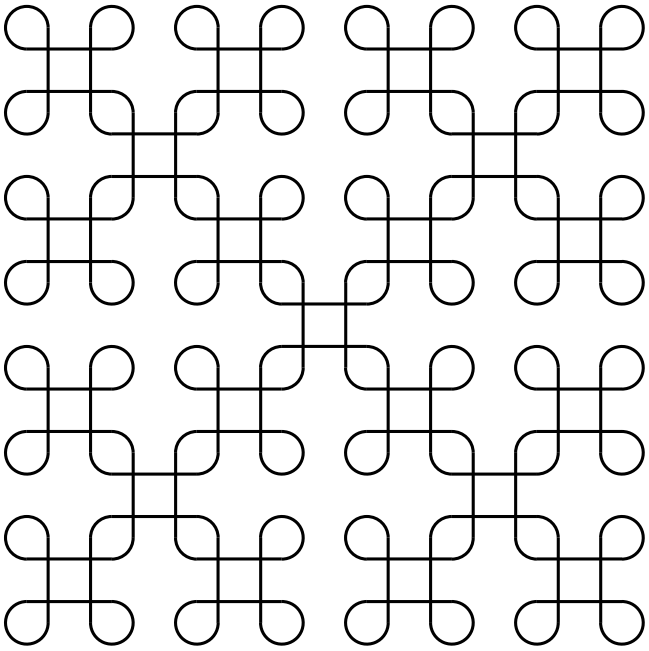
Loops & Complexity
in
DIGITAL SYSTEMS

*

Lecture Notes on Digital Design
in
Ten Giga-Gate/Chip Era

(work in endless progress)

Gheorghe M. Ştefan



– 2021 version –

This document was prepared with LaTeX 2_ε

Introduction

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

Paul Feyerabend[1]

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

Few legitimate questions about how to teach digital systems in *Ten Giga-Gate Per Chip Era* 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 *Ten Giga-Gate Per Chip Era*. 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 *Ten Giga-Gate Per Chip Era*.

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

[1]Paul 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.

processes? The dynamic of *adding circuits* and of adding *adding features* seems to be very different and governed by distinct mechanisms.

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 o 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 book ends with two annexes containing short reviews of the prerequisite knowledge.

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: ***Digital Circuits*** is an introductory text in the field of digital circuits seen from the conventional perspective of combinational and sequential circuits. In this first step in approaching the digital domain we 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: ***Growing & 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: ***Computing Machines*** 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 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

Contents (detailed)

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 does'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[1]

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 us take first examples with 1-bit signals.

Example 1.1 *A disciplined driver starts the car's engine only if all four doors are closed and, in all occupied seats, the seat belts are connected. The key contact and the previous condition are the ones that start the engine. (This example is from [1].)*

The car is equipped with sensors for each door (`d1, d2, d3, d4`*), for each seat (*`s1, s2, s3, s4`*), for each belt (*`b1, b2, b3, b4`*) and for the ignition key (*`k`*). The logic function that generates the start bit (s) is as follows:*

[1] They co-authored *iCon. Steve Jobs. The Greatest Second Act in the History of Business*, an unauthorized portrait of the co-founder of *Apple*.

```
s = (doors_are_closed) AND (each_occupied_with_belt_on) AND (key_is_on)
s = (d1 AND d2 AND d3 AND d4) AND
    ((b1 OR (NOT b1) AND (NOT s1)) AND
     (b2 OR (NOT b2) AND (NOT s2)) AND
     (b3 OR (NOT b3) AND (NOT s3)) AND
     (b4 OR (NOT b4) AND (NOT s4))) AND
    k)
```

In algebraic notation:

$$s = (d1 \cdot d2 \cdot d3 \cdot d4) \cdot ((b1 + b1' \cdot s1') \cdot (b2 + b2' \cdot s2') \cdot (b3 + b3' \cdot s3') \cdot (b4 + b4' \cdot s4')) \cdot k$$

Because the operator AND, "·", is usually omitted:

$$s = d1\ d2\ d3\ d4\ (b1 + b1'\ s1')(b2 + b2'\ s2')(b3 + b3'\ s3')(b4 + b4'\ s4')k$$

The expression ca be simplified because: $a + a'b = a + b$ (half-absorbtion rule).

Indeed, the car can start if each place has the belt on or is not occupied. Results the simplified form:

$$s = d1\ d2\ d3\ d4\ (b1 + s1')(b2 + s2')(b3 + s3')(b4 + s4')k$$

The Verilog description is:

```
module ignitionKey( output  s,
                    input   d1, d2, d3, d4, s1, s2, s3, s4,
                            b1, b2, b3, b4, k);
    assign s = d1 & d2 & d3 & d4 &  (b1 | ~s1) &
                                    (b2 | ~s2) &
                                    (b3 | ~s3) &
                                    (b4 | ~s4) & k  ;
endmodule
```

The result provided by the Vivado tool is represented in Figure 1.1.

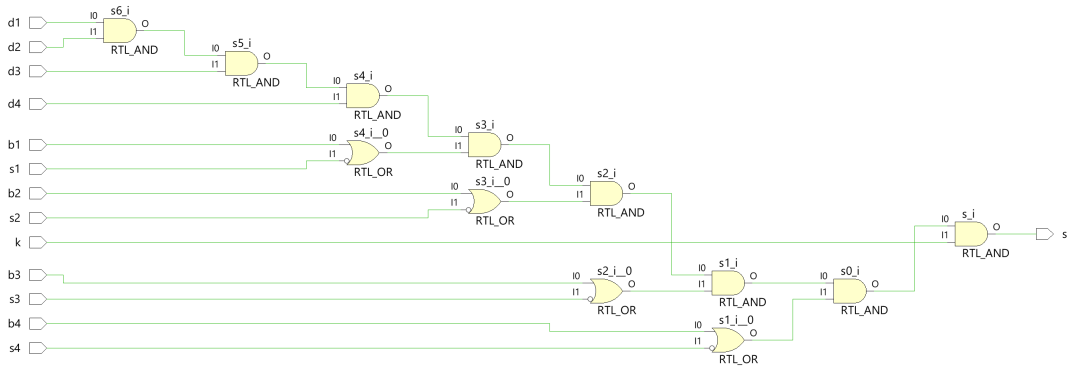


Figure 1.1: Ignition Key circuit.

◇

Example 1.2 *Be a store where customer access is restricted to a maximum of N people. The access is directed by a traffic light with two colors: green, allows access, and red, prohibits access. The access door is equipped with two sensors: one signals, by a pulse, the entry of a client and another the exit of a client by another pulse. When the number of customers in the store is greater than N, the traffic light is red, otherwise it is green. The store has only one door, so the two pulses that indicate the change in the number of customers cannot appear simultaneously.*

The Verilog description is:

```
module customerLimit(
        output          light    , // 0 means green; 1 means red
        input           inPulse  , // customer enter the store
        input           outPulse , // customer leave the store
        input   [3:0]   limit    );// customers accepted

    reg [10:0]  inCustRegister;      // records the number of entrants
    reg [10:0]  outCustRegister;     // records the number of left people

    initial begin inCustRegister    = 11'b0 ; // not recomended
                  outCustRegister   = 11'b0 ; // not recomended
          end

    always @(negedge inPulse)
        inCustRegister  <= inCustRegister + 1    ;

    always @(negedge outPulse)
        outCustRegister <= outCustRegister + 1   ;

    assign light = (inCustRegister - outCustRegister) > limit    ;

endmodule
```

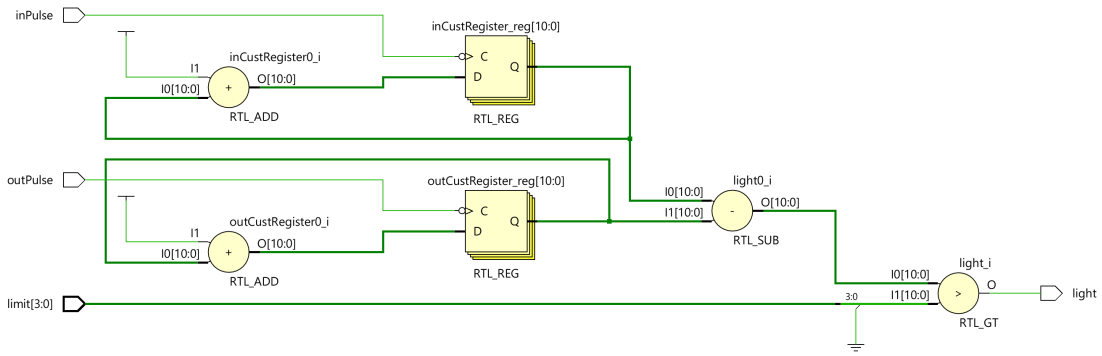


Figure 1.2: Customer Limit circuit.

◇

Let us take now an example with mode than 1 bit input signals.

Example 1.3 *Let be the analogue, continuously variable, signal in Figure 1.3. 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:*

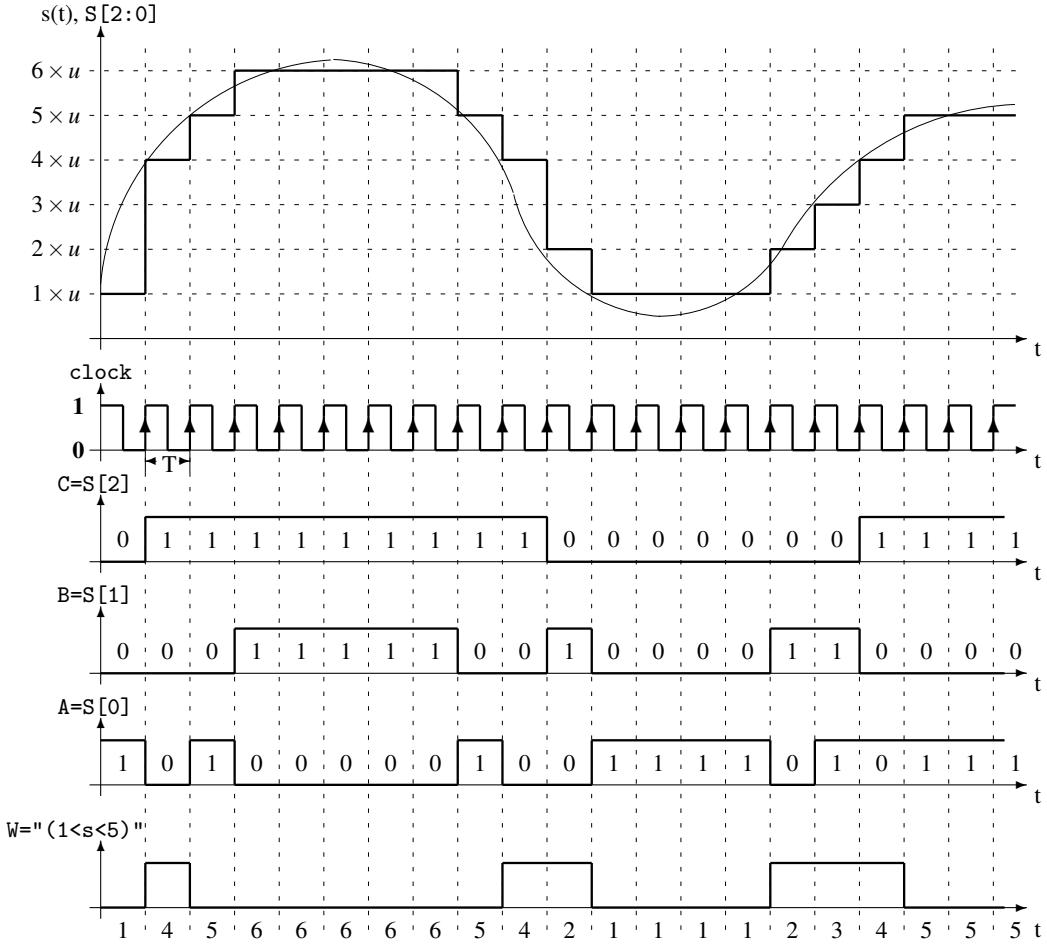


Figure 1.3: **Analogue to digital conversion.** The analog signal, $s(t)$, is sampled at each T using the unit measure u, and results the three-bit digital signal `S[2:0]`. **A first application**: the one-bit digital signal `W="(1<s<5)"` indicates, by its active value 1, the time interval when the digital signal is strictly included between $1u$ and $5u$. The three-bit result of conversion is `S[2:0]`.

$$
\begin{aligned}
s(0\times T) &= 1\,units \Rightarrow 001,\\
s(1\times T) &= 4\,units \Rightarrow 100,\\
s(2\times T) &= 5\,units \Rightarrow 101,\\
s(3\times T) &= 6\,units \Rightarrow 110,\\
s(4\times T) &= 6\,units \Rightarrow 110,\\
s(5\times T) &= 6\,units \Rightarrow 110,\\
s(6\times T) &= 6\,units \Rightarrow 110,
\end{aligned}
$$

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

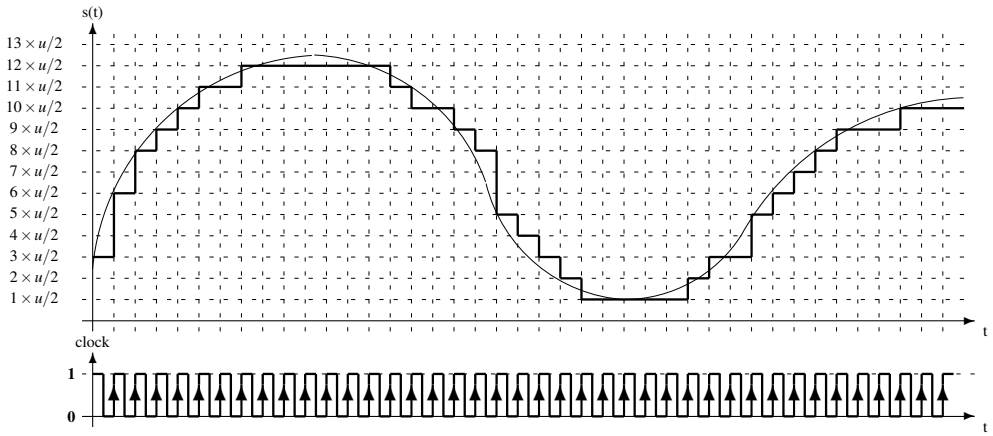


Figure 1.4: **More accurate analogue to digital.** The analogous signal is sampled at each $T/2$ using the unit measure $u/2$.

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 1.4 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, 1011, 1100, 1100, 1100, 1100, 1100, 1100,
1100, 1011, 1010, 1010, 1001, 1000, 0101, 0100, 0011, 0010, 0001, 0001, 0001,
0001, 0001, 0010, 0011, 0011, 0101, 0110, 0111, 1000, 1001, 1001, 1001, 1010,
1010, 1010, ...>
```

◇

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.

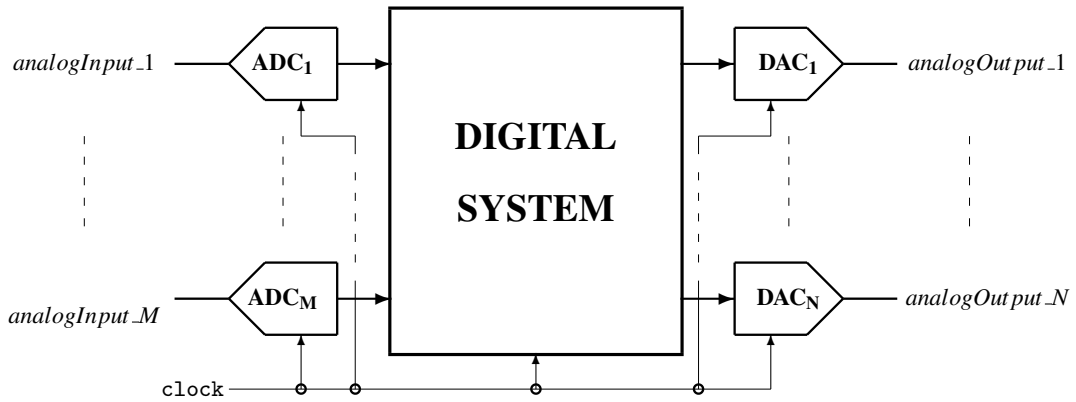


Figure 1.5: **Generic digital electronic system.**

The generic digital electronic system is represented in Figure 1.5, where:

- *analogInput_i*, for $i = 1, \ldots M$, provided by various sensors (microphones, ...), are sent to the input of M ADCs
- ADC_i converts *analogInput_i* 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 **D**igital-to-**A**nalog **C**onverters (DAC)
- 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 1.5 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 1.4 *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 1.3). The one-bit output,* `w`*, of the Boolean (logic) circuit*[2] *to be designed, let's call it* `window`*, must be active (on 1) each time when the result of conversion is less than 5 and greater than 1. In Figure 1.3 the wave form represents the signal* `w` *for the particular signal represented in the first wave form. The transfer function of the circuit is represented in the table from Figure 1.6a, where: for three binary input configurations,* `S[2:0] = {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 when  ((not C = 1) and (B = 1)     and (not A = 1)) or
            ((not C = 1) and (B = 1)     and (A = 1))     or
            ((C = 1)     and (not B = 1) and (not A = 1))
```

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 1.6b, 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'`

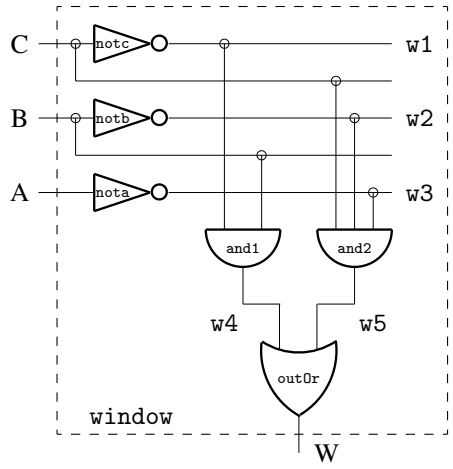
- *one 2-input OP circuit computes the final OR between the previous two functions.*

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

[2] See details about Boolean logic in the appendix **Boolan Functions**.

C	B	A	W
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

a.



b.

Figure 1.6: **The circuit** `window`**. a.** The truth table represents the behavior of the output for **all** binary configurations on the input. **b.** The circuit implementation.

```
/*********************************************************************
File name:      window.v
Circuit name:   Window
Description:    the circuit detect the input in the range of (1,5)
*********************************************************************/
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'
         nota(w3, A);           // the instance 'nota' of the generic 'not'
    and  and1(w4, w1, B),       // the instance 'and1' of the generic 'and'
         and2(w5, C, w2, w3);   // the instance 'and2' of the generic 'and'
    or   outOr(W, w4, w5);      // the instance 'outOr' of the generic 'or'

endmodule
```

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*

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`*:*

```
/*************************************************************************
File name:      simWindow.v
Circuit name:   Simulation module for simWindow.v
Description:    generate stimulus for the module simWindow.v
*************************************************************************/
    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 1.5 *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.

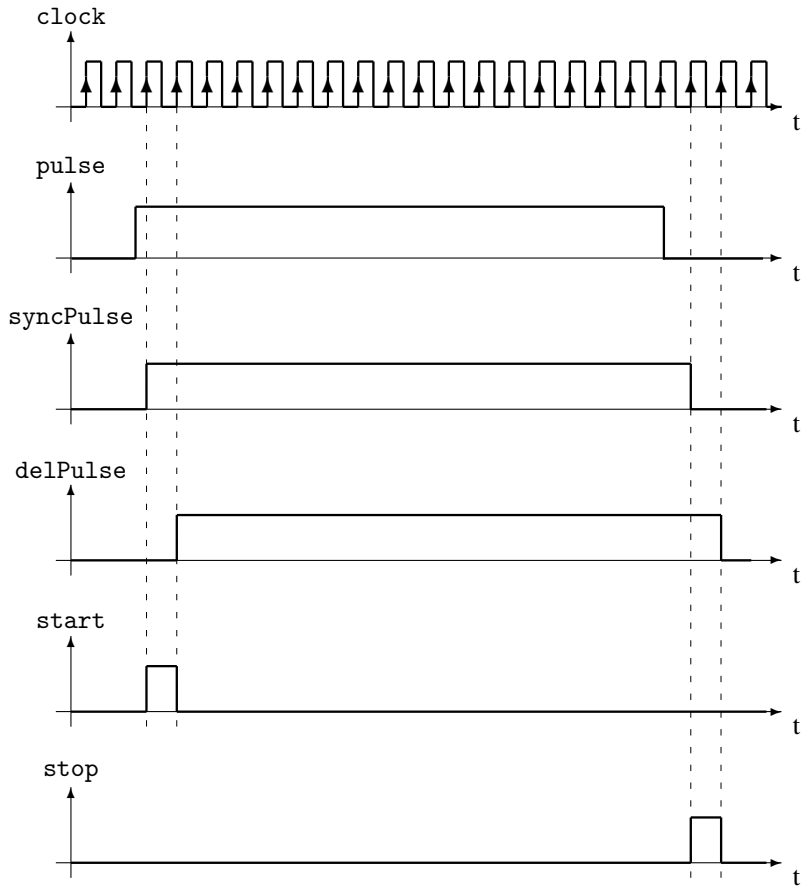


Figure 1.7: **The wave forms defining the `start/stop` circuit.** The `pulse` signal is asynchronously provided by a sensor. The signal `syncPulse` captures synchronously the signal to be processed. The signal `delPulse` is `syncPulse` delayed one clock cycle using a second one-bit register.

The 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* **aproximated** *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*

     `start = syncPulse · depPulse'`
   - *only in the first clock cycle after the end of* `syncPulse` *the signal* `delPulse` *is 1; then*

stop = syncPulse' · depPulse

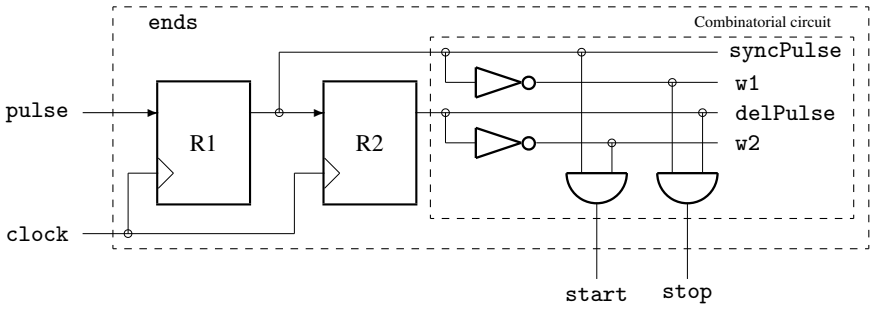


Figure 1.8: **The `ends` circuit.** The one-bit register R1 synchronises the raw signal `pulse`. The one-bit register R2 delays the synchronized signal to provide the possibility to emphasize the two ends of the synchronized pulse. The combinatorial circuit detects the two ends of the `pulse` signal approximated by the `syncPulse` signal.

The circuit (see Figure 1.8) 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:*

```
/*****************************************************************
File name:      ends.v
Circuit name:   Detector of ends
Description:    used to measure the length of a pulse
*****************************************************************/
    module ends(output  start    ,
                output  stop     ,
                input   pulse    ,
                input   clock    );

        reg syncPulse   ;
        reg delPulse    ;
        wire    w1, w2  ;

        always @(posedge clock) begin   syncPulse   <= pulse    ;
                                        delPulse    <= syncPulse;
                                end

        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:

```
/*************************************************************************
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;
    }
};
```

```
/*************************************************************************
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 calss 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.

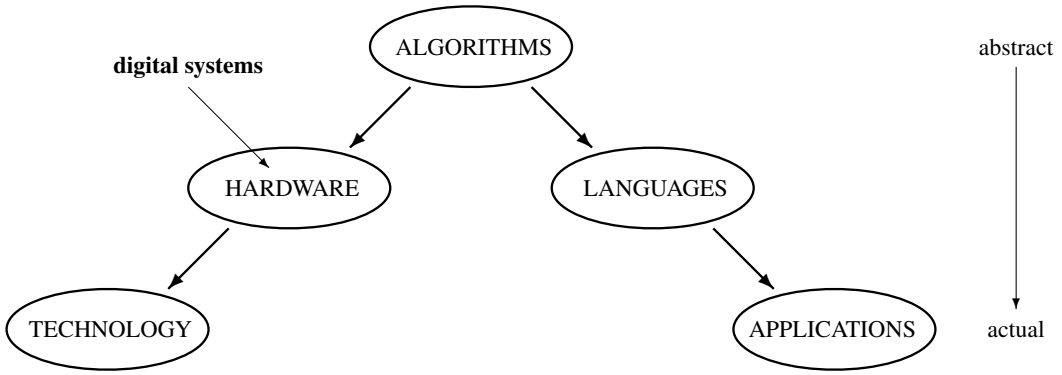


Figure 1.9: **What is computer science?** The domain of digital systems provides techniques for designing the hardware involved in computation.

Definition 1.1 *Computer science (see also Figure 1.9) 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 1.10 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 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.

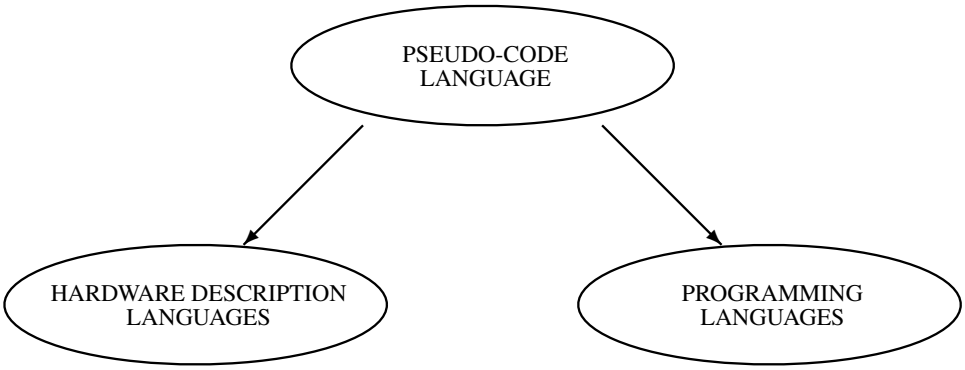


Figure 1.10: **The linguistic context in computer science.** Human mind uses pseudo-code languages to express informally a computation. To describe the circuit associated with the computation a rigorous HDL (hardware description language) is needed, and to describe the program executing the computation 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, de 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 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,*

$$f : (X \times S) \to S$$

is the **state transition function***, and*

$$g : (X \times S) \to Y$$

is the **output transition function**.

◇

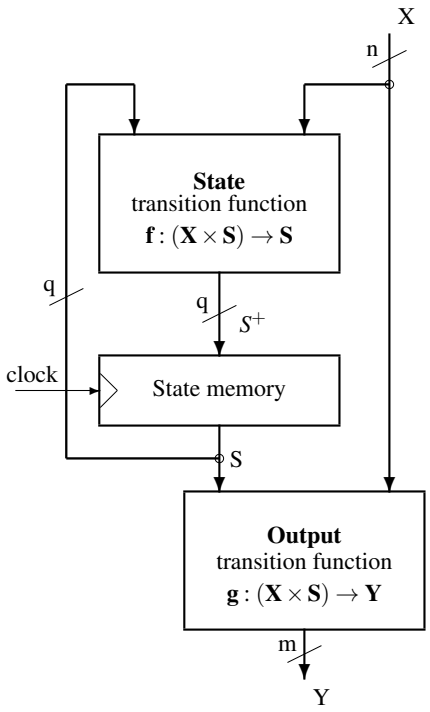


Figure 1.11: Digital system.

A digital system (see Figure 1.11) 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 $x_t \in X = \{0,1\}^n$, $y_t \in Y = \{0,1\}^m$, $s_t \in S = \{0,1\}^q$ 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.

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

- the elements of the sets X, Y and S are binary cods of n, m and q bits – 0s and 1s – which are be codded 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 1.12), 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)

- 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 1.13 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 required; it is the **register** used to *register*, synchronized with the active edge of the clock signal, the state computed at the moment $t-1$ 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 edge of clock, and must stay unchanged t_h (*hold* time) after; the propagation time after the clock is t_p.

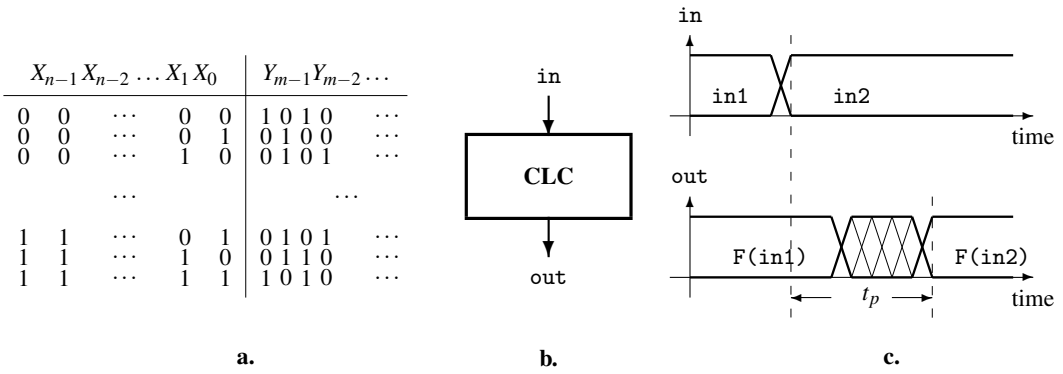


Figure 1.12: **The module for non-sequential functions.** **a.** The table used to define the function as a correspondence between **all** input binary configurations `in` and binary configurations `out`. **b.** The logic symbol for the *combinatorial logic circuit* – CLC – which computes `out = F(in)`. **c.** The wave forms describing the time behaviour of the circuit.

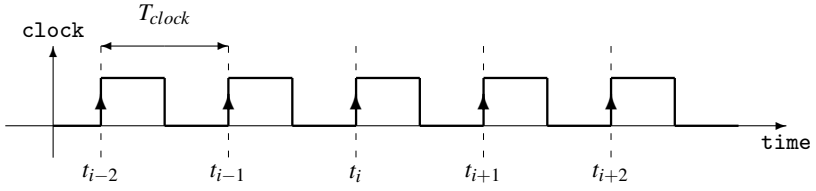


Figure 1.13: **The clock.** This clock signal is active on its positive edge (negative edge as active edge is also possible). The time interval between two positive transitions is the period T_{clock} of the clock signal. Each positive transition marks a discrete moment of time.

(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*

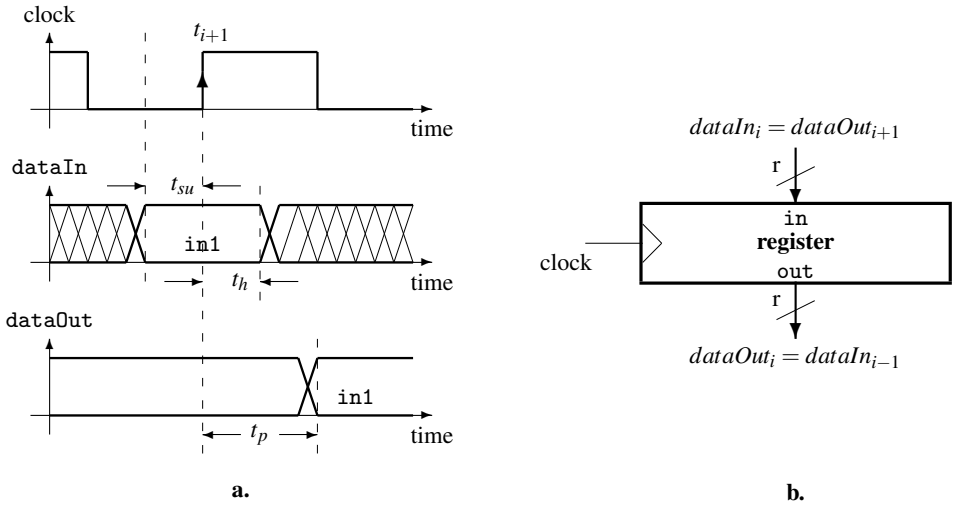


Figure 1.14: **The register. a.** The wave forms describing timing details about how the register swithces around the active edge of clock. **b.** The logic symbol used to define the static behaviour of the register when both, inputs and outputs are stable between two active edges of the clock signal.

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 1.15). 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*
- **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*

```
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(next ,
                               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 1.15: **The top module for the general form of a digital system.**

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.

```
/*************************************************************************
File name:      stateTransition.v
Circuit name:   State Transition dummy module
Description:    the connections of the state transition module
*************************************************************************/
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
```

```
/*************************************************************************
File name:        outputTransition.v
Circuit name:     Output Transi outputtion
Description:      the connections of the output transition module
*************************************************************************/
module outputTransition #(`include "0_parameter.v")
        (output [m-1:0] out   ,
         input  [q-1:0] state ,
         input  [n-1:0] in    );
    // describe here the output transition function
endmodule
```

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

```
/*************************************************************************
File name:        parameter.v
Circuit name:     it is not a circuit
Description:      the parameters involved in all module of the design are
                  defined here
*************************************************************************/
parameter n = 8, // the input is coded on 8 bits
          m = 8  // the output is coded on 8 b
```

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.

◇

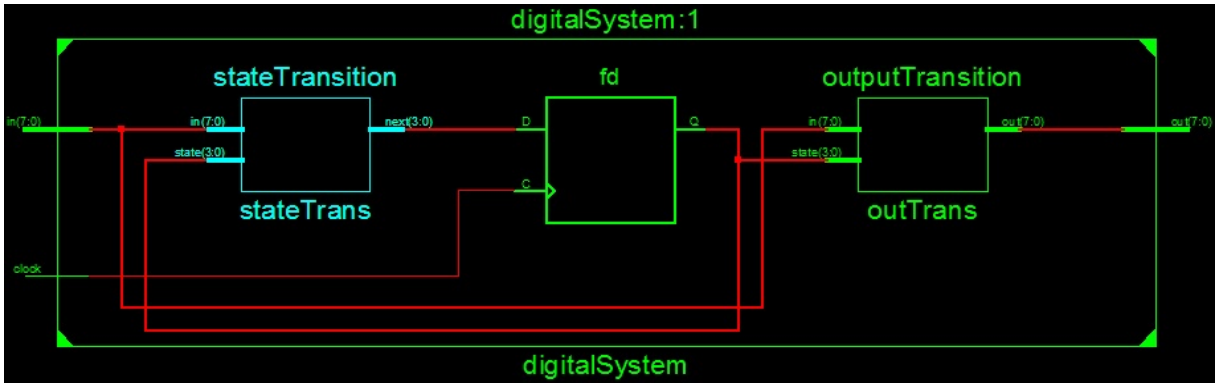


Figure 1.16: **The result of the synthesis for the module** `digitalSystem`**.**

The synthesis of the generic structure, just defined, is represented in Figure 1.16, 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*.

It's the time for few example using the simplest forms for the functions f and g. Let us consider first the simple case of a system with no internal state ($S = \emptyset$):

$$DS = (X, Y, g)$$

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

$$g : X \to Y$$

is the output transition function. Because the function g has the general form $y_t = g(q_t, x_t)$ the time evolution is not important and the actual system will be a clockless one with no internal registers to store the state. The following examples give us only a flavor about what digital design means.

Example 1.6 *Let us use the* Verilog *HDL to describe an adder for 4-bit numbers (see Figure 1.17a). The description which follows is a* **behavioral** *one, because we know* **what** *we intend to design, but we do not know yet* **how** *to design the internal structure of an adder.*

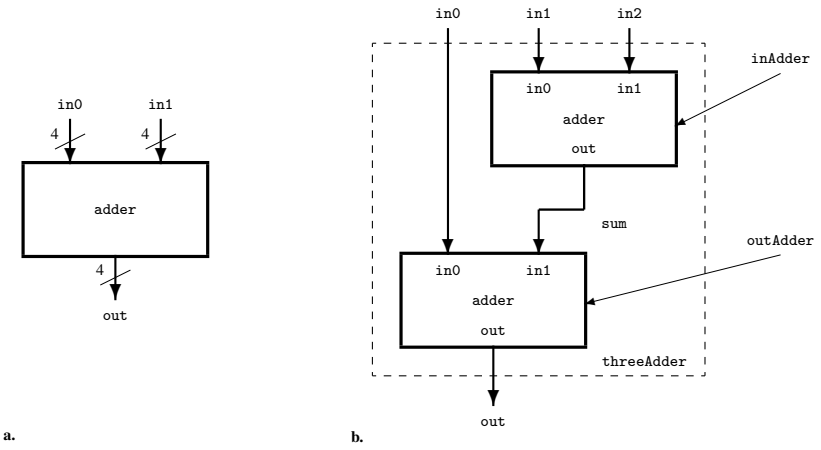


Figure 1.17: **The first examples of digital systems. a.** The two 4-bit numbers adder, called `adder`. **b.** The structure of an adder for 3 4-bit numbers, called `threeAdder`.

The Verilog code describing the module `adder` *is:*

```
/*****************************************************************
File name:      adder.v
Circuit name:   Adder
Description:    The module 'adder' has 2 4-bit inputs and one 4-bit output
                The circuit adds modulo 16; do not provide carry output
*****************************************************************/
module adder(out, in0, in1);
    output [3:0] out ;  // 4-bit output
    input  [3:0] in0 ;  // 4-bit input
    input  [3:0] in1 ;  // 4-bit input

    assign out = in0 + in1;
endmodule
```

The story just told by the previous Verilog module is: "the 4-bit adder has two inputs, `in0, in1`*, one output,* `out`*, and its output is continuously assigned to the value obtained by adding modulo 16 the two input numbers".*

◇

What we just learned from the previous first simple example is summarized in the following **Verilog-Summary**.

VerilogSummary 1 :

module : keyword which indicates the beginning of the description of a circuit as a module having the name which immediately follows (in our example, the name is: `adder`)

endmodule : keyword which indicates the end of the module's description which started with the previous keyword **module**

output : keyword used to declare a terminal as an output (in our example the terminal `out` is declared as output)

input : keyword used to declare the terminal as an input (in our example the terminals `in0` and `in1` are declared as inputs)

assign : keyword called the *continuous assignment*, used here to specify the function performed by the module (the output `out` takes continuously the value computed by adding the two input numbers)

(...) : delimiters used to delimit the list of terminals (external connections)

, : delimiter to separate each terminal within a list of terminals

; : delimiter for end of line

[...]: delimiters which contains the definition of the bits associated with a connection, for example [3:0] define the number of bits for the three connections in the previous example

+ : the operator `add`, the only one used in the previous example.

The description of a digital system is a **hierarchical construct** starting from a top module populated by modules, which are similarly defined. The process continues until very simple module are directly described. Thus, the functions f and g are specified by HDL programs (in our case, in the previous example by a Verilog program).

The main characteristic of the digital design is **modularity**. A problem is decomposed in many simpler problems, which are solved similarly, and so on until very simple problems are identified. Modularity means also to define as many as possible identical modules in each design. This allow to replicate many times the same module, already designed and validated. *Many & simple* modules! Is the main slogan of the digital designer. Let's take another example which uses as module the one just defined in the previous example.

Example 1.7 *The previously exemplified module (*`adder`*) will be used to design a modulo 16 3-number adder, called* `threeAdder` *(see Figure 1.17b). It adds 3 4-bit numbers providing a 4-bit result (modulo 16 sum). Follows the* **structural** *description:*

```
/*****************************************************************
File name:      threeAdder.v
Circuit name:   Three Input Adder
Description:    The module 'threeAdder' has 3 4-bit inputs and one 4-bit
                output. The circuit adds modulo 16 three numbers;
                do not provide carry output
*****************************************************************/
module threeAdder( output   [3:0]  out,
                   input    [3:0]  in0,
                   input    [3:0]  in1,
                   input    [3:0]  in2);

    wire  [3:0] sum;

    adder   inAdder(.out(sum),
                    .in0(in1),
                    .in1(in2)),
            outAdder(.out(out),
                     .in0(in0),
                     .in1(sum));
endmodule
```

Two modules of `adder` *type (defined in the previous example) are instantiated as* `inAdder`*,* `outAdder`*, they are interconnected using the wire* `sum`*, and are connected to the terminals of the* `threeAdder` *module. The resulting structure computes the sum of three numbers.* ◇

VerilogSummary 2 :

- Another way to specify the type of terminals, inside the list of terminals

- A new keyword: **wire** used to specify internal connections inside the current module (in our example: the 4-bit (numbered from 3 to 0) connection, `sum`, between the output of a module and the input of another module (see also Figure 1.17b))

- How a previously defined module (in our example: `adder`) is two times instantiated using two different names (`inAdder` and `outAdder` in our example)

- A "safe" way to allocate the terminals for a module previously defined and instantiated inside the current module: each original terminal name is preceded by a dot, and followed by a parenthesis containing the name of the wire or of the terminal where it is connected (in our example, `outAdder( ... .in1(sum))` means: the terminal `in1` of the instance `outAdder` is connected to the wire `sum`)

- The successive instantiations of the same module can be separated by a ",".

While the module `adder` is a *behavioral* description, the module `threeAdder` is a *structural* one. The first tells us *what* is the function of the module, and the second tells us *how* its functionality is performed by using a structure containing two instantiation of a previously defined subsystems, and an internal connection.

Once the design completed we need to know if the resulting circuit works correctly. A simple test must be provided in what we call a **simulation environment**. It is a *Verilog* module which contains, besides the **device under test** (`dut`) a stimulus generator and an "output device" used to monitor the behavior of `dut`. In Figure 1.18 this simple simulation environment is presented.

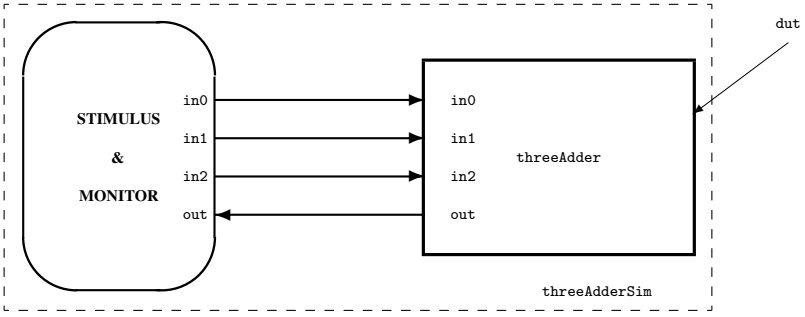


Figure 1.18: **Simulation environment: the module** `threeAdderSim`**.** The *device under test* – `threeAdder` – is interconnected with a "module" which provides the inputs and monitors the output.

Example 1.8 *The* **simulation** *for the circuit designed in the previous example is done using the following* Verilog *module.*

```
/*****************************************************************
File name:      threeAdderSim.v
Circuit name:   Simulator for Three Input Adder
Description:    The module has three sections:
        - stimulus section which:
            - generates the input signals (regs) for simulation
            - provide the connections (wires) for the monitor
        - one instance of the device (module) under test (dut)
        - a monitor to observe the behaviour of the module
*****************************************************************/
 module threeAdderSim;

    // STIMULUS
    reg     [3:0]   in0, in1, in2;  // inputs for dut
    wire    [3:0]   out;            // output of dut

    initial begin       in0 = 4'b0011   ;
                        in1 = 4'b0100   ;
                        in2 = 4'b1000   ;
                    #2  in0 = in0 + 1   ; // after 2 time units
                    #2  in2 = in2 + in0 ; // after another 2 time units
                    #2  in1 = 0         ;
                    #2  $stop           ; // stops the simulation process
            end

    // DEVICE UNDER TEST
    threeAdder dut(out, in0, in1, in2);

    // MONITOR SECTION
    initial
        $monitor
         ("time = %d, in0 = %d, in1 = %d, in2 = %d, out = %d, out = %b",
          $time,      in0,       in1,       in2,       out,       out       );
 endmodule
```

The module `threeAdderSim` *includes the device under test –* `threeAdder` *module –, and a module which provides the "environment" for the module to be tested. The environment-module contains registers used to store the input variables* `in0, in1, in2`*, and a "monitor" used to print the evolving values on the terminals of the device under test.*

The first initial block describes the evolution of the three input variables, starting from the time 0 until the end of simulation specified by `$stop`

The second initial block displays the evolution on the terminals of `dut`*. The output terminal* `out` *is twice displayed, once in decimal form and another time in binary form.*

The two initial blocks are executed in parallel starting from the time 0 of the simulation.

The simulation provides the following result:

```
# time = 0, in0 =  3, in1 =  4, in2 =  8, out = 15, out = 1111
```

```
# time = 2, in0 =  4, in1 =  4, in2 =  8, out =  0, out = 0000
# time = 4, in0 =  4, in1 =  4, in2 = 12, out =  4, out = 0100
# time = 6, in0 =  4, in1 =  0, in2 = 12, out =  0, out = 0000
```

◇

VerilogSummary 3 :

reg[n-1:0]: is an n-bit register used to store an n-bit state variable; it is usually loaded using as trigger the active edge of `clock`

test module is another circuit (usually without external connections), used to simulate the behavior of a specific module, containing:

- registers, or other circuits, for providing the input variables of the tested circuit
- an instantiation of the tested circuit
- a monitor to display the behavior of the outputs of the tested circuit (or a signal inside the tested circuit)

initial : initializes a block of commands executed only once, starting at the time 0 of the simulation

begin : used to delimit the beginning of a block of behavioral statements (equivalent with "{" in C programming language)

end : used to delimit the end of a block of behavioral statements (equivalent with "}" in C programming language)

#$<$ *number* $>$: specifies a delay of $<$ *number* $>$ time units (a time unit is an arbitrary unit of time), and is used in a description only for the purpose of simulating circuits, **but not for synthesizing them**

$stop : stops the process of simulation

$monitor : starts the simulation task of monitoring the signals selected in the associated list

$time : specify the time variable

%d : the number is represented as a decimal one

%b : the number is represented as a binary one

// : it is used to delimit the **comment**, the text which follows until the end of the current line.

The previous three examples represent a simple *bottom-up* approach in digital design. The first example defined an elementary module, the adder. The second, uses the adder to build the structure of a 3-number adder. The first two examples offered a simple image about what a no-state circuit can be, and about what the simulation is and how it can be used for a preliminary test for our design.

Next introductory example is to consider a simple digital system having an *internal state*.

Let us consider the case of adding numbers represented on multiple of 4 bits using circuits able to perform only 4-bit addition. As we know from elementary arithmetic the carry generated by each 4-bit adder can be used to provide sums bigger than 15. If two 12-bit numbers are added, then the process have three steps:

1. the least significant 4 bits of the two operands are added, results the least significant 4 bits of the result and a value for the one-bit signal *carry* which will be stored to be used in the next step

2. the next 4 significant bits of the operands are added and the result is added with the one bit number *carry*, resulting the next 4 significant bits of the result and a new value for *carry* to be stored for the last step

3. the last step adds the most significant 4 bits and the value of carry, resulting the most significant bits of the result. If the last value of *carry* is considered, then it is the 13th bit of the result.

The digital engine able to perform the previously described operations has an internal state, registered in the 1-bit ***register*** called *carry*. It is considered always in the *next* step of the process. It is described in the next example.

How *the next step* of the process is considered in our digital system? The time is marked by a special one-bit signal called ***clock***. The *clock* signal is a periodic pulse train.

Example 1.9 *The digital system for the sequential addition has two 4-bit data inputs,* `in0, in1`*, one 2-bit command input,* `com`*, a clock input,* `clk`*, and the 4-bit output* `sum`*. The command input "tells" the system what to do in each clock cycle: if* `com[1] = 0`*, then the state of the system do not change, else the state of the system takes the value of the carry resulting from adding the current values applied on the data inputs; if* `com[0] = 0`*, then the current addition ignores the value of carry stored as the internal state of the system, else the addition takes into account the state as the carry generated in the previous addition cycle.*

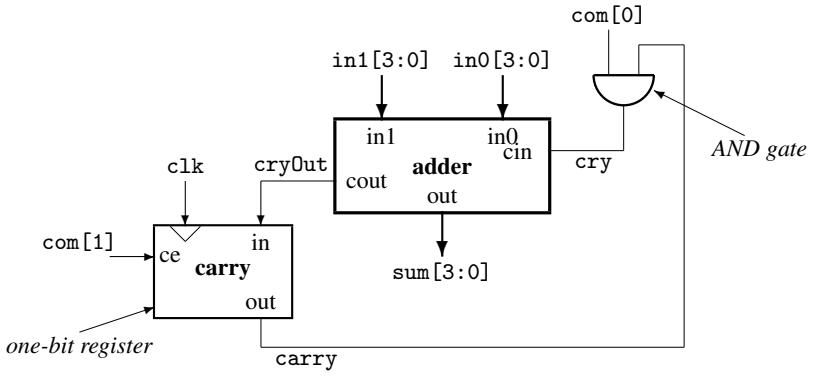


Figure 1.19: **Sequential adder.**

The Verilog code describing the system is the following:

```
/*************************************************************************
File name:      sequentialAdder.v
Circuit name:   Sequential Adder
Description:    The behavior of the circuit is defined by 'com':
                    com = 00 : {carry, sum} = {carry, in1 + in0}
                    com = 01 : {carry, sum} = {carry, in1 + in0 + carry}
                    com = 10 : {carry, sum} = in1 + in0
                    com = 11 : {carry, sum} = in1 + in0 + carry
*************************************************************************/
module sequentialAdder(output  [3:0]   sum,
                       input   [3:0]   in0,
                       input   [3:0]   in1,
                       input   [1:0]   com,
                       input           clk);
    reg     carry   ; // the state register of the system
    wire    cryOut  ;
    wire    cry     ;

    assign  cry = com[0] ? carry : 0       ;
    assign  {cryOut, sum} = in0 + in1 + cry ;

    always @(posedge clk) if (com[1]) carry = cryOut;
endmodule
```

The way this module is used is explained in the following simulation:

```
/*************************************************************************
File name:      testSequentialAdder.v
Circuit name:   Simulator for Sequential Adder
Description:
*************************************************************************/
module testSequentialAdder;
   reg   [3:0]  in0, in1;
   reg   [1:0]  com;
   reg          clk;
   wire  [3:0]  sum;

   initial begin              clk = 0   ;
                forever  #1   clk = ~clk;
            end
   initial begin         com = 2'b10  ;
                         in1 = 4'b1000;
                         in0 = 4'b1001;
                      #2 com = 2'b11  ;
                         in1 = 4'b0000;
                         in0 = 4'b0001;
                      #2 com = 2'b11  ;
                         in1 = 4'b0010;
```

```
                    in0 = 4'b0011;
                #2  com = 2'b11  ;
                    in1 = 4'b0010;
                    in0 = 4'b0011;
                #2  com = 2'b00  ;
                    $stop        ;
            end

    sequentialAdder
        dut( sum, in0, in1, com, clk);

    initial
    $monitor
    ("time = %d clk = %b in1 = %b in0 = %b com = %b sum = %b carry = %b",
    $time, clk, in1, in0, com, sum, dut.carry);
endmodule
```

The result of simulation is:

```
# time = 0 clk = 0 in1 = 1000 in0 = 1001 com = 10 sum = 0001 carry = x
# time = 1 clk = 1 in1 = 1000 in0 = 1001 com = 10 sum = 0001 carry = 1
# time = 2 clk = 0 in1 = 0000 in0 = 0001 com = 11 sum = 0010 carry = 1
# time = 3 clk = 1 in1 = 0000 in0 = 0001 com = 11 sum = 0001 carry = 0
# time = 4 clk = 0 in1 = 0010 in0 = 0011 com = 11 sum = 0101 carry = 0
# time = 5 clk = 1 in1 = 0010 in0 = 0011 com = 11 sum = 0101 carry = 0
# time = 6 clk = 0 in1 = 0010 in0 = 0011 com = 11 sum = 0101 carry = 0
# time = 7 clk = 1 in1 = 0010 in0 = 0011 com = 11 sum = 0101 carry = 0
```

The result of synthesis is presented in Figure 1.19, where:

- *the* **adder** *module adds two 4-bit numbers (*`in1[3:0]`*,* `in2[3:0]`*) and the one-bit number* `cin` *(carry-in) providing the 4-bit sum (*`out[3:0]`*) and the one bit number* `cout` *(carry-out)*
- *the one-bit register* **carry** *is used to store, synchronized with the positive edge of the clock signal* `clk`*, the value of the carry signal,* `cout` *generated by the adder, only when the input* `ce` *(clock enable) is 1 (*`com[1] = 1`*) enabling the switch of the register*
- *the 2-input AND gate applies the output of the register to the* `cin` *only when* `com[0] = 1`*; when* `com[0] = 0`*,* `cin = 0`

Both, simulation and synthesis are performed using specific software tools (for example: ModelSim *for simulations and* Xilinx ISE *for synthesis)*

◇

VerilogSummary 4 :

cond ? a : b : is the well known a C construct which selects `a` if `cond = 1`, else selects `b`

{a, b} : represent the concatenation of `a` with `b`

always @(*<activating condition>* **)** : is a *Verilog* construct activated whenever the condition *<activating condition>* is fulfilled

posedge : keyword used to specify the positive edge of the clock signal (**negedge** specifies the negative edge of the clock signal)

always @(posedge clock) : each positive transition of `clock` will trigger the action described inside the body of the construct **always**

if (cond) ... : used to specify the conditioned execution

forever : keyword indicating an unending repetition of the subsequent action

name1.name2 : selects the signal `name2` from the module `name1`

The previous examples offered a flavor about what the digital design is: using an HDL (Verilog, for example) two kinds of descriptions can be provided – behavioral and structural – both, being used to simulate and/or to synthesize a digital system. Shortly: ***describe, simulate, synthesize*** is the main triad of the digital design.

1.3 Different embodiment of digital systems

The physical embodiment of a digital system evolved, in the second part of the previous century, from circuits built using vacuum tubes to now a day complex systems implemented on a single die of silicon containing billions of components. We are here interested only by the actual stage of technology characterized by an evolutionary development and a possible revolutionary transition.

The evolutionary development is from the multi-chip systems approach to the *system on a chip* (SoC) implementations.

The revolutionary transition is from *Application Specific Integrated Circuit* (ASIC) approach to the **fully programmable solutions** for SoC.

SoC means integrating on a die a big system which, sometimes, involve more than one technology. Multi-chip approach was, and it is in many cases, necessary because of two reasons: (1) the big size of the system and, more important, (2) the need of use of few incompatible technologies. For example, there are big technological differences in implementing analog or digital circuits. If the circuit is analog, there is also a radio frequency sub-domain to be considered. The digital domain has also its specific sub-domain of the dynamic memories. Accommodating on the same silicon die different technologies is possible but the price is sometimes too big. The good news is that there are continuous technological developments providing cheap solutions for integrating previously incompatible technologies.

An ASIC provides very efficient solutions for well defined functions and for big markets. The main concern with this approach is the lack of functional flexibility on a very fast evolving market. Another problem with the ASIC approach is related with the “reusability” of the silicon area which is a very expensive resource in a digital system. For example, if the multiplication function is used in few stages of the algorithm performed by an ASIC, then a multiplication circuit must be designed and placed on

silicon few times even if the circuits stay some- or many-times unused. An alternative solution provides only one multiplier which is "shared" by different stages of the algorithm, if possible.

There are different types of "programmable" digital systems:

- **reconfigurable systems**: are physical structures, having a set of useful features, can be configured, to perform a specific function, by the binary content of some specific storage registers called *configuring registers*; the flexibility of this approach is limited to the targeted application domain
- **programmable circuits**: are general purpose structures whose interconnection and simple functionality are both programmed providing any big and complex systems; but, once the functionality in place, the system performs a fix function
- **programmable systems**: are designed using one or many programmable computing machines able to provide any transfer function between its inputs and outputs.

All these solutions must be evaluated takeing into account their flexibility, speed performance, complexity, power consumption, and price. The *flexibility* is minimal for configurable systems and maximal for programmable circuits. *Speed performance* is easiest to be obtained with reconfigurable systems, while the programmable circuits are the laziest at big complexities. *Complexity* is maximal for programmable circuits and limited for reconfigurable systems. *Power consumption* is minimal for reconfigurable solutions, and maximal for programmable circuits. *Price* is minimal for reconfigurable systems, and maximal for programmable circuits. In all the previous evaluations programmable systems are avoided. Maybe this is the reason for which they provide overall the best solution!

Designing digital circuits is about the hardware support of programmable systems. This book provides knowledge on circuits, but the final target is to teach how to build various programmable structures. Optimizing a digital system means to have a good balance between the physical structure of circuits and the informational structure of programs running on them. Because the future of complex systems belongs to the programmable systems, the hardware support offered by circuits must be oriented toward programmable structures, whose functionality is actualized by the embedded information (program).

Focusing on programmable structures does not mean we ignore the skills involved in designing ASICs or reconfigurable systems. All we discuss about programmable structures applies also to any kind of digital structure. What will happen will be that at a certain level in the development of digital systems features for accepting program control will be added.

1.4 Correlated domains

Digital design must be preceded and followed by other disciplines. There are various prerequisites for attending a digital design course. These disciplines are requested for two reasons:

- the student must be *prepared* with an appropriate pool of knowledge
- the student must be *motivated* to acquire a new skill.

In an ideal world, a student is prepared to attend digital design classes by having knowledge about: *Boolean algebra* (logic functions, canonic forms, minimizing logic expressions), *Automata theory* (formal languages, finite automata, ... Turing Machine), *Electronic devices* (MOS transistor, switching theory), *Switching circuits* (CMOS structure, basic gates, transmission gate, static & dynamic behavior of the basic structures).

In the same ideal world, a student can be motivated to approach the digital design domain if he payed attention to *Theory of computation*, *Microprocessor architecture*, *Assembly languages*.

Attending the classes of Digital Systems is only a very important step on a long journey which suppose to attend a lot of other equally important disciplines. The most important are listed bellow.

Verification & testing For complex digital system verification and testing become very important tasks. The design must be verified to be sure that the intended functionality is in place. Then in each stage, on the way from the initial design to the fabrication of the actual chip, various tests are performed. Specific techniques are developed for verification and testing depending on the complexity of the design. Specific design techniques are used to increase the efficiency of testing. *Design for testability* is a well developed sub-domain which helps us with design tricks for increasing the accuracy and speed of testing.

Physical design The digital system designer provides only a description. It is a program written in a HDL. This description must be used to build accurately an actual chip containing many hundred of million of circuits. It is a multi-stage process where after circuit design, simulation, synthesis, and functional verification, done by the digital design team, follow **layout design & verification**, **mask preparation**, **wafer fabrication**, **die test**. During this long process a lot of additional technical problem must be solved. A partial enumeration of them follows.

- **Clock distribution**: The *clock* signal is a pulse signal distributed almost uniformly on the whole area of the chip. For a big circuit the clock distribution is a critical problem because of the power involved and because of the accuracy of the temporal relation imposed for it.

- **Signal propagation**: Besides clock there are a lot of other signals which can be critical if they spread on big parts of the circuit area. The relation between these signals makes the problem harder.

- **Chip interface circuits**: The electrical charge of an interface circuit is much bigger than for the internal one. The capacitance load on pins being hundred times bigger the usual internal load, the output current for pin driver must be correspondingly.

- **Powering**: The switching energy is provided from a DC power supply. The main problem is to have enough energy right in time at the power connections of each circuit form the chip. Power distribution is made difficult by the inductive effect of the power connections.

- **Cooling**: The electrical energy introduced in circuit, through the power system, must be then, unfortunately, extracted as caloric energy (heat) by cooling it.

- **Packaging**: The silicon die is mounted in a package which must fulfil a lot of criteria. It must allow powering and cooling the die it contains. Also, it must provide hundreds or even thousands external connections. Not to mention protection to cosmic rays, ….

- **Board design**: The chips are designed to be mounted on boards where they are interconnected with other electronic components. Because of the very high density of connections, designing a board is a very complex job involving knowledge from a lot of related domains (electromagnetism, mechanics, chemistry, …).

