Loops & Complexity
in
DIGITAL SYSTEMS

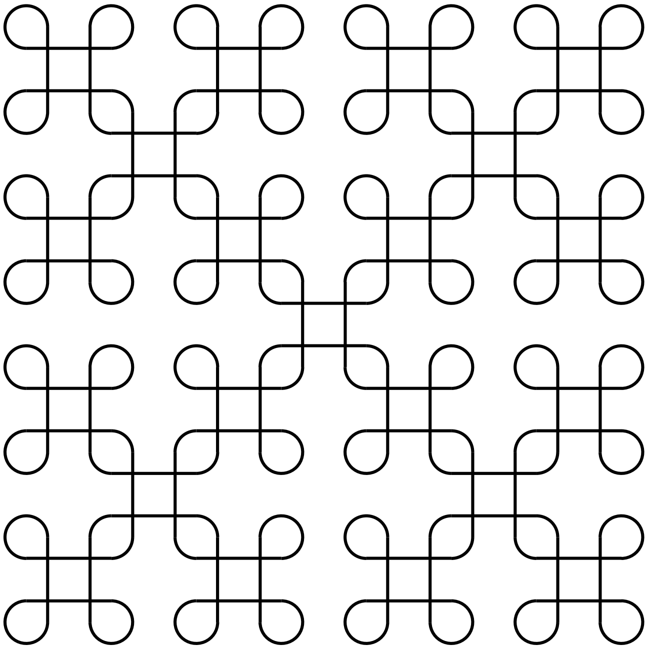
∗

Lecture Notes on Digital Electronics

Vol. 1

(work in endless progress)

Gheorghe M. Ştefan



– 2024 version –

This document was prepared with $\LaTeX\,2_\varepsilon$

Introduction

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 *Hundred Giga-Gate Per Chip Era*. The last improvements of the microelectronic technologies allow us to put on a Silicon die one hundred billions 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 *Hundred 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 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 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: *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 third 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 fourth 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 fifth 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 sixth 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)

Chapter 1

WHAT'S A DIGITAL SYSTEM?

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

1.1.1 Digital Domain

In the electronic digital domain we work with two values only (see Figure 1.1):

[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*.

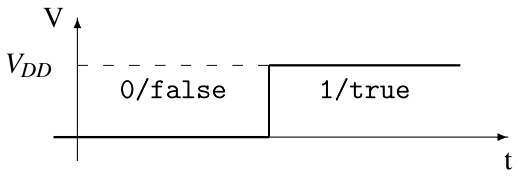


Figure 1.1: The two levels of the signal in the digital domain. Low level (0 Volt) for `0` or `false`, and high level (V_{DD} Volt) for `1` or `true`.

- 0, represented by the electrical value 0 V, having two meanings:
  - the numerical value `0`
  - the logic value `false`
- 1, represented by the electrical value V_{DD} V, having two meanings:
  - the numerical value `1`
  - the logic value `true`

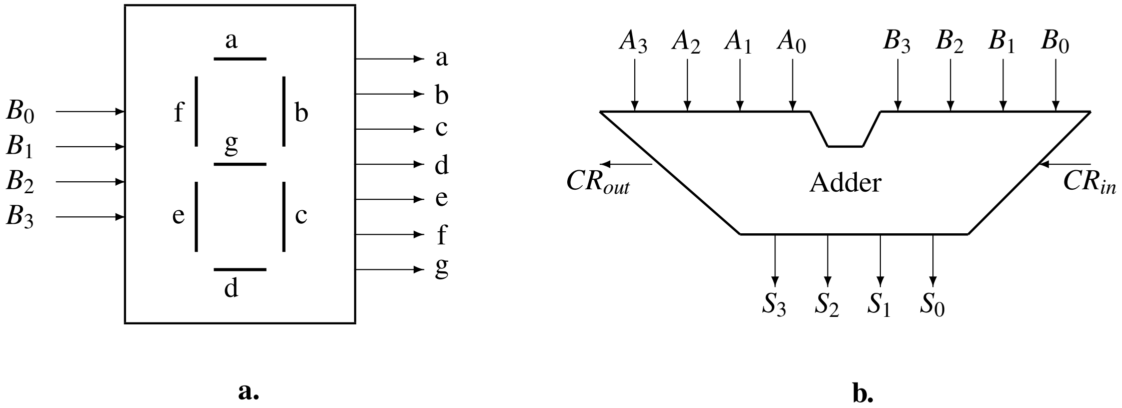


Figure 1.2: The version of digital circuits. **a.** Logic circuit: trans-coder for seven-segment display. **b.** Numeric circuit: four-bit numbers adder.

Consequently, there are two kinds of circuits (see Figure 1.2):

- logic circuits (Fig. 1.2a)
- numeric circuits (Fig. 1.2b)

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.

1.1.2 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:*

```
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, "$\cdot$", 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 System Verilog description is:

```
module ignitionKey( output  logic s,
                    input   logic 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
```

◇

Example 1.2 *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:*

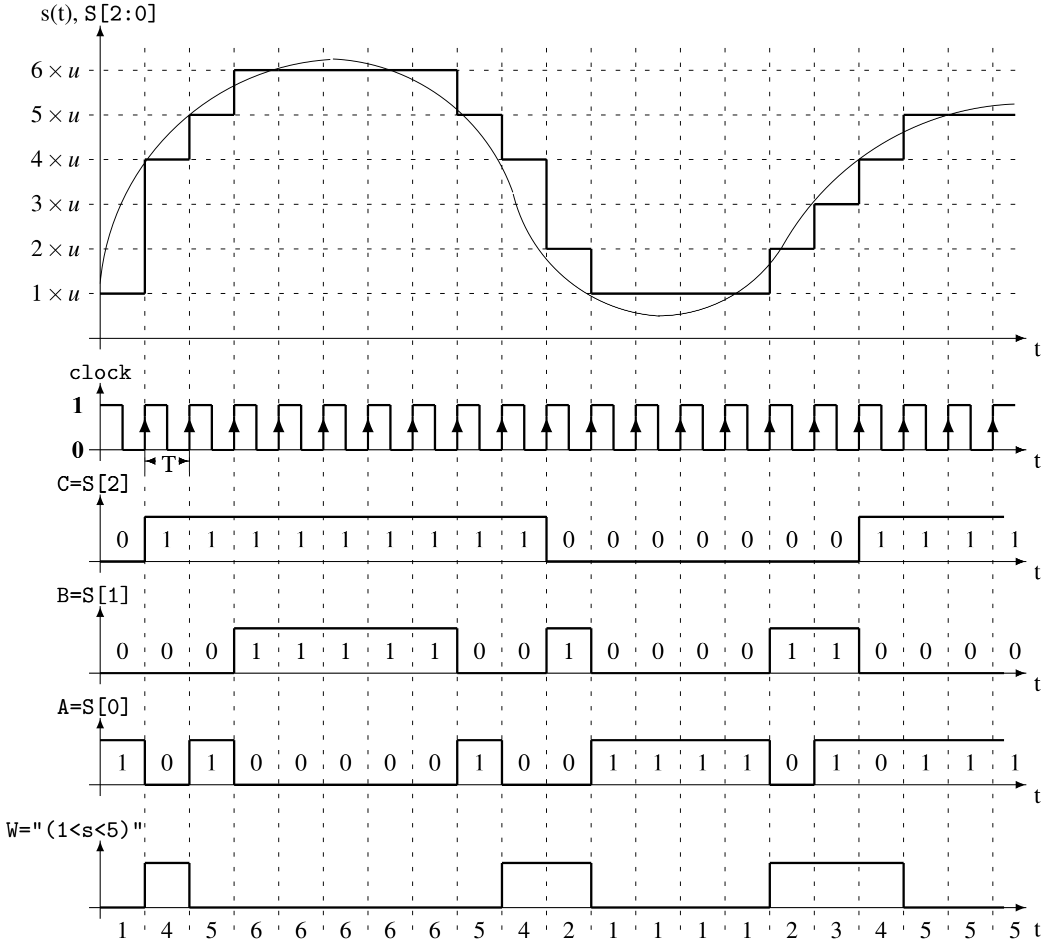


Figure 1.3: **Analogue to digital conversion.** The analog signal, $s(t)$, is sampled at each T using the unit measure u, and results the three-bit digital signal `S[2:0]`. **A first application**: the one-bit digital signal `W="(1<s<5)"` indicates, by its active value 1, the time interval when the digital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

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

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

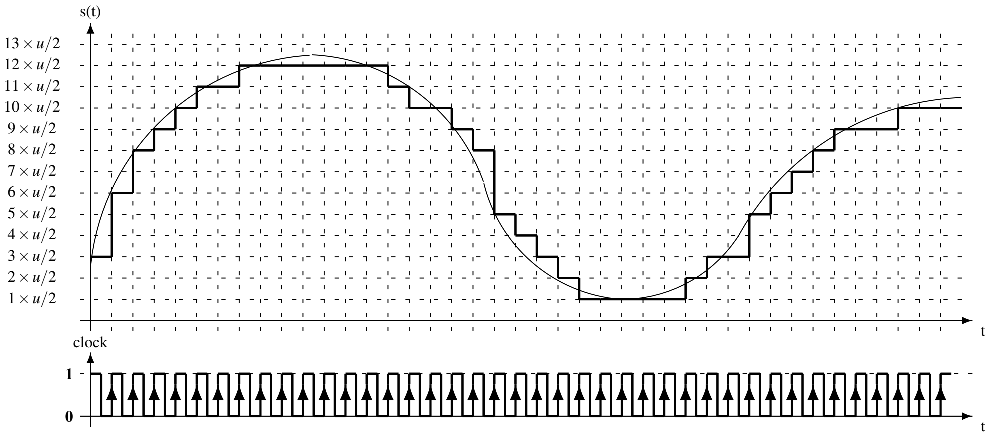


Figure 1.4: **More accurate analogue to digital.** The analogous signal is sampled at each $T/2$ using the unit measure $u/2$.

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.

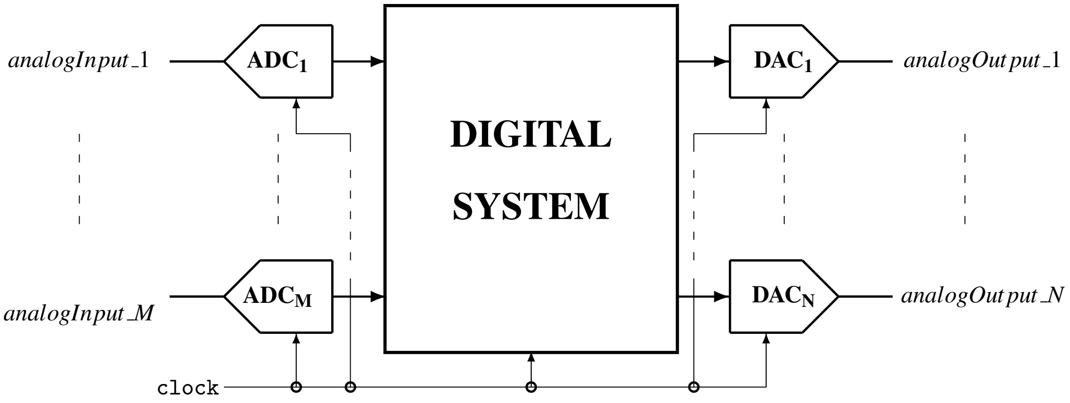


Figure 1.5: **Generic digital electronic system.**

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 1.3 *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:

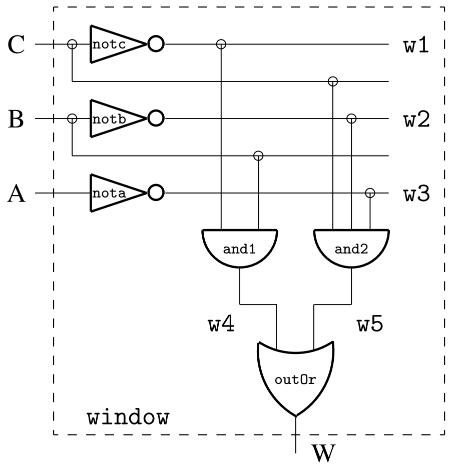
- *three NOT circuits are used for generating the negated values of the three input variables:* `C, B, A`
- *one 2-input AND circuit computes* `C'B`
- *one 3-input AND circuit computes* `CB'A'`
- *one 2-input OP circuit computes the final OR between the previous two functions.*

The circuit is simulated and synthesized using its description in the hardware description language (HDL) System 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.sv
Circuit name:    Window
Description:     the circuit detects the input in the range of (1,5)
***********************************************************************/
module window(  output  logic W,
                input   logic C, B, A);
    logic     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 System 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.sv
Circuit name:   Simulation module for simWindow.v
Description:    generate stimulus for the module simWindow.v
*********************************************************************/
module simWindow;
        logic A, B, C ;
        logic 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
```

◇

1.1.3 Modules in *System 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 System Verilog a four-input adder modulo 2^8.

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

    assign out = in0 + in1;
endmodule
```

```
/**********************************************************************
File: adder4.sv
Describes: four-input mod256 adder
**********************************************************************/
module adder4(  output logic [7:0] out,
                input  logic [7:0] in0, in1, in2,in3);

    logic [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.

1.1.4 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.

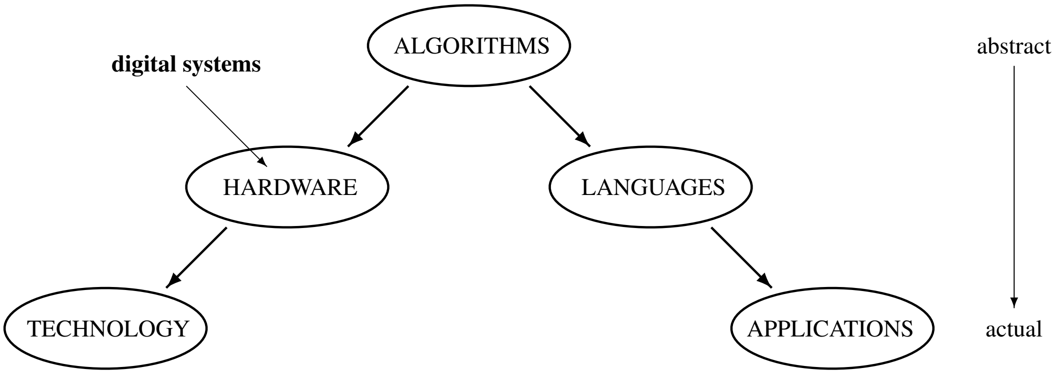


Figure 1.7: **What is computer science?** The domain of digital systems provides techniques for designing the hardware involved in computation.

Definition 1.1 *Computer science (see also Figure 1.7) means to study:*

- **algorithms***,*
- *their* **hardware** *embodiment*
- *and their* **linguistic** *expression*

with extensions toward

- *hardware* **technologies**
- *and real* **applications***.* ◇

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

The hardware embodiment of computations is done in **digital systems**. What kind of formal tools are used to describe, in the most flexible and efficient way, a complex digital system? Figure 1.8 presents the formal context in which the description tools are considered. **Pseudo-code language** is an easy to understand and easy to use way to express algorithms. Anything about computation can be expressed using this kind of languages. By the rule, in a pseudo-code language we express, for our (human) mind, preliminary, not very well formally expressed, ideas about an algorithm. The "main user" of this kind of language is only the human mind. But, for building *complex* applications or for accessing advanced technologies involved in building *big* digital systems, we need refined, rigorous formal languages and specific styles to express computation. More, for a rigorous formal language we must take into account that the "main user" is a merciless machine, instead of a tolerant human mind. Elaborated **programming languages** (such as C++, Java, Prolog, Lisp) are needed for developing complex contexts for computation and to write using them real applications. Also, for complex hardware embodiments specific **hardware description languages**, HDL, (such as Verilog, VHDL, SystemC) are proposed.

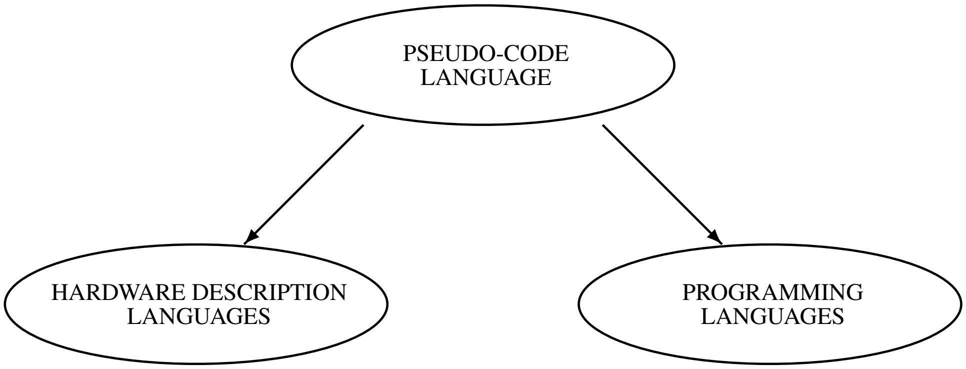


Figure 1.8: **The linguistic context in computer science.** Human mind uses pseudo-code languages to express informally a computation. To describe the circuit associated with the computation a rigorous HDL (hardware description language) is needed, and to describe the program executing the computation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

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

The starting point in designing a digital system is to describe it using what we call a **specification**, shortly, a **spec**. There are many ways to specify a digital system. In real life a hierarchy of specs are used, starting from high-level informal specs, and going down until the most detailed structural description is provided. In fact, 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,*

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

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

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

is the **output transition function**.

◇

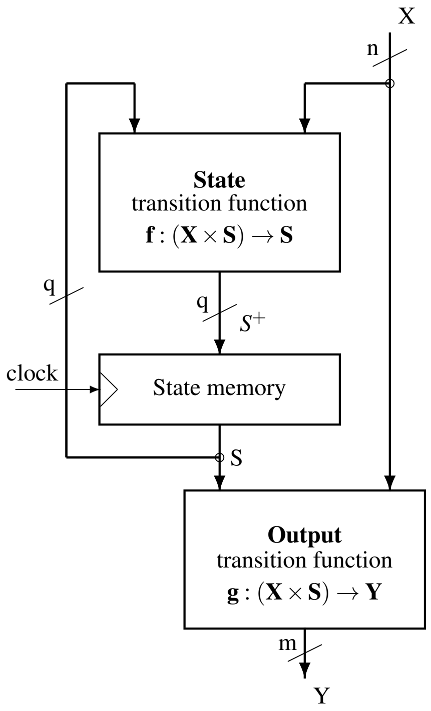

X
n
State
transition function
$\mathbf{f}: (\mathbf{X} \times \mathbf{S}) \rightarrow \mathbf{S}$
q
q
S^+
clock
State memory
S
Output
transition function
$\mathbf{g}: (\mathbf{X} \times \mathbf{S}) \rightarrow \mathbf{Y}$
m
Y

Figure 1.9: Digital system.

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

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

The internal state of the system determines the partial autonomy of the system. The system behaves on its outputs taking into account both, the current input and the current internal state.

Because all the sets involved in the previous definition have the form $\{0,1\}^b$, each of the b one-bit input, output, or state evolves in time switching between two values: 0 and 1. The previous definition specifies a system having a n-bit input, an m-bit output and a q-bit internal state. If $x_t \in X = \{0,1\}^n$, $y_t \in Y = \{0,1\}^m$, $s_t \in S = \{0,1\}^q$ 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 1.10), called ***combinational logic circuits*** – CLC –, to compute functions like $f(x_t, s_t)$ or $g(x_t, s_t)$, which *continuously follow*, by the evolution of their output values delayed with the *propagation time* t_p, any change on the inputs x_t and s_t (the shaded time interval on the wave `out` represent the transient value of the output)

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

- a storing support to memorize the state between two successive discrete moments of time is 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.

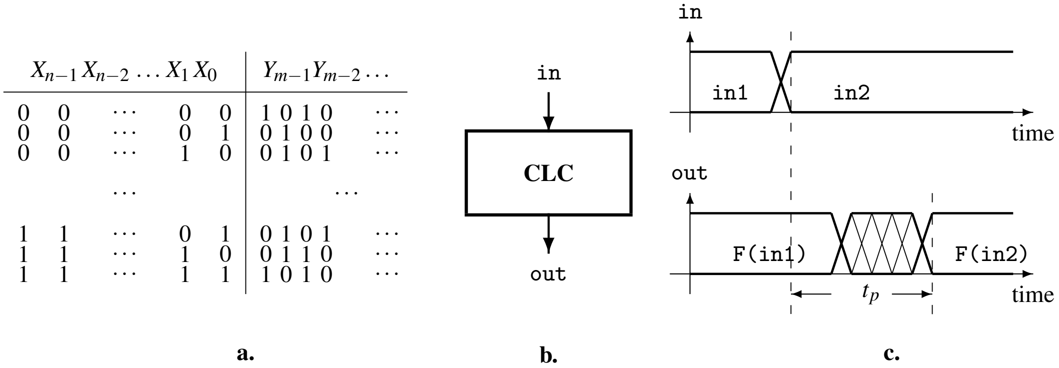


Figure 1.10: **The module for non-sequential functions. a.** The table used to define the function as a correspondence between **all** input binary configurations `in` and binary configurations `out`. **b.** The logic symbol for the *combinatorial logic circuit* – CLC – which computes `out = F(in)`.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

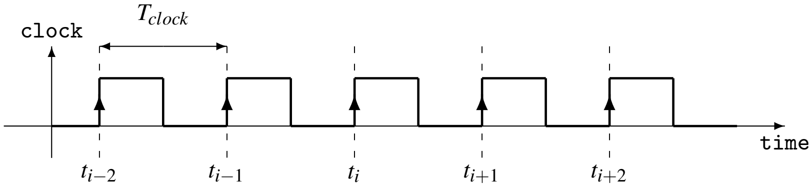


Figure 1.11: **The clock.** This clock signal is active on its positive edge (negative edge as active edge is also possible). The time interval between two positive transitions is the period T_{clock} of the clock signal. Each positive transition marks a discrete moment of time.

(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.

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.

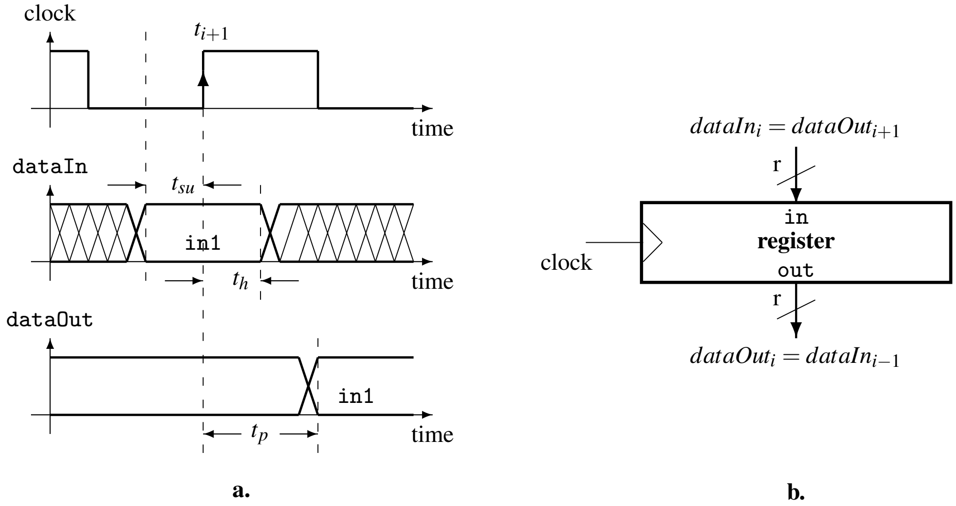


Figure 1.12: **The register. a.** The wave forms describing timing details about how the register swithces around the active edge of clock. **b.** The logic symbol used to define the static behaviour of the register when both, inputs and outputs are stable between two active edges of the clock signal.

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 System Verilog module:*

```
module fullAdder(   output logic [3:0]  out     ,
                    output logic        crOut   ,   // carry output
                    input  logic [3:0]  in0     ,
                    input  logic [3:0]  in1     ,
                    input  logic        crIn    );  // carry input
    logic [4:0] sum ;

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

[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.

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

```
module bigAdder(output logic [15:0]   out    ,
                output logic          crOut  ,  // carry output
                input  logic [15:0]   in0    ,
                input  logic [15:0]   in1    ,
                input  logic          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  logic [7:0]   out,
                    input   logic [7:0]   in1,
                    input   logic [7:0]   in2,
                    input   logic [7:0]   in3);
    logic    [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  logic [7:0]   out,
                    input   logic [7:0]   in1,
                    input   logic [7:0]   in2);
    // ...
 endmodule
```

Synthesize it to test your solution.

Problem 1.3 *Let be the schematic representation of the design* `topSyst` *in Figure 1.13. Write the Verilog description of what is described in Figure 1.13. Test the result by synthesizing it.*

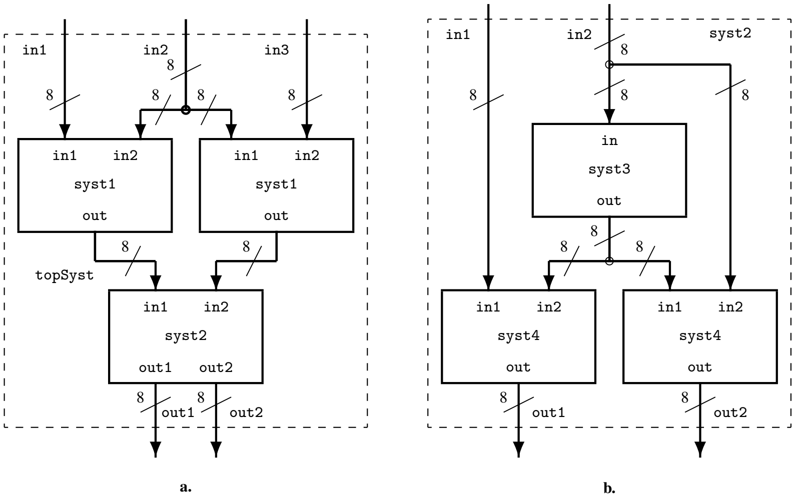


Figure 1.13: **The schematic of the design `topSyst`. a.** The top module `topSyst` **b.** The structure of the module `syst2`.

Chapter 2

GATES: Zero order, no-loop digital systems

> 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:

$$f : \{0,1\}^n \rightarrow \{0,1\}^m.$$

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!

[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.

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.

2.1 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).

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.

2.1.1 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 2.1) 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.

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.

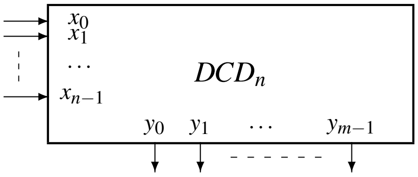


Figure 2.1: **The n-input decoder (DCD_n).**

Definition 2.1 *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.sv
Circuit name:     Decoder
Description:      behavioral description of a n-input decoder
*************************************************************************/
 module dec #(parameter inDim = n)(input  logic [inDim - 1:0]       sel,
                                   output logic [(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 2.2a) 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.

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 2.2b). 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.

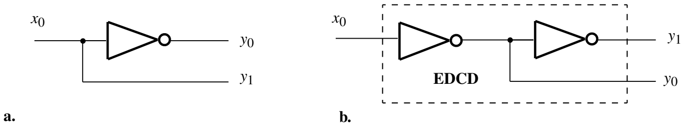


Figure 2.2: **The elementary decoder (EDCD). a.** The basic circuit. **b.** Buffered EDCD, a serial-parallel composition.

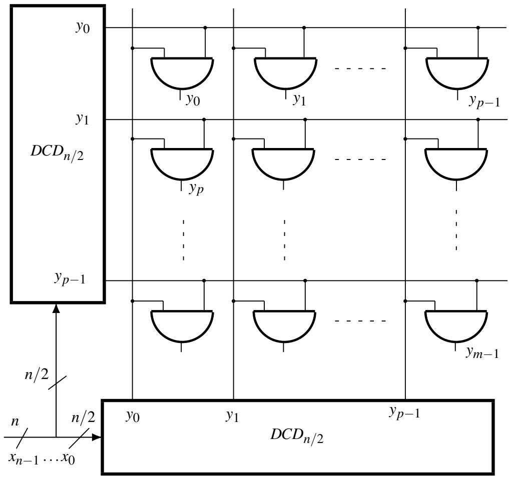


Figure 2.3: **The recursive definition of *n*-inputs decoder (DCD_n).** Two $DCD_{n/2}$ are used to drive a two dimension array of AND_2 gates. The same rule is applied for the two $DCD_{n/2}$, and so on until $DCD_1 = EDCD$ is needed.

Definition 2.2 *The structure of DCD_n is* recursive defined *by the rule represented in Figure 2.3. The DCD_1 is an EDCD (see Figure 2.2b).* ⋄

The previous definition is a constructive one, because provide an algorithm to construct a decoder for any n. It falls into the class of the *"divide & impera"* algorithms which reduce the solution of the problem for n to the solution of the same problem for $n/2$.

The quantitative evaluation of DCD_n offers the following results:

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

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 2.3) 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 2.3 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 2.4). 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.

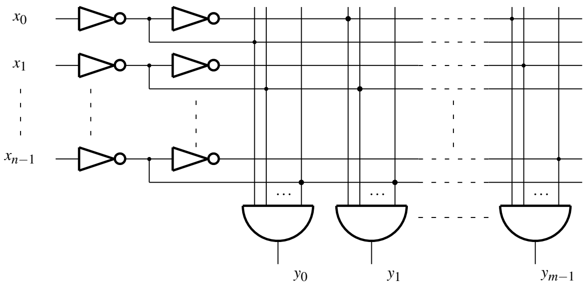


Figure 2.4: **"Constant depth" DCD** Applying the associative rule into the hierarchical network of AND_2 gates results the one level AND_n gates circuit driven by n EDCDs.

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.sv
Circuit name:    3-input Decoder
Description:     structural description of a 3-input decoder
*************************************************************************/
module dec3(output logic [7:0] out,
            input  logic [2:0] in );
 // internal connections
    logic 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 2.1 *Let be the 3-input 2-output function defined in the table from Figure 2.5. 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.*

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

⋄

2.1.2 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

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

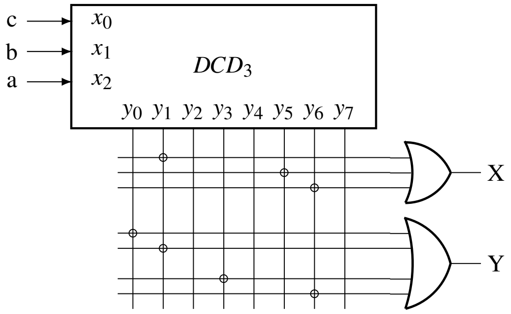


Figure 2.5:

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 2.6. 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 $O(2^n)$.

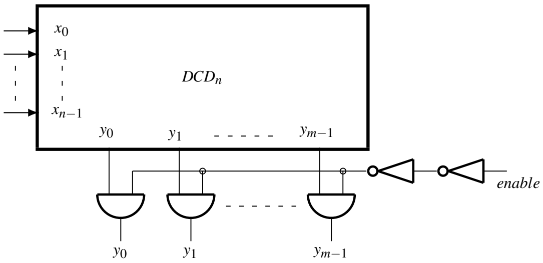


Figure 2.6: **Demultiplexor.** The n-input demultiplexor ($DMUX_n$) includes a DCD_n and 2^n AND_2 gates used to distribute the input *enable* in 2^n different places according to the n-bit selection code.

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 2.3 *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.sv
Circuit name:    Demultiplexor
Description:     behavioral description for a n-input demultiplexor
*********************************************************************** */
 module dmux #(parameter inDim = n)
        (   input  logic [inDim - 1:0]          sel    ,
            input  logic                        enable ,
            output logic [(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 2.7. 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 2.7.

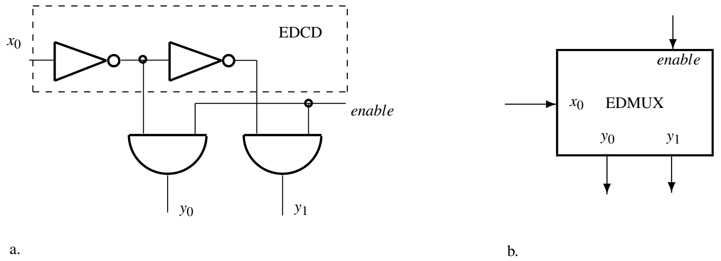


Figure 2.7: **The elementary demultiplexor. a.** The internal structure of an elementary demultiplexor (EDMUX) consists in an elementary decoder, 2 AND_2 gates, and an inverter circuit as input buffer. **b.** The logic symbol.

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

Definition 2.4 *$DMUX_n$ is defined as the structure represented in Figure 2.8, 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.

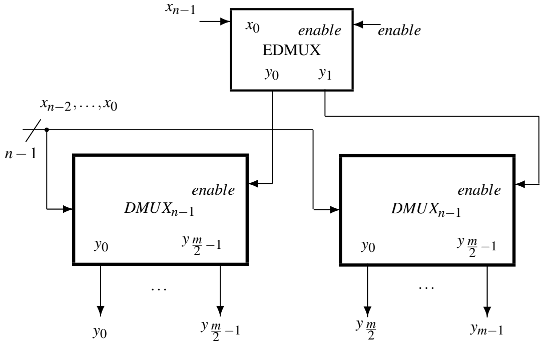


Figure 2.8: **The recursive definition of** $DMUX_n$**.** Applying the same rule for the two $DMUX_{n-1}$ a new level of 2 EDMUXs is added, and the output level is implemented using 4 $DMUX_{n-2}$. And so on until the output level is implemented using 2^{n-1} EDMUXs. The resulting circuit contains $2^n - 1$ EDMUXs.

2.1.3 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 2.9). 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 2.5 *A multiplexer MUX_n is a combinational circuit having n selection inputs $x_{n-1},\dots,x_0$ that selects to the output y one input from the $m = 2^n$ selectable inputs, $i_{m-1},\dots,i_0$. The Verilog description is:*

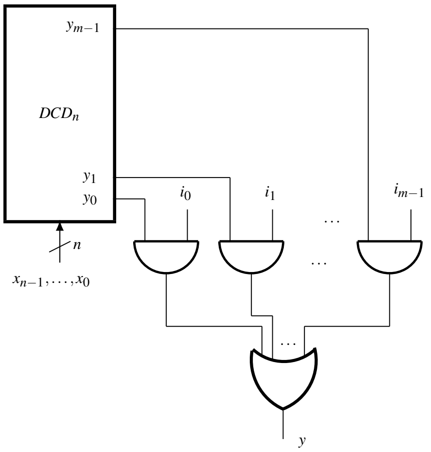


Figure 2.9: **Multiplexer.** The n selection inputs multiplexer MUX_n is made serial connecting a DCD_n with an AND-OR structure.

```
/***********************************************************************
File name:        mux.sv
Circuit name:     Multiplexor
Description:      behavioral description for a n selection inputs
                  multiplexor
***********************************************************************/
 module mux #(parameter inDim = n)
        (input  logic [inDim-1:0]         sel, // selection inputs
         input  logic [(1<<inDim)-1:0]    in , // selected inputs
         output logic                     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 2.10a, 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 2.10c).

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.

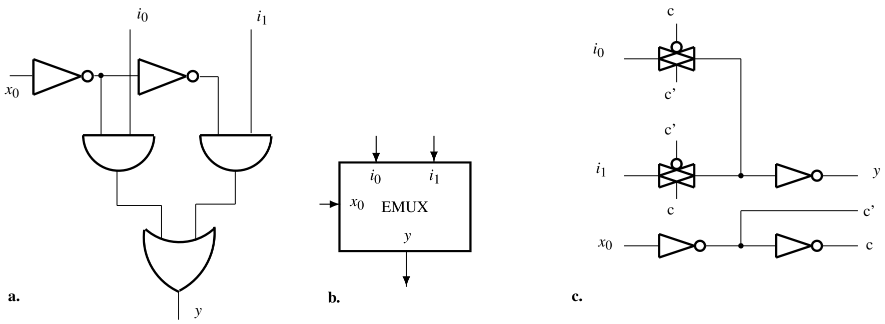


Figure 2.10: **The elementary multiplexer (EMUX). a.** The structure of EMUX containing an EDCD and the smallest AND-OR structure. **b.** The logic symbol of EMUX. **c.** A version of EMUX using transmission gates (see section *Basic circuits*).

Definition 2.6 *MUX_n can be made by serial connecting two parallel connected $MUX_{n/2}$ with an EMUX (see Figure 2.11 that is part of the definition), and $MUX_1 = EMUX$.* $\diamond$

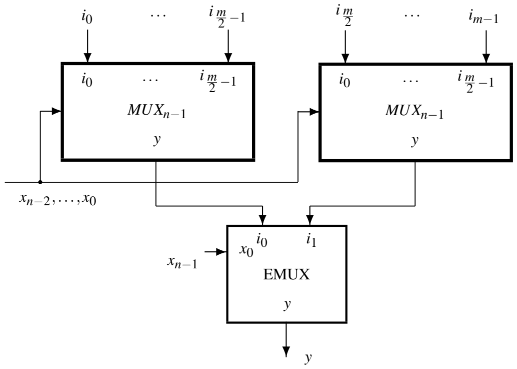


Figure 2.11: **The recursive definition of MUX_n.** Each MUX_{n-1} has a similar definition (two MUX_{n-2} and one EMUX), until the entire structure contains EMUXs. The resulting circuit is a binary tree of $2^n - 1$ EMUXs.

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:

$$y = x_{n-1}\ldots x_1 x_0 i_{m-1} + \ldots + x'_{n-1}\ldots x'_1 x_0 i_1 + x'_{n-1}\ldots x'_1 x'_0 i_0$$

any n-input logic function is specified by the binary vector $\{i_{m-1}, \ldots i_1, i_0\}$. Thus any n input logic function can be implemented with a MUX_n having on its selected inputs the binary vector defining it.

Example 2.2 *Let be function X defined in Figure 2.12 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

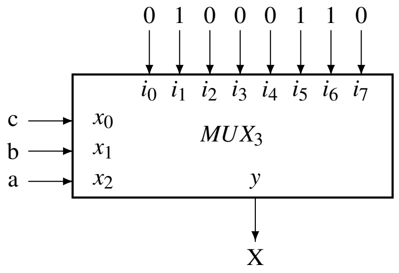


Figure 2.12:

◇

2.1.4 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

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

The increment circuit is built using as "brick" the **elementary increment circuit**, EINC, represented in Figure 2.13a, 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 2.13b: INC_n is composed using an INC_{n-1} serially connected with an EINC, where $INC_0 = EINC$.

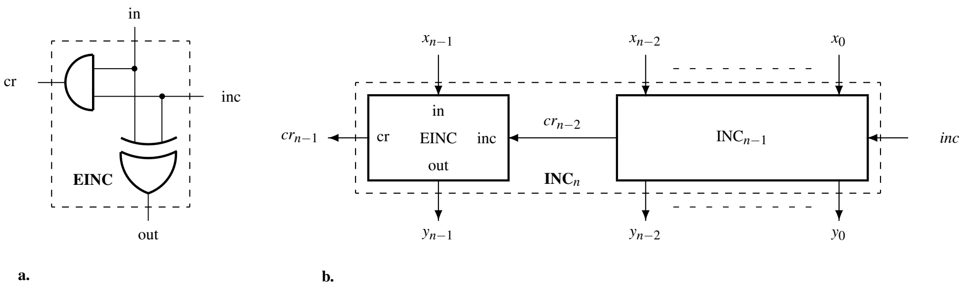


Figure 2.13: **Increment circuit.** **a.** The elementary increment circuit (called also **half adder**). **b.** The recursive definition for an n-bit increment circuit.

2.1.5 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 2.7 *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:

- $in1, in2$ *represent the i-th bits if two numbers*
- $in3$ *represents the carry signal generated by the* $i-1$ *stage of the addition process*
- $out0$ *represents the i-th bit of the result*
- $out1$ *represents the carry generated for the* $i+1$*-th stage of the addition process*

Follows the System Verilog description:

```
/************************************************************************
File name:      full_adder.sv
Circuit name:   Full Adder
Description:    behavioral description of a full adder
************************************************************************/
 module full_adder( output logic sum, carry_out,
                    input  logic 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.sv
Circuit name:   Half Adder
Description:    behavioral description of a half adder
*********************************************************************/
 module half_adder(output  logic sum, carry, input  logic in1, in2);

     assign  sum = in1 ^ in2,
             carry = in1 & in2;
 endmodule
```

◇

Note: The half adder circuit is also an elementary increment circuit (see Figure 2.13a).

