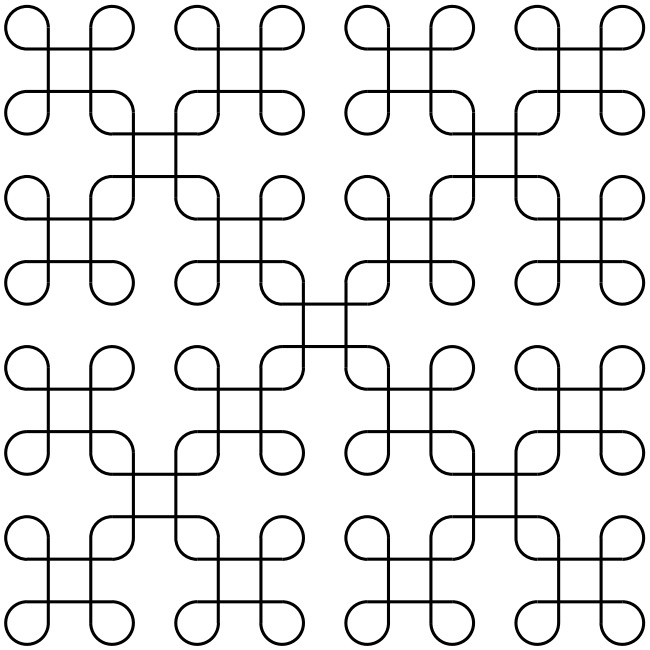
ADVANCED DIGITAL SYSTEMS

(work in progress)

Gheorghe M. Ştefan



– 2017 version –

Introduction

Acknowledgments

Contents

Chapter 1

GROWING & SPEEDING & FEATURING

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

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

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

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

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

Let us present in a systematic way, in the following subsections, the two mechanisms. Both are very

simple, but our goal is to emphasize, in the same time, some specific side effects as consequences of composing & looping, like the *pipeline connection* – used to accelerate the speed of the too deep circuits – or the *speculative mechanisms* – used to allow loops to be closed in pipelined structures.

Building a real circuit means mainly to interconnect simple and small components in order to *grow* an enough *fast* system appropriately *featured*. But, growing is a concept with no precise meaning. Many people do not make distinction between "growing the size" and "growing the complexity" of a system, for example. We will start making the necessary distinctions between "size" and "complexity" in the process of growing a digital system.

1.1 Size vs. Complexity

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

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

Definition 1.1 *The* **size** *of a digital circuit, $S_{digital\ circuit}$, is given by the dimension of the physical resources used to implement it.* $\diamond$

In order to provide a numerical expression for size we need a more detailed definition which takes into account technological aspects. In the '40s we counted electronic bulbs, in the '50s we counted transistors, in the '60s we counted SSI[1] and MSI[2] packages. In the '70s we started to use two measures: sometimes the number of transistors or the number of 2-input gates on the Silicon die and other times the Silicon die area. Thus, we propose two numerical measures for the size.

Definition 1.2 *The* **gate size** *of a digital circuit, $GS_{digital\ circuit}$, is given by the total number of CMOS pairs of transistors used for building the gates (see the appendix* Basic circuits*) used to implement it*[3]. $\diamond$

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

[1] *Small Size Integrated* circuits

[2] *Medium Size Integrated* circuits

[3] Sometimes gate size is expressed in the total number of 2-input gates necessary to implement the circuit. We prefer to count CMOS pairs of transistors (almost identical with the number of inputs) instead of equivalent 2-input gates because is simplest. Anyway, both ways are partially inaccurate because, for various reasons, the transistors used in implementing a gate have different areas.

Definition 1.3 *The* **area size** *of a digital circuit, $AS_{digital\ circuit}$, is given by the dimension of the area on Silicon used to implement it.* $\diamond$

The area size is useful to compute the price of the implementation because when a circuit is produced we pay for the number of wafers. If the circuit has a big area, the number of the circuits per wafer is small and the yield is low[4].

Definition 1.4 *The* **algorithmic complexity** *of a digital circuit, simply the* **complexity**, *$C_{digital\ circuit}$, has the magnitude order given by the minimal number of symbols needed to express its definition.* $\diamond$

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

Definition 1.5 *A* **simple circuit** *is a circuit having the complexity much smaller than its size:*

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

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

Definition 1.6 *A* **complex circuit** *is a circuit having the complexity in the same magnitude order with its size:*

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

Example 1.1 *The following Verilog program describes a* complex circuit*, because the size of its definition (the program) is*

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

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

$\diamond$

Example 1.2 *The following Verilog program describes a* simple circuit*, because the program that define completely the circuit is the same for any value of* n.

[4]The same number of errors make useless a bigger area of the wafer containing large circuits.

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

The prefixes of OR *circuit consists in* n OR_2 *gates connected in a very regular form. The definition is the same for any value of* n[5]. ◇

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

Definition 1.7 *The* depth *of a combinational circuit is equal with the total number of serially connected* constant *input gates (usually 2-input gates) on the longest path from inputs to the outputs of the circuit.* ◇

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

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

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

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

In the domain of circuit design we passed long ago beyond the stage of *minimizing* the number of gates in a few gates circuit. Now, the most important thing, in the multi-billion-transistor circuit era,

[5]A short discussion occurs when the dimension of the input is specified. To be extremely rigorous, the parameter *n* is expressed using a string o symbols in $O(\log n)$. But usually this aspect can be ignored.

is the *ability to describe*, by recursive definitions, simple (because we can't write huge programs), big (because we can produce more circuits on the same area) sized circuits. We must take into consideration that the Moore's Law applies to size not to complexity.

1.2 Time restrictions in digital systems

The most general form of a digital circuit (see Figure 1.1) includes both combinational and sequential behaviors. It includes two combinational circuits – (`comb_circ_1` and `comb_circ_2`) – and `register`. There are four critical propagation paths in this digital circuit:

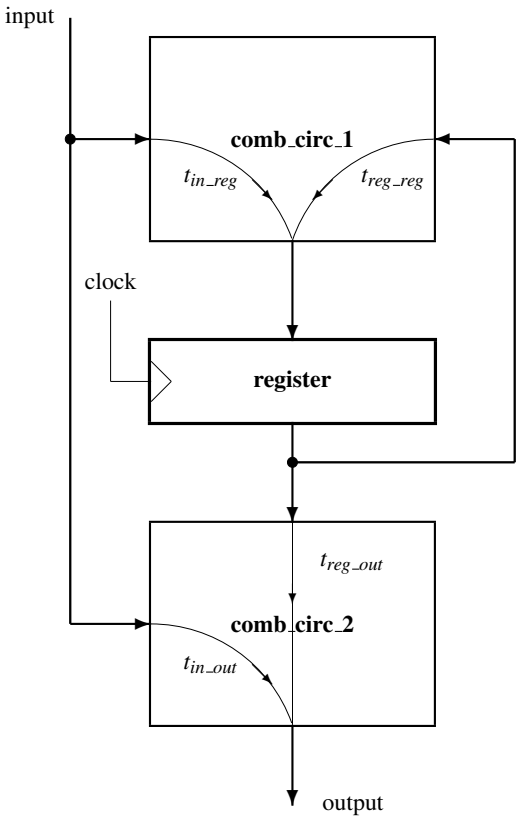


Figure 1.1: **The four critical propagation paths in digital circuits.** *Input-to-register* time (t_{in_reg}) is recommended to be as small as possible in order to reduce the time dependency from the previous sub-system. *Register-to-register* time (T_{min}) must be minimal to allow a high frequency for the clock signal. *Input-to-output* time (t_{in_out}) is good to be undefined to avoid hard to manage sub-systems interdependencies. *Register-to-output* time (t_{reg_out}) must be minimal to reduce the time dependency for the next sub-system

1. form input to register through `comb_circ_1`, which determines **minimum input arrival time before clock**: t_{in_reg}

2. from register to register through `comb_circ_1`, which determines **minimum period of clock**: $t_{reg\_reg} = T_{min}$, or **maximum frequency** of clock: $f_{max} = 1/T$
3. from input to output through `comb_circ_2`, which determines **maximum combinational path delay**: t_{in_out}
4. from register to output through `comb_circ_2`, which determines **maximum output required time after clock**: t_{reg_out}.

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

The loop is properly closed only if $T_{min} > t_{reg} + t_{cc_2} + t_{su}$ and $t_h < t_{reg} + t_{cc_2}$, where: t_{reg} is the propagation time through `register` from active edge of clock to output, and t_{cc_2} is the propagation time through `comb_circ_1` on the path 2.

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

Example 1.3 *Let us compute the propagation times for the four critical propagation paths of the counter circuit represented in Figure* **??**. *If we consider* $\#1 = 100ps$ *results:*

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

Example 1.4 *Let be the circuit from Figure 1.2, where:*

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

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

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

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

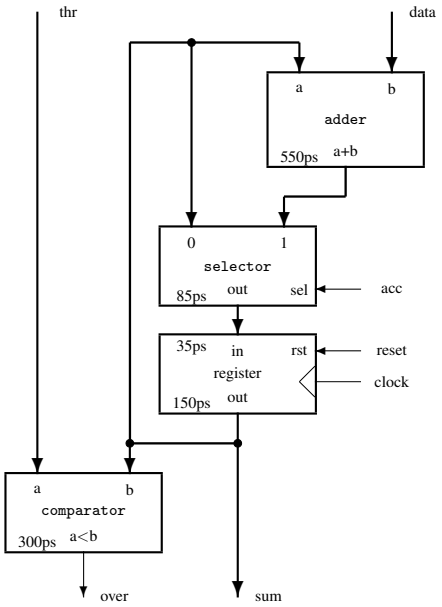


Figure 1.2: **Accumulate & compare circuit.** In the left-down corner of each rectangle is written the propagation time of each module. If `acc = 1` the circuit accumulates, else the content of `register` does not change.

- $t_{in\_out} = t_p(comparator) = 300ps$
- $t_{reg\_out} = t_p(register) + t_p(comparator) = 450ps$

◇

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

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

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

1.2.1 Pipelined connections

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

1. from input to output through: **it is not defined**

2. from register to output through: $t_{reg\_out} = t_{reg}$ – *it does not depend by the internal combinational structure of the module*, i.e., **the outputs are synchronous**, because they are generated directly from registers.

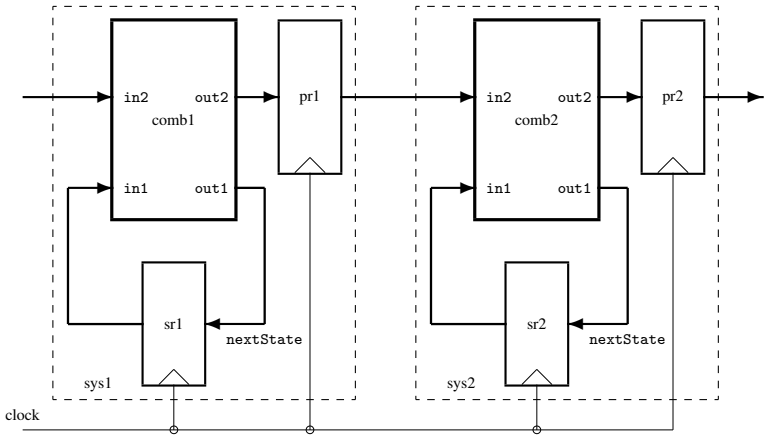


Figure 1.3: **Pipelined connections.**

Only two combinational paths are accepted: (1) from register to register, and (2) form input to register. In Figure 1.3 a generic configuration is presented. It is about two systems, `sys1` and `sys2`, pipeline connected using the output pipeline registers (`pr1` between `sys1` and `sys2`, and `pr2` between `sys2` and an external system). For the internal state are used the state registers `sr1` and `sr2`. The timing restrictions for the two combinational circuits `comb1` and `comb2` are not correlated. The maximum clock speed for each system does not depend by the design restrictions imposed for the other system.

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

The system from Figure 1.3 is descried by the following code.

```
module pipelineConnection( output  [15:0]  out    ,
                           input   [15:0]  in     ,
                           input           clock  );
   wire    [15:0]  pipeConnect ;
   sys sys1(   .pr     (pipeConnect),
               .in     (in         ),
               .clock  (clock      )),
       sys2(   .pr     (out        ),
               .in     (pipeConnect),
               .clock  (clock      ));
endmodule

module sys(output  reg [15:0]  pr      ,
```

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

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

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

- the **blocking assignment**, $=$: the whole statement is done before control passes to the next
- the ***non-blocking assignment***, $<=$: evaluate **all** the right-hand sides in the project for the current time unit and assign the left-hand sides only at the end of the time unit.

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

VeriSim 1.1 *The following simulation used 6 clocked registers. All of them switch on the positive edge. But, the code is written for three of them using the* **blocking assignment***, while for the other three using the* **non-blocking assignment***. The resulting behavior show us the difference between the two clock triggered assignment. The blocking assignment seems to be useless, because propagates the input through all the three registers in one clock cycle. The non-blocking assignment shifts the input along the three serially connected registers clock by clock. This second behavior can be used in real application to obtain clock controlled delays.*

```
module blockingNonBlocking(output  reg [1:0]   blockingOut     ,
                           output  reg [1:0]   nonBlockingOut  ,
                           input       [1:0]   in              ,
                           input               clock           );

   reg [1:0]   reg1, reg2, reg3, reg4;

   always @(posedge clock) begin   reg1            = in    ;
                                   reg2            = reg1  ;
```

```
                                    blockingOut     = reg2  ;   end

    always @(posedge clock) begin   reg3            <= in   ;
                                    reg4            <= reg3 ;
                                    nonBlockingOut  <= reg4 ;   end
endmodule

module blockingNonBlockingSimulation;
    reg             clock                       ;
    reg     [1:0]   in                          ;
    wire    [1:0]   blockingOut, nonBlockingOut ;
    initial begin               clock = 0       ;
                    forever #1  clock = ~clock  ;   end
    initial begin       in = 2'b01  ;
                    #2  in = 2'b10  ;
                    #2  in = 2'b11  ;
                    #2  in = 2'b00  ;
                    #7  $stop       ;   end
    blockingNonBlocking dut(blockingOut     ,
                            nonBlockingOut  ,
                            in              ,
                            clock           );
    initial
    $monitor
    ("clock=%b in=%b reg1=%b reg2=%b blockingOut=%b reg3=%b reg4=%b nonBlockingOut=%b",
    clock, in, dut.reg1, dut.reg2, blockingOut, dut.reg3, dut.reg4, nonBlockingOut);
endmodule
```

```
/*
 clock=0 in=01  reg1=xx reg2=xx blockingOut=xx    reg3=xx reg4=xx nonBlockingOut=xx
 clock=1 in=01  reg1=01 reg2=01 blockingOut=01    reg3=01 reg4=xx nonBlockingOut=xx
 clock=0 in=10  reg1=01 reg2=01 blockingOut=01    reg3=01 reg4=xx nonBlockingOut=xx
 clock=1 in=10  reg1=10 reg2=10 blockingOut=10    reg3=10 reg4=01 nonBlockingOut=xx
 clock=0 in=11  reg1=10 reg2=10 blockingOut=10    reg3=10 reg4=01 nonBlockingOut=xx
 clock=1 in=11  reg1=11 reg2=11 blockingOut=11    reg3=11 reg4=10 nonBlockingOut=01
 clock=0 in=00  reg1=11 reg2=11 blockingOut=11    reg3=11 reg4=10 nonBlockingOut=01
 clock=1 in=00  reg1=00 reg2=00 blockingOut=00    reg3=00 reg4=11 nonBlockingOut=10
 clock=0 in=00  reg1=00 reg2=00 blockingOut=00    reg3=00 reg4=11 nonBlockingOut=10
 clock=1 in=00  reg1=00 reg2=00 blockingOut=00    reg3=00 reg4=00 nonBlockingOut=11
 clock=0 in=00  reg1=00 reg2=00 blockingOut=00    reg3=00 reg4=00 nonBlockingOut=11
 clock=1 in=00  reg1=00 reg2=00 blockingOut=00    reg3=00 reg4=00 nonBlockingOut=00
 clock=0 in=00  reg1=00 reg2=00 blockingOut=00    reg3=00 reg4=00 nonBlockingOut=00
*/
```

It is obvious that the registers `reg1` *and* `reg2` *are useless because they are somehow "transparent".*
◇

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

VerilogSummary 1 :

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

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

1.2.2 Fully buffered connections

The most safe approach, the **synchronous** one, supposes fully registered inputs and outputs (see Figure 1.4 where the functionality is implemented using combinatorial circuits and registers and the interface with the rest of the system is implemented using only `input register` and `output register`).

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

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

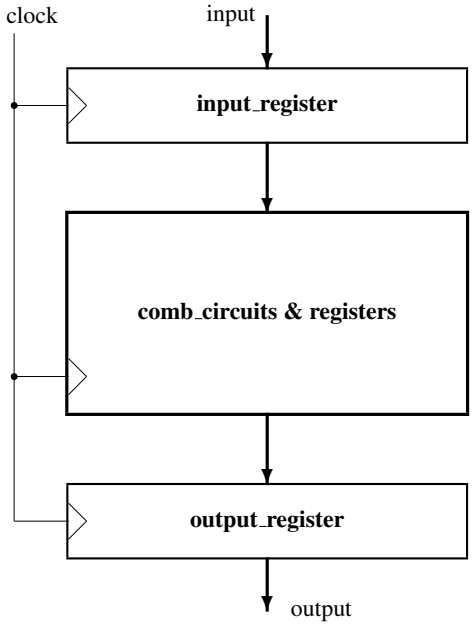


Figure 1.4: **The general structure of a module in a complex digital system.** If any big module in a complex design is buffered with input and output registers, then we are in the ideal situation when: t_{in_reg} and t_{reg_out} are minimized and t_{in_out} is not defined.

For the synchronously interfaced module represented in Figure 1.4 the timing restrictions are the following:

1. form input to register: $t_{in\_reg} = t_{su}$ – *it does not depend by the internal structure of the module*
2. from register to register: T_{min}, and $f_{max} = 1/T$ – *it is a system parameter*
3. from input to output through: **it is not defined**

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

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

1.3 Growing the size by composition

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

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

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

Definition 1.8 *The* **composition** *(see Figure 1.5) is a two level construct, which performs the function f using on the second level the m-ary function g and on the first level the functions* `h_1, h_2, ... h_m`*, described by the following, incompletely defined, but synthesisable, Verilog modules.*

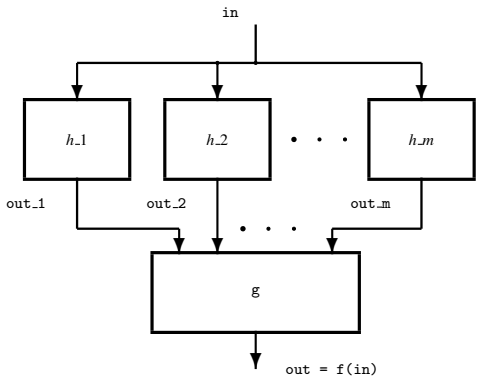


Figure 1.5: **The circuit performing composition.** The function g is composed with the functions `h_1, ... h_m` using a two level circuit. The first level contains m circuits computing *in parallel* the functions `h_i`, and on the second level there is the circuit computing the *reduction*-like function `g`.

```
module f #(`include "parameters.v")(    output  [sizeOut-1:0]   out ,
                                        input   [sizeIn-1:0]    in  );
   wire     [size_1-1:0]  out_1;
   wire     [size_2-1:0]  out_2;
```

```
    // ...
    wire    [size_m-1:0]  out_
    g   second_level(   .out    (out    ),
                        .in_1   (out_1  ),
                        .in_2   (out_2  ),
                        // ...
                        .in_m   (out_m  ));
    h_1 first_level_1(.out(out_1), .in(in));
    h_2 first_level_2(.out(out_2), .in(in));
    // ...
    h_m first_level_m(.out(out_m), .in(in));
endmodule

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

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

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

// ...

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

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

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

◇

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

Definition 1.9 *The* **serial composition** *(see Figure 1.6a) is the composition with* $m = 1$. *Results the Verilog description:*

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

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

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

◇

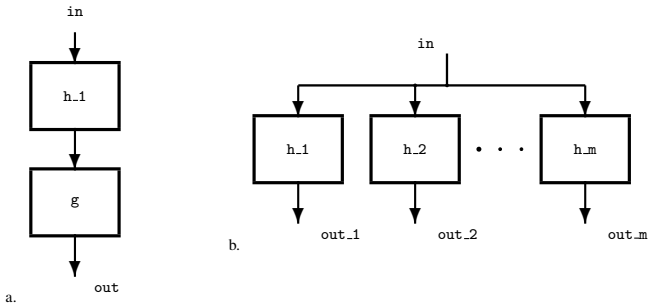


Figure 1.6: **The two limit forms of composition.** a. The serial composition, for $m = 1$, imposing an inherent sequential computation. b. The parallel composition, with no *reduction*-like function, performing data parallel computation.

Definition 1.10 *The* **parallel composition** *(see Figure 1.6b) is the composition in the particular case when g is the* identity function*. Results the following Verilog description:*

```
module f #(`include "parameters.v")(    output  [sizeOut-1:0]    out ,
                                        input   [sizeIn-1:0]     in  );
   wire     [size_1-1:0]  out_1;
   wire     [size_2-1:0]  out_2;
   // ...
   wire     [size_m-1:0]  out_m;

   assign out = {  out_m,
                   // ...
                   out_2,
                   out_1}; // g is identity function

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

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

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

// ...

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

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

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

◇

Example 1.5 *Using the mechanism described in Definition 1.3 the circuit computing the* scalar product *between two 4-component vectors will be defined, now in true Verilog. The test module for* `n = 8` *is also defined allowing to test the design.*

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

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

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

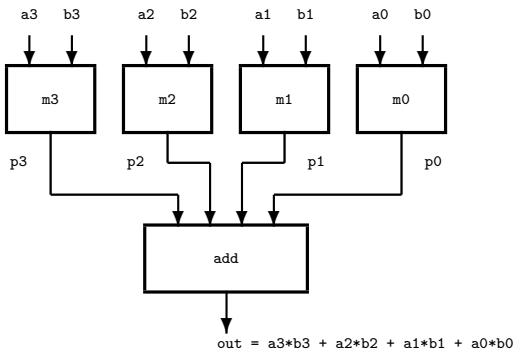


Figure 1.7: **An example of composition.** The circuit performs the scalar vector product for 4-element vectors. The first level compute in parallel 4 multiplications generating the vectorial product, and the second level *reduces* the resulting vector of products to a scalar.

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

```
parameter n = 8
```

The simulation is done by running the module:

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

 initial   begin   {a3, a2, a1, a0} = {8'd1, 8'd2, 8'd3, 8'd4};
                   {b3, b2, b1, b0} = {8'd5, 8'd6, 8'd7, 8'd8};
           end
 inner_prod  dut(out, a3, a2, a1, a0, b3, b2, b1, b0);
 initial $monitor("out=%0d", out);
endmodule
```

The test outputs: `out = 70`

The description is structural at the top level and behavioral for the internal sub-modules (corresponding to our level of understanding digital systems). The resulting circuit is represented in Figure 1.7. ◇

VerilogSummary 2 :

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

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

1.4 Speeding by pipelining

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

$$t_f = max(t_{h\_1}, \ldots, t_{h\_m}) + t_g$$

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

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

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

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

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

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

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

The typical case is given by the serial composition (see Figure 1.6a), where the function $out = f(in) = g(h\_1(in))$ must be computed using 2 serial connected circuits, $h\_1(in)$ and $g(int\_out)$, in time:

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

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

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

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

1.4.1 Register transfer level

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

The general structure of a system performing the function $f(x)$ is shown in Figure 1.8a, where it is presented in the fully buffered version. This kind of approach is called **register transfer level** (RTL) because data is transferred, modified by the function f, from a register, `input_reg`, to another register, `output_reg`. If $f = g(h\_1(x))$, then the clock frequency is limited to:

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

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

1.4.2 Pipeline structures

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

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

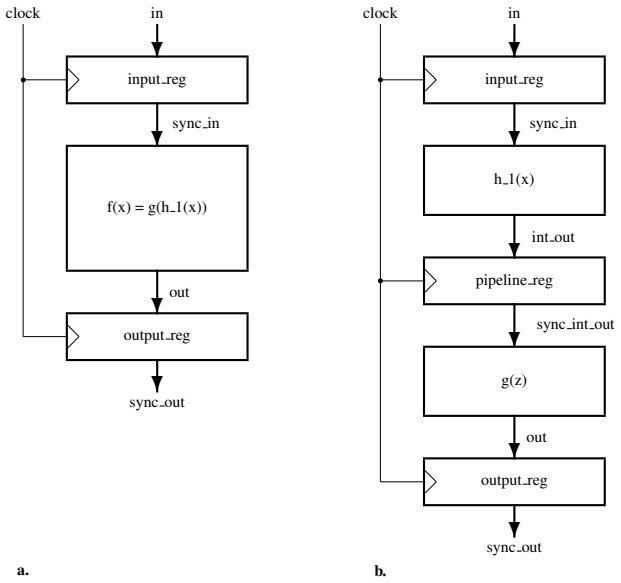


Figure 1.8: **Pipelined computation. a**. A typical Register Transfer Logic (RTL) configuration. Usually it is supposed a "deep" combinational circuit computes $f(x)$. **b**. The pipeline structure splits the combinational circuit associated with function $f(x)$ in two less "deep" circuits and inserts the *pipeline register* in between.

The only way to increase the clock rate is to divide the circuit designed for f in two serially connected circuits, one for h_1 and another for g, and to introduce between them a new register. Results the system represented in Figure 1.8b. Its clock frequency is:

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

and the processing time for the same string is:

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

The two systems represented in Figure 1.8 are equivalent. The only difference between them is that the second performs the processing in $n+3$ clock cycles instead of $n+2$ clock cycles for the first version. For big n, the current case, this difference is a negligible quantity. We call **latency** the number of the additional clock cycle. In this first example latency is: $\lambda = 1$.

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

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

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

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

$$\lambda = m << n$$

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

Example 1.6 *The pseudo-Verilog code for the 2-stage pipeline system represented in Figure 1.8 is:*

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

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

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

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

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

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

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

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

1.4.3 Data parallelism vs. time parallelism

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

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

The data parallelism is more obvious: m functions, $h\_1,\ldots,h\_m$, are performed in parallel by m circuits (see Figure 1.6b). But, time parallelism is not so obvious. It acts only in a *pipelined serial composition*, where the first stage is involved in computing the most recently received data, the second stage is involved in computing the previously received data, and so on. In an $(m+1)$-stage pipeline structure $m+1$ elements of the input stream are in different stages of computation, and at each clock cycle one result is provided. We can claim that in such a pipeline structure $m+1$ computations are done in parallel with the price of a latency $\lambda = m$.

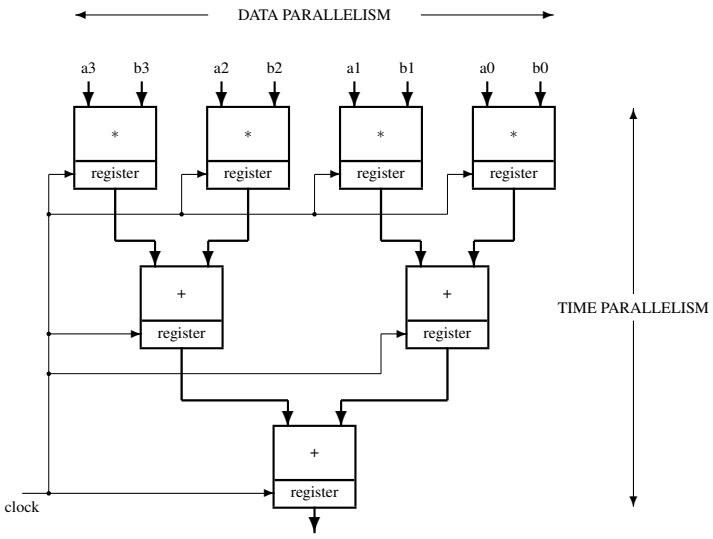


Figure 1.9: **The pipelined inner product circuit for 4-component vectors.** Each multiplier and each adder send its result in a pipeline register. For this application results a three level pipeline structure with different degree of parallelism. The two kind of parallelism are exemplified. Data parallel has the maximum degree on the first level. The degree of time parallelism is three: in each clock cycle three pairs of 4-element vectors are processed. One pair in the first stage of multiplications, another pair is the second stage of performing two additions, and one in the final stage of making the last addition.

The previous example of a 2-stage pipeline accelerated the computation because of the time parallelism which allows to work simultaneously on two input data, on one applying the function h_1 and in another applying the function g. Both being simpler than the global function f, the increase of clock frequency is possible allowing the system to deliver results at a higher rate.

Computer scientists stress on both type of parallelism, each having its own fundamental limitations. More, each form of parallelism bounds the possibility of the other, so as the parallel processes are strictly limited in now a day computation. But, for us it is very important to emphasize in this stage of the approach that:

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

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

Example 1.7 *Let us revisit the problem of computing the scalar product. We redesign the circuit in a pipelined version using only binary functions.*

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

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

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

◇

The structure of the pipelined *inner product* (dot product) circuit is represented in Figure 1.9. It shows us the two dimensions of the parallel computation. The horizontal dimension is associated with *data parallelism*, the vertical dimension is associated with *time parallelism*. The first stage allows 4 parallel computation, the second allows 2 parallel computation, and the last consists only in a single addition. The mean value of the **degree of data parallelism** is 2.33. The system has latency 2, allowing 7 computations in parallel. The **peak performance** of this system is the **whole degree of parallelism** which is 7. The peak performance is the performance obtained if the input stream of data is uninterrupted. If it is interrupted because of the lack of data, or for another reason, the latency will act reducing the peak performance, because some or all pipeline stages will be inactive.

1.5 Featuring by closing new loops

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

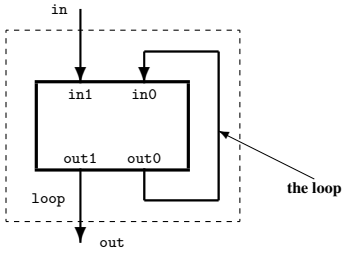


Figure 1.10: **The loop closed into a digital system.** The initial system has two inputs, `in1` and `in0`, and two outputs, `out1` and `out0`. Connecting `out0` to `in0` results a new system with `in` and `out` only.

Definition 1.11 *The loop consists in connecting some outputs of a system to some of its inputs (see Figure 1.10), as in the pseudo-Verilog description that follows:*

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

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

◇

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

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

Example 1.8 *Let be a synchronous adder. It has the outputs synchronized with an positive edge clocked register (see Figure 1.11a). If the output is connected back to one of its input, then results the structure of an accumulator (see Figure 1.11b). The Verilog description follows.*

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

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

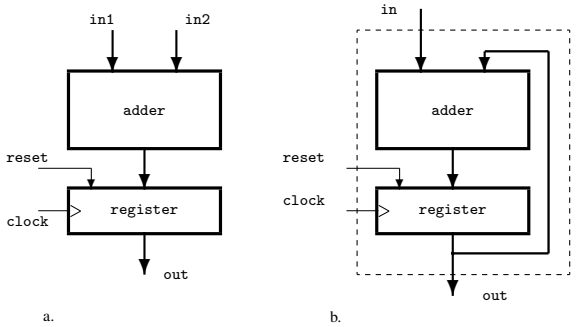


Figure 1.11: **Example of loop closed over an adder with synchronized output.** If the output becomes one of the inputs, results a circuit that accumulates at each clock cycle. **a**. The initial circuit: the synchronized adder. **b**. The resulting circuit: the accumulator.

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

```
module test_acc;
   reg clock, reset;
   reg[15:0] in;
   wire[19:0] out;
   initial begin   clock = 0;
                   forever #1 clock = ~clock;
           end // the clock
   initial begin   reset = 1;
                   #2 reset = 0;
                   #10 $stop;
           end
  always @(posedge clock) if (reset)   in = 0;
                                else   in = in + 1;
  acc  dut(out, in, clock, reset);
  initial $monitor("time=%0d clock=%b in=%d out=%d",
                   $time, clock, in, dut.out);
endmodule
```

By simulation results the following behavior:

```
# time=0 clock=0 in=    x out=      x
# time=1 clock=1 in=    0 out=      0
# time=2 clock=0 in=    0 out=      0
# time=3 clock=1 in=    1 out=      1
# time=4 clock=0 in=    1 out=      1
# time=5 clock=1 in=    2 out=      3
# time=6 clock=0 in=    2 out=      3
# time=7 clock=1 in=    3 out=      6
# time=8 clock=0 in=    3 out=      6
# time=9 clock=1 in=    4 out=     10
# time=10 clock=0 in=   4 out=     10
# time=11 clock=1 in=   5 out=     15
```

◇

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

1.5.1 Data dependency

The good news about loop is its "ability" to add new features. But any good news is accompanied by its own bad news. In this case is about the limiting of the degree of parallelism allowed in a system with a just added loop. It is mainly about the necessity to **stop** sometimes the input stream of data in order

to decide, inspecting an output, how to continue the computation. The input data waits for data arriving from an output a number of clock cycles related with the system latency. To do something special the system must be allowed to accomplish certain internal processes.

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

Example 1.9 *Let be the system performing the following function:*

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

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

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

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

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

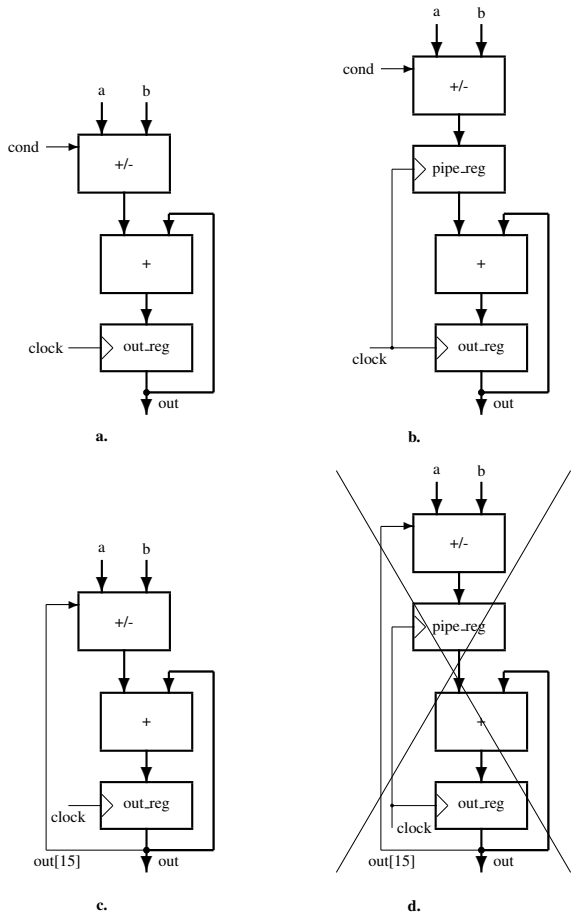


Figure 1.12: **Data dependency when a loop is closed in a pipelined structure. a.** The non-pipelined version. **b.** The pipelined version. **c.** Adding a loop to the non-pipelined version. **d.** To the pipelined version the loop can not be added without supplementary precautions because *data dependency* change the overall behavior. The selection between add and sub, performed by the looped signal comes too late.

```
                else        pipe <= a - b;
              out <= out + pipe;
        end
endmodule
```

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

```
module addapt_acc0(output  [15:0]  out,
                   input   [15:0]  a, b,
                   input           reset, clock);
  cond_acc0  cont_acc0(.out    (out),
                       .a      (a),
                       .b      (b),
                       .cond   (out[15]),    // the loop
                       .reset  (reset),
                       .clock  (clock));
endmodule
```

Figure 1.12a represents the first implementation of the `cond_acc` *circuit, characterized by a low clock frequency because both the* adder *and the* adder/subtracter *contribute to limiting the clock frequency:*

$$f_{clock} = \frac{1}{t_{+/-} + t_{+} + t_{reg}}$$

Figure 1.12b represents the pipelined version of the same circuit working faster because only one from adder *and the* adder/subtracter *contributes to limiting the clock frequency:*

$$f_{clock} = \frac{1}{max(t_{+/-}, t_{+}) + t_{reg}}$$

A small price is paid by $\lambda = 1$.

The 1-bit loop closed from the output `out[15]` *to* `cond` *input (see Figure 1.12c) allows the circuit* to decide itself *if the sum or the difference is accumulated. Its speed is identical with the initial, no-loop, circuit.*

Figure 1.12d warns us against the expected damages of closing a loop in a pipelined system. Because of the latency the "decision comes" to late and the functionality is altered. $\diamond$

In the system from Figure 1.12a the degree of parallelism is 1, and in Figure 1.12b the system has the degree of parallelism 2, because of the pipeline execution. When we closed the loop we where obliged to renounce to the bigger degree of parallelism because of the latency associated with the pipe. We have a new functionality – the circuit decides itself regarding the function executed in each clock cycle – but we must pay the price of reducing the speed of the system.

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

The circuit from Figure 1.12d is not a solution for our problem because the condition `cond` comes to late. It corresponds to the operation executed on the input stream excepting the most recently received pair of data. The condition comes too late, with a delay equal with the latency introduced by the pipeline execution.

1.5.2 Speculating to avoid limitations imposed by data dependency

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

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

Example 1.10 *The pipelined version of the circuit* `addapt_acc` *is provided by the following Verilog code:*

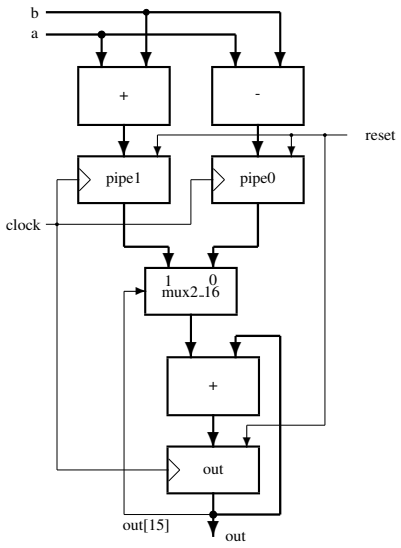


Figure 1.13: **The speculating solution to avoid data dependency.** In order to delay the moment of decision both addition and subtract are computed on the first stage of pipeline. Speculating means instead to decide what to do, addition or subtract, we decide what to consider after doing both.

```
module addapt_acc1(output  reg [15:0]  out,
                   input       [15:0]  a, b,
                   input               reset, clock);
 reg    [15:0]  pipe1, pipe0;
 always @(posedge clock)
   if (reset)  begin   out <= 0;
                       pipe1 <= 0;
                       pipe0 <= 0;
               end
    else   begin   pipe1 <= a + b;
                   pipe0 <= a - b;
                   if (out[15])     out <= out + pipe1;
```

```
                    else            out <= out + pipe0;
            end
endmodule
```

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

$$f_{clock} = \frac{1}{max(t_+, t_-, (t_+ + t_{mux})) + t_{reg}} \simeq \frac{1}{t_- + t_{reg}}$$

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

Roughly speaking, the price for the speed is: an adder & two registers & a multiplexer *(see for comparing Figure 1.12c and Figure 1.13). Sometimes it deserves!* $\diamond$

The procedure applied to design `addapr_acc1` involves the multiplication of the physical resources. We *speculated*, computing on the first level of pipe both the sum and the difference of the input values. On the second state of pipe the multiplexer is used to select the appropriate value to be added to `out`.

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

1.6 Concluding about composing & pipelining & looping

The basic ideas exposed in this section are:

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

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

- MIMD (multiple-instructions-multiple data), which means mainly having different programs working on different data
- SIMD (single-instructions-multiple-data), which means having one program working on different data,
- MISD (multiple-instructions-single-data), which means having different programs working on the same data.

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

- time parallelism, which is *somehow related* with MIMD execution, because in each temporal stage a different operation (instruction) can be performed
- data parallelism, which is identic with SIMD execution
- speculative parallelism, which is a sort of MISD execution.

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

1.7 Problems

Speculative circuits

Problem 1.1 *Let be the circuit described by the following Verilog module:*

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

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

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

Problem 1.2 *Provide the speculative solution for the next circuit.*

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

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

Problem 1.3 *The following circuit has two included loops. The speculation will increase the dimension of the circuit accordingly. Provide the speculative version of the circuit.*

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

    reg     [15:0]  acc;

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

1.8 Projects

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

Project 1.1

Chapter 2

THE TAXONOMY OF DIGITAL SYSTEMS

2.1 Loops & Autonomy

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

Example 2.1 *Let be the digital system* `initSyst` *from Figure 2.1a, with two inputs,* `in`*,* `lp`*, and one output,* `out`*. What hapend when is closed the loop from the output* `out` *to the input* `lp`*? Let's make it. The following Verilog modules describe the behavior of the resulting circuit.*

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

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

   initial out = 2'b11; // only for simulation purpose

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

```
endmodule
```

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

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

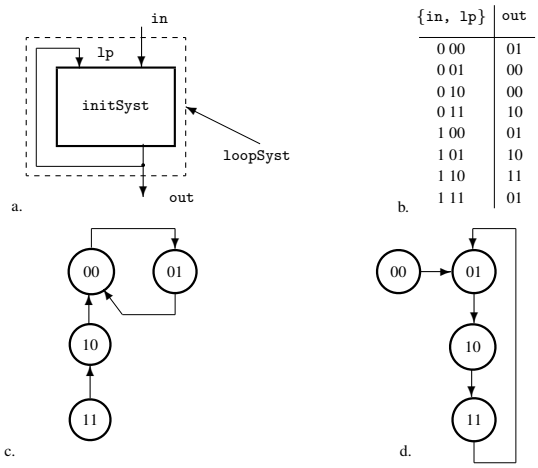


Figure 2.1: **Example illustrating the autonomy. a.** A system obtained from an initial system in which a loop is closed from output to one of its input. **b.** The transition table of the initial system where each output strict corresponds to the input value. **c.** The output evolution for constant input: `in = 0`. **d.** The output evolution for a different constant input: `in = 1`.

The simulation offers us the following behavior:

```
# time=0  in=0 out=11
# time=1  in=0 out=10
# time=2  in=0 out=00
# time=3  in=0 out=01
# time=4  in=0 out=00
# time=5  in=0 out=01
# time=6  in=0 out=00
# time=7  in=0 out=01
# time=8  in=0 out=00
```

```
# time=9  in=0 out=01
# time=10 in=1 out=00
# time=11 in=1 out=01
# time=12 in=1 out=10
# time=13 in=1 out=11
# time=14 in=1 out=01
# time=15 in=1 out=10
# time=16 in=1 out=11
# time=17 in=1 out=01
# time=18 in=1 out=10
# time=19 in=1 out=11
```

The main effect we want to emphasize is the evolution of the output under no variation *of the input* in. *The initial system, defined in the previous* `case`*, has an output that switches only responding to the input changing (see also the table from Figure 2.1b). The system which results closing the loop has its own behavior. This behavior depends by the input value, but is triggered by the events coming through the loop. Figure 2.1c shows the output evolution for* `in = 0` *and Figure 2.1d represents the evolution for* `in = 1`. ◇

VerilogSummary 3 :

- the register `reg[1:0] out` defined in the module `initSyst` is nor a register, it is a Verilog variable, whose value is computed by a `case` procedure anytime at least one of the two inputs change (`always @(in or lp)`)
- a register which changes its state "ignoring" a clock edge is not a register, it is a variable evolving like the output of a combinational circuit
- what is the difference between an `assign` and an `always (a or b or ...)`? The body of `assign` is continuously evaluated, rather than the body of `always` which is evaluated only if at least an element of the list of sensitivity (`(a or b or ...)`) changes
- in running a simulation an `assign` is more computationally costly in time than an `always` which is more costly in memory resources.

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

Definition 2.1 *In a digital system a behavior is called* **autonomous** *iff for the same input dynamic there are defined more than one distinct output transitions, which manifest in distinct moments.* ◇

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

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

Theorem 2.1 *In the respect of the previous definition for autonomy, closing an internal loop generates autonomous behaviors.* ◇

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

```
always @(in1 or in0) #1
   case (in1)
       ... :   case (in0)
                   ... : {out1, out0} = f_00(in1, in0);
                         ...
                   ... : {out1, out0} = f_0p(in1, in0);
               endcase
                   ...
       ... :   case (in0)
                   ... : {out1, out0} = f_q0(in1, in0);
                         ...
                   ... : {out1, out0} = f_qp(in1, in0);
               endcase
   endcase
```

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

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

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

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

◇

2.2 Classifying Digital Systems

The two mechanisms, of composing and of "looping", give us a very good instrument for a new classification of digital systems. If the system grows by different *compositions*, then it allows various kinds of *connections*. In this context the loops are difficult to be avoided. They occur sometimes in large systems

without the explicit knowledge of the designer, disturbing the design process. But, usually we design being aware of the effect introduced by this special connection – the loop. This mechanism leads us to design a complex network of loops which include each other. Thus, in order to avoid ambiguities in using the loops we must define what means "included loop". We shall use frequently in the next pages this expression for describing how digital systems are built.

Definition 2.2 *A loop includes another loop only when it is closed over a serial or a serial-parallel composition which have at least one subsystem containing an internal loop, called an included loop.* ◇

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

Example 2.2 *In Figure 2.2 the loop (1) is included by the loop (2). In a serial composition built with S_1 and S_2 interconnected by (3), we use the connection (2) to add a new loop.* ◇

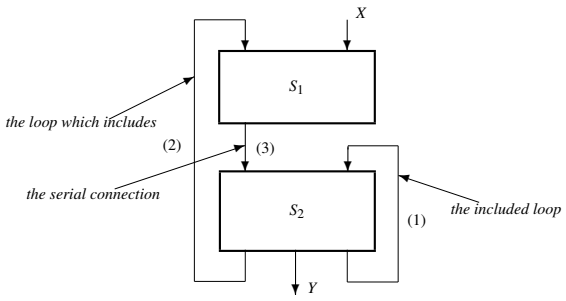


Figure 2.2: **Included loops.** The loop (2) includes loop (1), closed over the subsystem S_2, because S_2 is serially connected with the subsystem S_1 and loop (2) includes both S_1 and S_2.

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

Definition 2.3 *Let be a n-order system,* `n-OS`*. A* `(n+1)-OS` *can be built only adding a new loop which includes the first n loops. The* `0-OS` *contains only combinational circuits (the loop-less circuits).* ◇

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

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

0-OS - combinational circuits, with no autonomy

1-OS - memories, having the autonomy of internal state

2-OS - automata, with the autonomy to sequence

3-OS - processors, with the autonomy to control

4-OS - computers, with the autonomy to interpret

...

n-OS - systems with the highest autonomy: to self-organize.

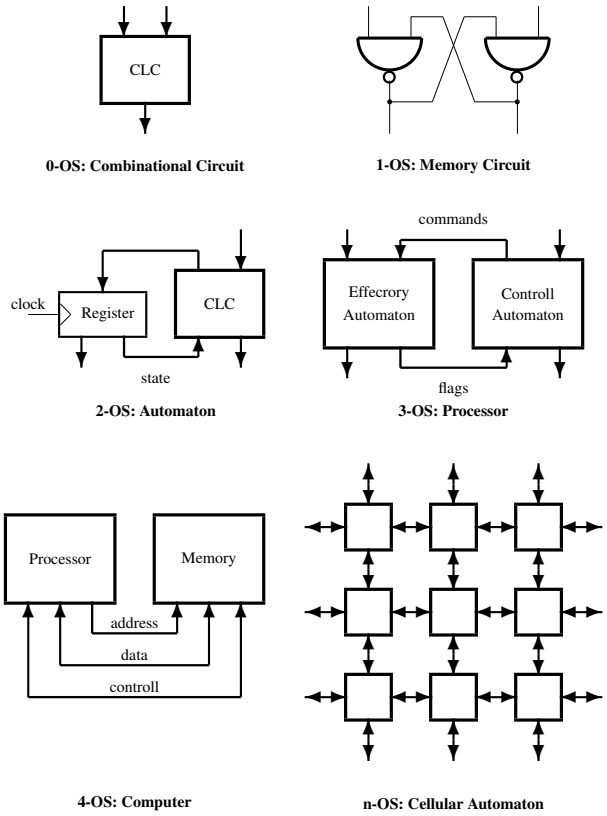


Figure 2.3: **Examples of circuits belonging to different orders.** A combinational circuit is in 0-OS class because has no loops. A memory circuit contains one-loop circuits and therefore it is in 1-OS class. Because the register belongs to 1-OS class, closing a loop containing a register and a combinational circuit (which is in 0-OS class) results an automaton: a circuit in 2-OS class. Two loop connected automata – a circuit in 3-OS class – can work as a processor. An example of 4-OS is a simple computer obtained loop connecting a processor with a memory. Cellular automata contains a number of loops related with the number of automata it contains.

This new classification can be exemplified[1] (see also Figure 2.3) as follows:

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

[1]For almost all the readers the following enumeration is now meaningless. They are kindly invited to revisit this end of chapter after assimilating the first 7 chapter of this book.

- 1-OS: elementary latch, master-slave flip-flop (serial composition), random access memory (parallel composition), register (serial-parallel composition), ...
- 2-OS: T flip-flop (the simplest two states automaton), J-K flip-flop (the simplest two input automaton), counters, automata, finite automata, ...
- 3-OS: automaton using loop closed through K-J flip-flops or counters, stack-automata, elementary processors, ...
- 4-OS: micro-controller, computer (as Processor & RAM loop connected), stack processor, co-processor
- ...
- n-OS: cellular automaton.

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

2.3 Digital Super-Systems

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

2.4 Preliminary Remarks On Digital Systems

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

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

Composing circuits & closing loops A big circuit results *composing* many small ones. A new kind of feature can be added only closing a new *loop*. The structural composing corresponds the the mathematical concept of composition. The loop corresponds somehow to the formal mechanism of recursion. Composing is an "additive" process which means to put together different simple function to obtain a

bigger or a more complex one. Closing a loop new behaviors occur. Indeed, when a snake eats a mouse nothing special happens, but if the Orouboros[2] serpent bits its own tail something very special must be expected.

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

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

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

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

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

[2]This symbol appears usually among the Gnostics and is depicted as a dragon, snake or serpent biting its own tail. In the broadest sense, it is symbolic of time and the continuity of life. The Orouboros biting its own tail is symbolic of self-fecundation, or the "primitive" idea of a self-sufficient Nature - a Nature, that is continually returning, within a cyclic pattern, to its own beginning.

The top view of the digital circuits domain is almost completely characterized by the previous features. Almost all of them are not technology dependent. In the following, the physical embodiment of these concepts will be done using CMOS technology. The main assumptions grounding this approach may change in time, but now they are enough robust and are simply stated as follows: *computation is an effective formally defined process, specified using finite descriptions, i.e., the length of the description is not related with the dimension of the processed data, with the amount of time and of physical resources involved.*

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

2.5 Problems

Autonomous circuits

Problem 2.1 *Prove the reciprocal of Theorem 1.1.*

Problem 2.2 *Let be the circuit from Problem 1.25. Use the Verilog simulator to prove its autonomous behavior. After a starting sequence applied on its inputs, keep a constant set of values on the input and see if the output is evolving.*

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

Problem 2.3 *Design a circuit which after the reset generates in each clock cycle the next Fibbonaci number starting from zero, until the biggest Fibbonaci number smaller than* 2^{32}*. When the biggest number is generated the machine will start in the next clock cycle from the beginning with 0. It is supposed the biggest Fibbonaci number smaller than* 2^{32} *in unknown at the design time.*

Problem 2.4 *To the previously designed machine add a new feature: an additional output generating the index of the current Fibbonaci number.*

2.6 Projects

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

Project 2.1

Chapter 3

GATES:
Zero order, no-loop digital systems

In this chapter we will forget for the moment about loops. Composition is the only mechanism involved in building a combinational digital system. No-loop circuits generate the class of history free digital systems whose outputs depend only by the current input variables, and are reassigned "continuously" at each change of inputs. Anytime the output results as a specific "combination" of inputs. No autonomy in combinational circuits, whose outputs obey "not to say a word" to inputs.

The combinational functions with n 1-bit inputs and m 1-bit outputs are called Boolean function and they have the following form:

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

For $n = 1$ only the NOT function is meaningful in the set of the 4 one-input Boolean functions. For $n = 2$ from the set of 16 different functions only few functions are currently used: AND, OR, XOR, NAND, NOR, NXOR. Starting with $n = 3$ the functions are defined only by composing 2-input functions. (For a short refresh see Appendix *Boolean functions*.)

Composing small gates results big systems. The growing process was governed in the last 40 years by Moore's Law[1]. For a few more decades maybe the same growing law will act. But, starting from millions of gates per chip, it is very important what kind of circuits grow exponentially!

Composing gates results two kinds of big circuits. Some of them are structured following some *repetitive patterns*, thus providing simple circuits. Others grow *patternless*, providing complex circuits.

3.1 Simple, Recursive Defined Circuits

The first circuits used by designers were small **and** simple. When they were grew a little they were called big **or** complex. But, now when they are huge we must talk, more carefully, about *big sized simple circuits* **or** about *big sized complex circuits*. In this section we will talk about simple circuits which can be actualized at any size, i.e., their definitions don't depend by the number, n, of their inputs.

In the class of n-inputs circuits there are 2^{2^n} distinct circuits. From this tremendous huge number of logical function we use currently an insignificant small number of simple functions. What is strange is that these functions are sufficient for almost all the problem which we are confronted (or we are limited to be confronted).

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

One fact is clear: we can not design very big complex circuits because we can not specify them. The complexity must get away in another place (we will see that this place is the world of symbols). If we need big circuit they must remain simple.

In this section we deal with simple, if needed big, circuits and in the next with the complex circuits, but only with ones having small size.

From the class of the simple circuits we will present only some very usual such as *decoders*, *demultiplexors*, *multiplexors*, *adders* and *arithmetic-logic units*. There are many other interesting and useful functions. Many of them are proposed as problems at the end of this chapter.

3.1.1 Shifters

One of the simplest arithmetic circuit is a circuit able to multiply or to divided with a number equal with a power of 2. The circuit is called also **shifter** because these operations do not change the relations between the bits of the number, they change only the position of the bits. The bits are *shifted* a number of positions to the left, for multiplication, or to the right for division.

The circuit used for implementing a shifter for n-bit numbers with $m-1$ positions is an m-input multiplexor having the selected inputs defined on n bits.

In the previous subsection were defined multiplexors having 1-bit selected inputs. How can be expanded the number of bits of the selected inputs? An elementary multiplexor for p-bit words, pEMUX, is made using p EMUXs connected in *parallel*. If the two words to be multiplexed are $a_{p-1},\dots a_0$ and $b_{p-1},\dots b_0$, then each EMUX is used to multiplex a pair of bits (a_i,b_i). The one-bit selection signal is shared by the p EMUXs. nEMUX is a *parallel extension* of EMUX.

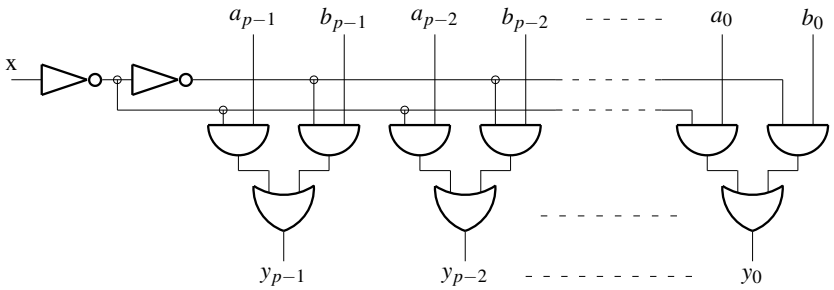


Figure 3.1: **The structure of pEMUX.** Because the selection bit is the same for all EMUXs one EDCD is shared by all of them.

Using pEMUXs an pMUX_n can be designed using the same recursive procedure as for designing MUX_n starting from EMUXs.

An 2^n-1 positions **left shifter** for p-bit numbers, p$LSHIFT_n$, is designed connecting the selected inputs of an pEMUX, $i_0,\dots i_{m-1}$ where $m=2^n$, to the number to be shifted $N=\{a_{n-1},\ a_{n-2},\ \dots\ a_0\}$ ($a_i \in \{0.1\}$ for $i=0,1,\dots(m-1)$) in 2^n-1 ways, according to the following rule:

$$i_j = \{\{a_{n-j-1},\ a_{n-j-2},\ \dots\ a_0\},\ \{j\{0\}\}\}$$

for: $j=0,1,\dots(m-1)$.

Example 3.1 *For $4LSHIFT_2$ an $4MUX_2$ is used. The binary code specifying the shift dimension is* `shiftDim[1:0]`*, the number to be shifted is* `in[3:0]`*, and the ways the selected inputs,* `in0, in1, in2, in3`*, are connected to the number to be shifted are:*

```
in0 = {in[3], in[2], in[1], in[0]}
in1 = {in[2], in[1], in[0], 0    }
in2 = {in[1], in[0], 0    , 0    }
in3 = {in[0], 0    , 0    , 0    }
```

Figure 3.2 represents the circuit.

The Verilog description of the shifter is done by instantiating a 4-way 4-bit multiplexor with its inputs connected according to the previously described rule.

```
module leftShifter(output [3:0] out     ,
                   input  [3:0] in      ,
                   input  [1:0] shiftDim);
   mux4_4 shiftMux(.out(out            ),
                   .in0(in             ),
                   .in1({in[2:0], 1'b0}),
                   .in2({in[1:0], 2'b0}),
                   .in3({in[0]  , 3'b0}),
                   .sel(shiftDim       ));
endmodule
```

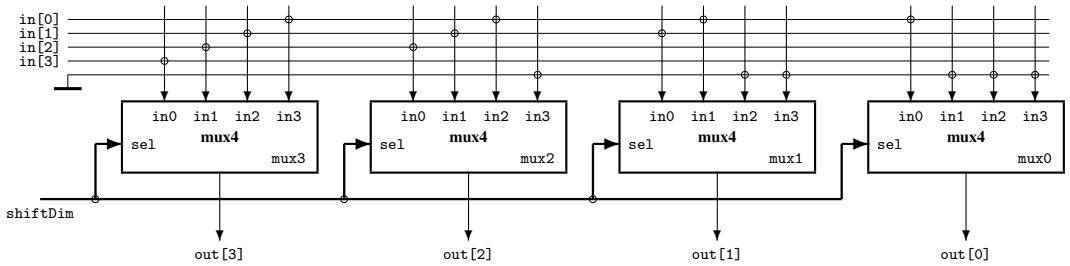


Figure 3.2: **The structure of 4$LSHIFT_2$, a maximum 3-position, 4-bit number left shifter.**

The multiplexor used in the previous module is built using 3 instantiations of an elementary 4-bit multiplexors. Results the two level tree of elementary multiplexors interconnected as the following Verilog code describes.

```
module mux4_4(output [3:0] out,
              input  [3:0] in0, in1, in2, in3,
              input  [1:0] sel); // 4-way 4-bit multiplexor (4MUX_2)
   wire[3:0]       out1, out0;   // internal connections between the two levels
   mux2_4  mux(out, out0, out1, sel[1]), // output multiplexor
           mux1(out1, in2, in3, sel[0]), // input multiplexor for in3 and in2
           mux0(out0, in0, in1, sel[0]); // input multiplexor for in1 and in0
endmodule
```

Any n-bit elementary multiplexer is described by the following parameterized module:

```
module mux2_4 #(parameter n = 4)(output [n-1:0] out,
                                 input  [n-1:0] in0, in1,
                                 input          sel); // 2-way 4-bit mux: 4EMUX
   assign  out = sel ? in1 : in0; // if (sel) then in1, else in0
endmodule
```

◇

The same idea helps us to design a special kind of shifter – **barrel shifter** – which performs a **rotate** operation described as follows: if the input number is $N = \{a_{n-1},\ a_{n-2},\ \ldots\ a_0\}$ ($a_i \in \{0.1\}$ for $i = 0, 1, \ldots (m-1)$), then rotating it with i positions will provide:

$$i_i = \{a_{n-i-1},\ a_{n-i-2},\ \ldots\ a_0,\ a_{n-1}, a_{n-2}, \ldots a_{n-i}\}$$

for: $i = 0, 1, \ldots (m-1)$. This *first solution* for the rotate circuit is very similar with the shift circuit. The only difference is: all the inputs of the multiplexor are connected to an input value. No 0s on any inputs of the multiplexor.

A *second solution* uses only elementary multiplexors. A version for 8-bit numbers is presented in the following Verilog code.

```
module leftRotate(output [7:0] out      ,
                  input  [7:0] in       ,
                  input  [2:0] rotateDim);
   wire [7:0] out0, out1;
   mux2_8
    level0(.out(out0), .in0(in  ), .in1({in[6:0], in[7]}  ),     .sel(rotateDim[0])),
    level1(.out(out1), .in0(out0), .in1({out0[5:0], out0[7:6]}), .sel(rotateDim[1])),
    level2(.out(out ), .in0(out1), .in1({out1[3:0], out1[7:4]}), .sel(rotateDim[2]));
endmodule

module mux2_8(output [7:0] out,
              input  [7:0] in0, in1,
              input        sel);
   assign  out = sel ? in1 : in0;
endmodule
```

While the *first solution* uses for n bit numbers n $MUX_{log_2\,(rotateDim)}$, the *second solution* uses $log_2\,(rotateDim)$ $nEMUX$s. Results:

$$S_{firstSolutionOfLeftRotate} = (n \times (rotateDim - 1)) \times S_{EMUX}$$

$$S_{secondSolutionOfLeftRotate} = (n \times log_2\,(rotateDim)) \times S_{EMUX}$$

3.1.2 Priority encoder

An encoder is a circuit which connected to the outputs of a decoder provides the value applied on the input of the decoder. As we know only one output of a decoder is active at a time. Therefore, the encoder compute the index of the activated output. But, a real application of an encoder is to encode binary configurations provided by any kind of circuits. In this case, more than one input can be active and the encoder must have a well defined behavior. One of this behavior is to encode the most significant bit and to ignore the rest of bits. For this reason the encoder is a *priority encoder*.

The n-bit input, enabled priority encoder circuit, $PE(n)$, receives $x_{n-1}, x_{n-2}, \ldots x_0$ and, if the enable input is activated, $en = 1$, it generates the number $Y = y_{m-1}, y_{m-2}, \ldots y_0$, with $n = 2^m$, where Y is the biggest index associated with $x_i = 1$ if any, else *zero* output is activated. (For example: **if** $en = 1$, for $n = 8$, and $x_7, x_6, \ldots x_0 = 00110001$, **then** $y_2, y_1, y_0 = 101$ and $zero = 0$) The following Verilog code describe the behavior of $PE(n)$.