- **System design**: Actual applications are finalized as packaged systems containing one or many boards, sometimes interconnected with electro-mechanical devices. Putting together many components, powering them, cooling them, protecting them from disturbing external (electromagnetic, chemical, mechanical, ...) factors, adding esthetic qualities require multi-disciplinary skills.

For all these problems specific knowledge must be acquired attending special classes, course modules, or full courses.

Computer architecture Architectural thinking is a major tendency in the contemporary word. It is a way to discuss about the functionality of an object ignoring its future actual implementation. The architectural approach helps us to clarify first what we intend to build, unrestricted by the implementation issues. Computer architecture is a very important sub-domain of computer science. It allow us to develop independently the hardware domain and the software domain maintaining in the same time a high "communicating channel" between the two technologies: one referring to the physical structures and another involving the informational structure of programs.

Embedded systems In an advanced stage of development of digital system the physical structure of the circuits start to be interleaved with the informational structure of programs. Thus, the *functional* flexibility of the system and its efficiency is maximized. A digital system tend to be more and more a computational system. The computation become embedded into the core of a digital system. The discipline of embedded system or embedded computation[3] starts to be a *finis coronat opus* of digital domain.

Project management Digital systems are complex systems. In order to finalize a real product a lot of activities must be correlated. Therefore, an efficient management is mandatory for a successful project. More, the management of the digital system project has some specific aspects to be taken into account.

Business & Marketing & Sales Digital systems are produced to be useful. Then, they must spread in our human community in the most appropriate way. Additional, but very related skills are needed to enforce on the market a new digital system. The knowledge about business, about marketing and sales is crucial for imposing a new design. A good, even revolutionary idea is necessary, but absolutely insufficient. The pure technical skills must be complemented by skills helping the access on the market, the only place where a design receives authentic recognition.

1.5 Problems

Problem 1.1 *Let be the full 4-bit adder described in the following Verilog module:*

```
module fullAdder(   output [3:0]    out     ,
                    output          crOut   ,   // carry output
                    input  [3:0]    in0     ,
                    input  [3:0]    in1     ,
```

[3] In DCAE chair of the Electronics Faculty, in Politehnica University of Bucharest this topics is taught as *Functional Electronics*, a course introduced in late 70s by the Professor Mihai Dr'ag'anescu.

```
                    input           crIn    );  // carry input

    wire    [4:0]   sum ;

    assign  sum     = in0 + in1 + crIn  ;
    assign  out     = sum[3:0]          ;
    assign  crOut   = sum[4]            ;

endmodule
```

Use the module `fullAdder` *to design the following 16-bit full adder:*

```
module bigAdder(output [15:0]   out     ,
                output          crOut   ,   // carry output
                input  [15:0]   in0     ,
                input  [15:0]   in1     ,
                input           crIn    );  // carry input

    // ???

endmodule
```

The resulting project will be simulated designing the appropriate test module.

Problem 1.2 *Draw the block schematic of the following design:*

```
module topModule(   output  [7:0]   out,
                    input   [7:0]   in1,
                    input   [7:0]   in2,
                    input   [7:0]   in3);
    wire    [7:0]   wire1, wire2;

    bottomModule    mod1(   .out(wire1  ),
                            .in1(in1    ),
                            .in2(in2    )),
                    mod2(   .out(wire2  ),
                            .in1(wire1  ),
                            .in2(in3    )),
                    mod3(   .out(out    ),
                            .in1(in3    ),
                            .in2(wire2  ));
endmodule
```

```
module bottomModule(output [7:0] out,
                    input  [7:0] in1,
                    input  [7:0] in2);
    // ...
endmodule
```

Synthesize it to test your solution.

Problem 1.3 *Let be the schematic representation of the design* `topSyst` *in Figure 1.20. Write the Verilog description of what is described in Figure 1.20. Test the result by synthesizing it.*

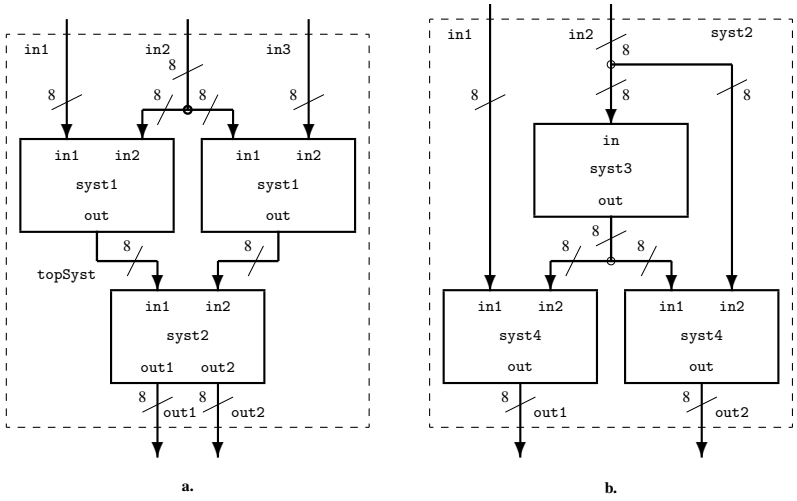


Figure 1.20: **The schematic of the design** `topSyst`**. a.** The top module `topSyst` **b.** The structure of the module `syst2`.

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 $f_{comb} : X \to Y$, 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; $f_{store} : X \to X$, 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:

```
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]                    ;
```

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:

```
if (test) out = action1;
  else    out = action2;
```

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

- **Add** function: `state[n-1:0] + state[q-1:2*n]`, 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).

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:

$$(a+b+c+d+e+f+g+h)' = (((((((a+b)+c)+d)+e)+f)+g)+h)'$$

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.

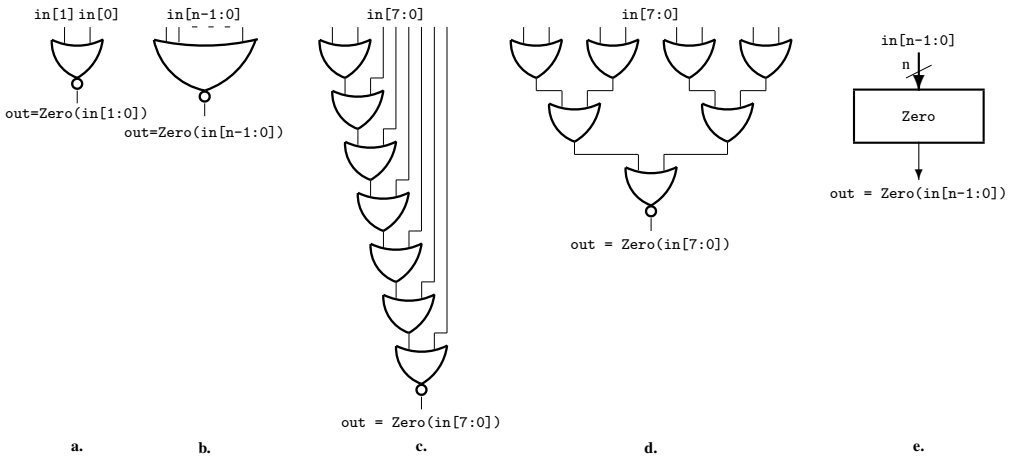


Figure 2.1: **The `Zero` circuit. a.** The 2-input `Zero` circuit is a 2-input NOR. **b.** The `n`-input `Zero` circuit is a n-input NOR. **c.** The 8-input `Zero` circuit as a degenerated tree of 2-input ORs. **d.** The 8-input `Zero` circuit as a balanced tree of 2-input ORs. **e.** The logic symbol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

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:

$$\texttt{out = sel'} \cdot \texttt{in0} + \texttt{sel} \cdot \texttt{in1}$$

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.

The selection circuit for two *n*-bit inputs is ***functionally*** (behaviorally) described by the following *Verilog* module:

```
/*****************************************************************
File name:      ifThenElse.v
Circuit name:   2-input multiplexer
Description:    one of inputs in1, in0 is selected by sel
*****************************************************************/
   module ifThenElse #(parameter n = 4)
            (output [n-1:0] out,
             input          sel,
             input  [n-1:0] in1, in0);

        assign out = sel ? in1 : in0;
    endmodule
```

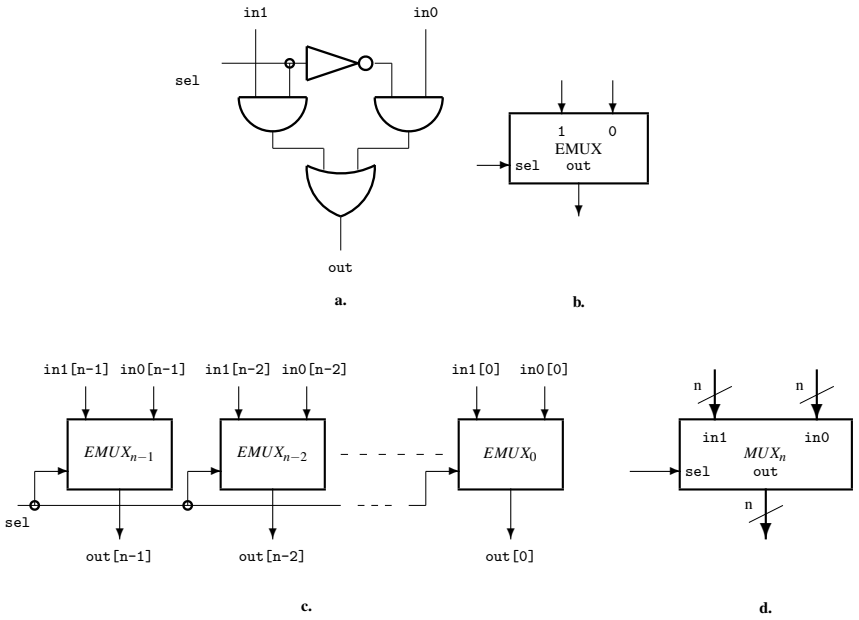


Figure 2.2: **The `selection` circuit. a.** The logic schematic for the elementary selector, EMUX (elementary multiplexer). **b.** The logic symbol for EMUX. **c.** The selector (multiplexor) for n-bit words, MUX_n. **d.** The logic symbol for MUX_n.

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`.

```
/*************************************************************************
File name:      eMux.v
Circuit name:   Elementary multiplexer
Description:    sel ? in1 : in 0
*************************************************************************/
    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
```

```
/*****************************************************************
File name:      ifThenElse.v
Circuit name:   Two n-input multiplexor
Description:    the use of generate statement for a 2-input multiplexor
*****************************************************************/
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.

```
/*****************************************************************
File name:      testIfThenElse.v
Circuit name:   Simulation module for ifThenElse.v
Description:    generate stimulus for a two-input multiplexor
*****************************************************************/
    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;
                   #1    sel = 1'b1;
                   #1    in1 = 4'b1100;
                   #1    $stop;
                end
        ifThenElse dut(out,
                       sel,
                       in1, in0);
        initial $monitor
         ("time = %d sel = %b in1 = %b in0 = %b out = %b",
           $time, sel, in1, in0, out);
    endmodule
```

The result of simulation is:

```
# time = 0 sel = 0_in1 = 0101 in0 = 1011 out = 1011
# time = 1 sel = 1_in1 = 0101 in0 = 1011 out = 0101
# time = 2 sel = 1_in1 = 1100 in0 = 1011 out = 1100
```

The result of synthesis is represented in Figure 2.3.

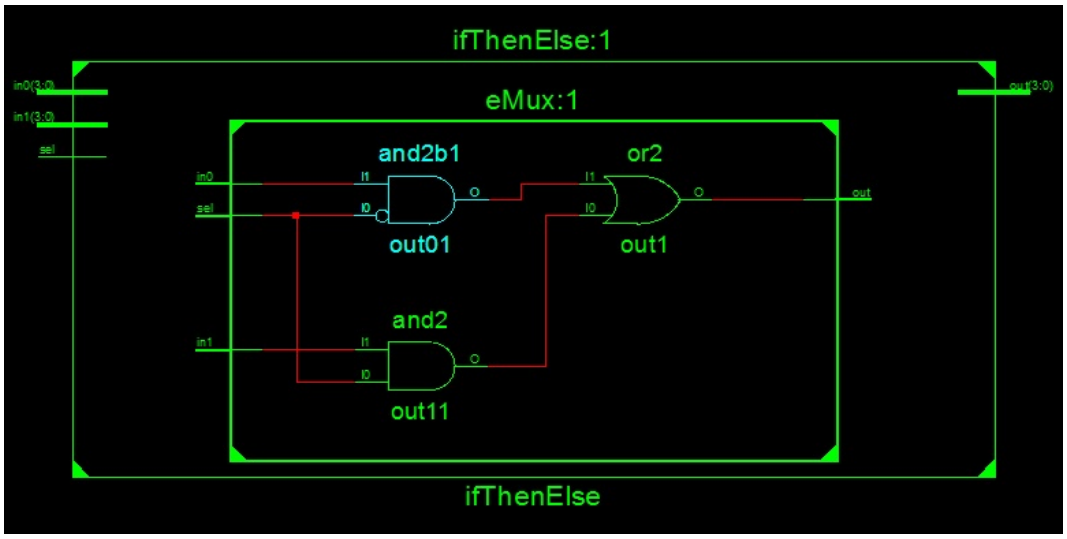


Figure 2.3: **The result of the synthesis for the module** `ifThenElse`**.**

2.1.3 Adder

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

```
/*************************************************************************
File name:      adder.v
Circuit name:   4-bit words adder
Description:    the circuit adds numbers represented on n bits, for n = 4
*************************************************************************/
    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

        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 actuala internal structure of an adder. We start from a 1-bit adder, whose output are described by the following Boolean equations:

$$sum = a \oplus b \oplus c$$

$$carry = a \cdot b + a \cdot c + b \cdot c$$

where, a, b and c are one bit Boolean variables. Indeed, the *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 *carry* signal is used by the next binary stage. The expression for carry is written taking into account that the carry signal is one if at least two of the input bits are one: carry is 1 if `a` **and** `b` **or** `a` **and** `c` **or** `b` **and** `c` (the function is the *majority function*). Its expression is embodied also in logic circuits, but not before optimizing its form as follows:

$$carry = a \cdot b + a \cdot c + b \cdot c = a \cdot b + c \cdot (a + b) = a \cdot b + c \cdot (a \oplus b)$$

Because $(a \oplus b)$ is already computed for *sum*, the circuit for *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 `carry` generate the carry signal for the next binary or range.

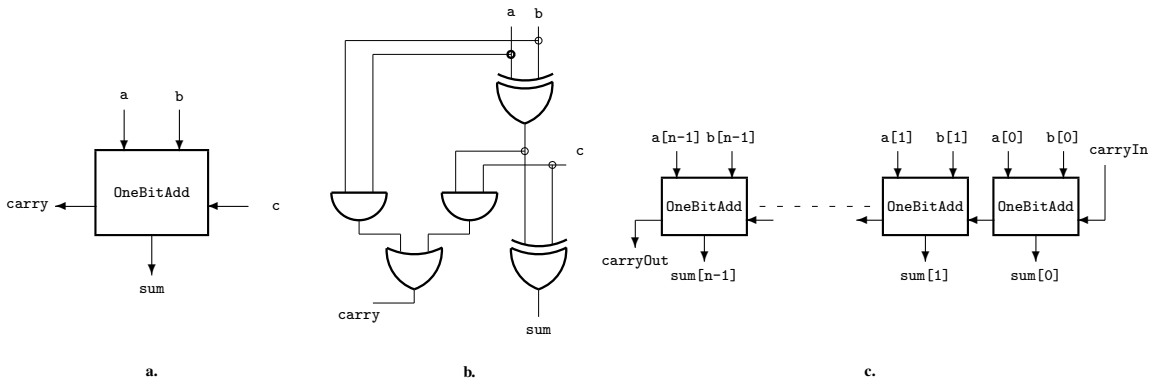


Figure 2.4: **The `adder` circuit. a.** The logic symbol for one-bit adder. **b.** The logic schematic for the one-bit adder. **c.** The block schematic for the n-bit adder.

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:

$$sum = a'b'c + a'bc' + ab'c' + abc$$

$$carry = a'bc + ab'c + abc' + abc$$

a	b	c	sum	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

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

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 `number[n-1:0]` divided by two become {`1'b0, 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 (`(Reset)' = 1`), while the *active* input value is 0 (`(Reset)' = 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_{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_{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 (`Set = 0`) while the *active* input value is 1 (`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 (`out = 0`), and another active-high (active on 1) input, S, to *set* the circuit (`out = 0`). The value 0 must remain to the input R' at least $2t_{pHL}$ 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 $2t_{pLH}$ 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.

VeriSim 2.1 *The Verilog structural description of NAND latch is:*

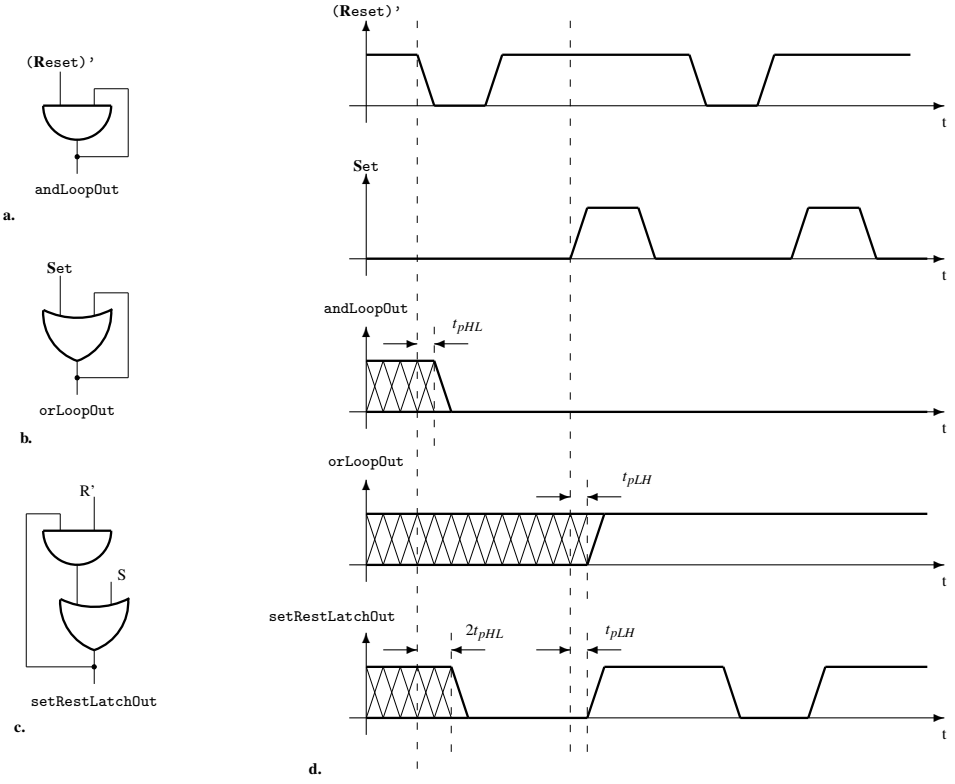


Figure 2.6: **The elementary latches.** Using the loop, closed from the output to one input, elementary storage elements are built. **a**. *AND loop* provides a *reset-only* latch. **b**. *OR loop* provides the *set-only* version of a storage element. **c**. The heterogeneous elementary *set-reset* latch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

```
/*****************************************************************
File name:      lementary_latch.v
Circuit name:   Elementary latch
Description:    the structural description of an elementary latch
*****************************************************************/
module elementary_latch(output out, not_out,
                        input  not_set, not_reset);
    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:

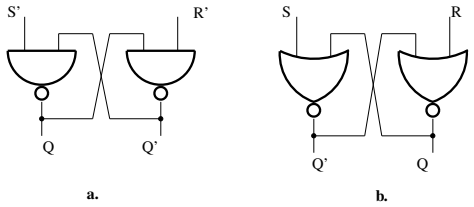


Figure 2.7: **Symmetric elementary latches. a.** Symmetric elementary *NAND latch* with low-active commands S' and R'. **b**. Symmetric elementary *NOR latch* with high-active commands S and R.

```
/*****************************************************************
File name:      test_shortest_input.v
Circuit name:   Tester for the elementary latch
Description:    the circuit generate stimulus for the elementary latch
*****************************************************************/
 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
                  #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 10*ut (*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* 1*ut is unable to switch the circuit. The third experiment, with* 2*ut set signal, generate an unstable simulated, but* **non-actual***, behavior (to be explained by the reader). The fourth experiment, with* 3*ut 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).

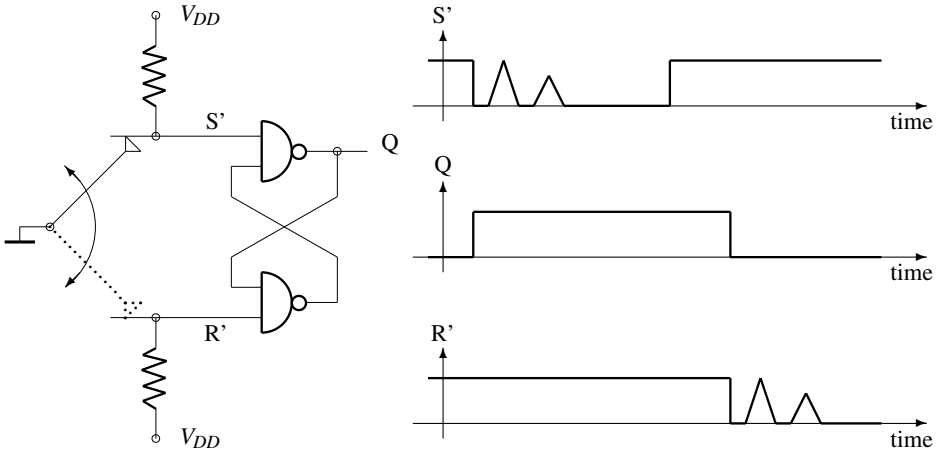


Figure 2.8: **The de-bouncing circuit.**

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.

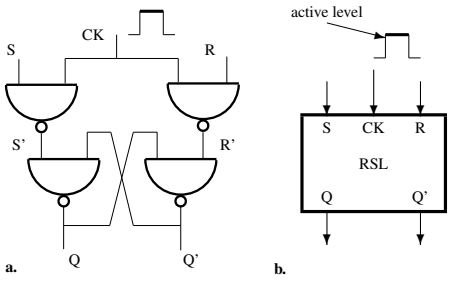


Figure 2.9: **Elementary clocked latch.** The transparent RS clocked latch is sensitive (transparent) to the input signals during the active level of the clock (the high level in this example). **a**. The internal structure. **b**. The logic symbol.

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.*

```
/*****************************************************************
File name:       clocked_nand_latch.v
Circuit name:    Clocked latch
Description:     the circuit is a clocked latch implemented with NAND gates
*****************************************************************/
 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 introducin 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) can not 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 **D***ata*). The circuit is called ***Data Latch***, or simple *D-latch*.

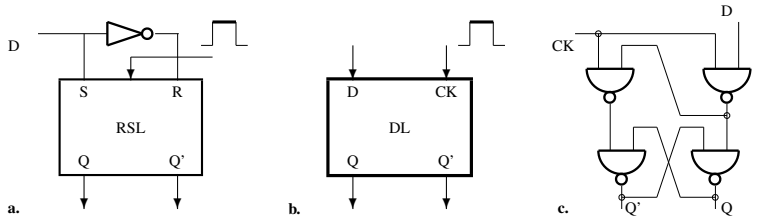


Figure 2.10: **The data latch.** Imposing the restriction $R = S'$ to an RS latch results the **D latch** without non-predictable transitions ($R = S = 1$ is not anymore possible). **a.** The structure. **b.** The logic symbol. **c.** An improved version for the data latch internal structure.

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.*

```
/*****************************************************************
File name:      data_latch.v
Circuit name:   Data Latch
Description:    data latch transparent on the high level of clock
*****************************************************************/
 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.*

```
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

module data_latch(output   out, not_out,
                  input    data, clock);

   not #2  data_inverter(not_data, data);

   clocked_nand_latch  rs_latch(out, not_out, data, not_data, clock);
endmodule
```

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

◇

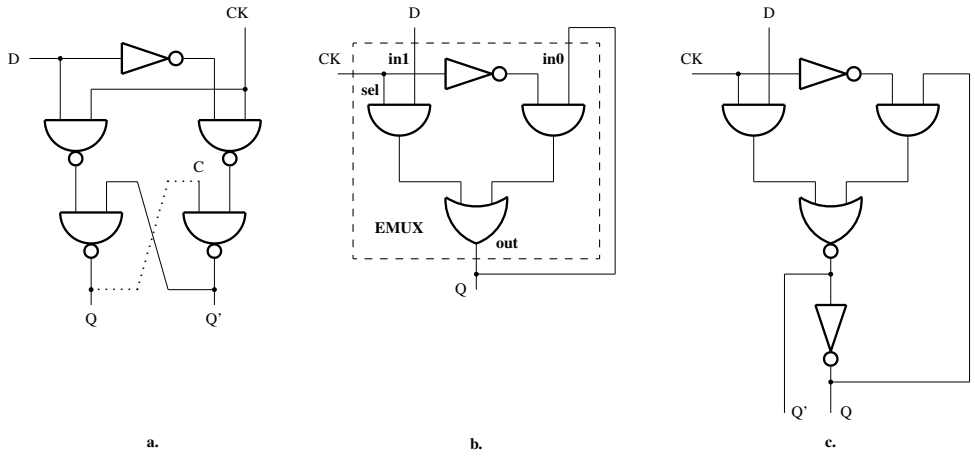


Figure 2.11: **The optimized data latch.** An optimized version is implemented closing the loop over an *elementary multiplexer*, EMUX. **a.** The resulting minimized structure for the circuit represented in Figure 2.10a. **b.** Implementing the minimized form using only inverting circuits.

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:

$$Q = ((D \cdot CK) + (C(D' \cdot CK)')$$

$$Q = ((D \cdot CK) + (C(D + CK'))$$

$$Q = D \cdot CK + C \cdot D + C \cdot CK' (anti - hasard\ redundancy^{1})$$

$$Q = D \cdot CK + C \cdot CK'$$

[1] Anti-hasard redundancy equivalence: `f(a,b,c) = ab + ac + bc' = ac + bc'`

Proof:

`f(a,b,c) = ab + ac + bc' + cc'`, `cc'` is ORed because `xx' = 0` 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')')'`, applying De Morgan law
`f(a,b,c) = (b'c' + a'c)'`, applying again De Morgan law
`f(a,b,c) = (ab'c' + a'b'c' + a'bc + a'b'c)' = (m4 + m0 + m3 + m1)'`, expanding to the disjunctive normal form
`f(a,b,c) = m2 + m5 + m6 + m7 = a'bc' + ab'c + abc' + abc`, using the "complementary" minterms
`f(a,b,c) = bc'(a + a') + ac(b + b') = ac + bc'`, q.e.d.

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.

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.*

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

```
/*****************************************************************
File name:      mux_latch.v
Circuit name:   Elementary muultiplexor
Description:    the multiplexor is used to implement a clocked data latch
*****************************************************************/
 module mux_latch(  output  q      ,
                    input   d, ck  );
    assign  q = ck ? d : q;
 endmodule
```

◇

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.

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.)

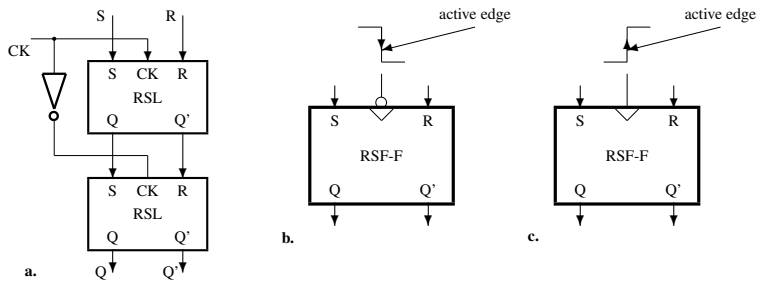


Figure 2.12: **The master-slave principle.** Serially connecting two RS latches, activated with different levels of the clock signal, results a non-transparent storage element. **a.** The structure of a RS master-slave flip-flop, active on the falling edge of the clock signal. **b.** The logic symbol of the RS flip-flop triggered by the *negative edge* of clock. **c.** The logic symbol of the RS flip-flop triggered by the *positive edge* of clock.

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 transparency 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.*

```
/*********************************************************************
File name:      master_slave.v
Circuit name:   Master-Slave set-reset flip-flop
Description:    the structural description of a master-slave flip-flop
*********************************************************************/
 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_+ or t_-) – 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).

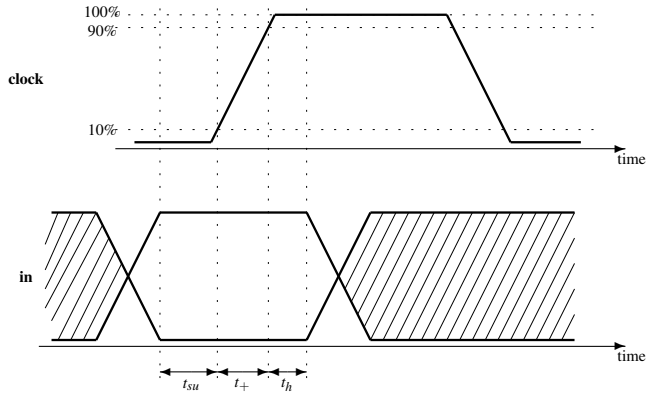


Figure 2.13: **Magnifying the transition of the active edge of the clock signal.** The input data must be stable around the active transition of the clock t_{su} (set-up time) before the beginning of the clock transition, during the transition of the clock, t_+ (active transition time), and t_h (hold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

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.

2.2.5 Metastability

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]).

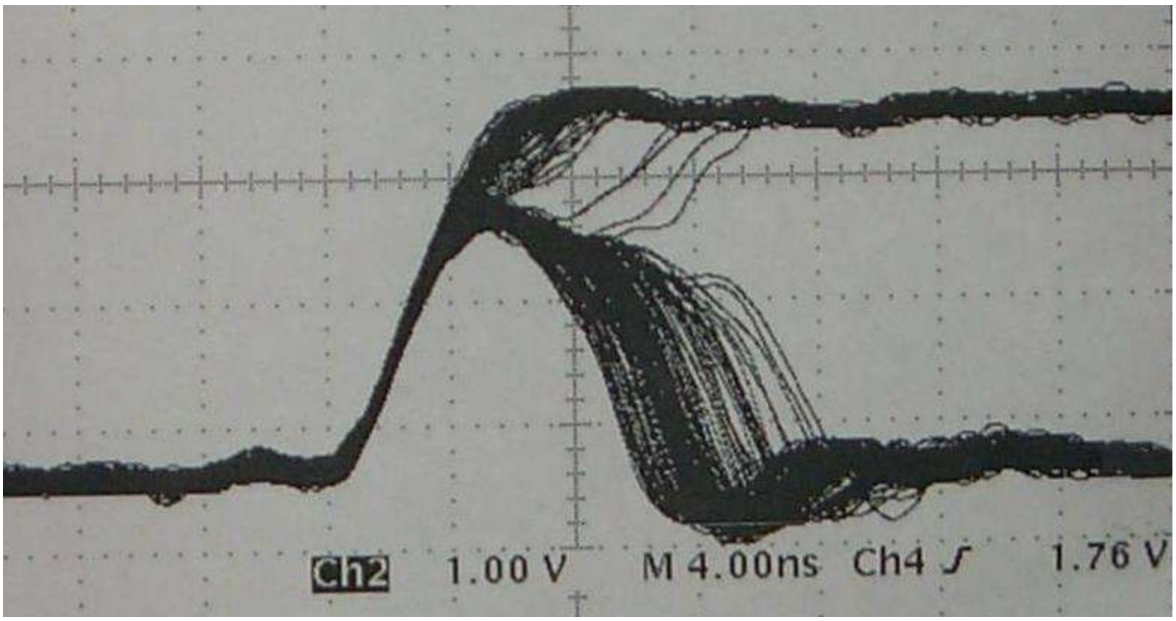


Figure 2.14: Metastability [webRef_4].

2.2.6 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 2.15a). 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 2.15c illustrates the delay effect of this kind of flip-flop.

Warrning! *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.

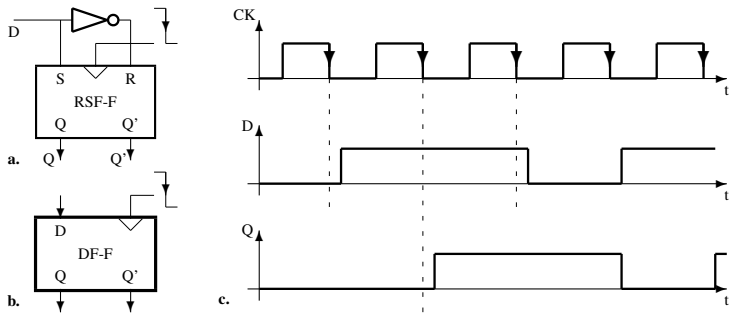


Figure 2.15: **The delay (D) flip-flop.** Restricting the two inputs of an RS flip-flop to $D = S = R'$, results an FF with predictable transitions. **a.** The structure. **b.** The logic symbol. **c.** The wave forms proving the delay effect of the D flip-flop.

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

```
/*********************************************************************
File name:      dff.v
Circuit name:   Delay Flip-Flop (DFF)
Description:    the structural description of a DFF
*********************************************************************/
 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:

```
/*********************************************************************
File name:      dff.v
Circuit name:   Delay Flip-Flop (DFF)
Description:    the behavioral description of a DFF
*********************************************************************/
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.

2.2.7 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.

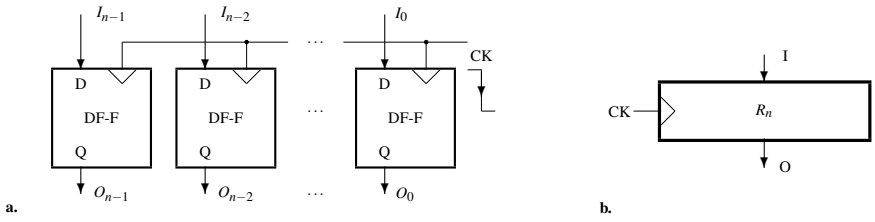


Figure 2.16: **The *n*-bit register. a.** The structure: a bunch of DF-F connected in parallel. **b.** The logic symbol.

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 2.16). 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:*

```
/*****************************************************************
File name:      register.v
Circuit name:   Register of n bits
Description:    the behavioral description of a n-bit register
*****************************************************************/
 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-sinthesizable representation.*

◇

Something very important is introduced by the last two Verilog modules: the distinction between *blocking* and **non-blocking** assignment:

- the **blocking assignment**, $=$: the whole statement is done before control passes to the next
- 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 2.9 *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.*

```
/*****************************************************************
File name:      blockingNonBlocking.v
Circuit name:   Illustrating module for blocking/nonblocking assignment
Description:    shows the effect of blocking and noonblocking assignments
*****************************************************************/
 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
```

```
/*****************************************************************
File name:      blockingNonBlockingSimulation.v
Circuit name:   Testbench for blockingNonBlockingSimulation module
Description:    generate stimulus for blockingNonBlocking module
*****************************************************************/
 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);
    initial $monitor
    ("clock=%b in=%b reg1=%b reg2=%b bOut=%b reg3=%b reg4=%b nbOut=%b",
    clock, in, dut.reg1, dut.reg2, blockingOut, dut.reg3, dut.reg4,
    nonBlockingOut);
 endmodule
```

```
/*************************************************************************
The monitor output
*************************************************************************/
 clock=0  in=01  reg1=xx  reg2=xx  bOut=xx  reg3=xx  reg4=xx  nbOut=xx
 clock=1  in=01  reg1=01  reg2=01  bOut=01  reg3=01  reg4=xx  nbOut=xx
 clock=0  in=10  reg1=01  reg2=01  bOut=01  reg3=01  reg4=xx  nbOut=xx
 clock=1  in=10  reg1=10  reg2=10  bOut=10  reg3=10  reg4=01  nbOut=xx
 clock=0  in=11  reg1=10  reg2=10  bOut=10  reg3=10  reg4=01  nbOut=xx
 clock=1  in=11  reg1=11  reg2=11  bOut=11  reg3=11  reg4=10  nbOut=01
 clock=0  in=00  reg1=11  reg2=11  bOut=11  reg3=11  reg4=10  nbOut=01
 clock=1  in=00  reg1=00  reg2=00  bOut=00  reg3=00  reg4=11  nbOut=10
 clock=0  in=00  reg1=00  reg2=00  bOut=00  reg3=00  reg4=11  nbOut=10
 clock=1  in=00  reg1=00  reg2=00  bOut=00  reg3=00  reg4=00  nbOut=11
 clock=0  in=00  reg1=00  reg2=00  bOut=00  reg3=00  reg4=00  nbOut=11
 clock=1  in=00  reg1=00  reg2=00  bOut=00  reg3=00  reg4=00  nbOut=00
 clock=0  in=00  reg1=00  reg2=00  bOut=00  reg3=00  reg4=00  nbOut=00
```

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 5 :

$=$ **: 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.

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.

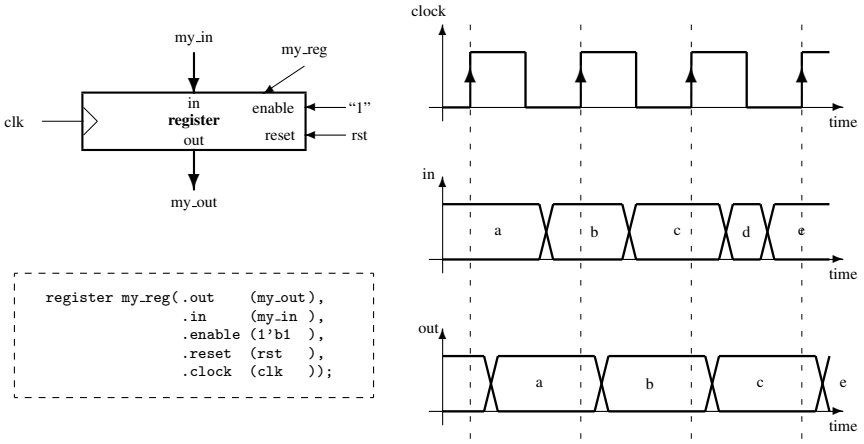


Figure 2.17: **Register at work.** At each active edge of clock (in this example it is the positive edge) the register's output takes the value applied on its inputs if `reset = 0` and `enable = 1`.

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 2.17 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 2.17 the input of the register changes somehow chaotically related to the active edge of the clock, but the output 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 2.17 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.

2.2.8 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:

```
/*************************************************************************
File name:      shiftRegister.v
Circuit name:   Shift Register
Description:    the behavioral description of a left/right shift register
*************************************************************************/
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".

2.2.9 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:

```
/*****************************************************************
File name:      counter.v
Circuit name:   Presetable Counter
Description:    the behavioral description of a presetable counter
*****************************************************************/
 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

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

suggests a selector (a multiplexor), while

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

imposes an increment circuit. Thus, the schematic represented in Figure 2.18 pops up in our mind.

The circuit $\mathbf{INC_8}$ in Figure 2.18 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 **??**, 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:

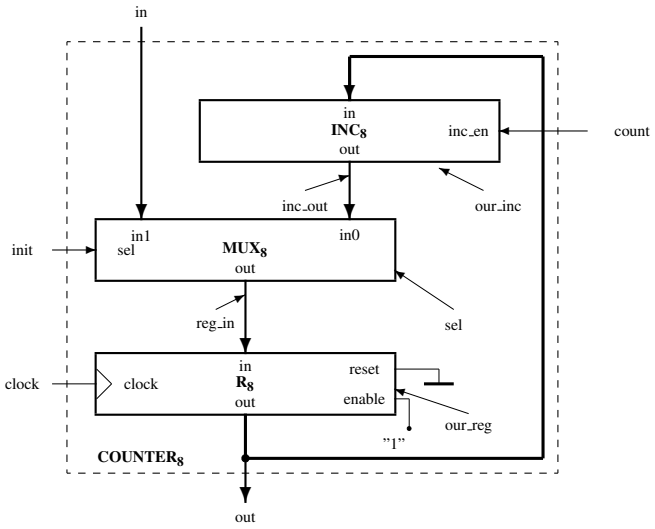


Figure 2.18: **The internal structure of a counter.** If *init* = 1, then the value of the register is initialized to `in`, else if *count* = 1 each active edge of clock loads in register its incremented value.

```
/*************************************************************************
File name:      pixelCorrector.v
Circuit name:   Pixel Corrector System
Description:    the circuit interpolates the missing values in a video
                stream
*************************************************************************/
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 2.19, 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
- 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.

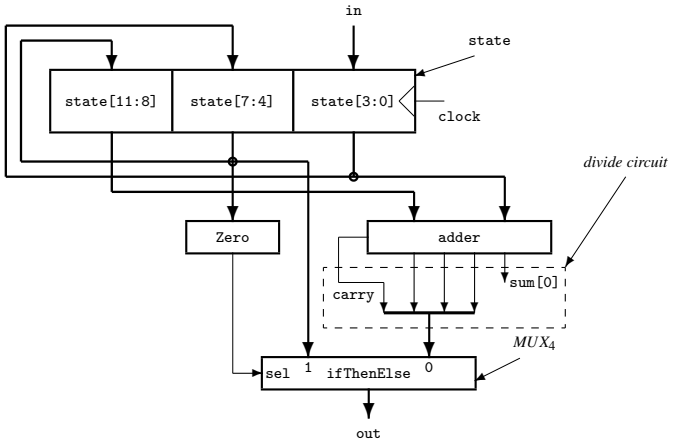


Figure 2.19: **The structure of the `pixelCorrector` circuit.**

Each received value is loaded first as `state[3:0]`, then it moves in the next section. Thus, an input value cames 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.*

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:*

```
/*************************************************************************
File name:      addSub.v
Circuit name:   Adder-Subtracter
Description:    the behavioral description of an adder substracter
*************************************************************************/
module addSub #(parameter n = 4)// defines a n-bit adder
        (output [n-1:0] sum,    // the n-bit result
         output         carry, // carry output (it is borrow 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:*

$$s'(t) = (2 \times s(t-2) + 6 \times s(t-1) + 6 \times s(t+1) + 6 \times s(t+2))/16$$

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 2.2.8, using a register an two input multiplexers, MUX_2. 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* $\mathbf{INC_8}$ *introduces in subsection 2.2.9.*

Problem 2.24 *Write the structural description for the module* `counter` *defined in subsection 2.2.9. Simulate and synthesize it.*

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

```
/*****************************************************************
File name:      accumulator.v
Circuit name:   Sequential Accumulator
Description:    is a dummy module defining the connections only
*****************************************************************/
module accumulator
       (output reg [?:0]  acc    , // register used to accumulate
        input      [15:0] number, // input receiving the stream of numbers
        input      [1:0]  com    , // 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 wave form 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 ggod 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 delay any input change, 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[1]

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,

[1] David Deutch'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 computation 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, $S_{digital\ circuit}$, is given by the dimension of the physical resources used to implement it.* $\diamond$

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, $GS_{digital\ circuit}$, 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]. $\diamond$

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, $AS_{digital\ circuit}$, is given by the dimension of the area on Silicon used to implement it.* $\diamond$

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.

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

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 $C_{digital\ circuit}$ 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:*

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

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

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

$$C_{complex\ circuit} \sim S_{complex\ circuit}.\diamond$$

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

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

```
/*************************************************************************
File name:       random_circ.v
Circuit name:    Example of a complex circuit
Description:     a small complex network of gates
*************************************************************************/
 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
```

$\diamond$

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.

```
/*************************************************************************
File name:       or_prefixes.v
Circuit name:    Example of simple circuit
Description:     a big simple circuit
*************************************************************************/
 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 OR_2 *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.

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

[6]A 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.

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_{in_reg}
2. from register to register through `comb_circ_1`, which determines **minimum period of clock**: $t_{reg\_reg} = T_{min}$, or **maximum frequency** of clock: $f_{max} = 1/T$
3. from input to output through `comb_circ_2`, which determines **maximum combinational path delay**: t_{in_out}
4. from register to output through `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 $T_{min} > t_{reg} + t_{cc_2} + t_{su}$ and $t_h < t_{reg} + t_{cc_2}$, where: t_{reg} is the propagation time through `register` from active edge of clock to output, and t_{cc_2} is the propagation time through `comb_circ_1` on the path 2.

If the system works with the same clock, then $t_{in\_out} < T_{min}$, preferably $t_{in\_out} << T_{min}$. 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 2.18. If we consider* $\#1 = 100ps$ *results:*

- $t_{in\_reg} = t_p(mux2\_8) = 0.1ns$
  (the set-up time for the register is considered too small to be considered)
- $f_{max} = 1/T = 1/(t_p(reg) + t_p(inc) + t_p(mux2\_8)) = 1/(0.2 + 0.1 + 0.1)ns = 2.5\ GHz$
- t_{in_out} *is not defined*

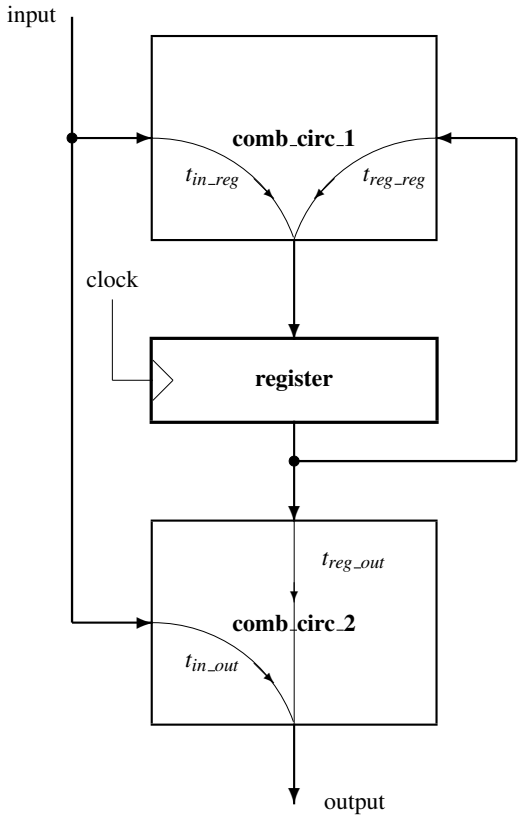


Figure 3.1: **The four critical propagation paths in digital circuits.** *Input-to-register* time (t_{in_reg}) is recommended to be as small as possible in order to reduce the time dependency from the previous sub-system. *Register-to-register* time (T_{min}) must be minimal to allow a high frequency 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

- $t_{reg\_out} = t_p(reg) = 0.2ns \diamond$

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

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

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`*.*

The propagation time for the four critical propagation path of this circuit are:

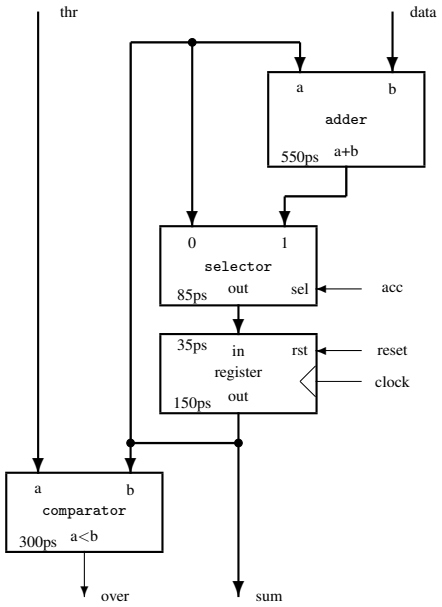


Figure 3.2: **Accumulate & compare circuit.** In the left-down corner of each rectangle is written the propagation time of each module. If `acc = 1` the circuit accumulates, else the content of `register` does not change.

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

◇

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: $t_{reg\_out} = t_{reg}$ – *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.

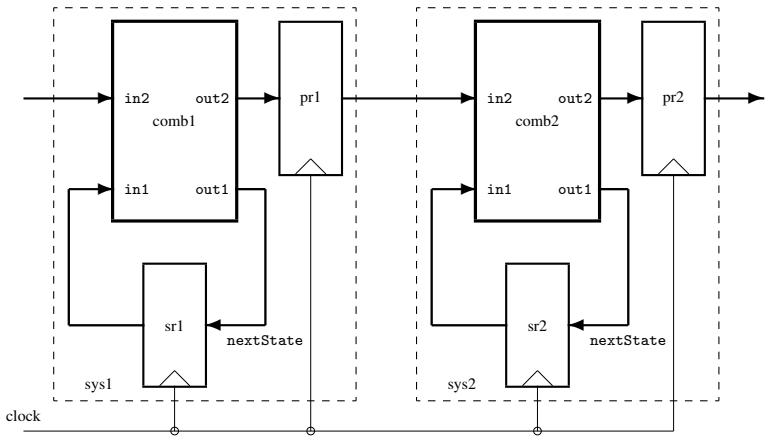


Figure 3.3: **Pipelined connections.**

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.

```
/*****************************************************************
File name:      sys.v
Circuit name:   Pipeline connected sub-systems
Description:    is a dummy code illustrating the principle of pipeline
                connection in a system
*****************************************************************/
module pipelineConnection( output  [15:0]  out     ,
                           input   [15:0]  in      ,
                           input           clock   );
   wire    [15:0]  pipeConnect ;
   sys sys1(   .pr     (pipeConnect),
               .in     (in         ),
               .clock  (clock      )),
       sys2(   .pr     (out        ),
               .in     (pipeConnect),
               .clock  (clock      ));
endmodule

module sys(output  reg [15:0]  pr      ,
           input       [15:0]  in      ,
           input               clock   );
   reg     [7:0]   sr          ;
   wire    [7:0]   nextState   ;
   wire    [15:0]  out         ;
   comb myComb(.out1   (nextState  ),
               .out2   (out        ),
               .in1    (sr         ),
               .in2    (in         ));
   always @ (posedge clock)    begin   pr <= out    ;
                                       sr <= nextState ;
                               end
endmodule

module comb(   output  [7:0]   out1,
               output  [15:0]  out2,
               input   [7:0]   in1 ,
               input   [15:0]  in2 );
   // ...
endmodule
```

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..

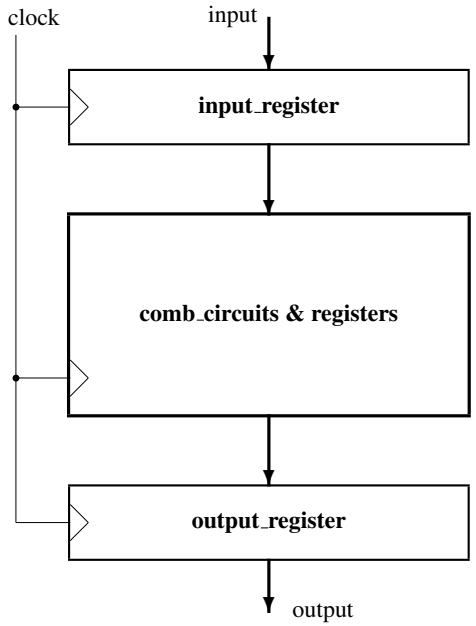


Figure 3.4: **The general structure of a module in a complex digital system.** If any big module in a complex design is buffered with input and output registers, then we are in the ideal situation when: t_{in_reg} and t_{reg_out} are minimized and t_{in_out} is not defined.

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

1. form input to register: $t_{in\_reg} = t_{su}$ – *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**

4. from register to output through: $t_{reg\_out} = t_{reg}$ – *it does not depend by the internal structure of the module*.

Results a very well encapsuled module easy to be integrate in a complex system. The price of this approach consists in an increasing number of circuits (the interface registers) and some restrictions in timing imposed by the additional pipeline stages introduced by the interface registers. These costs can be reduced by a good system level module partitioning.

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, ... h_m`*, described by the following, incompletely defined, but synthesisable, Verilog modules.*

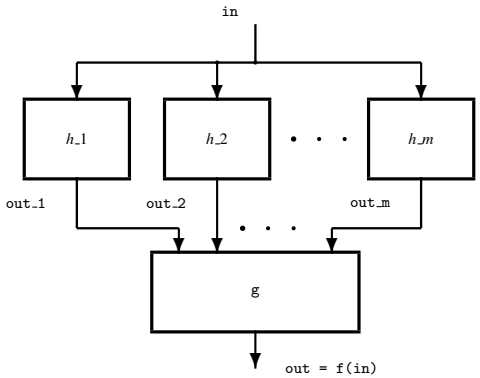


Figure 3.5: **The circuit performing composition.** The function g is composed with the functions `h_1, ... h_m` using a two level circuit. The first level contains m circuits computing *in parallel* the functions `h_i`, and on the second level there is the circuit computing the *reduction*-like function `g`.

```
/*************************************************************************
File name:      f.v
Circuit name:   Function f
Description:    is a dummy Verilog module describing the composition rule
*************************************************************************/
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   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  );
    // ...
endmodule

// ...

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

The content of the file `parameters.v` *is:*

```
/*****************************************************************
File name:      parameters.v
Circuit name:   Parameters file
Description:    define the parameters used in all modules of the design
*****************************************************************/
 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, ... 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:*

```
/*****************************************************************
File name:      f.v
Circuit name:   Dummy description for serial composition
Description:    shows the serial composition of two systems
*****************************************************************/
 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
```

◇

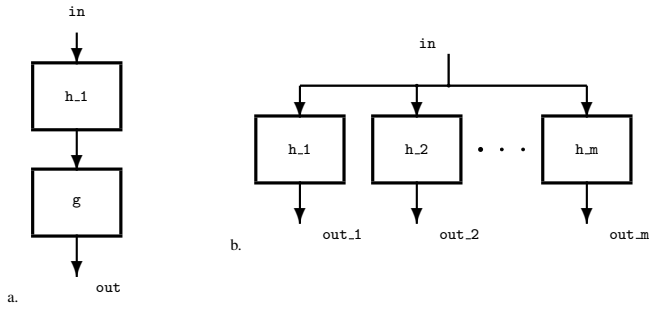


Figure 3.6: **The two limit forms of composition.** a. The serial composition, for $m = 1$, imposing an inherent sequential computation. b. The parallel composition, with no *reduction*-like function, performing data parallel computation.

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:*

```
/*************************************************************************
File name:      f.v
Circuit name:   Dummy description for parallel composition
Description:    shows how are parallel composed many systems
*************************************************************************/
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_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_m #(`include "parameters.v")( output [size_m-1:0] out  ,
                                       input  [sizeIn-1:0] in   );
endmodule
```

The content of the file `parameters.v` *is now:*

```
/*************************************************************************
File name:      parameters.v
Circuit name:   Parameter file
Description:    defines the parameter for the parallel composed system
*************************************************************************/
 parameter  sizeIn  =  8       ,
            size_1  =  12      ,
            // ...
            size_m  =  8       ,
            sizeOut =  size_1 +
                       // ...
                       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.*

```
/*************************************************************************
File name:      inner_prod.v
Circuit name:   Inner Product
Description:    the structural description of the inner product circuit
*************************************************************************/
 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
```

```
/*************************************************************************
File name:      mult.v
Circuit name:   Multiplier
Description:    the behavioral description of the multiply circuit
*************************************************************************/
 module mult #(`include "parameter.v")
        (output [(2*n-1):0] out, input [n-1:0]  m1, m0);
    assign  out = m1 * m0;
 endmodule
```

```
/*************************************************************************
File name:      add4.v
Circuit name:   Four-number Adder
Description:    the behavioral description of a four-number adder
*************************************************************************/
 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
```

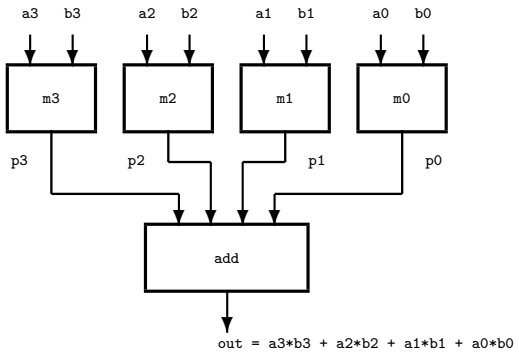


Figure 3.7: **An example of composition.** The circuit performs the scalar vector product for 4-element vectors. The first level compute in parallel 4 multiplications generating the vectorial product, and the second level *reduces* the resulting vector of products to a scalar.

The content of the file `parameter.v` *is:*

```
/*************************************************************************
File name:      parameter.v
Circuit name:   Parameter module
Description:    defines the parameter used in designing the inner product
                circuit
*************************************************************************/
 parameter n = 8
```

The simulation is done by running the module:

```
/*****************************************************************
File name:      test_inner_prod.v
Circuit name:   Simulator for Inner-Product
Description:    generate the simulation environment of the inner-product
                circuit
*****************************************************************/
 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};
            end
  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 6 :

- 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 limitation 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:

$$t_{inner\_product} = t_{multiplication} + t_{4\_number\_add}$$

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

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

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

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

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 $g(int\_out)$, in time:

$$t_f = t_{h\_1} + t_g.$$

A solution must be find 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:

$$f_{clock\_max} = \frac{1}{t_{reg} + t_f + t_{su}} = \frac{1}{t_{reg} + t_{h\_1} + t_g + t_{su}}$$

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:

$$T_{stream}(n) = \frac{1}{f_{clock}} \times (n+2) = (t_{reg} + t_{h\_1} + t_g + t_{su}) \times (n+2)$$

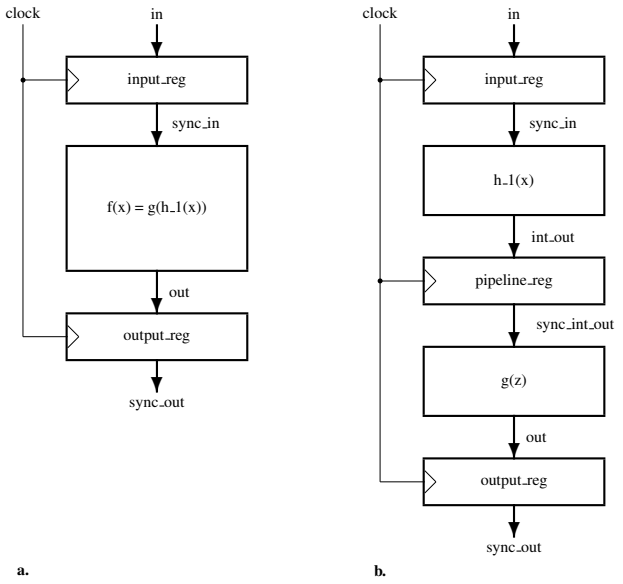


Figure 3.8: **Pipelined computation.** **a**. A typical Register Transfer Logic (RTL) configuration. Usually it is supposed a "deep" combinational circuit computes $f(x)$. **b**. The pipeline structure splits the combinational circuit associated with function $f(x)$ in two less "deep" circuits and inserts 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:

$$f_{clock\_max} = \frac{1}{max(t_{h\_1}, t_g) + t_{reg} + t_{su}}$$

and the processing time for the same string is:

$$T_{stream}(n) = (max(t_{h\_1}, t_g) + t_{reg} + t_{su}) \times (n+3)$$

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**.

