Lecture notes
on
COMPUTER ARCHITECTURE

*

(work in progress)

Gheorghe M. Ştefan

– 2024-25 academic year –

This document was prepared with LaTeX 2_ε

Introduction

In this introductory course on Computer Architecture we will travel fast, without required prerequisite, and hopefully efficient, the path from Boolean algebra and digital circuits to Instruction Set Architecture (in short Architecture) and computer organization. From the syllabus of this course we quote:

> *"TCOMP-245 INTRO COMPUTER ARCHITECTURE (3 Credits) This course is an introduction to the building blocks and organization of computers. Topics include: registers, memories and other logic building blocks; central processing unit pipelines; integer and floating point computer arithmetic; memory and cache design; paging and mass-storage systems; interrupt strategies; system bus protocols and shared-memory multiprocessors; contemporary input/output buses and techniques; and the interactions between hardware and the operating system."*

The course is based on the description of the structures and the simulation of their behavior in the System Verilog HDL. For this, the students will install the Vivado Design Suite on their computers (from `https://www.xilinx.com/content/xilinx/en/support/download.html/`).

The activity in this course involves theoretical exposure sessions interspersed with design and simulation sessions made in System Verilog and run in the Vivado environment. Homework is given weekly.

Contents

Contents (detailed)

Section 1

Combinational Circuits: No-Loop, Zero-Order Systems (0-OS)

Contents

In this first lesson, and in the next one, we make a compromise between a brief recapitulation and a quick introduction to the field of digital circuits in such a way as not to bore the already knowledgeable and not to stress the novices. We will introduce strictly those notions that are necessary to understand the principles on which the organization and architecture of computers are based. We will focus on concepts and less on optimizing the way of implementation. We will focus more on the competence of the described structures than on their performance. The functional approach will prevail over the structural one in the description of the components used to design a computer system. We will focus on some possible optimizations, in the last lessons, only at the level of organization and architecture of a computing system.

1.1 Digital Domain

1.1.1 Signals in Digital Domain

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

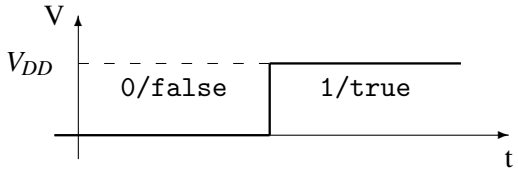


Figure 1.1: The two levels of the signal in the digital domain. Low level (0 Volt) for `0` or `false`, and high level (V_{DD} Volt) for `1` or `true`.

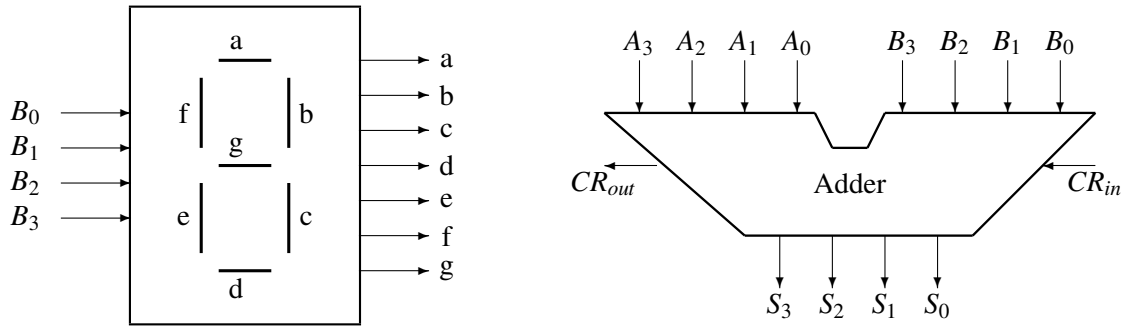
- 0, represented by the electrical value 0 V, having two meanings:
  - the numerical value `0`
  - the logic value `false`
- 1, represented by the electrical value V_{DD} V, having two meanings:
  - the numerical value `1`
  - the logic value `true`

The signals used in a digital system are of two types:

- non-periodic, with a certain structure (”random”) adapted to the process of simulating the operation of a digital system
- periods, usually with a strictly alternating structure, used as clock signals by which the operation of a digital system is synchronized.

In the following System Verilog module, two waveforms are generated, one ”random” and one usable as a signal. (See the simulation result of this module in Section A.1)

```
/*************************************************************************
File name: waveFormGenerator.sv
Circuit name: no circuit, only wave formes
Description: two waveforms are generated, a "random" one and a periodical
             one: a clock signal
*************************************************************************/
module waveFormGenerator();
    logic randomWave;
    logic clock     ;
    initial begin       randomWave = 0  ;
                  #2    randomWave = 1  ;
                  #6    randomWave = 0  ;
                  #4    randomWave = 1  ;
                  #8    randomWave = 0  ;
                  #5    $stop           ;
            end
    initial begin             clock = 0       ;
                  forever #2  clock = ~clock  ;
            end
endmodule
```



a. **b.**

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

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

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

1.1.2 Behavioral vs. Structural Descriptions

The description of a digital circuit can be done in two ways:

- by describing the functional behavior
- by describing the internal structure of the circuit

We exemplify the two methods in the case of the addition function.

Example 1.1 *Let us use the* System Verilog *HDL to describe an adder for 4-bit numbers (see Figure 1.3a). The description which follows is a* **behavioral** *one, because we know* **what** *we intend to design, but we do not know yet* **how** *to design the internal structure of an adder.*

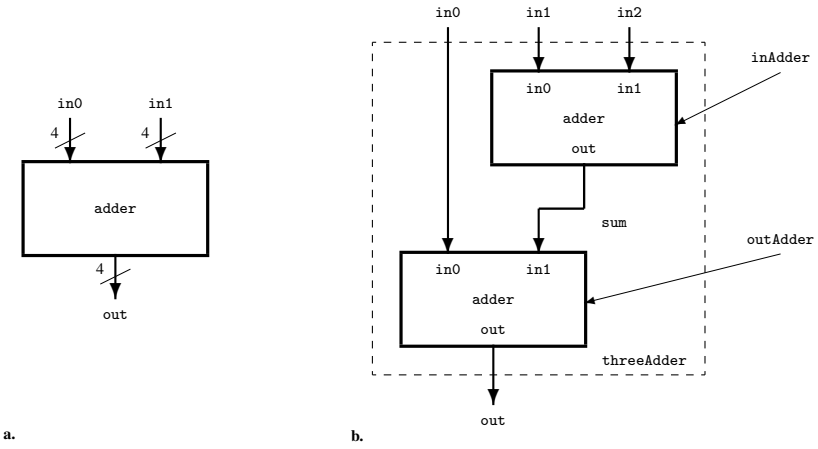


Figure 1.3: **The first examples of digital systems. a.** The two 4-bit numbers adder, called `adder`. **b.** The structure of an adder for 3 4-bit numbers, called `threeAdder`.

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

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

    assign out = in0 + in1;
endmodule
```

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

◇

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

SystemVerilogSummary 1 :

/* : begin comment

***/** : end comment

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

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

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

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

logic : a data type with 4-state bits: `0, 1, x, z` where 0 means low, 1 means high, x means unknown, and z means an undriven net.

initial : a block which starts at the beginning of simulation

begin ... end : block delimiters

: indicate a raw value

forever : indicate an unending operation

$\sim$: negation operator

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

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

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

; : delimiter for end of line

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

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

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

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

Example 1.2 *The previously exemplified module (*`adder`*) will be used to design a modulo 16 3-number adder, called* `threeAdder` *(see Figure 1.3b). It adds 3 4-bit numbers providing a 4-bit result (modulo 16 sum). Follows the* **structural** *description:*

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

    logic [3:0] sum ;

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

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

SystemVerilogSummary 2 :

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

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

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

1.2 One-input combinational circuits

1.2.1 Formal Description

Table 1.1: One-input circuits.

A		NOT	BUFFER	
0	0	1	0	1
1	0	0	1	1

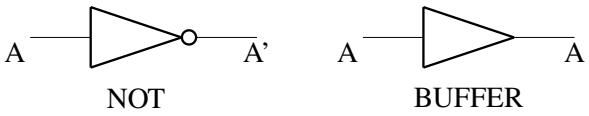


Figure 1.4: Graphic representation of the NOT and BUFFER circuits.

1.2.2 Physical Implementation of NOT Circuit

CMOS switches

A logic gates consists in a network of interconnected switches implemented using the two type of MOS transistors: p-MOS and n-MOS. How behaves the two type of transistors in specific configurations is presented in Figure 1.5.

A switch connected to V_{DD} is implemented using a p-MOS transistor. It is represented in Figure 1.5a *off* (generating *z*, which means *Hi-Z*: no signal, neither 0, nor 1) and in Figure 1.5b it is represented *on* (generating 1 logic, or *truth*).

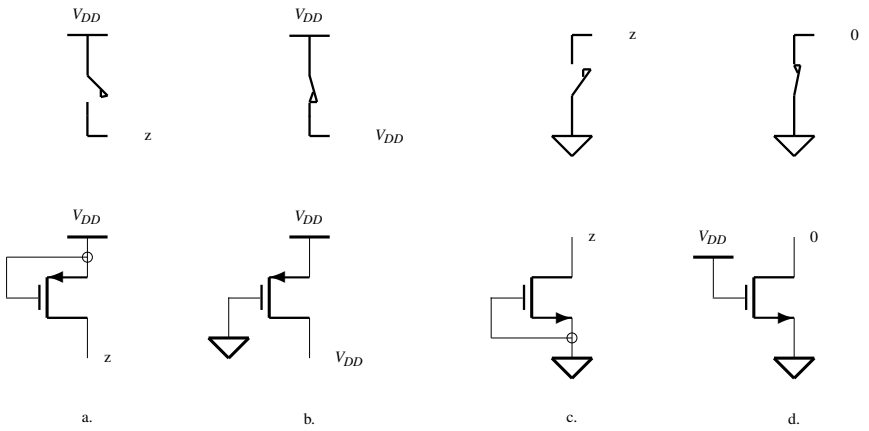


Figure 1.5: **Basic switches.** **a**. Open switch connected to V_{DD}. **b**. Closed switch connected to V_{DD}. **c**. Open switch connected to *ground*. **d**. Closed switch connected to *ground*.

A switch connected to *ground* is implemented using a n-MOS transistor. It is represented in Figure 1.5c *off* (generating *z*, which means *Hi-Z*: no signal, neither 0, nor 1) and in Figure 1.5e it is represented *on* (generating 0 logic, or *false*).

A MOS transistor works very well as an *on-off* switch connecting its drain to a certain potential. A p-MOS transistor can be used to connect its drain to a high potential when its gates is connected to ground, and an n-MOS transistor can connect its drain to ground if its gates is connected to a high potential. This complementary behavior is used to build the elementary logic circuits.

In Figure 1.6 is presented the *switch-resistor-capacitor model* (SRC). If $V_{GS} < V_T$ then the transistor is **off**, if $V_{GS} \geq V_T$ then the transistor is **on**. In both cases the input of the transistor behaves like a capacitor, the gate-source capacitor C_{GS}.

When the transistor is on its drain-source resistance is:

$$R_{ON} = R_n \frac{L}{W}$$

where: L is the channel length, W is the channel width, and R_n is the resistance per square. The length L is a constant characterizing a certain technology. For example, if $L = 0.13\mu m$ this means it is about a $0.13\mu m$ process.

The input capacitor has the value:

$$C_{GS} = \frac{\varepsilon_{OX} LW}{d}.$$

The value:

$$C_{OX} = \frac{\varepsilon_{OX}}{d}$$

where: $\varepsilon_{OX} \approx 3.9\varepsilon_0$ is the permittivity of the silicon dioxide, is the gate-to-channel capacitance per unit area of the MOSFET gate.

In this conditions the gate input current is:

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

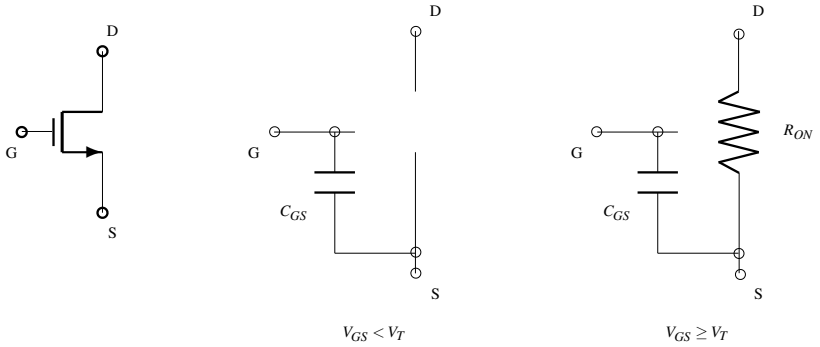


Figure 1.6: **The MOSFET switch.** The *switch-resistor-capacitor* model consists in the two states: OF ($V_{GS} < V_T$), and ON ($V_{GS} \geq V_T$). In both states the input is defined by the capacitor C_{GS}.

Example 1.3 *For an AND gate with low strength, with* $W = 1.8\mu m$*, in* $0.13\mu m$ *technology, supposing* $C_{OX} = 4fF/\mu m^2$*, results the input capacitance:*

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

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

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

◇

The Inverter

The static behavior. The smallest and simplest logic circuit – the invertor – can be built using a pair of complementary transistors, connecting together the two gates as input and the two drains as output, while the n-MOS source is connected to ground (interpreted as logic 0) and the p-MOS source to V_{DD} (interpreted as logic 1). Results the circuit represented in Figure 1.7.

The behavior of the invertor consist in combining the behaviors of the two switches previously defined. For $in = 0$ pMOS is *on* and nMOS is off the output generating V_{DD} which means 1. For $in = 1$ pMOS is off and nMOS is *on* the output generating 0.

The static behavior of the inverter (or NOT) circuit can be easy explained starting from the switches described in Figure 1.5. Connecting together a switch generating z with a switch generating 1 or 0, the connection point will generate 0 or 1.

Dynamic behavior. The propagation time of an inverter can be analyzed using the two serially connected invertors represented in Figure 1.8. The delay of the first invertor is generated by its capacitive load, C_L, composed by:

- its parasitic drain/bulk capacitance, C_{DB}, is the intrinsic output capacitance of the first invertor

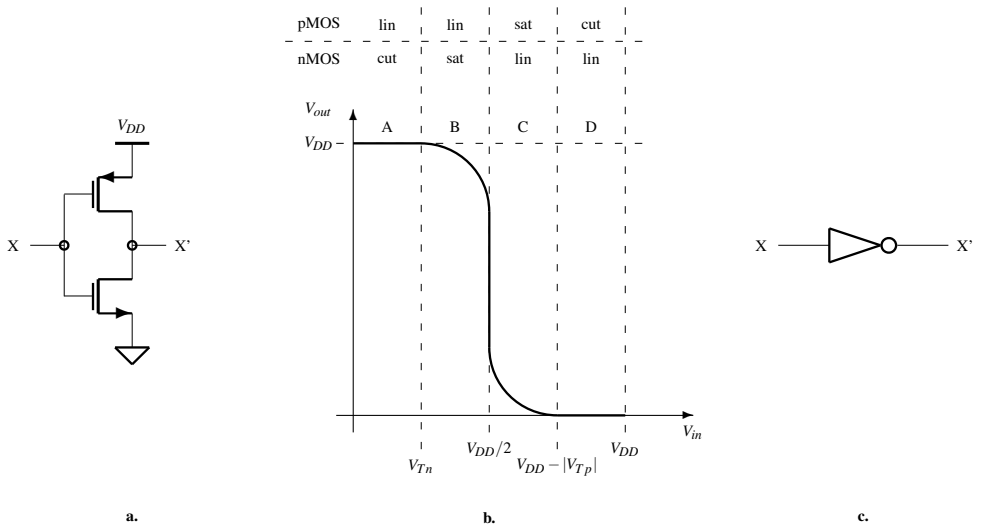


Figure 1.7: **Building an invertor. a**. The invertor circuit. **b**. The logic symbol for the invertor circuit.

- wiring capacitance, C_{wire}, which depends on the length of the wire (of width W_w and of length L_w) connected between the two invertors:

$$C_{wire} = C_{thickox} W_w L_w$$

- next stage input capacitance, C_G, approximated by summing the gate capacitance for pMOC and nMOS transistors:

$$C_G = C_{Gp} + C_{Gn} = C_{ox}(W_p L_p + W_n L_n)$$

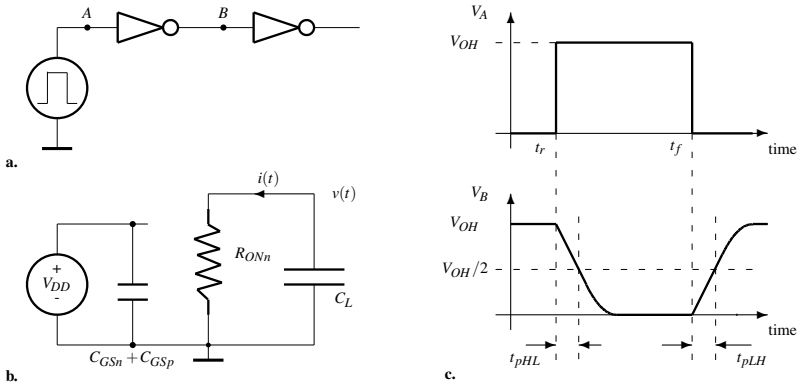


Figure 1.8: **The propagation time.**

The total load capacitance

$$C_L = C_{DB} + C_{wire} + C_G$$

is sometimes dominated by C_{wire}. For short connections C_G dominates, while for big *fan-out* both, C_{wire} and C_G must be considered.

The signal V_A is used to measure the propagation time of the first NOT in Figure 1.8a. It is generated by an ideal pulse generator with output impedance 0. Thus, the rising time and the falling time of this signal are considered 0 (the input capacitance of the NOT circuit is charged or discharged in no time).

The two delay times (see Figure 1.8c) associated to an invertor (to a gate in the general case) are defined as follows:

- t_{pLH}: the time interval between the moment the input switches in 0 and the output reaches $V_{OH}/2$ coming from 0
- t_{pHL}: the time interval between the moment the input switches in 1 and the output reaches $V_{OH}/2$ coming from V_{OH}

Let us consider the transition of V_A from 0 to V_{OH} at t_r (rise edge). Before transition, at t_r^-, C_L is fully charged and $V_B = V_{OH}$. In Figure 1.8b is represented the equivalent circuit at t_r^+, when pMOS is off and nMOS is on. In this moment starts the process of discharging the capacitance C_L at the constant current

$$I_{Dn(sat)} = \frac{1}{2}\mu_n C_{ox}\frac{W_n}{L_n}(V_{OH} - V_{Tn})^2$$

In Figure 1.9, at t_r^- the transistor is cut, $I_{Dn} = 0$. At t_r^+ the nMOS transistor switch in saturation and becomes an ideal *constant current generator* which starts to discharge C_L linearly at the constant current $I_{Dn(sat)}$. The process continue until $V_{OUT} = V_{OH}$, according to the definition of t_{pHL}.

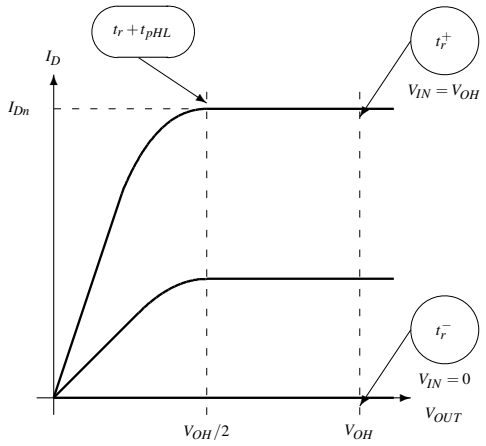


Figure 1.9: **The output characteristic of the nMOS transistor.**

In order to compute t_{pHL} we take into consideration the constant value of the discharging current which provide a linear variation of v_{OUT}.

$$\frac{dv_{out}}{dt} = \frac{d}{dt}(\frac{q_L}{C_L}) = \frac{-I_{Dn(sat)}}{C_L}$$

$$\frac{dv_{out}}{dt} = \frac{\frac{V_{OH}}{2} - V_{OH}}{t_{pHL}}$$

We solve the equations for t_{pHL}:

$$t_{pHL} = C_L \frac{1}{\mu_n C_{ox} \frac{W_n}{L_n}(V_{OH} - V_{Tn})} \frac{V_{OH}}{V_{OH} - V_{Tn}}$$

Because:

$$R_{ONn} = \frac{1}{\mu_n C_{ox} \frac{W_n}{L_n}(V_{OH} - V_{Tn})}$$

results:

$$t_{pHL} = C_L R_{ONn} \frac{1}{1 - \frac{V_{Tn}}{V_{OH}}} = k_n R_{ONn} C_L = k_n \tau_{nL}$$

where:

- τ_{nL} is the *constant time* associated to the H-L transition
- k_n is a constant associated to the technology we use; it goes down when V_{OH} increases or V_T decreases

The speed of a gate depends by its dimension and by the capacitive load it drives. For a big W the value of R_{ON} is small charging or discharging C_L faster.

For t_{pLH} the approach is similar. Results: $t_{pLH} = k_p \tau_{pL}$.

By definition the propagation time associated to a circuit is:

$$t_p = (t_{pLH} + t_{pHL})/2$$

its value being dominated by the value of C_L and the size (width) of the two transistors, W_n and W_p.

1.3 Two-input gates

1.3.1 Formal Description

Table 1.2: Two-input gates.

A	B		NOR					XOR	NAND	AND	NXOR					OR	
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

The currently used two-input gates are (see Tab. 1.2):

AND : A·B = AB = 1 only if both A and B are 1, else AB = 0
Numerical interpretation: arithmetic product
Symbolic interpretation: **gate** because A=1 opens the way for B

NAND : (AB)' = 0 only if both A and B are 1, else AB = 1

OR : A+B = only if at least one of A and B is 1, else A+B = 0

NOR : (A+B) = 0 only if at least one of A and B is 1, else A+B = 1

XOR : exclusive OR, because it is excluded the case A=B=1 in determining 1 on the output
A⊕B = 1 when only one of the input, A or B, is one
Logic interpretation: conditioned inverter, i.e., *if* A = 0 **then** A⊕B = B **else** A⊕B = B' Numerical interpretation: modulo-2 sum
Symbolic interpretation: anticoincidence

NXOR : (A⊕B')
Symbolic interpretation: coincidence

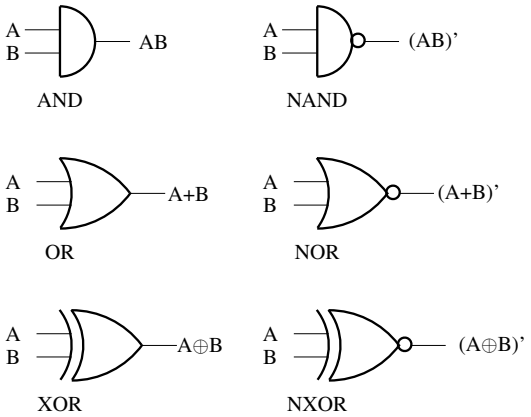


Figure 1.10: Graphical representation of the most used two-input gates.

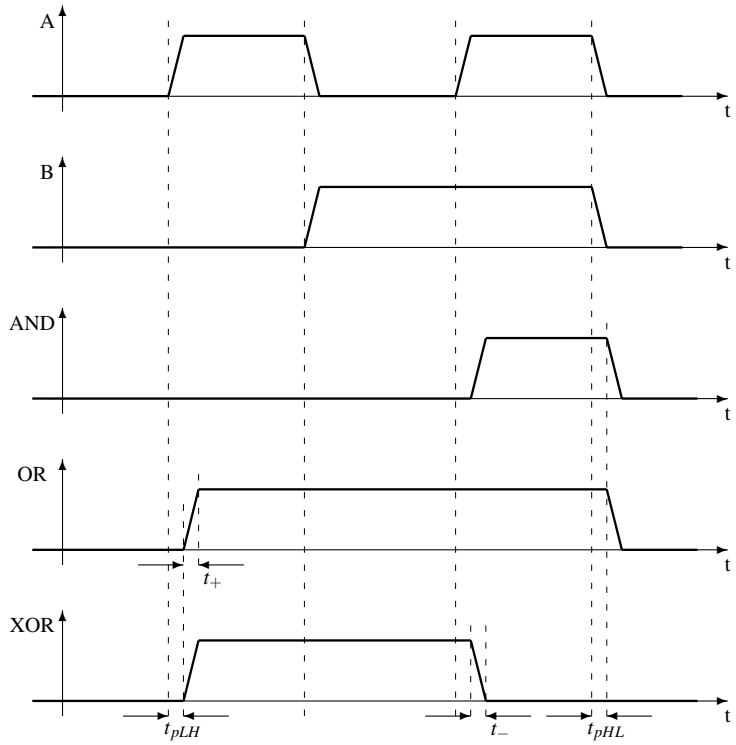


Figure 1.11: Wave forms for the non-inverting gates.

In Figure 1.11 is represented the behavior of the main gates, AND, OR and XOR, driven by the signal A snd B represented in the first two wave forms. Besides the logic response to the inputs A and B there are represented the main time characteristics:

- t_+: the positive transition time (the duration of the positive edge)
- t_-: the negative transition time (the duration of the negative edge)
- t_{pLH}: the propagation time of the signal through the gate for the transition from Low to High
- t_{pHL}: the propagation time of the signal through the gate for the transition from High to Low

All the previous times are different from positive and negative transition and from a type of gate to another. For our approach differences are small and unimportant.

1.3.2 Physical Implementation of NAND & NOR Gates

The 2-input AND circuit, $a \cdot b$, works like a "gate" opened by the signal a for the signal b. Indeed, the gate is "open" for b only if $a = 1$. This is the reason for which the AND circuit was baptised ***gate***. Then, the use imposed this alias as the generic name for any logic circuit. Thus, AND, OR, XOR, NAND, ... are all called *gates*.

The static behavior of gates

For 2-input NAND and 2-input NOR gates the same principle will be applied, interconnecting 2 pairs of complementary transistors to obtain the needed behaviors.

There are two kind of interconnecting rules for the same type of transistors, p-MOS or n-MOS. They can be interconnected serially or parallel.

A serial connection will establish an *on* configuration only if both transistors of the same type are *on*, and the connection is *off* if at least one transistor is *off*.

A parallel connection will establish an *on* configuration if at least one is *on*, and the connection is *off* only if both are *off*.

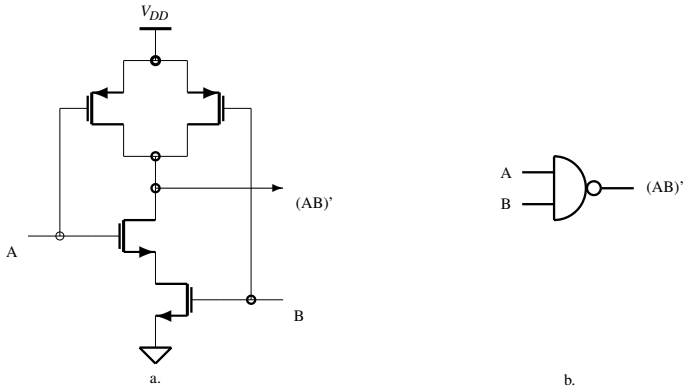


Figure 1.12: **The NAND gate. a**. The internal structure of a NAND gate: the output is 1 when at least one input is 0. **b**. The logic symbol for NAND.

Applying the previous rules result the circuits presented in Figures 1.12 and 1.13.

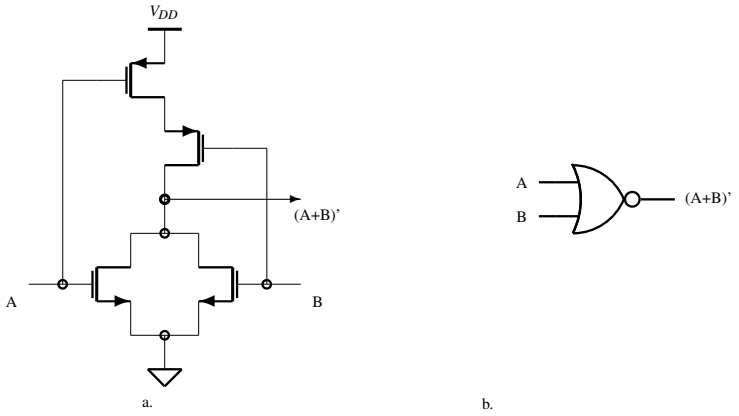


Figure 1.13: **The NOR gate. a**. The internal structure of a NOR gate: the output is 1 only when both inputs are 0. **b**. The logic symbol for NOR.

For the NAND gate the output is 0 if both n-MOS transistors are *on*, and the output is one when at

least on p-MOS transistor is *on*. Indeed, if $A = B = 1$ both n transistors are *on* and both p transistors are *off*. The output corresponds with the definition, it is 0. If $A = 0$ or $B = 0$ the output is 1, because at least one p transistor is *on* and at least one n transistor is *off*.

A similar explanation works for the NOR gate. The main idea is to design a gate so as to avoid the simultaneous connection of V_{DD} and ground potential to the output of the gate.

For designing an AND or an OR gate we will use an additional NOT connected to the output of an AND or an OR gate. The area will be a little bigger (maybe!), but the strength of the circuit will be increased because the NOT circuit works as a buffer improving the time performance of the non-inverting gate.

The propagation time for the 2-input main gates is computed in a similar way as the propagation for NOT circuit is computed. The only differences are due to the fact that sometimes R_{ON} must be substituted with $2 \times R_{ON}$.

Propagation time

Propagation time for NAND gate becomes, in the worst case when only one input switches:

$$t_{HL} = k_n(2R_{ONn})C_L$$

$$t_{LH} = k_p(R_{ONp})C_L$$

because the capacitor C_L is charged through one pMOS transistor and is discharged through two, serially connected, nMOS transistors.

Propagation time for NOR gate becomes, in the worst case when only one input switches:

$$t_{HL} = k_n(R_{ONn})C_L$$

$$t_{LH} = k_p(2R_{ONp})C_L$$

because the capacitor C_L is charged through two, serially connected, pMOS transistors and is discharged through one nMOS transistor.

It is obvious that we must prefer, when is is possible, the use of NAND gates instead of NOR gates, because, for the same area, $R_{ONp} > R_{ONn}$.

1.4 Many-input gates

The number N of n-input gates is: N^{2^n}, because the number of input binary configuration is 2^n. Therefore, they cannot be listed. For $n > 2$ the Boolean functions are managed using Boolean algebra rules.

Let's consider some significant examples:

- (A')' = A
- AB + AB' = A(B + B') = A · 1 = A
  because, when B=1 the expression takes the value A, and when B=0 the expression still takes the value A, we conclude that the value of B does not matter and the expression takes the value A.

- A + A'B = A + B
  to prove it the method of truth table is used, as follow:

Table 1.3: Example of proof using the truth table.

A	B	A'	A'B	A+A'B	A+B
0	0	1	0	0	0
0	1	1	1	1	1
1	0	0	0	1	1
1	1	0	0	1	1

- A $\oplus$ B = (A' $\oplus$ B)' = (A $\oplus$ B')' = A' $\oplus$ B'

- de Morgan rule: A + B = (A'B')'
  A or B is 1 is equivalent to the fact that it is not true that A is 0 and B is 0.

- de Morgan rule: AB = (A' + B')'
  A and B are 1 is equivalent to the fact that it is not true that A is 0 or B is 0.

Fortunately for us, synthesis software tools (Hardware Description Languages (HDLs) such as Verilog or VHDL) "know" how to use Boolean algebra very efficiently, saving us the effort of minimizing combinational logic circuits.

1.4.1 Seven-Segment Display

A seven-segment display is driven by a trans-coder which receives four-bit coded decimal numbers, from 0 to 9, and convert them in 7-bit codes, a to g, indicating the segments to be activated, according to Table 1.4.

The logic expression for the output `a` is:

$$a = B_3' \cdot B_2' \cdot B_1' \cdot B_0' + B_3' \cdot B_2' \cdot B_1 \cdot B_0' + B_3' \cdot B_2' \cdot B_1 \cdot B_0 + B_3' \cdot B_2 \cdot B_1' \cdot B_0 + B_3' \cdot B_2 \cdot B_1 \cdot B_0' + B_3' \cdot B_2 \cdot B_1 \cdot B_0 + B_3 \cdot B_2' \cdot B_1' \cdot B_0' + B_3 \cdot B_2' \cdot B_1' \cdot B_0$$

or a simpler version by negatin the negated function:

$$a = (a')' = (B_3' \cdot B_2' \cdot B_1' \cdot B_0 + B_3' \cdot B_2 \cdot B_1' \cdot B_0')' = (B_3' \cdot B_1' \cdot (B_2 \oplus B_0))' = B_3 + B_1 + (B_2 \oplus B_0)'$$

Table 1.4: Seven-segment display.

B_3	B_2	B_1	B_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1						
0	0	1	1	1						
0	1	0	0	0						
0	1	0	1	1						
0	1	1	0	1						
0	1	1	1	1						
1	0	0	0	1						
1	0	0	1	1						

The System Verilog description of the circuit is sketched in the following:

```
/*************************************************************************
File name:      sevenSegmDys.sv
Circuit name:   Seven-Segment Display
Description:
*************************************************************************/
module sevenSegmDys(output  logic [6:0] seg     ,
                    input   logic [3:0] number  );

    always_comb case(number)
                    4'b0000: seg = 7'b1111110   ;
                    4'b0001: seg = 7'b0110000   ;
                    4'b0010: seg = 7'b  ;
                    4'b0011: seg = 7'b  ;
                    4'b0100: seg = 7'b  ;
                    4'b0101: seg = 7'b  ;
                    4'b0110: seg = 7'b  ;
                    4'b0111: seg = 7'b  ;
                    4'b1000: seg = 7'b  ;
                    4'b1001: seg = 7'b  ;
                    default: seg = 7'b  ;
                endcase
endmodule
```

SystemVerilogSummary 3 :

always_comb : defines a block which describes the behavior of a combinational computed variable (`seg` in the previous module). The blocking assignment, `=`, is used to assure the combinational behavior.

4'b0010 : instantiate a variable as a 4-bit number

case : selects a block of code to be executed based on the value of a given signal in our design. Once a match is found for the input signal value, the action associated with that value will be executed

default : if no condition is fulfilled, the default action is selected for execution.

1.5 Generic combinational circuits

1.5.1 Decoder

Decoding means to identify with a one bit signal each binary code applied to the input of the circuit. Therefor, a n-bit input code applied on the input of a decoder implies a number of 2^n outputs. Only one output is active at a time corresponding to the input code. Thus, for the decoder represented in Figure 1.14, if $\{x_{n-1},\ldots,x_0\} = 00\ldots0101$, then $y_5 = 1$ and all the other outputs are 0.

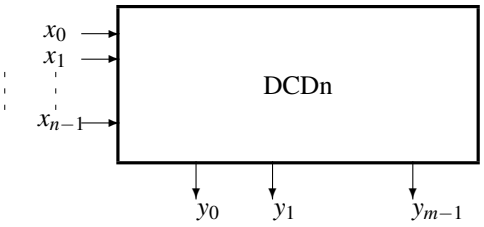


Figure 1.14: Decoder with n-bit input (DCDn).

The System Verilog code for the decoder circuit follows. The description is behavioral.

SystemVerilogSummary 4 :

<< : shift left logical, i.e., multiplication with 2 to the power of the number of binary shifts

---

```
/*****************************************************************
File name:      dcd.sv
Circuit name:   4-input decoder
Description:
*****************************************************************/
module dcd( output  logic [15:0]    dcdOut  ,
            input   logic [3:0]     dcdIn   );

     assign dcdOut = 1 << dcdIn ;
endmodule
```

In Figure 1.15a, is represented the smallest decoder, the elementary DCD, eDCD. Using two eDCD and 4 ANDs, a two-input decoder, DCD2, is designed (see Figure 1.15b). The rule to increase the number of inputs becomes evident with the third example: DCD3 (see Figure 1.15c).

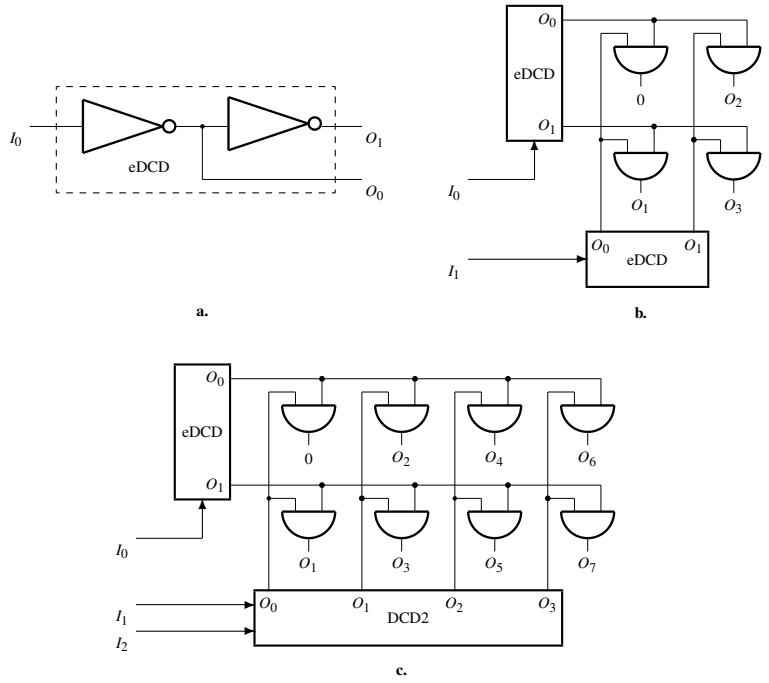


Figure 1.15: Recursively defined 3-input decoder (DCD). **a.** Elementary decoder (eDCD). **b.** Two-input decoder (DCD2). **c.** Three-input DCD (DCD3).

1.5.2 Multiplexor

1.5.3 Elementary multiplexor

Any combinational circuit can be expressed using NAND function. But a more interesting "fundamental brick" could be a circuit corresponding to the ubiquitous ***if-then-ellse*** statement. It is the elementary multiplexor (eMUX) represented in Figure 1.16, where:

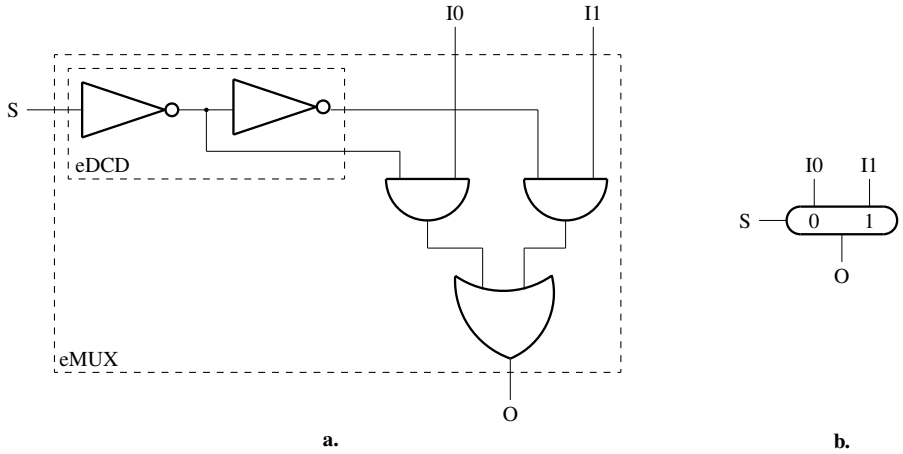


Figure 1.16: Elementary multiplexor (eMUX). **a.** The circuit structure: an eDCD serially connected with an AND-OR circuit. **b.** The logic symbol.

O = **if** (S) **then** I1 **else** I0

1.5.4 Many-input multiplexor

The multiplexor grabs bits from $m = 2^n$ inputs selected by a n-bit code.

SystemVerilogSummary 5 :

---

a[b]: selects the bit indicated by `b` from the binary word `a`

```
/*****************************************************************
File name:      mux.sv
Circuit name:   4-input multiplexor
Description:
*****************************************************************/
module mux( output  logic           muxOut  ,
            input   logic [15:0]    muxIn   ,
            input   logic [3:0]     muxSel  );

    assign muxOut = muxIn[muxSel] ;
endmodule
```

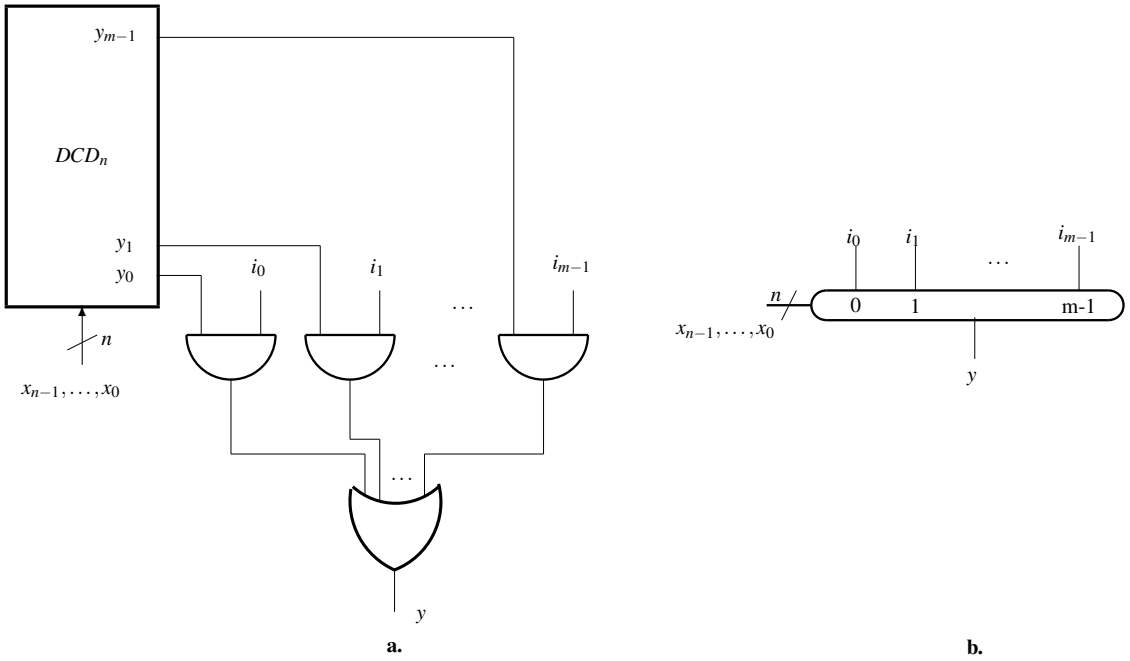


Figure 1.17: The general definition of the multiplexor circuit (MUXn). **a.** The structure of MUXn: a DCDn serially connected with a $m = 2^n$ ANDs on the first level of an AND-OR structure. **b.** The logic symbol for MUXn.

1.5.5 Demultiplexor

The demultiplexer transfers the input signal X to one of the $m = 2^n$ outputs, $y_0, y_1, \ldots, y_{m-1}$, indicated by the n-bit selection input, $x_0, x_1, \ldots, x_{n-1}$. In Figure 1.18 a demultiplexer is built using a decoder which open only one of the m AND gate sending the X input to the selected output.

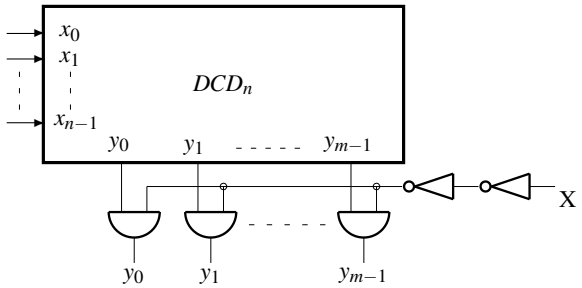


Figure 1.18: Demultiplexor as a DCD whose outputs are ANDed with the one bit signal to be distributed according to the n-bit input code.

For example: if

$$\{x_{n-1}x_{n-2},\ldots,x_1,x_0\} = 00\ldots011$$

then

$$\{y_{m-1}y_{m-2}\dots y_0y_1\} = 00\dots0X000$$

The code describing behaviorally the circuit follows:

```
/*************************************************************************
File name:       dMux.sv
Circuit name:    4-input demultiplexor
Description:     Demultiplexor is an enabled decoder
*************************************************************************/
module dMux(output  logic [15:0]    dMuxOut      ,
            input   logic [3:0]     dMuxIn       ,
            input   logic           dMuxEnable   );

     assign dMuxOut = dMuxEnable << dMuxIn ;
endmodule
```

Now, instead of left-shifting the number 1 as in the decoder, the one-bit value X is left-positioned.

1.6 Function oriented combinational circuits

1.6.1 Half Adder

As we have seen, the XOR circuit calculates the sum modulo 2 but does not differentiate between 0+0 and 1+1. This differentiation involves the calculation of the arithmetical overshoot, which we denote by carry (to the next binary order). Carry signal is activated when both inputs are 1. Therefore, besides the XOR gate we must add an AND gate as in Figure 1.19. We call the resulting circuit half adder because it does not take into account the carry signal provided by the previous binary order.

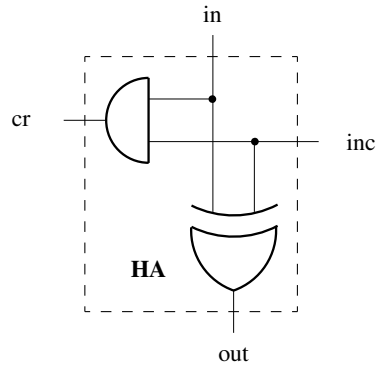


Figure 1.19: Half adder (HA).The XOR circuit compute the *modulo-2* sum while the AND gate compute the carry value.

1.6.2 Increment

Incrementing means adding 1 to a number. The schematic for a 4-bit increment circuit, INC4, is represented in Figure 1.20, where if `inc = 1` the output $out_0 <= in_0'$ and `cr` of HA_0 if it is activated command the increment for HA_1, and so on.

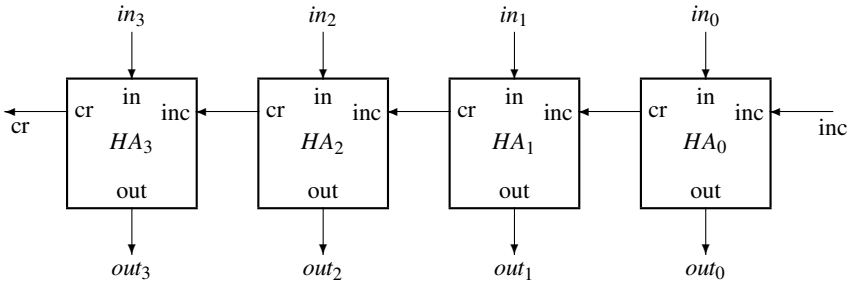


Figure 1.20: Increment circuit for 4-bit numbers (INC4) implemented by serially connected 4 HAs.

1.6.3 Adder/subtractor

Full adder (FA) adds two one-bit numbers with the carry provided from the previous binary position and generate the sum and the carry value for the next binary position. In Figure 1.21a, a FA is composed from two HAs and an OR circuit used to grab the carry generated by the first HA or by the second HA. The first FA adds the two input bits, A and B, while the second FA adds the value of the carry, C, generated by the previous binary order.

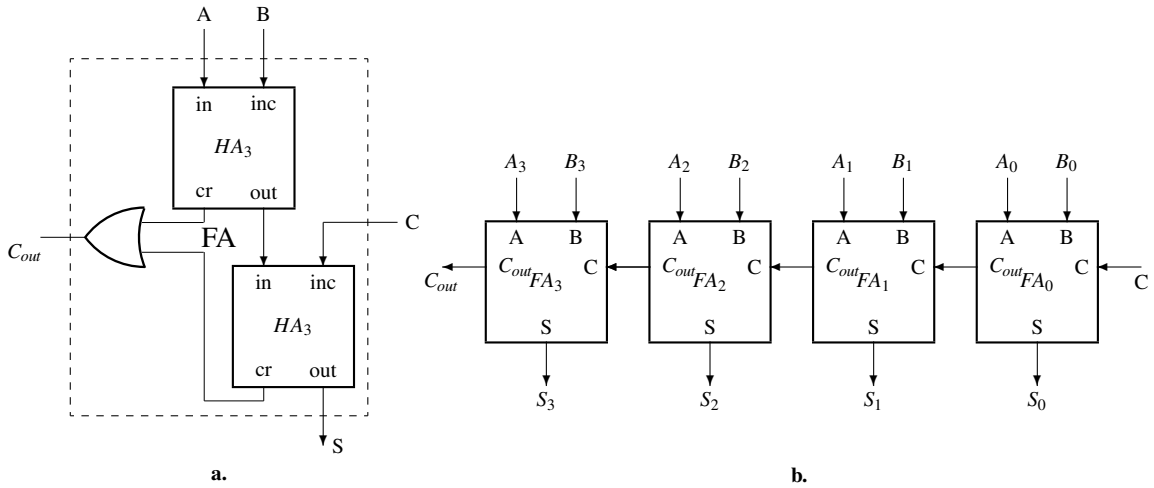


Figure 1.21: Adder. **a.** Full Adder (FA). **b.** 4-bit adder (ADD4).

The main problem to be solved, but not in this lecture, is the execution time, i.e., the maximum time interval from the moment when an input changes until the moment when all the outputs of the circuit reach the correct final value. This time interval is due to propagation through logic gates on the longest

path. In our case, the longest path is the one from input C to output C_{out}. On this path there are 8 logic gates, one AND and one OR for each binary order. In the general case, the time is proportional to n (it is in $O(n)$).

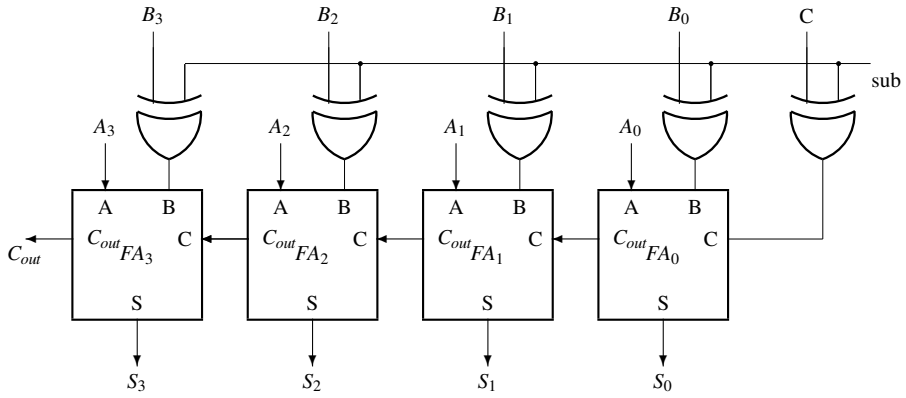


Figure 1.22: Adder/Substractor

Subtract operation is performed by adding the 2s complement:

```
+5 = 0_0101
-5 = 1_1010 + 1 = 1_1011
+5 = 0_0100 + 1 = 0_0101

-5 + 2 = 1_1011 +
         0_0010 =
         1_1101 = -3, because 0_0010 + 1 = 0_0011 = 3
```

Therefore, each B_i is complemented and the increment with 1 is generated by inverting the carry input. All the inversions are performed under de command `sub` applied to 5 XORs.

1.6.4 *Arithmetic & Logic Unit*

An Arithmetic & Logic Unit (ALU) has two kinds of connections: data inputs and outputs, for the n-bit operands, and command inputs for the operations applied to operands (see Fig 1.23). In Figure 1.24, is represented the generic form of an ALU with 15 functions each performed in a distinct module, `func0, ..., func15`, whose outputs are selected by a multiplexor as the result by the `func` code. The two operands, `leftOp[n-1:0], rightOp[n-1:0]`, are applied to all the 15 modules.

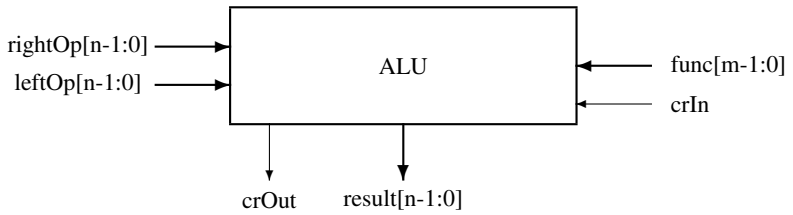


Figure 1.23: ALU.

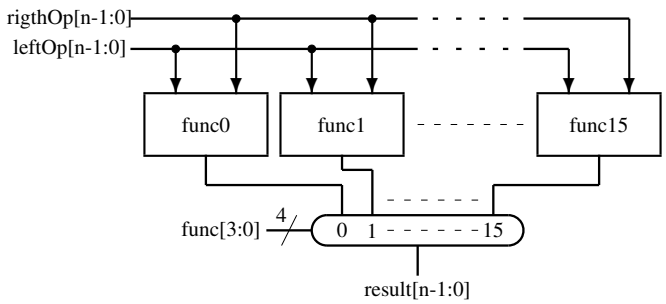


Figure 1.24: The generic form of a 16-function Arithmetic & Logic Unit for *n*-bit words (ALUn).

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

  always_comb case(func)
                3'b000: {crOut, out} = left + right + crIn;     //add
                3'b001: {crOut, out} = left - right - crIn;     //sub
                3'b010: {crOut, out} = {1'b0, left & right};    //and
                3'b011: {crOut, out} = {1'b0, left | right};    //or
                3'b100: {crOut, out} = {1'b0, left ^ right};    //xor
                3'b101: {crOut, out} = {1'b0, ~left};           //not
                3'b110: {crOut, out} = {1'b0, left};            //left
                3'b111: {crOut, out} = {1'b0, left >> 1};       //shr
                default {crOut, out} = 33'b0 - 1'b1;
              endcase
endmodule
```

SystemVerilogSummary 6 :

{a,b} : concatenates the binary word `b` to the binary word `a`

- : is the operator `minus`

& : is the operator `bitwise AND`

| : is the operator `bitwise OR`

^ : is the operator `bitwise XOR`

>> : is the operator `shift right logical`

Actual implementations are sometimes optimized by implementing two ore more modules sharing partially the same circuits. A small and simple example is represented in Figure 1.25, where:

- `func[2:0]` to select the function:

  `func = 000 add:` {`crOut, result`} = `leftOp + rightOp + crIn`

  `func = 100 sub:` {`crOut, result`} = `leftOp - rightOp - crIn`

  `func = 010 and:` {`crOut, result`} = {`1'b0, leftOp & rightOp`} (bitwise AND)

  `func = 001 xor:` {`crOut, result`} = {`1'b0, leftOp` $\oplus$ `rightOp`} (bitwise XOR)

  `func = 011 shr:` {`crOut, result`} = {`leftOp[0], serialIn, leftOp[n-1:1]`}

- `leftOp[n-1:0]`
- `rightOp[n-1:0]`
- `crIn`
- `crOut`
- `serialIn`: the value loaded in the most significant position in the shift right operation, `shr`, allowing the following types of shifts:
  - `shr:` {`crOut, result`} = {`leftOp[0], 1'b0, leftOp[n-1:1]`}
  - `ash:` {`crOut, result`} = {`leftOp[0], leftOp[n-1], leftOp[n-1:1]`}
  - `rot:` {`crOut, result`} = {`leftOp[0], leftOp[0], leftOp[n-1:1]`}
  - `csh:` {`crOut, result`} = {`leftOp[0], crIn, leftOp[n-1:1]`}
- `result[n-1:0]`

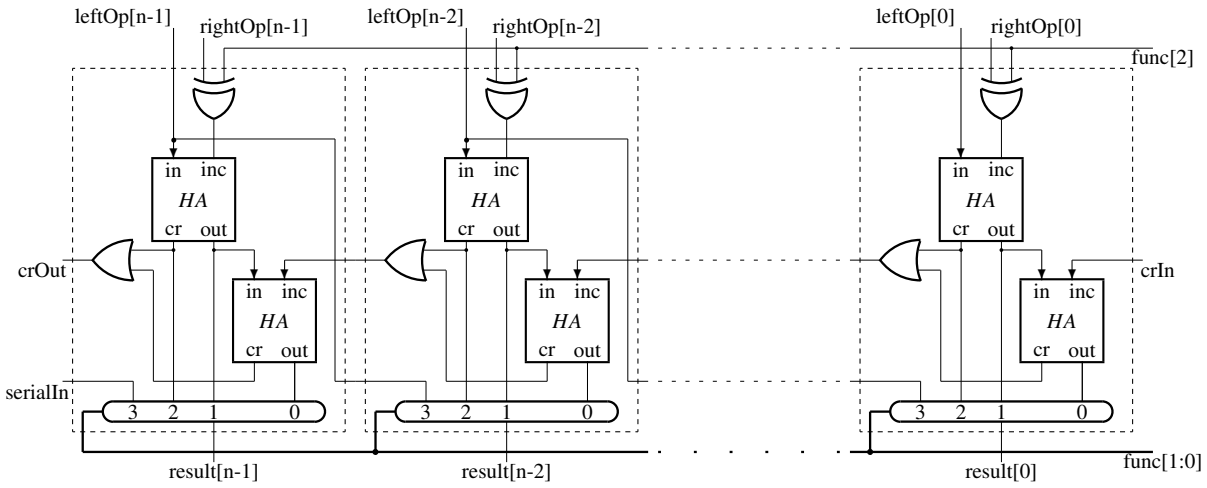


Figure 1.25: A simple Arithmetic & Logic Unit for *n*-bit words (ALUn).

The implementation takes into account the fact that a HA is implemented using an AND gate and a XOR gate (see Figure 1.19). Thus, in each of the n slices of ALU only an adder/subtractor is implemented, a XOR is added for the subtract function and for the two logic functions the outputs of the first HA are directly used.

1.7 Problems

1.7.1 Waveforms

Problem 1.1 *Generate the following waveforms:*

$$W = a^3c^4ba^2$$

where: $a = 2'b01$, $b = 2'b10$, $c = 2'b11$. *and each value is maintained 2 time units (#2).*

◇

Solution

Because each symbol is codded on 2 bits, results 2 wave forms: w_1 and w_0, as follows:

$$\mathrm{W} = \begin{bmatrix} w_1 \\ w_0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}$$

```
/*************************************************************************
File name:        waveform1.sv
Description:      W = aaaccccbaa =
*************************************************************************/
module waveform1;
    logic [1:0] W;

    parameter   a = 2'b01,
                b = 2'b10,
                c = 2'b11;

    initial begin       W = a   ;
                    #6  W = c   ;
                    #8  W = b   ;
                    #2  W = a   ;
                    #4  $stop   ;
            end
endmodule
```

The resulting waveforms are captured in Figure 1.26.

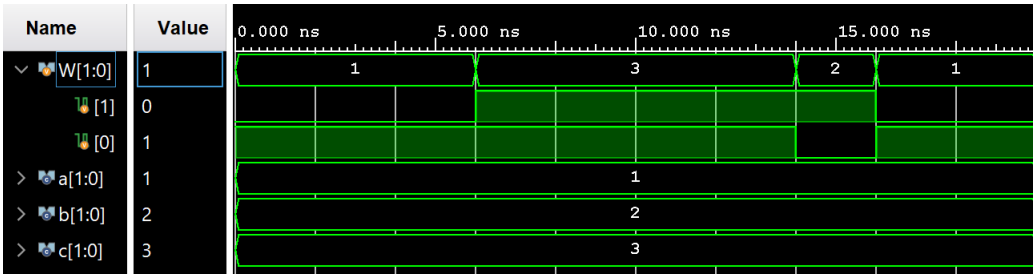


Figure 1.26:

Problem 1.2 *Generate the following waveforms:*

$$(ab^2c)^3$$

where: $W = a = 2'b11$, $b = 2'b10$, $c = 2'b01$, *and each value is maintained 2 time units (#4).*

◇

Problem 1.3 *Generate a clock signal with 2 time units period and the following waveforms:*

$$(a^3b^2c)^2$$

where: $W = a = 2'b11$, $b = 2'b10$, $c = 2'b01$, *and each value is changed synchronously with the negative edge of the clock signal.*

◇

1.7.2 Combinatorial Circuits

Problem 1.4 *Design in System Verilog the structural solution at the gate level for a half adder.*

◇

Solution

A half adder is defined by the following Boolean expressions:

$$sum = a \oplus b$$

$$cr = a \cdot b$$

```
/*************************************************************************
File name:      halfAdder.sv
Description:
*************************************************************************/
module halfAdder(   input   logic a, b,
                    output  logic sum, cr);

    xor myXor(sum, a, b);
    and myAnd(cr, a, b);
endmodule
```

Using Vivado toll the elaborated design represented in Figure 1.27 is provided.

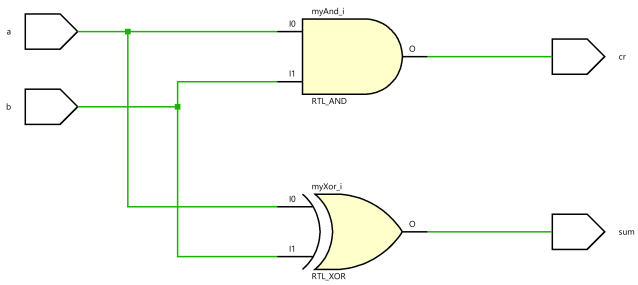


Figure 1.27:

Problem 1.5 *Design in System Verilog the structural solution at the gate level for a full-half adder using two solutions:*

- *describe at the gate level*
- *a hierarchical approach based on the solution of the previous problem using a top module where two half-adders are instantiated.*

Provide the simulation of the circuit to test exhaustively its behavior.

◇

Solution

For the first solution we have the following design:

```
/*************************************************************************
File name:      fullAdder.sv
Description:
*************************************************************************/
module fullAdder(   input   logic a, b, crIn,
                    output  logic sum, crOut);
    logic sum0, cr0, cr1;

    xor myXor(sum0, a, b);
    xor outXor(sum, sum0, crIn);
    and myAnd1(cr0, a, b);
    and myAnd2(cr1, sum0, crIn);
    or myOr(crOut, cr0, cr1);
endmodule
```

Using Vivado toll the elaborated design represented in Figure 1.28 is provided.

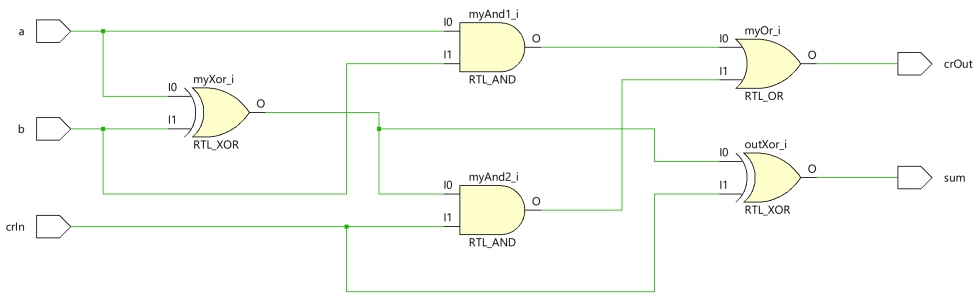


Figure 1.28:

For the second solution we have the following design:

```
/*************************************************************************
File name:      hFullAdder.sv
Description:    Hierarchical Full-Adder
*************************************************************************/
module hFullAdder(  input   logic a, b, crIn,
                    output  logic sum, crOut);
    logic sum0, cr0, cr1;

    halfAdder   ha0(.a  (a      ),
                    .b  (b      ),
                    .sum(sum0   ),
                    .cr (cr0    )),
                ha1(.a  (sum0   ),
                    .b  (crIn   ),
                    .sum(sum    ),
                    .cr (cr1    ));
    or outOr(crOut, cr0, cr1);
endmodule
```

Using Vivado toll the elaborated design represented in Figure 1.29 is provided.

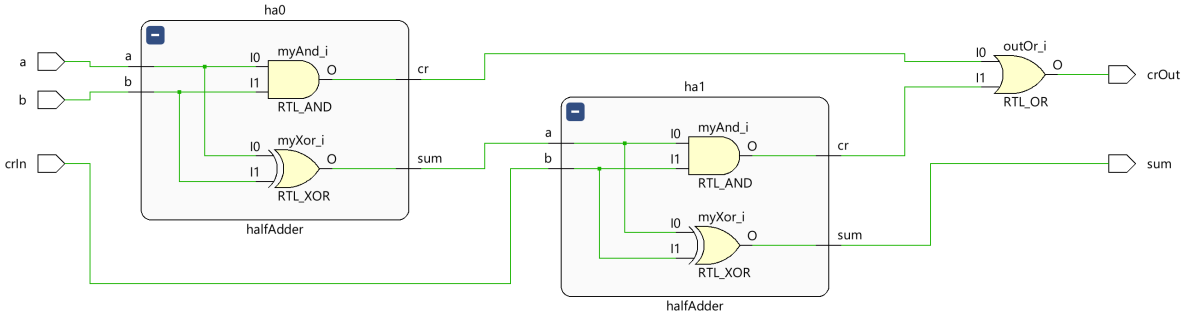


Figure 1.29:

For testing the functionality of the full-adder the following module is designed:

```
/*************************************************************************
File name:      testFullAdder.sv
Description:    Hierarchical Full-Adder
*************************************************************************/
module testFullAdder();
    logic a, b, crIn, sum, crOut;

    hFullAdder dut(a, b, crIn, sum, crOut);

    initial begin       {a,b,crIn} = 3'b000 ;
                  #1    {a,b,crIn} = 3'b001 ;
                  #1    {a,b,crIn} = 3'b010 ;
                  #1    {a,b,crIn} = 3'b011 ;
                  #1    {a,b,crIn} = 3'b100 ;
                  #1    {a,b,crIn} = 3'b101 ;
                  #1    {a,b,crIn} = 3'b110 ;
                  #1    {a,b,crIn} = 3'b111 ;
                  #1    $stop               ;
          end
endmodule
```

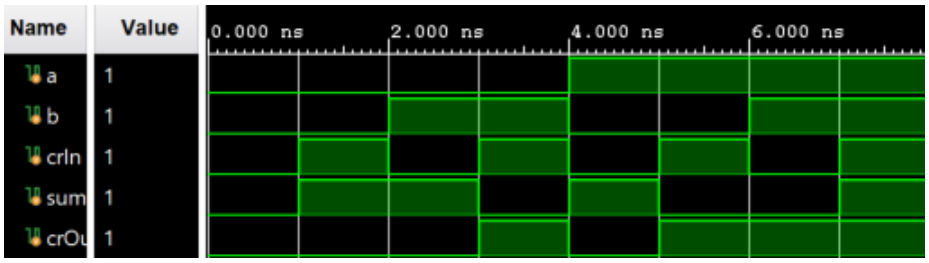


Figure 1.30:

Problem 1.6 *Design a 4-bit number adder with carry input and output.*

◇

Problem 1.7 *Design a 4-bit number subtractor with carry input and output.*

◇

Problem 1.8 *Design a 4-bit number adder/subtractor with carry input and output. The input* `op` *designates addition for* `op = 0`*, and subtract for* `op = 1`.

◇

Problem 1.9 *Fill in the missing lines in Table 1.4 and complete the project partially defined by* `sevenSegDys.sv`. *Test the correct operation.*

◇

Problem 1.10 *Write a full tester for* `dcd.sv` *amd run it.*

◇

Problem 1.11 *Design in System Verilog a 4-input decoder using the recursive definition provided in Figure 1.15. Test its correct functioning.*

◇

Problem 1.12 *Design in System Verilog a 4-input demultiplexor using DCD4 designed as solution for Problem 1.11. Test its correct functioning.*

◇

Problem 1.13 *Design in System Verilog a 4-input multiplexor using DCD4 designed as solution for Problem 1.11. Test its correct functioning*

◇

Problem 1.14 *Consider a decoder with 4 inputs, denoted* `A[3:0]`*, whose outputs are connected to the selected inputs of a multiplexer with output* `F` *and with 4 selection inputs, denoted* `B[3:0]`. *What is the transfer function,* F = f(A,B)*, of the system with two 4-bit inputs,* `A` *and* `B` *and a one-bit output* `F` *?*

◇

Problem 1.15 *Provide an optimal solution for a circuit which receives 2 8-bit words,* `A[7:0]` *and* `B[7:0]`*, and provide one bit output, EQ, telling if the two words are identical or not:*

$$EQ = \begin{cases} 1 & if\ A == B \\ 0 & if\ A\ !== B \end{cases} \tag{1.1}$$

◇

Problem 1.16 *Test the behavior of the ALU defined by* `alu.sv` *in Section 1.6.4.*

◇

Problem 1.17 *Modify the module* `alu.sv` *from Section 1.6.4 to save the bit* `left[0]`*, when the right shift is performed, to the output crOut.*

◇

Problem 1.18 *Add to the module* `alu.sv` *from Section 1.6.4 the following functionality and test it:*

- *right shift with* `crIn` *on the most significant position*
- *arithmetic (right) shift which maintains the sign of the signed integer*
- *rotate left*
- *rotate right*
- *equality comparison:* `left == right`
- *inequality comparison:* `left >= right`
- *select the maximum:* `out = (left =< right) ?  left :  right`
- *select minimum:* `out = (left > right) ?  left :  right`

◇

Problem 1.19 *Design structurally at the gate level in System Verilog the slice of the simple ALU defined in Figure 1.25.*

◇

Problem 1.20 ◇

Section 2

Memory Circuits: One-Loop, First-Order Systems (1-OS)

Contents

In the second lesson we will introduce the memory circuits that are obtained through a first loop that brings to the input of a circuit the effect of applying a signal to another input. This reverse connection allows the circuit to maintain a state according to a command. Thus the memory function is obtained.

We will not go into details because we have a precise target: the internal structure of a computing system that includes a processor, with its internal storage resources, and the memories in which data and software are stored.

2.1 Latch

The simplest circuit will allow the latching of the simplest event: the temporary transition of a digital signal from 1 to 0, or the temporary transition of some circuit from 0 to 1.

2.1.1 Closing the first loop

Closing a feedback loop requires an output of a logic circuit to be connected to one of its inputs. In the Figure 2.1a, the output of an AND gate is connected to one of the inputs, and the (Reset)' pulses are applied to the other input. If the transition to zero lasts long enough so that the signal propagates through the circuit to the second input, then the output is fixed at the zero value and the signal (Reset)' from the input can disappear. The transition event from the input is fixed to the output forever.

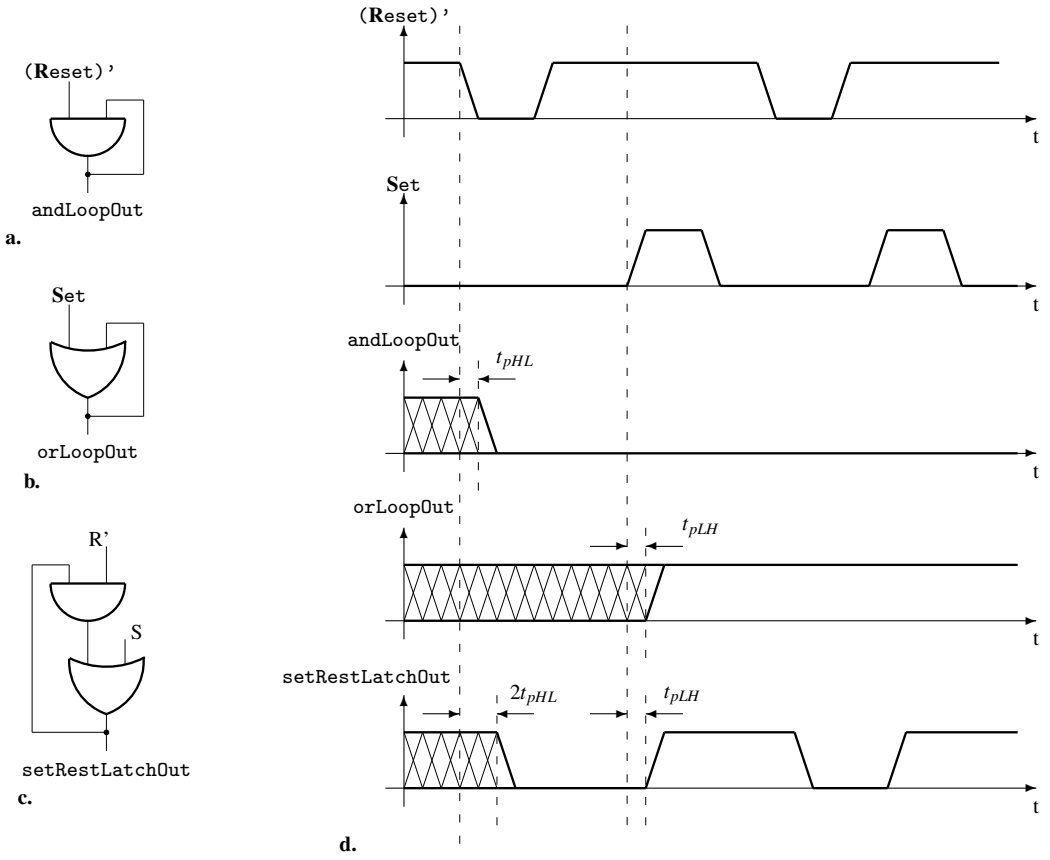


Figure 2.1: **The elementary latches.** Using the loop, closed from the output to one input, elementary storage elements are built. **a**. *AND loop* provides a *reset-only* latch. **b**. *OR loop* provides the *set-only* version of a storage element. **c**. The heterogeneous elementary *set-reset* latch results combining the *reset-only* latch with the *set-only* latch. **d**. The wave forms describing the behavior of the previous three latch circuits.

Now let's take an OR gate and connect its output to one of the inputs, and on the other, apply a transition from 0 to 1. If the duration of the input signal allows the propagation of its effect on the reaction loop, the output of the circuit will be permanently fixed on the value 1.

The good news is that we can capture temporary events indefinitely. We say that the two circuits memorize, one the temporary transition to 0, the other the temporary transition to 1. The bad news

is that we cannot make these circuits ”forget” what they have memorized. But, by combining the two circuits, we manage to obtain a fundamental memory structure that can change its state according to the set command, S, to 1 or according to the reset command, R', to 0. The set command is active on 1, in while the reset command is active on 0.

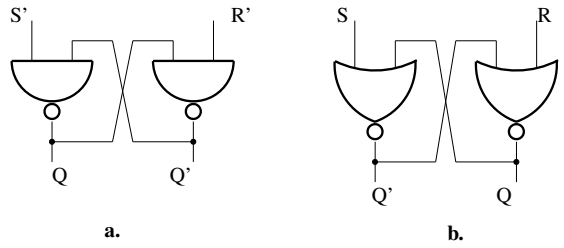


Figure 2.2: **Symmetric elementary latches. a.** Symmetric elementary *NAND latch* with low-active commands S' and R'. **b**. Symmetric elementary *NOR latch* with high-active commands S and R.

We obtained a circuit with two states between which it can switch according to asymmetrical commands, one active on zero and the other active on one. For reasons of use, it is preferable that both commands are of the same type. We will use the two forms of Morgan's law to transform the circuit in Figure 2.1c and we will obtain two symmetrical structures ordered with signals of the same type. Using the de Morgan rule A + B = (A'B')', the latch from Figure 2.1c becomes the latch from Figure 2.2a, and using the de Morgan rule AB = (A'+B')', the latch from Figure 2.1c becomes the latch from Figure 2.2b.

2.1.2 Clocked latch

For the clocked latch two NAND gate are added (see Figure 2.3) to allow the R and R inputs conditioned by clock¿

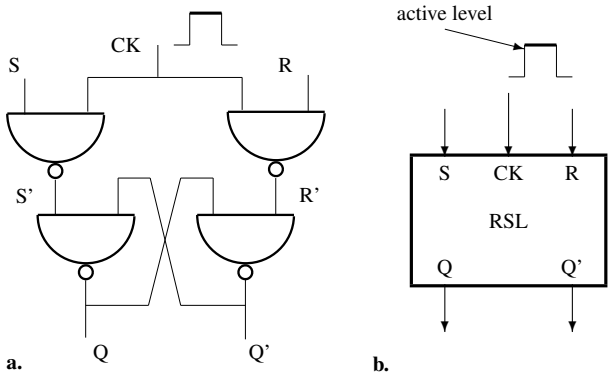


Figure 2.3: **Elementary clocked latch.** The transparent RS clocked latch is sensitive (transparent) to the input signals during the active level of the clock (the high level in this example). **a**. The internal structure. **b**. The logic symbol.

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

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

The first problem is solved in Figure 2.3a by conditioning the application of signals S' and R' to the latch in Figure 2.2a by an additional signal that we will call clock, CK. Thus, we will use the S and R inputs to specify how the circuit switches, and the CK input to specify when the circuit switches. In this way, the decoupling of "*when*" from "*how*" is obtained.

For the second problem, we have a more "brutal" solution achieved through a restriction. We will connect between the two inputs a NOT that will not allow the simultaneous application of setting and resetting the circuit (see Figure 2.4a). The problem is not solved. We avoid doing a wrong use.

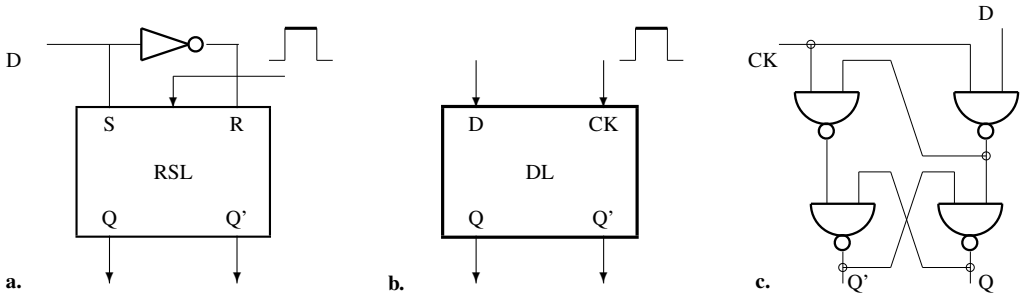


Figure 2.4: **The data latch.** Imposing the restriction $R = S'$ to an RS latch results the **D latch** without non-predictable transitions ($R = S = 1$ is not anymore possible). **a.** The structure. **b.** The logic symbol. **c.** An improved version for the data latch internal structure.

For the time interval in which the clock is active, CK = 1, it says that the latch is transparent to the control signal S and R. The latch becomes non-transparent in the time interval in which CK = 0. Indeed, in the period in which the latch is transparent, the decoupling between "when" and "how" is not achieved, and the circuit switches conditioned by the evolution of the S and R inputs. A correct use of the latch assumes that during the transparency period the S and R inputs are stable.

We have to admit that we did not separate the "when" from the "how" rigorously enough. We will do it in the following.

2.2 Serial extension: Master-Slave Principle

The clocked latch, in the SR version as well as in the D version (data latch), allows switching at any time during the transparency. This fact limits the applications of this circuit. Many applications require switching precisely determined by the active, positive or negative transition of the clock signal. Applying the *master-slave principle* will allow this behavior.

Figure 2.5a shows a structure where transparency is blocked by the fact that the two RS latches have the clock applied in antiphase. On the active level of the clock, the positive one, the first latch, the master,

is transparent, a fact that allows it to be switched according to the S and R inputs without affecting the output of the circuit because the second latch is not transparent. On the inactive level of the clock, the second latch, the slave, copies the state of the master which can no longer be changed. In this way, the entire circuit switches as a consequence of the negative transition of the clock signal. So, the active edge of the clock applied to the master-slave structure is the negative one. In Figure 2.5b, the logical symbol of the structure in Figure 2.5a is represented.

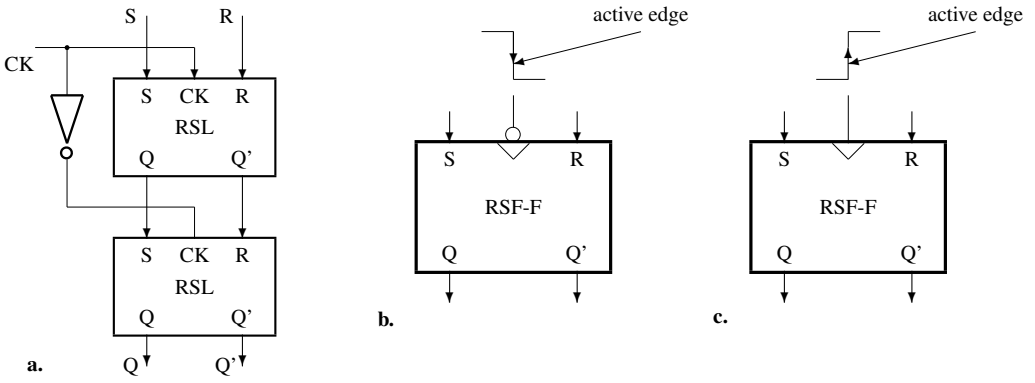


Figure 2.5: **The master-slave principle.** Serially connecting two RS latches, activated with different levels of the clock signal, results a non-transparent storage element. **a.** The structure of a RS master-slave flip-flop, active on the falling edge of the clock signal. **b.** The logic symbol of the RS flip-flop triggered by the *negative edge* of clock. **c.** The logic symbol of the RS flip-flop triggered by the *positive edge* of clock.

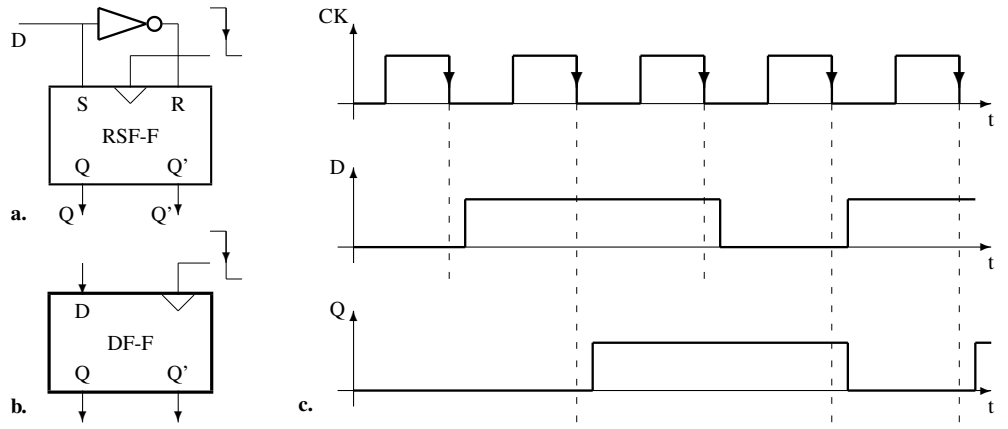


Figure 2.6: **The delay (D) flip-flop.** Restricting the two inputs of an RS flip-flop to $D = S = R'$, results an FF with predictable transitions. **a.** The structure. **b.** The logic symbol. **c.** The wave forms proving the delay effect of the D flip-flop.

The second latch problem also propagates at the level of the master-slave structure. The solution we can come up with at this level is the one that introduces the limitation by connecting the inputs through

an inverter circuit (see Figure 2.6a). The result is the D-FF (delay flip-flop) circuit. The name is justified by the waveforms represented in Figure 2.6c where the output of the circuit follows the evolution of the input with a delay equal to the period of the clock signal.

2.3 Serial-parallel extension: Register

2.3.1 Structure

The serial-parallel extension in the field of first-order systems (1-OS) is represented by the register. By connecting in parallel n serially extended structures (the delay flip-flop type master-slave circuit) is obtained a ***register*** by applying the same clock signal on each D-FF (see Figure 2.7a). A register stores an n-bit configuration synchronously with the active edge of the clock.

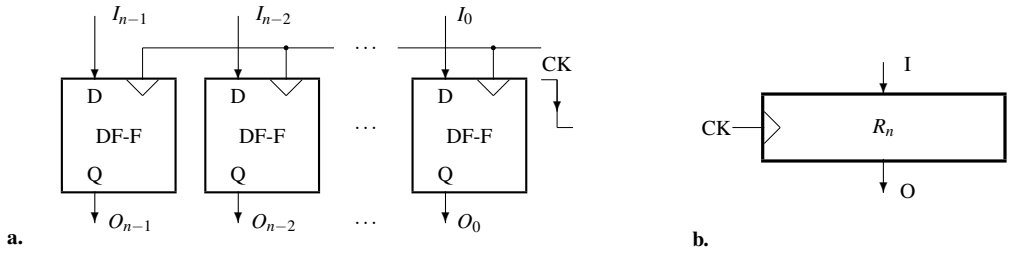
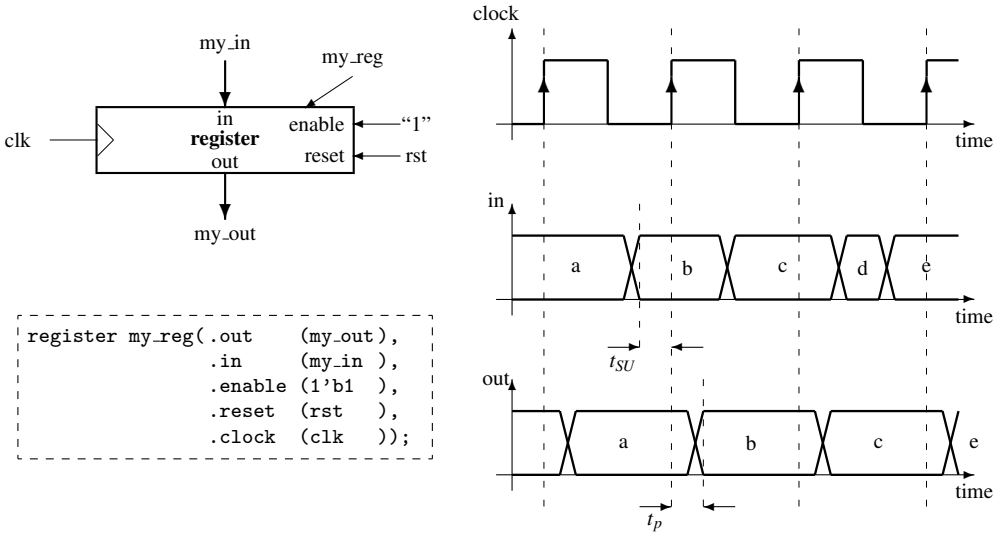


Figure 2.7: **The n-bit register. a.** The structure: a bunch of DF-F connected in parallel. **b.** The logic symbol.

The Figure 2.8 shows the effect of the transfer through a register.




```
register my_reg(.out    (my_out),
                .in     (my_in ),
                .enable (1'b1  ),
                .reset  (rst   ),
                .clock  (clk   ));
```

Figure 2.8: **Register at work.** At each active edge of clock (in this example it is the positive edge) the register's output takes the value applied on its inputs if `reset = 0` and `enable = 1`.

The time behavior is mainly characterized by two times interval:

minimal set-up time : $t_{SU_{min}}$ is the minimum time interval that the input must be kept stable before the active edge of the clock so that the transition of the output corresponds to the input

maximal propagation time : t_p is the propagation time to the output related to the active edge of clock.

2.3.2 Applications

Follows the list of the main applications of the register in digital systems.

Storing

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

Buffering

The registers can be used to *buffer* (to isolate, to separate) two distinct blocks so as some behaviors are not transmitted through the register. For example, in Figure 2.8 the transitions from `c` to `d` and from `d` to `e` at the input of the register are not transmitted to the output.

Synchronizing

For various reasons the digital signals are generated "unaligned in time" to the inputs of a system, but they are needed to be received very well controlled in time. We say usually, the signals are applied *asynchronously* but they must be received *synchronously*. For example, in Figure 2.8 the input of the register changes somehow chaotically related to the active edge of the clock, but the output of the register switches with a constant delay after the positive edge of clock. We say the inputs are synchronized to the output of the register. Their behavior is "time tempered".

Delaying

The input value applied in the clock cycle `n` to the input of a register is generated to the output of the register in the clock cycle `n+1`. In other words, the input of a register is delayed one clock cycle to its output. See in Figure 2.8 how the occurrence of a value in one clock cycle to the register's input is followed in the next clock cycle by the occurrence of the same value to the register's output.

Looping

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

Pipelining

Example 2.1 *The previously exemplified* `threeAdder.sv` *module performs the addition of three numbers in twice the time of the sum of two numbers. We can reduce the execution time through a pipeline solution that uses two registers. The solution is presented in the following module:*

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

    adder   inAdder(    .out(sum    ),
                        .in0(in0    ),
                        .in1(in1    )),
            outAdder(   .out(out    ),
                        .in0(syncReg),
                        .in1(pipeReg));
    always_ff @(posedge clock)  begin   syncReg <= in2  ;
                                        pipeReg <= sum  ;
                                end
endmodule
```

Two modules of `adder` *defined Example 1.1 are instantiated as* `inAdder` *and* `outAdder`*, they are interconnected using the pipeline register* `pipeReg`*. The register* `subcReg` *is used to synchronize the* `in2` *input with the pipelined result provided by* `inAdder`*.*

◇

2.4 Parallel extension: Random-Access Memory

The parallel extension in first-order systems (1-OS) is represented by random access memory (RAM).

2.4.1 Generic structure

The generic structure of a memory with 2^p one-bit locations is represented in the Figure 2.9.

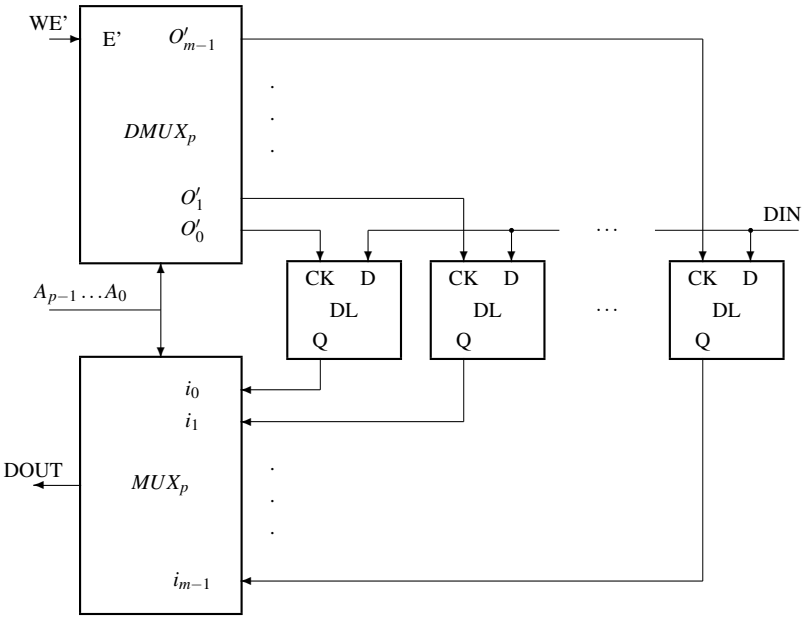


Figure 2.9: **The principle of the random access memory (RAM).** The clock is distributed by a DMUX to one of $m = 2^p$ DLs, and the data is selected by a MUX from one of the m DLs. Both, DMUX and MUX use as selection code a p-bit address. The one-bit data DIN can be stored in the clocked DL.

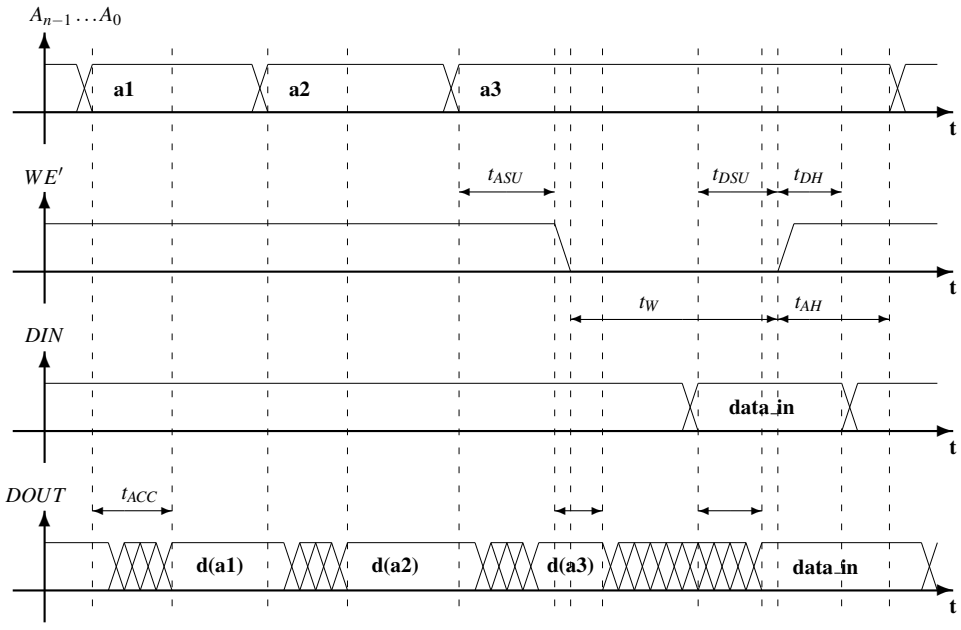


Figure 2.10: **Read and write cycles for an asynchronous RAM.** Reading is a combinational process of selecting. The access time, t_{ACC}, is given by the propagation through a big MUX. The write enable signal must be strictly included in the time interval when the address is stable (see t_{ASU} and t_{AH}). Data must be stable related to the positive transition of WE' (see t_{DSU} and t_{DH}).

The waveforms that define the operation of the structure in Figure 2.9 are represented in Figure A.1 where the main restrictions are represented by:

t_{ASU} : address set-up time represents the time interval in which the address must be stable before the activation of the WE signal to allow the decoder in the DMUX to do its work.

t_{DSU} : data set-up time represents the time interval in which DIN must be stable before de-activating the WE signal to allow the correct closing of the reaction loop in the selected latch.

t_{DH} : data hold represents the time interval in which DIN must be stable after deactivating the WE signal to allow the correct closing of the reaction loop in the selected latch.

t_W : width of the write enable signal which ensures correct writing in the selected latch

t_{AH} : address hold time is the time interval in which the address must be maintained after the WE signal is provided

t_{ACC} : access time is the time interval after which the content of the selected cell is accessible at the output

All the previously listed times are minimum values.

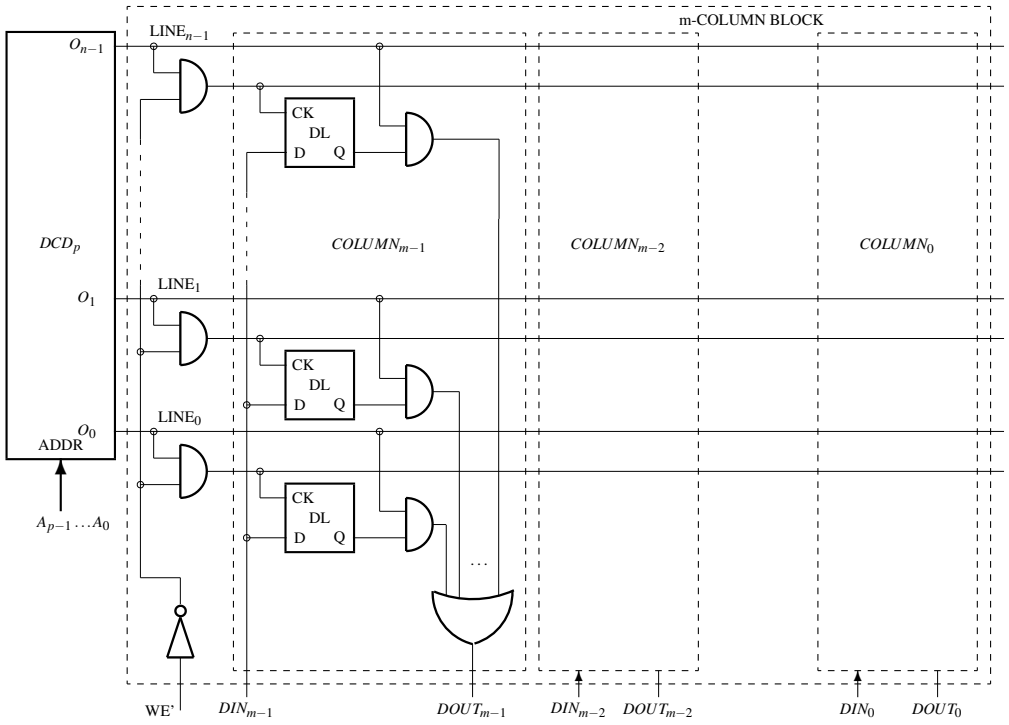


Figure 2.11: **The asynchronous *m*-bit word RAM.** Expanding the number of bits per word means to connect in parallel one-bit word memories which share the same decoder. Each COLUMN contains storing latches and AND-OR circuits for one bit.

How can you build a RAM memory that stores words of m bits? The generic solution is presented in Figure 2.11 where for each bit a column containing 2^p latches is defined.

In Figure 2.11, DCD_p is shared between the DMUX and the MUX in Figure 2.9. The ANDs associated with the demultiplexing function are selected with the WE signal. The AND-OR structure is repeated m times, once in each $COLUMN_i$ column.

2.4.2 Synchronous RAM

At very high speeds, on the order of GHz, the time restrictions illustrated in Figure A.1 are very difficult to fulfill in the already described asynchronous version. To increase the degree of controllability of our projects, synchronous RAM memories are used. In Figure 2.12, the behavior of a synchronous memory (SRAM) is illustrated in the sense that all the time intervals that characterize the behavior of the memory are related to the active edge of a clock signal. In this case, the designer of a system that includes a synchronous memory can more rigorously control the time behavior of the system.

For the memories used by the system designer, *memory libraries* are provided by specialized companies in the form of IPs (intellectual property), guaranteeing the correctness of the code used in the description.

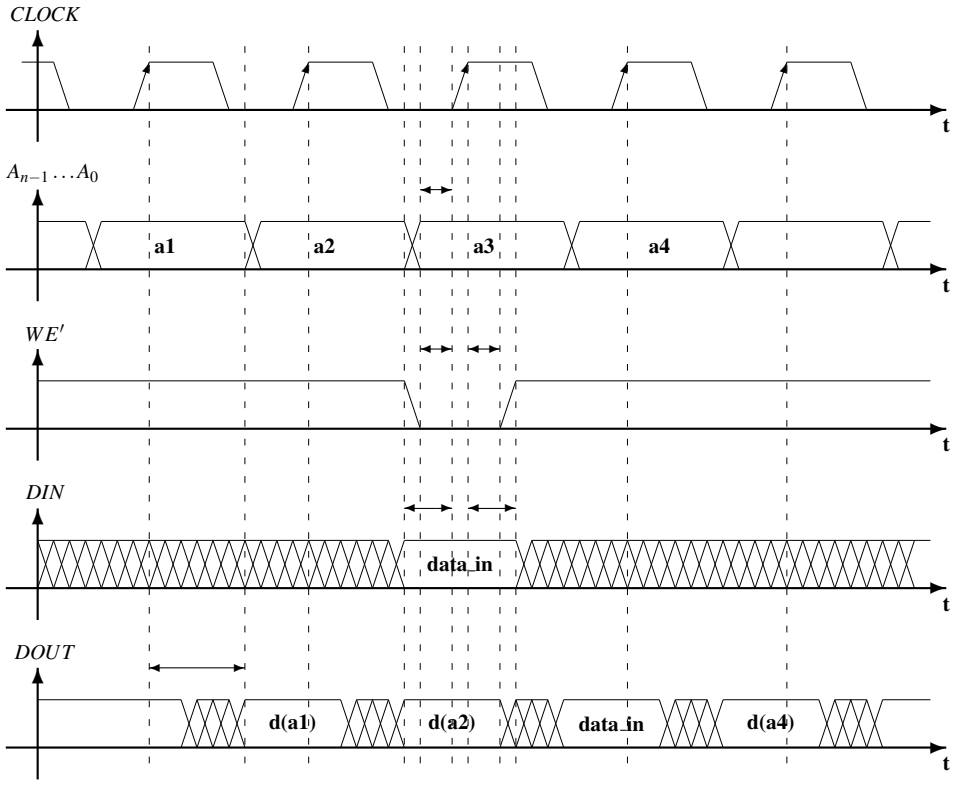


Figure 2.12: **Read and write cycles for Synchronous RAM (SRAM).** For the *flow-through* version of a *SRAM* the time behavior is similar to a register. The set-up and hold time are defined related to the active edge of clock for all the input connections: *data*, *write-enable*, and *address*. The data output is also related to the same edge.

```
/*************************************************************************
File name:      syncRAM.sv
Circuit name:   Synchronous RAM
Description:    4096 32-bit words synchronous RAM; asynchronous read
*************************************************************************/
module syncRAM( output  logic [31:0]    out     ,
                input   logic [31:0]    in      ,
                input   logic [12:0]    addr    ,
                input   logic           we      ,
                input   logic           clock   );
    logic [31:0]    mem[0:4095]   ;

    always_ff @(posedge clock) if (we) mem[addr] <= in  ;
    assign out  = mem[addr] ;
endmodule
```

2.4.3 Synchronous pipelined RAM

The time t_{ACC} offered by a RAM can sometimes be too high for the speed requirements of the system. To increase the clock frequency, a register is used at the memory output. Thus, the data is obtained at the new output very quickly: t_{ACC} is the propagation time through the register. But there is a price: the ***latency*** of one clock cycle (we remember that the register consists of *delay* flip-flops).

A skilled designer will almost always be able to hide the effect of latency and take advantage of the increase in system frequency obtained through the pipeline register from the RAM output.

```
/*************************************************************************
File name:      syncPipeRAM.sv
Circuit name:   Synchronous pipelined RAM
Description:    4096 32-bit words synchronous pipelined RAM
*************************************************************************/
module syncPipeRAM( output  logic [31:0]    out     ,
                    input   logic [31:0]    in      ,
                    input   logic [12:0]    addr    ,
                    input   logic           we      ,
                    input   logic           clock   );
    logic [31:0]    mem[0:4095]   ;

    always_ff @(posedge clock)  begin if (we)   mem[addr] <= in     ;
                                                out  <= mem[addr]   ;
                                end
endmodule
```

2.4.4 *Register file*

A register file is a battery of registers from which, in each clock cycle, any two registers can be accessed for reading and any register for writing. The implementation of such a circuit is done with a synchronous memory with three ports: two for reading and one for writing (see Figure 2.13), where:

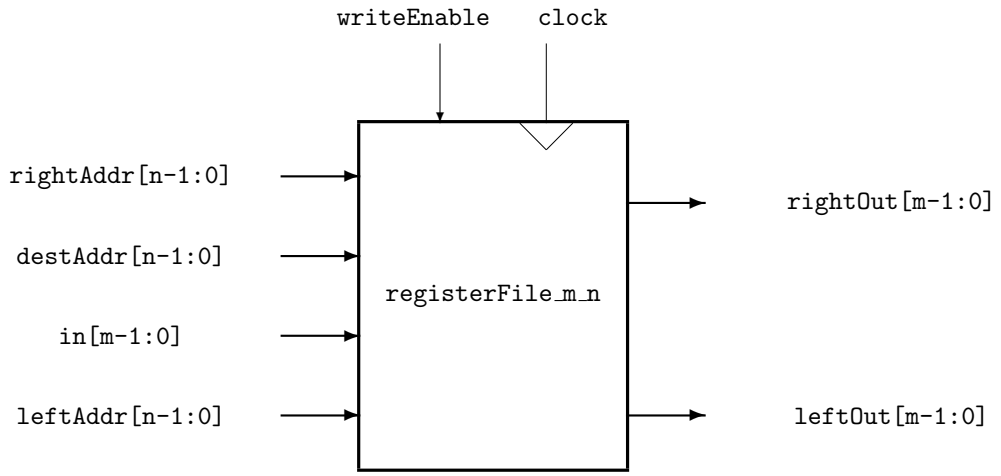


Figure 2.13: **Register file.** In this example it contains 2^n m-bit registers. In each clock cycle any two registers can be read and writing can be performed in anyone.

`leftAddr[n-1:0]`: is the address used to select data to the `leftOut[m-1:0]` output

`rightAddr[n-1:0]`: is the address used to select data to the `rightOut[m-1:0]` output

`destAddr[n-1:0]`: is the address used to select the location where the value applied on input `in[m-1:0]` is stored at the positive edge of the `clock` signal

`writeEnable`: is the write enable command.

The internal structure of a file register is characterized by the two multiplexers that ensure access to any pair of stored values.

```
/*****************************************************************
File name:      regFile.sv
Circuit name:   Register File
Description:    Three-port, 32 32-bit words implemented with a
                synchronous RAM
*****************************************************************/
module regFile( output  logic [31:0]    leftOut     ,
                output  logic [31:0]    rightOut    ,
                input   logic [31:0]    in          ,
```

```
                input   logic [4:0]     leftAddr    ,
                input   logic [4:0]     rightAddr   ,
                input   logic [4:0]     destAddr    ,
                input   logic           we          ,
                input   logic           clock       );
    logic [31:0]    mem[0:31]   ;

    always_ff @(posedge clock) if (we) mem[destAddr] <= in  ;

    assign leftOut  = mem[leftAddr] ;
    assign rightOut = mem[rightAddr];

endmodule
```

SystemVerilogSummary 7 :

logic [m-1:0] mem[0:n-1]: defines a block of memory of n m-bit words

always_ff : defines a block which describes the behavior of register-like sequential circuits; it is followed always at least by @(**posedge** `clock`) or @(**negedge** `clock`) specifying the clock signal and it active edge. The non-blocking assignment, <=, is used to assure the sequential behavior.

2.5 Problems

2.5.1 Registers

Problem 2.1 *Design a system that receives a signed integer in each clock cycle and outputs the sum of the last four received numbers. When the system is reset, it is considered that the last four received numbers had the value 0.*

If $t_{SU_{min}} = \#1$, $t_p = \#5$, $t_{sum} = \#50$, then what is the minimum period of the clock at which the system can present a correct result when entering a register.

◇

Solution

The system consists in 4 registers, `reg0, ..., reg3`, serially connected and a tree of three adders used to sum the content of the registers. The first register, `reg0`, receives in each clock cycle the input value, `in`. Register `reg1` receives the content of `reg1`, `reg2` receives the content of `reg1`, and `reg3` the content of `reg2`. The first two and the last two registers are added using two adders whose outputs are added using the output adder. The division by 4 is simply performed selecting the 32 most significant bits from the output of the last adder.

```
/*************************************************************************
File name:      mean.sv
Circuit name:
Description:
*************************************************************************/
module mean(output  logic [31:0]    out     ,
            input   logic [31:0]    in          ,
            input   logic           reset       ,
            input   logic           clock   );
    logic [31:0]    reg0;
    logic [31:0]    reg1;
    logic [31:0]    reg2;
    logic [31:0]    reg3;
    logic [32:0]    sum0;
    logic [32:0]    sum1;
    logic [33:0]    sum;

    always_ff @(posedge clock)  if( reset)  begin   reg0 <= 0   ;
                                                    reg1 <= 0   ;
                                                    reg2 <= 0   ;
                                                    reg3 <= 0   ;
                                            end
                                else    begin   reg0 <= in  ;
                                                reg1 <= reg0;
                                                reg2 <= reg1;
                                                reg3 <= reg2;
                                            end
    assign sum0 = reg0 + reg1   ;
    assign sum1 = reg2 + reg3   ;
    assign sum  = sum0 + sum1   ;
    assign out  = sum[33:2]     ;
endmodule
```

The minimal period of clock is: $T_{clock} = t_p + 2 \times t_{sum} + t_{SU_{min}} = \#106$

Problem 2.2 *Revisit the Problem 2.1 and insert the first two adders and the output adder pipe registers.*

1. *What is the resulting minimal period of clock*
2. *Design this pipelined version*
3. *Simulate the resulting design.*

◇

Problem 2.3 ◇

Problem 2.4 ◇

2.5.2 Memories

Problem 2.5 *Design a synchronous pipelined RAM for* $2^{addressSize}$ $(wordSize)$*-bit words.*

◇

Solution

```
/*****************************************************************
File name:      defines.vh
*****************************************************************/
`define wordSize     32
`define addressSize  16
```

```
/*****************************************************************
File name:      syncPipeRAM.sv
Circuit name:   Synchronous pipelined RAM
Description:    2^(`addressSize) locations of (`wordSize)-bit words
                synchronous pipelined RAM
*****************************************************************/
`include "DEFINES.vh"
module syncPipeRAM( output  logic [`wordSize-1:0]       out     ,
                    input   logic [`wordSize-1:0]       in      ,
                    input   logic [`addressSize-1:0]    addr    ,
                    input   logic                       we      ,
                    input   logic                       clock   );
    logic [`wordSize-1:0]   mem[0:2**{`addressSize}-1]  ;

    always_ff @(posedge clock)  begin if (we)   mem[addr] <= in     ;
                                                out   <= mem[addr]  ;
                                end
endmodule
```

Problem 2.6 *Consider memory modules of 1KB and design with them a memory of 4K 32-bit word.*

◇

Problem 2.7 ◇

Section 3

Automata: Two-Loop, Second-Order Systems (2-OS)

Contents

In this lesson we will close a second feedback loop in digital systems. This loop will increase the autonomy of the system. If in 0-OS, because we had no loop, the output of the combinational systems is a combination that derives strictly from the current value applied to the input, then in 1-OS a partial autonomy was obtained: the autonomy of the state that could be maintained outside the range of during which the signal that determined it acted. The second loop, which defines 2-OS, will allow the autonomy of the behavior of the system in which it closes, that is, we will obtain the possibility of an evolution of the output even in the absence of a variation of the input signal.

Our final target in this lesson is RALU: ***Registers with Arithmetic & Logic Unit***.

3.1 Definitions

The typical circuit in 2-OS is the automaton. We will highlight two categories of automata: small & complex finite automata and large & simple functional automata.

3.1.1 Generic Definition

We will start with a generic definition because it applies to all kinds of automata introduced in this lecture.

Definition 3.1 *An automaton, A, is defined by the following 5-uple:*

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

where:

X *: the finite set of input variables*

Y *: the finite set of output variables*

Q *: the set of state variables*

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

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

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

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

◇

A strict initial automaton is defined by:

$$A = (X, Y, Q, f, g; q_0)$$

and has a special input, called **reset**, used to led the automaton in the initial state q_0. If the automaton is initial, the input reset switches the automaton in one, specially selected, initial state.

3.1.2 Size vs. complexity

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

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

Definition 3.2 *The* **size** *of a digital circuit, $S_{digital\ circuit}$, is given by the dimension of the physical resources used to implement it.*

◇

In order to provide a numerical expression for size we need a more detailed definition which takes into account technological aspects. In the '40s we counted electronic bulbs, in the '50s we counted transistors, in the '60s we counted SSI[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 3.3 *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 used to implement it*[3].

◇

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

Definition 3.4 *The* **area size** *of a digital circuit, $AS_{digital\ circuit}$, is given by the dimension of the area on silicon used to implement it.*

◇

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

Definition 3.5 *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.*

◇

Definition 3.5 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. Our $C_{digital\ circuit}$ can be associated to the shortest unambiguous circuit description in a certain HDL (in the most of cases it is about a behavioral description).

Definition 3.6 *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: $C_{simple\ circuit} \in O(1)$.

◇

[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.

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

Definition 3.7 *A* **complex circuit** *is a circuit having the complexity in the same magnitude order with its size:*

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

◇

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

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

```
/*************************************************************************
File name:       random_circ.sv
Circuit name:    Example of a complex circuit
Description:     a small complex network of gates
*************************************************************************/
module random_circ(output   logic f, g,
                   input    logic a, b, c, d, e);
   logic      w1, w2;

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

◇

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

```
/*************************************************************************
File name:       or_prefixes.sv
Circuit name:    Example of simple circuit
Description:     a big simple circuit
*************************************************************************/
 module or_prefixes #(parameter n = 256)
        (output logic [0:n-1] out,
         input  logic [0:n-1] in);

    integer k;
    always_comb  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]

◇

[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.

SystemVerilogSummary 8 :

and, or, xor, not : are reserved names for predefined circuits which can be instantiated under specific names

parameter : used to define a local parameter

integer k: defined a positive integer

for : defined the well known loop running under the index `k`

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

Definition 3.8 *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.*
◇

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

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

3.1.3 Taxonomy

Starting from the generic definition, in Figure 3.1 are represented the generic structures of the automata used in digital design.

The structures found in current practice are:

half automaton : is an automaton whose output is identical to the internal state (the combinational circuit that transforms the state into the output is missing); the latency of the Y output compared to the X input is one clock cycle

Mealy immediate automaton : is an automaton whose output is calculated by a combinational circuit depending on state and input; the latency of the Y output compared to the X input is zero

Moore immediate automaton : is an automaton whose output is calculated by a combinational circuit depending only by the state; the latency of the Y output compared to the X input is one clock cycle

Mealy delayed automaton : is a Mealy immediate automaton with the output delayed through a pipeline register; the latency of the Y output compared to the X input is one clock cycle

Moore delayed automaton : is a Moore immediate automaton with the output delayed through a pipeline register; the latency of the Y output compared to the X input is two clock cycles.

Depending on the way the machine is integrated into the system we are designing, we will choose the most suitable version. Latency can sometimes be advantageous.

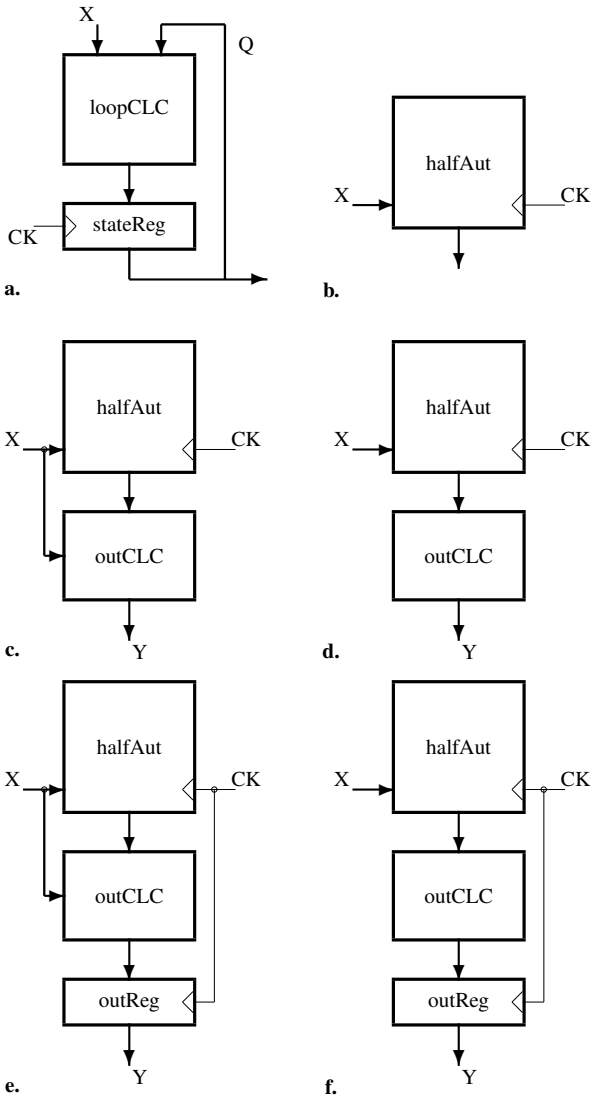


Figure 3.1: **Automata types. a.** The structure of the half-automaton ($A_{1/2}$), the no-output function automaton: the state is generated by the previous state and the previous input. **b.** The logic symbol of half-automaton. **c.** Immediate Mealy automaton: the output is generated by the current state and the current input. **d.** Immediate Moore automaton: the output is generated by the current state. **e.** Delayed Mealy automaton: the output is generated by the previous state and the previous input. **f.** Delayed Moore automaton: the output is generated by the previous state.

We will make another important distinction: that between complex automata and simple ones.

Definition 3.9 *A complex automaton has a description proportional with the number of state it has.*

◇

Definition 3.10 *A simple automaton has a description of constant size, independent of the number of its states.*

◇

3.2 Finite (complex) Automata

Definition 3.11 *A finite automaton is characterized by $|Q| \in O(1)$, i.e., the size of the set Q is constant and independent on the length of the string of symbols it receives.*

◇

We will exemplify with two typical cases: a recognizing automaton and a control automaton.

3.2.1 Recognizing automata

Example 3.3 *The binary strings 1^n0^m, for $n \geq 1$ and $m \geq 1$, are recognized by a finite automaton. Let's define and design it. The transition diagram defining the behavior of the automaton is presented in Figure 3.2, where the state set Q contains:*

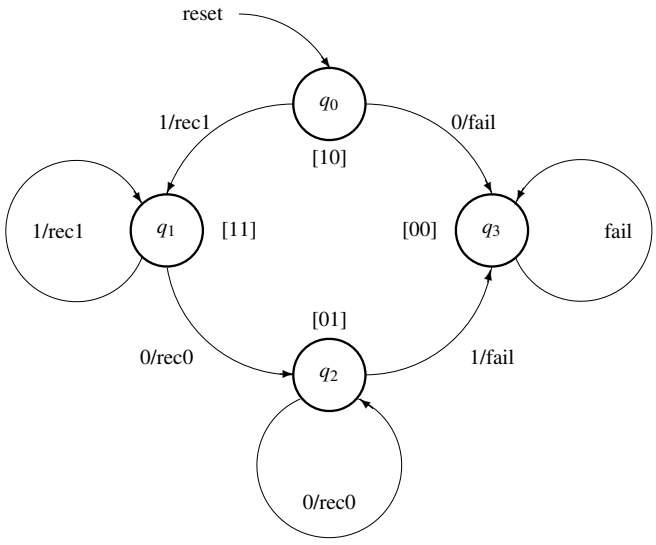


Figure 3.2: **Transition diagram.** The transition diagram for the half-automaton which recognizes strings of form 1^n0^m, for $n \geq 1$ and $m \geq 1$. Each circle represent a state, each (marked) arrow represent a (conditioned) transition.

- q_0 - *is the* initial *state in which:*
  - *if 1 is received, the output is* `rec1` *and the automaton switches in the state* q_1

  - *if 0 is received the automaton generate on output* `fail` *and switches in* q_3

- q_1 *- in this state at least one 1 was received and*
  - *if 1 is received, the output is* `rec1` *and the state remains the same*
  - *if 0 is received, the output is* `rec0` *the the state becomes* q_2
- q_2 *- this state acknowledges a well formed string:*
  - *if 1 is received, the output is* `fail` *and the state switches in* q_3
  - *if 0 is received, the output is* `rec0` *and the state remains the same*
- q_3 *- is the error state because an incorrect string was; the state remains unconditionally the same until a new* `reset` *is aplied. received.*

The first step in implementing the structure of the just defined automaton is to **assign binary codes** *to each state. In this stage we have the absolute freedom. Any assignment can be used. The only difference will be in the resulting structure but not in the resulting behavior.*

Table 3.1: Transition state table.

Q_1	Q_0	X	Q_1^+	Q_0^+	Y_1	Y_0
0	0	0	0	0	1	1
0	0	1	0	0	1	1
0	1	0	0	1	0	1
0	1	1	0	0	1	1
1	0	0	0	0	1	1
1	0	1	1	1	1	0
1	1	0	0	1	0	1
1	1	1	1	1	1	0

Let be the codes, symbolized by Q_1 and Q_2, assigned in square brackets in Figure 3.2. For the outputs, the symbols Y_1 and Y_2 are used with the following meanings:

```
rec0: 01
rec1: 10
fail: 11
```

The input signal is symbolized by X.

Results the transition table, for the functions f and g, presented in Table 3.1. The resulting transition functions the resulting immediate Mealy automaton are:

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

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

$$Y_1 = (Q_0 \cdot X)'$$

$$Y_0 = Q_1 \cdot X' = (Q_1' + X)'$$

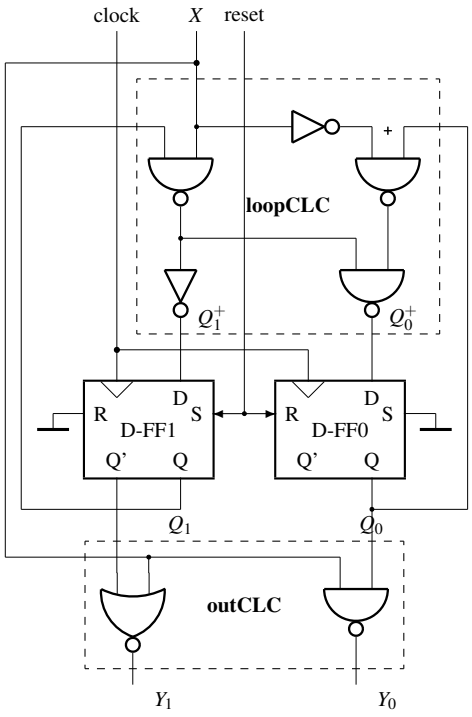


Figure 3.3: **A 4-state finite automaton.** The structure of the finite automaton used to recognize binary string belonging to the 1^n0^m set of strings.

The circuit is represented in Figure 3.3 in a version using inverted gated only. The 2-bit state register is designed by 2 D flip-flops. The `reset` *input is applied on the* set *input of D-FF1 and on the* reset *input of D-FF0.*

The recognition process begins with the application of the `reset` *signal through which the automaton goes to the initial state, q_0, coded by $Q_1, Q_0 = 10$. A correct string must start with 1, otherwise the automaton goes to the final state, q_3, which signals* `fail`*. In state q_1, the automaton receives 1s, until the first 0 is applied to the input and the automaton passes to state q_2 in which it signals, through* `rec0`*, that the string received is correct. If in state q_2 it is received in 1, then the string is cataloged as not belonging to those recognized by blocking the automaton in state q_3.*

◇

Designing a finite automaton using a description in System Verilog HDL involves writing the following files:

- `defines.vh` in which the binary values that the state, input and output variables take are defined
- `topAutomaton.sv` in which the module that describes the automaton defining the state register, instantiating the combinational calculation module of the loop, `loopCLC`, and the module `outCLC` that describes the output circuit is defined
- `loopCLC.sv` which describes the state transition function

- `outCLC.sv` which describes the output transition function.

Example 3.4 *Let's revisit the previous example, recognizing the language* $L = (a^n b^m | n,m > 0)$*, and solve the design using a System Verilog description.*

```
/*************************************************************************
File name: defines.vh
Description: binary codes for input, output and state variables
*************************************************************************/
// input codes
    `define a 1'b1
    `define b 1'b0
// output codes
    `define recA    2'b10 // automaton receivs a
    `define recB    2'b01 // automaton receivs b
    `define fail    2'b00 // automaton faild to receive correct string
// internal states
    `define init    2'b10 // initial state
    `define runA    2'b11 // automaton received at least one b
    `define runB    2'b01 // automaton received at least one a
    `define eror    2'b00 // fail state
```

```
/*************************************************************************
File name: regLang.sv
Circuit name: Automaton that tecognize the regular language
                  L = (a^nb^m | n,m >0)
Description: structural description of the immediate Mealy automaton
             designed to recognize the language L
*************************************************************************/
`include "defines.vh"
module regLang( input   logic       in      ,
                output  logic [1:0] out     ,
                input   logic       reset   ,
                                    clock   );
    logic [1:0]   state, nextState  ;

    always_ff @(posedge clock) begin: state_register
        if (reset)      state   <= `init    ;
         else       state   <= nextState;
    end: state_register
    loopCLC lC( in          ,
                state       ,
                nextState   );
    outCLC oC(  state   ,
                in      ,
                out     );
endmodule
```

```
/*************************************************************************
File name:      loopCLC.sv
Circuit name:   loopCLCimmMealy
Description:    combinational circuit used to compute the loop for
                immediate Mealy language L detector automaton
*************************************************************************/
`include "defines.vh"
module loopCLC( input   logic          in        ,
                input   logic [1:0]    state     ,
                output  logic [1:0]    nextState );

    always_comb begin: loop_clc
        case(state)
            `init   : nextState = (in == `a) ? `runA   : `error ;
            `runA   : nextState = (in == `a) ? `runA   : `runB  ;
            `runB   : nextState = (in == `b) ? `runB   : `error ;
            `error  : nextState = `error                        ;
        endcase
    end: loop_clc
endmodule
```

```
/*************************************************************************
File name: outCLC.sv
Circuit name: outputCLCimmMmealy
Description: combinational circuit used to compute the output for
             immediate Mealy bcb detector automaton
*************************************************************************/
`include "defines.vh"
module outCLC(  input       [1:0]   state   ,
                input               in      ,
                output  bit [1:0]   out     );

    always_comb begin: out_clc
        case(state)
            `init   : out = (in == `a) ? `recA : `fail   ;
            `runA   : out = (in == `a) ? `recA : `recB   ;
            `runB   : out = (in == `a) ? `fail : `recB   ;
            `error  : out = `fail                        ;
        endcase
    end: out_clc
endmodule
```

◇

SystemVerilogSummary 9 :

`define : used to define global variable

‘include ”fileName.vh” : used to include code already defined

‘name : used to assert a name already defined using `‘define`.

The main point: for any values for m and n, the number of states of the finite automaton is constant, equal with 4. But the description of the of the automaton's behavior is proportional with the number of states (see the number of lines of the truth table defining the state transition) and independent by values m and n.

3.2.2 Control automata

Another use of a finite automaton is to control. Controlling means to send a sequence of commands to a digital systems and to determine the evolution of this sequence according to signals receiving back from the controlled system. Thus, the evolution in the state space depends: (1) on the internal loop of the automat and (2) on the flags received form the controlled system.

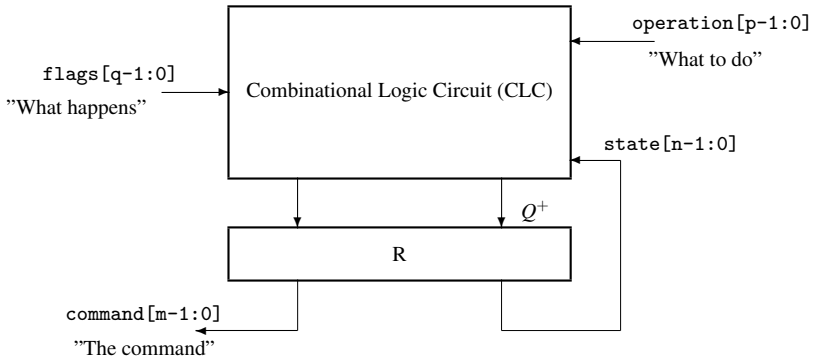


Figure 3.4: **Control Automaton.** The functional definition of control automaton. Control means to issue commands and to receive back signals (flags) characterizing the effect of the command.

In Figure 3.4, a Mealy delayed version is presented. It works as follows:

- the input `operation[p-1:0]` is a binary code used to initialize the automaton in 2^p different initial states, each associated to a control sequence of commands to be issued to the controlled system
- the inputs `flags[q-1:0]` represent independent bits used to characterise the behavior of the controlled system (for example if an arithmetic operation provided carry)
- the output `command[m-1:0]` represent the command issued in each clock cycle toward the control system; it can be organized in one ore few fields to control different part of the system
- `state[n-1:0]` represent the inner loop of the control automaton

The size of the CLC of this generic version is, in most of cases, to big to be optimally implementable. Indeed, because in the general case a CLC with N inputs is proportional with 2^N, our CLC could have a size proportional with 2^{p+q+n}.

But, to our luck, real applications have certain characteristics that allow a drastic simplification of the generic solution. We present it in the Figure 3.5.

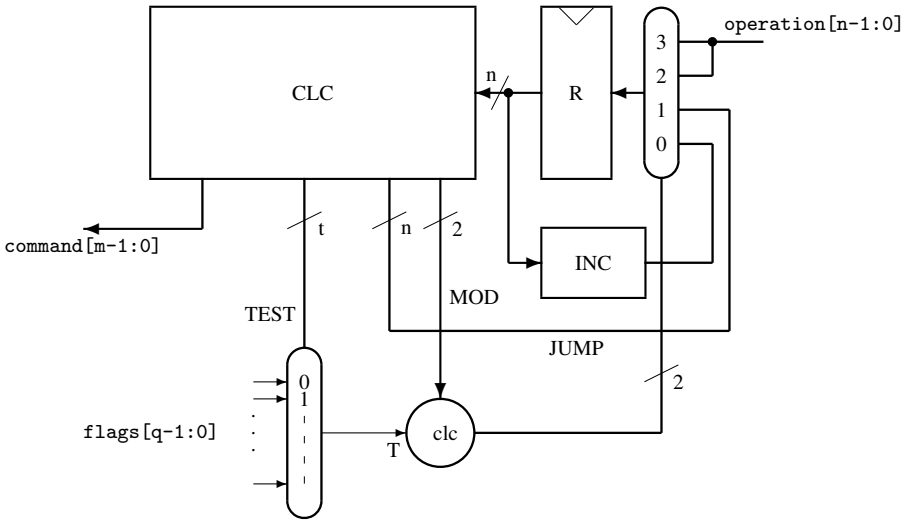


Figure 3.5: **The simplest Controller.** The Moore version of a control automaton is optimized by: (1) using a multiplexor to select, using MOD and T combined by `clc`, the next state from different sources, (2) using an increment circuit (INC) to compute the most frequent next address for CLC, and (3) using a multiplexor to select, using TEST, for each cycle the appropriate flag.

We arrived at the structure in Figure 3.5 starting from the following observations:

1. the entry `operation` is taken into account in the calculation of the transition only in certain states, those of initializing a new command sequence

2. the `flags` entries are not all significant in every state

3. the command sequences contain subsequences that are chained unconditionally, a fact that allows their encoding by incrementing

3.3 Simple Automata

A simple automaton has the loop closed through a simple combinatorial circuit. We will provide two examples on our way to build a computing engine.

3.3.1 Counters

T Flip-Flop

The smallest and the simplest automaton is a half-automaton with 2 states and one input (see Figure 3.6a). What could be the behavior of an automaton with only two internal states (Q and Q') and a single input bit T? The only solution:

$$Q \Leftarrow T\ ?\ Q' : Q$$

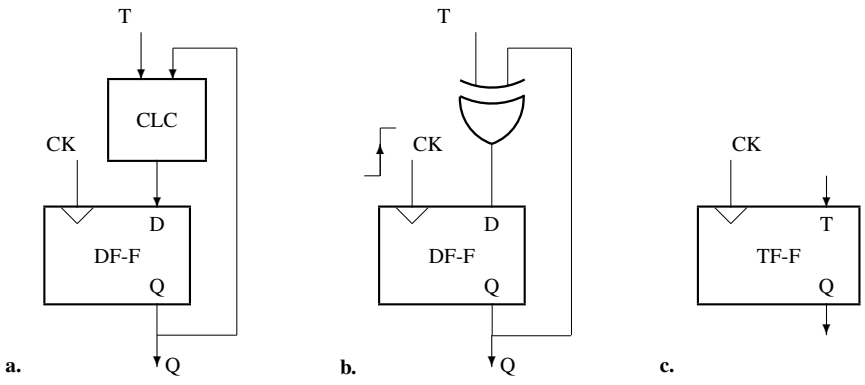


Figure 3.6: **The T flip-flop. a.** It is the simplest automaton because: has 1-bit state register (a DF-F), a 2-input loop circuit (one as automaton input and another to close the loop), and direct output from the state register. **b.** The structure of the T flip-flop: the XOR_2 circuits complements the state if $T = 1$. **c.** The logic symbol.

Let us remember on of the XOR's function: commanded inverter. Then the actual structure of the smallest and simplest automaton is represented in Figure 3.6b.

The numerical function of the TF-F is 2 modulo counter under the command T. If T=1 then the circuit counts, else it preserves the last state.

Generic Counter

The counter is basically a simple half-automaton that has an increment circuit on the reverse connection. Each active edge of the clock loads the incremented state register value into the state register. In real application, the functionality is extended to the following set of operations:

- `op = 00 = nop:  out <= out`
- `op = 01 = reset:  out <= 0`
- `op = 10 = load:  out <= in`
- `op = 11 = count:  out <= down ?  out - 1 :  out + 1`
- `down`: 2's complement of `out` to the increment circuit's input

The structure of the generic counter is represented in Figure 3.7.

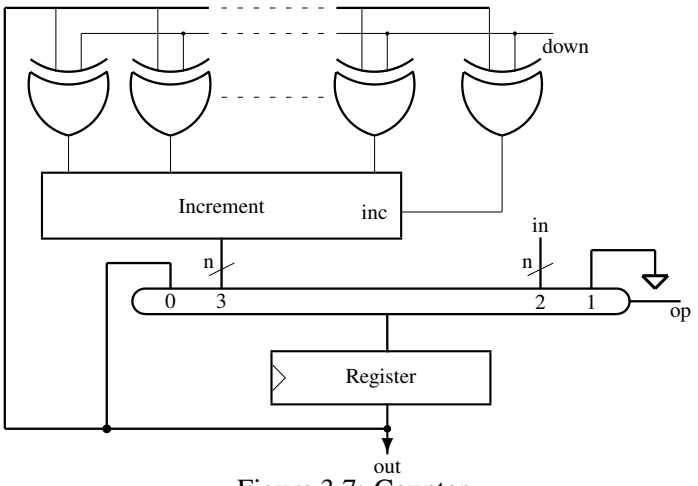


Figure 3.7: Counter.

3.3.2 *Program Counter*

A specific application of the counter simple automaton is the system used to compute the next address in the process of running a program (see Figure 3.8). The set of functions, specified on the input `next`, is:

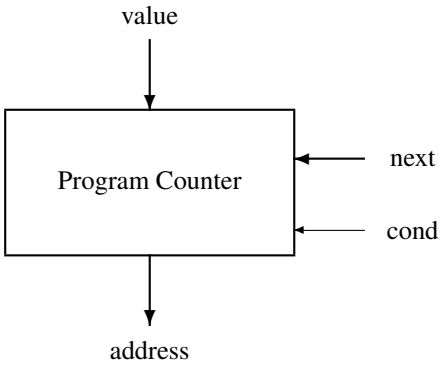


Figure 3.8: Program Counter.

- `nop:` `addres <= address` for `halt` instruction
- `rst:` `address <= 0` for system reset
- `inc:` `address <= address + 1` for program counter increment
- `rjmp:` `address <= address + relValue` for relative jump
- `crjmp:` `address <= (absValue = 0) ? address + relValue : address + 1` for conditioned branch
- `ajmp:` `address <= absValue` for absolute jump

```
/*************************************************************************
File name:      programCounter.sv
Circuit name:   Program Counter
Description:
*************************************************************************/
module programCounter #(parameter n=32)
        (   output  logic [31:0]    progAddr,
            input   logic [2:0]     sel     ,
            input   logic [31:0]    relValue,
            input   logic [31:0]    absValue,
            input   logic           reset   ,
            input   logic           clock   );
    logic [31:0]    pc      ;
    logic [31:0]    nextPC  ;

    always_ff @(posedge clock) if (reset)   pc <= 0     ;
                               else         pc <= nextPC;
    always_comb
     case(sel)
         3'b000: nextPC = pc                                        ;
         3'b001: nextPC = pc + relValue                             ;
         3'b010: nextPC = (absValue == 0) ? pc + relValue : pc + 1  ;
         3'b011: nextPC = (absValue == 0) ? pc + 1 : pc + relValue  ;
         3'b100: nextPC = absValue                                  ;
        default: nextPC = pc + 1                                    ;
     endcase

     assign progAddr = pc       ; // for pipelined version
     //assign progAddr = nextPC ; // for non-pipelined version
endmodule
```

3.3.3 *Registers with Arithmetic & Logic Unit (RALU)*

If we connect in a loop a register file with a logical-arithmetic unit we will obtain the simplest form of an executive core of a processor (see Figure 3.9). The external connections of a Register with Arithmetic & Logic Unit (RALU) type module are:

`func[3:0]` : selects one of the maximum 8 functions performed by ALU

`carryIn` : carry input for arithmetic functions (is borrow for subtract)

`carryOut` : carry output for arithmetic function (is borrow for subtract)

`in[31:0]` : data input

`load` : selects data input as left operand for ALU

`leftAddr[4:0]` : selects left output or left operand for ALU when `load = 0`

`rightAddr[4:0]` : selects right output or right operand for ALU

`clock` : is the clock signal active on the positive edge

`we` : write enable for the register file

`destAddr[4:0]` : destination address for the value `out` provided by ALU

`leftOut[31:0]` : left output of RALU

`rightOut[31:0]` : right output of RALU

According to the connection list, RALU performs no more than 8 functions, on 32-bit words stored in a 32-word register file.

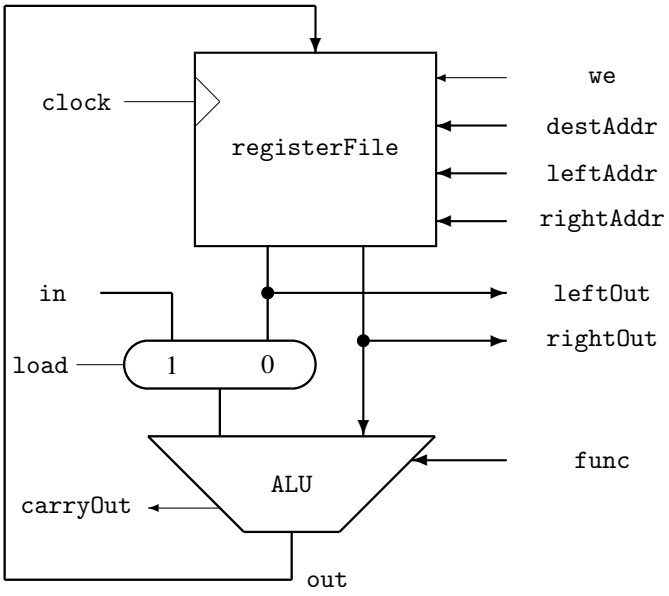


Figure 3.9: 32-bit RALU.

In each clock cycle, the function `func` is applied on two operands selected from the register file and the result, `out`, is loaded back in the register file if `writeEnable = 1`. If `load = 1`, then `leftOp = in`. If `writeEnable = 0`, the content of the register file remains unchanged and the only purpose of the operation is to send toward the external systems the outputs: `carryOut, leftOut, rightOut`.

The code `func` selects the functions defined in the file `define.vh`:

```
/*************************************************************************
File name:      define.vh
Circuit name:
Description:
*************************************************************************/
```

```
`define bigv  4'b0001 // rf[dest] = {rf[left][15:0], rf[right][15:0]}
`define add   4'b0010 // rf[dest] = rf[left] + rf[right]
`define sub   4'b0011 // rf[dest] = rf[left] - rf[right]
`define addcr 4'b0100 // rf[dest] = (rf[left] + rf[right])[32]
`define subcr 4'b0101 // rf[dest] = (rf[left] - rf[right])[32]
`define lsh   4'b1000 // rf[dest] = rf[left] >> 1
`define ash   4'b1001 // rf[dest] = {rf[left][31], rf[left][31:1]}
`define move  4'b1010 // rf[dest] = rf[left]
`define mult  4'b1011 // rf[dest] = rf[left] * rf[right]
`define bwand 4'b1101 // rf[dest] = rf[left] & rf[right]
`define bwor  4'b1110 // rf[dest] = rf[left] | rf[right]
`define bwxor 4'b1111 // rf[dest] = rf[left] ^ rf[right]
```

The top module is:

```
/*************************************************************************
File name:      RALU.sv
Circuit name:   register file with arithmetic and logic unit
Description:
*************************************************************************/
module RALU(output  logic [31:0]    leftOut     ,
            output  logic [31:0]    righttOut   ,
            output  logic           crOut       ,
            input   logic [4:0]     leftAddr    ,
            input   logic [4:0]     rightAddr   ,
            input   logic [4:0]     destAddr    ,
            input   logic [31:0]    in          ,
            input   logic           load        ,
            input   logic           we          ,
            input   logic [3:0]     func        ,
            input   logic           clock       );
    logic [31:0]    out  ;

    regFile rf( .leftOut    (leftOut    ),
                .righttOut  (righttOut  ),
                .in         (out        ),
                .leftAddr   (leftAddr   ),
                .rightAddr  (rightAddr  ),
                .destAddr   (destAddr   ),
                .we         (we         ),
                .clock      (clock      ));

    ALU alu(.func   (func               ),
            .left   (load ? in : leftOut),
            .right  (righttOut          ),
            .crOut  (crOut              ),
            .out    (out                ));
endmodule
```

```
/*************************************************************************
File name:      regFile.sv
Circuit name:   Register File
Description:    Three-pport, 32 32-bit words implemented with a
                synchronous RAM
*************************************************************************/
module regFile( output  logic [31:0]    leftOut     ,
                output  logic [31:0]    righttOut   ,
                input   logic [31:0]    in          ,
                input   logic [4:0]     leftAddr    ,
                input   logic [4:0]     rightAddr   ,
                input   logic [4:0]     destAddr    ,
                input   logic           we          ,
                input   logic           clock       );
    logic [31:0]    mem[0:31]   ;

    always_ff @(posedge clock) if (we) mem[destAddr] <= in  ;

    assign leftOut  = mem[leftAddr] ;
    assign rightOut = mem[rightAddr];

endmodule
```

```
/*************************************************************************
File name:      alu.sv
Circuit name:   arithmetic and logic unit
Description:    the circuit selects, using the selection code 'func', one
                of the 8 functions
*************************************************************************/
`include "define.vh"
module ALU( input   logic [3:0]   func  ,
            input   logic [31:0]  left  ,
            input   logic [31:0]  right ,
            output  logic         crOut ,
            output  logic [31:0]  out   );
    logic [32:0] sum    ;
    logic [32:0] dif    ;

    assign sum = left + right   ;
    assign dif = left - right   ;
    always_comb
        case(func)
            `bigv  : {crOut, out} <= {1'b0, left[15:0], right[15:0]};
            `add   : {crOut, out} <=  sum                           ;
            `sub   : {crOut, out} <=  dif                           ;
            `addcr : {crOut, out} <= {32'b0, sum[32]}               ;
            `subcr : {crOut, out} <= {32'b0, dif[32]}               ;
```

```
            `lsh    : {crOut, out} <= {left[0], left[31:1]}          ;
            `ash    : {crOut, out} <= {left[31], left[31:1]}         ;
            `move   : {crOut, out} <= {1'b0, left}                   ;
            `mult   : {crOut, out} <= {1'b0, left * right}           ;
            `bwand  : {crOut, out} <= {1'b0, left & right}           ;
            `bwor   : {crOut, out} <= {1'b0, left | right}           ;
            `bwxor  : {crOut, out} <= {1'b0, left ^ right}           ;
            default {crOut, out} <=  33'b0 - 1'b1                    ;
        endcase
endmodule
```

Let us use the RALU unit we just defined to exemplify simple calculations.

Example 3.5 *The following sequence o operations: load two numbers (12 and 5) in register file, add them, divide the result by 4, add with 23, and send out the result on the right output.*

Table 3.2: Sequence of commands applied to RALU in Example 3.5

func	in	load	leftAddr	rightAddr	we	destAddr	COMMENT
1111	xx	0	00000	00000	1	00000	R0 <= 0
0010	12	1	xxxxx	00000	1	00001	R1 <= 12+R0
0010	5	1	xxxxx	00000	1	00010	R2 <= 5+R0
0010	xx	0	00001	00010	1	00011	R3 <= R1+R2
1001	xx	0	00011	xxxxx	1	00011	R3 <= R3/2
1001	xx	0	00011	xxxxx	1	00011	R3 <= R3/2
0010	23	1	xxxxx	00011	1	00011	R3 <= R3+23
xxxx	xx	x	xxxxx	00011	0	xxxxx	rightOut = R3

◇

3.4 Problems

3.4.1

3.4.2

Section 4

Processors: Three-Loop, Third-Order Systems (3-OS)

Contents

In this lesson we will accelerate the closing of the loops, but we will focus on the third loop that allows us to introduce the concept of the processor exemplified with a simple engine called ***toyRISC***. We will practice the processor concept in the context provided by the fifth loop, the one that defines the computer concept in the currently practiced understanding, that of Reduced Instruction Set Computer (RISC) type computing systems. Only for those interested and familiar with Verilog HDL, at the end of this text there is an appendix that allows the simulation of the system described in this lesson.

4.1 Architecture vs. Organization

Architecture: describes what the computer does.
Organization: describes how it does it.

In the beginning it was only the hardware and the software. At some point there was a need for an interface that would provide the software with a more friendly development environment. What it is? If in a suite of hardware implementations the set of instructions changes radically with each new version, then the programs written for previous versions are thrown into the trash every time a new hardware version is instantiated. This bad habit is cured with the design, at the beginning of the 1960s, of the IBM System/360. At that moment it was decided that:

> *... to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation.* [Amdahl '64]

And so the Hardware – Software duet turned into the Hardware – Architecture – Software triplet. Architecture being the short form for Instruction Set Architecture.

Table 4.1: Architecture vs. Organization.

Architecture	**Organization**
describes what the computer does	describes how it does it
deals with the functional behavior	deals with a structural relationship
architecture is fixed first	organization is decided after
comprises instruction sets, registers, data types, and addressing modes	consists of multiplexors, adders, multipliers, peripherals, memories, ...
the software developer is aware of it	it escapes the software programmer's detection

4.2 Processor: Three-Loop, Three-Order System (3-OS)

There are several ways to close the third loop over a generic 2-OS represented by an automaton in order to obtain the generic structure o a **processor**. On the closed loop over an automaton we can connect a 0-OS (a combinational circuit), a 1-OS (a memory), or a 2-OS (an automaton). The last version is the most significant (see Figure 4.1). It is the subject of this lesson.

The two machines that configure a processor have distinct functions:

- functional automaton that performs data processing functions; it is a simple automaton
- control automaton that ensures the sequential operation of the processor implemented in two possible versions:
  - a complex controller of the type presented in 3.2.2, when, in addition to fetching the instruction from the program memory, it is also necessary to break it down into a sequence of micro-instructions executed by the functional automaton
  - a simple controller of the counter type presented in 3.3.2, when it is sufficient to bring the instruction from the program memory because the instruction is simple enough to be executed directly by the functional automaton

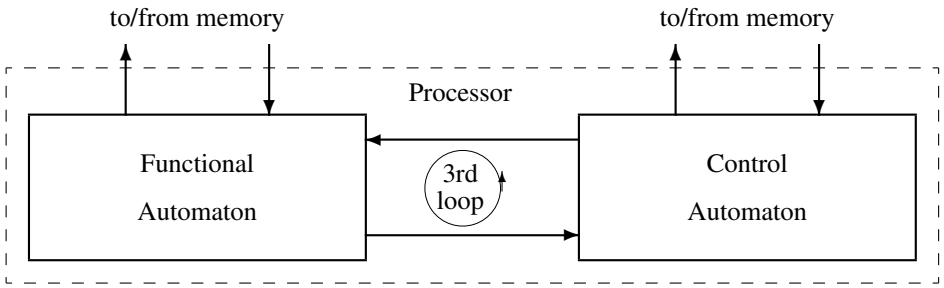


Figure 4.1: Generic Processor

The two types of control generate two types of processors, as follows in the next section.

4.2.1 Interpreting Processor (CISC processor)

The first type of processor, the one that interprets the instruction by decomposing it into a sequence of microinstructions, is represented in Figure 4.2, where the control is performed with a complex automaton. An instruction can involve complex operations implemented sequentially through a series of micro-instructions sent by the controller to the functional automaton that can send back to the controller flags that can characterize the result of the application of each micro-instruction. For this reason, this processor version is called CISC (Complex Instruction Set Computer).

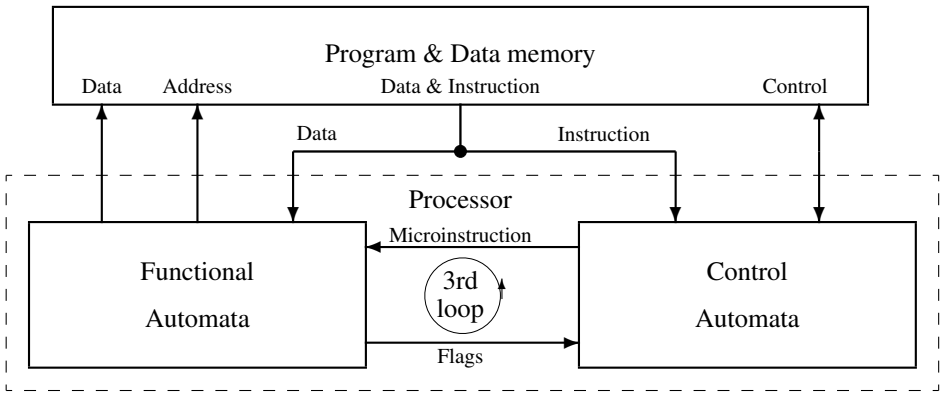


Figure 4.2: Interpreting Processor

For each instruction the CISC processor must perform:

1. instruction fetch
2. instruction execute
3. compute the address for the next instruction

so that an instruction is executed in a variable number of at least several clock cycles.

4.2.2 Executing Processor (RISC processor)

The second type of processor is designed in such a way that it executes each instruction in constant time, usually one clock cycle, but no more than two clock cycles. Because each instruction requires fetching it from memory, and some instructions also require writing or reading data from memory, it is necessary to structure the memory into two sub-memories, one for programs and another for data. This results in the structure in Figure 4.3, where the processor has two access paths to memory resources. In this case, the control automaton can only take care of fetching the instructions from the program memory and the functional automaton will execute sufficiently simple instructions to require only one clock cycle to be operated.

How did you get to this way of designing a processor that executes only simple, executable instructions in one clock cycle? Around 1980, the following facts were evident:

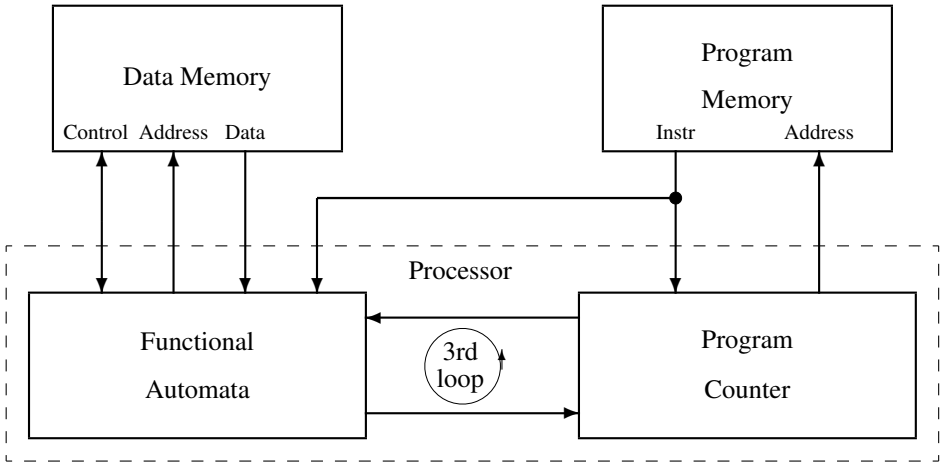


Figure 4.3: Executing Processor

- the frequency with which the instructions of a CISC processor appeared in the programs was very variable: a small number of instructions had a very high frequency of use, and the majority were used with low frequencies
- if the instructions were listed in descending order of frequency of use, then a line could be drawn in this list delimiting a set of frequently used instructions that also had the quality of being able to implement the instructions below the line through a suitable sequencing of them
- the pleasant surprise was that only simple functions were found above the line
- thus, for frequent operations, the effector structure could be imagined to work quickly, and for the less frequent ones, the possibility of their performance is ensured

Thus, a RISC processor executes instead of interpreting. Programs represent sequences of simple instructions, which are executed in parallel with the control of their evolution. Indeed, the functional automaton operates on the data while the control automaton calculates the address from which the next

instruction will be read. Both types of operations fall into the category of simple ones, the fact that it allows us to consider a RISC processor as simple to distinguish it from a CISC one which is complex due to the control automaton.

4.3 von Neumann Computer Version: Four-Loop, Four-Order System (4-OS)

The two types of processors defined in the previous section were used to define two abstract computer models.

Historical note: in the 1940s, the first electronic computers were made in two versions that generated two models for the subsequent evolution. Traditionally, but erroneously, these models are called *architectures*. The concept of architecture appeared in the 1960s. For this reason, we will call these concepts *abstract models* instead of architectures.

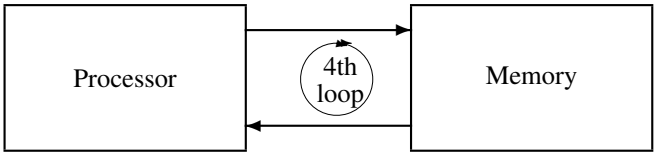


Figure 4.4: The abstract model of computer proposed by John von Neumann.

The interpretive processor is used in the definition of the *von Neumann abstract model*, in which the processor is loop connected to a single memory in which both programs and data are stored (see Figure 4.4). Thus, a 4-OS is obtained by connecting a 3-OS (Processor) with a 1-OS (Memory).

4.4 Harvard Computer Version: Five-Loop, Five-Order System (5-OS)

The Harvard abstract model assumes a fifth loop. The processor together with the program memory forms a 4-OS, and by closing another loop through the data memory, a 5-OS is obtained (see Figure 4.5).

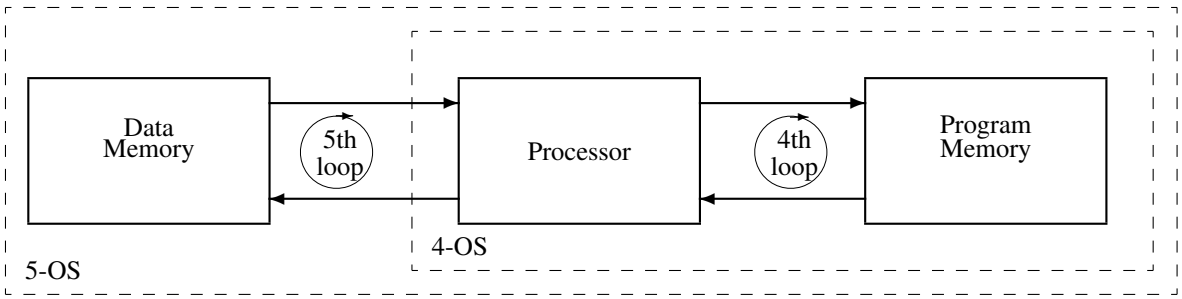


Figure 4.5: The Harvard abstract model of computer.

We will continue to focus, for obvious reasons, on RISC-type processors that allow the implementation of the Harvard abstract model.

4.5 *ToyRISC Processor*

The executing processor is simpler than an interpreting processor. The complexity of computation moves almost completely from the physical structure of the processor into the programs executed by the processor, because this kind of processor (called Reduced Instruction Set Computer, RISC) has an organization containing mainly simple, recursively defined circuits.

4.5.1 Organization

The Harvard abstract model of a RISC executing machine determines the internal structure of the processor to have mechanisms allowing in each clock cycle cu address both, the program memory and the data memory. Thus, the RALU-type functional automaton, directly interfaced with the data memory, is loop-connected with a control automaton designed to fetch in each clock cycle a new instruction from the program memory. The control automaton does not "know" the function to be performed, it "knows" how to "fetch the function" from an external storage support, the program memory.

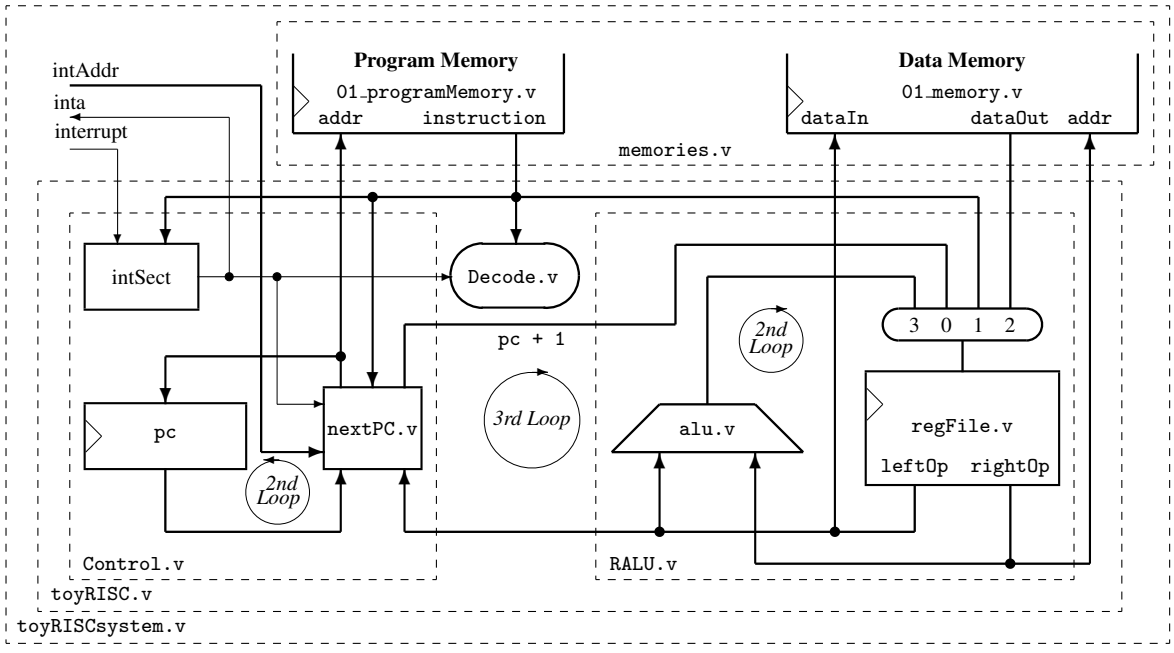


Figure 4.6: The organization of the ***toyRISC*** processor in its simulation environment where the program memory and data memory are included.

Important note: the organization presented in the following is designed only from the circuit point of view with emphasis on the *competence* of the system. In order to maximize the performance the design

must apply specific techniques to increase the frequency of the clock cycle. This concern goes beyond the aims of this lecture. We will partially address these issues in the next lesson. Books on the organization and architecture of computing systems deal extensively with issues of maximizing performance. (Books on the architecture and organization of computers written by John Hennessy and David Patterson are recommended [Hennessy '19] [Patterson '05].)

In Figure 4.6, the organization of the processor ***toyRISC*** is represented. It contains two loop-connected automata, `Control` and `RALU`, serially connected with the with the interrupt section, `intSect`.

Control

The Control section is a simple functional automaton whose state, stored in the register called Program Counter, (`pc`), is used to compute in each clock cycle the address in Program Memory from where the next instruction is fetched. There are two modes to compute the next address: (1) incrementing, with 1 or a signed number, the current address, or (2) independently from the current value of Program Counter, using a value fetched from an internal register or a value generated by the currently executed instruction. More, the current `pc+1` can be stored in an internal register when the control of the program *calls* a function and a return is needed. For all the previously described behaviors the combinational circuit **nextPC** (described in `nextPc.v` module of the design presented in Appendix B) is designed to close the loop over the Program Counter register.

RALU

The RALU section is also a simple functional automaton whose state is structured as an array of variables and is stored in a Register File (a small memory with two output ports and one input port). In each clock cycle two values are fetched from this memory and are submitted to be operated in the ALU unit. The result is write back in a location selected by a destination address. The section accepts data coming form the data memory, from the currently executed instruction, or from the **Control** automaton, thus closing the 3rd loop.

Both, the **Control** automaton and the **RALU** automaton are simple, recursively defined automata. The computational complexity is completely moved in the code stored inside the program memory.

Interrupt section

In computers, an interrupt is a signal (`interrupt`) for the processor to interrupt currently executing code. If the interrupt is accepted, the processor will suspend its current activities, save its state, and execute the function requested by the interrupt. This interruption is temporary, allowing the software to resume normal activities after the program associated to the interrupt finishes.

There are two types of interrupts:

- Hardware interrupts when the interrupt signal generated from external devices (for example, in a keyboard if we press a key to do some action this pressing of the keyboard generates a signal that is given to the processor to do action). They are classified into two types:
  - Maskable Interrupt: interrupts that can be delayed when a highest priority interrupt has occurred to the processor.
  - Non Maskable Interrupt: that cannot be delayed and immediately be serviced by the processor.

- Software interrupts generated from internal devices divided into two types:
  - Normal Interrupts: are caused by the software instructions are called software instructions.
  - Exception: an unplanned interruption while executing a program (for example, while executing a program if we got a value that is divided by zero).

In our very simple example we illustrated a maskable hardware interrupt. The circuit `intSect` consists of a finite automaton with two states, `enableInterrupt` and `disableInterrupt`, controlled with the instructions `ei` and `di`. When the system is reset, the machine switches to the `disableInterrupt` state. If the automaton is in the `enableInterrupt` state and the `interrupt` signal is activated, then `inta` (interrupt acknowledge) is generated, the `pc+1` value is saved in `regFile`, to be able to return the program to the point where it was interrupted, and the instruction from `intAddr` where the program associated with the interruption treatment is located is read. This program ends with an unconditional jump to the address saved in `regFile`.

4.5.2 Instruction Set Architecture

The storage resources on which the ISA definition is based are the following:

```
reg    [31:0]  programMemory[0:65535]  ;
reg    [31:0]  dataMemory[0:1023]      ;
reg    [15:0]  pc                      ; // program counter
reg    [31:0]  registers[0:31]         ; // register file
reg            intEnable               ; // interrupt enable FF
```

The structure of the arithmetic & logic instructions have two forms:

- `destinationRegiste <= leftOperamd OP rightOperand`
- `destinationRegiste <= leftOperamd OP immediateValue`

with the following two formats:

```
instruction[31:0] = {opCode[5:0],       // operation code
                     dest[4:0],         // destination register
                     left[4:0],         // left operand
                     right[4:0],        // right operand
                     noUse[10:0]}
                    |
                    {opCode[5:0],
                     dest[4:0],
                     left[4:0],
                     immediate[15:0]}   // value
```

Instruction Set Architecture, ISA, of the ***toyRISC*** processor is described in the Figure 4.7. ISA, in short the architecture of a computing system includes at least the following types of instructions:

1. control instructions

2. arithmetic and logic instructions
3. memory access instructions
4. interrupt control instructions

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

Figure 4.7: toyRISC ISA: `0_ISAdefine.vh`.

The first field of the instruction, `opCode`, contains the information used to determine what is the the action performed by the current instruction. Each instruction is executed in one clock cycle. In Appendix B the entire design is listed. This design is meant to illustrate mainly the competence of the circuits without aiming to reach some performances in execution. In order to maximize the performance,

it is necessary to apply special techniques (usually in the category of pipelines) related to the field of quantitative optimization of the organization of the computer system. See more on the quantitative approach in [Hennessy '07].

4.5.3 Assembly Code

In order to generate the code executable by the toyRISC processor, a code generator translates the mnemonics into binary code. In Table 4.2, the mnemonics used to write assembly programs are listed. In the second column the action performed specified and in the last column the binary codes are listed. The binary code is organized in 5 or 4 fields. The first field represents the operation code, the second the destination address of the result in the register file (`ddddd` is the binary code of the actual value when it matters), the third field represents the address for the left operand (`lllll` is the binary code of the actual value when it matters). The least significant 16 bits are organized in 2 fields or in one field. In the first case 5 bits (`rrrrr` is the binary code of the actual value when it matters) represents the address of the right operand and the rest 11 bits have no meaning, while in the second case all the 16 bits represents a value (`vvvvvvvvvvvvvvvv` is the binary code of the actual value when it matters).

Table 4.2: Assembly language mnemonics, their meanings and binary form.

Mnemonics	Description	Binary form
NOP	pc <= pc + 1	000000_00000_00000_00000_00000000000
RJMP(label)	pc <= pc + value	000001_00000_00000_vvvvvvvvvvvvvvvv
BRZ(left,label)	pc <= (left = 0) ? pc+immediate : pc+1	000010_00000_lllll_vvvvvvvvvvvvvvvv
BRNZ(left,label)	pc <= (left = 0) ? pc+1 : pc+immediate	000011_00000_lllll_vvvvvvvvvvvvvvvv
RET	pc <= rf[left]	000101_00000_lllll_00000_00000000000
HALT	pc <= pc	000110_00000_00000_00000_00000000000
EINT	intEnable <= 1 (enable interrupt)	001000_00000_00000_00000_00000000000
DINT	intEnable <= 0 (disable interrupt)	001001_00000_00000_00000_00000000000
ADD(dest,left,right)	rf[dest] <= rf[left] + rf[right]	110000_ddddd_lllll_rrrrr_00000000000
SUB(dest,left,right)	rf[dest] <= rf[left] - rf[right]	110001_ddddd_lllll_rrrrr_00000000000
ADDV(dest,left,value)	rf[dest] <= rf[left] + value	110010_ddddd_lllll_vvvvvvvvvvvvvvvv
MULT(dest,left,right)	rf[dest] <= rf[left] * rf[right]	110011_ddddd_lllll_rrrrr_00000000000
MULTV(dest,left,value)	rf[dest] <= rf[left] * rf[right]	110100_ddddd_lllll_vvvvvvvvvvvvvvvv
ADDC(dest,left,right)	rf[dest] <= (rf[left] + rf[right])[32]	110101_ddddd_lllll_rrrrr_00000000000
SUBC(dest,left,right)	rf[dest] <= (rf[left] - rf[right])[32]	110110_ddddd_lllll_rrrrr_00000000000
ADDVC(dest,left,value)	rf[dest] <= (rf[left] + value)[32]	110111_ddddd_lllll_vvvvvvvvvvvvvvvv
LSH(dest,left)	rf[dest] <= {0, rf[left][31:1]}	111000_ddddd_lllll_00000_00000000000
ASH(dest,left)	rf[dest] <= {rf[left][31],rf[left][31:1]}	111001_ddddd_lllll_00000_00000000000
MOVE(dest,left)	rf[dest] <= rf[left]	111010_ddddd_lllll_00000_00000000000
SWAP(dest,left)	rf[dest] <= rf[left][15:0],rf[left][31:16]	111011_ddddd_lllll_00000_00000000000
NOT(dest,left)	rf[dest] <= rf[left]	111100_ddddd_lllll_00000_00000000000
AND(dest,left,right)	rf[dest] <= rf[left & rf[right]	111101_ddddd_lllll_rrrrr_00000000000
OR(dest,left,right)	rf[dest] <= rf[left] | rf[right]	111110_ddddd_lllll_rrrrr_00000000000
XOR(dest,left,right)	rf[dest] <= rf[left] $\oplus$ rf[right]	111111_ddddd_lllll_rrrrr_00000000000
READ(left)	read from dataMemory[left]	100000_00000_lllll_00000_00000000000
LOAD(dest)	rf[dest] <= dataOut	100111_ddddd_00000_00000_00000000000
STORE(left,right)	dataMemory[left] <= rf[right]	101000_00000_lllll_rrrrr_00000000000
VAL(dest,val)	rf[dest] <= {11val[20], value}	010111_ddddd_00000_vvvvvvvvvvvvvvvv

Toy Assembler

The binary codes in the 4.2 table are generated, for reasons of maintaining a simple design and simulation environment, using a program also written in System Verilog. This program, `RISCcodeGenerator.sv`,

receives a sequence of instructions in the file `program.sv` and generates the executable binary code in the program memory `progMemory`. A portion of this generator is listed below to illustrate the idea of two-pass encoding; the whole is attached in the B.2 section. In the first pass, the absolute positions of the labels are identified through the `LB` task in the `labelTab` table, and in the second pass through the `ULB` task, the relative jump is calculated, which is correctly completed in the code binary. The two tasks are necessary because some relative jumps are at advanced locations compared to the position of the jump instruction.

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

    `include "DEFINES.vh"

    task endLine; // assemble the executable code line by line
     begin
        progMemory[addrCounter][31:0] =
            {   opCode  ,
                d       ,
                l       ,
                v       }                   ;
            addrCounter = addrCounter + 1   ;
     end
    endtask

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

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

        v = labelTab[labelIndex] - addrCounter;
    endtask

// CONTROL INSTRUCTIONS
    task NOP; // no operation
```

```
        begin   opCode  = `addv ;
                d       = 5'b0  ;
                l       = 5'b0  ;
                v       = 16'b0 ;
                endLine         ;
        end
    endtask

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

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

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

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

    // ...


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

    // ...

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

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

    // ...

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

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

    // ...



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

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

    // ...


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

        begin   opCode  = `read ;
                d       = 5'b0  ;
```

```
                l       = left  ;
                v       = 16'b0 ;
                endLine         ;
        end
    endtask

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

SystemVerilogSummary 10 :

task taskName; **input** [...:...] inputName; ... **begin** taskBody **end endtask** : describes the task `taskName` of parameters `inputName` ... which performs the actions defined by `taskBody`.

Simulator

The simulator used to validate the processor design is described in detail in the **??** section, where the processor was instantiated and the data, `dataMemory`, and program, `ProgMemory` memories were added. The interrupt signal `intIn` has been activated, which will be taken into account immediately by the instruction `DI` will allow it.

Assembly Programs

Example 4.1 *Let by a following simple program:*

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

The code generator loads in the program memory the following binary form of the program:

```
progMemory[0]    = 11001000000000000000000000000000
progMemory[1]    = 01011100000000000000000000000001
progMemory[2]    = 01011100001000000000000000000010
```

```
progMemory[3]    = 01011100010000000000000000000011
progMemory[4]    = 01011100011000000000000000000100
progMemory[5]    = 01011100100000000000000000000101
progMemory[6]    = 11000000000000000000100000000000
progMemory[7]    = 11000000000000000001000000000000
progMemory[8]    = 11000000000000000001100000000000
progMemory[9]    = 11000000000000000010000000000000
progMemory[10]   = 00011000000000000000000000000000
progMemory[11]   = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

The clock period is equal with 2 time units. The program starts at the time unit `t=5` *after the 4 time units action of the* `reset` *signal. The value of the program counter and of the first 2 registers in the register file evolve as follows:*

```
t=0  \bigotimespc=   x  RF=[x, x, x, x] ei=x inta=x
t=1  pc=1023  RF=[x, x, x, x] ei=0 inta=0
t=5  pc=   0  RF=[x, x, x, x] ei=0 inta=0
t=7  pc=   1  RF=[x, x, x, x] ei=0 inta=0
t=9  pc=   2  RF=[1, x, x, x] ei=0 inta=0
t=11 pc=   3  RF=[1, 2, x, x] ei=0 inta=0
t=13 pc=   4  RF=[1, 2, 3, x] ei=0 inta=0
t=15 pc=   5  RF=[1, 2, 3, 4] ei=0 inta=0
t=17 pc=   6  RF=[1, 2, 3, 4] ei=0 inta=0
t=19 pc=   7  RF=[3, 2, 3, 4] ei=0 inta=0
t=21 pc=   8  RF=[6, 2, 3, 4] ei=0 inta=0
t=23 pc=   9  RF=[10, 2, 3, 4] ei=0 inta=0
t=25 pc=  10  RF=[15, 2, 3, 4] ei=0 inta=0
```

◇

Example 4.2 *Let us take an example to show how the interrupt acts. The interrupt is applied from the beginning of the test, but it is enabled only after the instruction* `EI` *runs.*

```
            VAL(31,10)  ;
            VAL(2,23)   ;
            VAL(0,13)   ;
            EI          ;
            ADDV(0,0,2) ;
            NOP         ;
            ADDV(0,0,4) ;
            HALT        ;
            NOP         ;
            NOP         ;
// subroutine triggered by interrupt
            DI          ;
            VAL(3,44)   ;
            RET(30)     ;
```

The toy assembler generates in `progMemory` *the following executable code:*

```
progMemory[0]    = 01011111111000000000000000001010
progMemory[1]    = 01011100010000000000000000010111
progMemory[2]    = 01011100000000000000000000001101
```

```
progMemory[3]    = 00100000000000000000000000000000
progMemory[4]    = 11001000000000000000000000000010
progMemory[5]    = 11001000000000000000000000000000
progMemory[6]    = 11001000000000000000000000000100
progMemory[7]    = 00011000000000000000000000000000
progMemory[8]    = 11001000000000000000000000000000
progMemory[9]    = 11001000000000000000000000000000
progMemory[10]   = 00100100000000000000000000000000
progMemory[11]   = 01011100011000000000000000101100
progMemory[12]   = 00010100000111100000000000000000
progMemory[13]   = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

The simulation provides:

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

◇

Example 4.3 *Let be a wrong program which runs forever because of an unconditional jump.*

```
          VAL(0,3)    ;
  LB(1);  ADDV(0,0,-1);
          NOP         ;
          RJMP(1)     ;
          HALT        ;
```

The simulation provides:

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

```
t=25 pc=   1  RF=[0, x, x, x] ei=0 inta=0
t=27 pc=   2  RF=[4294967295, x, x, x] ei=0 inta=0
t=29 pc=   3  RF=[4294967295, x, x, x] ei=0 inta=0
t=31 pc=   1  RF=[4294967295, x, x, x] ei=0 inta=0
t=33 pc=   2  RF=[4294967294, x, x, x] ei=0 inta=0
t=35 pc=   3  RF=[4294967294, x, x, x] ei=0 inta=0
t=37 pc=   1  RF=[4294967294, x, x, x] ei=0 inta=0
t=39 pc=   2  RF=[4294967293, x, x, x] ei=0 inta=0
t=41 pc=   3  RF=[4294967293, x, x, x] ei=0 inta=0
t=43 pc=   1  RF=[4294967293, x, x, x] ei=0 inta=0
```

◇

Example 4.4 *Let be a program which works with* `dataMemory`.

```
VAL(0,1)    ;
VAL(1,55)   ;
STORE(0,1)  ;
READ(0)     ;
LOAD(2)     ;
HALT        ;
```

The simulation provides:

```
t=0  pc=   x  RF=[x, x, x, x]   ei=x inta=x
t=1  pc=1023  RF=[x, x, x, x]   ei=0 inta=0
t=5  pc=   0  RF=[1, x, x, x]   ei=0 inta=0
t=7  pc=   1  RF=[1, x, x, x]   ei=0 inta=0
t=9  pc=   2  RF=[1, 55, x, x]  ei=0 inta=0
t=11 pc=   3  RF=[1, 55, x, x]  ei=0 inta=0
t=13 pc=   4  RF=[1, 55, x, x]  ei=0 inta=0
t=15 pc=   5  RF=[1, 55, 55, x] ei=0 inta=0
```

◇

4.5.4 Time performance

The longest combinational path in a system using our ***toyRISC***, which imposes the minimum clock period, is:

$$T_{clock} = t_{clock\_to\_instruction} + t_{leftAddr\_to\_leftOp} + t_{throughALU} + t_{throughMUX} + t_{fileRegSU}$$

Because the system is not buffered the clock frequency depends also by the time behavior of the system directly connected with ***toyRISC***. In this case $t_{clock\_to\_instruction}$ – the access time of the program memory, related to the active edge of the clock – is an extra-system parameter limiting the speed of our design. The internal propagation time to be considered are: the read time from the file register ($t_{leftAddr\_to\_leftOp}$ or $t_{rightAddr\_to\_rightOp}$), the maximum propagation time through ALU (dominated by the time for an 32-bit arithmetic operation), the propagation time through a 4-way 32-bit multiplexer, and the *set-up time* on the file register's data inputs. The way from the output of the file register through **Next PC** circuit is "shorter" because it contains a 16-bit adder, comparing with the 32-bit one of the ALU.

The increase in speed performance will be illustrated in the next lesson by using pipeline registers.

4.6 How is Designed an Instruction Set Architecture

As we already said, we use the term ISA to refer to the actual programmer-visible instruction set. The ISA serves as the boundary between the software engineer and hardware engineer. We will list and comment further on the main characteristics of an ISA [**?**]:

- Class of ISA
  - register-memory ISAs such as x86, with complex memory access instructions
  - load-store ISAs such as ARM, RISC V, with only simple load and store insructions for memory access
- Memory addressing is byte addressing with two versions:
  - byte aligned (ex.: ARM) if the object accessed at address A has s bytes, then the $Amods = 0$
  - with no alignment (ex.: x86 and RISC V) which produces slower access
- Addressing modes
  - few simple: register, immediate, displacement, ... (RISC V, ARM)
  - many complex: the previous and more (x86)
- Types and sizes of operands
  - integer: 8 bit (ASCII), 16 bit (Unicode character), 32-bit (word), 64-bit (double word)
  - floats: standard IEEE 754 32-bit floating point and 64-bit floating point
- Operations:
  - data transfer
  - arithmetic
    - integer: add, sub, mult, div, rem, rightShift, ...
    - floats: add, sub, mult, div, rem
  - logic: minimally AND and XOR
  - control: jumps, conditional branches, calls, ret, ...
- Control flow instructions:
  - RISC V tests the value of a register for conditional branches
  - x86 and ARM test flags in a state register
  - x86 save the return address from subroutine on a stack memory organized in the main mamory
- Encoding an ISA:
  - variable length for x86
  - fix length for RISC V and ARM

4.7 Problems

4.7.1

4.7.2

Section 5

Instruction-Level Parallelism

Contents

There are various form of instruction-level parallelism. We will consider is this short course only the parallelism introduced by the pipeline mechanism and the multiple execution units. Thus, the 5th lesson deals with the pipeline mechanism, used to accelerate the processor's operation, and the implications of its use. But as we will see any gain obtained from the application of an improvement technique comes with a price.

5.1 Pipelining

Combinational logic circuits (CLCs) that have a very high depth cause the system clock frequency to be reduced. The solution that allows increasing the clock frequency is the introduction of pipeline registers.

5.1.1 Pipeline Acceleration

In Figure 5.1 the pipeline technique is illustrated. In Figure 5.1a, the frequency at which the system clock can work is given by the propagation time through the $CLC_{f \circ g}$ to which is added the time associated with the propagation through the registers.

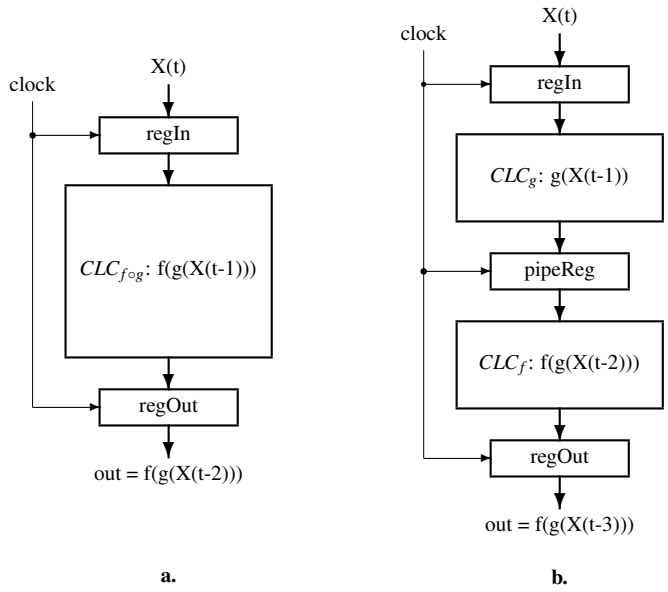


Figure 5.1: Pipelining. **a.** The initial circuit, without pipeline register. **b.** The pipelined version of the circuit.

$$T_{clock} = t_{regProp} + t_{CLC_{f\circ g}} + t_{set-up} \simeq t_{CLC_{f\circ g}}$$

In Figure 5.1b, the frequency at which the system clock can work will be increased because the period of the clock is given by

$$T_{clock} = t_{regProp} + max(t_{CLC_f}, t_{CLC_g}) + t_{set-up}$$

The resulting acceleration is:

$$\alpha \simeq \frac{t_{CLC_{f\circ g}}}{max(t_{CLC_f}, t_{CLC_g})}$$

The maximum efficiency, $\alpha \simeq 2$, is obtained when

$$t_{CLC_f} \simeq t_{CLC_g}$$

If the resulting frequency is not high enough, the technique is applied to the deepest circuit or both. The process can continue until the requested speed is reached.

5.1.2 *Pipelined Version of toyRISC*

Structure

The operating frequency of the toyRISC processor presented in the previous lesson can be increased by inserting some pipeline registers on the critical path of the combinatorial propagation. A version, adapted to be accelerated, is shown in Figure 5.2 where the two memories are not represented. Instead, only the connections to those memories are represented. The program memory is a synchronous memory (see 2.4.2) connected in our system in combinational mode through the output multiplexers. The data

memory is a synchronous pipelined memory (see 2.4.3) which responds with one cycle latency to the read command.

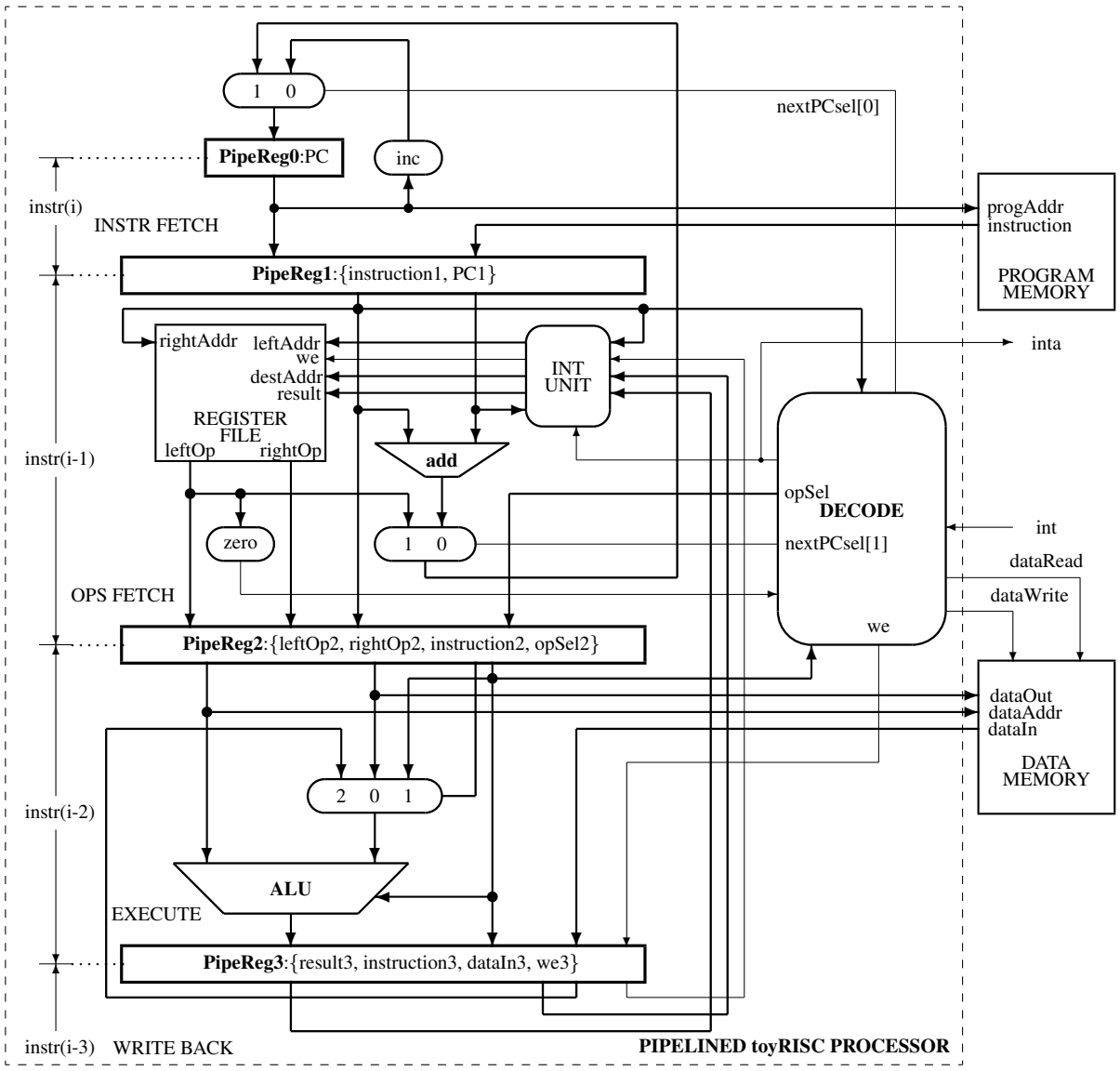


Figure 5.2: The pipelined version of toyRISC.

Micro-architecture

The micro-architecture is:

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

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

The three pipeline registers introduced in design divide the execution of an instruction in four stages:

INSTRUCTION FETCH (IF) : the content of the register `pc` (program counter) addresses in the program memory the binary code of an instruction; in the same time the module `next` computes the value for the next `pc`. The execution time for this stage is:

$$t_{IF} = t_{pc} + max(t_{next}, t_{ACCtoProgramMemory}) + t_{suPipeReg1}$$

In the pipeline register **PipeReg_1** is loaded the information requested to finish the execution of the fetched instruction

OPERANDS FETCH (OF) : from the register are fetched the two operands of the instruction using the fields `leftAddr` and `rightAddr` fetched in the previous cycle from the program memory. In

pipeReg_2 are loaded the two fetched operands and the information needed in the next cycles to end the execution of the instruction. The execution time for this stage is:

$$t_{OF} = t_{PipeReg1} + t_{regFile} + t_{suPipeReg2}$$

EXECUTION (EX) : ALU performs the operation according to the code `opCode`, propagated to this stage through the two previous pipeline registers, generate `result` and the predicate `zero = (rezult == 0)`; the `add` module adds to the program counter the signed value of `value` to perform the branch operation. The execution time for this stage is:

$$t_{EX} = t_{PipeReg2} + max((t_{mux} + t_{alu}), (t_{add} + t_{mux}), t_{dataMemPipeReg}) + t_{suPipeReg3}$$

WRITE BACK (WB) : acts on the first two levels in the pipeline: it allows the modification of the `pc` to perform jumps and writes the result of the current operation in the register file. The execution time for this stage is:

$$t_{WB} = t_{PipeReg3} + max(t_{suRegFile}, (t_{next} + t_{suPC}))$$

The maximum clock frequency is limited by:

$$T_{clock} = max(t_{IF}, t_{OF}, t_{EX}, t_{WB})$$

Architecture

The Instruction Set Architecture is:

```
/*****************************************************************************
                    toyRISC'S ARCHITECTURE
*****************************************************************************/
    NOP         // no operation
    RJMP(lb)    // relative jumpto label 'lb'
    BRZ(l,lb)   // branch if rf[l]=zero at label 'lb'
    BRNZ(l,lb)  // branch if rf[l]!=zero at label 'lb'
    RET(l)      // return from subroutine: pc<=rf[l]
    HALT        // halt until interrupt is received, pc = pc
 // for the following instructions: pc<=pc+1;
    EINT        // set enable interrupt
    DINT        // set disable interrupt
    ADD(d,l,r)  // rf[d]<=rf[l]+rf[r];
    SUB(d,l,r)  // rf[d]<=rf[l]-rf[r];
    ADDV(d,l,v) // rf[d]<=rf[l]+v;
    MULT(d,l,r) // rf[d]<=rf[l]*rf[r];
    MULTV(d,l,v)// rf[d]<=rf[l]*v;
    ADDC(d,l,r) // rf[d]<=(rf[l]+rf[r]}[32];
    SUBC(d,l,r) // rf[d]<=(rf[l]-rf[r])[32];
    ADDVC(d,l,v)// rf[d]<=(rf[l]+v)[32];
    LSH(d,l)    // rf[d]<=rf[l] >> 1;
    ASH(d,l)    // rf[d]<={rf[l][31],rf[l][31:1]};
```

```
MOVE(d,l)   // rf[d]<=rf[l];
SWAP(d,l)   // rf[d]<={rf[l][15:0],rf[l][31:16]};
NOT(d,l)    // rf[d]<=~rf[l];
AND(d,l,r)  // rf[d]<=rf[l]&rf[r];
OR(d,l,r)   // rf[d]<=rf[l]|rf[r];
XOR(d,l,r)  // rf[d]<=rf[l]^rf[r];
READ(l)     // read from dataMemory[rf[l]];
LOAD(d)     // rf[d]<=dataOut;
STORE(l,r)  // dataMemory[rf[l]]<=rf[r];
VAL(d,v)    // rf[d]<={{16*{v[15]}},v};
```

5.1.3 Latency

In each clock cycle the execution of one instruction ends with a latency of three clock cycles. The entire structure performs in parallel 4 successive instructions, each stage being involved in the execution of one instruction.

Example 5.1 *Let us see how is executed the following sequence of instructions:*

```
XOR(5,0,0)
VAL(1,12)
SUB(2,3,4)
ADD(0,0,3)
AND(1,3,2)
```

In Table 5.1 the distribution along the pipe of the execution is represented. Each line in the table represents a pipeline level. The execution of the first instruction, XOR*, starts in the i-th clock cycle, with* XOR0*, and ends in the* $(i+3)$*-th cycle with* XOR3*. Then in each clock cycle one of the following instruction ends.*

Table 5.1: Pipelined execution.

Pipe Stage \ Time	t=i	t=i+1	t=i+2	t=i+3	t=i+4	t=i+5	t=i+6	t=i+7
Pipe0:IF	XOR0	VAL0	SUB0	ADD0	AND0			
Pipe1:OF		XOR1	VAL 1	SUB1	ADD1	AND1		
Pipe2:EX			XOR2	VAL2	SUB2	ADD2	AND2	
Pipe3:WB				XOR3	VAL3	SUB3	ADD3	AND3

◇

In each clock cycle, an instruction completes with a latency of 3 clock cycles. The good news: the clock frequency increases significantly due to pipeline organization. The bad news: the latency introduced by the pipeline organization creates unwanted dependencies.

5.2 Hazards Generated by Dependencies

There are three types of dependencies:

1. Structural Dependency
2. Data Dependency
3. Control Dependency

These dependencies generate hazardous behaviors of the pipelined processor due to the parallel execution of instructions which partially overlap. In the following some of them will be illustrated. Due to the fact that we used the Harvard abstract model for our toyRISC processor, main structural dependency are avoided. Therefore we concentrate only to the next two type of dependencies.

Structural dependencies are typically represented by those related to processor memory. In the case of our processor, toyRISC, they do not appear because we have an abstract model of the Harvard type with two memories, one for programs and another for data. Thus, in what follows, we will focus on the other two types of dependencies.

5.2.1 Data dependency

When the current instruction uses values generated by a previous instruction that has not been completed, the effect of data dependency appears. There are 3 types of data dependencies that we've been talking about:

- RAW: Read after Write
- WAR: Write after Read
- WAW: Write after Write

We will focus on the first, RAW, which is typical for our processor version. For the other two, they are illustrated in Example 5.11.

Example 5.2 *Let us see how is executed the following sequence of instructions:*

```
XOR(5,0,0)
VAL(1,12)
SUB(2,3,4)
ADD(0,2,1)
AND(1,1,2)
```

In Table 5.2 the distribution along the pipe of the execution is represented.

Table 5.2: Data dependency in the pipelined execution .

Pipe Stage \ Time	t=i	t=i+1	t=i+2	t=i+3	t=i+4	t=i+5	t=i+6	t=i+7
Pipe0:IF	XOR0	VAL0	SUB0	ADD0	AND0			
Pipe1:OF		XOR1	VAL1	SUB1	ADD1	AND1		
Pipe2:EX			XOR2	VAL2	SUB2	***ADD2***	AND2	
Pipe3:WB				XOR3	VAL3	***SUB3***	ADD3	AND3

The WB cycle for SUB, SUB3*, is executed at* i+5*, too late to have the content of* rf2 *actualized with the result of* SUB(2,3,4) *for the instruction* ADD(0,2,1) *which is supposed to have* rf2 *actualized*

by the previous instruction in `i+4`. `ADD2` *must be preceded by* `SUB3`. `SUB3` *is delayed with one clock cycle for a proper execution of the code. The instruction* `ADD(0,2,1)` *uses the value in* `rf2` *before the execution of* `SUB2,3,4`.

◇

Example 5.3 *Data dependency is illustrated also by running on our simulator the program run in Example 4.1:*

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

The result is correct because of the two `NOP`*s inserted in the program from Example 4.1 before the* `ADD` *instruction:*

```
t=0  pc=   x  RF=[x, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=x
t=1  pc=1023  RF=[x, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=x
t=5  pc=   0  RF=[x, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=x
t=7  pc=   1  RF=[x, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=x
t=9  pc=   2  RF=[x, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=x
t=11 pc=   3  RF=[x, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=x
t=13 pc=   4  RF=[x, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=1
t=15 pc=   5  RF=[1, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=2
t=17 pc=   6  RF=[1, 2, x, x, x, x, x, x] leftOp2=1 rightOp2=1 result3=3
t=19 pc=   7  RF=[1, 2, 3, x, x, x, x, x] leftOp2=1 rightOp2=1 result3=4
t=21 pc=   8  RF=[1, 2, 3, 4, x, x, x, x] leftOp2=1 rightOp2=1 result3=5
t=23 pc=   9  RF=[1, 2, 3, 4, 5, x, x, x] leftOp2=1 rightOp2=1 result3=1
t=25 pc=  10  RF=[1, 2, 3, 4, 5, x, x, x] leftOp2=5 rightOp2=4 result3=1
t=27 pc=  10  RF=[1, 2, 3, 4, 5, x, x, x] leftOp2=1 rightOp2=1 result3=9
t=29 pc=  10  RF=[1, 2, 3, 4, 5, 9, x, x] leftOp2=1 rightOp2=1 result3=1
```

If the two `NOP`*s are omitted,*

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

then the processor behaves incorrectly, as follows:

```
t=0  pc=   x  RF=[x, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=x
t=1  pc=1023  RF=[x, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=x
t=5  pc=   0  RF=[x, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=x
t=7  pc=   1  RF=[x, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=x
t=9  pc=   2  RF=[x, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=x
t=11 pc=   3  RF=[x, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=x
t=13 pc=   4  RF=[x, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=1
t=15 pc=   5  RF=[1, x, x, x, x, x, x, x] leftOp2=x rightOp2=x result3=2
t=17 pc=   6  RF=[1, 2, x, x, x, x, x, x] leftOp2=1 rightOp2=1 result3=3
t=19 pc=   7  RF=[1, 2, 3, x, x, x, x, x] leftOp2=1 rightOp2=1 result3=4
t=21 pc=   8  RF=[1, 2, 3, 4, x, x, x, x] leftOp2=x rightOp2=x result3=5
t=23 pc=   8  RF=[1, 2, 3, 4, 5, x, x, x] leftOp2=1 rightOp2=1 result3=x
t=25 pc=   8  RF=[1, 2, 3, 4, 5, x, x, x] leftOp2=1 rightOp2=1 result3=1
```

because the operands for the arithmetic operations are not yet in place being delayed on the execution pipe.

◇

To solve the problem of data dependency just emphasized there are used two solutions: stalling and forwarding.

Stalling

An inefficient solution, for the execution time, is to stall by introducing a `NOP` instruction between `SUB(2,3,4)` and `ADD(0,2,1)`, thus delaying the use of the content of `rf2` with one clock cycle. Result the following sequence of instructions:

```
XOR(5,0,0)
VAL(1,12)
SUB(2,3,4)
NOP
ADD(0,2,1)
AND(1,1,2)
```

whose execution is illustrated in Table 5.3.

Table 5.3: Stalling

Instr \ Time	t=i	t=i+1	t=i+2	t=i+3	t=i+4	t=i+5	t=i+6	t=i+7	t=t+8
IF	XOR0	VAL0	SUB0	NOP0	ADD0	AND0			
OF		XOR1	VAL1	SUB1	NOP1	ADD1	AND1		
EX			XOR2	VAL2	SUB2	NOP2	***ADD2***	AND2	
WB				XOR3	VAL3	***SUB3***	NOP3	ADD3	AND3

Now, `ADD2` is preceded by `SUB3` and addition is performed takeing into account the result of the subtract operation.

Reordering

Sometimes, **but only sometimes**, there is a very simple and efficient solution for data dependency: reordering the instructions in the sequence of instructions. In Example 5.2, the instruction `XOR(5,0,0)` can be moved, without affecting the result of running the sequence of instructions, between the instructions `SUB` and `ADD`. as follows:

```
VAL(1,12)
SUB(2,3,4)
XOR(5,0,0)
ADD(0,2,1)
AND(1,1,2)
```

Thus, the `XOR` instruction is used for stalling instead of the `NOP` instruction, but now the execution time is not affected.

Forwarding

By adding hardware, the data dependency can be resolved, **in all cases**, with no time penalty. The solution is to connect `result` not only to the register file input, but also directly to the ALU input when the value just computed is needed for the next instruction. The mechanism is called *forwarding* and requires the addition of one input to the ALU input as left operand or as right operand. But, because there are binary operations (operations with two operands) sometimes both operands arrive to late in the register file to be fetched for the current instruction.

Thus, forwarding control is done by a circuit which analyse the binary code of three successive instructions. Three situations can occur:

```
xxxxxx_00001_xxx...x                // identified in PipeReg_3
xxxxxx_yyyyy_00001_xxx...x          // identified in PipeReg_2
```

or

```
xxxxxx_00001_xxx...x                // identified in PipeReg_3
xxxxxx_yyyyy_zzzzz_00001_xxx...x    // identified in PipeReg_2
```

or

```
xxxxxx_00001_xxx...x                // identified in PipeReg_3
xxxxxx_yyyyy_00001_00001_xxx...x    // identified in PipeReg_2

xxxxxx_00001_xxx...x                // identified in PipeReg_4
xxxxxx_00011_xxx...x                // identified in PipeReg_3
xxxxxx_yyyyy_00011_00001_xxx...x    // identified in PipeReg_2

xxxxxx_00001_xxx...x                // identified in PipeReg_4
xxxxxx_00011_xxx...x                // identified in PipeReg_3
xxxxxx_yyyyy_00001_00011_xxx...x    // identified in PipeReg_2
```

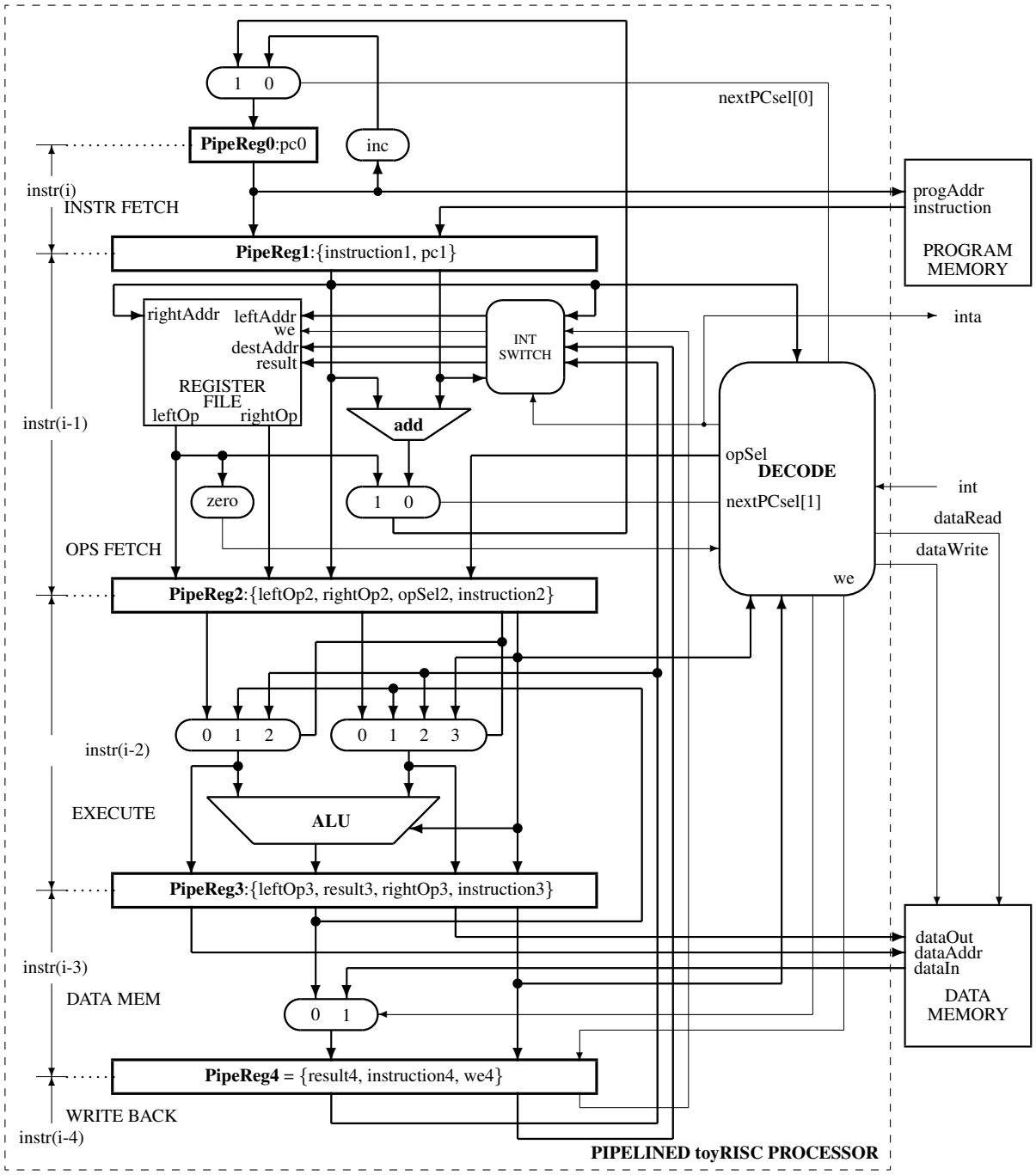


Figure 5.3: The change used to reduce hazards imposed by data dependencies.

The structure of the processor is modified as it is shown in Figure 5.3, where a new level in pipe is added and the operands at ALU inputs are selected using multiplexors with additional inputs. The

selection for the augmented multiplexors are computed combinational as `opSel[3:0]` in `DECODE` and synchronized in **PipeReg2** as `opSel2[3:0]` to avoid an additional delay in selecting ALU operands. Thus, `result` is used as operand in the EXECUTE state of the pipeline, if the forwarding mechanism decides, as `result3` connected the inputs 1 or as `result4` to the inputs 2 to the selection multiplexors at ALU operand inputs. The value of `result3` is used to be forwarded when the dependency is identified with the result of the instructions issued in one clock cycle before the current one. The value of `result4` is used to be forwarded when the dependency is identified with the result of the instructions issued in two clock cycles before the current one.

Example 5.4 *Data dependency solved by using forwarding is exemplified starting from Example 5.4:*

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

Now the processor behaves incorrectly without the aditional `NOP`*s, as follows:*

```
t=0  pc=   x  RF=[x, x, x, x, x, x, x, x]
t=1  pc=1023  RF=[x, x, x, x, x, x, x, x]
t=5  pc=   0  RF=[x, x, x, x, x, x, x, x]
t=7  pc=   1  RF=[x, x, x, x, x, x, x, x]
t=9  pc=   2  RF=[x, x, x, x, x, x, x, x]
t=11 pc=   3  RF=[x, x, x, x, x, x, x, x]
t=13 pc=   4  RF=[x, x, x, x, x, x, x, x]
t=15 pc=   5  RF=[x, x, x, x, x, x, x, x]
t=17 pc=   6  RF=[1, x, x, x, x, x, x, x]
t=19 pc=   7  RF=[1, 2, x, x, x, x, x, x]
t=21 pc=   8  RF=[1, 2, 3, x, x, x, x, x]
t=23 pc=   8  RF=[1, 2, 3, 4, x, x, x, x]
t=25 pc=   8  RF=[1, 2, 3, 4, 5, x, x, x]
t=27 pc=   8  RF=[1, 2, 3, 4, 5, 9, x, x]
```

because the operands for the arithmetic operations are now forwarded using the two multiplexors to the ALU's inputs.

◇

5.2.2 Control Dependency

The pipelined processor always fetches the instruction immediately after any taken branch.

Example 5.5 *Let be the following sequence of instructions which contains a unconditioned (relative) jump:*

```
i:          XXX
i+1:        RJMP(12)
i+2: LB(2)  YYY
i+3:        UUU
i+4:        VVV
...         ...
i+j: LB(12) ZZZ
```

Its execution is supposed to be:

```
XXX
RJMP(12)
ZZZ
```

but in our processor it will be:

```
XXX
RJMP(12)
YYY
ZZZ
```

because the instruction `YYY` *are inserted into the pipe before the execution of the jump instruction completed in* `t=i+2` *when the jump address is available (see Table 5.4).*

Table 5.4: Hazard generated by the control dependency

Instr \ Time	t=i	t=i+1	t=i+2	t=i+3	t=i+4	t=i+5	t=i+6	t=i+7
IF	XXX0	RJMP0	YYY0	ZZZ0				
OF		XXX1	RJMP1	YYY1	ZZZ1			
EX			XXX2	----	YYY2	ZZZ2		
DM				XXX3	----	YYY3	ZZZ3	
WB					XXX4	----	YYY4	ZZZ4

◇

Example 5.6 *In the following example the effect of interrupt is illustrated together with the effect of the control dependency due to the unwanted execution of* `VAL(4,33)` *positioned after the* `RET30` *instruction.*

```
VAL(31,13)      ;
VAL(2,23)       ;
VAL(0,13)       ;
VAL(1,13)       ;
EI              ;
ADDV(0,0,1)     ;
VAL(1,1)        ;
VAL(2,222)      ;
NOP             ;
ADDV(0,0,4)     ;
```

```
            HALT            ;
            HALT            ;
            NOP             ;
// subroutine triggered by interrupt
            ADDV(30,30,-1)  ;
            NOP             ;
            NOP             ;
            NOP             ;
            RET(30)         ;
            VAL(4,33)       ;
```

The result of simulation is:

```
t=0  pc=   x  RF=[x, x, x, x, x, x, x, x]       intState=xxx inta=x
t=1  pc=1023  RF=[x, x, x, x, x, x, x, x]       intState=000 inta=0
t=5  pc=   0  RF=[x, x, x, x, x, x, x, x]       intState=000 inta=0
t=7  pc=   1  RF=[x, x, x, x, x, x, x, x]       intState=000 inta=0
t=9  pc=   2  RF=[x, x, x, x, x, x, x, x]       intState=000 inta=0
t=11 pc=   3  RF=[x, x, x, x, x, x, x, x]       intState=000 inta=0
t=13 pc=   4  RF=[x, x, x, x, x, x, x, 13]      intState=000 inta=0
t=15 pc=   5  RF=[x, x, 23, x, x, x, x, 13]     intState=000 inta=0
t=17 pc=   6  RF=[13, x, 23, x, x, x, x, 13]    intState=001 inta=0
t=19 pc=   7  RF=[13, 13, 23, x, x, x, x, 13]   intState=010 inta=1
t=21 pc=  13  RF=[13, 13, 23, x, x, x, 6, 13]   intState=011 inta=0
t=23 pc=  13  RF=[13, 13, 23, x, x, x, 6, 13]   intState=100 inta=0
t=25 pc=  13  RF=[13, 13, 23, x, x, x, 6, 13]   intState=000 inta=0
t=27 pc=  14  RF=[13, 13, 23, x, x, x, 6, 13]   intState=000 inta=0
t=29 pc=  15  RF=[13, 13, 23, x, x, x, 5, 13]   intState=000 inta=0
t=31 pc=  16  RF=[13, 13, 23, x, x, x, 5, 13]   intState=000 inta=0
t=33 pc=  17  RF=[13, 13, 23, x, x, x, 5, 13]   intState=000 inta=0
t=35 pc=  18  RF=[13, 13, 23, x, x, x, 5, 13]   intState=000 inta=0
t=37 pc=   5  RF=[13, 13, 23, x, x, x, 5, 13]   intState=000 inta=0
t=39 pc=   6  RF=[13, 13, 23, x, x, x, 5, 13]   intState=000 inta=0
t=41 pc=   7  RF=[13, 13, 23, x, x, x, 5, 13]   intState=000 inta=0
t=43 pc=   8  RF=[13, 13, 23, x, 33, x, 5, 13]  intState=000 inta=0
t=45 pc=   9  RF=[14, 13, 23, x, 33, x, 5, 13]  intState=000 inta=0
t=47 pc=  10  RF=[14, 1, 23, x, 33, x, 5, 13]   intState=000 inta=0
t=49 pc=  11  RF=[14, 1, 222, x, 33, x, 5, 13]  intState=000 inta=0
t=51 pc=  11  RF=[14, 1, 222, x, 33, x, 5, 13]  intState=000 inta=0
t=53 pc=  11  RF=[18, 1, 222, x, 33, x, 5, 13]  intState=000 inta=0
t=55 pc=  11  RF=[18, 1, 222, x, 33, x, 5, 13]  intState=000 inta=0
t=57 pc=  11  RF=[18, 1, 222, x, 33, x, 5, 13]  intState=000 inta=0
```

The execution of `VAL(4,33)` *is obvious at* `t=43`*. The program syntax does not assume this.* ◇

There are few solutions for solving the previously emphasized hazard.

Stalling

By introducing a stall using a `NOP` instruction the sequence of instruction becomes:

```
i:          XXX
i+1:        RJMP(12)
i+2:        NOP
i+3: LB(2)  YYY
...         ...
i+j: LB(12) ZZZ
```

the execution will be:

```
XXX
RJMP(12)
NOP
ZZZ
```

This solution results in a branch penalty (the number of stalls introduced during the branch operations in the pipelined processor) of 1 cycle (the NOP introduced after `RJMP(12)`).

Reordering

Sometimes the branch penalty can be reduced by reordering.

Because, in the previous example, the jump is unconditional, which means the execution of the instruction `XXX` does not have implications on the jump instruction, the sequence of instruction can be reordered, as follows:

```
i:          RJMP(12)
i+1:        XXX
i+2: LB(2)  YYY
...         ...
i+j: LB(12) ZZZ
```

thus, executing the instruction `XXX` after jump no penalty is introduced.

Example 5.7 *Reordering is exemplified by the following code:*

```
        NOP         ;
        VAL(0,-3)   ;
        NOP         ;
        NOP         ;
LB(1);  NOP         ;
        NOP         ;
        BRNZ(0,1)   ;
        ADDV(0,0,1) ;
        HALT        ;
        HALT        ;
```

where the increment of the register 0 is specified out of loop, but is executed associated with each branch. The result of simulation is:

```
t=0  pc=   x  RF=[x, x, x, x, x, x, x, x]
t=1  pc=1023  RF=[x, x, x, x, x, x, x, x]
t=5  pc=   0  RF=[x, x, x, x, x, x, x, x]
t=7  pc=   1  RF=[x, x, x, x, x, x, x, x]
t=9  pc=   2  RF=[x, x, x, x, x, x, x, x]
t=11 pc=   3  RF=[x, x, x, x, x, x, x, x]
t=13 pc=   4  RF=[x, x, x, x, x, x, x, x]
t=15 pc=   5  RF=[4294967293, x, x, x, x, x, x, x]
t=17 pc=   6  RF=[4294967293, x, x, x, x, x, x, x]
t=19 pc=   7  RF=[4294967293, x, x, x, x, x, x, x]
t=21 pc=   4  RF=[4294967293, x, x, x, x, x, x, x]
t=23 pc=   5  RF=[4294967293, x, x, x, x, x, x, x]
t=25 pc=   6  RF=[4294967293, x, x, x, x, x, x, x]
t=27 pc=   7  RF=[4294967294, x, x, x, x, x, x, x]
t=29 pc=   4  RF=[4294967294, x, x, x, x, x, x, x]
t=31 pc=   5  RF=[4294967294, x, x, x, x, x, x, x]
t=33 pc=   6  RF=[4294967294, x, x, x, x, x, x, x]
t=35 pc=   7  RF=[4294967295, x, x, x, x, x, x, x]
t=37 pc=   4  RF=[4294967295, x, x, x, x, x, x, x]
t=39 pc=   5  RF=[4294967295, x, x, x, x, x, x, x]
t=41 pc=   6  RF=[4294967295, x, x, x, x, x, x, x]
t=43 pc=   7  RF=[0, x, x, x, x, x, x, x]
t=45 pc=   8  RF=[0, x, x, x, x, x, x, x]
t=47 pc=   9  RF=[0, x, x, x, x, x, x, x]
t=49 pc=   9  RF=[0, x, x, x, x, x, x, x]
t=51 pc=   9  RF=[1, x, x, x, x, x, x, x]
t=53 pc=   9  RF=[1, x, x, x, x, x, x, x]
t=55 pc=   9  RF=[1, x, x, x, x, x, x, x]
```

◇

Static Branch Prediction

Let us see now how the conditioned jumps, the branches, are managed to be performed efficiently. There are two cases: backward branches or forward branches. Static prediction presumes that backward branches will be taken and that forward branches will not.

In this case, most of the predictions will be correct. Indeed, if we have a jump back to execute a loop that repeats n times, then once in $n+1$ situations the prediction will have to be corrected.

Example 5.8 *A backward branch is one that has a target address that is lower than its own address. For example, it refers to the following type of loop:*

```
i:          VAL(1,23)
i+1:        VAL(2,1)
i+2:  LB(3) SUB(1,1,2)
i+3:        NZJMP(1,3)
i+4:        NOP
i+5:        NOP
i+6:        ...
```

This technique can help with prediction accuracy of loops, which are usually backward-pointing branches, and are taken more often than not taken.

◇

Example 5.9 *A forward branch is one that has a target address that is higher than its own address. For example, it refers to the following type of loop:*

```
i:          VAL(1,23)
i+1:        VAL(2,1)
i+2:        SUB(1,1,2)
i+3:        ZJMP(1,2)
i+4:        XXX
            ...
i+j: LB(2)  YYY
```

◇

In static prediction, all decisions are made at compile time, before the execution of the program

Note: In order to reduce the branch penalty, in the case of unconditional jumps, certain changes can be made in the hardware. For example, the calculation of unconditional jump addresses can be moved to the OP FETCH (OF) level. The execution latency of unconditional jumps is thus reduced by one unit.

Dynamic Branch Prediction

Dynamic branch prediction predicts branches based on dynamic information collected at run-time. It requires additional hardware. As examples, I have chosen two of the simplest prediction mechanisms.

Last-time, one-bit predictor is the simplest solution which uses a two state automaton whose behavior is described in Figure 5.4. We will code the state `prediction not taken` with 0, and the state `prediction taken` with 1. The automaton is a *saturated* 1-bit counter. If is incremented from 0 goes to 1, if is decremented from 1 goes to 0. But if it is incremented from 1, stays in 1, while decremented from 0 stays in 0. This 2-state automaton says whether the branch was recently taken or not. Based on this, the processor fetches the next instruction from the target address or sequential address. If the prediction is wrong, flushes the pipeline and also flips prediction. So, every time a wrong prediction is made, the prediction bit is flipped.

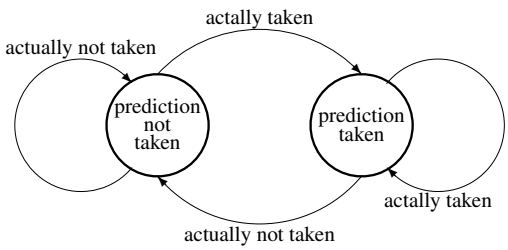


Figure 5.4: The saturated one-bit up/down counter as last-time branch predictor.

This mechanism always mispredicts the last iteration and the first iteration of a loop branch, thus the accuracy for a loop with N iterations is $(100 \times (N-2)/N)\%$. For large values of N the accuracy of the prediction increases, while for small values this mechanism is not very efficient.

Two-Bit Counter Based Predictor is a two-state finite automaton. Its behavior is described in Figure 5.5. The advantage of the two-bit predictor over a one-bit predictor is that a conditional jump has to deviate twice from what it has done most in the past before the prediction changes. For example, a loop-closing conditional jump is mispredicted once rather than twice.

Each state of the automaton is interpreted as follows:

- 00: strongly not taken
- 01: weakly not taken
- 10: weakly taken
- 11: strongly taken

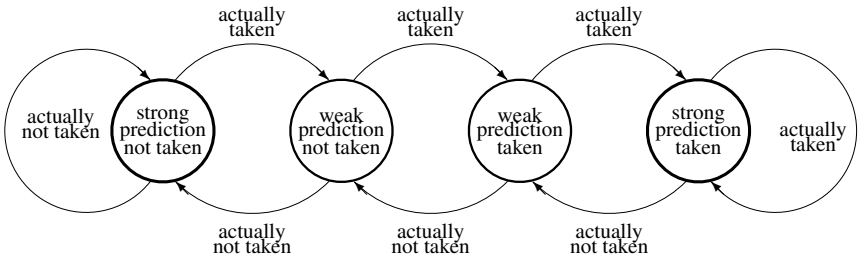


Figure 5.5: Two-bit saturated counter based predictor.

The automaton is a *saturated* 2-bit counter. It increments for `actually taken` and decrements for `actually not taken`. This predictor changes prediction only on two successive mispredictions.

5.3 Superscalar Processor

If the processor is designed with several execution units (for example: one integer ALU, two floating-point units, two load/store units), the potential parallelism thus obtained should be activated at the highest possible level. This means that in each clock cycle a maximum number of resources should be active, if possible all of them.

In this case, the processor, called *superscalar*, can execute instructions in an order imposed by the availability of data and execution units, rather than by their initial sequence in the program. For example, the *PowerPC 970* processor fetches and decodes up to eight instructions, dispatch up to five to reserve stations (register files), issue up to eight to the execution units and retire up to five per cycle.

5.3.1 Register renaming

The sequence in which the instructions are executed must be strictly respected when there are data dependencies. But when certain independent sequences can be identified, strictly sequential execution is

no longer mandatory, especially in the case of a superscalar processor that allows parallelism at the level of instructions. We can apply in such cases the *register renaming* technique.

Example 5.10 *Consider this sequence of code running on an out-of-order processor:*

```
READ(0,4)   // read in rf[0] from the address in rf[4]
ADD(0,0,12) // rf[0] <= rf[0] + rf[12]
STORE(0,7)  // dataMemory[rf[7]] <= rf[0]
READ(0,5)
ADD(0,0,6)
STORE(0,21)
```

The instructions in the last three lines are **independent** *of the first three lines. The processor cannot execute* `STORE(0,21)` *until* `STORE(0,7)` *is done. We can eliminate this restriction involving in our code a supplementary register, as follows:*

```
READ(0,4)   // read in rf[0] from the address in rf[4]
ADD(0,0,12) // rf[0] <= rf[0] + rf[12]
STORE(0,7)  // dataMemory[rf[7]] <= rf[0]
READ(2,5)
ADD(2,2,6)
STORE(2,21)
```

The first three instructions involves `rf[0]`*, while the last three instructions work on the register* `rf[2]`*, allowing the last three instructions to be executed in parallel with the first three.*

◇

5.3.2 Out-of-Order Execution

Out-of-order execution is a mechanism used in high-performance processing units to make use efficiently of instruction cycles. Using this mechanism, a processor executes instructions in an order governed by the availability of input data and execution units, rather than by their original order in a program. Thus the processor avoids being idle while waiting for the preceding instruction to complete and can, in the meantime, proceed to execution of the next instructions that are able to run immediately and independently. The mechanism is efficient for a processor with multiple execution units **with speed difference between instructions**. It is the case of superscalar processor with ISA performing integer and floating-point arithmetic operations. There are also speed difference between arithmetic and logic instructions and memory access instructions.

While in a pipelined scalar processor an instruction is executed in the following steps:

1. Instruction fetch.
2. Operands fetch, if input operands are available in processor's register file, else the processor stalls until they are available.
3. The instruction is executed.
4. The results is write back to the register file.

in a superscalar processor the out-of-order execution is requested. It consists of the following steps:

1. Instruction fetch.
2. Instruction dispatch to an instruction queue (also called instruction buffer or reservation stations).
3. The instruction waits in the queue until its input operands are available. The instruction can leave the queue before older instructions.
4. The instruction is issued to the appropriate idle functional unit.
5. The results are queued.
6. Only after all older instructions have their results written back to the register file, then this result is written back to the register file.

The benefit of out-of-order execution processing grows as the instruction pipeline deepens and the speed difference between main memory or cache memory and the processor widens. On modern machines, the processor runs few times faster than the cache memory and many times faster then system memory, so during the time an in-order processor waits for data, a large number of instructions could be executed.

Example 5.11 *In the following sequence of instructions (see [Patterson '05], p. 185):*

```
DIV(0,2,4)
ADD(6,0,8)
STR(6,1)    // store
SUB(8,9,7)
MUL(6,9,8)
```

here are the following dependencies:

1. *between ADD and SUB if SUB finishes before ADD starts (a WAR hazard) is an antidependence*
2. *between ADD and MUL if ADD finishes later than MUL (WAW hazard) is an output dependence*
3. *between DIV and ADD a true data dependency (a RAW hazard)*
4. *between SUB and MUL a true data dependency (a RAW hazard)*
5. *between ADD and STORE a true data dependency (a RAW hazard)*

First, let us apply register renaming technique involving two additional registers, `rf(10)` *and* `rf(11)`*:*

```
DIV(0, 2, 4)
ADD(10,0, 8)
STR(10,1)
SUB(11,9, 7)
MUL(6, 9, 11)
```

Any WAR and WAW dependencies are now removed statically by the compiler. For RAW dependency the forwarding mechanism works but only in a processor with one integer ALU. What can be done for a superscalar processor with floating point arithmetic?

◇

The hardware added for out-of-order execution is big sized and complex. With the increase in the number of pipeline levels, the costs (area plus complexity in use) increase. Multiprocessing and multithreading will allow more efficient alternative solutions.

Floating-point representation of real numbers

To expand the range in which the numbers are represented, the following form is used:

$$N = sgn(1 + frac) \times 2^{exp-127}$$

where:

$$sgn \in \{-,+\} = \{0,1\}$$
$$frac \in [0,1)$$
$$exp \in [0,255]$$

The binary representation is:

$$\langle sgn|exp|frac\rangle$$

For example the 32-bit version:

$$0\_10000001\_10000000000000000000000$$

represents the number: $+(1+0.5) \times 2^{129-127} = 6$

More at: `https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/`

For the purpose we are pursuing, it is enough to understand that the operation with represented floating-point numbers involves a pipeline or a sequence of elementary operations of one clock cycle. The number of cycles will be different depending on the operation, minimum for addition and maximum for division.

Tomasulo's Algorithm

The execution unit of a scalar processor has, in addition to the ALU for integers that we discussed, several units that perform operations with floating-point numbers. For out-of-order execution, it is not enough to have a register file. Next to each execution unit, for whole or real, complex storage units called Resevation Stations (RS) will be added. Each recording in a RS has the following fields [Patterson '19]:

- Op—The operation to perform on source operands S1 and S2.
- Qj, Qk—The reservation stations that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary.
- Vj, Vk—The value of the source operands. Note that only one of the V fields or the Q field is valid for each operand. For loads, the Vk field is used to hold the offset field.
- A—Used to hold information for the memory address calculation for a load or store. Initially, the immediate field of the instruction is stored here; after the address calculation, the effective address is stored here.
- Busy—Indicates that this reservation station and its accompanying functional unit are occupied.

while the register file has a field, Qi:

- Qi—The number of the reservation station that contains the operation whose result should be stored into this register. If the value of Qi is blank (or 0), no currently active instruction is computing a result destined for this register, meaning that the value is simply the register contents.

5.4 Problems

5.4.1

5.4.2

Section 6

Computers: Four-Loop, Fourth-Order Systems (4-OS)

Contents

The processor is the core of a computing system that contains also a memory subsystem and an input-output subsystem. The physical resources involved are tightly interleaved in various form of actual organization. The memory associated with the processor contains programs and data, while the input-output system has two main functions: expanding the memory capacity and ensuring the interaction of the system with the outside world.

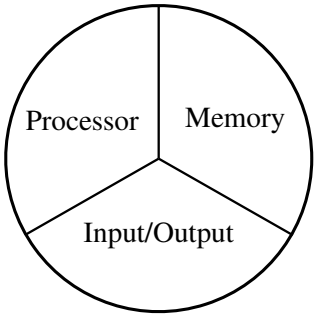


Figure 6.1: The three-partite functionality of a computing system.

The way in which the tripartite functionality is distributed in a given system shows significant variations from o ne implementation to another. In the following, we will limit ourselves to a simple version, but not simpler than one that allows us to introduce the main concepts and problems.

6.1 Memory

6.1.1 Memory Gap

The evolution of the performance of the three components of a computer system occurred at a very different rate due to strictly technological aspects. The most disturbing difference, with major structural consequences, occurred in the case of processor and memory speed performances. The frequency of the processor clock increased much faster than the memory access time (see Figure 6.2. Processor clock frequency increased with 60% per year, while the access time for memories with only 10% per year.

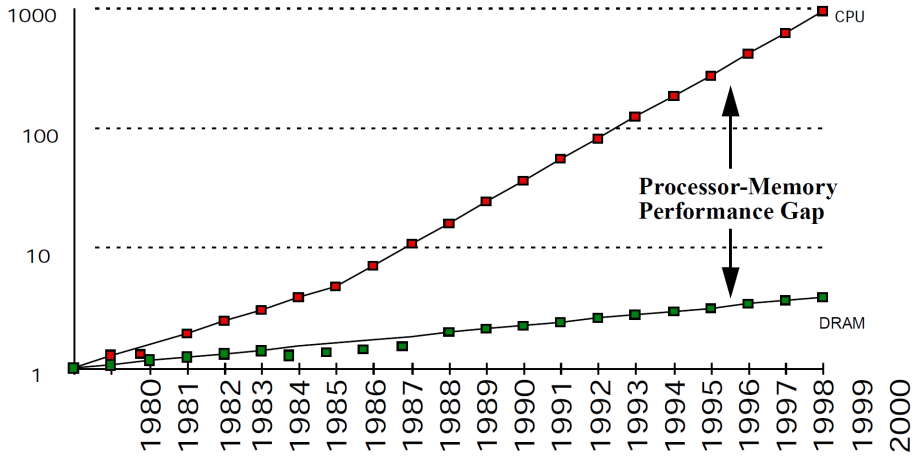


Figure 6.2: Processor-Memory Performance Gap [Patterson '97].

Due to this difference, what we now call a memory gap occurred. The fast processor cannot wait for the slow memory to deliver or receive its data. And the memory gap imposed the memory hierarchy of the computing system based on the locality principle.

Locality principle says that if a certain location is accessed in the data or program memory, then the next accesses will be made with a high probability in a reasonably small vicinity.

Applying the locality principle allowed the definition of the strategies that govern the conception of a hierarchy of effective memories.

6.1.2 Memory Hierarchy

The hierarchy of memory is based on the fact that when we access a location that is in a memory that is too slow in relation to the speed of processing in the processor, a block or a page of bytes that contains the requested bytes will be transferred to a faster intermediate memory. This transfer is done with the hope that the following accesses will be made in the intermediate memory if the updated page or block in it was large enough.

Figure 6.3 shows the current way in which the memory hierarchy is implemented. At the top of the hierarchy there are registers from register files whose content is accessible at the level of the clock signal cycle. Next comes system memory in the form of the closest level as cache memory. The name comes from the French *caché* which means hidden. Indeed, this level of memory/memories is transparent to the architecture of the computing system in most cases.

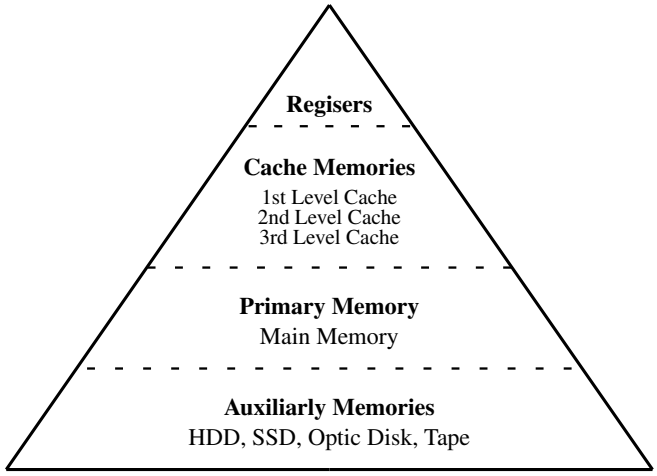


Figure 6.3: Memory hierarchy from fast and small register files to slow and large auxiliary storage memories such as HDD (Hard Disk Drive), SSD (Solid-State Drive), optical disk, tape.

Registers are managed by compilers, their size is $< 4KB$ (in most of cases $\sim 128B$, i.e., 32 32-bit words), the access time is $< 0.5ns$.

Cache memories are managed by hardware, their size is $< 16MB$, the maximum access time is $\sim 0.5ns$.

The main memory is managed by the operating system, its size is in the range of $1 \div 16GB$, the access time is $\sim 100ns$.

The disk memory is managed by the operating system or human operator, its size is $> 128GB$, the access time is $\sim 5ms$.

6.1.3 Virtual Memory Mechanism

How relate two successive levels in the hierarchy? Let us consider two levels in the memory hierarchy, $Level_i$ containing m pages and $Level_{i+1}$ containing n pages, with $n >> m$. Each page in $Level_i$, called *physical level* is a copy of a page from $Level_{i+1}$ called *virtual level*. In Figure 6.4a, pages 0, 2, and $m-2$ from the level i were brought from pages 3, 4, and 0 of level $i+1$. Any change in the physical level must actualized in the associated virtual level. The "user" accesses the virtual level while the virtual mechanism accesses the physical level. It is the job of the virtual memory mechanism to perform transparently the access and to keep coherent the content of the virtual level with the physical level.

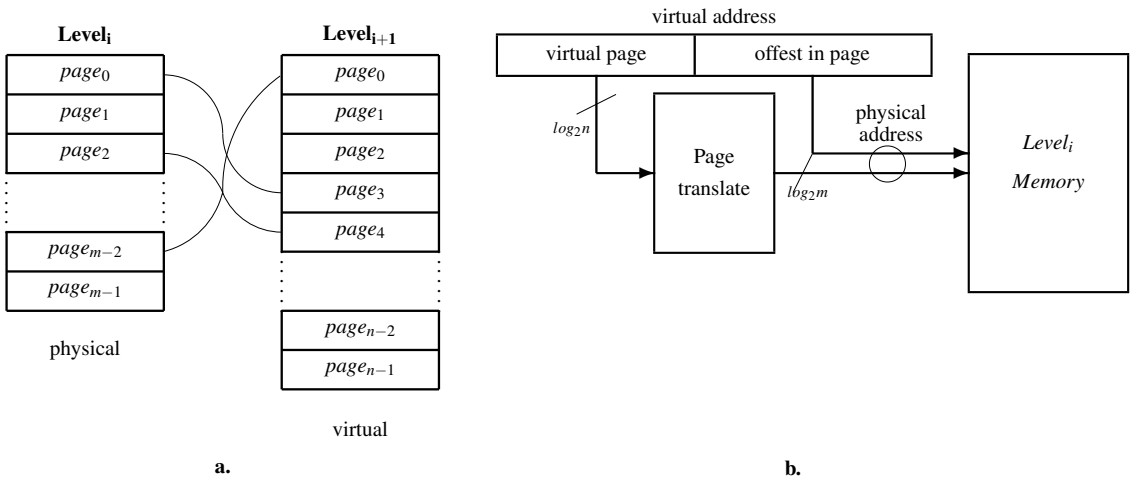


Figure 6.4: Virtual memory mechanism. **a.** Page correspondence. **b.** Translation mechanism.

The addressing mechanism is represented in Figure 6.4b, where the access to the physical memory is done mediated by a translation mechanism performed by the *Page translate* block. The "user" issues a virtual address having two parts:

- virtual page address
- offset in the page

When a new page is loaded into physical memory, the content of this translator is updated according to the correspondence shown in the figure 6.4a.

Page translation mechanism is implemented in various form.

The simplest one is a RAM memory containing n $log_2 m$-bit words. The RAM memory is in this case almost empty, because only m locations from n are used, because usually $n >> m$.

A more efficient solution is to use an associative memory to make the translation.

6.1.4 Associative Memory-Based Page Translator

Content-Addressable Memory

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

Q1: *what is 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.

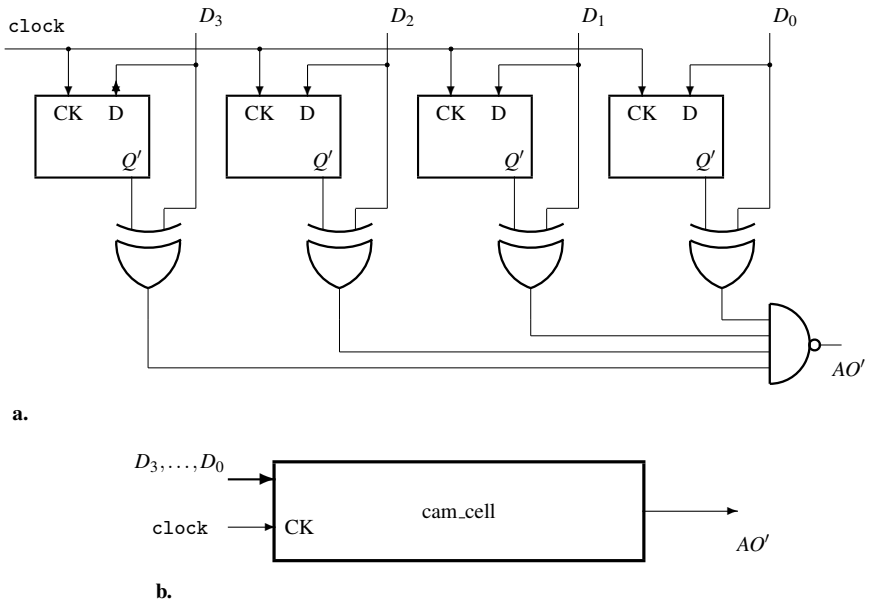


Figure 6.5: **Content Addressable Cell. a.** The structure: data latches whose content is compared against the input data using 4 XORs and one NAND. Write is performed applying the clock with stable data input. **b.** The logic symbol.

In Figure 6.5 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 6.6a 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.

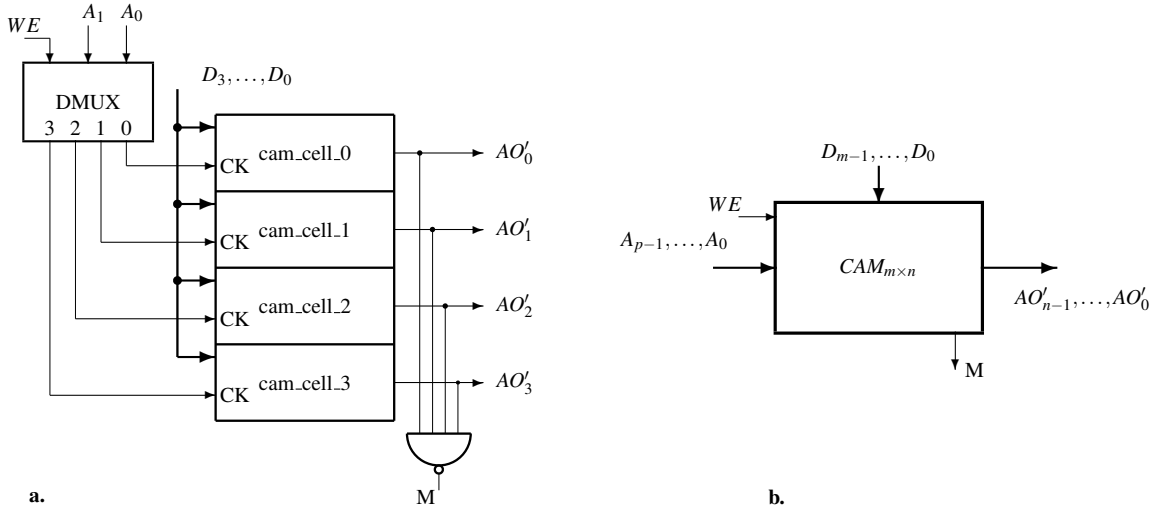


Figure 6.6: **The Content Addressable Memory (*CAM*). a.** A 4-word CAM is built using 4 content addressable cells, a demultiplexor to distribute the write enable (WE) signal, and a $NAND_4$ to generate the match signal (M). **b.** The logic symbol.

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 $D_{m-1},\ldots,D_0$ 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 6.6b 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.

Associative Memory

A partially used RAM can be an associative memory, but a very inefficient one. Indeed, let be a RAM addressed by $A_{n-1}\dots A_0$ containing 2-field words $\{V,\ D_{m-1}\dots D_0\}$. The *objects* are codded using the address, the *values* of the unique property P are codded by the data field $D_{m-1}\dots D_0$. 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 6.1 *Let be the 1Mword RAM addressed by $A_{19}\dots A_0$ containing 2-field 17-bit words $\{V,\ D_{15}\dots D_0\}$. 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:

- *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

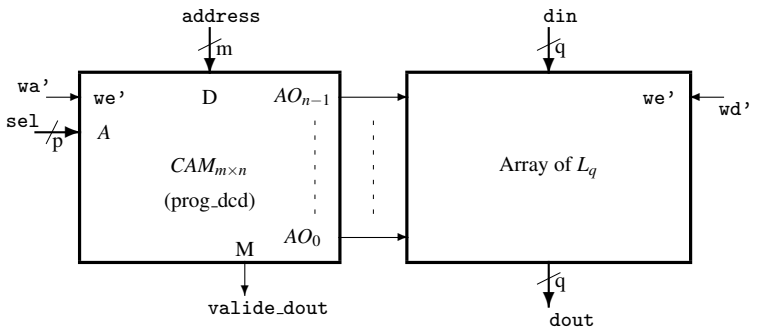


Figure 6.7: **An associative memory (*AM*).** The structure of an AM can be seen as a RAM with a programmable decoder implemented with a CAM. The decoder is programmed loading CAM with the considered addresses.

select the accessed location of a RAM containing the value of the property P. In Figure 6.7 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.

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.

Translation Lookaside Buffer (TLB)

The associative memory will allow us to design a translation lookaside buffer (TLB) with a size given by the number of pages in the physical memory, unlike the one based on a RAM type memory which has a size given by the much larger number of pages in the virtual memory.

With a minimal number of differences, the described virtualization mechanism is applied in managing the relationship between all the hierarchical levels in the memory of the computer system.

In what follows, we will consider the simple example in which the hierarchy contains a single cache level, the main memory and the hard disk. We will compare the main parameters that characterize the cache-main memory relationship and the main memory-hard disk relationship.

- the typical amount of data moved when physical memory is actualized from the virtual memory
  - block size for cache: 128B
  - page size main memory: 4KB
- the typical access time when the addressing is hit
  - 1 clock cycle (very rarely 2 or 3) for cache
  - $\sim$ 100 clock cycles because of board technology and silicon technology for memories (the first is reduced by multi-chip packaging) for main memory
- the typical miss penalty:
  - 100 clock cycles for cache memory
  - 10^6 clock cycles for main memory
- the typical miss rate:
  - 1% for cache memory
  - 10^{-4}% for main memory

6.2 System Organization

How looks the actual structure of a computing system? In Figure 6.8 is exemplified a system where are emphasized two buses: the internal bus and the input-output bus. They are interconnected by two units: Bus Bridge and DMA (Direct Memory Access).

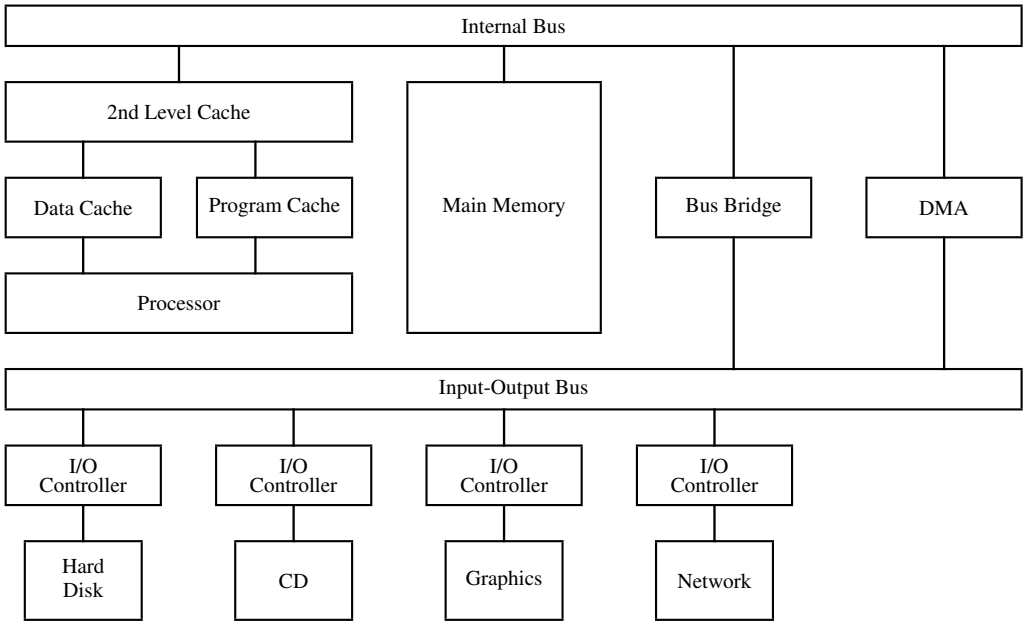


Figure 6.8: The organization of a computing system.

On Internal Bus are connected fast devices while on the Input-Output Bus are connected devices involving a lot of data transfer.

The memory hierarchy is distributed in the version represented in Figure 6.8 as follows:

1. Register File in the processor, having a capacity of 32W ÷ 1024W, accessed for reading or writing in a single clock cycle

2. the two 1st level cache memories, each having a capacity of 128KB ÷ 1MB, accessed for reading or writing in 1 ÷ 3 clock cycles:

   - Program Cache
   - Data Cache

   so that at this level the system is structured according to the Harvard abstract model

3. 2nd Level Cache memory, having a capacity of 1MB ÷ 8MB, so that at this level the system is structured according to the von Neumann abstract model

4. Hard Disk connected through:

   - Bus Bridge
   - DMA

6.3 I/O

The input-output system includes certain specific mechanisms that we will briefly describe first. Then we will briefly describe the most important input-output devices.

6.3.1 Bus

A ***Bus*** is a communication system that transfers data between components inside a computer, or between computers. There are two types of buses: serial (those that transfer data bib by bit) and parallel (those that transfer data word by word of n bits). The main problem that arises is that of the synchronization of the transfer. We will discuss it under its essential aspects for the synchronous transfer of n-bit words.

In Figure 6.9, the waveforms for data transfer on the Internal Bus (see Figure 6.8) between Main Memory and 2nd Level Cache are presented. Three important moments are marked in the mentioned waveforms:

t_1 : the active edge of clock takes a read command (Read = 0) from the address Read Address. Both Read and Read Address must be stable at least minimum ***set-up time*** before the active edge of clock (the positive one) and ***hold time*** after the positive edge of clock

t_2 : the active edge of clock can take the data from the output of Main Memory only if the signal Wait is not active (Wait = 0) thus completing the read cycle. The signal Wait is necessary when the Main Memory is not a fast enough memory, or the clock frequency is too high.

t_3 : the active edge of clock takes Valid Input Data to write it at Write Address because Write signal is active (Write' = 0).

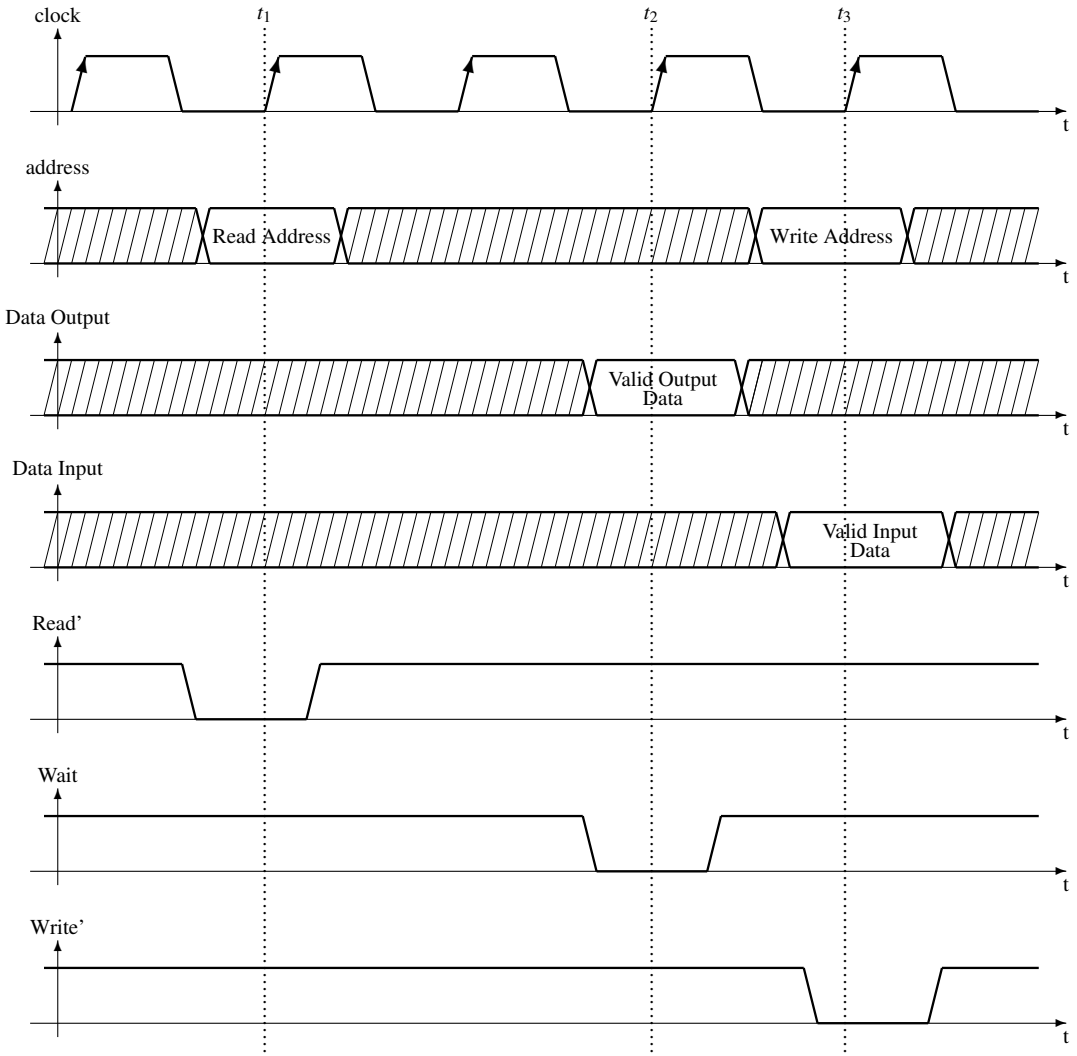


Figure 6.9: The waveforms for a read cycle followed by a write cycle.

Q.6.1: In the waveforms in Figure 6.9 the distance between t_1 and t_2 is only one clock cycle. In reality, this distance can be tens or even hundreds of clock cycles. What is the explanation of this fact?

Q.6.2: Why do you think that a single clock cycle was enough for writing?

The Wait signal is deactivated (Wait = 0) at t_2 introducing a latency of 2 clock cycles in the example illustrated by Figure 6.9. Usually, this latency is very high, reaching hundreds of cycles.

6.3.2 DMA

Direct memory access (DMA) (see Figure 6.8) is an important feature of computer systems. It allows to access the main memory system independently of the central processor.

When the processor is running an input/output program, it is occupied for the entire duration of the read or write operation. Thus the processor is unavailable to perform other work. With DMA, the processor initiates the transfer only, then it does other operations while the transfer is performed. When the transfer is completed the processor receives an interrupt from the DMA (about interrupt see 4.5.1 in this lecture notes).

An example of using the DMA unit is for transferring a page of 4KB from Hard Dist to Main Memory.

6.3.3 FIFO

First-In-First-Out (FIFO) memory is a special memory device represented in Figure 6.10. It has the following connections (see Figure 6.10a):

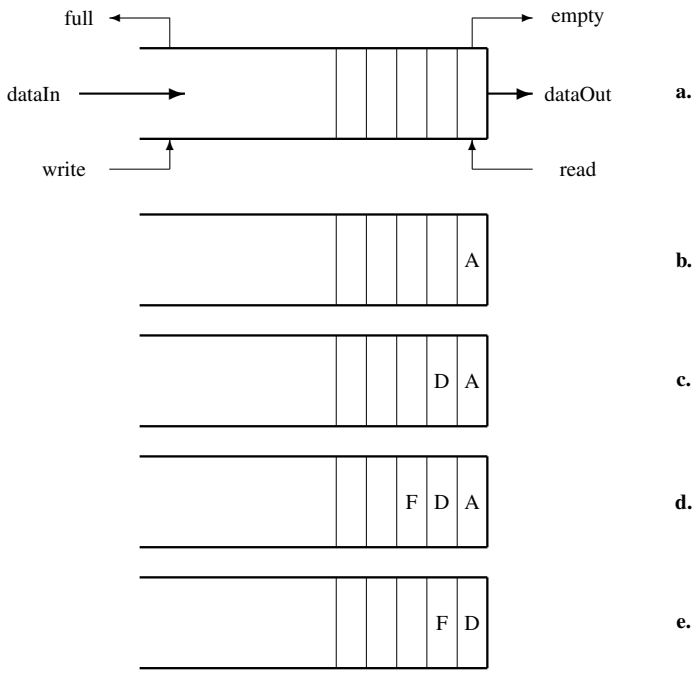


Figure 6.10: **a.** Block schematic of FIFO. **b.** `write A` operation. **c.** `write D`. **d.** `write F`. **e.** `read`.

dataIn : the data entry of n bits to which the word that will be recorded in the rightmost unoccupied location is applied.

dataOut : the n bit data output from which the oldest unextracted record is extracted

full : it signals the fact that the memory is full and will not accept a new write before executing at least one read

write : if activated when `full = 0`, then `dataIn` is queued

empty : it signals the fact that the memory is empty and will not accept a new read before executing at least one write

read : if activated when `empty = 0`, then `dataOut` is extracted from the queue

In Figures from 6.10b to 6.10e is exemplified the way FIFO works, as follows:

`write(A)` : in the empty FIFO A is stored in the rightmost position (see Figure 6.10b)

`write(D)` : B is stored in the rightmost free position, immediately after A (see Figure 6.10c)

`write(F)` : F is stored in the rightmost free position, immediately after B (see Figure 6.10cd)

`read` : A is extracted from FIFO and the queue ”advances” one step (see Figure 6.10e)

There are two main types of FIFO:

- synchronous FIFO: the sending system and the receiving system run with the same clock signal (for those who are interested can consult Appendix **??**)
- asinchronous FIFO: the sending system and the receiving system run with different clock signals

Asynchronous FIFO are used to solve the problem of crossing data between systems running in different clock domain. The module BUS Bridge (see Figure 6.8) contains almost sure an asynchronous FIFO.

6.3.4 I/O Devices

There are two types of I/O devices:

- Storage devices, mainly represented by hard disk, flash memory, optical disk, tape.
- Functional devices, mainly represented by display system, network interface, ...

We will briefly present the storage devices that extend the memory hierarchy outside the central computing unit.

Hard Disk

The most common auxiliary memory is the hard disk drive. Its internal structure is represented in Figure 6.11.

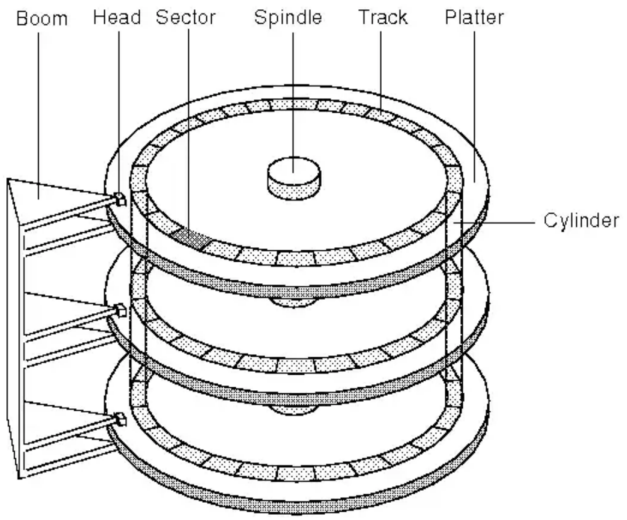


Figure 6.11: Hard Disk Drive Structure [Source: Google-Images]

Main parameters:

- storage capacity: $10 \div 30$ TB
- average access time: $2.5 \div 10$ ms
- MTBF: $\sim$285 years (MTBF stands for mean time between failures; it is the predicted elapsed time between inherent failures of a mechanical or electronic system, during normal system operation.)
- bytes per sector: $512 \div 4096$
- shock tolerance
  - operating: $10 \div 100$ g (g stands for gravitational acceleration: $9.8 m/s^2$.)
  - nonoperating: $100 \div 100$ g

Optical Disks

Optical disks are read-only mediums. There are two versions:

- CD: optical compact disk (0.65 GB)
- DVD: digital video disk (4.7 GB); sometimes written on both sides to double capacity

Flash Memory

Flash memory is a type of EEPROM (electronically erasable programmable readonly memory), which is normally read-only but can be erased. Solid-state devices (SSDs) based on flash memory technology do not have moving parts.

The capacity per flash memory chip increased by about 50%–60% per year.

Typical SSDs will have a the time taken for a disk drive to locate the area on the disk where the data to be read is stored between 0.08 and 0.16 ms.

Time to **read** 2 KB from a flash memory takes about 75 μS, while DDR SDRAM takes less than 500 ns. Then flash memory is about 150 times slower. But compared to HDD is ~ 400 times faster. We conclude: the flash memory can not replace DRAM for main memory, but is a good candidate to replace HDD.

Time to **write** for flash memory is about 1500 times slower then SDRAM, and about 8–15 times faster than HDD.

Tapes

Magnetic-tape data storage is a system for storing information on magnetic tape using digital recording.

Typically 9-track tape are used to store bytes with CRC. Magnetic tape packaged in cartridges and cassettes. The device that performs the writing or reading of data is called a tape drive.

Tape data storages are now used mainly for system backup, data archive or data exchange.

Uncompressed/Native capacity: $\sim$20TB.

Compressed capacity: $\sim$50TB.

Section 7

Open Problems

Contents

We are still faced, after three quarters of a century, with problems for which solutions are still waiting. Among the most important are parallelism, technological limits, and theoretical limits.

7.1 Parallelism

Instruction Level parallelism is not the genuine parallelism we expect from multi-cell structures.

While the hardware's size accommodated on a single die of silicon increases exponentially, according to Moore's Law [Moore '65, Moore '75], its use for increasing the complexity and the intensity of computation grows much slowly. We are able to build big machines but we fail in using all their computational power efficiently. For example:

- Tensor Processing Unit, is reported for $> 90\%$ of the applications with the use of 3% to 13.4% from its peak performance [Jouppi '17] [Hennessy '19]. Only for one application (involving exclusively convolutional layers in a deep neural network) used in less than 5% of cases, 93% from its peak performance is used
- the real time image recognition, where Nvidia's Titan X GPU, uses maximum 63 GFLOPs/sec from its peak performance of 6 TFLOPs/sec [Redmon '16]
- Intel's Xeon Phi accelerator with 57 cores, having peak performance at 2 TFLOPs/sec, involves only 35.2 GFLOPs/sec for a similar task [Raina '16].

If we can't effectively use many-core structures, why are we still struggling with parallelism? Because we have no other solution to make use of the potential offered by increasing the number of transistors on a chip. In Figure 7.1, the only parameter that increases is the number of transistors.

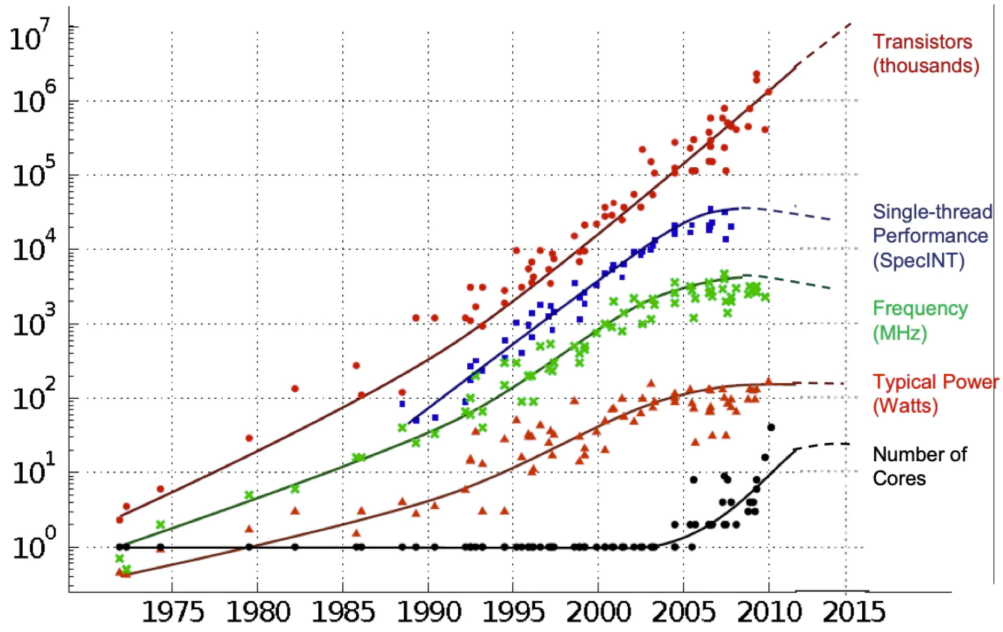


Figure 7.1: Why Parallelism? Because only the transistors number goes up [Moore '11].

Is it hard to explain why we use such a small percentage of the peak computing power of these many-core programmable accelerators. We assume that it highlights an architectural and organizational inadequacy such as:

1. a bad correlation between *exercising the control* and *implementing functions* is responsible for our inability to put at work the exponentially increasing hardware power. Indeed, the *complexity* of control and the *intensity* of the functional aspects of computation challenge the current solutions we have for one-chip many-core computation. Tens of billions of transistors on silicon dies, which is approaching an area of 10 cm^2, provide huge peak functional performance, performance which is not accompanied by an adequate capacity to put this huge computational capacity to work by an appropriate control mechanism. Too much emphasis on deploying huge arithmetic resources, and too little concern for the way the computational resources are put on work.

2. the same attention we must pay to the way these resources are interconnected and connected to the memory resources in order to maximize their use. The time and energy used to fetch data and to interconnect the computational resources are too many times much higher compared to those involved in the actual computation.

3. on the currently available many-core accelerators the organization of the computational resources is related rather with geometrical criteria (circle, mesh, hypercube, ...) than with theoretically imposed computational reasons related to the way the *sequence* of data is structured.

Many people agree on an imminent change. Some see this change as a gradual one, others as a radical one. Almost all of them are in the middle.

7.1.1 *Ad Hoc* Parallelism

After the year 2000, the clock speed race started to slowdown [Asanovic '06, Asanovic '09]. Instead of a clock with increasing frequency, the number of cores per chip started to grow with the hope that the programmer will find a way to take out more performance form the silicon maintaining the clock frequency around of few GHz. The situation is very clearly expressed by David Patterson:

> *... the semiconductor industry threw the equivalent of a Hail Mary pass when it switched from making microprocessors run faster to putting more of them on a chip – doing so without any clear notion of how such devices would in general be programmed. The hope is that someone will be able to figure out how to do that, but at the moment, the ball is still in the air.* [Patterson '10]

Computing emerged from deciding

The conceptual stages leading to computation:

- 7th or 6th century BC: a Cretan philosopher-poet, Epimenides of Knossos, stated that "Cretans, always liars" thus challenging, for two and half millennia, the logically driven minds of the Occident
- 1900-1928: David Hilbert [Hilbert 1900, Hilbert & Ackermann '28] reformulated rigorously the problem as the ***decision problem***
- 1931: Kurt Gödel [Gödels '31] proved that the decision problem, and consequently the liar paradox, has no a logic solution
- 1936: Alonzo Church [Church '36], Stephene Kleene [Kleene '36], Emil Post [Post '36], Alan Turing [Turing '36] published independently their mathematical version of Gödel's approach thus providing four equivalent ***mathematical models*** for computing as a mechanism based on logic decision
- 1937: Claude E. Shannon [Shannon '48] defended at MIT its master thesis which became the foundation of practical digital circuit design during and after World War II
- 1946: John von Neumann [von Neumann '45], based on the Turing's approach and on the design and implementation made by John Mauchly and J. Presper Eckert for ENIAC, provided the ***abstract model*** for a mono-core computing machine (his approach is paralleled by the *Harvard* version)
- 1952: IBM announced the first mass-produced computer: IBM 701
- 1954: John Backus made the draft specification for the first high level programming language: FORTRAN
- 1964: the concept of computer ***architecture***, as the term used *"to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation"* [Amdahl '64], is introduced when specifying the IBM 360 computer.

Thus, the most important negative result in the history of mathematics, the Gödel's incompleteness theorem, founded theoretically the computing science.

Around 2010, market started to provide the (oxymoronic) General Purpose Graphic Processing Units (GPGPUs) chips with many lightweight cores. In the same decade, many consecrated cores were de-

ployed on the same silicon die with the hope that, having a well known instruction set architecture, the programming work will be simpler. But, unfortunately, the jump from a few number of cores to many cores per one silicon die has made the programming problem a very hard one. Putting more than one core per chip was done *without any criteria coming from a theoretic computational model*. The criteria involved in defining the organization and the architecture of the on the market one chip parallel engines were dominated by hardware restrictions or by the application domain requirements. The multi- or many-core product are *ad hoc* gathering of cores or application oriented organizations of huge computational resources. Under these circumstances, we must not be surprised by the inefficiency with which the computational resources are used.

The hardware-driven emergence of the parallel computing domain

The first steps leading to parallel computation:

- 1962 – ***manufacturing in quantity***: the first symmetrical MIMD engine is introduced on the computer market by Burroughs Corporation
- 1965-75 – ***architectural issues***: Edsger W. Dijkstra formulates, starting with [Dijkstra '65], the first concerns about specific parallel programming issues (such as critical regions problem, semaphores, the dining philosophers problem, guarded commands)
- 1974-82 – ***abstract machine models***: proposals of the first abstract models (bit vector models in [Pratt '74] and PRAM models in [Fortune '78], [Goldschlage '82]) start to come in after almost two decades of non-systematic experiments (started in the late 1950s) and the too early market production
- **?** – ***mathematical parallel computation model***: no one yet really considered it as mandatory, regrettably confusing it with abstract machine models, although it is there and wait to be considered (see Kleene's mathematical model for computation [Kleene '36]).

Now, in the 3rd decade of the 3rd millennium, after more than half century of chaotic development, it is obvious that *the history of parallel computing is distorted by missing stages and uncorrelated evolutions*.

Many people seem to accept the fact that the evolution of the parallel computing domain is driven mainly by corporations, and the gap between hardware and software is due to the too many different parallel computing machines and their fast evolution, when, in fact, things happens exactly the other way around: the gap exists because of the lack of an appropriate approach which must start from a well founded mathematical model for parallel computation.

7.1.2 Mathematical Model-Based Parallelism

The systems with global loops can be related with the model of *partial recursive functions* proposed by Stephen Cole Kleene [Kleene '36].

Kleene's Definition of Partial Recursion

Let be the positive integers $x, y, i \in \mathbf{N}$ and the sequence $X = \langle x_0, x_1, \ldots, x_{n-1} \rangle \in \mathbf{N}^n$. Any partial recursive function $f : \mathbf{N}^n \to \mathbf{N}$ can be computed, according to [Kleene '36], using three ***initial functions***:

- $ZERO(x) = 0$: the variable x takes the value **zero**
- $INC(x) = x+1$: **increment**s the variable $x \in \mathbf{N}$
- $SEL(i,X) = x_i$: i **select**s the value of x_i from the sequence of positive integers X

and the application of the following three ***rules***:

- **Composition:** $f(X) = g(h_1(X),\ldots,h_p(X))$, where: $f : \mathbf{N}^n \to \mathbf{N}$ is a total function if $g : \mathbf{N}^p \to \mathbf{N}$ and $h_i : \mathbf{N}^n \to \mathbf{N}$, for $i = 1,\ldots p$, are total functions
- **Primitive recursion:** $f(X,y) = g(X, f(X,(y-1)))$, with $f(X,0) = h(X)$ where: $f : \mathbf{N}^{n+1} \to \mathbf{N}$ is a total function if $g : \mathbf{N}^{n+1} \to \mathbf{N}$ and $h : \mathbf{N}^n \to \mathbf{N}$ are total functions.
- **Minimization:** $f(x) = \mu y[g(x,y) = 0]$, which means: the value of the function $f : \mathbf{N} \to \mathbf{N}$ is the smallest y, ***if any***, for which the function $g : \mathbf{N}^2 \to \mathbf{N}$ takes the value $g(x,y) = 0$.

While for initial functions the circuit aspects are obvious, for the three rules we must do some work. We will see that the composition rule has a direct circuit correspondent, but for the other two rules, primitive recursiveness and minimization, we where obliged to prove two theorems (see [Stefan '20]) which reduce them to multiple applications of specific forms of the first rule.

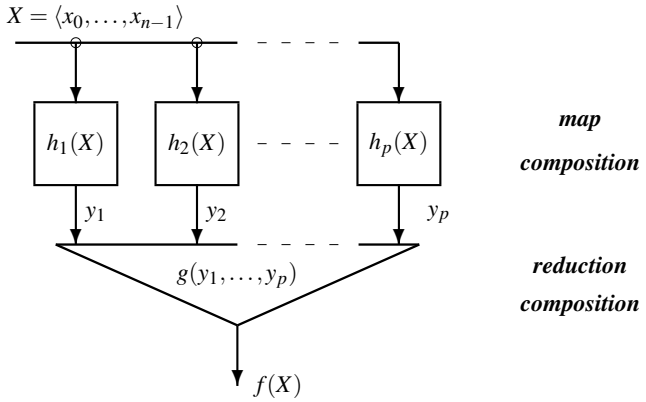


Figure 7.2: **The circuit version of composition:** $f(X) = g(h_1(X),\ldots,h_p(X))$**.** It is a two-layer construct: the parallel expanded *map* layer serially connected with the *log*-dept *reduction* layer.

Starting from the suggestion offered by the representation in Figure 7.2 one can conceive, based on the mathematical model of partially recursive functions, an abstract model for what a parallel computing system must be. A first approach is outlined in [Ştefan '14].

7.2 Main Limits in Computation

The main technological challenges we face in computer science are on the one hand technological and on the other hand theoretical, without the distinction being very clear. Some technological limitations come from theoretical limitations.

7.2.1 Technological Limitations

John von Neumann Bottleneck

We have focused on the development of powerful processors and large memories, but we have not yet found a way to connect them efficiently. The problem became more serious with the development of multi- and many-core systems, which prove more hungry for data because their processing capacity has increased significantly. The term *"von Neumann Bottleneck"* was introduced by John Backus [Backus '78] and refers to the limitation introduced by the connection, too narrow, between a processor and its data and program memory.

Recently, we have been proposing cellular solutions that assume the advanced interleaving of processing elements with memory structures in the so-called in-memory-processing systems, but we still do not manage to control them effectively.

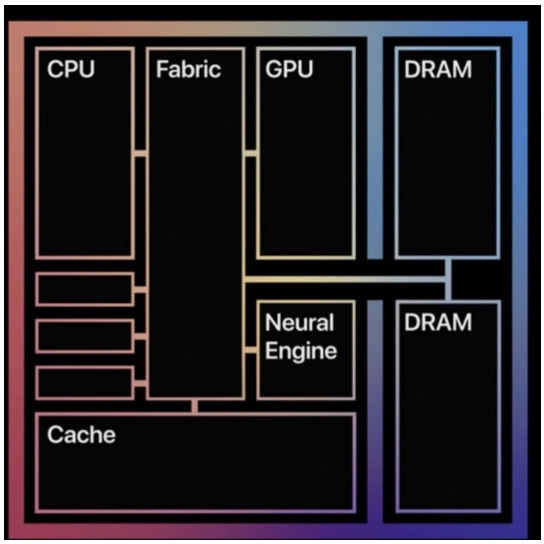


Figure 7.3: Apple's M1 chip which accommodates in the same package the processing structure (CPU & accelerators) with two memory chips to mitigate the von Neumann Bottleneck effect.

Possible solutions:

- very wide transfer buses (384 bits or higher) but which consume a lot of energy
- multi-chip solutions in a single package (example: Apple's MI, see Figure 7.3).
- ... we are waiting for other solutions.

Speed

Around 2002, processor manufacturers stopped the race to increase the clock frequency. The main reason: the energy consumed and the growth of the memory wall.

Energy

Energy is proportional with the chip area and the clock frequency. The designers decided to keep the frequency low and to increase area moderately by reducing the size of transistors. Increasing area helps in heat dissipation.

7.2.2 Theoretical Limitations

N=NP

One of the most studied problem in computer science is: ***is it true that P=NP?***. Until now, the answer to that problem, accepted by the majority of the academic world, is mainly "no".

Big O notation : $f(x) \in O(g(x))$ if there is a constant C and a value x_0 such that

$$|f(x)| \leq C \times g(x)$$

for all $x \geq x_0$.

The execution time for function $f(n)$, where n is the input size expressed in the number of bits, is $t_f(n)$. Then, using the Big O notation the function can be classified as follows:

- $t_f(n) \in O(1)$: the function $f(n)$ is executed in constant-time (independent of n)
- $t_f(n) \in O(n)$ the function $f(n)$ is executed in linear-time
- $t_f(n) \in O(n^2)$ the function $f(n)$ is executed in quadratic-time
- $t_f(n) \in O(n^k)$ the function $f(n)$ is executed in polynomial-time
- $t_f(n) \in O(2^n)$ the function $f(n)$ is executed in exponential-time
- $t_f(n) \in O(n!)$ the function $f(n)$ is executed in factorial-time

The easy-to-hard scale of computational problems (see also Figure 7.4):

P easy problems are quick to solve because the execution time for them is in $O(n^k)$.

NP medium problems are quick to verify (in polynomial-time) but slow to solve (in exponential-time)

NP-Complete hard problems are also quick to verify, slow to solve and can be reduced to any other NP-Complete problem in polynomial-time

NP-Hard hardest problems are slow to verify, slow to solve and can be reduced to any other NP problem, but some of these problems aren't even decidable (ex.: traveling salesman traveling, graph coloring, k-means clustering)

Q.7: What is the complexity class to which the calculation of the scalar product of two vectors belongs?

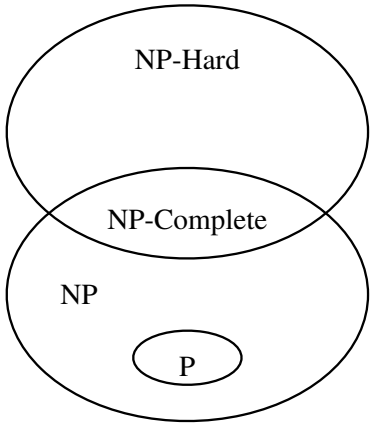


Figure 7.4: The relationship between computational problems on the easy-to-hard scale.

A question that's fascinated computer scientists is whether or not all algorithms in NP belong to P, because if only one is solved P-time, then (almost) all of them are solved. The main consequence: we must reconsider all the aspects of security in our systems because prime factorization is in NP.

Halting Problem

Halting problem was formulated in computational terms in 1936 by Alonzo Church, Stephen Kleene, Emil Post and Alan Turing as a reaction to the incompleteness theorem published by Kurt Gödel in 1931. And thus, the most important negative result in the history of mathematics underpins the science of computation.

> **Halting problem**: *Can we have a procedure that, receiving a program and the data it processes, decides if the program will stop executing in a finite time?*
>
> ***Not!***

Computing science started from this negative answer.

Appendices

Appendix A

Simulations

A.1 Waveforms Generator

ny simulation is based on the generation of waveforms with which the simulated circuit is stimulated on its inputs. These waveforms can be configured according to the simulated process or have the periodic shape of a clock used to synchronize the operation of the circuit.