```
module priority_encoder #(parameter m = 3)(input       [(1'b1<<m)-1:0] in      ,
                                           input                       enable  ,
                                           output  reg [m-1:0]         out     ,
                                           output  reg                 zero    );
   integer i;
   always @(*) if (enable) begin   out = 0;
                                   for(i=(1'b1 << m)-1; i>=0; i=i-1)
                                       if ((out == 0) && in[i]) out = i;
                                   if (in == 0)    zero = 1;
                                       else        zero = 0;
                           end
               else        begin   out = 0;
                                   zero = 1;
                           end
endmodule
```

For testing the previous description the following test module is used:

```
module test_priority_encoder #(parameter   m = 3);
   reg     [(1'b1<<m)-1:0]     in      ;
   reg                         enable  ;
   wire    [m-1:0]             out     ;
   wire                        zero    ;
   initial     begin       enable = 0;
                           in = 8'b11111111;
                       #1  enable = 1;
```

```
                    #1  in = 8'b00000001;
                    #1  in = 8'b0000001x;
                    #1  in = 8'b000001xx;
                    #1  in = 8'b00001xxx;
                    #1  in = 8'b0001xxxx;
                    #1  in = 8'b001xxxxx;
                    #1  in = 8'b01xxxxxx;
                    #1  in = 8'b1xxxxxxx;
                    #1  in = 8'b110;
                    #1  $stop;
            end
  priority_encoder  dut(in      ,
                        enable  ,
                        out     ,
                        zero    );
  initial $monitor  ($time, "enable=%b in=%b out=%b zero=%b",
                            enable, in, out, zero);
endmodule
```

Running the previous code the simulation provides the following result:

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

It is obvious that this circuit computes the integer part of the base 2 logarithm. The output *zero* is used to notify that the input value is unappropriate for computing the logarithm, and "prevent" us from takeing into account the output value.

3.1.3 Prefix computation network

There is a class of circuits, called *prefix computation networks*, $PCN_{func}(n)$, defined for n inputs and having the characteristic function $func$. If $func$ is expressed using the operation $\circ$, then the function of $PCN_{\circ}(n)$ is performed by a circuit having the inputs $x_0, \ldots x_{n-1}$ and the outputs $y_0, \ldots y_{n-1}$ related as follows:

$$y_0 = x_0$$

$$y_1 = x_0 \circ x_1$$

$$y_2 = x_0 \circ x_1 \circ x_2$$

$$\ldots$$

$$y_{n-1} = x_0 \circ x_1 \circ \ldots \circ x_{n-1}$$

where the operation "$\circ$" is an associative and commutative operation. For example, $\circ$ can be the arithmetic operation *add*, or the logic operation *AND*. In the first case x_i is an m-bit binary number, and in the second case it is a 1-bit Boolean variable.

Example 3.2 *If $\circ$ is the Boolean function AND, then $PCN_{AND}(n)$ is described by the following behavioral description:*

```
module and_prefixes #(parameter n = 64)(input      [0:n-1] in ,
                                        output reg [0:n-1] out);
   integer k;
   always @(in) begin out[0] = in[0];
                      for (k=1; k<n; k=k+1) out[k] = in[k] & out[k-1];
                end
endmodule
```

$\diamond$

There are many solutions for implementing $PCN_{AND}(n)$. If we use AND gates with up to n inputs, then there is a **first** direct solution for PCN_{AND} starting from the defining equations (it consists in one 2-input gate, plus one 3-input gate, ... plus one (n-1)-input gate). A very large high-speed circuit is obtained. Indeed, this direct solution offers a circuit with the size $S(n) \in O(n^2)$ and the depth $D(n) \in O(1)$. We are very happy about the speed (depth), but the price paid for this is too high: the squared size. In the same time our design experience tells us that this speed is not useful in current applications because of the time correlations with other subsystems. (There is also a discussion about gate having n inputs. These kind of gates are not realistic.)

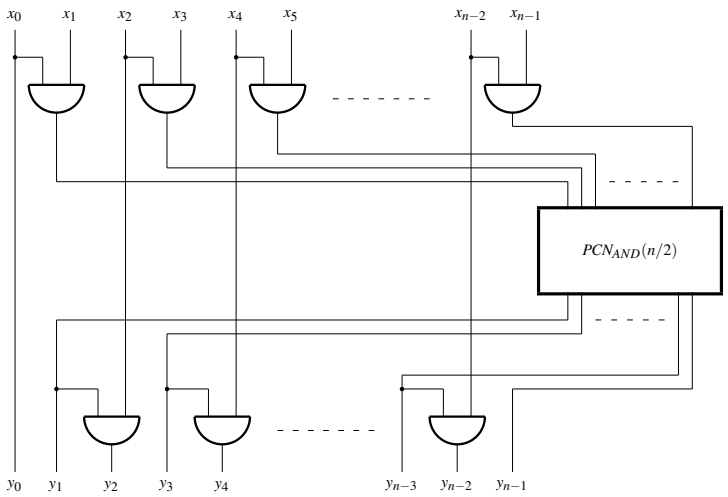


Figure 3.3: **The internal structure of $PCN_{AND}(n)$.** It is recursively defined: if $PCN_{AND}(n/2)$ is a prefix computation network, then the entire structure is PCN_n.

A **second** solution offers a very good size but a too slow circuit. If we use only 2-input AND gates, then the definition becomes:

$$y_0 = x_0$$

$$y_1 = y_0 \,\&\, x_1$$

$$y_2 = y_1 \,\&\, x_2$$

$$\ldots$$

$$y_{n-1} = y_{n-2} \,\&\, x_{n-1}$$

A direct solution starting from this new form of the equations (as a degenerated binary tree of ANDs) has $S(n) \in O(n)$ and $D(n) \in O(n)$. This second solution is also very inefficient, now because of the speed which is too low.

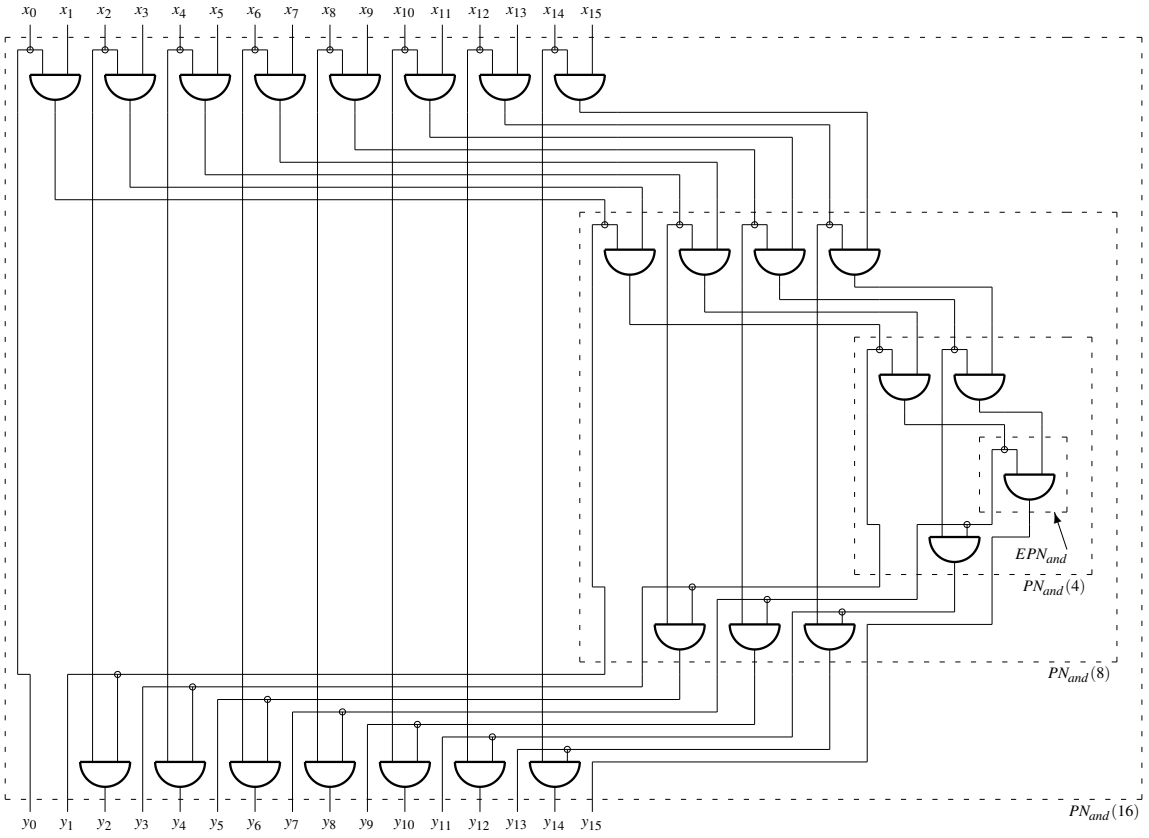


Figure 3.4: $PCN_{AND}(16)$

The **third** implementation is a optimal one. For $PCN_{AND}(n)$ is used the recursive defined network represented in Figure 3.3 [Ladner '80], where in each node, for our application, there is a 2-inputs *AND* gate. If $PCN_{AND}(n/2)$ is a well-functioning prefix network, then all the structure works as a prefix network. Indeed, $PCN_{AND}(n/2)$ computes all even outputs because of the $n/2$ input circuits that perform the y_2 function between successive pairs of inputs. On the output level the odd outputs are computed using even outputs and odd inputs. The $PCN_{AND}(n/2)$ structure is built upon the same rule and so on until $PCN_{AND}(1)$ that is a system without any circuit. The previous recursive definition is applied for $n = 16$ in Figure 3.4.

The size of $PCN_{AND}(n)$, $S(n)$ (with n a power of 2), is evaluated starting from: $S(1) = 0$, $S(n) = S(n/2) + (n-1)S_{y_1}$ where S_{y_1} is the size of the elementary circuit that defines the network (in our case is a 2-inputs AND gate). The next steps leads to:

$$S(n) = S(n/2^i) + (n/2^{i-1} + \ldots + n/2^0 - i)S_{y_1}$$

and ends, for $i = log_2 n$ (the rule is recursively applied $log\ n$ times), with:

$$S(n) = (2n - 2 - log_2 n)S_{y_1} \in O(n).$$

(The network consists in two binary trees of elementary circuits. The first with the bottom root having $n/2$ leaves on the first level. The second with the top root having $n/2 - 1$ leaves on the first level. Therefore, the first tree has $n-1$ elementary circuits and the second tree has $n - 1 - log\ n$ elementary circuits.) The depth is $D(n) = 2D_{y_2} log_2\ n \in O(log\ n)$ because $D(n) = D(n/2) + 2$ (at each step two levels are added to the system).

3.1.4 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 3.5). The half adder has two roles in the structure:

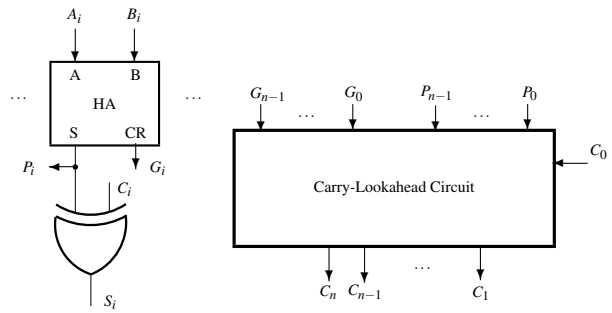


Figure 3.5: **The fast n-bit adder.** The n-bit Carry-Lookahead Adder (CLA_n) consists in n HAs, n 2-input XORs and the Carry-Lookahead Circuit used to compute faster the n C_i, for $i = 1, 2, \ldots n$.

- 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.

3.1.5 Prefix-Based Carry-Look-Ahead Adder

Let us consider the expressions for C_1, $C_2, \ldots C_i, \ldots$. It looks like each product from C_{i-1} is a prefix of a product from C_i. This suggests a way to reduce the too big size of the carry-look-ahead circuit from the previous paragraph. Following [Cormen '90] (see section 29.2.2), an optimal carry-look-ahead circuit (with asymptotically linear size and *log* depth) is described in this paragraph. It is known as Brent-Kung adder.

Let be x_i, the *carry state*, the information used in the stage i to determine the value of the carry. The carry state takes three values, according to the table represented in Figure 3.6, where A_i and B_i are the i-th bits of the numbers to be added. If $A_i = B_i = 0$, then the carry bit is 0 (it is *killed*). If $A_i = B_i = 1$, then the carry bit is 1 (it is *generated*). If $A_i = 0$, and $B_i = 1$ or $A_i = 1$ and $B_i = 0$, then the carry bit is equal with the carry bit generated in the previous stage, C_{i-1} (the carry from the previous range *propagates*). Therefore, in each binary stage the carry state has three values: *k, p, g*.

A_{i-1}	B_{i-1}	C_i	x_i
0	0	0	**k***ill*
0	1	C_{i-1}	**p***ropagate*
1	0	C_{i-1}	**p***ropagate*
1	1	1	**g***enerate*

Figure 3.6: **Kill-Propagate-Generate table.**

We define the function $\otimes$ which composes the carry states of the two binary ranges. In Figure 3.7 the states x_i and x_{i-1} are composed generating the carry state of two successive bits, $i-1$ an i. If $x_i = k$, then

the resulting composed state is independent by x_{i-1} and it takes the value k. If $x_i = g$, then the resulting composed state is independent by x_{i-1} and it takes the value g. If $x_i = p$, then the resulting composed state *propagates* and it is x_{i-1}.

The carry state for the binary range i is determined using the expression:

$$y_i = y_{i-1} \otimes x_i = x_0 \otimes x_1 \otimes \ldots x_i$$

starting with:

$$y_0 = x_0 = \{k, g\}$$

which is associated with the carry state used to specify the carry input C_0 (for k is no carry, for g is carry, while p has no meaning for the input carry state).

		x_i		
	$\otimes$	k	p	g
	k	k	k	g
x_{i-1}	p	k	p	g
	g	k	g	g

Figure 3.7: **The elementary Carry-Look-Ahead (eCLA) function [Cormen '90].**

Thus, carry states are computed as follows:

$y_0 = x_0$
$y_1 = x_0 \otimes x_1$
$y_2 = x_0 \otimes x_1 \otimes x_2$
$\ldots$
$y_i = x_0 \otimes x_1 \otimes x_2 \ldots x_i$

It is obvious that we have a prefix computation. Let us call the function $\otimes$ eCLA (elementary Carry-Look-Ahead). Then, the circuit used to compute the carry states $y_0, y_1, y_2, \ldots, y_i$ is ***prefixCLA***.

Theorem 3.1 *If $x_0 = \{k, g\}$, then $y_1 = \{k, g\}$ for $i = 1, 2, \ldots i$ and the value of C_i is set to 0 by k and to 1 by g.*

Proof*: in the table from Figure 3.7 if the line p is not selected, then the value p does not occur in any carry state. Because, $x_0 = \{k, g\}$ the previous condition is fulfilled.*

$\diamond$

An appropriate codding of the three values of x will provide a simple implementation. We propose the codding represented in the table from Figure 3.8. The two bits used to code the state x are P and G (with the meaning used in the previous paragraph). Then, $y_i[0] = C_i$.

The first, direct form of the adder circuit, for $n = 7$, is represented in Figure 3.9, where there are three levels:

- the input level of half-adders which compute $x_i = \{P_i, G_i\}$ for $1 = 1, 2, \ldots 6$, where:

$$P_i = A_{i-1} \oplus B_{i-1}$$

$$G_i = A_{i-1} \cdot B_{i-1}$$

x_i	P	G
k	0	0
p	1	0
g	0	1

Figure 3.8: **Coding the carry state.**

- the intermediate level of ***prefixCLA*** which computes the carry states $y_1, y_2, \ldots y_7$ with a chain of *eCLA* circuits
- the output level of XORs used to compute the sum S_i starting from P_{i+1} and C_i.

The size of the resulting circuit is in $O(n)$, while the depth is in the same order of magnitude. The next step is to reduce the depth of the circuit to $O(log\,n)$.

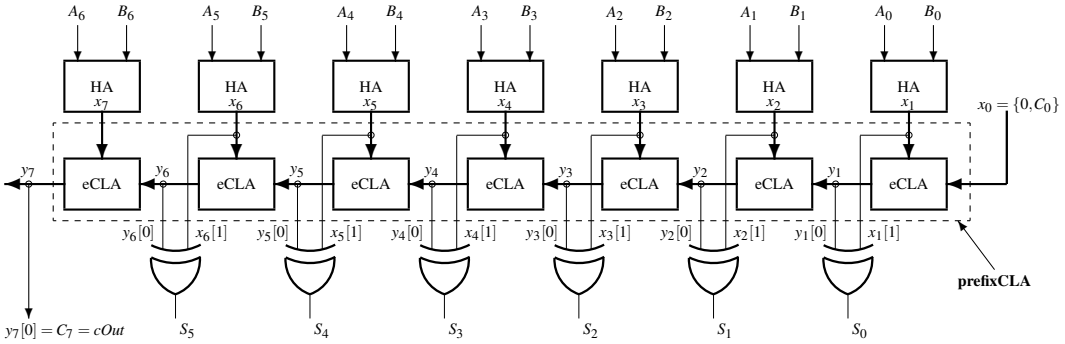


Figure 3.9: **The 7-bit adder with ripple eCLA.**

The eCLA circuit is designed using the eCLA function defined inFigure 3.7 and the codding of the carry state presented in Figure 3.8. The resulting logic table is represented in Figure 3.10. It is build takeing into consideration that the code 11 is not used to define the carry state x. Thus in the lines containing $x_i = \{P_i, G_i\} = 11$ and $x_{i-1} = \{P_{i-1}, G_{i-1}\} = 11$ the function is not defined (instead of 0 or 1 the value of the function is "don't care"). The expressions for the two outputs of the eCLA circuit are:

$$P_{out} = P_i \cdot P_{i-1}$$

$$G_{out} = G - i + P - i \cdot G_{i-1}$$

For the pure serial implementation of the ***prefixCLA*** circuit from Figure 3.9, because $x_0 = \{0, C_0\}$, in each eCLA circuit $P_{out} = 0$. The full circuit will be used in the log-depth implementation of the ***prefixCLA*** circuit. Following the principle exposed in the paragraph *Prefix Computation network* the ***prefixCLA*** circuit is redesigned optimally for the adder represented in Figure 3.11. If we take from Figure 3.4 the frame labeled $PN_{and}(8)$ and the 2-input AND circuits are substituted with eCLA circuits, then results the ***prefixCLA*** circuit from Figure 3.11.

Important notice: the eCLA circuit is not a symmetric one. The most significant inputs, P_i and G_i correspond to the i-th binary range, while the other two correspond to the previous binary range. This

P_i	G_i	P_{i-1}	G_{i-1}	P_{out}	G_{out}
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	-	-
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	-	-
1	0	0	0	0	0
1	0	0	1	0	1
1	0	1	0	1	0
1	0	1	1	-	-
1	1	0	0	-	-
1	1	0	1	-	-
1	1	1	0	-	-
1	1	1	1	-	-

Figure 3.10: **The logic table for eCLA.**

order should be taken into consideration when the ***log*-depth prefixCLA** (see Figure 3.11) circuit is organized.

While in the ***prefixCLA*** circuit from Figure 3.9 all eCLA circuits have the output $P_{out} = 0$, in the ***log*-depth prefixCLA** from Figure 3.11 some of them are fully implemented. Because $x_0 = \{0, C_0\}$, eCLA indexed with 0 and all the eCLAs enchained after it, indexed with 4,6, 7, 8, 9, and 10, are of the simple form with $P_{out} = 0$. Only the eCLA indexed with 1, 2, 3, and 5 are fully implemented, because their inputs are not restricted.

This version of n-bit adder is asymptotically optimal, because the size of the circuit is in $O(n)$ and the depth is in $O(log\, n)$. Various versions of this circuit are presented in [webRef_3].

3.1.6 Carry-Save Adder

For adding m n-bit numbers there is a faster solution than the one which supposes to use the direct circuit build as a tree of $m-1$ 2-number adders. The depth and the size of the circuit is reduced using a **carry-save adder** circuit.

The carry save adder receives three n-bit numbers:

$$x = \{x_{n-1}, x_{n-2}, \ldots x_1, x_0\}$$

$$y = \{y_{n-1}, y_{n-2}, \ldots y_1, y_0\}$$

$$z = \{z_{n-1}, z_{n-2}, \ldots z_1, z_0\}$$

and generate two $(n+1)$-bit numbers:

$$c = \{c_{n-1}, c_{n-2}, \ldots c_1, c_0, 0\}$$

$$s = \{0, s_{n-1}, s_{n-2}, \ldots s_1, s_0\}$$

where:

$$c + s = x + y + z.$$

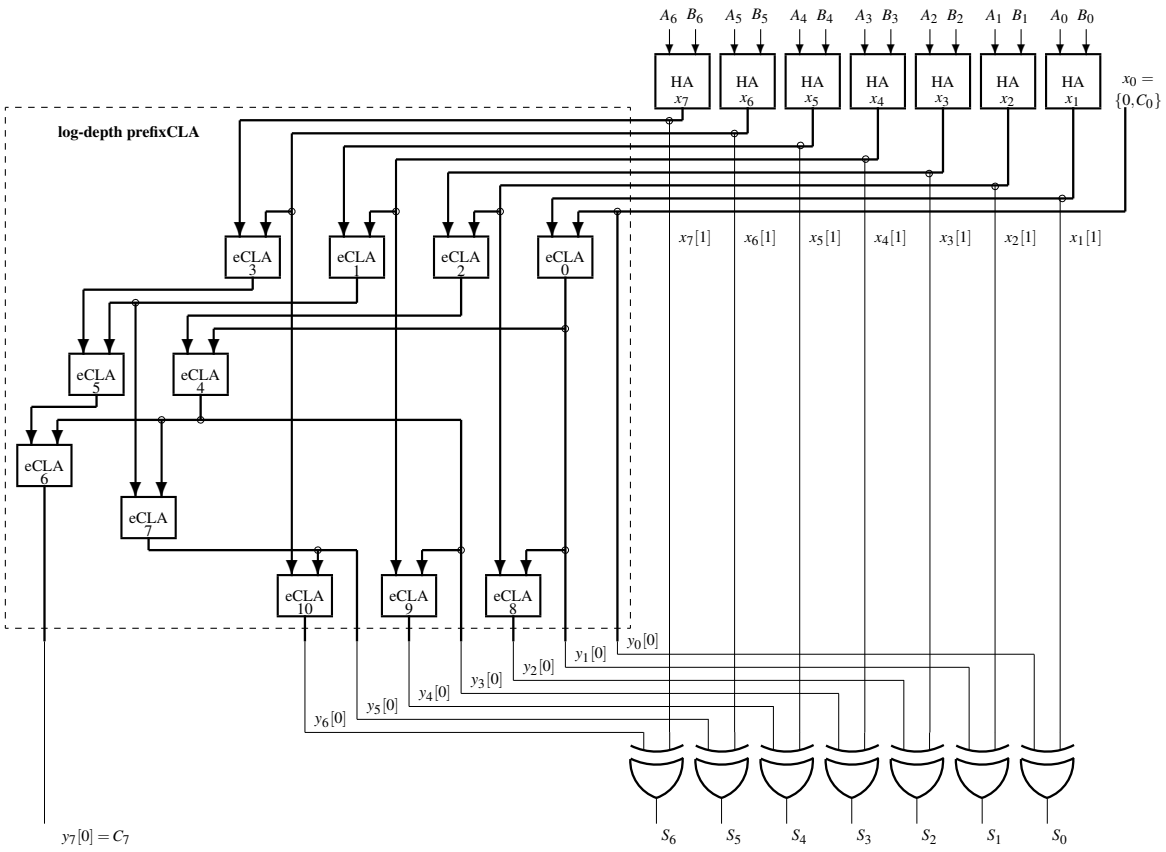


Figure 3.11: **The adder with the prefix based carry-look-ahead circuit.**

The function of the circuit is described by the following transfer function applied for $i = 0, \ldots n-1$:

$$x_i + y_i + z_i = \{c_i, s_i\}.$$

The internal structure of the carry-save adders contains n circuits performing the previous function which is the function of a full adder. Indeed, the binary variables x_i and y_i are applied on the two inputs of the FA, z_i is applied on the carry input of the same FA, and the two inputs c_i, s_i are the *carry-out* and the *sum* outputs. Therefore, an *elementary carry-save adder*, ECSA, has the structure of a FA (see Figure 3.12a).

To prove for the functionality of CSA we write for each ECSA:

$$\{c_{n-1}, s_{n-1}\} = x_{n-1} + y_{n-1} + z_{n-1}$$

$$\{c_{n-2}, s_{n-2}\} = x_{n-2} + y_{n-2} + z_{n-2}$$

$$\ldots$$

$$\{c_1, s_1\} = x_1 + y_1 + z_1$$

$$\{c_0, s_0\} = x_0 + y_0 + z_0$$

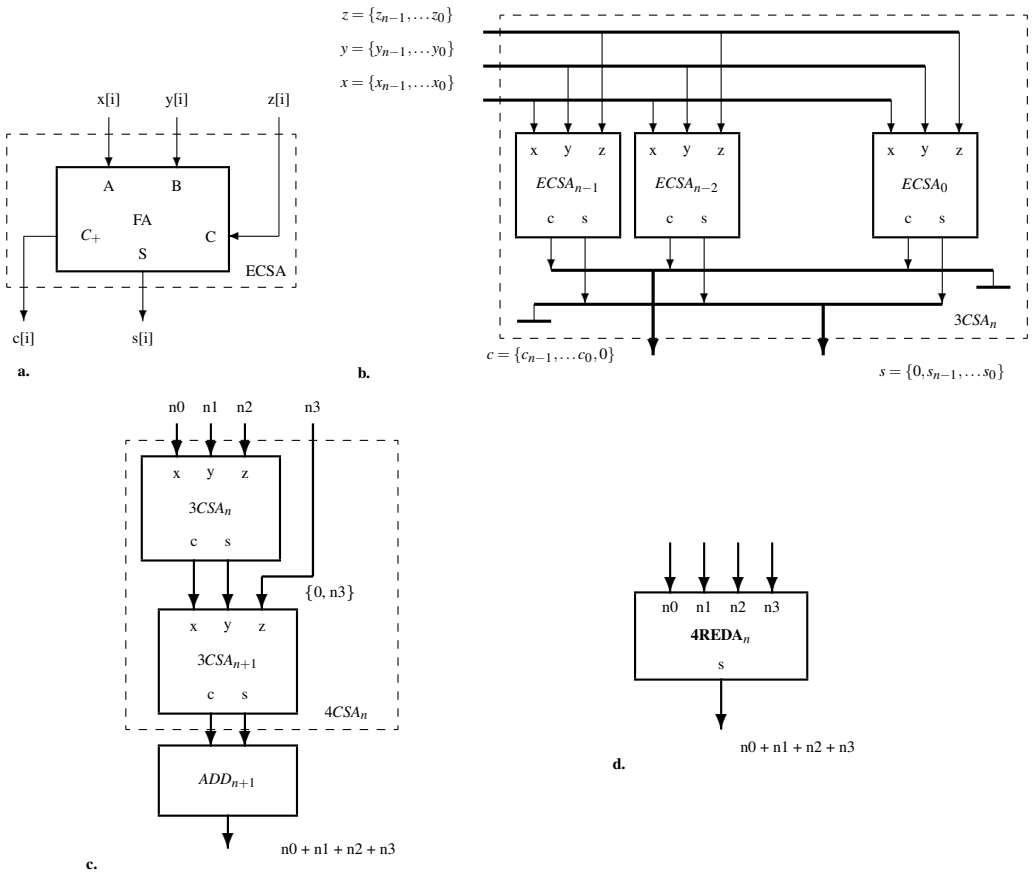


Figure 3.12: **Carry-Save Adder. a.** The elementary carry-save adder. **b.** CSA for 3 n-bit numbers: $3CSA_n$. **c.** How to use carry-save adders to add 4 n-bit numbers. **d.** The logic symbol for a 4 n-bit inputs reduction adder (**4REDA**$_n$) implemented with two $3CSA_n$ and a $(n+1)$-bit adder.

from which results that:

$$x+y+z = \{c_{n-1}, s_{n-1}\} \times 2^{n-1} + \{c_{n-2}, s_{n-2}\} \times 2^{n-2} + \ldots \{c_1, s_1\} \times 2^1 + \{c_0, s_0\} \times 2^0 =$$

$$\{c_{n-1}, c_{n-2}, \ldots c_1, c_0\} \times 2 + \{s_{n-1}, s_{n-2}, \ldots s_1, s_0\} = \{c_{n-1}, c_{n-2}, \ldots c_1, c_0, 0\} + \{0, s_{n-1}, s_{n-2}, \ldots s_1, s_0\}.$$

Figure 3.12c shows a 4-number adder for n-bit numbers. Instead of 3 adders, 2 $3CSA_n$ circuits and a $(n+1)$-bit adder are used. The logic symbol for the resulting 4-input reduction adder – $4REDA_n$ – is represented in Figure 3.12d. The depth of the circuit is of 2 FAs and an $(n+1)$-bit adder, instead of the depth associated with one n-bit adder and an $(n+1)$-bit adder. The size is also minimized if in the standard solution carry-look-ahead adders are used.

Example 3.3 *The module $3CSA_m$ is generated using the following template:*

```
module csa3_m #(parameter n=m)(input   [n-1:0] in0, in1, in2   ,
                               output  [n:0]   sOut, cOut      );
```

```
    wire    [n-1:0] out ;
    wire    [n-1:0] cr  ;

    genvar  i   ;
    generate for (i=0; i<n; i=i+1) begin: S
        fa adder(in0[i], in1[i], out[i], cr[i]);
        end
    endgenerate

    assign sOut = {1'b0, out}   ;
    assign cOut = {cr, 1'b0}    ;
endmodule
```

where: (1) an actual value for m must be provided and (2) the module `fa` *is defined in the previous example.* ◇

Example 3.4 *For the design of a 8 1-bit input adder (*$8REDA_1$*) the following modules are used:*

```
module reda8_1(    output  [3:0]   out ,
                   input   [7:0]   in  );
   wire    [2:0]   sout, cout  ;
   csa8_1 csa( in[0],in[1],in[2],in[3],in[4],in[5],in[6],in[7],cout,sout);
   assign out = cout + sout    ;
endmodule

module csa8_1( input           in0, in1, in2, in3, in4, in5, in6, in7,
               output  [2:0]   cout, sout                              );
   wire    [1:0]   sout0, cout0, sout1, cout1;
   csa4_1  csa0(in0, in1, in2, in3, sout0, cout0),
           csa1(in4, in5, in6, in7, sout1, cout1);
   csa4_2  csa2(sout0, cout0, sout1, cout1, cout, sout);
endmodule
```

where `csa4_1` *and* `csa4_2` *are instances of the following generic module:*

```
module csa4_p( input   [1:0]   in0, in1, in2, in3  ,
               output  [2:0]   sout, cout          );
   wire    [2:0]   s1, c1  ;
   wire    [3:0]   s2, c2  ;
   csa3_p csa0(in0, in1, in2, s1, c1);
   csa3_q csa1(s1, c1, {1'b0, in3}, s2, c2);
   assign sout = s2[2:0], cout = c2[2:0];
endmodule
```

for $p = 1$ *and* $p = 2$*, while* $q = p + 1$*. The module* `csa3_m` *is defined in the previous example.* ◇

The efficiency of this method to add many numbers increases, compared to the standard solution, with the number of operands.

3.1.7 Combinational Multiplier

One of the most important application of the CSAs circuits is the efficient implementation of a combinational multiplier. Because multiplying n-bit numbers means to add n $2n-1$-bit partial products, a $nCSA_{2n-1}$ circuit provides the best solution. Adding n n-bit numbers using standard carry-adders is done, in the best case, in time belonging to $O(n \times log\, n)$ on a huge area we can not afford. The proposed solution provides a linear time.

In Figure 3.13 is an example for how for the binary multiplication for $n = 4$ works. Thus, the combinational multiplication is done in the following three stages:

			a_3	a_2	a_1	a_0	$\times$	multiplicand: a				1	0	1	1 $\times$
			b_3	b_2	b_1	b_0		multiplier: b				1	1	0	1
			p_{03}	p_{02}	p_{01}	p_{00}		partial product: pp_0				1	0	1	1
		p_{13}	p_{12}	p_{11}	p_{10}			partial product: pp_1			0	0	0	0	
	p_{23}	p_{22}	p_{21}	p_{20}				partial product: pp_2		1	0	1	1		
p_{33}	p_{32}	p_{31}	p_{30}					partial product: pp_3	1	0	1	1			
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	final product: p	1	0	0	0	1	1	1 1

Figure 3.13: **Multiplying 4-bit numbers.**

1. compute n partial products – $pp_0, \ldots pp_{n-1}$ – using a two-dimension array of $n \times n$ AND_2 circuits (see in Figure 3.14 the "partial products computation" dashed box); for $n = 4$ the following relations describe this stage:

$$pp_0 = \{a_3 \cdot b_0,\ a_2 \cdot b_0,\ a_1 \cdot b_0,\ a_0 \cdot b_0\}$$

$$pp_1 = \{a_3 \cdot b_1,\ a_2 \cdot b_1,\ a_1 \cdot b_1,\ a_0 \cdot b_0\}$$

$$pp_2 = \{a_3 \cdot b_2,\ a_2 \cdot b_2,\ a_1 \cdot b_2,\ a_0 \cdot b_0\}$$

$$pp_3 = \{a_3 \cdot b_3,\ a_2 \cdot b_3,\ a_1 \cdot b_3,\ a_0 \cdot b_0\}$$

2. shift each partial product pp_i i binary positions left; results n $(2n-1)$-bit numbers; for $n = 4$:

$$n0 = pp_0 << 0 = \{0,0,0,pp_0\}$$

$$n1 = pp_1 << 1 = \{0,0,pp_1,0\}$$

$$n2 = pp_2 << 2 = \{0,pp_2,0,0\}$$

$$n3 = pp_3 << 3 = \{pp_3,0,0,0\}$$

easy to be done, with no circuits, by an appropriate connection of the AND array outputs to the next circuit level (see in Figure 3.14 the "hardware-less shifter" dashed box)

3. add the resulting n numbers using a $nCSA_{2n-1}$; for $n = 4$:

$$p = \{0,0,0,pp_0\} plus \{0,0,pp_1,0\} plus \{0,pp_2,0,0\} plus \{pp_3,0,0,0\}$$

The combinational multiplier circuit is presented in Figure 3.14 for the small case of $n = 4$. The first stage – partial products computation – generate the partial products pp_i using 2-input ANDs as one bit multipliers. The second stage of the computation request a hardware-less shifter circuit, because the multiplying n-bit numbers with a power of 2 no bigger than $n-1$ is done by an appropriate connection of each pp_i to the $(2n-1)$-bit inputs of the next stage, filling up the unused positions with zeroes. The third stage consists of a reduction carry-save adder – $nREDA_{2n-1}$ – which receives, as $2n-1$-bit numbers, the partial products pp_i, each multiplied with 2^i.

The circuit represented in Figure 3.14 for $n = 4$, has in the general case the size in $O(n^2)$ and the depth in $O(n)$. But, the actual size and depth of the circuit is established by the 0s applied to the input of the $nREDA_{2n-1}$ circuit, because some full-adders are removed from design and others are reduced to half-adders. The actual size of $nREDA_{2n-1}$ results to be very well approximated with the actual size of a $nREDA_n$. The actual depth of the combinational multiplier is well approximated with the depth of a $2.5n$-bit adder.

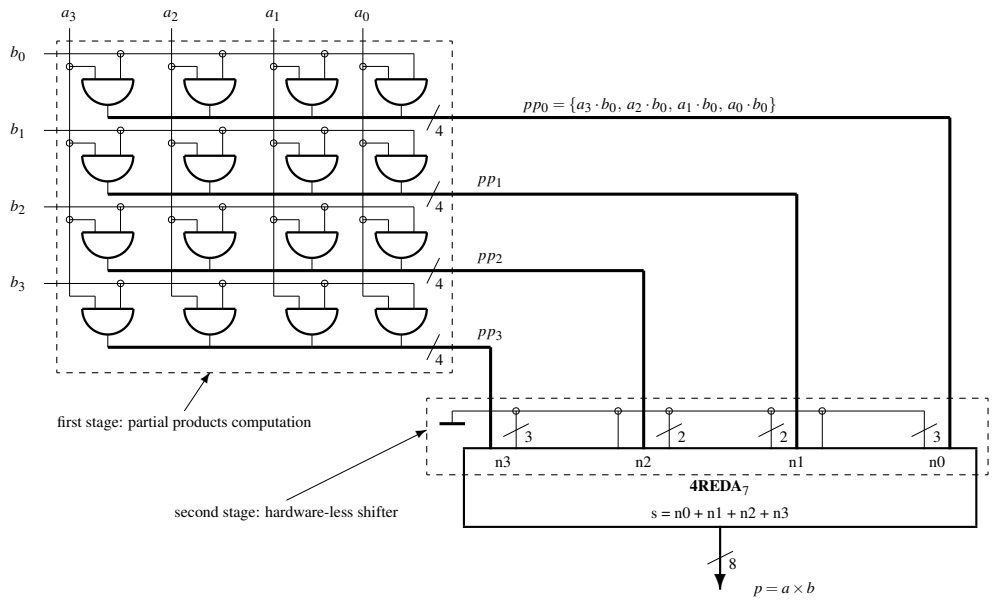


Figure 3.14: **4-bit combinational multiplier.** .

The decision to use a combinational multiplier must be done takeing into account (1) its area, (2) the acceleration provided for the function it performs and (3) the frequency of the multiplication operation in the application.

3.1.8 Comparator

Comparing functions are used in decisions. Numbers are compared to decide if they are equal or to indicate the biggest one. The n-bit comparator, $COMP_n$, is represented in Figure 3.15a. The numbers to be compared are the n-bit positive integers `a` and `b`. Three are the outputs of the circuit: `lt_out`, indicating by 1 that $a < b$, `eq_out`, indicating by 1 that $a = b$, and `gt_out`, indicating by 1 that $a > b$. Three additional inputs are used as expanding connections. On these inputs is provided information

about the comparison done on the higher range, if needed. If no higher ranges of the number under comparison, then these thre inputs must be connected as follows: $lt\_in = 0$, $eq\_in = 1$, $gt\_in = 0$.

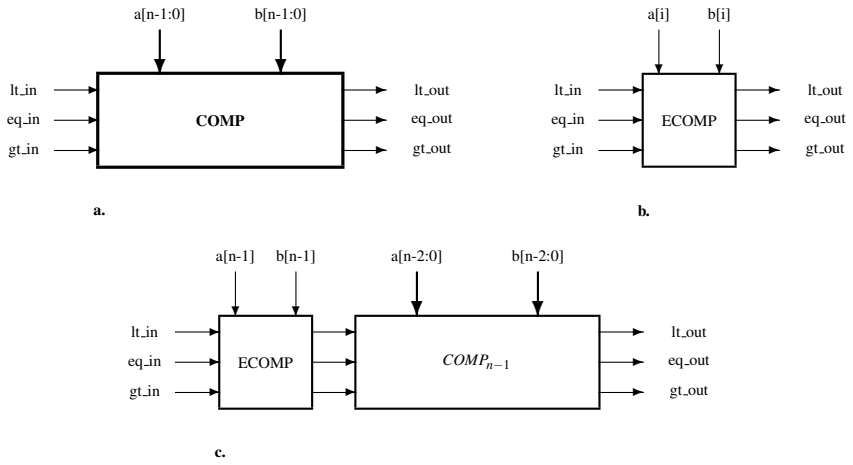


Figure 3.15: **The n-bit comparator, $COMP_n$. a.** The n-bit comparator. **b.** The elementary comparator. **c.** A recursive rule to built an $COMP_n$, serially connecting an $ECOMP$ with a $COMP_{n-1}$

The comparison is a numerical operation which starts inspecting the most significant bits of the numbers to be compared. If $a[n-1] = b[n-2]$, then the result of the comparison is given by comparing $a[n-2:0]$ with $b[n-1:0]$, else, the decision can be done comparing only $a[n-1]$ with $b[n-1]$ (using an elementary comparator, $ECOMP = COMP_1$ (see Figure 3.15b)), ignoring $a[n-2:0]$ and $b[n-2:0]$. Results a recursive definition for the comparator circuit.

Definition 3.1 *An n-bit comparator, $COMP_n$, is obtained serially connecting an $COMP_1$ with a $COMP_{n-1}$. The Verilog code describing $COMP_1$ (ECOMP) follows:*

```
module e_comp( input   a       ,
                       b       ,
                       lt_in   ,  // the previous e_comp decided lt
                       eq_in   ,  // the previous e_comp decided eq
                       gt_in   ,  // the previous e_comp decided gt
               output  lt_out  ,  // a < b
                       eq_out  ,  // a = b
                       gt_out  ); // a > b);
   assign  lt_out = lt_in | eq_in & ~a & b,
           eq_out = eq_in & ~(a ^ b),
           gt_out = gt_in | eq_in & a & ~b;
endmodule
```

◇

The size and the depth of the circuit resulting from the previous definition are in $O(n)$. The size is very good, but the depth is too big for a high speed application.

An optimal comparator is defined using another recursive definition based on the *divide et impera* principle.

Definition 3.2 *An n-bit comparator, $COMP_n$, is obtained using two $COMP_{n/2}$, to compare the higher and the lower half of the numbers (resulting* {`lt_out_high, eq_out_high, gt_out_high`} *and* {`lt_out_low, eq_out_low, gt_out_low`}*), and a $COMP_1$ to compare* `gt_out_low` *with* `lt_out_low` *in the context of* {`lt_out_high, eq_out_high, gt_out_high`}*. The resulting circuit is represented in Figure 3.16.* ◇

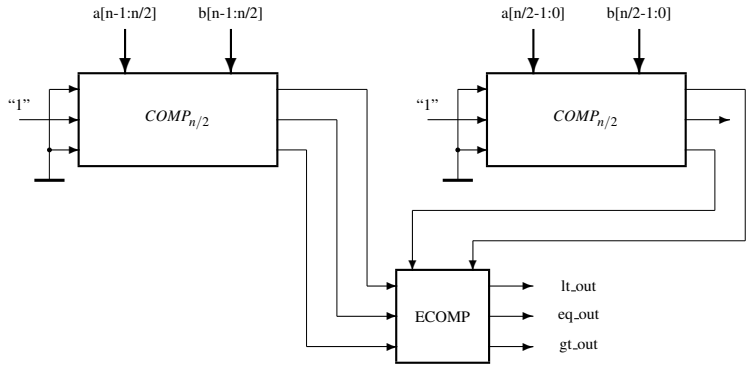


Figure 3.16: **The optimal *n*-bit comparator.** Applying the *divide et impera* principle a $COMP_n$ is built using two $COMP_{n/2}$ and an *ECOMP*. Results a *log*-depth circuit with the size in $O(n)$.

The resulting circuit is a *log*-level binary tree of ECOMPs. The size remains in the same order[2], but now the depth is in $O(log\ n)$.

The bad news is: the HDL languages we have are unable to handle safely recursive definitions. The good news is: the synthesis tools provide good solutions for the comparison functions starting from a very simple behavioral description.

3.1.9 Sorting network

In the most of the cases numbers are compared in order to be sorted. There are a lot of algorithms for sorting numbers. They are currently used to write programs for computers. But, in *G2CE* the sorting function will migrate into circuits, providing specific accelerators for general purpose computing machines.

To solve in circuits the problem of sorting numbers we start again from an elementary module: the *elementary sorter* (ESORT).

Definition 3.3 *An elementary sorter (ESORT) is a combinational circuit which receives two n-bit integers,* `a` *and* `b` *and generate outputs them as* `min(a,b)` *and* `max(a,b)`*. The logic symbol of ESORT is represented in Figure 3.17b.* ◇

The internal organization of an ESORT is represented in Figure 3.17a. If $COMP_n$ is implemented in an optimal version, then this circuit is optimal because its size is linear and its depth is logarithmic.

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

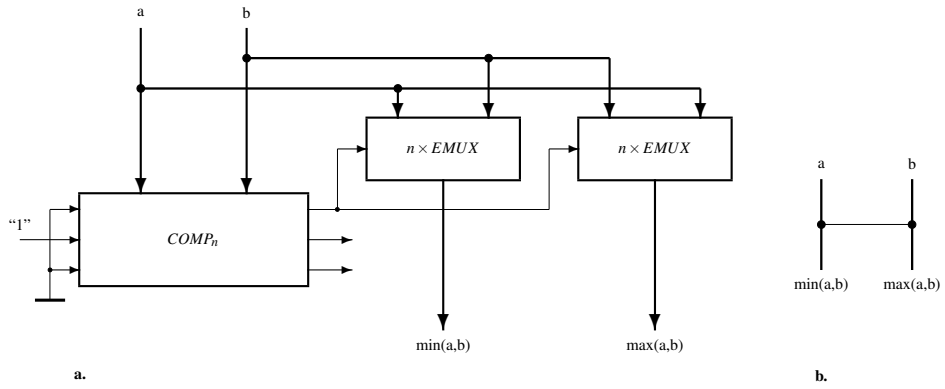


Figure 3.17: **The elementary sorter. a.** The internal structure of an elementary sorter. The output `lt_out` of the comparator is used to select the input values to output in the received order (if `lt_out = 1`) or in the crossed order (if `lt_out = 0`). **b.** The logic symbol of an elementary sorter.

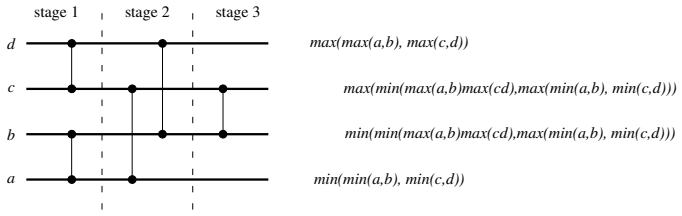


Figure 3.18: **The 4-input sorter.** The 4-input sorter is a three-stage combinational circuit built by 5 elementary sorters.

The circuit for sorting a vector of n numbers is build by ESORTs organized on many stages. The resulting combinational circuit receives the input vector $(x_1, x_2, x_3, \ldots x_n)$ and generates the sorted version of it. In Figure 3.18 is represented a small network of ESORTs able to sort the vector of integers (a,b,c,d). The sorted is organized on three stages. On the first stage two ESORTs are used sort separately the sub-vectors (a,b) and (c,d). On the second stage, the minimal values and the maximal values obtained from the previous stage are sorted, resulting the the smallest value (the minimal of the minimals), the biggest value (the maximal of the maximal) and the two intermediate values. For the last two the third level contains the last ESORT which sorts the middle values.

The resulting 4-input sorting circuit has the depth $D_{sort}(4) = 3 \times D_{esort}(n)$ and the size $S_{sort}(4) = 5 \times S_{esort}(n)$, where n is the number of bits used to represent the sorted integers.

Bathcer's sorter

What is the rule to design a sorter for a n-number sequence? This topic will pe presented using [Batcher '68], [Knuth '73] or [Parberry '87].

The n-number sequence sorter circuit, S_n, is presented in Figure 3.19. It is a double-recursive construct containing two $S_{n/2}$ modules and the *merge* module M_n, which has also a recursive definition,

because it contains two $M_{n/2}$ modules and $n/2-1$ elementary sorters, S_2. The M_n module is defined as a sorter which sorts the input sequence of n numbers only if it is formed by two $n/2$-number sorted sequences.

In order to prove that Batcher's sorter represented in Figure 3.19 sorts the input sequence we need the following theorem.

Theorem 3.2 *An n-input comparator network is a sorting network* iff *it works as sorter for all sequences of n symbols of zeroes and ones.* $\diamond$

The previous theorem is known as Zero-One Principle.

We must prove that, if $M_{n/2}$ are merge circuits, then M_n is a merge circuit. The circuit M_2 is an elementary sorter, S_2.

If $\{x_0,\ldots,x_{n-1}\}$ is a sequence of 0 and 1, and $\{a_0,\ldots,a_{n-1}\}$ is a sorted sequence with g zeroes, while $\{b_0,\ldots,b_{n-1}\}$ is a sorted sequence with h zeroes, then the left $M_{n/2}$ circuit receives $\lceil g/2\rceil+\lceil h/2\rceil$ zeroes[3], and the right $M_{n/2}$ circuit receives $\lfloor g/2\rfloor+\lfloor h/2\rfloor$ zeroes[4]. The value:

$$Z=\lceil g/2\rceil+\lceil h/2\rceil-(\lfloor g/2\rfloor+\lfloor h/2\rfloor)$$

takes only three values with the following output behavior for M_n:

Z=0 : at most one S_2 receives on its inputs the unsorted sequence $\{1,0\}$ and does it work, while all the above receive $\{0,0\}$ and the bellow receive $\{1,1\}$

Z=1 : $y_0=0$, follows a number of elementary sorters receiving $\{0,0\}$, while the rest receive $\{1,1\}$ and the last output is $y_{n-1}=1$

Z=2 : $y_0=0$, follows a number of elementary sorters receiving $\{0,0\}$, then one sorter with $\{0,1\}$ on its inputs, while the rest receive $\{1,1\}$ and the last output is $y_{n-1}=1$

Thus, no more than one elementary sorter is used to reverse the order in the received sequence.

The size and depth of the circuit is computed in two stages, corresponding to the two recursive levels of the definition. The size of the n-input merge circuit, $S_M(n)$, is iteratively computed starting with:

$$S_M(n)=2S_M(n/2)+(n/2-1)S_S(2)$$

$$S_M(2)=S_S(2)$$

Once the size of the merge circuit is obtained, the size of the n-input sorter, $S_S(n)$, is computed using:

$$S_S(n)=2S_S(n/2)+S_M(n)$$

Results:

$$S_S(n)=(n(log_2\,n)(-1+log_2\,n)/4+n-1)S_S(2)$$

A similar approach is used for the computation of the depth. The depth of the n-input merge circuit, $D_M(n)$, is iteratively computed using:

$$D_M(n)=D_M(n/2)+D_S(2)$$

[3] $\lceil a\rceil$ means rounded up integer part of the number a.

[4] $\lfloor a\rfloor$ means rounded down integer part of the number a.

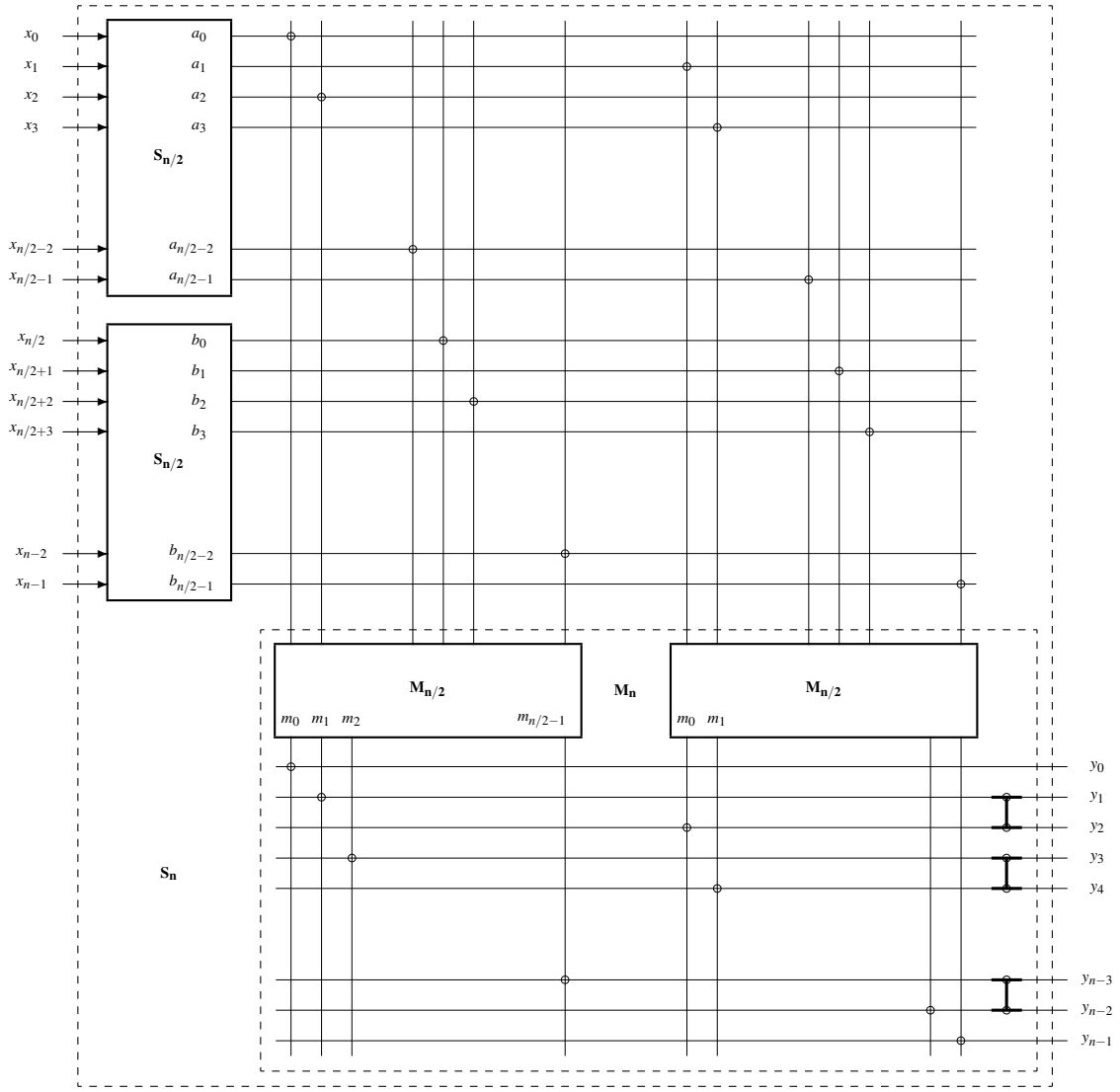


Figure 3.19: **Batcher's sorter.** The n-input sorter, S_n, is defined by a double-recursive construct: "$S_n = 2 \times S_{n/2} + M_n$", where the merger M_n consists of "$M_n = 2 \times M_{n/2} + (n/2-1)S_2$".

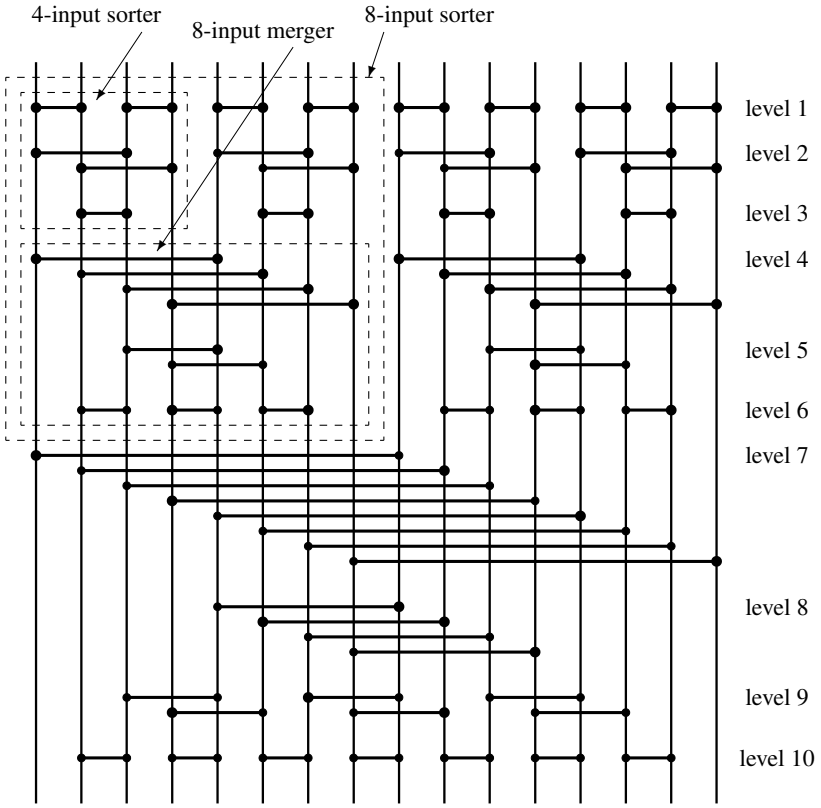


Figure 3.20: **16-input Batcher's sorter.**

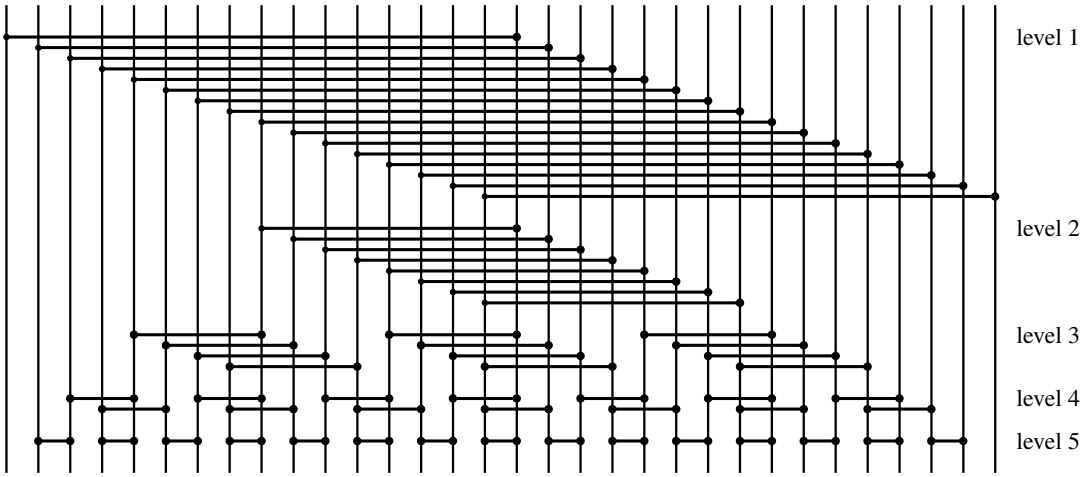


Figure 3.21: **32-input Batcher's merege circuit.**

$$D_M(2) = D_S(2)$$

while the depth of the n-input sorter, $D_S(n)$, is computed with:

$$D_S(n) = D_S(n/2) + D_M(n)$$

Results:

$$D_S(n) = (log_2\, n)(log_2\, n + 1)/2$$

We conclude that: $S_S(n) \in O(n \times log^2 n)$ and $D_S(n) \in O(log^2 n)$.

In Figure 3.20 a 16-input sorter designed according to the recursive rule is shown, while in Figure 3.21 a 32-input merge circuit is detailed. With 2 16-input sorters and a 32-input merger a 32-input sorted can be build.

In [Ajtai '83] is a presented theoretically improved algorithm, with $S_S(n) \in O(n \times log\, n)$ and $D_S(n) \in O(log\, n)$. But, the constants associated to the magnitude orders are too big to provide an optimal solution for currently realistic circuits characterized by $n < 10^9$.

The recursive Verilog description is very useful for this circuit because of the difficulty to describe in a HDL a double recursive circuit.

The top module describe the first level of the recursive definition: a n-input sorter is built using two $n/2$-input sorters and a n-input merger, and the 2-input sorter is the elementary sorter. Results the following description in Verilog:

```
module sorter #(`include "0_parameters.v")
        (   output  [m*n-1:0]   out ,
            input   [m*n-1:0]   in  );

    wire    [m*n/2-1:0] out0;
    wire    [m*n/2-1:0] out1;

    generate
    if (n == 2) eSorter eSorter(.out0   (out[m-1:0]     ),
                                .out1   (out[2*m-1:m]   ),
                                .in0    (in[m-1:0]      ),
                                .in1    (in[2*m-1:m]    ));
        else    begin   sorter #(.n(n/2))   sorter0(.out(out0           ),
                                                    .in (in[m*n/2-1:0]  )),
                                                sorter1(.out(out1           ),
                                                    .in (in[m*n-1:m*n/2]));
                        merger  #(.n(n))    merger( .out(out            ),
                                                    .in ({out1, out0}   ));
                end
    endgenerate
endmodule
```

For the elementary sorter, `eSorter`, we have the following description:

```
module eSorter #(`include "0_parameters.v")
        (   output  [m-1:0] out0,
            output  [m-1:0] out1,
            input   [m-1:0] in0 ,
            input   [m-1:0] in1 );
```

```
    assign out0 = (in0 > in1) ? in1 : in0   ;
    assign out1 = (in0 > in1) ? in0 : in1   ;
endmodule
```

The n-input merge circuit is described recursively using two $n/2$-input merge circuits and the elementary sorters as follows:

```
module merger #(`include "0_parameters.v")
        (   output  [m*n-1:0]   out ,
            input   [m*n-1:0]   in  );

    wire    [m*n/4-1:0] even0   ;
    wire    [m*n/4-1:0] odd0    ;
    wire    [m*n/4-1:0] even1   ;
    wire    [m*n/4-1:0] odd1    ;
    wire    [m*n/2-1:0] out0    ;
    wire    [m*n/2-1:0] out1    ;

    genvar i;
    generate
    if (n == 2) eSorter eSorter(.out0   (out[m-1:0]     ),
                                .out1   (out[2*m-1:m]   ),
                                .in0    (in[m-1:0]      ),
                                .in1    (in[2*m-1:m]    ));
     else   begin
            for (i=0; i<n/4; i=i+1) begin : oddEven
                assign  even0[(i+1)*m-1:i*m] = in[2*i*m+m-1:2*i*m]                  ;
                assign  even1[(i+1)*m-1:i*m] = in[m*n/2+2*i*m+m-1:m*n/2+2*i*m]      ;
                assign  odd0[(i+1)*m-1:i*m] = in[2*i*m+2*m-1:2*i*m+m]               ;
                assign  odd1[(i+1)*m-1:i*m] = in[m*n/2+2*i*m+2*m-1:m*n/2+2*i*m+m]   ;
            end
            merger #(.n(n/2))   merger0(.out(out0           ),
                                        .in ({even1, even0} )),
                                merger1(.out(out1           ),
                                        .in ({odd1, odd0}   ));
            for (i=1; i<n/2; i=i+1) begin : elSort
                eSorter eSorter(.out0   (out[(2*i-1)*m+m-1:(2*i-1)*m]   ),
                                .out1   (out[2*i*m+m-1:2*i*m]           ),
                                .in0    (out0[i*m+m-1:i*m]              ),
                                .in1    (out1[i*m-1:(i-1)*m]            ));
            end
            assign out[m-1:0]           = out0[m-1:0]               ;
            assign out[m*n-1:m*(n-1)]   = out1[m*n/2-1:m*(n/2-1)]   ;
            end
    endgenerate
endmodule
```

The parameters of the sorter are defined in the file `0_parameters.v`:

```
 parameter  n = 16, // number of inputs
            m = 8   // number of bits per input
```

3.1.10 First detection network

Let be the vector of Boolean variable: `inVector = {x0, x1, ... x(n-1)}`. The function `firstDetect` outputs three vectors of the same size:

```
first       = {0, 0, ... 0, 1, 0, 0, ... 0}
beforeFirst = {1, 1, ... 1, 0, 0, 0, ... 0}
afterFirst  = {0, 0, ... 0, 0, 1, 1, ... 1}
```

indicating, by turn, the position of the first 1 in `inVector`, all the positions before the first 1, and all the positions after the first 1.

Example 3.5 *Let be a 16-bit input circuit performing the function* `firstDetect`.

```
inVector    = {0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1}

first       = {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
beforeFirst = {1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
afterFirst  = {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

◇

The circuit performing the function `firstDetect` is described by the following Verilog program:

```
module firstDetect #(parameter n =4)( input  [0:n-1] in         ,
                                      output [0:n-1] first      ,
                                      output [0:n-1] beforeFirst,
                                      output [0:n-1] afterFirst );
   wire [0:n-1] orPrefixes;

   or_prefixes prefixNetwork(.in (in        ),
                             .out(orPrefixes));

   assign first       = orPrefixes & ~(orPrefixes >> 1),
          beforeFirst = ~orPrefixes                    ,
          afterFirst  = (orPrefixes >> 1)              ;
endmodule

module or_prefixes #(parameter n =4)(input      [0:n-1] in ,
                                     output reg [0:n-1] out);
   integer k;
   always @(in) begin out[0] = in[0];
                      for (k=1; k<n; k=k+1) out[k] = in[k] | out[k-1];
                end
endmodule
```

The function `firstDetect` classifies each component of a Boolean vector related to the first occurrence of the value 1[5].

