
TECHNICAL UNIVERSITY OF CRETE
Electronics and Computer Engineering Department

Microprocessor and Hardware Laboratory


Diploma Thesis

# *"Extension of the Leon2 processor for more effective execution of cryptography algorithms"*


Spanos Nikolaos

**Advisor**
Assistant Prof. Ioannis Papaeystathiou
**Examination Committee**
Assistant Prof. Ioannis Papaeystathiou
Prof. Apostolos Dollas
Associate Prof. Dionisis Pnevmatikatos

# Acknowledgements

First of all, I would like to thank my advisor assistant Prof. Ioannis Papaeystathiou for all his support and useful advices during this project, the other two members of the examination committee Prof. Apostolos Dollas and associate Prof. Dionisis Pneymatikatos.

Also, from this point, I would like to thank all my friends with who, through these 6 years in Chania, I have shared the best moments of my life so far.
Zotos Alexandros, Skoyrtis Vasilis, Livanos Giorgos, Malafouris Zafeiris, Kagarakis Leonidas, Karakasiliotis Kostas, Koyfidakis Manolis.

To my family

# Table of contents

# Introduction

In a increasingly connected world, information security has become a top priority. Many applications — electronic mail, electronic banking, medical databases, and electronic commerce — require the exchange of private information. For example, when engaging in electronic commerce, customers provide credit card numbers when purchasing products. If the connection is not secure, an attacker can easily obtain this sensitive data. For this reason protocols have been designed that create secure connections and protect data transmission from malicious internet users.

Cryptography is a Greek word that literally means the art of writing secrets. In practice, cryptography is the task of transforming information into a form that is incomprehensible, but at the same time allows the intended recipient to retrieve the original information using a secret key. Cryptographic algorithms (or *ciphers*, as they are often called) are special programs designed to protect sensitive information on public communications networks. During *encryption*, ciphers transform the original message (called *plaintext*) in such a way as to hide its substance. An encrypted message is *ciphertext*. *Decryption* is the process of retrieving plaintext from ciphertext [5].

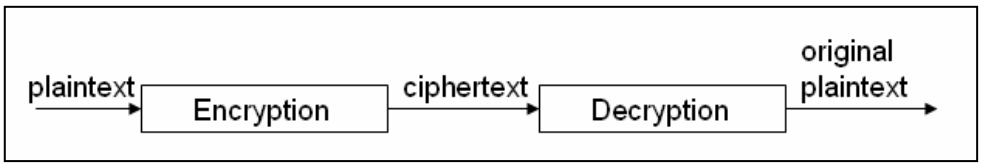



Figure 1.1: Encryption and Decryption

Two forms of cryptography are commonly used in information systems today: secret-key ciphers and public-key ciphers. Secret-key ciphers (sometimes referred to as symmetric-key ciphers) use a single private key to encrypt and decrypt. Public key ciphers (or asymmetric-key ciphers) use a well-known public key to encrypt and require a different private key to decrypt. Symmetric-key algorithms tend to be significantly faster than public-key algorithms. So, during a secure data exchange, at first a private key is shared between the users with a public key cipher and then all

other data are being transmitted with a private key cipher, that uses the previous shared key for encryption/decryption.

Nowadays, there is the demand of greater data encryption and decryption rates. To achieve this goal, ciphers have been implemented with software routines, directly in hardware or a combination of both. A software only approach is the lowest-cost solution but with accordingly low performance. The advantages of a software implementation include ease of use, ease of upgrade, ease of design, portability, and flexibility. However, a software implementation offers only limited physical security, especially with respect to key storage [11]. Conversely, cryptographic algorithms that are implemented in hardware are by nature more physically secure as they cannot easily be read or modified by an outside attacker when the key is stored in special memory internal to the device [12]. As a result, the attacker does not have easy access to the key storage area and cannot discover or alter its value in a straightforward manner [11].

When using a general-purpose processor, even the fastest software implementations of block ciphers cannot satisfy the required data encryption rates for high-end applications. As a result, hardware implementations are necessary for block ciphers to achieve this required performance level. Although traditional hardware implementations lack flexibility, configurable hardware devices offer a promising alternative for the implementation of processors via the use of IP cores in Application Specific Integrated Circuit (*ASIC*) and Field Programmable Gate Array (*FPGA*) technology. In this thesis, our goal is the enhancement of an existing Instruction Set Architecture (ISA) of the free LEON2 processor core with new instructions that will help faster processing of the enciphering and deciphering many of today's symmetric-key ciphers.

The extended ISA contains of instructions related with the integer unit (rotate-double), control transfer instructions (loop), and memory elements called Sboxes in cryptographic parlance. In order to realize how we incorporated these instructions (shown in Chapter 3), we should primarily understand the way the leon2 processor works (Chapter 2). Before that, what follows is a brief overview of previous work regarding implementations via software, hybric architectures cryptographic co-

processors, and instruction set extensions (Chapter 1). Finally in chapter 4 we show the verification process that was followed by running simulation tests, as well as the resource utilization and maximum frequency results.

# 1. Related work

This chapter focuses on related work that has been done in software and hardware level. The flexibility offered by software, as it stated before, is often not enough due to the small performance evidenced when targeting a general purpose processor whose instruction set cannot provide a fast and efficient implementation. In contrast there are many hardware specific implementations, based on FPGA devices or ASICs. This implementation category provides ultra speed performance (much higher than in software) for each symmetric algorithm, because of the dedicated hardware processors. One of the fastest implementations is presented in [18], Alireza Hodjat use a Virtex II Pro FPGA [19] and achieves a 21.4 Gbits/sec throughput of the AES algorithm.

We mainly focused to a category of designs that either extend an existing processor's architecture or introduce new co-processors specifically for the efficient execution of symmetric ciphers. Instruction Set Extensions (ISE) result in significant performance improvements versus traditional software implementations with considerable reduced logic resource requirements versus hardware-only solutions.

The most recent of these designs [13], presents a general purpose instruction set extension to a 32-bit SPARC V8 compatible processor core that accelerates the performance of Galois Field fixed field constant multiplication. This design improves the existing ISA, while maintaining a generalized implementation format capable of supporting other algorithms that use Galois Field fixed field constant multiplication.

Burke et al in [14]0 are trying to improve the performance of symmetric ciphers for the Alpha 21264 processor by examining eight algorithms. After analysis of bottleneck in these ciphers, they conclude to an extended ISA that consists of hardware rotations, modulo multiplication, permutation and Sbox access instructions and may achieve up to a 74% speedup over the baseline machine

Murat Fiskiran et al in [15] study the effect of different addressing modes that can be used to calculate the effective address during Sbox access. More specifically they determinate how performance is affected on 1, 2, 4 and 8 wide EPIC (Explicitly Parallel Instruction Computer) processors depending on addressing mode of the architecture, issue width of the processor and number of memory ports. The results indicate that speedups exceeding 2x can be obtained when fast addressing modes are used.

Another similar approach comes from [16], where the same authors describe a new hardware module called PTLU (Parallel Table Look Up). It consists of multiple LUTs that can be accessed in parallel and its purpose is again Sbox access acceleration. Their results show maximum speedups of 7.7x for AES and 5.4x for DES, all tested on a single-issue 64-bit RISC processor.

Hardware co-processors have been developed to accelerate cryptographic algorithm implementations. The CryptoManiac VLIW co-processor [20] was developed as a result of instruction set extensions designed to accelerate the performance of a number of the AES candidate algorithms. CryptoManiac uses an sbox instruction to read its four 1kB on-chip caches in order to improve the table lookups functions. Furthermore it features the execution of up to four instructions per cycle and the use of instructions with up to three operands to allow for the combination of short latency instructions for single cycle execution.

# 2. Leon2 Architecture

In this chapter we focus on describing Leon's architecture. Leon2 has been chosen because it is one of the most developed free processor cores available and it may be implemented on a variety of hardware solutions. We will concentrate on explaining the main functions of Leon and give extra attention to the components that are important for the specific project. The rest of the chapter is organized as follows: in the next section we provide a brief introduction to the LEON2 processor, and section 2.2 presents with more detail the instructions that are supported, their format and the way pipelining works.

## 2.1 Leon2 Microprocessor

LEON2 is a microprocessor which implements a RISC architecture conforming to the SPARC v8 definition [1]. It is a synthesizable core written in VHDL and can be implemented both on FPGAs and ASICs. It is distributed under the terms of the GNU LGPL license so it is an open hardware [2] and it is specifically designed for embedded applications. It was originally developed by the European Space Agency and nowadays it is maintained by Gaisler Research. The Leon2 32-bit core implements the full SPARC v8 standard, it uses big-endian byte ordering, has 32-bit internal registers, 72 different instructions in 3 different instruction formats and 3 addressing modes (immediate, displacement and indexed). It implements signed and unsigned multiply, divide and MAC operations and has a 5-stage instruction pipeline (Instruction Fetch, Decode, Execute, Memory and Write). It also implements two separate instruction and data cache interfaces, Harvard Architecture [3].

The VHDL model is fully synthesizable with most of the commonly synthesis tools, it is configurable and it uses the AMBA-2.0 AHB/APB onchip buses [4]. All these features makes Leon2 an ideal microprocessor for System-on-Chip applications. A block diagram of Leon2 architecture can be seen in figure 2.1. Many of those blocks are optional and can be removed from the model our application implements.

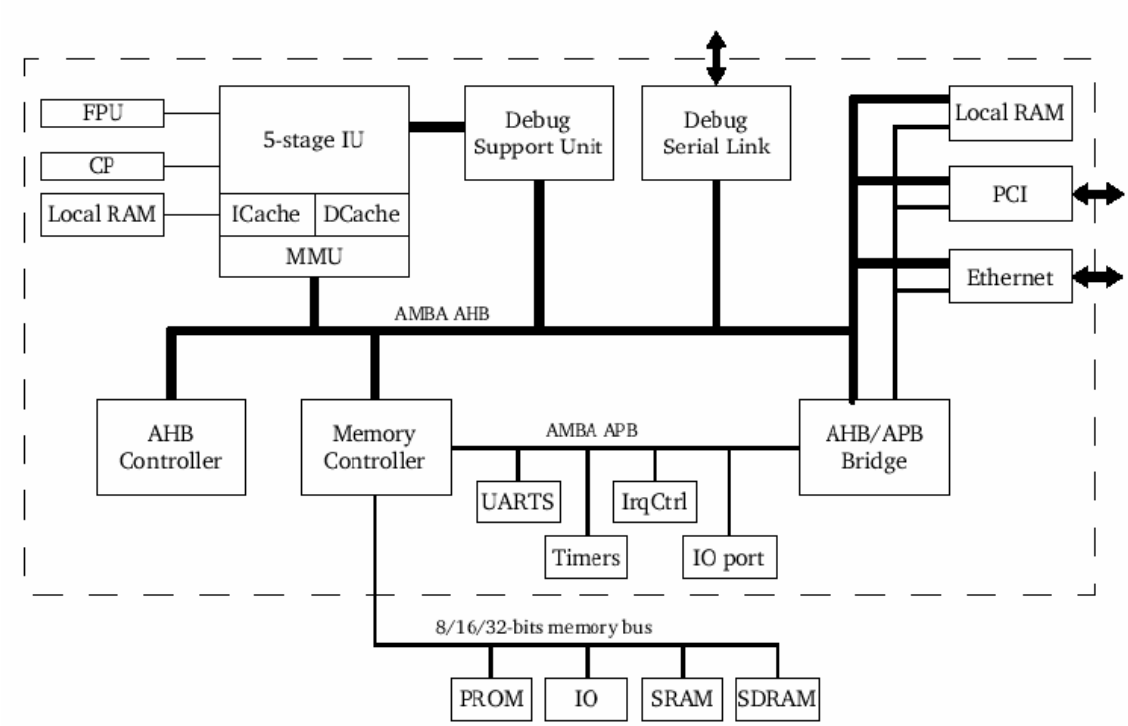



Figure 2.1: LEON2 microprocessor architecture

LEON2 implements the following features:

· *32 bits RISC microprocessor*

· *SPARC v8 compliant*

· *5-stage instruction pipeline*

· *multiply/divide/mac operations on hardware*

· *separated instruction and data caches*

· *memory management unit, MMU*

· *memory interfaces for FLASH, SRAM, SDRAM & PROM*

· *on-chip RAM*

· *interrupt handler*

· *interface for a floating point unit, FPU*

· *debug support unit, DSU*

· *two 24-bit timers*

SPARC v8 processor defines three main units, integer unit, floating-point unit and a custom coprocessor, each one with its own 32-bit internal registers. The latter two

units are optional, not mandatory for the processor. Leon2 implements the integer unit completely and the interfaces for the other two units in its core. Gaisler Research also has a commercial high performance FPU for Leon2 available [CATO03]. Leon2 also can provide a generic interface for a custom user defined co-processor which will work in parallel with the main processor in order to increase performance. Figure 2.2 presents the overview of the main processor. Except for the co-processor and the floating point unit (fpu), integer unit connects with the Register File (RF) and with system's cache, which consists of separate instruction and data controllers. Signal pairs *ici-ico* and *dci-dco* are used for transferring data from instruction and data cache respectively. Component cachemem is the configurable memory where data is stored.

Leon2 uses the **AMBA-2.0 AHB** bus to connect the main processor with high-speed controllers like memory and other optional units like the onchip RAM or PCI or Ethernet interfaces. In the default configuration the processor of LEON2 is the only master of the AHB bus. Cache component (in figure 2.2) is the sub-section inside the processor that manages all accesses on the bus and specifically uses signals *ahbi–ahbo* to communicate with AHB slaves. For example, if the required data from the integer unit is not found in cachemem then cache component uses the AHB bus to load/store data from/to the external memory.

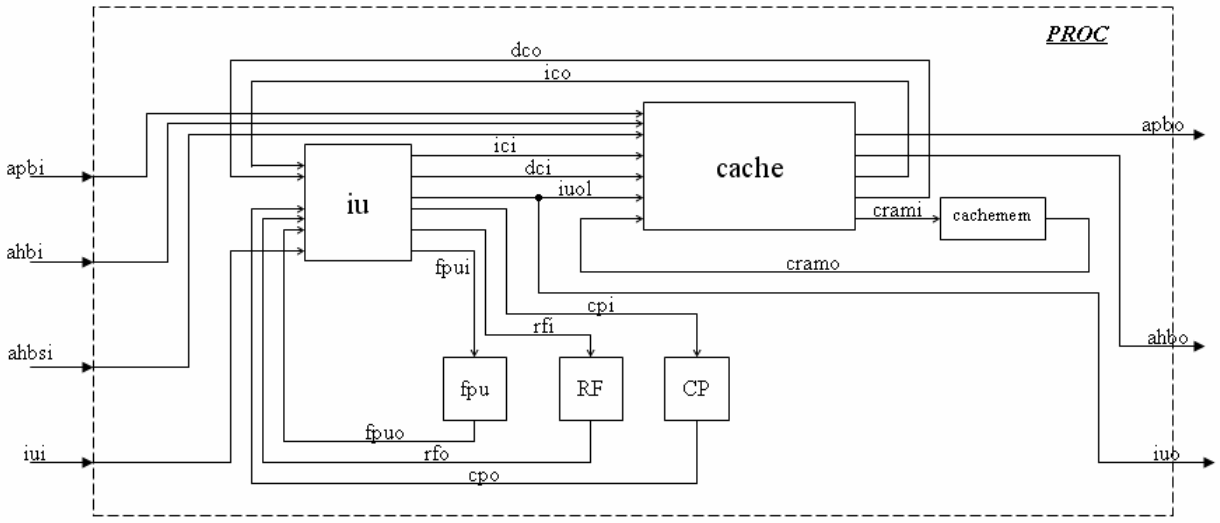



Figure 2.2: Main processor architecture

Figure 2.3 shows the connections on the AHB bus at the default configuration. The LEON processor core is normally connected as master 0, while the memory controller (component mctrl) and APB bridge are connected as slaves 0 and 1. The AHB controller (component ahbarb) controls the AHB bus and implements the bus arbiter. The AHB bus can connect up to 16 masters and any number of slaves, but in this case where only one master is connected, no arbitration scheme is needed. The debug support unit (component dsu) can read data returned to the processor on the AHB bus. The figure does not contain all the ports of the components; it just shows the way the processor transfers and receives data from other modules on the AHB bus.

