uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Livio Dancea
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.Sc. (Systems Science)
GRADE / DEGREE

Systems Science
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

New Architecture to Implement Product Terms in Complex Programmable Logic Devices

TITRE DE LA THÈSE / TITLE OF THESIS

Professor Voicu Groza
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Professor Wail Gueaieb

Professor Amiya Nayak

Gary W. Slater
Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

New Architecture to Implement Product Terms in Complex Programmable Logic Devices

Livio Dancea,

B. A. Sc., University of Ottawa

Thesis submitted to the Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the degrée in Master's of Applied Science

uOttawa

L'Université canadienne
Canada's university

School of Information Technology and Engineering

University of Ottawa

Canada

Your file Votre référence
ISBN: 978-0-494-18410-3
Our file Notre référence
ISBN: 978-0-494-18410-3

Dedicated to Tata

New Architecture to Implement Product Terms in Complex Programmable Logic Devices

written by: Livio Dancea

Submitted for the degree of

Master's of Applied Science

Department of Systems Science

May 2006

The final copy of this thesis has been examined by the reviewing committee and both the content and the form meet acceptable presentation standards of scholarity work in the above mentioned discipline.

Abstract

Nowadays, the vast majority of programmable logic devices cannot be partially re-programmed at run-time, preventing them to support real-time reconfigurable computing applications. A new family of computational logic blocks having a dynamic reconfigurable structure is introduced in the thesis. The presented architecture implements any combinational circuit with multiple outputs. The input/output relation of these circuits is expressed by logical equations in the sum-of-product form. Each product term is translated into context words which characterize the logical behaviour of the circuit and are arranged in an array of registers. A development software environment generating necessary data for programming the circuit is given. Finally an implementation of the product terms method on an existing reconfigurable device is given to demonstrate proven advantages of the work.

Declaration

The work in this thesis is based on research carried out at the University of Ottawa, Canada. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work, unless referenced to the contrary in the text.

Acknowledgements

I would like to thank my thesis supervisor Dr. Voicu Groza for his ongoing support and great assistance with this thesis. His bias towards the classical methodology helped me garner the tools necessary to defend the emerging topic. Special thanks go to my family for all their support during this period. It is always comforting to know that I have people backing me in my whatever endeavors I may partake in.

Altera, Incorporated and the Canadian Microelectronics Corporation (CMC) are thanked for their donations of hardware and software, respectively.

Contents

List of Figures

List of Figures

List of Tables

Chapter 1

OVERVIEW

As custom semiconductor development costs continue to escalate, some market segments are focused on taking advantage of the benefits offered by programmable technologies.

1.1 Evolution of Digital Circuit Implementations

Over the past decades, the process of designing digital hardware has significantly changed. Traditionally, a designer had to connect many standard Small-Scale Integrated (SSI) and Medium-Scale Integrated (MSI) chips together in order to build a circuit on a board [21],[22],[23]. These chips had low gate density, were referred to as building blocks for complex circuits and implement frequently used functions in two family types of logic circuits. Logic circuits expressing the relationship between inputs and outputs only by logic equations are defined as *combinational logic* circuits. Typical examples are decoders, encoders and multiplexers. Logic circuits whose outputs depend upon both the current circuit inputs as well as previous values

of circuit states are known as *sequential logic* circuits. Counters and registers are just a few examples of circuits also referred to as circuits with feedback equations.

As integrated circuit technology progressed, the density of elementary gates doubled on a single chip approximately every sixteen months.[1] Due to this astounding progress rate, the mentioned design practice of complex hardware structures became inefficient primarily because the area covered by these low functionality chips exceeded several times the available space for the proposed hardware structure. To counter these limitations, new solutions have been developed with several tenth of millions transistors embedded on a single unit.

One approach has been to fully develop Very Large Scale Integrated (VLSI) devices at transistor level. This requires several years of engineering prototyping to successfully build up custom VLSI devices but as a result, optimal performances are reached. Due to its high costs, this approach focuses on very large volume sale such as microprocessors or blocks of Random Access Memory (RAM).

A second solution has been the semi-custom design approach known as Application Specific Integrated Circuits (ASIC) [27]. In this solution it is possible to design a custom chip from arrays of pre-manufactured logic cells. A single logic cell can be implemented by a few gates and/or a memory element. To build the desired circuit, the designer customizes the interconnection pattern to be implemented on the layout of the integrated circuit. Then, a final manufacturing action is necessary to interconnect the logic cells and is done by creating custom photographic masks. Since an ASIC development requires custom intervention by the manufacturer, ad-

[1] Informally known as Moores Law

ditional time (several months) and increased costs are involved. Any design error encountered on the chip will lead to further development delays and inevitably to augmented costs.

As an alternative to devices with fixed functionality, programmable logic devices were introduced in the early 80's [4],[10]. These chips have a reconfigurable hardware structure which enables them to implement a wide range of different logic circuits with relatively no additional hardware cost. Reconfigurable devices contain in their architecture a large number of programmable switches that allows internal gates to be connected in several different ways [15],[16],[17]. A particular bit stream reconfigures the programmable switches with desired functionalities. Two families of programmable logic devices requiring programming are frequently used today - Complex Programmable Logic Devices (CPLD) and Field Programmable Gate Arrays (FPGA). The performance tradeoffs between full custom design, ASICs, CPLDs, and FPGAs are permanently improving with the delivery of new generation of devices and of design tools. The actual performance tradeoffs of different technologies is shown in Figure 2.1 [19].

1.2 Motivations

Rapidly changing markets and new technologies are subverting traditional system design paradigms. Nowadays, a common procedure is to prototype a system with programmable logic devices (PLDs) and then to redesign using ASICs as soon as possible for cost reduction. The idea is to leverage the flexibility and development time advantage of programmable logic until the design is stabilized and then to

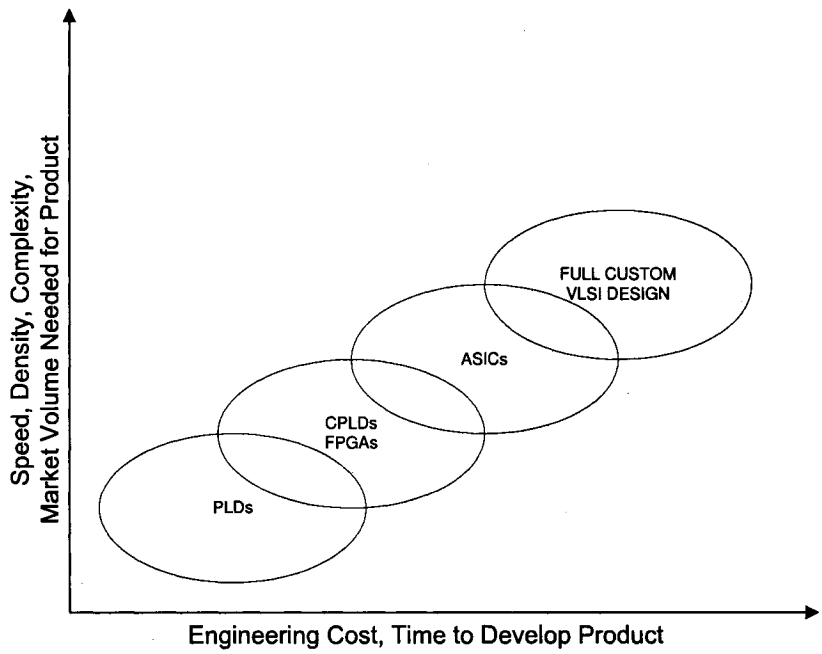

Figure 1.1: Digital Technological Tradeoffs

convert the design to a lower-unit cost. The primary motivation in converting the PLD design to one based on an ASIC is unit cost. ASICs, in general, are available at lower unit cost than PLDs.

In the past, programmable logic had lower density, lower performance, and more limited in capabilities. In some market segments, characterized by relatively mature technologies, established standards, and a stable competitive environment, the PLD-to-ASIC design strategy is still optimal today. For rapidly changing markets, however, this strategy becomes questionable. For many advanced electronic products such as plasma televisions, home networking hardware, and digital audio/video equipment the market is anything but stable. Baseline standards for advanced products are ever-changing, features demanded by customers shift constantly, new technological capabilities are continually being introduced, and competitors are al-

ways trying to gain a better market position resulting in many short-lived products. These volatile market pressures give a significant competitive advantage to flexible and first-to-market products. In these markets, ASICs with their long lead times, high initial costs, high minimum order quantities, higher risks, and inflexibility are not a compelling solution. ASICs usually do hold the unit cost advantage over PLDs, but this cost differential is shrinking.

Two families of programmable logic devices are used today; CPLDs and FPGAs. These logic devices may be reprogrammed at different points of the life cycle since they are not restricted to a particular function. FPGAs and CPLDs are programmed by turning on switches which make connections between circuit nodes and the metal routing tracks. The connection may be made by a transistor switch (which are controlled by a programmable memory element) or by an antifuse. The transistor may be controlled by an volatile SRAM cell or a non-volatile EEPROM flash cell. Nonetheless, these programmable devices need to have the configuration loaded. This reprogramming action takes an amount of time varying from hundreds of milliseconds to tens of seconds. The motivations to analyze and implement a new architecture for CPLDs are presented below.

1. Short-term motivations:

   - build an autonomous system, precisely the logic structure of a sequential circuit with clock input only using one of existing programmable logic device (FPGA) as a proof of concept for the new architecture;
   - prove that the presented new architecture does not need any programmable switch at nanocell level;

- prove that the new architecture does not require reconfiguration of CPLDs when the command of changing the implemented logic circuit is triggered;
- prove that in this new architecture, the speed of changing the implemented circuit reaches it highest limit possible, precisely only a clock pulse and does not have any restrictions on the number of write cycles.

1. Long-term motivation:

- build an actual VLSI device based on the new architecture;
- show that this new architecture device is superior in speed and density at components level versus existing programmable devices
- show that the new proposed CPLDs can change the classical architecture of some future computers

1.3 Outline of this Thesis

The remainder of this thesis is organized as follows. In Chapter 2, a basic introduction to the technical details of various reconfigurable architectures is presented. Chapter 3 defines two solutions to indirectly implement product terms. Chapter 4 presents the proof of concept on an existing reconfigurable technology.

Chapter 2

SYNOPSIS ON

RECONFIGURABLE CIRCUITS

Programmable integrated circuits are known in the art and include programmable logic devices (PLDs), Programmable Array Logic (PALs) and Programmable Logic Arrays (PLAs). A brief introduction will be given for each of these programmable circuits having an input AND logic plane followed by an OR logic plane.

2.1 Programmable Logic Array

The first Programmable Logic Device to be developed was the Programmable Logic Array [1],[6]. The general architecture of a PLA is presented in Figure 2.1. A PLA is based on the idea that any logic equation can be expressed in the sum-of-product form. The PLA's inputs are connected to inverters making the true and the complemented values available for use. The inputs to the gates in the AND plane were traditionally linked through individual fuses and later through programmable

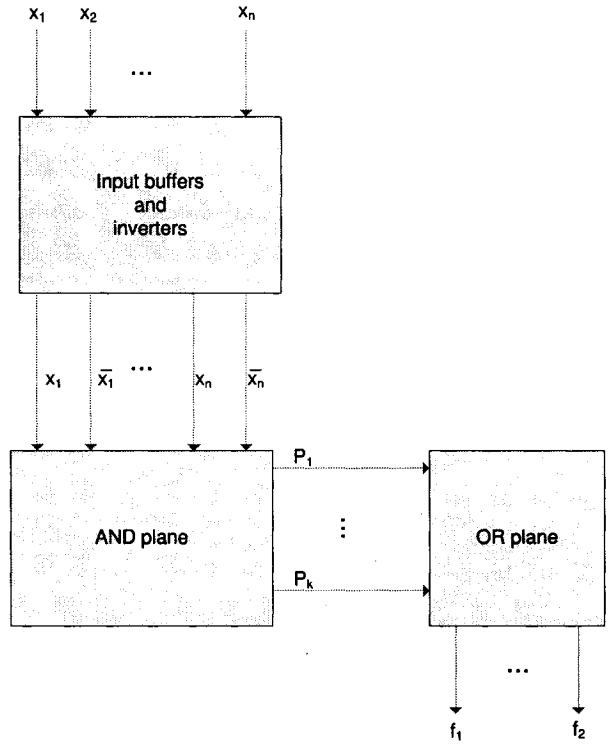

Figure 2.1: General architecture of a PLA

switches. This allows selecting, for each product term, the logical value as they appear in the sum-of-product equations. The outputs of product terms serve as inputs to the second plane formed by OR gates producing the logic value of output functions. The first layer of AND gates is linked to the second layer of OR gates through another group of programmable switches. Therefore, each PLA's output can be configured to realize any hardware structure expressed by a group of sum-of-product (SoP) equations, limited only by the number of inputs, the number of outputs and number of product terms.

A programmable switch has two transistors connected in series, precisely a NMOS transistor and an Electrically Erasable Programmable Read Only Memory (EEPROM) transistor [9]. When the switch is in the original unprogrammed state,

the EEPROM transistor is turned on and acts like a normal NMOS transistor. Programming an EEPROM transistor is accomplished by applying a threshold voltage, which implies a large current flow through the transistors channel. This current causes the Fowler-Nordheim tunneling [9], in Electrically Erasable Programmable Read-Only Memory which some of the electrons are trapped in the channel. When a switch is programmed, the electrons that are trapped prevent other electrons from entering the channel and consequently acts as open switch. Once an EEPROM transistor is programmed, it remains in the programmed state permanently. To erase a programmable switch a voltage that is of opposite polarity must be applied. The EEPROM transistor returns to its original state and again acts like a normal NMOS transistor. Based on Boolean algebra, AND-OR network can be replaced by NAND-NAND or by NOR-NOR network. An example of NOR-NOR PLA implementing a SoP equations and using EEPROM technology[1] is depicted in Figure 2.2.

A simple combination circuit, defined by equation 2.1 and implemented on a small PLA is shown in Figure 2.3. The implemented combinational circuit has four inputs three outputs and five product terms. Only active AND gates and active OR gates are presented.

$$\begin{aligned} X &= a \cdot b + c \cdot d; \\ Y &= a \cdot b + \bar{c} \cdot d; \\ Z &= \bar{a} \cdot \bar{b} \cdot \bar{c} + a \cdot b + \bar{c} \cdot \bar{d} \end{aligned} \tag{2.1}$$

Each AND gate in the AND plane of the simple PLA has eight inputs, related to

[1] circuits known as GALs and made by Lattice

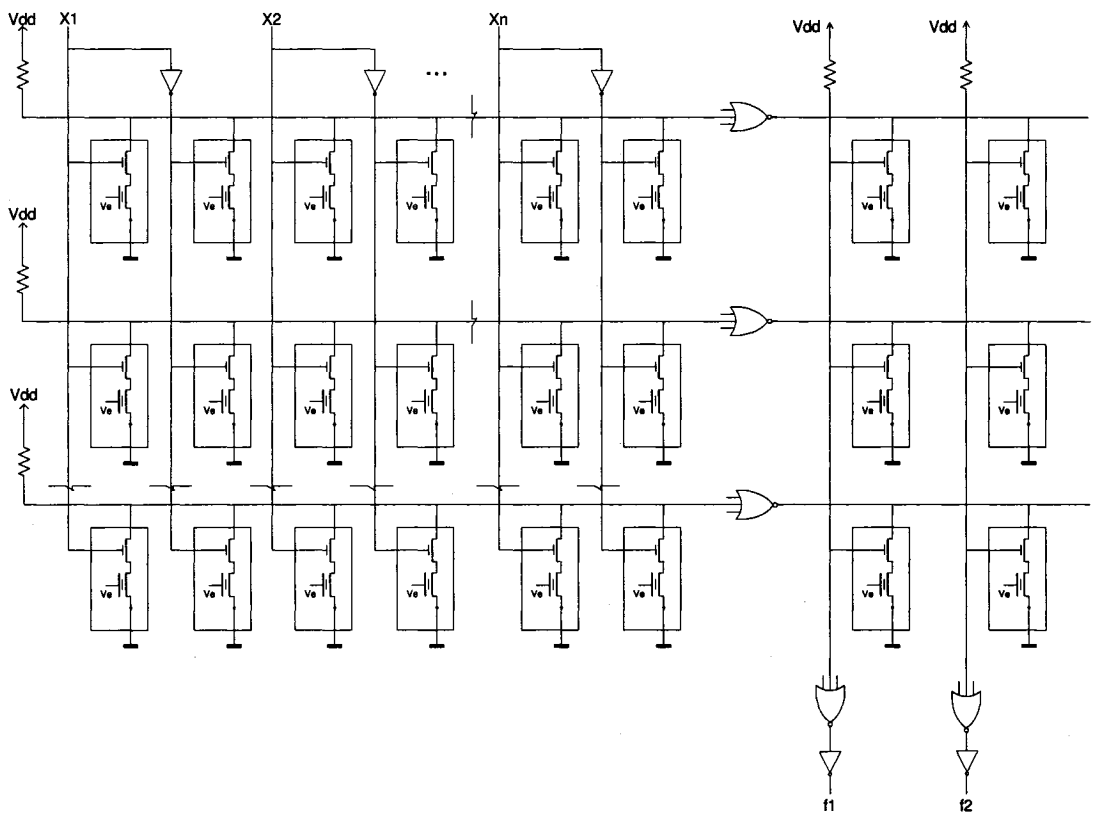

Figure 2.2: A NOR-NOR PLA used for sum-of-products

the true and complement value of the four input variables. Every connection to an AND gate is programmable. For instance, if the true value of an input variable is a component of the considered product term (for instance a) the corresponding programmed switch is marked with 1 and a line shows its connection to the matching input of the AND gate. In the same time, the programmed switch of the complemented value of considered variable (in our case not a) is marked with 0 and a broken line is shown to the matching input of AND gate. If an input variable is not a component of the considered product term, both programmed switches (for instance b and not b) are marked with 0 and broken lines are shown to the matching inputs of AND gate. The circuitry is designed such that any unconnected input (electrically it is at high impedance) does not affect its output. The same observations can be made for the switches that serve the gates in the OR plane. A product term that is a constituent of several equations (outputs), as the product term a.b in the considered example, is materialized only once. Figure 2.3 shows also the values of output functions for the following values of inputs: $a = 0$; $b = 0$; $c = 0$ and $d = 1$.

2.2 Programmable Array Logic

The Programmable Array Logic devices are similar to PLA devices. The inputs to the gates in the AND plane are linked through individual programmable switches as was mentioned for PLA devices. This allows, for each product term, the selection of logical values of variables as they appear in the sum-of-product equations. The PAL devices have only the AND plane programmable and the outputs of AND gates have a fixed connection to the unique OR gate that materializes an independent

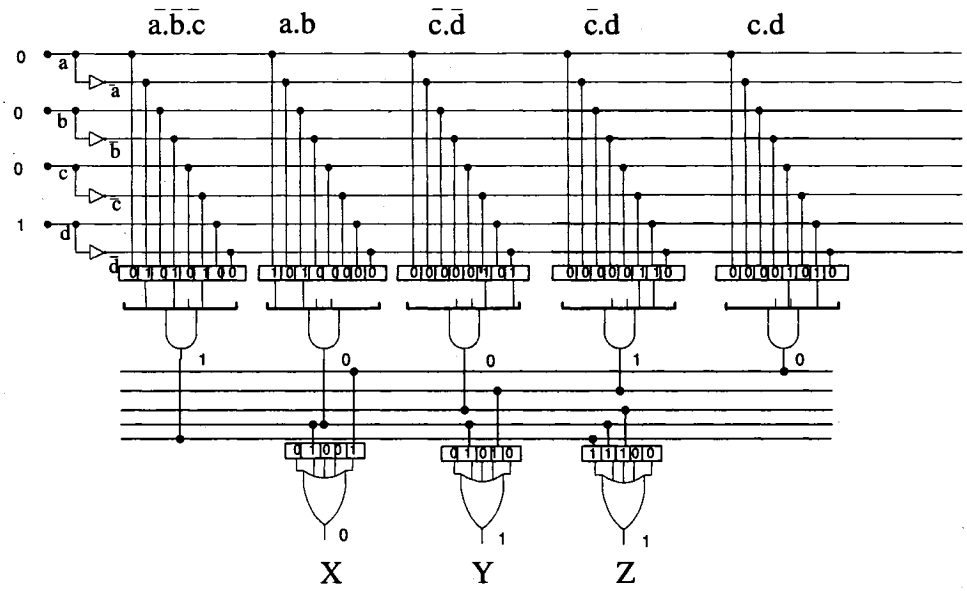

Figure 2.3: Gate level of a small PLA device for multi-output circuit defined by equation 2.1

logic function of product terms. Usually, a PAL device is developed to implement a group of sum-of-product equations, so it contains an OR plane with several fixed OR gates. Each fixed OR gate has its fixed number of independent self-programmable AND gates. A product term redundant in several equations (outputs) is materialized several times, once for each logic independent equation. Therefore, normally for the same group of logic equations, the number of programmable AND gates are higher in a PAL device. However PAL devices offer better performances (higher speed and fewer switches on chip) and are less expensive than PLA devices which make the PALs the preferred solution for practical applications.

A small PAL device is shown in Figure 2.4, implementing a combinational circuit defined by the following single logic equation:

$$Z = \bar{a} \cdot \bar{b} \cdot \bar{c} + a \cdot b + \bar{c} \cdot \bar{d} \tag{2.2}$$

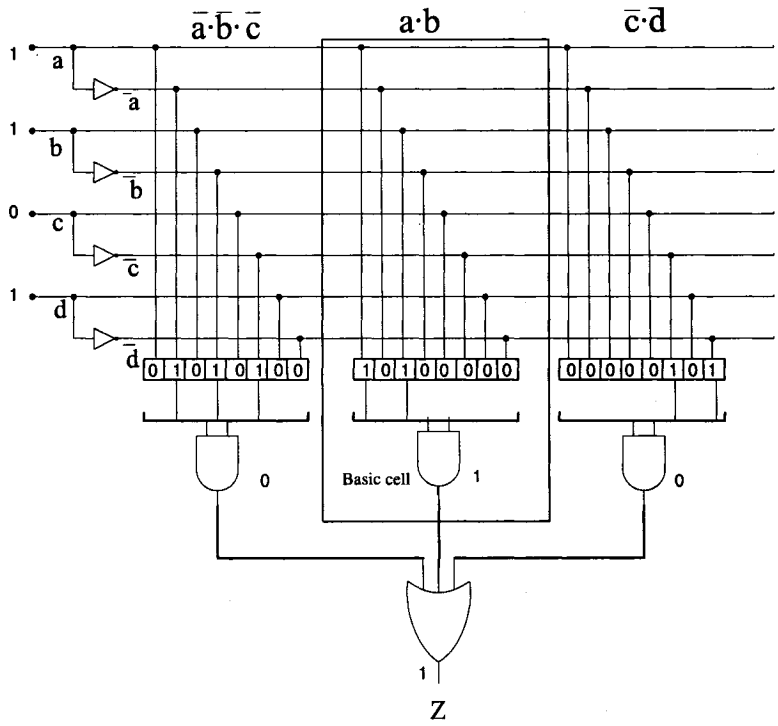

Figure 2.4: Gate level of a small PAL device for a single logic equation.

As was mentioned, in a PAL device only the AND plane is programmable. The programming task is done exactly as was described for the AND plane of a PLA device. Figure 2.4 shows also the values of unique output function Z for the following values of inputs: $a = 1$; $b = 1$; $c = 0$ and $d = 1$.

Figure 2.5 shows how in a small PAL the hardware structure defined by multi-output logic equation 2.1 are implemented. In this solution each equation is considered as an independent equation. Therefore the product term $a \cdot b$, which is a corposant of all three logic equations, is implemented three times, once for every output function.

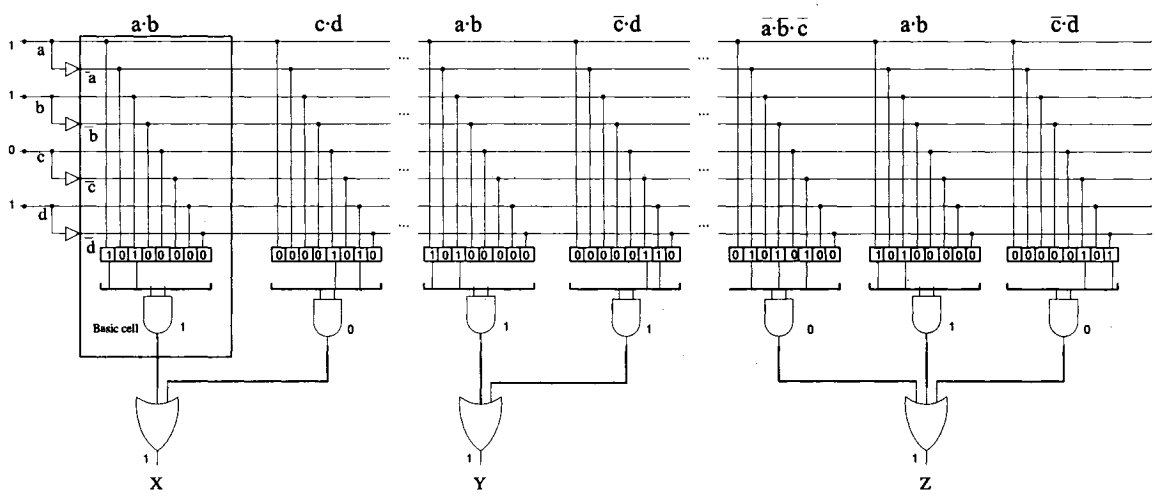

Figure 2.5: Gate level of a small PAL device for multi-output circuit defined by equation 2.1

2.3 Complex Programmable Logic Device

PLAs and PALs were used to implement a wide diversity of simple digital circuits that require a small number of inputs, a small number of outputs and the internal functionality described by few uncomplicated logic functions. Usually, such a digital circuit accomplishes a single well defined function, like decoder, encoder, multiplexer, counter and so on. To implement complex circuits requiring more inputs, more outputs and numerous interrelated logic functions a first solution is to use multiple interconnected PLAs or PALs. As the programmable circuit technology progressed, a more sophisticated solution known as Complex Programmable Logic Device (CPLD) became available [8],[26].

A CPLD device is realized by several hardware blocks implemented on a single chip, with internal interconnection routes. Each hardware block is like a specific PLA device or like a specific PAL device, and the interconnection routes are formed by a group of wire and switches. Figure 2.6 shows the structure of a CPLD formed by

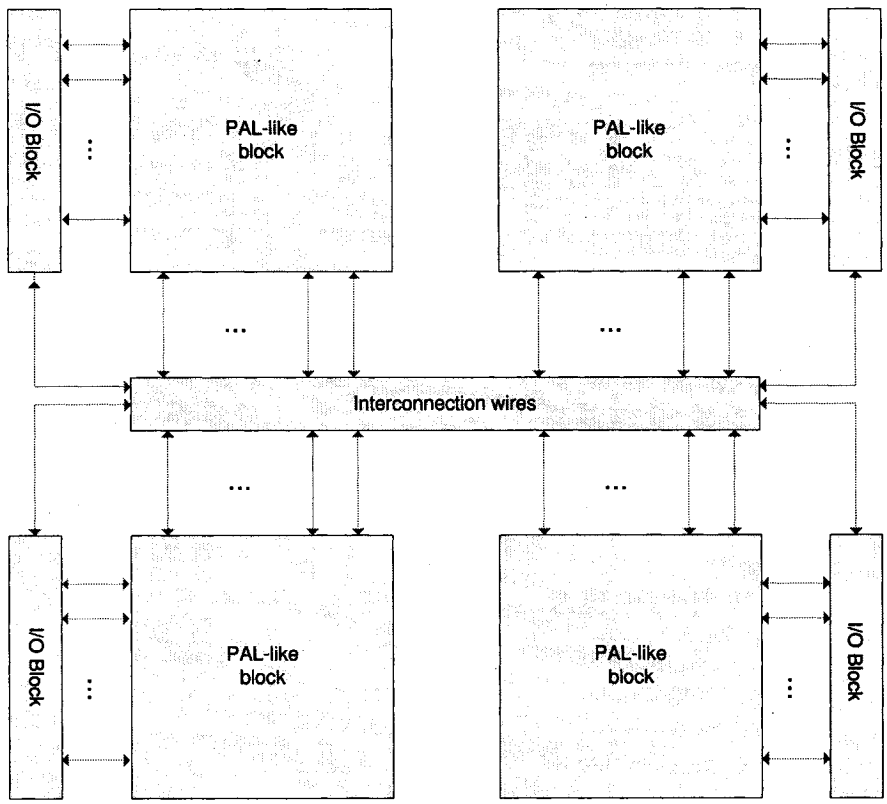

Figure 2.6: General structure of a CPLD

PAL-like blocks and a group of interconnection wires. A PAL-like block can be connected to an I/O block, which is attached to a number of the chips input and output pins. Every PAL-like bloc is composed by macrocells, where the main function of a macrocell is to materialize a logic function with a limited number of product terms. A macrocell usually include other circuitry, like a flop-flop and special gates that allows interconnecting the adjacent macrocells. The interconnection wiring contains programmable switches, usually EEPROM switches, which are used to connect the PAL-like blocks. The number of switches is chosen to provide sufficient flexibility for implementing typical target circuits without wasting many switches in practice.

On the market there are a large variety of commercial available CPLDs. As example, we briefly describe a widely used CPLD family, the ALTERA MAX 7000 [9]. The MAX 7000 family includes chips that range in size from 32 to 512 macrocells. The structure of a MAX 7000 device is illustrated in Figure 2.7. The macrocells are combined into groups of 16. This group of 16 macrocells is named logic array blocks (LABs). Each LAB is connected to an I/O control block. The I/O control block contains tri-state buffers linked to I/O pins. These pins can be programmed as input, output or tri-state bi-directional. Each LAB is also connected to the programmable interconnect array (PIA). The PIA consists of a set of wires and switches that cover the entire device. All connections between macrocells are completed using the PIAs wires and programmable switches. The programmable switches are based on read only memory (EEPROM), and consequently the configuration is saved when power is removed. The MAX 7000 can be reprogrammed in circuit.

Figure 2.8 shows the structure of a macrocell. Five product terms can be imple-

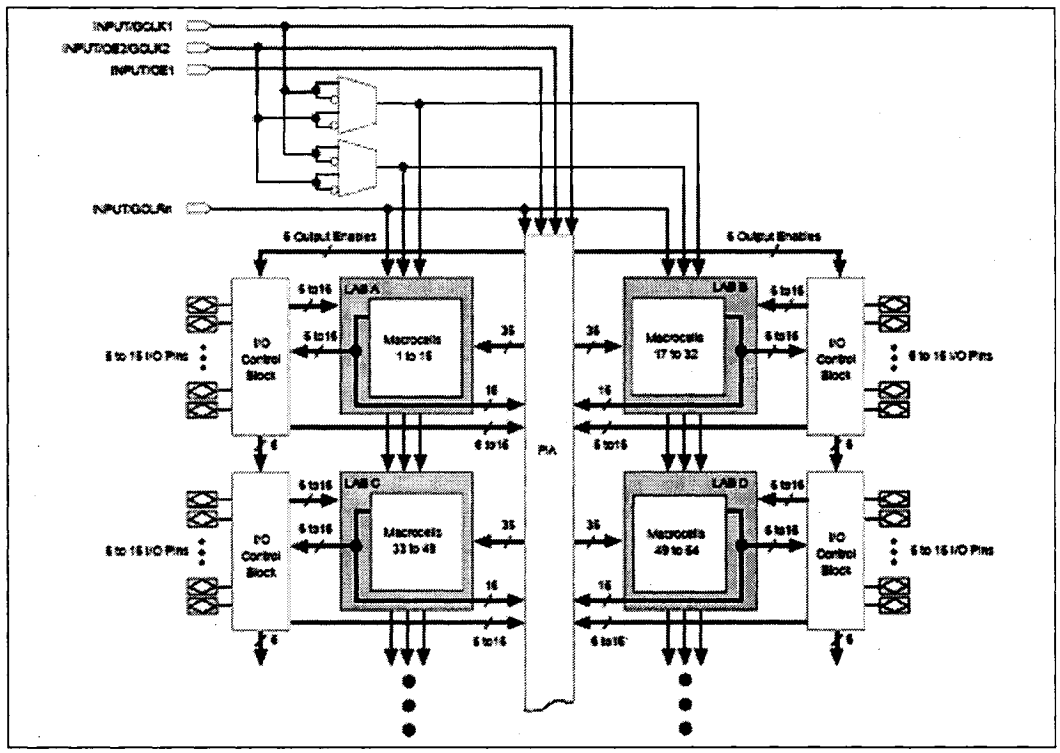

Figure 2.7: MAX 7000 CPLD architecture

mented and connected through the product term select matrix to an OR gate. If more than five product terms are required, additional product terms can be shared from other macrocells. To allow this expansion, the OR gate in a macrocell has an extra input that can be connected to the output of the OR gate of precedent macrocell. In this way functions with maximum 20 product terms can be implemented. The output from the AND/OR network is fed into a programmable flip-flop, which can be bypassed. Each specific MAX 7000 device is available in a large range of speed grade.

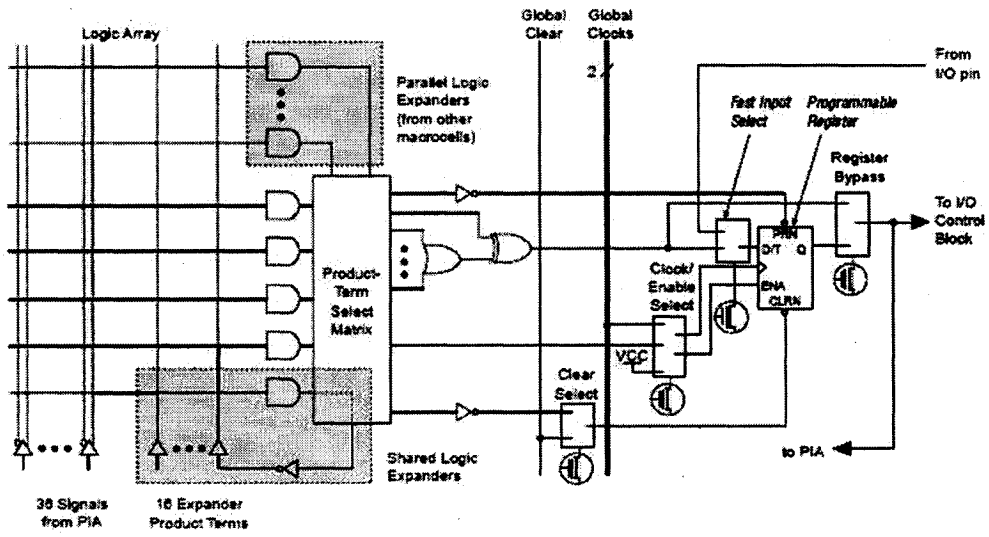

Figure 2.8: Macrocell of MAX 7000

2.4 Memory as Programmable Logic

Programmable Read only Memories (PROMs) were the first devices to implement programmable logic at beginning of 60. The architecture of a PROM consists of an array of storage cells with address lines for inputs and data lines for outputs. In terms of logic implementation, the PROM solution materializes the truth table of a combinational structure and it is equivalent to a fixed AND gates array followed by a programmable OR gates array. In the fixed AND gates array, each AND gate materialize a fundamental product term (minterm). Figure 2.9 shows how the multi-output combinational logic equation 2.1 are implemented in a small PROM. Only active memory cells required for logic equation 2.1 are presented.

The basic principle that leads PROMs to implement logic was extending to RAM memories, and particularly to SRAM memories. Gate logic implemented by SRAMs refers also to truth tables, usually known in this case as look up tables

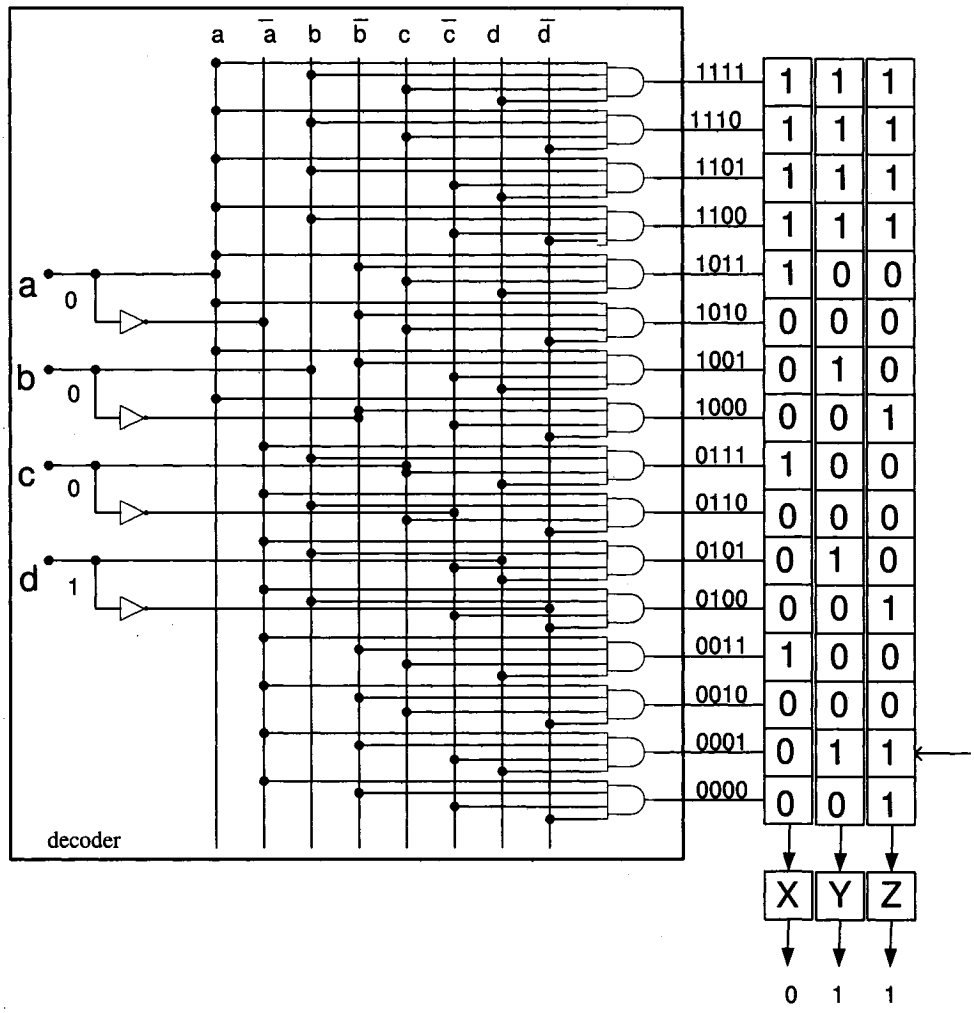

Figure 2.9: Small PROM device for the circuit defined by equation 2.1

(LUT). Normally a LUT define a logic circuit with few inputs (4 to 6) and a single output. It is used to build up a part of the logic blocks for a more sophisticated device named field programmable gate array (FPGA).

2.5 Field Programmable Gate Array

The field programmable gate array (FPGA) device is a programmable logic structure used to implement large logic circuits. A FPGA contains four types of resources:

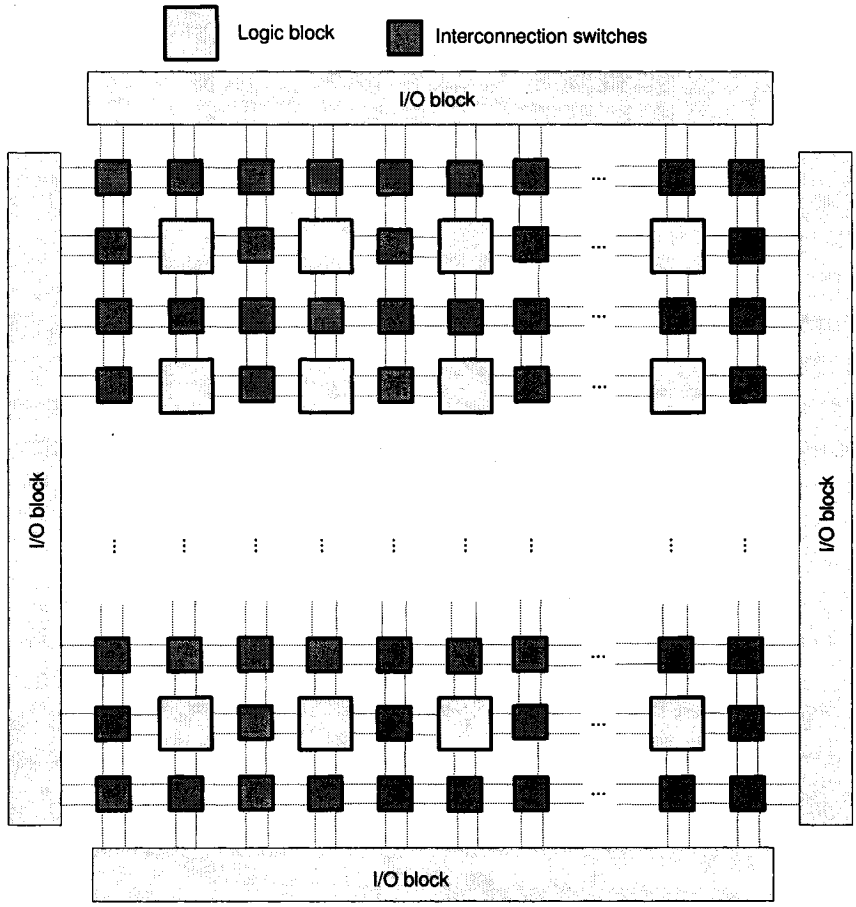

Figure 2.10: General structure of an FPGA

logic blocks to implement universal logic functions, inner general memory blocks to implement specific logic functions (arithmetic functions) or to store specific information, I/O blocks for connecting to outside words through the pins of package and interconnection wires and switches (Figure 2.10).

These interconnection wires and switches are ordered as horizontal and vertical routing channels between rows and columns of logic blocks.

Each logic block in an FPGA contains as principal constituent a small memory (different from the inner general memory blocks), to implement the truth table of a small logic functions, known as was mentioned, lookup table (LUT). The LUT

replace the AND and the OR planes found in CPLD. When a logic structure is implemented in an FPGA the logic blocks are programmed to realize the required small functions and the routing channels are programmed to make the necessary interconnections between logic blocks (for large functions or for interrelated functions). FPGAs do not use EEPROM technology for programmable switches. SRAM memories are used for stoking the truth tables in LUTs and, also, SRAMs switches are used for interconnection wires between LUTs. A switch is materialized by an NMOS transistor, with its gate terminal controlled by an SRAM memory cell. Such a switch is known as a pass-transistor switch. If a 0 is stored in an SRAM memory cell, then the associated NMOS transistor is turned off. But if a 1 is stored in the SRAM memory cell, then the NMOS transistor is turned on. The SRAM-like switches lost their information when the power is removed.

On the market it can be found a large variety of commercial FPGAs. As example, the ALTERA FLEX 10K is described in [3]. The architecture of a FLEX 10K device is illustrated in Figure 2.11. This architecture contains as a principal component a collection of logic array blocks (LABs). Devices in the FLEX 10K family contain from 72 to 1520 LABs. Figure 2.12 shows a logic array block (LAB) of the FLEX 10K family devices. A logic array block is composed of eight logic elements (LEs). Programmable local LAB wires and chip-wide row and column interconnecting wires are available. The architecture of the LE in a LAB of FLEX 10K chips that range in size from 10,000 to 250,000 gates. The FLEX 10K devices is configured by loading internal static random access memory (SRAM). Since SRAM is used, the configuration will be lost at any time when power is removed. Figure

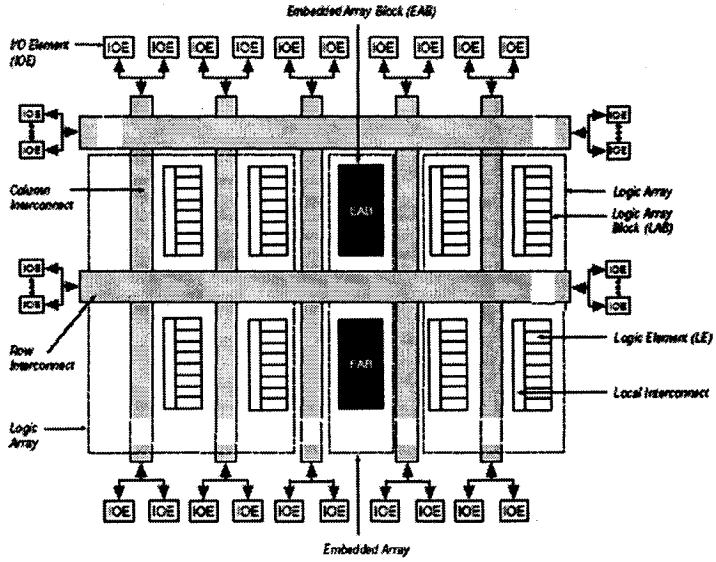

Figure 2.11: Architecture of FLEX 10K FPGA.

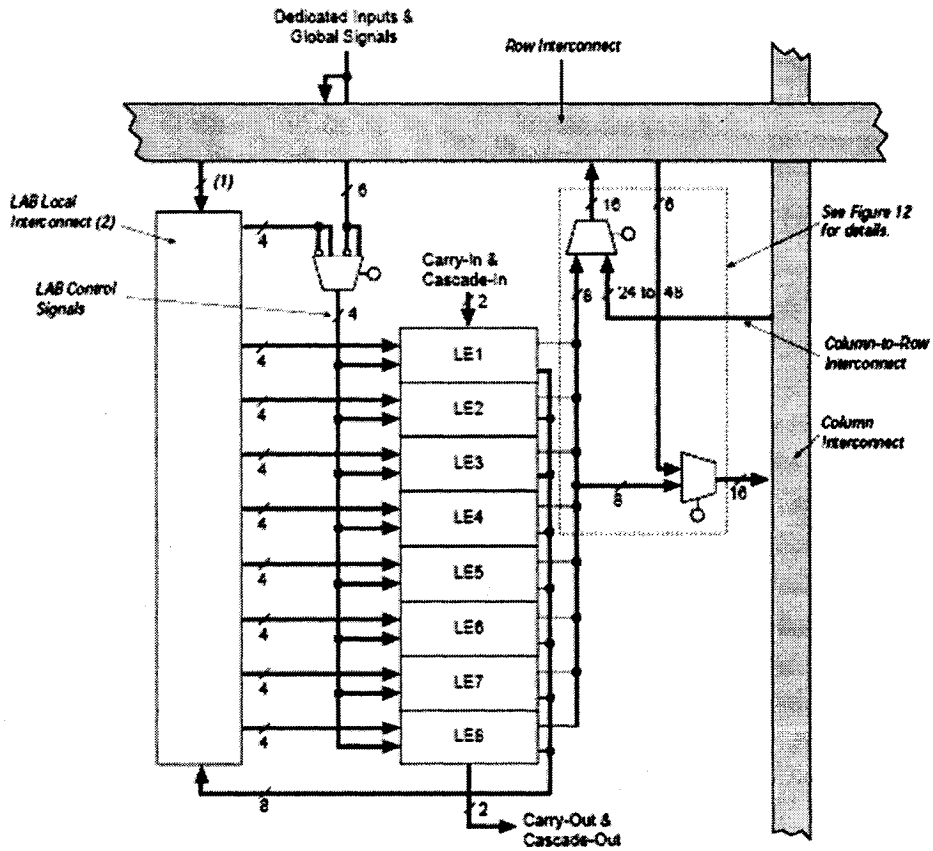

Figure 2.12: Architecture of the LAB in a FLEX 10K FPGA

2.13 shows a logic element (LE) of FLEX 10K devices. The principal part is a high speed LUT having the classical architecture of a SRAM memory, with 16 addresses (every address formed by 4 bits) and 1 bit output. Consequently, a single LUT can implement any single logic function having a maximum of four inputs. Other

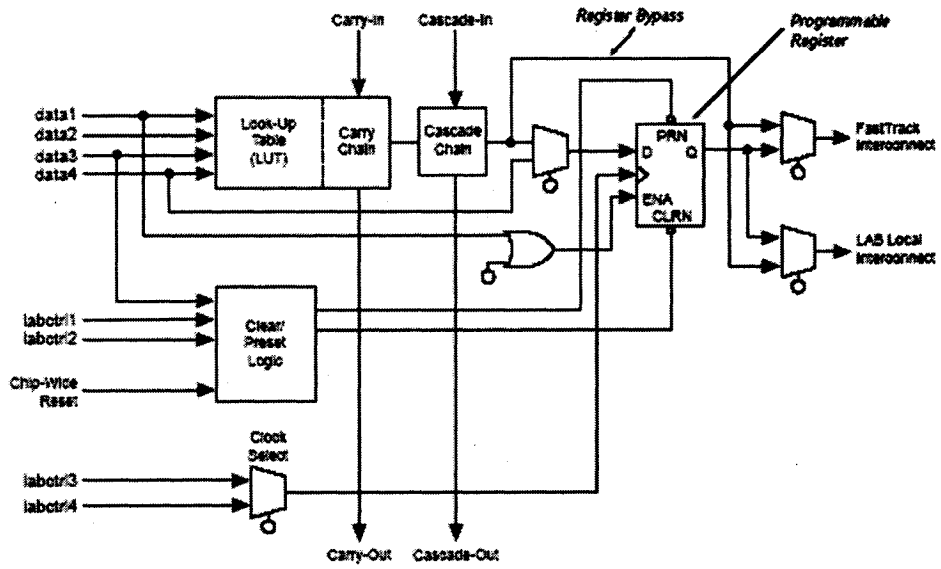

Figure 2.13: Architecture of the LE in a LAB of FLEX 10K

important blocks found in the architecture of this FPGA are the internal general memories blocks (Figure 2.14), known as embedded array blocks (EAB). An EAB contains 2048 bits of memory. A FLEX 10K devices contain from three to twenty EABs. Each EAB can be configured as 256 by 8, 512 by 4, 1024 by 2 or 2048 by 1 SRAM memory or ROM memory.

The matrix of LABs and EABs are connected via programmable row and column interconnects. The input-output elements (IOE) serve the I/O pins. IOEs contain programmable tri-state driver and each pin can be programmed as input, as output or as bi-directional. The FLEX 10K can be in circuit reprogrammed.

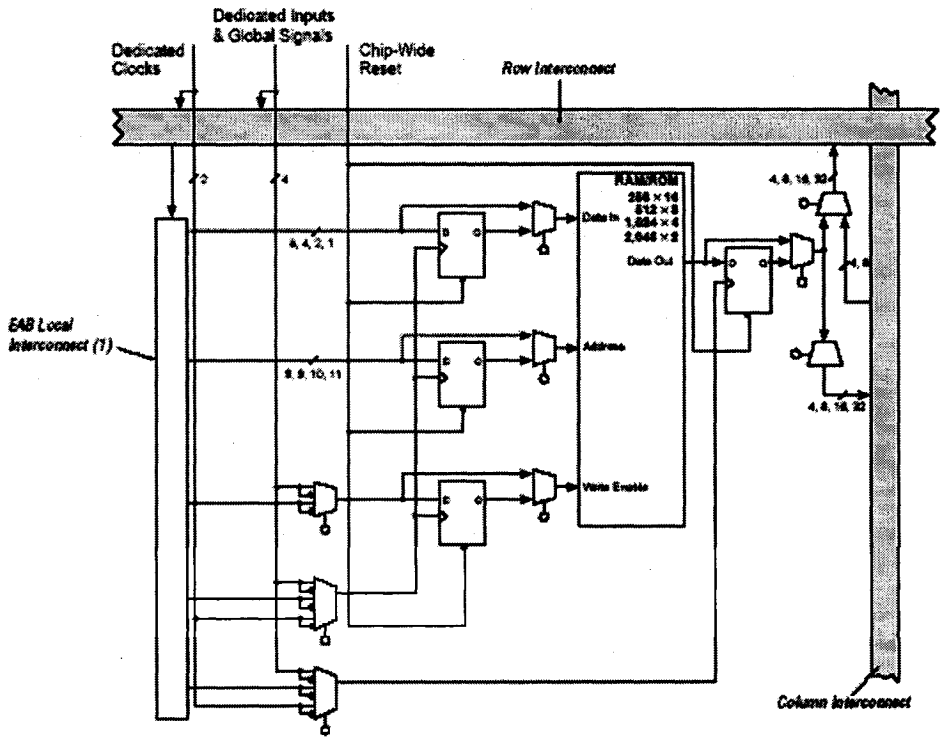

Figure 2.14: Architecture of the EAB in a FLEX 10K FPGA

Given the extensive length of this chapter, as well as the wide variety of topics that were introduced, this chapter serves as a summary to introduce the reader with numerous reprogrammable technologies.

Chapter 3

CONCEPT OF THE NEW ARCHITECTURE

In currently produced PALs and PLAs, special type of programmable gates directly implement product terms. This chapter introduces two solutions of a new type of reconfigurable logic devices based on an indirect implementation of the sum-of-product logic equations.

3.1 Review of Literature

Runtime Reconfiguration (RTR) relies on the ability of the underlying hardware architectures or devices to rapidly alter the functionalities of its components to mirror the target circuit. Key to this ability is reconfiguration handling and speed. Theoretical models and algorithms have established reconfiguration as a computing paradigm. Practical considerations make these models difficult to realize.

One can in the literature an indirect implementation of the sum-of-product logic

equations known as the product terms method [11],[12],[14].

3.2 Implementation of Combinational Circuits onto New Architecture: First Solution

A correlative relationship can be done between the first solution of the new architecture and PAL circuits. Recall that in a PAL, any product term is implemented directly by an AND gate having the inputs of the product term chosen by programmable switches. For any given values of input variables, the logic value of the product term results at the output of an AND gate. For CPLDs which are PAL-like blocks, the same product term found in multiple equations must be implemented separately since product terms cannot be shared. Therefore, if the same product term appears in several logical equations, it must be materialized in all equations where it appears, by a separate AND gate. Let us consider the SoP logical equation that describes the behavior of a combinational circuit with a single output, which will be referred to as the *implemented combinational circuit* on the new architecture. For each product term of the SoP equation, the new architecture attaches two data words stored in specific registers as shown in Figure 3.1.

The first data word is entitled the **mask word** and will be stored in the **mask register** (MR). Its role is to extract all active inputs that form the product term. The second data word is labeled the **control word** and will be stored in the **control register** (CR). Its role is to indicate the values of active input variables (true or complemented) that give the logic 1 to the considered product term. These words

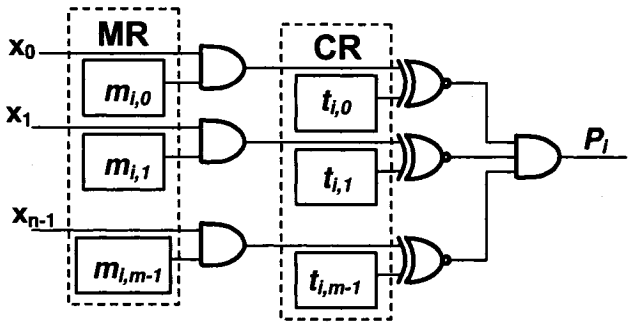

Figure 3.1: New architecture for implementing combinational circuit (first solution)

perform an indirect logic manipulation for establishing the value of the product terms for given values of input variables.

For any given inputs, a product term may either have a logic value of 1 or 0. In order to find this value and therefore its contribution to the output, the first solution of the product terms method is described in three steps. These steps are done in parallel for all the product terms of the same logic equation. First, a masking operation between the inputs and the **mask word** (bit by bit AND operation) is done to extract the values of active variables that make up the product term. Second, a bit by bit comparison of the previous result with the **control word** (bit by bit XNOR operation) is performed to determine, if for the given inputs, each active variable has the same logic value as it does in the control word. Finally, an AND operation is done with all resulting bits of the second step operation. If the output of this final AND gate results in the logic value of 1, then all active variables have the same logic value as they have in the control word and the product term has a logic value of 1 respectively. Otherwise, the logic value of the product term is 0 for the corresponding input variables. Evidently, to obtain the logic value of the SoP equation for the given inputs, all outputs of the final AND gates, thus the values of

all product terms, are OR-ed together.

The hardware implementation of the first solution is a reconfigurable circuit capable of implementing any single output combinational circuits onto it. The maximum number of inputs q and product terms n must first be established. The logic function is build from a set of product terms $\{P_i | \, i = 0, 1, 2, ..., n-1\}$ that can be programmed to fit the logic definition of the function at hand. For q input variables $\{x_j \, | \, j = 0, 1, 2, ..., q-1\}$, the reconfigurable logic block has provisions to implement the SoP form of output Z. The input variables to be taken into the product term P_i are selected with a set of corresponding mask bits $\{m_{i,j} | \, j = 0, 1, 2, ..., q-1\}$. Each input variable x_j of every product term P_i has attached a control product bit $t_{i,j}$ which decides if the corresponding variable is employed in the product term at hand in its true $(t_{i,j} = 1)$ or complemented form $(t_{i,j} = 0)$.

The generalized equation of a product term P_i is given by the following equation:

$$P_i = \left\{ \prod_{j=0}^{q-1} [\overline{(x_j \cdot m_{i,j}) \oplus t_{i,j}}] \right\} \; for \, i = 0, 1, 2, ..., n-1 \tag{3.1}$$

and the SoP output function Z by:

$$Z = \sum_{i=0}^{n-1} P_i \tag{3.2}$$

The subsequent observations must be pointed out:

- Equations 3.1 and 3.2 characterize the combinational part of the first solution in a reconfigurable structure. To complete the hardware device, additional registers must be added to every product term to store the mask and control

words.

- The reconfigurable structure is used to materialize a single output combinational circuit.
- The data words (inputs, mask, product) found in equation 3.1 can implement any combinational circuit limited by the constraints. When new datum is stored in registers the behavior and consequently the output of implemented combinational circuit change automatically. Furthermore, the internal hardware structure remains unchanged.
- These structures can be used to implement multiple outputs combinational circuits. In this case, a separate structure is used to implement each output independently. If the same product term occurs in several logic equations, it is independently implemented in hardware for each equation.

3.3 Example of Implementing Combinational Circuits: First Solution

Let us consider a reconfigurable combinational structure. Assume that the reconfigurable structure is capable of implementing any combinational circuit defined by a logic equation having a maximum of 8 inputs and a maximum of 8 product terms. The example given below has fewer inputs and less product terms than the maximum number allowed. The combinational circuit inputs are considered to be stored in a virtual register with the less significant bit first. Let us assume that the combinational circuit implemented on the new architecture is given by equation 3.3, which

gives the output Z as logical function of inputs x_3, x_2, x_1 and x_0:

$$Z = x_3 \cdot x_2 + \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1} + \overline{x_1} \cdot \overline{x_0} \tag{3.3}$$

The inputs are packaged to form a 1 byte input word I as shown in Table 3.1. Unused bits in the input word are filled with bit 0 (not explicitly shown) and their contribution is always 0 and will be ignored.

Table 3.1: Input package in first example

Inputs					x_3	x_2	x_1	x_0
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Equation 3.3 contains three product terms, namely $P_0 = x_3 \cdot x_2$; $P_1 = \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1}$ and $P_2 = \overline{x_1} \cdot \overline{x_0}$. The values of mask and control word are established by adding the weights (power of 2) of the input variables materializing the product term at hand. These words are known as context words and exist for each product term in equation 3.3. Input variables in the I word are right aligned. In this way, the context words can be expressed by hexadecimal integers (or for easier manipulation by decimal numbers), which once stored in memory always locates the corresponding logic variables in a correct position in the considered word.

Each **mask word** is determined by the sum of weights of active inputs that makes up the product term. The mask word M_0 for product term $P_0(x_3 \cdot x_2)$ is obtained by the sum of weights of active inputs x_3 ($2^3 \times 1$) and x_2 ($2^2 \times 1$) resulting in $M_0 = 8 + 4 = 12 = (0C)_H$. The mask word for the product term $P_1(\overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1})$ has the value of $M_1 = 14 = (0E)_H$, because it is composed by the active inputs x_3, x_2 and x_1. Table 3.2 displays the mask words weights for products terms in equation 3.3.

Table 3.2: Weights of mask words for equation 3.3

Inputs					x_3	x_2	x_1	x_0
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Mask Word	m_7	m_6	m_5	m_4	m_3	m_2	m_1	m_0
M_0	0	0	0	0	1	1	0	0
M_1	0	0	0	0	1	1	1	0
M_2	0	0	0	0	0	0	1	1

Each **control word** represents the sum of the weights of the active input variables that give a logic value 1 for the product term at hand. The control word T_0 for product term $P_0(x_3 \cdot x_2)$ is obtained by the sum of weights, which gives a value of logic 1 to the considered product term, precisely $x_3 = 1$ (logic) and $x_2 = 1$ (logic). Adding weight of x_3 $(2^3 \times 1)$ with weight of x_2 $(2^2 \times 1)$ results in $T_0 = 8 + 4 = 12 = (0C)_H$. Similarly, the control word for the product term $P_1(\overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1})$ has the value of $T_1 = 0$, because it has the value of logic 1 when x_3 =0 (logic) and $x_2 = 0$ (logic) and $x_1 = 0$ (logic). Adding these weights results in the value of the product term, respectively $T_1 = (2^3 \times 0) + (2^2 \times 0) + (2^1 \times 0) = 0 = (00)_H$. Table 3.3 summarizes the control words weights for the product terms in equation 3.3.

Table 3.3: Weights of control words for equation 3.3

Inputs					x_3	x_2	x_1	x_0
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Control Word	t_7	t_6	t_5	t_4	t_3	t_2	t_1	t_0
T_0	0	0	0	0	1	1	0	0
T_1	0	0	0	0	0	0	0	0
T_2	0	0	0	0	0	0	0	0

Table 3.4: Context words for the product terms in equation 3.3

Product Term	Mask Word	Control Word
$P_0 = x_3 \cdot x_2$	$M_0 = 12$	$T_0 = 12$
$P_1 = \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1}$	$M_1 = 14$	$T_1 = 0$
$P_2 = \overline{x_1} \cdot \overline{x_0}$	$M_2 = 3$	$T_2 = 0$

3.3. Example of Implementing Combinational Circuits: First Solution

Table 3.4 portrays the decimal values of mask and control words for the product terms in the same equation. Furthermore, the context words are regrouped to form the context words list similar to Table 3.6. The context words list characterizes the implemented circuit behavior.

The hardware implementation will be described in Figures 3.2 and Figure 3.3.

Table 3.6: Context words list for product terms in equation 3.3

Product Term	$P_0 = x_3 \cdot x_2$	$P_1 = \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1}$	$P_2 = \overline{x_1} \cdot \overline{x_0}$
Context Word	$(0C0C)_H$	$(0E00)_H$	$(0300)_H$

Figure 3.2 shows the hardware structure of the device, based on the first proposed

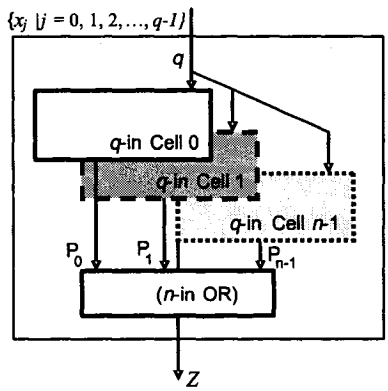

Figure 3.2: Block diagram of reconfigurable circuit (first solution)

solution of the product terms method implementation, where single output combinational circuit is materialized. This reconfigurable structure can be used to materialize any combinational circuits with single output, having maximum 8 inputs, and maximum 8 product terms. Usually this structure represents a macrocell, like those found in current CPLDs devices.

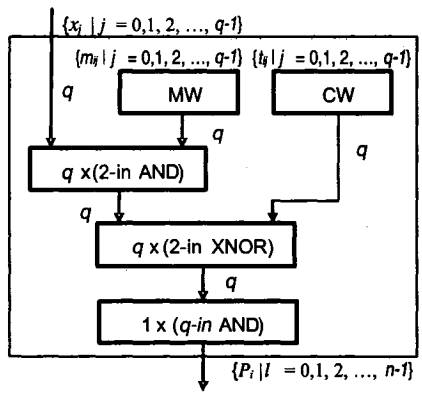

Figure 3.3: Block diagram of simplified reconfigurable cell (first solution)

Figure 3.3 shows the structure of an eight bit basic cell, found in Figure 3.2. A basic cell is needed for each product term of the logic equation. The input variables are AND-ed bit by bit with the mask word using two inputs AND gates. All registers in Figure 3.3 have $q=8$ bits. Eight 2 inputs AND gates, each one implementing the AND operation between an input bit with the corresponding bit (same weight) of the mask word are needed for the first logic level. Next, the control word and the intermediate result obtained above are compared (second logic level). Precisely, each bit of the intermediate result is XNOR-ed with the bit found in the same position in the control word. The logic value of the product term is then determined by AND-ing the 8 bits at the output of XNOR gates using an 8 inputs AND gate (third logic level). If the output of this AND gate is at logic 1 for the considered inputs, the corresponding product term has the logic value 1. There is only one OR gate for the circuit of Figure 3.2 which determines the logic values of the implemented equation. The cells act as a filter between the information that characterizes the product terms and the output and enables implementation in parallel of product terms. It must be noted that the input register and the OR gate are not part of the

basic cell.

Assume that k active product terms must be implemented on the reconfigurable circuit similar to Figure 3.2 where $k \leq n$ (n is the number of real basic hardware cells that correspond to the maximum number of product terms that can be implemented; in this example, $n = 8$). During the initial stage the $2 \times k$ registers in the k active basic cells are loaded with the information that corresponds to existing k product terms in the considered equation similarly to Table 3.4. The context words list is generated by a software system, according to logic equations of the implemented circuit. The unused $k - n$ basic cells on the reconfigurable circuit must be initialized in such a way that the logic value of the unused product terms is always 0 for all inputs. During the same initial stage of combinational circuits onto new architecture, the unused $n-k$ mask registers are loaded with 0 $(00)_H$ and the unused $n - k$ product registers are loaded with 256 $(FF)_H$. For different logical equations, the number of active k product terms can be different and the logic behavior of the implemented circuit changes accordingly. It is important to underline that the physical structure of the reconfigurable circuit remains unchanged. To implement a plurality of combinational circuits on the new hardware structure and allow flexibility in changing these circuits on ad-hoc basis, the number n of real basic cells must be as large as technologically possible.

Figure 3.4 shows how the context words are applied to the associated inputs to provide the outputs for the implemented circuit. Only the first three basic cells are presented, knowing that equation 3.3 comprises three product terms ($k = 3$). The bi-directional arrows indicate logic operations between bits, while a unidirectional

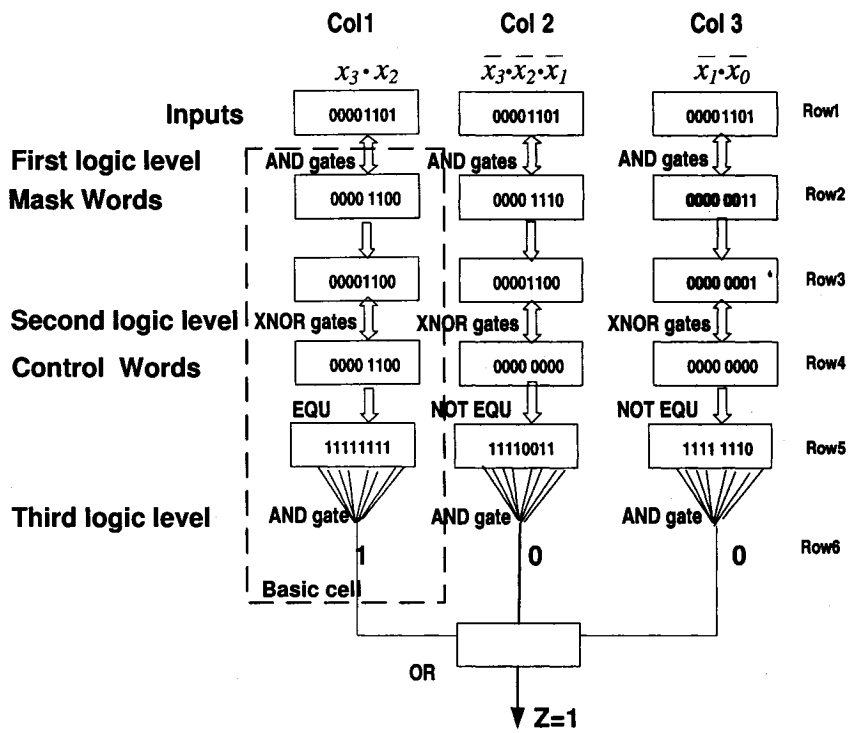

Figure 3.4: Data flow in the example for equation 3.3

arrow shows the result of logic operations. In this example, equation 3.3 and input byte $IN = (01)_H$ are considered (therefore $IN = (0000\ 0001)_B$ in binary, namely $x_3 = 1$; $x_2 = 1$; $x_1 = 0$ and $x_0 = 1$). Thus, the outputs determined from equation 3.3 are:

$$Z = 1 + 0 + 0 = 1 \tag{3.4}$$

Columns *Col1* to *Col 3* specify the operations for each product term $P_0 = x_3 \cdot x_2$; $P_1 = \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1}$ and $P_2 = \overline{x_1} \cdot \overline{x_0}$. *Row1* shows the inputs and *Row2* shows the mask words as given by Table 3.4. The first intermediate results are shown in *Row3*. These intermediate results are obtained by performing 2 bits AND operations between the inputs and corresponding mask words. *Row4* shows the control words associated

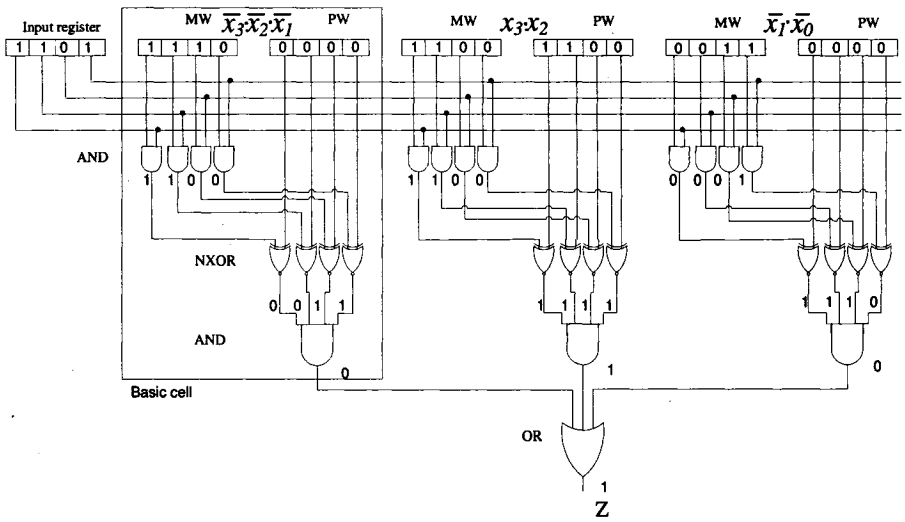

Figure 3.5: Hardware implementation at gate level (first solution)

with each product term, as given in Table 3.4. The second intermediate results are shown in *Row5*. These intermediate results are obtained by 2 bits XNOR operations (Table 3.8). *Row6* shows AND operations between all bits of second intermediate

Table 3.8: XNOR Truth Table

x_1	x_2	$x_1 \odot x_2$
0	0	1
0	1	0
1	0	0
1	1	1

results for *Row5* found in a column, to determine if, for the considered inputs, the respective product term has the logic value of 1 or the logic value of 0. Bits found on *Row6* are OR-ed together, to obtain the output.

Figure 3.5 shows the hardware implementation at gate level, of the first proposed solution for the product terms method, where single output combinational circuit is implemented, considering equation 3.3 and the attached registers. To simplify the design, only three basic cells that materialize the three existing active product

terms in equation are presented above. The flow of binary information takes into consideration the established values for the mask words and the control words and supposing the inputs as $x_3 = 1$; $x_2 = 1$; $x_1 = 0$ and $x_0 = 1$.

As stated previously, the reconfigurable structure in the proposed first solution can be used to implement multiple outputs combinational circuits. In this case, a separate reconfigurable structure is used to implement each output of the considered combinational circuit as an independent output. If the same product term is encountered in numerous logical equations of the implemented circuit, this product term must be materialized in hardware for each independent equation. Let us assume that multi-output combinational circuits are defined by equation 3.5 (same as equation 1.1). Outputs X, Y, and Z are expressed as logical functions of inputs x_3, x_2, x_1 and x_0.

$$\begin{aligned} Z_2 &= x_3 \cdot x_2 + x_1 \cdot x_0 \\ Z_1 &= x_3 \cdot x_2 + \overline{x_1} \cdot x_0 \\ Z_0 &= \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1} + x_3 \cdot x_2 + \overline{x_1} \cdot \overline{x_0} \end{aligned} \tag{3.5}$$

The input variables are placed in a virtual register having the same weights as mentioned in Table 3.1. Table 3.9 lists the decimal weights of the mask and control words for each of the five product terms defined by equation 3.5. The context words that characterizes the circuit behavior can be regrouped to form a single context word hexadecimal list, as shown in Table 3.11. Figure 3.6 shows the hardware reconfigurable structure, based on the first proposed solution of the product terms method, where multiple outputs combinational circuit is materialized. Assume the inputs are as follows: $x_3 = 1$; $x_2 = 1$; $x_1 = 0$ and $x_0 = 1$. To simplify

Table 3.9: Context words of the product terms of equation 3.5

Product Term	Mask Word	Control Word
$P_0 = x_3 \cdot x_2$	$M_0 = 12$	$T_0 = 12$
$P_1 = x_1 \cdot x_0$	$M_1 = 3$	$T_1 = 3$
$P_2 = \overline{x_1} \cdot x_0$	$M_2 = 3$	$T_2 = 1$
$P_3 = \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1}$	$M_3 = 14$	$T_3 = 0$
$P_4 = \overline{x_1} \cdot \overline{x_0}$	$M_4 = 3$	$T_4 = 0$

Table 3.11: List of context words for product terms of equation 3.5

Product Term	$P_0 = x_3 \cdot x_2$	$P_1 = x_1 \cdot x_0$	$P_2 = \overline{x_1} \cdot x_0$	$P_3 = \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1}$	$P_4 = \overline{x_1} \cdot \overline{x_0}$
List	$(0C0C)_H$	$(0303)_H$	$(0301)_H$	$(0E00)_H$	$(0300)_H$

the design, only basic cells that materialize the existing active product terms in each equation are presented. This reconfigurable block can be used to materialize any combinational circuits with multiple outputs limited by the constraints established previously such as q maximum inputs, r maximum outputs and n maximum product terms for each equation. It must be pointed out that in Figure 3.6 the product terms $P_0 = x_3 \cdot x_2$, which is a common to all three equations, is implemented thrice. The second solution presents the product terms method with the possibility of sharing the same product terms among several equations.

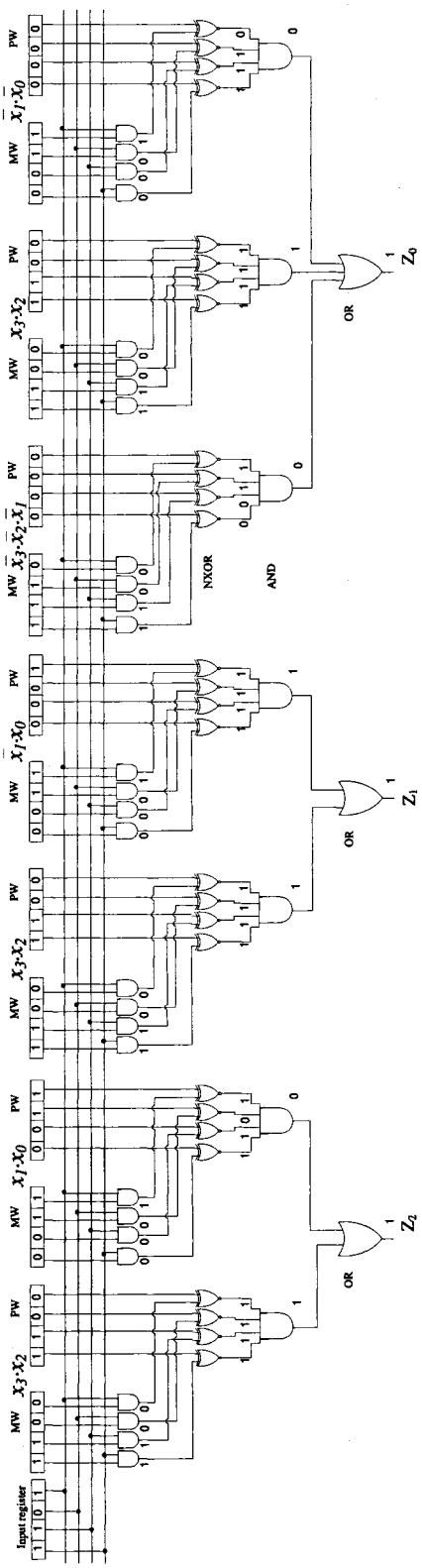

Figure 3.6: Hardware implementation at gate level for equation 3.5 (first solution)

3.4 Implementation of Combinational Circuits onto New Architecture: Second Solution

A correlative relationship can be done between the second solution of the new architecture and PLA circuits. Remember that in a PLA solution, any product term is implemented directly by an AND gate having the inputs chosen by programmable switches. The gates in the OR plane are also chosen by programmable switches. In a CPLD based on PLAs, any product terms found in several logical equations can be shared between outputs. In other words, should the same product term appear in several logical equations, it will be materialized only once by a single AND gate as shown in Figure 3.7.

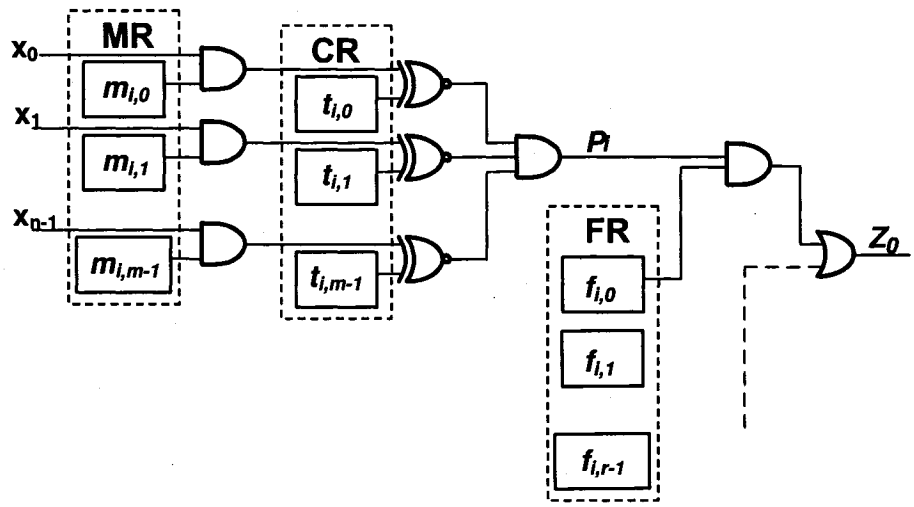

Figure 3.7: New architecture for implementing combinational circuit (second solution)

Conceptually, the second solution of the product terms method uses the same logic as the first solution. This second solution is applicable to SoP equations that describe the behavior of multi-output combinational circuits. Each product term of

the SoP equations is now characterized by three data words. All context words will be independently stored in specific registers. The first data word is the mask word and is stored in the mask register. The second data word is the control word and is stored in the product register. The third data is called the **function word** and is stored in the **function register** (FR). Its role is to indicate which output equations contain the product term. The three words establish the value of the product terms at hand for given values of input variables, without directly implementing product terms in hardware. In order to find this value, and therefore its contribution to the outputs, four steps are involved in the second solution which can be performed in parallel for all product terms in all equations. The first three steps (bit by bit masking with AND gates, bit by bit comparison with XNOR gates and AND operation between all the result bits of the second step) are identical with the first solution. In the fourth step, the bits having the same weight in the function words of active product terms (product terms with logic values 1 for given input variables) are OR-ed together to determine the value of outputs.

The hardware implementation of the second solution is a reconfigurable circuit capable of implementing any multi-output circuits $\{Z_l \mid l = 0,\ 1,\ 2, ...,\ r-1\}$ onto it. The generalized equation of a product term P_i is given by equation 3.6 and is similar to the first solution. The maximum number of inputs q, the maximum number of accepted outputs r, and the maximum number of accepted product terms n for all equations must first be established. Note that the signification of n is different from that found in the first solution. For q input variables $\{x_j \mid j = 0,\ 1,\ 2, ..., q-1\}$, the reconfigurable logic block has provisions to implement the SoP form of outputs

Z_l. The input variables to be taken into the product term P_i are selected with a set of corresponding mask bits $\{m_{i,j} | \, j = 0, 1, 2, ..., q-1\}$. Each input variable x_j of every product term P_i has attached a control product bit $t_{i,j}$ which decides if the corresponding variable is employed in the product term at hand, in its true ($t_{i,j} = 1$) or complemented form ($t_{i,j} = 0$).

$$P_i = \left\{ \prod_{j=0}^{q-1} [\overline{(x_j \cdot m_{i,j}) \oplus t_{i,j}}] \right\} \; for \; i = 0, 1, 2, ..., n-1 \tag{3.6}$$

The same product term can be employed to implement different logic output functions, if their definitions allow to conducting the minimization process towards such a goal. Consequently, since different subsets of P_i might be used in various equations, a third set of logic variables $\{f_{i,l} \, | \, l = 0, 1, 2, ..., r-1\}$ specify the set of functions $\{Z_l \, | \, l = 0, 1, 2, ..., r-1\}$ which use the product term at hand in their expression:

$$Z_l = \sum_{i=0}^{n-1} P_i \cdot f_{i,l} \; for \; l = 0, 1, 2, ..., r-1 \tag{3.7}$$

The following observations must be pointed out:

- Equations 3.6 and 3.7 characterize the combinational part of the second solution of a reconfigurable structure. Specific registers must be added to every basic product term for storing the mask, product and function words to finalize the hardware device.
- The reconfigurable structure is used to materialize multi-output combinational circuits.

- The data (inputs, mask words, control words, function words) found in equation 3.6 can describe any combinational circuit limited by the constraints; when new datum is stored in registers, the behavior and consequently the output of implemented circuit change automatically.
- When the implemented circuit is changed, the internal structure of the reconfigurable circuit remains unchanged.

3.5 Example for Implementing Combinational Circuits: Second Solution

The multi-output combinational circuit defined given by equation 3.8 are (same as equation 1.1) :

$$\begin{aligned} Z_2 &= x_3 \cdot x_2 + x_1 \cdot x_0 \\ Z_1 &= x_3 \cdot x_2 + \overline{x_1} \cdot x_0 \\ Z_0 &= \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1} + x_3 \cdot x_2 + \overline{x_1} \cdot \overline{x_0} \end{aligned} \tag{3.8}$$

Inputs x_3, x_2, x_1 and x_0 are right aligned and packaged into a 1 byte input word (I word). The same procedure is done with the O word for outputs Z_2, Z_1, Z_0 as shown in Table 3.13. Unused bits for both input and output words are set with bit

Table 3.13: The package of inputs and outputs in the second example

Inputs					x_3	x_2	x_1	x_0
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Outputs						Z_2	Z_1	Z_0

0 (not explicitly shown). Their contribution to the sum of weights is always 0 and

will be ignored. Equation 3.8 comprises five distinct product terms: $P_0 = x_3 \cdot x_2$, $P_1 = x_1 \cdot x_0$, $P_2 = \overline{x_1} \cdot x_0$, $P_3 = \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1}$ and $P_4 = \overline{x_1} \cdot \overline{x_0}$.

Each **mask word** is determined by the sum of each input weight product term as shown in Table 3.14. The mask word for product term $P_1(x_1 \cdot x_0)$ is obtained by the sum of weights of inputs x_1 (2^1) and x_0 (2^0), which gives $M_1 = 3 = (03)_H$. The mask word for product term $P_2(\overline{x_1} \cdot x_0)$ has the same value since the same inputs make up the product term.

Table 3.14: Weights of mask words for equation 3.8

Inputs					x_3	x_2	x_1	x_0
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Mask Word	m_7	m_6	m_5	m_4	m_3	m_2	m_1	m_0
M_0	0	0	0	0	1	1	0	0
M_1	0	0	0	0	0	0	1	1
M_2	0	0	0	0	0	0	1	1
M_3	0	0	0	0	1	1	1	0
M_4	0	0	0	0	0	0	1	1

Each **control word** represents the sum of weights for inputs that give a logic value 1 for product term as shown in Table 3.15. The control word for product term $P_1(x_1 \cdot x_0)$ is obtained by the sum of weights, which gives a value of logic 1 to the analyzed product term, precisely $x_1 = 1$ (logic) and $x_0 = 1$ (logic). Weights x_1 ($2^1 \times 1$) and x_0 ($2^0 \times 1$) are added to obtain $T_1 = 3 = (03)_H$. The control word for product term $P_2(\overline{x_1} \cdot x_0)$ has a different weight since it has logic value 1 when $x_1 =$ 0 (logic) and $x_0 = 1$ (logic). Consequently, x_1 ($2^1 \times 0$) can be added to x_0 ($2^0 \times 1$) to obtain $T_2 = 1 = (01)_H$.

Table 3.15: Weights of control words for equation 3.8

Inputs					x_3	x_2	x_1	x_0
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Control Word	t_7	t_6	t_5	t_4	t_3	t_2	t_1	t_0
T_0	0	0	0	0	1	1	0	0
T_1	0	0	0	0	0	0	1	1
T_2	0	0	0	0	0	0	0	1
T_3	0	0	0	0	0	0	0	0
T_4	0	0	0	0	0	0	0	0

The **function word** indicates the contribution of the product term to the outputs as portrayed in Table 3.16. For example, product term $P_0(x_3 \cdot x_2)$ is found in output equation X that has weight 2^2 in the O word. The same product term $P_0(x_3 \cdot x_2)$ is found in output equation Y and has weight 2^1 in the O word. Additionally, it can also be found in output equation Z, which has weight 2^0 in the O word. Therefore, the function word for product term $P_0(x_3 \cdot x_2)$ has the value: $F_0 = 2^2 + 2^1 + 2^0 = 7 = (07)_H$.

Table 3.16: Weights of function words for equation 3.8

Outputs						Z_2	Z_1	Z_0
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Function Word	f_7	f_6	f_5	f_4	f_3	f_2	f_1	f_0
F_0	0	0	0	0	0	1	1	1
F_1	0	0	0	0	0	1	0	0
F_2	0	0	0	0	0	0	1	0
F_3	0	0	0	0	0	0	0	1
F_4	0	0	0	0	0	0	0	1

Table 3.17 lists the decimal values of mask word, control word and function for each product term in equation 3.8. The information describing the circuit behavior can be further grouped to form a single context words hexadecimal list as shown in Table 3.18. The hardware implementation will be described in Figure 3.8 and Figure 3.9.

Table 3.17: Context words of the product terms in equation 3.8

Product Term	Mask Word	Control Word	Function Word
$P_0 = x_3 \cdot x_2$	12	12	7
$P_1 = x_1 \cdot x_0$	3	3	4
$P_2 = \overline{x_1} \cdot x_0$	3	1	2
$P_3 = \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1}$	14	0	1
$P_4 = \overline{x_1} \cdot \overline{x_0}$	3	0	1

Table 3.18: List of context words for product terms in equation 3.8

Product Term	$P_0 = x_3 \cdot x_2$	$P_1 = x_1 \cdot x_0$	$P_2 = \overline{x_1} \cdot x_0$	$P_3 = \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1}$	$P_4 = \overline{x_1} \cdot \overline{x_0}$
List	$(0C0C07)_H$	$(030304)_H$	$(030102)_H$	$(0E0001)_H$	$(030001)_H$

Figure 3.8 shows the reconfigurable circuit based on the second solution of the product terms method. This reconfigurable structure can be used to materialize any combinational circuits having a maximum of $q = 8$ inputs, a maximum of $r = 8$ outputs and a maximum of n product terms for all equations. Usually this reconfigurable structure is a macrocell similar to those found in existing CPLD devices.

Figure 3.9 shows the basic cell for the second solution. A basic cell is needed for each product term of equations. At gate level, the input variables are AND-ed with the mask word. The context registers have 8 bits and require eight 2 inputs AND gates. Each 2 input AND gate implements the operation between an input bit with the corresponding bit (same position) of the mask word. Next, the control word and the intermediate result obtained above are compared. Precisely, each bit of the intermediate result is XNOR-ed with the bit found in the same position in the control word. The logic value of the product term is then determined by AND-ing the 8 bits at the output of XNOR gates using an 8 inputs AND gate. When the output of this AND gate is at logic 1 for the given inputs, the product term at hand

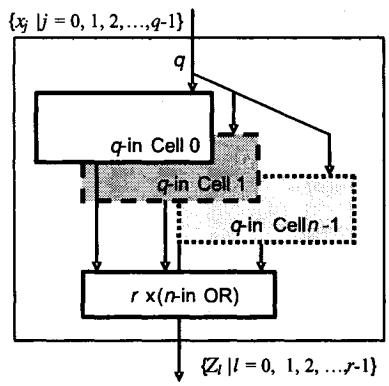

Figure 3.8: Block diagram of product-term reconfigurable circuit (second solution)

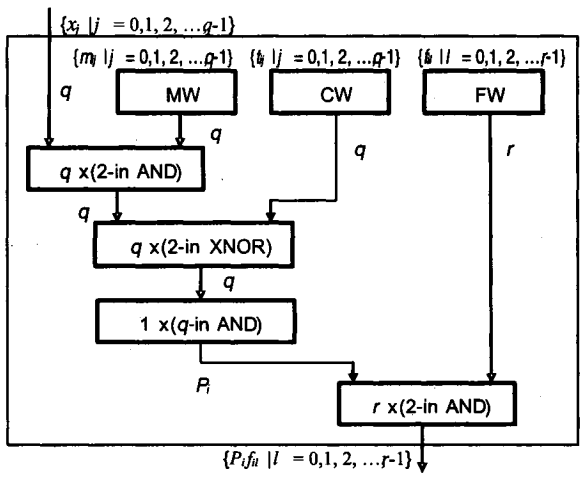

Figure 3.9: Block diagram of reconfigurable basic cell (second solution)

has logic value 1 in all equations where it is present. As a result, the eight 2 inputs AND gates enables the corresponding bits of function words to be OR-ed and to determine the circuit outputs. There is one group of OR gates for the circuit of Figure 3.8, each OR gate having n inputs, where n expresses the total number of product terms. If n is large for a specific circuit, the block of OR gates may be realized in a multi-level hardware implementation. It must be noted that the input register and the OR gates are not part of the basic cell. The detailed logic diagram of the generic reconfigurable circuit is presented in Figure 3.10.

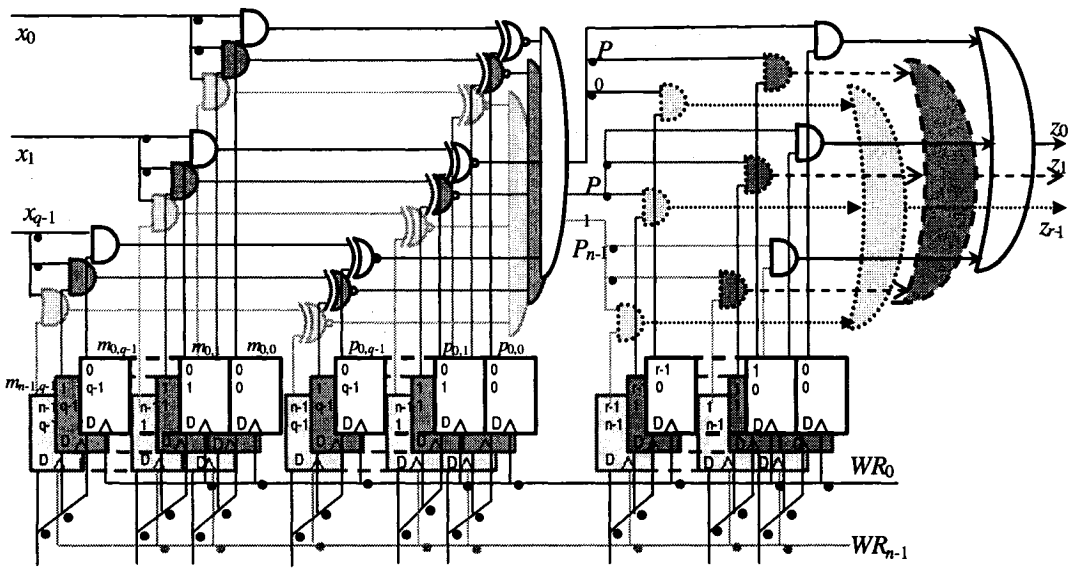

Figure 3.10: Block Diagram of Reconfigurable Circuit (second solution)

During the initial phase, $3k$ registers for k basic cells are stored with information corresponding to k active product terms for all equations. A context word list is produced with the use of a software system to compute weights according to given logic equations ($k \leq n$, where n is the number of reconfigurable cells on the circuit and corresponds to the maximum number of product terms that can be implemented). Context registers (mask, product and function) of unused $n - k$ basic cells are all loaded with 0 $(00)_H$ during the same initial phase. For different logic equations, the number of active k product terms can be different and the logical behavior of the circuit changes accordingly, but the physical structure of the reconfigurable structure remains unchanged.

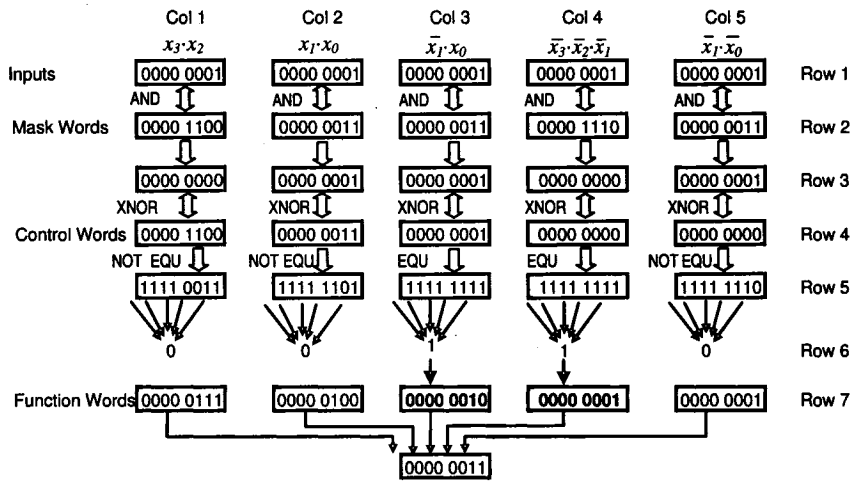

Figure 3.11: Data flow for the circuit expressed by equation 3.8

Figure 3.11 illustrates how mask words, control words and function words are applied to the associated inputs to provide the outputs for the circuit of the second solution. Only five basic cells are portrayed since equation 3.8 contains five distinct product terms ($k = 5$). The bi-directional arrows indicate logic operations between bits, while a unidirectional arrow shows the result of logic operations. In this example, input byte $IN = (01)_H$ is considered (therefore $IN = (0000\ 0001)_B$ in binary, namely $x_3 = 0$, $x_2 = 0$, $x_1 = 0$, and $x_0 = 1$). Outputs determined from equation 3.8 are given by:

$$\begin{aligned} Z_2 &= 0 + 0 = 0; \\ Z_1 &= 0 + 1 = 1; \\ Z_0 &= 1 + 0 + 0 = 1; \end{aligned} \tag{3.9}$$

Columns *Col1* to *Col5* specify all operations for each product term: $P_0 = x_3 \cdot x_2$, $P_1 = x_1 \cdot x_0$, $P_2 = \overline{x_1} \cdot x_0$, $P_3 = \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1}$ and $P_4 = \overline{x_1} \cdot \overline{x_0}$. *Row1* displays the inputs and *Row2* shows the mask words as shown in Table 3.17. The first

intermediate results are shown on *Row3*. These intermediate results are obtained by performing 2 bits AND operations between inputs and corresponding mask words. *Row4* shows the control words associated with each product term. The second intermediate results are shown on *Row5*. These intermediate results are obtained by 2 bits XNOR operations. *Row6* shows AND operations between all bits of second intermediate results (*Row5*) found in a column, to determine if, for the considered inputs, the respective product term has logic value of 1 or logic value of 0. Bits found in the same position as the function words of Row 7, for which the corresponding product term has the logic value of 1, are OR-ed together, to obtain the outputs.

Any new implementation of a different circuit only requires to update the context registers. New tables will provide the logic behavior for the new circuit. Let us assume a new multiple output combinational circuit implemented on the same reconfigurable architecture defined by equation 3.10:

$$\begin{aligned} Z_2 &= x_3 \cdot x_2 + x_1 \cdot x_0; \\ Z_1 &= x_3 \cdot x_2 + \overline{x_1} \cdot x_0; \\ Z_0 &= \overline{x_3} \cdot \overline{x_2} \cdot x_1 + x_3 \cdot x_2 + \overline{x_1} \cdot \overline{x_0}; \end{aligned} \tag{3.10}$$

The name and number of inputs (x_3, x_2, x_1, x_0) and outputs (Z_2, Z_1, Z_0) are not changed. Table 3.20 now reflects the behavior of the new circuit. This table is constructed similarly to Table 3.18. For this second example, when the input byte

Table 3.20: List of context words for product terms of equation

Product Term	$P_0 = x_3 \cdot x_2$	$P_1 = x_1 \cdot x_0$	$P_2 = \overline{x_1} \cdot x_0$	$P_3 = \overline{x_3} \cdot \overline{x_2} \cdot x_1$	$P_4 = \overline{x_1} \cdot \overline{x_0}$
List	$(0C0C07)_H$	$(030304)_H$	$(030202)_H$	$(0E0201)_H$	$(030101)_H$

is $IN = (01)_H$, the outputs determined from equation 3.10 are:

$$\begin{aligned} Z_2 &= 0 + 0 = 0; \\ Z_1 &= 0 + 0 = 0; \\ Z_0 &= 0 + 0 + 1 = 1; \end{aligned} \tag{3.11}$$

It is evident that the proposed reconfigurable circuit uses the same reconfigurable circuit to implement a large number of combinational circuits, where each circuit has its own context words specifying its logical behavior.

Chapter 4

IMPLEMENTATION OF THE PRODUCT TERMS METHOD ON AN EXISTING RECONFIGURABLE DEVICE

Due to limited resources, it is very difficult to manufacture an actual reconfigurable hardware device based on the product terms method. Therefore, an existing reconfigurable hardware device referred to as the *hardware support* is used to host the new reconfigurable structure. A list of recommendations is given at the end of this chapter to increase the performances on an actual reconfigurable device based on the product term method.

4.1 Selection of the Development Board

Numerous development boards currently exist on the market with integrated reconfigurable circuits. The decision to take the ALTERA FLEX EPF10K70 FPGA device placed on an University Program Board UP2 is not an arbitrary one. The tool support is presented in Figure 4.1. The EPF10K70 device is based on Static Random Access Memory (SRAM) technology. It is available in a 240-pin Plastic Power Quad Flat Package (RQFP) and has 3744 logic elements (LEs) and nine embedded array blocks (EABs). Each LE consists of a four-input LUT, a programmable flip-flop, and dedicated signal paths for carry-and-cascade function. Each EAB provides 2048 bits of memory which can be used to create RAM, ROM, or first-in first-out (FIFO) functions. EABs can also implement logic functions, such as multipliers. With 70000 typical gates, the EPF10K70 is a reconfigurable device ideal for our proof of concept. Additionally, ALTERA offers an extended reference for the software[1] used to configure the FPGA device. Structure that implements general Moore sequential circuitsAll programs are written in VHDL language. The FLEX seven-segments LED (FLEX_DIGIT) on the ALTERA University Program board displays active outputs and requires no external wiring with the EPF10K70 device.

4.2 Implementing Synchronous Sequential Circuits

Synchronous sequential circuits, always use a synchronization clock and memory elements. Information is loaded in memory elements to establish the initial state or

[1] MaxIIPlus version 6

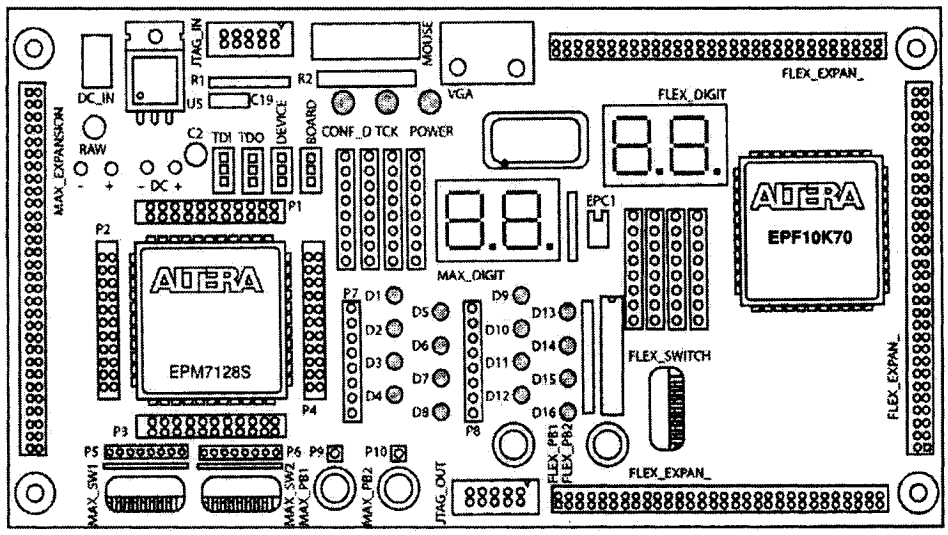

Figure 4.1: The ALTERA University Program board (UP2)

the next state of the circuit. The model of a synchronous sequential circuit in which outputs depend on both inputs and state variables is called Mealy machine model. In another model of synchronous sequential circuit known as Moore machine model, outputs depend only on state variables. It is always possible for any synchronous sequential circuit, to switch its representation from one machine model to another [7],[13]. The equivalence between the two models is done by using the sequential machine theory as it is described in []. In our proof of concept, the Moore machine model is chosen to represent all synchronous sequential circuits due to it simpler output equations.

Figure 4.2 shows an example of a reconfigurable circuit implementing a synchronous sequential circuits with clock input only. A feedback connection generates the next state at every clock input, based on the next state feedback equations. The outputs of the circuit are taken directly from the state register. The synchronous sequential hardware structure of Figure 4.2 works for each clock signal, in a similar

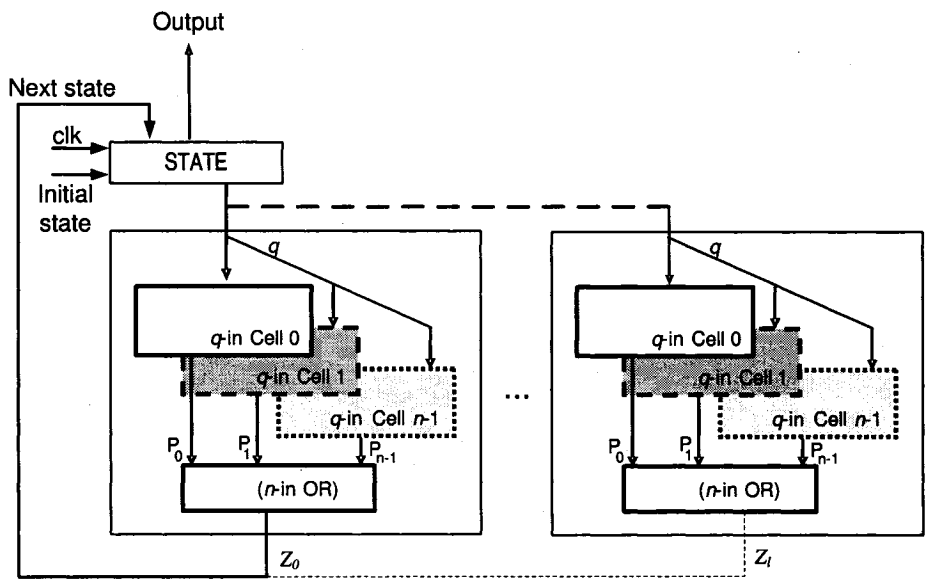

Figure 4.2: Structure that implements sequential circuits with clock input only

manner than that described for multi-output combinational circuits of Figure 3.2 or of Figure 3.8. When few product terms are repeated in the next state equations of the sequential circuits, the circuit similar to Figure 3.2 (like PAL solution) can be replaced in the feedback connection for each equation. Otherwise, when numerous product terms are repeated in next state equations of the sequential circuit, the circuit similar to Figure 3.8 (like PLA solution) is recommended.

Two input multiplexers can be added to the outputs of the next state circuit to select either the flip-flop outputs or the combinational part as actual outputs. In this way a new circuit is obtained and bounded by q inputs (or states), r outputs (or feedback equations) and n product terms, limits that define the reconfigurable circuit to implement any combinational circuit or any synchronous sequential circuit with clock input only.

Figure 4.3 illustrates a structure implementing a plurality of general Moore synchronous sequential circuits, using two reconfigurable circuits with the product terms

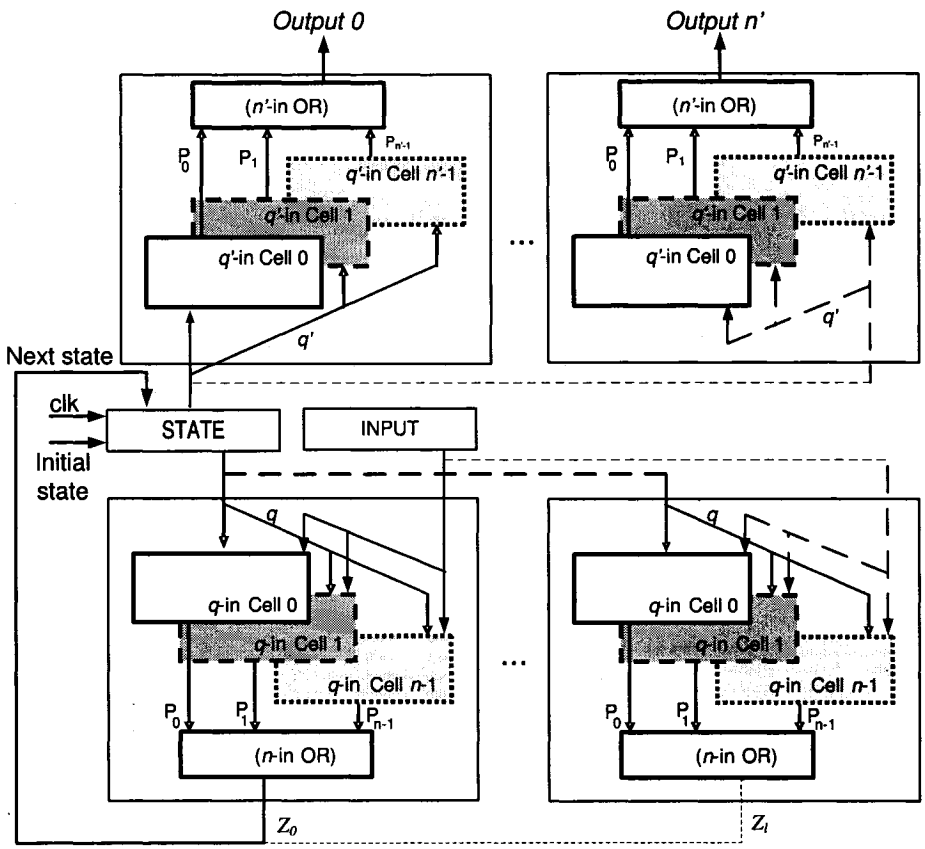

Figure 4.3: Structure that implements general Moore sequential circuits

method, namely the Next state circuit and the Combinational circuit for outputs. Only one basic cell is illustrated for both structures. The Next state circuit may have n number of basic cells and the Combinational circuit for outputs may have n' number of basic cells for implementing a maximum of n product terms in the next state equations and and a maximum of n' product terms in the output equations. In the mentioned figure, the input register and the state register have 8 bits, the registers for basic cells of the Next state circuit have 16 bits and the registers for basic cells of the Combinational circuit for outputs have 8 bits. In the general case the Next state circuit and the Combinational circuit for outputs can be of type Figure 3.2 or of type Figure 3.8.

The reconfigurable circuit presented can be combined to allow the flow of information in a similar way to that known for a classical direct-implemented network composed of several levels of gates. When the required number of product terms is larger than the number of basic cells for a given reconfigurable circuit of product terms method, two or several of them, even of different type, may be interconnected.

An example is presented in Figure [Figure 3.13], where a reconfigurable circuit based on product terms method as in Figure 4.2 is associated with two circuit for circuits as in Figure 3.8 to obtain a circuit that can implement different direct and reverse counters with outputs that use different decoders to 7 segments displays. In this particular case, if all three hardware structures are implemented on a single chip, the two input registers of the circuit specified in Figure 3.8 can be eliminated, because the state register of the circuit of Figure 4.2 can replace the eliminated registers.

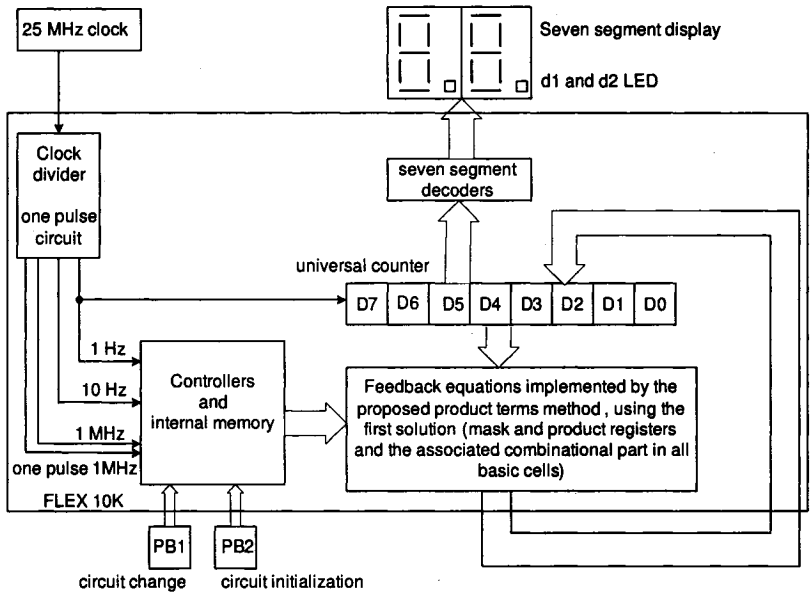

Figure 4.4: The proposed architecture of the implemented reconfigurable structure.

4.3 The Proposed Reconfigurable Structure

An actual implementation materializing any eight bit sequential circuit with clock input only will be given to exhibit the advantages of using the product terms method for a reconfigurable circuit. For this matter, any eight bit counter, any eight bit sequential controller, any eight bit or less pseudorandom generators, any traffic light with eight lamps or less can be considered for implementation on the reconfigurable circuit. Furthermore, the implementation generalizes any eight bit sequential circuit having only clock input. The proposed architecture of the implemented reconfigurable structure is developed with the following main functional components (Figure 4.4) in mind.

1. The hardware structure of product terms method materializes numerous sequential circuits with clock input only, having a maximum of 8 inputs, a maximum of 8 outputs and a maximum of 8 product terms for each feedback

equation. The first solution of the product terms method is considered with two registers (mask and product) in every basic cell as described in Figure 3.3.

2. The EAB memory blocks stores data characterizing feedback equations of the sequential circuits to be implemented.

3. A controller program offers the possibility to change the sequential circuits to be implemented.

4. A second controller program loads specific register arrays of basic cells with the corresponding data stored in EAB memories when the user requests to change the sequential circuit.

The performances of the proposed architecture such as speed and components density are limited to the board technology. For example, the possibility of organizing the embedded memory is limited by the EPF10K70's architecture.

The revolutionary approach is governed by the new hardware structure that physically implements the first solution of the product terms method. It is capable of materializing any sequential circuits with clock input only. As mentioned in Figure 4.4, the hardware structure has a register (synchronous eight bits D flip-flop register) and a reconfigurable block designed to materialize feedback equations. The reconfigurable block is built on basic cells, where any basic cell contains two context registers (mask and product) and a combinational part for each product term of feedback equations. Each feedback equation can have eight or less product terms. Consequently, 16 context registers of eight bits are created for each feedback equation.

During the initial phases, EAB blocks are loaded with context data characterizing product terms of feedback equations. Each EAB is formatted as a memory of 256 words of 8 bits. Each EAB stores context data for a single feedback equation of any sequential circuit. The hardware structure can implement any sequential circuit defined by eight or less feedback equations. Consequently, eight EABs (EAB0 to EAB7) are used and initialized accordingly. Figure 4.5 displays the first EAB (EAB0) which stores data for the first feedback equation for all considered sequential circuits. The demonstrative structure enables the implementation of eight different sequential circuits. Memory EAB0 stores context data of the first feedback equation among the eight different sequential circuits. Consequently, 16 different addresses are allocated for each one of these first equations.

The second controller program loads the context register arrays of basic cells with corresponding data stored in EAB memories. When the user requests to change the sequential circuit, all 8 feedback equations must be implemented in the proposed structure. Context data characterizing this circuit are placed at beginning of EAB memories. The first context register stores the mask word of first product term of first feedback equation (MASK00). The second context register stores the control word of first product term of first feedback equation (PRODUCT00). The process is repeated for the remaining context words of the first feedback equations as shown in Figure 4.6. During the initial stage, the 16 words characterizing the first feedback equation are stored in the first 16 memory words (addresses from $(00)_H$ to $(0F)_H$) of EAB0. When the second controller is activated, MASK00 word found in EAB0 at address $(00)_H$ is transferred in first register *R0* and the word PRODUCT00 found

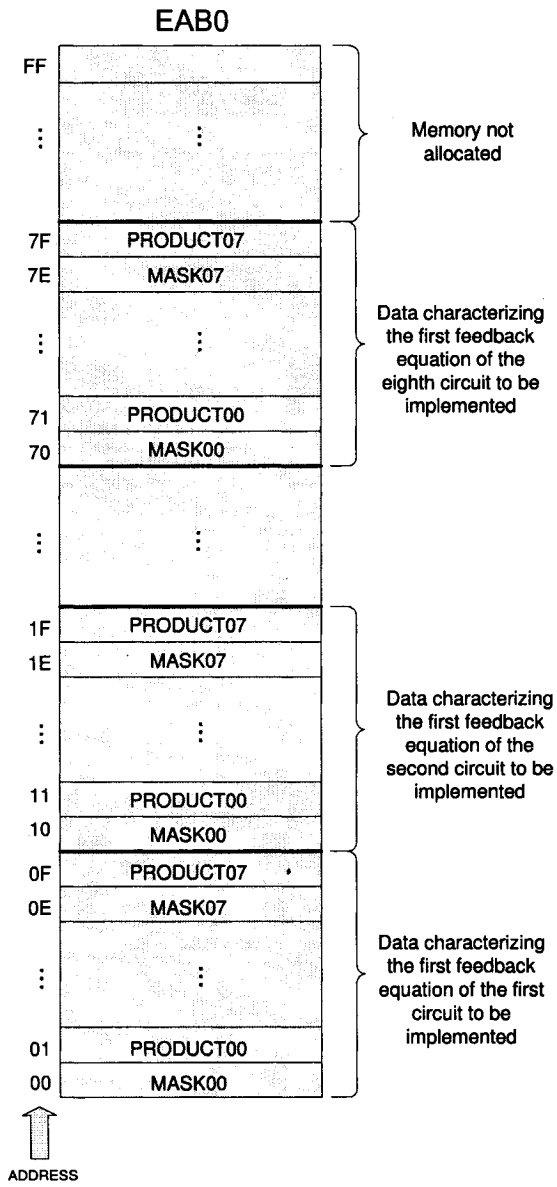

Figure 4.5: Data stored in EAB0 memory.

in EAB0 at address $(01)_H$ is transferred in the second register *R1*. In the same way, specific context registers are allocated with the data of the second feedback equation. Precisely, the 17^{th}specific register (R16) is allocated with the mask word of the first product term of the second feedback equation (MASK10), respectively the 18^{th}specific register (R17) with the control word of first product term of second feedback equation (PRODUCT10) and so on. It must be pointed out that during the initial stage, the 16 words which characterize the second feedback equations are stored in the first 16 memory words having the same addresses, from $(00)_H$ to $(0F)_H$, of the second EAB, precisely EAB1. For all other logic equation of the circuit, the context registers and the EABs are loaded and controlled in the same way by the controller program. It must be noted that data MASK00 has the same memory address, precisely $(00)_H$, in EAB0 as data MASK10 in EAB1. Therefore, when the command to change the sequential circuit is done, data is transferred in parallel from EAB memories to context registers.

4.4 Description of VHDL Programs for Implementing Circuits With Clock Input Only

The VHDL programs are written in a hierarchical approach [5],[24],[25],[29].

- *circ_counter.vhd* (PROGRAM VHDL_1) is the top-level program in the design. It includes the hierarchical highest level program *counter.vhd* that defines the reconfigurable architecture. Also *circ_counter.vhd* includes programs to generate numerous clock signals and to link the FLEX10K device with onboard

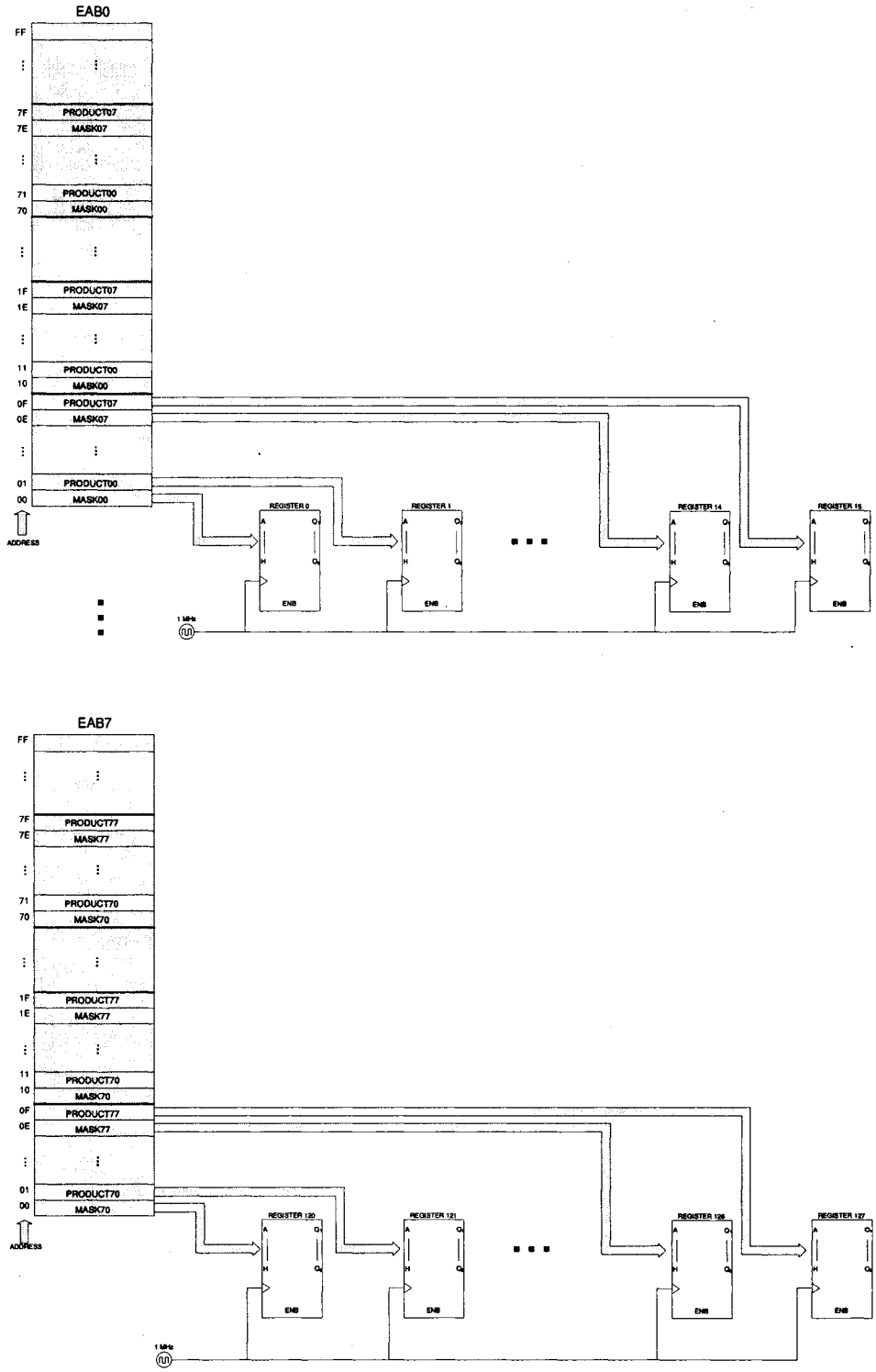

Figure 4.6: Loading of context words in specific registers.

peripherals. The UP2 core library functions are used in VHDL for these tasks. PORT in the ENTITY of the program *circ_counter.vhd* connects the top-level program with the FLEX10K onboard components. The two 7-segment display devices tied to the FLEX10K are used to output the information. MSD is for the most significant digit and LSD is for the least significant digit. *pb1* and *pb2* are the command pushbuttons. *pb1* changes the circuits in a cycle and *pb2* initialize the active circuit. *pb1_single_pulse* is a single pulse signal (1Mhz) used by the *pb1* pushbutton when the command is triggered by the user.

- *clk_div.vhd* (from UP2 core library functions) provides all clock signals:

1. clock (1Hz) controls active states of the implemented sequential circuit. This clock signal is chosen at low speed, for having the possibility to follow visually on the 7-segment LEDs the states during the motion of sequential circuit;

2. clock3 (1Mhz) controls at high speed all operations to load context registers from memories after the user has triggered the command to change the implemented circuit.

- *debounce.vhd* (from UP2 core library functions) is a pushbutton debounce program used to filter mechanical bounce on the *pb1* pushbutton.

- *onepulse.vhd* (from UP2 core library functions) is a single pulse circuit required by the *debounced.vhd* pushbutton program.

- *counter.vhd* is the highest level program which demonstrates the product terms method, and in particular, it implements the controller program used when the user commands a change in sequential circuits.

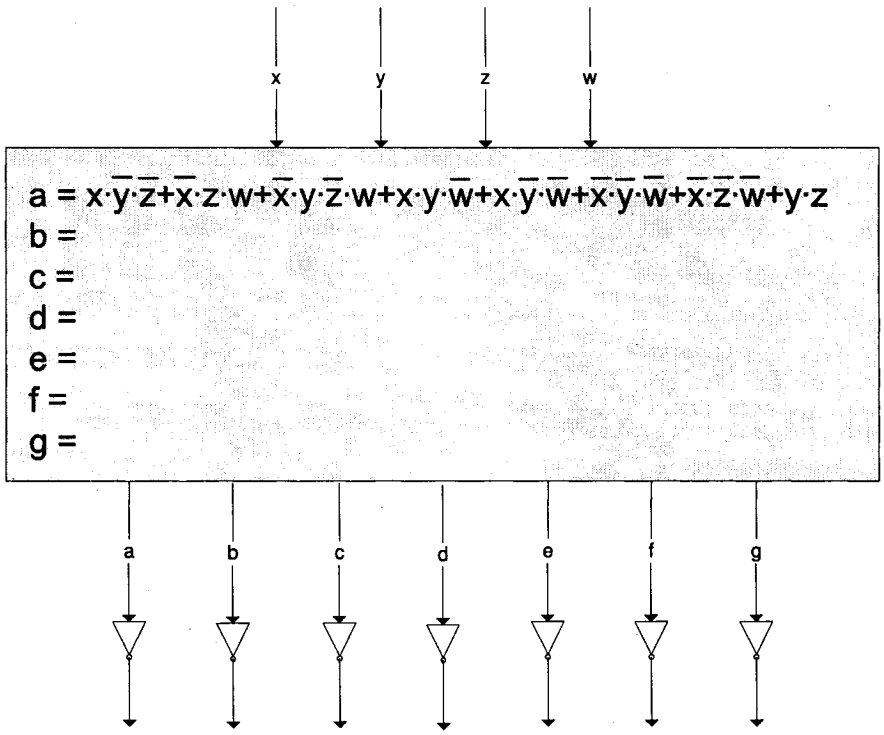

Figure 4.7: Block diagram of Hex_7seg.vhd program

- *hex_7seg.vhd* is a (PROGRAM VHDL_2) hexadecimal to 7-segment decoder with active low outputs (the block diagram is shown in Figure 4.7 and is different from the similar program existing in the UP2 core library). The decoder uses directly the logical equations of the segments (a, b, c, d, e, f, g). This program implements the hardware structures twice. Precisely, *decoder1* is a seven-segment decoder for the least significant hexadecimal digit and *decoder2* is a seven-segment decoder for the most significant hexadecimal digit. LED driver circuits are inverted, as the University Program board needed it.

- *counter.vhd* (PROGRAM VHDL_3) has a hierarchical structure. It includes the hierarchical second lower level program named *circ0.vhd* that defines the memory blocks and implements the second controller program, which allows

to load the specific register arrays of basic cells with the corresponding data stored in EABs, when a command of changing the target sequential target circuit is done. It also includes the program *reg_count.vhd* that implements the synchronous eight bits D flip-flop register as a part of the structure designed to materialize sequential target circuits with clock input only. Additionally, *counter.vhd* accomplishes two major tasks:

1. it connects the block of feedback equations and the eight D flip-flops of the mentioned synchronous register, where *count_in* and *count_out* are the inputs and the outputs of the register and *CINPUT* and *reac* correspond to the inputs and to the outputs of the block that materializes the feedback equations;

2. using n as a three bit parameter, it implements the controller program (Figure 4.8) which allows the user to choose in a cycle mode among several sequential circuits preloaded on EAB memories of the device.

To implement the controller that allows choosing between several sequential circuits, a process statement is written in *circ_counter.vhd* program. The value of three bit parameter n is used to initialize all the pointers which indicate the address of the first data in all eight feedback equations that characterize a circuit. Remember, the data describing the product terms of feedback equations are stored in parallel in EAB memories. With n as three bit parameter, eight different circuits can be materialized as presented in Table 4.1. If more circuits must be materialized the parameter n must be changed to more bits end the process statement must be modified. The seven-segment display decimal points, precisely *d1* (right) and *d2* (left) visually indicates which is the active sequential circuit. $d1 = 0$, means d1 off,

Table 4.1: List of Implemented Reconfigurable Circuits

parameter n (3 bits)	Circuit Implemented	*d1*	*d2*
000	8 bit Direct Binary Counter	0	0
001	8 bit Inverse Binary Counter	1	1
010	2 Decade Direct BCD Counter	1	0
011	2 Decade Inverse BCD Counter	0	1
100	Modulo 12 Direct Counter	*clock*	*clock*
101	Excess-3 Direct Counter	*clock1*	*clock1*
110	8 bit Pseudo-Random Generator	*clock1*	0
111	8 bit Special Controller	0	*clock1*

d1 = 1, means *d1 on,* d1 = *clock*, means *d1* blinking at low speed and *d1* = clock1, means *d1* blinking at high speed. With this explication the process chooses, by the value of parameter n, one of the eight possible sequential circuits.

Linking the block of feedback equations and the eight D flip-flops of the synchronous register is done by connecting the block outputs *reac* to the register inputs *count_in* and the register outputs *count_out* to the block inputs *CINPUT.*

- *reg_count.vhd* (PROGRAM VHDL_4) is a classical synchronous 8 bits D flip-flops register program, having D as inputs, Q as outputs, *ck* as clock input and *rt* as reset input. It works at 1Hz clock. Its inputs D are connected to the outputs and its outputs Q are connected to the inputs of the block that implements the feedback equations of the sequential circuit using the product terms method.

- *circ0.vhd* (PROGRAM VHDL_5) has a hierarchical structure and it contains two parts:

1. implements the 8 EAB memory blocks (Embedded Array Blocks) for storing the data that define the sequential circuits, and

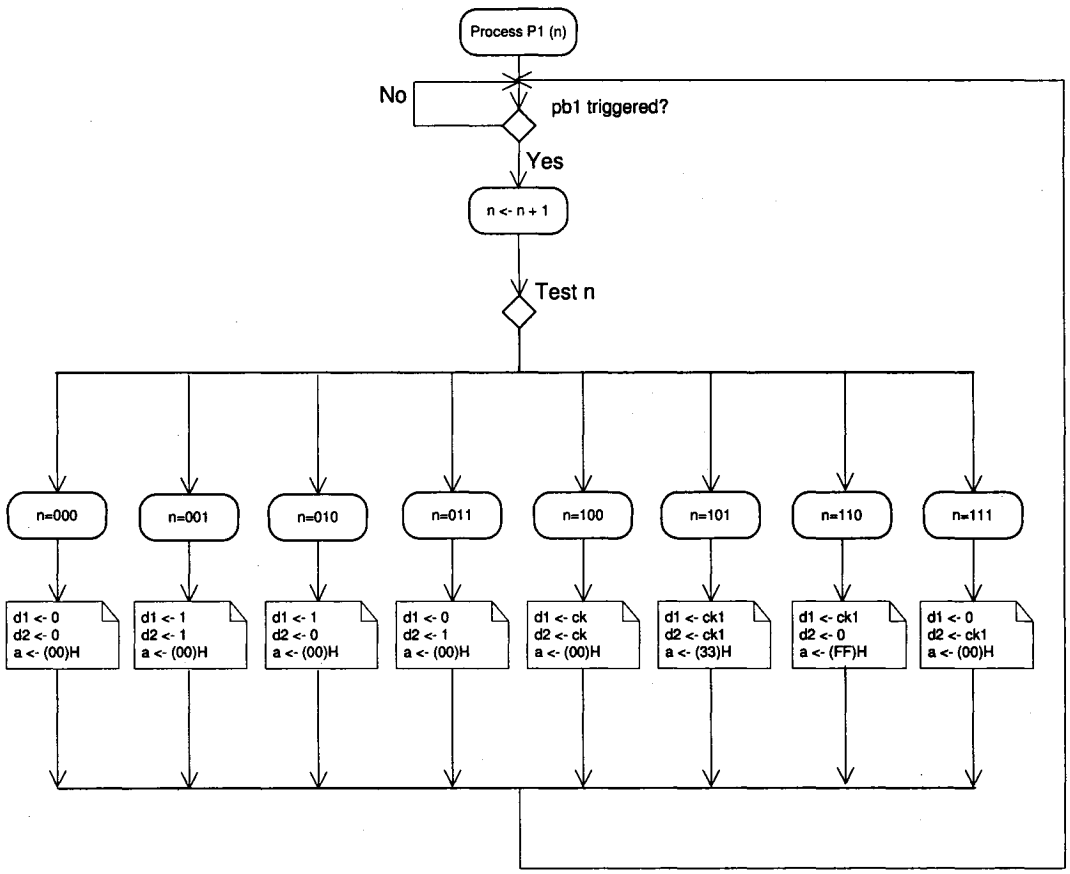

Figure 4.8: Block diagram of the controller that allows choosing between several sequential circuits.

2. implements a controller program to load specific register arrays with the corresponding data stored in memory, when a command of changing the sequential circuit is done.

INPUT is the input vector (8 bits) of the feedback logical equations. *My_OUTPUT* is the output vector (8 bits) of the feedback logical equations. *R0* to *R7* are register arrays (each register array has 16 registers). A register array stores 16 words (every word has 8 bits) that characterize eight product terms defining a logic equation. Precisely eight mask words, *MASK0* to *MASK7*, and eight control words, *PRODUCT0* to *PRODUCT7*. *RA* is an array of 8 address registers. *RD* is an array of 8 output data registers.

Memory blocks, *mem0* to *mem7*, are implemented by 8 EABs with the *lpm_ram_dq* function. *mem0* stores the information concerning the first feedback equation of all circuits. *mem1* stores data for the second feedback equation and so on. Each EAB memory is configured as 256 words of 8 bits and is loaded with a *.mif* file during initial phase. A *RA* register and a *RD* register are attached to each EAB to allow memory read operations. A sequential circuit is defined by 8 feedback equations. An EAB stores the information characterizing one feedback equation at 16 successive addresses. Data defining 8 feedback equations are distributed in the 8 EABs having the same address for the words characterizing the 8 different feedback equations which must be loaded in registers. The data transfer from the 8 EABs to registers at same address in register arrays is done in parallel. The pointer is initialized at the beginning of data blocks (8 blocks that work in parallel) characterizing the feedback equations defining the circuit. *contr_counter* controls addresses inside of data

blocks for the feedback equations defining a circuit. *state* defines the five state of the controller used to load registers from memory. These states are: *init_reg_address*, *test_contr_counter*, *calculate_address*, *write_register*, *end_load.*

For a feedback equation of any circuit, a memory allocates 16 words, 8 for storing mask words and 8 for storing control words. Feedback equations of circuit to be implemented are limited by 8 product terms. For equations having less than 8 product terms, unused product terms (for example if an equation has 5 product terms, the number of unused product terms are 8 minus 5, so 3) must still be initialized in the *.mif* file. Mask words are loaded with $(00)_H$ and control words are loaded with $(FF)_H$.

circ0.vhd uses in its lower level, *function0*, *function1*, ... programs that implement the product terms method for generating the outputs of feedback logical equations *(func.vhd)*. The controller program loads the registers with data from EAB memories when the command of changing the sequential circuit is done. This controller is the most difficult part of the programming task. The difficulty is caused by synchronizations problems during the evolution from one state to other in sequential circuits. As was mentioned, the controller program part uses five different states:

1. the state *init_reg_address* initializes the address registers and the pointer considering the parameter n;

2. the state *test_contr_counter* tests the address controller to find the end of loading;

3. the state *calculate_address* calculates memory addresses of data to be loaded in parallel into registers;

4. the state *write_register* transfers data (8 data in parallel) from memories to the corresponding context registers;

5. the state *end_load* indicates the end of register's loading.

The functionality of this controller is shown in the block diagram of Figure 4.9. Normally the transfer of a word, from memory to the corresponding register, needs three clocks for passing by states *test_contr_counter*, *calculate_address* and *write_register*. Consequently, the total time needed for changing from one circuit to another circuit is 48 clocks. The states *init_reg_address* and *end_load* are used only once for each changing command of the circuit. A snapshot of the time diagram portrays the functionality of the controller in Figure 4.10. The clock is tested to '0' to produce a delay (a semi period) of the memory read operation. In this way the address remains stable during the reading process. *p1* is tested to '1', because it produces a positive one pulse signal. During the address calculation, the same address is loaded in all address registers and therefore the transfer of data from memory registers of the 8 EABs to registers allocated in the same position in register arrays is done in parallel.

- *func.vhd* (PROGRAM VHDL_6) implements the combinational circuit that calculate a value of a logical function using the product terms method. *MASK0* to *MASK7* and *PRODUCT0* to *PRODUCT7* are inputs for the program. *INP* represents the active inputs (8 bits) of logical circuit and *RES* represents the output of logical circuit. Every logical equation is expressed by sum-of-product form, and can have a maximum of eight product terms. To calculate the value of a product term for given inputs the product term method implements

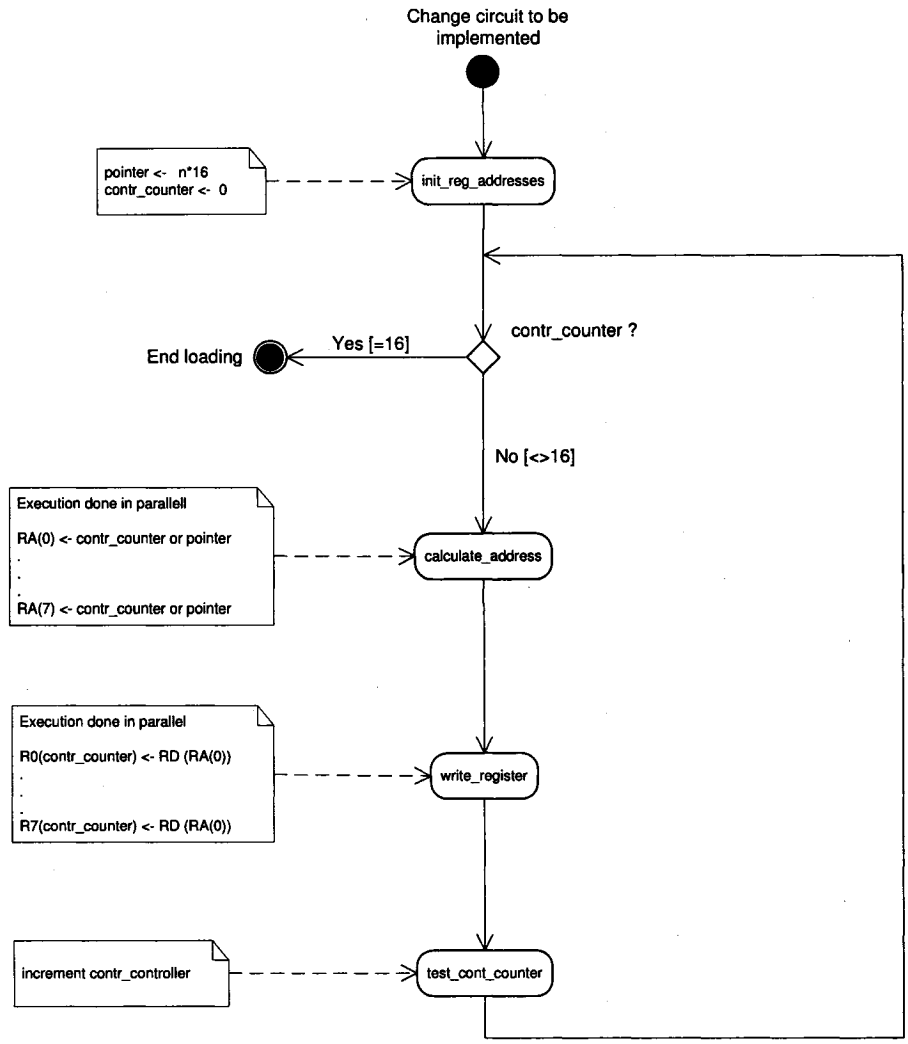

Figure 4.9: The block diagram of the controller program which loads the specific registers with the corresponding data stored in EABs.