Definition 2.8 *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 2.14). ADD_1 is a full adder.*

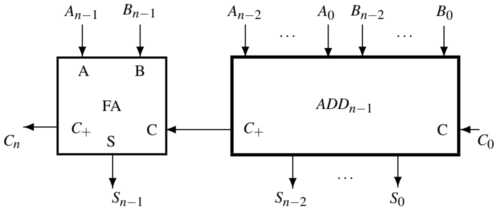


Figure 2.14: **The recursive defined n-bit ripple-carry adder (ADD_n).** ADD_n is simply designed adding to an ADD_{n-1} a full adder (FA), so as the carry signal ripples from one FA to the next.

```
/*************************************************************************
File name:      adder.sv
Circuit name:   Adder
Description:    recursive structural description of a n-bit adder using
                the conditional generate statement
*************************************************************************/
 module adder #(parameter n = 4)(output  logic [n-1:0] out ,
                                 output  logic         cry ,
                                 input   logic [n-1:0] in1 ,
                                 input   logic [n-1:0] in2 ,
                                 input   logic         cin );
    logic [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.sv
Circuit name:   Full Adder
Description:    behavioral description of a full adder
*************************************************************************/
module fullAdder(   output  logic out, cry ,
                    input   logic 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 2.3 *Generated n-bit adder:*

```
/*************************************************************************
File name:      add.sv
Circuit name:   Adder
Description:    structural description of a n-input adder using the
                generate statement
*************************************************************************/
module add #(parameter n=8)(   input   logic [n-1:0] in1, in2,
                               input   logic         cIn     ,
                               output  logic [n-1:0] out     ,
                               output  logic         cOut    );
    logic [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
```

```
/*************************************************************************
File name:      fa.sv
Circuit name:   Full Adder
Description:    structural description of a full adder
*************************************************************************/
module fa( input   logic in1, in2, cIn   ,
           output  logic out, cOut       );
    logic     xr  ;

    assign xr   = in1  ^ in2             ;
    assign out  = xr  ^ cIn              ;
    assign cOut = in1  & in2  | cIn  & xr ;
endmodule
```

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

◇

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.sv
Circuit name:    Adder
Description:     behavioral description of an adder
***********************************************************************/
 module add #(parameter n=8)(   input   logic [n-1:0] in1, in2,
                                input   logic         cIn     ,
                                output  logic [n-1:0] out     ,
                                output  logic         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 2.15). The half adder has two roles in the structure:

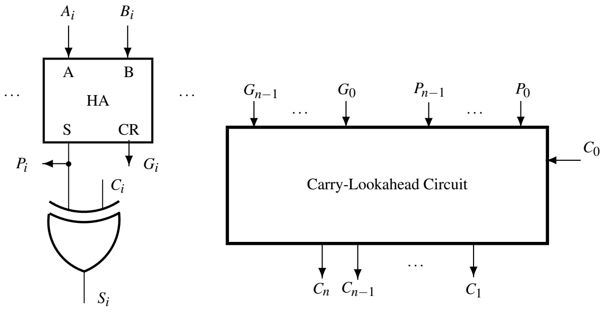


Figure 2.15: **The fast n-bit adder.** The n-bit Carry-Lookahead Adder (CLA_n) consists in n HAs, n 2-input XORs and the Carry-Lookahead Circuit used to compute faster the n C_i, for $i = 1, 2, \ldots n$.

- 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* $(i+1)$ *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.

2.1.6 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 2.4 *The 8-function ALU working on 32-bit numbers is described by the following Verilog module:*

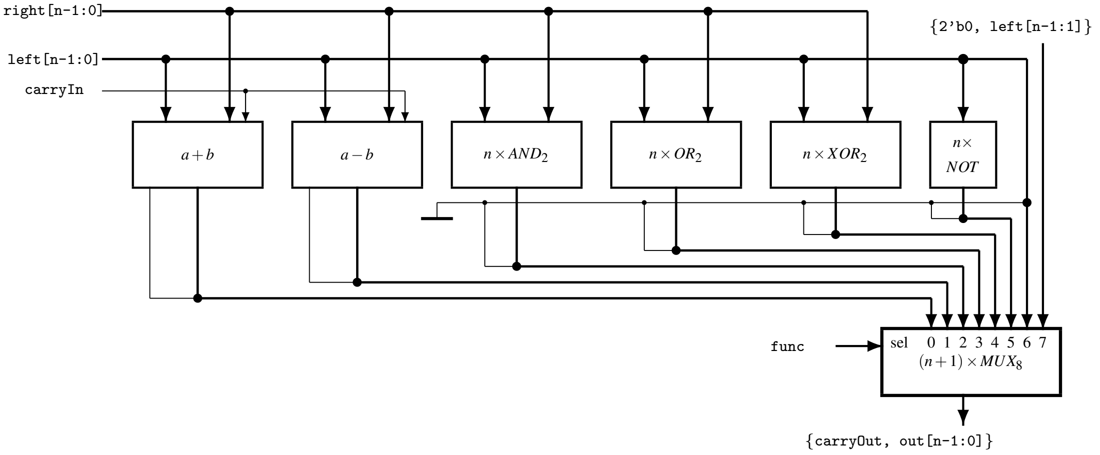


Figure 2.16: **The internal structure of the speculative version of an arithmetic and logic unit.** Each function is performed by a specific circuit and the output multiplexer selects the desired result.

```
/*************************************************************************
File name:      alu.sv
Circuit name:   arithmetic and logic unit
Description:    the circuit selects, using the selection code ’func’, one
                of the 8 functions
*************************************************************************/
module ALU( input   logic           carryIn     ,
            input   logic   [2:0]   func        ,
            input   logic   [31:0]  left, right ,
            output  logic           carryOut    ,
            output  logic   [31:0]  out         );

  always_comb
    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 2.16) described by the *Verilog* module from Example 2.4, 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 2.5 *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:*

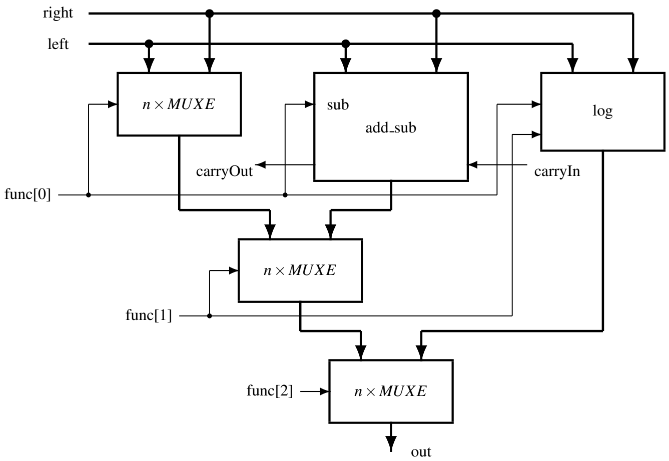


Figure 2.17: **The internal structure of an area optimized version of an ALU.** The `add_sub` module is smaller than an adder and a subtractor, but the operation "starts" only when `func[0]` is valid.

```
/*********************************************************************
File name:      structuralAlu.sv
Circuit name:   ALU
Description:    structural description for
*********************************************************************/
module structuralAlu( output  logic [31:0]  out     ,
                      output  logic         carryOut,
                      input   logic         carryIn ,
                      input   logic [31:0]  left    , right   ,
                      input   logic [2:0]   func    );
    logic [31:0]  shift, add_sub, arith, bool;

    addSub addSub(.out    (add_sub ),
                  .cout   (carryOut),
                  .left   (left    ),
                  .right  (right   ),
                  .cin    (carryIn ),
                  .sub    (func[0] ));
    log log(.out    (bool     ),
            .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(bool   ),
                   .sel(func[2]));
endmodule
```

```
/*********************************************************************
File name:      mux2.sv
Circuit name:
Description:
*********************************************************************/
 module mux2(input   logic         sel     ,
             input   logic [31:0]  in0, in1,
             output  logic [31:0]  out     );

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

```
/***********************************************************************
File name:      addSub.v
Circuit name:
Description:
***********************************************************************/
 module addSub(output logic [31:0]  out        ,
               output logic         cout       ,
               input  logic [31:0]  left, right ,
               input  logic         cin, sub   );

    assign {cout, out} = left + (right ^ {32{sub}}) + (cin ^ sub);
 endmodule
```

```
/***********************************************************************
File name: log.sv
Circuit name:
Description:
***********************************************************************/
module log( output logic [31:0] out         ,
            input  logic [31:0] left, right ,
            input  logic [1:0]  op          );
    integer i;
    logic [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  logic        o    ;
     input   logic        l, r ;
     input   logic [3:0]  f    ;

     o = f[{l,r}];
    endtask
endmodule
```

The resulting circuit is represented in Figure 2.17. 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].

2.2 The many-output random circuit: Read Only Memory

The simple solution for the following many-output random circuits having the same inputs:

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

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

$$\ldots$$

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

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 2.9 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 2.18a is adopted. The DCD circuit is shared for implementing the functions $f,\ g,\ldots s$. The shared DCD is used to compute all possible *minterms* (see *Appendix C.4*) needed to compute an n-variable Boolean function.

Figure 2.18b is an example of using the generic structure from Figure 2.18a 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 2.18c, 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.

If De Morgan transformation is applied, the circuit from Figure 2.18c is transformed in the circuit represented in Figure 2.19a, 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 2.20 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).

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.

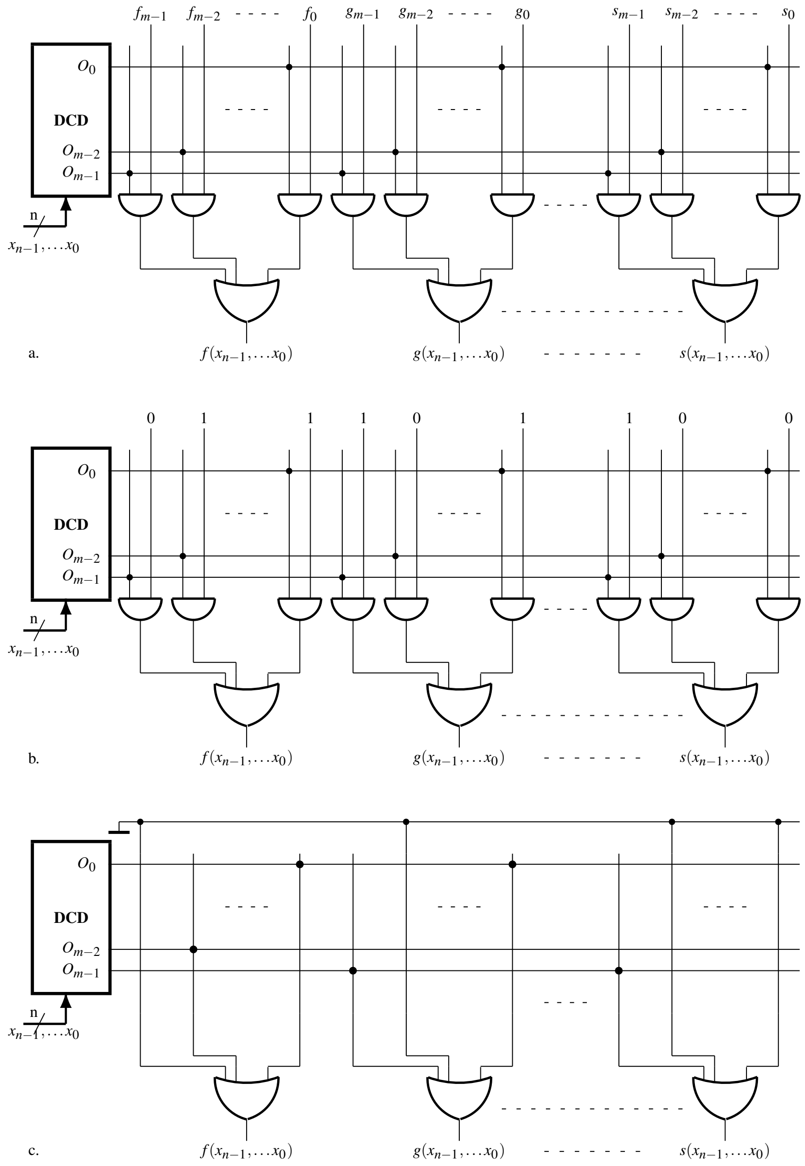


Figure 2.18: **Many-output random circuit. a.** One DCD and many AND-OR circuits. **b.** An example. **c.** The version using programmable connections.

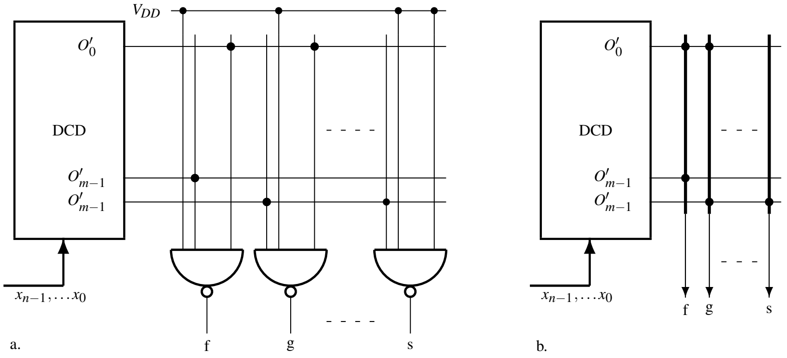


Figure 2.19: **The internal structure of a Read Only Memory used as trans-coder. a.** The internal structure. **b.** The simplified logic symbol where a thick vertical line is used to represent an m-input NAND gate.

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

Figure 2.20: **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.

Example 2.6 *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 2.22).*

The System Verilog code describing the circuit is:

```
/*************************************************************************
File name:      even_segments.sv
Circuit name:   Seven-Segment Transcoder
Description:    behavioral description of the seven-segment transcoder
*************************************************************************/
 module seven_segments( output  logic  [6:0]   out ,
                        input   logic  [3:0]   in  );
    always_comb 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 2.21a. If inverted outputs are needed results the circuit from Figure 2.21b.

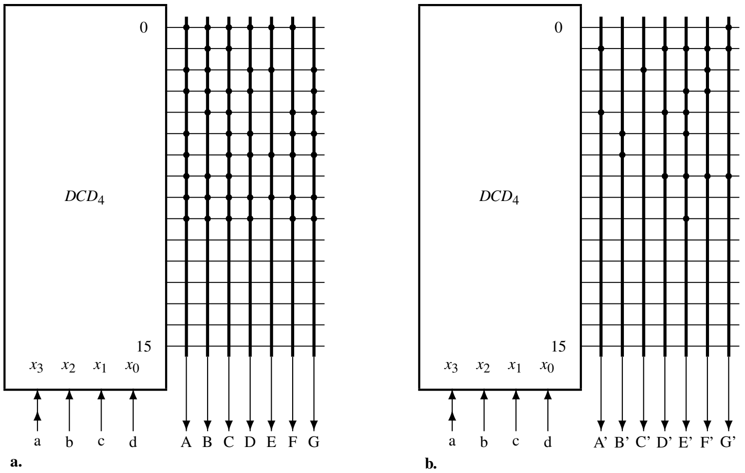


Figure 2.21: **The CLC as trans-coder designed serially connecting a DCD with an encoder.** Example: BCD to 7-segment trans-coder. **a.** The solution for non-inverting functions. **b.** The solution for inverting functions.

◇

2.3 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

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 2.22: **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".

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.

2.4 Problems

Problem 2.1 *Let be the circuits which receives two numbers:* `a[7:0]` *and* `b[7:0]`. *It has two outputs:*

- `outMin = mun(a,b)`
- `outMax = max(a,b)`

It is requested:

- *block schematic of the circuit*
- *description in System Veriilog*
- *simulation*

Problem 2.2 *Design using a behavioral description and simulate the combinational circuit which receives two 16-bit positive integers and generates the modulus of the difference of their squares. Draw the block schematic of the circuit.*

Problem 2.3 *It is required:*

1. *to draw the structure of a subtraction circuit for 8-bit numbers*
2. *implementation in System Verilog*
3. *simulation.*

Problem 2.4 *Design a circuit that receives as input,* `in`*, only numbers strictly between 0 and 64 and generates as output 1 if* $12 < in < 20$*, and otherwise generates 0.*

Problem 2.5 *Implement two functions of 4 binary variables with a 4-bit word DCD:*

1. *F1: returns 1 if most of the inputs are 1*
2. *F2: returns 1 if 2 inputs are 1.*

Problem 2.6 *Design an increment/decrement circuit for 8-bit numbers. The control bit* `inc=0` *determines decrement, and* `inc = 1` *determines increment.*

Problem 2.7 *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.*

Problem 2.8 *A barrel shifter for* 16*-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  logic [15:0] out     ,
                       input   logic [15:0] in      ,
                       input   logic [3:0]  shift   );
   ...
endmodule
```

Write a behavioral code and a minimal structural version in Verilog.

Problem 2.9 *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.*

g

Chapter 3

MEMORIES:
First order, 1-loop digital systems

> *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:

[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*.

- 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.

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).

3.1 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 3.1a).

The unstable loop is closed connecting the output `y0` of the elementary decoder to its input `x0` (see Figure 3.1b). 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 $f_{osc} = 1/(t_{pLH} + t_{pHL})$.

The stable loop is obtained connecting the output `y1` of the elementary decoder to the input `x0` (see Figure 3.1c). 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.

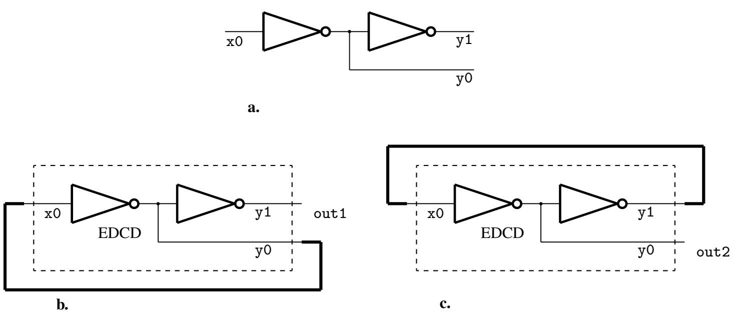


Figure 3.1: **The two loops closed over an elementary decoder. a.** The simplest combinational circuit: the one-input, elementary decoder. **b.** The unstable, inverting loop containing one (odd) inverting logic level(s). **c.** The stable, non-inverting loop containing two (even) inverting levels.

What is the main structural distinction between the two loops?

- The unstable loop has an *odd number of inverting levels*, thus the signal comes back to the output having the complementary value.
- The stable loop has an *even number of inverting levels*, thus the signal comes back to the output having the same value.

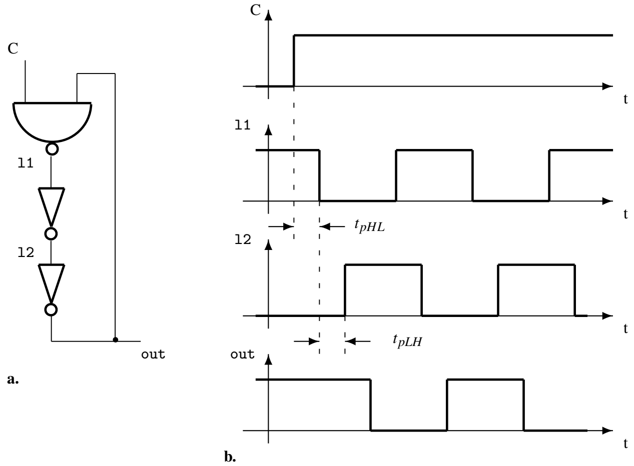


Figure 3.2: **The unstable loop.** The circuit version used for a low-cost and low-performance clock generator. **a.** The circuit with a three (odd) inverting circuits loop coupled. **b.** The wave forms drawn takeing into account the propagation times associated to the low-high transitions (t_{pLH}) and to the high-low transitions (t_{pHL}).

Example 3.1 *Let be the circuit from Figure 3.2a, 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 3.2b. 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.)*◇

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.

3.2 Elementary Structures

3.2.1 Elementary Latches

This chapter is devoted to introduce the elementary structure used to build memory systems: flip-flops, registers and random access memories. In order to be stable, all these elementary circuits have one loops with even (zero or two) inverting levels.

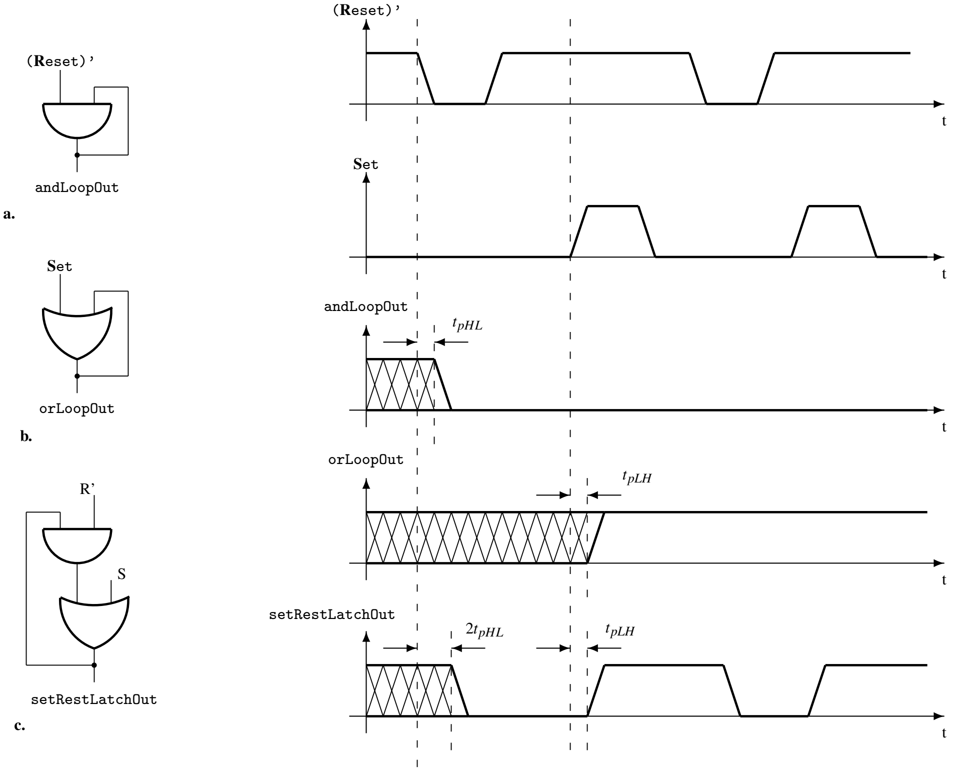


Figure 3.3: **The elementary latches.** Using the stable non-inverting loop (even inverting levels) elementary storage elements are built. **a**. *AND loop* provides a *reset-only* latch. **b**. *OR loop* provides the *set-only* version of a storage element. **c**. The heterogeneous elementary *set-reset* latch results combining the *reset-only* latch with the *set-only* latch.

The *reset-only latch* is the *AND loop* circuit represented in Figure 3.3a. The *passive* input value for*AND loop* is 1 (`(`**R**`eset)' = 1`), while the *active* input value is 0 (`(`**R**`eset)' = 0`). If the passive input value is applied, then the output of the circuits is not affected (the output depends only by the other input of the AND circuit). It can be 0 or 1, depending by the previous values applied on the input. When the active value is temporary applied, then the state of the circuit (the value of its output) switches in 0 and remains forever in this state, independent on 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 *set-only latch* is the *OR loop* circuit represented in Figure 3.3b. The *passive* value for *OR loop* is 0 (**S**`et = 0`) while the *active* input value is 1 (**S**`et = 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.

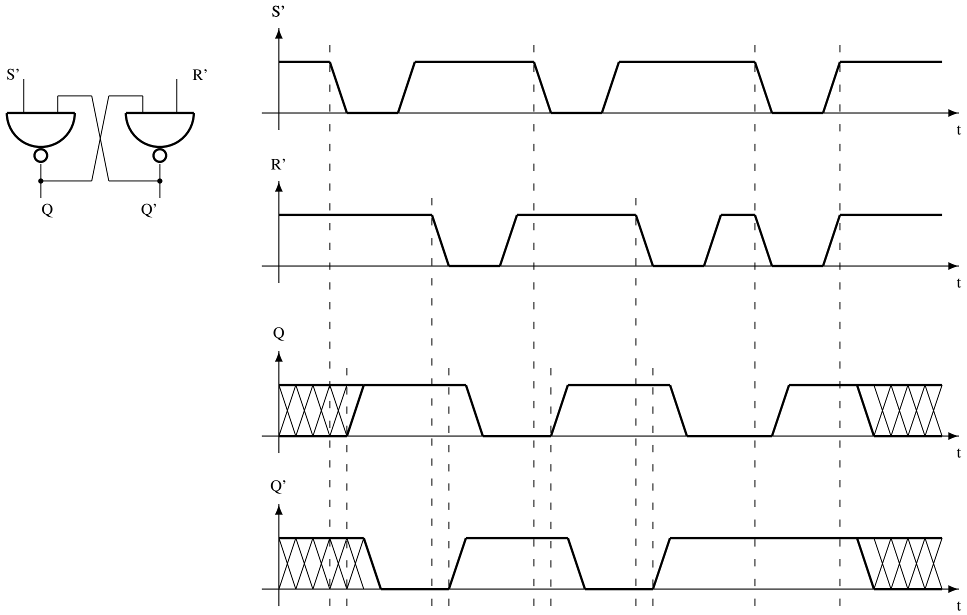


Figure 3.4: **Symmetric NAND elementary latches.**

The heterogenous *set-reset latch* results by combining the previous two latches (see Figure 3.3c). The circuit has zero inverting levels on the loop and 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.

The symmetric set-reset NAND latch is 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 resulting the circuit from Figure 3.4. The passive input value for the NAND elementary latch is 1. The active input value for the NAND elementary latch is 0. The symmetric structure of these latches have two outputs, Q and Q'.

The symmetric set-reset NOR latch is obtained by applying De Morgan's law to the heterogenous elementary latch. The second version of the symmetric version (see Figure 3.5) is obtained applying the de Morgan law to the AND circuit. The passive input value for the OR elementary latch is 0. The active input value for the OR elementary latch is 1. The symmetric structure have also two outputs, Q and Q'.

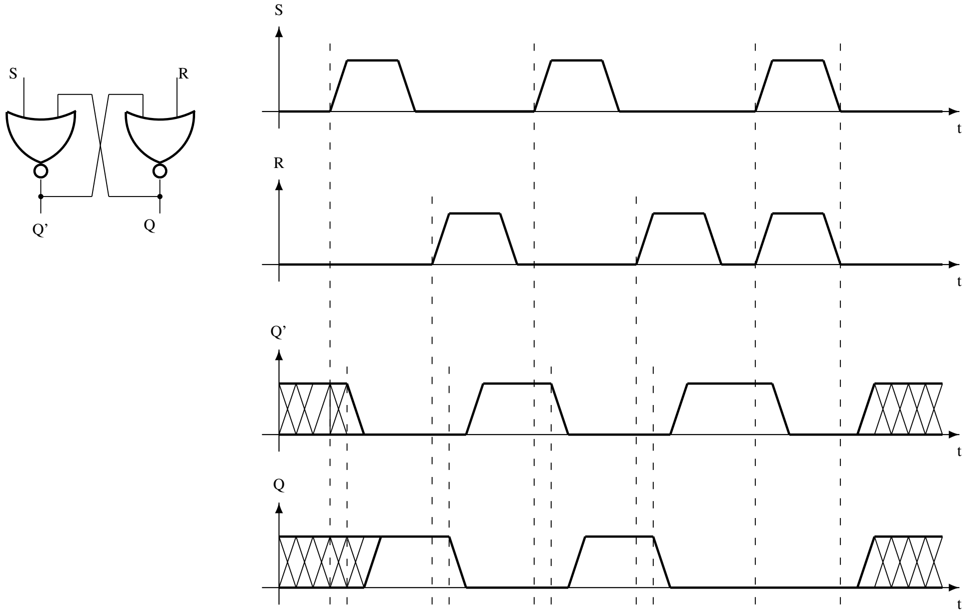


Figure 3.5: **Symmetric NOR elementary latches.**

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 (see Figure 3.4). or S=1 and R=1 on the inputs of OR latch (see Figure 3.5), 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 paragraph 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 the next chapter 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. And, as we already know, more autonomy means at least a new loop.

VeriSim 3.1 *The Verilog description of NAND latch is:*

```
module elementary_latch( output logic out, not_out,
                         input  logic not_set, not_reset);

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

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

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

Application: debouncing 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 3.6).

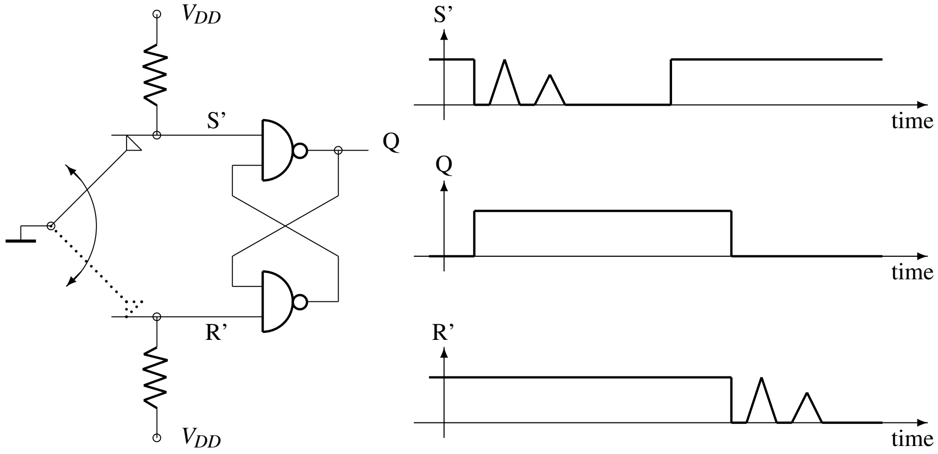


Figure 3.6: **The debouncing circuit.**

The debouncing circuit provide clean transitions when digital signals must generated by electro-mechanical switches. In Figure 3.6 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.

3.2.2 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.

The NAND latch is used to exemplify (see Figure 3.7a) 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.

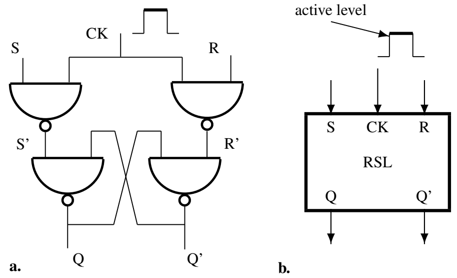


Figure 3.7: **Elementary clocked latch.** The transparent RS clocked latch is sensitive (transparent) to the input signals during the active level of the clock (the high level in this example). **a**. The internal structure. **b**. The logic symbol.

VeriSim 3.2 *The following System Verilog code can be used to understand how the elementary clocked latch works.*

```
module clocked_nand_latch( output   logic out, not_out,
                           input    logic 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
```

◇

3.2.3 Data Latch

In the first order circuits class the second latch problem can be only avoided, **not removed**, defining a restriction on the input of the clocked latch. Indeed, introducing an inverter between the inputs of the RS clocked latch, as is shown in Figure 3.8a, the ambiguous command (simultaneous set and reset) can not be applied. We name the new input D (from **D***ata*). Now, the situation $R = S = 1$ is avoided. The output is synchronized with the clock only if on the active level of CK the input D is stable.

The output of this new circuit, called **D latch**, follows all the time the input D. 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.

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

    always_comb if (clock) out = data;
```

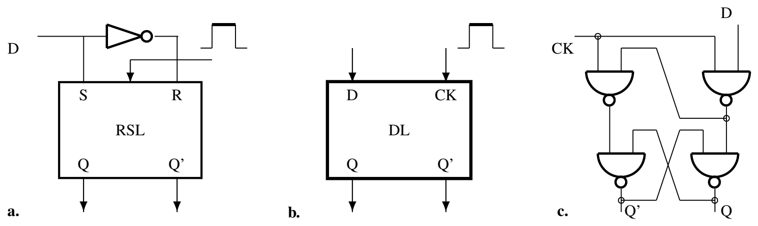


Figure 3.8: **The data latch.** Imposing the restriction $R = S'$ to an RS latch results the **D latch** without non-predictable transitions ($R = S = 1$ is not anymore possible). **a.** The structure. **b.** The logic symbol. **c.** An improved version for the data latch internal structure.

```
    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 3.8c an improved version of the circuit is presented. The number of components are minimized, the maximum depth of the circuit is maintained and the *fan-in* for the input D is reduced from 2 to 1.

VeriSim 3.3 *The following Verilog code can be used to understand how a D latch works.*

```
module test_data_latch;
    logic 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   logic out, not_out,
                   input    logic 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.*

◇

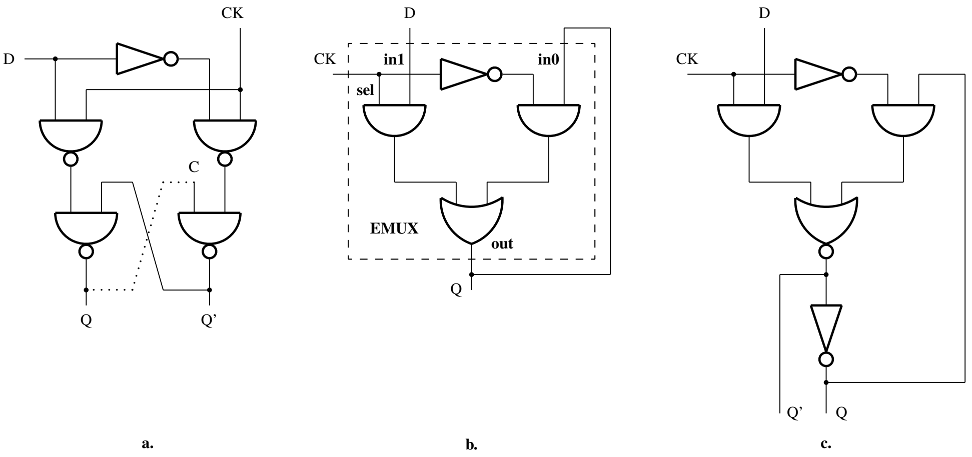


Figure 3.9: **The optimized data latch.** An optimized version is implemented closing the loop over an *elementary multiplexer*, EMUX. **a.** The resulting minimized structure for the circuit represented in Figure 3.8a. **b.** Implementing the minimized form using only inverting circuits.

The internal structure of the data latch (4 2-input NANDs and an inverter in Figure 3.8a) 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:

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

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

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

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

The resulting circuit is an *elementary multiplexor* (the selection input is CK and the selected inputs are D, by $CK = 1$, and C, by $CK = 0$. Closing back the loop, by connecting Q to C, results the circuit represented in Figure 3.9a. The actual circuit has also the inverted output Q' and is implemented using only inverted gates as in Figure 3.9b. The circuit from Figure 3.8a (using the RSL circuit from Figure 3.7a) is implemented with 18 transistors, instead of 12 transistors supposed by the minimized form Figure 3.9b.

VeriSim 3.4 *The following Verilog code can be used as one of the shortest description for a D latch represented in Figure 3.9a.*

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

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

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

◇

The *elementary decoder* was used to start the discussion about latches (see Figure 3.1). We ended using the *elementary multiplexer* to describe the most complex latch.

3.3 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.

3.3.1 The Master-Slave Principle

In order to remove the transparency of the clocked latches, disconnecting completely the *how* from the *when*, the *master-slave principle*[3] was introduced. This principle allows us to build a two state circuit

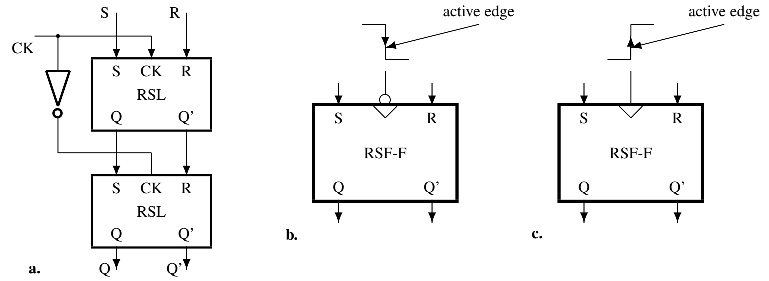


Figure 3.10: **The master-slave principle.** Serially connecting two RS latches, activated with different levels of the clock signal, results a non-transparent storage element. **a.** The structure of a RS master-slave flip-flop, active on the falling edge of the clock signal. **b.** The logic symbol of the RS flip-flop.

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 3.10). 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. We say the *RS master-slave flip-flop* switches with the falling (negative) edge of the clock. (The version triggered by the positive edge of clock is also possible.)

The switching moment of a master-slave structure is determined exclusively by the active edge of clock signal. Unlike the RS latch or data latch, which can sometimes be triggered (in the 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.

[3] The term ”Master-Slave” has been re-evaluated in recent years in the interests of political correctness. In light of the racial connotations associated with the term, many professionals question its necessity in engineering. Several companies have introduced their own approaches when reconsidering the terms ”master” and ”slave”. Alternative solutions [Charboneau ’20] are:

- Primary and secondary
- Primary and replica
- Primary and standby
- Leader and follower
- Conductor and follower
- Source and sink
- Main

We prefer Primary–Secondary.

VeriSim 3.5 *The following Verilog code can be used to understand how a master-slave flip-flop works.*

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

    logic     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 must catch our attention in designing digital systems with edge triggered flip-flops:

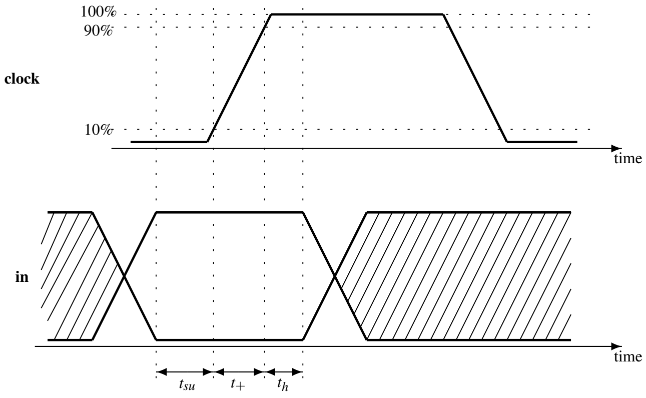


Figure 3.11: **Magnifying the transition of the active edge of the clock signal.** The input data must be stable around the active transition of the clock t_{su} (set-up time) before the beginning of the clock transition, during the transition of the clock, t_+ (active transition time), and t_h (hold time) after the end of the active edge.

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 (see Figure 3.11)

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

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

3.3.2 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:

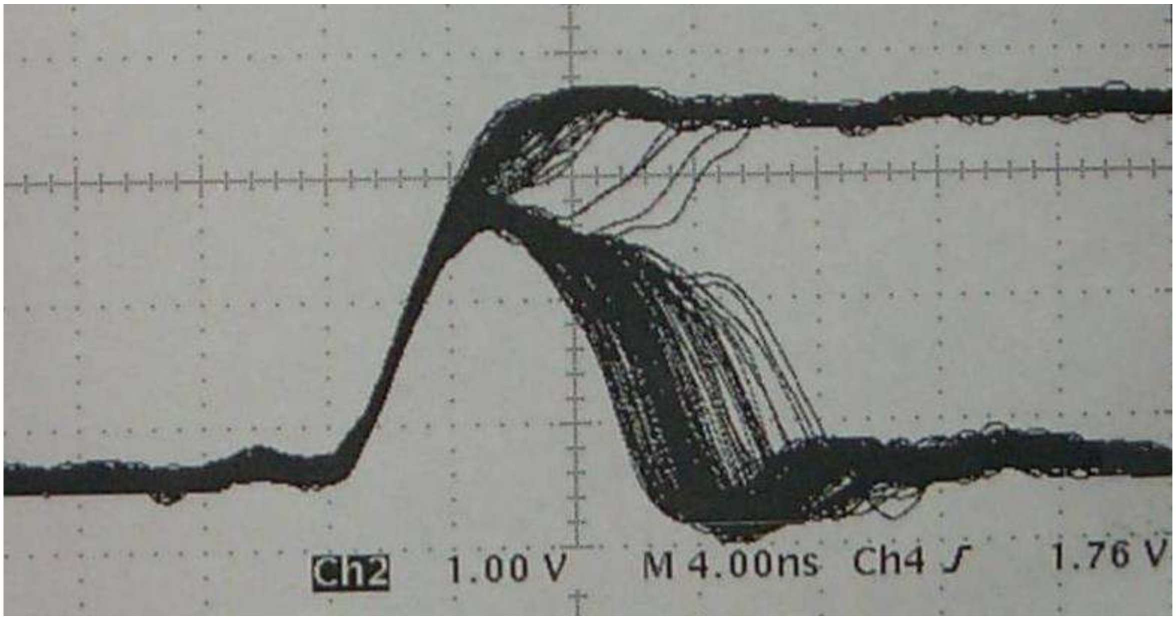


Figure 3.12: Metastability [webRef_4].

- 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]).

3.3.3 The 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 3.13a). 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 3.13c 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.

VeriSim 3.6 *The structural Verilog description of a D flip-flop, provided only for simulation purpose, follows.*

```
module dff( output   logic out, not_out,
            input    logic d, clock    );
    logic    not_d;

    not #2  data_inverter(not_d, d);
    master_slave    rs_ff(out, not_out, d, not_d, clock);
endmodule
```

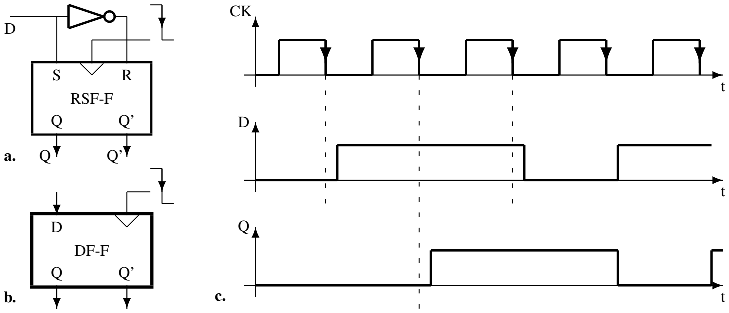


Figure 3.13: **The delay (D) flip-flop.** Restricting the two inputs of an RS flip-flop to $D = S = R'$, results an FF with predictable transitions. **a.** The structure. **b.** The logic symbol. **c.** The wave forms proving the delay effect of the D flip-flop.

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

```
module dff(output logic out     ,
           input  logic 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 asynchronous systems. Both are useful, but in the first kind of systems the complexity is easiest manageable.