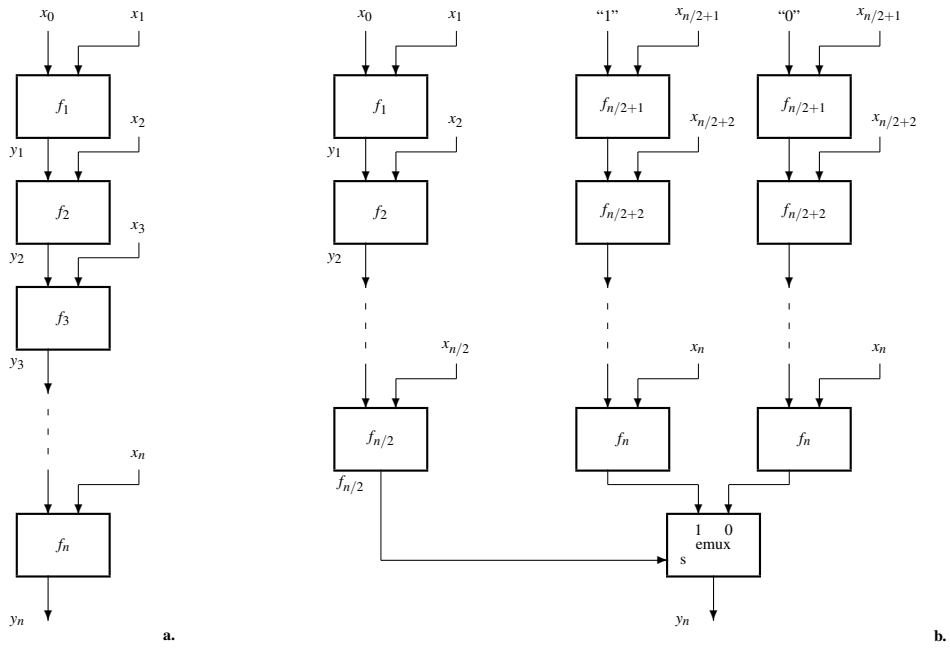


Figure 3.22:

3.1.11 Spira's theorem

3.2 Complex, Randomly Defined Circuits

3.2.1 An Universal circuit

Besides the simple, recursively defined circuits there is the huge class of the complex or random circuits. Is there a *general solution* for these category of circuits? A general solution asks a general circuit and this circuit surprisingly exists. Now rises the problem of how to catch the huge diversity of random in this approach. The following theorem will be the first step in solving the problem.

Theorem 3.3 *For any n, all the functions of n binary-variables have a solution with a combinational logic circuit.* $\diamond$

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

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

where:

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

[5]The function `firstDetect` becomes very meaningful related to the *minimalization* rule in Kleene's computational model [Kleene '36].

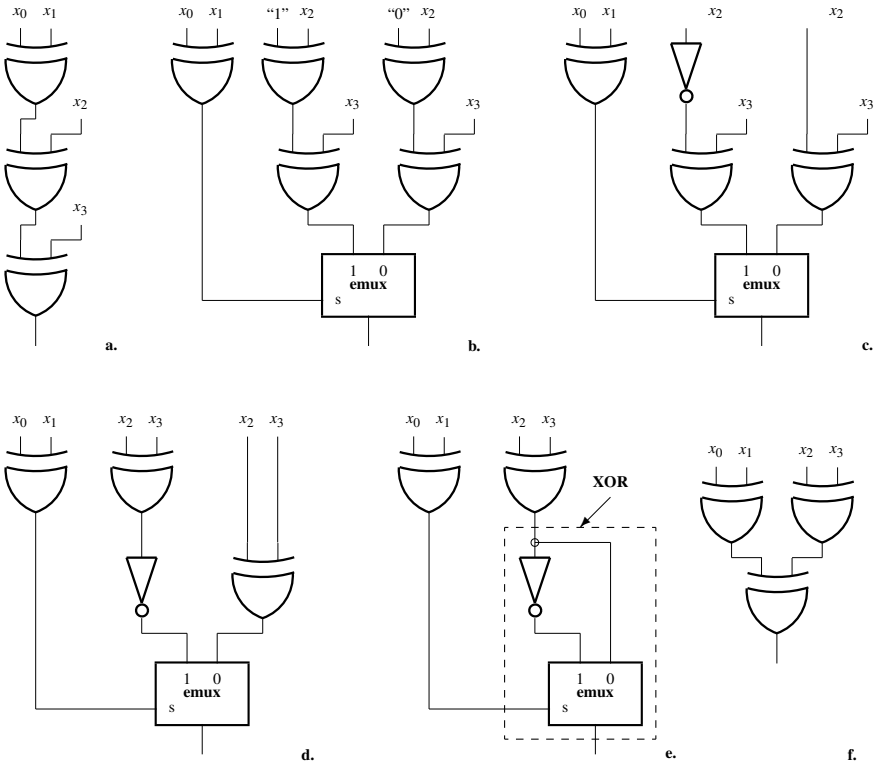


Figure 3.23:

Therefore, the computation of any n-variable function can be reduced to the computation of two other $(n-1)$-variables functions and an EMUX. The circuit, and in the same time the algorithm, is represented in Figure 3.24. For the functions g and h the same rule may applies. And so on until the two zero-variable functions: the value 0 and the value 1. The solution is an n-level binary tree of EMUXs having applied to the last level zero-variable functions. Therefore, solution is a MUX_n and a binary string applied on the 2^n selected inputs. The binary sting has the length 2^n. Thus, for each of the 2^{2^n} functions there is a distinct string defining it. $\diamond$

The universal circuit is indeed the best example of a big simple circuit, because it is described by the following code:

```
module nU_circuit #(‘include "parameter.v")
    (   output                      out     ,
        input   [(1’b1 << n)-1:0]   program ,
        input   [n-1:0]             data    );
```

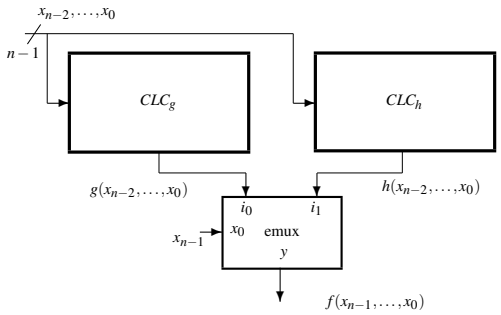


Figure 3.24: **The universal circuit.** For any $CLC_f(n)$, where $f(x_{n-1},\ldots,x_0)$ this recursively defined structure is a solution. EMUX behaves as an elementary universal circuit.

```
    assign  out = program[data];
endmodule
```

The file `parameter.v` contains the value for n. But, attention! The size of the circuit is: $S_{nU\_circuit}(n) \in O(2^n)$.

Thus, *circuits are more powerful than Turing Machine* (TM), because TM solve only problem having the solution algorithmically expressed with a sequence of symbols that does not depend by n. Beyond the Turing-computable function there are many functions for which the solution is a *family of circuits*.

The solution imposed by the previous theorem is an universal circuit for computing the n binary variable functions. Let us call it n**U-circuit** (see Figure 3.25). The size of this circuit is $S_{universal}(n) \in O(2^n)$ and its complexity is $C_{universal}(n) \in O(1)$. The functions is specified by the "program" $P = m_{p-1}, m_{p-2}, \ldots m_0$ which is applied on the selected inputs of the n-input multiplexer MUX_n. It is about a huge simple circuit. The functional complexity is associated with the "program" P, which is a binary string.

This universal solution represents the strongest **segregation** between a *simple physical structure* - the n-input MUX - and a *complex symbolic structure* - the string of 2^n bits applied on the selected inputs which works like a "program". Therefore, this is THE SOLUTION, MUX is THE CIRCUIT and we can stop here our discussion about digital circuits!? ... **But, no**! There are obvious reasons to continue our walk in the world of digital circuits:

- first: the *exponential size* of the resulting physical structure
- second: the huge size of the "programs" which are in a tremendous majority represented by random, uncompressible, strings (hard or impossible to be specified).

  The strongest segregation between simple and complex is not productive in no-loop circuits. Both resulting structure, the simple circuit and the complex binary string, grow exponentially.

3.2.2 Using the Universal circuit

We have a chance to use MUX_n to implement $f(x_{n-1},\ldots,x_0)$ only if one of the following conditions is fulfilled:

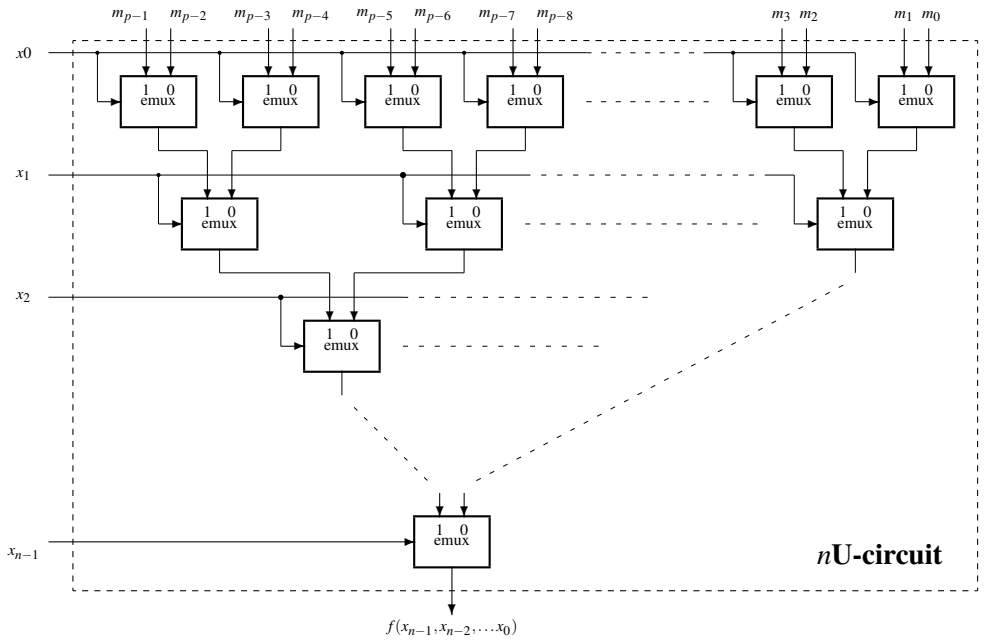


Figure 3.25: **The Universal Circuit as a tree of EMUXs.** The depth of the circuit is equal with the number, n, of binary input variables. The size of the circuit increases exponentially with n.

1. n is small enough resulting realizable and efficient circuits
2. the "program" is a string with useful regularities (patterns) allowing strong minimization of the resulting circuit

Follows few well selected examples. First is about an application with n enough small to provide an useful circuit (it is used in Connection Machine as an "universal" circuit performing anyone of the 3-input logic function [Hillis '85]).

Example 3.6 *Let be the following Verilog code:*

```
module three_input_functions(  output          out             ,
                               input   [7:0]   func            ,
                               input           in0, in1, in2   );
   assign  out = func[{in0, in1, in2}];
endmodule
```

The circuit `three_input_functions` *can be programmed, using the 8-bit string* `func` *as "program", to perform anyone 3-input Boolean function. It is obviously a MUX_3 performing*

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

where the function f is specified ("programmed") by an 8-bit word ("program"). $\diamond$

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

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

where f is "programmed" by a 16-bit word applied on the selected inputs of the multiplexer.

The bad news is: we can not go to far on this way because the size of the resulting universal circuits increases exponentially. The good news is: usually we do not need universal but particular solution. The circuits are, in most of cases, specific not universal. They "execute" a specific "program". But, when a specific binary word is applied on the selected inputs of a multiplexer, the actual circuit is minimized using the following **removing rules** and **reduction rules**.

An EMUX defined by:

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

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

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

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

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

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

- **if** $\{in1, in0\} = y0$ **then** $out = xy$
- **if** $\{in1, in0\} = y1$ **then** $out = y + x'$
- **if** $\{in1, in0\} = 0y$ **then** $out = x'y$
- **if** $\{in1, in0\} = 1y$ **then** $out = x + y$

Results: the first level of 2^{n-1} EMUXs of a MUX_n is reduced, and on the inputs of the second level (of n^{n-2} EMUXs) is applied a word containing binary values (0s and 1s) and binary variables (x_0s and x_0's). For the next levels the removing rules or the reducing rules are applied.

Example 3.7 *Let us solve the problem of majority function for three Boolean variable. The function $maj(x_2, x_1, x_0)$ returns 1 if the majority of inputs are 1, and 0 if not. In Figure 3.26 a "programmable" circuit is used to solve the problem.*

Because we intend to use the circuit only for the function $maj(x_2, x_1, x_0)$ the first layer of EMUXs can be removed resulting the circuit represented in Figure 3.27a.

On the resulting circuit the reduction rules are applied. The result is presented in Figure 3.27b. $\diamond$

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

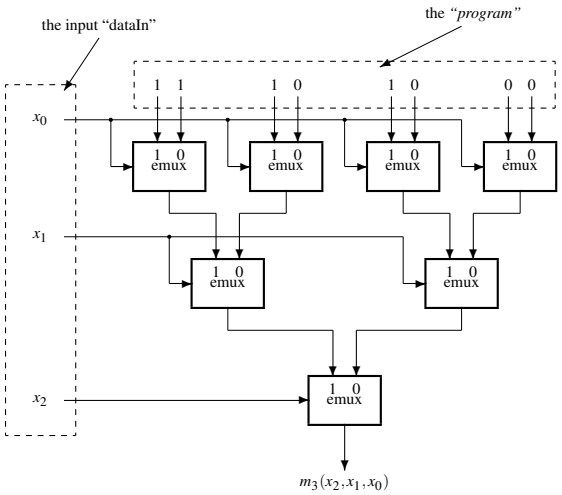


Figure 3.26: **The majority function.** The majority function for 3 binary variables is solved by a 3-level binary tree of EMUXs. The actual "program" applied on the "leafs" will allow to minimize the tree.

Example 3.8 *If the "program" is 128-bit string* $i_{127}\dots i_0 = (10)^{64}$*, it corresponds to a function of form:*

$$f(x_6,\dots,x_0)$$

where: the first bit, i_0 *is the value associated to the input configuration* $x_6,\dots,x_0 = 0000000$ *and the last bit* i_{127} *is the value associated to input configuration* $x_6,\dots,x_0 = 1111111$ *(according with the representation from Figure* **??** *which is equivalent with Figure 3.24).*

The obvious regularities of the "program" leads our mind to see what happened with the resulting tree of EMUXs. Indeed, the structure collapses under this specific "program". The upper layer of 64 EMUXs are selected by x_0 *and each have on their inputs* $i_0 = 1$ *and* $i_1 = 0$*, generating* x_0' *on their outputs. Therefore, the second layer of EMUXs receive on all selected inputs the value* x_0'*, and so on until the output generates* x_0'*. Therefore, the circuit performs the function* $f(x_0) = x_0'$ *and the structure is reduced to a simple inverter.*

In the same way the "program" $(0011)^{32}$ *programs the 7-input MUX to perform the function* $f(x_1) = x_1$ *and the structure of EMUXs disappears.*

For the function $f(x_1,x_0) = x_1x_0'$ *the "program" is* $(0010)^{32}$*.*

For a 7-input AND the"program" is $0^{127}1$*, and the tree of MUXs degenerates in 7 EMUXs serially connected each having the input* i_0 *connected to 0. Therefore, each EMUX become an* AND_2 *and applying the associativity principle results an* AND_7*.*

In a similar way, the same structure become an OR_7 *if it is "programmed" with* 01^{127}*.* $\diamond$

It is obvious that if the "program" has some uniformities, these uniformities allow to minimize the size of the circuit in polynomial limits using *removing* and *reduction* rules. *The simple "programs" lead toward small circuits*.

3.3 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

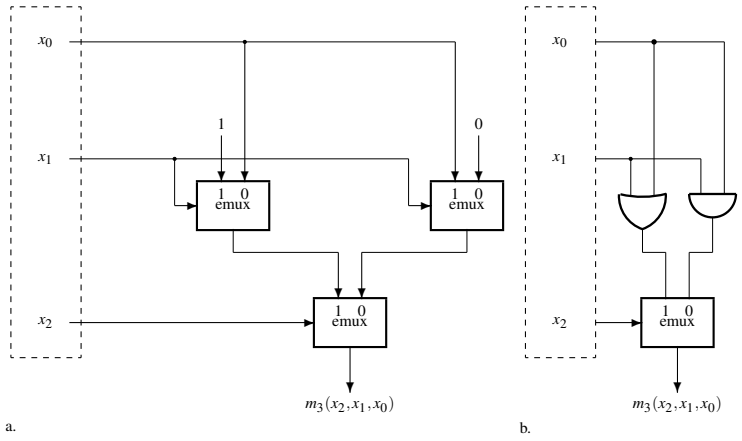


Figure 3.27: **The reduction process. a.** For any function the first level of EMUSs is reduced to a binary vector $((1,0)$ in this example). **b.** For the actual "program" of the 3-input majority function the second level is supplementary reduced to simple gates (an AND_2 and an OR_2).

are a lot of important circuits undiscussed in this chapter. Some of them are introduced as problems at the end of this chapter.

Simple circuits vs. complex circuits Two very distinct class of combinational circuits are emphasized. The first contains simple circuits, the second contains complex circuits. The complexity of a circuit is distinct from the size of a circuit. Complexity of a circuit is given by the size of the definition used to specify that circuit. Simple circuits can achieve big sizes because they are defined using a repetitive pattern. A complex circuit can not be very big because its definition is dimensioned related with its size.

Simple circuits have recursive definitions Each simple circuit is defined initially as an elementary module performing the needed function on the smallest input. Follows a recursive definition about how can be used the elementary circuit to define a circuit working for any input dimension. Therefore, any big simple circuit is a network of elementary modules which expands according to a specific rule. Unfortunately, the actual HDL, Verilog included, are not able to manage without (strong) restrictions recursive definitions neither in simulation nor in synthesis. The recursiveness is a property of simple circuits to be fully used only for our mental experiences.

Speeding circuits means increase their size Depth and size evolve in opposite directions. If the speed increases, the pay is done in size, which also increases. We agree to pay, but in digital systems the pay is not fair. We conjecture the bigger is performance the bigger is the unit price. Therefore, the pay increases more than the units we buy. It is like paying urgency tax. If the speed increases n times, then the size of the circuit increases more than n times, which is not fair but it is real life and we must obey.

Big sized complex circuits require programmable circuits There are software tolls for simulating and synthesizing complex circuits, but the control on what they generate is very low. A higher level of control we have using programmable circuits such as ROMs or PLA. PLA are efficient only if non-

arithmetic functions are implemented. For arithmetic functions there are a lot of simple circuits to be used. ROM are efficient only if the randomness of the function is very high.

Circuits represent a strong but ineffective computational model Combinational circuits represent a *theoretical* solution for any Boolean function, but not an effective one. Circuits can do more than algorithms can describe. The price for their universal completeness is their ineffectiveness. In the general case, both the needed physical structure (a tree of EMUXs) and the symbolic specification (a binary string) increase exponentially with n (the number of binary input variables). More, in the general case only a *family of circuits* represents the solution.

To provide an effective computational tool new features must be added to a digital machine and some restrictions must be imposed on what is to be computable. The next chapters will propose improvements induced by successively closing appropriate loops inside the digital systems.

3.4 Problems

Gates

Problem 3.1 *Determine the relation between the total number, N, of n-input m-output Boolean functions ($f : \{0,1\}^n \rightarrow \{0,1\}^m$) and the numbers n and m.*

Problem 3.2 *Let be a circuit implemented using 32 3-input AND gates. Using the appendix evaluate the area if 3-input gates are used and compare with a solution using 2-input gates. Analyze two cases: (1) the fan-out of each gate is 1, (2) the fan-out of each gate is 4.*

Decoders

Problem 3.3 *Draw DCD_4 according to Definition 2.9. Evaluate the area of the circuit, using the cell library from Appendis E, with the* placement efficiency[6] 70%. *Estimate the maximum propagation time. The wires are considered enough short to be ignored their contribution in delaying signals.*

Problem 3.4 *Design a constant depth DCD_4. Draw it. Evaluate the area and the maximum propagation time using the cell library from Appendix E. Compare the results with the results of the previous problem.*

Problem 3.5 *Propose a recursive definition for DCD_n using EDMUXs. Evaluate the size and the depth of the resulting structure.*

Multiplexors

Problem 3.6 *Draw MUX_4 using EMUX. Make the structural Verilog design for the resulting circuit. Organize the Verilog modules as hierarchical as possible. Design a tester and use it to test the circuit.*

Problem 3.7 *Define the 2-input XOR circuit using an EDCD and an EMUX.*

Problem 3.8 *Make the Verilog behavioral description for a constant depth left shifter by maximum $m-1$ positions for m-bit numbers, where $m = 2^n$. The "header" of the project is:*

[6]For various reason the area used to place gates on Silicon can not completely used. Some unused spaces remain between gates. Area efficiency measures the degree of area use.

```
module left_shift( output  [2m-2:0]    out     ,
                   input   [m-1:0]     in      ,
                   input   [n-1:0]     shift   );
   ...
endmodule
```

Problem 3.9 *Make the Verilog structural description of a log-depth (the depth is $log_2 16 = 4$) left shifter by 16 positions for 16-bit numbers. Draw the resulting circuit. Estimate the size and the depth comparing the results with a similar shifter designed using the solution of the previous problem.*

Problem 3.10 *Draw the circuit described by the Verilog module* `leftRotate` *in the subsection Shifters.*

Problem 3.11 *A barrel shifter for m-bit numbers is a circuit which rotate the bits the input word a number of positions indicated by the shift code. The "header" of the project is:*

```
module barrel_shift(   output  [m-1:0] out     ,
                       input   [m-1:0] in      ,
                       input   [n-1:0] shift   );
   ...
endmodule
```

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

Prefix network

Problem 3.12 *A prefix network for a certain associative function f,*

$$P_f(x_0, x_1, \ldots x_{n-1}) = \{y_0, y_1, \ldots y_{n-1}\}$$

receives n inputs and generate n outputs defined as follows:

$y_0 = f(x_0)$
$y_1 = f(x_0, x_1)$
$y_2 = f(x_0, x_1, x_2)$
...
$y_{n-1} = f(x_0, x_1, \ldots x_{n-1})$.

Design the circuit $P_{OR}(n)$ for $n = 16$ in two versions: (1) with the smallest size, (2) with the smallest depth.

Problem 3.13 *Design $P_{OR}(n)$ for $n = 8$ and the best product size $\times$ depth.*

Problem 3.14 *Design $P_{addition}(n)$ for $n = 4$. The inputs are 8-bit numbers. The addition is a mod256 addition.*

Recursive circuits

Problem 3.15 *A comparator is circuit designed to compare two n-bit positive integers. Its definition is:*

```
module comparator( input   [n-1:0] in1 ,   // first operand
                   input   [n-1:0] in2 ,   // second operand
                   output          eq  ,   // in1 = in2
                   output          lt  ,   // in1 < in2
                   output          gt  );  // in1 > in2
   ...
endmodule
```

1. *write the behavioral description in Verilog*
2. *write a structural description optimized for size*
3. *design a tester which compare the results of the simulations of the two descriptions: the behavioral description and the structural description*
4. *design a version optimized for depth*
5. *define an expandable structure to be used in designing comparators for bigger numbers in two versions: (1) optimized for depth, (2) optimized for size.*

Problem 3.16 *Design a comparator for signed integers in two versions: (1) for negative numbers represented in 2s complement, (2) for negative numbers represented a sign and number.*

Problem 3.17 *Design an expandable priority encoder with* minimal size *starting from an elementary priority encoder, EPE, defined for* $n = 2$*. Evaluate its depth.*

Problem 3.18 *Design an expandable priority encoder,* $PE(n)$*, with* minimal depth*.*

Problem 3.19 *What is the numerical function executed by a priority encoder circuit if the input is interpreted as an n-bit integer, the output is an m-bit integer and n_ones is a specific warning signal?*

Problem 3.20 *Design the Verilog structural descriptions for an 8-input adder in two versions: (1) using 8 FAs and a ripple carry connection, (2) using 8 HAs and a carry look ahead circuit. Evaluate both solutions using the cell library from Appendix E.*

Problem 3.21 *Design an expandable carry look-ahead adder starting from an elementary circuit.*

Problem 3.22 *Design an enabled incrementer/decrementer circuit for n-bit numbers. If* $en = 1$*, then the circuit increments the input value if* $inc = 1$ *or decrements the input value if* $inc = 0$*, else, if* $en = 0$*, the output value is equal with the input value.*

Problem 3.23 *Design an expandable adder/subtracter circuit for 16-bit numbers. The circuit has a carry input and a carry output to allow expandability. The 1-bit command input is sub. For* $sub = 0$ *the circuit performs addition, else it subtracts. Evaluate the area and the propagation time of the resulting circuit using the cell library from Appendix E.*

Problem 3.24 *Provide a "divide et impera" solution for the circuit performing* `firstDetect` *function.*

Random circuits

Problem 3.25 *The Gray counting means to count, starting from 0, so as at each step only one bit is changed. Example: the three-bit counting means 000, 001, 011, 010, 110, 111, 101, 100, 000, ... Design a circuit to convert the binary counting into the Gray counting for 8-bit numbers.*

Problem 3.26 *Design a converter from Gray counting to binary counting for n-bit numbers.*

Problem 3.27 *Write a Verilog structural description for ALU described in Example 2.3. Identify the longest path in the resulting circuit. Draw the circuit for* $n = 8$*.*

Problem 3.28 *Design a 8-bit combinational multiplier for $a_7, \ldots a_0$ and $b_7, \ldots b_0$, using as basic "brick" the following elementary multiplier, containing a FA and an AND:*

```
module em(carry_out, sum_out, a, b, carry, sum);
   input   a, b, carry, sum;
   output  carry_out, sum_out;

   assign {carry_out, sum_out} = (a & b) + sum + carry;
endmodule
```

Problem 3.29 *Design an adder for 32 1-bit numbers using the carry save adder approach.*
Hint*: instead of using the direct solution of a binary tree of adders a more efficient way (from the point of view of both size and depth) is to use circuits to "compact" the numbers. The first step is presented in Figure 3.28, where 4 1-bit numbers are transformed in two numbers, a 1-bit number and a 2-bit number. The process is similar in Figure 3.29 where 4 numbers, 2 1-bit numbers and 2 2-bit numbers are compacted as 2 numbers, one 2-bit number and one 3-bit number. The result is a smaller and a faster circuit than a circuit realized using adders.*

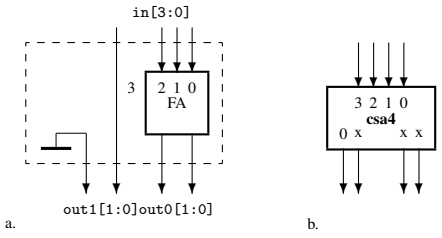


Figure 3.28: **4-bit compacter.**

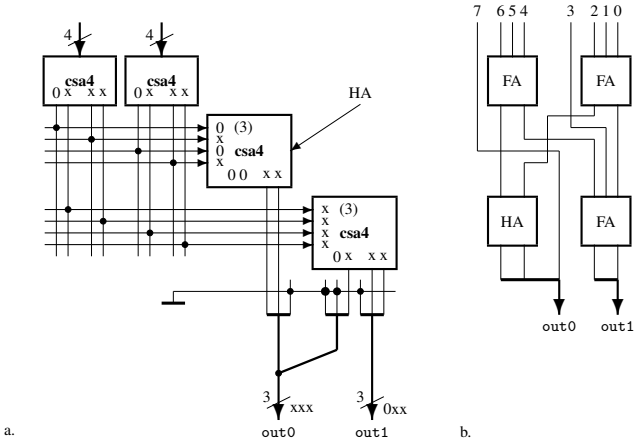


Figure 3.29: **8-bit compacter.**

Compare the size and depth of the resulting circuit with a version using adders.

Problem 3.30 *Design in Verilog the behavioral and the structural description of a multiply and accumulate circuit, MACC, performing the function:* $(a \times b) + c$*, where a and b are 16-bit numbers and c is a 24-bit number.*

Problem 3.31 *Design the combinational circuit for computing*

$$c = \sum_{i=0}^{7} a_i \times b_i$$

where: a_i, b_i *are 16-bit numbers. Optimize the size and the depth of the 8-number adder using a technique learned in one of the previous problem.*

Problem 3.32 *Exemplify the serial composition, the parallel composition and the serial-parallel composition in 0 order systems.*

Problem 3.33 *Write the logic equations for the BCD to 7-segment trans-coder circuit in both high active outputs version and low active outputs version. Minimize each of them individually. Minimize all of them globally.*

Problem 3.34 *Applying removing rules and reduction rules find the functions performed by 5-level universal circuit programmed by the following binary strings:*

1. $(0100)^8$
2. $(01000010)^4$
3. $(0100001011001010)^2$
4. $0^{24}(01000010)$
5. 00000001001001001111000011000011

Problem 3.35 *Compute the biggest size and the biggest depth of an n-input, 1-output circuit implemented using the universal circuit.*

Problem 3.36 *Provide the prof for Zero-One Principle.*

3.5 Projects

Project 3.1 *Finalize Project 1.1 using the knowledge acquired about the combinational structures in this chapter.*

Project 3.2 *Design a combinational floating point single precision (32 bit) multiplier according to the ANSI/IEEE Standard 754-1985, Standard for Binary Floating Point Arithmetic.*

Chapter 4

MEMORIES:
First order, 1-loop digital systems

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*.

4.1 Field Programmable Gate Array – FPGA

Few decades ago the prototype of a digital system was realized in a technology very similar with the one used for the final form of the product. Different types of standard integrated circuits where connected according to the design on boards using a more or less flexible interconnection technique. Now we do not have anymore standard integrated circuits, and making an Application Specific Integrated Circuit (ASIC) is a very expensive adventure. Fortunately, now there is a wonderful technology for prototyping (which can be used also for small production chains). It is based on a one-chip system called **Field Programmable Gate Array** – FPGA. The name comes from its flexibility to be configured by the user after manufacturing, i.e., "in the field". This generic circuit can be *programmed* to perform any digital function.

In this subsection the basic configuration of an FPGA circuit will be described[1]. The internal cellular structure of the system is described for the simplest implementation, letting aside details and improvements used by different producer on this very diverse market (each new generation of FPGA integrates different usual digital blocks in order to help efficient implementations; for example: multipliers, block RAMs, ...; learn more about this from the on-line documentation provided by the FPGA producers).

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

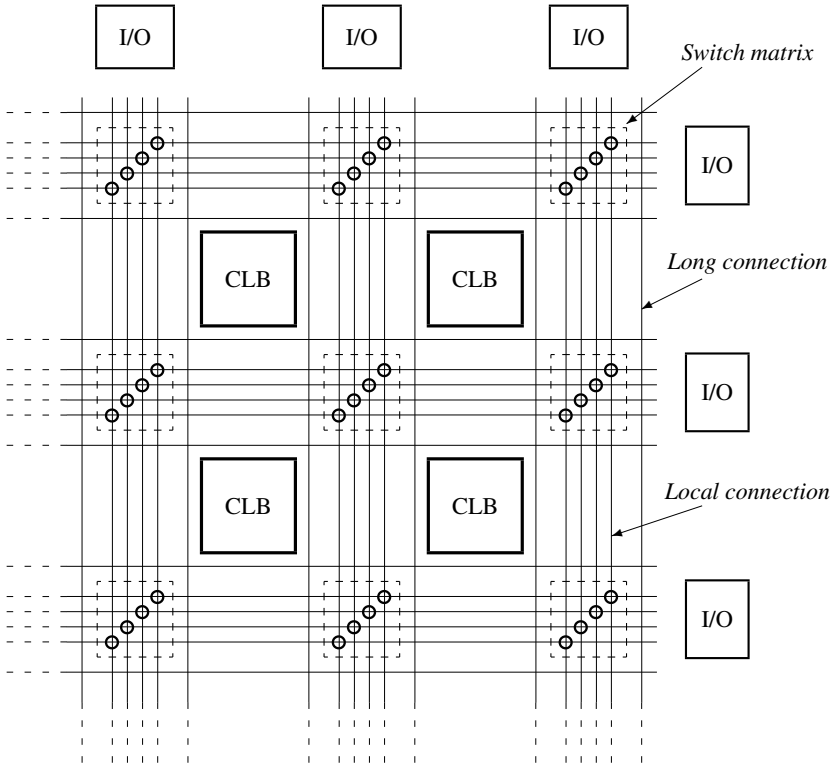


Figure 4.1: **Top level organization of FPGA.**

4.1.1 The system level organization of an FPGA

The FPGA chip has a cellular structure with three main programmable components, whose function is defined by setting on 0 or on 1 control bits stored in memory elements. An FPGA can be seen as a big structured memory containing million of bits used to control the state of million of switches. The main type of cells are:

- **Configurable Logic Blocks** (CLB) used to perform a programmable combinational and/or sequential function
- **Switch Nodes** which interconnect in the most flexible way the CLB modules and some of them to the IO pins, using a matrix of programmed switches
- **Input-Output Interfaces** are two-direction programmable interfaces, each one associated with an IO pin.

Figure 4.1 provides a simplified representation of the internal structure of an FPGA at the top level. The area of the chip is filled up with two interleaved arrays. One of the CLBs and another of the Switch Nodes. The chip is boarded by IO interfaces.

The entire functionality of the system can be programmed by an appropriate binary configuration distributed in all the cells. For each IO pin is enough one bit to define if the pin is an input or an output. For a Switch Node more bits are needed because each switch asks for 6 bits to be configured. But,

most of bits (in some implementations more than 100 per CLB) are used to program the functions of the combinational and sequential circuits in each node containing a CLB.

4.1.2 The IO interface

Each signal pin of the FPGA chip can be assigned to be an input or an output. The simplest form of the interface associated to each IO pin is presented in Figure 4.2, where:

- **D-FF0**: is the D master-slave flip-flop which synchronously receives the value of the I/O pin through the associated input non-inverting buffer
- **m**: the storage element which contains the 1-bit program for the input interface used to command the tristate buffer; if `m = 1` then the tristate buffer is enabled and interface is in the output mode, else the tristate buffer is disabled and interface is in the input mode
- **D-FF1**: is the flip-flop loaded synchronously with the output bit to be sent to the I/O pin if `m = 1`.

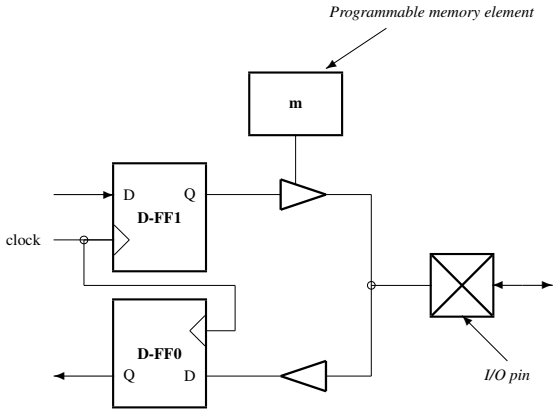


Figure 4.2: **Input-Output interface.**

The storage element `m` is part of the big distributed RAM containing all the storage elements used to program the FPGA.

4.1.3 The switch node

The switch node (Figure 4.3a) consists of a number of programmable switches (4 in our description). Each switch (Figure 4.3b) manages 4 wires, connection them in different configurations using 6 nMOS transistors, each commanded by the state of 1-bit memory (Figure 4.3c). If `mi = 1` then the associated nMOS transistor is *on* and between its drain end source the resistor has a small value. If `mi = 0` then the associated nMOS transistor is *off* and the two ends of the switch are not connected.

For example, the configuration shown in Figure 4.3d is programmed as follows:

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

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

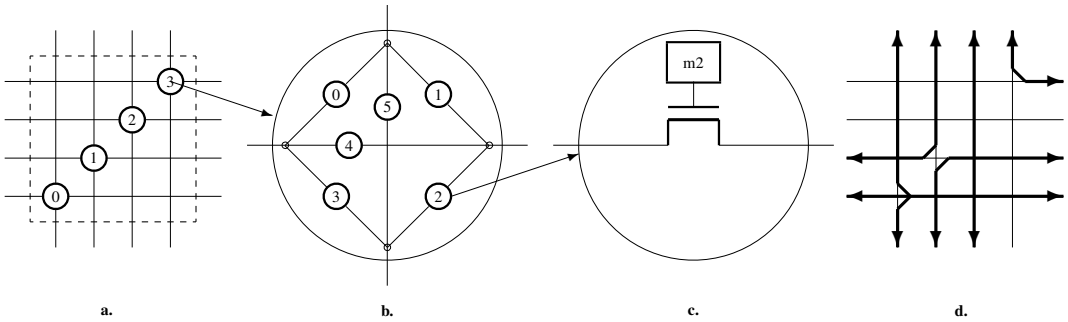


Figure 4.3: **The structure of a Switch Node. a.** A Switch Node with 4 switches. **b.** The organization of a switch. **c.** A line switch. **d.** An example of actual connections.

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

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

Any connection is a two-direction connection.

4.1.4 The basic building block

Because any digital circuit can be composed by properly interconnected gates and flip-flops, each CLB contains a number of basic building blocks, called **bit slices** (BSs), each able to provide at least an n-input, 1-output programmable combinational circuit and an 1-bit register.

In the previous chapter was presented an Universal combinational circuit: the n-input multiplexer able to perform any n-variable Boolean function. It was *programmed* applying on its selected inputs an m-bit binary configuration (where $m = 2^n$). Thereby, an MUX_n and a memory for storing the m-bit program provide the structure able to be programmed to perform any n-input 1-output combinational circuit. In Figure 4.4 it is represented, for $n = 4$, by the multiplexer MUX and the 16 memory elements m0, m1, ... m15. The entire sub-module is called LUT (from **look-up table**). The memory elements m0, m1, ... m15, being part of the big distributed RAM of the FPGA chip, can be loaded with any out of 65536 binary configuration used to define the same number of 4-input Boolean function.

Because the arithmetic operations are very frequently used the BS contains a specific circuit for any arithmetic operation: the circuit computing the value of the carry signal. The module **carry** Figure 4.4 has also its specific propagation path defined by a specific input, carryIn, and a specific output carryOut.

The BS module contains also the one-bit register D-FF. Its contribution can be considered in the current design if the memory element md is programmed appropriately. Indeed, if md = 1, then the output of the BS comes form the output of D-FF, else the output of the BS is a combinational one, the flip-flop being shortcut.

The memory element mc is used to program the selection of the **LUT** output or of the **Carry** output to be considered as the programmable combinational function of this BS.

The total number of bits used to program the function of the BS previously described is 18. Real FPGA circuits are now featured with much more complex BSs (please search on their web pages for details).

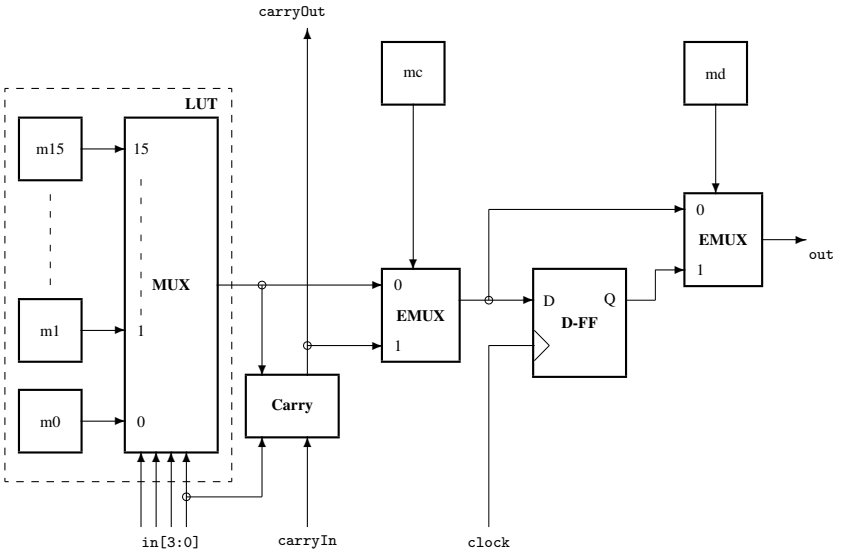


Figure 4.4: **The basic building block: the bit slice (BS).**

There are two kinds of BS: logic type and memory type. The logic type uses LUT to implement combinational functions. The memory type uses LUT for implementing both, combinatorial functions and memory function (RAM or serial shift register).

4.1.5 The configurable logic block

The main cell used to build an FPGA, CLB (see Figure 4.1) contains many BSs organized in slices. The most frequent organization is of 2 slices, each having 4 BSs (see Figure 4.5). There are slices containing logic type BSs (usually called SLICEL), or slices containing memory type BSs (usually called SLICEM). Some CLBs are composed by two SLICEL, others are composed by one SLICEL and one SLICEM.

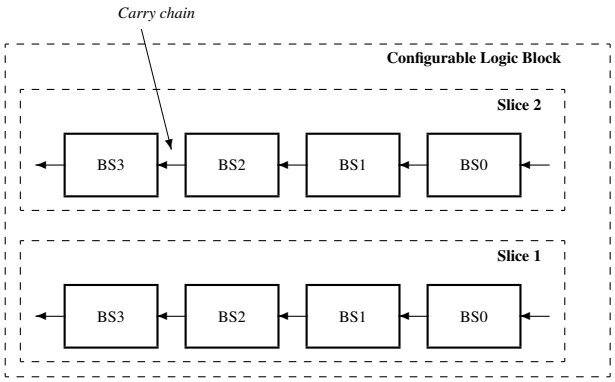


Figure 4.5: **Configurable logic block.**

A slice has some fix connections between its BSs. In our simple description, the fix connections refers to the carry chain connections. Obviously, we can afford to make fix connections for circuits having specific function.

4.2 Content Addressable Memory

A normal way to "question" a memory circuit is to ask for:

Q1: *the value of the property A of the object B*

For example: *How old is George?* The *age* is the property and the *object* is George. The first step to design an appropriate device to be questioned as previously is exemplified is to define a circuit able to answer the question:

Q2: *where is the object B?*

with two possible answers:

1. *the object B is not in the searched space*
2. *the object B is stored in the cell indexed by X.*

The circuit for answering Q2-type questions is called *Content Addressable Memory*, shortly: *CAM*. (About the question Q1 in the next subsection.)

The basic cell of a CAM is consists of:

- the storage elements for binary objects
- the "questioning" circuits for searching the value applied to the input of the cell.

In Figure 4.6 there are 4 D latches as storage elements and four XORs connected to a 4-input NAND used as comparator. The cell has two functions:

- **to store**: the active level of the clock modify the content of the cell storing the 4-bit input data into the four D latches
- **to search**: the input data is continuously compared with the content of the cell generating the signal $AO' = 0$ if the input matches the content.

The cell is *written* as an m-bit latch and is continuously *interrogated* using a combinational circuit as comparator. The resulting circuit is an 1-OS because results serially connecting a memory, one-loop circuit with a combinational, no-loop circuit. No additional loop is involved.

An n-word CAM contains n CAM cells and some additional combinational circuits for distributing the clock to the selected cell and for generating the global signal M, activated for signaling a successful match between the input value and one or more cell contents. In Figure 4.7a a 4-word of 4 bits each is represented. The write enable, WE, signal is demultiplexed as clock to the appropriate cell, according to the address codded by A_1A_0. The 4-input NAND generate the signal M. If, at least one address output, AO'_i is zero, indicating match in the corresponding cell, then $M = 1$ indicating a successful search.

The *input address* $A_{p-1},\ldots,A_0$ is binary codded on $p = log_2 n$ bits. The *output address* $AO_{n-1},\ldots,AO_0$ is an *unary code* indicating the place or the places where the data input $D_{m-1},\ldots,D_0$

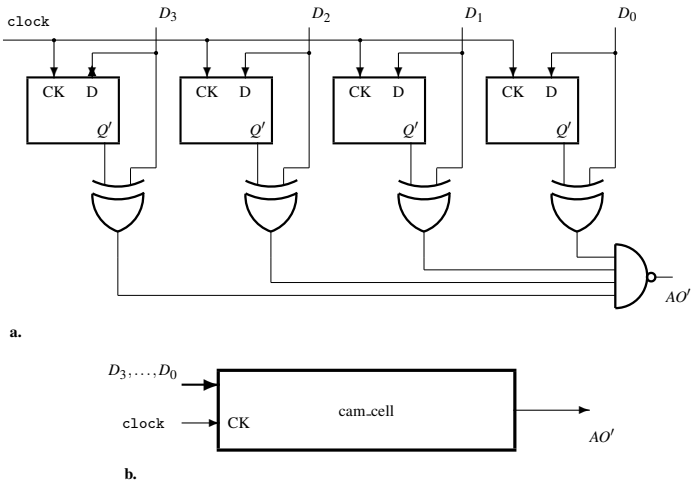


Figure 4.6: **The Content Addressable Cell. a.** The structure: data latches whose content is compared against the input data using 4 XORs and one NAND. Write is performed applying the clock with stable data input. **b.** The logic symbol.

matches the content of the cell. The output address must be unary codded because there is the possibility of match in more than one cell.

Figure 4.7b represents the logic symbol for a CAM with n m-bit words. The input WE indicate the function performed by CAM. Be very careful with the set-up time and hold time of data related to the WE signal!

The CAM device is used to locate an object (to answer the question **Q2**). Dealing with the properties of an object (answering **Q1**-type questions) means to use o more complex devices which *associate* one or more properties to an object. Thus, the *associative memory* will be introduced adding some circuits to CAM.

4.3 An Associative Memory

A partially used RAM can be an associative memory, but a very inefficient one. Indeed, let be a RAM addressed by $A_{n-1}\ldots A_0$ containing 2-field words $\{V,\ D_{m-1}\ldots D_0\}$. The *objects* are codded using the address, the *values* of the unique property P are codded by the data field $D_{m-1}\ldots D_0$. The one-bit field V is used as a validation flag. If $V = 1$ in a certain location, then there is a match between the object designated by the corresponding address and the value of property P designated by the associated data field.

Example 4.1 *Let be the 1Mword RAM addressed by $A_{19}\ldots A_0$ containing 2-field 17-bit words $\{V,\ D_{15}\ldots D_0\}$. The set of objects, OBJ, are codded using 20-bit words, the property P associated to OBJ is codded using 16-bit words. If*

$$RAM[11110000111100001111] = 1\_0011001111110000$$

$$RAM[11110000111100001010] = 0\_0011001111110000$$

then:

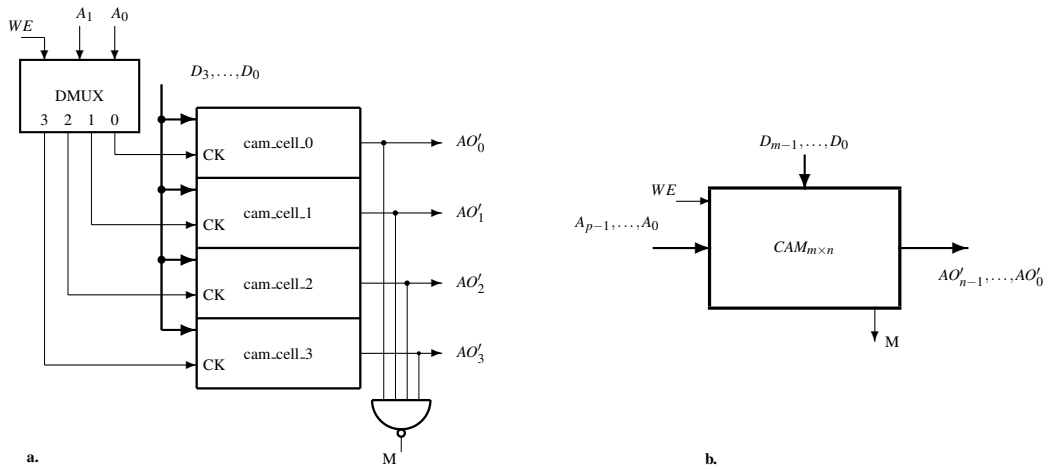


Figure 4.7: **The Content Addressable Memory (*CAM*). a.** A 4-word CAM is built using 4 content addressable cells, a demultiplexor to distribute the write enable (WE) signal, and a $NAND_4$ to generate the match signal (M). **b.** The logic symbol.

- *for the object* 11110000111100001111 *the property P is defined (V* $= 1$*) and has the value* 0011001111110000
- *for the object* 11110000111100001010 *the property P is not defined (V* $= 0$*) and the data field is meaningless.*

Now, let us consider the 20-bit address codes four-letter names using for each letter a 5-bit code. How many locations in this memory will contain the field V instantiated to 1? Unfortunately, only extremely few of them, because:

- *only 24 from 32 binary configurations of 5 bits will be used to code the 24 letters of Latin alphabet (*$24^4 < 2^{20}$*)*
- *but more important: how many different name expressed by 4 letters can be involved in a real application? Usually no more than few hundred, meaning almost nothing related to* 2^{20}.

◇

The previous example teaches us that a RAM used as associative memory is a very inefficient solution. In real applications are used names codded very inefficiently:

$$number\_of\_possible\_names >>> number\_of\_actual\_names.$$

In fact, the natural memory function means almost the same: to remember about something immersed in a huge set of possibilities.

One way to implement an efficient associative memory is to take a CAM and to use it as a *programmable decoder* for a RAM. The (extremely) limited subset of the actual objects are stored into a CAM, and the address outputs of the CAM are used instead of the output of a combinational decoder to

select the accessed location of a RAM containing the value of the property P. In Figure 4.8 this version of an associative memory is presented. $CAM_{m\times n}$ is usually dimensioned with $2^m >>> n$ working as a decoder programmed to decode **any** very small subset of n addresses expressed by m bits.

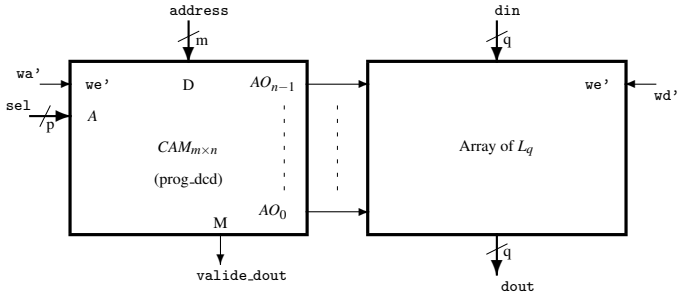


Figure 4.8: **An associative memory (*AM*).** The structure of an AM can be seen as a RAM with a programmable decoder implemented with a CAM. The decoder is programmed loading CAM with the considered addresses.

Here are the three working mode of the previously described associative memory:

define_object : write the name of an object to the selected location in CAM
```
wa' = 0, address = name_of_object, sel = cam_address
wd' = 1, din = don't_care
```

associate_value : write the associated value in the randomly accessed array to the location selected by the active address output of CAM
```
wa' = 1, address = name_of_object, sel = don't_care
wd' = 0, din = value
```

search : search for the value associated with the name of the object applied to the address input
```
wa' = 1, address = name_of_object, sel = don't_care
wd' = 1, din = don't_care
```
`dout` is valid only if `valide_dout = 1`.

This associative memory will be dimensioned according to the dimension of the actual subset of names, which is significantly smaller than the virtual set of the possible names ($2^p <<< 2^m$). Thus, for a searching space with the size in $O(2^m)$ a device having the size in $O(2^p)$ is used.

4.4 Beneš-Waxman Permutation Network

In 1968, even though it was defined by others before, Václav E. Beneš promoted a permutation network [Benes '68] and Abraham Waxman published an optimized version [Waksman '68] of it.

A permutation circuit is a network of programmable switching circuits which receives the sequence $x_1,\ldots,x_n$ and can be programmed to to provide on its outputs any of the $n!$ possible permutations.

The two-input programmable switching circuit is represented in Figure 4.9 a. It consists of a D-FF to store the programming bit and two 2-input multiplexors to perform the programmed switch. The circuit works as follows:

D-FF = 0 : $x'_1 = x_1$ and $x'_2 = x_2$

D-FF = 1 : $x_1' = x_2$ and $x_2' = x_1$

The input `enable` allows to load D-FF with the programming bit. The storage element is a master-slave structure because in a complex network the D flip-flops are chained because they are loaded with the programming bits by shifting. The logic symbol for this elementary circuit is represented in Figure 4.9 b (the clock input and enable input are omitted for simplicity).

Beneš-Waxman permutation network (we must recognize credit for this circuit, at least, for both, Beneš and Waxman) with n inputs has the recursive definition presented in Figure 4.9 c. (The only difference between the definition provided by Beneš and the optimization done by Waxman refers to the number of switches on the output layer: in Waxman's approach there are only $n/2-1$ switches, inetead of n for the version presented by Beneš.)

Theorem 4.1 *The switches of Beneš-Waxman permutation network can be set to realize any permutation.*

◇

Proof. For $n = 2$ the permutation network is a programmable switch circuit. For $n > 2$ we consider, for simplicity, n as a power of 2.

If the two networks $LeftP_{n/2}$ and $RightP_{n/2}$ are permutation networks with $n/2$ inputs, then we will prove that it is possible to program the input switches so as each sequence of two successive value on the outputs contains values that reach the output switch going through different permutation network. If the "local" order, on the output pair, is not the desired one, then the output switch is used to fix the problem by an appropriate programming.

The following steps can be followed to establish the programming Boolean sequence – $\{qIn_1, \ldots, qIn_{n/2}\}$ – for the $n/2$ input switches and the programming Boolean sequence – $\{qOut_2, \ldots, qOut_{n/2}\}$ – for the $n/2-1$ output switches:

1. because $y_1 \leftarrow x_i$, the input switch where x_i is connected is programmed to $qIn_{\lceil i/2 \rceil} = i + 1 - 2 \times \lceil i/2 \rceil$ in order to let x_i to be applied on the $LeftP_{n/2}$ permutation network (if i is an odd number, then $qIn_{\lceil i/2 \rceil} = 0$, else $qIn_{\lceil i/2 \rceil} = 1$); the $x_{i-(-1)^i}$ input is the "companion" of x_i on the same switch; it is consequently routed to the input of $RightP_{n/2}$

2. the output switch $\lceil j/2 \rceil$ is identified using the correspondence $y_j \leftarrow x_{i-(-1)^i}$; the state of the switch is set to $sOut_{\lceil j/2 \rceil} = j - 2 \times \lfloor j/2 \rfloor$ (if j is an odd number, then $qOut_{\lceil j/2 \rceil} = 1$, else $qOut_{\lceil j/2 \rceil} = 0$)

3. for $y_{j-(-1)^j} \leftarrow x_k$, the "companion" of y_j on the $j/2$-th output switch, we go back to the step 1 until in the step 2 we reach the second output, y_2; for the first two outputs there is no need of a switch because a partial or the total programming cycle ends when y_2 receives its value from the output of the $RightP_{n/2}$ permutation network

4. if all the output switches are programmed the process stops, else, we start again from the left output of an un-programmed output switch.

Any step 1, let us call it *up-step*, programs an input switch, while any step 2, let us call it *down-step*, programs an output switch. Any *up-step*, which solves the connection $y_i \leftarrow x_j$, is feasible because the source x_j is always connected to a still un-programmed switch. Similarly, any *down-step* is also feasible.

◇

The size and depth of the permutation network P_n is computed using the relations:

$$S_P(n) = (2 \times S_P(n/2) + n - 1) \times S_P(2)$$

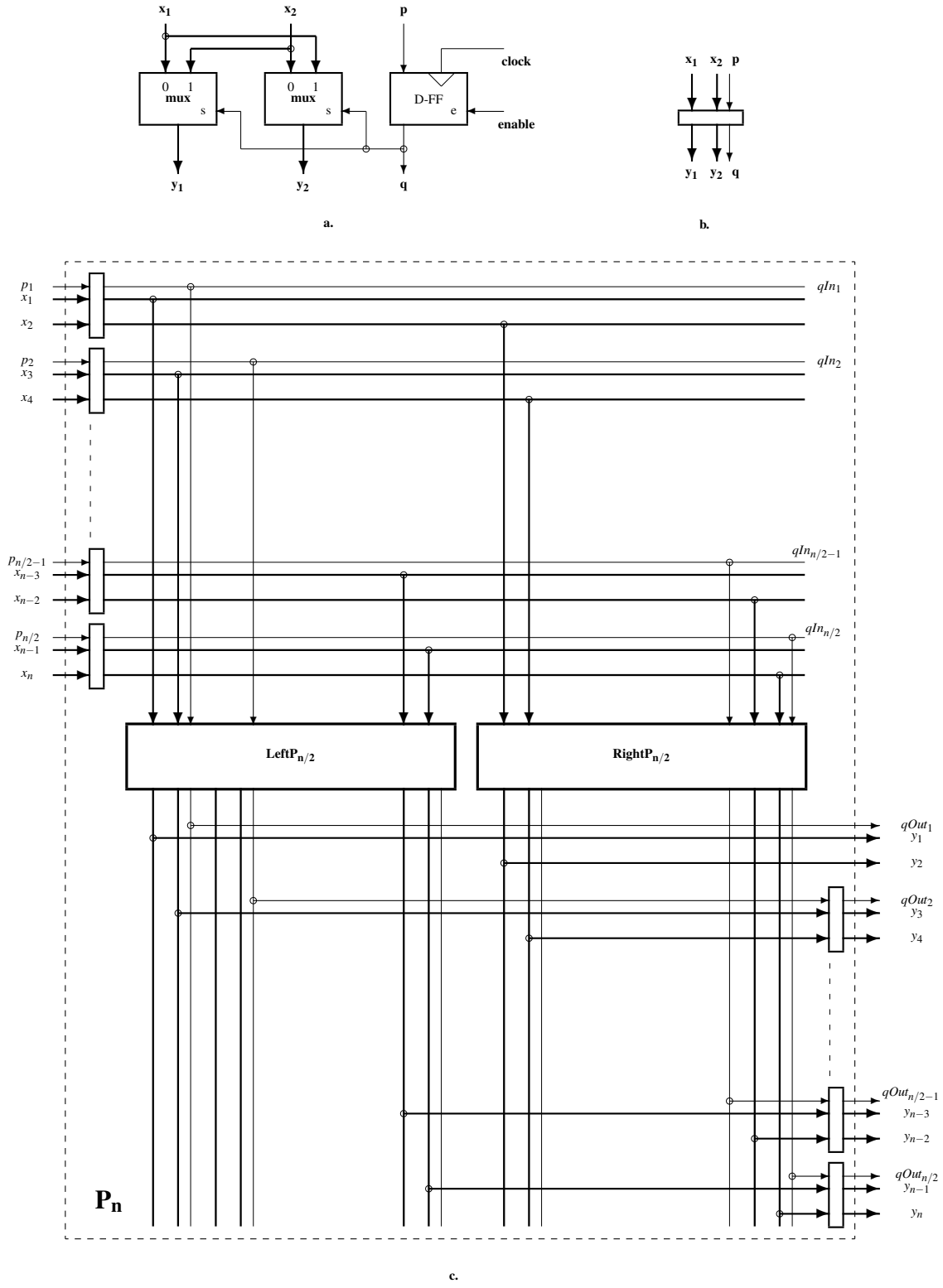


Figure 4.9: **Beneš-Waxman permutation network. a.** The programmable switching circuit. **b.** The logic symbol for the programmable switching circuit. **c.** The recursive definition of a n-input Beneš-Waxman permutation network, $\mathbf{P_n}$.

$$S_P(2) \in O(1)$$
$$D_P(n) = D_P(n) + 2 \times D_P(2)$$
$$D_P(2) \in O(1)$$

Results:

$$S_P(n) = (n \times log_2 n - n + 1) \times S_P(2) \in O(n log n)$$
$$D_P(n) = -1 + 2 \times log_2 n \in O(log n)$$

Example 4.2 *Let be an 8-input permutation network which must be programmed to perform the following permutation:*

$$\{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\} \rightarrow \{x_8, x_6, x_3, x_1, x_4, x_7, x_5, x_2\} = \{y_1, \ldots, y_8\}$$

The permutation network with 8 inputs is represented in Figure 4.10. It is designed starting from the recursive definition.

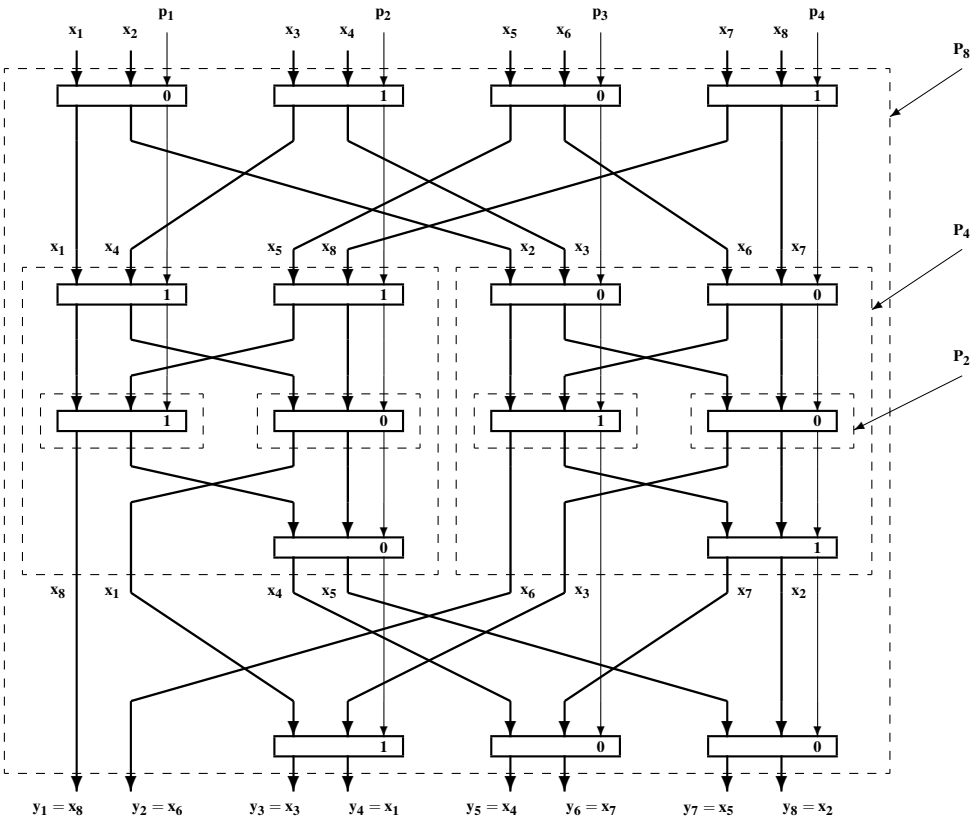


Figure 4.10: .