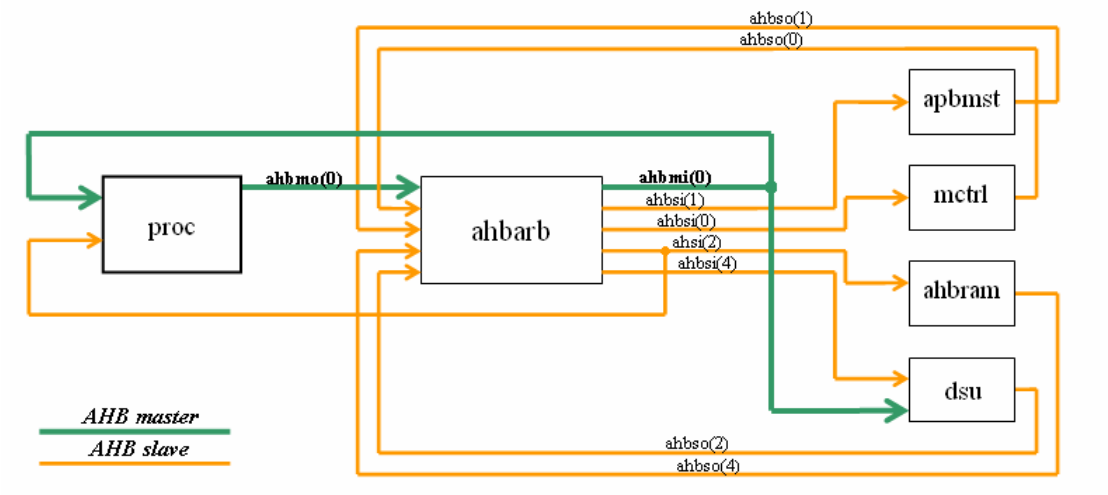



Figure 2.3: AHB bus connections

Another AMBA-2.0 bus is used to access most onchip peripherals, the **APB** bus. It is optimised for simple operation and low-power consumption and it is connected to the AHB bus via the AHB/APB bridge (component apbmst), which is the master of that bus.

The memory bus provides a direct interface to PROM, memory mapped I/O devices, asynchronous static ram (SRAM) and synchronous dynamic ram (SDRAM). Leon2 external memory access is provided by a programmable memory controller. The controller can decode a map of up to 2 Gbytes. Figure 2.4 shows how the connection to the different device types is made.

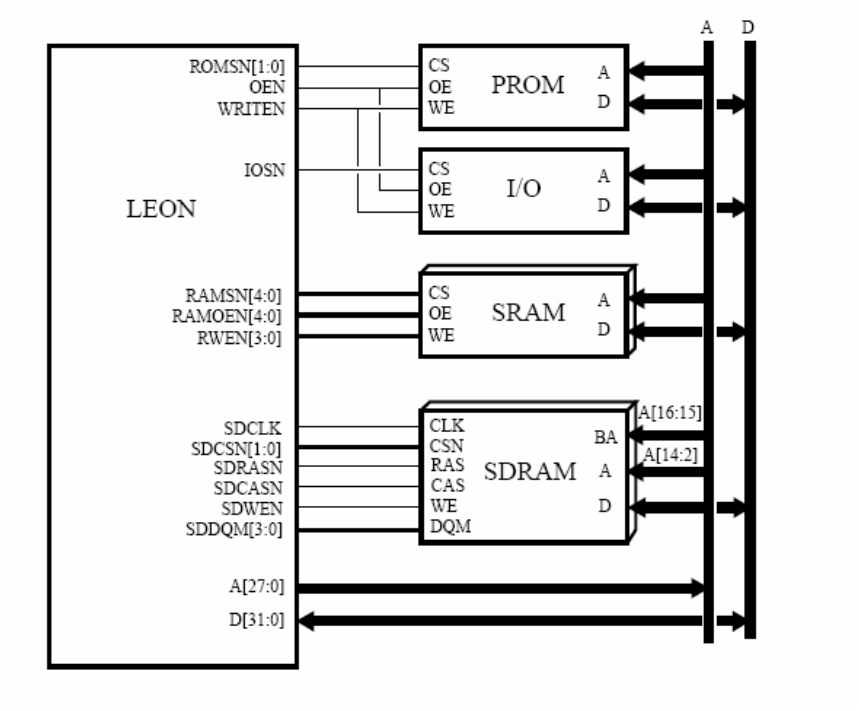



Figure 2.4: Memory device interface

## 2.2 Leon2 Integer Unit (IU)

The IU contains the general-purpose registers and controls the overall operation of the processor. The IU executes the integer arithmetic instructions and computes memory addresses for loads and stores. It also maintains the program counters (PC) and controls instruction execution for the FPU and the CP.

### 2.2.1 Sparc Architecture

The LEON integer unit implements SPARC integer instructions as defined in SPARC Architecture Manual version 8. SPARC includes the following *principal features*:

- A linear, 32-bit address space.
- Few and simple instruction formats — All instructions are 32 bits wide, and are aligned on 32-bit boundaries in memory. There are only three basic

instruction formats, and they feature uniform placement of opcode and register address fields. Only load and store instructions access memory and I/O.

- Few addressing modes — A memory address is given by either "register + register" or "register + immediate."
- Triadic register addresses— Most instructions operate on two register operands (or one register and a constant), and place the result in a third register.
- A large "windowed" register file — At any one instant, a program sees 8 global integer registers plus a 24-register window into a larger register file. The windowed registers can be described as a cache of procedure arguments, local values, and return addresses.
- Delayed control transfer— The processor always fetches the next instruction after a delayed control-transfer instruction. It either executes it or not, depending on the control-transfer instruction's "annul" bit.

A SPARC processor includes two types of registers: general-purpose or "working" data registers and control/status registers. The IU's general-purpose registers are called ***r*** registers. IU control/status registers include of several registers, such as the program counters (PC and nPC) and Processor State Register (PSR) which contains various fields that control the processor and hold status information.

An implementation of the IU may contain from 40 to 520 general-purpose 32-bit ***r*** registers. The total number of registers is implementation-dependent but at a given time an instruction can access the 8 *global*s and a 24-register window.

### 2.2.2 Instructions

The processor can be in either of two modes: **user** or **supervisor**. In supervisor mode, the processor can execute any instruction, including the privileged (supervisor-only) instructions. In user mode, an attempt to execute a privileged instruction will cause a trap to supervisor software.

Instructions fall into four basic categories:

1. Load/store
2. Arithmetic/logical/shift
3. Control transfer
4. Read write control register

**Load/Store**

Load/store instructions are the only instructions that access memory. They use two *r* registers or an *r* register and a signed 13-bit immediate value to calculate a 32-bit, byte-aligned memory address. The IU appends to this address an **address space identifier**, or **ASI** which encodes whether the processor is in supervisor or user mode, and that it is a data access. The destination field of the load/store instructions specifies an r register that supplies the data for a store or receives the data from a load. Integer load and store instructions support byte, halfword (16-bit), word (32-bit), and doubleword (64-bit) accesses.

**Arithmetic/Logical/Shift**

The arithmetic/logical/shift instructions perform arithmetic, logical, and shift operations. With one exception, these instructions compute a result that is a function of two source operands; the result is either written into a destination register, or discarded. The exception is a specialized instruction, SETHI, which writes a 22-bit constant from the instruction into the high-order bits of the destination register.

**Control Transfer**

Control-transfer instructions (**CTI**s) include PC-relative branches and calls, register-indirect jumps, and conditional traps. Most of the control-transfer instructions are delayed control-transfer instructions (**DCTI**s), where the instruction immediately following the DCTI is executed before the control transfer to the target address is completed.

The instruction following a delayed control-transfer instruction is called a **delay** instruction. The delay instruction is always fetched, even if the delayed control transfer is an unconditional branch. However, a bit in the delayed control transfer instruction can cause the delay instruction to be annulled (that is, to have no effect) if the branch is not taken (or in the branch always case, if the branch is taken).

**State Register Access**

The Read/Write Register instructions read and write the contents of software visible state/status registers. There are also read/write "ancillary state register" instructions that software can use to read/write unique implementation- dependent processor registers.

## 2.2.3 Instruction Execution – Instruction Formats

Architecturally, an instruction is read from memory at the address given by the program counter (PC). It is then executed or not, depending on whether the previous instruction was an annulling branch. An instruction may also generate a trap due to the detection of an exceptional condition, caused by the instruction itself (precise trap), a previous instruction (deferred trap), an external interrupt (interrupting trap), or an external reset request. If an instruction is executed, it may change program-visible processor and/or memory state.

Instructions are encoded in three major 32-bit formats, which are presented in Figure 2.5

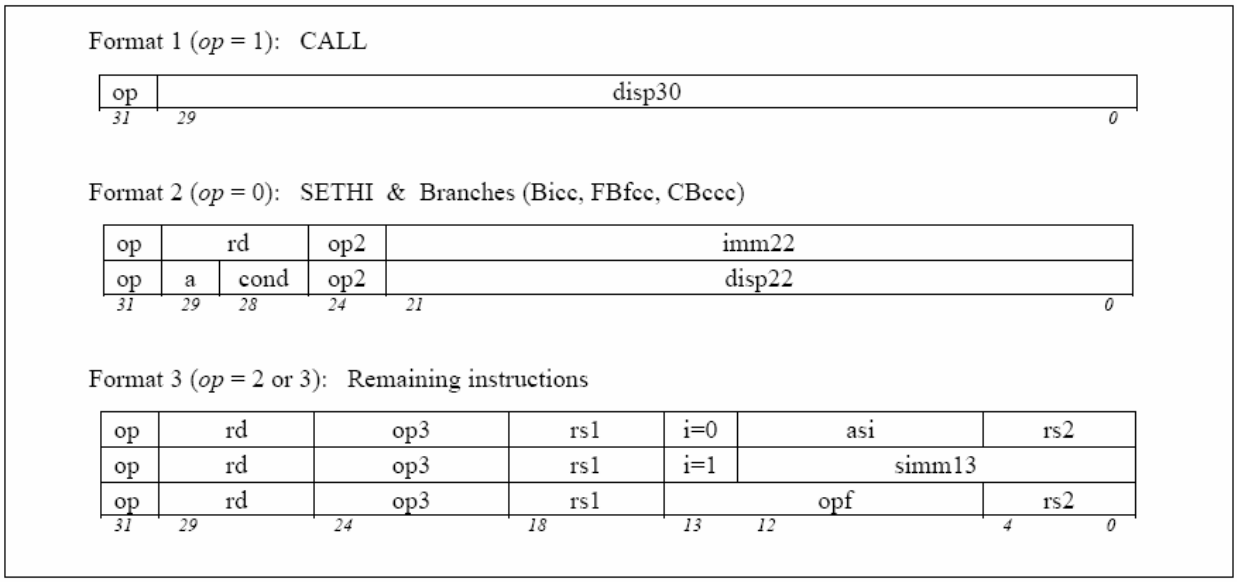



Figure 2.5: Summary of instructions formats

The instruction fields are interpreted as follows:

op and op2

These 2- and 3-bit fields encode the 3 major formats and the format 2 instructions according to Figures 2.6 and 2.7.

| Format | *op* | Instructions |
|---|---|---|
| 1 | 1 | CALL |
| 2 | 0 | Bicc, FBfcc, CBccc, SETHI |
| 3 | 3 | memory instructions |
| 3 | 2 | arithmetic, logical, shift, and remaining |

Figure 2.6: Op encoding (all formats)

| *op2* | Instructions |
|---|---|
| 0 | UNIMP |
| 1 | unimplemented |
| 2 | Bicc |
| 3 | unimplemented |
| 4 | SETHI |
| 5 | unimplemented |
| 6 | FBfcc |
| 7 | CBccc |

Figure 2.7: Op2 Encoding (format 2)

rd

This 5-bit field is the address of the destination (or source) *r* or *f* or coprocessor register(s) for a load/arithmetic (or store) instruction. For an instruction that read/writes a double (or quad), the least significant one (or two) bits are unused and should be supplied as zero by software.

a

The *a* bit in a branch instruction annuls the execution of the following instruction if the branch is conditional and untaken or if it is unconditional and taken.

cond

This 4-bit field selects the condition code(s) to test for a branch instruction.

imm22

This 22-bit field is a constant that SETHI places in the upper end of a destination register.

disp22 and disp30

These 30-bit and 22-bit fields are word-aligned, sign-extended, PC-relative displacements for a call or branch, respectively.

op3

This 6-bit field (together with 1 bit from *op*) encodes the format 3 instructions.

i

The i bit selects the second ALU operand for (integer) arithmetic and load/store instructions. If *i* = 0, the operand is r[rs2]. If *i* = 1, the operand is *simm13*, sign-extended from 13 to 32 bits.

asi

This 8-bit field is the address space identifier supplied by a load/store alternate instruction.

rs1

This 5-bit field is the address of the first *r* or *f* or coprocessor register(s) source operand. For an instruction that reads a double (or quad), the least significant bit (or 2 bits) are unused and should be supplied as zero by software.

rs2

This 5-bit field is the address of the second *r* or *f* or coprocessor register(s) source operand when *i* = 0. For an instruction that reads a double-length (or quad-length) register sequence, the least significant bit (or 2 bits) are unused and should be supplied as zero by software.

simm13

This 13-bit field is a sign-extended 13-bit immediate value used as the second ALU operand for an (integer) arithmetic or load/store instruction when *i* = 1.

opf

This 9-bit field encodes a floating-point operate (FPop) instruction or a coprocessor operate (CPop) instruction.

Appendix B contains the complete instructions set our processor.

### 2.2.4 Instruction pipeline

The LEON integer unit uses a single instruction issue pipeline with 5 stages (schematically represented in Figure 2.8):

1. FE (Instruction Fetch): If the instruction cache is enabled, the instruction is fetched from the instruction cache. Otherwise, the fetch is forwarded to the memory controller. The instruction is valid at the end of this stage and is latched inside the IU.
2. DE (Decode): The instruction is decoded and the operands are read. Operands may come from the register file or from internal data bypasses. CALL and Branch target addresses are generated in this stage.

3. EX (Execute): ALU, logical, and shift operations are performed. For memory operations (e.g., LD) and for JMPL/RETT, the address is generated.
4. MEM (Memory): Data cache is accessed. For cache reads, the data will be valid by the end of this stage, at which point it is aligned as appropriate. Store data read out in the execution stage is written to the data cache at this time.
5. WR (Write): The result of any ALU, logical, shift, or cache read operations are written back to the register file.

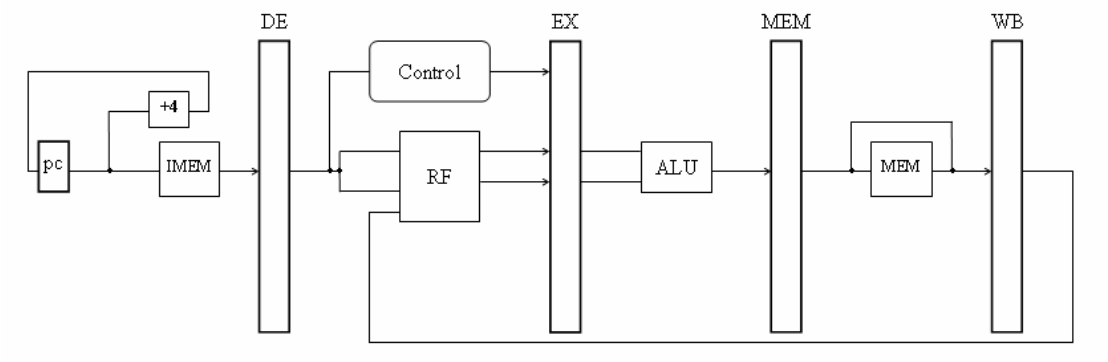



Figure 2.8: Simplified five-stage datapath

The major problem of all pipelined processors is the problem of hazards. A hazard occurs when an instruction in the pipeline cannot be executed. Figure 2.9 shows an abstract block diagram of the integer unit, mainly focused on bypassing signals which is a strategy to resolve the problem of data hazards. This type of hazard occurs when an instruction depends on the result of a previous instruction. In order to avoid stalling the pipeline, some signals are forwarded from later stages. There are 4 cases when data need to be forwarded by a pipeline register:

- Ex → Dec (for logical/arithmetic instructions)
- Mem → Dec (for logical/arithmetic instructions)
- Wb → Dec (both for memory – logical/arithmetic instructions)
- Wb → Ex (both for memory – logical/arithmetic instructions)

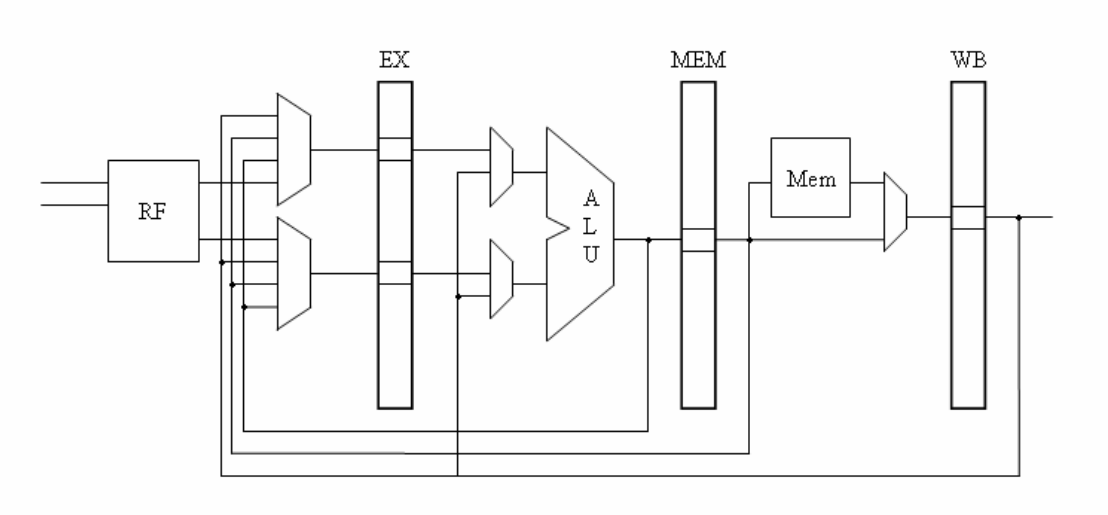



Figure 2.9: Abstracted datapath of IU, without control signals

The forwarding control was in the decode stage of the datapath. The control of these cases leads to control lines for multiplexors that select either the normal register values or one of the forwarded values. Signals that are used in later phases are passed via the pipeline registers. Figure 2.10 shows how this unit works.