3.3.4 The Serial Register

Starting from the delay function of the last presented circuit (see Figure 3.13) 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 3.1 *An n-bit serial register, SR_n, is made by serially connecting a D flip-flop with an SR_{n-1}. SR_1 is a D flip-flop.* ◇

In Figure 3.14 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".

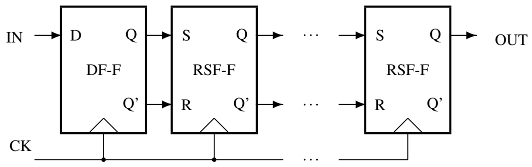


Figure 3.14: **The n-bit serial register (SR_n).** Triggered by the active edge of the clock, the content of each RSF-F is loaded with the content of the previous RSF-F.

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 3.7 *The functional description currently used for an n-bit serial register active on the positive edge of clock is:*

```
/*********************************************************************
File name:       serial_register.sv
Circuit name:    Serial register
Description:     behavioral description of a n-bit serial register
*********************************************************************/
module serial_register #(parameter n = 1024)
         (output logic out              ,
          input  logic in, enable, clock);
    logic [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
```

◇

3.4 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.

3.4.1 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 3.8 *A 16-bit latch is described in Verilog as follows:*

```
/* ************************************************************************
File name:      n_latch.sv
Circuit name:   n-Bit Latch
Description:    behavioral description of a n-bit latch
************************************************************************ */
module  n_latch #(parameter n = 16)(output  logic [n-1:0] out    ,
                                    input   logic [n-1:0] in     ,
                                    input   logic         clock  );

    always_comb 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.

3.4.2 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 3.2 *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 3.15. 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* $A_{n-1},\ldots,A_0$.[4] The wave forme describing the relation between the input and output signals of a RAM are represented in Figure 3.16, 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

[4] 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.)

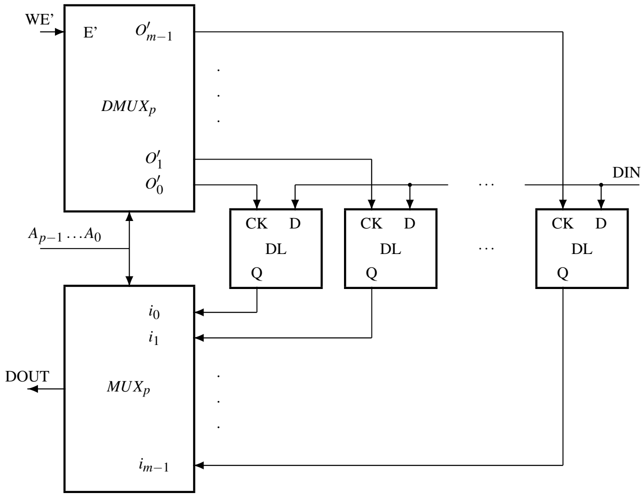


Figure 3.15: **The principle of the random access memory (RAM).** The clock is distributed by a DMUX to one of $m = 2^p$ DLs, and the data is selected by a MUX from one of the m DLs. Both, DMUX and MUX use as selection code a p-bit address. The one-bit data DIN can be stored in the clocked DL.

- 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 3.15 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 3.9 *The functional description of a asynchronous $n = 2^p$ m-bit words RAM follows:*

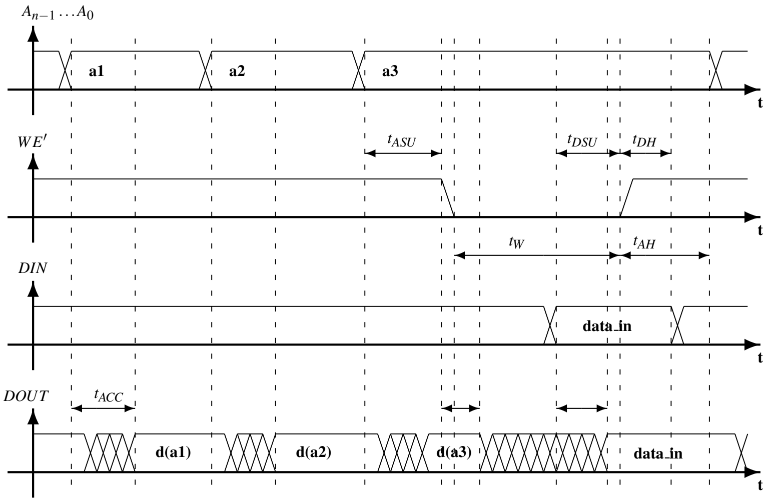


Figure 3.16: **Read and write cycles for an asynchronous RAM.** Reading is a combinational process of selecting. The access time, t_{ACC}, is given by the propagation through a big MUX. The write enable signal must be strictly included in the time interval when the address is stable (see t_{ASU} and t_{AH}). Data must be stable related to the positive transition of WE' (see t_{DSU} and t_{DH}).

```
/*************************************************************************
File name:      ram.sv
Circuit name:   Asynchronous RAM
Description:    behavioral descriiption of an asynchronous random-access
                memory
*************************************************************************/
 module ram(input   logic [m-1:0]   din ,   // data input
            input   logic [p-1:0]   addr,   // address
            input   logic           we  ,   // write enable
            output  logic [m-1:0]   dout);  // data out
    logic     [m-1:0]   mem[(1'b1<<p)-1:0]; // the memory

    assign  dout = mem[addr];               // reading
    always_comb 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 3.15 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 3.17 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 2.6 and 2.9, 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 3.17 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 3.18 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]`.

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.

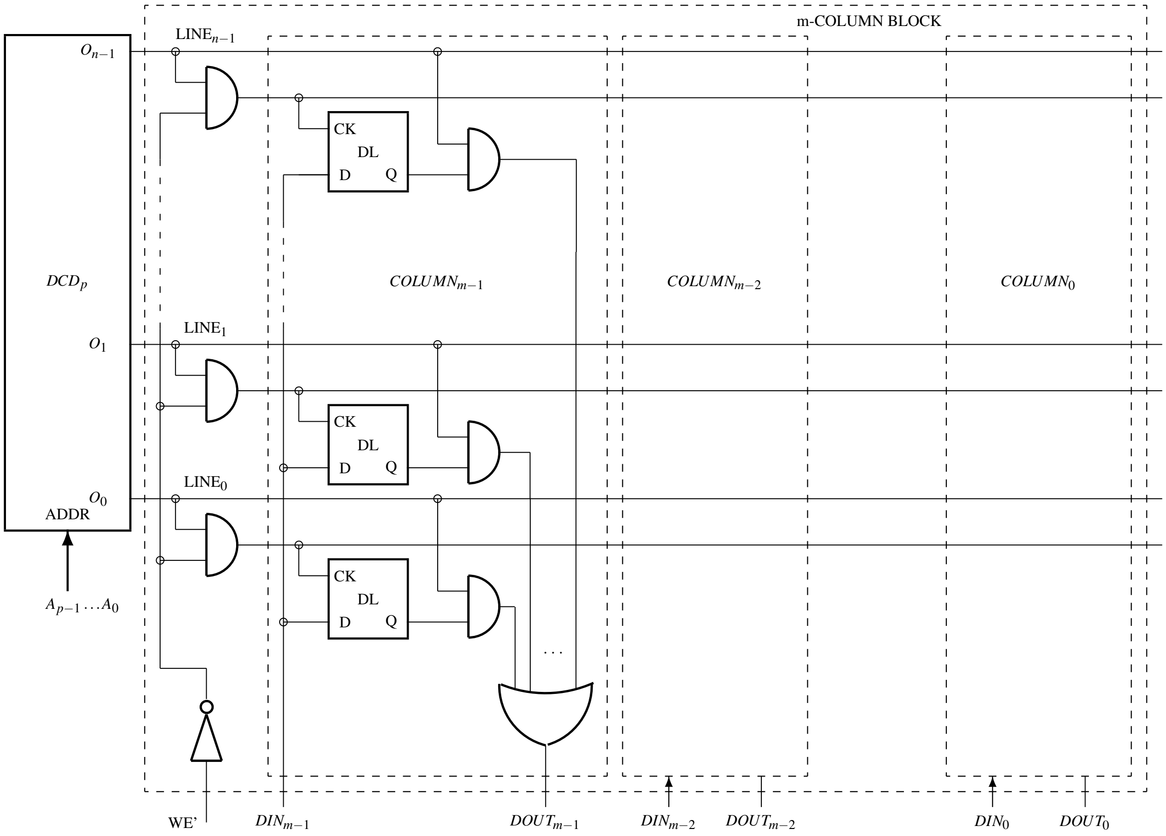


Figure 3.17: **The asynchronous *m*-bit word RAM.** Expanding the number of bits per word means to connect in parallel one-bit word memories which share the same decoder. Each COLUMN contains the storing latches and the AND-OR circuits for one bit.

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 3.18 – which distributes the write enable signals, we, to the selected m-column block. The same decoder is shared by the m s-input MUXs used to select the output word from the many-word selected by ROW DECODE.

The well known principle of ”divide et impera” (*divide and conquer*) is applied when the address is divided in two parts, one for selecting a row and another for selecting a column. The access circuits is thus minimized.

Unfortunately, RAM has not the *function of memorizing*. It is only a storage support. Indeed, if we want to *“memorize”* the number 13, for example, we must store it to the address 131313, for example, and to keep in mind (to memorize) the value 131313, the place where the number is stored. And than, what’s the help provided us by a the famous RAM memory? No one. Because RAM is not a memory, it becomes a memory only if the associated processor runs an appropriate procedure which allows us to forget about the address 131313. Another solution is provided by additional circuits used to improve the functionality (see the subsection about *Associative Memories*.)

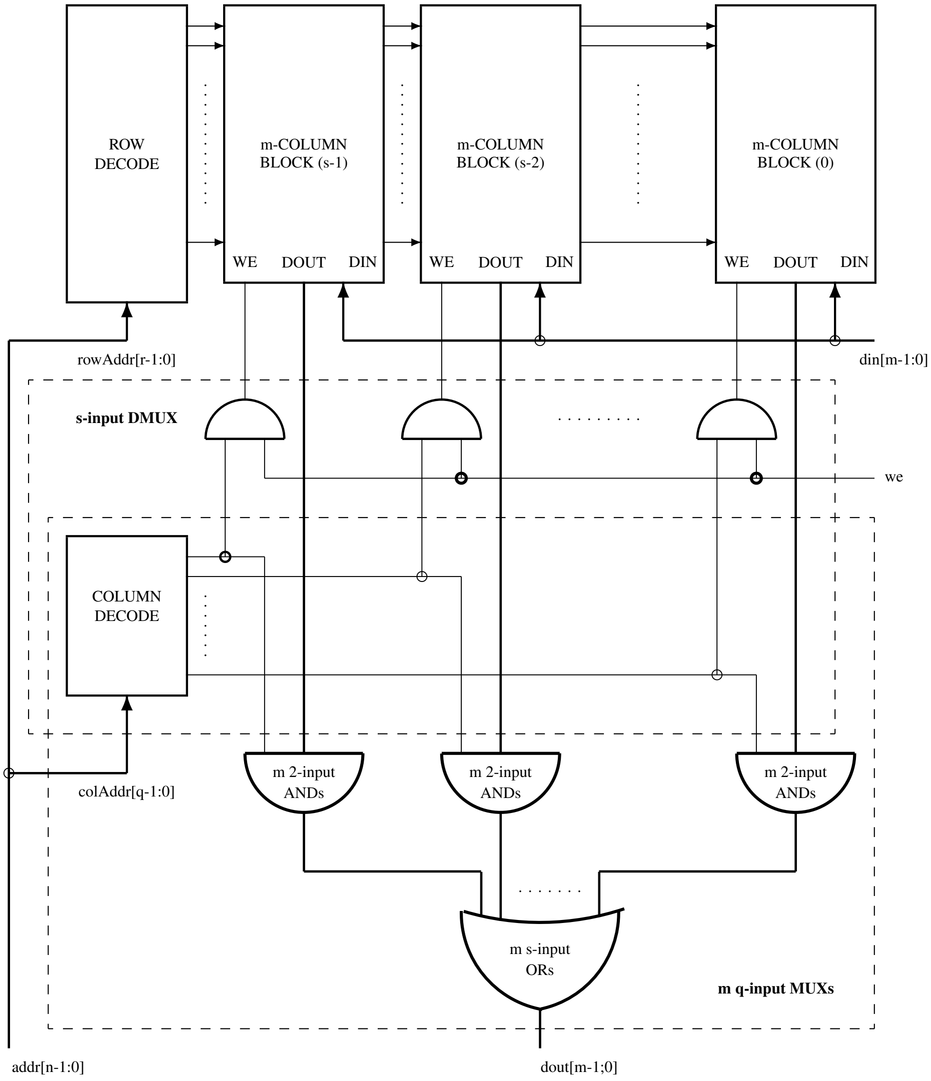


Figure 3.18: **RAM version with two dimension storage array.** A number of m-bit blocks are parallel connected and driven by the same row decoder. The column decoder selects to outoput an *m*-bit word from the $(s \times m)$-bit row.

3.5 The Serial-Parallel Composition: the Register

The last composition in 1-OS is the *serial-parallel composition*. The most representative circuit of this class is the *register*. The main application of register is to support the synchronous processes in a digital system. There are two typical use of the register:

- provides the *pipeline* connection between subsystems (see the subsections 2.5.1 *Pipelined connections*, and 3.3.2 *Pipeline structures*).
- stores the internal state of an automata (see the next chapter); the register is used to close of the second loop in a digital system.

Unlike the parallel compositions that *store asynchronously*, the circuits resulting from the serial-parallel compositions *store synchronously* the value applied on their inputs. The parallel compositions are used for designing *memory* systems, instead of the serial-parallel compositions, used to support the designing of the *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 (simple uniform or complex random) 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 unstable loops.

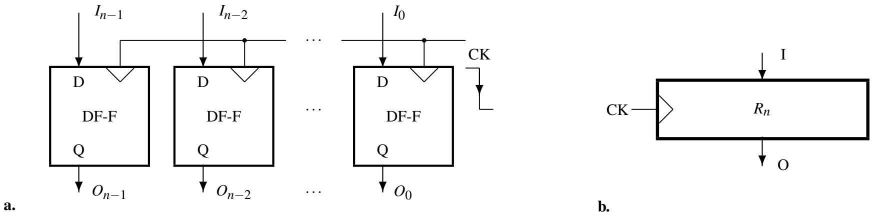


Figure 3.19: **The n-bit register. a.** The structure: a bunch of DF-F connected in parallel. **b.** The logic symbol.

Definition 3.3 *An n-bit register, R_n, is made by parallel connecting a R_{n-1} with a D (master-slave) flip-flop. R_1 is a D flip-flop.* ⋄

The register R_n, represented in Figure 3.19, is a *serial-parallel composition* in 1-OS because its elementary component, the D flip-flops, are serial compositions in 1-OS. Another possible definition is to build the register by serially connecting two n-bit latches. We know that the n-bit latch is a parallel extension in 1-OS. The clock must be applied to the two n-bit latches avoiding the simultaneous transparency.

VeriSim 3.10 *An* 8*-bit enabled and resetable register with 2 unit time delay is described by the following Verilog module:*

```
module register #(parameter n = 8)
            (output logic [n-1:0] out                ,
             input  logic [n-1:0] in                 ,
             input  logic         reset, enable, clock);

    always_ff  @(posedge clock) #2 if (reset)         out <= 0     ;
                                   else if (enable)   out <= in    ;
                                        else          out <= out   ;
endmodule
```

◇

The main feature of the register assures its non-transparency, excepting an ”undecided transparency” during a short time interval, $t_{SU} + t_H$, 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.

3.6 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*.

3.6.1 Synchronous RAM

It is very hard to consider the time restriction imposed by the wave forms presented in Figure 3.16 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 3.20).

VeriSim 3.11 *The functional description of a synchronous RAM (0.5K of 64-bit words) follows:*

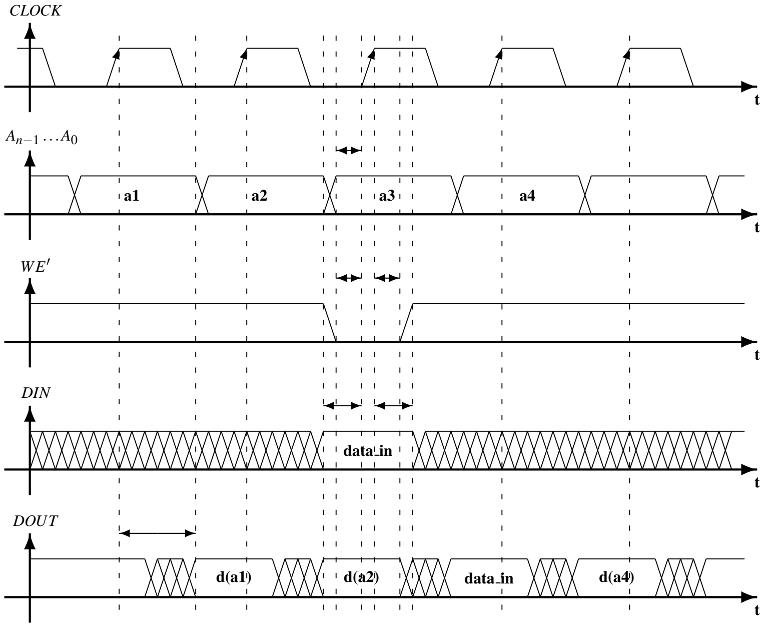


Figure 3.20: **Read and write cycles for SRAM.** For the *flow-through* version of a *SRAM* the time behavior is similar to a register. The set-up and hold time are defined related to the active edge of clock for all the input connections: *data*, *write-enable*, and *address*. The data output is also related to the same edge.

```
/* ************************************************************************
File name:         sram.sv
Circuit name:      Synchronous RAM
Description:       behavioral description of a synchronous RAM
************************************************************************ */
module sram(input   logic [63:0]  din ,
            input   logic [8:0]   addr,
            output  logic [63:0]  dout,
            input   logic         we, clk);
    logic [63:0]  mem[511:0];

    always_ff @(posedge clk) if (we) dout <= din          ;
                             else    dout <= mem[addr]    ; // reading
    always_ff @(posedge clk) if (we) mem[addr] <= din     ; // writing
endmodule
```

◇

The previously described SRAM is the *flow-through* version of a SRAM. A pipelined version is also possible. It introduces another clock cycle delay for the output data.

3.6.2 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 3.21).

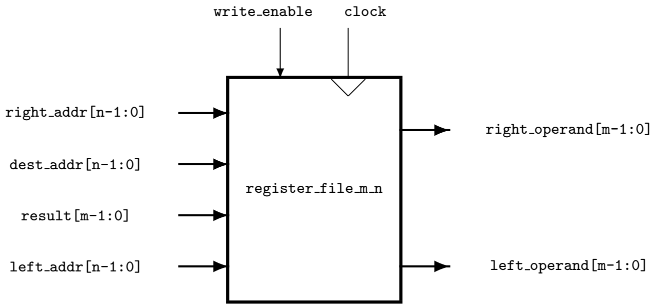


Figure 3.21: **Register file.** In this example it contains 2^n m-bit registers. In each clock cycle any two registers can be read and writing can be performed in anyone.

VeriSim 3.12 *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.*

```
/***********************************************************************
File name:        register_file.sv
Circuit name:
Description:
***********************************************************************/
module register_file(   output  logic [31:0]    left_operand    ,
                        output  logic [31:0]    right_operand   ,
                        input   logic           result          ,
                        input   logic [4:0]     left_addr       ,
                        input   logic [4:0]     right_addr      ,
                        input   logic 4:0]      dest_addr       ,
                        input   logic           write_enable    ,
                        input   logic           clock           );
    logic   [31:0]   file[0:31];

    assign  left_operand    = file[left_addr]   ,
            right_operand   = file[right_addr]  ;
    always_ff @(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.

3.7 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.

3.8 Problems

Problem 3.1 *The block diagram, System Verilog description and simulation of a 2048-word 16-bit RAM memory are required. The contents of the first 8 locations will be initialized with arbitrary values to be used for testing.*

Problem 3.2 *Design a system that receives a 16-bit number in each clock cycle and outputs the arithmetic average of the last 4 numbers received.*

Problem 3.3 *Add to a register 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 3.4 *Design a serial-parallel register which shifts 16 16-bit numbers.*

Definition 3.4 *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.* $\diamond$

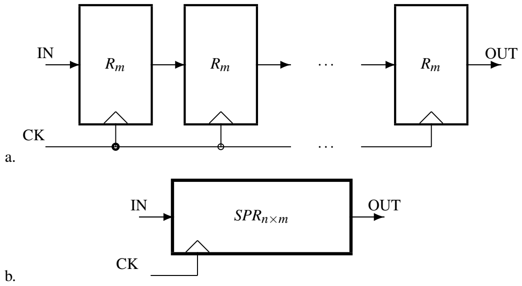


Figure 3.22: **The serial-parallel register. a.** The structure. **b.** The logic symbol.

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 3.22 is shown the serial-parallel $SPR_{n\times m}$.*

Problem 3.5 *Draw* `register_file_16_4` *at the level of registers, multiplexors and decoders.*

Chapter 4

AUTOMATA: Second order, 2-loop digital systems

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.

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:

- *asynchronous automata*, for which the loop is closed over **unclocked latches**, through combinational circuit and/or **unclocked** latches as in Figure 4.1a

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

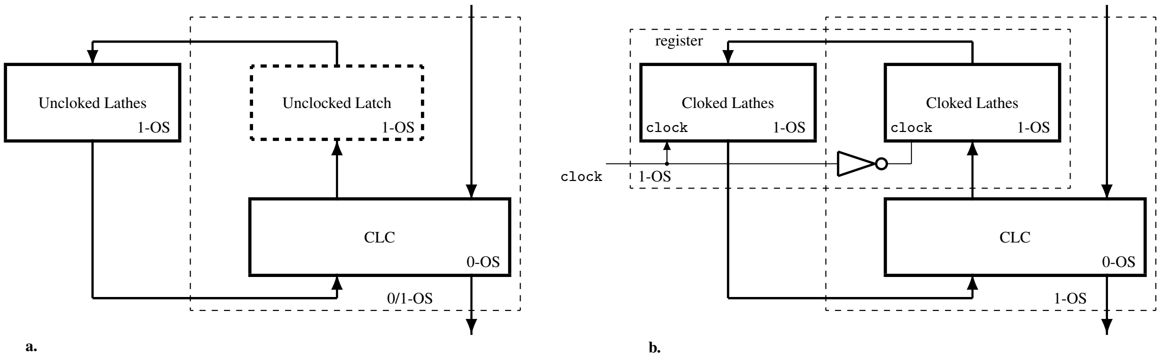


Figure 4.1: **The two type of 2-OS. a.** The asynchronous automata with a hazardous loop over a transparent latch. **b.** The synchronous automata with a edge clock controlled loop closed over a non-transparent register.

- *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 4.1b).

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.

4.1 Basic definitions in automata theory

Definition 4.1 *An automaton, A, is defined by the following 5-uple:*

$$A = (X, Y, Q, f, g)$$

where:

X *: the finite set of input variables*

Y *: the finite set of output variables*

Q *: the set of state variables*

f *: the state transition function, described by* $f : X \times Q \rightarrow Q$

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

- $g : X \times Q \rightarrow Y$ *for Mealy type automaton*
- $g : Q \rightarrow Y$ *for Moore type automaton*
- $g(q) = q$ *for* $Y \equiv Q$*, where* $q \in Q$ *for half-automaton, symbolized with* $A_{1/2}$.

At each clock cycle the state of the automaton switches and the output takes the value according to the new state (and the current input, in Mealy's approach). $\diamond$

Definition 4.2 *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 4.3 *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 4.4 *An* **initial state** *is a state having no predecessor state.* $\diamond$

Definition 4.5 *An* **initial automaton** *is an automaton having a set of initial states, Q', which is a subset of Q, $Q' \subset Q$.* $\diamond$

Definition 4.6 *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 4.7 *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 4.2a,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 4.2 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 4.2c) 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 4.2d) 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 4.2e); this automaton reacts also to the input signal in the next clock cycle, but the output is hazard free because it is registered

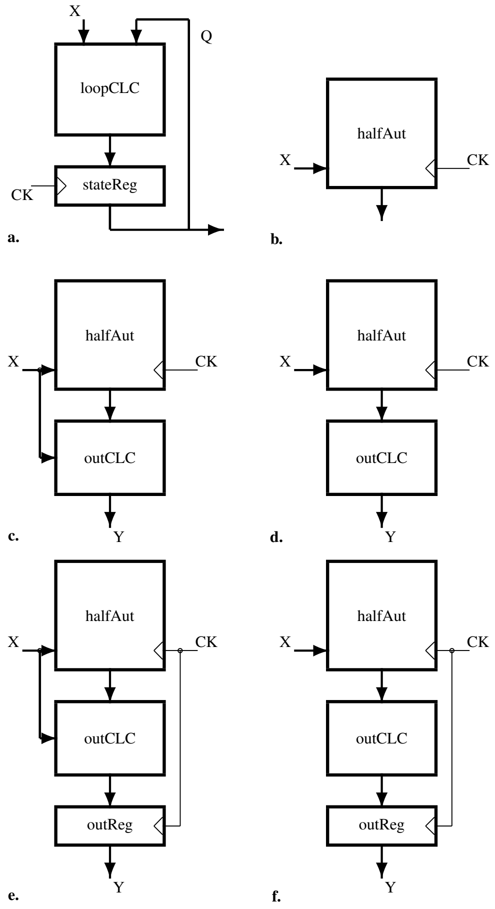


Figure 4.2: **Automata types. a.** The structure of the half-automaton ($A_{1/2}$), the no-output automaton: the state is generated by the previous state and the previous input. **b.** The logic symbol of half-automaton. **c.** Immediate Mealy automaton: the output is generated by the current state and 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.

delayed Moore automaton : results serially connecting a register, R, to the output of the Moore automaton (Figure 4.2f); 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 4.1 *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:*

- $y(t) = g(x(t-1), q(t-1))$ *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. ⋄

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.

4.2 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).

4.2.1 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 (+).

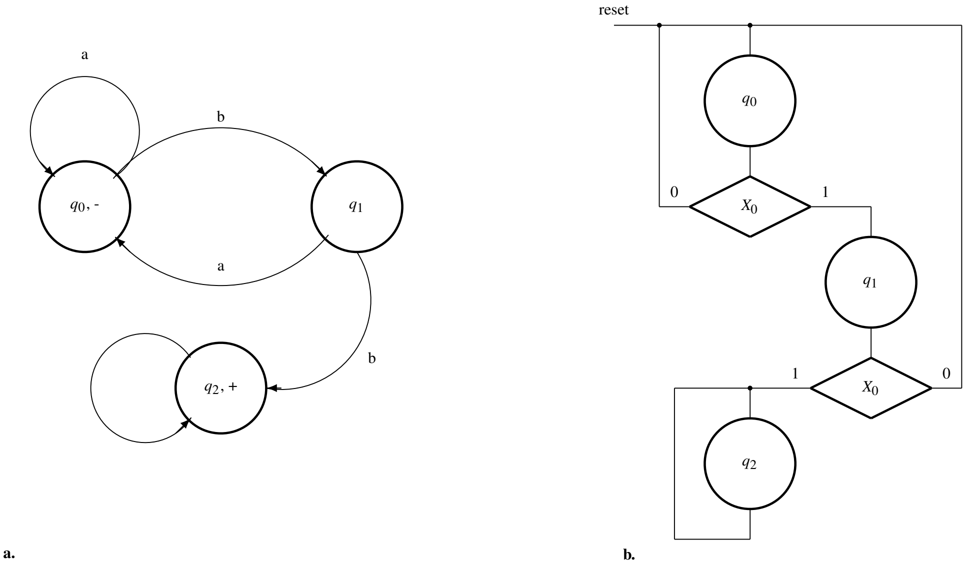


Figure 4.3: **Example of flow-chart for a half-automaton.** The machine is a "double *b* detector". It stops when the first *bb* occurs.

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 4.1 *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 4.3a. 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 4.3b). The input I takes the binary value 0 for the the symbol a and the binary value 1 for the symbol b. ⋄

The second version is used mainly when a circuit implementation is envisaged.

The flow-chart for a Moore automaton When an automaton is represented the output behavior must be also included.

The first, pure symbolic version contains in each circle besides, the name of the 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.

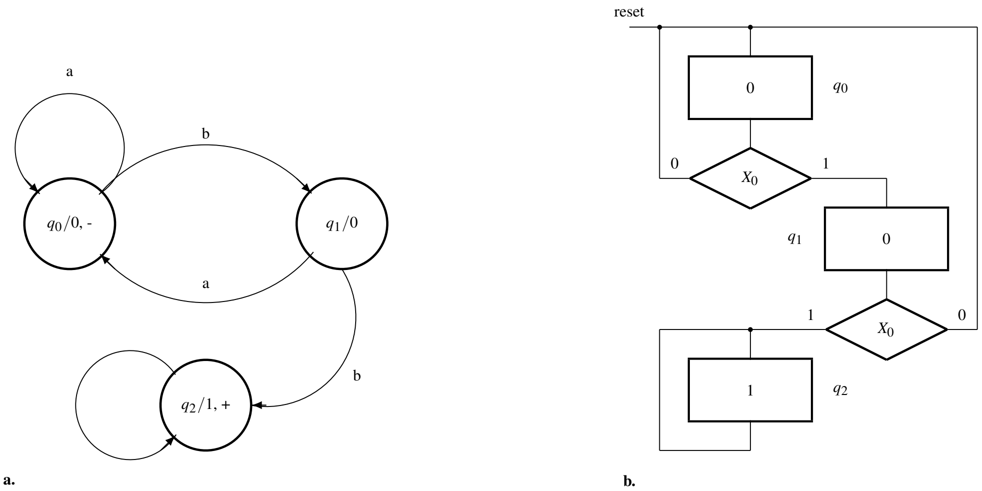


Figure 4.4: **Example of flow-chart for a Moore automaton.** The output of this automaton tells us: "*bb* was already detected".

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 4.2 *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 4.4a. 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 4.4b) uses three rectangles, one for each state. ⋄

A meaningful event on the input of a Moore automaton is shown on the output with a delay of a clock cycle. All goes through the state set. In the previous example, if the second *b* from *bb* is applied on the input in the period T_i of the clock cycle, then the automaton points out the event in the period T_{i+1} of the clock cycle.

The flow-chart for a Mealy automaton The first, pure symbolic version contains on each arrow besides, the name of the condition, the value of the output generated in the state where the arrow starts with the input specified on the arrow.

The Mealy automaton reacts on its outputs more promptly to a meaningful input event. The output value depends on the input value from the same clock cycle.

The second, implementation oriented version uses rectangles to specify the output's behavior.

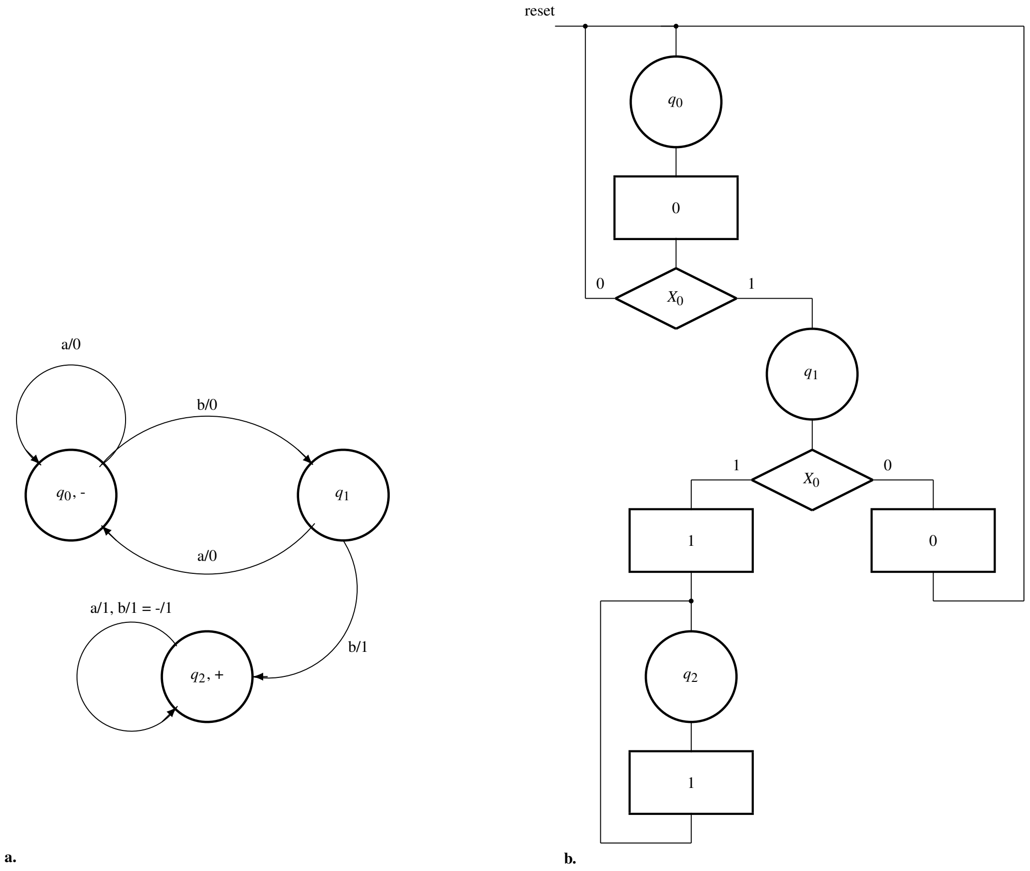


Figure 4.5: **Example of flow-chart for a Mealy automaton.** The occurrence of the second *b* from *bb* is detected as fast as possible.

Example 4.3 *Let us solve again the same problem of bb detection using a Mealy automaton. The resulting flow-chart is in Figure 4.5a. 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 4.5b represents the second representation. ⋄

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

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: $a \rightarrow X_0 = 0$ and $b \rightarrow X_0 = 1$.

Because the actual value of the state is "hidden" from the user, the designer has the freedom to assign the binary values according to its own (engineering) criteria. Because the present approach is a theoretical one, we do not have engineering criteria. Therefore, we are completely free to assign the binary codes. Two option are presented:

option 1: $q_0 = 00$, $q_1 = 01$, $q_2 = 10$

option 2: $q_0 = 00$, $q_1 = 10$, $q_2 = 11$

For both the external behavior of the automaton must be the same.

Transition diagrams for half-automata The transition diagram maps the reference VKD into the next state VKD, thus defining the state transition function. Results a representation ready to be used to design and to optimize the physical structure of a finite half-automaton.

Example 4.4 *The flow-chart from Figure 4.3 has two different correspondent representations as transition diagrams in Figure 4.6, one for the option 1 of coding (Figure 4.6a), and another for the option 2 (Figure 4.6b).*

In VKD S_1, S_0 each box contains a 2-bit code. Three of them are used to code the states, and one will be ignored. VKD S_1^+, S_0^+ represents the transition from the corresponding states. Thus, for the first coding option:

- *from the state codded 00 the automaton switch in the state $0x$, that is to say:*
    - *if $X_0 = 0$ then the next state is 00 (q_0)*
    - *if $X_0 = 1$ then the next state is 01 (q_1)*

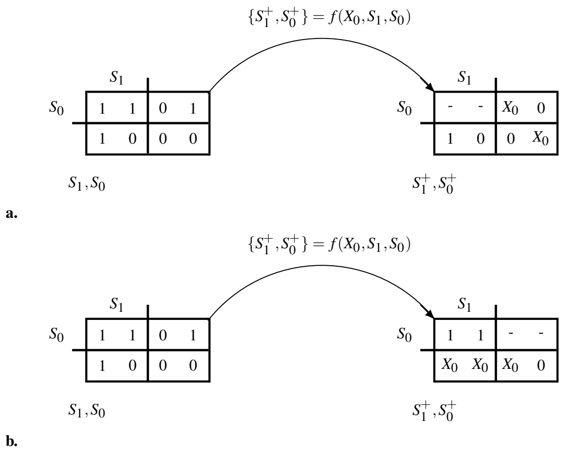


Figure 4.6: **Example of transition diagram for a half-automaton.** **a.** For the option 1 of coding. **b.** For the option 2 of coding.

- *from the state codded 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* $X_0 X_0$*, 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 4.5 *The Boolean function of the half-automaton working as "double b detector" can be extracted from the transition diagram represented in Figure 4.6a (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 4.6 *The flow-chart from Figure 4.4 have a correspondent representation in the transition diagrams from Figure 4.7a or Figure 4.7b. 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).*

◇

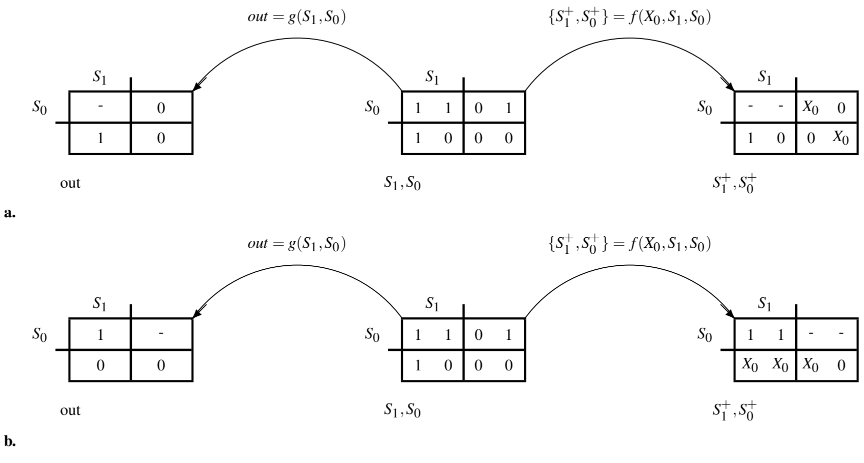


Figure 4.7: **Example of transition diagram for a Moore automaton.**

Example 4.7 *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 4.8, 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.* ⋄

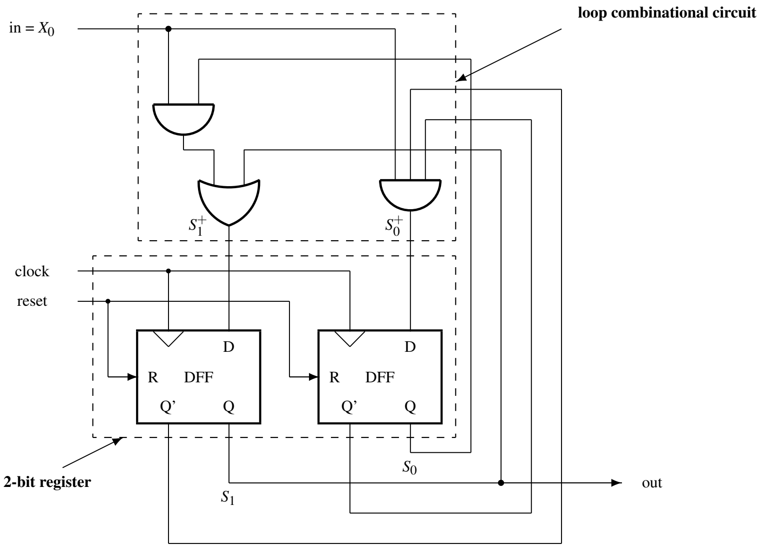


Figure 4.8: **The Moore version of "bb detector" automaton.**

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 4.8 *Revisiting the same problem result, in Figure 4.9 the transition diagrams associated to the flow-chart from Figure 4.5.*

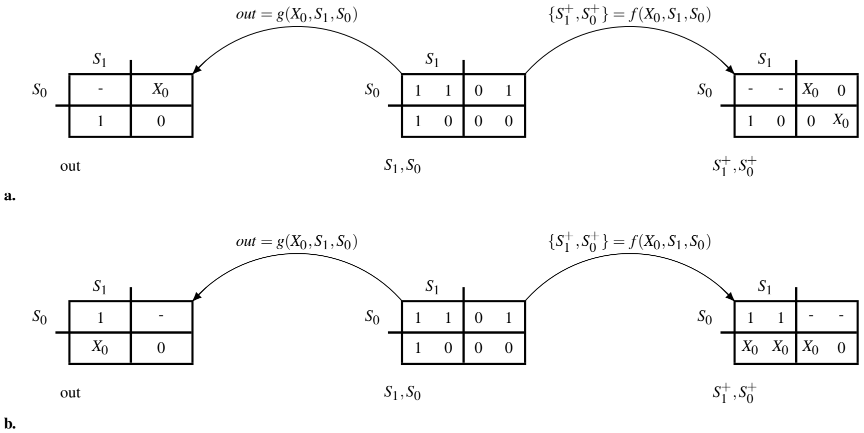


Figure 4.9: **Example of transition diagram for a Mealy automaton.**

The two functions f are the same. The function g is defined for the first coding option (Figure 4.9a) 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:*

$$out = S_1 + X_0S_0$$

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

In the description and synthesis of finite automata, we will directly use the representations in HDL (in our case Verilog), avoiding the descriptions that use graphs, charts or Veitch-Karnaugh diagrams. We will start from the symbolic description of the sets X, Y, Q converted into a binary form (see `defines.hv`), and we will continue with the description of the associated semiautomaton (see `halfAutomaton.v`) followed by the description of the output function in the 4 possible forms given by the Mealy-Moore and immediate-delayed distinctions.

Defines The problem solved by the finite automaton used as example is the detection of the sequence `bb` in the stream of symbols belonging to the set {`a`,`b`}.

The file describing the variables is the following:

```
/* *********************************************************
File name:      defines.vh
Circuit name:
Description:
********************************************************* */
// input codes
    ‘define a (1’b0)
    ‘define b (1’b1)
// internal state
    ‘define init_state  (2’b00) // initial state
    ‘define one_b_state (2’b01) // one b received
    ‘define final_state (2’b10) // final state
// output codes
    ‘define no  (1’b0) // no bb yet received
    ‘define yes (1’b1) // bb have been received
```

The binary codes associated to the input and output set are defined by the used of the design, while the binary codes associated to the internal states of the automaton are defined by the designer. It is very important that the designer has the freedom to associate the binary code according to its criteria. Designer criteria take into account, as we will see in the **??** section, optimization or even physical realizability criteria.

Half-Automaton Solving the problem of detecting the sequence `bb` is done at the level of the semiautomaton and is independent of the way the automaton reports the result on the outputs. For this reason, the semiautomaton can receive a separate description, which will be used in the final form of the design by inserting it into one of the 4 forms that the automaton can take depending on the user’s requirements.