The programming bits for each switch are established according to the algorithm described in the previous proof. In the first stage the programming bits for the first layer – $\{p_{11}, p_{12}, p_{13}, p_{14}\}$ – and last layer – $\{p_{52}, p_{53}, p_{54}\}$ – are established, as follows:

1. $(y_1 = x_8) \Rightarrow (x_8 \rightarrow leftP_4) \Rightarrow (p_{14} = 1)$
2. $(p_{14} = 1) \Rightarrow (x_7 \rightarrow rightP_4) \Rightarrow (p_{53} = 0)$
3. $(y_5 = x_4) \Rightarrow (x_4 \rightarrow leftP_4) \Rightarrow (p_{12} = 1)$
4. $(p_{12} = 1) \Rightarrow (x_3 \rightarrow rightP_4) \Rightarrow (p_{52} = 1)$
5. $(y_4 = x_1) \Rightarrow (x_1 \rightarrow leftP_4) \Rightarrow (p_{11} = 0)$
6. $(p_{11} = 0) \Rightarrow (x_2 \rightarrow rightP_4) \Rightarrow (p_{54} = 0)$
7. $(y_7 = x_5) \Rightarrow (x_5 \rightarrow leftP_4) \Rightarrow (p_{13} = 0)$
8. $(p_{13} = 0) \Rightarrow (x_6 \rightarrow rightP_4) \Rightarrow$ *the first stage of programming closes successfully.*

In the second stage there are two P_4 permutation networks to be programmed: $leftP_4$ and $rightP_4$. From the first stage of programming resulted the following permutations to be performed:

for $leftP_4$**:** $\{x_1, x_4, x_5, x_8\} \rightarrow \{x_8, x_1, x_4, x_5\}$

for $rightP_4$**:** $\{x_2, x_3, x_6, x_7\} \rightarrow \{x_6, x_3, x_7, x_2\}$

The same procedure is applied now twice providing the programming bits for the second and the fourth layers of switches.

The last step generate the programming bits for the third layer of switches.

The programming sequences for the five layers of switches are:

```
prog1 = {0 1 0 1}
prog2 = {1 1 0 0}
prog3 = {1 0 1 0}
prog4 = {- 0 - 1}
prog5 = {- 1 0 0}
```

To insert in five clock cycles the programming sequences into the permutation network on the inputs $\{p_1, p_2, p_3, p_4\}$ (see Figure 4.10) are successively applied the following 4-bit words: 0100, 0001, 1010, 1100, 0101. During the insertion the input `enable` *on each switch is activated.*

◇

4.5 First-Order Systolic Systems

When a very intense computational function is requested for an Application Specific Integrated Circuit (ASIC) systolic systems represent an appropriate solution. In a systolic system data are inserted and/or extracted rhythmically in/from a uniform modular structure. H. T. Kung and Charles E. Leiserson published the first paper describing a systolic system in 1978 [Kung '79] (however, the first machine known to use a a systolic approach was the Colossus Mark II in 1944). The following example of systolic system is taken from this paper.

Let us design the circuit which multiplies a band matrix with a vector as follows:

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 & \cdots \\ a_{21} & a_{22} & a_{23} & 0 & 0 & \cdots \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 & \cdots \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & \cdots \\ 0 & 0 & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & 0 & \ddots & \ddots & \ddots \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ \vdots \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ \vdots \end{pmatrix}$$

The main operation executed for matrix-vector operations is *multiply and accumulate* (MACC):

$$Z = A \times B + C$$

for which a specific combinational module is designed. Interleaving MACCs with memory circuits is provided a structure able to compute and to control the flow of data in the same time. The systolic vector-matrix multiplier is represented in Figure 4.11.

The systolic module is represented in Figure 4.11a, where a combinational multiplier ($M = A \times B$) is serially connected with an combinational adder ($M + C$). The result of MACC operation is latched in the output latch which latches besides the result of the computation, the two input value A and B. The latch is transparent on the high level of the clock. It is used to buffer intermediary results and to control the data propagation through the system.

The system is configured using pairs of modules to generate a master-slave structures, where one module receives `ck` and another `ck'`. The resulting structure is a non-transparent one ready to be used in a pipelined connection.

For a band matrix having the width 4, two non-transparent structures are used (see Figure 4.11c). Data is inserted in each phase of the clock (correlate data insertion with the phase of clock represented in Figure 4.11b) as follows:

The result of the computation is generated sequentially to the output y_i of the circuit from Figure 4.11c, as follows:

$$\begin{aligned} y_1 &= a_{11}x_1 + a_{12}x_2 \\ y_2 &= a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \\ y_3 &= a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 \\ y_4 &= a_{42}x_2 + a_{43}x_3 + a_{44}x_4 + a_{45}x_5 \\ y_5 &= \dots \\ &\dots \end{aligned}$$

The output X of the module is not used in this application (it is considered for matrix matrix multiplication only). The state of the system in each phase of the clock (see Figure 4.11b) is represented by two quadruples:

$$(Y_1, Y_2, Y_3, Y_4)$$

$$(Z_1, Z_2, Z_3, Z_4)$$

If the initial state of the system is unknown,
$(Y_1, Y_2, Y_3, Y_4) = (-, -, -, -)$
$(Z_1, Z_2, Z_3, Z_4) = (-, -, -, -)$
then the state of the system in the first 10 phases of the clock, numbered in Figure 4.11c, are the following:

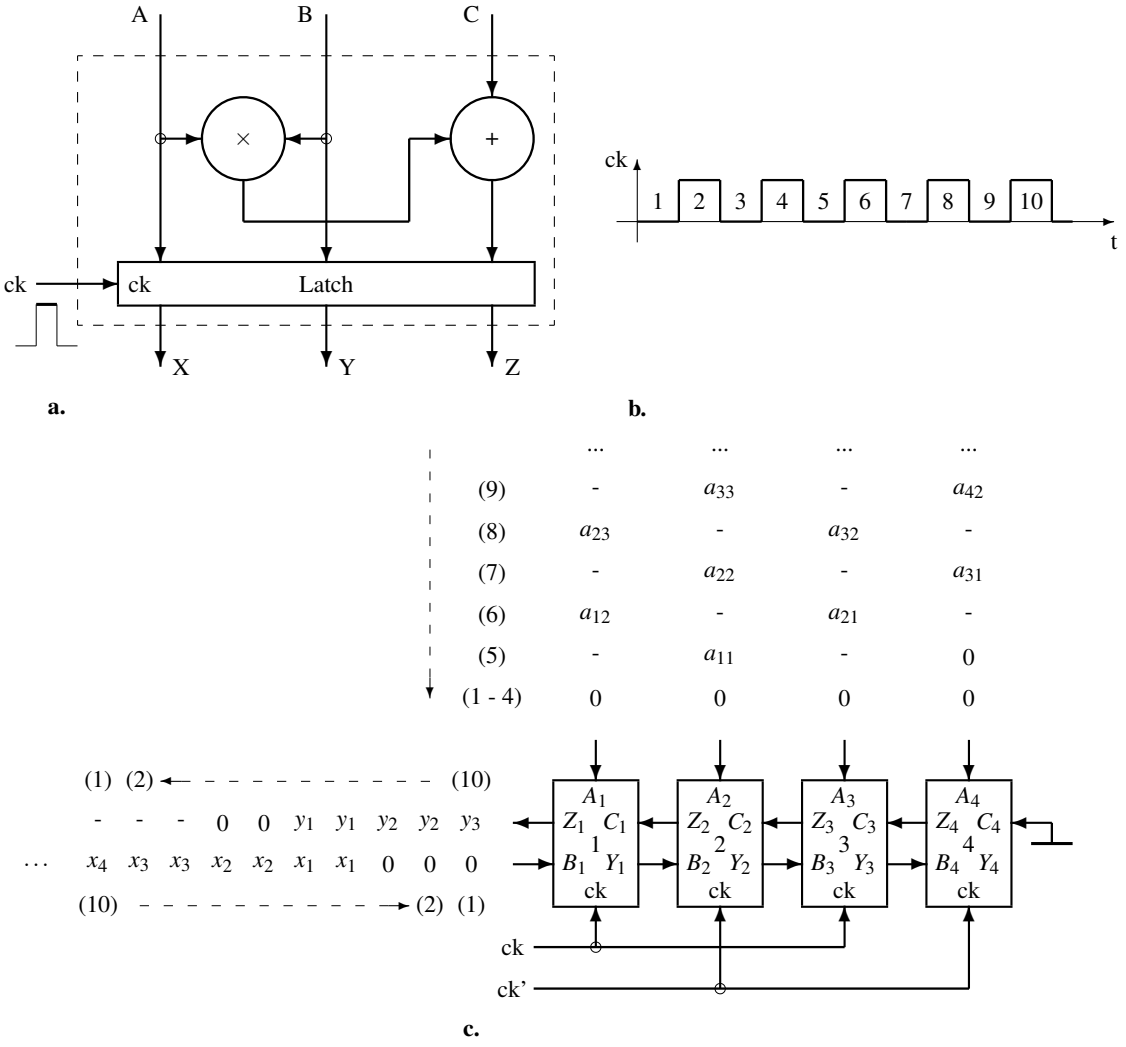


Figure 4.11: **Systolic vector-matrix multiplier.** **a.** The module. **b.** The clock signal with indexed half periods. **c.** How the modular structure is fed with the data in each half period of the clock signal.

Phase: (1)
$(Y_1,Y_2,Y_3,Y_4) = (-,-,-,-)$
$(Z_1,Z_2,Z_3,Z_4) = (-,-,-,0)$

Phase: (2)
$(Y_1,Y_2,Y_3,Y_4) = (0,-,-,-)$
$(Z_1,Z_2,Z_3,Z_4) = (-,-,0,0)$

Phase: (3)
$(Y_1,Y_2,Y_3,Y_4) = (0,0,-,-)$
$(Z_1,Z_2,Z_3,Z_4) = (-,0,0,0)$

Phase: (4)
$(Y_1,Y_2,Y_3,Y_4) = (x_1,0,0,-)$
$(Z_1,Z_2,Z_3,Z_4) = (0,0,0,0)$
(5)
$(Y_1,Y_2,Y_3,Y_4) = (x_1,x_1,0,0)$
$(Z_1,Z_2,Z_3,Z_4) = (0,a_{11}x_1,0,0)$
(6)
$(Y_1,Y_2,Y_3,Y_4) = (x_2,x_1,x_1,0)$
$(Z_1,Z_2,Z_3,Z_4) = (a_{11}x_1+a_{12}x_2,a_{11}x_1,a_{21}x_1,0)$
(7)
$(Y_1,Y_2,Y_3,Y_4) = (x_2,x_2,x_1,x_1)$
$(Z_1,Z_2,Z_3,Z_4) = (y_1,a_{21}x_1+a_{22}x_2,a_{21}x_1,a_{31}x_1)$
(8)
$(Y_1,Y_2,Y_3,Y_4) = (x_3,x_2,x_2,x_1)$
$(Z_1,Z_2,Z_3,Z_4) = (a_{21}x_1+a_{22}x_2+a_{23}x_3,a_{21}x_1+a_{22}x_2,a_{31}x_1+a_{32}x_2,a_{31}x_1)$
(9)
$(Y_1,Y_2,Y_3,Y_4) = (x_3,x_3,x_2,x_2)$
$(Z_1,Z_2,Z_3,Z_4) = (y_2,a_{31}x_1+a_{32}x_2+a_{33}x_3,a_{31}x_1+a_{32}x_2,a_{42}x_2)$
(10)
$(Y_1,Y_2,Y_3,Y_4) = (x_4,x_3,x_3,x_2)$
$(Z_1,Z_2,Z_3,Z_4) = (a_{31}x_1+a_{32}x_2+a_{33}x_3+a_{34}x_4,a_{31}x_1+a_{32}x_2+a_{33}x_3,a_{42}x_2+a_{43}x_3,a_{42}x_2)$
(11)
$(Y_1,Y_2,Y_3,Y_4) = (x_4,x_4,x_3,x_3)$
$(Z_1,Z_2,Z_3,Z_4) = (y_3,\ldots)$
...
In each clock cycle 4 multiplications and 4 additions are performed. The pipeline connections allow the synchronous insertion and extraction of data. The maximum width of the matrix band determines the number of modules used to design the systolic system.

4.6 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.

4.7 Problems

RAMs

Problem 4.1 *Explain the reason for t_{ASU} and for t_{AH} in terms of the combinational hazard.*

Problem 4.2 *Explain the reason for t_{DSU} and for t_{DH}.*

Problem 4.3 *Provide a structural description of the RAM circuit represented in Figure* **??** *for $m = 256$. Compute the size of the circuit emphasizing both the weight of storing circuits and the weight of the access circuits.*

Problem 4.4 *Design a 256-bit RAM using a two-dimensional array of 16×16 latches in order to balance the weight of the storing circuits with the weight of the accessing circuits.*

Problem 4.5 *Design the flow-through version of SRAM defined in Figure* **??**.
Hint*: use additional storage circuits for address and input data, and relate the WE' signal with the clock signal.*

Problem 4.6 *Design the register to latch version of SRAM defined in Figure 4.12.*
Hint*: the write process is identical with the flow-through version.*

Problem 4.7 *Design the pipeline version of SRAM defined in Figure 4.12.*
Hint*: only the output storage device must be adapted.*

Problem 4.8 *Provide a recursive description of an n-bit register. Prove that the (algorithmic) complexity of the concept of register is in $O(n)$ and the complexity of a ceratin register is in $O(\log n)$.*

Problem 4.9 *Draw the schematic for an 8-bit enabled and resetable register. Provide the Verilog environment for testing the resulting circuit.* Main restriction*: the clock signal must be applied only directly to each D flip-flop.*
Hint*: an enabled device performs its function only if the enable signal is active; to reset a register means to load it with the value 0.*

Problem 4.10 *Add to the register designed in the previous problem the following feature: the content of the register is shifted one binary position right (the content is divided by two neglecting the reminder) and on most significant bit (MSB) position is loaded the value of the one input bit called SI (serial input). The resulting circuit will be commanded with a 2-bit code having the following meanings:*

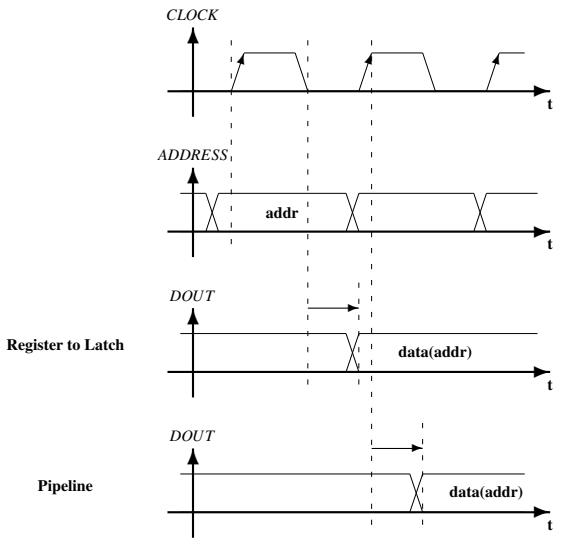


Figure 4.12: **Read cycles.** Read cycle for the *register to latch* version and for the *pipeline* version of *SRAM*.

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 4.11 *Design a serial-parallel register which shifts 16 16-bit numbers.*

Definition 4.1 *The serial-parallel register, $SPR_{n\times m}$, is made by a $SPR_{(n-1)\times m}$ serial connected with a R_m. The $SPR_{1\times m}$ is R_m.* $\diamond$

Hint*: the serial-parallel register, $SPR_{n\times m}$ can be seen in two manners. $SPR_{n\times m}$ consists in m parallel connected serial registers SR_n, or $SPR_{n\times m}$ consists in n serially connected registers R_m. We prefer usually the second approach. In Figure 4.13 is shown the serial-parallel $SPR_{n\times m}$.*

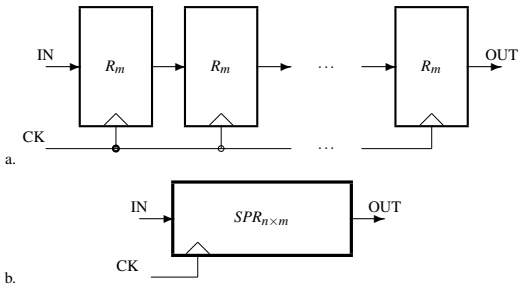


Figure 4.13: **The serial-parallel register. a.** The structure. **b.** The logic symbol.

Problem 4.12 *Let be t_{SU}, t_H, t_p, for a register and t_{pCLC} the propagation time associated with the CLC loop connected with the register. The maximal and minimal value of each is provided. Write the relations governing these time intervals which must be fulfilled for a proper functioning of the loop.*

Problem 4.13 *Explain what is wrong in the following* `always` *construct used to describe a pipelined system.*

```
module pipeline    #(parameter   n = 8, m = 16, p = 20)
                    (output    reg[m-1:]   output_reg,
                     input     wire[n-1:0] in,
                     clock);

   reg[n-1:0]      input_reg;
   reg[p-1:0]      pipeline_reg;
   wire[p-1:0]     out1;
   wire[m-1:0]     out2;
   clc1    first_clc(out1, input_reg);
   clc2    second_clc(out2, pipeline_reg);

   always @(posedge clock) begin   input_reg = in;
                                   pipeline_reg = out1;
                                   output_reg = out2;
                           end
endmodule
module  clc1(out1, in1);
   // ...
endmodule
module  clc2(out2, in2);
   // ...
endmodule
```

Hint*: revisit the explanation about blocking and nonblocking evaluation in Verilog.*

Problem 4.14 *Draw* `register_file_16_4` *at the level of registers, multiplexors and decoders.*

Problem 4.15 *Evaluate for* `register_file_32_5` *minimum input arrival time before clock (t_{in_reg}), minimum period of clock (T_{min}), maximum combinational path delay (t_{in_out}) and maximum output required time after clock (t_{reg_out}) using circuit timing from Appendix* Standard cell libraries.

Problem 4.16 *Design a CAM with binary codded output address, which provides as output address the first location containing the searched binary configuration, if any.*

Problem 4.17 *Design an associative memory, AM, implemented as a* maskable *and* readable *CAM. A CAM is maskable if any of the m input bits can be masked using an m-bit mask word. The masked bit is ignored during the comparison process. A CAM is readable if the* full *content of the first matched location in sent to the data output.*

Problem 4.18 *Find examples for the inequality*

$$number\_of\_possible\_names >>> number\_of\_actual\_names$$

which justify the use of the associative memory concept in digital systems.

4.8 Projects

Project 4.1 *Let be the module* `system` *containing* `system1` *and* `system2` *interconnected through the two-direction memory buffer module* `bufferMemory`. *The signal* `mode` *controls the sense of the transfer: for* `mode = 0` `system1` *is in read mode and* `system2` *in write mode, while for* `mode = 1` `system2` *is in read mode and* `system1` *in write mode. The module library provide the memory block described by the module* `memory`.

```
module system( input   [m-1:0] in1     ,
               input   [n-1:0] in2     ,
               output  [p-1:0] out1    ,
               output  [q-1:0] out2    ,
               input           clock   );
   wire    [63:0]  memOut1 ;
   wire    [63:0]  memIn1  ;
   wire    [13:0]] addr1   ;
   wire            we1     ;
   wire    [255:0] memOut2 ;
   wire    [255:0] memIn2  ;
   wire    [11:0]  addr2   ;
   wire            we2     ;
   wire            mode    ; // mode = 0: system1 reads, system2 writes
                             // mode = 1: system2 reads, system1 writes
   wire    [1:0]   com12, com21   ;
   system1 system1(in1, out1, com12, com21,
                   memOut1 ,
                   memIn1  ,
                   addr1   ,
                   we1     ,
                   mode    ,
                   clock   );
   system2 system2(in2, out2, com12, com21,
                   memOut2 ,
                   memIn2  ,
                   addr2   ,
                   we2     ,
                   clock   );
   bufferMemory    bufferMemory(   memOut1 ,
                                   memIn1  ,
                                   addr1   ,
                                   we1     ,
                                   memOut2 ,
                                   memIn2  ,
                                   addr2   ,
                                   we2     ,
                                   mode    ,
                                   clock   );
endmodule

module memory #(parameter n=32, m=10)
       (   output  reg [n-1:0] dataOut     ,   // data output
           input       [n-1:0] dataIn      ,   // data input
```

```
            input       [m-1:0] readAddr    ,   // read address
            input       [m-1:0] writeAddr   ,   // write address
            input               we          ,   // write enable
            input               enable      ,   // module enable
            input               clock       );

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

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

Design the module `bufferMemory`.

Project 4.2 *Design a systolic system for multiplying a band matrix of maximum width 16 with a vector. The operands are stored in serial registers.*

Chapter 5

AUTOMATA:
Second order, 2-loop digital systems

5.1 Two States Automata

The smallest two-state half-automata can be explored almost systematically. Indeed, there are only 16 one-input two-state half-automata and 256 with two inputs. We choose only two of them: the *T flip-flop*, the *JK flip-flop*, which are automata with $Q = Y$ and $f = g$. For simple 2-operand computations 2-input automata can be used. One of them is the *adder automaton*. This section ends with a small and simple *universal automaton* having 2 inputs and 2 states.

5.1.1 Serial Arithmetic

As we know the ripple carry adder has the size in $O(n)$ and the depth also in $O(n)$ (remember Figure **??**). If we agree with the time in this magnitude order, then there is a better solution where a second order circuit is used.

The best solution for the n-bit adder is a solution involving a small and simple automaton. Instead of storing the two numbers to be added in (parallel) registers, as in the pure combinational solution, the sequential solutions needs serial registers for storing the operands. The system is presented in Figure 5.1, containing three serial registers (two for the operands and one for the result) and the *adder automaton*.

The adder automaton is a two states automaton having in the loop the carry circuit of a full adder (FA). The one-bit state register contains the carry bit from the previous cycle. The inputs A and B of FA receive synchronously, at each clock cycle, bits having the same binary range from the serial registers. First, LSBs are read from the serial registers. Initially, the automaton is in the state 0, that means $CR = 0$. The output S is stored bit by bit in the third serial register during n clock cycles. The final $(n+1)$-bit result is contained in the output serial register and in the state register.

The operation time remains in the order of $O(n)$, but the structure involved in computation becomes the constant structure of the adder automaton. The product of the size, $S_{ADD}(n)$, into the time, $T_{ADD}(n)$ is in $O(n)$ for this sequential solution. Again, Conjecture 2.1 acts emphasizing the slowest solution as optimal. Let us remember that for a carry-look-ahead adder, the fastest $O(1)$ variant, the same product was in $O(n^3)$. The price for the constant execution time is, in this example, in $O(n^2)$. I believe it is too much. We will prefer *architectural* solutions which allow us to avoid the *structural* necessity to perform the addition in constant time.

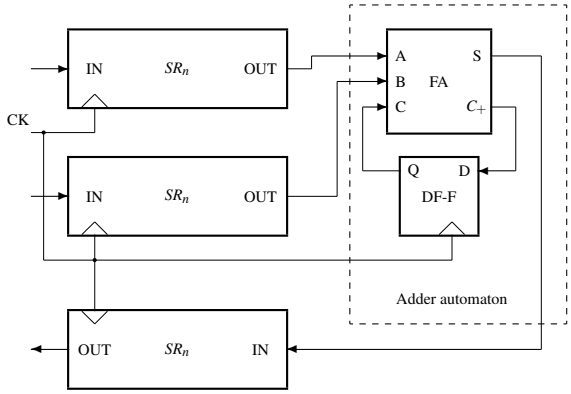


Figure 5.1: **Serial *n*-bit adder.** The state of the adder automaton has the value of the carry generated adding the previous 2 bits received from the output of the two serial registers containing the operands.

5.1.2 Hillis Cell: the Universal 2-Input, 1-Output and 2-State Automaton

Any binary (two-operand) simple operation on *n*-bit operands can be performed serially using a 2-state automaton. The internal state of the automaton stores the "carry" information from one stage of processing to another. In the *adder automaton*, just presented, the internal state is used to store the *carry* bit generated adding the *i*-th bits of a number. It is used in the next stage for adding the $(i+1)$-th bits. This mechanism can be generalized, resulting an universal 2-input (for binary operation), one-output and 2-state (for "carry" bit) automaton.

Definition 5.1 *An Universal 2-input (in*1*, in*2*),one-output, 2-state (codded by state*$[0]$*) automaton is a programmable structure using a 16-bit program word,* $\{next\_state\_func[7:0], out\_func[7:0]\}$*. It is defined by the following Verilog code:*

```
module univAut( output      out                      , // output
                input       in1, in2                 , // operands
                input [7:0] nextStateFunc, outFunc   , // loop program, output program
                input       reset, clock             );
    reg state;
    assign out = outFunc[{state, in2, in1}];
    always @(posedge clock) state <= reset ? 1'b0 : nextStateFunc[{state, in2, in1}];
endmodule
```

◇

The universal programmable automaton is implemented using two 3-input universal combinational circuits (8 to 1 multiplexers), one for the output function and another for the loop function (Figure 5.2. The total number of automata can be programmed on this structure is 2^{16} (the total number of 16-bit "programs"). Most of them are meaningless, but the simplicity of solution deserves our attention. Let us call this universal automaton *Hillis Cell* because, as far as I know, this small and simple circuit was first used by Daniel Hillis as execution unit in *Connection Machine* parallel computer he designed in 1980 years [Hillis '85].

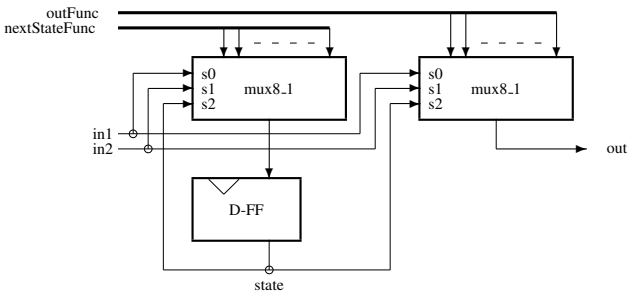


Figure 5.2: ***Hillis cell.***

5.2 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.

5.2.1 Accumulator Automaton

The *accumulator automaton* is a generalization of the counter automaton. A counter can add 1 to the value of its state in each clock cycle. An accumulator automaton can add in each clock cycle any value applied on its inputs.

Many applications require the accumulator function performed by a system which adds a string of numbers returning the final sum and all partial results – the *prefixes*. Let be p numbers $x_1,\ldots,x_p$. The *sum*-prefixes are:

$y_1 = x_1$
$y_2 = x_1 + x_2$
$y_3 = x_1 + x_2 + x_3$
...
$y_p = x_1 + x_2 + \ldots + x_p$.

This example of arithmetic automata generates at each clock cycle one prefix starting with y_1. The initial value in the register R_{m+n} is zero. The structure is presented in Figure 5.3 and consists in an adder, ADD_{m+n}, two multiplexors and a state register, R_{m+n}. This automaton has 2^{m+n} states and computes sum prefixes for $p = 2^m$ numbers, each represented with n bits. The supplementary m bits are needed because in the worst case adding two numbers of n bits results a number of $n+1$ bits, and so on, ... adding 2^m n-bit numbers results, in the worst case, a $n+m$-bit number. The automaton must be dimensioned such as in the worst case the resulting prefix can be stored in the state register. The two multiplexors are used to initialize the system clearing the register (for `acc = 0` and `clear = 1`), to maintain unchanged the accumulated value (for `acc = 0` and `clear = 0`), or to accumulate the n-bit input value (for `acc = 1` and `clear = 0`). It is obvious the accumulate function has priority: for `acc = 1` and `clear = 1` the automaton accumulates ignoring the clear command.

The size of the systems depends on the speed of adder and can be found between $O(m+n)$ (for ripple

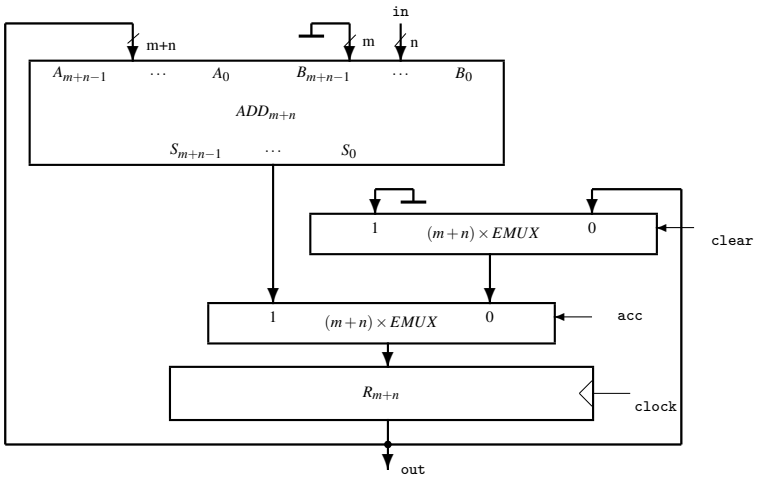


Figure 5.3: **Accumulator automaton.** It can be used as *sum prefix automaton* because in each clock cycle outputs a new value as a result of a sequential addition of a stream of signed integers.

carry adder) and $O((m+n)^3)$ (for carry-look-ahead adder).

It is evident that this automaton is a simple one, having a constant sized definition. The four components are all simple recursive defined circuits. This automaton can be build for any number of states using the same definition. In this respect this automaton is a "non-finite", functional automaton.

5.2.2 Sequential multiplication

Multiplication is performed sequentially by repeated additions and shifts. The n-bit multiplier is inspected and the multiplicand is accumulated shifted according to the position of the inspected bit or bits. If in each cycle one bit is inspected (radix-2 multiplication), then the multiplication is is performed in n cycles. If 2 bits are inspected (radix-2 multiplication) in each cycle, then the operation is performed in $n/2$ cycles, and so on.

Radix-2 multiplication

The generic three-register structure for radix-2 multiplication is presented in the following Verilog module.

```
module rad2mult #(parameter n = 8)(output  [2*n-1:0]   product , // result output
                                   input   [n-1:0]     op      , // operands input
                                   input   [2:0]       com     , // command input
                                   input               clock   );

   reg     [n-1:0] op2, op1, prod  ; // multiplicand, multiplier, upper result
   wire    [n:0]   sum ;

   assign  sum = prod + op2;

   always @(posedge clock) if (com[2])
       case(com[1:0])
```

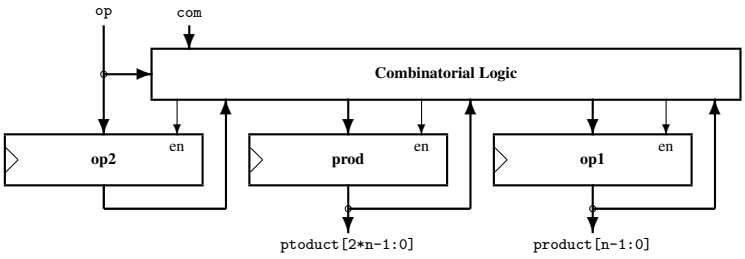


Figure 5.4: **Radix-2 sequential multiplier.**

```
          2'b00:  prod    <= 0    ;                                    // clear prod
          2'b01:  op1     <= op   ;                                    // load multiplier
          2'b10:  op2     <= op   ;                                    // load multiplicand
          2'b11:  {prod, op1} <= (op1[0] == 1) ?  {sum, op1[n-1:1]} :
                                                  {prod, op1} >> 1  ; // multiplication step
       endcase

   assign product = {prod, op1};
endmodule
```

The sequence of commands applied to the previous module is:

```
initial begin       com = 3'b100        ;
                #2  com = 3'b101        ;
                    op  = 8'b0000_1001  ;
                #2  com = 3'b110        ;
                    op  = 8'b0000_1100  ;
                #2  com = 3'b111        ;
                #2  com = 3'b111        ;
                #2  com = 3'b111        ;
                #2  com = 3'b111        ;
                #2  com = 3'b111        ;
                #2  com = 3'b111        ;
                #2  com = 3'b111        ;
                #2  com = 3'b111        ;
                #2  com = 3'b000        ;
                #2  $stop               ;
        end
```

In real application the first three steps can be merged in one, depending on the way the multiplier is connected. The effective number of clock cycles for multiplication is n.

Radix-4 multiplication

The time performance for the sequential multiplication is improved if in each clock cycle 2 bits are considered instead of one. In order to keep simple the operation performed in each cycle a small and simple two-state automaton is included in design.

Let us consider positive integer multiplication. The design inspects by turn each 2-bit group of the multiplier, `m[i+1:i]` for `i = 0, 2, ... n-2`, doing the following simple actions:

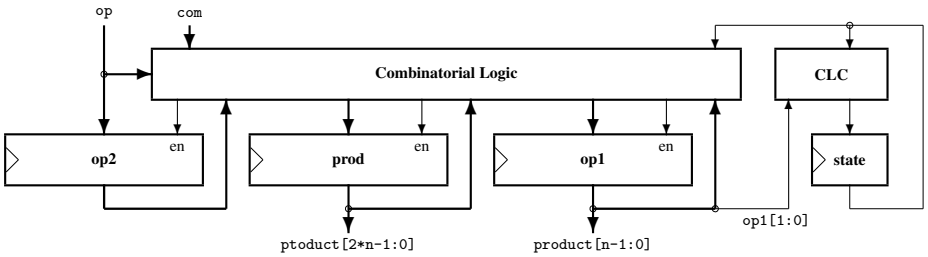


Figure 5.5: **Radix-4 sequential multiplier.**

`m[i+1:i] = 00` : adds 0 to the result and multiply by 4 the multiplicand

`m[i+1:i] = 01` : adds the multiplicand to the result and multiply by 4 the multiplicand

`m[i+1:i] = 10` : adds twice the multiplicand to the result and multiply by 4 the multiplicand

`m[i+1:i] = 11` : subtract the multiplicand from the results, multiply by 4 the multiplicand, and ***sends to the next cycle the information that the current value of multiplicand must be added 4 times to the result***

The inter-cycle message is stored in the state of the automaton. In the initial cycle the state of the automaton is ignored, while in the next stages it is added to the value of `m[i+1:i]`.

```
module rad4mult #(parameter n = 8)(output  [2*n-1:0]   product , // result output
                                   input   [n-1:0]     op      , // operands input
                                   input   [2:0]       com     , // command input
                                   input               clock   );

   reg     [n-1:0]     op2, op1    ; // multiplicand, mutiplier, upper part of result
   reg     [n+1:0]     prod        ; // upper part of result
   reg                 state       ; // the state register of a two-state automaton

   reg     [n+1:0]     nextProd    ;
   reg                 nextState   ;

   wire    [2*n+1:0]   next        ;
   /*
   com = 3'b000 // nop
   com = 3'b001 // clear prod register & initialize automaton
   com = 3'b010 // load op1
   com = 3'b011 // load op2
   com = 3'b101 // mult
   com = 3'b110 // lastStep
   */
   always @(posedge clock)
       case(com)
           3'b001: begin   prod        <= 0            ;
                           state       <= 0            ;
                   end
           3'b010: op1                 <= op           ;
```

```
            3'b011: op2                 <= op           ;
            3'b101: begin   {prod, op1} <= next         ;
                            state       <= nextState    ;
                    end
            3'b110: if (state) prod     <= prod + op2   ;
            default prod                <= prod         ;
        endcase

    assign next = {{2{nextProd[n+1]}}, nextProd, op1[n-1:2]};
    // begin algorithm
    always @(*) if (state)  case(op1[1:0])
                                2'b00:  begin   nextProd    = prod + op2        ;
                                                nextState   = 0                 ;
                                        end
                                2'b01:  begin   nextProd    = prod + (op2 << 1) ;
                                                nextState   = 0                 ;
                                        end
                                2'b10:  begin   nextProd    = prod - op2        ;
                                                nextState   = 1                 ;
                                        end
                                2'b11:  begin   nextProd    = prod              ;
                                                nextState   = 1                 ;
                                        end
                            endcase
                    else    case(op1[1:0])
                                2'b00:  begin   nextProd    = prod              ;
                                                nextState   = 0                 ;
                                        end
                                2'b01:  begin   nextProd    = prod + op2        ;
                                                nextState   = 0                 ;
                                        end
                                2'b10:  begin   nextProd    = prod + (op2 << 1) ;
                                                nextState   = 0                 ;
                                        end
                                2'b11:  begin   nextProd    = prod - op2        ;
                                                nextState   = 1                 ;
                                        end
                            endcase
    // end algorithm
    assign product = {prod[n-1:0], op1};
endmodule
```

The sequence of commands for $n = 8$ is:

```
    initial begin       com = 3'b001        ;
                    #2  com = 3'b010        ;
                        op  = 8'b1100_1111  ;
                    #2  com = 3'b011        ;
                        op  = 8'b1111_1111  ;
                    #2  com = 3'b101        ;
                    #2  com = 3'b101        ;
                    #2  com = 3'b101        ;
                    #2  com = 3'b101        ;
                    #2  com = 3'b110        ;
                    #2  com = 3'b000        ;
                    #2  $stop               ;
            end
```

The effective number of clock cycles for positive integer radix-4 multiplication is $n/2+1$.

Let's now solve the problem multiplying signed integers. The Verilog description of the circuit is:

```
module signedRad4mult #(parameter n = 8)(  output  [2*n-1:0]   product , // result output
                                           input   [n-1:0]     op      , // operands input
                                           input   [2:0]       com     , // command input
                                           input               clock   );

    reg     [n-1:0] op1         ; // signed mutiplier
    reg     [n:0]   op2         ; // signed multiplicand
    reg     [n:0]   prod        ; // upper part of the signed result
    reg             state       ; // the state register of a two-state automaton

    reg     [n:0]   nextProd    ;
    reg             nextState   ;

    wire    [2*n:0] next        ;
    /*
    com = 3'b000 // nop
    com = 3'b001 // clear prod register & initialize automaton
    com = 3'b010 // load op1
    com = 3'b011 // load op2 with one bit sign expansion
    com = 3'b100 // mult
    */
    always @(posedge clock)
        case(com)
            3'b001: begin   prod        <= 0            ;
                            state       <= 0            ;
                    end
            3'b010: op1                 <= op           ;
            3'b011: op2                 <= {op[n-1], op}; // to allow left shift
            3'b100: begin   {prod, op1} <= next         ;
                            state       <= nextState    ;
                    end
            default prod                <= prod         ;
        endcase

    assign next = {{2{nextProd[n]}}, nextProd, op1[n-1:2]};
    // begin algorithm
    always @(*) if (state)  case(op1[1:0])
                                2'b00:  begin   nextProd    = prod + op2        ;
                                                nextState   = 0                 ;
                                        end
                                2'b01:  begin   nextProd    = prod + (op2 << 1) ;
                                                nextState   = 0                 ;
                                        end
                                2'b10:  begin   nextProd    = prod - op2        ;
                                                nextState   = 1                 ;
                                        end
                                2'b11:  begin   nextProd    = prod              ;
                                                nextState   = 1                 ;
                                        end
                            endcase
                    else    case(op1[1:0])
                                2'b00:  begin   nextProd    = prod              ;
                                                nextState   = 0                 ;
                                        end
```

```
                        2'b01:  begin   nextProd    = prod + op2        ;
                                        nextState   = 0                 ;
                                end
                        2'b10:  begin   nextProd    = prod - (op2 << 1) ;
                                        nextState   = 1                 ;
                                end
                        2'b11:  begin   nextProd    = prod - op2        ;
                                        nextState   = 1                 ;
                                end
                    endcase
   // end algorithm
   assign product = {prod[n-1:0], op1};
endmodule
```

The sequence of commands for $n = 8$ is:

```
initial begin       com = 3'b001        ;
                #2  com = 3'b010        ;
                    op  = 8'b1111_0001  ;
                #2  com = 3'b011        ;
                    op  = 8'b1111_0001  ;
                #2  com = 3'b100        ;
                #2  com = 3'b100        ;
                #2  com = 3'b100        ;
                #2  com = 3'b100        ;
                #2  com = 3'b000        ;
                #2  $stop               ;
        end
```

The effective number of clock cycles for signed integer radix-4 multiplication is $n/2$.

Multiplying with the "Bit-eater" automaton

A very useful function is to search the bits of a binary word in order to find the positions occupied by the 1s. For example, inspecting the number 00100100 we find in 2 steps a 1 on the 5-th position and another on the 2-nd position. A simple automaton does this operation in a number of clock cycles equal with the number of 1s contained in its initial state. In Figure 5.6 is represented The "bit-eater" automaton which is a simple machine containing:

- an n-bit state register
- a multiplexer used to initialize the automaton with the number to be inspected, if `load = 1`, then the register takes the input value, else the automaton's loop is closed
- a priority encoder circuit which computes the index of the most significant bit of the state and activates the demultiplexer ($E' = 0$), if its enable input is activated, (`eat = 1`)
- an enabled decoder (or a demultiplexor) which decodes the value generated by the priority encoder applying 1 only to the input of one XOR circuit if $zero = 0$ indicating at least one bit of the state word is 1
- n 2-input XORs used to complement the most significant 1 of the state.

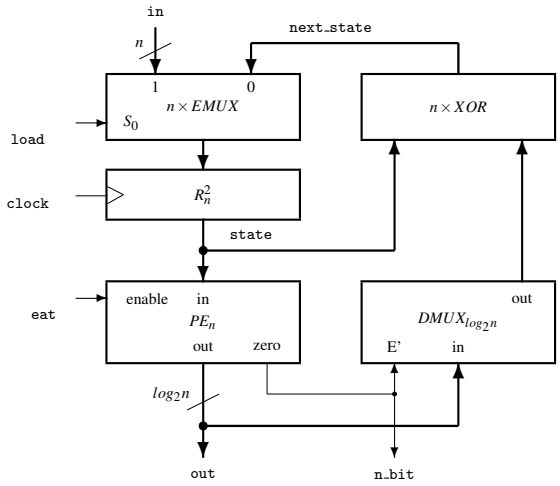


Figure 5.6: **"Bit-eater" automaton.** Priority encoder outputs the index of the most significant 1 in register, and the loop switches it into 0 using the demultiplexer to "point" it and a XOR to invert it.

In each clock cycle after the initialization cycle the output takes the value of the index of the most significant bit of state having the value 1, and the next state is computed clearing the pointed bit.

Another, numerical interpretation is: while $state \neq 0$, the output of the automaton takes the integer value of the base 2 logarithm of the state value, $|log_2(state)|$, and the next state will be $next\_state = state - |log_2(state)|$. If $state = 0$, then $n\_bit = 1$, the output takes the value 0, and the state remains in 0.

5.2.3 Sequential divider

The sequential divisor circuit receives two n-bit positive integers, the dividend and the divisor, and returns other n-bit positive integers: the quotient and the remainder. The sequential algorithm to compute:

$$dividend/divisor = quotient + remainder$$

a sort of "trial & error" algorithm which computes the n bits of the quotient starting with `quotient[n-1]`. Then in the first step `remainder = dividend - (divisor << (n-1))` is computed and if the result is a positive number the most significant bit of the quotient is 1 and the operation is validated as the new state of the circuit, else the most significant bit of the quotient is 0. Next step we try with `divisor << (n-2)` computing `quotient[n-2]`, and so on until the `quotient[o]` bit is determined. The structure of the circuit is represented in figure 5.7.

The Verilod description of the circuit is:

```
module divisor #(parameter n = 8)( output  reg [n-1:0] quotient    ,
                                   output  reg [n-1:0] remainder   ,
                                   output              error       ,
                                   input       [n-1:0] dividend    ,
                                   input       [n-1:0] divisor     ,
                                   input       [1:0]   com         ,
                                   input               clk         );
   parameter   nop = 2'b00,
```

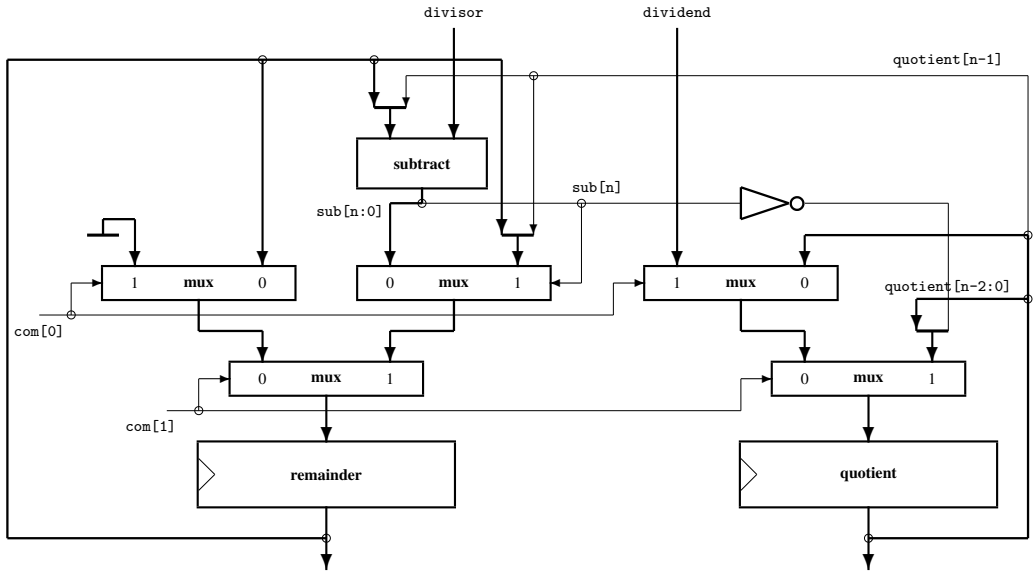


Figure 5.7: **Sequential divisor.**

```
            ld  = 2'b01,
            div = 2'b10;

    wire    [n:0]   sub;

    assign error    = (divisor == 0) & (com == div)            ;
    assign sub = {remainder, quotient[n-1]} - {1'b0, divisor}  ;

    always @(posedge clk)
        if (com == ld)          begin   quotient    <= dividend ;
                                        remainder   <= 0        ;
                                end
         else if (com == div)   begin   quotient    <= {quotient[n-2:0], ~sub[n]}          ;
                                        remainder   <= sub[n] ?
                                                        {remainder[n-2:0], quotient[n-1]} :
                                                        sub[n-1:0]                         ;
                                end
endmodule
```

5.3 Composing with simple automata

Using previously defined simple automata some very useful subsystem can be designed. In this section are presented some subsystems currently used to provide solutions for real applications: Last-In First-Out memory (LIFO), First-In-First-Out memory (FIFO), and a version of the multiply accumulate circuit (MACC). All are simple circuits because result as simple compositions of simple circuits, and all are expandable for any $n \ldots m$, where $n \ldots m$ are a parameters defining different part of the circuit. For example, the memory size and the word size are independent parameters is a FIFO implementation.

5.3.1 LIFO memory

The LIFO memory or the *stack memory* has many applications in structuring the processing systems. It is used both for building the control part of the system, or for designing the data section of a processing system.

Definition 5.2 *LIFO memory implements a data structure which consists of a string* `S = <s0, s1, s2, ...>` *of maximum* 2^m *n-bit recordings accessed for write, called* **push***, and read, called* **pop***, at the same end,* `s0`*, called* **top of stack** *(*`TOS`*).* ◇

Example 5.1 *Let be the stack* `S = <s0, s1, s2, ...>`*. It evolve as follows under the sequence of five commands:*

```
push a --> S = <a, s0, s1, s2, ...>
push b --> S = <b, a, s0, s1, s2, ...>
pop    --> S = <a, s0, s1, s2, ...>
pop    --> S = <s0, s1, s2, ...>
pop    --> S = <s1, s2, ...>
```

◇

Real applications request additional functions for a LIFO used for expression evaluation. An minimally expanded set of functions for the LIFO `S = <s0, s1, s2, ...>` contains the following operations:

- `nop`: no operation; the contents of `S` in untouched
- `write a`: write `a` in `TOS`
  `<s0, s1, s2, ...> --> <a, s1, s2, ...>`
  used for unary operations; for example:
  `a = s0 + 1`
  the TOS is incremented and write back in TOS (`pop, push = write`)
- `pop`:
  `<s0, s1, s2, ...> --> <s1, s2, ...>`
- `popwr a`: pop & write
  `<s0, s1, s2, ...> --> <a, s2, ...>`
  used for binary operations; for example:
  `a = s0 + s1`
  the first two positions in LIFO are popped, added and the result is pushed back into the LIFO memory (`pop, pop, push = pop, write`)
- `push a`:
  `<s0, s1, s2, ...> --> <a, s0, s1, s2, ...>`

A possible implementation of such a LIFO is presented in Figure 8.1, where:

- **Register File** is organized, using the logic surrounding it, as a 2^m n-bit stream of words accessed at TOS

- **LeftAddrReg** is a *m*-bit register containing the pointer to `TOS = s0`
- **Dec** is the decrement circuit pointing to `s1`
- **IncDec** is the circuit which increment, decrement or do not touch the content of the register **LeftAddrReg** as follows:
  - increment for `push`
  - decrement for `pop` or `popwr`
  - keeps unchanged for `nop` or `write`

  Its output is used to select the destination in **Register File** and to up date, in each clock cycle, the content of the register **LeftAddrReg**.

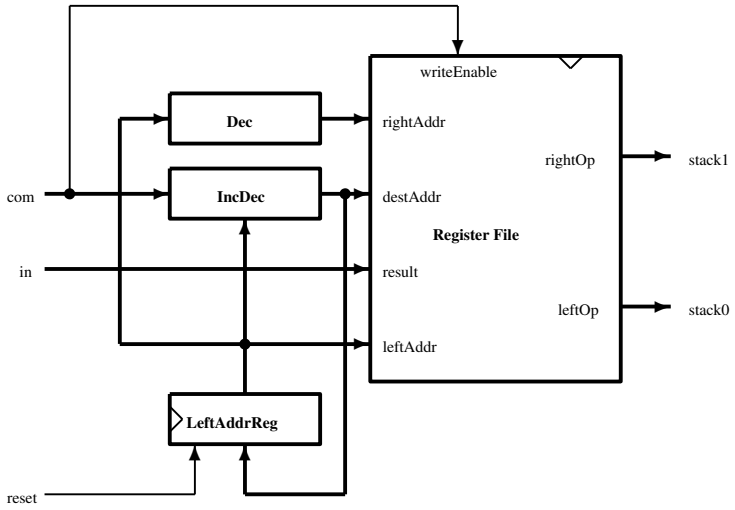


Figure 5.8: **LIFO memory.** The **LeftAddrReg** register is, in conjunction with **IncDec** commbinational circuit, an up/downn counter used as *stack pointer* to organize in **Register File** an expression evaluation stack.

A Verilog description of the previously defined LIFO (stack) is:

```
module lifo #(‘include "0_parameters.v")
        (   output [m-1:0] stack0, stack1,
            input  [m-1:0] in           ,
            input  [2:0]   com          ,
            input          reset , clock );
/*  The command codes
    nop   = 3’b000, //
    write = 3’b001, // we
    pop   = 3’b010, // dec
    popwr = 3’b011, // dec, we
    push  = 3’b101; // inc, we  */

    reg     [n-1:0] leftAddr;   // the main pointer
    wire    [n-1:0] nextAddr;
    // The increment/decrement circuit
    assign nextAddr = com[2] ? (leftAddr + 1’b1) : (com[1] ? (leftAddr - 1’b1) : leftAddr);
```

```
    // The address register for TOS
    always @(posedge clock) if (reset)  leftAddr <= 0       ;
                               else     leftAddr <= nextAddr;
    // The register file
    reg [m-1:0]  file[0:(1'b1 << n)-1];
        assign stack0 = file[leftAddr]       ,
               stack1 = file[leftAddr - 1'b1];
    always @(posedge clock) if (com[0]) file[nextAddr] <= in;
endmodule
```

Faster implementations can be done using registers instead of different kind of RAMs and counters (see the chapter *SELF-ORGANIZING STRUCTURES: N-th order digital systems*). For big stacks, optimized solutions are obtained combining a small register implemented stack with a big RAM based implementation.

5.3.2 FIFO memory

The FIFO memory, or the *queue memory* is used to interconnect subsystems working logical, or both logical and electrical, asynchronously.x

Definition 5.3 *FIFO memory implements a data structure which consists in a string of maximum* 2^m *n-bit recordings accessed for write and read, at its two ends.* **Full** *and* **empty** *signals are provided indicating the write operation or the read operation are not allowed.* $\diamond$

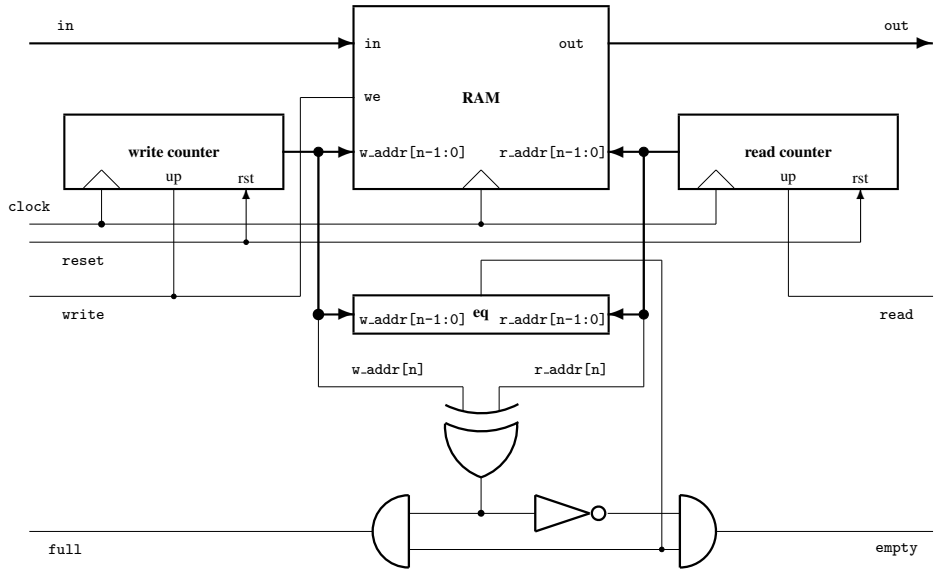


Figure 5.9: **FIFO memory.** Two pointers, evolving in the same direction, and a two-port RAM implement a LIFO (queue) memory. The limit flags are computed combinational from the addresses used to write and to read the memory.

A FIFO is considered synchronous if both *read* and *write* signals are synchronized with the same clock signal. If the two commands, *read* and *write*, are synchronized with different clock signals, then the FIFO memory is called asynchronous.

In Figure 5.9 is presented a solution for the synchronous version, where:

- **RAM** is a 2^n m-bit words two-port asynchronous random access memory, one port for write to the address `w_addr` and another for read form the address `r_addr`
- **write counter** is an $(n+1)$-bit resetable counter incremented each time a write is executed; its output is `w_addr[n:0]`, initially it is reset
- **read counter** is an $(n+1)$-bit resetable counter incremented each time a read is executed; its output is `r_addr[n:0]`, initially it is reset
- **eq** is a comparator activating its output when the least significant n bits of the two counters are identical.

FIFO works like a circular memory addressed by two pointers (`w_addr[n-1:0]` and `r_addr[n-1:0]`) running on the same direction. If the write pointer *after* a write operation becomes equal with the read pointer, then the memory is full and the `full` signal is 1. If the read pointer *after* a read operation becomes equal with the write pointer, then the memory is empty and the `empty` signal is 1. The $n+1$-th bit in each counter is used to differentiate between empty and full when `w_addr[n-1:0]` and `r_addr[n-1:0]` are the same. If `w_addr[n]` and `r_addr[n]` are different, then `w_addr[n-1:0] = r_addr[n-1:0]` means full, else it means empty.

The circuit used to compare the two addresses is a combinational one. Therefore, its output has a hazardous behavior which affects the outputs `full` and `empty`. These two outputs must be used carefully in designing the system which includes this FIFO memory. The problem can be managed because the system works in the same clock domain (`clock` is the same for both ends of FIFO and for the entire system). We call this kind of FIFO *synchronous FIFO*.

VeriSim 5.1 *A Verilog synthesisable description of a synchronous FIFO follows:*

```
module simple_fifo(output  [31:0]  out     ,
                   output          empty   ,
                   output          full    ,
                   input   [31:0]  in      ,
                   input           write   ,
                   input           read    ,
                   input           reset   ,
                   input           clock   );
   wire    [9:0]   write_addr, read_addr;

   counter write_counter(  .out        (write_addr ),
                           .reset      (reset      ),
                           .count_up   (write      ),
                           .clock      (clock      )),
           read_counter(   .out        (read_addr  ),
                           .reset      (reset      ),
                           .count_up   (read       ),
                           .clock      (clock      ));

   dual_ram memory(.out        (out            ),
                   .in         (in             ),
                   .read_addr  (read_addr[8:0] ),
                   .write_addr (write_addr[8:0]),
                   .we         (write          ),
```

```
                    .clock      (clock          ));

   assign  eq       = read_addr[8:0] == write_addr[8:0] ,
           phase    = ~(read_addr[9] == write_addr[9])  ,
           empty    = eq & phase   ,
           full     = eq & ~phase    ;
endmodule

module counter(output  reg [9:0]   out     ,
               input               reset   ,
               input               count_up,
               input               clock   );

   always @(posedge clock) if (reset)          out <= 0;
                           else if (count_up) out <= out + 1;
endmodule

module dual_ram(   output  [31:0]  out         ,
                   input       [31:0]  in          ,
                   input       [8:0]   read_addr   ,
                   input       [8:0]   write_addr  ,
                   input               we          ,
                   input               clock       );
   reg     [63:0]  mem[511:0];
   assign   out = mem[read_addr]   ;
   always @(posedge clock) if (we) mem[write_addr] <= in   ;
endmodule
```

◇

An *asynchronous FIFO* uses two independent clocks, one for `write counter` and another for `read counter`. This type of FIFO is used to interconnect subsystems working in different clock domains. The previously described circuit is unable to work as an asynchronous FIFO. The signals `empty` and `full` are meaningless, being generated in two clock domains. Indeed, `write counter` and `read counter` are triggered by different clocks generating the signal `eq` with hazardous transitions related to two different clocks: *write clock* and *read clock*. This signal can not be used neither in the system working with *write clock* nor in the system working with *read clock*. *Read clock* is unable to avoid the hazard generated by *write clock*, and *write clock* is unable to avoid the hazard generated by *read clock*. Special tricks must be used.

5.3.3 The Multiply-Accumulate Circuit

The functional automata can be composed in order to perform useful functions in a digital system. Otherwise, we can say that a function can be decomposed in many functional units, some of them being functional automata, in order to implement it efficiently. Let's take the example of the Multiply-Accumulate Circuit (MACC) and implement it in few versions. It is mainly used to implement one of the most important numerical functions performed in our digital machines, the scalar product of two vectors: $a_1 \times b_1 + \ldots + a_n \times b_n$.

We will offer in the following a solution involving two serially connected functional automata: an *accumulator automaton* and *"bits eater" automaton*.

The starting idea is that the multiplication is also an accumulation. Thus we use an accumulator automaton for implementing both operations, the multiplication and the sum of products, without any

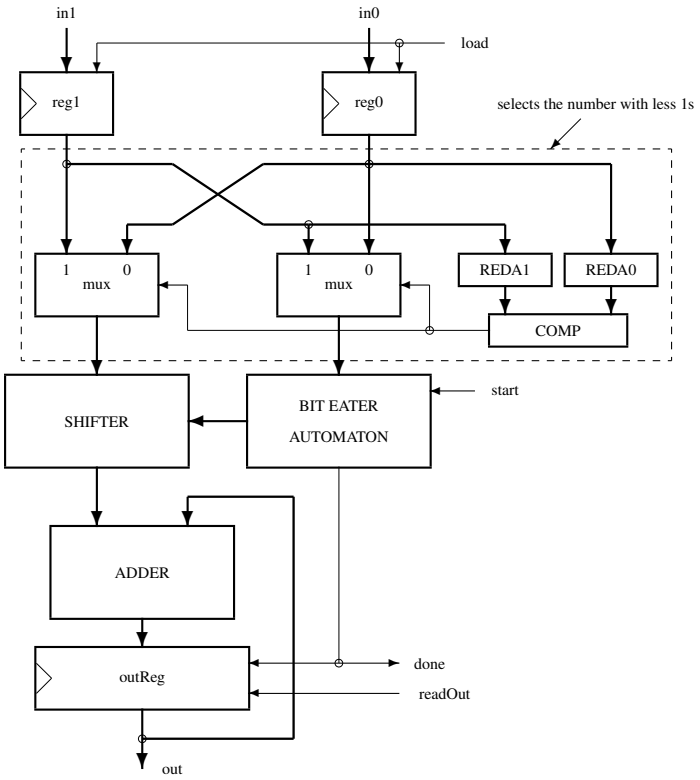


Figure 5.10:

loss in the execution time.

The structure of the multiply-accumulate circuit is presented in Figure 5.11 and consists in:

"bits eater" automaton – used to indicates successively the positions of the bits having the value 1 from the first operand a_i; it also points out the end of the multiplication (see Figure 5.6)

combinational shifter – shifts the second operand, b_i, with a number of positions indicated by the previous automaton

accumulate automaton – performs the partial sums for each multiplication step and accumulate the sum of products, if it is not cleared after each multiplication (see Figure 5.3).

In order to execute a multiplication only we must execute the following steps:

- load the "beat-eater" automaton with the first operand and clear the content of the output register in accumulator automaton
- select to the input the second operand which remains applied to the input of the shifter circuit during the operation
- wait for the end of operation indicated by the `done` output.

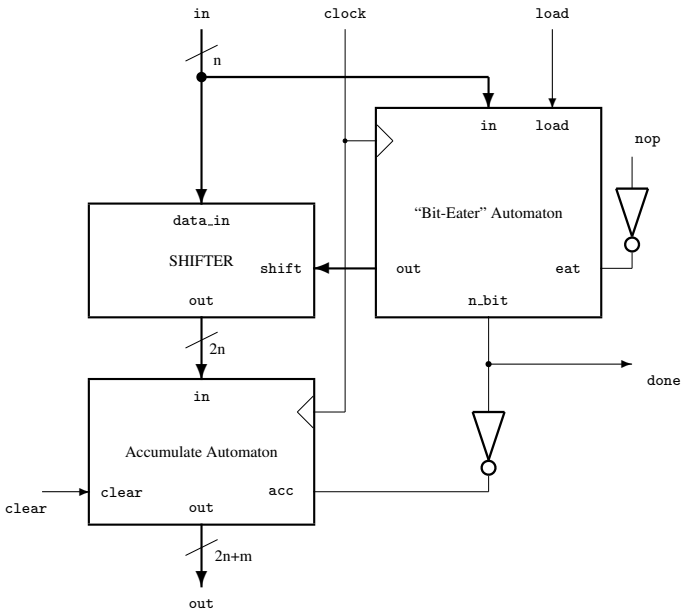


Figure 5.11: **A multiply-accumulate circuit (*MAC*).** A sequential version of MAC results serially connecting an automaton, generating by turn the indexes of the binary ranges equal with 1 in multiplier, with a combinational shifter and an accumulator.

The operation is performed in a number of clock cycles equal with the number of 1s of the first operand. Thus, the mean execution time is proportional with $n/2$. To understand better how this machine works the next example will be an automaton which controls it.

If a MACC function is performed the clear of the state register of the accumulator automaton is avoided after each multiplication. Thus, the register accumulates the results of the successive multiplications.

5.4 Control Automata: the First "Turning Point"

A very important class of finite automata is the class of *control automata*. A control automaton is embedded in a system using three main connections (see Figure 5.12):

- 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]`.

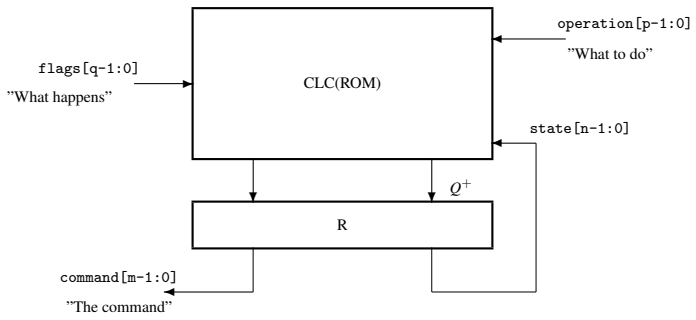


Figure 5.12: **Control Automaton.** The functional definition of control automaton. Control means to issue commands and to receive back signals (flags) characterizing the effect of the command.

The size and the complexity of the control sequence asks the replacement of the PLA with a ROM, at least for the designing and testing stages in implementing the application. The size of the ROM has the magnitude order:

$$S_{ROM}(n,p,q) \in O(2^{n+p+q}).$$

In order to reduce the ROM's size we start from the actual applications which emphasize two very important facts:

1. the automaton can "store" the information about "what to do" in the state space, i.e., each current state belongs to a path through the state space, **started** in one initial state given by the code used to specify the operation

2. in most of the states the automaton tests only one bit from the `flags[q-1:0]` input and if not, a few additional states in the flow-chart solve the problem in most of the cases.

Starting from these remarks the structure of the control automaton can be modified (see Figure 5.13). Because the sequence is **only** initialized using the code `operation[n-1:0]`, this code is used only for addressing the first command line from ROM in a single state in which $MOD = 1$. For this feature we must add n EMUXs and a new output to the ROM to generate the signal MOD. This change allows us to use in a more flexible way the "storing space" of ROM. Because a control sequence can have the dimension very different from the dimension of the other control sequence it is not efficient to allocate fix size part of ROM for each sequence as in we did in the initial solution. The version presented in Figure 5.13 uses for each control sequence only as much of space as needed to store all lines of command.

The second modification refers to the input `flags[q-1:0]`. Because the bits associated with this input are tested in different states, MUX_q selects in each state the appropriate bit using the t-bit field TEST. Thus, the $q-1$ bits associated to the input `flags[q-1:0]` are removed from the input of the ROM, adding only t output bits to ROM. Instead of around q bits we connect only one, T, to the input of ROM.

This new structure works almost the same as the initial structure but the size of ROM is very strongly minimized. Now the size of the ROM is estimated as being:

$$S_{ROM}(2^n).$$

Working with the control automaton in this new version we will make another remark: *the most part of the sequence generated is organized in a linear sequence.* Therefore, the commands associated to the

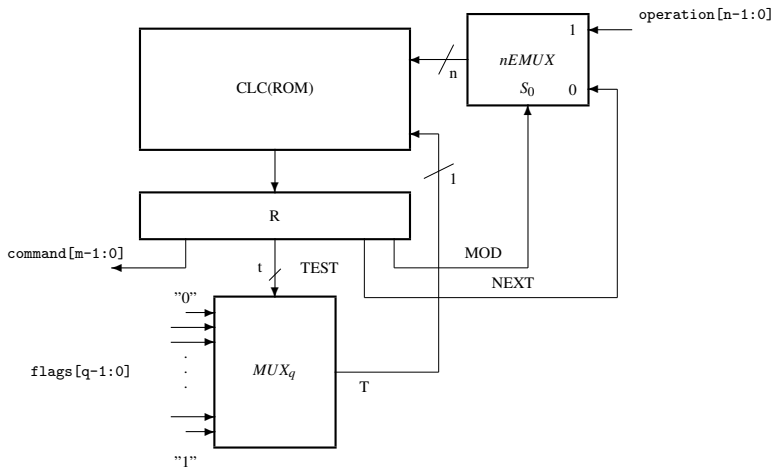


Figure 5.13: **Optimized version of control automata.** The flags received from the controlled system have independent meaning considered in distinct cycles. The flag selected by the code TEST, T, decides from what half of ROM the next state and output will be read.

linear sequences can be stored in ROM at the successive addresses, i.e., the next address for ROM can be obtained incrementing the current address stored in the register R. The structure represented in Figure 5.14 results. What is new in this structure is an increment circuit connected to the output of the state register and a small combinational circuit that transcodes the bits M_1, M_0, T into S_1 and S_0. There are 4 transition modes coded by M_1, M_0:

- `inc`, codded by $M_1M_0 = 00$: the next address for ROM results by incrementing the current address; the selection code must be $S_1S_0 = 00$
- `jmp`, codded by $M_1, M_0 = 01$: the next address for ROM is given by the content of the one field to the output of ROM; the selection code must be $S_1S_0 = 01$
- `cjmp`, codded by $M_1, M_0 = 10$: **if** the value of the selected flag, T, is 1, **then** the next address for ROM is given by the content of the one field to the output of ROM, **else** the next address for ROM results by incrementing the current address; the selection code must be $S_1S_0 = 0T$
- `init`, codded by $M_1, M_0 = 11$: the next address for ROM is selected by $nMUX_4$ from the initialization input `operation`; the selection code must be $S_1S_0 = 1-$

Results the following logic functions for the transcoder TC: $S_1 = M_1M_0$, $S_0 = M_1T + M_0$.

The output of ROM can be seen as a *microinstruction* defined as follows:

```
<microinstruction>::= <setLabel> <Command> <Mod> <Test> <useLabel>;
<command>::= <to be defined when use>;
<mod>::=  jmp | cjmp | init | inc ;
<test>::= <to be defined when use>;
<setLabel>::= setLabel(<number>);
```

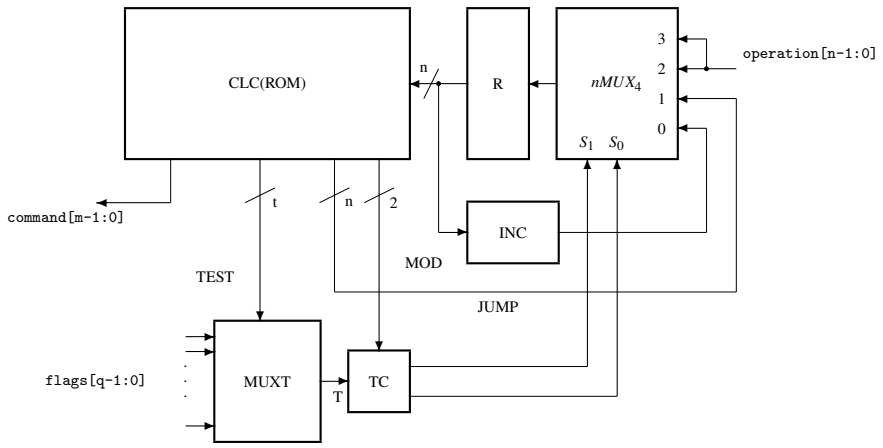


Figure 5.14: **The simplest Controller with ROM (CROM).** The Moore form of control automaton is optimized using an incremented circuit (INC) to compute the most frequent next address for ROM.

