University of Western Macedonia

Thesis Work

# Design and Implementation of a Soft Processor with a Custom FPU Addition

*Author:*
*Angelos-Eystathios Ntasios*

*Supervisor:*
*Minas Dasygenis*

Department of Informatics and telecommunications

July 7, 2014

# Declaration of Authorship

I, Angelos-Eystathios Ntasios, declare that this thesis titled, "Design, Implementation and Verification of a Soft Processor with FPU Support" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

*"I would like to thank my family for their support during my academic years. I would also like to express my gratitude for my supervising professor, Dr. Minas Dasygenis, without whom I would not be able to fulfil this thesis."*

Angelos Ntasios

# *Abstract*

The ever growing need for flexibility and low production cost in hardware implementations, has led to a wider use of reprogrammable and reconfigurable hardware such as PLDs and FPGAs, which can be programmed with hardware description languages. Using reconfigurable hardware provides the option to customize existing soft-cores and soft processors in order to adapt to different design requirements.

In this thesis, an implementation of a processor based on the PLX 1.1 instruction set is presented. Since the processor is intended for multimedia data processing, it is necessary to include a floating point arithmetic unit. All the required steps that had to be taken in order to embed a floating point unit in the processor are described in detail. The whole design and implementation process of the soft core microprocessor as well as the FPU are presented, along with the customization by embedding the double precision FPU. All the stages of the work are accompanied by simulation results and FPGA implementation metrics.

# *Περίληψη*

Η ολοένα και αυξανόμενη ανάγκη για ευελιξία και χαμηλό κόστος παραγωγής σε υλοποιήσεις υλικού, οδήγησε σε μία ευρύτερη χρήση επαναπρογραμματιζόμενου και επαναδιαμορφούμενου εξοπλισμού, όπως FPGAs και PLDs, τα οποία μπορούν να προγραμματιστούν με γλώσσες περιγραφής υλικού. Η χρήση του επαναπρογραμματιζόμενου υλικού προσφέρει την επιλογή της προσαρμογής ήδη υπαρχόντων επεξεργαστών έτσι ώστε να προσαρμοτούν σε διαφορετικές σχεδιαστικές ανάγκες.

Σε αυτή την πτυχιακή, παρουσιάζεται μία υλοποίηση ενός επεξεργαστή που βασίζεται στο σύνολο εντολών PLX 1.1 . Από τη στιγμή που ο επεξεργαστής προορίζεται για επεξεργασία δεδομένων πολυμέσων , είναι αναγκαίο να περιλαμβάνει και μονάδα επεξεργασίας αριθμών κινητής υποδιαστολής. Όλα τα απαραίτητα βήματα που έπρεπε να παρθούν έτσι ώστε να προσαρμόσουμε μία τέτοια μονάδα περιγράφονται λεπτομερώς. Η όλη σχεδίαση και υλοποίηση του επεξεργαστή και της μονάδας επεξεργασίας αριθμών κινητής υποδιαστολής παρουσιαζονται, μαζί με την διαδικασία προσαρμογής του επεξεργαστή έτσι ώστε να μπορέσει να λειτουργήσει με τη νέα μονάδα. Όλα τα στάδια της εργασίας συνοδεύονται με αποτελέσματα προσομοίωσης και με στατιστικά από την υλοποίηση σε FPGA.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **ALU** | **A**rithmetic **L**ogic **U**nit |
| **FPU** | **F**loating **P**oint arithmetic **U**nit |
| **FPGA** | **F**ield **P**rogrammable **G**ate **A**rray |
| **HDL** | **H**ardware **D**escription **L**anguage |
| **ISA** | **I**nstruction **S**et **A**rchitecture |
| **LSB** | **L**east **S**ignificant **B**it |
| **MSB** | **M**ost **S**ignificant **B**it |
| **MHz** | **M**ega **H**eartz |
| **PC** | **P**rogram **C**ounter |
| **PLD** | **P**rogrammable **L**ogic **D**evice |
| **PROM** | **P**rogrammable **R**ead **O**nly **M**emmory |
| **VLSI** | **V**ery **L**arge **S**cale **I**ntegration |

# Chapter 1

# Introduction

The modern trend in digital circuits and modern data processing, is gradually moving away from inflexible and big circuits such as desktop computer processors and is rapidly turning to more compact, adaptable and cheaper implementations. The field of embedded systems focuses in the construction of a computer system with a dedicated function, which is a crucial component of a larger installment, i.e. it is embedded within a larger system[1]. Usually these systems are in the form of micro-controllers which are processors along with all the required peripheral units, such as memory and external chips, already installed.

## 1.1 Embedded Systems

The key factor in embedded systems is their flexibility. Engineers can adapt the design very easily in order to meet certain requirements. These requirements can come in the form of area or power restrictions. Since these installments are often used in areas where the environment places a lot of restrictions, these easily adjustable systems are the go-to choice. As it is clearly evident the utilization spectrum is vast, ranging from cars and household equipment to satellites and space shuttles[2] Figure1.1.

The embedded systems design requires fast system prototyping to validate the feasibility of various implementations and to guide toward the optimal selection. In this context, almost every embedded system engineer has at his arsenal one or more re-programmable FPGA hardware circuits. Re-programmable hardware is a term used to describe a

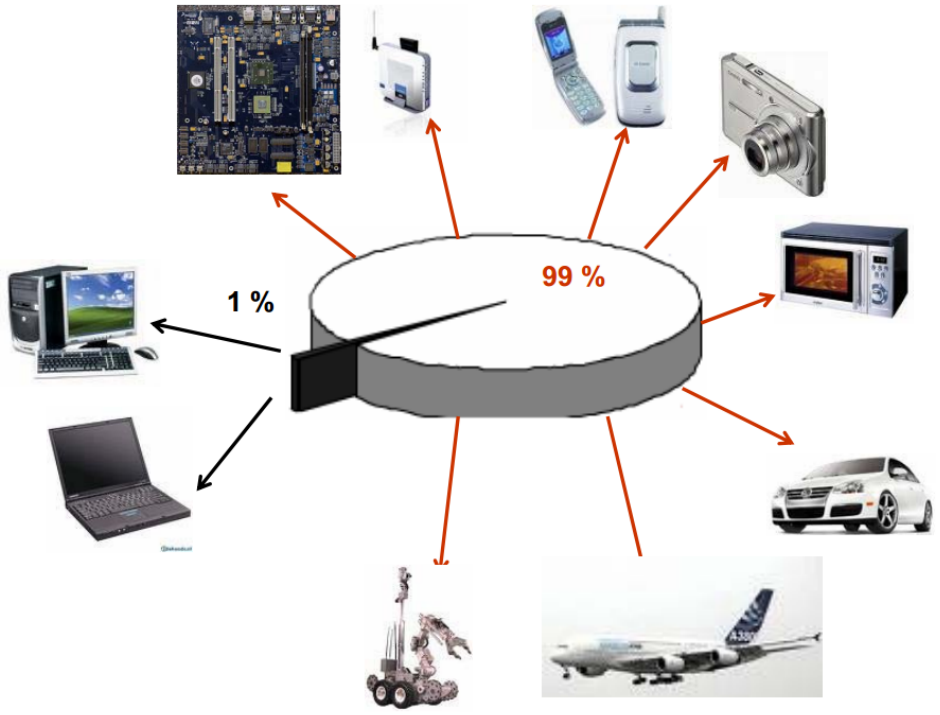


FIGURE 1.1: Embedded Systems market share

certain class of circuits. These circuits do not implement any specific function, but they have the potential to implement any hardware block, according to size constrains. They can be described as vast arrays of circuit blocks, each capable of functioning in various ways. By providing a specific bitstream configuration, the complete array can operate as the desired circuit e.g. processor, signal processing unit and many others. In fact there is almost no limit as to what can be implemented on these devices. Compared to the application specific integrated circuits (ASICs), they are more flexible, providing lower engineering cost, reduced development time, reduced debugging time and lower implementation risk .

The usage spectrum of FPGAs is mainly focused in areas where flexibility, low cost and rapid prototyping is mandatory. Such examples vary from radio-astronomy and particle physics to chip multiprocessor emulation and derivative pricing. All those fields have a common factor, which is the dynamic environment and alternating circumstances. It is clear that in such situations adapting, combining and incorporating IP-cores to large scale projects is mandatory.

## 1.2 Intellectual Property Cores

With the advancement of the embedded systems market, the complexity of many designs has increased since an even wider variety of peripherals and components had to be supported. FPGAs play a major role in this advancement since the flexibility they offer allows the engineer to constantly adapt and improve his designs in order fit to different circumstances. Many modules designed and created in HDLs , are often created for a similar purpose. The ease of use offered by software allows users to easily share and even sell their work, in this case software described hardware. This gave rise to the IP-core market.

IP-cores are modules written in a hardware description language implementing a certain circuit. They can be reused and acquired either commercially or free of charge and can be used as an embedded feature in another design. The increasing complexity of the designs has rendered the usage of IP-cores a viable alternative[3]. One can simply acquire an IP-core and use it at will in his work. For example one can buy a random number generator and use it as is, in an implementation for network security. The advantages of using IP-cores are huge, as they are a solid answer to the problem of time-to-market reduction, which requires the fastest possible design and creation of a project. This is achieved because by using a preprogrammed and pretested module, there is no need of designing and debugging.

IP-cores in the electronic design industry have had a profound impact on the design of systems on a chip. By licensing a design multiple times, an IP core licensor spreads the cost of development among multiple chip makers. IP cores for standard processors, interfaces, and internal functions have enabled chip makers to put more of their resources into developing the differentiating features of their chips. As a result, chip makers have developed innovations more quickly.

The licensing and use of IP-cores in chip design came into common practice in the 1990s. There were many licensors and also many foundries competing on the market. Today, the most widely licensed IP-cores are from Synopsys, Imagination Technology, Cadence Design and ARM Holdings[4].

Typically IP-cores are offered as Soft cores or Hard cores. Soft cores are offered as synthesizable RTL designs, usually in a HDL such as Verilog or VHDL. IP cores are

also sometimes offered as generic gate-level netlists. The netlist is a boolean-algebra representation of the IP's logical function implemented as generic gates or process specific standard cells. An IP core implemented as generic gates is portable to any process technology. A gate-level netlist is analogous to an assembly-code listing in the field of computer programming. A netlist gives the IP core vendor reasonable protection against reverse engineering. Both netlist and synthesizable cores are called "soft cores", as both allow a synthesis, placement and route (SPR) design flow. Hard cores on the other hand are offered by their physical low-level description, thus making them more predictable in terms of timing performance and area utilization.Such cores, whether analog or digital, are called "hard cores" (or hard macros), because the core's application function cannot be meaningfully modified by chip designers. Transistor layouts must obey the target foundry's process design rules, and hence, hard cores delivered for one foundry's process cannot be easily ported to a different process or foundry.

## 1.3 Soft Microprocessors

As mentioned earlier, embedded systems are a vital part of our everyday interaction with technology. These systems often have special restrictions, mainly in power consumption and cost. Usually these systems contain embedded processors either in the form of micro-controllers or soft processors. In addition to these constraints, many embedded system developers are faced with tight timeto- market deadlines. Hence, the hardware/software codesign methodology is often used to design embedded systems in order to help reduce the amount of time spent on development and debugging. Soft microprocessor is a microprocessor solely implemented using logic synthesis and HDLs. They can be implemented in varisous reprogrammable semiconductor devices such as FPGAs, PLDs, ASICs. The use of soft-core processors holds many advantages for the designer of an embedded system. First, soft-core processors are flexible and can be customized for a specific application with relative ease. Second, since soft-core processors are technology independent and can be synthesized for any given target ASIC or FPGA technology, they are therefore more immune to becoming obsolete when compared with circuit or logic level descriptions of a processor. Finally, since a softcore processor's architecture and behavior are described at a higher abstraction level using an HDL, it becomes much easier to understand the overall design.[3]

### 1.3.1 The Picoblaze Soft Microprocessor

Picoblaze is a soft microprocessor offered by Xilinx. It is based on an 8-bit RISC architecture and can reach speeds up to 100 MIPS on the Virtex 4 FPGA's family. The processor has an 8-bit address and data port for access to a wide range of peripherals. The license of the cores allows their free use, albeit only on Xilinx devices, and they come with development tools. The PicoBlaze microcontroller core is totally embedded within the target FPGA and requires no external resources. The basic functionality is easily extended and enhanced by connecting additional FPGA logic to the microcontroller's input and output ports. the PicoBlaze peripheral set can be customized to meet the specific features, function, and cost requirements of the target application. Because the PicoBlaze microcontroller is delivered as synthesizable VHDL source code, the core is future-proof and can be migrated to future FPGA architectures, effectively eliminating product obsolescence fears. Being integrated within the FPGA, the processor reduces board space, design cost, and inventory[5].

### 1.3.2 The Microblaze Soft Microprocessor

MicroBlaze is a 32-bit RISC Harvard architecture soft processor core which is created and offered for free by Xilinx. They key feature of this processor is its high customizability and flexibility. MicroBlaze contains over 70 user-configurable options, enabling virtually any processor use case from a very small footprint state machine or microcontroller to a high performance compute-intensive micrprocessor-based system running Linux, operating in either 3-stage pipeline mode to optimize size, or 5-stage pipeline mode to optimize speed. These parameters include an optional IEEE-754 compatible single precision FPU, a hardware divider, a barrel shifter, data and instruction caches, exception handling capabilities, hardware debug logic and many more[3]. Both Picoblaze and Microblaze come freely with the Embedded Development Kit (EDK) which is offered from Xilinx. Both of these processors are small and focus mainly in flexibility and low power consumption.

### 1.3.3 The Xtensa Microprocessors

Tensilica's Xtensa technology provides a variety of soft processors for embedded systems. This processor family is focused mainly in flexibility and customizability. A series of options is offered from which to choose the ones that are needed. One important key feature is that they are extensible in that custom instructions and custom units can be added to the processor. This feature is supported by a custom Verilog language created by Tensilica, the Tensilica Instruction Extension, which is used to describe the new instructions[3]. An automated HDL generator is also available for this processor, which creates HDL modules of the customized processor.

### 1.3.4 LEON Microprocessor

LEON is a 32-bit CPU microprocessor core, based on the SPARC-V8 RISC architecture and instruction set. It was originally designed by the European Space Research and Technology Centre (ESTEC), part of the European Space Agency (ESA), and after that by Gaisler Research. It is described in synthesizable VHDL. LEON has a dual license model: An LGPL/GPL FLOSS license that can be used without licensing fee, or a proprietary license that can be purchased for integration in a proprietary product. The core is configurable through VHDL generics, and is used in system-on-a-chip (SOC) designs both in research and commercial settings.

The LEON project was started by the European Space Agency (ESA) in late 1997 to study and develop a high-performance processor to be used in European space projects. The objectives for the project were to provide an open, portable and non-proprietary processor design, capable to meet future requirements for performance, software compatibility and low system cost. Another objective was to be able to manufacture in a Single event upset (SEU) sensitive semiconductor process. To maintain correct operation in the presence of SEUs, extensive error detection and error handling functions were needed. The goals have been to detect and tolerate one error in any register without software intervention, and to suppress effects from Single Event Transient (SET) errors in combinational logic.

The LEON family includes the first LEON1 VHSIC Hardware Description Language (VHDL) design that was used in the LEONExpress test chip developed in 0.25 µm

technology to prove the fault-tolerance concept. The second LEON2 VHDL design was used in the processor device AT697 from Atmel (F) and various system-on-chip devices. These two LEON implementations were developed by ESA. Gaisler Research, now Aeroflex Gaisler, developed the third LEON3 design and has announced the availability of the fourth generation LEON, the LEON4 processor[6].

### 1.3.5 The OpenRISC Microprocessor

OpenRISC is the original flagship project of the OpenCores community. This project aims to develop a series of general purpose open source RISC CPU architectures. The first (and currently only) architectural description is for the OpenRISC 1000, describing a family of 32 and 64-bit processors with optional floating point and vector processing support.

A team from OpenCores provided the first implementation, the OpenRISC 1200, written in the Verilog hardware description language. The hardware design was released under the GNU Lesser General Public License (LGPL), while the models and firmware were released under the GNU General Public License (GPL). A reference SoC implementation based on the OpenRISC 1200 was developed, known as ORPSoC (the OpenRISC Reference Platform System-on-Chip).

The instruction set is a reasonably simple MIPS-like traditional RISC using a 3-operand load-store architecture, with 16 or 32 general-purpose registers and a fixed 32-bit instruction length. The instruction set is mostly identical between the 32 and 64 bit versions of the specification, the main difference being the register width (32 or 64 bits) and pagetable layout. The OpenRISC specification includes all features common to modern desktop/server processors: a supervisor mode and virtual memory system, optional read, write and execute control for memory pages, and instructions for synchronization and interrupt handling between multiple processors[7].

## 1.4 Floating point arithmetic

In computer science the programs or the circuits that are implemented to deal wih calculations, always deal with natural numbers. Since the fundamental basis of the

calculations is the bit, all other numbers such as rational, fractions and irrational are represented, and often represented by an approximation, by real numbers. As it is clear a standard had to be established in order to represent these numbers so as to be used in computer science. The term floating point refers to the fact that a number's radix point in computers, can "float". That is, it can be placed anywhere relative to the significant digits of the number. This position is indicated as the exponent component in the internal representation, and floating point can thus be thought of as a computer realization of scientific notation. All floating point numbers are represented with the following formula:

$$SignificantDigits * base^{exponent} \tag{1.1}$$

The numbers are, in general, represented approximately to a fixed number of significant digits (the significand) and scaled using an exponent. The base for the scaling is normally 2, 10 or 16. The idea of floating-point representation over intrinsically integer fixed-point numbers, which consist purely of significand, is that expanding it with the exponent component achieves greater range. For instance, to represent large values, e.g. distances between galaxies, there is no need to keep all 39 decimal places down to femtometre-resolution (employed in particle physics). Assuming that the best resolution is in light years, only the 9 most significant decimal digits matter, whereas the remaining 30 digits carry pure noise, and thus can be safely dropped. This represents a savings of 100 bits of computer data storage. Instead of these 100 bits, much fewer are used to represent the scale (the exponent), e.g. 8 bits or 2 decimal digits. Given that one number can encode both astronomic and subatomic distances with the same nine digits of accuracy, but because a 9-digit number is 100 times less accurate than the 11 digits reserved for scale, this is considered a trade-off exchanging range for precision. The example of using scaling to extend the dynamic range reveals another contrast with fixed-point numbers: Floating-point values are not uniformly spaced. Small values, close to zero, can be represented with much higher resolution (e.g. one femtometre) than large ones because a greater scale (e.g. light years) must be selected for encoding significantly larger values.[1] That is, floating-point numbers cannot represent point coordinates with atomic accuracy at galactic distances, only close to the origin.

### 1.4.1 Trade offs between range and precision

It is crucial to point out that by using the floating point representation we have not managed to represent more numbers. If for example we use a 64 bit floating point we can still represent only $2^{64}$ distinct numbers. However we have spread those distinct representations to a wider range.

There will always be a compromise between the range and the desired precision. Since there is only a given number of representation bits, by increasing the number of the exponent bit we achieve a greater range however there is a significand loss of precision as to which numbers in this range we can represent. Likewise by increasing the number of bits in the significand part we increase the precision but there is a smaller range of numbers to represent[8].

### 1.4.2 The floating point representation

A number representation specifies a way of storing a number in the form of a string of digits. There are several mechanics by which strings of digits can represent numbers. the digit string can be of any length, and the location of the radix point is indicated by placing an explicit "point" character (dot or comma) there. If the radix point is not specified then it is implicitly assumed to lie at the right end of the string. In fixed-point systems, some specific assumption is made about where the radix point is located in the string. For example, the convention could be that the string consists of 8 decimal digits with the decimal point in the middle, so that "00012345" has a value of 1.2345. In scientific notation, the given number is scaled by a power of 10 so that it lies within a certain range—typically between 1 and 10, with the radix point appearing immediately after the first digit. The scaling factor, as a power of ten, is then indicated separately at the end of the number.

The floating point representation, as shown in 1.1, consists of three parts.

- A signed string of digits of a given length, that represent the base of the number. It is usually referred to as the significand or mantissa. The length of this string determines the precision of the representation to be implemented. For example a

10 digit string can represent a value up to 10 digits worth of accuracy. I.e. the number 1/3 would be equal to 0.3333333333.

- A signed string of digits that represent the exponent, that modifies the magnitude of the number. The length of the string represents the range of the numbers that can be represented.
- An unsigned number which represents the base which is usually the number 2.

To derive the value of the floating-point number, one must multiply the significand by the base raised to the power of the exponent, equivalent to shifting the radix point from its implied position by a number of places equal to the value of the exponent to the right if the exponent is positive or to the left if the exponent is negative.

### 1.4.3 The IEEE 754 standard

The most used floating point format used today is the IEEE 754. The IEEE has standardized the representation used in most today's computers with some exceptions. The technical standard was established in 1985 and defines the following:

- arithmetic formats: sets of binary and decimal floating-point data, which consist of finite numbers (including signed zeros and subnormal numbers), infinities, and special "not a number" values (NaNs)
- interchange formats: encodings (bit strings) that may be used to exchange floating-point data in an efficient and compact form
- rounding rules: properties to be satisfied when rounding numbers during arithmetic and conversions
- operations: arithmetic and other operations on arithmetic formats
- exception handling: indications of exceptional conditions (such as division by zero, overflow, etc.)

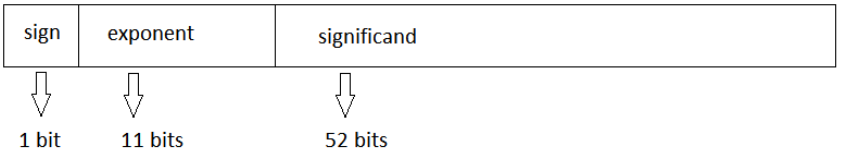


FIGURE 1.2: The double precision floating point format

### 1.4.4 Basic IEEE 754 formats

The standard defines five basic formats, which differ in that each has a different length in the strings of the significand and the exponent. Two of these format refer to a decimal representation but they are not going to be described here. The three binary representations are encoded with 32, 64, 128 bits respectively.

TABLE 1.1: The IEEE 754 floating point format

| Name | Common name | Base | Digits | E min | E max | Decimal digits | Decimal E max |
|---|---|---|---|---|---|---|---|
| binary16 | Half precision | 2 | 10+1 | -14 | +15 | 3.31 | 4.51 |
| binary32 | Single precision | 2 | 23+1 | -126 | +127 | 7.22 | 38.23 |
| binary64 | Double precision | 2 | 52+1 | -1022 | +1023 | 15.95 | 307.95 |
| binary128 | Quadruple precision | 2 | 112+1 | -16382 | +16383 | 34.02 | 4931.77 |
| decimal32 | | 10 | 7 | -95 | +96 | 7 | 96 |
| decimal64 | | 10 | 16 | -383 | +384 | 16 | 384 |
| decimal128 | | 10 | 34 | -6143 | +6144 | 34 | |

## 1.5 The IEEE 754 double precision floating point format

Since in our implementation we use a double precision floating point arithmetic unit, we will explain in more details the double precision format. The double precision format uses 64 bits in total to represent floating point numbers. Figure 1.2 displays the format of the number.