```
/* *********************************************************
File name:      halfAutomaton.sv
Circuit name:   HA for double b detector
Description:    behavioral description of the half-automaton
                designed to detect ’bb’ in a stream
                of symbols belonging to the set {a,b}
********************************************************* */
‘include ”defines.vh”
module halfAutomaton(   output  logic [1:0]   state ,
                        input   logic         in    ,
                        input   logic         reset ,
                        input   logic         clock );

// f: the state transition function
    always_ff @(posedge clock)
      if (reset) state <= ‘init_state;
       else
         case(state)
          ‘init_state   : if (in == ‘b) state <= ‘one_b_state;
```

```
                          else          state <= ‘init_state  ;
          ‘one_b_state : if (in == ‘b) state <= ‘final_state;
                          else          state <= ‘init_state  ;
          ‘final_state :                state <= ‘final_state;
          default      :                state <= ‘init_state  ;
        endcase
endmodule
```

For safety in operation, but also for an easy validation of the project, the automaton has an initial state, i.e., it is a strictly initial automaton in the `init_state` state.

Note the ”friendly” way in which the description is made. The automaton’s behavior can be read very easily due to the way in which we represented the behavior symbolically. We can make the following reading:

> *At* `reset = 1` *the automaton goes into* `init_state`. *In* `init_state` *if* `b` *is received, then the automaton goes to the state in which a* `b` *was received:* `one_b_state`*, if not it remains in* `init_state`. *In* `one_b_state`*, if the same symbol* `b` *is received on the input, then the automaton goes to the final recognition state,* `final_state`*, if not then it returns to* `init_state` *to restart the search. In* `final_state` *it automatically remains blocked until a new signal* `reset` *reinitializes the search.*

Immediate Moore To complete the project of the finite automaton, we must include the two previously defined files in the topmodule that provides the output signal. A first form is the one in which the automaton responds immediately strictly according to its internal state. The transition function of the output does not depend on the folded value of the input. It is the form of an immediate Moore type automaton whose description follows:

```
/* ***********************************************************
File name:       immediateMooreAutomaton.sv
Circuit name:    Double b detector
Description:     behavioral description of the Moore finite
                 automaton designed to detect ’bb’ in a stream
                 of symbols belonging to the set {a,b}
*********************************************************** */
‘include ”defines.vh”
module immediateMooreAutomaton( output  logic out          ,
                                input   logic in           ,
                                input   logic reset, clock);
    logic [1:0] state   ;

    halfAutomaton ha(state, in, reset, clock);
// g: the output combinational transition function
    always_comb case(state)
                ‘init_state  : out = ‘no  ;
                ‘one_b_state : out = ‘no  ;
                ‘final_state : out = ‘yes ;
```

```
                        default      : out = 1’bx ;
                    endcase
endmodule
```

In the Moore form, the automaton immediately responds with a latency of one clock cycle, signaling the appearance of the second b. This method of implementation is the simplest, having the disadvantage of an output signal that can be loaded by parasitic transitions due to the hazard phenomenon. Sometimes, the latency of a cycle can be a problem.

Delayed Moore The output signal will be able to be cleaned in a radical way from the phenomena of combinational hazard by opting for the delayed Moore version. The output register will work as a pipeline register and allow a ”clean” signal synchronized with the system clock. The price paid for this advantage is the increase in latency by one more unit. For the delayed version there is the following code:

```
/**********************************************************
File name:      delayedMooreAutomaton.sv
Circuit name:   An example of Moore−type automaton
Description:    behavioral description of the delayed Moore
                finite automaton designed to detect ’bb’ in
                streams of symbols belonging to the set {a,b}
**********************************************************/
‘include ”defines.vh”
module delayedMooreAutomaton(   output  logic out         ,
                                input   logic in          ,
                                input   logic reset, clock);
    logic   [1:0]   state   ;

    halfAutomaton ha(state, in, reset, clock);

// g: the delayed transition function
    always_ff @(posedge clock) case(state)
                                    ‘init_state  : out <= ‘no ;
                                    ‘one_b_state : out <= ‘no ;
                                    ‘final_state : out <= ‘yes;
                                    default      : out <= 1’bx;
                                endcase
endmodule
```

The delayed Moore version is the simplest and the more robust implementation of the detector we design. the output circuit is simple, because it do not depend by input, and the output signal is easy to use because i fast and clean. It is recommended if the two-cycle latency can be ”absorbed” in the system design.

Immediate Mealy If we are looking for the fastest response of the automaton, the Mealy immediate version is the solution. But the price we will pay is not negligible:

- the output signal depends on the temporal behavior of the circuit that generates the input of the automaton we are designing
- the combinational hazard cannot be eliminated
- the circuit that calculates the output function, g, is larger and more complex.

Whenever possible, this version should be avoided. We have an extra chance when we use it if we control the whole system in which the machine works.

```
/* *******************************************************
File name:        immediateMealyAutomaton.sv
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}
******************************************************* */
‘include ”defines.vh”
 module immediateMealyAutomaton(output  logic out        ,
                                input   logic in         ,
                                input   logic reset, clock);
    logic [1:0]   state   ;

    halfAutomaton ha(state, in, reset, clock);

// g: the output combinational transition function
    always_comb
        case(state)
            ‘init_state  :              out = ‘no ;
            ‘one_b_state : if (in == ‘b) out = ‘yes;
                            else         out = ‘no ;
            ‘final_state :              out = ‘yes;
            default      :              out = 1'bx;
        endcase
 endmodule
```

The set-up time of the input signal is defined only for the state transition function, f, because for the output transition function it is not possible because of the combinational nature of the function g.

Delayed Mealy The situation starts to become more controllable in the case of the delayed Mealy version. There still remains the problem of the set-up time, which must be defined both with respect to the state register and with respect to the output pipeline register.

```
/* *******************************************************
File name:        mealy_delayed_automaton.sv
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}
************************************************************/
‘include ”defines.vh”
module delayedMealyAutomaton(   output  logic out         ,
                                input   logic in          ,
                                input   logic reset, clock);
    logic [1:0]   state   ;

    halfAutomaton ha(state, in, reset, clock);
// g: the delayed transition function
    always_ff @(posedge clock)
        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
```

From the point of view of latency, this version behaves the same as the immediate Moore automaton, but has the advantage of a synchronous output with the system clock.

4.2.2 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 4.9 *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 4.10, where:*

- 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*

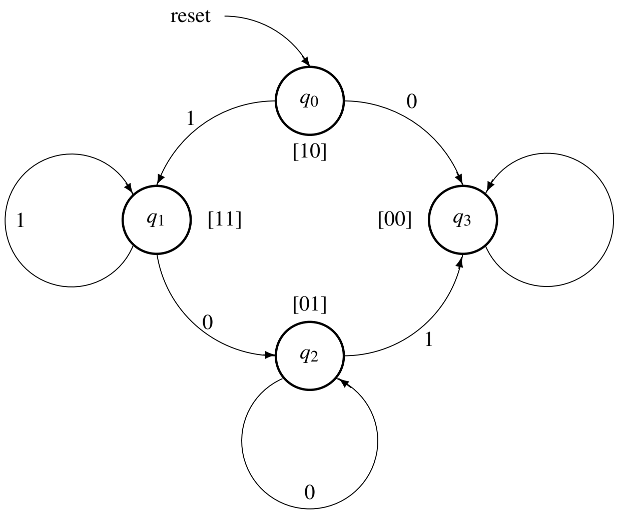


Figure 4.10: **Transition diagram.** The transition diagram for the half-automaton which recognizes strings of form 1^n0^m, for $n \geq 1$ and $m \geq 1$. Each circle represent a state, each (marked) arrow represent a (conditioned) transition.

- q_3 - *the* error *state: an incorrect string was received.*

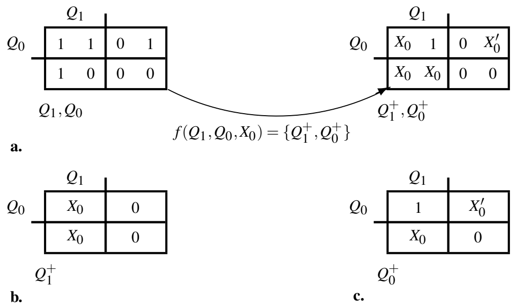


Figure 4.11: **VK transition maps.** The VK transition map for the half-automaton used to recognize 1^n0^m, for $n \geq 1$ and $m \geq 1$. **a**. The state transition function f. **b**. The VK diagram for the next most significant state bit, extracted from the previous full diagram. **c**. The VK diagram for the next least significant state bit.

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 4.10. Results the transition diagram presented in Figure 4.11. The resulting transition functions are:

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

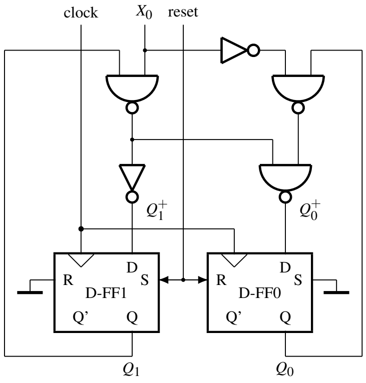


Figure 4.12: **A 4-state finite half-automaton.** The structure of the finite half-automaton used to recognize binary string belonging to the 1^n0^m set of strings.

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

(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 4.34 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.sv
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  logic [1:0]  state  ,
                input   logic        in     ,
                input   logic        reset  ,
                input   logic        clock  );

    always_ff @(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 4.10 *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:*

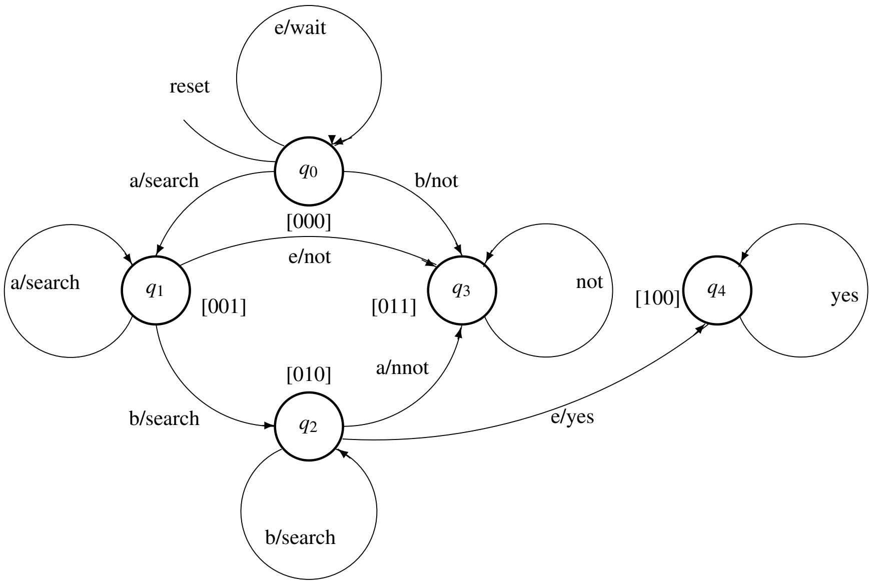


Figure 4.13:

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

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 4.13. The automaton has the following 5 states:

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

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:

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 4.1: The truth table for the transition functions.

```
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 4.1 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 4.14, 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*

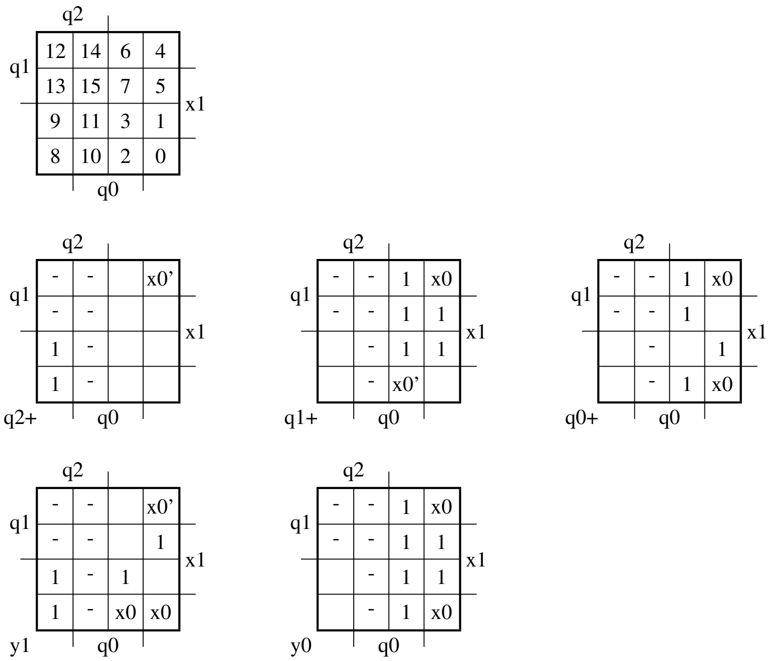


Figure 4.14: The V-K diagrams for the state and output transition functions.

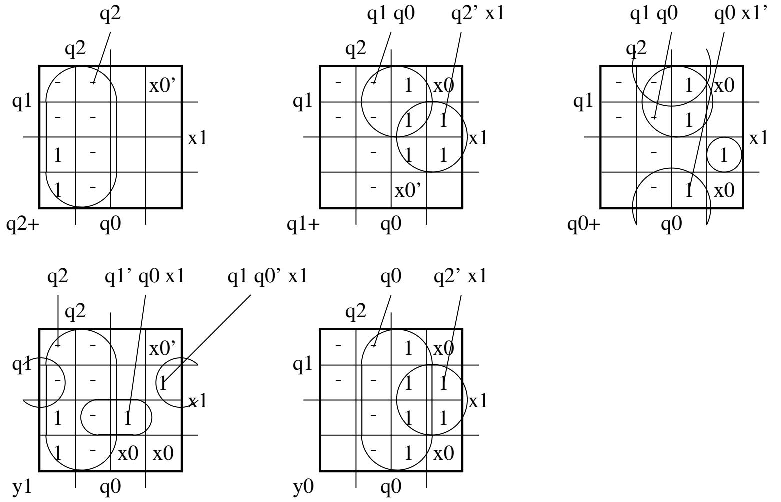


Figure 4.15: The first stage in the extracting algebraic expressions from V-K diagrams: the functions included in diagrams are ignored.

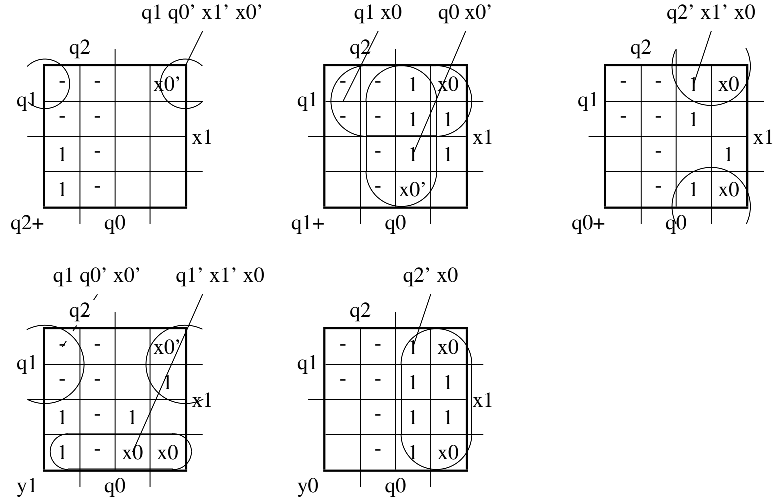


Figure 4.16: The second stage in the extracting algebraic expressions from V-K diagrams: the 1s are considered "don't care"s.

- *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`
- *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 4.15. The second stage (which considers the 1s as "don't care"s) is represented in Figure 4.16 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.

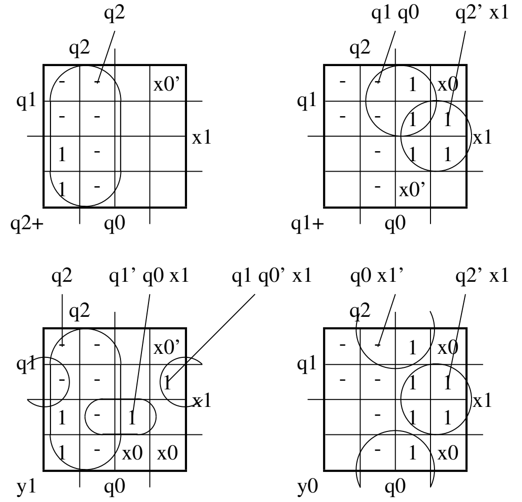


Figure 4.17: The first stage in the extracting algebraic expressions from V-K diagrams.

In Figure 4.17 the diagram for `y0` *is modified: instead of the surface* `q0`*, emphasize in Figure 4.15, 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.*

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 4.19 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 4.11 *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 4.20

From V-K diagrams result the following expressions :

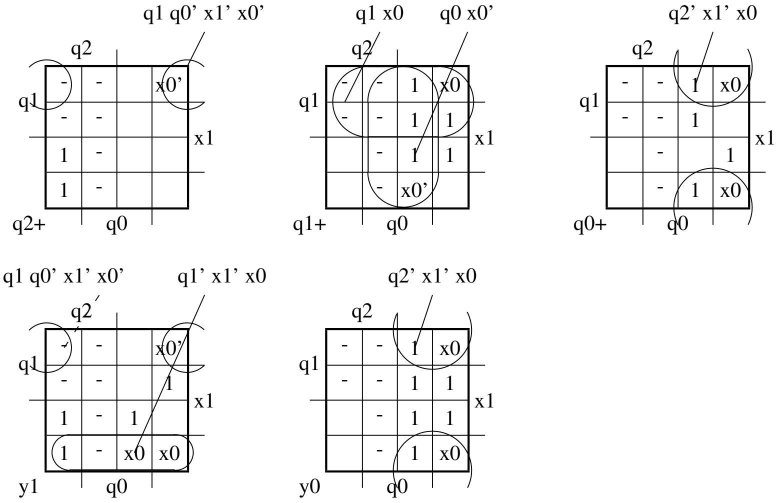


Figure 4.18: The second stage in the extracting algebraic expressions from V-K diagrams.

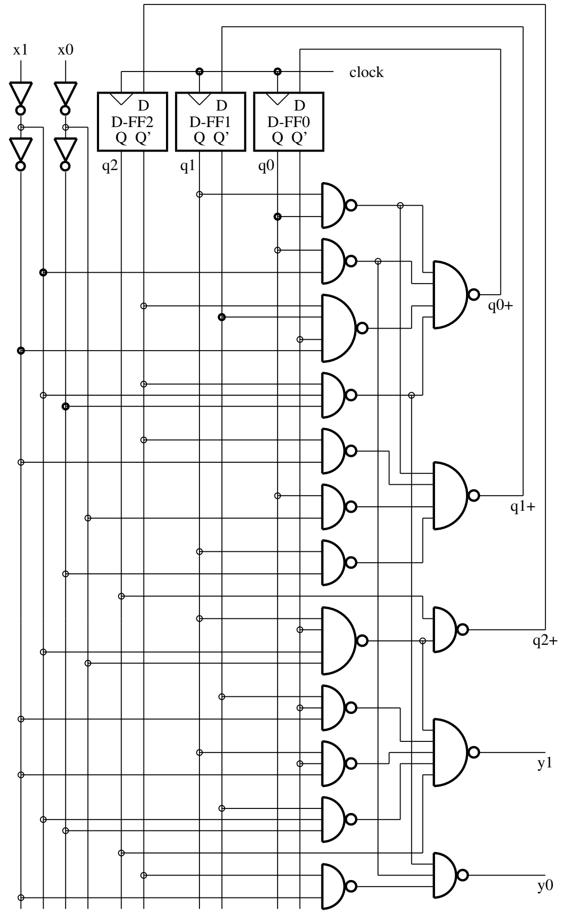


Figure 4.19: The resulting circuit.

q2

q1	- - -	- - -	0 1 1	0 1 0
	- - -	x1 1 (x1 + x0)	x1 x0' 1	0 x1 (x1 + x0)

q2+ q1+ q0+ q0

q2

q1	- -	- -	0 1	1 0
	- -	x0' (x1 + x0)	(x1 + x0) 1	x0 (x1 + x0)

y1 y0 q0

Figure 4.20:

```
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 4.21

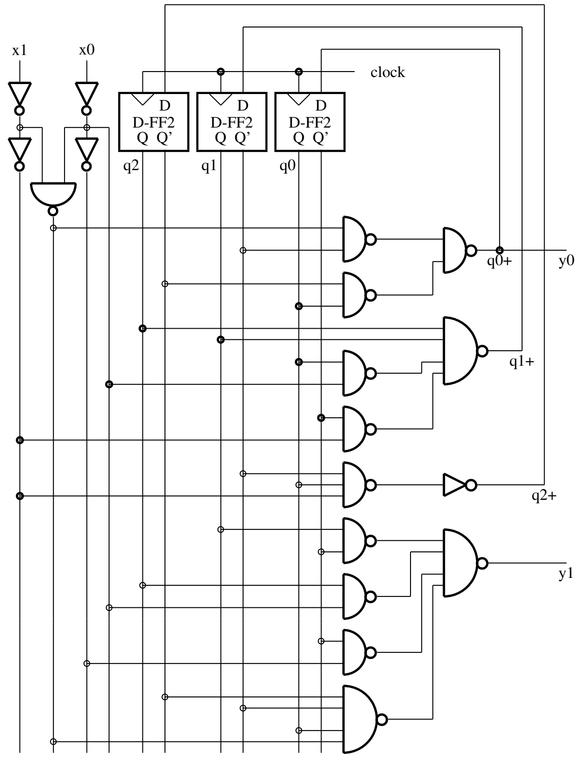


Figure 4.21: The circuit for the codding dominated by the reduce dependency coding style.

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

◇

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 4.12 *Let be a first example showing the structural dependency by the state encoding. The automaton described in Figure 4.22a 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 4.22b:*

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

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

$$Y_1 = Q_0 + X_0 Q_1'$$

$$Y_0 = Q_1' Q_0'.$$

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 4.22b:*

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

$$Q_0^+ = Q_1'$$

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

$$Y_0 = Q_0'.$$

Obviously the second codding version provides a simpler and smaller combinational circuit associated to the same external behavior. In Figure 4.23 the resulting circuit is represented. ⋄

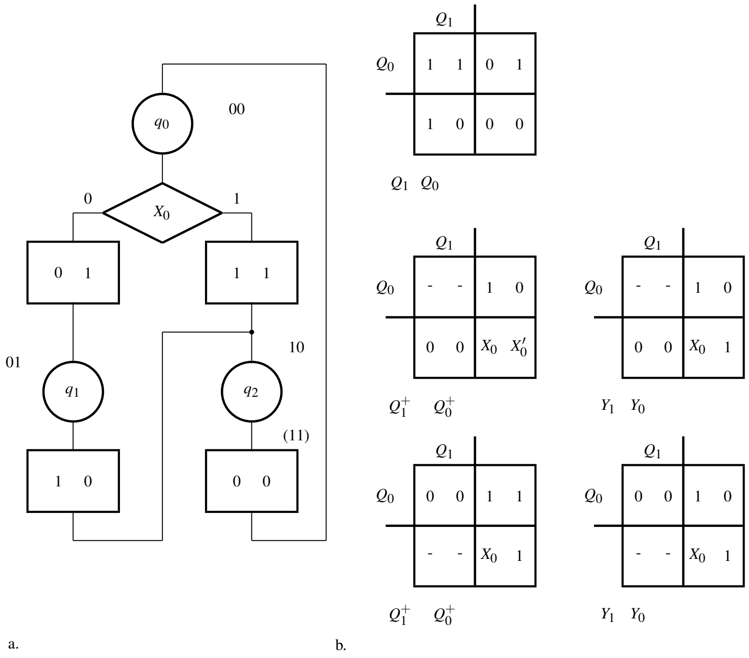


Figure 4.22: **A 3-state automaton with two different state encoding. a.** The flow-chart describing the behavior. **b.** The VK diagrams used to implement the automaton: the reference diagram for states, two transition diagrams used for the first code assignment, and two for the second state assignment.

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

Definition 4.8 *Codding with minimal variation means successive state are codded with minimal Hamming distance.* ⋄

Example 4.13 *Let be the fragment of a flow chart represented in Figure 4.24a. 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 4.14 *Some times the minimal variation encoding is not possible. An example is presented in Figure 4.24b, 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.

Definition 4.9 *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).* ⋄

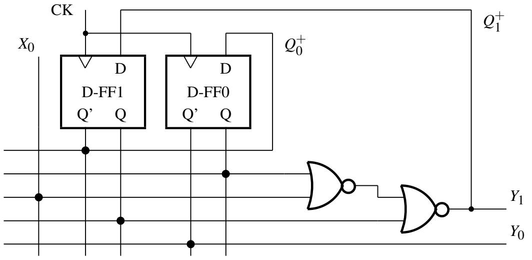


Figure 4.23: **The resulting circuit** It is done for the second state assignment of the automaton defined in Figure 4.22a.

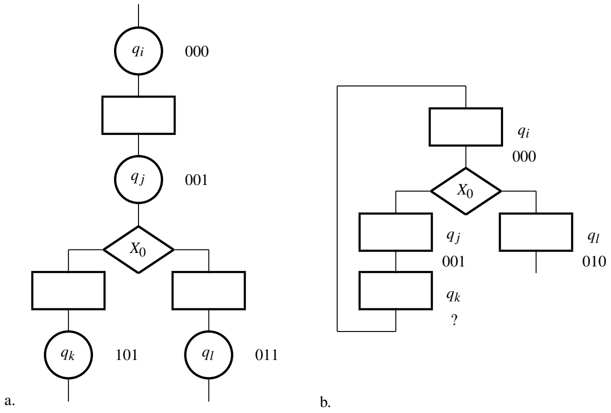


Figure 4.24: **Minimal variation encoding. a.** An example. **b.** An example where the minimal variation encoding is not possible.

Example 4.15 *In Figure 4.25a 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 4.16 *In Figure 4.25b 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.

Theorem 4.2 *If the transition from a certain state depends by more than one 1-bit variable, the reduced dependency encoding can not be provided for more than one of them.* ⋄

The reduced dependency encoding is used to minimize the transition function because it allows to minimize the number of included variables in the VK state transition diagrams. Also, we will learn soon that this encoding style is very helpful in dealing with asynchronous input variables.

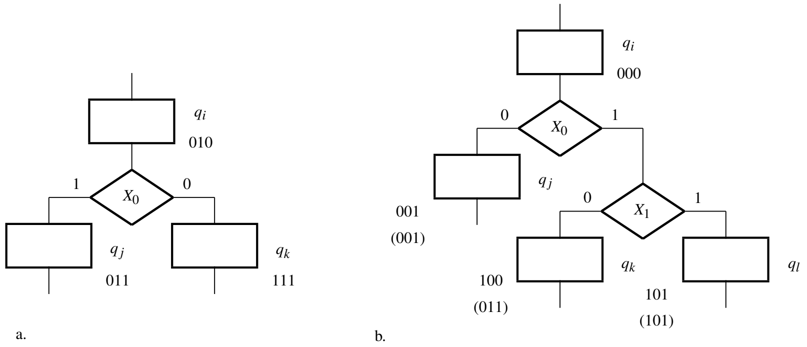


Figure 4.25: **Examples of reduced dependency encoding. a.** The transition from the state is conditioned by the value of a single 1-bit variable. **b.** The transition from the state is conditioned by two 1-bit variables.

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 4.10 *Incremental encoding means to assign, whenever it is possible, for a state following q_i a code determined by incrementing the code of q_i.* ⋄

Incremental encoding can be useful for reducing the complexity of a big automaton, even if sometimes the price will be to increase the size. But, as we more frequently learn, bigger size is a good price for reducing complexity.

One-hot state encoding The register is the simple part of an automaton and the combinational circuits computing the state transition function and the output function represent the complex part of the automaton. More, the speed of the automaton is limited mainly by the size and depth of the associated combinational circuits. Therefore, in order to increase the simplicity and the speed of an automaton we can use a 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 4.11 *The one-hot state encoding associates to each state a bit, and consequently the state register has a number of flip-flops equal with the number of states.* ⋄

All previous state encodings used a log-number of bits to encode the state. The size of the state register will grow, using one-hot encoding, from $O(log\ n)$ to $O(n)$ for an n-state finite automaton. Deserves to pay sometimes this price for various reasons, such as speed, signal accuracy, simplicity, ...

Minimizing finite automata

There are formal 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 4.17 *Let be the finite automaton described by the flow-chart from Figure 4.26. 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:

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

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

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

$$Y_2 = Q_2 + Q_1Q_0$$

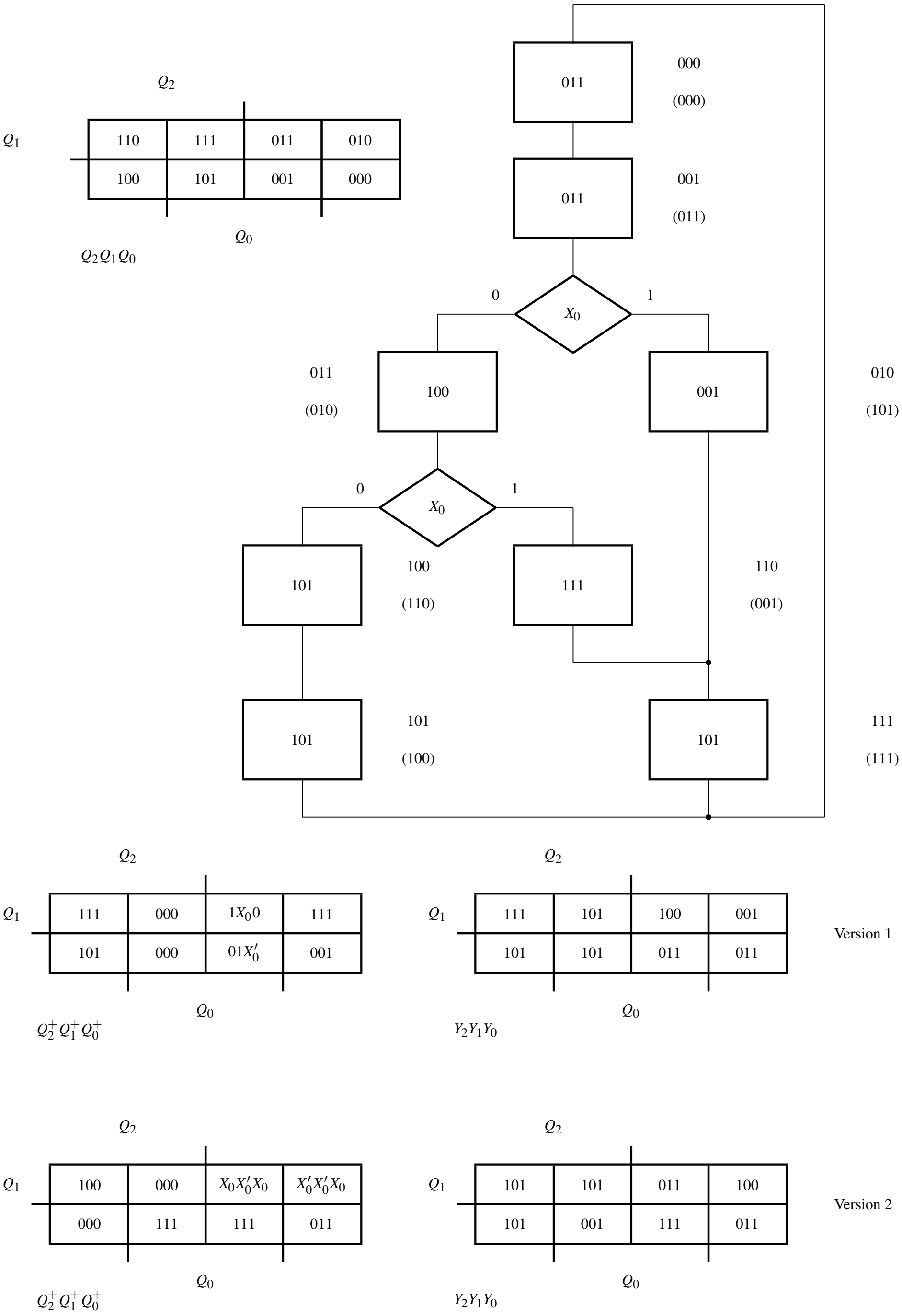
$$Y_1 = Q_2Q_1Q_0' + Q_2'Q_1'$$

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

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

For the second codding version results the expressions:

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



$Q_2Q_1Q_0$ (Q_2, Q_1, Q_0)

Q_1	110	111	011	010
	100	101	001	000

Version 1

$Q_2^+Q_1^+Q_0^+$ (Q_2, Q_1, Q_0)

Q_1	111	000	$1X_00$	111
	101	000	$01X_0'$	001

$Y_2Y_1Y_0$ (Q_2, Q_1, Q_0)

Q_1	111	101	100	001
	101	101	011	011

Version 2

$Q_2^+Q_1^+Q_0^+$ (Q_2, Q_1, Q_0)

Q_1	100	000	$X_0X_0'X_0$	$X_0'X_0'X_0$
	000	111	111	011

$Y_2Y_1Y_0$ (Q_2, Q_1, Q_0)

Q_1	101	101	011	100
	101	001	111	011

Figure 4.26: **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.

$$Q_1^+ = Q_1'Q_0 + Q_2'Q_1' + Q_2'X_0'$$
$$Q_0^+ = Q_1'Q_0 + Q_2'Q_1' + Q_2'X_0$$
$$Y_2 = Q_2Q_0' + Q_2Q_1 + Q_2'Q_1'Q_0 + Q_1Q_0'$$
$$Y_1 = Q_2'Q_0 + Q_2'Q_1'$$
$$Y_0 = Q_2 + Q_1' + Q_0$$

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

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 4.18 *Let be the automaton described by the flow-chart from Figure 4.27, 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 ($Q_2Q_1Q_0$).*

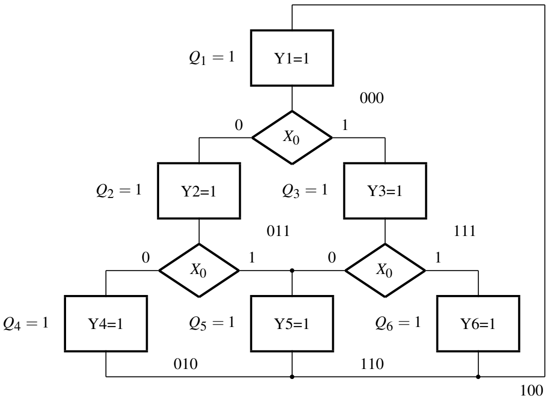


Figure 4.27: Minimizing the complexity using one-hot encoding.

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:*

$$Q^+{}_1 = Q_4 + Q_5 + Q_6$$
$$Q^+{}_2 = Q_1X_0'$$
$$Q^+{}_3 = Q_1X_0$$
$$Q^+{}_4 = Q_2X_0'$$
$$Q^+{}_5 = Q_2X_0 + Q_3X_0'$$

$$Q^+{}_6 = Q_3X_0$$

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 4.27 for the actual binary codes) are:*

$$Q^+{}_2 = Q_2Q_0 + Q_0X_0 + Q_2'Q_1'X_0$$

$$Q^+{}_1 = Q_2'Q_0 + Q_2'Q_1' + Q_0X_0'$$

$$Q^+{}_0 = Q_2'Q_1'$$

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. $\diamond$

4.2.3 Control Automata (CROM)

When we are faced with problems that require a complex automaton of large dimensions, there is the possibility of segregating in its structure simple substructures that allow the total compactness of the system to be reduced. This is the case of some *automatic control devices* used, for example, as subsystems in microprogrammed systems.

A control automaton is included in a system using three main connections (see Figure 4.28):

- the p-bit input `operation[p-1:0]` selects the control sequence to be executed by the control automaton (it *receives* the information about "what to do"); it is used to part the ROM in 2^p parts, each having the **same** dimension; in each part a sequence of maximum 2^n operation can be "stored" for execution

- the m-bit command output, `command[m-1:0]`, the control automaton uses to *generate* "the command" toward the controlled subsystem

- the n-bit input `flags[q-1:0]` the control automaton uses to *receive* information, represented by some *independent bits*, about "what happens" in the controlled subsystems commanded by the output `command[m-1:0]`.

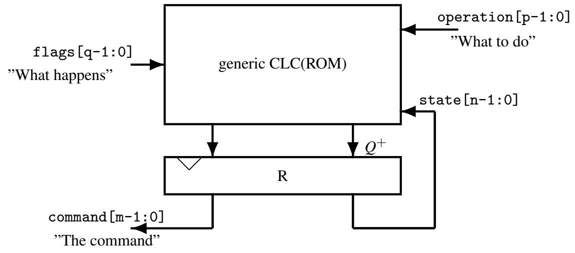


Figure 4.28: **Control Automaton.** The functional definition of control automaton. Control means to issue commands and to receive back signals (flags) characterizing the effect of the command.

Since, as we know, the maximum theoretical size of a random (complex) combinational circuit depends exponentially on the number of inputs, we must treat the number of inputs of the system in the generic version of Figure 6.5 very carefully. Some very useful observations can be made that allow the drastic reduction of the complex combinational circuit:

- the input `operation[p-1:0]` is considered only when a sequence of micro-instructions is initiated
- the inputs `flags[q-1:0]` have independent meaning and are considered independently at different times of the generation of microinstruction sequences
- an important share of the transitions of this automaton generates linear sequences of microinstructions, so the next state can be coded by incrementing the current one

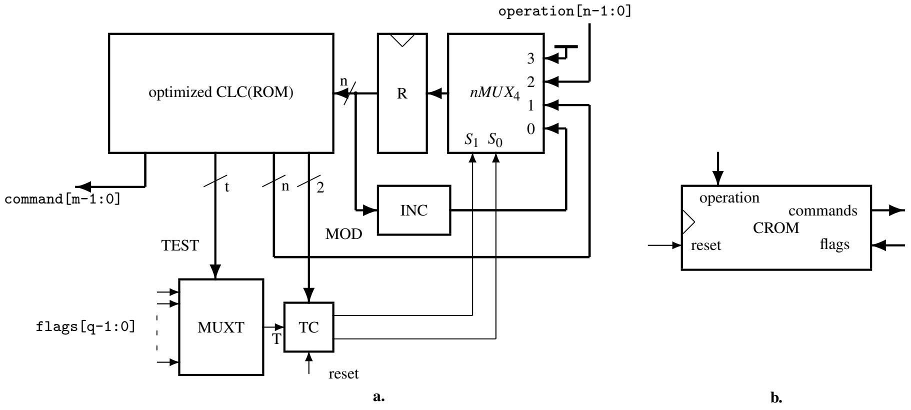


Figure 4.29: **The simplest Controller with ROM (CROM). a.** The Moore form of control automaton is optimized using an incremented circuit (INC) to compute the most frequent next address for ROM. **b.** The logic symbol for CROM.

In Figure 4.29 is represented an optimized form of the control automaton in which a series of simple circuits have been introduced that allow the minimization of the large and complex circuit represented by the combinational logic circuit, CLC, which can be implemented in the form of a ROM (Read - Only Memory). The role of these circuits is as follows:

MUXT : is used to select in each state only one flag from the set of q, because the current use of such a system thought us that the sequence of microinstructions depends, in most o f cases, only by one flag in any cycle

$nMUX_4$: selects the transition mode of the automaton:

$S_1S_0 = 00$: the system is initialised in state $00\ldots0$

$S_1S_0 = 01$: the system takes one of its initial state corresponding to the microprogram

$S_1S_0 = 10$: the next state of the system is selected from the output of the random CLC

$S_1S_0 = 11$: the next state of the system is obtained by incrementing the value of the current state

INC : is the simple circuit of an increment circuit used to determine the next state of the system for the linear sequence of the microprogram

TC : is a very simple and small combinational circuit used to select the transition mode of the automaton according to the current state and of the flag selected by MUXT.

The size of `optimized CLC(ROM)` is dramatically reduced compared to the size of `generic CLC(ROM)` because, as we know, each remove of an input of a CLC reduces its theoretical size to half.

CROM is a very good example for using, whenever possible, the segregation of simplicity from an apparently very complex reality. With this circuit, we approach the class of automata circuits whose loop is mainly closed by simple, functional circuits.

4.3 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.

4.3.1 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 4.30, 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*.

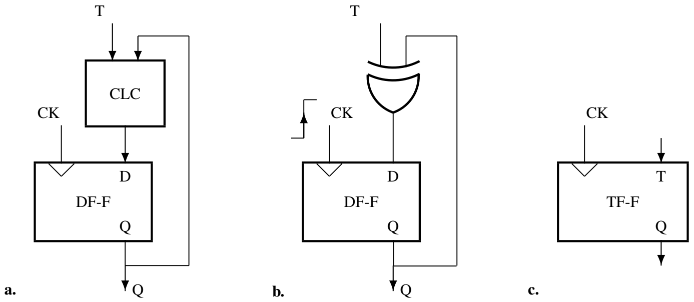


Figure 4.30: **The T flip-flop. a.** It is the simplest automaton because: has 1-bit state register (a DF-F), a 2-input loop circuit (one as automaton input and another to close the loop), and direct output from the state register. **b.** The structure of the T flip-flop: the XOR_2 circuits complements 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 4.30b). 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).

4.3.2 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 4.12 *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 4.31).* ⋄

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

Definition 4.13 *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 4.31). $COUNT_1$ is a T flip-flop and an AND_2 with the inputs Q_0 and INC_0 which generates INC_1.* ⋄

Example 4.19 **The Verilog description of a synchronous counter follows:*

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

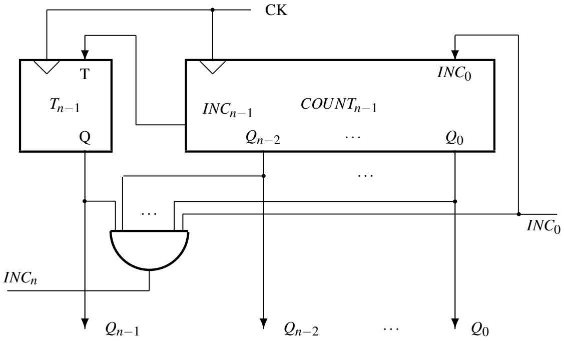


Figure 4.31: **The synchronous counter.** The recursive definition of a synchronous counter has $S_{COUNT}(n) \in O(n^2)$ and $T_{COUNT}(n) \in O(log\, n)$, because for the i-th range one TF-F and one AND_i are added.

