UE719

EMBEDDED SYSTEMS

LAMRI NEHAOUA

LAMRI.NEHAOUA@UNIV-EVRY.FR

UNIVERSITÉ D'EVRY PARIS-SACLAY

Table of Contents

Chapter 1

Design methodology

1 Introduction

Digital systems are ubiquitous in our everyday life. A digital system processes information in binary form. However, the information coming from the environment is in general in analog form. Sampling and quantification methods allows to make this information usable by different processors. In the literature, there are two major classifications for digital systems:

1. *General Purpose Systems*: combination of hardware and software for public/private use.

2. *Embedded systems*: generally implemented into an another system where the name of *embedded*. They are designed to perform specific functions.

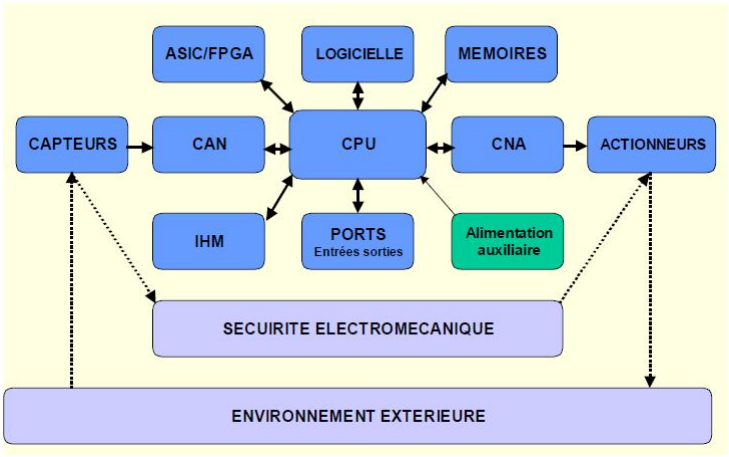


Figure 1.1: General block diagram of a digital system.

According to figure 1.1, the hardware part of a digital system can be broken down into two main parts: processor and peripherals. The processor manages the processing of information, while the peripherals allow interaction with the outside world via the various inputs and outputs or, to provide the processor with additional capacities to accomplish its task via the memory blocks and co-processors. On the other hand, the software

part is also important. The software part includes all the tools necessary for application development: programming languages, compilers, debugging tools, emulators, and operating system.

Unlike general-purpose systems, an embedded systems system must meet more constraints as time scheduling. Therefore, functions and tasks are handled by a specific operating system designed around a real-time kernel, *RTOS* and, in this case, the system is said to be real-time.

1.1 Processor

A term processor is a generic name of a processing unit whose purpose is to process the input information in a sequential manner by a succession of simple operations and instructions. There are two main types of processors: general purpose and special purpose. *general purpose* processors can be programmed and are therefore very versatile. The memory associated with the processor contains a series of instructions to be executed at each clock cycle. This type of processor is often referred to by the name **CPU**, *Central Processing Unit*. These processors are more suitable for applications without constraints of resources, memory, or autonomy. On the other hand, *specific use* processors are *CPU* to which, additional components are added for a specific application such as signal or image processing (figure 1.2).

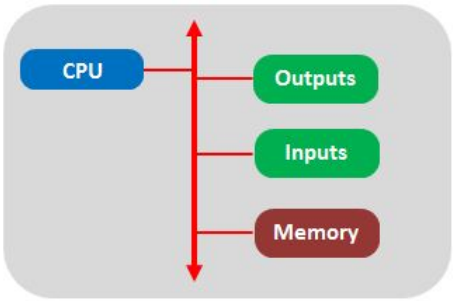


Figure 1.2: The simplest model of a digital system. A CPU for processing data, memory for storing data and/or instructions, and a set of I/O ports.

An example of special purpose processors are **DSP**, *Digital Signal Processor*, **FPGA**, *Field-Programmable Gate Array*, **ASIC**, *Application-Specific Integrated Circuit*, etc. More recently, and to meet more and more constraints of integration and complexity, the use of **SoC**, *System on-Chip*, has become a standard in the design of digital systems . Figure 1.3 shows the motherboard of a Samsung Galaxy S8 smartphone and figure 1.4 shows a Qualcomm Snapdragon 835 SoC chip developed by Samsung for these smartphones. It should be noted that to implement a given function, a general purpose processor uses more resources than a specific purpose processor because the latter uses tailor-made logic.

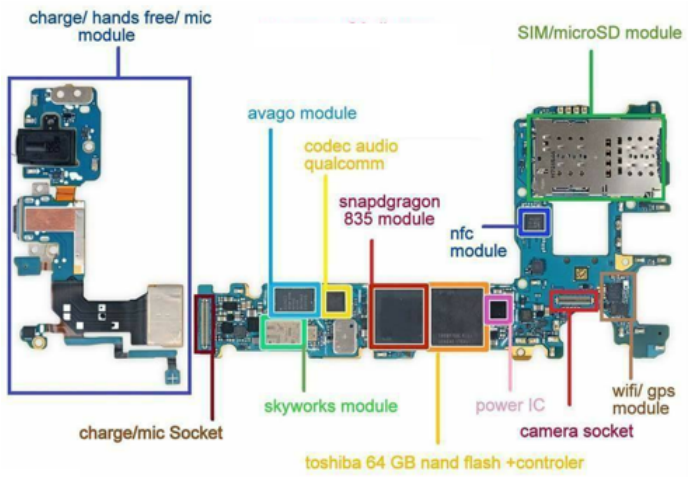


Figure 1.3: The motherboard of a Samsung Galaxy S8 smartphone.

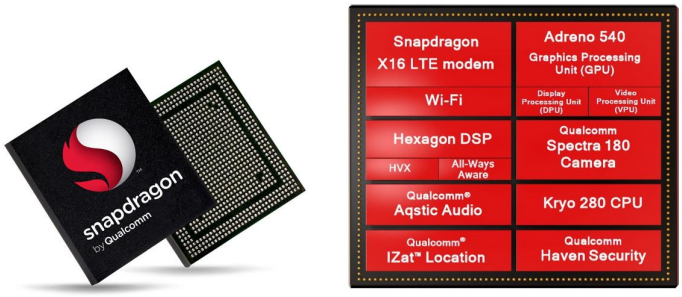


Figure 1.4: SoC Qualcomm Snapdragon 835: more than 3 billion transistors. It contains a GPU, DSP, modem LTE, audio codec, camera ISP and a CPU Qualcomm Kryo 280 (8 cores, 4 x 2.45 GHz with 2MB cache L2 + 4 x 1.9 GHz with 1MB cache L2).

A CPU can be broken down into two main parts: the *datapath* and the *control unit*. The datapath includes registers and functional units, such as **ALU**, *Arithmetic and Logic Unit*, as well as several multiplexers for transferring and manipulating data. Figure 1.5 represents the basic datapath of a microcontroller of the AVR family. The datapath receives data, performs calculations, and produces results. It also receives signals from the control unit indicating the operations to be carried out and the blocks to be activated. The datapath transmits status signals (*status*) to the control unit, indicating the status of the operations performed.

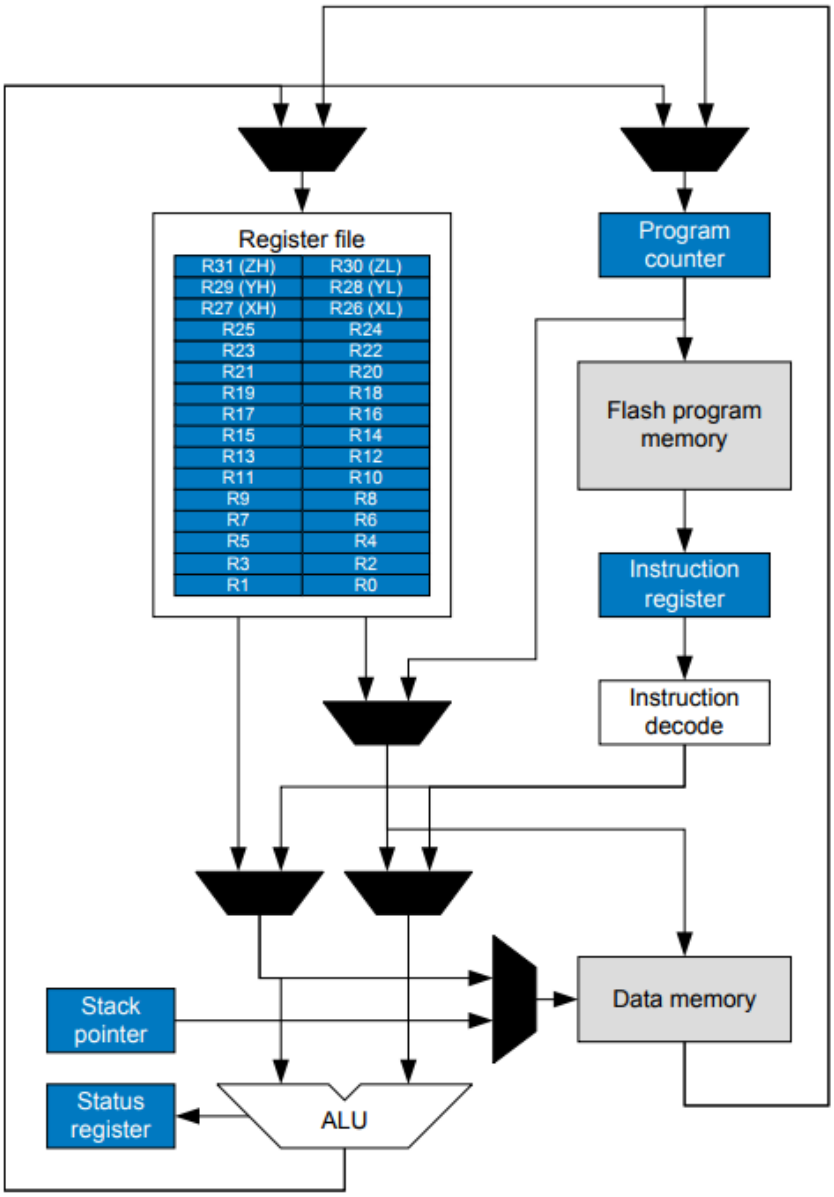


Figure 1.5: Datapath of an AVR architecture.

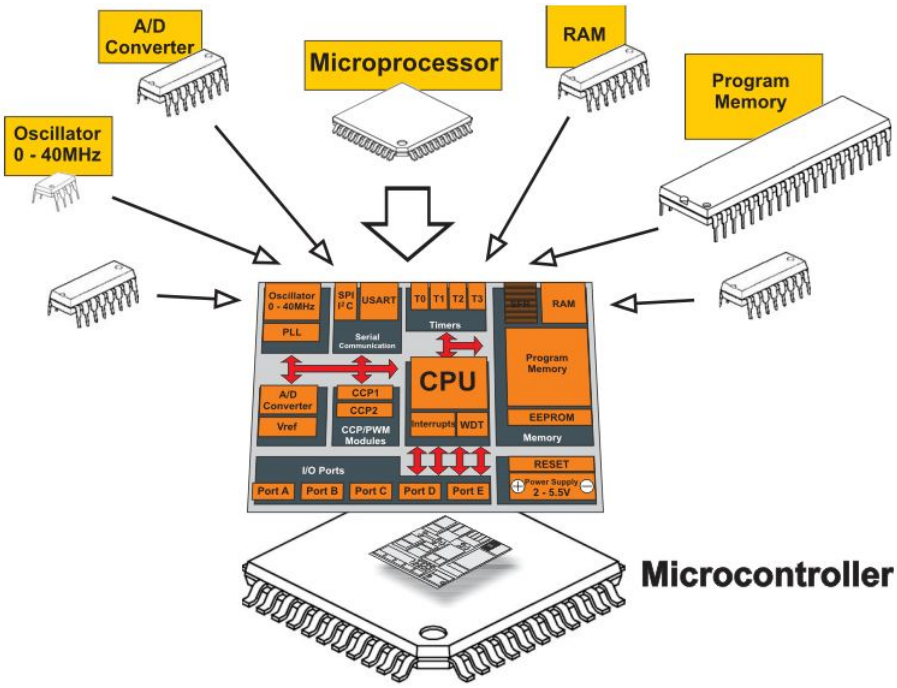


Figure 1.6: And what about the μC? A μC combines CPU, ROM, RAM, and I/O ports on the same chip. Other components can also be integrated: timers, ADC, communication interfaces, etc.

1.2 Semiconductor market

The global semiconductor market will be 655.6 B$ in 2025 compared to 342.7 B$ in 2015 with a CAGR of 6.7% (Compound Annual Growth Rate).

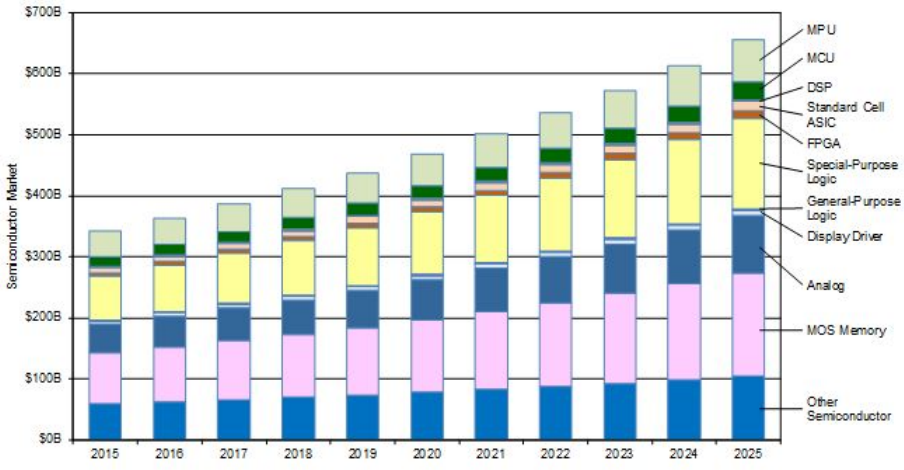


Figure 1.7: The global semiconductor market.

While the overall semiconductor market growth will be lower than in the past, several areas will experience significantly higher growth than the overall semiconductor market. For example, the semiconductor and sensor markets for **IoT**, *Internet of Things*, are expected to reach 114.2 B$ in 2025, up from 27.6B$ in 2015, with a CAGR of 15.3%. Major semiconductors in IoT applications include controllers, wireless connectivity, and

built-in non-volatile memory. Also, the IoT market consists of many segments, including commerce, automotive, medical, logistics and home automation, which have high growth potential, but with the need for very low power consumption energy for mobile devices.

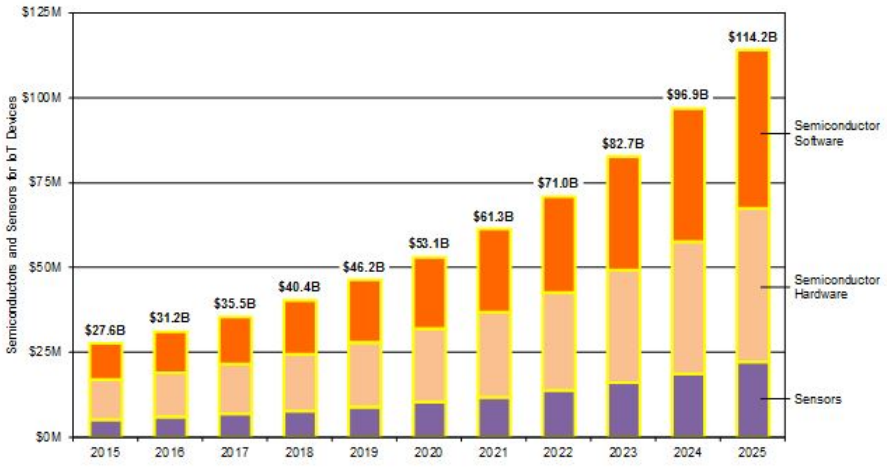


Figure 1.8: Growing demand for semiconductors in the Internet of Things.

Nonetheless, the semiconductor industry's margins are under pressure. For example, companies that produced three to five 65nm chips per year could effectively use their operations teams. However, the move to 40nm and below has dramatically changed the economics model. A 28nm *tape-out* [1] takes 78% of design time in addition and 40 % non-recurring investments [2] more than a 40nm tape-out.

Despite these issues, progress is being made whether with tools, methodologies or platform approaches that rely heavily on reuse. While projections show that it will cost up to 300 B$ to develop new SoCs (figure 1.9), the actual costs are generally much lower, provided that there are a lot of reusable IP products.

[1] In electronic design, tape-out is the end result of the design process of integrated circuits before they are sent for manufacture.

[2] Refers to the one-time costs of research, design, development and test of a new product, Wikipedia.

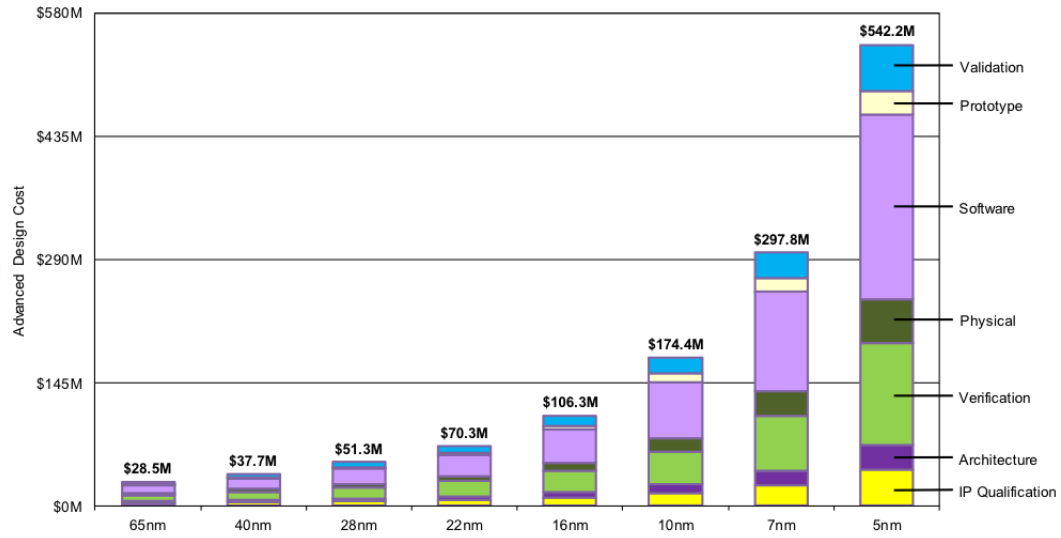


Figure 1.9: Development cost. The design costs of integrated circuits have gone from 51.3 M$ for 28nm chip to 297.8 m$ for 7nm and 542.2 M$ 5nm chip.

2 Wired logic

The development of digital systems through wired logic was widely used until the mid-1980s. Wired logic allows a wide assortment of integrated circuits (**IC**) to be connected, each containing only a few logic gates, performing a single logic function. This solution results in fixed digital systems without any possible evolution or flexibility in the design. An example of the most basic logic gate is the NOT gate, shown in figure 1.10.

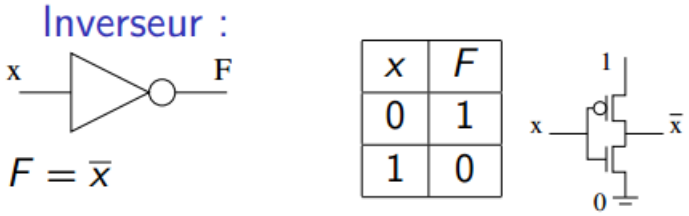


Figure 1.10: Logic NOT gate.

Among the most widely used IC are those of the 74 series. Figure 1.11 shows the integrated circuit 7404, which includes six *NOT* gates, in the form of a DIP package, *dual-inline package*. For each circuit in the 7400 series, several variants are built with different technologies. The 74LS00 variant is built using **TTL**, *transistor-transistor level* technology, while the 74HC00 variant is built using **CMOS**, *complementary metal oxide semiconductor*.

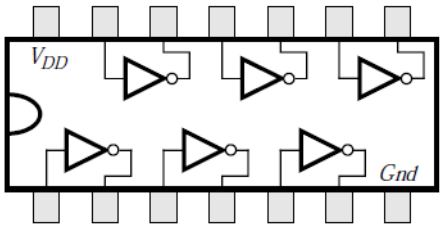


Figure 1.11: 7404 IC: two pins are used to connect to *VDD* and *Gnd*. The other pins provide a connection to the *NOT* gates.

Figure 1.12 shows an example of wiring three logic IC to implement the function $f = x_1x_2 + \bar{x}_2x_3$.

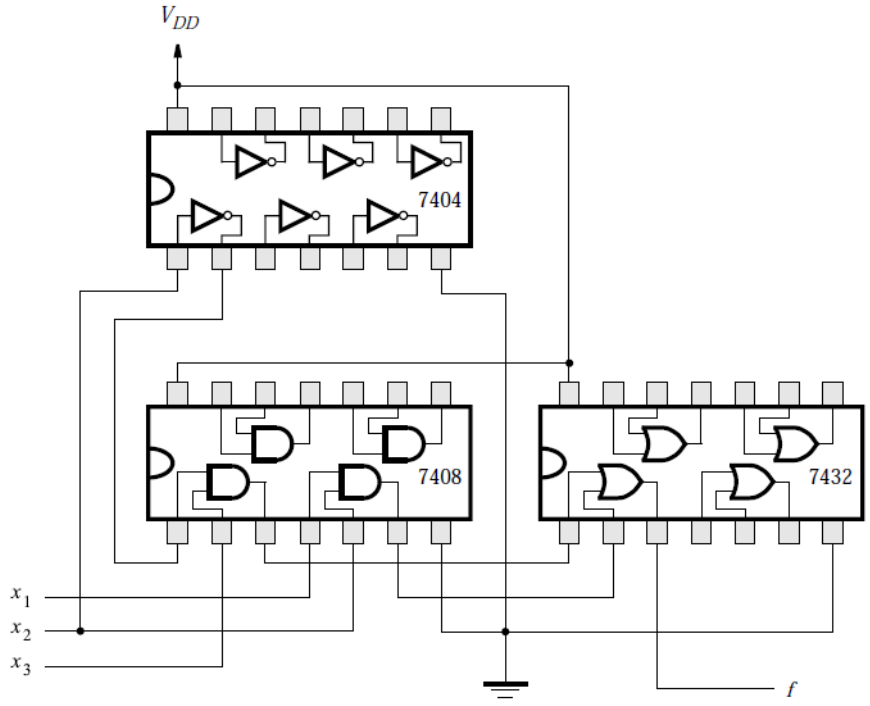


Figure 1.12: Example of a wired logic circuit: fonction $f = x_1x_2 + \bar{x}_2x_3$ is implemented by using 3 IC, 7404, 7408 and 7432.