### 1.5.1 The sign bit

The first bit of the 64 bits, or the most significand bit, is used to represent the sign of the number. Zero is used to represent positive numbers and one to represent negative numbers.

### 1.5.2 The exponent

The next 11 bits are used to represent the exponent. It is crucial to mention here that the actual value of the exponent is not the value that is stored. For E the value stored the true value of the exponent is given by equation 1.2, $E_{bias}$ is the number 1023. In order to represent negative as well as positive numbers the number 1023 is subtracted from the number to get the real value. With 11 bits the largest number that can be represented is 2047 and the lowest 0, after the bias subtraction we get 1024 as the highest value and -1023 as the lowest that can be represented.

$$e = E - E_{bias} \tag{1.2}$$

### 1.5.3 The significand

The significand of the floating point number is represented by a string of 52 bits. However the actual number is not just these 52 bits. The fraction f stored in these 52 bits is in the range [0-1). The significand S is calculated with the equation 1.3. A more detailed explanation of the leading '1' digit follows in the next section.

$$S = 1.f \tag{1.3}$$

### 1.5.4 Floating point normalization

As mentioned earlier the actual bits of the significand part of the number are 53 and not 52. This happens because the MSB is always assumed to be one and therefore does not need to be stored which leads to storing area reduction. The final result of the calculation of any two floating point numbers, must have the MSB of the significand part equal to 1. However after a calculation it is often that the MSB will not be equal to 1, in which case the number must be normalised so as to change the MSB to 1. If the number is not normalized there are two possibilities, one the MSB is zero and second an overflow has occurred and a extra bit has been concatenated to the left of the significand. If the second has occurred no actual normalization has to take place regarding the significand part of the number however the exponent should be right shifted by one(A more detailed

explanation is given in chapter ******* about the Floating point implementation). If the MSB is zero then the significand must be shifted left by one and a zero must be concatenated to the right of the LSB as displayed in figure 1.4.

$$\overleftarrow{\text{left shift by one}} \overbrace{0.111010100...111}^{\text{64 bits}}$$

$$\overbrace{1.111010100...110}^{\text{64 bits}} \overleftarrow{\text{a zero has been concatenated}} \tag{1.4}$$

## 1.6 The goals of the thesis

It is crucial at this point to mention the goal of the work implemented as well as the motivation. This thesis work presents a way of designing and creating a soft processor which is based on ISA PLX 1.1[9]. Furthermore it is also important to highlight the flexibility of soft processors and for this reason, we implemented a customization on this processor by adding a custom FPU core. The whole process is described in full detail in the following chapters.

The full methodology of how to customize an IP-core and adjust it to someone's needs is presented, allowing someone by using it to create and customize his own cores. This is very important since it can save up valuable time which would be otherwise spent in an effort to figure out the most efficient way to accomplish the creation and customization of a module.

A complete implementation of a Floating Point Arithmetic Unit is also presented, providing a clear picture as to how it was implemented and tested. An existing design, specifically created to target FPGA devices, was used[10] and implemented. A testbench was also created allowing the user to verify the integrity of some features of the FPU.

## 1.7 The Following work structure

The following paper is structured as follows: Chapter 2 describes the processor implemented and its main characteristics as well as the key features and the goals behind the design. Chapter 3 describes the implementation of the FPU as well as the verification. Chapter 4 describes the processor customization in order to augment the FPU module as well as the testing of the processor. Chapter 5 describes all the necessary programming tools that were used along with the hardware. Chapter 6 is the epilogue with suggestions for improvements and acknowledged compromises of the design.

# Chapter 2

# The processor

In this chapter the whole process of designing and implementing the processor that was used is explained in detail. Most of the soft processors available, are general purpose with a main focus in flexibility and an all around usability. However all of these processors are well understood and developed and there is very little room for customization. Almost all the popular softcores have a development environment that fully supports customization. As a result the processor chosen to be implemented here is based on ISA PLX 1.1[9], an instruction set developed by professor Ruby B. Lee from Princeton University. It is a small processor that supports parallel subword instructions and is intended for multimedia processing. The full instruction set encoding tables can be found here [11]. Where needed the assumed register order in the instruction encoding tables is as follows: 1) Register Rd, 2) Register Rs1, 3) Register Rs3.

## 2.1 Architecture Highlights

The instruction set is designed for RISC architecture implementation. It is optimized for high speed multimedia processing. There are two features that distinguish multimedia processors from simple general purpose processors: Large amounts of parallel subword data processing and use of low precision data[12]. Both of these features are characteristics of the PLX instruction set. The processor utilizes 32 general-integer registers numbered R0 to R31. The register size is the same as the word width and an be 32, 64 or 128 bits long. Adjusting the size does not require any changes to the instruction set.

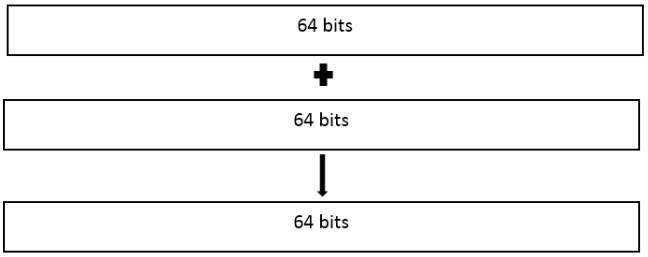


FIGURE 2.1: Addition between the whole 64bit words

### 2.1.1 Datapath Size

The datapath size chosen for the soft processor is 64 bits wide. The ISA itself supports 32, 64 and 128 bits datapaths, however the 64 bit wide is the silver lining among the other two[12]. Another feature that led to the decision of the 64bit wide datapath is the fact that the FPU implementation described at chapter 3 is a double precision one, hence the 64 bits. This is yet another example of the flexibility that is offered by soft core implementations; designers are free to choose the datapath and experiment with various sizes and make the best decision for their design. Smaller datapath reduces consumption and cost, while larger datapaths increase the performance. Register R0 is hardwired to 0 meaning that no changes can be made to this register and the value returned when requested will always be zero. R31 is the designated jump register used for the jump instructions.

### 2.1.2 Subword Parallelism

The key characteristic of the processor is the subword parallelism support. This means that certain instructions can be executed simultaneously to all the subwords of the 64bit word. The subword size can be chosen by the program and is 8, 16, 32 or 64 bit long. For example when executing the addition instruction between two words a choice can be made so as to add the two 64 bit words or add separately their subwords as displayed in figures 2.1, 2.2, 2.3, 2.4. Subword parallelism can increase the speed by up to 8 times depending on the word size chosen. However it is evident that there is a significand loss in precision when using the parallel instructions, but multimedia processing has a large margin for these kinds of precision losses.

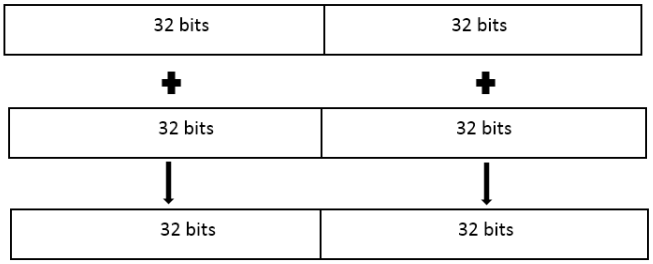


FIGURE 2.2: Addition between the two 32bit words

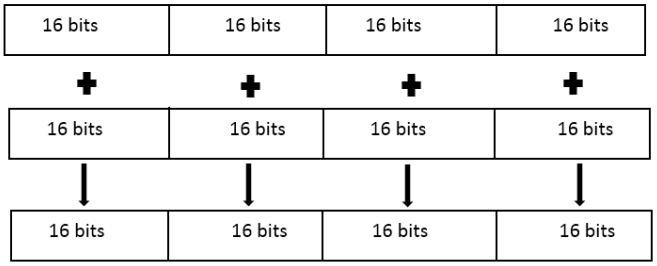


FIGURE 2.3: Addition between the four 16bit

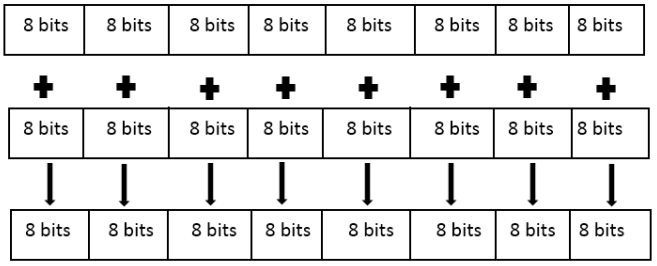


FIGURE 2.4: Addition between the 8 8bit words

### 2.1.3 Predication

All the instructions in PLX are predicated. This means that every instruction has a flag bit attached, if this bit is one then the instruction is executed else it is not. This results in the processor not having branch instructions or branch hazards. All of the instructions are propagated through the pipeline and those that have a zero predicate flag are simply not executed. There are eight 1-bit predicate registers, numbered P0 to P7. There are 16 of there 1byte predicate registers forming the predicate file. At any given time only one register is active which is chosen by software. P0 is hardwired to 1 meaning that all instructions assigned to this predicate register will be executed.

## 2.2 Processor Implementation

The main goals for the implementation were efficiency in terms of speed and area consumption. It is crucial at this point to mention that the goals set are solely for educational purposes and no actual research was done in order to determine the optimal designing goals.

The first step is to choose the overall design method for the processor. A five stage pipelined non-superscalar design was chosen. The five stages were chosen in order to isolate time consuming components, such as the memory access and data processing. After many experiments the five stage pipeline was found to be the most ideal choice. The data and instruction memories as well as the shifting unit were the slowest components and had to be isolated. Further increase in the number of pipeline stages was unnecessary and would result in no gain, because the memory access and the shifting unit operations could not be divided further. Thus, by isolating all the data processing units, along with the shifting module, we achieved the best performance, than every other number of pipeline stages. Since the project targets an FPGA board, area was of utmost importance so the superscalar approach was ignored. Appendix A Figure A.1 displays the RTL implementation of the PLX processor, using 5 pipeline stages.

## 2.3 The first pipeline stage

The first stage of the pipeline contains the program counter and the instruction memory. The program counter is responsible for providing the proper address of the Instruction Memory, that contains the next instruction to be fetched and executed by the processor. Appendix A Figure A.2 displays the RTL schematic of the program counter.

The module has seven ports, 6 input ports and 1 output which is the value of the program counter. Table 2.3 shows the ports of the program counter module.

### 2.3.1 The Program Counter

Depending on the instruction, the program counter increments by 1, by the value "Imm" provided by the instruction or by the value stored in the Rd register. The value 1 is the

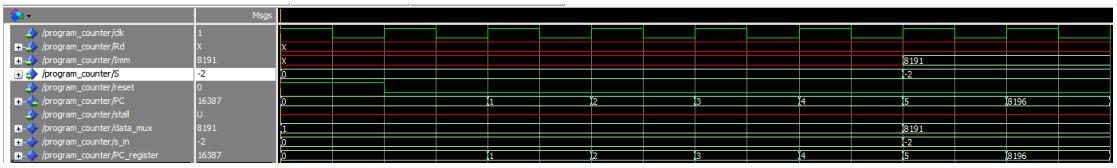


FIGURE 2.5: PC simulation

TABLE 2.1: Device Utilization Summary for the Program Counter

| Logic Utilization | Used | Available | Utilization |
|---|---|---|---|
| Number of Slice Registers | 64 | 93120 | 0% |
| Number of slice LUTs | 134 | 46560 | 0% |
| Number of used LUT-FF pairs | 64 | 134 | 47% |
| Number of bonded IOBs | 156 | 240 | 65% |
| Number of BUFG/BUFGCTRLs | 1 | 32 | 3% |

default value added to the current PC to calculate the next address of the Instruction Memory since the width of the memory is 32bits which is equal to the instruction word size. As a result there is one instruction per line stored in the memory and the program counter needs to increment by one to point to the next instruction. Some instructions require the program counter to increment by a certain value; this value can be stored either in the specified register Rd or it can be acquired directly from the instruction stored in the instruction memory (input Imm). The width size of the program counter value is 9 bits since the instruction memory size is 512 words. The value of the program counter is reset either by an external signal or if the value reaches 512.

After the design of the module, it needs to be verified for proper functioning. The program "Modelsim PE student edition" was used to simulate the design and test the correct functionality. In figure 2.5 a short simulation is displayed where the counter increases for 5 cycles and in the sixth a constant value is added via the "Imm" value.

### 2.3.2 Program Counter Metric Statistics

The maximum achieved frequency is 294.638MHz[1]. The size of this module is very small since it only consists of a 64bit adder, a 64bit register and 3 multiplexers. Table 2.1 shows the device utilization summary as generated by the ISE development kit. The total power consumption is 1.293Watts.

[1]The device used for the implementation is the Xilinx Virtex ML605; more details provided in Chapter 6

TABLE 2.2: The Instruction Memory Ports

| Name | Type | Size | Explanation |
|---|---|---|---|
| clka | input signal | 1 | clock signal |
| wea | input signal | 1 | memory initialization signal |
| addra | input signal | 32 | memory address |
| dina | input signal | 32 | input data |
| douta | output signal | 1 | output data |

TABLE 2.3: The program counter ports

| Name | Type | Size | Explanation |
|---|---|---|---|
| Sin | input signal | 2 | functionality signals |
| Rd | input signal | 64 | Rd register value |
| Imm | input signal | 23 | immediate value |
| clk | input signal | 1 | clock signal |
| res | input signal | 1 | reset signal |
| stall | input signal | 1 | stall signal |
| PC | output signal | 64 | program counter output |

### 2.3.3 The Instruction Memory

The instruction memory used is a 512x32bits block memory as mentioned earlier. The memory is generated be the Xilinx CORE Generator System[13]. This tool provides an easy way to communicate with the on-board memory found on the FPGA used. A more thorough explanation for the tool is given in chapter 6. This module has 3 ports plus two "hidden". The 3 standard ports are a 9bit address input port, a 32bit output data port and a 1bit clock input port. The "hidden" ports are the ones used to initialize the memory, i.e. to store the program to be executed. There is a 1bit initialization signal which stalls the memory and the processor and a 32bit signal carrying one instruction per clock cycle. To initialize the memory one simply has to raise the initialization signal to high, which automatically stalls the processor, and then input the instructions one per clock cycle to subsequent addresses. The standard frequency for the memory module is 144MHz. Table 2.2 displays all the port signals of the Instruction Memory.

### 2.3.4 The data multiplexers

Stage one of the pipeline contains two multiplexers that provide the two addresses required for the register file. Multiplexer M1 provides the address for the Rs2 or Rd register depending on the instruction. There are three possible locations in the instruction word: a) bits 12 down to 8, b) bits 17 down to 13, c) bits 22 down to 18. Multiplexer

M2 provides the address for the Rs1 register and the value is either in bits 17 down to 13 or in bits 22 down to 18. Both multiplexers are controled by the control unit and their value is automatically provided.

### 2.3.5 The stage 1 data flow

Stage one operates in one clock cycle. The program counter operates on the falling edge of the clock and provides the new value when clock has value '0'. The instruction memory operates on opposite clock cycle, i.e. rising edge of the clock. This allows the instruction memory to provide the new instruction in the same clock cycle as the program counter produces the new value, merging the two components in one stage. As soon as the clock takes the low value the program counter provides a new 64bit value. However only the 9 least significand bits are used to acquire the new instruction memory since its size is 512 words, hence 9 bits are required to encode all the addresses. The program counter produces a 64 bit value because the value needs to be used in the next stages. As soon as the instruction memory receives a new value it provides the new instruction which is driven in the control unit. The value that is fed in the control unit is the concatenation of the following bit strings: a) 28 down to 23, b) 17 down to 16, c) 7 down to 0. Bits 22 down to 0 and 31 down to 29 are driven to the multiplexers M1 and M2. The control unit provides the proper signals for this stage immediately after the new value is provided by the instruction memory.

## 2.4 The second pipeline stage

The second stage contains the reading part of the register file and the two data multiplexers. The register file stores 32 64bit words. Table 2.4 displays all the ports for the register file. The reading of the data in the register file is done in the falling edge of the clock while the writing, which is done in stage 5, is done in the rising edge so as to avoid data corruption in case a read an write instruction of the same register is performed simultaneously. This stage also contains the two bypassing multiplexers M3 and M4, more details about the data bypassing in the Hazard Dealing section.

Figure 2.6 displays a short simulation of the Register File. After the module is being reset two values 1024 and 1025 are driven to the addresses 13 and 14 respectively. However

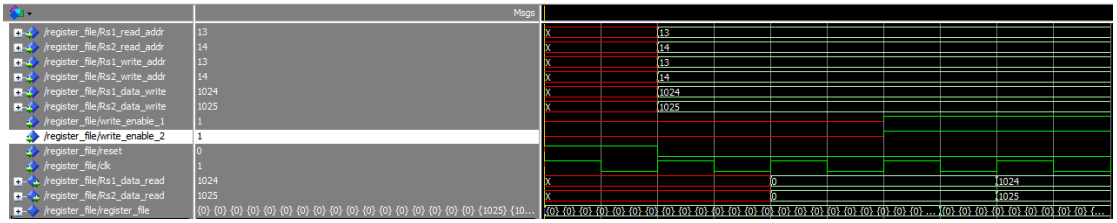


FIGURE 2.6: Register File Simulation

TABLE 2.4: The Register File Ports

| Name | Type | Size | Explanation |
|---|---|---|---|
| Rs1_read_addr | input signal | 5 | read address |
| Rs2_read_addr | input signal | 5 | read address |
| Rs1_write_addr | input signal | 5 | write address |
| Rs2_write_addr | input signal | 5 | write address |
| Rs1_data_write | input signal | 64 | write data |
| Rs2_data_write | input signal | 64 | write data |
| write_enable_1 | input signal | 1 | write enable |
| write_enable_2 | input signal | 1 | write enable |
| reset | input signal | 1 | reset |
| clk | input signal | 1 | clock signal |
| Rs1_data_read | output signal | 64 | data read output |
| Rs2_data_read | output signal | 64 | data read output |

the read ports do not register any change until the write enable signals are activated. As soon as the write enable signals are activated the data read ports provide the value in the next clock cycle. The register file operates at a maximum frequency of 382,117 MHz and occupies less than 1% of the FPGA

## 2.5 The third pipeline stage

The third stage of the pipeline is the data processing stage. Here all the data processing takes place and all the data processing units are contained. This stage is the most time consuming since it contains the slowest unit which is the shifter. This stage also contains the predicate file and a sign extension unit along with some multiplexers.

### 2.5.1 The Arithmetic Logic Unit(ALU)

The ALU is responsible for the basic calculations(add, subtract), the comparisons and the logic calculations. The top level design contains two components, the adder and the logic calculations module as well as some logic for the calculation of the test_bit. Table

TABLE 2.5: The ALU Ports

| Name | Type | Size | Explanation |
|---|---|---|---|
| Rs1 | input signal | 64 | Rs1 register data |
| Rs2 | input signal | 64 | Rs2 register data |
| S | input signal | 1 | operation signals |
| clk | input signal | 1 | clock input |
| enable | input signal | 1 | enable signal |
| reset | input signal | 1 | reset signal |
| Rd | output signal | 64 | Result data |
| trap | output signal | 1 | trap flag |
| OVF | output signal | 8 | overflow/underflow flag |
| T_F | output signal | 1 | true/false flag |

2.5 displays the ports used by the ALU module.This unit is responsible for the results from the following instructions: Addi, And, Andcm, Andi, Cmp, Cmpi, Not, Or, Ori, Padd, Paddincr, Pavg, Pcmp, Pmax, Pmin, Psub, Psubavg, Psubdecr, Psubavg, Subi, Testbit, Xor, Xori.

ALU operates in one, two or three cycles depending on the operation. All the logic calculations(or,nor,xor,not,and) require one cycle. Addition is performed also in one cycle with the exception of the addincr instruction, which adds two values and increments the result by one, which is done in two cycles. Subtraction is performed in two clock cycles, since the subtraction is performed with an adder the two's compliment method is used. The value to be subtracted is first converted to its two's compliment negative equivalent, this is done using the formula 2.1 where $n$ the two's compliment number and $b$ the original number. In the first cycle a '1' is subtracted from the numbers' compliment and in the second cycle the addition takes place. Similarly if the number needs to be decremented by 1 the operation takes up one more clock cycle. The ALU operates at 218,627 MHz and occupies 1% of the FPGA slices.

$$n = \bar{b} - 1 \tag{2.1}$$

The actual adder of the ALU is an array of 8 1byte adders. Since the addition can be performed on a varying word size the smallest possible word size adder must be used. The basic 1byte full adder is a standard full adder with the exception of providing an extra output the carry out of the 6th half adder. This carry out is required in order to detect overflows in signed calculations. Each adder has the carry out connected to

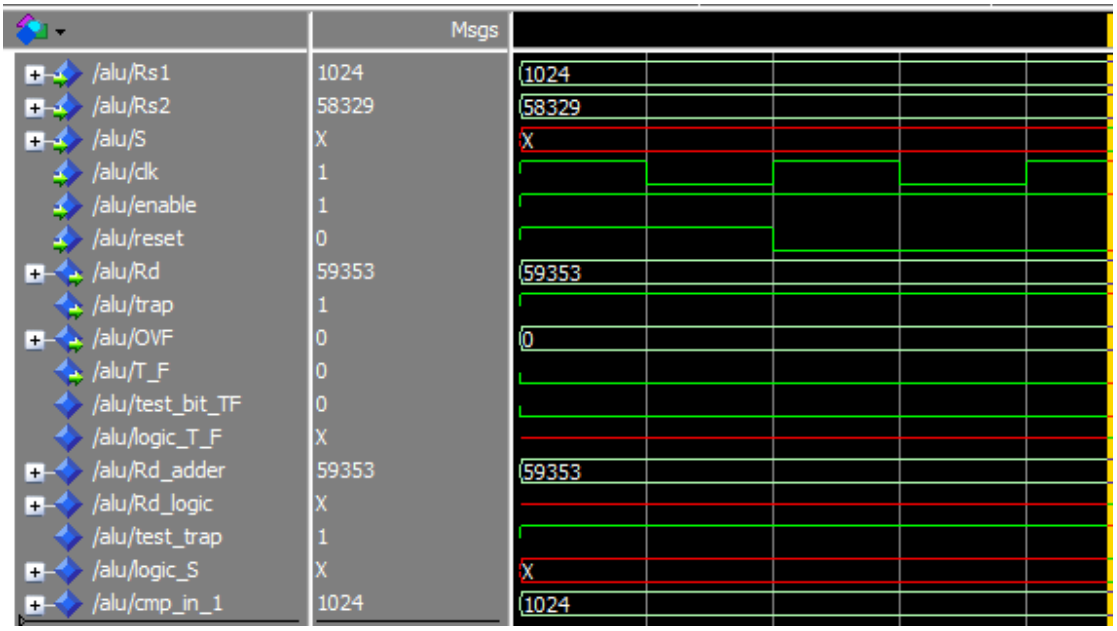


FIGURE 2.7: ALU simulating an addi operation

the next adder as well as the proper adders required when executing subword additions. For example if the addition is performed between 4 32bit size words, the fourth adders' carry out will be driven to the first adder and the 8th adder's carry out will be driven to the 5th adder as displayed in Figure 2.8 and 2.9. This carry out propagation is required only for the modular addition where the carry out is fed back to the carry in. The adder can perform signed/unsigned and modular/non-modular additions/subtractions.