```
/* *************************************************************************
File name:        sync_counter.sv
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  logic [n-1:0] out    ,
                                        output  logic         inc_n  ,
                                        input   logic         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.sv
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  logic [n-1:0] out     ,
                                      input   logic [n-1:0] in      ,
                                      input   logic         reset   ,
                                                            clock   );
    always_ff @(posedge clock) if (reset)  out <= 0;
                               else        out <= out ^ in;
endmodule
```

The reset input is added because it is used in real applications. Also, a reset input is good in simulation because makes the simulation possible allowing an initial value for the flip-flops (`reg[n-1:0] out` *in module* `t_reg`*) used in design.* ⋄

It is obvious that $C_{COUNT}(n) \in O(1)$ 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.sv
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  logic [n-1:0] out    ,
                                        input   logic [n-1:0] in     ,
                                        input   logic         reset  ,
                                                            load     ,
                                                            down     ,
                                                            count    ,
                                                            clock    );
    always_ff @(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.

Program Counter (PC)

Program Counter (PC) is a special counter can be used as a logic block in the structure of a processor. It control the evolution of the program execution. A version of a simple PC is represented in Figure 4.32, where:

- a 4-input multiplexor selects to the input of a register:
  - the incremented value of the PC stored in register in order to provide the address for the next instruction on the linear part of the program
  - the value of PC added with the jump address in order to allow program to perform a unconditioned or a conditioned jump
  - the address of an absolute jump provided by the current instruction
  - the address for an absolute jump when a call or a return instruction is executed
- a zero detector combinational circuit provide the jump condition to
- a small random combinational circuit which generate the selection code for the multiplexor and the increment command for the increment circuit according the command `next` generated by one field of the instruction and according to the condition provided by `zero` and the interrupt signal `int`.

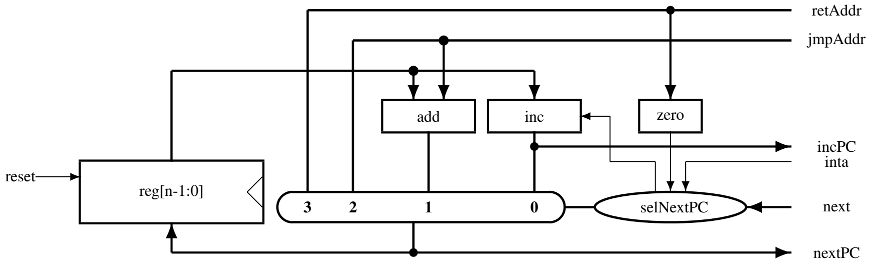


Figure 4.32: Program Counter

The transition set this circuit is controlled by the `selNextPC` combinational circuit described as follows::

```
/* *******************************************************
File: selNextPC.sv
Name:
Description:
******************************************************* */
module selNextPC(   output  logic [1:0]   sel ,
                    output  logic         inc ,
                    input   logic [2:0]   next,
                    input   logic         zero,
                    input   logic         int );

    always_comb
     if (int)       {inc, sel} = 3'bx11 ;
      else
        case(next)
                    // increment PC
            3'b000: {inc,sel} = 3'b100  ;
                    // unconditional relative jump
            3'b001: {inc,sel} = 3'bx01  ;
                    // unconditional absolute jump
            3'b010: {inc,sel} = 3'bx10  ;
                    // return from subroutine
            3'b011: {inc,sel} = 3'bx11  ;
                    // branch if zero
            3'b100: {inc,sel} = zero ? 3'bx01 : 3'b100  ;
                    // branch if not zero
            3'b101: {inc,sel} = !zero ? 3'bx01 : 3'b100 ;
                    // halt
            3'b110: {inc,sel} = 3'b000  ;
            default:{inc,sel} = 3'b100  ;
        endcase
endmodule
```

In a real system, the connection of this circuit will be done taking into account the temporal relationships that must be optimized. As a rule, such relationships are established using pipeline registers.

4.3.3 Structured State Space Automaton(S^{3}A)

Definition 4.14 *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 4.15 *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* $\mathscr{S}$
- $g : (L \times R) \to (S_L \times S_R)$ *is the output transition function*
- $f : (H \times X \times D \times L \times R \times \mathscr{S}) \to S_D$ *is the state transition function of form* $h_H : (X, S_L, S_R) \to S_D$.

◇

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.

4.3.4 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 4.16 *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:

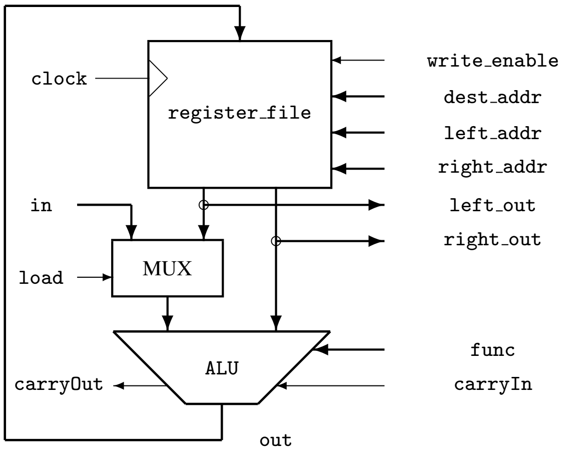


Figure 4.33: 32-bit RALU.

- $\mathbf{Q} = (Q_0 \times Q_1 \times \ldots, \times Q_{s-1})$ *: 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* $i \in LA$, $j \in RA$, $k \in DA$

◇

Example 4.20 *Let be a RALU designed with two modules already presented in the previous sections: the ALU exemplified in Example 2.4 and the register file presented in Simulation 3.12. In Figure 4.33 is represented the schematic of a 32-bit RALU.*

```
/*************************************************************************
File:             RALU.sv
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  logic [31:0]  left_out    ,
                output  logic [31:0]  right_out   ,
                output  logic         carryOut    ,
                input   logic         load        ,
                input   logic [3:0]   right_addr  ,
                input   logic [3:0]   dest_addr   ,
                input   logic         write_enable,
                input   logic [31:0]  in          ,
                input   logic         carryIn     ,
                input   logic [2:0]   func        ,
                input   logic         clock       );
    logic  [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
```

◇

4.4 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[3]. 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.

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?

4.5 Problems

Problem 4.1 *Design (schematic on paper and System Verilog code) and simulate a sequential circuit with the state stored in* `serialReg[4:0]` *which evolves as follows in each clock cycle:*

```
   serialReg[4:0] <= XOR(serialReg[2], serialReg[1]), serialReg[4:1]
```

[3] Asynchronous automata are possible but their design is restricted by to complex additional criteria. Therefore, asynchronous design is avoided until stronger reason will force us to use it.

Problem 4.2 *Design an automaton which receives sequences of the form* 0^n1 *and detects if the number of 0s is odd or even. Draw the graph, write the System Verilog code and simulate the behavior.*

Problem 4.3 *Design a finite automaton which receives streams of symbols fro the set* $\{\texttt{a,b,c}\}$ *and detects the sequences of form* a^nb^mc*, with* $n,m \geq 0$.

Problem 4.4 *Design in System Verilog and test an automaton that validates the entry of a given text. The text is ”cat”. The data is validated by ”enable” active high and the 2 outputs are activated as follows: ”pass” is activated when the password was entered correctly, ”fail” is activated when the password was entered incorrectly. The automaton must not remain blocked so that the text can be re-entered.*

It is requested the flowchart / transition graph of the automaton is also required. Interface:

```
input   logic          clock,
input   logic          reset,
input   logic          enable,
input   logic   [7:0]  dataIn,
output  logic          pass,
output  logic          fail,
```

Problem 4.5 *The implementation and simulation of a frequency divider is required that receives a clock signal with a period of* #2 *and generates a signal with a period of* #26 *(fixed).*

Problem 4.6 *Design and simulate a system that receives at its input two pulse trains that change synchronously with the clock but have a random shape (the alternation of 0 and 1 is arbitrary). The system starts from reset and stops with the output of an active bit on 1 when the difference between the clock intervals in which the inputs were identical and the clock intervals in which they were different exceeds the value of the input* `in[7:0]`.

Problem 4.7 *Design a finite automaton that receives as input strings formed by symbols belonging to the set* $\{\texttt{a, b, c, e}\}$ *and recognizes sequences of the type:*

$$\ldots eea^nb^2c^mee\ldots$$

for $n,m > 0$.

1. *Draw the graph for the associated Mealy automaton*
2. *Simulate the automaton in the delayed Mealy version*

Problem 4.8 *Design and simulate an automaton that measures the length of a synchronously received pulse. The automaton has the following states:*

- *Initial state in which it waits for the appearance of pulse X*
- *State in which it “measures” the duration of the pulse in the number of clocks it has been at 1*
- *State in which it emits, together with a validation signal, the number of maximum 32 bits obtained. From this state it returns to the initial state.*

```
module aut( input   logic           x, reset, clock,
            output  logic           valid,
            output  logic   [31:0]  value);

...
endmodule
```

Problem 4.9 *Design a finite automaton that receives as input strings formed by symbols belonging to the set* {`a, b, c, e`} *and recognizes sequences of the type:*

$$\ldots eea^n bc^m ee \ldots$$

for $n, m > 0$.

1. *Draw the graph*
2. *Draw the flowchart for the associated Mealy automaton*
3. *Simulate the automaton.*

Problem 4.10 *Write the Verilog structural description for the universal 2-input, 2-state programmable automaton.*

Problem 4.11 *Design a reversible counter defined as follows:*

```
module smartest_counter   #(parameter n = 16)
       (   output  logic [n-1:0] out    ,
           input   logic [n-1:0] in     ,  // preset value
           input   logic         reset  ,  // reset counter to zero
           input   logic         load   ,  // load counter with 'in'
           input   logic         down   ,  // counts down if (count)
           input   logic         count  ,  // counts up or down
           input   logic         clock  );
   //...
endmodule
```

Problem 4.12 *Let be the finite automaton defined in Figure 4.34. Do the following:*

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 two versions of the finite automaton.*

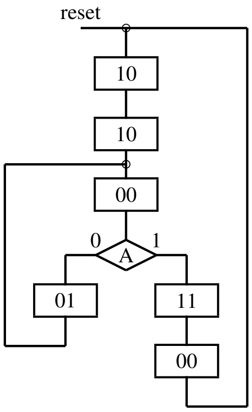


Figure 4.34:

Chapter 5

PROCESSORS: Third order, 3-loop digital systems

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*.

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 5.1a).
- The second type (see Figure 5.1b) has on the loop a *memory* circuit (1-OS).
- The third type connects in a loop two automata (see Figure 5.1c). 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**.

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

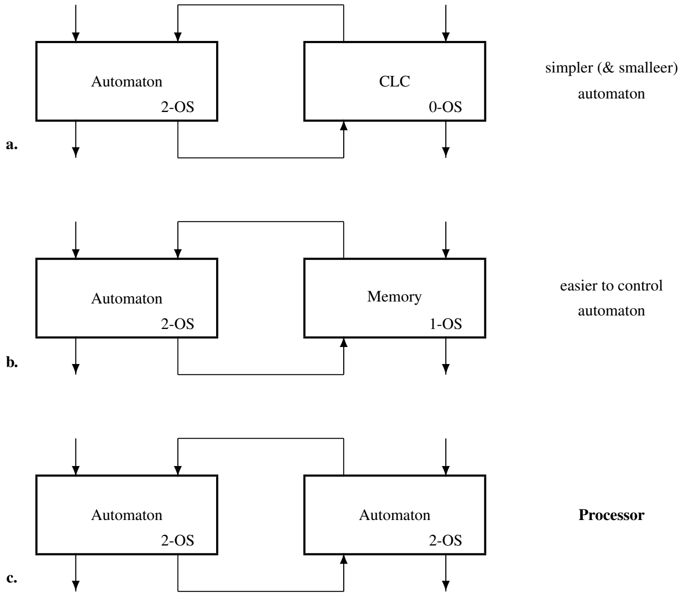


Figure 5.1: **The three types of 3-OS machines. a.** The third loop is closed through a combinational circuit resulting less complex, sometimes smaller, finite automaton. **b.** The third loop is closed through memories allowing a simplest control. **c.** The third loop is closed through another automaton resulting the **Processor**: the most complex and powerful circuit.

5.1 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). 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 5.1 *The half-automaton associated to the codes assignment written in parenthesis in Figure* **??** *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*

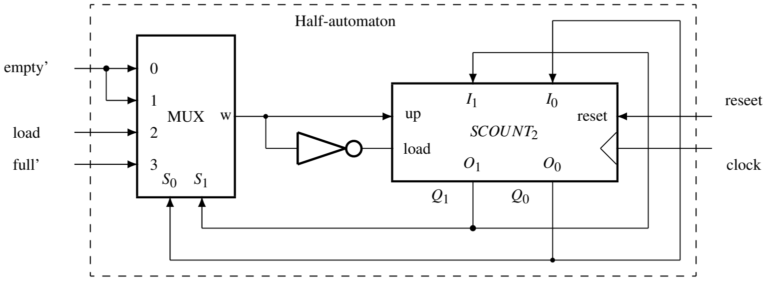


Figure 5.2: **Finite half-automaton implemented with a setable counter.** The last implementation of the half-automaton associated with FA from Figure **??** (with the function defined in Figure **??** where the states coded in parenthesis). A synchronous two-bit counter is used as state register. The simple four-input MUX commands the counter.

Results the very simple (not necessarily very small) implementation represented in Figure 5.2, 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 **??***, the version with counter is simpler, eventually smaller. But, the most important effect is the reducing complexity.* ⋄

5.2 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 – will be used (see Figure 5.3a). 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.

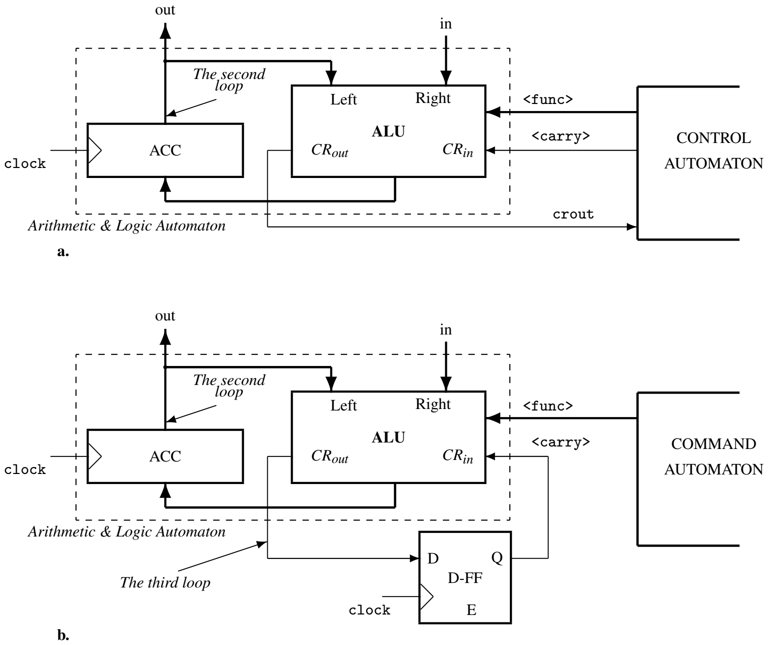


Figure 5.3: **The third loop closed over an arithmetic and logic automaton. a.** The basic structure: a simple automaton (its loop is closed through a simple combinational circuit: ALU) working under the supervision of a control automaton. **b.** The improved version, with an additional 1-bit state register 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 1: the controlled Arithmetic & Logic Automaton

In the first case ALA is **controlled** (see Figure 5.3a) 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.

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.

5.3 Processors

Third-order systems are mainly represented by processing circuits. By processing we understand the modification of the symbolic structure of data in a way described by the symbolic structure of programs. Both data and programs are stored in RAM memories. The processing involves the following operations:

FETCH : reading from memory the current instruction

OPERATION : operation according to the current instruction which may consist of

- changing the internal state of the processor
  - using only the processor's internal salt
  - data fetched from memory
- modification of the data contained in the memory

NEXT : calculating the address of the next instruction

In the last 80 years, two extreme forms of implementing a processor have been imposed:

- the CISC processor (Complex Instruction Set Computer)
- the RISC processor (Reduction Instruction Set Computer)

derived from two abstract models configured in the 1940s:

- the von Neumann abstract model
- the Harvard abstract model

The distinction between these models materialized in hardware structures that were differentiated into two clearly distinct categories:

- processors that operated on the data by interpreting the instructions, which involves decomposing each instruction, usually complex, from the program into a sequence of microinstructions
- processors that operate on data by executing instructions, which involves the operation of a simple instruction in a single clock cycle

In the set of instructions of a RISC processor, only those instructions will be found that are simple, frequent and allow the realization of any instruction from the set of a CISC processor through a sequence of instructions.

5.3.1 Interpretive Processor: CISC Processor (RALU & CROM)

Generic structure of a CISC processor is represented in FIgure 5.4, where:

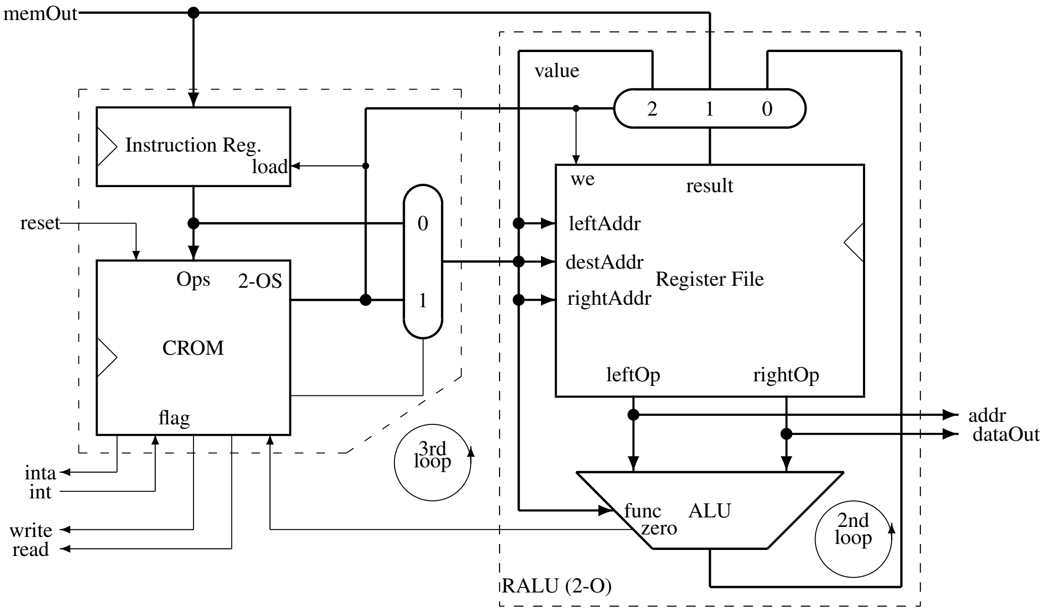


Figure 5.4: Hardware structure of a simple CISC.

Instruction Register : stores the current instruction during the interpretation which is a multi-cycle process

RALU : plays a role similar to the one in the EP structure (see **??**); additionally has the role of memorizing the address of the subroutine run as a result of accepting the interrupt signal (`inta`), to which it adds a register for saving the return address from the subroutine

CROM : controls all the stages in the interpretation process by generating the enable signals for Instruction Register, Register File, and external memory; the `inta` signal is managed according to its internal state (in some cases a `inta` can be managed by a small and simple additional automaton serially connected in CROM.)

Multiplexor : allows, under the control of CROM, to use the command fields directly form Instruction Register or generated by CROM

The too high complexity that the CROM units reached in the economy of a processor and the statistics regarding the frequency with which the complex instructions were operated, led to the abandonment of this path of evolution. But history sometimes has surprising cycles. So the CISC approach also deserves a little attention.

5.3.2 Executive Processor: RISC Processor (RALU & PC)

Generic structure of a RISC processor is represented in FIgure 5.5, where:

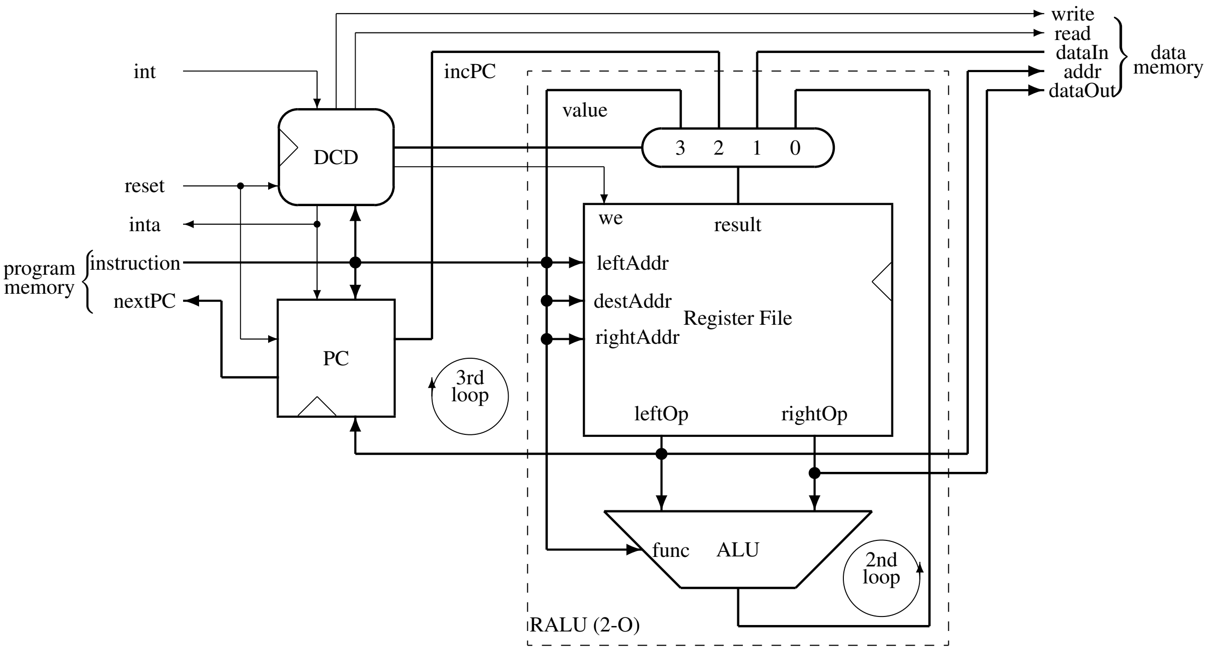


Figure 5.5: Hardware structure used for defining the architecture of a simple RISC.

RALU : plays a role similar to the one in the EP structure (see **??**), but we can give up locating the PC here, for which we promote a specific structure

PC : it operates on the PC in parallel with the RALU that operates on the data content of the Register File, a fact that allows the execution of each instruction in a single cycle (of course, if we have separate and fast enough memories for program and data)

DCD : it is a complex circuit but of very small size if we design an efficient coding of the instructions

The structure of the RISC processor is very simple, because the size of the DCD is insignificant. Instead of the very complex and large combinational circuit (or ROM) in CROM, we now have a simple circuit. From the size point of view, we can afford to increase the size of some resources because they are simple. For example, we can have a larger Register File or/and an ALU with much more and more complex functions.

Next, we will delve deeper into issues related to RISC processors.

5.4 Case Study: *toyRISC* Processor

We will present in detail the RISC version of the processor through a case study of a simple but elaborate enough structure to illustrate the processor concept. We call ***toyRISC*** the processor that we will define, design and simulate.

5.4.1 The Concept of Processor's Architecture

We will define the concept of architecture with a small historical introduction. At the beginning of the 1960s, the company IBM (International Business Machine) had already launched several computer versions on the market, enough to highlight an unpleasant effect: for each new computer, the entire software development had to be reconsidered due to the hardware structure that justifiably suffered major changes. As a consequence, on the occasion of the launch of a new series in [**?**] [**?**], the concept of the *architecture of a computing system* is proposed.

By the architecture of a processor we understand the *structural resources* by which the internal state of the processor is defined and the *set of instructions* by which this internal state evolves. Nothing about the way the structural resources are designed, or about their performance. The internal structure and its performance are at the discretion of the hardware designer. On the other hand, the way in which the set of instructions is used by the software designers is not the object of the architectural definition.

The architecture is, consequently, an interface between the hard and soft designers so that for long periods of time (in which several versions of the hard can be implemented) the already written software can be run on any new hard version . The assumed high cost of software development imposed this ”inheritance” mechanism.

The architectural approach proves very useful for a limited number of hardware generations, but becomes a burden when technological conditions and market requirements change significantly.

The representation from Figure 5.5 represents the first stage in defining the architecture of a simple RISC processor, let's call it *toyRISC*. The next stage will be the micro-architectural definition used to specify the one-cycle micro-operations performed by each blocks. Finally, the Instruction Set Architecture (ISA) defined the instructions used to assembly the programs loaded in the program memory of the system.

5.4.2 toyRISC Micro-architecture

The micro-architecture of the processor toyRISC exemplified in Figure 5.5 is defined by the following storage resources:

`pc[15:0]` : the register used as program counter

`ei` : the state of an automaton used to enable the action of the interrupt signal `int`

`rf[0:31][31:0]` : the register file

while the file `DEFINES.vh` with the micro-operations executed by PC (the control operations), RALU (the arithmetic and logic operations) and DCD (memory transfer commands). The defined micro-operations represent only a part of what can be defined on the physical support provided by the structure represented in Figure 5.5. The reader can add additional operations using the following defined mechanisms.

```
/*************************************************************************
File name: DEFINES.vh
                    MICROARCHITECTURE
```

```
*************************************************************************/
// CONTROL
‘define nop     6’b00_0000 // no operation: pc<=pc+1;
‘define rjmp    6’b00_0001 // relative jump: pc<=pc+v;
‘define zbr     6’b00_0010 // pc<=(rf[l]=0) ? pc+v:pc+1
‘define nzbr    6’b00_0011 // pc<=!(rf[l]=0) ? pc+v:pc+1
‘define ret     6’b00_0101 // return: pc<=rf[l][15:0];
‘define halt    6’b00_0110 // halt unitil interrupt
‘define eint    6’b00_1000 // set enable interrupt
‘define dint    6’b00_1001 // set disable interrupt
// ARITHMETIC & LOGIC, for these instructions: pc<=pc+1;
‘define add     6’b11_0000 // rf[d]<=rf[l]+rf[r];
‘define sub     6’b11_0001 // rf[d]<=rf[l]-rf[r];
‘define addv    6’b11_0010 // rf[d]<=rf[l]+v;
‘define mult    6’b11_0011 // rf[d]<=rf[l]*rf[r];
‘define multv   6’b11_0100 // rf[d]<=rf[l]*v;
‘define addc    6’b11_0101 // rf[d]<=(rf[l]+rf[r]}[32];
‘define subc    6’b11_0110 // rf[d]<=(rf[l]-rf[r])[32];
‘define addvc   6’b11_0111 // rf[d]<=(rf[l]+v)[32];
‘define lsh     6’b11_1000 // rf[d]<=rf[l] >> 1;
‘define ash     6’b11_1001 // rf[d]<={rf[l][31], rf[l][31:1]};
‘define move    6’b11_1010 // rf[d]<=rf[l];
‘define swap    6’b11_1011 // rf[d]<={rf[l][15:0], rf[l][31:16]};
‘define bwnot   6’b11_1100 // rf[d]<=~rf[l];
‘define bwand   6’b11_1101 // rf[d]<=rf[l]&rf[r];
‘define bwor    6’b11_1110 // rf[d]<=rf[l]|rf[r];
‘define bwxor   6’b11_1111 // rf[d]<=rf[l]^rf[r];
// MEMORY, for these instructions: pc=pc+1;
‘define read    6’b10_0000 // read from dataMemory[rf[l]];
‘define load    6’b10_0111 // rf[d]<=dataOut;
‘define store   6’b10_1000 // dataMemory[rf[l]]<=rf[r];
‘define val     6’b01_0111 // rf[d]<={{16*{v[15]}}, v};
```

The signal `reset` acts in `PC` (`pc <= -1`) and in DCD by initializing the `enable interrupt automaton` `ei` (`ei <= 0`, which means disable). Nothing in RALU is submitted to the initialization.

The PC block, controls the evolution of the program counter register `pc`. The content of this register evolves:

- by increment with 1 on the linear part of the program
- by increment with the immediate value provided by the instruction code
- by set to a value provided by the content of a register from the register file
- by set to a value provided by the instruction code (can be used to expand the set of operations already defined in Figure **??**).

The RALU block, is mainly under the direct and full control of the instruction code `instr` provided directly from the output of the program memory. Onlu the write back signal `we` is provided by DCD.

There are two instruction formats:

```
instr[31:0] = {opCode[5:0],d[4:0],l[4:0],r[4:0],noUse[10:0]} |
              {opCode[5:0],d[4:0],l[4:0],v[15:0]}
```

where:

opCode : is the operation code which specifies three types of operations:

- control operations acting on:
  - the value of the program counter, PC, which can be incremented of set to values according to the jumps or branches executed in program unconditionally or conditionally (in our simple processor, the only condition tested is if the value of the left operand is zero
  - the state of the interrupt automaton used to enable the action of the interrupt signal `intIn`
- arithmetic-logic operations modify the content the register file according to the operations performed by ALU
- data transfer operations modify the content of the register file loading the immediate data or the memory data; the content of the external data memory is modified according to address and data stored in register file

d : specifies the destination of result provided by the ALU

l : specifies the left operand of the current operation

r : specifies the right operand of the current operation

v : is the immediate value used as right operand in the current operation

The first format of instruction operate only with the content of registers, while the second operate with the content of registers and an immediate value provided in the code of instruction: `v`.

5.4.3 toyRISC Instruction Set Architecture

The micro-architecture generates ISA by associating each micro operation of the operands and the destination located in the file register. In Figure **??** is listed an inital form of ISA (it can be expanded by adding micro-operations in the file `DEFINE.hv`).

```
/**************************************************************************
                         toyRISC'S ARCHITECTURE
**************************************************************************/
    NOP          // no operation
    RJMP(lb)     // relative jumpto label 'lb'
    ZBR(l,lb)    // branch if rf[l]=zero at label 'lb'
    NZBR(l,lb)   // branch if rf[l]!=zero at label 'lb'
    RET(l)       // return from subroutine: pc<=rf[l]
    HALT         // halt until interrupt is received, pc = pc
 // for the following instructions: pc<=pc+1;
    EINT         // set enable interrupt
    DINT         // set disable interrupt
```

```
ADD(d,l,r)     // rf[d]<=rf[l]+rf[r];
SUB(d,l,r)     // rf[d]<=rf[l]-rf[r];
ADDV(d,l,v)    // rf[d]<=rf[l]+v;
MULT(d,l,r)    // rf[d]<=rf[l]*rf[r];
MULTV(d,l,v)   // rf[d]<=rf[l]*v;
ADDC(d,l,r)    // rf[d]<=(rf[l]+rf[r]}[32];
SUBC(d,l,r)    // rf[d]<=(rf[l]-rf[r])[32];
ADDVC(d,l,v)   // rf[d]<=(rf[l]+v)[32];
LSH(d,l)       // rf[d]<=rf[l] >> 1;
ASH(d,l)       // rf[d]<={rf[l][31], rf[l][31:1]};
MOVE(d,l)      // rf[d]<=rf[l];
SWAP(d,l)      // rf[d]<={rf[l][15:0], rf[l][31:16]};
NOT(d,l)       // rf[d]<=~rf[l];
AND(d,l,r)     // rf[d]<=rf[l]&rf[r];
OR(d,l,r)      // rf[d]<=rf[l]|rf[r];
XOR(d,l,r)     // rf[d]<=rf[l]^rf[r];
READ(l)        // read from dataMemory[rf[l]];
LOAD(d)        // rf[d]<=dataOut;
STORE(l,r)     // dataMemory[rf[l]]<=rf[r];
VAL(d,v)       // rf[d]<={{16*{v[15]}},v};
```

5.4.4 toyRISC Implementation

The toyRISC processor will be implemented using a behavioral description in what follows, to provide a first picture of how circuits and information ”work together” to provide complex functionality using a hardware structure dominated by *simple* structures and a *complex* software program. Indeed, the majority of the physical structure is made up of RALU and PC which are structures made up of simple circuits, and the DCD is made up of some complexly configured circuits, in the sense that their definition is in the same range as their size. On the other hand, the program that uses the toyRISC processor is a complex binary configuration, in the sense that it does not allow a lossy compression that, alone, could provide a compact representation. The program is what it is: a complex binary configuration.

The advantage of the combination between simple circuits and complex programs is at the functional level. We can build large circuits, because they are simple and we can afford complex programs because their design is done in a flexible environment where the error is tolerable because it is easily corrected. It is not so easy to correct a circuit error. In this way, the functionality of digital systems can reach very high levels of complexity.

We intend to test the competence of the toyRISC processor ignoring, at this stage, the performance that does not represent the target we are pursuing. For performance, hardware and software techniques are applied that exceed the circuit level to which we limit ourselves in this book. The next level of performance is extensively addressed in [**?**] [**?**].

Behavioral description

Because the project that we describe below emphasizes only the functional aspects leaving aside the aspects related to the performance, the description used for the main blocks are behavioral.

`toyRISC.sv` **file** looks at it is structurally described because the three files included are associated to the three main blocks represented in Figure 5.5: DCD, PC, RALU.

```
/**********************************************************************
File name: toyRISC.sv
                         toyRISC
**********************************************************************/
‘include ”DEFINES.vh”
module toyRISC( input  logic [31:0]  instr   ,
                output logic [15:0]  nextPC  ,
                input  logic         intIn   ,
                output logic         inta    ,
                input  logic [31:0]  dataIn  ,
                output logic [31:0]  dataOut ,
                output logic [15:0]  addr    ,
                output logic         read    ,
                output logic         write   ,
                input  logic         reset   ,
                input  logic         clock   );
    logic [15:0]  pc      ;
    logic         ei      ;
    logic [31:0]  rf[0:31];
    logic [5:0]   opCode  ;
    logic [4:0]   d, l, r ; // dest, lft, right
    logic [31:0]  v       ; // immediate value
    logic         we      ;
    logic [1:0]   muxSel  ;

    assign opCode   = instr[31:26]                    ;
    assign d        = instr[25:21]                    ;
    assign l        = instr[20:16]                    ;
    assign r        = instr[15:11]                    ;
    assign v        = {{16{instr[15]}}, instr[15:0]};
    ‘include ”DCDtoyRISC.sv”
    ‘include ”PCtoyRISC.sv”
    ‘include ”RALUtoyRISC.sv”
endmodule
```

A design for a real product is designed more carefully in terms of speed. For example, there are some places where pipe registers are needed to increase the clock frequency. We are content in our approach to illustrate the processing function as an important turning point in the structural evolution of digital systems towards structure-information symbiosis. Only the functional *competence* of the mixture circuit-program is considered in our approach. The `performance` is minimally considered or completely ignored.

`DCDtoyRISC.sv` **file** is the first file that we include in the top module (see Figure **??**) describes the behavior of the decoder. It contains the 2-state interrupt automaton and the circuit which decodes the signals sent to the data memory.

```
/* *********************************************************************
File name: DCDtoyRISC.sv
                          toyRISC's DCD
********************************************************************* */
    always_ff @(posedge clock)
        if (reset)                              ei <= 0 ;
         else begin if (opCode == `eint)        ei <= 1 ;
                    if (opCode == `dint)        ei <= 0 ;
                    if (intIn & ei)             ei <= 0 ;
              end

    assign inta = intIn & ei;

    always_comb
        casex(opCode)
            `read       : read  = 1'b1              ;
            `store      : write = 1'b1              ;
            default     : {read, write} = 2'b0      ;
        endcase
```

The interrupt automaton is designed to manage the acceptance of the action of the interrupt signal. Initially, the automaton is set on the state `disable interrupt` (`ei = 0`), because the program decides when the interrupt can be accepted, not before the register `rf[31]` is loaded with the address where the program associated to the interrupt is loaded. When the interrupt is accepted (`inta = 1`), the automaton switched in the state `disable interrupt` protecting the program from the action of another interrupt before the current one does it work. At the end of the program launched by the interrupt the interrupt automaton can be switched in the `enable interrupt`.

Important note: if the `halt` instruction runs and the interrupt automaton is in the `disable state`, then the entire system is blocked and the only solution to enable its behavior is to reset it.

`PCoyRISC.sv` **file** is the second file included in the `toyRISC.sv` file. It describes a simple automaton, the automaton whose state register in the program counter, `pc`. The automaton is an initial one. It can be initialized by the `reset` signal in the state `-1` to allow the evolution immediately after the end of `reset` starting with the instruction stored ar the address `0` in the program memory. The PC automaton can also be initialized with the value stored in `rf[1]`; mechanism that allows the return of the program execution from the execution of the program associated with the interruption.