```
/*************************************************************************
File name: waveFormGenerator.sv
Circuit name: no circuit, only wave formes
Description: two waveforms are generated, a "random" one and a periodical
             one: a clock signal
*************************************************************************/
module waveFormGenerator();
    logic randomWave;
    logic clock     ;

    initial begin       randomWave = 0  ;
                    #2  randomWave = 1  ;
                    #6  randomWave = 0  ;
                    #4  randomWave = 1  ;
                    #8  randomWave = 0  ;
                    #5  $stop           ;
            end
    initial begin               clock = 0       ;
                    forever #2  clock = ~clock  ;
            end
endmodule
```

By simulating the `waveFormGenerator.sv` module, the behavior represented in Figure A.1 results.

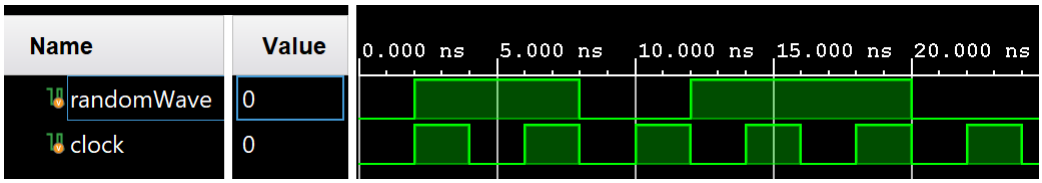


Figure A.1: Waveforms generated for a Vivado System Verilog simulation

A.2 Combo Simulations

A.2.1 ALU Simulation

The simplest behavioral design of an 8-function ALU is listed in the following module:

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

  always_comb case(func)
                3'b000: {crOut, out} = left + right + crIn;     //add
                3'b001: {crOut, out} = left - right - crIn;     //sub
                3'b010: {crOut, out} = {1'b0, left & right};    //and
                3'b011: {crOut, out} = {1'b0, left | right};    //or
                3'b100: {crOut, out} = {1'b0, left ^ right};    //xor
                3'b101: {crOut, out} = {1'b0, ~left};           //not
                3'b110: {crOut, out} = {1'b0, left};            //left
                3'b111: {crOut, out} = {1'b0, left >> 1};       //shr
                default {crOut, out} = 33'b0 - 1'b1;
              endcase
endmodule
```

The elaborated design provided by the Vivado tool based on the previous System Verilog is represented in Figure A.2.

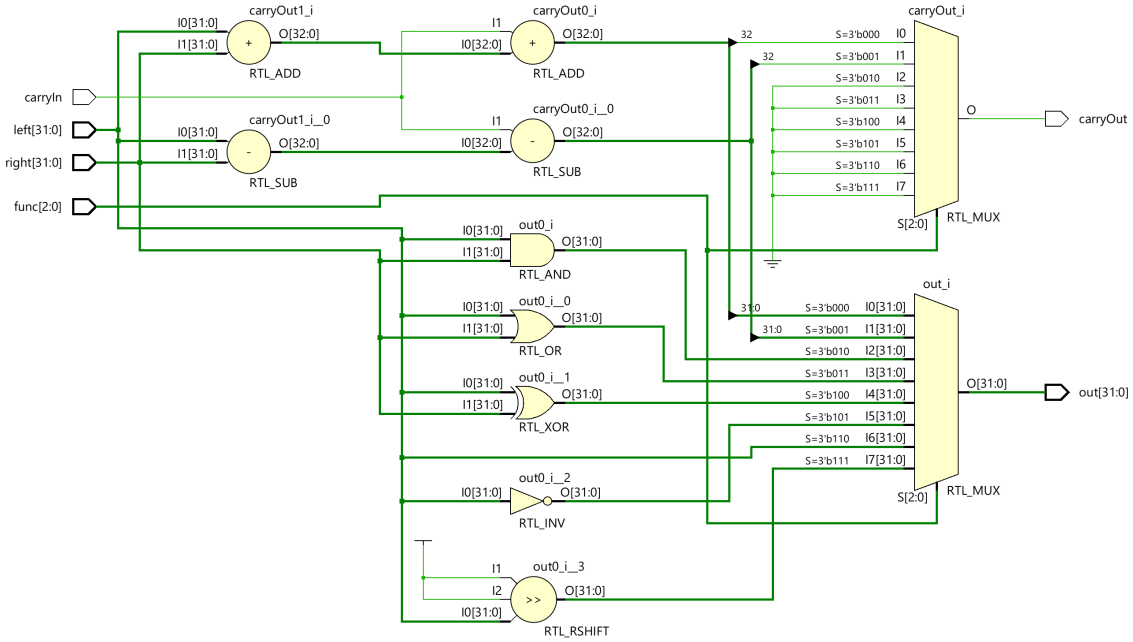


Figure A.2:

A.3 Memory Simulations

A.3.1 Pipelined Three Number Adder

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

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

```
                        .in0(syncReg),
                        .in1(pipeReg));
    always_ff @(posedge clock)  begin    syncReg <= in2  ;
                                         pipeReg <= sum  ;
                                end
endmodule
```

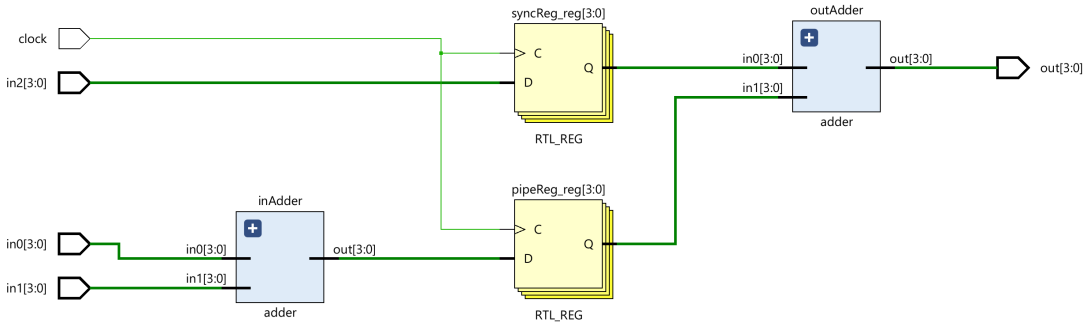


Figure A.3:

A.3.2 *Read-During-Write* Register File

In the design of the pipeline processor with forwarding mechanism, a register file that provides the currently written value at the output proved useful. The register file version that provides *read-during-write* facility requires an additional structure that performs comparison and multiplexing operations (see Figure A.4 obtained following the project described in the following module).

```
/*****************************************************************
File name:      fastRegFile.sv
Circuit name:   Register File
Description:    Three-pport, 32 32-bit words implemented with a
                synchronous RAM
*****************************************************************/
module fastRegFile( output  logic [31:0]    leftOut     ,
                    output  logic [31:0]    rightOut    ,
                    input   logic [31:0]    in          ,
                    input   logic [4:0]     leftAddr    ,
                    input   logic [4:0]     rightAddr   ,
                    input   logic [4:0]     destAddr    ,
                    input   logic           we          ,
                    input   logic           clock       );
    logic [31:0]    rf[0:31]    ;
```

```
    always_ff @(posedge clock) if (we) rf[destAddr] <= in    ;

    assign leftOut  =
        ((destAddr == leftAddr) && we) ? in : rf[leftAddr]  ;
    assign rightOut =
        ((destAddr == rightAddr) && we) ? in : rf[rightAddr];
endmodule
```

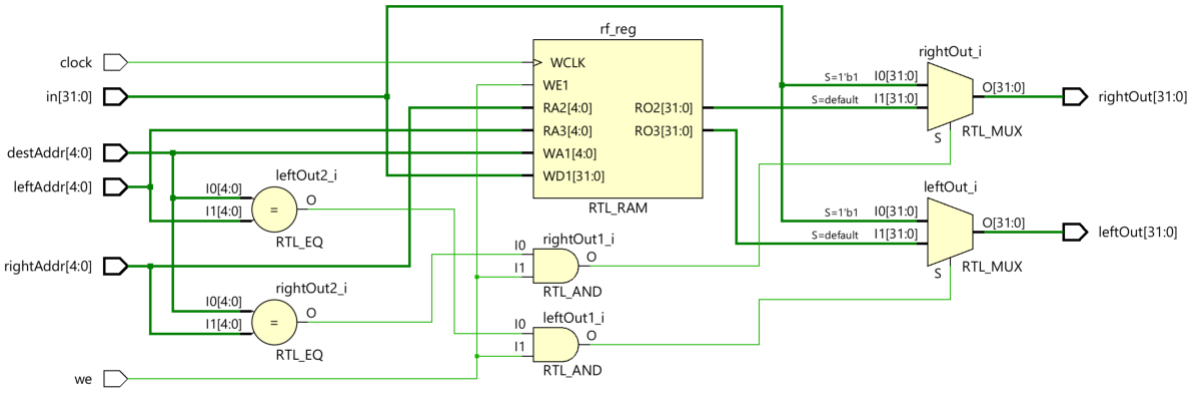


Figure A.4:

A.3.3

A.3.4

A.3.5

Appendix B

toyRISC Structural Implementation

B.1 Structure

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

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

    DCDtoyRISC  DCD(opCode      ,
                    intIn       ,
                    inta        ,
                    dataRead    ,
```

```
                    dataWrite    ,
                    reset        ,
                    clk          );

    PCtoyRISC    PC( nextPC   ,
                    pc       ,
                    leftOp   ,
                    v[9:0]   ,
                    opCode   ,
                    inta     ,
                    reset    ,
                    clk      );

    RALUtoyRISC RALU(   dataOut  ,
                        leftOp   ,
                        pc       ,
                        opCode   ,
                        dataIn   ,
                        v        ,
                        l        ,
                        r        ,
                        d        ,
                        inta     ,
                        clk      );

    assign addr = leftOp[15:0]  ;
endmodule // Synthesis results: #LUT=455, #FF=11, #DSP=6
```

```
/*****************************************************************
File name: DCDtoyRISC.sv
Circuit name:
Description:
*****************************************************************/
`include "DEFINES.vh"
module DCDtoyRISC( input   logic   [5:0]   opCode      ,
                   input   logic           intIn       ,
                   output  logic           inta        ,
                                           dataRead    ,
                                           dataWrite   ,
                   input   logic           reset       ,
                                           clk         );

    logic   ei  ;
    always_ff @(posedge clk) if (reset)                         ei <= 0;
                             else begin if (opCode == `eint)    ei <= 1;
                                        if (opCode == `dint)    ei <= 0;
                                        if (intIn & ei)         ei <= 0;
                                  end
```

```
    assign inta = intIn & ei;

    always_comb begin
        dataRead    = (opCode == `read)     ? 1'b1  : 1'b0  ;
        dataWrite   = (opCode == `store)    ? 1'b1  : 1'b0  ;
    end
endmodule
```

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

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

```
/*****************************************************************
File name: RALUtoyRISC.sv
                    toyRISC's RALU
*****************************************************************/
`include "DEFINES.vh"
module RALUtoyRISC( output  logic   [31:0]  dataOut ,
                    output  logic   [31:0]  leftOp  ,
                    input   logic   [9:0]   pc      ,
                    input   logic   [5:0]   opCode  ,
```

```
                    input   logic   [31:0]  dataIn   ,
                                            v       ,
                    input   logic   [4:0]   l       ,
                                            r       ,
                                            d       ,
                    input   logic           inta    ,
                                            clk     );
    logic   [31:0]  leftIn  ;
    logic   [31:0]  rightIn ;
    logic   [31:0]  muxOut  ;
    logic   [31:0]  aluOut  ;

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

    assign dataOut  = rightIn   ;
    assign leftOp   = leftIn    ;

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

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

```
/*************************************************************************
File name: registerFile.sv
Circuit name:
Description:
*************************************************************************/
`include "DEFINES.vh"
module registerFile(output  logic   [31:0]  leftOp      ,
                                            rightOp     ,
```

```
                    input   logic   [31:0]  in          ,
                    input   logic   [4:0]   leftAddr    ,
                                            rightAddr   ,
                                            destAddr    ,
                    input                   inta        ,
                    input   logic   [5:0]   opCode      ,
                    input   logic           clk         );

    logic   [31:0]  rf[0:31];
    logic           we      ;

    assign  we = inta | (opCode[5:4] == 2'b11) |
                        (opCode[5:4] == 2'b01) |
                        (opCode == 6'b10_0111)  ;

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

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

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

    always_comb
        case(opCode)
            `add    : out = leftIn + rightIn            ;
            `sub    : out = leftIn - rightIn            ;
            `addv   : out = leftIn + value              ;
            `mult   : out = leftIn * rightIn            ;
            `multv  : out = leftIn * value              ;
            `addc   : out = (leftIn + rightIn) > 2**32 - 1;
            `subc   : out = (leftIn - rightIn) > 2**32 - 1;
            `addvc  : out = (leftIn + value) > 2**32 - 1  ;
            `lsh    : out = leftIn >> 1                 ;
            `ash    : out = {leftIn[31], leftIn[31:1]}  ;
            `move   : out = leftIn                      ;
            `swap   : out = {leftIn[15:0], leftIn[31:16]} ;
            `bwnot  : out = ~leftIn                     ;
```

```
            `bwand  : out = leftIn & rightIn            ;
            `bwor   : out = leftIn | rightIn            ;
            `bwxor  : out = leftIn ^ rightIn            ;
            default : out = leftIn                      ;
        endcase
endmodule
```

```
/*****************************************************************
File name: bigMux.sv
Circuit name:
Description:
*****************************************************************/
`include "DEFINES.vh"
module bigMux( input   logic   [31:0]  aluOut  ,
                                       dataIn  ,
                                       value   ,
               input   logic   [9:0]   pc      ,
               input   logic   [5:0]   opCode  ,
               input   logic           inta    ,
               output  logic   [31:0]  muxOut  );
    logic   [1:0]   sel ;

    assign sel = inta ? 2'b00 : opCode[5:4] ;

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

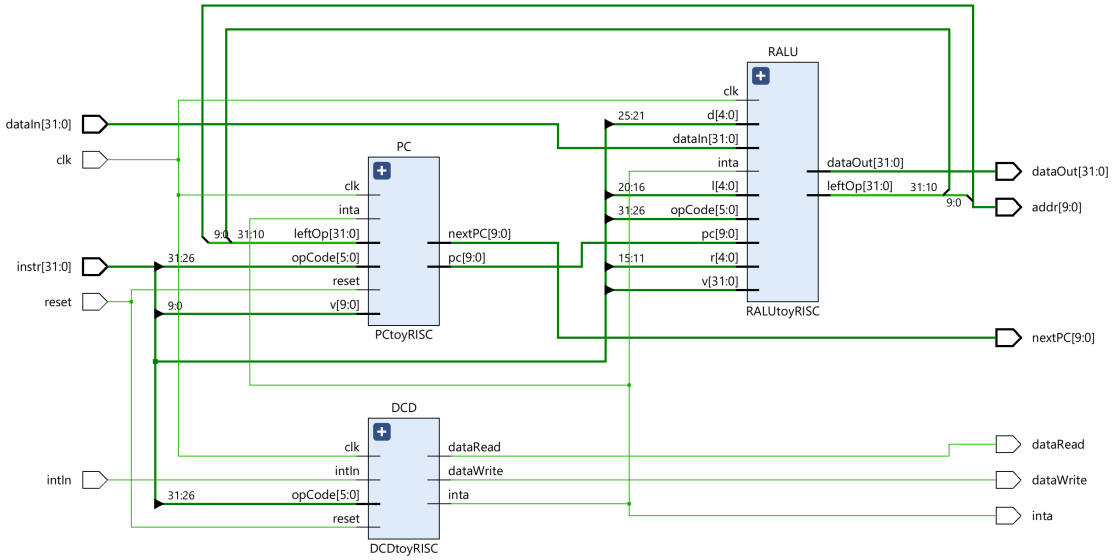


Figure B.1:

B.2 Code generator

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

    `include "DEFINES.vh"

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

```
    endtask

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

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

        v = labelTab[labelIndex] - addrCounter;
    endtask

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

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

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

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

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

    task BRNZ; // branch if not zero
```

```
        input   [4:0]   left    ;
        input   [9:0]   label   ;

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

    task RET; // return from subroutine
        input   [4:0] left  ;

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

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

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

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

// ARITHMETIC & LOGIC INSTRUCTIONS
```

```
task ADD; // addition
    input   [4:0]   dest    ;
    input   [4:0]   left    ;
    input   [4:0]   right   ;

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

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

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

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

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

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

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

```
    endtask

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

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

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

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

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

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

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

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

```
            endLine         ;
    end
endtask

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

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

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

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

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

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

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

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

```
    endtask

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

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

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

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

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

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

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

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

```
        end
    endtask

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

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

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

        begin   opCode  = `load ;
                d       = dest  ;
                l       = 5'b0  ;
                v       = 16'b0 ;
                endLine         ;
        end
    endtask

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

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

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

        begin   opCode  = `val  ;
                d       = dest  ;
                l       = 5'b0  ;
                v       = value;;
                endLine         ;
        end
    endtask
```

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

B.3 Simulator

```
/*************************************************************************
File name: testRISC.sv
Circuit name:
Description:
*************************************************************************/
`include "DEFINES.vh"
module testRISC;
    logic           inta    ;
    logic           intIn   ;
    logic           reset   ;
    logic           clk     ;

    integer i   ;

    `include "RISCcodeGenerator.sv"

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

    initial begin
            intIn       = 1 ;
            reset       = 1 ;
            // DISPLAY THE CONTENT OF PROGRAM MEMORY
            for (i=0; i<8; i=i+1)
                $display("progMemory[%0d] \t = %b", i,
                          progMemory[i]) ;
        #4  reset       = 0 ;
        #40 $finish     ;
    end

    logic   [31:0]  instr       ;
    logic   [9:0]   nextPC      ;
    logic   [31:0]  dataIn      ;
    logic   [31:0]  dataOut     ;
    logic   [9:0]   addr        ;
    logic           dataRead    ;
```

```
    logic            dataWrite    ;

    logic    [31:0]  dataMemory[0:1023]  ;
    logic    [31:0]  progMemory[0:1023]  ;
    logic    [31:0]  dataMemOut          ;

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

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

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

B.4 Testing

```
/*************************************************************************
File name: program.sv
Description: programs to validate the main types of instructions
*************************************************************************/
// Testing instantiating registers & arithmetic operation
/*
            VAL(0,2)     ;
            VAL(1,4)     ;
            VAL(2,8)     ;
            MULT(3,1,2)  ;
            HALT         ;
//*/
// Testing interrupt mechanism
//*
            VAL(31,10)   ;
            VAL(2,23)    ;
            VAL(0,13)    ;
            EI           ;
            ADDV(0,0,2)  ;
            NOP          ;
            ADDV(0,0,4)  ;
            HALT         ;
            NOP          ;
            NOP          ;
        // subroutine triggered by interrupt
            DI           ;
            VAL(3,44)    ;
            RET(30)      ;
//*/
// Testing jump & branch
/*
            VAL(0,3)     ;
    LB(1);  ADDV(0,0,-1);
            NOP          ;
            RJMP(1)      ; // BRNZ(0,1)
            HALT         ;
//*/
// Testing data memory
/*
            VAL(0,1)     ;
            VAL(1,55)    ;
            STORE(0,1)   ;
            READ(0)      ;
            LOAD(2)      ;
            HALT         ;
//*/
```

```
progMemory[0]    = 01011111111000000000000000001010
progMemory[1]    = 01011100010000000000000000010111
progMemory[2]    = 01011100000000000000000000001101
progMemory[3]    = 00100000000000000000000000000000
progMemory[4]    = 11001000000000000000000000000010
progMemory[5]    = 11001000000000000000000000000000
progMemory[6]    = 11001000000000000000000000000100
progMemory[7]    = 00011000000000000000000000000000
progMemory[8]    = 11001000000000000000000000000000
progMemory[9]    = 11001000000000000000000000000000
progMemory[10]   = 00100100000000000000000000000000
progMemory[11]   = 01011100011000000000000000101100
progMemory[12]   = 00010100000111100000000000000000
progMemory[13]   = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
progMemory[14]   = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
progMemory[15]   = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

t=0  pc=   x  RF=[x, x, x, x]    out=x  leftIn=x  rightIn=x  value=x  opCode=x  ei=x inta=x
t=1  pc=1023  RF=[x, x, x, x]    out=x  leftIn=x  rightIn=x  value=x  opCode=x  ei=0 inta=0
t=3  pc=1023  RF=[x, x, x, x]    out=x  leftIn=x  rightIn=x  value=10 opCode=23 ei=0 inta=0
t=5  pc=   0  RF=[x, x, x, x]    out=x  leftIn=x  rightIn=x  value=10 opCode=23 ei=0 inta=0
t=7  pc=   1  RF=[x, x, x, x]    out=x  leftIn=x  rightIn=x  value=23 opCode=23 ei=0 inta=0
t=9  pc=   2  RF=[x, x, 23, x]   out=x  leftIn=x  rightIn=x  value=13 opCode=23 ei=0 inta=0
t=11 pc=   3  RF=[13, x, 23, x]  out=13 leftIn=13 rightIn=13 value=0  opCode=8  ei=0 inta=0
t=13 pc=   4  RF=[13, x, 23, x]  out=12 leftIn=10 rightIn=13 value=2  opCode=50 ei=1 inta=1
t=15 pc=  10  RF=[13, x, 23, x]  out=13 leftIn=13 rightIn=13 value=0  opCode=50 ei=0 inta=0
t=17 pc=  11  RF=[13, x, 23, x]  out=13 leftIn=13 rightIn=13 value=44 opCode=23 ei=0 inta=0
t=19 pc=  12  RF=[13, x, 23, 44] out=4  leftIn=4  rightIn=13 value=0  opCode=5  ei=0 inta=0
t=21 pc=   4  RF=[13, x, 23, 44] out=15 leftIn=13 rightIn=13 value=2  opCode=50 ei=0 inta=0
t=23 pc=   5  RF=[15, x, 23, 44] out=15 leftIn=15 rightIn=15 value=0  opCode=50 ei=0 inta=0
t=25 pc=   6  RF=[15, x, 23, 44] out=19 leftIn=15 rightIn=15 value=4  opCode=50 ei=0 inta=0
t=27 pc=   7  RF=[19, x, 23, 44] out=19 leftIn=19 rightIn=19 value=0  opCode=6  ei=0 inta=0
```