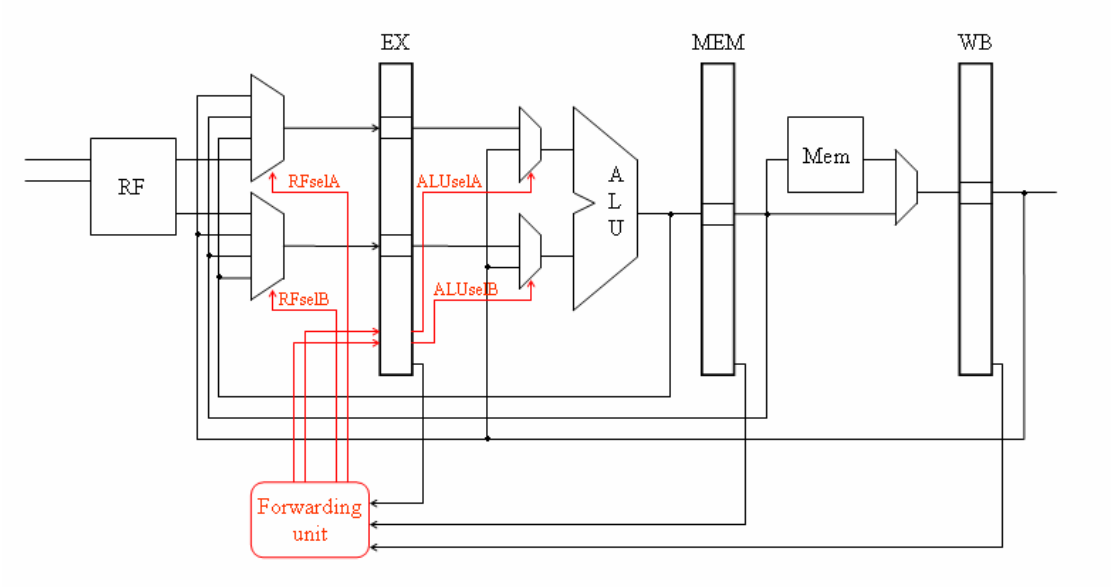



Figure 2.10: Forwarding unit

# 3. Leon2_ISE Architecture

In this chapter we will refer to the changes made in the Leon2 processor, with goal the expantion of its ISA to include symmetric key ciphers. All changes, were based on the master thesis of D.*Theodoropoulos "CCproc : A VLIW cryptography co-processor for symmetric key ciphers" [6]*. Theodoropoulos designed, after thoroughly analysed most cryptography algorithms, a co-processor called CCproc (Cryptography CoProcessor) and its Instruction Set, capable to support these ciphers. The first section makes an overview of Theodropoulos' analysis of private key ciphers, as well as the necessary structures and arithmetic operations needed to implement them. Section 3.2 presents the design considerations and the differences with our project. Section 3.3 shows the complete instruction set that we should add to the Leon2. The changes made in Leon2's Instruction Set Architecture are presented in sections 3.4 (arithmetical/logical/branch instructions) and 3.5(specific cipher instructions).

## 3.1 Private Key Ciphers

Ciphers can be categorized as *stream ciphers* or *block ciphers*. The former operate on the plaintext a single bit (or sometimes byte) at a time. The latter operate on the plaintext in groups of bits. The group of bits are called blocks, and the algorithms are called block ciphers. These algorithms typically have three operational parameters: key size, block size and number of rounds. The key size is the length of the key used to encrypt or decrypt data. The block size is the amount of data processed each time the cipher kernel is invoked. The number of rounds specifies the total number of iterations executed by the cipher kernel loop.

Figure 3.1 shows a generic schematic for the encryption/decryption process. Before message encryption starts, every symmetric cipher has an initialization phase. After this phase is complete, a certain number of various types of arithmetic operations are being applied on the plaintext for a specific number of rounds. Once the defined round number has been reached, encryption is finished and ciphertext is ready to be

transmitted. Decryption process in most cases, if it is not identical, then it is almost the same.

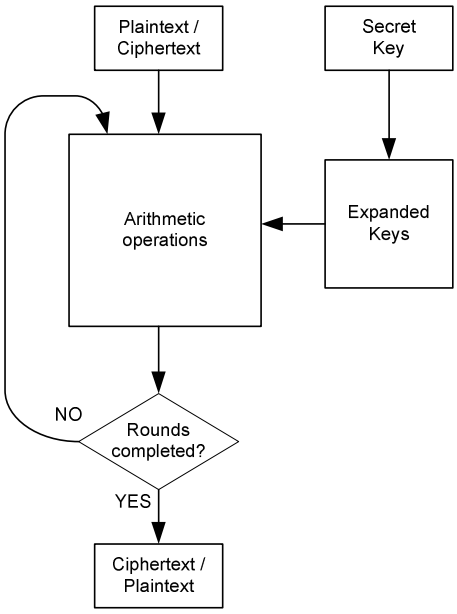



Figure 3.1: Encryption/Decryption process

After analysis, *Theodoropoulos* concluded that the operations and structures most commonly used are:

1. Unsigned addition and subtraction modulo $2^{32}$
2. Multiplication modulo $2^{32}$
3. Exclusive or (xor) between 32-bit data
4. Fixed shifts and rotations
5. Data depended shifts and rotations
6. Finite field polynomial multiplication in $2^8$ modulo a prime polynomial
7. Expansions and permutations (Xboxes)
8. Substitution boxes (Sboxes)
9. Feistel network structures

In Leon2, like most 32-bit processors, operations from 1 to 6 are implemented very fast, except for the field polynomial multiplication (FFM) modulo a prime polynomial.

Besides arithmetic operations, there are common structures among ciphers. Sboxes are usually non-linear structures that map an n-bit value to an m-bit value, essentially Look Up Tables (LUT). A symmetric cipher may have one or more different Sboxes, with each one of them having arbitrary dimensions, as shown in Figure 3.2.

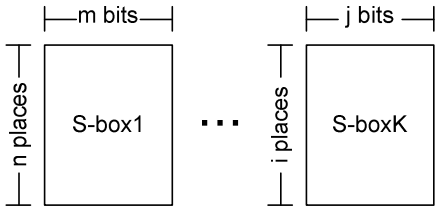



Figure 3.2: Sboxes

## 3.2 Design considerations

### 3.2.1 Theodoropoulos' CCproc considerations

In order to implement the necessary instructions, some considerations had to be taken into account:

- The operations of bit permutations are not adopted by the newest ciphers and specifically none of the AES round two finalists except Serpent uses them. Furthermore they require a considerable amount of hardware and though it was decided not to be used.
- The research revealed a high frequency occurrence of two dependent, back-to-back instructions. Examples are double additions, subtractions and XORs operations, as well as addition-subtraction followed by a XOR.
- A fact that characterizes every symmetric cipher is the determined number of rounds during the key expansion process, plus encryption / decryption. As a result they can be written in a way that requires absolutely no branch tests. This observation let to the decision to support a "loop" instruction that would a priori know the number of rounds and so pre-evaluate the direction, adding no branch-related pipeline stalls.
- Due to complexity of the key expansion process, Theodoropoulos supported an extra functional unit: a KRF (Key Register File) memory module in order to store all expanded keys.

- Many times during processing, symmetric ciphers require 64-bit, 96-bit or even 128-bit data values at the same time in order to proceed, so with a single 32-bit RISC datapath additional clock cycles are spent on fetching all appropriate data to the functional unit that will use them. This performance obstacle led to the decision to examine and finally implement a VLIW 5-stage pipelined processor, that would consist of four 32-bit clusters, capable to process four 32-bit instructions in one clock cycle. Its abstract schematic overview is shown in Figure 2.6

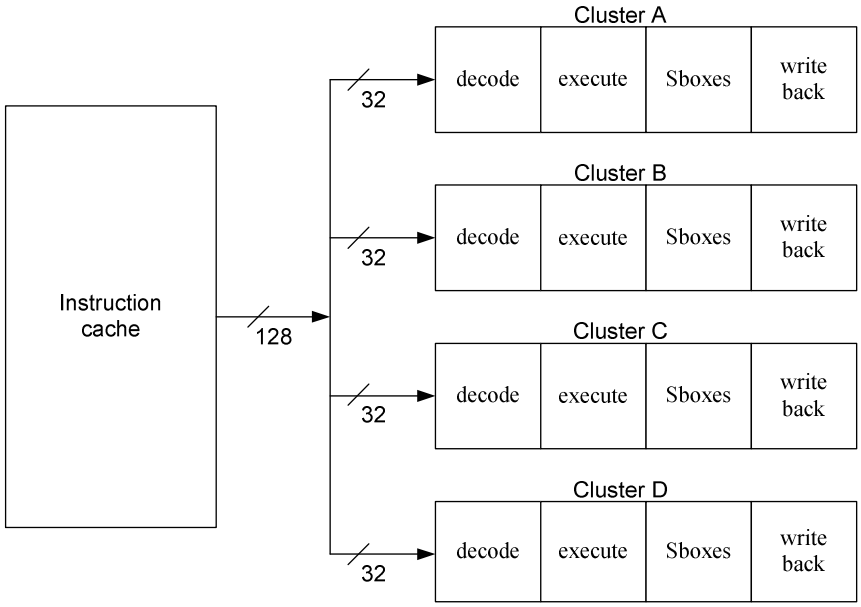



Figure 3.3: CCproc's abstract schematic overview

### 3.2.2 Leon2 Design differences

- The main difference of our design is that we have a single 32-bit RISC datapath structure extended to support an enhanced symmetric cipher ISA. As a result any instruction that may require 128-bit data values at the same time in order to proceed, would be delayed. The narrower datapath does not affect the effectiveness of our instructions but merely their execution speed.
- Leon2's RF (Register File) can access a large number of registers (up to 520). So, in our design, it was decided *not* to include a KRF, and store all key values

into the RF preventing from any additional stall to support operations between RF's data and KRF's data

## 3.3 CC proc Instruction Set

After evaluating all the above considerations, Theodoropoulos designed the ISA of his CCproc, categorizing the instructions to four primary formats, Register, Immediate, loop and cipher. As stated before, some of these instructions are already implemented in Leon2. Table 1 summarizes all instructions *supported* in CCproc and **not** supported in Leon2 in R, I and loop formats.

| Format | Operation | Syntax | Description |
|---|---|---|---|
| R | Gfm | gfm rdx,rsa | galois field multiplication in GF(28) between rsa and GF operand x and the result is stored to rdx |
| | krfpaz | krfpaz | resets KRF's pointer to first address |
| | ldgfmr | ldgfmr rsa | loads 8-bit mr register with rsa's value, which holds modulo polynomial in Galois Field multiplication |
| | ldgfopx | ldgfopx rsa | loads Galois Field operand x with rsa's value |
| | ldlc | ldlc #a | loads 6-bit lc register with #a |
| | r2c/c2r | r2c / c2r rdx,rsa | toggles between rows and columns in a 128-bit data value |
| | Rol | rol rdx,rsa,rsb | rotates left rsa by the amount specified from rsb's 5 LSBs and stores the result to rdx |
| | Ror | ror rdx,rsa,rsb | rotates right rsa by the amount specified from rsb's 5 LSBs and stores the result to rdx |
| | **addadd** | addadd rdx,rsa,rsb,rsc | adds rsa with rsb, adds the result to rsc and stores it to rdx |
| | **addsub** | addsub rdx,rsa,rsb,rsc | adds rsa with rsb, subtracts rsc from the result and stores it to rdx |
| | **addxor** | addxor rdx,rsa,rsb,rsc | adds rsa with rsb, logic xor between rsc and the result and stores it to rdx |
| | **subadd** | subadd rdx,rsa,rsb,rsc | subtracts rsb from rsa, adds the result to rsc and stores it to rdx |
| | **subsub** | subsub rdx,rsa,rsb,rsc | subtracts rsb from rsa, subtracts rsc from the result and stores it to rdx |
| | **subxor** | subxor rdx,rsa,rsb,rsc | subtracts rsa with rsb, logic xor between rsc and the result and stores it to rdx |

| | | | |
|---|---|---|---|
| | **xoradd** | xoradd rdx,rsa,rsb,rsc | logic xor between rsa and rsb, adds the result to rsc and stores it to rdx |
| | **xorsub** | xorsub rdx,rsa,rsb,rsc | logic xor between rsa and rsb, subtracts rsc from the result and stores it to rdx |
| | **xorxor** | subxor rdx,rsa,rsb,rsc | logic xor between rsa and rsb, logic xor between rsc and the result and stores it to rdx |
| I | roli | roli rdx,rsa,#a | rotates left rsa by the amount specified from #a's 5 LSBs and stores the result to rdx |
| | rori | rori rdx,rsa,#a | rotates right rsa by the amount specified from #a's 5 LSBs and stores the result to rdx |
| loop | loop | loop label | jumps to the beginning of a loop which starts at address “label” |

Table 1: Supported operations in R, I and loop formats. Bold means double instructions.

From the above instructions those referring to the KRF module and Galois Field Multiplication were not implemented. Moreover, the instruction r2c/c2r has no meaning in the single RISC processor.

The supported cipher instructions which access different memory modules called Sboxes, are presented in Table 2.

| **Instruction** | **Syntax** | **Description** |
|---|---|---|
| *aesX* | aesX rdx,rsa | Sbox access during AES encryption or decryption (X=E,D) with rsa and the result is stored to rdx |
| *marsX* | marsX rdx,rsa | Sbox access during MARS forward mode, backward mode, or E function (X=F,B,E) with rsa and the result is stored to rdx |
| *serX* | serX rdx,rsa | Sbox access during Serpent encryption or decryption (X=E,D) with rsa and the result is stored to rdx |
| *tX* | tsld rsa,rsb / tsbox rdx,rsa | during Twofish, loads to S0 and S1 rsa and rsb respectively / Sbox access with rsa and the result is stored to rdx |

Table 2: Supported cipher instructions

## 3.4 Arithmetic/Logical/Branch Instructions

In figure 2.5 we presented the three basic formats of instructions for Leon2 Instruction Set. Our goal is the enhancement of the existing Instruction Set Architecture (ISA) with the new instructions presented in the previous chapter.

Instructions with op = 2, are divided according to op3 value. Table 3 lists the possible instructions for op3[5:0]. With red colour, we distinguish the new instructions which were added to the design.

| | | op3 [5:4] | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 |
| op3 [3:0] | 0 | ADD | ADDcc | TADDcc | WRASR† WRY‡ |
| | 1 | AND | ANDcc | TSUBcc | WRPSR |
| | 2 | OR | ORcc | TADDccTV | WRWIM |
| | 3 | XOR | XORcc | TSUBccTV | WRTBR |
| | 4 | SUB | SUBcc | MULScc | FPop1 **See Table F-5** |
| | 5 | ANDN | ANDNcc | SLL | FPop2 **See Table F-6** |
| | 6 | ORN | ORNcc | SRL | CPop1 |
| | 7 | XNOR | XNORcc | SRA | CPop2 |
| | 8 | ADDX | ADDXcc | RDASR* RDY** STBAR*** | JMPL |
| | 9 | TSLD | | RDPSR | RETT |
| | A | UMUL | UMULcc | RDWIM | Ticc **See Table F-7** |
| | B | SMUL | SMULcc | RDTBR | FLUSH |
| | C | SUBX | SUBXcc | IROR | SAVE |
| | D | DBL | | IROL | RESTORE |
| | E | UDIV | UDIVcc | ILOOP | |
| | F | SDIV | SDIVcc | LDLC | |

Figure 3.4: Arithmetic/Logic instructions in Leon2

We will present one instruction at a time, and the changes made in Leon2 integer unit. We tried to keep the basic format that these instructions follow. Of course every instruction has its specific format, described in detail on appendix A. The TSLD instruction will be presented in the next chapter as it is connected with a cipher instruction.

### 3.4.1 Iror/Irol

Inside the execution stage, we added the rotation of register (rs1) left or right by the distance indicated by either the register's rs2 five least significant bits, or the 5-bit *rotcnt* value. The result is saved into destination register rd.

The difference from the Shift instructions (SRL, SRA) is that after the shift, the vacated positions are not filled with zeroes but with register's MSB or LSB depending on the direction of the rotation.

### 3.4.2 Double Instructions

The main difference of these instructions is that double-instructions require three operands, something that it is not feasible as RF reads only two registers. In order to accomplish a double instruction at one clock cycle, we have to read 3 operands from the RF. So, we created a second copy of Register File's memory.