```
/* *********************************************************************
File name: PCtoyRISC.sv
                          toyRISC's PC
********************************************************************* */
    always_ff @(posedge clock) if (reset)       pc <= -1     ;
                               else if (inta) pc <= rf[31];
                                       else    pc <= nextPC;
    always_comb
```

```
        case(opCode)
          `rjmp    : nextPC = pc + v                              ;
          `zbf     : nextPC = (rf[l] == 0) ? pc + v : pc + 1      ;
          `nzbf    : nextPC = (rf[l] != 0) ? pc + v : pc + 1      ;
          `ret     : nextPC = rf[l]                               ;
          `halt    : nextPC = pc                                  ;
        default    : nextPC = pc + 1                              ;
```

Otherwise, the automaton evolves depending on the state it is in, `pc`, the command received via `opCode` and depending on the value of the left operand which is tested if it is or not zero.

`RALUtoyRISC.sv` **file** is the third file included in the `toyRISC.sv` file. It describes also a simple automaton. Its structured state, stored in a memory organized as a register file, is submitted to the processing defined as the sequence of the arithmetic and logic operations performed by a simple circuit: ALU.

The signal `inta` it also acts here, as in the case of the PC, having priority over the operation code received from the program memory: register `rf[30]` takes the return value (`pc+1`) from the program (subroutine) run as a result of the interruption.

The `always` form describes a half-automaton with the input:

```
{instr, inta, dataIn, pc}
```

and the internal state as the following Cartesian product:

```
RF = {rf[0], rf[1], ..., rf[31]}
```

```
/*************************************************************************
File name: RALUtoyRISC.sv
                          toyRISC's RALU
*************************************************************************/
    logic    [32:0] addition ;
    logic    [32:0] subtract ;
    logic    [32:0] addval   ;

    assign   addition = lrf[l]+rf[r] ;
    assign   subtract = rf[l]-rf[r]  ;
    assign   addval   = rf[l]+v      ;

    always_ff @(posedge clock)
     if (inta)          rf[30] <= pc + 1                         ;
      else
        case(opCode)
            `add    : rf[d] <= rf[l]+rf[r]                       ;
            `sub    : rf[d] <= rf[l]-rf[r]                       ;
            `addv   : rf[d] <= rf[l]+v                           ;
            `mult   : rf[d] <= rf[l]*rf[r]                       ;
            `multv  : rf[d] <= rf[l]*v                           ;
            ‘addc    : rf[d] <= addition[32]                 ;
            ‘subc    : rf[d] <= subtract[32]                 ;
            ‘addvc   : rf[d] <= addval[32]                   ;
            ‘lsh     : rf[d] <= rf[l] >> 1                   ;
            ‘ash     : rf[d] <= {rf[l][31],rf[l][31:1]}      ;
            ‘move    : rf[d] <= rf[l]                        ;
            ‘swap    : rf[d] <= {rf[l][15:0],rf[l][31:16]}   ;
            ‘bwnot   : rf[d] <= ~rf[l]                       ;
            ‘bwand   : rf[d] <= rf[l]&rf[r]                  ;
            ‘bwor    : rf[d] <= rf[l]|rf[r]                  ;
            ‘bwxor   : rf[d] <= rf[l]^rf[r]                  ;
            ‘load    : rf[d] <= dataIn                       ;
            ‘val     : rf[d] <= v                            ;
            default  : rf[0] <= rf[0]                        ;
        endcase
    assign addr      = rf[l][9:0];
    assign dataOut   = rf[r]     ;
```

The outputs associated to the half-automaton are described by the last two `assign` form which take the address and data for data memory from directly form the register file's output.

Structural description and testing

To obtain an image about how the processor toyRISC is organized we need a structural description. The micro-architecture previously defined in file `DEFINES.vh` will be implemented starting with a top module described structurally.

`toyRISC.sv` **file** is the file containing the top module for the toyRISC processor.

```
/*************************************************************************
File name: toyRISC.sv
*************************************************************************/
module toyRISC( input   logic [31:0]  instr     ,
                output  logic [9:0]   nextPC    ,
                input   logic         intIn     ,
                output  logic         inta      ,
                input   logic [31:0]  dataIn    ,
                output  logic [31:0]  dataOut   ,
                output  logic [9:0]   addr      ,
                output  logic         dataRead  ,
                output  logic         dataWrite ,
                input   logic         reset     ,
                input   logic         clk       );
    logic   [5:0]   opCode  ;
    logic   [4:0]   d, l, r ; // dest, left, right addresses for rf
    logic   [31:0]  v       ; // immediate value
    logic   [31:0]  leftOp  ;
    logic    [9:0]    pc        ;
    logic             we        ;
    logic    [1:0]    sel       ;

    assign opCode    = instr[31:26]                  ;
    assign d         = instr[25:21]                  ;
    assign l         = instr[20:16]                  ;
    assign r         = instr[15:11]                  ;
    assign v         = {{16{instr[15]}}, instr[15:0]};

    DCDtoyRISC   DCD(opCode        ,
                     intIn         ,
                     inta          ,
                     dataRead      ,
                     dataWrite     ,
                     we            ,
                     sel           ,
                     reset         ,
                     clk           );
    PCtoyRISC    PC( nextPC   ,
                     pc       ,
                     leftOp   ,
                     v[9:0]   ,
                     opCode   ,
                     inta     ,
                     reset    ,
                     clk      );
    RALUtoyRISC RALU(    dataOut  ,
                         leftOp   ,
                         pc       ,
                         opCode   ,
                         dataIn   ,
                         v        ,
                         l        ,
                         r        ,
                         d        ,
                         sel      ,
                         we       ,
                         inta     ,
                         clk      );
    assign addr = leftOp[15:0]   ;
endmodule // Synthesis results: #LUT=455, #FF=11, #DSP=6
```

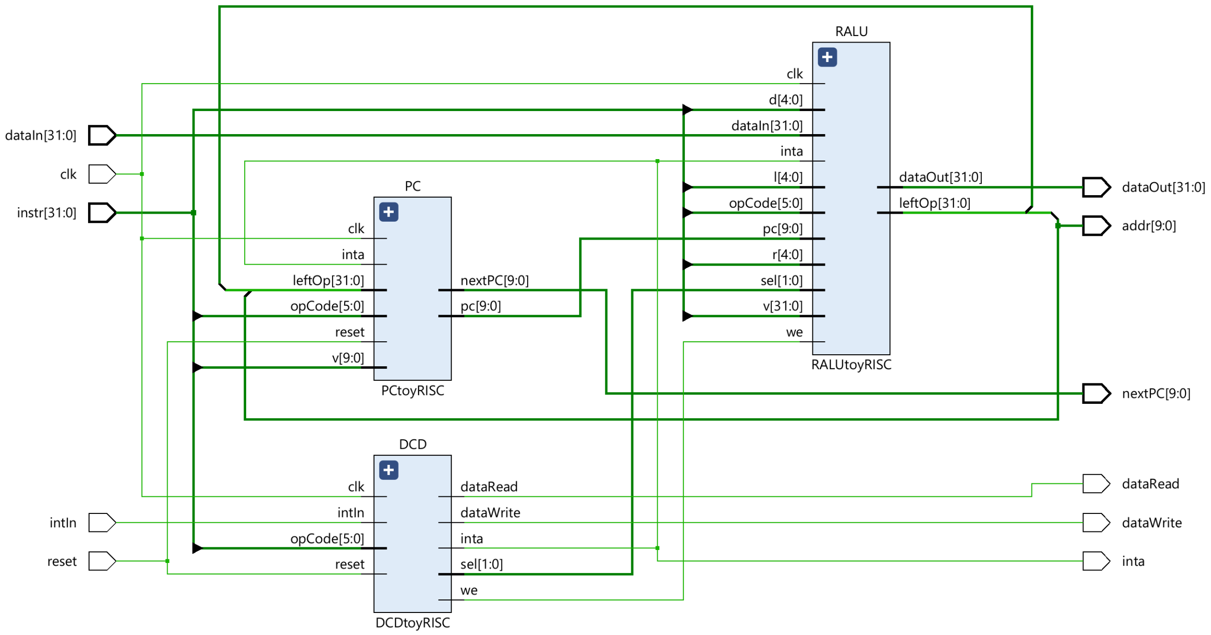


Figure 5.6: Top level of the toyRISC processor.

`DCDtoyRISC.sv` **file** is the file describing the decoder module of the toyRISC processor.

```
/*************************************************************************
File name: DCDtoyRISC.sv
                          toyRISC's DCD
*************************************************************************/
‘include ”DEFINES.vh”
module  DCDtoyRISC( input   logic   [5:0]   opCode      ,
                    input   logic           intIn       ,
                    output  logic           inta        ,
                                            dataRead    ,
                                            dataWrite   ,
                                            we          ,
                    output  logic   [1:0]   sel         ,
                    input   logic           reset       ,
                                            clk         );
    // Interrupt automaton
    logic   ei  ; // enable interrupt = state resgister
    always_ff @(posedge clk) if (reset)                          ei <= 0;
                        else begin  if (opCode == ‘eint)   ei <= 1;
                                    if (opCode == ‘dint)   ei <= 0;
                                    if (intIn & ei)        ei <= 0;
                                end
    assign inta = intIn & ei;
    // Decoding
    assign dataRead  = (opCode == ‘read)    ? 1’b1 : 1’b0    ;
    assign dataWrite = (opCode == ‘store)   ? 1’b1 : 1’b0    ;
    assign we        = inta | (opCode[5:4] == 2’b11) |
                              (opCode[5:4] == 2’b01) |
                              (opCode == 6’b10_0111)         ;

    always_comb if (inta)                          sel = 2’b00 ;
                else if (opCode == ‘load)          sel = 2’b10 ;
                        else if (opCode == ‘val)   sel = 2’b01 ;
                                else               sel = 2’b11 ;

endmodule
```

`PCtoyRISC.sv` **file** describes the program counter section of the toyRISC processor.

```
/* ***********************************************************************
File name: PCtoyRISC.sv
*********************************************************************** */
module PCtoyRISC(   output  logic   [9:0]   nextPC  ,
                    output  logic   [9:0]   pc      ,
                    input   logic   [31:0]  leftOp  ,
                    input   logic   [9:0]   v       ,
                    input   logic   [5:0]   opCode  ,
                    input   logic           inta    ,
                                            reset   ,
                                            clk     );

    always_ff @(posedge clk) if (reset)         pc <= -1            ;
                             else if (inta)     pc <= leftOp[9:0]   ;
                                  else          pc <= nextPC        ;
    always_comb case(opCode)
                    ‘rjmp   : nextPC = pc + v                        ;
                    ‘zbr    : nextPC = (leftOp == 0) ? pc + v : pc + 1;
                    ‘nzbr   : nextPC = (leftOp != 0) ? pc + v : pc + 1;
                    ‘ret    : nextPC = leftOp                        ;
                    ‘halt   : nextPC = pc                            ;
                    default : nextPC = pc + 1                        ;
                endcase
endmodule
```

`RALUtoyRISC.sv` **file** describes the RALU unit of the toyRISC processor.

```
/**********************************************************************
File name: RALUtoyRISC.sv
**********************************************************************/
module RALUtoyRISC( output  logic   [31:0]  dataOut ,
                    output  logic   [31:0]  leftOp  ,
                    input   logic   [9:0]   pc      ,
                    input   logic   [5:0]   opCode  ,
                    input   logic   [31:0]  dataIn  ,
                                            v       ,
                    input   logic   [4:0]   l       ,
                                            r       ,
                                            d       ,
                    input   logic   [1:0]   sel     ,
                    input   logic           we      ,
                                            inta    ,
                                            clk     );
    logic   [31:0]  leftIn  ;
    logic   [31:0]  rightIn ;
    logic   [31:0]  muxOut  ;
    logic   [31:0]  aluOut  ;

    registerFile    regFile(.leftOp     (leftIn  ),
                            .rightOp    (rightIn ),
                            .in         (muxOut  ),
                            .leftAddr   (l       ),
                            .rightAddr  (r       ),
                            .destAddr   (d       ),
                            .we         (we      ),
                            .inta       (inta    ),
                            .opCode     (opCode  ),
                            .clk        (clk     ));

    assign  dataOut  = rightIn   ;
    assign  leftOp   = leftIn    ;

    ALU alu(.out     (aluOut),
            .leftIn  (leftIn ),
            .rightIn (rightIn),
            .value   (v      ),
            .opCode  (opCode ));

    bigMux  bigMux(  .aluOut  (aluOut),
                     .dataIn  (dataIn),
                     .value   (v      ),
                     .pc      (pc     ),
                     .sel     (sel    ),
                     .muxOut  (muxOut ));
endmodule
```

`registerFile.sv` **file** describes the register file from RALU unit.

```
/***********************************************************************
File name: registerFile.sv
***********************************************************************/
module registerFile(output  logic   [31:0]  leftOp      ,
                                            rightOp     ,
                    input   logic   [31:0]  in          ,
                    input   logic   [4:0]   leftAddr    ,
                                            rightAddr   ,
                                            destAddr    ,
                    input   logic           we          ,
                    input   logic           inta        ,
                    input   logic   [5:0]   opCode      ,
                    input   logic           clk         );
    logic   [31:0]  rf[0:31];

    always_ff @(posedge clk) if (we) rf[inta ? 5'b11110 : destAddr] <= in
;

    assign leftOp   = rf[inta ? 5'b11111 : leftAddr];
    assign rightOp  = rf[rightAddr]                 ;
endmodule
```

`ALU.sv` **file** describes the ALU unit from RALU.

```
/***********************************************************************
File name: ALU.sv
***********************************************************************/
`include "DEFINES.vh"
module ALU( output  logic   [31:0]  out     ,
            input   logic   [31:0]  leftIn  ,
                                    rightIn ,
                                    value   ,
            input   logic   [5:0]   opCode  );

    logic   [32:0]  addition ;
    logic   [32:0]  subtract ;
    logic   [32:0]  addval   ;

    assign  addition = leftIn + rightIn ;
    assign  subtract = leftIn - rightIn ;
    assign  addval   = leftIn + value   ;

    always_comb
        case(opCode)
            `add    : out = addition[31:0]                  ;
            `sub    : out = subtract[31:0]                  ;
            `addv   : out = addval[31:0]                    ;
            `mult   : out = leftIn * rightIn                ;
            `multv  : out = leftIn * value                  ;
            `addc   : out = addition[32]                    ;
            `subc   : out = subtract[32]                    ;
            `addvc  : out = addval[32]                      ;
            `lsh    : out = leftIn >> 1                     ;
            `ash    : out = {leftIn[31], leftIn[31:1]}      ;
            `move   : out = leftIn                          ;
            `swap   : out = {leftIn[15:0], leftIn[31:16]}   ;
            `bwnot  : out = ~leftIn                         ;
            `bwand  : out = leftIn & rightIn                ;
            `bwor   : out = leftIn | rightIn                ;
            `bwxor  : out = leftIn ^ rightIn                ;
            `rotate : out = {leftIn[0], leftIn[31:1]}       ;
            default : out = leftIn                          ;
        endcase
endmodule
```

`bigMux.sv` **file** describes the multiplexor from the input of the register file.

```
/***********************************************************************
File name: bigMux.sv
***********************************************************************/
module bigMux(  input   logic   [31:0]  aluOut  ,
                                        dataIn  ,
                                        value   ,
                input   logic   [9:0]   pc      ,
                input   logic   [1:0]   sel     ,
                output  logic   [31:0]  muxOut  );

    always_comb case(sel)
                    2'b00: muxOut = pc      ;
                    2'b01: muxOut = value   ;
                    2'b10: muxOut = dataIn  ;
                    2'b11: muxOut = aluOut  ;
                endcase
endmodule
```

RISCarch.sv file contains the mnemonics used in the definition of the assembly language for the toyRISC processor.

```
/***********************************************************************
```

```
File name: RISCarch.sv
************************************************************************/
    NOP           // no operation
    RJMP(lb)      // relative jumpto label 'lb'
    BRZ(l,lb)     // branch if rf[l]=zero at label 'lb'
    BRNZ(l,lb)    // branch if rf[l]!=zero at label 'lb'
    RET(l)        // return from subroutine: pc<=rf[l]
    HALT          // halt until interrupt is received, pc = pc
 // for the following instructions: pc<=pc+1;
    EINT          // set enable interrupt
    DINT          // set disable interrupt
    ADD(d,l,r)    // rf[d]<=rf[l]+rf[r];
    SUB(d,l,r)    // rf[d]<=rf[l]-rf[r];
    ADDV(d,l,v)   // rf[d]<=rf[l]+v;
    MULT(d,l,r)   // rf[d]<=rf[l]*rf[r];
    MULTV(d,l,v)  // rf[d]<=rf[l]*v;
    ADDC(d,l,r)   // rf[d]<=(rf[l]+rf[r]}[32];
    SUBC(d,l,r)   // rf[d]<=(rf[l]-rf[r])[32];
    ADDVC(d,l,v)  // rf[d]<=(rf[l]+v)[32];
    LSH(d,l)      // rf[d]<=rf[l] >> 1;
    ASH(d,l)      // rf[d]<={rf[l][31], rf[l][31:1]};
    MOVE(d,l)     // rf[d]<=rf[l];
    SWAP(d,l)     // rf[d]<={rf[l][15:0], rf[l][31:16]};
    NOT(d,l)      // rf[d]<=~rf[l];
    AND(d,l,r)    // rf[d]<=rf[l]&rf[r];
    OR(d,l,r)     // rf[d]<=rf[l]|rf[r];
    XOR(d,l,r)    // rf[d]<=rf[l]^rf[r];
    READ(l)       // read from dataMemory[rf[l]];
    LOAD(d)       // rf[d]<=dataOut;
    STORE(l,r)    // dataMemory[rf[l]]<=rf[r];
    VAL(d,v)      // rf[d]<={{16*{v[15]}},v};
```

`testRISC.sv` **file** The test module, which is is the top module of the project, includes the two memories needed for the simulation: the data memory and program memory. The content of the program memory is generated using the module `RISCcodeGenerator` which is also included in the next module.

```
/************************************************************************
File name: testRISC.sv
************************************************************************/
module testRISC;
    logic            inta     ;
    logic            intIn    ;
    logic            reset    ;
    logic            clk      ;
    logic   [31:0]   instr        ;
    logic   [9:0]    nextPC       ;
    logic   [31:0]   dataIn       ;
    logic   [31:0]  dataOut         ;
    logic   [9:0]   addr            ;
    logic           dataRead        ;
    logic           dataWrite       ;
    logic   [31:0]  dataMemory[0:1023]  ;
    logic   [31:0]  progMemory[0:1023]  ;
    logic   [31:0]  dataMemOut          ;

    integer i   ;

    `include "RISCcodeGenerator.sv"

    initial begin                   clk = 0     ;
                    forever #1      clk = ~clk  ;
            end

    initial begin
            intIn       = 1 ;
            reset       = 1 ;
            // DISPLAY THE CONTENT OF PROGRAM MEMORY
            for (i=0; i<16; i=i+1)
                $display("progMemory[%0d]␣\t␣=␣%b", i,
                         progMemory[i]) ;
        #4  reset       = 0 ;
        #40 $finish         ;
    end

    always_ff @(posedge clk) begin
        if (dataRead) dataIn <= dataMemory[addr]        ;
        if (dataWrite) dataMemory[addr] <= dataOut      ;
        instr <= progMemory[nextPC]                     ;
    end

    toyRISC dut(instr       ,
                nextPC      ,
                intIn       ,
                inta        ,
                dataIn      ,
                dataOut     ,
                addr        ,
                dataRead    ,
                dataWrite   ,
                reset       ,
                clk         );

    // MONITOR FOR PROGRAM LAOD & CONTROLLER
    initial begin
        $monitor("t=%0d␣pc=%d␣␣RF=[%0d,␣%0d,␣%0d,␣%0d]␣ei=%b␣inta=%b",
                $time,
                dut.PC.pc,
                  dut.RALU.regFile.rf[0],
                  dut.RALU.regFile.rf[1],
                  dut.RALU.regFile.rf[2],
                  dut.RALU.regFile.rf[3],
                  dut.DCD.ei,
                  dut.inta);
    end
endmodule
```

`RISCcodeGenerator.sv` **file** describes the code generator used to translate the program written in assembly into the executable code for toyRISC processor.

```
/************************************************************************
File name: RISCcodeGenerator.sv
************************************************************************/
    reg [5:0]   opCode          ;
    reg [4:0]   d               ;
    reg [4:0]   l               ;
    reg [4:0]   r               ;
    reg [15:0]  v               ;
    reg [9:0]   addrCounter     ;
    reg [9:0]   labelTab[0:1023];

    `include "DEFINES.vh"

    task endLine;
     begin
        progMemory[addrCounter][31:0] =
            {   opCode   ,
                d        ,
                l        ,
                v        }                  ;
            addrCounter = addrCounter + 1   ;
     end
    endtask

    // sets labelTab in the first pass
    // associating 'counter' with 'labelIndex'
    task LB ;
        input [5:0] labelIndex;

        labelTab[labelIndex] = addrCounter;
    endtask
    // uses the content of labelTab in the second pass
    task ULB;
        input [5:0] labelIndex;
        v = labelTab[labelIndex] - addrCounter;
    endtask

// CONTROL INSTRUCTIONS
    task NOP; // no operation
        begin   opCode  = `addv ;
                d       = 5'b0  ;
                l       = 5'b0  ;
                v       = 16'b0 ;
                endLine         ;
        end
    endtask

    task RJMP; // relative jump
        input   [15:0]  label   ;

        begin   opCode  = `rjmp ;
                d       = 5'b0  ;
                l       = 5'b0  ;
                ULB(label)      ;
                endLine         ;
        end
    endtask

    task BRZ; // branch if zero
        input   [4:0]   left    ;
        input   [9:0]   label   ;

        begin   opCode  = `zbr  ;
                d       = 5'b0  ;
                l       = left  ;
                ULB(label)      ;
                endLine         ;
        end
    endtask

    task BRNZ; // branch if not zero
        input   [4:0]   left    ;
        input   [9:0]   label   ;

        begin   opCode  = `nzbr ;
                d       = 5'b0  ;
                l       = left  ;
                ULB(label)      ;
                endLine         ;
        end
    endtask

    task RET; // return from subroutine
        input   [4:0] left  ;
        begin   opCode  = ‘ret   ;
                d       = 5’b0   ;
                l       = left   ;
                v       = 16’b0  ;
                endLine          ;
        end
    endtask

    task HALT; // halt running
        begin   opCode  = ‘halt  ;
                d       = 5’b0   ;
                l       = 5’b0   ;
                v       = 16’b0  ;
                endLine          ;
        end
    endtask

    task EI; // enable interrupt
        begin   opCode  = ‘eint  ;
                d       = 5’b0   ;
                l       = 5’b0   ;
                v       = 16’b0  ;
                endLine          ;
        end
    endtask

    task DI; // disable interrupt
        begin   opCode  = ‘dint  ;
                d       = 5’b0   ;
                l       = 5’b0   ;
                v       = 16’b0  ;
                endLine          ;
        end
    endtask

// ARITHMETIC & LOGIC INSTRUCTIONS
    task ADD; // addition
        input   [4:0]   dest     ;
        input   [4:0]   left     ;
        input   [4:0]   right    ;

        begin   opCode  = ‘add              ;
                d       = dest              ;
                l       = left              ;
                v       = {right , 11’b0};
                endLine                     ;
        end
    endtask
```

```
task SUB; // subtract
    input   [4:0]   dest    ;
    input   [4:0]   left    ;
    input   [4:0]   right   ;

    begin   opCode  = `sub              ;
            d       = dest              ;
            l       = left              ;
            v       = {right, 11'b0};
            endLine                     ;
    end
endtask

task ADDV; // addition with value
    input   [4:0]   dest    ;
    input   [4:0]   left    ;
    input   [15:0]  value   ;

    begin   opCode  = `addv ;
            d       = dest  ;
            l       = left  ;
            v       = value ;
            endLine         ;
    end
endtask

task MULT; // multiplication
    input   [4:0]   dest    ;
    input   [4:0]   left    ;
    input   [4:0]   right   ;

    begin   opCode  = `mult             ;
            d       = dest              ;
            l       = left              ;
            v       = {right, 11'b0};
            endLine                     ;
    end
endtask

task MULTV; // multiplication with value
    input   [4:0]   dest    ;
    input   [4:0]   left    ;
    input   [15:0]  value   ;

    begin   opCode  = `multv;
            d       = dest  ;
            l       = left  ;
            v       = value ;
            endLine         ;
    end
```

```
endtask

task ADDC; // carry from addition
    input   [4:0]   dest    ;
    input   [4:0]   left    ;
    input   [4:0]   right   ;

    begin   opCode  = `addc         ;
            d       = dest          ;
            l       = left          ;
            v       = {right, 11'b0};
            endLine                 ;
    end
endtask

task SUBC; // carry from subtract
    input   [4:0]   dest    ;
    input   [4:0]   left    ;
    input   [4:0]   right   ;

    begin   opCode  = `subc         ;
            d       = dest          ;
            l       = left          ;
            v       = {right, 11'b0};
            endLine                 ;
    end
endtask

task ADDVC; // carry from addition with value
    input   [4:0]   dest    ;
    input   [4:0]   left    ;
    input   [15:0]  value   ;

    begin   opCode  = `addvc;
            d       = dest  ;
            l       = left  ;
            v       = value ;
            endLine         ;
    end
endtask

task LSH; // logic shift with one position
    input   [4:0]   dest    ;
    input   [4:0]   left    ;

    begin   opCode  = `lsh  ;
            d       = dest  ;
            l       = left  ;
            v       = 16'b0 ;
            endLine         ;
```

```
        end
    endtask

    task ASH; // arithmetic shift with one porition
        input   [4:0]   dest    ;
        input   [4:0]   left    ;

        begin   opCode  = `ash   ;
                d       = dest   ;
                l       = left   ;
                v       = 16'b0  ;
                endLine          ;
        end
    endtask

    task MOVE; // data move inside register file
        input   [4:0]   dest    ;
        input   [4:0]   left    ;

        begin   opCode  = `move  ;
                d       = dest   ;
                l       = left   ;
                v       = 16'b0  ;
                endLine          ;
        end
    endtask

    task SWAP; // swap in register
        input   [4:0]   dest    ;
        input   [4:0]   left    ;

        begin   opCode  = `swap  ;
                d       = dest   ;
                l       = left   ;
                v       = 16'b0  ;
                endLine          ;
        end
    endtask

    task NOT; // bitwise NOT
        input   [4:0]   dest    ;
        input   [4:0]   left    ;

        begin   opCode  = `bwnot ;
                d       = dest   ;
                l       = left   ;
                v       = 16'b0  ;
                endLine          ;
        end
    endtask
    task AND; // bitwise AND
        input   [4:0]   dest    ;
        input   [4:0]   left    ;
        input   [4:0]   right   ;

        begin   opCode  = 'bwand          ;
                d       = dest            ;
                l       = left            ;
                v       = {right, 11'b0};
                endLine                   ;
        end
    endtask

    task OR; // bitwise OR
        input   [4:0]   dest    ;
        input   [4:0]   left    ;
        input   [4:0]   right   ;

        begin   opCode  = 'bwor       ;
                d       = dest            ;
                l       = left            ;
                v       = {right, 11'b0};
                endLine                   ;
        end
    endtask

    task XOR; // bitwise XOR
        input   [4:0]   dest    ;
        input   [4:0]   left    ;
        input   [4:0]   right   ;

        begin   opCode  = 'bwxor          ;
                d       = dest            ;
                l       = left            ;
                v       = {right, 11'b0};
                endLine                   ;
        end
    endtask

// DATA TRANSFER INSTRUCTIONS
    task READ; // data read
        input   [4:0]   left    ;

        begin   opCode  = 'read ;
                d       = 5'b0  ;
                l       = left  ;
                v       = 16'b0 ;
                endLine         ;
        end
    endtask

    task LOAD; // data load
        input   [4:0]   dest    ;

        begin   opCode  = ‘load ;
                d       = dest  ;
                l       = 5’b0  ;
                v       = 16’b0 ;
                endLine         ;
        end
    endtask

    task STORE; // data store
        input   [4:0]   left    ;
        input   [4:0]   right   ;

        begin   opCode  = ‘store            ;
                d       = 5’b0              ;
                l       = left              ;
                v       = {right, 11’b0};
                endLine                     ;
        end
    endtask

    task VAL; // value load
        input   [4:0]   dest    ;
        input   [15:0]  value   ;

        begin   opCode  = ‘val  ;
                d       = dest  ;
                l       = 5’b0  ;
                v       = value ;
                endLine         ;
        end
    endtask

    // RUNNING
    initial begin   addrCounter = 0;
                    ‘include ”program.sv” // first pass
                    addrCounter = 0;
                    ‘include ”program.sv” // second pass
            end
```

Example 5.2 *Registers from 0 to 4 are loaded with values 1 to 5 then in* `rf[0]` *all the values are added.*

```
/* ******************************************************************
File name: program.sv
```

```
*************************************************************************/
            VAL(0,1)      ;
            VAL(1,2)      ;
            VAL(2,3)      ;
            VAL(3,4)      ;
            VAL(4,5)      ;
            ADD(0,0,1)    ;
            ADD(0,0,2)    ;
            ADD(0,0,3)    ;
            ADD(0,0,4)    ;
            HALT          ;
```

The monitor provides the following result of the running:

```
t=0  pc=   x aluOut=  x  RF=[x,  x, x, x, x] ei=x inta=x
t=1  pc=1023 aluOut=  x  RF=[x,  x, x, x, x] ei=0 inta=0
t=5  pc=   0 aluOut=  1  RF=[1,  x, x, x, x] ei=0 inta=0
t=7  pc=   1 aluOut=  1  RF=[1,  x, x, x, x] ei=0 inta=0
t=9  pc=   2 aluOut=  1  RF=[1,  2, x, x, x] ei=0 inta=0
t=11 pc=   3 aluOut=  1  RF=[1,  2, 3, x, x] ei=0 inta=0
t=13 pc=   4 aluOut=  1  RF=[1,  2, 3, 4, x] ei=0 inta=0
t=15 pc=   5 aluOut=  3  RF=[1,  2, 3, 4, 5] ei=0 inta=0
t=17 pc=   6 aluOut=  6  RF=[3,  2, 3, 4, 5] ei=0 inta=0
t=19 pc=   7 aluOut= 10  RF=[6,  2, 3, 4, 5] ei=0 inta=0
t=21 pc=   8 aluOut= 15  RF=[10, 2, 3, 4, 5] ei=0 inta=0
t=23 pc=   9 aluOut= 15  RF=[15, 2, 3, 4, 5] ei=0 inta=0
```

◇

Example 5.3 *The memory operations are exemplified by the following program:*

```
/*************************************************************************
File name: program.sv
*************************************************************************/
            VAL(0,1)      ;
            VAL(1,55)     ;
            STORE(0,1)    ;
            READ(0)       ;
            LOAD(2)       ;
            HALT          ;
```

The monitor provides the following result of the running:

```
t=0  pc=   x aluOut= x  RF=[x, x,  x,  x, x] ei=x inta=x
t=1  pc=1023 aluOut= x  RF=[x, x,  x,  x, x] ei=0 inta=0
t=5  pc=   0 aluOut= 1  RF=[1, x,  x,  x, x] ei=0 inta=0
```

```
t=7  pc=   1 aluOut= 1  RF=[1, x,  x,  x, x] ei=0 inta=0
t=9  pc=   2 aluOut= 1  RF=[1, 55, x,  x, x] ei=0 inta=0
t=11 pc=   3 aluOut= 1  RF=[1, 55, x,  x, x] ei=0 inta=0
t=13 pc=   4 aluOut= 1  RF=[1, 55, x,  x, x] ei=0 inta=0
t=15 pc=   5 aluOut= 1  RF=[1, 55, 55, x, x] ei=0 inta=0
```

◇

Example 5.4 *The interrupt works as in the following program:*

```
/**********************************************************************
File name: program.sv
**********************************************************************/
            VAL(31,10)    ;
            VAL(2,23)     ;
            VAL(0,13)     ;
            EI            ;
            ADDV(0,0,2)   ;
            NOP           ;
            ADDV(0,0,4)   ;
            HALT          ;
            //NOP              ;
            NOP           ;
            NOP           ;
        // subroutine triggered by interrupt
            DI            ;
            VAL(3,44)     ;
            RET(30)       ;
```

```
t=0  pc=   x aluOut=  x  RF=[x,  x, x,  x,  x] ei=x inta=x
t=1  pc=1023 aluOut=  x  RF=[x,  x, x,  x,  x] ei=0 inta=0
t=5  pc=   0 aluOut=  x  RF=[x,  x, x,  x,  x] ei=0 inta=0
t=7  pc=   1 aluOut=  x  RF=[x,  x, x,  x,  x] ei=0 inta=0
t=9  pc=   2 aluOut=  x  RF=[x,  x, 23, x,  x] ei=0 inta=0
t=11 pc=   3 aluOut= 13  RF=[13, x, 23, x,  x] ei=0 inta=0
t=13 pc=   4 aluOut= 12  RF=[13, x, 23, x,  x] ei=1 inta=1
t=15 pc=  10 aluOut= 13  RF=[13, x, 23, x,  x] ei=0 inta=0
t=17 pc=  11 aluOut= 13  RF=[13, x, 23, x,  x] ei=0 inta=0
t=19 pc=  12 aluOut=  4  RF=[13, x, 23, 44, x] ei=0 inta=0
t=21 pc=   4 aluOut= 15  RF=[13, x, 23, 44, x] ei=0 inta=0
t=23 pc=   5 aluOut= 15  RF=[15, x, 23, 44, x] ei=0 inta=0
t=25 pc=   6 aluOut= 19  RF=[15, x, 23, 44, x] ei=0 inta=0
t=27 pc=   7 aluOut= 19  RF=[19, x, 23, 44, x] ei=0 inta=0
```

◇

Example 5.5 *The conditional branch is exemplified by the following program:*

```
/************************************************************************
File name: program.sv
************************************************************************/
            VAL(0,3)      ;
    LB(1);  ADDV(0,0,-1);
            NOP           ;
            BRNZ(0,1)     ;
            HALT          ;
```

```
t=0  pc=   x aluOut= x  RF=[x, x, x, x, x] ei=x inta=x
t=1  pc=1023 aluOut= x  RF=[x, x, x, x, x] ei=0 inta=0
t=5  pc=   0 aluOut= 3  RF=[3, x, x, x, x] ei=0 inta=0
t=7  pc=   1 aluOut= 2  RF=[3, x, x, x, x] ei=0 inta=0
t=9  pc=   2 aluOut= 2  RF=[2, x, x, x, x] ei=0 inta=0
t=11 pc=   3 aluOut= 2  RF=[2, x, x, x, x] ei=0 inta=0
t=13 pc=   1 aluOut= 1  RF=[2, x, x, x, x] ei=0 inta=0
t=15 pc=   2 aluOut= 1  RF=[1, x, x, x, x] ei=0 inta=0
t=17 pc=   3 aluOut= 1  RF=[1, x, x, x, x] ei=0 inta=0
t=19 pc=   1 aluOut= 0  RF=[1, x, x, x, x] ei=0 inta=0
t=21 pc=   2 aluOut= 0  RF=[0, x, x, x, x] ei=0 inta=0
t=23 pc=   3 aluOut= 0  RF=[0, x, x, x, x] ei=0 inta=0
t=25 pc=   4 aluOut= 0  RF=[0, x, x, x, x] ei=0 inta=0
```

◇

Example 5.6 *Adding of two 64-bit numbers is illustrated by the following two programs:*

```
/************************************************************************
File name: program.sv
************************************************************************/
            VAL(0,-3);
            VAL(1,5);
            VAL(2,6);
            VAL(3,7);
    //: {rf[1],rf[0]} <= {rf[1],rf[0]} + {rf[3],rf[2]}
            ADDC(4,0,2);
            ADD(0,0,2);
            ADD(1,1,3);
            ADD(1,1,4);
            HALT;
```

```
t=0  pc=   x aluOut=         x  RF=[x,          x,  x, x, x] ei=x inta=x
t=1  pc=1023 aluOut=         x  RF=[x,          x,  x, x, x] ei=0 inta=0
t=5  pc=   0 aluOut=4294967293  RF=[4294967293, x,  x, x, x] ei=0 inta=0
```

```
t=7  pc=   1 aluOut=4294967293  RF=[4294967293, x,  x, x, x] ei=0 inta=0
t=9  pc=   2 aluOut=4294967293  RF=[4294967293, 5,  x, x, x] ei=0 inta=0
t=11 pc=   3 aluOut=4294967293  RF=[4294967293, 5,  6, x, x] ei=0 inta=0
t=13 pc=   4 aluOut=         1  RF=[4294967293, 5,  6, 7, x] ei=0 inta=0
t=15 pc=   5 aluOut=         3  RF=[4294967293, 5,  6, 7, 1] ei=0 inta=0
t=17 pc=   6 aluOut=        12  RF=[3,          5,  6, 7, 1] ei=0 inta=0
t=19 pc=   7 aluOut=        13  RF=[3,          12, 6, 7, 1] ei=0 inta=0
t=21 pc=   8 aluOut=         3  RF=[3,          13, 6, 7, 1] ei=0 inta=0
```

```
/*****************************************************************************
File name: program.sv
*****************************************************************************/
            VAL(0,3);
            VAL(1,5);
            VAL(2,6);
            VAL(3,7);
    //: {rf[1],rf[0]} <= {rf[1],rf[0]} + {rf[3],rf[2]}
            ADDC(4,0,2);
            ADD(0,0,2);
            ADD(1,1,3);
            ADD(1,1,4);
            HALT;
```

```
t=0  pc=   x aluOut=         x  RF=[x,  x, x, x, x] ei=x inta=x
t=1  pc=1023 aluOut=         x  RF=[x,  x, x, x, x] ei=0 inta=0
t=5  pc=   0 aluOut=         3  RF=[3,  x, x, x, x] ei=0 inta=0
t=7  pc=   1 aluOut=         3  RF=[3,  x, x, x, x] ei=0 inta=0
t=9  pc=   2 aluOut=         3  RF=[3,  5, x, x, x] ei=0 inta=0
t=11 pc=   3 aluOut=         3  RF=[3,  5, 6, x, x] ei=0 inta=0
t=13 pc=   4 aluOut=         0  RF=[3,  5, 6, 7, x] ei=0 inta=0
t=15 pc=   5 aluOut=         9  RF=[3,  5, 6, 7, 0] ei=0 inta=0
t=17 pc=   6 aluOut=        12  RF=[9,  5, 6, 7, 0] ei=0 inta=0
t=19 pc=   7 aluOut=        12  RF=[9, 12, 6, 7, 0] ei=0 inta=0
t=21 pc=   8 aluOut=         9  RF=[9, 12, 6, 7, 0] ei=0 inta=0
```

◇

5.5 Concluding about the third loop

The third loop is closed through simple automata avoiding the fast increasing of the complexity in digital circuit domain. It allows the autonomy of the control mechanism.

”Intelligent registers” ask less structural control maintaining the complexity of a finite automaton at the smallest possible level. Intelligent, loop driven circuits can be controlled using smaller complex circuits.

The loop through a storage element ask less symbolic control at the micro-architectural level. Less symbols are used to determine the same behavior because the local loop through a memory element generates additional information about the recent history.

Looping through a memory circuit allows a more complex "understanding" because the controlled circuits "knows" more about its behavior in the previous clock cycle. The circuit is somehow "conscious" about what it did before, thus being more "responsible" for the operation it performs now.

Looping through an automaton allows any effective computation. Using the theory of computation (see chapter *Recursive Functions & Loops* in this book) can be proved that any effective computation can be done using a three loop digital system. More than three loops are needed only for improving the efficiency of the computational structures.

The third loop allows the symbolic functional control using the arbitrary meaning associated to the binary codes embodied in instructions or micro-instructions. Both, the coding and the decoding process being controlled at the design level, the binary symbols act actualizing the potential structure of a programmable machine.

Real processors use circuit level parallelism discussed in the first chapter of this book. They are: data parallelism, time parallelism and speculative parallelism. How all these kind of parallelism are used is a computer architecture topic, beyond the goal of these lecture notes.

5.6 Problems

Problem 5.1 *Let be system having two functionalities:*

1. *when the temperatire sensor detects a temperature under the value "1101" the hitting system is activated*
2. *at each 30 seconds the value of the temperature is stored in a memory having 16kB.*

Design the block schematic and implement it in System Verilog with the following interface:

```
input   logic           clock,
input   logic           reset,
input   logic [7:0]     digitalTempSensor,
input   logic [13:0]    memoryReadAddress,
output  logic           activateHittingSystem,
output  logic [7:0]     memoryOutput,
```

Clock frequency: 1 MHz.

Problem 5.2 *It is desired to install a smart light bulb on the staircase of the block so that it turns on when the main switch is open or when it receives a signal from the motion sensor in the staircase. If it receives a signal from the sensor, the light bulb must remain on for 30s. The sensor emits a signal lasting one clock cycle.*

Design a system with a controller that can help solve this situation. It is requested the complete system diagram (logic gates / blocks), System Verilog code with the following interface:

```
input   logic   clock,
input   logic   reset,
input   logic   mainSwitch,
input   logic   motionSensor,
output  logic   bulb,
```

Clock frequency: 1 KHz.

Problem 5.3 *Design and simulate a system that receives a string of symbols belonging to the set* {`a,b,c`}*. The system, after applying the reset signal, continuously displays on three outputs the number of a's, b's and c's received, until one of the outputs reaches the value 255, when the counting stops and is restarted at the next reset. The inputs are coded as follows:* `a = 00, b = 01, c = 10`.

1. *Draw the block diagram of the system*
2. *Describe it in System Verilog*
3. *Simulate the system.*

Problem 5.4 *Design a system that upon reset goes to the initial state with the output Y=0000, a state in which it stays as long as it accumulates, starting from 0, the input value x[7:0] in each cycle. When the accumulated value exceeds 1024, then it generates the sequence on the output:*

