

UNIVERSITY OF ROME TOR VERGATA

DOTTORATO DI RICERCA IN SISTEMI E TECNOLOGIE
PER LO SPAZIO XXII CICLO

**Reconfigurable digital architecture for
high speed Digital Signal Processing**

Relatore

Prof. Gian Carlo Crdarilli

Dottorando

Luca Di Nunzio

Anno Accademico 2009/2010

Alla mia famiglia e a Marta

Ringraziamenti

Desidero ringraziare ed esprimere la mia riconoscenza nei confronti di tutte le persone con le quali ho lavorato in questi magnifici tre anni di Dottorato (Professori, Ricercatori, Dottorandi e Studenti). I miei piú sentiti ringraziamenti vanno al Prof Gian Carlo Cardarilli che mi ha dato la possibilitá di vivere questa fantastica esperienza.

Abstract

Low cost microprocessors and DSPs are optimized to perform arithmetic and logic operations on data having a fixed size, typically 16,32 or 64 bit. On the other hand, their efficiency decreases when data shorter respect than their native wordlength are processed (more clock cycles per operation are required). Recently different solutions have been proposed to overcome this problem. Among those, the ones based on a main processor with a Reconfigurable Unit used as hardware accelerator are the most interesting in terms of performance and flexibility. Typically those architectures are similar to very small FPGA; they consist in arrays of Look-Up Tables (LUTs) interconnected by pass transistors networks.

This work proposes a new Reconfigurable Accelerator called ADAPTO (Adder-based Dynamic Architecture for Processing Tailored Operators). The main different between ADAPTO and the others Reconfigurable Units proposed in literature is the reduced hardware complexity in terms of silicon area. This feature give the possibility to integrate ADAPTO in embedded low cost microprocessors and DSPs (Digital Signal Processors), in fact, for these kind of processors, the area occupation and therefore the cost is a very critical aspect. The ADAPTO Unit supports both hardware reconfiguration and instruction execution in the same processor clock cycle. These goals have been obtained with the multicontext approach using a reconfigurable unit based on full adders, instead LUTs. As discussed in this work this choice allows to the multicontext technique a reduced wasting of hardware resources.

Contents

1	Reconfigurable Computing	10
1.1	Parallel computing vs Serial computing	10
1.2	Reconfigurable systems	11
1.2.1	Field-Programmable Gate Array FPGA	11
1.2.2	Reconfiguration strategies	13
2	State of the art	18
2.1	Reconfigurable Architecture as hardware accelerators	18
2.2	PRISC	20
2.3	The Chimaera Unit	21
2.4	A solution for the hardware wasting	24
3	The ADAPTO Reconfigurable Functional Unit	29
3.1	ADAPTO architecture	29
3.1.1	Logic Block	30
3.1.2	Interconnect	33
3.1.3	Context Memories	36
3.2	The ADAPTO IC design	37
3.2.1	Delay	38
3.2.2	Setup Time and Hold Time	38

3.2.3	Input Capacitances	39
3.2.4	Power Consumption	39
3.2.5	Logic Block	40
3.2.6	Full adder	42
3.2.7	ADAPTO	43
3.3	ADAPTO IC main features	43
3.3.1	Delay time	43
3.3.2	Power consumption	44
3.3.3	Area occupation	44
4	ADAPTO applications	55
4.1	Modular algebra	55
4.1.1	Modular Addition	55
4.1.2	Montgomery Multiplication Algorithm	59
4.2	The AES algorithm	63
4.2.1	ADAPTO implementation	66
4.2.2	$GF(2^8)$ constant multiplication in ADAPTO	66
4.2.3	ADAPTO implementation of Mixcolumn	67
4.2.4	ADAPTO implementation of InvMixcolumn	69
5	ADAPTO in Leon 2 processor	73
5.1	LEON-2 processor	74
5.2	Experimental results	75
5.2.1	Bit reversal permutation	76
5.2.2	MPEG-2 encoding	77
5.2.3	GRP instruction	79
5.2.4	Endian coversion	79

6	Future works	82
6.1	The NIOS II processor	82
6.1.1	Custom Logic Custom Instructions	83
6.1.2	ADAPTO integration	86

Introduction

The use of reconfigurable units for speeding-up algorithm implementation is a very interesting solution for high-performance DSP systems [1]. In fact, the literature shows that even simple algorithms can lead to a significant reduction of the performance also in efficient and expensive processors. Normally, these algorithms are characterized by very special operations, operating at bit level or on very short wordlengths. These operations don't completely exploit the processor resources. In fact in the processors are present very fast computation elements operating on data whose wordlength is characteristic for the selected processor.

The above drawbacks can be solved in different way. A possible solution is based on the introduction of special operators with reduced wordlength that are able to perform specific class of operations (see for example 8-bit addition used in [3]). A more general solution can make use of a Reconfigurable Unit (RU). This unit can be reconfigured in order to execute any required operation working at bit level or with short wordlengths. In general, processing performance of conventional embedded processors and DSP degrades when the implemented algorithm requires a big amount of short data operations; a fixed size datapath is not the most efficient hardware to compute these operations. Applications that require short data manipulation are for example the bit reversal that reverse bits order in a data (usually the index of an array), the bit packing/unpacking operations in which different sub-words coming from different words are concatenated to create a new word.

To solve this problem several solutions have been proposed in the literature, both software [4] and hardware [1]. Among the hardware solutions the most interesting ones in terms of flexibility and performance are the Reconfigurable Functional Units (RFUs). Usually these architectures are similar to small FPGAs (array of LUTs and pass-transistor interconnect) connected in parallel with the ALU in the datapath of the processor.

A RFU is a tightly coupled integrated hardware accelerator used by a standard processor to speed-up the computation of particular operations. An RFU can be considered an hardware Instruction Set Architecture (ISA) expansion. Normal operations are performed by the ALU meanwhile non standard operations are executed by the RFU. More details about the RFUs are shown in Chapter 2. Two techniques can be used to reconfigure the RFU: bitstream reloading and multicontext approach. The first one is slow in terms of reconfiguration time because the bitstream reload requires several clock cycles [1],[2]. On the other hand, the multicontext approach (consisting in N replicas of the basic structure, where N is the number of contexts) offer faster reconfiguration time but it is more expensive in terms of silicon area.

In order to overcome these limitations in this work a new architecture called ADAPTO (Adder-based Dynamic Architecture for Processing Tailored Operators), has been proposed.

In the first part of our research we identified the algorithms to accelerate, then we use the results of this research in order to identify the basic hardware structure of our architecture. In this stage we assumed that the arithmetic unit of microprocessors hosts all the arithmetic functions optimized for the native wordlength. The main requirements for of the proposed RFU are:

- High flexibility (in order to be easily reconfigured for different operations, including the boolean ones).

-
- High reconfiguration speed, for resource reuse without the stalling of the processor during the program execution.
 - Low cost in terms of silicon area and power

After this identification we started with the ADAPTO architecture design. The ADAPTO architecture is a Reconfigurable array composed by reconfigurable elements, that perform both logical and arithmetic operations, and programmable interconnections composed by pass transistors devices. ADAPTO is composed by three alternated stripes of full adder based Logic Blocks (LBs) and interconnection networks based on pass transistors devices. The ADAPTO architecture has been developed to speed-up sub-word arithmetic operations and/or bit manipulation that are often used in Digital Signal Processing algorithms. It can be used to accelerate simple bits manipulation operations, static bit packing and unpacking but also more complex operations as modular additions, Montgomery Multiplication and operations in the Galois Field Galois Field are widely used in a large set of applications starting from cryptography [20], error detection and correction codes [21], digital signal processing [22].

Chapter 1

Reconfigurable Computing

In this chapter a brief overview about Reconfigurable Systems is done. The main features of the actual FPGA (Field Programmable Gate Array) architectures are described and the main reconfiguration strategies are illustrated. More details can be found in [13] and in [16].

1.1 Parallel computing vs Serial computing

There are two primary methods in traditional computing for the execution of algorithms [13]. The first is to use an Application Specific Integrated Circuit, or ASIC, to perform the operations in hardware. Because these ASICs are designed specifically to perform a given computation, they are very fast and efficient when executing the exact computation for which they were designed. However, after fabrication the circuit cannot be altered. Microprocessors are a far more flexible solution. Processors execute a set of instructions to perform a computation. By changing the software instructions, the functionality of the system is altered without changing the hardware. However, the downside of this flexibility is that the performance suffers, and is far below that of an ASIC. The processor must read each instruction

from memory, determine its meaning, and only then execute it. This results in a high execution overhead for each individual operation. Reconfigurable computing is intended to fill the gap between hardware and software, achieving potentially much higher performance than software, while maintaining a higher level of flexibility than hardware.

1.2 Reconfigurable systems

1.2.1 Field-Programmable Gate Array FPGA

Around the beginning of the 1980s, it became apparent that there was a gap in the digital IC continuum. At one end, there were programmable devices like SPLDs and CPLDs, which were highly configurable and had fast design and modification times, but which could not support large or complex functions. At the other end of the spectrum were ASICs [16]. These could support extremely large and complex functions, but they were painfully expensive and time-consuming to design. Furthermore, once a design had been implemented as an ASIC it was effectively frozen in silicon. In order to solve this problem, Xilinx [12] introduced a new class of Reconfigurable integrated circuit called FPGA which they made available in 1984. The actual FPGA are arrays of reconfigurable block interconnected by reconfigurable interconnection Fig 1.1.

The reconfigurable blocks are usually composed by LUTs, registers and multiplexer. in Fig 1.2 is shown a simple LUT based Logic Block, the LUT perform the logical operations and the output multiplexes select the registered or the not registered output.

The LUTs are typically SRAM based and are configured storing in the SRAM cell of the LUT the truth table of the desired boolean function as show in Fig 1.3. Each

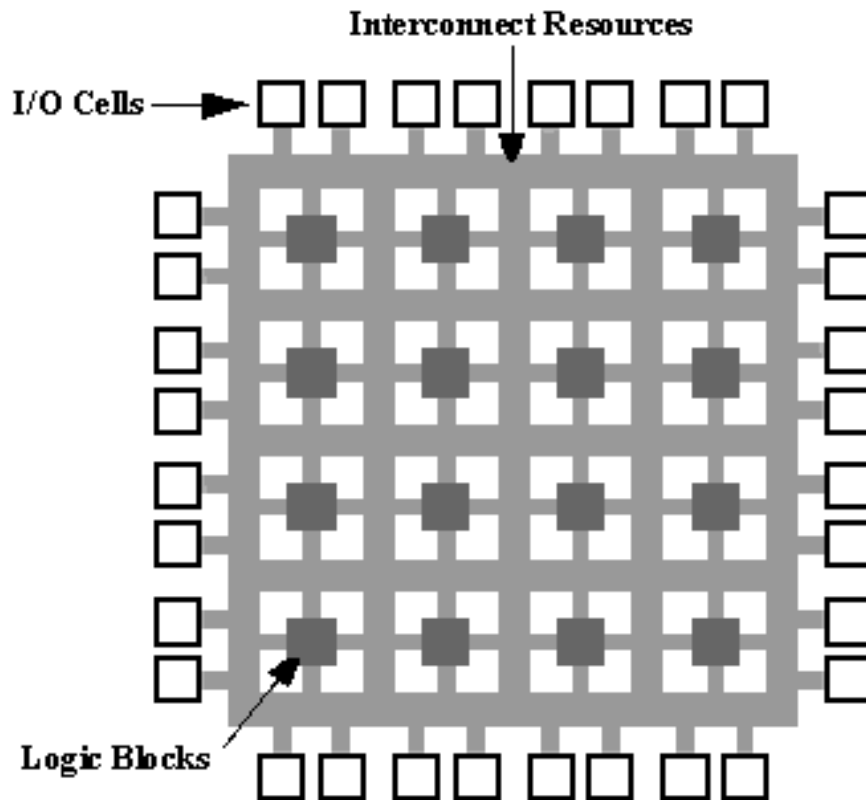


Fig. 1.1: FPGA architecture

LB output pin can connect to any of the wiring segments in the channels adjacent to it. In addition to the local interconnect, there would also be global (high-speed) interconnection paths that could transport signals across the chip without having to go through multiple local switching elements (Fig. 1.4). The device also include primary I/O pins and pads. All of the sequential elements inside an FPGA as the registers configured to act as flip-flops inside the programmable LBs need to be driven by a clock signal.

The clock signals comes typically into the FPGA via a special clock input pin, and is then routed through the device and connected to the appropriate registers. Actual FPGA are more complex respect than the simple architecture above described they can integrate Block Ram, microprocessors, embedded multipliers, embedded adders,

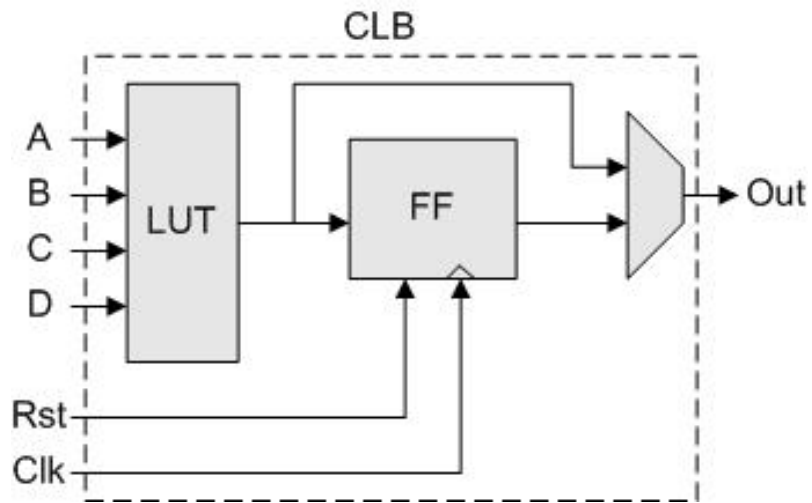


Fig. 1.2: LUT based Logic Block (LB)

DSPs and PLL to manage the clock signal. More informations about the FPGA world can be found in in [16].

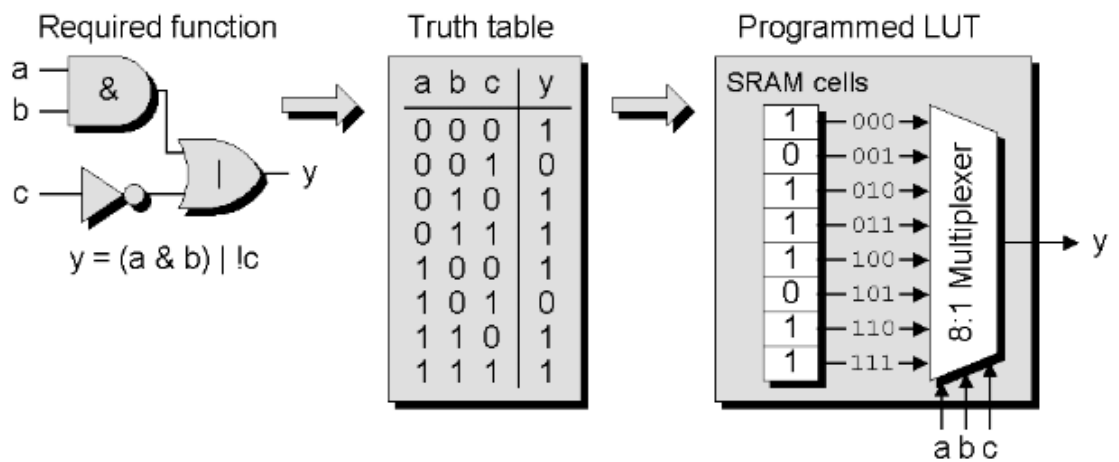


Fig. 1.3: LUT implementation of boolean expression

1.2.2 Reconfiguration strategies

There are a few different configuration memory styles that can be used with reconfigurable systems.

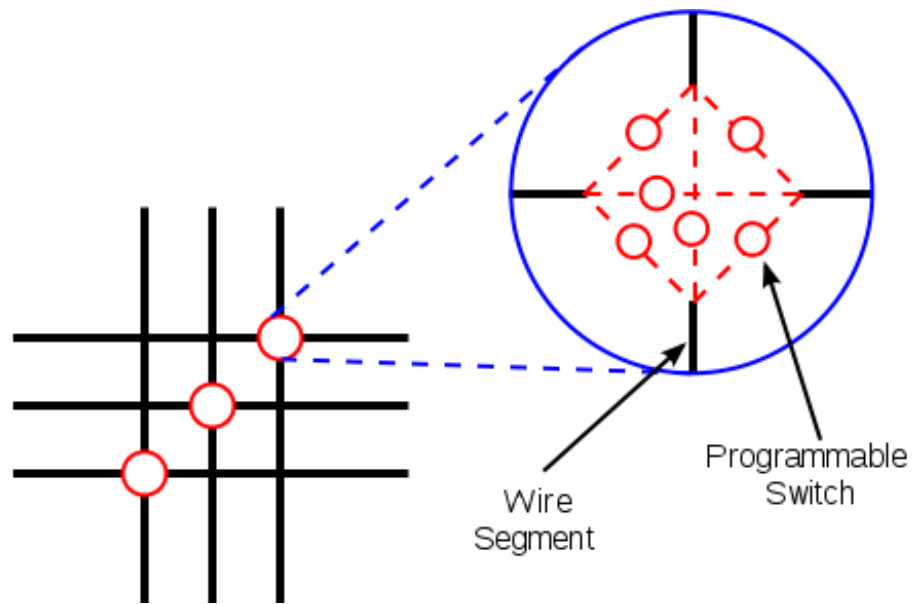


Fig. 1.4: FPGA programmable interconnect

A single context device Fig. 1.5 is a serially programmed chip that requires a complete reconfiguration in order to change any of the programming bits. Most commercial FPGAs are of this variety. To implement runtime reconfiguration on this type of device, configurations must be grouped into full contexts, and the complete contexts are swapped in and out of the hardware as needed.

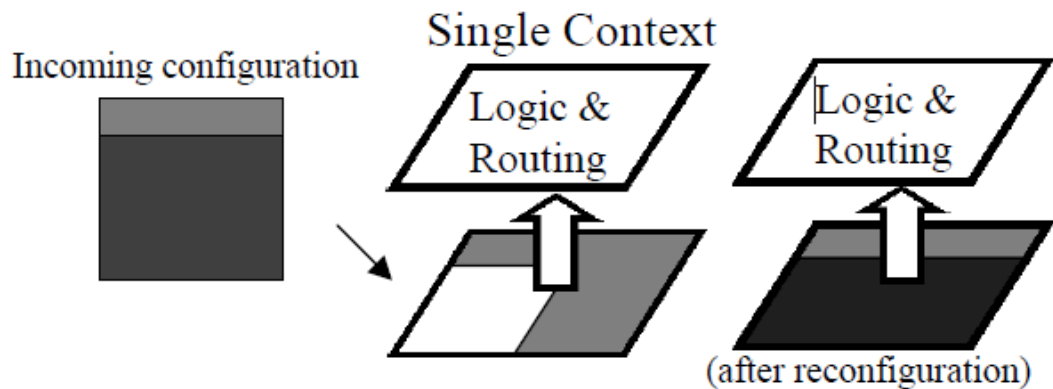


Fig. 1.5: Single context

A multi-context [14], [15] device Fig. 1.6 has multiple layers of programming

bits, where each layer can be active at a different point in time. An advantage of the multi-context FPGA over a single-context architecture is that it allows for an extremely fast context switch (on the order of nanoseconds), whereas the single-context may take milliseconds or more to reprogram. The multi-context design does allow for background loading, permitting one context to be configuring while another is in execution. Each context of a multi-context device can be viewed as a separate single-context device. In Fig. 1.7 is shown a 4 input 4 context LUT; the output multiplexer select the right context. The main disadvantage of this approach are:

- The large area occupation, the reconfigurable array has to be replicated for every context.
- The power consumption [14].