3 Programmable logic

The function provided by each of the 7400 series circuits is fixed and cannot be adapted to all possible design situations. In addition, each circuit contains only a few logic gates, making these circuits inefficient for the realization of large digital systems. It was necessary to manufacture circuits with a high density of logic gates having a more flexible structure. These new circuits were introduced in the 1970s and are called **PLD**, *programmable logic devices*. Therefore, a PLD is a generic name of the programmable devices used in the realization of logic circuits. A PLD contains a collection of logic gate elements and programmable switches that can be customized in different ways.

Programmable logic					
SPLD				CPLD	FPGA
PLA	PLD	PAL	GAL		
combinational sequential	combinational	PLA 1 plan	micro-cell	complex GAL	LUT
PROM	PROM	PROM	EEPROM	EEPROM	DRAM/SRAM

Table 1.1: The different derivations of the PLD.

Several types of PLD are commercially available as shown in table 1.1. The first PLD marketed was the **PLA**, *Programmable Logic Arrays*, introduced in the 1970s. The general structure of a PLA is illustrated in figure 1.13. The basic idea comes from the fact that logical functions can be realized as a sum of products. PLAs are networks of unassigned gates, traversing a plane of *AND* gates which in turn feeds a set of *OR* gates. The configuration of the networks is determined by programming each interconnection. So the PLA inputs, namely, $x_1, \cdots, x_n$ go through a set of *buffer*, which provide both the value x_i and its complement $\bar{x}_i$, to a circuit block called *AND* plane. The latter produces a set of product terms $P_1, \cdots, P_k$ where each of these terms can be configured to implement a *AND* function of $x_1, \cdots, x_n$. The product terms serve as inputs to a *OR* plan, which produces the outputs $f_1, \cdots, f_m$. Each output can be configured to achieve any sum of P_1, $cdots, P_k$.

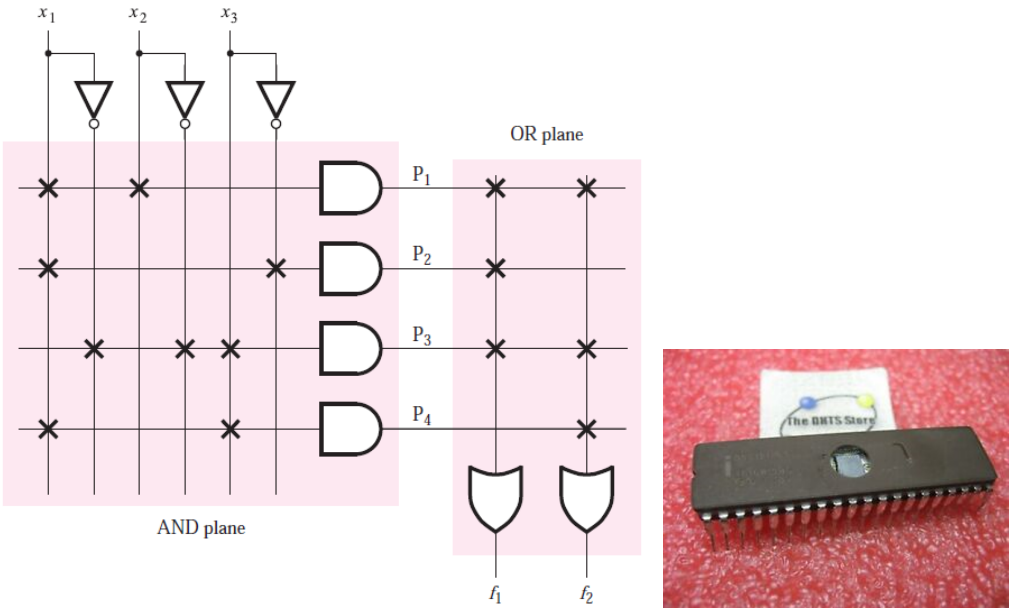


Figure 1.13: PLA: programmable logic array where each point of intersection is built by a fuse. The *AND* and *OR* plans are programmable where unwanted connections can be removed by blowing the corresponding fuses. Some versions of PROM-based PLA are programmed by ultraviolet exposure. The first output is: $f_1 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + x_1x_3$.

Historically, programmable switches (fuses) presented two difficulties for manufacturers: they were difficult to manufacture correctly and reduced the speed performance

of the implemented circuit. These drawbacks led, towards the end of the 1970s, to the development of a similar circuit in which the *AND* plane is programmable, but the *OR* plane is fixed, as shown in figure 1.14. Such a circuit is called **PAL**, *Programmable Array Logic*.

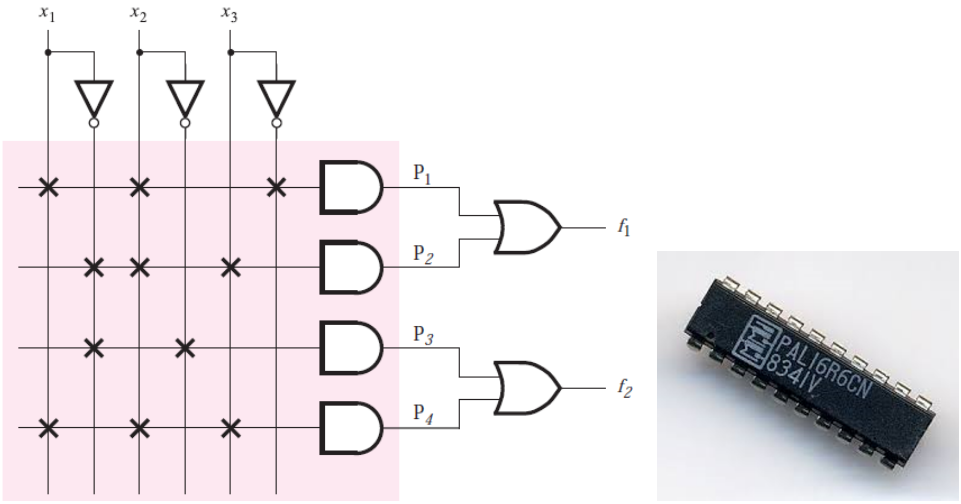


Figure 1.14: PAL: programmable array logic. Outputs are: $f_1 = x_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 x_3$ and $f_2 = \bar{x}_1 \bar{x}_2 + x_1 x_2 x_3$. Usually, a PAL can be programmed electrically only once and then the PROM can no longer be programmed.

In many PALs, additional circuitry has been added to the output of each *OR* gate in order to provide additional flexibility (figure 1.15). The *OR* gate combined with the extra circuit is often referred to as **macro-cell**.

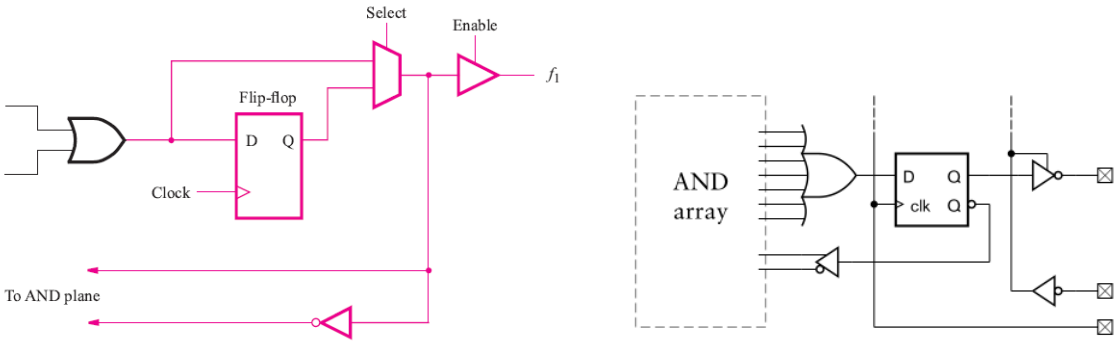


Figure 1.15: Example of a macro-cell.

The **GAL**, *Generic Array Logic*, introduced in the early 80's. Unlike PAL and PLA which use PROM memories, programmable only once, the *GAL* uses EEPROM. Figure 1.16 shows a GAL22V10 with 10 OLMCs with AND planes of varying sizes. For example, the two OLMCs associated with pins 14 and 23 have an AND plan of 8 terms. Each of the macro-cells has two main functional modes: combinatorial and register (figure 1.17). The mode is selected by the *select* inputs of the 4-input multiplexer. The latter also allows to define the logical polarity of the output signal at the pin level as being *active high* or *active low*. In combinatorial mode, the pin associated with an OLMC is driven by the

output of the OR gate. The tri-state gate allows a pin to be defined as dynamic input, output, or I/O. In register mode, the output pin associated with an OLMC is controlled by the Q output of the D flip-flop.

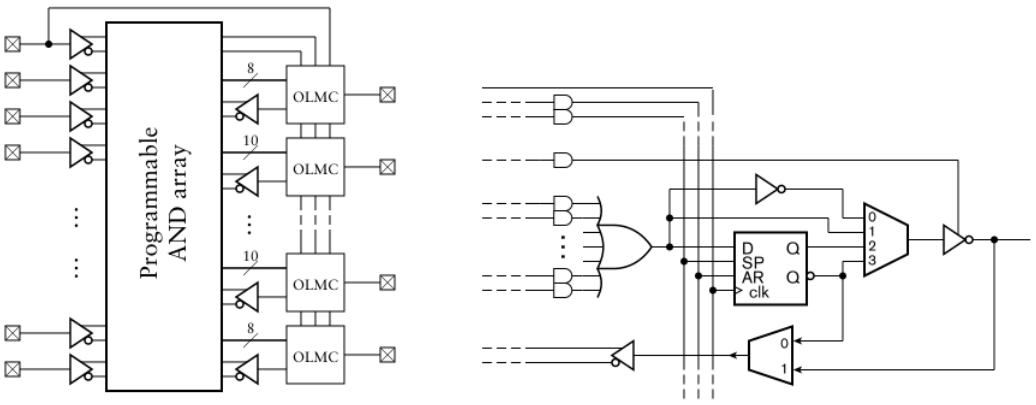


Figure 1.16: Lattice GAL22V10 uses OLMC, Output Logic Macro-cell.

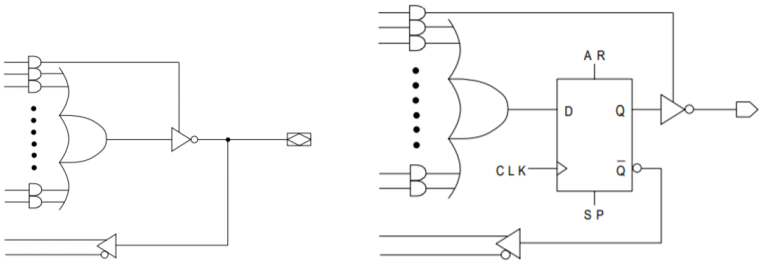


Figure 1.17: Combinational and register mdoes.

PLAs and PALs are useful for implementing a wide variety of small digital circuits. These circuits are limited to fairly modest sizes, generally supporting a combined number of inputs/outputs not exceeding 32. For the implementation of circuits requiring more inputs and outputs, a more dense circuit type can be used called **CPLD**, *Complex PLD*, represented in figure 1.18.

Figure 1.18: CPLD EPM7128S MAX 7000 from Altera includes CMOS EEPROM cells, 2500 logic gates, 128 macr-ocells, 8 LAB and 100 I/O. Each LAB is formed by 16 macro-cells.

CPLDs, introduced in mid-1980s, include multiple circuit blocks in the same chip, with internal wiring resources to connect the circuit blocks (figure 1.19). Each circuit block is similar to a PAL or PLA with a programmable AND plane and each macro-cell has a fixed OR plane and a configurable register with clock, enable, clear and preset functions.

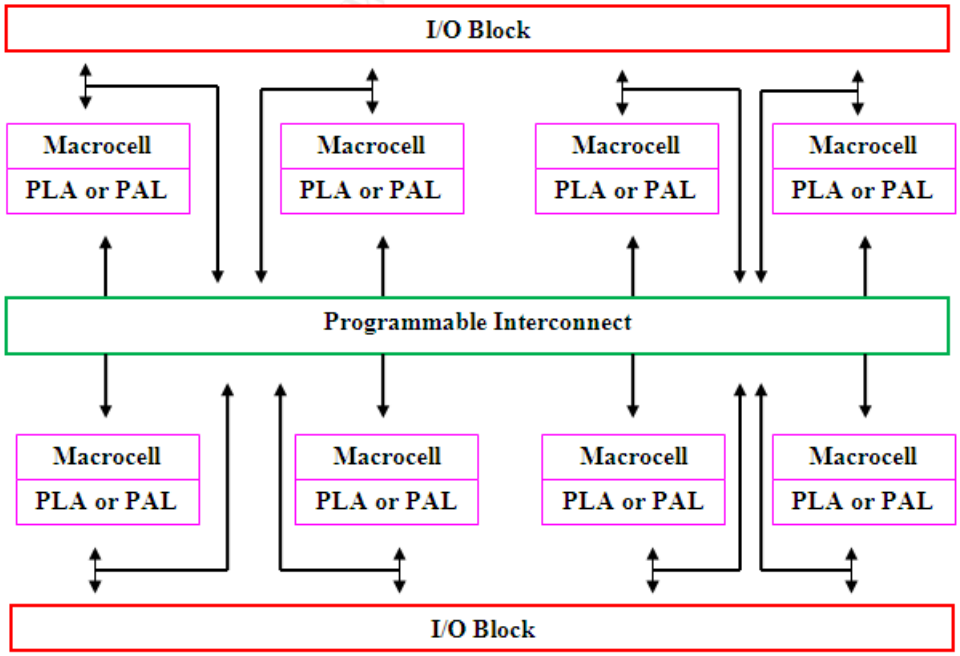


Figure 1.19: CPLD: complex programmable logic device.

In CPLDs, macrocells are combined into groups called **LAB**, *Logic Array Blocks* as shown in figure 1.20. Several LABs are linked together by a **PIA**, *Programmable Interconnect Array* representing a global bus powered by all dedicated inputs, I/O pins and macro-cells. To create complex logical functions, each macro-cell can be supplemented with **shareable expander product terms** and **high-speed parallel expander product terms** to provide up to 32 product terms per macro-cell.

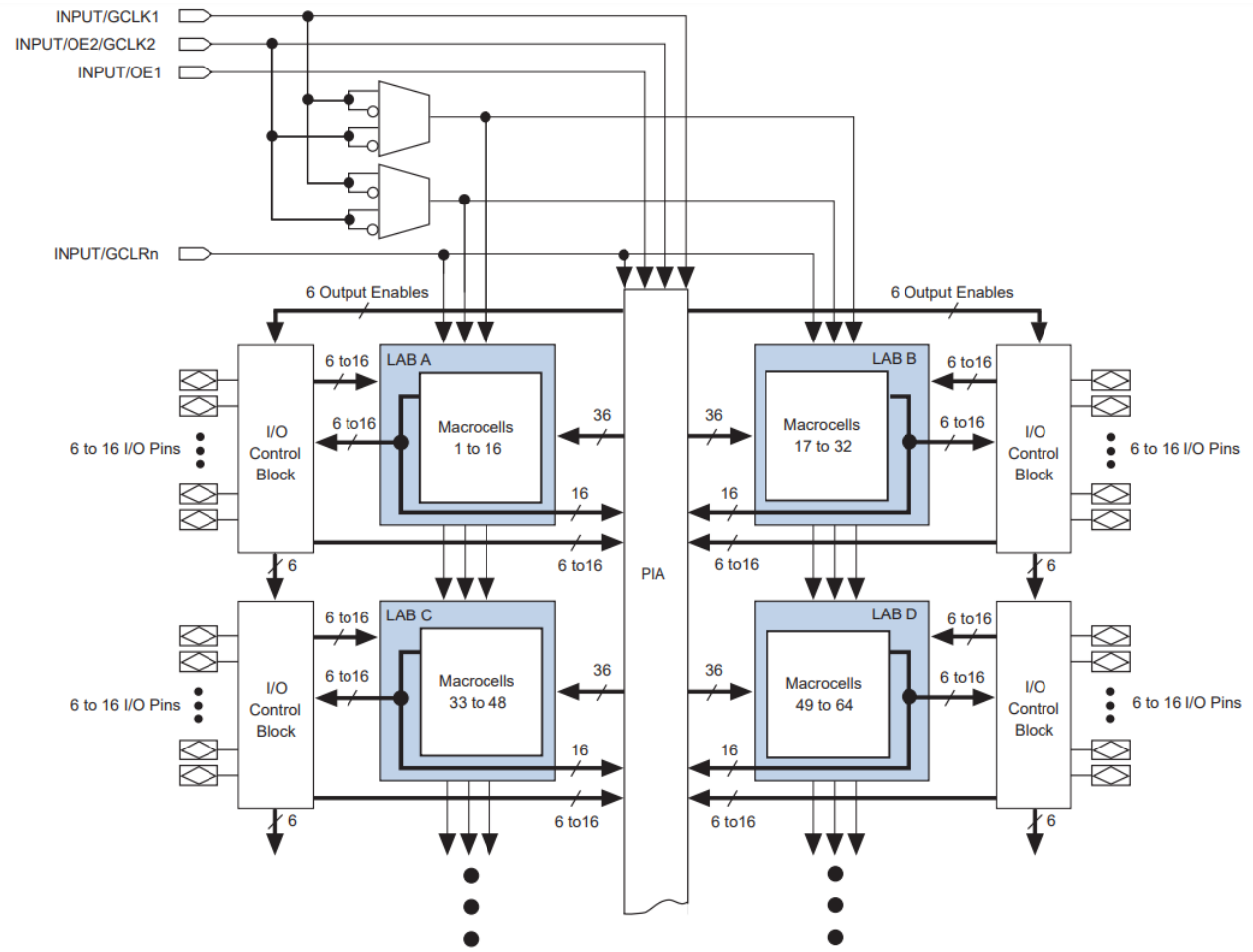


Figure 1.20: CPLD: macro-cell, LAB and PIA.

The macro-cell can be individually configured for sequential or combinational logic operation. The macro-cell consists of three functional blocks: logic array, product-term select matrix, and programmable register (figure 1.21).

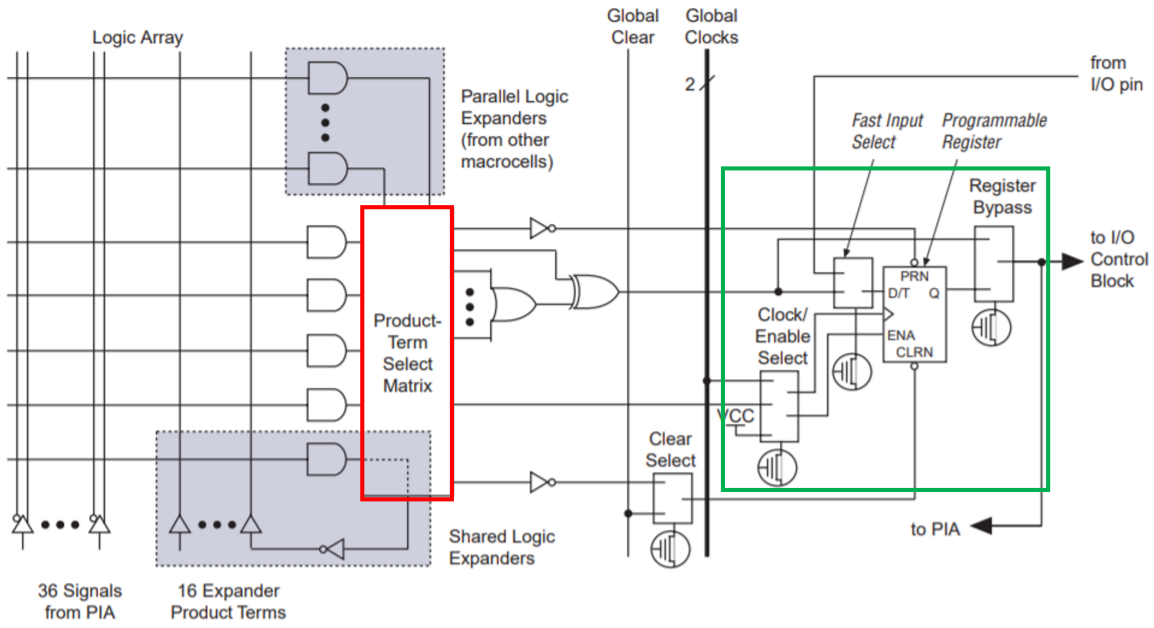


Figure 1.21: Zoom inside a macrocell within a CPLD.

Logic array is used to implement combinational functions. **Product-term select matrix** allows directing the product terms either as primary logic inputs (at OR and XOR gates) to implement combinational functions, or as secondary inputs to register control functions (clear, preset and enable). **Shareable expander product terms** are inverted product terms and are fed back into the logic array (figure 1.22). **Parallel expanders** are product terms borrowed from adjacent macro-cells. The development software automatically optimizes the allocation of product terms based on design requirements.

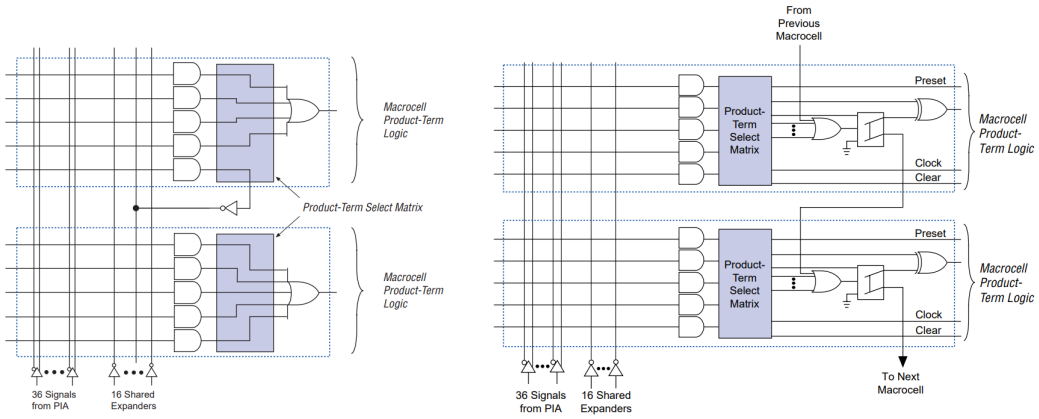


Figure 1.22: Shareable Expanders and Parallel Expanders.