```
Y=0100
Y=0110
Y=1000
Y=1001
```

and returns to the initial state.

Problem 5.5 *Design a system that after reset generates:*

1. *001 for 3 clock cycles*
2. *010 for 2 clock cycles*
3. *100 for 7 clock cycles*
4. *000 until the reset signal is activated again.*

Problem 5.6 *Design and simulate a system that receives as input strings of symbols from the set* {`a, b, e`} *and recognizes strings of the form*

$$\ldots eea^n b^3 a^n bee \ldots$$

for $n < 32$.

Problem 5.7 *Consider the system defined by:*

```
module syst(input   logic [7:0] in0, in1, in2, in3,
            output  logic       out0, out1, out2, out3,
            input   logic       reset, clock);

...
endmodule
```

which generates in the order out0, out1, out2, out3 pulses of durations in0, in1, in2, in3, where the 8-bit positive integer values represented the number of clock cycles that the corresponding outputs are active on 1. Required:

1. *Drawing the internal block diagram of the system*
2. *Verilog description of the system*
3. *Simulation of the operation.*

Problem 5.8 *Design a system that after reset generates:*

1. *001 for 5 clock cycles*
2. *011 for 2 clock cycles*
3. *100 for 7 clock cycles ı.tem 000 until the reset signal is activated again*

Problem 5.9 *Design and simulate a system that receives a string of symbols belonging to the set* {a,b,c}. *The system, after applying the reset signal, counts the occurrences of the symbol* a *in the first 112 symbols received, after which it passes into an inactive (idle) state from which it exits only by reset.*

The inputs are coded as follows: a = 00, b = 01, c = 10.

The output signals the counting state by 0 and the idle state by 1.

Problem 5.10 *Download the* toyRISCgeneric.zip *archive from* https://users.dcae.pub.ro/~gstefan/2ndLevel/digital_circuits.html. *Unzip and open the project it contains. Add the* left rotate *instruction to the instruction set:*

```
‘define lrot 6’b01_1001 // rf[d] <= {rf[l][30:0], rf[l][31] }
```

and make the necessary changes to the project. Write a simple program to test the correct operation of the new instruction.

Problem 5.11 *Download the* toyRISCgeneric.zip *archive from* https://users.dcae.pub.ro/~gstefan/2ndLevel/digital_circuits.html. *Unzip and open the project it contains. Add the* right rotate *instruction to the instruction set:*

```
‘define rrot 6’b01_1010 // rf[d] <= {rf[l][0], rf[l][31:1] }
```

and make the necessary changes to the project. Write a simple program to test the correct operation of the new instruction.

Problem 5.12 *Download the* `toyRISCgeneric.zip` *archive from* `https://users.dcae.pub.ro/~gstefan/2ndLevel/digital_circuits.html`. *Unzip and open the project it contains. Add the* `bitwise NAND` *instruction to the instruction set:*

```
‘define bwnand 6’b01_1011 // rf[d] <= !(rf[l] && rf[r])
```

and make the necessary changes to the project. Write a simple program to test the correct operation of the new instruction.

Problem 5.13 *Download the* `toyRISCgeneric.zip` *archive from* `https://users.dcae.pub.ro/~gstefan/2ndLevel/digital_circuits.html`. *Unzip and open the project it contains. Add the* `modulus of difference` *instruction to the instruction set:*

```
‘define msub 6’b01_1100 // rf[d] <= ((rf[l]-rf[r])<0) ? rf[r]-rf[l]:rf[l]-rf[r]
```

and make the necessary changes to the project. Write a simple program to test the correct operation of the new instruction.

Chapter 6

COMPUTING MACHINES: $\geq$4–loop digital 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*.

System belonging to 4-OS can be obtained also by increasing the order of the components inside the structure of the processor circuit. Such a system becomes an 4-OS even if the functionality remains that of a processor. The main effect obtained on this way is the increase of the autonomy of the components of the processor with an important effect on the programmability. Thus, we start with presenting a processor as a 4-OS, and then we shortly present the systems obtained by closing various loop over a processor.

6.1 Processors as 4-OS

Let us revisit the toyRISC processor investigated in the previous chapter, and remembering the loop closed over an automaton through a memory element (see Section 5.2) we add a carry flip-flop (one-bit

[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.

register) as is represented in Figure 6.1. In this way we added the simplest state register in our structure by updating the execution unit RALU to a execution unit as a 3-OS: RALU loop coupled with a one-bit state register.

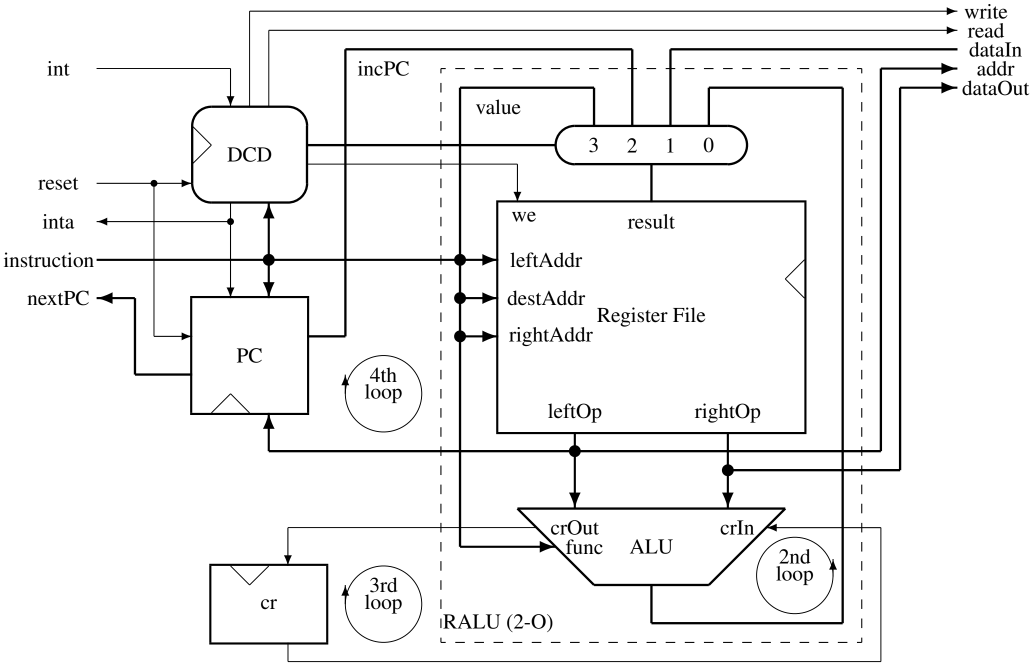


Figure 6.1: The toyRISC processor with carry state register.

`DEFINES.vh` **file** defines the micro-architecture of the improved toyRISC processor. The instructions `addc, subc`, and `addvc` are redefined.

```
/*************************************************************************
File name: DEFINES.vh
                    MICROARCHITECTURE
*************************************************************************/
// CONTROL
`define nop     6'b00_0000 // no operation: pc<=pc+1;
`define rjmp    6'b00_0001 // relative jump: pc<=pc+v;
`define zbr     6'b00_0010 // pc<=(rf[l]=0) ? pc+v:pc+1
`define nzbr    6'b00_0011 // pc<=!(rf[l]=0) ? pc+v:pc+1
`define ret     6'b00_0101 // return: pc<=rf[l][15:0];
`define halt    6'b00_0110 // halt unitil interrupt
`define eint    6'b00_1000 // set enable interrupt; pc<=pc+1;
`define dint    6'b00_1001 // set disable interrupt; pc<=pc+1;
// ARITHMETIC & LOGIC, for these instructions: pc<=pc+1;
```

```
‘define add     6’b11_0000 // {cr, rf[d]}<=rf[l]+rf[r];
‘define sub     6’b11_0001 // {cr, rf[d]}<=rf[l]-rf[r];
‘define addv    6’b11_0010 // {cr, rf[d]}<=rf[l]+v;
‘define mult    6’b11_0011 // {1’b0, rf[d]}<=rf[l]*rf[r];
‘define multv   6’b11_0100 // {1’b0, rf[d]}<=rf[l]*v;
‘define addc    6’b11_0101 // {cr, rf[d]}<=(rf[l]+rf[r]}[32]+cr;
‘define subc    6’b11_0110 // {cr, rf[d]}<=(rf[l]-rf[r])[32]-cr;
‘define addvc   6’b11_0111 // {cr, rf[d]}<=(rf[l]+v)[32]+cr;
‘define lsh     6’b11_1000 // {rf[l][0], rf[d]}<=rf[l] >> 1;
‘define ash     6’b11_1001 // {rf[l][0], rf[d]}<={rf[l][31],rf[l][31:1]};
‘define move    6’b11_1010 // {1’b0, rf[d]}<=rf[l];
‘define swap    6’b11_1011 // {1’b0, rf[d]}<={rf[l][15:0],rf[l][31:16]};
‘define bwnot   6’b11_1100 // {1’b0, rf[d]}<=~rf[l];
‘define bwand   6’b11_1101 // {1’b0, rf[d]}<=rf[l]&rf[r];
‘define bwor    6’b11_1110 // {1’b0, rf[d]}<=rf[l]|rf[r];
‘define bwxor   6’b11_1111 // {1’b0, rf[d]}<=rf[l]^rf[r];
// DATA TRANSFER, for these instructions: pc=pc+1;
‘define read    6’b10_0000 // read from dataMemory[rf[l]];
‘define store   6’b10_1000 // dataMemory[rf[l]]<=rf[r];
‘define load    6’b10_0111 // {1’b0, rf[d]}<=dataOut;
‘define val     6’b01_0111 // {1’b0, rf[d]}<={{16*{v[15]}},v};
‘define rotate  6’b01_0000 // {rf[l][0], rf[d]}<={rf[l][0], rf[l][31:1]}
```

Consequently, some of the files defined in the structural descriptions provided in the previous chapter are redesigned. In the next paaragraph are listed only the modules that have undergone changes.

`toyRISC.sv` **file** is the top module of the improved toyRISC processor where the additional loop through the carry flip-flop `cr` is introduced.

```
/***********************************************************************
File name: toyRISC.sv
                    toyRISC
***********************************************************************/
‘include ”DEFINES.vh”
module toyRISC( input   logic [31:0]  instr     ,
                output  logic [9:0]   nextPC    ,
                input   logic         intIn     ,
                output  logic         inta      ,
                input   logic [31:0]  dataIn    ,
                output  logic [31:0]  dataOut   ,
                output  logic [9:0]   addr      ,
                output  logic         dataRead  ,
                output  logic         dataWrite ,
                input   logic         reset     ,
                input   logic         clk       );
    logic   [5:0]   opCode  ;
    logic   [4:0]   d, l, r ; // dest, left, right addresses for rf
    logic   [31:0]  v       ; // immediate value
    logic   [31:0]  leftOp  ;
    logic   [9:0]   pc      ;
    logic           we      ;
    logic   [1:0]   sel     ;
    logic           cr      ; // carry ff
    logic           crOut   ;

    assign opCode   = instr[31:26]                     ;
    assign d        = instr[25:21]                     ;
    assign l        = instr[20:16]                     ;
    assign r        = instr[15:11]                     ;
    assign v        = {{16{instr[15]}}, instr[15:0]};

    DCDtoyRISC  DCD(opCode      ,
                    intIn       ,
                    inta        ,
                    dataRead    ,
                    dataWrite   ,
                    we          ,
                    sel         ,
                    reset       ,
                    clk         );
    PCtoyRISC   PC( nextPC  ,
                    pc      ,
                    leftOp  ,
                    v[9:0]  ,
                    opCode  ,
                    inta    ,
                    reset   ,
                    clk     );
    RALUtoyRISC RALU(   dataOut ,
                        leftOp  ,
                        crOut   ,
                        cr      ,
                        pc      ,
                        opCode  ,
                        dataIn  ,
                        v       ,
                        l       ,
                        r       ,
                        d       ,
                        sel     ,
                        we      ,
                        inta    ,
                        clk     );
    always_ff @(posedge clk) cr <= crOut     ;
    assign addr = leftOp[15:0]  ;
endmodule
```

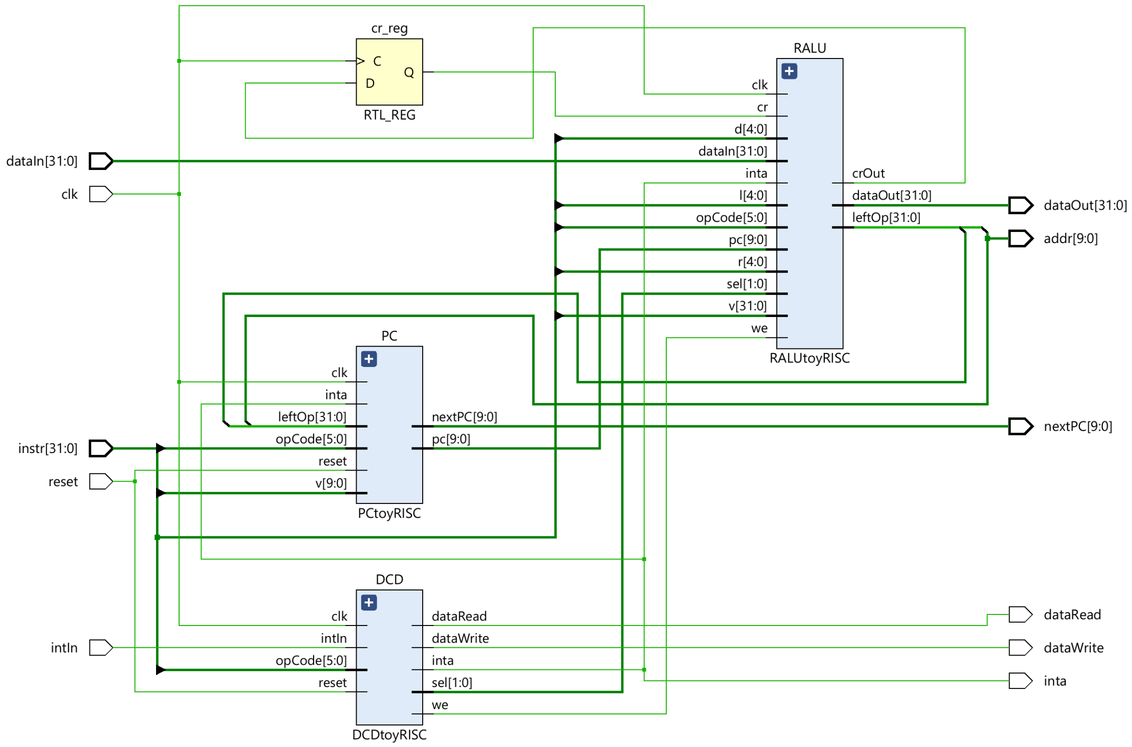


Figure 6.2: Top level of the toyRISC processor with carry FF.

`RALUtoyRISC.sv` **file** contains the modified RALU.

```
/*****************************************************************************
File name: RALUtoyRISC.sv
                         toyRISC's RALU
*****************************************************************************/
‘include ”DEFINES.vh”
module RALUtoyRISC( output  logic   [31:0]  dataOut ,
                    output  logic   [31:0]  leftOp  ,
                    output  logic           crOut   ,
                    input   logic           cr      ,
                    input   logic   [9:0]   pc      ,
                    input   logic   [5:0]   opCode  ,
                    input   logic   [31:0]  dataIn  ,
                                            v       ,
                    input   logic   [4:0]   l       ,
                                            r       ,
                                            d       ,
                    input   logic   [1:0]   sel     ,
                    input   logic           we      ,
                                            inta    ,
                                            clk     );
    logic   [31:0]  leftIn  ;
    logic   [31:0]  rightIn ;
    logic   [31:0]  muxOut  ;
    logic   [31:0]  aluOut  ;

    registerFile    regFile(.leftOp     (leftIn ),
                            .rightOp    (rightIn),
                            .in         (muxOut ),
                            .leftAddr   (l      ),
                            .rightAddr  (r      ),
                            .destAddr   (d      ),
                            .we         (we     ),
                            .inta       (inta   ),
                            .opCode     (opCode ),
                            .clk        (clk    ));

    assign dataOut  = rightIn   ;
    assign leftOp   = leftIn    ;

    ALU alu(.out     (aluOut),
            .crOut   (crOut  ),
            .cr      (cr     ),
            .leftIn  (leftIn ),
            .rightIn(rightIn),
            .value   (v      ),
            .opCode  (opCode ));

    bigMux bigMux(  .aluOut (aluOut),
                    .dataIn (dataIn),
                    .value  (v      ),
                    .pc     (pc     ),
                    .sel    (sel    ),
                    .muxOut (muxOut ));
endmodule
```

`ALU` **file** contains the modified ALU

```
/*****************************************************************************
File name: ALU.sv
                        toyRISC's ALU
*****************************************************************************/
module ALU( output  logic   [31:0]  out     ,
            output  logic           crOut   ,
            input   logic           cr      ,
            input   logic   [31:0]  leftIn  ,
                                    rightIn ,
                                    value   ,
            input   logic   [5:0]   opCode  );
```

```
    always_comb
      case(opCode)
        `add    : {crOut,out} = leftIn + rightIn                     ;
        `sub    : {crOut,out} = leftIn - rightIn                     ;
        `addv   : {crOut,out} = leftIn + value                       ;
        `mult   : {crOut,out} = {1'b0, leftIn * rightIn}             ;
        `multv  : {crOut,out} = {1'b0, leftIn * value}               ;
        `addc   : {crOut,out} = leftIn + rightIn + cr                ;
        `subc   : {crOut,out} = leftIn - rightIn - cr                ;
        `addvc  : {crOut,out} = leftIn + value + cr                  ;
        `lsh    : {crOut,out} = {leftIn[0], 1'b0, leftIn[31:1]}      ;
        `ash    : {crOut,out} = {leftIn[0], leftIn[31], leftIn[31:1]} ;
        `move   : {crOut,out} = {1'b0, leftIn}                       ;
        `swap   : {crOut,out} = {1'b0, leftIn[15:0], leftIn[31:16]}  ;
        `bwnot  : {crOut,out} = {1'b0, ~leftIn}                      ;
        `bwand  : {crOut,out} = {1'b0, leftIn & rightIn}             ;
        `bwor   : {crOut,out} = {1'b0, leftIn | rightIn}             ;
        `bwxor  : {crOut,out} = {1'b0, leftIn ^ rightIn}             ;
        `rotate : {crOut,out} = {1'b0, leftIn[0], leftIn[31:1]}      ;
        default : {crOut,out} = {1'b0, leftIn}                       ;
      endcase
endmodule
```

The main effect of the carry FF is the way operations on big numbers are performed. The next example refers the Example 5.6.

Example 6.1 *Adding of two 64-bit numbers is illustrated by the following two programs:*

```
/*********************************************************************
File name: program.sv
*********************************************************************/
            VAL(0,-3);
            VAL(1,5);
            VAL(2,6);
            VAL(3,7);
    //: {rf[1],rf[0]} <= {rf[1],rf[0]} + {rf[3],rf[2]}
            ADD(0,0,2);
            ADDC(1,1,3);
            HALT;
```

```
t=0  pc=   x  aluOut=         x cr=x RF=[x,          x,  x, x, x] ei=x inta=x
t=1  pc=1023  aluOut=         x cr=0 RF=[x,          x,  x, x, x] ei=0 inta=0
t=5  pc=   0  aluOut=4294967293 cr=0 RF=[4294967293, x,  x, x, x] ei=0 inta=0
t=7  pc=   1  aluOut=4294967293 cr=0 RF=[4294967293, x,  x, x, x] ei=0 inta=0
t=9  pc=   2  aluOut=4294967293 cr=0 RF=[4294967293, 5,  x, x, x] ei=0 inta=0
```

```
t=11 pc=   3  aluOut=4294967293 cr=0 RF=[4294967293, 5,  6, x, x] ei=0 inta=0
t=13 pc=   4  aluOut=         3 cr=0 RF=[4294967293, 5,  6, 7, x] ei=0 inta=0
t=15 pc=   5  aluOut=        13 cr=1 RF=[3,          5,  6, 7, x] ei=0 inta=0
t=17 pc=   6  aluOut=         3 cr=0 RF=[3,          13, 6, 7, x] ei=0 inta=0
```

```
/*************************************************************************
File name: program.sv
*************************************************************************/
            VAL(0,3);
            VAL(1,5);
            VAL(2,6);
            VAL(3,7);
    //: {rf[1], rf[0]} <= {rf[1], rf[0]} + {rf[3], rf[2]}
            ADD(0,0,2);
            ADDC(1,1,3);
            HALT;
```

```
t=0  pc=   x  aluOut= x cr=x RF=[x,  x, x, x, x] ei=x inta=x
t=1  pc=1023  aluOut= x cr=0 RF=[x,  x, x, x, x] ei=0 inta=0
t=5  pc=   0  aluOut= 3 cr=0 RF=[3,  x, x, x, x] ei=0 inta=0
t=7  pc=   1  aluOut= 3 cr=0 RF=[3,  x, x, x, x] ei=0 inta=0
t=9  pc=   2  aluOut= 3 cr=0 RF=[3,  5, x, x, x] ei=0 inta=0
t=11 pc=   3  aluOut= 3 cr=0 RF=[3,  5, 6, x, x] ei=0 inta=0
t=13 pc=   4  aluOut= 9 cr=0 RF=[3,  5, 6, 7, x] ei=0 inta=0
t=15 pc=   5  aluOut=12 cr=0 RF=[9,  5, 6, 7, x] ei=0 inta=0
t=17 pc=   6  aluOut= 9 cr=0 RF=[9, 12, 6, 7, x] ei=0 inta=0
```

◇

The *increased autonomy* of the execution unit, because of the additional luull through the carry FF, had as its main consequence a 2-instruction program instead of the 4-instruction program for adding double size numbers.

6.2 Types of fourth order systems

There are four main types of fourth order systems based on loops closed over a processor (see Figure 6.3) 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 6.3a 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 6.3b) that stores both data and programs

3. **P & LIFO** is a 4-OS with loop closed through a 2-OS - in Figure 6.3c 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 6.3d 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.

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.

6.3 The computer – support for the strongest segregation

The ROM content is defined symbolically and after that it is converted in the actual physical structure of ROM. Instead, the RAM content remains in symbolic form and has, in consequence, more flexibility. This is the main reason for considering the PROCESSOR & RAM = COMPUTER as the most representative in 4-OS.

The computer is not a circuit. It is a new entity with a special functional definition, currently called **computer architecture**. Mainly, the computer architecture is given by the machine language. A program written in this language is interpreted or executed by the processor. The program is stored in the RAM memory. In the same subsystem are stored data on which the program "acts". Each architecture can have many associated computer structures (organizations).

Starting from the level of four order systems the behavior of the system is controlled mainly by the symbolic structure of programs. The architectural approach settles the distinction between the physical structures and the symbolic structures. Therefore, any computing **machine** supposes the following triadic definition (suggested by ["Milutinovic" '89]):

- the machine language (usually called *architecture*)

- the storage containing programs written in the machine language

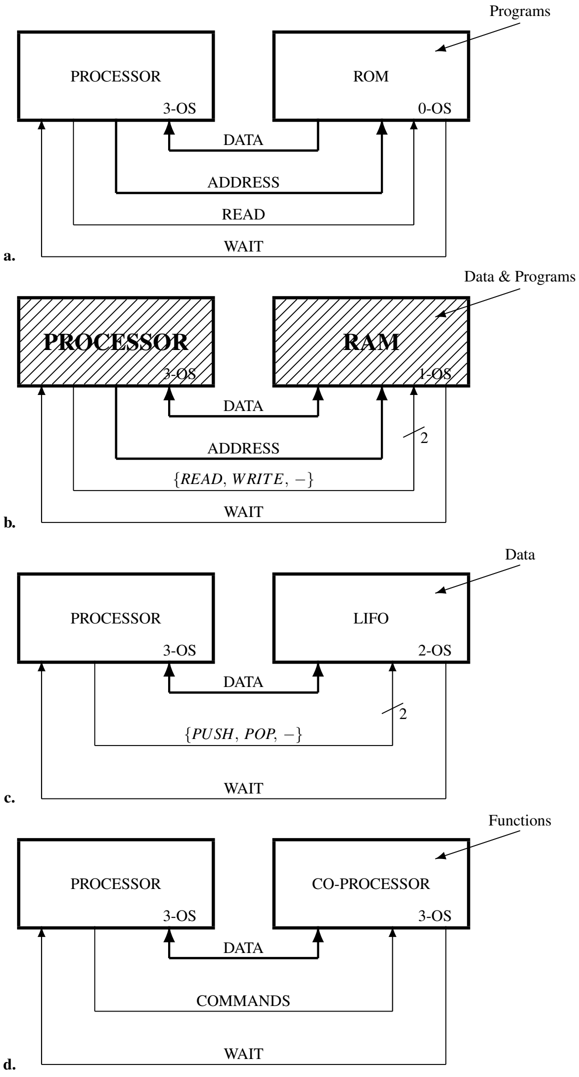


Figure 6.3: **The four types of 4-OS machines.** **a.** Fix program computers usual in embedded computation. **b.** General purpose computer. **c.** Specialized computer working working on a restricted data structure. **d.** Accelerated computation supported by a specialized co-processor.

- the **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*.

6.3.1 Four-Loop Circuits (4-OS) & Controlling by Information

In the previous subsection, the information interacts directly with the physical structure. All the information is executed or interpreted by the circuits. The next step disconnects partially the information from circuits. In a system, having four loops the information can be interpreted by another information acting to the lower level in the system. The typical 4-OS is the *computer* structure (see Chapter **??**). This structure is more than we need for computing. Indeed, as we said in subsection **??**, the partial recursive functions can be computed in 3-OSs. Why are we interested in using 4-OS for performing computations? The answer is: *for segregating more the simple circuits from random (complex) informational structure*. In a system having four loops the simple and the complex are maximal segregated, the first in circuits and the second in information.

At the 3-OS level, the information also interacts, and thereby acts, with the structure of the circuits through the flags. The control performed depends on what happens directly in the controlled circuits. At the 4-OS level, the control is taken over by the information in an imperative way, a way that no longer depends at all on the signals coming directly from the circuits.

Starting from the level of the fourth order systems the functional aspects of a digital system is imposed mainly by the information. The role of the circuits decreases. Circuits become simple even if they gain in size. The complexity of the computation switches from circuits to information.

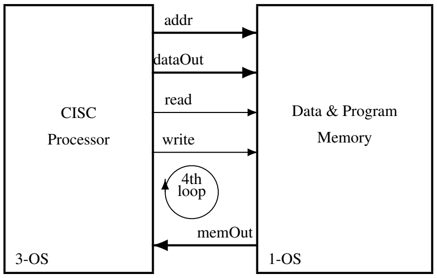


Figure 6.4: von Neumann abstract model for computer.

6.3.2 Five-Loop Circuits (5-OS): Computer with RISC Processor

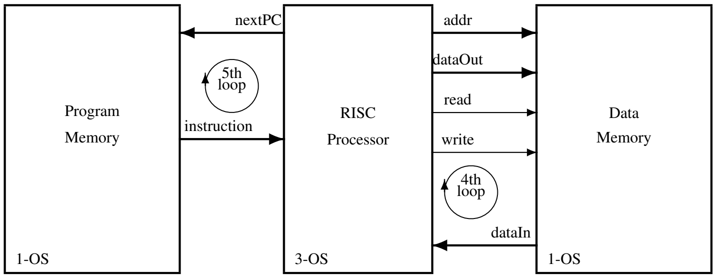


Figure 6.5:

ANNEXES

Appendix A

Binary Arithmetic

A.1 Binary representations

A.1.1 Positive integers

Positive integer denoted by $\mathbb{Z}^+$, and they are the solution to the simple linear recurrence equation $a_n = a_{n-1} + 1$ with $a_1 = 1$.

A n-bit number:

$$B_{n-1}B_{n-2}\ldots B_1B_0 \Rightarrow B_{n-1} \times 2^{n-1} + B_{n-2} \times 2^{n-2} + \ldots + B_1 \times 2^1 + B_0 \times 2^0$$

where $B_i \in \{0,1\}$ for $i = 0,1,\ldots n-1$.

A.1.2 Decimal to binary conversion

The algorithm of converting the decimal number D in a n-bit binary form is:

```
step 1  $D_0 = \lfloor D/2 \rfloor$                    // the whole part of D divided by 2
        $B_0 = D - \lfloor D/2 \rfloor \times 2$        // the remainder of dividing D by 2

step 2  $D_1 = \lfloor D_0/2 \rfloor$
        $B_1 = D_0 - \lfloor D_0/2 \rfloor \times 2$

  ...

step n  $D_{n-1} = \lfloor D_{n-2}/2 \rfloor$
        $B_{n-1} = D_{n-2} - \lfloor D_{n-2}/2 \rfloor \times 2$
```

A.1.3 Signed integers

Sign-magnitude representation

```
{sign, magnitude}

0_0000  => +0
0_0001  => +1
0_0010  => +2
```

```
...
0_1111  => +15
1_0000  => -0
1_0001  => -1
1_0010  => -2
...
1_1111  => -15
```

Ones' complement representation

Ones' complement is bitwise negation.

```
{sign, magnitude}

0_0000  => +0
0_0001  => +1
0_0010  => +2
...
0_1111  => +15
1_0000  => -15
1_0001  => -14
1_0010  => -13
...
1_1111  => -0
```

Two's complement representation

Two's complement is ones' complement plus 1

```
{sign, magnitude}

0_0000  => +0
0_0001  => +1
0_0010  => +2
...
0_1111  => +15
1_0000  => -16
1_0001  => -15
1_0010  => -14
...
1_1111  => -1
```

A.1.4 Fix point fractionary numbers

$$B_{n-1} \dots B_1 B_0 \,.\, F_1 F_2 \dots \Rightarrow B_{n-1} \times 2^{n-1} + \dots + B_1 \times 2^1 + B_0 \times 2^0 + F_1 \times 2^{-1} + F_2 \times 2^{-2} + \dots$$

where $B_i, F_i \in \{0, 1\}$ for $i = 0, 1, \dots$.

A.1.5 Floating point numbers

```
{sign, exponent, fraction}
```

$$sgn\ 1.fraction \times 2^{exponent-127}$$

IEEE half-precision 16-bit float: deep learning artificial intelligence

```
{sign, exponent[4:0], fraction[9:0]}
```

Google's brain float bfloat16 is a 16-bit float:

```
{sign, exponent[7:0], fraction[6:0]}
```

NVidia's TensorFloat 19-bit float:

```
{sign, exponent[7:0], fraction[9:0]}
```

AMD's fp24 24-bit float:

```
{sign, exponent[6:0], fraction[14:0]}
```

Pixar's PXR24 24-bit float:

```
{sign, exponent[7:0], fraction[14:0]}
```

IEEE 754 single-precision 16-bit float:

```
{sign, exponent[7:0], fraction[22:0]}
```

Example A.1 *In IEEE 754 single-precision, the number:*

```
0_10000010_11000000000000000000000
```

corresponds to:

$$+(1+(0.5+0.25)) \times 2^{130-127} = 14$$

◇

A.2 Adding/Substracting

The most efficient representation for add/sub is twos complement.

A.2.1 Adding positive integers

Carry represents the overflow for positive integer addition. We consider integers represented on 4 bits.

A first example with no carry:

```
   +1       0001
   +6       0110
   --       ----
   +5       0111
```

An example with carry:

```
   +10      1010
   +8       1000
   --       ----
(1)+16   (1)0010
```

A.2.2 Adding signed integers

Simply add the numbers and ignore any carry out of the highest bit.

```
   -1       1_1111
   +6       0_0110
   --       ------
   +5    (1)0_0101

   +1       0_0001
   -4       1_1100
   --       ------
   -3    (0)1_1101
```

A.2.3 Subtracting

A-B means A+(2s compl of B)

```
3-2 => 0_0011+NOT(0_0010)+1 => 0_0011+1_1101+1 => (1)0_0001 = 0_0001
```

A.2.4 Overflow

In signs of the operands, `sgn1` and `sgn2`, are different, overflow is not possible. It they are the same overflow is possible if the sign of the result, `sgnR`, is different from the common signs of operands.