The comparator performs all the basic comparisons(equality, comparison, minimum, maximum) between the two 64bit words as well as between all the subwords separately depending on the instruction.

Figure 2.7 displays an example of the simulation of the ALU. Presenting here every single instruction and subword combination is not possible, however all instructions have been simulated and verified. Appendix D provides a full list with all the components, including the ALU, and their signal encoding according to the instructions; refer to this table to set the proper signals in the simulations.

### 2.5.2 The Multiplier

The multiplier performs all the basic multiplying instructions. It can perform signed or unsigned multiplications between 16bit size words. These words can be either the odd

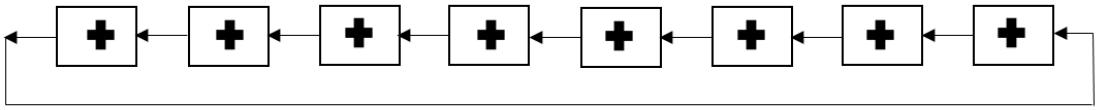

FIGURE 2.8: Carry out drive between two 64 bit numbers

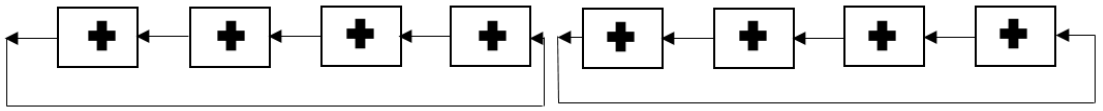

FIGURE 2.9: Carry out drive between four 32 bit numbers

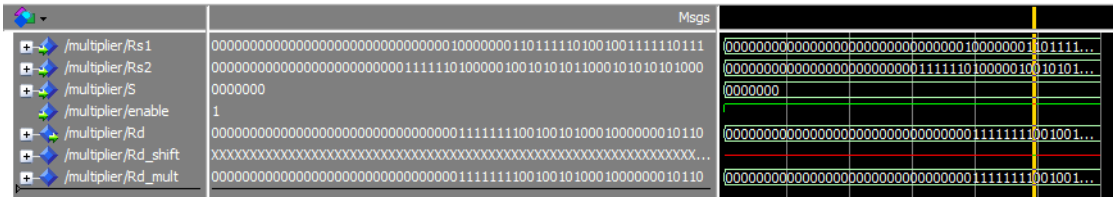


FIGURE 2.10: Multiplier Simulation

TABLE 2.6: The multiplier ports

| Name | Type | Size | Explanation |
|---|---|---|---|
| Rs1 | input signal | 64 | Rs1 register data |
| Rs2 | input signal | 64 | Rs2 register data |
| S | input signal | 7 | operation signals |
| enable | input signal | 1 | enable signal |
| Rd | output signal | 64 | Result data |

or even indexed subwords as shown in figures 2.11 and 2.12. The two 32bit size products of the multiplication form the new result. The multiplier performs the multiplication instructions, which are the following: Pmul, Pmulshr. Table 2.6 displays the list of the ports used by the multiplier module.

Figure 2.10 displays a simulation of the multiplier. The pmul.odd instruction is executed where the odd indexed subwords(63 donwto 48 and 31 downto 16) of each of the values in Rs1 and Rs2 are multiplied and the two 32 bit results are written to the left and right half word respectively. Since the multiplier does not operate with a clock the maximum frequency is calculated from the maximum propagation delay and is found to be 177,366 MHZ and it takes 1% of the FPGA area.

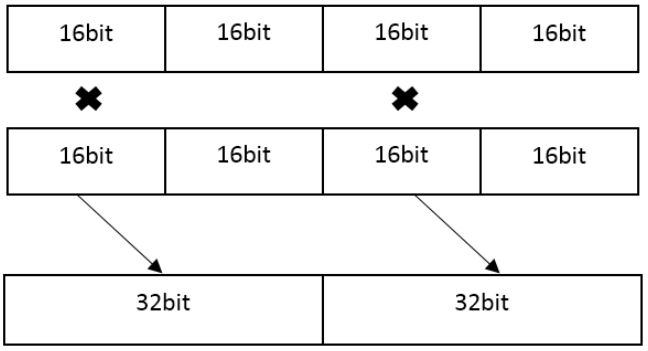


FIGURE 2.11: Odd indexed multiplication

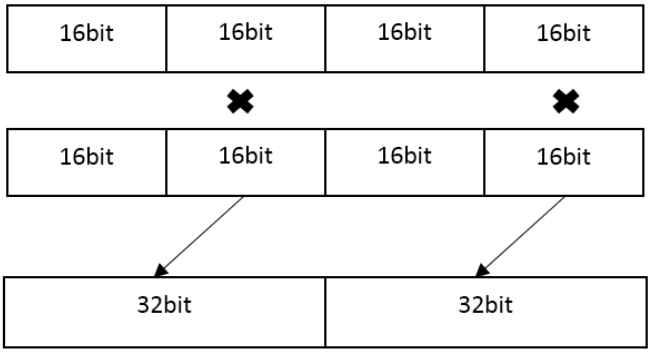


FIGURE 2.12: Even indexed multiplication

### 2.5.3 The Mix Unit

The Mix Unit is responsible for all the mix and multiplex instructions. It is basically a series of multiplexers and depending on the instruction it receives two inputs and rearranges the subwords accordingly. The mix unit performs the following mixing operations: Mix, Mux, Perm. Table 2.7 displays the list of the ports used by the Mix Unit.

Figure 2.13 displays the simulation of the Mix unit. A mix.l.4 instruxtion is executed where odd-indexed subwords are selected alternately from Rs1 and Rs2, and written to Rd. The first subword of Rd is the first subword of Rs1. The maximum frequency is found from the propagation delay and is calculated to be 269,759 MHz. The occupied area is 1%.

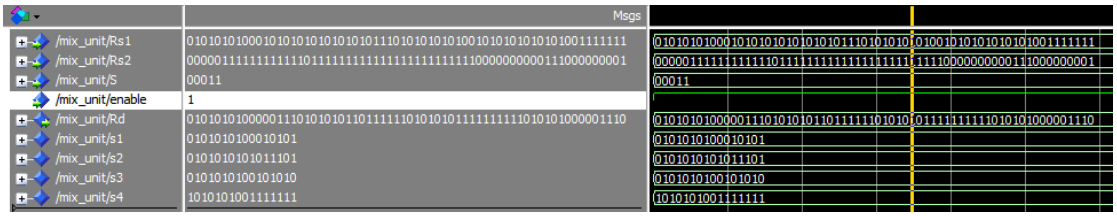


FIGURE 2.13: Mix Unit Simulation

TABLE 2.7: The Mix Unit ports

| Name | Type | Size | Explanation |
|---|---|---|---|
| Rs1 | input signal | 64 | Rs1 register data |
| Rs2 | input signal | 64 | Rs2 register data |
| S | input signal | 5 | operation signals |
| enable | input signal | 1 | enable signal |
| Rd | output signal | 64 | Result data |


FIGURE 2.14: Shifter Unit Simulation

### 2.5.4 The Shifter Unit

The shifter performs logic and arithmetic parallel shifts. The shifts can be either left or right and the shift amount is determined by the instruction. The shifter consists of an array of 8 barrel shifters. The formation is similar to that of the parallel adder since the output bit of each 1byte shifter is redirected to the input of a specific shifter depending on the subword size. The Shift unit is responsible for the shift instructions: Pshift, Pshiftadd, Shrp, Slli, Srai, Srli. Table 2.8 displays all the ports used by the shifter unit.

Figure 2.14 displays an example simulation of the shifter where the pshift.l.8 instruction is executed, where subwords of Rs1 are logically shifted to the left by Rs2 bits. The maximum frequency is found from the propagation delay and is calculated to be 133,832 MHz. The occupied area is 1%.

TABLE 2.8: The Shifter Unit ports

| Name | Type | Size | Explanation |
|---|---|---|---|
| Rs1 | input signal | 64 | Rs1 register data |
| Rs2 | input signal | 64 | Rs2 register data |
| S | input signal | 8 | operation signals |
| Imm | input signal | 13 | Immediate value input |
| enable | input signal | 1 | enable signal |
| Rd | output signal | 64 | result output value |

### 2.5.5 Predicate File, Sign Extension Unit and multiplexers

Stage 4 contains also a Sign extension unit which performs a sign, zero or one extension to provide a 64bit output. The Predicate File is also in the fourth stage. Since the predicate file is updated according to the compare instructions which are executed by the ALU, this update is performed in the same cycle as the comparisons take place eliminating any data hazard potentials that could arise from a reference to a predicate signal not yet set properly. Stage 4 contains 3 data multiplexers and one predicate multiplexer. The M5 and M6 data multiplexers drive the proper data to the processing units. Since only one unit can be active at any time all the processing units receive the same inputs. Multiplexers M7 and M8 drive the proper result form the processing units to the next stage. The Predicate Multiplexer provides the predicate signal which is attached to the specific instruction. This bit is driven to the next stages where the data update takes place.

## 2.6 The fourth Pipeline

Stage four contains the access to data memory. The memory has only one port so in a clock cycle either a read or a write is performed. It contains 1024 words with a word size of 64 bits and it is always enabled. In order for the memory to write or update a value the "Write Enable(WE)" signal must be activated. This signal is the "and" of the control unit signal that activates the write in the memory and the predicate signal. If the predicate signal is not 1 there will not be any update on the memory and therefore the instruction is presumed as not executed. The memory is automatically produced by the CORE Generator System provided by Xilinx in order to use the FPGA on board memory. Its maximum frequency os 144MHz and it operates in the falling edge of the clock cycle.

## 2.7 The Fifth Pipeline Stage

Stage five is the write back stage where the register file is updated. It is critical to point out that if the predicate bit of the instruction is zero at this point no update will take place. The register file write back operates in the rising edge of the clock to avoid data corruption in case the same register is read and written in the same clock cycle.

### 2.7.1 The register Input Unit

This unit is used to execute the instructions "extract", "deposit" and "loadi" which require, besides the standard Rs1 or Rs2 registers, also the Rd register to be processed. As a result the Rd register needs to be read and written in the same instruction. This is achieved by having the Rs1 processed normally and the Rd read and propagated through the pipeline to the register input unit. Here all the necessary replacements take place according to the instruction executed. The other option would be to make the register file in such way to have the ability to update specific bits and bytes from various words which would make it even more time consuming and area inefficient.

## 2.8 The control Unit

The control unit is responsible for monitoring the proper function of the processor and sending the proper signals to all the stages. The control unit implemented is a hard-wired control unit. This means that generally it uses sequential logic units, featuring a finite number of gates that can generate specific results based on the instructions that were used to invoke those responses. Hardwired control units are generally faster than microprogrammed designs. The hardwire feature of the control unit renders it fast but inflexible. However its easier to implement in a reprogrammable environment such as the FPGA and much faster. Figure A.4 Appendix A, displays the RTL schematic of the control unit. Table 2.11 displays all the ports used by the control unit.

The control unit contains 6 components:

- **The operation decoder**. Responsible for decoding the instructions.
- **The Bypass Unit**. Responsible for data bypassing and forwarding.

```
architecture op_dcdr of op_decoder is
        begin
        process(op, subop)
        begin
                case op(15 downto 10) is
                when "000000" =>--jmp
                        PC_control_signals          <= "10";
                        ALU_control_signals         <= "0XXXXXXXXXXXXXXXXXXX";
                        MULT_control_signals        <= "0XXXXXXX";
                        SHIFT_control_signals       <= "0XXXXXXXX";
                        MIX_control_signals         <= "0XXXXX";
                        FPU_control_signals         <= "0XXX";
                        SIGN_EXT_control_signals    <= 'X';
                        DATA_MEM_control_signals    <= "00000000";
                        REGISTER_F_control_signals  <= "00";
                        PREDICATE_control_signals   <= "000";
                        REG_INPUT_control_signals   <= "0000";
                        illegal                                    <= '0';
```

FIGURE 2.15: The "jump" instruction decoding

- **The predicate signals unit**. Responsible for controlling the predicate file and each signals.
- **The stall unit**. Responsible for the stall signals.
- **The illegal flags unit**. This unit produces the system flags that are sent to the program to determine program flaws such us overflows and underflows.
- **The Multiplexers unit**. This unit controls all the multiplexers of the processor.

### 2.8.1 The operation decoder

This unit is responsible for decoding the instruction arriving from the instruction memory and dispatching the appropriate signals. This module is purely composed of combinational logic and the input and output ports can be seen at table 2.9. The decoder is a vast array of decoders and multiplexers. It's a very simple circuit in the design but a very extensive one. A future addition could be to be automatically generated by another software program. The way it operates is the following. For every unique combination of the operation and sub-operation it receives it produces the respective unique output signals. Figure one shows a sample code for the "jump" instruction. The MSB of every output signal is the "enable" signal, '1' for enabled '0' for disabled. Since only the program counter is involved in the "jump" instructions all the modules are disabled but the PC. The rest of the bits are the control signals of each module which can be seen in D.

TABLE 2.9: The OP Decoder Unit ports

| Name | Type | Size | Explanation |
|---|---|---|---|
| op | input signal | 16 | operation signals |
| subop | input signal | 3 | sub-operation signals |
| PC_control_signals | output signal | 2 | program counter signals |
| ALu_control_signals | output signal | 20 | ALU signals |
| MULT_control_signals | output signal | 8 | multiplier signals |
| SHIFT_control_signals | output signal | 9 | shifter |
| MIX_control_signals | output signal | 6 | MIX signals |
| FPU_control_signals | output signal | 4 | FPU signals |
| SIGN_ext_control_signals | output signal | 1 | sign extension signals |
| DATA_MEM_control_signals | output signal | 8 | data memory signals |
| REGISTER_F_control_signals | output signal | 2 | register file signals |
| PREDICATE_control_signals | output signal | 3 | predicate file signals |
| REG_INPUT_control_signals | output signal | 4 | register input signals |
| illegal | output signal | 1 | illegal operation flag |

TABLE 2.10: The Stall Unit ports

| Name | Type | Size | Explanation |
|---|---|---|---|
| st_1_op | input signal | 6 | stage 1 operation |
| st_2_op | input signal | 6 | stage 2 operation |
| st_3_op | input signal | 6 | stage 3 operation |
| subop_st_3 | input signal | 6 | stage 3 sub-operation |
| clk | input signal | 1 | clock signal |
| reset | input signal | 1 | reset signal |
| trap | input signal | 1 | trap signal |
| illegal | input signal | 1 | illegal flag input |
| FPU_done | input signal | 1 | FPU done signal |
| PC_Stall | output signal | 1 | PC stall |
| Global_stall | output signal | 1 | Global stall |
| Reg_flush_st_1 | output signal | 1 | register flush stage 1 |

### 2.8.2 The stall unit

This unit is responsible for stalling the processor when necessary in order to avoid hazards. Stalling is enabled in multi-cycle operations, jump instructions and load instructions. There are two kinds of stalling implemented. The first is the global stall where all the stages of the processor halt execution indefinitely and the second is the PC stall where the program counter is stalled and a NOP instruction is inserted in the pipeline. The stall unit is aware at any given time of the operations executed in stage 1, stage 2 and stage 3 of the processor so as to assess the situation and stall if necessary. Table 2.10 displays the port list used by the module.

TABLE 2.11: The Control Unit ports

| Name | Type | Size | Explanation |
|---|---|---|---|
| op | input signal | 16 | The operation to be executed |
| subop | input signal | 3 | Sub-operation(required for some instructions) |
| Rs1A_st2 | input signal | 5 | Rs1 stage 2 address |
| Rs2A_st2 | input signal | 5 | Rs1 stage 2 address |
| Rda_st3 | input signal | 5 | Rd stage 3 address |
| Rda_st4 | input signal | 5 | Rd stage 4 address |
| Rda_st5 | input signal | 5 | Rd stage 5 address |
| Pd_st3 | input signal | 1 | predicate signal from stage 3 |
| Pd_st4 | input signal | 1 | predicate signal from stage 4 |
| Pd_st5 | input signal | 1 | predicate signal from stage 5 |
| trap | input signal | 1 | trap signal from ALU |
| ALu_OU | input signal | 8 | Overflow/Underflow signal from ALU |
| ALU_TF | input signal | 1 | True/False signal from ALU |
| clk | input signal | 1 | clock signal |
| reset | input signal | 1 | reset signal |
| controls_st1 | output signal | 2 | Stage 1 control signals |
| controls_st3 | output signal | 51 | Stage 3 control signals |
| controls_st4 | output signal | 8 | Stage 4 control signals |
| controls_st5 | output signal | 6 | Stage 5 control signals |
| mux1 | output signal | 2 | M1 control signals |
| mux2 | output signal | 1 | M2 control signals |
| mux3 | output signal | 3 | M3 control signals |
| mux4 | output signal | 3 | M4 control signals |
| mux5 | output signal | 2 | M5 control signals |
| mux6 | output signal | 1 | M6 control signals |
| mux7 | output signal | 3 | M7 control signals |
| mux8 | output signal | 1 | M8 control signals |
| mux9 | output signal | 1 | M9 control signals |
| PC_Stall | output signal | 1 | program counter stall signal |
| trap_out | output signal | 1 | illegal trap flag |
| alu_ou_out | output signal | 8 | overflow/underflow flag |
| illegal_out | output signal | 1 | illegal instruction flag |
| st_1_flush | output signal | 1 | stage 1 flush instruction |
| G_stall | output signal | 1 | global stall signal |

### 2.8.3 The Flag Unit

The flag unit is responsible for handling and sending the proper flag signals when an error occurs. The way they are handled is by sending an output signal from the processor top module for one clock cycle on the cycle that the error was detected. This means that an external unit for proper handling and storing these flag signals is required. Table 2.12 shows the ports used by the module. The illegal flag is raised when an instruction that is not encoded is requested. The trap flag is raised when when the instruction

TABLE 2.12: The Flag Unit ports

| Name | Type | Size | Explanation |
|---|---|---|---|
| illegal | input signal | 1 | illegal signal |
| trap_in | input signal | 1 | trap instruction |
| ALU_OUF | input signal | 8 | ALU overflow/underflow |
| FPU_OUF | input signal | 2 | FPU overflow/underflow |
| trap_o | output signal | 1 | trap flag |
| ALU_OU_o | output signal | 8 | ALU overflow/underflow flag |
| FPU_OU_o | output signal | 2 | FPU overflow/underflow flag |
| illegal_o | output signal | 1 | illegal flag |

"test bit" produces a '1'. The ALU overflow/underflow flag is raised when an overflow or underflow occurs in the ALU adder. This flag is 8 bits long one bit for every 1 byte adder contained in the adder.

## 2.9 The processor top module

The maximum achieved frequency for the processor in the specific board is 58,289 MHz. However calculating the instructions per second requires real time simulation with designated algorithms. The distribution of the frequency, at which every instruction is used, is crucial to calculate properly the performance. The instructions that require more than one clock cycle to complete are: Addf 3 cycles, Subf 3 cycles, Divf 55 cycles, Sqrrtf 59 cycles, Cmp 3 cycles, Cmpi 3 cycles, Jmp 2 cycles, Load 5 cycles , Loadi 5 cycles, Loadx 5 cycles, Paddincr 3 cycles, Psub 2 cycles, Psubdecr 3 cycles. According to Lee[12], cmp appears on average of 4%, Jmp 1.5%, load 5%, loadi 1%. We can assume that the rest of the instructions with multi-cycle completion time have a negligible effect on the final performance. Assuming that Addf appears on average of 5%, Subf 4%, Multf 2%, Divf 1% and Sqrrtf 1% we can calculate the instructions per second (IPS), using Eq. 2.2.

$$IPS = A_1 \times F_1 + A_2 \times F_2 + A_3 \times F_3 + \ldots + A_n \times F_n \tag{2.2}$$

Where $A_n$ the appearance frequency and $F_n = F_{max}/required\_clock\_cycles$. With this formula we calculated the instructions per second for our implementation to be 52955260 IPS.

TABLE 2.13: Read after write hazard example

| | T1 | T2 | T3 | T4 | T5 |
|---|---|---|---|---|---|
| stage 1 | add(R28, R21, R23) | add(R20, R28, R21) | | | |
| stage 2 | | add(R28, R21, R23) | add(R20,R28, R21) | | |
| stage 3 | | | add(R28, R21, R23) | add(R20, R28, R21) | |
| stage 4 | | | | add(R28, R21, R23) | add(R20, R28, R21) |
| stage 5 | | | | | add(R28, R21, R23) |

## 2.10 Hazards and data corruption

While the pipeline implementation boosts the performance the main drawback is the hazards that can occur while the execution takes place. Instructions in a pipelined processor are performed in several stages, so that at any given time several instructions are being processed in the various stages of the pipeline, such as fetch and execute. There are many different instruction pipeline micro-architectures, and instructions may be executed out-of-order. A hazard occurs when two or more of these simultaneous (possibly out of order) instructions conflict.

### 2.10.1 Read after write hazard

A read after write hazard occurs when an instruction tries to read a result that has not yet been calculated and therefore reads corrupt data. This occurs because even though the instruction is executed after the previous instruction the result has not yet fully propagated through the pipeline. If for example in our implementation an add operation is being executed in the third stage, the actual register update will take place after two cycles in the fifth stage. As a result any data read that occurs in the second stage that targets the same register as the one that is bound to be updated in stage 3, will receive corrupt data. In table 2.13 an example of such hazard is displayed. While the first add instruction is in the third stage of the pipeline, it still has not updated the R28 register. However at the same T3 time the next add instruction is already reading the R28 register which has corrupt data.

### 2.10.2 Branch Hazards

Despite the fact that using the predicate system eliminates the branch prediction hazards, there is one exception. This hazard can occur with the ”jump” instruction. Since all instructions are predicated so is the ”jmp” instruction, if there is a compare instruction to be executed in the pipeline(which is responsible for updating the predicate file)

TABLE 2.14: Branch Hazard example

| | T1 | T2 | T3 | T4 | T5 |
|---|---|---|---|---|---|
| stage 1 | cmp.gt(PD2,PD3) | jmp(PD2,Rd) | | | |
| stage 2 | | cmp.gt(PD2,PD3) | jmp(PD2,Rd) | | |
| stage 3 | | | cmp.gt(PD2,PD3) | jmp(PD2,Rd) | |
| stage 4 | | | | cmp.gt(,PD2,PD3) | jmp(PD2,Rd) |
| stage 5 | | | | | cmp.gt(PD2,PD3) |

that targets the same predicate that is used for the "jump" instruction there is a potential hazard in the execution as displayed in table 2.14. The "jump" instruction must not be executed before the predicate file is updated.