Even for CPLDs, only moderately sized logic circuits can be integrated into a single chip. A simple method to quantify the size of a circuit to be developed is by using simple logic gates. A commonly used metric is the total number of two-input NAND gates that would be required to build the system, this metric is often referred to as the number of equivalent gates. By using this metric, the size of a 7400 IC is simple to measure because each chip contains only single gates. For PLDs and CPLDs, the typical metric used is that

each macro-cell represents about 20 equivalent gates. Thus, a typical PAL of 8 macro-cells can support a circuit requiring up to approximately 160 gates. A large CPLD of 500 macro-cell circuits can support a circuit requiring up to about 10 000 equivalent gates. To implement larger circuits, one must use a **FPGA**, *Field-Programmable Gate Array*. FPGAs are very different from PLDs and CPLDs in that they do not contain *AND* or *OR* plans but it is constructed based on the philosophy as shown in figure 1.23.

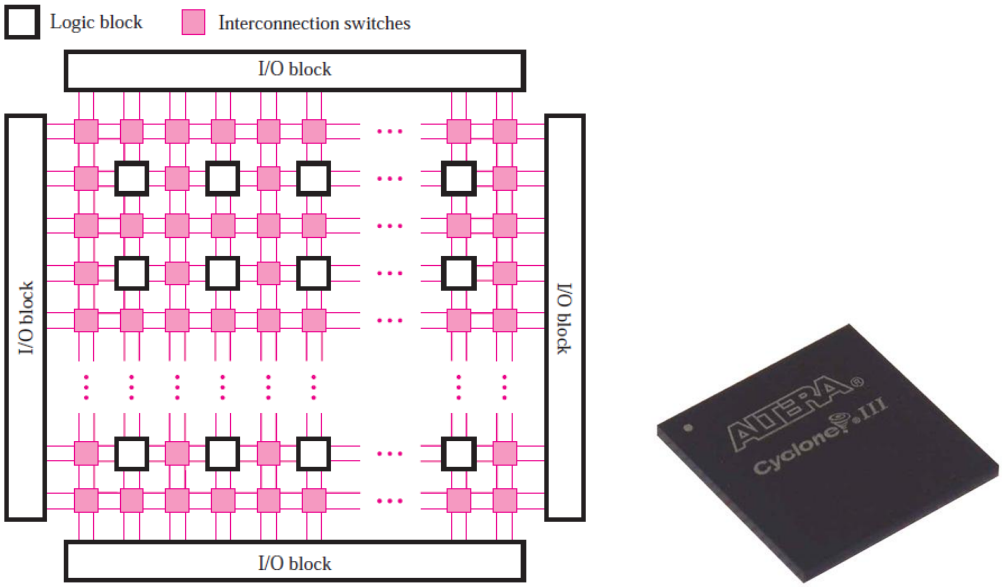


Figure 1.23: An FPGA contains three types of resources: logic blocks arranged in a two-dimensional array, I/O blocks for connection to pins, and programmable interconnect switches organized into horizontal and vertical routing channels. Example, FPGA cyclone III from Altera, 15408 LE, 346 I/O, 4 PLL, 56 M9K Memory blocks, 504 Kb RAM, 56 Multiplier

A variety of FPGA products are available in the market, with different types of logic blocks. The most commonly used logical block is **LUT**, *lookup table*. Figure 1.24 shows a *LUT*, in truth table form, containing storage SRAM cells used to implement a small logic function. LUTs of different sizes can be created, the size being defined by the number of inputs.

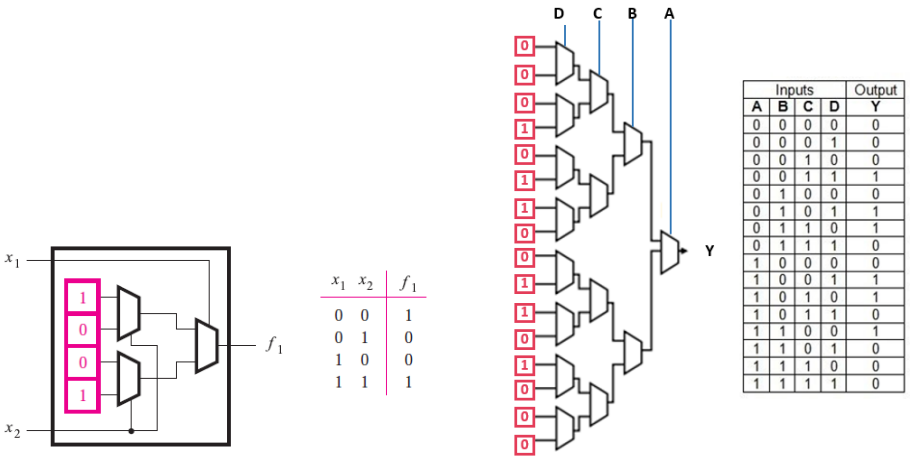


Figure 1.24: 2-inputs LUT and 4-inputs LUT.

When a circuit is implemented in an FPGA, the logic blocks are programmed to perform the necessary functions and the routing channels are programmed to establish the required interconnections between the logic blocks. Storage cells are volatile, which means they lose their stored contents every time the power is off. Therefore, the FPGA must be programmed each time power is applied. Figure 1.25 shows an example of internal interconnection in FPGA to set up a logical function f. In this example, 2-input LUTs are used and connected by four wire routing channels. Switches displayed in black are disabled.

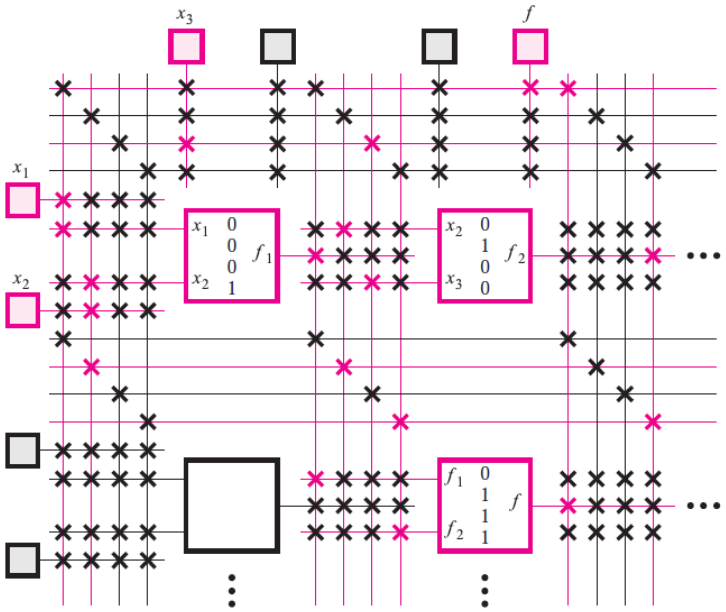


Figure 1.25: A preview on a small section of a programmed FPGA. The output function is $f = f_1 + f_2$ where $f_1 = x_1 x_2$ and $f_2 = \bar{x}_2 x_3$.

For reasons internal to the various FPGA manufacturers, several terminologies exist to designate the internal architecture of an FPGA. For Cyclone family, Altera uses the term **LE**, *Logic Element*, to denote a basic cell including a LUT, an adder and a register. A **LAB**, *Logic Array Block*, groups 10 LE. For the Stratix family, Altera replaced the LE with **ALM**, *Adaptive Logic Modules*. An ALM consists of two LUTs, two adders and two registers. For

the Stratix family, a LAB brings together 10 ALMs. For FPGAs of the Spartan and Virtex families, Xilinx uses the term **slice** for a basic module including two LUTs, two adders and two registers. A **CLB**, *Configurable Logic Block*, groups two or four slices, depending on the FPGA family.

Figure 1.26 shows an example of LE for an FPGA of the Cyclone III family from Altera (Intel since 2015). The logical elements, LE, are the smallest logical units in the architecture of this family. Each LE is composed of a LUT with four inputs allowing to implement any function of four variables, a programmable register, a *carry chain* connection (for addition), a *register chain connection* (to form registers of more than one bit). LEs operate in the following modes: normal and arithmetic. The development software automatically chooses the appropriate mode. Normal mode is suitable for general logic applications and combinatorial functions. The arithmetic mode is ideal for implementing adders, counters, accumulators and comparators.

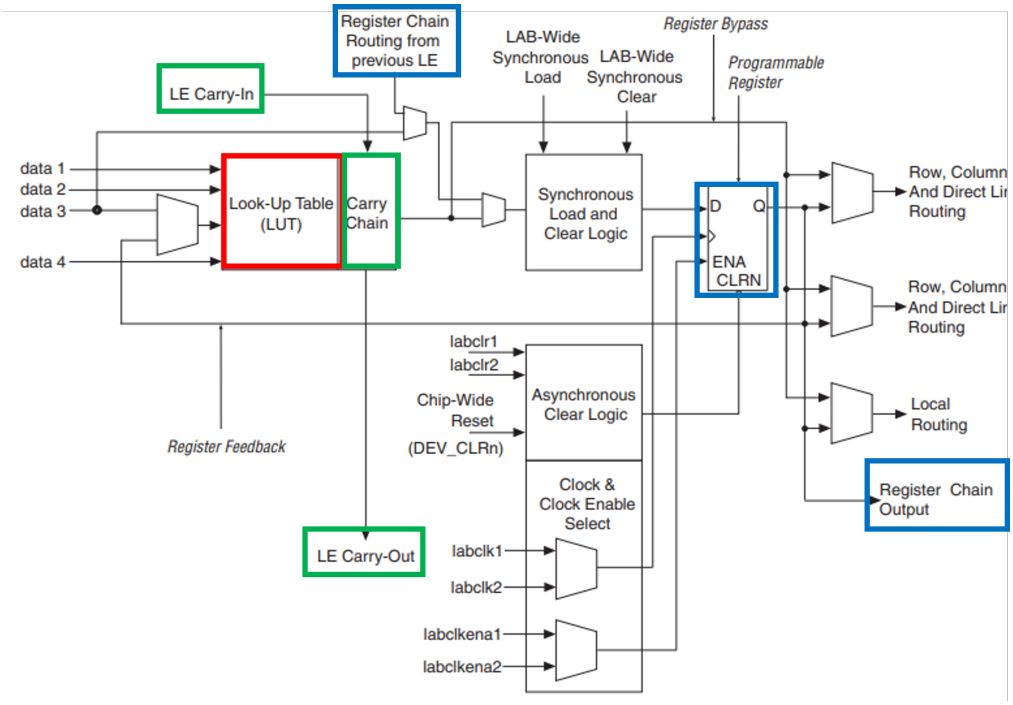


Figure 1.26: Logic element, LE, of an FPGA of the Cyclone III family.

The programmable register can be configured to be a D, T, JK or SR flip-flop. Each register has data, clock, enable and clear entries. Each LE has three outputs that drive local, row, and column routing resources. Register feedback mode allows the output of the register to return to the LUT.

3.1 FPGA Market

The global FPGA market is expected to grow from 5.9 B$ in 2020 to 8.6 B$ by 2025. It is expected to grow at a CAGR [3] of 7.6% from 2020 to 2025. This growth is fueled by

[3] Compound Annual Growth Rate or CAGR Rate

the increased adoption of the FPGA in AI applications[4], IoT[5] and ADAS[6]. FPGAs offer capabilities such as high compute density and low power consumption (for low power FPGA series), making them the preferred architecture for various applications requiring high data flow and processing. continuous data. Here are some features:

In 2019, the APAC[7] region held the largest share of the global FPGA industry as this region is home to large semiconductor companies.

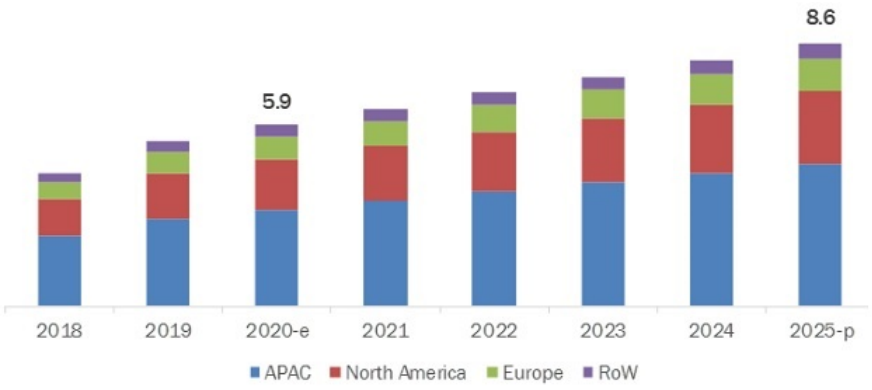


Figure 1.27: FPGA market in billion USD.

FPGAs are also used in military equipment. In November 2018, Xilinx launched Kintex and Virtex FPGAs for space and military applications. These circuits are designed to withstand harsh environments and ensure safe and reliable operation. Also, FPGAs have been adopted as an IaaS resource [8]. Several cloud service providers are deploying FPGAs to accelerate service-oriented tasks such as network encryption, deep learning, web page ranking and video conversion. For example, the Amazon.com site uses an FPGA in its virtual machine for hardware accelerations. Likewise, Microsoft uses FPGAs in its Azure platform for Deep Neural Network assessment and search ranking through its Bing engine. In February 2019, Google announced an investment of over 13B$ in data centers in the United States. On the other hand, the automotive industry is increasingly using FPGAs in the design of ADAS systems, primarily for vision processing applications that require high level processing and fine parallelism. In November 2019, Xilinx launched the new automotive qualified FPGAs, FinFET+, which target ADAS and self-driving cars.

[4]Artificial Intelligence
[5]Internet of Objects
[6]Advanced driver-assistance systems
[7]Asia-Pacific
[8]Infrastructure as a Service, a model of cloud computing.

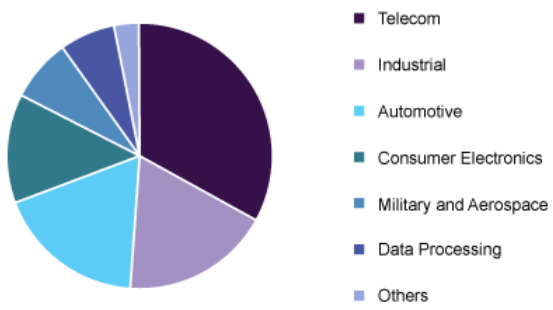


Figure 1.28: Global FPGA market share by domain in 2019.

From an FPGA manufacturing technology perspective, the SRAM[9] segment accounted for the largest share of the market. SRAM offers better flexibility, re-programmability, high integration and high performance for various applications. SRAM-based FPGAs are widely adopted in military and aerospace, telecoms, and wireless communications systems. Fuse-based FPGAs are more rugged than SRAM-based ones, especially in radiation environments. On the other hand, a high demand is known for FPGAs based on Flash memory. For example, in October 2019, Lattice Semiconductor launched the CrossLinkPlus FPGA family for the on-board vision system based on the MIPI D-PHY interface[10]. Nevertheless, FPGAs based on EEPROM memory are still relevant today.

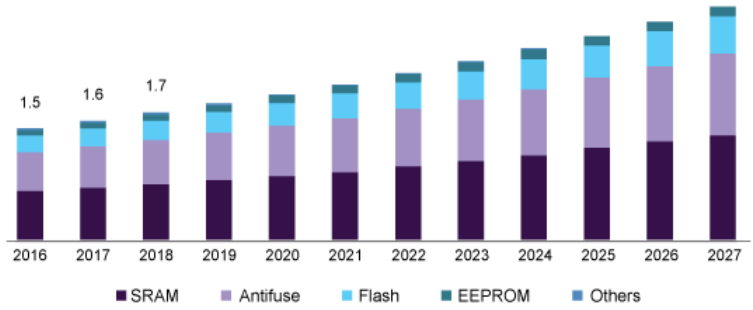


Figure 1.29: United States FPGA Market by Technology in billion USD.

3.2 ASIC

ASIC, *Application Specific Integrated Circuit*, are designed for a single application and function the same throughout their lifetime, and its function cannot be changed. The logic function of ASIC is developed in the same way as in the case of FPGAs, using HDL languages such as VHDL. However, in the case of an ASIC, the resulting circuit is permanently etched in silicon while for FPGAs, the circuit is made by programming a certain number of configurable blocks.

[9] Static-RAM
[10] MIPI D-PHY connects megapixel cameras and high-resolution displays to a processor.

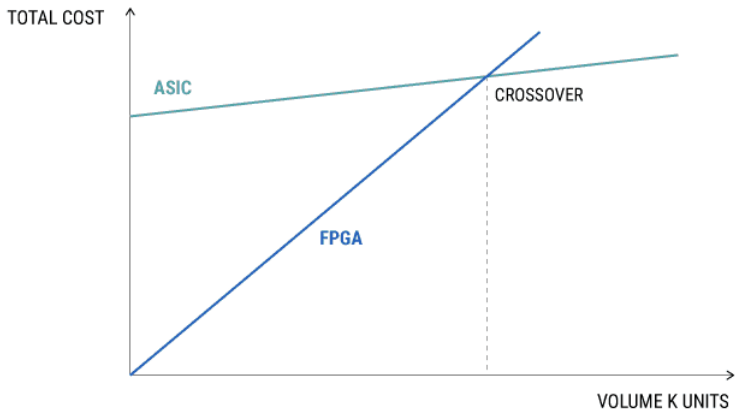


Figure 1.30: ASIC vs. FPGA costs. ASIC costs are higher initially even for low volume due to very high NRE costs, but the slope is flatter. However, at a certain production volume, the cost of an ASIC and an FPGA intersect where the ASIC becomes more cost effective.

ASICs have very high one-time engineering costs (NREs), while in the case of FPGAs this cost is almost zero. ASIC prototyping in small quantities is very expensive, but in large volumes the cost becomes more attractive. In the case of FPGAs, the cost of the IC is quite high, so in large volumes it becomes expensive compared to ASICs. Therefore, choosing between an FPGA and an ASIC depends on the end product, target market, budget, and speed required. Nevertheless, it is very convenient to at least prototype and validate the circuit using FPGAs before moving to an ASIC with very specific requirements.

3.3 IP

Traditionally, the implementation of algorithms in a digital circuit has been accomplished using ASICs or software-programmed microprocessors. These processors execute a set of instructions to perform a calculation. The alternative approach is to use reconfigurable circuits such as the FPGA, which is intended to bridge the gap between hardware and software approaches, achieving much higher performance than microprocessors, while maintaining a higher level of flexibility than ASIC. However, FPGA systems require a great deal of development and programmability, a factor that limited their early adoption.

Thus, FPGA vendors have gradually integrated more functionalities into their programmable circuits, by integrating in the logic resources, particular devices like DSP blocks and memory blocks in the form of wired functions, as shown in figure 1.31. Also, another trend has emerged namely the introduction of complete solutions of microprocessors in the FPGA structure, called hard processors.

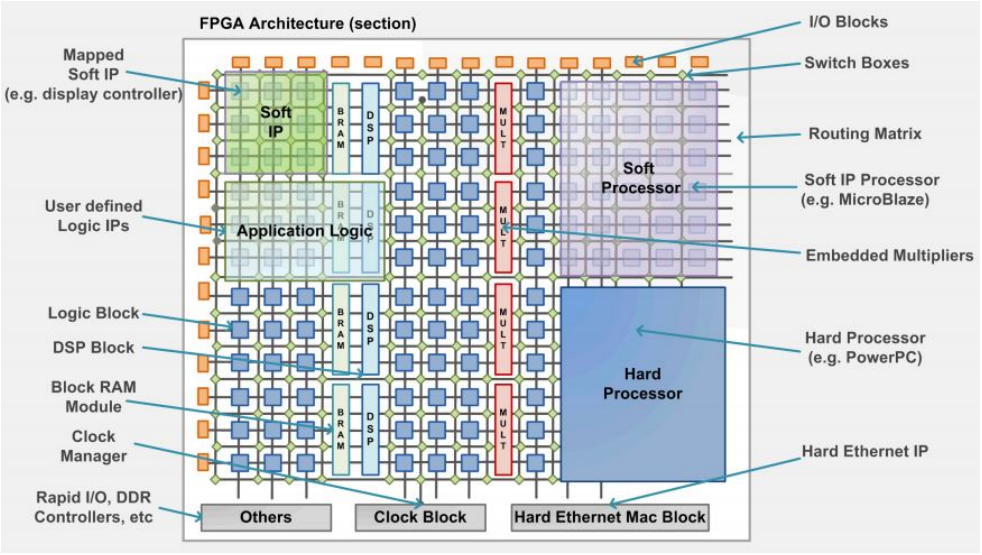


Figure 1.31: Heterogeneous FPGA platform with general configurable resources, hard blocks and any software IP blocks.

On the other hand, *soft IP* refers to *Intellectual Property* libraries of high level functions that can be included in the design. These functions are generally represented using an HDL language at the RTL[11] abstraction level. These software IP functions are then synthesized into a group of programmable logic blocks and then mapped to the FPGA.

4 HDL design

4.1 Methodology

Since the appearance of the first complex systems, the design approach has continued to evolve. It involves raising the level of representation of the system as illustrated by figure 1.32. This approach has gone from a *full custom* design (a description at the transistor level) to a *cell based* design (a description at the logic gate level) then, in the form of a hardware description language. With this logic, we look more and more at the functionality of the system without necessarily questioning its electronic composition. Therefore, we design a system at a high level of abstraction (figure 1.33), and today we talk about design at the system level itself.

[11] Register Transfer Level

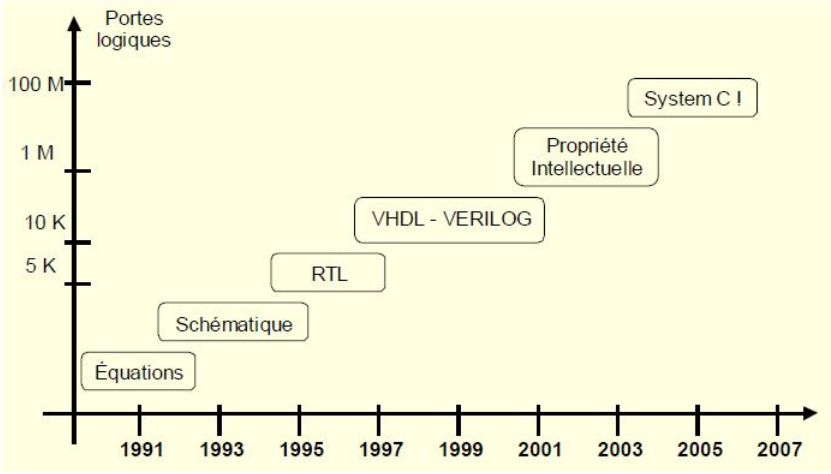


Figure 1.32: Évolution des méthodologies de conception.