```
overflow = (sgn1 ^ sgn0)' & (sgn1 ^ sgnR)
```

A.3 Multiply/Divede

A.3.1

A.3.2

Appendix B

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.

B.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.

B.2 Elementary circuits: gates

Definition B.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 B.2 *We call* n**-bit binary variable** *an element of the set* $\{0,1\}^n$.

Definition B.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 $m > 1$.

B.2.1 Zero-input logic circuits

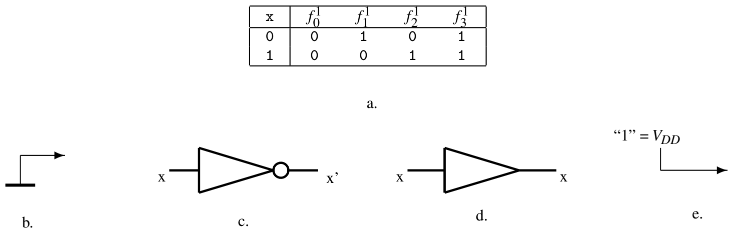
Definition B.4 *The* **0-bit logic function** *are* $f_0^0 = 0$ *(the false-function) which generates the one bit coded 0, and* $f_1^0 = 1$ *(the true-function) which generate the one bit coded 1.*

They are useful for generating initial values in computation (see the *zero* function as basic function in partial recursivity).

B.2.2 One input logic circuits

Definition B.5 *The* **1-bit logic functions**, *represented by* **true-tables** *in Figure B.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*



x	f_0^1	f_1^1	f_2^1	f_3^1
0	0	1	0	1
1	0	0	1	1

Figure B.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.

B.2.3 Two inputs logic circuits

Definition B.6 *The* **2-bit logic functions** *are represented by true-tables in Figure B.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:

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

a.

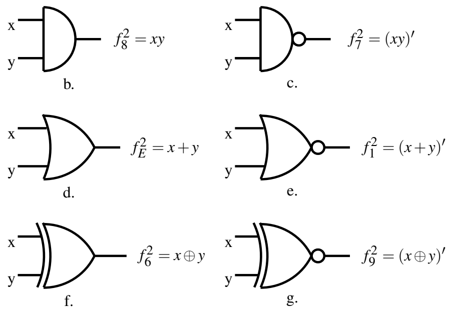


Figure B.2: **Two-bit logic functions.** **a.** The table of all two-bit logic functions. **b.** AND gate – the original gate. **c.** NAND gate – the most used gate. **d.** OR gate. **e.** NOR gate. **f.** XOR gate – modulo2 adder. **g.** NXOR gate – coincidence circuit.

  - 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*.

B.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 B.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.

B.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 B.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 B.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.
$a+b = (a'b')'$ $ab = (a'+b')'$
$a'+b' = (ab)'$ $a'b' = (a+b)'$

B.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.

B.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 B.7 *A* **minterm** *associated to an n-input logic function is a logic product (AND logic function) depending by all n binary variable.* $\diamond$

Definition B.8 *A* **maxterm** *associated to an n-input logic function is a logic sum (OR logic function) depending by all n binary variable.* $\diamond$

Definition B.9 *The* **disjunctive normal form***, DNF, of an n-input logic function is a logic sum of minterms.* $\diamond$

Definition B.10 *The* **conjunctive normal form***, CNF, of an n-input logic function is a logic product of maxterms.* $\diamond$

Example B.1 *Let be the combinational multiplier for 2 2-bit numbers described in Figure B.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 B.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: $S_{mult2} = 54$*.* $\diamond$

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 B.4: **Combinatinal circuit represented a a truth table.** The truth table of the combinational circuit performing 2-bit multiplication.

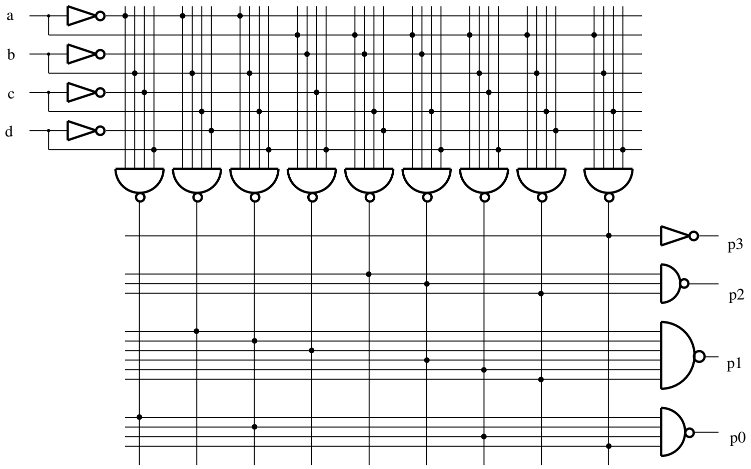


Figure B.5: **Direct implementation of a combinational circuit.** The direct implementation starting from DNF of the 2-bit multiplier.

B.4.2 Algebraic minimization

Minimal depth minimization

Example B.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 B.6. ⋄

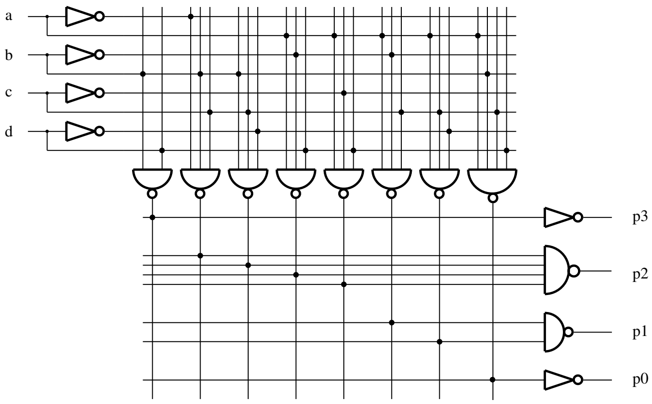


Figure B.6: **Minimal depth minimiztion** The first, minimal depth minimization of the 2-bit multiplier.

Multi-level minimization

Example B.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 B.7 with size 22. ◇

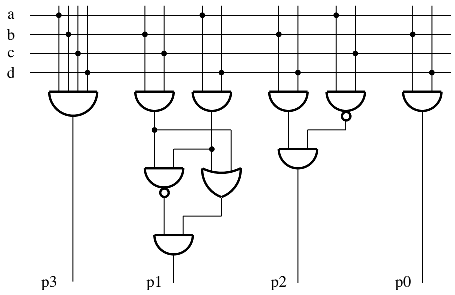


Figure B.7: **Multi-level minimization.** The second, multi-level minimization of the 2-bit multiplier.

Many output circuit minimization

Example B.4 *Inspecting carefully the schematics from Figure B.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 B.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. ◇

B.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.

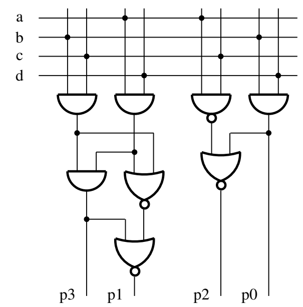


Figure B.8: **Multiple-output minimization.** The third, multiple-output minimization of the 2-bit multiplier.

Definition B.11 *The Hamming distance between two minterms is given by the total numbers of binary variable which occur distinct in the two minterms.* ⋄

Example B.5 *The Hamming distance between $m_9 = ab'c'd$ and $m_4 = a'bc'd'$ is 3, because only the variable b occurs in the same form in both minterms.*

The Hamming distance between $m_9 = ab'c'd$ and $m_1 = a'b'c'd$ is 1, because only the variable which occurs distinct in the two minterms is a. ⋄

Two n-variable terms having the Hamming distance 1 are minimized, using the exclusion principle, to one $(n-1)$-variable term. The size of the associated circuit is reduced from $2(n+1)$ 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 B.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.

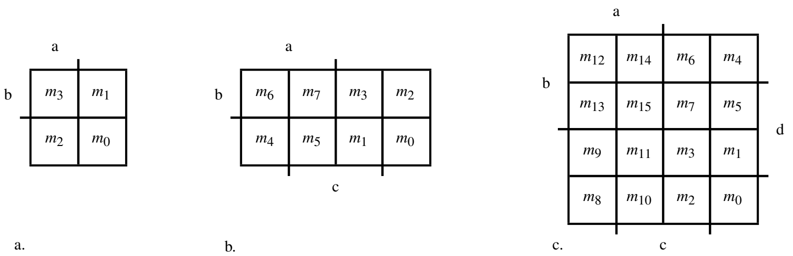

a
b
m_3 m_1
m_2 m_0
a.
a
b
m_6 m_7 m_3 m_2
m_4 m_5 m_1 m_0
c
b.
a
b
m_{12} m_{14} m_6 m_4
m_{13} m_{15} m_7 m_5
m_9 m_{11} m_3 m_1
m_8 m_{10} m_2 m_0
d
c.
c

Figure B.9: **Veitch diagrams.** The Veitch diagrams for 2, 3, and 4 variables.

Example B.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 B.10 p1 and p2 are represented.

◇

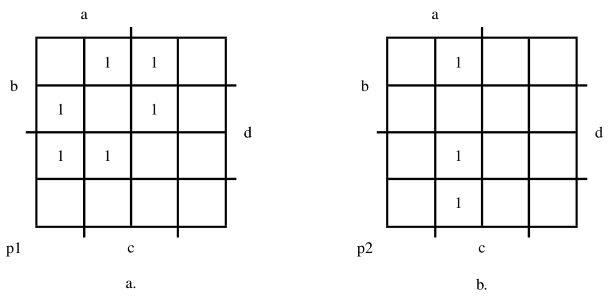


Figure B.10: **Using Veitch diagrams.** The Veitch diagrams for the functions p1 and p2.

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.

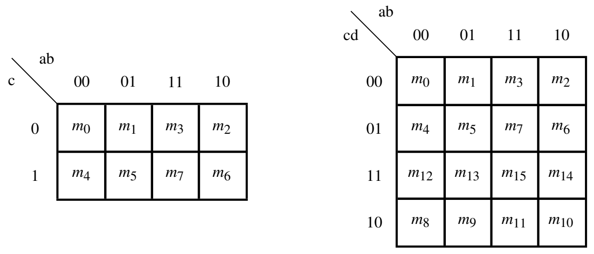

ab
c
00 01 11 10
0 m_0 m_1 m_3 m_2
1 m_4 m_5 m_7 m_6
ab
cd
00 01 11 10
00 m_0 m_1 m_3 m_2
01 m_4 m_5 m_7 m_6
11 m_{12} m_{13} m_{15} m_{14}
10 m_8 m_9 m_{11} m_{10}

Figure B.11: **Karnaugh diagrams.** The Karnaugh diagrams for 3 and 4 variables.

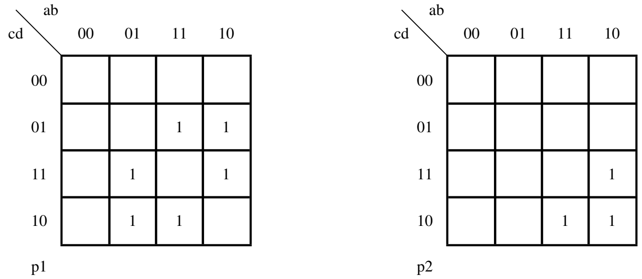
In Figure B.12 the same functions, p1 and p2, are represented. The distribution of the surface is different but the degree of adjacency is identical.

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



cd \ ab	00	01	11	10
00				
01			1	1
11		1		1
10		1	1	

p1

cd \ ab	00	01	11	10
00				
01				
11				1
10			1	1

p2

Figure B.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.

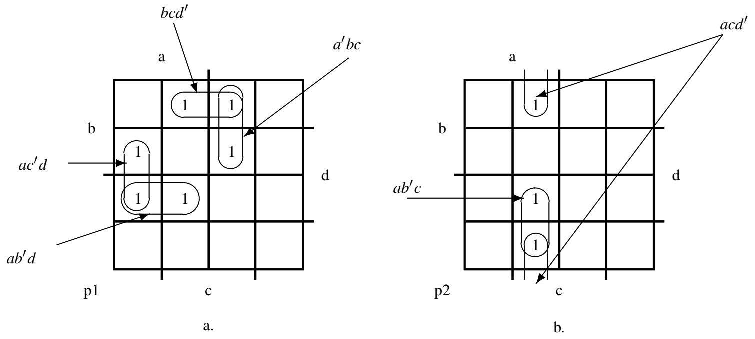


Figure B.13: **Minimizing with V-K diagrams.** Minimizing the functions $p1$ and $p2$.

Example B.7 *Let's take the V-K diagrams from Figure B.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. ◇

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).

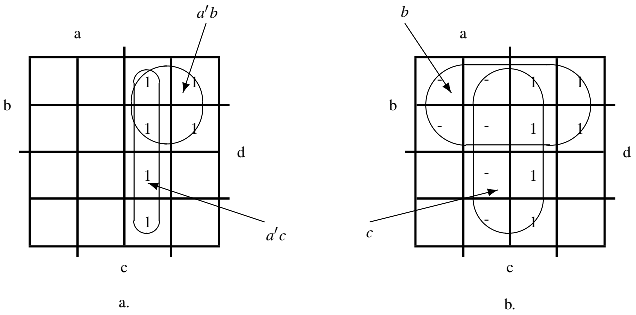


Figure B.14: **Minimizing incomplete defined functions. a.** The minimization of y (Example 1.8) ignoring the *"don't care"* terms. **b.** The minimization of y (Example 1.8) considering the *"don't care"* terms.

Example B.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 B.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 B.14b. Now each of the two surfaces are doubled resulting a more simplified form: $y = b + c$. ⋄

V-K diagrams with included functions

For various reasons in a V-K diagram we need to include instead of a logic value, 0 or 1, a logic function of variables which are different from the variables associated with the V-K diagram. For example, a minterm depending on a, b, c, d can be defined as taking a value which is depending on another logic 2-variable function by s, t.

A *simplified rule* to extract the minimized form of a function from a V-K diagram containing included functions is the following:

1. consider first only the 1s from the diagram and the rest of the diagram filed only with 0s and extract the resulting function

2. consider the 1s as "don't care"s for surfaces containing the same function and extract the resulting function "multiplying" the terms with the function

3. "add" the two functions.

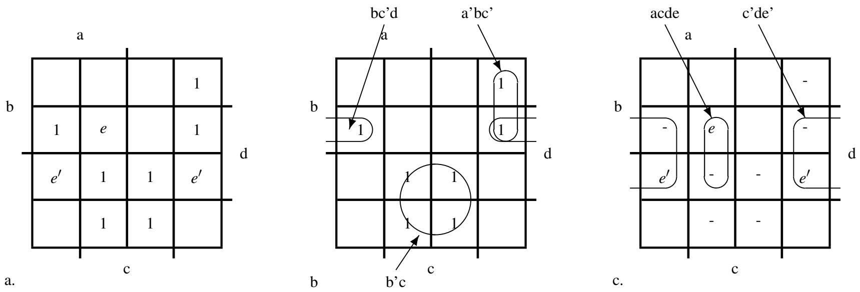


Figure B.15: **An example of V-K diagram with included functions.** **a.** The initial form. **b.** The form considered in the first step. **c.** The form considered in the second step.

Example B.9 *Let be the function defined in Figure B.15a. The first step means to define the surfaces of 1s ignoring the squares containing functions. In Figure B.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 B.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.

Appendix C

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).

C.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 C.1, where is defined the boundary between the values interpreted as 0 and the values interpreted as 1.

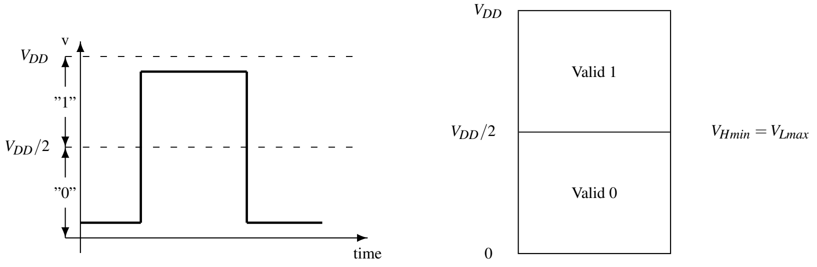


Figure C.1: **Defining 0-logic and 1-logic.** The circuit is supposed to interpret any value under $V_{DD}/2$ as 0, and any value bigger than $V_{DD}/2$ are interpreted as 1.

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 C.2, where $V_L < V_{Lmax}$ and $V_H > V_{Hmin}$.

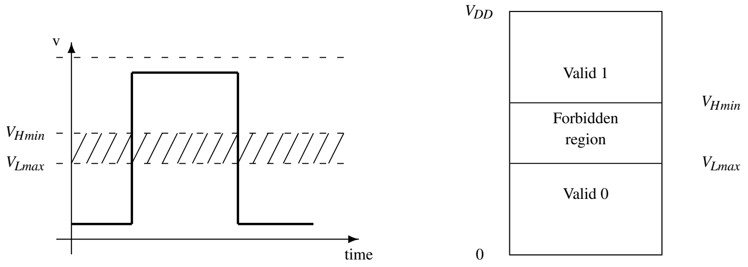


Figure C.2: **Defining the “forbidden region” for logic values.** A robust design asks a net distinction between the electrical values interpreted as 0 and the electrical values interpreted as 1.

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 C.3 the noise generator simulate the parasitic effects of the circuits switching in a small neighborhood.

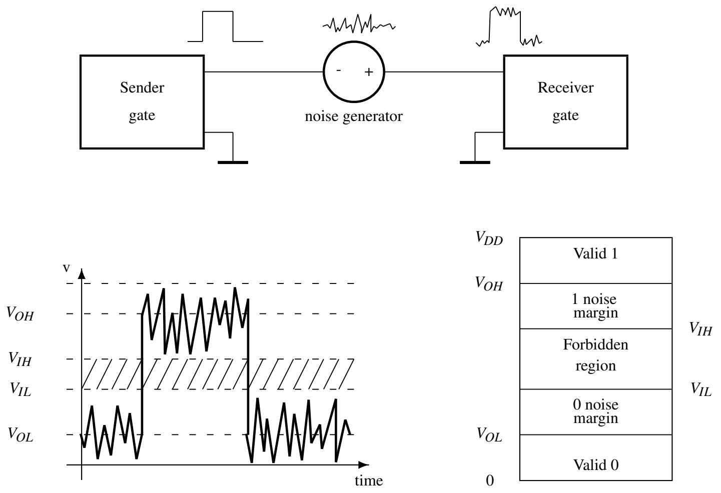


Figure C.3: **The noise margin.** The output signal must be generated with more restrictions to allow the receivers to “understand” correct input signals loaded with noise.

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.

C.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 C.4.

A switch connected to V_{DD} is implemented using a p-MOS transistor. It is represented in Figure C.4a *off* (generating *z*, which means *Hi-Z*: no signal, neither 0, nor 1) and in Figure C.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 C.4c *off* (generating *z*, which means *Hi-Z*: no signal, neither 0, nor 1) and in Figure C.4e it is represented *on* (generating 0 logic, or *false*).

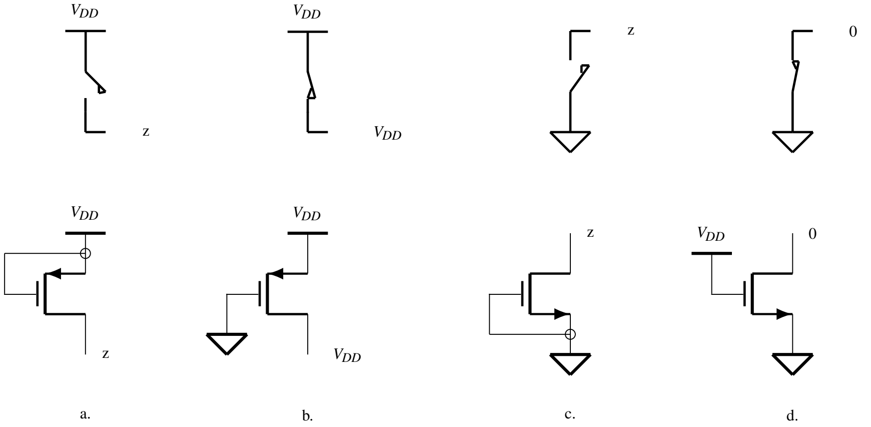


Figure C.4: **Basic switches.** **a**. Open switch connected to V_{DD}. **b**. Closed switch connected to V_{DD}. **c**. Open switch connected to *ground*. **d**. Closed switch connected to *ground*.

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 C.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:

$$i_G = C_{GS}\frac{dv_{GS}}{dt}$$

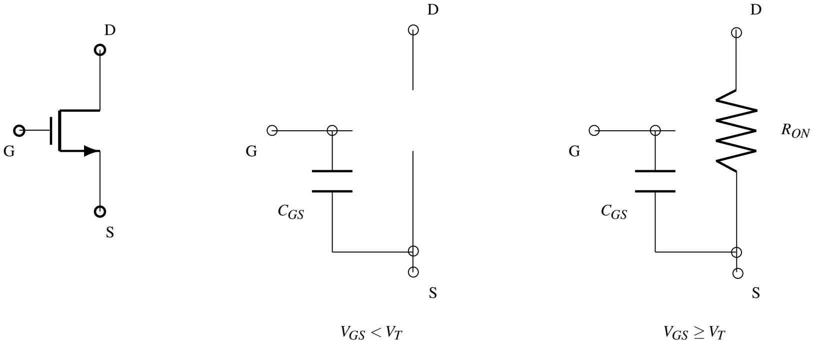


Figure C.5: **The MOSFET switch.** The *switch-resistor-capacitor* model consists in the two states: OF ($V_{GS} < V_T$), and ON ($V_{GS} \geq V_T$). In both states the input is defined by the capacitor C_{GS}.

Example C.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:*

$$C_{GS} = 4 \times 0.13 \times 1.8 fF = 0.936 fF$$

Assuming $R_n = 5K\Omega$*, results for the same gate:*

$$R_{ON} = 5 \times \frac{0.13}{1.8} K\Omega = 361\Omega$$

◇

C.3 The Inverter

C.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

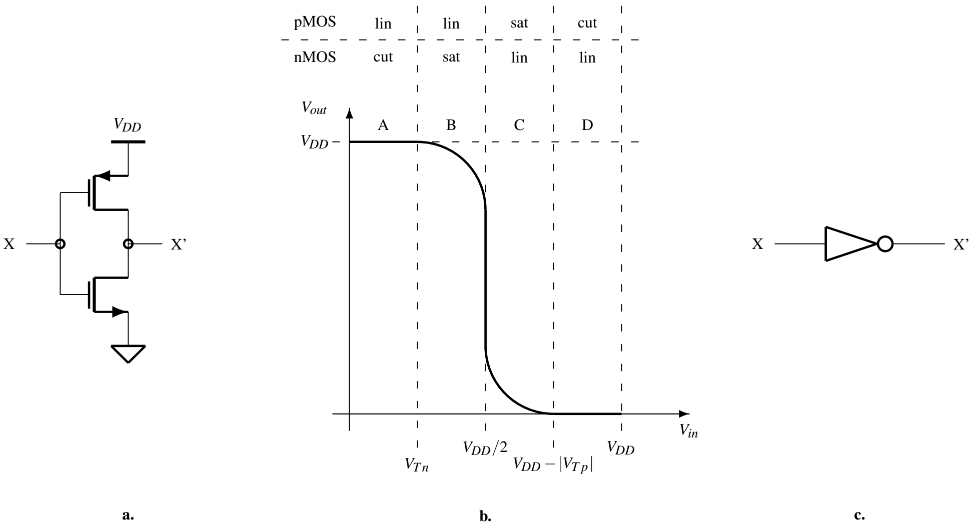


Figure C.6: **Building an invertor.** **a**. The invertor circuit. **b**. The logic symbol for the invertor circuit.

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 C.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 C.4. Connecting together a switch generating z with a switch generating 1 or 0, the connection point will generate 0 or 1.

C.3.2 Dynamic behavior

The propagation time of an inverter can be analyzed using the two serially connected invertors represented in Figure C.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

$$C_L = C_{DB} + C_{wire} + C_G$$

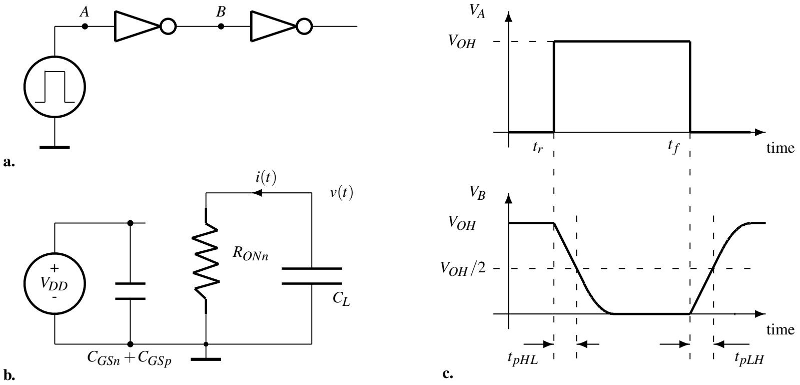


Figure C.7: **The propagation time.**

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 C.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 C.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 C.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 C.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}.

$$\frac{dv_{out}}{dt} = \frac{d}{dt}(\frac{q_L}{C_L}) = \frac{-I_{Dn(sat)}}{C_L}$$

$$\frac{dv_{out}}{dt} = \frac{\frac{V_{OH}}{2} - V_{OH}}{t_{pHL}}$$

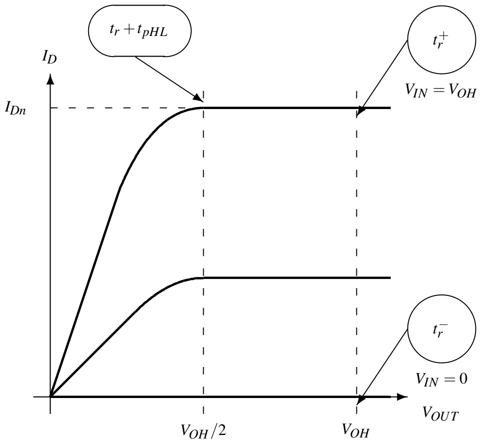


Figure C.8: **The output characteristic of the nMOS transistor.**

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.

C.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 C.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 C.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:

$$t_{p(buffered)} = t_{p0}(\frac{W_1}{W_{driver}} + \frac{W_2}{W_1} + \frac{W_{load}}{W_2})$$

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.

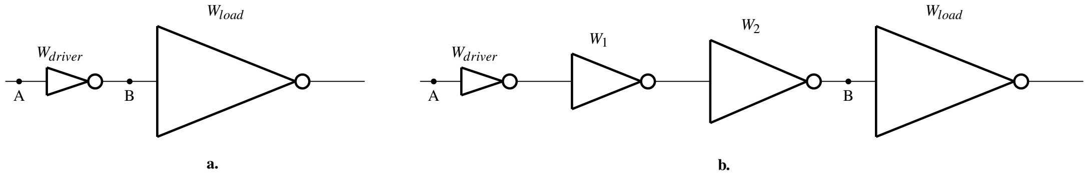


Figure C.9: **Buffered connection. a.** An invertor with small W is not able to handle at high frequency a circuit with big W. **b.** The buffered connection with two intermediary buffers.

In order to design the size of the circuits in Figure C.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.

C.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:

$$Q_L = C_L V_{DD}$$

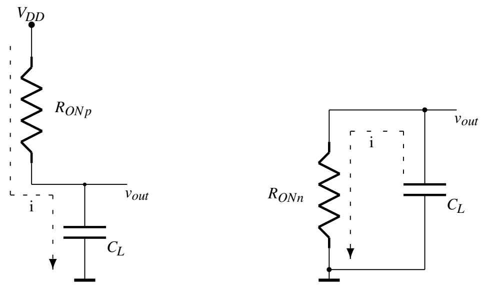


Figure C.10: **The main power consuming process.** For $V_{in} = 0$ C_L is loaded by the current provided by R_{ONp}. The charge from C_L is transferred to the ground through R_{ONn} for $V_{in} = V_{OH}$.

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:

$$p_{switch} = \frac{V_{DD}C_LV_{DD}}{T} = C_LV_{DD}^2f_{clock}$$

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 C.11). Thus is consumed the short-circuit power.

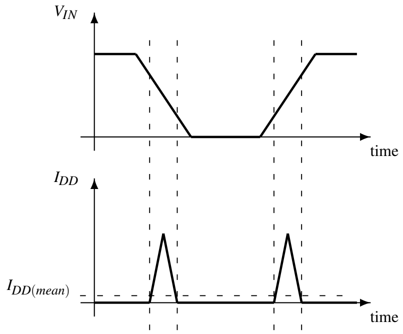


Figure C.11: **Direct flow of current from V_{DD} to ground.** This current due to the non-zero edge to the circuit input can be neglected.

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 C.12 the two components of the leakage current are presented for a NOT circuit with $V_{in} = 0$.

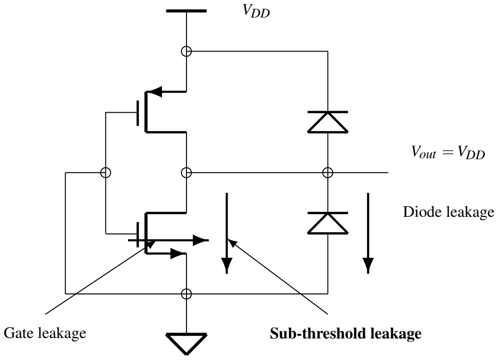


Figure C.12: **The two main components of the leakage current.** .

C.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*.

C.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 C.13 and C.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.

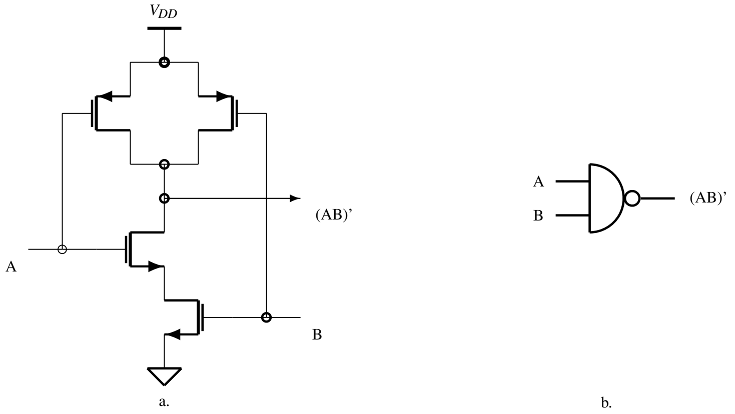


Figure C.13: **The NAND gate.** **a**. The internal structure of a NAND gate: the output is 1 when at least one input is 0. **b**. The logic symbol for NAND.

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, $R_{ONp} > R_{ONn}$.

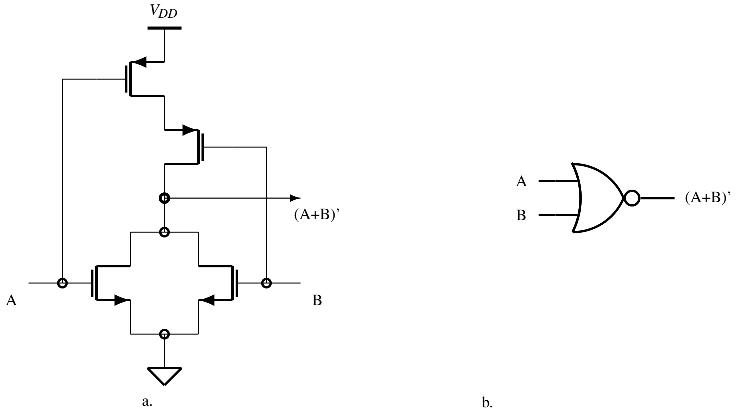


Figure C.14: **The NOR gate.** **a**. The internal structure of a NOR gate: the output is 1 only when both inputs are 0. **b**. The logic symbol for NOR.

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 C.15a).

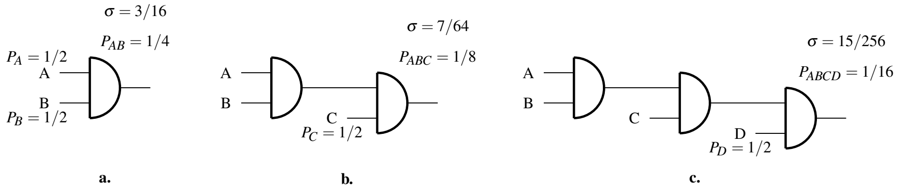


Figure C.15: **Switching activity σ and the output probability of 1.** **a**. For 2-input AND. **b**. For 3-input AND. **c.** For 4-input AND.

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 C.15a).

Switching activity for 3-input AND with the inputs A, B, and C is $\sigma_{NAND3} = 7/64$ (see Figure C.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 C.16. From the logical point of view, when the inputs switch form $ABC = 010$ to $ABC = 111$ the output must maintain its value on 1. Unfortunately, because the effect of the inputs A and B are affected by the extra delay introduced by the first gate, the unexpected ***glitch*** manifests to the output. Follow the wave forms form Figure C.16 to understand why.

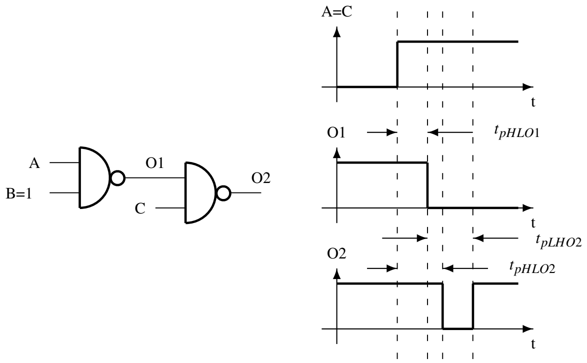


Figure C.16: **Glitching effect.** When the input value switch from $ABC = 010$ to $ABC = 111$ the output of the circuit must remain on 1. But, a short glitch occurs because of the delay, t_{pHLO1}, introduced by the first NAND.

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 **??**c and Figure **??**d. 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.

C.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 C.17a, is recommended to be designed as a binary tree of two input gates, see Figure C.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 C.17a, an 8-input NAND uses a similar arrangement as in Figure C.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 C.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 C.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

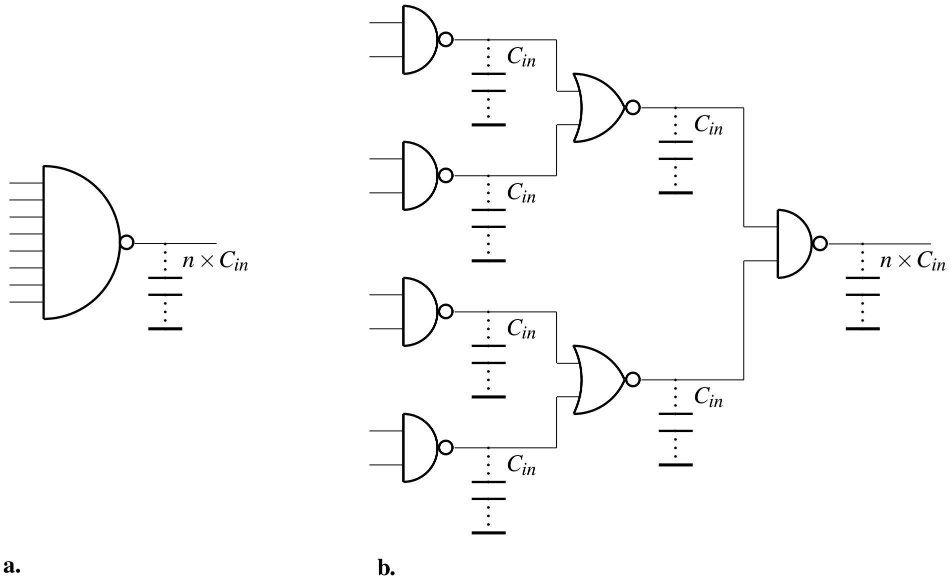


Figure C.17: **How to manage a many-input gate.** **a** An *NAND*$_8$ gate with fan-out *n*. **b**. The log-depth equivalent circuit.

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.

C.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 C.18.

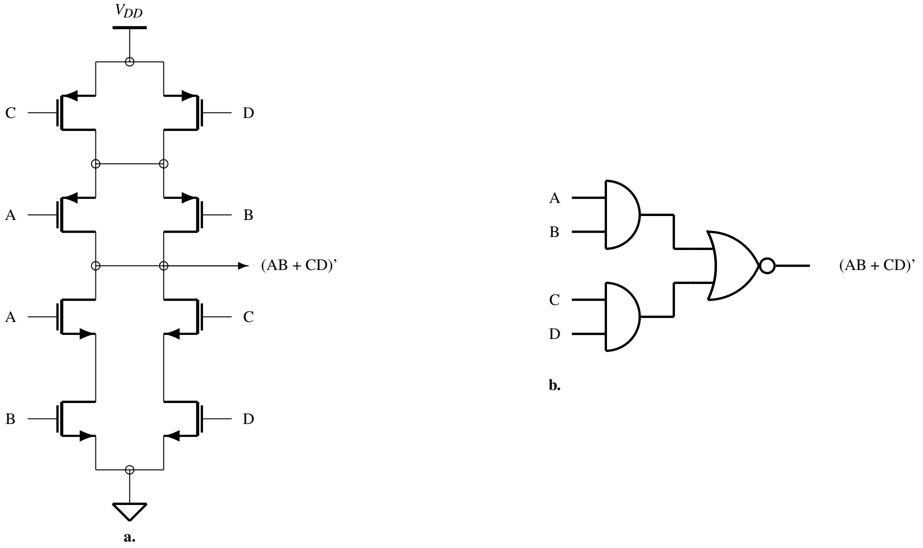


Figure C.18: **The AND-NOR gate. a**. The circuit. **b**. The logic symbol for the AND-NOR gate.

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.

C.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 C.19 and C.20.

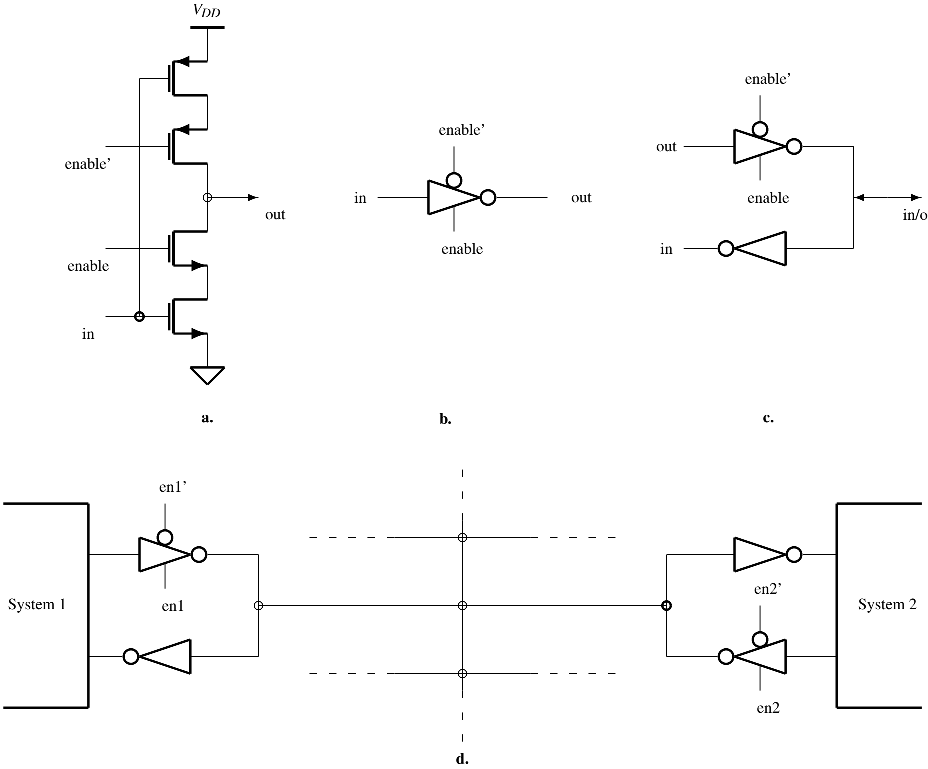


Figure C.19: **Tristate inverting buffer.** **a**. The circuit. **b**. The logic symbol for the inverting tristate buffer. **c.** Two-direction connection on one wire. For `enable = 1, in/out = out'`, while for `enable = 0, in = in/out'`. **d.** Interconnecting two systems. For `en1 = 1, en2 = 0`, System 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.

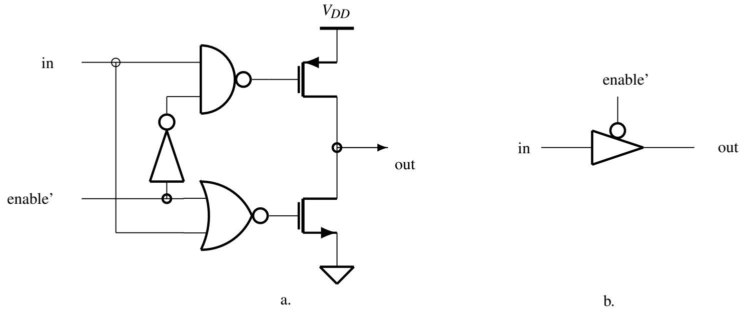


Figure C.20: **Tristate non-inverting buffer.** **a**. The circuit. **b**. The logic symbol for the non-inverting tristate buffer.

C.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 C.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 C.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.

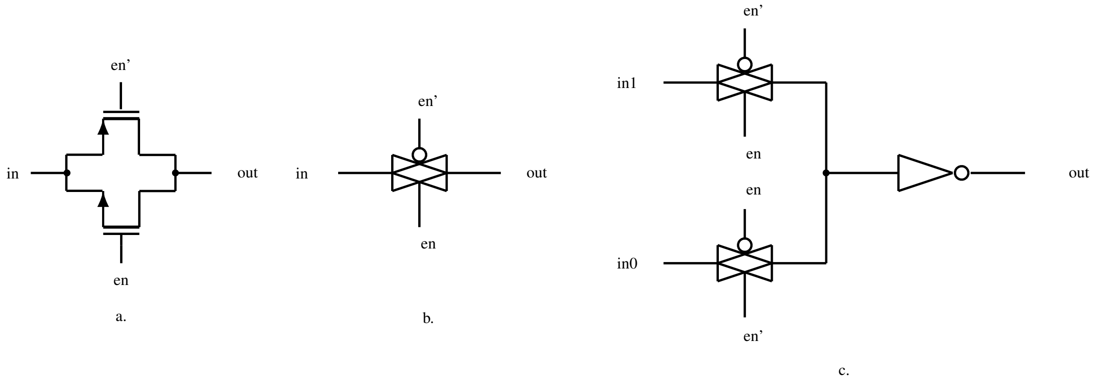


Figure C.21: **The transmission gate.** **a**. The complementary transmission gate. **b**. The logic symbol. **c**. An application: the elementary inverting multiplexer.

Appendix D

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 **??**).

D.1 Analog circuits

The operational amplifier is a concept that refers to an ideal circuit that is quite well approximated by real circuits.

Figure D.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}.

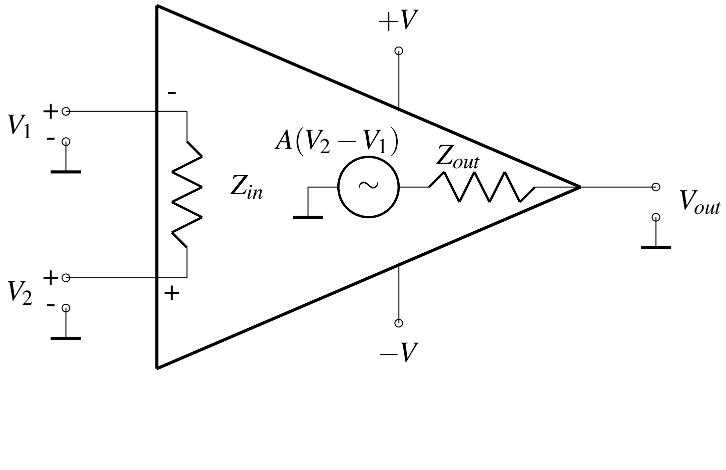


Figure D.1: Operational amplifier

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 D.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

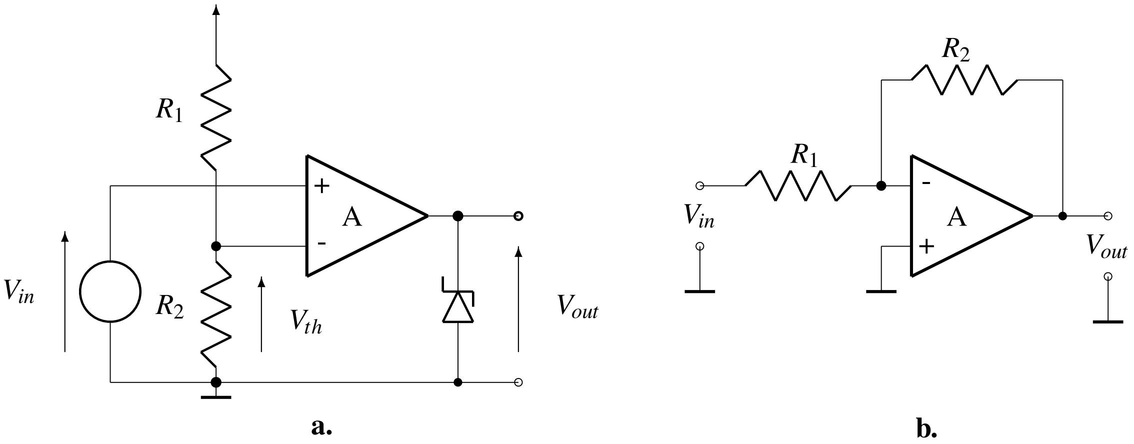


Figure D.2: Operational amplifier applications. **a.** Analog comparator. **b.** Amplifier.

$$V_{out} = A(V_{in} - V_{th})$$

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 D.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:

$$a = \frac{V_{out}}{V_{in}} = -\frac{R_2}{R_1}$$

D.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.

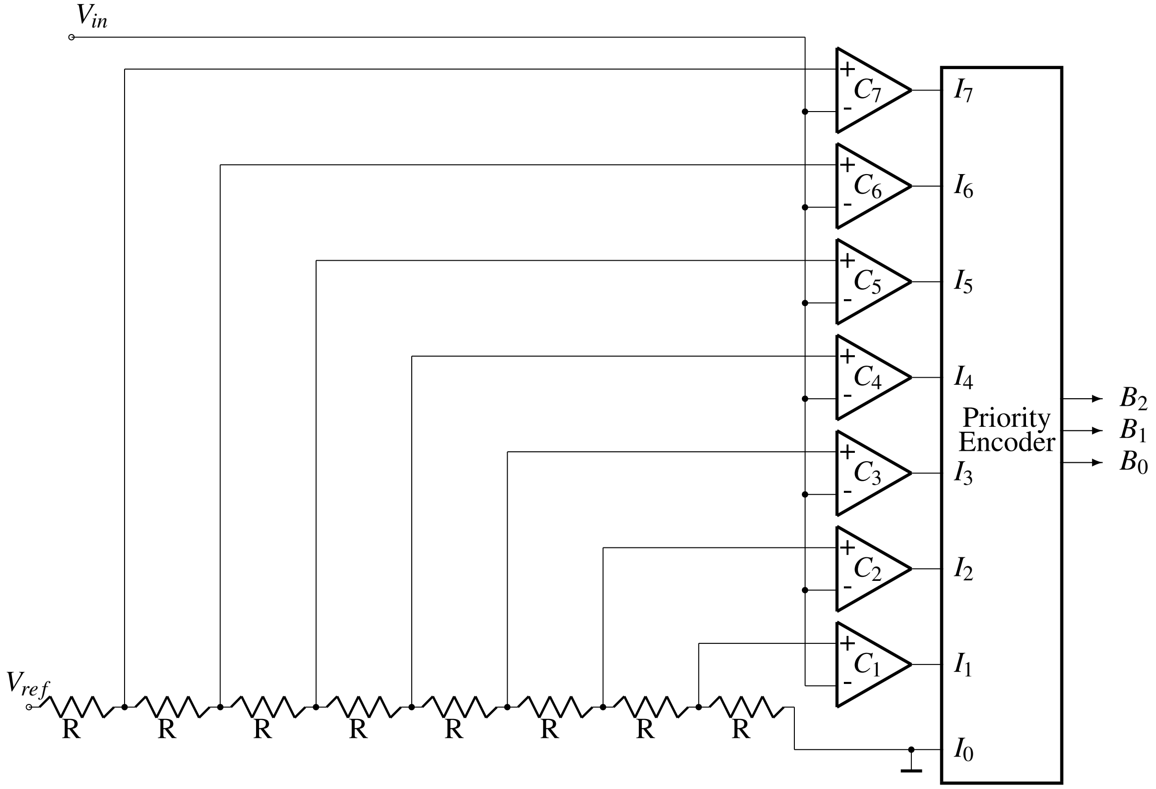


Figure D.3: ADC

D.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 D.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 D.4 results:

$$V_{out} = -V_{DD}(B_2/2^0 + B_1/2^1 + B_0/2^2)$$

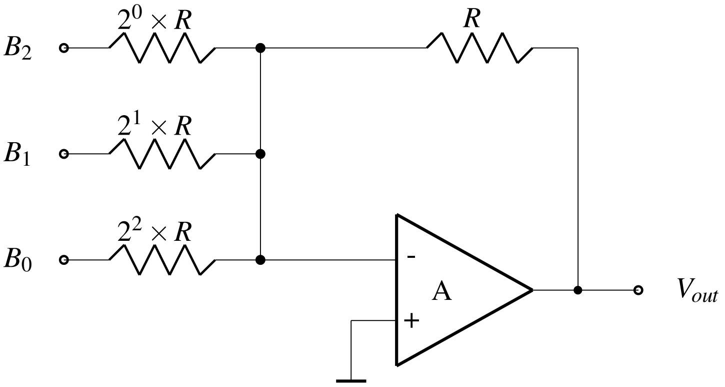


Figure D.4: DAC

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 199

[Chaitin '06] Gregory Chaitin: "The Limit of Rason", in *Scientific American*, Martie, 2006.

[Charboneau '20] Tyler Charboneau: "How "Master" and "Slave" Terminology is Being Reexamined in Electrical Engineering", *All About Circuits*, Oct. 06, 2020.

[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.europment.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.