### 2.10.3 Structural Hazards

structural hazard occurs when a part of the processor's hardware is needed by two or more instructions at the same time. In this case this type of hazard can occur in the register file and specifically if there is a read and write instruction targeting at the same time the same register. However this type of hazard is dealt as mentioned earlier by forcing the read and write instruction to access the register file at different clock edges.

### 2.10.4 Pipeline bubbling

There are several ways to deal with data hazards in the pipelined processors. Bubbling or pipeline stall is the simplest solution. Pipeline stalling is a way of preventing data, structural, and branch hazards from occurring. As instructions are fetched, control logic determines whether a hazard could/will occur. If this is true, then the control logic inserts NOPs into the pipeline. Thus, before the next instruction (which would cause the hazard) is executed, the previous one will have had sufficient time to complete and prevent the hazard.

This method being the simplest is extensively used in the processor. Specifically it is used for all the multicycle instructions such as the subtraction which requires two cycles to complete. A more efficient way of dealing with the multicycle instructions would be to modify the processor to a superscalar one, where there would be multiple instances of the processing units, such as the ALU. While one component is busy calculating, the other unit can be utilised for another instruction. However since area constraints is a major factor the pipeline solution was chosen.

TABLE 2.15: The Bypassing unit

| Name | Type | Size | Explanation |
|---|---|---|---|
| Rd_addr_stg3 | input signal | 5 | stage 3 Rd address |
| Rd_addr_stg4 | input signal | 5 | stage 4 Rd address |
| Rd_addr_stg5 | input signal | 5 | stage 5 Rd address |
| Rs1_addr | input signal | 5 | Rs1 read address |
| Rs2_addr | input signal | 5 | Rs2 read address |
| pred_stg_3 | input signal | 1 | stage 3 pred signal |
| pred_stg_4 | input signal | 1 | stage 4 pred signal |
| pred_stg_5 | input signal | 1 | stage 5 pred signal |
| op | input signal | 6 | operation |
| mux_3_s | output signal | 3 | M3 signals |
| mux_4_s | output signal | 3 | M4 signals |

Several other solutions could be applied to solve the problems of the multi-cycle instructions, such as out-of-order-execution parallel module operation, where other units can operate while others are busy, and scoreboarding, but this requires a lot of research and experimentation which would be off the limits of this thesis work.

### 2.10.5 Register Forwading

Another solution which is extensively used in the processor is data forwarding. Forwarding involves feeding output data into a previous stage of the pipeline to avoid the reading of corrupt data. This is achieved by having the control unit constantly monitoring all the stages and the register data used by those stages. If one stage has available data that is required by another stage then those data are forwarded to this stage. For example in table 2.13 the way to deal with the hazard that occurs in T3 is to forward the value of the R28 at stage 3 back to stage 2. This is achieved by a simple multiplexor which drives the proper data.

In this implementation there is data forwarding from stages 3,4,5 back to stage 2. The control unit monitors from every stage the instruction, the read and write registers, as well as the predicate of each instruction and with a simple combinational logic forwards the proper data through the multiplexors M3 and M4. Table 2.15 displays the ports used by the bypassing unit.

# Chapter 3

# The Floating Point Unit(FPU)

A floating point unit is a part of the processor that is specialized to carry out operations on floating point numbers. Some of the operations are addition, subtraction, multiplication, division and square root. Some modern modules can also perform trigonometric calculations, however most systems use libraries dedicated for these operations. Most systems have an integrated FPU however some systems implement the floating point calculations via a co-processor. Many computers use emulators for such operations, where the floating point is emulated and in the final stage the calculation is converted to integer calculations. The drawback of this procedure is the low time efficiency and poor resource management since not only the floating calculation takes more time but valuable processing time is being taken from the main processor.

Most modern processors use embedded FPUs and even dedicated floating-point registers such as the Intel x86 instruction set. In some cases, FPUs may be specialized, and divided between simpler floating-point operations (mainly addition and multiplication) and more complicated operations, like division. In some cases, only the simple operations may be implemented in hardware or microcode, while the more complex operations are implemented as software.

In this processor case a support for floating point calculations is mandatory, since multimedia processing often perform repeated parallel data processing, mostly floating point calculations, floating point arithmetic support is mandatory for multimedia applications[14]. For our purposes a FPU specification was selected, which could fit within our design and would be suitable for FPGA implementation. The design

chosen[10] is a small double precision floating point arithmetic unit design, which is suitable for such implementations as it focuses mainly in area optimization and cost reduction. The design supports exception flags and overflow-underflow checks that are given as outputs to be handled by the higher level design, in our case the processor. The implementation of the double precision FPU is ours and it follows the published work of Paschalakis et al[10], with some alterations. For example for the multiplier an alteration was made so the instruction is completed in 1 cycle instead of ten with only 20% more area increase, but it performs much faster.

## 3.1 Floating point addition-subtraction

Adding and subtracting two floating point numbers is more complicated than the corresponding addition-subtraction performed for integer numbers. This is due to the representation of the floating point numbers. Suppose we wish to add two floating point numbers, A and B. A has the form of $A = \pm S_A * 2^{E_A}$ and B has the form of $B = \pm S_B * 2^{E_B}$. In order to perform an addition between those two numbers the first step is to modify the exponents so as to have the same value. This is done by calculating the absolute difference $|E_A - E_B|$ of the two exponents and adjusting the significand S accordingly. This is possible because the significand is stored in binary and therefore it can be divided by 2. The following example displays this adjustment.

$$A = 16 * 2^7$$

$$B = 32 * 2^6$$

The absolute difference of the two exponents is 1, so the smaller exponent needs to adjust.

$$B = 32 * 2^6 = 32 * 2^{-1} * 2^6 * 2^{+1} = 16 * 2^7$$

This is only possible because the significand will always be a multiple of 2 and therefore a simple shift to the right by the corresponding amount will adjust the number. The next step is to simply add or subtract the significands and normalize the result. This procedure is standard but the actual implementations in hardware vary. The schematic

implemented(Appendix A A.5) is the one from [10] where the whole dataflow is explained in full detail.

The adder requires 3 clock cycles to complete the operation and the maximum achieved frequency is 81.155MHz with a power consumption of 1.293Watts. Table 3.1 displays the FPGA area utilization.

### 3.1.1 FPU Adder testbench

All the components of the adder where tested and verified with Modelsim. However the fact that the adder is very complex and the possible results and the amount of values that need to be tested is vast, a testbench has been created for this purpose. A testbench is a program that provides a n automated method of testing and verifying modules. Testbenches are usually composed by two parts. 1) The software simulating program, which is a program written in any desired language(in our case c) that simulates the hardware function in software. Since the same operation is much simpler and easier to execute in software it is more reliable and stable. For example, while the adder consists of approximately 1000 code lines, the corresponding C program consists of approximately 300 code lines.

The first step is to create a large number of random generated vectors that will be used as test inputs. Since the vectors we need to use as inputs are two 64bit numbers the rand() function was used with a number cap 2 in order to produce a random binary number. This means that this function will provide a random number $< 2$ i.e. 0 or 1. Figure 3.2 displays the code. The vectors are then stored in two arrays which are then printed in a .txt file. Since the program only creates two vectors each time, Sa and Sb, a simple windows shell script (.bat) program was created to rerun the program several times and produce a desired amount of vectors. The .bat program is shown in Figure 3.1. Since the rand() function uses a timer as a seed for the random generation a 1 second interval is created between the generated vectors to avoid identical copies.

After the vector are generated the program continues execution and simulates the dataflow of the hardware. The results produced are stored in a .txt file along with two flags for overflow or underflow. The problem that was encountered, was that the hardware executes binary calculations but the same could not be done in software. So

```
@echo off
for /l %%i in (1,1,100) do (
        fpu_add_sub.exe
        timeout /t 1
)
```

FIGURE 3.1: The .bat program

```
int sa[64]={0};
int sb[64]={0};

void rand_vars(int *p) {
        int i;
        for (i=0;i<64;i++) {
                p[i]=rand()%2;
        }
}

rand_vars(sa);
rand_vars(sb);
```

FIGURE 3.2: The random vector generator

TABLE 3.1: Device Utilization Summary for the FPU adder

| **Logic Utilization** | **Used** | **Available** | **Utilization** |
|---|---|---|---|
| Number of Slice Registers | 154 | 93120 | 0% |
| Number of slice LUTs | 1400 | 46560 | 3% |
| Number of used LUT-FF pairs | 142 | 1412 | 10% |
| Number of bonded IOBs | 199 | 240 | 82% |
| Number of BUFG/BUFGCTRLs | 1 | 32 | 3% |

some calculations like addition and subtraction were executed in decimal numbers and others such as shift were executed in binary. Two functions were created to convert decimal to binary and vice versa. Figure 3.3 shows the two functions along with the type definition that had to be used to represent so large numbers.

The next step is to create the VHDL testbench that automatically uses inputs and runs the simulation for many vectors. This is done by creating a top module that contains the module to be tested. The top module reads the .txt file generated by the program and feeds the vectors to the component in predetermined time intervals, which are the clock cycles required to complete the execution. The outputs are also stored in the same .txt generated by the program so as to be compared.

```
typedef unsigned long long int unit64;

void conv_to_bin(unit64 s,int *d,int n) {
        int i;
        unit64 tmp=s;
        for (i=n-1;i>-1;i--) {
                d[i]=tmp%2;
                tmp/=2;
        }
}

unit64 conv_dec(int *p,int m,int n) {
        int i;
        unit64 tmp=0;
        for (i=m;i<n;i++) {
                if (p[i]==1)
                        tmp+=pow(2,n-1-i);
        }
        return tmp;
}
```

FIGURE 3.3: The two conversion functions

## 3.2 Floating point multiplication

The multiplying process is much simpler than the addition/subtraction. For two numbers A and B with $A = \pm S_A * 2^{E_A}$ and $B = \pm S_B * 2^{B_B}$ the first step is to calculate the sum $E_A + E_B - E_{bias}$ where $E_{bias} = 1023$. The $E_{bias}$ is subtracted since the exponents are in the form of $E + 1023$ and therefore the extra E_bias must be subtracted. The next step is to multiply the significands and normalize the result. Similarly to the adder there are many implementations that can be used. The one used here is the one described in [10] and the corresponding RTL schematic can be found in Appendix A A.6. However some alternations were implemented. The main alternation is that the multiplier operates in 1 cycle instead of 10. This is done because the actual multiplying unit implemented is not the one described in [10] and instead a classic IEEE std logic 1164 multiplier was used which occupies only 20% more area.

The multiplier is a combinational circuit so the maximum frequency is calculated by the propagation delay, $F_{max} = 1/(8.535ns) = 117MHz$. Table 3.2 shows the area utilization for the specified board. The power consumption is 1.293Watts.

Figure 3.4 displays the simulation results for the multiplier. A and B are the two 64bit numbers to be multiplied. The following numbers were used:

$A = 1101010100010101010101010101110101010101001010101010101001111111$

$B = 1100111101011011010111011011011011010111011011011011011101100111$

Table 3.2: Device Utilization Summary for the FPU multiplier

| **Logic Utilization** | **Used** | **Available** | **Utilization** |
|---|---|---|---|
| Number of Slice Registers | 446 | 93120 | 0% |
| Number of slice LUTs | 2382 | 46560 | 5% |
| Number of used LUT-FF pairs | 351 | 2477 | 14% |
| Number of bonded IOBs | 201 | 240 | 83% |
| Number of BUFG/BUFGCTRLs | 2 | 32 | 6% |
| Number of DSP48E1s | 15 | 288 | 65 |

and the following result was produced:

R = 0110010010000010001111101000000010111100001111010000100001011100

After breaking down the result to the actual components we get the following:

$Sign_A = 1$

$Sign_B = 1$

$Sign_R = 0$

$E_A = 10101010001$

$E_B = 10011110101$

$E_R = 11001001000$

$S_A = 0101010101010101110101010101001010101010101001111111$

$S_B = 1011010111011011011011010111011011011011011101100111$

$S_R = 0010001111101000000010111100001111010000100001011100$

The sign result is 0 i.e. positive since minus times minus is plus. The exponent result is $E_A + E_B - 1023 + 1$ the +1 is the result of rounding and normalization. The actual significand result is $1.S_R$ due to the hidden bit that is not stored and the 52 bits of $S_R$ are the 52 most significant bits of the multiplication result.

## 3.3 Floating point division

The division algorithm employed here is the simple non-performing sequential algorithm. Division in general is a much less frequent calculation so a very similar implementation of the design found here [10] was chosen since speed is not of utmost concern. The only differences is that the final rounding and normalizing stages of the calculation have been merged to one cycle resulting in a 55 cycle operation. The actual divisor module

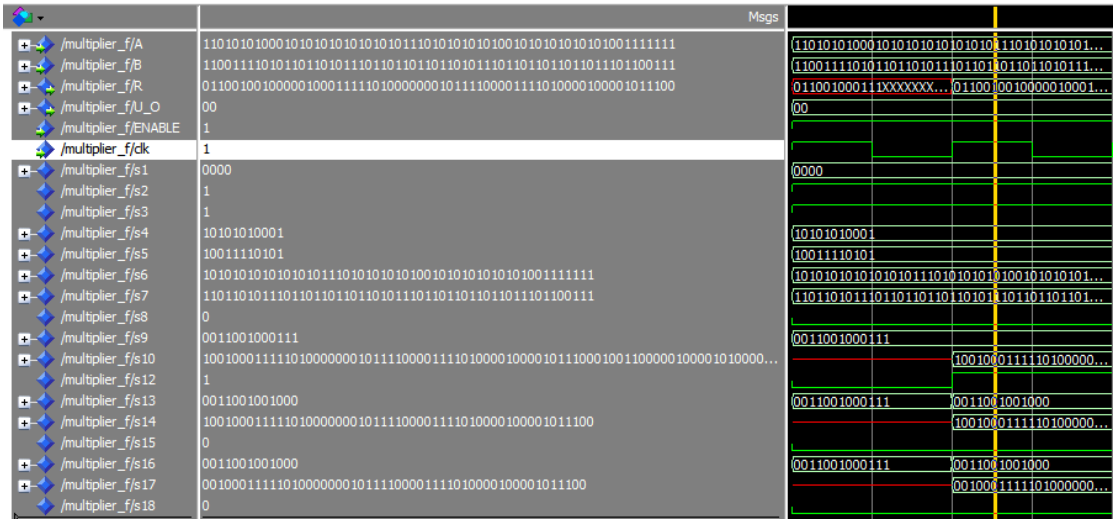


FIGURE 3.4: The FPU multiplier simulation

TABLE 3.3: Device Utilization Summary for the FPU Divider

| **Logic Utilization** | **Used** | **Available** | **Utilization** |
|---|---|---|---|
| Number of Slice Registers | 446 | 93120 | 0% |
| Number of slice LUTs | 2382 | 46560 | 5% |
| Number of used LUT-FF pairs | 351 | 2477 | 14% |
| Number of bonded IOBs | 201 | 240 | 83% |
| Number of BUFG/BUFGCTRLs | 2 | 32 | 6% |
| Number of DSP48E1s | 15 | 288 | 65 |

is shown in Figure 3.6 and the whole division module can be found at Appendix A A.7. This module produces one bit per entry of the final result.

The first step in calculating the result is similar to this of the multiplication. The exponent is calculated with the following formula: $E_R = E_A - E_B + E_{bias}$. As with the multiplication since the actual exponents are in the form of $E + E_{bias}$ when two exponents are subtracted the two $E_{bias}$ cancel each other out, so an extra $E_{bias}$ must be added to the final result.

The next step is calculating the division between $S_A$ and $S_B$. Since the two significands are in the range of [1,2) the result will be $S_R$ will be in the range (0.5,2), i.e. if not already normalized, it will require a left shift by just one place. The final step is to normalize and round the result which is performed in the final cycle.

The divider operates at a maximum frequency of 254.704MHz and the power consumption is 1.293Watts.Table 3.3 shows the area utilization by the divider.

Figure 3.5 displays the simulation results for the divider. A and B are the two 64bit numbers to be divided. The following numbers were used:

$A = 1101010100010101010101010101110101010101001010101010101001111111$
$B = 1100111101011011010111011011011011010111011011011011011101100111$

and the following result was produced:

R = 0100010110110000000000000001000100000000000000000000000000000000

After breaking down the result to the actual components we get the following:

$Sign_A = 1$
$Sign_B = 1$
$Sign_R = 0$

$E_A = 10101010001$
$E_B = 10011110101$
$E_R = 10001011011$

$S_A = 0101010101010101110101010101001010101010101001111111$
$S_B = 1011010111011011011011010111011011011011011101100111$
$S_R = 0000000000000001000100000000000000000000000000000000$

The sign result is 0 i.e. positive since minus by minus is plus. The exponent result is the difference between the two exponents with the addition of E_bias = 1023,$E_R = E_A - E_B + 1023$. The significand result is $1.S_R = 1.S_A/1.S_B$. The simulation lasts for 5600ms or 56 cycles of 100ms each. One cycle is the reset process and the rest 55 cycles is the calculation of the result.

## 3.4 The FPU Top Module

In order to easily access and make use of the floating point processing units(Adder/Subtracter, Multiplier, Divisor), a top module was created to include them. The RTL schematic can be found here A.8. The synthesized top module has a maximum frequency of 81.155MHz which is limited by the slowest component, the adder. In this implementation it is not possible for the modules to operate simultaneously, however it is considered a necessary

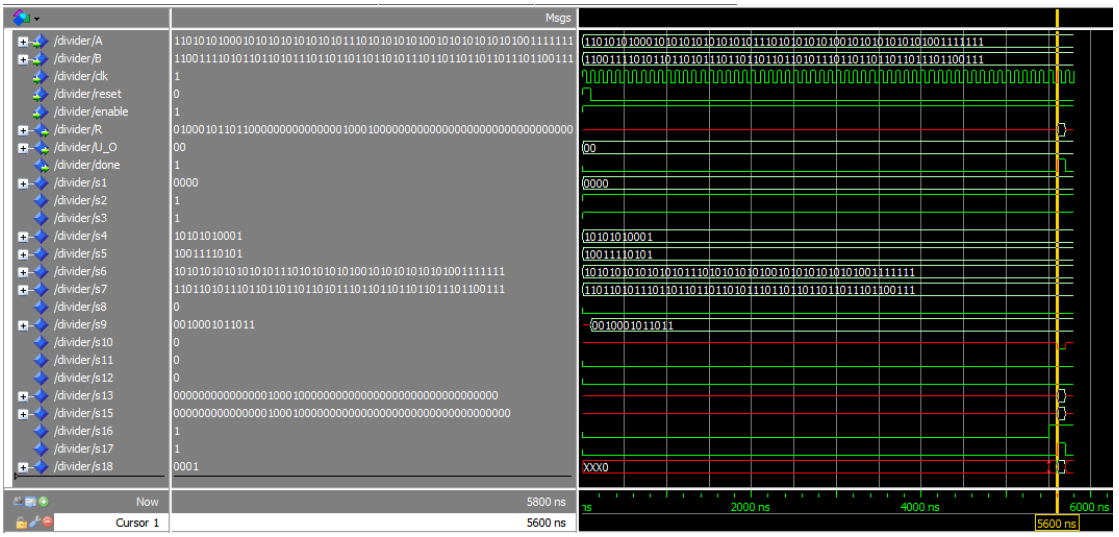


FIGURE 3.5: The FPU divider simulation

```
--FPU.DIVIDER.SIGNIFICAND_DIVISION.COMBINATIONAL_DIVISION--
--ANGELOS NTASIOS--

--------------LIBRARIES---------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
-------------------------------------------------

entity combinational_division is
        port(
                remainder_in  : in std_logic_vector(52 downto 0);
                Sb            : in std_logic_vector(52 downto 0);
                remainder_out : out std_logic_vector(52 downto 0);
                Sr_out        : out std_logic
        );
end;

architecture cmntnl_dvsn of combinational_division is
begin
        process(remainder_in,Sb)
        begin
                if( unsigned(remainder_in) >= unsigned(Sb) ) then
                        Sr_out <= '1';
                        remainder_out <= std_logic_vector(unsigned(remainder_in)
            - unsigned(Sb));
                else
                        Sr_out <= '0';
                        remainder_out <= remainder_in;
                end if;
        end process;
end;
```

FIGURE 3.6: The Divisor module

modification for the future, especially since the divider requires 55 cycles to complete one calculation, during which the other components can operate freely. Table 3.4 displays a detailed report of the area utilization for the FPU.

Table 3.4: Detailed device utilization summary for FPU

| **Slice Logic Utilization** | **Used** | **Available** | **Utilization** |
|---|---|---|---|
| Number of Slice Registers | 447 | 93,120 | 1% |
| Number used as Flip Flops | 392 | | |
| Number used as Latches | 54 | | |
| Number used as Latch-thrus | 0 | | |
| Number used as AND/OR logics | 1 | | |
| Number of Slice LUTs | 2,223 | 46,560 | 4% |
| Number used as logic | 2,215 | 46,560 | 4% |
| Number using O6 output only | 1,807 | | |
| Number using O5 output only | 50 | | |
| Number using O5 and O6 | 358 | | |
| Number used as ROM | 0 | | |
| Number used as Memory | 0 | 16,720 | 0% |
| Number used exclusively as route-thrus | 8 | | |
| Number with same-slice register load | 5 | | |
| Number with same-slice carry load | 3 | | |
| Number with other load | 0 | | |
| Number of occupied Slices | 650 | 11,640 | 5% |
| Number of LUT Flip Flop pairs used | 2,262 | | |
| Number with an unused Flip Flop | 1,875 | 2,262 | 82% |
| Number with an unused LUT | 39 | 2,262 | 1% |
| Number of fully used LUT-FF pairs | 348 | 2,262 | 15% |
| Number of unique control sets | 7 | | |
| Number of bonded IOBs | 201 | 240 | 83% |
| Number of RAMB36E1/FIFO36E1s | 0 | 156 | 0% |
| Number of RAMB18E1/FIFO18E1s | 0 | 312 | 0% |
| Number of BUFG/BUFGCTRLs | 2 | 32 | 6% |
| Number used as BUFGs | 2 | | |
| Number used as BUFGCTRLs | 0 | | |
| Number of ILOGICE1/ISERDESE1s | 0 | 360 | 0% |
| Number of OLOGICE1/OSERDESE1s | 0 | 360 | 0% |
| Number of BSCANs | 0 | 4 | 0% |
| Number of BUFHCEs | 0 | 72 | 0% |
| Number of BUFIODQSs | 0 | 36 | 0% |
| Number of BUFRs | 0 | 18 | 0% |
| Number of CAPTUREs | 0 | 1 | 0% |
| Number of DSP48E1s | 15 | 288 | 5% |
| Number of EFUSE_USRs | 0 | 1 | 0% |
| Number of FRAME_ECCs | 0 | 1 | 0% |
| Number of GTXE1s | 0 | 8 | 0% |
| Number of IBUFDS_GTXE1s | 0 | 6 | 0% |
| Number of ICAPs | 0 | 2 | 0% |
| Number of IDELAYCTRLs | 0 | 9 | 0% |
| Number of IODELAYE1s | 0 | 360 | 0% |
| Number of MMCM_ADVs | 0 | 6 | 0% |
| Number of PCIE_2_0s | 0 | 1 | 0% |
| Number of STARTUPs | 1 | 1 | 100% |
| Number of SYSMONs | 0 | 1 | 0% |
| Number of TEMAC_SINGLEs | 0 | 4 | 0% |
| Average Fanout of Non-Clock Nets | 3.58 | | |

# Chapter 4

# The Processor Customization

This chapter presents the whole methodology and all steps that were taken in order to augment the FPU described in the previous chapter to the processor. Customizing the processor is not a easy task since there are many variables that need to be taken into account. Many things can go wrong since the initial design was not intended to support such additions. However the flexibility provided by soft cores provides the necessary tools to fulfill such task. The main goal is to embed a new module, the FPU, in the existing processor design in order to support floating point arithmetic calculations which are considered mandatory for multimedia processors. The strategy is to affect as little as possible the overall processor performance while providing a sufficient interface for the proper utilization of the new component.

