );
always @(posedge clock) begin if (inRegEnable ) inRegister <= inStream ;
   if (outRegEnable) outRegister <= leftOp[15:0] ;
   if (carryEnable) carry  <= carryOut ;
   end
regFile
   regFile(.destAddr  (inta ? 5'bi11110 : destAddr ),
       .writeBack (writeBack ),
       .leftAddr  (inta ? 5'bi11111 : leftAddr ),
       .rightAddr (rightAddr ),
       .in  (result ),
       .leftOut (leftOp ),
       .rightOut (rightOp ),
       .clock (clock ));
alu alu(result, carryOut, leftOp, rightOp, carry, inRegister ,
       programCounter, dataIn, immValue, arithLogOp, resultSel );
endmodule

Figure 10.15: The module dataSection.
10.3. EMBEDDED COMPUTATION

module regFile( input [4:0] destAddr ,
input writBack ,
input [4:0] leftAddr ,
input [4:0] rightAddr ,
input [31:0] in ,
output [31:0] leftOut ,
output [31:0] rightOut ,
input clock );

reg [31:0] regFile[0:31] ;

always @(posedge clock) if (writeBack) regFile[destAddr] <= in ;

assign leftOut = regFile[leftAddr] ;
assign rightOut = regFile[rightAddr];
endmodule

Figure 10.16: The module regFile.

- programCounter: is saved as the “return” address to be used after running the program started by the acknowledged interrupt

The first three of these inputs are selected according to the current instructions by the selection code resultSel generated by the module dataDecode for the multiplexor resultMux. The last one is forced at the input of the register file by the occurrence of the signal inta.

The register file description uses the code presented in the subsection Register file. Only the sizes are adapted to our design (see Figure 10.16.

The module called alu (see Figure 10.17) performs the arithmetic-logic functions of our small instruction set. Because the current synthesis tools are able to synthesize very efficiently uniform arithmetic and logic circuits, this Verilog contains a behavioral description.

The dataDecode block, described in our design by the Verilog module dataDecoder, takes the opCode field from instruction, the dialog signals, readyIn and readyOut, and inta and trans-codes them. This is the only complex module from the data section.

Multiplexors

The design of toyMachine uses a lot of multiplexors. Their description is part of the project.

As for the usual functions from an ALU, or small combinational circuits, for multiplexors behavioral descriptions work very well because the software synthesis tools are enough “smart” to “know” how to provide optimal solutions.

Concluding about toyMachine

For the same system – toyMachine – we have now two distinct descriptions: toyMachine, the initial behavioral description (see Chapter OUR FINAL TARGET), and the structural description toyMachineStructure just laid down in this subsection. Both descriptions, the structural and the behavioral, are synthesisable, but the resulting structures are very different.
module alu( output [31:0] result , 
    output carryOut , 
    input [31:0] leftOp , 
    input [31:0] rightOp , 
    input carry , 
    input [15:0] inRegister , 
    input [31:0] programCounter , 
    input [31:0] dataIn , 
    input [31:0] immValue , 
    input [1:0] arithLogOp , 
    input [2:0] resultSel );

wire [31:0] logicResult ;
wire [31:0] arithResult ;

logicModule
logicModule(leftOp , 
    rightOp , 
    arithLogOp , 
    logicResult );

arithModule
arithModule(leftOp , 
    rightOp , 
    carry , 
    arithLogOp , 
    arithResult , 
    carryOut );

mux8_32 resultMux( .out(result ) , 
    .in0(arithResult ) , 
    .in1(logicResult ) , 
    .in2({leftOp[31], leftOp[31:1]} ) , 
    .in3(immValue ) , 
    .in4({immValue[15:0], leftOp[15:0]} ) , 
    .in5({{16{inRegister[15]}}, inRegister} ) , 
    .in6(programCounter ) , 
    .in7(dataIn ) , 
    .sel(resultSel ) );

endmodule

Figure 10.17: The module alu.
module arithModule( input [31:0] leftOp ,
  input [31:0] rightOp ,
  input carry ,
  input [1:0] arithLogOp ,
  output [31:0] arithResult ,
  output carryOut );

  assign {carryOut, arithResult} = leftOp + (rightOp ^ {32{arithLogOp[0]}}) +
    (arithLogOp[1] & (arithLogOp[0] & carry));
endmodule

module logicModule( input [31:0] leftOp ,
  input [31:0] rightOp ,
  input [1:0] arithLogOp ,
  output reg [31:0] logicResult );

  always @(*) case(arithLogOp)
    2'b00: logicResult = ~leftOp ;
    2'b01: logicResult = leftOp & rightOp ;
    2'b10: logicResult = leftOp | rightOp ;
    2'b11: logicResult = leftOp ^ rightOp ;
  endcase
endmodule

Figure 10.18: The module arithModule.

Figure 10.19: The module logicModule.
module dataDecode(  
  output reg [1:0] arithLogOp,  
  output reg [2:0] resultSel,  
  output reg writeBack,  
  output reg inRegEnable,  
  output reg outRegEnable,  
  output reg carryEnable,  
  output reg readIn,  
  output reg writeOut,  
  output reg write,  
  input [5:0] opCode,  
  input readyIn,  
  input readyOut,  
  input inta );

'include "0_toyMachineArchitecture.v"

always @(*) begin  
  arithLogOp  = 2'b00 ;  
  resultSel  = 3'b000;  
  writeBack  = 1'b0 ;  
  inRegEnable  = 1'b0 ;  
  outRegEnable  = 1'b0 ;  
  carryEnable  = 1'b0 ;  
  readIn  = 1'b0 ;  
  writeOut  = 1'b0 ;  
  write  = 1'b0 ;  
  if (inta) begin  
    resultSel  = 3'b110;  
    writeBack  = 1'b1 ;  
  end  
else case(opCode)
  add : begin  
    arithLogOp  = 2'b00 ;  
    resultSel  = 3'b000;  
    writeBack  = 1'b1 ;  
    carryEnable  = 1'b1 ;  
  end
  sub : begin  
    arithLogOp  = 2'b01 ;  
    resultSel  = 3'b000;  
    writeBack  = 1'b1 ;  
    carryEnable  = 1'b1 ;  
  end
  addc : begin  
    arithLogOp  = 2'b10 ;  
    resultSel  = 3'b000;  
    writeBack  = 1'b1 ;  
    carryEnable  = 1'b1 ;  
  end
  subc : begin  
    arithLogOp  = 2'b11 ;  
    resultSel  = 3'b000;  
    writeBack  = 1'b1 ;  
    carryEnable  = 1'b1 ;  
  end
  ashr : begin  
    resultSel  = 3'b010;  
    writeBack  = 1'b1 ;  
  end

endcase

end

Figure 10.20: The module dataDecode.
neg : begin arithLogOp = 2'b00;
    resultSel = 3'b001;
    writeBack = 1'b1;
end

bwand : begin arithLogOp = 2'b01;
    resultSel = 3'b001;
    writeBack = 1'b1;
end

bwor : begin arithLogOp = 2'b10;
    resultSel = 3'b001;
    writeBack = 1'b1;
end

bwxor : begin arithLogOp = 2'b11;
    resultSel = 3'b001;
    writeBack = 1'b1;
end

val : begin resultSel = 3'b011;
    writeBack = 1'b1;
end

hval : begin resultSel = 3'b100;
    writeBack = 1'b1;
end

get : begin resultSel = 3'b101;
    writeBack = 1'b1;
end

send : outRegEnable = 1'b1;
receive : if (readyIn)
    begin inRegEnable = 1'b1;
        readIn = 1'b1;
    end
issue : if (readyOut) writeOut = 1'b1;
datawr : write = 1'b1;
datard : begin resultSel = 3'b111;
         writeBack = 1'b1;
end
default begin arithLogOp = 2'b00;
    resultSel = 3'b000;
    writeBack = 1'b0;
    inRegEnable = 1'b0;
    outRegEnable = 1'b0;
    carryEnable = 1'b0;
    readIn = 1'b0;
    writeOut = 1'b0;
    write = 1'b0;
endcase
endmodule

Figure 10.21: The module dataDecode (continuation).
module mux4_32(output reg [31:0] out ,
                 input [31:0] in0 ,
                 input [31:0] in1 ,
                 input [31:0] in2 ,
                 input [31:0] in3 ,
                 input [1:0] sel);
    always @(*) case(sel)
        2'b00: out = in0;
        2'b01: out = in1;
        2'b10: out = in2;
        2'b11: out = in3;
    endcase
endmodule

module mux4_1( output reg out ,
               input in0 ,
               input in1 ,
               input in2 ,
               input in3 ,
               input [1:0] sel);
    always @(*) case(sel)
        2'b00: out = in0;
        2'b01: out = in1;
        2'b10: out = in2;
        2'b11: out = in3;
    endcase
endmodule

module mux8_32(output reg [31:0] out ,
                input [31:0] in0 ,
                input [31:0] in1 ,
                input [31:0] in2 ,
                input [31:0] in3 ,
                input [31:0] in4 ,
                input [31:0] in5 ,
                input [31:0] in6 ,
                input [31:0] in7 ,
                input [3:0] sel);
    always @(*) case(sel)
        3'b000: out = in0;
        3'b001: out = in1;
        3'b010: out = in2;
        3'b011: out = in3;
        3'b100: out = in4;
        3'b101: out = in5;
        3'b110: out = in6;
        3'b111: out = in7;
    endcase
endmodule

Figure 10.22: The multiplexor modules multiplexors.
The synthesis of toyMachine design provides a number of components 5.85 times bigger than the synthesis of the module toyMachineStructure. A detailed description (about 7105 symbols, without spaces) provided a smallest structure then the structure provided by the behavioral description (about 3083 symbols, without spaces).

The actual structure generated by the behavioral description is not only bigger, but it is completely unstructured. The structured version provided by the alternative design is easy to understand, to debug and to optimize.

10.3.2 Interrupt automaton: the asynchronous version

Sometimes for the interrupt automaton a more rigorous solution is requested. In the already provided solution the int signal must be stable until inta is activated. In many systems this is an unacceptable restriction. Another restriction is the synchronous switch of int.

This new version for the interrupt automaton accepts an asynchronous int signal having any width exceeding the period of the clock. The flow chart describing the automaton is in Figure 10.23. It has 4 states:

- **dis**: the initial state of the automaton when the interrupt action is disabled
- **en**: the state when the interrupt action is enabled
- **mem**: is the state memorizing the occurrence of an interrupt when interrupt is disabled
- **inta**: is the acknowledge state.

The input signals are:

- **int**: is the asynchronous interrupt signal
- **ei**: is a synchronous bit resulting from the decode of the instruction ei (enable interrupt)
- **di**: is a synchronous bit resulting from the decode of the instruction di (disable interrupt)

The output signal is asyncInta. It is in fact a synchronous hazardous signal which will be synchronized using a D–FF.

Because int is asynchronous it must be used to switch the automaton in another state in which asyncInta will be eventually generated.

The state coding style applied for this automaton is imposed by a asynchronous int signal. It will be of the reduced dependency by the asynchronous input variable int. Let us try first the following binary codes (see the codes in square brackets in Figure 10.23) for the four states of the automaton:

- **dis**: \[ Q_1Q_0 = 00 \]
- **en**: \[ Q_1Q_0 = 11 \]
- **mem**: \[ Q_1Q_0 = 01 \]
- **inta**: \[ Q_1Q_0 = 10 \]

The critical transitions are from the states **dis** and **en**, where the asynchronous input int is tested. Therefore, the transitions from these two states takes place as follows:
Figure 10.23: **Interrupt automaton for a limited width and an asynchronous int signal.**

- from state `dis = 00`: if `(ei = 0)` then `{Q_1^+, Q_0^+} = \{0, int\}; else `{Q_1^+, Q_0^+} = \{1, int'\}; therefore: `{Q_1^+, Q_0^+} = \{ei, ei \oplus int\}

- from state `en = 11`: if `(di = 0)` then `{Q_1^+, Q_0^+} = \{1, int'\}; else `{Q_1^+, Q_0^+} = \{0, int\}; therefore: `{Q_1^+, Q_0^+} = \{di', di' \oplus int\}

Therefore, the transitions triggered by the asynchronous input int influence always only one state bit.

For an implementation with registers results the following equations for the state transition and output functions:

\[
Q_1^{+1} = Q_1 Q_0 d_1' + Q_1' Q_0 e i
\]
\[
Q_0^{+1} = Q_1 Q_0 (d_1' \oplus int) + Q_1' Q_0 e i + Q_1' Q_0 (e i \oplus int)
\]
\[
asyncInta = Q_1 Q_0' + Q_1' Q_0 e i
\]

We are not very happy about the resulting circuits because the size is too big to my taste. Deserve to try another equivalent state coding, preserving the condition that the transitions depending on the int input are reduced dependency type. The second coding proposal is (see the un-bracketed codes in Figure 10.23):

- `dis : Q_1 Q_0 = 00`
- `en : Q_1 Q_0 = 10`
- `mem : Q_1 Q_0 = 01`
module interruptAutomaton(input int ,
input ei ,
input di ,
output regasyncInt,
input reset ,
input clock );

reg [1:0] state  ;
reg [1:0] nextState;

always @(posedge clock) if (reset) state <= 0  ;
else state <= nextState;

always @(int or ei or di or state)
case(state)
  2'b00: if (int) if (ei) {nextState, asyncInt} = 3'b11_0;
         else {nextState, asyncInt} = 3'b01_0;
  2'b01: if (ei) {nextState, asyncInt} = 3'b00_1;
         else {nextState, asyncInt} = 3'b01_0;
  2'b10: if (int) if (di) {nextState, asyncInt} = 3'b01_0;
         else {nextState, asyncInt} = 3'b11_0;
  2'b11: {nextState, asyncInt} = 3'b00_1;
endcase
endmodule

Figure 10.24: The module interruptAutomaton.

inta : $Q_1Q_0 = 11$

The new state transition functions are:

$Q_1^+ = Q_1Q_0di' + Q_1'Q_0ei$

$Q_0^+ = Q_0' + Q_0'Q_0ei'$

The Verilog behavioral description for this version is presented in Figure 10.24.

If we make another step re-designing the loop for an "intelligent" JK register, then results for the loop the following expressions:

$J_1 = Q_0ei$

$K_1 = di + Q_0$

$J_0 = int$

$K_0 = Q_1 + ei$

and for the output transition:

asyncInta = $Q_0(Q_1 + ei) = Q_0K_0$
A total of 4 2-input gates for the complex part of the automaton. The final count: 2 JK-FFs, 2 ANDs, 2 ORs. Not bad! The structural description for this version is presented in Figure 10.25 and in Figure 10.26.

```verilog
module interruptAutomaton(input int ,
    input ei , di,
    output asyncInta,
    input reset ,
    input clock);
wire q1, q0, notq1, notq0;
JKflipFlop ff1(.Q (q1   ),
    .notQ (notq1  ),
    .J (notq0 & ei),
    .K (di | q0  ),
    .reset(reset ),
    .clock(clock ));
JKflipFlop ff0(.Q(q0),
    .notQ (notq0  ),
    .J (int   ),
    .K (q1 | ei),
    .reset(reset ),
    .clock(clock ));
assign asyncInta = q0 & (q1 | ei);
endmodule
```

Figure 10.25: The structural description of the module interruptAutomaton implemented using JK-FFs.

```verilog
module JKflipFlop(output reg Q ,
    output notQ ,
    input J , K,
    input reset,
    input clock);
assign notQ = ~Q;
always @(posedge clock) if (reset) Q <= 0 ;
    else Q <= J & notQ | ~K & Q;
endmodule
```

Figure 10.26: The module JKflipFlop.

The synthesis process will provide a very small circuit with the complex part implemented using only 4 gates. The module interruptUnit in the toyMachine design must be redesigned including the just presented module interruptAutomaton. The size of the overall project will increase, but the interrupt mechanism will work with less electrical restrictions imposed to the external connections.

10.4 Problems

Problem 10.1 Interpretative processor with distinct program counter block.
10.5 Projects

Project 10.1
Chapter 11

# SELF-ORGANIZING STRUCTURES: N-th order digital systems

In the previous chapter
the concept of computer, as at least four-loop system, was introduced. The basic part of the section
is contained in the Problems section. Adding few loops the functionality of the system remains the
same - basic computation - the only effect is optimizing area, power, speed.

In this chapter
the self-organizing systems are supported by a cellular approach consisting in n-loop systems.
The main structure discussed are:

- the stack memory, as the simplest n-loop system
- the cellular automata, as the simplest self-organizing system
- fractal structures, as “non-uniform” network of processors

In the next chapter
we make only the first step in closing a new kind of loops over an n-order system, thus introducing
the new category of systems with global loops.
Von Neumann’s architecture is supported by a structure which consists of two distinct subsystems. The first is a processor and the second is a memory that stores the bits to be processed. In order to be processed a bit must be carried from the memory to the processor and many times back, from the processor to the memory. Too much time is wasted and many structures are involved only to move bits inside the computing machine. The functional development in the physical part of a digital systems stopped when this universal model was adopted. In the same time the performances of the computation process are theoretically limited. All the sort of parallelism pay tribute to this style that is sequentially founded. We have only one machine, each bit must be accessed, processed and after that restored. This “ritual” stopped the growing process of digital machines around the fifth order. There are a small number of useful systems having more than five loops.

The number of loops can become very large if we give up this model and we have the nerve to store “each bit” near its own “processor”. A strange, but maybe a winning solution is to “interleave” the processing elements with the storage circuits [Moto-Oka ‘82]. Many of us believe that this is a more “natural” solution. Until now this way is only a beautiful promise, but this way deserves more attention.

Definition 11.1 A digital system, DS, belongs to n-OS if having the size in $O(f(n))$ contains a number of internal loop in $O(n)$.

A paradigmatic example of n-loop digital system is the cellular automaton (CA). Many applications of CA model self-organizing systems.

For the beginning, as a simple introduction, the stack memory is presented in a version belonging to n-OS. The next subject will be a new type of memory which tightly interleaves the storage elements with processing circuits: the Connex memory. This kind of memory allows fine grain deep parallel processes in computation. We end with the eco-chip, a spectacular application of the two-dimensional cellular automata enhanced with the Connex memory’s functions.

The systems belonging to n-OS support efficiently different mechanisms related to some parallel computation models. Thus, there are many chances to ground true parallel computing architecture using such kind of circuits.

11.1 N-counter extended finite automata

11.2 Push-Down Stack as n-OS

There are only a few “exotic” structures that are implemented as digital systems with a great number of loops. One of these is the stack function that needs at least two loops to be realized, as a system in 2-OS (reversible counter & RAM serially composed). There is another, more uniform solution for implementing the push-down stack function or LIFO (last-in first-out) memory. This solution uses a simple, i.e., recursive defined, structure.

Definition 11.2 The n-level push-down stack, LIFO$_n$, is built serial connecting a LIFO$_{n-1}$ with a LIFO$_1$ as in Figure 11.1. The one level push-down stack is a register, R$_m$, loop connected with $m \times$ MUX$_2$, so as:

$S_1S_0 = 00$ means: **no op** - the content of the register does not change

$S_1S_0 = 01$ means: **push** - the register is loaded with the input value, IN

$S_1S_0 = 10$ means: **pop** - the register is loaded with the extension input value, EXTIN.

It is evident that LIFO$_n$ is a bi-directional serial-parallel shift register. Because the content of the serial-parallel register shifts in both directions each R$_m$ is contained in two kind of loops:

- through its own MUX for no op function
- through two successive LIFO$_1$

Thus, LIFO$_1$ is a 2-OS, LIFO$_2$ is a 3-OS, LIFO$_3$ is a 4-OS, …, LIFO$_i$ is a $(i+1)$OS, ….

The push-down stack implemented as a bi-directional serial-parallel register is an example of digital system having the order related with the size. Indeed: LIFO$_{n-1}$ is a n – OS.
11.3 Cellular automata

A cellular automaton consists of a regular grid of cells. Each cell has a finite number of states. The grid has a finite number of dimensions, usually no more than three. The transition function of each cell is defined in a constant neighborhood. Usually, the next state of the cell depends on its own state and the states of the adjacent cells.

11.3.1 General definitions

The linear cellular automaton

**Definition 11.3** The one-dimension cellular automaton is linear array of \( n \) identical cells, where each cell is connected in a constant neighborhood of +/- \( m \) cells, see Figure 11.2 for \( m = 1 \). Each cell is a \( s \)-state finite automaton.

**Definition 11.4** An elementary cellular automaton is a one-dimension cellular automaton with \( m = 1 \) and \( s = 2 \). The transition function of each automaton is a three-input Boolean function defined by the decimally expressed associated Boolean vector.

**Example 11.1** The Boolean vector of the three-input function

\[
f(x_2, x_1, x_0) = x_2 \oplus (x_1 + x_0)
\]

is:

00011110

and defines the transition rule 30.

**Definition 11.5** The Verilog definition of the elementary cellular automaton is:

```verilog
module eCellAut #(parameter n = 127) // n-cell elementary cellular automaton
  ( output [n-1:0] out ,
    input [7:0] func,   // the Boolean vector for the transition rule
    input [n-1:0] init,  // used to initialize the cellular automaton
    input rst ,      // loads the initial state of the cellular automaton
    input clk );
```

Figure 11.1: The LIFO structure as \( n - OS \)
Figure 11.2: **Cellular automaton.** a. One-dimension cellular automaton. b. Two-dimension cellular automaton with von Neumann neighborhood. c. Two-dimension cellular automaton with Moore neighborhood. d. Two-dimension cellular automaton with toroidal shape. e. Two-dimension cellular automaton with rotated toroidal shape.
genvar i;
genvar for (i=0; i<n; i=i+1) begin: C
eCell eCell(.out (out[i]
).func (func
).init (init[i]
).in0 ((i==0) ? out[n-1] : out[i-1]
).in1 ((i==n-1) ? out[0] : out[i+1]
).rst (rst
).clk (clk ));
endgenerate
endmodule

where the elementary cell, eCell, is:

module eCell // elementary cell
(output reg out,
input [7:0] func,
input init,
in0 , // input form the previous cell
input in1 , // input from the next cell
input rst,
input clk );
always @(posedge clk) if (rst) out <= init ;
else out <= func[{in1, out, in0}];
endmodule

Example 11.2 The elementary cellular automaton characterized by the rule 90 (01011010) provides, starting from the initial state 1'b1 << n/2, the behavior represented in Figure 11.3, where the sequence of lines of bits represent the sequence of the states of the cellular automaton starting from the initial state.

The shape generated by the elementary cellular automaton 90 is the Sierpinski triangle or the Sierpinski Sieve. It is a fractal named after the Polish mathematician Waclaw Sierpinski who first described it in 1915.

Example 11.3 The elementary cellular automaton characterized by the rule 30 (00011110) provides, starting from the initial state 1'b1 << n/2, the behavior represented in Figure 11.4, where the sequence of lines of bits represent the sequence of the states of the cellular automaton starting from the initial state.
Figure 11.3:
Figure 11.4:
The two-dimension cellular automaton

**Definition 11.6** The two-dimension cellular automaton consists of a two-dimension array of identical cells, where each cell is connected in a constant neighborhood, see Figure 11.2b (the von Neumann neighborhood) and 11.2c (the Moore neighborhood). Each cell is a $s$-state finite automaton.

There are also many ways of connecting the border cells. The simplest one is to connect them to ground. Another is close the array so as the surface takes a toroidal shape (see Figure 11.2d). A more complex form is possible if we intend to preserve also a linear connection between the cells. Results a twisted toroidal shape (see Figure 11.2e).