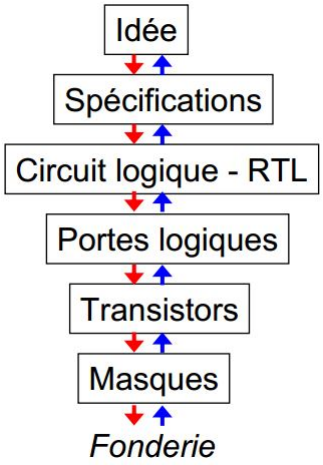


Figure 1.33: Abstraction Levels: at each level, the characteristics of the system must be analyzed and tested in order to optimize the digital system. We start from the basic component towards the architecture (bottom-up methodology) or the reverse (up-bottom methodology) or in both directions.

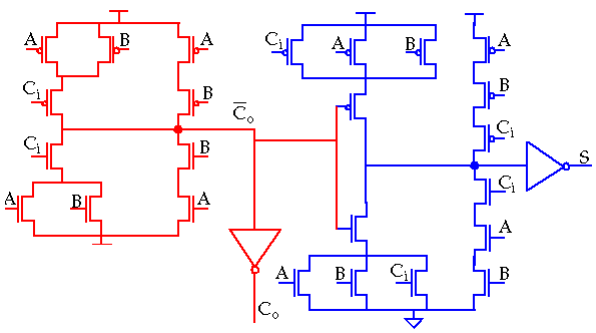
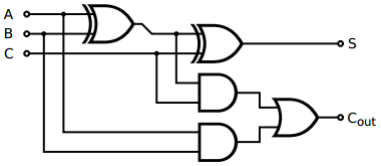


Figure 1.34: Transistor level for a Full-Adder circuit.



entrées			sorties	
a	*b*	*c*	*S*	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 1.35: Full-Adder circuit: perform the addition between two bits A, B, by taking account for the carry input C. It gives at its outputs the addition result S and a carry output bit C_{out}

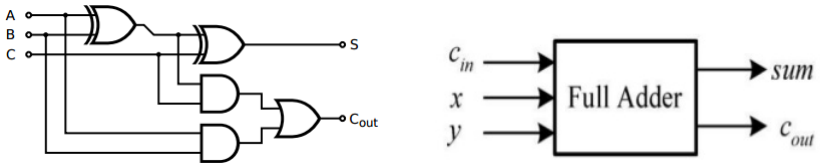


Figure 1.36: Gate and Register Transfer level for a Full-Adder circuit.

4.2 Design flow

A design flow is a sequence of step procedures to design a digital circuit and implement it in a given technology. As shown in figure 1.37, design begins with establishing the specifications by breaking down the digital system into basic modules. The description of the behavior of each module can be done by a combination of code *HDL*.

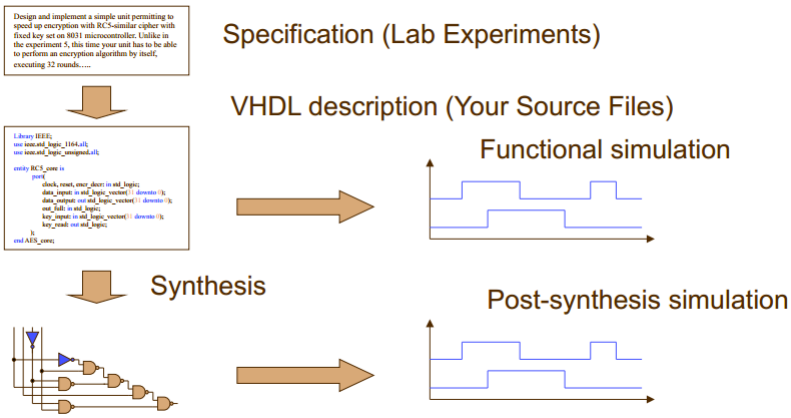


Figure 1.37: Synthesis step.

Then, the **synthesis** step is the process of generating a logic circuit from the HDL description. CAD tools, like *Quartus*, generate circuit implementations from these specifications. The process of translating, or compiling, an HDL code to a network of logic gates is part of the synthesis. The output is a set of logical expressions describing the logical functions necessary for the realization of the circuit (figure 1.37). However, regardless of the type of design used, it is impossible for a digital systems designer to pro-

duce optimal circuits manually. Thus, one of the important tasks of synthesis tools is to manipulate the user's design to automatically generate an equivalent but better circuit.

Functional Simulation occurs just after the synthesis step. A circuit represented as logical expressions can be simulated to verify that it will work as expected. A functional simulator, like *ModelSim*, uses the logical expressions generated during synthesis, and assumes that these expressions will be implemented with perfect gates through which signals propagate instantly. The simulator requires the user to specify a test-bench to be applied during the simulation. The results of the simulation are usually provided in the form of a timing diagram that the user can examine to verify that the circuit is functioning as required.

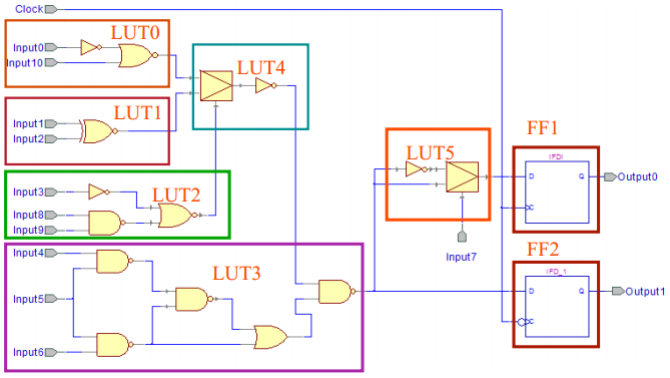


Figure 1.38: Mapping of the resources available on the FPGA to achieve the functionalities described by an HDL language. In this figure, 6 LUTs and 2 flip-flops are used to map the 11 inputs to the 2 outputs.

Physical Design comes just after the logical synthesis step. It consists of determining exactly how to implement the circuit on a given chip. The physical design tools map a circuit specified as logical expressions by using the FPGA available resources as shown in figure 1.38. They determine the placement of specific logical elements and what wiring connections should be made between these elements. The last step is commonly referred to as **routing**.

Implementation, in this step we split the list of interconnections into components available on the target FPGA (figure 1.39). The implementation includes the steps of *placement* and *routing*. Placement consists of arranging the components of the circuit in rows and columns by respecting certain time and/or space constraints imposed by the developer. Routing consists in choosing the paths followed by the interconnection wires between the components of the circuit. This step is also subject to constraints, usually of time.

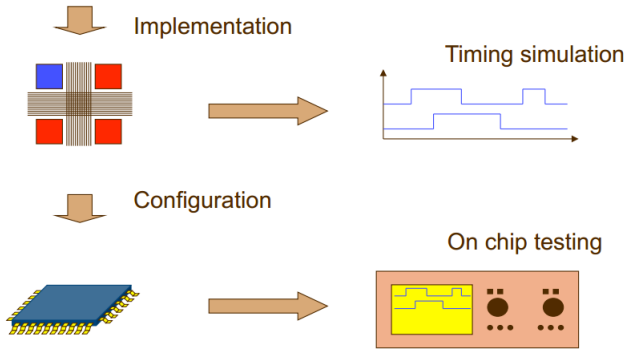


Figure 1.39: Implementation on FPGA: placement and routing.

Timing Simulation, a very important step in the design flow. Logic gates and other logical elements such as registers and flip-flops cannot perform their function with zero delay. When the values of inputs of the circuit change, it takes some time before a corresponding change occurs at the output. This is called a circuit propagation delay. Each logic element needs time to generate a valid output signal whenever there are changes in the values of its inputs. In addition, there is a delay caused by the signals which must propagate along the wires which connect various logic elements. The combined effect is that real circuits exhibit delays, which significantly impact the performances. A timing simulator evaluates the expected delays of a designed logic circuit. Its results can be used to determine whether the generated circuit meets the timing and speed requirements imposed by the specifications. If the requirements are not met, the designer can re-optimize the physical design by incorporating the timing constraints.

The last step generally consists in programming the FPGA or generating the masks which will make it possible to buildand ASIC. Figure 1.40 summarizes the design flow set for a digital system using an FPGA circuit.

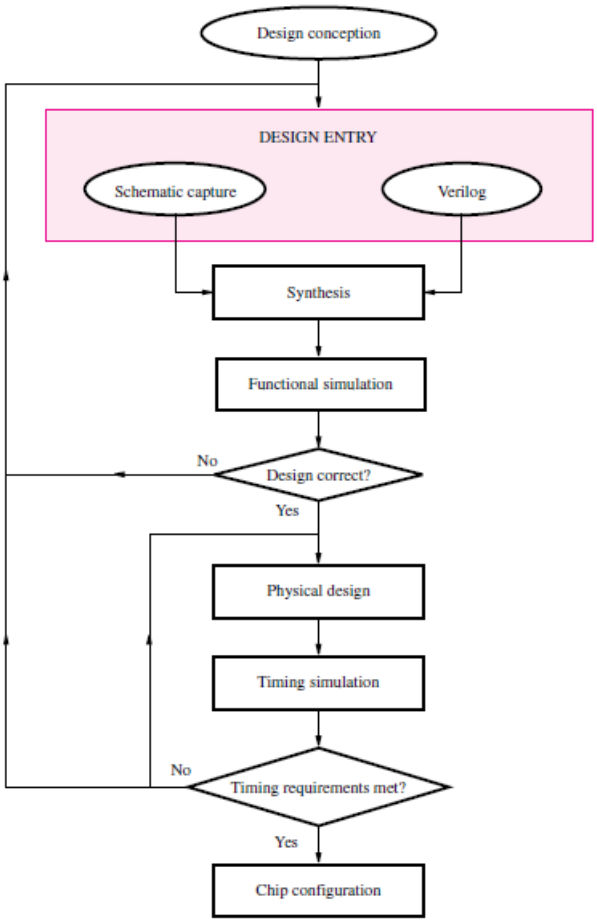


Figure 1.40: Design flow of a digital system.

Chapter 2

VHDL

The acronym VHDL stands for *Very high speed integrated circuit Hardware Description Language*. It is standardized by the IEEE, *Institute of Electrical and Electronics Engineers*. The first standard dates back to 1987, then updates were made in 1993, 2000, 2002 and 2008.

1 HDL

Hardware description language, is similar to a programming language except that HDL is used to **describe** a digital system rather than being execute on the digital system. Many HDL languages are available, some are proprietary, and can only be used to implement circuits only in proprietary technology as is the case for *Altera* and *Xilinx*. Others are supported by standardization associations like Verilog and VHDL.

2 HDL Description

The description of a digital system in VHDL has three parts:

1. declaration of libraries,
2. declaration of entities,
3. entity architecture.

2.1 Entity

Entity is a black-boxed description of the system. The entity is defined by declaring its ports. Each port is characterized by a **type**, and a **mode**.

```
entity entity_name is
  port (liste_ports)
  {Types_specifique_pour_entity}
end entity_name;

liste_ports :
```

```
  (port_identifier : [mode] indication_type [:= expression])

mode : in | out | buffer | inout

--exemple:
entity porte_logique is
 port ( x : in std_logic; y : out std_logic);
end porte_logique;

entity and_or_inv is
  port ( a1, a2, b1, b2 : in bit := '1'; y : out bit );
end and_or_inv;
```

2.2 Architecture

An **architecture** provides the detail of the circuit declared in the entity. It has two main parts: the declarative section and the body of the architecture. The declarative section can be used to declare signals, new types, variables, constants, functions, components and more. The body of the architecture represents the functional description of the system. Two descriptions are possible: **concurrent** and **sequential**.

```
architecture identifier of entity_name is
  { block_declarative }
begin
  { concurrent_statement }
end identifier;

--exemple
architecture abstract of adder is
begin
  sum <= a + b;
end abstract;

--exemple
architecture primitive of and is
  signal and_a, a1, a2 : std_logic;
begin
    and_a <= a1 and a2;
end primitive
```

2.3 Library

A **librairy** includes essential packages for the design of a circuit. It consists of two clauses: **library** and **use**. Three librairies exist:

1. **WORK**: built-in default library. It includes the project being designed before and after compilation.

2. **STD**: built-in. It includes the definition of logical, relational and other operators.

3. **IEEE** must be explicitly declared. It includes other types as the *std_logic* and the corresponding operations defined in the *ieee.std_logic_1164* package. A specific type can be defined but we can invoke the set of types by the keyword *all*.

```
library IEEE;
use IEEE.std_logic_1164.all;
```

2.4 Concurrence

In VHDL, everything is concurrent, and hence the order of expressions within a VHDL description does not have any importance. However, VHDL also allows to describe sequential operations via the **process** mechanism. A *process* is evaluated at each change of signals' state in its sensitivity list.

```
--syntaxe
process (signal_names | all ) is
  {process_declarative_item}
begin
  {sequential_statement}
end process;

--exemple
 process (A, B, C, D)
 begin
  if (A = '1' and B = '0') nor (C = '0' and D = '0') then F <= '1';
  else F <= '0';
  end if;
end process;
```

3 Description styles

There are three styles of VHDL description:

1. Dataflow design,
2. Behavioral design,
3. Structural design.

To explain each of these styles, we consider the circuit of the figure 2.1.

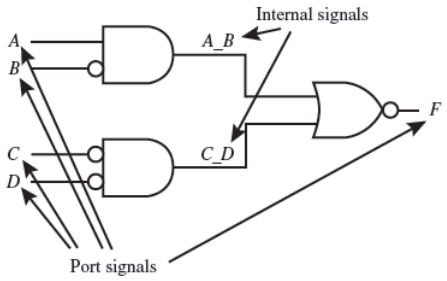


Figure 2.1: Example.

3.1 Behavioral

The behavioral description of a digital system is similar to programming with any procedural language like *C* or *Java*. This description provides a high level of abstraction in describing the system's behavior where the *process* is the basic unit. We can use Boolean equations, conditional and selection tests. In the body of a *process*, expressions are evaluated sequentially, unlike expressions written in the body of the architecture, evaluated in a concurrent fashion.

```
architecture behavior of bloc is
  signal A_B, C_D: std_logic;      --internal signal declarations
begin
 process (A, B, C, D, A_B, C_D)
 begin
  A_B <= A and not B;
  C_D <= not C and not D;
  F <= A_B nor C_D;
 end process;
end behavioral;

--ou encore
architecture behavior of bloc is
begin
  process (A, B, C, D)
  begin
    if (A = '1' and B = '0') nor (C = '0' and D = '0') then F <=
    '1';
    else F <= '0';
    end if;
  end process;
end behavioral;

--ou encore
architecture behavior of bloc is
begin
 process (A, B, C, D)
 begin
    case (A and not B) nor (not C and not D) is
      when '1' => F <= '1';
      when '0' => F <= '0';
      when others => null;
    end case;
  end process;
end behavioral;
```

Il est à noter que les assignations de valeurs aux objets *signal* ne sont effectuées que lorsque le processus se termine. Si plusieurs assignations sont faites à un objet *signal* dans un processus, seule la dernière sera en fait effectuée. Les expressions contenant des objets *signal* sont évaluées avec les valeurs présentes lorsque le processus est lancé. Il faut donc utiliser les assignations multiples et interdépendantes d'objets *signal* à l'intérieur d'un *process* avec beaucoup de prudence. A l'inverse, les objets *vari-*

able prennent immédiatement la valeur qui leur est assignée par une expressions. C'est l'essence même de la description *behavior* qui veut se rapprocher de la programmation procédurale. Cependant, ces variables ne sont pas synthétisables et il va falloir être très prudent lors de la synthèse du circuit sur *FPGA*.
Note that assignments of values to *signal* objects are only made when the process ends. If more than one assignment is made to a *signal* object in a process, only the last one will actually be available. Conversely, *variable* objects immediately take the value assigned to them by an expression. This is the very essence of the description *behavior* which wants to get closer to procedural programming. However, these variables cannot be synthesized and it will be necessary to be very careful when synthesizing the circuit on *FPGA*.

3.2 Dataflow

In this description, the values of signals and ports are established by concurrent assignments. We use Boolean equations, conditional and selection tests.

```
architecture dataflow of bloc is
begin
  --Boolean equations
  F <= (A and not B) nor (not C and not D);

  --conditional signal assignments
  F <= '1' when ((A and not B) nor (not C and not D)) = '1' else
    '0';

  --selected signal assignments
  with (A and not B) nor (not C and not D) select
  F <= '1' when '1',
       '0' when '0',
       '0' when others;
end dataflow;
```

3.3 Structural

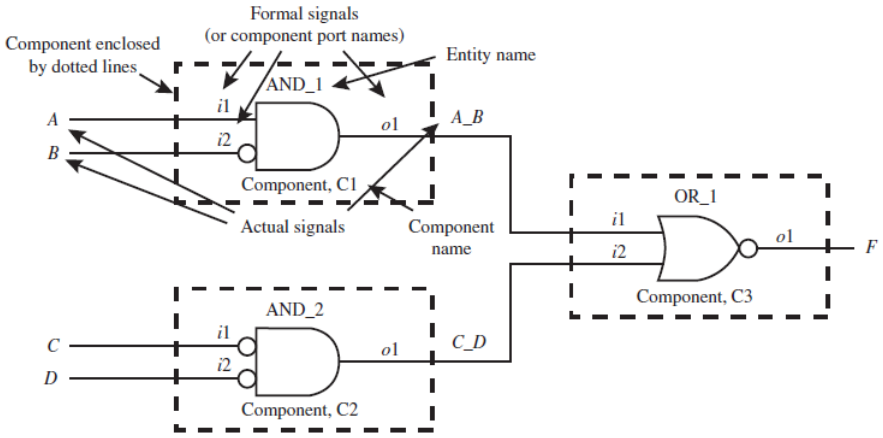
Hierarchical design using primitives named **component**. The description of the digital system is based on a simple assembly of the different components that constitute it.

```
component my_component is
  port ( port_interface_list )
end component;
```

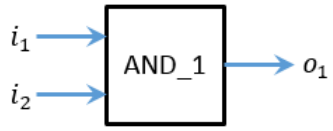
instantiation: create a copy of an existing component.
port map: interconnection of a component's generic I/O with real I/O or other intermediate signals.

```
instantiation_label : component component_name port map (
   port_association_list);
port_association_list :
  port_name => ( signal_name | expression | open )
```

1. Partitioning:



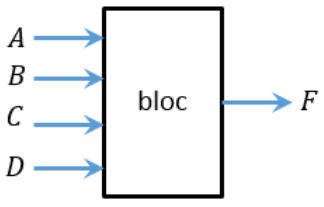
2. Define the components:




```
--Component definition for AND_1
library IEEE;
use IEEE.std_logic1164.all;

entity and_1 is
  port ( i1, i2: in std_logic; o1: out std_logic);
end and_1;
architecture dataflow of and_1 is
begin
  o1 <= i1 and not i2;
end dataflow;
```

3. Declare the top-level entity:




```
--Top Level
library IEEE;
use IEEE.std_logic1164.all;
entity bloc is
  port (A, B, C, D : in std_logic; F : out std_logic);
end bloc;
```

4. Declare the components in the architecture's declarative section of the top-level entity:

```
component and_1 is port (i1, i2: in std_logic; o1: out std_logic);
end component;
component and_2 is port (i1, i2: in std_logic;o1: out std_logic);
end component;
component or_1 is port (i1, i2: in std_logic;o1: out std_logic);
end component;
```

4. Instantiations and port mapping:

```
C1: and_1 port map (i1 => A, i2 => B, o1 => A_B);
C2: and_2 port map (i1 => C, i2 => D, o1 => C_D);
C3: or_1  port map (i1 => A_B, i2 => C_D, o1 => F);
```

Two ways for port mappin:

1. named association: mapping is done by name from a generic I/O like $i1$ to a real signal like (A). The order is not important.

```
C1: and_1 port map (o1 => A_B, i1 => A, i2 => B);
C2: and_2 port map (i1 => C, o1 => C_D, i2 => D);
C3: or_1  port map (i2 => C_D, o1 => F, i1 => A_B);
```

2. positional association: generic I/O are omitted to have more readability but we must respect the order of the component's I/O as they declared in its entity.

```
C1: and_1 port map (A, B, A_B);
C2: and_2 port map (C, D, C_D);
C3: or_1  port map (A_B, C_D, F);
```

4 Data types

4.1 Objects and types

The information in VHDL is represented as objects. Three types of objects are defined: signals, constants and variables. Each object in VHDL has a value of a particular type. The type defines the set of values and the set of operations (figure 2.3).
We use the **signal** object to represent scalar logic signals, in the form of buses or binary numbers. A scalar signal takes two possible values '0' or '1'. An example of a 4-bit encoded bus signal is "1001". Integers can also be specified in decimal without using quotes.