## 4.1 Compatibility

The first thing that must be taken care off is the compatibility. The main concern in such situations is the word size. A module cannot be easily used by another processing unit if the don't both use the same word size. While it is possible to augment such piece, by usually using multi-cycle instructions and stalling, it severely affects the performance and requires more area, especially temporary storing registers. For this reason the FPU implementation chosen is a double precision design in order to conform with the 64bit word size processor. Specifically for FPU module, it is very hard to use designs that do not have the same word size as the top module, since the storing representation cannot

be easily adjusted to a different word size. Other module such as shifters or adders are more easy to make use off, usually by adding two or as many as required parallel modules working together.

A future work addition, supports the ability to use multiple FPUs and other modules for testing purposes. This is achieved by creating a general interface of ports which all the same class modules must comply to. Those that have a different interface can be adapted so they can be used in those designs. A series of other steps and alternations must take place in order to support this implementation, which are clearly presented here[15].

## 4.2 ISA Customization

The first adaptation that needs to be done is to the instruction set. The required instructions to support the floating point calculations do not exist. For this purpose 4 simple 32bit instructions were created each for the perspective calculation. These instructions are just the basic ones that are required for the FPU to function, however more complex instructions can be added that involve for example immediate values and shifted results.

The four new instructions added are the following:

1. **addf** This is the instruction for the floating point addition between two numbers.
2. **subf** This is the instruction for the floating point subtraction between two numbers.
3. **divf** This is the instruction for the floating point division between two numbers.
4. **multf** This is the instruction for the floating point multiplication between two numbers.

No further encoding slots or bits had to be added since several encoding slots were free, figure 4.1 displays the encoding of the addf instruction. Four consecutive slots were occupied in order to easily handle them. These encoding slots would previsouly raise an ”illegal operation” flag, however they can now be used for the floating point instructions.

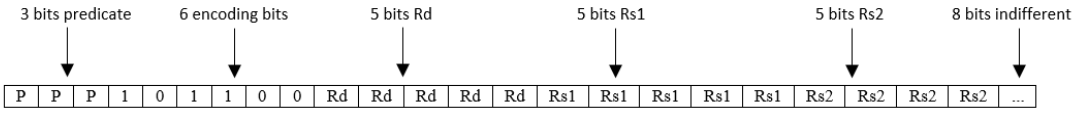


FIGURE 4.1: Stage 2 of the pipeline

For addf the new encoding is "101100", for subf "101101", for multf "101110" and divf "101111". Appendix C Figure C.1 displays the signals encoded to run the instructions.

## 4.3 Wiring and stalling

The next necessary step is to connect and fully support the function of the new module. The FPU was placed in the stage 3 of the pipeline, where all the other data processing units are also placed. The adaptations there are in the form of a few simple extra wiring and the new control signals targeting the FPU. The outputs from the two data multiplexers M6 and M5 are now also driven to the respective RS1 and Rs2 inputs of the FPU. Multiplexer M7 that is responsible of providing the proper result to the next stage has the output of the FPU now also connected. Control signals are also connected to the new module. Regarding the top module level no other alternation must take place.

The component that requires some alternations is the control unit.The control unit being responsible for organizing synchronizing and controlling all the modules and units of the processor needs to adapt. The first major change is the internal wiring of the control unit, as it needs to support extra inputs, outputs, flag signals, and control signals. The first change is the addition of three more outputs in the stage 3 control signals output, which are the new FPU control signals. Next a signal for the overflow/underflow FPU output and a new flag for the FPU overflow/underflow. Since all the FPU instructions are multicycle a signal "done" is also connected to inform the control unit that the operation has been completed. This is done to avoid adding unnecessary counting circuits in the control unit which would not be area efficient. The FPU instructions have also been added to the stalling list where the processor stalls until the "done" signal is activated.

## 4.4 Processor Testing

As it is clear the addition of new components and the customization of various components compromises the integrity and functionality of the processor. There are many parameters that need to be tested to reassure a 100% proper functionality such us timing, data synchronization, instruction hazards, resetting capabilities, interrupt signals and many more. A very robust debugging can be performed with an assembler a compiler and various simulating scripts, however due to limited time these tools could not be developed. A basic timing and data synchronizing as well as distinct instruction execution were tested.

### 4.4.1 Data Synchronization

A problem encountered in the development of this processor and the pipeline design was the data synchronization through the pipeline propagation. The problem arises when data from different components and different time constraints are propagated to the next pipeline stage. Since the processor is pipelined many stages contain modules which operate with as well as without a clock signal. This is a problem since the module operating on a clock signal can cause data to propagate at different cycles than modules that don't operate with a clock. The Register File reading stage is such an example. As seen in Figure 4.2, there are 3 main databuses propagating: Data from register R1, data from the Register File and data from register R2. The pipeline registers for the Register File address reading have been removed as they caused data to be desynchronized, since they would require 2 clock cycles to propagate to the third stage.

In order to better visualize and verify the proper data propagation, a mirror design of the processor's RTL which simulates the data propagation through registers was created as seen in figure 4.3. Even though the second path involving all the major contains two more clock dependent units, data arrives at the same clock cycle at the fifth stage as seen in figure 4.4. This is possible due to the fact that various components operate at different clock cycles. The same behavior is expected from the processor since the registers and combined with the sequential circuits form a similar mirror design. The code for the sample mirror RTL design can be found in Appendix C

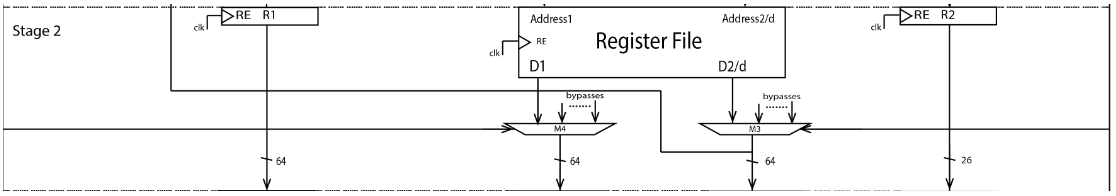


FIGURE 4.2: Stage 2 of the pipeline

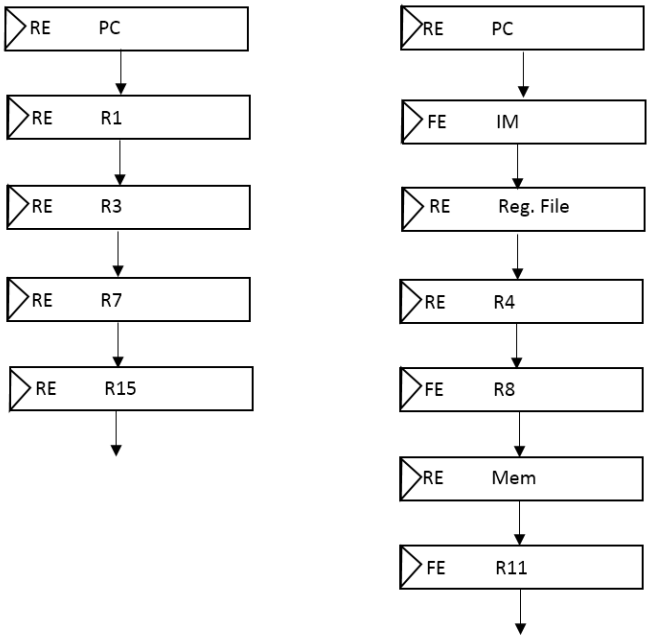


FIGURE 4.3: Mirror RTL schematic for data propagation

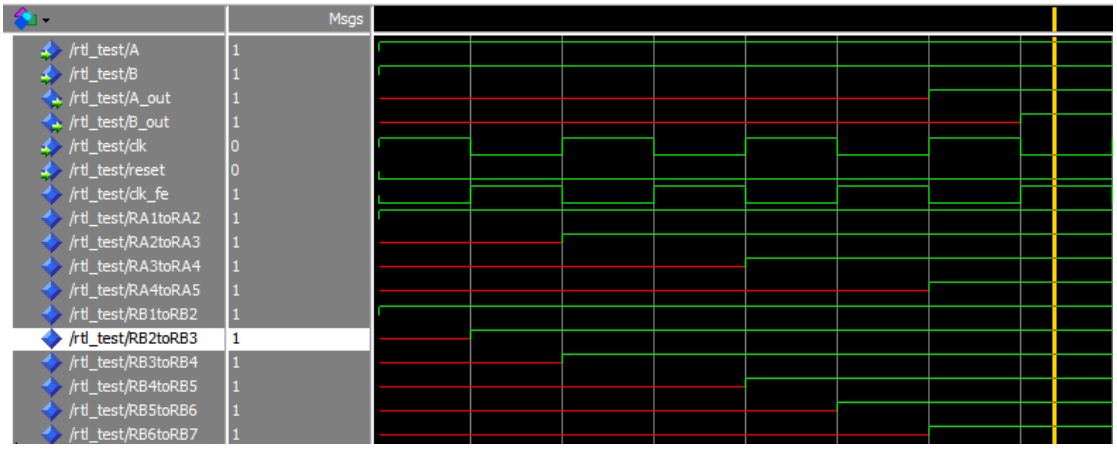


FIGURE 4.4: Mirror RTL simulation

```
1) loadi.k.0            000 000101 00001 00 1001001010010101
2) loadi.k.1            000 000101 00001 01 0010100101001001
3) loadi.k.2            000 000101 00001 10 1101111111111110
4) loadi.k.3            000 000101 00001 11 0100101001010100

5) loadi.k.0            000 000101 00010 00 0010010101010100
6) loadi.k.1            000 000101 00010 01 0101010100000000
7) loadi.k.2            000 000101 00010 10 0010101010110000
8 )loadi.k.3            000 000101 00010 11 1101010101010111

9) and                  000 110000 00011 00001 00010 110000 00
```

FIGURE 4.5: The "and" machine code

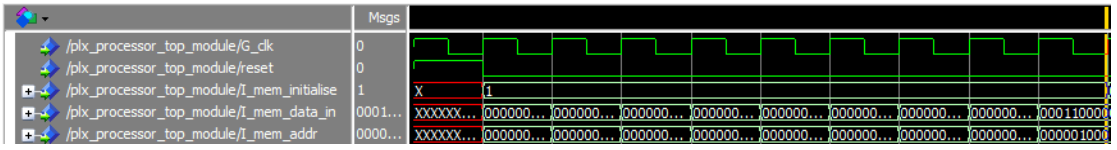


FIGURE 4.6: The processor initialization process

### 4.4.2 Execution Testing

In order to fully test the processor a compiler is required, however this was not implemented due to time constraints. All the instructions have been verified individually and found to operate properly. A sample execution follows which implements a simple and instruction. This specific example was chosen because it utilises many components of the processor: Register file read/write, program counter incrementation/stall, instruction memory initialization/read, ALU and instruction. Figure 4.5 shows the instructions used along with the instruction codes.

The first four instructions load the register "00001" with 16 bits of data each filling a different subword. The next four instructions fill the register "00010" with data similarly to the first to form another random 64 bit number. The final instruction is the "and" instruction which ands the values in the registers "00001" and "00010", and then stores the result in the register "00011". All instructions are predicated to the "000" register which is hardwired to '1' so as to be always executed.

The first step in the processor function is to reset the top module. Second step is to initialize the memory and insert all the instruction in consecutive instruction memory slots, starting from position "000000000" up to position "000001000". These steps are shown in figure 4.6. The simulation lasts 1000ns, since the clock step is 100ns nine cycles are for the memory initialization process and one for the resetting.

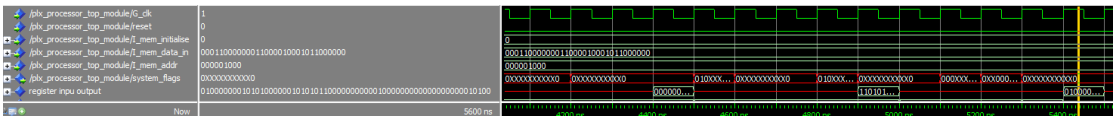


FIGURE 4.7: The execution completion stage

After the initialization the processor starts execution and the whole process lasts another 44 cycles. Each load instruction lasts five cycles since the processor stalls to avoid hazards. Eight load instructions makes up to 40 cycles plus another 4 for the register write of the "and" instruction. Figure 4.7 displays the completion of the whole process at clock cycle 54 where the "register input" module gives the correct result.

As expected the register file, after the execution, contains three values:

- "01001010010101001101111111111110001010010100100110010010100101010" is in the address "00001"
- "11010101010101110010101010110000010101010000000000100101010101000" is contained in the address "00010"
- "01000000010101000000101010110000000000010000000000000000000010100" the "and" result is contained in address "00011". The address "00000" was not used since it is hardwired to zero.

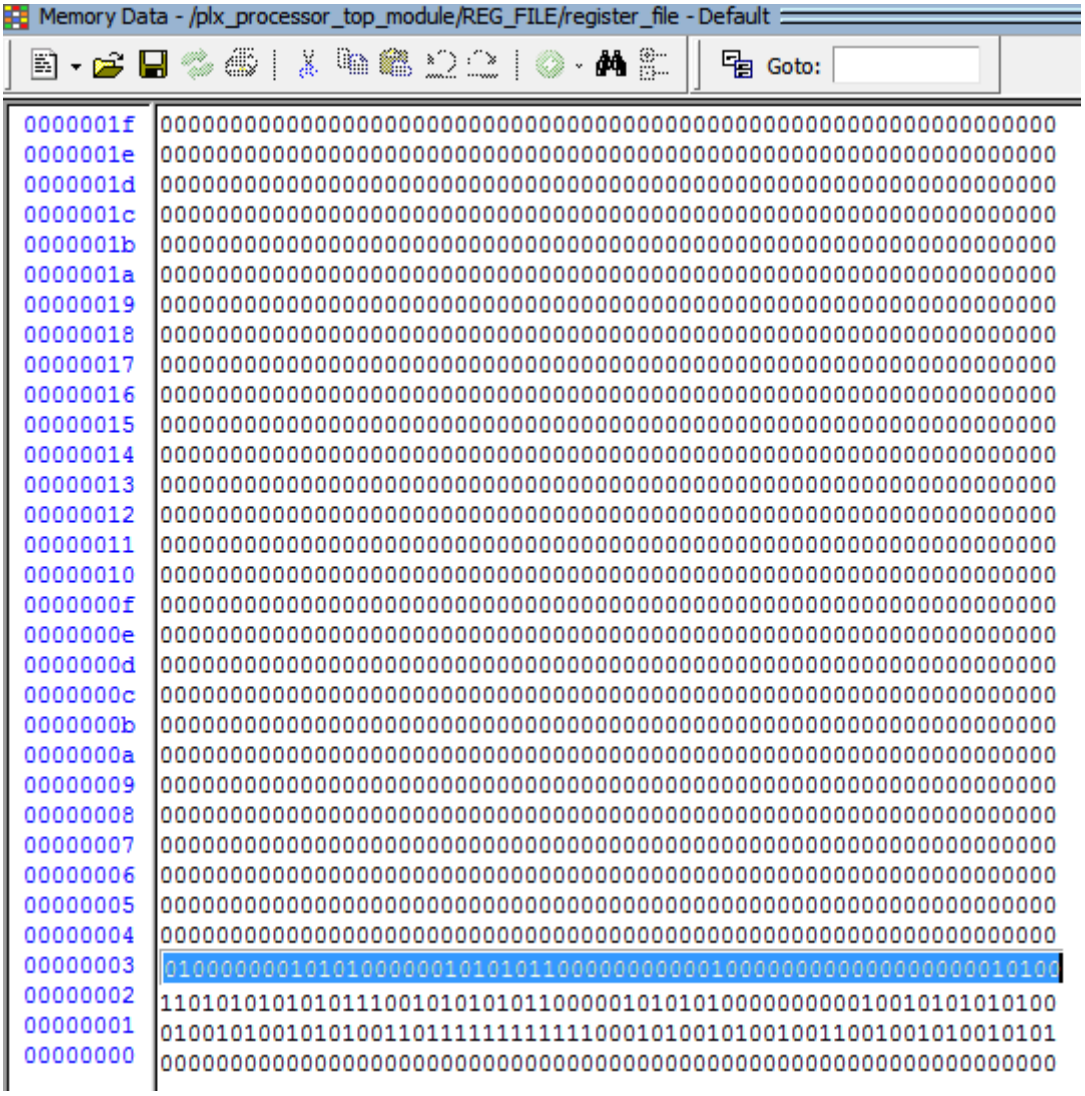


FIGURE 4.8: The register file values

# Chapter 5

# Tools Utilized

While making this thesis work, a series of programming languages, software programs and hardware has been used in order to design, test and implement the processor. For creating and programming the modules themselves, VHDL was used, which is a Hardware Description Language. C was also used for the creation of the testbench. In terms of IDEs, the hardware simulation program Modelsim was used and for the hardware implementation Xilinx ISE and the FPGA Vyrtex ML605.

## 5.1 VHDL

Hardware description languages first appeared in the late 1960s and they had the form of conventional languages. In 1971 C. Gordon Bell and Allen Newell introduced the concept of register transfer level to describe the behavior of circuits. Many other implementations appeared until the late 1970s when Verilog ,the first HDL for VLSI, appeared. In 1987, a request from the U.S. Department of Defense led to the development of VHDL (VHSIC Hardware Description Language).HDL was based on the Ada programming language, as well as on the experience gained with the earlier development of ISPS.[9] Initially, Verilog and VHDL were used to document and simulate circuit designs already captured and described in another form (such as schematic files). HDL simulation enabled engineers to work at a higher level of abstraction than simulation at the schematic level, and thus increased design capacity from hundreds of transistors to thousands[16].

TABLE 5.1: std_logic values

| Character | Value |
|---|---|
| 'U' | uninitialized |
| 'X' | strong drive, unknown logic value |
| '0' | strong drive, zero |
| '1' | strong drive, one |
| 'Z' | high impedance |
| 'W' | weak drive, unknown logic value |
| 'L' | weak drive, logic zero |
| 'H' | weak drive, logic one |
| '-' | don't care |

The introduction of logic synthesis for HDLs pushed HDLs from the background into the foreground of digital design. Synthesis tools compiled HDL source files (written in a constrained format called RTL) into a manufacturable netlist description in terms of gates and transistors. A circuit design from a skilled engineer, using labor-intensive schematic-capture/hand-layout, would almost always outperform its logically-synthesized equivalent, but the productivity advantage held by synthesis soon displaced digital schematic capture to exactly those areas that were problematic for RTL synthesis: extremely high-speed, low-power, or asynchronous circuitry.

### 5.1.1 The IEEE standard

The IEEE Standard 1076 defines the VHSIC Hardware Description Language or VHDL. It was originally developed under contract F33615-83-C-1003 from the United States Air Force awarded in 1983 to a team with Intermetrics, Inc. as language experts and prime contractor, with Texas Instruments as chip design experts and IBM as computer system design experts. The language has undergone numerous revisions and has a variety of sub-standards associated with it that augment or extend it in important ways.

The IEEE 1164 standard uses "multi-valued" signals, where a signal's drive strength (none, weak or strong) and unknown values are also considered. As a result a signal of type std_logic can acquire 9 values according to table 5.1.

his system promoted a useful set of logic values that typical CMOS logic design could utilize in the vast majority of modeling situations. The 'Z' literal makes tri-state buffer logic easy. The 'H' and 'L' weak drives permit wired-AND and wired-OR logic. Additionally, the 'U' state is the default value for all object declarations so that during

simulations uninitialized values are easily detectable and thus easily corrected if necessary.

VHDL has file input and output capabilities, and can be used as a general-purpose language for text processing, but files are more commonly used by a simulation testbench for stimulus or verification data.

### 5.1.2 Advantages

The key advantage of VHDL, when used for systems design, is that it allows the behavior of the required system to be described (modeled) and verified (simulated) before synthesis tools translate the design into real hardware (gates and wires). Another advantage is that it allows the description of concurrent systems, where data flow is concurrent and parallel, in contrast to conventional programming languages. Probably the largest advantage is that the projects and modules created are reusable. For example once a register file or an adder is created, it can be used again in other design with no alternations as long as the interface is the same.

## 5.2 FPGAs

Field Programmable Gate Arrays are a family of hardware used for implementing reprogrammable hardware. FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC). FPGAs by themselves do not actually implement any specific hardware and they have no particular function. They contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together". FIgure 5.1 displays the general pattern of the cells in a FPGA. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. Figure 5.2 displays a simplified design of the basic block of a Xilinx xc4000 FPGA. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

Logic resources are resources on the FPGA that can perform logic functions. Logic resources are grouped in slices to create configurable logic blocks. A slice contains a

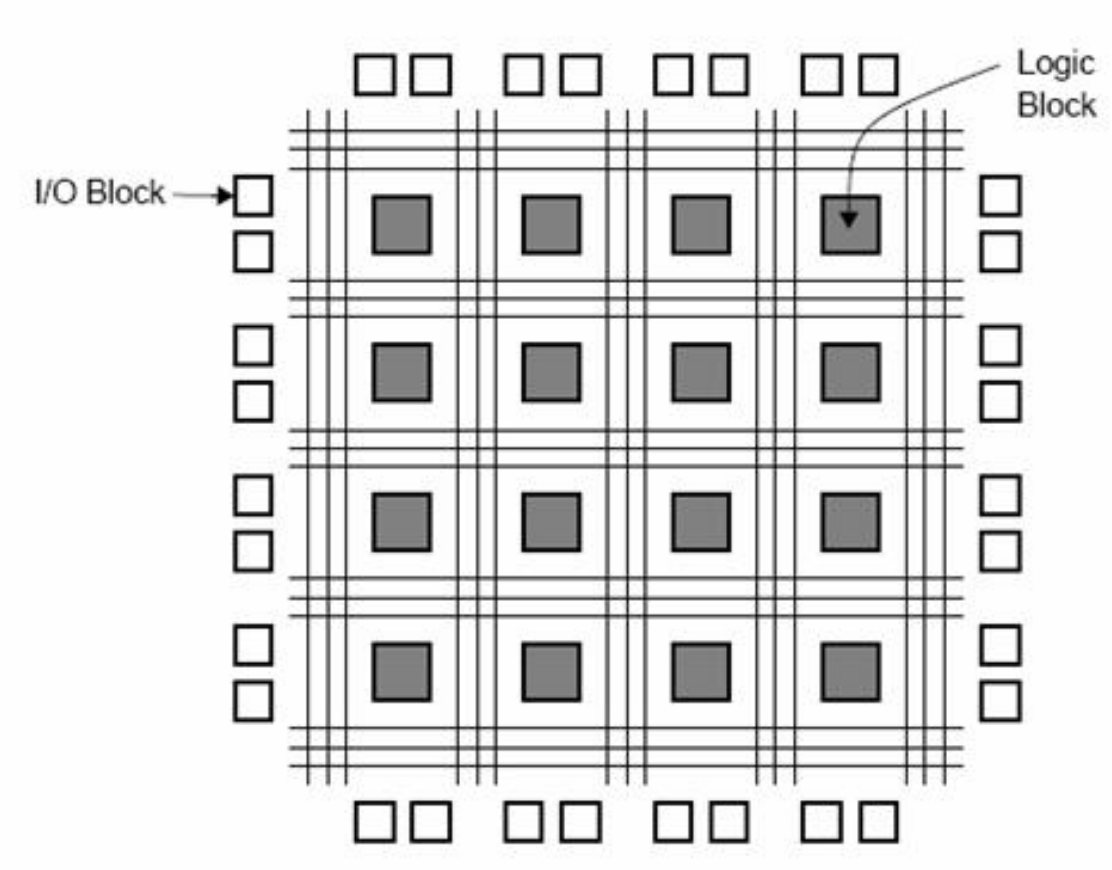


FIGURE 5.1: FPGA block array.

set number of LUTs, flip-flops and multiplexers. A LUT is a collection of logic gates hard-wired on the FPGA. LUTs store a predefined list of outputs for every combination of inputs and provide a fast way to retrieve the output of a logic operation. A flip-flop is a circuit capable of two stable states and represents a single bit. A multiplexer, also known as a mux, is a circuit that selects between two or more inputs and outputs the selected input.