After this change was made, we added a module inside memory stage capable of adding/subtracting or making a Xor operation between two operands. This extension helps us executing the second phase of the instruction, as the first one is implemented inside the ALU during the execution stage. We preferred not to insert both operations in execution stage, so that we avoid having a greater delay on the critical path.

The figure 3.5 presents the datapath with a 4-port RF and the added module that is used during the memory stage of any double instruction.

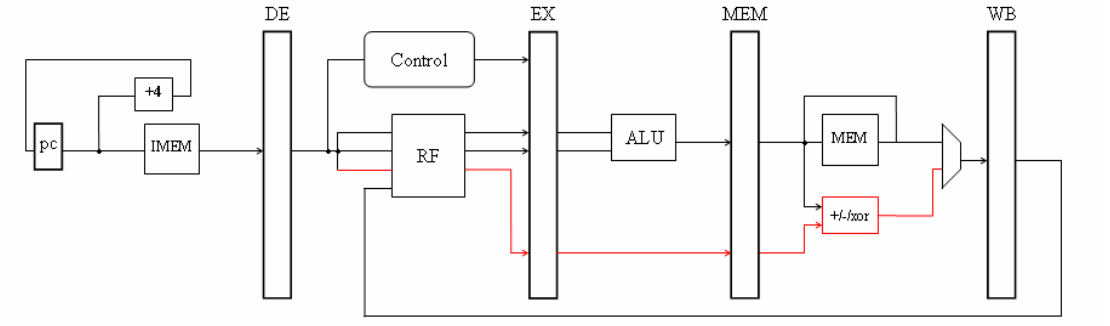



Figure 3.5: Datapath that includes double instructions; changes are in colour

Unlike all arithmetic/logic instructions which produce their result at the end of the execution stage, double instructions produce their result at the end of the memory stage. This leads to a data hazard, when a double instruction writes the register the following instruction wants to read. Figure 3.6a presents this case. A solution to this

problem is stalling the dependent instruction in the decode stage for one cycle. Figure 3.6b shows how this stall removes the problem. Data is then forwarded from the write back (WB) pipeline register to the dependent instruction.

To achieve these, we had to extend the operation of the Control Unit. This necessitated that the pipeline registers Ex, Mem are extended with 1 bit register which shows that we refer to a double instruction.

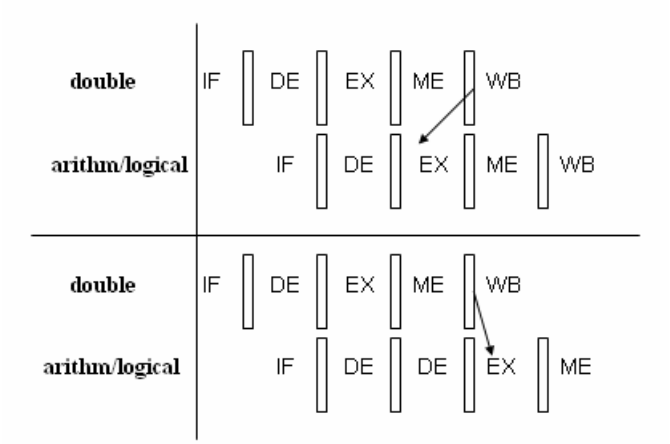



Figure 3.6: Pipeline dependency during a double instruction

To conclude, all double instructions, execute in one cycle. If the instruction that follows, reads the register written by the double instruction, then there is a one-cycle stall. All other data dependencies are removed.

### 3.4.3 Loop Instruction

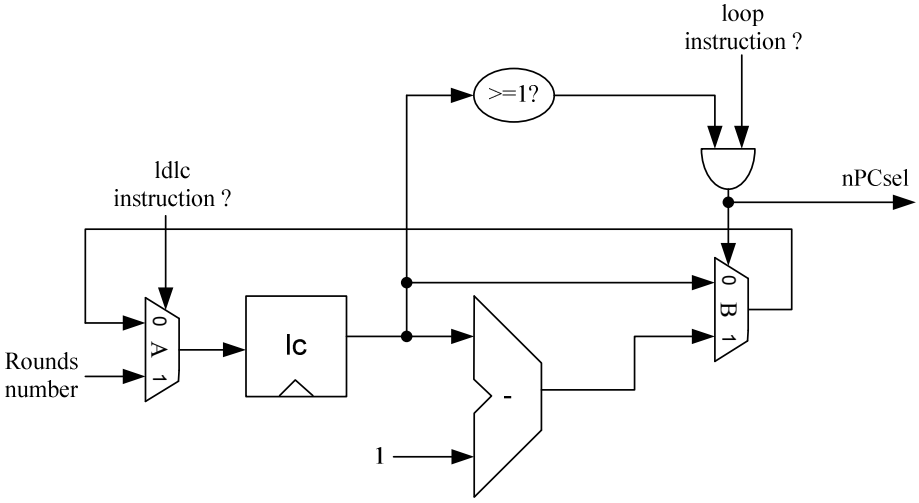
A control-transfer instruction changes the value of the next program counter (nPC). There are five basic control-transfer instruction (**CTI**) types in Leon2:

- Conditional branch (Bicc, FBfcc, CBccc)
- Call and Link (CALL)
- Jump and Link (JMPL)
- Return from trap (RETT)
- Trap (Ticc)

As we can see control-transfer instructions can be categorized to conditional and non-conditional. Only the first type of instructions decide whether to change the normal execution order or not, according to the value of icc (integer conditions codes).

Loop instruction is a conditional CTI. It examines the value of a loop counter (LC), and if that value is not zero, then the branch is taken. Otherwise, the branch is not taken and the next program counter is PC+4. As we mentioned before, this instruction is important for cases when we already know the number of rounds, resulting to a less number of instructions needed and consequently smaller execution time.

To implement this instruction, we used the loop controller circuit that Theodoropoulos had proposed. As it can be seen from figure 3.7, there is "lc" register, two multiplexers A and B, a '1' constant subtraction unit and a comparator. When an instruction is being fetched from instruction cache, it is checked if an "ldlc" or "loop" occurred. If it is the first case, then multiplexer A gives to "lc" the rounds number that a loop will be repeated. The latter is complete when a "loop" instruction occurs, and if "lc" value is greater than 1, then its current value is reduced by 1 and "nPCsel" signal is asserted, in order to enable a new instructions loop commencement. If "lc" is 0, it means that the appropriate rounds number has been completed, "nPCsel" is not asserted and program execution continues normally




. Figure 3.7: Loop controller

The loop counter (lc) register was added to the execution pipeline registers, and the loop controller logic was put in the decode stage, where main control executes. Figure 3.8 shows this change in the decode stage. The value of register "lc" may :

- Remain the same, if the instruction is neither a ldlc nor a loop.
- Take the value rounds_number, if it is a ldlc instruction
- Take the value ex.lc – 1, if it is a taken loop instruction.

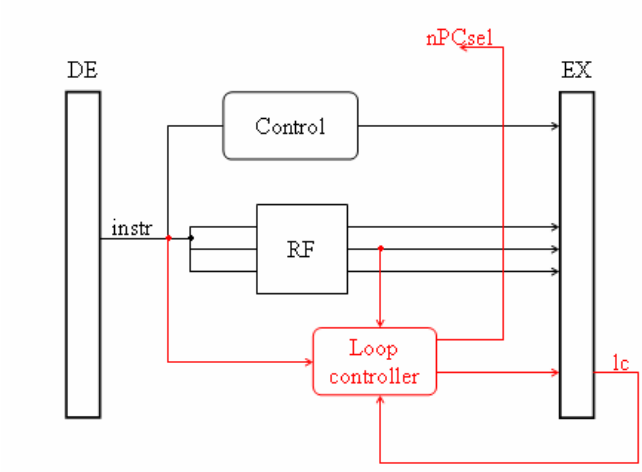



Figure 3.8: Loop controller in datapath

So far, we showed how the signal nPCsel is produced. When it is asserted a branch occurs. The target address is calculated by sign-extending its immediate field to 32 bits, left-shifting that word displacement by two bits to create a byte displacement, and adding the resulting byte displacement to the contents of the PC. This procedure is the same with the one instructions CALL and JMPL follow. The only difference amongst these instructions is that LOOP has an immediate field of 13 bits, CALL of 30 bits, and finally JMPL has a 22 bits field. If we consider that the loop controller is an extension on the existing control unit, then we can represent graphically the changes made in the IU's decode stage in figure 3.9. The "branch" signal is asserted when one of the three instructions changes PC's contents. Signals "jump" and "trap" concern the Control Transfer Instructions JMPL and TRAP respectively. These signals may be asserted during execute/write-back stages.

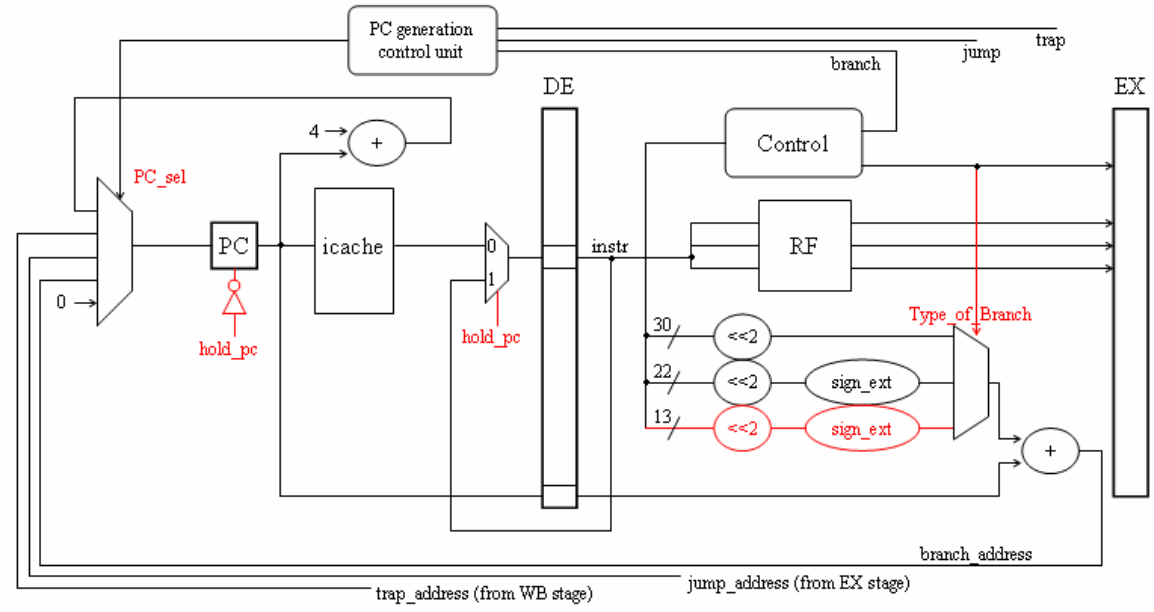



Figure 3.9 Control Transfer Instructions in datapath

LOOP instruction is a delayed control transfer instruction. Specifically, it changes control to the instruction at the target address after a 1-instruction delay. The delay instruction executed after the CTI is executed before the target of the CTI is executed.

## 3.5 Cipher Instructions

As it was mentioned in chapter 3.1, besides arithmetic operations, there are structures used by ciphers. These structures called Substitution boxes (Sboxes) are essentially memory modules or Look Up Tables (LUT). This section focuses on the description of these Sboxes separately for every cipher, and their integration to the entire design. All Sboxes have been placed to leon's memory stage.

The primary goal was to extend the interface of the integer unit so that includes the communication signals with the controllers of the ciphers. This is shown in Figure 3.10 where we omitted the optional units such as the co-processor and the floating point unit (fpu). As it can be seen, cipher Sboxes can be considered as small caches, working in parallel with the processor's main memory.

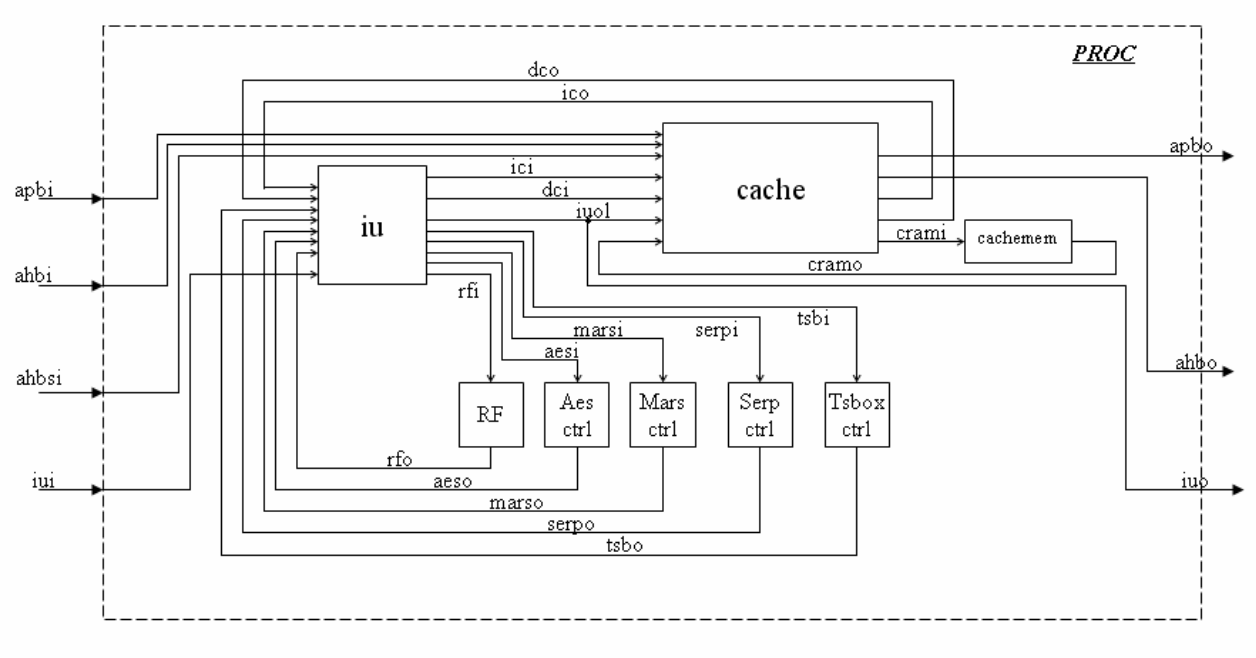



Figure 3.10: Main processor architecture including ciphers' Sboxes

### 3.5.1 AES cipher Sboxes

The AES cipher uses two 256x8 Sboxes, one for the encryption and one for the decryption processs. Moreover, we wanted to be able to read 4 values of each memory

so we created one copy of each dual port memory in order to maximize parallelism. Figure 3.11 shows how Sboxes have been implemented, where "E" stands for encryption, and "D" stands for decryption mode.

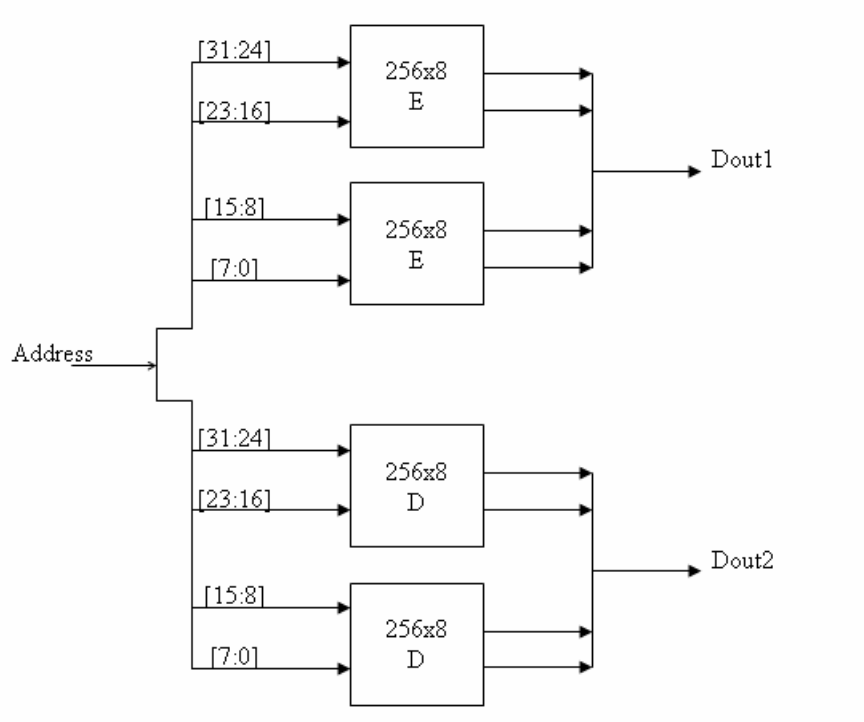



Figure 3.11: AES Sboxes

After Sbox access, there have been produced four bytes from E-Sboxes and four from D-Sboxes, which are concatenated into two 32-bit words. The interface of the aes controller is presented below. Inputs of the controller except for the address, are the write enable (we) signal, 8-bits write data (din), and signals encryption and decryption that are asserted every time the instruction refers to the respective memory.