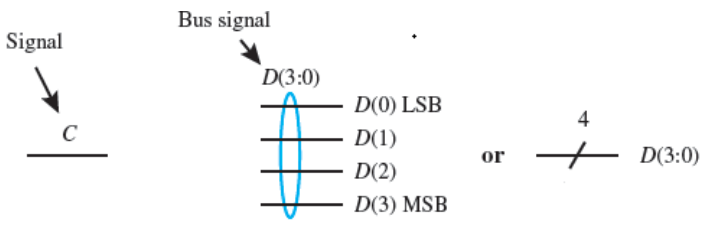


Figure 2.2: Signal: scalar and bus.

signal objects can be declared in the entity declaration, in the declarative section of an architecture and in the declarative section of a *package*. A signal must be declared with an associated type, as follows:

```
signal nom_signal : type_signal;
-- type_signal peut etre:
bit, bit_vector,
std_logic, std_logic_vector,
signed, unsigned, integer.
```

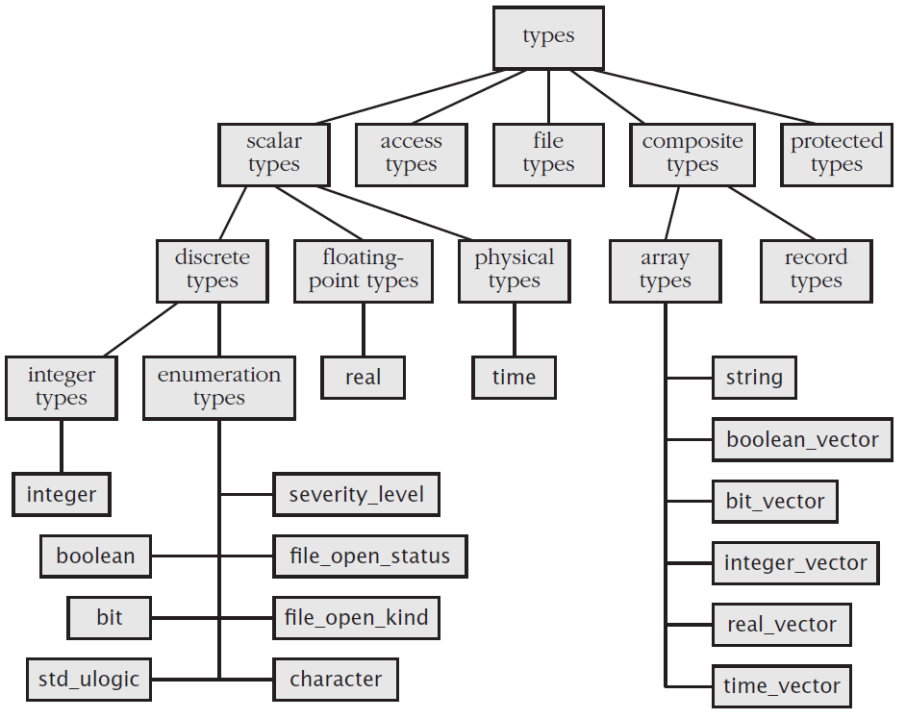


Figure 2.3: Predefined types in VHDL.

The type **bit** and **bit_vector** are predefined in the *STD* library. In reality, the *bit* type is just an **enumeration** of two values ('0', '1').

```
signal x: bit;
signal y: bit_vector(1 to 4);
signal z: bit_vector(7 downto 0);
```

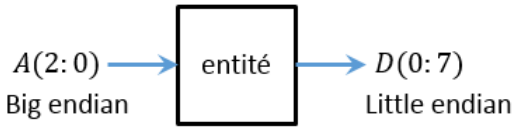


Figure 2.4: Two possible ways to represent a bus object: little endian (downto) and big endian (to).

The **std_logic** type was added to the VHDL standard in IEEE 1164. It offers more capability than the *bit* type. To use this type, we must include the following two declarations:

```
library IEEE;
use     IEEE.std_logic_1164.all;
```

In reality, the *std_logic* type is also a **enumeration** of nine values allowing to consider other electrical levels:

```
library ieee;
use ieee.std_logic_1164.all;
type std_ulogic is (
  'U', -- Uninitialized
  'X', -- Forcing Unknown
  '0', -- Forcing zero
  '1', -- Forcing one
  'Z', -- High Impedance
  'W', -- Weak Unknown
  'L', -- Weak zero
  'H', -- Weak one
  '-' -- Don't care
  );
```

It is possible to include more options in the declaration of a signal such as the delays defined during a transaction as shown in the following listing:

```
--syntaxe:
signal nom_signal : type_signal;
nom_signal <= forme_signal | valeur [after expression_delai];
expression_delai :
  delai | unaffected

--exemple
y <= not x after 5 ns;                  -- une transaction
z <= '1'   after T_p, '0' after 2*T_p; -- deux transactions

-- unaffected == null
if u = 0 then
  w1 <= w1 + '1';
  w2 <= unaffected;
else
  w1 := unaffected;
  w2 <= '1';
end if;
```

Packages *std_logic_signed* and *std_logic_unsigned* use another package named *std_logic_arith*. This latter define **signed** and **unsigned** types which are equivalent to the *std_logic_vector* type. The type *unsigned* allows to work with unsigned numbers or Natural numbers $\mathbb{N}$. The type *signed* allows to consider signed number $\mathbb{Z}$ codded in two-complement.
Constant allows to give to an object a value wich cannot be modified. Unlike a *signal*, a constant does not represent a physical resource so this object cannot be synthesized on FPGA. The purpose of a constant is to improve the readability of the code.

```
--syntaxe:
constant identifier : indication_type [:= expression]

--exemples
constant nombre_octet    : integer := 4;
constant nombre_bits     : integer := 8 * nombre_octet;
```

```
constant e                : real    := 2.718281828;
constant delai            : time    := 3 ns;
constant size_limit       : integer := 255;
```

Like a constant, a **variable** cannot also be synthesized on FPGA. They are sometimes used to hold calculation results and for use in loops.

```
--syntaxe:
variable identifier : indication_type [:= expression ]
--exemples
variable index                  : integer := 0;
variable start, finish          : time := 0 ns;
variable sum, average, largest : real;
```

5 New types

5.1 Syntaxe

In some situations, it makes sense to create new types from a basic types. The new types do not affect the synthesis of the circuit. On the other hand, they are very useful for improving and speeding up coding. The new type can be used in *port* declaration of an *entity* or in the declarative part of an *architecture*.
You can create almost any new type from the base type *integer* as follows:

```
-- syntaxe:
type identifier is type_indication;
--indication:
  range simple_expression (to|downto) simple_expression;

--exemple
type apples is range 0 to 100;
type oranges is range 0 to 100;
type day_of_month is range 0 to 31;
type set_index_range is range 21 downto 11;
variable set_index : set_index_range;
```

Also new types for handling physical quantities as follows:

```
-- Types physiques, syntaxe:
type identifier is type_indication
  units
    identifier;
    identifier = physical_literal ;
  end units;

--exemple
type resistance is range 0 to 1E9
units
  ohm;
  kohm = 1000 ohm;
  Mohm = 1000 kohm;
end units resistance;
```

Also a new type from base type *enumeration* as follows:

```
--Enumerations, syntaxe:
type identifier is (enumeration_set)

--exemple
type alu_function is (disable, pass, add, subtract, multiply, divide
   );
type octal_digit  is ('0', '1', '2', '3', '4', '5', '6', '7');
variable alu_op     : alu_function;
variable last_digit : octal_digit := '0';
alu_op     := subtract;
last_digit := '7';

--exemple surcharge enumeration: overloading
type logic_level is (unknown, low, undriven, high);
variable control : logic_level;
type water_level is (dangerously_low, low, ok);   --"low" est
   surcharge
variable water_sensor : water_level;
control := low;
water_sensor := low;
```

Example of creating a new type to be used as a port type:

```
package int_types is
  type small_int is range 0 to 255;
end package int_types;

use work.int_types.all;
entity small_adder is
  port ( a, b : in small_int; s : out small_int );
end entity small_adder;
```

5.2 Qualification of type

e can apply some attribute on a type to recover some usefull informations. A qualified expression is an expression whose type is explicitly specified. Example: shared member of an enumerations.

```
--Qualification syntaxe:
type_name'(expression)

--exemple de qualification
type logic_level is (unknown, low, undriven, high);
type system_state is (unknown, ready, busy);
logic_level'(unknown)
system_state'(unknown)

--exemple de qualification
subtype valid_level is logic_level range low to high;
logic_level'(high)
valid_level'(high)
```

5.3 Subtype

Useful for restricting the scope of a base type. All the operators applicable on a base type are also applicable for a subtype. The results are of basic type and not of subtypes.

```
--subtype syntaxe
subtype identifier is type_base subtype_indication
--exemple
subtype small_int is integer range -128 to 127;
variable deviation  : small_int;
variable adjustment : integer;
deviation := deviation + adjustment;
```

There are predefined subtypes in VHDL:

1. Natural: $\mathbb{N}$
2. Positive: $\mathbb{N}^*$
3. Delay_length: **time**.

```
subtype natural is integer range 0 to highest_integer;
subtype positive is integer range 1 to highest_integer;
subtype delay_length is time range 0 ns to highest_time;
```

5.4 Attributes

Attributes provide useful information about the characteristics of a type or an object. VHDL includes predefined attributes, but you can also define new ones. The predefined attributes are divided into five classes: value, function, signal, type and range of values.

```
type resistance is range 0 to 1E9
  units
    ohm;
    kohm = 1000 ohm;
    Mohm = 1000 kohm;
  end units resistance;

resistance'left  -- leftmost value, 0 ohm
resistance'right -- rightmost value, 1E9 ohm
resistance'low   -- least value, 0 ohm
resistance'high  -- greatest value, 1E9 ohm
resistance'ascending -- ascending range? true
resistance'image(2 kohm) -- 2000 ohm
resistance'value("5 Mohm") -- 5_000_000 ohm
```

```
type logic_level is (unknown, low, undriven, high);
logic_level'pos(unknown)   -- position of unknown, 0
logic_level'val(3)         -- value at position 3, high
logic_level'succ(unknown)  -- value at position of unknown + 1, low
logic_level'pred(undriven) -- value at position of unknown - 1, low
```

```
-- if there is an event on clk and if the value of clk just before
-- the last event on clk
if clk'event and clk = '1' and clk'last_value = '0' then
  assert d'last_event >= 2ns
  -- The time interval since the last event on d.
    report "Timing error: d changed within setup time of clk";
end if;

-- test the pulse width of the clock
assert not clk'event or clk'delayed'last_event >= Tpw_clk
  report "Clock frequency too high";
```

6 Arrays

An *array* is a set of values of the same type. The position of each element is given by an *index*.

6.1 Constrained array

In constrained array, the limits of an index are established when the array type is defined.

```
-- syntaxe
type identifier is array (index_ranges) of
   element_subtype_indication
index_ranges:
  discrete_subtype_indication | simple_expression (to|downto)
   simple_expression
element_subtype_indication:
  type_base [range simple_expression (to|downto) simple_expression]

-- 1D array with one index range do not have to be numeric
type word is array (31 downto 0) of bit;

-- Example 2: index_ranges
type controller_state is (initial, idle, active, error);
type state_counts is array (idle to error) of natural;
type state_counts is array (controller_state range idle to error) of
    natural;

-- Example 3:
subtype ram_address is integer range 0 to 63;
type coeff_ram_address is array (ram_address) of real;

-- Example 4:
type RAM is array (0 to 31) of integer range 0 to 255;
```

6.2 Unconstrained array

A definition of type *array* can be unconstrained, that is, of index undefined length. the *bit_vector* and *std_logic_vector* types are defined in this way. An object of type unconstrained array must have its range of index types defined when it is declared.

```
--syntaxe
type identifier is array (type_base range <>) of
   element_subtype_indication
element_subtype_indication :
  type_base [discrete_range]

--exemple
type int_vector is array (integer range <>) of integer;
variable int_table : int_vector(0 to 9);
```

6.3 Aggregates

Objects of type *array* can also be assigned using concatenation operator (&) or aggregates. By default, the assignment is made by taking the position into account.

```
-- association par position
type point is array (1 to 3) of real;
constant origin : point := (0.0, 0.0, 0.0);
variable view_point : point := (10.0, 20.0, 0.0);

-- association par nom
variable view_point : point := (1 => 10.0, 2 => 20.0, 3 => 0.0);

-- association par nom
type coeff_array is array (coeff_ram_address) of real;
variable coeff : coeff_array := (0 => 1.6, 1 => 2.3, 2 => 1.6, 3 to
   63 => 0.0);
variable coeff : coeff_array := (0 => 1.6, 1 => 2.3, 2 => 1.6,
   others => 0.0);
variable coeff : coeff_array := (0 | 2 => 1.6, 1 => 2.3, others =>
   0.0);

-- Pas de melange
variable coeff : coeff_array := (1.6, 2.3, 2 => 1.6, others => 0.0);
    -- illegal
```

6.4 Array attributes

```
type A is array (1 to 4, 31 downto 0) of boolean;
A'left(1)            -- 1
A'low(1)             -- 1
A'right(2)           -- 0
A'high(2)            -- 31
A'range(1)           -- 1 to 4
```

```
A'reverse_range(2)  -- 0 to 31
A'length(1)         -- 4
A'length(2)         -- 32
A'ascending(1)      -- true
A'ascending(2)      -- false
A'element           -- boolean

--example
count := 0;
for index in A'range(2) loop
  if A(index) then count := count + 1;
  end if;
end loop;
```

7 Code reuse

Speed up design and share code: Package, Component, Procedure, Function.

7.1 package

A package is a collection of declarations that serves as a repository. It is used to contain general purpose VHDL code like type, subtype, constant, signals, functions and others. The package can be included for use in a VHDL description. It allows you to separate declarations from implementations. Like an architecture, a package can have two main parts, the declaration and the body which is optional. The general form of a package declaration is shown as follows:

```
--package_declaration :
package identifier is
  { package_declarative_item }
end [package] [identifier] ;

--example declaration
-- constantes and types to model a CPU:
package cpu_types is
  constant word_size : positive := 16;
  constant address_size : positive := 24;
  subtype word is bit_vector(word_size - 1 downto 0);
  subtype address is bit_vector(address_size - 1 downto 0);
  type status_value is ( halted, idle, fetch, mem_read, mem_write,
   io_read, io_write, int_ack );
end package cpu_types;

--example: modeling an address decoder
entity address_decoder is
  port (addr    : in work.cpu_types.address;
        status  : in work.cpu_types.status_value;
        mem_sel, int_sel, io_sel : out bit );
end entity address_decoder;
```

```
22
23 architecture functional of address_decoder is
24 begin
25   mem_sel <= '1' when
26             (work.cpu_types."=" (status, work.cpu_types.fetch)
27             or work.cpu_types."="(status, work.cpu_types.mem_read)
28             or work.cpu_types."="(status, work.cpu_types.mem_write)
    )
29             else '0';
30 end architecture functional;
```

```
1 -- by using the clause use
2 use work.cpu_types.all
3
4 entity address_decoder is
5   port (addr    : in address;
6         status  : in status_value;
7         mem_sel, int_sel, io_sel : out bit );
8 end entity address_decoder;
9
10 architecture functional of address_decoder is
11 begin
12   mem_sel <= '1' when status = (fetch or mem_read or mem_write) else
       '0';
13 end architecture functional;
```

7.2 Procedures

A procedure is a subroutine with a list of parameters. Parameters can have modes *in*, *out* and *inout*. A procedure accepts input values and returns results, but it can also modify parameters passed to it. A procedural call is considered as a concurrent statement. Finally, the procedures can be overloaded, that is, two procedures can have the same identifier but a list of different parameters. Procedures are most useful for writing part of code that is used repeatedly in a VHDL description.

```
1 procedure identifier [ (parameter_interface_list) ] is
2   { subprogram_declarative_part }
3 begin
4   { sequential_statement }
5 end [procedure] [identifier];
6
7 --exemple: averaging an array of data
8 procedure average_samples is
9   variable total : real := 0.0;
10 begin
11   assert samples'length > 0 severity failure; -- message d'erreur
12   for index in samples'range loop
13     total := total + samples(index);
14   end loop;
15   average := total / real(samples'length);
16 end procedure average_samples;
```

```

-- procedure call statement
average_samples;
```

```
-- exemple: memory read procedure
process is
  variable mem_address_reg, mem_data_reg, prog_counter, instr_reg,
   accumulator, index_reg : word;
  ...
procedure read_memory is
begin
  address_bus <= mem_address_reg;
  mem_read <= '1';
  mem_request <= '1';
  wait until mem_ready or reset;
  if reset then return;
  end if;
  mem_data_reg := data_bus_in;
  mem_request <= '0';
  wait until not mem_ready
end procedure read_memory;

begin
-- initialization
  loop
    mem_address_reg := prog_counter;  -- fetch next instruction
    read_memory; -- call procedure
    instr_reg := mem_data_reg;
    ...
    case opcode is
      ...
      when load_mem =>
        mem_address_reg := index_reg + displacement;
        read_memory; -- call procedure
        accumulator := mem_data_reg;
      ...
    end case;
  end loop;
end process;
```

```
procedure identifier [ (parameter_interface_list) ] is
  { subprogram_declarative_part }
begin
  { sequential_statement }
end [procedure] [identifier];

parameter_interface_list :
  ( [constant | variable | signal] identifier {,...} :
    [ mode ] subtype_indication [:= static_expression] ) { ; ... }

mode : in | out | inout

```

```
--exemple
procedure do_arith_op ( op : in func_code ) is
  variable result : integer;
begin
  case op is
    when add => result := op1 + op2;
    when subtract => result := op1 - op2;
  end case;
  dest <= result after Tpd;
  Z_flag <= result = 0 after Tpd;
end procedure do_arith_op;

-- procedure call
do_arith_op (add);
```

7.3 Function

A function is a form of a generalized operator. It is a subroutine that returns a single result. A function can accept parameters as input. These parameters cannot be changed by the function. We use a function call in an expression.

```
[pure | impure] function identifier [ (parameter_interface_list) ]
   return type_mark is
  {subprogram_declarative_item}
begin
  { sequential_statement }
end [function] [identifier] ;

--exemple: whether a value is within given bounds?
function limit (value,min,max : integer) return integer is
begin
  if    value > max then return max;
  elsif value < min then return min;
  else return value;
  end if;
end function limit;

-- A call to this function might be included in a variable
   assignment statement
new_temperature := limit (current_temperature + increment,10,100);

-- A call to this function might be used in further computation
new_motor_speed := old_motor_speed + scale_factor * limit (error
   ,-10,+10);
```

Chapter 3

Datapath and FSM

1 Datapath and Control path

Designing a circuit to perform simple operations on simple data, such as an adder, can be done through a combinatorial approach. However, when the data and/or the operations become complex, for example the computation of a series on numbers of several bits, the design of the circuit involves several steps. Therefore, a design based on an algorithm-based sequential approach is more relevant.

```
1 --algorithm
2 size = 4; sum = 0;
3 for i in (0 to size-1) do
4   sum = sum + a(i);
5 q = sum/8;
6 r = sum mod 8;
7 if (r > 3)
8   q = q+1;
9 outp = p;
```

Listing 3.1: Algorithme

Figure 3.1 shows an example of the design of a sequential system corresponding to the algorithm in listing 3.1 with a description *data flow*. To add four numbers, a *data flow* description sets up three concurrent adders. Consequently, this description requires as many adder as number of elements in a, hence an overuse of the resources of the FPGA.

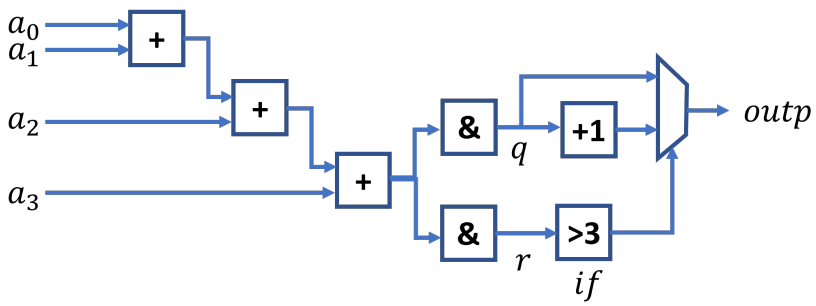


Figure 3.1: Design of a sequential system with a description *dataflow*.

A sequential description, with a single adder and an accumulator register, to store the

intermediate additions would have been sufficient to design the circuit of the algorithm regardless of the number of elements. This description adopts the same architecture as micro-controllers using a datapath as in figure 3.2.

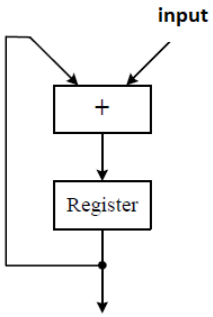


Figure 3.2: Design of a sequential system with a description *sequential*.

For the datapath to behave correctly, all of the elementary blocks must be perfectly synchronized and controlled. Therefore, an additional circuit is needed to generate these control signals. An example of the most basic control signal is the select signal for multiplexers, read/write enable signals for registers, address lines for register banks and the enable signal for three states logic gates. Thus, the operation mode of the datapath is determined by which control signals are active or not at a given time. In a microprocessor or micro-controller, these control signals are generated by the control unit based on a finite state machine **FSM**.

This introduction gives us some elements for the implementation of an algorithm on an FPGA. The Register Transfer **RT** methodology introduces hardware resources to synthesize variables and sequential expressions which are the characteristic elements of an algorithm. This methodology uses registers to model symbolic variables, a datapath to implement operations and an FSM to specify the order of operations.

2 Sequential circuits

These are circuits where the notion of **state** plays a primordial role. There are two types of sequential circuit: synchronous and asynchronous. Let recall some basic sequential elements.

2.1 Latch and flip-flop

Figure 3.3 shows the simplest memory element consisting of a loop with two inverters. If we assume that $A = 0$, then $B = 1$ and the circuit will hold these values indefinitely. We say that the circuit is in the state defined by these values. If we assume that $A = 1$, then $B = 0$, and the circuit will remain in this second state indefinitely. Thus, the circuit has two possible states. However, it is not possible to change the state of this circuit by external inputs.

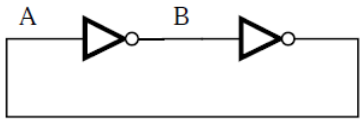
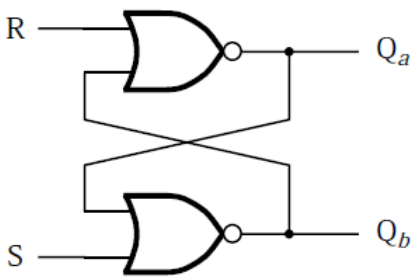


Figure 3.3: The simplest memory element.

Latch

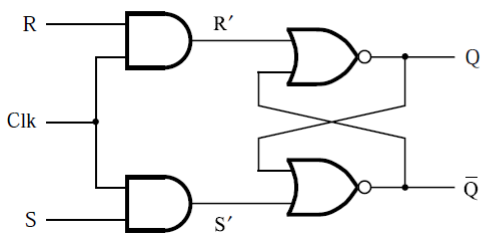
Figure 3.4 presents a memory element constructed with NOR gates. Its inputs, *Set* and *Reset* allow to change the state, Q, of the circuit. This circuit is called **latch** or basic **lock**.



S	R	Q_a	Q_b
0	0	0/1	1/0
0	1	0	1
1	0	1	0
1	1	0	0

Figure 3.4: RS Latch with NOR gates and its truth table.

The basic RS latch can serve as a memory element in which, we can change the memorized state from inputs. State changes occur when changes in the input signals occur. The modified circuit of figure 3.5 includes two AND gates which provide control of the latch via an external signal clk. The resulting circuit is usually referred *gated latch*.