Different FPGA families implement slices and LUTs differently. For example, a slice on a Virtex-II FPGA has two LUTs and two flip-flops but a slice on a Virtex-5 FPGA has four LUTs and four flip-flops. In addition, the number of inputs to a LUT, commonly two to six, depend on the FPGA family[17].The amount of logic blocks has increased drastically in the past years from approximately 8000 in 1982 to practically millions today[18].

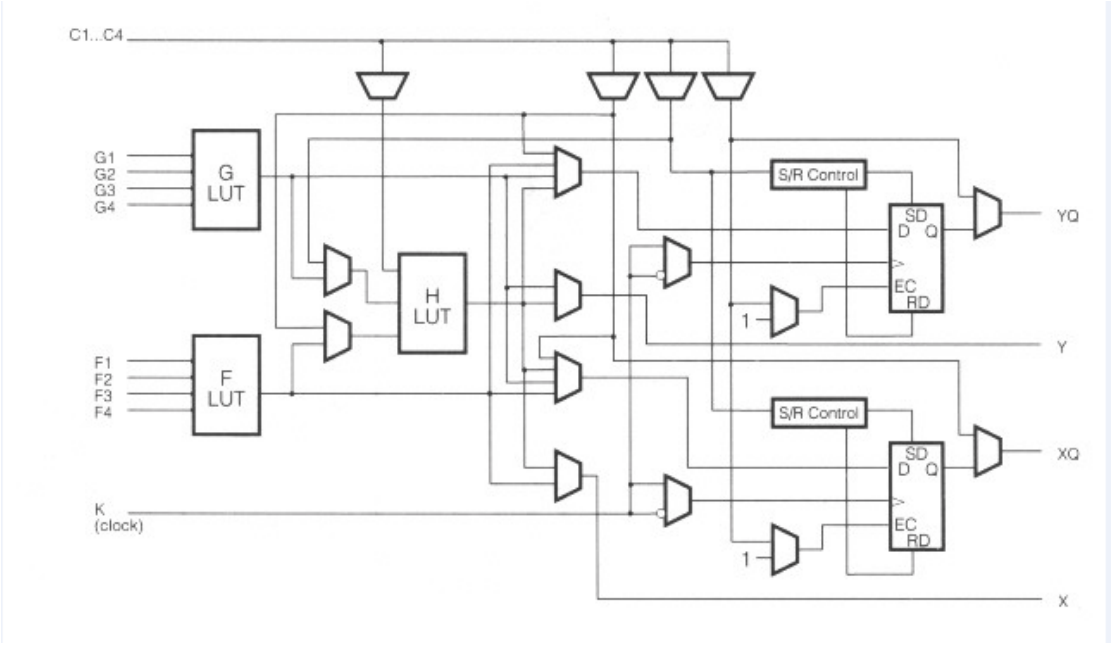


FIGURE 5.2: The basic block of a Xilinx XC4000 FPGA

### 5.2.1 History

FPGAs originated from PROMs and PLDs. The first reprogrammable hardware were the PROMs. PROM is a non-volatile memory that can be loaded with information. Different types of PROMs can be either mass programmed at the factory(Mask Programmable Devices) or by the user(Field programmable). Later came the PLDs that had a major contribution towards the creation of FPGAs. Although there are several different types of PLDs, the most common implements a set of fixed logical OR gates preceded by an array of programmable AND logic gates. Like PROM, PLDs are also manufactured as both factory programmable and user programmable. These MPLDs (Mask Programmable Logic Devices) have programmable logic for the AND gates; however, they are hard-wired between logic gates. In 1985 with the invention of FPGAs, Xilinx created a device that would not only have programmable gates, but also have programmable interconnections between gates. The FPGAs were thus simply the next step in the evolution of PLDs.

Another early field programmable logic device was proposed by Steve Casselman in 1987 to the National Science Foundation. The ideas behind the proposal were to create a computer chip that used the new technology of programmable gate arrays and was able to be completely programmed using software. Within the proposed experiment

Casselman set forth two goals: to determine a way to interconnect the planes of arrays, and to create a compiler which would be able to program functions into these new chips. Like the design of Xilinx, Casselman's chip would rely on the technology of EEPROM registers. In the coming years, aid from the Naval Surface Warfare Department was applied for, and received, to develop a computer that would implement a total of 600,000 reprogrammable array gates. In 1992 a patent was granted for this system[19].

### 5.2.2 Modern developments

Recent trend has been, to take the coarse-grained architectural approach a step further by combining the logic blocks and interconnects of traditional FPGAs with embedded microprocessors and related peripherals to form a complete "system on a programmable chip". This work mirrors the architecture by Ron Perlof and Hana Potash of Burroughs Advanced Systems Group which combined a reconfigurable CPU architecture on a single chip called the SB24. That work was done in 1982. Examples of such hybrid technologies can be found in the Xilinx Zynq$^{TM}$-7000 All Programmable SoC, which includes a 1.0 GHz dual-core ARM Cortex-A9 MPCore processor embedded within the FPGA's logic fabric or in the Altera Arria V FPGA which includes an 800 MHz dual-core ARM Cortex-A9 MPCore. The Atmel FPSLIC is another such device, which uses an AVR processor in combination with Atmel's programmable logic architecture. The Actel SmartFusion devices incorporate an ARM Cortex-M3 hard processor core (with up to 512 kB of flash and 64 kB of RAM) and analog peripherals such as a multi-channel ADC and DACs to their flash-based FPGA fabric[20].

FPGAs have opened the way to a new concept of reconfigurable computing. Reconfigurable computing is a computer architecture combining some of the flexibility of software with the high performance of hardware by processing with very flexible high speed computing fabrics like field-programmable gate arrays (FPGAs). The principal difference when compared to using ordinary microprocessors is the ability to make substantial changes to the datapath itself in addition to the control flow. On the other hand, the main difference with custom hardware, i.e. application-specific integrated circuits (ASICs) is the possibility to adapt the hardware during runtime by "loading" a new circuit on the reconfigurable fabric.

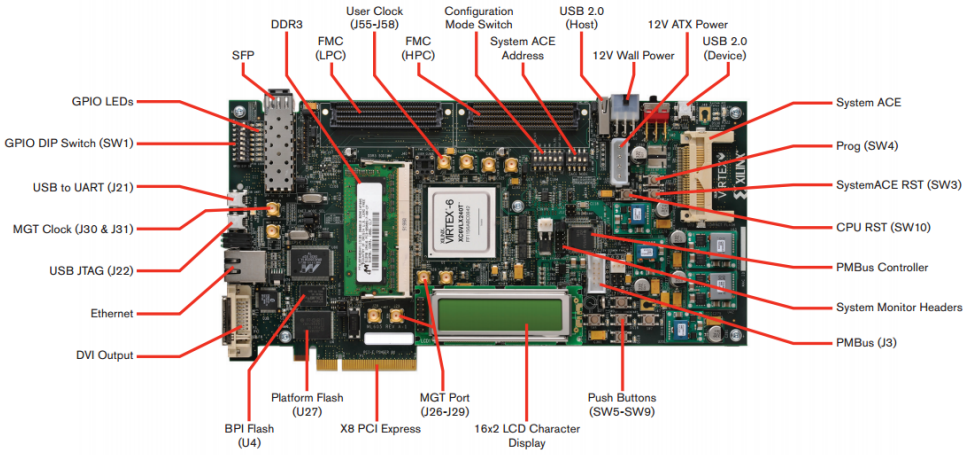


FIGURE 5.3: The Virtex 6 FPGA board

## 5.3 The Xilinx Virtex ML605

For implementing the processor, a device that would feet the needs had to be chosen. The processor being relatively extensive in size than simple designs requires a correspondingly large FPGA. Initial implementations involved the Xilinx Spartan 3A board, how ever the final design was too big to fit this board. The Virtex ML 605 board was used to synthesize and implement the design. Figure 5.3 displays the board used.

The Virtex®-6 FPGA ML605 Evaluation Kit is the Xilinx base platform for developing system designs that demand high-performance, serial connectivity and advanced memory interfacing. This yields design applications for markets such as wired telecommunications, wireless infrastructure, broadcast and many others. Integrated tools help streamline the creation of elegant solutions to complex design requirements. The ML605 Evaluation Kit is based on the XC6VLX240T-1FFG1156 Virtex-6 FPGA. This FPGA contains 241,152 logic cells, a rating that reflects the increased logic capacity offered by the 6-input LUT architecture[21].

The basic features of the board are the following:

- DDR3 SODIMM
- 16MB Platform Flash XL
- 32 MB Linear BPI Flash
- System ACE CF

- USB JTAG
- 16 x 2 LCD character display
- Video VGA
- USB host and peripheral
- Ethernet (10/100/1000) with SGMII
- GTX port with SMA x4
- 200 MHz differential clock, 66 MHz socketed oscillator, clock SMA connectors
- MGT clocking SMA x4
- VITA 57.1 FMC HPC connector
- VITA 57.1 FMC LPC connector
- PCIe® Gen1 (8-lane), Gen2 (4-lane)
- UART (via USB cable)
- IIC EEPROM
- LEDs
- DIP switch
- Pushbuttons
- System Monitor
- Power monitoring
- Power supply:12V AC adapter or 12V 4-pin ATX

## 5.4 Xilinx ISE

Xilinx ISE(Integrated Software Environment) is a software tool produced by Xilinx for synthesis and analysis of HDL designs, enabling the developer to synthesize (”compile”)

their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer. Throughout the designing, making and testing the processor this tool was used extensively since the target was to create a design that was intended to be implemented in hardware. Many HDL designs that appear functional and simulate properly in simulating programs and designs are in fact not feasible to implement in hardware. Another feature helpful to the process is the fact that ISE can detect various bugs which would not be otherwise detected easily, e.g. unconnected wires, bad connections between modules, FF latches and unused signals.

The edition that was used is the Web edition, which is a free version of Xilinx ISE that can be downloaded at no charge. It provides synthesis and programming for a limited number of Xilinx devices. In particular, devices with a large number of I/O pins and large gate matrices are disabled. The low-cost Spartan family of FPGAs is fully supported by this edition, as well as the family of CPLDs, meaning small developers and educational institutions have no overheads from the cost of development software. All the speed, power and area estimations were generated by this program.

### 5.4.1 The CORE Generator

In order to utilize the on board memory which is embedded on the FPGA board, a series of time consuming and difficult synchronization constraints and interface problems had to be dealt with. A series of approximately 100 pins and signals have to be set appropriately for the memory interface. Xilinx offers a few tools to utilize and communicate with the on board memory through the CORE Generator. The CORE Generator offers the choice to create custom Xilinx IP Cores to utilize in the project. Some of these IP Cores are used for memory interface. For our purposes the Block memory generator was used which created a block-like interface memory Figure 5.4.

In the processor design two separate memories are used, the instruction memory and the data memory. The instruction memory is 512x32bits and the data memory 1024x64bits. After following a few steps in a generating wizard the memory is created with the interface shown in figure 5.5 and the VHDL code in figure 5.6.

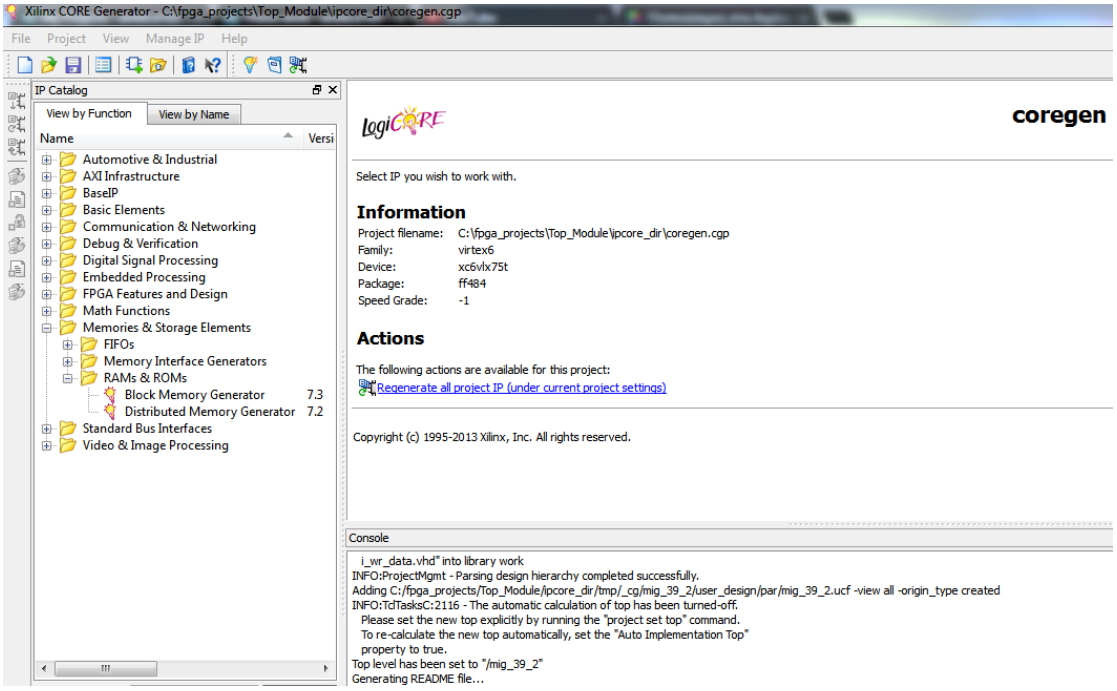


FIGURE 5.4: Block memory generator

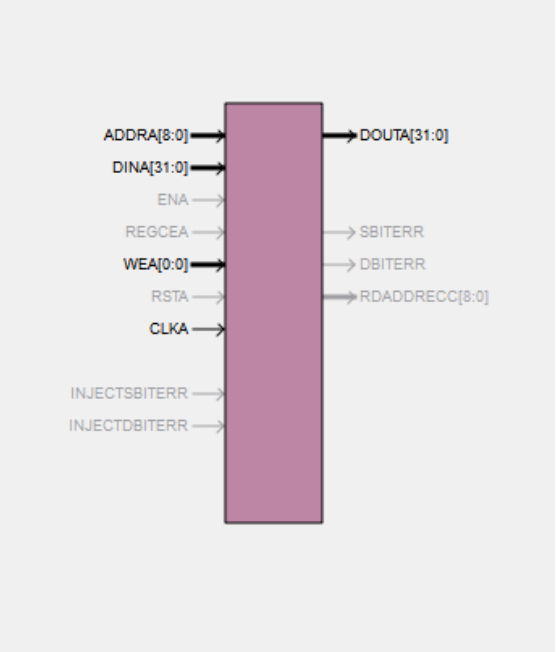


FIGURE 5.5: The instruction memory

```
component instruction_memory
  port (
        clka  : in std_logic;
        wea   : in std_logic_vector(0 downto 0);
        addra : in std_logic_vector(8 downto 0);
        dina  : in std_logic_vector(31 downto 0);
        douta : out std_logic_vector(31 downto 0)
  );
end component;
```

FIGURE 5.6: The instruction memory VHDL interface

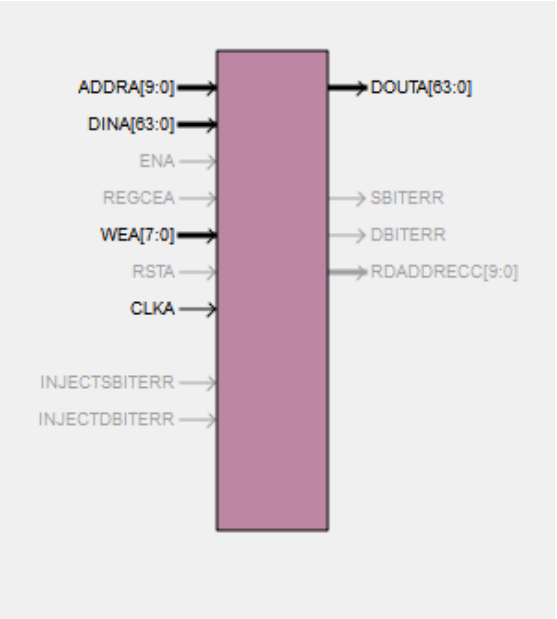


FIGURE 5.7: The data memory

```
component data_memory
  port (
    clka  : in std_logic;
    wea   : in std_logic_vector(7 downto 0);
    addra : in std_logic_vector(9 downto 0);
    dina  : in std_logic_vector(63 downto 0);
    douta : out std_logic_vector(63 downto 0)
  );
end component;
```

FIGURE 5.8: The data memory VHDL interface

The data memory was generated similarly with the difference of the write enable signal. The CORE Generator offers the choice to write single bytes of the words stored. For example a 64bit word that is stored in the memory, if a change had to be made in the first 32 bits of the word. the processor had to read the memory first, alter the bits and then store it again. By using this feature single byte word updates are possible and since the processor makes extensive use of subwords and parallel data updating, this feature is very helpful. For this reason the illegal trap that had to be raised on unaligned memory accesses according to [13] is ignored. Figure 5.7 shows the memory block and figure 5.8 shows the VHDL interface. Both memories operate at a fixed 144MHz speed.

## 5.5 Modelsim Simulation Program

Modelsim is a software package created by Mentor Graphics designed for simulating HDL modules. The program includes a text editor, waveform viewer and a RTL generator. The modelsim PE student edition was used which is freely available.

The main usage of the program was for debugging purposes. The debugging purposes involve two stages the first was the pre-synthesize debugging. In the pre-synthesize debugging the HDL modules are tested for proper function and the production of the expected results. Even if the modules operate as intended , it is not guaranteed that they will function properly in hardware implementation, as many problems could arise. After the synthesis some fixes occasionally occurred. As a result a second post-synthesis simulation was also executed reassuring the module's proper functionality.

The main debugging feature that was extensively used is the waveform viewer. This feature displays all the signals used by a component and its respective waveform. The waveform view provides a clear picture as to how the circuit and the signals evolve with time, allowing the user to easily point out problems and bugs.

# Chapter 6

# Conclusions

In this thesis a thorough and detailed report and explanation of the process of designing, implementing and customizing a processor was presented. The main purpose of this work, was to show a simple procedure that can be followed in order to exploit the flexibility offered by technologies such as the FPGAs and the HDLs, in order to adapt hardware implementations to one's needs. Many problems and possible bugs that could arise, have been pointed out and a competent solution was presented alongside. This work does not aim in the optimization of the core presented, but aims in presenting generally the procedures followed to modify such modules.

## 6.1 Acknowledgements and Compromises

The size of the soft core created, somewhat compromises its integrity. Almost certainly the processor is not bug-free, however a more than adequate implementation was presented. Since the design was created from scratch, there is no compiler or a machine code generator available, which would help the debugging process. It is crucial at this stage to point out that the design used to create the processor is not the optimum, regarding speed, area and power consumption, nor was it designed to be such. The process of optimizing requires work from many people and various fields of expertise. The guideline for the design chosen was the basic MIPS pipelined processor and was based upon this. All the optimization that occurred is in the pipeline design itself and the module placement within it.

Many compromises have occurred during the design and implementation, mainly due to time restrictions. These compromises are the following:

- No post synthesis design optimization was implemented.
- No post simulate design optimization was implemented..
- No Cache memory was utilized.
- No parallel module utilization was implemented.
- No floating point square root was implemented.
- No program interrupt mechanism was implemented.
- No efficient data hazard prediction and out of order execution was implemented.
- No compiler was implemented.
- No machine code generator/Processor testbench was implemented.

## 6.2 Future Work

In future work a few ideas have risen from this work. Some include the optimization and work on the processor itself and are summarized in the compromise list above. Others include work that targets in further exploiting the flexibility of FPGAs and offering more choices to the user. One such example is the adaptation of a processor so that we can swap and add many different module of a kind e.g. FPUs, ALUs. This will allow the engineer to test and try many modules before choosing the appropriate. Another possible future work is the adaptation of a more popular processor, e.g. picoblaze, with a similar procedure. Also the creation of self adapting HDL modules is within the possible future additions.

# Appendix A

# RTL schematics

This appendix includes all the RTL schematics used in the process of designing the processor. These include both manually drawn designs and automatically produced schematics by Xilinx IDE.