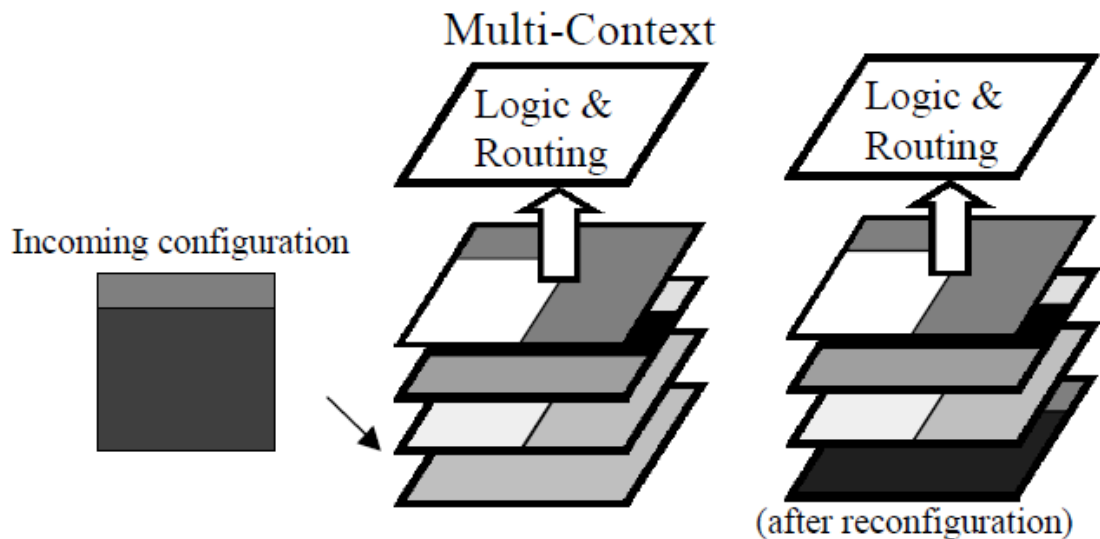


Fig. 1.6: Multi context

Devices that can be selectively programmed without a complete reconfiguration are called partially reconfigurable Fig. 1.8. The partially reconfigurable FPGA is also more suited to run-time reconfiguration than the single-context, because small

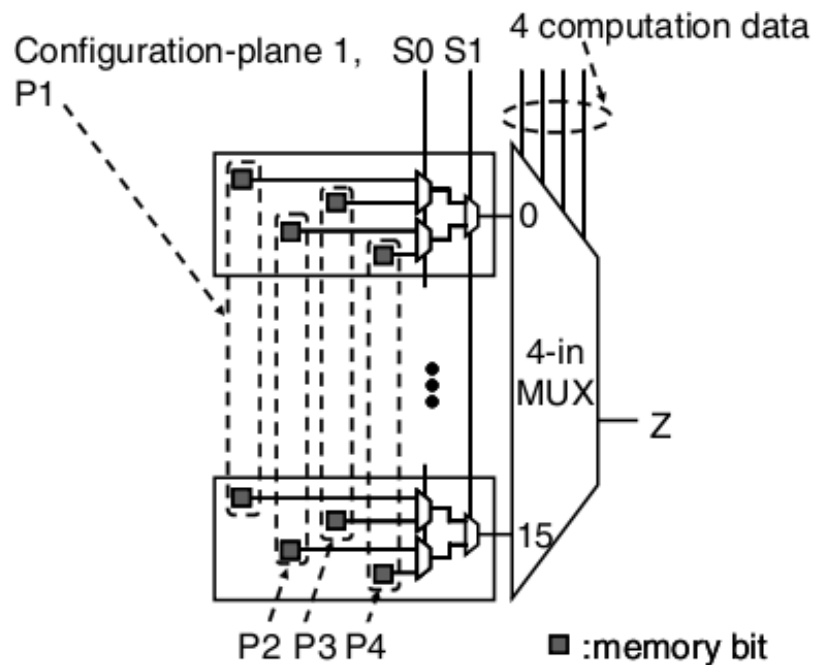


Fig. 1.7: 4 input N context LUT

areas of the array can be modified without requiring that the entire logic array be reprogrammed. This allows configurations which occupy only a part of the total area to be configured onto the array without removing all of the configurations already present. Furthermore, individual configurations can be selectively modified based on run-time conditions, such as changing registered constant values or a constant coefficient multiplier structure over time. These small reconfigurations require much less time than a full-chip reconfiguration due to the reduced data traffic.

For all of these run-time reconfigurable architectures, there are also a number of compilation issues that are not encountered in systems that only configure at the beginning of an application. Compilers must consider the run-time reconfigurability when generating the different circuit mappings, not only to be aware of the increase in time-multiplexed capacity, but also to schedule reconfigurations so as to minimize the configuration overhead. This is in order to ensure that the overhead of the reconfiguration does not eclipse the benefit gained by hardware acceleration. Stalling

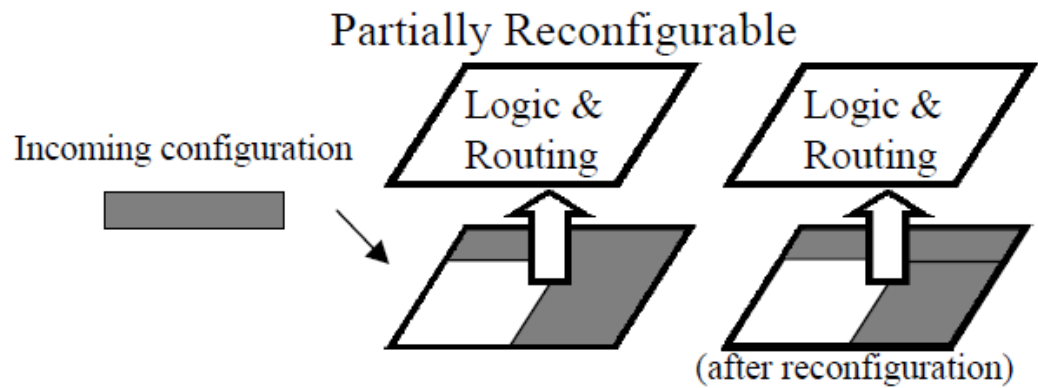


Fig. 1.8: Partial programming

execution of either the host processor or the reconfigurable hardware because of configuration is clearly undesirable. In some cases over 98% of execution time can be spent in reconfiguration. Therefore, fast configuration is an important area of research for run-time reconfigurable systems.

Chapter 2

State of the art

2.1 Reconfigurable Architecture as hardware accelerators

The requirement of increasing performances in processor system has motivated the development of special hardware units for the acceleration of the most time consuming operations. In order to reuse the same acceleration hardware in different operations, frequently these units use a reconfigurable structure. In this mixed architecture (based on a Processor core and a Reconfigurable Unit) the computing elements can interact in different ways. The specific interaction depends on the relative position and interfacing between the processor core and the reconfigurable unit. Generally, a tighter coupling leads to a smaller communication overhead and is suitable to accelerate micro-operations. On the other hand, a looser coupling requires a coarser granularity of operations to be implemented on the reconfigurable unit, in order to reduce the data transfer. Couplings can be classified into the following three main categories:

1. **Reconfigurable Functional Unit (RFU)**

The Reconfigurable Unit is integrated in the core of the processor as any other Functional Unit. This solution implies the extension of ISA (Instruction Set Architecture). The processor core fetches and decodes instructions and issues the instructions to the corresponding units. This approach allows a very fast interaction between the core and the RU but in general, if an RFU interface is not present in the processor, it requires a core redesign. RFUs as other Functional Unit use the core Register File to write and read data. Typically for RFU exist two types of operations: instructions that start the RFU reconfigurations and instructions that execute the operations.

2. Coprocessor

Coprocessors are very similar to RFU but the Reconfigurable Unit is placed outside the core, it's connected to the processor by a coprocessor interface that avoids the core redesign. As for the RFU the use of a coprocessor implies the ISA extension but different to RFU in addition to the Reconfiguration and Execution operations there is a data transfer operation between the Core and the Reconfigurable Unit. Coprocessors cannot read data from the core Register File, often they have an own Register File.

3. Attached processing unit

The Reconfigurable Unit is placed outside the processor using a BUS and there is not any ISA extension. The data transfer between the processor and the Reconfigurable Unit is slower respect the previous solutions.

The above mentioned integration techniques are shown in fig 2.1.

Among these three solutions the RFU is surely the most efficient in terms of acceleration. By using this approach, the best execution speed is obtained by reconfiguring the RFU as fast as possible. In this way the RFU can execute the selected operations as soon as they are scheduled.

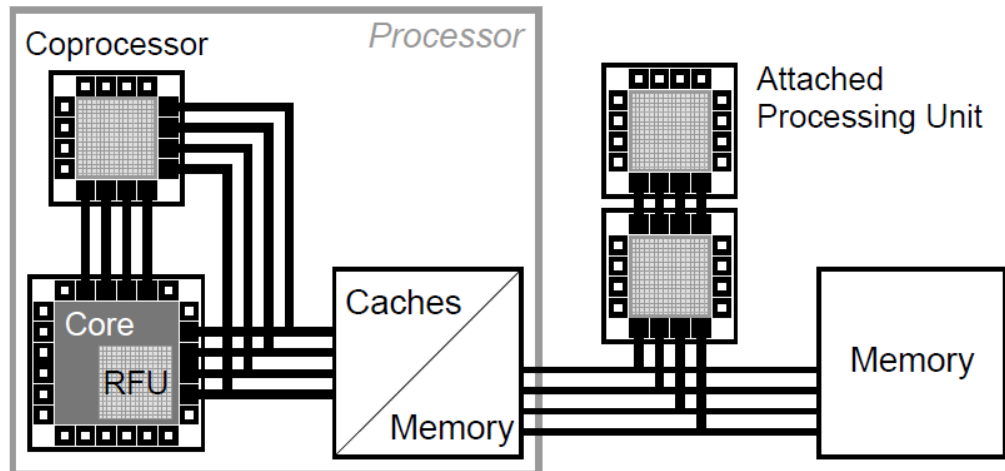


Fig. 2.1: Possible integrations between processor and reconfigurable unit

2.2 PRISC

The PRISC architecture explores a novel way to incorporate hardware-programmable resources into a processor microarchitecture to improve the performance of general-purpose applications [1] [2]. In the PRISC architecture, the hardware-programmable resources is inserted directly to the CPU datapath as a RFUs. Typically, the implementation of a particular function in a RFU is significantly slower than the implementation of the same function in a highly-customized functional unit. As such, PFUs are added in parallel with the existing functional units so that they augment (not replace or replicate) the existing datapath functionality (Fig. 2.2). The RFU has two input ports where it accepts operands and a single output port for the result.

This reconfigurable array is composed by alternating layers of two basic components: interconnection matrices and logic evaluation units. Each possible interconnection point in the interconnection matrix is implemented with a CMOS nchannel transistor. By appropriately setting the value in the memory cell, we can connect or disconnect the two lines. Each logic evaluation unit implements a hardware truth

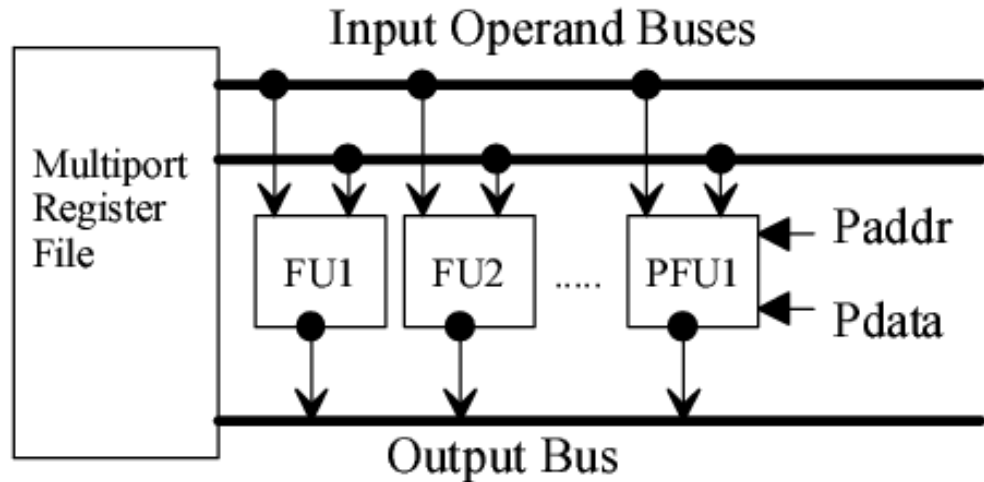


Fig. 2.2: The PRISC architecture

table, called a Look-Up Table (LUT). Each LUT memory cell in a PFU is addressable, and in fact, all of the PFU memory cells can be viewed as a large SRAM which is loaded by using the PFU Paddr and Pdata ports. Programming a PFU to implement a particular function then consists of loading the appropriate values into the interconnection matrix memory cells and the LUT memory cells. The PRISC architecture consist 3 alternating layers of interconnect and LUTs requires 30.528 transistors for a 32 bit datapath. The Reconfigurable Array is a single context (Chapter 1.2.2) architecture and require about 500 clock cycles to be reconfigured. It implies that, in order to avoid this waste of time, it is a good solution use more than one Reconfigurable Array in parallel. In this way the obtained architecture can be see as a multicontext array.

2.3 The Chimaera Unit

The Chiamera architecture integrates a small and fast LUT based RFU into the pipeline of a superscalar processor [7]. The Chimaera system treats the reconfigurable logic not as a fixed resource, but instead as a cache for RFU instructions. Those

instructions that have recently been executed, or that we can otherwise predict might be needed soon, are kept in the reconfigurable logic. If another instruction is required, it is brought into the RFU, overwriting one or more of the currently loaded instructions. In this way, the system uses partial run time reconfiguration techniques to manage the reconfigurable logic. The chimaera architecture is shown in Fig 2.3. The main component of the system is the Reconfigurable Array, which consists of FPGA like logic designed to support high performance computations. It is here that all RFU instructions will actually be executed. This array gets its inputs directly from the host processors register file.

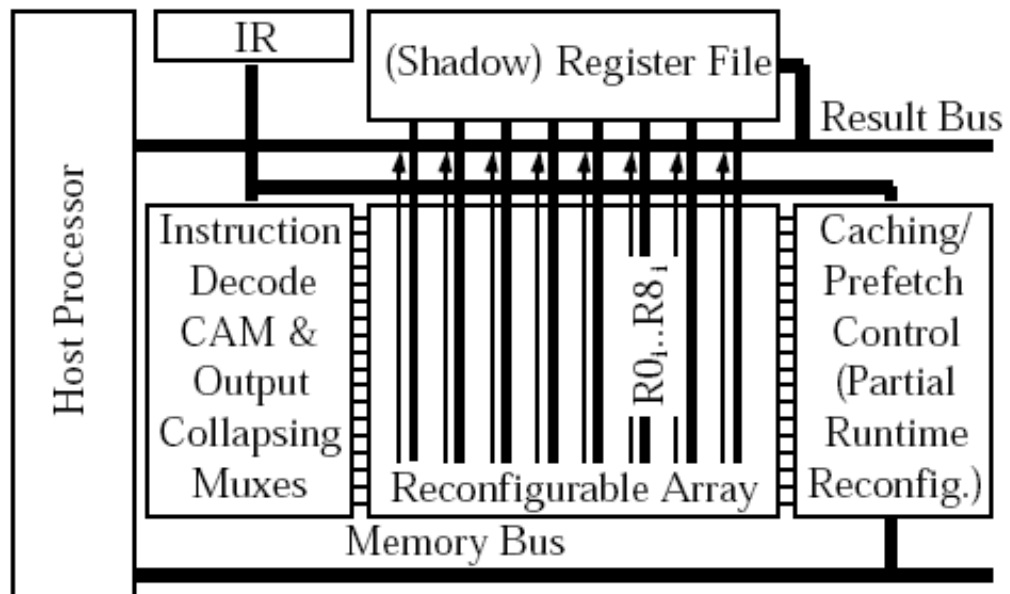


Fig. 2.3: The CHIMAERA reconfigurable Unit

Next to the array is a set of Content Addressable Memory locations, one per row in the Reconfigurable Array, which determine which of the loaded instructions are completed. The CAMs look at the next instruction in the instruction stream and determine if the instruction is an RFUOP, and if so whether it is currently loaded. If the value in the CAM matches the RFUOP ID, the value from that row in the Reconfigurable Array is written onto the result bus, and thus sent back to

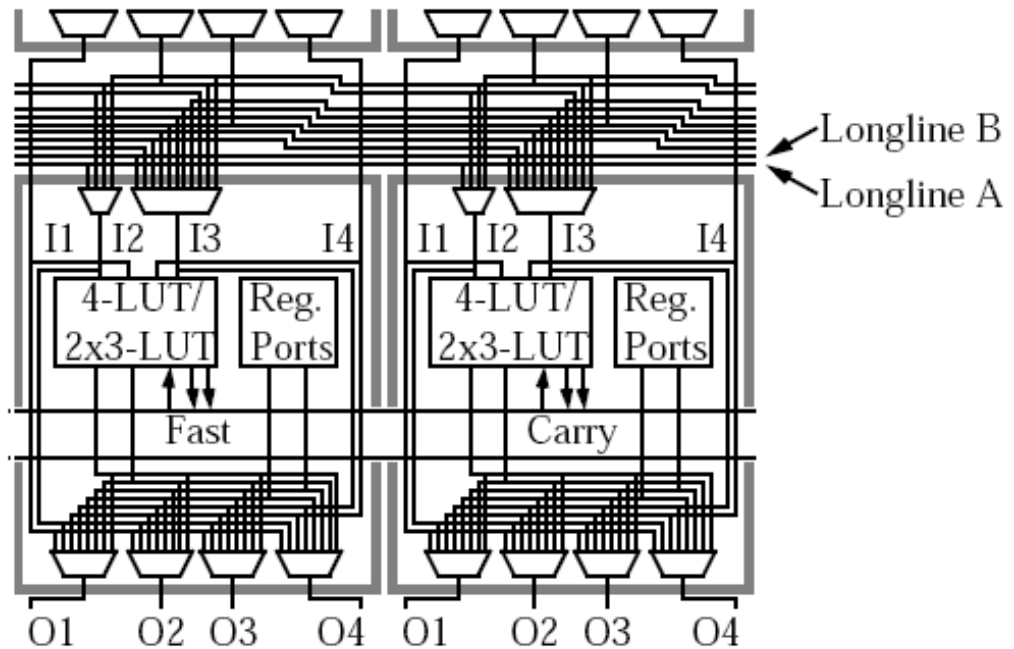


Fig. 2.4: The CHIMAERA reconfigurable array

the register file. If the instruction corresponding to the RFUOP ID is not present, the Caching/Prefetch control logic stalls the processor, and loads the proper RFU instruction from memory into the Reconfigurable Array. The caching logic also determines which parts of the Reconfigurable Array are overwritten by the instruction being loaded, and attempts to retain those RFU instructions most likely to be needed in the near future. Reconfiguration is done on a per-row basis, with one or more rows making up a given RFU instruction.

The Reconfigurable Array itself is shown in Fig.2.4 and. This architecture has been inspired by the Triptych FPGA [8] and [9]

The reconfigurable logic is broken into rows of logic cells between routing channels. Within that row, there is one cell per bit in the processor's memory word, so for a 32-bit processor there are 32 cells per row. All cells in a given column I have access to the I th bit of registers R0- R8, allowing it to access any two of these bits. Thus, a cell in the rightmost (0th) column in the reconfigurable array can read

any two least significant bits from registers R0 through R8. Which register(s) a cell accesses is determined by its configuration, and different cells within the array can choose which registers to access independently.

The Chimaera approach maps several context in different part of the reconfigurable array in order to avoid reconfiguration time. In this way there is a very fast context switching but there is a waste of hardware: the entire architecture is never used.

2.4 A solution for the hardware wasting

The main problem in using a RFU based on LUTs is related to the impossibility to reload the configuration memory in a very short time. In fact, the use of LUTs implies the reloading of all the configuration bits, making it incompatible with a runtime reconfiguration. Similar problem arises for the reconfiguration of interconnect, but in this case it is possible to find simple solutions for reducing the bit amount. For this reason, every time the RFU need to perform an operation, the following steps must be performed:

1. Stopping of the program execution,
2. Loading of the configuration bits in the LUTs,
3. Execute the operation.

In order to increase the reconfiguration speed different strategies can be used:

1. Map several contexts in different parts of reconfigurable LUT array [7].
2. Use partial reconfiguration of LUTs in more complex structures (for instance the pipeline reconfiguration used by Piperench [10]).

3. Use a multicontext approach.
4. Replace the LUTs with another element characterized by a simpler and faster reconfiguration procedure.

The main drawback of the first three solutions is the great hardware complexity and power consumption. Only a small part of the computational units (LUTs) are used during the execution of an Instruction. For example, in the second solution the partial reconfiguration does not allow the complete exploitation of the available resources (those involved in the reconfiguration phase).

Moreover, the reconfiguration policy could limit the algorithms that can be implemented. In particular, the architecture proposed in [10] is based on processing stripes and the pipeline reconfiguration is carried out on a subset of stripes. Consequently, during the RFU computation this subset is not available for the processing. Moreover, only pipelined applications can be efficiently executed using this approach [11]. These characteristics reduce the performance and the flexibility of the structure, reducing the overall efficiency.

To solve this problem we coupled the third and the fourth solution. The Architecture presented in this work uses a multicontext approach but logical and arithmetic operations are not performed by LUTs. The LUT is very useful if maximum flexibility is required, it assures the maximum reconfiguration level through the reloading of the reconfiguration stream. RFU using multi-context approach can be used to perform more than one operation after the reconfiguration. LUT is not very efficient in the case of multi-context architecture. In this case each context is stored in a different LUT and context switching corresponds to select the output of a different LUT. This technique introduces a resource wasting because every LUT has to be reloaded for every context incrementing the cost in terms of silicon area.

To reduce the resource wasting, we can use a different approach based on a reconfigurable cell, whose configuration is stored in a local memory. These two different approaches are shown in Fig. 2.5. and 2.6..