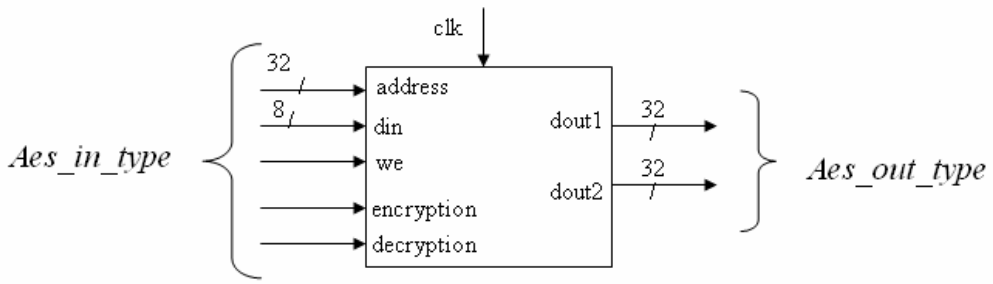



Figure 3.12: AES controller

As it is seen in Figure 3.13, aes caches are synchronous. The result of a Aes load instruction is produced at the end of the memory stage. Instruction bit 9 determines the choice between the two outputs.

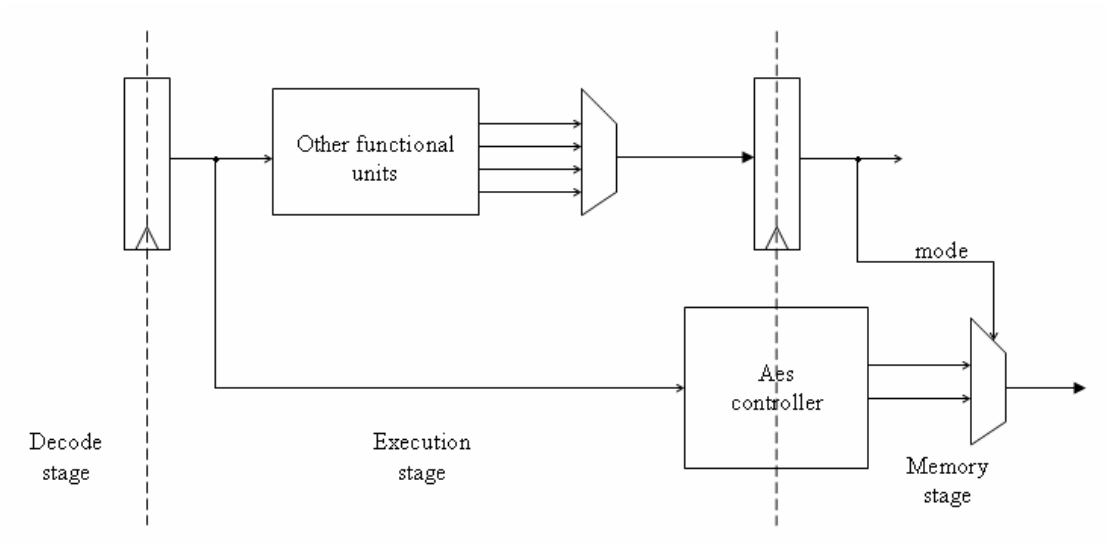



Figure 3.13: Implementation of AES controller in datapath

### 3.5.2 Twofish cipher Sboxes

Twofish uses a different Sbox structure from other ciphers, in a way that its final result depends on the secret key that is being used. Also, Twofish uses the same Sbox structure for both encryption and decryption process. Figure 3.14 shows its Sbox structure, where "S0" and "S1" are two of the subkeys. "In" is the Sbox input, and "q0" and "q1" are 8x8 Sboxes.

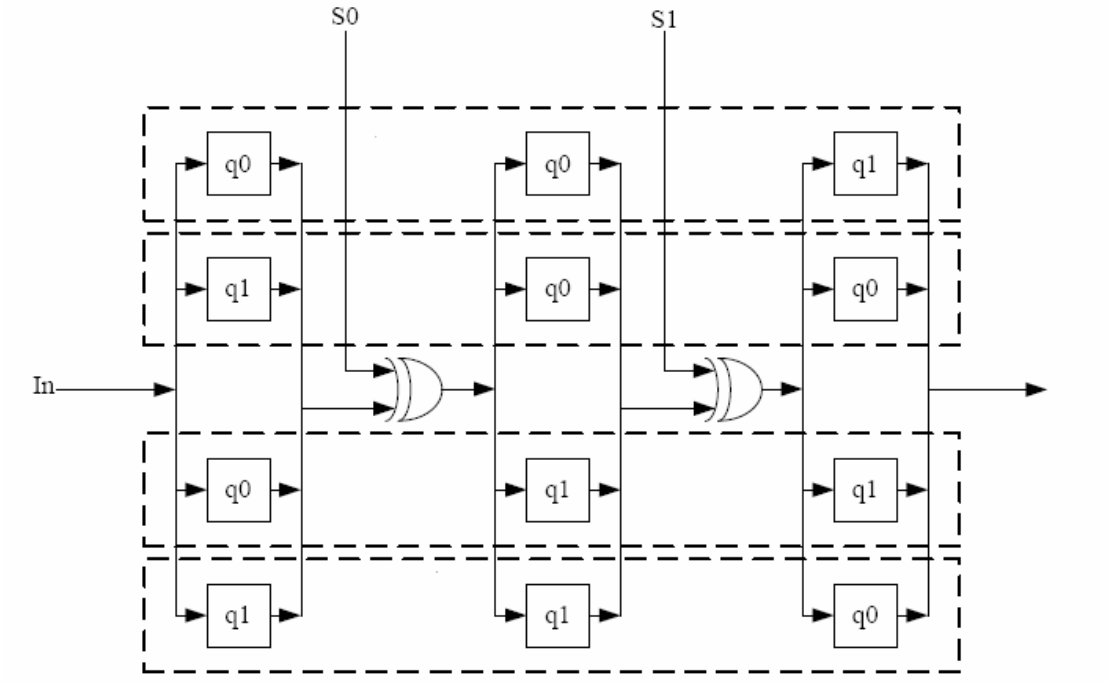



Figure 3.14: Twofish Sboxes

Our first action was to enhance the existing Instruction Set with an instruction that loads the subkeys "S0" and "S1". This is done with the instruction TSLD (tsbox load), which reads two values from the register file (general purpose/r registers) and stores them into 2 new status registers (which were added in the design) during the execution phase. The value of these status registers is altered, only if another TSLD instruction is executed.

Due to the fact that during a single load Twofish Sbox instruction, as shown in figure 3.14, three independent Sbox structures accesses as well as two xor operations between them are needed, we could not place this logic as a whole in a combinational circuit. In the beginning we considered to split Figure 3.14 in two pieces, using two types of Twofish Sboxes memories, a synchronous and an asynchronous, both having identical data. Figure 3.15 shows the first implementation of twofish sboxes in datapath.

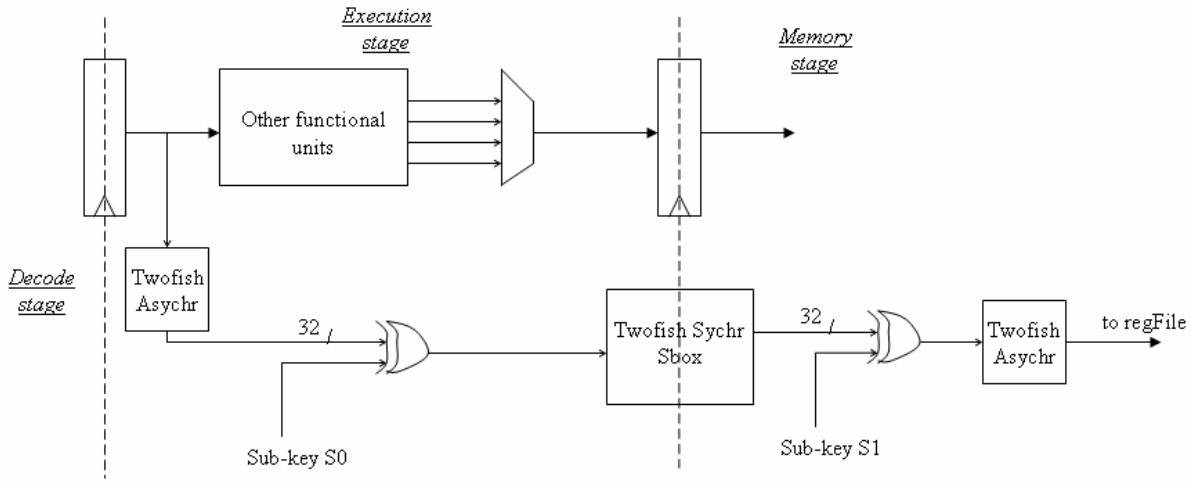



Figure 3.15: Twofish Sboxes in datapath

After evaluating this implementation, we discovered that there was a negative impact to the frequency of our design, so a next attempt was to implement it in 3 separate cycles keeping only the synchronous Twofish controller. Specifically, during the first cycle data is read from the Twofish controller. In the second cycle a xor operation is made between the data read and the first sub-key and the result accesses the Twofish controller. Finally, in the third cycle another xor operation is made between the data

read from cycle2 and the second sub-key and the result accesses the Twofish controller. The final result is stored in the register file during the write back stage.

The afore-mentioned 3 cycle implementation requires that a TSBOX load instruction is designed as a multi-cycle instruction. To achieve this, we used the existing datapath of Leon2 shown in Figure 3.15. During normal execution, signal "hold_pc" is not asserted, so the next instruction that accesses instruction cache is PC + 4. When we want to stall the pipeline, "hold_pc" is asserted keeping the same instruction in decode stage. In case there is a multi-cycle instruction then signal "cnt" is incremented by one.

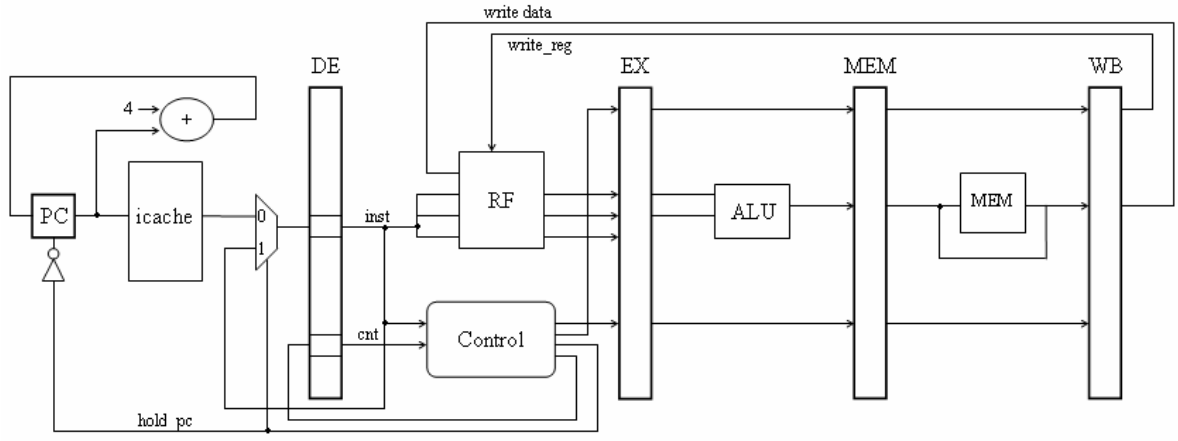



Figure 3.16: *Existing* datapath focused on multi-cycle instructions

The first changes we had to make, were to extend the control unit to check the cycle of the TSBOX instruction and assert the respective bits.

```
If ( instr = TSBOX) {
        If (cnt = "000")  then
                write_reg = '0'; holdn = '1'; new_cnt = "001";
        else if (cnt = "001") then
                write_reg = '0'; holdn = '1'; new_cnt = "010";
        else if (cnt = "010") then
                write_reg = '1'; holdn = '0'; new_cnt = "000";
}
```

During the cycles 2, 3 the xor operations use as operands the two sub-keys stored during a TSLD instruction, and the result produced from the Twofish controller. So, another necessary change was the extension of the multiplexors during the execution stage.

The final implementation led to a single synchronous Twofish controller, that reads two values at a time of each memory q0, q1. Figure 3.16 shows how the read address accesses twofish controller. During a single TSBOX load instruction, Twofish controller is accessed 3 times, one time for each cycle.

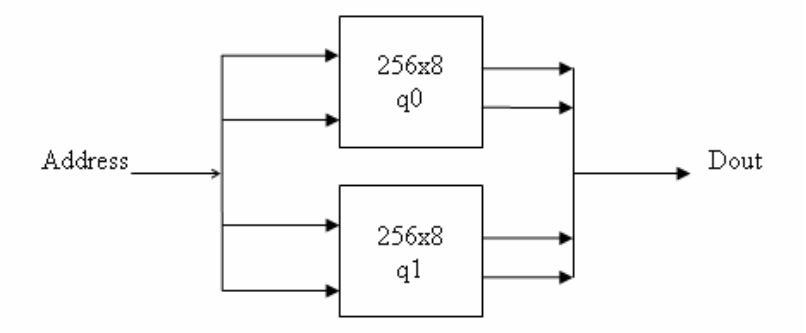



Figure 3.17: Twofish controller read access

### 3.5.3 Serpent cipher Sboxes

Serpent is somehow different from the ciphers already mentioned, as it demands an address of 128 bits (4 words) to access its Sboxes. This happens, in order to succeed data permutation in the beginning and end of processing. Sbox access will be presented through the following example.

Suppose that X0, X1, X2, X3 are the four 32-bit words of plaintext where X0 is the most significant one, and consider that each word's four MS bits in hexadecimal are:
X0 = hex"6…", X1 = hex "a…", X2 = hex "f…", X3 = hex "8…"
Table 3 shows these numbers also in binary while each column indicates the respective bit. Last column "weight" shows the value that emerges when computing each column's in decimal.

| hex | bit 31 | bit 30 | bit 29 | bit 28 | weight |
|---|---|---|---|---|---|
| 6 | 0 | 1 | 1 | 0 | $2^0$ |
| a | 1 | 0 | 1 | 0 | $2^1$ |
| f | 1 | 1 | 1 | 1 | $2^2$ |
| 8 | 1 | 0 | 0 | 0 | $2^3$ |

Table 3 – Serpent Sbox access example

For example, "bit31" $= 0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 = 14$, which is the Sbox's access address. Then, we assume (arbitrarily) that Sbox [14] = 9. Similarly the other columns emerge the following values:

"bit30" $= 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 = 5$, Sbox [5] = 6

"bit29" $= 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 = 7$, Sbox [7] = 11

"bit28" $= 0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 = 4$, Sbox [4] = 10

In Table 4, "bit" columns contain the above results, in binary according the "weight" column. Indeed "bit31" = 9, "bit30" = 6, "bit29" = 11 and "bit28" = 10.

| hex | bit 31 | bit 30 | bit 29 | bit 28 | weight |
|---|---|---|---|---|---|
| a | 1 | 0 | 1 | 0 | $2^0$ |
| 7 | 0 | 1 | 1 | 1 | $2^1$ |
| 4 | 0 | 1 | 0 | 0 | $2^2$ |
| b | 1 | 0 | 1 | 1 | $2^3$ |

Table 4 – Serpent Sbox results example

Finally, if the resulting lines are considered as binary values, with each cell in "bit31" column containing the MSB, column "hex" translates them to hexadecimal and these are the final replacements: 6↔a, a↔7, f↔4, and 8↔b.

As we can see, Serpent requires all four 32-bit words at the same time leading to an implementation shown in Figure 3.18. The only difference from our implementation is that we used 16 dual-port memories instead of 32 single-port ones.

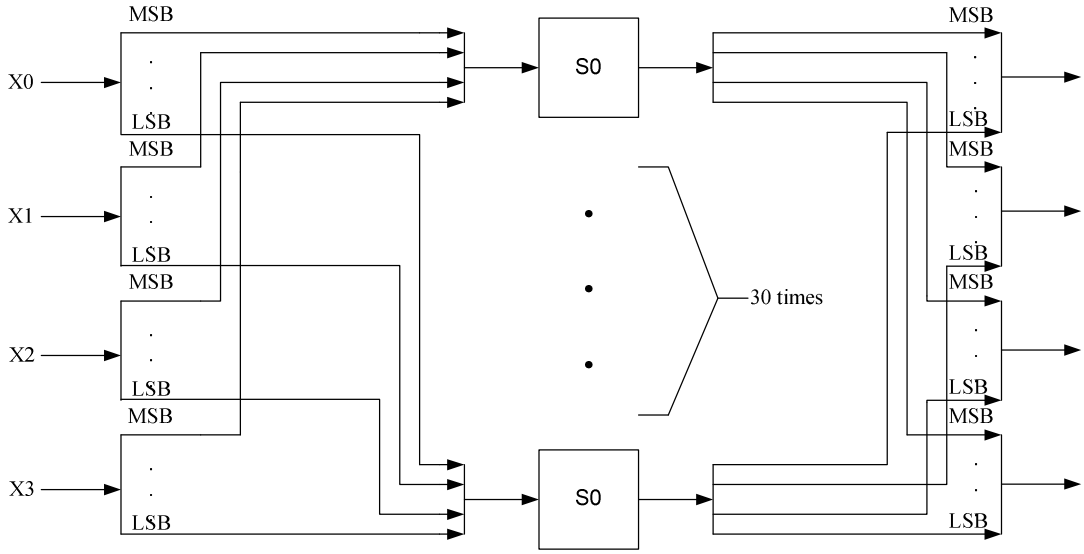



Figure 3.18: Serpent Sboxes

In LEON2 processor it is not possible to read four register values during a single instruction. This led to the decision to split the serpent load instruction in two phases. When a load-phase1 instruction comes, the two most significant words of the address are read from RF and the first two destination registers are stored in the execution pipeline registers (control registers added to the execution stage). The load-phase2 instruction is the one that really accesses Serpent sboxes. This instruction reads the two least significant words of the address (from RF) and besides contains the third and fourth destination register of the instruction. During this instruction, the 4 words are used as the address of the Serpent controller and the result is stored in the four destination registers. Of course, this result is 128 bits length, so there is a 3-cycle stall to write the data produced to each of the four destination registers. Figure 3.19 shows how Serpent controller in integrated in datapath. The “cnt” bit which counts the cycle of the instruction, decides which of the four words to choose.