Clk	S	R	Q(t + 1)
0	x	x	Q(t)
1	0	0	Q(t)
1	0	1	0
1	1	0	1
1	1	1	x

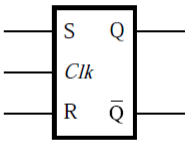
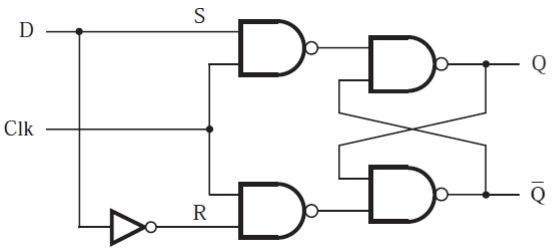


Figure 3.5: Gated latch = RS LATCH with a control signal clk.

Figure 3.6 shows a D-latch based on the SR-lock with a single data entry D. State changes only occur when $Clk = 1$.



Clk	D	Q(t + 1)
0	x	Q(t)
1	0	0
1	1	1

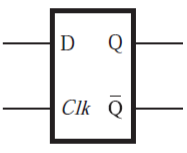


Figure 3.6: D-latch with a control signal clk.

Flip-flop

For latchs, the state's change is sensitive to the level of the input signals, *level-sensitive*. But, there was a need for a storage elements which can only change state once during a clock cycle, these are flip-flops.
The term flip-flop designates a storage element which changes its output at the edge of a control clock signal. One example is the circuit in figure 3.7, which consists of two D-latches. The first, called *master*, changes state as long as $clk = 1$, while the second, called *slave*, changes state as long as $clk = 0$. Therefore, when the clock is at logic high level, the master follows the input signal D and the slave does not change and when the clock signal is at the level logic low, the slave follows the output of the master. Thus, the master-slave circuit changes state on the falling edge of the clock regardless of the number of changes in input D during one clock cycle.

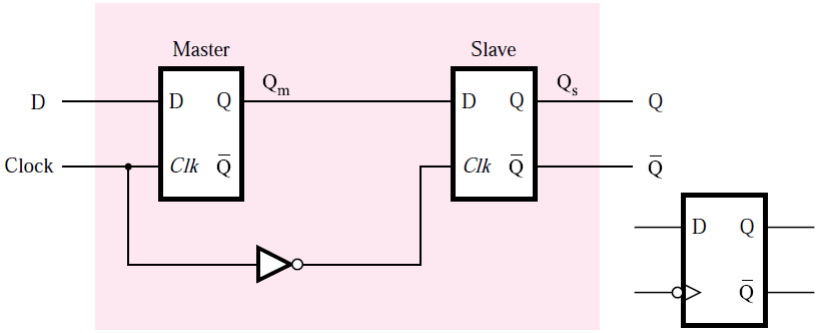


Figure 3.7: Master-slave D-flipflop.

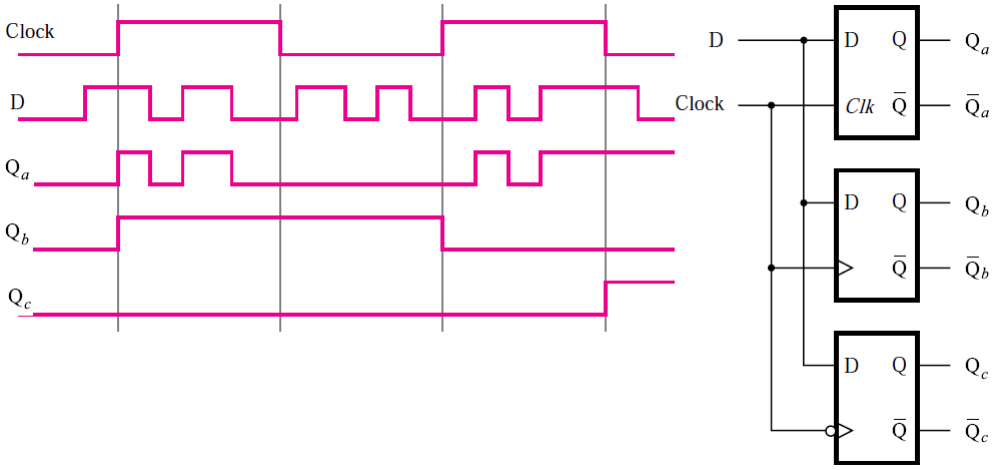


Figure 3.8: Comparison between D-latch and D-flipflop.

Registers

A flip-flop stores one bit of information. When a set of n flip-flops are used to store n bits of information we call this set a **register**. A common clock is used for each flip-flop in a register. Figure **??** shows an example of a right shift register.

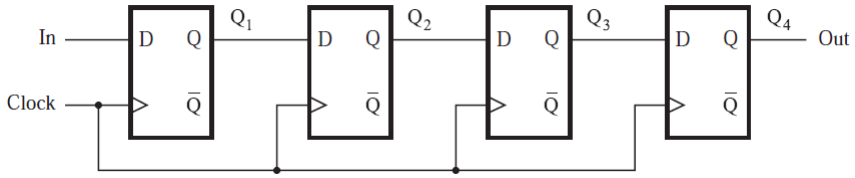


Figure 3.9: Right shift register.

Counters

Counting circuits are used in digital systems for many purposes. They can count the number of occurrences of certain events, generate time intervals for controlling various tasks in a system, track the time elapsed between specific events, etc. Counters can be implemented using a combinatorial circuit and flip-flops.

There are two types of counters: asynchronous and synchronous. Asynchronous counters are simple, but not very fast because of the delays caused by the cascade synchronization scheme.

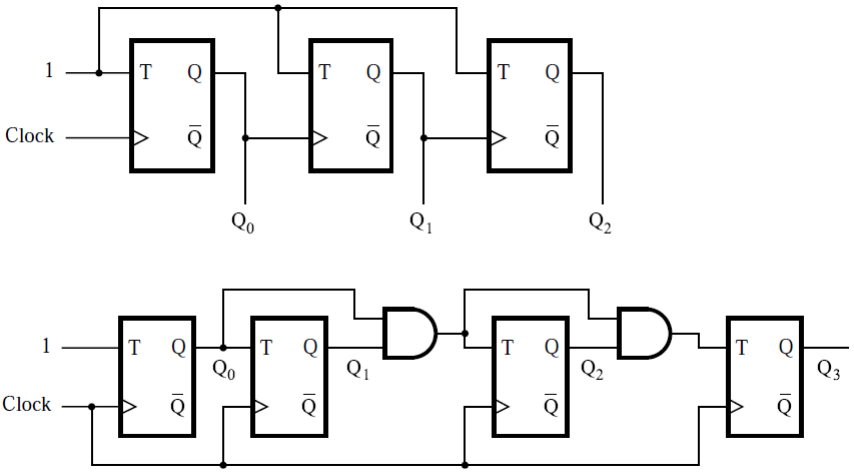


Figure 3.10: Comparison between asynchronous and synchronous modulo-8 counter.

2.2 Synchronous sequential circuits

As shown in figure 3.11, synchronous sequential circuits are realized using combinatorial logic and one or more flip-flops. Synchronization is generally performed by a clock signal *clock*.

The circuit has a set of primary inputs, W, and produces a set of outputs, Z. The outputs of the flip-flops are called the state Q of the circuit. Under the control of the clock signal, the outputs of the flip-flops change state according to the function determined by the combinatorial logic. Thus, the circuit passes from one state to another. To ensure that only one transition from one state to another occurs during one clock cycle, the flip-flops must be of the edge triggered type. They can be triggered by either a rising edge or a falling edge.

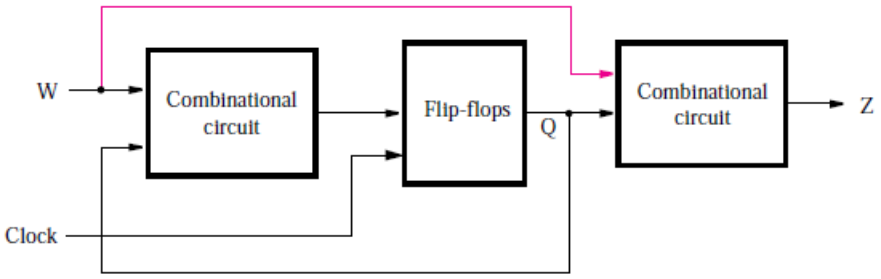


Figure 3.11: General diagram of a synchronous sequential circuit.

The figure indicates that the outputs of the sequential circuit are generated by another combinational circuit, so that the outputs are a function of the present state of the flip-flops and the primary inputs. Although the results always depend on the present state, they do not necessarily have to depend directly on the primary inputs. To distinguish these two possibilities, it is customary to say that the sequential circuits whose outputs depend only on the state of the circuit are of type **Moore**, while those whose outputs depend on both the state and the state of the circuit. primary entries are of type **Mealy**. The diagram of figure 3.11 changes to the diagram of figure 3.12.

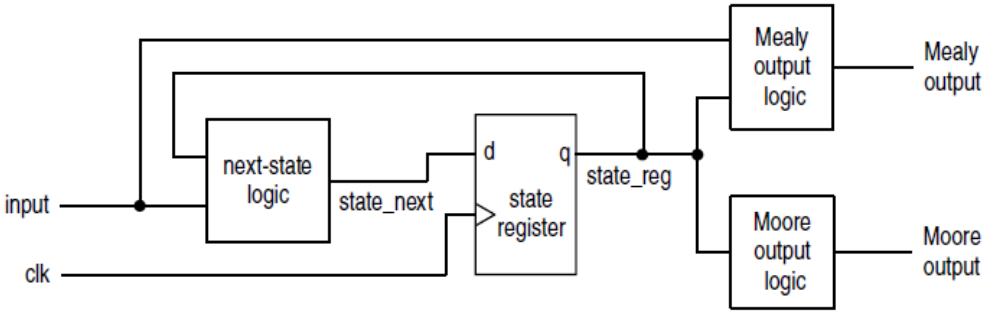


Figure 3.12: Synchronous sequential circuit: the state register block represents the flip-flops which keep the present state.

A *Mealy* type machine has some advantages and disadvantages:

- uses less state because dependence on input signals allows multiple output values in the same state,
- faster,
- transparent to bugs, it passes them directly to the outputs.
- preferable for event signals (*edge sensitive*)

A *Moore* machine is rather preferable for level signals (*level sensitive*).
Sequential circuits are also called finite state machines, **FSM**. This name comes from the fact that the functional behavior of these circuits can be represented using a finite number of states. There are also other names like **automata**.

The simplest example of a synchronous sequential circuit is the D-flipflop. The output Q takes the input D present at each rising edge of the clock. For the rest of the time Q keeps the information in memory until the next rising edge.

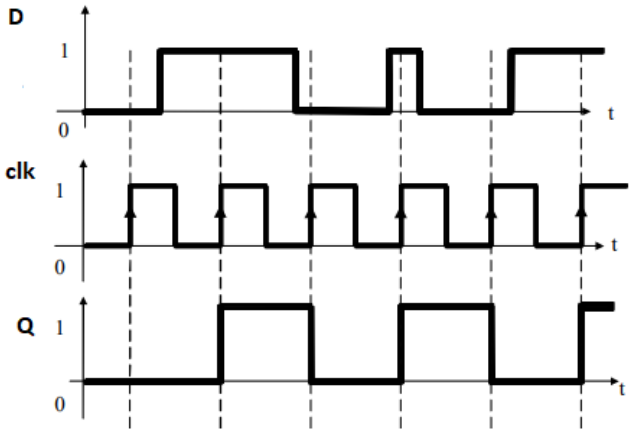


Figure 3.13: Timing of a D-flipflop.

2.3 State diagram

A digital system can involve complex tasks or algorithms that can be expressed as a sequence of actions based on the state of the system and the input signals. Therefore, the design of an FSM starts with a description of all the tasks to determine the number of states required as well as the possible transitions between states. This description can be transcribed in abstract graphical form, or in the form of an **FSM state diagram** or in the form of a **ASM** (algorithmic state machine).

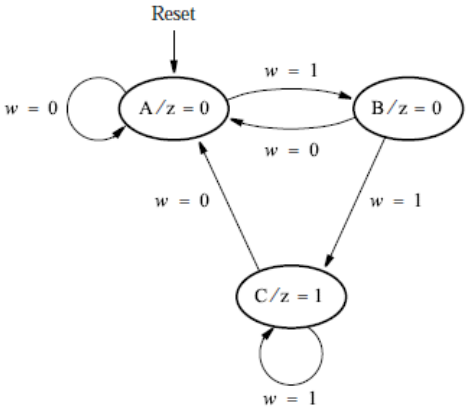


Figure 3.14: FSM diagram of consecutive '1' detector circuit.

Lets take an example of a detector of a binary sequence of figure 3.14. The output $z = 1$ if, for two immediately preceding clock cycles, the input $w = 1$, otherwise, the value of $z = 0$. So the circuit can detect two consecutive '1'.

- The first state is always a starting state. This is the state that the circuit should enter when it is powered or when a reset signal is applied. This state is often called **idle state**. As long as the input $w = 0$, the circuit remains in its starting state.

- The next state arrives as soon as the input $w = 1$. This transition takes place on the next active clock edge. In state B, as in state A, the circuit must hold the value from the output z to 0.
- When a second '1' on the input w is detected, the circuit goes to another state C, in which the output of the circuit z is equal to '1'.
- However, when in state B, the input $w = 0$, the circuit returns again to state A.
- At all times, when *reset* signal is active, the circuit goes to the starting state.

As shown in figure 3.15, an FSM diagram is used to model a state machine in the form of nodes and arcs.

- A node represents a single state.
- An arc represents a **transition** from one state to another. An arc is labeled with a **condition**.
- An output from a Moore automaton is shown inside the state circle.
- An output of a Mealy automaton is placed on the arc with the transition condition.

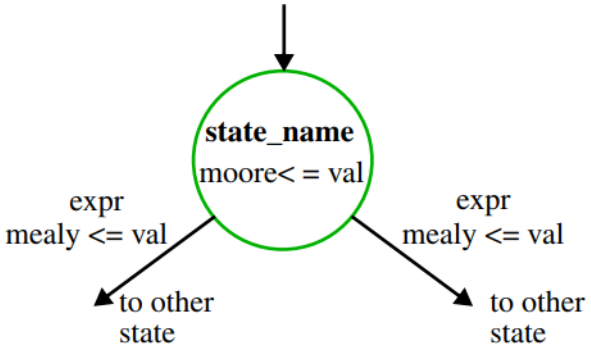


Figure 3.15: Modeling an FSM diagram as set of circles and arcs.

2.4 State table

Although the state diagram provides a description of the behavior of a sequential circuit, to proceed with the implementation, it is convenient to translate the state diagram into a **state table** form as shown in the figure 3.16. The table shows all the transitions from the *present state* to the *next state* for different values of the input signal.

Present state Q	Next state Q^+		Output z
	$\overline{w}$	w	
A	A	B	0
B	A	C	0
C	A	C	1

Figure 3.16: Transition state of the FSM of figure 3.14.

When implemented in a logic circuit, each state A, B and C is represented by a particular assignment where each each two state variable can be implemented by one flip-flop. In the example of figure 3.16, we need two flip-flops to encode the three states, and hence, each of these states is coded on two bits Q_1Q_0. This encoding is carried out in a table called **transition table**, in which we also introduce the flip-flop transitions chosen for the design to achieve the transition from the *present state* Q to the *next state* Q^+.

Present state Q_1Q_0	Next state $Q_1^+Q_0^+$		Output z
	$\overline{w}$	w	
00	00	01	0
01	00	10	0
10	00	10	1
11	X	X	X

Figure 3.17: Transition table or state assignment table for the sequential circuit.

2.5 Example: serial adder

We consider two unsigned numbers $A = a_{n-1} \cdots a_0$ and $B = b_{n-1} \cdots b_0$. Adding A and B gives a sum $S = S_{n-1} \cdots S_0$. We try to design a circuit that will perform the addition in a serial way, bit by bit. The process begins by adding the bits a_0 and b_0. In the next clock cycle, the bits a_1 and b_1 are added, including a possible carry. Figure 3.18 shows the RTL level of this circuit.

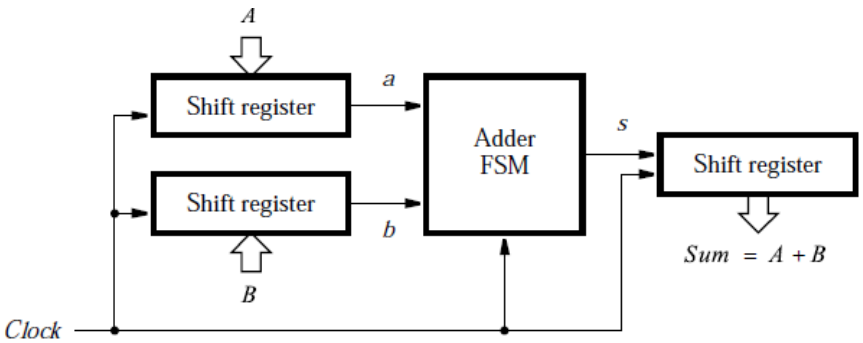


Figure 3.18: RTL level of serial adder.

Even if the addition is a combinatorial operation, this circuit cannot be a combinatorial circuit because different actions will have to be performed depending on the value of the carry of the previous addition. Two states are therefore necessary S_1 where the carry is '0' and S_2 or it is '1', respectively.

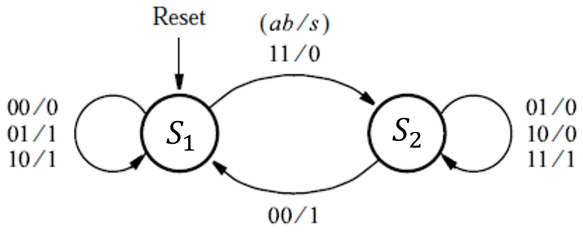


Figure 3.19: FSM of the *Mealy* machine: the s output depends on both the present state and the value of a and b inputs. Each transition is labeled using the notation ab/s, which indicates the value of s for a given pair ab.

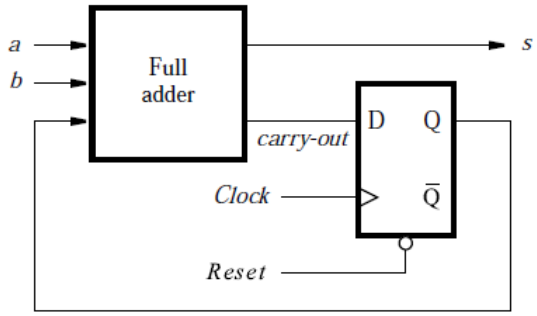


Figure 3.20: Circuit for the serial adder, we need one flipflop for the two states S_1 and S_2.

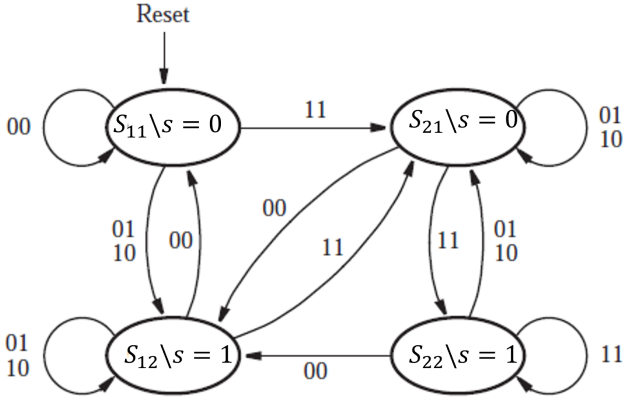


Figure 3.21: FSM of the *Moore* machine: output s depend only on the present state.

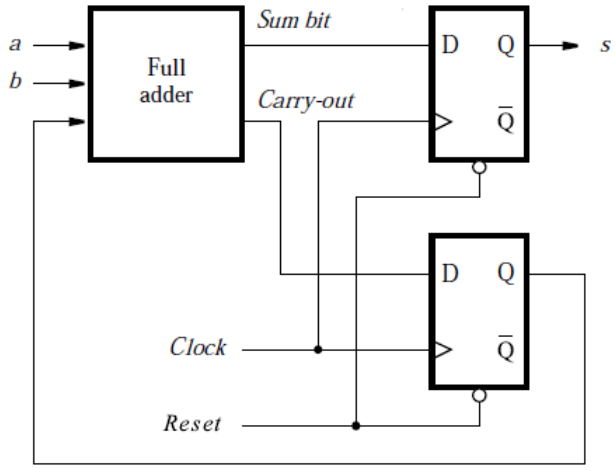


Figure 3.22: Circuit for the serial adder, we need two flipflop for the four states G_0, G_1, H_0 and H_1.

3 ASM

3.1 Description

State diagrams FSM are useful for describing the behavior of sequential machine that have only a few inputs and outputs. For more complex systems, designers often use a different form of representation, called a **ASM**, *Algorithmic State Machine* diagram. An ASM graph is a type of flowchart that can be used to represent state transitions and the generated outputs. The three types of elements used in ASM charts:

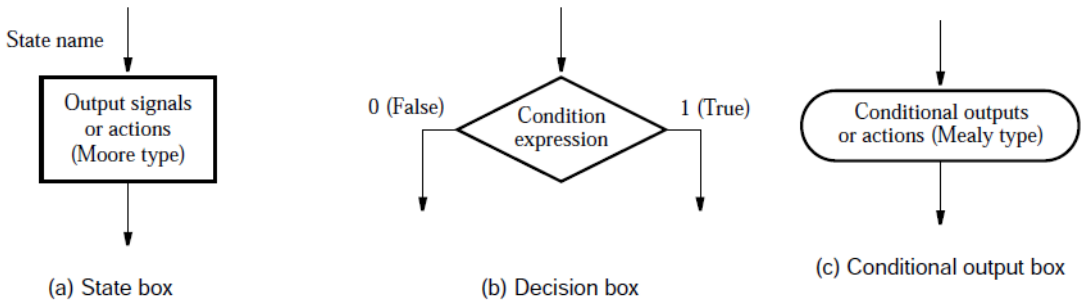


Figure 3.23: Basic Elements used in ASM flowchart.

- *state box*: the state is represented by a rectanglar box. The name of the state is shown outside of the box in the upper left corner. Outputs of type *Moore* are listed in the box. It is customary to write only the name of the signal to be asserted. So, it suffices to write z rather than $z = 1$.
- *decision box*: a diamond box indicates that the specified condition expression should be tested and the output path should be chosen accordingly.
- *Conditional output box*: an oval box indicates the output signals of type *Mealy*.