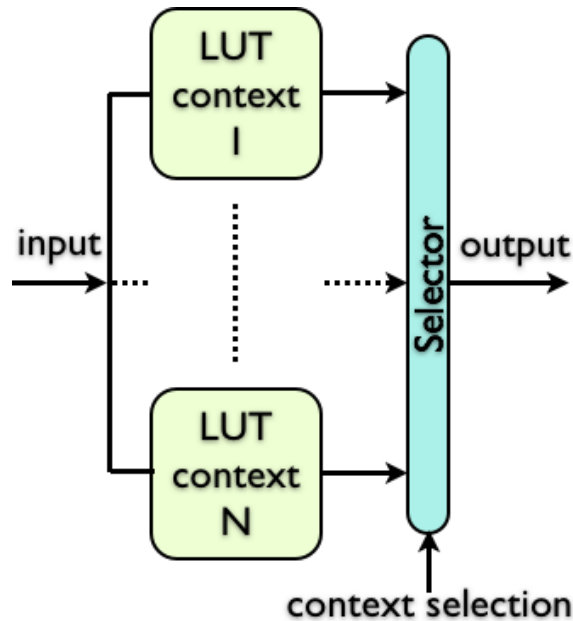


Fig. 2.5: Multicontext LUT

If we analyze the performance results coming from different algorithm implementations, we observe that the most important operators responsible for performance reduction in conventional microprocessor and DSP are:

- Additions and subtractions on sub-words.
- Special operators, as the minimum and maximum selection.
- Logical operators (OR, AND, ...) operated at individual bit level.
- Bit interleaving (special cases are bit shifting and bit selection)

A possible methodology for the design of reconfigurable cell can be based on the following steps.

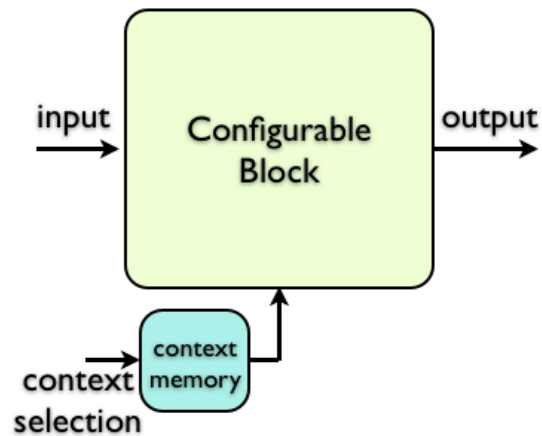


Fig. 2.6: Resource sharing

1. Select the most complex bit-level operation to perform and use its implementation as first attempt of reconfigurable cell.
2. Verify if all the other operations required can be obtained from this cell through reconfiguration and selection of input and output bits.
3. If the above point is not verified, modify the cell in order to cover the missing operators.

In our case, the most complex operation corresponds to the 1-bit full addition (including carry in and carry out for cell cascade). Consequently the optimum reconfigurable cell (in terms of area) is the full-adder. In order to confirm this choice we have to demonstrate that each of the operators in the list can be obtained by a suitable configuration of the full-adder. Forcing one or two pin of the full-adder cell to 1 or 0 it can be easily configured to perform the following operations:

- one bit addition
- logical NOT

- logical PASS
- 2 input AND
- 2 input OR
- 2 and xor
- 3 input xor
- 3 Majority function

These operations are the required ones for our intention, for this reason we decide to use the full-adder as computational element in the ADAPTO architecture.

More detailed informations about the hardware architecture of the ADAPTO unit are shown in 3.

Chapter 3

The ADAPTO Reconfigurable Functional Unit

In this Chapter the ADAPTO Reconfigurable Functional Unit is described. In section 3.1 the description of the architecture will be done meanwhile in 3.2 a brief description of the principal circuit composing ADAPTO will be given. A detailed description of all the subcircuit composing ADAPTO is in 3.2.

3.1 ADAPTO architecture

The ADAPTO (Adder-based Dynamic Architecture for Processing Tailored Operators) RFU is a multicontext reconfigurable architecture composed by three stripes of reconfigurable circuits called Logic Block (LB) and three stripe of reconfigurable interconnect. Each LB stripe is connected with the stripe below using one stripe of programmable interconnect as shown in Fig. 3.1. ADAPTO has three 32-bit input and one 32-bit output for the data and an additional N-bit input for the context selection (the 16 context implemented version of ADAPTO requires a 4 bit input for this purpose). In order to reduce the number of transistors and conse-

quently the silicon area, data can flow only from the up to the button of the array without the possibility of any feedback.

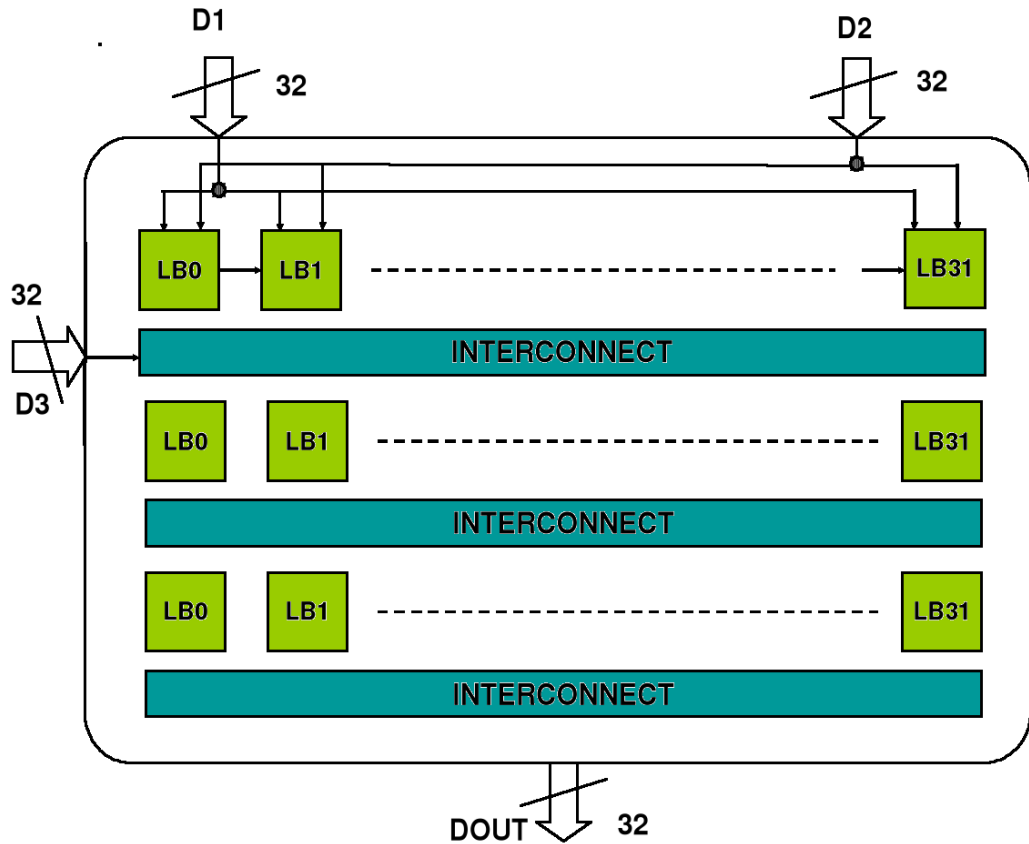


Fig. 3.1: The ADAPTO architecture

3.1.1 Logic Block

As mentioned in Chapter .2 in order to reduce the reconfiguration's time of the RFU and the resource wasting we choose to utilize a different computational unit with respect the conventional LUT. We shown that the full adder can be easily configured for performing the following operations: one bit addition, NOT and PASS, 2 input AND, 2 input OR, 2 input XOR, 3 input XOR and 3 Majority. The different functions can be selected by forcing one or more input pins of the FA to a fixed value. For instance, a 2 input AND is obtained by putting the C_{IN} input to 0 and

taking as output the C_{OUT} pin. Clearly, this structure is less flexible than one based on LUTs, but on the other side its reconfiguration is fast because it requires to change few configuration bits. Another advantage of this solution is the lower number of MOSFETs required for its implementation respect than the LUT approach.

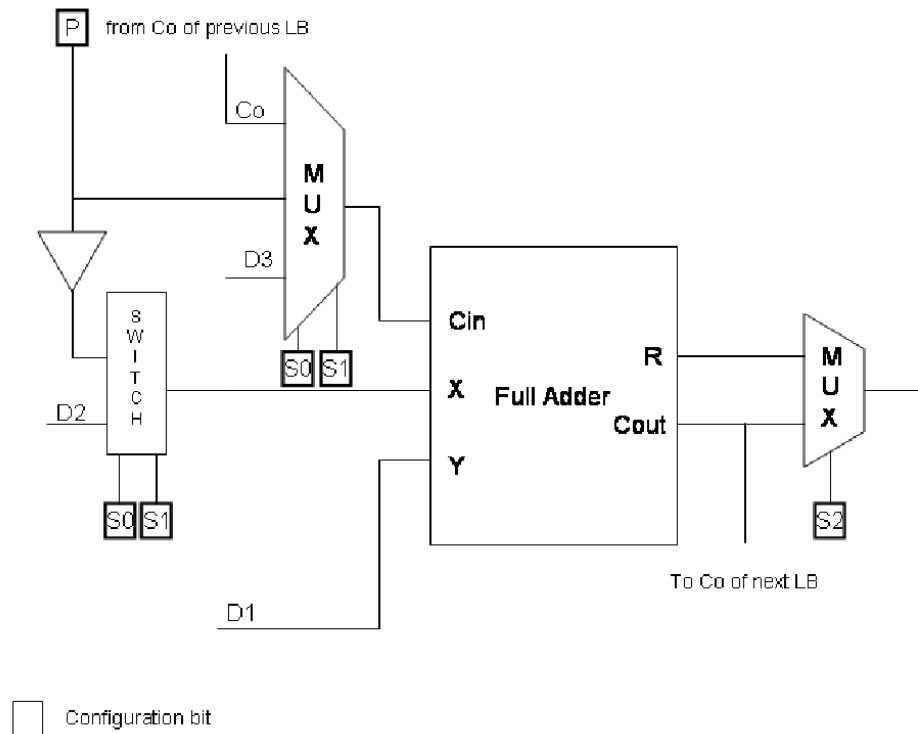


Fig. 3.2: The ADAPTO Logic Block

The basic computational element of the new architecture, shown in Fig. 3.2, is the LB (Logic Block). It is based on two multiplexers, a selector (realized by a multiplexer with a suitable coding of selection bits) and a full adder. The input and the output multiplexer together with the selector are used for programming purposes.

In particular inputs S_0 , S_1 , S_2 , and P are used to select the operation to be performed and the operands (Fig. 3.3), these configuration bits are stored in the context memory (see Fig. 3.4). The signal C_O is directly connected to the signal C_{out} of the previous LB. If a zero carry is required (for example in the case of the

C_{in}	X	Y	C_{out}	R
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Fig. 3.3: Full adder's table of truth

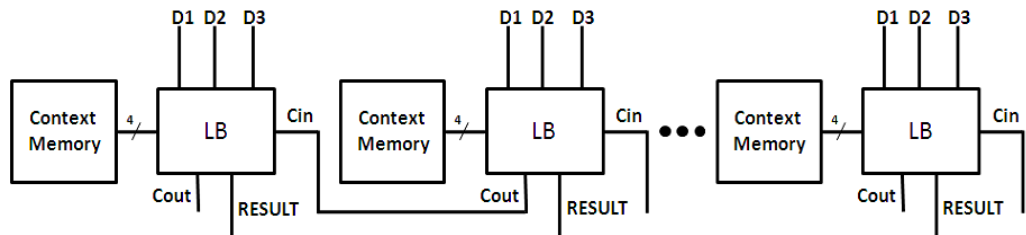


Fig. 3.4: A multicontext ROW

LSB of a multi-bit adder) 0 input is selected by the configuration bits S_0 and S_1 . In Fig. 3.1.1 are shown all the operations an ADAPTO's LB can perform.

A LUT based implementation of the LB is surely more flexible respect than the proposed one; using LUT it is possible realize more boolean fuctions as illusted in Chapter 2.4. A LUT based LB able to perform the one bit addition with the result and the carry out generation is show in Fig. 3.5 . This circuit require about 200 transistors for the implementation, but if we consider a 16 context LB the total amount of transistor is about 3000. On the other way the 16 context full adder

P	S_0	S_1	S_2	<i>Operation</i>
-	0	0	-	SUM
0	0	1	1	2 AND
0	0	1	0	2 XOR
1	0	1	1	2 OR
1	0	1	0	2 XOR'
-	1	0	0	3 XOR
-	1	0	1	3 MAJORITY
1	1	1	0	3 NOT
0	1	1	0	3 PASS

Table 3.1: Full adder's operations

based LB proposed as computational unit for the ADAPTO architecture require 631 transistors.

The ADAPTO's carry chain uses a direct interconnect (linking adjacent LBs) for the speeding-up of the carry propagation in multibit adders (Fig. 3.4)

3.1.2 Interconnect

Interconnect network is based on pass-transistor devices and it's used for link purpose and shift operations with 1 or 0 insertion. All the output coming from the previous stripe's LBs must have the possibility to be connected to every input of every LB of the next stripe. It's clear that in order to have a multicontext interconnect we have to connect the gate of every pass transistor to a multicontext memory. Let's consider an interconnect stripe having 32 LB with 3 inputs and 1 output, a complete interconnect structure requires for each output $3 * 32$ pass transistor and, consequently, $3 * 32 * N$ configuration bits (where N is the number of the

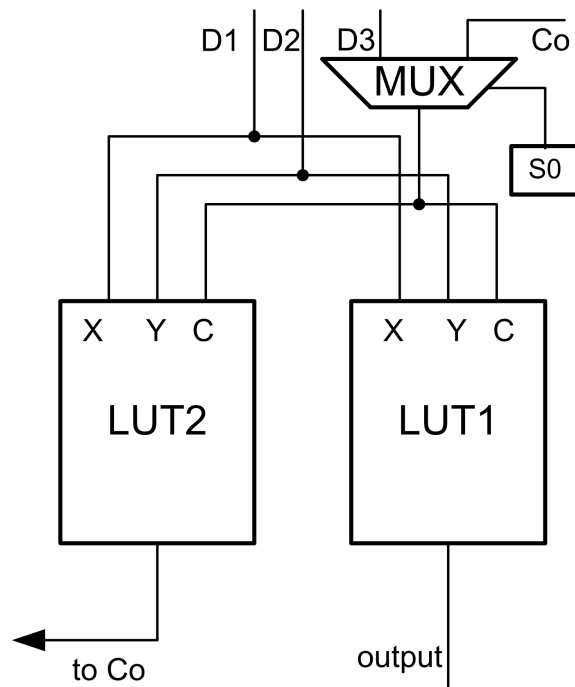


Fig. 3.5: LUT based LB

contexts). Using the above strategy, a multicontext interconnection between two rows of the array needs $3 * 32 * (32 + 1) * N = 3168 * N$ configuration bits, where the 1 value is added in order to consider the additional wire utilized for 0/1 insertion operation. Observing that an LB input must be connected to a single LB output of the previous row, it's possible to reduce considerably the number of reconfiguration bit using binary coding, as showed in Fig. 3.6. In this structure the selection of one of the $(32 + 1)$ outputs coming from the previous row is performed by a decoder addressed by the output of the context memory (reducing the amount of data that must be stored). This solution is shown in Fig. 3.7

Every interconnect stripe has a different size. The first one is the most complex because, in addition to the 32 output coming from the first LB stripe, there are the 32 signal coming from the third input operand. Moreover, we have to consider that for each of the three stripes of interconnect there are two additional wires directly connected to 1 and 0 used for constant additions and shift with 1 or 0 insertion.

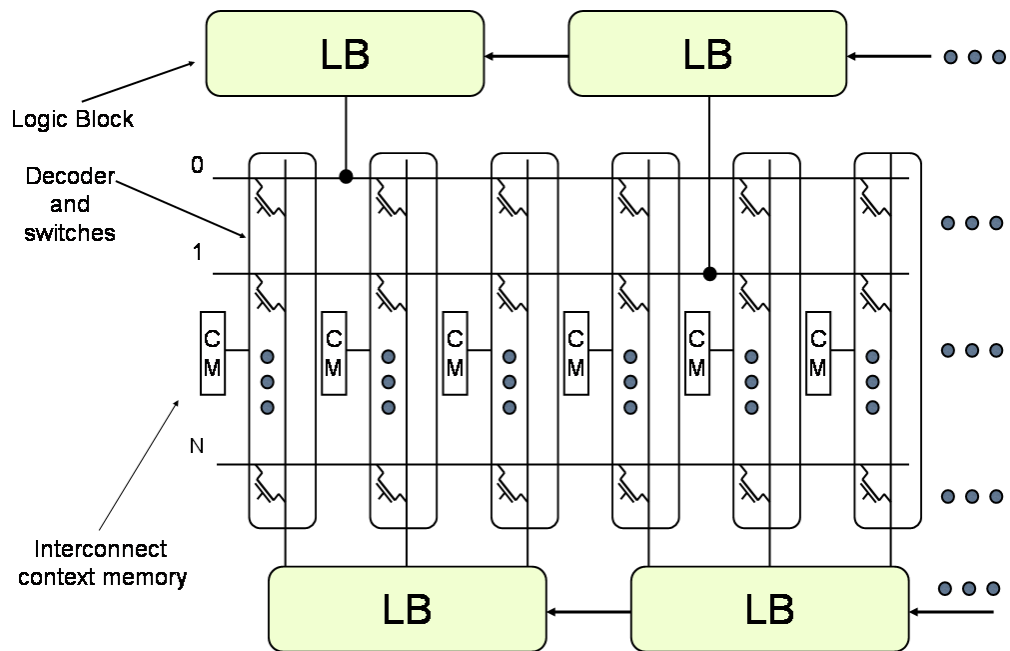


Fig. 3.6: Interconnect network

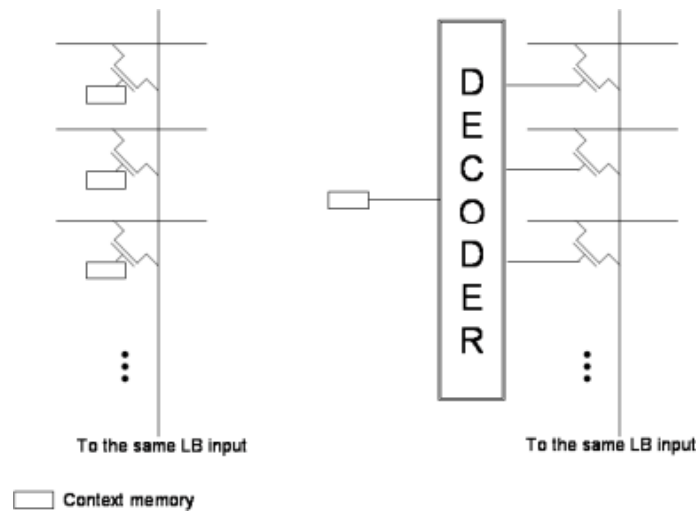


Fig. 3.7: Interconnect reduction

The total number of wires for the first level of interconnect is $32 + 32 + 2 = 66$. The decoders used for the interconnect reduction have 66 outputs and 7 configuration bits. Because the inputs of the second level of LB are $332 = 96$ 96 decoders are required and the number of pass transistors required is $9666 = 6336$

In the second interconnect level there are the 32 outputs coming from the second

LB stripe plus the two wires linked to 1 and 0. For this reason the decoders have 34 outputs and 6 inputs. The inputs of the following LB stripe are 96 so there are 98 decodes and $96 \times 34 = 3264$ pass-transistors. Finally in the third interconnect level, similar to the second there are the 32 output coming from the previous LB stripes, the two lines for the 1 and the 0 but all these signals can be linked only to the 32 bit of the results so we have only 32 decoders instead of 96. The number of pass transistors is $32 \times 96 = 3072$.

3.1.3 Context Memories

The ADAPTO architecture proposed in this work is a 16 context reconfigurable array. This means that ADAPTO, at every processor clock cycle can select one between 16 prestored configurations. In order to write the Context Memories a 32 bus is used. This bus is connected to all the memories in the architecture. To use in a efficient way the 32 bit of the bus we exploit that all the memories has size $16 \times N$ where N is the number of bit of the stored word. For this reason is useful write in two memories simultaneously. The first 16 bits of the bus are stored in the even memories meanwhile the second 16 bit are stored in the odd ones as shown in Fig 3.8. In order to manage the write operation, the generation of a control signal is required. For this purpose it is implemented a chain of D flip flop configured as shift register. The output of every flip flop is linked to the N^{th} line 16 bit of every memories couple (one even and one odd) as shown in Fig. 3.9. The writing operation begin putting a 1 in the flip flop chain configured as shifter register. For each of the three stripes composing the architecture the LB memories are written before and the interconnect memories after starting from left to right. The total memory capacity is $6144 + 23040 = 29184 \text{ bit}$. Writing 32 bit at every clock cycle, 912 clock cycles are required. The implemented ADAPTO unit works at 100 MHz so for the

reconfiguration $9.12 \mu s$ are required.