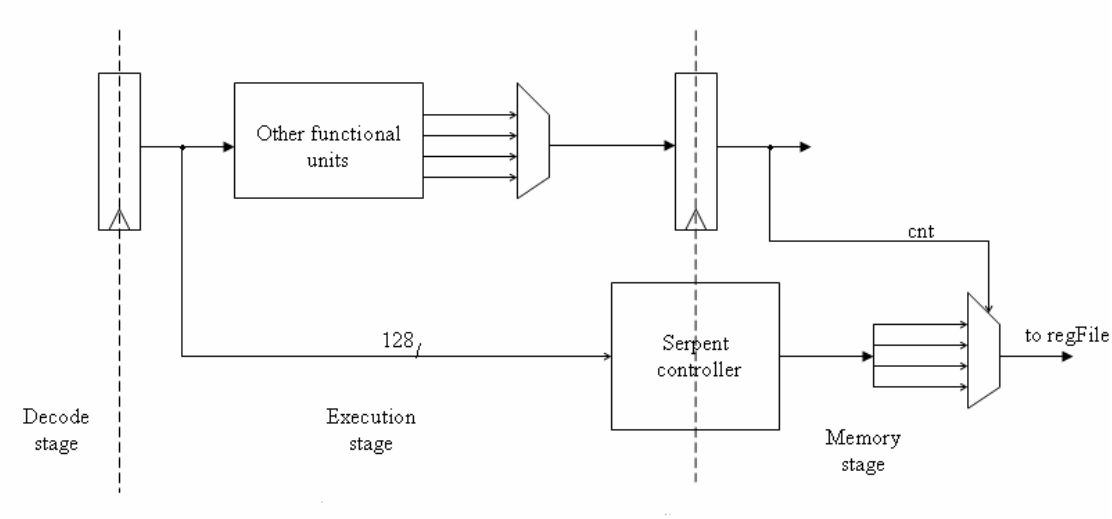



Figure 3.19: Serpent controller in datapath

Like the TSBOX load instruction, Serpent load-phase2 instruction is a multi-cycle instruction. As it is seen from figure 3.19 the result produced from Serpent controller should remain the same, so that all four cycles read the same result produced. This is achieved, by asserting an output enable signal *only* during the first cycle of the multi-cycle instruction. Thus, the value is kept the same until the execution of another Serpent load-phase2 instruction.

During decode phase, we extended the control unit to check the order at which registers would be written. Particularly, the order which is followed is only important if the same register is written twice. In this occasion, register's value is the last value written. It should be noted that registers rd2, rd3 refer to the destination registers of Serpent load-phase2, whereas rd0, rd1 refer to the ones of Serpent load-phase1. The following section of code is added to the control unit (decode stage) and relates to the signals shown in a previous figure (3.16) which focuses on multi-cycle instructions.

```
If ( instr = SERP & load-phase2) {
        write_reg = '1'
        If (cnt = "000") then
                rd = rd2; holdn = '1'; new_cnt = "001";
        else if (cnt = "001") then
                rd=rd3; holdn = '1'; new_cnt = "010";
        else if (cnt = "010") then
```

```
            rd=rd0; holdn = '1'; new_cnt = "011";
        else if (cnt = "011") then
            rd=rd1; holdn = '0'; new_cnt = "000";
}
```

Finally, concerning Serpent instruction, it is not unusual that a destination register is also a source register. That means that the value that is being read during load-phase2 (registers rd2, rd3) should remain the same even if the value of these registers may have changed due to a write of the same instruction. This paradox can happen because the instruction has 4 cycles. The problem was resolved with the use of bypassing. Specifically, registers' values are forwarded from memory stage back to the execution stage during the 3 final cycles of the instruction.

### 3.5.4 Mars cipher Sboxes

MARS algorithm uses two Sboxes S0 and S1, 256x32 each, however it has the unique property that they are also used concatenated as one 512x32 Sbox. Thus, we created a Mars controller that consists of such a dual-port memory. There are four possible options when accessing a Mars load instruction. The type of load is chosen amongst the types shown in Figure 3.20. Before Sbox access there are one or two multiplexers, which in combination with control signals "E", "F" and "B", pass the appropriate byte Depending on cipher's current processing mode, again a control signal "mode" selects the required 32-bit word.

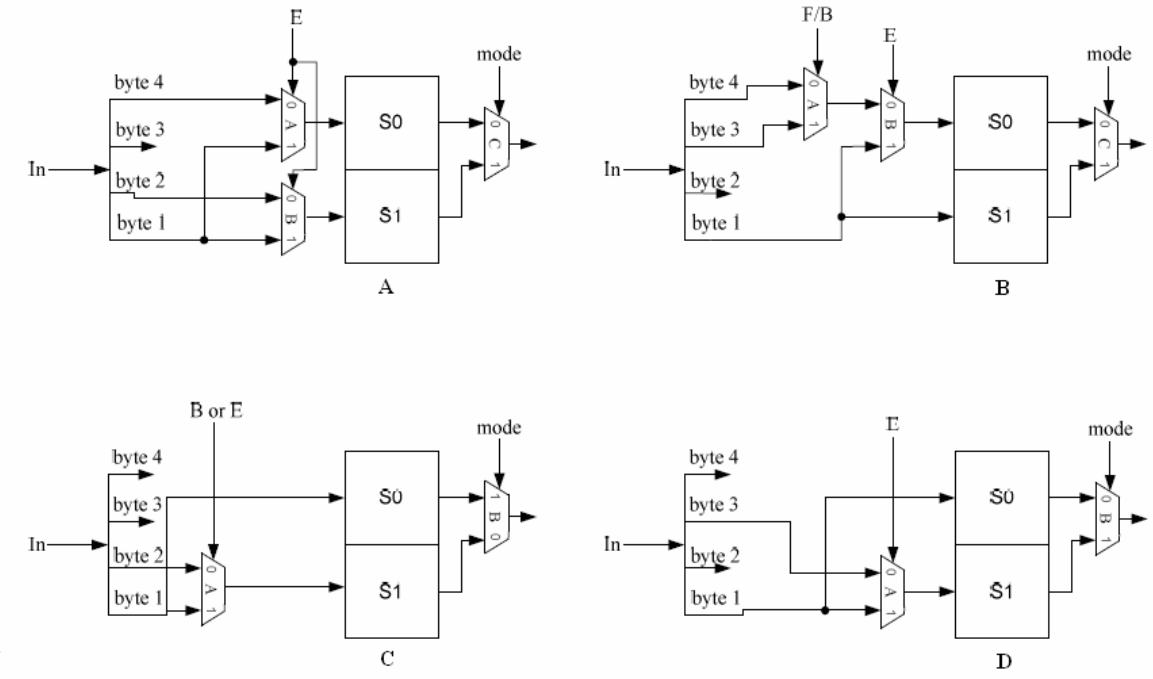


Figure 3.20: Mars Sboxes

To summarize, all cipher instructions need only 1 cycle during store data, and may need from 1 to 4 cycles during the load phase. Specifically Serpent load, is split to two phases. Though, it needs the smallest portion of memory (only 128 bytes), it has the greatest latency from all other cipher instructions due to having to store four destination registers.

| cipher | *AES* | *Twofish* | *Serpent* | *Mars* |
|---|---|---|---|---|
| | | | | |
| **cycles** | 1 | 3 | 4 | 1 |
| **memory (KB)** | 1KB | 512B | 128B | 2KB |

**Table 5 – Summary Cipher Instructions**

During the memory stage, a 2-bit register had to be added to the existing pipeline registers that decides which cipher's controller was accessed.

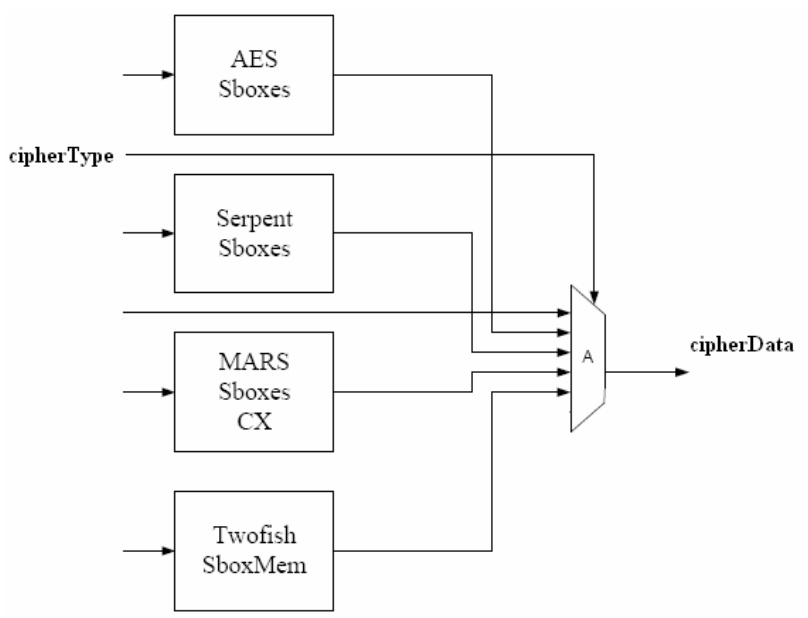



Figure 3.21: Sboxes in memory stage

The last thing that had to be considered was the data dependencies that are created when a cipher load instruction is executed. All cipher load instructions produce their result at the end of the memory stage. This means that are similar with the double

instructions that are referred in chapter 3.3. Just like them, to be able to resolve the data hazard, presented in Figure 3.6a, the same steps are followed. The dependent instruction is stalled in the decode stage for one cycle. Figure 3.6b shows how this stall removes the problem. Data is then forwarded from the write back (WB) pipeline register to the dependent instruction.

Consequently, the operation of the Forwarding Unit had to be extended. This necessitated that the pipeline registers Ex, Mem are extended with one bit register "cipher", which shows that a cipher instruction is executed, regardless of the cipher's type.

# 4. Verification and Performance Evaluation of Leon2_ISE.

## 4.1 Verification Procedure

After the detailed description of the extensions made in Leon2, in this chapter we focus on verification tests that were made in order to confirm the functionality. Specific evaluation of hardware results such as operating frequency and occupied area are examined. Note that for synthesis and implementation we used the Xilinx's ISE Foundation Series 7.1i [8] and for simulation Mentor Graphics Modelsim SE 6.0a [9].

A first prototype has been built based on Xilinx's Virtex 2 FPGAs resources. All memories needed for the Sboxes have been mapped to distributed memory modules [10], usually consisting of dual port memories

The model of Leon2 is distributed as a gzipped tar-file; leon-2.1.30.tar.gz. We unpacked the model in top-level directory. LEON's structure consists of three main sub-directories. The one that contains the source code (VHDL) is leon/leon sub-directory. All testbenches exist in leon/tbench directory, and the data used for simulation is found in leon/tsource sub-directory.

The first step followed, was the compilation of the existing design. This is done, by running the compile.bat files in the leon and tbench directories. Each time a new source file was created, it had to be added correctly to the compilation order inside the compile.bat file. The next step was the simulation of the design, using the Modelsim. A generic test bench that is provided in /leon/tbench directory was used. This file specifies that the contents of memories ROM and SRAM are found in the files ASCII files *rom.dat* and *ram.dat* respectively (inside tsource directory). Particularly, the execution of the processor starts from ROM and after completion of these instructions, the control is transferred to the instructions that exist in SRAM.

As it is shown if Figure 4.1, simulation is performed in two stages For every new insertion of an instruction, functional simulation was used to verify that the results produced (via a series of tests that we will present below) match the expected ones. Post-Place and Route (PAR) simulation was done after the whole designed was

implemented, meaning that all instructions were inserted. Of course, to determine that the system works as expected, post-PAR simulation of the design is needed. The drawback is that functional simulation allows us to observe internal signals and consequently locate possible mistakes. In contrast, only I/O signals of Leon2 are visible during post-PAR simulation, so the only way to confirm that an executable code has succeeded was to use memory-referenced instructions (usually a Store instruction).

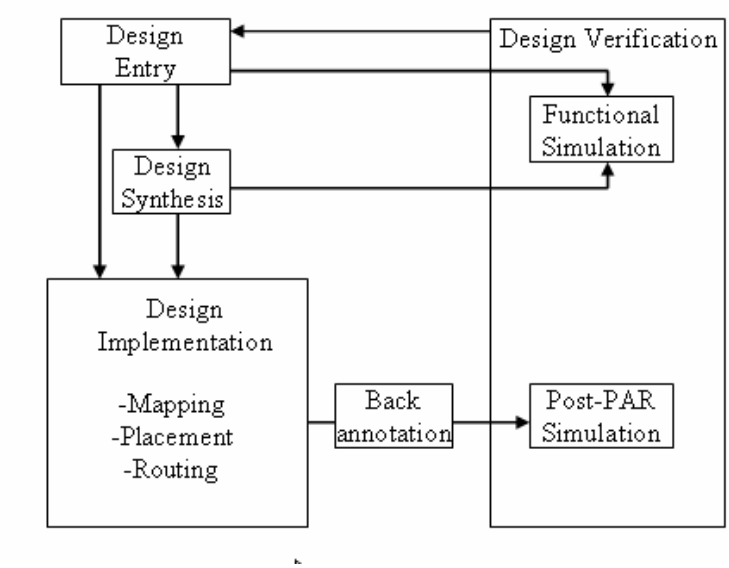



Figure 4.1: Design Flow

Functional simulation of every new instruction individually was our first goal. As we mentioned before, the executable code exists in files rom.dat and ram.dat. In order to able to recognize the instructions, we had to use a disassembler. This tool translates a file of binary machine instructions into a file of assembly language. Although, such a program already existed, we wanted to be able to disassembly the instructions that we would create, so it was decided to built our own disassembler. It was written in C, and the compiler used was Microsoft Visual Studio 6.0. This enabled as to understand the existing code, and to verify the code inserted each time we wanted to check a new instruction.

The main function of the executable code in ROM, is to “initialise” the processor. After doing so, we observed that its final instruction transfers control to a specific instruction in SRAM. We modified that last instruction to transfer control to the starting address of a block of instructions that we created inside the SRAM. The final

instruction of our block, is in turn a control transfer instruction that changes the Program Counter to the starting address of the SRAM. Thus, our block of instructions precedes any other instruction of SRAM.

After this, we were able to insert the machine language to make tests for the instructions we wanted to add. Specifically, the procedure that we followed for every new instruction is presented in the figure below.

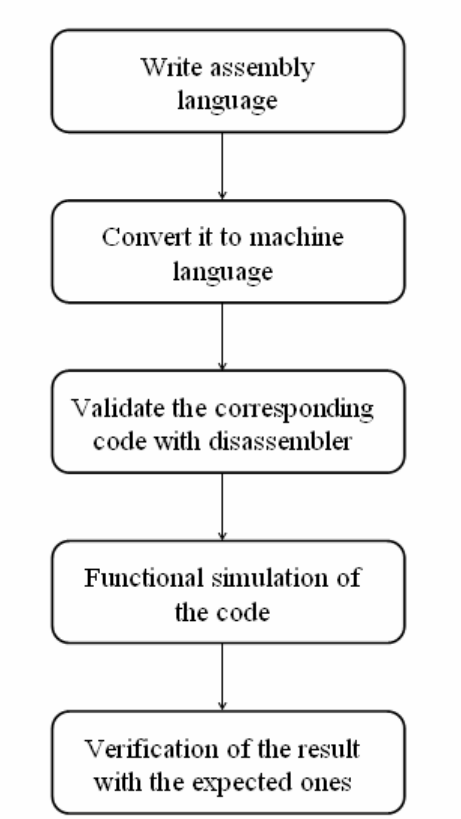



Figure 4.2: Procedure followed for verification

## 4.2 Verification Tests

When the construction of an instruction has been completed, it is imperative that a well defined set of tasks is used to validate the instruction and demonstrate that the system works. So, tests were made at the end of every instruction. The final goal was the verification of the results simulated with the expected results. A simple example for the verification of the double instruction is the below one. The leftmost register is always the destination register, while all remaining registers are the source registers.

```
Initial values (r23 = 800, r22 = 400, r21 = 1)

DBL r23, r23, r22 ,r21  (subsub)     → r23 = (800 – 400) - 1 = 3ff
DBL r23, r23, r22 ,r21  (addxor)     → r23 = (3ff + 400) xor 001 =7ff xor 001 = 7fe
DBL r23, r23, r22 ,r21  (addsub)     → r23 = (7fe + 400) - 1 = bfd
SETHI  r22, '100021'X                → r22 = 40008400
ST r23, r22, '0000'X                 → MEM[r22] = r23 = bfd;
```

As mentioned before the only signals that can give us information of the execution in the integer unit during post-PAR simulation, are the memory-related signals "address" and "data". So, we add a Store instruction, to be able to observe if the last result is correctly computed. The same procedure is followed at the examples presented below. More complicated tests, that include cases of data hazards, are available on the thesis' CD.

### 4.2.1 Twofish – AES example

An example that shows how instructions referring to Sboxes work, is presented. The expected results are verified using the Modelsim simulator. At the beginning of this program, a few store instructions are called in order to initialize the Sboxes. The names "*Q0*", "*Q1*" refer to the Sboxes of Twofish, while "*Encry*" refers to the AES Sbox. Figure 4.3 shows the values of these LUTs. The left part indicates the indices of the tables and the right part their data. All other values are initialized to zero.

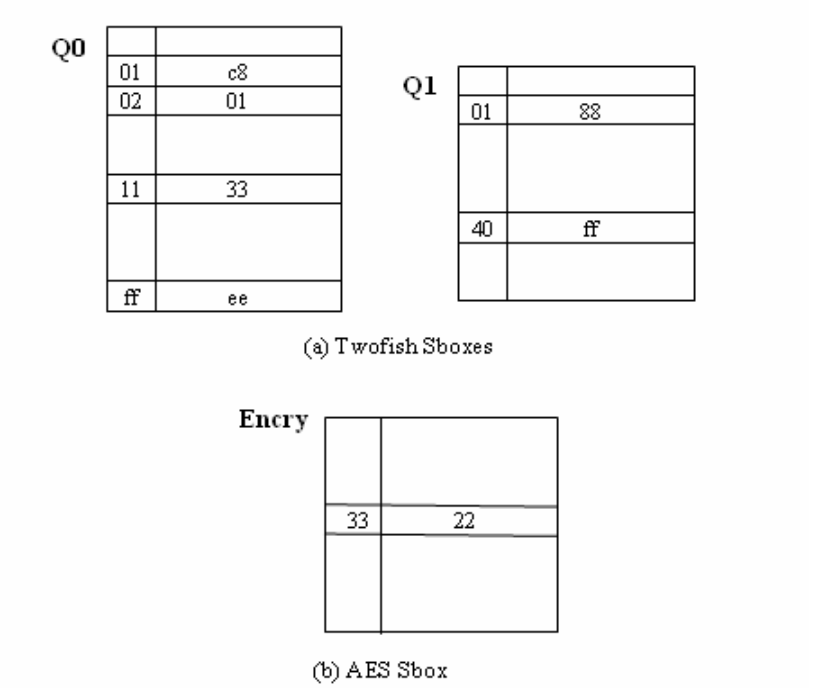



Figure 4.3: Initial Sbox values

```
SETHI  r1, '100000'X
ADD r1, r1, '00cc'X    → r1 = 400000cc
ADD r23, r0, '0322'X
ADD r21, r0, '0033'X
AES r23, r21 ( st, encryption )  → Encry[33] = 22;
TSLD r1, '1002'X    → S0 = r1 = 400000cc,  S1 = sign_ext('1002') = FFFFF102

SETHI  r0, '000000'X  → nop instruction
ADD r21, r0, '0301'X
ADD r22, r0, '07ce'X
TWOFISH r22, r21 (st, q0)   → Q0[01] = 'c8';

SETHI  r0, '000000'X
ADD r21, r0, '0011'X
ADD r22, r0, '0033'X
TWOFISH r22, r21 (st, q0)   → Q0[11] = '33';

SETHI  r0, '000000'X
ADD r21, r0, '0040'X
ADD r22, r0, '00ff'X
TWOFISH r22, r21 (st, q1)   → Q1[40] = 'ff';

SETHI  r20, '001000'X       → r20 = 00400000
ADD r21, r0, '00ff'X
ADD r22, r0, '00ee'X
TWOFISH r22, r21 (st, q0)   → Q0[ff] = 'ee';

ADD r21, r0, '0302'X
ADD r22, r0, '0701'X
TWOFISH r22, r21 (st, q0)   → Q0[02] = '01';

SETHI  r0, '000000'X
ADD r21, r0, '0301'X
ADD r22, r0, '0788'X
TWOFISH r22, r21 (st, q1)   → Q1[01] = '88';
```

We emphasized the meaning of the instructions by typing them in bold. Instruction "TSLD" loads the sub-keys S0 and S1. After these steps are completed, we insert a few instructions that have various data dependencies and contain twofish and Aes load instructions. As it was mentioned before, twofish load lasts for 3 cycles. For better comprehension, we show graphically the steps followed during the two twofish load instructions in figure 4.4, plus the values of the internal results that are produced during twofish instructions that follow (table 6). A is the input (address) and Result is the output.

```
ADD r21, r0, '0301'X
TWOFISH r16, r21  ( ld )        → r16 = Twofish[00 00 03 01] = 00 ee 00 01
TWOFISH r22, r20  ( ld )        → r22 = Twofish[00 40 00 00] = 00 33 00 01
AES r17, r22 ( ld, encryption ) → r17 = Encry[00 33 00 01]; = 00 22 00 00
ADD r22, r21, r22               → r22 = r22 + r21 = 00 33 03 02;
ST r22, r19, '0000'X            → ST Mem[r19] << r22
ST r16, r19, '0004'X            → ST Mem[r19+4] << r16
ST r17, r19, '0008'X            → ST Mem[r19+8] << r17
```

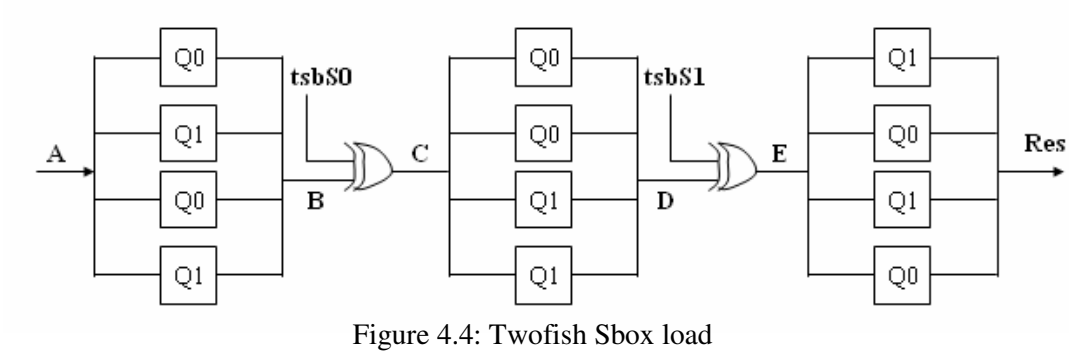



Figure 4.4: Twofish Sbox load

| | *End of cycle 1* | | *End of cycle 2* | | |
|---|---|---|---|---|---|
| **A** | **B** | **C** | **D** | **E** | **Result** |
| **00 00 03 01** | 00 00 00 88 | 40 00 00 44 | 00 00 00 00 | ff ff f1 02 | **00 ee 00 01** |
| **00 40 00 00** | 00 ff 00 00 | 40 ff 00 cc | 00 ee 00 00 | ff 11 f1 02 | **00 33 00 01** |

Table 6: Twofish load internal results

The results produced are stored into registers r16, r17 and r22. In order to validate their values we used 3 store instructions to main memory. The simulation verifies the expected results, shown in figure 4.5.

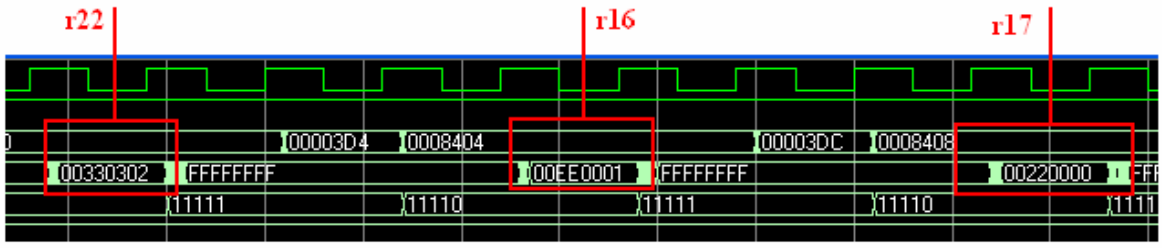



Figure 4.5: PAR simulation

### 4.2.2 AES – Loop – double - rotate example

An example that verifies the operation of the AES instruction as well as the non-cipher instructions that were added to leon2 (rotate, double, loop) is the following. Our simple loop starts from the 4th instruction and ends with the loop instruction. It is run for 3f (hex) times and each time a new value is stored in AES Sboxes (encryption and decryption). Figure 4.6 shows the Sboxes' values after the end of the loop.

```
ADD r22, r0, '00ff'X            → r22 = 00 00 00 ff;
ADD r21, r0, '0000'X            → r21 = 00 00 00 00;
LDLC r0, r0, '003f'X            → lc = 3f; (loop counter)
AES r21, r21  ( st, encryption )  → Encr[r21] = r21;
AES r22, r21  ( st, decryption )  → Decry[r21] = r22;
ADD r21, r21, '0001'X           → r21 ++;
SUB r22, r22, '0001'X           → r22 ++;
LOOP r0, r0, '1ffc'X            → npc = pc – 4 * 4  (back 4 instructions)
```

Encry

| | |
|---|---|
| 00 | 00 |
| 01 | 01 |
| ⋮ | |
| 0f | 0f |
| 10 | 10 |
| ⋮ | |
| 1f | 1f |
| 20 | 20 |
| ⋮ | |
| 3f | 3f |
| 40 | 00 |
| ⋮ | |
| fe | 00 |
| ff | 00 |

Decry

| | |
|---|---|
| 00 | ff |
| 01 | fe |
| ⋮ | |
| 0f | f0 |
| 10 | ef |
| ⋮ | |
| 1f | e0 |
| 20 | df |
| ⋮ | |
| 3f | c0 |
| 40 | 00 |
| ⋮ | |
| fe | 00 |
| ff | 00 |

Figure 4.6: AES sbox values

Following the loop, we access AES' LUTs and use the data produced in arithmetic operations. The results produced are shown in Figure 4.7.

```
SETHI  r19, '100021'X           → r19 = 40 00 84 00;
SETHI  r21, '204bfa'X
ADD r21, r21, '0033'X           → r21 = 81 2f e8 33;
AES r23, r21  ( ld, encryption )  → r23 = Encry[81 1f e8 33] = 00 2f 00 33
AES r22, r21  ( ld, decryption )  → r22 = Decry[81 1f e8 33] = 00 d0 00 cc
ROR r20, r23, r22               → r20 = r23 << c (12bits right) = 03 30 02 f0
DBL r23, r23, r20 ,r22  (xorxor)  → r23 = r23 xor r20 xor r22 = 03 cf 02 0f;
SETHI  r0, '000000'X
ST r23, r19, '0000'X            → ST Mem[r19] << r23
ST r22, r19, '0004'X            → ST Mem[r19+4] << r22
```

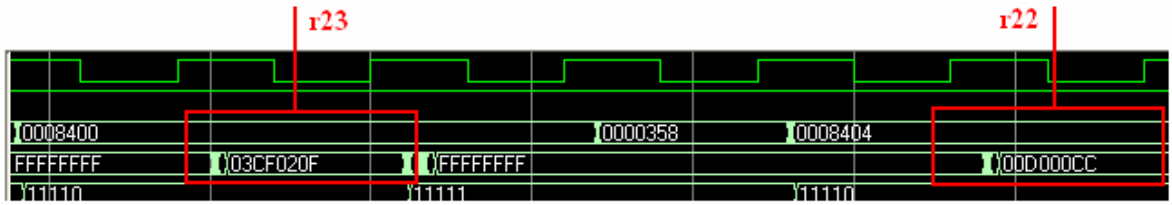



Figure 4.7: PAR simulation

### 4.2.3 Serpent example

Similarly wih the first example, we initialize the Serp Sbox with the values shown in Figure 4.8. Then, we perform a Serpent load instruction and the results are stored in main memory. Figure 4.9 shows the Sbox access for the two least significant bits. The rightmost 30 bits are zeros, so they were neglected. Figure 4.10 shows the verification of the results, in PAR simulation.

Serp

| | |
|---|---|
| 7 | 5 |
| | |
| d | 6 |
| | |

Figure 4.8: Serp Sbox

```
ADD r21, r0, '0007'X
ADD r22, r0, '0005'X
SERP r22, r21  ( st )     → Serp[7]  = 5;
ADD r21, r0, '000d'X  → r21 = d;
ADD r22, r0, '0006'X → r22 = 6;
SERP r22, r21  ( st )     → Serp[d]  = 6;

ADD r21, r0, '0002'X  → r21 = 2;
ADD r22, r0, '0003'X → r22 = 3;
ADD r23, r0, '0001'X  → r23 = 1;
SERP r20 (rd0), r23 (rd1), r21, r22  ( ld (phase1) ) → Serp load phase1
SERP r22 (rd2), r21 (rd3), r23, r22  ( ld (phase2) ) → Serp load phase2 (r20=0,r23 =3,
                                                                  r22 =2, r21 =1)
DBL r24, r23, r22 ,r21  (addsub)        →  r24 = r23 + r22 – r21 = 3 + 2 – 1 = 4

ST r21, r19, '0000'X            → ST Mem[r19] << r21 = 1
ST r23, r19, '0004'X            → ST Mem[r19+4] << r23 = 3
ST r24, r19, '0008'X            → ST Mem[r19+8] << r24 = 4
```

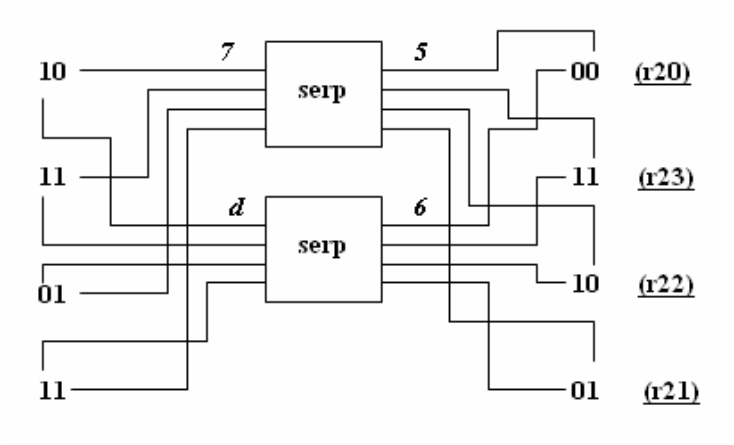



Figure 4.9: Serpent load phase2

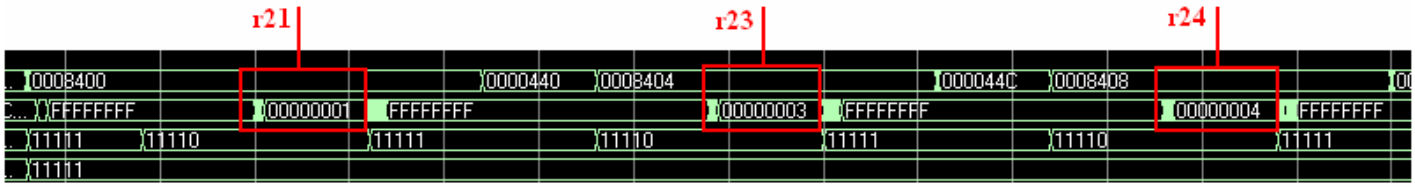



Figure 4.10: PAR simulation

## 4.3 Performance Statistics

This section focuses on the design implementation stage. After making the changes with a hardware description language, the next step is to convert the register transfer level design (which focuses on describing the flow of the signals between registers) to a gate-level description of the circuit by a logic synthesis tool. The synthesis tool used is Xilinx XST (Xilinx Synthesis Tool). The synthesis results are then used by placement and routing tools to create a physical layout. Table 7 shows the results produced for a variety of FPGAs. The rightmost columns show the maximum frequency and utilization before any changes were made. The XC4VLX200 FPGA has the greatest operating frequency for our design at 83,96 MHz. It should be noted that the smallest of these devices that our design may fit is XC3S1500 FPGA with a utilization of 79%.

| FPGA | Speed Grade | Leon_ISE Max.Freq. (MHz) | Utilization | *Leon Max.Freq. (MHz)* | *Utilization* |
|---|---|---|---|---|---|
| xc2v8000 | -5 | 61.94 | 22% | *69.53* | *16%* |
| xc4vlx200 | -11 | 83.96 | 11% | *98,41* | *7%* |
| xc4vlx200 | -10 | 74,34 | 11% | *86,31* | *7%* |
| xc3s5000 | -5 | 49,27 | 31% | *57,44* | *20%* |
| xc3s1500 | -5 | 49,27 | 79% | *57.44* | *58%* |

Table 7 – Performance Statistics

## 4.4 Conclusions and Future Work

As we mentioned in the beginning of this thesis report, internet is growing larger over the years that pass. Also various embedded processors are more and more used in many wireless communications devices, such as cell phones, PDAs (Personal Digital Assistants), televisions. Consequently, secure communication and confidentiality are crucial, in order to avoid virus infection, privacy loss, and stop digital crime activities from malicious users.