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:

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

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

$$\lambda = m << n$$

In this ideal case the speed is increased almost m times. Obviously, no one of these condition 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:*

```
/*************************************************************************
File name:      pipelined_f.v
Circuit name:   Pseudo-Verilog modules for defining the pipeline mechanism
Description:    dummy modules pipeline connected
*************************************************************************/
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 = ...;
endmodule

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

Suppose, the unit time is $1ns$*. The maximum clock frequency for the pipeline version is:*

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

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

$$f_{clock} = \frac{1}{15+18+2} GHz = 28.57MHz$$

Adding only a simple register and accepting a minimal latency ($\lambda = 1$), the speed of the system increased with 75%. $\diamond$

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 $\lambda = m$.

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.*

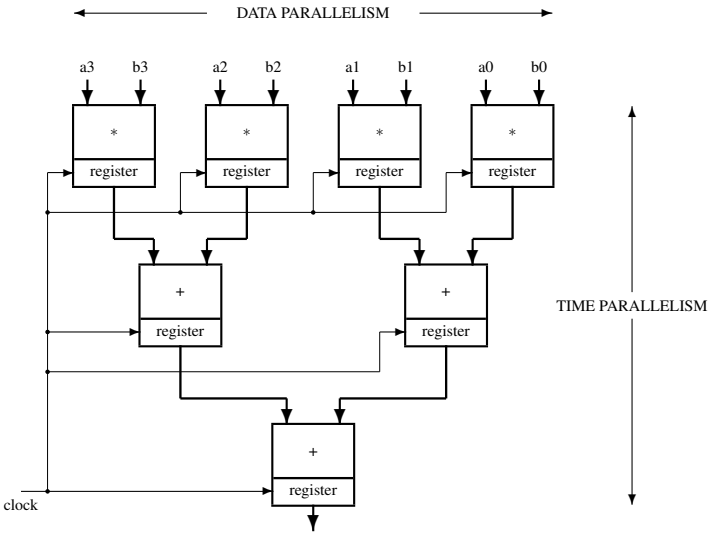


Figure 3.9: **The pipelined inner product circuit for 4-component vectors.** Each multiplier and each adder send its result in a pipeline register. For this application results a three level pipeline structure with different degree of parallelism. The two kind of parallelism are exemplified. Data parallel 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.

```
/*****************************************************************
File name:      pipelined_inner_prod.v
Circuit name:   Pipelined Inner Product circuit
Description:    the structural description of pipelined inner product
                circuit for 2 vectors of 4 8-bit numbers
*****************************************************************/
 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
```

```
/*****************************************************************
File name:      mult.v
Circuit name:   Pipelined 8-bit multiplier
Description:    the behavioral description of the pipelined multiplier
*****************************************************************/
 module mult(   output  reg [15:0]  out,
                input       [7:0]   m1, m0,
                input               clock);
    always @(posedge clock) out <= m1 * m0;
 endmodule
```

```
/*****************************************************************
File name:      add.v
Circuit name:   Pipelined adder
Description:    the behavioral description of the pipelined adder
*****************************************************************/
 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.

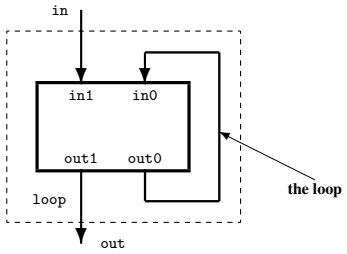


Figure 3.10: **The loop closed into a digital system.** The initial system has two inputs, `in1` and `in0`, and two outputs, `out1` and `out0`. Connecting `out0` to `in0` results a new system with `in` and `out` only.

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:*

```
/*************************************************************************
File name:      loop_system.v
Circuit name:   Loop System
Description:    pseudo-Verilog description of the loop closing in a system
*************************************************************************/
module 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.*

```
/*****************************************************************
File name:      acc.v
Circuit name:   Accumulator
Description:    the structural description of the accumulator circuit
*****************************************************************/
 module acc(output  [19:0]  out   ,
            input   [15:0]  in    ,
            input           clock, reset);
   sync_add our_add(out, in, out, clock, reset);
 endmodule
```

```
/*****************************************************************
File name:      sync_ad.v
Circuit name:   Synchronous Adder
Description:    the behavioral description of a synchronous adder
*****************************************************************/
 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
```

In order to make a simulation the next `test_acc` *module is written:*

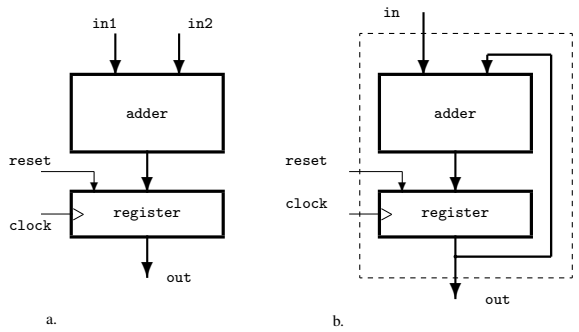


Figure 3.11: **Example of loop closed over an adder with synchronized output.** If the output becomes one of the inputs, results a circuit that accumulates at each clock cycle. **a**. The initial circuit: the synchronized adder. **b**. The resulting circuit: the accumulator.

```
/*********************************************************************
File name:      test_acc.v
Circuit name:   Testbench for the accumulator circuit
Description:    generates the stimulus for accumulator
*********************************************************************/
 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;
                    #2 reset = 0;
                    #10 $stop;
            end
   always @(posedge clock) if (reset)   in = 0;
                           else     in = in + 1;
   acc  dut(out, in, clock, reset);
   initial $monitor("time=%0d clock=%b in=%d out=%d",
                    $time, clock, in, dut.out);
 endmodule
```

By simulation results the following behavior:

```
/*************************************************************************
The output of the monitor
*************************************************************************/
# time=0  clock=0 in=    x out=      x
# time=1  clock=1 in=    0 out=      0
# time=2  clock=0 in=    0 out=      0
# time=3  clock=1 in=    1 out=      1
# time=4  clock=0 in=    1 out=      1
# time=5  clock=1 in=    2 out=      3
# time=6  clock=0 in=    2 out=      3
# time=7  clock=1 in=    3 out=      6
# time=8  clock=0 in=    3 out=      6
# time=9  clock=1 in=    4 out=     10
# time=10 clock=0 in=    4 out=     10
# time=11 clock=1 in=    5 out=     15
```

◇

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 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.

3.6 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.*

```
module  topModule(  input       [3:0]   in1     ,
                    input       [3:0]   in2     ,
                    input       [3:0]   in3     ,
                    output  reg [5:0]   out     ,   // propagation time: 50 ps
                                                    // hold time 15: ps
                                                    // set-up time: 20 ps
                    input               clock   );
    reg [4:0] reg1, reg2;                           // propagation time: 50 ps
                                                    // hold time 15: ps
                                                    // set-up time: 20 ps
    wire[4:0] w1, w2    ;
    wire[5:0] w3        ;

    always @(posedge clock) begin   reg1 <= w1  ;
                                    reg2 <= w2  ;
                                    out  <= w3  ;
                            end

    clc1 c1(.inA    (in1    ),
            .inB    (in2    ),
            .out    (w1     ));
    clc2 c2(.inA    (w1[3:0]),
            .inB    (in3    ),
            .out    (w2     ));
    clc3 c3(.inA    (reg1   ),
            .inB    (reg2   ),
            .out    (w3     ));
endmodule

module clc1(input [3:0] inA     ,
            input [3:0] inB ,
            output[4:0] out );
    // timpul de propagare inA2out = 200ps
    // timpul de propagare inB2out = 150ps
    // ...
endmodule

module clc2(input [3:0] inA     ,
            input [3:0] inB ,
            output[4:0] out );
    // timpul de propagare inA2out = 100ps
    // timpul de propagare inB2out = 250ps
    // ...
endmodule

module clc3(input [4:0] inA     ,
            input [4:0] inB ,
            output[5:0] out );
    // timpul de propagare inA2out = 400ps
    // timpul de propagare inB2out = 150ps
    // ...
endmodule
```

3.7 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, takeing 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[1]

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,* `in`*,* `lp`*, and one output,* `out`*. What hapend when is closed the loop from the output* `out` *to the input* `lp`*? Let's make it. The following Verilog modules describe the behavior of the resulting circuit.*

```
/*****************************************************************
File name:      loopSyst.v
Circuit name:   Loop System
Description:    the way a loop increase autonomy of the system
*****************************************************************/
 module loopSyst(    output  [1:0]   out,
                     input           in);
    initSyst noLoopSyst(.out(out), .in(in), .loop(out));
 endmodule
```

[1] From [Chaitin '06]

```
/*****************************************************************
File name:      initSyst.v
Circuit name:   No-Loop System
Description:    describe the no-loop system
*****************************************************************/
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;
                            endcase
endmodule
```

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.*

```
/*****************************************************************
File name:      test.v
Circuit name:   Testing Loop-System
Description:    generates the stimulus for testing the loop-system
*****************************************************************/
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:

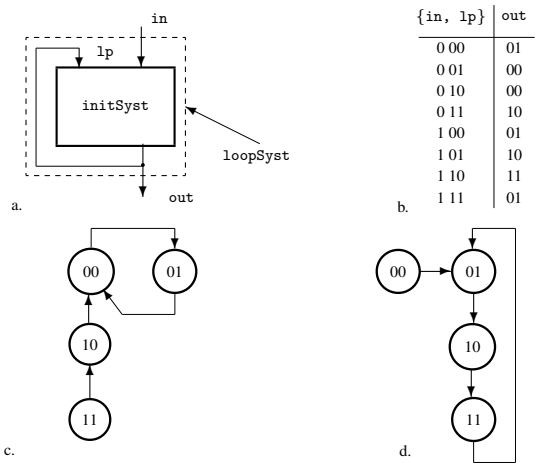


Figure 4.1: **Example illustrating the autonomy. a.** A system obtained from an initial system in which a loop is closed from output to one of its input. **b.** The transition table of the initial system where each output strict corresponds to the input value. **c.** The output evolution for constant input: `in = 0`. **d.** The output evolution for a different constant input: `in = 1`.

```
/*************************************************************************
The monitor output
*************************************************************************/
    # 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
    # time=8  in=0 out=00
    # time=9  in=0 out=01
    # time=10 in=1 out=00
    # time=11 in=1 out=01
    # time=12 in=1 out=10
    # time=13 in=1 out=11
    # time=14 in=1 out=01
    # time=15 in=1 out=10
    # time=16 in=1 out=11
    # time=17 in=1 out=01
    # time=18 in=1 out=10
    # time=19 in=1 out=11
```

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 `in = 0` *and Figure 4.1d represents the evolution for* `in = 1`. ◇

VerilogSummary 7 :

- the register `reg[1:0] out` defined in the module `initSyst` is nor a register, it is a Verilog variable, whose value is computed by a `case` procedure anytime at least one of the two inputs change (`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 or b 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 (`(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 `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:

```
always @(in1 or in0) #1
   case (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
```

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:

```
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.* ◇

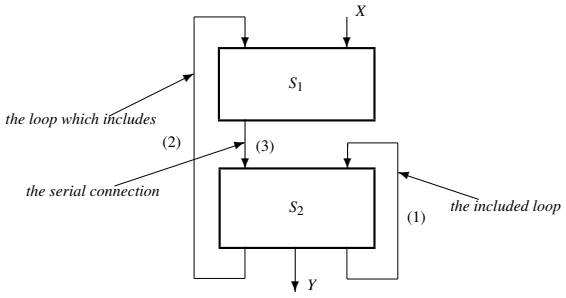


Figure 4.2: **Included loops.** The loop (2) includes loop (1), closed over the subsystem S_2, because S_2 is serially connected with the subsystem S_1 and loop (2) includes both S_1 and S_2.

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:

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.

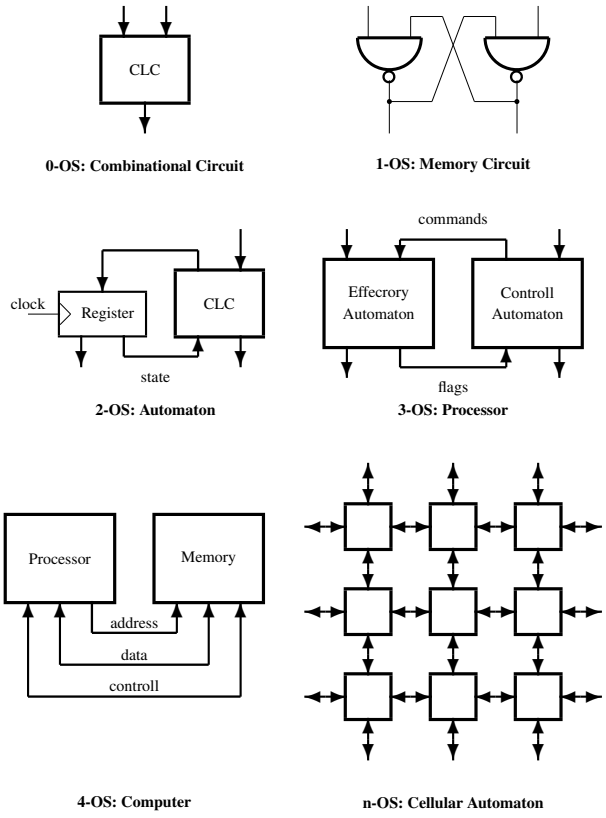


Figure 4.3: **Examples of circuits belonging to different orders.** A combinational circuit is in 0-OS class because has no loops. A memory circuit contains one-loop circuits and therefore it is in 1-OS class. Because the register belongs to 1-OS class, closing a loop containing a register and a combinational 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, ...
- 1-OS: elementary latch, master-slave flip-flop (serial composition), random access memory (parallel composition), register (serial-parallel composition), ...

[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.

- 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 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[3] serpent bits its own tail something very special must be expected.

[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.

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, totaly 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.

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.

4.4 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 Fibbonaci number starting from zero, until the biggest Fibbonaci 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 Fibbonaci 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 Fibbonaci number.*

4.5 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 [1]

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, f needed, `dataMemory` stores intermediate data or support data. The result is issued as the data stream `outputStream[15:0]`.

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 Witgenstein'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.

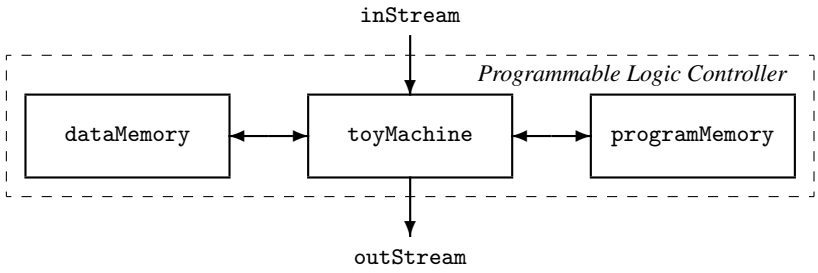


Figure 5.1: **Programmable Logic Controller designed with *toyMachine*.**

`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 ort 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:

  `inStream` : the input stream of data

  `outStream` : the output stream of data

  `progAddr` : the address for the programm 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

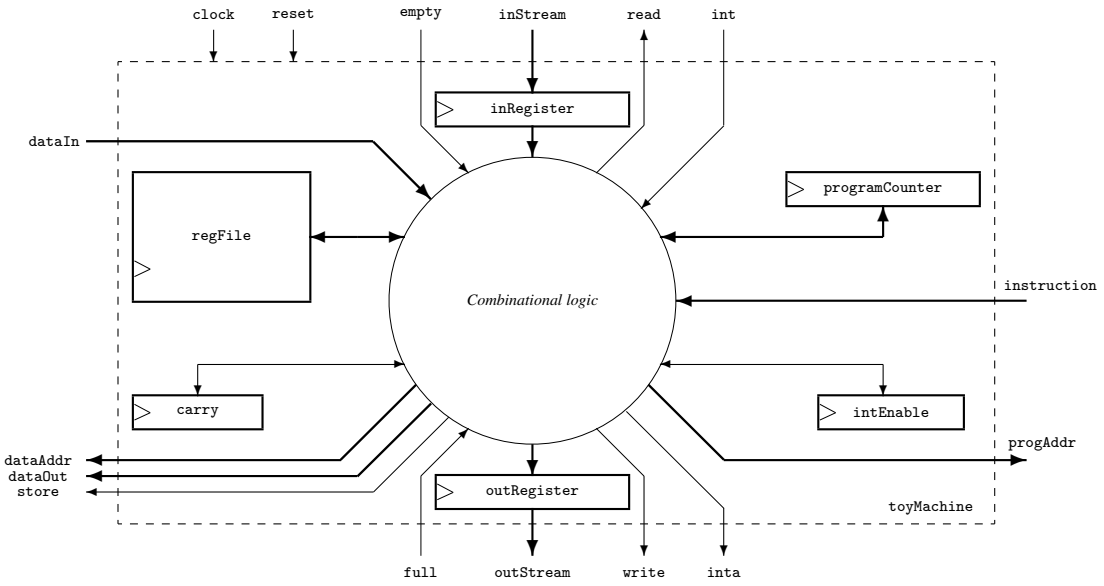


Figure 5.2: **The internal state of *toyMachine*.**

`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 `outStram` 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:

```
instruction = {opCode[5:0], destAddr[4:0], leftAddr[4:0], rightAddr[4:0], 11'b0}  |
              {opCode[5:0], destAddr[4:0], leftAddr[4:0], 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.

```
/*************************************************************************
File name:      toyMachineArchitecture.v.v
Circuit name:   Instruction Set Architecture
Description:    defines the binary form of the instruction set
*************************************************************************/
    parameter
        nop     = 6'b000000,    // no operation: increment programCounter
                                // CONTROL INSTRUCTIONS
        jmp     = 6'b000001,    // programCounter loaded form a register
        zjmp    = 6'b000010,    // jump if the selected register is 0
        nzjmp   = 6'b000011,    // jump if the selected register is not 0
        rjmp    = 6'b000100,    // relative jump: pc = pc + immVal
        ei      = 6'b000110,    // enable interrupt
        di      = 6'b000111,    // disable interrupt
        halt    = 6'b001000,    // programCounter does not change
                                // DATA INSTRUCTIONS: pc = pc + 1
                                // Arithmetic & logic instructions
        neg     = 6'b010000,    // bitwise not
        bwand   = 6'b010001,    // bitwise and
        bwor    = 6'b010010,    // bitwise or
        bwxor   = 6'b010011,    // bitwise exclusive or
        add     = 6'b010100,    // add
        sub     = 6'b010101,    // subtract
        addc    = 6'b010110,    // add with carry
        subc    = 6'b010111,    // subtract with carry
        move    = 6'b011000,    // move
        ashr    = 6'b011001,    // arithmetic shift right one position
        val     = 6'b011010,    // load immediate with sign extension
        hval    = 6'b011011,    // append immediate on high positions
                                // Input output instructions
        receive = 6'b100000,    // load inRegister if empty = 0
        issue   = 6'b100001,    // send outRegister if full = 0
        get     = 6'b100010,    // load in file register the inRegister
        send    = 6'b100011,    // load outRegister register's content
        datard  = 6'b100100,    // read from data memory
        datawr  = 6'b100101;    // write to data memory
```

Figure 5.3: ***toyMachine*'s ISA defined by the file `0_toyMachineArchitecture.v`.**

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. 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 acknowledged, the interrupt is disabled, letting the program decide when another interrupt is welcomed. The interrupt is disabled by executing the instruction `di` – disable interrupt.

The program flow is controlled by unconditioned and conditioned jump instructions. But, the `inta` signal once activated, has priority, allowing the load of the program counter with the value stored in `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, `programCounter`, is by default incremented with 1, but when a control instruction is executed its value can be incremented with the signed integer `instruction[15:0]` or set to the value of a register contained in the register file. The program control instructions are:

```
/*************************************************************************
File name:      toyMachine.v
Circuit name:   Toy Machine
Description:    the top level of the processor Toy Machine
*************************************************************************/
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 full
        output             write         , // write form outRegister
        input              interrupt     , // interrupt input
        output             inta          , // interrupt acknowledge
        output      [31:0] dataAddr      , // address for data memory
        output      [31:0] dataOut       , // data for data memory
        output             store         , // store dataOut at dataAddr
        input       [31:0] dataIn        , // data from data memory
        output reg  [31:0] programCounter, // address for program memory
        input       [31:0] instruction   , // instruction from memory
        input              reset         , // reset input
        input              clock         );// clock input // 2429 LUTs
    // 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.**

```
/*************************************************************************
File name:      instructionStructure.v
Circuit name:   Instruction Structure
Description:    file used to detail the instruction's structure
*************************************************************************/
    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`**.**

jmp : absolute jump with the value selected from the register file by the field `leftAddr`; the register `programCounter` takes the value contained in the selected register

zjmp : relative jump with the signed value `immValue` if the content of the register selected by `leftAddr` from the register file is 0, else `programCounter = programCounter + 1`

nzjmp : relative jump with the signed value `immValue` if the content of the register selected by `leftAddr` from the register file is not 0, else `programCounter = programCounter + 1`

receive : relative jump with the signed value `immValue` if `readyIn` is 1, else `programCounter = programCounter + 1`

issue : relative jump with the signed value `immValue` if `readyOut` is 1, else `programCounter = programCounter + 1`

halt : the program execution halts, `programCounter = programCounter` (it is a sort of `nop` instruction without incrementing the register `programCounter = programCounter`).

Warning! If `intEnable = 0` when the instruction `halt` is executed, then the overall system is blocked. The only way to turn it back to life is to activate the `reset` signal.

The file `dataFunction.v` (see Figure 5.7) describes the behavior of the data instructions. The signal `inta` has the highest priority. It forces the register 30 of the register file to store the current state of the register `programCounter`. It will be used to continue the program, interrupted by the acknowledged interrupt signal `int`, by executing a `jmp` instruction with the content of `regFile[30]`.

The following data instructions are described in this file:

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

```
/*********************************************************************
File name:      controlFunction.v
Circuit name:   Control Function
Description:    describes the control function of the processor
*********************************************************************/
/*********************************************************************
File name:      controlFunction.v
Circuit name:   Control Function
Description:    describes the control function of the processor
*********************************************************************/
    assign inta = intEnable & interrupt;
    always @(posedge clock)
        if (reset)                          intEnable <= 0  ;
         else if (inta)                     intEnable <= 0  ;
               else if (opCode == ei)       intEnable <= 1  ;
                     else if (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`**.**

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

addc : 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`

neg : every bit contained in the register selected by `leftAddr` are inverted and the result is stored in the register selected by `destAddr`

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 varue 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 th elocatioin `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]`

```
/*********************************************************************
File name:      dataFunction.v.v
Circuit name:   Data Function
Description:    describes the data functions of the processor
*********************************************************************/
    always @(posedge clock)
     if (inta)     regFile[30]   <= programCounter                    ;
      else
       case(opCode)
        add     : {carry, regFile[destAddr]}
                    <= regFile[leftAddr] + regFile[rightAddr]        ;
        sub     : {carry, regFile[destAddr]}
                    <= regFile[leftAddr] - regFile[rightAddr]        ;
        addc    : {carry, regFile[destAddr]}
                    <= regFile[leftAddr] + regFile[rightAddr] + carry ;
        subc    : {carry, regFile[destAddr]}
                    <= regFile[leftAddr] - regFile[rightAddr] - carry ;
        move    : regFile[destAddr] <= regFile[leftAddr]             ;
        ashr    : regFile[destAddr]
                    <= {regFile[leftAddr][31], regFile[leftAddr][31:1]};
        neg     : regFile[destAddr] <= ~regFile[leftAddr]            ;
        bwand   : regFile[destAddr]
                    <= regFile[leftAddr] & regFile[rightAddr]        ;
        bwor    : regFile[destAddr] <=
                     regFile[leftAddr] | regFile[rightAddr]          ;
        bwxor   : regFile[destAddr]
                    <= regFile[leftAddr] ^ regFile[rightAddr]        ;
        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

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 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 *toyMachine* 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.

[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.

```
/*****************************************************************
File name:      toyMachineCodeGenerator.v
Circuit name:   Simple assembler for toy Machine
Description:    generates the binary code in program memory; only typical
                instructions are assembled
*****************************************************************/
// 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
    // LB task sets labelTab in the first pass associating 'counter'
    // with 'labelIndex'
    task LB ;
        input [4:0] labelIndex;
        labelTab[labelIndex] = addrCounter;
    endtask
    // ULB task uses the content of labelTab in the second pass
    task ULB;
        input [4:0] labelIndex;
        {rightAddr, value} = labelTab[labelIndex] - addrCounter;
    endtask

    task NOP;
        begin   opCode              = nop   ;
                destAddr            = 5'b0  ;
                leftAddr            = 5'b0  ;
                {rightAddr, value}  = 16'b0 ;
                endLine                     ;
        end
    endtask

    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
    endtask
    // ...
```

```
task ADD;
    input   [4:0]   dest    ;
    input   [4:0]   left    ;
    input   [4:0]   right   ;
    begin   opCode              = add           ;
            destAddr            = dest          ;
            leftAddr            = left          ;
            {rightAddr, value}  = {right, 11'b0};
            endLine                             ;
    end
endtask
// ...
task ADDC;
    input   [4:0]   dest    ;
    input   [4:0]   left    ;
    input   [4:0]   right   ;
    begin   opCode              = addc          ;
            destAddr            = dest          ;
            leftAddr            = left          ;
            {rightAddr, value}  = {right, 11'b0};
            endLine                             ;
    end
endtask
// ...
task VAL;
    input   [4:0]   dest    ;
    input   [15:0]  immVal  ;
    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
// ...
task DATARD;
    input   [4:0]   dest;
    begin   opCode              = datard;
            destAddr            = dest  ;
            leftAddr            = 5'b0  ;
            {rightAddr, value}  = 16'b0 ;
```

```
            endLine                         ;
    end
endtask
// ...
// RUNNING
initial begin    addrCounter = 0;
                 `include "0_theProgram.v"; // first pass
                 addrCounter = 0;
                 `include "0_theProgram.v"; // second pass
        end
```

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 fists 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(m,n)` : 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 regiter, `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 `toyMachineSimlator.v` is in the folder pointed by `http://arh.pub.ro/gstefan/toyMachine.zip`, while, in the following, a little edited form is presented.

```
/*************************************************************************
File name:      toyMachineSimulator.v
Circuit name:   Simulator module for Toy Machine
Description:    simulate a system with Toy Machine
*************************************************************************/
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 from 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          ;
            end

    always @(posedge clock)
        if (reset)  inStream[2:0] <= 0                              ;
         else       inStream[2:0] <= readIn ? inStream[2:0] + 1 :
                                             inStream[2:0]      ;

    toySystem dut(  inStream    ,
                    empty       ,
                    read        ,
                    outStream   ,
                    full        ,
                    write       ,
                    interrupt   ,
                    inta        ,
                    reset       ,
                    clock       );

    initial
     $monitor("time=%0d \t rst=%b pc=%0d ...", // !!!
                    $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:

```
/*****************************************************************
File name:      toySystem.v
Circuit name:   Toy System
Description:    a system build around Toy Machine
*****************************************************************/
module toySystem
       (input  [15:0] inStream   , // input stream of data
        input         empty      , // input stream is ready
        output        read       , // read one element from the stream
        output [15:0] outStream  , // output stream of data
        input         full       , // ready to receive from output stream
        output        write      , // write the element form outRegister
        input         interrupt  , // interrupt input
        output        inta       , // interrupt acknowledge
        input         reset      , // reset input
        input         clock      );// clock input

    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 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`*.*

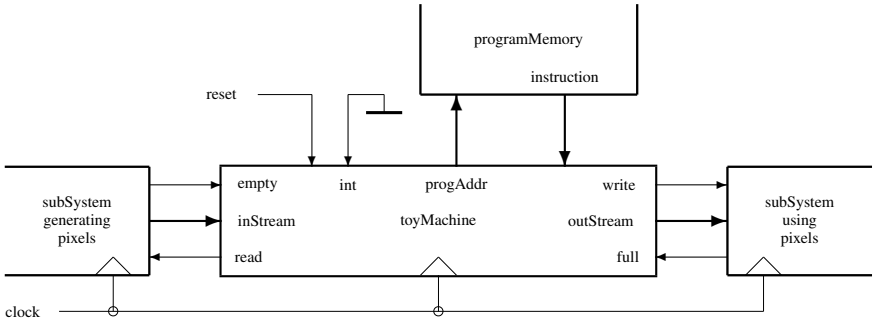


Figure 5.8: **Programmed logic implementation for the `interpol` circuit.**

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-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*

[4]The true *macros* are used in assembly languages. For this level of the machine language a more rudimentary form of macro is used.

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`.

```
                        // initializing the never ending loop
        RECEIVE;        // load inRegister if (empty = 0) else wait
        GET(0);         // regFile[0] <= inRegister
        RECEIVE;        // load inRegister 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;        // load inRegister 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 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)`*).*

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, compute the value to be sent out and sends it.

The behavior of the system, provided by the simulator, is:

```
progMem[0]  = 10000000000000000000000000000000
progMem[1]  = 10001000000000000000000000000000
progMem[2]  = 10000000000000000000000000000000
progMem[3]  = 10001000001000000000000000000000
progMem[4]  = 01100000010000010000000000000000
progMem[5]  = 01100000001000000000000000000000
progMem[6]  = 10000000000000000000000000000000
progMem[7]  = 10001000000000000000000000000000
progMem[8]  = 00001000000000010000000000000110
progMem[9]  = 00000000000000000000000000000000
progMem[10] = 00000000000000000000000000000000
progMem[11] = 10001100000000010000000000000000
progMem[12] = 10000100000000000000000000000000
progMem[13] = 00010000000000001111111111110111
progMem[14] = 01010000001000000001000000000000
progMem[15] = 01100100001000010000000000000000
progMem[16] = 10001100000000010000000000000000
progMem[17] = 10000100000000000000000000000000
progMem[18] = 00010000000000001111111111110010
```

Figure 5.10: **The binary form of the program for `interpol`.**

```
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]=2 rf[1]=0 rf[2]=1 read=0 inReg=2 write=0 outReg=x
time=19  rst=0 pc=9  cr=x rf[0]=2 rf[1]=0 rf[2]=1 read=0 inReg=2 write=0 outReg=x
time=21  rst=0 pc=10 cr=0 rf[0]=2 rf[1]=3 rf[2]=1 read=0 inReg=2 write=0 outReg=x
time=23  rst=0 pc=11 cr=0 rf[0]=2 rf[1]=1 rf[2]=1 read=0 inReg=2 write=0 outReg=x
time=25  rst=0 pc=12 cr=0 rf[0]=2 rf[1]=1 rf[2]=1 read=0 inReg=2 write=1 outReg=1
time=27  rst=0 pc=13 cr=0 rf[0]=2 rf[1]=1 rf[2]=1 read=0 inReg=2 write=0 outReg=1
time=29  rst=0 pc=4  cr=0 rf[0]=2 rf[1]=1 rf[2]=1 read=0 inReg=2 write=0 outReg=1
time=31  rst=0 pc=5  cr=0 rf[0]=2 rf[1]=1 rf[2]=1 read=0 inReg=2 write=0 outReg=1
time=33  rst=0 pc=6  cr=0 rf[0]=2 rf[1]=2 rf[2]=1 read=1 inReg=2 write=0 outReg=1
time=35  rst=0 pc=7  cr=0 rf[0]=2 rf[1]=2 rf[2]=1 read=0 inReg=3 write=0 outReg=1
time=37  rst=0 pc=8  cr=0 rf[0]=3 rf[1]=2 rf[2]=1 read=0 inReg=3 write=0 outReg=1
time=39  rst=0 pc=11 cr=0 rf[0]=3 rf[1]=2 rf[2]=1 read=0 inReg=3 write=0 outReg=1
time=41  rst=0 pc=12 cr=0 rf[0]=3 rf[1]=2 rf[2]=1 read=0 inReg=3 write=1 outReg=2
time=43  rst=0 pc=13 cr=0 rf[0]=3 rf[1]=2 rf[2]=1 read=0 inReg=3 write=0 outReg=2
time=45  rst=0 pc=4  cr=0 rf[0]=3 rf[1]=2 rf[2]=1 read=0 inReg=3 write=0 outReg=2
time=47  rst=0 pc=5  cr=0 rf[0]=3 rf[1]=2 rf[2]=2 read=0 inReg=3 write=0 outReg=2
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=1 outReg=3
```

```
time=59  rst=0 pc=13 cr=0 rf[0]=4 rf[1]=3 rf[2]=2 read=0 inReg=4 write=0 outReg=3
time=61  rst=0 pc=4  cr=0 rf[0]=4 rf[1]=3 rf[2]=2 read=0 inReg=4 write=0 outReg=3
time=63  rst=0 pc=5  cr=0 rf[0]=4 rf[1]=3 rf[2]=3 read=0 inReg=4 write=0 outReg=3
time=65  rst=0 pc=6  cr=0 rf[0]=4 rf[1]=4 rf[2]=3 read=1 inReg=4 write=0 outReg=3
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]=5 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]=4 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
time=109 rst=0 pc=4  cr=0 rf[0]=7 rf[1]=6 rf[2]=5 read=0 inReg=7 write=0 outReg=6
time=111 rst=0 pc=5  cr=0 rf[0]=7 rf[1]=6 rf[2]=6 read=0 inReg=7 write=0 outReg=6
time=113 rst=0 pc=6  cr=0 rf[0]=7 rf[1]=7 rf[2]=6 read=1 inReg=7 write=0 outReg=6
time=115 rst=0 pc=7  cr=0 rf[0]=7 rf[1]=7 rf[2]=6 read=0 inReg=0 write=0 outReg=6
time=117 rst=0 pc=8  cr=0 rf[0]=0 rf[1]=7 rf[2]=6 read=0 inReg=0 write=0 outReg=6
time=119 rst=0 pc=11 cr=0 rf[0]=0 rf[1]=7 rf[2]=6 read=0 inReg=0 write=0 outReg=6
time=121 rst=0 pc=12 cr=0 rf[0]=0 rf[1]=7 rf[2]=6 read=0 inReg=0 write=1 outReg=7
time=123 rst=0 pc=13 cr=0 rf[0]=0 rf[1]=7 rf[2]=6 read=0 inReg=0 write=0 outReg=7
time=125 rst=0 pc=4  cr=0 rf[0]=0 rf[1]=7 rf[2]=6 read=0 inReg=0 write=0 outReg=7
time=127 rst=0 pc=5  cr=0 rf[0]=0 rf[1]=7 rf[2]=7 read=0 inReg=0 write=0 outReg=7
time=129 rst=0 pc=6  cr=0 rf[0]=0 rf[1]=0 rf[2]=7 read=1 inReg=0 write=0 outReg=7
time=131 rst=0 pc=7  cr=0 rf[0]=0 rf[1]=0 rf[2]=7 read=0 inReg=1 write=0 outReg=7
time=133 rst=0 pc=8  cr=0 rf[0]=1 rf[1]=0 rf[2]=7 read=0 inReg=1 write=0 outReg=7
time=135 rst=0 pc=9  cr=0 rf[0]=1 rf[1]=0 rf[2]=7 read=0 inReg=1 write=0 outReg=7
time=137 rst=0 pc=10 cr=0 rf[0]=1 rf[1]=8 rf[2]=7 read=0 inReg=1 write=0 outReg=7
time=139 rst=0 pc=11 cr=0 rf[0]=1 rf[1]=4 rf[2]=7 read=0 inReg=1 write=0 outReg=7
time=141 rst=0 pc=12 cr=0 rf[0]=1 rf[1]=4 rf[2]=7 read=0 inReg=1 write=1 outReg=4
time=143 rst=0 pc=13 cr=0 rf[0]=1 rf[1]=4 rf[2]=7 read=0 inReg=1 write=0 outReg=4
time=145 rst=0 pc=4  cr=0 rf[0]=1 rf[1]=4 rf[2]=7 read=0 inReg=1 write=0 outReg=4
time=147 rst=0 pc=5  cr=0 rf[0]=1 rf[1]=4 rf[2]=4 read=0 inReg=1 write=0 outReg=4
time=149 rst=0 pc=6  cr=0 rf[0]=1 rf[1]=1 rf[2]=4 read=1 inReg=1 write=0 outReg=4
time=151 rst=0 pc=7  cr=0 rf[0]=1 rf[1]=1 rf[2]=4 read=0 inReg=2 write=0 outReg=4
time=153 rst=0 pc=8  cr=0 rf[0]=2 rf[1]=1 rf[2]=4 read=0 inReg=2 write=0 outReg=4
time=155 rst=0 pc=11 cr=0 rf[0]=2 rf[1]=1 rf[2]=4 read=0 inReg=2 write=0 outReg=4
time=157 rst=0 pc=12 cr=0 rf[0]=2 rf[1]=1 rf[2]=4 read=0 inReg=2 write=1 outReg=1
time=159 rst=0 pc=13 cr=0 rf[0]=2 rf[1]=1 rf[2]=4 read=0 inReg=2 write=0 outReg=1
time=161 rst=0 pc=4  cr=0 rf[0]=2 rf[1]=1 rf[2]=4 read=0 inReg=2 write=0 outReg=1
time=163 rst=0 pc=5  cr=0 rf[0]=2 rf[1]=1 rf[2]=1 read=0 inReg=2 write=0 outReg=1
time=165 rst=0 pc=6  cr=0 rf[0]=2 rf[1]=2 rf[2]=1 read=1 inReg=2 write=0 outReg=1
time=167 rst=0 pc=7  cr=0 rf[0]=2 rf[1]=2 rf[2]=1 read=0 inReg=3 write=0 outReg=1
time=169 rst=0 pc=8  cr=0 rf[0]=3 rf[1]=2 rf[2]=1 read=0 inReg=3 write=0 outReg=1
time=171 rst=0 pc=11 cr=0 rf[0]=3 rf[1]=2 rf[2]=1 read=0 inReg=3 write=0 outReg=1
time=173 rst=0 pc=12 cr=0 rf[0]=3 rf[1]=2 rf[2]=1 read=0 inReg=3 write=1 outReg=2
time=175 rst=0 pc=13 cr=0 rf[0]=3 rf[1]=2 rf[2]=1 read=0 inReg=3 write=0 outReg=2
time=177 rst=0 pc=4  cr=0 rf[0]=3 rf[1]=2 rf[2]=1 read=0 inReg=3 write=0 outReg=2
time=179 rst=0 pc=5  cr=0 rf[0]=3 rf[1]=2 rf[2]=2 read=0 inReg=3 write=0 outReg=2
time=181 rst=0 pc=6  cr=0 rf[0]=3 rf[1]=3 rf[2]=2 read=1 inReg=3 write=0 outReg=2
time=183 rst=0 pc=7  cr=0 rf[0]=3 rf[1]=3 rf[2]=2 read=0 inReg=4 write=0 outReg=2
```

```
time=185 rst=0 pc=8  cr=0 rf[0]=4 rf[1]=3 rf[2]=2 read=0 inReg=4 write=0 outReg=2
time=187 rst=0 pc=11 cr=0 rf[0]=4 rf[1]=3 rf[2]=2 read=0 inReg=4 write=0 outReg=2
time=189 rst=0 pc=12 cr=0 rf[0]=4 rf[1]=3 rf[2]=2 read=0 inReg=4 write=1 outReg=3
time=191 rst=0 pc=13 cr=0 rf[0]=4 rf[1]=3 rf[2]=2 read=0 inReg=4 write=0 outReg=3
time=193 rst=0 pc=4  cr=0 rf[0]=4 rf[1]=3 rf[2]=2 read=0 inReg=4 write=0 outReg=3
time=195 rst=0 pc=5  cr=0 rf[0]=4 rf[1]=3 rf[2]=3 read=0 inReg=4 write=0 outReg=3
time=197 rst=0 pc=6  cr=0 rf[0]=4 rf[1]=4 rf[2]=3 read=1 inReg=4 write=0 outReg=3
time=199 rst=0 pc=7  cr=0 rf[0]=4 rf[1]=4 rf[2]=3 read=0 inReg=5 write=0 outReg=3
time=201 rst=0 pc=8  cr=0 rf[0]=5 rf[1]=4 rf[2]=3 read=0 inReg=5 write=0 outReg=3
time=203 rst=0 pc=11 cr=0 rf[0]=5 rf[1]=4 rf[2]=3 read=0 inReg=5 write=0 outReg=3
```

◇

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 catched 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 (synchronic) 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[1]
>
> *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).

[1] Jaron Lanier coined the term *virtual reality*. He is a computer scientist and a musician.

[2] The Moore's Law says the physical performances in microelectronics improve exponentially in time.

One fact is clear: we can not design very big complex circuits because we can not 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 *n* 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.

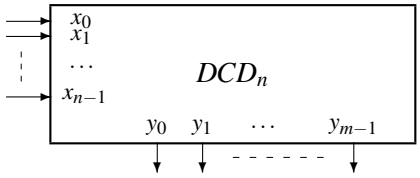


Figure 6.1: **The *n*-input decoder (DCD_n).**

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, $x_{n-1}, \ldots, x_0$, and the m-bit output Y, $y_{m-1}, \ldots, y_0$, where: $m = 2^n$, with the behavioral Verilog description:*

```
/*****************************************************************
File name:      dec.v
Circuit name:   Decoder
Description:    behavioral description of a n-input decoder
*****************************************************************/
 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 synthesisable 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: $f_2^1(x_0) = y_1 = x_0$ (identity function) and $f_1^1(x_0) = y_0 = x_0'$ (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.

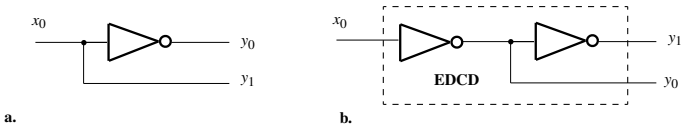


Figure 6.2: **The elementary decoder (EDCD)**. **a.** The basic circuit. **b.** Buffered EDCD, a serial-parallel composition.

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).* ◇

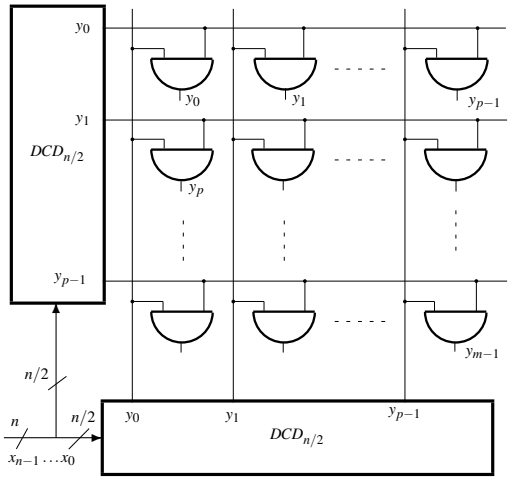


Figure 6.3: **The recursive definition of n-inputs decoder (DCD_n).** Two $DCD_{n/2}$ are used to drive a two dimension array of AND_2 gates. The same rule is applied for the two $DCD_{n/2}$, and so on until $DCD_1 = EDCD$ is needed.

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: $GS_{DCD}(n) = 2^n GS_{AND}(2) + 2GS_{DCD}(n/2) = 2(2^n + GS_{DCD}(n/2))$
$GS_{DCD}(1) = GS_{EDCD} = 2$
$GS_{DCD}(n) \in O(2^n)$

Depth: $D_{DCD}(n) = D_{AND}(2) + D_{DCD}(n/2) = 1 + D_{DCD}(n/2) \in O(log\ n)$
$D_{DCD}(1) = D_{EDCD} = 2$

Complexity: $C_{DCD} \in O(1)$ because the definition occupies a constant drown 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 takeing 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 as a binary tree of 2-input ANDs.

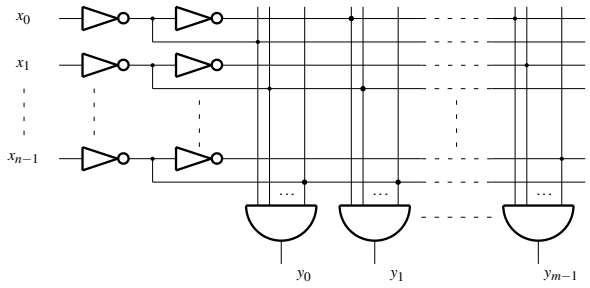


Figure 6.4: **"Constant depth" DCD** Applying the associative rule into the hierarchical network of AND_2 gates results the one level AND_n gates circuit driven by n EDCDs.

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:

```
/*****************************************************************
File name:      dec3.v
Circuit name:   3-input Decoder
Description:    structural description of a 3-input decoder
*****************************************************************/
module dec3(output [7:0] out,
            input  [2:0] in );
 // internal connections
    wire in0, nin0, in1, nin1, in2, nin2;
 // EDCD for in[0]
    not not00(nin0, in[0]), not01(in0, nin0)  ;
 // EDCD for in[1]
    not not10(nin1, in[1]), not11(in1, nin1)  ;
 // EDCD for in[2]
    not not20(nin2, in[2]), 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:

$$
\begin{aligned}
y_0 &= x'_{n-1} \cdot x'_{n-2} \cdot \ldots x'_1 \cdot x'_0 \\
y_1 &= x'_{n-1} \cdot x'_{n-2} \cdot \ldots x'_1 \cdot x_0 \\
y_2 &= x'_{n-1} \cdot x'_{n-2} \cdot \ldots x_1 \cdot x'_0 \\
&\ldots \\
y_{m-2} &= x_{n-1} \cdot x_{n-2} \cdot \ldots x_1 \cdot x'_0 \\
y_{m-1} &= x_{n-1} \cdot x_{n-2} \cdot \ldots x_1 \cdot x_0
\end{aligned}
$$

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.*

a	b	c	X	Y
0	0	0	0	1
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	1	0
1	1	0	1	1
1	1	1	0	0

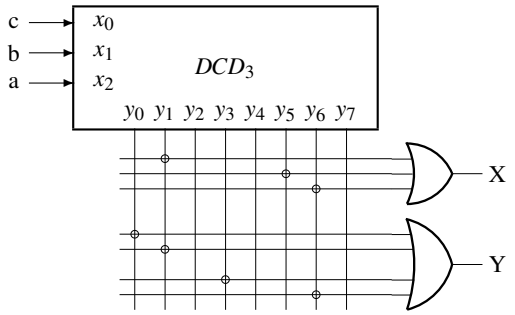


Figure 6.5:

Each min-term is computed only once, but it can be used as many times as the implemented functions suppose.

◇

6.1.2 Demultiplexors

The structure of the decoder is included in the structure of the other usual circuits. Two of them are the *demultiplexor* 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 demultiplexor 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 $O(2^n)$.

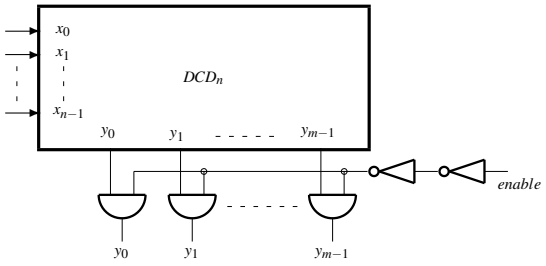


Figure 6.6: **Demultiplexor.** The n-input demultiplexor ($DMUX_n$) includes a DCD_n and 2^n AND_2 gates used to distribute the input *enable* in 2^n different places according to the n-bit selection code.

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.

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:*

```
/*****************************************************************
File name:      dmux.v
Circuit name:   Demultiplexor
Description:    behavioral description for a n-input demultiplexor
*****************************************************************/
 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.

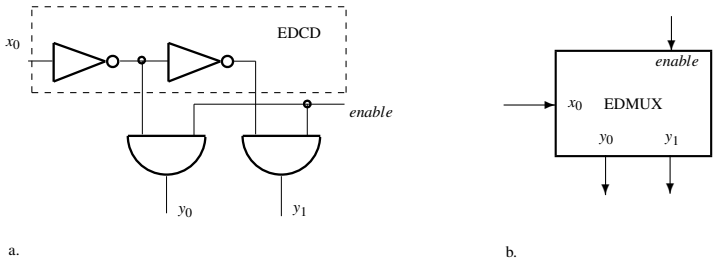


Figure 6.7: **The elementary demultiplexor. a.** The internal structure of an elementary demultiplexor (EDMUX) consists in an elementary decoder, 2 AND_2 gates, and an inverter circuit as input buffer. **b.** The logic symbol.

The same rule – *divide & impera* – is used to define an n-input demultiplexor, as follows:

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.

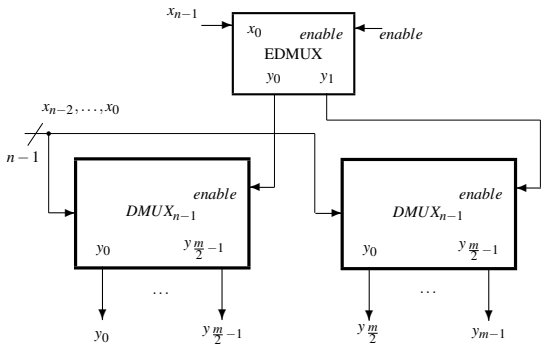


Figure 6.8: **The recursive definition of $DMUX_n$.** Applying the same rule for the two $DMUX_{n-1}$ a new level of 2 EDMUXs is added, and the output level is implemented using 4 $DMUX_{n-2}$. And so on until the output level is implemented using 2^{n-1} EDMUXs. The resulting circuit contains $2^n - 1$ EDMUXs.

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.

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:*

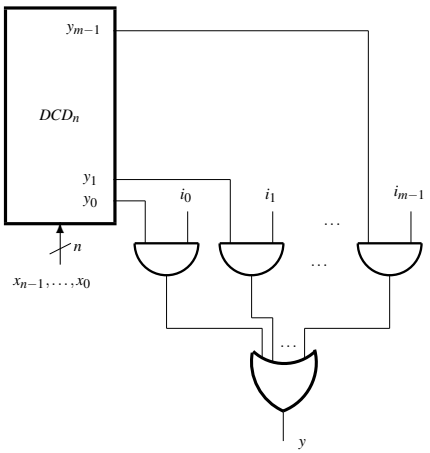


Figure 6.9: **Multiplexer.** The n selection inputs multiplexer MUX_n is made serial connecting a DCD_n with an AND-OR structure.

```
/*************************************************************************
File name:        mux.v
Circuit name:     Multiplexor
Description:      behavioral description for a n selection inputs
                  multiplexor
*************************************************************************/
 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 MUSs 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.

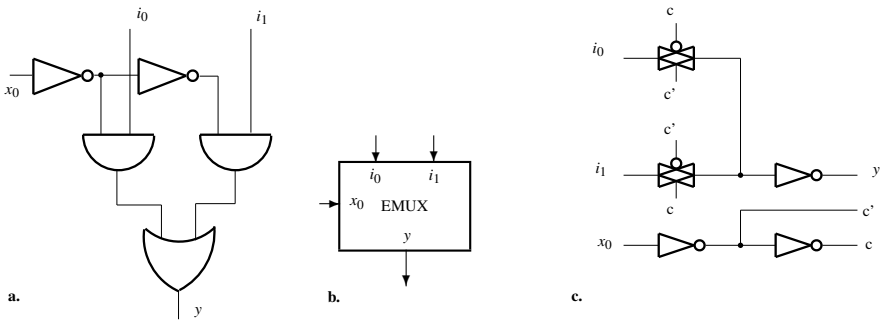


Figure 6.10: **The elementary multiplexer (EMUX). a.** The structure of EMUX containing an EDCD and the smallest AND-OR structure. **b.** The logic symbol of EMUX. **c.** A version of EMUX using transmission gates (see section *Basic circuits*).

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 $MUX_1 = EMUX$.* $\diamond$

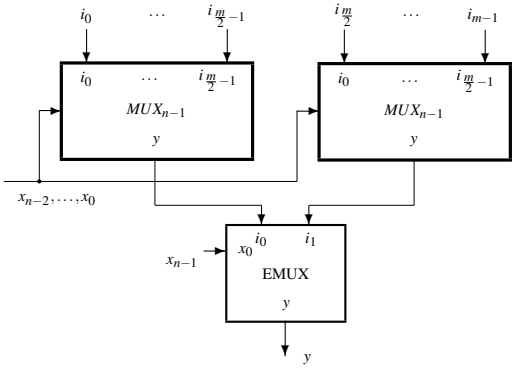


Figure 6.11: **The recursive definition of MUX_n.** Each MUX_{n-1} has a similar definition (two MUX_{n-2} and one EMUX), until the entire structure contains EMUXs. The resulting circuit is a binary tree of $2^n - 1$ EMUXs.

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

$$y = x_1(x_0 i_3 + x_0' i_2) + x_1'(x_0 i_1 + x_0' i_0)$$

that becomes

$$y = x_1 x_0 i_3 + x_1 x_0' i_2 + x_1' x_0 i_1 + x_1' x_0' i_0.$$

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:

$$y = x_{n-1}\ldots x_1x_0i_{m-1} + \ldots + x'_{n-1}\ldots x'_1x_0i_1 + x'_{n-1}\ldots x'_1x'_0i_0$$

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.*

a	b	c	X
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

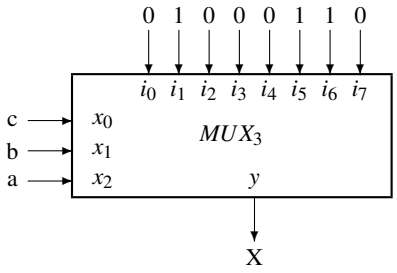


Figure 6.12:

◇

6.1.4 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 $PE(n)$.

```
/*****************************************************************
File name:      priority_encoder.v
Circuit name:   Priority Encoder
Description:    behavioral description of an 8-bit input priority encoder
*****************************************************************/
 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:

```
/*************************************************************************
File name:      .v
Circuit name:
Description:
*************************************************************************/
 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'b000001xx;
                        #1  in = 8'b00001xxx;
                        #1  in = 8'b0001xxxx;
                        #1  in = 8'b001xxxxx;
                        #1  in = 8'b01xxxxxx;
                        #1  in = 8'b1xxxxxxx;
                        #1  in = 8'b110;
                        #1  $stop;
                end
    priority_encoder    dut(in      ,
                            enable  ,
                            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:

```
    time = 0    enable = 0  in = 11111111   out = 000   zero = 1
    time = 1    enable = 1  in = 11111111   out = 111   zero = 0
    time = 2    enable = 1  in = 00000001   out = 000   zero = 0
    time = 3    enable = 1  in = 0000001x   out = 001   zero = 0
    time = 4    enable = 1  in = 000001xx   out = 010   zero = 0
    time = 5    enable = 1  in = 00001xxx   out = 011   zero = 0
    time = 6    enable = 1  in = 0001xxxx   out = 100   zero = 0
    time = 7    enable = 1  in = 001xxxxx   out = 101   zero = 0
    time = 8    enable = 1  in = 01xxxxxx   out = 110   zero = 0
    time = 9    enable = 1  in = 1xxxxxxx   out = 111   zero = 0
    time =10    enable = 1  in = 00000110   out = 010   zero = 0
```

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 unappropriate for computing the logarithm, and "prevent" us from takeing into account the output value.

6.1.5 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, *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 $inc = 1$, **then**

$$\{cr_{n-1}, y_{n-1},\ldots y_0\} = \{x_{n-1},\ldots x_0\} + 1$$

else

$$\{cr_{n-1}, y_{n-1},\ldots y_0\} = \{0, x_{n-1},\ldots x_0\}.$$

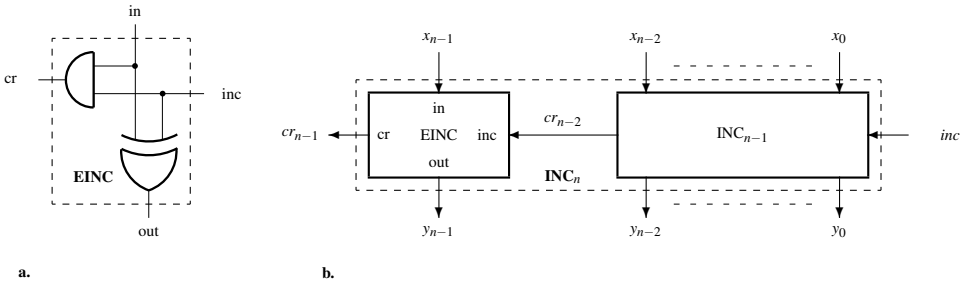


Figure 6.13: **Increment circuit. a.** The elementary increment circuit (called also **half adder**). **b.** The recursive definition for an n-bit increment circuit.

The increment circuit is built using as "brick" the **elementary increment circuit**, EINC, represented in Figure 6.13a, where the XOR circuit generate the increment of the input if $inc = 1$ (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, INC_n is recursively defined in Figure 6.13b: INC_n is composed using an INC_{n-1} serially connected with an EINC, where $INC_0 = EINC$.

6.1.6 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_2. An n-bit adder could be done in a recursive manner as the following definition says.

Definition 6.7 *The full adder, FA, is a circuit which adds three 1-bit numbers generating a 2-bit result:*

$$FA(in1, in2, in3) = \{out1, out0\}$$

FA is used to build n-bit adders. For this purpose its connections are interpreted as follows:

- *in*1, *in*2 *represent the i-th bits if two numbers*
- *in*3 *represents the carry signal generated by the* $i-1$ *stage of the addition process*
- *out*0 *represents the i-th bit of the result*
- *out*1 *represents the carry generated for the* $i+1$*-th stage of the addition process*

Follows the Verilog description:

```
/*************************************************************************
File name:      full_adder.v
Circuit name:   Full Adder
Description:    behavioral description of a full adder
*************************************************************************/
 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
```

```
/*************************************************************************
File name:      half_adder.v
Circuit name:   Half Adder
Description:    behavioral description of a half adder
*************************************************************************/
 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 6.13a).

Definition 6.8 *The n-bits ripple carry adder,* (ADD_n)*, is made by serial connecting on the carry chain an* ADD_{n-1} *with a FA (see Figure 6.14).* ADD_1 *is a full adder.*

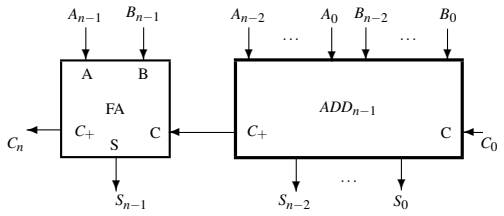


Figure 6.14: **The recursive defined n-bit ripple-carry adder (ADD_n).** ADD_n is simply designed adding to an ADD_{n-1} a full adder (FA), so as the carry signal ripples from one FA to the next.

```
/*****************************************************************
File name:      adder.v
Circuit name:   Adder
Description:    recursive structural description of a n-bit adder using
                the conditional generate statement
*****************************************************************/
 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
endmodule
```

```
/*************************************************************************
File name:      fullAdder.v
Circuit name:   Full Adder
Description:    behavioral description of a full adder
*************************************************************************/
module fullAdder(   output  out, cry ,
                    input   in1, in2, cin);
    assign  cry = in1 & in2 | (in1 ^ in2) & cin ;
    assign  out = in1 ^ in2 ^ cin               ;
endmodule
```

◇

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 6.3 *Generated n-bit adder:*

```
/*************************************************************************
File name:      add.v
Circuit name:   Adder
Description:    structural description of a n-input adder using the
                generate statemnt
*************************************************************************/
 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])); end
    endgenerate
 endmodule
```

[3]The use of the conditioned generation block for recursive definition was suggested to me by my colleague Radu Hobincu.

```
/*****************************************************************
File name:      fa.v
Circuit name:   Full Adder
Description:    structural description of a full adder
*****************************************************************/
 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:

```
/*****************************************************************
File name:      add.v
Circuit name:   Adder
Description:    behavioral description of an adder
*****************************************************************/
 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 6.15). The half adder has two roles in the structure:

- sums the bits A_i and B_i on the output S
- computes the signals G_i (that *generates* carry as a local effect) and P_i (that allows the *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 *carry-look-ahead circuit*. The equations describing it start from the next rule: *the carry toward the level*

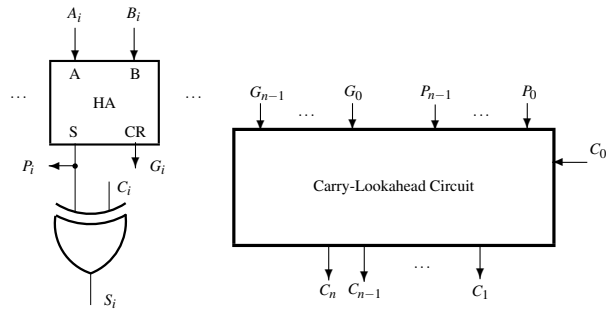


Figure 6.15: **The fast n-bit adder.** The n-bit Carry-Lookahead Adder (CLA_n) consists in n HAs, n 2-input XORs and the Carry-Lookahead Circuit used to compute faster the n C_i, for $i = 1,2,\ldots n$.

$(i+1)$ *is* **generated** *if both A_i and B_i inputs are 1* **or** *is* **propagated** *from the previous level if only one of A_i or B_i are 1*. Results:

$$C_{i+1} = A_iB_i + (A_i + B_i)C_i = A_iB_i + (A_i \oplus B_i)C_i = G_i + P_iC_i.$$

Applying the previous rule we obtain the general form of C_{i+1}:

$$C_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + P_iP_{i-1}P_{i-2}G_{i-3} + \ldots + P_iP_{i-1}\ldots P_1P_0C_0$$

for $i = 0,\ldots,n$.

Computing the size of the carry-look-ahead circuit results $S_{CL}(n) \in O(n^3)$, 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 *Basic circuits* appendix (see section *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 $D_{CLA} \in O(log\ n)$.

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 ADD_n is divided in two equal sections and two carry look-ahead circuits are built for each, resulting two serial connected $CLA_{n/2}$. 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.

6.1.7 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 6.4 *The 8-function ALU working on 32-bit numbers is described by the following Verilog module:*

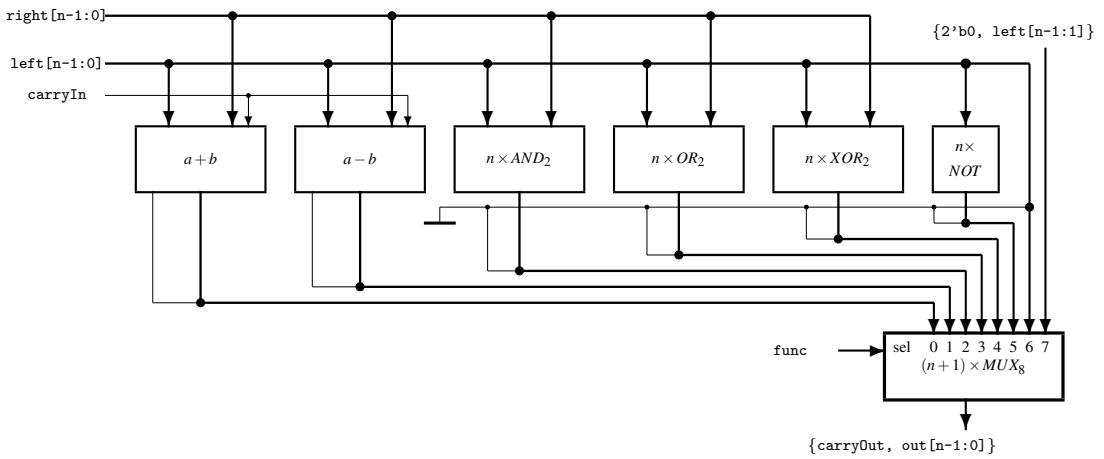


Figure 6.16: **The internal structure of the speculative version of an arithmetic and logic unit.** Each function is performed by a specific circuit and the output multiplexer selects the desired result.

```
/*************************************************************************
File name:      alu.v
Circuit name:   arithmetic and logic unit
Description:    the circuit selects, using the selection code 'func', one
                of the 8 functions
*************************************************************************/
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
```

◇

The ALU circuit can be implemented in many forms. One of them is the *speculative* version (see Figure 6.16) described by the *Verilog* module from Example 6.4, 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.)

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 and adder and an 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 6.5 *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:*

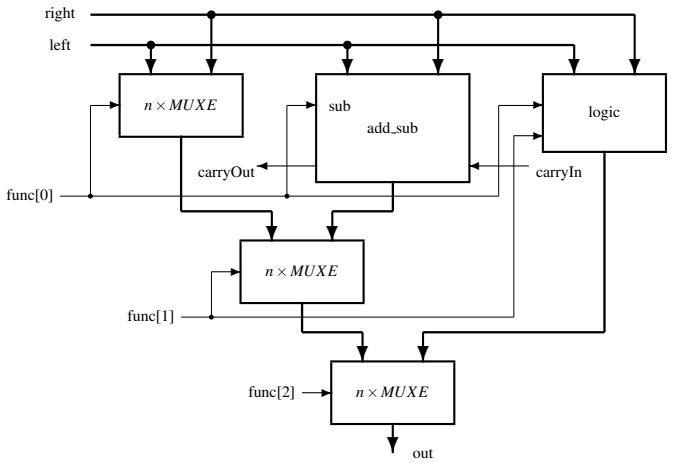


Figure 6.17: **The internal structure of an area optimized version of an ALU.** The `add_sub` module is smaller than an adder and a subtractor, but the operation "starts" only when `func[0]` is valid.

```
/*****************************************************************
File name:      structuralAlu.v
Circuit name:   ALU
Description:    structural description for
*****************************************************************/
 module structuralAlu(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   ),
                  .cin    (carryIn ),
                  .sub    (func[0] ));
    logic log(  .out    (logic    ),
                .left   (left     ),
                .right  (right    ),
                .op     (func[1:0]));
    mux2    shiftMux(.out(shift             ),
                     .in0(left              ),
                     .in1({1'b0, left[31:1]}),
                     .sel(func[0]           )),
            arithMux(.out(arith  ),
                     .in0(shift  ),
                     .in1(add_sub),
                     .sel(func[1])),
            outMux(.out(out    ),
                   .in0(arith  ),
                   .in1(logic  ),
                   .sel(func[2]));
 endmodule
```

```
/*****************************************************************
File name:      .v
Circuit name:
Description:
*****************************************************************/
 module mux2(input           sel     ,
             input   [31:0]  in0, in1,
             output  [31:0]  out     );
    assign  out = sel ? in1 : in0;
 endmodule
```

```
/*************************************************************************
File name:      .v
Circuit name:
Description:
*************************************************************************/
 module addSub(output [31:0]  out         ,
               output         cout        ,
               input  [31:0]  left, right ,
               input          cin, sub    );
    assign {cout, out} = left + (right ^ {32{sub}}) + (cin ^ sub);
 endmodule
```

```
/*************************************************************************
File name:      .v
Circuit name:
Description:
*************************************************************************/
 module logic(output reg [31:0] out         ,
              input      [31:0] left, right ,
              input      [1:0]  op          );
    integer i;
    wire [3:0] f;
    assign f = {op[0], ~(~op[1] & op[0]), op[1], ~|op};
    always @(left or right or f)
     for(i=0; i<32; i=i+1) logicSlice(out[i], left[i], right[i], f);

    task   logicSlice;
     output          o;
     input           l, r;
     input   [3:0]   f;
     o = f[{l,r}];
    endtask
 endmodule
```

The resulting circuit is represented in Figure 6.17. 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* `sub = func[0]` *is received. Usually, the code* `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 `task` to describe a circuit used many times in implementing a simple, repetitive structure.

The internal structure of ALU consists mainly in n slices, one for each input pair `left[i]`, `rught[i]` and a carry-look-ahead circuit(s) used for the arithmetic section. It is obvious that ALU is also a simple circuit. The magnitude order of the size of 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

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].

6.1.8 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, $COMP_n$, is represented in Figure 6.18a. The numbers to be compared are the n-bit positive integers `a` and `b`. Three are the outputs of the circuit: `lt_out`, indicating by 1 that $a < b$, `eq_out`, indicating by 1 that $a = b$, and `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 thre inputs must be connected as follows: $lt\_in = 0$, $eq\_in = 1$, $gt\_in = 0$.

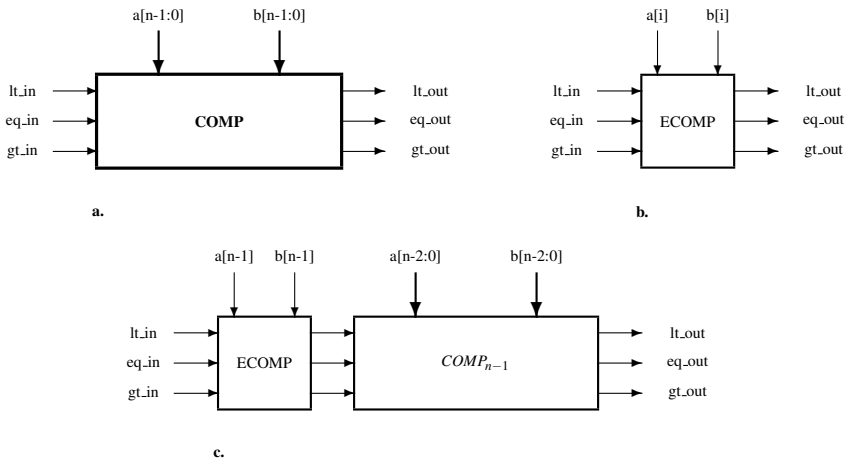


Figure 6.18: **The n-bit comparator, $COMP_n$. a.** The n-bit comparator. **b.** The elementary comparator. **c.** A recursive rule to built an $COMP_n$, serially connecting an $ECOMP$ with a $COMP_{n-1}$

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, $ECOMP = COMP_1$ (see Figure 6.18b)), ignoring $a[n-2:0]$ and $b[n-2:0]$. Results a recursive definition for the comparator circuit.

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:*

```
/*****************************************************************
File name:      e_comp.v
Circuit name:   Elementary Comparator
Description:    behavioral description of an elementary comparator
*****************************************************************/
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 6.19.* ◇

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.

6.2 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

[4]The actual size of the circuit can be minimized takeing into account that: (1) the compared input of ECOMP cannot be both 1, (2) the output eq_out of one $COMP_{n/2}$ is unused, and (3) the expansion inputs of both $COMP_{n/2}$ are all connected to fix values.

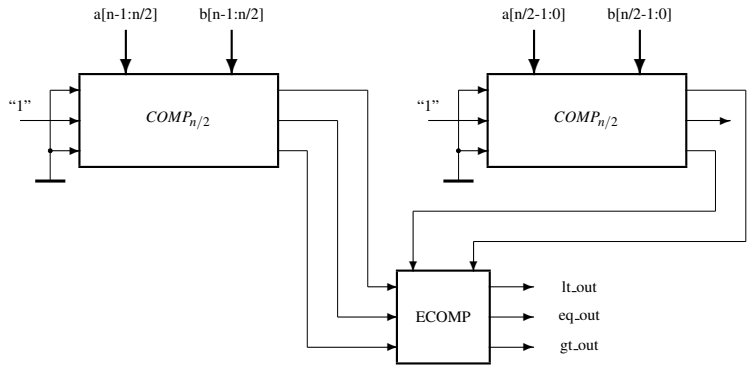


Figure 6.19: **The optimal n-bit comparator.** Applying the *divide et impera* principle a $COMP_n$ is built using two $COMP_{n/2}$ and an $ECOMP$. Results a log-depth circuit with the size in $O(n)$.

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 6.1 *For any n, all the functions of n binary-variables have a solution with a combinational logic circuit.* $\diamond$

Proof Any Boolean function of n variables can be written as:

$$f(x_{n-1},\ldots,x_0) = x'_{n-1}g(x_{n-2},\ldots,x_0) + x_{n-1}h(x_{n-2},\ldots x_0).$$

where:

$$g(x_{n-2},\ldots,x_0) = f(0,x_{n-2},\ldots,x_0)$$
$$h(x_{n-2},\ldots,x_0) = f(1,x_{n-2},\ldots,x_0)$$

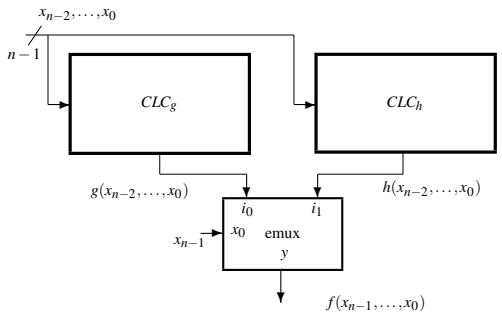


Figure 6.20: **The universal circuit.** For any $CLC_f(n)$, where $f(x_{n-1},\ldots,x_0)$ this recursively defined structure is a solution. EMUX behaves as an elementary universal circuit.

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 6.20. 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 sting 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:

```
/*************************************************************************
File name:        nU_circuit.v
Circuit name:     Universal Logic Cirsuit
Description:      behavioral description of the n-input universal logic
                  circuit
*************************************************************************/
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 the Turing-computable function there are many functions for which the solution is a *family of circuits*.

The solution imposed by the previous theorem is an universal circuit for computing the n binary variable functions. Let us call it n**U-circuit** (see Figure 6.21). The size of this circuit is $S_{universal}(n) \in O(2^n)$ and its complexity is $C_{universal}(n) \in O(1)$. 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
- 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.

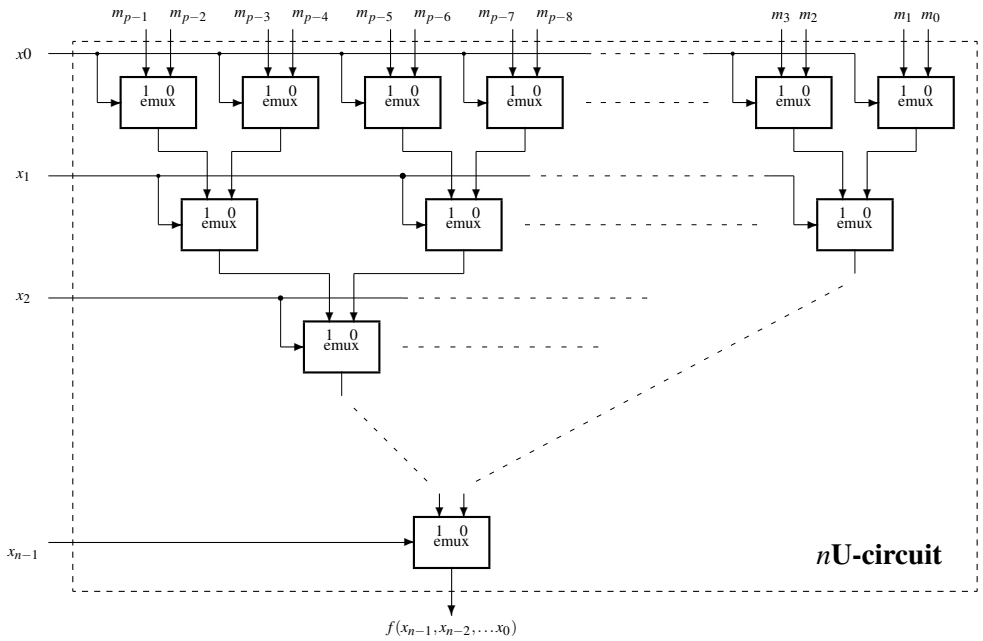


Figure 6.21: **The Universal Circuit as a tree of EMUXs.** The depth of the circuit is equal with the number, n, of binary input variables. The size of the circuit increases exponentially with n.

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 6.6 *Let be the following Verilog code:*

```
/*************************************************************************
File name:      three_input_functions.v
Circuit name:   Template for any 3-input logic function
Description:    behavioral description for any 3-input logic function
*************************************************************************/
 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").* $\diamond$

The "programmable" circuit for any 4-input Boolean function is obviously MUX_4:

$$out = f(in0, in1, in2, in3)$$

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 to 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:

```
out = x ? in1 : in0;
```

can be *removed*, if on its selected inputs specific 2-bit binary words are applied, according to the following rules:

- **if** $\{in1, in0\} = 00$ **then** $out = 0$
- **if** $\{in1, in0\} = 01$ **then** $out = x'$
- **if** $\{in1, in0\} = 10$ **then** $out = x$
- **if** $\{in1, in0\} = 11$ **then** $out = 1$

or, if the same variable, y, is applied on both selected inputs:

- **if** $\{in1, in0\} = yy$ **then** $out = y$

An EMUX can be *reduced*, if on one its selected inputs a 1-bit binary word are applied, being substituted with a simpler circuit according to the following rules:

- **if** $\{in1, in0\} = y0$ **then** $out = xy$
- **if** $\{in1, in0\} = y1$ **then** $out = y + x'$
- **if** $\{in1, in0\} = 0y$ **then** $out = x'y$
- **if** $\{in1, in0\} = 1y$ **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^{n-2} EMUXs) is applied a word containing binary values (0s and 1s) and binary variables (x_0s and x_0's). For the next levels the removing rules or the reducing rules are applied.

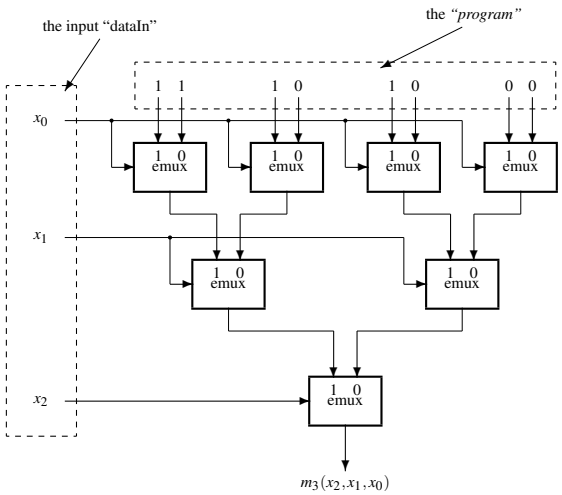


Figure 6.22: **The majority function.** The majority function for 3 binary variables is solved by a 3-level binary tree of EMUXs. The actual "program" applied on the "leafs" will allow to minimize the tree.

Example 6.7 *Let us solve the problem of majority function for three Boolean variable. The function* $maj(x_2,x_1,x_0)$ *returns 1 if the majority of inputs are 1, and 0 if not. In Figure 6.22 a "programmable" circuit is used to solve the problem.*

Because we intend to use the circuit only for the function $maj(x_2,x_1,x_0)$ *the first layer of EMUXs can be removed resulting the circuit represented in Figure 6.23a.*

On the resulting circuit the reduction rules are applied. The result is presented in Figure 6.23b. $\diamond$

The next examples refer to big n, but "program" containing repetitive patterns.

Example 6.8 *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 6.20).*

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 $0^{127}1$*, 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}*.* $\diamond$

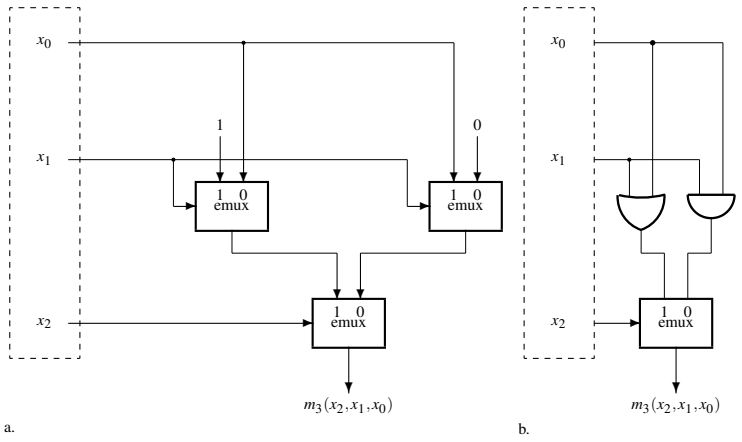


Figure 6.23: **The reduction process. a.** For any function the first level of EMUSs is reduced to a binary vector $((1,0)$ in this example). **b.** For the actual "program" of the 3-input majority function the second level is supplementary reduced to simple gates (an AND_2 and an OR_2).

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:

$$f(x_{n-1},\dots x_0)$$

$$g(x_{n-1},\dots x_0)$$

$$\dots$$

$$s(x_{n-1},\dots x_0)$$

is to connect in parallel many one-output circuits. The inefficiency of the solution become 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 time. One DCD is enough for many MUXs if the structure from Figure 6.24a is adopted. The DCD circuit is shared for implementing the functions f, $g,\dots s$. The shared DCD is used to compute all possible *minterms* (see *Appendix C.4*) needed to compute an n-variable Boolean function.

Figure 6.24b is an example of using the generic structure from Figure 6.24a 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 each OR to the i-th DCD output. The equivalent resulting circuit is represented in Figure 6.24c, where some OR inputs are connected to *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 *uniform* structure is "programmed" with a more or less *random* distribution of connections.

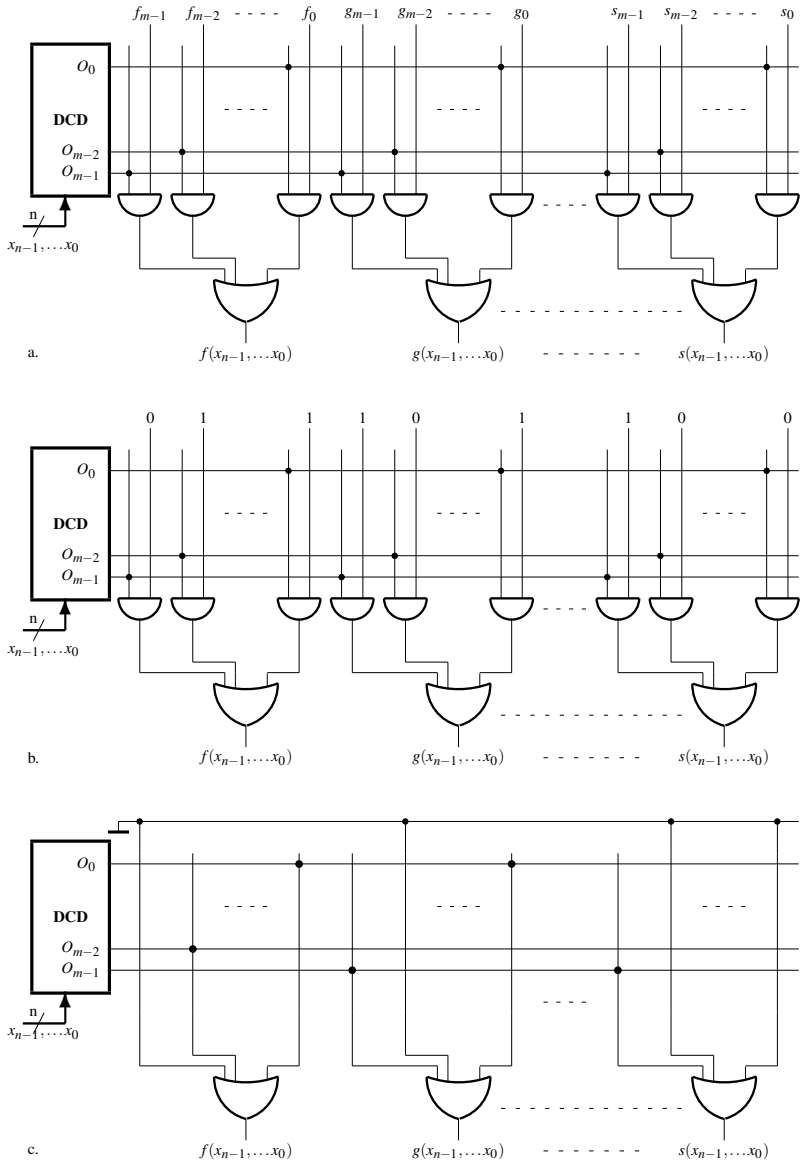


Figure 6.24: **Many-output random circuit. a.** One DCD and many AND-OR circuits. **b.** An example. **c.** The version using programmable connections.

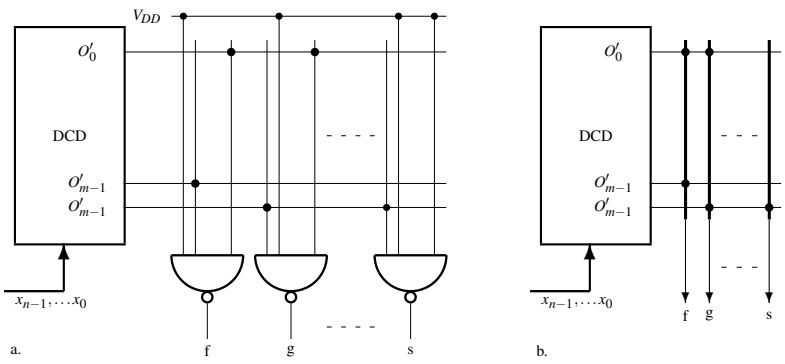


Figure 6.25: **The internal structure of a Read Only Memory used as trans-coder. a.** The internal structure. **b.** The simplified logic symbol where a thick vertical line is used to represent an m-input NAND gate.

If De Morgan transformation is applied, the circuit from Figure 6.24c is transformed in the circuit represented in Figure 6.25a, 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 *decode* the input binary word, the same as the gates used to *encode* the output binary word. Thus, a multi-output Boolean function works like a **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 6.26 represents the **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).

Input	Output
00 ... 00	11 ... 0
...	...
11 ... 10	10 ... 0
11 ... 11	01 ... 1

Figure 6.26: **The truth table for a multi-output Boolean function.** The input rows can be seen as addresses, from $00\ldots0$ to $11\ldots1$ and the output columns as the content stored at the corresponding addresses.

The trans-coder circuit can be interpreted as a *fix content memory*. Indeed, it works like a memory containing at the location 00...00 the word 11...0, ... 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 **read only memory**, ROM.

Example 6.9 *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 6.28).*

The Verilog code describing the circuit is:

```
/*****************************************************************
File name:      even_segments.v
Circuit name:   Seven-Segment Transcoder
Description:    behavioral description of the seven-segment transcoder
*****************************************************************/
 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'bxxxxxxx;
                endcase
 endmodule
```

The first solution is a 16-location of 7-bit words ROM (see Figure 6.27a. If inverted outputs are needed results the circuit from Figure 6.27b.

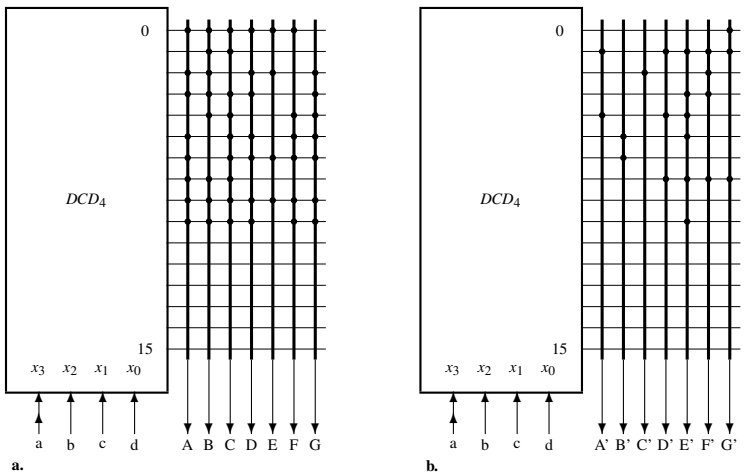


Figure 6.27: **The CLC as trans-coder designed serially connecting a DCD with an encoder.** Example: BCD to 7-segment trans-coder. **a.** The solution for non-inverting functions. **b.** The solution for inverting functions.

◇

abcd	ABCDEFG
0000	1111110
0001	0110000
0010	1101101
0011	1111001
0100	0110011
0101	1011011
0110	1011111
0111	1110000
1000	1111111
1001	1111011
1010	-------
....	....
1111	-------

Figure 6.28: **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".

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.

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 tolls 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 DCD_4 according to Definition 2.9. Evaluate the area of the circuit, using the cell library from Appendis E, with the* placement efficiency[5] 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 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 DCD_n using EDMUXs. Evaluate the size and the depth of the resulting structure.*

Multiplexors

Problem 6.6 *Draw MUX_4 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.*

[5]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.

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.

6.4.1 Recursive circuits

Problem 6.12 *A comparator is circuit designed to compare two n-bit positive integers. Its definition is:*

```
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 6.13 *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 6.14 *Design an expandable priority encoder with* minimal size *starting from an elementary priority encoder, EPE, defined for* $n = 2$*. Evaluate its depth.*

Problem 6.15 *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 6.16 *Design an expandable carry look-ahead adder starting from an elementary circuit.*

Problem 6.17 *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 6.18 *Design an expandable adder/subtracter 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.*

6.4.2 Random circuits

Problem 6.19 *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 6.20 *Design a converter from Gray counting to binary counting for n-bit numbers.*

Problem 6.21 *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 6.22 *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 6.23 *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 6.24 *Exemplify the serial composition, the parallel composition and the serial-parallel composition in 0 order systems.*

Problem 6.25 *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 6.26 *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 6.27 *Compute the biggest size and the biggest depth of an n-input, 1-output circuit implemented using the universal circuit.*

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[1]

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.

[1] She 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 `y0` of the elementary decoder to its input `x0` (see Figure 7.1b). Suppose that `y0 = 0 = x0`. After the time interval equal with t_{pLH}[2] the output `y0` becomes 1. After another time interval equal with t_{pHL} the output `y0` 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 $f_{osc} = 1/(t_{pLH} + t_{pHL})$.

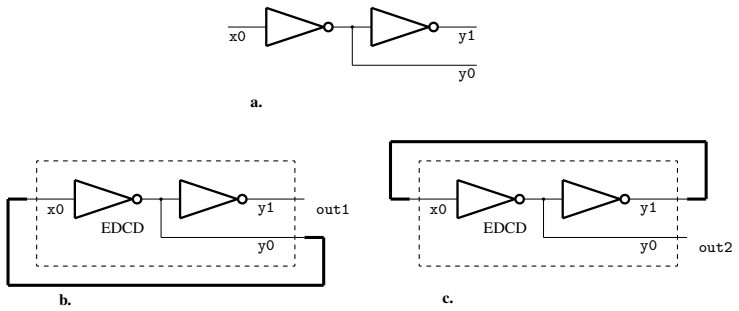


Figure 7.1: **The two loops closed over an elementary decoder. a.** The simplest combinational circuit: the one-input, elementary decoder. **b.** The unstable, inverting loop containing one (odd) inverting logic level(s). **c.** The stable, non-inverting loop containing two (even) inverting levels.

The stable loop is obtained connecting the output `y1` of the elementary decoder to the input `x0` (see Figure 7.1c). If `y1 = 0 = x0`, then `y0 = 1` fixing again the value 0 to the output `y1`. If `y1 = 1 = x0`, then `y0 = 0` fixing again the value 1 to the output `y1`. 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.)

[2] the propagation time through the inverter when the output switches from the low logic level to the high level.

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.)*$\diamond$

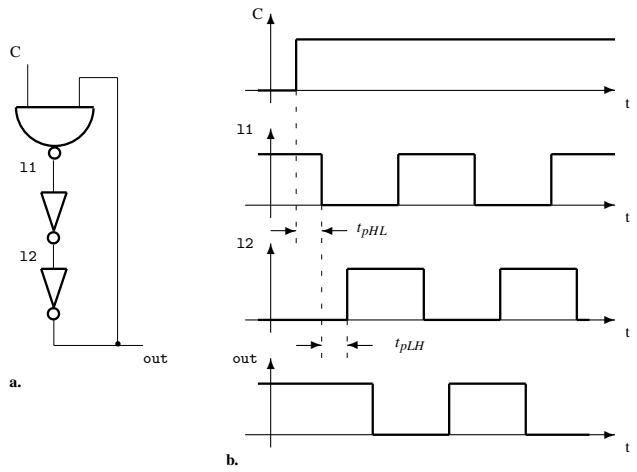


Figure 7.2: **The unstable loop.** The circuit version used for a low-cost and low-performance clock generator. **a.** The circuit with a three (odd) inverting circuits loop coupled. **b.** The wave forms drawn takeing into account the propagation times associated to the low-high transitions (t_{pLH}) 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, unuseful 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 fist 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 2.15) 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.* $\diamond$

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".

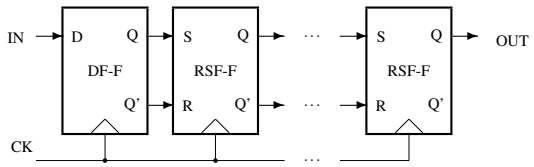


Figure 7.3: **The n-bit serial register (SR_n).** Triggered by the active edge of the clock, the content of each RSF-F is loaded with the content of the previous RSF-F.

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:*

```
/*****************************************************************
File name:      serial_register.v
Circuit name:   Serial register
Description:    behavioral description of a n-bit serial register
*****************************************************************/
 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 *n-bit latch*
- the asynchronous *random access memory* (RAM),

both involved in building big but simple recursive structures.

7.3.1 The *n*-Bit Latch

The 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.

VeriSim 7.2 *A 16-bit latch is described in Verilog as follows:*

```
/*****************************************************************
File name:      n_latch.v
Circuit name:   n-Bit Latch
Description:    behavioral description of a n-bit latch
*****************************************************************/
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.

7.3.2 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.

Definition 7.2 *The m-bit random access memory, 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* $A_{n-1},\ldots,A_0$.[3] 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_{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

[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.)

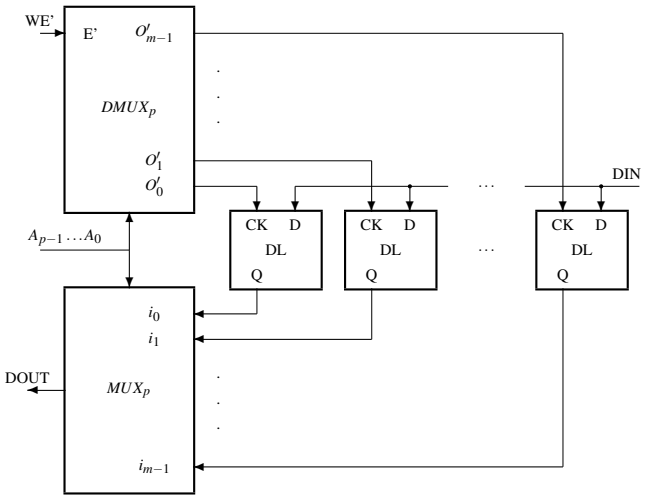


Figure 7.4: **The principle of the random access memory (RAM).** The clock is distributed by a DMUX to one of $m = 2^p$ DLs, and the data is selected by a MUX from one of the m DLs. Both, DMUX and MUX use as selection code a p-bit address. The one-bit data DIN can be stored in the clocked DL.

- 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
- t_{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_{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_{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_{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:*

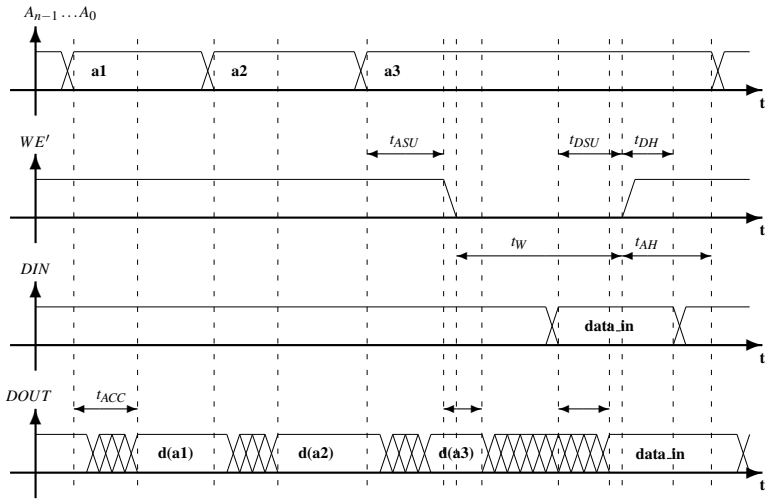


Figure 7.5: **Read and write cycles for an asynchronous RAM.** Reading is a combinational process of selecting. The access time, t_{ACC}, is given by the propagation through a big MUX. The write enable signal must be strictly included in the time interval when the address is stable (see t_{ASU} and t_{AH}). Data must be stable related to the positive transition of WE' (see t_{DSU} and t_{DH}).

```
/*************************************************************************
File name:      ram.v
Circuit name:   Asynchronous RAM
Description:    behavioral descriiption of an asynchronous random-access
                memory
*************************************************************************/
 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

    always @(din or addr or we) if (we) mem[addr] = din; // writing
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 it 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_{n-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 demultimplexing circuit used to clock, when WE = 1, the latches selected (addressed) by the current value of the address: $A_{p-1},\ldots A_0$
- 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:

```
addr[p-1:0] = {rowAddr[r-1:0], colAddr[q-1:0]}
```

The row address `rowAddr[r-1:0]` selects a many-word, while from the selected many-word, the column address `colAddr[q-1:0]` selects the word addressed by the address `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 `addr[p-1:0]`.

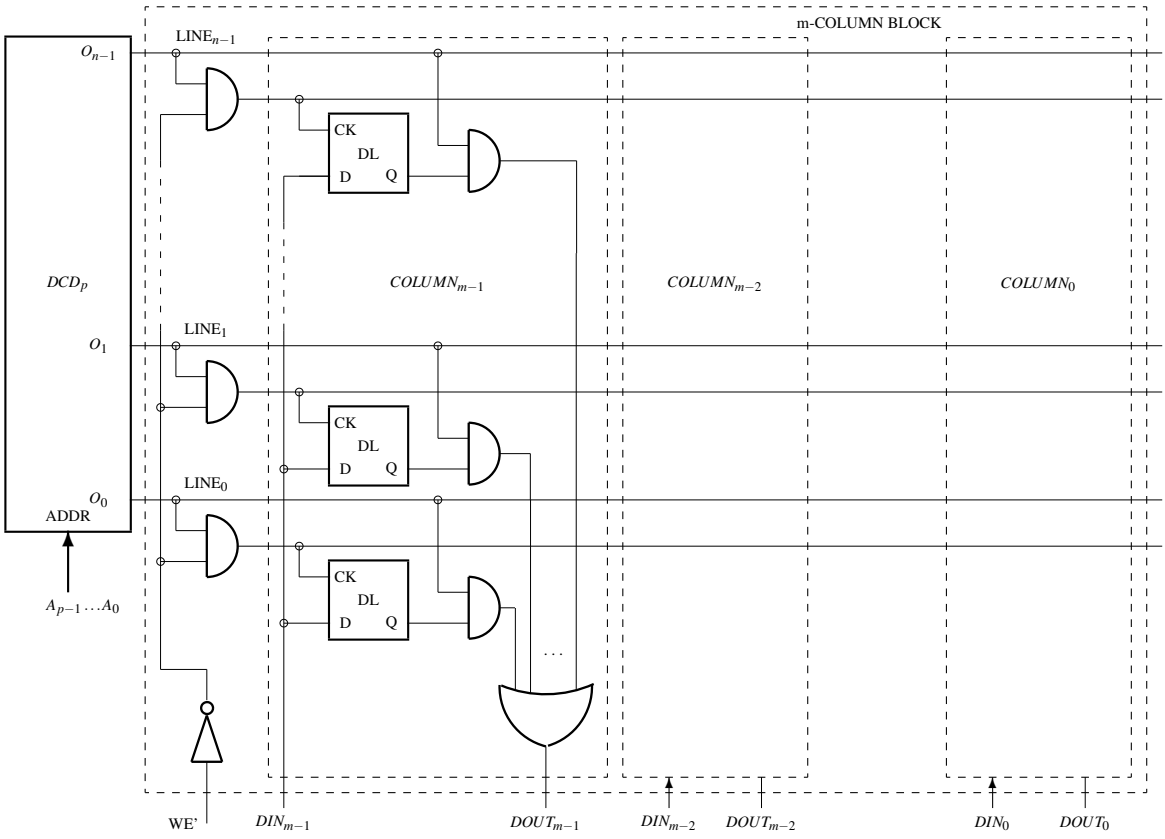


Figure 7.6: **The asynchronous *m*-bit word RAM.** Expanding the number of bits per word means to connect in parallel one-bit word memories which share the same decoder. Each COLUMN contains the storing latches and the AND-OR circuits for one bit.

While the size decoder for a one block memory version is in the same order with the number of words ($S_{DCD_p} \in 2^p$), 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.

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,

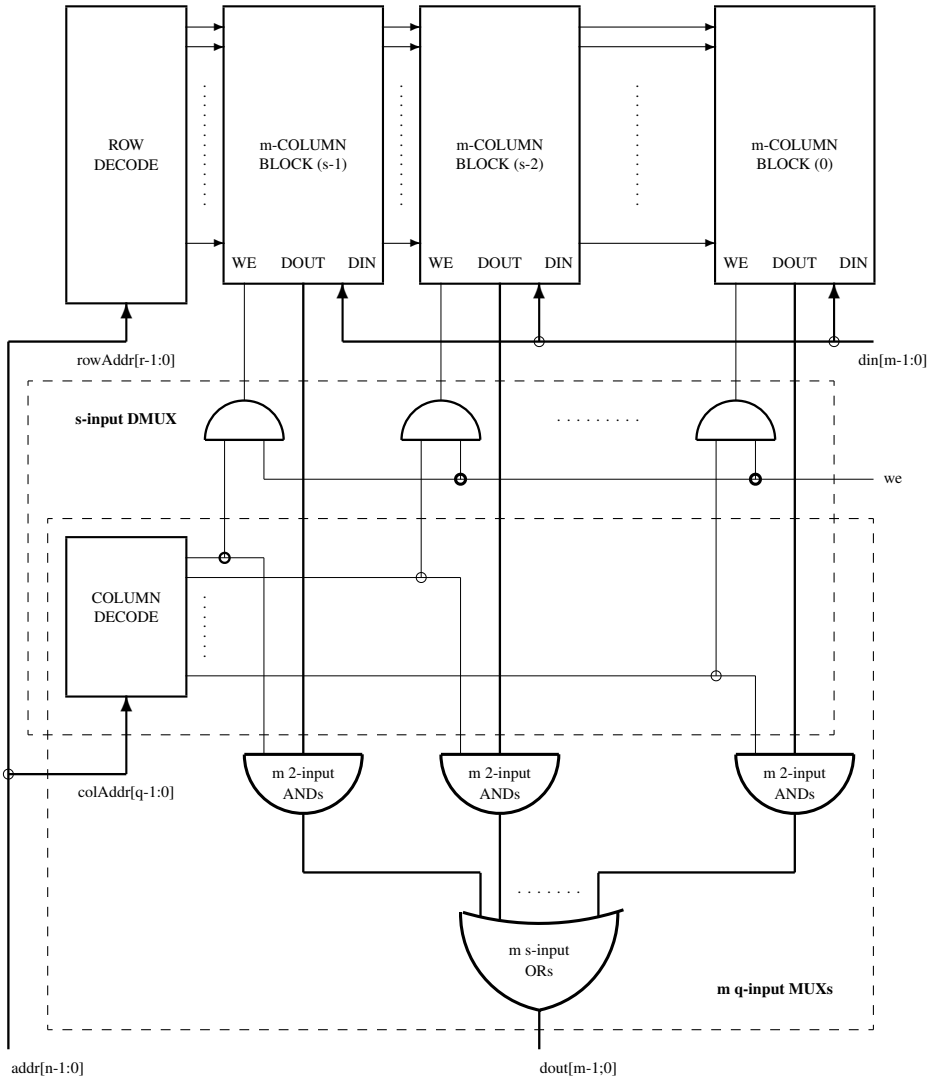


Figure 7.7: **RAM version with two dimension storage array.** A number of m-bit blocks are parallel connected and driven by the same row decoder. The column decoder selects to outoput an m-bit word from the $(s \times m)$-bit row.

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:*

```
/*****************************************************************
File name:      sram.v
Circuit name:   Synchronous RAM
Description:    behavioral description of a synchronous RAM
*****************************************************************/
 module sram(   input       [63:0]  din ,
                input       [8:0]   addr,
                output  reg [63:0]  dout,
                input               we, clk);
    reg     [63:0]  mem[511:0];
    always  @(posedge clk)  if (we) dout <= din          ;
                             else   dout <= mem[addr]    ; // reading
    always  @(posedge clk)  if (we) mem[addr] <= din     ; // writing
 endmodule
```

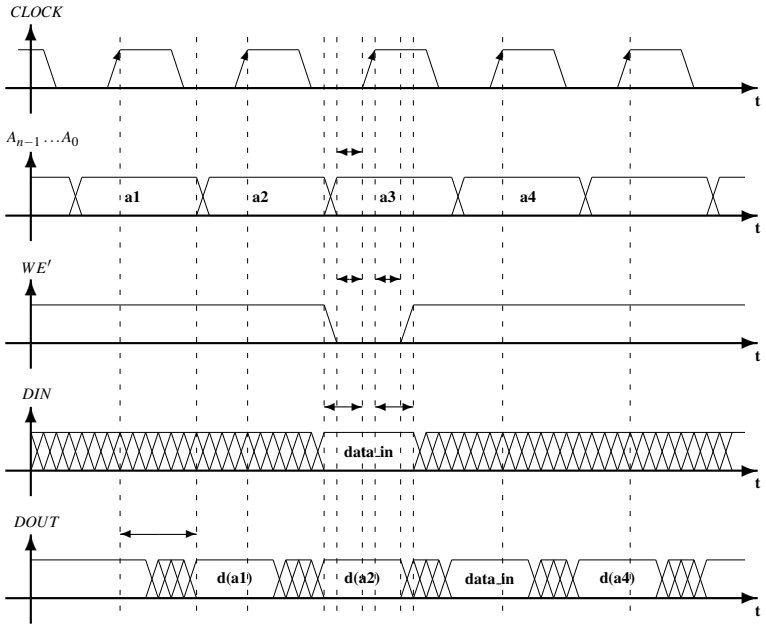


Figure 7.8: **Read and write cycles for SRAM.** For the *flow-through* version of a *SRAM* the time behavior is similar to a register. The set-up and hold time are defined related to the active edge of clock for all the input connections: *data*, *write-enable*, and *address*. The data output is also related to the same edge.

◇

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.*

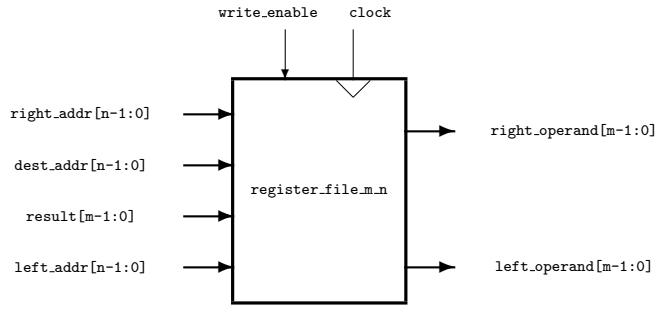


Figure 7.9: **Register file.** In this example it contains 2^n m-bit registers. In each clock cycle any two registers can be read and writing can be performed in anyone.

```
/*************************************************************************
File name:      register_file.v
Circuit name:
Description:
*************************************************************************/
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 2 m-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[4]. 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).

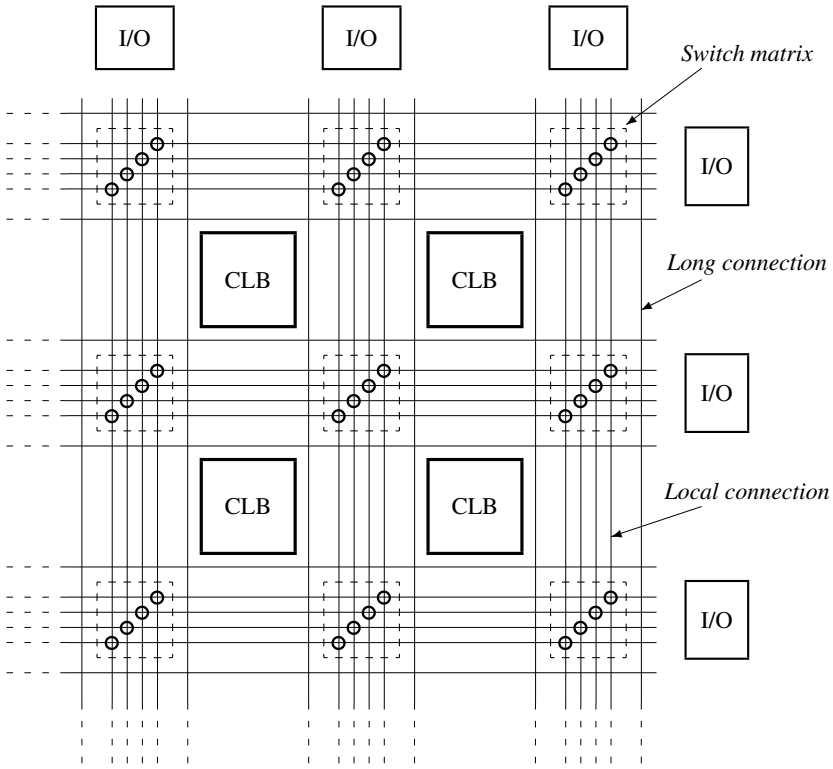


Figure 7.10: **Top level organization of FPGA.**

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

[4]The terminology introduced in this section follows the Xlilinx style in order to support the associated lab work.

- **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`.

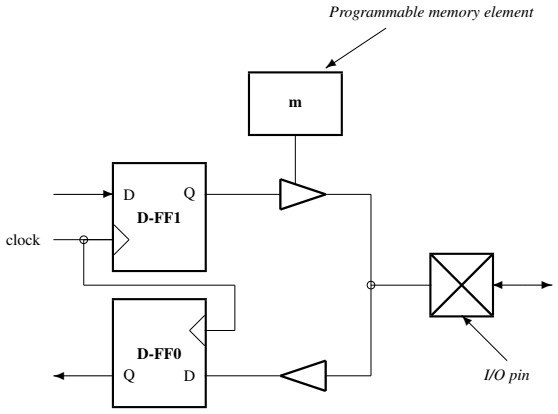


Figure 7.11: **Input-Output interface.**

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 `mi = 1` then the associated nMOS transistor is *on* and between its drain end source the resistor has a small value. If `mi = 0` then the associated nMOS transistor is *off* and the two ends of the switch are not connected.

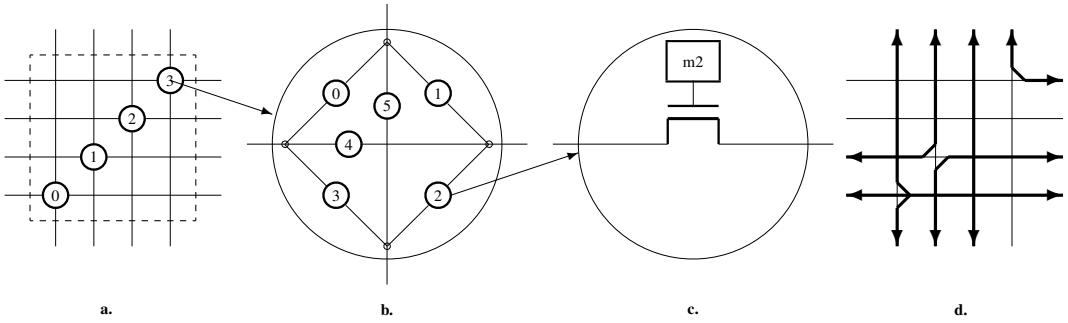


Figure 7.12: **The structure of a Switch Node. a.** A Switch Node with 4 switches. **b.** The organization of a switch. **c.** A line switch. **d.** An example of actual connections.

For example, the configuration shown in Figure 7.12d is programmed as follows:

switch 0 : {`m0, m1, m2, m3, m4, m5`} = `011010;`

switch 1 : {`m0, m1, m2, m3, m4, m5`} = `101000;`

switch 2 : {`m0, m1, m2, m3, m4, m5`} = `000001;`

switch 3 : {`m0, m1, m2, m3, m4, m5`} = `010000;`

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 `m0, m1, ... m15`. The entire sub-module is called LUT (from **look-up table**). The memory elements `m0, m1, ... m15`, 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

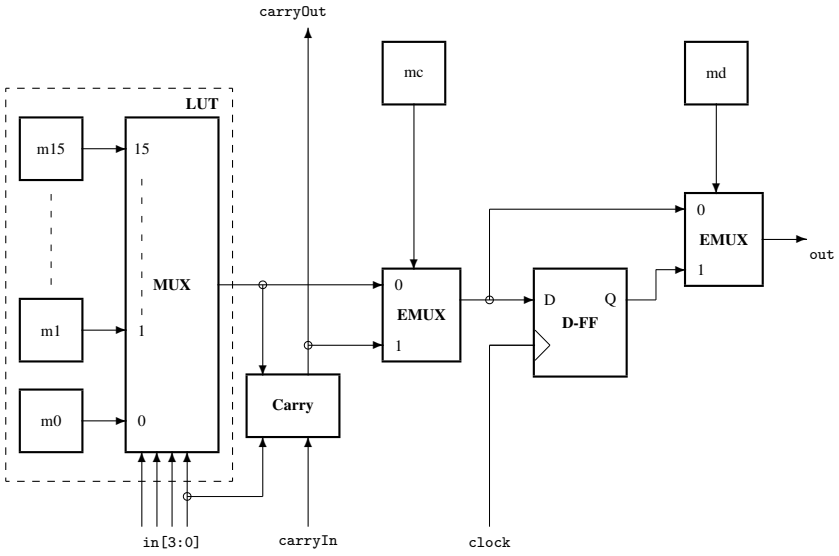


Figure 7.13: **The basic building block: the bit slice (BS).**

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 `md` is programmed appropriately. Indeed, if `md = 1`, then the output of the BS comes form 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.

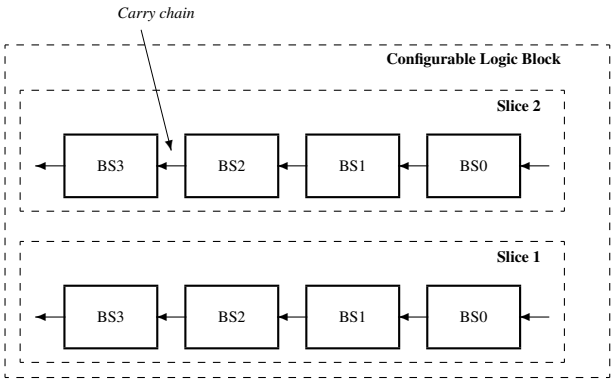


Figure 7.14: **Configurable logic block.**

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 vey 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 7.15. Indicate the functionality and explain it.*
Hint: *emphasize the structure of an elementary multiplexer.*

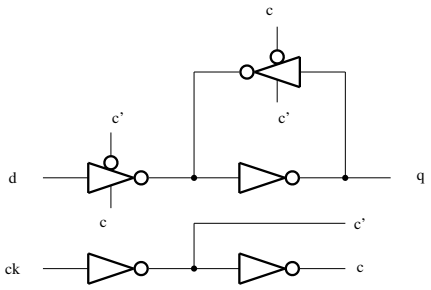


Figure 7.15: **?**

Problem 7.6 *Explain how it works and find an application for the circuit represented in Figure 7.16.*
Hint: *Imagine the tristate drivers are parts of two big multiplexors.*

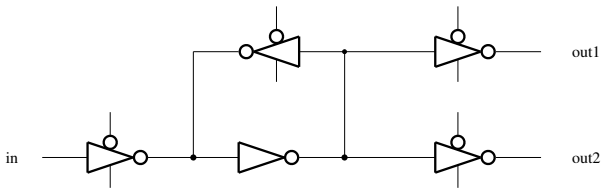


Figure 7.16: **?**

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 7.17. 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.*

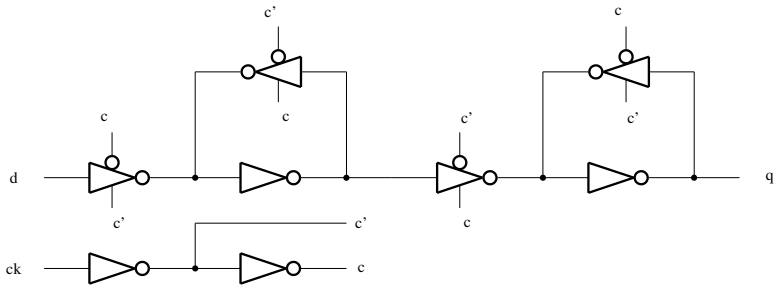


Figure 7.17: **?**

Problem 7.12 *Let be the circuit from Figure 7.18. 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.*

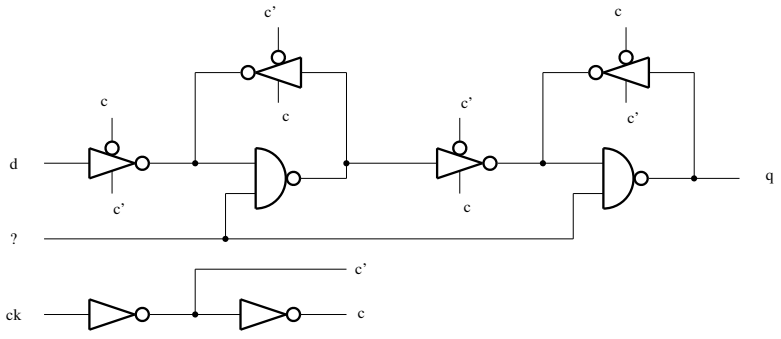


Figure 7.18: **?**

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 16×16 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 7.19.*
Hint*: the write process is identical with the flow-through version.*

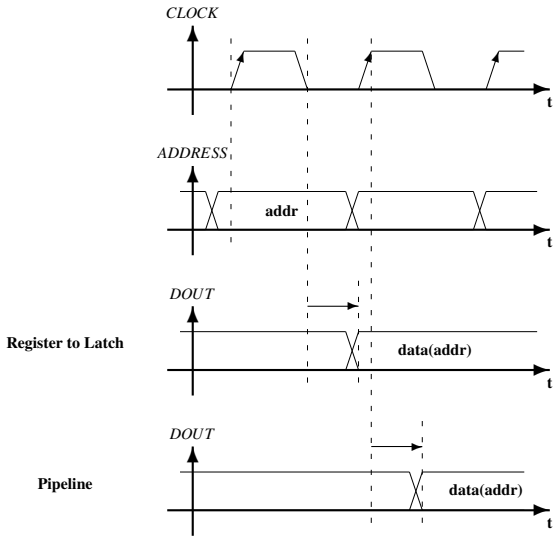


Figure 7.19: **Read cycles.** Read cycle for the *register to latch* version and for the *pipeline* version of *SRAM*.

Problem 7.21 *Design the pipeline version of SRAM defined in Figure 7.19.*
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 ceratin 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 7.20 is shown the serial-parallel $SPR_{n\times m}$.*

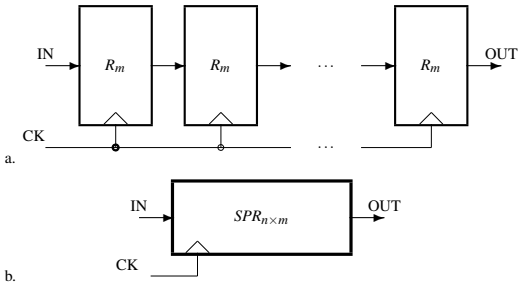


Figure 7.20: **The serial-parallel register. a.** The structure. **b.** The logic symbol.

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.*

```
module pipeline     #(parameter   n = 8, m = 16, p = 20)
                      (output     reg[m-1:]   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);
   // ...
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_out}) and maximum output required time after clock (t_{reg_out}) using circuit timing from Appendix* Standard cell libraries*.*

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     ;
    wire            mode    ; // mode = 0: system1 reads, system2 writes
                              // mode = 1: system2 reads, system1 writes
    wire    [1:0]   com12, com21   ;
    system1 system1(in1, out1, com12, com21,
                    memOut1 ,
                    memIn1  ,
                    addr1   ,
                    we1     ,
                    mode    ,
                    clock   );
    system2 system2(in2, out2, com12, com21,
                    memOut2 ,
                    memIn2  ,
                    addr2   ,
                    we2     ,
                    clock   );
    bufferMemory    bufferMemory(   memOut1 ,
                                    memIn1  ,
                                    addr1   ,
                                    we1     ,
                                    memOut2 ,
                                    memIn2  ,
                                    addr2   ,
                                    we2     ,
                                    mode    ,
                                    clock   );
endmodule
```

```
module memory #(parameter n=32, m=10)
       (    output  reg [n-1:0] dataOut     ,   // data output
            input       [n-1:0] dataIn      ,   // data input
            input       [m-1:0] readAddr    ,   // read address
            input       [m-1:0] writeAddr   ,   // write address
            input               we          ,   // write enable
            input               enable      ,   // module enable
            input               clock       );

   reg [n-1:0] memory[0:(1 << m)-1];
```

```
    always @(posedge clock) if (enable) begin
                                  if (we) memory[writeAddr] <= dataIn ;
                                  dataOut <= memory[readAddr]         ;
                            end
endmodule
```

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[1]

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.

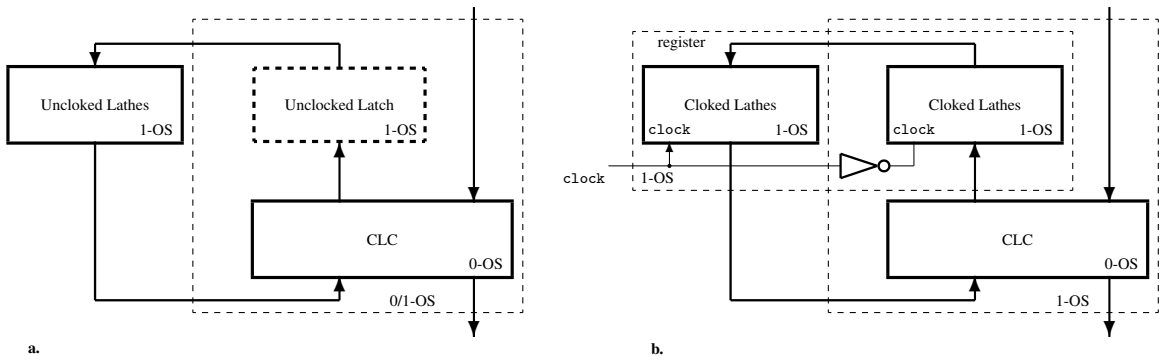


Figure 8.1: **The two type of 2-OS. a.** The asynchronous automata with a hazardous loop over a transparent latch. **b.** The synchronous automata with a edge clock controlled loop closed over a non-transparent register.

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:

[1]Quote from *Tao Te King* of Lao Tzu translated by Brian Browne Walker.

- *asynchronous automata*, for which the loop is closed over **unclocked latches**, through combinational 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* $f : X \times Q \to Q$

g *: the output transition function, with one of the following definitions:*

- $g : X \times Q \to Y$ *for Mealy type automaton*
- $g : Q \to Y$ *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). $\diamond$

Definition 8.2 *A* **finite automaton***, FA, is an automaton with Q a finite set.* $\diamond$

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* $|Q| \in O(f(n))$. $\diamond$

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.* $\diamond$

Definition 8.5 *An* **initial automaton** *is an automaton having a set of initial states,* Q'*, which is a subset of Q,* $Q' \subset Q$. $\diamond$

Definition 8.6 *A* **strict initial automaton** *is an automaton having only one initial state,* $Q' = \{q_0\}$. $\diamond$

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.* $\diamond$

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:*

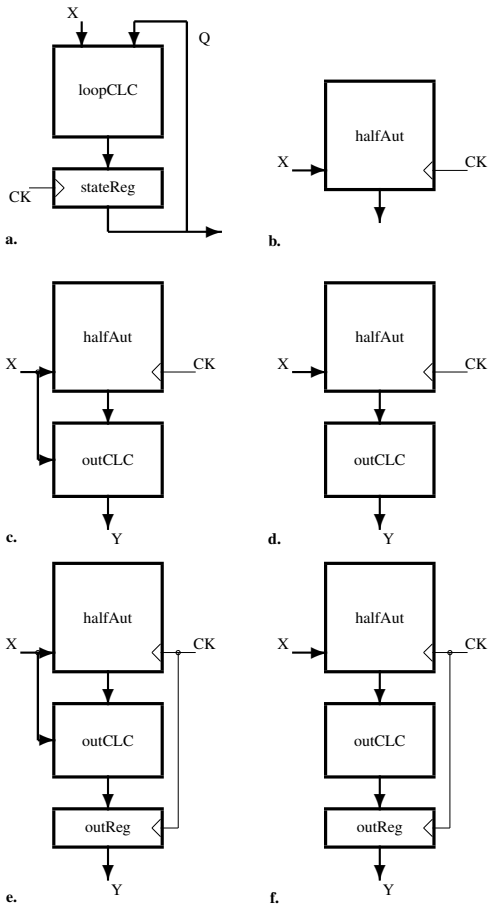


Figure 8.2: **Automata types. a.** The structure of the half-automaton ($A_{1/2}$), the no-output automaton: the state is generated by the previous state and the previous input. **b.** The logic symbol of half-automaton. **c.** Immediate Mealy automaton: the output is generated by the current state and 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:*

$$y(t) = g(q(t-1)) = g(f(x(t-2), q(t-2)).\diamond$$

Proof The proof is evident starting from the previous two definitions. $\diamond$

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)

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.*

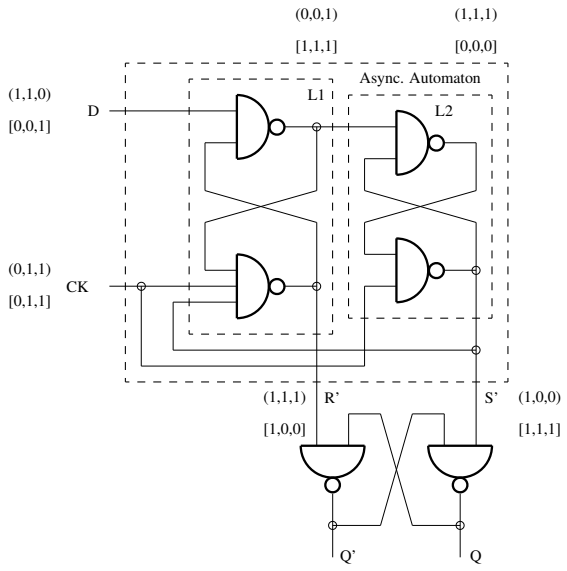


Figure 8.3: **The D flip-flop implemented as a 2-OS system.** The asynchronous automaton built up loop connecting two unclocked latches allows to trigger the output latch according to the input data value available at the positive transition of clock.

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 in 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*.

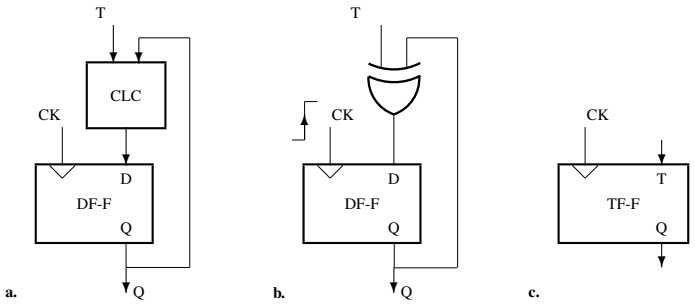


Figure 8.4: **The T flip-flop. a.** It is the simplest automaton because: has 1-bit state register (a DF-F), a 2-input loop circuit (one as automaton input and another to close the loop), and direct output from the state register. **b.** The structure of the T flip-flop: the XOR_2 circuits complements the state is $T = 1$. **c.** The logic symbol.

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 invertor, 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

clock frequency is f_{CK}, then the frequency of the signal received to the output Q is $f_{CK}/2$ (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).

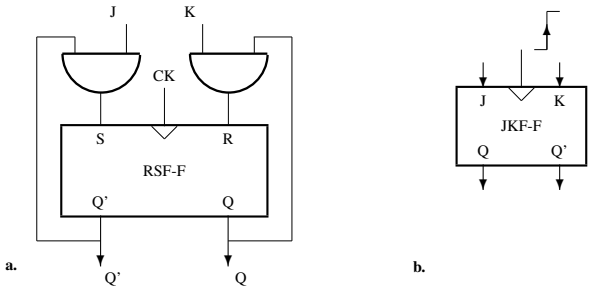


Figure 8.5: **The JK flip-flop.** It is the simplest two-input automaton. **a.** The structure: the loop is closed over a master-slave RSF-F using only two AND_2. **b.** The logic symbol.

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).

8.3 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 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 AND_{i+1} must be used.

Definition 8.8 *The n-bit* synchronous counter*, $COUNT_n$, has a clock input, CK, a command input, INC_0, an n-bit data output, $Q_{n-1},\ldots Q_0$, and an expansion output, INC_n. If $INC_0 = 1$, the active edge of clock increments the value on the data output (see Figure 8.6).* $\diamond$

There is also a recursive, constructive, definition for $COUNT_n$.

Definition 8.9 *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 8.6). $COUNT_1$ is a T flip-flop and an AND_2 with the inputs Q_0 and INC_0 which generates INC_1.* $\diamond$

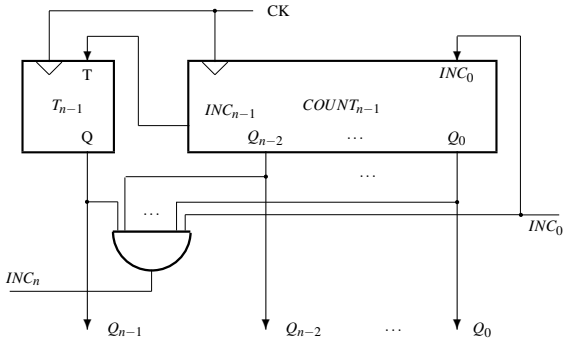


Figure 8.6: **The synchronous counter.** The recursive definition of a synchronous counter has $S_{COUNT}(n) \in O(n^2)$ and $T_{COUNT}(n) \in O(log\, n)$, because for the i-th range one TF-F and one AND_i are added.

Example 8.1 **The Verilog description of a synchronous counter follows:*

[2]There exist also asinchronous counters. They are simpler but less performant.

```
/*********************************************************************
File name:      sync_counter.v
Circuit name:   Synchronous Counter
Description:    structural description of a synchronous counter as a
                T-type register loop connected with an AND prefix network
*********************************************************************/
 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
```

```
/*********************************************************************
File name:      t_reg.v
Circuit name:   T-type Register
Description:    behavioral description of a register built using T-type
                flip-flops instead of D-type flip flops
*********************************************************************/
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.* $\diamond$

It is obvious that $C_{COUNT}(n) \in O(1)$ 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 $COUNT_n$). The size of $COUNT_n$, according to the *Definition 4.4*, can be computed starting from the following iterative form:

$$S_{COUNT}(n) = S_{COUNT}(n-1) + (n+1) + S_T$$

and results:

$$S_{COUNT}(n) \in O(n^2)$$

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:

$$T_{COUNT}(n) = t_{pT} + t_{pAND_n} + t_{SU} \in O(log\ n)$$

where: $t_{pT} \in O(1)$ is the propagation time through the T flip-flop, $t_{pAND_n} \in O(log\ n)$ is the propagation time through the AND_n (in the fastest version it is implemented using a tree of AND_2 gates) gate and $t_{SU} \in O(1)$ 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:

```
/*************************************************************************
File name:      full_counter.v
Circuit name:   Full Counter
Description:    behavioral description of a counter with all the possible
                features (reset, load, up-count, down-count)
*************************************************************************/
 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            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 feeded 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,* `sReg[3:0]`*, and the loop closed through a 2-input XOR in two versions:*

- *from* `sReg[3]` *and* `sReg[0]` *(see Figure 8.7a)*
- *from* `sReg[3]` *and* `sReg[1]` *(see Figure 8.7b)*

Let's consider the initial state in both cases: `sReg = 4'b0001`*. The circuit from Figure 8.7a generates the following periodic sequence starting from the initial state:*

```
0001
0011
0111
1111
1110
1101
1010
0101
1011
0110
1100
1001
0010
0100
1000
0001
...
```

while the circuit form Figure 8.7b generate a shorter periodic sequence, as follow:

```
0001
0010
0101
1010
0100
1000
0001
...
```

Important note*: the initial state* `sReg = 4'b0000` *must be avoided, because from this state there is no evolution.*

◇

[3]See a tutorial at: http://homepages.cae.wisc.edu/ẽce553/handouts/LFSR-notes.PDF

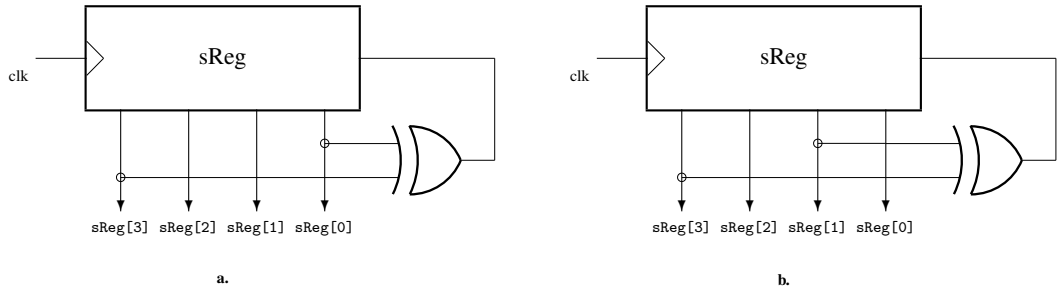


Figure 8.7: **Examples of LFSR.**

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:

$$sReg[7] \oplus sReg[5] \oplus sReg[1]$$

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:

[4] `https://users.ece.cmu.edu/~koopman/lfsr/index.html`

```
/*****************************************************************
File:           progLFSR.v
Circuit name:   Programmable Linear Feedback Shift Register
Description:    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
*****************************************************************/
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 8.10 *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.

◇

Definition 8.11 *A 3-port S^3A is defined by: $S^3A = (F \times X \times D \times L \times R; Y; \mathscr{S}; f, g)$ where:*

- $\mathscr{S} = (S_0 \times S_1 \times \ldots, \times S_{m-1})$ *with* $S_i = \{0,1\}^n$ *for* $i = 0,\ldots,m-1$ *is the structured state space*
- $H = \{0,1\}^{log_2\, p}$ *is used to select a function from the set* $\{h_0, h_1, \ldots, h_p\}$
- $X = \{0,1\}^n$ *is the finite set of inputs binary represented on n bits*

- $Y = (\{0,1\}^n \times \{0,1\}^n)$ *is the finite set of outputs binary represented by two n-bit words*
- $D = L = R = \{0,1\}^{log_2\, m}$ *are sets of pointers in the Cartesian product* $\mathscr{S}$
- $g : (L \times R) \rightarrow (S_L \times S_R)$ *is the output transition function*
- $f : (H \times X \times D \times L \times R \times \mathscr{S}) \rightarrow S_D$ *is the state transition function of form* $h_H : (X, S_L, S_R) \rightarrow S_D$.

$\diamond$

An S^3A is implemented using a synchronous RAM to store the state. The inputs D, L, R are the address which select the elements of the Cartesian product stored in the m 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 m 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 $1/m$. The size of the combinational circuit for f belongs, in the worst case, to $O(2^{2n+log_2 p})$, while for the standard automaton it belongs, in the worst case, to $O^{mn+log_2 p}$. Results a decrease in size belonging to $O(2^{n(m-2)})$. The time performance decreases linearly with m, while the size decreases exponentially with m. There is no room for debate: when possible, the S^3A is the solution.

Multi-port S^{3}A

Because the binary functions dominate the class of arithmetic and logic functions, multi-port S^{3}As are used in designing the executing core of any processing element. The most frequently used multi-port S^{3}A is a 3-port S^{3}A. 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.gg

Definition 8.12 *A 3-port Structured State Space Half-Automaton,* S^3HA *is defined as following:*

$$S^3HA = (X \times DA \times LA \times RA, \mathbf{Q}, f)$$

where:

- $\mathbf{Q} = (Q_0 \times Q_1 \times \ldots, \times Q_{s-1})$ *: 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* $\mathbf{Q}$ *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* $\mathbf{Q}$ *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* $\mathbf{Q}$ *to be used as right operand in the current state transition*
- $f : (X \times LA \times RA \times \mathbf{Q}) = (X \times P(i,s,\mathbf{Q}) \times P(j,s,\mathbf{Q})) = (X \times Q_i \times Q_j) \rightarrow P(k,s,\mathbf{Q}) = Q_k$ *is the state transition function where* $i \in LA$, $j \in RA$, $k \in DA$

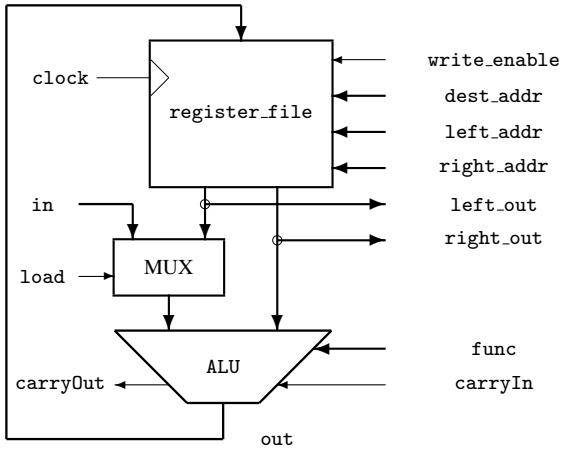


Figure 8.8: 32-bit RALU.

◇

Example 8.3 *Let be a RALU designed with two modules already presented in the previous sections: the ALU exemplified in Example 6.4 and the register file presented in Simulation 7.5. In Figure 8.8 is represented the schematic of a 32-bit RALU.*

```
/*****************************************************************
File:           RALU.v
Circuit name:   RALU: Register file with Arithmetic and Logic Unit
Description:    register file with 16 32-bit register and an ALU with 8
                generic arithmetic and logic functions.
*****************************************************************/
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           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  ),
                        .clock           (clock         ));

    ALU alu(.carryIn    (carryIn                ),
            .func       (func                   ),
            .left       (load ? in : left_out   ),
            .right      (right_out              ),
            .carryOut   (carryOut               ),
            .out        (out                    ));
endmodule
```

◇

8.4 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).

8.4.1 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 "runs" 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 (+).

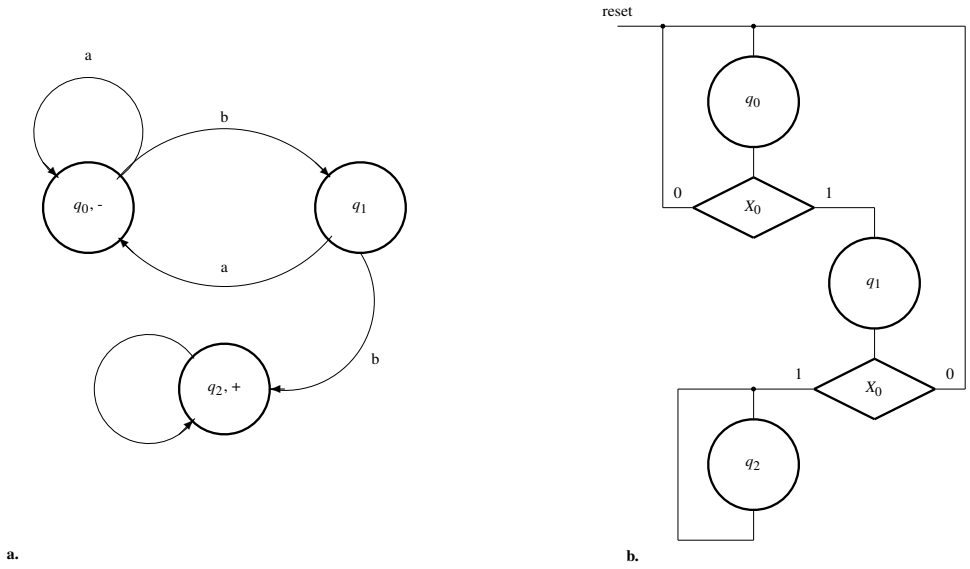


Figure 8.9: **Example of flow-chart for a half-automaton.** The machine is a "double b detector". It stops when the first bb occurs.

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 8.4 *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 8.9a. 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*

- *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 8.9b). The input I takes the binary value 0 for the the symbol a and the binary value 1 for the symbol b. $\diamond$

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 sate, the value of the output in that sates. 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.

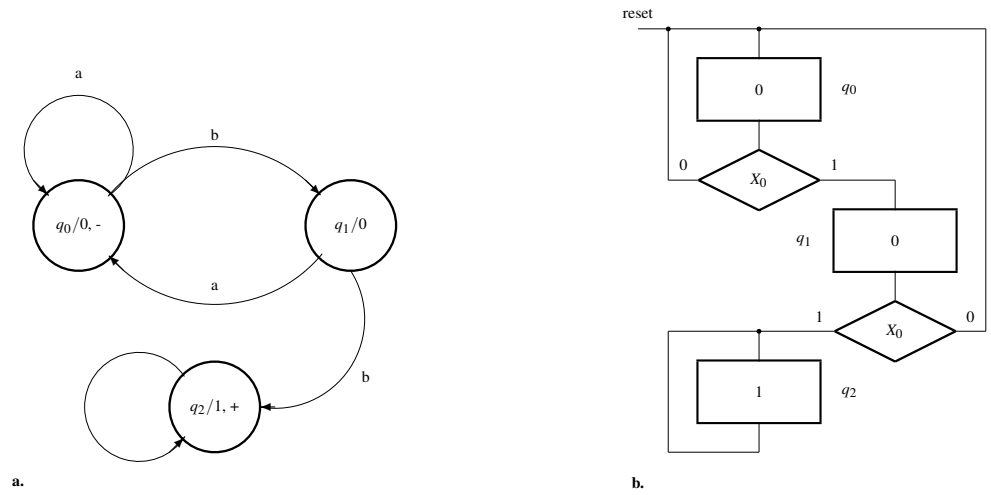


Figure 8.10: **Example of flow-chart for a Moore automaton.** The output of this automaton tells us: "*bb* was already detected".

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 8.5 *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.*

The associated flow-chart is in, in the first version represented by Figure 8.10a. 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 8.10b) uses three rectangles, one for each state. $\diamond$

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 8.6 *Let us solve again the same problem of bb detection using a Mealy automaton. The resulting flow-chart is in Figure 8.11a. 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 8.11b represents the second representation. $\diamond$

We can say the Mealy automaton is a "transparent" automaton, because a meaningful change on its inputs goes directly to its output.

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 codded binary, because a real machine work with bits, 0 and 1, not with symbols.

The output is already codded 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: $a \to X_0 = 0$ and $b \to X_0 = 1$.

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:

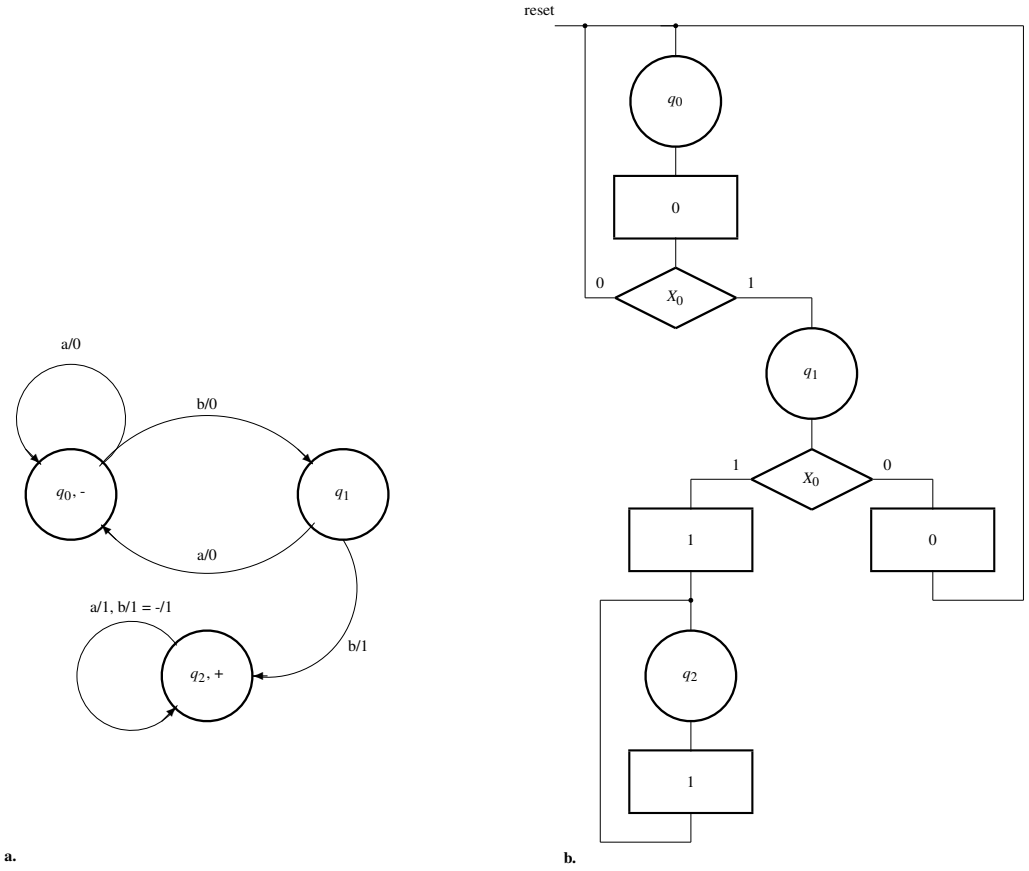


Figure 8.11: **Example of flow-chart for a Mealy automaton.** The occurrence of the second *b* from *bb* is detected as fast as possible.

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 8.7 *The flow-chart from Figure 8.9 has two different correspondent representations as transition diagrams in Figure 8.12, one for the option 1 of coding (Figure 8.12a), and another for the option 2 (Figure 8.12b).*

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:

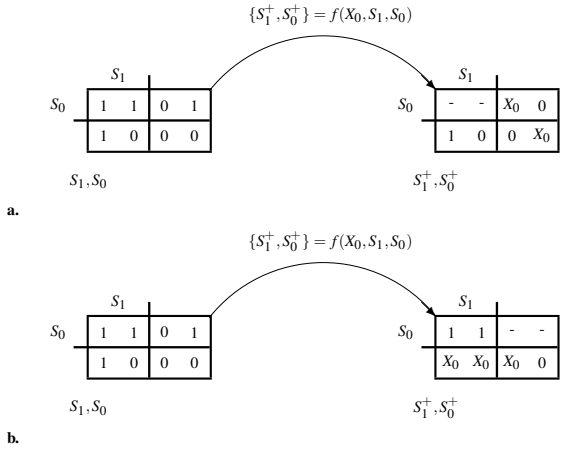


Figure 8.12: **Example of transition diagram for a half-automaton. a.** For the option 1 of coding. **b.** For the option 2 of coding.

- *from the state codded 00 the automaton switch in the state* 0*x, 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*)*
- *from the state codded 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, S_0 *(defined in the reference VKD), then in the next clock cycle,* T_{i+1}*, the automaton switches in the state* S_1^+, S_0^+ *(defined in the next state VKD).*

For the second coding option:

- *from the state codded 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 codded 10 the automaton switch in the state* X_0X_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 codded 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 8.8 *The Boolean function of the half-automaton working as "double b detector" can be extracted from the transition diagram represented in Figure 8.12a (for the first coding option). Results:*

$$S_1^+ = S_1 + X_0 S_0$$

$$S_0^+ = X_0 S_1' S_0'$$

◇

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 8.9 *The flow-chart from Figure 8.10 have a correspondent representation in the transition diagrams from Figure 8.13a or Figure 8.13b. 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).*

◇

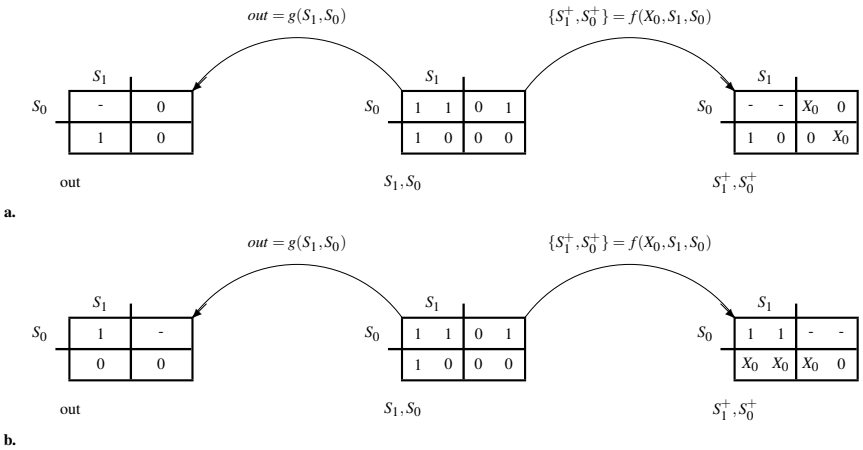


Figure 8.13: **Example of transition diagram for a Moore automaton.**

Example 8.10 *The resulting output function is:*

$$out = S_1.$$

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 8.14, 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 `reset = 1` *the two flip-flops switch in 0. When* `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.* ◇

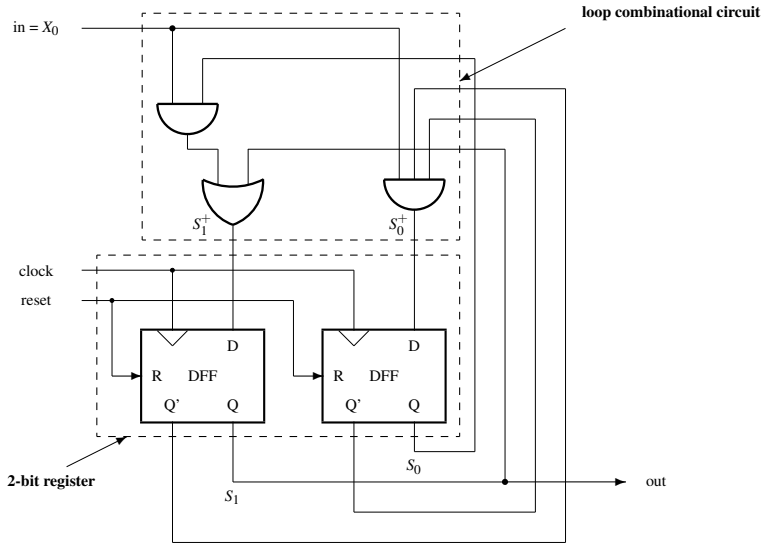


Figure 8.14: **The Moore version of "bb detector" automaton.**

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 8.11 *Revisiting the same problem result, in Figure 8.15 the transition diagrams associated to the flow-chart from Figure 8.11.*

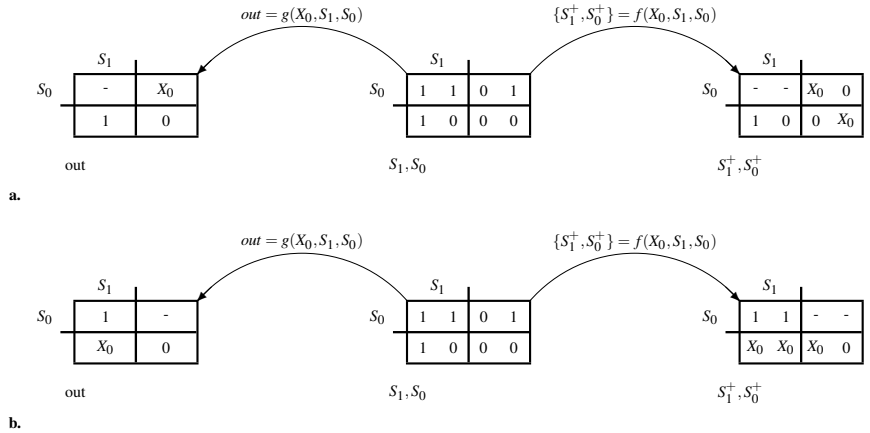


Figure 8.15: **Example of transition diagram for a Mealy automaton.**

The two functions f are the same. The function g is defined for the first coding option (Figure 8.15a) as follows:

- *in the state coded by 00 (q_0) the output takes value 0*
- *in the state coded by 01 (q_1) the output takes value x*
- *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 `out` *results:*

$$out = S_1 + X_0 S_0$$

a more complex from compared with the Moore version. (But fortunately $out = S_1^+$, 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 8.12 *The same "bb detector" is used to exemplify the procedures used for the Moore automaton representation.*

```
/*****************************************************************
File name:      moore_automaton.v
Circuit name:   An example of Moore-type automaton
Description:    behavioral description of the Moore finite automaton
                designed to detect 'bb' in a stream of symbols belonging
                to the set {a,b}
*****************************************************************/
 module moore_automaton(out, in, reset, clock);
// input codes
    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;
            endcase
// g: the output combinational transition function
    always @(state) case(state)
                        init_state  : out = no  ;
                        one_b_state : out = no  ;
                        final_state : out = yes ;
                        default     : out = 1'bx;
                    endcase
 endmodule
```

For the delayed version there is the following code:

```
/*********************************************************************
File name:      moore_delayed_automaton.v
Circuit name:   An example of Moore-type automaton
Description:    behavioral description of the delayed Moore finite
                automaton designed to detect 'bb' in a stream of symbols
                belonging to the set {a,b}
*********************************************************************/
module moore_delayed_automaton(out, in, reset, clock);
// input codes
    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 register
// 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;
            endcase
// g: the delayed transition function
    always @(posedge clock) case(state)
                                init_state  : out <= no ;
                                one_b_state : out <= no ;
                                final_state : out <= yes;
                            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 8.13 *The same "bb detector" is used to exemplify the procedures used for the Mealy automaton representation.*

```
/*****************************************************************
File name:      mealy_automaton.v
Circuit name:   An example of Mealy-type automaton
Description:    behavioral description of the Mealy finite automaton
                designed to detect 'bb' in a stream of symbols belonging
                to the set {a,b}
*****************************************************************/
 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;
            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
```

For the delayed version:

```
/*****************************************************************
File name:      mealy_delayed_automaton.v
Circuit name:   An example of Mealy-type automaton
Description:    behavioral description of the Mealy finite automaton
                designed to detect 'bb' in a stream of symbols belonging
                to the set {a,b}
*****************************************************************/
 module mealy_delayed_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  reg out;
    reg [1:0]   state;
    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;
            endcase
    always @(posedge clock)
        case(state)
            init_state  :               out <= no   ;
            one_b_state : if (in == b)  out <= yes  ;
                          else          out <= no   ;
            final_state :               out <= yes  ;
        endcase
 endmodule
```

◇

The procedural representations are used as inputs for automatic design tools.

8.4.2 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 this 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 8.14 *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 8.16, where:*

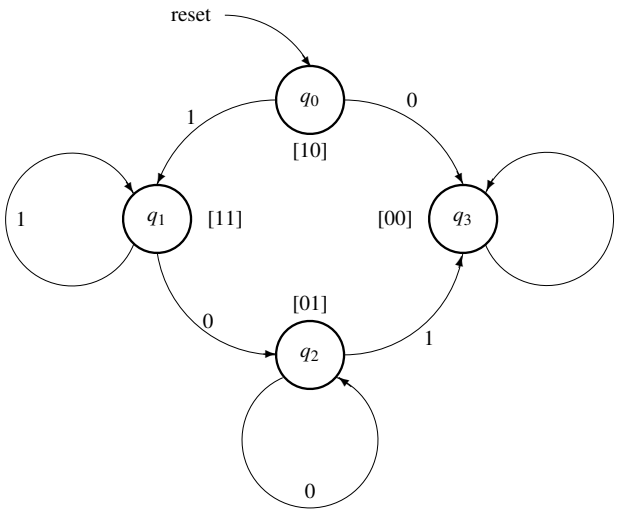


Figure 8.16: **Transition diagram.** The transition diagram for the half-automaton which recognizes strings of form 1^n0^m, for $n \geq 1$ and $m \geq 1$. Each circle represent a state, each (marked) arrow represent a (conditioned) transition.

- 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* acknowledge*s 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 int square brackets in Figure 8.16. Results the transition diagram presented in Figure 8.17. The resulting transition functions are:

$$Q_1^+ = Q_1 \cdot X_0 = ((Q_1 \cdot X_0)')'$$

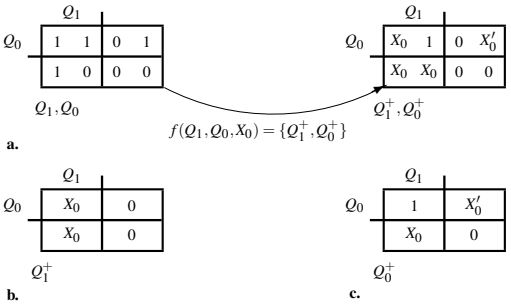


Figure 8.17: **VK transition maps.** The VK transition map for the half-automaton used to recognize 1^n0^m, for $n \geq 1$ and $m \geq 1$. **a**. The state transition function f. **b**. The VK diagram for the next most significant state bit, extracted from the previous full diagram. **c**. The VK diagram for the next least significant state bit.

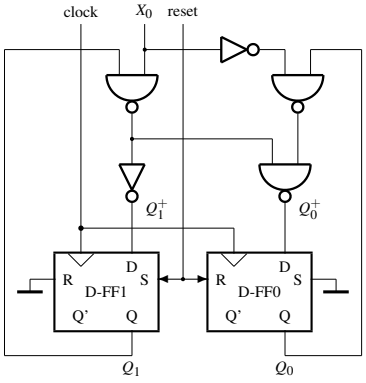


Figure 8.18: **A 4-state finite half-automaton.** The structure of the finite half-automaton used to recognize binary string belonging to the 1^n0^m set of strings.

$$Q_0^+ = Q_1 \cdot X_0 + Q_0 \cdot X_0' = ((Q_1 \cdot X_0)' \cdot (Q_0 \cdot X_0'))'$$

(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 8.46 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:

```
/*****************************************************************
File name:      rec_aut.v
Circuit name:   Recognizing Automaton for streams of form a^nb^m
Description:    behavioral description of the automaton used to recognize
                streams of symbols of form a^nb^m
*****************************************************************/
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 8.15 *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:*

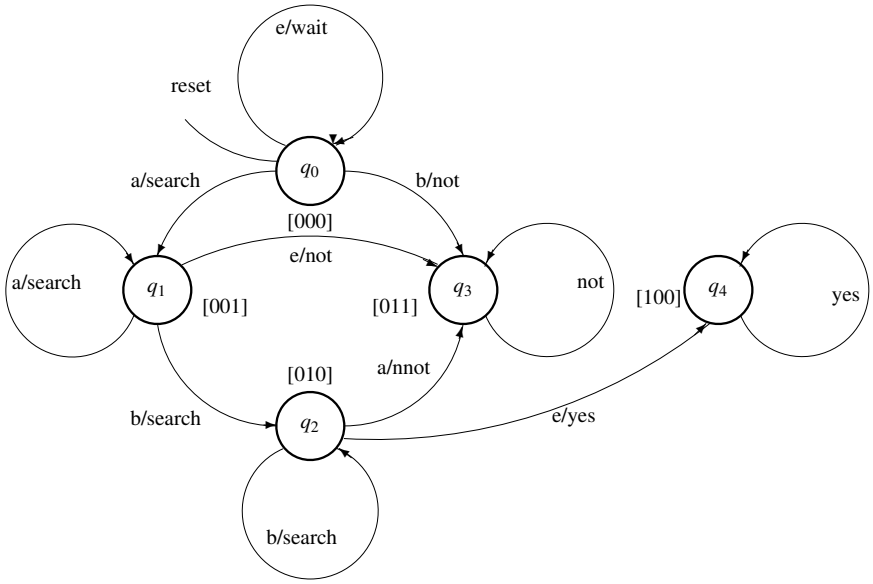


Figure 8.19:

$\ldots eeaa \ldots abb \ldots bee \ldots$

The stream is considers 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 8.19. The automaton has the following 5 states:

q2	q1	q0	x1	x0	q2+	q1+	q0+	y1	y0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1	1	1
0	0	0	1	0	0	1	1	0	1
0	0	0	1	1	-	-	-	-	-
0	0	1	0	0	0	1	1	0	1
0	0	1	0	1	0	0	1	1	1
0	0	1	1	0	0	1	0	1	1
0	0	1	1	1	-	-	-	-	-
0	1	0	0	0	1	0	0	1	0
0	1	0	0	1	0	1	1	0	1
0	1	0	1	0	0	1	0	1	1
0	1	0	1	1	-	-	-	-	-
0	1	1	0	0	0	1	1	0	1
0	1	1	0	1	0	1	1	0	1
0	1	1	1	0	0	1	1	0	1
0	1	1	1	1	-	-	-	-	-
1	0	0	0	0	1	0	0	1	0
1	0	0	0	1	1	0	0	1	0
1	0	0	1	0	1	0	0	1	0
1	0	0	1	1	-	-	-	-	-
1	0	1	0	0	-	-	-	-	-
⋮	⋮	⋮	⋮	⋮	-	-	-	-	-

Table 8.1: The truth table for the transition functions.

q_0 *: the initial state in which the automaton goes by* `reset`*, and if*

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* $1^n0^m | n,m \geq 1$

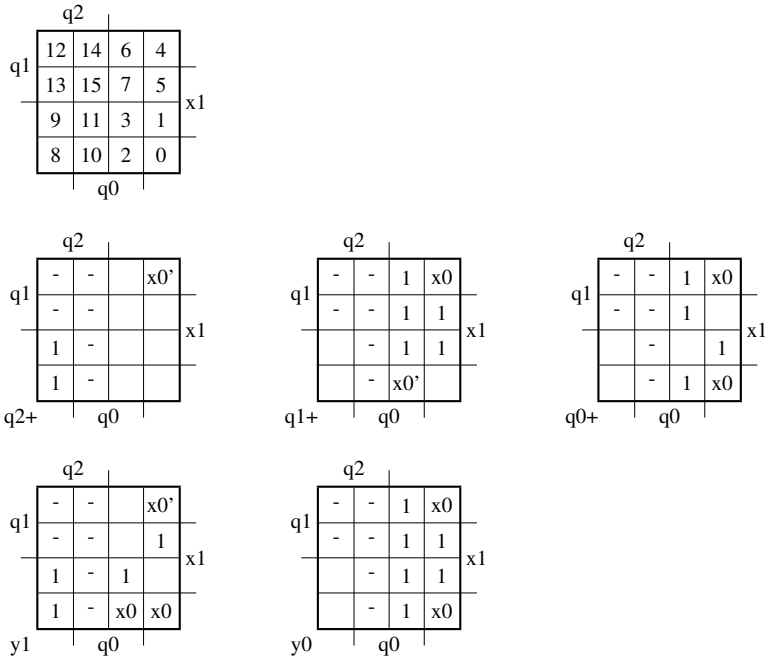


Figure 8.20: The V-K diagrams for the state and output transition functions.

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 codded as follows:

```
X = {a, b, e} = {01, 10, 00}
Y = {wait, search, not, yes} = {00, 11, 01, 10}
Q = {q0, q1, q2, q3, q4} = {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.

We have to solve 5 functions of 5 variables. Let us use V-K diagrams for 4 variables (`q2, q1, q0, x1`*) and the 5th variable,* `x0`*, will be used to define the value of some boxes belonging to the diagrams. In Figure 8.20, 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* `q2+` *is built:*

- *in the box 0 is filled with 0, because for* {`q2, q1, q0, x1`} = {`0 0 0 0`} *the output* `q2+` *does not depend on* `x0` *and takes the value 0*
- *in the box 1 in filled with 0, because for* {`q2, q1, q0, x1`} = {`0 0 0 1`} *the output* `q2+` *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* `x0'`*, because for* {`q2, q1, q0, x1`} = {`0 1 0 0`} *the output* `q2+` *takes the value 1, if* `x0 = 0` *and the value 0 if* `x0 = 1`

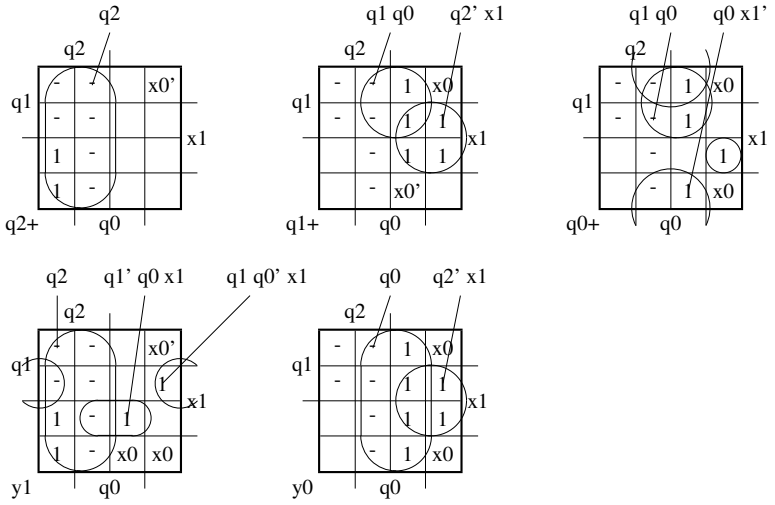


Figure 8.21: The first stage in the extracting algebraic expressions from V-K diagrams: the functions included in diagrams are ignored.

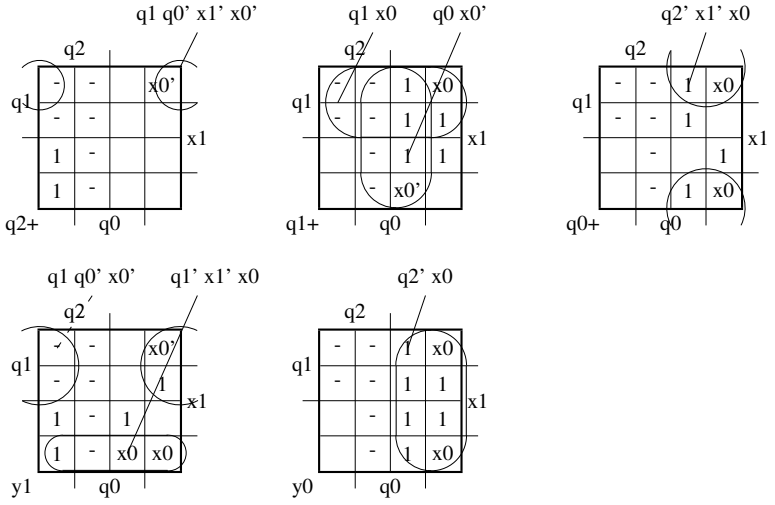


Figure 8.22: The second stage in the extracting algebraic expressions from V-K diagrams: the 1s are considered “don’t care”s.

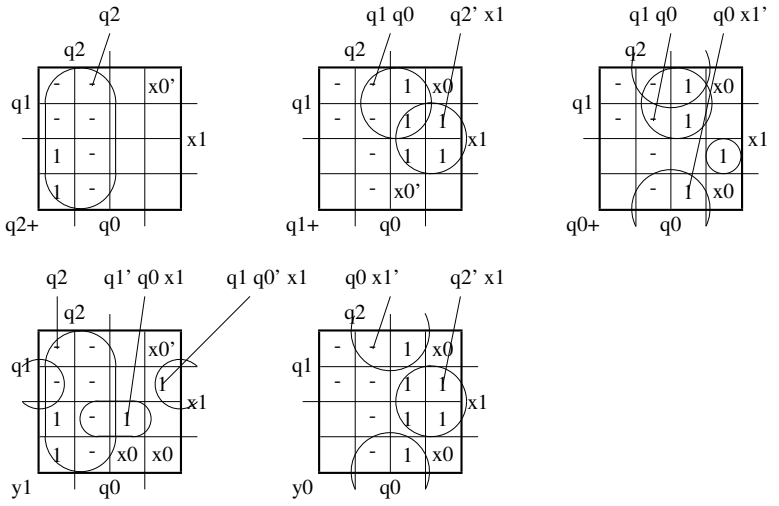


Figure 8.23: The first stage in the extracting algebraic expressions from V-K diagrams.

- *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* {q2, q1, q0, x1} = {1 0 0 0} *the output* q2+ *does not depend on* x0 *and takes the value 1*
- *in the box 9 in filled with 1, because for* {q2, q1, q0, x1} = {1 0 0 1} *the output* q2+ *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 care*s*

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 8.21. The second stage (which considers the 1s as "don't care"s) is represented in Figure 8.22 The resulting expressions are the following:

```
q2+ = q2 + q1 q0' x1' x0'
q1+ = q2' x1 + q1 q0 + q0 x0' + q1 x0
q0+ = q1 q0 + q0 x1' + q2' q1' q0' x1 + q2' x1' x0
y1 = q2 + q1'q0 x1 + q1 q0' x1 + q1 q0' x0' + q1' x1' x0
y0 = q0 + q2' x1 + q2'x0
```

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.

In Figure 8.23 the diagram for y0 *is modified: instead of the surface* q0*, emphasize in Figure 8.21, here we have a smaller one,* q0 x1'*, because this surface is selected also in the diagram for* q0+*. The impact on the final circuit is minimal: the fan-out of the D-FF0 is reduced.*

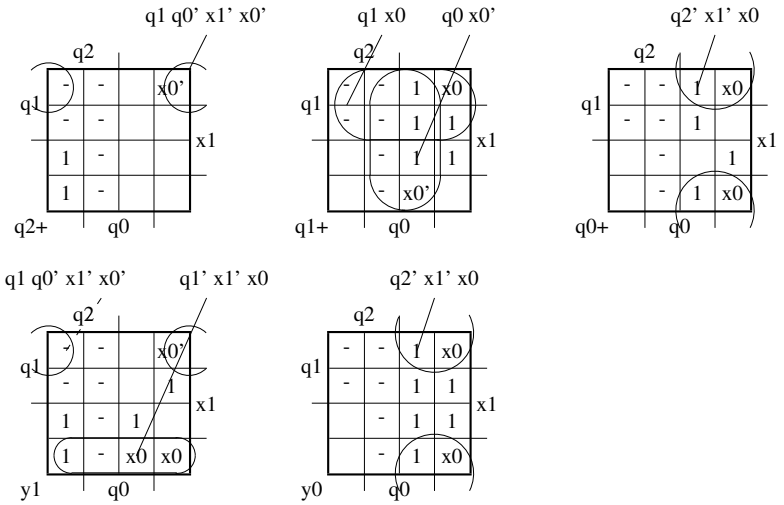


Figure 8.24: The second stage in the extracting algebraic expressions from V-K diagrams.

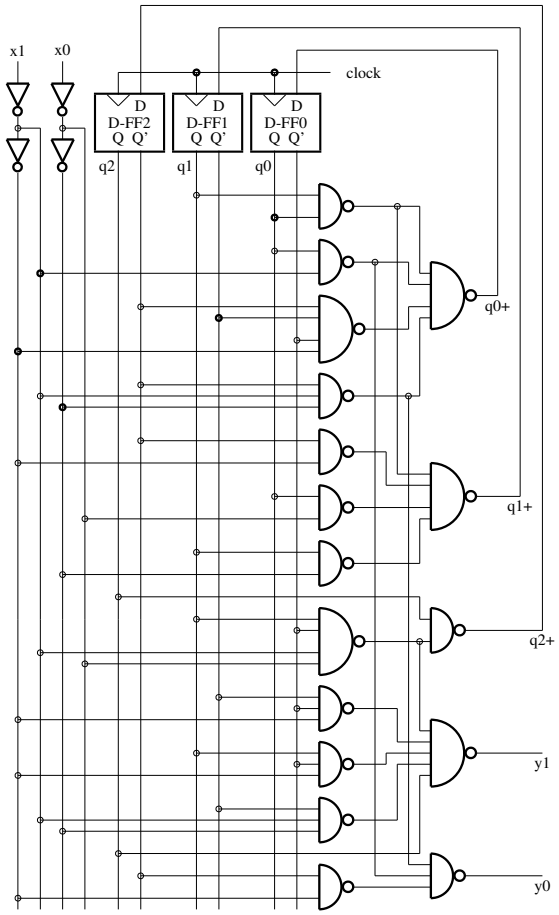


Figure 8.25: The resulting circuit.

The impact of this approach in the second stage is more important: the NAND circuit for `q2' q1' x0` *is shared for the implementation of* `q0+` *and* `y0`*, and the NAND circuit for* `q1 q0' x1' x0'` *is shared for the implementation of* `q2+` *and* `y1`*.*

The resulting expressions are (with various brackets are emphasized the shared logic products):

```
q2+ = q2 + [q1 q0' x1' x0']
q1+ = q2' x1 + <q1 q0> + q0 x0' + q1 x0
q0+ = <q1 q0> + (q0 x1') + q2' q1' q0' x1 + {q2' x1' x0}
y1 = q2 + q1'q0 x1 + q1 q0' x1 + [q1 q0' x1' x0'] + q1' x1' x0
y0 = (q0 x1') + q2' x1 + {q2' x1' x0}
```

In Figure 8.25 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 codding 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 8.16 *Let us revisit the previous example using another state coding:*

```
Q = {q0, q1, q2, q3, q4} = {000, 001, 111, 011, 010}
```

Then, the Table **??** *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 8.26

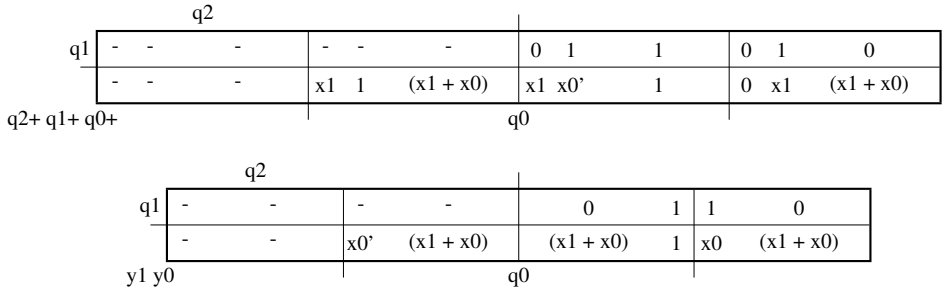


Figure 8.26:

From V-K diagrams result the following expressions :

```
q2+ = q1' q0 x1
q1+ = q2 + q1 + q0 x0' + q0' x1
q0+ = q2' q0 + q1 (x1 + x0)
y1 = q1 q0' + q2 x0' + q0' x0 + q2' q1' q0 (x1 + x0)
y0 = q2' q0 + q1' (x1 + x0) = q0+
```

The resulting circuit is represented in Figure 8.27

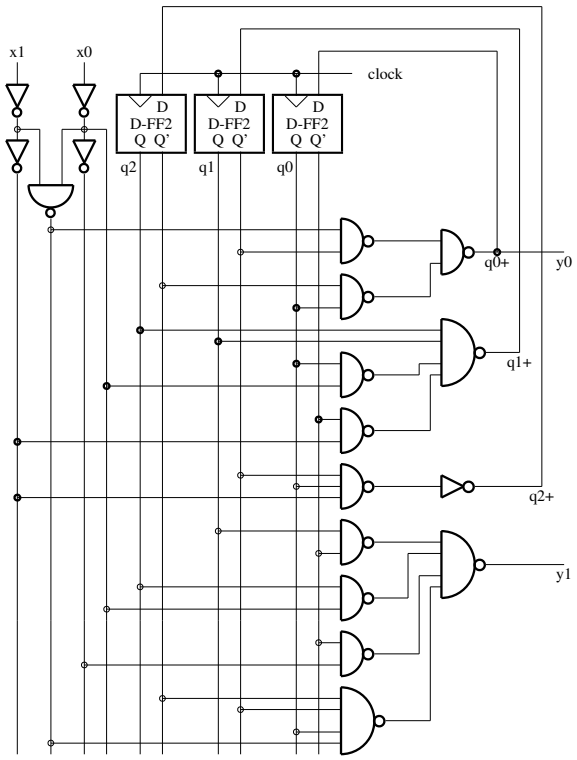


Figure 8.27: The circuit for the codding dominated by the reduce dependency coding style.

The size of the combinational circuits is only 70% from the previous solution. This reduction was obtained only by changing the state coding.

◇

Example 8.17 ∗ *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 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 8.28). The flow-chart describing the version for performing multiplications is presented in Figure 8.29, 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 8.30) or in a Verilog description. From the VK transition maps result the following equations describing the combinational circuits for the loop ($q1^+, q0^+$) and for the outputs.

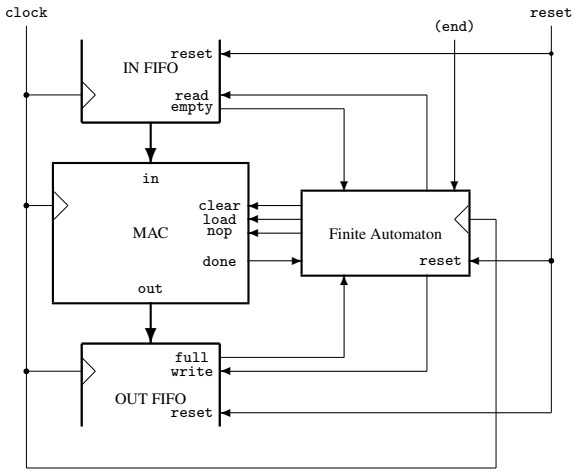


Figure 8.28: **The Multiply-Accumulate System.** The system consists in a multiply-accumulate circuit (MAC), two FIFOs and a finite automaton (FA) controlling all of them.

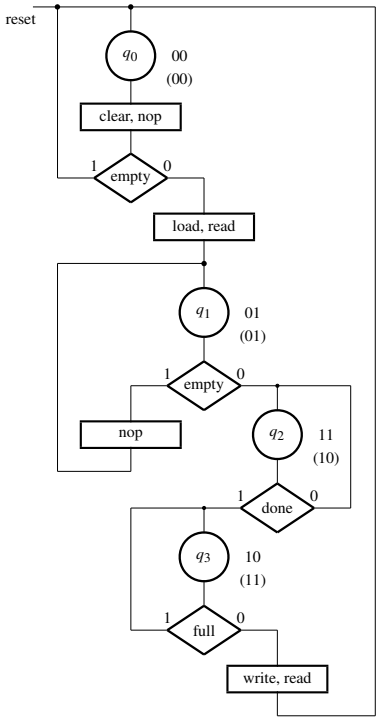


Figure 8.29: **Flow chart describing a Mealy finite automaton.** The flow-chart describes the finite automaton FA from Figure 8.28, which controls MAC and the two FIFOs in MAC system. (The state coding shown in parenthesis will be used in the next chapter.)

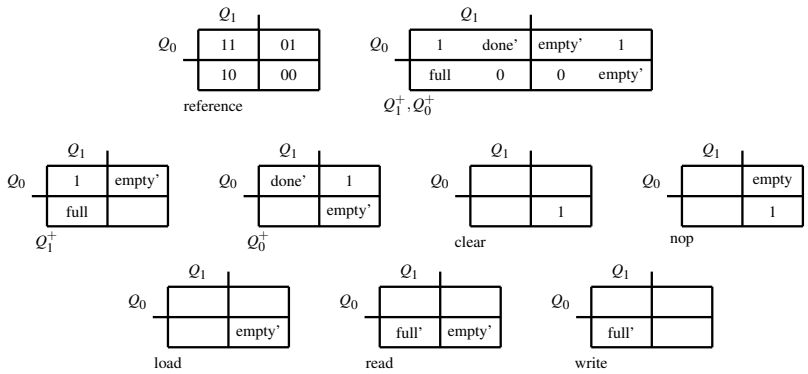


Figure 8.30: **Veitch-Karnaugh transition diagrams.** The transition VK diagrams for FA (see Figure 8.28). The *reference* diagram has a box for each state. The state transition diagram, $Q_1^+ Q_1^+$, contains in the same positions the description of the next state. For each output a diagram describe the output's behavior in the corresponding state.

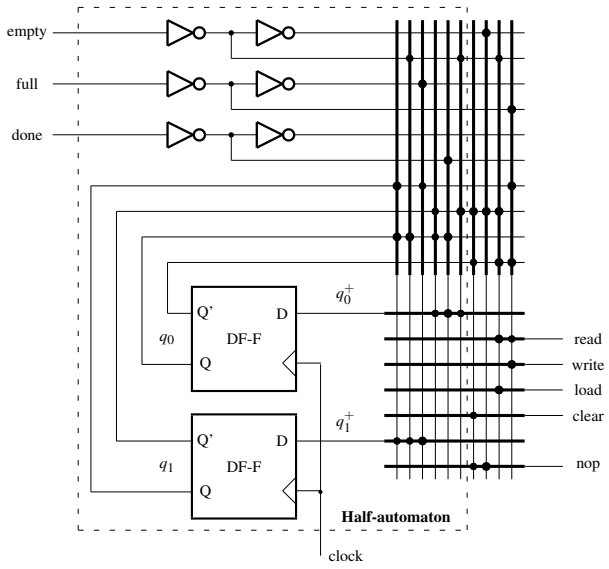


Figure 8.31: **FA's structure.** The FA is implemented with a two-bit register and a PLA with 5 input variables (2 for state bits, and 3 for the input sibnals), 7 outputs and 10 products.

$$Q_1^+ = Q_1 \cdot Q_0 + Q_0 \cdot empty' + Q_1 \cdot full$$
$$Q_0^+ = Q_1' \cdot Q_0 + Q_0 \cdot done' + Q_1' \cdot empty'$$
$$clear = Q_1' \cdot Q_0'$$
$$nop = Q_1' \cdot Q_0' + Q_1' \cdot empty$$
$$load = Q_1' \cdot Q_0' \cdot empty'$$
$$read = Q_1 \cdot Q_0' \cdot full' + Q_1' \cdot Q_0' \cdot empty'$$
$$write = Q_1 \cdot Q_0' \cdot full'$$

The resulting circuit is represented in Figure 8.31, 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.

```
/*****************************************************************
File name:      macc_control.v
Circuit name:   Multiply & Accumulate Control automaton
Description:    behavioral description of the automaton used to control a
                of FIFOs used to feed and discard a MACC unit
*****************************************************************/
module macc_control(output  read    ,   // read from IN FIFO
                    output  write   ,   // write in OUT FIFO
                    output  load    ,   // load the multiplier in MAC
                    output  clear   ,   // reset the output of MAC
                    output  nop     ,   // stops the multiplication
                    input   empty   ,   // IN FIFO is empty
                    input   full    ,   // OUT FIFO is full
                    input   done    ,   // multiplication ended
                    input   reset   , clock   );
    reg [1:0]   state;
    reg         read, write, load, clear, nop;  // as variables
    parameter   wait_first      = 2'b00,
                wait_second     = 2'b01,
                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 @(*)
     case(state)
      wait_first  : if (empty) {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
endmodule
```

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 codding 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.

```
/*****************************************************************
File name:      macc_control.v
Circuit name:   Multiply & Accumulate Control automaton
Description:    a more detailed description of the automaton used to
                control a of FIFOs used to feed and discard a MACC unit
*****************************************************************/
module macc_control(output read    ,   // read from IN FIFO
                    output write   ,   // write in OUT FIFO
                    output load    ,   // load the multiplier in MAC
                    output clear   ,   // reset the output of MAC
                    output nop     ,   // stops the multiplication
                    input  empty   ,   // IN FIFO is empty
                    input  full    ,   // OUT FIFO is full
                    input  done    ,   // the multiplication is concluded
                    input  reset   , clock);
    reg [1:0]   st; // state register
// THE STATE TRANSITION FUNCTION
    always @(posedge clock)
     if (reset) st <= 2'b00;
      else st <= {(st[1] & st[0] | st[0] & ~empty  | st[1] & full)  ,
                  (~st[1] & st[0] | st[0] & ~done  | ~st[1] & ~empty)};
    assign  read    = ~st[1] & ~st[0] & ~empty | st[1] & ~st[0] & ~full,
            write   = st[1] & ~st[0] & ~full                           ,
            load    = ~st[1] & ~st[0] & ~empty                         ,
            clear   = ~st[1] & ~st[0]                                  ,
            nop     = ~st[1] & ~st[0] | ~st[1] & empty                 ;
endmodule
```

The resulting circuit will be eventually an optimized form of the version represented in Figure 8.31 because instead a PLA, the current tools use an minimized network of gates.

For delayed Mealy version the code is:

```
/*****************************************************************
File name:      macc_control.v
Circuit name:   Multiply & Accumulate Control automaton
Description:    behavioral description of the delayed automaton used to
                control a of FIFOs used to feed and discard a MACC unit
*****************************************************************/
module macc_delayed_control
        (output  reg read    , // read from IN FIFO
         output  reg write   , // write in OUT FIFO
         output  reg load    , // load the multiplier in MAC
         output  reg clear   , // reset the output of MAC
         output  reg nop     , // stops the multiplication
         input       empty   , // IN FIFO is empty
         input       full    , // OUT FIFO is full
         input       done    , // multiplication ended
         input       reset   , clock   );
    reg [1:0]   state;
    parameter   wait_first      = 2'b00,
                wait_second     = 2'b01,
                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 (DELAYED MEALY)
    always @(posedge clock)
     case(state)
      wait_first  : if (empty) {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
endmodule
```

◇

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 8.18 *Let be a first example showing the structural dependency by the state encoding. The automaton described in Figure 8.32a 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 8.32b:*

$$Q_1^+ = Q_0 + X_0 Q_1'$$

$$Q_0^+ = Q_1' Q_0' X_0'$$

$$Y_1 = Q_0 + X_0 Q_1'$$

$$Y_0 = Q_1' Q_0'.$$

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 8.32b:*

$$Q_1^+ = Q_1' Q_0 + X_0 Q_1' = (Q_1 + (Q_0 + X_0)')'$$

$$Q_0^+ = Q_1'$$

$$Y_1 = Q_1' Q_0 + X_0 Q_1' = (Q_1 + (Q_0 + X_0)')'$$

$$Y_0 = Q_0'.$$

Obviously the second codding version provides a simpler and smaller combinational circuit associated to the same external behavior. In Figure 8.33 the resulting circuit is represented. $\diamond$

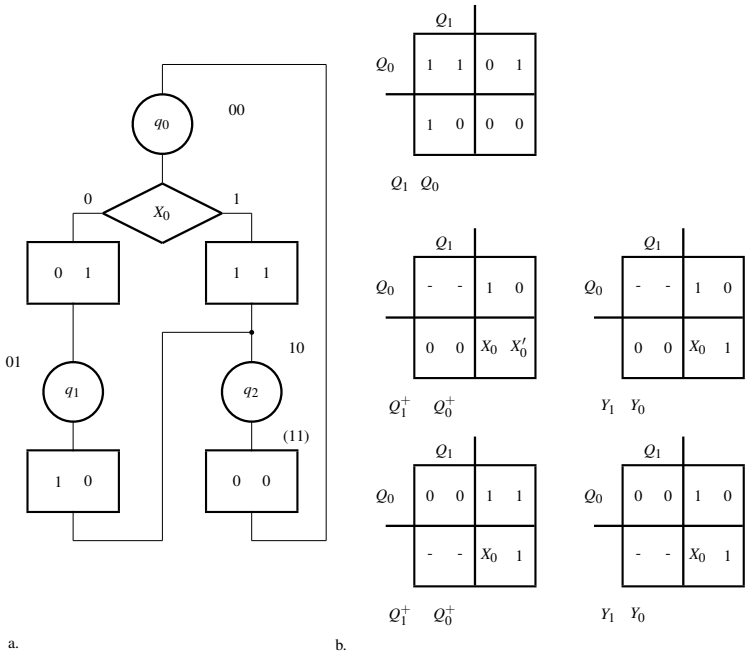


Figure 8.32: **A 3-state automaton with two different state encoding. a.** The flow-chart describing the behavior. **b.** The VK diagrams used to implement the automaton: the reference diagram for states, two transition diagrams used for the first code assignment, and two for the second state assignment.

Minimal variation encoding Minimal variation state assignment (or encoding) refers to the codes assigned to successive states.

Definition 8.13 *Codding with minimal variation means successive state are codded with minimal Hamming distance.* ◇

Example 8.19 *Let be the fragment of a flow chart represented in Figure 8.34a. 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 8.20 *Some times the minimal variation encoding is not possible. An example is presented in Figure 8.34b, where q_k can not be codded with minimal variation.* ◇

The minimal variation codding 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.

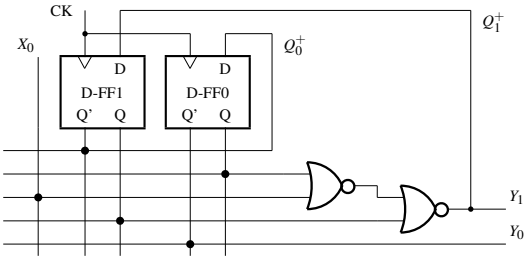


Figure 8.33: **The resulting circuit** It is done for the second state assignment of the automaton defined in Figure 8.32a.

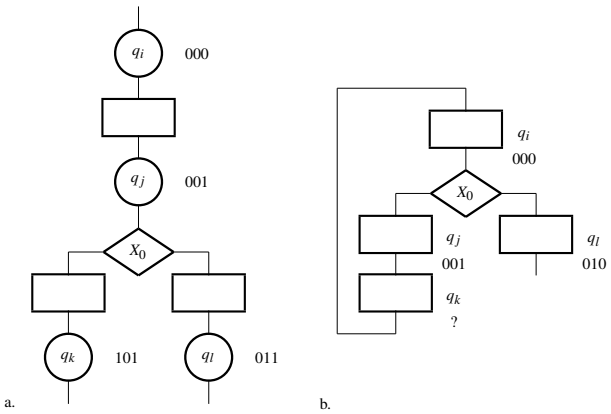


Figure 8.34: **Minimal variation encoding.** **a.** An example. **b.** An example where the minimal variation encoding is not possible.

Definition 8.14 *Reduced dependency encoding means the states which conditionally follow a certain state to be codded with binary configurations which differs minimal (have the Hamming distance minimal).* ◇

Example 8.21 *In Figure 8.35a 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 8.22 *In Figure 8.35b the transition from the state q_i depends by two 1-bit variable, X_0 and X_1. A reduced dependency codding is possible by only one of them. Without parenthesis is a reduced dependency codding by the variable X_1. With parenthesis is a reduced dependency codding by X_0.* ◇

The reader is invited to provide the proof for the following theorem.

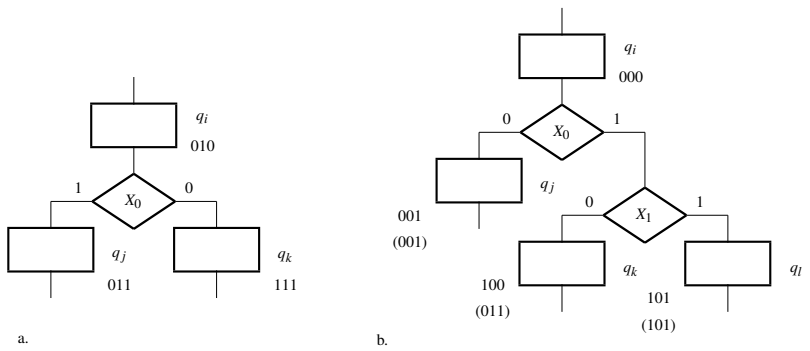


Figure 8.35: **Examples of reduced dependency encoding. a.** The transition from the state is conditioned by the value of a single 1-bit variable. **b.** The transition from the state is conditioned by two 1-bit variables.

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.* $\diamond$

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 codding 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 8.15 *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.* $\diamond$

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 codding stile 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 8.16 *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.* $\diamond$

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 procedure 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 codding 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 codding 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 codding 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 codded 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 8.23 *Let be the finite automaton described by the flow-chart from Figure 8.36. Are proposed two codding versions, a good one (the first), using the codding rules previously listed, and a bad one (the second with the codes written in parenthesis), ignoring the rules.*

For the first codding version results the expressions:

$$Q_2^+ = Q_2Q_0' + Q_2'Q_1$$

$$Q_1^+ = Q_1Q_0' + Q_2'Q_1'Q_0 + Q_2'Q_0X_0$$

$$Q_0^+ = Q_0' + Q_2'Q_1'X_0'$$

$$Y_2 = Q_2 + Q_1Q_0$$

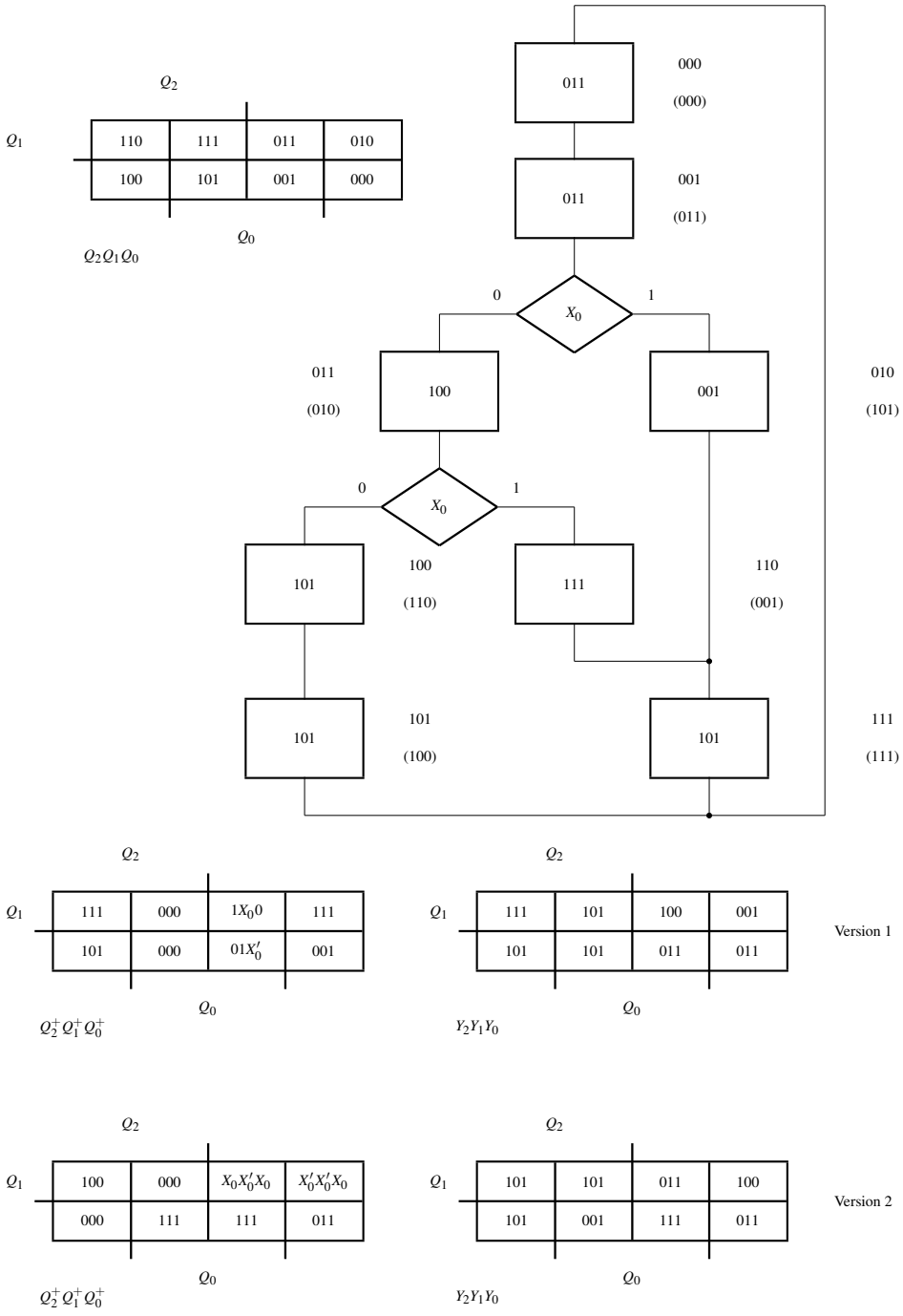


Figure 8.36: **Minimizing the structure of a finite automaton.** Applying appropriate codding rules the occurrence of the input variable X_0 in the transition diagrams can be minimized, thus resulting smaller Boolean forms.

$$Y_1 = Q_2Q_1Q_0' + Q_2'Q_1'$$

$$Y_0 = Q_2 + Q_1' + Q_0'$$

the resulting circuit having the size $S_{CLCver1} = 37$.

For the second codding version results the expressions:

$$Q_2^+ = Q_2Q_1Q_0' + Q_1'Q_0 + Q_2'Q_0X_0 + Q_1Q_0'X_0'$$

$$Q_1^+ = Q_1'Q_0 + Q_2'Q_1' + Q_2'X_0'$$

$$Q_0^+ = Q_1'Q_0 + Q_2'Q_1' + Q_2'X_0$$

$$Y_2 = Q_2Q_0' + Q_2Q_1 + Q_2'Q_1'Q_0 + Q_1Q_0'$$

$$Y_1 = Q_2'Q_0 + Q_2'Q_1'$$

$$Y_0 = Q_2 + Q_1' + Q_0$$

the resulting circuit having the size $S_{CLCver2} = 50$. $\diamond$

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 8.24 *Let be the automaton described by the flow-chart from Figure 8.37, for which two codding version are proposed: a one-hot encoding using 6 bits* ($Q_6 \ldots Q_1$)*, and a compact binary encoding using only 3 bits* ($Q_2Q_1Q_0$)*.*

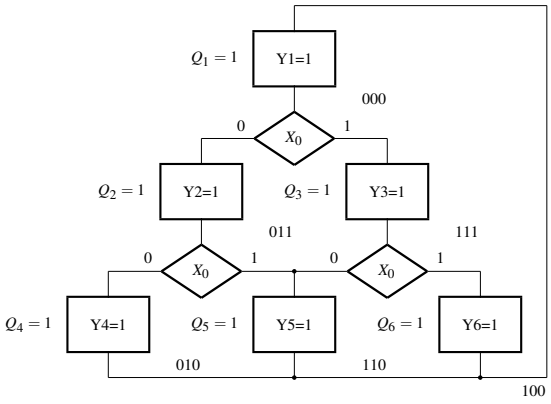


Figure 8.37: Minimizing the complexity using one-hot encoding.

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^+, $i = 1, \ldots, Q_6^+$, can be written directly inspecting the definition. Results:*

$$Q^+{}_1 = Q_4 + Q_5 + Q_6$$
$$Q^+{}_2 = Q_1 X_0'$$
$$Q^+{}_3 = Q_1 X_0$$
$$Q^+{}_4 = Q_2 X_0'$$
$$Q^+{}_5 = Q_2 X_0 + Q_3 X_0'$$
$$Q^+{}_6 = Q_3 X_0$$

Because in each state only one output bit is active, results:

$$Y_i = Q_i, \ \ pentru \ \ i = 1, \ldots, 6.$$

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 $S_{CLC,var1} = 18$, 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 codding *The state transition functions for this codding version (see Figure 8.37 for the actual binary codes) are:*

$$Q^+{}_2 = Q_2 Q_0 + Q_0 X_0 + Q_2' Q_1' X_0$$
$$Q^+{}_1 = Q_2' Q_0 + Q_2' Q_1 + Q_0 X_0'$$
$$Q^+{}_0 = Q_2' Q_1'$$

For the output transition function an additional decoder, DCD_3, is needed. The resulting combinational circuit has the size $S_{CLC,var2} = 44$, with the additional disadvantage of generating the outputs signal using a combinational circuit, the decoder. $\diamond$

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_H 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 $1AA0$ (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 codding for the transition from the state in which X_0^* is tested. If the state assignment will allow, for example, a transition to $11X_00$, 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 8.25 *In Figure 8.38 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:

$$Q_1^+ = Q_0' + Q_1' X_0$$

$$Q_0^+ = Q_1 + Q_0 X_0.$$

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. $\diamond$

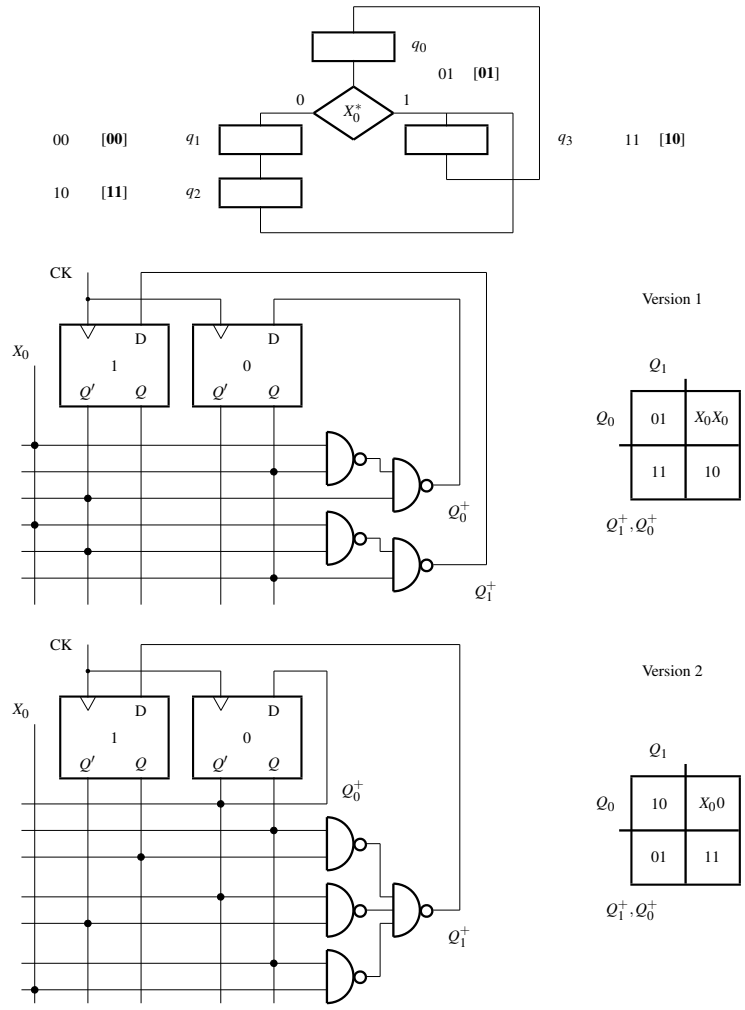


Figure 8.38: **Implementing a finite automaton with an asynchronous input.**

Obviously, transition determined by more than one asynchronous variable must be avoided, because, as we already know, the reduced dependency codding 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 tray to synchronize the two variable, A^* and B^*, using a register. This attempt is represented in

Figure 8.39. But, unfortunately, this solution does'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 $10 \rightarrow 01 \rightarrow 01$.

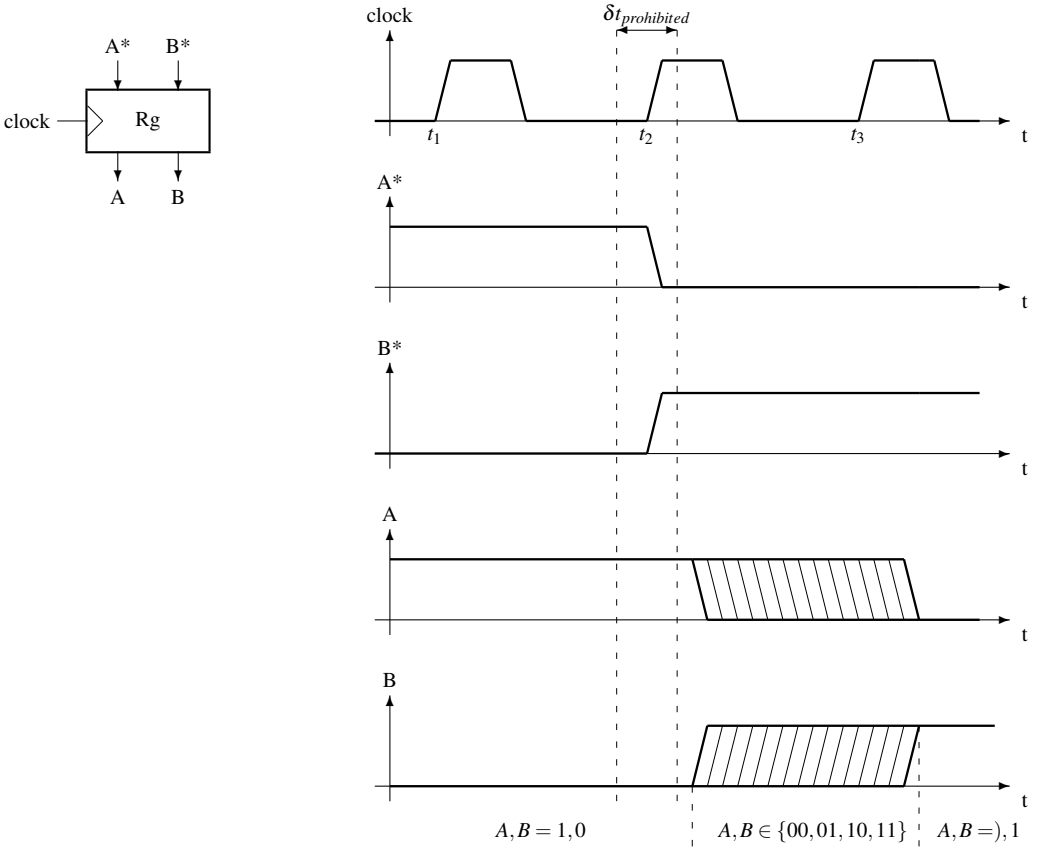


Figure 8.39:

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 8.26 *Let be the fragment of flowchart from Figure 8.40a, where two asynchronous variable, A^* and B^*, are tested. An additional state is added in Figure 8.40b. In this new state the output is the same with the first state.*

◇

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

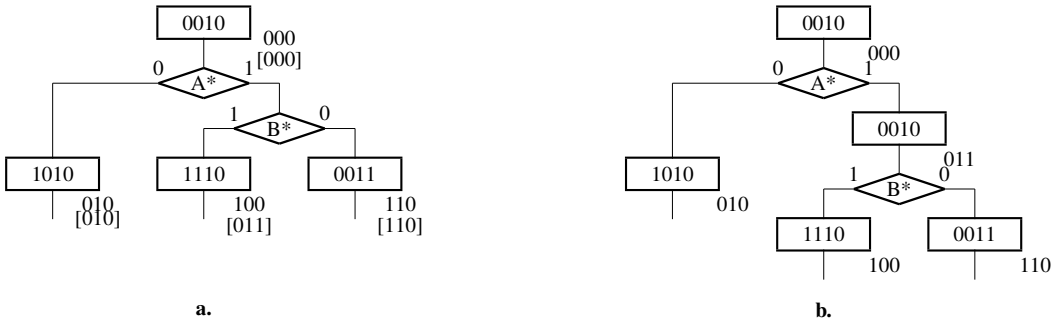


Figure 8.40: Reduce dependency coding. **a.** The coding is possible only related to one asynchronous variable. The first version is according to B^*, while the second (in in square brackets) is according to A^*. **b.** The second version added a new state, doubling the first state.

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 o gates and the effect of an input switch reaches the output going trough different logic path. The propagation trough 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 8.41a representing the typical gates receiving the signal A and B, ideally represented in Figure 8.41b. Ideally means the two signals switches synchronously. They are considered ideal because no synchronous signal can be actually generated. In Figure 8.41c and Figure 8.41d 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 asynchronnicity between the input signals some parasitic transitions occur to the outputs of the three gates (see Figure 8.41c and Figure 8.41d). 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.

In order to offer an additional explanation for this kind of hazard VK diagrams are used in Figure 8.42, 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

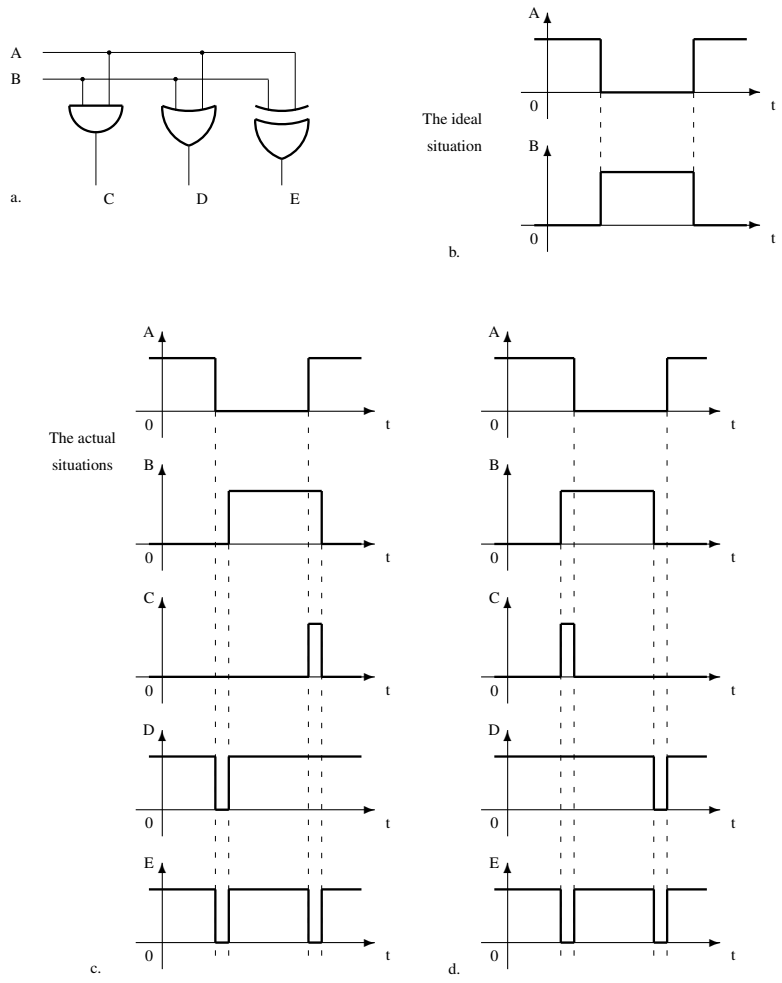


Figure 8.41: How the asynchronous inputs generate hazard.

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.

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 8.43a, where two input are stable ($A = C = 1$) and *only one* input switches. The problem of asynchronoous inputs is not an issue because only one input is in transition. In Figure 8.43b 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

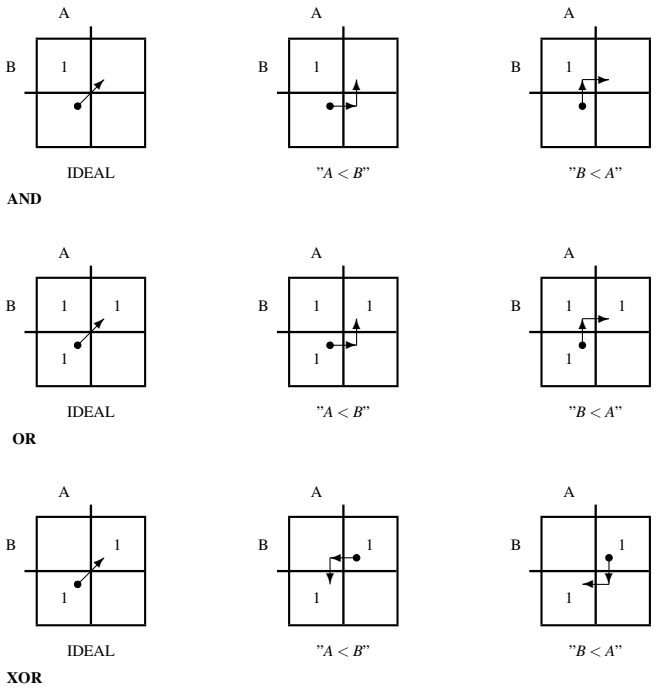


Figure 8.42: VK diagrams explaining the hazard due to the asynchronous inputs. "$A < B$" means the input A switch before the input B, and "$A > B$" means the input B switch before the input A.

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 8.43c) 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 8.43d) 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 8.43e.

Example 8.27 *Let be the function presented in VK diagram from Figure 8.44a. An immediate solution is shown in Figure 8.44b, 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 $A'BCD'$ and $A'B'CD'$ are adjacent, and the same for $AB'C'D$ and $A'B'C'D$. Therefore, the solution to completely avoid the hazard is presented in Figure 8.44c, where two additional surfaces are added.* $\diamond$

Theorem 8.3 *If the expression of the Boolean function*

$$f(x_{n-1}, \ldots x_0)$$

takes the form

$$x_i + x_i'$$

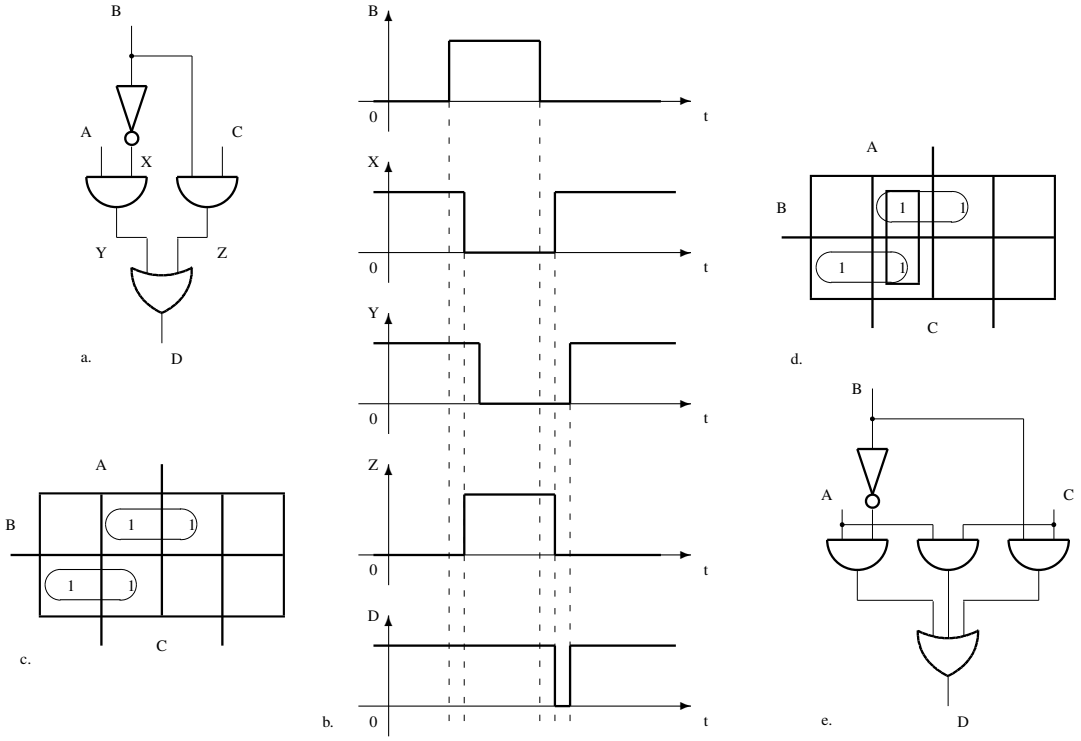


Figure 8.43: The typical example of propagation hazard. **a.** The circuit. **b.** The wave forms. **c.** The VK diagram of the function executed by the circuit. **d.** The added surface allowing the behavior of the circuit to have a continuous 1 surface. **e.** The equivalent circuit without hazard.

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) $\diamond$

Example 8.28 *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. $\diamond$

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 8.45 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).

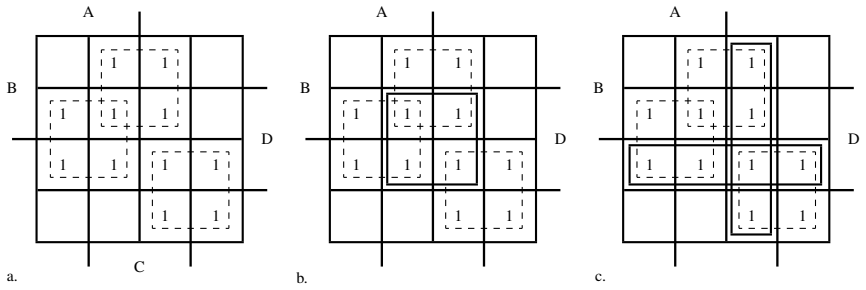


Figure 8.44: **a.** A hazardous combinational circuit. **b.** A partial solution to avoid the hazard. **c.** A full protection with two additional surfaces.

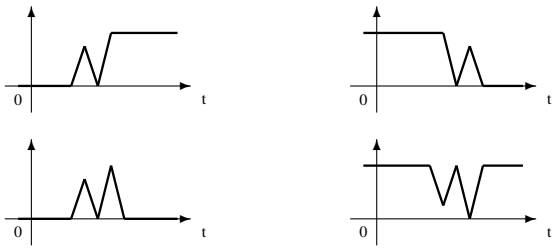


Figure 8.45: Examples of dynamic hazards.

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 form 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 took 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 codding styles are incompatible.

8.5 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[5]. 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, it 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.

[5]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.

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?

8.6 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.*

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* **??***).*

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:*

```
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:

```
/*********************************************************************
File name:      dec_spyke.v
Circuit name:   Simulation module to emphasize the spyke to the output of
                decoder driven by a counter
Description:    describes a system with a clock generator, a counter and
                a decoder, in two versions: with delays and without
                delays associated to the gates
*********************************************************************/
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
```

```
/*****************************************************************
File name:      dmux.v
Circuit name:   DMUX
Description:    structural description of a DMUX with and without delays
                associated to the gates
*****************************************************************/
 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;
// with no delay version
/*
    assign {out0, out1, out2, out3, out4, out5, out6, out7} = 1'b1 << in;
//*/
// with delays version
//*
    not     #1  not0(nin2, in[2]);
    not     #1  not1(nin1, in[1]);
    not     #1  not2(nin0, in[0]);
    not     #1  not3(in2, nin2);
    not     #1  not4(in1, nin1);
    not     #1  not5(in0, nin0);

    nand    #2  nand0(out0, nin2, nin1, nin0, enable);
    nand    #2  nand1(out1, nin2, nin1,  in0, enable);
    nand    #2  nand2(out2, nin2,  in1, nin0, enable);
    nand    #2  nand3(out3, nin2,  in1,  in0, enable);
    nand    #2  nand4(out4,  in2, nin1, nin0, enable);
    nand    #2  nand5(out5,  in2, nin1,  in0, enable);
    nand    #2  nand6(out6,  in2,  in1, nin0, enable);
    nand    #2  nand7(out7,  in2,  in1,  in0, enable);
//*/
endmodule
```

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* **??**.

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:*

```
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    = 3'110b, // 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* **??**.

Problem 8.18 *Redesign the FIFO memory represented in Figure* **??** *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 then 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:*

```
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:*

```
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* **??***. Test it using a special test module.*

Problem 8.25 *Redesign the MAC circuit represented in Figure* **??** *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 sting.*

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* **??***). 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* **??**.

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 8.46. Do the following:*

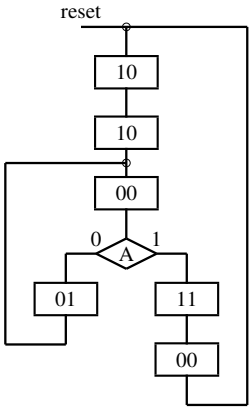


Figure 8.46:

1. *assign the sate 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.*

8.7 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:

```
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:*

```
   parameter    n = 16   ,    // word size
                m = 8         // number of levels
```

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*
  `{stack0, stack1, stack2, ...} <= {(stack0 + stack1), stack2, stack3,...}`
- *sub: subtract*
  `{stack0, stack1, stack2, ...} <= {(stack0 - stack1), stack2, stack3,...}`

- *inc: increment*
  `{stack0, stack1, stack2, ...} <= {(stack0 + 1), stack1, stack2, ...}`
- *dec: decrement*
  `{stack0, stack1, stack2, ...} <= {(stack0 - 1), stack1, stack2, ...},`
- *and: bitwise AND*
  `{stack0, stack1, stack2, ...} <= {(stack0 & stack1), stack2, stack3,...}`
- *or: bitwise OR*
  `{stack0, stack1, stack2, ...} <= {(stack0 | stack1), stack2, stack3,...}`
- *xor: bitwise XOR*
  `{stack0, stack1, stack2, ...} <= {(stack0 ⊕ stack1), stack2, stack3,...}`
- *not: bitwise NOT*
  `{stack0, stack1, stack2, ...} <= {(∼stack0), stack1, stack2, ...}`
- *over:*
  `{stack0, stack1, stack2, ...} <= {stack1, stack0, stack1, stack2, ...}`
- *dup: duplicate*
  `{stack0, stack1, stack2, ...} <= {stack0, stack0, stack1, stack2, ...}`
- *rightShift: right shift one position (integer division)*
  `{stack0, stack1, ...} <= {({1'b0, stack0[31:1]}), stack1, ...}`
- *arithShift: arithmetic right shift one position*
  `{stack0, stack1, ...} <= {({stack0[31], stack0[31:1]}), stack1, ...}`
- *get: push dataIn in top of stack*
  `{stack0, stack1, stack2, ...} <= {dataIn, stack0, stack1, ...},`
- *acc: accumulate dataIn*
  `{stack0, stack1, stack2, ...} <= {(stack0 + dataIn), stack1, stack2, ...},`
- *swp: swap the last two recordings in stack*
  `{stack0, stack1, stack2, ...} <= {stack1, stack0, stack2, ...}`
- *nop: no operation*
  `{stack0, stack1, stack2, ...} <= {stack0, stack1, stack2, ...}.`

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 (stack0 == stack1)*
- `ltFlag` *: is one if (stack0 ¡ stack1)*
- `zeroFlag` *: is one if (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[1]

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*.

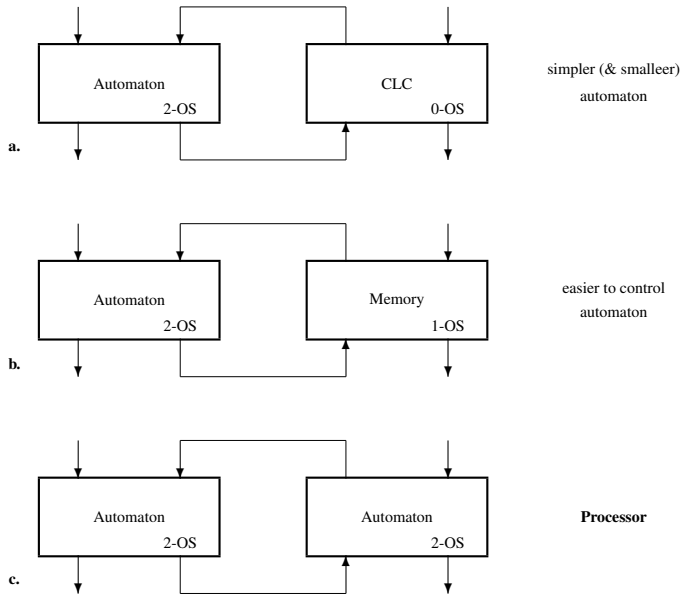


Figure 9.1: **The three types of 3-OS machines. a.** The third loop is closed through a combinational circuit resulting less complex, sometimes smaller, finite automaton. **b.** The third loop is closed through memories allowing a simplest control. **c.** The third loop is closed through another automaton 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).

[1]Quote from *Tao Te King* of Lao Tzu translated by Brian Browne Walker.

- The third type connects in a loop two automata (see Figure 9.1c). This last type is typical for 3-OS, having the *processor* as the main component.

All these types of loops will be exemplified emphasizing a new and very important process appearing at the level of the third order system: **the segregation of the simple from the complex in order to reduce the global (apparent) complexity**.

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.

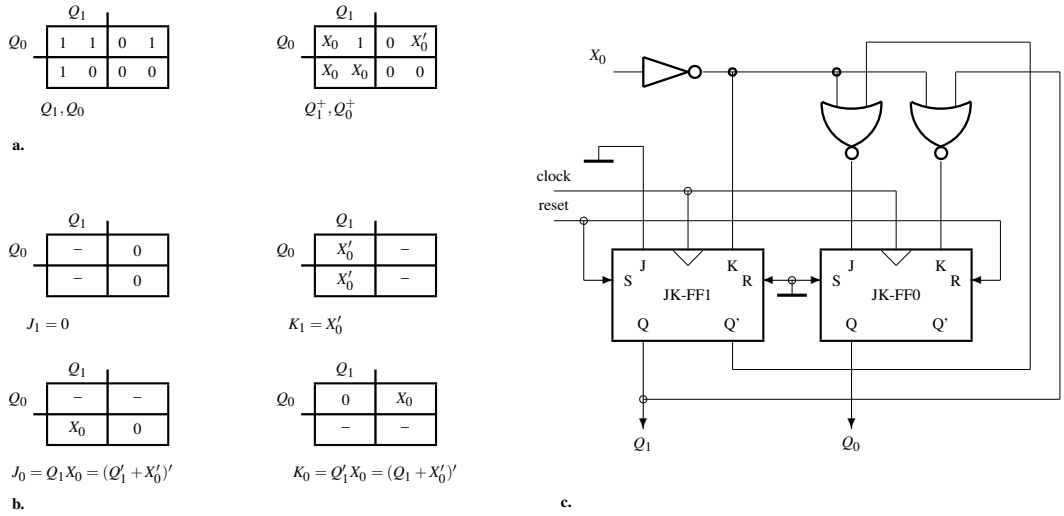


Figure 9.2: **Translating D transition diagrams in the corresponding JK transition diagrams.** The transition VK diagrams for the JK implementation of the finite half-automaton used to recognize binary string belonging to the 1^n0^m set of strings.

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 8.17 (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. $\diamond$

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:*

$$J_1 = Q_0 \cdot empty'$$
$$K_1 = Q_0' \cdot full'$$
$$J_0 = Q_1' \cdot empty'$$
$$K_0 = Q_1 \cdot done$$

If we compare with the previous D flip-flop solution where the loop circuit is defined by

$$Q_1^+ = Q_1 \cdot Q_0 + Q_0 \cdot empty' + Q_1 \cdot full$$
$$Q_0^+ = Q_1' \cdot Q_0 + Q_0 \cdot done' + Q_1' \cdot empty'$$

results a big reduction of complexity. $\diamond$

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*

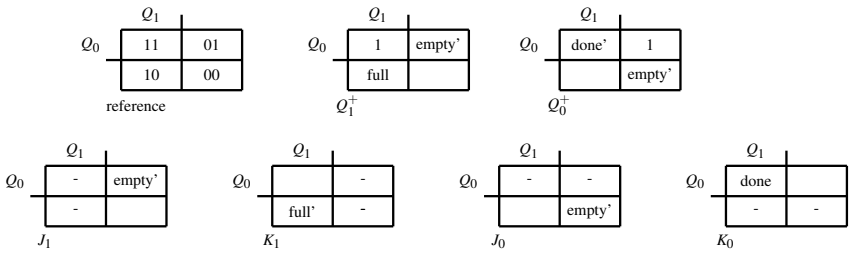


Figure 9.3: **Translating D transition diagrams in the corresponding JK transition diagrams.** The transition VK diagrams for the JK implementation of the finite automaton used to control MAC circuit (see Example 4.3).

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 AND_2 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" sunken 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 **s**etable **count**er, $SCOUNT_n$. 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).

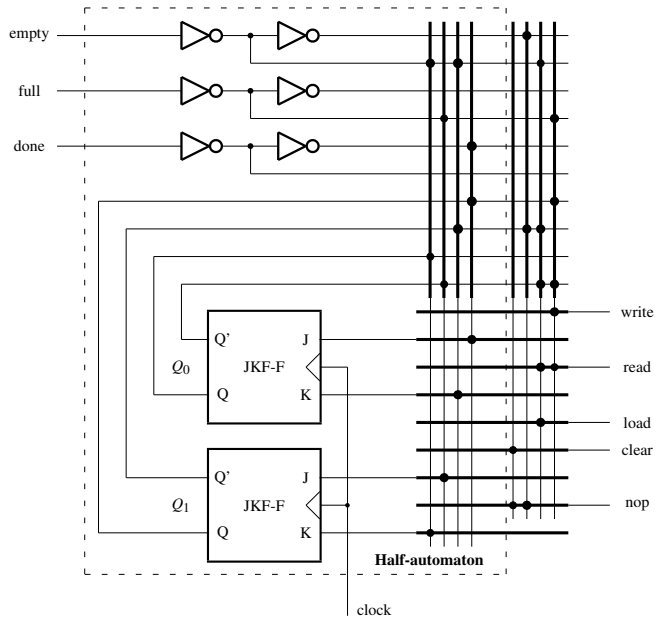


Figure 9.4: **Finite automaton with smart "JK register".** The new implementation of FA from Figure 8.28 using a "JK register" as a state register. The associated half-automaton is simpler (the corresponding PLA is smaller).

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 8.29 is implemented using an $SCOUNT_n$ with $n = 2$. Because the states are codded using increment encoding, the state transitions in the flow-chart can be interpreted as follows:*

- *in the state q_0 if $empty = 0$, then the state code is incremented, else it remains the same*
- *in the state q_1 if $empty = 0$, then the state code is incremented, else it remains the same*
- *in the state q_2 if $done = 1$, then the state code is incremented, else it remains the same*
- *in the state q_3 if $full = 0$, 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. $\diamond$

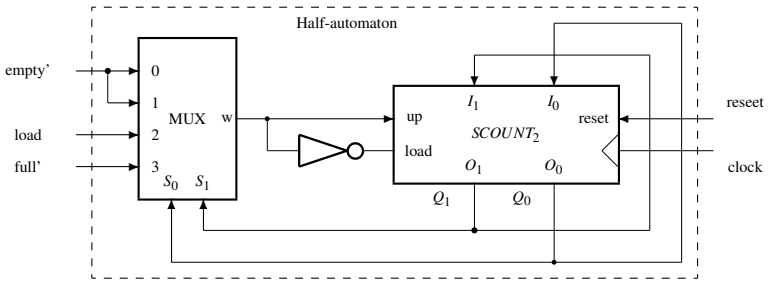


Figure 9.5: **Finite half-automaton implemented with a setable counter.** The last implementation of the half-automaton associated with FA from Figure 8.28 (with the function defined in Figure 8.29 where the states coded in parenthesis). A synchronous two-bit counter is used as state register. The simple four-input MUX commands the counter.

Example 9.4 *This example is also a remake. The half-automaton of the automaton which controls the operation* `macc` *in Example 4.6 will be implemented using a presetable counter as register. See Figure* **??** *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 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 I_3, I_2, I_1 and I_0 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 $SCOUNT_n$ substitutes the command **switch** with the same simple expressed, but more powerful, command **increment**.

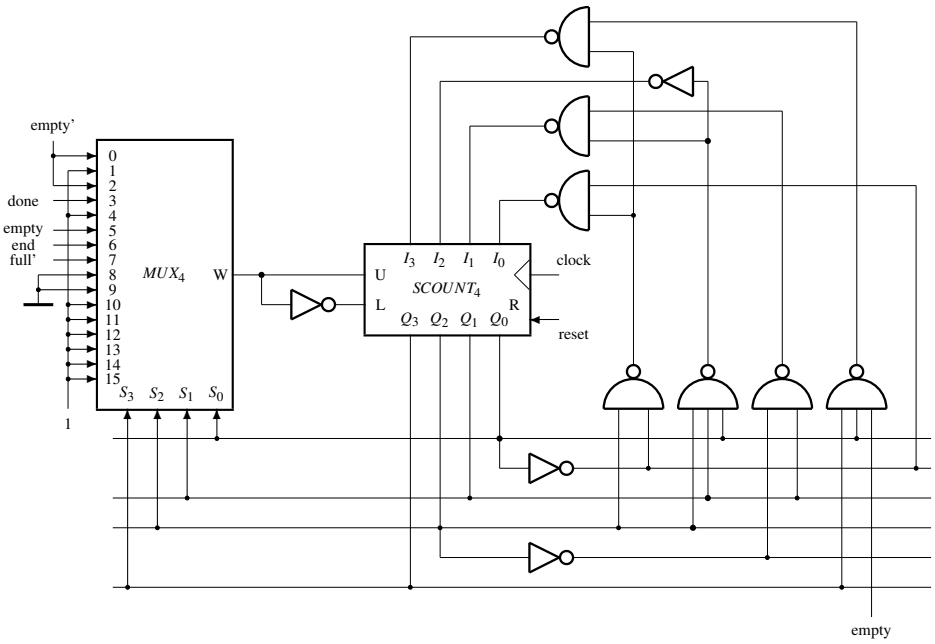


Figure 9.6: **Finite half-automaton for controlling the function** `macc`**.** The function was previously implemented using a CROM in Example 4.6.

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 belongs 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

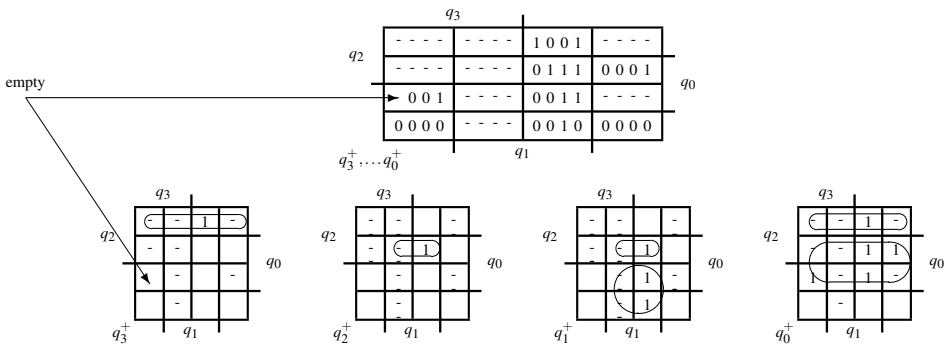


Figure 9.7: **Transition diagrams for the presetable counter used as state register.** The complex (random) part of the automaton is represented by the loop closed to the load input of the presetable counter.

– 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 CR_{out} signal, in each clock cycle when it is enabled (E = 1), in order to be applied on the CR_{in} 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.

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:

```
<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:

```
        inc cjmp crout bubu // ACC = in + 1
        right jmp cucu      // ACC = in
bubu    inc                 // ACC = in + 1
cucu    ...
```

The first increment command is followed by different operarion 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.

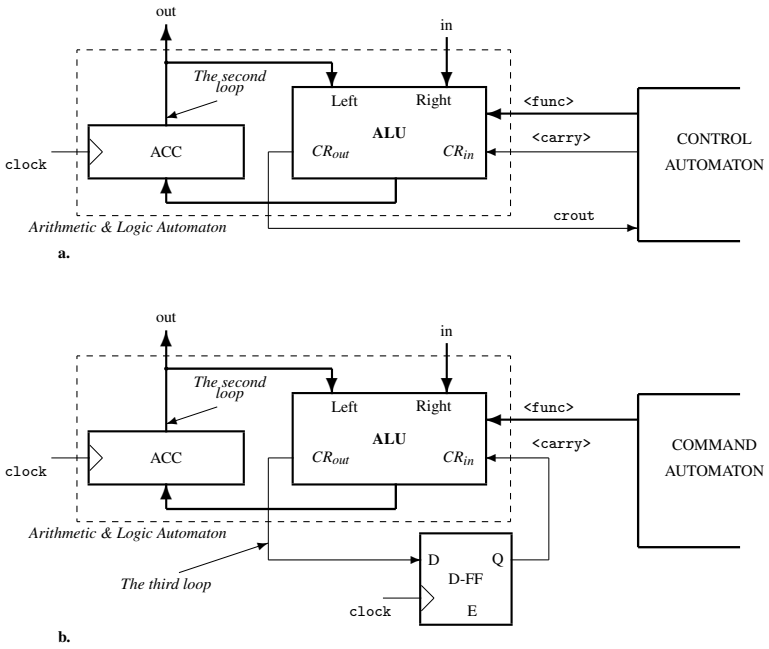


Figure 9.8: **The third loop closed over an arithmetic and logic automaton. a.** The basic structure: a simple automaton (its loop is closed through a simple combinational circuit: ALU) working under the supervision of a control automaton. **b.** The improved version, with an additional 1-bit state register 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.

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:

```
<command> ::= <func>;
<func> ::= right | and | or | xor | add |
           sub | inc | shl | addcr | subcr | inccr | shlcr;
<test> ::= - ;
```

The field `<test>` is not used, and the control automaton can be substituted by a command automaton. The field `<func>` is codded 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, inccr, shlcr`. The instructions `xxxcr` 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}
addcr: {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

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 8.14 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 *microprogram*. 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 arithmetic 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* `intepolEP` *is presented in Figure 9.9.*

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 multiplexor 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:* `inR` *and* `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:*

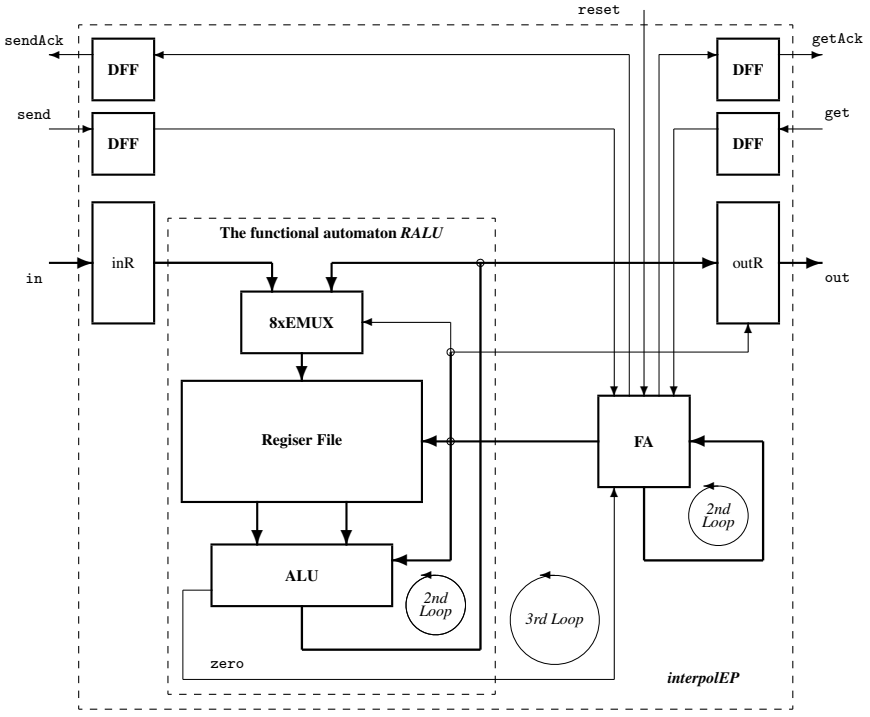


Figure 9.9: **The elementary processor** `interpolEP`**.**

```
STATE       FUNCTION                TEST        EXT. SIGNAL  NEXT STATE

waitSend    reg0 <= inReg,          if (send)   sendAck,     next = test;
                                        else                 next = waitSend;
test        reg1 <= reg1,           if (zero)                next = add;
                                        else                 next = waitGet;
waitGet     outReg <= reg1,         if (get)    getAck,      next = move1;
                                        else                 next = waitGet;
move1       reg2 <= reg1,                                    next = move0;
move0       reg1 <= reg0,                                    next = waitSend;
add         reg1 <= reg0 + reg2,                             next = divide;
divide      reg1 <= reg1 >> 1,                               next = waitGet;
```

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 `outR`*.* ◇

Example 9.6 ∗ *The EP structure is exemplified framed inside the simple system represented in Figure 9.10, where:*

inFIFO *: provides the input data for EP when* `read = 1` *if* `empty = 0`

outFIFO *: receives the output data generated by EP when* `write = 1` *if* `full = 0`

LIFO *: stores intermediary data for EP if* `push = 1` *and send back the last sent data if* `pop`

Elementary Processor *: is one of the simplest embodiment of an EP containing:*

Control Automaton *: a strict initial control automaton (see CROM from Figure* **??***)*

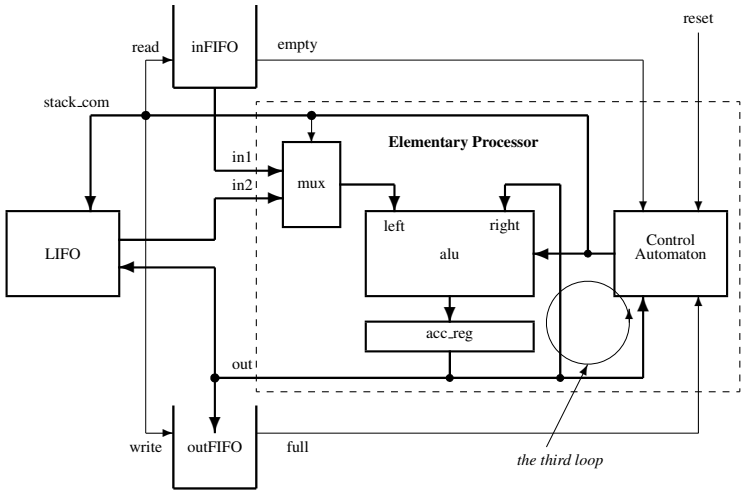


Figure 9.10: **An example of elementary processor (*EP*).** The third loop is closed between a simple execution automaton (`alu & acc_reg`) and a complex control automaton used to generate **the** sequence of operations to be performed by `alu` and to control the data flow between EP and the associated memory resources: `LIFO, inFIFO, outFIFO`.

alu *: an Arithmeetic & 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 inFIFO or from LIFO.*

The control automaton is a one function CROM *that commands the functional automaton, receiving from it only the carry output,* `cr`*, of the adder embedded in ALU.*

The description of PE must be supplemented with the associated microprogramming language*, as follows:*

```
<microinstruction> ::= <label> <command> <mod> <test> <next>;
<label> ::= <any string having maximum 6 symbols>;
<command> ::= <func> <inout>;
<mod> ::= jmp | cjmp | - ;
<test> ::= zero | notzero | cr | notcr | empty | nempty | full | nfull;
<next> ::= <label>;
<func> ::= left | add | half0 | half1 | - ;
<inout> ::= read | write | push | pop ;
```

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:

```
 inc for <mode>
 right: acc_reg <= acc_reg
```

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:

```
microprogram mean;
    bubu    read,   cjmp,   empty,  bubu,   left;
    cucu    cjmp,   empty,  cucu;
            read,   add,    cjmp,   cr,     one;
            half0;
    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* `cr = 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` *send 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.* $\diamond$

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 ::= <op_code> <value>

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*.

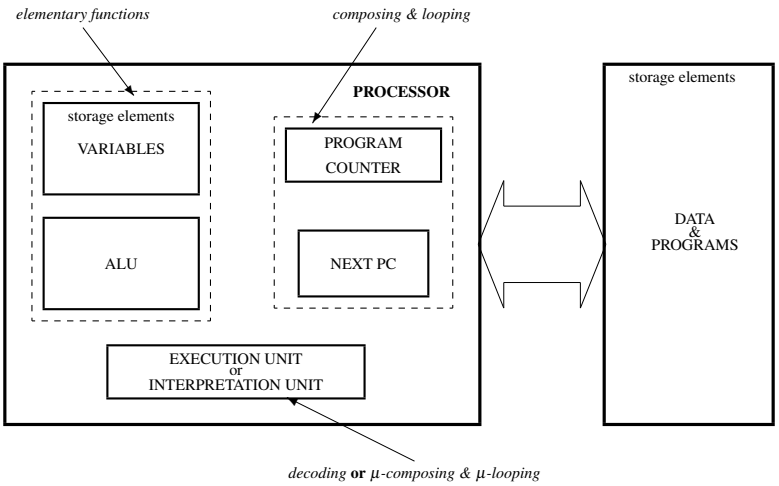


Figure 9.11: **The processor (*P*) in its environment.** P works loop connected with an external memory containing data and programs. Inside P *elementary function*, applied to a small set of very accessible variables, are composed in *linear or looped* sequences. The instructions read from the external 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:
  - 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 9.12a)
- **von Neumann architecture** with only one external memory used for storing both data and programs (see Figure 9.12b).

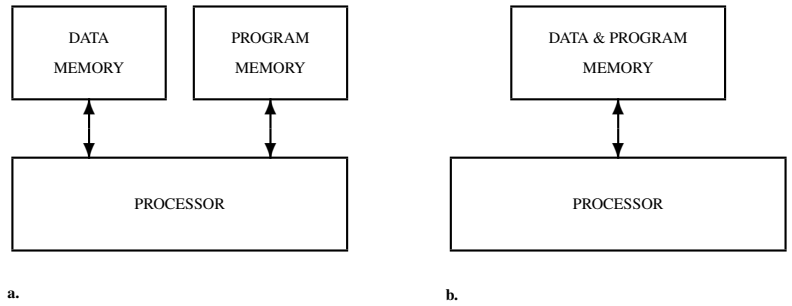


Figure 9.12: **The two main computer architectures. a.** Harvard Architecture: data and programs are stored in two different memories. **b.** Von Neumann Architecture: both data and programs are stored in the same memory.

The preferred embodiment for an executing processor is a Hardvare 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 9.12a) determine the internal structure of the processor to have mechanisms allowing in each clock cycle cu 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

[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.

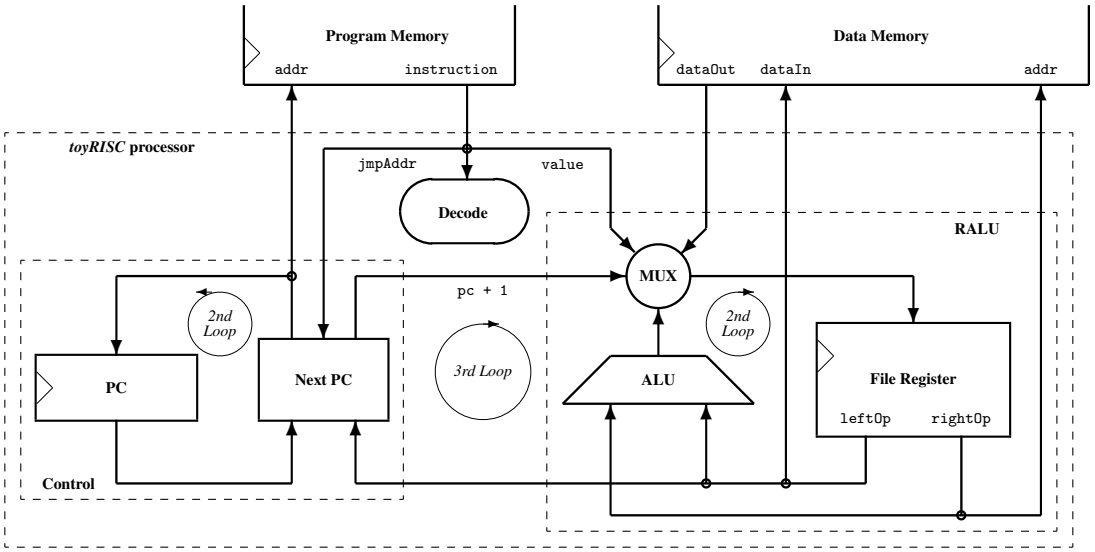


Figure 9.13: **The organization of *toyRISC* processor.**

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 3dr 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.

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.

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.

```
/*************************************************************************
 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,   // pc = (rf[left] = 0) ? pc + value : pc+1
 nzjmp   = 6'b00_0011,   // pc = !(rf[left] = 0) ? pc + value : pc+1
 ret     = 6'b00_0101,   // return from subroutine: pc = rf[left][15:0];
 ajmp    = 6'b00_0110,   // pc = value;
 call    = 6'b00_0111,   // subroutine call: pc = value; rf[dest] = pc+1;
 // ARITHMETIC & LOGIC, for all these instructions: pc = pc+1;
 inc     = 6'b11_0000,   // rf[dest] = rf[left] + 1;
 dec     = 6'b11_0001,   // rf[dest] = rf[left] - 1;
 add     = 6'b11_0010,   // rf[dest] = rf[left] + rf[right];
 sub     = 6'b11_0011,   // rf[dest] = rf[left] - rf[right];
 inccr   = 6'b11_0100,   // rf[dest] = (rf[left] + 1)[32];
 deccr   = 6'b11_0101,   // rf[dest] = (rf[left] - 1)[32];
 addcr   = 6'b11_0110,   // rf[dest] = (rf[left] + rf[right])[32];
 subcr   = 6'b11_0111,   // rf[dest] = (rf[left] - rf[right])[32];
 lsh     = 6'b11_1000,   // rf[dest] = rf[left] >> 1;
 ash     = 6'b11_1001,   // rf[dest] = {rf[left][31], rf[left][31:1]};
 move    = 6'b11_1010,   // rf[dest] = rf[left];
 swap    = 6'b11_1011,   // rf[dest] = {rf[left][15:0],rf[left][31:16]};
 neg     = 6'b11_1100,   // rf[dest] = ~rf[left];
 bwand   = 6'b11_1101,   // rf[dest] = rf[left] & rf[right];
 bwor    = 6'b11_1110,   // rf[dest] = rf[left] | rf[right];
 bwxor   = 6'b11_1111,   // rf[dest] = rf[left] ^ rf[right];
 // MEMORY, for all these instructions: pc = pc+1;
 read    = 6'b10_0000,   // read from dataMemory[rf[right]];
 load    = 6'b10_0111,   // rf[dest] = dataOut;
 store   = 6'b10_1000,   // dataMemory[rf[right]] = rf[left];
 val     = 6'b01_0111;   // rf[dest] = {{16*{value[15]}}, value};
```

Figure 9.14: **The architecture of *toyRISC* processor.**

```
/*************************************************************************
File name:      toyRISC.v
Circuit name:   Toy Risc
Description:    structural description of Toy Risc 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.

```
/*************************************************************************
File name:      Decode.v
Circuit name:   Instruction Decodeer
Description:    describe the decoding circuits for Toy RISC processor
*************************************************************************/

module  Decode( output          we          ,
                output          writeEnable ,
                input   [5:0]   opCode      );

    assign  we          = opCode == 6'b101000           ;
    assign  writeEnable = &opCode[5:4] | &opCode[2:0]   ;

endmodule
```

Figure 9.16: **The module** `Decode` **of the *toyRISC* processor.**

```
/*************************************************************************
File name:      Control.v
Circuit name:   Control Section of toyRISC Processor
Description:    strucutrural description of the control section in
                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.**

```
/*************************************************************************
File name:      RALU.v
Circuit name:   Register & Arithmetic-Logic Unit
Description:    structural description of the RALU used in toyRISC
*************************************************************************/

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.**

```
/*************************************************************************
File name:      arithmetic.v
Circuit name:   Arithmetic Section of ALU (first version)
Description:    behavioral description of the arithmetic section of ALU
*************************************************************************/
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'b000: {nextCarry, arithOut} = leftIn + 1'b1          ; // inc
      3'b001: {nextCarry, arithOut} = leftIn - 1'b1          ; // dec
      3'b010: {nextCarry, arithOut} = leftIn + rightIn       ; // add
      3'b011: {nextCarry, arithOut} = leftIn - rightIn       ; // sub
      3'b100: {nextCarry, arithOut} = leftIn + carry         ; // inccr
      3'b101: {nextCarry, arithOut} = leftIn - carry         ; // deccr
      3'b110: {nextCarry, arithOut} = leftIn + rightIn + carry; // addcr
      3'b111: {nextCarry, arithOut} = leftIn - rightIn - carry; // subcr
     endcase
endmodule
```

Figure 9.19: **The version 1 of the module `alu` of the *toyRISC* processor.**

```
/*************************************************************************
File name:      arithmetic.v
Circuit name:   Arithmetic Section of ALU (second version)
Description:    behavioral description of the arithmetic section of ALU
*************************************************************************/

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.**

```
/*************************************************************************
File name:      arithmetic.v
Circuit name:   Arithmetic Section of ALU (third version)
Description:    behavioral description of the arithmetic section of ALU
*************************************************************************/
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.**

The time performance

The longest combinational path in a system using our ***toyRISC***, which imposes the minimum clock period, is:

$$T_{clock} = t_{clock\_to\_instruction} + t_{leftAddr\_to\_leftOp} + t_{throughALU} + t_{throughMUX} + t_{fileRegSU}$$

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 $t_{clock\_to\_instruction}$ – 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 ($t_{leftAddr\_to\_leftOp}$ or $t_{rightAddr\_to\_rightOp}$), 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.

9.4 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.

9.5 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 *Our CISC Processor: how must be codded the instruction set to avoid FUNC MUX?*

9.6 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:*

```
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.*

Chapter 10

COMPUTING MACHINES: $\geq$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 abut

- 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[1]

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.

[1]Niklaus 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.

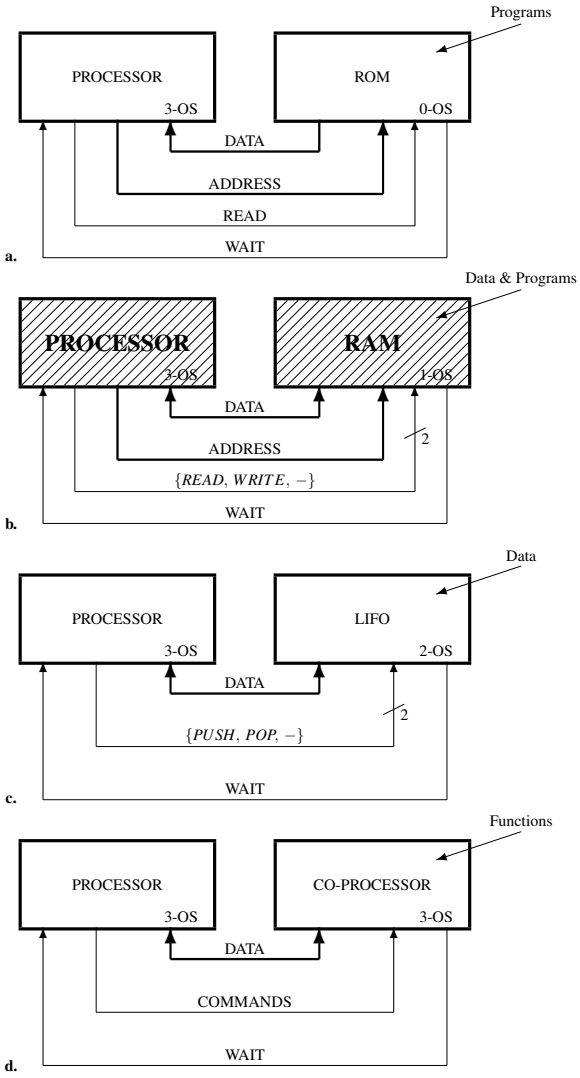


Figure 10.1: **The four types of 4-OS machines. a.** Fix program computers usual in embedded computation. **b.** General purpose computer. **c.** Specialized computer working working on a restricted data structure. **d.** Accelerated computation supported by a specialized co-processor.

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 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 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.

10.2.1 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 10.2):

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 10.3). 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 10.3).

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 interruupt. The module `dataSection` sends back to the module `controlSection` the absolute jump address.

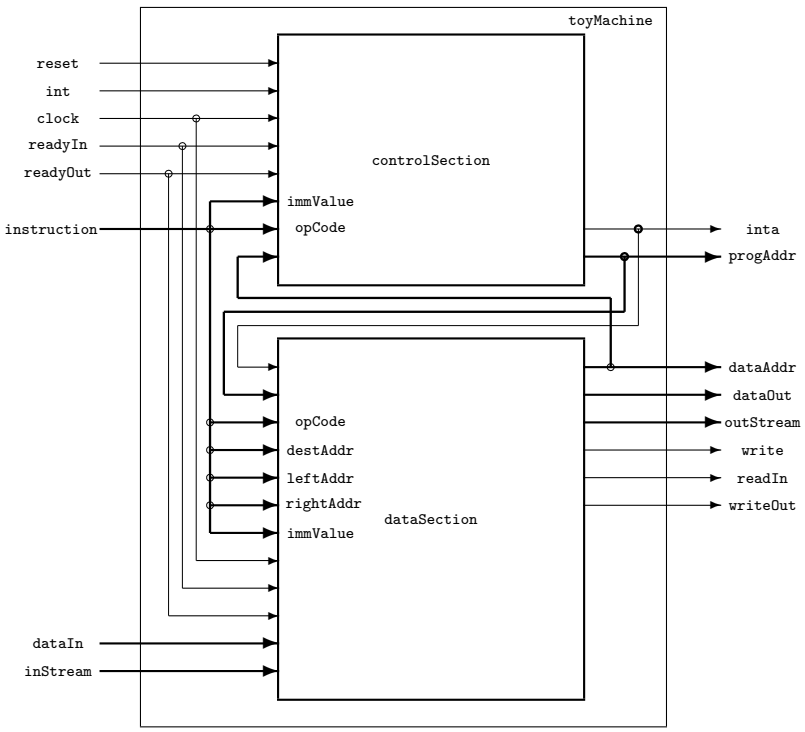


Figure 10.2: **The top level block schematic of the *toyMachine* design.**

See in Figure 10.4 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 `toyMachineStucture` 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

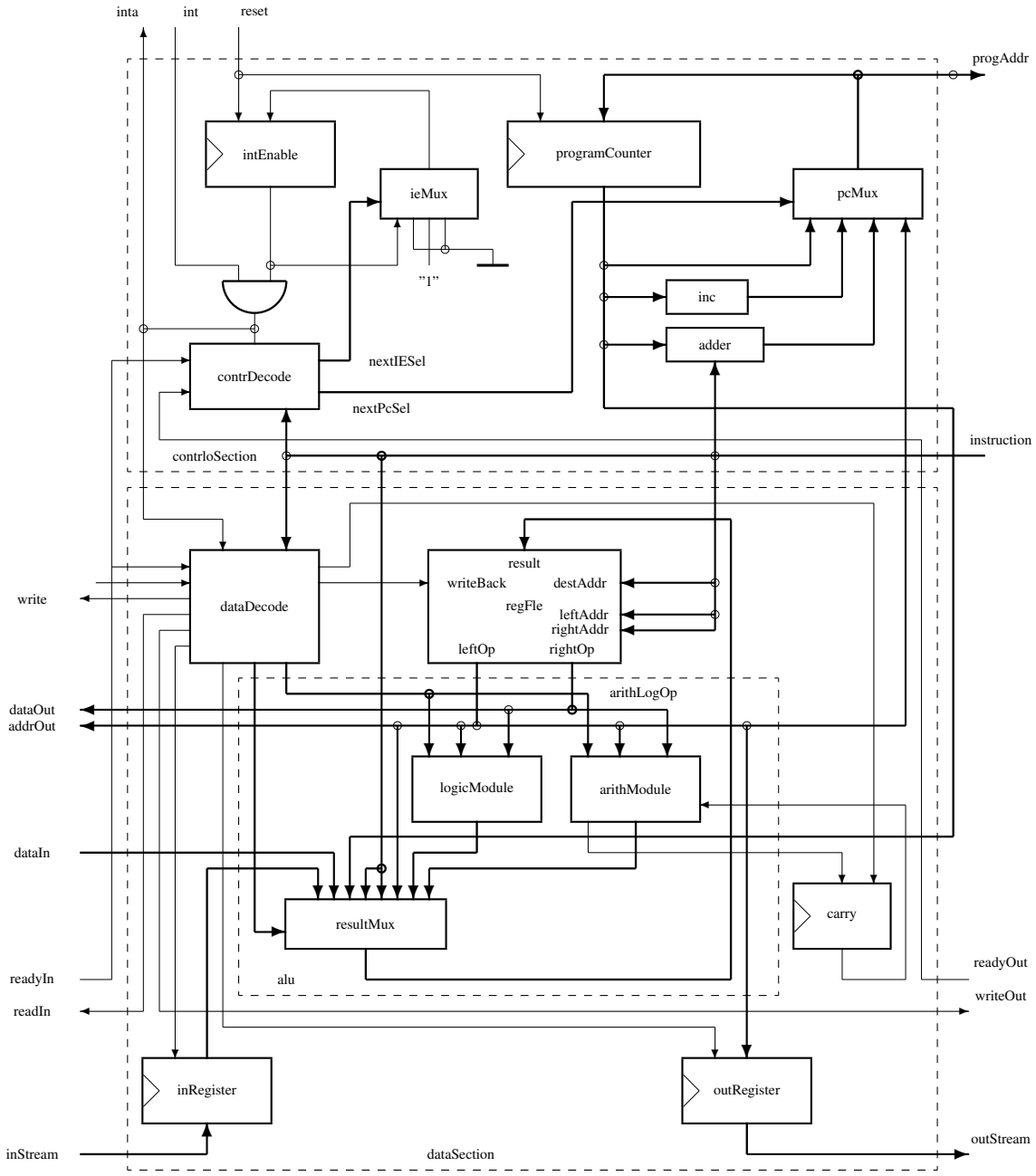

inta
int
reset
progAddr
intEnable
programCounter
ieMux
pcMux
"1"
inc
adder
contrDecode
nextIESel
nextPcSel
instruction
contrloSection
result
writeBack
destAddr
regFle
leftAddr
rightAddr
leftOp
rightOp
write
dataDecode
arithLogOp
dataOut
addrOut
logicModule
arithModule
dataIn
carry
resultMux
readyOut
readyIn
alu
writeOut
readIn
inRegister
outRegister
outStream
inStream
dataSection

Figure 10.3:

```
/*************************************************************************
File name:      toyMachineStructure.v
Circuit name:
Description:
*************************************************************************/
module toyMachineStructure
        (   input   [15:0]  inStream                            ,
            input   [31:0]  dataIn, instruction                 ,
            input           readyIn, readyOut, int, reset, clock,
            output          readIn , writeOut, inta, write      ,
            output  [15:0]  outStream                           ,
            output  [31:0]  dataAddr, dataOut, progAddr         );
    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, readyOut, inta, clock  ,
                            readIn, write                   ,
                            outStream                       ,
                            writeOut                        ,
                            dataAddr, dataOut               ,
                            dataIn, programCounter, immValue,
                            opCode                          ,
                            destAddr, leftAddr, rightAddr   );
    controlSection controlSection(int, readyIn, readyOut, reset, clock,
                                  inta                                 ,
                                  progAddr, programCounter             ,
                                  dataAddr, immValue                   ,
                                  opCode                               );
endmodule
```

Figure 10.4: **The top module *toyMachineStructure***. (Implemented on 321 LUTs, at 205 MHz)

```
module controlSection
        (input          int, readyIn, readyOut, reset, clock    ,
         output         inta                                    ,
         output  [31:0] progAddr, programCounter                ,
         input   [31:0] dataAddr, immValue                      ,
         input   [5:0]  opCode                                  );
    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 10.5: **The module** `controlSection`.

```
/*************************************************************************
File name:      contrDecode.v
Circuit name:
Description:
*************************************************************************/
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)                                  nextPcSel = 2'b11   ;
      else case(opCode)
             jmp     :                          nextPcSel = 2'b11   ;
             zjmp    : if (dataAddr == 0)       nextPcSel = 2'b10   ;
                        else                    nextPcSel = 2'b01   ;
             nzjmp   : if (dataAddr !== 0)      nextPcSel = 2'b10   ;
                        else                    nextPcSel = 2'b01   ;
             receive : if (readyIn)             nextPcSel = 2'b01   ;
                           else                 nextPcSel = 2'b00   ;
             issue   : if (readyOut)            nextPcSel = 2'b01   ;
                           else                 nextPcSel = 2'b00   ;
             halt :                             nextPcSel = 2'b00   ;
             default                            nextPcSel = 2'b01   ;
          endcase
endmodule
```

Figure 10.6: **The module** `contrDecode`.

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 10.3) 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 10.6). The type `reg` in this description must be understood as a variable. The actual structure of a register is not generated.

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 10.12).

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`
- `programCounter`: is saved as the “return” address to be used after running the program started by the acknowledged interrupt

```
/*************************************************************************
File name:      dataSection.v
Circuit name:
Description:
*************************************************************************/
module dataSection( input   [15:0]  inStream                          ,
                    input           readyIn, readyOut, inta, clock    ,
                    output          readIn, write                     ,
                    output  [15:0]  outStream                         ,
                    output          writeOut                          ,
                    output  [31:0]  dataAddr, dataOut                 ,
                    input   [31:0]  dataIn, programCounter, immValue,
                    input   [5:0]   opCode                            ,
                    input   [4:0]   destAddr, leftAddr, rightAddr     );
    reg     [15:0]  inRegister, outRegister ;
    reg             carry                   ;
    wire    [31:0]  result, leftOp, rightOp ;
    wire    [1:0]   arithLogOp  ;
    wire    [2:0]   resultSel   ;
    wire            writeBack, carryOut, inRegEnable, outRegEnable,
                    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'b11110 : destAddr ),
                .writeBack  (writeBack                  ),
                .leftAddr   (inta ? 5'b11111 : 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 10.7: **The module** `dataSection`**.**

```
/*************************************************************************
File name:      regFile.v
Circuit name:
Description:
*************************************************************************/
module regFile( input   [4:0]   destAddr    ,
                input           writeBack   ,
                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 10.8: **The module** `regFile`.

```
/*************************************************************************
File name:      alu.v
Circuit name:
Description:
*************************************************************************/
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                              ));
```

Figure 10.9: **The module** `alu`.

```
/*************************************************************************
File name:      arithModule.v
Circuit name:
Description:
*************************************************************************/
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
```

Figure 10.10: **The module** `arithModule`.

```
/*************************************************************************
File name:      logicModule.v
Circuit name:
Description:
*************************************************************************/
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 10.11: **The module** `logicModule`.

```
/*************************************************************************
File name:      dataDecode.v
Circuit name:
Description:
*************************************************************************/
module dataDecode(output reg [1:0] arithLogOp                          ,
                  output reg [2:0] resultSel                           ,
                  output reg       writeBack                           ,
                  output reg       inRegEnable , outRegEnable,
                                   carryEnable, readIn,writeOut, write,
                  input      [5:0] opCode                               ,
                  input            readyIn, readyOut, 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
```

Figure 10.12: **The module** `dataDecode`.

```
/*****************************************************************
File name:        dataDecode.v (continued)
*****************************************************************/
                        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  ;
                                end
                    endcase
            end
endmodule
```

Figure 10.13: **The module** `dataDecode` (continuation).

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 10.8.

The module called `alu` (see Figure 10.9) 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.

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.

10.2.2 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 10.15. 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

```
/*****************************************************************
Description:   various multiplexors
*****************************************************************/
module  mux4_32(output  reg [31:0]  out                 ,
                input       [31:0]  in0, in1, in2, 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 , in1 , in2 , 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, in1, in2, in3  ,
                                    in4, in5, in6, in7  ,
                input       [2:0]   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 10.14: **The** `multiplexor` **modules** `multiplexors`.

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 codding 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 10.15) 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:

- 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:

$$Q_1^+ = Q_1Q_0di' + Q_1'Q_0'ei$$

$$Q_0^+ = Q_1Q_0(di' \oplus int) + Q_1'Q_0ei + Q_1'Q_0'(ei \oplus int)$$

$$asyncInta = Q_1Q_0' + Q_1'Q_0ei$$

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 10.15):

dis : $Q_1Q_0 = 00$

en : $Q_1Q_0 = 10$

mem : $Q_1Q_0 = 01$

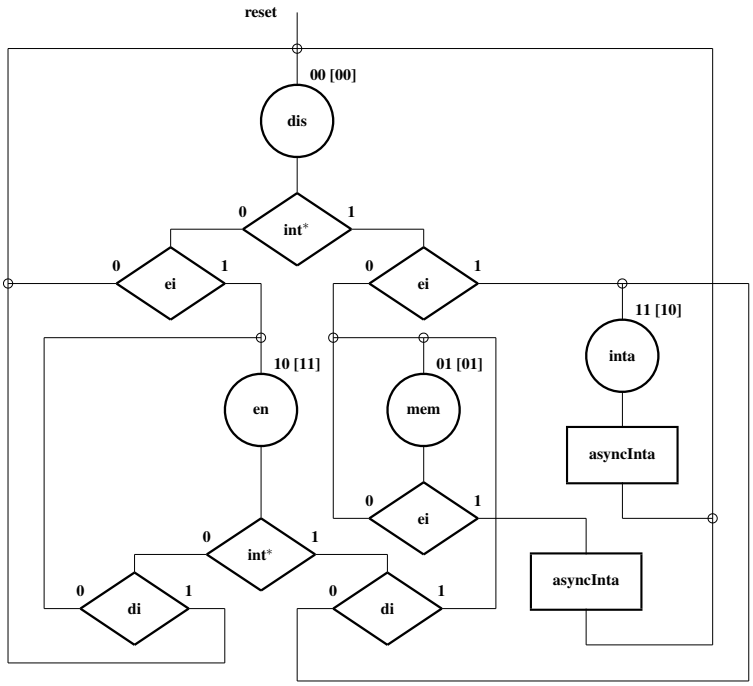


Figure 10.15: **Interrupt automaton for a limited width and an asynchronous `int` signal.**

inta : $Q_1Q_0 = 11$

The new state transition functions are:

$$Q_1^+ = Q_1Q_0'di' + Q_1'Q_0'ei$$

$$Q_0^+ = Q_0'int + Q_1'Q_0ei'$$

The Verilog behavioral description for this version is presented in Figure 10.16.

If we make another step re-designing the loop for an "intelligent" JK register, then results for the loop the following expressions:

$$J_1 = Q_0'ei$$

$$K_1 = di + Q_0$$

$$J_0 = int$$

$$K_0 = Q_1 + ei$$

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 10.17 and in Figure 10.18

```
/*************************************************************************
File name:      .v
Circuit name:
Description:
*************************************************************************/

 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;
                 else if (ei)     {nextState, asyncInt} = 3'b10_0;
                          else    {nextState, asyncInt} = 3'b00_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;
                 else if (di)     {nextState, asyncInt} = 3'b00_0;
                          else    {nextState, asyncInt} = 3'b10_0;
          2'b11:                  {nextState, asyncInt} = 3'b00_1;
     endcase
 endmodule
```

Figure 10.16: **The module** `interruptAutomaton`.

```
/*************************************************************************
File name:      .v
Circuit name:
Description:
*************************************************************************/
/*************************************************************************
File name:      interruptAutomaton.v
Circuit name:
Description:
*************************************************************************/
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 10.17: **The structural description of the module** `interruptAutomaton` **implemented using JK-FFs.**.

```
/*************************************************************************
File name:      JKflipFlop.v
Circuit name:
Description:
*************************************************************************/
 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 10.18: **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.

10.3 Problems

Problem 10.1 *Interpretative processor with distinct program counter block.*

10.4 Projects

Project 10.1

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*

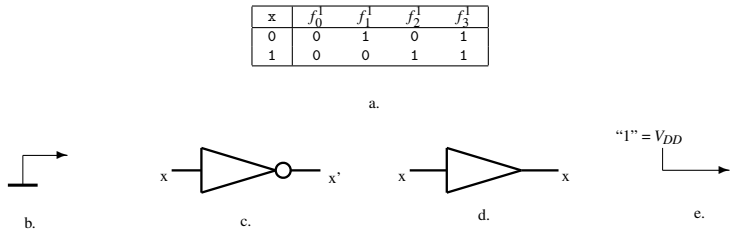


Figure A.1: **One-bit logic functions. a.** The truth table for 1-variable logic functions. **b.** The circuit for "0" (false) by connecting to the ground potential. **c.** The logic symbol for the inverter circuit. **d.** The logic symbol for driver function. **e.** The circuit for "1" (true) by connecting to the high potential.

Numerical interpretation of the NOT circuit: **one-bit incrementer**. 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_8^2 : 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_6^2 : XOR (exclusiv OR) is:

a.

x y	f_0^2	f_1^2	f_2^2	f_3^2	f_4^2	f_5^2	f_6^2	f_7^2	f_8^2	f_9^2	f_A^2	f_B^2	f_C^2	f_D^2	f_E^2	f_F^2
0 0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0 1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1 1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

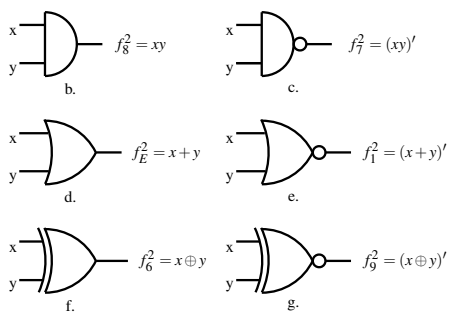


Figure A.2: **Two-bit logic functions. a.** The table of all two-bit logic functions. **b.** AND gate – the original gate. **c.** NAND gate – the most used gate. **d.** OR gate. **e.** NOR gate. **f.** XOR gate – modulo2 adder. **g.** NXOR gate – coincidence circuit.

  - the 2-modulo adder
  - NEQ (not-equal) circuit, a comparator pointing out when the two 1-bit numbers on the input are inequal
  - an enabled inverter:
    if $x = 1$ output is y'; **else** output is y;
  - a modulo 2 incrementer.

- f_B^2 : the logic implication is also used to compare 1-bit numbers because the output is 1 for $y < x$
- f_1^2 : 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* $N = 2^{2^n}$. $\diamond$

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.
$a+(b+c) = (a+b)+c = a+b+c$
$a(bc) = (ab)c = abc$
$a \oplus (b \oplus c) = (a \oplus b) \oplus c = a \oplus b \oplus c$.

Commutativity Commutativity allows us to connect to the inputs of **some** gates the variable in any order.
$a+b = b+a$
$ab = ba$
$a \oplus b = b \oplus a$

Distributivity Distributivity offers the possibility to define **all** logical functions as *sum of products* or as *product of sums*.
$a(b+c) = ab+ac$
$a+bc = (a+b)(a+c)$
$a(b \oplus c) = ab \oplus ac$.
Not all distributions are possible. For example:

$$a \oplus bc \neq (a \oplus b)(b \oplus c).$$

The table in Figure A.3 can be used to prove the previous inequality.

a	b	c	bc	a ⊕ bc	a⊕b	a⊕c	(a⊕b)(a⊕c)
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	0	0
1	1	0	0	1	0	1	0
1	1	1	1	0	0	0	0

Figure A.3: **Proving by tables.** Proof of inequality $a \oplus bc \neq (a \oplus b)(b \oplus c)$.

Absorbtion Absorbtion 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-absorbtion The half-absorbtion allows only a smaller, but non-neglecting, simplification.
$a+a'b=a+b$
$a(a'+b)=ab$.

Substitution The substitution principles say us what happen when a variable is substituted with a value.
$a+0=a$
$a+1=1$
$a0=0$
$a1=a$
$a\oplus 0=a$
$a\oplus 1=a'$.

Exclusion The most powerful simplification occurs when the exclusion principle is applicable.
$ab+a'b=b$
$(a+b)(a'+b)=b$.

Proof. For the first form:

$$ab+a'b=b$$

applying successively distribution, absorbtion 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.
$a+b=(a'b')'\quad ab=(a'+b')'$
$a'+b'=(ab)'\quad a'b'=(a+b)'$

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* $\{p3,p2,p1,p0\}$. *The logic equations result direct as 4 DNFs, one for each output bit:*
$p3 = abcd$
p2 = ab'cd' + ab'cd + abcd'
p1 = a'bcd' + a'bcd + ab'c'd + ab'cd + abc'd + abcd'
$p0 = a'bc'd + a'bcd + abc'd + abcd$.
Indeed, the p3 bit takes the value 1 only if $a = 1$ **and** $b = 1$ **and** $c = 1$ **and** $d = 1$. *The bit p2 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: $p3 = ((abcd)')'$
$p2 = ((ab'cd')'(ab'cd)'(abcd')')'$
$p1 = ((a'bcd')'(a'bcd'(ab'c'd)'(ab'cd)'(abc'd)'(abcd')')'$
$p0 = ((a'bc'd)'(a'bcd)'(abc'd)'(abcd)')'$.

These forms are more efficient in implementation because involve the same type of circuits (NANDs), and because the inverting circuits are usually faster.

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: $S_{mult2} = 54$. ◇

ab	cd	p3	p2	p1	p0
00	00	0	0	0	0
00	01	0	0	0	0
00	10	0	0	0	0
00	11	0	0	0	0
01	00	0	0	0	0
01	01	0	0	0	1
01	10	0	0	1	0
01	11	0	0	1	1
10	00	0	0	0	0
10	01	0	0	1	0
10	10	0	1	0	0
10	11	0	1	1	0
11	00	0	0	0	0
11	01	0	0	1	1
11	10	0	1	1	0
11	11	1	0	0	1

Figure A.4: **Combinatinal circuit represented a a truth table.** The truth table of the combinational circuit performing 2-bit multiplication.

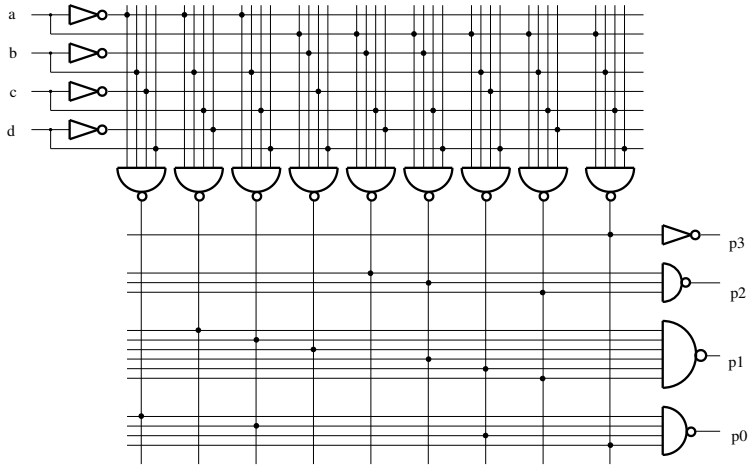


Figure A.5: **Direct implementation of a combinational circuit.** The direct implementation starting from DNF of the 2-bit multiplier.

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:*

$$p0 = a'bc'd + a'bcd + abc'd + abcd.$$

We will apply the following steps:

$$p0 = (a'bd)c' + (a'bd)c + (abd)c' + (abd)c$$

to emphasize the possibility of applying twice the exclusion principle, resulting

$$p0 = a'bd + abd.$$

Applying again the same principle results:

$$p0 = bd(a' + a) = bd1 = bd.$$

The exclusion principle allowed us to reduce the size of the circuit from 22 to 2.

We continue with the next output:

$$p1 = a'bcd' + a'bcd + ab'c'd + ab'cd + abc'd + abcd' =$$

$= a'bc(d' + d) + ab'd(c' + c) + abc'd + abcd' =$
$= a'bc + ab'd + abc'd + abcd' =$
$= bc(a' + ad') + ad(b' + bc') =$
$= bc(a' + d') + ad(b' + c') =$
$= a'bc + bcd' + ab'd + ac'd.$
Now we used also the half-absorbtion principle reducing the size from 28 to 16.

Follows the minimization of p2:

$$p2 = ab'cd' + ab'cd + abcd' =$$

$= ab'c + abcd' =$
$= ab'c + acd'$
The p3 output can not be minimized. De Morgan law is used to transform the expressions to be implemented with NANDs.

$$p3 = ((abcd)')'$$

$p2 = ((ab'c)'(acd')')'$
$p1 = ((a'bc)'(bcd')'(ab'd)'(ac'd)')'$
$p1 = ((abcd)')'.$
Results the circuit from Figure A.6. $\diamond$

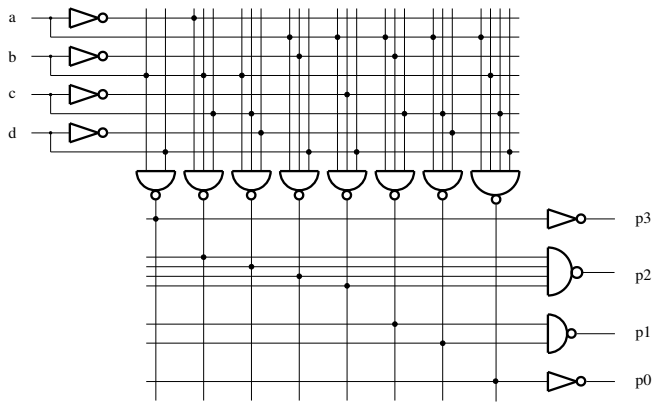


Figure A.6: **Minimal depth minimiztion** The first, minimal depth minimization of the 2-bit multiplier.

Multi-level minimization

Example A.3 *The same circuit for multiplying 2-bit numbers is used to exemplify the multilevel minimization. Results:*

$$p3 = abcd$$

$p2 = ab'c + acd' = ac(b' + d') = ac(bd)'$
$p1 = a'bc + bcd' + ab'd + ac'd = bc(a' + d') + ad(b' + c') = bc(ad)' + ad(bc)' = (bc) \oplus (ad)$
$p0 = bd.$
Using for XOR the following form:

$$x \oplus y = ((x \oplus y)')' = (xy + x'y')' = (xy)'(x'y')' = (xy)'(x + y)$$

results the circuit from Figure A.7 with size 22. $\diamond$

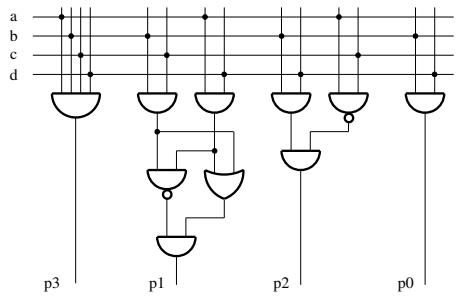


Figure A.7: **Multi-level minimization.** The second, multi-level minimization of the 2-bit multiplier.

Many output circuit minimization

Example A.4 *Inspecting carefully the schematics from Figure A.7 results: (1) the output p3 can be obtained inverting the NAND's output from the circuit of p2, (2) the output p0 is computed by a part of the circuit used for p2. Thus, we are encouraged to rewrite same of the functions in order to maximize the common circuits used in implementation. Results:*

$$x \oplus y = (xy)'(x + y) = ((xy) + (x + y)')'.$$

$$p2 = ac(bd)' = ((ac)' + bd)'$$

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. $\diamond$

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.

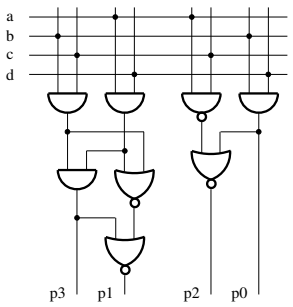


Figure A.8: **Multiple-output minimization.** The third, multiple-output minimization of the 2-bit multiplier.

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.* $\diamond$

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.* $\diamond$

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 $2(n+1)$ 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.

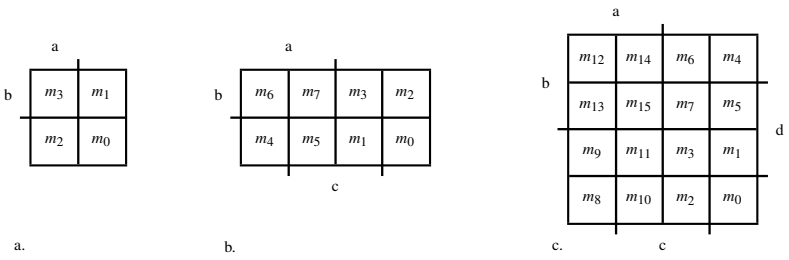


Figure A.9: **Veitch diagrams.** The Veitch diagrams for 2, 3, and 4 variables.

Example A.6 *Let be the function p1 and p2, two outputs of the 2-bit multiplier. Rewriting them using minterms results::*

$$p1 = m_6 + m_7 + m_9 + m_{11} + m_{13} + m_{14}$$

$$p2 = m_{10} + m_{11} + m_{14}.$$

In Figure A.10 p1 and p2 are represented.

◇

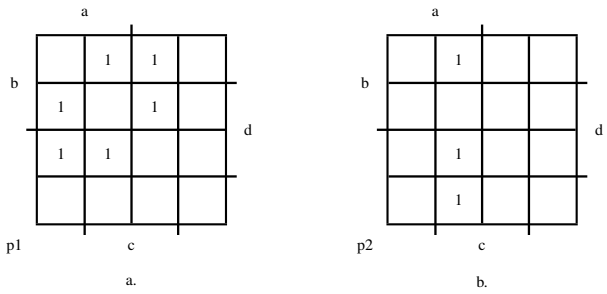


Figure A.10: **Using Veitch diagrams.** The Veitch diagrams for the functions p1 and p2.

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.

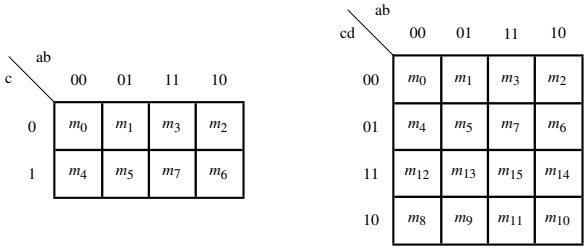


Figure A.11: **Karnaugh diagrams.** The Karnaugh diagrams for 3 and 4 variables.

In Figure A.12 the same functions, p1 and p2, 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 the 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

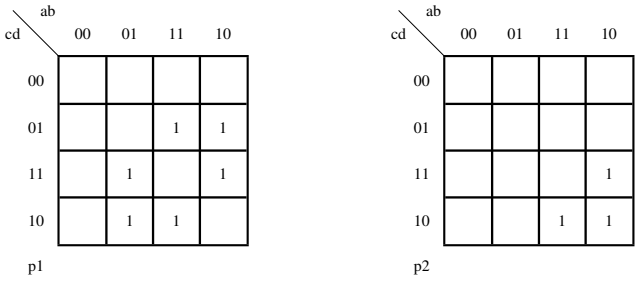


Figure A.12: **Using Karnaugh diagrams.** The Karnaugh diagrams for the functions p1 and p2.

  - of *rectangular* surfaces containing only 1's
  - 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.

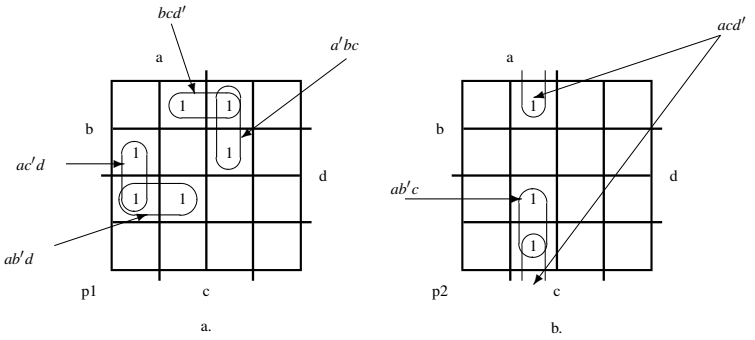


Figure A.13: **Minimizing with V-K diagrams.** Minimizing the functions $p1$ and $p2$.

Example A.7 *Let's take the V-K diagrams from Figure A.10. In the V-K diagram for p1 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:* $(b = 1)AND(c = 1)AND(d = 0)$.

Because the horizontal edges are considered adjacent, in the V-K diagram for p2 m_{14} *and* m_{10} *are adjacent forming a surface having* acd' *as associated term.*

The previously known form of p1 and p2 result if the terms resulting from the two diagrams are logically added. $\diamond$

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).

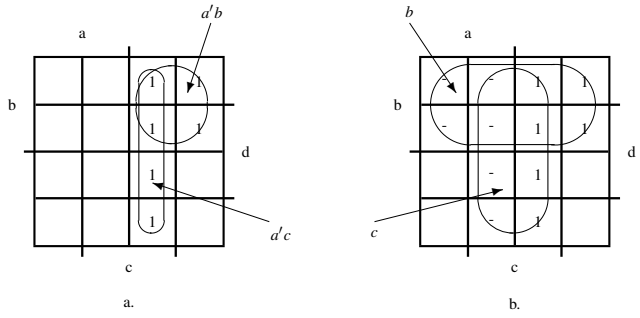


Figure A.14: **Minimizing incomplete defined functions. a.** The minimization of y (Example 1.8) ignoring the *"don't care"* terms. **b.** The minimization of y (Example 1.8) considering the *"don't care"* terms.

Example A.8 *Let be the 4-input circuit receiving the binary codded 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: $y = b + c$. $\diamond$

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.

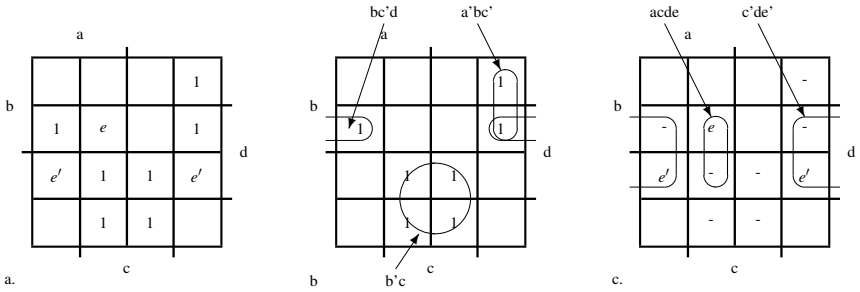


Figure A.15: **An example of V-K diagram with included functions. a.** The initial form. **b.** The form considered in the first step. **c.** The form considered in the second step.

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:*

$$c'de' + acde$$

In the third step the two forms are "added" resulting:

$$f(a,b,c,d,e) = bc'd + a'bc' + b'c + c'de' + acde.$$

◇

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.

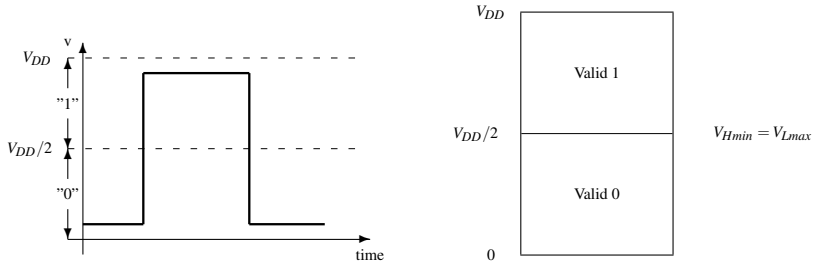


Figure B.1: **Defining 0-logic and 1-logic.** The circuit is supposed to interpret any value under $V_{DD}/2$ as 0, and any value bigger than $V_{DD}/2$ are interpreted as 1.

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 $V_H > V_{Hmin}$.

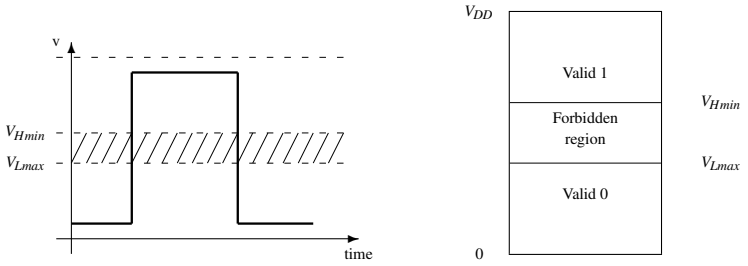


Figure B.2: **Defining the "forbidden region" for logic values.** A robust design asks a net distinction between the electrical values interpreted as 0 and the electrical values interpreted as 1.

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.

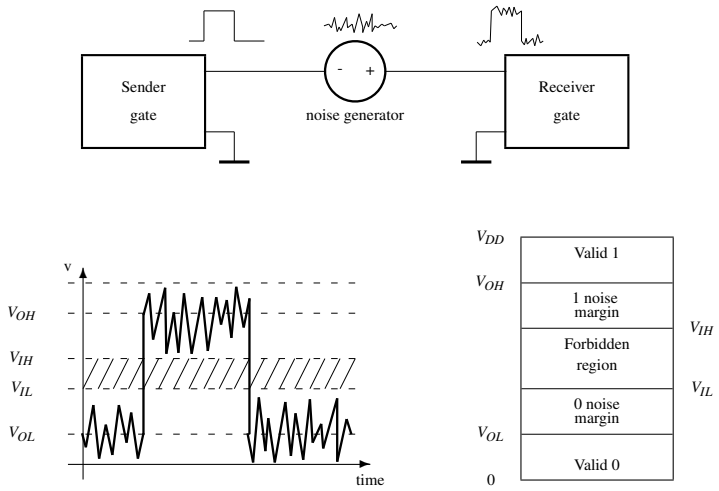


Figure B.3: **The noise margin.** The output signal must be generated with more restrictions to allow the receivers to "understand" correct input signals loaded with noise.

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*).

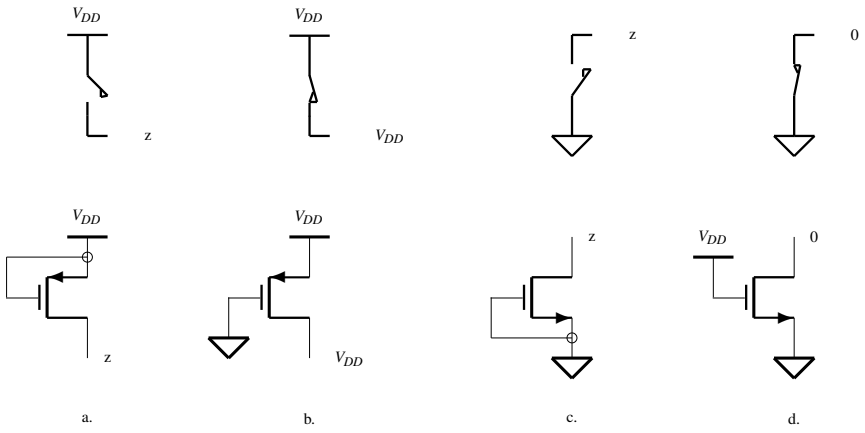


Figure B.4: **Basic switches.** **a**. Open switch connected to V_{DD}. **b**. Closed switch connected to V_{DD}. **c**. Open switch connected to *ground*. **d**. Closed switch connected to *ground*.

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:

$$i_G = C_{GS} \frac{dv_{GS}}{dt}$$

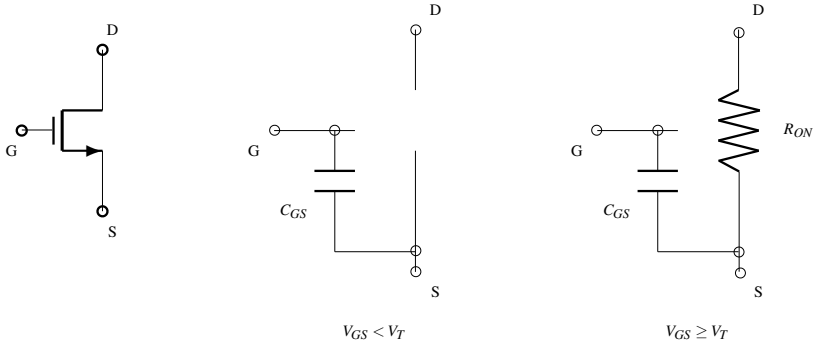


Figure B.5: **The MOSFET switch.** The *switch-resistor-capacitor* model consists in the two states: OF ($V_{GS} < V_T$), and ON ($V_{GS} \geq V_T$). In both states the input is defined by the capacitor C_{GS}.

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 invertor – 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

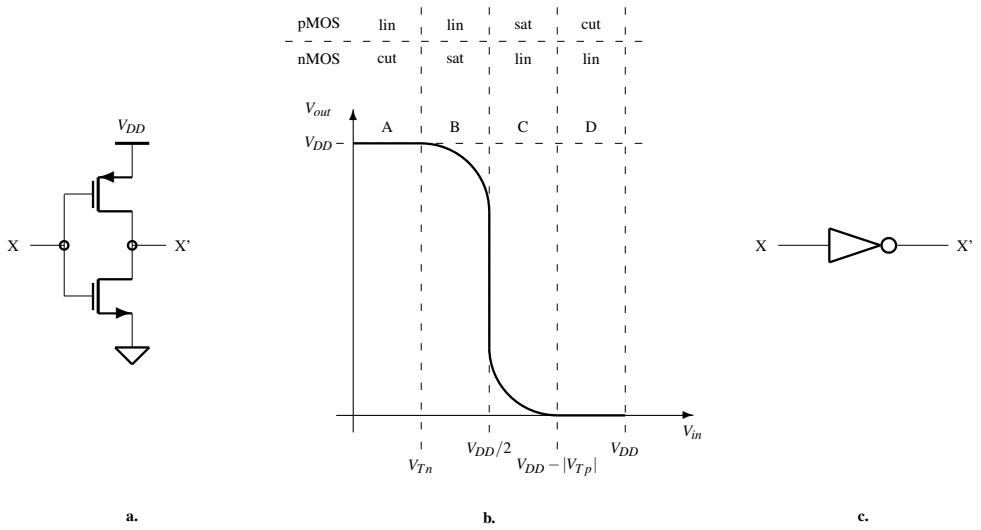


Figure B.6: **Building an invertor. a**. The invertor circuit. **b**. The logic symbol for the invertor circuit.

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 invertor 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.3.2 Dynamic behavior

The propagation time of an inverter can be analyzed using the two serially connected invertors represented in Figure B.7. The delay of the first invertor 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 invertor
- 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:

$$C_{wire} = C_{thickox} W_w L_w$$

- next stage input capacitance, C_G, approximated by summing the gate capacitance for pMOC and nMOS transistors:

$$C_G = C_{Gp} + C_{Gn} = C_{ox}(W_p L_p + W_n L_n)$$

The total load capacitance

$$C_L = C_{DB} + C_{wire} + C_G$$

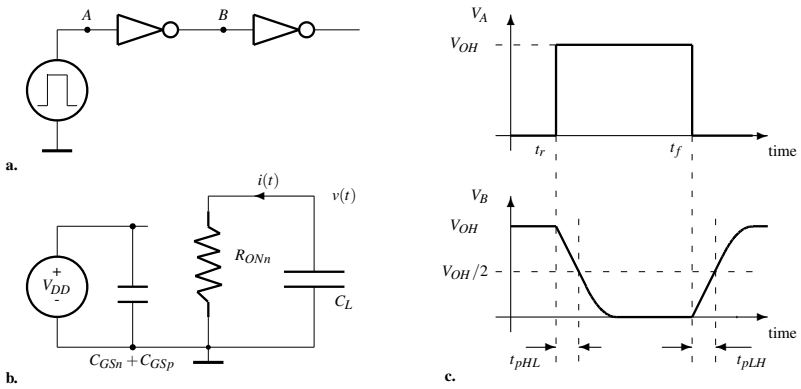


Figure B.7: **The propagation time.**

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}.

$$\frac{dv_{out}}{dt} = \frac{d}{dt}(\frac{q_L}{C_L}) = \frac{-I_{Dn(sat)}}{C_L}$$

$$\frac{dv_{out}}{dt} = \frac{\frac{V_{OH}}{2} - V_{OH}}{t_{pHL}}$$

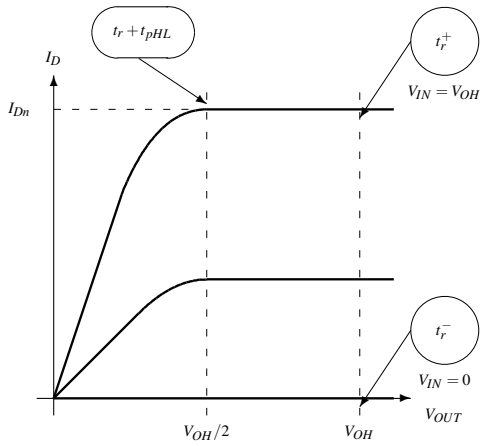


Figure B.8: **The output characteristic of the nMOS transistor.**

We solve the equations for t_{pHL}:

$$t_{pHL} = C_L \frac{1}{\mu_n C_{ox} \frac{W_n}{L_n}(V_{OH} - V_{Tn})} \frac{V_{OH}}{V_{OH} - V_{Tn}}$$

Because:

$$R_{ONn} = \frac{1}{\mu_n C_{ox} \frac{W_n}{L_n}(V_{OH} - V_{Tn})}$$

results:

$$t_{pHL} = C_L R_{ONn} \frac{1}{1 - \frac{V_{Tn}}{V_{OH}}} = k_n R_{ONn} C_L = k_n \tau_{nL}$$

where:

- τ_{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 invertor 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:

$$t_{p(no-buffer)} = t_{p0}\frac{W_{load}}{W_{driver}}$$

the buffered solution provide the propagation time:

$$t_{p(buffered)} = t_{p0}\left(\frac{W_1}{W_{driver}} + \frac{W_2}{W_1} + \frac{W_{load}}{W_2}\right)$$

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:

$$W_2 = \sqrt{W_1 W_{load}}$$

$$\frac{W_2}{W_1} = \frac{W_{load}}{W_2} = \sqrt{\frac{W_{load}}{W_1}}$$

We conclude: in order to add a minimal delay, the size ratio of successive drivers in a chain must be the same.

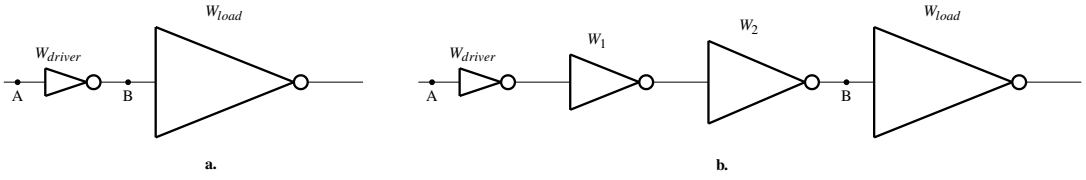


Figure B.9: **Buffered connection. a.** An invertor with small W is not able to handle at high frequency a circuit with big W. **b.** The buffered connection with two intermediary buffers.

In order to design the size of the circuits in Figure B.9b, let us consider $\frac{W_{load}}{W_{driver}} = n$. Then,

$$\frac{W_1}{W_{driver}} = \frac{W_2}{W_1} = \frac{W_{load}}{W_2} = \sqrt[3]{n}$$

The acceleration is

$$\alpha = \frac{t_{p(no-buffer)}}{t_{p(buffered)}} = \frac{\sqrt[3]{n^2}}{3}$$

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 $clock = 0$ the load capacitor is loaded from the power supply with the charge:

$$Q_L = C_L V_{DD}$$

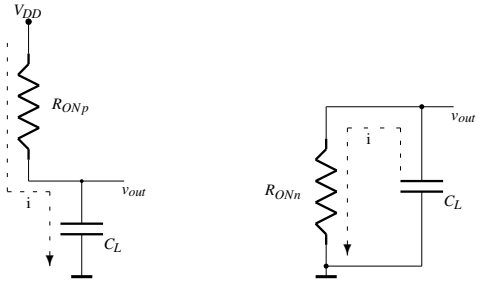


Figure B.10: **The main power consuming process.** For $V_{in} = 0$ C_L is loaded by the current provided by R_{ONp}. The charge from C_L is transferred to the ground through R_{ONn} for $V_{in} = V_{OH}$.

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 $clock = 1$, the same charge is transferred from the capacitor to ground.

Therefore the 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:

$$p_{switch} = \frac{V_{DD}C_L V_{DD}}{T} = C_L V_{DD}^2 f_{clock}$$

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 invertor 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.

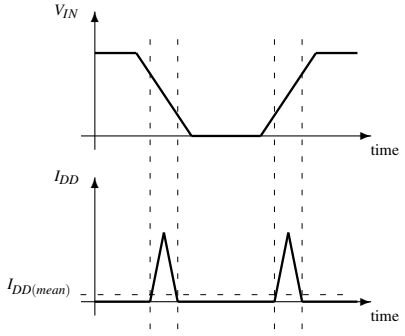


Figure B.11: **Direct flow of current from V_{DD} to ground.** This current due to the non-zero edge to the circuit input can be neglected.

The amount of power wasted by these temporary short-cuts is:

$$p_{sc} = I_{DD(mean)} V_{DD}$$

where $I_{DD(mean)}$ 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 65*nm* 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.

$$p_{leakage} = I_{leakage} V_{DD}$$

where $I_{leakage}$ 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 $V_{in} = 0$.

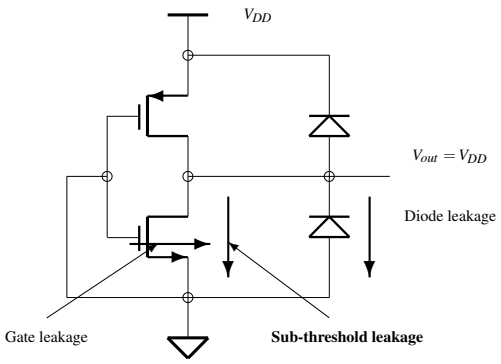


Figure B.12: **The two main components of the leakage current.** .

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.

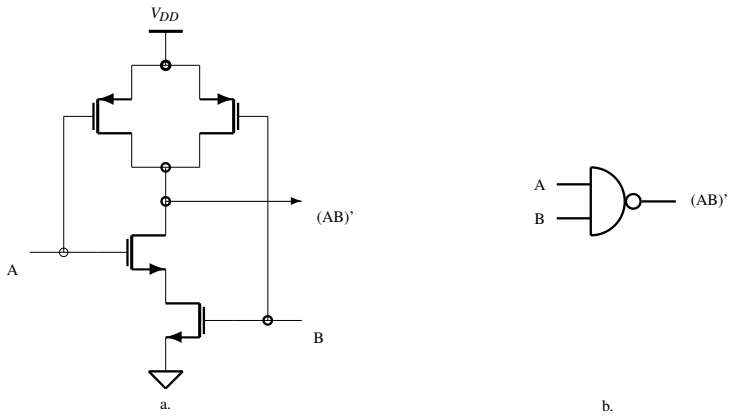


Figure B.13: **The NAND gate. a**. The internal structure of a NAND gate: the output is 1 when at least one input is 0. **b**. The logic symbol for NAND.

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, $R_{ONp} > R_{ONn}$.

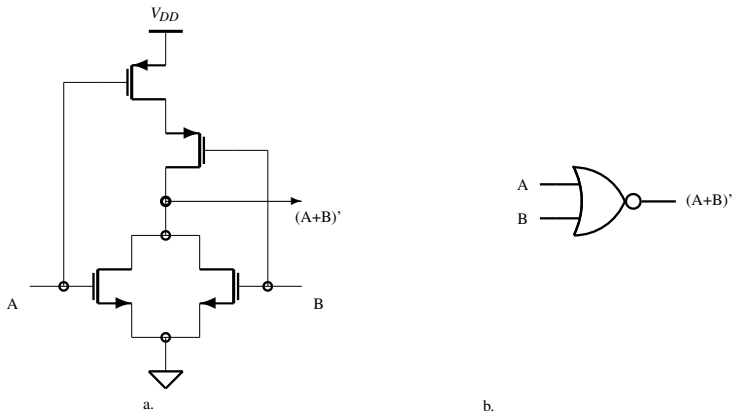


Figure B.14: **The NOR gate. a**. The internal structure of a NOR gate: the output is 1 only when both inputs are 0. **b**. The logic symbol for NOR.

Power consumption & switching activity

The power consumption is determined by the 0 to 1 transitions of the output of a logic gates. 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_{0-1}, is given by the logic function. We define ***switching activity***, σ, this probability of switching from 0 to 1.

Switching activity for 2-input AND with the inputs A and B is:

$$\sigma = P_{0-1} = P_{OUT=0}P_{OUT=1} = (1 - P_A P_B)P_A P_B$$

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).

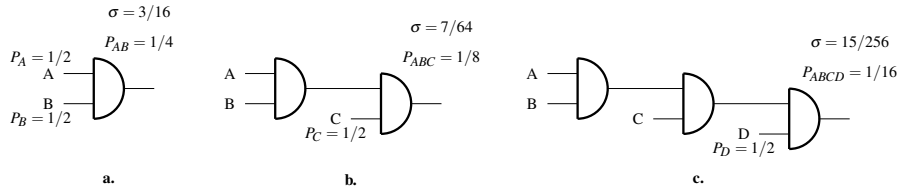


Figure B.15: **Switching activity σ and the output probability of 1. a**. For 2-input AND. **b**. For 3-input AND. **c.** For 4-input AND.

If the input are not conditioned, $P_A = P_B = 0.5$, then the switching activity for a 2-input NAND is $\sigma_{NAND2} = 3/16$ (see Figure B.15a).

Switching activity for 3-input AND with the inputs A, B, and C is $\sigma_{NAND3} = 7/64$ (see Figure B.158). The probability of 1 to the output of a 3-input AND is only $1/8$ leading to a smaller σ.

Switching activity for n-input AND is:

$$\sigma_{NANDn} = \frac{2^n - 1}{2^{2n}} \simeq \frac{1}{2^n}$$

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 σC_L:

$$p_{switch} = \sigma C_L V_{DD}^2 f_{clock}$$

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 $\sigma_{global} \simeq 1/8$. Actual measurements provide frequently $\sigma_{global} \simeq 1/10$.

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

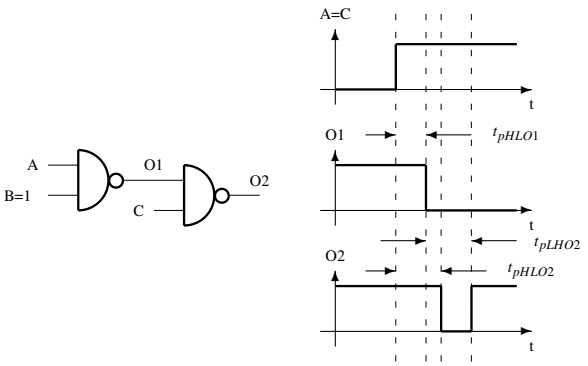


Figure B.16: **Glitching effect.** When the input value switch from $ABC = 010$ to $ABC = 111$ the output of the circuit must remain on 1. But, a short glitch occurs because of the delay, t_{pHLO1}, introduced by the first NAND.

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, discharge 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:

$$(4RONn + R_{ONp})/2$$

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.

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 $m \times R_{ON}$, 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

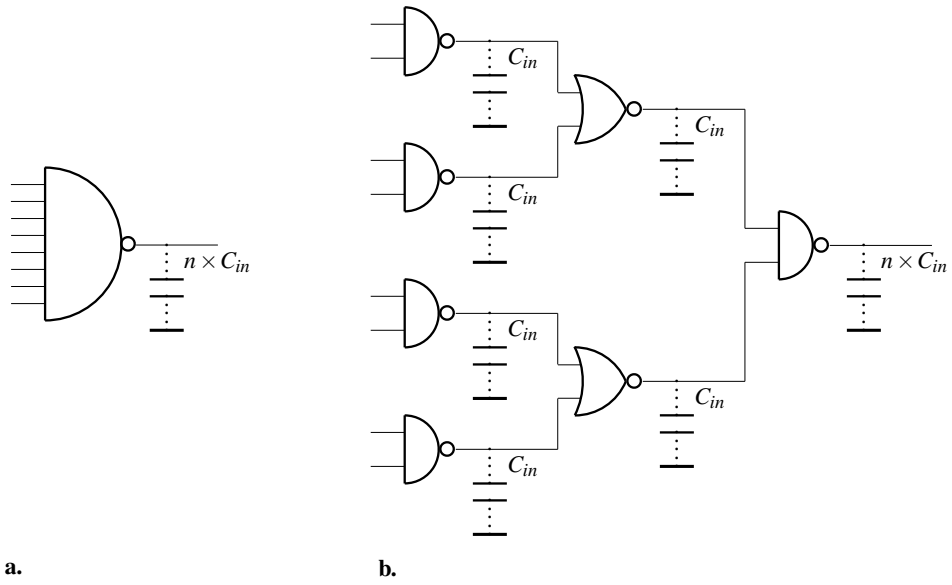


Figure B.17: **How to manage a many-input gate.** **a** An $NAND_8$ gate with fan-out n. **b**. The log-depth equivalent circuit.

in four NANDs and the second level of two ANDs in two NORs). While the maximum propagation time for the 8-input NAND is

$$t_{pHL(one-level)} = k_n \times 8 \times R_{ONn} \times (n \times C_{in})$$

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

$$t_{pHL(log-levels)} = k_n((2 \times 2 \times R_{ONn} \times C_{in}) + 2 \times R_{ONn} \times (n \times C_{in}))$$

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, α, 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.

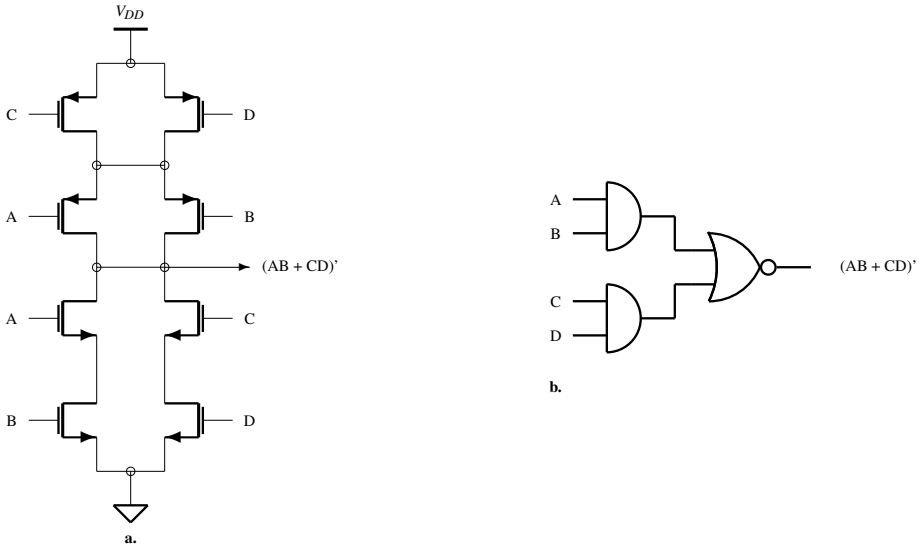


Figure B.18: **The AND-NOR gate. a**. The circuit. **b**. The logic symbol for the AND-NOR gate.

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.

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.

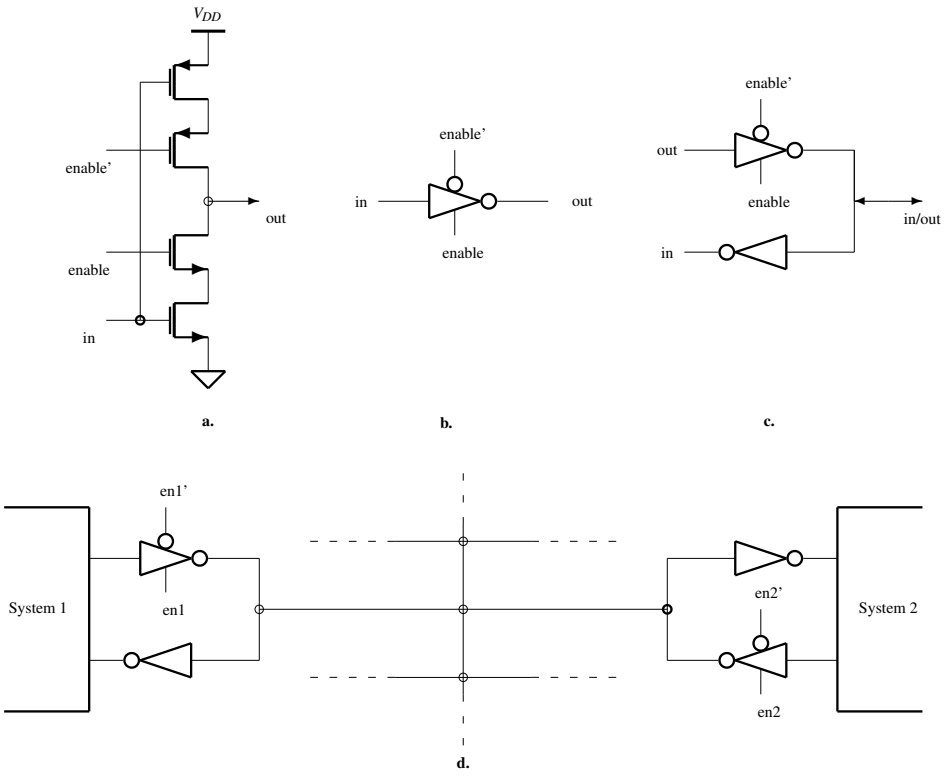


Figure B.19: **Tristate inverting buffer.** **a**. The circuit. **b**. The logic symbol for the inverting tristate buffer. **c.** Two-direction connection on one wire. For `enable = 1, in/out = out'`, while for `enable = 0, in = in/out'`. **d.** Interconnecting two systems. For `en1 = 1, en2 = 0`, System 1 sends and System 2 receives; for `en1 = 0, en2 = 1`, System 2 sends and System 1 receives; `en1 = en2 = 0` booth systems are receivers, while `en1 = en2 = 1` is not allowed.

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.

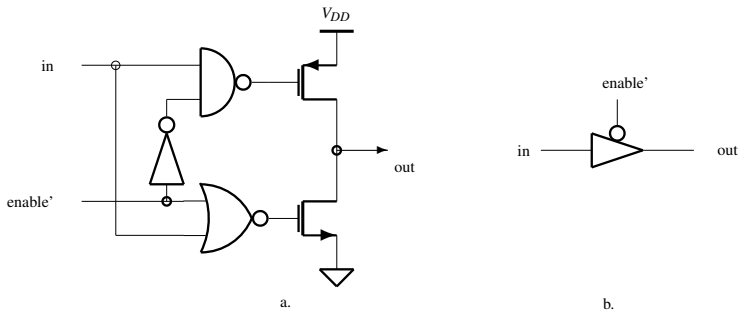


Figure B.20: **Tristate non-inverting buffer.** **a**. The circuit. **b**. The logic symbol for the non-inverting tristate buffer.

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 on transistors 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.

When the propagation time is not critical the use of this gate is recommended because, both, area and power are saved.

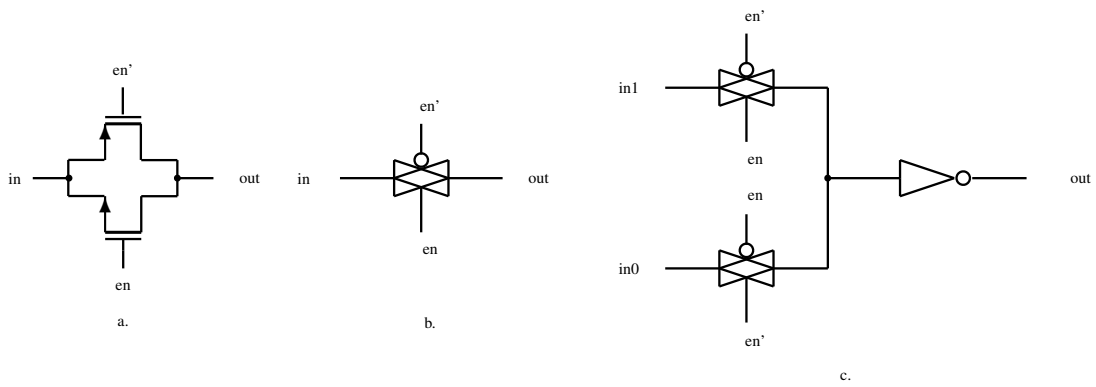


Figure B.21: **The transmission gate.** **a**. The complementary transmission gate. **b**. The logic symbol. **c**. An application: the elementary inverting multiplexer.

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

Problem B.2

Problem B.3

Problem B.4

Flop-flops

Problem B.5

Problem B.6

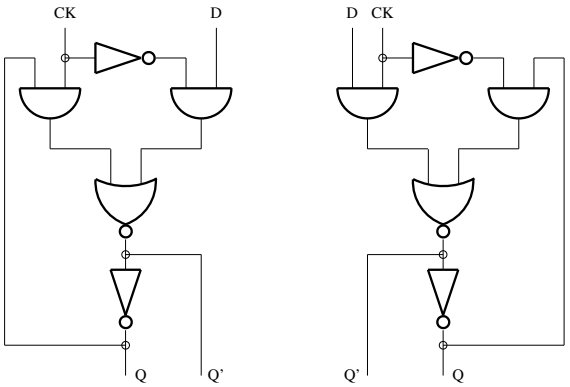


Figure B.22: **Data latches. a**. Transparent from D to Q (D = Q) for `ck = 0`. For `ck = 1` the loop is closed and D input has no effect on output. **b**. Transparent from D to Q for `ck = 1`. For `ck = 0` the loop is closed and D input has no effect on output.

Problem B.7

Problem B.8

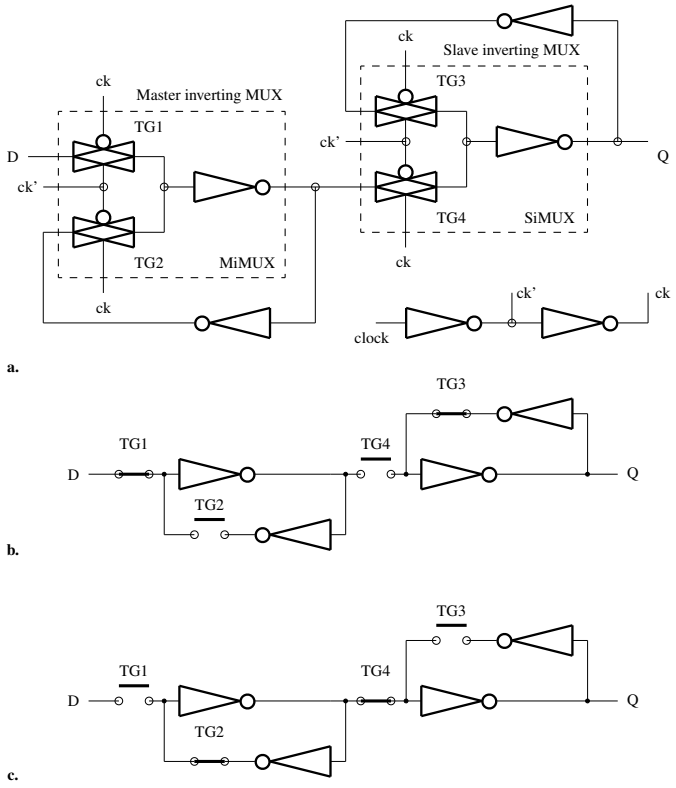


Figure B.23: **Master-slave delay flip-flop (DF-F) with the `clock` signal active on the positive transition.** **a**. Implemented with data latches based on transmission gates. **b**. The equivalent schematic for `ck = 0`. **c**. The equivalent schematic for `ck = 1`.

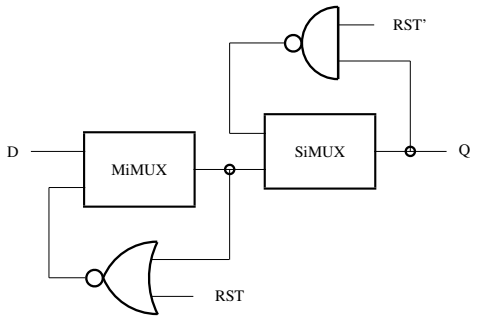


Figure B.24: **Master-slave delay flip-flop with asynchronous reset.**

Appendix C

Introduction in ADC & DAC Convertors

This appendix contains a brief introduction to AD conversion and DA conversion. The aim is to give a preliminary picture of what it means to convert from analog to digital and vice versa. Presentation involves knowledge of the concept of operational amplifier and how it is used to deal with a comparator and a voltage amplifier. Also, the function of the digital priority encoder circuit must be known (see subsection 6.1.4).

C.1 Analog circuits

The operational amplifier is a concept that refers to an ideal circuit that is quite well approximated by real circuits.

Figure C.1 shows the symbol used for the operational amplifier. In the ideal case the amplification A is infinite (in reality it is very large, usually 10,000+). Another important characteristic of operational amplifiers is that they have a high input impedance Z_{in}. Input impedance is measured between the negative and positive input terminals, and its ideal value is infinity, which minimizes loading of the source. Also, an operational amplifier ideally has zero output impedance, Z_{out}.

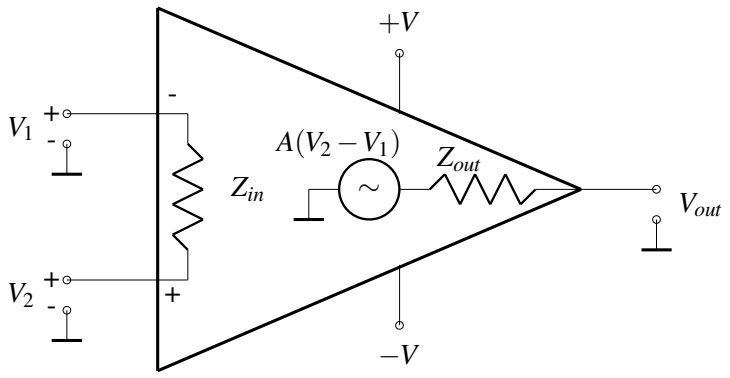


Figure C.1: Operational amplifier

We will use the operational amplifier in two established configurations: to implement the analog

comparison function and to perform the amplification used for the analog summation.

The operation of an analog comparator (see Figure C.2a) is the generation of binary-valued voltages that switch between the two levels when an analog input crosses a threshold voltage, V_{th}. Because

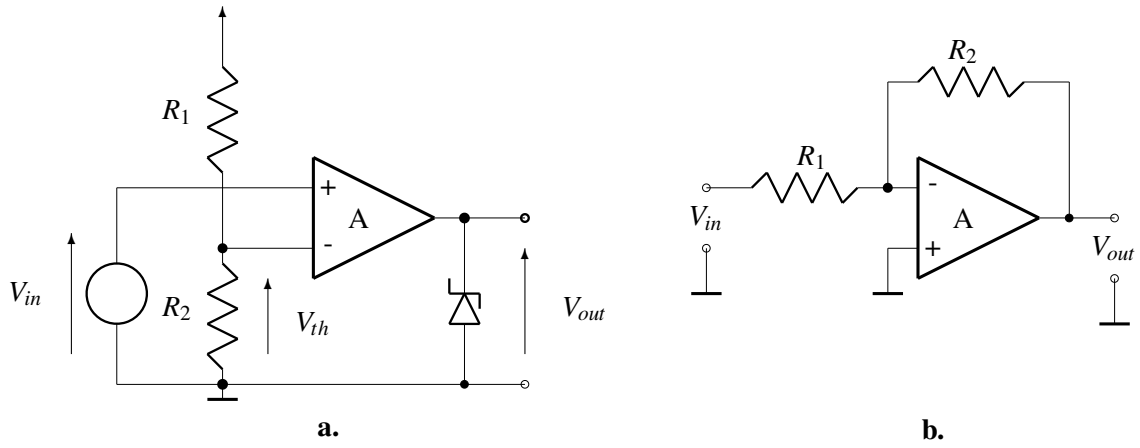


Figure C.2: Operational amplifier applications. **a.** Analog comparator. **b.** Amplifier.

$$V_{out} = A(V_{in} - V_{th})$$

a practical approximate model for the comparator is given by:

$$V_{out} = V_z \text{ for } V_{in} > V_{th}$$
$$V_{out} \simeq 0 \text{ for } V_{in} < V_{th}$$

where V_z is the Zener voltage. Because A is infinite (actually very big) the output switches as soon as the input value reaches the threshold value, ensuring a very accurate threshold detection.

An inverting operational amplifiers (see Figure C.2b) is based on the fat that the operational amplifiers forces the negative terminal to equal the positive terminal, which is connected to ground. Indeed, the very high value of A generates an appropriate value on the output for a very small, practically zero, value of $V_1 - V_2$. Thus, V_2, the inverting input, is practically connected to zero. Therefore the currents flowing through the resistors R_1 and R_2 are identical. Results:

$$\frac{V_{in}}{R_1} = -\frac{V_{out}}{R_2}$$

and the transfer function of the inverting amplifier is:

$$a = \frac{V_{out}}{V_{in}} = -\frac{R_2}{R_1}$$

C.2 ADC

The analog-digital conversion is based on the use of comparators and a resistor network. The accuracy with which the conversion is performed depends on the accuracy with which the resistance of the resistors is ensured and on the accuracy with which the comparators work.

For $V_{in} = 0$ all comparators have zero output. For $V_{in} > 0$ a number of comparators are activated and the encoder inputs are active from I_0 to I_i. Then the output of the encoder will generate the number i represented in binary code.

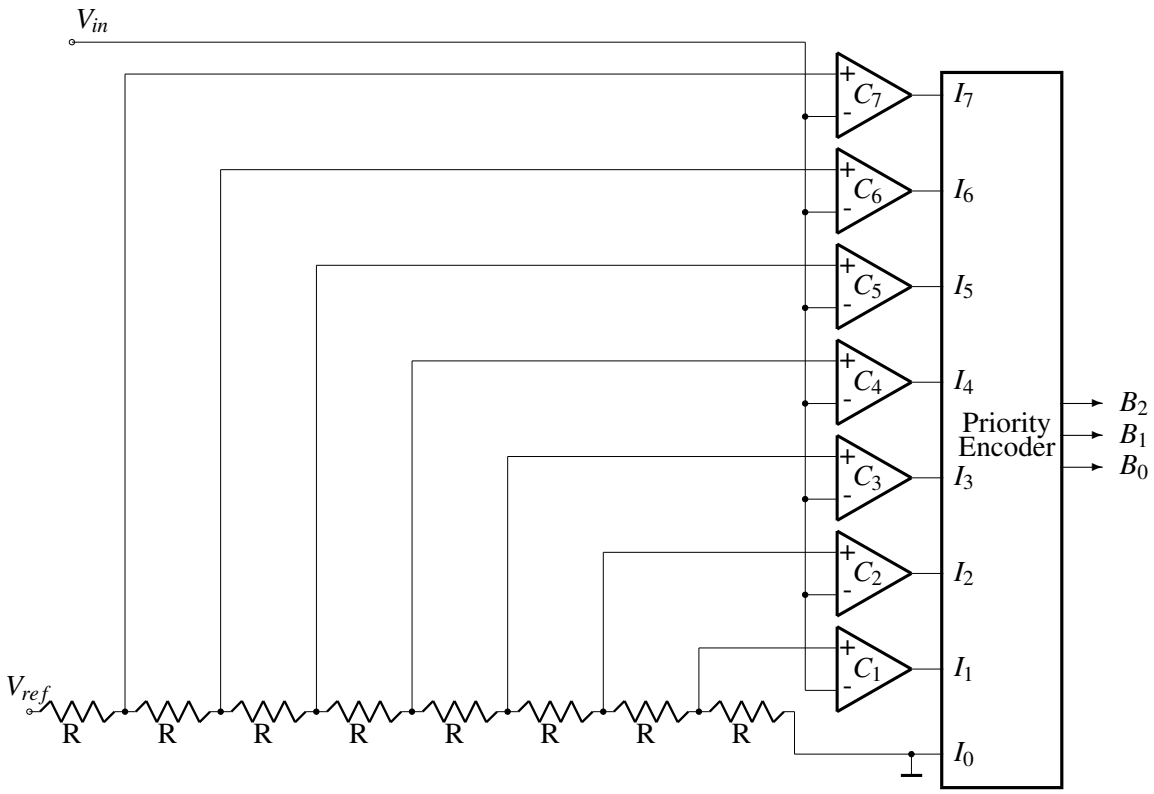


Figure C.3: ADC

C.3 DAC

For digital-to-analog conversion, a multi-input amplifier is used that allows the summation of several currents passing through resistors subjected to the same potential. The size of the resistors is inversely proportional to the associated binary order. Figure C.4 shows a DAC that converts 3-bit binary numbers. MSB is associated with the lowest resistance, of R value. The middle bit controls the current through a $2R$ value resistor, and the LSB commands a $4R$ value resistor. The sum of the currents passing through these resistors is equal to the current flowing through the reaction resistor R connected from the output of the operational amplifier to its reversing input.

If B_i, for $= 0, 1, 2$, takes value in the set $\{0, 1\}$ and the truth value 0 is represented by 0 V and the truth

value 1 is represented by V_{DD}, then because the input current on the inverting input of the operational amplifier is zero we can write:

$$\frac{B_0}{2^2}+\frac{B_1}{2^1}+\frac{B_2}{2^0}=-\frac{V_{out}}{R}$$

and the output of the circuit represented in Figure C.4 results:

$$V_{out}=-V_{DD}(B_2/2^0+B_1/2^1+B_0/2^2)$$

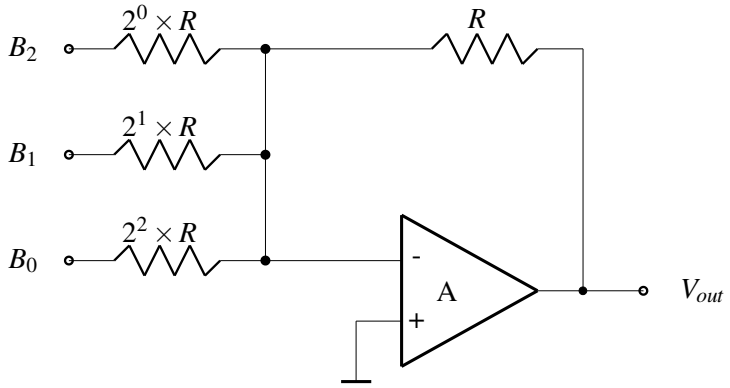


Figure C.4: DAC

For example, if $\{B_2, B_1, B_0\} = 101$, then the value on the output of the amplifier is: $1.25V_{DD}$.

Bibliography

[Alfke '73] Peter Alfke, Ib Larsen (eds.): The TTL Applications Handbook. Prepared by the Digital Application Staff of Fairchild Semiconductor, August 1973.

[Alfke '05] Peter Alfke: "Metastable Recovery in Virtex-II Pro FPGAs", *Application Note: Virtex-II Pro Family*, `http://www.xilinx.com/support/documentation/application_notes/xapp094.pdf`, XILINX, 2005.

[Andonie '95] Răzvan Andonie, Ilie Gârbacea: *Algoritmi fundamentali. O perspectivă C^{++}*, Ed. Libris, Cluj-Napoca, 1995. (in Roumanian)

[Ajtai '83] M. Ajtai, et al.: "An O(n log n) sorting network", Proc. 15th Ann. ACM Symp. on Theory of Computing, Boston, Mass., 1983.

[Batcher '68] K. E. Batcher: "Sorting networks and their applications", in *Proc. AFIPS Spring Joint Computer Conference*, vol. 32, 1968.

[Benes '68] Václav E. Beneš: *Mathematical Theory of Connecting Networks and Telephone Traffic*. New York: Academic, 1968.

[1] T. R. Blakeslee: *Digital Design with Standard MSI and LSI*, John Wiley & Sons, 1979.

[Booth '67] T. L. Booth: *Sequential Machines and Automata Theory*, John Wiley & Sons, Inc., 1967.

[Bremermann '62] H. G. Bremermann: "Optimization through Evolution and Recombination", in *Self-Organizing Systems*, ed.: M. C. Yovits, S. Cameron, Washington DC, Spartan, 1962.

[Calude '82] Cristian Calude: *Complexitatea calculului. Aspecte calitative* (The Complexity of Computation. Qualitative Aspects), Ed. Stiintifica si Enciclopedica, Bucuresti, 1982.

[Calude '94] Cristian Calude: *Information and Randomness*, Springer-Verlag, 1994.

[Casti '92] John L. Casti: *Reality Rules: II. Picturing the World in Mathematics - The Frontier*, John Wiley & Sons, Inc., 1992.

[Cavanagh '07] Joseph Cavanagh: *Sequential Logic. Analysis and Synthesis*, CRC Taylor & Francis, 2007.

[Chaitin '66] Gregory Chaitin: "On the Length of Programs for Computing Binary Sequences", *J. of the ACM*, Oct., 1966.

[Chaitin '70] Gregory Chaitin: "On the Difficulty of Computation", in *IEEE Transactions of Information Theory*, ian. 1970.

[Chaitin '77] Gregory Chaitin: "Algorithmic Information Theory", in *IBM J. Res. Develop.*, Iulie, 1977.

[Chaitin '87] Gregory Chaitin: *Algorithmic Information Theory*, Cambridge University Press, 1987.

[Chaitin '90] Gregory Chaitin: *Information, Randomness and Incompletness*, World Scientific,1990.

[Chaitin '94] Gregory Chaitin: *The Limits of Mathematics IV*, IBM Research Report RC 19671, e-print chao-dyn/9407009, July 1994.

[Chaitin '06] Gregory Chaitin: "The Limit of Rason", in *Scientific American*, Martie, 2006.

[Chomsky '56] Noam Chomsky, "Three Models for the Description of Languages", *IEEE Trans. on Information Theory*, 2:3 , 1956.

[Chomsky '59] Noam Chomsky, "On Certain Formal Properties of Grammars", *Information and Control*, 2:2, 1959.

[Chomsky '63] Noam Chomsky, "Formal Properties of Grammars", *Handbook of Mathematical Psychology*, Wiley, New-York, 1963.

[Church '36] Alonzo Church: "An Unsolvable Problem of Elementary Number Theory", in *American Journal of Mathematics*, vol. 58, pag. 345-363, 1936.

[Clare '72] C. Clare: *Designing Logic Systems Using State Machines*, Mc Graw-Hill, Inc., 1972.

[Cormen '90] Thomas H. Cormen, Charles E. Leiserson, Donsld R. Rivest: *Introduction to Algorithms*, MIT Press, 1990.

[Dascălu '98] Monica Dascălu, Eduard Franţi, Gheorghe Ştefan: "Modeling Production with Artificial Societies: the Emergence of Social Structure", in S. Bandini, R. Serra, F. Suggi Liverani (Eds.): *Cellular Automata: Research Towars Industry. ACRI '98 - Proceedings of the Third Conference on Cellular Automata for Research and Industry*, Trieste, 7 - 9 October 1998, Springer Verlag, 1998. p 218 - 229.

[Dascălu '98a] Monica Dascălu, Eduard Franţi, Gheorghe Ştefan: "Artificial Societies: a New Paradigm for Complex Systems' Modeling", in *IFAC Conference on Supplemental Ways for Improving International Stability - SWIIIS '98*, May 14-16, Sinaia, 1998. p.62-67.

[Drăgănescu '84] Mihai Drăgănescu: "Information, Heuristics, Creation", in Plauder, I. (ed): *Artificial Inteligence and Information Control System of Robots*, Elsevier Publishers B. V. (North-Holland), 1984.

[Drăgănescu '91] Mihai Drăgănescu, Gheorghe Ştefan, Cornel Burileanu: *Electronica functională*, Ed. Tehnică, Bucureşti, 1991 (in Roumanian).

[Einspruch '86] N. G. Einspruch ed.: *VLSI Electronics. Microstructure Science. vol. 14 : VLSI Design*, Academic Press, Inc., 1986.

[Einspruch '91] N. G. Einspruch, J. L. Hilbert: *Application Specific Integrated Circuits (ASIC) Technology*, Academic Press, Inc., 1991.

[Ercegovac '04] Miloš D. Ercegovac, Tomás Lang: *Digital Arithmetic*, Morgan Kaufman, 2004.

[Flynn '72] Flynn, M.J.: "Some computer organization and their affectiveness", *IEEE Trans. Comp.* C21:9 (Sept. 1972), pp. 948-960.

[Gheolbanoiu '14] Alexandru Gheolbanoiu, Dan Mocanu, Radu Hobincu, Lucian Petrica: "Cellular Automaton pRNG with a Global Loop for Non-Uniform Rule Control", 18th International Conference on Ciruits, Systems, Communications and Computers (CSCC 2014), Santorini Island, Greece, July 17-21, 2014, vol. II, 415-420. `http://www.euroment.org/library/2014/santorini/bypaper/COMPUTERS/COMPUTERS2-15.pdf`

[Glushkov '66] V. M. Glushkov: *Introduction to Cybernetics*, Academic Press, 1966.

[Gödels '31] Kurt Gödel: "On Formally Decidable Propositions of Principia Mathematica and Related Systems I", reprinted in S. Fefermann et all.: *Collected Works I: Publications 1929 - 1936,* Oxford Univ. Press, New York, 1986.

[Hartley '95] Richard I. Hartley: *Digit-Serial Computation*, Kulwer Academic Pub., 1995.

[Hascsi '95] Zoltan Hascsi, Gheorghe Ştefan: "The Connex Content Addressable Memory (C^2AM)", *Proceedings of the Twenty-first European Solid-State Circuits Conference*, Lille -France, 19-21 September 1995, pp. 422-425.

[Hascsi '96] Zoltan Hascsi, Bogdan Mîţu, Mariana Petre, Gheorghe Ştefan, "High-Level Synthesis of an Enchanced Connex memory", in *Proceedings of the International Semiconductor Conference*, Sinaia, October 1996, p. 163-166.

[Head '87] T. Head: "Formal Language Theory and DNA: an Analysis of the Generative Capacity of Specific Recombinant Behaviours", in *Bull. Math. Biology*, 49, p. 737-759, 1987.

[Helbing 89'] Walter A. Helbing, Veljko M. Milutinovic: "Architecture and Design of a 32-bit GaAs Microprocessor", in [Milutinovic 89'].

[Hennessy '07] John L. Hennessy, David A. Patterson: *Computer Architecture: A Quantitative Approach*, Fourth Edition, Morgan Kaufmann, 2007.

[Hennie '68] F. C. Hennie: *Finite-State Models for Logical Machine*, John Wiley & Sons, Inc., 1968.

[Hillis '85] W. D. Hillis: *The Connection Machine*, The MIT Press, Cambridge, Mass., 1985.

[Kaeslin '01] Hubert Kaeslin: *Digital Integrated Circuit Design*, Cambridge Univ. Press, 2008.

[Keeth '01] Brent Keeth, R. jacob Baker: *DRAM Circuit Design. A Tutorial*, IEEE Press, 2001.

[Kleene '36] Stephen C. Kleene: "General Recursive Functions of Natural Numbers", in *Math. Ann*., 112, 1936.

[Karim '08] Mohammad A. Karim, Xinghao Chen: *Digital Design*, CRC Press, 2008.

[Knuth '73] D. E. Knuth: *The Art of Programming. Sorting and Searching*, Addison-Wesley, 1973.

[Kolmogorov '65] A.A. Kolmogorov: "Three Approaches to the Definition of the Concept "Quantity of Information" ", in *Probl. Peredachi Inform.*, vol. 1, pag. 3-11, 1965.

[Kung '79] H. T. Kung, C. E. Leiserson: "Algorithms for VLSI processor arrays", in [Mead '79].

[Ladner '80] R. E. Ladner, M. J. Fischer: "Parallel prefix computation", *J. ACM*, Oct. 1980.

[Lindenmayer '68] Lindenmayer, A.: "Mathematical Models of Cellular Interactions in Development I, II", *Journal of Theor. Biology*, 18, 1968.

[Maliţa '06] Mihaela Maliţa, Gheorghe Ştefan, Marius Stoian: "Complex vs. Intensive in Parallel Computation", in *International Multi-Conference on Computing in the Global Information Technology - Challenges for the Next Generation of IT&C - ICCGI*, 2006 Bucharest, Romania, August 1-3, 2006

[Maliţa '07] Mihaela Maliţa, Gheorghe Ştefan, Dominique Thiébaut: "Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation" in *International Workshop on Advanced Low Power Systems held in conjunction with 21st International Conference on Supercomputing* June 17, 2007 Seattle, WA, USA.

[Maliţa '13] Mihaela Maliţa, Gheorghe M. Ştefan: "Control Global Loops in Self-Organizing Systems", *ROMJIST*, Volume 16, Numbers 2–3, 2013, 177-191.
`http://www.imt.ro/romjist/Volum16/Number16_2/pdf/05-Malita-Stefan2.pdf`

[Markov '54] Markov, A. A.: "The Theory of Algorithms", *Trudy Matem. Instituta im V. A. Steklova*, vol. 42, 1954. (Translated from Russian by J. J. Schorr-kon, U. S. Dept. of Commerce, Office of Technical Services, no. OTS 60-51085, 1954)

[Mead '79] Carver Mead, Lynn Convay: *Introduction to VLSI Systems*, Addison-Wesley Pub, 1979.

[MicroBlaze] *** *MicroBlaze Processor. Reference Guide.* posted at:
http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/mb_ref_guide.pdf

[Milutinovic 89'] Veljko M. Milutinovic (ed.): *High-Level Language Computer Architecture*, Computer Science Press, 1989.

[Mindell '00] Arnold Mindell: *Quantum Mind. The Edge Between Physics and Psychology*, Lao Tse Press, 2000.

[Minsky '67] M. L. Minsky: *Computation: Finite and Infinite Machine*, Prentice - Hall, Inc., 1967.

[Mîţu '00] Bogdan Mîţu, Gheorghe Ştefan, "Low-Power Oriented Microcontroller Architecture", in *CAS 2000 Proceedings*, Oct. 2000, Sinaia, Romania

[Moto-Oka '82] T. Moto-Oka (ed.): *Fifth Generation Computer Systems*, North-HollandPub. Comp., 1982.

[Omondi '94] Amos R. Omondi: *Computer Arithmetic. Algorithm, Architecture and Implementation*, Prentice Hall, 1994.

[Palnitkar '96] Samir Palnitkar: *Verilog HDL. AGuide to Digital Design and Synthesis*, SunSoft Press, 1996.

[Parberry 87] Ian Parberry: *Parallel Complexity Theory*. Research Notes in Theoretical Computer science. Pitman Publishing, London, 1987.

[Parberry 94] Ian Parberry: *Circuit Complexity and Neural Networks*, The MIT Presss, 1994.

[Patterson '05] David A. Patterson, John L.Hennessy: *Computer Organization & Design. The Hardware / Software Interface*, Third Edition, Morgan Kaufmann, 2005.

[Păun '95a] Păun, G. (ed.): *Artificial Life. Grammatical Models*, Black Sea University Press, 1995.

[Păun '85] A. Păun, Gh. Ştefan, A. Birnbaum, V. Bistriceanu, "DIALISP - experiment de structurare neconventionala a unei masini LISP", in *Calculatoarele electronice ale generatiei a cincea*, Ed. Academiei RSR, Bucuresti 1985. p. 160 - 165.

[Post '36] Emil Post: "Finite Combinatory Processes. Formulation I", in*The Journal of Symbolic Logic*, vol. 1, p. 103 -105, 1936.

[Prince '99] Betty Prince: *High Performance Memories. New architecture DRAMs and SRAMs evolution anad function*, John Wiley & Sons, 1999.

[Rafiquzzaman '05] Mohamed Rafiquzzaman: *Fundamentals of Digital Logic and Microcomputer Design*, Fifth Edition, Wiley – Interscience, 2005.

[Salomaa '69] Arto Salomaa: *Theory of Automata*, Pergamon Press, 1969.

[Salomaa '73] Arto Salomaa: *Formal Languages*, Academic Press, Inc., 1973.

[Salomaa '81] Arto Salomaa: *Jewels of Formal Language Theory*, Computer Science Press, Inc., 1981.

[Savage '87] John Savage: *The Complexity of Computing*, Robert E. Krieger Pub. Comp., 1987.

[Shankar '89] R. Shankar, E. B. Fernandez: *VLSI Computer Architecture*, Academic Press, Inc., 1989.

[Shannon '38] C. E. Shannon: "A Symbolic Annalysis of Relay and Switching Circuits", *Trans. AIEE*, vol. 57, p.713-723, 1938.

[Shannon '48] C. E. Shannon: "A Mathematical Theory of Communication", *Bell System Tech. J.*, Vol. 27, 1948.

[Shannon '56] C. E. Shannon: "A Universal Turing Machine with Two Internal States", in *Annals of Mathematics Studies, No. 34: Automata Studies*, Princeton Univ. Press, pp 157-165, 1956.

[Sharma '97] Ashok K. Sharma: *Semiconductor Memories. Techology, Testing, and Reliability*, Wiley – Interscience, 1997.

[Sharma '03] Ashok K. Sharma: *Advanced Smiconductor Memories. Architectures, Designs, and Applications*, Whiley-Interscience, 2003.

[Solomonoff '64] R. J. Solomonoff: "A Formal Theory of Inductive Inference", in *Information and Control*, vol. 7, pag. 1- 22 , pag. 224-254, 1964.

[Spira '71] P. M. Spira: "On time-Hardware Complexity Tradeoff for Boolean Functions", in *Preceedings of Fourth Hawaii International Symposium on System Sciences*, pp. 525-527, 1971.

[Stoian '07] Marius Stoian, Gheorghe Ştefan: "Stacks or File-Registers in Cellular Computing?", in *CAS, Sinaia 2007*.

[Streinu '85] Ileana Streinu: "Explicit Computation of an Independent Gödel Sentence", in *Recursive Functions Theory Newsletter*, June 1985.

[Ştefan '97] Denisa Ştefan, Gheorghe Ştefan, "Bi-thread Microcontroller as Digital Signal Processor", in *CAS '97 Proceedings, 1997 International Semiconductor Conference*, October 7 -11, 1997, Sinaia, Romania.

[Ştefan '99] Denisa Ştefan, Gheorghe Ştefan: "A Procesor Network without Interconnectio Path", in *CAS 99 Proceedings, Oct., 1999*, Sinaia, Romania. p. 305-308.

[Ştefan '80] Gheorghe Ştefan: *LSI Circuits for Processors*, Ph.D. Thesis (in Roumanian), Politechnical Institute of Bucharest, 1980.

[Ştefan '83] Gheorghe Ştefan: "Structurari neechilibrate in sisteme de prelucrare a informatiei", in *Inteligenta artificiala si robotica*, Ed. Academiei RSR, Bucuresti, 1983. p. 129 - 140.

[Ştefan '83] Gheorghe Ştefan, et al.: *Circuite integrate digitale*, Ed. Did. si Ped., Bucuresti, 1983.

[Ştefan '84] Gheorghe Ştefan, et al.: "DIALISP - a LISP Machine", in *Proceedings of the ACM Symposium on LISP and Functional Programming*, Austin, Texas, Aug. 1984. p. 123 - 128.

[Ştefan '85] Gheorghe Ştefan, A. Păun, "Compatibilitatea functie - structura ca mecanism al evolutiei arhitecturale", in *Calculatoarele electronice ale generatiei a cincea*, Ed. Academiei RSR, Bucuresti, 1985. p. 113 - 135.

[Ştefan '85a] Gheorghe Ştefan, V. Bistriceanu, A. Păun, "Catre un mod natural de implementare a LISP-ului", in *Sisteme cu inteligenta artificiala*, Ed. Academiei Romane, Bucuresti, 1991 (paper at *Al doilea simpozion national de inteligenta artificiala*, Sept. 1985). p. 218 - 224.

[Ştefan '86] Gheorghe Stefan, M. Bodea, "Note de lectura la volumul lui T. Blakeslee: Proiectarea cu circuite MSI si LSI", in *T. Blakeslee: Prioectarea cu circuite integrate MSI si LSI*, Ed. Tehnica, Bucuresti, 1986 (translated from English by M. Bodea, M. Hancu, Gh. Stefan). p. 338 - 364.

[Ştefan '86a] Gheorghe Stefan, "Memorie conexa" in *CNETAC 1986* Vol. 2, IPB, Bucuresti, 1986, p. 79 - 81.

[Ştefan '91] Gheorghe Ştefan: *Functie si structura in sistemele digitale*, Ed. Academiei Romane, 1991.

[Ştefan '91] Gheorghe Ştefan, Drăghici, F.: "Memory Management Unit - a New Principle for LRU Implementation", *Proceedings of 6th Mediterranean Electrotechnical Conference*, Ljubljana, Yugoslavia, May 1991, pp. 281-284.

[Ştefan '93] Gheorghe Ştefan: *Circuite integrate digitale*. Ed. Denix, 1993.

[Ştefan '95] Gheorghe Ştefan, Maliţa, M.: "The Eco-Chip: A Physical Support for Artificial Life Systems", *Artificial Life. Grammatical Models*, ed. by Gh. Păun, Black Sea University Press, Bucharest, 1995, pp. 260-275.

[Ştefan '96] Gheorghe Ştefan, Mihaela Maliţa: "Chaitin's Toy-Lisp on Connex Memory Machine", *Journal of Universal Computer Science*, vol. 2, no. 5, 1996, pp. 410-426.

[Ştefan '97] Gheorghe Ştefan, Mihaela Maliţa: "DNA Computing with the Connex Memory", in *RECOMB 97 First International Conference on Computational Molecular Biology*. January 20 - 23, Santa Fe, New Mexico, 1997. p. 97-98.

[Ştefan '97a] Gheorghe Ştefan, Mihaela Maliţa: " The Splicing Mechanism and the Connex Memory", *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*, Indianapolis, April 13 - 16, 1997. p. 225-229.

[Ştefan '98] Gheorghe Ştefan, "Silicon or Molecules? What's the Best for Splicing", in Gheorghe Păun (ed.): *Computing with Bio-Molecules. Theory and Experiments.* Springer, 1998. p. 158-181

[Ştefan '98a] Gheorghe Ştefan, " "Looking for the Lost Noise" ", in *CAS '98 Proceedings, Oct. 6 - 10, 1998*, Sinaia, Romania. p.579 - 582.
http://arh.pub.ro/gstefan/CAS98.pdf

[Ştefan '98b] Gheorghe Ştefan, "The Connex Memory: A Physical Support for Tree / List Processing" in *The Roumanian Journal of Information Science and Technology*, Vol.1, Number 1, 1998, p. 85 - 104.

[Ştefan '98] Gheorghe Ştefan, Robrt Benea: "Connex Memories & Rewrieting Systems", in *MELECON '98*, Tel-Aviv, May 18 -20, 1998.

[Ştefan '99] Gheorghe Ştefan, Robert Benea: "Experimente in info cu acizi nucleici", in M. Drăgănescu, Ştefan Trăusan-Matu (eds): *Natura realitatii fizice si a informatiei*, Editura Tempus, 1999.

[Ştefan '99a] Gheorghe Ştefan: "A Multi-Thread Approach in Order to Avoid Pipeline Penalties", in *Proceedings of 12th International Conference on Control Systems and Computer Science*, Vol. II, May 26-29, 1999, Bucharest, Romania. p. 157-162.

[Ştefan '00] Gheorghe Ştefan: "Parallel Architecturing starting from Natural Computational Models", in *Proceedings of the Romanian Academy*, Series A: Mathematics, Physics, Technical Sciences, Information Science, vol. 1, no. 3 Sept-Dec 2000.

[Ştefan '01] Gheorghe Ştefan, Dominique Thiébaut, "Hardware-Assisted String-Matching Algorithms", in *WABI 2001, 1st Workshop on Algorithms in BioInformatics, BRICS*, University of Aarhaus, Danemark, August 28-31, 2001.

[Ştefan '04] Gheorghe Ştefan, Mihaela Maliţa: "Granularity and Complexity in Parallel Systems", in *Proceedings of the 15 IASTED International Conf, 2004*, Marina Del Rey, CA, ISBN 0-88986-391-1, pp.442-447.

[Ştefan '06] Gheorghe Ştefan: "Integral Parallel Computation", in *Proceedings of the Romanian Academy, Series A: Mathematics, Physics, Technical Sciences, Information Science*, vol. 7, no. 3 Sept-Dec 2006, p.233-240.

[Ştefan '06a] Gheorghe Ştefan: "A Universal Turing Machine with Zero Internal States", in *Romanian Journal of Information Science and Technology*, Vol. 9, no. 3, 2006, p. 227-243

[Ştefan '06b] Gheorghe Ştefan: "The CA1024: SoC with Integral Parallel Architecture for HDTV Processing", invited paper at *4th International System-on-Chip (SoC) Conference & Exhibit*, November 1 & 2, 2006, Radisson Hotel Newport Beach, CA

[Ştefan '06c] Gheorghe Ştefan, Anand Sheel, Bogdan Mîţu, Tom Thomson, Dan Tomescu: "The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing", in *Hot Chips: A Symposium on High Performance Chips*, Memorial Auditorium, Stanford University, August 20 to 22, 2006.

[Ştefan '06d] Gheorghe Ştefan: "The CA1024: A Massively Parallel Processor for Cost-Effective HDTV", in *SPRING PROCESSOR FORUM: Power-Efficient Design*, May 15-17, 2006, Doubletree Hotel, San Jose, CA.

[Ştefan '06e] Gheorghe Ştefan: "The CA1024: A Massively Parallel Processor for Cost-Effective HDTV", in *SPRING PROCESSOR FORUM JAPAN*, June 8-9, 2006, Tokyo.

[Ştefan '07] Gheorghe Ştefan: "Membrane Computing in Connex Environment", invited paper at *8th Workshop on Membrane Computing (WMC8)* June 25-28, 2007 Thessaloniki, Greece

[Ştefan '07a] Gheorghe Ştefan, Marius Stoian: "The efficiency of the register file based architectures in OOP languages era", in *SINTES13* Craiova, 2007.

[Ştefan '07b] Gheorghe Ştefan: "Chomsky's Hierarchy & a Loop-Based Taxonomy for Digital Systems", in *Romanian Journal of Information Science and Technology* vol. 10, no. 2, 2007.

[Ştefan '14] Gheorghe M. Stefan, Mihaela Malita: "Can One-Chip Parallel Computing Be Liberated From Ad Hoc Solutions? A Computation Model Based Approach and Its Implementation", *18th International Conference on Ciruits, Systems, Communications and Computers (CSCC 2014)*, Santorini Island, Greece, July 17-21, 2014, 582-597.
http://www.inase.org/library/2014/santorini/bypaper/COMPUTERS/COMPUTERS2-42.pdf

[Sutherland '02] Stuart Sutherland: *Verilog 2001. A Guide to the New Features of the Verilog Hardware Description Language*, Kluwer Academic Publishers, 2002.

[Tabak '91] D. Tabak: *Advanced Microprocessors*, McGrow- Hill, Inc., 1991.

[Tanenbaum '90] A. S. Tanenbaum: *Structured Computer Organisation* third edition, Prentice-Hall, 1990.

[Thiébaut '06] Dominique Thiébaut, Gheorghe Ştefan, Mihaela Maliţa: "DNA search and the Connex technology" in *International Multi-Conference on Computing in the Global Information Technology - Challenges for the Next Generation of IT&C - ICCGI*, 2006 Bucharest, Romania, August 1-3, 2006

[Tokheim '94] Roger L. Tokheim: *Digital Principles*, Third Edition, McGraw-Hill, 1994.

[Turing '36] Alan M. Turing: "On computable Numbers with an Application to the Eintscheidungsproblem", in *Proc. London Mathematical Society,* 42 (1936), 43 (1937).

[Vahid '06] Frank Vahid: *Digital Design*, Wiley, 2006.

[von Neumann '45] John von Neumann: "First Draft of a Report on the EDVAC", reprinted in *IEEE Annals of the History of Computing,* Vol. 5, No. 4, 1993.

[Uyemura '02] John P. Uyemura: *CMOS Logic Circuit Design*, Kluver Academic Publishers, 2002.

[Ward '90] S. A. Ward, R. H. Halstead: *Computation Structures*, The MIT Press, McGraw-Hill Book Company, 1990.

[Wedig '89] Robert G. Wedig: "Direct Correspondence Architectures: Principles, Architecture, and Design" in [Milutinovic '89].

[Waksman '68] Abraham Waksman, "A permutation network," in *J. Ass. Comput. Mach.*, vol. 15, pp. 159-163, Jan. 1968.

[webRef_1] `http://www.fpga-faq.com/FAQ_Pages/0017_Tell_me_about_metastables.htm`

[webRef_2] `http://www.fpga-faq.com/Images/meta_pic_1.jpg`

[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`

[Weste '94] Neil H. E. Weste, Kamran Eshraghian: *Principle of CMOS VLSI Design. ASystem Perspective*, Second Edition, Addisson Wesley, 1994.

[Wolfram '02] Stephen Wolfram: *A New Kind of Science*, Wolfram Media, Inc., 2002.

[Zurada '95] Jacek M. Zurada: *Introductin to Artificial Neural network*, PWS Pub. Company, 1995.

[Yanushkevich '08] Svetlana N. Yanushkevich, Vlad P. Shmerko: *Introduction to Logic Design*, CRC Press, 2008.