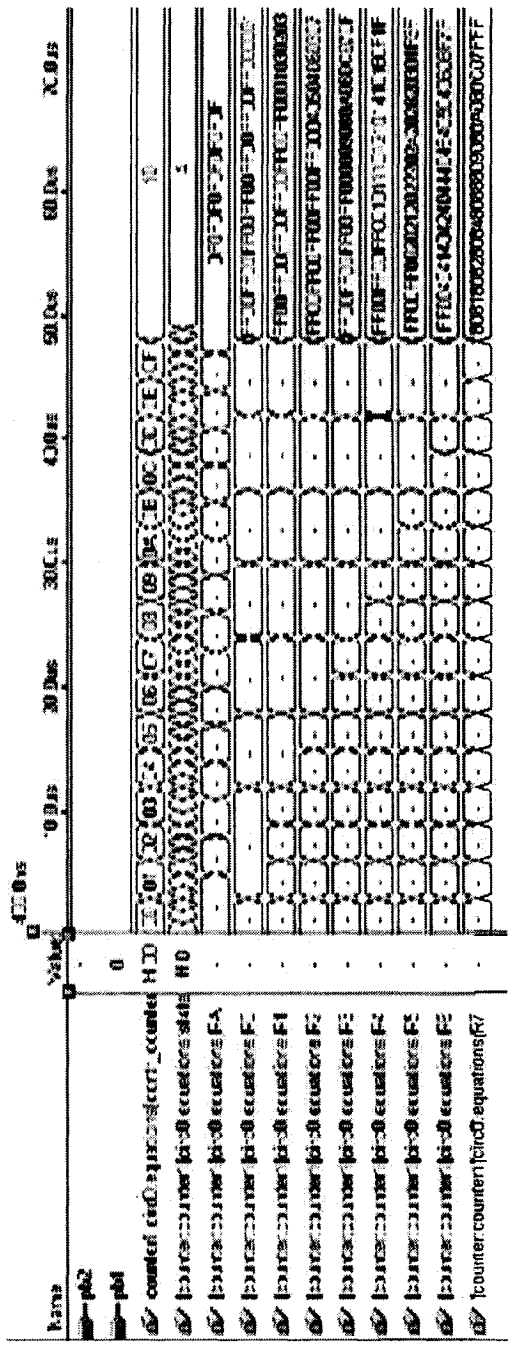

Figure 4.10: Snapshot of time diagram for the controller

the program *cell.vhd* and connect the eight cells with the final *orgate.vhd* for determining the value of logical function.

- *cell.vhd* (PROGRAM VHDL_7) calculates the logical value t of a product term for given inputs I (the logic of the cell was shown in Figure 3.3). Every cell needs at input a mask word M and a control word P. The combinational part which calculate the value of a product term is implemented by the AND operations between the inputs and the mask word bits to establish the variables of the product term (first logical level) followed by the XNOR operations between the precedent result and the control word bits to establish if, for the given inputs, each active variable has the same logical value as it has in the control word (second logical level) and followed by the final AND operation to establish if the considered product term has the logical value 1 for the considered inputs (third logical level).

- *orgate.vhd* (PROGRAM VHDL_8) calculates the value of a logical equation for given active inputs, by OR-ing the outputs of the eight product terms.

4.5 Development of Software Environment in C#

The manual computation of tables describing the behavior of any implemented combinational circuit or any combinational part of a sequential circuit is cumbersome and error prone. Consequently, the development of a software environment is recommended to generate the context words required for the product terms method. A software environment of this nature is not unique. It is dependent on numerous

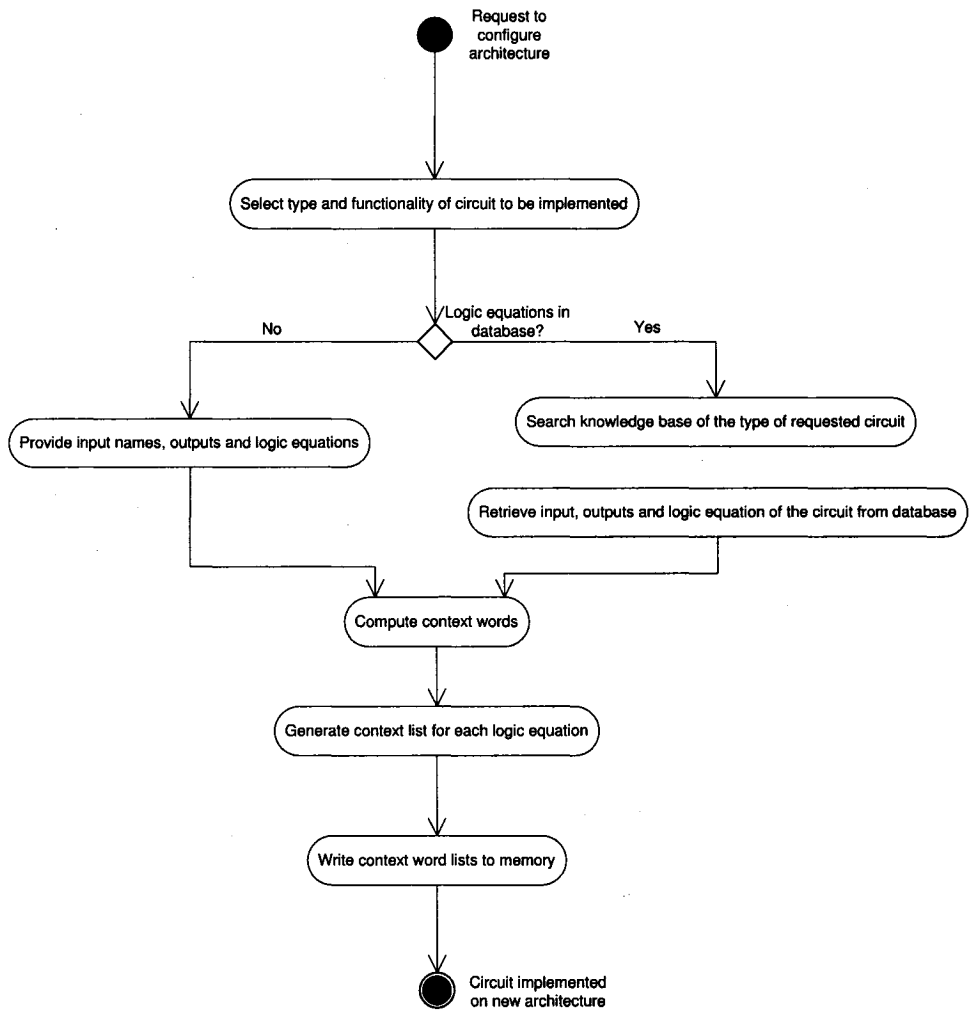

Figure 4.11: Activity diagram of software environment

parameters such as the number of context words used in the reconfigurable cell. An activity diagram describes the behavior of the software environment in Figure 4.11 for generating the context words list used in the product terms method. Once a request to implement the circuit on the new architectures has been raised, the software environment interacts with the user or with a running program and offers the following choices:

- insert the behavior of the selected circuit for inputs, outputs and SoP equa-

tions, or

- consult a database with behavior of circuits stored in association with a large number of possible circuits, and retrieving the inputs, the outputs and the equations associated with a required circuit.

For the latter choice, a knowledge base driven by a classical backward-chaining inference engine displays the results. The inference engine works in query mode, matching the criteria of the required circuit. The knowledge base contains parametric description of circuits that can be implemented with the corresponding record in the database. Once the circuit has been identified from the database, the system loads data describing the circuit behavior, namely input variables, outputs and logic equations. The description of several groups of logic equations with their inputs and outputs will be provided in the database to characterize a complex structure.

The most elaborate part of the software environment is the implementation of the algorithm to compute the context words used by the product terms method (Figure 4.12).

Initially, input variables, outputs and symbolic representation of logic equations are checked to verify if the following conditions, which are established in a conventional mode and can be modified in number and each one in its state are applied accordingly:

- each input variables appear as lowercase letters;
- the number of input variables does not exceed the input word length;
- the same letter is not repeated in the input string;

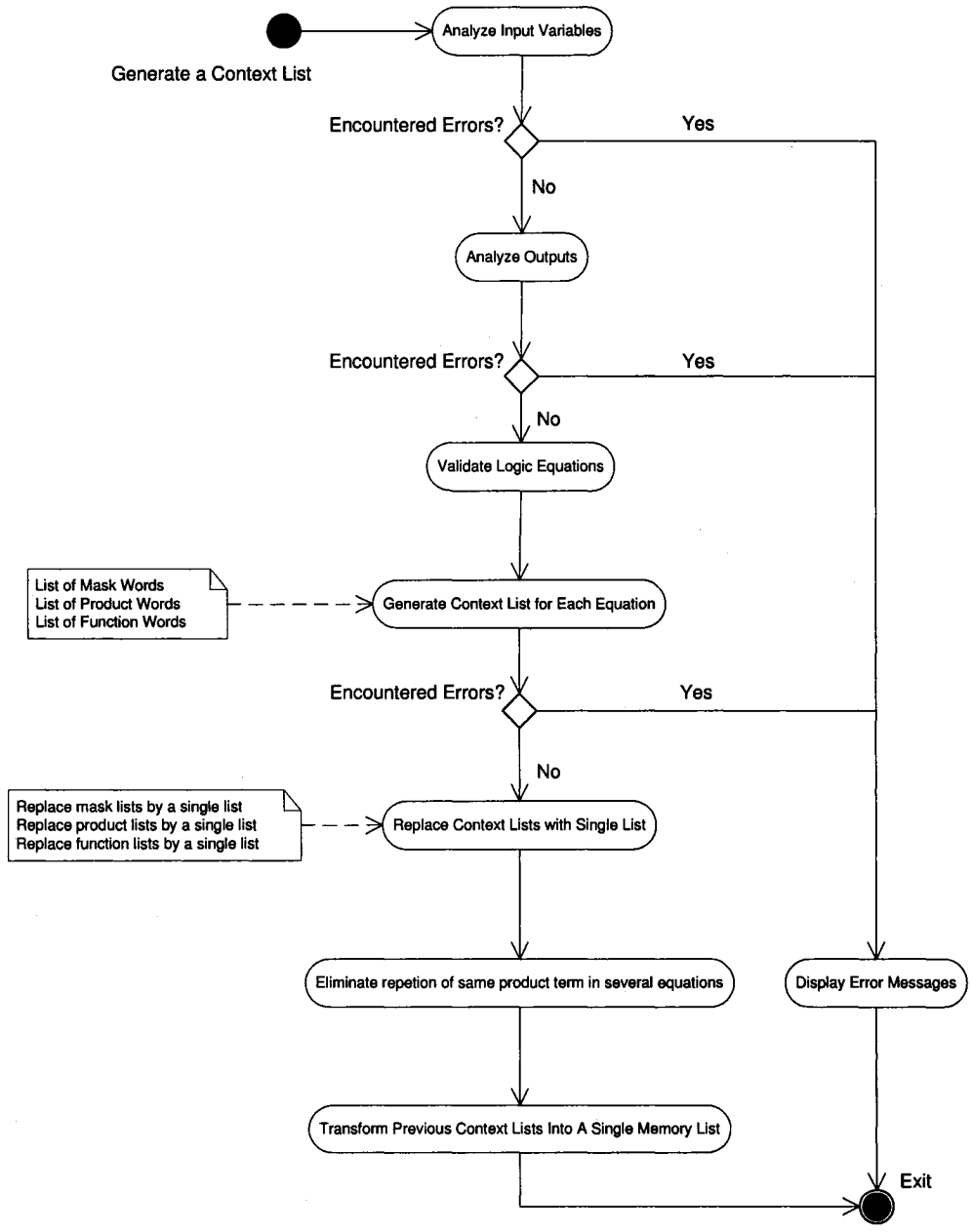

Figure 4.12: Activity diagram for generating a context list

- outputs appear as uppercase letters;
- the number of outputs does not exceed word length;
- the same letter is not repeated in the output string;
- a logic equation contains only lowercase letter, underscore character (used to denote negation) and plus sign;
- the same variable is not repeated in a product term;
- a product term does not begin with the underscore symbol;
- underscore symbols do not follow one another;
- OR logic operations do not follow one another;
- a logic equation does not begin with the plus symbol;
- a logic equation does not end with the plus symbol;

If all these conditions are satisfied, the software environment generates the lists of mask words, the lists of control words, and the lists of function words. Next, the software transforms a group of lists of the same type into a single context list and eliminates the repetitions of the same product term. Finally, the software environment combines the three lists (mask, control and functions) into a single list that expresses the behavior of the combinational circuit or of the combinational part of the sequential circuit.As was mentioned, the proposed architecture of the reconfigurable structure can implement sequential circuits with clock input only in the limits established by the parameters: q (maximum of inputs), r (maximum of

outputs) and n (maximum of product terms in each logical feedback equation). As was mentioned also, in the first solution of the proposed product terms method, each product terms of any logical equation is characterized by two data: the mask word and the control word. The manual computation of context lists containing the mask words and the control words are cumbersome and error prone. Therefore, this task is usually completed by a software environment. The software environment must be developed according to hardware architecture of the served structure, and must be integrated with others software (like compiler, fitter, time analyzer, etc.) which help to implement the target circuit.

We now propose to build a simple software environment written in C# [20],[28] to compute the context words as described above. It can be written in any other popular programming language such as C, C++ or Java. The software environment must be adapted to the proposed hardware structure of the product terms method that implements synchronous circuit with clock input only limited by the following constraints:

- maximum 8 inputs;
- maximum 8 feedback equations (outputs);
- maximum 8 product terms in each equation.

The software environment generates all data from logic equations. Therefore, the software environment input is a text file which respects a rigorous syntax, and its outputs are arrays formed by hexadecimal numbers expressed by two digits (to be loaded in 8 bit memory words or/and 8 bit registers). To simplify the program, the

software environment does not make any other task, as to validate the inputs, or to validate the outputs, or to validate the syntax of equations. We suppose that the input text files were verified (manually or by a specific program), so these text files do not have writing or syntax errors.

The syntax rules of the text input file must first be established for the development of software environment. In our example, input text files must fulfill the following syntactic conditions:

- the input file contains lines of text, where each line is formed by groups of characters named tokens;
- a token is an association of one or several characters, without the space character;
- space character is the tokens separator;
- each line must end with an *end* token;
- the end token was chosen the word *end*;
- the first line of the input file define the input variable;
- the second line of the input file do not have useful data, it serves to understand the signification of subsequent lines, by containing the token word *equations* followed by the *end* token;
- the equation lines begin with the line three;
- the underline character placed after the name of a variable specify the negation of this variable in the considered equation line.

The *StreamReader* function was instantiated in *C#* to read data from a file. The program uses this function to open and to read the text file containing the information that characterizes the circuit. As example, input file *BINinverse.txt* contains the information that describes, respecting the mentioned rigorous syntax, an eight bit binary inverse counter. After opening the file, the read operation followed by a write operation is completed on the console. Next, the line is split in tokens using the *split()* function and the space character as separator. In the following step the tokens of a line are analyzed successively, to extract the useful tokens. During the examination of equation lines which contain product terms as useful tokens, it must be pointed out the subsequent observations:

- the tokens that represents product terms in feedback equation lines begin with the line 3;
- the tokens that represent product terms in feedback equation lines were placed beginning with the third position in such a line;
- the tokens + and *end* must be eliminated from the useful tokens find in a equation line.

Once then a product term token was separated, it is placed in the two dimensional string array named *word1*, where the first index i mean the equation index and the second index j mean the product term index in the considered equation. In the same time, to follow the program evolution, the identified component of the string array *word1*, which is the array of product terms, is written on console, where the considered element of the array, with correct indexes and the literal expression of

the product term, are printed. Through this task of selecting the useful tokens from equation lines, attention was made to a right evolution of indexes. For instance, when a token analyzed is + the index j do not increase, or when the token analyzed is *end* the index j must be initialized to 0. In this way all useful product term tokens are placed in *word1* string array, subsequent to a reading operation, line after line, which was done in a loop from the input file.

In the next stage of the program the token containing the input variables is split to identify its components. Considering the order of these variables in the mentioned token, for each one a weight having a positional value of power of two is attributed. At the end of its treatment, the name of each variable, its position and its weight are printed.

Having now the string array of product terms *word1* and the numerical value array of input variable weights, the next stage of the program calculates the values of mask words and the values of control words. First, the string representing a product term is split in the characters which compose this product term. The integer r controls the number of characters, not the number of variables, in a product term. This is done, because among the characters of type letter which represent the input variables, the character _ (negation) can be found. Next the two dimensional integer arrays *mask* and *prod* are initialized at 0. Follow a loop analysis of characters which compose a product term. If a character represents an input variable found in a product term, the corresponding mask word *mask(x,y)* and control word *prod(x,y)* are augmented with the value of variables weight. If the character is underline that means that the previous character was in negation form, the control word is

decreased with the value of correspondent weight. When the loop analyzing of a product term was finished, the corresponding mask word and corresponding control word are printed. A second high level loop repeats all these operations for all product terms.

In the next stage of the expert system program the memory words matching to mask words and to control words are calculated. The proposed expert system, serving the proposed reconfigurable structure, considers all target circuits as sequential circuits with clock input only, having eight bits in state registers, as a result eight inputs and eight feedback equations, where each equation contains a maximum of eight product terms. Therefore, for each equation eight product terms are considered and implemented in hardware, with 16 specific registers (eight for mask words and 8 for control words). If a real equation has less than 8 product terms the hardware implemented but not used specific registers must be filled, precisely a not used mask word register with $(00)_H$ and a not used control word register with $(FF)_H$. In the proposed environment software program all mask word registers are filled at beginning with $(00)_H$ and all control word registers are filled at beginning with $(FF)_H$ as an initializing step. For the numerical values of components that characterize the product terms find in feedback equations are used the two dimensional integer array *word2*. In the subsequently step each real calculated value of a mask word or of a control word characterizing a real product term replace in *word2* array the previous initializing value.

In the last stage of the expert system the numerical values of mask words and control words are printed line by line, which correspond to succession of feedback

equations in hexadecimal forms. These values must be loaded in the EABs memory using .mif files at configuration phase of EPF10K70 device.

The following listing displays the input file *BINinverse.txt* as an example which characterizes the binary inverse counter.

```
input variables: abcdefgh end

equatons; end

f0 = a_ end

f1 = a_b_ + ab end

f2 = a_b_c_ + ac + bc end

f3 = a_b_c_d_ + ad + bd + cd end

f4 = a_b_c_d_e_ + ae + be + ce + de end

f5 = a_b_c_d_e_f_ + af + bf + cf + df + ef end

f6 = a_b_c_d_e_f_g_ + ag + bg + cg + dg + eg + fg end

f7 = a_b_c_d_e_f_g_h_ + ah + bh + ch + dh + eh + fh + gh end
```

Considering the order of input variables in the token found in the first line of the input file, each variable has a weight of power of two, taking into consideration its position in the token (a is the first variable having the weight 2^0) as it shown below.

```
a=1

b=2

c=4

d=8

e=16

f=32
```

```
g=64

h=128
```

After execution of the software for the given example, the output file is shown in the following listing.

Memory words for f0 in hexadecimal:

```
1,0,0,FF,0,FF,0,FF,0,FF,0,FF,0,FF,0,FF
Memory words for f1 in hexadecimal:
3,0,3,3,0,FF,0,FF,0,FF,0,FF,0,FF,0,FF
Memory words for f2 in hexadecimal:
7,0,5,5,6,6,0,FF,0,FF,0,FF,0,FF,0,FF
Memory words for f3 in hexadecimal:
F,0,9,9,A,A,C,C,0,FF,0,FF,0,FF,0,FF
Memory words for f4 in hexadecimal:
1F,0,11,11,12,12,14,14,18,18,0,FF,0,FF,0,FF
Memory words for f5 in hexadecimal:
3F,0,21,21,22,22,24,24,28,28,30,30,0,FF,0,FF
Memory words for f6 in hexadecimal:
7F,0,41,41,42,42,44,44,48,48,50,50,60,60,0,FF
Memory words for f7 in hexadecimal:
FF,0,81,81,82,82,84,84,88,88,90,90,A0,A0,C0,C0
```

4.6 Recommended Hardware Architecture of a Actual VLSI Device Based on the Product Terms Method

As was mentioned earlier, we used an existing reconfigurable hardware VLSI device, precisely an FPGA device (ALTERA EPF10K70) to implement the proposed reconfigurable structure using the product terms method which allows materializing the example sequential circuits with clock input only. Consequently, this implementation is limited in performance by the architecture of the hardware support. When building an actual VLSI device following the product terms method, a set of recommended modifications of its hardware structure must be taken into consideration. To have an easy comparative analysis, we will focus our recommendations for the hardware architecture of a structure having the same characteristics as that shown in Figure 4.4, particularly:

1. VLSI device designed to materialize sequential circuits with clock input only with maximum of 8 inputs, maximum of 8 outputs and a maximum of 8 product terms for each feedback equation.

2. VLSI device having a memory block for storing data characterizing the feedback equations that express the sequential target circuits.

3. VLSI device having a controller program, which allows to the user to choose between several sequential circuits and to load the specific register arrays of basic cells with the corresponding data stored in memories, when a command

of changing the sequential target circuit is done.

To implement the basic cells, the first solution of the new architecture of product terms method, which works similar to the PAL structural design must be chosen. Remember, in the architecture of CPLDs the PALs are preferred, because they offer better performance (higher speed and fewer switches on chip) and because they are less expensive (only the AND plane is programmable). Similar observations can be formulated in the case of the product terms method, where the first solution is preferred.

But the biggest modification that can be done is by changing the structure of the internal memory that stores the data characterizing the feedback equations of circuits to be implemented. Keep in mind, when a command to implement a new circuit is done, the controller of the actual implemented structure, which load the specific register arrays with the data stored in memories is activated. Even if this loading is accomplished in parallel by equations, it is done serially by product terms of each equation. And each register need three clocks to be loaded, so two times three clocks for each product term (mask and product registers). We can implement a structure where all data needed by all product terms of all feedback equations will be loaded in a single clock impulse in a unique resister. This unique register replaces the array of arrays concerning the mask and product registers attached to product terms. Precisely, for the mentioned structure with 8 inputs, 8 outputs and 8 product terms for each feedback equations characterizing a circuit, the unique register must have 1024 bits (8 equations times 8 product terms times 16 bits for mask and product registers). Therefore, instead of several arrays of 8 bits registers

a single register with 128 fields of 8 bits of information must be created. In this single register field 1 replace the mask register for the first product term of first feedback equation, field 2 replace the product register for the first product term of first feedback equation, and so on, as it is shown in Figure 4.13. In the same time all internal memories are replaced by a single SRAM internal memory having very long words, in our example 1024 bits. More than that, the controller that loads the mask and product register can be eliminated. The hardware structure becomes simpler and the speed of changing the target circuit attains its highest limit possible, precisely only a single clock pulse.

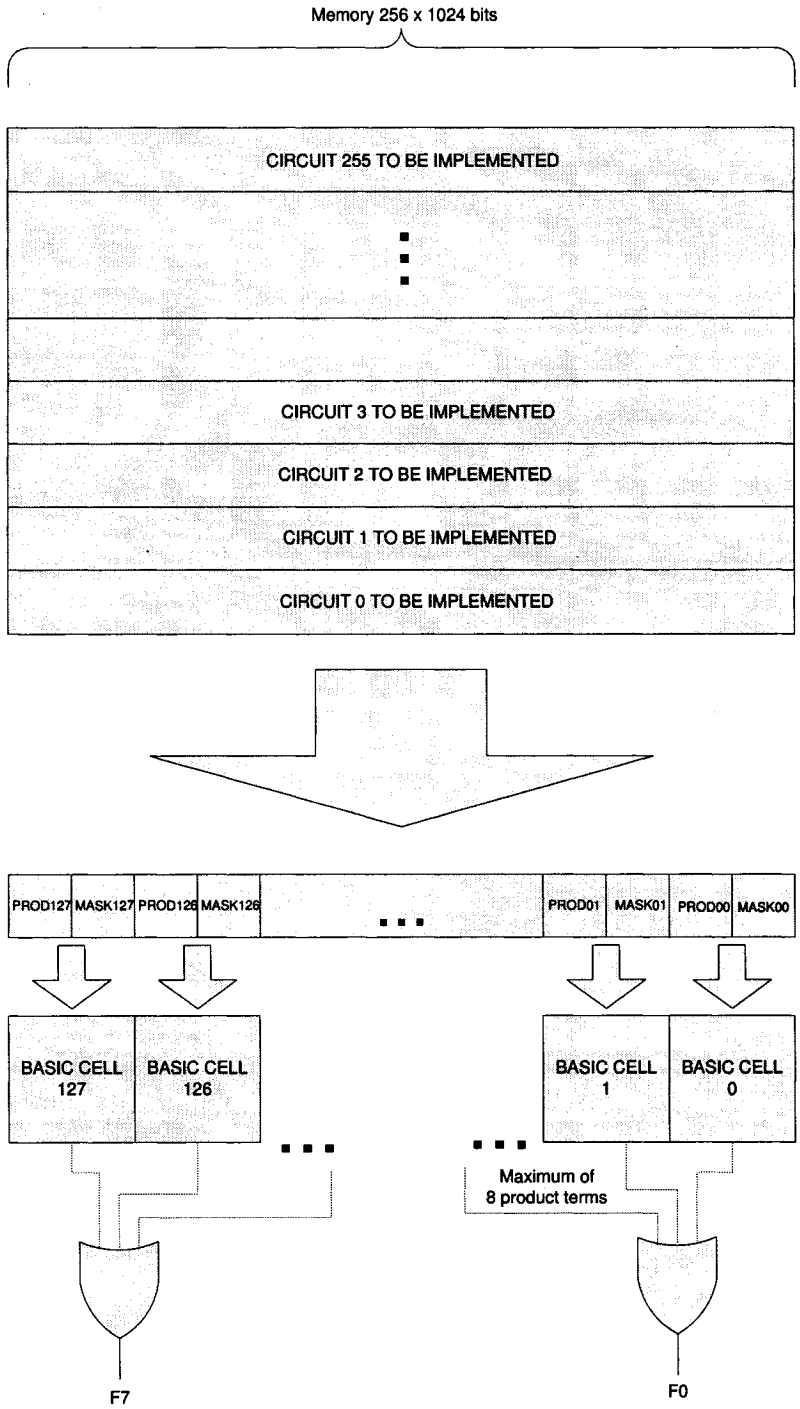

Figure 4.13: Architecture of future VLSI device which implements the product terms method

Chapter 5

CONCLUSION

Today's PLDs have aggressively driven down both die size and package costs to provide a persuasive ASIC alternative. The advanced system capabilities of modern PLDs allow designers to integrate discrete component functions into the PLD reducing board and total component costs. Two families of programmable logic devices are used today. Complex Programmable Logic Devices (CPLDs) directly implement the SoP form of logic equations describing the circuit at hand and Field Programmable Gate Arrays (FPGAs) implement through Look-Up tables, the truth table of logic circuits. CPLDs are usually register-limited in comparison to FPGAs, but have the capacity for very complex combinational logic with a single macrocell. This allows a complex function to propagate very quickly, often faster than in a FPGA of equivalent speed grade. CPLD are typically non-volatile, consequently it must be reprogrammed for a new configuration and may result in the physical smallest implementation using programmable logic. On the other hand, the architecture of a FPGA is usually more flexible than that of a CPLD and is better in register-heavy and pipeline applications.

In our work, we developed a new architecture for CPLDs, where the logic equations that describe the circuits are implemented indirectly. The advantages of CPLD over FPGA which are mentioned in the literature are applicable (constant delay, faster for minimized equations) [2]. Comparatively to the actual architecture of CPLDs, new advantages can be found that rise the performances in specific applications. The most important advantages are:

- at macrocell level, the programmable switches are eliminated, and consequently there is no need to reprogram the circuit when changing the circuit. Remember that reprogramming a CPLD requires time ranging from hundreds of milliseconds to tens of seconds;
- at macrocell level, the hardware architecture does not change but rather the information stored in context registers;
- at macrocell level, the speed of changing an implemented circuit reaches its highest possible limit, precisely a single clock pulse;

At the same time, the following disadvantages were noticed:

- the number of elementary gates (AND, OR, XNOR) needed to materialize a product term of any logic equation is greater. Subsenquently for the same logic capacity, the total number of transistors found in a macrocell is superior compared to a macrocell which implements the product terms in classic CPLDs;
- during initialization, unused registers (or fields) located in all macrocells must also be initialized (for example, when implementing the first solution, the mask

words must be loaded with $(00)_H$ and the product terms with $(FF)_H$);

- it is necessary to develop a propriatary software environment for generating the context words values. In our work, the software environment is simplistic and requires the equations to follow specific rules. Other functions could be added such as minimization equations algorithms;

It goes without saying that in any classical computer, a processor executes successively binary instructions at each clock impulse. During this sequential execution, for any particular binary instruction, the processor takes a specific hardware configuration which implements one or several logic equations. Future work could look at replacing a simple processor executing a set of different binary instructions implemented explicitly in a circuit based on the product terms method. Linking these macrocells in a CPLD implementing the logic equations would correspond to the binary instructions. In this way, the processor executing the program can be eliminated and the actual program execution could be optimal.

Bibliography

[1] Alford C., Programmable logic Designers Guide, Howard W. Sams & Company, 1989.

[2] Altera Corporation, CPLDs vs FPGAs Comparing High Capacity Programmable Logic, 1995.

[3] Altera Corporation, FLEX 10K Embedded Programmable Logic Device Family, 2003.

[4] Barr M., Programmable Logic: What's it to Ya?, Embedded Systems Programming, June 1999.

[5] Bhasker J., A VHDL Primer, Prentice-Hall, 1995.

[6] Bolton M., Digital Systems Design with Programmable Logic, Addison-Wesley Publishing Company, 1990.

[7] Booth T. L., Sequential Machines and Automata Theory, John Wiley and Sons, Inc., 1967.

[8] Brown S. and Rose J., Architecture of FPGAs and CPLDs: A Tutorial, Dep. of Electrical and Computer Engineering University of Toronto, 1999.

[9] Brown S., and Vranesic Z., Fundamentals of Digital Logic with VHDL Design, McGraw Hill, New-York, 2000.

[10] Compton K. and Hauck S., Reconfigurable Computing: A Survey of Systems and Software, ACM Computing Survey, Vol 34, pp 171-210, June 2002.

[11] Dancea I., A Software method for Implementation of Digital Circuits in Microcomputer Systems, Proceedings of the ISMM International Symposium, Beverly Hills, California, February 5-7, 1987 pp 145-148.

[12] Dancea I., Dynamically Changing the Logical Behavior of a Microcomputer Interface, IEEE Micro, April 1989, pp. 39- 51.

[13] Dancea I., and Marchand P., Architecture des Ordinateurs, (in French) Gaetan Morin Edition, Montreal, 1993.

[14] Dancea I., Groza V., Dancea L., A Family of VLSI Circuits for Run-time Reconfigurable Computing Periodica Politehnica, Transaction on Automatic Control and Computer Science Vol.49(63), 2004.

[15] DeHon A., PhD Thesis, Reconfigurable Architectures for General-Purpose Computing. AI Technical Report 1586, MIT Artificial Intelligence Laboratory, September 1996.

[16] DeHon A. and J. Wawrzynek. "Reconfigurable Computing: What, Why, and Design Automation Requirements?" Proceedings of the 1999 Design Automation Conference (DAC 1999, June 21-25, 1999).

[17] DeHon A., Very Large Scale Spatial Computing, 3rd International Conference on Unconventional Models of Computation (UMC 2002, October 15-19, 2002).

[18] Gupta B. et al., Adding Reconfigurable Logic to SOC Design, IEEE Design & Test, July-August 2001, pp 65-71.

[19] Hamblen J. O. and Furman M. D., Rapid Prototyping of Digital Systems, Kluwer Academic Publisher, 2003.

[20] Liberty J., Programming C#, O'Reilly, 2002.

[21] Mano M. M., Digital Design, Prentice-Hall, 1991.

[22] Mano M. M. and Kime C. R., Logic and Computer Design Fundamentals, Prentice-Hall, 1997.

[23] McCluskey E. J., Logic Design Principles, Prentice-Hall 1986.

[24] Perry D. L., VHDL: Programming by Example, McGraw-Hill, 2002.

[25] Skahill K., VHDL for Programmable Logic, Addison-Wesley, New-York, 1996.

[26] Van den Bout D., The Practical Xilinx Designer Lab Book, Version 1.5, Prentice Hall Edition, 1999.

[27] Wakerly J. F., Digital Design Principles and Practices, Prentice-Hall 2000.

[28] Watson K., Beginning Visual C# for .NET v1.0, Wrox Press Ltd., 2002.

[29] Yalamanchili S., VHDL Starter's Guide, Prentice-Hall, 1998.

[30] Zilinc Z. and Vranesic Z. G., Using BDDs to Design ULMs for FPGAs, Proceedings of the International Symposium of Field Programmable Gates Arrays, pp 24-30, February 1996.

Appendix A

VHDL Code

A.1 CIRC_COUNTER.VHD

```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

-- circ_counter.vhd is the top-level program in the design

-- the PORT in the ENTITY allows to connect the top-level program
-- with the FLEX10K outside world;

-- the input "data" is not used;
```

```
-- two 7 segment display devices to output the information
-- MSD for the most significant digit (7 segment display)
-- LSD for the least significant digit (7 segment display)

-- pb1, pb2 command pushbuttons
-- pb1 change the target circuit
-- pb2 initialize the active target circuit

ENTITY circ_counter IS
PORT ( cl_25MHz, pb1, pb2 : IN STD_LOGIC;
data : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
MSD_SEG_A : OUT STD_LOGIC;
MSD_SEG_B : OUT STD_LOGIC;
            MSD_SEG_C : OUT STD_LOGIC;
           MSD_SEG_D : OUT STD_LOGIC;
            MSD_SEG_E : OUT STD_LOGIC;
            MSD_SEG_F : OUT STD_LOGIC;
           MSD_SEG_G : OUT STD_LOGIC;
MSD_DP     : OUT STD_LOGIC;
            LSD_SEG_A : OUT STD_LOGIC;
LSD_SEG_B : OUT STD_LOGIC;
            LSD_SEG_C : OUT STD_LOGIC;
```

```
            LSD_SEG_D : OUT STD_LOGIC;
            LSD_SEG_E : OUT STD_LOGIC;
            LSD_SEG_F : OUT STD_LOGIC;
            LSD_SEG_G : OUT STD_LOGIC;
LSD_DP    : OUT STD_LOGIC);
END circ_counter;

ARCHITECTURE structural OF circ_counter IS

-- pb1_single_pulse is a single pulse signal (1Mhz)
-- used by the pb1 pushbutton
-- when the command of changing to a new target circuit is done

SIGNAL  cl_1Hz, cl_10Hz, cl_100Hz,
cl_1MHz, pb1_debounced,
pb1_single_pulse: STD_LOGIC;
SIGNAL MSD: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL LSD: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL LSDP, MSDP: STD_LOGIC;

-- Clk_Div.vhd (from UP2 core Library Functions) provides
-- clock signals, precisely:
-- clock (1Hz) controls the changing of the active state
```

```
-- of the implemented target circuit,

-- clock3 (1Mhz) controls the registers loading when the command

-- of changing to a new target circuit is done


COMPONENT clk_div

PORT (clock_25MHz : IN STD_LOGIC;

  clock_1MHz : OUT STD_LOGIC;

      clock_100KHz : OUT STD_LOGIC;

      clock_10KHz : OUT STD_LOGIC;

      clock_1KHz : OUT STD_LOGIC;

      clock_100Hz : OUT STD_LOGIC;

      clock_10Hz : OUT STD_LOGIC;

      clock_1Hz : OUT STD_LOGIC);

END COMPONENT;


-- debounce.vhd (from UP2 core Library Functions) is a

-- pushbutton debounce program used to filter mechanical

-- bounce on the pb1 pushbutton


COMPONENT debounce

PORT( pb, clock_100Hz  : IN STD_LOGIC;

      pb_debounced : OUT STD_LOGIC);

END COMPONENT;
```

```
-- onepulse.vhd (from UP2 core Library Functions) is
-- a single pulse circuit program
-- needed by the debounced.vhd pushbutton program

COMPONENT onepulse
PORT( pb_debounced, clock: IN STD_LOGIC;
  pb_single_pulse : OUT STD_LOGIC);
END COMPONENT;

-- counter.vhd is the program (hierarchical structure)
-- developed to demonstrate the product terms method,
-- in the particular case of an universal
-- eight bits sequential target circuit
-- with clock input only

COMPONENT counter
PORT(
        data : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
clock, reset, p1, clock1, clock3 : IN STD_LOGIC;
q0, q1, q2, q3, q4, q5, q6, q7, d1, d2 : OUT STD_LOGIC);
END COMPONENT;
```

```
-- hex_7seg.vhd is a program for a hexadecimal to
--  seven-segments decoder with active low outputs
-- (different from that existing in the UP2 core library);
-- the decoder uses directly the logical equations of the decoder


COMPONENT hex_7seg
PORT( x, y, z, w, dp : IN STD_LOGIC;
a1, b1, c1, d1, e1, f1, g1, dp1 : OUT STD_LOGIC);
END COMPONENT;


BEGIN


divider: clk_div PORT MAP ( clock_25MHz => cl_25MHz,
clock_1MHz => cl_1MHz,
clock_100Hz => cl_100Hz,
                                clock_10Hz => cl_10Hz,
clock_1Hz => cl_1Hz);


debounce1: debounce PORT MAP ( pb => pb1,
clock_100Hz => cl_100Hz,
pb_debounced => pb1_debounced);


single_pulse: onepulse PORT MAP ( pb_debounced => pb1_debounced,
```

```
clock => cl_1MHz,

pb_single_pulse => pb1_single_pulse);


counter1: counter PORT MAP (  data => data,

clock => cl_1Hz,  reset => pb2,

p1 => pb1_single_pulse,

clock1 => cl_10Hz,

clock3 => cl_1MHz,

                                    q0 => LSD(0),

q1 => LSD(1),

q2 => LSD(2),

q3 => LSD(3),

q4 => MSD(0),

q5 => MSD(1),

q6 => MSD(2),

q7 => MSD(3),

d1 => LSDP,

d2 => MSDP);


-- decoder1 is a 7 segment decoder for the least

-- significant hexadecimal digit


decoder1 : hex_7seg PORT MAP ( w => LSD(0), z => LSD(1),
```

```
y => LSD(2), x => LSD(3), dp => LSDP,

a1 => LSD_SEG_A,

b1 => LSD_SEG_B,

c1 => LSD_SEG_C,

d1 => LSD_SEG_D,

e1 => LSD_SEG_E,

f1 => LSD_SEG_F,

g1 => LSD_SEG_G,

dp1 => LSD_DP);


-- decoder2 is a 7 segment decoder for the most significant

-- hexadecimal digit


decoder2 : hex_7seg PORT MAP ( w => MSD(0), z => MSD(1),

y => MSD(2), x => MSD(3), dp =>MSDP,

a1 => MSD_SEG_A,

b1 => MSD_SEG_B,

c1 => MSD_SEG_C,

d1 => MSD_SEG_D,

e1 => MSD_SEG_E,

f1 => MSD_SEG_F,

g1 => MSD_SEG_G,

dp1 => MSD_DP);
```

```
  END structural;
```

A.2 HEX_7SEG.VHD

```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.all;

ENTITY hex_7seg IS
PORT ( x, y, z, w, dp :IN STD_LOGIC;
a1, b1, c1, d1, e1, f1, g1, dp1: OUT STD_LOGIC);
END hex_7seg;

-- hex_7seg.vhd is a hexadecimal to 7 segments decoder
-- program which uses directly the equations
-- of segments (a, b, c, d, e, f, g)

ARCHITECTURE dec OF hex_7seg IS
SIGNAL a, b, c, d, e, f, g: STD_LOGIC;
BEGIN

a <= ( x AND NOT y AND NOT z ) OR
   ( NOT x AND z AND w ) OR
   ( NOT x AND y AND NOT z AND w ) OR
```

```
    ( x AND y AND NOT w ) OR
    ( x AND NOT y AND NOT w ) OR
    ( NOT x AND NOT y AND NOT w ) OR
    ( NOT x AND z AND NOT w ) OR
    ( y AND z );

b <= ( x AND y AND NOT z AND w ) OR
  ( NOT x AND z AND w ) OR
  ( NOT x AND NOT z AND NOT w ) OR
  ( NOT y AND NOT z ) OR
  ( x AND NOT y AND NOT w ) OR
  ( NOT x AND NOT y AND NOT w );

c <= ( x AND y AND NOT z AND w ) OR
  ( NOT x AND z AND w ) OR
  ( NOT x AND y AND NOT z AND w ) OR
  ( NOT x AND y AND NOT w ) OR
  ( NOT x AND NOT z AND NOT w ) OR
  ( NOT y AND NOT z ) OR
  ( x AND NOT y AND NOT w ) OR
  ( NOT y AND z AND w );

d <= ( x AND NOT y AND NOT z ) OR
```

```
 ( x AND y AND NOT z AND w ) OR
 ( NOT x AND y AND NOT z AND w ) OR
 ( x AND y AND NOT w ) OR
 ( NOT x AND NOT y AND NOT w ) OR
 ( NOT x AND z AND NOT w ) OR
 ( NOT y AND z AND w );

e <= ( x AND y AND NOT z AND w ) OR
 ( x AND y AND NOT w ) OR
 ( x AND NOT y AND NOT w ) OR
 ( NOT x AND NOT y AND NOT w ) OR
 ( NOT x AND z AND NOT w ) OR
 ( x AND z );

f <= ( x AND NOT y AND NOT z ) OR
 ( NOT x AND y AND NOT z AND w ) OR
 ( NOT x AND y AND NOT w ) OR
 ( NOT x AND NOT z AND NOT w ) OR
 ( x AND y AND NOT w ) OR
 ( x AND NOT y AND NOT w ) OR
 ( x AND z );

g <= ( x AND NOT y AND NOT z ) OR
```

```
 ( x AND y AND NOT z AND w ) OR

 ( NOT x AND y AND NOT z AND w ) OR

 ( NOT x AND y AND NOT w ) OR

 ( x AND NOT y AND NOT w ) OR

 ( NOT x AND z AND NOT w ) OR

 ( NOT y AND z AND w ) OR

 ( x AND z );



       -- LED driver circuits are inverted


a1  <= NOT a;

b1  <= NOT b;

c1  <= NOT c;

d1  <= NOT d;

e1  <= NOT e;

f1  <= NOT f;

g1  <= NOT g;

dp1 <= NOT dp;


END dec;
```

A.3 COUNTER.VHD

```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

-- counter.vhd has a hierarchical structure

-- counter.vhd is developed to demonstrate
-- the product terms method in the case
-- of implementing an eight bits sequential
-- target circuit with clock input only;
-- it connects the block of feedback equations and
-- the the storage register of sequential target circuit

ENTITY counter IS
PORT ( data : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
          clock, reset, p1, clock1, clock3: IN STD_LOGIC;
  q0, q1, q2, q3, q4, q5, q6, q7, d1, d2 : OUT STD_LOGIC);
  END counter;

ARCHITECTURE behaviour OF counter IS
```

```
-- count_in and count_out are the inputs and the outputs
-- of the register used to implement the storege part
-- of the sequential target circuit;
-- CINPUT and reac correspond to the inputs and to the outputs
-- of the combinational part that implements the
-- next state equations of the sequential target circuit
-- using the product terms method;
-- n is a three bit parameter that choses
-- the type of the sequential target circuit from n posibilities

SIGNAL  count_in, count_out, a:  STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  CINPUT, reac:  STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL n: STD_LOGIC_VECTOR (2 DOWNTO 0);

-- circ0.vhd is the program that generates
-- the output vector (My_OUTPUT) of feedback logical equations
-- (8 inputs and 8 outputs) using the product terms method;

COMPONENT circ0
PORt( data : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
  INPUT : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
      My_OUTP: OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
```

```
  n: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
  p1: IN STD_LOGIC;
  clock3: IN  STD_LOGIC);
END COMPONENT;


-- reg_count.vhd is a program that implements
-- an eight bit register; it is a classical synchronous
-- D flip-flops register, having D as inputs, Q as outputs,
-- ck the 1Hz clock input and rt the reset input


COMPONENT reg_count
    PORT (  D       : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                rinit       : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
ck, rt : IN STD_LOGIC;
                Q       : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END COMPONENT;

BEGIN


univ_counter: reg_count PORT MAP (
```

```
D => count_in, rinit => a,

ck => clock, rt => reset,

Q => count_out);


equations: circ0 PORT MAP (data, CINPUT, reac, n, p1, clock3);


-- the following process chooses, by the value of parameter n,

-- one of the eight possible target sequential circuits;


-- the "LEDs period" of the seven segment dysplays,

-- precisely d1(right) and d2(left),

-- visually indicates which is the active sequential target circuit;

-- d1 = 0 means d1 off, d1 = 1 means d1 on,

-- d1 = clock means d1 blinking at low speed

-- and d1 = clock1 means d1 blinking at high speed


-- n = "000" the target circuit is an eight bit direct binary counter

-- n = "001" the target circuit is an eight bit inverse binary counter

-- n = "010" the target circuit is a two decade direct bcd counter

-- n = "011" the target circuit is a two decade inverse bcd counter

-- n = "100" the target circuit is a modulo 12 direct counter

-- n = "101" target circuit is an excess three direct counter

-- n = "110" target circuit is an eight bit pseudo-random generator
```

```
-- n = "111" target circuit is an eight bit special controller


PROCESS (p1, n)

  BEGIN

   IF (p1'EVENT AND p1 = '1') THEN

              n <= n + 1;

          END IF;

IF n = "000" THEN

d1 <= '0';

d2 <= '0';

                    a <= "00000000";

               ELSIF n = "001" THEN

d1 <= '1';

d2 <= '1';

                    a <= "00000000";

ELSIF n = "010" THEN

d1 <= '1';

d2 <= '0';

                    a <= "00000000";

ELSIF n = "011" THEN

d1 <= '0';

d2 <= '1';

                    a <= "00000000";
```

```
                ELSIF n = "100" THEN
d1 <= clock;
d2 <= clock;
                    a <= "00000000";
                ELSIF n = "101" THEN
d1 <= clock1;
d2 <= clock1;
                    a <= "00110011";
                ELSIF n = "110" THEN
d1 <= clock1;
d2 <= '0';
                    a <= "11111111";
                ELSIF n = "111" THEN
d1 <= '0';
d2 <= clock1;
                    a <= "00000000";
                END IF;
END PROCESS;


-- assembling the eight bits register and the block
-- that implements the next state logical equations
-- of the sequential target circuit using the product terms method
```

```
-- is done by connecting the block outputs "reac" to the register

-- inputs "count_in"

-- and the register outputs "count_out" to the block inputs "CINPUT"


count_in <= reac;

CINPUT <= count_out;


-- the register outputs count_out are also connected to the inputs

-- of the seven segment display devices


q0 <= count_out(0);

q1 <= count_out(1);

q2 <= count_out(2);

q3 <= count_out(3);

q4 <= count_out(4);

q5 <= count_out(5);

q6 <= count_out(6);

q7 <= count_out(7);


END behaviour;
```

A.4 REG_COUNT.VHD

```
LIBRARY IEEE;
```

```
USE IEEE.STD_LOGIC_1164.ALL;


-- reg_count.vhd is a classical 8 bits
-- register program; it works at 1Hz clock its
-- inputs D are connected to the outputs and its
-- outputs Q are connected to the inputs of the block
-- that implements the feedback equations of the target
-- sequential circuit using the product terms method


ENTITY reg_count IS
  PORT ( D      : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
         rinit  : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
 ck, rt: IN STD_LOGIC;
         Q      : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END reg_count;


ARCHITECTURE Behaviour OF reg_count IS
BEGIN
PROCESS (ck, rt, rinit)
BEGIN
IF (rt = '0') THEN
           Q <= rinit;
ELSIF (ck'EVENT AND ck = '1') THEN
```

```
                Q <= D;

END IF;

END PROCESS;

END Behaviour;
```

A.5 CIRC0.VHD

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;

USE ieee.STD_LOGIC_ARITH.all;

USE ieee.STD_LOGIC_UNSIGNED.all;

LIBRARY lpm;

USE lpm.lpm_components.all;



-- circ0.vhd has a hierarchical structure and

-- it contains two parts:

-- (1) implements the 8 EAB blocks (Embeded Array Blocks)

-- for storing the data that define the sequential circuits,

-- (2) implements a controller program to load specific

-- register arrays with the corresponding

-- data stored in memory, when a command of

-- changing the sequential target circuit is done;
```

```
-- INPUT is the input vector (8 bits) of the feedback
-- logical equations My_OUTPUT is the output vector (8 bits)
-- of the feedback logical equations


ENTITY circ0 IS
PORT(
    data : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    INPUT: IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    My_OUTP: OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
n: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
p1: IN STD_LOGIC;
clock3: IN  STD_LOGIC);
END circ0;


ARCHITECTURE k0 OF circ0 IS

-- the memory blocks mem0 to mem7 are implementated by 8 EABs;
-- each EAB is configured as 256 words of 8 bits;

-- R0 to R7 are register arrays (each register array
```

```
-- has 16 registers); a register array stores the 16 words
-- (every word has 8 bits) that characterize the eight
-- product terms defining a logical equation,
-- precisely eight mask words, MASK0 to MASK7,
-- and eight product words, PRODUCT0 to PRODUCT7;

-- each EAB works with a register array;
-- RA are address registers (array of 8 registers);
-- RD are output data registers (array of 8 registers);
-- one RA register and one RD register are attached
-- to each EAB to allow the memory read operation;

-- a target sequential circuit is defined by 8 feedback
-- equations; an EAB stores at 16 succesif addresses the
-- information characterising one feedback equation;
-- the data defining the 8 feedback equations are distributed
-- in the 8 EABs having the same address for the words
-- characterizing the 8 different equations which must
-- be loaded in the same position; (for instance in each equation
-- the first register
-- - corresponding to MASK0 register -
-- is placed in the first position);
-- the transfer of data from the 8 EABs to registers placed
```

```
-- in the same position in register arrays is done in parallel;


-- pointer points to the beginning of the blocks of data
-- (8 blocks that work in parallel) characterizing
-- the feedback equations which define a target circuit;
-- contr_counter control the addreses inside of blocks
-- of data characterizing the feedback equations
-- which define a target circuit;


-- state defines the five state of the controller
-- used to control the register loading from memory;
-- these states are: init_reg_address, test_contr_counter,
-- calculate_address, write_register, end_load



TYPE R IS ARRAY(15 DOWNTO 0) OF STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  R0, R1, R2, R3, R4, R5, R6, R7: R;
TYPE P IS ARRAY(7 DOWNTO 0) OF STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL RA, RD: P;
TYPE STATE_TYPE IS (init_reg_address, test_contr_counter,
calculate_address, write_register, end_load);
SIGNAL state: STATE_TYPE;
SIGNAL contr_counter, pointer: STD_LOGIC_VECTOR (7 DOWNTO 0);
```

```
SIGNAL we1:  STD_LOGIC;


COMPONENT func
PORT (  MASK0     : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT0  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
            MASK1     : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT1  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
            MASK2     : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT2  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
            MASK3     : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT3  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
            MASK4     : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT4  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
            MASK5     : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT5  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
            MASK6     : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT6  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
            MASK7     : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT7  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
INP       : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
RES       : OUT STD_LOGIC);
END COMPONENT;
```

```
BEGIN

-- creation of 8 actifs 256 words memories (each word of 8 bits),
-- mem0 to mem7 by calling the lpm_ram_dq function;

-- mem0 stocks the information concerning the first feedback equation
-- of all target circuits;
-- mem1 stocks the information for the second feedback equation,
-- of all target circuits, and so on;

-- at initialization, each memory is charged with a .mif file;
-- for a feedback equation of any target circuit, a memory
-- reserves 16 words, 8 for storing the mask words and 8 for
-- storing the product words; consequentlly, each implemented
-- logical equation must have allways 8 product terms;

-- a real feedback equation of a target circuit can have
-- less than 8 real product terms; in equations that have less
-- than 8 real product terms, the not used, but
-- hardware implemented product terms (for example if an equation
-- has 5 real product terms, the number of not used but hardware
-- implemented product terms are 8 minus 5, so 3)
-- must be considered in the .mif file, where the mask words
```

```
-- are charged with 00h and the product words are charged with ffh


mem0: lpm_ram_dq


GENERIC MAP  (lpm_widthad => 8,

lpm_outdata => "REGISTERED",

lpm_indata => "REGISTERED",

lpm_address_control => "UNREGISTERED",

lpm_file => "func0.mif",

lpm_width => 8)


PORT MAP  (data => data, address => RA(0), we => we1,

                inclock => clock3, outclock => clock3, q => RD(0));


mem1: lpm_ram_dq


GENERIC MAP  (lpm_widthad => 8,

lpm_outdata => "REGISTERED",

lpm_indata => "REGISTERED",

lpm_address_control => "UNREGISTERED",

lpm_file => "func1.mif",

lpm_width => 8)
```

```
PORT MAP  (data => data, address => RA(1), we => we1,
                inclock => clock3, outclock => clock3, q => RD(1));


mem2: lpm_ram_dq


GENERIC MAP  (lpm_widthad => 8,
lpm_outdata => "REGISTERED",
lpm_indata => "REGISTERED",
lpm_address_control => "UNREGISTERED",
lpm_file => "func2.mif",
lpm_width => 8)


PORT MAP  (data => data, address => RA(2), we => we1,
                inclock => clock3, outclock => clock3, q => RD(2));


mem3: lpm_ram_dq


GENERIC MAP  (lpm_widthad => 8,
lpm_outdata => "REGISTERED",
lpm_indata => "REGISTERED",
lpm_address_control => "UNREGISTERED",
lpm_file => "func3.mif",
lpm_width => 8)
```

```
PORT MAP  (data => data, address => RA(3), we => we1,
                inclock => clock3, outclock => clock3, q => RD(3));

mem4: lpm_ram_dq

GENERIC MAP  (lpm_widthad => 8,
lpm_outdata => "REGISTERED",
lpm_indata => "REGISTERED",
lpm_address_control => "UNREGISTERED",
lpm_file => "func4.mif",
lpm_width => 8)

PORT MAP  (data => data, address => RA(4), we => we1,
                inclock => clock3, outclock => clock3, q => RD(4));

mem5: lpm_ram_dq

GENERIC MAP  (lpm_widthad => 8,
lpm_outdata => "REGISTERED",
lpm_indata => "REGISTERED",
lpm_address_control => "UNREGISTERED",
lpm_file => "func5.mif",
```

```
lpm_width => 8)

PORT MAP  (data => data, address => RA(5), we => we1,
                inclock => clock3, outclock => clock3, q => RD(5));

mem6: lpm_ram_dq

GENERIC MAP  (lpm_widthad => 8,
lpm_outdata => "REGISTERED",
lpm_indata => "REGISTERED",
lpm_address_control => "UNREGISTERED",
lpm_file => "func6.mif",
lpm_width => 8)

PORT MAP  (data => data, address => RA(6), we => we1,
                inclock => clock3, outclock => clock3, q => RD(6));

mem7: lpm_ram_dq

GENERIC MAP  (lpm_widthad => 8,
lpm_outdata => "REGISTERED",
lpm_indata => "REGISTERED",
lpm_address_control => "UNREGISTERED",
```

```
lpm_file => "func7.mif",
lpm_width => 8)

PORT MAP  (data => data, address => RA(7), we => we1,
                inclock => clock3, outclock => clock3, q => RD(7));

-- circ0.vhd uses in its lower level, functions0, function1,
-- programs that implement the product terms method for
-- generating the outputs -- of feedback logical equations (func.vhd);


function0: func PORT MAP (MASK0 => R0(0), PRODUCT0 => R0(1),
                          MASK1 => R0(2), PRODUCT1 => R0(3),
                          MASK2 => R0(4), PRODUCT2 => R0(5),
                          MASK3 => R0(6), PRODUCT3 => R0(7),
                          MASK4 => R0(8), PRODUCT4 => R0(9),
                          MASK5 => R0(10), PRODUCT5 => R0(11),
                          MASK6 => R0(12), PRODUCT6 => R0(13),
                          MASK7 => R0(14), PRODUCT7 => R0(15),
  INP => INPUT, RES => My_OUTP(0));

function1: func PORT MAP (MASK0 => R1(0), PRODUCT0 => R1(1),
                          MASK1 => R1(2), PRODUCT1 => R1(3),
```

```
                          MASK2 => R1(4), PRODUCT2 => R1(5),
                          MASK3 => R1(6), PRODUCT3 => R1(7),
                          MASK4 => R1(8), PRODUCT4 => R1(9),
                          MASK5 => R1(10), PRODUCT5 => R1(11),
                          MASK6 => R1(12), PRODUCT6 => R1(13),
                          MASK7 => R1(14), PRODUCT7 => R1(15),
  INP => INPUT, RES => My_OUTP(1));

function2: func PORT MAP (MASK0 => R2(0), PRODUCT0 => R2(1),
                          MASK1 => R2(2), PRODUCT1 => R2(3),
                          MASK2 => R2(4), PRODUCT2 => R2(5),
                          MASK3 => R2(6), PRODUCT3 => R2(7),
                          MASK4 => R2(8), PRODUCT4 => R2(9),
                          MASK5 => R2(10), PRODUCT5 => R2(11),
                          MASK6 => R2(12), PRODUCT6 => R2(13),
                          MASK7 => R2(14), PRODUCT7 => R2(15),
  INP => INPUT, RES => My_OUTP(2));

function3: func PORT MAP (MASK0 => R3(0), PRODUCT0 => R3(1),
                          MASK1 => R3(2), PRODUCT1 => R3(3),
                          MASK2 => R3(4), PRODUCT2 => R3(5),
                          MASK3 => R3(6), PRODUCT3 => R3(7),
                          MASK4 => R3(8), PRODUCT4 => R3(9),
```

```
                        MASK5 => R3(10), PRODUCT5 => R3(11),
                        MASK6 => R3(12), PRODUCT6 => R3(13),
                        MASK7 => R3(14), PRODUCT7 => R3(15),
  INP => INPUT, RES => My_OUTP(3));

function4: func PORT MAP (MASK0 => R4(0), PRODUCT0 => R4(1),
                        MASK1 => R4(2), PRODUCT1 => R4(3),
                        MASK2 => R4(4), PRODUCT2 => R4(5),
                        MASK3 => R4(6), PRODUCT3 => R4(7),
                        MASK4 => R4(8), PRODUCT4 => R4(9),
                        MASK5 => R4(10), PRODUCT5 => R4(11),
                        MASK6 => R4(12), PRODUCT6 => R4(13),
                        MASK7 => R4(14), PRODUCT7 => R4(15),
  INP => INPUT, RES => My_OUTP(4));

function5: func PORT MAP (MASK0 => R5(0), PRODUCT0 => R5(1),
                        MASK1 => R5(2), PRODUCT1 => R5(3),
                        MASK2 => R5(4), PRODUCT2 => R5(5),
                        MASK3 => R5(6), PRODUCT3 => R5(7),
                        MASK4 => R5(8), PRODUCT4 => R5(9),
                        MASK5 => R5(10), PRODUCT5 => R5(11),
                        MASK6 => R5(12), PRODUCT6 => R5(13),
                        MASK7 => R5(14), PRODUCT7 => R5(15),
```

```
  INP => INPUT, RES => My_OUTP(5));


function6: func PORT MAP (MASK0 => R6(0), PRODUCT0 => R6(1),
                          MASK1 => R6(2), PRODUCT1 => R6(3),
                          MASK2 => R6(4), PRODUCT2 => R6(5),
                          MASK3 => R6(6), PRODUCT3 => R6(7),
                          MASK4 => R6(8), PRODUCT4 => R6(9),
                          MASK5 => R6(10), PRODUCT5 => R6(11),
                          MASK6 => R6(12), PRODUCT6 => R6(13),
                          MASK7 => R6(14), PRODUCT7 => R6(15),
  INP => INPUT, RES => My_OUTP(6));


function7: func PORT MAP (MASK0 => R7(0), PRODUCT0 => R7(1),
                          MASK1 => R7(2), PRODUCT1 => R7(3),
                          MASK2 => R7(4), PRODUCT2 => R7(5),
                          MASK3 => R7(6), PRODUCT3 => R7(7),
                          MASK4 => R7(8), PRODUCT4 => R7(9),
                          MASK5 => R7(10), PRODUCT5 => R7(11),
                          MASK6 => R7(12), PRODUCT6 => R7(13),
                          MASK7 => R7(14), PRODUCT7 => R7(15),
  INP => INPUT, RES => My_OUTP(7));


-- the controller program loads the registers with data from
```

```
-- EAB memories when the command of changing the sequential
-- circuit is done;

-- this controller is the most difficult part of the programmation
-- task, the difficulty is caused by synchronizations problems
-- durring the passing from one state to other in sequential circuits

-- the controller program part uses five diffrent states

-- the state init_reg_address which initializes the address registers
-- and the pointer considering the parameter n;

-- the state test_contr_counter which test the address controller
-- to find the end of loading;

-- the state calculate_address which calculate the memory addresses
-- of the data to be loaded in parallel to registers;

-- the state write_register which effectively transfers data
-- (8 data in parallel) from memories to the corresponding registers;

-- the state end_load which indicate the end of register's loading
```

```
-- normally a transfer of a word, from memory to the
-- corresponding register, needs three clocks for passing
-- by the states test_contr_counter, calculate_address and
-- write_register; consequently, the total time needed for changing
-- from one target circuit to other circuit is 48 clocks
-- the states init_reg_address and end_load
-- are used only once for each changing command of the circuit


PROCESS (p1, clock3)
   BEGIN
IF (clock3'EVENT AND clock3 = '0') THEN
           -- the clock is tested to '0' to produce a delay
  IF (p1 = '1') THEN
         state <= init_reg_address;
           -- (a semi period) of the memory read operation
  END IF;
       CASE state IS
  WHEN init_reg_address =>
           -- in this way the address became stable at
contr_counter <= "00000000";
  CASE n IS
           -- the reading instant; p1 is tested to '1'
      WHEN "000" =>
```

```
pointer <= "00000000";
            -- because it produces a positive one pulse signal
      WHEN "001" =>
pointer <= "00010000";
      WHEN  "010" =>
pointer <= "00100000";
      WHEN  "011" =>
pointer <= "00110000";
              WHEN  "100" =>
pointer <= "01000000";
              WHEN  "101" =>
pointer <= "01010000";
              WHEN  "110" =>
pointer <= "01100000";
              WHEN  OTHERS =>
pointer <= "01110000";
  END CASE;
   state <= calculate_address;
WHEN test_contr_counter =>
IF (CONV_INTEGER(contr_counter) = 16) THEN
state <= end_load;
ELSE
state <=  calculate_address; END IF;
```

```
WHEN calculate_address =>

            -- the same address is loaded in all address registers

    RA(0) <= contr_counter OR pointer;
                    RA(1) <= contr_counter OR pointer;
                    RA(2) <= contr_counter OR pointer;
                    RA(3) <= contr_counter OR pointer;
                    RA(4) <= contr_counter OR pointer;
                    RA(5) <= contr_counter OR pointer;
                    RA(6) <= contr_counter OR pointer;
                    RA(7) <= contr_counter OR pointer;
state <=  write_register;
WHEN write_register =>
          CASE contr_counter (7 DOWNTO 0) IS

-- the transfer of data from memory registers of the 8 EABs to
-- registers placed in the same position in register arrays is
-- done in parallel

WHEN "00000000" =>
    R0(0) <= RD(0);
                    R1(0) <= RD(1);
```

```
                    R2(0) <= RD(2);

                    R3(0) <= RD(3);

                    R4(0) <= RD(4);

                    R5(0) <= RD(5);

                    R6(0) <= RD(6);

                    R7(0) <= RD(7);

WHEN "00000001" =>

    R0(1) <= RD(0);

                    R1(1) <= RD(1);

                    R2(1) <= RD(2);

                    R3(1) <= RD(3);

                    R4(1) <= RD(4);

                    R5(1) <= RD(5);

                    R6(1) <= RD(6);

                    R7(1) <= RD(7);

WHEN "00000010"=>

        R0(2) <= RD(0);

                    R1(2) <= RD(1);

                    R2(2) <= RD(2);

                    R3(2) <= RD(3);

                    R4(2) <= RD(4);

                    R5(2) <= RD(5);

                    R6(2) <= RD(6);
```

```
                    R7(2) <= RD(7);

        WHEN "00000011" =>

    R0(3) <= RD(0);

                    R1(3) <= RD(1);

                    R2(3) <= RD(2);

                    R3(3) <= RD(3);

                    R4(3) <= RD(4);

                    R5(3) <= RD(5);

                    R6(3) <= RD(6);

                    R7(3) <= RD(7);

WHEN "00000100" =>

    R0(4) <= RD(0);

                    R1(4) <= RD(1);

                    R2(4) <= RD(2);

                    R3(4) <= RD(3);

                    R4(4) <= RD(4);

                    R5(4) <= RD(5);

                    R6(4) <= RD(6);

                    R7(4) <= RD(7);

WHEN "00000101" =>

    R0(5) <= RD(0);

                    R1(5) <= RD(1);

                    R2(5) <= RD(2);
```

```
                    R3(5) <= RD(3);

                    R4(5) <= RD(4);

                    R5(5) <= RD(5);

                    R6(5) <= RD(6);

                    R7(5) <= RD(7);

WHEN "00000110" =>

    R0(6) <= RD(0);

                    R1(6) <= RD(1);

                    R2(6) <= RD(2);

                    R3(6) <= RD(3);

                    R4(6) <= RD(4);

                    R5(6) <= RD(5);

                    R6(6) <= RD(6);

                    R7(6) <= RD(7);

WHEN "00000111" =>

    R0(7) <= RD(0);

                    R1(7) <= RD(1);

                    R2(7) <= RD(2);

                    R3(7) <= RD(3);

                    R4(7) <= RD(4);

                    R5(7) <= RD(5);

                    R6(7) <= RD(6);

                    R7(7) <= RD(7);
```

```
WHEN "00001000" =>

    R0(8) <= RD(0);

                    R1(8) <= RD(1);

                    R2(8) <= RD(2);

                    R3(8) <= RD(3);

                    R4(8) <= RD(4);

                    R5(8) <= RD(5);

                    R6(8) <= RD(6);

                    R7(8) <= RD(7);

WHEN "00001001" =>

    R0(9) <= RD(0);

                    R1(9) <= RD(1);

                    R2(9) <= RD(2);

                    R3(9) <= RD(3);

                    R4(9) <= RD(4);

                    R5(9) <= RD(5);

                    R6(9) <= RD(6);

                    R7(9) <= RD(7);

WHEN "00001010" =>

    R0(10) <= RD(0);

                    R1(10) <= RD(1);

                    R2(10) <= RD(2);

                    R3(10) <= RD(3);
```

```
                    R4(10) <= RD(4);
                    R5(10) <= RD(5);
                    R6(10) <= RD(6);
                    R7(10) <= RD(7);
WHEN "00001011" =>
    R0(11) <= RD(0);
                    R1(11) <= RD(1);
                    R2(11) <= RD(2);
                    R3(11) <= RD(3);
                    R4(11) <= RD(4);
                    R5(11) <= RD(5);
                    R6(11) <= RD(6);
                    R7(11) <= RD(7);
WHEN "00001100" =>
    R0(12) <= RD(0);
                    R1(12) <= RD(1);
                    R2(12) <= RD(2);
                    R3(12) <= RD(3);
                    R4(12) <= RD(4);
                    R5(12) <= RD(5);
                    R6(12) <= RD(6);
                    R7(12) <= RD(7);
WHEN "00001101" =>
```

```
    R0(13) <= RD(0);
                    R1(13) <= RD(1);
                    R2(13) <= RD(2);
                    R3(13) <= RD(3);
                    R4(13) <= RD(4);
                    R5(13) <= RD(5);
                    R6(13) <= RD(6);
                    R7(13) <= RD(7);
WHEN "00001110" =>
    R0(14) <= RD(0);
                    R1(14) <= RD(1);
                    R2(14) <= RD(2);
                    R3(14) <= RD(3);
                    R4(14) <= RD(4);
                    R5(14) <= RD(5);
                    R6(14) <= RD(6);
                    R7(14) <= RD(7);
WHEN OTHERS =>
    R0(15) <= RD(0);
                    R1(15) <= RD(1);
                    R2(15) <= RD(2);
                    R3(15) <= RD(3);
                    R4(15) <= RD(4);
```

```
                    R5(15) <= RD(5);
                    R6(15) <= RD(6);
                    R7(15) <= RD(7);
   END CASE;

-- increment the contr_counter to calculate the next memory address

            contr_counter <= contr_counter + 1;
state <= test_contr_counter;
   WHEN end_load =>
       we1 <= '0';
  END CASE;
END IF;
END PROCESS;

END k0;
```

A.6 FUNC.VHD

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

```
-- func.vhd is the program that implements the
-- combinational circuit that calculate a value of a
-- logical function using the product terms method;
-- MASK0 to MASK7 and PRODUCT0 to PRODUCT7 are inputs
-- for the program; INP represents the active inputs
-- of logical circuit RES represents the output of
-- logical circuit

ENTITY func IS
PORT (
    MASK0     : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT0  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
            MASK1     : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT1  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
            MASK2     : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT2  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
            MASK3     : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT3  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
            MASK4     : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT4  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
            MASK5     : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT5  : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
```

```
            MASK6      : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT6   : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
            MASK7      : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    PRODUCT7   : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    INP        : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
    RES        : OUT STD_LOGIC);
END func;

ARCHITECTURE g OF func IS

    SIGNAL  FF        : STD_LOGIC_VECTOR (7 DOWNTO 0);

-- every logical equation is expresed by sum-of-product form,
-- and can have a maximum of eight product terms
-- to calculate the value of a product term for given inputs
-- the product term method implements the program cell.vhd

COMPONENT cell
PORT  (M      : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
   P      : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
   i      : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
   t      : OUT STD_LOGIC);
END COMPONENT;
```

```
COMPONENT orgate
PORT ( b  : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
   f  : OUT STD_LOGIC);
END COMPONENT;

BEGIN

-- connection of the eight cells with the final OR gate
-- for determining the value of logical function

cell_0: cell PORT MAP ( MASK0, PRODUCT0, INP, FF(0));
cell_1: cell PORT MAP ( MASK1, PRODUCT1, INP, FF(1));
cell_2: cell PORT MAP ( MASK2, PRODUCT2, INP, FF(2));
cell_3: cell PORT MAP ( MASK3, PRODUCT3, INP, FF(3));
cell_4: cell PORT MAP ( MASK4, PRODUCT4, INP, FF(4));
cell_5: cell PORT MAP ( MASK5, PRODUCT5, INP, FF(5));
cell_6: cell PORT MAP ( MASK6, PRODUCT6, INP, FF(6));
cell_7: cell PORT MAP ( MASK7, PRODUCT7, INP, FF(7));
orgate1: orgate PORT MAP (FF, RES);

END g;
```

A.7 CELL.VHD

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;


-- cell.vhd calculate the logical value t of

-- a product term for given inputs (i);

-- every cell needs at input a mask word (M)

-- and a product word (P)


ENTITY cell IS

PORT( M : IN STD_LOGIC_VECTOR (7 DOWNTO 0);

  P : IN STD_LOGIC_VECTOR (7 DOWNTO 0);

  i : IN STD_LOGIC_VECTOR (7 DOWNTO 0);

            t : OUT STD_LOGIC);

END cell;


ARCHITECTURE s OF cell IS



SIGNAL r: STD_LOGIC_VECTOR (7 DOWNTO 0);

SIGNAL v: STD_LOGIC_VECTOR (7 DOWNTO 0);


BEGIN
```

```
-- the combinational part which calculate the value

-- of a product term


-- the AND operations between the inputs and the

-- mask word bits to establish the variables of

-- the product term  - first logical level



r(0) <= i(0) AND M(0);

r(1) <= i(1) AND M(1);

r(2) <= i(2) AND M(2);

r(3) <= i(3) AND M(3);

r(4) <= i(4) AND M(4);

r(5) <= i(5) AND M(5);

r(6) <= i(6) AND M(6);

r(7) <= i(7) AND M(7);


-- the XNOR operations between the precedent result

-- and the product word bits to establish which variables

-- are actives for the considered inputs - second logical level


v(0) <= NOT (r(0) XOR P(0));
```

```
v(1) <= NOT (r(1) XOR P(1));

v(2) <= NOT (r(2) XOR P(2));

v(3) <= NOT (r(3) XOR P(3));

v(4) <= NOT (r(4) XOR P(4));

v(5) <= NOT (r(5) XOR P(5));

v(6) <= NOT (r(6) XOR P(6));

v(7) <= NOT (r(7) XOR P(7));


-- the final AND operation to establish if the considered

-- product term is active for the considered inputs - third

-- logical level



t <= v(0) AND v(1) AND v(2) AND v(3) AND v(4) AND v(5)

AND v(6) AND v(7);


END s;
```

A.8 ORGATE.VHD

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;
```

A.8. ORGATE.VHD

```
-- orgate.vhd is a program that calculate the value

--  of a logical equation for given inputs, by OR-ing

-- the outputs of the eight product terms


ENTITY orgate IS

PORT( b   : IN STD_LOGIC_VECTOR (7 DOWNTO 0);

          f   : OUT STD_LOGIC);

END orgate;


ARCHITECTURE a OF orgate IS


BEGIN


f <= b(0) OR b(1) OR b(2) OR b(3) OR b(4) OR b(5) OR b(6) OR b(7);


END a;
```

Appendix B

C# Code

B.1 CLASS1.CS

```
using System;
using System.IO;

namespace ConsoleApplication3
{
 class Class1
{
 [STAThread]
 static void Main (string[] args)
 {
// underline character for negation of previous input
 const char aa = '_';
```

```
// space character represents the token's separator
 const char bb = ' ';

// strLine[i] is the line i (character's form) from the input file
 string[] strLine = new string [12];

// world1[] is the array of tokens (character's form)
 string[,] word1 = new string[12,20];

// mask[] is the array of mask words - integer values
 int[,] mask = new int[12,20];

// prod[] is the array of product words - integer values
 int[,] prod = new int[12,20];

// world2[] is the array of integers (bytes) loaded in memory
 int[,] word2 = new int[8,16];

// InpVariable[] is the array of input variables(characters)
 char[] InpVariable = new char[8];

// ProductTerm[] is the array of characters of a product term
 char[] ProductTerm = new char[16];
```

```
// weight[] is the array of weights for input variables
 int[] weight = new int[8];

// m[] is the array which indicates for each equation
// the number of product terms - maximum 8
 int[] m = new int[12];

// control integers
 int a, b, c, g, h, i, j, k, l, n, p, r, s, x, y, w ;

 try
 {
// open to read the .txt file containing the information
// that characterizes a target circuit (variables and equations)
// (maximum 8 inputs and maximum 8 outputs)
// in this example the file "BINinverse.txt" contains the information
// that caracterizes an 8 bit binary invers counter
// (8 variables and 8 equations)

// use StreamReader class te read data from a file

 FileStream aFile = new FileStream ("BINinverse.txt", FileMode.Open);
```

```
 StreamReader sr = new StreamReader(aFile);


// j controls the column index of token's array in a line

 j = 0;


// i controls the line index of token's arrays

                i = 0;


// l controls the line index of equations

 l = 0;


// read data in str.Line[i], line by line, from .txt file


 strLine[i] = sr.ReadLine();


// continue to read till the end of the file is found

 while (strLine[i] != null)

 {


// write the line


 Console.WriteLine("\n");

 Console.WriteLine("{0}" , strLine[i]);
```

```
// split every input line in tokens
//- a token is a strig of characters using the Split function;
// myWords[] is an array having all the tokens of a line;
// the separator is space
// (bb cannot be used, because the separator1 must be an array of character)

 char[] separator1 = {' '};
 string[] myWords;
 myWords = strLine[i].Split(separator1);

 foreach (string word in myWords)
 {
 word1[i,j] = word;

// extraction of "useful" tokens that represent product terms
// the tokens that represents product terms begin with the line 3;
// in these lines, the first token that represent a product term
// begins with the third position;
// the tokens "+" and "end" must be eliminated from the "useful" tokens

 if ( i >= 2 & j >= 2 & word1[i,j] != "+" & word1[i,j] != "end" )
 {
```

```
// write the tokens that represent product terms


 Console.WriteLine("word1[{0},{1}] = {2}",

 i -2, j -2, word1[i,j]);


// for any equation, m[l] indicates the # of product terms


 m[l] = m[l] + 1;

 }


// if the token is "+" do not increse the index j


 if (word1[i,j] == "+")

 j = j - 1;


// if a token is "end" initialize the index j

// and increase the line index of equations


    else if (word1[i,j] == "end")

 {

 l = l + 1;

 j = -1;

 }
```

```
 j = j + 1;
 }
 i = i + 1;

//read from the input file the next input line and loop

 strLine[i] = sr.ReadLine();
 }
                Console.WriteLine("\n");

// word1[0,2] is the token of input variables
// split the token of input variables in it's composants (characters)
                k = 0;
 foreach (char character in word1[0,2])
 {
 if (character != bb)
       InpVariable[k] = character;

// k controls the number of input variables
       k = k + 1;
 }

// calculate and write the weight of first input variable
```

```
                weight[0] = 1;
 Console.WriteLine("InpVariable[0] is {0}, weight[0] = {1}",
 InpVariable[0], weight[0]);
 p = 1;
 n = 1;

// use the parameter Length

 while (n < word1[0,2].Length )
 {
 p = p * 2;
 weight[n] = p;

// calculate and write the weight of other input variables

 Console.WriteLine("InpVariable[{0}] is {1}, weight[{0}] = {2}",
 n, InpVariable[n], weight[n]);
 n = n+ 1;
 }

// establish the characters which compose each product term
// and calculate the values of mask words and of product words
```

```
                    Console.WriteLine("\n");
x = 2;
l = 2;

for (c =2; c < i; c++)
{
y = 2;
for (b = 0; b < m[l]; b++)
{
                              r = 0;

// split a product term in characters

foreach (char character in word1[x,y])
{
if (character != bb)
{
ProductTerm[r] = character;
r = r + 1;

// r controls the number of characters (not variables)
// in a product term (among the characters which reprezent the inputs
```

```
//the  negation character '_' can be found)


 }

 }


// initialize the mask word

 mask[x,y] = 0;


// initialize the product word

 prod[x,y] = 0;

 a = 0;

 for (s =0; s < r; s++)

 {

 for (w = 0; w < k; w++)

 {


// calculate the mask and the product words for each product term


 if (ProductTerm[s] == InpVariable[w])

 {


// initialy the value of product word

// is the same as the value of mask word
```

```
mask[x,y] = mask [x,y] + weight[w];

prod[x,y] = prod [x,y] + weight[w];

a = weight[w];

}

else

{

if (ProductTerm[s] == aa)


// if the character is underline, means that the previous character

// was a variable in negation form, and the product term

// must be decreased with the value of correspondet weight

{

prod[x,y] = prod[x,y] - a;


// a is reinitialized after the decrementation


a = 0;

}

}

}

}
```

```
// write the mask and the product words

 Console.WriteLine("mask[{0},{1}] = {2}, prod[{0},{1}] = {3}",
 x - 2, y - 2, mask[x,y], prod[x,y]);
 y = y + 1;
}

// go to the next equation

 l = l + 1;
 x = x + 1;
 }

// calculate the memory words and print them in hexa
// each equation can have a maximum of 8 mask words and a maximum
// of 8 product words at initialization all mask words are loaded
// with 00(hexa) and all product words are loaded with FF(hexa)

 h = 0;
 g = 0;
                for (g = 0; g < 8; g++)
 {
 h = 0;
```

```
 while (h <16)
 {
 word2[g,h] = 0;
 h = h + 1;
 word2[g,h] = 255;
 h = h + 1;
 }
}

// for the real product terms find in each equation, the real mask words and
// the real product words, replace the initialization values

                a = 0;
 l = 2;
 x = 0;
 for (c =2; c < i; c++)
 {
 y = 0;
 for (b = 0; b < m[l]; b++)
 {
 word2[x,y] = mask[x+2,b+2];
 a = prod[x+2,b+2];
 y = y +1;
```

```
word2[x,y] = a;

y = y + 1;

}

x = x + 1;

l = l + 1;

}


// convert and write the memory words in hexadecimal


Console.WriteLine("\n");

for (g = 0; g < 8; g++)

{

Console.WriteLine("\n Memory words for func{0} in hexadecimal: \n" &

"{1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}, {10}, {11}, {12}," &

"{13}, {14}, {15}, {16}", g,

String.Format("{0,0:X}", word2[g,0]),

String.Format("{0,0:X}", word2[g,1]),

String.Format("{0,0:X}", word2[g,2]),

String.Format("{0,0:X}", word2[g,3]),

String.Format("{0,0:X}", word2[g,4]),

String.Format("{0,0:X}", word2[g,5]),

String.Format("{0,0:X}", word2[g,6]),

String.Format("{0,0:X}", word2[g,7]),
```

```
    String.Format("{0,0:X}", word2[g,8]),
String.Format("{0,0:X}", word2[g,9]),
String.Format("{0,0:X}", word2[g,10]),
String.Format("{0,0:X}", word2[g,11]),
String.Format("{0,0:X}", word2[g,12]),
String.Format("{0,0:X}", word2[g,13]),
String.Format("{0,0:X}", word2[g,14]),
String.Format("{0,0:X}", word2[g,15]));
 }


 sr.Close();
 }
 catch (IOException e)
 {
 Console.WriteLine("An IO exception has been thrown !");
 Console.WriteLine(e.ToString());
 return;
 }
 return;
 }
}
}
```