Cryptography is a major issue, which should be taken into account when designing new processors that will be used for communication and data exchange. This project was focused on enhancing the existing ISA of Leon2 with instructions that may improve the efficiency and the performance of cryptographic algorithms. In summary our design has added the following characteristics:

- Instructions such as Rotate and Loop were added to the ISA.
- Increased the number of register file's (RF) read ports to three. This was done, so that double instructions may be calculated.
- Supports different Sbox structures for a variety of ciphers.
- Fits in FPGAs, without a great decrease in maximum operating frequency

In addition, there are a few potential improvements that can be made:

- Better use of the third port in RF, to utilize instructions that require three input operands.

- Addition of the Galois Field Multiplication, an operation commonly used by the AES cipher.
- Calculation of the performance that this design achieves, and comparison with pure software implementation (assembly).

In summary, cipher algorithms have been developing through the years and they will always be, while concurrently many ways are being discovered to unlock even the securest ones. For this reason, new designs should be developed to protect people from malicious users.

# 5. References

[1] "The *SPARC Architecture Manual, Version 8*", SPARC International Inc., 1992.

[2] Graham Seaman: "*Free Hardware: Past, Present & Future*", Erste Oekonux Konferenz, 2002

[3] John L. Hennessy, David A. Patterson: "*Computer Architecture, a Quantitative Approach*", Ed. Mc-Graw-Hill 1993.

[4] "*AMBA (tm) Specification, Rev. 2.0*", ARM Limited, 1999. http://www.arm.com/

[5] James F.Kurose, Keith W.Ross: "*Computer Networking: A Top-Down Approach Featuring the Internet*", Prentice Hall 2003

[6] Theodoropoulos Dimitris,*"CCproc: A custom VLIW cryptography co-processor for symmetric key ciphers",*2005.

[7] LEON Sparc v8 CPU Core, Gaisler Research, http://www.gaisler.com/

[8] http://www.xilinx.com

[9] http://www.model.com

[10] Xilinx Corporation, "*Distributed memory v7.1*", January 2005

[11]B. Schneier, *Applied Cryptography*, John Wiley & Sons Inc., New York, New York, USA, 2nd edition, 1996.

[12] R.Doud, "Hardware Crypto Solutions Boost VPN", Electronic Engineering Times, 1999.

[13] AJ Elbirt, "Fast and Efficient Implementation of AES Via Instruction Set Extensions", 2007.

[14] J. Burke, J. McDonald, T. Austin, "*Architectural Support for Fast Symmetric-Key Cryptography*", ASPLOS 2000

[15] A. Murat Fiskiran and Ruby B. Lee, "*Performance Impact of Addressing Modes on Encryption Algorithms*", Proceedings of the International Conference on Computer Design (ICCD 2001), pp. 542-545, September 2001

[16] A. Murat Fiskiran and Ruby B. Lee, "*On-Chip Lookup Tables for Fast Symmetric-Key Encryption*", Proceedings of the IEEE 16th International Conference

on Application-Specific Systems, Architectures and Processors (ASAP), pp. 356-363, July 23-25, 2005

[17] A. Murat Fiskiran and Ruby B.Lee, "Fast Parallel Table Lookups to Accelerate Symmetric-Key Cryptography", 2005

[18] Alireza Hodjat, Ingrid Verbauwhede, "*A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA*", IEEE Symposium on Field-Programmable Custom Computing Machines, April 2004

[19] Xilinx Corporation, "*Virtex-II Pro and Virtex-II Pro X Platform FPGAs*", March 2005

[20] Lisa Wu, Chris Weaver, and Todd Austin, "*Cryptomaniac: A Fast Flexible Architecture for Secure Communication*", ISCA 2001, June 2001

[21] Katherine Compton, Scott Hauck, "Reconfigurable Computing: A survey of Systems and Software", ACM Computing Surveys (CSUR), 2002

# Appendix A: Leon2's Instruction Set Extension

## 1. Iror/Irol


Rotate instructions format

| Suggested Assembly Language Syntax | |
|---|---|
| Ror | reg(rs1), reg(rs2) or imm, reg(rd) |
| Rol | reg(rs1), reg(rs2) or imm, reg(rd) |

Description :

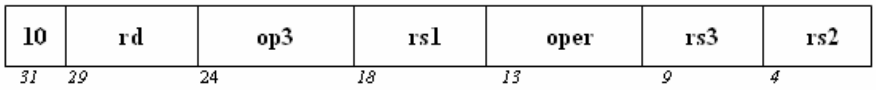
The rotate count for these instructions is the least significant five bits of r[rs2] if the i field is zero, or the value in rotcnt if the i field is one.
When i is 0, the most significant 27 bits of the value in r[rs2] are ignored.
Ror rotates the value of r[rs1] right by the number of bits given by the rotate count.
Rol rotates the value of r[rs1] left by the number of bits given by the rotate count.

No shift occurs when the rotate count is zero. These instructions do **not** modify the instructions code

## 2. Double Instructions

| instr | oper | operation1 | operation2 |
|---|---|---|---|
| ADDADD | 0000 | ADD | ADD |
| ADDSUB | 0001 | ADD | SUB |
| ADDXOR | 0010 | ADD | XOR |
| SUBADD | 0100 | SUB | ADD |
| SUBSUB | 0101 | SUB | SUB |
| SUBXOR | 0110 | SUB | XOR |
| XORADD | 0011 | XOR | ADD |
| XORSUB | 0111 | XOR | SUB |
| XORXOR | 1000 | XOR | XOR |

/



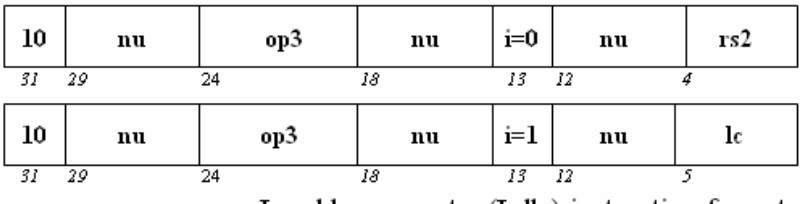

Double instructions format

| Suggested Assembly Language Syntax | |
|---|---|
| *Double* | reg(rs1), reg(rs2), reg(rs3), reg(rd) |

Description :

These instructions calculate the result of the operations defined in **oper** field. They compute r[rs1] **operation1** r[rs2] **operation2** r[rs3] and write the result into r[rd].

## 3. Ldlc/Iloop




Load loop counter (Ldlc) instruction format

| Suggested Assembly Language Syntax | |
|---|---|
| Ldlc | reg(rs2) or lc |

Description :

TSLD stores r[rs2(5:0)] if the i field is zero, or the lc field if the i field is one to loop counter register (lc).


Loop instruction format

| Suggested Assembly Language Syntax |
|---|
| Loop *label* |

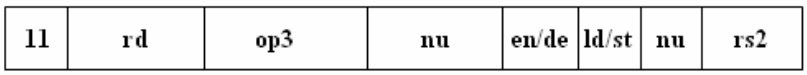
Description :

The LOOP instruction is a conditional Control Transfer Instruction. It evaluates the loop counter register (lc). If it is greater than zero, then the branch is taken, that is, the instruction causes a PC-relative delayed control transfer to the address "PC + (4 x sign_ext(*imm13*))". Else, the branch is not taken.

## 4. cipher instructions

| | | op3 [5:4] | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 |
| op3 [3:0] | 0 | LD | LDA | LDF | LDC |
| | 1 | LDUB | LDUBA | LDFSR | LDCSR |
| | 2 | LDUH | LDUHA | | |
| | 3 | LDD | LDDA | LDDF | LDDC |
| | 4 | ST | STA | STF | STC |
| | 5 | STB | STBA | STFSR | STCSR |
| | 6 | STH | STHA | STDFQ | STDCQ |
| | 7 | STD | STDA | STDF | STDC |
| | 8 | AES | | | |
| | 9 | LDSB | LDSBA | | |
| | A | LDSH | LDSHA | | |
| | B | TSBOX | | | |
| | C | SERP | | | |
| | D | LDSTUB | LDSTUBA | | |
| | E | MARS | | | |
| | F | SWAP | SWAPA | | |

## 5. Aes




Aes instruction format

| Suggested Assembly Language Syntax | |
|---|---|
| Aes | reg(rs2), reg(rd), st/ld, en/de |

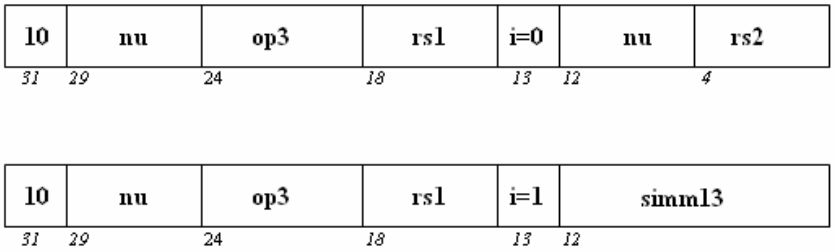
Description :

AES has a load and a store choice. Load moves a word from aes cache controller into register r[rd]. Store saves the least significant byte of register r[rd] into the two copies of encryption's or decryption's cache address.

The effective read address is r[rs2]. The effective write address is the least significant byte of r[rs2].

Bit 9 determines which of either the encryption or the decryption cache to use.

## 6. Tsld/TSBOX




Tsld instruction format

| Suggested Assembly Language Syntax | |
|---|---|
| Tsld | reg(r1), reg(r2) or imm |

Description :

TSLD stores r[rs1] to twofish sub-key zero (tsbS0) and r[rs2] if the i field is zero, or sign_ext(simm13) if the i field is one to twofish sub-key one (tbbS1).


Twofish instruction format

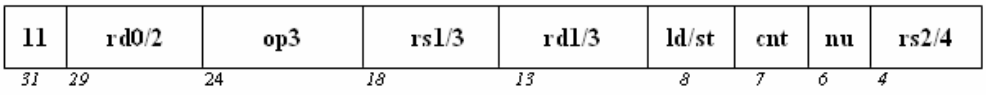
| Suggested Assembly Language Syntax | |
|---|---|
| tsbox | reg(rd), reg(rs2) ,st/ld |

Description :

TSBOX has a load and a store choice. Load saves the result produced from twofish Sbox procedure into register r[rd]. Store copies the least significant byte of register r[rd] into q0/1 cache depending on instruction bit 9.

The effective read address of the sbox is r[rs2]. The effective write address is the least significant byte of r[rs2].

## 7. SERP




Serpent instruction format

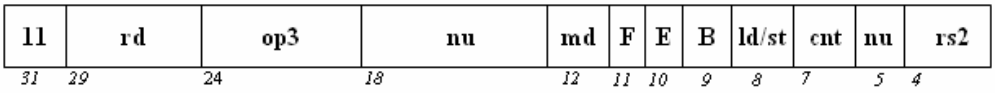
| Suggested Assembly Language Syntax | |
|---|---|
| serpX (st) | reg(rd), reg(r2) |
| serpX (ld1) | reg(rd0), reg(rd1), reg(rs1), reg(rs2) |
| serpX (ld2) | reg(rd2), reg(rd3), reg(rs3), reg(rs4) |

Description :

SERPX has a load and a store choise. Load has two phases determined by the cnt bit (0 first, 1 second phase) in order to read the 4 words which are required. The result produced from serpent Sbox procedure in the second phase of the load command has a length of 128 bits (4 words), so there is 3-cycle stall to save into registers r[rd0-rd3] the appropriate word for each one. Store copies the 4 least significant bits of register r[rd] into the appropriate serpent cache.

The effective read address of the sbox are the four registers r[rs1-rs4]. The effective write address is the least significant four bits of r[rs2] .

# 8. Mars




Mars instruction format

| Suggested Assembly Language Syntax | |
|---|---|
| MarsX | reg(rd), reg(rs2) , st/ld, X = E, F, S, mode |

Description :

MARS has a load and a store choise. Load moves a word from mars cache into register r[rd]. The type of load is chosen among 4 given types and it is determined by the **cnt** bits. There are the control signals "**E**", "**F**" and "**B**", as well as the bit **md** which is the cipher's current processing mode and decides the output from two Sboxes S0, S1. Store copies the word in register r[rd] into mars cache address.

The effective address is r[rs2(7:0)]. Bit 8 determines if it is a load or a store choice.

# Appendix B: Leon's complete Instruction Set

| *Opcode* | *Name* |
|---|---|
| LDSB (LDSBA†) | Load Signed Byte (from Alternate space) |
| LDSH (LDSHA†) | Load Signed Halfword (from Alternate space) |
| LDUB (LDUBA†) | Load Unsigned Byte (from Alternate space) |
| LDUH (LDUHA†) | Load Unsigned Halfword (from Alternate space) |
| LD (LDA†) | Load Word (from Alternate space) |
| LDD (LDDA†) | Load Doubleword (from Alternate space) |
| LDF | Load Floating-point |
| LDDF | Load Double Floating-point |
| LDFSR | Load Floating-point State Register |
| LDC | Load Coprocessor |
| LDDC | Load Double Coprocessor |
| LDCSR | Load Coprocessor State Register |
| STB (STBA†) | Store Byte (into Alternate space) |
| STH (STHA†) | Store Halfword (into Alternate space) |
| ST (STA†) | Store Word (into Alternate space) |
| STD (STDA†) | Store Doubleword (into Alternate space) |
| STF | Store Floating-point |
| STDF | Store Double Floating-point |
| STFSR | Store Floating-point State Register |
| STDFQ† | Store Double Floating-point deferred-trap Queue |
| STC | Store Coprocessor |
| STDC | Store Double Coprocessor |
| STCSR | Store Coprocessor State Register |
| STDCQ† | Store Double Coprocessor deferred-trap Queue |
| LDSTUB (LDSTUBA†) | Atomic Load-Store Unsigned Byte (in Alternate space) |
| SWAP (SWAPA†) | Swap r Register with Memory (in Alternate space) |
| SETHI | Set High 22 bits of r Register |
| NOP | No Operation |
| AND (ANDcc) | And (and modify icc) |
| ANDN (ANDNcc) | And Not (and modify icc) |
| OR (ORcc) | Inclusive-Or (and modify icc) |
| ORN (ORNcc) | Inclusive-Or Not (and modify icc) |
| XOR (XORcc) | Exclusive-Or (and modify icc) |
| XNOR (XNORcc) | Exclusive-Nor (and modify icc) |
| SLL | Shift Left Logical |
| SRL | Shift Right Logical |
| SRA | Shift Right Arithmetic |

| | |
|---|---|
| ADD (ADDcc) | Add (and modify icc) |
| ADDX (ADDXcc) | Add with Carry (and modify icc) |
| TADDcc (TADDccTV) | Tagged Add and modify icc (and Trap on overflow) |
| SUB (SUBcc) | Subtract (and modify icc) |
| SUBX (SUBXcc) | Subtract with Carry (and modify icc) |
| TSUBcc (TSUBccTV) | Tagged Subtract and modify icc (and Trap on overflow) |
| MULScc | Multiply Step (and modify icc) |
| UMUL (UMULcc) | Unsigned Integer Multiply (and modify icc) |
| SMUL (SMULcc) | Signed Integer Multiply (and modify icc) |
| UDIV (UDIVcc) | Unsigned Integer Divide (and modify icc) |
| SDIV (SDIVcc) | Signed Integer Divide (and modify icc) |
| SAVE | Save caller's window |
| RESTORE | Restore caller's window |
| Bicc | Branch on integer condition codes |
| FBfcc | Branch on floating-point condition codes |
| CBccc | Branch on coprocessor condition codes |
| CALL | Call and Link |
| JMPL | Jump and Link |
| RETT | Return from Trap |
| Ticc | Trap on integer condition codes |
| RDASR | Read Ancillary State Register |
| RDY | Read Y Register |
| RDPSR | Read Processor State Register |
| RDWIM | Read Window Invalid Mask Register |
| RDTBR | Read Trap Base Register |
| WRASR | Write Ancillary State Register |
| WRY | Write Y Register |
| WRPSR | Write Processor State Register |
| WRWIM | Write Window Invalid Mask Register |
| WRTBR | Write Trap Base Register |
| STBAR | Store Barrier |
| UNIMP | Unimplemented |
| FLUSH | Flush Instruction Memory |
| FPop | Floating-point Operate |
| CPop | Coprocessor Operate: implementation-dependent |