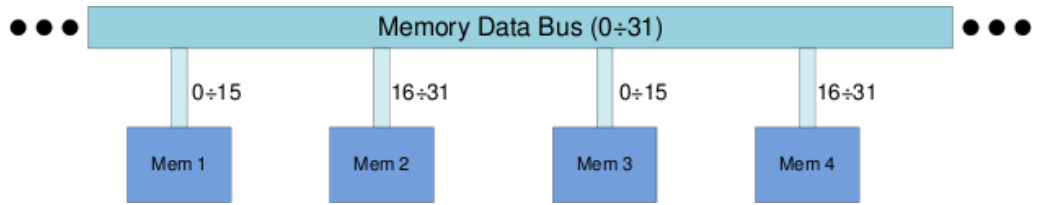


Fig. 3.8: Context memories Bus

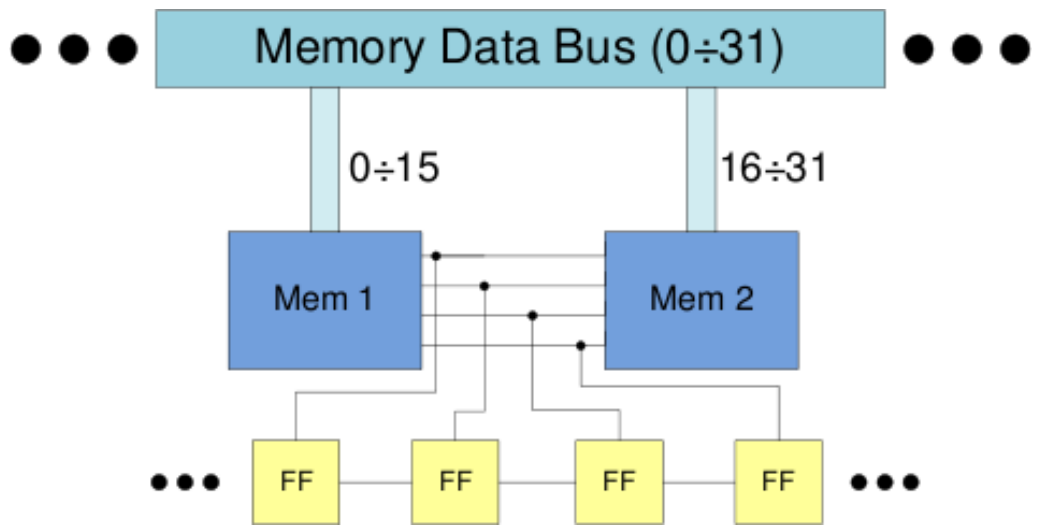


Fig. 3.9: Context Memories configuration

3.2 The ADAPTO IC design

In this section the integrated circuit layout development is described. In order to realize the entire architecture we start implementing a standard cell library composed by simple elements (inverters, adders, decoders...) suitable to implement more elaborated circuits. The library is realized using the *TSMC0.18 m* technology, for the implementation the following software are been used:

1. Cadence ps spice for the pre layout simulation.

2. Cadence Virtuoso for the layout realization.
3. Cadence Spectre for the post layout simulation.

The simulations are performed using 1.8 V for the Voltage and 27°C for the temperature. The extrapolated parameters by the simulation are reported in the following. More informations about the complete standard cell library developed for the ADAPTO design can be found in [17]

3.2.1 Delay

Propagation delay is the time required by a signal to propagate through a gate or net. Propagation delay of a gate or cell is the time it takes for a signal at the input pin to affect the output signal at output pin.

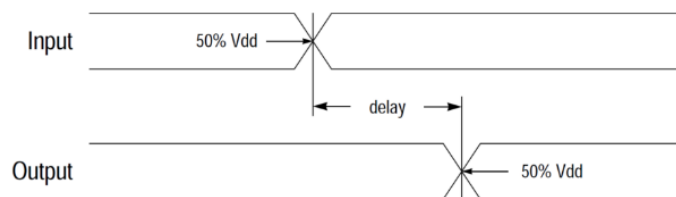


Fig. 3.10: gate delay time

For any gate, the propagation delay is measured between 50% of input transition to the corresponding 50% of output transition (Fig. 3.10). The delay time is calculated considering the load variation and both the high->low transition and the low->high one are estimated. The delay time is expressed in ns and in function of the unitary inverter realized with the same technology.

3.2.2 Setup Time and Hold Time

The following definition are defined only for sequential systems. Setup time Fig. 3.11 is the time relative to a clock event during which the data input to a latch or flip-flop

must remain stable in order to guarantee that the latched data is correct. Hold time Fig. 3.12 is the time following a clock event during which the data input to a latch or flip-flop must remain stable in order to guarantee that the latched data is correct.

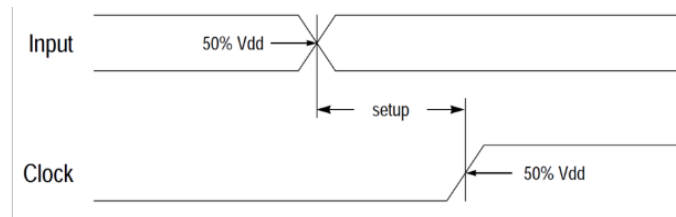


Fig. 3.11: Setup time

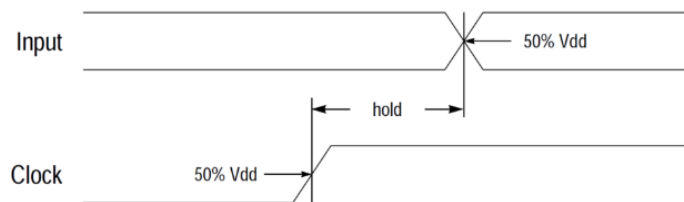


Fig. 3.12: Hold time

3.2.3 Input Capacitances

The input Capacitances affects the delay time of digital circuit. This important parameter has been estimated using linear fitting method.

3.2.4 Power Consumption

The power consumption in a digital circuit depends from the following parameters:

- **Clock Frequency.** As shown in Chapter. 3.3.1 the ADAPTO unit can work until 100 MHz.
- **Activity Coefficient.** This factor represent the probability that the output of a digital circuit has a transition on the clock raising edge. We estimate the

Power consumption assuming an Activity Coefficient of 100%

3.2.5 Logic Block

The Logic Block (LB) symbol is shown in 3.13, it has eight one bit inputs, four for the operands and for for configuration bits. The configurations bits are stored in context memories.

The data input are D_1 D_2 and D_3 meanwhile configuration inputs are S0 S1 S2 and P. The LB circuit, as mentioned in the previous section is composed by the following subsystems: a 3 INPUT Mux, a Selector, a FULL Adder and a 2 input Mux. (see Fig. 3.14) The output Y is buffered by an inverter used in order to load the following interconnections meanwhile the output Cout is linked to the Cin of the adjacent LB for the carry propagation. The implemented layout is shown in 3.15.

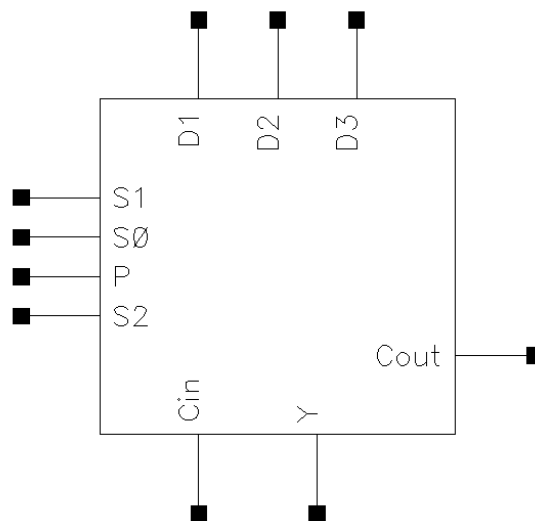


Fig. 3.13: The LB symbol

Tables 3.2.5, 3.2.5 show the LB performance in terms of area, capacity, delay time and power consumption, all the information are extrapolated using the above mentioned softwares. In Fig. 3.16 and 3.17 are shown the simulation results of the delay time for the C_{OUT} and Y pins.

name	length	weigth
<i>LB</i>	9.99	50.22

Table 3.2: LB size, dimension are expressed in micron

Pin name	Capacity (fF)
<i>P</i>	7.454
<i>S0</i>	10.24
<i>S1</i>	10.12
<i>S2</i>	24.48
<i>D1</i>	2.781
<i>D2</i>	3.114
<i>D3</i>	2.559
<i>C0</i>	4.45

Table 3.3: LB Pin capacity

Pin name	delay	delay respect inveter C=0
$Y_{LH}(PS)$	1205	35, 78
$Y_{HL}(PS)$	1366	56, 52
$C_{outLH}(PS)$	586	18.31
$C_{outHL}(PS)$	601	26.13

Table 3.4: LB delay

Consumption	Consumption on real load	static
<i>Power</i>	$36,55\mu$ W	703.6 pW

Table 3.5: LB power consumption

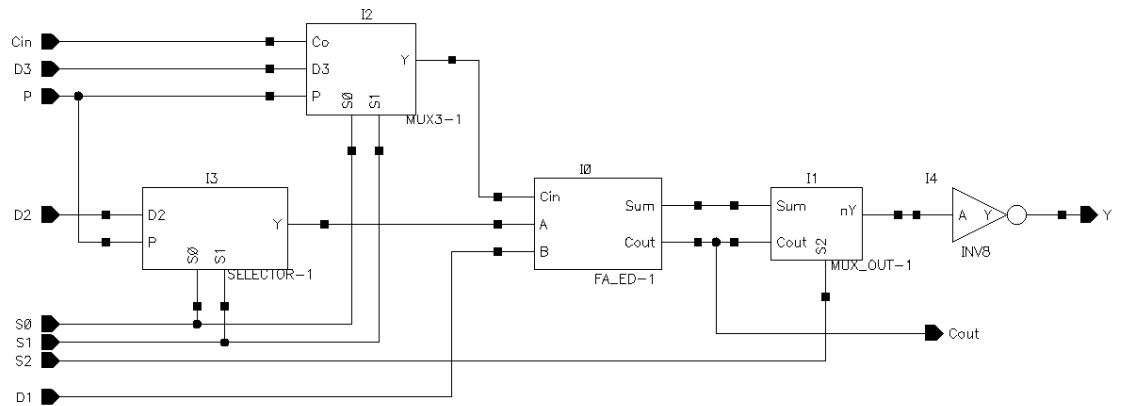


Fig. 3.14: The LB schematic

3.2.6 Full adder

As described in Chapter 2 the ADAPTO computational element is the full adder that, properly configured, can be used to perform both logical and boolean operations. The Full Adder used for ADAPTO is presented in [18], it is a pass transistors architecture and it is shown in in Fig. 3.18. The realized layout is show in Fig. 3.19 and the mains features are in Tab. 3.2.6, Tab. 3.2.6, Tab. 3.2.6 and Tab. 3.2.6.

name	length	weigth
<i>LB</i>	9.99	17.1

Table 3.6: Full Addee size, dimension are expressed in micron

Pin name	Capacity (fF)
<i>A</i>	2.562
<i>B</i>	2.781
C_{IN}	2.339

Table 3.7: FA Pin capacity

Pin name	$C = 0$	$C = 5$	$C = 12.5$	$C = 25$	$C = 75$	$C = 150$	t_{dinv}
$SUM_{LH}(ps)$	166	207	272	329	532	868	5.187
$SUM_{LH}(ps)$	399	441	490	562	802	1139	17.34
$C_{outLH}(ps)$	233	261	298	357	584	921	7.27
$C_{outHL}(ps)$	154	195	241	305	556	864	6.695

Table 3.8: FA delay

	$C = 0$	$C = 5$	$C = 12.5$	$C = 25$	$C = 75$	$C = 150$	<i>static</i>
Power μW	8.888	10.16	12.01	14.91	25.34	40.86	2559.9pW

Table 3.9: FA Power consumption

3.2.7 ADAPTO

The floorplan of the ADAPTO IC is shown in Fig.3.20. The input data D_1 and D_2 enter directly in the first LB layer meanwhile the third operand D_3 enter in the first stripe of interconnect. The architecture has additional ports for the output the context selection the clock and the reconfiguration. In Fig. 3.21 is shown the ADAPTO layout meanwhile Fig.3.22 underline the main blocks of the architecture.

3.3 ADAPTO IC main features

In this section the main features of the ADAPTO IC are shown. Such features are in terms of delay, power consumption and area occupation

3.3.1 Delay time

The delay time of the propose architecture is the sum of two contributions

- Context switch time. The time required to a prestored configuration to another.
- Computational time The time required for the propagation of data from the

input to the output of ADAPTO.

$$t_{CONFIG} = 976 \text{ ps}$$

$$t_{COMP} = 8.906 \text{ ns}$$

$$t_{Dtot} = t_{CONFIG} + t_{COMP} = 9.882 \text{ ns}$$

This delay implies a maximum frequency about 100 MHz that is compatible with the frequency of actual embedded processors. Better performance can be obtained using a more scaled technology.

3.3.2 Power consumption

The power consumption of ADAPTO has been estimated from the power consumption of the single elements of the implemented standard cell composing the architecture. [17]. In the following are shown the static and dynamic power consumption:

$$P_S = 15.587 \text{ } \mu\text{W}$$

$$P_D = 1.593 \text{ W}$$

These data are estimated from simulation in which the outputs activities of every block has been assumed 100%, this implies that the real power consumption can be lower of a 30% 40%. Simulations are performed at 100MHz.

Form the power estimation the static and dynamic currents are extrapolated:

$$I_{Stot} = P_{Stot} / V_{DD} = 8.659 \text{ } \mu\text{A}$$

$$I_{Dtot} = P_{Dtot} / V_{DD} = 0.885 \text{ A}$$

3.3.3 Area occupation

The number of transistors of the ADAPTO architecture can be estimated considering all the component that composed the system:

- Logic Blocks

- LB Context Memory
- Interconnect (pass transistor)
- Interconnect Decoder
- Context Memories Interconnect
- Memory decoders
- FLip FLOp chain
- Clock tree
- additional inverters used as buffers

The total number of transistors is:

$$\begin{aligned}
 NT_{TOT} &= N_{LB} + N_{MemLB} + N_{IC} + N_{DecIC} + N_{MemIC} + N_{DecAdd} + N_{FFchain} + \\
 N_{CLK} + N_{Buff} &= 96 \times 79 + 96 \times 552 + (96 \times 66 + 128 \times 34) + (96 \times 432 + 128 \times \\
 258) &+ (96 \times 942 + 128 \times 812) + 6 \times 120 + 912 \times 16 + 36 + 1280 = 356756
 \end{aligned}$$

This number match with the one obtained by the LVS check performed on the global layout of the entire integrated circuit. In the following are shown more details about number of transistors partition:

- LB= 7584
- Interconnect (decoder + pass transistor)= 85184
- Context memory LB : 53352
- Context memory interconnect: 194728
- Flip-flop chain + CLK tree= 14628

The 69.5% of the transistors is used for the Context Memories implementation. The total area occupation (WxH) is 5.606 x 2.086 mm for a total area of 11.694 mm^2 .