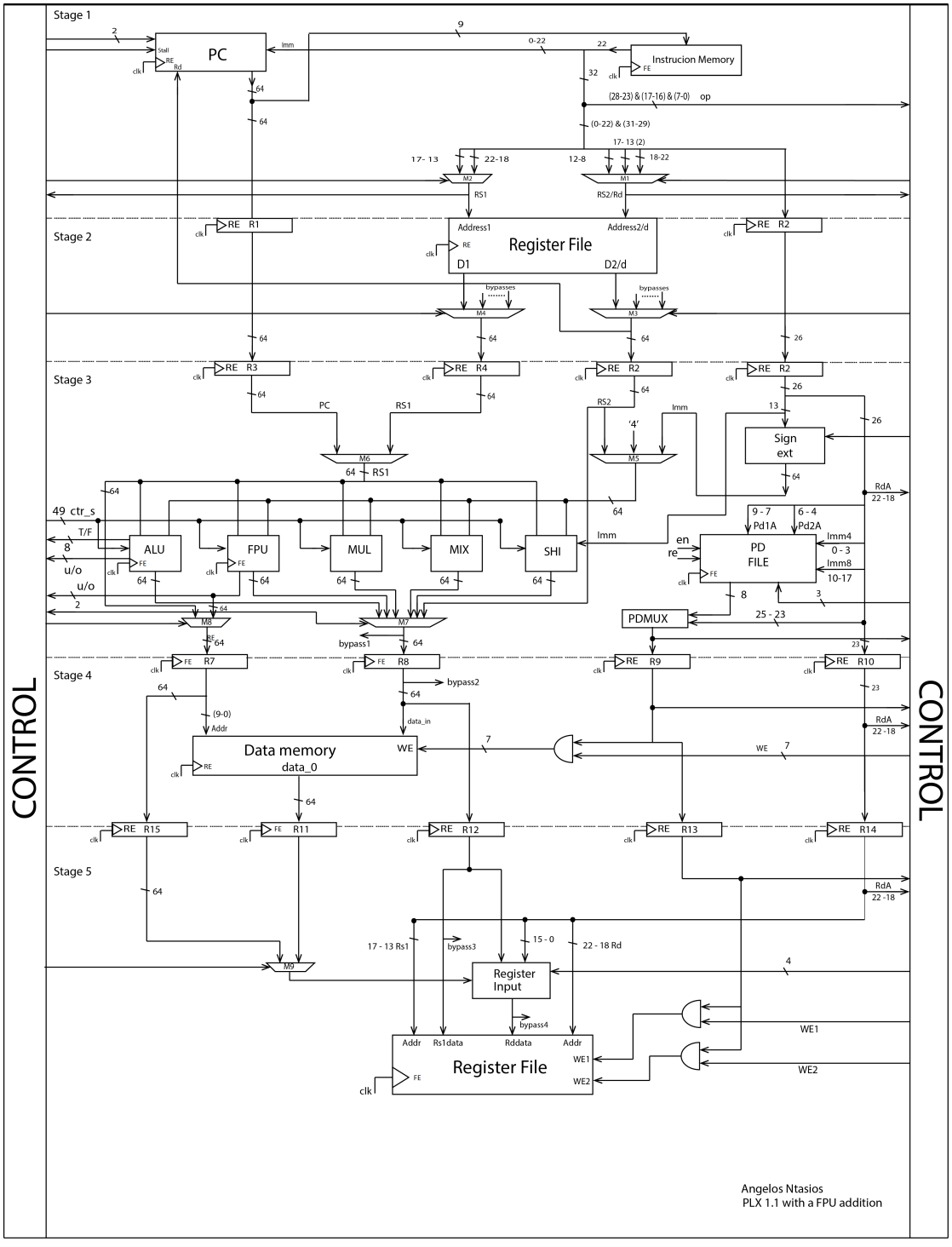


FIGURE A.1: The CPU RTL schematic

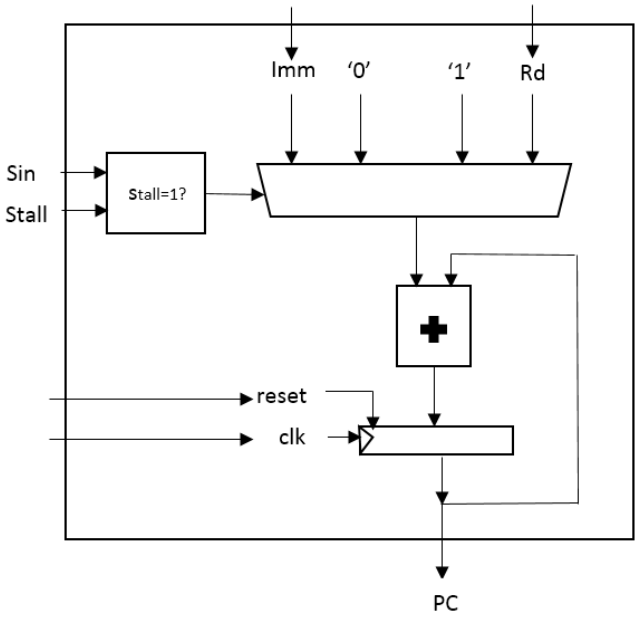


FIGURE A.2: The PC RTL schematic

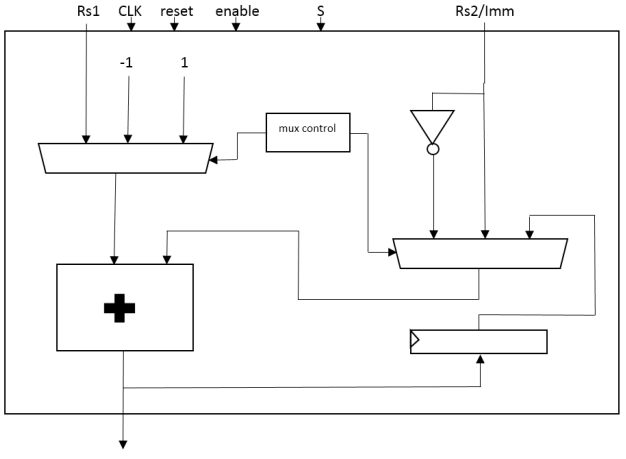


FIGURE A.3: The ALU adder RTL Schematic

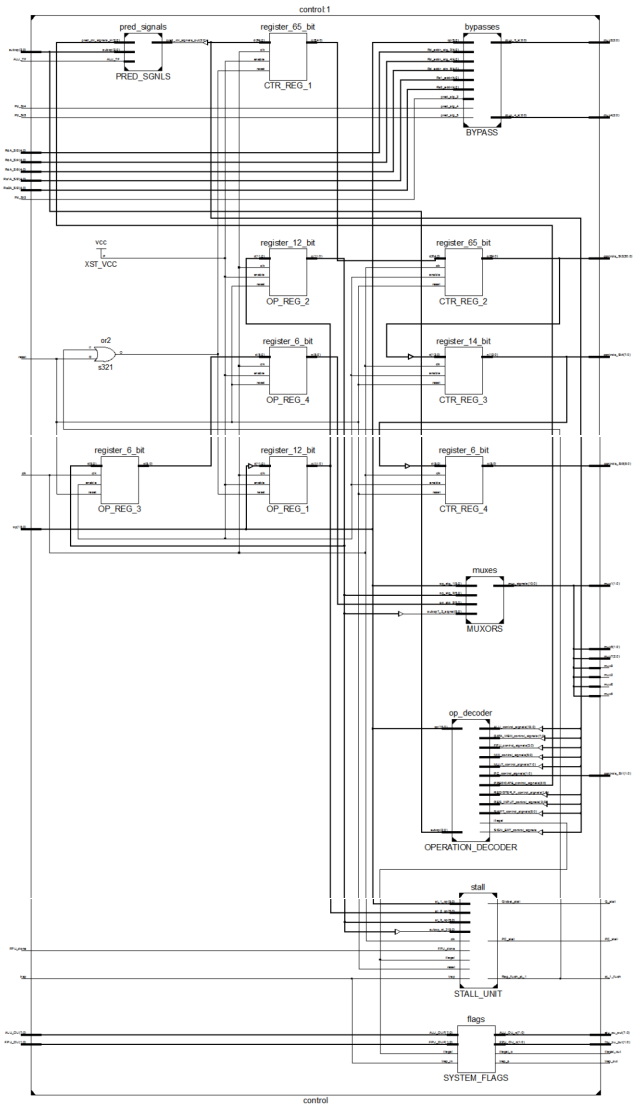


FIGURE A.4: The control RTL Schematic

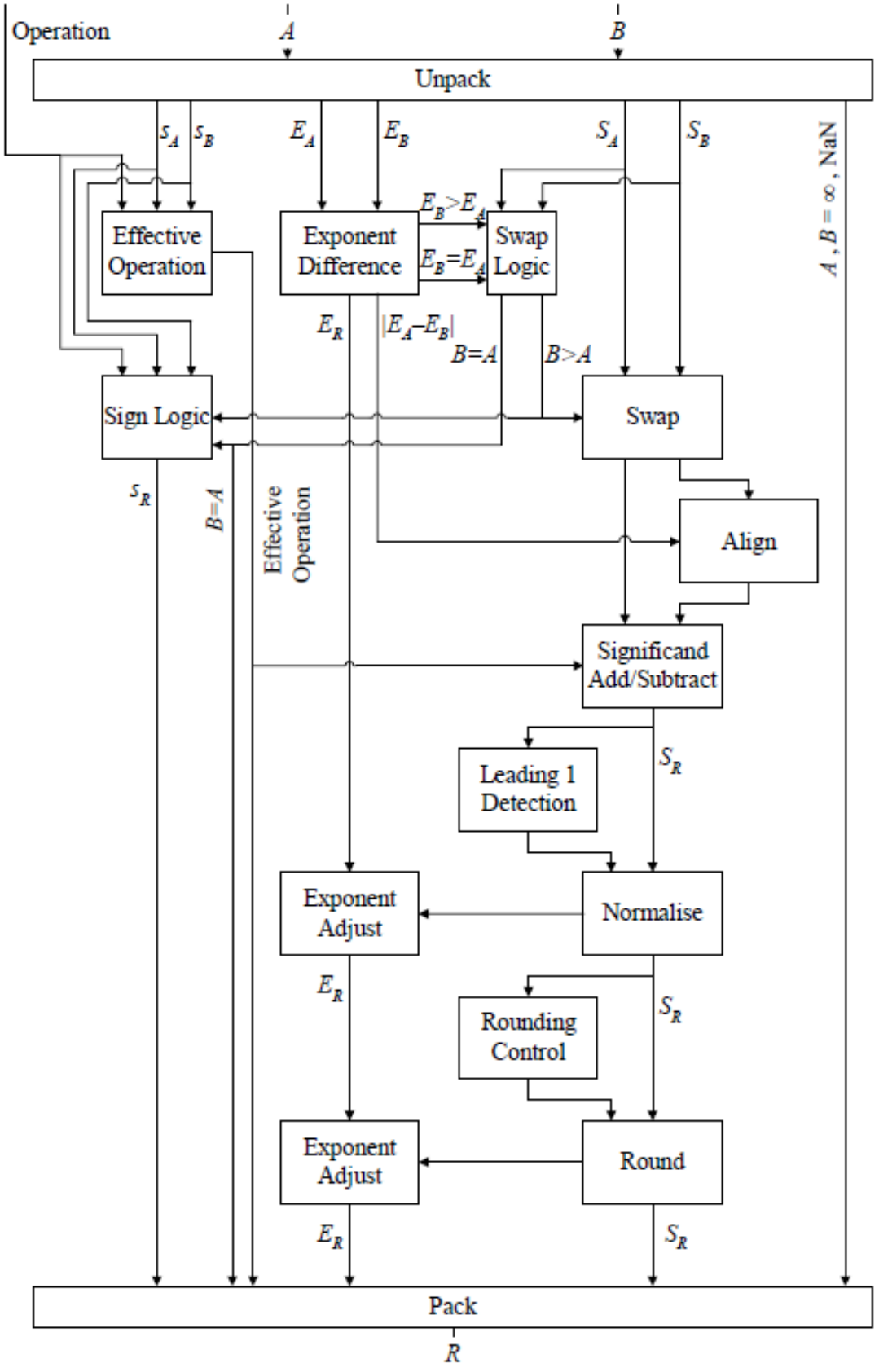


FIGURE A.5: The FPU adder

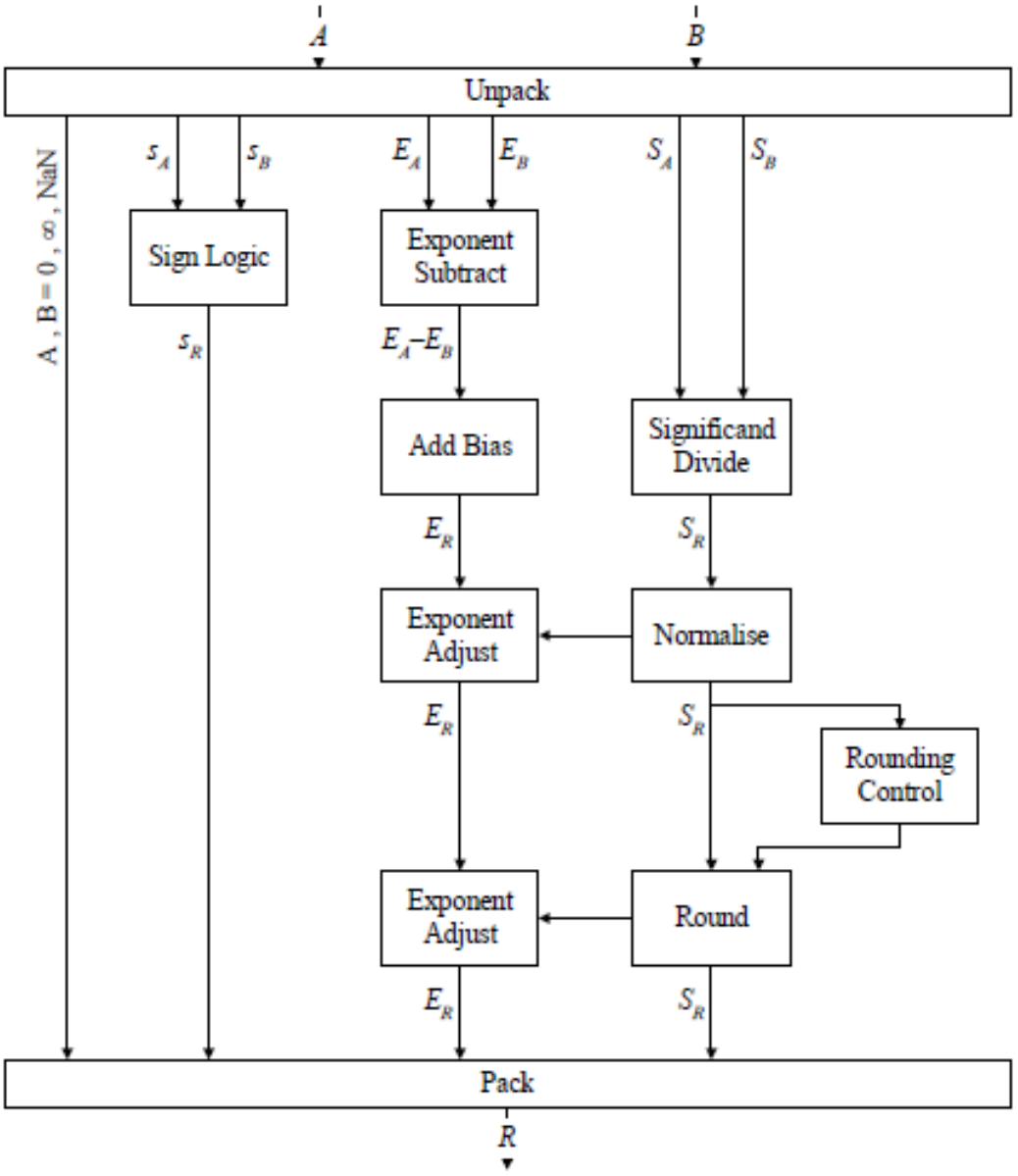


FIGURE A.6: The FPU multiplier

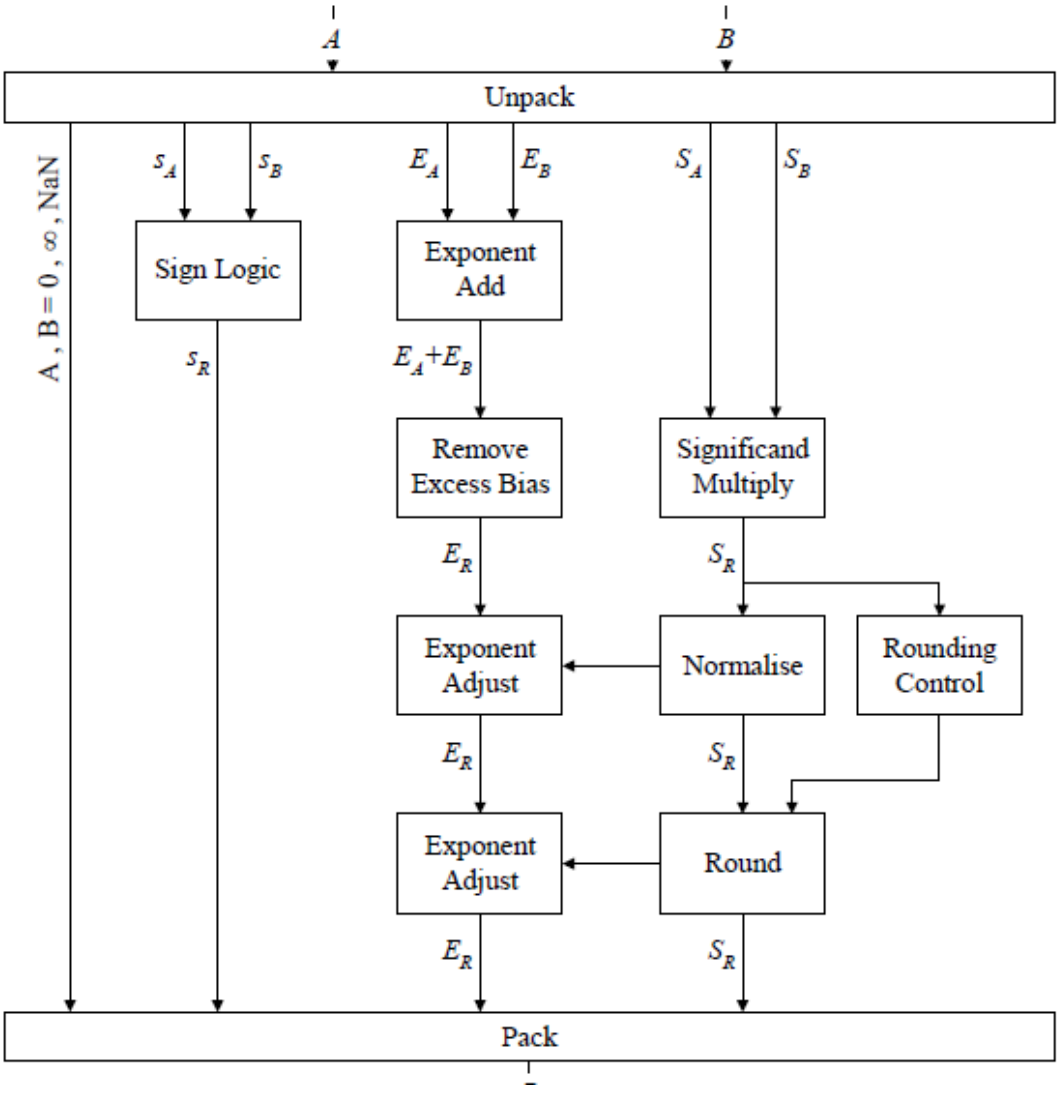


FIGURE A.7: The FPU divider

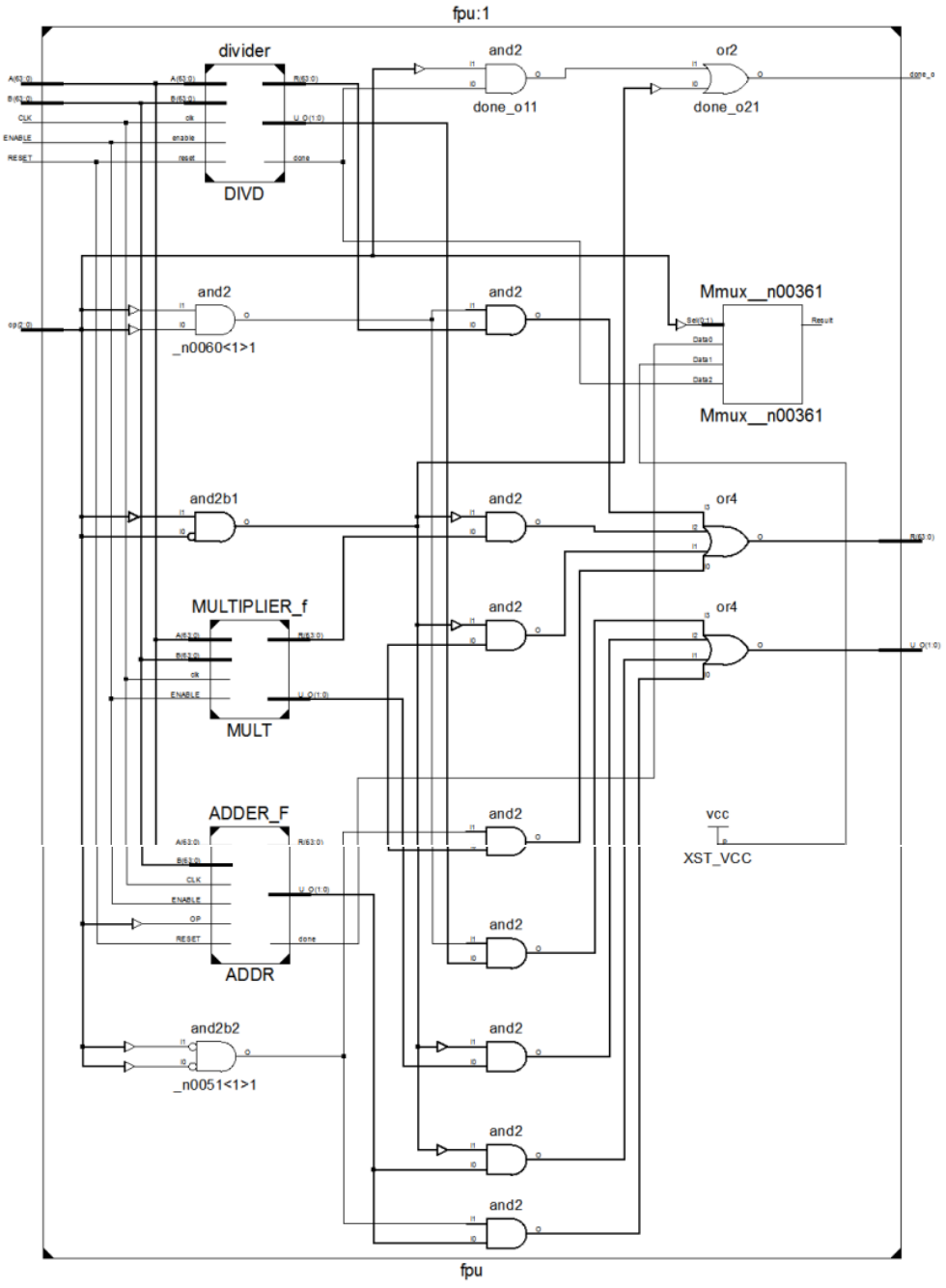


FIGURE A.8: The FPU RTL schematic

# Appendix B

# PLX 1.1 Instruction Set Architecture

This appendix contains all the instructions used by the processor without the instructions for the floating point calculations which are explained in detail in Chpater 3. A more detailed explanation of every instruction can be found at [13].