**Definition 11.7** The Verilog definition of the two-dimension elementary cellular automaton with a toroidal shape (Figure 11.2d) is:

```verilog
module eCellAut4 #(parameter n = 8) // n*n-cell two-dimension cellular automaton
  ( output [n*n-1:0] out ,
    input [31:0] func, // the Boolean vector for the transition rule
    input [n*n-1:0] init, // used to initialize the cellular automaton
    input rst , // loads the initial state of the cellular automaton
    input clk );

genvar i;
genenerate for (i=0; i<n*n; i=i+1) begin: C
  eCell4 eCell4( .out (out[i] ),
    .func (func ),
    .init (init[i] ),
    .in0 (out[(i/n)*n+(i-((i/n)*n)+n-1)%n] ), // east
    .in1 (out[(i/n)*n+(i-((i/n)*n)+1)%n] ), // west
    .in2 (out[(i+n*n-n)%(n*n)] ), // south
    .in3 (out[(i+n)%(n*n)] ), // north
    .rst (rst ),
    .clk (clk ) );
endgenerate
endmodule

where the elementary cell, eCell4, is:

```verilog
module eCell4 // 4-input elementary cell
  ( output reg out ,
    input [31:0] func,
    input init, //
    input in0 , // north connection
    input in1 , // east connection
    input in2 , // south connection
    input in3 , // west connection
    input rst ,
    input clk );

always @(posedge clk) if (rst) out <= init ;
else out <= func[{in3, in2, out, in1, in0}] ;
endmodule
```
Example 11.4 Let be a $8 \times 8$ cellular automaton with a von Neumann neighborhood and a toroidal shape. The cells are 2-state automata. The transition function is a 5-input Boolean OR, and the initial state is state 1 in the bottom right cell and 0 the the rest of cells. The system will evolve until all the cells will switch in the state 1. Figure 11.5 represents the 8-step evolution from the initial state to the final state.

$$
\begin{array}{cccccccc}
00000000 & 00000001 & 10000111 & 11011111 & 11111111 & 11111111 & 11111111 & 11111111 \\
00000000 & 00000000 & 10000011 & 11000111 & 11101111 & 11111111 & 11111111 & 11111111 \\
00000000 & 00000000 & 00000000 & 10000111 & 11000111 & 11101111 & 11111111 & 11111111 \\
00000000 & 00000000 & 00000000 & 00000000 & 10000011 & 11000111 & 11101111 & 11111111 \\
00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 10000111 & 11000111 & 11101111 \\
00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 11000111 & 11101111 \\
00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 11101111 \\
00000001 & 10000011 & 11000111 & 11101111 & 11111111 & 11111111 & 11111111 & 11111111 \\
00000001 & 10000011 & 11000111 & 11101111 & 11111111 & 11111111 & 11111111 & 11111111 \\
\end{array}
$$

Figure 11.5:

Definition 11.8 The Verilog definition of the two-dimension elementary cellular automaton with linearly connected cells (Figure 11.2e) is:

```verilog
template eCellAut4L #(parameter n = 8) // two-dimension cellular automaton
    output [n*n-1:0] out ,
    input [31:0] func, // the Boolean vector for the transition rule
    input [n*n-1:0] init, // used to initialize the cellular automaton
    input rst , // loads the initial state of the cellular automaton
    input clk );

genvar i;
genenerate for (i=0; i<n*n; i=i+1) begin: C
    eCell4 eCell4( .out (out[i]
        .func (func
        .init (init[i]
        .in0 (out[(i+n*n-1)%n*n]) // east
        .in1 (out[(i+1)%n*n]) // west
        .in2 (out[(i+n*n-n)%n*n]) // south
        .in3 (out[(i+n)%n*n]) // north
        .rst (rst
        .clk (clk
        ));
end
generate
endmodule
```

where the elementary cell, eCell4, is the same as in the previous definition.

Example 11.5 Let us do the same for the two-dimension elementary cellular automaton with linearly connected cells (Figure 11.2e). The insertion of 1s in all the cells is done now in 7 steps. See Figure 11.6.

Looks like a twisted toroidal shape offers a better neighborhood than a simple toroidal shape.
CHAPTER 11. SELF-ORGANIZING STRUCTURES: N-TH ORDER DIGITAL SYSTEMS

11.3.2 Applications

Figure 11.6:
11.4 Systolic systems

Leiserson’s systolic sorter. The initial state: in each cell = \infty. For no operation: \text{in1} = +\infty, \text{in2} = -\infty. To insert the value v: \text{in1} = v, \text{in2} = -\infty. For extract: \text{in1} = \text{in2} = +\infty.

![Diagram of systolic sorter](image)

Figure 11.7: **Systolic sorter.** a. The internal structure of cell. b. The logic symbol of cell. c. The organization of the systolic sorter.

```verilog
module systolicSorterCell #(parameter n=8)(input [n-1:0] a, b, c, output reg [n-1:0] x, y, z, input rst, ck);
wire [n-1:0] a1, b1 ; // sorter's first level outputs
wire [n-1:0] a2, c2 ; // sorter's second level outputs
wire [n-1:0] b3, c3 ; // sorter's third level outputs
assign a1 = (a < b) ? a : b ;
assign b1 = (a < b) ? b : a ;
assign a2 = (a1 < c) ? a1 : c ;
assign c2 = (a1 < c) ? c : a1 ;
assign b3 = (b1 < c2) ? b1 : c2 ;
assign c3 = (b1 < c2) ? c2 : b1 ;

always @(ck or rst or a2 or b3 or c3)
    if (rst & ck) begin x = {n{1'b1}} ;
y = {n{1'b1}} ;
z = {n{1'b1}} ;
    end

    else if (ck) begin x = a2 ;
y = b3 ;
z = c3 ;
    end
endmodule

module systolicSorter #(parameter n=8, m=7)( output [n-1:0] out, input [n-1:0] in1, in2, input rst, ck1, ck2);
wire [n-1:0] x[0:m];
wire [n-1:0] y[0:m-1];
wire [n-1:0] z[0:m-1];
```
assign y[0] = in1;
assign z[0] = in2;
assign out = x[1];
assign x[m] = {n{1’b1}};

genvar i;
generate for(i=1; i<m; i=i+1) begin: C
    systolicSorterCell systolicCell(.a (x[i+1]),
        .b (y[i-1]),
        .c (z[i-1]),
        .x (x[i]),
        .y (y[i]),
        .z (z[i]),
        .rst(rst),
        .ck (((i/2)*2 == i) ? ck2 : ck1));
end
endgenerate
endmodule

module systolicSorterSim #(parameter n=8);
    reg ck1, ck2, rst;
    reg [n-1:0] in1, in2;
    wire [n-1:0] out;

    initial begin
        ck1 = 0;
        forever begin
            #3 ck1 = 1;
            #1 ck1 = 0;
        end
    end

    initial begin
        ck2 = 0;
        #2 ck2 = 0;
        forever begin
            #3 ck2 = 1;
            #1 ck2 = 0;
        end
    end

    initial begin
        rst = 1;
        in2 = 0;
        in1 = 8’b1000;
        #8 rst = 0;
        #4 in1 = 8’b0010;
        #4 in1 = 8’b0100;
        #4 in1 = 8’b0010;
        #4 in1 = 8’b0001;
        #4 in1 = 8’b11111111;
        in2 = 8’b11111111;
        #30 $stop;
    end

systolicSorter dut( out,
    in1, in2,
11.4. SYSTOLIC SYSTEMS

```
rst, ck1, ck2);

initial

$monitor("time = %d ck1 = %b ck2 = %b rst = %b in1 = %d in2 = %d out = %d ",
$time, ck1, ck2, rst, in1, in2, out);

endmodule

The result of simulation is:

# time = 0 ck1 = 0 ck2 = 0 rst = 1 in1 = 8 in2 = 0 out = x
# time = 3 ck1 = 1 ck2 = 0 rst = 1 in1 = 8 in2 = 0 out = 255
# time = 4 ck1 = 0 ck2 = 0 rst = 1 in1 = 8 in2 = 0 out = 255
# time = 5 ck1 = 0 ck2 = 1 rst = 1 in1 = 8 in2 = 0 out = 255
# time = 6 ck1 = 0 ck2 = 0 rst = 1 in1 = 8 in2 = 0 out = 255
# time = 7 ck1 = 1 ck2 = 0 rst = 1 in1 = 8 in2 = 0 out = 255
# time = 8 ck1 = 0 ck2 = 0 rst = 0 in1 = 8 in2 = 0 out = 0
# time = 9 ck1 = 0 ck2 = 1 rst = 0 in1 = 8 in2 = 0 out = 0
# time = 10 ck1 = 0 ck2 = 0 rst = 0 in1 = 8 in2 = 0 out = 0
# time = 11 ck1 = 1 ck2 = 0 rst = 0 in1 = 8 in2 = 0 out = 0
# time = 12 ck1 = 0 ck2 = 0 rst = 0 in1 = 2 in2 = 0 out = 0
# time = 13 ck1 = 0 ck2 = 1 rst = 0 in1 = 2 in2 = 0 out = 0
# time = 14 ck1 = 0 ck2 = 0 rst = 0 in1 = 2 in2 = 0 out = 0
# time = 15 ck1 = 1 ck2 = 0 rst = 0 in1 = 2 in2 = 0 out = 0
# time = 16 ck1 = 0 ck2 = 0 rst = 0 in1 = 4 in2 = 0 out = 0
# time = 17 ck1 = 0 ck2 = 1 rst = 0 in1 = 4 in2 = 0 out = 0
# time = 18 ck1 = 0 ck2 = 0 rst = 0 in1 = 4 in2 = 0 out = 0
# time = 19 ck1 = 1 ck2 = 0 rst = 0 in1 = 4 in2 = 0 out = 0
# time = 20 ck1 = 0 ck2 = 0 rst = 0 in1 = 2 in2 = 0 out = 0
# time = 21 ck1 = 0 ck2 = 1 rst = 0 in1 = 2 in2 = 0 out = 0
# time = 22 ck1 = 0 ck2 = 0 rst = 0 in1 = 2 in2 = 0 out = 0
# time = 23 ck1 = 1 ck2 = 0 rst = 0 in1 = 2 in2 = 0 out = 0
# time = 24 ck1 = 0 ck2 = 0 rst = 0 in1 = 1 in2 = 0 out = 0
# time = 25 ck1 = 0 ck2 = 1 rst = 0 in1 = 1 in2 = 0 out = 0
# time = 26 ck1 = 0 ck2 = 0 rst = 0 in1 = 1 in2 = 0 out = 0
# time = 27 ck1 = 1 ck2 = 0 rst = 0 in1 = 1 in2 = 0 out = 0
# time = 28 ck1 = 0 ck2 = 0 rst = 0 in1 = 0 in2 = 255 in2 = 255 out = 0
# time = 29 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out = 0
# time = 30 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 0
# time = 31 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 1
# time = 32 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 1
# time = 33 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out = 1
# time = 34 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 1
# time = 35 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 1
# time = 36 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 2
# time = 37 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out = 2
# time = 38 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 2
# time = 39 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out = 2
# time = 40 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 2
# time = 41 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out = 2
# time = 42 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 2
# time = 43 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out = 2
# time = 44 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 4
# time = 45 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out = 4
```
11.5 Interconnection issues

Simple circuits scale up easy generating big interconnection problems.

11.5.1 Local vs. global connections

The origin of the memory wall is in the inability to avoid global connection on memory arrays, while in the logic areas the local connections are easiest to impose.

Memory wall

11.5.2 Localizing with cellular automata

11.5.3 Many clock domains & asynchronous connections

The clock signal uses a lot of energy and area and slows down the design when the area of the circuit became too big.

A fully synchronous design generate also power distribution issues, which come with all the associated problems.

11.6 Neural networks

Artificial neural network (NN) is a technical construct inspired from the biological neural networks. NN are composed of interconnected artificial neurons. An artificial neuron is a programmed or circuit construct that mimic the property of a biological neuron. A multi-layer NN is used as a connectionist computational model. The introductory text [Zurada ‘95] is used for a short presentation of the concept of NN.

11.6.1 The neuron

The artificial neuron (see Figure 11.8) receives the inputs $x_1, \ldots, x_n$ (corresponding to $n$ dendrites) and process them to produce an output $o$ (synapse). The sums of each node are weighted, using the weight vector $w_1, \ldots, w_n$ and the sum, $net$, is passed through a non-linear function, $f(net)$, called activation function or transfer function. The transfer functions usually have a sigmoid shape (see Figure 11.9) or step functions.

Formally, the transfer function of a neuron:

$$o = f(\sum_{i=1}^{n} w_i x_i) = f(net)$$
Figure 11.8: The general form of a neuron. a. The circuit structure of a $n$-input neuron. b. The logic symbol.

where $f$, the typical activation function, is:

$$f(y) = \frac{2}{1 + \exp(-\lambda y)} - 1$$

The parameter $\lambda$ determines the steepness of the continuous function $f$. For big value of $\lambda$ the function $f$ becomes:

$$f(y) = \text{sgn}(y)$$

The neuron works as a combinational circuit performing the scalar product of the input vector $x = [x_1, x_2, \ldots, x_n]$ with the weight vector $w = [w_1, w_2, \ldots, w_n]$ followed by the application of the activation function. The activation function $f$ is simply implemented using as a look-up table using a Read-Only Memory.

### 11.6.2 The feedforward neural network

A feedforward NN is a collection of $m$ $n$-input neurons (see Figure 11.10). Each neuron receives the same input vector $x = [x_1, x_2, \ldots, x_m]$ and is characterized by its own weight vector $w_i = [w_1, w_2, \ldots, w_m]$. The entire NN provides the output vector $o = [o_1, o_2, \ldots, o_m]^T$. The activation function is the same for each neuron.
Each NN is characterized by the weight matrix

$$W = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{pmatrix}$$

having for each output a line, while for each input it has a column. The transition function of the NN is:

$$o(t) = G[wx(t)]$$

where:

$$\begin{pmatrix} f(\cdot) & 0 & \cdots & 0 \\ 0 & f(\cdot) & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & f(\cdot) \end{pmatrix}$$

The feedforward NN is of “instantaneous” type, i.e., it behaves as a combinational circuit which provides the result in the same “cycle”. The propagation time associated do not involve storage elements.

**Example 11.6** The shaded area in Figure 11.11 must be recognized by a two-layer feedforward NN. Four conditions must be met to define the surface:

- $x_1 - 1 > 0 \Rightarrow \text{sgn}(x_1 - 1) = 1$
- $x_1 - 2 < 0 \Rightarrow \text{sgn}(-x_1 + 2) = 1$
- $x_2 > 0 \Rightarrow \text{sgn}(x_2) = 1$
- $x_2 - 3 < 0 \Rightarrow \text{sgn}(-x_2 + 3)$

For each condition a neuron from the first layer is used. The second layer determines whether all the conditions tested by the first layer are fulfilled.

The first layer is characterized the weight matrix

$$W_{43} = \begin{pmatrix} 1 & 0 & 1 \\ -1 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & -1 & -3 \end{pmatrix}$$

The weight vector for the second layer is

$$W = [1 \hspace{1em} 1 \hspace{1em} 1 \hspace{1em} 3.5]$$

On both layers the activation function is sgn.
11.6. NEURAL NETWORKS

11.6.3 The feedback neural network

The feedback NN is a sequential system. It provides the output with a delay of a number of clock cycles after the initialization with the input vector \( x \). The structure of a feedback NN is presented in Figure 11.12. The multiplexer \( \text{mux} \) is used to initialize the loop closed through \( \text{register} \). If init = 1 the vector \( x \) is applied to \( NN_{m,n}(f) \) one clock cycle, then init is switched to 0 and the loop is closed.

In the circuit approach of this concept, after the initialization cycle the output of the network is applied to the input through the feedback register. The transition function is:

\[
o(t + T_{\text{clock}}) = \Gamma[Wo(t)]
\]

where \( T_{\text{clock}} \) (the clock period) is the delay on the loop. After \( k \) clock cycles the state of the network is described by:

\[
o(t + k \times T_{\text{clock}}) = \Gamma[WT] \cdots \Gamma[Wo(t)] \cdots
\]

A feedback NN can be considered as an initial automaton with few final states mapping disjoint subsets of inputs.

**Example 11.7** Let be a feedback NN with 4 4-input neurons with one-bit inputs and outputs. The activation function is sgn. The feedback NN can be initialized with any 4-bit binary configuration from

---

**Figure 11.10:** The single-layer feedforward neural network. **a.** The organization of a feedforward NN having \( m \) \( n \)-input neurons. **b.** The logic symbol.
Figure 11.11: A two-layer feedforward NN. 

- **a.** The structure.
- **b.** The two-dimension space mapping.

\[ x = \begin{bmatrix} -1 & -1 & -1 & -1 \end{bmatrix} \]

\( \rightarrow \)

\[ x = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \]

and the system has two final states:

- \( o_{t4} = \begin{bmatrix} 1 & 1 & 1 & -1 \end{bmatrix} \)
- \( o_{t1} = \begin{bmatrix} -1 & -1 & -1 & 1 \end{bmatrix} \)

reached in a number of clock cycles after the initialization.

The resulting discrete-time recurrent network has the following weight matrix:

\[
W_{44} = \begin{bmatrix}
0 & 1 & 1 & -1 \\
1 & 0 & 1 & -1 \\
1 & 1 & 0 & -1 \\
-1 & -1 & -1 & 0
\end{bmatrix}
\]

The resulting structure of the NN is represented in Figure 11.13, where the weight matrix is applied on the four 4-bit inputs destined for the weight vectors.

The sequence of transitions are computed using the form:

\[
o(t + 1) = \begin{bmatrix} \text{sgn}(net_1(t)) & \text{sgn}(net_2(t)) & \text{sgn}(net_3(t)) & \text{sgn}(net_4(t)) \end{bmatrix}
\]
11.6. NEURAL NETWORKS

Figure 11.12: The single-layer feedback neural network.

Figure 11.13: The feedback NN with two final states.

Some sequences end in $o_{14} = [1 \ 1 \ 1 \ -1]$, while others in $o_1 = [-1 \ -1 \ -1 \ 1]$.

11.6.4 The learning process

The learning process is used to determine the actual form of the matrix $W$. The learning process is an iterative one. In each iteration, for each neuron the weight vector $w$ is adjusted with $\Delta w$, which is proportional with the input vector $x$ and the learning signal $r$. The general form of the learning signal is:

$$r = r(w, x, d)$$

where $d$ is the desired response (the teacher’s signal). Thus, in each step the weight vector is adjusted as follows:

$$w(t + 1) = w(t) + c \times r(w(t), x(t), d(t)) \times x(t)$$

where $c$ is the learning constant. The learning process starts from an initial form of the weight vector (established randomly or by a simple “hand calculation”) and uses as a set of training input vectors.

There are two types of learning:
unsupervised learning:

\[ r = r(w, x) \]

the desired behavior is not known; the network will adapt its response by “discovering” the appropriate values for the weight vectors by self-organization.

supervised learning:

\[ r = r(w, x, d) \]

the desired behavior, \( d \), is known and can be compared with the actual behavior of the neuron in order to find how to adjust the weight vector.

In the following both types will be exemplified using the Hebbian rule and the perceptron rule.

**Unsupervised learning: Hebbian rule**

The learning signal is the output of the neuron. In each step the vector \( w \) will be adjusted (see Figure 11.14) as follows:

\[ w(t+1) = w(t) + c \times f(w(t), x(t)) \times x(t) \]

The learning process starts with small random values for \( w \).

![Figure 11.14: The Hebbian learning rule](image)

**Example 11.8** Let be a four-input neuron with the activation function \( \text{sgn} \). The initial weight vector is:

\[ w(t_0) = [1 -1 0.5] \]

The training inputs are:

\[ x_1 = [1 -2 1.5 0], \]
\[ x_2 = [1 -0.5 -2 -1.5], \]
\[ x_3 = [0 1 -1 1.5] \]

Applying by turn the three training input vectors for \( c = 1 \) we obtain:

\[ w(t_0 + 1) = w(t_0) + \text{sgn}(\text{net}) \times x_1 = w(t_0) + \text{sgn}(3) \times x_1 = w(t_0) + x_1 = [2 -3 1.5 0.5] \]
\[ w(t_0 + 2) = w(t_0 + 1) + \text{sgn}(\text{net}) \times x_2 = w(t_0 + 1) + \text{sgn}(-0.25) \times x_2 = w(t_0 + 1) - x_2 = [1 -2.5 3.5 2] \]
\[ w(t_0 + 3) = w(t_0 + 2) + \text{sgn}(\text{net}) \times x_3 = w(t_0 + 2) + \text{sgn}(-3) \times x_3 = w(t_0 + 2) - x_3 = [1 -3.5 4.5 0.5] \]
Supervised learning: perceptron rule

The perceptron rule performs a supervised learning. The learning is guided by the difference between the desired output and the actual output. Thus, the learning signal for each neuron is:

\[ r = d - o \]

In each step the weight vector is updated (see Figure 11.15) according to the relation:

\[ w(t+1) = w(t) + c \times (d(t) - f(w(t), x(t))) \times x(t) \]

The initial value for \( w \) does not matter.

![Neuron diagram](image)

**Figure 11.15: The perceptron learning rule**

**Example 11.9** Let be a four-input neuron with the activation function \( \text{sgn} \). The initial weight vector is:

\[ w(t_0) = [1 \ 1 \ 0 \ 0.5] \]

The training inputs are:

- \( x_1 = [1 \ 2 \ 0 \ -1] \),
- \( x_2 = [0 \ 1.5 \ -0.5 \ -1] \),
- \( x_3 = [-1 \ 1 \ 0.5 \ -1] \)

and the desired output for the three input vectors are: \( d_1 = -1, d_2 = -1, d_3 = 1 \). The learning constant is \( c = 0.1 \).

Applying by turn the three training input vectors for \( c = 1 \) we obtain:

**step 1** : because \( (d - \text{sgn} \text{(net})) \neq 0 \)

\[ w(t_0 + 1) = w(t_0) + 0.1 \times (-1 + \text{sgn} \text{(net)}) \times x_1 = w(t_0) + 0.1 \times (-1 - 1) \times x_1 = [0.8 \ -0.6 \ 0.7] \]

**step 2** : because \( (d - \text{sgn} \text{(net)}) \neq 0 \) no correction is needed in this step

\[ w(t_0 + 2) = w(t_0 + 1) \]

**step 3** : because \( (d - \text{sgn} \text{(net)}) = 2 \)

\[ w(t_0 + 3) = w(t_0 + 2) + 0.1 \times 2 \times x_3 = [0.6 \ -0.4 \ 0.1 \ 0.5] \]
11.6.5 Neural processing

NN are currently used to model complex relationships between inputs and outputs or to find patterns in streams of data. Although NN has the full power of a Universal Turing Machine (some people claim that the use of irrational values for weights results in a machine with "super-Turing" power), the real application of this paradigm are limited only to few functions involving specific complex memory functions (please do not use this paradigm to implement a text editor). They are grouped in the following categories:

- auto-association: the input (even a degraded input pattern) is associated to the closest stored pattern
- hetero-association: the association is made between pairs of patterns; distorted input patterns are accepted
- classification: divides the input patterns into a number of classes; each class is indicated by a number (can be understood as a special case of hetero-association which returns a number)
- recognition: is a sort of classification with input patterns which do not exactly correspond to any of the patterns in the set
- generalization: is a sort of interpolation of new data applied to the input.

What is specific for this computational paradigm is that its “program” – the set of weight matrices generated in the learning process – do not provide explicit information about the functionality of the net. The content of the weight matrices can not be read and understood as we read and understand the program performed by a conventional processor built by a register file, an ALU, .... The representation of an actual function of a NN defies any pattern based interpretation. Maybe this is the price we must pay for the complexity of the functions performed by NN.

11.7 Problems

Problem 11.1 Design a stack with the first two recordings accessible.

Problem 11.2 Design a stack with the following features in reorganizing the first recordings.

Problem 11.3 Design a stack with controlled deep access.

Problem 11.4 Design an expandable stack.

Problem 11.5 The global loop on a linear cellular automata providing a pseudo-noise generator.

Problem 11.6

Problem 11.7

Problem 11.8
In the previous chapter we ended to discuss about closing local loops in digital systems introducing the $n$-th order systems.

In this chapter a new kind of loop will be introduced: the global loop. This loop has the following characteristics:

- it is closed over an $n$ order system
- usually carries only control information about the state of the entire system
- the feedback introduced classifies each subsystems in the context of the entire system

In the next chapter the second part of this book starts, with a discussion about recursive functions and their circuit implementations.

- The computation model of partial functions is presented.
- To each level (basic functions, composition, primitive recursiveness and minimalization) an digital order is associated.
- Conclusion: computation means at least 3-loop circuits.
A super-system is characterized by global loops closed over an $n$-order digital system. A global loop receives on its inputs information distributed over an array of digital modules and sends back in each digital module information related to the whole content of its inputs.

In the first section an introductory examples are provided closing global loops over one-dimension or two-dimension cellular automata. The second section introduces ConnexArray™, a cellular engine used as a general purpose parallel computing engine. It is controlled by one global loop closed over a linear cells of execution elements. The third section closes the second global loop over the same linear array. The fourth section provides a high level description of the system described in the previous two sections.

12.1 Global loops in cellular automata

A first attempt to close a loop over a simple cellular automaton is presented in [Ștefan ’98a]. The effect of a global loop on the behavior of a cellular automaton is presented in [Malita ’13]. In [Gheolbanoiu ’14] the attempt from [Ștefan ’98a] is finalized as an actual circuit.

How new features or an increased autonomy can be added in a $n$-OS? Closing global loops which consider the global state of the system. Let us take, as a $n$-OS, the simple case of a linear cellular system. Three kinds of global loops closed in a linear cellular system are represented in Figure 12.1:

- **a.** Scan global loop: receives a data vector from the cells and sends back a vector computed according to the global state of the array (Figure 12.1a); for example, a scan function which computes the sum prefixes

![Figure 12.1](image)

**Figure 12.1:** Super-loops over $n$-OS. **a.** Scan global loop. **b.** Reduction global loop. **c.** Controlled global loop.
12.2. THE GENERIC CONNEXARRAY™: THE FIRST GLOBAL LOOP

- **reduction global loop**: receives a data vector from the cells and sends back to each cell a value which corresponds to the global state of the system (Figure 12.1b); for example, the sum of the vector’s components or the maximal value

- **controlled global loop**: receives a vector from cells and sends back to each cell a command computed according to the global value provided by the reduction network (Figure 12.1c); for example, CONTROLLER could be a processing element which decides, according to its program and the current output of the reduction network, the command issued to the cellular system in the next cycle.

Each cell of n-OS could be a circuit, starting from a simple 2-state automaton [Ştefan ’98a] until an execution or a processing element. What is the degree of generality of these super-loop circuits? It is very important to evaluate the possibility to use them efficiently as parallel computational engines. But, what means a “parallel computational engine”?

12.2 The Generic ConnexArray™: the first global loop

In 1936 Stephen Kleene defined [Kleene ’36] the concept of **partial recursive function** as the general framework for computing any function of form:

\[ f : \{0, 1\}^n \rightarrow \{0, 1\} \]

In [Ştefan ’14] is proved that from the three basic rules proposed by Kleene only the first, the composition rule, is independent. Therefore, computation could be defined as repeated application of the composition having the form:

\[ f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_p(x_1, \ldots, x_n)) \]

In Figure 12.2a the two-level circuit version of the composition rule is represented. Each function \( h_i \) is computed by a module on the first level, while the function \( g \) reduces the resulting vector to a scalar. In Figure 12.2b, the limit case for \( p = 1 \) is represented. The repeated application of a composition requests the additional structures represented in Figure 12.2c. In [Ştefan ’14] the transition from Figure 12.2a and Figure 12.2b to Figure 12.2c is described.

The two-direction connections between the cells provide the \( p = n \) levels of loops which gives the order \( n \) to the system, while the global loop is closed through the CONTROLLER.

The simplest version of the engine is behaviorally described in the next subsection as the generic ConnexArray™ system.

12.2.1 The Generic ConnexArray™

The cell used in the generic \( n \)-order array with the first global loop contains a data memory for the local data and a simple accumulator based engine.

The cell’s structure is presented in Figure 12.3a, while the controller’s structure is presented in Figure 12.3b. The behavioral description uses the following memory resources:

```plaintext
// vectorial resources
reg [x-1:0] ixVect[0:(1<<x)-1] ; // index vector
reg boolVect[0:(1<<x)-1] ; // Boolean vector
reg [n-1:0] accVect[0:(1<<x)-1] ; // accumulator vector
reg crVect[0:(1<<x)-1] ; // carry vector
reg [v-1:0] addrVect[0:(1<<v)-1] ; // address vector
reg [n-1:0] vectMem[0:(1<<x)-1][0:(1<<v)-1] ; // vector memory
// control resources
reg [p-1:0] pc ; // program counter
reg [63:0] ir ; // instruction register
reg [63:0] progMem[0:(1<<p)-1] ; // program memory
```
Figure 12.2: From the mathematical model to the abstract machine model. a. The structure associated to the composition rule in Kleene’s model. b. The limit case for $p = 1$. It provides the pipelined connection which can be generalized for serially connected $p$ cells. c. The abstract machine model for parallel computing. The MAP section consists of the parallel connected cells, the REDUCE section stands for the function $g$, while the serial connections between cells in MAP section provides the serial pipelined connection for the limit case of $p = 1$.

```vhdl
// scalar resources
reg [31:10] acc ; // scalar accumulator
reg cr ; // scalar carry
reg [s-1:0] addr ; // scalar address
reg [n-1:0] mem[0:(1<<s)-1] ; // scalar memory

which are dimensioned as follows:

parameter n = 32 , // word size
x = 4 , // index size
v = 6 , // vector memory address size
s = 8 , // scalar memory address size
p = 5 // program memory address size

The system above defined consists of:

- **vectorial resources** describes the resources distributed in array (for the contribution of each cell see Figure 12.4)
  - **ixVect**: index vector used to associate an index, from 0 to $2^x - 1$, to each cell
  - **boolVect**: Boolean vector used to enable the execution in $i$-th cell; if $\text{boolVect}[i]$ is 1, then the cell is active, else the instruction received in the current cycle is ignored (substituted with nop)
  - **accVect**: is the scalar vector containing the accumulator registers of the cells; the execution unit is accumulator based, thus $\text{accVect}[i]$ is the accumulator of the cell $i$
  - **crVect**: is the Boolean vector containing the carry registers of each cell; $\text{crVect}[i]$ stores the carry bit generated in cell $i$ by the last arithmetic operation
  - **vectMem**: is the vector memory distributed along the cells; each cell, $\text{cell}_i$, stores in its local memory the $i$-th components of all the $2^v$ vectors
  - **addrVect**: used to specify a locally computed address in each cell
```
12.2. THE GENERIC CONNXARRAY™: THE FIRST GLOBAL LOOP

Figure 12.3: The components of the abstract machine model. a. The cell’s structure. b. The controller’s structure.

**control resources** describes the resources involved in the sequential control

- \( pc \) : \( p \)-bit program counter
- \( ir \) : 64-bit instruction register
- \( progMem \) : the program memory organized in \( 2^p \) 64-bit words

**scalar resources** describes the resources involved in the scalar computation

- \( acc \) : controller’s accumulator
- \( cr \) : the carry flip-flop
- \( addr \) : the address register used to compute the address for controller’s data memory
- \( mem \) : controller’s data memory

The input received by each cell (see Figure 12.3a and Figure 12.4) is:

\[
in = \{\text{instruction}[31:0], \text{data}[n-1:0], \text{address}[v-1:0]\}
\]

while the output is:

\[
out = \{\text{boolVect}[i], (\text{boolVect}[i] ? \text{accVect}[i][n-1:0] : 0)\}
\]

From the program memory, in each cycle is read a pair of instructions: one (\( progMem[nextPc][31:0] \)) for the use of controller and another (\( progMem[nextPc][63:32] \)) to be executed in each active cell (where \( \text{boolVect}[i] = 1 \)). The structure of the two instructions is:

assign contrOpCode = ir[31:27]; // operation code for controller
assign contrOperand = ir[26:24]; // right selection operand for controller
assign contrScalar = ir[23:0]; // immediate value for controller
assign arrayOpCode = ir[63:59]; // operation code for the array of cells
assign arrayOperand = ir[58:56]; // right selection operand for array
assign arrayScalar = ir[55:32]; // immediate value for array
Figure 12.4: The resources used in the behavioral description. a. Cell’s internal state support. b. Controller’s internal state support.

The Instruction Set Architecture

The arithmetic-logic operations performed in each cell and in the controller are very similar. Only the sequential control in controller and the spatial control in the array of cells differentiate the instruction set architecture (ISA) which describes the controller and the cells. In the list of instruction which follows the instruction specific for controller are labeled with #, while the instruction specific for the array’s cells are labeled with $. The instructions are specified by two fields: one for operation (opCode) and one for the right operand (operand). The left operand is always the accumulator.

The operand code meaning is:

parameter // selects the right operand or destination
val = 3’b000, // immediate value: {24{scalar[7]}}, scalar} (operand only)
mab = 3’b001, // absolute: mem[scalar[s/v-1:0]]
mri = 3’b010, // relative: mem[addr + scalar[s/v-1:0]]
mri = 3’b011, // relative and increment: mem[addr + scalar[s-1:0]]; // addr <= mem[addr + scalar[s-1:0]]
ads = 3’b100, // address register for data memory (destination only)
cop = 3’b101, //$ co-operand: acc (only for array; operand only)
idx = 3’b111, //$ index (only for array; operand only)
rad = 3’b110, /// reduction add (only in scalar section; operand only)
rmx = 3’b111, /// reduction max (only in scalar section; operand only)
rfl = 3’b111; /// reduction flag (only in scalar section; operand only)

The conditions tested by the instructions where and wheren are:

parameter
zero = 3’b000, // accVect[i] == 0
carry = 3’b001, // crVect[i] == 1
first = 3’b010, // firstVect[i] == 1
next = 3’b011; // nextVect[i] == 1
The opCode code meaning is:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>00000</td>
<td>acc &lt;= acc + op;</td>
</tr>
<tr>
<td>addc</td>
<td>00001</td>
<td>acc &lt;= acc + op + cr;</td>
</tr>
<tr>
<td>sub</td>
<td>00010</td>
<td>acc &lt;= acc - op;</td>
</tr>
<tr>
<td>rsub</td>
<td>00011</td>
<td>operand - acc;</td>
</tr>
<tr>
<td>subc</td>
<td>00100</td>
<td>acc &lt;= acc - op - cr;</td>
</tr>
<tr>
<td>rsubc</td>
<td>00101</td>
<td>op - acc - cr;</td>
</tr>
<tr>
<td>mult</td>
<td>00110</td>
<td>acc &lt;= acc * op;</td>
</tr>
<tr>
<td>load</td>
<td>00111</td>
<td>acc &lt;= op;</td>
</tr>
<tr>
<td>store</td>
<td>01000</td>
<td>operand &lt;= acc;</td>
</tr>
<tr>
<td>bband</td>
<td>01001</td>
<td>acc &lt;= acc &amp; op;</td>
</tr>
<tr>
<td>bwor</td>
<td>01010</td>
<td>acc &lt;= acc</td>
</tr>
<tr>
<td>bwxor</td>
<td>01011</td>
<td>acc &lt;= acc ^ op;</td>
</tr>
<tr>
<td>gshift</td>
<td>01100</td>
<td>acc[i] &lt;= acc[i+scalar];</td>
</tr>
<tr>
<td>insval</td>
<td>01101</td>
<td>acc &lt;= {acc[23:0], scalar}</td>
</tr>
<tr>
<td>shrightc</td>
<td>01110</td>
<td>(cr, acc) &lt;= {acc[0], cr, acc[n-1:1]}</td>
</tr>
<tr>
<td>shright</td>
<td>01111</td>
<td>(cr, acc) &lt;= {acc[0], 1'b0, acc[n-1:1]}</td>
</tr>
<tr>
<td>sharight</td>
<td>01111</td>
<td>acc &lt;= {acc[n-1], acc[n-1:1]}</td>
</tr>
<tr>
<td>jmp</td>
<td>10000</td>
<td>pc &lt;= pc + scalar;</td>
</tr>
<tr>
<td>brz</td>
<td>10001</td>
<td>pc &lt;= acc==0 ? pc + scalar : pc + 1;</td>
</tr>
<tr>
<td>brnz</td>
<td>10010</td>
<td>pc &lt;= acc==0 ? pc + 1 : pc + scalar;</td>
</tr>
<tr>
<td>brzdec</td>
<td>10011</td>
<td>pc &lt;= acc==0 ? pc + scalar : pc + 1; acc &lt;= acc - 1</td>
</tr>
<tr>
<td>brnzdec</td>
<td>10100</td>
<td>pc &lt;= acc==0 ? pc + 1 : pc + scalar; acc &lt;= acc - 1</td>
</tr>
<tr>
<td>where</td>
<td>10101</td>
<td>boolVect &lt;= cond[operand] ? 1 : 0;</td>
</tr>
<tr>
<td>wheren</td>
<td>10110</td>
<td>boolVect &lt;= cond[operand] ? 0 : 1;</td>
</tr>
<tr>
<td>elsew</td>
<td>10111</td>
<td>boolVect &lt;= ~boolVect;</td>
</tr>
<tr>
<td>endwhere</td>
<td>11000</td>
<td>boolVect &lt;= 1;</td>
</tr>
<tr>
<td>search</td>
<td>11001</td>
<td>boolVect &lt;= (acc == op) ? 1 : 0;</td>
</tr>
<tr>
<td>csearch</td>
<td>11010</td>
<td>boolVect &lt;= (acc == op) &amp; (boolVect &gt;&gt; 1) ? 1 : 0;</td>
</tr>
<tr>
<td>read</td>
<td>11011</td>
<td>boolVect &lt;= boolVect &gt;&gt; 1;</td>
</tr>
<tr>
<td>insert</td>
<td>11100</td>
<td>insert at first</td>
</tr>
<tr>
<td>delete</td>
<td>11101</td>
<td>delete first</td>
</tr>
<tr>
<td>vload</td>
<td>11110</td>
<td>vectMem[i][accVect[i]] &lt;= mem[acc + i*contrScalar]</td>
</tr>
<tr>
<td>vstore</td>
<td>11111</td>
<td>mem[acc + i*contrScalar] &lt;= vectMem[i][accVect[i]]</td>
</tr>
</tbody>
</table>

Program Control Section

The program control section of the controller works as it is described in the following:

```
always @(posedge cycle) if (reset) begin pc <= {p{1'b1}} ;
    ir <= 0 ;
end
else begin pc <= nextPc ;
    ir <= progMem[nextPc] ;
end
```

always @(*)
case(contrOpCode)
    jmp : nextPc = pc + contrScalar[p-1:0] ;
    brz : nextPc = (acc == 0) ? (pc + contrScalar[p-1:0]) : (pc + 1'b1) ;
brnz : nextPc = (acc == 0) ? (pc + 1'b1) : (pc + contrScalar[p-1:0]) ;
brzdec : nextPc = (acc == 0) ? (pc + contrScalar[p-1:0]) : (pc + 1'b1) ;
brnzdec : nextPc = (acc == 0) ? (pc + 1'b1) : (pc + contrScalar[p-1:0]) ;
default nextPc = pc + 1'b1 ;
endcase

The Execution in Controller and in Cells

Operand selection in controller is described by the following code:

```verilog
always @(*)
case(contrOperand) // selects the right operand for controller
  mrl: op = mem[addr + contrScalar[s-1:0]] ;
  mri: op = mem[addr + contrScalar[s-1:0]] ;
  val: op = {{{n-8}{contrScalar[7]}}, contrScalar} ;
  rad: begin op = accVect[0] ;
       for (j=1; j<(1<<x); j=j+1)
       op = op + accVect[j] ;
   end
  rmx: begin op = accVect[0] ;
       for (j=1; j<(1<<x); j=j+1)
       op = (op < accVect[j]) ? accVect[j] : op ;
   end
  rfl: begin op = {{{n-1}{1'b0}}, boolVect[0]} ;
       for (j=1; j<(1<<x); j=j+1)
       op = {{(n-1){1'b0}}, op[0] | boolVect[j]} ;
   end
  default op = mem[contrScalar[s-1:0]] ;
endcase
```

Operand selection in the array’s cells is described by the following code:

```verilog
always @(*) for (k=0; k<(1<<x); k=k+1)
begin
case(arrayOperand) // selects the right operand in each cell
  mrl: opVect[k] = vectMem[k][addrVect[k] + arrayScalar[v-1:0]] ;
  mri:opVect[k] = vectMem[k][addrVect[k] + arrayScalar[v-1:0]] ;
  val: opVect[k] = {{{n-c}{arrayScalar[7]}}, arrayScalar} ;
  cop: opVect[k] = acc ;
  idx: opVect[k] = k ;
  default opVect[k] = vectMem[k][arrayScalar[v-1:0]] ;
endcase
pxVect[k] = (k == 0) ? boolVect[0] : (boolVect[k] | pxVect[k-1]) ;
firstVect[k] = (k == 0) ? pxVect[0] : (pxVect[k] & ~pxVect[k-1]) ;
nextVect[k] = (k == 0) ? 1'b0 : pxVect[k-1] ;
condVect[k] = {nextVect[k], firstVect[k], crVect[k], (accVect[k] == 0)} ;
end
```

Data operations in controller is performed using as operands the accumulator, acc, and data selected by contrOperand, as follows:

```verilog
always @(posedge cycle)
case(contrOpCode)
```
add : \{cr, acc\} <= acc + op ;
addc : \{cr, acc\} <= acc + op + cr ;
sub : \{cr, acc\} <= acc - op ;
rsub : \{cr, acc\} <= op - acc ;
subc : \{cr, acc\} <= acc - op - cr ;
rsubc : \{cr, acc\} <= op - acc - cr ;
mult : \{cr, acc\} <= \{cr, acc \ast op\} ;
load : \{cr, acc\} <= \{cr, op\} ;
store : case(contrOperand)
    mab : mem[contrScalar[s-1:0]] <= acc ;
mrl : mem[contrScalar[s-1:0] + addr] <= acc ;
mri : begin
    addr <= contrScalar[s-1:0] + addr ;
    addr <= contrScalar[s-1:0] + addr ;
end
    ads : addr <= acc[s-1:0] ;
    default addr <= acc[s-1:0] ;
endcase
bwand : \{cr, acc\} <= \{cr, acc \& op\} ;
bwor : \{cr, acc\} <= \{cr, acc | op\} ;
bxor : \{cr, acc\} <= \{cr, acc ^ op\} ;
insval : \{cr, acc\} <= \{cr, acc[23:0], contrScalar\} ;
shrightc: \{cr, acc\} <= \{acc[0], cr, acc[n-1:1]\} ;
shright : \{cr, acc\} <= \{acc[0], 1'b0, acc[n-1:1]\} ;
sharight: \{cr, acc\} <= \{acc[0], acc[n-1], acc[n-1:1]\} ;
vstore : for (l=0; l<(1<<x); l=l+1)
    mem[acc + l*contrScalar] <= vectMem[l][accVect[l]] ;
brzdec : \{cr, acc\} <= acc - 1'b1 ;
brnzdec : \{cr, acc\} <= acc - 1'b1 ;
default \{cr, acc\} <= \{cr, acc\} ;
endcase

Data operations in the array’s cells is performed using as operands the accumulator, accVect[i], and data selected by arrayOperand, as follows:

always @\(\{\text{posedge cycle}\}\) for \(i=0; i<(1<<x); i=i+1\) begin
    if (arrayOperand == mri) addrVect[i] <= addrVect[i] + arrayScalar[v-1:0] ;
case(arrayOpCode)
    add : \{crVect[i], accVect[i]\} <= accVect[i] + opVect[i] ;
    addc : \{crVect[i], accVect[i]\} <= accVect[i] + opVect[i] + crVect[i] ;
    sub : \{crVect[i], accVect[i]\} <= accVect[i] - opVect[i] ;
    rsub : \{crVect[i], accVect[i]\} <= opVect[i] - accVect[i] ;
    subc : \{crVect[i], accVect[i]\} <= accVect[i] - opVect[i] - crVect[i] ;
    rsubc : \{crVect[i], accVect[i]\} <= opVect[i] - accVect[i] - crVect[i] ;
    mult : \{crVect[i], accVect[i]\} <= \{crVect[i], accVect[i] \ast opVect[i]\} ;
    load : \{crVect[i], accVect[i]\} <= \{crVect[i], op Vect[i]\} ;
    store : case(arrayOperand)
        mab : vectMem[i][arrayScalar[v-1:0]] <= accVect[i] ;
mrl : vectMem[i][arrayScalar[v-1:0] + addrVect[i]] <= accVect[i] ;
mri : vectMem[i][arrayScalar[v-1:0] + addrVect[i]] <= accVect[i] ;
endcase
end


Spatial selection allows to decide, according to the internal state of each cell, if the current instruction is executed or not in cell\[i\]. The spatial selection instructions act on the content of the Boolean vector, as follows:

\begin{verbatim}
always @(posedge cycle) for (i=0; i<(1<<x); i=i+1)
    case(arrayOpCode)
        where : boolVect\[i\] <= (condVect\[i\][arrayOperand[1:0]]) \? 1\?1'b0;
        wheren : boolVect\[i\] <= (condVect\[i\][arrayOperand[1:0]]) \? 1\?1'b1;
        elsew : boolVect\[i\] <= ~boolVect\[i\];
        endwhere : boolVect\[i\] <= 1\?1'b1;
        default boolVect\[i\] <= boolVect\[i\];
    endcase
endmodule
\end{verbatim}

Vector transfer instructions are used to exchange data between the vector memory distributed in the array of cell and the controller’s memory. The transfer is strided with stride given by \texttt{contrScalar}. The definitions are:

\begin{verbatim}
always @(posedge cycle) for (i=0; i<(1<<x); i=i+1)
    begin
        if (arrayOpCode == vload) vectMem\[i\][accVect\[i\]] <= mem[acc + i*contrScalar];
        if (contrOpCode == vstore) mem[acc + l*contrScalar] <= vectMem[l][accVect[l]];
    end
\end{verbatim}

12.2.2 Assembler Programming the Generic ConnexArray\textsuperscript{TM}

Each line of program must contain code for both instructions: the instruction issued for the array and the instruction performed by the controller. For conditioned or unconditioned relative jumps in program some lines are labeled;
LB(i) denote the label i. The use of the label is indicated by by the value used by control instructions (example: cJMP(2)).

**Example 12.1** The program which compute in the accumulator of the controller the inner product of the index vector ix with itself is:

```c
chNOP; ENDWHERE; // activate all cells
cchNOP; IXLOAD; // accVect[i] <= ixVect[i]
cchNOP; IXMUL; // accVect[i] <= accVect[i] * ixVect[i]
cRSLOAD; NOP; // load acc with the reduction sum
cHALT; NOP;
```

The content of program memory is:

```
progMem[0] = 11000000000000000000000000000000
progMem[1] = 00111111000000000000000000000000
progMem[2] = 00110111000000000000000000000000
progMem[3] = 00000000000000000111101000000000
progMem[4] = 00000000000000001000000000000000
```

The result of simulation:

```
time=0 rst=1 pc= x acc=x aVect[0]=x aVect[1]=x aVect[2]=x ... aVect[15]= x bVect=xxxx xxxx

time=1 rst=1 pc=31 acc=x aVect[0]=x aVect[1]=x aVect[2]=x ... aVect[15]= x bVect=xxxx xxxx

time=3 rst=0 pc= 0 acc=x aVect[0]=x aVect[1]=x aVect[2]=x ... aVect[15]= x bVect=xxxx xxxx

time=5 rst=0 pc= 1 acc=x aVect[0]=x aVect[1]=x aVect[2]=x ... aVect[15]= x bVect=xxxx xxxx

time=7 rst=0 pc= 2 acc=x aVect[0]=0 aVect[1]=1 aVect[2]=2 ... aVect[15]= 15 bVect=1111 1111


```

**Example 12.2** The program which load the accumulator the index in each cell, stores it incremented in 12 successive addresses starting from the address 2, than add in accumulator the stored values. The program is:

```c
cVLOAD(12); ENDWHERE; // acc <= 12; activate all cells
cchNOP; VLOAD(2); // accVect[i] <= 2
cchNOP; ADDRLD; // addrVect[i] <= accVect[i]
cchNOP; IXLOAD; // accVect[i] <= index
LB(1); cchNOP; RISTORE(1); // vectMem[i][addrVect + 1] <= accVect[i];
        // addrVect <= addrVect + 1
cCBRNZDEC(1); VADD(1); // if (acc !== 0) branch to LB(1); acc <= acc - 1;
        // accVect[i] <= accVect[i] + 1;
cVLOAD(13); VLOAD(2); // acc <= 13; accVect[i] <= 2
cchNOP; ADDRLD; // addrVect[i] <= accVect[i]
cchNOP; VLOAD(0); // accVect[i] <= 0
LB(2); cCBRNZDEC(2); RIADD(1); // if (acc !== 0) branch to LB(2); acc <= acc - 1;
        // accVect[i] <= accVect[i] + vectMem[i][addrVect + 1];
        // addrVect <= addrVect + 1
cHALT; NOP; // halt
```

⋄
12.3 Search Oriented ConnexArray<sup>TM</sup>: the Second Global Loop

The global loop closed in the previous section sends back to the array the same data for each cell. A new feature is added when another loop sends back to the array specific, differentiated information for each cell. Let us start with a very simple function associated to this second global loop: it takes the Boolean vector \( \text{boolVect}[0:(1<<x)-1] \) distributed along the array of cells and sends back two Boolean vectors:

- \( \text{firstVect}[0:(1<<x)-1] \): with 1 only on the position of the first occurrence of 1 in \( \text{boolVect} \); it is used to indicate the first active cell
- \( \text{nextVect}[0:(1<<x)-1] \): with 1 in all the positions next to the 1 in the \( \text{firstVect} \) Boolean vector; it is used to indicate all the cells next to the first active cell

There are 5 instructions supported by this second global loop:

```verilog
always @(posedge cycle) begin
    case(arrayOpCode)
        search : boolVect[i] <= (accVect[i] == opVect[i]) ? 1'b1 : 1'b0 ;
        csearch : boolVect[i] <= (i == 0) ? 1'b0 : (((accVect[i] == opVect[i]) & boolVect[i-1]) ? 1'b1 : 1'b0) ;
        read : boolVect[i] <= (i == 0) ? 1'b0 : boolVect[i-1] ;
        insert : begin
            if (firstVect[i])
                accVect[i] <= opVect[i] ;
            else if (nextVect[i])
                accVect[i] <= (i == 0) ? accVect[i] : accVect[i-1] ;
        end
        delete : if (firstVect[i] | nextVect[i])
            accVect[i] <= (i == ((1<<x)-1)) ? 0 : accVect[i+1] ;
        default boolVect[i] <= boolVect[i] ;
    endcase
end
```

The instruction \( \text{search} \) identifies all the positions in array where the accumulator has a certain value, while the \( \text{csearch} \) supports the search of a certain string of values.

**Example 12.3** The program which load the index in each \( \text{acc[i]} \), identifies the occurrence of the stream \( <1, 2, 3> \) in \( \text{accVect} \) and adds in \( \text{acc} \) the next four numbers:

```verilog
cNOP; // set active all cells
cVLOAD(1); IXLOAD; // acc = 1; load index in each cell
cVLOAD(2); SEARCH; // acc = 2; search 'acc' in each acc[i]
cVLOAD(3); CSEARCH; // acc = 3; search 'acc' after each active cell
chnop; // search 'acc' after each active cell
chnop; // boolVect >> 1
crsload; READ; // acc = reduceSum; boolVect >> 1
crsadd; READ; // acc = acc + reduceSum; boolVect >> 1
crsadd; READ; // acc = acc + reduceSum; boolVect >> 1
crsadd; NOP; // acc = acc + reduceSum
chalt; NOP; // halt
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>31</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
12.4. HIGH LEVEL ARCHITECTURE FOR CONNEXARRAY™

Example 12.4 The program which load the index in each acc[i], identifies the occurrence(s) of the stream <0 1> in accVect and insert in the vector accVect the sequence <13 15>.

```
chOP;                        ENDWHERE; // set active all cells
chOP;                        IXLOAD;    // load index in each cell
chOP;                        VSEARCH(1); // search '0' in each acc[i]
chOP;                        VCSEARCH(2); // search '1' after each active cell
chOP;                        READ;       // boolVect >> 1
chOP;                        INSERT(13); // insert '13' in the first active cell
chOP;                        INSERT(15); // insert '15' in the first active cell
chHALT;                      NOP;        // halt
```

The second global loop adds specific features which support search applications, sparse matrix/vector operations, ... .

12.3.1 List processing

12.3.2 Sparse matrix operations

12.4 High Level Architecture for ConnexArray™

Connex System is simulated in SCHEME1 providing an easy to use tool for architectural design and algorithm development. The Connex System (see Figure 12.5) is defined by the following set of global variables:

```
(define MEMORY ()); list of scalars
(define ARRAY ()); 2-level list of lists
(define M 0); memory size
(define P 0); vector size
(define N 0); number of vectors
(define INDEX ()); constant vector
(define ACTIVE ()); vector of up/down counters
(define (SYSTEM) (list ARRAY ACTIVE INDEX MEMORY))
```

ARRAY represents the internal distributed memory of N P-word vectors. MEMORY is the external M-word RAM of the system. The vector INDEX is a constant vector used to individualize the P cells of the hardware array. Each cell has its own local N-word RAM. The vector ACTIVE is used to select the active cell in each cycle of computation: if ai = 0 then the i-th cell is active. Every time a cell is omitted from selection the associated counter in ACTIVE is incremented. The selections are restored by an unsigned integer decrement of the vector ACTIVE.

The code of the functions used in simulation is posted at http://arh.pub.ro/gstefan/BCsystem/neoConnex.rkt.

---

1 The DrRacket implementation downloadable form http://racket-lang.org/download/.
12.4.1 System initialization

The following functions are used to initialize the whole system:

**ResetSystem** : is the function used to initialize the shape of a SYSTEM having \( n \) \( p \)-word vectors in ARRAY and an \( m \)-word MEMORY.

\[
\begin{align*}
\text{(define (ResetSystem } n \ p \ m) & ; \text{ is the function used to initialize the shape of a SYSTEM having } n \ p \text{-word vectors in ARRAY} \\
& \text{ and an } m \text{-word MEMORY.} \\
\text{(set! ARRAY (make-list } n \ ( \text{make-list } p \ ')))); \\
\text{(set! MEMORY (make-list } m \ ')))); \\
\text{(set! ACTIVE (make-list } p \ 0)))); \\
\text{(set! N } n) ; \text{number of vectors in ARRAY} \\
\text{(set! P } p) ; \text{length of a vector max } P=1024 \\
\text{(set! M } m) ; \text{number of scalars in MEMORY} \\
\text{(set! INDEX (Index } P))
\end{align*}
\]

The vector ACTIVE is a vector of counters used to keep track of the embedded selections. For \( \text{ACTIVE}[i] = 0 \) the cell \( i \) is active. The vector INDEX is used to index the cells of the system. \( \text{INDEX}[i] = i \) in the cell \( i \).

Two auxiliary functions are defined:

**pp** : is the pretty printing SYSTEM

**rs** : defined as:

\[
\text{(define (rs) (ResetSystem 4 8 32))}
\]
to be used to make easier the initialization.

**Example 12.5** A system initialization runs the function \texttt{rs}.

\begin{verbatim}
> (rs)
> (pp)
0:(' - - - - - - - )
1:(' - - - - - - - )
2:(' - - - - - - - )
3:(' - - - - - - - )
A:('0 0 0 0 0 0 0 0)
M:(^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^)
\end{verbatim}

provides a system with 4 8-word vectors in ARRAY, a 32-word memory and all cells selected. The initial content of ARRAY and MEMORY is filled up with dummy symbols.

\begin{verbatim}
\textbf{test} : is defined for initializing the content of the system:

\begin{verbatim}
(define (test)
  (ResetSystem 4 8 32)(newline)
  (SetVector 0 '(2 1 4 3 4 6 7 0))
  (SetVector 1 (makeAll 5))
  (SetVector 2 '(0 1 2 3 4 5 6 7))
  (SetVector 3 '(7 6 5 4 3 2 1 0))
  (SetActive '(1 0 0 0 0 0 0 1))
  (SetStream 0 (Index 32))
  (pp)
)
\end{verbatim}
\end{verbatim}

where:

\begin{verbatim}
\textbf{Index} : (Index n) returns the list of n successive numbers starting with 0.

> (Index 12)
(0 1 2 3 4 5 6 7 8 9 10 11)
\end{verbatim}

The function is used as (Index P) to initialize the constant vector INDEX used to name each cell in the array of P cells.

\begin{verbatim}
\textbf{MakeAll} : (MakeAll val) returns a P-length vector with val on each position. (TBD: change name in Fill)

> (MakeAll '13)
(13 13 13 13 13 13 13)
\end{verbatim}

**Example 12.6** Let's apply the function \texttt{test}.

\begin{verbatim}
> (test)
0:('2 1 4 3 4 6 7 0)
1:('5 5 5 5 5 5 5 5)
2:('0 1 2 3 4 5 6 7)
3:('7 6 5 4 3 2 1 0)
A:('1 0 0 0 0 0 0 1)
M:(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)
\end{verbatim}

Now, the content of the four vectors is initialized, the first and the last cell are inactivated and the memory is loaded with data.

\begin{verbatim}
12.4.2 Addressing functions

**Vec**: \((\text{Vec } a\text{Addr})\) address the vector \(a\text{Addr}\) in ARRAY

\(\langle\text{Vec 3}\rangle\)
\(\langle 7\ 6\ 5\ 4\ 3\ 2\ 1\ 0\rangle\)

**Mem**: \((\text{Mem } m\text{Addr})\) address the location \(m\text{Addr}\) in MEMORY

\(\langle\text{Mem 7}\rangle\)

7

12.4.3 Setting functions

**Copy**: \((\text{Copy } v\text{Source } v\text{Dest})\) returns vector \(v\text{Source}\) copied over the vector \(v\text{Dest}\) in all places where \(\text{ACTIVE} = 0\).

\(\langle\text{Copy } (\text{Vec 2}) (\text{Vec 1})\rangle\)
\(\langle 5\ 1\ 2\ 3\ 4\ 5\ 6\ 5\rangle\)

\(\langle pp\rangle\)

0: \(\langle 2\ 1\ 4\ 3\ 4\ 6\ 7\ 0\rangle\)

1: \(\langle 5\ 5\ 5\ 5\ 5\ 5\ 5\ 5\rangle\)

2: \(\langle 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\rangle\)

3: \(\langle 7\ 6\ 5\ 4\ 3\ 2\ 1\ 0\rangle\)

A: \(\langle 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\rangle\)

M: \(\langle 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15\ 16\ 17\ 18\ 19\ 20\ 21\ 22\ 23\ 24\ 25\ 26\ 27\ 28\ 29\ 30\ 31\rangle\)

\(\langle\text{Copy } (\text{Vec 2}) \ ' (\text{a a a a a a a a})\rangle\)

\(\langle\text{a 1 2 3 4 5 6 a}\rangle\)

\(\langle\text{Copy ' (a a a a a a a a) } (\text{Vec 2})\rangle\)

\(\langle 0\ a\ a\ a\ a\ a\ a\ 7\rangle\)

**SetVector**: \((\text{SetVector } a\text{Addr } \text{vector})\) initialize the vector positioned in ARRAY at \(a\text{Addr}\) with the value \(\text{vector}\) in any active positions (only where \(\text{ACTIVE}[i] = 0\)).

\(\langle\text{SetVector 1 ' (4 4 3 3 6 6 7 7)\rangle}\)
\(\langle 5\ 4\ 3\ 3\ 6\ 6\ 7\ 5\rangle\)

\(\langle pp\rangle\)

0: \(\langle 2\ 1\ 4\ 3\ 4\ 6\ 7\ 0\rangle\)

1: \(\langle 5\ 4\ 3\ 3\ 6\ 6\ 7\ 5\rangle\)

2: \(\langle 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\rangle\)

3: \(\langle 7\ 6\ 5\ 4\ 3\ 2\ 1\ 0\rangle\)

A: \(\langle 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\rangle\)

M: \(\langle 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15\ 16\ 17\ 18\ 19\ 20\ 21\ 22\ 23\ 24\ 25\ 26\ 27\ 28\ 29\ 30\ 31\rangle\)

The first and the last 5 in the vector 1 are untouched, because the first and the last cell in the system are inactive (\(A[0] = A[7] = 0\)).

**SetAll**: \((\text{SetAll } a\text{Addr } \text{vector})\) initialize the vector positioned in ARRAY at \(a\text{Addr}\) with the value \(\text{vector}\).

\(\langle\text{SetAll 1 ' (9 9 9 9 9 9 9 9 9 9)\rangle}\)
\(\langle 9\ 9\ 9\ 9\ 9\ 9\ 9\ 9\ 9\ 9\rangle\)

\(\langle pp\rangle\)

0: \(\langle 2\ 1\ 4\ 3\ 4\ 6\ 7\ 0\rangle\)

1: \(\langle 9\ 9\ 9\ 9\ 9\ 9\ 9\ 9\ 9\ 9\rangle\)

2: \(\langle 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\rangle\)

3: \(\langle 7\ 6\ 5\ 4\ 3\ 2\ 1\ 0\rangle\)

A: \(\langle 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\rangle\)

M: \(\langle 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15\ 16\ 17\ 18\ 19\ 20\ 21\ 22\ 23\ 24\ 25\ 26\ 27\ 28\ 29\ 30\ 31\rangle\)

All cells in vector 1 are set on 9, whatever the content of ACTIVE vector is.
12.4. HIGH LEVEL ARCHITECTURE FOR CONNEXARRAY™

**SetStream** : (SetStream mAddr stream) sets the content of MEMORY at stream starting with the address mAddr

> (SetStream 23 '(a b c d e))
(a b c d e)
> (pp)

0: (2 1 4 3 4 6 7 0)
1: (9 9 9 9 9 9 9 9)
2: (0 1 2 3 4 5 6 7)
3: (7 6 5 4 3 2 1 0)
A: (1 0 0 0 0 0 0 1)
M: (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)

Starting from the location 23 in MEMORY the stream a b c d e is initialized.

12.4.4 Transfer functions

**Basic transfer functions**

**StoreVector** : (StoreVector aAddr mAddr) store in MEMORY, starting with mAddr, the vector located in ARRAY at aAddr.

> (StoreVector 2 24)
(0 1 2 3 4 5 6 7)
> (pp)

0: (2 1 4 3 4 6 7 0)
1: (9 9 9 9 9 9 9 9)
2: (0 1 2 3 4 5 6 7)
3: (7 6 5 4 3 2 1 0)
A: (1 0 0 0 0 0 0 1)
M: (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)

**LoadVector** : (LoadVector aAddr mAddr) load in ARRAY at the location aAddr the P-length stream of data starting in MEMORY at mAddr.

> (LoadVector 1 16)
(16 17 18 19 20 21 22 a)
> (pp)

0: (2 1 4 3 4 6 7 0)
1: (16 17 18 19 20 21 22 a)
2: (0 1 2 3 4 5 6 7)
3: (7 6 5 4 3 2 1 0)
A: (1 0 0 0 0 0 0 1)
M: (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)

**StoreVectorStrided** : (StoreVectorStrided aAddr mAddr burst stride) store in MEMORY, starting with mAddr, the vector located in ARRAY at a Addr in bursts of burst words with address having the stride stride.

> (StoreVectorStrided 2 0 2 5)
((6 7) (4 5) (2 3) (0 1))
> (pp)

0: (2 1 4 3 4 6 7 0)
1: (5 5 5 5 5 5 5 5)
2: (0 1 2 3 4 5 6 7)
3: (7 6 5 4 3 2 1 0)
A: (1 0 0 0 0 0 0 1)
M: (0 1 2 3 4 2 3 7 8 9 4 5 12 13 14 6 7 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)
CHAPTER 12. # * GLOBAL-LOOP SYSTEMS

LoadVectorStrided : (LoadVectorStrided aAddr mAddr burst stride) load in ARRAY at the location aAddr the stream of data from MEMORY starting at mAddr in bursts of burst words gathered with the stride stride.

> (test)
> (LoadVectorStrided 1 2 2 6)
(2 3 8 9 14 15 20 21)
> (pp)
0:(2 1 4 3 4 6 7 0)
1:(2 3 8 9 14 15 20 21)
2:(0 1 2 3 4 5 6 7)
3:(7 6 5 4 3 2 1 0)
A:(1 0 0 0 0 0 0 1)
M:(0 1 2 3 4 2 3 7 8 9 4 5 12 13 14 6 7 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)

12.4.5 Arithmetic functions

The arithmetic functions are defined on scalars or on vectors. Therefore, the variables x and y are considered in any combinations:

- (x y) : (scalar scalar) returning scalar; the operations is performed by the controller
- (x y) : (scalar vector) returning vector; the operation is performed in array with the scalar operand provided by the controller
- (x y) : (vector scalar) returning vector; the operation is performed in array with the scalar operand provided by the controller
- (x y) : (vector vector) returning vector; the operation is performed in array

Abs : (Abs x) computes the absolute value of x. If x is scalar, then returns a scalar. If x is a vector, the returns a vector of absolute values.

(test)
> (SetVector 1 '(-1 -2 -3 -4 5 6 7 8))
(5 -2 -3 -4 5 6 7 5)
> (SetVector 1 (Abs (Vec 1)))
(5 2 3 4 5 6 7 5)
> (pp)
0:(2 1 4 3 4 6 7 0)
1:(5 2 3 4 5 6 7 5)
2:(0 1 2 3 4 5 6 7)
3:(7 6 5 4 3 2 1 0)
A:(1 0 0 0 0 0 0 1)
M:(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)
> (SetStream 24 '(-7 -8 -9))
(-7 -8 -9)
> (pp)
0:(2 1 4 3 4 6 7 0)
1:(5 2 3 4 5 6 7 5)
2:(0 1 2 3 4 5 6 7)
3:(7 6 5 4 3 2 1 0)
A:(1 0 0 0 0 0 0 1)
M:(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)
> (SetStream 24 (Abs (Mem 24)))
(7)
> (pp)
0:(2 1 4 3 4 6 7 0)
12.4. \textit{HIGH LEVEL ARCHITECTURE FOR CONNEXARRAY}\textsuperscript{TM}

\begin{verbatim}
1: (5 2 3 4 5 6 7 5)
2: (0 1 2 3 4 5 6 7)
3: (7 6 5 4 3 2 1 0)
A: (1 0 0 0 0 0 0 1)
M: (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 7 -8 -9 27 28 29 30 31)
\end{verbatim}

**Inc** \(\text{Inc} \ x\) adds 1 to all components of \(x\).

**Dec** \(\text{Dec} \ x\) subtract 1 form each components of \(x\).

**Add** \((\text{Add} \ x \ y)\) defines the addition in four forms, depending of how the operands are, as follows:
- if \((x \ y) = (\text{scalar} \ \text{scalar})\), then add the two scalars
- if \((x \ y) = (\text{scalar} \ \text{vector})\), then add \(x\) to each component of \(y\)
- if \((x \ y) = (\text{vector} \ \text{scalar})\), then add \(y\) to each component of \(x\)
- if \((x \ y) = (\text{vector} \ \text{vector})\), then add the two vectors

\begin{verbatim}
> (Add 2 8)
10
> (Add '1 2 3) '(2 3 4))
(3 5 7)
> (Add 3 '(3 4 5))
(6 7 8)
> (Add '(2 4 6) 5)
(7 9 11)
> (Add '(1 3 5) '(2 4 6))
(3 7 11)
> (SetVector 2 (Add (Vec 2) (Mem 5)))
(0 6 7 8 9 10 11 7)
> (pp)
0: (2 1 4 3 4 6 7 0)
1: (5 2 3 4 5 6 7 5)
2: (0 6 7 8 9 10 11 7)
3: (7 6 5 4 3 2 0)
A: (1 0 0 0 0 0 1)
M: (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 7 -8 -9 27 28 29 30 31)
\end{verbatim}

**Sub** \((\text{Sub} \ x \ y)\) defines the subtract in four forms, depending of how the operands are, as follows:
- if \((x \ y) = (\text{scalar} \ \text{scalar})\), then subtract \(y\) from \(x\)
- if \((x \ y) = (\text{scalar} \ \text{vector})\), then subtract from \(x\) each component of \(y\) providing the vector containing the resulting differences
- if \((x \ y) = (\text{vector} \ \text{scalar})\), then subtract \(y\) from each component of \(x\)
- if \((x \ y) = (\text{vector} \ \text{vector})\), then subtract the vector \(y\) from the vector \(x\)

**Mult** \((\text{Mult} \ x \ y)\) multiply \(x\) by \(y\).

\begin{verbatim}
> (Mult 2 -6)
-12
> (Mult 2 '(1 2 3 4))
(2 4 6 8)
> (Mult '(2 3 4 5) -3)
\end{verbatim}
(-6 -9 -12 -15)
> (Mult '(1 3 5 7) '(1 3 5 7))
(1 9 25 49)
> (Mult '(1 2) '(1 2 3))
.map: all lists must have same size;

**Div**: \((\text{Div } x \ y)\) divide \(x\) by \(y\), providing the quotient as a real number.

> (Div 7 3.0) 2.3333333333333335
> (Div '(1 3 5) 2.0)
(0.5 1.5 2.5)
> (Div 2 '(1.0 3 5))
(2.0 2/3 2/5)
> (Div 2.0 '(1 3 5))
(2.0 0.6666666666666666 0.4)
> (Div '(1.0 3.0 5.0) '(2 4 6))
(0.5 0.75 0.8333333333333334)

**Quotient**: \((\text{Quotient } x \ y)\) integer division of \(x\) by \(y\)

> (Quotient 7 3) 2
> (Quotient '(1 3 5) 2)
(0 1 2)
> (Quotient 2 '(1 3 5))
(2 0 0)
> (Quotient '(1 3 5) '(2 4 6))
(0 0 0)

**Rem**: \((\text{Rem } x \ y)\) provides the remaining of the division of \(x\) by \(y\)

> (Rem 3 2) 1
> (Rem '(1 3 5) 2)
(1 1 1)
> (Rem 2 '(1 3 5))
(0 2 2)
> (Rem '(3 5 7) '(2.0 3.0 4.0))
(1.0 2.0 3.0)

**Not**: \((\text{Not } x)\)

**And**: \((\text{And } x \ y)\)

**Or**: \((\text{Or } x \ y)\)

**Xor**: \((\text{Xor } x \ y)\)

### 12.4.6 Reduction functions

Reduction functions are defined on vectors returning scalars (they “reduces” a vector to a scalar).
12.4. HIGH LEVEL ARCHITECTURE FOR CONNEXARRAY™

RedAdd : \(\text{RedAdd} \ (\text{vector})\) adds all the selected (where ACTIVE == 0) components of vector.

\[
> \ (\text{test})
> \ (\text{SetStream} \ 16 \ (\text{RedAdd} \ (\text{Vec} \ 1)))
> \ (\text{pp})
0:(2 1 4 3 4 6 7 0)
1:(5 5 5 5 5 5 5 5)
2:(0 1 2 3 4 5 6 7)
3:(7 6 5 4 3 2 1 0)
A:(1 0 0 0 0 0 0 1)
M:(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 30 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)
\]

RedAddAll : \(\text{RedAddAll} \ (\text{vector})\) adds all components of vector (the content of ACTIVE vector is ignored).

\[
> \ (\text{SetStream} \ 16 \ (\text{RedAddAll} \ (\text{Vec} \ 1)))
> \ (\text{pp})
(40)
\]

RedMax : \(\text{RedMax} \ (\text{vector})\) returns the maximum value from the selected components (where ACTIVE = 0) of vector.

\[
> \ (\text{RedMax} \ (\text{Vec} \ 3))
6
\]

RedMaxAll : \(\text{RedMaxAll} \ (\text{vector})\) returns the maximum value from the components of vector (the content of ACTIVE vector is ignored).

\[
> \ (\text{RedMaxAll} \ (\text{Vec} \ 3))
7
\]

Select : \(\text{Select} \ (i \ \text{vector})\) selects the \(i\)-th component from the active components (where ACTIVE = 0) of vector vector.

\[
> \ (\text{Select} \ 3 \ (\text{Vec} \ 3))
4
> \ (\text{Select} \ 0 \ (\text{Vec} \ 3))
\text{ERROR--SELECT}
\]

SelectFromAll : \(\text{SelectFromAll} \ (i \ \text{vector})\) selects the \(i\)-th component from the vector vector (the content of ACTIVE vector is ignored).

AnyActive : \(\text{AnyActive}\) return 1 only if at least one component of ACTIVE is 0.

\[
> \ (\text{AnyActive})
1
> \ (\text{SetActive} \ (\text{Vec} \ 1))
> \ (\text{AnyActive})
0
\]
12.4.7 Test functions

The test functions are defined on scalars or on vectors. Therefore, the variables $x$ and $y$ are considered in any combinations:

- $(x \ y) : (\text{scalar scalar})$ returning scalar; the operations is performed by the controller and the result is used to control the program running on controller
- $(x \ y) : (\text{scalar vector})$ returning vector; the operation is performed in array with the scalar operand provided by the controller and the resulting vector is used to make selections in array
- $(x \ y) : (\text{vector scalar})$ returning vector; the operation is performed in array with the scalar operand provided by the controller and the resulting vector is used to make selections in array
- $(x \ y) : (\text{vector vector})$ returning vector; the operation is performed in array and the resulting vector is used to make selections in array

Let $(Lt \ x \ y)$ defines the Less than comparison in four forms, depending of how the operands are, as follows:

- if $((x \ y) = (\text{scalar scalar}))$, then compare the two scalars returning 1 if $x < y$
- if $((x \ y) = (\text{scalar vector}))$, then compare $x$ to each component of $y$ returning a Boolean vector
- if $((x \ y) = (\text{vector scalar}))$, then compare $y$ to each component of $x$ returning a Boolean vector
- if $((x \ y) = (\text{vector vector}))$, then add the two vectors returning a Boolean vector

```
(test)
> (Lt 2 4)
1
> (Lt (Vec 0) (Vec 1))
(1 1 1 1 0 0 0 0)
> (Lt (Vec 1) (Vec 0))
(0 0 0 1 1 0 0 0)
> (Lt (Vec 1) 3)
(1 1 1 1 0 0 0 0)
> (Lt 3 (Vec 0))
(0 0 1 0 1 1 1 0)
> (SetStream 15 (Lt 3 (Vec 0)))
(0 0 1 0 1 1 1 0)
> (SetVector 1 (Lt 3 (Vec 0)))
(5 0 1 0 1 1 1 5)
> (pp)
0:(2 1 4 3 4 6 7 0)
1:(5 0 1 0 1 1 5)
2:(0 1 2 3 4 5 6 7)
3:(7 6 5 4 3 2 1 0)
A:(1 0 0 0 0 0 1)
M:(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 0 0 0 1 0 1 1 0 23 24 25 26 27 28 29 30 31)
```

Let $(Gt \ x \ y)$ defines the Greater than comparison in the four forms.

Leq : $(\text{Leq \ x \ y})$ defines the Less than or equal comparison in the four forms.

Geq : $(\text{Geq \ x \ y})$ defines the Greater than or equal comparison in the four forms.
12.4. HIGH LEVEL ARCHITECTURE FOR CONNEXARRAY™

**Eq** : defines the *Equal* comparison in the four forms. *(Eq x y)*

**Even?** : *(Even? x)* returns 1 for even integers and 0 for odd integers

### 12.4.8 Control functions

**ResetActive** : *(ResetActive)* set the value of the vector ACTIVE to *(MakeAll 0)*

> (ResetActive)
> (pp)
> 0:(2 1 4 3 4 6 7 0)
> 1:(5 5 5 5 5 5 5 5)
> 2:(0 1 2 3 4 5 6 7)
> 3:(8 9 10 11 12 13 14 15)
> A:(0 0 0 0 0 0 0 0)
> M:(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)

**SetActive** : *(SetActive vector)* set the value of the vector ACTIVE to *vector*

> (SetActive '(1 0 0 1 1 0 0 1))
> (pp)
> 0:(2 1 4 3 4 6 7 0)
> 1:(5 0 1 0 1 1 1 5)
> 2:(0 1 2 3 4 5 6 7)
> 3:(7 6 5 4 3 2 1 0)
> A:(1 0 0 1 0 0 0 1)
> M:(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)

**SaveActive** : *(SaveActive vector)* save the value of the vector ACTIVE in *vector*

... TBD

**Where** : *(Where booleanVector)* restrain the selection according to the Boolean vector *vector*. The ACTIVE vector takes the value of the vector *(Not booleanVector)*

**EndWhere** : *(EndWhere)* resets ACTIVE.

**ElseWhere** : *(ElseWhere)* complements the components of the vector ACTIVE.

**First** : *(First)* all the 0s in ACTIVE, except the first, if any, are turned to 1.

> (SetActive '(1 1 1 0 0 0 1))
> (First)
> (pp)
> 0:(2 1 4 3 4 6 7 0)
> 1:(5 5 5 5 5 5 5 5)
> 2:(0 1 2 3 4 5 6 7)
> 3:(7 6 5 4 3 2 1 0)
> A:(1 1 1 0 0 1 1 1)
> M:(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)

**FirstIndex** : *(FirstIndex)* returns the index of the first cell where ACTIVE[i] == 0

> (SetActive '(1 1 1 0 0 0 1))
> (FirstIndex)
> 4
12.4.9 Stream functions

Permute : (Permute v1 v2), permutes the vector v1, according to the indexes provided by the vector v2

> (Permute (Vec 2) '(5 6 7 1 2 5 5 5))
(5 6 7 1 2 5 5 5)

ShiftLeft : (ShiftLeft many vector) shifts left the content of the vector vector with many positions and insert 0s in the right end. The vector ACTIVE is ignored.

> (ShiftLeft 3 (Vec 2))
(3 4 5 6 7 0 0 0)
> (SetVector 1 (ShiftLeft 3 (Vec 2)))
(5 4 5 6 7 0 0 5)

ShiftLeftEl : (ShiftLeftEl many vector el) shifts left the content of the vector vector with many positions and insert el symbols in the right end. The vector ACTIVE is ignored.

ShiftRight : (ShiftRight many vector) shifts right the content of the vector vector with many positions and insert 0s in the left end. The vector ACTIVE is ignored.

> (SetVector 1 (ShiftRight 3 (Vec 2)))
(5 0 0 0 1 2 3 5)
> (pp)
0:(2 1 4 3 4 6 7 0)
1:(5 0 0 0 1 2 3 5)
2:(0 1 2 3 4 5 6 7)
3:(8 9 10 11 12 13 14 15)
A:(1 0 0 0 0 0 0 1)
M:(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)

ShiftRightEl : (ShiftRightEl many vector el) shifts right the content of the vector vector with many positions and insert el symbols in the left end. The vector ACTIVE is ignored.

RotateLeft : (RotateLeft many vector) rotate left vector with many positions. The vector ACTIVE is ignored.

> (SetVector 1 (RotateLeft 3 (Vec 2)))
(5 4 5 6 7 0 1 5)
> (pp)
0:(2 1 4 3 4 6 7 0)
1:(5 4 5 6 7 0 1 5)
2:(0 1 2 3 4 5 6 7)
3:(7 6 5 4 3 2 1 0)
A:(1 0 0 0 0 0 0 1)
M:(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)
12.4. **HIGH LEVEL ARCHITECTURE FOR CONNEXARRAY™**

**RotateRight**: (RotateRight many vector) rotate right vector with many positions. The vector ACTIVE is ignored.

> (SetVector 1 (RotateRight 3 (Vec 2)))
(5 6 7 0 1 2 3 5)
>
**Search**: (Search el vector) in the active cells of vector the symbol el is searched: all ACTIVE[i], except where ((ACTIVE[i] == 0) & (vector[i] == el)), are set to 1.

... // TBD

**SearchCond**: (SearchCond el vector) performs a conditional search: where ((ACTIVE[i-1] == 0) & (vector[i] == el)) ACTIVE[i] = 0, elsewhere ACTIVE[i] = 1.

... // TBD

**InsertPos**: (InsertPos stream at vector) inserts the stream stream in the position at in the vector vector. The vector ACTIVE is ignored.

> (InsertPos '(a 7 a) 2 (Vec 3))
(7 6 a 7 a 5 4 3 2 1 0)
> (SetVector 3 (InsertPos '(a 7 a) 2 (Vec 3)))
(7 6 a 7 a 5 4 0)
> (pp)
0:(2 1 4 3 4 6 7 0)
1:(5 6 7 0 1 2 3 5)
2:(0 1 2 3 4 5 6 7)
3:(7 6 a 7 a 5 4 0)
A:(1 0 0 0 0 0 0 1)
M:(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)

**DeletePos**: (DeletePos many at vector) deletes many components starting form at in vector filling the empty positions with dummy symbols.

> (SetVector 1 (DeletePos 2 3 (Vec 2)))
(5 1 2 5 6 7 0 1)
CHAPTER 12. *GLOBAL-LOOP SYSTEMS*

### 12.5 Index

GLOBAL VARIABLES:
- (define (SYSTEM) (list ARRAY MEMORY))
- (define ARRAY()); array of vectors
- (define MEMORY()); array of scalars
- (define ACTIVE()); vector of u/d counters
- (define INDEX()); the index vector
- (define M 0); m = memory size
- (define P 0); p = vector size
- (define N 0); n= number of vectors

- (define (ResetSystem n p m)
  (set! ARRAY (make-list n (make-list p '-)))
  (set! MEMORY (make-list m '-))
  (set! ACTIVE (make-list p 1))
  (set! N n); number of vectors in ARRAY
  (set! P p); length of a vector
  (set! M m); number of scalars in MEMORY
  (set! INDEX (Index P)))

- (define (rs) (ResetSystem 4 8 32)) (define (test)
  (ResetSystem 4 8 32)(newline)
  (SetVector 0 '(2 1 4 3 4 6 7 0))
  (pp))

AUXILIARY FUNCTIONS
- (pp); pretty printing
- (Index n)
- (MakeAll val)

ADDRESSING FUNCTIONS:
- (Vec aAddr); ARRAY[aAddr]
- (Mem mAddr); MEMORY[mAddr]

SETTING FUNCTIONS:
- (Copy vSource vDest)
- (SetVector aAddr vector); ARRAY[aAddr]<=vector
- (SetAll aAddr vector); !!!!!!!!!!!!!!!!!!!! TBD
- (SetStream mAddr stream)

TRANSFER FUNCTIONS:
- (StoreVector aAddr mAddr)
- ; MEMORY[mAddr:mAddr+P] <= (Vec aAddr)
- (LoadVector aAddr mAddr)
- ; (Vec aAddr) <= MEMORY[mAddr:mAddr+P]
- (StoreVectorStrided aAddr mAddr burst stride)
- (LoadVectorStrided aAddr mAddr burst stride)

ARITHMETIC & LOGIC FUNCTIONS:
- (Abs x)
- (Inc x)
- (Dec x)
- (Add x y)
- (Sub x y)
- (Mult x y)
- (Div x y)
- (Quotient x y)
- (Rem x y)
- (Not x)
- (And x y)
- (Or x y)
- (Xor x y)

REDUCTION FUNCTIONS:
- (RedAdd vector)
- (RedAddAll vector)
- (RedMax vector)
- (RedMaxAll vector)
- (Select i vector)
- (SelectFromAll i vector); TBD
- (AnyActive); -> 1 if at least one is active

TEST FUNCTIONS:
- (Lt x y)
- (Gt x y)
- (Leq x y)
- (Geq x y)
- (Eq x y)
- (Even? x)

CONTROL FUNCTIONS
- (ResetActive); ACTIVE <= (0 0 ... 0)
- (SetActive vector); ACTIVE <= vector
- (SaveActive booleanVector); TBD
- (Where vector)
- (EndWhere)
- (ElseWhere)
- (First); -> leaves just first active
- (FirstIndex); -> index of the first active

STREAM FUNCTIONS:
- (Permute v1 v2)
- (ShiftLeft many vector)
- (ShiftLeftEl many vector el); TBD
- (ShiftRight many vector)
- (ShiftRightEl many vector el); TBD
- (RotateLeft many vector)
- (RotateRight many vector)
(Search el vector)
(SearchCond el vector)
(InsertPos stream at vector)
(DeletePos many at vector)

**NON-CONNEX FUNCTIONS:**
(make-list n x); list of n x
(cut v pos n); cut from pos n elem in vect v
(CheckZero x); -> #t if all elem. of v are 0
(put v pos w); set w on pos in vector v
12.6 Problems

Problem 12.1

Problem 12.2

Problem 12.3
Part III

ANNEXES
Appendix A

Boolean functions

Searching the truth, dealing with numbers and behaving automatically are all based on logic. Starting from the very elementary level we will see that logic can be “interpreted” arithmetically. We intend to offer a physical support for both the numerical functions and logical mechanisms. The logic circuit is the fundamental brick used to build the physical computational structures.

A.1 Short History

There are some significant historical steps on the way from logic to numerical circuits. In the following some of them are pointed.

Aristotle of Stagira (382-322) a Greek philosopher considered as founder for many scientific domains. Among them logics. All his writings in logic are grouped under the name Organon, that means instrument of scientific investigation. He worked with two logic values: true and false.

George Boole (1815-1864) is an English mathematician who formalized the Aristotelian logic like an algebra. The algebraic logic he proposed in 1854, now called Boolean logic, deals with the truth and the false of complex expressions of binary variables.

Claude Elwood Shannon (1916-2001) obtained a master degree in electrical engineering and PhD in mathematics at MIT. His Master’s thesis, A Symbolic Analysis of Relay and Switching Circuits [Shannon ’38], used Boolean logic to establish a theoretical background of digital circuits.

A.2 Elementary circuits: gates

Definition A.1 A binary variable takes values in the set \{0, 1\}. We call it bit.

The set of numbers \{0, 1\} is interpreted in logic using the correspondences: 0 \rightarrow false, 1 \rightarrow true in what is called positive logic, or 1 \rightarrow false, 0 \rightarrow true in what is called negative logic. In the following we use positive logic.

Definition A.2 We call n-bit binary variable an element of the set \{0, 1\}^n.

Definition A.3 A logic function is a function having the form f : \{0, 1\}^n \rightarrow \{0, 1\}^m with n \geq 0 and m > 0.

In the following we will deal with m = 1. The parallel composition will provide the possibility to build systems with m > 1.
A.2.1 Zero-input logic circuits

**Definition A.4** The 0-bit logic function are \( f_0^0 = 0 \) (the false-function) which generates the one bit coded 0, and \( f_1^0 = 1 \) (the true-function) which generate the one bit coded 1.

They are useful for generating initial values in computation (see the zero function as basic function in partial recursivity).

A.2.2 One input logic circuits

**Definition A.5** The 1-bit logic functions, represented by true-tables in Figure A.1, are:

- \( f_0^1(x) = 0 \) – the false function
- \( f_1^1(x) = x' \) – the invert (not) function
- \( f_2^1(x) = x \) – the driver or identity function
- \( f_3^1(x) = 1 \) – the true function

<table>
<thead>
<tr>
<th>( x )</th>
<th>( f_0^1 )</th>
<th>( f_1^1 )</th>
<th>( f_2^1 )</th>
<th>( f_3^1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure A.1: One-bit logic functions. 

- **a.** The truth table for 1-variable logic functions.
- **b.** The circuit for “0” (false) by connecting to the ground potential.
- **c.** The logic symbol for the inverter circuit.
- **d.** The logic symbol for driver function.
- **e.** The circuit for “1” (true) by connecting to the high potential.

Numerical interpretation of the NOT circuit: **one-bit incremenetr.** Indeed, the output represents the modulo 2 increment of the inputs.

A.2.3 Two inputs logic circuits

**Definition A.6** The 2-bit logic functions are represented by true-tables in Figure A.2.

Interpretations for some of 2-input logic circuits:

- \( f_2^2 \) : AND function is:
  - a multiplier for 1-bit numbers
  - a **gate**, because \( x \) opens the gate for \( y \):
    - if \( x = 1 \) output = \( y \); else output = 0;

- \( f_3^2 \) : XOR (exclusiv OR) is:
  - the 2-modulo adder
  - NEQ (not-equal) circuit, a comparator pointing out when the two 1-bit numbers on the input are unequal
  - an enabled inverter:
    - if \( x = 1 \) output is \( y' \); else output is \( y \);
  - a modulo 2 incremenetr.
### A.3. HOW TO DEAL WITH LOGIC FUNCTIONS

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Figure A.2: Two-bit logic functions.

- **a.** The table of all two-bit logic functions.
- **b.** AND gate – the original gate.
- **c.** NAND gate – the most used gate.
- **d.** OR gate.
- **e.** NOR gate.
- **f.** XOR gate – modulo2 adder.
- **g.** NXOR gate – coincidence circuit.

- $f_B^2$: the logic implication is also used to compare 1-bit numbers because the output is 1 for $y < x$
- $f_I^2$: NOR function detects when 2-bit numbers have the value zero.

All logic circuits are *gates*, even if a true gate is only the AND gate.

### A.2.4 Many input logic circuits

For enumerating the 3-input function a table with 8 line is needed. On the left side there are 3 columns and on the right side 256 columns (one for each 8-bit binary configuration defining a logic function).

**Theorem A.1** The number of $n$-input one output logic (Boolean) functions is $N = 2^{2^n}$. ⋄

Enumerating is not a solution starting with $n = 3$. Maybe the 3-input function can be defined using the 2-input functions.

### A.3 How to Deal with Logic Functions

The systematic and formal development of the *theory* of logical functions means: (1) a set of elementary functions, (2) a minimal set of axioms (of formulas considered true), and (3) some rule of deduction.

Because our approach is a pragmatic one: (1) we use an extended (non-minimal) set of elementary functions containing: NOT, AND, OR, XOR (a minimal one contains only NAND, or only NOR), and (2) we will list a set of useful principles, i.e., a set of *equivalences*.

**Identity principle** Even if the natural tendency of existence is becoming, we stone the value $a$ to be identical with itself: $a = a$. Here is one of the fundamental limits of digital systems and of computation based on them.

**Double negation principle** The negation is a “reversible” function, i.e., if we know the output we can deduce the input (it is a very rare, somehow unique, feature in the world of logical function): $(a')' = a$. Actually, we can not found the reversibility in existence. There are logics that don’t accept this principle (see the intuitionist logic of Heyting & Brower).
**Associativity**  Having 2-input gates, how can be built gates with much more inputs? For some functions the associativity helps us.

\[
a + (b + c) = (a + b) + c = a + b + c
\]

\[
a(bc) = (ab)c = abc
\]

\[
a ⊕ (b ⊕ c) = (a ⊕ b) ⊕ c = a ⊕ b ⊕ c.
\]

**Commutativity**  Commutativity allows us to connect to the inputs of some gates the variable in any order.

\[
a + b = b + a
\]

\[
ab = ba
\]

\[
a ⊕ b = b ⊕ a
\]

**Distributivity**  Distributivity offers the possibility to define all logical functions as sum of products or as product of sums.

\[
a(b + c) = ab + ac
\]

\[
a + bc = (a + b)(a + c)
\]

\[
a(b ⊕ c) = ab ⊕ ac.
\]

Not all distributions are possible. For example:

\[
a ⊕ bc ≠ (a ⊕ b)(b ⊕ c).
\]

The table in Figure A.3 can be used to prove the previous inequality.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>bc</th>
<th>a ⊕ bc</th>
<th>a ⊕ b</th>
<th>a ⊕ c</th>
<th>(a ⊕ b)(a ⊕ c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure A.3: Proving by tables.  Proof of inequality \(a ⊕ bc ≠ (a ⊕ b)(b ⊕ c)\).

**Absorption**  Absorption simplify the logic expression.

\[
a + a' = 1
\]

\[
a + a = a
\]

\[
aa' = 0
\]

\[
aa = a
\]

\[
a + ab = a
\]

\[
a(a + b) = a
\]

Tertium non datur: \(a + a' = 1\).

**Half-absorption**  The half-absorption allows only a smaller, but non-neglecting, simplification.

\[
a + a'b = a + b
\]

\[
a(a' + b) = ab.
\]
A.4. MINIMIZING BOOLEAN FUNCTIONS

Substitution  The substitution principles say us what happen when a variable is substituted with a value.

\[
\begin{align*}
    a + 0 &= a \\
    a + 1 &= 1 \\
    a0 &= 0 \\
    a1 &= a \\
    a \oplus 0 &= a \\
    a \oplus 1 &= a'.
\end{align*}
\]

Exclusion  The most powerful simplification occurs when the exclusion principle is applicable.

\[
\begin{align*}
    ab + a'b &= b \\
    (a + b)(a' + b) &= b.
\end{align*}
\]

Proof.  For the first form:

\[
ab + a'b = b
\]

applying successively distribution, absorption and substitution results:

\[
ab + a'b = b(a + a') = b1 = b.
\]

For the second form we have the following sequence:

\[
(a + b)(a' + b) = (a + b)a' + (a + b)b = aa' + a'b + ab + bb = \\
0 + (a'b + ab + b) = a'b + ab + b = a'b + b = b.
\]

De Morgan laws  Some times we are interested to use inverting gates instead of non-inverting gates, or conversely. De Morgan laws will help us.

\[
\begin{align*}
    a + b &= (a'b')' \\
    ab &= (a' + b')' \\
    a' + b' &= (ab)' \\
    a'b' &= (a + b)'
\end{align*}
\]

A.4  Minimizing Boolean functions

Minimizing logic functions is the first operation to be done after defining a logical function. Minimizing a logical function means to express it in the simplest form (with minimal symbols). To a simple form a small associated circuit is expected. The minimization process starts from canonical forms.

A.4.1  Canonical forms

The initial definition of a logic function is usually expressed in a canonical form. The canonical form is given by a truth table or by the rough expression extracted from it.

Definition A.7  A minterm associated to an n-input logic function is a logic product (AND logic function) depending by all n binary variable.

Definition A.8  A maxterm associated to an n-input logic function is a logic sum (OR logic function) depending by all n binary variable.

Definition A.9  The disjunctive normal form, DNF, of an n-input logic function is a logic sum of minterms.

Definition A.10  The conjunctive normal form, CNF, of an n-input logic function is a logic product of maxterms.
Example A.1  Let be the combinational multiplier for 2 2-bit numbers described in Figure A.4. One number is the 2-bit number \( \{a, b\} \) and the other is \( \{c, d\} \). The result is the 4-bit number \( \{p_3, p_2, p_1, p_0\} \). The logic equations result direct as 4 DNFs, one for each output bit:

- \( p_3 = abcd \)
- \( p_2 = ab'cd' + ab'cd + abcd' \)
- \( p_1 = a'b'cd' + a'b'cd + ab'c'd + ab'cd + abc'd + abcd' \)
- \( p_0 = a'b'c'd' + a'b'cd + ab'c'd + ab'cd + abc'd + abcd' \)

Indeed, the \( p_3 \) bit takes the value 1 only if \( a = 1 \) and \( b = 1 \) and \( c = 1 \) and \( d = 1 \). The bit \( p_2 \) is 1 only one of the following three 4-input ADNs takes the value 1: \( ab'cd' \), \( ab'cd \), \( abcd' \). And so on for the other bits.

Applying the De Morgan rule the equations become:

- \( p_3 = \overline{(ab'cd')'(abcd')'(abcd')'(abcd')} \)
- \( p_2 = \overline{(ab'cd')'(abcd')'(abcd')'(abcd')} \)
- \( p_1 = \overline{(a'b'cd')'(ab'c'd')'(ab'cd')'(abc'd')'(abcd')'(abcd')} \)
- \( p_0 = \overline{(a'b'cd')'(abcd')'(abcd')'(abcd')} \)

These forms are more efficient in implementation because involve the same type of circuits (NANDs), and because the inverting circuits are usually faster.

<table>
<thead>
<tr>
<th>ab</th>
<th>cd</th>
<th>p3</th>
<th>p2</th>
<th>p1</th>
<th>p0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure A.4: Combinational circuit represented as a truth table. The truth table of the combinational circuit performing 2-bit multiplication.

The resulting circuit is represented in Figure A.5. It consists in two layers of ADNs. The first layer computes only minterms and the second “adds” the minterms thus computing the 4 outputs.

The logic depth of the circuit is 2. But in real implementation it can be bigger because of the fact that big input gates are composed from smaller ones. Maybe a real implementation has the depth 3. The propagation time is also influenced by the number of inputs and by the fan-out of the circuits.

The size of the resulting circuit is very big also: \( S_{\text{mult}2} = 54 \).

A.4.2 Algebraic minimization

Minimal depth minimization

Example A.2 Let’s revisit the previous example for minimizing independently each function. The least significant output has the following form:

\[ p_0 = a'b'c'd' + a'b'cd + abc'd + abcd \]

We will apply the following steps:

\[ p_0 = (a'b'd)c' + (a'bd)c + (abd)c' + (abd)c \]

to emphasize the possibility of applying twice the exclusion principle, resulting

\[ p_0 = a'bd + abd. \]
A.4. MINIMIZING BOOLEAN FUNCTIONS

Figure A.5: Direct implementation of a combinational circuit. The direct implementation starting from DNF of the 2-bit multiplier.

Applying again the same principle results:

\[ p_0 = bd(a' + a) = bd \]

The exclusion principle allowed us to reduce the size of the circuit from 22 to 2.

We continue with the next output:

\[ p_1 = a'b'cd' + a'b'cd + ab'c'd + ab'cd + abc'd + abcd' = \]
\[ = a'b(\neg d + d) + ab'd(\neg c + c) + abc'd + abcd' = \]
\[ = ab' + ab'd + abc'd + abcd' = \]
\[ = bc(a' + ad') + ad(b' + bc') = \]
\[ = bc(a' + d') + ad(b' + c') = \]
\[ = a'b'c + ab'cd = \]
\[ = ab'c + acd' \]

The p3 output can not be minimized. De Morgan law is used to transform the expressions to be implemented with NANDs.

\[ p_3 = ((abcd)'')' \]
\[ p_2 = ((ab'c)'(acd')')' \]
\[ p_1 = ((a'bc)'(bcd)'(ab'd)'(ac'd')')' \]
\[ p_1 = ((abcd')')' \]

Results the circuit from Figure A.6.

Multi-level minimization

Example A.3 The same circuit for multiplying 2-bit numbers is used to exemplify the multilevel minimization.

Results:

\[ p_3 = abcd \]
Figure A.6: **Minimal depth minimization** The first, minimal depth minimization of the 2-bit multiplier.

\[
p_2 = ab'c + ac'd' = ac(b' + d') = ac(bd)'
\]
\[
p_1 = a'bc + bc'd' + ab'd + ac'd = bc(a' + d') + ad(b' + c') = bc(ad)' + ad(bc)' = (bc) \oplus (ad)
\]
\[
p_0 = bd.
\]

Using for XOR the following form:

\[
x \oplus y = ((x \oplus y)'')' = (x'y' + x'y)'' = (xy)'(x'y)' = (xy)'(x+y)
\]

results the circuit from Figure A.7 with size 22.

Figure A.7: **Multi-level minimization.** The second, multi-level minimization of the 2-bit multiplier.

**Many output circuit minimization**

**Example A.4** Inspecting carefully the schematics from Figure A.7 results: (1) the output \( p_3 \) can be obtained inverting the NAND’s output from the circuit of \( p_2 \), (2) the output \( p_0 \) is computed by a part of the circuit used for \( p_2 \). Thus, we are encouraged to rewrite some of the functions in order to maximize the common circuits used in implementation. Results:

\[
x \oplus y = (xy)'(x+y) = ((xy) + (x+y))'.
\]
\[
p_2 = ac(bd)' = ((ac)' + bd)'
\]

allowing the simplified circuit from Figure A.8. The size is 16 and the depth is 3. But, more important: (1) the circuits contains only 2-input gates and (2) the maximum fan-out is 2. Both last characteristics led to small area and high speed.
A.4. MINIMIZING BOOLEAN FUNCTIONS

A.4.3 Veitch-Karnaugh diagrams

In order to apply efficiently the exclusion principle we need to group carefully the minterms. Two dimension diagrams allow to emphasize the best grouping. Formally, the two minterms are adjacent if the Hamming distance in minimal.

Definition A.11 The Hamming distance between two minterms is given by the total numbers of binary variable which occur distinct in the two minterms. ♦

Example A.5 The Hamming distance between \( m_9 = ab'c'd \) and \( m_4 = a'bc'd' \) is 3, because only the variable \( b \) occurs in the same form in both minterms.

The Hamming distance between \( m_9 = ab'c'd \) and \( m_1 = a'b'c'd \) is 1, because only the variable which occurs distinct in the two minterms is \( a \). ♦

Two \( n \)-variable terms having the Hamming distance 1 are minimized, using the exclusion principle, to one \((n-1)\)-variable term. The size of the associated circuit is reduced from \( 2^n \) to \( n-1 \).

A \( n \)-input Veitch diagram is a two dimensioned surface containing \( 2^n \) squares, one for each \( n \)-value minterm. The adjacent minterms (minterms having the Hamming distance equal with 1) are placed in adjacent squares. In Figure A.9 are presented the Veitch diagrams for 2, 3 and 4-variable logic functions. For example, the 4-input diagram contains in the left half all minterms true for \( a = 1 \), in the upper half all minterms true for \( b = 1 \), in the two middle columns all the minterms true for \( c = 1 \), and in the two middle lines all the minterms true for \( d = 1 \). Results the lateral columns are adjacent and the lateral line are also adjacent. Actually the surface can be seen as a toroid.

Example A.6 Let be the function \( p1 \) and \( p2 \), two outputs of the 2-bit multiplier. Rewriting them using minterms results:

\[
p1 = m_6 + m_7 + m_9 + m_{11} + m_{13} + m_{14}
\]
\[ p_2 = m_{10} + m_{11} + m_{14}. \]

In Figure A.10 \( p_1 \) and \( p_2 \) are represented.

\[ \begin{array}{c|cc|c}
    & a & b & c \\
\hline
  p_1 & 0 & 0 & 0 \\
  & 1 & 0 & 0 \\
  & 0 & 1 & 0 \\
  & 1 & 1 & 0 \\
\end{array} \quad \begin{array}{c|cc|c}
    & a & b & c \\
\hline
  p_2 & 1 & 1 & 1 \\
  & 1 & 0 & 1 \\
  & 1 & 1 & 1 \\
\end{array} \]

Figure A.10: Using Veitch diagrams. The Veitch diagrams for the functions \( p_1 \) and \( p_2 \).

The Karnaugh diagrams have the same property. The only difference is the way in which the minterms are assigned to squares. For example, in a 4-input Karnaugh diagram each column is associated to a pair of input variable and each line is associated with a pair containing the other variables. The columns are numbered in Gray sequence (successive binary configurations are adjacent). The first column contains all minterms true for \( ab = 00 \), the second column contains all minterms true for \( ab = 01 \), the third column contains all minterms true for \( ab = 11 \), the last column contains all minterms true for \( ab = 10 \). A similar association is made for lines. The Gray numbering provides a similar adjacency as in Veitch diagrams.

\[ \begin{array}{c|cc|c}
    & a & b & c \\
\hline
  m_0 & 0 & 0 & 0 \\
  m_1 & 0 & 1 & 0 \\
  m_2 & 0 & 1 & 1 \\
  m_3 & 1 & 0 & 1 \\
  m_4 & 1 & 0 & 0 \\
  m_5 & 1 & 1 & 1 \\
  m_6 & 1 & 1 & 0 \\
\end{array} \]

Figure A.11: Karnaugh diagrams. The Karnaugh diagrams for 3 and 4 variables.

In Figure A.12 the same functions, \( p_1 \) and \( p_2 \), are represented. The distribution of the surface is different but the degree of adjacency is identical.

In the following we will use Veitch diagrams, but we will name them V-K diagrams to be fair with both Veitch and Karnaugh.

Minimizing with V-K diagrams

The rule to extract the minimized form of a function from a V-K diagram supposes:

- to define:
  - the smallest number
  - of rectangular surfaces containing only 1’s
A.4. MINIMIZING BOOLEAN FUNCTIONS

Figure A.12: **Using Karnaugh diagrams.** The Karnaugh diagrams for the functions p1 and p2.

- including all the 1's
- each surface having a maximal area
- and containing a power of two number of 1's

- to extract the logic terms (logic product of Boolean variables) associated with each previously emphasized surface
- to provide de minimized function adding logically (logical OR function) the terms associated with the surfaces.

Figure A.13: **Minimizing with V-K diagrams.** Minimizing the functions $p_1$ and $p_2$.

**Example A.7** Let’s take the V-K diagrams from Figure A.10. In the V-K diagram for $p_1$ there are four 2-square surfaces. The upper horizontal surface is included in the upper half of V-K diagram where $b = 1$, it is also included in the two middle columns where $c = 1$ and it is included in the surface formed by the two horizontal edges of the diagram where $d = 0$. Therefore, the associated term is $bcd'$ which is true for: $(b = 1) \text{AND}(c = 1) \text{AND}(d = 0)$.

Because the horizontal edges are considered adjacent, in the V-K diagram for $p_2$ $m_{14}$ and $m_{10}$ are adjacent forming a surface having $acd'$ as associated term.

The previously known form of $p_1$ and $p_2$ result if the terms resulting from the two diagrams are logically added.

Minimizing incomplete defined functions

There are logic functions incompletely defined, which means for some binary input configurations the output value does not matter. For example, the designer knows that some inputs do not occur anytime. This lack in definition
can be used to make an advanced minimization. In the V-K diagrams the corresponding minterms are marked as “don’t care”s with “-”. When the surfaces are maximized the “don’t care”s can be used to increase the area of 1’s. Thus, some “don’t care”s will take the value 1 (those which are included in the surfaces of 1’s) and some of “don’t care”s will take the value 0 (those which are not included in the surfaces of 1’s).

![V-K diagrams](image)

**Figure A.14:** Minimizing incomplete defined functions. **a.** The minimization of $y$ (Example 1.8) ignoring the “don’t care” terms. **b.** The minimization of $y$ (Example 1.8) considering the “don’t care” terms.

**Example A.8** Let be the 4-input circuit receiving the binary coded decimals (from 0000 to 1001) indicating on its output if the received number is contained in the interval $[2, 7]$. It is supposed the binary configurations from 1010 to 1111 are not applied on the input of the circuit. If by hazard the circuit receives a meaningless input we do not care about the value generated by the circuit on its output.

In Figure A.14a the V-K diagram is presented for the version ignoring the “don’t care”s. Results the function: $y = a'b + a'c = a'(b + c)$.

If “don’t care”s are considered results the V-K diagram from Figure A.14b. Now each of the two surfaces are doubled resulting a more simplified form: $y = b + c$.

**V-K diagrams with included functions**

For various reasons in a V-K diagram we need to include instead of a logic value, 0 or 1, a logic function of variables which are different from the variables associated with the V-K diagram. For example, a minterm depending on $a, b, c, d$ can be defined as taking a value which is depending on another logic 2-variable function by $s, t$.

A simplified rule to extract the minimized form of a function from a V-K diagram containing included functions is the following:

1. consider first only the 1s from the diagram and the rest of the diagram filed only with 0s and extract the resulting function
2. consider the 1s as “don’t care”s for surfaces containing the same function and extract the resulting function “multiplying” the terms with the function
3. “add” the two functions.

**Example A.9** Let be the function defined in Figure A.15a. The first step means to define the surfaces of 1s ignoring the squares containing functions. In Figure A.15b are defined 3 surfaces which provide the first form depending only by the variables $a, b, c, d$:

$$bc'd + a'bc' + b'c$$

The second step is based on the diagram represented in Figure A.15c, where a surface $(c'd)$ is defined for the function $e'$ and a smaller one $(acd)$ for the function $e$. Results:

$$c'de' + acde$$
The initial form.

b. The form considered in the first step.

c. The form considered in the second step.

In the third step the two forms are “added” resulting:

\[ f(a, b, c, d, e) = bc'd + a'bc'd + b'c + c'de' + acde. \]

Sometimes, an additional algebraic minimization is needed. But, it deserves because including functions in V-K diagrams is a way to expand the number of variable of the functions represented with a manageable V-K diagram.

A.5 Problems

Problem A.1
Appendix B

Basic circuits

Basic CMOS circuits implementing the main logic gates are described in this appendix. They are based on simple switching circuits realized using MOS transistors. The inverting circuit consists in a pair of two complementary transistors (see the third section). The main gates described are the NAND gate and the NOR gate. They are built by appropriately connecting two pairs of complementary MOS transistors (see the fourth section). Tristate buffers generate an additional, third “state” (the Hi-Z state) to the output of a logic circuit, when the output pair of complementary MOS transistors are driven by appropriate signals (see the sixth section). Parallel connecting a pair of complementary MOS transistors provides the transmission gate (see the seventh section).

B.1 Actual digital signals

The ideal logic signals are 0 Volts for false, or 0, and $V_{DD}$ for true, or 1. Real signals are more complex. The first step in defining real parameters is represented in Figure B.1, where is defined the boundary between the values interpreted as 0 and the values interpreted as 1.

![Figure B.1: Defining 0-logic and 1-logic.](image)

This first definition is impossible to be applied because supposes:

$$V_{Hmin} = V_{Lmax}.$$ 

There is no engineering method to apply the previous relation. A practical solution supposes:

$$V_{Hmin} > V_{Lmax}$$

generating a “forbidden region” for any actual logic signal. Results a more refined definition of the logic signals represented in Figure B.2, where $V_L < V_{Lmax}$ and $V_H > V_{Hmin}$.
A robust design asks a net distinction between the electrical values interpreted as 0 and the electrical values interpreted as 1.

In real applications we are faced with nasty realities. A signal generated to the output of a gate is sometimes received to the input of the receiving gate distorted by parasitic signals. In Figure B.3 the noise generator simulate the parasitic effects of the circuits switching in a small neighborhood.

Because of the noise captured from the "environment" a noise margin must be added to expand the forbidden region with two noise margin regions, one for 0 level, $NM_0$, and another for 1 level, $NM_1$. They are defined as follows:

\[ NM_0 = V_{IL} - V_{OL} \]

\[ NM_1 = V_{OH} - V_{IH} \]

making the necessary distinctions between the $V_{OH}$, the 1 at the output of the sender gate, and $V_{IH}$, the 1 at the input of the receiver gate.
B.2 CMOS switches

A logic gates consists in a network of interconnected switches implemented using the two type of MOS transistors: p-MOS and n-MOS. How behaves the two type of transistors in specific configurations is presented in Figure B.4.

A switch connected to $V_{DD}$ is implemented using a p-MOS transistor. It is represented in Figure B.4a off (generating z, which means Hi-Z: no signal, neither 0, nor 1) and in Figure B.4b it is represented on (generating 1 logic, or truth).

A switch connected to ground is implemented using a n-MOS transistor. It is represented in Figure B.4c off (generating z, which means Hi-Z: no signal, neither 0, nor 1) and in Figure B.4e it is represented on (generating 0 logic, or false).

![Figure B.4: Basic switches. a. Open switch connected to $V_{DD}$. b. Closed switch connected to $V_{DD}$. c. Open switch connected to ground. d. Closed switch connected to ground.](image)

A MOS transistor works very well as an on-off switch connecting its drain to a certain potential. A p-MOS transistor can be used to connect its drain to a high potential when its gates is connected to ground, and an n-MOS transistor can connect its drain to ground if its gates is connected to a high potential. This complementary behavior is used to build the elementary logic circuits.

In Figure B.5 is presented the switch-resistor-capacitor model (SRC). If $V_{GS} < V_T$ then the transistor is off, if $V_{GS} \geq V_T$ then the transistor is on. In both cases the input of the transistor behaves like a capacitor, the gate-source capacitor $C_{GS}$.

When the transistor is on its drain-source resistance is:

$$R_{ON} = R_n \frac{L}{W}$$

where: $L$ is the channel length, $W$ is the channel width, and $R_n$ is the resistance per square. The length $L$ is a constant characterizing a certain technology. For example, if $L = 0.13\mu m$ this means it is about a 0.13$\mu m$ process.

The input capacitor has the value:

$$C_{GS} = \frac{\varepsilon_{OX} LW}{d}$$

The value:

$$C_{OX} = \frac{\varepsilon_{OX}}{d}$$

where: $\varepsilon_{OX} \approx 3.9\varepsilon_0$ is the permittivity of the silicon dioxide, is the gate-to-channel capacitance per unit area of the MOSFET gate.
In this conditions the gate input current is:

\[ i_G = C_{GS} \frac{dV_{GS}}{dt} \]

Figure B.5: The MOSFET switch. The switch-resistor-capacitor model consists in the two states: OF (\(V_{GS} < V_T\)), and ON (\(V_{GS} \geq V_T\)). In both states the input is defined by the capacitor \(C_{GS}\).

Example B.1 For an AND gate with low strength, with \(W = 1.8 \mu m\), in 0.13\(\mu m\) technology, supposing \(C_{OX} = 4fF/\mu m^2\), results the input capacitance:

\[ C_{GS} = 4 \times 0.13 \times 1.8fF = 0.936fF \]

Assuming \(R_n = 5K\Omega\), results for the same gate:

\[ R_{ON} = 5 \times \frac{0.13}{1.8} K\Omega = 361\Omega \]

⋄

B.3 The Inverter

B.3.1 The static behavior

The smallest and simplest logic circuit – the inverter – can be built using a pair of complementary transistors, connecting together the two gates as input and the two drains as output, while the n-MOS source is connected to ground (interpreted as logic 0) and the p-MOS source to \(V_{DD}\) (interpreted as logic 1). Results the circuit represented in Figure B.6.

The behavior of the inverter consist in combining the behaviors of the two switches previously defined. For \(in = 0\) pMOS is on and nMOS is off the output generating \(V_{DD}\) which means 1. For \(in = 1\) pMOS is off and nMOS is on the output generating 0.

The static behavior of the inverter (or NOT) circuit can be easy explained starting from the switches described in Figure B.4. Connecting together a switch generating \(z\) with a switch generating 1 or 0, the connection point will generate 0 or 1.
B.3. THE INVERTER

B.3.2 Dynamic behavior

The propagation time of an inverter can be analyzed using the two serially connected inverters represented in Figure B.7. The delay of the first inverter is generated by its capacitive load, $C_L$, composed by:

- its parasitic drain/bulk capacitance, $C_{DB}$, is the intrinsic output capacitance of the first inverter
- wiring capacitance, $C_{wire}$, which depends on the length of the wire (of width $W_w$ and of length $L_w$) connected between the two invertors:
  \[ C_{wire} = C_{thickox}W_wL_w \]
- next stage input capacitance, $C_G$, approximated by summing the gate capacitance for pMOC and nMOS transistors:
  \[ C_G = C_{Gp} + C_{Gn} = C_{ox}(W_pL_p + W_nL_n) \]

The total load capacitance
\[ C_L = C_{DB} + C_{wire} + C_G \]
is sometimes dominated by $C_{wire}$. For short connections $C_G$ dominates, while for big fan-out both, $C_{wire}$ and $C_G$ must be considered.

The signal $V_A$ is used to measure the propagation time of the first NOT in Figure B.7a. It is generated by an ideal pulse generator with output impedance 0. Thus, the rising time and the falling time of this signal are considered 0 (the input capacitance of the NOT circuit is charged or discharged in no time).

The two delay times (see Figure B.7c) associated to an invertor (to a gate in the general case) are defined as follows:

- $t_{pLH}$: the time interval between the moment the input switches in 0 and the output reaches $V_{OH}/2$ coming from 0
- $t_{pHL}$: the time interval between the moment the input switches in 1 and the output reaches $V_{OH}/2$ coming from $V_{OH}$
Let us consider the transition of $V_A$ from 0 to $V_{OH}$ at $t_r$ (rise edge). Before transition, at $t_r^-$, $C_L$ is fully charged and $V_B = V_{OH}$. In Figure B.7b is represented the equivalent circuit at $t_r^+$, when pMOS is off and nMOS is on. In this moment starts the process of discharging the capacitance $C_L$ at the constant current

$$I_{Dn(sat)} = \frac{1}{2} \mu_n C_{ox} \frac{W_n}{L_n} (V_{OH} - V_{Tn})^2$$

In Figure B.8, at $t_r^-$ the transistor is cut, $I_{Dn} = 0$. At $t_r^+$ the nMOS transistor switch in saturation and becomes an ideal constant current generator which starts to discharge $C_L$ linearly at the constant current $I_{Dn(sat)}$. The process continue until $V_{OUT} = V_{OH}$, according to the definition of $t_{pHL}$.

In order to compute $t_{pHL}$ we take into consideration the constant value of the discharging current which provide a linear variation of $V_{OUT}$.

$$\frac{dv_{out}}{dt} = \frac{d}{dt} \left( \frac{q_L}{C_L} \right) = \frac{-I_{Dn(sat)}}{C_L}$$
We solve the equations for $t_{pHL}$:

$$t_{pHL} = C_L \frac{1}{\mu_n C_{ox} \frac{W_n}{T_n} (V_{OH} - V_{TN})} \frac{V_{OH}}{V_{OH} - V_{TN}}$$

Because:

$$R_{ONn} = \frac{1}{\mu_n C_{ox} \frac{W_n}{T_n} (V_{OH} - V_{TN})}$$

results:

$$t_{pHL} = C_L R_{ONn} \frac{1}{1 - \frac{V_{OH}}{V_{OH}}} = k_n R_{ONn} C_L = k_n \tau_{nl}$$

where:

- $\tau_{nl}$ is the constant time associated to the H-L transition
- $k_n$ is a constant associated to the technology we use; it goes down when $V_{OH}$ increases or $V_T$ decreases

The speed of a gate depends by its dimension and by the capacitive load it drives. For a big $W$ the value of $R_{ON}$ is small charging or discharging $C_L$ faster.

For $t_{pLH}$ the approach is similar. Results: $t_{pLH} = k_p \tau_{pl}$.

By definition the propagation time associated to a circuit is:

$$t_p = (t_{pLH} + t_{pHL})/2$$

its value being dominated by the value of $C_L$ and the size (width) of the two transistors, $W_n$ and $W_p$.

### B.3.3 Buffering

It is usual to be confronted, in designing a big systems, with the buffering problem: a logic signal generated by a small, “weak” driver must be used to drive a big, “strong” circuit (see Figure B.9a) maintaining in the same time a high clock frequency. The driver is an inverter with a small $W_n = W_p = W_{drive}$ (to make the model simple), unable to provide an enough small $R_{ON}$ to move fast the charge from the load capacitance of a circuit with a big $W_n = W_p = W_{load}$. Therefore the delay introduced between A and B is very big. For our simplified model,

$$t_p = t_{p0} \frac{W_{load}}{W_{driver}}$$

where: $t_{p0}$ is the propagation time when the driver circuit and the load circuit are of the same size.

The solution is to interpose, between the small driver and the big load, additional drivers with progressively increased area as in Figure B.9b. The logic is preserved, because two NOTs are serially connected. While the no-buffer solution provides, between A and B, the propagation time:

$$t_{p(no\text{-buffer})} = t_{p0} \frac{W_{load}}{W_{driver}}$$

the buffered solution provide the propagation time:

$$t_{p(buffered)} = t_{p0} \left( \frac{W_1}{W_{drive}} + \frac{W_2}{W_1} + \frac{W_{load}}{W_2} \right)$$

How are related the area of the circuits in order to obtain a minimal delay, i.e., how are related $W_{driver}$, $W_1$ and $W_2$? The relation is given by the minimizing of the delay introduced by the two intermediary circuits. Then, the first derivative of

$$\frac{W_2}{W_1} + \frac{W_{load}}{W_2}$$
must be 0. Results:
\[ W_2 = \sqrt{W_1 W_{\text{load}}} \]
\[ \frac{W_2}{W_1} = \frac{W_{\text{load}}}{W_2} = \sqrt{\frac{W_{\text{load}}}{W_1}} \]

We conclude: in order to add a minimal delay, the size ratio of successive drivers in a chain must be the same.

Figure B.9: **Buffered connection.** a. An invertor with small \( W \) is not able to handle at high frequency a circuit with big \( W \). b. The buffered connection with two intermediary buffers.

In order to design the size of the circuits in Figure B.9b, let us consider \( \frac{W_{\text{load}}}{W_{\text{driver}}} = n \). Then,
\[ \frac{W_1}{W_{\text{driver}}} = \frac{W_2}{W_1} = \frac{W_{\text{load}}}{W_2} = \sqrt{n} \]

The acceleration is
\[ \alpha = \frac{t_p(\text{no-buffer})}{t_p(\text{buffered})} = \frac{\sqrt{n^2}}{3} \]

For example, for \( n = 1000 \) the acceleration is \( \alpha = 33.3 \).

The hand calculation, just presented, is approximative, but has the advantages to provide an intuitive understanding about the propagation phenomenon, with emphasis on the buffering mechanism.

The price for the acceleration obtained by buffering is the area and energy consumed by the two additional circuits.

### B.3.4 Power dissipation

There are three major physical processes involved in the energy requested by a digital circuit to work:

- **switching energy**: due to charging and discharging of load capacitances, \( C_L \)
- **short-circuit energy**: due to non-zero rise/fall times of the signals
- **leakage current energy**: which becomes more and more important with the decreasing of device sizes

From the power supply, which provide \( V_{DD} \) with enough current, the circuit absorbs as much as needed current.

#### Switching power

The average switching power dissipated is the energy dissipated in a clock cycle divided by the clock cycle time, \( T \). Suppose the clock is applied to the input of an invertor. When \( \text{clock} = 0 \) the load capacitor is loaded from the power supply with the charge:

\[ Q_L = C_L V_{DD} \]

We assume in \( T/2 \) the capacitor is charged (else the frequency is too big for the investigated circuit). During the next half-period, when \( \text{clock} = 1 \), the same charge is transferred from the capacitor to ground. Therefore the
B.3. THE INVERTER

Figure B.10: The main power consuming process. For \( V_{in} = 0 \) \( C_L \) is loaded by the current provided by \( R_{ONp} \). The charge from \( C_L \) is transferred to the ground through \( R_{ONn} \) for \( V_{in} = V_{OH} \).

charge \( Q_L \) is transferred from \( V_{DD} \) to ground in the time \( T \). The amount of energy used for this transfer is \( V_{DD} Q_L \), and the switching power results:

\[
p_{\text{switch}} = \frac{V_{DD} C_L V_{DD}}{T} = C_L V_{DD}^2 f_{\text{clock}}
\]

While a big \( V_{OH} = V_{DD} \) helped us in reducing \( t_p \), now we have difficulties due to the square dependency of switching power by the same \( V_{DD} \).

Short-circuit power

When the output of the inverter switches between the two logic levels, for a very short time interval around the moment when \( V_{OUT} = V_{DD}/2 \), both transistors have \( I_{DD} \neq 0 \) (see Figure B.11). Thus is consumed the short-circuit power.

Figure B.11: Direct flow of current from \( V_{DD} \) to ground. This current due to the non-zero edge to the circuit input can be neglected.

The amount of power wasted by these temporary short-cuts is:

\[
p_{sc} = I_{DD(\text{mean})} V_{DD}
\]

where \( I_{DD(\text{mean})} \) is the mean value of the current spikes. If the edge of the signal is short and the mean frequency of switchings is low, then the resulting value is low.
Leakage power

The last source of energy waste is generated by the leakage current. It will start to be very important in sub 65nm technologies (for 65nm the leakage power is 40% of the total power consumption). The leakage current and the associated power is increasing exponentially with each new technology generation and is expected to become the dominant part of total power. Device threshold voltage scaling, shrinking device dimensions, and larger circuit sizes are causing this dramatic increase in leakage. Thus, increasing the amount of leakage is critical for power constraint integrated circuits.

Figure B.12: The two main components of the leakage current.

\[ P_{\text{leakage}} = I_{\text{leakage}} V_{DD} \]

where \( I_{\text{leakage}} \) is the sum of subthreshold and gate oxide leakage current. In Figure B.12 the two components of the leakage current are presented for a NOT circuit with \( V_{in} = 0 \).

B.4 Gates

The 2-input AND circuit, \( a \cdot b \), works like a “gate” opened by the signal \( a \) for the signal \( b \). Indeed, the gate is “open” for \( b \) only if \( a = 1 \). This is the reason for which the AND circuit was baptised gate. Then, the use imposed this alias as the generic name for any logic circuit. Thus, AND, OR, XOR, NAND, ... are all called gates.

B.4.1 NAND & NOR gates

The static behavior of gates

For 2-input NAND and 2-input NOR gates the same principle will be applied, interconnecting 2 pairs of complementary transistors to obtain the needed behaviors.

There are two kind of interconnecting rules for the same type of transistors, p-MOS or n-MOS. They can be interconnected serially or parallel.

A serial connection will establish an on configuration only if both transistors of the same type are on, and the connection is off if at least one transistor is off.

A parallel connection will establish an on configuration if at least one is on, and the connection is off only if both are off.

Applying the previous rules result the circuits presented in Figures B.13 and B.14.

For the NAND gate the output is 0 if both n-MOS transistors are on, and the output is one when at least on p-MOS transistor is on. Indeed, if \( A = B = 1 \) both \( n \) transistors are on and both \( p \) transistors are off. The output
corresponds with the definition, it is 0. If $A = 0$ or $B = 0$ the output is 1, because at least one $p$ transistor is on and at least one $n$ transistor is off.

A similar explanation works for the NOR gate. The main idea is to design a gate so as to avoid the simultaneous connection of $V_{DD}$ and ground potential to the output of the gate.

For designing an AND or an OR gate we will use an additional NOT connected to the output of an AND or an OR gate. The area will be a little bigger (maybe!), but the strength of the circuit will be increased because the NOT circuit works as a buffer improving the time performance of the non-inverting gate.

The propagation time for the 2-input main gates is computed in a similar way as the propagation for NOT circuit is computed. The only differences are due to the fact that sometimes $R_{ON}$ must be substituted with $2 \times R_{ON}$.

**Propagation time**

**Propagation time for NAND gate**

becomes, in the worst case when only one input switches:

$$t_{HL} = k_n(2R_{ONn})C_L$$

$$t_{LH} = k_p(R_{ONp})C_L$$

because the capacitor $C_L$ is charged through one pMOS transistor and is discharged through two, serially connected, nMOS transistors.

**Propagation time for NOR gate**

becomes, in the worst case when only one input switches:

$$t_{HL} = k_n(R_{ONn})C_L$$

$$t_{LH} = k_p(2R_{ONp})C_L$$

because the capacitor $C_L$ is charged through two, serially connected, pMOS transistors and is discharged through one nMOS transistor.

It is obvious that we must prefer, when is is possible, the use of NAND gates instead of NOR gates, because, for the same area, $R_{ONp} > R_{ONn}$. 
Figure B.14: The NOR gate. a. The internal structure of a NOR gate: the output is 1 only when both inputs are 0. b. The logic symbol for NOR.

Power consumption & switching activity

The power consumption is determined by the 0 to 1 transitions of the output of a logic gate. The problem is meaningless for a NOT circuit because the transitions of the output has the same probability as of the transition of the input. But, for a \(n\)-input gate the probability of an output transition depends on the function performed by the gate.

For a certain gate, with unbiased 0 and 1 applied on the inputs, the output probability of switching from 0 to 1, \(P_{0\rightarrow1}\), is given by the logic function. We define switching activity, \(\sigma\), this probability of switching from 0 to 1.

Switching activity for 2-input AND with the inputs A and B is:

\[
\sigma = P_{0\rightarrow1} = P_{OUT=0}P_{OUT=1} = (1 - P_A P_B) P_A P_B
\]

where: \(P_A\) is the probability of having 1 on the input A, \(P_B\) is the probability of having 1 on the input B, and \(P_{OUT=0}\) is the probability of having 0 on output, while \(P_{OUT=1} = P_{AB}\) is the probability of having 1 on output (see Figure B.15a).

Figure B.15: Switching activity \(\sigma\) and the output probability of 1. a. For 2-input AND. b. For 3-input AND. c. For 4-input AND.

If the input are not conditioned, \(P_A = P_B = 0.5\), then the switching activity for a 2-input NAND is \(\sigma_{\text{NAND2}} = 3/16\) (see Figure B.15a).
Switching activity for 3-input AND with the inputs A, B, and C is $\sigma_{\text{NAND}} = 7/64$ (see Figure B.158). The probability of 1 to the output of a 3-input AND is only 1/8 leading to a smaller $\sigma$.

Switching activity for n-input AND is:

$$\sigma_{\text{NANDn}} = \frac{2^n - 1}{2^n} \approx 1/2^n$$

The switching activity decreases exponentially with the number of inputs in AND, OR, NAND, NOR gates. This is a very good news.

Now, we must reconsider the computation of the power substituting $C_L$ with $\sigma C_L$:

$$P_{\text{switch}} = \sigma C_L V_{DD}^2 f_{\text{clock}}$$

In big systems, a conservative assumption is that the mean value of the inputs of the logic gates is 3, and, therefore a global value for switching activity could be $\sigma_{\text{global}} \approx 1/8$. Actual measurements provide frequently $\sigma_{\text{global}} \approx 1/10$.

Power consumption & glitching

In the previous paragraph we learned that the output of a circuit switch due to the change on the inputs. This is an ideal situation. Depending on the circuit configuration and on the various delays introduced by gates, unexpected “activity” manifests sometimes in our network of gates. See the simple example form Figure B.16. From the logical point of view, when the inputs switch form $ABC = 010$ to $ABC = 111$ the output must maintain its value on 1. Unfortunately, because the effect of the inputs A and B are affected by the extra delay introduced by the first gate, the unexpected glitch manifests to the output. Follow the wave forms form Figure B.16 to understand why.

The glitch is undesired for various reasons. The most important are two:

- the signal can be latched by a memory circuit (such an elementary latch), thus triggering the switch of a memory circuit; a careful design can avoid this effect
- the temporary, useless transition discharge and charge back the load capacitor increasing the energy consumed by the circuit.

Let us go back to the Zero circuit represented in two versions in Figure 2.1c and Figure 2.1d. We have now an additional reason to prefer the second version. The balanced delays to the inputs of the intermediary circuits allow us to avoid almost totaly the glitching contribution to the power consumption.
B.4.2 Many-Input Gates

How can be built 3-input NAND or a 3-input NOR applying the same rule? For a 3-input NAND 3 n-MOS transistors will be connected serially and 3 p-MOS transistors will be connected parallel. Similar for the 3-input NOR gate.

How “much” this rule can be applied to built \( n \)-input gates? Not too much because of the propagation time which is increased when too many serially connected \( R_{ON} \) resistors will be used to transfer the electrical charge in or out from the load capacitor \( C_L \). A 4-input NAND, for example, discharges \( C_L \) trough 4 serially connected \( R_{ONn} \), while a 4-input NOR loads \( C_L \) with a constant time \( 4R_{ONp}C_L \). The mean worst case (when only one input switches) time constants used to compute \( t_p \) become:

\[
(4RON + R_{ONp})/2
\]

for NAND, and

\[
(4RONp + R_{ONn})/2
\]

for NOR.

Fortunately, there is another way to increase the number of inputs of a certain gate. It is by composing the function using an appropriate number of 2-input gates organized as a balanced binary tree.

![Diagram](image)

**Figure B.17: How to manage a many-input gate.** a An NAND\(_8\) gate with fan-out \( n \). b. The log-depth equivalent circuit.

For example, an 8-input NAND gate, see Figure B.17a, is recommended to be designed as a binary tree of two input gates, see Figure B.17b, as follows:

\[
(a \cdot b \cdot c \cdot d \cdot e \cdot f \cdot g \cdot h)' = (((((a \cdot b)' + (c \cdot d)')' \cdot ((e \cdot f)' + (g \cdot h)')')')
\]

The form results as the application of the De Morgan law.

In the first case, represented in Figure B.17a, an 8-input NAND uses a similar arrangement as in Figure B.13a, where instead of two parallel connected pMOS transistors and two serially connected nMOS transistors are used 8 pMOSs and 8 nMOSs. Generally speaking, for each new input an additional pair, nMOS & pMOS, is added.
Increasing in this way the number of inputs the propagation time is increased linearly because of the serially connected channels of the nMOS transistors. The load capacitor is discharged to the ground through \( R_{ON} \), where \( m \) represents the number of inputs.

The second solution, see Figure B.17b, is to build a balanced tree of gates. In the first case the propagation time is in \( O(n) \), while in the second it is in \( O(\log n) \) for implementations using transistors having the same size.

For an \( m \)-input gate results a \( \log_2 m \) depth network of 2-input gates. For example, see Figure B.17, where an 8-input NAND is implemented using a 3-level network of gates (first to the 8-input gate the divide & impera principle is applied, and then the De Morgan rule transformed the first level of four ANDs in four NANDs and the second level of two ANDs in two NORs). While the maximum propagation time for the 8-input NAND is

\[
t_{pHL(\text{one-level})} = k_n \times 8 \times R_{ONn} \times (n \times C_{in})
\]

where \( C_{in} \) is the value of the input capacitor in a typical gate and \( n \) is the fan-out of the circuit, the maximum propagation time for the equivalent log-depth net of gates is

\[
t_{pHL(\text{log-levels})} = k_n \times ((2 \times 2 \times R_{ONn} \times C_{in}) + 2 \times R_{ONn} \times (n \times C_{in}))
\]

For \( n = 3 \) results a 2.4 times faster circuit if the log-depth version is adopted, while for \( n = 4 \) the acceleration is 2.67.

Generally, for fan-in equal with \( m \) and fan-out equal with \( n \) result the acceleration for the log-depth solutions, \( \alpha \), expressed by the formula:

\[
\alpha = \frac{m \times n}{2 \times (n - 1 + \log m)}
\]

Example: \( n = 4, m = 32, \alpha = 8 \).

The log-depth circuit has two advantages:

- the intermediary \((-1 + \log m)\) stages are loaded with a constant and minimal capacitor \(-C_{in}\) – given by only one input
- only the final stage drives the real load of the circuit \(-n \times C_{in}\) – but its driving capability does not depend by fan-in.

Various other solutions can be used to speed-up a many-input gate. For example:

\[
(a \cdot b \cdot c \cdot d \cdot e \cdot f \cdot g \cdot h)' = (((a \cdot b \cdot c \cdot d)' + (e \cdot f \cdot g \cdot h)')')'
\]

could be a better solution for an 8-input NAND, mainly because the output is generated by a NOT circuit and the internal capacitors are minimal, making the 4-input NANDs harmless.

### B.4.3 AND-NOR gates

For implementing the logic function:

\[
(AB + CD)'
\]

besides the solution of composing it from the previously described circuits, there is a direct solution using 4 CMOS pairs of transistors, one associated for each input. The resulting circuit is represented in Figure B.18.

The size of the circuit according to Definition 2.2 is 4. (Implementing the function using 2 NANDs, 2 invertors, and a NOR provides the size 8. Even if the de Morgan rule is applied results 3 NANDs and and invertor, which means the size is 7.)

The same rule can be applied for implementing any NOR of ANDs. For example, the circuit performing the logic function

\[
f(A, B, C) = (A(B + C))'
\]

has a simple implementation using a similar approach. The price will be the limited speed or the over-dimensioned transistors.
Appendix B. Basic Circuits

The AND-NOR gate.

Figure B.18: The AND-NOR gate. a. The circuit. b. The logic symbol for the AND-NOR gate.

B.5 The Tristate Buffers

A tristate circuit has the output able to generate three values: 0, 1, x (which means nothing). The output value x is unable to impose a specific value, we say the output of the circuit is unconnected or it is off.

Two versions of this kind of circuit are presented in Figures B.19 and B.20.

The inverting version of the tristate buffer uses one additional pair of complementary transistors to disconnect the output from any potential. If enable = 0 the CMOS transistors connected to the output are both off. Only if enable = 1 the circuit works as an inverter.

For the non-inverting version the two additional logic gates are used to control the gates of the two output transistors. Only if enable = 0 the two logic gates transfer the input signal inverted to the gates of the two output transistor.

B.6 The Transmission Gate

A simple and small version of a gate is the transmission gate which works connecting directly the signal from a source to a destination. Figure B.21a represents the CMOS version. If enable = 1 then out = in because at least one transistor is on. If in = 0 the signal is transmitted by the n-MOS transistor, else, if in = 1 the signal is transmitted by the p-MOS transistor.

The transmission gate is not a regenerative gate in contrast to the previously described gates which were regenerative gates. A transmission gate performs a true two-direction electrical connection, with all its goods and bad involved.

The main limitation introduced by the transmission gate is its $R_{ON}$ which is serially connected to the $C_L$ increasing the constant time associated to the delay.

The main advantage of this gate is the absence of a connection to the ground or to $V_{DD}$. Thus, the energy consumed by this gate is lowered.

One of the frequently used application of the transmission gate is the inverting multiplexor (see Figure B.21c). The two transmission gates are enabled by in a complementary mode. Thus, only one gate is active at a time, avoiding the “fight” of two opposite signals to impose the value to the inverter’s input.
Figure B.19: Tristate inverting buffer. a. The circuit. b. The logic symbol for the inverting tristate buffer. 
   c. Two-direction connection on one wire. For \( \text{enable} = 1 \), \( \text{in/out} = \text{out}' \), while for \( \text{enable} = 0 \), \( \text{in} = \text{in/out}' \). d. Interconnecting two systems. For \( \text{en1} = 1 \), \( \text{en2} = 0 \), System 1 sends and System 2 receives; for \( \text{en1} = 0 \), \( \text{en2} = 1 \), System 2 sends and System 1 receives; \( \text{en1} = \text{en2} = 0 \) both systems are receivers, while \( \text{en1} = \text{en2} = 1 \) is not allowed.

When the propagation time is not critical the use of this gate is recommended because, both, area and power are saved.
APPENDIX B. BASIC CIRCUITS

Figure B.20: **Tristate non-inverting buffer.** a. The circuit. b. The logic symbol for the non-inverting tristate buffer.

Figure B.21: **The transmission gate.** a. The complementary transmission gate. b. The logic symbol. c. An application: the elementary inverting multiplexer.

B.7 Memory Circuits

B.7.1 Flip-flops

Data latches and their transparency

Master-slave DF-F

Resetable DF-F

B.7.2 # Static memory cell

B.7.3 # Array of cells

B.7.4 # Dynamic memory cell

B.8 Problems

Gates

Problem B.1
Figure B.22: **Data latches.**  

a. Transparent from D to Q (D = Q) for ck = 0. For ck = 1 the loop is closed and D input has no effect on output.  
b. Transparent from D to Q for ck = 1. For ck = 0 the loop is closed and D input has no effect on output.

Problem B.2  
Problem B.3  
Problem B.4  

Flop-flops  
Problem B.5  
Problem B.6  
Problem B.7  
Problem B.8
APPENDIX B. BASIC CIRCUITS

Figure B.23: Master-slave delay flip-flop (DF-F) with the clock signal active on the positive transition. a. Implemented with data latches based on transmission gates. b. The equivalent schematic for $ck = 0$. c. The equivalent schematic for $ck = 1$.

Figure B.24: Master-slave delay flip-flop with asynchronous reset.
Appendix C

Standard cell libraries

Modern design procedures suppose standard cell libraries containing gates, latches, flip-flops, and many other frequently used typical modules.

C.1 Main parameters

Each component of a cell library is characterized for area, propagation time, and power. Each factory has three type of processes: slow, typical, and fast. A derating factor is associated with each process:

\[
K_{\text{process (typical)}} = 1
\]
\[
K_{\text{process (slow)}} = 1.3
\]
\[
K_{\text{process (fast)}} = 0.8
\]

The power supply variation introduces two derating factors, one for the negative variations and another for the positive variation:

\[
K_{\text{volt}}(-10\%) = -2.1/V
\]
\[
K_{\text{volt}}(+10\%) = -1.4/V
\]

The temperature variation introduce the last two derating factors:

\[
K_{\text{temp}}(\text{from } 25^0\text{C to } -40^0\text{C}) = 0.0009\text{C}
\]
\[
K_{\text{temp}}(\text{from } 25^0\text{C to } 125^0\text{C}) = 0.0007\text{C}
\]

Figure C.1 represents how the propagation time, \(t_p\), is measured in combinational circuits.

Figure C.2 represent temporal relations in a clocked circuit, where the active edge of clock is the positive edge. The frequency of clock is:

\[
f_c = \frac{1}{t_{\text{ckw1}} + t_{\text{ckw0}}}
\]

The following parameters are defined:

- **set-up time**: \(t_{st}\) because a clocked circuit must receive the input data before the active edge of clock (the circuit must “learn” before what to do)
- **hold time**: \(t_H\) because the input data can not disappears immediately after the active edge of clock (the circuit must “understand” firmly what to do)

\(^1\)The values used in the following are typical values for a 13\(\mu\)m process in a good factory.
APPENDIX C. STANDARD CELL LIBRARIES

Figure C.1: Propagation time in combinational circuits.

**recovery time**: $t_R$ is necessary to change the control of the circuit from the imperium of set/reset devices to the imperium of clock (the set/reset operation is very “stressful” and the circuits must recover).

**propagation time**: $t_{pHL}$ or $t_{pLH}$ is the response time of the output of the circuit.

The propagation time for a circuit can be estimated using the following forms:

$$t_p = K_{process} \times (1 + K_{volt} \times \Delta V_{DD}) \times (1 + K_{temp} \times \Delta T) \times t_{typical}$$

where:

- $t_p$ – propagation time in $\text{ns}$ taking into account all derating factors
- $t_{typical}$ – propagation time in $\text{ns}$ at nominal $V_{DD}$, $25^0\text{C}$, for typical process
- $t_{intrinsic}$ – circuit delay in $\text{ns}$ with no loaded output
- $K_{load}$ – load factor expressed in $\text{ns}/\text{pF}$
- $C_{load}$ – output load expressed in $\text{pF}$.

### C.2 Basic cells

A standard library contains cells having the same height. An usual value is around $3.7\mu m$. The differences between inputs are ignored. The power is expressed for each input pin in $\mu W/\text{MHz}$. The tables contain mean typical values.

#### C.2.1 Gates

**Two input AND gate: AND2**

**Implementation**: $y = ((a \cdot b)^{t'})$ 

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width($\mu m$)</th>
<th>$t_{intrinsic}(LH/HL)$</th>
<th>$Z_{load}(LH/HL)$</th>
<th>$C_{pin}$</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.8</td>
<td>0.053 / 0.073</td>
<td>4.28 / 2.2</td>
<td>0.0014</td>
<td>0.0044</td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
<td>0.047 / 0.069</td>
<td>1.9 / 1.1</td>
<td>0.002</td>
<td>0.006</td>
</tr>
<tr>
<td>4</td>
<td>2.8</td>
<td>0.044 / 0.057</td>
<td>0.96 / 0.56</td>
<td>0.004</td>
<td>0.012</td>
</tr>
</tbody>
</table>
Two input OR gate: OR2
Implementation : \( y = ((a + b)')' \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width (( \mu m ))</th>
<th>( t_{\text{intrinsic}}(LH/HL) )</th>
<th>( Z_{\text{load}}(LH/HL) )</th>
<th>( C_{\text{pin}} )</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.8</td>
<td>0.045 / 0.095</td>
<td>4.2 / 2.3</td>
<td>0.0015</td>
<td>0.004</td>
</tr>
<tr>
<td>2</td>
<td>2.3</td>
<td>0.04 / 0.085</td>
<td>1.9 / 1.25</td>
<td>0.0025</td>
<td>0.006</td>
</tr>
<tr>
<td>4</td>
<td>2.8</td>
<td>0.04 / 0.075</td>
<td>1.1 / 0.6</td>
<td>0.004</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Three input AND gate: AND3
Implementation : \( y = ((a \cdot b \cdot c)')' \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width (( \mu m ))</th>
<th>( t_{\text{intrinsic}}(LH/HL) )</th>
<th>( Z_{\text{load}}(LH/HL) )</th>
<th>( C_{\text{pin}} )</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.3</td>
<td>0.073 / 0.090</td>
<td>4.3 / 2.2</td>
<td>0.0016</td>
<td>0.0047</td>
</tr>
<tr>
<td>2</td>
<td>2.3</td>
<td>0.061 / 0.086</td>
<td>1.9 / 1.2</td>
<td>0.0023</td>
<td>0.0075</td>
</tr>
<tr>
<td>4</td>
<td>3.2</td>
<td>0.065 / 0.067</td>
<td>0.97 / 0.56</td>
<td>0.0042</td>
<td>0.013</td>
</tr>
</tbody>
</table>

Three input OR gate: OR3
Implementation : \( y = ((a + b + c)')' \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width (( \mu m ))</th>
<th>( t_{\text{intrinsic}}(LH/HL) )</th>
<th>( Z_{\text{load}}(LH/HL) )</th>
<th>( C_{\text{pin}} )</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.75</td>
<td>0.05 / 0.14</td>
<td>4.3 / 2.5</td>
<td>0.0017</td>
<td>0.005</td>
</tr>
<tr>
<td>2</td>
<td>2.75</td>
<td>0.05 / 0.1</td>
<td>1.9 / 1.2</td>
<td>0.0028</td>
<td>0.008</td>
</tr>
<tr>
<td>4</td>
<td>3.68</td>
<td>0.045 / 0.1</td>
<td>1 / 0.6</td>
<td>0.0054</td>
<td>0.015</td>
</tr>
</tbody>
</table>
Four input AND gate: AND4
Implementation : \( y = ((a \cdot b \cdot c \cdot d)^{\prime})^{\prime} \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width((\mu m))</th>
<th>( t_{\text{intrinsic}} (LH/HL) )</th>
<th>( Z_{\text{load}} (LH/HL) )</th>
<th>( C_{\text{pin}} )</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.8</td>
<td>0.082 / 0.097</td>
<td>4.3 / 2.2</td>
<td>0.0017</td>
<td>0.0051</td>
</tr>
<tr>
<td>2</td>
<td>3.2</td>
<td>0.075 / 0.092</td>
<td>1.9 / 1.1</td>
<td>0.0027</td>
<td>0.009</td>
</tr>
<tr>
<td>4</td>
<td>4.6</td>
<td>0.073 / 0.082</td>
<td>0.97 / 0.56</td>
<td>0.0053</td>
<td>0.016</td>
</tr>
</tbody>
</table>

Four input OR gate: OR4
Implementation : \( y = ((a + b + c + d)^{\prime})^{\prime} \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width((\mu m))</th>
<th>( t_{\text{intrinsic}} (LH/HL) )</th>
<th>( Z_{\text{load}} (LH/HL) )</th>
<th>( C_{\text{pin}} )</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.8</td>
<td>0.06 / 0.17</td>
<td>4.3 / 2.8</td>
<td>0.002</td>
<td>0.0055</td>
</tr>
<tr>
<td>2</td>
<td>3.2</td>
<td>0.05 / 0.13</td>
<td>2 / 1.3</td>
<td>0.003</td>
<td>0.009</td>
</tr>
<tr>
<td>4</td>
<td>4.6</td>
<td>0.05 / 0.12</td>
<td>1 / 0.65</td>
<td>0.006</td>
<td>0.016</td>
</tr>
</tbody>
</table>

Four input AND-OR gate: ANDOR22
Implementation : \( y = ((a \cdot b + c \cdot d)^{\prime})^{\prime} \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width((\mu m))</th>
<th>( t_{\text{intrinsic}} (LH/HL) )</th>
<th>( Z_{\text{load}} (LH/HL) )</th>
<th>( C_{\text{pin}} )</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.8</td>
<td>0.07 / 0.14</td>
<td>4.3 / 2.4</td>
<td>0.0017</td>
<td>0.0056</td>
</tr>
<tr>
<td>2</td>
<td>3.2</td>
<td>0.065 / 0.13</td>
<td>1.9 / 1.3</td>
<td>0.0027</td>
<td>0.0094</td>
</tr>
<tr>
<td>4</td>
<td>4.2</td>
<td>0.06 / 0.1</td>
<td>1 / 0.65</td>
<td>0.0053</td>
<td>0.016</td>
</tr>
</tbody>
</table>

Invertor: NOT
Implementation : \( y = a^{\prime} \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width((\mu m))</th>
<th>( t_{\text{intrinsic}} (LH/HL) )</th>
<th>( Z_{\text{load}} (LH/HL) )</th>
<th>( C_{\text{pin}} )</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.92</td>
<td>0.015 / 0.014</td>
<td>2.9 / 2.7</td>
<td>0.0017</td>
<td>0.0026</td>
</tr>
<tr>
<td>2</td>
<td>1.38</td>
<td>0.014 / 0.0135</td>
<td>1.9 / 1.9</td>
<td>0.0027</td>
<td>0.0037</td>
</tr>
<tr>
<td>4</td>
<td>1.84</td>
<td>0.013 / 0.014</td>
<td>0.95 / 1</td>
<td>0.0053</td>
<td>0.0067</td>
</tr>
<tr>
<td>8</td>
<td>2.76</td>
<td>0.013 / 0.013</td>
<td>0.48 / 0.48</td>
<td>0.0053</td>
<td>0.0137</td>
</tr>
<tr>
<td>16</td>
<td>7.36</td>
<td>0.0125 / 0.013</td>
<td>0.12 / 0.13</td>
<td>0.0053</td>
<td>0.053</td>
</tr>
</tbody>
</table>

Two input NAND gate: NAND2
Implementation : \( y = (a \cdot b)^{\prime} \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width((\mu m))</th>
<th>( t_{\text{intrinsic}} (LH/HL) )</th>
<th>( Z_{\text{load}} (LH/HL) )</th>
<th>( C_{\text{pin}} )</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.38</td>
<td>0.027 / 0.016</td>
<td>4.2 / 2.4</td>
<td>0.003</td>
<td>0.0028</td>
</tr>
<tr>
<td>2</td>
<td>2.3</td>
<td>0.013 / 0.025</td>
<td>1.9 / 1.2</td>
<td>0.0054</td>
<td>0.0053</td>
</tr>
<tr>
<td>4</td>
<td>3.2</td>
<td>0.013 / 0.025</td>
<td>1 / 0.7</td>
<td>0.01</td>
<td>0.0095</td>
</tr>
</tbody>
</table>

Two input NOR gate: NOR2
Implementation : \( y = (a + b)^{\prime} \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width((\mu m))</th>
<th>( t_{\text{intrinsic}} (LH/HL) )</th>
<th>( Z_{\text{load}} (LH/HL) )</th>
<th>( C_{\text{pin}} )</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.38</td>
<td>0.026 / 0.014</td>
<td>6.1 / 2</td>
<td>0.003</td>
<td>0.003</td>
</tr>
<tr>
<td>2</td>
<td>2.3</td>
<td>0.025 / 0.0135</td>
<td>3 / 1</td>
<td>0.0054</td>
<td>0.0055</td>
</tr>
<tr>
<td>4</td>
<td>3.2</td>
<td>0.023 / 0.0134</td>
<td>1.5 / 0.5</td>
<td>0.01</td>
<td>0.01</td>
</tr>
</tbody>
</table>
Three input NAND gate: NAND3
Implementation: \( y = (a \cdot b \cdot c)' \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width((\mu m))</th>
<th>( t_{\text{intrinsic}}(\text{LH/HL}) )</th>
<th>( Z_{\text{load}}(\text{LH/HL}) )</th>
<th>( C_{\text{pin}} )</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.84</td>
<td>0.035 / 0.023</td>
<td>4.25 / 2.9</td>
<td>0.003</td>
<td>0.0048</td>
</tr>
<tr>
<td>2</td>
<td>3.22</td>
<td>0.03 / 0.021</td>
<td>2 / 1.5</td>
<td>0.0061</td>
<td>0.0086</td>
</tr>
<tr>
<td>4</td>
<td>4.6</td>
<td>0.031 / 0.022</td>
<td>1.1 / 0.85</td>
<td>0.0105</td>
<td>0.015</td>
</tr>
</tbody>
</table>

Three input NOR gate: NOR3
Implementation: \( y = (a + b + c)' \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width((\mu m))</th>
<th>( t_{\text{intrinsic}}(\text{LH/HL}) )</th>
<th>( Z_{\text{load}}(\text{LH/HL}) )</th>
<th>( C_{\text{pin}} )</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.84</td>
<td>0.056 / 0.022</td>
<td>7.7 / 2.3</td>
<td>0.0032</td>
<td>0.0058</td>
</tr>
<tr>
<td>2</td>
<td>2.76</td>
<td>0.046 / 0.021</td>
<td>3.8 / 1.2</td>
<td>0.0063</td>
<td>0.0095</td>
</tr>
<tr>
<td>4</td>
<td>2.52</td>
<td>0.042 / 0.021</td>
<td>1.7 / 0.6</td>
<td>0.0133</td>
<td>0.0195</td>
</tr>
</tbody>
</table>