Appendix C

Pipelined toyRISC

C.1 Structure

```
/*************************************************************************
File name: toyRISCsystem.sv
Circuit name:
Description:
*************************************************************************/
`include "0_DEFINES.vh"
module toyRISCsystem(   output  logic   inta    ,
                        input   logic   intIn   ,
                                        reset   ,
                                        clk     );
    logic   [31:0]  instruction ;
    logic   [9:0]   PC          ;
    logic   [31:0]  dataIn      ;
    logic   [31:0]  dataOut     ;
    logic   [9:0]   dataAddr    ;
    logic           dataRead    ;
    logic           dataWrite   ;

    toyRISC processor(  instruction ,
                        PC          ,
                        intIn       ,
                        inta        ,
                        dataIn      ,
                        dataOut     ,
                        dataAddr    ,
                        dataRead    ,
                        dataWrite   ,
                        reset       ,
                        clk         );
    memorySystem memSys(instruction ,
                        PC          ,
                        dataIn      ,
                        dataOut     ,
```

```
                        dataAddr    ,
                        dataRead    ,
                        dataWrite   ,
                        clk         );
endmodule
```

```
/*****************************************************************
File name: memorySystem.sv
Circuit name:
Description:
*****************************************************************/
module memorySystem(output  logic   [31:0]  instruction ,
                    input   logic   [9:0]   PC          ,
                    output  logic   [31:0]  dataIn      ,
                    input   logic   [31:0]  dataOut     ,
                    input   logic   [9:0]   dataAddr    ,
                    input   logic           dataRead    ,
                    input   logic           dataWrite   ,
                    input   logic           clk         );

    logic   [31:0]  dataMemory[0:1023]  ;
    logic   [31:0]  progMemory[0:1023]  ;

    assign instruction = progMemory[PC] ;

    always_ff @(posedge clk) begin
        if (dataWrite) dataMemory[dataAddr] <= dataOut  ;
    end

    assign dataIn   = dataMemory[dataAddr];

endmodule
```

```
/*****************************************************************
File name: toyRISC.sv
Circuit name:
Description:
*****************************************************************/
`include "0_DEFINES.vh"
module toyRISC( input   logic [31:0]  instruction   ,
                output  logic [9:0]   PC            ,
                input   logic         intIn         ,
                output  logic         inta          ,
                input   logic [31:0]  dataIn        ,
                output  logic [31:0]  dataOut       ,
```

```
            output  logic [9:0]   dataAddr      ,
            output  logic          dataRead      ,
            output  logic          dataWrite     ,
            input   logic          reset         ,
            input   logic          clk           );
logic [31:0]    instruction1;
logic [9:0]     PC1         ;
logic [31:0]    leftOp      ;
logic [1:0]     nextPCsel   ;
logic [31:0]    results     ;
logic [5:0]     opCode1     ;
logic [31:0]    leftOp2     ;
logic [31:0]    rightOp2    ;
logic [31:0]    value2      ;
logic [5:0]     opCode2     ;
logic [4:0]     destAddr2   ;
logic           we3         ;
logic [4:0]     destAddr3   ;
logic [31:0]    result3     ;
logic [5:0]     decode      ;
logic           we          ;
logic [1:0]     opSel       ;
logic [1:0]     opSel2      ;
logic           zero        ;
logic [5:0]     opCode3     ;

DECODE dcd( .opCode1    (instruction1[31:26]),
            .zero       (zero               ),
            .opCode2    (opCode2            ),
            .intIn      (intIn              ),
            .inta       (inta               ),
            .nextPCsel  (nextPCsel          ),
            .opSel      (opSel              ),
            .we         (we                 ),
            .dataRead   (dataRead           ),
            .dataWrite  (dataWrite          ),
            .reset      (reset              ),
            .clk        (clk                ));

INSTRFETCH progCount(   instruction ,
                        PC          ,
                        instruction1,
                        PC1         ,
                        leftOp      ,
                        nextPCsel   ,
                        reset       ,
                        clk         );

OPSFETCH regFile(   instruction1,
                    PC1         ,
```

```
                        inta        ,
                        we3         ,
                        destAddr3   ,
                        result3     ,
                        opSel       ,
                        opSel2      ,
                        zero        ,
                        leftOp      ,
                        leftOp2     ,
                        rightOp2    ,
                        value2      ,
                        opCode2     ,
                        destAddr2   ,
                        clk         );

    EXECUTE execute(leftOp2     ,
                    rightOp2    ,
                    value2      ,
                    opCode2     ,
                    destAddr2   ,
                    dataIn      ,
                    we          ,
                    opSel2      ,
                    dataAddr    ,
                    dataOut     ,
                    we3         ,
                    destAddr3   ,
                    result3     ,
                    opCode3     ,
                    clk         );
endmodule // Synthesis results: #LUT=335, #FF=204, #DSP=3
```

```
/*************************************************************************
File name: DECODE.sv
Circuit name:
Description:
*************************************************************************/
`include "0_DEFINES.vh"
module DECODE(  input   logic [5:0] opCode1     ,
                input   logic       zero        ,
                input   logic [5:0] opCode2     ,
                input   logic       intIn       ,
                output  logic       inta        ,
                output  logic [1:0] nextPCsel   ,
                output  logic [1:0] opSel       ,
                output  logic       we          ,
                output  logic       dataRead    ,
                output  logic       dataWrite   ,
```

```
                input   logic       reset       ,
                input   logic       clk         );
    logic [2:0] intState    ;

/*
    intState = 000: intDisable
    intState = 001: intEnable
    intState = 010: intExec1
    intState = 011: intExec2
    intState = 100: intExec3
*/
// Interrupt automaton
    always_ff @(posedge clk)
     if (reset)                                 intState <= 3'b000  ;
      else  begin
            if (opCode1 == `eint)               intState <= 3'b001  ;
            if (opCode1 == `dint)               intState <= 3'b000  ;
            if (intIn && (intState == 3'b001))  intState <= 3'b010  ;
            if (intState == 3'b010)             intState <= 3'b011  ;
            if (intState == 3'b011)             intState <= 3'b100  ;
            if (intState == 3'b100)             intState <= 3'b000  ;
        end
    //assign intr = intState == 2'b10   ;
    assign inta = intState == 3'b010    ;

    logic [5:0] jmpSel  ;

    assign jmpSel = (intState > 3'b001) ? `halt : opCode1   ;

    always_comb
        if (inta)           nextPCsel = 2'b11               ;
         else   case(jmpSel)
                    `halt   : nextPCsel = 2'b00             ;
                    `rjmp   : nextPCsel = 2'b10             ;
                    `zbr    : nextPCsel = zero ? 2'b10 : 2'b01  ;
                    `nzbr   : nextPCsel = zero ? 2'b01 : 2'b10  ;
                    `ret    : nextPCsel = 2'b11             ;
                    default : nextPCsel = 2'b01             ;
                endcase

    always_comb case(opCode1)
                    `load   : opSel = 2'b10 ;
                    `addv   : opSel = 2'b01 ;
                    `multv  : opSel = 2'b01 ;
                    `addvc  : opSel = 2'b01 ;
                    `val    : opSel = 2'b01 ;
                    default : opSel = 2'b00 ;
                endcase
```

```
    assign dataWrite    = opCode2 == `store                          ;
    assign dataRead     = opCode2 == `read                           ;
    assign we           = ((opCode2[5:4] == 2'b11)  |
                           (opCode2 == `load)       |
                           (opCode2 == `val)) & !(intState > 3'b001) ;
endmodule
```

```
/*****************************************************************
File name: INSTRFETCH.sv
Circuit name:
Description:
*****************************************************************/
module INSTRFETCH(  input   logic [31:0]    instruction ,
                    output  logic [9:0]     PC          ,
                    output  logic [31:0]    instruction1,
                    output  logic [9:0]     PC1         ,
                    input   logic [31:0]    leftOut     ,
                    input   logic [1:0]     nextPCsel   ,
                    input   logic           reset       ,
                    input   logic           clk         );

    // PipeReg0
    always @(posedge clk)
        if (reset)  begin           PC <= -1                        ;
                                    instruction1 <= 0               ;
                                    //PC1        <= 0               ;
                    end
        else    begin   case(nextPCsel)
                            2'b00:  PC <= PC                        ;
                            2'b01:  PC <= PC + 1                    ;
                            2'b10:  PC <= instruction1[9:0] + PC1   ;
                            2'b11:  PC <= leftOut                   ;
                        endcase
    // PipeReg1
                        instruction1 <= instruction ;
                        PC1          <= PC          ;
                end
endmodule
```

```
/*****************************************************************
File name: OPSFETCH.sv
Circuit name:
Description:
*****************************************************************/
module OPSFETCH(input   logic [31:0]    instruction1 ,
```

```
                input   logic [9:0]     PC1         ,
                input   logic           inta        ,
                input   logic           we3         ,
                input   logic [4:0]     destAddr3   ,
                input   logic [31:0]    result3     ,
                input   logic [1:0]     opSel       ,
                output  logic [1:0]     opSel2      ,
                output  logic           zero        ,
                output  logic [31:0]    leftOp      ,
                output  logic [31:0]    leftOp2     ,
                output  logic [31:0]    rightOp2    ,
                output  logic [31:0]    value2      ,
                output  logic [5:0]     opCode2     ,
                output  logic [4:0]     destAddr2   ,
                input   logic           clk         );
    logic   [31:0]  rf[0:31]    ;
    logic   [31:0]  rightOp     ;
    logic           we          ;
    logic   [4:0]   destAddr    ;
    logic   [31:0]  result      ;
    logic   [4:0]   leftAddr    ;
    logic   [4:0]   rightAddr   ;


    always_ff @(posedge clk) if (we) rf[destAddr] <= result ;

    assign leftOp       = rf[leftAddr]          ;
    assign rightOp      = rf[rightAddr]         ;
    assign zero         = leftOp == 0           ;
    assign rightAddr    = instruction1[15:11]   ;
// PipeReg2:
    always_ff @(posedge clk) begin
        leftOp2     <=  leftOp                                      ;
        rightOp2    <=  rightOp                                     ;
        value2      <=  {{16{instruction1[15]}}, instruction1[15:0]};
        opSel2      <=  opSel                                       ;
        opCode2     <=  instruction1[31:26]                         ;
        destAddr2   <=  instruction1[25:21]                         ;
    end

    intUnit intunit(.we         (we                     ),
                    .destAddr   (destAddr               ),
                    .result     (result                 ),
                    .leftAddr   (leftAddr               ),
                    .we3        (we3                    ),
                    .destAddr3  (destAddr3              ),
                    .result3    (result3                ),
                    .leftAddr1  (instruction1[20:16]    ),
                    .PC1        (PC1                    ),
                    .inta       (inta                   ));
```

```
endmodule
```

```
/*************************************************************************
File name: EXECUTE.sv
Circuit name:
Description:
*************************************************************************/
`include "0_DEFINES.vh"
module EXECUTE( input   logic [31:0]   leftOp2     ,
                input   logic [31:0]   rightOp2    ,
                input   logic [31:0]   value2      ,
                input   logic [5:0]    opCode2     ,
                input   logic [4:0]    destAddr2   ,
                input   logic [31:0]   dataIn      ,
                input   logic          we          ,
                input   logic [1:0]    opSel2      ,
                output  logic [9:0]    dataAddr    ,
                output  logic [31:0]   dataOut     ,
                output  logic          we3         ,
                output  logic [4:0]    destAddr3   ,
                output  logic [31:0]   result3     ,
                output  logic [5:0]    opCode3     ,
                input   logic          clk         );

    logic [31:0]    rightIn ;
    logic [31:0]    out     ;
    logic [31:0]    dataIn3 ;

    assign dataAddr = leftOp2[9:0]  ;
    assign dataOut  = rightOp2      ;

    always_comb case(opSel2)
                    2'b00: rightIn = rightOp2   ;
                    2'b01: rightIn = value2     ;
                    2'b10: rightIn = dataIn3    ;
                    2'b11: rightIn = 31'b0      ;
                endcase

    ALU alu(.out    (out    ),
            .leftIn (leftOp2),
            .rightIn(rightIn),
            .opCode (opCode2));

    // PipeReg3
    always_ff @(posedge clk)    begin   we3         <= we       ;
                                        destAddr3   <= destAddr2;
                                        result3     <= out      ;
                                        opCode3     <= opCode2  ;
```

```
                                        dataIn3     <= dataIn   ;
                                    end
endmodule
```

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

    logic [32:0]    sum ;
    logic [32:0]    dif ;

    assign sum = leftIn + rightIn   ;
    assign dif = leftIn - rightIn   ;

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

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

```
File name: intUnit.sv
Circuit name:
Description:
*************************************************************************/
module intUnit( output  logic           we          ,
                output  logic   [4:0]   destAddr    ,
                output  logic   [31:0]  result      ,
                output  logic   [4:0]   leftAddr    ,
                input   logic           we3         ,
                input   logic   [4:0]   destAddr3   ,
                input   logic   [31:0]  result3     ,
                input   logic   [4:0]   leftAddr1   ,
                input   logic   [9:0]   PC1         ,
                input   logic           inta        );

    assign we       = inta ? 1'b1           : we3       ;
    assign destAddr = inta ? 5'b11110       : destAddr3 ;
    assign result   = inta ? {21'b0, PC1}   : result3   ;
    assign leftAddr = inta ? 5'b11111       : leftAddr1 ;
endmodule
```

C.2 Code generator

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

    `include "0_DEFINES.vh"

    task endLine;
     begin
        dut.memSys.progMemory[addrCounter][31:0] =
            {   opCode  ,
                d       ,
                l       ,
                v       }                   ;
            addrCounter = addrCounter + 1   ;
```

```
     end
    endtask

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

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

        v = labelTab[labelIndex] - addrCounter  ;
        endtask

// CONTROL INSTRUCTIONS
    task NOP; // no operation
        begin   opCode  = `nop  ;
                {d,l,v} = 26'b0 ;
                endLine         ;
        end
    endtask

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

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

    task BRZ; // branch if zero
        input   [4:0]   left    ;
        input   [15:0]  label   ;

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

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

```
        input   [15:0]  label   ;

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

    task RET; // return from subroutine
        input   [4:0] left  ;

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

    task HALT; // halt running
        begin   opCode  = `halt ;
                {d,l,v} = 26'b0 ;
                endLine         ;
        end
    endtask

    task EI; // enable interrupt
        begin   opCode  = `eint ;
                {d,l,v} = 26'b0 ;
                endLine         ;
        end
    endtask

    task DI; // disable interrupt
        begin   opCode  = `dint ;
                {d,l,v} = 26'b0 ;
                endLine         ;
        end
    endtask

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

        begin   opCode  = `add          ;
                d       = dest          ;
```

```
            l       = left          ;
            v       = {right, 11'b0};
            endLine                 ;
    end
endtask

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    begin   opCode  = `bwnot;
```

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

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

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

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

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

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

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

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

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

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

    begin   opCode  = `load ;
            d       = dest  ;
            l       = 5'b0  ;
            v       = 16'b0 ;
            endLine         ;
    end
endtask

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

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

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

    begin   opCode  = `val  ;
            d       = dest  ;
            l       = 5'b0  ;
            v       = value ;
            endLine         ;
    end
endtask

// RUNNING
initial begin   addrCounter = 0;
                `include "0_program.sv" // first pass
                addrCounter = 0;
                `include "0_program.sv" // second pass
        end
```

C.3 Simulator

```
/*************************************************************************
File name: testRISC.sv
Circuit name:
Description:
*************************************************************************/
`include "0_DEFINES.vh"
module testRISC;
    logic   inta    ;
    logic   intIn   ;
    logic   reset   ;
    logic   clk     ;

    integer i   ;

    `include "0_RISCcodeGenerator.sv"

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

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

    toyRISCsystem dut(  inta    ,
                        intIn   ,
                        reset   ,
                        clk     );

    // MONITOR FOR PROGRAM LAOD & CONTROLLER
    initial begin
     $monitor("t=%0d pc=%d  RF=[%0d, %0d, %0d, %0d, %0d, %0d, %0d, %0d]
            leftOp2=%0d rightOp2=%0d result3=%0d   intState=%b inta=%b",
                $time,
                dut.processor.PC,
                dut.processor.regFile.rf[0],
                dut.processor.regFile.rf[1],
                dut.processor.regFile.rf[2],
                dut.processor.regFile.rf[3],
                dut.processor.regFile.rf[4],
                dut.processor.regFile.rf[5],
```

```
                dut.processor.regFile.rf[30],
                dut.processor.regFile.rf[31],
                dut.processor.leftOp2,
                dut.processor.rightOp2,
                dut.processor.result3,
                dut.processor.dcd.intState,
                dut.processor.dcd.inta);
    end
endmodule
```

C.4 Testing

```
/*************************************************************************
File name: program.sv
Circuit name:
Description:
*************************************************************************/
/*
            NOP         ;
            VAL(0,1)    ;
            VAL(1,2)    ;
            VAL(2,3)    ;
            VAL(3,4)    ;
            VAL(4,5)    ;
            NOP         ;
            NOP         ;
            ADD(5,4,3)  ;
            HALT        ;
            HALT        ;
//*/


/*
            VAL(0,2)    ;
            VAL(1,4)    ;
            VAL(2,8)    ;
            NOP         ;
            MULT(3,1,2) ;
            NOP         ;
            HALT        ;
//*/
//*
            VAL(31,13)  ;
            VAL(2,23)   ;
            VAL(0,13)   ;
            VAL(1,13)   ;
```

```
                EI              ;
                ADDV(0,0,1)     ;
                VAL(1,1)        ;
                VAL(2,222)      ;
                NOP             ;
                ADDV(0,0,4)     ;
                HALT            ;
                HALT            ;
                NOP             ;
          // subroutine triggered by interrupt
                ADDV(30,30,-1)  ;
                NOP             ;
                NOP             ;
                NOP             ;
                RET(30)         ;
//*/
/*
                NOP             ;
                VAL(0,-3)       ;
                NOP             ;
                NOP             ;
     LB(1);     ADDV(0,0,1);
                NOP             ;
                BRNZ(0,1)       ;
                NOP             ;
                HALT            ;
                HALT            ;
//*/
/*
                VAL(0,1)        ;
                VAL(1,55)       ;
                STORE(0,1)      ;
                READ(0)         ;
                LOAD(2)         ;
                HALT            ;
//*/
```

Appendix D

Forwarding toyRISC

D.1 Structure

```
/*************************************************************************
File name: toyRISCsystem.sv
Circuit name:
Description:
*************************************************************************/
`include "0_DEFINES.vh"
module toyRISCsystem(   output  logic   inta    ,
                        input   logic   intIn   ,
                                        reset   ,
                                        clk     );
    logic   [31:0]  instruction ;
    logic   [9:0]   PC          ;
    logic   [31:0]  dataIn      ;
    logic   [31:0]  dataOut     ;
    logic   [9:0]   dataAddr    ;
    logic           dataRead    ;
    logic           dataWrite   ;

    toyRISC processor(  instruction ,
                        PC          ,
                        intIn       ,
                        inta        ,
                        dataIn      ,
                        dataOut     ,
                        dataAddr    ,
                        dataRead    ,
                        dataWrite   ,
                        reset       ,
                        clk         );
    memorySystem memSys(instruction ,
                        PC          ,
                        dataIn      ,
                        dataOut     ,
```

```
                        dataAddr    ,
                        dataRead    ,
                        dataWrite   ,
                        clk         );
endmodule
```

```
/*************************************************************************
File name: memorySystem.sv
Circuit name:
Description:
*************************************************************************/
module memorySystem(output  logic   [31:0]  instruction ,
                    input   logic   [9:0]   PC          ,
                    output  logic   [31:0]  dataIn      ,
                    input   logic   [31:0]  dataOut     ,
                    input   logic   [9:0]   dataAddr    ,
                    input   logic           dataRead    ,
                    input   logic           dataWrite   ,
                    input   logic           clk         );

    logic   [31:0]  dataMemory[0:1023]  ;
    logic   [31:0]  progMemory[0:1023]  ;

    assign instruction = progMemory[PC] ;

    always_ff @(posedge clk) begin
        if (dataWrite) dataMemory[dataAddr] <= dataOut  ;
    end

    assign dataIn   = dataMemory[dataAddr];

endmodule
```

```
/*************************************************************************
File name: .sv
Circuit name:
Description:
*************************************************************************/
/*************************************************************************
File name: toyRISC.sv
                        toyRISC
*************************************************************************/
`include "0_DEFINES.vh"
module toyRISC( input   logic [31:0]  instruction   ,
                output  logic [9:0]   pc            ,
```

```
              input   logic          intIn       ,
              output  logic          inta        ,
              input   logic [31:0]   dataIn      ,
              output  logic [31:0]   dataOut     ,
              output  logic [9:0]    dataAddr    ,
              output  logic          dataRead    ,
              output  logic          dataWrite   ,
              input   logic          reset       ,
              input   logic          clk         );
logic [31:0]     instruction1;
logic [31:0]     instruction2;
logic [31:0]     instruction3;
logic [31:0]     instruction4;
logic [9:0]      pc1         ;
logic [31:0]     leftOp      ;
logic [1:0]      nextPCsel   ;
logic            we4         ;
logic [31:0]     result4     ;
logic [3:0]      opSel       ;
logic [3:0]      opSel2      ;
logic            zero        ;
logic [31:0]     leftOp2     ;
logic [31:0]     rightOp2    ;
logic [31:0]     leftOp3     ;
logic [31:0]     rightOp3    ;
logic [31:0]     result3     ;
logic            we          ;
logic            memSel      ;

DECODE dcd( .instruction1   (instruction1   ),
            .instruction2   (instruction2   ),
            .instruction3   (instruction3   ),
            .zero           (zero           ),
            .intIn          (intIn          ),

            .inta           (inta           ),
            .nextPCsel      (nextPCsel      ),
            .opSel          (opSel          ),
            .memSel         (memSel         ),
            .we             (we             ),
            .dataRead       (dataRead       ),
            .dataWrite      (dataWrite      ),

            .reset          (reset          ),
            .clk            (clk            ));

INSTRFETCH progCount(   instruction ,
                        leftOp      ,
                        nextPCsel   ,
```

```
                        pc          ,
                        instruction1,
                        pc1         ,

                        reset       ,
                        clk         );

OPSFETCH regFile(   instruction1,
                    instruction4,
                    pc1         ,
                    inta        ,
                    we4         ,
                    result4     ,
                    opSel       ,

                    zero        ,
                    leftOp      ,
                    leftOp2     ,
                    rightOp2    ,
                    opSel2      ,
                    instruction2,

                    clk         );

EXECUTE execute(leftOp2     ,
                rightOp2    ,
                opSel2      ,
                instruction2,
                result4     ,

                leftOp3     ,
                result3     ,
                rightOp3    ,
                instruction3,

                clk         );

DATAMEM datamem(result3     ,
                instruction3,
                we          ,
                memSel      ,
                dataIn      ,

                result4     ,
                instruction4,
                we4         ,

                clk         );

assign dataAddr = leftOp3   ;
```

```
    assign dataOut   = rightOp3   ;
endmodule // Synthesis results: #LUT=335, #FF=204, #DSP=3
```

```
/*************************************************************************
File name: DECODE.sv
Circuit name:
Description:
*************************************************************************/
`include "0_DEFINES.vh"
module DECODE(  input   logic [31:0]    instruction1,
                input   logic [31:0]    instruction2,
                input   logic [31:0]    instruction3,
                input   logic           zero        ,
                input   logic           intIn       ,

                output  logic           inta        ,
                output  logic [1:0]     nextPCsel   ,
                output  logic [3:0]     opSel       ,
                output  logic           memSel      ,
                output  logic           we          ,
                output  logic           dataRead    ,
                output  logic           dataWrite   ,

                input   logic           reset       ,
                input   logic           clk         );

    logic [5:0] opCode1     ;
    logic [4:0] destAddr1   ;
    logic [4:0] leftAddr1   ;
    logic [4:0] rightAddr1  ;
    logic [4:0] destAddr2   ;
    logic [5:0] opCode3     ;
    logic [4:0] destAddr3   ;
    logic       leftFwd     ;
    logic       rightFwd    ;

    assign opCode1      = instruction1[31:26]   ;
    assign leftAddr1    = instruction1[20:16]   ;
    assign rightAddr1   = instruction1[15:11]   ;
    assign destAddr2    = instruction2[25:21]   ;
    assign destAddr3    = instruction3[25:21]   ;
    assign opCode3      = instruction3[31:26]   ;


// Interrupt automaton
    logic [2:0] intState    ;
/*
    intState = 000: intDisable
```

```
    intState = 001: intEnable
    intState = 010: intExec1
    intState = 011: intExec2
    intState = 100: intExec3
    intState = 101: intExec4
*/
    always_ff @(posedge clk)
        if (reset)                                  intState <= 3'b000  ;
         else
            begin
            if (opCode1 == `eint)                   intState <= 3'b001  ;
            if (opCode1 == `dint)                   intState <= 3'b000  ;
            if (intIn && (intState == 3'b001))      intState <= 3'b010  ;
            if (intState == 3'b010)                 intState <= 3'b011  ;
            if (intState == 3'b011)                 intState <= 3'b100  ;
            if (intState == 3'b100)                 intState <= 3'b101  ;
            if (intState == 3'b101)                 intState <= 3'b000  ;
            end
    assign inta = intState == 3'b011    ;
// end Interrupt automaton

// Program control
    logic [5:0] jmpSel  ;

    assign jmpSel = (intState > 3'b011) ? `nop : opCode1    ;

    always_comb if (inta)         nextPCsel = 2'b11                 ;
                 else
                    case(jmpSel)
                        `halt   : nextPCsel = 2'b00                 ;
                        `rjmp   : nextPCsel = 2'b10                 ;
                        `zbr    : nextPCsel = zero ? 2'b10 : 2'b01  ;
                        `nzbr   : nextPCsel = zero ? 2'b01 : 2'b10  ;
                        `ret    : nextPCsel = 2'b11                 ;
                        default : nextPCsel = 2'b01                 ;
                    endcase
// end prorgram control

// Forwarding section
    assign leftFwdY  =
        (leftAddr1  == destAddr2) && (opCode1[5] == 1'b1);
    assign rightFwdY =
        (rightAddr1 == destAddr2) && (opCode1[5] == 1'b1);
    assign leftFwdO  =
        (leftAddr1  == destAddr3) && (opCode1[5] == 1'b1);
    assign rightFwdO =
        (rightAddr1 == destAddr3) && (opCode1[5] == 1'b1);
    always_comb
        case(opCode1)
            `addv   :   opSel[1:0] = 2'b11  ;
```

```
            `multv   :   opSel[1:0] = 2'b11  ;
            `addvc   :   opSel[1:0] = 2'b11  ;
            `val     :   opSel[1:0] = 2'b11  ;
            default  :   if (rightFwdY)          opSel[1:0] = 2'b01  ;
                          else if (rightFwdO)    opSel[1:0] = 2'b10  ;
                                else             opSel[1:0] = 2'b00  ;
        endcase
    always_comb
        if (leftFwdY)       opSel[3:2] = 2'b01  ;
         else if (leftFwdO) opSel[3:2] = 2'b10  ;
              else          opSel[3:2] = 2'b00  ;
// end forwarding section

    assign dataWrite    = opCode3 == `store                          ;
    assign dataRead     = opCode3 == `read                           ;
    assign memSel       = opCode3 == `read                           ;
    assign we           = ((opCode3[5:4] == 2'b11)  |
                           (opCode3 == `val)        |
                           (opCode3 == `read)) & !(intState > 3'b001);
endmodule
```

```
/*****************************************************************
File name: INSTRFETCH.sv
Circuit name:
Description:
*****************************************************************/
module INSTRFETCH ( input   logic [31:0]    instruction ,
                    input   logic [31:0]    leftOp      ,
                    input   logic [1:0]     nextPCsel   ,

                    output  logic [9:0]     pc          ,
                    output  logic [31:0]    instruction1,
                    output  logic [9:0]     pc1         ,

                    input   logic           reset       ,
                    input   logic           clk         );

    // PipeReg0
    always_ff @(posedge clk)
        if (reset)  begin       pc <= -1                        ;
                                instruction1 <= 0               ;
                    end
         else begin case(nextPCsel)
                        2'b00:  pc <= pc                        ;
                        2'b01:  pc <= pc + 1                    ;
                        2'b10:  pc <= instruction1[9:0] + pc1   ;
                        2'b11:  pc <= leftOp                    ;
                    endcase
```

```
    // PipeReg1
                        instruction1 <= instruction ;
                        pc1          <= pc          ;
                end
endmodule
```

```
/*****************************************************************
File name: OPSFETCH.sv
Circuit name:
Description:
*****************************************************************/
module OPSFETCH(    input   logic [31:0]    instruction1,
                    input   logic [31:0]    instruction4,
                    input   logic [9:0]     pc1         ,
                    input   logic           inta        ,
                    input   logic           we4         ,
                    input   logic [31:0]    result4     ,
                    input   logic [3:0]     opSel       ,

                    output  logic           zero        ,
                    output  logic [31:0]    leftOp      ,
                    output  logic [31:0]    leftOp2     ,
                    output  logic [31:0]    rightOp2    ,
                    output  logic [3:0]     opSel2      ,
                    output  logic [31:0]    instruction2,

                    input   logic           clk         );

    logic   [31:0]  rf[0:31]    ;
    logic   [31:0]  rightOp     ;
    logic           we          ;
    logic   [4:0]   destAddr    ;
    logic   [31:0]  result      ;
    logic   [4:0]   leftAddr    ;
    logic   [4:0]   rightAddr   ;


    always_ff @(posedge clk) if (we) rf[destAddr] <= result ;

    assign leftOp       =
        ((destAddr == leftAddr) && we) ? result : rf[leftAddr]    ;
    assign rightOp      =
        ((destAddr == rightAddr) && we) ? result : rf[rightAddr]  ;
    assign zero         = leftOp == 0           ;
    assign rightAddr    = instruction1[15:11]   ;

    // PipeReg2:
    always_ff @(posedge clk)
```

```
        begin
            leftOp2         <=  leftOp          ;
            rightOp2        <=  rightOp         ;
            opSel2          <=  opSel           ;
            instruction2    <=  instruction1    ;
        end

    intUnit intunit(.we         (we                     ),
                    .destAddr   (destAddr               ),
                    .result     (result                 ),
                    .leftAddr   (leftAddr               ),

                    .we4        (we4                    ),
                    .destAddr4  (instruction4[25:21]    ),
                    .result4    (result4                ),
                    .leftAddr1  (instruction1[20:16]    ),
                    .pc1        (pc1                    ),
                    .inta       (inta                   ));
endmodule
```

```
/*****************************************************************
File name: EXECUTE.sv
Circuit name:
Description:
*****************************************************************/
`include "0_DEFINES.vh"
module EXECUTE( input   logic [31:0]    leftOp2     ,
                input   logic [31:0]    rightOp2    ,
                input   logic [3:0]     opSel2      ,
                input   logic [31:0]    instruction2,
                input   logic [31:0]    result4     ,

                output  logic [31:0]    leftOp3     ,
                output  logic [31:0]    result3     ,
                output  logic [31:0]    rightOp3    ,
                output  logic [31:0]    instruction3,

                input                   clk         );

    logic [31:0]    rightIn ;
    logic [31:0]    leftIn  ;
    logic [31:0]    out     ;
    logic [31:0]    value   ;

    assign value = {{16{instruction2[15]}}, instruction2[15:0]} ;

    always_comb case(opSel2[1:0])
                    2'b00: rightIn = rightOp2   ;
```

```
                    2'b01: rightIn = result3    ;
                    2'b10: rightIn = result4    ;
                    2'b11: rightIn = value      ;
                endcase
    always_comb case(opSel2[3:2])
                    2'b00: leftIn = leftOp2;
                    2'b01: leftIn = result3;
                    2'b10: leftIn = result4;
                    2'b11: leftIn = 31'b0   ;
                endcase

    ALU alu(.out    (out                ),
            .leftIn (leftIn             ),
            .rightIn(rightIn            ),
            .opCode (instruction2[31:26]));

    // PipeReg3
    always_ff @(posedge clk)
        begin   leftOp3         <= leftIn       ;
                result3         <= out          ;
                rightOp3        <= rightIn      ;
                instruction3    <= instruction2 ;
        end
endmodule
```

```
/*************************************************************************
File name: DATAMEM.sv
Circuit name:
Description:
*************************************************************************/
module DATAMEM( input   logic [31:0]    result3     ,
                input   logic [31:0]    instruction3,
                input   logic           we          ,
                input   logic           memSel      ,
                input   logic [31:0]    dataIn      ,

                output  logic [31:0]    result4     ,
                output  logic [31:0]    instruction4,
                output  logic           we4         ,

                input   logic           clk         );

    always_ff @(posedge clk)
        begin   result4         <= memSel ? dataIn : result3;
                instruction4    <= instruction3             ;
                we4             <= we                       ;
        end
endmodule
```

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

    logic [32:0]    sum ;
    logic [32:0]    dif ;

    assign sum = leftIn + rightIn   ;
    assign dif = leftIn - rightIn   ;

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

```
/*************************************************************************
File name: intUnit.sv
Circuit name:
Description:
*************************************************************************/
```

```
module intUnit( output  logic           we          ,
                output  logic   [4:0]   destAddr    ,
                output  logic   [31:0]  result      ,
                output  logic   [4:0]   leftAddr    ,

                input   logic           we4         ,
                input   logic   [4:0]   destAddr4   ,
                input   logic   [31:0]  result4     ,
                input   logic   [4:0]   leftAddr1   ,
                input   logic   [9:0]   pc1         ,
                input   logic           inta        );

    assign we       = inta ? 1'b1           : we4       ;
    assign destAddr = inta ? 5'b11110       : destAddr4 ;
    assign result   = inta ? {21'b0, pc1}   : result4   ;
    assign leftAddr = inta ? 5'b11111       : leftAddr1 ;
endmodule
```

D.2 Code generator

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

    `include "0_DEFINES.vh"

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

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

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

        v = labelTab[labelIndex] - addrCounter  ;
        endtask

// CONTROL INSTRUCTIONS
    task NOP; // no operation
        begin   opCode  = `nop  ;
                {d,l,v} = 26'b0 ;
                endLine         ;
        end
    endtask

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

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

    task BRZ; // branch if zero
        input   [4:0]   left    ;
        input   [15:0]  label   ;

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

    task BRNZ; // branch if not zero
        input   [4:0]   left    ;
        input   [15:0]  label   ;

        begin   opCode  = `nzbr ;
```

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

    task RET; // return from subroutine
        input   [4:0] left  ;

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

    task HALT; // halt running
        begin   opCode  = `halt ;
                {d,l,v} = 26'b0 ;
                endLine         ;
        end
    endtask

    task EI; // enable interrupt
        begin   opCode  = `eint ;
                {d,l,v} = 26'b0 ;
                endLine         ;
        end
    endtask

    task DI; // disable interrupt
        begin   opCode  = `dint ;
                {d,l,v} = 26'b0 ;
                endLine         ;
        end
    endtask

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

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

```
    end
endtask

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

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

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

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

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

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

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

    begin   opCode  = `multv;
            d       = dest  ;
            l       = left  ;
```

```
            v       = value ;
            endLine         ;
    end
endtask

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

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

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

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

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

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

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

    begin   opCode  = `lsh  ;
            d       = dest  ;
```

```
            l       = left  ;
            v       = 16'b0 ;
            endLine         ;
    end
endtask

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

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

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

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

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

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

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

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

```
                endLine         ;
        end
    endtask

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

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

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

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

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

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

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

        begin   opCode  = `read ;
                d       = dest  ;
```

```
            l       = left  ;
            v       = 16'b0 ;
            endLine         ;
        end
    endtask

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

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

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

        begin   opCode  = `val  ;
                d       = dest  ;
                l       = 5'b0  ;
                v       = value ;
                endLine         ;
        end
    endtask

    // RUNNING
    initial begin   addrCounter = 0;
                    `include "0_program.sv" // first pass
                    addrCounter = 0;
                    `include "0_program.sv" // second pass
            end
```

D.3 Simulator

```
/*****************************************************************
File name: testRISC.sv
Circuit name:
Description:
*****************************************************************/
`include "0_DEFINES.vh"
module testRISC;
```

```
    logic   inta    ;
    logic   intIn   ;
    logic   reset   ;
    logic   clk     ;

    integer i   ;

    `include "0_RISCcodeGenerator.sv"

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

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

    toyRISCsystem dut(  inta    ,
                        intIn   ,
                        reset   ,
                        clk     );

    // MONITOR FOR PROGRAM LAOD & CONTROLLER
    initial begin
     $monitor("t=%0d pc=%d  RF=[%0d, %0d, %0d, %0d, %0d, %0d, %0d, %0d]
              dataWrite=%0b DM=[%0d, %0d, %0d, %0d] intState=%b inta=%b",
                $time,
                dut.processor.pc,
                dut.processor.regFile.rf[0],
                dut.processor.regFile.rf[1],
                dut.processor.regFile.rf[2],
                dut.processor.regFile.rf[3],
                dut.processor.regFile.rf[4],
                dut.processor.regFile.rf[5],
                dut.processor.regFile.rf[30],
                dut.processor.regFile.rf[31],
                dut.processor.dataWrite,
                dut.memSys.dataMemory[0],
                dut.memSys.dataMemory[1],
                dut.memSys.dataMemory[2],
                dut.memSys.dataMemory[3],
                dut.processor.dcd.intState,
                dut.processor.dcd.inta);
```

```
    end
endmodule
```

D.4 Testing

```
/*************************************************************************
File name: program.sv
Circuit name:
Description:
*************************************************************************/
/*
            NOP             ;
            VAL(0,1)        ;
            VAL(1,2)        ;
            VAL(2,3)        ;
            VAL(3,4)        ;
            VAL(4,5)        ;
            //NOP               ;
            //NOP               ;
            //NOP               ;
            ADD(5,4,3)      ;
            HALT            ;
            HALT            ;
//*/


/*
            VAL(0,2)        ;
            VAL(1,4)        ;
            VAL(2,8)        ;
            MULT(3,1,2)     ;
            ADDV(0,0,5)     ;
            NOP             ;
            HALT            ;
            HALT            ;
//*/
/*
            VAL(31,13)      ;
            VAL(2,23)       ;
            VAL(0,24)       ;
            VAL(1,13)       ;
            EI              ;
            ADDV(2,31,1);
            VAL(1,1)        ;
            VAL(2,222)      ;
            NOP             ;
```

```
                ADD(0,1,2)    ;
                HALT          ;
                HALT          ;
                NOP           ;
          // subroutine triggered by interrupt
                ADDV(30,30,-2)   ;
                VAL(3,44)        ;
                NOP              ;
                NOP              ;
                NOP              ;
                RET(30)          ;
                NOP              ;
//*/
//*
                NOP           ;
                VAL(0,5)      ;
     LB(1);     ADDV(0,0,-1);
                NOP           ;
                NOP           ;
                BRNZ(0,1)     ;
                NOP           ;
                HALT          ;
                HALT          ;
//*/
/*
                VAL(1,55)     ;
                VAL(0,1)      ;
                STORE(0,1)    ;
                NOP           ;
                NOP           ;
                READ(2,0)     ;
                HALT          ;
                HALT          ;
//*/
```

Bibliography

[Amdahl '64] G. M. Amdahl, G. A. Blaauw, F. P. Brooks, Jr. (1964) Architecture of the IBM System/360, *IBM Journal of Research and Development*, **8**(2):87–101.

[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

[Asanovic '06] K. Asanovic, *et al.* (2006) *The landscape of parallel computing research: A view from Berkeley*, Technical Report EECS-2006-183, Berkeley University, Berkeley, CA. At: `https://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf`

[Asanovic '09] K. Asanovic, *et al.* (2009) *A View of the Parallel Computing Landscape*, Communications of the ACM, **52**(10):56–67.

[Backus '78] J. Backus (1978) Can programming be liberated from the von neumann style? a functional style and its algebra of programs, *Communications of the ACM*, **21**(8):613–641.

[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.

[Casti '92] John L. Casti: *Reality Rules: II. Picturing the World in Mathematics - The Frontier*, John Wiley & Sons, Inc., 1992.

[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.

[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.

[Cormen '90] Thomas H. Cormen, Charles E. Leiserson, Donsld R. Rivest: *Introduction to Algorithms*, MIT Press, 1990.

[Dijkstra '65] E. W. Dijkstra (1965) Cooperating sequential processes. Technical Report EWD-123, Eindhoven University of Technology, Netherlands. Reprinted in *Programming Languages* Academic Press, New York, 43–112. At: https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html

[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.

[Fortune '78] S. Fortune, J. C. Wyllie (1978) Parallelism in random access machines. *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, 114–118.

[Goldschlage '82] L. M. Goldschlage (1982) A universal interconnection pattern for parallel computers. *Journal of the ACM* **29**(4):1073–1086.

[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. Available at: http://www.eurOpment.org/library/2014/santorini/bypaper/COMPUTERS/COMPUTERS2-15.pdf

[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.

[Hennessy '07] John L. Hennessy, David A. Patterson: *Computer Architecture: A Quantitative Approach*, Fourth Edition, Morgan Kaufmann, 2007.

[Hennessy '19] J. L. Hennessy, D. A. Patterson (2019) *Computer Adchitecture. A Quantitative Approach. Sixth Edition*, Morgan Kaufmann.

[Hill & Reddi '21] Mark D. Hill, Vijay Janapa Reddi: "Accelerator-Level Parallelism", *Communications of the ACM*, **64**(12):36-38, 2021.

[Hillis '85] W. D. Hillis: *The Connection Machine*, The MIT Press, Cambridge, Mass., 1985.

[Hilbert 1900] D. Hilbert, *Mathematical Problems. Lecture Delivered Before the International Congress of Mathematicians at Paris in 1900*. Available at: https://www.ams.org/journals/bull/1902-08-10/S0002-9904-1902-00923-3/S0002-9904-1902-00923-3.pdf

[Hilbert & Ackermann '28] David Hilbert, Wilhelm Ackermann (1928). *Grundzüge der theoretischen Logik* (Principles of Mathematical Logic). Springer-Verlag.

[Jouppi '17] Jouppi, N. P., Young, C., Patil, N., Patterson, D. *et al.* (2017) In-Datacenter Performance Analysis of a Tensor Processing Unit^{TM}, *44th International Symposium on Computer Architecture (ISCA)*, Toronto, Canada, June 26. At: https://drive.google.com/file/d/0Bx4hafXDDq2EMzRNcy1vSUxtcEk/view

[Kleene '36] Stephen C. Kleene: "General Recursive Functions of Natural Numbers", in *Math. Ann*., 112, 1936.

[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.

[Maliţa '13] Mihaela Maliţa, Gheorghe M. Ştefan: "Control Global Loops in Self-Organizing Systems", *ROMJIST*, Volume 16, Numbers 2–3, 2013, 177-191.
http://www.imt.ro/romjist/Volum16/Number16_2/pdf/05-Malita-Stefan2.pdf

[Mead '79] Carver Mead, Lynn Convay: *Introduction to VLSI Systems*, Addison-Wesley Pub, 1979.

[Milutinovic 89'] Veljko M. Milutinovic (ed.): *High-Level Language Computer Architecture*, Computer Science Press, 1989.

[Moore '11] Chuck Moore (2011) Data Processing in Exascale-class Computing Systems, *The Salishan Conference on High Speed Computing* `https://www.lanl.gov/conferences/salishan/salishan2011/3moore.pdf`

[Moore '65] G. E. Moore (1965) Cramming more components onto integrated circuits, *Electronics*, **38**(8):114-117.

[Moore '75] G. E. Moore (1975) Progress In Digital Integrated Electronics, *IEEE, International Electron Devices Meeting. Technical Digest* pp.11-13.

[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.

[Parberry 87] Ian Parberry: *Parallel Complexity Theory*. Research Notes in Theoretical Computer science. Pitman Publishing, London, 1987.

[Patterson '10] David A. Patterson (2010) The trouble with multicore. *IEEE Spectrum*, **47**(7):28-32.

[Patterson '97] David Patterson, et all, A case for intelligent RAM (1997) *IEEE Micro* **17**(2)34:44.

[Patterson '05] David A. Patterson, John L.Hennessy: *Computer Organization & Design. The Hardware / Software Interface*, Third Edition, Morgan Kaufmann, 2005.

[Patterson '19] David A. Patterson, John L.Hennessy: *Computer Organization & Design. The Hardware / Software Interface*, Sixth Edition, Morgan Kaufmann, 2019.

[Post '36] Emil Post: "Finite Combinatory Processes. Formulation I", in*The Journal of Symbolic Logic*, vol. 1, p. 103 -105, 1936.

[Pratt '74] V. R. Pratt, M. O. Rabin, L. J. Stockmeyer (1974) A characterization of the power of vector machines. *Proceedings of STOC'1974*, pp. 122-134.

[Raina '16] G. Raina (2016) Deep Convolutional Netvork evaluation on the Xeon Phi: Where Subword Parallelism meets Many-Core, *Eindhoven University of Technology*, Master Thesis. At: `https://repository.tue.nl/844256`

[Redmon '16] J. Redmon, S. Divvala, R. Girshick, A. Farhadi (2016) You only look once: Unified, real-time object detection, *Cornell Univ. Library*.

[Savage '87] John Savage: *The Complexity of Computing*, Robert E. Krieger Pub. Comp., 1987.

[Shannon '38] C. E. Shannon: "A Symbolic Analysis of Relay and Switching Circuits", *Trans. American Institute of Electrical Engineers*, **57**(12):713-723, 1938. (The paper is an abstract of the thesis presented in 1937 at MIT for the degree of master in science.)

[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.

[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.

[Streinu '85] Ileana Streinu: "Explicit Computation of an Independent Gödel Sentence", in *Recursive Functions Theory Newsletter*, June 1985.

[Stefan '20] G. M. Ştefan (2020) *Composition is the only independent rule in Kleene's model of partial recursive functions*, At: `http://users.dcae.pub.ro/~gstefan/2ndLevel/composition.html`

[Ştefan '91] Gheorghe Ştefan: *Funcție și structură în sistemele digitale* (Function and Structure in Digital Systems), Ed. Academiei Române, 1991.

[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 '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 '06a] Gheorghe Ştefan: "A Universal Turing Machine with Zero Internal States", in *Romanian Journal of Information Science and Technology*, **9**(3):227-243, 2006.

[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 Circuits, 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`

[Ştefan '21a] Gheorghe M. Stefan: "Let's consider Moore's law in its entirety", CAS 2021 PROCEEDINGS : 2021 International Semiconductor Conference : 44nd edition, , pp. 3-10, October 6-8, Sinaia, Romania.

[Ştefan '21] Gheorghe M. Stefan: "Pseudo-Reconfigurable Systems", ROMJIST, **24**(12):366-383, 2021.

[Turing '36] Alan M. Turing: "On computable Numbers with an Application to the Eintscheidungsproblem", in *Proc. London Mathematical Society,* 42 (1936), 43 (1937).

[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.

[Waksman '68] Abraham Waksman, "A permutation network," in *J. Ass. Comput. Mach.*, vol. 15, pp. 159-163, Jan. 1968.

[webRef_3] `http://www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html#fsa_pfx`

[Zurada '95] Jacek M. Zurada: *Introductin to Artificial Neural network*, PWS Pub. Company, 1995.