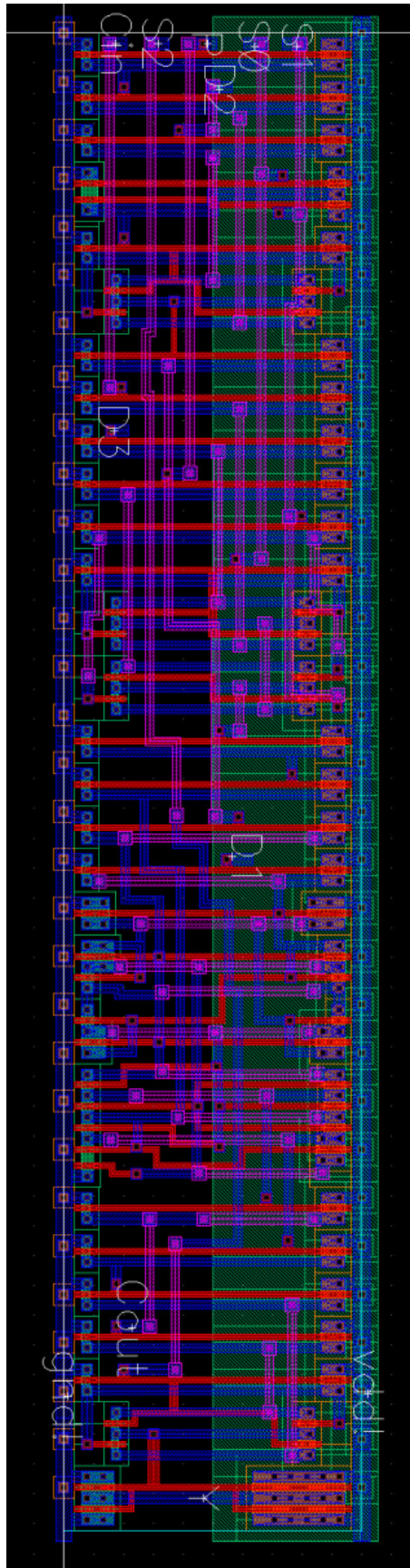


Fig. 3.15: The LB layout

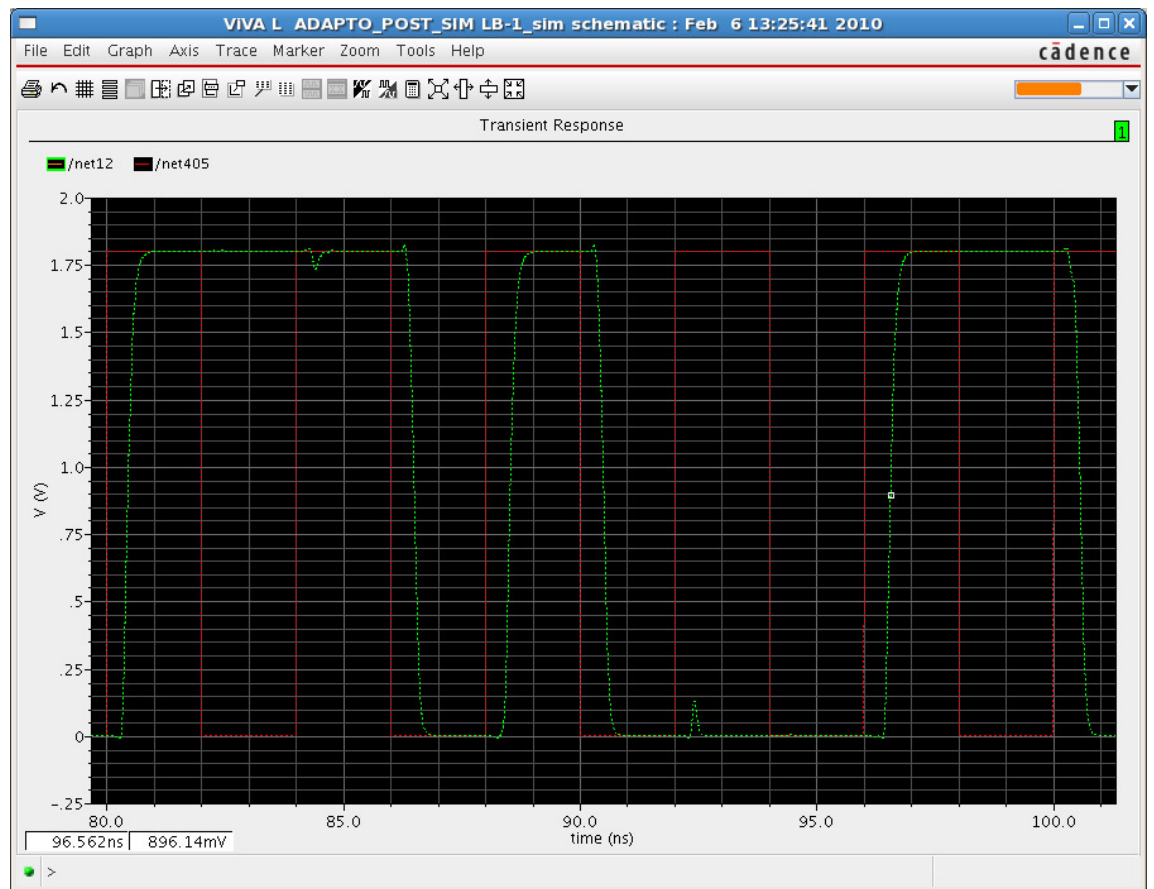


Fig. 3.16: LB delay time of the Cout pin

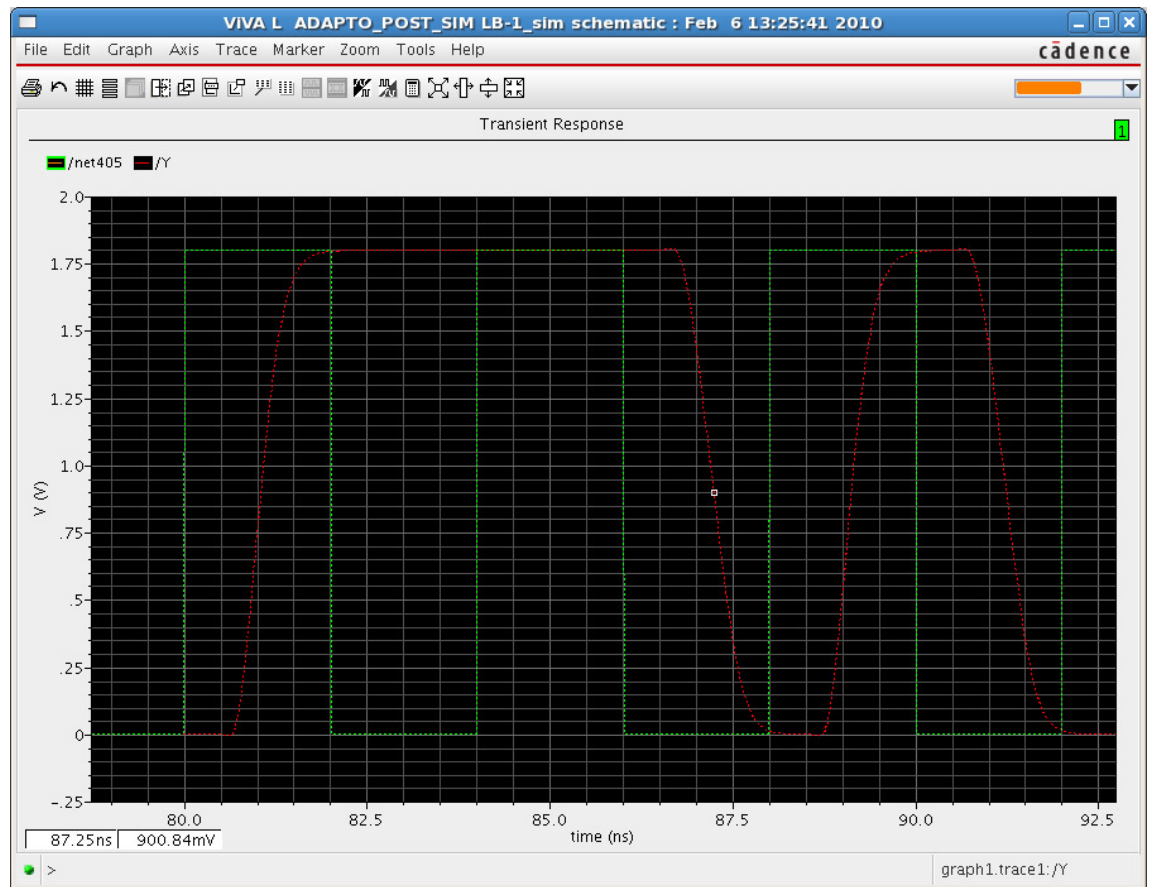


Fig. 3.17: LB delay time of the Y pin

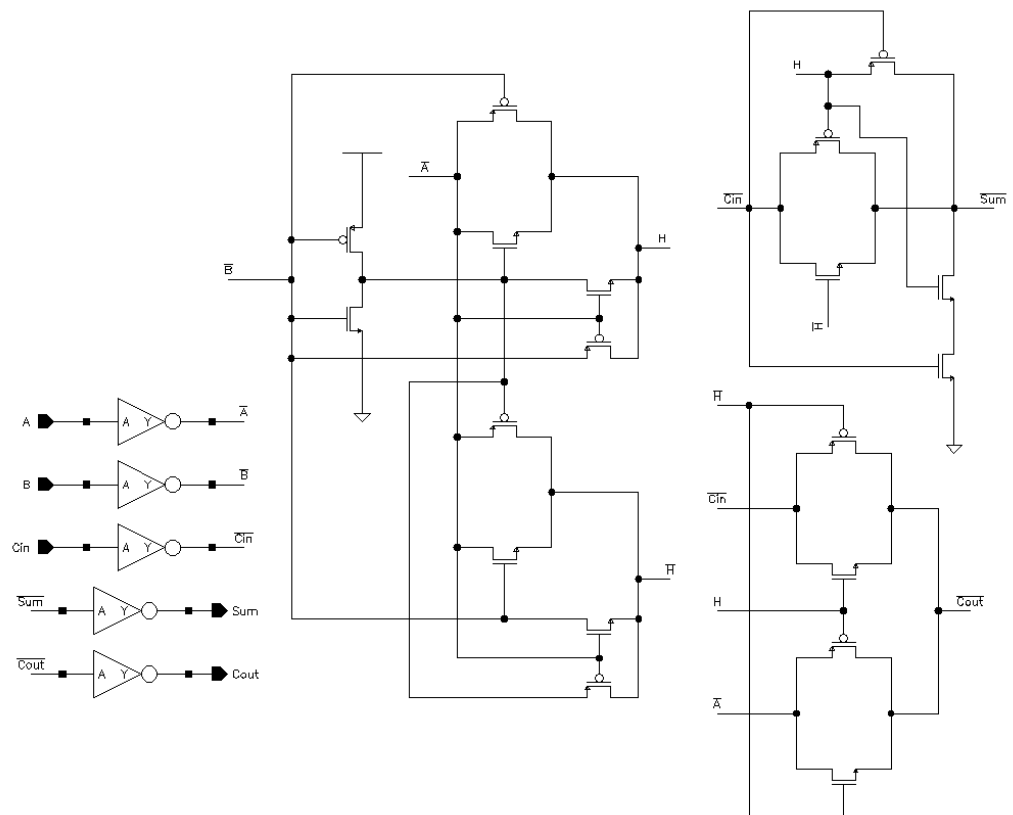


Fig. 3.18: Full Adder circuit

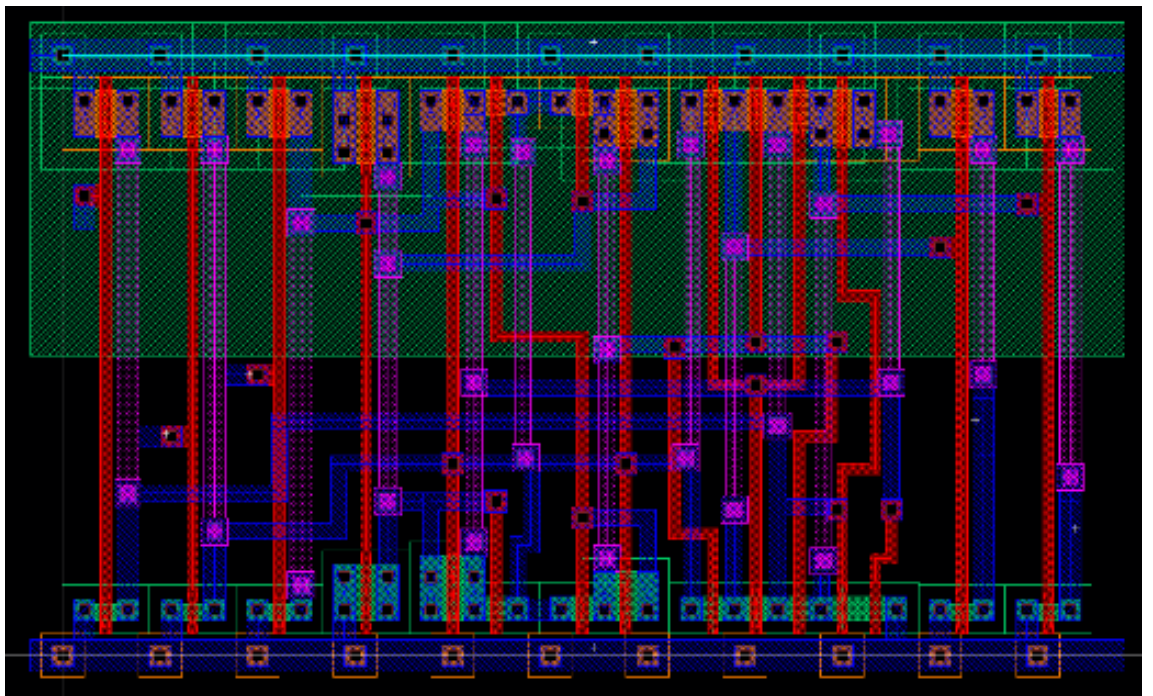


Fig. 3.19: Full Adder layout

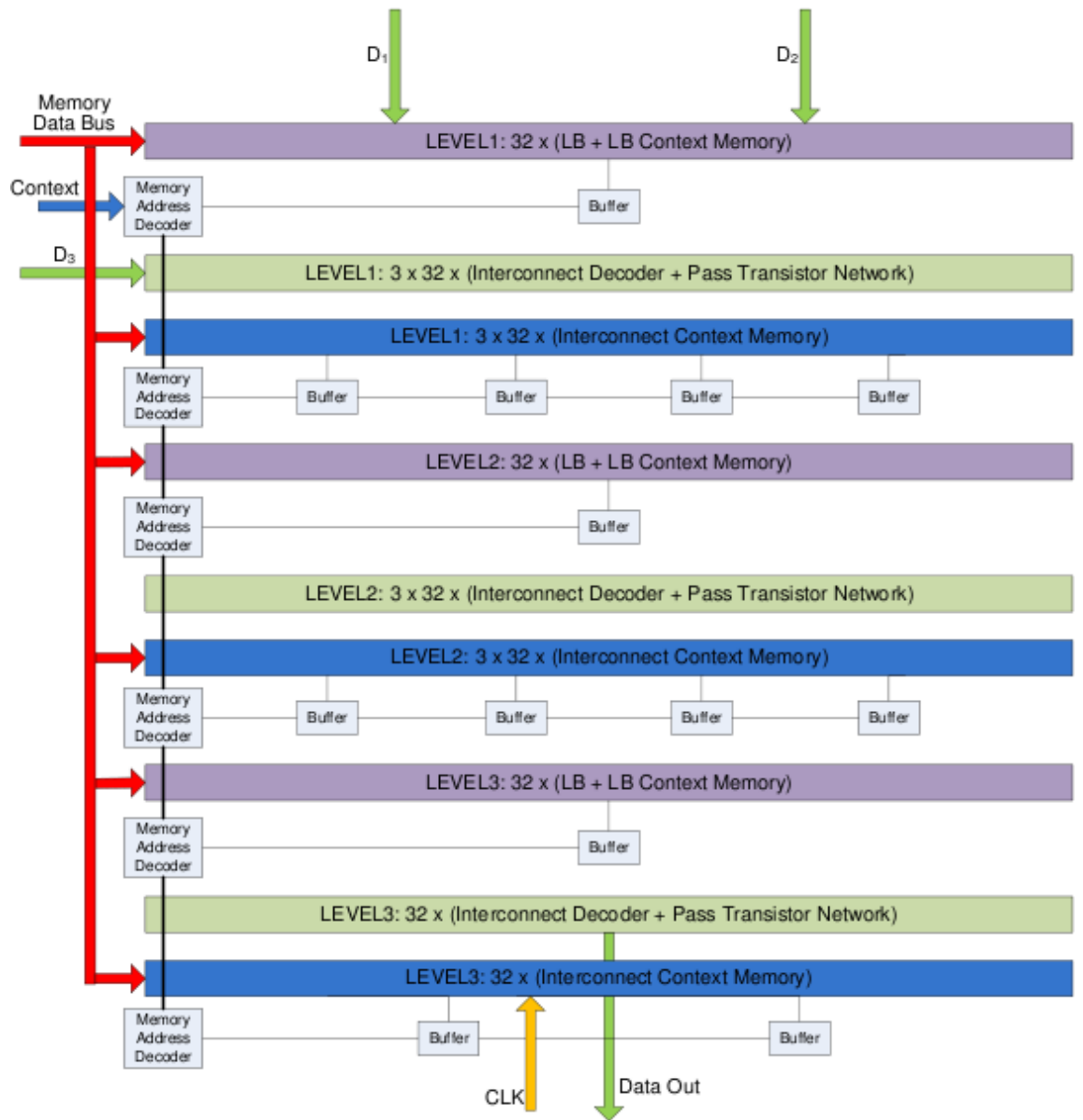


Fig. 3.20: ADAPTO architecture

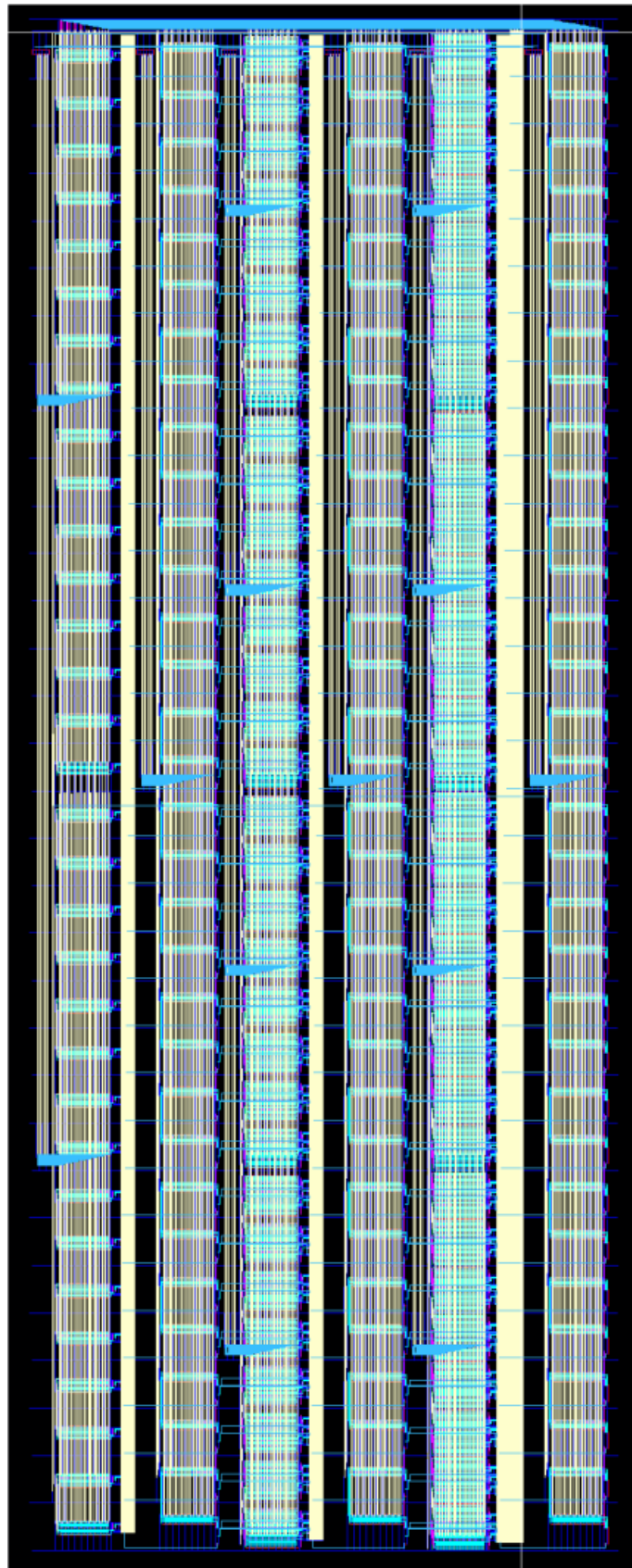


Fig. 3.21: ADAPTO Layout

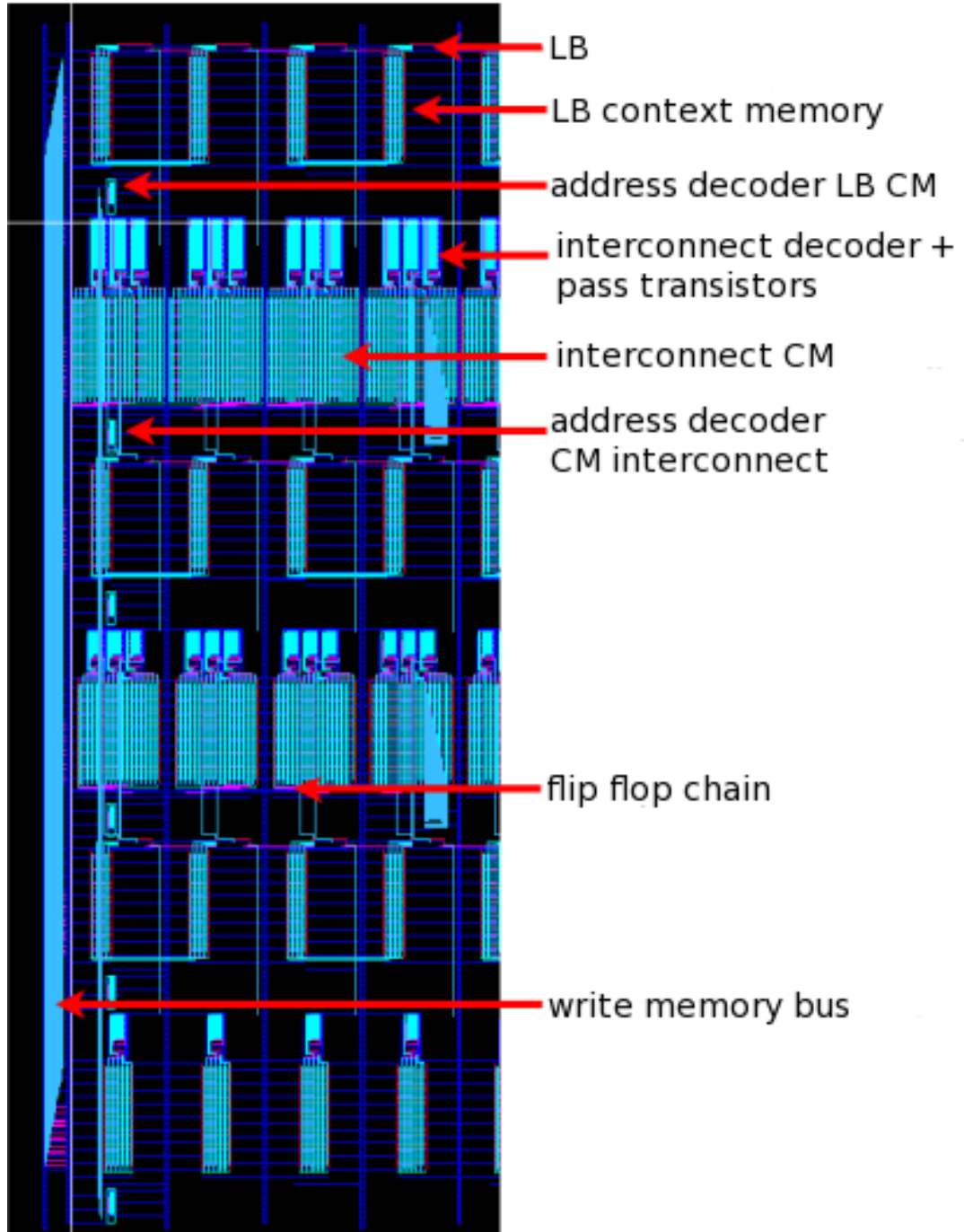


Fig. 3.22: ADAPTO Layout main blocks

Chapter 4

ADAPTO applications

4.1 Modular algebra

4.1.1 Modular Addition

Usually modular addition is defined between operands that are in the range $[0 \dots M-1]$ and obviously the result belongs to the same range. A simple way to perform this operation is to add X and Y and if the result is greater than M , M is subtracted from the result of the sum. This approach is illustrated below

Algorithm 1: Modular Addition

Input: $M, X < M, Y < M$

Output: $(X + Y) \bmod M$.

```
1:  $R \leftarrow X + Y$ 
2: if  $R \geq M$  then
3:    $R \leftarrow R - M$ 
4: end if
5: RETURN  $R$ 
```

This algorithm can be implemented both in software and in hardware. A typical

hardware implementation is reported in Fig. 4.1.