```
<useLabel>::= useLabel(<number>);
<number>::= 0 | 1 | ... | 9 | <number><number>;
```

This last version will be called CROM (**C**ontroller with **ROM**) and will be considered, in the present approach, to have enough functional features to be used as controller for the most complex structures described in this book.

Very important comment! The previous version of the control automaton's structure is characterized by two processes:

- the first is the increasing of the structural complexity.
- the second is the decreasing of the dimension and of the complexity of the binary configuration "stored" in ROM.

In this third step both, the size and the complexity of the system grows without any functional improvement. The only effect is reducing the (algorithmic) complexity of ROM's content.

We are in a very important moment of digital system development, in which the physical complexity starts to *compensate* the "symbolic" complexity of ROM's content. Both, circuits and symbols, are structures but there is a big difference between them. The physical structures have simple recursive definitions. The symbolic content of ROM is (almost) random and has no simple definition.

We agree to grow a little the complexity of the physical structure, even the size, in order to create the condition to reduce the effort to set up the complex symbolic content of ROM.

This is the first main "**turning point**" in the development of digital systems. We have here the first sign about the higher complexity of symbolic structures. Using recursive defined objects the physical structures are maintained at smaller complexity, rather than the symbolic structures, that must assume the complexity of the actual problems to be solved with the digital machines. The previous defined CROM structure is so thought as the content of ROM to be *easy designed*, *easy tested* and *easy maintained* because it is complex. This is the first moment, in our approach, when the symbolic structure has more importance than the physical structure of a digital machine.

Example 5.2 *Let's revisit the automaton used to control the MAC system. Now, because a more powerful tool is available, the control automaton will perform three functions,* multiply, multiply and accumulate, no operation*, codded as follows:*

```
mult:  op = 01,
macc:  op = 11,
noop:  op = 00.
```

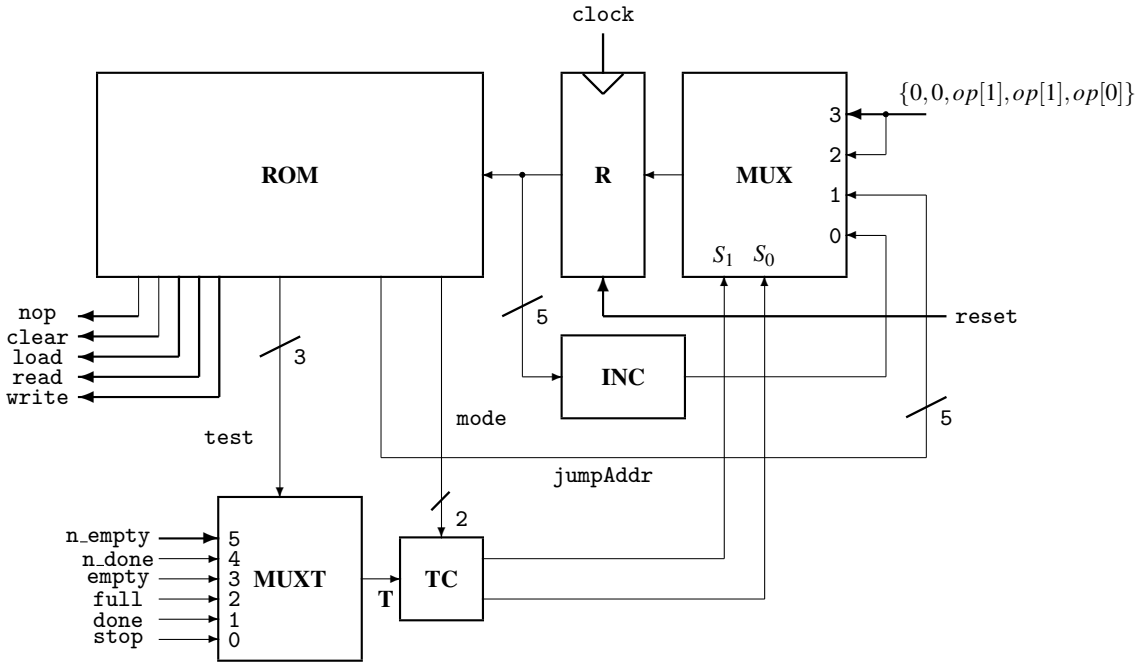


Figure 5.15: **Using a CROM.** A more complex control can be done for Multiply Accumulate System using a CROM instead of a standard finite automaton.

The CROM circuit is actualized in Figure 5.15 with the word of ROM organized as follows:

```
<microinstruction>::= <setLabel><Command> <Mod> <Test> <useLabel>
<command>::= <c1> <c2> <c3> <c4> <c5>
<c1> ::= nop | -
<c2> ::= clear | -
<c3> ::= load | -
<c4> ::= read | -
<c5> ::= write | -
<mod>::=  jmp | cjmp | init | inc
<test>::= empty | full | done | stop | n_done | n_empty
<setLabel>::= setLabel(<number>);
<useLabel>::= useLabel(<number>);
<number>::= 0 | 1 | ... | 9 | <number><number>;
```

The fields `<c1> ... <c5>` *are one-bit fields takeing the value 0 for "-". When nothing is specified, then in the corresponding position is 0. The bit* `end` *is used to end the accumulation. If* `stop = 0`

the `macc` *operation does not end, the system waits for a new pairs of numbers to be multiplied and accumulated. The result is sent out only when* `stop = 1`.

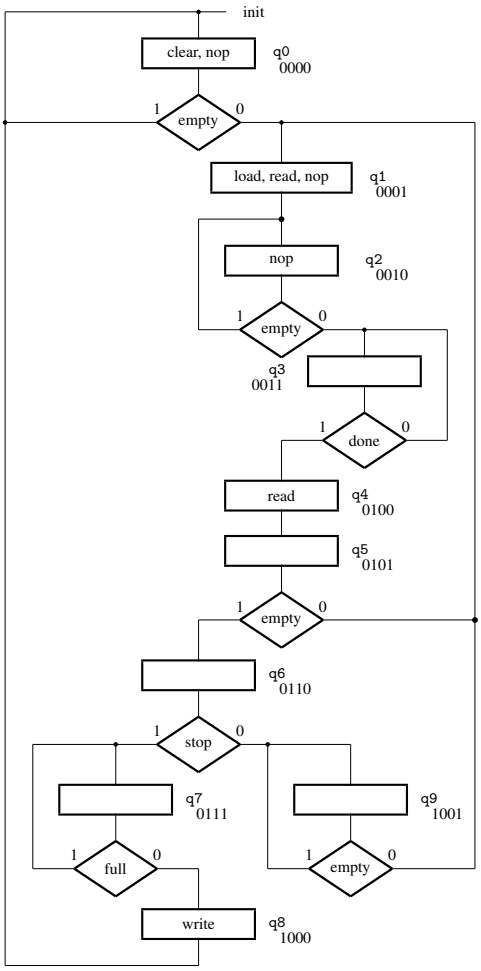


Figure 5.16: **Control flowchart.** The control flowchart for the function `macc`.

The function `mult` *is defined in the flowchart from Figure* **??** *as a Mealy automaton. Because the CROM automaton is defined as a Moore automaton the code sequence will be defined takeing into account the Moore version of the control multiply-accumulate automaton. The function* `macc` *is defined in Figure 5.16 as a Moore automaton. The function* `nop` *consist in looping in the reset state waiting for a command different from* `nop`*. The content of ROM has the following symbolic definition:*

```
// no operation
    setLabel(0)       init;                                          // 00000
// multiplication
    setLabel(1)       nop    clear    cjmp          empty   setLabel(1);
                      nop    load     read          inc;
    setLabel(2)       nop    cjmp     empty         setLabel(2);
    setLabel(3)       cjmp   done     setLabel(3);
    setLabel(4)       cjmp   full     setLabel(4);
```

```
                    read   write    jmp          setLabel(0);
// multiply and accumulate
    setLabel(5)     nop    clear    cjmp         empty   setLabel(5);  // q0
    setLabel(8)     nop    load     read         inc;                  // q1
    setLabel(6)     nop    cjmp     empty        setLabel(6);          // q2
    setLabel(7)     cjmp   n_done   setLabel(7);                       // q3
                    read   inc;                                       // q4
                    cjmp   n_empty  setLabel(8);                      // q5
                    cjmp   stop     setLabel(10);                     // q6
    setLabel(9)     cjmp   empty    setLabel(9);                      // q9
                    jmp    loop8;                                     // q10!
    setLabel(10)    cjmp   full     setLabel(10);                     // q7
                    write  jmp      setLabel(0);                      // q8
```

The binary sequence is stored in ROM starting from the address zero with the line labelled as `setLabel(0)`. *The sequence associated to the function* `mult` *has 6 lines because a Moore automaton has usually more states when the equivalent Mealy version. For* `macc` *function the correspondence with the state are included in commentaries on each line. An additional state (*`q10`*) occurs also here, because this version of CROM can not consider jump addresses depending on the tested bits; only one jump address per line is available.*

The binary image of the previous code asks codes for the fields acting on the loop. ◇

5.4.1 Verilog descriptions for CROM

The most complex part in defining a CROM unit is the specification of the ROM's content. There are few versions to be used. One is to provide the bits using a binary file, another is to generate the bits using a Verilog program. Let us start with the first version.

The description of the unit CROM implies the specification of the following parameters used for dimensioning the ROM:

- `comDim`: the number of bits used to encode the command(s) generated by the control automaton; it depends by the system under control
- `adrDim`: the number of bits used to encode the address for ROM; it depends on the number of state of the control automaton
- `testDim`: the number of bits used to select one of the flags coming back from the controlled system; it depends by the functionality performed by the entire system.

The following description refers to the CROM represented in Figure 5.14. It is dimensioned to generate a 5-bit command, to have maximum 32 internal states, and to evolve according to maximum 8 flags (the dimensioning fits with the simple application presented in the previous example). Adjusting these parameters, the same design can by reused in different projects. Depending on the resulting size of the ROM, its content is specified in various ways. For small sizes the ROM content can be specified by a *hand written* file of bits, while for big sizes it must be generated automatically starting from a "friendly" definition.

A generic Verilog description of the simple CROM already introduced follows:

```
module crom #(`include "0_parameter.v")(output [comDim - 1:0]         command  ,
                                        input  [addrDim - 1:0]        operation,
```

```
                                        input  [(1 << testDim) - 1:0] flags    ,
                                        input                       reset    ,
                                        input                       clock    );
    reg  [addrDim - 1:0] stateRegister;

    wire [comDim + testDim + addrDim + 1:0] romOut  ;
    wire                                    flag    ;
    wire [testDim - 1:0]                    test    ;
    wire [1:0]                              mode    ;
    wire [addrDim - 1:0]                    nextAddr;
    wire [addrDim - 1:0]                    jumpAddr;

    rom rom(.address (stateRegister),
            .data    (romOut       ));

    assign {command, test, jumpAddr, mode} = romOut,
           flag = flags[test];

    mux4 addrSelMux(.out(nextAddr                          ),
                    .in0(stateRegister + 1                 ),
                    .in1(jumpAddr                          ),
                    .in2(operation                         ),
                    .in3(operation                         ),
                    .sel({&mode, (mode[1] & flag | mode[0])}));

    always @(posedge clock) if (reset) stateRegister <= 0       ;
                               else    stateRegister <= nextAddr;
endmodule
```

The simple uniform part of the previous module consists in two multiplexer, an increment circuit, and a register. The complex part of the module is formed by a very small one (the transcoder) and a big one the ROM.

From the simple only the `addrSelMux` multiplexor asks for a distinct module. It follows:

```
module mux4 #(‘include "0_parameter.v")(out, in0, in1, in2, in3, sel);
   input  [1:0]           sel               ;
   input  [addrDim - 1:0] in0, in1, in2, in3;
   output [addrDim - 1:0] out               ;
   reg    [addrDim - 1:0] out               ;
   always @(in0 or in1 or in2 or in3 or sel)
       case(sel)
           2’b00: out = in0;
           2’b01: out = in1;
           2’b10: out = in2;
           2’b11: out = in3;
       endcase
endmodule
```

The big complex part has a first version described by the following Verilog module:

```
module rom #(‘include "0_parameter.v")
            (input  [addrDim - 1:0]                    address,
```

```
            output [comDim + testDim + addrDim + 1:0] data   );
    reg [comDim + testDim + addrDim + 1:0] mem [0:(1 << addrDim) - 1];

    initial $readmemb("0_romContent.v", mem); // the fix content of the memory

    assign data = mem[address]; // it is a read only memory
endmodule
```

The file `0_parameter.v` defines the dimensions used in the project `crom`. It must be placed in the same folder with the rest of the files defining the project. For our example its content is:

```
parameter comDim  = 5,
          addrDim = 5,
          testDim = 3
```

The `initial` line loads in background, in a transparent mode, the memory module `mem`. The module `rom` does not have explicit writing capabilities, behaving like a "read only" device. The synthesis tools are able to infer from the previous description that it is about a ROM combinational circuit.

The content of the file `0_romContent.v` is filled up according to the micro-code generated in *Example 7.6*. Obviously, after the first 4 line the our drive to continue is completely lost.

```
/* 00 */    00000_000_00000_11
/* 01 */    11000_011_00001_10
/* 02 */    10100_000_00000_00
/* 03 */    10000_011_00011_10
//          ...
/* 30 */    00000_000_00000_00
/* 31 */    00000_000_00000_00
```

Obviously, after filling up the first 4 lines our internal drive to continue is completely lost. The full solution asks for 270 bits free of error bits. Another way to generate them must be found!

5.4.2 Binary code generator

Instead of defining and writing bit by bit the content of the ROM, using a hand written file (in our example `0_romContent.v`), is easiest to design a Verilog description for a "machine" which takes a file containing lines of *microinstructions* and translate it into the corresponding binary representation. Then, in the module `rom` the line `initial ...;` must be substituted with the following line:

```
‘include "codeGenerator.v" // generates ROM’s content using to ’theDefinition.v’
```

which will act by including the the description of a loading mechanism for the memory `mem`. The code generating machine is a program which has as input a file describing the behavior of the automaton. Considering the same example of the control automaton for MAC system, the file `theDefinition` is a possible input for our code generator. It has the following form:

```
// MAC control automaton
    setLabel(0);  init;
    // multiplication
    setLabel(1);  nop;   clear;       cjmp;        empty;       useLabel(1);
                  nop;   load;        read;        inc;
    setLabel(2);  nop;   cjmp;        empty;       useLabel(2);
    setLabel(3);  cjpm;  done;        useLabel(3);
    setLabel(4);  cjmp;  full;        useLabel(4);
```

```
                read;  write;       jmp;         useLabel(0);
    // multiply & accumulate
    setLabel(5);  nop;   clear;     cjmp;        empty;       useLabel(5);
    setLabel(8);  nop;   load;      read;        inc;
    setLabel(7);  cjmp;  notDone;   useLabel(7);
                  read;  inc;
                  cjmp;  notEmpty;  useLabel(8);
                  cjmp;  stop;      useLabel(10);
    setLabel(9);  cjmp;  empty      useLabel(9);
                  jmp;   useLabel(8);
    setLabel(10); cjmp;  full;      useLabel(10);
                  write; jmp;       useLabel(0);
```

The `theDefinition` file consist in a stream of Verilog **tasks**. The execution of these tasks generate the ROM'a content.

The file `codeGenerator.v` "understand" and use the file `theDefinition`, whose content follows:

```
// Generate the binary content of the ROM
    reg       nopReg        ;
    reg       clearReg      ;
    reg       loadReg       ;
    reg       readReg       ;
    reg       writeReg      ;
    reg [1:0] mode          ;
    reg [2:0] test          ;
    reg [4:0] address       ;
    reg [4:0] counter       ;
    reg [4:0] labelTab[0:31];

    task endLine;
      begin
        mem[counter] =
          {nopReg, clearReg, loadReg, readReg, writeReg, mode, test, address};
        nopReg   = 1'b0;
        clearReg = 1'b0;
        loadReg  = 1'b0;
        readReg  = 1'b0;
        writeReg = 1'b0;
        counter = counter + 1;
      end
    endtask

    // sets labelTab in the first pass associating 'counter' with 'labelIndex'
    task setLabel; input [4:0] labelIndex; labelTab[labelIndex] = counter; endtask
    // uses the content of labelTab in the second pass
    task useLabel; input [4:0] labelIndex; begin address = labelTab[labelIndex];
                                                 endLine;
                                           end
    endtask
    // external commands
    task nop  ; nopReg   = 1'b1; endtask
    task clear; clearReg = 1'b1; endtask
```

```
task load ; loadReg  = 1'b1; endtask
task read ; readReg  = 1'b1; endtask
task write; writeReg = 1'b1; endtask
// transition mode
task inc ; begin mode = 2'b00; endLine; end endtask
task jmp ; mode = 2'b01; endtask
task cjmp; mode = 2'b10; endtask
task init; begin mode = 2'b11; endLine; end endtask
// flag selection
task empty  ; test = 3'b000; endtask
task full   ; test = 3'b001; endtask
task done   ; test = 3'b010; endtask
task stop   ; test = 3'b011; endtask
task notDone; test = 3'b100; endtask
task notEmpt; test = 3'b101; endtask

initial begin counter  = 0;
              nopReg   = 0;
              clearReg = 0;
              loadReg  = 0;
              readReg  = 0;
              writeReg = 0;
              `include "theDefinition.v"; // first pass
              `include "theDefinition.v"; // second pass
        end
```

The file `theDefinition` is included twice because if a label is used before it is defined, only at the second pass in the memory `labelTab` the right value of a label will be found when the task `useLabel` is executad.

5.5 Automata vs. Combinational Circuits

As we saw, both combinational circuits (0-OS) and automata (2-OS) execute digital functions. Indeed, there are combinational circuits performing addition or multiplication, but there are also sequential circuits performing the same functions. What is the correlation between a gates network and an automaton executing the same function? What are the conditions in which we can transform a combinational circuit in an automaton or conversely? The answer to this question will be given in this last section.

Let be a Mealy automaton, his two CLCs (*LOOP CLC* and *OUT CLC*), the initial state of the automaton, $q(t_0)$ and the input sequence for the first n clock cycle: $x(t_0),\ldots,x(t_{n-1})$. The combinational circuit that generates the corresponding output sequence $y(t_0),\ldots,y(t_{n-1})$ is represented in Figure 5.17. Indeed, the first pair *LOOP CLC*, *OUT CLC* computes the first output, $y(t_0)$, and the next state, $q(t_1)$ to be used by the second pair of CLCs to compute the second output and the next state, and so on.

Example 5.3 *The ripple carry adder (Figure* **??***) has as correspondent automaton the* adder automaton *from the serial adder (Figure 5.1).* $\diamond$

Should be very interesting to see how a complex problem having associated a finite automaton can be solved starting from a combinational circuit and reducing it to a finite automaton. Let us revisit in the next example the problem of recognizing strings from the set 1^a0^b, for $a,b>0$.

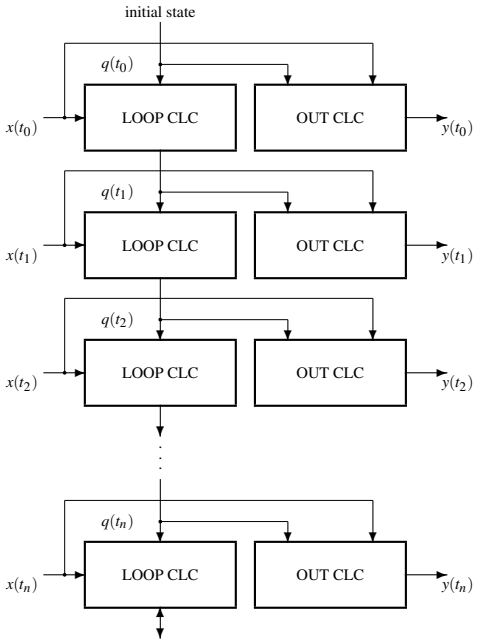


Figure 5.17: **Converting an automata into a combinational circuit.** The conversion rule from the finite (Mealy) automaton into a combinational logic circuit means to use a pair of circuits (LOOP CLC, OUTPUT CLC) for each clock cycle. The time dimension is transformed in space dimension.

Example 5.4 *The universal combinational circuit (see 2.3.1) is used to recognize all the strings having the form:*

$$x_0, x_1, \ldots x_i, \ldots x_{n-1} \in 1^a 0^b$$

for $a, b > 0$, and $a + b = n$. The function performed by the circuit will be:

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

which takes value 1 for the following inputs:

$$\begin{aligned} x_{n-1}, \ldots, x_0 &= 0000 \ldots 01 \\ x_{n-1}, \ldots, x_0 &= 000 \ldots 011 \\ x_{n-1}, \ldots, x_0 &= 00 \ldots 0111 \\ &\ldots \\ x_{n-1}, \ldots, x_0 &= 00011 \ldots 1 \\ x_{n-1}, \ldots, x_0 &= 0011 \ldots 11 \\ x_{n-1}, \ldots, x_0 &= 011 \ldots 111 \end{aligned}$$

Any function $f(x_{n-1}, \ldots, x_0)$ of n variables can be expressed using certain minterms from the set of 2^n minterms of n variables. Our functions uses only $n-2$ minterms from the total number of 2^n. They are:

$$m_{2^i - 1}$$

for $i = 1, \ldots (n-1)$, i.e., the functions takes the value 1 for m_1 **or** *m_3* **or** *m_7* **or** *m_{15}* **or** *....*

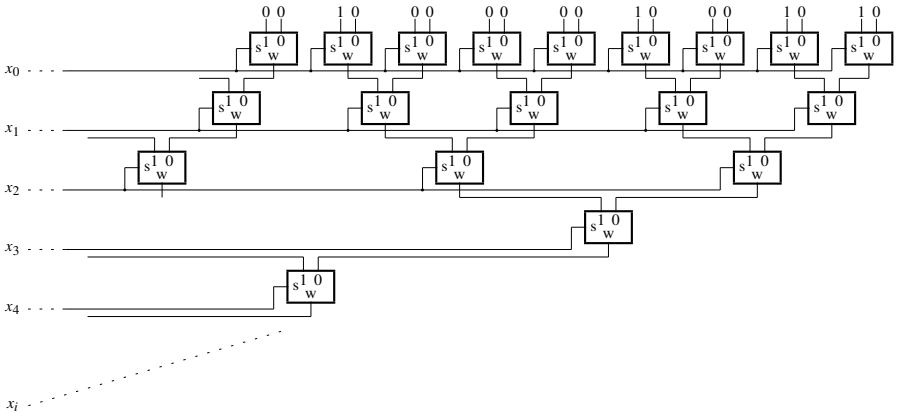


Figure 5.18: **The Universal Circuit "programmed" to recognize 1^a0^b.** A full tree of 2^n EMUXs are used to recognize the strings belonging to 1^a0^b. The "program" is applied on the selected inputs of the first level of EMUXs.

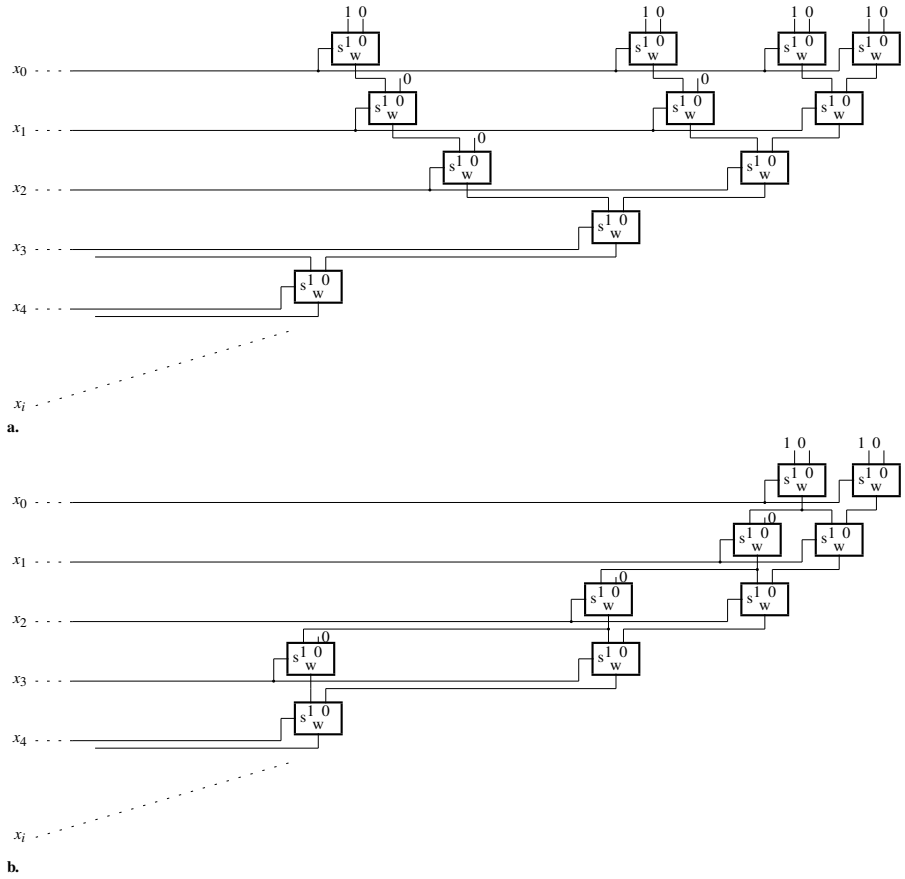


Figure 5.19: **Minimizing the Universal Circuit "programmed to recognize 1^a0^b. a.** The first step of minimizing the full tree of $2^n - 1$ EMUXs to a tree containing $0.5n(n+1)$ EMUXs. Each EMUX selecting between 0 and 0 is substituted with a connection to 0. **b.** The minimal combinational network of EMUXs obtained removing the duplicated circuits. The resulting network is a linear stream of identical CLCs.

Figure 5.18 represents the universal circuits receiving as "program" the string:

$$\ldots 001000000010001010$$

where 1s corresponds to minterms having the value 1, and 0s to the minterms having the value 0.

Initially the size of the resulting circuit is too big. For an n-bit input string from x_0 to x_{n-1} the circuits contains $2^n - 1$ elementary multiplexors. But, a lot of EMUXs have applied 0 on both selected inputs. They will generate 0 on their outputs. If the multiplexors generating 0 are removed and substituted with connections to 0, then the resulting circuit containing only $n(n-1)/2$ EMUXs is represented in Figure 5.19a.

The circuit can be more reduced if we take into account that some of them are identical. Indeed, on the first line all EMUXs are identical an the third (from left to right) can do the "job" of the first tree circuits. Therefore, the output of the third circuit from the first line will be connected to the input of all the circuits from the second line. Similarly, on the second line we will maintain only two EMUXs, and so on on each line. Results the circuit from Figure 5.19b containing $(2n-1)$ EMUXs.

This last form consists in a serial composition made using the same combinational circuit: an EMUX and an 2-input AND (the EMUX with the input 0 connected to 0). Each stage of the circuit receives one input value starting with x_0. The initial circuit receives on the selected inputs a fix binary configuration (see Figure 5.19b). It can be considered as the initial state of the automaton. Now we are in the position to transform the circuit in a finite half-automaton connecting the emphasized module in the loop with a 2-bit state register (see Figure 5.20a).

The resulting half-automaton can be compared with the half-automaton from Figure **??***, reproduced in Figure 5.20b. Not-surprisingly they are identical.* $\diamond$

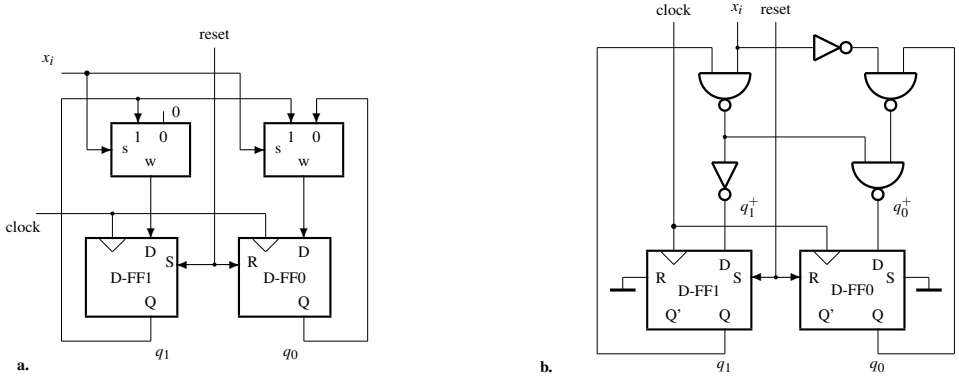


Figure 5.20: **From a big and simple CLC to a small and complex finite automata.** **a**. The resulting half-automaton obtained collapsing the stream of identical circuits. **b**. Minimizing the structure of the two EMUXs results a circuit identical with the solution provided in Figure **??** for the same problem.

To transform a combinational circuit in a (finite) automaton the associated tree (or trees) of EMUXs must degenerate into a linear graph of identical modules. An interesting problem is: how many of "programs", $P = m_{p-1}, m_{p-2}, \ldots m_0$, applied as "leaves" of Universal Circuit allows the tree of EMUXs to be reduced to a linear graph of identical modules?

5.6 The Circuit Complexity of a Binary String

Greg Chaitin taught us that simplicity means the possibility to compress. He expressed the complexity of a binary string as being the length of the shortest program used to generate that string. An alternative form to express the complexity of a binary string is to use the size of the smallest circuit used to generate it.

Definition 5.4 *The* **circuit complexity** *of a binary string P of length p, $CC_P(p)$, is the size of the minimized circuit used to generate it.* $\diamond$

Definition 5.5 *The universal circuit used to generate any p-bit sting, pU-Generator, consists in a nU-Circuit programmed with the string to be generated and triggered by a resetable counter (see Figure 5.21).* $\diamond$

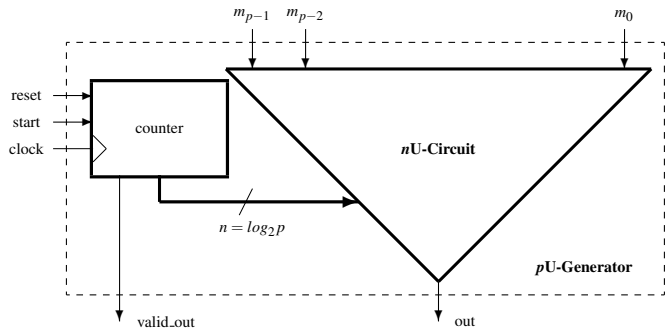


Figure 5.21: **The universal string generator.** The counter, starting from zero, selects to the output `out` the bits of the "program" one by one starting with m_0.

According to the actual content of the "program" $P = m_{p-1} \ldots m_0$ the nU-Circuit can be reduced to a minimal size using techniques previously described in the section 2.3. The minimal size of the counter is in $O(log\ p)$ (the "first" proposal for an actual value is $11(1 + log_2\ p) + 5$). Therefore, the minimal size of pU-Generator, used to generate an actual string of p bits is the very precisely defined number $CC_P(p)$.

Example 5.5 *Let us compute the circuit size of the following 16-bit strings:*

$$P1 = 0000000000000000$$

$$P2 = 1111111111111111$$

$$P3 = 0101000001010000$$

$$P4 = 0110100110110001$$

For both, P1 and P2 the nU-Circuit is reduced to circuits containing no gates. Therefore, $CC(P1) = CC(P2) = 11(1 + log_2\ 16) + 5 + 0 = 60$.

For P3, applying the removing rules the first level of EMUXs in nU-Circuitis is removed and to the inputs of the second level the following string is applied:

$$x_0', x_0', 0, 0, x_0', x_0', 0, 0$$

We continue applying removing and reducing rules. Results the inputs of the third level:

$$x_0' x_2, x_0' x_2$$

The last level is removed because its inputs are identic. The resulting circuit is: $x_0' x_2$. It has the size 3. Therefore $CC(P3) = 60 + 3 = 63$.

For P4, applying the removing rules results the following string for the second level of EMUXs:

$$x_0', x_0, x_0, x_0', x_0, 1, 0, x_0'$$

No removing or reducing rule apply for the next level. Therefore, the size of the resulting circuit is: $CC(P4) = 1 + 7S_{EMUX} + 88 = 103$. $\diamond$

The main problem in computing the circuit complexity of a string is to find the minimal form of a Boolean function. Fortunately, there are rigorous formal procedures to minimize logic functions (see Appendix C.4 for some of them). (**Important note**: the entire structure of pU-Generator can be designed *composing* and closing *loops* in a structure containing only elementary multiplexors and inverters. In the langauge of the partial recursive functions these circuits perform the elementary *selection* and the elementary *increment*. "Programming" uses only the function *zero* and the elementary *increment*. No restrictions imposed by primitive recursiveness or minimalization are applied!)

An important problem rises: *how many of the n-bit variable function are simple?* The answer comes from the next theorem.

Theorem 5.1 *The weight, w, of Turing-computable functions, of n binary variables, in the set of the formal functions decreases twice exponentially with n.* $\diamond$

Proof Let be a given n. The number of formal n-input function is $N = 2^{2^n}$, because the definition are expressed with 2^n bits. Some of this functions are Turing-computable. Let be these functions defined by the compressed m-bit strings. The value of m depends on the actual function, but is realized the condition that $max(m) < 2^n$ and *m does not depends by n*. Each compressed form of m bits corresponds only to one 2^n-bit uncompressed form. Thus, the ratio between the Turing-computable function of and the formal function, both of n variables, is smaller than

$$max(w) = 2^{-(2^n - max(m))}.$$

And, because $max(m)$ does not depends by n, the ratio has the same form for no matter how big becomes n. Results:

$$max(w) = const/2^{2^n}.$$

$\diamond$

A big question arises: how could be combinational circuits useful with this huge ratio between complex circuits and simple circuits? An answer could be: potentially this ratio is very high, but actually, in the real world of problems this ratio is very small. It is small because we do not need to compute too

many complex functions. Our mind is usually attracted by simple functions in a strange manner for which we do not have (yet?) a simple explanation.

The Turing machine is **limited** to perform only *partial recursive functions* (see Chapter 9 in this book). The *halting problem* is an example of a problem that has no solutions on a Turing machine (see subsection 9.3.5???? in this book). Circuits are more powerful but they are not so easy"programmed" as the Turing Machine, and the related systems. We are in a paradoxical situation: *the circuit does not need algorithms and Turing Machine is limited only to the problems that have an algorithm*. But without algorithms many solutions exist and we do not know the way to find them. **The complexity of the way to find of a solution becomes more and more important**.

The **working hypothesis** will be that *at the level of combinational (without autonomy) circuits the segregation between simple circuits and complex programs is not productive*. In most of cases the digital system grows toward higher orders where *the autonomy of the structures allow an efficient segregation between simple and complex.*

5.7 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[1]. 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.

[1] 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.

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?

5.8 Problems

Problem 5.1 *Justify the reason for which the LIFO circuit works properly without a reset input, i.e., the initial state of the address counter does not matter.*

Problem 5.2 *How behaves* `simple_stack`.

Problem 5.3 *Design a LIFO memory using a synchronous RAM (SRAM) instead of an asynchronous one as in the embodiment represented in Figure 8.1.*

Problem 5.4 *Some applications ask the access to the last two data stored into the LIFO. Call them* `tos`*, for the last pushed data, and* `prev_tos` *for the previously pushed data. Both accessed data can be popped from stack. Double push is allowed. The accessed data can be rearranged swapping their position. Both,* `tos` *and* `prev_tos` *can be pushed again in the top of stack. Design such a LIFO defined as follows:*

```
module two_head_lifo(  output  [31:0]  tos         ,
                       output  [31:0]  prev_tos    ,
                       input   [31:0]  in          ,
                       input   [31:0]  second_in   ,
                       input   [2:0]   com         , // the operation
                       input           clock       );
   // the semantics of 'com'
   parameter   nop         = 3'b000, // no operation
               swap        = 3'b001, // swap the first two
               pop         = 3'b010, // pop tos
               pop2        = 3'b011, // pop tos and prev_tos
               push        = 3'b100, // push in as new tos
               push2       = 3'b101, // push 'in' and 'second_in'
               push_tos    = 3'110b, // push 'tos' (double tos)
               push_prev   = 3'b111; // push 'prev_tos'
endmodule
```

Problem 5.5 *Write the Verilog description of the FIFO memory represented in Figure 5.9.*

Problem 5.6 *Redesign the FIFO memory represented in Figure 5.9 using a synchronous RAM (SRAM) instead of the asynchronous RAM.*

Problem 5.7 *There are application asking for a warning signal before the FIFO memory is full or empty. Sometimes full and empty come to late for the system using the FIFO memory. For example,* no more then 3 write operation are allowed, *or* no more than 7 read operation are allowed *are very useful in systems designed using pipeline techniques. The threshold for this warning signals is good to be programmable. Design a 256 8-bit entries FIFO with warnings activated using a programmable threshold. The interconnection of this design are:*

```
module th_fifo(output  [7:0]   out     ,
               input   [7:0]   in      ,
               input   [3:0]   write_th, // write threshold
               input   [3:0]   read_th , // read threshold
               input           write   ,
               input           read    ,
               output          w_warn  , // write warning
               output          r_warn  , // read warning
               output          full    ,
               output          empty   ,
               input           reset   ,
               input           clock   );
endmodule
```

Problem 5.8 *A synchronous FIFO memory is written or read using the same clock signal. There are many applications which use a FIFO to interconnect two subsystems working with different clock signals. In this cases the FIFO memory has an additional role: to cross from the clock domain* `clock_in` *into another clock domain,* `clock_out`*. Design an* **asynchronous FIFO** *using a synchronous RAM.*

Problem 5.9 *A serial memory implements the data structure of a fix length circular list. The first location is accessed, for write or read operation, activating the input* `init`*. Each read or write operation move the access point one position right. Design an 8-bit word serial memory using a synchronous RAM as follows:*

```
module serial_memory(  output  [7:0]   out     ,
                       input   [7:0]   in      ,
                       input           init    ,
                       input           write   ,
                       input           read    ,
                       input           clock   );
endmodule
```

Problem 5.10 *A list memory is a circuit in which a list can be constructed by* `insert`*, can be accessed by* `read_forward`*,* `read_back`*, and modified by* `insert`*,* `delete`*. Design such a circuit using two LIFOs.*

Problem 5.11 *Design a sequential multiplier using as combinational resources only an adder, a multiplexors.*

Problem 5.12 *Write the behavioral and the structural Verilog description for the MAC circuit represented in Figure 5.11. Test it using a special test module.*

Problem 5.13 *Redesign the MAC circuit represented in Figure 5.11 adding pipeline register(s) to improve the execution time. Evaluate the resulting speed performance using the parameters form Appendix E.*

Problem 5.14 *How many 2-bit code assignment for the half-automaton from Example 4.2 exist? Revisit the implementation of the half-automaton for four of them different from the one already used. Compare the resulting circuits and try to explain the differences.*

Problem 5.15 *Ad to the definition of the half-automaton from Example 4.2 the output circuits for: (1)* `error`*, a bit indicating the detection of an incorrectly formed string, (2)*`ack`*, another bit indicating the acknowledge of a well formed sting.*

Problem 5.16 *Multiplier control automaton can be defined testing more than one input variable in some states. The number of states will be reduced and the behavior of the entire system will change. Design this version of the multiply automaton and compare it with the circuit resulted in Example 4.3. Reevaluate also the execution time for the multiply operation.*

Problem 5.17 *Revisit the system described in Example 4.3 and design the finite automaton for multiply and accumulate (MACC) function. The system perform MACC until the input FIFO is empty and* `end = 1`.

Problem 5.18 *Design the structure of TC in the CROM defined in 4.4.3 (see Figure 5.14). Define the codes associated to the four modes of transition (*`jmp, cjmp, init, inc`*) so as to minimize the number of gates.*

Problem 5.19 *Design an easy to actualize Verilog description for the CROM unit represented in Figure 5.14.*

Problem 5.20 *Generate the binary code for the ROM described using the symbolic definition in* Example 4.4.

Problem 5.21 *Design a fast multiplier converting a sequential multiplier into a combinational circuit.*

5.9 Projects

Project 5.1 *Finalize Project 1.2 using the knowledge acquired about the combinational and sequential structures in this chapter and in the previous two.*

Project 5.2 *The idea of simple FIFO presented in this chapter can be used to design an actual block having the following additional features:*

- *fully buffered inputs and outputs*
- *programmable thresholds for generating the* `empty` *and* `full` *signals*

- *asynchronous clock signals for input and for output (the design must take into consideration that the two clocks –* `clockIn`*,* `clockOut` *– are considered completely asynchronous)*
- *the* `read` *or* `write` *commands are executed only if the it is possible (reads only if not-empty, or writes only if not-full).*

The module header is the following:

```
module asyncFIFO #(‘include "fifoParameters.v")
   (   output  reg [n-1:0] out     ,
       output  reg         empty   ,
       output  reg         full    ,
       input       [n-1:0] in      ,
       input               write   ,
       input               read    ,
       input       [m-1:0] inTh    , // input threshold
       input       [m-1:0] outTh   , // output threshold
       input               reset   ,
       input               clockIn ,
       input               clockOut);
   // ...
endmodule
```

The file `fifoParameters.v` *has the content:*

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

Project 5.3 *Design a stack execution unit with a 32-bit ALU. The stack is 16-level depth (*`stack0, stack1, ... stack15`*) with* `stack0` *assigned as the top of stack. ALU has the following functions:*

- *add: addition*
  `{stack0, stack1, stack2, ...} <= {(stack0 + stack1), stack2, stack3,...}`
- *sub: subtract*
  `{stack0, stack1, stack2, ...} <= {(stack0 - stack1), stack2, stack3,...}`
- *inc: increment*
  `{stack0, stack1, stack2, ...} <= {(stack0 + 1), stack1, stack2, ...}`
- *dec: decrement*
  `{stack0, stack1, stack2, ...} <= {(stack0 - 1), stack1, stack2, ...},`
- *and: bitwise AND*
  `{stack0, stack1, stack2, ...} <= {(stack0 & stack1), stack2, stack3,...}`
- *or: bitwise OR*
  `{stack0, stack1, stack2, ...} <= {(stack0 | stack1), stack2, stack3,...}`
- *xor: bitwise XOR*
  `{stack0, stack1, stack2, ...} <= {(stack0` $\oplus$ `stack1), stack2, stack3,...}`

- *not: bitwise NOT*
  `{stack0, stack1, stack2, ...} <= {(~stack0), stack1, stack2, ...}`
- *over:*
  `{stack0, stack1, stack2, ...} <= {stack1, stack0, stack1, stack2, ...}`
- *dup: duplicate*
  `{stack0, stack1, stack2, ...} <= {stack0, stack0, stack1, stack2, ...}`
- *rightShift: right shift one position (integer division)*
  `{stack0, stack1, ...} <= {({1'b0, stack0[31:1]}), stack1, ...}`
- *arithShift: arithmetic right shift one position*
  `{stack0, stack1, ...} <= {({stack0[31], stack0[31:1]}), stack1, ...}`
- *get: push dataIn in top of stack*
  `{stack0, stack1, stack2, ...} <= {dataIn, stack0, stack1, ...},`
- *acc: accumulate dataIn*
  `{stack0, stack1, stack2, ...} <= {(stack0 + dataIn), stack1, stack2, ...},`
- *swp: swap the last two recordings in stack*
  `{stack0, stack1, stack2, ...} <= {stack1, stack0, stack2, ...}`
- *nop: no operation*
  `{stack0, stack1, stack2, ...} <= {stack0, stack1, stack2, ...}.`

All the register buffered external connections are the following:

- `dataIn[31:0]` *: data input provided by the external subsystem*
- `dataOut[31:0]` *: data output sent from the top of stack to the external subsystem*
- `aluCom[3:0]` *: command code executed by the unit*
- `carryIn` *: carry input*
- `carryOut` *: carry output*
- `eqFlag` *: is one if (stack0 == stack1)*
- `ltFlag` *: is one if (stack0 ¡ stack1)*
- `zeroFlag` *: is one if (stack0 == 0)*

Project 5.4

Chapter 6

PROCESSORS:
Third order, 3-loop digital systems

6.1 Implementing finite automata with "intelligent registers"

The automaton function rises at the second order level, but this function can be better implemented using the facilities offered by the systems having a higher order. Thus, in this section we resume a previous example using the feature offered by 3-OS. The main effect of these new approaches: the *ratio between the simple circuits and the complex circuits grows*, without spectacular changes in the size of circuits. The main conclusion of this section: *more autonomy means less complexity*.

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 6.1 *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*

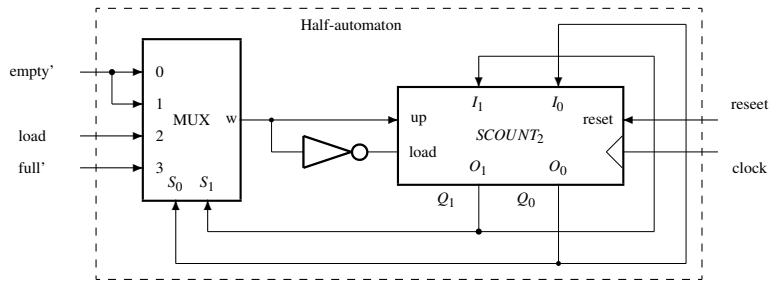


Figure 6.1: **Finite half-automaton implemented with a setable counter.** The last implementation of the half-automaton associated with FA from Figure **??** (with the function defined in Figure **??** where the states coded in parenthesis). A synchronous two-bit counter is used as state register. The simple four-input MUX commands the counter.

Results the very simple (not necessarily very small) implementation represented in Figure 6.1, 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.* $\diamond$

Example 6.2 *This example is also a remake. The half-automaton of the automaton which controls the operation* `macc` *in Example 4.6 will be implemented using a presetable counter as register. See Figure 5.16 for the state encoding. The idea is to* have in the flow-chart as many as possible transitions by incrementing.

Building the solution starts from a $SCOUNT_4$ and a MUX_4 connected as in Figure 6.2. The multiplexer selects the counter's operation (load or up-increment) in each state according to the flow-chart description. For example in the state 0000 the transition is made by counting if $empty = 0$, else the state remains the same. Therefore, the multiplexer selects the value of $empty'$ to the input U of the counter.

The main idea is that the loading inputs I_3, I_2, I_1 and I_0 must have correct values only if in the current state the transition can be made by loading a certain value in the counter. Thus, in the definition of the logical functions associated with these inputs we have many "don't care"s. Results the circuit represented in Figure 6.2. The random part of the circuit is designed using the transition diagrams from Figure 6.3.

The resulting structure has a minimized random part. We assumed even the risk of increasing the recursive defined part of the circuit in order to reduce the random part of it. $\diamond$

Now, the autonomous device that allows reducing the randomness is the counter used as state register. An adequate state assignment implies many transitions by incrementing the state code. Thus, the basic function of the counter is many times involved in the state transition. Therefore, the second loop of the system, the simple defined "loop that counts", is frequently used by the third loop, the random loop. The simple command UP, on the third loop, is like a complex "macro" executed by the second loop using simple circuits. This hierarchy of autonomies simplifies the system, because at the higher level the loop uses simple commands for complex actions. Let us remember:

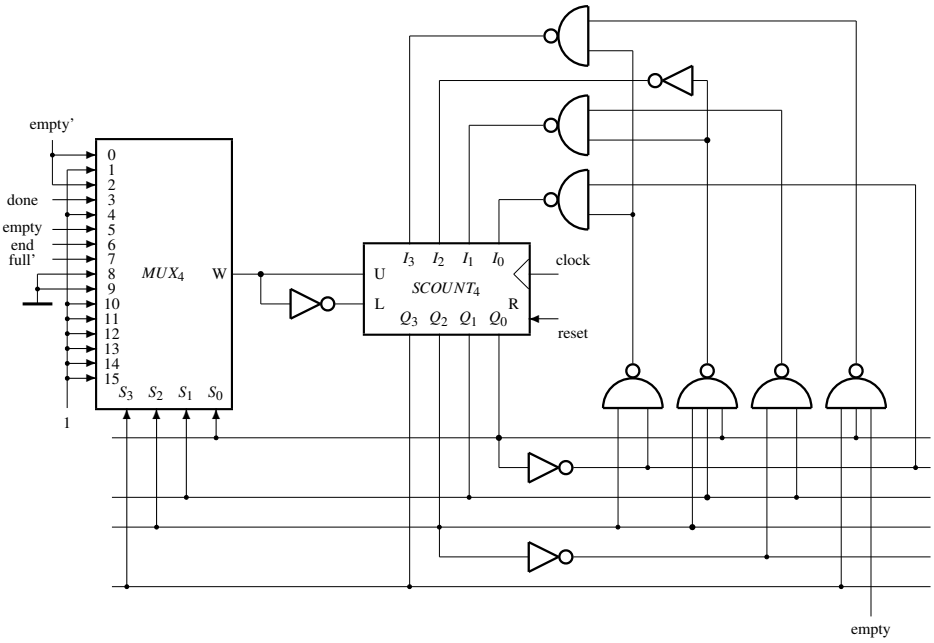


Figure 6.2: **Finite half-automaton for controlling the function** `macc`**.** The function was previously implemented using a CROM in Example 4.6.

- the loop over a true register (in 2-OS) uses the simple commands for the simplest actions: **load 0** in D flip-flop and **load 1** in D flip-flop

- the loop over a "JK register" (in 3-OS) uses beside the previous commands the following: **no op** (remain in the same state!) and **switch** (switch in the complementary state!)

- the loop over a $SCOUNT_n$ substitutes the command **switch** with the same simple expressed, but more powerful, command **increment**.

The "architecture" used on the third loop is more powerful than the two previous. Therefore, the effort of this loop to implement the same function is smaller, having the simpler expression: a reduced random circuit.

The segregation process is more deep, thus we imply in the designing process more simple, recursive defined, circuits. The *apparent complexity* of the previous solution is reduced towards, maybe on, the actual complexity. The complexity of the simple part is a little increased in order to "pay the price" for a strong minimization of the random part of the system. The quantitative aspects of our small example are not very significant. Only the design of the actual large systems offers a meaningful example concerning the quantitative effects.

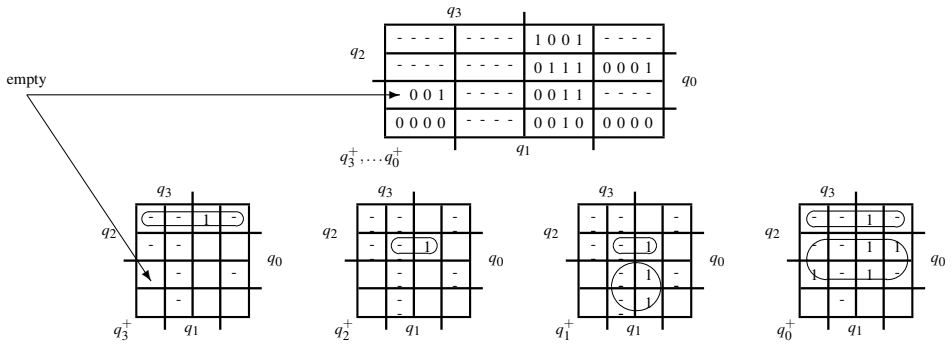


Figure 6.3: **Transition diagrams for the presetable counter used as state register.** The complex (random) part of the automaton is represented by the loop closed to the load input of the presetable counter.

6.2 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 6.4a). This circuit performs logic and arithmetic functions on data stored in its own state register called accumulator – ACC –, used as `left` operand and on the data received on its input `in`, used as `right` operand. A first version uses a **control** automaton to send commands to ALA, receiving back one flag: `crout`.

A second version of the system contains an additional D flip-flop used to store the value of the CR_{out} signal, in each clock cycle when it is enabled (E = 1), in order to be applied on the CR_{in} input of ALU. The control automaton is now substituted with a **command** automaton, used only to issue commands, without receiving back any flag.

Follow two example of using this ALA, one without an additional loop and another with the third loop closed trough a simple D flip-flop.

Version 1: the controlled Arithmetic & Logic Automaton

In the first case ALA is **controlled** (see Figure 6.4a) 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:

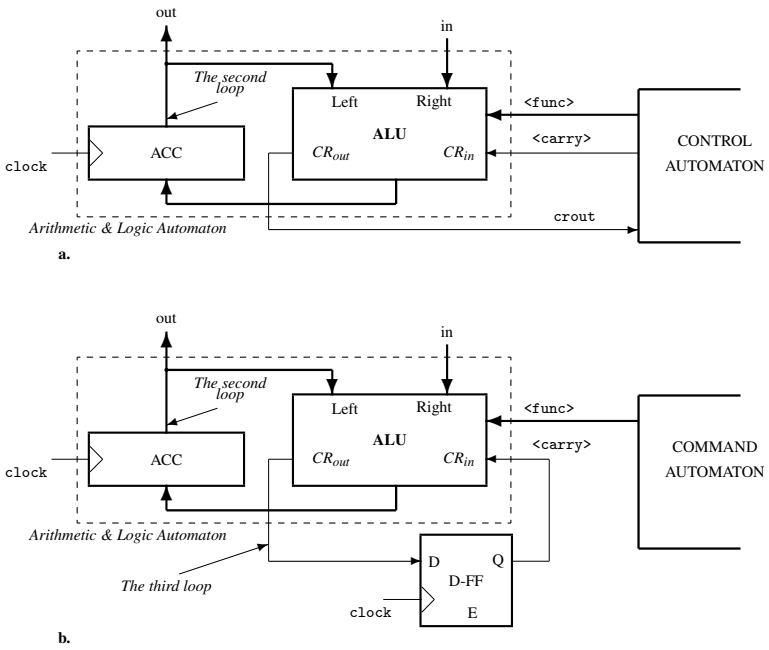


Figure 6.4: **The third loop closed over an arithmetic and logic automaton. a.** The basic structure: a simple automaton (its loop is closed through a simple combinational circuit: ALU) working under the supervision of a control automaton. **b.** The improved version, with an additional 1-bit state register 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.

```
        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.

6.3 Loop coupled automata: the second "turning point"

This last step in building 3-OS stresses specifically on the maximal segregation between the **simple physical structure** and the **complex symbolic structures**. The third loop allows us to make a deeper

segregation between simple and complex.

We are in the point where the process of segregation between simple and complex physical structures ends. The physical structures reach the stage from which the evolution can be done only coupled with the symbolic structures. From this point a machine means: *circuits that* **execute** *or* **interpret** *bit configurations structured under restrictions imposed by the formal languages used to describe the functionality to be performed.*

6.3.1 Push-down automata

The first example of loop coupled automata uses a finite automaton and a functional automaton: the stack (LIFO memory). A finite complex structure is interconnected with an "infinite" but simple structure. The simple and the complex are thus perfectly segregated. This approach has the role of minimizing the size of the random part. More, this loop affects the *magnitude order* of the randomness, instead of the previous examples (*Arithmetic & Logic Automaton*) in which the size of randomness is reduced only by a constant. The proposed structure is a well known system having many theoretical and practical applications: the *push-down automaton*.

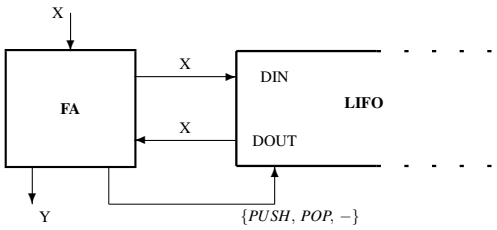


Figure 6.5: **The push-down automaton (*PDA*).** A finite (random) automaton loop-coupled with an "infinite" stack (a simple automaton) is an enhanced toll for dealing with formal languages.