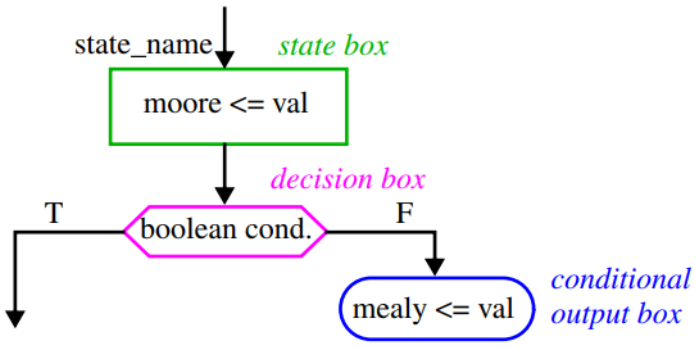


Figure 3.24: Example of the interconnection of elements of an ASM.

ASM flowcharts are similar to *flowchart* diagrams but unlike a flowchart, the ASM includes synchronization.

3.2 Examples

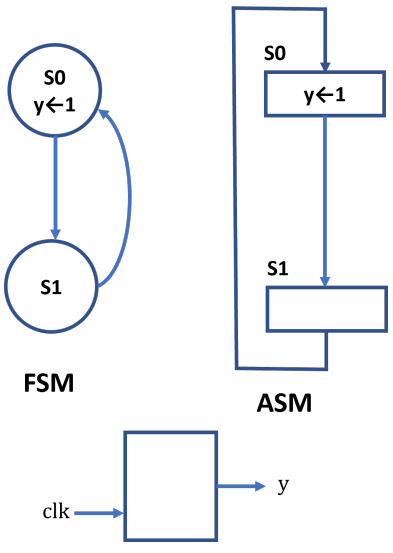


Figure 3.25: Example 1.

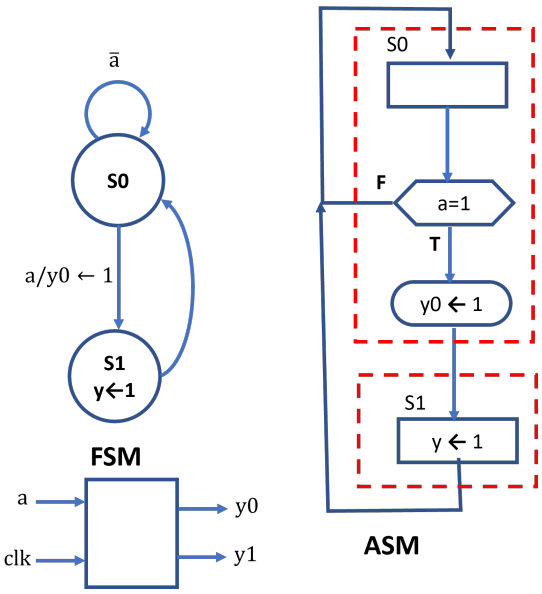


Figure 3.26: Example 2.

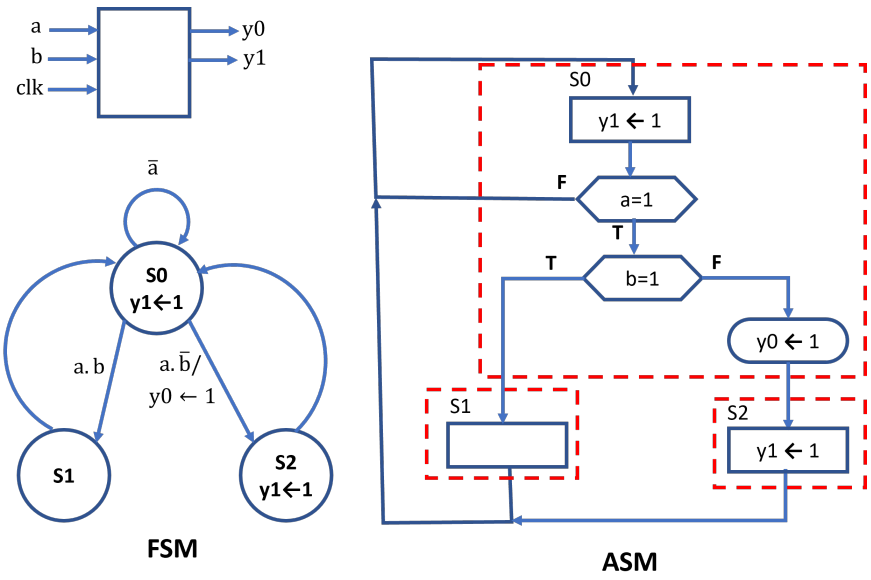


Figure 3.27: Example 3.

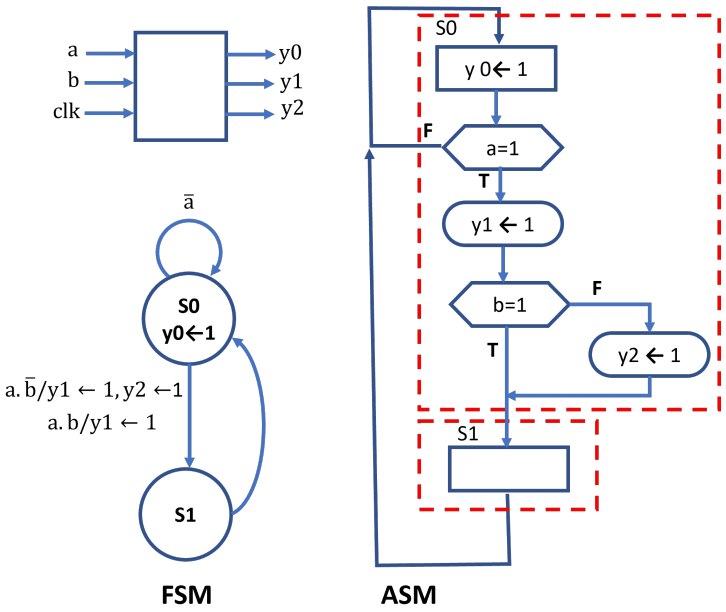


Figure 3.28: Example 4.

Basic rules

- For a given input combination, there is a **unique output path**.
- An output path of an ASM block must always lead to a states box or a conditional output box.

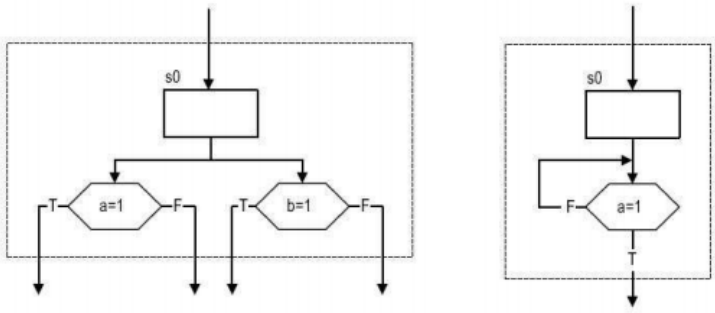


Figure 3.29: Incorrect ASM interconnections: (left): two exit paths from a state box. (right): no exit path when a='0', the path 'T' must goes to another state box or a conditional output box.

4 FSM/ASM and VHDL

4.1 VHDL description

There are two styles to describe an FSM or an ASM in VHDL:

1. **Multi-Segment**: one VHDL code segment for each block,

2. **Two-Segment**: combine next states and output logic into a single segment of VHDL code.

```
entity mem_ctrl is
  port (clk, reset, mem, rw, burst: in std_logic;
        oe, we, we_me              : out std_logic);
end mem_ctrl;

architecture multi_seg_arch of mem_ctrl is
  type states is (idle, read1, read2, read3, read4, write);
  signal state_reg, state_next: states;
begin
  --state register
  process(clk,reset)
  begin
    if (reset = '1') then state_reg <= idle;
    elsif (clk'event and clk = '1') then state_reg <= state_next;
    end if;
  end process;
  -- next-state logic
  process(state_reg,mem,rw,burst)
  begin
    case state_reg is
      when idle =>
        if (mem = '1') then
          if (rw = '1') then state_next <= read1;
          else               state_next <= write;
          end if;
        else
          state_next <= idle;
        end if;
      when write => state_next <= idle;
      when read1 =>
        if (burst = '1') then state_next <= read2;
        else                  state_next <= idle;
        end if;
      ...
    end case;
  end process;

  -- Moore output logic
  process(state_reg)
  begin
    we <= '0'; oe <= '0'; -- default values
    case state_reg is
      when idle  =>
      when write => we <= '1';
      when read1 => oe <= '1';
      ...
    end case;
  end process;
```

```
49
50 -- Mealy output logic
51   process(state_reg,mem,rw)
52   begin
53     we_me <= '0'; -- default values
54     case state_reg is
55       when idle  =>
56         if (mem = '1') and (rw = '0') then we_mem <='1';
57         end if;
58       when write =>
59       when read1 =>
60       ...
61     end case;
62   end process;
63 end mult_seg_arch;
64
65 -- next-state logic and output logic
66   process(state_reg,mem,rw,burst)
67   begin
68     oe <= '0'; we <= '0'; we_me <= '0'; -- default value
69     case state_reg is
70       when idle =>
71         if (mem = '1') then
72           if (rw = '1') then state_next <= read1;
73           else               state_next <= write; we_me <= '1';
74           end if;
75         else
76           state_next <= idle;
77         end if;
78       when write =>
79         state_next <= idle; we <= '1';
80       when read1 =>
81         if (burst = '1') then state_next <= read2;
82         else                  state_next <= idle;
83         end if;
84         oe <= '1';
85       ....
86     end case;
87   end process;
88 end two_seg_arch;
```

4.2 State assignment

Some binary assignments can reduce the complexity of logic blocks. Tables show some coding rule to assign a binary code to a given state.

	Binary assignment	Gray code assignment	One-hot assignment	Almost one-hot assignment
idle	000	000	000001	00000
read1	001	001	000010	00001
read2	010	011	000100	00010
read3	011	010	001000	00100
read4	100	110	010000	01000
write	101	111	100000	10000

Figure 3.30: Assignment representation: binary, Gray, one-hot, almost one-hot.

We can impose one assignment in a VHDL description in two manners:

1. implicitly: we use the **attribute** directive. The standard **1076.6** défines the attribut *enum_encoding*.

2. Explicitly: in the declarative section of states.

```
--implicit assignment

type state is (idle, write, read1, read2,  read3, read4);
attribute enum_encoding: string;
attribute enum_encoding of state: type is "0000 0100 1000 1001 1010
    1011";

--explicit assignment

constant idle: std_logic_vector(3 downto 0):="0000";
constant write: std_logic_vector(3 downto 0):="0100";
```

5 Register Transfer Methodology

Designing a complex digital system based on state machines is difficult, because the number of states to be managed and controlled would be substantial. To overcome this difficulty, digital systems are designed using a modular approach. The system is divided into subsystems performing a given function. Modules are built from simple digital circuits such as registers, decoders, multiplexers, arithmetic elements. The different modules are interconnected to build a datapath.

The functionalities of a digital system are best represented in the form of **algorithms**. These algorithms define a set of registers and the operations performed on the binary information stored in them as shown in figure 3.31. Examples of registry operations are *shift, count, clear, and load*. The registers are supposed to be the basic components of the digital system. The flow of information and processing performed on the data stored in registers are called register transfer operations. A control unit, *FSM*, defines the sequencing and scheduling of the various operations.

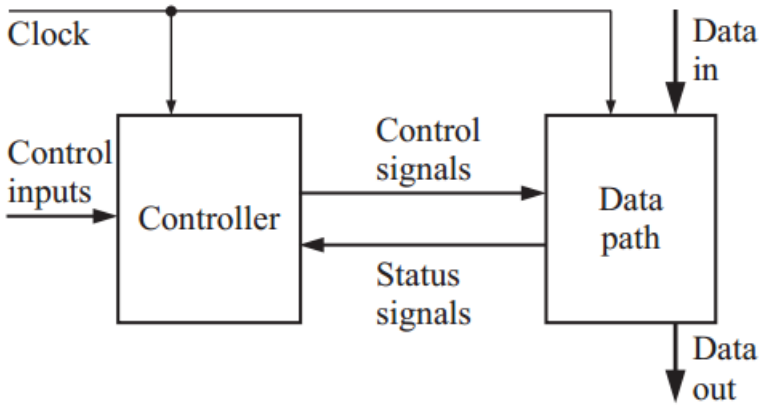


Figure 3.31: General datapath architecture.

RTL methodology, *Register Transfer Level*, is a design process in the form of a sequential sequence of operations and read/write in registers. The designer specifies the processor registers, the data transfers between these registers, the operations to be performed and the control signals to manage these tasks. Each register transfer operation must complete within a clock cycle, which is equivalent to a state of the FSM.

5.1 Micro-operation

A micro-operation is an elementary operation performed on data. A micro-operation specifies the transfer of information from one register to another. This transfer is designated in symbolic form by the operator f as in the following statement:

$$r_{\text{dest}} \leftarrow f(r_{\text{src},1}, r_{\text{src},2}, \cdots)$$

In this form, there are two main types of micro-operations:

1. the arithmetic and logical micro-operations modeled by a generic function f.
2. the transfer of the contents of the source registers $r_{\text{src, i}}$ to the destination register r_{dest},

Example 1:

We consider the instruction $r_1 \leftarrow r_1 + r_2$. This instruction is converted into two micro-operations as follows:

```
r1_next <= r1_reg + r2_reg
r1_reg <= r1_next          -- on the next rising edge of clk
```

In figure 3.32, the two registers are controlled by a clock signal clk. At instant $t = t_0$, each register contains the present value $r_{i,\text{reg}}$. At the next rising edge of the clock, the register r_1 receives new data on its input port d, which is the result calculated by the adder. At the next rising edge of the clock, the result of the addition is available on the output port q of the register r_1. Therefore, micro-operations only take effect during an active transition of the datapath clock signal.

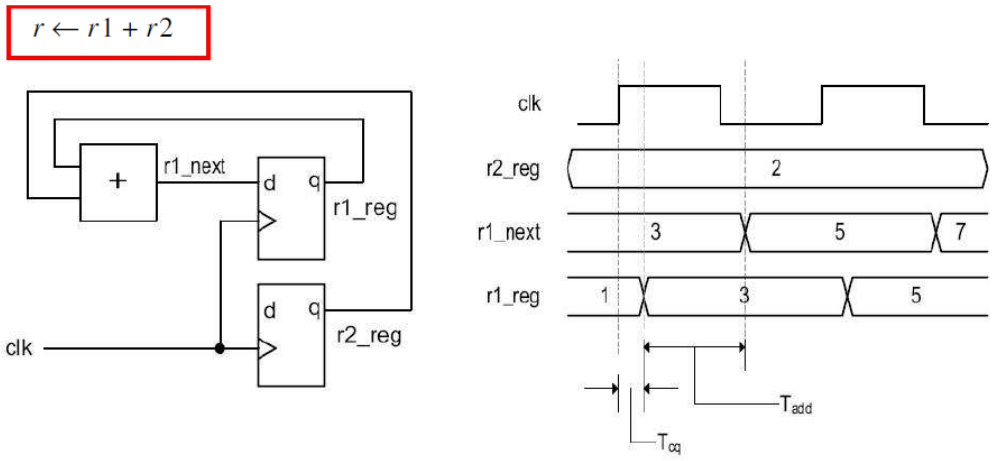
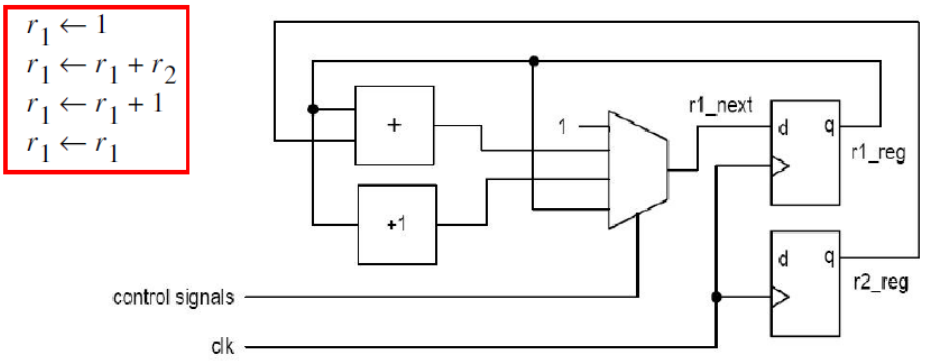


Figure 3.32: Transfer and timing of example 1.

Example 2:



- Since r_1 is the destination register for multiple operations, a MUX is used.
- To derive the control signals, we use an FSM.

5.2 Specific datapath

The goal of designing a **dedicated** or **specific** datapath is to create a tailor-made circuit to deploy a specific algorithm. Each instruction of the algorithm is broken down into micro-instructions to determine register transfers. In this design process, we have to answer some questions:

- how many registers to use and what types, storage, offset or bank of registers?
- what are the functional units to use: adders, multiplexers, decoders, comparators and others?
- what are the functional units that should be shared between two or more operations?

Registers

Consider the assignment statement $A \leftarrow A + c$ where c is a constant. This instruction takes the value that is stored in the variable A and adds the constant c to it to obtain the result of storing the same variable A (a simple accumulation operation). So in this example the datapath must have a register to store the variable A and an adder. The constant c can be wired into the circuit as a binary value.
As we saw at the beginning of this chapter, the value stored in the register is A_{reg}. The result of the addition presents the value to be stored in the register A_{next} as shown in figure 3.33. Storing the result A_{next} in the register is accomplished by activating the control signal Load_{A}, defined by the FSM machine, at the next cycle clock.

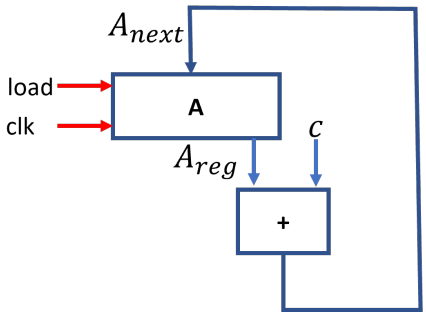


Figure 3.33: Example of a transfer register $A \leftarrow A + c$ in a dedicated datapath.

we now consider the assignment statement $A \leftarrow B + C$ where B and C are two variables. Since we have three variables, we need three registers as shown in figure 3.34.
During the current clock cycle, the adder performs the addition $B + C$, and the result of the adder A_{next} must be available before the end of the current cycle clock. At the next clock edge, A_{reg} is updated by A_{next}.

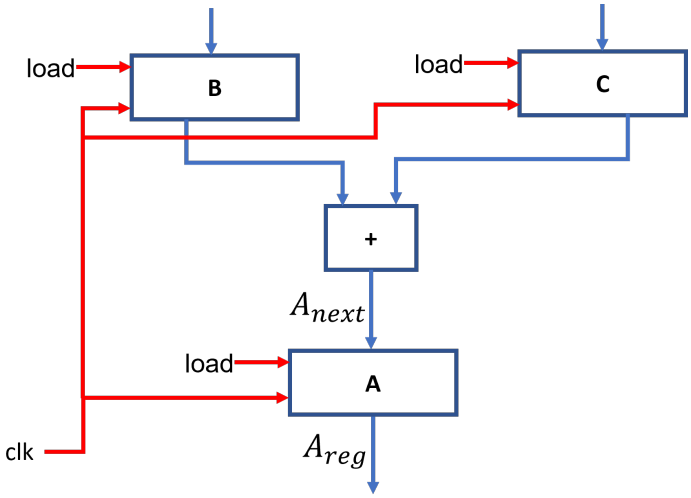


Figure 3.34: Example of a transfer register $A \leftarrow B + C$ in a dedicated datapath.

Multiplexer

We consider subsequently, the two combined transfers $A \leftarrow B + C$ and then $A \leftarrow A + c$. In these transfers we have three variables (A, B, C) requiring three registers and a constant

c which can be directly wired into the FPGA. We combine the two figures 3.33 and 3.34, we obtain the figure 3.35.

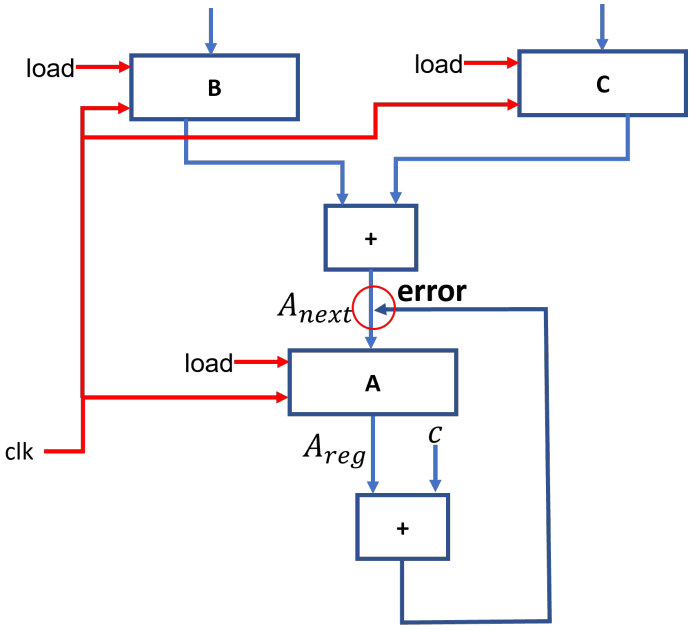


Figure 3.35: Example of two transfer register $A \leftarrow B + C$ and $A \leftarrow A + c$ in a dedicated datapath.

The datapath in the figure 3.35 shows an error in register A wiring. Indeed, a register has only one data input, but the register A requires to switch between two results, that of first transfer $A \leftarrow B+C$ and another of second transfer $A \leftarrow A+c$. To achieve this, the simplest solution is to use a multiplexer as shown in figure 3.36. Here also, the multiplexer has a selection input that must be controlled. Note that these two instructions cannot be executed in the same clock cycle since they share the same register.

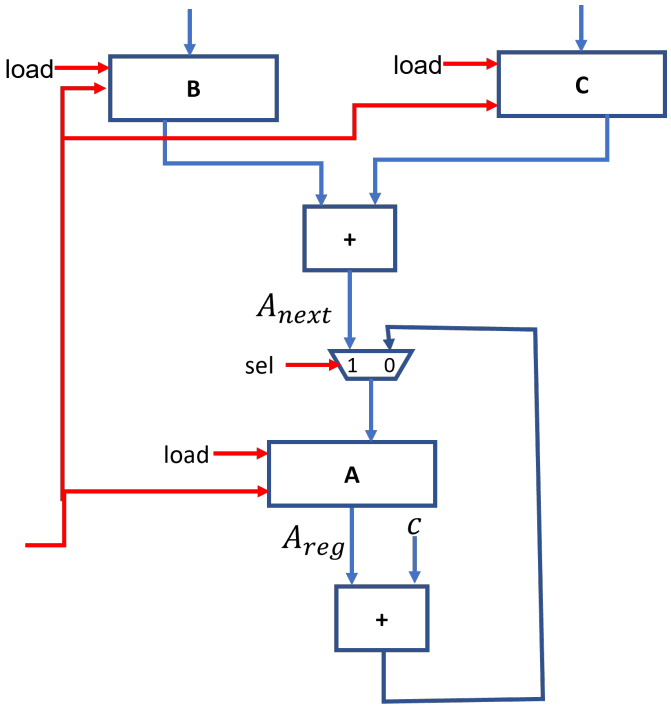


Figure 3.36: Using a multiplexer to resolve the conflict at the register input A.

Exercice

Synthesize a specific datapath for the two transfers $A \leftarrow B + C$ and $A \leftarrow A + c$ by using a single adder.

Three-state buffer

Another case study in which several sources and destinations share the same data bus. In this case, only one source can use the bus at a time. In order to manage access to the bus, Three-state buffers are used to guarantee access to a single source to the bus, as shown in figure 3.37. The other buffers will be deactivated, so the connection will be in high impedance ('Z'). In addition, with the use of a Three-state buffer, several sources can share a bidirectional bus. Note also that the data input and output of a register can both be connected to the same Three-state bus; while the input and output of a functional unit (such as adder or ALU) cannot be connected to the same Three-state bus.

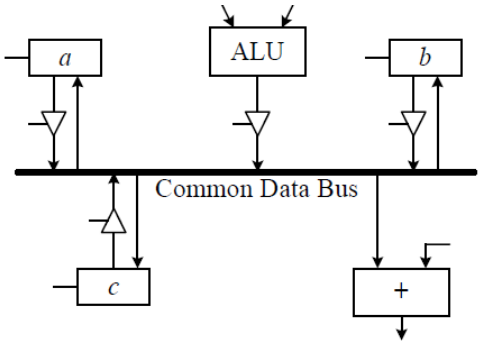


Figure 3.37: Use of three-state buffer for access to the data bus.

Status signal

Although the FSM is responsible for controlling the datapath, the last one must provide conditional test results for the FSM. These results allow the FSM machine to determine its future state corresponding to the next micro-instruction to be executed. The conditional test signals are called *status signals* and they are generally generated by comparators (figure 3.38.a).

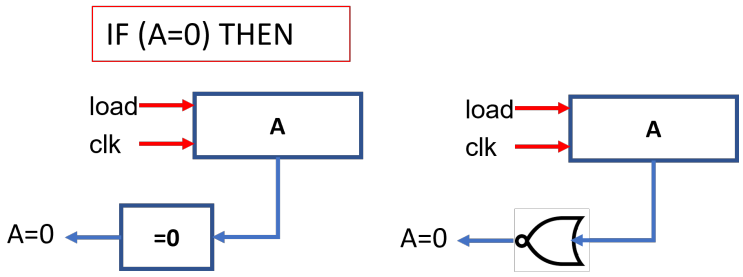


Figure 3.38: Comparator for the generation of datapath status signals to be sent to the FSM machine.

Conditional tests are generally of type *IF (A = 0) THEN*. In this test, the value of the register A is compared with the constant 0 and the comparator output is *TRUE* when the condition ($A = 0$) is true. This comparator can be implemented by a simple NOR gate as in figure 3.38.b).

5.3 Generic Datapath

A generic datapath is not reserved for a specific algorithm. It can be used to deploy various algorithms, as long as it has all required functional units. The idea of using a generic datapath comes from the architecture of micro-controllers and microprocessors. However, compared to a specific datapath, a general data path may contain more functionality than necessary. Therefore, it not only increases the size of the circuit, but also the power consumption.

As shown in figure 3.39, a generic datapath includes an ALU, an n-bit register for data storage, a multiplexer, and a three-state buffer. The input of the operand A of the ALU can come through the multiplexer, either from an external data input, or from a constant (c). The ALU operand B always comes from the contents of the register.

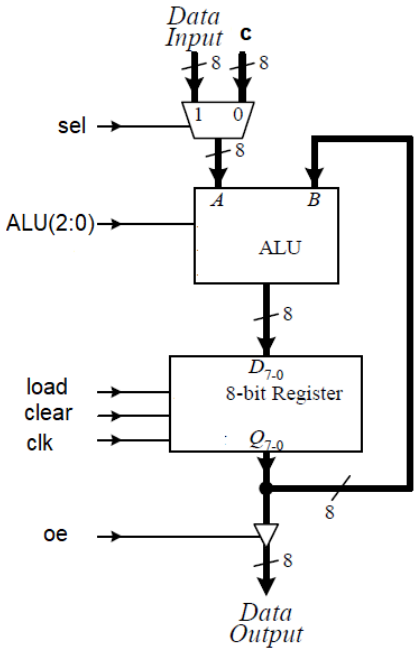


Figure 3.39: Generic Datapath.

The operation of the ALU is determined by the three command lines $ALU_{(2:0)}$, where their combination defines the operation to be performed. The register contains two inputs, *load* for loading and *clear* to reset it to 0 asynchronously. The output of the register can be sent to the external data bus by activating the *OE* control signal of the three-state buffer.

$ALU_{(2:0)}$	operation
000	Pass through ALU
001	A and B
010	A or B
011	A not B
100	$A + B$
101	A - B
110	$A + 1$
111	A - 1

The various control signals for controlling the operation of the datapath form a control word. For example, to load a value from the external data input into the register, we would define the control word as follows:

sel	$ALU_{(2:0)}$	*load*	*clear*	*oe*
1	000	1	0	0

We consider the example of calculating the sum $\sum_{i=1}^{9} i$. The corresponding algorithm is very simple:

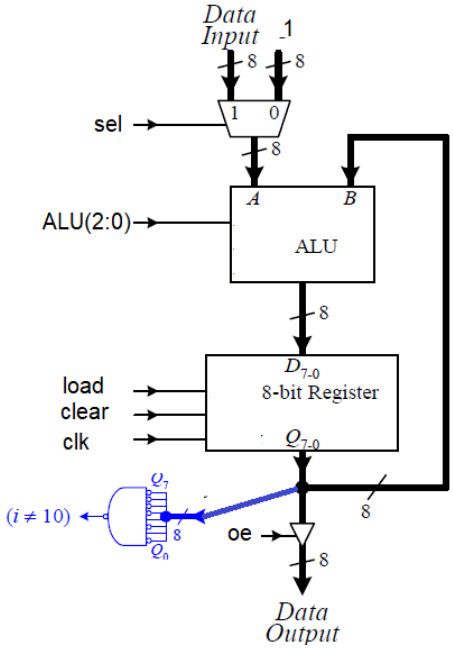
```
--initialisation
i = 0
--operation
while (i < 10) {
  i = i + 1
}
--result
output i
```

We generate a control word to control the datapath for each of the three instructions above.

control word	instruction	*sel*	$ALU_{(2:0)}$	*load*	*clear*	*oe*
1	$i = 0$	x	xxx	0	1	0
2	$i = i + 1$	0	100	1	0	0
3	output i	x	xxx	0	0	1

- **Control word 1**: initializes i to 0 by activating the *clear* signal of the register. The lines *sel* and $ALU_{(2:0)}$ have no impact, and therefore set to state 'X' (don't care). The *load* = *0* signal since there is no need to save a value in the register when exiting the ALU. The same remark for the *oe* signal.
- **Control word 2**: corresponds to the operation $i = i + 1$. The operation code is $ALU_{(2:0)} = 100$, and *sel* = 0 to perform the addition $i = i + 1$.
- **Command word 3**: is used to output the result via the 3-state buffer by setting *oe* = 1.

- Finally, the datapath must supply the condition signal (i <10) to the FSM machine by adding a comparator.



5.4 Generic datapath with register file

We consider the following algorithm of the series $\sum_{n=0}^{N} n$:

```
sum = 0
input n
while (n != 0) {
  sum = sum + n
  n   = n - 1
}
output sum
```

This algorithm requires the use of two variables n to track the current iteration number and sum to save the intermediate sum. In this case, the datapath in figure 3.39 is upgraded to include one or more additional registers. The simplest solution is to use a register bank (register file) like micro-controller and micro-processor architectures.

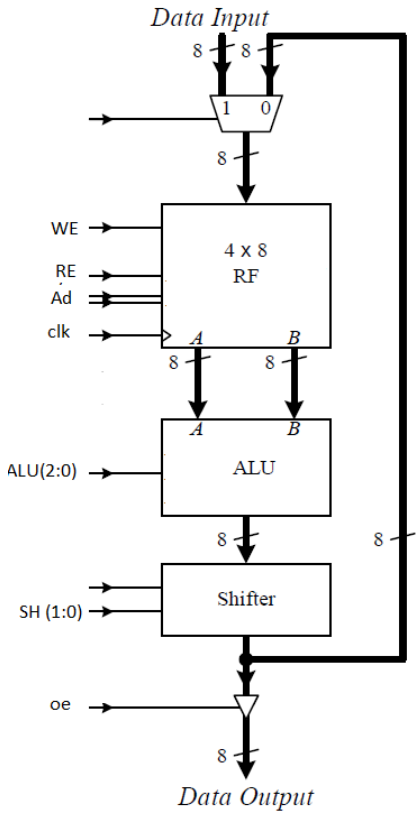


Figure 3.40: Generic datapath with register file (RF).

As shown in figure 3.40, the register file has one *write* port and two *read* ports. To access a particular port, the activation line for that port must be confirmed and the location *address* set. The read ports A and B can be read simultaneously and they are connected to the two input operands A and B of the ALU, respectively. Also, the result of the ALU is passed through a *shift register* which allows more complex operations to be carried out.

$SH_{(1:0)}$	operation
00	Pass
01	shift left
10	shift right
11	rotate right

The following control words can be used to sequence the operation of the datapath for the previous algorithm (here the address *Add* signal represents the address of the 3 registers).

ontrol word	instruction	*sel*	*WE*	*RE*	*Add*	$ALU_{(2:0)}$	$SH_{(1:0)}$	*oe*
1	$sum = 0$	0	1	1	00 00 00	000	00	0
2	input n	1	1	0	01 xx xx	xxx	xx	0
3	$sum = sum + n$	0	1	1	01 00 01	100	00	0
4	$n = n - 1$	0	1	1	01 01 xx	111	00	0
5	output sum	x	x	1	xx 00 xx	000	00	1

6 FSMD/ASMD

FSMD and ASMD are *FSM/ASM with datapath*. These diagrams were developed to clarify the information inclosed in FSM/ASM diagram. This versions provide an efficient tool for designing a control unit, *FSM*, for a particular datapath.
An *ASMD* diagram differs from an *ASM* diagram in two important ways:

1. an *ASMD* diagram lists register transfers in a status box,
2. Conditional output box can also includes register transfers,

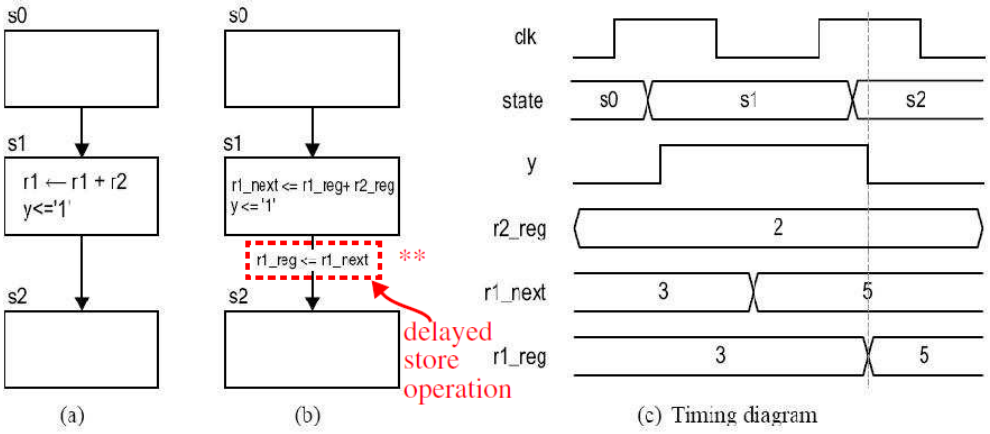


Figure 3.41: The new value of r_1 is available when the FSM exits state s_1

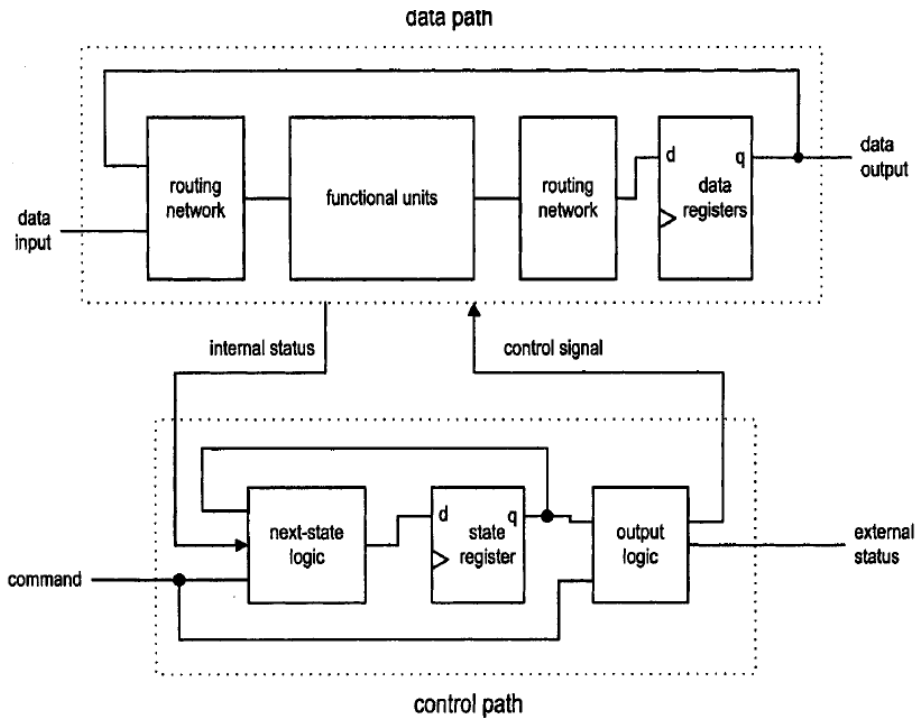


Figure 3.42: Block diagram of an FSMD. The datapath performs all Register Transfer operations: (1) data registers, (2) operations, (3) routing. The control path: an FSM which sequences the logic of the transitions between states and defines the logic of the output signals.

Chapter 4

Exercises

1 VHDL

1.1 Circuit description

Write a VHDL code for the following circuits:

- half-adder, full-adder, 4-bit adder and its generic version n-bit
- D flip-flops, shift register, counter.

1.2 Types

Write a package containing type declarations for:

- short 8-bit non-negative integers,
- real numbers between -1.0 and +1.0,
- electric current with the units of nA, μA, mA and A,
- traffic light colors.

1.3 Type attributs

Consider the following *subtype* declarations:

```
subtype pulse_range is time range 1 ms to 100 ms;
subtype word_index is integer range 31 downto 0;
```

What are the values returned by the followxing attributes 'left, 'right, 'low, 'high et 'ascending?
Consider the following statement:

```
type state is (off, standby, active1, active2);
```

What are the values returned by the followxing attributes

```
state'pos(standby)
state'val(2)
state'succ(active2)
state'pred(active1)
state'leftof(off)
state'rightof(off)
```

1.4 Arrays

1. Create a new type named *interval* from the unconstrained array type whose elements are of type *time_vector*, indexed by *positive* values.

   - Declare a *sample1* variable of this type to save 100 elements of 20 values.
   - Creat a *subtype*, named *interval_4*, representing an array of 4 elements without constraint starting from the base type *interval*.
   - Declare a variable *sample2* using the previous *subtype* where each element has 10 sub-elements.

2. Write a ROM entity modeled at an RTL abstraction level. The ROM has an input *address* of type *address_index*, which is a 16-bit integer. The output *data* is of type *std_logic_vector* of 7 bits. Initialize the content with numbers of your choice.

3. Herein some array decelration, analyze them:

   - Set 1:

```
type t_Memory is array (0 to 127) of std_logic_vector(7
   downto 0);
signal r_Mem : t_Memory;

type t_Integer_Array is array (integer range <>) of integer
   ;
variable r_Integers : t_Integer_Array(0 to 15);

type t_Data is array (0 to 3) of std_logic;
signal r_Data : t_Data := (Bit1, Bit2, Bit3, Bit4);

type t_Multiplier is array (0 to 2) of real;
signal r_Multiplier : t_Multiplier := (0.25, 0.5, 0.75);

type t_Five is array (0 to 4) of bit_vector(15 downto 0);
signal r_Calc : t_Five := (others => (others => '0'));

type t_Row_Col is array (0 to 3, 0 to 2) of integer range 0
    to 9;
signal r_Number : t_Row_Col;
r_Choice <= r_Number(0, 1);
r_Number(3, 2) <= 9;
```

2 FSM and datapath

2.1 Rising edge detector

It is a circuit which generates a short pulse of one clock cycle when the input signal goes from '0' to '1'.

- Machine Moore:
  - Draw the FSM diagram of this circuit using a Moore type machine.
  - Convert the FSM diagram to an algorithmic diagram ASM.
  - Draw the timing diagram.
- Repeat the same question for a Mealy-type machine.
- Compare the two machines.
- Write the VHDL code for the two types of machine.

2.2 Fibonacci sequence

It is the sequence of numbers 0, 1, 1, 2, 3, 5, 8, 13, 21, ..., given by the following equation:

$$f(i) = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ f(i-1) + f(i-2) & \text{if } i > 1 \end{cases}$$

The calculation of this sequence is done iteratively. This approach requires two registers reg_1 and reg_2 to store the last two calculated values, $f(i-1)$ and $f(i-2)$ as well as an index register reg_n to follow the number i of iterations. In addition to the input i and output f signals, the system has a control signal $start$, which signifies the start of the calculation, and two status signals $ready$ (ready to calculate) and $done$ (end of calculation).
Draw the FSM and ASM diagrams.

2.3 Homework

Manchester encoding is a baseband encoding technique for transmission in local area networks. In this coding, a '0' bit is represented by a rising edge transition of the signal to be transmitted. A '1' bit is represented by a falling edge transition of the signal to be transmitted. The figure below shows an example of coding.

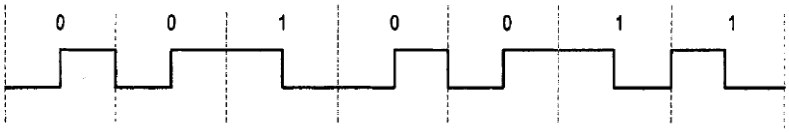


Figure 4.1: Exemple de codage Manchester.

Suggest an FSM diagram and the corresponding VHDL description.

2.4 Unsigned multiplication

By using a specific datapath, implement the multiplication of two unsigned numbers A and B. Find the control signals corresponding to this datapath. Below, you are given the multiplication algorithm.

```
prod = 0
INPUT A
INPUT B
while (B != 0){
  prod = prod + A
  B = B - 1
}
OUTPUT prod
```

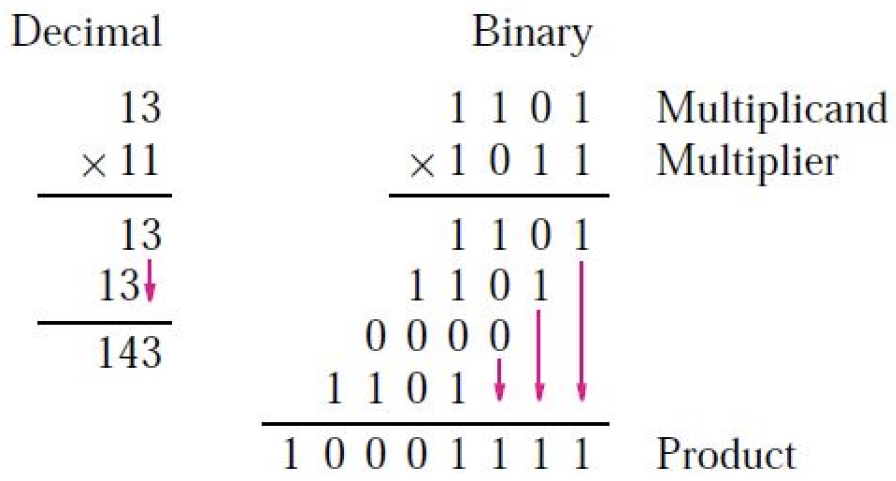
2.5 Greatest Common Divisor

Propose a specific datapath with its control signals to solve the GCD problem (Greatest Common Divisor). Below, we give you the algorithm.

```
while (X != Y){
  if (X < Y) then Y = Y - X;
  else            X = X - Y;
  end if
}
```

2.6 Unsigned multiplication

By using a specific datapath, implement the multiplication of two unsigned numbers A and B. Find the control signals corresponding to this datapath.



```
P = 0
for i=0 to n-1 do
  if bi=1 then
    P=P+A
  end if
  Left-shift A
end for
```

2.7 Mean value

Implement a circuit that computes the mean value M of a n-bit k numbers loaded into registers $R_0, \cdots, R_{k-1}$. For each iteration, compute the cumulative sum Sum and implement a division to get the mean value $M = Sum/k$.

```
Sum = 0
for i=k-1 downto 0 do
  Sum=Sum+Ri
end for
M=Sum-k
```

2.8 Sorting

Let given k unsigned number of n-bit loaded into registers $R_0, \cdots, R_{k-1}$. We look for implementing a circuit for sorting in ascending order. The algorithm is based on the sliding window where we search the most small number in a given window, and the result is saved in register R_i for $i = 1, 2, \cdots, k-2$.

```
for i=0 to k-2 do
  A=Ri
  for j=i+1 to k-1 do
    B=Rj
    if B<A then
      Ri=B
      Rj=A
      A=Ri
    end if
  end for
end for
```

2.9 Parity

Implement a circuit by using an 8-bit specific datapath for the following algorithm. Use only on adder and substracter.

```
w = 0, x = 0, y = 0
INPUT z
while (z != 0) {
  w = w - 2
  if (z est impair) then x = x + 2
  else                   y = y + 1
```

```
7     end if
8     z = z - 1
9 }
```