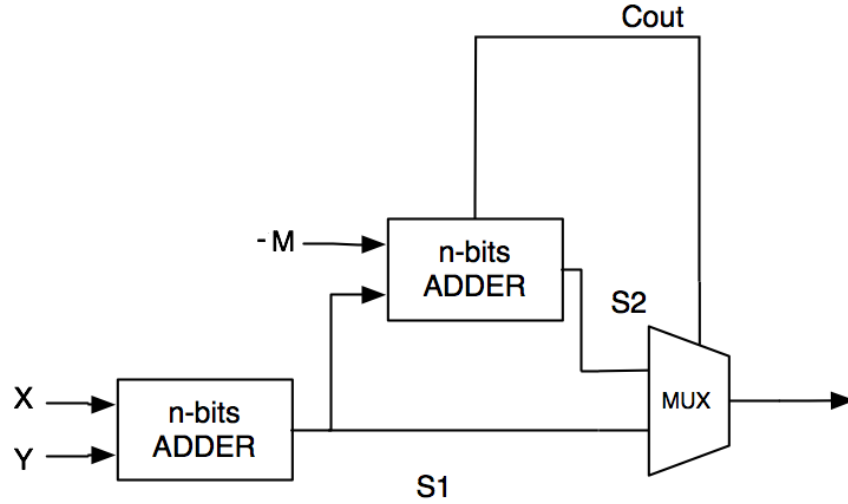


Fig. 4.1: Modulo M adder

In the hardware implementation the two values $S1 = X + Y$ and $S2 = X + Y - M$ are computed and the MUX selects between $S1$ and $S2$ depending on the value of C_{out} . In fact, C_{out} is the carry of $S2 = X + Y - M$ and therefore it is equal to 1 only if $X + Y - M < 0$. This condition corresponds to select $S1$ if $X + Y < M$, $S2$ otherwise. The **if** condition in Algorithm 1 correspond to the MUX in the hardware implementation shown in Fig. 4.1. Modular addition by using ADAPTO can be implemented in the following way. The first slice computes $S1 = X + Y$. The second slice computes $S2 = S1 - M$. It must be noticed that M is a constant value and therefore it is given as input to the adder by programming the first interconnect slice in order to provide $-M$ as the second input. The input corresponding to $-M$ is the binary representation of $2^{n+1} - M$, where n is the number of bits used to represent M . These two stages are identical to the adders presented in fig. 4.1. In the third stage the adder should select $S1$ and $S2$ as a function of C_{out} . Unfortunately, ADAPTO can not send to the third slice of adders both the results of the first and the second stage ($S1$ and $S2$) and therefore the MUX in fig. 4.1 cannot be directly

implemented. To supersede this limitation the third slice is programmed to perform the following operation:

```
if  $C_{out} = 1$  then  
     $R \leftarrow S2 + M$   
else  
     $R \leftarrow S2$   
end if
```

In this way the final result is $R = X + Y - M$ if $X + Y \geq M$, otherwise $R = (X + Y - M) + M = X + Y$. Now we take the binary representation of $M = m_{n-1} \dots m_0$ and compute the bitwise AND between the bits of M and C_{out} obtaining $CM = C_{out} \cdot m_{n-1} \dots C_{out} \cdot m_0$. The conditional operation described before can be performed by the third stage of ADAPTO as $R \leftarrow S2 + CM$. M is constant and therefore the operand CM correspond to give as second input of the i FA C_{out} if $m_i = 1$, zero if $m_i = 0$. In Fig. 4.2 the configuration of ADAPTO performing the modular addition is presented.

The adder performs addition modulo 45 (101101 in binary). The first stage is a standard adder. The second stage is an adder with a constant input ($-M$). We can see that in this stage the C_{Out} of the second stage is presented as input of the third stage by using a FA of the second stage as a routing element. In fact, ADAPTO do not allows to directly provide as output both the carry and the output of the FA, but only one of the two outputs. So, the C_{Out} is given as input by the next FA with $a_{n+1} = 0$ and $b_{n+1} = 0$ and the result of the addition is taken as output. In this way C_{Out} has been routed to the next stage. The use of a logic resource as a routing element is widely used in FPGA, in which the LUT is configured as pass-through, to save routing resources or to form a shortest path between to points of the FPGA. In our case the use of the FA is mandatory to route both the sum

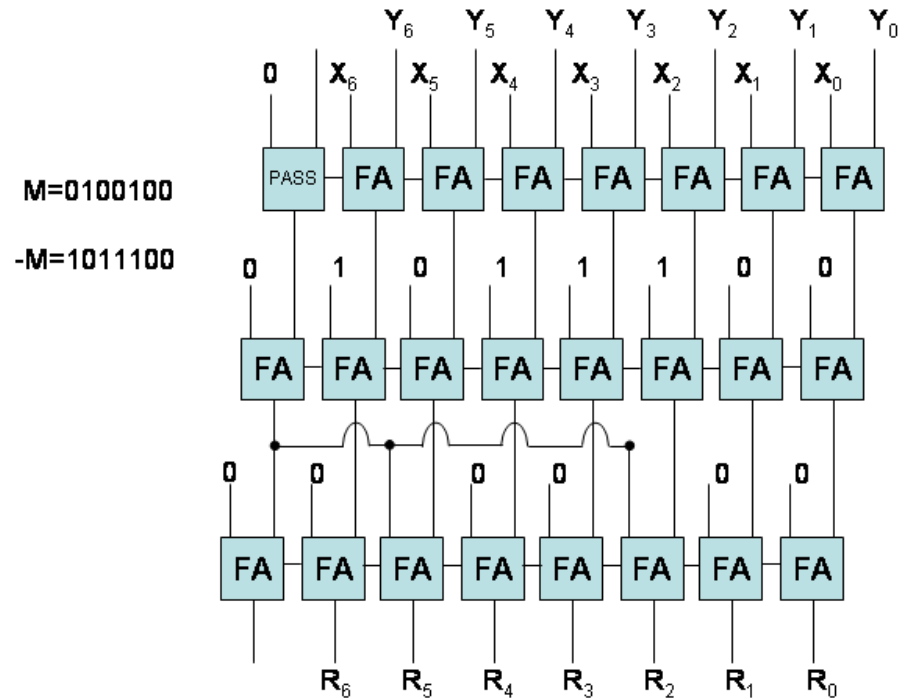


Fig. 4.2: Configuration of ADAPTO performing the modular addition

and the carry of a FA to the next stage. Instead, a modification of the interconnect matrix of the architecture of ADAPTO will require a doubling in the number of transistor needed to route both the outputs. The third stage is a standard adder that has as inputs $S2$, *i.e.* the result of the second stage, and CM . The final result is therefore the modular addition between the two inputs. We notice that the 32 bits width of ADAPTO allows performing different one modular additions in parallel. In particular a modulo that can be represented with n bits requires $n + 2$ columns of ADAPTO. One bit more than n allows the two complement representation of $-M$, while the other bit is lost for the FA used as pass-through. For example ADAPTO can be configured to implement four different modulo additions with moduli of 6 bits or three modular additions with two 9 bits moduli and one 8 bit modulus.

4.1.2 Montgomery Multiplication Algorithm

In [19] Peter Montgomery proposed a method for avoiding expensive reductions modulo M after multiplication modulo p . It uses the so-called Montgomery representation for integers. The Montgomery representation of an integer $a \in [0, M - 1]$ is $A \cdot Z^{-1} \bmod M$ where $Z > M$ such that $\gcd(Z, M) = 1$. The Montgomery multiplication is defined as $R = A \cdot B \cdot Z^{-1} \bmod M$ and its computation is particularly simple if $Z = 2^{-n}$, where n is the number of bits needed to represent M .

The Montgomery representation does not give any computational advantage for a single multiplication. Instead, when several multiplication are involved the Montgomery gives an advantage due to the few computing resources needed to perform Montgomery multiplication. Hence, Montgomery representation is useful in modular exponentiation, operations performed on ECC etc. In these complex operations the integers are firstly converted in the Montgomery representation, the computations are performed in the Montgomery domain, finally the result is reconverted in the traditional integer number representation.

The conversion between the traditional integer representation of a number A and the Montgomery representation is the Montgomery multiplication between A and 1, *i.e.*

$$R = A \cdot 1 \cdot Z^{-1} \bmod M = A \cdot Z^{-1} \bmod M$$

Instead, the reverse conversion is the Montgomery multiplication between A and Z^2 , *i.e.*

$$R = A \cdot Z^2 \cdot Z^{-1} \bmod M = A \cdot Z \bmod M$$

The algorithm that computes the Montgomery multiplication is (Algorithm 2)

Algorithm 2: Montgomery Multiplication

Input: $M, A < M, B < M, n$ **Output:** $A \cdot B \cdot 2^{-n} \bmod M$.

```
1:  $R \leftarrow 0$ 
2: for ( $i = 0; i < n; i++$ ) do
3:    $R \leftarrow R + B \cdot a_i$ 
4:   if  $R$  is odd then
5:      $R \leftarrow R + M$ 
6:   else
7:      $R \leftarrow R$ 
8:   end if
9:    $R \leftarrow R/2$ 
10: end for
11: RETURN  $R$ 
```

The core of the algorithm is represented by lines 3 to 9, *i.e.* the context of the **for** loop. We will show how to configure the ADAPTO engine to perform the operations inside the loop in one clock cycle. Similarly to the previous case, we rewrite the algorithm in order to avoid the use of the **if then** construct. The new version of the algorithm is illustrated below (Algorithm 3).

Algorithm 3: Montgomery Multiplication for ADAPTO

Input: $M, A < M, B < M, n$ **Output:** $A \cdot B \cdot 2^{-n} \bmod M$.

```
1:  $R \leftarrow 0$ 
2: for ( $i = 0; i < n; i++$ ) do
3:    $S1 \leftarrow B \cdot a_i$ 
4:    $S2 \leftarrow S1 + R$ 
5:    $S3 \leftarrow S2 + M \cdot S2[0]$ 
6:    $R \leftarrow S3/2$ 
7: end for
8: RETURN  $R$ 
```

Line 3 of the revised algorithm can be implemented by the first slice of ADAPTO simply by configuring the FAs as an AND between the bits of the operand B and the i bit of the operand A. The second slice of FAs uses the third input of ADAPTO in order to accumulate the partial results of the Montgomery operation. In particular the third input correspond to the partial result obtained at the end of the previous loop cycle. The third slice of adders implements the conditional sum expressed by lines 4-8 of Algorithm 2 in a way that is similar to the one used for the modular addition. The addition is performed by masking a constant value (the modulo M) with a bit corresponding to the control value of the conditional statement. In this case this bit is the least significant bit of $S2$ and allows identifying if $S2$ is even or odd. If $S2$ is odd M is added to $S2$, else $S2$ remains unchanged. Finally, the interconnect matrix after the third adder slice implements the right shift of the result $S3$, diving it by two. The output R of this operation is given to a register in the register file that will be provided as the third input of ADAPTO at the next iteration of the loop. At the end of the loop the result stored in R correspond to the result of the Montgomery multiplication. In Fig. 4.3 the configuration of ADAPTO

performing the Montgomery multiplication is presented.

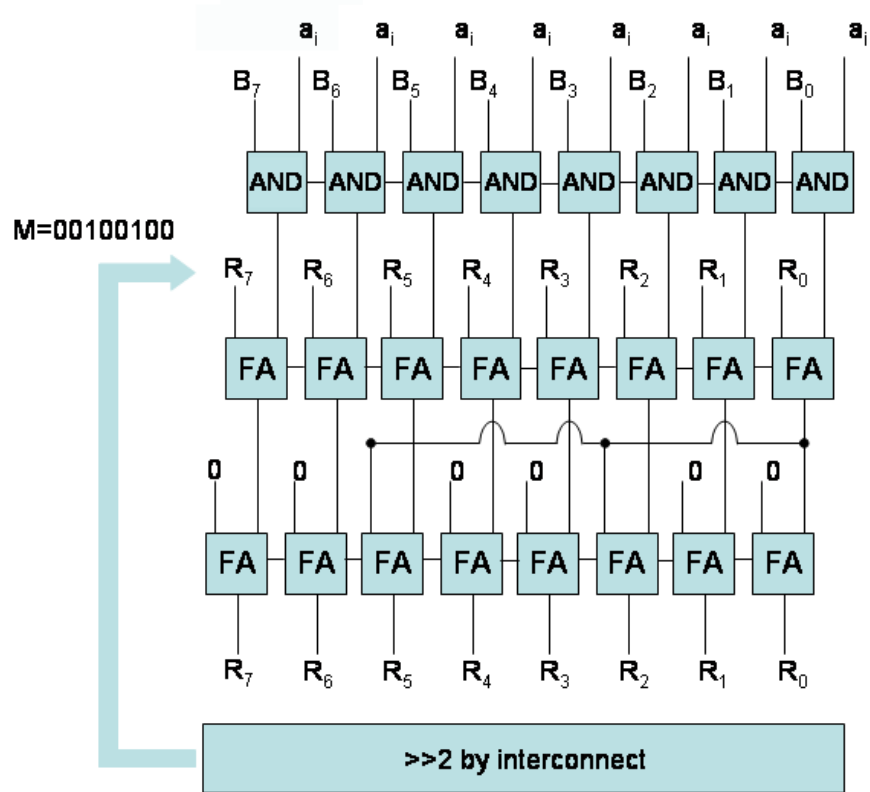


Fig. 4.3: ADAPTO performing a step of the MM

In Fig. 4.3 we can see the configuration of the first slice as an array of AND gates that takes as inputs the operand B and the bit a_i of the operand A. This slice performs line 3 of Algorithm 3. The second slice performs line 4 of algorithm 3, while the third slice performs the line 5 of Algorithm 3. We notice that, differently from the modular addition, the M value is masked by the least significant bit of the result of the previous slice (*i.e.* $S2[0]$). Finally, the last interconnect stage performs the division by 2 shifting right its input.

4.2 The AES algorithm

Galois Field of the form $GF(2^n)$, that are commonly used in cryptographic operations, such as in the Rijndael AES algorithm [24], can be efficiently manipulated by our ADAPTO architecture by several reasons:

- the basic computational element (the full adder) provide the XOR operation that is widely used for implementation of $GF(2^n)$.
- constant multiplication in $GF(2^n)$ can be easily performed by using a suitable logic and interconnection configuration as will be described in the paper.
- the word length of 32 bits of ADAPTO allows to perform 4 operations on the field $GF(2^8)$ in parallel.

The AES algorithm is designed to use only byte (8-bits) operations. The encryption of a data block is composed by an initial XOR step, several round transformations, and an final round different by the previous ones. In case of 128 bits block size, the data are arranged as 16 bytes and are ordered in a matrix format as follows:

$$\mathbf{I} = \begin{pmatrix} A & E & I & M \\ B & F & J & N \\ C & G & K & O \\ D & H & L & P \end{pmatrix} \quad (4.1)$$

The encryption of a 128 bits block size requires 9 rounds, composed by four transformations:

1. SubBytes: a non-linear substitution step where each byte is replaced with another according to a lookup table.
2. ShiftRows: a transposition step where each row of the state is shifted cyclically a certain number of steps.

3. MixColumns: a mixing operation which operates on the columns of the state, combining the four bytes in each column
4. AddRoundKey: the byte of the state is combined with the round key.

For the final step, the Mixcolumn transformation is not performed. For data decryption, however, different transformations are used:

1. InvSubBytes: the inverse of SubBytes,
2. InvShiftRows: the inverse of ShiftRows,
3. InvMixColumns: the inverse of MixColumns,
4. AddRoundKey: similar to encryption

The software implementation of AES encryption/decryption operations can easily perform the SubBytes and InvSubBytes operations by storing in a memory the corresponding lookup table. Since the SubBytes and InvSubBytes operations are performed on a 8bit data, each lookup table requires 256 bytes of memory. Therefore the software implementation of this operation is simple and efficient. In addition, the ShiftRows and InvShiftRows operations are simply byte reordering, therefore also this operation is very simple. Moreover, these operations can be merged with SubBytes and InvSubBytes operations and therefore the impact of these operation on the performance of the AES algorithm are negligible. Also the AddRoundKey is a simple operation: it is a two inputs 32-bit XOR operation. The only operations to be accelerated by an RFU are the couple MixColumns/InvMixColumns. Implementing MixColumns on a ARM926EJ-S RISC we notice that for a column computation are required about 50 assembly instruction. Therefore in the following we analyze in detail the Mixcolumn/InvMixcolumn transformations in order to implement these

operation by using ADAPTO. The MixColumn transformation is based on matrix multiplication in $GF(2^8)$.

The matrix \mathbf{I} is multiplied by the following matrix:

$$\mathbf{M} = \begin{pmatrix} 0x02 & 0x03 & 0x01 & 0x01 \\ 0x01 & 0x02 & 0x03 & 0x01 \\ 0x01 & 0x01 & 0x02 & 0x03 \\ 0x03 & 0x01 & 0x01 & 0x02 \end{pmatrix} \quad (4.2)$$

Each matrix elements (expressed in an hexadecimal notation) correspond to a polynomial of degree 7 in which the polynomial coefficients are the coefficient of the binary representation (*e.g.* $0x0A = 0b00001010 = x^3 + x$). We remark that constant multiplications are performed on $GF(2^8)$ and use $x^8 + x^4 + x^3 + x + 1$ as the generator polynomial. For the InvMixcolumn transformation all columns are multiplied by the inverse matrix of the one used in Mixcolumn:

$$\mathbf{M}^{-1} = \begin{pmatrix} 0x0E & 0x0B & 0x0D & 0x09 \\ 0x09 & 0x0E & 0x0B & 0x0D \\ 0x0D & 0x09 & 0x0E & 0x0B \\ 0x0B & 0x0D & 0x09 & 0x0E \end{pmatrix} \quad (4.3)$$

The multiplications of the matrix elements of InvMixcolumn are harder than the multiplications involved in Mixcolumn and therefore it is the more expensive task of all the AES algorithm. In the next section we show how these operation are implemented by using ADAPTO.

4.2.1 ADAPTO implementation

Data allocation strategy

The first step of the implementation of Mixcolumn and InvMixcolumn in ADAPTO is to define where and how the 16 input bytes defined in equation (1) are stored in the CPU registers. We decided to store the matrix by column, as presented in table I.

	Stored bytes			
Register/part	[31..24]	[23..16]	[15..8]	[7..0]
R1	A	B	C	D
R2	E	F	G	H
R3	I	J	K	L
R4	M	N	O	P

Table 4.1: RF allocation of AES matrix

4.2.2 $GF(2^8)$ constant multiplication in ADAPTO

In this subsection, $GF(2^8)$ constant multiplication is illustrated (GF multiplication corresponds to a conventional polynomial multiplication followed by a division by the polynomial generator). In order to illustrate the use of ADAPTO, we consider the following example. Take into account the multiplication of a generic 8-bit polynomial $P = p_7 \cdot x^7 + p_6 \cdot x^6 + p_5 \cdot x^5 + p_4 \cdot x^4 + p_3 \cdot x^3 + p_2 \cdot x^2 + p_1 \cdot x^1 + p_0$ by the constant polynomial $(x + 1)$, corresponding to the hexadecimal number 0x03. The operation $0x03 \cdot P$ in the GF gives $0x03 \cdot P = (p_7 \cdot x^7 + p_6 \cdot x^6 + p_5 \cdot x^5 + p_4 \cdot x^4 + p_3 \cdot x^3 + p_2 \cdot x^2 + p_1 \cdot x + p_0) + (p_6 \cdot x^7 + p_5 \cdot x^6 + p_4 \cdot x^5 + p_3 \cdot x^4 + p_2 \cdot x^3 + p_1 \cdot x^2 + p_0 \cdot x) + p_7 \cdot (x^4 + x^3 + x + 1)$. Shortly we can write

$$0x03 \cdot P = P + (P \ll 1) + p_7 \cdot (x^4 + x^3 + x + 1)$$

Figure 4.4 shows the ADAPTO implementation of the constant multiplication by 0x03. The shift operations, as the left shift $A \ll 1$ required in the above multiplication, are performed directly by the ADAPTO interconnection network. Also $a_7 \cdot (x^4 + x^3 + x + 1)$ can be implemented by the interconnection network. In fact, let us call $Z = z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0$ the byte representing the result of $a_7 \cdot (x^4 + x^3 + x + 1)$. We have $z_7 = z_6 = z_5 = z_2 = 0$ and $z_4 = z_3 = z_1 = z_0 = a_7$. Therefore the interconnect can compute Z by imposing some LB inputs to zero and connecting a_7 to the remaining LBs. The stripe following the interconnection is configured as a three input XOR, performing the required three additions on $GF(2^8)$.

Any constant multiplications that can be translated in a bit rearrangement and a sum of three terms can be performed with the above method. In our work any constant multiplication is computed using the set of basic multiplications shown in the Table II.

C	Implementation $C * \cdot P$
0x02	$(P \ll 1) + p_7 \cdot (x^4 + x^3 + x + 1)$.
0x03	$P + (P \ll 1) + p_7 \cdot (x^4 + x^3 + x + 1)$
0x04	$(P \ll 2) + p_7 \cdot (x^5 + x^4 + x^2 + x) +$ $+ p_6 \cdot (x^4 + x^3 + x + 1)$

Table 4.2: Basic constant multiplications

4.2.3 ADAPTO implementation of Mixcolumn

Using the above results, in this subsection we describe the implementation on ADAPTO of the multiplication of a row of \mathbf{M} by a column of \mathbf{I} . We suppose that each column of \mathbf{I} is stored in the RF, according to above discussed data allocation strategy. D1,

D2, D3 are the ADAPTO input operands coming from the RF, and R is the final result that will be returned to the RF. Moreover α , β , and γ are the partial results present at the output of the three LB stripes of ADAPTO.

As an example, we consider the multiplication of the first row of \mathbf{M} by the first column of \mathbf{I} .