Definition 6.1 *The* push-down automaton*, PDA, (see Figure 6.5) built by a finite automaton loop connected with a push-down stack (LIFO), is defined by the six-tuple:*

$$PDA = (X \times X', Y \times Y' \times X, Q, f, g, z_0)$$

where:

X *: is the finite alphabet of the machine; the input string is in* X^*

X' *: is the finite alphabet of the stack,* $X' = X' \cup \{z_0\}$

Y *: is the finite output set of the machine*

Y' *: is the set of commands issued by the finite automaton toward LIFO,* $\{PUSH, POP, -\}$

Q *: is the finite set of the automaton states (i.e.,* $|Q| \neq h(max\, l(s))$*, where* $s \in X^*$ *is received on the input of the machine)*

f *: is the state transition function of the* machine

$$f : X \times X' \times Q \to Q \times X \times Y'$$

(i.e., depending on the received symbol, by the value of the top of stack (TOS) and by the automaton's state, the automaton switches in a new state, a new value can be sent to the stack and the stack receives a new command (PUSH, POP or NOP))

g *: is the output transition function of the* automaton - $g : Q \to Y$

z_0 *: is the initial value of TOS.* $\diamond$

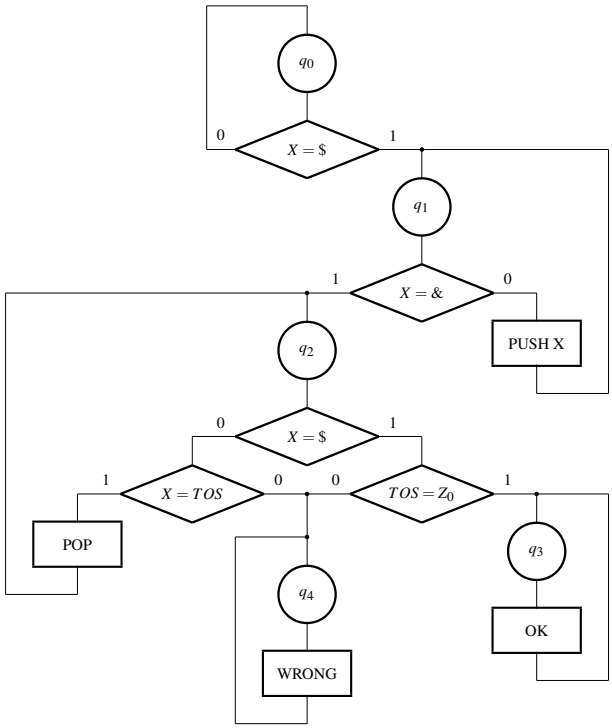


Figure 6.6: **Defining the behavior of a PDA.** The algorithm detecting the antisymmetrical sequences of symbols.

Example 6.3 *The problem to be solved is designing a machine that recognizes strings having the form* $\$x\&y\$$*, where* $\$, \& \in X$ *and* $x, y \in X^*$*, X being a finite alphabet and y is the antisymmetric version of x.*

The solution is to use a PDA with f and g described by the flow-chart given in Figure 6.6. Results a five state, initial (in q_0*) automaton, each state having the following meaning and role:*

q_0 *: is the initial state in which the machine is waiting for the first* \$

q_1 *: in this state the received symbols are pushed into the stack, excepting* & *that switches the automaton in the next state*

q_2 : *in this state, each received symbol is compared with TOS, that is poped on, while the received symbol is not* \$*; when the input is* \$ *and* $TOS = z_0$ *the automaton switches in* q_3*, else, if the received symbols do not correspond with the successive value of the TOS or the final value of TOS differs from* z_0*, the automaton switches in* q_4

q_3 : *if the automaton is in this state the received string was recognized as a well formed string*

q_4 : *if the automaton is in this state the received string was wrong.* $\diamond$

The reader can try to solve the problem using only an automaton. For a given X set, especially for a small set, the solution is possible and small, but the LOOP PLA of the resulting automaton will be a circuit with the size and the form depending by the dimension and by the content of the set X. If only one symbol is added or at least is changed, then the entire design process must be restarted from scratch. The automaton imposes a solution in which the simple, recursive part of the solution is mixed up with the random part, thus all the system has a very large apparent complexity. The automaton must store in the state space what PDA stores in stack. You imagine how huge become the state set in a such crazy solution. Both, the size and the complexity of the solution become unacceptable.

The solution with PDA, just presented, does not depend by the content and by the dimension of the set X. In this solution the simple is well segregated from the complex. The simple part is the "infinite" stack and the complex part is a small, five-state finite automaton.

6.3.2 An interpreting processor

The interpreting processor are known also as processors having a Complex Instruction Set Computer (CISC) architecture, or simply as CISC Processors. The interpreting approach allows us to design complex instructions which are transformed at the hardware level in a sequence of operations. Lets remember that an executing (RISC) processor has almost all instructions implemented in one clock cycle. It is not decided what style of designing an architecture is the best. Depending on the application sometimes a RISC approach is mode efficient, sometimes a CISC approach is preferred.

The organization

Our CISC Processor is a machine characterized by using a register file to store the internal (the most frequently used) variables. The top level view of this version of processor is represented in Figure 6.7. It contains the following blocks:

- REGISTER & ALU – RALU – 32 32-bit registers organized in a register file, and an ALU; the registers are used also for control purposes (program counter, return address, stack pointer in the external memory, ...)
- INPUT & OUTPUT BUFFER REGISTERS used to provide full synchronous connections with the external "world", minimizing t_{in_reg}, t_{reg_out}, maximizing f_{max}, and avoiding t_{in_out} (see subsection 1.1.5); the registers are the following:

  COM REG : sends out the 2-bit read or write command for the external data & program memory

  ADDR REG : sends out the 32-bit address for the external data & program memory

  OUT REG : sends out the 32-bit data for the external memory

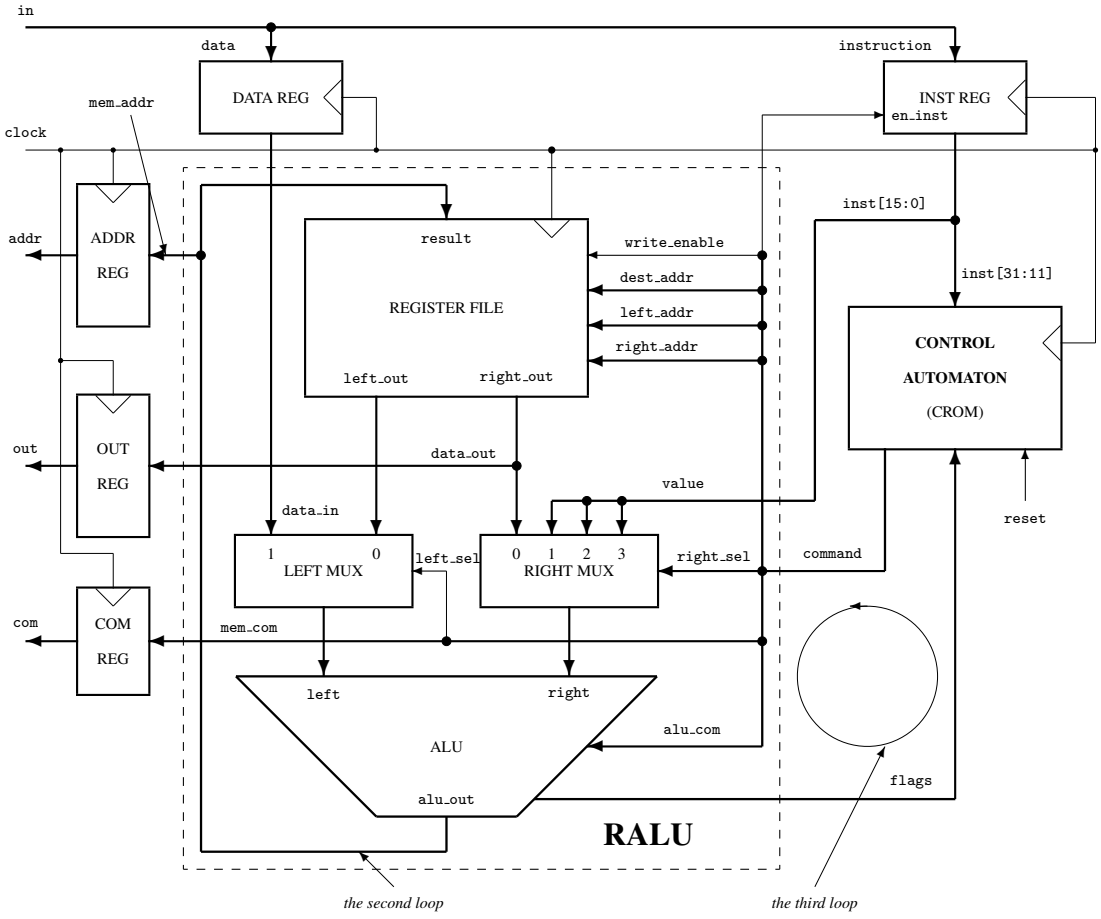


Figure 6.7: **An interpreting processor.** The organization is simpler because only one external memory is used.

DATA REG : receives back, with one clock cycle delay related to the command loaded in COM REG, 32-bit **data** from the external data & program memory

INST REG : receives back, with one clock cycle delay related to the command loaded in COM REG, 32-bit **instruction** from the external data & program memory

- CONTROL AUTOMATON used to control the fetch and the interpretation of the instructions stored in the external memory; it is an initial automaton initialized, for each new instruction, by the operation code (`inst[31:26]` received from INST REG)

The instruction of our CISC Processor has two formats. The first format is:

```
{   opcode[5:0]     ,    // operation code
    dest_addr[4:0]  ,    // selects the destination
    left_addr[4:0]  ,    // selects the left operand
    right_addr[4:0] ,    // selects the right operand
```

```
        rel_addr[10:0]  }    // small signed jump for program address
        = instr[31:0];
```

The relative address allows a positive or a negative jump of 1023 instructions in the program space. It is sufficient for almost all jumps in a program. If not, special absolute jump instruction can solve this very rare cases.

The second format is used when the right operand is a constant `value` generated at the compiling time in the instruction body. It is:

```
 {   opcode[5:0]     ,
     dest_addr[4:0]  ,
     left_addr[4:0]  ,
     value[15:0]     }     // signed integer
     = instr[31:0];
```

When the instruction is fetched from the external memory it is memorized in INST REG because its content will be used in *different* stages of the interpretation, as follows:

- `inst[31:26] = opcode[5:0]` to initialize CONTROL AUTOMATON in the state from which flows the sequence of commands used to interpret the current instruction
- `inst[29:26] = opcode[3:0]` to command the function performed by ALU in the step associated to perform the main operation associated with the current instruction (for example, if the instruction is `add 12, 3, 7`, then the bits `opcode[3:0]` are used to command the ALU to do the addition of registers 3 and 7 in the appropriate step of interpretation)
- `inst[25:11] =` {`dest_addr, left_addr, right_addr`} is used to address the REGISTER FILE unit when the main operation associated with the current instruction is performed
- `inst[15:0] = value` is selected to form the right operand when an instruction operating with immediate value is interpreted
- `inst[10:0] = rel_addr` is used in jump instructions, in the appropriate clock cycle, to compute the next program address.

The REGISTER FILE unit contains 32 32-bit registers. In each clock cycle, any ordered pair of registers can be selected as operands, and the result can be stored back in any of them. They have the following use:

- `r0, r1, ... r29` are general purpose registers;
- `r31` is used as **Program Counter** (PC);
- `r30` is used to store the *Return Address* (RA) when the call instruction is interpreted (no embedded calls are allowed for this simple processor[1]).

CONTROL AUTOMATON has the structure presented in Figure 6.8. In the fetch cycle `init = 1` allows the automaton to jump into the state codded by `opcode`, from which a sequence of operations

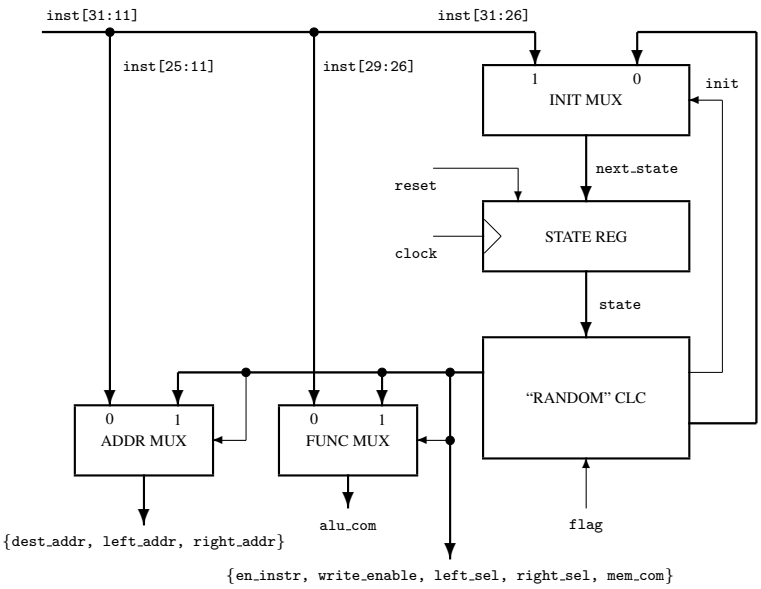


Figure 6.8: **The control automaton for our CISC Processor.** It is a more compact version of CROM (see Figure 5.14). Instead of a CLC used for the complex part of executing processor, for an interpreting processor a sequential machine is used to solve the problem of complexity.

flows with `init = 0`, ignoring the initialization input. This is the simplest way to associate for each instruction the interpreting sequence of elementary operation.

The output of CONTROL AUTOMATON commands all the top level blocks of our CISC Processor using the following fields:

```
{en_inst        ,  // write enable for the instruction register
 write_enable   ,  // write back enable for the register file
 dest_addr[4:0] ,  // destination address in file register
 left_addr[4:0] ,  // left operand address in file register
 alu_com[3:0]   ,  // alu functions
 right_addr[4:0],  // right operand address in file register
 left_sel       ,  // left operand selection
 right_sel[1:0] ,  // right operand selection
 mem_com[1:0]   }  // memory command
 = command
```

The fields `dest_addr, left_addr, right_addr, alu_com` are sometimes selected from INST REG (see ADDR MUX and FUNC MUX in Figure 6.8) and sometimes their value is generated by CONTROL AUTOMATON according to the operation to be executed in the current clock cycle. The other command fields are generated by CONTROL AUTOMATON in each clock cycle.

[1] If embedded calls are needed, then this register contains the stack pointer into a stack organized in the external memory. We are not interested in adding the feature of embedded calls, because in this digital system lessons we intend to keep the examples small and simple.

```
module cisc_processor(input                clock   ,
                      input                reset   ,
                      output    reg [31:0] addr_reg,   // memory address
                      output    reg [1:0]  com_reg ,   // memory command
                      output    reg [31:0] out_reg ,   // data output
                      input         [31:0] in      );  // data/inst input
 // INTERNAL CONNECTIONS
    wire    [25:0]  command;
    wire            flag;
    wire    [31:0]  alu_out, left, right, left_out, right_out;
 // INPUT & OUTPUT BUFFER REGISTERS
    reg [31:0]  data_reg, inst_reg;
    always @(posedge clock) begin   if (command[25]) inst_reg <= in ;
                                    data_reg <= in                  ;
                                    addr_reg <= left_out            ;
                                    out_reg  <= right_out           ;
                                    com_reg  <= command[1:0]        ;   end
 // CONTROL AUTOMATON
    control_automaton control_automaton(.clock  (clock          ),
                                        .reset  (reset          ),
                                        .inst   (inst_reg[31:11]),
                                        .command(command        ),
                                        .flag   (alu_out[0]     ));
 // REGISTER FILE
    register_file register_file(.left_out       (left_out       ),
                                .right_out      (right_out      ),
                                .result         (alu_out        ),
                                .left_addr      (command[18:14] ),
                                .right_addr     (command[9:5]   ),
                                .dest_addr      (command[23:19] ),
                                .write_enable   (command[24]    ),
                                .clock          (clock          ));
 // MULTIPLEXERS
    mux2 left_mux(  .out(left       ),
                    .in0(left_out   ),
                    .in1(data_reg   ),
                    .sel(command[4]));
    mux4 right_mux( .out(right                                  ),
                    .in0(right_out                              ),
                    .in1({{21{inst_reg[10]}}, inst_reg[10:0]}   ),
                    .in2({16'b0, inst_reg[15:0]}                ),
                    .in3({inst_reg[15:0], 16'b0}                ),
                    .sel(command[3:2]                           ));
 // ARITHMETIC & LOGIC UNIT
    cisc_alu alu(   .alu_out(alu_out        ),
                    .left   (left           ),
                    .right  (right          ),
                    .alu_com(command[13:10] ));
endmodule
```

Figure 6.9: **The top module of our CISC Processor.**

CONTROL AUTOMATON receives back from ALU only one flag: the least significant bit of ALU, `alu_out[0]`; thus closing the third loop[2].

In each clock cycle the content of two registers can be operated in ALU and the result stored in a third register.

The left operand can be sometimes `data_in` if `left_sel = 1`. It must be addressed two clock cycles before use, because the external memory is supposed to be a synchronous one, and the input register introduces another one cycle delay. The sequence generated by CONTROL AUTOMATON takes care by this synchronization.

The right operand can be sometimes `value = instr_reg[15:0]` if `right_sel = 2'b1x`. If `right_sel = 2'b01` the right operand is the 11-bit signed integer `rel_addr = instr_reg[10:0]`

The external memory is addressed with a delay of one clock cycle using the value of `left_out`. We are not very happy about this additional delay, but this is the price for a robust design. What we loose in number of clock cycles used to perform some instructions is, at least partially, recuperated by the possibility to increase the frequency of the system clock.

Data to be written in the external memory is loaded into OUT REG from the right output of FILE REG. It is synchronous with the address.

The command for the external memory is also delayed one cycle by the synchronization register COM REG. It is generated by CONTROL AUTOMATON.

Data and instructions are received back from the external memory with two clock cycle delay, one because of we have an external synchronous memory, and another because of the input re-synchronization done by DATA REG and INST REG.

The structural Verilog description of the top level of our CISC Processor is in Figure 6.9.

Microarchitecture

The complex part of our CISC Processor is located in the block called CONTROL AUTOMATON. More precisely, the only complex circuit in this design is the loop of the automaton called "RANDOM CLC" (see Figure 6.8). The Verilog module describing CONTROL AUTOMATON is represented in Figure 6.10.

The micro-architecture defines all the fields used to command the simple parts of this processor. Some of them are used inside the `control_automaton` module, while others command the top modules of the processor.

The inside used fields command are the following:

`init` : allows the jump of the automaton into the initial state associated with each instruction when `init = new_seq`

`addr_sel` : the three 5-bit addresses for FILE REGISTER are considered only if the field `addr_sel` takes the value `from_inst`, else three special combinations of addresses are generated by the control automaton

`func_sel` : the field `alu_com` is considered only if the field `func_sel` takes the value `from_out`, else the code `opcode[3:0]` selects the ALU's function

The rest of fields command the function performed in each clock cycle by the top modules of our CISC Processor. They are:

[2]The second loop is closed once in the big & simple automaton RALU, and another in the complex finite automaton CONTROL AUTOMATON. The first loop is closed in each flip-flop used to build the registers.

`en_inst` : enables the load of data received from the external memory only when it represents the next instruction to be interpreted

`write_enable` : enables write back into FILE REGISTER the result from the output of ALU

`alu_com` : is a 4-bit field used to command ALU's function for the specific purpose of the interpretation process (it is considered only if `func_sel = from_aut`)

`left_sel` : is the selection code for LEFT MUX (see Figure 6.7)

`right_sel` : is the selection code for RIGHT MUX (see Figure 6.7)

`mem_com` : generated the commands for the external memory containing both data and programs.

The micro-architecture (see Figure 6.11) is subject of possible changes during the definition of the transition function of CONTROL AUTOMATON.

Instruction set architecture (ISA)

There is a big flexibility in defining the ISA for a CISC machine, because we accepted to interpret each instruction using a sequence of micro-operations. The control automaton is used as sequencer for implementing instructions beyond what can be simply envisaged inspecting the organization of our simple CISC processor.

An executing (RISC) processor displays its architecture in its organization, because the control is very simple (the decoder is a combinational circuit used to *trans-code* only). The complexity of the control of an interpreting processor hides the architecture in the complex definition of the control automaton (which can have a strong generative power).

In Figure 6.12 is sketched a possible instruction set for our CISC processor. There are at least the following classes of instructions:

- Arithmetic & Logic Instructions: the destination register takes the value resulted from operating any two registers (unary operations, such as increment, are also allowed)
- Data Move Instructions: data exchange between the external memory and the file register are performed
- Control Instructions: the flow of instruction is controlled according to the fix or data dependent patterns
- ...

We limit our discussion to few and small classes of instructions because our goal is to offer only a structural image about what an interpretative processor is. An exhaustive approach is an architectural one, which is far beyond out intention in these lessons about digital systems, not about computational systems.

```
module control_automaton(   input           clock   ,
                            input           reset   ,
                            input   [20:0]  inst    ,
                            output  [25:0]  command ,
                            input   [3:0]   flags   );

    `include "micro_architecture.v"
    `include "instruction_set_architecture.v"

     // THE STRUCTURE OF 'inst'
    wire    [5:0]   opcode  ;   // operation code
    wire    [4:0]   dest    ,   // selects destination register
                    left_op ,   // selects left operand register
                    right_op;   // selects right operand register
    assign {opcode, dest, left_op, right_op} = inst;

 // THE STRUCTURE OF 'command'
    reg             en_inst     ; // enable load a new instruction in inst_reg
    reg             write_enable; // writes the output of alu at dest_addr
    reg     [4:0]   dest_addr   ; // selects the destination register
    reg     [4:0]   left_addr   ; // selects the left operand in file register
    reg     [3:0]   alu_com     ; // selects the operation performed by the alu
    reg     [4:0]   right_addr  ; // selects the right operand in file register
    reg             left_sel    ; // selects the source of the left operand
    reg     [1:0]   right_sel   ; // selects the source of the right operand
    reg     [1:0]   mem_com     ; // generates the command for memory
    assign command = {en_inst, write_enable, dest_addr, left_addr, alu_com,
                        right_addr, left_sel, right_sel, mem_com};

    // THE STATE REGISTER
    reg     [5:0]   state_reg   ; // the state register
    reg     [5:0]   next_state  ; // a "register" used as variable
    always @(posedge clock) if (reset)  state_reg <= 0          ;
                            else        state_reg <= next_state ;

    `include "the_control_automaton's_loop.v"
endmodule
```

Figure 6.10: **Verilog code for** `control_automaton`**.**

```
// MICRO-ARCHITECTURE
   // en_inst
   parameter   no_load     = 1'b0, // disable instruction register
               load_inst   = 1'b1; // enable instruction register
   // write_enable
   parameter   no_write    = 1'b0,
               write_back  = 1'b1; // write back the current ALU output
   // alu_func
   parameter   alu_left    = 4'b0000, // alu_out = left
               alu_right   = 4'b0001, // alu_out = right
               alu_inc     = 4'b0010, // alu_out = left + 1
               alu_dec     = 4'b0011, // alu_out = left - 1
               alu_add     = 4'b0100, // alu_out = left + right
               alu_sub     = 4'b0101, // alu_out = left - right
               alu_shl     = 4'b0110, // alu_out = {1'b0, left[31:1]}
               alu_half    = 4'b0111, // alu_out = {left[31], left[31:1]}
               alu_zero    = 4'b1000, // alu_out = {31'b0, (left == 0)}
               alu_equal   = 4'b1001, // alu_out = {31'b0, (left == right)}
               alu_less    = 4'b1010, // alu_out = {31'b0, (left < right)}
               alu_carry   = 4'b1011, // alu_out = {31'b0, add[32]}
               alu_borrow  = 4'b1100, // alu_out = {31'b0, sub[32]}
               alu_and     = 4'b1101, // alu_out = left & right
               alu_or      = 4'b1110, // alu_out = left | right
               alu_xor     = 4'b1111, // alu_out = left ^ right
   // left_sel
   parameter   left_out    = 1'b0, // left out of the reg file as left op
               from_mem    = 1'b1; // data from memory as left op
   // right_sel
   parameter   right_out   = 2'b00, // right out of the reg file as right op
               jmp_addr    = 2'b01, // right op = {{22{inst[10]}}, inst[9:0]}
               low_value   = 2'b10, // right op = {{16{inst[15]}}, inst[15:0]}
               high_value  = 2'b11; // right op = {inst[15:0], 16'b0}
   // mem_com
   parameter   mem_nop     = 2'b00,
               read        = 2'b10, // read from memory
               write       = 2'b11; // write to memory
```

Figure 6.11: **The micro-architecture of our CISC Processor.**

```
// INSTRUCTION SET ARCHITECTURE (only samples)
   // arithmetic & logic instructions & pc = pc + 1
   parameter
   move    = 6'b10_0000, // dest_reg = left_out
   inc     = 6'b10_0010, // dest_reg = left_out + 1
   dec     = 6'b10_0011, // dest_reg = left_out - 1
   add     = 6'b10_0100, // dest_reg = left_out + right_out
   sub     = 6'b10_0101, // dest_reg = left_out - right_out
   bwxor   = 6'b10_1111; // dest_reg = left_out ^ right_out
   // ...
   // data move instructions & pc = pc + 1
   parameter
   read    = 6'b01_0000, // dest_reg = mem(left_out)
   rdinc   = 6'b01_0001, // dest_reg = mem(left_out + value)
   write   = 6'b01_1000, // mem(left_out) = right_out
   wrinc   = 6'b01_1001; // mem(left_out + value) = right_out
   // ...
   // control instructions
   parameter
   nop     = 6'b11_0000, // pc = pc + 1
   jmp     = 6'b11_0001, // pc = pc + value
   call    = 6'b11_0010, // pc = value, ra = pc + 1
   ret     = 6'b11_0011, // pc = ra
   jzero   = 6'b11_0100, // if (left_out = 0) pc = pc + value;
                         // else pc = pc + 1
   jnzero  = 6'b11_0101; // if (left_out != 0) pc = pc + value;
                         // else pc = pc + 1
   // ...
```

Figure 6.12: **The instruction set architecture of our CISC Processor.** The partial definition of the file `instruction_set_architecture.v` included in the `conttrol_automaton.v` file.

Implementing ISA

Implementing a certain Instruction Set Architecture means to define the transition functions of the control automaton:

- the output transition function, in our case to specify for each state the value of the `command` code (see Figure 6.10)
- the state transition function, which specifies the value of `next_state`

The content of the file `the_control_automaton's_loop.v` contains the description of the combinational circuit associated to the control automaton. It generates both the 26-bit `command` code and the 6-bit `next_state` code. The following Verilog code is the most compact way to explain how the control automaton works. Please read the next "`always`" as the single way to explain rigorously how out CISC machine works.

```
// THE CONTROL AUTOMATON'S LOOP
   always @(state_reg or opcode or dest or left_op or right_op or flag)
       begin   en_inst      = 1'bx     ;
```

```
    write_enable = 1'bx     ;
    dest_addr    = 5'bxxxxx ;
    left_addr    = 5'bxxxxx ;
    alu_com      = 4'bxxxx  ;
    right_addr   = 5'bxxxxx ;
    left_sel     = 1'bx     ;
    right_sel    = 2'bxx    ;
    mem_com      = 2'bxx    ;
    next_state   = 6'bxxxxxx;
// INITIALIZE THE PROCESSOR
if (state_reg == 6'b00_0000)
// pc = 0
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = 5'b11111     ;
            left_addr    = 5'b11111     ;
            alu_com      = alu_xor      ;
            right_addr   = 5'b11111     ;
            left_sel     = left_out     ;
            right_sel    = right_out    ;
            mem_com      = mem_nop      ;
            next_state   = state_reg + 1;
        end
// INSTRUCTION FETCH
if (state_reg == 6'b00_0001)
// rquest for a new instruction & increment pc
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = 5'b11111     ;
            left_addr    = 5'b11111     ;
            alu_com      = alu_inc      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = left_out     ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_read     ;
            next_state   = state_reg + 1;
        end
if (state_reg == 6'b00_0010)
// wait for memory to read doing nothing (synchronous memory)
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = 4'bxxxx      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = 1'bx         ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = state_reg + 1;
```

```
        end
if (state_reg == 6'b00_0011)
// load the new instruction in instr_reg
        begin
            en_inst      = load_inst    ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = 4'bxxxx      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = 1'bx         ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = state_reg + 1;
        end
if (state_reg == 6'b00_0100)
// initialize the control automaton
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = 4'bxxxx      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = 1'bx         ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = opcode[5:0]  ;
        end
// EXECUTE THE ONE CYCLE FUNCTIONAL INSTRUCTIONS
if (state_reg[5:4] == 2'b10)
// dest = left_op OPERATION right_op
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = dest         ;
            left_addr    = left_op      ;
            alu_com      = opcode[3:0]  ;
            right_addr   = right_op     ;
            left_sel     = left_out     ;
            right_sel    = right_out    ;
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
// EXECUTE MEMORY READ INSTRUCTIONS
if (state_reg == 6'b01_0000)
// read from left_reg in dest_reg
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = left_op      ;
```

```
            alu_com      = alu_left     ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = left_out     ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_read     ;
            next_state   = 6'b01_0010   ;
        end
    if (state_reg == 6'b01_0001)
    // read from left_reg + <value> in dest_reg
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = left_op      ;
            alu_com      = alu_add      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = left_out     ;
            right_sel    = low_value    ;
            mem_com      = mem_read     ;
            next_state   = 6'b01_0010   ;
        end
    if (state_reg == 6'b01_0010)
    // wait for memory to read doing nothing
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = 4'bxxxx      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = 1'bx         ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = state_reg + 1;
        end
    if (state_reg == 6'b01_0011)
    // the data from memory is loaded in data_reg
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = 4'bxxxx      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = 1'bx         ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = state_reg + 1;
        end
    if (state_reg == 6'b01_0100)
    // data_reg is loaded in dest_reg & go to fetch
        begin
```

```
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = dest         ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = alu_left     ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = from_mem     ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
// EXECUTE MEMORY WRITE INSTRUCTIONS
if (state_reg == 6'b01_1000)
// write right_op to left_op & go to fetch
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = left_op      ;
            alu_com      = alu_left     ;
            right_addr   = right_op     ;
            left_sel     = left_out     ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_write    ;
            next_state   = 6'b00_0001   ;
        end
if (state_reg == 6'b01_1000)
// write right_op to left_op + <value>  & go to fetch
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = left_op      ;
            alu_com      = alu_add      ;
            right_addr   = right_op     ;
            left_sel     = left_out     ;
            right_sel    = low_value    ;
            mem_com      = mem_write    ;
            next_state   = 6'b00_0001   ;
        end
// CONTROL INSTRUCTIONS
if (state_reg == 6'b11_0000)
// no operation & go to fetch
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = 4'bxxxx      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = 1'bx         ;
            right_sel    = 2'bxx        ;
```

```
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
    if (state_reg == 6'b11_0001)
    // jump to (pc + <value>) & go to fetch
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = 5'b11111     ;
            left_addr    = 5'b11111     ;
            alu_com      = alu_add      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = left_out     ;
            right_sel    = low_value    ;
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
    if (state_reg == 6'b11_0010)
    // call: first step: ra = pc + 1
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = 5'b11110     ;
            left_addr    = 5'b11111     ;
            alu_com      = alu_left     ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = left_out     ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = 6'b11_0110;
        end
    if (state_reg == 8'b0011_0110)
    // call: second step: pc = value
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = 5'b11111     ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = alu_right    ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = 1'bx         ;
            right_sel    = jmp_addr     ;
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
    if (state_reg == 6'b11_0011)
    // ret: pc = ra
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = 5'b11111     ;
            left_addr    = 5'b11110     ;
```

```
            alu_com      = alu_left     ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = left_out     ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
if ((state_reg == 6'b11_0100) && flag)
// jzero: if (left_out = 0) pc = pc + value;
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = 5'b11111     ;
            left_addr    = 5'b11111     ;
            alu_com      = alu_add      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = left_out     ;
            right_sel    = low_value    ;
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
if ((state_reg == 6'b11_0100) && ~flag)
// jzero: if (left_out = 1) pc = pc + 1;
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = 4'bxxxx      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = 1'bx         ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
if ((state_reg == 6'b11_0100) && ~flag)
// jnzero: if (left_out = 1) pc = pc + value;
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = 5'b11111     ;
            left_addr    = 5'b11111     ;
            alu_com      = alu_add      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = left_out     ;
            right_sel    = low_value    ;
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
if ((state_reg == 6'b11_0100) && flag)
// jnzero: if (left_out = 0) pc = pc + 1;
        begin
```

```
                    en_inst      = no_load      ;
                    write_enable = no_write     ;
                    dest_addr    = 5'bxxxxx     ;
                    left_addr    = 5'bxxxxx     ;
                    alu_com      = 4'bxxxx      ;
                    right_addr   = 5'bxxxxx     ;
                    left_sel     = 1'bx         ;
                    right_sel    = 2'bxx        ;
                    mem_com      = mem_nop      ;
                    next_state   = 6'b00_0001   ;
                end
        end
```

The automaton described by the previous code has 36 states for the 25 instructions implemented (see Figure 6.12). More instructions can added if new state are described in the previous "`always`". Obviously, the most complex part of the processor is this combinational circuit associated to the control automaton.

Time performance

The representation from Figure 6.13 is used to evaluate the time restrictions imposed by our CISC processor.

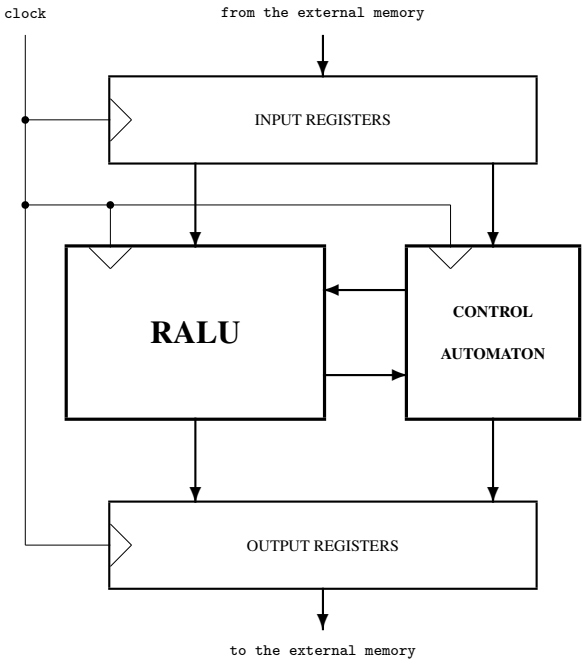


Figure 6.13: **The simple block diagram of our CISC processor.** The fully buffered solution imposed for designing this interpretative processor minimizes the depth of signal path entering and emerging in/from the circuit, and avoid a going through combinational path.

The full registered external connections of the circuit allows us to provide the smallest possible values for *minimum input arrival time before clock*, t_{in_reg}, *maximum output required time after clock*, t_{reg_out}, and no path for *maximum combinational path delay*, t_{in_out}. The maximum clock frequency is fully determined by the internal structure of the processor, by the path on the loop closed inside RALU or between RALU and CONTROL AUTOMATON. The actual time characterization is:

- $t_{in\_reg} = t_{su}$ – the set-up time for the input registers
- $t_{reg\_out} = t_{reg}$ – the propagation time for the output registers
- $T_{min} = max(t_{RALU\_loop}, t_{processor\_loop})$, where:

$$t_{RALU\_loop} = t_{state\_reg} + t_{aut\_out\_clc} + t_{reg\_file} + t_{mux} + t_{alu} + t_{reg\_file\_su}$$

$$t_{processor\_loop} = t_{state\_reg} + t_{aut\_out\_clc} + t_{reg\_file} + t_{mux} + t_{alu\_flag} + t_{aut\_in\_clc} + t_{state\_reg\_su}$$

This well packed version of a simple processor is very well characterized as time behavior. The price for this is the increasing number of clock cycle used for executing an instruction. The effect of the increased number of clock cycles is sometimes compensated by the possibility to use a higher clock frequency. But, all the time the modularity is the main benefit.

Concluding about our CISC processor

A CISC processor is more complex than a stack processor because for each instruction the operands must be selected from the file register. The architecture is more flexible, but the loop closed in RALU is longer than the loop closed in SALU.

A CISC approach allows more complex operations performed during an instruction because it is interpreted, not simply executed in one clock cycle.

Interpretation allows a single memory for both data and programs with all the resulting advantages and disadvantages.

An interpreting processor contains a simple automaton – RALU – and a complex one – Control Automaton – because its complex behavior.

Both, Stack Processor and CISC Processor are only simple exercises designed for presenting the circuit aspects of the closing of hte third loop. The real and complex architectural aspects are minimally presented because this text book is about circuits not abut computation.

6.4 The assembly language: the lowest programming level

The instruction set represent the machine language: the lowest programming level in a computation machine. The programming is very difficult at this level because of the concreteness of the process. Too many details must be known by the programmer. The main improvement added by a higher level language is the level of abstraction used to present the computational resources. Writing a program in machine language we must have in mind a lot of physical details of the machine. Therefore, a real application must be developed in a higher level language.

The machine language can be used only for some very critical section of the algorithms. The automatic translation done by a compiler from a high level language into the machine language is some times unsatisfactory for high performance application. Only in this cases small part of the code must be generated “manually” using the machine language.

6.5 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.

6.6 Problems

Problem 6.1 *Interrupt automaton with asynchronous input.*

Problem 6.2 *Solving the second degree equations with an elementary processor.*

Problem 6.3 *Compute y if x, m and n is given with an elementary processor..*

Problem 6.4 *Modify the unending loop of the processor to avoid spending time in testing if a new instruction is in inFIFO when it is there.*

Problem 6.5 *Define an instruction set for the processor described in this chapter using its microarchitecture.*

Problem 6.6 *Is it closed another loop in our Stack Processor connecting* `tos` *to the input of DECODE unit?*

Problem 6.7 *Our CISC Processor: how must be codded the instruction set to avoid FUNC MUX?*

6.7 Projects

Project 6.1 *Design a specialized elementary processor for rasterization function.*

Project 6.2 *Design a system integrating in a parallel computational structure 8 rasterization processors designed in the previous project.*

Project 6.3 *Design a floating point arithmetic coprocessor.*

Project 6.4 *Design the RISC processor defined by the following Verilog behavioral description:*

```
module risc_processor(
                          );

endmodule
```

Project 6.5 *Design a version of Stack Processor modifying SALU as follows: move MUX4 to the output of ALU and the input of STACK.*

Chapter 7

COMPUTING MACHINES: $\geq$4–loop digital systems

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*.

7.1 Types of fourth order systems

There are four main types of fourth order systems (see Figure 7.1) depending on the order of the system through which the loop is closed:

1. **P & ROM** is a 4-OS with loop closed through a 0-OS - in Figure 7.1a the combinational circuit is a ROM containing only the programs executed or interpreted by the processor
2. **P & RAM** is a 4-OS with loop closed through a 1-OS - is the **computer**, the most representative structure in this order, having on the loop a RAM (see Figure 7.1b) that stores both data and programs
3. **P & LIFO** is a 4-OS with loop closed through a 2-OS - in Figure 7.1c the automaton is represented by a push-down stack containing, by the rule, data (or sequences in which the distinction between data and programs does not make sense, as in the Lisp programming language, for example)
4. **P & CO-P** is a 4-OS with loop closed through a 3-OS - in Figure 7.1d COPROCESSOR is also a processor but a specialized one executing efficiently critical functions in the system (in most of cases the coprocessor is a floating point arithmetic processor).

The representative system in the class of **P & ROM** is the *microcontroller* the most successful circuit in 4-OS. The microcontroller is a “best seller” circuit realized as a one-chip computer. The core of a microcontroller is a processor executing/interpreting the programs stored in a ROM.

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.

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

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

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

- the machine language (usually called *architecture*)
- the storage containing programs written in the machine language
- the **machine** that *interprets* the programs, containing:
  - the machine language ...
  - the storage ...
  - the **machine** ... containing:
    * ...

and so on until the **machine** *executes* the programs.

Does it make any sense to add new loops? Yes, but not too much! It can be justified to add loops inside the processor structure to improve its capacity to interpret fast the machine language by using simple circuits. Another way is to see PROCESSOR & COPROCESSOR or PROCESSOR & LIFO as performant processors and to add over them the loop through RAM. But, mainly these machines remain structures having the computer function. The computer needs at least four loops to be *competent*, but currently it is implemented on system having more loops in order to become *performant*.

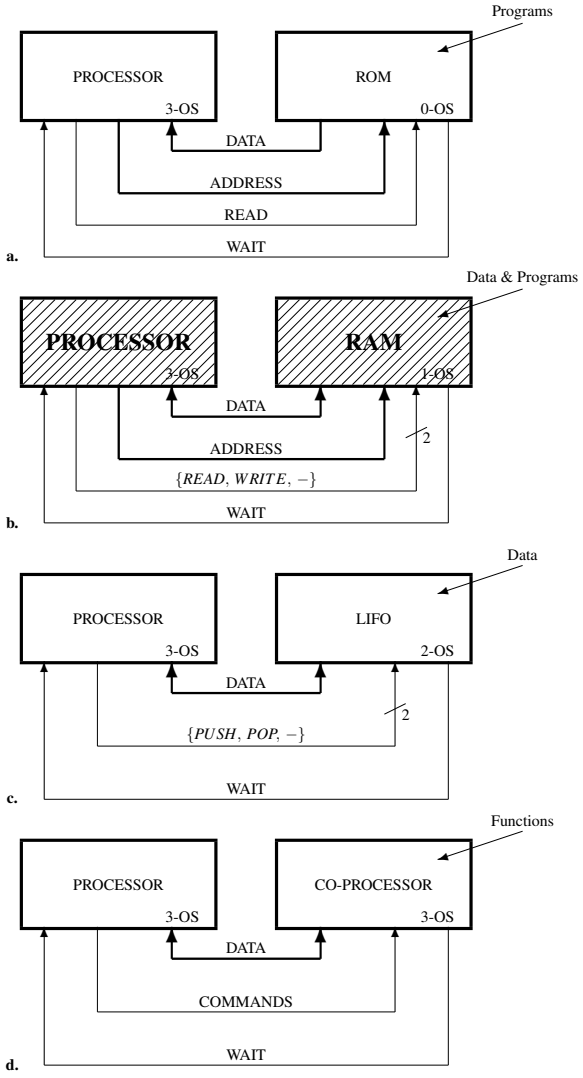


Figure 7.1: **The four types of 4-OS machines. a.** Fix program computers usual in embedded computation. **b.** General purpose computer. **c.** Specialized computer working working on a restricted data structure. **d.** Accelerated computation supported by a specialized co-processor.

7.2 The stack processor – a processor as 4-OS

The best way to explain how to use the concept of architecture to design an executive processor is to use an example having an appropriate complexity. One of the simplest model of computing machine is the stack machine. A stack machine finds always its operands in the first two stages of a stack (LIFO) memory. The last two pushed data are the operands involved in the current operation. The computation must be managed to have accessible the current operand(s) in the data stack. The stack used in a stack processor have some additional features allowing an efficient data management. For example: double pop, swap, ...

The high level description of a stack processor follows. The purpose of this description is to offer an example of how starts the design of a processor. Once the functionality of the machine is established at the higher level of the architecture, there are many ways to implement it.

7.2.1 The organization

Our **Stack Processor** is a sort of simple processing element characterized by using a stack memory (LIFO) for storing the internal variables. The top level internal organization of a version of Stack Processor (see Figure 7.2) contains the following blocks:

- STACK & ALU – SALU – is the unit performing the elementary computations; it contains:
  - a two-output stack; the top of stack (`stack0` or `tos`) and the previous recording (`stack1`) are accessible
  - an ALU with the operands from the top of stack (`left_op = stack0` and `right_io = stack1`)
  - a selector for the input of stack grabbing data from: (0) the output of ALU, (1) external data memory, (2) the value provided by the instruction, or (3) the value of `pc +1` to be used as return address
- PROGRAM FETCH – a unit used to generate in each clock cycle a new address for fetching from the external program memory the next instruction to be executed
- DECODER – is a combinational circuit used to trans-code the operation code – `op_code` – into commands executed by each internal block or sub-block.

Figure 7.3 represents the Verilog top module for our Stack Processor (`stack_processor`).

The two loop connected automata are SALU and PROGRAM FETCH. Both are simple, recursive defined structures. The complexity of the Stack Processor is given by the DECODE unit: a combinational circuit used to trans-code `op_code` providing 5 small command words to specify how behaves each component of the system. The Verilog `decode` module uses `test_in = tos` and `mem_ready` to make decisions. The value of `tos` can be tested (if it is zero or not, for example) to decide a conditional jump in program (on this way only PROGRAM FETCH module is affected). The `mem_ready` input received from data memory allows the processor to adapt itself to external memories having different access time.

The external data and program memories are both synchronous: the content addressed in the current clock cycle is received back in the next clock cycle. Therefore, `instruction` received in each clock cycle corresponds to `instr_addr` generated in the previous cycle. Thus, the fetch mechanism fits perfect with the behavior of the synchronous memory. For data memory `mem_ready` flag is used to "inform" the `decode` module to delay one clock cycle the use of the data received from the external data memory.

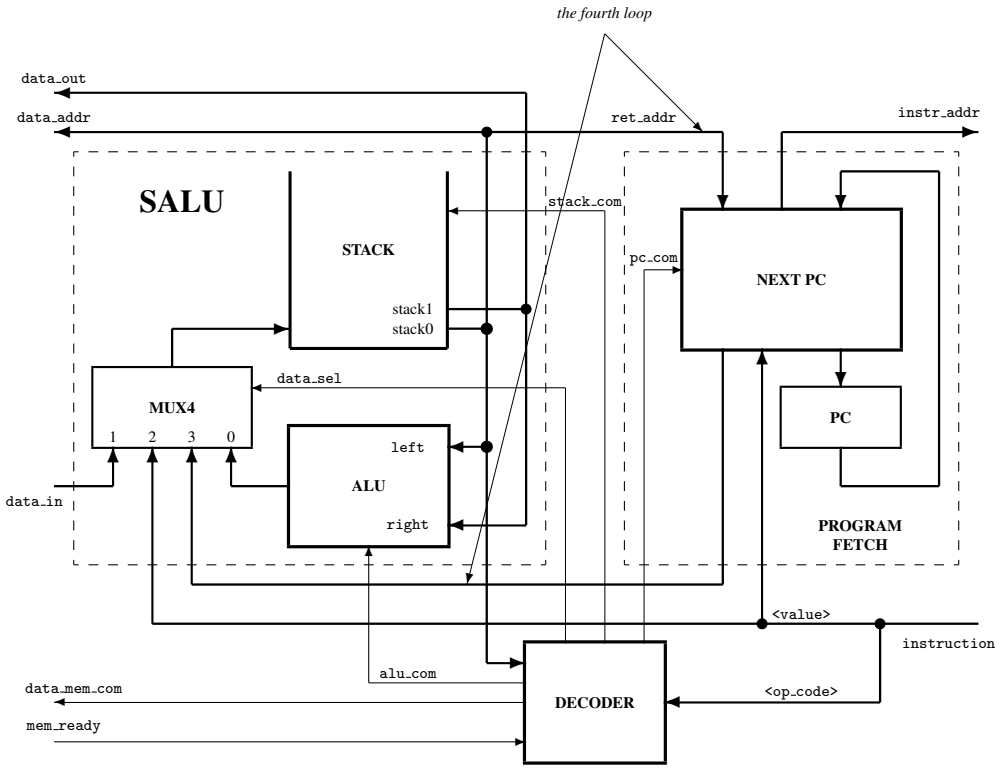


Figure 7.2: **An executing Stack Processor.** Elementary functions are performed by ALU on variables stored in a stack (LIFO) memory. The decoder supports the one-cycle *execution* of the instructions fetched from the external memory.

In each clock cycle ALU unit from SALU receives on its data inputs the two outputs of the stack, and generates the result of the operation selected by the `alu_com` code. If MUX4 has the input 0 selected by the `data_sel` code, then the result is applied to the input of stack. The result is written back in `tos` if a unary operation (increment, for example) is performed (`write` the result of increment in `tos` is equivalent with the sequence `pop`, `increment` & `push`). If a binary operation (addition, for example) is performed, then the first operand is popped from stack and the result is written back over the the new `tos` (`double pop` & `push` involved in a binary operation is equivalent with `pop & write`).

MUX4 selects for the stack input, according to the command `data_sel`, besides the output of ALU, data received back from the external data memory, the value carried by the currently executed instruction, or the value `pc+1` (to be used as return address).

The unit PC generates in each clock cycle the address for program memory. It uses mainly the value from the register PC, which contains the last used address, to fetch an instruction. The content of `tos` or the value contained in the current instruction are also used to compute different conditioned or unconditioned jumps.

To keep this example simple, the program memory is a synchronous one and it contains anytime the addressed instruction (no misses in this memory).

```
module stack_processor(input   clock              ,
                       input   reset              ,
                       output  [31:0]  instr_addr  ,   // instruction address
                       output  [31:0]  data_addr   ,   // data address
                       output  [1:0]   data_mem_com,   // data memory command
                       output  [31:0]  data_out    ,   // data output
                       input   [23:0]  instruction ,   // instruction input
                       input   [31:0]  data_in     ,   // data input
                       input           mem_ready   );  // data memory is ready
// INTERNAL CONNECTIONS
   wire    [2:0]   stack_com   ;   // stack command
   wire    [3:0]   alu_com     ;   // alu command
   wire    [1:0]   data_sel    ;   // data selection for SALU
   wire    [2:0]   pc_com      ;   // program counter command
   wire    [31:0]  tos         ,   // top of stack
                   ret_addr    ;   // return from subroutine address
// INSTRUCTION DECODER UNIT
   decode decode(  .op_code        (instruction[23:16]) ,
                   .test_in        (tos)                ,
                   .mem_ready      (mem_ready)          ,
                   .stack_com      (stack_com)          ,
                   .alu_com        (alu_com)            ,
                   .data_sel       (data_sel)           ,
                   .pc_com         (pc_com)             ,
                   .data_mem_com   (data_mem_com)       );
// SALU: STACK WITH ARITHMETIC & LOGIC UNIT
   salu    salu(   .stack0     (tos)                       ,
                   .stack1     (data_out)                  ,
                   .in1        (data_in)                   ,
                   .in2        ({16'b0, instruction[15:0]}),
                   .in3        (ret_addr)                  ,
                   .s_com      (stack_com)                 ,
                   .data_sel   (data_sel)                  ,
                   .alu_com    (alu_com)                   ,
                   .reset      (reset)                     ,
                   .clock      (clock)                     );
   assign  data_addr = tos;
// PROGRAM COUNTER UNIT
   program_counter pc( .clock      (clock)             ,
                       .reset      (reset)             ,
                       .addr       (instr_addr)        ,
                       .inc_pc     (ret_addr)          ,
                       .value      (instruction[15:0]) ,
                       .tos        (tos)               ,
                       .pc_com     (pc_com)            );
endmodule
```

Figure 7.3: **The top level structural description of a Stack Processor.** The Verilog code associated to the circuit represented in Figure 7.2.

```
module decode( input   [7:0]   op_code     ,
               input   [31:0]  test_in     ,
               input           mem_ready   ,
               output  [2:0]   stack_com   ,
               output  [3:0]   alu_com     ,
               output  [1:0]   data_sel    ,
               output  [2:0]   pc_com      ,
               output  [1:0]   data_mem_com);

   `include "micro_architecture.v"
   `include "instruction_set_architecture.v"
   `include "decoder_implementation.v"
endmodule
```

Figure 7.4: **The** `decode` **module.** It contains the three complex components of the description of Stack Processor.

Because our Stack Processor is designed to be an executing machine, besides the block associated with the *elementary functions* (SALU) and the block used to *compose & and loop* them (PC) there is only a decoder used as *execution unit* (see Figure **??**. The decoder module – `decode` – is the most complex module of Stack Processor (see Figure 7.4). It contains three sections:

- `micro-architecture`: it describes the micro-operations performed by each top level block listing the meaning of all binary codes used to command them
- `instruction set architecture`: describe each instruction performed by Stack Processor
- `decoder implementation`: describe how the micro-architecture is used to implement the instruction set architecture.

7.2.2 The micro-architecture

Any architecture can be implemented using various micro-architectures. For our Stack Processor one of them is presented in Figure 7.5.

The decoder unit generates in each clock cycle a `command` word having the following 5-**field** structure:

$$\{\texttt{alu\_com, data\_sel, stack\_com data\_mem\_com, pc\_com}\} = \texttt{command}$$

where:

- `alu_com`: is a 4-bit code used to select the arithmetic or logic operation performed by ALU in the current cycle; it specifies:
  - well known binary operations such as: `add, subtract, and, or, xor`
  - usual unary operations such as: `increment, shifts`
  - test operations indicating by `alu_out[0]` the result of testing, for example: if an input is zero or if an input is less than another input
- `data_sel`: is a 2-bit code used to select the value applied on the input of the stack for the current cycle as one from the following:

```
// MICROARCHITECTURE
    // pc_com
    parameter
        stop        = 3'b000, // pc = pc
        next        = 3'b001, // pc = pc + 1
        small_jmp   = 3'b010, // pc = pc + value
        big_jmp     = 3'b011, // pc = pc + tos
        abs_jmp     = 3'b100, // pc = value
        ret_jmp     = 3'b101; // pc = tos
    // alu_com
    parameter
        alu_left    = 4'b0000,  // alu_out = left
        alu_right   = 4'b0001,  // alu_out = right
        alu_inc     = 4'b0010,  // alu_out = left + 1
        alu_dec     = 4'b0011,  // alu_out = left - 1
        alu_add     = 4'b0100,  // alu_out = left + right = add[32:0]
        alu_sub     = 4'b0101,  // alu_out = left - right = sub[32:0]
        alu_shl     = 4'b0110,  // alu_out = {1'b0, left[31:1]}
        alu_half    = 4'b0111,  // alu_out = {left[31], left[31:1]}
        alu_zero    = 4'b1000,  // alu_out = {31'b0, (left == 0)}
        alu_equal   = 4'b1001,  // alu_out = {31'b0, (left == right)}
        alu_less    = 4'b1010,  // alu_out = {31'b0, (left < right)}
        alu_carry   = 4'b1011,  // alu_out = {31'b0, add[32]}
        alu_borrow  = 4'b1100,  // alu_out = {31'b0, sub[32]}
        alu_and     = 4'b1101,  // alu_out = left & right
        alu_or      = 4'b1110,  // alu_out = left | right
        alu_xor     = 4'b1111;  // alu_out = left ^ right
    // data_sel
    parameter
        alu         = 2'b00,    // stack_input = alu_out
        mem         = 2'b01,    // stack_input = data_in
        val         = 2'b10,    // stack_input = value
        return      = 2'b11;    // stack_input = ret_addr
    // stack_com
    parameter
        s_nop       = 3'b000,   // no operation
        s_swap      = 3'b001,   // swap the content of the first two
        s_push      = 3'b010,   // push
        s_write     = 3'b100,   // write in tos
        s_pop       = 3'b101,   // pop
        s_popwr     = 3'b110,   // pop2 & push
        s_pop2      = 3'b111;   // pops two values
    // data_mem_com
    parameter
        mem_nop     = 2'b00,    // no data memory command
        read        = 2'b01,    // read from data memory
        write       = 2'b10;    // write to data memory
```

Figure 7.5: **The micro-architecture of our Stack Processor.** The content of file `micro_architecture.v` defines each command word generated by the decoder describing the associated micro-commands and their binary codes.

  - the output of ALU
  - data received from data memory addressed by `tos` (with a delay of one clock cycle controlled by `mem_ready` signal because the external data memory is synchronous)
  - the 16-bit integer selected from the current instruction
  - `pc+1`, generated by the PROGRAM FETCH module, to be pushed in stack when the a call instruction is executed

- `stack_com`: is a 3-bit code used to select the operation performed by the stack unit in the current cycle (it is correlated with the ALU operation selected by `alu_com`); the following micro-operations are codded:
  - `push`: it is the well known standard writing operation into a stack memory
  - `pop`: it is the well known standard reading operation into a stack memory
  - `write`: it writes in top of stack, which is equivalent with popping an operand and pushing back the result of operation performed on it (used mainly in performing unary operations)
  - `pop & write`: it is equivalent with popping two operands from stack and pushing back the result of operation performed on them (used mainly in performing binary operations)
  - `double pop`: it is equivalent with two successive pops, but is performed in one clock cycle; some instructions need to remove both the content of `stack0` and of `stack1` (for example, after a data write into the external data memory)
  - `swap`: it exchange the content of `stack0` and of `stack1`; it is useful, for example to make a subtract in the desired order.

- `data_mem_com`: is a 2-bit command for the external data memory; it has three instantiations:
  - `memory nop`: keep memory doing nothing is a very important command
  - `read`: commands the read operation from data memory with the address from tos; the data will be returned in the next clock cycle; in the current cycle `mem_read` is activated to allow stoping the processor one clock cycle (the associated read instruction will be executed in two clock cycles)
  - `write`: the data contained in `stack1` is written to the address contained in `stack0` (both, address and data will be popped from stack)

- `pc_com`: is a 3-bit code used to command how is computed the address for the fetching of the next instruction; 6 modes are used:
  - `stop`: program counter is not incremented (the processor halts or is waiting for a condition to be fulfilled)
  - `next`: it is the most frequent mode to compute the program counter by incrementing it
  - `small jump`: compute the next program counter adding to it the value contained in the current instruction (`instruction[15:0]`) interpreted as a 16-bit signed integer; a relative jump in program is performed
  - `big jump`: compute the next program counter adding to it the value contained in `tos` interpreted as a 32-bit signed integer; a relative big jump in program is performed

- `absolute jump`: the program counter takes the value of `instruction[15:0]`; thhe processor performs an absolute jump in program
- `return jump`: is an absolute jump performed using the content of `tos` (usually performs a return from a subroutine, or is used to call a subroutine in a big addressing space)

The 5-field just explained can not be filled up without inter-restrictions imposed by the meaning of the micro-operations. There exist inter-correlations between the micro-operations assembled in a command. For example, if ALU performs an addition, then the stack must perform mandatory `pop & poop& & push == pop_write`. If the ALU operation is increment, then the stack must perform `write`. Some fields are sometimes meaningless. For example, when an unconditioned small jump is performed the fields `alu_com` and `data_sel` can take `don't care` values. But, for obvious reasons, no times `stack_com` and `data_mem_com` can take `don't care` values.

Each unconditioned instruction has associated one 5-field commands, and each conditioned instructions is defined using two 5-field commands.

7.2.3 The instruction set architecture

Instruction set architecture is the *interface* between the hardware and the software part of a computing machine. It grounds the definition of the lowest level programming language: the **assembly language**. It is an interface because allows the parallel work of two teams once its definitions is frozen. One is the hardware team which starts to design the physical structure, and the other is the software team which starts to grow the symbolic structure of the hierarchy of programs. Each architecture can be embodied in many forms according to the technological restrictions or to the imposed performances. The main benefit of this concept is the possibility to change the hardware without throwing out the work done by the software team.

The implementation of our Stack Processor has, as the majority of the currently produced processors, an instruction set architecture containing the following class of instructions:

arithmetic and logic instructions having the form:

- `[stack0, stack1, s2, ...] = [op(stack0, stack1), s2, ...]`
  where: `stack0` is the *top of stack*, `stack1` is the *next recording* in stack, and `op` is an arithmetic or logic binary operation
- `[stack0, stack1, s2, ...] = [(op(stack0), stack1, s2, ...]`
  if the operation `op` is unary

input-output instructions which uses `stack0` as `data_addr` and `stack1` as `data_out`

stack instructions (only for stack processors) used to immediate load the stack or to change the content in the first two recordings (`stack0` and `stack1`)

test instructions used to test the content of stack putting the result of the test back into the stack

control instructions used to execute unconditioned or conditioned jumps in the instruction stream by modifying the variable `program_counter` used to address in the program space.

The instruction set architecture is given as part of the Verilog code describing the module `decode`: the content of the file `instruction_set_architecture.v` (a more complete stage of this module in *Appendix: Designing a stack processor*). Figure 7.6 contains an incipient form of file

```
// INSTRUCTION SET ARCHITECTURE
 // arithmetic & logic instructions (pc <= pc + 1)
  parameter
  nop     = 8'b0000_0000, // s0, s1, s2 ... <= s0, s1, s2, ...
  add     = 8'b0000_0001, // s0, s1, s2 ... <= s0 + s1, s2, ...
  inc     = 8'b0000_0010, // s0, s1, s2 ... <= s0 + 1, s1, s2, ...
  half    = 8'b0000_0011; // s0, s1, s2 ... <= s0/2, s1, s2, ...
  // ...
 // input output instructions (pc <= pc + 1)
  parameter
  load    = 8'b0001_0000, // s0, s1, s2 ... <= data_mem[s0], s1, s2, ...
  store   = 8'b0001_0001; // s0, s1, s2 ... <= s2, s3, ...; data_mem[s0] = s1
 // stack instructions  (pc <= pc + 1)
  parameter
  push    = 8'b0010_0000, // s0, s1, s2 ... <= value, s0, s1,  ...
  pop     = 8'b0010_0010, // s0, s1, s2 ... <= s1, s2, ...
  dup     = 8'b0010_0011, // s0, s1, s2 ... <= s0, s0, s1, s2, ...
  swap    = 8'b0010_0100, // s0, s1, s2 ... <= s1, s0, s2, ...
  over    = 8'b0010_0101; // s0, s1, s2 ... <= s1, s0, s1, s2, ...
  // ...
 // test instructions  (pc <= pc + 1)
  parameter
  zero    = 8'b0100_0000, // s0, s1, s2 ... <= (s0 == 0), s1, s2, ...
  eq      = 8'b0100_0001; // s0, s1, s2 ... <= (s0 == s1), s2, ...
  // ...
 // control instructions
  parameter
  jmp     = 8'b0011_0000, // pc = pc + value
  call    = 8'b0011_0001, // pc = s0; s0, s1, ... <= pc + 1, s1, ...
  cjmpz   = 8'b0011_0010, // if (s0 == 0) pc <= pc + value, else pc <= pc + 1
  cjmpnz  = 8'b0011_0011, // if (s0 == 0) pc <= pc + 1, else pc <= pc + value
  ret     = 8'b0011_0111; // pc = s0; s0, s1, ... <= s1, s2, ...
  // ...
```

Figure 7.6: **Instruction set architecture of our Stack Processor.** From each subset few typical example are shown. The content of data stack is represented by: `s0, s1, s2, ...`.

`instruction_set_architecture.v`. From each class of instructions only few examples are shown. Each instruction is performed in one clock cycle, except `load` whose execution can be delayed if `data_ready = 0`.

7.2.4 Implementation: from micro-architecture to architecture

Designing a processor (in our case designing the Stack Processor) means to use the micro-architecture to implement the instruction set architecture. For an executing processor the ”connection” between micro-architecture and architecture is done by the decoder which is a combinational structure.

The main body of the `decode` module – `decoder_implementation.v` – contains the description of the Stack Processor's instruction set architecture in term of micro-architecture.

The structure of the file `decoder_implementation.v` is suggested in Figure 7.7, where the output variables are the 5 command fields (declared as registers) and the input variables are: the operation

code from `instruction`, the value of `tos` received as `test_in` and the flag received from the external memory: `mem_ready`.

The main body of this vile consists in a big `case` structure with an entry for each instruction. In Figure 7.7 only few instructions are implemented (`nop, add, load`) to show how an unconditioned instruction `nop, add` or a conditioned instruction `load` is executed.

Instruction `nop` does not affect the state of stack and PC is incremented. We'must take care only about three command fields. PC must be incremented (`next`, and the fields commanding memory resources (stack, external data memory) must be set on "no operation" (`s_nop, mem_nop`. The operation performed by ALU and data selected as `right` operand have no meaning for this instruction.

Instruction `add` pops the two last recordings in stack, adds them, pushes back the result in `tos`, and increments PC. Meantime the data memory receives no active command.

Instruction `load` is executed in two clock cycles. In the first cycle, when `mem_ready = 0`, the command `read` is sent to the external data memory, and the PC is maintained unchanged. The operation performed by ALU does not matter. The selection code for MUX4 does not matter. In the next clock cycle data memory sets it flag on 1 (`mem_ready = 1` means the requested data is available), data selected is from memory `mem`), and the output of MUX4 is pushed in stack ((`s_push`).

By *default* the decoder generates "dont'care" commands. Another possibility is to have `nop` instruction the "by default" instruction. Or by default to have a halt instruction which stops the processor. The first version is good as a final solution because generates a minimal solution. The last version is preferred in the initial stage of development because provides an easy testing and debugging solution.

Follows the description of some typical instructions from a possible instruction set executed by our Stack Processor.

Instruction `inc` increments the top of stack, and increments also PC. The right operand of ALU does not matter. The code describing this instruction, to be inserted into the big `case` sketched in Figure 7.7, is the following:

```
inc     :   begin   pc_com      = next      ;
                    alu_com     = alu_inc   ;
                    data_sel    = alu       ;
                    stack_com   = s_write   ;
                    data_mem    = m_nop     ;
            end
```

Instruction `store` stores the value contained in `stack1` at the address from `stack0` in external data memory. Both, data and address are popped from stack. The associated code is:

```
store   :   begin   pc_com      = next  ;
                    alu_com     = 4'bx  ;
                    data_sel    = 2'bx  ;
                    stack_com   = s_pop2;
                    data_mem    = write ;
            end
```

```
// THE IMPLEMENTATION

    reg     [3:0]   alu_com                 ;
    reg     [2:0]   pc_com, stack_com       ;
    reg     [1:0]   data_sel, data_mem_com  ;

    always @(op_code or test_in or mem_ready      )
        case(op_code)
    // arithmetic & logic instructions
                nop :   begin   pc_com          = next      ;
                                alu_com         = 4'bx      ;
                                data_sel        = 2'bx      ;
                                stack_com       = s_nop     ;
                                data_mem_com    = mem_nop   ;
                        end
                add :   begin   pc_com          = next      ;
                                alu_com         = alu_add   ;
                                data_sel        = alu       ;
                                stack_com       = s_popwr   ;
                                data_mem_com    = mem_nop   ;
                        end
                // ...
    // input output instructions
                load    :   if (mem_ready)
                                begin   pc_com          = next      ;
                                        alu_com         = 4'bx      ;
                                        data_sel        = mem       ;
                                        stack_com       = s_write   ;
                                        data_mem_com    = mem_nop   ;
                                end
                              else
                                begin   pc_com          = stop  ;
                                        alu_com         = 4'bx  ;
                                        data_sel        = 2'bx  ;
                                        stack_com       = s_nop ;
                                        data_mem_com    = read  ;
                                end
                // ...
    // ...
                default     begin   pc_com          = 3'bx  ;
                                    alu_com         = 4'bx  ;
                                    data_sel        = 2'bx  ;
                                    stack_com       = 3'bx  ;
                                    data_mem_com    = 2'bx  ;
                            end
        endcase
```

Figure 7.7: **Sample from the file** `decoder_implementation.v`**.** Implementation consists in a big `case` form with an entry for each instruction.

Instruction `push` pushes {`16'b0, instruction[15:0]`} in in stack. The code is:

```
push    :   begin   pc_com      = next      ;
                    alu_com     = 4'bx      ;
                    data_sel    = val       ;
                    stack_com   = s_push    ;
                    data_mem    = m_nop     ;
                end
```

Instruction `dup` pushes in stack the top of stack, thus duplicating it. ALU performs `alu_left`, the right operand does not matter, and in the stack is pusher the output of ALU. The code is:

```
dup     :   begin   pc_com      = next      ;
                    alu_com     = alu_left  ;
                    data_sel    = alu       ;
                    stack_com   = s_push    ;
                    data_mem    = m_nop     ;
                end
```

Instruction `over` pushes `stack1` in stack, thus duplicating the second stage of stack. ALU performs `alu_right`, and in the stack is pusher the output of ALU.

```
over    :   begin   pc_com      = next      ;
                    alu_com     = alu_right ;
                    data_sel    = alu       ;
                    stack_com   = s_push    ;
                    data_mem    = m_nop     ;
                end
```

The sequence of instructions:

```
over;
over;
```

duplicates the first two recordings in stack to be reused later in another stage of computation.

Instruction `zero` substitute the top of stack with 1, if its content is 0, or with 0 if the content is different from 0.

```
zero    :   begin   pc_com      = next      ;
                    alu_com     = alu_zero  ;
                    data_sel    = alu       ;
                    stack_com   = s_write   ;
                    data_mem    = m_nop     ;
                end
```

This instruction is used in conjunction with a conditioned jump (`cjmpz` or `cjmpnz`) to decide according to the value of `stack0`.

Instruction `jmp` adds to PS the signed value `instruction[15:0]`.

```
jmp     :   begin   pc_com      = rel_jmp   ;
                    alu_com     = 4'bx      ;
                    data_sel    = 2'bx      ;
                    stack_com   = s_nop     ;
                    data_mem    = m_nop     ;
                end
```

This instruction is expressed as follows:

```
jmp <value>
```

where, `<value>` is expressed sometimes as an explicit signed integer, but usually as a **label** which takes a numerical value only when the program is assembled. For example:

```
jmp loop1;
```

Instruction `call` performs an absolute jump to the subroutine placed at the address `instruction[15:0]`, and saves in `tos` the return address (`ret_addr`) which is `pc + 1`. The address saved in stack will be used by `ret` instruction to return the processor from the subroutine into the main program.

```
call    :   begin   pc_com      = abs_jmp   ;
                    alu_com     = 4'bx      ;
                    data_sel    = return    ;
                    stack_com   = s_write   ;
                    data_mem    = m_nop     ;
                end
```

The instruction is used, for example, as follows:

```
jmp subrt5;
```

where `subrt5` is the label of a certain subroutine.

Instruction `cjmpz` performs a relative jump if the content of `tos` is zero; else PC is incremented. The content of stack is unchanged. (A possible version of this instruction pops the tested value from the stack.)

```
cjmpz   :   if (test_in == 32'b0)
                begin   pc_com      = small_jmp ;
                        alu_com     = 4'bx      ;
                        data_sel    = 2'bx      ;
                        stack_com   = s_nop     ;
                        data_mem    = m_nop     ;
                end
              else
                begin   pc_com      = next  ;
                        alu_com     = 4'bx  ;
                        data_sel    = 2'bx  ;
                        stack_com   = s_nop ;
                        data_mem    = m_nop ;
                end
```

The instruction is used, for example, as follows:

```
jmp george;
```

where `george` is a label to be converted in a signed 16-bit integer in the assembly process.

Instruction `ret` performs a jump from subroutine back into the main program using the address popped from `tos`.

```
ret     :   begin   pc_com      = ret_jmp   ;
                    alu_com     = 4'bx      ;
                    data_sel    = 2'bx      ;
                    stack_com   = s_pop     ;
                    data_mem    = m_nop     ;
                end
```

The hardware resources of this Stack Processor permits up to 256 instructions to be defined. For this simple machine we do not need to define too many instructions. Therefore, a "smart" codding of instructions will allow minimizing the size of decoder. More, for some critical paths the depth of decoder can be also minimized, eventually reduced to zero. For example, maybe it is possible to set

```
alu_com = instruction[19:16]
data_sel = instruction[21:20]
```

allowing the critical loop to be closed faster.

7.2.5 Time performances

Evaluating the time behavior of the just designed machine does not make us too happy. The main reason is provided by the fact that all the external connections are unbuffered.

All the three inputs, `instruction, data_in, mem_ready` must be received long time before the active edge of clock because their combinational path inside the Stack Processor are too deep. More, these paths are shared partially with the internal loops responsible for the maximum clock frequency. Therefore, optimizing the clock interferes with optimizing t_{in_reg}.

Similar comments apply to the output combinational paths.

The most disturbing propagation path is the combinational path going from inputs to outputs (for example: from `instruction` to `data_mem_com`). The impossibility to avoid t_{in_out} make this design very unfriendly at the system level. Connecting this module together with a data memory a program memory and some input-output circuits will generate too many (restrictive) time dependencies.

This kind of approach can be useful only if it is strongly integrated with the design of the associated memories and interfaces in a module having all inputs and outputs strictly registered.

The previously described Stack Processor remains to be a very good bad example of a pure functionally centered design which ignores the basic electrical restrictions.

7.2.6 Concluding about our Stack Processor

The simple processor exemplified by Stack Processor is typical for a computational engine: it contains an simple working 3loop system – SALU – and another simple automaton – Program Fetch – both driven by a decoder to execute what is codded in each fetched instruction. Therefore, the resulting system is

a 4th order one. This is not **the** solution! A lot of improvement are possible, and a lot of new features can be added. But it is very useful to exemplify one of the main virtue of the fourth loop: the 4-loop processing. A processor with more than the minimal 3 loops is easiest to be controlled. In our cases the operands are automatically selected by the stack mechanism. Results a lot of advantages in control and some performance loss. But, the analysis of pros & cons is not a circuit design problem. It is a topics to be investigated in the computer architecture domain.

The main advantages of a stack machine is its simplicity. The operands are in each cycle already selected, because they are the first to recording in the top of stack. Results a simple instruction containing only two fields: `op_code[7:0]` and `value[15:0]`.

The loop inside SALU is very short allowing a high clock frequency (if other loop do not impose a smaller one).

The main disadvantage of a stack machine is the stack discipline which sometimes adds new instructions in the code generated by the compiler.

Writing a compiler for this kind of machine is simple because the discipline in selecting the operands is high. The efficiency of the resulting code is debatable. Sometimes a longer sequence of operation is compensated by the higher frequency allowed by a stack architecture.

A real machine can adopt a more sophisticated stack in order to remove some limitation imposed by the restricted access imposed by the discipline.

7.3 Problems

Problem 7.1 *Interpretative processor with distinct program counter block.*

7.4 Projects

Project 7.1

Chapter 8

SELF-ORGANIZING STRUCTURES: N-th order digital systems

Von Neumann's architecture is supported by a structure which consists of two distinct subsystems. The first is a *processor* and the second is a *memory* that stores the bits to be processed. In order to be processed a bit must be carried from the memory to the processor and many times back, from the processor to the memory. Too much time is wasted and many structures are involved only to move bits inside the computing machine. The functional development in the physical part of a digital systems stopped when this universal model was adopted. In the same time the performances of the computation process are theoretically limited. All the sort of parallelism pay tribute to this style that is sequentially founded. We have only one machine, each bit must be accessed, processed and after that restored. This "ritual" stopped the growing process of digital machines around the fifth order. There are a small number of useful systems having more than five loops.

The number of loops can become very large if we give up this model and we have the nerve to store "each bit" near its own "processor". A strange, but maybe a winning solution is to "interleave" the processing elements with the storage circuits [Moto-Oka '82]. Many of us believe that this is a more "natural" solution. Until now this way is only a beautiful promise, but this way deserves more attention.

Definition 8.1 *A digital system, DS, belongs to n-OS if having the size in $O(f(n))$ contains a number of internal loop in $O(n)$.* $\diamond$

A paradigmatic example of n-loop digital system is the cellular automaton (CA). Many applications of CA model *self-organizing* systems.

For the beginning, as a simple introduction, the *stack memory* is presented in a version belonging to n-OS. The next subject will be a new type of memory which tightly interleaves the storage elements with processing circuits: the **Connex memory**. This kind of memory allows fine grain deep parallel processes in computation. We end with the *eco-chip*, a spectacular application of the two-dimensional cellular automata enhanced with the *Connex memory's* functions.

The systems belonging to n-OS support efficiently different mechanisms related to some parallel computation models. Thus, there are many chances to ground true parallel computing architecture using such kind of circuits.

8.1 Push-Down Stack as n-OS

There are only a few "exotic" structures that are implemented as digital systems with a great number of loops. One of these is the stack function that needs at least two loops to be realized, as a system in 2-OS (reversible counter & RAM serially composed). There is another, more uniform solution for implementing the *push-down stack* function or LIFO (last-in first-out) memory. This solution uses a simple, i.e., recursive defined, structure.

Definition 8.2 *The n-level push-down stack, $LIFO_n$, is built serial connecting a $LIFO_{n-1}$ with a $LIFO_1$ as in Figure 8.1. The one level push-down stack is a register, R_m, loop connected with $m \times MUX_2$, so as:*

$S_1S_0 = 00$ *means:* **no op** *- the content of the register does not change*

$S_1S_0 = 01$ *means:* **push** *- the register is loaded with the input value, IN*

$S_1S_0 = 10$ *means:* **pop** *- the register is loaded with the extension input value, EXTIN.* $\diamond$

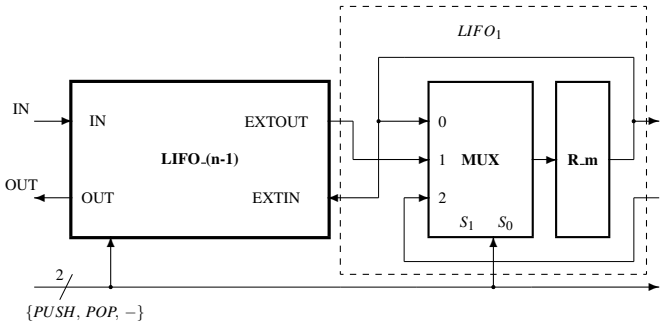


Figure 8.1: The LIFO structure as $n-OS$

It is evident that $LIFO_n$ is a bi-directional serial-parallel shift register. Because the content of the serial-parallel register shifts in both directions each R_m is contained in two kind of loops:

- through its own MUX for **no op** function
- through two successive $LIFO_1$

Thus, $LIFO_1$ is a 2-OS, $LIFO_2$ is a 3-OS, $LIFO_3$ is a 4-OS, ..., $LIFO_i$ is a $(i+1)OS$, ...

The push-down stack implemented as a bi-directional serial-parallel register is an example of digital system having the order related with the size. Indeed: $LIFO_{n-1}$ *is a* $n-OS$.

8.2 Cellular automata

A cellular automaton consists of a regular grid of cells. Each cell has a finite number of states. The grid has a finite number of dimensions, usually no more than three. The transition function of each cell is defined in a constant neighborhood. Usually, the next state of the cell depends on its own state and the states of the adjacent cells.

8.2.1 General definitions

The linear cellular automaton

Definition 8.3 *The one-dimension cellular automaton is linear array of n identical cells, where each cell is connected in a constant neighborhood of +/- m cells, see Figure 8.2a for* $m = 1$. *Each cell is a s-state finite automaton.*

◇

Definition 8.4 *An elementary cellular automaton is a one-dimension cellular automaton with* $m = 1$ *and* $s = 2$. *The transition function of each automaton is a three-input Boolean function defined by the decimally expressed associated Boolean vector.*

◇

Example 8.1 *The Boolean vector of the three-input function*

$$f(x_2, x_1, x_0) = x_2 \oplus (x_1 + x_0)$$

is:

$$00011110$$

and defines the transition rule 30.

◇

Definition 8.5 *The Verilog definition of the elementary cellular automaton is:*

```
module eCellAut #(parameter n = 127)   // n-cell elementary cellular automaton
   (   output  [n-1:0] out ,
       input   [7:0]   func,   // the Boolean vector for the transition rule
       input   [n-1:0] init,   // used to initialize the cellular automaton
       input           rst ,   // loads the initial state of the cellular automaton
       input           clk );

   genvar i;
   generate for (i=0; i<n; i=i+1) begin: C
       eCell eCell(.out    (out[i]                          ),
                   .func   (func                            ),
                   .init   (init[i]                         ),
                   .in0    ((i==0) ? out[n-1] : out[i-1]    ),
                   .in1    ((i==n-1) ? out[0] : out[i+1]    ),
                   .rst    (rst                             ),
                   .clk    (clk                             ));
       end
   endgenerate
endmodule
```

where the elementary cell, `eCell`*, is:*

```
module eCell                        // elementary cell
   (   output  reg         out ,
       input       [7:0]   func,
       input               init,
```

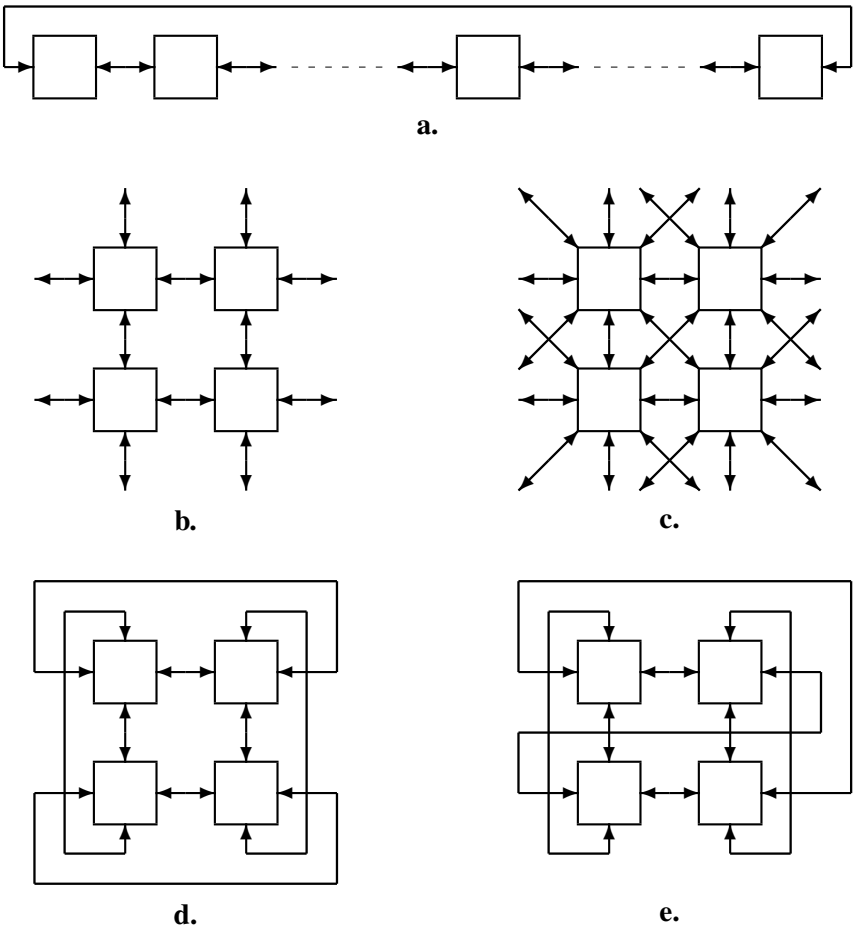


Figure 8.2: **Cellular automaton. a.** One-dimension cellular automaton. **b.** Two-dimension cellular automaton with von Neumann neighborhood. **c.** Two-dimension cellular automaton with Moore neighborhood. **d.** Two-dimension cellular automaton with toroidal shape. **e.** Two-dimension cellular automaton with rotated toroidal shape.

```
        input               in0 ,   // input form the previous cell
        input               in1 ,   // input from the next cell
        input               rst ,
        input               clk );

    always @(posedge clk)   if (rst)    out <= init                 ;
                             else       out <= func[{in1, out, in0}];
endmodule
```

◇

Example 8.2 *The elementary cellular automaton characterized by the rule 90 (01011010) provides, starting from the initial state* `1'b1 << n/2`*, the behavior represented in Figure 8.3, where the sequence of lines of bits represent the sequence of the states of the cellular automaton starting from the initial state.*

The shape generated by the elementary cellular automaton 90 is the Sierpinski triangle or the Sierpinski Sieve. It is a fractal named after the Polish mathematician Waclaw Sierpinski who first described it in 1915.

◇

Example 8.3 *The elementary cellular automaton characterized by the rule 30 (00011110) provides, starting from the initial state* `1'b1 << n/2`*, the behavior represented in Figure 8.4, where the sequence of lines of bits represent the sequence of the states of the cellular automaton starting from the initial state.*

◇

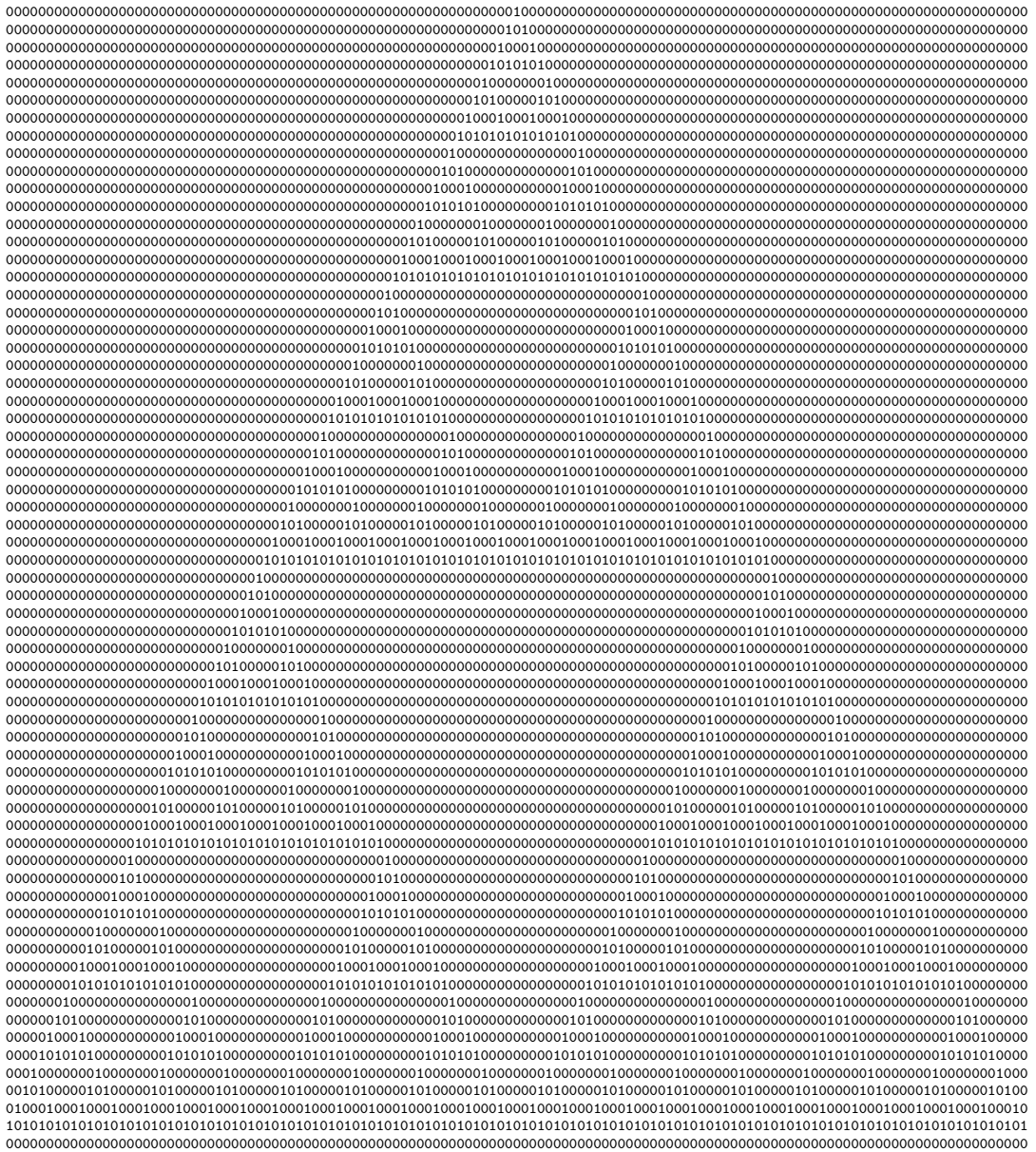


Figure 8.3:

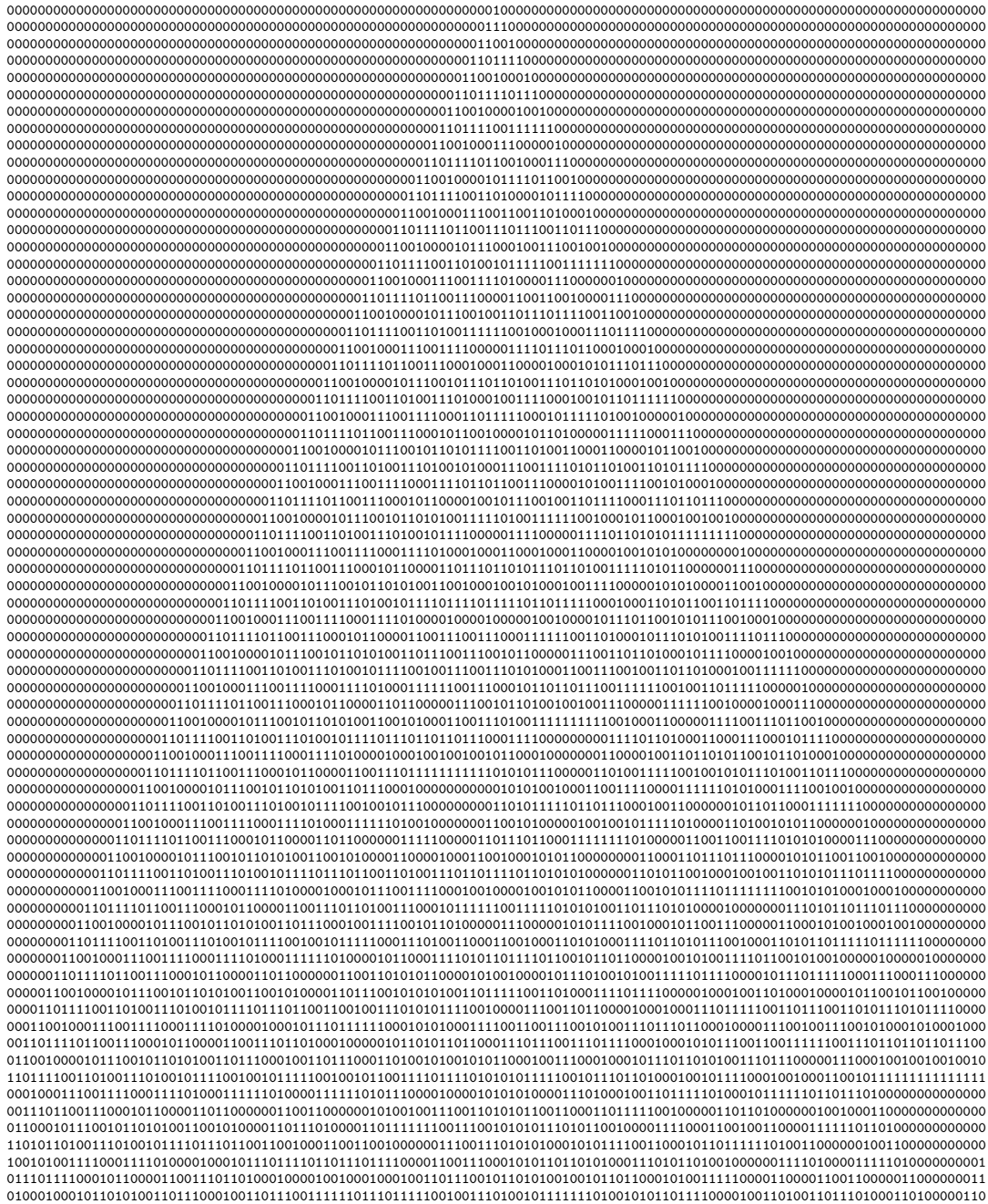


Figure 8.4:

The two-dimension cellular automaton

Definition 8.6 *The two-dimension cellular automaton consists of a two-dimension array of identical cells, where each cell is connected in a constant neighborhood, see Figure 8.2b (the von Neumann neighborhood) and 8.2c (the Moore neighborhood). Each cell is a s-state finite automaton.*

◇

There are also many ways of connecting the border cells. The simplest one is to connect them to ground. Another is close the array so as the surface takes a toroidal shape (see Figure 8.2d). A more complex form is possible if we intend to preserve also a linear connection between the cells. Results a twisted toroidal shape (see Figure 8.2e).

Definition 8.7 *The Verilog definition of the two-dimension elementary cellular automaton with a toroidal shape (Figure 8.2d) is:*

```
module eCellAut4 #(parameter n = 8)    // n*n-cell two-dimension cellular automaton
       (   output  [n*n-1:0]   out ,
           input   [31:0]      func,   // the Boolean vector for the transition rule
           input   [n*n-1:0]   init,   // used to initalize the cellular automaton
           input               rst ,   // loads the inital state of the cellular automaton
           input               clk );

   genvar i;
   generate for (i=0; i<n*n; i=i+1) begin: C
       eCell4 eCell4(  .out    (out[i]                             ),
                       .func   (func                               ),
                       .init   (init[i]                            ),
                       .in0    (out[(i/n)*n+(i-((i/n)*n)+n-1)%n]   ),  // east
                       .in1    (out[(i/n)*n+(i-((i/n)*n)+1)%n]     ),  // west
                       .in2    (out[(i+n*n-n)%(n*n)]               ),  // south
                       .in3    (out[(i+n)%(n*n)]                   ),  // north
                       .rst    (rst                                ),
                       .clk    (clk                                ));
       end
   endgenerate
endmodule
```

where the elementary cell, `eCell4`*, is:*

```
module eCell4                       // 4-input elementary cell
   (   output  reg         out ,
       input       [31:0]  func,
       input               init,   //
       input               in0 ,   // north connection
       input               in1 ,   // east connectioin
       input               in2 ,   // south connection
       input               in3 ,   // west connectioin
       input               rst ,
       input               clk );

   always @(posedge clk)   if (rst)    out <= init                             ;
                            else       out <= func[{in3, in2, out, in1, in0}]  ;
endmodule
```

◇

Example 8.4 *Let be a* 8×8 *cellular automaton with a von Neumann neighborhood and a toroidal shape. The cells are 2-state automata. The transition function is a 5-input Boolean OR, and the initial state is state 1 in the bottom right cell and 0 the the rest of cells. The system will evolve until all the cells will switch in the state 1. Figure 8.5 represents the 8-step evolution from the initial state to the final state.*

```
00000000  00000001  10000011  11000111  11101111  11111111  11111111  11111111  11111111
00000000  00000000  00000001  10000011  11000111  11101111  11111111  11111111  11111111
00000000  00000000  00000000  00000001  10000011  11000111  11101111  11111111  11111111
00000000  00000000  00000000  00000000  00000001  10000011  11000111  11101111  11111111
00000000  00000000  00000000  00000001  10000011  11000111  11101111  11111111  11111111
00000000  00000000  00000001  10000011  11000111  11101111  11111111  11111111  11111111
00000000  00000001  10000011  11000111  11101111  11111111  11111111  11111111  11111111
00000001  10000011  11000111  11101111  11111111  11111111  11111111  11111111  11111111

initial    step 1    step 2    step 3    step 4    step 5    step6     step 7     final
```

Figure 8.5:

◇

Definition 8.8 *The Verilog definition of the two-dimension elementary cellular automaton with linearly connected cells (Figure 8.2e) is:*

```
module eCellAut4L #(parameter n = 8)   // two-dimension cellular automaton
   (   output  [n*n-1:0]   out ,
       input   [31:0]      func,   // the Boolean vector for the transition rule
       input   [n*n-1:0]   init,   // used to initalize the cellular automaton
       input               rst ,   // loads the inital state of the cellular automaton
       input               clk );

   genvar i;
   generate for (i=0; i<n*n; i=i+1) begin: C
       eCell4 eCell4(  .out    (out[i]                 ),
                       .func   (func                   ),
                       .init   (init[i]                ),
                       .in0    (out[(i+n*n-1)%(n*n)]   ),  // east
                       .in1    (out[(i+1)%(n*n)]       ),  // west
                       .in2    (out[(i+n*n-n)%(n*n)]   ),  // south
                       .in3    (out[(i+n)%(n*n)]       ),  // north
                       .rst    (rst                    ),
                       .clk    (clk                    ));
       end
   endgenerate
endmodule
```

where the elementary cell, `eCell4`*, is the same as in the previous definition.*

◇

Example 8.5 *Let us do the same for the two-dimension elementary cellular automaton with linearly connected cells (Figure 8.2e). The insertion of 1s in all the cells is done now in 7 steps. See Figure 8.6.*

Looks like a twisted toroidal shape offers a better neighborhood than a simple toroidal shape.

◇

```
00000000  10000001  11000011  11100111  11111111  11111111  11111111  11111111
00000000  00000000  10000001  11000011  11100111  11111111  11111111  11111111
00000000  00000000  00000000  10000001  11000011  11100111  11111111  11111111
00000000  00000000  00000000  00000000  10000001  11000011  11100111  11111111
00000000  00000000  00000000  00000001  00000011  10000111  11001111  11111111
00000000  00000000  00000001  00000011  10000111  11001111  11111111  11111111
00000000  00000001  00000011  10000111  11001111  11111111  11111111  11111111
00000001  00000011  10000111  11001111  11111111  11111111  11111111  11111111

initial    step 1    step 2    step 3    step 4    stap 5    step 6    final
```

Figure 8.6:

8.2.2 Applications

8.3 Systolic systems

Leiserson's systolic sorter. The initial state: in each cell $= \infty$. For *no operation*: $in1 = +\infty, in2 = -\infty$. To *insert* the value v: $in1 = v$, $in2 = -\infty$. For *extract*: $in1 = in2 = +\infty$.

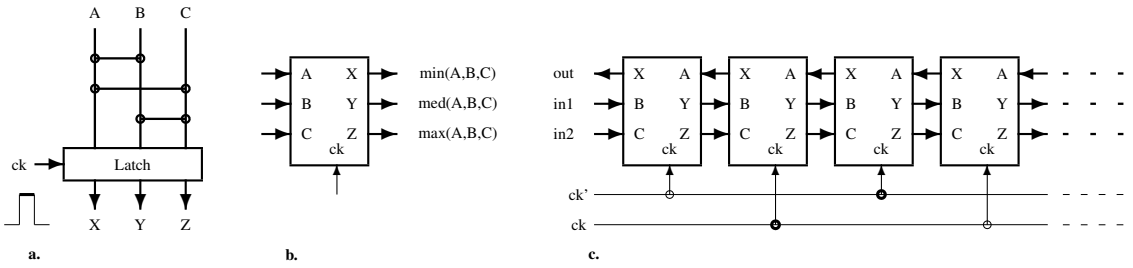


Figure 8.7: **Systolic sorter. a.** The internal structure of cell. **b.** The logic symbol of cell. **c.** The organization of the systolic sorter.

```
module systolicSorterCell #(parameter n=8)(input       [n-1:0] a, b, c,
                                           output  reg [n-1:0] x, y, z,
                                           input               rst, ck);
   wire    [n-1:0] a1, b1  ; // sorter's first level outputs
   wire    [n-1:0] a2, c2  ; // sorter's second level outputs
   wire    [n-1:0] b3, c3  ; // sorter's third level outputs

   assign a1 = (a < b) ? a : b ;
   assign b1 = (a < b) ? b : a ;
   assign a2 = (a1 < c) ? a1 : c   ;
   assign c2 = (a1 < c) ? c : a1   ;
   assign b3 = (b1 < c2) ? b1 : c2 ;
   assign c3 = (b1 < c2) ? c2 : b1 ;

   always @(ck or rst or a2 or b3 or c3)
       if (rst & ck)   begin   x = {n{1'b1}}   ;
                               y = {n{1'b1}}   ;
                               z = {n{1'b1}}   ;
                       end
        else if (ck)   begin   x = a2  ;
                               y = b3  ;
                               z = c3  ;
                       end
endmodule

module systolicSorter #(parameter n=8, m=7)(   output  [n-1:0] out,
                                               input   [n-1:0] in1, in2,
                                               input           rst, ck1, ck2);
   wire    [n-1:0] x[0:m];
   wire    [n-1:0] y[0:m-1];
```

```
   wire    [n-1:0] z[0:m-1];

   assign y[0] = in1       ;
   assign z[0] = in2       ;
   assign out  = x[1]      ;
   assign x[m] = {n{1'b1}} ;

   genvar i;
   generate for(i=1; i<m; i=i+1)   begin: C
       systolicSorterCell  systolicCell(   .a  (x[i+1]),
                                           .b  (y[i-1]),
                                           .c  (z[i-1]),
                                           .x  (x[i]),
                                           .y  (y[i]),
                                           .z  (z[i]),
                                           .rst(rst),
                                           .ck (((i/2)*2 == i) ? ck2 : ck1));
       end
   endgenerate
endmodule

module systolicSorterSim #(parameter n=8);
   reg             ck1, ck2, rst   ;
   reg     [n-1:0] in1, in2;
   wire    [n-1:0] out ;

   initial begin   ck1 = 0 ;
                   forever begin   #3 ck1 = 1  ;
                                   #1 ck1 = 0  ;
                           end
           end
   initial begin       ck2 = 0 ;
                   #2  ck2 = 0 ;
                   forever begin   #3 ck2 = 1  ;
                                   #1 ck2 = 0  ;
                           end
           end

   initial begin       rst = 1 ;
                       in2 = 0 ;
                       in1 = 8'b1000;
                   #8  rst = 0 ;
                   #4  in1 = 8'b0010;
                   #4  in1 = 8'b0100;
                   #4  in1 = 8'b0010;
                   #4  in1 = 8'b0001;
                   #4  in1 = 8'b11111111;
                       in2 = 8'b11111111;
                   #30 $stop;
           end

   systolicSorter dut( out,
```

```
                        in1, in2,
                        rst, ck1, ck2);

    initial
        $monitor("time = %d ck1 = %b ck2 = %b rst = %b in1 = %d in2 = %d out = %d ",
                $time, ck1, ck2, rst, in1, in2, out);
endmodule
```

The result of simulation is:

```
# time =  0 ck1 = 0 ck2 = 0 rst = 1 in1 =   8 in2 =   0 out =   x
# time =  3 ck1 = 1 ck2 = 0 rst = 1 in1 =   8 in2 =   0 out = 255
# time =  4 ck1 = 0 ck2 = 0 rst = 1 in1 =   8 in2 =   0 out = 255
# time =  5 ck1 = 0 ck2 = 1 rst = 1 in1 =   8 in2 =   0 out = 255
# time =  6 ck1 = 0 ck2 = 0 rst = 1 in1 =   8 in2 =   0 out = 255
# time =  7 ck1 = 1 ck2 = 0 rst = 1 in1 =   8 in2 =   0 out = 255
# time =  8 ck1 = 0 ck2 = 0 rst = 0 in1 =   8 in2 =   0 out =   0
# time =  9 ck1 = 0 ck2 = 1 rst = 0 in1 =   8 in2 =   0 out =   0
# time = 10 ck1 = 0 ck2 = 0 rst = 0 in1 =   8 in2 =   0 out =   0
# time = 11 ck1 = 1 ck2 = 0 rst = 0 in1 =   8 in2 =   0 out =   0
# time = 12 ck1 = 0 ck2 = 0 rst = 0 in1 =   2 in2 =   0 out =   0
# time = 13 ck1 = 0 ck2 = 1 rst = 0 in1 =   2 in2 =   0 out =   0
# time = 14 ck1 = 0 ck2 = 0 rst = 0 in1 =   2 in2 =   0 out =   0
# time = 15 ck1 = 1 ck2 = 0 rst = 0 in1 =   2 in2 =   0 out =   0
# time = 16 ck1 = 0 ck2 = 0 rst = 0 in1 =   4 in2 =   0 out =   0
# time = 17 ck1 = 0 ck2 = 1 rst = 0 in1 =   4 in2 =   0 out =   0
# time = 18 ck1 = 0 ck2 = 0 rst = 0 in1 =   4 in2 =   0 out =   0
# time = 19 ck1 = 1 ck2 = 0 rst = 0 in1 =   4 in2 =   0 out =   0
# time = 20 ck1 = 0 ck2 = 0 rst = 0 in1 =   2 in2 =   0 out =   0
# time = 21 ck1 = 0 ck2 = 1 rst = 0 in1 =   2 in2 =   0 out =   0
# time = 22 ck1 = 0 ck2 = 0 rst = 0 in1 =   2 in2 =   0 out =   0
# time = 23 ck1 = 1 ck2 = 0 rst = 0 in1 =   2 in2 =   0 out =   0
# time = 24 ck1 = 0 ck2 = 0 rst = 0 in1 =   1 in2 =   0 out =   0
# time = 25 ck1 = 0 ck2 = 1 rst = 0 in1 =   1 in2 =   0 out =   0
# time = 26 ck1 = 0 ck2 = 0 rst = 0 in1 =   1 in2 =   0 out =   0
# time = 27 ck1 = 1 ck2 = 0 rst = 0 in1 =   1 in2 =   0 out =   0
# time = 28 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   0
# time = 29 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out =   0
# time = 30 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   0
# time = 31 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   1
# time = 32 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   1
# time = 33 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out =   1
# time = 34 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   1
# time = 35 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 36 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 37 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 38 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 39 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 40 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 41 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 42 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   2
# time = 43 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   4
# time = 44 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   4
```

```
# time = 45 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out =   4
# time = 46 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   4
# time = 47 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   8
# time = 48 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   8
# time = 49 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out =   8
# time = 50 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out =   8
# time = 51 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 255
# time = 52 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 255
# time = 53 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out = 255
# time = 54 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 255
# time = 55 ck1 = 1 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 255
# time = 56 ck1 = 0 ck2 = 0 rst = 0 in1 = 255 in2 = 255 out = 255
# time = 57 ck1 = 0 ck2 = 1 rst = 0 in1 = 255 in2 = 255 out = 255
```

8.4 Interconnection issues

Simple circuits scale up easy generating big interconnection problems.

8.4.1 Local vs. global connections

The origin of the *memory wall* is in the inability to avoid global connection on memory arrays, while in the logic areas the local connections are easiest to impose.

Memory wall

8.4.2 Localizing with cellular automata

8.4.3 Many clock domains & asynchronous connections

The clock signal uses a lot of energy and area and slows down the design when the area of the circuit became too big.

A fully synchronous design generate also power distribution issues, which come with all the associated problems.

8.5 Neural networks

Artificial **neural network** (NN) is a technical construct inspired from the biological neural networks. NN are composed of interconnected artificial **neurons**. An artificial neuron is a programmed or circuit construct that mimic the property of a biological neuron. A multi-layer NN is used as a connectionist computational model. The introductory text [Zurada '95] is used for a short presentation of the concept of NN.

8.5.1 The neuron

The artificial neuron (see Figure 8.8) receives the inputs $x_1, \ldots, x_n$ (corresponding to *n dendrites*) and process them to produce an output o (*synapse*). The sums of each node are weighted, using the weight vector $w_1, \ldots, w_n$ and the sum, *net*, is passed through a ***non-linear function***, $f(net)$, called activation

function or transfer function. The transfer functions usually have a sigmoid shape (see Figure 8.9) or step functions.

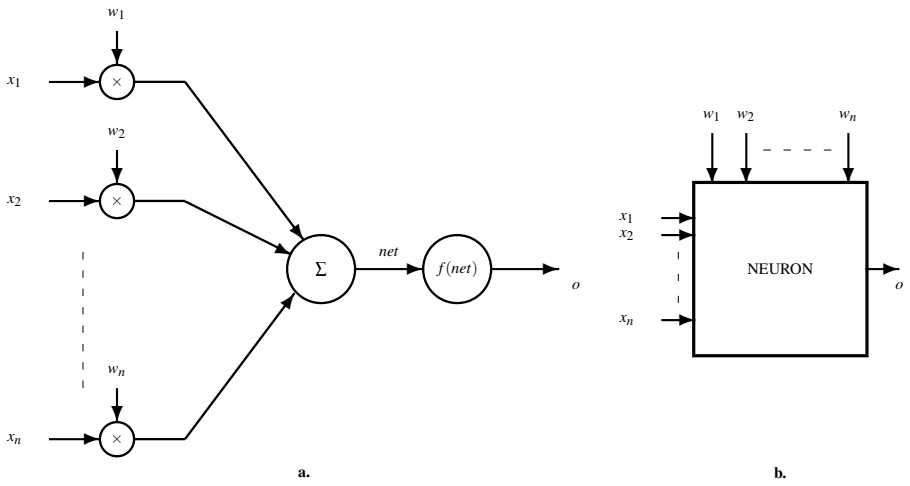


Figure 8.8: **The general form of a neuron. a.** The circuit structure of a n-input neuron. **b.** The logic symbol.

Formally, the transfer function of a neuron:

$$o = f(\sum_{i=1}^{n} w_i x_i) = f(net)$$

where f, the typical activation function, is:

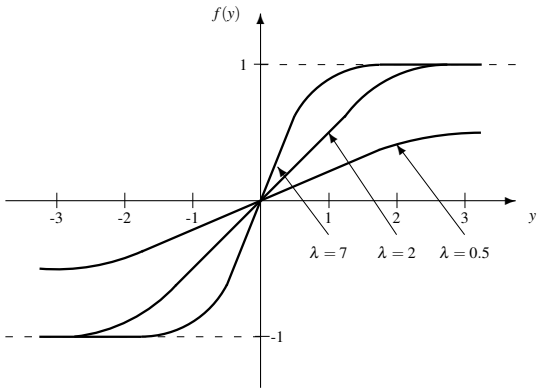


Figure 8.9: **The activation function.**

$$f(y) = \frac{2}{1 + exp(-\lambda y)} - 1$$

The parameter λ determines the steepness of the continuous function f. For big value of λ the function f becomes:

$$f(y) = sgn(y)$$

The neuron works as a combinational circuit performing the scalar product of the input vector

$$\mathbf{x} = [x_1\ x_2\ \ldots\ x_n]$$

with the weight vector

$$\mathbf{w} = [w_1\ w_2\ \ldots\ w_n]$$

followed by the application of the activation function. The activation function f is simply implemented using as a look-up table using a Read-Only Memory.

8.5.2 The feedforward neural network

A feedforward NN is a collection of m n-input neurons (see Figure 8.10). Each neuron receives the same input vector

$$\mathbf{x} = [x_1\ x_2\ \ldots\ x_m]$$

and is characterized by its own weight vector

$$\mathbf{w}_i = [w_1\ w_2\ \ldots\ w_m]$$

The entire NN provides the output vector

$$\mathbf{o} = [o_1\ o_2\ \ldots\ o_m]^t$$

The activation function is the same for each neuron.

Each NN is characterized by the weight matrix

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & \ldots & w_{1n} \\ w_{21} & w_{22} & \ldots & w_{2n} \\ \vdots & \vdots & \ldots & \vdots \\ w_{m1} & w_{m2} & \ldots & w_{mn} \end{pmatrix}$$

having for each output a line, while for each input it has a column. The transition function of the NN is:

$$\mathbf{o}(t) = \Gamma[\mathbf{W}\mathbf{x}(t)]$$

where:

$$\blacksquare[\cdot] = \begin{pmatrix} f(\cdot) & 0 & \ldots & 0 \\ 0 & f(\cdot) & \ldots & 0 \\ \vdots & \vdots & \ldots & \vdots \\ 0 & 0 & \ldots & f(\cdot) \end{pmatrix}$$

The feedforwaed NN is of "instantaneous" type, i.e., it behaves as a combinational circuit which provides the result in the same "cycle". The propagation time associated do not involve storage elements.

Example 8.6 *The shaded area in Figure 8.11 must be recognized by a two-layer feedforward NN. Four conditions must be met to define the surface:*

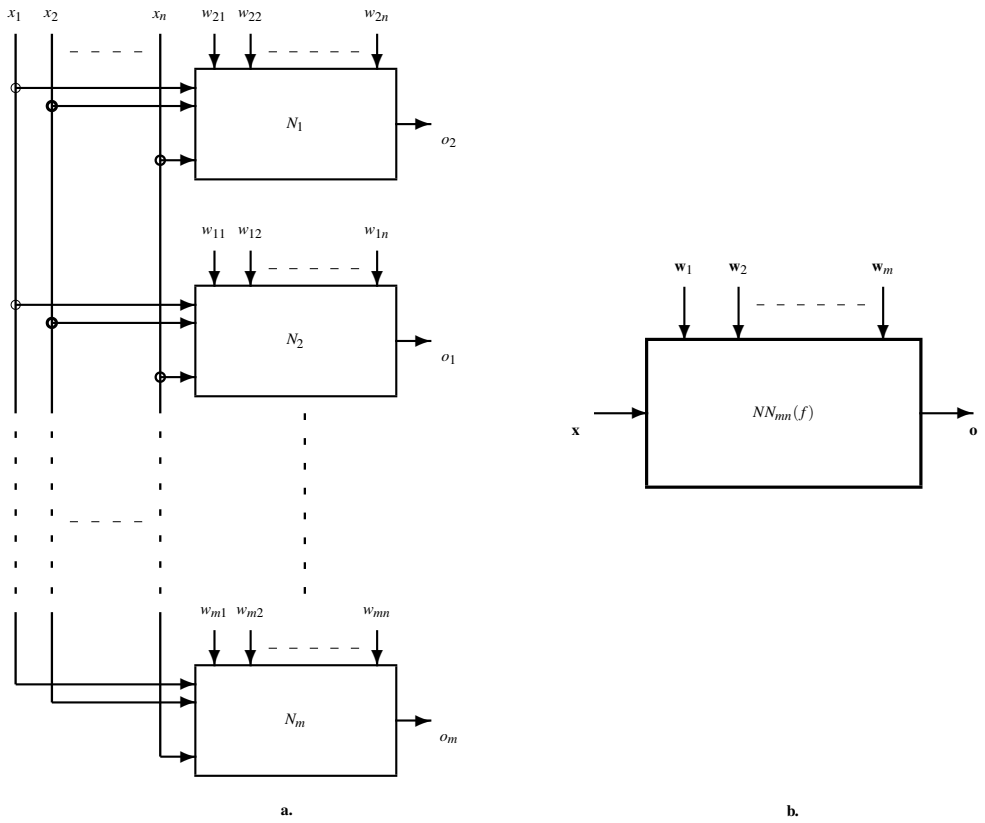


Figure 8.10: **The single-layer feedforward neural network. a.** The organization of a feedforward NN having m n-input neurons. **b.** The logic symbol.

$$
\begin{aligned}
&x_1 - 1 > 0 \rightarrow sgn(x_1 - 1) = 1\\
&x_1 - 2 < 0 \rightarrow sgn(-x_1 + 2) = 1\\
&x_2 > 0 \quad\;\; \rightarrow sgn(x_2) = 1\\
&x_2 - 3 < 0 \rightarrow sgn(-x_2 + 3)
\end{aligned}
$$

For each condition a neuron from the first layer is used. The second layer determines whether all the conditions tested by the first layer are fulfilled.

The first layer is characterized the weight matrix

$$\mathbf{W}_{43} = \begin{pmatrix} 1 & 0 & 1 \\ -1 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & -1 & -3 \end{pmatrix}$$

The weight vector for the second layer is

$$\mathbf{W} = [1 \;\; 1 \;\; 1 \;\; 1 \;\; 3.5]$$

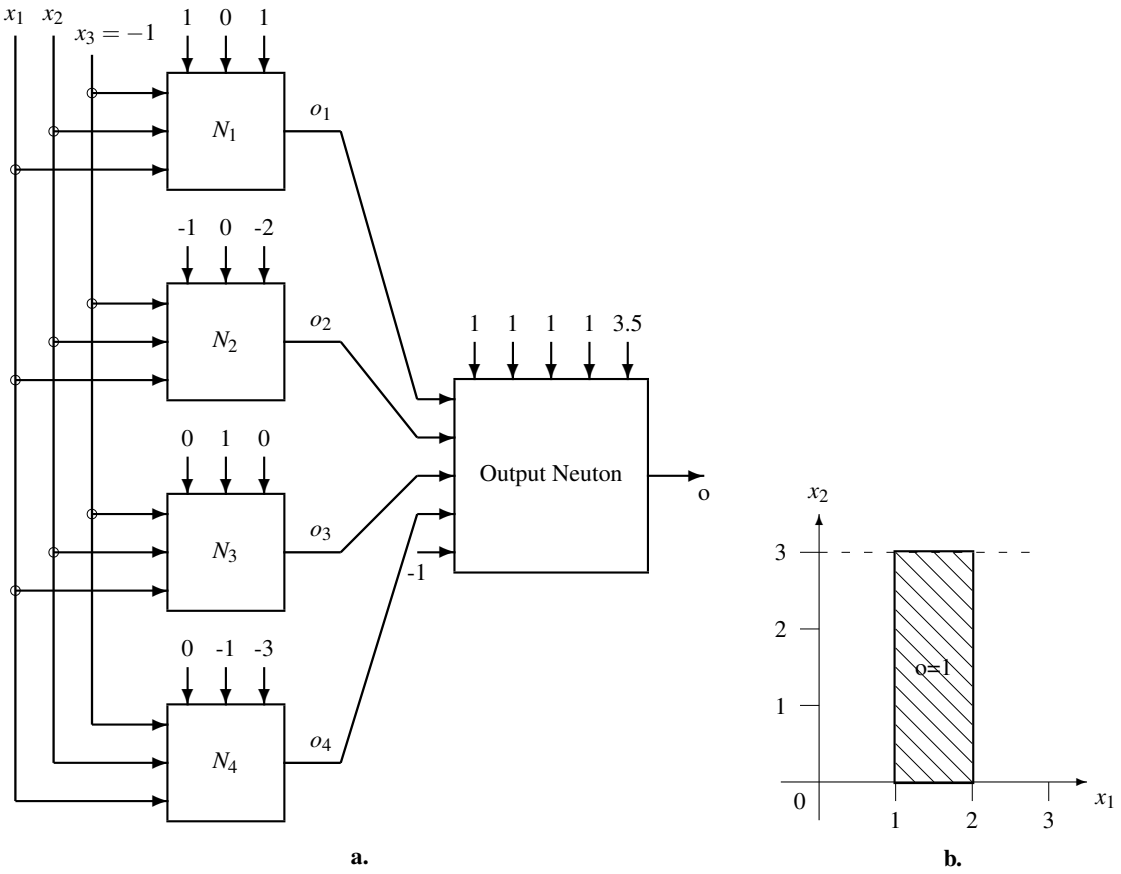


Figure 8.11: **A two-layer feedforward NN. a.** The structure. **b.** The two-dimension space mapping.

On both layers the activation function is sgn.

◇

8.5.3 The feedback neural network

The feedback NN is a sequential system. It provides the output with a delay of a number of clock cycles after the initialization with the input vector **x**. The structure of a feedback NN is presented in Figure 8.12. The multiplexor **mux** is used to initialize the loop closed through **register**. If *init = 1* the vector **x** is applied to $NN_{mn}(f)$ one clock cycle, then *init* is switched to 0 and the loop is closed.

In the circuit approach of this concept, after the initialization cycle the output of the network is applied to the input through the feedback register. The transition function is:

$$\mathbf{o}(t + T_{clock}) = \Gamma[\mathbf{W}\mathbf{o}(t)]$$

where T_{clock} (the clock period) is the delay on the loop. After k clock cycles the state of the network is

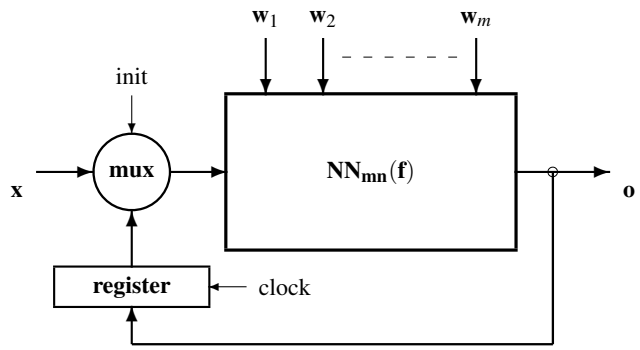


Figure 8.12: **The single-layer feedback neural network.**

described by:

$$\mathbf{o}(t+k\times T_{clock}) = \Gamma[\mathbf{W}\Gamma[\ldots\Gamma[\mathbf{W}\mathbf{o}(t)]\ldots]]$$

A feedback NN can be considered as an initial *automaton* with few final states mapping disjoint subsets of inputs.

Example 8.7 *Let be a feedback NN with 4 4-input neurons with one-bit inputs and outputs. The activation function is* sgn*. The feedback NN can be initialized with any 4-bit binary configuration from*
x = `[-1 -1 -1 -1]`
to
x = `[1 1 1 1]`
and the system has two final states:
$\mathbf{o}_{14}$ = `[1 1 1 -1]`
$\mathbf{o}_1$ = `[-1 -1 -1 1]`
reached in a number of clock cycles after the initialization.

The resulting discrete-time recurrent network has the following weight matrix:

$$\mathbf{W}_{44} = \begin{pmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{pmatrix}$$

The resulting structure of the NN is represented in Figure 8.13, where the weight matrix is applied on the four 4-bit inputs destined for the weight vectors.

The sequence of transitions are computed using the form:

$$\mathbf{o}(t+1) = [sgn(net_1(t))\ \ sgn(net_2(t))\ \ sgn(net_3(t))\ \ sgn(net_4(t))]$$

Some sequences end in $\mathbf{o}_{14}$ = `[1 1 1 -1]`*, while others in* $\mathbf{o}_1$ = `[-1 -1 -1 1]`.

◇

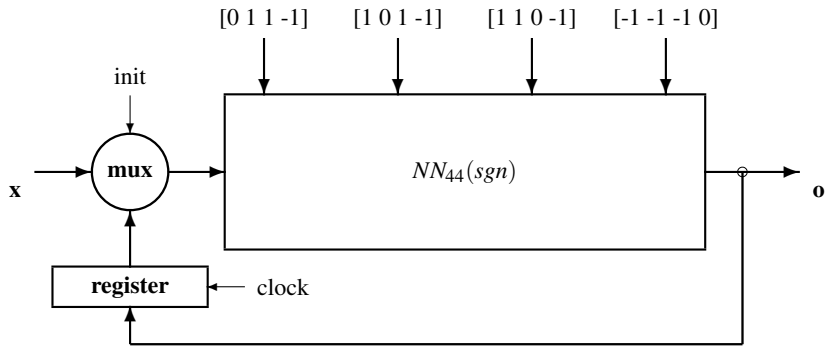


Figure 8.13: **The feedback NN with two final states.**

8.5.4 The learning process

The learning process is used to determine the actual form of the matrix $\mathbf{W}$. The learning process is an iterative one. In each iteration, for each neuron the weight vector $\mathbf{w}$ is adjusted with $\Delta\mathbf{w}$, which is proportional with the input vector $\mathbf{x}$ and the learning signal r. The general form of the learning signal is:

$$r = r(\mathbf{w}, \mathbf{x}, d)$$

where d is the desired response (the teacher's signal). Thus, in each step the weight vector is adjusted as follows:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + c \times r(\mathbf{w}(t), \mathbf{x}(t), d(t)) \times \mathbf{x}(t)$$

where c is the *learning constant*. The learning process starts from an initial form of the weight vector (established randomly or by a simple "hand calculation") and uses as a set of training input vectors.

There are two types of learning:

unsupervised learning :

$$r = r(\mathbf{w}, \mathbf{x})$$

the desired behavior is not known; the network will adapt its response by "discovering" the appropriate values for the weight vectors by *self-organization*

supervised learning :

$$r = r(\mathbf{w}, \mathbf{x}, d)$$

the desired behavior, d, is known and can be compared with the actual behavior of the neuron in order to find how to adjust the weight vector.

In the following both types will be exemplified using the *Hebbian rule* and the *perceptron rule*.

Unsupervised learning: Hebbian rule

The learning signal is the output of the neuron. In each step the vector $\mathbf{w}$ will be adjusted (see Figure 8.14) as follows:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + c \times f(\mathbf{w}(t), \mathbf{x}(t)) \times \mathbf{x}(t)$$

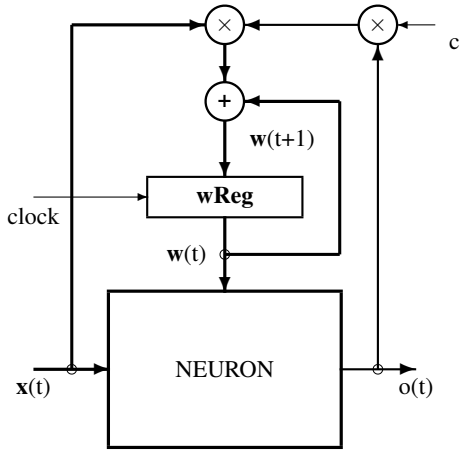


Figure 8.14: **The Hebian learning rule**

The learning process starts with small random values for w_i.

Example 8.8 *Let be a four-input neuron with the activation function sgn. The initial weight vector is:*

$$\mathbf{w}(t_0) = [1\ -1\ 0\ 0.5]$$

The training inputs are:
$\mathbf{x}_1 = [1\ -2\ 1.5\ 0]$,
$\mathbf{x}_2 = [1\ -0.5\ -2\ -1.5]$,
$\mathbf{x}_3 = [0\ 1\ -1\ 1.5]$

Applying by turn the three training input vectors for $c = 1$ we obtain:
$\mathbf{w}(t_0+1) = \mathbf{w}(t_0) + sgn(net) \times \mathbf{x}_1 = \mathbf{w}(t_0) + sgn(3) \times \mathbf{x}_1 = \mathbf{w}(t_0) + \mathbf{x}_1 = [2\ -3\ 1.5\ 0.5]$
$\mathbf{w}(t_0+2) = \mathbf{w}(t_0+1) + sgn(net) \times \mathbf{x}_2 = \mathbf{w}(t_0+1) + sgn(-0.25) \times \mathbf{x}_2 = \mathbf{w}(t_0+1) - \mathbf{x}_2 = [1\ -2.5\ 3.5\ 2]$
$\mathbf{w}(t_0+3) = \mathbf{w}(t_0+2) + sgn(net) \times \mathbf{x}_3 = \mathbf{w}(t_0+2) + sgn(-3) \times \mathbf{x}_3 = \mathbf{w}(t_0+2) - \mathbf{x}_3 = [1\ -3.5\ 4.5\ 0.5]$
◇

Supervised learning: perceptron rule

The perceptron rule performs a supervised learning. The learning is guided by the difference between the desired output and the actual output. Thus, the learning signal for each neuron is:

$$r = d - o$$

In each step the weight vector is updated (see Figure 8.15) according to the relation:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + c \times (d(t) - f(\mathbf{w}(t), \mathbf{x}(t))) \times \mathbf{x}(t)$$

The initial value for $\mathbf{w}$ does not matter.

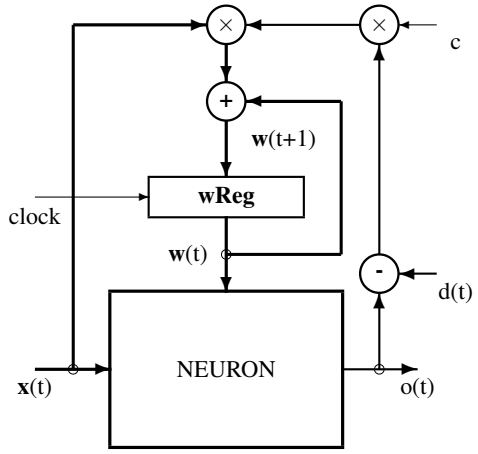


Figure 8.15: **The perceptron learning rule**

Example 8.9 *Let be a four-input neuron with the activation function sgn. The initial weight vector is:*

$$\mathbf{w}(t_0) = [1\ -1\ 0\ 0.5]$$

The training inputs are:
$\mathbf{x}_1 = [1\ -2\ 0\ -1]$,
$\mathbf{x}_2 = [0\ 1.5\ -0.5\ -1]$,
$\mathbf{x}_3 = [-1\ 1\ 0.5\ -1]$
and the desired output for the three input vectors are: $d_1 = -1$, $d_2 = -1$, $d_3 = 1$. *The learning constant is* $c = 0.1$.

Applying by turn the three training input vectors for $c = 1$ *we obtain:*

step 1 *: because* $(d - sgn(net)) \neq 0$
$\mathbf{w}(t_0+1) = \mathbf{w}(t_0) + 0.1 \times (-1 + sgn(net)) \times \mathbf{x}_1 = \mathbf{w}(t_0) + 0.1 \times (-1-1) \times \mathbf{x}_1 = [0.8\ -0.6\ 0\ 0.7]$

step 2 *: because* $(d - sgn(net) \neq 0)$ *no correction is needed in this step*
$\mathbf{w}(t_0+2) = \mathbf{w}(t_0+1)$

step 3 *: because* $(d - sgn(net)) = 2$
$\mathbf{w}(t_0+3) = \mathbf{w}(t_0+2) + 0.1 \times 2 \times \mathbf{x}_3 = [0.6\ -0.4\ 0.1\ 0.5]$

◇

8.5.5 Neural processing

NN are currently used to model complex relationships between inputs and outputs or to find patterns in streams of data. Although NN has the full power of a Universal Turing Machine (some people claim that the use of irrational values for weights results in a machine with *"super-Turing"* power), the real application of this paradigm are limited only to few functions involving specific complex memory functions (please do not use this paradigm to implement a text editor). They are grouped in the following categories:

- auto-association: the input (even a degraded input pattern) is associated to the closest stored pattern
- hetero-association: the association is made between pais of patterns; distorted input patterns are accepted
- classification: divides the input patterns into a number of classes; each class is indicated by a number (can be understood as a special case of hetero-association which returns a number)
- recognition: is a sort of classification with input patterns which do not exactly correspond to any of the patterns in the set
- generalization: is a sort of interpolation of new data applied to the input.

What is specific for this computational paradigm is that its "program" – the set of weight matrices generated in the learning process – do not provide explicit information about the functionality of the net. The content of the weight matrices can not be read and understood as we read and understand the program performed by a conventional processor built by a register file, an ALU, ... The representation of an actual function of a NN defies any pattern based interpretation. Maybe this is the price we must pay for the complexity of the functions performed by NN.

8.6 Problems

Problem 8.1 *Design a stack with the first two recordings accessible.*

Problem 8.2 *Design a stack with the following features in reorganizing the first recordings.*

Problem 8.3 *Design a stack with controlled deep access.*

Problem 8.4 *Design an expandable stack.*

Problem 8.5 *The global loop on a linear cellular automata providing a pseudo-noise generator.*

Problem 8.6

Problem 8.7

Problem 8.8

Chapter 9

GLOBAL-LOOP SYSTEMS

A super-system is characterized by global loops closed over an n-order digital system. A global loop receives on its inputs information distributed over an array of digital modules and sends back in each digital module information related to the whole content of its inputs.

In the first section an introductory examples are provided closing global loops over one-dimension or two-dimension cellular automata. The second section introduces ConnexArray^{TM}, a cellular engine used as a general purpose parallel computing engine. It is controlled by one global loop closed over a linear cells of execution elements. The third section closes the second global loop over the same linear array. The fourth section provides a high level description of the system described in the previous two sections.

9.1 Global loops in cellular automata: the third "turning point"

A first attempt to close a loop over a simple cellular automaton is presented in [Ştefan '98a]. The effect of a global loop on the behavior of a cellular automaton is presented in [Maliţa '13]. In [Gheolbanoiu '14] the attempt from [Ştefan '98a] is finalized as an actual circuit.

9.2 The Generic ConnexArray^{TM}: the first global loop

In 1936 Stephen Kleene defined [Kleene '36] the concept of *partial recursive function* as the general framework for computing any function of form:

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

In [Ştefan '14] is proved that from the three basic rules proposed by Kleene only the first, the composition rule, is independent. Therefore, computation could be defined as repeated application of the composition having the form:

$$f(x_1,\ldots x_n) = g(h_1(x_1,\ldots x_n),\ldots h_p(x_1,\ldots x_n))$$

In Figure 9.1a the two-level circuit version of the composition rule is represented. Each function h_i is computed by a module on the first level, while the function g reduces the resulting vector to a scalar. In Figure 9.1b, the limit case for $p = 1$ is represented. The repeated application of a composition requests the additional structures represented in Figure 9.1c. In [Ştefan '14] the transition from Figure 9.1a and Figure 9.1b to Figure 9.1c is described.

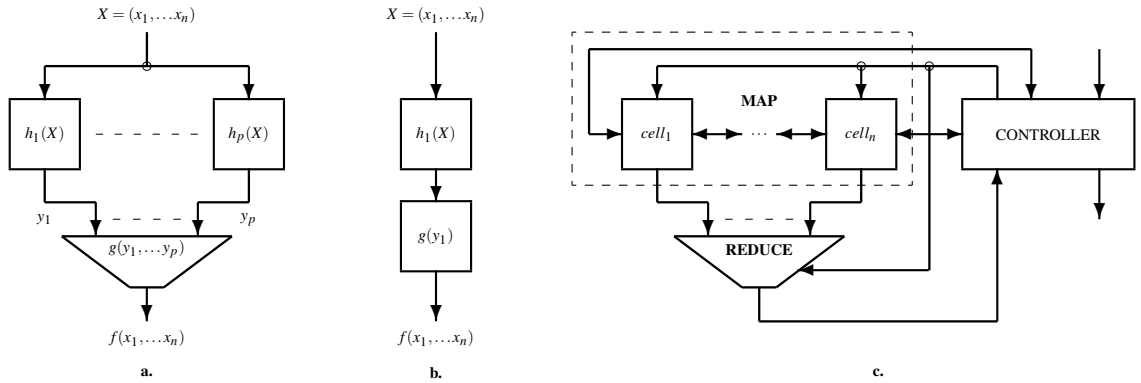


Figure 9.1: **From the mathematical model to the abstract machine model. a.** The structure associated to the composition rule in Kleene's model. **b.** The limit case for $p = 1$. It provides the pipelined connection which can be generalized for serially connected p cells. **c.** The abstract machine model for parallel computing. The MAP section consists of the parallel connected cells, the REDUCE section stands for the function g, while the serial connections between cells in MAP section provides the serial pipelined connection for the limit case of $p = 1$.

The two-direction connections between the cells provide the $p = n$ levels of loops which gives the order n to the system, while the global loop is closed through the CONTROLLER.

The simplest version of the engine is behaviorally described in the next subsection as the generic ConnexArray^{TM} system.

9.2.1 The Generic ConnexArray^{TM}

The cell used in the generic n-order array with the first global loop contains a data memory for the local data and a simple accumulator based engine.

The cell's structure is presented in Figure 9.2a, while the controller's structure is presented in Figure 9.2b.

The behavioral description uses the following memory resources:

```
// vectorial resources
reg [x-1:0] ixVect[0:(1<<x)-1]              ; // index vector
reg         boolVect[0:(1<<x)-1]            ; // Boolean vector
reg [n-1:0] accVect[0:(1<<x)-1]             ; // accumulator vector
reg         crVect[0:(1<<x)-1]              ; // carry vector
reg [v-1:0] addrVect[0:(1<<x)-1]            ; // address vector
reg [n-1:0] vectMem[0:(1<<x)-1][0:(1<<v)-1] ; // vector memory
// control resources
reg [p-1:0] pc                              ; // program counter
reg [63:0]  ir                              ; // instruction register
reg [63:0]  progMem[0:(1<<p)-1]             ; // program memory
// scalar resources
reg [31:10] acc                             ; // scalar accumulator
reg         cr                              ; // scalar carry
```

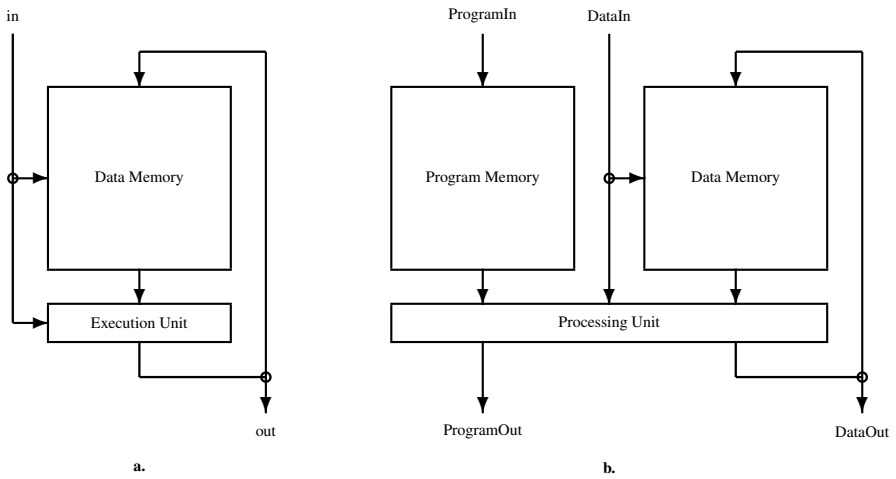


Figure 9.2: **The components of the abstract machine model. a.** The cell's structure. **b.** The controller's structure.

```
reg [s-1:0] addr                          ; // scalar address
reg [n-1:0] mem[0:(1<<s)-1]               ; // scalar memory
```

which are dimensioned as follows:

```
parameter   n = 32  , // word size
            x = 4   , // index size
            v = 6   , // vector memory address size
            s = 8   , // scalar memory address size
            p = 5     // program memory address size
```

The system above defined consists of:

vectorial resources describes the resources distributed in array (for the contribution of each cell see Figure 9.3)

ixVect : index vector used to associate an index, from 0 to $2^x - 1$, to each cell

boolVect : Boolean vector used to enable the execution in i-th cell; if `boolVect[i]` is 1, then the cell is active, else the instruction received in the current cycle is ignored (substituted with `nop`)

accVect : is the scalar vector containing the accumulator registers of the cells; the execution unit is accumulator based, thus `accVect[i]` is the accumulator of the cell i

crVect : is the Boolean vector containing the carry registers of each cell; `crVect[i]` stores the carry bit generated in cell i by the last arithmetic operation

vectMem : is the vector memory distributed along the cells; each cell, $cell_i$, stores in its local memory the i-th components of all the 2^v vectors

addrVect : used to specify a locally computed address in each cell

control resources describes the resources involved in the sequential control

pc : p-bit program counter

ir : 64-bit instruction register

progMem : the program memory organize in 2^p 64-bit words

scalar resources describes the resources involved in the scalar computation

acc : controller's accumulator

cr : the carry flip-flop

addr : the address register used to compute the address for controller's data memory

mem : controller's data memory

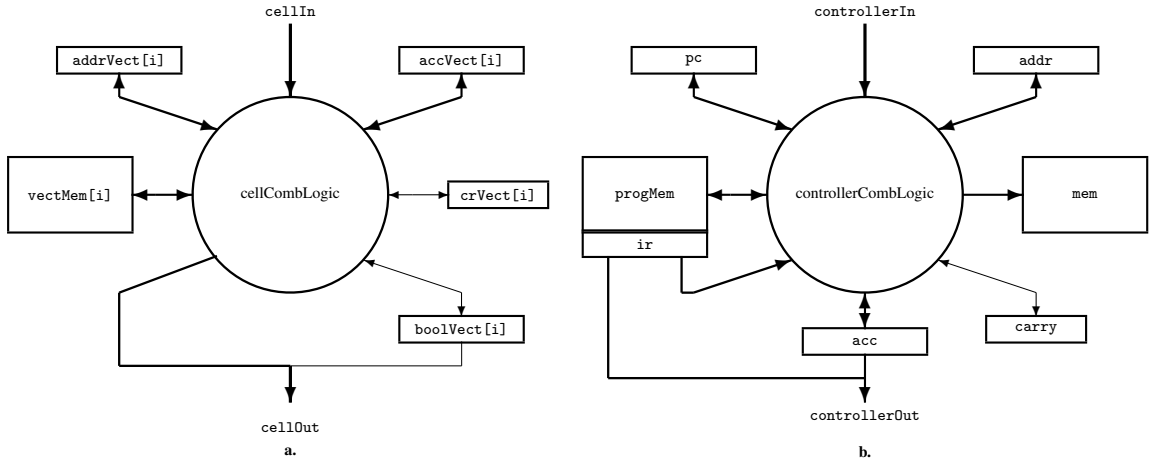


Figure 9.3: **The resources used in the behavioral description.** **a.** Cell's internal state support. **b.** Controller's internal state support.

The input received by each cell (see Figure 9.2a and Figure 9.3) is:

```
in = {instruction[31:0], data[n-1:0], address[v-1:0]}
```

while the output is:

```
out = {boolVect[i], (boolVect[i] ?  accVect[i][n-1:0] :  0)}
```

From the program memory, in each cycle is read a pair of instructions: one (`progMem[nextPc][31:0]`) for the use of controller and another (`progMem[nextPc][63:32]`) to be executed in each active cell (where `boolVect[i] = 1`). The structure of the two instructions is:

```
assign contrOpCode  = ir[31:27] ; // operation code for controller
assign contrOperand = ir[26:24] ; // right selection operand for controller
assign contrScalar  = ir[23:0]  ; // immediate value for controller
assign arrayOpCode  = ir[63:59] ; // operation code for the array of cells
assign arrayOperand = ir[58:56] ; // right selection operand for array
assign arrayScalar  = ir[55:32] ; // immediate value for array
```

The Instruction Set Architecture

The arithmetic-logic operations performed in each cell and in the controller are very similar. Only the sequential control in controller and the spatial control in the array of cells differentiate the instruction set architecture (ISA) which describes the controller and the cells. In the list of instruction which follows the instruction specific for controller are labeled with #, while the instruction specific for the array's cells are labeled with $. The instructions are specified by two fields: one for operation (`opCode`) and one for the right operand (`operand`). The left operand is always the accumulator.

The `operand` code meaning is:

```
parameter         //  selects the right operand or destination
    val = 3'b000, //  immediate value: {24{scalar[7]}}, scalar} (operand only)
    mab = 3'b001, //  absolute: mem[scalar[s/v-1:0]]
    mrl = 3'b010, //  relative: mem[addr + scalar[s/v-1:0]]
    mri = 3'b011, //  relative and increment: mem[addr + scalar[s-1:0]];
                  //  addr <= mem[addr + scalar[s-1:0]]
    ads = 3'b100, //  address register for data memory (destination only)
    cop = 3'b101, //$ co-operand: acc (only for array; operand only)
    idx = 3'b111, //$ index (only for array; operand only)
    rad = 3'b101, //# reduction add (only in scalar section; operand only)
    rmx = 3'b110, //# reduction max (only in scalar section; operand only)
    rfl = 3'b111; //# reduction flag (only in scalar section; operand only)
```

The conditions tested by the instructions `where` and `wheren` are:

```
parameter
    zero    = 3'b000, // accVect[i] == 0
    carry   = 3'b001, // crVect[i] == 1
    first   = 3'b010, // firstVect[i] == 1
    next    = 3'b011; // nextVect[i] == 1
```

The `opCode` code meaning is:

```
parameter                   //  opCode
    add         = 5'b00000, //  {cr, acc} <= acc + op;
    addc        = 5'b00001, //  {cr, acc} <= acc + op + cr;
    sub         = 5'b00010, //  {cr, acc} <= acc - op;
    rsub        = 5'b00011, //  {cr, acc} <= operand - acc;
    subc        = 5'b00100, //  {cr, acc} <= acc - op - cr;
    rsubc       = 5'b00101, //  {cr, acc} <= op - acc - cr;
    mult        = 5'b00110, //  acc <= acc * op;
    load        = 5'b00111, //  acc <= op;
    store       = 5'b01000, //  operand <= acc;
    bwand       = 5'b01001, //  acc <= acc & op;
    bwor        = 5'b01010, //  acc <= acc | op;
    bwxor       = 5'b01011, //  acc <= acc ^ op;
    gshift      = 5'b01100, //$ acc[i] <= acc[i+scalar]; (global shift)
    insval      = 5'b01100, //# acc <= {acc[23:0], scalar}
    shrightc    = 5'b01101, // {cr, acc} <= {acc[0], cr, acc[n-1:1]}
    shright     = 5'b01110, // {cr, acc} <= {acc[0], 1'b0, acc[n-1:1]}
```

```
sharight    = 5'b01111, // acc <= {acc[n-1], acc[n-1:1]}
jmp         = 5'b10000, //# pc <= pc + scalar;
brz         = 5'b10001, //# pc <= acc=0 ? pc + scalar : pc + 1;
brnz        = 5'b10010, //# pc <= acc=0 ? pc + 1 : pc + scalar;
brzdec      = 5'b10011, //# pc <= acc=0 ? pc + scalar : pc + 1; acc <= acc - 1
brnzdec     = 5'b10100, //# pc <= acc=0 ? pc + 1 : pc + scalar; acc <= acc - 1
where       = 5'b10101, //$ boolVect <= cond[operand] ? 1 : 0;
wheren      = 5'b10110, //$ boolVect <= cond[operand] ? 0 : 1;
elsew       = 5'b10111, //$ boolVect <= ~boolVect;
endwhere    = 5'b11000, //$ boolVect <= 1;
search      = 5'b11001, //$ boolVect <= (acc = op) ? 1 : 0;
csearch     = 5'b11010, //$ boolVect <= (acc = op) & (boolVect >> 1) ? 1 : 0;
read        = 5'b11011, //$ boolVect <= boolVect >> 1;
insert      = 5'b11100, //$ insert at first
delete      = 5'b11101, //$ delete first
vload       = 5'b11110, //  vectMem[i][accVect[i]] <= mem[acc + i*contrScalar]
vstore      = 5'b11111; //  mem[acc + i*contrScalar] <= vectMem[i][accVect[i]]
// #: controller only
// $: array only
```

Program Control Section

The program control section of the controller works as it is described in the following:

```
always @(posedge cycle) if (reset)  begin   pc  <= {p{1'b1}}        ;
                                            ir  <= 0                ;
                                    end
                        else        begin   pc  <= nextPc           ;
                                            ir  <= progMem[nextPc]  ;
                                    end
always @(*)
 case(contrOpCode)
    jmp     : nextPc = pc + contrScalar[p-1:0]                                ;
    brz     : nextPc = (acc == 0) ? (pc + contrScalar[p-1:0]) : (pc + 1'b1) ;
    brnz    : nextPc = (acc == 0) ? (pc + 1'b1) : (pc + contrScalar[p-1:0]) ;
    brzdec  : nextPc = (acc == 0) ? (pc + contrScalar[p-1:0]) : (pc + 1'b1) ;
    brnzdec : nextPc = (acc == 0) ? (pc + 1'b1) : (pc + contrScalar[p-1:0]) ;
    default   nextPc = pc + 1'b1                                              ;
 endcase
```

The Execution in Controller and in Cells

Operand selection in controller is described by the following code:

```
always @(*)
    case(contrOperand)  // selects the right operand for controller
        mrl:    op = mem[addr + contrScalar[s-1:0]]             ;
        mri:    op = mem[addr + contrScalar[s-1:0]]             ;
        val:    op = {{(n-8){contrScalar[7]}}, contrScalar}     ;
        rad: begin  op = accVect[0]                             ;
                    for (j=1; j<(1<<x); j=j+1)
                    op = op + accVect[j]                        ;
```

```
            end
        rmx: begin  op = accVect[0]                              ;
                    for (j=1; j<(1<<x); j=j+1)
                    op = (op < accVect[j]) ? accVect[j] : op     ;
            end
        rfl: begin  op = {{(n-1){1'b0}}, boolVect[0]}            ;
                    for (j=1; j<(1<<x); j=j+1)
                    op = {{(n-1){1'b0}}, op[0] | boolVect[j]}    ;
            end
        default     op = mem[contrScalar[s-1:0]]                 ;
    endcase
```

Operand selection in the array's cells is described by the following code:

```
always @(*) for (k=0; k<(1<<x); k=k+1)
 begin
     case(arrayOperand) // selcts the right operand in each cell
        mrl: opVect[k] = vectMem[k][addrVect[k] + arrayScalar[v-1:0]]       ;
        mri:opVect[k] = vectMem[k][addrVect[k] + arrayScalar[v-1:0]]        ;
        val: opVect[k] = {{(n-c){arrayScalar[7]}}, arrayScalar}             ;
        cop: opVect[k] = acc                                                ;
        idx: opVect[k] = k                                                  ;
     default opVect[k] = vectMem[k][arrayScalar[v-1:0]]                     ;
     endcase
    pxVect[k] = (k == 0) ? boolVect[0] : (boolVect[k] | pxVect[k-1])        ;
    firstVect[k] = (k == 0) ?   pxVect[0] : (pxVect[k] & ~pxVect[k-1])      ;
    nextVect[k] = (k == 0) ?    1'b0 : pxVect[k-1]                          ;
    condVect[k] = {nextVect[k], firstVect[k], crVect[k], (accVect[k] == 0)} ;
 end
```

Data operations in controller is performed using as operands the accumulator, `acc`, and data selected by `contrOperand`, as follows:

```
always @(posedge cycle)
    case(contrOpCode)
        add     : {cr, acc} <= acc + op                                 ;
        addc    : {cr, acc} <= acc + op + cr                            ;
        sub     : {cr, acc} <= acc - op                                 ;
        rsub    : {cr, acc} <= op - acc                                 ;
        subc    : {cr, acc} <= acc - op - cr                            ;
        rsubc   : {cr, acc} <= op - acc - cr                            ;
        mult    : {cr, acc} <= {cr, acc * op}                           ;
        load    : {cr, acc} <= {cr, op}                                 ;
        store   : case(contrOperand)
                    mab : mem[contrScalar[s-1:0]]              <= acc   ;
                    mrl : mem[contrScalar[s-1:0] + addr]       <= acc   ;
                    mri : begin mem[contrScalar[s-1:0] + addr] <= acc   ;
                                addr <= contrScalar[s-1:0] + addr       ;
                          end
                    ads : addr <= acc[s-1:0]                            ;
                    default addr <= acc[s-1:0]                          ;
                  endcase
```

```
        bwand   : {cr, acc} <= {cr, acc & op}                          ;
        bwor    : {cr, acc} <= {cr, acc | op}                          ;
        bwxor   : {cr, acc} <= {cr, acc ^ op}                          ;
        insval  : {cr, acc} <= {cr, acc[23:0], contrScalar}            ;
        shrightc: {cr, acc} <= {acc[0], cr, acc[n-1:1]}                ;
        shright : {cr, acc} <= {acc[0], 1'b0, acc[n-1:1]}              ;
        sharight: {cr, acc} <= {acc[0], acc[n-1], acc[n-1:1]}          ;
        vstore  : for (l=0; l<(1<<x); l=l+1)
                    mem[acc + l*contrScalar] <= vectMem[l][accVect[l]] ;
        brzdec  : {cr, acc} <= acc - 1'b1                              ;
        brnzdec : {cr, acc} <= acc - 1'b1                              ;
        default   {cr, acc} <= {cr, acc}                               ;
    endcase
```

Data operations in the array's cells is performed using as operands the accumulator, `accVect[i]`, and data selected by `arrayOperand`, as follows:

```
always @(posedge cycle) for (i=0; i<(1<<x); i=i+1) begin
if (boolVect[i]) begin
 if (arrayOperand == mri)  addrVect[i] <= addrVect[i] + arrayScalar[v-1:0]  ;
 case(arrayOpCode)
    add     : {crVect[i], accVect[i]} <= accVect[i] + opVect[i]             ;
    addc    : {crVect[i], accVect[i]} <= accVect[i] + opVect[i] + crVect[i] ;
    sub     : {crVect[i], accVect[i]} <= accVect[i] - opVect[i]             ;
    rsub    : {crVect[i], accVect[i]} <= opVect[i] - accVect[i]             ;
    subc    : {crVect[i], accVect[i]} <= accVect[i] - opVect[i] - crVect[i] ;
    rsubc   : {crVect[i], accVect[i]} <= opVect[i] - accVect[i] - crVect[i] ;
    mult    : {crVect[i], accVect[i]} <= {crVect[i], accVect[i] * opVect[i]};
    load    : {crVect[i], accVect[i]} <= {crVect[i], opVect[i]}             ;
    store   : case(arrayOperand)
                mab : vectMem[i][arrayScalar[v-1:0]] <= accVect[i]          ;
                mrl : vectMem[i][arrayScalar[v-1:0] + addrVect[i]]
                                     <= accVect[i]                          ;
                mri : vectMem[i][arrayScalar[v-1:0] + addrVect[i]]
                             <= accVect[i]                                  ;
                ads : addrVect[i] <= accVect[i][v-1:0]                      ;
                default addrVect[i] <= addrVect[i]                          ;
              endcase
    bwand   : {crVect[i], accVect[i]} <= {crVect[i], accVect[i] & opVect[i]};
    bwor    : {crVect[i], accVect[i]} <= {crVect[i], accVect[i] | opVect[i]};
    bwxor   : {crVect[i], accVect[i]} <= {crVect[i], accVect[i] ^ opVect[i]};
    gshift  : accVect[i] <= accVect[(i + arrayScalar[x-1:0])]               ;
    shrightc: {crVect[i], accVect[i]} <=
                {accVect[i][0], crVect[i], accVect[i][n-1:1]}               ;
    shright : {crVect[i], accVect[i]} <=
                {accVect[i][0], 1'b0, accVect[i][n-1:1]}                    ;
    sharight: {crVect[i], accVect[i]} <=
                {accVect[i][0], accVect[i][n-1], accVect[i][n-1:1]}         ;
    vload   : vectMem[i][accVect[i]] <= mem[acc + i*contrScalar]            ;
    default   {crVect[i], accVect[i]} <= {crVect[i], accVect[i]}            ;
 endcase
end
```

```
    case(arrayOpCode)
        where   : boolVect[i] <= (condVect[i][arrayOperand[1:0]]) ? 1'b1 : 1'b0 ;
        wheren  : boolVect[i] <= (condVect[i][arrayOperand[1:0]]) ? 1'b0 : 1'b1 ;
        elsew   : boolVect[i] <= ~boolVect[i]                                   ;
        endwhere: boolVect[i] <= 1'b1                                           ;
        default   boolVect[i] <= boolVect[i]                                    ;
    endcase
    end
endmodule
```

Spatial selection allows to decide, according to the internal state of each cell, if the current instruction is executed or not in `cell[i]`. The spatial selection instructions act on the content of the Boolean vector, as follows:

```
    always @(posedge cycle) for (i=0; i<(1<<x); i=i+1)
        case(arrayOpCode)
        where   : boolVect[i] <= (condVect[i][arrayOperand[1:0]]) ? 1'b1 : 1'b0 ;
        wheren  : boolVect[i] <= (condVect[i][arrayOperand[1:0]]) ? 1'b0 : 1'b1 ;
        elsew   : boolVect[i] <= ~boolVect[i]                                   ;
        endwhere: boolVect[i] <= 1'b1                                           ;
        default   boolVect[i] <= boolVect[i]                                    ;
    endcase
```

Vector transfer instructions are used to exchange data between the vector memory distributed in the array of cell and the controller's memory. The transfer is strided with stride given by `contrScalar`. The definitions are:

```
    always @(posedge cycle) for (i=0; i<(1<<x); i=i+1)
     begin
      if (arrayOpCode == vload)  vectMem[i][accVect[i]] <= mem[acc + i*contrScalar];
      if (contrOpCode == vstore) mem[acc + l*contrScalar] <= vectMem[l][accVect[l]];
     end
```

9.2.2 Assembler Programming the Generic ConnexArray^{TM}

Each line of program must contain code for both instructions: the instruction issued for the array and the instruction performed by the controller. For conditioned or unconditioned relative jumps in program some lines are labeled; `LB(i)` denote the label *i*. The use of the label is indicated by by the value used by control instructions (example: `cJMP(2)`).

Example 9.1 *The program which compute in the accumulator of the controller the inner product of the index vector* `ix` *with itself is:*

```
        cNOP;       ENDWHERE;     // activate all cells
        cNOP;       IXLOAD;       // accVect[i] <= ixVect[i]
        cNOP;       IXMULT;       // accVect[i] <= accVect[i] * ixVect[i]
        cRSLOAD;    NOP;          // load acc with the reduction sum
        cHALT;      NOP;
```

The content of program memory is:

```
progMem[0] = 11000000000000000000000000000000
progMem[1] = 00111111000000000000000000000000
progMem[2] = 00110111000000000000000000000000
progMem[3] = 00000000000000000011110100000000
progMem[4] = 00000000000000001000000000000000
```

The result of simulation:

```
time=0  rst=1 pc= x acc=x    aVect[0]=x aVect[1]=x aVect[2]=x ... aVect[15]=  x bVect=xxxx_xxxx
time=1  rst=1 pc=31 acc=x    aVect[0]=x aVect[1]=x aVect[2]=x ... aVect[15]=  x bVect=xxxx_xxxx
time=3  rst=0 pc= 0 acc=x    aVect[0]=x aVect[1]=x aVect[2]=x ... aVect[15]=  x bVect=xxxx_xxxx
time=5  rst=0 pc= 1 acc=x    aVect[0]=x aVect[1]=x aVect[2]=x ... aVect[15]=  x bVect=1111_1111
time=7  rst=0 pc= 2 acc=x    aVect[0]=0 aVect[1]=1 aVect[2]=2 ... aVect[15]= 15 bVect=1111_1111
time=9  rst=0 pc= 3 acc=x    aVect[0]=0 aVect[1]=1 aVect[2]=4 ... aVect[15]=225 bVect=1111_1111
time=11 rst=0 pc= 4 acc=1240 aVect[0]=0 aVect[1]=1 aVect[2]=4 ... aVect[15]=225 bVect=1111_1111
```

◇

Example 9.2 *The program which load the accumulator the index in each cell, stores it incremented in 12 successive addresses starting from the address 2, than add in accumulator the stored values. The program is:*

```
        cVLOAD(12);     ENDWHERE;    // acc <= 12; activate all cells
        cNOP;           VLOAD(2);    // accVect[i] <= 2
        cNOP;           ADDRLD;      // addrVect[i] <= accVect[i]
        cNOP;           IXLOAD;      // accVect[i] <= index
LB(1);  cNOP;           RISTORE(1);  // vectMem[i][addrVect + 1] <= accVect[i];
                                     // addrVect <= addrVect + 1
        cBRNZDEC(1);    VADD(1);     // if (acc !== 0) branch to LB(1); acc <= acc - 1;
                                     // accVect[i] <= accVect[i] + 1;
        cVLOAD(13);     VLOAD(2);    // acc <= 13; accVect[i] <= 2
        cNOP;           ADDRLD;      // addrVect[i] <= accVect[i]
        cNOP;           VLOAD(0);    // accVect[i] <= 0
LB(2);  cBRNZDEC(2);    RIADD(1);    // if (acc !== 0) branch to LB(2); acc <= acc - 1;
                                     // accVect[i] <=
                                     // accVect[i] + vectMem[i][addrVect + 1];
                                     // addrVect <= addrVect + 1
        cHALT;          NOP;         // halt
```

◇

9.3 Search Oriented ConnexArray^{TM}: the Second Global Loop

The global loop closed in the previous section sends back to the array the same data for each cell. A new feature is added when another loop sends back to the array specific, differentiated information for each cell. Let us start with a very simple function associated to this second global loop: it takes the Boolean vector `boolVect[0:(1<<x)-1]` distributed along the array of cells and sends back two Boolean vectors:

- `firstVect[0:(1<<x)-1]`: with 1 only on the position of the first occurrence of 1 in `boolVect`; it is used to indicate the first active cell
- `nextVect[0:(1<<x)-1]`: with 1 in all the positions next to the 1 in the `firstVect` Boolean vector; it is used to indicate all the cells next to the first active cell

There are 5 instructions supported by this second global loop:

```
always @(posedge cycle) for (i=0; i<(1<<x); i=i+1) begin
    case(arrayOpCode)
        search  : boolVect[i] <= (accVect[i] == opVect[i]) ? 1'b1 : 1'b0        ;
        csearch : boolVect[i] <= (i == 0) ? 1'b0 :
                    (((accVect[i] == opVect[i]) & boolVect[i-1]) ? 1'b1 :
                    1'b0)                                                       ;
        read    : boolVect[i] <= (i == 0) ? 1'b0 : boolVect[i-1]                ;
        insert  : begin
                    if (firstVect[i])
                        accVect[i] <= opVect[i]                                 ;
                     else if (nextVect[i])
                            accVect[i] <= (i == 0) ? accVect[i] : accVect[i-1]  ;
                  end
        delete  : if (firstVect[i] | nextVect[i])
                    accVect[i] <= (i == ((1<<x)-1)) ? 0 : accVect[i+1]          ;
        default   boolVect[i] <= boolVect[i]                                    ;
    endcase
    end
```

The instruction `search` identifies all the positions in array where the accumulator has a certain value, while the `csearch` supports the search of a certain string of values.

Example 9.3 *The program which load the index in each* `acc[i]`*, identifies the occurrence of the stream* `<1 2 3>` *in* `accVect` *and adds in* `acc` *the next four numbers:*

```
            cNOP;           ENDWHERE; // set active all cells
            cVLOAD(1);      IXLOAD;   // acc = 1; load index in each cell
            cVLOAD(2);      SEARCH;   // acc = 2; search 'acc' in each acc[i]
            cVLOAD(3);      CSEARCH;  // acc = 3; search 'acc' after each active cell
            cNOP;           CSEARCH;  // search 'acc' after each active cell
            cNOP;           READ;     // boolVect >> 1
            cRSLOAD;        READ;     // acc = reduceSum; boolVect >> 1
            cRSADD;         READ;     // acc = acc + reduceSum; boolVect >> 1
            cRSADD;         READ;     // acc = acc + reduceSum; boolVect >> 1
            cRSADD;         NOP;      // acc = acc + reduceSum
            cHALT;          NOP;      // halt
```

```
time=0  rst=1 pc= x acc=x  accVect[0]=x accVect[1]=x accVect[2]=x accVect[3]=x bVect=xxxx_xxxx
time=1  rst=1 pc=31 acc=x  accVect[0]=x accVect[1]=x accVect[2]=x accVect[3]=x bVect=xxxx_xxxx
time=3  rst=0 pc= 0 acc=x  accVect[0]=x accVect[1]=x accVect[2]=x accVect[3]=x bVect=xxxx_xxxx
time=5  rst=0 pc= 1 acc=x  accVect[0]=x accVect[1]=x accVect[2]=x accVect[3]=x bVect=1111_1111
time=7  rst=0 pc= 2 acc=1  accVect[0]=0 accVect[1]=1 accVect[2]=2 accVect[3]=3 bVect=1111_1111
time=9  rst=0 pc= 3 acc=2  accVect[0]=0 accVect[1]=1 accVect[2]=2 accVect[3]=3 bVect=0100_0000
time=11 rst=0 pc= 4 acc=3  accVect[0]=0 accVect[1]=1 accVect[2]=2 accVect[3]=3 bVect=0010_0000
time=13 rst=0 pc= 5 acc=3  accVect[0]=0 accVect[1]=1 accVect[2]=2 accVect[3]=3 bVect=0001_0000
time=15 rst=0 pc= 6 acc=3  accVect[0]=0 accVect[1]=1 accVect[2]=2 accVect[3]=3 bVect=0000_1000
time=17 rst=0 pc= 7 acc=4  accVect[0]=0 accVect[1]=1 accVect[2]=2 accVect[3]=3 bVect=0000_0100
time=19 rst=0 pc= 8 acc=9  accVect[0]=0 accVect[1]=1 accVect[2]=2 accVect[3]=3 bVect=0000_0010
time=21 rst=0 pc= 9 acc=15 accVect[0]=0 accVect[1]=1 accVect[2]=2 accVect[3]=3 bVect=0000_0001
time=23 rst=0 pc=10 acc=22 accVect[0]=0 accVect[1]=1 accVect[2]=2 accVect[3]=3 bVect=0000_0001
```

◇

Example 9.4 *The program which load the index in each* `acc[i]`*, identifies the occurrence(s) of the stream* `<0 1>` *in* `accVect` *and insert in the vector* `accVect` *the sequence* `<13 15>`.

```
cNOP;          ENDWHERE;     // set active all cells
cNOP;          IXLOAD;       // load index in each cell
cNOP;          VSEARCH(1);   // search '0' in each acc[i]
cNOP;          VCSEARCH(2);  // search '1' after each active cell
cNOP;          READ;         // boolVect >> 1
cNOP;          INSERT(13);   // insert '13' in the first active cell
cNOP;          INSERT(15);   // insert '15' in the first active cell
cHALT;         NOP;          // halt
```

◇

The second global loop adds specific features which support search applications, sparse matrix/vector operations, ...

9.4 Problems

Problem 9.1

Problem 9.2

Problem 9.3

Part I

ANNEXES

Appendix A

Designing a simple CISC processor

A.1 The project

```
cisc_processor.v
    cisc_alu.v
    control_automaton.v
    mux2.v
    mux4.v
        mux2.v
    register_file.v
```

A.2 RTL code

The module `cisc_processor.v`

```
module cisc_processor(input                clock   ,
                      input                reset   ,
                      output    reg [31:0] addr_reg,   // memory address
                      output    reg [1:0]  com_reg ,   // memory command
                      output    reg [31:0] out_reg ,   // data output
                      input         [31:0] in      );  // data/inst input
 // INTERNAL CONNECTIONS
    wire    [25:0]  command;
    wire            flag;
    wire    [31:0]  alu_out, left, right, left_out, right_out;
 // INPUT & OUTPUT BUFFER REGISTERS
    reg [31:0]  data_reg, inst_reg;
    always @(posedge clock) begin   if (command[25]) inst_reg <= in;
                                    data_reg <= in;
                                    addr_reg <= left_out    ;
                                    out_reg  <= right_out   ;
                                    com_reg  <= command[1:0];
                            end
 // CONTROL AUTOMATON
```

```
    control_automaton control_automaton(.clock  (clock          ),
                                        .reset  (reset          ),
                                        .inst   (inst_reg[31:11]),
                                        .command(command        ),
                                        .flag   (alu_out[0]     ));
 // REGISTER FILE
    register_file register_file(.left_out       (left_out       ),
                                .right_out      (right_out      ),
                                .result         (alu_out        ),
                                .left_addr      (command[18:14] ),
                                .right_addr     (command[9:5]   ),
                                .dest_addr      (command[23:19] ),
                                .write_enable   (command[24]    ),
                                .clock          (clock          ));
 // MULTIPLEXERS
    mux2 left_mux(  .out(left       ),
                    .in0(left_out   ),
                    .in1(data_reg   ),
                    .sel(command[4]));
    mux4 right_mux( .out(right                                      ),
                    .in0(right_out                                  ),
                    .in1({{21{inst_reg[10]}}, inst_reg[10:0]}       ),
                    .in2({16'b0, inst_reg[15:0]}                    ),
                    .in3({inst_reg[15:0], 16'b0}                    ),
                    .sel(command[3:2]                               ));
 // ARITHMETIC & LOGIC UNIT
    cisc_alu alu(   .alu_out(alu_out        ),
                    .left   (left           ),
                    .right  (right          ),
                    .alu_com(command[13:10] ));
endmodule
```

The module `cisc_alu.v`

```
module cisc_alu(output  reg [31:0]  alu_out ,
                input       [31:0]  left    ,
                input       [31:0]  right   ,
                input       [3:0]   alu_com );
    wire    [32:0]  add, sub;

    assign  add = left + right,
            sub = left - right;

    always @(alu_com or left or right or add or sub)
        case(alu_com)
            4'b0000: alu_out = left                     ;
```

```
            4'b0001: alu_out = right                    ;
            4'b0010: alu_out = left + 1                 ;
            4'b0011: alu_out = left - 1                 ;
            4'b0100: alu_out = add[31:0]                ;
            4'b0101: alu_out = sub[31:0]                ;
            4'b0110: alu_out = {1'b0, left[31:1]}       ;
            4'b0111: alu_out = {left[31], left[31:1]}   ;
            4'b1000: alu_out = {31'b0, (left == 0)}     ;
            4'b1001: alu_out = {31'b0, (left == right)} ;
            4'b1010: alu_out = {31'b0, (left < right)}  ;
            4'b1011: alu_out = {31'b0, add[32]}         ;
            4'b1100: alu_out = {31'b0, sub[32]}         ;
            4'b1101: alu_out = left & right             ;
            4'b1110: alu_out = left | right             ;
            4'b1111: alu_out = left ^ right             ;
        endcase
endmodule
```

The module `control_automaton.v`

```
module control_automaton(   input           clock   ,
                            input           reset   ,
                            input   [31:11] inst    ,
                            output  [25:0]  command ,
                            input           flag    );

 // THE STRUCTURE OF 'inst'
    wire    [5:0]   opcode  ;   // operation code
    wire    [4:0]   dest    ,   // selects destination register
                    left_op ,   // selects left operand register
                    right_op;   // selects right operand register
    assign {opcode, dest, left_op, right_op} = inst;

 // THE STRUCTURE OF 'command'
    reg       en_inst     ; // enable load a new instruction in inst_reg
    reg       write_enable; // writes the output of alu at dest_addr
    reg [4:0] dest_addr   ; // selects the destination register
    reg [4:0] left_addr   ; // selects the left operand in file register
    reg [3:0] alu_com     ; // selects the operation performed by the alu
    reg [4:0] right_addr  ; // selects the right operand in file register
    reg       left_sel    ; // selects the source of the left operand
    reg [1:0] right_sel   ; // selects the source of the right operand
    reg [1:0] mem_com     ; // generates the command for memory
    assign command = {en_inst, write_enable, dest_addr, left_addr,
                      alu_com, right_addr, left_sel, right_sel, mem_com};
```

```
// MICRO-ARCHITECTURE
   // en_inst
   parameter
   no_load     = 1'b0, // disable instruction register
   load_inst   = 1'b1; // enable instruction register
   // write_enable
   parameter
   no_write    = 1'b0,
   write_back  = 1'b1; // write back the current ALU output
   // alu_func
   parameter
   alu_left    = 4'b0000, // alu_out = left
   alu_right   = 4'b0001, // alu_out = right
   alu_inc     = 4'b0010, // alu_out = left + 1
   alu_dec     = 4'b0011, // alu_out = left - 1
   alu_add     = 4'b0100, // alu_out = left + right
   alu_sub     = 4'b0101, // alu_out = left - right
   alu_shl     = 4'b0110, // alu_out = {1'b0, left[31:1]}
   alu_half    = 4'b0111, // alu_out = {left[31], left[31:1]}
   alu_zero    = 4'b1000, // alu_out = {31'b0, (left == 0)}
   alu_equal   = 4'b1001, // alu_out = {31'b0, (left == right)}
   alu_less    = 4'b1010, // alu_out = {31'b0, (left < right)}
   alu_carry   = 4'b1011, // alu_out = {31'b0, add[32]}
   alu_borrow  = 4'b1100, // alu_out = {31'b0, sub[32]}
   alu_and     = 4'b1101, // alu_out = left & right
   alu_or      = 4'b1110, // alu_out = left | right
   alu_xor     = 4'b1111; // alu_out = left ^ right
   // left_sel
   parameter
   left_out    = 1'b0, // left out of the reg file as left op
   from_mem    = 1'b1; // data from memory as left op
   // right_sel
   parameter
   right_out   = 2'b00, // right out of the reg file as right op
   jmp_addr    = 2'b01, // right op = {{22{inst[10]}}, inst[10:0]}
   low_value   = 2'b10, // right op = {{16{inst[15]}}, inst[15:0]}
   high_value  = 2'b11; // right op = {inst[15:0], 16'b0}
   // mem_com
   parameter
   mem_nop     = 2'b00,
   mem_read    = 2'b10, // read from memory
   mem_write   = 2'b11; // write to memory

// INSTRUCTION SET ARCHITECTURE (only samples)
   // arithmetic & logic instructions & pc = pc + 1
   parameter
```

```
   move    = 6'b10_0000, // dest_reg = left_out
   inc     = 6'b10_0010, // dest_reg = left_out + 1
   dec     = 6'b10_0011, // dest_reg = left_out - 1
   add     = 6'b10_0100, // dest_reg = left_out + right_out
   sub     = 6'b10_0101, // dest_reg = left_out - right_out
   bwxor   = 6'b10_1111; // dest_reg = left_out ^ right_out
   // ...
   // data move instructions & pc = pc + 1
   parameter
   read    = 6'b01_0000, // dest_reg = mem(left_out)
   rdinc   = 6'b01_0001, // dest_reg = mem(left_out + value)
   write   = 6'b01_1000, // mem(left_out) = right_out
   wrinc   = 6'b01_1001; // mem(left_out + value) = right_out
   // ...
   // control instructions
   parameter
   nop     = 6'b11_0000, // pc = pc + 1
   jmp     = 6'b11_0001, // pc = pc + value
   call    = 6'b11_0010, // pc = value, ra = pc + 1
   ret     = 6'b11_0011, // pc = ra
   jzero   = 6'b11_0100, // if (left_out = 0) pc = pc + value;
                         // else pc = pc + 1
   jnzero  = 6'b11_0101; // if (left_out != 0) pc = pc + value;
                         // else pc = pc + 1
   // ...

// THE STATE REGISTER
   reg     [5:0]   state_reg   ; // the state register
   reg     [5:0]   next_state  ; // a "register" used as variable
   always @(posedge clock) if (reset)  state_reg <= 0          ;
                           else        state_reg <= next_state ;

// THE CONTROL AUTOMATON'S LOOP
   always @(state_reg or opcode or dest or left_op or right_op or flag)
       begin   en_inst      = 1'bx;//no_load       ;
               write_enable = 1'bx;//no_write  ;
               dest_addr    = 5'bxxxxx     ;
               left_addr    = 5'bxxxxx     ;
               alu_com      = 4'bxxxx      ;
               right_addr   = 5'bxxxxx     ;
               left_sel     = 1'bx         ;
               right_sel    = 2'bxx        ;
               mem_com      = 2'bxx;//mem_nop      ;
               next_state   = 6'bxxxxxx;//state_reg + 1;
           // INITIALIZE THE PROCESSOR
           if (state_reg == 6'b00_0000)
```

```
// pc = 0
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = 5'b11111     ;
            left_addr    = 5'b11111     ;
            alu_com      = alu_xor      ;
            right_addr   = 5'b11111     ;
            left_sel     = left_out     ;
            right_sel    = right_out    ;
            mem_com      = mem_nop      ;
            next_state   = state_reg + 1;
        end
// INSTRUCTION FETCH
if (state_reg == 6'b00_0001)
// rquest for a new instruction & increment pc
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = 5'b11111     ;
            left_addr    = 5'b11111     ;
            alu_com      = alu_inc      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = left_out     ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_read     ;
            next_state   = state_reg + 1;
        end
if (state_reg == 6'b00_0010)
// wait for memory to read doing nothing
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = 4'bxxxx      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = 1'bx         ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = state_reg + 1;
        end
if (state_reg == 6'b00_0011)
// load the new instruction in instr_reg
        begin
            en_inst      = load_inst    ;
```

```
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = 4'bxxxx      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = 1'bx         ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = state_reg + 1;
        end
    if (state_reg == 6'b00_0100)
    // initialize the control automaton
        begin
            en_inst      = load_inst    ;
            write_enable = write_back   ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = 4'bxxxx      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = 1'bx         ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = opcode[5:0]  ;
        end
    // EXECUTE THE ONE CYCLE FUNCTIONAL INSTRUCTIONS
    if (state_reg[5:4] == 2'b10)
    // dest = left_op OPERATION right_op
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = dest         ;
            left_addr    = left_op      ;
            alu_com      = opcode[3:0]  ;
            right_addr   = right_op     ;
            left_sel     = left_out     ;
            right_sel    = right_out    ;
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
    // EXECUTE MEMORY READ INSTRUCTIONS
    if (state_reg == 6'b01_0000)
    // read from left_reg in dest_reg
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
```

```
            left_addr    = left_op      ;
            alu_com      = alu_left     ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = left_out     ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_read     ;
            next_state   = 6'b01_0010   ;
        end
    if (state_reg == 6'b01_0001)
    // read from left_reg + <value> in dest_reg
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = left_op      ;
            alu_com      = alu_add      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = left_out     ;
            right_sel    = low_value    ;
            mem_com      = mem_read     ;
            next_state   = 6'b01_0010   ;
        end
    if (state_reg == 6'b01_0010)
    // wait for memory to read doing nothing
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = 4'bxxxx      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = 1'bx         ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = state_reg + 1;
        end
    if (state_reg == 6'b01_0011)
    // the data from memory is loaded in data_reg
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = 4'bxxxx      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = 1'bx         ;
```

```
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = state_reg + 1;
        end
    if (state_reg == 6'b01_0100)
    // data_reg is loaded in dest_reg & go to fetch
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = dest         ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = alu_left     ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = from_mem     ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
    // EXECUTE MEMORY WRITE INSTRUCTIONS
    if (state_reg == 6'b01_1000)
    // write right_op to left_op & go to fetch
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = left_op      ;
            alu_com      = alu_left     ;
            right_addr   = right_op     ;
            left_sel     = left_out     ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_write    ;
            next_state   = 6'b00_0001   ;
        end
    if (state_reg == 6'b01_1000)
    // write right_op to left_op + <value>  & go to fetch
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = left_op      ;
            alu_com      = alu_add      ;
            right_addr   = right_op     ;
            left_sel     = left_out     ;
            right_sel    = low_value    ;
            mem_com      = mem_write    ;
            next_state   = 6'b00_0001   ;
```

```
        end
// CONTROL INSTRUCTIONS
if (state_reg == 6'b11_0000)
// no operation & go to fetch
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = 4'bxxxx      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = 1'bx         ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
if (state_reg == 6'b11_0001)
// jump to (pc + <value>) & go to fetch
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = 5'b11111     ;
            left_addr    = 5'b11111     ;
            alu_com      = alu_add      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = left_out     ;
            right_sel    = low_value    ;
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
if (state_reg == 6'b11_0010)
// call: first step: ra = pc + 1
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = 5'b11110     ;
            left_addr    = 5'b11111     ;
            alu_com      = alu_left     ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = left_out     ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = 6'b11_0110;
        end
if (state_reg == 8'b0011_0110)
// call: second step: pc = value
```

```
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = 5'b11111     ;
            left_addr    = 5'bxxxxx     ;
            alu_com      = alu_right    ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = 1'bx         ;
            right_sel    = jmp_addr     ;
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
if (state_reg == 6'b11_0011)
// ret: pc = ra
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = 5'b11111     ;
            left_addr    = 5'b11110     ;
            alu_com      = alu_left     ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = left_out     ;
            right_sel    = 2'bxx        ;
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
if ((state_reg == 6'b11_0100) && flag)
// jzero: if (left_out = 0) pc = pc + value;
        begin
            en_inst      = no_load      ;
            write_enable = write_back   ;
            dest_addr    = 5'b11111     ;
            left_addr    = 5'b11111     ;
            alu_com      = alu_add      ;
            right_addr   = 5'bxxxxx     ;
            left_sel     = left_out     ;
            right_sel    = low_value    ;
            mem_com      = mem_nop      ;
            next_state   = 6'b00_0001   ;
        end
if ((state_reg == 6'b11_0100) && ~flag)
// jzero: if (left_out = 1) pc = pc + 1;
        begin
            en_inst      = no_load      ;
            write_enable = no_write     ;
            dest_addr    = 5'bxxxxx     ;
```

```
                    left_addr    = 5'bxxxxx     ;
                    alu_com      = 4'bxxxx      ;
                    right_addr   = 5'bxxxxx     ;
                    left_sel     = 1'bx         ;
                    right_sel    = 2'bxx        ;
                    mem_com      = mem_nop      ;
                    next_state   = 6'b00_0001   ;
                end
            if ((state_reg == 6'b11_0100) && ~flag)
            // jnzero: if (left_out = 1) pc = pc + value;
                begin
                    en_inst      = no_load      ;
                    write_enable = write_back   ;
                    dest_addr    = 5'b11111     ;
                    left_addr    = 5'b11111     ;
                    alu_com      = alu_add      ;
                    right_addr   = 5'bxxxxx     ;
                    left_sel     = left_out     ;
                    right_sel    = low_value    ;
                    mem_com      = mem_nop      ;
                    next_state   = 6'b00_0001   ;
                end
            if ((state_reg == 6'b11_0100) && flag)
            // jnzero: if (left_out = 0) pc = pc + 1;
                begin
                    en_inst      = no_load      ;
                    write_enable = no_write     ;
                    dest_addr    = 5'bxxxxx     ;
                    left_addr    = 5'bxxxxx     ;
                    alu_com      = 4'bxxxx      ;
                    right_addr   = 5'bxxxxx     ;
                    left_sel     = 1'bx         ;
                    right_sel    = 2'bxx        ;
                    mem_com      = mem_nop      ;
                    next_state   = 6'b00_0001   ;
                end
        end
endmodule
```

The module `register_file.v`

```
module register_file(   output  [31:0]  left_out    ,
                        output  [31:0]  right_out   ,
                        input   [31:0]  result      ,
                        input   [4:0]   left_addr   ,
                        input   [4:0]   right_addr  ,
```

```
                        input   [4:0]   dest_addr   ,
                        input           write_enable,
                        input           clock       );

    reg [31:0]  file[0:31];

    assign  left_out    = file[left_addr]   ,
            right_out   = file[right_addr]  ;

    always @(posedge clock) if (write_enable) file[dest_addr] <= result;

endmodule
```

The module `mux4.v`

```
module mux4(output  [31:0]  out,
            input   [31:0]  in0,
            input   [31:0]  in1,
            input   [31:0]  in2,
            input   [31:0]  in3,
            input   [1:0]   sel);
    wire[31:0]      out1, out0;
    mux2    mux(out, out0, out1, sel[1]);
    mux2    mux1(out1, in2, in3, sel[0]),
            mux0(out0, in0, in1, sel[0]);
endmodule
```

The module `mux2.v`

```
module mux2(output  [31:0]  out,
            input   [31:0]  in0,
            input   [31:0]  in1,
            input           sel);
    assign  out = sel ? in1 : in0;
endmodule
```

A.3 Testing `cisc_processor.v`

Appendix B

Meta-stability

Any asynchronous signal applied the the input of a clocked circuit is a source of *meta-stability* [webRef_1] [Alfke '05]. There is a **dangerous timing window** "centered" on the clock transition edge specified by the sum of *set-up time*, *edge transition time* and *hold time*. If the data input of a D-FF switches in this window, then there are three possible behaviors for its output:

- the output does not change according to the change on the flip-flop's input (the flip-flop does not catch the input variation)
- the output change according to the change on the flip-flop's input (the flip-flop catches the input variation)
- the output goes meta-stable for t_{MS}, then goes unpredictable in 1 or 0 (see the wave forms [webRef_2]).

Bibliography

[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.

[Blakeslee '79] T. R. Blakeslee: *Digital Design with Standard MSI and LSI*, John Wiley & Sons, 1979.

[Booth '67] T. L. Booth: *Sequential Machines and Automata Theory*, John Wiley & Sons, Inc., 1967.

[Bremermann '62] H. G. Bremermann: "Optimization through Evolution and Recombination", in *Self-Organizing Systems*, ed.: M. C. Yovits, S. Cameron, Washington DC, Spartan, 1962.

[Calude '82] Cristian Calude: *Complexitatea calculului. Aspecte calitative* (The Complexity of Computation. Qualitative Aspects), Ed. Stiintifica si Enciclopedica, Bucuresti, 1982.

[Calude '94] Cristian Calude: *Information and Randomness*, Springer-Verlag, 1994.

[Casti '92] John L. Casti: *Reality Rules: II. Picturing the World in Mathematics - The Frontier*, John Wiley & Sons, Inc., 1992.

[Cavanagh '07] Joseph Cavanagh: *Sequential Logic. Analysis and Synthesis*, CRC Taylor & Francis, 2007.

[Chaitin '66] Gregory Chaitin: "On the Length of Programs for Computing Binary Sequences", *J. of the ACM*, Oct., 1966.

[Chaitin '70] Gregory Chaitin: "On the Difficulty of Computation", in *IEEE Transactions of Information Theory*, ian. 1970.

[Chaitin '77] Gregory Chaitin: "Algorithmic Information Theory", in *IBM J. Res. Develop.*, Iulie, 1977.

[Chaitin '87] Gregory Chaitin: *Algorithmic Information Theory*, Cambridge University Press, 1987.

[Chaitin '90] Gregory Chaitin: *Information, Randomness and Incompletness*, World Scientific,1990.

[Chaitin '94] Gregory Chaitin: *The Limits of Mathematics IV*, IBM Research Report RC 19671, e-print chao-dyn/9407009, July 1994.

[Chaitin '06] Gregory Chaitin: "The Limit of Rason", in *Scientific American*, Martie, 2006.

[Chomsky '56] Noam Chomsky, "Three Models for the Description of Languages", *IEEE Trans. on Information Theory*, 2:3 , 1956.

[Chomsky '59] Noam Chomsky, "On Certain Formal Properties of Grammars", *Information and Control*, 2:2, 1959.

[Chomsky '63] Noam Chomsky, "Formal Properties of Grammars", *Handbook of Mathematical Psychology*, Wiley, New-York, 1963.

[Church '36] Alonzo Church: "An Unsolvable Problem of Elementary Number Theory", in *American Journal of Mathematics*, vol. 58, pag. 345-363, 1936.

[Clare '72] C. Clare: *Designing Logic Systems Using State Machines*, Mc Graw-Hill, Inc., 1972.

[Cormen '90] Thomas H. Cormen, Charles E. Leiserson, Donsld R. Rivest: *Introduction to Algorithms*, MIT Press, 1990.

[Dascălu '98] Monica Dascălu, Eduard Franţi, Gheorghe Ştefan: "Modeling Production with Artificial Societies: the Emergence of Social Structure", in S. Bandini, R. Serra, F. Suggi Liverani (Eds.): *Cellular Automata: Research Towars Industry. ACRI '98 - Proceedings of the Third Conference on Cellular Automata for Research and Industry*, Trieste, 7 - 9 October 1998, Springer Verlag, 1998. p 218 - 229.

[Dascălu '98a] Monica Dascălu, Eduard Franţi, Gheorghe Ştefan: "Artificial Societies: a New Paradigm for Complex Systems' Modeling", in *IFAC Conference on Supplemental Ways for Improving International Stability - SWIIIS '98*, May 14-16, Sinaia, 1998. p.62-67.

[Drăgănescu '84] Mihai Drăgănescu: "Information, Heuristics, Creation", in Plauder, I. (ed): *Artificial Inteligence and Information Control System of Robots*, Elsevier Publishers B. V. (North-Holland), 1984.

[Drăgănescu '91] Mihai Drăgănescu, Gheorghe Ştefan, Cornel Burileanu: *Electronica functională*, Ed. Tehnică, Bucureşti, 1991 (in Roumanian).

[Einspruch '86] N. G. Einspruch ed.: *VLSI Electronics. Microstructure Science. vol. 14 : VLSI Design*, Academic Press, Inc., 1986.

[Einspruch '91] N. G. Einspruch, J. L. Hilbert: *Application Specific Integrated Circuits (ASIC) Technology*, Academic Press, Inc., 1991.

[Ercegovac '04] Miloš D. Ercegovac, Tomás Lang: *Digital Arithmetic*, Morgan Kaufman, 2004.

[Flynn '72] Flynn, M.J.: "Some computer organization and their affectiveness", *IEEE Trans. Comp.* C21:9 (Sept. 1972), pp. 948-960.

[Gheolbanoiu '14] Alexandru Gheolbanoiu, Dan Mocanu, Radu Hobincu, Lucian Petrica: "Cellular Automaton pRNG with a Global Loop for Non-Uniform Rule Control", 18th International Conference on Ciruits, Systems, Communications and Computers (CSCC 2014), Santorini Island, Greece, July 17-21, 2014, vol. II, 415-420. `http://www.euroment.org/library/2014/santorini/bypaper/COMPUTERS/COMPUTERS2-15.pdf`

[Glushkov '66] V. M. Glushkov: *Introduction to Cybernetics*, Academic Press, 1966.

[Gödels '31] Kurt Gödel: "On Formally Decidable Propositions of Principia Mathematica and Related Systems I", reprinted in S. Fefermann et all.: *Collected Works I: Publications 1929 - 1936,* Oxford Univ. Press, New York, 1986.

[Hartley '95] Richard I. Hartley: *Digit-Serial Computation*, Kulwer Academic Pub., 1995.

[Hascsi '95] Zoltan Hascsi, Gheorghe Ştefan: "The Connex Content Addressable Memory (C^2AM)", *Proceedings of the Twenty-first European Solid-State Circuits Conference*, Lille -France, 19-21 September 1995, pp. 422-425.

[Hascsi '96] Zoltan Hascsi, Bogdan Mîţu, Mariana Petre, Gheorghe Ştefan, "High-Level Synthesis of an Enchanced Connex memory", in *Proceedings of the International Semiconductor Conference*, Sinaia, October 1996, p. 163-166.

[Head '87] T. Head: "Formal Language Theory and DNA: an Analysis of the Generative Capacity of Specific Recombinant Behaviours", in *Bull. Math. Biology*, 49, p. 737-759, 1987.

[Helbing 89'] Walter A. Helbing, Veljko M. Milutinovic: "Architecture and Design of a 32-bit GaAs Microprocessor", in [Milutinovic 89'].

[Hennessy '07] John L. Hennessy, David A. Patterson: *Computer Architecture: A Quantitative Approach*, Fourth Edition, Morgan Kaufmann, 2007.

[Hennie '68] F. C. Hennie: *Finite-State Models for Logical Machine*, John Wiley & Sons, Inc., 1968.

[Hillis '85] W. D. Hillis: *The Connection Machine*, The MIT Press, Cambridge, Mass., 1985.

[Kaeslin '01] Hubert Kaeslin: *Digital Integrated Circuit Design*, Cambridge Univ. Press, 2008.

[Keeth '01] Brent Keeth, R. jacob Baker: *DRAM Circuit Design. A Tutorial*, IEEE Press, 2001.

[Kleene '36] Stephen C. Kleene: "General Recursive Functions of Natural Numbers", in *Math. Ann*., 112, 1936.

[Karim '08] Mohammad A. Karim, Xinghao Chen: *Digital Design*, CRC Press, 2008.

[Knuth '73] D. E. Knuth: *The Art of Programming. Sorting and Searching*, Addison-Wesley, 1973.

[Kolmogorov '65] A.A. Kolmogorov: "Three Approaches to the Definition of the Concept "Quantity of Information" ", in *Probl. Peredachi Inform.*, vol. 1, pag. 3-11, 1965.

[Kung '79] H. T. Kung, C. E. Leiserson: "Algorithms for VLSI processor arrays", in [Mead '79].

[Ladner '80] R. E. ladner, M. J. Fischer: "Parallel prefix computation", *J. ACM*, Oct. 1980.

[Lindenmayer '68] Lindenmayer, A.: "Mathematical Models of Cellular Interactions in Development I, II", *Journal of Theor. Biology*, 18, 1968.

[Maliţa '06] Mihaela Maliţa, Gheorghe Ştefan, Marius Stoian: "Complex vs. Intensive in Parallel Computation", in *International Multi-Conference on Computing in the Global Information Technology - Challenges for the Next Generation of IT&C - ICCGI*, 2006 Bucharest, Romania, August 1-3, 2006

[Maliţa '07] Mihaela Maliţa, Gheorghe Ştefan, Dominique Thiébaut: "Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation" in *International Workshop on Advanced Low Power Systems held in conjunction with 21st International Conference on Supercomputing* June 17, 2007 Seattle, WA, USA.

[Maliţa '13] Mihaela Maliţa, Gheorghe M. Ştefan: "Control Global Loops in Self-Organizing Systems", *ROMJIST*, Volume 16, Numbers 23, 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, no. 2 , 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`

[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.