Table B.1: Main Instructions and Mnemonics

| **Mnemonic** | **Instruction** |
|---|---|
| Addi | Add Immediate |
| And | And |
| Andcm | And Complement |
| Andi | And Immediate |
| Changepr | Chane predicate Register set |
| Cmp | Compare |
| Cmpi | Compare Immediate |
| Deposit | Deposit |
| Extract | Extraxt |
| Jmp | Jump |
| Load | Load |
| Loadi | Load Immediate |
| Loadx | Load Indexed |
| Mix | Mix |
| Mux | Mux |
| Not | Not |
| Or | Or |
| Ori | Ori Immediate |
| Padd | Parallel Add |
| Paddincr | Parallel Add Increment |
| Pavg | Parallel Average |
| Pcmp | Parallel Compare |
| Perm | Permute |
| Pmax | Parallel Maximum |
| Pmin | Parallel Minimum |
| Pmul | Parallel Multiplication |
| Pmulshr | Parallel Multiply Shift Right |
| Pshift | Parallel Shift |
| Pshiftadd | Parallel Shift Add |
| Pshifti | Parallel Shift Immediate |
| Psub | Parallel Subtract |
| Psubavg | Parallel Subtract Average |
| Psubdecr | Parallel Subtract Decrement |
| Shrp | Shift Right Pair |
| Slli | Shift Left Logical Immediate |
| Srai | Shift Right Arithmetic Immediate |
| Srli | Shift Right Logical Immediate |
| Store | Store |
| Subi | Subtract Immediate |
| Testbit | Testbit |
| Trap | Trap |
| Xor | Xor |
| Xori | Xor Immediate |

# Appendix C

# Segments of Code

### The VHDL testbench

```
--FPU.ADD/SUB_testbench--
--ANGELOS NTASIOS--

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;

entity add_sub_testbench is

end;

architecture add_sub_testb of add_sub_testbench is

component ADDER_F is
        port (
                A           : in std_logic_vector(63 downto 0);         --A
                B           : in std_logic_vector(63 downto 0);         --B
                OP              : in std_logic;
                R               : out std_logic_vector(63 downto 0);    --R
                U_O             : out std_logic_vector(1 downto 0);     --underflow/overflow
                CLK             : in  std_logic;        --Clock
                ENABLE  : in  std_logic;        --ENABLE
                RESET   : in  std_logic;        --RESET(Global)
                done    : out std_logic
        );
end component;

signal clk_g : std_logic := '0';
```

```
signal A_in : std_logic_vector(63 downto 0);
signal B_in : std_logic_vector(63 downto 0);
signal OP_in : std_logic;
signal reset_in : std_logic;
signal R_out : std_logic_vector(63 downto 0);
signal O_U_out : std_logic_vector(1 downto 0);
signal done_out : std_logic;
signal clk_g_in : std_logic;
begin
clk_g <= not clk_g after 500 ns;
ADDER_top_module : ADDER_F port map(A_in, B_in, OP_in, R_out, O_U_out,
clk_g_in, '1', reset_in, done_out);

process
variable reset_done : bit := '0';

file      infile   : text open read_mode is path&"P A T H\testbench.txt";
variable  inline   : line;
file      outfile  : text open write_mode is path&"P A T H\testbench_results_VHDL.txt";
variable  outline  : line;
variable  tmp1 : bit_vector(63 downto 0);
variable  tmp2 : string(2 downto 2);
variable tmp3 : string(2 downto 2);
begin
        wait until clk_g = '1' and clk_g'event;


                        if (not endfile(infile)) then
                                readline(infile, inline);
                        end if;

                        for i in inline'range loop
                                if (i < 65) then
                                        if (inline(i) = '1') then
                                                A_in(64-i) <= '1';
                                        else
                                                A_in(64-i) <= '0';
                                        end if;
                                end if;

                                if (i = 66) then
                                        if (inline(i) = '1') then
                                                OP_in <= '1';
                                        else
                                                OP_in <= '0';
                                        end if;
                                end if;
```

```
                        if (i > 67 and i < 132) then
                                if (inline(i) = '1') then
                                        B_in(131-i) <= '1';
                                else
                                        B_in(131-i) <='0';
                                end if;
                        end if;
                end loop;

                if (reset_done = '0') then
                        reset_in <= '1';
                        reset_done := '1';
                else
                        reset_in <= '0';
                end if;
                wait for 50 ns;
                clk_g_in <= '1';
                wait for 50 ns;
                reset_in <= '0';
                clk_g_in <= '0';
                wait for 50 ns;
                clk_g_in <= '1';
                wait for 50 ns;
                clk_g_in <= '0';
                wait for 50 ns;
                clk_g_in <= '1';
                wait for 50 ns;
                clk_g_in <= '0';

                if (O_U_out(1)='1') then
                        tmp2(2) := '1';
                else
                        tmp2(2) := '0';
                end if;
                if (O_U_out(0)='1') then
                        tmp3(2) := '1';
                else
                        tmp3(2) := '0';
                end if;

                tmp1 := to_bitvector(R_out);
                write(outline, tmp1);
                write(outline, " U="&tmp2(2)&" O="&tmp3(2));
                writeline(outfile, outline);
end process;
end;
```

```
when "101100" =>--addf
        PC_control_signals          <= "00";--program counter(+4)
        ALU_control_signals         <= "0XXXXXXXXXXXXXXXXXXX";--ALU not enabled
        MULT_control_signals        <= "0XXXXXXX";--Multiplier not enabled
        SHIFT_control_signals       <= "0XXXXXXXX";--Shifter not enabled
        MIX_control_signals         <= "0XXXXX";--MIX/MUX not enabled
        FPU_control_signals         <= "1000";--FPU enabled
        SIGN_EXT_control_signals    <= 'X';--SIGN_EXTENSION not enabled
        DATA_MEM_control_signals    <= "00000000";--DATA MEMORY not enabled
        REGISTER_F_control_signals  <= "01";--REGISTER FILE write enable enabled
        PREDICATE_control_signals   <= "000";--PREDICATE FILE NULL op
        REG_INPUT_control_signals   <= "0000";--register input Rs2
        illegal                                 <= '0';--instruction OK
when "101101" =>--subf
        PC_control_signals          <= "00";--program counter(+4)
        ALU_control_signals         <= "0XXXXXXXXXXXXXXXXXXX";--ALU not enabled
        MULT_control_signals        <= "0XXXXXXX";--Multiplier not enabled
        SHIFT_control_signals       <= "0XXXXXXXX";--Shifter not enabled
        MIX_control_signals         <= "0XXXXX";--MIX/MUX not enabled
        FPU_control_signals         <= "1001";--FPU enabled
        SIGN_EXT_control_signals    <= 'X';--SIGN_EXTENSION not enabled
        DATA_MEM_control_signals    <= "00000000";--DATA MEMORY not enabled
        REGISTER_F_control_signals  <= "01";--REGISTER FILE write enable enabled
        PREDICATE_control_signals   <= "000";--PREDICATE FILE NULL op
        REG_INPUT_control_signals   <= "0000";--register input Rs2
        illegal                                 <= '0';--instruction OK
when "101110" =>--multf
        PC_control_signals          <= "00";--program counter(+4)
        ALU_control_signals         <= "0XXXXXXXXXXXXXXXXXXX";--ALU not enabled
        MULT_control_signals        <= "0XXXXXXX";--Multiplier not enabled
        SHIFT_control_signals       <= "0XXXXXXXX";--Shifter not enabled
        MIX_control_signals         <= "0XXXXX";--MIX/MUX not enabled
        FPU_control_signals         <= "1010";--FPU enabled
        SIGN_EXT_control_signals    <= 'X';--SIGN_EXTENSION not enabled
        DATA_MEM_control_signals    <= "00000000";--DATA MEMORY not enabled
        REGISTER_F_control_signals  <= "01";--REGISTER FILE write enable enabled
        PREDICATE_control_signals   <= "000";--PREDICATE FILE NULL op
        REG_INPUT_control_signals   <= "0000";--register input Rs2
        illegal                                 <= '0';--instruction OK
when "101111" =>--divf
        PC_control_signals          <= "00";--program counter(+4)
        ALU_control_signals         <= "0XXXXXXXXXXXXXXXXXXX";--ALU not enabled
        MULT_control_signals        <= "0XXXXXXX";--Multiplier not enabled
        SHIFT_control_signals       <= "0XXXXXXXX";--Shifter not enabled
        MIX_control_signals         <= "0XXXXX";--MIX/MUX not enabled
        FPU_control_signals         <= "1011";--FPU enabled
        SIGN_EXT_control_signals    <= 'X';--SIGN_EXTENSION not enabled
        DATA_MEM_control_signals    <= "00000000";--DATA MEMORY not enabled
        REGISTER_F_control_signals  <= "01";--REGISTER FILE write enable enabled
        PREDICATE_control_signals   <= "000";--PREDICATE FILE NULL op
        REG_INPUT_control_signals   <= "0000";--register input Rs2
        illegal                                 <= '0';--instruction OK
```

FIGURE C.1: The FPU instruction signals

```
--------------LIBRARIES---------------------------
library ieee;
use ieee.std_logic_1164.all;

entity RTL_test is
        port(
                A : in std_logic;
                B : in std_logic;
                A_out : out std_logic;
                B_out : out std_logic;
                clk : in std_logic;
                reset : in std_logic
        );
end;

architecture rtl of RTL_test is

component register_1_bit is
   port (
      d          : in std_logic;
      reset  : in std_logic;
          enable : in std_logic;
      clk    : in std_logic;
      q      : out std_logic
         );
end component;


signal clk_fe : std_logic;

signal RA1toRA2 : std_logic;
signal RA2toRA3 : std_logic;
signal RA3toRA4 : std_logic;
signal RA4toRA5 : std_logic;

signal RB1toRB2 : std_logic;
signal RB2toRB3 : std_logic;
signal RB3toRB4 : std_logic;
signal RB4toRB5 : std_logic;
signal RB5toRB6 : std_logic;
signal RB6toRB7 : std_logic;

begin
process(clk)
begin
        if (clk'event and clk = '1') then
                clk_fe <= '0';
        elsif(clk'event and clk = '0') then
                clk_fe <= '1';
        end if;
end process;

RA1 : register_1_bit port map (A , reset, '1', clk, RA1toRA2);
RA2 : register_1_bit port map (RA1toRA2 , reset, '1', clk, RA2toRA3);
RA3 : register_1_bit port map (RA2toRA3 , reset, '1', clk, RA3toRA4);
RA4 : register_1_bit port map (RA3toRA4 , reset, '1', clk, RA4toRA5);
RA5 : register_1_bit port map (RA3toRA4 , reset, '1', clk, A_out);

RB1 : register_1_bit port map (B , reset, '1', clk, RB1toRB2);
RB2 : register_1_bit port map (RB1toRB2 , reset, '1', clk_fe, RB2toRB3);
RB3 : register_1_bit port map (RB2toRB3 , reset, '1', clk, RB3toRB4);--rf
RB4 : register_1_bit port map (RB3toRB4 , reset, '1', clk, RB4toRB5);--r4
RB5 : register_1_bit port map (RB4toRB5 , reset, '1', clk_fe, RB5toRB6);--r8
RB6 : register_1_bit port map (RB5toRB6 , reset, '1', clk, RB6toRB7);--mem
RB7 : register_1_bit port map (RB6toRB7 , reset, '1', clk_fe, B_out);--r11

end;
```

FIGURE C.2: The RTL mirror design

# Appendix D

# Module control signals

TABLE D.1: ALU control signals

| INSTR | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| addi | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| and/andi | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | X | X | X | X | X | 0 | X | X | X | X | X | X |
| andcm | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | X | X | X | X | X | 0 | X | X | X | X | X | X |
| cmp/cmpi.eq | 0 | 0 | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | X | X | X |
| cmp/cmpi.ne | 0 | 0 | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | X | X | X | X | X | X |
| cmp/cmpi.lt | 0 | 0 | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X |
| cmp/cmpi.le | 0 | 0 | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | X | X | X | X | X | X |
| cmp/cmpi.gt | 0 | 0 | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | X | X | X | X | X | X |
| cmp/cmpi.ge | 0 | 0 | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | X | X | X | X | X | X |
| cmp/cmpi.ltu | 0 | 0 | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | X | X | X | X | X | X |
| cmp/cmpi.leu | 0 | 0 | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | X | X | X | X | X | X |
| cmp/cmpi.gtu | 0 | 0 | 1 | X | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | X | X | X | X | X | X |
| cmp/cmpi.geu | 0 | 0 | 1 | X | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | X | X | X | X | X | X |
| Loadx.4/load.4 | 1 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | X | X | X | X | X | 0 | 1 |
| Loadx.8/load.8 | 1 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | X | X | X | X | X | 0 | 0 |
| Load.x.u.4/load.u.4 | 0 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | X | X | X | X | X | 0 | 1 |
| Load.x.u.8/load.u.8 | 0 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | X | X | X | X | X | 0 | 0 |
| Store.1 | 1 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | X | X | X | X | X | 1 | 1 |
| Store.2 | 1 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | X | X | X | X | X | 1 | 0 |
| Store.4 | 1 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | X | X | X | X | X | 0 | 1 |
| Store.8 | 1 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | X | X | X | X | X | 0 | 0 |
| Store.u.1 | 0 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | X | X | X | X | X | 1 | 1 |
| Store.u.2 | 0 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | X | X | X | X | X | 1 | 0 |
| Store.u.4 | 0 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | X | X | X | X | X | 0 | 1 |
| Store.u.8 | 0 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | X | X | X | X | X | 0 | 0 |
| not | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | X | X | X | X | X | 0 | X | X | X | X | X | X |
| or/ori | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | X | X | X | X | X | 0 | X | X | X | X | X | X |
| padd.1 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | X | 1 | 1 | 1 |
| padd.2 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | X | 1 | 1 | 0 |
| padd.4 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | X | 1 | 0 | 1 |
| padd.8 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | X | 1 | 0 | 0 |
| padd.1.u/pavg.1 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| padd.2.u/pavg.2 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| padd.4.u | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| padd.8.u | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| padd.1.s | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| padd.2.s/pshiftadd | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| padd.4.s | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| padd.8.s | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| paddincr.1 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 1 | X | 1 | 1 | 1 |
| paddincr.2 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 1 | X | 1 | 1 | 0 |
| paddincr.4 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 1 | X | 1 | 0 | 1 |
| paddincr.8 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 0 | 1 | X | 1 | 0 | 0 |
| pavg.1 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| pavg.2 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| pavg.raz.1 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| Pavg.raz.2 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| pcmp.1.eq | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 0 | 1 | 0 | X | X | X | X | 1 | 1 |
| pcmp.2.eq | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 0 | 1 | 0 | X | X | X | X | 1 | 0 |
| pcmp.4.eq | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 0 | 1 | 0 | X | X | X | X | 0 | 1 |
| pcmp.8.eq | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 0 | 1 | 0 | X | X | X | X | 0 | 0 |
| pcmp.1.gt | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 0 | 0 | 0 | X | X | X | X | 1 | 1 |
| pcmp.2.gt | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 0 | 0 | 0 | X | X | X | X | 1 | 0 |
| pcmp.4.gt | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 0 | 0 | 0 | X | X | X | X | 0 | 1 |
| pcmp.8.gt | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 0 | 0 | 0 | X | X | X | X | 0 | 0 |
| pmax.1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 1 | 1 | 0 | X | X | X | X | 1 | 1 |
| pmax.2 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 1 | 1 | 0 | X | X | X | X | 1 | 0 |
| pmin.1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 1 | 0 | 0 | X | X | X | X | 1 | 1 |
| pmin.2 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | X | X | 1 | 0 | 0 | X | X | X | X | 1 | 0 |
| psub.1 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 0 | X | 1 | 1 | 1 |
| psub.2 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 0 | X | 1 | 1 | 0 |
| psub.4 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 0 | X | 1 | 0 | 1 |
| psub.8 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 0 | X | 1 | 0 | 0 |
| psub.1.u | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| psub.2.u | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| psub.4.u | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| psub.8.u | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| psub.1.s | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| psub.2.s | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| psub.4.s | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| psubavg.1 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| psubavg.2 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| psubdecr.1 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 1 | X | 1 | 1 | 1 |
| psubdecr.2 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 1 | X | 1 | 1 | 0 |
| psubdecr.4 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 1 | X | 1 | 0 | 1 |
| psubdecr.8 | 0 | 0 | 1 | 0 | 0 | X | X | X | X | X | X | X | X | 0 | 1 | 1 | X | 1 | 0 | 0 |
| store1 | 1 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| store2 | 1 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| store.u.1 | 0 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| store.u.2 | 0 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| store.u.4 | 0 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| store.u.8 | 0 | 0 | 1 | 0 | 1 | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| testbit | 0 | 1 | 0 | X | 0 | X | X | X | X | X | X | X | X | 0 | X | X | X | X | X | X |
| Xor/xori | 0 | 0 | 1 | 1 | X | 1 | 0 | 1 | X | X | X | X | X | 0 | X | X | X | X | X | X |

Table D.2: FPU Control Signals

| INSTR | 2 | 1 | 0 |
|---|---|---|---|
| Add.f | 0 | 0 | 0 |
| Sub.f | 0 | 0 | 1 |
| Mult.f | 0 | 1 | 0 |
| div.f | 0 | 1 | 1 |

Table D.3: Shifter Control Signals

| INSTR | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Pshift.l.2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Pshift.l.4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Pshift.l.8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Pshift.r.2 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| Pshift.r.4 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| Pshift.r.8 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Pshift.r.a.2 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| Pshift.r.a.4 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| Pshift.r.a.8 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| Pshiftadd.l.1 | 0 | 0 | 1 | X | X | 0 | X | X |
| Pshiftadd.l.2 | 0 | 1 | 0 | X | X | 0 | X | X |
| Pshiftadd.l.3 | 0 | 1 | 1 | X | X | 0 | X | X |
| Pshiftadd.r.1 | 0 | 0 | 1 | X | X | 1 | X | X |
| Pshiftadd.r.2 | 0 | 1 | 0 | X | X | 1 | X | X |
| Pshiftadd.r.3 | 0 | 1 | 1 | X | X | 1 | X | X |
| Pshifti.l.2 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| Pshifti.l.4 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| Pshifti.l.8 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Pshifti.r.2 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| Pshifti.r.4 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| Pshifti.r.8 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| Pshifti.r.a.2 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| Pshifti.r.a.4 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| Pshifti.r.a.8 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| Shrp | 1 | 0 | 0 | X | X | X | X | X |
| Slli | 1 | 0 | 1 | X | X | X | X | X |
| Srai | 1 | 1 | 0 | X | X | X | X | X |
| Srli | 1 | 1 | 1 | X | X | X | X | X |

Table D.4: Mix Unit Control Signals

| INSTR | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|
| mix.l.1 | 0 | 0 | 0 | 1 | 1 |
| Mix.l.2 | 0 | 0 | 0 | 1 | 0 |
| mix.l.4 | 0 | 0 | 0 | 0 | 1 |
| mix.r.1 | 0 | 0 | 1 | 1 | 1 |
| mix.r.2 | 0 | 0 | 1 | 1 | 0 |
| mix.r.4 | 0 | 0 | 1 | 0 | 1 |
| permute | 1 | 1 | 1 | X | X |
| Mux.rev | 0 | 1 | 0 | X | X |
| Mux.mix | 0 | 1 | 1 | X | X |
| Mux.shuf | 1 | 0 | 0 | X | X |
| Mux.alt | 1 | 0 | 1 | X | X |
| Mux.1.brcst | 1 | 1 | 0 | 1 | 1 |
| Mux.2.prcst | 1 | 1 | 0 | 1 | 0 |

Table D.5: Multiplier Control Signals

| INSTR | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| Pmul.odd | X | 0 | X | X | 0 | 0 | 0 |
| Pmul.odd.u | X | 0 | X | X | 0 | 0 | 1 |
| pmul.even | X | 0 | X | X | 0 | 1 | 0 |
| Pmul.even.u | X | 0 | X | X | 0 | 1 | 1 |
| Pmulshr.0 | 0 | 1 | 0 | 0 | 1 | X | X |
| Pmulshr.8 | 0 | 1 | 0 | 1 | 1 | X | X |
| Pmulshr.15 | 0 | 1 | 1 | 0 | 1 | X | X |
| Pmulshr.16 | 0 | 1 | 1 | 1 | 1 | X | X |
| Pmulshr.a.0 | 1 | 1 | 0 | 0 | 1 | X | X |
| Pmulshr.a.8 | 1 | 1 | 0 | 1 | 1 | X | X |
| Pmulshr.a.15 | 1 | 1 | 1 | 0 | 1 | X | X |
| Pmulshr.a.16 | 1 | 1 | 1 | 1 | 1 | X | X |

# Bibliography


[1] 2014. URL http://en.wikipedia.org/wiki/Embedded_system.

[2] 2014. URL http://www.embeddedcraft.org/ES%20Trends.pdf.

[3] M. Khalid J. Tong, I. Anderson. Soft-core processors for embedded systems, in microelectronics. 2006.

[4] 2014. URL http://en.wikipedia.org/wiki/Semiconductor_intellectual_property_core.

[5] Xilinx, 2014. URL http://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf.

[6] ESA, 2014. URL http://en.wikipedia.org/wiki/LEON.

[7] 2014. URL http://en.wikipedia.org/wiki/OpenRISC.

[8] Ντελή Χασάν Μουσταφά Μουτλού. ΑΡΙΘΜΟΙ και ΑΡΙΘΜΗΤΙΚΗ ΚΙΝΗΤΗΣ Υ-ΠΟΔΙΑΣΤΟΛΗΣ.

[9] 2014. URL http://palms.ee.princeton.edu/node/22.

[10] Paschalakis, s. lee, p. double precision floating-point arithmetic on fpgas, in proc. 2003 2nd ieee international conference on field programmable technology (fpt '03), tokyo, japan, dec. 15-17, pp. 352-358, 2003.

[11] 2014. URL http://palms.princeton.edu/system/files/PLX+1.1+ISA+Encoding.pdf.

[12] Zhijie Shi Xiao Yang Ruby B. Lee, A. Murat Fiskiran. Refining instruction set architecture for high-performance multimedia proceswsing in constrained environments.

[13] *The PLX ISA*, 2014. URL http://palms.princeton.edu/system/files/PLX+1.1+ISA+Reference.pdf.

[14] Jeff Sondeen Jeff Draper Joong-Seok Moon, Taek-Jun Kwon. An area-efficient standard-cell floating-point unit design for a processing-in-memory system.

[15] Minas Dasygenis Angelos Ntasios. "design, implementation and verification of a customizing ip soft core with fpu support". *Ecescon 7*, 2014.

[16] Hardware description language history. URL http://en.wikipedia.org/wiki/Hardware_description_language#History.

[17] Basic fpga block cell design, 2014. URL http://zone.ni.com/reference/en-XX/help/371599G-01/lvfpgaconcepts/fpga_basic_chip_terms/.

[18] Clive Maxfield. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. Elsevier, 2004.

[19] Fpga history, 2014. URL http://web.archive.org/web/20070412183416/http://filebox.vt.edu/users/tmagin/history.htm.

[20] Short description of fpgas, 2014. URL http://en.wikipedia.org/wiki/Field-programmable_gate_array.

[21] Xilinx, 2014. URL http://www.xilinx.com/support/documentation/boards_and_kits/ug535.pdf.