Algorithm 1 describes the implementation of this multiplication. The first stripe of ADAPTO is configured as a pass-thru and connects directly A, B, C, D (output α) to the first level of interconnect. The constant multiplications $0x02 \cdot A$, and $0x03 \cdot B$, with the sum $C + D$ are computed by the interconnect and the second LB stripe (output β). It must be noticed that this mixed operations require 24 LBs configured as three inputs XOR, while the eight rightmost LBs are unused. In the algorithm we represent these unused output as don't care '-'. Starting β , the last LB stripe, configured as three input XOR, provides the final mod. 8 sum γ .

Algorithm 4: Multiplication 1st row of \mathbf{M} by the 1st data column

Input: D1(A,B, C, D); D2=(-.-.-); D3=(-.-.-)

Output: $R[7 : 0] = 0x02 \cdot A + 0x03 \cdot B + C + D$.

$\alpha \leftarrow (A, B, C, D)$

$\beta \leftarrow (0x02 \cdot A, 0x03 \cdot B, C + D, -)$

$\gamma \leftarrow (-, -, -, 0x02 \cdot A + 0x03 \cdot B + C + D)$

RETURN $R[7 : 0] \leftarrow \gamma[7 : 0]$

The product of the same column by a different row requires the reconfiguration of ADAPTO. Therefore the entire MixColumn operation requires four different ADAPTO contexts (of the 16 available in the current architecture). However, the computation of the product of different columns by the same row is computed by using the same context.

By using **Algorithm 1** each row by column multiplication requires 1 ADAPTO instruction. Consequently the whole *MixColumn* operation is computed in 16 assem-

bly instructions (corresponding to 16 clock cycles, since one clock cycle is required for each ADAPTO context [5], [6]). In Fig. 4.4 we show the implementation of the constant multiplication by $0x03$ using a row of ADAPTO. This implementation has been compared with a *Mixcolumns* software implementation present in the benchmark suite described in [25]. This function, compiled on a **ARM926EJ-S RISC**, architecture requires about 200 assembly instructions. Thus the speed-up obtained with ADAPTO is about 12.5x

4.2.4 ADAPTO implementation of InvMixcolumn

The InvMixColumn operation $\mathbf{I} \times \mathbf{M}^{-1}$ is more complex due to the structure of the entries of the matrix \mathbf{M}^{-1} (here \mathbf{I} is different from that used in Mixcolumn). To simplify the computation, the constant coefficient of multiplications are expressed in terms of the elementary constants of Table II. Table III is shows the decomposition of the multiplications by complex constants.

C	Decomposition of $C \cdot P$
$0x08$	$0x02 \cdot (0x04 \cdot P)$
$0x09$	$0x08 \cdot P + 0x01 \cdot P$
$0x0B$	$0x08 \cdot P + 0x03 \cdot P$
$0x0C$	$0x03 \cdot (0x04 \cdot P)$
$0x0D$	$0x0C \cdot P + 0x01 \cdot P$
$0x0E$	$0x0C \cdot P + 0x02 \cdot P$

Table 4.3: Decomposition of complex constants

The decomposition of Table III can require more contexts for the computation of a constant multiplication. For example, the multiplication by $0x0E$ is performed in two phases (corresponding to 2 ADAPTO contexts). For the multiplication of first

row of \mathbf{M}^{-1} by the first column of data matrix \mathbf{I} we decompose the computation it in two terms

$$\begin{aligned} R &= 0x0E \cdot A + 0x0B \cdot B + 0x0D \cdot C + 0x09 \cdot D = \\ &= (0x0C \cdot A + 0x08 \cdot B + 0x0C \cdot C + 0x08 \cdot D) + \\ &\quad (0x02 \cdot A + 0x03 \cdot B + 0x01 \cdot C + 0x01 \cdot D) \end{aligned}$$

The result R is evaluated in two phases. In the first phase, the first partial results are computed by ADAPTO as constant multiplications of A, B, C, D by $0x04$, (corresponding to α) followed by four multiplications by $0x03$ or $0x02$ (γ computation). These operations are implemented in the first ADAPTO context. In the second phase the input $D1$ contains A, B, C, D , and the inputs $D2$ and $D3$ store the results of the previous phase. We use the ADAPTO inputs $D1$ and $D2$ to compute $0x0C \cdot C + C$ and $0x08 \cdot D + D$. Instead $D3$ is used to input the previous results to the input of the second stripe of LB . So we compute the three terms $0x0C \cdot A + 0x02 \cdot A$, $0x03 \cdot B$ and $0x08 \cdot B + 0x0D \cdot C + 0x09 \cdot D$. The third strip of the ADAPTO sums (XOR) these three terms. The two phases are shown in **Algorithm 2**.

The two constants of **Algorithm 2** are valid until the row of \mathbf{M}^{-1} is unchanged. When we go to another row the constants change and two other contexts must be used. Consequently, the computation of the whole *InvMixColumn* in principle requires 8 contexts. Some simplification can be carried out observing the properties of the entries of \mathbf{M}^{-1} . For example, PHASE 1 can be shared between the first and third rows, and between the second and the fourth rows. In fact the first and the third rows have the common term $(0x0C \cdot A + 0x08 \cdot B + 0x0C \cdot C + 0x08 \cdot D)$, while the second and the fourth rows have $(0x08 \cdot A + 0x0C \cdot B + 0x08 \cdot C + 0x0C \cdot D)$. This property allows to reduce the number of contexts and number of time the contexts must be reconfigured. Therefore the computation of the complete output matrix

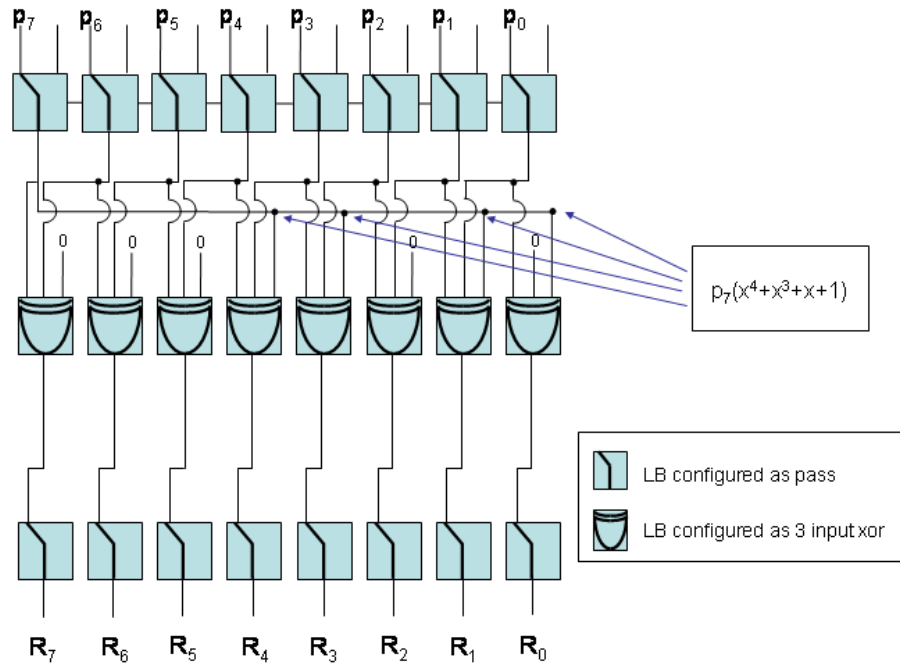


Fig. 4.4: Implementation of $0x03 \cdot B$ in ADAPTO

requires 16 runs of PHASE 2 and 8 runs PHASE 1, for a total of 24 ADAPTO reconfigurations corresponding to 24 assembly instructions. On the other hand, the software implementation of *InvMixColumn* requires again 200 instructions. So, a speed-up of about 8.3x is obtained.

Algorithm 5: multiplication of a row of the matrix \mathbf{M}^{-1} by a column

Input: $D1(A,B, C, D)$; $D2=(-,-,-,-)$; $D3=(-,-,-,-)$

Output: $R[7 : 0] = 0x0E \cdot A + 0x0B \cdot B + 0x0D \cdot C + 0x09 \cdot D$.

PHASE 1 (context 1)

$\alpha \leftarrow (A, B, C, D)$

$\beta \leftarrow (0x04 \cdot A, 0x04 \cdot B, 0x04 \cdot C, 0x04 \cdot D)$

$\gamma \leftarrow (0x0C \cdot A, 0x08 \cdot B, 0x0C \cdot C, 0x08 \cdot D)$

PHASE 2 (context 2)

$D1 \leftarrow (A, B, C, D)$; $D2 \leftarrow \gamma$; $D3 \leftarrow \gamma$;

$\alpha \leftarrow (A, B, 0x0D \cdot C, 0x09 \cdot D)$

$\beta \leftarrow (0x0E \cdot A, 0x03 \cdot B, 0x08 \cdot B + 0x0D \cdot C + 0x09 \cdot D, -)$

$\gamma \leftarrow (-, -, -, 0x0E \cdot A + 0x0B \cdot B + 0x0D \cdot C + 0x09 \cdot D)$

Chapter 5

ADAPTO in Leon 2 processor

After the ADAPTO design and before starting with the IC realization, the ADAPTO unit has been integrated in a soft processors in order to evaluate the speedup factor. For our purpose we choose the open source LEON 2 soft processor. Unfortunately integrate ADAPTO as RFU in this processor is impossible without modify the VHDL description of the microprocessor. However the Leon 2 processor provides a coprocessor interface suitable for general purpose coprocessor so the ADAPTO unit was integrated as coprocessor. Using this strategy two considerations must be made:

- The coprocessor approach is slower in terms of acceleration respect than the RFU approach as show in Chapter. 2
- In order to integrate ADAPTO on the LEON 2 coprocessor interface we have to use only two of the three inputs.

Considered these issues we can say that the obtained experimental results are worse than the potential ones. Additional informations can be found in [32].

5.1 LEON-2 processor

The LOEN-2 processor used in our experiment is a VHDL model of a 32-bit processor conforming to the IEEE-1754 (SPARC-V8) architecture. It is designed for embedded applications and it is characterized by having separate instruction and data cache. A block diagram of LEON-2 is shown in Fig.5.1.

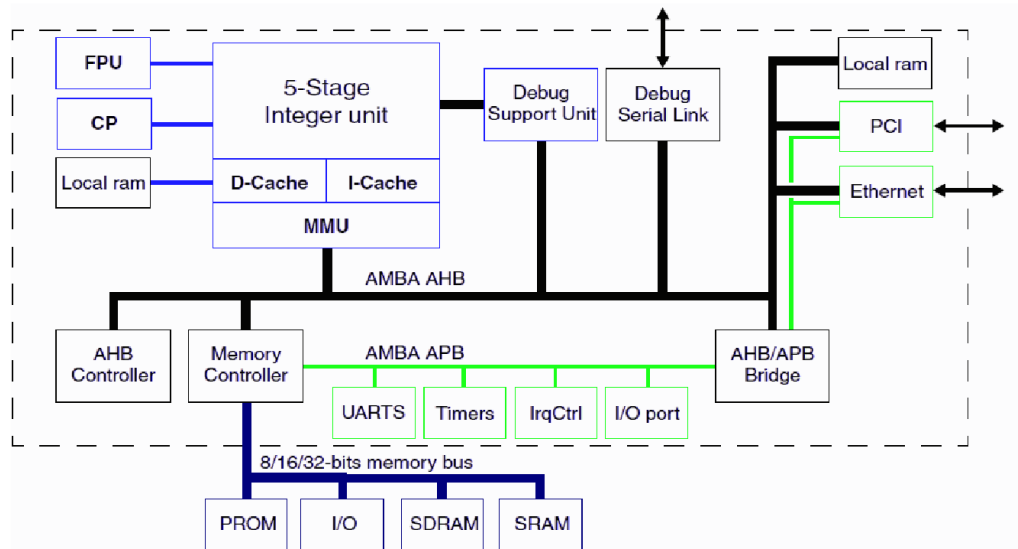


Fig. 5.1: LEON-2 block diagram

The LEON-2 processor can be configured to provide a generic interface to a special-purpose co-processor in which we attach the ADAPTO unit. One co-processor instruction can be started each cycle as long as there are no data dependencies. When finished, the result is written back to the co-processor register file. As shown in Fig.5.2, the LEON-2 architecture has two separated set of register file (configurable up to 32) for processor integer unit (IU) and co-processor. In order to exchange data between this two computational element an assembly "load" instruction is required; this means that at least three clock cycles are required for data exchange.

For our experiment we use a simplified version of ADAPTO having only two input, because LEON 2 provide a two-input one-output co-processor interconnect as

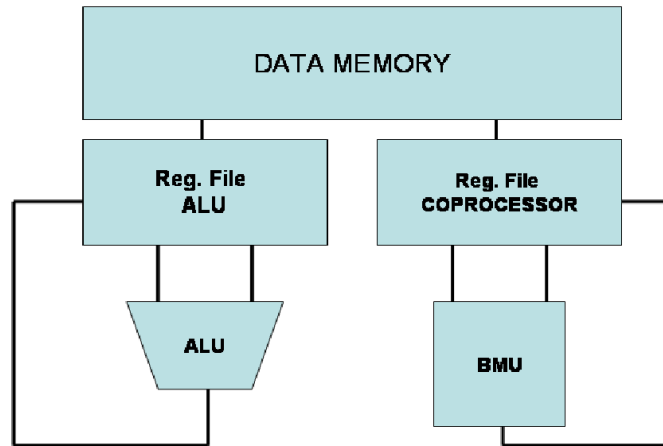


Fig. 5.2: Co-Processor interface

shown in Fig.5.2.

5.2 Experimental results

To test the ADAPTO-LEON-2 system performances we used TSIM2 Professional, a LEON-2 simulator that allow to add a shared object (written in C language and compiled with Bare Cross Compiler - BCC) that emulate ADAPTO behavior. So we were able to simulate the system with and without the shared object and highlight the differences in term of executed instructions number. There are various applications where is possible to verify algorithm acceleration, but we have chosen four applications: Bit reversal, dist1 function for MPEG-2 encoding, GRP instruction, big/little endian conversion. All source codes of our experiments was compiled with follow BCC options:

```
sparc-elf-gcc -msoft-float -g -O2 -mv8 source.c -o file.exe
```

The option *-msoft-float* indicates the absence in our simulation of a real Floating Point Unit so it was emulated by the processor, while the option *-mv8* indicates that multiplications simulation occur with use of a real hardware multiplier based on Sparc V8 architecture.

To avoid changing LEON-2 instruction set assembler (ISA), the ADAPTO instructions has been added by assembly commands. In particular in each experiment there are *load* and *store* instructions to move data from LEON-2 register file to co-processor (ADAPTO) register file and was created a *cpopx* macro which defines the instruction carried out by ADAPTO. There are three arguments for *cpopx* instruction which specify the two input operands, the single output and the code representing the particular ADAPTO operation. Below, will be shown the results of simulation.

5.2.1 Bit reversal permutation

Bit reversal is an operation that invert bit position representing an integer number (see Fig. 5.3). It's used especially in Fast Fourier Trasformate (FFT).









Original		Reversed	
DEC	BIN	Inversion	BIN DEC
0	= 000		000 = 0
1	= 001		100 = 4
2	= 010		010 = 2
3	= 011		110 = 6
4	= 100		001 = 1
5	= 101		101 = 5
6	= 110		011 = 6
7	= 111		111 = 7

Fig. 5.3: Bit-reversal logic flow

To perform bit reversal we have chosen one of most used algorithm based on

shift and boolean operations. The function C code is shown in Algorithm 6 and its execution block diagram in 5.3.

Algorithm 6: Source code used for bit reversal implementation without ADAPTO

```
[fontsize=\relsize{-0.5}]  
  
unsigned char shift(unsigned char x){  
    x = (x & 0x0f) <<4 | (x & 0xf0) >> 4;  
    x = (x & 0x33) <<2 | (x & 0xcc) >> 2;  
    x = (x & 0x55) <<1 | (x & 0xaa) >> 1;  
    return x;}
```

To compute Algorithm 6 LEON-2 without ADAPTO uses 15 instructions instead with ADAPTO the instructions used are only 3: a '*ld*' instruction to load the value in co-processor register file, a co-processor instruction ('*cpopx*') to compute the bit reversal operation and a '*st*' instruction to store the result in LEON-2 register file.

5.2.2 MPEG-2 encoding

MPEG2 encoding is a part of MediaBench benchmark suite, this benchmark regards compression and video processing. About 89% of the execution time for MPEG2 encoding is spent for the computation of *dist1* function. So, we decided to consider this function for our acceleration experiments. The Algorithm 7 show C code for dist1 encoding implementation. LEON-2 processor to compute one step cycle uses 32 instructions instead the same algorithm with use of ADAPTO is computed with 19 instructions.

In this case, we need two '*ld*' instructions (see Algorithm 8) because the two input operands representing the firsts values of two arrays.

Algorithm 7: Source code used for dist1 function without ADAPTO

```
[fontsize=\relsize{-0.5}]
for(temp=0; temp < (lenght- 1); temp++){
v = p1[temp] + p1[temp+1] + 1;
v = v >> 1;
v = v- p2[temp];
if( v >= 0)
s += v;
else
s -= v;}
return (s);
```

Algorithm 8: Source code used for dist1 function with ADAPTO

```
[fontsize=\relsize{-0.5}]
for(temp=0; temp < (lenght- 1); temp++){
k = p1[temp] + p1[temp+1];
z = p2[temp];
asm("ld %0, %%c00" : : "m" (k) );
asm("ld %0, %%c01" : : "m" (z) );
asm(cpopx("0x12", "0x0", "0x1", "0x2"));
asm("st %%c02, [%0]" : "=g" (v) );}
```

5.2.3 GRP instruction

GRP (see [28],[29] and [30]) is a primitive bit permutation instruction that gathers to the right the data bits selected by 1s in a mask (r_3), and to the left those selected by 0s in the same mask (see Fig. 5.4). An arbitrary bit permutations can be accomplished by a sequence of at most $\log_2(n)$ of these grp instructions, where n is the word-size of the processor (32 bit in LEON-2 case).

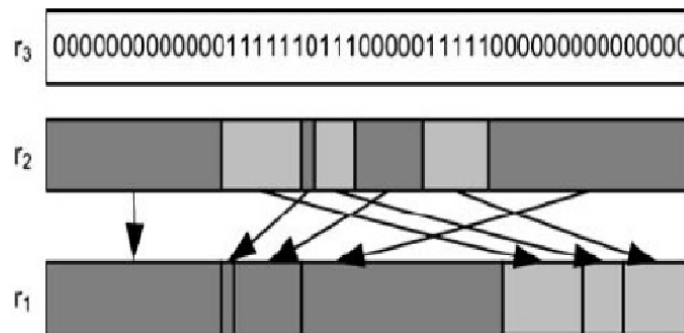


Fig. 5.4: GRP bit permutation

The C code used to compute this instruction is shown in algorithm 9.

To execute this code, LEON-2 processor uses 34 assembly instructions, instead with ADAPTO the instructions used are only 4 how shown in Algorithm 10.

So, with ADAPTO it's possible to compute a generic bits permutation only with 4 instructions than $\log_2(n)$ used by LEON-2 or most general purpose processors.

5.2.4 Endian coversion

The endianness is the byte ordering used to represent data flow. In our case we considered the conversion between big and little endian, that is very used in telecommunications field. The conversion is symmetrical, i.e. it's used the same C function to convert from little to big endian and vice versa. The C code used for the conversion is shown Algorithm 11

Algorithm 9: C-code of grp instruction without ADAPTO

[fontsize=\relsize{-1}]

```
int grp(int mask, int reg_in, int reg_out,
        int tmp1, int tmp2, int num_bit){
while (tmp1 < num_bit)
    {if(((mask >> tmp1) & 1) == 1)
        {reg_out=reg_out | (((reg_in >> tmp1) &1)<<tmp2);
        tmp1++; tmp2++;}
    else {tmp1++;}}
tmp1 = 0;
while (tmp1 < num_bit)
    {if(((mask >> tmp1) & 1) == 0)
        reg_out=reg_out | (((reg_in >> tmp1) &1)<<tmp2);
        tmp1++; tmp2++;}
    else {tmp1++;}}
return (reg_out);}
```

Algorithm 10: Assembly code for grp instruction with ADAPTO

[fontsize=\relsize{-0.5}]

```
asm("ld %0, %%c00" : : "m" ((int)mask_new) );
asm("ld %0, %%c01" : : "m" ((int)reg_in_new));
asm(cpopx("0x11", "0x0", "0x1", "0x2"));
asm("st %%c02, %0" : "=m" (*y) );
```

Algorithm 11: C code used by LEON-2 for endian conversion

```
[fontsize=\relsize{-0.5}]

unsigned long End_Conv(unsigned long data)

    return (((data>>24)& 0x000000FF) |
            ((data>>8) & 0x0000FF00) |
            ((data<<8) & 0x00FF0000) |
            ((data<<24)& 0xFF000000));
```

To execute a conversion of a 32 bit word data LEON-2 processor uses 12 assembly instructions, while the use of ADAPTO increases significantly the performances because the same conversion is computed only with 3 instructions.