Four input NAND gate: NAND4
Implementation: \( y = (a \cdot b \cdot c \cdot d)' \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width((\mu m))</th>
<th>( t_{\text{intrinsic}}(\text{LH/HL}) )</th>
<th>( Z_{\text{load}}(\text{LH/HL}) )</th>
<th>( C_{\text{pin}} )</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.3</td>
<td>0.04 / 0.03</td>
<td>4.26 / 3.4</td>
<td>0.0033</td>
<td>0.0055</td>
</tr>
<tr>
<td>2</td>
<td>3.7</td>
<td>0.035 / 0.025</td>
<td>2.2 / 2</td>
<td>0.0056</td>
<td>0.009</td>
</tr>
<tr>
<td>4</td>
<td>7.35</td>
<td>0.034 / 0.024</td>
<td>1 / 0.85</td>
<td>0.0127</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Four input NOR gate: NOR4
Implementation: \( y = (a + b + c + d)' \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width((\mu m))</th>
<th>( t_{\text{intrinsic}}(\text{LH/HL}) )</th>
<th>( Z_{\text{load}}(\text{LH/HL}) )</th>
<th>( C_{\text{pin}} )</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.3</td>
<td>0.06 / 0.022</td>
<td>8 / 2</td>
<td>0.0036</td>
<td>0.0061</td>
</tr>
<tr>
<td>2</td>
<td>4.14</td>
<td>0.083 / 0.023</td>
<td>4.5 / 1.1</td>
<td>0.007</td>
<td>0.012</td>
</tr>
<tr>
<td>4</td>
<td>7.82</td>
<td>0.061 / 0.023</td>
<td>2.4 / 0.6</td>
<td>0.013</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Two input multiplexer: MUX2
Implementation: \( y = s' \cdot a + s \cdot b \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width((\mu m))</th>
<th>( t_{\text{intrinsic}}(\text{LH/HL}) )</th>
<th>( Z_{\text{load}}(\text{LH/HL}) )</th>
<th>( C_{\text{pin}} )</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.7</td>
<td>0.09 / 0.075</td>
<td>4.25 / 2.3</td>
<td>0.0038</td>
<td>0.0072</td>
</tr>
<tr>
<td>2</td>
<td>4.15</td>
<td>0.085 / 0.07</td>
<td>1.9 / 1.2</td>
<td>0.006</td>
<td>0.012</td>
</tr>
<tr>
<td>4</td>
<td>5.5</td>
<td>0.09 / 0.075</td>
<td>1.1 / 0.6</td>
<td>0.0072</td>
<td>0.018</td>
</tr>
</tbody>
</table>

Two input inverting multiplexer: NMUX2
Implementation: \( y = s' \cdot a + s \cdot b \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width((\mu m))</th>
<th>( t_{\text{intrinsic}}(\text{LH/HL}) )</th>
<th>( Z_{\text{load}}(\text{LH/HL}) )</th>
<th>( C_{\text{pin}} )</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.22</td>
<td>0.045 / 0.06</td>
<td>5.1 / 2.7</td>
<td>0.0043</td>
<td>0.006</td>
</tr>
<tr>
<td>2</td>
<td>4.6</td>
<td>0.04 / 0.06</td>
<td>2.6 / 1.5</td>
<td>0.0073</td>
<td>0.0115</td>
</tr>
<tr>
<td>4</td>
<td>6.9</td>
<td>0.035 / 0.05</td>
<td>1.2 / 0.7</td>
<td>0.0124</td>
<td>0.02</td>
</tr>
</tbody>
</table>
APPENDIX C. STANDARD CELL LIBRARIES

Two input XOR: XOR2
Implementation : \( y = a \cdot b' + d' \cdot b \)

<table>
<thead>
<tr>
<th>Strength</th>
<th>Width((\mu m))</th>
<th>(t_{\text{intrinsic}}(LH/HL))</th>
<th>(Z_{\text{load}}(LH/HL))</th>
<th>(C_{\text{pin}})</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.22</td>
<td>0.035 / 0.055</td>
<td>4 / 2.2</td>
<td>0.003</td>
<td>0.007</td>
</tr>
<tr>
<td>2</td>
<td>5.5</td>
<td>0.035 / 0.05</td>
<td>2 / 1.1</td>
<td>0.005</td>
<td>0.012</td>
</tr>
<tr>
<td>4</td>
<td>8.3</td>
<td>0.03 / 0.04</td>
<td>1 / 0.6</td>
<td>0.01</td>
<td>0.02</td>
</tr>
</tbody>
</table>

C.2.2 # Flip-flops
Appendix D

Memory compilers
Appendix E

FPGA
Appendix F

How to make a project

F.1 How to organize a project

Project specification

Project architecture

Behavioral description

Simulation & test

Synthesis

Testing the design

Evaluation

F.2 An example: FIFO memory

F.3 About team work

There are few principles governing the team work:

1. Each member of the team must know all the aspects of the project but must work only on a specific part of the project

2. Even if the final contribution of each member will be unbalanced, the effort of each member must be maximized

3. A project is a creative work and creation has multiple facets to be covered by the diversity of skills and talents of the members of the team

4. A leader is good but not mandatory for a small team

5. A good chemistry between the members of the team is a good starting point

6. The professional value of the team is not the “sum” of the professional values of its components (a team of “geniuses” sometimes will be unable to finalize a project)
7. the team must work having in mind that the current project is not the last the team faces.

The previous list is open.
Appendix G

Designing a simple CISC processor

G.1 The project

cisc_processor.v
cisc_alu.v
control_automaton.v
mux2.v
mux4.v
mux2.v
register_file.v

G.2 RTL code

The module cisc_processor.v

module cisc_processor(input clock ,
        input reset ,
        output reg [31:0] addr_reg, // memory address
        output reg [1:0] com_reg , // memory command
        output reg [31:0] out_reg , // data output
        input [31:0] in ); // data/inst input

// INTERNAL CONNECTIONS
wire [25:0] command;
wire flag;
wire [31:0] alu_out, left, right, left_out, right_out;

// INPUT & OUTPUT BUFFER REGISTERS
reg [31:0] data_reg, inst_reg;
always @(posedge clock) begin if (command[25]) inst_reg <= in;
data_reg <= in;
addr_reg <= left_out ;
out_reg  <= right_out ;
com_reg  <= command[1:0];
end

// CONTROL_AUTOMATON
control_automaton control_automaton(.clock (clock ),
        .reset (reset ),
        .inst  (inst_reg[31:11]));
APPENDIX G. DESIGNING A SIMPLE CISC PROCESSOR

```
module cisc_alu output reg [31:0] alu_out ,
    input [31:0] left ,
    input [31:0] right ,
    input [3:0] alu_com );

wire [32:0] add, sub;

assign add = left + right,
    sub = left - right;

always @(alu_com or left or right or add or sub)
    case(alu_com)
    4'b0000: alu_out = left ;
    4'b0001: alu_out = right ;
    4'b0010: alu_out = left + 1 ;
    4'b0011: alu_out = left - 1 ;
    4'b0100: alu_out = add[31:0] ;
    4'b0101: alu_out = sub[31:0] ;
    4'b0110: alu_out = {1'b0, left[31:1]} ;
    4'b0111: alu_out = {left[31], left[31:1]} ;
    4'b1000: alu_out = {31'b0, (left == 0)} ;
    4'b1001: alu_out = {31'b0, (left == right)} ;
endcase
```
The module control_automaton.v

module control_automaton( input clock ,
                   input reset ,
                   input [31:11] inst ,
                   output [25:0] command ,
                   input flag );

// THE STRUCTURE OF 'inst'
wire [5:0] opcode ; // operation code
wire [4:0] dest , // selects destination register
          left_op , // selects left operand register
          right_op; // selects right operand register
assign {opcode, dest, left_op, right_op} = inst;

// THE STRUCTURE OF 'command'
reg en_inst ; // enable load a new instruction in inst_reg
reg write_enable; // writes the output of alu at dest_addr
reg [4:0] dest_addr ; // selects the destination register
reg [4:0] left_addr ; // selects the left operand in file register
reg [3:0] alu_com ; // selects the operation performed by the alu
reg [4:0] right_addr ; // selects the right operand in file register
reg left_sel ; // selects the source of the left operand
reg [1:0] right_sel ; // selects the source of the right operand
reg [1:0] mem_com ; // generates the command for memory
assign command = {en_inst, write_enable, dest_addr, left_addr,
                  alu_com, right_addr, left_sel, right_sel, mem_com};

// MICRO-ARCHITECTURE
// en_inst
parameter
no_load   = 1'b0, // disable instruction register
load_inst = 1'b1; // enable instruction register
// write_enable
parameter
no_write   = 1'b0,
write_back = 1'b1; // write back the current ALU output
// alu_func
parameter
alu_left   = 4'b0000, // alu_out = left
alu_right  = 4'b0001, // alu_out = right
alu_inc    = 4'b0010, // alu_out = left + 1
alu_dec    = 4'b0011, // alu_out = left - 1
APPENDIX G. DESIGNING A SIMPLE CISC PROCESSOR

alu_add = 4'b0100, // alu_out = left + right
alu_sub = 4'b0101, // alu_out = left - right
alu_shl = 4'b0110, // alu_out = {1'b0, left[31:1]}
alu_half = 4'b0111, // alu_out = {left[31], left[31:1]}
alu_zero = 4'b1000, // alu_out = {31'b0, (left == 0)}
alu_equal = 4'b1001, // alu_out = {31'b0, (left == right)}
alu_less = 4'b1010, // alu_out = {31'b0, (left < right)}
alu_carry = 4'b1011, // alu_out = {31'b0, add[32]}
alu_borrow = 4'b1100, // alu_out = {31'b0, sub[32]}
alu_and = 4'b1101, // alu_out = left & right
alu_or = 4'b1110, // alu_out = left | right
alu_xor = 4'b1111; // alu_out = left ^ right

// left_sel
parameter
left_out = 1'b0, // left out of the reg file as left op
from_mem = 1'b1; // data from memory as left op

// right_sel
parameter
right_out = 2'b00, // right out of the reg file as right op
jmp_addr = 2'b01, // right_op = {22{inst[10]}}, inst[10:0]
low_value = 2'b10, // right_op = {16{inst[15]}}, inst[15:0]
high_value = 2'b11; // right_op = {inst[15:0], 16'b0}

// mem_com
parameter
mem_nop = 2'b00,
mem_read = 2'b10, // read from memory
mem_write = 2'b11; // write to memory

// INSTRUCTION SET ARCHITECTURE (only samples)
// arithmetic & logic instructions & pc = pc + 1
parameter
move = 6'b10_0000, // dest_reg = left_out
inc = 6'b10_0010, // dest_reg = left_out + 1
dec = 6'b10_0011, // dest_reg = left_out - 1
add = 6'b10_0100, // dest_reg = left_out + right_out
sub = 6'b10_0101, // dest_reg = left_out - right_out
bwxor = 6'b10_1111; // dest_reg = left_out ^ right_out

// data move instructions & pc = pc + 1
parameter
read = 6'b01_0000, // dest_reg = mem(left_out)
rdinc = 6'b01_0001, // dest_reg = mem(left_out + value)
write = 6'b01_1000, // mem(left_out) = right_out
wrinc = 6'b01_1001; // mem(left_out + value) = right_out

// control instructions
parameter
nop = 6'b11_0000, // pc = pc + 1
jmp = 6'b11_0001, // pc = pc + value
call = 6'b11_0010, // pc = value, ra = pc + 1
ret = 6'b11_0011, // pc = ra
jzero = 6'b11_0100, // if (left_out = 0) pc = pc + value;
// else pc = pc + 1
jnzero = 6'b11_0101; // if (left_out != 0) pc = pc + value;
// else pc = pc + 1

// THE STATE REGISTER
reg [5:0] state_reg ; // the state register
reg [5:0] next_state ; // a "register" used as variable
always @(posedge clock) if (reset) state_reg <= 0 ;
else state_reg <= next_state ;

// THE CONTROL AUTOMATON'S LOOP
always @(state_reg or opcode or dest or left_op or right_op or flag)
begin en_inst = 1'bx;//no_load ;
write_enable = 1'bx;//no_write ;
dest_addr = 5'bxxxxx ;
left_addr = 5'bxxxxx ;
alu_com = 4'bxxxx ;
right_addr = 5'bxxxxx ;
left_sel = 1'bx ;
right_sel = 2'bxx ;
mem_com = 2'bxx;//mem_nop ;
next_state = 6'bxxxxxx;//state_reg + 1;
end

// INITIALIZE THE PROCESSOR
if (state_reg == 6'b00_0000)
// pc = 0
begin
  en_inst = no_load ;
  write_enable = write_back ;
dest_addr = 5'b11111 ;
left_addr = 5'b11111 ;
alu_com = alu_xor ;
right_addr = 5'b11111 ;
left_sel = left_out ;
right_sel = right_out ;
mem_com = mem_nop ;
next_state = state_reg + 1;
end

// INSTRUCTION FETCH
if (state_reg == 6'b00_0001)
// request for a new instruction & increment pc
begin
  en_inst = no_load ;
  write_enable = write_back ;
dest_addr = 5'b11111 ;
left_addr = 5'b11111 ;
alu_com = alu_inc ;
right_addr = 5'bxxxxx ;
left_sel = left_out ;
right_sel = 2'bxx ;
mem_com = mem_read ;
next_state = state_reg + 1;
end
APPENDIX G. DESIGNING A SIMPLE CISC PROCESSOR

end

if (state_reg == 6'b00_0010)
// wait for memory to read doing nothing
begin
  en_inst = no_load;
  write_enable = no_write;
  dest_addr = 5'bxxxxx;
  left_addr = 5'bxxxxx;
  alu_com = 4'bxxxx;
  right_addr = 5'bxxxxx;
  left_sel = 1'bx;
  right_sel = 2'bxx;
  mem_com = mem_nop;
  next_state = state_reg + 1;
end

if (state_reg == 6'b00_0011)
// load the new instruction in instr_reg
begin
  en_inst = load_inst;
  write_enable = no_write;
  dest_addr = 5'bxxxxx;
  left_addr = 5'bxxxxx;
  alu_com = 4'bxxxx;
  right_addr = 5'bxxxxx;
  left_sel = 1'bx;
  right_sel = 2'bxx;
  mem_com = mem_nop;
  next_state = state_reg + 1;
end

if (state_reg == 6'b00_0100)
// initialize the control automaton
begin
  en_inst = load_inst;
  write_enable = write_back;
  dest_addr = 5'bxxxxx;
  left_addr = 5'bxxxxx;
  alu_com = 4'bxxxx;
  right_addr = 5'bxxxxx;
  left_sel = 1'bx;
  right_sel = 2'bxx;
  mem_com = mem_nop;
  next_state = opcode[5:0];
end

// EXECUTE THE ONE CYCLE FUNCTIONAL INSTRUCTIONS
if (state_reg[5:4] == 2'b10)
// dest = left_op OPERATION right_op
begin
  en_inst = no_load;
  write_enable = write_back;
  dest_addr = dest;
  left_addr = left_op;
  alu_com = opcode[3:0];
  right_addr = right_op;
end
right_addr = right_op ;
left_sel = left_out ;
right_sel = right_out ;
mem_com = mem_nop ;
next_state = 6'b00_0001 ;
end

// EXECUTE MEMORY READ INSTRUCTIONS
if (state_reg == 6'b01_0000)
// read from left_reg in dest_reg
begin
  en_inst = no_load ;
  write_enable = no_write ;
  dest_addr = 5'bxxxxx ;
  left_addr = left_op ;
  alu_com = alu_left ;
  right_addr = 5'bxxxxx ;
  left_sel = left_out ;
  right_sel = 2'bxx ;
  mem_com = mem_read ;
  next_state = 6'b01_0010 ;
end
if (state_reg == 6'b01_0001)
// read from left_reg + <value> in dest_reg
begin
  en_inst = no_load ;
  write_enable = no_write ;
  dest_addr = 5'bxxxxx ;
  left_addr = left_op ;
  alu_com = alu_add ;
  right_addr = 5'bxxxxx ;
  left_sel = left_out ;
  right_sel = low_value ;
  mem_com = mem_read ;
  next_state = 6'b01_0010 ;
end
if (state_reg == 6'b01_0010)
// wait for memory to read doing nothing
begin
  en_inst = no_load ;
  write_enable = no_write ;
  dest_addr = 5'bxxxxx ;
  left_addr = 5'bxxxxx ;
  alu_com = 4'bxxxx ;
  right_addr = 5'bxxxxx ;
  left_sel = 1'bx ;
  right_sel = 2'bxx ;
  mem_com = mem_nop ;
  next_state = state_reg + 1;
end
if (state_reg == 6'b01_0011)
// the data from memory is loaded in data_reg
begin
APPENDIX G. DESIGNING A SIMPLE CISC PROCESSOR

en.inst = no_load;
write_enable = no_write;
dest_addr = 5'bxxxxx;
left_addr = 5'bxxxxx;
alu_com = 4'bxxxx;
right_addr = 5'bxxxxx;
left_sel = 1'bx;
right_sel = 2'bxx;
mem_com = mem_nop;
next_state = state_reg + 1;
end
if (state_reg == 6'b01_0100)
  // data_reg is loaded in dest_reg & go to fetch
  begin
    en.inst = no_load;
    write_enable = write_back;
dest_addr = dest;
    left_addr = 5'bxxxxx;
    alu_com = alu_left;
    right_addr = 5'bxxxxx;
    left_sel = from_mem;
    right_sel = 2'bxx;
    mem_com = mem_nop;
    next_state = 6'b00_0001;
  end
// EXECUTE MEMORY WRITE INSTRUCTIONS
if (state_reg == 6'b01_1000)
  // write right_op to left_op & go to fetch
  begin
    en.inst = no_load;
    write_enable = no_write;
dest_addr = 5'bxxxxx;
    left_addr = left_op;
    alu_com = alu_left;
    right_addr = right_op;
    left_sel = left_out;
    right_sel = 2'bxx;
    mem_com = mem_write;
    next_state = 6'b00_0001;
  end
if (state_reg == 6'b01_1000)
  // write right_op to left_op + <value> & go to fetch
  begin
    en.inst = no_load;
    write_enable = no_write;
dest_addr = 5'bxxxxx;
    left_addr = left_op;
    alu_com = alu_add;
    right_addr = right_op;
    left_sel = left_out;
    right_sel = low_value;
    mem_com = mem_write;
  end
G.2. RTL CODE

```verilog
next_state = 6'b00_0001;
end

// CONTROL INSTRUCTIONS
if (state_reg == 6'b11_0000)
  // no operation & go to fetch
  begin
    en_inst = no_load;
    write_enable = no_write;
    dest_addr = 5'bxxxxxx;
    left_addr = 5'bxxxxxx;
    alu_com = 4'bxxxx;
    right_addr = 5'bxxxxxx;
    left_sel = 1'bx;
    right_sel = 2'bxx;
    mem_com = mem_nop;
    next_state = 6'b00_0001;
  end
if (state_reg == 6'b11_0001)
  // jump to (pc + <value>) & go to fetch
  begin
    en_inst = no_load;
    write_enable = write_back;
    dest_addr = 5'b11111;
    left_addr = 5'b11111;
    alu_com = alu_add;
    right_addr = 5'bxxxxxx;
    left_sel = left_out;
    right_sel = low_value;
    mem_com = mem_nop;
    next_state = 6'b00_0001;
  end
if (state_reg == 8'b0011_0110)
  // call: first step: ra = pc + 1
  begin
    en_inst = no_load;
    write_enable = write_back;
    dest_addr = 5'b11111;
    left_addr = 5'b11111;
    alu_com = alu_left;
    right_addr = 5'bxxxxxx;
    left_sel = left_out;
    right_sel = 2'bxx;
    mem_com = mem_nop;
    next_state = 6'b11_0110;
  end
if (state_reg == 6'b11_0010)
  // call: second step: pc = value
  begin
    en_inst = no_load;
    write_enable = write_back;
    dest_addr = 5'b11111;
    left_addr = 5'bxxxxxx;
  end
```

511
```vhdl
alu_com = alu_right;
right_addr = 5'bxxxxx;
left_sel = 1'bx;
right_sel = jmp_addr;
mem_com = mem_nop;
next_state = 6'b00_0001;
end
if (state_reg == 6'b11_0011)
  // ret: pc = ra
  begin
    en_inst = no_load;
    write_enable = write_back;
    dest_addr = 5'b11111;
    left_addr = 5'b11110;
    alu_com = alu_left;
    right_addr = 5'bxxxxx;
    left_sel = left_out;
    right_sel = 2'bxx;
    mem_com = mem_nop;
    next_state = 6'b00_0001;
  end
if ((state_reg == 6'b11_0100) && flag)
  // jzero: if (left_out = 0) pc = pc + value;
  begin
    en_inst = no_load;
    write_enable = write_back;
    dest_addr = 5'b11111;
    left_addr = 5'b11111;
    alu_com = alu_add;
    right_addr = 5'bxxxxx;
    left Sel = left_out;
    right_sel = low_value;
    mem_com = mem_nop;
    next_state = 6'b00_0001;
  end
if ((state_reg == 6'b11_0100) && ~flag)
  // jnzero: if (left_out = 1) pc = pc + value;
  begin
    en_inst = no_load;
    write_enable = no_write;
    dest_addr = 5'bxxxxx;
    left_addr = 5'bxxxxx;
    alu_com = 4'bxxxx;
    right_addr = 5'bxxxxx;
    left_sel = 1'bx;
    right_sel = 2'bxx;
    mem_com = mem_nop;
    next_state = 6'b00_0001;
  end
if ((state_reg == 6'b11_0100) && ~flag)
  // jnzero: if (left_out = 1) pc = pc + value;
  begin
```
### G.2. RTL CODE

```verilog
en_inst = no_load;
write_enable = write_back;
dest_addr = 5'b11111;
left_addr = 5'b11111;
alu_com = alu_add;
right_addr = 5'bxxxxx;
left_sel = left_out;
right_sel = low_value;
mem_com = mem_nop;
next_state = 6'b00_0001;
end

if ((state_reg == 6'b11_0100) && flag)
// jnzero: if (left_out = 0) pc = pc + 1;
begin
en_inst = no_load;
write_enable = no_write;
dest_addr = 5'bxxxxx;
left_addr = 5'bxxxxx;
alu_com = 4'bxxxx;
right_addr = 5'bxxxxx;
left_sel = 1'bx;
right_sel = 2'bxx;
mem_com = mem_nop;
next_state = 6'b00_0001;
end
end
endmodule

The module register_file.v

```verilog
module register_file( output [31:0] left_out ,
output [31:0] right_out ,
input [31:0] result ,
input [4:0] left_addr ,
input [4:0] right_addr ,
input [4:0] dest_addr ,
input write_enable,
input clock );

reg [31:0] file[0:31];

assign left_out = file[left_addr] ,
right_out = file[right_addr] ;

always @(posedge clock) if (write_enable) file[dest_addr] <= result;
endmodule

The module mux4.v

```verilog
module mux4(output [31:0] out,
input [31:0] in0,
input [31:0] in1,
input [31:0] in2,
input [31:0] in3,
input [1:0] sel);
wire[31:0] out1, out0;
mux2 mux(out, out0, out1, sel[1]);
mux2 mux1(out1, in2, in3, sel[0]),
mux0(out0, in0, in1, sel[0]);
endmodule

The module mux2.v
module mux2(output [31:0] out,
            input [31:0] in0,
            input [31:0] in1,
            input sel);
    assign out = sel ? in1 : in0;
endmodule

G.3 Testing cisc_processor.v
Appendix H

# Meta-stability

Any asynchronous signal applied the the input of a clocked circuit is a source of meta-stability [webRef_1] [Alfke ’05] [webRef_4]. There is a dangerous timing window “centered” on the clock transition edge specified by the sum of set-up time, edge transition time and hold time. If the data input of a D-FF switches in this window, then there are three possible behaviors for its output:

- the output does not change according to the change on the flip-flop’s input (the flip-flop does not catch the input variation)
- the output change according to the change on the flip-flop’s input (the flip-flop catches the input variation)
- the output goes meta-stable for $t_{MS}$, then goes unpredictable in 1 or 0 (see the wave forms [webRef_2]).

Figure H.1: Metastability [webRef_4].
APPENDIX H. # META-STABILITY
Bibliography


http://arch.pub.ro/gstefan/CAS98.pdf


[webRef_1] http://www.fpga-faq.com/FAQ_Pages/0017_Tell_me_about_metastables.htm
[webRef_3] http://www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html#fsa_pfx
[webRef_4] https://techdocs.altium.com/display/FPGA/Reducing+Metastability+in+FPGA+Designs


Index

D flip-flop: delay flip-flop, is the master-slave structure used for the embodiment of registers, 48
D-latch: is the elementary clocked latch with R=S=D, 43

fan-out: the number on input gates connected to the output of a digital circuit, 461
flip-flop: the edge clocked basic storage element, 46

master-slave principle: the mechanism used to remove the transparency of a storage element, 46