Chapter 6

Future works

In Chapter.5 the experimental results about the integration of the ADAPTO unit in the LEON 2 processor have been shown. As discussed in this chapter, the LEON 2 processor does not support the integration of a RFU so the ADAPO unit has been integrated as coprocessor (see 1) using the LEON 2 coprocessor interface. In order to overcome this aspect actually we are working on the integration of the ADAPTO Unit in the NIOS II soft processor. We choose this microprocessor because it has a very interesting feature called "Custom Logic" (Chapter 6.1.1). This feature permit the integration of a RFU without the microprocessor redesign because a custom operator in parallel with the ALU is supported.

6.1 The NIOS II processor

The NIOS-II embedded processor [36] is a general-purpose RISC processor core produced by Altera. The main features of NIOS-II are:

- Full 32-bit instruction set, data path and address space
- 32 general-purpose registers
- Optional shadow registers sets

- Single-instruction 32 x 32 multiply and divide unit producing a 32-bit result
- Dedicated instructions for computing 64-bit and 128-bit product of multiplication
- Floating point instructions for single precision floating point operations
- Single instruction barrel shifter
- Hardware assisted debug module enabling processor start, stop step and trace under control of the Nios-II software development tools
- Optional memory management unit (MMU) to support operating systems requiring MMUs
- Software development environment based on the GNU C/C++ tool chain and the Nios-II Software Build Tools for Eclipse
- Instruction set architecture (ISA) compatible across all Nios-II processor systems
- Performance up to 250 DMIPS

A Nios-II processor (Fig. 6.1) system consists of a Nios-II processor core, a set of on-chip peripherals, an on-chip memory and some interfaces to off-chip memory. All these blocks are implemented on a single Altera device. The version of Nios-II used in our experiments is 9.1, the last available in Altera's library.

6.1.1 Custom Logic Custom Instructions

One of the most important feature of Nios-II processor is the possibility to add user-defined Functional Units called Custom Logic [37]. Custom instructions allow the possibilities to tailor the Nios-II processor core to meet the needs of a particular

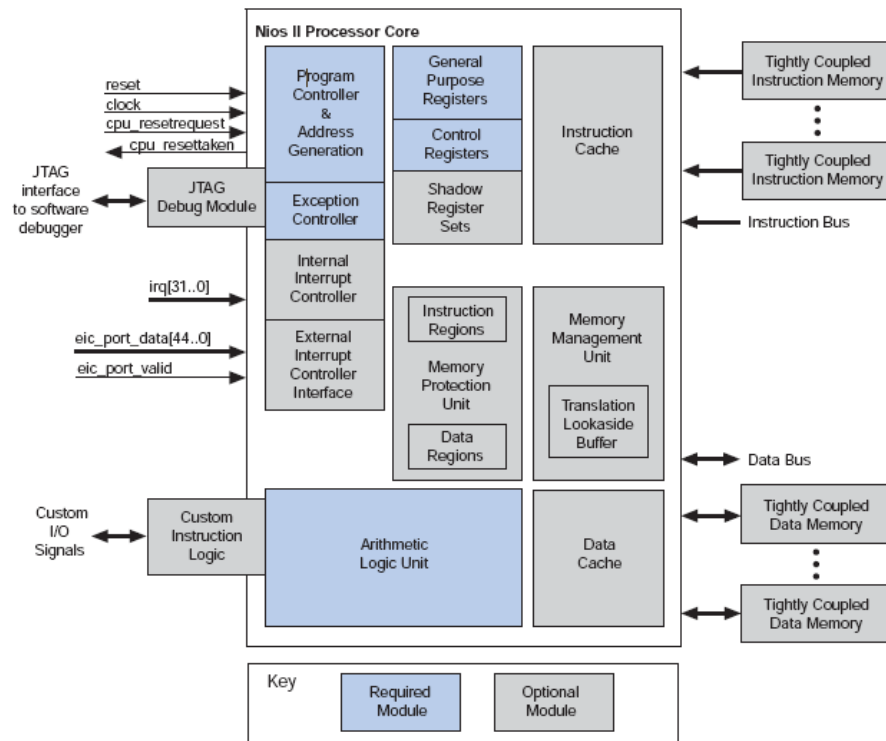


Fig. 6.1: NIOS II block diagram

application. In this way it is possible to accelerate time critical software algorithms by implementing some steps into specialized hardware blocks. These blocks must be created using either VHDL or Verilog language. Physically, the Custom Logic blocks is placed inside the Nios-II processor in parallel to the ALU as shown in Fig. 6.2

The basic operation of Nios-II custom instruction logic is to receive input on the `dataa` and/or `datab` port, and drive out the result on its result port. The Nios-II processor supports different types of custom instructions. Fig. 5 lists the additional ports that accommodate different custom instruction types. Only the ports used for the specific custom instruction implementation are required. To manage Custom Logic, the Nios-II system also provides the Custom Instructions. For each custom instruction, the Nios-II Integrated Development Environment (IDE) generates a macro in the system header file, `system.h`. These macro can be called using

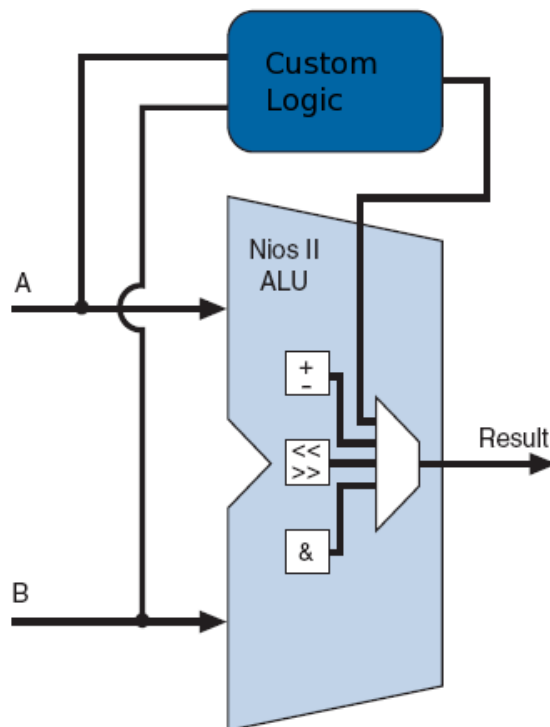


Fig. 6.2: NIOS II Custom Logic

the C-functions defined in main program. This is a big advantage, because the programmers don't need to understand assembly language to use custom instruction. Similar to native Nios-II instructions, custom instructions can take values from up to two source registers and optionally writes back a result to a destination register. The algorithm 1 shows an example of C-function based on a built-in macro.

```

/* opcode to select ADAPTO operation */
#define ADAPTO 0x2

/* adapto Butterfly user-defined functions */
#define ADAPTO(data1 data2,data3,op)
__builtin_custom_inii(ADAPTO,data1 data2,data3,op);

```

6.1.2 ADAPTO integration

As discussed in Chapter 3, the ADAPTO architecture has three 32 inputs, however the NIOS II “Custom Logic“ interface present two input bus. This problem can be solved using a state register as shown in Fig. 6.3. In this way, if the third operand is required (this operand is not used every time), an additional operation is required to store the required value in the state register. Once the data has been stored the three input function can be performed by ADAPTO. This strategy assures the possibility to use all the three inputs but surely add a latency to the computation. On the other hand this is the only solution possible to solve this problem if the processor has not a three output Register File.

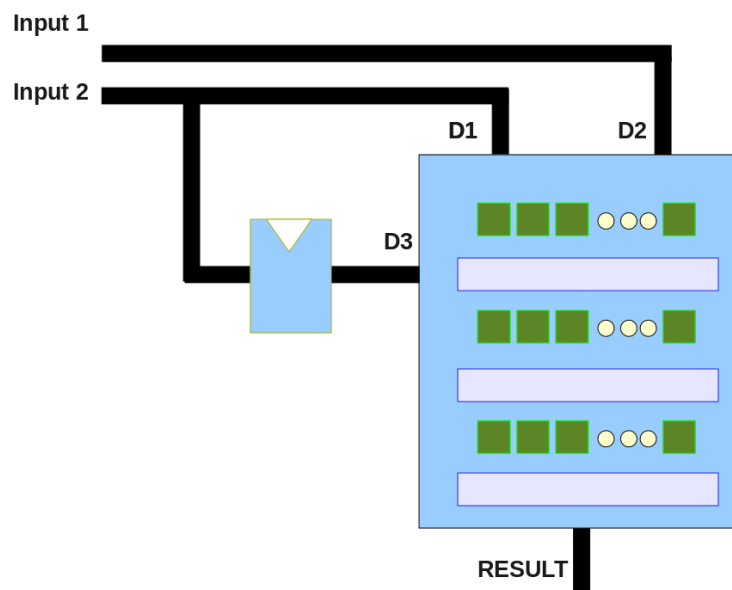


Fig. 6.3: Adapto integration in NIOS II

Conclusions

In this work a new Reconfigurable Functional Unit (RFU) is presented. This architecture can be used by embedded microprocessors to accelerate short data manipulation. As discussed in Chapter 2, in order to have an adequate acceleration, fast reconfiguration is a very critical aspect. Actual reconfigurable systems obtain this feature using the multicontext approach. The actual multicontext reconfigurable arrays are based on multicontext LUTs that required an area occupation directly proportional to the context number. This solution can be used for very expensive FPGA where the cost is not an important aspect but it cannot be used for low cost embedded world.

The integration of a very expensive multicontext architecture in a Low cost embedded microprocessor would go in contrast with the low cost requirement. In order to solve this problem in this work a new multicontext approach has been proposed; our solution provides to use a different computational element respect the LUT. This choice implies less flexibility respect than the conventional LUT but this lack can be tolerated for a set of applications.

The proposed architecture can be used to speed up short data/bit level operations based on boolean expression or short addition but also for packing/unpacking data. Speeding up experiments was performed integrating the ADAPTO unit in the LEON II processor. This soft processor doesn't provide an RFU interface so we integrate ADAPTO as coprocessor using the LEON 2 coprocessor interface. Nevertheless this

limitation the experimental results show a considerable acceleration factor.

Actually we are working on the integration between ADAPTO and the NIOS 2 processor, this soft processor unlike the LEON 2 has a “Custom Logic“ interface that allows an easy integration of an RFU (Chapter. 6).

Simultaneously the integration experiments, the ADAPTO IC layout was realized using the *TSMC0.18 m* technology, The simulations are performed using 1.8 V for the Voltage and 27C for the temperature. The ADAPTO IC has a maximum frequency about 100 MHz, compatible with low cost embedded processors.

List of publications

1. G. C. Cardarilli, L. Di Nunzio, R. Fazzolari, M. Re “Algorithm acceleration on LEON-2 processor using a Reconfigurable Bit Manipulation Unit” 2010 8th IEEE Workshop on Intelligent Solutions in Embedded Systems
2. G. C. Cardarilli, L. Di Nunzio, M. Re “Speed-Up of RISC Processor Computation Using ADAPTO” 2009 IEEE International Symposium on Circuits and Systems
3. G. C. Cardarilli, L. Di Nunzio, M. Re “Arithmetic/Logic Blocks for Fine-Grained Reconfigurable Units” 2009 IEEE International Symposium on Circuits and Systems
4. G. C. Cardarilli, L. Di Nunzio, M. Re “High Performance Reconfigurable blocks for real-time reconfigurable unit (ADAPTO)” 2008 ReCoSoc
5. G. C. Cardarilli, L. Di Nunzio, M. Re “A full-adder based reconfigurable architecture for fine grain applications: ADAPTO” 2008 IEEE International Conference on Electronics, Circuits, and Systems
6. G. C. Cardarilli, L. Di Nunzio, M. Re, A. Nannarelli “ADAPTO: Full-Adder Based Reconfigurable Architecture for Bit Level” 2008 IEEE International Symposium on Circuits and Systems

List of Figures

1.1	FPGA architecture	12
1.2	LUT based Logic Block (LB)	13
1.3	LUT implementation of boolean expression	13
1.4	FPGA programmable interconnect	14
1.5	Single context	14
1.6	Multi context	15
1.7	4 input N context LUT	16
1.8	Partial programming	17
2.1	Possible integrations between processor and reconfigurable unit	20
2.2	The PRISC architecture	21
2.3	The CHIMAERA reconfigurable Unit	22
2.4	The CHIMAERA reconfigurable array	23
2.5	Multicontext LUT	26
2.6	Resource sharing	27
3.1	The ADAPTO architecture	30
3.2	The ADAPTO Logic Block	31
3.3	Full adder's table of truth	32
3.4	A multicontext ROW	32
3.5	LUT based LB	34

LIST OF FIGURES

3.6	Interconnect network	35
3.7	Interconnect reduction	35
3.8	Context memories Bus	37
3.9	Context Memories configuration	37
3.10	gate delay time	38
3.11	Setup time	39
3.12	Hold time	39
3.13	The LB symbol	40
3.14	The LB schematic	42
3.15	The LB layout	47
3.16	LB delay time of the Cout pin	48
3.17	LB delay time of the Y pin	49
3.18	Full Adder circuit	50
3.19	Full Adder layout	51
3.20	ADAPTO architecture	52
3.21	ADAPTO Layout	53
3.22	ADAPTO Layout main blocks	54
4.1	Modulo M adder	56
4.2	Configuration of ADAPTO performing the modular addition	58
4.3	ADAPTO performing a step of the MM	62
4.4	Implementation of $0x03 \cdot B$ in ADAPTO	71
5.1	LEON-2 block diagram	74
5.2	Co-Processor interface	75
5.3	Bit-reversal logic flow	76
5.4	GRP bit permutation	79

LIST OF FIGURES

6.1	NIOS II block diagram	84
6.2	NIOS II Custom Logic	85
6.3	Adapto integration in NIOS II	86

List of Tables

3.1	Full adder's operations	33
3.2	LB size, dimension are expressed in micron	41
3.3	LB Pin capacity	41
3.4	LB delay	41
3.5	LB power consumption	41
3.6	Full Addee size, dimension are expressed in micron	42
3.7	FA Pin capacity	42
3.8	FA delay	43
3.9	FA Power consumption	43
4.1	RF allocation of AES matrix	66
4.2	Basic constant multiplications	67
4.3	Decomposition of complex constants	69

List of Algorithms

1	Modular Addition	55
2	Montgomery Multiplication	60
3	Montgomery Multiplication for ADAPTO	61
4	Multiplication 1 st row of \mathbf{M} by the 1 st data column	68
5	multiplication of a row of the matrix \mathbf{M}^{-1} by a column	72
6	Source code used for bit reversal implementation without ADAPTO	77
7	Source code used for dist1 function without ADAPTO	78
8	Source code used for dist1 function with ADAPTO	78
9	C-code of grp instruction without ADAPTO	80
10	Assembly code for grp instruction with ADAPTO	80
11	C code used by LEON-2 for endian conversion	81

Bibliography

- [1] M. D. Razdan, R. Smith, "A high-performance microarchitecture with hardware programmable functional units", Proc. of MICRO-27, Nov. 1994, pp. 172-180
- [2] M. D. Razdan, R. Brace, K. Smith, "PRISC software acceleration techniques", Proc. IEEE 1994 Intl. Conf. on Computer Design: VLSI in Computer & Processors, pp. 145-149.
- [3] Analog Devices Blackfin Processor http://www.analog.com/en/embedded-processing-dsp/blackfin/content/blackfin_core_basics/fca.html
- [4] Bengu Li, Rajiv Gupta, "Bit Section Instruction Set Extension of ARM for Embedded Applications", International Conference on Compilers, Architecture, and Synthesis of Embedded Systems (CASES) 2002
- [5] Gian Carlo Cardarilli, Luca Di Nunzio, Marco Re, "High Performance Reconfigurable blocks for real-time reconfigurable unit (ADAPTO)", 2008 ReCoSoc.
- [6] Gian Carlo Cardarilli, Luca Di Nunzio, Marco Re, "A full-adder based reconfigurable architecture for fine grain applications: ADAPTO", 2008 IEEE International Conference on Electronics, Circuits, and Systems

- [7] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, Jeffrey P. Kao, "The Chimaera Reconfigurable Functional Unit", IEEE Trans. on VLSI Systems Vol. 12, Issue 2, Feb. 2004, pp. 206-217.
- [8] G. Borriello, C. Ebeling, S. Hauck, S. Burns "The Triptych FPGA Architecture", IEEE Transactions on VLSI Systems, Vol. 3, No. 4, pp. 491-501, December, 1995.
- [9] Data Book, San Jose, CA: Altera Corp.1995.
- [10] Schmit, H. Whelihan, D. Tsai, A. Moe, M. Levine, B. Reed Taylor, R. *PipeRench: A virtualized programmable datapath in 0.18 micron technology* , Custom Integrated Circuits Conference, 2002. Proceedings of the IEEE 2002.
- [11] H. Schmit "Incremental Reconfiguration for Pipelined Applications", IEEE Symposium on FPGAs for Custom Computing Machines.
- [12] <http://www.xilinx.com>
- [13] Scott Hauck Katherine Compton "An Introduction to Reconfigurable Computing", IEEE Computer, Apr, 2000.
- [14] S. Trimberger. "A time multiplexed FPGA", FCCM 97 Proceedings, pages 22-28, 1997.
- [15] W. Chong, S. Ogata, M. Hariyama and M. Kameyama "Architecture of a Multi-Context FPGA Using Reconfigurable Context Memory", Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 05)
- [16] Clive Maxfield "The design warrior's guide to FPGAs: devices, tools and flows"
- [17] Christian Lenci "CMos implementation of a Reconfigurable Funcional unit" internal report.

- [18] Ahmed M. Shams, Tarek K. Darwish, Magdy A. Bayoumi "Performance Analysis of Low-Power 1-Bit CMOS Full Adder Cells", IEEE transaction on very large scale integration (VLSI) system, VOL. 10, NO. 1, february 2002
- [19] P.L. Montgomery, "Modular multiplication without trial division", Mathematics of Computation, vol. 44, no. 170, Apr. 1985, pp. 519-521.
- [20] A.J. Menezes, and P.C. Van Oorschot, S.A. Vanstone, "Handbook of applied cryptography", 1997, CRC press
- [21] W.W. Peterson, and EJ Weldon, "Error-correcting codes", 1972, The MIT Press
- [22] ED Di Claudio, F. Piazza, G. Orlandi, "Fast combinatorial RNS processors for DSP applications", IEEE transactions on computers, vol. 44, no. 5, pages 624–633, 1995
- [23] Gian Carlo Cardarilli, Luca Di Nunzio, Salvatore Pontarelli, Marco Re, Adelio Salsano, "A Reconfigurable Functional Unit For Modular Operations", submitted to 2009 IEEE Reconfig
- [24] Joan Daemen and Vincent Rijmen, "The Design of Rijndael: AES - The Advanced Encryption Standard." Springer-Verlag, 2002.
- [25] M.. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown "MiBench: A free, commercially representative embedded benchmark suite"
- [26] S. Mamidi, E.R. Blem, M.J. Schulte, J. Glossner, D. Iancu, A. Iancu, M. Moudgill, S. Jinturkar, "Instruction set extensions for software defined radio on a multithreaded processor", ACM Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems, pages 266–273, 2005.

- [27] Ashruf, R. and Gaydadjiev, G. and Vassiliadis, S. and NL, D.E.T.T.U.D., “Reconfigurable implementation for the AES algorithm”, Proceedings of ProRISC 2002, the 13rd Annual Workshop on Circuits, Systems and Signal Processing
- [28] Yedidya Hilewitz, Ruby B. Lee, *Fast Bit Gather, Bit Scatter and Bit Permutation Instructions for Commodity Microprocessors* Springer Science + Business Media, LLC. Manufactured in The United States 2008
- [29] Zhijie Shi, Ruby B. Lee, *Bit Permutation Instructions for Accelerating Software Cryptography* Department of Electrical Engineering, Princeton University 2000
- [30] Yedisya Hilewitz Zhijie Shi, Ruby B. Lee, *Comparing Fast Implementation of Bit Permutation Instructions* Annual Asilomar Conference on Signals ,Systems and Computers.
- [31] Gaisler web site:
[http : //www.gaisler.com/](http://www.gaisler.com/)
- [32] R.Fazzolari ”Studio e realizzazione di un sistema integrato processore LEON-2 / Bit Manipulation Unit riconfigurabile e analisi delle prestazioni” internal report.
- [33] Gian Carlo Cardarilli, Luca Di Nunzio, Marco Re, ”Arithmetic/Logic Blocks for Fine-Grained Reconfigurable Units”, 2009 IEEE International Symposium on Circuits and Systems
- [34] M. Vesterbacka, ”A 14-transistor CMOS full adder with full voltageswing nodes”, SiPS 99, IEEE Workshop on Signal Processing Systems, pp.713-722.
- [35] A.A. Fayed, M.A.Bayoumi, ”A low power 10-transistor full adder cell for embedded architectures”, Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE Intl. Symposium

BIBLIOGRAPHY

[36] Altera web site:

[http : //www.altera.com/products/ip/processors/nios2/ni2 – index.html](http://www.altera.com/products/ip/processors/nios2/ni2-index.html)

[37] "Nios II Custom Instruction User Guide" ALTERA web site:

[http : //www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf](http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf)