

FPGA BASED FLOATING POINT ARITHMETIC UNIT FOR TURBINE EFFICIENCY MEASUREMENT

A DISSERTATION

*Submitted in partial fulfillment of the
requirements for the award of the degree
of*
MASTER OF TECHNOLOGY
in
ELECTRICAL ENGINEERING
(With Specialization in Measurement & Instrumentation)

By
LOKESH SHARMA

IP



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE-247 667 (INDIA)

JUNE, 2006

Dedicated to my Grandfather, Shri Shiv kumar Sharma

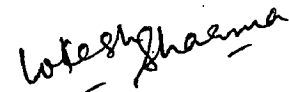
CANDIDATE'S DECLARATION

I hereby declare that the work which is being presented in this dissertation entitled, "FPGA BASED FLOATING POINT ARITHMETIC UNIT FOR TURBINE EFFICIENCY MEASUREMENT", submitted towards the partial fulfillment of the requirements for the award of the degree of **Master of Technology in Electrical Engineering**, with specialization in **Measurement & Instrumentation**, I.I.T. Roorkee, India is an authentic record of my own work carried out from June 2005 to June 2006 under the supervision of **Dr. H. K. Verma**, Professor; Electrical Engineering Department, Indian Institute of Technology, Roorkee, India and **Dr. R. S. Anand**, Associate Professor, Electrical Engineering Department, Indian Institute of Technology, Roorkee, India.

The matter embodied in this dissertation report has not been submitted by me for the award of any other degree or diploma.

Place: Roorkee

Dated: June 30, 2006



LOKESH SHARMA

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of our knowledge.



Dr. R. S. Anand

Associate Professor

Electrical Engineering Department

IIT Roorkee

Roorkee-247667 (India)



Dr. H. K. Verma 20/6/06

Professor

Electrical Department

IIT Roorkee

Roorkee-247667 (India)

ABSTRACT

Due to inherent limitations of Fixed-point representation, it is sometimes desirable to perform arithmetic operations in the floating-point format. Although an established standard for floating-point arithmetic operations exist, the growing demand for high-performance computing platforms has pushed the computing community to work upon new architectures and algorithms for floating point arithmetic operations. Performing the arithmetic operations on IEEE Floating-point numbers imposed challenges beyond the challenges of Fixed-Point arithmetic. These challenges particularly include the task of normalization and IEEE compliant rounding. For some time now the researchers have been working on use of FPGAs to solve the problem. The presented work is also exploring an application area of FPGA to develop independent System on Programmable Chip (SOPC) design. This work describes the implementation of Floating-point arithmetic unit in FPGA chip, using VHDL programming on Xilinx ISE 7.1 platform supported by Modelsim and Aldec Active HDL simulation environment. Besides implementing the addition, subtraction, multiplication, division, square root, and absolute unit, some other supporting units like general purpose registers, control registers, tag register, status register etc are also implemented to make it work in stand-alone mode. This feature also provides flexibility of writing programs to the end user. The input/output number format confirms IEEE-754 standard single precision real numbers. Internally, calculations are performed according to IEEE-754 standard double-extended precision real numbers (as incorporated in Intel Pentium4 processor). This inherited feature assists floating-point arithmetic unit in enhancing the accuracy. A special care has been taken through the tag word. The tag register checks the validity of number before performing a complex arithmetic computation and thereby saves clock cycles in case the data register is empty or contains zero, infinity or invalid number. I also

implemented the normalization unit and all four possible rounding modes. In essence, this dissertation presents a well thought FPGA implementation of all the basic arithmetic operations and a successful attempt has been made to save silicon area and reduce overall latency. An implementation of turbine efficiency measurement is presented, illustrating the use of Floating-point arithmetic unit. Simulation and Synthesis results of all sub-components within the FPU and the efficiency measurement are also presented.

ACKNOWLEDGEMENTS

At the outset, I express my deepest sense of gratitude to Dr. H. K. Verma, for giving me the opportunity to work on an exciting project in my area of interest and for his support throughout the dissertation. I remember with great emotion, the constant encouragement and help extended to me by him that went even beyond the realm of academics.

I would like to thank Dr. R. S. Anand, for all the effort and support he has given me throughout my dissertation work. All his technical insight and motivation through my work has been very helpful. His guidance in writing this dissertation was indispensable. I owe him a great debt of gratitude.

My sincere thanks are due to all the faculty members of the department for the voluntary help, direct and indirect, extended to me during the course of the work.

Special thanks to my friends at IIT Roorkee for making my post graduation life a memorable experience.

Finally, I am indebted to my mother who have built my educational foundation, encouraged me throughout my studies and given me the choice and chance to pursue what I desired. Through the course of my studies, I have been fortunate to enjoy unwavering support and encouragement of my little sister, Charu Sharma. My relatives have always been at my side in all the vicissitudes of my M.Tech life. To them I owe a debt of gratitude that can scarcely be contained in this acknowledgement.

LOKESH SHARMA

CONTENTS

<i>CANDIDATE'S DECLARATION</i>	<i>iii</i>
<i>ABSTRACT</i>	<i>iv</i>
<i>ACKNOWLEDGEMENTS</i>	<i>vi</i>
<i>CONTENTS</i>	<i>vii</i>
<i>LIST OF FIGURES</i>	<i>xi</i>
<i>LIST OF TABLES</i>	<i>xiii</i>

CHAPTER 1: INTRODUCTION **1-8**

- 1.1 Motivation
- 1.2 Research Focus
- 1.3 Literature Review
- 1.4 Study Approach
- 1.5 Organization of Report

CHAPTER 2: TURBINE EFFICIENCY MEASUREMENT **9-14**

- 2.1 Turbine Efficiency Measurement: An Application
- 2.2 Elements of Hydroelectric Power Station
- 2.3 Method used for Turbine Efficiency Measurement

CHAPTER 3: IEEE 754: STANDARD FOR BINARY FLOATING-POINT ARITHMETIC **15-20**

- 3.1 Formats
- 3.2 Normalization
- 3.3 Special Values
- 3.4 Exceptions
 - 3.4.1 Invalid Operation
 - 3.4.2 Division by Zero
 - 3.4.3 Overflow
 - 3.4.4 Underflow

- 3.4.5 Inexact
- 3.5 Rounding Modes
 - 3.5.1 Round to Nearest Number
 - 3.5.2 Round to Zero
 - 3.5.3 Round Up
 - 3.5.4 Round Down

CHAPTER 4: HARDWARE MODULES OF FLOATING-POINT ARITHMETIC

UNIT	21-66
-------------	--------------

- | | |
|--|--|
| 4.1 Architecture of Floating-Point Arithmetic Unit | |
| 4.1.1 FPU Interfaces | |
| 4.1.2 Specifications | |
| 4.2 Data Registers | |
| 4.3 Control Register | |
| 4.4 Status Register | |
| 4.5 Tag Register | |
| 4.6 Decode Unit | |
| 4.7 Precision Converter | |
| 4.7.1 Conversion of Single Precision to Extended-Double Precision | |
| 4.7.2 Conversion of Extended-Double Precision to Single Precision | |
| 4.8 Addition/Subtraction Unit | |
| 4.8.1 Use of Tag Word | |
| 4.8.2 Algorithm | |
| 4.8.3 Different Types of Integer Adders and their Comparative Study | |
| 4.9 Multiplication Unit | |
| 4.9.1 Use of Tag Word | |
| 4.9.2 Algorithm | |
| 4.9.3 Different Types of Integer Multipliers and their Comparative Study | |
| 4.10 Division Unit | |
| 4.10.1 Use of Tag Word | |
| 4.10.2 Algorithm | |

4.10.3 Different Types of Integer Dividers and their Comparative Study	
4.10.4 Division Parameters	
4.11 Square Root Unit	
4.11.1 Use of Tag Word	
4.11.2 Algorithm	
4.11.3 Different Types of Integer Square Root Algorithms and their Comparative Study	
4.12 Absolute Unit	
4.13 Exception Generation Unit	
CHAPTER 5: DESIGNING WITH FPGAs	67-76
5.1 Introduction to FPGAs	
5.2 Basic Architectures	
5.3 Programming with FPGAs	
CHAPTER 6: EXPERIMENTAL RESULTS AND VERIFICATION	77-102
6.1 Introduction to Experimental Approaches	
6.1.1 Design Environment	
6.1.2 FPGA Design Flow	
6.2 Simulation Results	
6.2.1 Data Registers	
6.2.2 Control Register	
6.2.3 Tag Register	
6.2.4 Precision Converter	
6.2.5 Addition/Subtraction Unit	
6.2.6 Multiplication Unit	
6.2.7 Division Unit	
6.2.8 Square Root Unit	
6.2.9 Absolute Unit	
6.2.10 Exception Generation Unit	
6.2.11 Turbine Efficiency Measurement	

6.3 Verification of Simulation Results	
CHAPTER 7: CONCLUSION AND FUTURE WORK	103-104
7.1 Conclusion	
7.2 Suggestions for Future Work	
<i>REFERENCES</i>	105-110
<i>APPENDIX A Glossary</i>	111-124
<i>APPENDIX B Design Customized Instruction and their usage</i>	125-130
<i>APPENDIX C Synthesis Report</i>	131-133

LIST OF FIGURES

Name	Page
Figure 1.1 Study Approach	6
Figure 2.1 Elements of Hydroelectric Power Station	10
Figure 2.2 Efficiency Measurement	13
Figure 4.1 (a) Architecture of Floating-point arithmetic unit	21
Figure 4.1 (b) Execution unit	22
Figure 4.2 Data register stack	25
Figure 4.3 Control register	26
Figure 4.4 Status register	27
Figure 4.5 Tag Register	28
Figure 4.6 Addition/Subtraction algorithm	33
Figure 4.7 Ripple Carry Adder	35
Figure 4.8 Carry Look Ahead Adder	38
Figure 4.9 Carry Select Adders	39
Figure 4.10 Multiplication Algorithm	42
Figure 4.11 (a) Basic Cell	46
Figure 4.11 (b) Shift-Add Multiply	46
Figure 4.12 Architecture of the Booth multiplier	47
Figure 4.13 Partial Product Generation	49
Figure 4.14 Wallace Tree	50
Figure 4.15 Division Algorithm	52
Figure 4.16 Square root unit Algorithm	60
Figure 4.17 Absolute unit Algorithm	65
Figure 5.1 FPGA Architecture	67
Figure 5.2 Simple Logic Block Structure	69
Figure 5.3 The Four FPGA Architectural Classes	71

Figure 5.4 Typical CAD system design flow for FPGAs	72
Figure 5.5 Designing with FPGA	74
Figure 6.1 FPGA Design Flow	78
Figure 6.1 Simualtion Waveform for Stack operation	80
Figure 6.3 Simulation Waveform for Conrol word	81
Figure 6.4 Simulation Waveform for Tag word	82
Figure 6.5 Simulation Waveform of Single Precision to Extended-Double Precision	82
Figure 6.6 Simulation Waveform of Extended-Double Precision to Single Precision	83
Figure 6.7 Simulation Waveform of Addition	83
Figure 6.8 Simulation Waveform for Multiplication	85
Figure 6.9 Simulation Waveform of Division	86
Figure 6.10 Simulation Waveform of Square Root	87
Figure 6.11 Simulation Waveform for Absolute Unit	88
Figure 6.12 Simulation Waveform for Exception Generation	89-90
Figure 6.13 Simulation Waveform of Unit Efficiency Measurement	93-98
Figure 6.13 Simulation Waveform of Unit Efficiency Measurement	100-101

LIST OF TABLES

Name	Page
Table 3.1 Summary of Format parameters	17
Table 3.2 (a) Special values	18
Table 3.2 (b) Example: Special values	18
Table 4.1 FPU interface	23
Table 4.2 Booth Multiplication	48
Table 6.1 Example of Test Vectors	101
Table C.1 Device utilization for FPU	131
Table C.2 Device utilization for Stack Register	132
Table C.3 Device utilization for Store (part of stack)	132
Table C.4 Device utilization for Load (part of stack)	132
Table C.5 Device utilization for Decoder Unit	132
Table C.6 Device utilization for Addition Unit	132
Table C.7 Device utilization for Multiplication Unit	133
Table C.8 Device utilization for Division Unit	133
Table C.9 Device utilization for Absolute Unit	133

CHAPTER 1***INTRODUCTION***

A floating-point number is a digital representation for a number in a certain subset of the rational numbers, and is often used to approximate an arbitrary real number on a computer. The term floating point is derived from the fact that there is no fixed number of digits before and after the decimal point; that is, the decimal point can float. There are also representations in which the number of digits before and after the decimal point is set, called fixed-point representations. In general, floating-point representations are slower and less accurate than fixed-point representations, but they can handle a larger range of numbers.

The Floating Point Arithmetic Unit (FPU), designed in this dissertation, is a specialized computation unit that manipulates numbers more quickly than the basic microprocessor circuitry. The FPU does this by means of instructions that focus entirely on large mathematical operations.

System-on-a-chip (SoC) is a new insight of integrating all components of a computer system into a single chip [21]. This chip may contain digital, analog, mixed-signal all on the same die. These chips are rapidly replacing more sophisticated computer systems in many applications [41], especially when the silicon space available is a concern. The presented FPU offers programming flexibility to the end user.

Digital systems are either conventional hard-core systems (application specific integrated circuits – ASICs) which have limited hardware programming (customization) capability, or soft-core systems (Field Programmable Gate Array – FPGA based systems) which are fully programmable and customizable.

Each of these systems has their own pros and cons discussed as below,

- **Hard-core Systems (ASICs):**

An ASIC is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use. These systems have portability problems, and using a full fledged microprocessor for every task is not feasible. Thus, hard-core systems are

generally ordered by the customer to the manufacturer to do a specific task. Also, the development time and NRE cost for ASICs is very high.

- **Soft-core Systems (FPGA based systems):**

FPGA based processors are fully programmable (customizable) systems. Any general FPGA based system can be programmed in two levels: Low level and High level using HDL [19][20]. FPGAs have almost zero NRE cost and available in the market all the time. The FPGA core can be programmed to be any digital system one can think of, from a simple logic AND gate to a full fledged microprocessor. The same FPGA core can be reprogrammed (re-customized) later, to serve other purposes.

Thus, because of the “hardware customization” concept that is introduced by the FPGA based systems, two entirely different systems can be constructed using the same chip with different HDL files. This is a relatively new technology, and limited numbers of soft-core FPGAs exist in the market.

In today’s processing-power hungry applications, the extended dynamic range and precision offered by floating-point arithmetic is quickly becoming a requirement in numerous signal processing algorithms that are being used in graphics, advanced wireless communications, instrumentation, industrial control, audio and medical imaging applications. This growing use of floating-point arithmetic places a requirement for area efficient and high performance solutions on hardware engineers. One such application is turbine efficiency computation. To calculate unit efficiency, one needs to process large number of variables acquired from different locations of hydro power stations. An independent Floating-point arithmetic unit is designed that is supported by eight general purpose registers, tag register, control register, and status register. These supporting registers make FPU to work in stand-alone mode.

1.1 Motivation

For the most part, the digital design companies have resolved to FPGA design instead of ASICs due to its effective time to market, adaptability and most importantly,

its low cost. Floating-point arithmetic unit is one of the most important custom applications needed in most hardware designs as it adds accuracy and ease of use. A lot of work has been done on floating-point operations and FPGAs that is summarized in section 1.3. However, to the best of my knowledge, there is no work which gives implementation of Floating-point arithmetic unit in FPGA with a facility to program it and a detailed analysis of architectural implementation of sub operations for floating-point arithmetic unit in FPGA. Since the area occupied by floating-point unit in FPGA is well known to be very large, I investigated several approaches/algorithms to reduce the area and execution time of Floating-point arithmetic unit. A successful implementation of turbine efficiency measurement is presented, illustrating the use of floating-point unit.

1.2 Research Focus

The main contribution and objective of our work is to implement and analyze algorithms for floating-point operations and hardware modules used to compute these algorithms. These algorithms and modules are implemented using Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL), and then are synthesized using Xilinx ISE 7.1 platform supported by Modelsim and Aldec Active HDL simulation environment [15][1]. These implementations are placed and routed in the FPGA device. Area and timing information for each design approach and algorithm is analyzed.

1.3 Literature Review

One of the earliest investigations into using FPGAs to implement floating-point arithmetic was done by Fagin et al. [4] who in 1994 showed that implementing IEEE single precision operators was possible, but also impracticable on then current FPGA technology. The circuits designed by the author were an adder and a multiplier and both had full implementation of all four rounding modes specified by IEEE 754 standard. Area was the critical constraint, with the authors reporting that no device in existence could contain a single precision multiplier circuit. Therefore,

the authors purpose adopting smaller, custom formats which may be more appropriate to FPGA architectures than the full IEEE formats.

This line of thought was expanded on by the significant work of Shirazi et al. [5] who suggested application specific formats in width of 16(1-6-9) and 18(1-7-10) bits, as opposed to full 32(1-8-23) bits in the IEEE 754 standard. Modules for addition/subtraction, multiplication, and division were presented, though no work was done on implementing rounding or error-handling.

Another significant work came from Louca et al. [6] in which the authors, building on the work of Shirazi and others, abstract the normalization operation away from the actual arithmetic operators, in an effort to conserve area. No rounding capability was implemented by the authors, due to area constraints.

Ligon et al. [7] presented IEEE single precision adder and multiplier circuits on the then newly available Xilinx 4000 series FPGAs. Both circuits supported rounding to nearest, but didn't used a separate normalizing unit. Similar work by Stamoulis et al. [8] presented IEEE single precision adder/subtractor, multiplier and division circuits. However, the authors don't present any rounding capability and normalizing unit.

Work by Sahin et al [9] present adder/subtractor, multiplier and accumulator circuits, but again only in IEEE single precision format. Also rounding capability is not implemented. Dido et al. [10] discusses flexible floating point formats which were different from IEEE 754 standard, but they successfully implemented the hardware modules without support for rounding. Their format contains no sign bit or bias of exponent.

Work by Y. Li et al. [22] present single precision square root algorithm and its VLSI implementation. Although the design was targeted for CMOS technology, it gives good implementation details of Non-Restoring method. In 2003, Xiaojun Wang et al. [23] provided the tradeoffs of implementing division and square root on Virtex FPGAs.

Ling Zhuo et al. [29] presented FPGA based area reduction circuits. Using Xilinx's Virtex II pro as the targeted device, Ling Zhuo and others implemented floating point adder circuits.

One of the most recent works published related to my work is published by G. Govindu, L. Zhuo, S. Choi, and V. Prasanna [11] on the analysis of high-performance floating-point arithmetic on FPGAs. This paper has been an excellent resource for our implementation and discussions throughout the research process, and provides possible explanations. All the implementations are done with the latest Xilinx Virtex 2p FPGA. Another recent work by A. Malik et al. [12] discusses an effective implementation of floating-point adder using the pipelined version of Leading One Predictor (LOP).

Work by Prof. H. K. Verma et al. [2] shows that efficiency test on turbine-generator unit in a hydro power station needs simultaneous measurement of a number of variables located in different places in the station using suitable instruments placed close to respective variables. The measurement data from these instruments can be acquired simultaneously by connecting them in a network using RS-485 serial data standard. I have designed the FPGA based Floating point arithmetic unit capable to further process the accumulated data. The Floating-Point Arithmetic unit in my work is the generalized superset of all these works. It not only supports the IEEE 754 format while implementing all arithmetic operations viz. addition/subtraction, multiplication, division, square root, and absolute value of a number but also provides the programming flexibility to the end user. Also, I abstract normalization as well as rounding functionality with a choice of all four rounding modes. My Floating-Point unit also provides Status register, Control register, eight General Purpose registers and a lot more. All this will be discussed in subsequent chapters. In essence, the features are taken from Intel Pentium4 [14].

1.4 Study Approach

The approach towards this dissertation is to study, implement and analyze different existing algorithms and selects the one which gives the best performance in terms of area and latency.

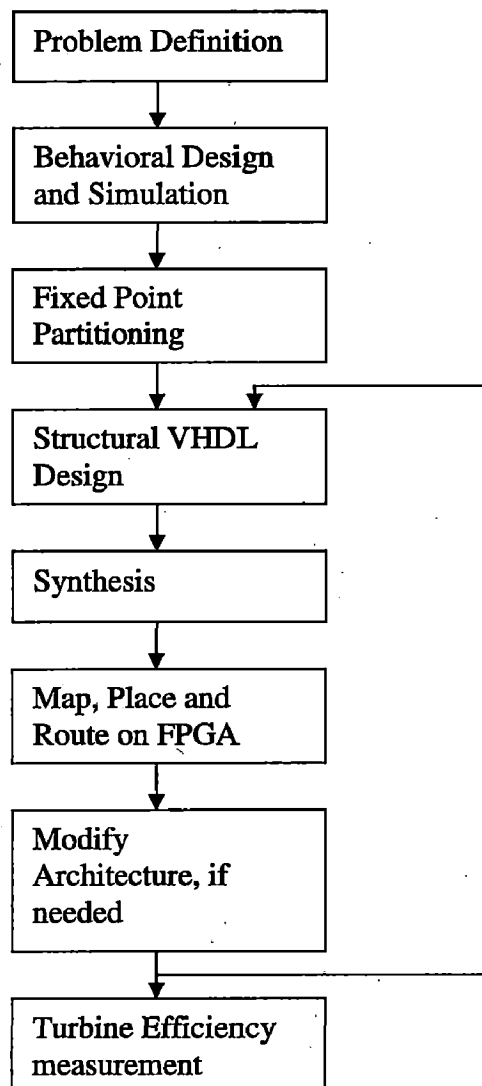


Figure 1.1 Study Approach

To facilitate the design process, a solid problem definition was defined.

Problem Definition

- ✓ Main aim of this dissertation is to explore some new application areas of FPGA to develop independent System on Programmable Chip (SOPC).

- ✓ To develop a FPGA based Floating-Point Arithmetic unit capable of performing all arithmetic operations such as addition/subtraction, division, multiplication, square root and absolute value of a number. The unit should provide programming functionality to the end user.
- ✓ To test the system for Turbine Efficiency measurement.

The Behavioral VHDL modules undergo simulation using Active HDL Tools to provide the level of correctness before the synthesize stage. Synthesize is done using Xilinx ISE 7.1 and the design is mapped, placed and routed on the Xilinx FPGA board. Timing reports are also generated and changes made to the architecture are back annotated to the structural design and synthesized to be placed and routed. Figure 1.1 depicts the methods to be used in this research.

1.5 Organization of Report

Chapter 2 (Turbine Efficiency measurement) discusses the method used for turbine efficiency measurement. Basic elements of hydro power station have also been discussed in this chapter.

Chapter 3 (IEEE 754: Standard for Binary Floating-Point Arithmetic) presents the introduction to IEEE 754 standard for binary floating-point arithmetic. It gives the details of basic and extended floating-point number, exception generation and their handling, normalization and rounding units.

Chapter 4 (Hardware Modules of Floating-Point Arithmetic unit) presents the architecture and specifications of Floating-point arithmetic unit. Internal hardware modules of FPU including their functions and structures are described in detail. The simulation and synthesis results of these modules are presented in Chapter 6 and Appendix C respectively.

Chapter 5 (*Designing with FPGAs*) gives overview of FPGA which is followed by design flow of FPGAs. The chapter also presents the detailed architecture of Xilinx' Virtex II Pro FPGA kit (targeted device in this dissertation).

Chapter 6 (*Experimental Results and Verification*) presents the experimental results of all subunits and test bench written for turbine efficiency measurement. Software and Hardware Environments used during the various phases of dissertation are also presented.

Chapter 7 (*Conclusion and Future Work*) concludes and suggests the future work which can be done in this area.

CHAPTER 2***TURBINE EFFICIENCY MEASUREMENT***

2.1 Turbine efficiency measurement: An application

Efficiency test on turbine-generator unit in a hydro power station needs simultaneous measurement of a number of variables located in different places in the station using suitable instruments placed close to respective variables [2]. The measurement data from these instruments was acquired simultaneously by connecting them in a network using RS-485 serial data standard and was finally collected in a computer for further processing. Now devoting the whole computer or Laptop for processing of this data is a costly affair. So I believe that FPGA is best suited for such an application as it is reconfigurable, allows parallel processing, cost effective, and offers many other advantages. In this thesis, I have developed a FPGA based Floating-point arithmetic unit for turbine efficiency measurement. This is a system on chip design which accepts the real numbers at its input ports, does calculation as programmed by the end user, and provides the output at its output port. A successful attempt is made to save silicon space and reduce latency. I believe this project will help in saving thousands of bucks while providing efficient results.

I have taken it for granted that these variables are already measured and the measurement data from different instruments can be acquired easily. This chapter discusses the method employed for the computation of turbine efficiency. Before going into the depth of the matter, I would like to discuss elements of hydroelectric system.

2.2 Elements of Hydroelectric Power Station

Hydropower plants harness water energy and use simple mechanics to convert water energy into electric energy. Hydroelectric systems are actually based on a rather simple concept – water flowing through a dam turns a turbine, which turns a generator. The basic components of a conventional hydroelectric system are shown in Figure 2.1.

- **Dam:** Most hydropower plants rely on a dam that holds back water, creating a large reservoir.
- **Intake:** Intake is the highest point of hydroelectric system where the gravity pulls the water through the penstock, a pipeline that feeds the turbine, when the gates on the dam open. Water builds up pressure as it flows through the pipe.

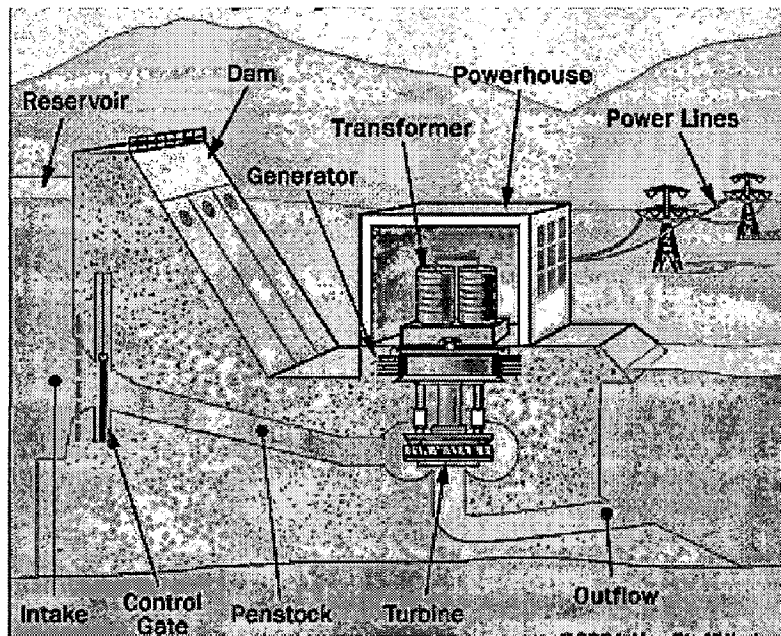


Figure 2.1 Elements of Hydroelectric Power Station [50]

- **Penstock:** Penstock, the pipeline not only moves the water to the turbine, but is also the enclosure that creates head pressure as the vertical drop increases.
- **Turbine:** The water strikes and turns the large blades of a turbine, which is attached to generator above it by way of a shaft. In essence, the turbine is the heart of hydroelectric system, where the water power is converted into rotational force that drives generator. For maximum efficiency, turbine should be designed to match head and flow of the hydroelectric system. Turbines can be divided into two major types:

Reaction Turbines use runners that operate fully immersed in water, e.g., Francis, Propeller, and Kaplan etc.

Impulse Turbines use runners that operate without being immersed in water, e.g., Pelton, Turgo etc.

- **Generators:** As the turbine blades turn, so do a series of magnets inside the generator. Giant magnet rotate the copper coils, producing alternating current (AC) by moving electrons.
- **Drive system:** Drive system couples the turbine to the generator, which converts the rotational energy from the turbine into electricity.
- **Powerhouse:** Powerhouse is simply a building that houses turbine, generator and other necessary system components.
- **Transformer:** The transformer inside the power house takes the AC and converts it to higher voltage current.
- **Power Lines:** Out of every power plant come four wires: three phases of power being produced simultaneously plus a neutral or ground common to all three.
- **Outflow:** Used water is carried through pipelines, called **tailraces**, and reenters the river downstream.

2.3 Method used for Turbine Efficiency measurement

Determination of turbine efficiency requires measurement of hydraulic power input to the turbine and electric power output from the turbine-generator unit, and calculation of the ratio of two quantities.

The hydraulic power input to the turbine, P_i is given by

$$P_i = \rho g H Q$$

where, g is the acceleration due to gravity, m/s^2 ,

ρ is the density of water, kg/m^3 ,

$H = H_1 - H_2$ is the net water head, m,

H_1 is water head at inlet, m,

H_2 is water head at outlet, m,

Q is the water discharge through the turbine, m^3/s .

The international standard value of g is 9.806 m/s^2 . However, its actual value at a given location is a function of the latitude and altitude of the location. Values of g are given in IEC-60041 [3] that shall be used for achieving higher accuracy.

The density of water is approximately 1000 kg/m^3 and it is a function of temperature and pressure. Values for density of water, ρ are given in IEC-60041 [3].

There are number of methods available to find the net water head, H and discharge, Q . Details of these methods are available in IEC-60041 manual [3].

The electrical output from the turbine-generator unit can be measured by using a wattmeter. Let the output from turbine-generator unit be represented by P_e .

The following figure 2.2 gives an idea of the method used to measure turbine efficiency.

Unit efficiency is given by

$$\text{Unit Efficiency, } \eta_U = \frac{\text{Generator Output, } P_e}{\text{Hydraulic Input, } P_i} \times 100\%$$

Knowing the value of the generator efficiency, η_g , the turbine efficiency can be calculated by the following relation:

$$\text{Turbine Efficiency, } \eta_T = \frac{\text{Unit Efficiency, } \eta_U}{\text{Generator Efficiency, } \eta_g} \times 100\%$$

So efficiency measurement based on the above concepts requires the measurement of following parameters:

- 1 Water density, ρ
- 2 Gravity, g
- 3 Water head at inlet, H_1
- 4 Water head at outlet, H_2
- 5 Discharge, Q
- 6 Generator output, P_e .

To learn about acquiring the measurement data simultaneously, please refer to [2].

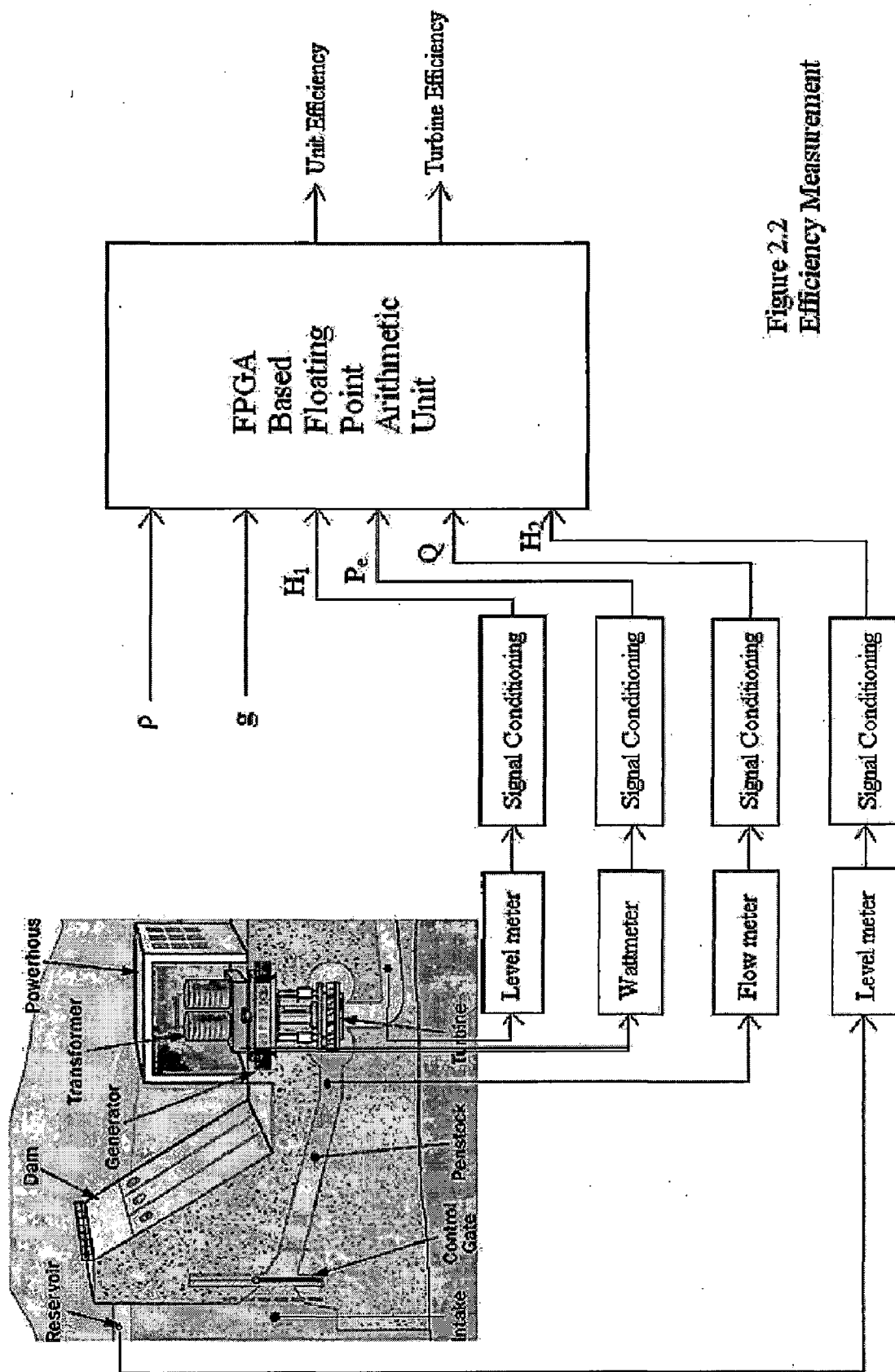


Figure 2.2
Efficiency Measurement

CHAPTER 3

IEEE 754:

STANDARD FOR BINARY FLOATING-POINT ARITHMETIC

Floating Point is a representation of real (fractional) numbers. In this representation, the location of the fractional point can be moved from one position to another according to the precision. In the early days of computers, vendors start developing their own floating-point representations and methods of calculations. These different approaches lead to different results in calculations. So the IEEE organization defined in the IEEE-754 standard a representation of the floating point numbers and the operations [13]. The standard specifies:

- Basic and extended floating-point number formats
- Add, subtract, multiply, divide, square root, remainder, and compare operations
- Conversions between integer and floating-point formats
- Conversions between different floating-point formats
- Conversions between basic format floating-point numbers and decimal strings
- Floating-point exceptions and their handling, including non numbers (NaNs)

3.1 Formats

IEEE Floating-point representation divides the number of bits into three groups:

- **Sign bit:** The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.
- **The Biased-Exponent part:** The exponent is that component of the binary floating-point number that normally signifies the integer power to which two is raised in determining the value of the represented number. And the biased-

exponent is the sum of the exponent and a constant (bias) chosen to make the biased-exponent range non-negative.

$$\textit{Biased-Exponent, } e = E + \textit{bias}$$

The range of the unbiased exponent E shall include every integer between two values E_{min} and E_{max} , inclusive and also two other reserved values $E_{min}-1$ to encode ± 0 and denormalized numbers, $E_{max}+1$ to encode $\pm\infty$ and NaNs. The foregoing parameters are given in Table 3.1.

- **The Fractional part** (also known as **mantissa**): The field of the significant that lies to right of its implied binary point.

$$\textit{Fraction, } f = .b_1b_2\dots b_{p-1}$$

The numbers are of the form $(-1)^s 2^E (b_0.b_1b_2\dots b_{p-1})$

where, $s = 0$ or 1 ;

$E =$ any integer between E_{min} and E_{max} ;

$b_p = 0$ or 1 ;

The IEEE 754 standard defines five floating-point formats in two groups, basic and extended [13]. The basic format is further divided into single-precision with 32-bits wide, double-precision with 64-bits wide and quad-precision with 128-bits wide. Then there is single-extended precision format and double-extended precision format. Extended format is implementation dependent and does concerns my project. The inputs to Floating-point unit are single-precision real numbers but internally, all the calculations are carried out in double-extended format to enhance the accuracy of result.

Table 3.1 Summary of Format parameters

Parameter	Single	Single Extended	Double	Double Extended	Quad Precision
Total bits	32	≥ 43	64	≥ 79	128
Precision bits, p	24	≥ 32	53	≥ 64	113
Sign bits, s	1	1	1	1	1
Fraction bits, f	23	≥ 32	52	≥ 64	112
Exponent bits, e	8	≥ 11	11	≥ 15	15
E _{max}	+127	$\geq +1023$	+1023	$\geq +16383$	+16383
E _{min}	-126	≤ -1022	-1022	≤ -16382	-16382
Exponent <i>bias</i>	+127	Unspecified	+1023	Unspecified	+16383

3.2 Normalization

Normalization is the act of shifting the fractional part in order to make the most significant bit of the fractional part one. During this shifting, the exponent is incremented. In essence normalized numbers have their MSB 1 in the most left bit of the fractional part and denormalized numbers are just the opposite of normalized numbers.

Some operations like addition/subtraction require that the exponent field should be same for all operands. In such a case, one of the operand should be denormalized. I have done denormalization of the smaller operand.

Denormalized numbers have important use in some operations and numbers. For example, assume minimum exponent of some format is -88, and the number of digits is 3. It is required to perform $x - y$ where $x = 5.87 \times 10^{-87}$ and $y = 5.81 \times 10^{-87}$. The result of this operation is 0.06×10^{-89} which is too small to be represented as a normalized number. If we try to represent it as normalized number, the result becomes zero which is incorrect but if it is denormalized we will get the correct result.

3.3 Special values

The IEEE 754 standard supports some special values viz positive zero, negative zero, positive infinity, negative infinity and Not a Number (NaN) as given in Table 3.2.

Table 3.2 (a) Special values

Name	Exponent	Fraction	Sign	Exp bits	Fract bits
+0	Min - 1	= 0	+	All zeros	All zeros
-0	Min - 1	= 0	-	All zeros	All zeros
Number	Min ≤ e ≤ Max	Any	Any	Any	Any
+∞	Max + 1	= 0	+	All ones	All zeros
-∞	Max + 1	= 0	-	All ones	All zeros
NaN	Max + 1	≠ 0	Any	All ones	Any

Below is a table with the corresponding values for a given representation (single precision, in this case) to help better understand the above table.

Table 3.2 (b) Example: Special values

Sign	Exponent	Fraction	Value
0	00000000	000000000000000000000000	+0
1	00000000	000000000000000000000000	-0
1	00000000	100000000000000000000000	$-2^{0-127} \times 0.(2^{-1})$
0	00000000	000000000000000000000001	$+2^{0-127} \times 0.(2^{-23})$
0	10000000	000000000000000000000000	+∞
1	10000000	000000000000000000000000	-∞
0	10000000	100000000000000000000000	NaN

3.4 Exceptions

There are five types of exceptions that shall be signaled when detected. For each type of exception the implementation will provide a bit in the status register that will be set on any occurrence of the corresponding exception. The presented implementation of Floating-point arithmetic unit will provide the end user a way to read and write the status register.

3.4.1 Invalid Operation

The invalid operation exception is signaled if an operand is invalid for the operation on to be performed. The invalid operations are:

- Any operation on a *NaN*
- Addition or subtraction: $\infty + (-\infty)$
- Multiplication: $\pm 0 \times \pm \infty$
- Division: $\pm 0 / \pm 0$ or $\pm \infty / \pm \infty$
- Square root: if the operand is less than zero

3.4.2 Division by zero

If the divisor is zero and the dividend is a finite nonzero number, then the division by zero exception shall be signaled. The result will be correctly signed ∞ .

3.4.3 Overflow

The overflow exception is signaled whenever the result exceeds the maximum value that can be represented due to the restricted exponent range. It is not signaled when one of the operands is infinity, because infinity arithmetic is always exact. Division by zero also doesn't trigger this exception.

3.4.4 Underflow

Two events cause the underflow exception to be signaled, tininess and loss of accuracy. Tininess is detected after or before rounding when a result lies between $\pm 2^{E_{min}}$. Loss of accuracy is detected when the result is simply inexact or only when a denormalization loss occurs. The implemented FPU core signals an underflow exception

whenever tininess or denormalization loss is detected after rounding and at the same time the result is inexact.

3.4.5 Inexact

This exception will be signaled whenever the result of an arithmetic operation is not exact due to the restricted exponent and/or precision range.

3.5 Rounding Modes

Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit in the destination's format. To increase the precision of the result and to make best use of rounding modes, all the internal calculations are carried out in double-extended format. The IEEE 754 standard specifies four rounding modes and my FPU supports all these rounding modes:

3.5.1 Round to nearest number

This is the standard default rounding. The value is rounded up or down to the nearest infinitely precise result. If the value is exactly halfway between two infinitely precise results, then it should be rounded up to the nearest infinitely precise number. e.g., 3.4 will be rounded to 3.0 and 3.5 to 4.0.

3.5.2 Round-to-Zero

Basically in this mode the number will not be rounded. The excess bits will simply get truncated, e.g. 3.47 will be truncated to 3.4.

3.5.3 Round-Up

The number will be rounded up towards $+\infty$, e.g. 3.2 will be rounded to 4.0, while -3.2 to -3.

3.5.4 Round-Down

The opposite of round-up, the number will be rounded up towards $-\infty$, e.g. 3.2 will be rounded to 3.0, while -3.2 to -4.

CHAPTER 4

HARDWARE MODULES OF FLOATING-POINT ARITHMETIC UNIT

This chapter presents the architecture and specifications of Floating-point arithmetic unit. Internal hardware modules of FPU including their functions and structures are described in detail.

4.1 Architecture of Floating-point Arithmetic unit

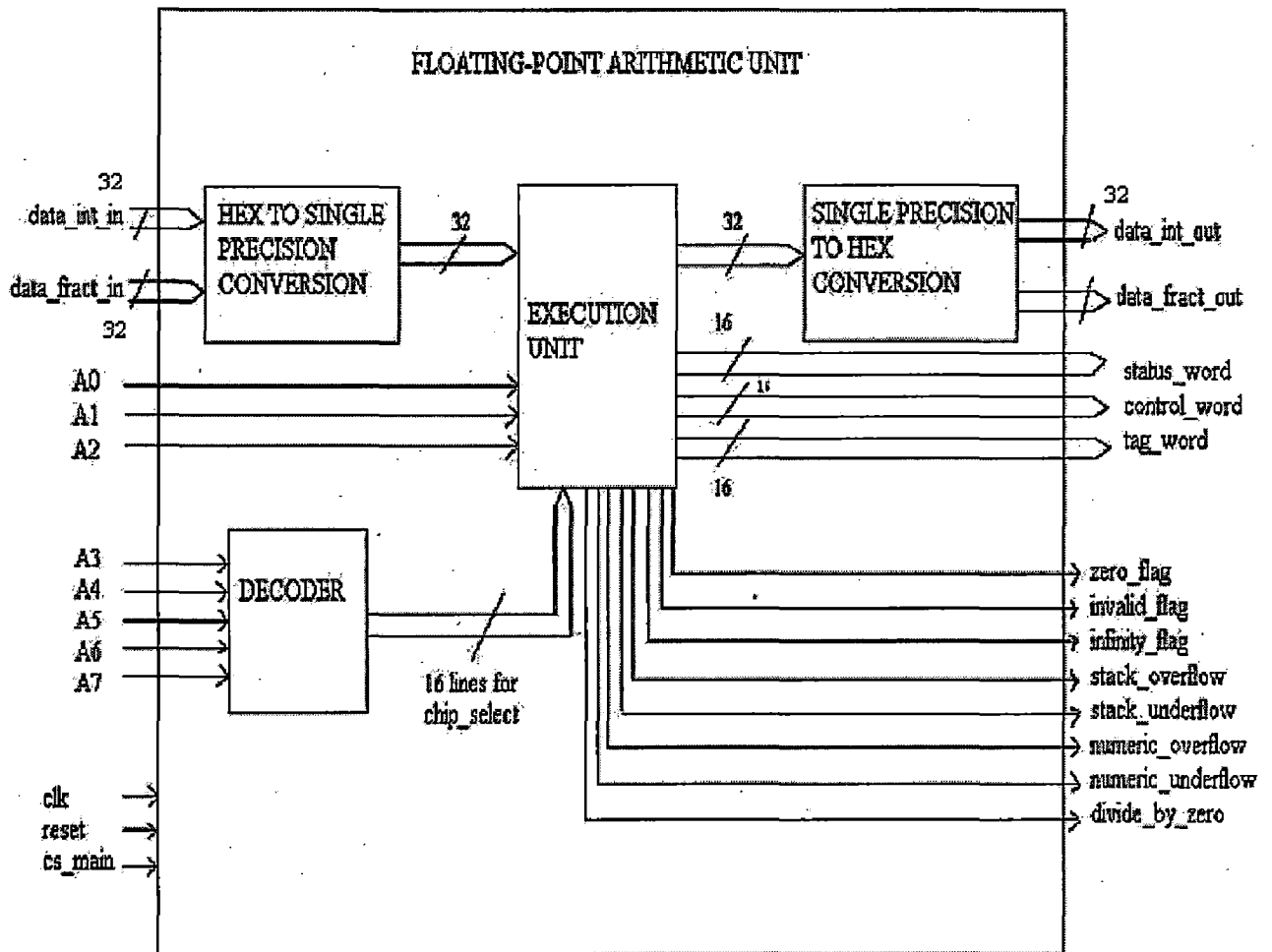


Figure 4.1 (a) Architecture of Floating-point arithmetic unit

Inside of execution unit is as shown below in Figure 4.1 (b).

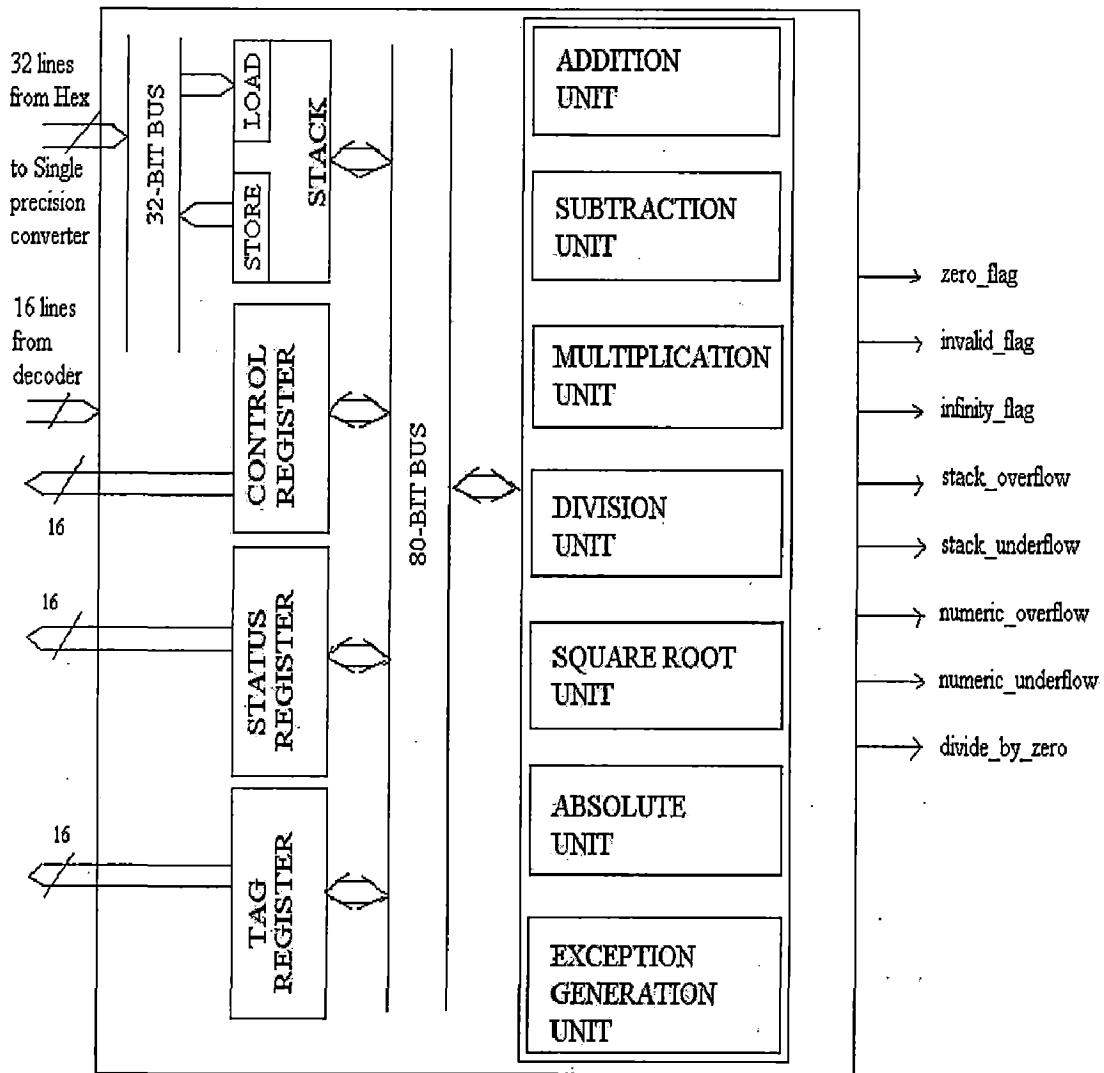


Figure 4.1 (b) Execution unit

4.1.1 FPU interfaces

The following table gives the description of the interfaces of floating-point arithmetic unit.

Table 4.1 FPU interface

Name	Direction	Size	Description
data_int_in	Input	[31:0]	Integer part of real number given to FPU. Its MSB is taken to find the sign of this number.
data_fract_in	Input	[31:0]	Fractional part of real number given to FPU.
A	Input	[7:0]	Address bus predicts which operation is to be performed and on which register.
clk	Input	1	Clock (+ve edge trigger)
reset	Input	1	Reset all data to zero.
data_int_out	Output	[31:0]	Integer part of real number given to FPU. Its MSB is taken to find the sign of this number
data_fract_out	Output	[31:0]	Integer part of real number given to FPU. Its MSB is taken to find the sign of this number
Status_word	Output	[15:0]	Gives the status of FPU
Control_word	Output	[15:0]	Controls the rounding method used and masks the exceptions.
Tag_word	Output	[15:0]	Gives information about validity of data in register stack.
Zero_flag	Output	1	Tells whether the data is zero
invalid_flag	Output	1	Tells whether the data is invalid
infinity_flag	Output	1	Tells whether the data is invalid
stack_overflow	Output	1	Tells if the stack is full
stack_underflow	Output	1	Tells if the stack is empty
numeric_overflow	Output	1	Tells whether result of some operation has crossed the maximum limit
numeric_underflow	Output	1	Tells whether result of some operation has gone below the minimum limit
divide_by_zero	Output	1	Tells whether division by zero is attempted

4.1.2 Specifications

Following are the system specifications of Floating-Point Arithmetic unit which were keeping in mind while designing:

- The core should be complaint with the IEEE-754 standard.
- Although the core is an execution unit, it should work in the stand alone mode.
- The FPU works has storage registers and the result data or intermediate results can be stored in that. No need for CPU (works in stand alone mode)
- The end user will issue instructions to FPU.
- The core should provide the status of FPU.
- The core should provide the functionality to mask the exceptions as described in IEEE 754 standard and control the rounding modes.
- One instruction is executed at a time.
- The user should have ability to read all interfaces.

4.2 Data Registers

Data registers incorporated in Floating-point unit is similar to that of x87 FPU data registers in Intel processor. It consists of eight 80-bit registers. Values are stored in these registers in the double-extended precision floating-point format (IEEE 754 standard) [13].

On execution of Load instruction, the single precision number from memory or external world is loaded into data register. Here, the value is automatically converted into double-extended precision format. Similarly, on execution of Store instruction, the double-extended precision number (content of data register) is converted into single precision format and transferred back into memory or external world.

The FPU instructions treat the eight data register as a register stack. All addressing of the data registers is relative to the register on the top of the stack. The register number of the current top-of-stack is stored in the TOP (stack TOP) field in the FPU status word. Load operations decrement TOP by one and load a value into the new top-of-stack register, and store operations store the value from the current TOP register in memory (or external world) and then increment TOP by one.

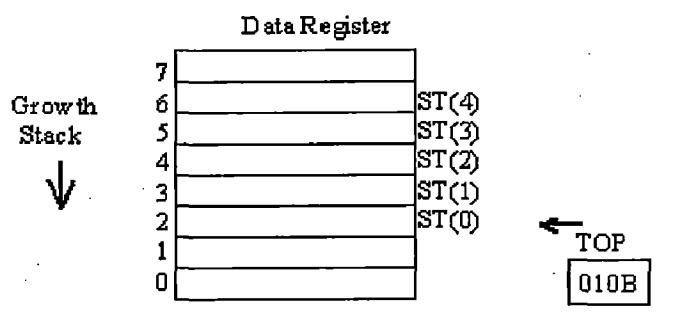


Figure 4.2 Data register stack

If a load operation is performed when TOP is at bottom of the stack (i.e. at 0), register wraparound occurs and the new value of TOP is set to 7. The floating-point stack-overflow exception (see *exception generation unit 4.13*) indicates when wraparound might cause an unsaved value to be overwritten. Internally, assembler supports register addressing mode to operate on the top of the stack, using the expression R0 to represent the current stack top and R_i to specify the i th register from the TOP in the stack ($0 \leq i \leq 7$). For example, if TOP contains 010B (assume, register 2 in the top of the stack), the following instruction would multiply the contents of two registers in the stack (register 2 and 6):

FMUL ST(4);

4.3 Control Register

The 16 bit FPU control word (see Figure 4.3) controls the rounding method used. It also contains the exception mask bits. The contents of this register can be loaded with the load control word instruction.

When the FPU is initialized with the either an initialization instruction or upon reset the control word is set to 0C00H which unmask all floating-point exceptions and sets rounding to nearest.

The exception flag mask bits (bit 0 through 4 of the control word) mask the five floating point exception in the FPU status word. When one of these mask bits is set, its corresponding exception is blocked from being generated.

The bits 10 and 11 of the control register controls how the results of FPU are rounded. Rounding control is designed as per IEEE 754 standard [13].

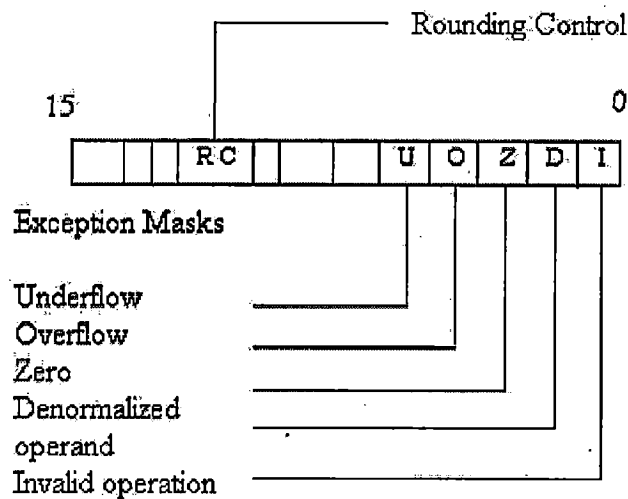


Figure 4.3 Control register

4.4 Status Register

The 16-bit FPU status register (see Figure 4.4) indicates the current state of the FPU. The flags in the status register include the busy flag, top-of-stack (TOP) pointer, condition code flags, stack fault flag, and exception flags. The FPU sets the flags in this register to show the results of operations. The contents of the status register can be stored in memory or external world using the FRSW instruction.

- **Busy Flag:** Indicates FPU is busy i.e. executing an instruction.
- **Top of Stack (TOP) Pointer:** A pointer to the FPU data register that is currently at the top of the register stack is contained in bits 11 through 13 of the FPU status word. This pointer, which is commonly referred to as TOP (for top-of-stack), is a binary value from 0 to 7.
- **Condition Code Flags:** It consist of zero flag, infinity flag, and sign flag for indicating different conditions after executing asked operation.
- **FPU Exception Flags:** The four exception flags (bits 0 and 2 through 4) of the FPU status word indicate that one or more have been detected since the bits were last cleared. The individual exception flags (IE, ZE, OE and UE) are described in

detail in Section 4.13. Each of the exception flags can be masked by an exception mask bit in the FPU control word.

- **Stack Fault:** The stack fault flags (bit 6 and 7 of the status register) indicate that stack overflow and stack underflow has occurred with data in the data register stack.

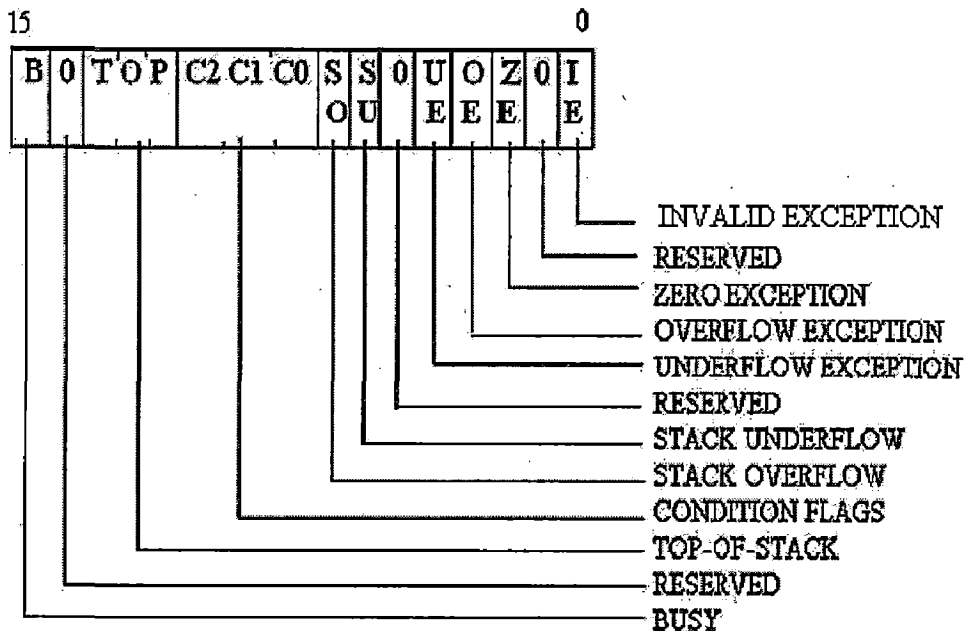


Figure 4.4 Status register

4.5 Tag Register

The 16-bit Tag register gives the information about validity of contents of FPU data register stack (one 2-bit tag per register). The tag indicates whether a register contains a valid number, zero or a special floating number (NaN, infinity), or whether it is empty [14]. On reset, FPU tag word is set to FFFFH, which marks all the FPP data register as empty. Each tag in the tag word corresponds to a physical register (number 0 to 7).

Instruction FADD, FSUB, FMUL, FDIV, FSQRT, FABS use this tag information to check the content of data register before performing their operations, this assist FPU to prevent from performing complex operation and allows save clock cycles.

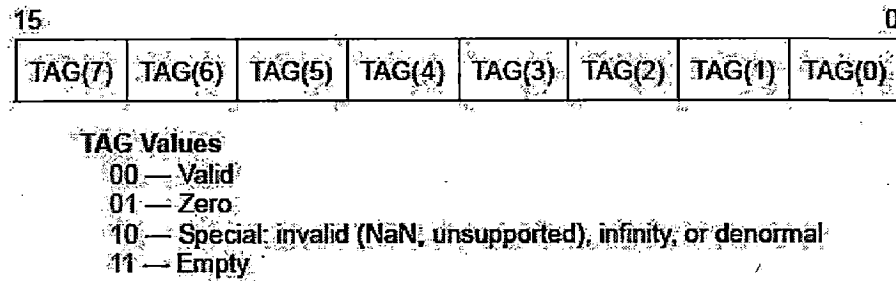


Figure 4.5 Tag Register

Tag in the tag word changes only when the write operation is performed on the FPU data registers. Tag word marks empty the appropriate tag (top of stack) after read operation. End user cannot directly load or modify the tags in the tag register.

4.6 Decode unit

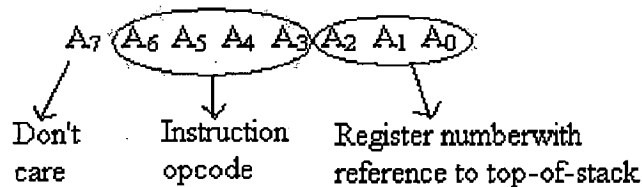


Figure 4.6 address lines to decoder

The address bus consists of eight address lines. These address lines are distributed as shown below in Figure 4.6.

The instruction decoder logic is 4:16 lines i.e. input to the decoder is 4 address lines (A₆A₅A₄A₃) and output from the decoder is 16 1-bit signal. Each of the lines either activate or deactivate chip select signal. At a time, only one chip is activated. The remaining address lines (A₂A₁A₀) identifies the register number with reference to top-of-stack for arithmetic operations.

For example,
FMUL ST(3)

Assuming that top-of-stack is ST(2).

The opcode for this operation is 00100011

where, 3-LSB bits i.e. 011 represents ST(3) and next four bits i.e. 0100 represents multiplication bit. MSB does not have any meaning, that is don't care.

4.7 Precision converter

4.7.1 Conversion of Single precision to Extended-Double precision

The input number format is in accordance to the IEEE 754 standard single precision real number, and to enhance the accuracy of FPU result, internally the floating-point calculations are carried out in IEEE-754 standard extended-double precision format.

Algorithm

The Algorithm to convert single precision number to extended-double precision format is as follows:

- Step 1:** Place 31st bit (sign bit) of single precision to 79th of extended-double precision.
- Step 2:** Set the 63rd bit of extended-double precision to 1 (normalizing).
- Step 3:** Add $16383 - 127 = 16256$ to exponent field of single precision and place it in the exponent field of extended-double precision.
- Step 4:** Place 23 bits of mantissa part of single precision to MSB 23 bits of mantissa part of extended-double precision. Place the trailing zeros in the mantissa part of extended-double precision.

4.7.2 Conversion of Extended Double precision to Single precision

The output number is single precision floating-point format and as discussed in the previous section, the internal calculations are carried out in extended double precision format. Hence, the reverse conversion is also necessary. The conversion from extended-double precision of single precision is a trivial task. Here we need to consider the

rounding mode of the mantissa part of number and overflow and underflow of the exponent part of the number.

Algorithm

The Algorithm to convert extended-double number to single precision format is as follows:

Step 1: If the extended-double precision number corresponds to zero, NaN or infinite, replace with corresponding format of zero, NaN and infinite in single precision.

Step 2: If the number is valid number, subtract 16256 from the exponent field of extended-double precision and place it into exponent field of single precision. In case of overflow, place infinite number into single precision. In case of underflow, place zero number into single precision.

Step 3: Place MSB 22 bits of mantissa field of extended-double precision in the MSB 22bits of mantissa field of single precision. 23rd bit of mantissa field of single precision depends on the selected rounding mode.

Step 4: Place '0' in 23rd bit of mantissa field of single precision if the selected rounding mode is round to down. Place '1' in 23rd bit of mantissa field of single precision if the selected rounding mode is round to up. Place 40th bit of extended-double precision into 23rd bit of mantissa field of single precision if the selected rounding mode is round to zero. Place 39th bit of extended-double into 23rd bit of mantissa field of single precision if the selected rounding mode is round to nearest.

4.8 Addition/Subtraction unit

In this section, we will discuss the floating-point addition algorithm architecture and the hardware modules designed as part of this algorithm including their function, structure and use. The 80-bit FP adder/subtractor has a latency of 18 clock cycles (see simulation result in Figure 6.7).

4.8.1 Use of Tag word

The two operands are checked for their validity using tag word. If the Tag word for any operand contains any value other than 00, one of the following operations will be performed:

- If tag word for any operand is 01 or 11, the result is replaced with other number as these tag words represent zero and empty registers.
- If tag word for any operand is 10, the result is replaced with infinity as these tag words represent that infinity is contained in the corresponding register.

In above cases, the clock cycles consumed are 7. Thereby, tag register helps in saving the clock cycles and reduce latency.

4.8.2 Algorithm

Let S_1 ; E_1 ; F_1 and S_2 ; E_2 ; F_2 be the signs, exponents, and mantissas of two input floating-point operands. Given these two numbers, Figure 4.6 shows the flowchart of the floating-point adder algorithm. A description of the algorithm is as follows.

Stage 1: Unpacking Operands

The two sign, exponent and mantissa bits for operand A and operand B are latched in registers which are 1-bit, 15-bits, and 64-bits in length. The inputs are checked for special values: Infinity, Not a Number and Zero and the appropriate flags are set which is passed on through all stages.

Stage 2: Exponent Difference Module

The second stage in the adder uses comparator logic to place the larger of the two operands as operand A. The combinational VHDL process compares the exponents. If the exponents are equal, the logic then compares the mantissa values. The comparator is left to the synthesis tool. Sign bits of the two operands are XOR'ed. Sign bits do not affect the comparison.

Stage 3: Shift Mantissa Stage

In order to add two floating-point values in scientific notation, the two values must have the same exponent in both sign and magnitude. The adder must perform this operation by shifting one of the operands and making adjustments to the operand exponent value. Stage 2 has taken the difference of the two operand exponents to determine how many shifts are needed on operand B and accordingly exponent of operand b is adjusted. By shifting to the right, the operand stands to lose only lower significant bits. The maximum number of shifts needed is 64.

Stage 4: Mantissa Addition Stage

In this stage, the addition/subtraction of the two mantissa integer values is performed in accordance with the sign bits. Note that since operand A is greater than operand B, a borrow cannot happen in subtraction, and thus, the carry-out bit of the result is cleared. The carry-out bit becomes important in the next stage which may indicate the result needs no further normalization or exponent adjustment. If an addition took place with a carry-out, an immediate adjustment to the exponent must be done prior to the normalization stage since the bit does not take part in the 64-bit mantissa result vector. To do so, the stage must shift the result vector to the right by one to accommodate the carry-out bit as the new leading-one.

Different fixed-point adders are studied to determine which gives the best performance. The different kinds of integer adder used for comparative study are

1. Ripple Carry adder,
2. Carry Lookahead adder,
3. Carry Select adder and
4. Carry Save adder.

All adders are implemented at the logic-level and the working of each adder is explained in Section 4.8.3. The exponent and sign bits are stored in delay registers.

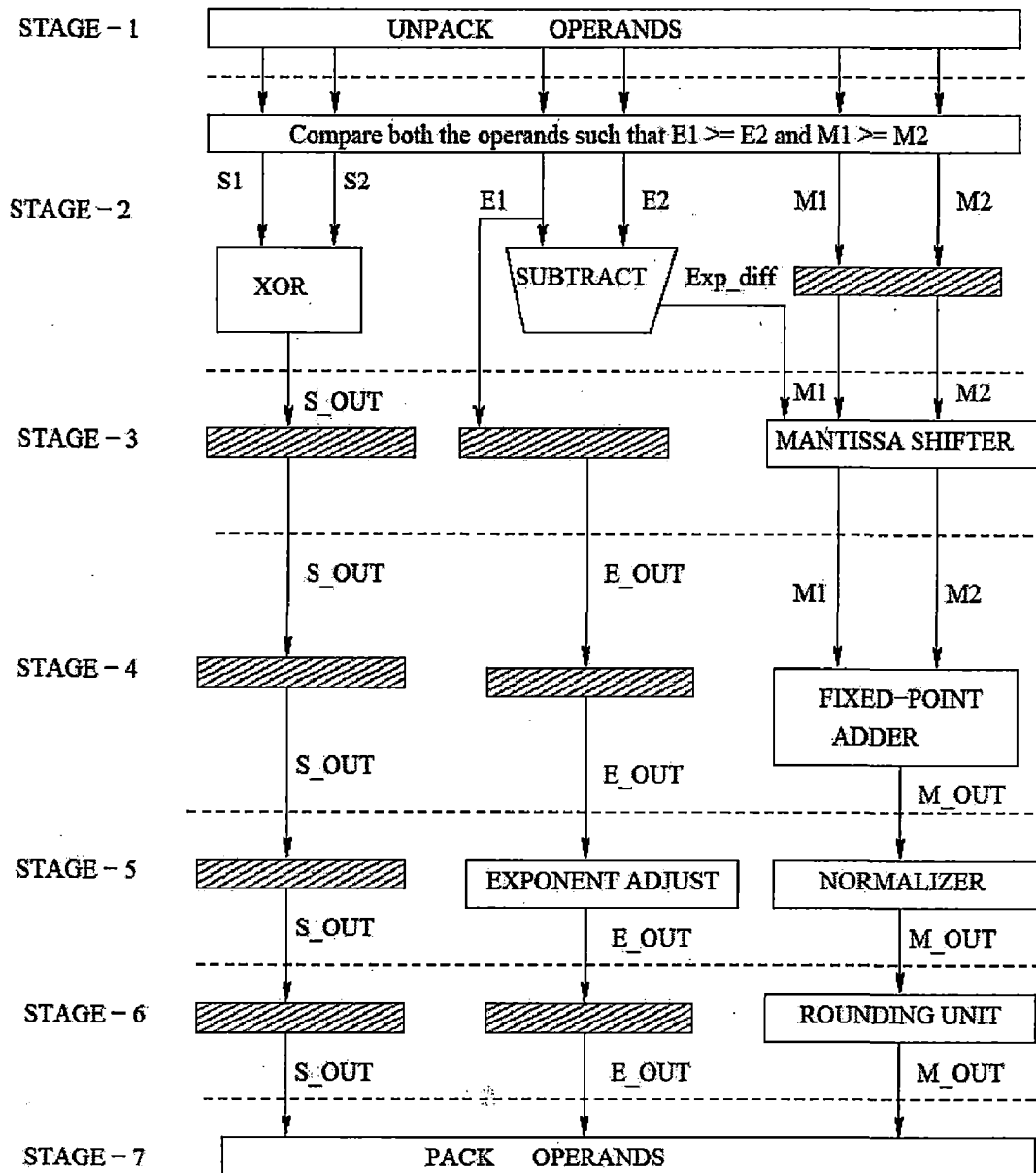


Figure 4.6 Addition/Subtraction algorithm

Stage 5 and Stage 6: Leading One Detector and Normalization Shift stage

After the addition, the next step is to normalize the result. The first step is to identify the leading or first one in the result [25][35][36]. Comparator logic is used here to find the first leading-one digit from the MSB. A counter maintains the number of comparisons made which is equal to the number of shifts needed. The shift value is

used to normalize the mantissa such that the leading-one in the mantissa resides in the most significant bit location. This stage also uses the shift value to adjust the exponent to the number of shifts required. Shifter in this stage is left to the synthesis tool.

Stage 7: Pack operands

Finally sign, exponent and mantissa are concatenated to form the 80-bits results and passed as an output from the FPU. The special condition flags are checked and if any of the flags are set high, then the result vary accordingly. The result is stored back in a 80-bit register.

4.8.3 Different types of integer adders and their comparative study

Different fixed-point adders are studied to determine which gives the best performance. Each adder chosen in this study has its own advantage of either having a simple design or high speed [29]. After a careful analysis, Block Carry Look Ahead adder is chosen for FPU design.

1 Ripple Carry Adder

The implementation of a ripple carry adder for two operands $x_{n-1}, x_{n-2}, \dots, x_0$ and $y_{n-1}, y_{n-2}, \dots, y_0$ is through the use of n basic units of full adder. A full adder (FA) is a logical circuit that accepts two operand bits, say x_i and y_i and an incoming carry, denoted by C_i . The outgoing carry, C_{i+1} is also the incoming carry for the subsequent FA, which has x_{i+1} and y_{i+1} as input bits. The FA is a combinatorial digital circuit implementing the binary addition of three bits through the following Boolean equations:

$$s_i = x_i \oplus y_i \oplus C_i$$

$$C_{i+1} = x_i \cdot y_i + C_i \cdot (x_i + y_i)$$

A RCA consisting of FA's for $n = 4$ is depicted in following figure. In parallel arithmetic unit, all $2n$ input bits (x_i and y_i) are usually available to the adder in the same time. However, the carries have to propagate from the FA in the position 0 (the position

of the FA whose inputs are x_0 and y_0) to position i in order for the FA in that position to produce the correct sum and carry-out bits. That is, we need to wait until the carries ripple through all n FAs before we can claim the sum outputs are correct.

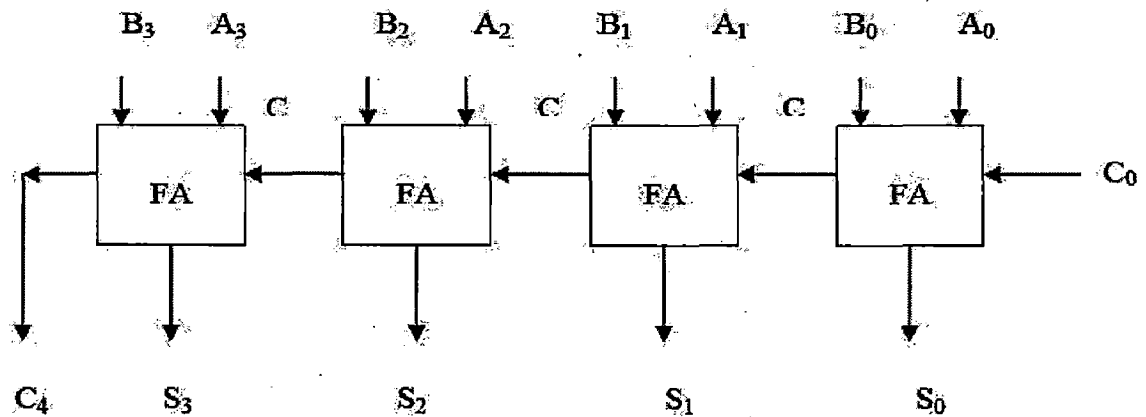


Figure 4.7 Ripple Carry Adder

The FA in position i has a combinatorial circuit with an incoming carry $c_i = 0$ at the beginning of the operation, and will accordingly produce a bit s_i . Ripple effect can be observed at the sum outputs of the adder as well, continuing until the carry propagation is done. The incoming carry in at position 0, c_0 , is always zero.

Disadvantage:

The obvious disadvantage of a RCA is the long carry propagation time. The worst case delay for a RCA is $n \cdot T_d$, where n is the number of bits and T_d is the operation time (delay) of an FA, assuming that the delays associated with generating the sum output and the carry-out are equal.

Advantage:

The advantage of this adder is the simplicity of the design and area occupied by the adder which is not very high.

2 Carry-Look-Ahead Adders

The main idea behind carry-look-ahead addition is an attempt to generate all incoming carries in parallel (for all $n-1$ high order FAs) and avoid the need to wait until the correct carry propagates from the stage (FA) of the adder where it has been generated. This is possible since the carries generated and the way they propagate depend only on the digits of the original numbers $x_{n-1}, x_{n-2}, \dots, x_0$ and $y_{n-1}, y_{n-2}, \dots, y_0$. These digits are available simultaneously to all stages of the adder and consequently each stage can have all the information it needs in order to calculate the correct value of the incoming carry and compute the sum bit accordingly. This leads to large inputs which may be reduced at each stage by extracting the information from the input digits needed to determine whether new carries will be generated and whether they will be propagated.

There are stages in the adder where $x_i = y_i = 1$, in which a carry-out is generated regardless of the incoming carry, and as a result, no additional information on the previous input digits is required. Other stages are only capable of propagating the incoming carry.

Following logic functions are defined to assimilate the information regarding generation and propagation of carries, using logic functions OR and AND operation. Let $G_i = x_i \cdot y_i$ denote the generated carry and let $P_i = x_i + y_i$ denote the propagated carry. Hence the boolean expression for carry out is:

$$c_{i+1} = x_i \cdot y_i + c_i \cdot x_i + y_i = G_i + c_i \cdot P_i$$

Replacing $c_i = G_{i-1} + c_{i-1}P_{i-1}$ in the above expression

$$c_{i+1} = G_i + G_{i-1}P_i + c_{i-1}P_{i-1}P_i$$

Further substitutions allows us to calculate all the carries in parallel from the original digits $x_{n-1}, x_{n-2}, \dots, x_0$ and $y_{n-1}, y_{n-2}, \dots, y_0$ and forced carry c_0 . For example, for a 4-bit adder, the carries are

$$c_1 = G_0 + c_0P_0$$

$$c_2 = G_1 + G_0P_1 + c_0P_0P_1$$

$$c_3 = G_2 + G_1P_2 + G_0P_1P_2 + c_0P_0P_1P_2$$

$$c_4 = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 + c_0P_0P_1P_2P_3$$

The following figure depicts the working of 4-bit CLA.

More generally for any j with $i < j$, $j + 1 < k$, we have the recursive relations

$$c_{k+1} = G_{ik} + P_{ik}c_i$$

$$G_{ik} = G_{j+1,k} + P_{j+1,k}G_{ij}$$

$$P_{ik} = P_{ij}P_{j+1,k}$$

The above equation says that carry is generated out of the block consisting of bits i through j inclusive if it is generated in the high-order part of the block ($j+1, k$) or if it is generated in the low-order part of the block (i, j) and then propagated through the high part.

Advantage:

The bits in a CLA must pass through about $\log_2 n$ logic levels, compared with $2n$ for a ripple-carry adder. This is a substantial speed improvement, especially for a large n .

Disadvantage:

Comparing area, ripple carry adder had n cells, whereas the CLA has $2n$ cells.

The point is that a small investment in size pays off in a dramatic improvement in speed.

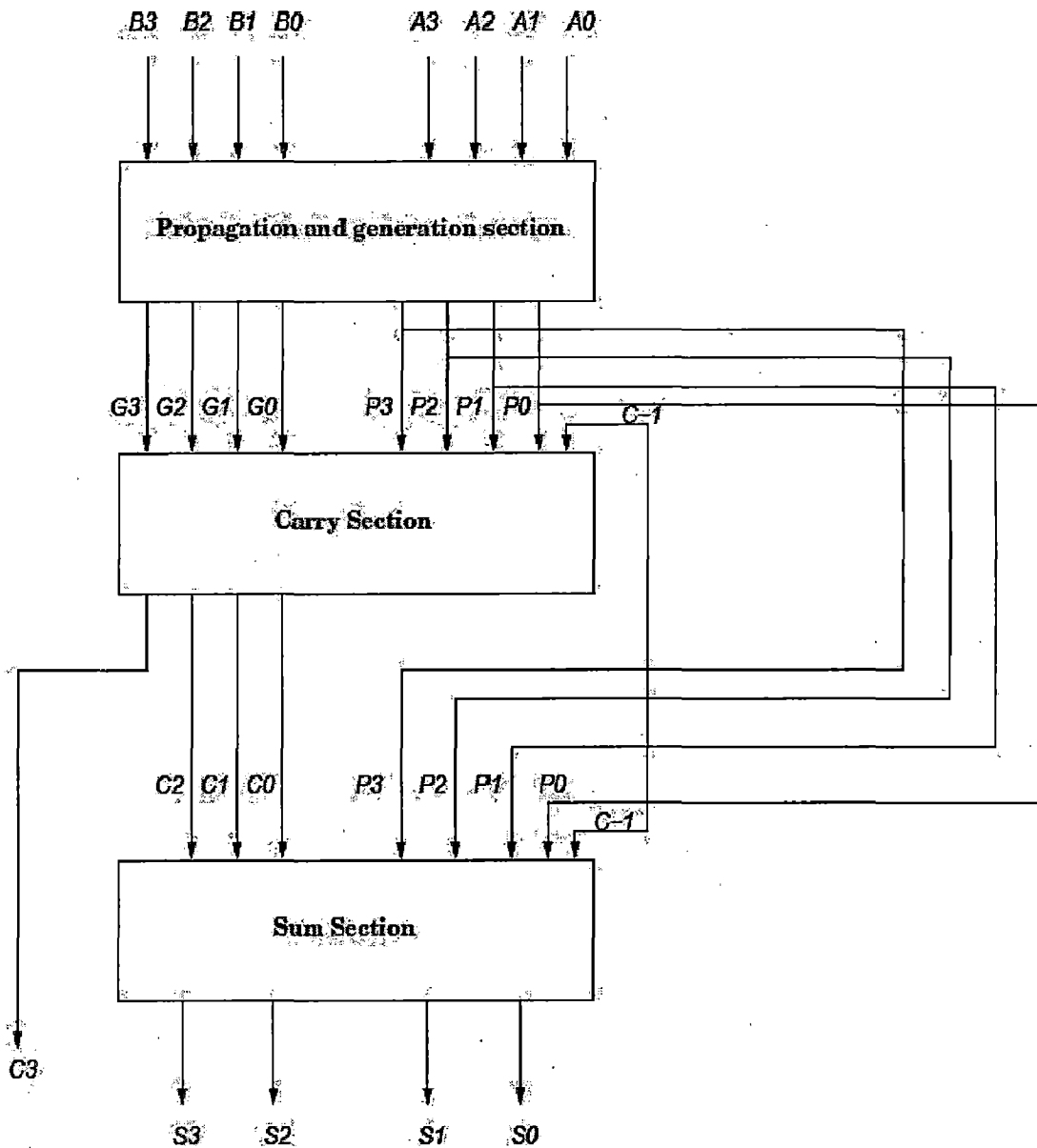


Figure 4.8 Carry Look Ahead Adder

3 Carry Select Adders

Carry select adder is another fast adder that provides a logarithmic speed-up [27]. The principle behind this scheme is to generate two sets of outputs for a given group of operand bits, say k bits.

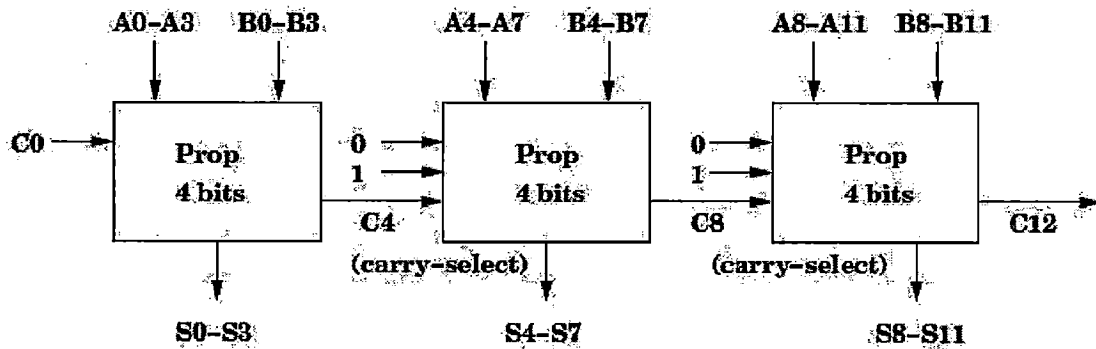


Figure 4.9 Carry Select Adders

Each set includes k sum bits and an outgoing carry. One set assumes that eventually incoming carry will be zero, while the other assumes that it will be one. Once the incoming carry is known, we need only to select the correct set of outputs (out of the two sets) without waiting for the carry to further propagate through the k positions.

This idea should not be applied to all n operand bits at the beginning of the add operation, since we will then have to wait until the carry propagates through all n positions before making the selection. Therefore the above idea has to be applied after given n bits is divided into smaller groups. This allows serial carry-propagation inside the individual groups to be performed in parallel which reduces the overall execution time. Each group generates two sets of sum bits and an outgoing carry bit. The incoming carry selects one of these two sets. The working of a 12-bit CSA is shown in above figure.

Comparison with previous two adders:

In general CSAs require more gates than CLAs and CSA have almost the same speed as CLA. The design of CSA is however less modular than CLA and this is the main reason for higher popularity of CLA. The delay of CSA is proportional to \sqrt{n} , which is lesser than the delay for RCAs but greater than CLAs.

4 Block Carry-Look-Ahead Adder

As we discussed earlier in this section, CLA needs an extremely large number of gates and more importantly, gates with high fan-in are required. This can be compensated

by reducing the span of the look-ahead at the expense of speed. For this, we have to divide the n stages into groups and have a separate carry lookahead in each group.

The groups can be interconnected by the ripple carry method. By dividing the adders into equal sized groups, modularity increases. A group size of 4 is chosen, as it is a common factor of most word sizes, and also because of technology dependent constraints (example, the available number of input/output pins).

For n bits and groups of size 4, there are $n/4$ groups. T_o propagate a carry through a group once the P_i 's, G_i 's and C_o are available, we need $2T_G$ time units. Thus, $1T_G$ is needed to generate all P_i and G_i and $2T_G$ is needed to generate the sum outputs, for a total of

$$(\Delta 2n/4 + 3)\Delta_G = (n/2 + 3)\Delta_G$$

This is almost fourfold reduction in delay compared to the $2nT_G$ of a ripple carry adder. Group-generated carry, G^* and a group-propagated carry, P^* , for a group of size 4 are as follows:

$G^* = 1$ if a carry-out is generated internally and $P^* = 1$ if a carry-in is propagated internally to produce a carry-out. The Boolean equations for these carries are

$$G^* = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3$$

$$P^* = P_0P_1P_2P_3$$

The group-generated and group-propagated carries for several groups can now be used to generate carry-ins in a manner similar to single-bit carry-ins in above equation. A combinatorial circuit implementing these equations is called a carry look-ahead generator. As the number of bits, n , increases, more levels of carry lookahead generators can be added in order to speed up the addition. The overall addition time of a carry lookahead adder is therefore proportional to $\log_b n$, where b is the blocking factor.

All the above integer adder units were simulated and tested to meet the desired specifications. Based on the simulation results, I selected Block Carry Look Ahead adder because of its high speed and area efficient characteristics.

4.9 Multiplication unit

The second basic arithmetic operation needed to perform FPU operations in this thesis is the multiplication. Constructing a fast multiplier in an FPGA presents a challenge due to the sheer amount of logic required. Traditional integer multiplier has been studied and number of stages has been modified to give the best performance and area.

According to [16], floating-point multiplication is inherently easier to design than floating-point addition. Multiplication requires integer addition of operand exponents and integer multiplication of significands which facilitate normalization when multiplying normalized significands. These independent operations within a multiplier make it ideal for pipelining.

The fixed point multipliers used in *multiply mantissa* stage can be non-pipelined, partially pipelined or fully pipelined. In this thesis, study of pipelined and non-pipelined fixed point multipliers has been done. The pipeline latency remains the only drawback which is not a concern in this study as the FPU is receives operands every clock cycle. By using a pipelined multiplier, the resource consumption not only decreases but the speed actually increase.

4.9.1 Use of Tag word

The two operands are checked for their validity using tag word. If the Tag word for any operand contains any value other than 00, one of the following operations will be performed:

- If tag word for any operand is 01, the result is replaced with zero as this tag word represent zero in the corresponding register.
- If tag word for any operand is 11, the result is replaced with another operand as this tag word represents empty registers.
- If tag word for any operand is 10, the result is replaced with infinity as these tag words represent that infinity is contained in the corresponding register.

In above cases, the clock cycles consumed are 7. This allows save clock cycles in certain case.

4.9.1 Algorithm

The following section describes the different stages of floating point multiplication algorithm. The 80-bit FP multiplier has a latency of 4τ clock cycles. All symbols have usual meaning.

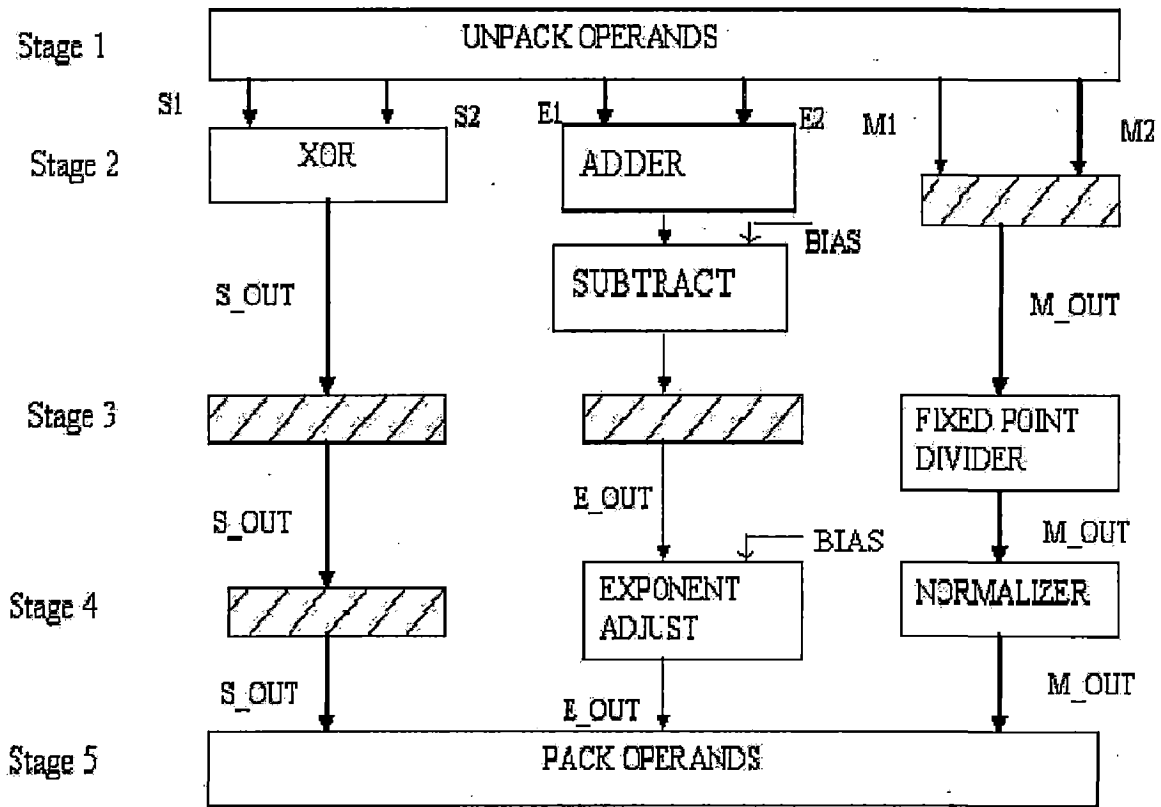


Figure 4.10 Multiplication Algorithm

Stage 1: Unpacking Operands

The two sign, exponent and mantissa bits for operand A and operand B are latched in registers which are 1-bit, 15-bits, and 64-bits in length. The inputs are checked for special values: Infinity, Not a Number and Zero and the appropriate flags are set which is passed on through all stages.

Stage 2: Calculate exponent and sign

The exponents of operand A and operand B are added together and the bias is subtracted from the result which gives the resultant biased exponent. The sign bits of the two operands are XOR'ed to give the resultant sign bit. The sign and exponent output are passed on through all stages.

Stage 3: Multiply mantissa

The mantissa fields of operand A and operand B are multiplied. The output of the fixed-point multiplier is double the mantissa length.

The different kinds of fixed-point multipliers used for comparative study are

1. Shift-Add multiplier,
2. Booth multiplier

All multipliers are implemented at the logic-level and the working of each adder is explained in Section 4.9.3. The exponent and sign bits are stored in delay registers.

Stage 4: Normalization Shift stage

After the multiplication, the next step is to normalize the result. The first step is to identify the leading or first one in the result. In multiplication, leading one is either available at MSB or next to MSB. Comparator logic is used here to find the first leading-one digit from the MSB. The shifting is used to normalize the mantissa such that the leading-one in the mantissa resides in the most significant bit location and accordingly the exponent is adjusted. The upper 64 bits of the result are retained as mantissa.

Stage 5: Pack operands

Finally sign, exponent and mantissa are concatenated to form the 80-bits results and passed as an output from the FPU. The special condition flags are checked and if any of the flags are set high, then the result vary accordingly. The result is stored back in a 64-bit register.

4.9.3 Different types of integer multipliers and their comparative study

Different fixed point multipliers are studied to finalize the fixed-point multiplier which gives the best performance in terms of area and delay. On the basis of simulation results, obtained using Active HDL 6.1, Booth's algorithm is found to be the best suitable algorithm available for my FPU design. High speed multipliers can be classified as parallel, sequential and array multipliers. The first generates all partial products in parallel and uses a high-speed adder to accumulate them, whereas the second generates the partial products sequentially and adds them together. Array multipliers are made up of identical cells that generate new partial products and accumulate them simultaneously. After a vast study of previous work the following multipliers are studied which either have reduced execution time or less hardware complexity.

1. Shift add multiplier (SA)
2. Booth multiplier

Multiplications are essentially a series of additions. The different integer multiplier discussed deals with two main operations: generating partial products and different ways of adding partial products. There are different methods to encode the multiplicand which can mainly be classified as non-booth and booth encoding. Hence two an adder from each category has been chosen for this study being Shift-Add multiplier and Booth multiplier.

How to speed up the addition/subtraction?

There are two ways that the additions can be speeded up:

1. Speeding up each addition.

To speed the addition the one of the fast adders which has already been studied can be used.

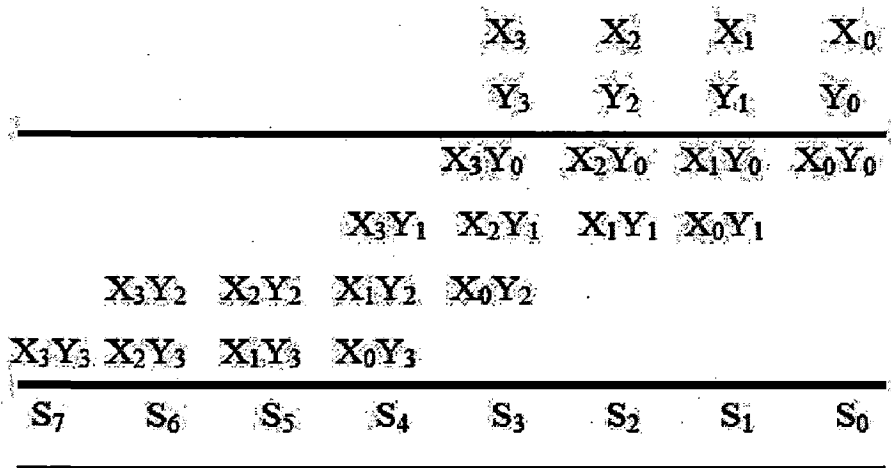
2. Reducing the number of additions required.

The number of additions can be reduced in two ways. One is to shift over strings of 0's and 1's without doing any addition and the other is to scan two or more multiplier digits each cycle and hence add larger multiples of the multiplicand in each cycle. After the partial products are produced they can be added using a fast adder like carry save adder.

1 Shift Add Multiplier

The first and simplest method for encoding is non-Booth. This algorithm is simply a shift and add algorithm where the multiplicand is conditionally added to produce the final result. Shift-add multiplier uses the concept of an array multiplier which is described in detail in this section. An array multiplier is constructed by an array of identical elementary processor units, each of which processes single-bit data. The basic unit usually consists of a partial-product bit generator and a full adder. A new partial-product bit is generated and added to the previous accumulated partial product in one cell [37].

To illustrate the operation of a shift-add multiplier, consider a 4 x 4 multiplication shown in following figure, which contains all 16 partial-product bits in the form of $X_i Y_j$. The array adds the first two levels of partial product bits e.g. $X_3 Y_0$ to $X_0 Y_0$ and $X_3 Y_1$ to $X_0 Y_1$ together in the second row of array (first row of the array adds the partial products from previous stage and the first level of partial product bits together if it is in multiple length multiplication) after proper alignment. The results of the second row are then transferred to the third row and added to the third level of partial product bits $X_3 Y_2$ to $X_0 Y_2$, and so on. All additions are done using a Block Carry lookahead adder.



The block diagram of the basic processor cell and the 4 x 4 array multiplier is depicted in following figure 4.11.

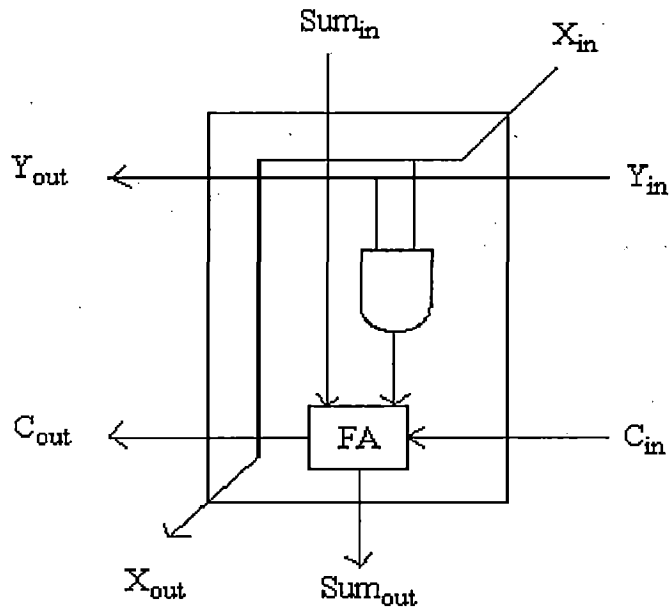


Figure 4.11 (a) Basic cell

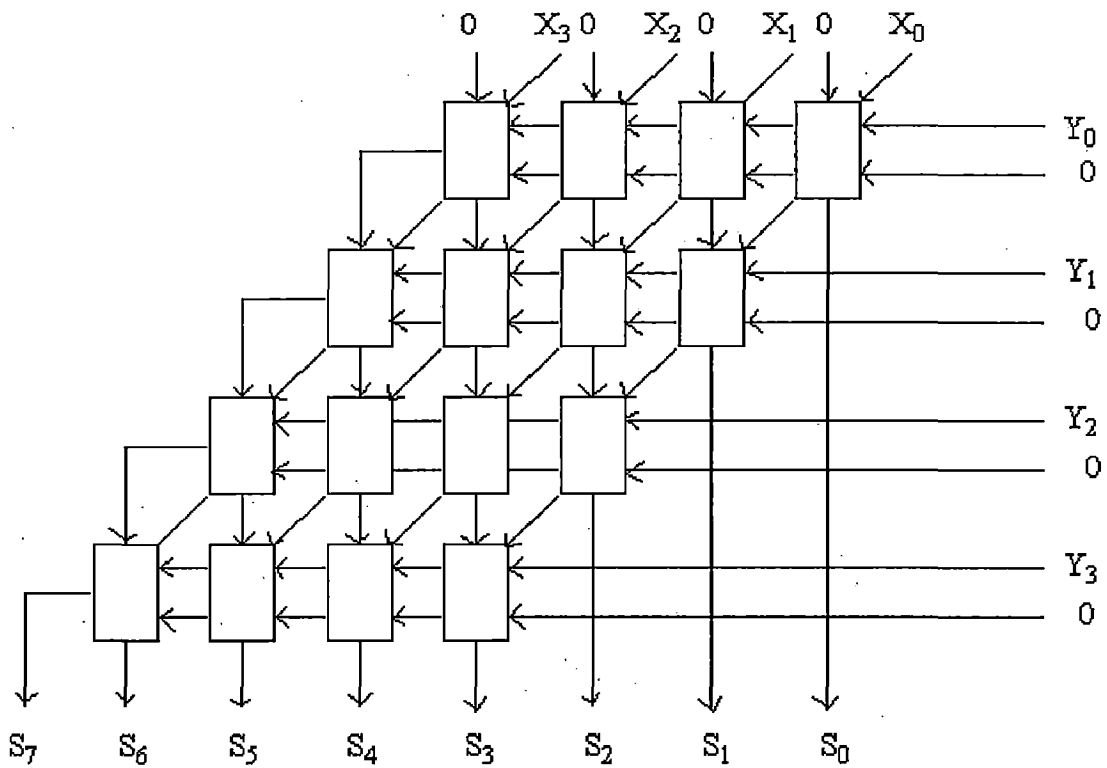


Figure 4.11 (b) Shift-Add Multiply

Advantages:

The simple and regular structure of the shift-add multiplier makes the design and layout processes easy and suitable for automatic generation.

Disadvantages:

It requires a large amount of silicon area and the speed is low since the delay depends on the depth of the array. It is also inefficient because as the number propagates through the array, each row of the processor units is used only once. Unfortunately, there is no reduction in the number of multiplicands that need to be summed to produce the final result. Although it is easy to pipeline which increases the throughput and utilization greatly, the additional latches needed increase both the hardware and latency.

2 Booth algorithm for multiplication

The high-level block diagram of the multiplier is shown in figure 4.12. It consists of four distinct components. They are the Booth Encoder, Partial Product Generator, Carry Save adder, and the Carry Lookahead adder.

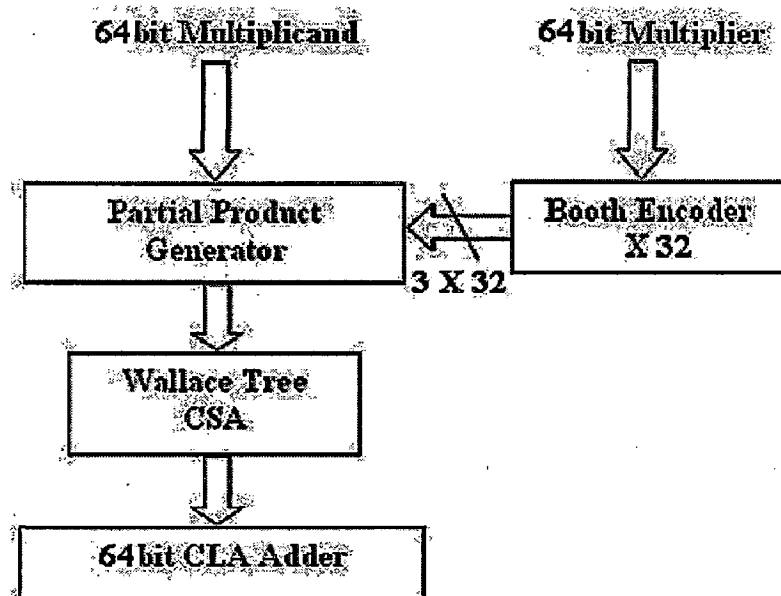


Figure 4.12 Architecture of the Booth multiplier

There are two main techniques that can be used to increase the speed of the multiplication process [29]. First technique is to reduce the number of partial product and the second is to increase the speed at which the partial products are added. The proposed architecture employs both of these techniques in the design. The individual components are shown and explained in detail below.

a) Booth encounter

This module encodes the 64-bit multiplier using radix 4 Booth's algorithm. Radix 4 encoding reduces the total number of multiplier digits by a factor of two, which means in this case the number of multiplier digits will reduce from 64 to 32.

Table 4.2 Booth Multiplication

Multiplier Bits			Output bits		Operation on Multiplicand
Y_{i+1}	Y_i	Y_{i-1}	NEG	2 1	
0	0	0	0	0 0	0x
0	0	1	0	0 1	+1x
0	1	0	0	0 1	+1x
0	1	1	0	1 0	+2x
1	0	0	1	1 0	-2x
1	0	1	1	0 1	-1x
1	1	0	1	0 1	-1x
1	1	1	1	0 0	0x

This algorithm groups the original multiplier into groups of three consecutive digits where the outermost digit in each group is shared with the outermost digit of the adjacent group. Each of these groups of three binary digits then corresponds to one of the numbers from the set {2, 1, 0, -1, -2}. Each encoder produces a 3-bit output where the first bit represents the number 1 and the second bit represents the number 2. The third and final bit indicates whether the number in the first or second bit is negative. Since there are 64 input bits, there will be a total of 32 Booth encoder modules in the overall multiplier architecture. The way the outputs are determined is shown in Table 4.2.

b) Partial Product Generation(PPG)

The output from the Booth encoder is used in this module to generate the partial products. Since there are 32 Booth encoders there will be a total of 32 partial products. The multiplication by two is implemented by shifting the multiplicand left one bit and the negation is implemented by taking the two's complement of the multiplicand. The architecture of the partial product generator (for a 16-bit number) is shown in Figure 4.13.

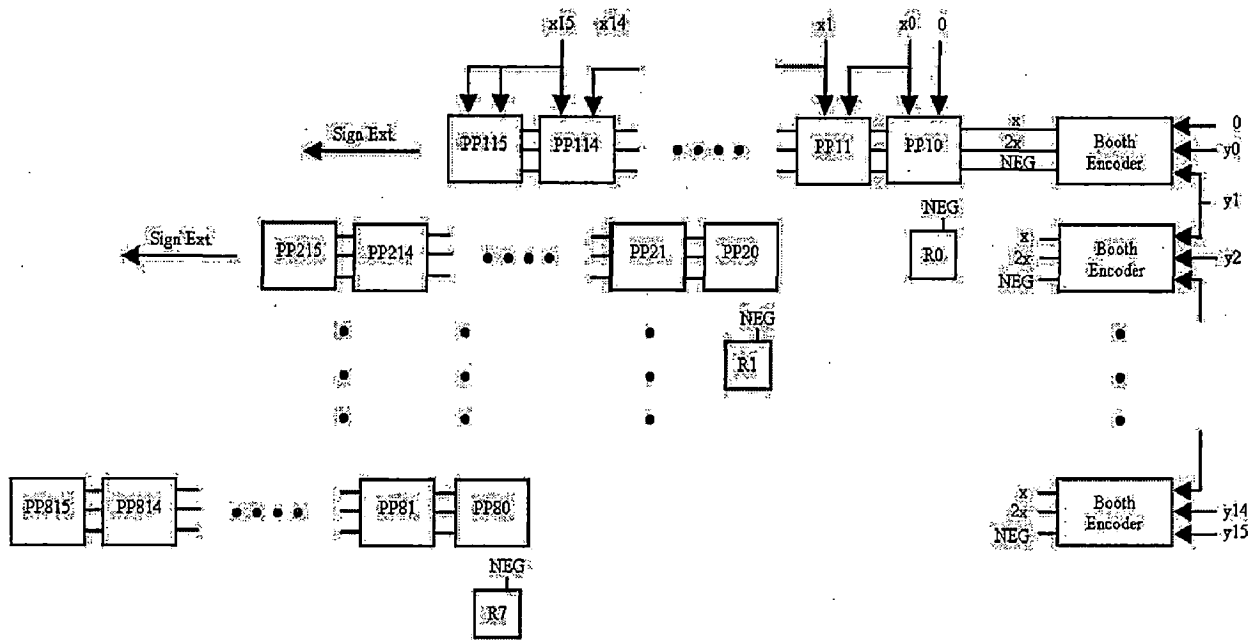


Figure 4.13 Partial Product Generation

Each row of the diagram corresponds to one partial product. Even though the diagram does not show it, there are eight such rows corresponding to eight partial products. Also, each partial product is shifted two bits to the left relative to the partial product above it to account for the radix 4 Booth encoding of the multiplier.

c) Wallace Tree

This module is responsible for adding the partial products that were generated in the PPG module. This module uses 3 to 2 carry save adders (CSA) to implement the Wallace Tree. The individual CSAs are nothing more than full adders except for the

fact that the carry-ins and the carry-outs are handled in a special way. Each column of numbers in the partial product is added using this method. Figure 4.14 below shows how this method works for adding 8 bits. The carry-outs generated in each stage of addition are transferred to the Wallace Tree of the column of bits of partial products on the left and the carry-ins comes from the column to the right.

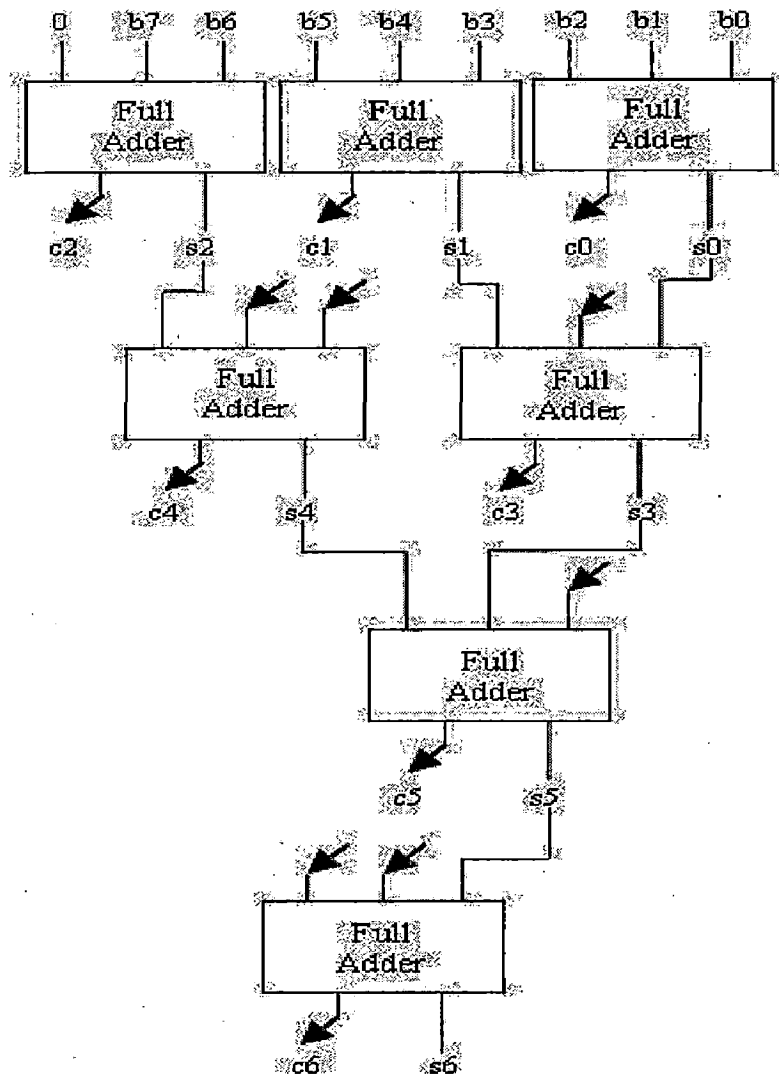
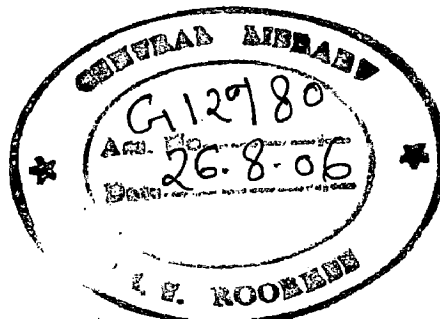


Figure 4.14 Wallace Tree

The advantage of using a Wallace Tree structure for addition is that for adding eight bits the result is available only after four full adder delays. If the same addition were to be performed using a ripple carry adder, it would have required seven full adder



delays. Therefore, although the structure of the adder might be a little complicated, it greatly increases the speed of addition.

d) Carry Lookahead adder

This unit is used to add the final sum and carry vectors generated by the Wallace Trees for each column of bits from the partial products. Only a 64 bit CLA is needed, instead of full 128 bits, because some of the bits of the final result are already available from the Wallace Trees.

4.10 Division unit

Division is the most time-consuming and infrequent operation amongst the arithmetic operations. Many algorithms have been developed for implementing division in hardware. These algorithms differ in many aspects, including quotient convergence rate, fundamental hardware primitives, and mathematical formulations. Division algorithms are divided into classes based upon the differences in the hardware operations used in their implementations, such as multiplication, subtraction, and table look-up.

Division algorithms can be divided into five classes: digit recurrence, functional iteration, very high radix, table look-up, and variable latency. In this work, I have considered three algorithms which are simulated using Active HDL 6.1. Of these digit-recurrence division algorithm is chosen for my design because of its low latency. Performance of the divider is very important in this study as the output of the divider shall be the input to the multiplier/subtractor or some other unit of FPU [23]. The 80-bit floating point divider has a latency of 18 clock cycles.

4.10.1 Use of Tag word

The two operands are checked for their validity using tag word. If the Tag word for any operand contains any value other than 00, one of the following operations will be performed:

- If tag word for divider is 01, the result is set with infinity as this tag word represent zero in the corresponding register. And if the tag word for divisor is 01, then the result is set to zero.

- If tag word for any operand is 11, the result is unknown as this tag word represent empty registers.
- If tag word for divider is 10, the result is set with zero as this tag word represent infinity in the corresponding register. And if the tag word for divisor is 10, then the result is set to infinity.

In above cases, the clock cycles consumed are 7. This allows save clock cycles in certain case.

4.10.2 Algorithm

This section describes the algorithm implemented.

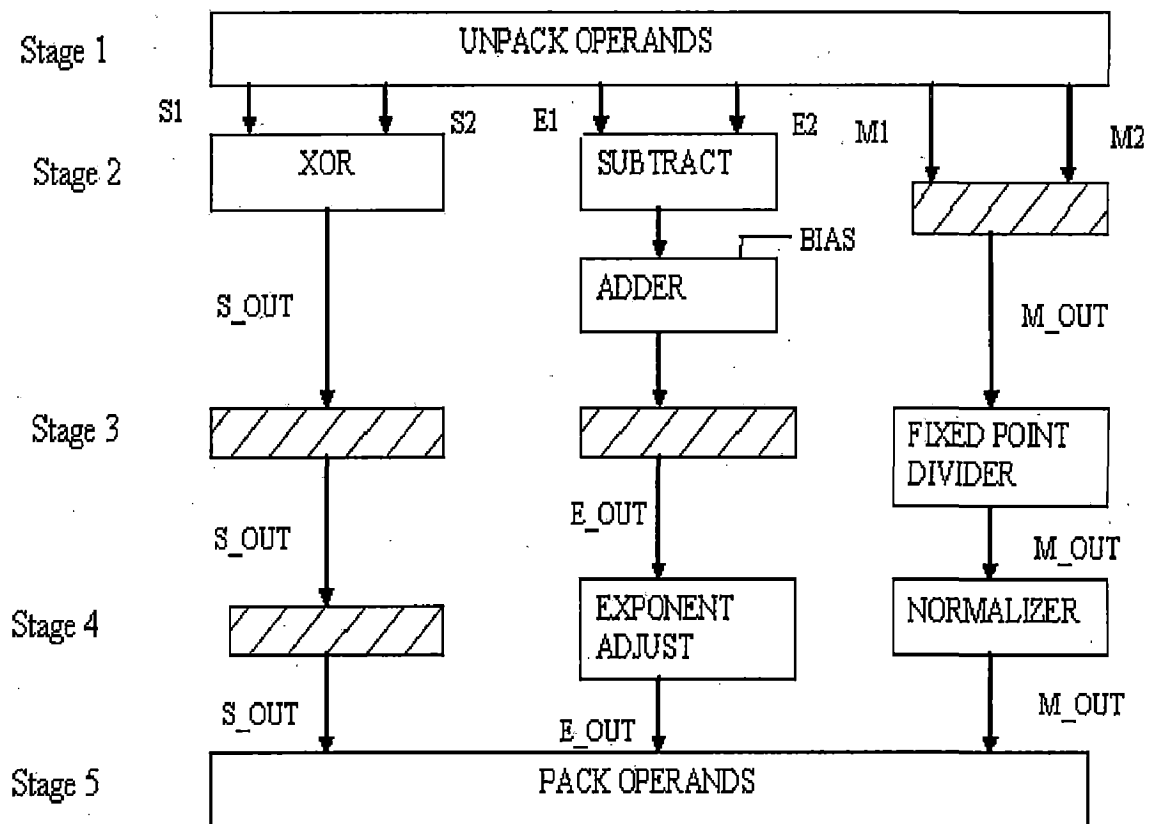


Figure 4.15 Division Algorithm

Stage 1: Unpacking Operands

The two sign, exponent and mantissa bits for operand A and operand B are latched in registers which are 1-bit, 15-bits, and 64-bits in length. The inputs are checked

for special values: Infinity, Not a Number and Zero and the appropriate flags are set which is passed on through all stages.

Stage 2: Calculate exponent and sign

The exponent of operand B is subtracted from that of operand A and the bias is added to the result which gives the resultant biased exponent. The sign bits of the two operands are XOR'ed to give the resultant sign bit. The sign and exponent output are passed on through all stages.

Stage 3: Divide mantissa

The mantissa field of operand A is divided by that of operand B.

The different integer dividers studied and simulated are :

1. Storing and non-restoring division
2. SRT division

All dividers are implemented at the logic-level and the working of each divider is explained in Section 4.10.3. The exponent and sign bits are stored in delay registers.

Stage 4: Normalization Shift stage

After the division, the next step is to normalize the result. The first step is to identify the leading or first one in the result. In division, leading one is either available at MSB or next to MSB. Comparator logic is used here to find the first leading-one digit from the MSB. The shifting is used to normalize the mantissa such that the leading-one in the mantissa resides in the most significant bit location and accordingly the exponent is adjusted. The upper 64 bits of the result are retained as mantissa.

Stage 5: Pack operands

Finally sign, exponent and mantissa are concatenated to form the 80-bits results and passed as an output from the FPU. The special condition flags are checked and if any

of the flags are set high, then the result vary accordingly. The result is stored back in a 80-bit register.

4.10.3 Different types of integer dividers and their comparative study

Following are the algorithms studied and simulated using Active HDL 6.1. It is found that Digit Recurrence algorithm is easy to design, occupies less space and has low latency.

a) Digit Recurrence Algorithm

The simplest and most widely implemented class of division algorithms is digit recurrence [42]. Digit recurrence algorithms retire a fixed number of quotient bits in every iteration. Implementations of digit recurrence algorithms are typically of low complexity, utilize small area, and have relatively large latencies. The fundamental choices in the design of a digit recurrence divider are the radix, the allowed quotient digits, and the representation of the partial remainder. The radix determines how many bits of quotient are retired in an iteration. Larger radices can reduce the latency, but increase the time for each iteration. This section introduces the principles of digit recurrence division, along with an analysis of methods for increasing the performance of digit recurrence implementations.

b) Restoring and non-restoring division

Given a dividend I and a divisor D , the quotient Q and remainder R are defined by:

$$I = Q \cdot D + R \text{ with } 0 \leq R$$

The division is performed by a sequence of subtractions and multiplications, as described by the following recursion formula.

$$P_i = r \cdot P_{i-1} - q_i \cdot D$$

where, r is the radix. P_i is the new partial remainder after the i th iteration, q_i is the i th quotient digit which is determined by comparing P_{i-1} and D . The comparison process is

usually done by subtracting $q_i \cdot D$ from $r \cdot P_{i-1}$. If the result of the subtraction is positive, q_i will be increased and the subtraction process will be repeated until the result becomes negative. When P_i is negative, q_i is decreased by 1 and the partial remainder P_i must to be restored to its previous value by adding one divisor D to it. Therefore, this method is called restoring division [45].

Non-restoring division also based on above equation, but the quotient digit is not corrected and the remainder is not restored immediately if it is negative. The correcting operations are postponed to later steps. By allowing the quotient digit to be negative, the restoring operation can be avoided. In restoring division, q_i can be only 1 or 0, but in non-restoring division, q_i belongs to the digit set $\{-1,1\}$. Notice that 0 is not allowed in non-restoring division. When the shifted partial remainder $2P_{i-1}$ equal to 0, the division process terminates.

c) SRT division

Each step in the division is dependent on previous ones, so the next step cannot begin unless the current remainder is known. The quotient digits are obtained from the current partial remainder. If the quotient digit-selection process in each step can be simplified, the time to complete each step can be shortened and the speed of division can be increased [30]. In radix-2 restoring/non-restoring division, the quotient digits are obtained by comparing the divisor and the partial remainder in full precision. If the comparison process can be speed up, then the division process can be accelerated. To achieve this goal, the precision used in the comparison must be reduced. This is the idea of SRT division.

SRT division is the most common implementation of digit recurrence division in modern microprocessors, taking its name from the initials of Sweeney, Robertson and Tocher, who developed the algorithm independently at approximately the same time. We analyze the algorithm for an n -bit number. The input operands are assumed to be represented in a normalized floating-point format with n bit significands in sign and magnitude representation. The quotient is defined to comprise k radix- r digits with

$$r = 2^b$$

$$k = n/b$$

where a division algorithm that retires b bits of quotient in each iteration is said to be a radix- r algorithm. Such an algorithm requires k iterations to compute the final n bit result and thus has a latency of k cycles. The cycle time of the divider is defined as the maximum time to compute one iteration of the algorithm. In this thesis, we analyze radix-2 SRT algorithm, hence k , no of iterations is equal to the number of bits in the mantissa.

4.10.4 Division Parameters

a) Radix Selection

The fundamental overall method of decreasing the latency of the algorithm is to increase the radix r of the algorithm, typically chosen to be a power of 2. However, as the radix increases, the quotient digit selection becomes very complicated, which may increase the cycle time. Moreover the generation of all required divisor multiples may become impractical for higher radices. Oberman shows that the delay of quotient selection tables increases linearly with increasing radix, while the area increase quadratically. Pre-scaling of the input operands reduces the table complexity at the expense of additional latency. The limitation in generating all of the required divisor multiples for radix-8 and higher limits practical divider implementations to radix-2 and radix-4.

b) Quotient Digit Set

For a given choice of radix r , some range of digits is decided upon for the allowed values of the quotient in each iteration. The simplest case is where, for radix r , there are exactly r allowed values of the quotient. However, to increase the performance of the algorithm, a *redundant digit set* is used. This allows a quotient digit to be selected based upon an approximation of the partial remainder representation as discussed in the next section. Such a digit set is composed of symmetric signed-digit consecutive

integers, where the maximum digit is a . The digit set is made redundant by having more than r digits, the complexity and latency of the quotient selection function is reduced. However choosing a smaller number of allowed digits for the quotient simplifies the generation of the multiple of the divisor. Quotient digit-set is said to be canonical if $0 \leq q_j \leq r - 1$. Whereas it is said to be redundant when, $q_j \in D_a = (-a, -a + 1, \dots, -1, 0, 1, \dots, a)$, where the redundancy factor $\rho = a/r - 1$ ($\rho > 1/2$). Specifically for radix 2, the digit set is $\{-1, 0, 1\}$.

c) Partial-Remainder Representation

The partial remainder can also be represented in two different forms, either redundant or non-redundant. Each iteration of the algorithm requires a subtraction to compute the next partial remainder. If this partial remainder is in a non-redundant form, then this operation requires a time-consuming full width carry propagate adder, increasing the cycle time. Therefore the partial remainder is typically stored in redundant form so that a fast carry save adder, can be used in the partial remainder calculation.

Quotient digit selection and remainder representation in radix-2

By allowing 0 to be one of the quotient digit choices, the radix-2 quotient digit selection is changed to equation shown below.

$$q_i = \begin{cases} 1 & \text{if } 2 \cdot R_{i-1} \geq D \\ 0 & \text{if } -D \leq 2 \cdot R_{i-1} < D \\ -1 & \text{if } 2 \cdot R_{i-1} < -D \end{cases}$$

Note that the regions overlap. This is good as it means we have freedom to pick a digit even if we don't know exactly what P and D are. For binary SRT, it's particularly easy, as we can get away with just looking at the sign bit of P . If it's 0, we know $0 \leq P$, so we can pick $D = 1$. If it's 1, we know $P < 0$, so we can pick $D = -1$. In other words, non-restoring division is a special case of SRT division. If D is a normalized fractional number, such that $1/2 < |D| < 1$, the thresholds in above equation can be reduced from

$|D|$ to $1/2$. This is because if $|D|$ is larger than or equal to $1/2$, then the range $[-D, D]$ must include the range $[-1/2, 1/2]$. That means

$$|-D| \leq -1/2 \leq 2 \cdot R_{i-1} < 1/2 \leq |D|$$

Therefore, the comparison operation (in above equation) can be reduced to the following simplified form.

$$q_i = \begin{cases} 1 & \text{if } 2 \cdot R_{i-1} \geq 1/2 \\ 0 & \text{if } -1/2 \leq 2 \cdot R_{i-1} < 1/2 \\ -1 & \text{if } 2 \cdot R_{i-1} < -1/2 \end{cases}$$

Now the partial remainder $2 \cdot R_{i-1}$ can be compared to either $1/2$ or $-1/2$, instead of D or $-D$. This reduces the time required to generate quotient digits. A binary fraction is larger than or equal to $1/2$ if it starts with (0.1) . Similarly, a binary fraction is smaller than $-1/2$ if it starts with (1.0) . That means, only the first two bits of $2 \cdot R_{i-1}$ need to be examined to determine the quotient digit, instead of full comparison.

This is the basis of radix-2 SRT division. The following rules need to be followed for the selection process when D is normalized for a radix-2 SRT division.

- 1 If $2P_{j+1} < -1/2$, $q_{j+1} = -1$ and $P_{j+1} = 2P_j + D$
- 2 If $-1/2 \leq P_{j+1} < 1/2$, $q_{j+1} = 0$ and $P_{j+1} = 2P_j$
- 3 If $2P_{j+1} \geq 1/2$, $q_{j+1} = 1$ and $P_{j+1} = 2P_j - D$

The number of iterations required for SRT division decreases as the radix increases. Therefore, for high-radix SRT division, the complexity of the quotient selection process increases may eliminate the advantage of the reduction in number of iterations. This makes SRT division impractical for high radices, such as 256 or 512.

4.11 Square root unit

Square root algorithm is hard to implement on FPGAs because of complexity of the algorithms. In this thesis, I worked upon three algorithms and finally implemented a non-restoring square root algorithm. The operation latency is 66 clock cycles.

4.11.1 Use of Tag word

The operand is checked for its validity using tag word. If the Tag word contains any value other than 00, one of the following operations will be performed:

- If tag word is 01, the result is set with zero as this tag word represent zero in the corresponding register.
- If tag word is 11, the result is unknown as this tag word represents empty registers.
- If tag word is 10, the result is set with zero as this tag word represents zero in the corresponding register.

In all of the above cases, the clock cycles consumed are seven. The above mentioned work of tag helps in saving the clock cycles and thus reduces latency.

4.11.2 Algorithm

This section describes the algorithm of square root unit.

Stage 1: Unpacking Operands

The sign, exponent and mantissa bits of the operand are latched in register which are 1-bit, 15-bits, and 64-bits in length. The inputs are checked for special values: Infinity, Not a Number and Zero and the appropriate flags are set which is passed on through all stages.

Stage 2: Check Exponent Module

The second stage in the adder uses comparator logic to check whether the exponent is odd. If yes, mantissa is shifted only one place towards right and "000000000000001" is added to exponent to compensate. If exponent happens to be even, mantissa is shifted to right by two places and "000000000000010" is added to exponent to compensate. The combinational VHDL process compare the exponent's LSB with '0'. The comparator is left to the synthesis tool.

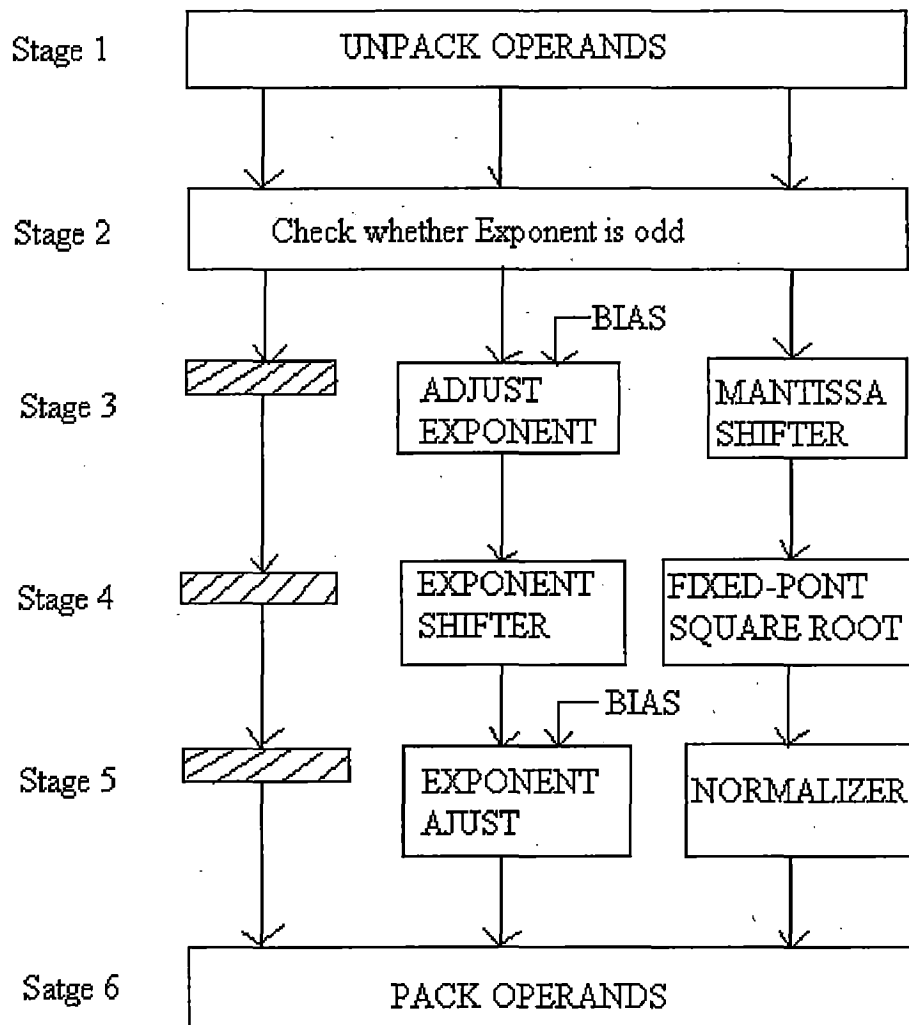


Figure 4.16 Square root unit Algorithm

Stage 3: Mantissa Shifter

As explained in stage 2, 1-bit or 2-bit right shifting on mantissa is performed depending on whether exponent is odd or even respectively. The maximum number of shifts required is two. The bias is subtracted from the exponent in this stage.

Stage 4: Mantissa Square root Stage

In this stage, the square root of mantissa is taken using non-restoring square root algorithm [21][22][23] which is explained in detail in next section. This algorithm does

not restore the remainder [21]. This stage consumes maximum number of clock cycles and so it is area of research in my algorithm.

Different fixed-point square root algorithms are studied to determine which gives the best performance. The different kinds of algorithms used for comparative study are

1. Newton-Raphson method
2. SRT-Redundant method
3. Non-Redundant method

All methods are implemented at the logic-level and the working of each algorithm is explained in Section 4.11.3. The sign bits are stored in delay registers.

Stage 5: Normalization Shift stage

After the square root, normalization is the next step. Since shifting has already been performed on mantissa before taking square root, normalization shall be required in case if MSB is non-zero. So, the first step is to identify the leading or first one in the result. Comparator logic is used here to find the first leading-one digit from the MSB. A counter maintains the number of comparisons made which is the equal to the number of shifts needed. I would like to make a point that this value can not be greater than two. The shift value is used to normalize the mantissa such that the leading-one in the mantissa resides in the most significant bit location. This stage also uses the shift value to adjust the exponent to the number of shifts required. The bias is also added to the exponent in this stage. Shifter in this stage is left to the synthesis tool.

Stage 6: Pack operands

Finally sign, exponent and mantissa are concatenated to form the 80-bits result and passed as an output from the FPU. The special condition flags are checked and if any of the flags are set high, then the result vary accordingly. The result is stored back in a 80-bit register.

4.11.3 Different types of fixed point square root algorithms and their comparative study

Following algorithms are studied and finally, Non-redundant method is used [23] and based on simulation results, Non-Redundant method is used because it provides lowest latency (66 clock cycles) and consumes least possible silicon space. Its algorithm is described in detail.

a) Newton-Raphson method

The Newton-Raphson method has been adopted in many applications. In order to calculate $Y = \sqrt{x}$, an appropriate value is calculated by iterations. For example, Newton-Raphson method can be used on

$$f(T) = 1/T^2 - x$$

to derive the iteration equation

$$T_{i+1} = T_i \times (3 - T_i^2 \times x)/2$$

where, T_i is an approximate value of $1/\sqrt{x}$.

After an n iterations, an approximate square root can be obtained by equation

$$Y = \sqrt{x} \approx T_n \times x$$

This method needs a ROM Table for generating T_0 . At each iteration, multiplication and addition/subtractions are needed.

Disadvantage:

Although the fast multipliers are available, this design is costly because the multipliers require large number of gate counts.

b) SRT-Redundant method

This method is based on the recursive relationship

$$X_{i+1} = 2x_i - 2Y_i y_{i+1} - y_{i+1}^2 2^{-(i+1)}$$

$$Y_{i+1} = Y_i + y_{i+1} 2^{-(i+1)}$$

where, x_i is the i th partial remainder,

Y_i is the i th partially developed square root with $Y_0 = 0$,

y_i is the i th square root bit,
and $y_i \in \{-1, 0, +1\}$.

The y_i is obtained by applying the digit selection method. In each iteration, there are four sub-computations:

1. One digit shift of x_i to produce $2x_i$.
2. Determination of y_{i+1} .
3. Formation of $F = -2Yiy_{i+1} - y_{i+1}^2 2^{-(i+1)}$.
4. Addition of F and $2x_i$ to produce x_{i+1} .

Disadvantage:

A CSA can be used to speed up the addition of F and $2x_i$, but F needs to be converted to the two's complemented represented before feeding to CSA. Moreover, the selection function is also complex.

c) Non-Redundant method

This method does not restore the remainder [23]. There is no need to do the F conversion and the calculation of $Y_i - 2^{-(i+1)}$ that appear in SRT method.

The radicand (in stack register) is in extended-double precision i.e. 80-bit format and the mantissa is of 64-bits:

$$\text{Mantissa, } D = D_{63}D_{62}D_{61}\dots D_1D_0.$$

For each pair of bits of the radicand, the integer part of the square root has one bit. Thus the integer part of square root for a 64-bit radicand has 32-bits:

$$Q = Q_{31}Q_{30}Q_{29}\dots Q_1Q_0$$

The remainder, $R = D - Q \times Q$ has 33 bits:

$$R = R_{16}R_{15}R_{14}\dots R_1R_0.$$

Reason of 33-bits in R:

$$D = Q \times Q + R < (Q + 1) \times (Q + 1).$$

$$\Rightarrow R < (Q + 1) \times (Q + 1) - Q \times Q$$

$$\Rightarrow R = 2 \times Q + 1$$

$$\text{i.e. } R \leq 2 \times Q$$

because the remainder R is an integer. It means that the remainder has at most one binary bit more than the square root.

Algorithm

Step 1: Set $q_{16} = 0$, $r_{16} = 0$ and then iterate from $k = 15$ to 0 .

Step 2: If $r_{k+1} \geq 0$, $r_k = r_{k+1}D_{2k+1}D_{2k} - q_{k+1}01$,
 Else $r_k = r_{k+1}D_{2k+1}D_{2k} + q_{k+1}11$.

Step 3: If $r_k \geq 0$, $q_k = q_{k+1}1$ (i.e. $Q_k = 1$),
 Else $q_k = q_{k+1}0$ (i.e. $Q_k = 0$),

Step 4: Repeat steps 2 and 3, until $k = 0$.
 If $r_0 < 0$, $r_0 = r_0 + q_01$.

Where, $q_k = Q_{31}Q_{30}Q_{29} \dots Q_k$ has $(31 - k)$ bits,
 r_k has $(17 - k)$ bits.

In this method, I do not need multiplications, instead shift operations do the necessary work.

4.12 Absolute unit

Absolute unit performs the conversion of floating point number to integer number. This unit has the latency of 3 clock cycles.

4.12.1 Algorithm

This section describes the algorithm implemented.

Stage 1: Unpacking Operands

The sign, exponent and mantissa bits for operand are latched in registers which are 1-bit, 15-bits, and 64-bits in length. The inputs are checked for special values:

Infinity, Not a Number and Zero and the appropriate flags are set which is passed on through all stages.

Stage 2: Conversion into integer

The Exponent and mantissa bits are retained as it is and passed on to the next stages. The sign bit is set to 0 to make the number positive.

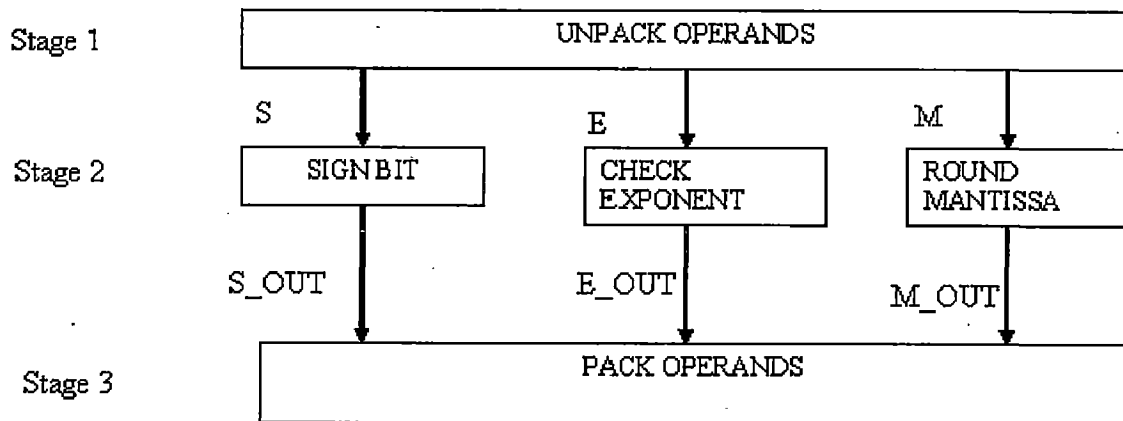


Figure 4.17 Absolute unit Algorithm

Stage 3: Packing Operands

Finally sign, exponent and mantissa are concatenated to form the 80-bit result and passed as an output from the FPU. The special condition flags are checked and if any of the flags are set high, then the result vary accordingly. The result is stored back in 80-bit register.

4.13 Exception Generation unit

This section describes the various conditions that cause a floating point exception to be generated by the FPU [13][14].

- **Stack Overflow Exception:** Occurs when Load instruction attempts to load a non-empty data register. A non-empty register is defined as a register containing a zero (tag value of 01), a valid value (tag value of 00), or a special value (tag value

of 10). When this exception occurs, its sets the bit 5 in the status register and FPU data register wraparound occurs and the new value of TOP is set to 7.

- **Stack Underflow Exception:** Occurs when Store instruction references an empty data register as a source operand, including attempting to write the contents of an empty register to memory or external world. An empty register has a tag value of 11. When this exception occurs, its sets the bit 6 in the status register and FPU data register wraparound occurs and the new value of TOP is set to 0.
- **Invalid Arithmetic Operand Exception:** Occurs when an arithmetic instruction attempts to operate on empty data registers. When this exception occurs, its sets the bit 0 in the status register and the set the output value to NaN.
- **Divide by Zero Exception:** Occurs when an instruction attempts to divide a finite non-zero operand by 0. When this exception occurs, its sets the bit 2 in the status register and the set the output value to infinite.
- **Numeric Overflow Exception:** Occurs whenever the rounded result of an arithmetic instruction exceeds the largest allowable finite value that will fit into the floating-point format of the destination operand (32 bit). When this exception occurs, its sets the bit 3 in the status register and the set the output value to infinite.
- **Numeric Underflow Exception:** Occurs whenever the rounded result of an arithmetic instruction is tiny; that is, less than the smallest possible normalized, finite value that will fit into the floating-point format of the destination operand. When this exception occurs, its sets the bit 4 in the status register and the set the output value to zero.

CHAPTER 5

DESIGNING WITH FPGAs

Field Programmable Gate Arrays are a relatively new class of integrated circuit; first introduced by the Xilinx company in 1985. Since that time, the FPGA market has expanded dramatically with many different competing designs developed by companies including, Actel, Advanced Micro Devices, Algotronix, Altera, Atmel, AT&T, Crosspoint Solutions, Cypress, Intel, Lattice, Motorola, QuickLogic, and Texas Instruments. A field-programmable gate array (FPGA) is kind of like a CPLD turned inside out[1][17].

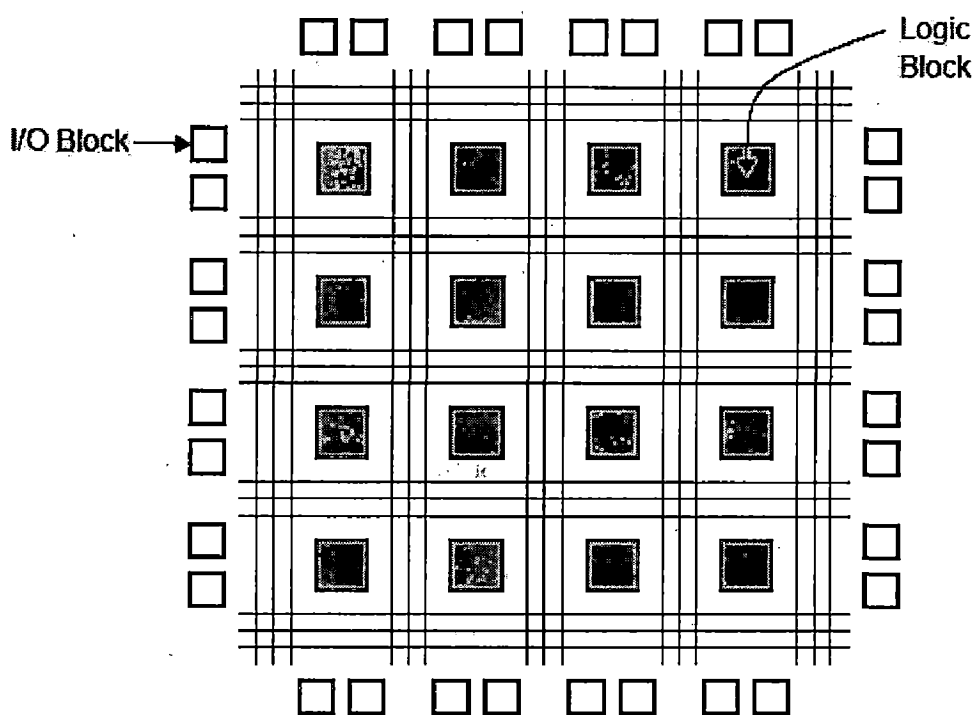


Figure 5.1 FPGA Architecture

As shown in Fig. 5.1, the logic is broken into large number of programmable logic blocks that are individually smaller than a PLD. They are distributed across the entire chip in a sea of programmable interconnections which can be configured by the

user at the point of application, & the entire array is surrounded by programmable I/O blocks. User programming specifies both the logic function of each block and the connections between the blocks. An FPGA's programmable logic block is less capable than a typical PLD, but an FPGA chip contains a lot more logic blocks than a CPLD of the same die size has PLDs.

5.1 Introduction to FPGAs

The FPGA is an integrated circuit that contains numerous (over 10,000) identical logic cells that can be viewed as standard components. Each logic cell can independently take on any one of a limited set of personalities. The individual cells are interconnected by a programmable interconnect (matrix of wires and programmable switches). A user's logic design is implemented by specifying the simple logic function for each cell and selectively closing the switches in the programmable interconnect matrix. The cell's combinatorial logic is physically implemented as a small look-up table memory (LUT) or as a set of multiplexers and gates. LUT devices tend to be a bit more flexible and provide more inputs per cell than multiplexer cells at the expense of propagation delay. The array of logic cells and interconnects form a fabric of basic building blocks for logic circuits (also named as Logic elements - LE). Complex designs are formed by combining these Logic elements to build the desired circuit.

Field Programmable means that the FPGA's function is defined by a user's program rather than by the manufacturer of the device. A typical integrated circuit performs a particular function defined at the time of manufacture. In contrast, a program written by someone other than the device manufacturer defines the FPGA's function. Depending on the particular device, the program is either 'burned' in permanently or semi-permanently as part of a board assembly process, or is loaded from an external memory each time the device is powered up. This user programmability gives the user access to complex integrated designs without the high engineering costs associated with application specific integrated circuits.

Figure 5.2 shows an example of a logic block consisting of a 3-LUT, and a flip-flop. An 8-to-1 multiplexer in a LUT is implemented using 2-to-1 multiplexers. Therefore, the propagation delay from inputs to the output is not the same for all the

inputs. Input IN 1 experiences the shortest propagation delay, because the signal passes through fewer multiplexers than signals IN 2 and IN 3. Since a LUT can implement any function of its input variables, inputs to the LUTs should be mapped in such a way that the signals on a critical path pass through as few multiplexers as possible. Logic blocks also include a flip-flop to allow the implementation of sequential logic. An additional multiplexer is used to select between the LUT and the flip-flop output. Logic blocks in modern FPGAs are usually more complex than the one presented here.

Each logic block can implement only small functions of several variables. Programmable interconnection, also called *routing*, is used to connect logic blocks into larger circuits performing the required functionality. Routing consists of wires that span one or more logic blocks. Connections between logic blocks and routing, I/O blocks and routing, and among wires themselves is programmable, which allows for the flexibility of circuit implementation. Routing is a very important aspect of FPGA devices, because it dominates the chip area and most of the circuit delay is due to the routing delays.

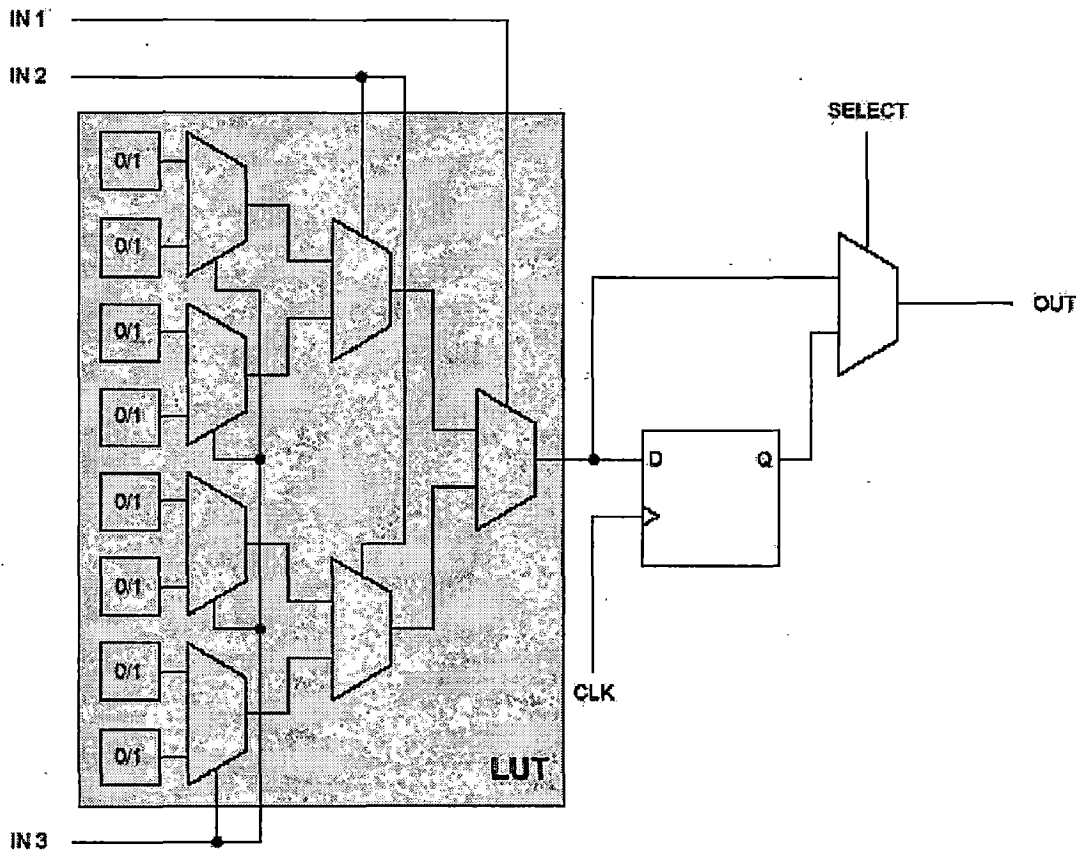


Fig 5.2 Simple Logic Block Structure

I/O blocks in an FPGA connect the internal logic to the outside pins. Depending on an actual device, most pins can be configured as either input, output, or bidirectional. Devices supporting more than one I/O standard allow configuration of different pins for different standards.

Programmability of FPGAs is commonly achieved using one of three technologies: SRAM cells, antifuses, and floating gate devices. Most devices use SRAM cells. The SRAM cells drive pass transistors, multiplexers, and tri-state buffers, which in turn control the configurable routing, logic and I/O blocks. Since the content of SRAM cells is lost when the device is not powered, the configuration needs to be reloaded into the device on each power-up. This is done using a configuration device that loads the configuration stored in some form of non-volatile memory.

Programmability of FPGAs comes at a price. Resources necessary for the programmability take up chip area and consume power. Therefore, circuits implemented in FPGAs take up more area and consume more power than in equivalent ASIC implementations. Furthermore, since the routing in FPGAs is achieved using programmable switches, as opposed to metal wires in ASICs, circuit delays in FPGAs are higher. Because of that, care has to be taken to exploit the resources in an FPGA efficiently. Circuit speed is important for high-throughput applications like Digital Signal Processing (DSP), while power is important for embedded applications. CAD tools are used by the designer to meet these requirements.

5.2 Basic Architectures

FPGAs are commercially available in many different architectures and organizations. Although each company's offerings have unique characteristics, FPGA architectures can be generically classified into one of four categories:

- A Symmetrical Array.
- B Row Based.
- C Hierarchical PLD.
- D Sea of Gates.

Figure 5.3 illustrates this classification based on the general internal organization of the design.

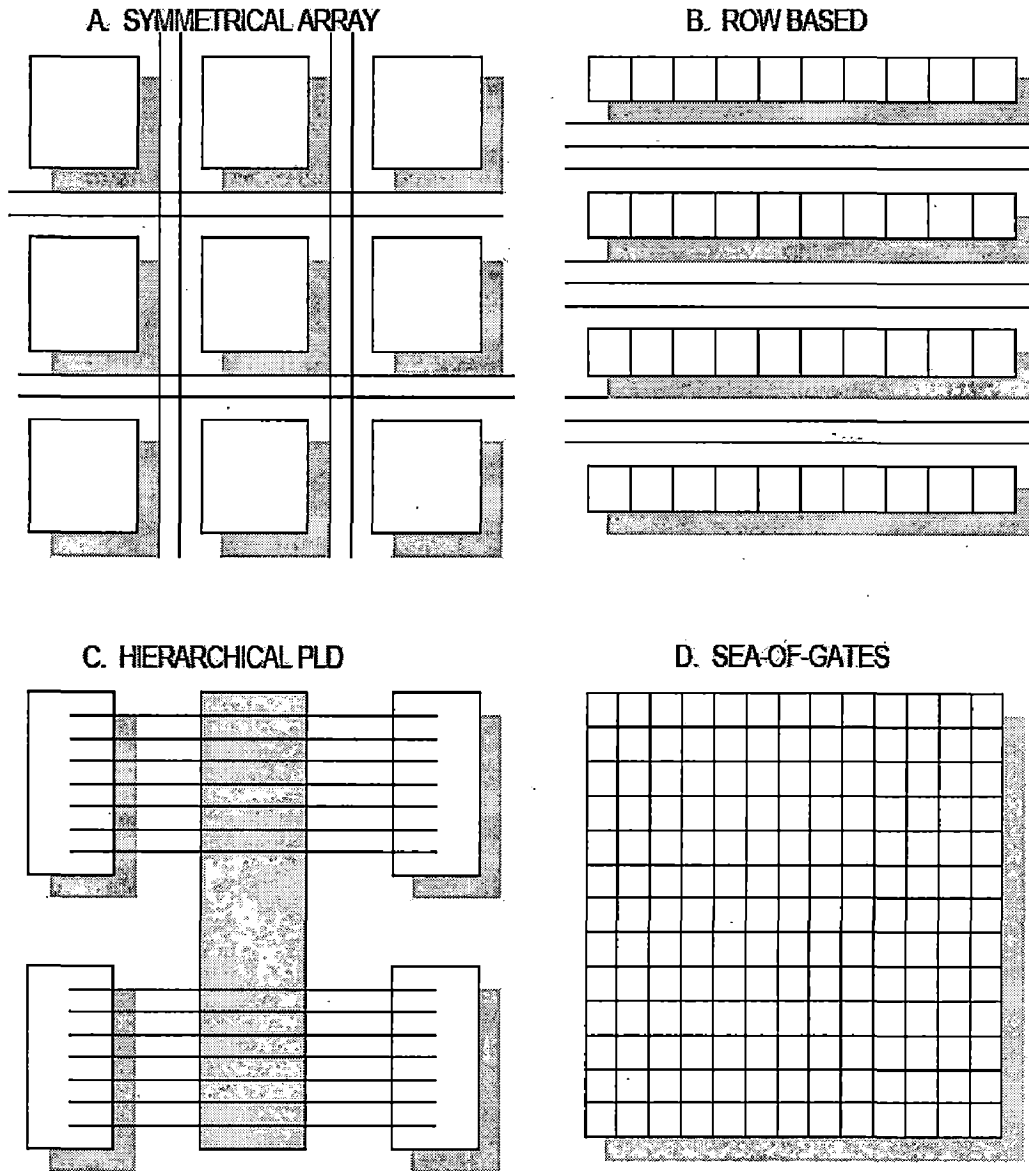


Figure 5.3 The Four FPGA Architectural Classes

The target FPGA kit in the dissertation is Xilinx's Virtex II Pro whose architecture is similar to that of Symmetrical Array (see figure5.3 (a)). All the synthesis results are generated for Virtex II Pro only.

5.3 Programming with FPGAs

Although early PLD and FPGA designs were generated largely by hand, access to today's complex programmable logic devices requires the use of an integrated Computer-Aided Design (CAD) system.

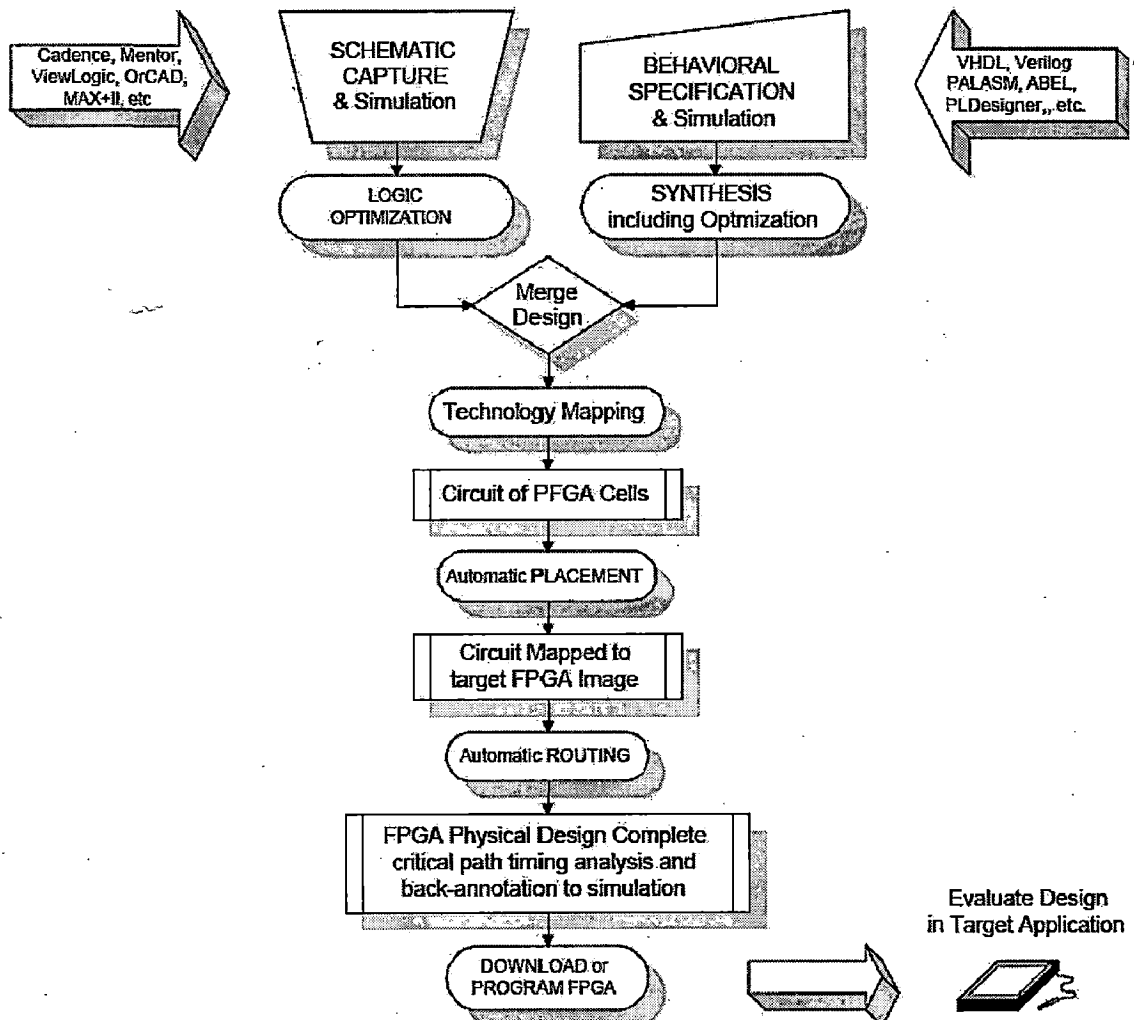


Figure 5.4 Typical CAD system design flow for FPGAs

Figure 5.4 illustrates the typical sequence of operations needed to go from concept to programmed chip. Both commercial CAD tool vendors and FPGA companies offer appropriate tools. For example, traditional Electronic Design Automation (EDA) vendors such as Cadence, Mentor Graphics, Synopsys, and ViewLogic all offer tools to

support FPGA design. These tools are typically used for the front-end design entry and simulation operations and provide the necessary interfaces to vendor-specific back-end tools for chip placement and routing.

Examples of vendor specific tools are the Xilinx XST system and the Altera Quartus II software. It is worth noting that Altera's Quartus II software supports the entire design flow illustrated in Figure 5.4 on either PC or workstation platforms. I have used Aldec' Active HDL 6.1 and Mentor's Modelsim for simulation and Xilinx ISE 7.1 for Synthesize the design. A detailed discussion of available FPGA CAD tools is outside the scope of this chapter. Rather, the following discussion is meant to be indicative of the general operations and steps required in FPGA design. Where appropriate, examples are taken from the Xilinx and Altera CAD design flows to illustrate the generic operations.

The starting point in any logic or digital system design is a set of architectural or behavioral specifications. Traditionally, a designer uses schematic capture tools for graphical entry of a logic design which has been manually generated to meet the architectural or behavioral specifications. The upper left hand arrow in Figure 5.4 identifies some of the commercial CAD tools available for FPGA schematic capture. One of the more significant recent innovations in the EDA industry is the development of tools which allow the designer to move from the gate level to the behavioral level for design entry. A behavioral design specification is created using a Hardware Description Language (HDL) [19][20], and then a synthesis tool automatically compiles the gate level schematic or netlist from the behavioral description. The upper right hand arrow in Figure 5.4 indicates some of the HDLs currently being used for FPGA behavioral modeling.

Options for behavioral description of designs include the VHSIC Hardware Description Language (VHDL), the Verilog hardware description language, timing diagrams, logic state diagrams, and PLD description languages such as ABEL. As an example of how pervasive the behavioral design style has become, the PC-based Xilinx ISE 7.1 software provides multiple options for behavioral design entry. In addition to traditional schematic capture it will accept VHDL, text design description in the Xilinx Hardware Description Language (including truth tables and Boolean expressions), and Timing Diagrams which describe the desired input and output waveforms. Whichever

behavioral design entry method is chosen, the design system provides logic synthesis, which automatically creates gate-level schematics.

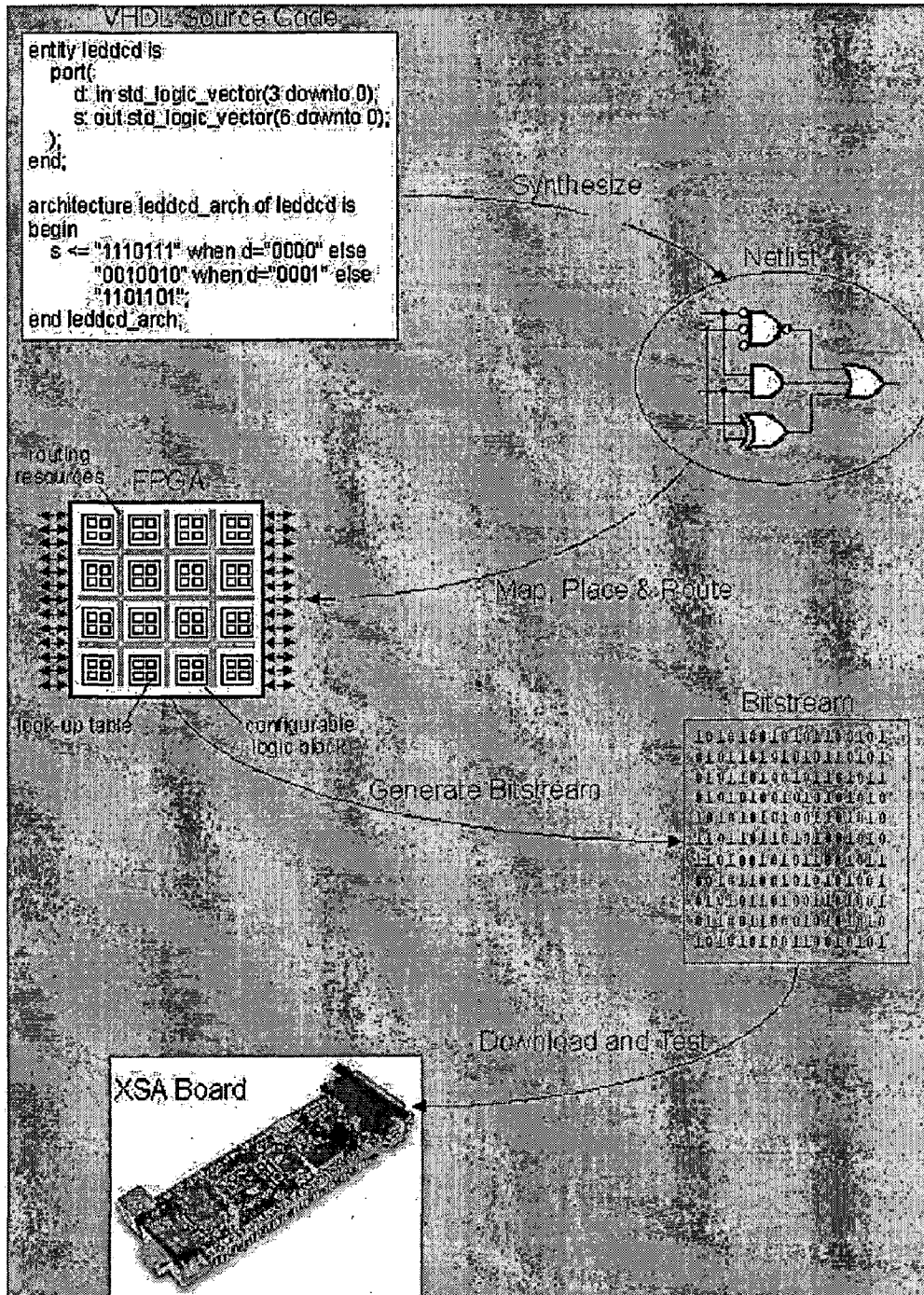


Figure 5.5 Designing with FPGA

No matter what method is used for initial design entry, the next step in FPGA design is to translate the entire design into a standard form which can be processed by a logic optimization tool. The goal of logic optimization is to perform minimization of the Boolean expressions and eliminate redundancy, thus minimizing the area of the final circuit. The tool may also be constrained to maximize speed at the expense of area by limiting the number of logic levels between clocked registers. This optimization process is usually merged with the logic synthesis step when behavioral design entry is employed. Simulation is performed both before and after the logic optimization steps to verify that the design meets the original system requirements for functionality and timing. The next step is to convert the generic gate level design into one which uses the FPGA circuit building blocks of the target technology.

Let me take a concrete example, the Xilinx XST design system flow is used (in dissertation) to illustrate the steps needed to go from logic design to programmed FPGA. In the Xilinx design flow, the native format of the logic design (Aldec's Active HDL, Modelsim etc.) must first be translated into the Xilinx Netlist Format (XNF) which is understood by the Xilinx tools. Next, the XNF circuit description must be mapped into Xilinx Configurable Logic Blocks (CLBs). This is the technology mapping step referred to in Figure 5.4. Xilinx calls this step "partitioning", and the XST tools also attempt to optimize the circuit during this step. For example, circuitry associated with unused logic block inputs or outputs is eliminated from the design. In addition, the partitioning program attempts to minimize either the total number of CLBs used or the number of logic stages in the critical delay path.

The next step is to place and route the design on the selected chip image. The XST system allows manual and/or automatic placement and routing. In the automatic placement operation, each CLB generated during the "partitioning" step is assigned to a physical location on the chip. Xilinx uses a Simulated Annealing algorithm which starts with a random placement, and then goes through a series of improvement passes. This program can be run multiple times with different starting random seeds in an attempt to generate a more optimal placement. Following placement, interconnections between the CLBs must be routed using the available interconnect segments and switch matrix elements. XST uses an automatic Maze Routing Algorithm to perform this operation.

With the physical placement and routing completed, exact timing values can now be used to determine chip performance. The XST tools provide a critical path timing analyzer which provides delay information on the longest through shortest paths through the chip. In addition, the physical layout timing information can also be back-annotated to the schematics to get more accurate functional simulation results. The final step in the Xilinx or Altera design flow is the creation of the BIT file which contains the binary programming data needed to configure the SRAM bits of the target chip. This file is then downloaded to configure the chip for final functional and timing tests of the programmed chip.

CHAPTER 6***EXPERIMENTAL RESULTS AND VERIFICATION***

This chapter presents the experiments conducted to determine the performance of the FPU:

6.1 Introduction to experimental approaches

I assess the FPU performance by considering each component in turn that is, the floating point adder, floating point multiplier, floating point divider, floating-point square root and absolute. My objective was to design the efficient FPU with the least possible latency and silicon area. The design was targeted on a Xilinx Virtex II Pro FPGA whose synthesis results are presented in Appendix C.

6.1.1 Design Environment

The software and hardware design environment is presented in this section. It gives information about the development tools used in this dissertation.

a) Software Environment:

- Operating System : Windows XP-Pro
- Processor : Pentium 4
- RAM : 256 MB
- Processor Speed : 2.40 GHz.
- HDL used : VHDL
- Simulation Tool : Aldec Active HDL 6.1

b) Hardware Environment:

- Processor : Pentium 4
- RAM : 256 MB
- Processor Speed : 2.40 GHz.

- Development Kit : Xilinx
- FPGA Device Family : Xilinx Virtex II Pro
- Package : xc2vp100
- Speed Grade : -5
- Top-level Module Type : HDL
- Synthesis Tool : Xilinx ISE 7.1i (using VHDL)
- Simulator : Aldec Active HDL 6.1 and
Modelsim XE 5.7c
- Generated Simulation Language : VHDL

6.1.2 FPGA Design Flow

Since the goal of this dissertation is to create a full custom processor design in FPGA, for this reason the implementation of embedded processor requires FPGA design flow steps to be followed. Figure 6.1 shows a standard design flow for a FPGA design:

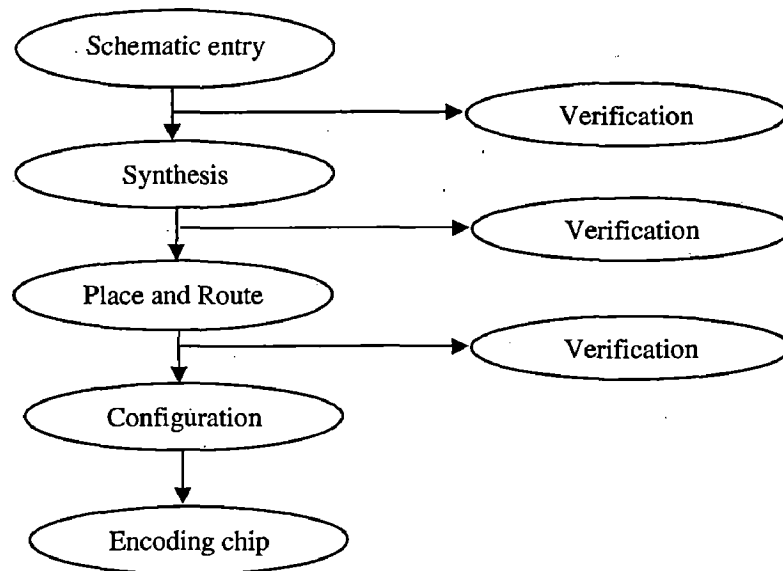


Figure 6.1 FPGA Design Flow

Schematic entry

The design is entered into a synthesis design system using a hardware description language. The language used here is VHDL.

Synthesis

A netlist is generated using the VHDL code and a logic synthesis tool (Xilinx ISE 7.1)

Place and Route

The place process decides the best location of the cells in a block based on the logic and desired performance. The route process makes the connections between the cells and the blocks. Automatic place and route is done by the synthesis tool after generating netlist.

Configuration

This is done by loading the configuration data into the internal memory. Synthesis tool generates a bit stream file after placing and routing, which is then downloaded in FPGA. I used Xilinx's JTAG cable to load my design in the FPGA.

Verification

At each step of the design process, I verified my architecture using software simulation. Initially I used Aldec Active HDL 6.1 software package for simulating my VHDL code.

6.2 Simulation results

This section shows the simulation results of each kind of hardware module implemented in the design.

6.2.1 Data registers

Signals `data_int_in` and `data_fract_in` are the input data's integer and fractional part. Signals `data_int_out` and `data_fract_out` are the output data's integer and fractional part. Bit No. 13, 12, 11 of status register indicates TOP field of stack. In address field (opcode), 00 is the opcode for Load instruction and 08 is for Store instruction. Signal `cs_main` is Chip select.

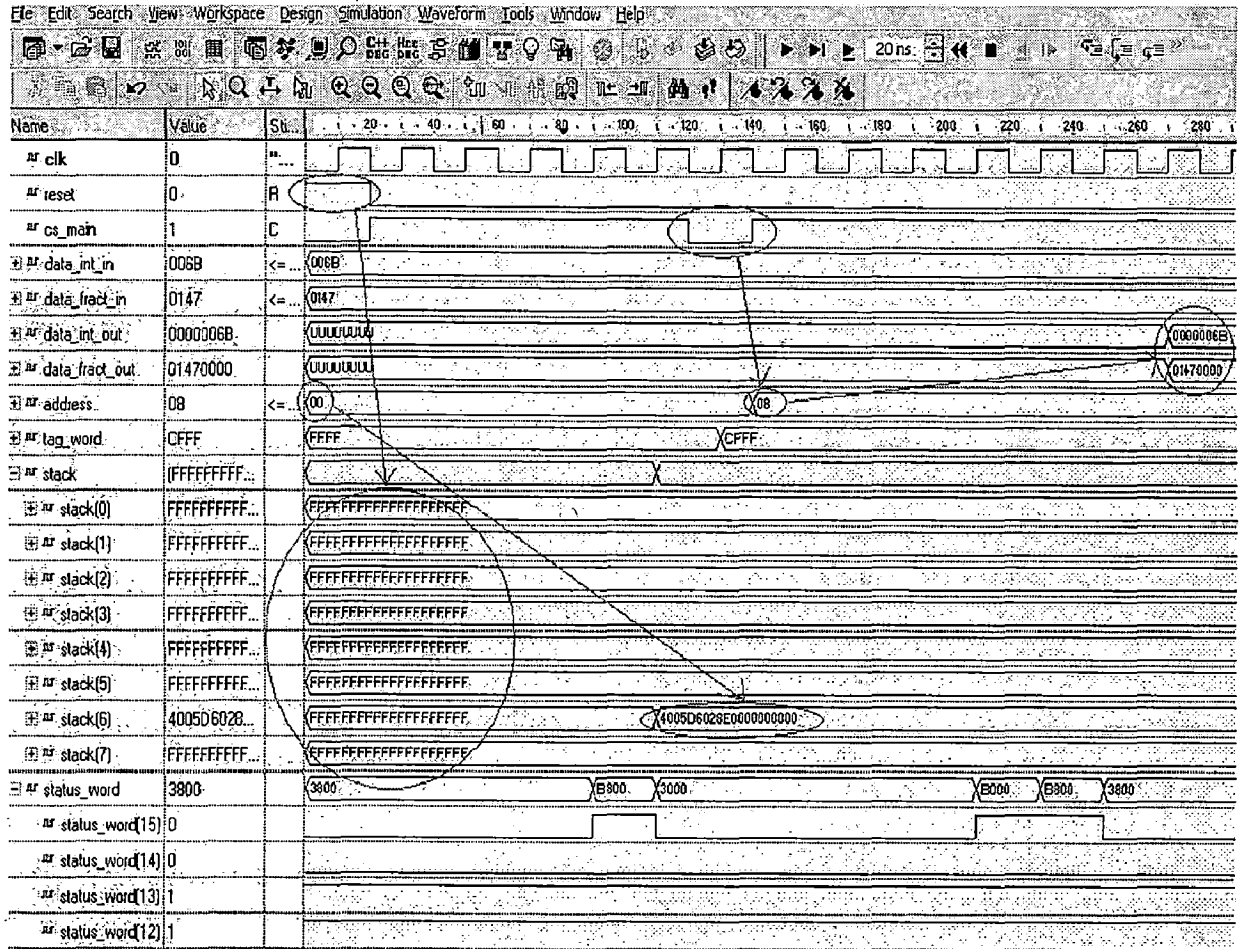


Figure 6.2 Simualtion Waveform for Stack operation

On reset, data registers contains FFFFFFFFH, which marks data registers as empty. When chip select is active, the decoder decodes the opcode (address field); the execution unit executes the decoded instruction (i.e. FLOAD). FLOAD instruction decrements the top by one and loads the input value into the stack (i.e. at 6th location). Here, the input data (data_int_in and data_fract_in) in hex format is automatically converted into single precision format and then to the extended double precision format as shown in Figure 6.2.

To execute the next instruction chip select should be deactivated after the busy signal goes low and reactivated in the following clock pulse.

When chip select is reactivated, decoder decodes the fresh opcode; the execution unit executes the decoded instruction (i.e. FSTORE). FSTORE instruction outputs the top of stack value and increments the top by one. Here, stack content in the extended-double

precision format is automatically converted into single precision and then to the hex format as shown in the Figure 6.2

6.2.2 Control Register

As shown in Figure 6.3, on reset, the control word is set to 0C00 which marks all floating-point exception unmasked and sets the rounding to nearest. The content of control word alters after FLDCW instruction loads it.

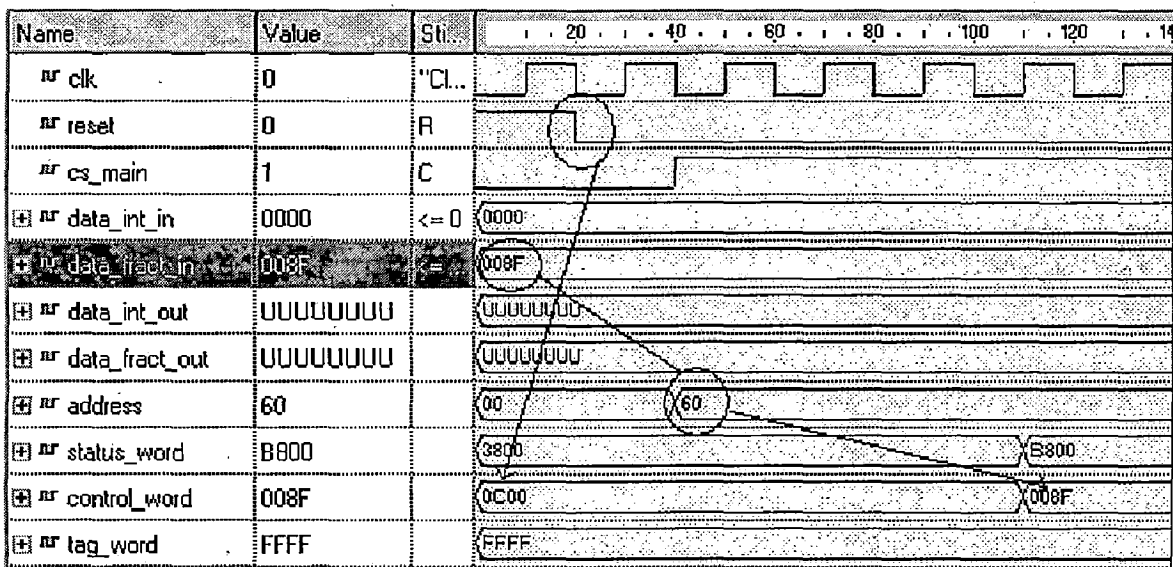


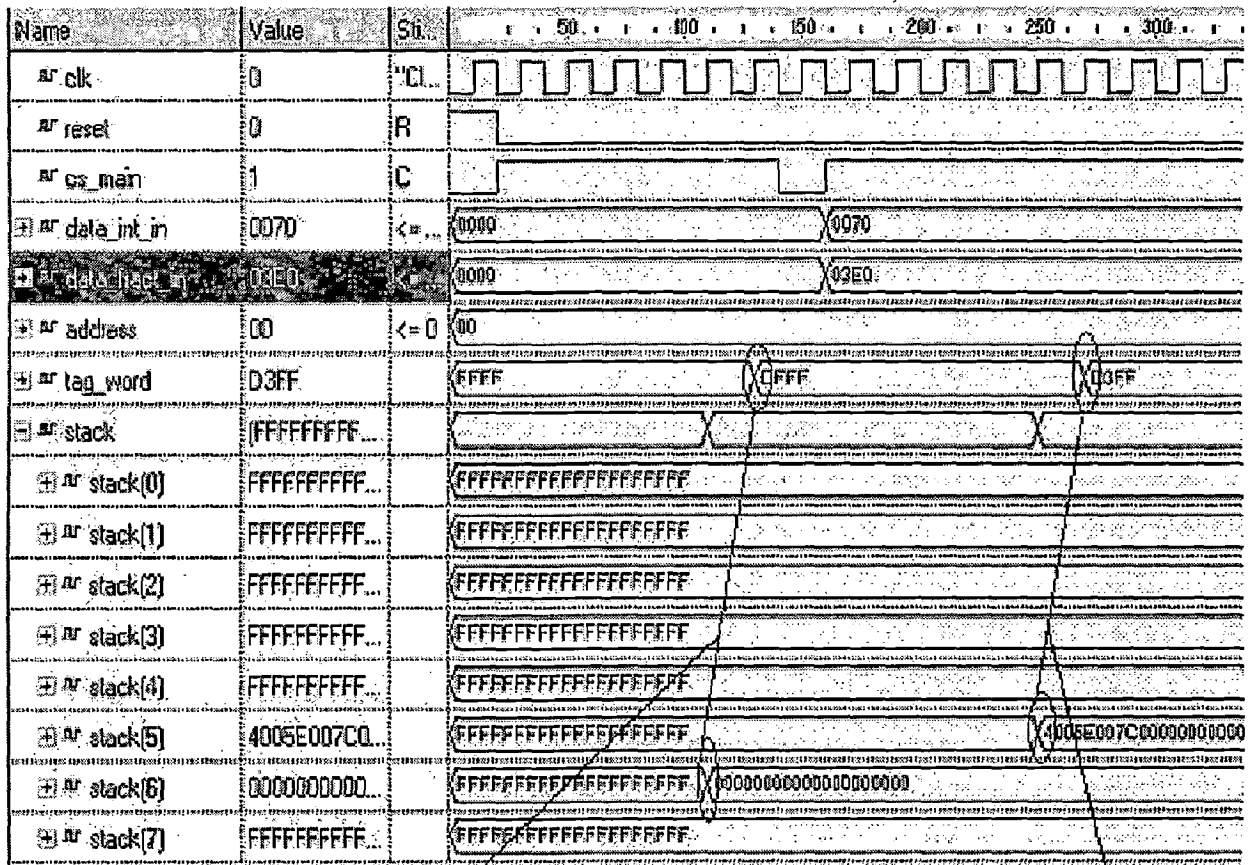
Figure 6.3 Simulation Waveform for Control word

6.2.3 Tag register

On reset, the tag word is set to FFFFH (i.e. each tag is set to 11) which marks all FPU data registers are empty.

As shown in Figure 6.4, loading a valid number in the data register stack changes the corresponding tag value to 00. Similarly, loading a zero number in the data register stack changes the tag value to 01 and loading an infinite or invalid number on the FPU data register stack changes the tag value to 10.

The tag word content D3FF (i.e. 110100111111111) indicates, the R6 contains zero number and R5 contains a valid number and rest of the data registers are empty.



Loading a zero in R6 changed corresponding tag to 01

Loading a valid number in R5 changed corresponding tag to 00

Figure 6.4 Simulation Waveform for Tag word

6.2.4 Precision Converter

The opa represents single precision 32-bit input and opa_80 represents 80-bit extended-double precision output.

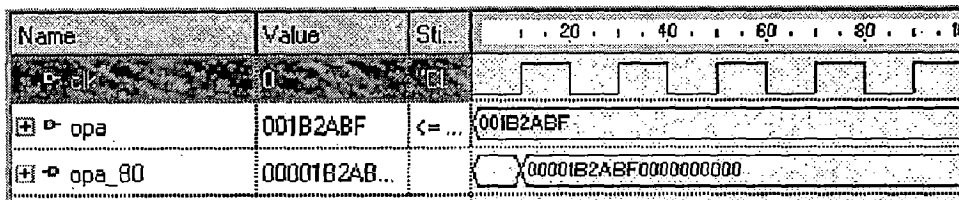


Figure 6.5 Simulation Waveform of Single Precision to Extended-Double Precision

The result_80 represents 80-bit extended double precision input and result_32 represents single precision 32-bit output, rmode represents rounding mode.

Name	Value	Sti...	1	20	40	60	80	100	
clk	0	"Cl...	[Timing diagram showing clock pulses]						
result_80	4007FC0000...	<= 0	[Hexadecimal value: 4007FC000000000000000000]						
result_32	43FC0000		[Hexadecimal value: 43FC0000]						
rmode	3		[Hexadecimal value: 3]						
overflow	0		[Signal line]						
underflow	0		[Signal line]						

Figure 6.6 Simulation Waveform of Extended-Double Precision to Single Precision

6.2.5 Addition/Subtraction Unit

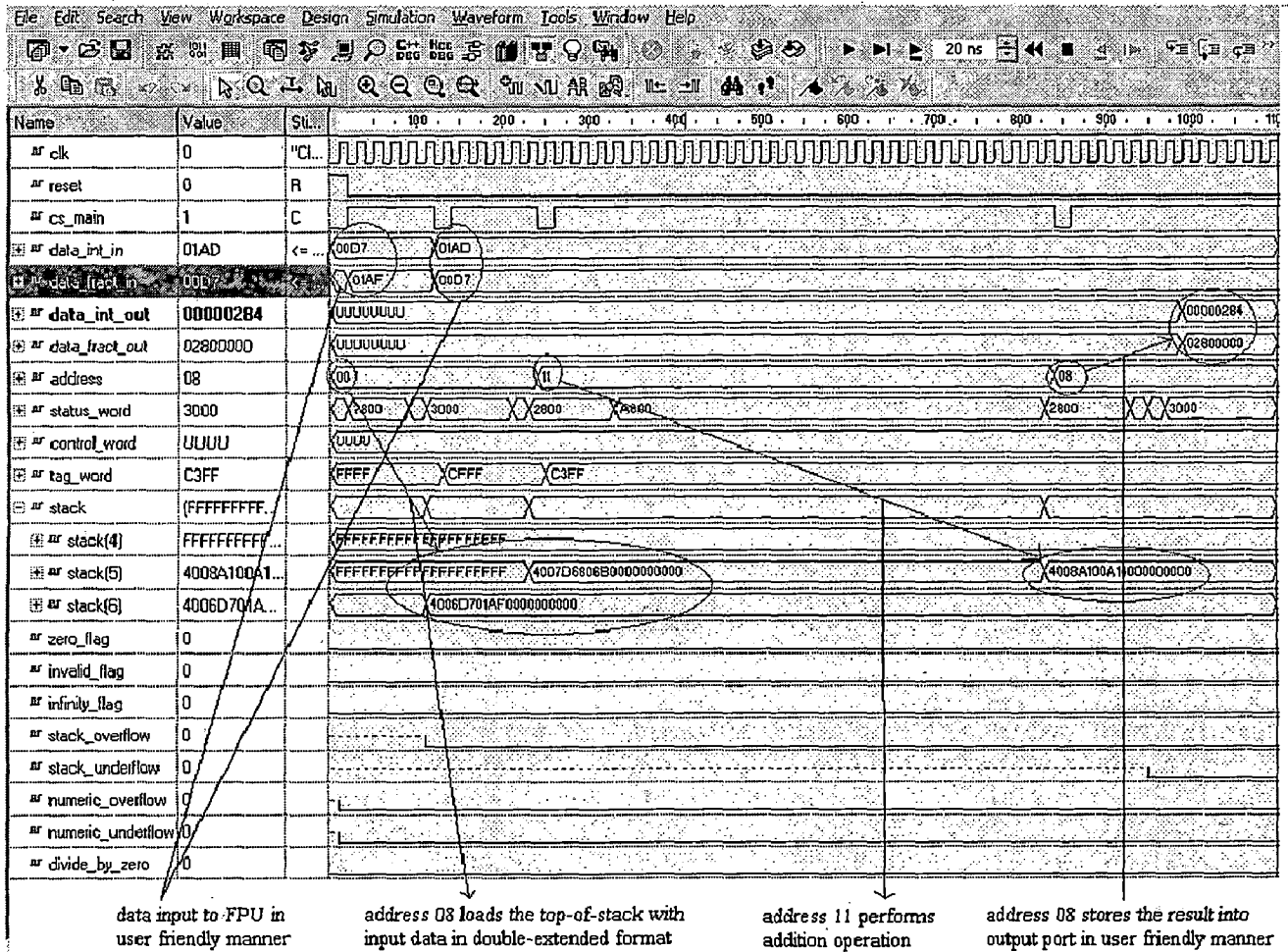


Figure 6.7 Simulation Waveform of Addition

The Floating point Addition/Subtraction unit accepts the data in double extended precision format only. The operands can be available in any of the eight data registers or stack. All addressing of the data registers is relative to the register on the top of the stack. The data contained by the top-of-stack is taken as operand-A while the operand-B is contained in the register whose effective address is given by “top-of-stack + offset given in instruction”. Addition/Subtraction operations keep the 80-bit result in the register as indicated by the top-of-stack. Please note that the TOP (stack TOP) field is not modified in this process and as a result, operand-A will be lost.

Consider the following example:

```
FADD ST (5);
```

Let the top-of-stack is fourth register (ST(5)) which is containing operand-A and operand-B is contained in ST(5). The result of addition operation is stored back in ST(4). Simulation waveforms are shown above.

6.2.6 Multiplication Unit

The Floating point Multiplication unit accepts the data in double extended precision format only. The operands can be available in any of the eight data registers or stack. All addressing of the data registers is relative to the register on the top of the stack. The data contained by the top-of-stack is taken as operand-A while the operand-B is contained in the register whose effective address is given by “top-of-stack + offset given in instruction”. Multiplication operations keep the 80-bit result in the register as indicated by the top-of-stack. Please note that the TOP (stack TOP) field is modified in this process and as a result, operand-A will be lost.

Consider the following example:

```
FMUL ST (5);
```


Let the top-of-stack is fourth register which is containing operand-A and operand-B is contained in ST(5). The result of Multiplication operation is stored back in ST(4). Simulation waveforms are shown in figure 6.8.

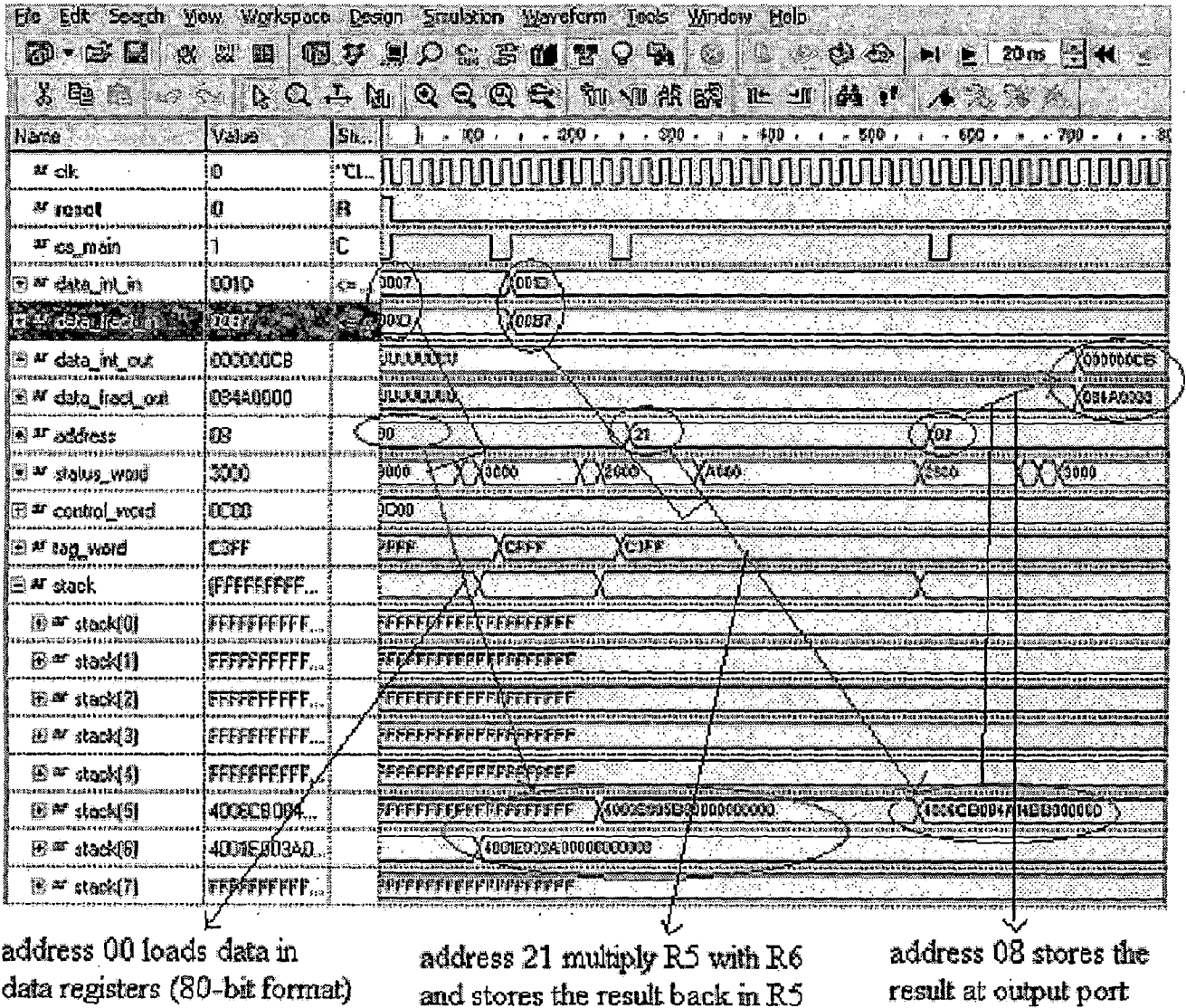
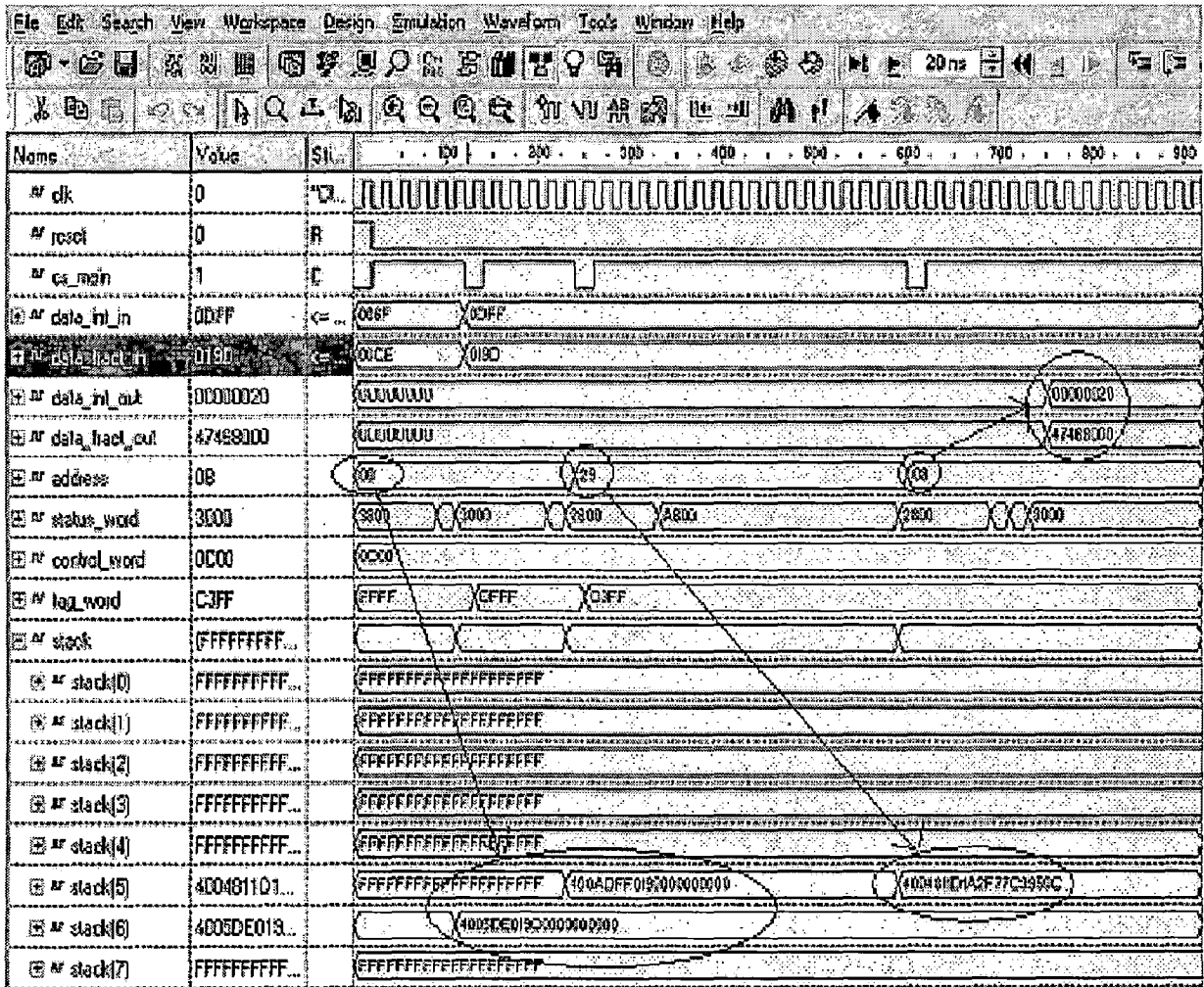


Figure 6.8 Simulation Waveform for Multiplication

6.2.7 Division Unit

The Floating point Division unit accepts the data in double extended precision format only. The operands can be available in any of the eight data registers or stack. All

addressing of the data registers is relative to the register on the top of the stack. The data contained by the top-of-stack is taken as operand-A or dividend while the operand-B or divisor is contained in the register whose effective address is given by “top-of-stack + offset given in instruction”. Division operations keep the 80-bit result in the register as indicated by the top-of-stack. Please note that the TOP (stack TOP) field is modified in this process and as a result, operand-A will be lost.



The explanation to above waveforms is similar to previous ones

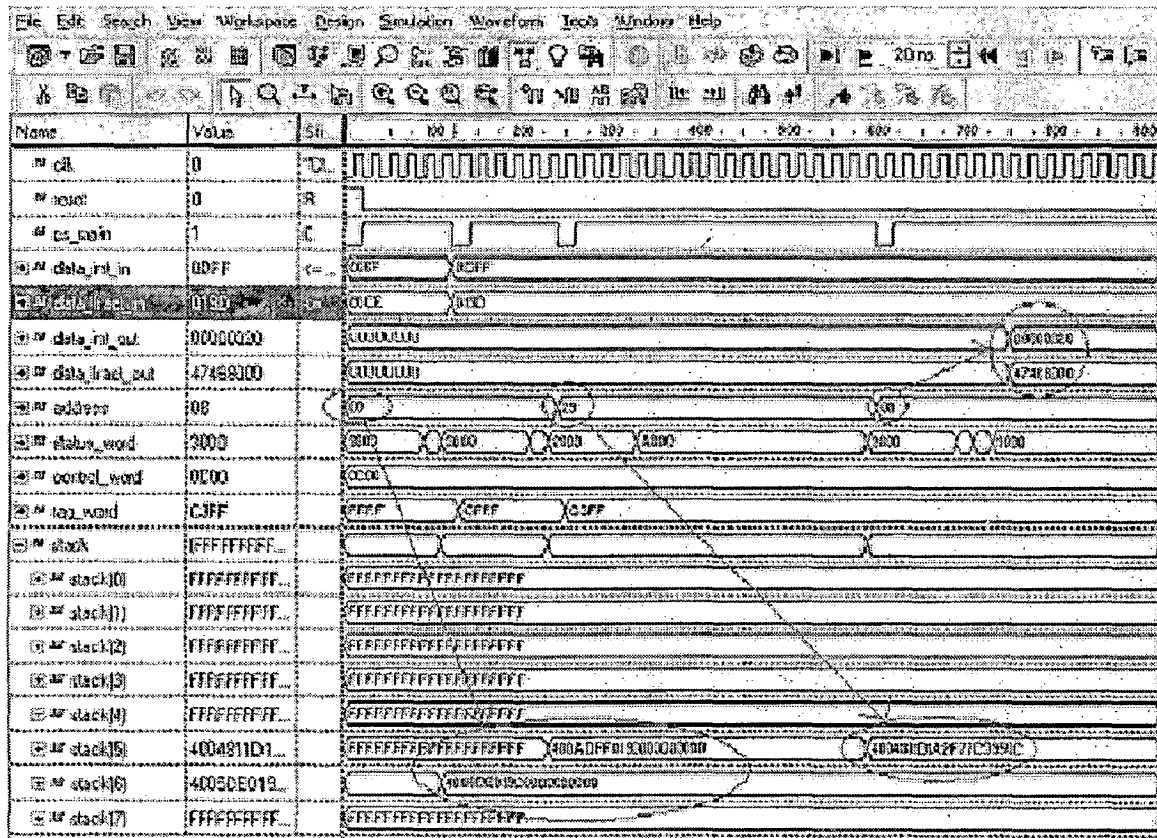
Figure 6.9 Simulation Waveform of Division

Consider the following example:

FDIV ST (5);

Let the top-of-stack is fourth register which is containing operand-A or dividend and operand-B or divisor is contained in ST(5). The result of Division operation is stored back in ST(4). Simulation waveforms are shown in figure 6.9.

6.2.8 Square Root Unit



address 00 loads data in data register (80 bit format) Address 38 takes square root of R5 and stores the result into R5 address 08 stores the result on the output port

Figure 6.10 Simulation Waveform of Square Root

Stack(5) contains the number whose square root is to be taken. The opcode for square root is 38. When this instruction is executed, square root of top-of-stack i.e. ST(5) is taken and the result is outputted through address 08. Since square root operation consumes 66 clock cycles, above waveform is modified so that it can be contained on this page.

6.2.10 Absolute unit

Address 30 takes the absolute of a number and stores the result in the same register. The following waveform depicts the operation.

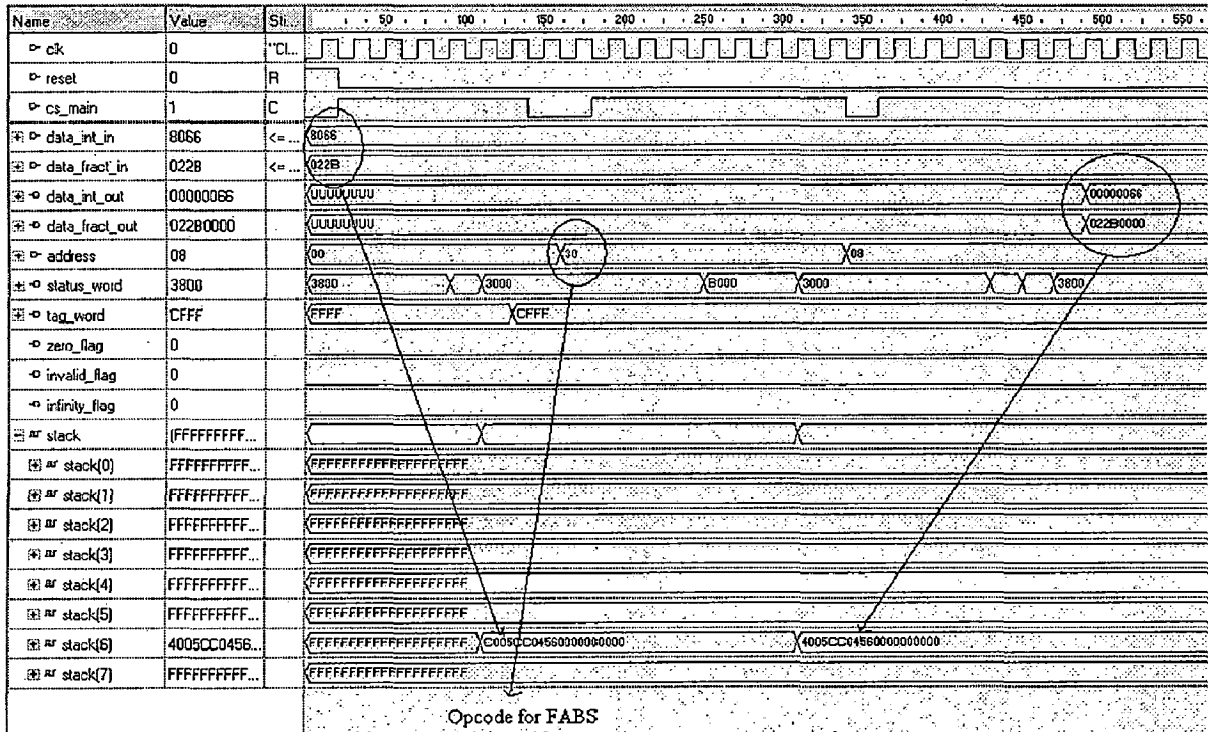


Figure 6.11 Simulation Waveform for Absolute Unit

6.2.11 Exception Generation Unit

The waveform for each exception is as shown in figure 6.12. On reset, the TOP field of the stack is set to 7; hence an attempt to read (FSTORE instruction opcode 08) will lead to stack underflow exception. As shown in figure 6.12 (b) when stack is full i.e. when TOP field is set to 0, an attempt to write (FLOAD instruction opcode 00) will lead to stack overflow exception. Similarly, Figure 6.12 (c) shows the divide by zero exception, Figure 6.12 (a) shows numeric overflow exception.

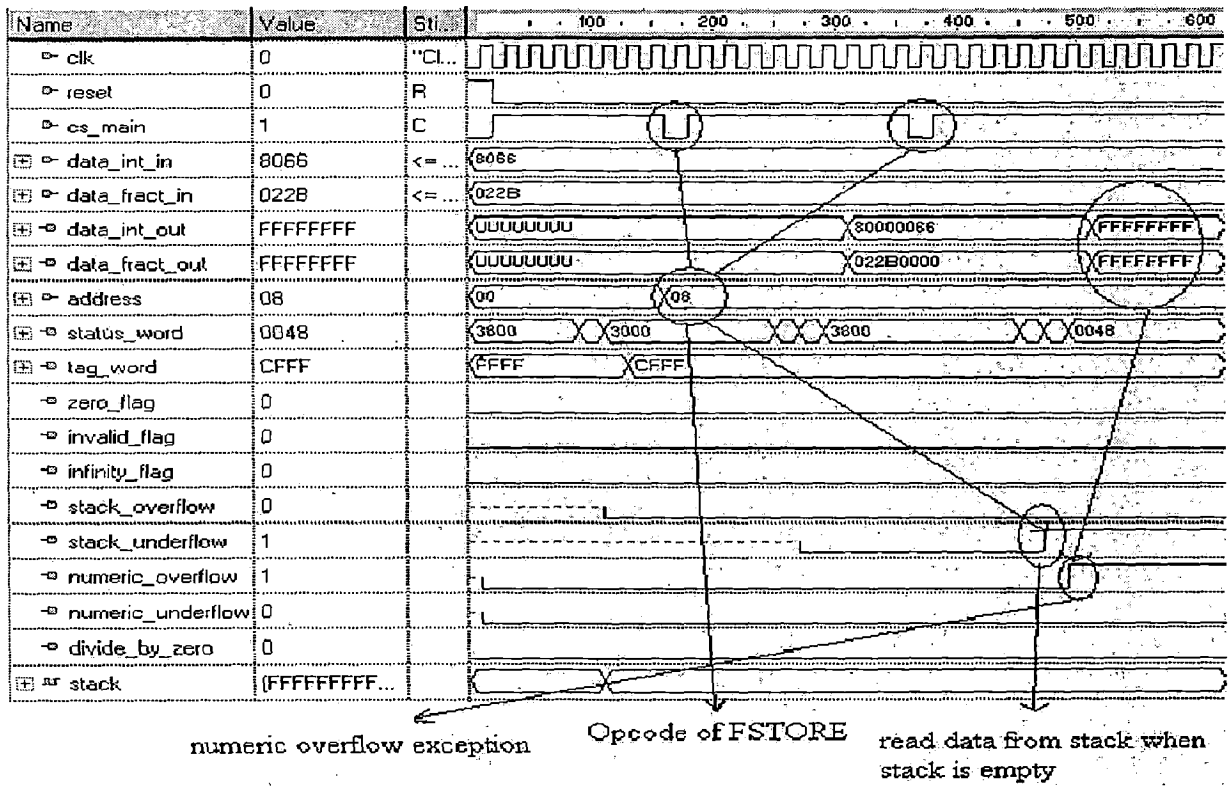


Figure 6.12 (a) Simulated Waveform for Exception Generation

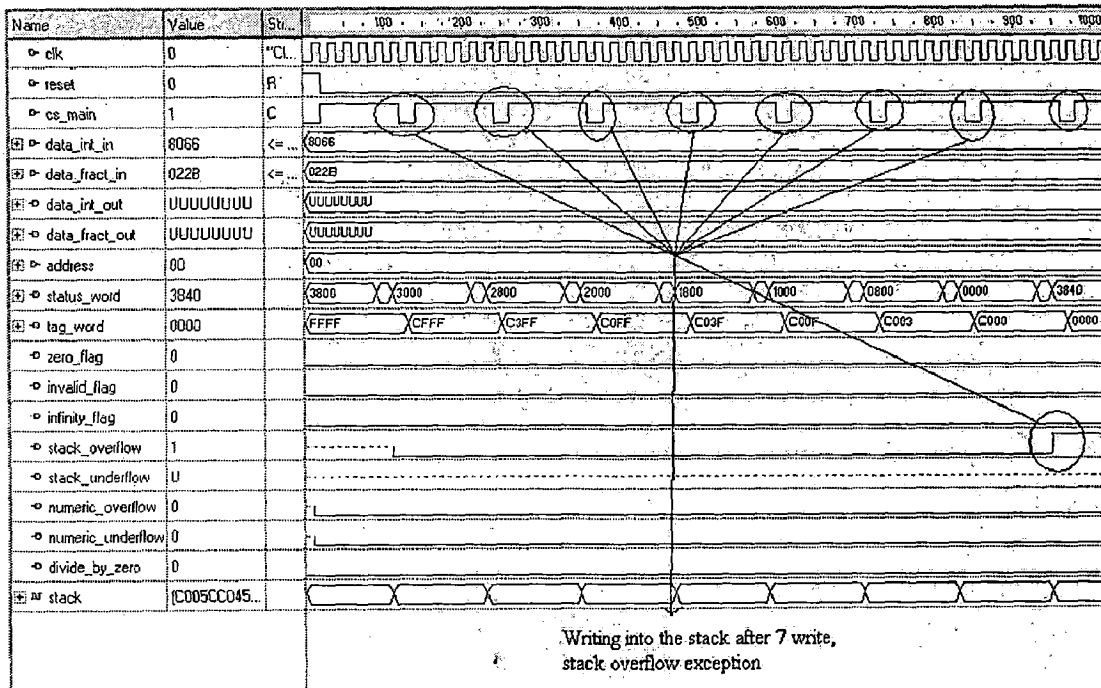


Figure 6.12 (b) Simulated Waveform for Exception Generation

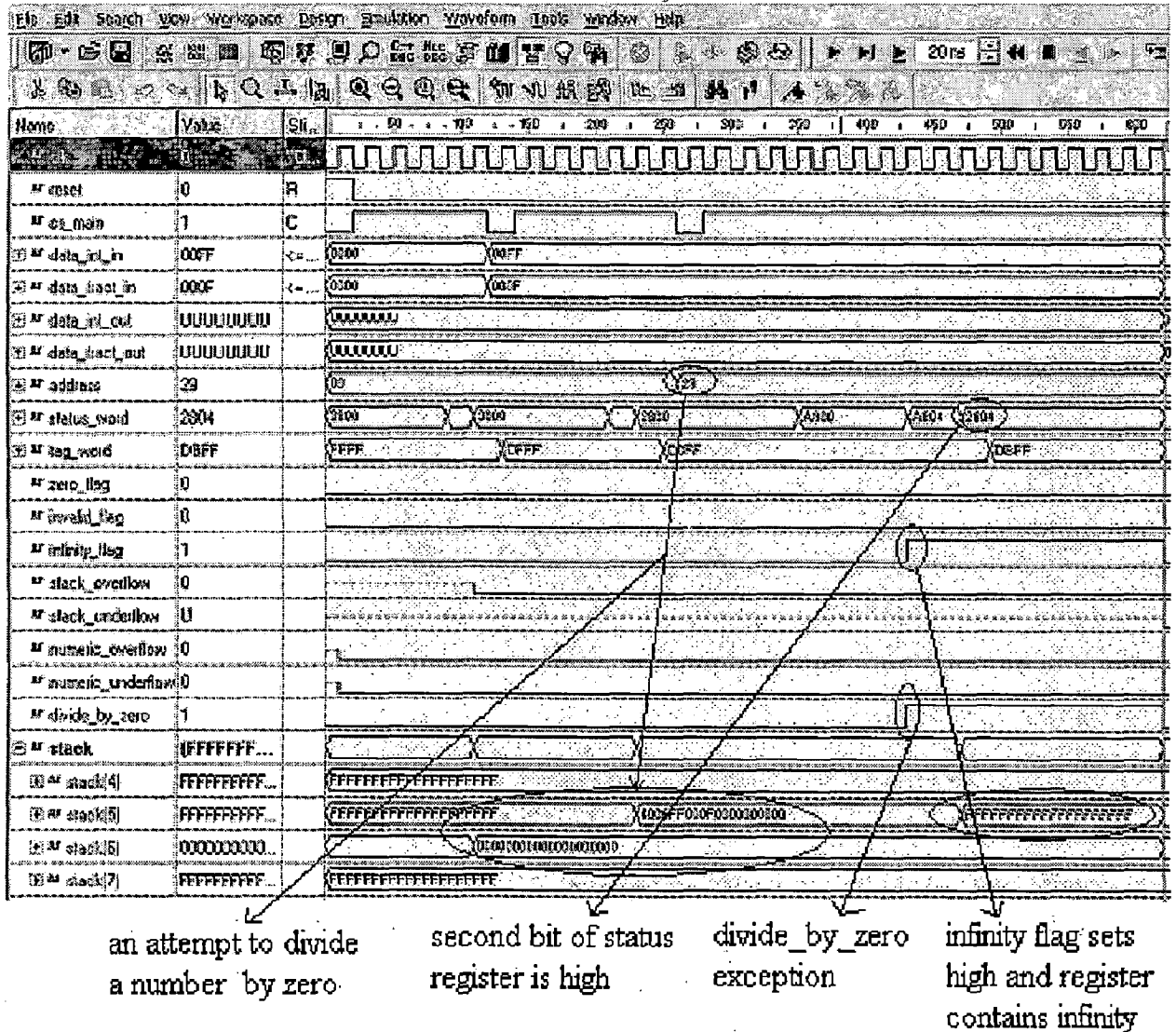


Figure 6.12 (c) Simulated Waveform for Exception Generation

6.2.11 Turbine Efficiency Measurement

The formula used for unit efficiency measurement (see chapter 2) is

$$\text{Unit Efficiency, } \eta_u = \frac{\text{Generator_Output, } P_e}{\text{Hydraulic_Input, } P_i} \times 100\%$$

where, Hydraulic input, $P_i = gHQ$

g is the acceleration due to gravity, m/s^2 ,

ρ is the density of water, kg/m^3 ,

$H = H_1 - H_2$ is the net water head, m,

H_1 is water head at inlet, m,

H_2 is water head at outlet, m,

Q is the water discharge through the turbine, m³/s

Description of Unit Efficiency Measurement Waveform:

Signals `data_int_in` and `data_fract_in` are the input data's integer and fractional part. Signals `data_int_out` and `data_fract_out` are the output data's integer and fractional part. Bit No. 13, 12, 11 of status register indicates TOP field of stack. Bit 15 of status register is busy bit as described in chapter 4. In address field (opcode), 00 is the opcode for Load instruction and 08 is for Store instruction. Signal `cs_main` is Chip select.

On reset, FPU data registers contains FFFFFFFFH, which marks data registers as empty. When chip select is active, the decoder decodes the opcode (address field); the execution unit executes the decoded instruction (i.e. FLOAD). FLOAD instruction decrements the top by one and loads the input value (1000.0 (=)) into the stack (i.e. at 6th location). Here, the input data (`data_int_in` and `data_fract_in`) in hex format is automatically converted into single precision format and then to the extended double precision format as shown in figure 6.13 (a).

To execute the next instruction chip select should be deactivated after the busy signal goes low and reactivated in the following clock pulse.

Similarly, next number (9.806 (=g) in hex format) is loaded into the stack. Then the address in address field is changed to 21 (opcode of FMUL ST(1)). This instruction multiplies top of stack (R5 or 5th location of FPP data register) with content of top + 1 of stack and stores the result back into top of stack (R5) as shown in figure 6.13 (b).

Again, next data i.e. 16.5 (= H) is loaded into the stack in the same manner. Now the top of stack points to 4th location or R4 as shown in figure 6.13 (c). Multiplication is performed after loading and the steps are repeated to find Hydraulic input, P_i . This followed by loading of generator output, P_e into R2, as shown in figure 6.13 (e) followed by division (opcode 0x29) of result as shown in figure 6.13 (f) which gives the unit efficiency.

Finally, FSTORE instruction outputs the top of stack value and increments the top by one. Here, stack content in the extended double precision format is automatically converted into single precision and then to the hex format as shown in the figure 6.13(f) (data_int_out and data_fract_out).

Program:

The assembly language program written to compute the above expression is as follows:

Assume that initially the top-of-stack is R7.

```
FLOAD (= 1000 Kg/m3); /*decrements top-of-stack and loads in extended-double
                        format into R6*/
FLOAD g (= 9.806 m/s2); /*decrements top-of-stack and loads g in extended-double
                        format into R5*/
FMUL ST(1);           /*R5 ← R5 × R6*/
FLOAD H (= 16.5 m);  /*decrements top-of-stack and loads H in extended-double
                        format into R4*/
FMUL ST(1);           /*R4 ← R4 × R5*/
FLOAD Q (= 0.3125 m3/s); /*decrements top-of-stack and loads Q in extended-double
                        format into R3*/
FMUL ST(1);           /*R3 ← R3 × R4*/
FLOAD Pe (= 31.326 kW); /*decrements top-of-stack and loads Pe in extended-double
                        format into R2*/
FDIV ST(1);           /*R2 ← R2 ÷ R3*/
FSTORE;               /* stores U in output port and increments top-of-stack */
```

Machine language codes for all operations are given in Appendix B (Design Customized Instructions and their Usage).

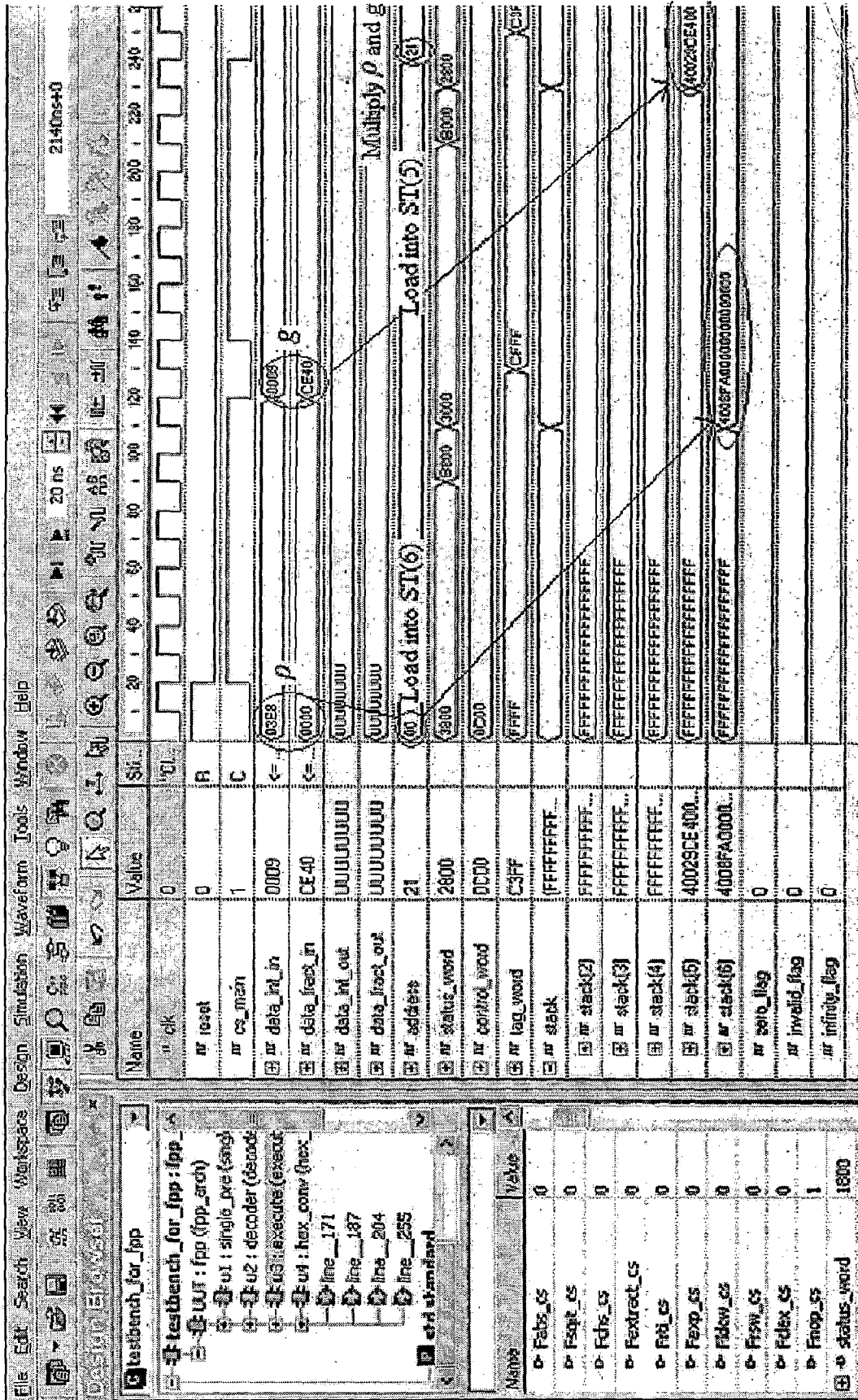


Figure 6.13 (a) Simulation Result of Unit Efficiency Measurement

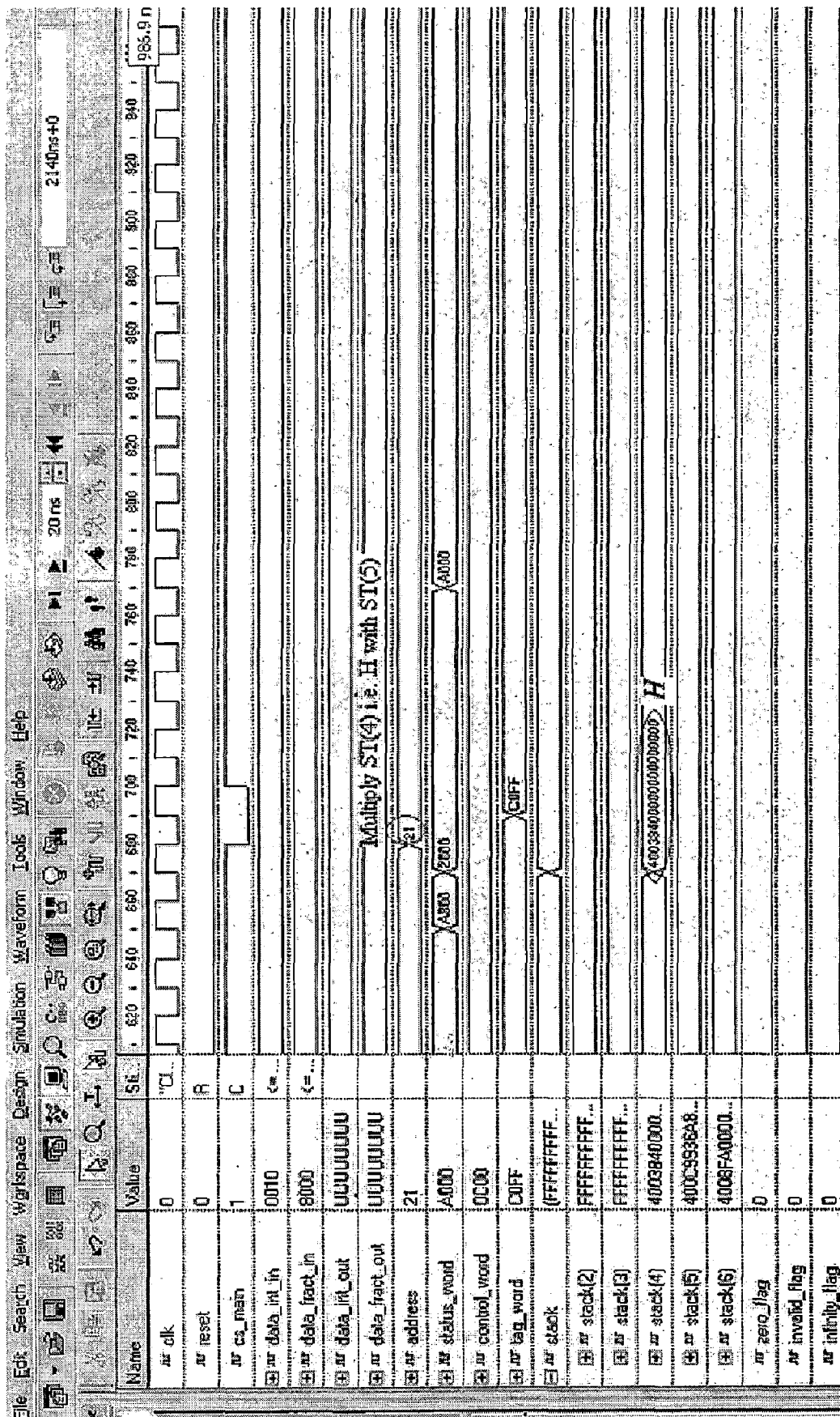


Figure 6.13 (c) Simulation Result of Unit Efficiency Measurement

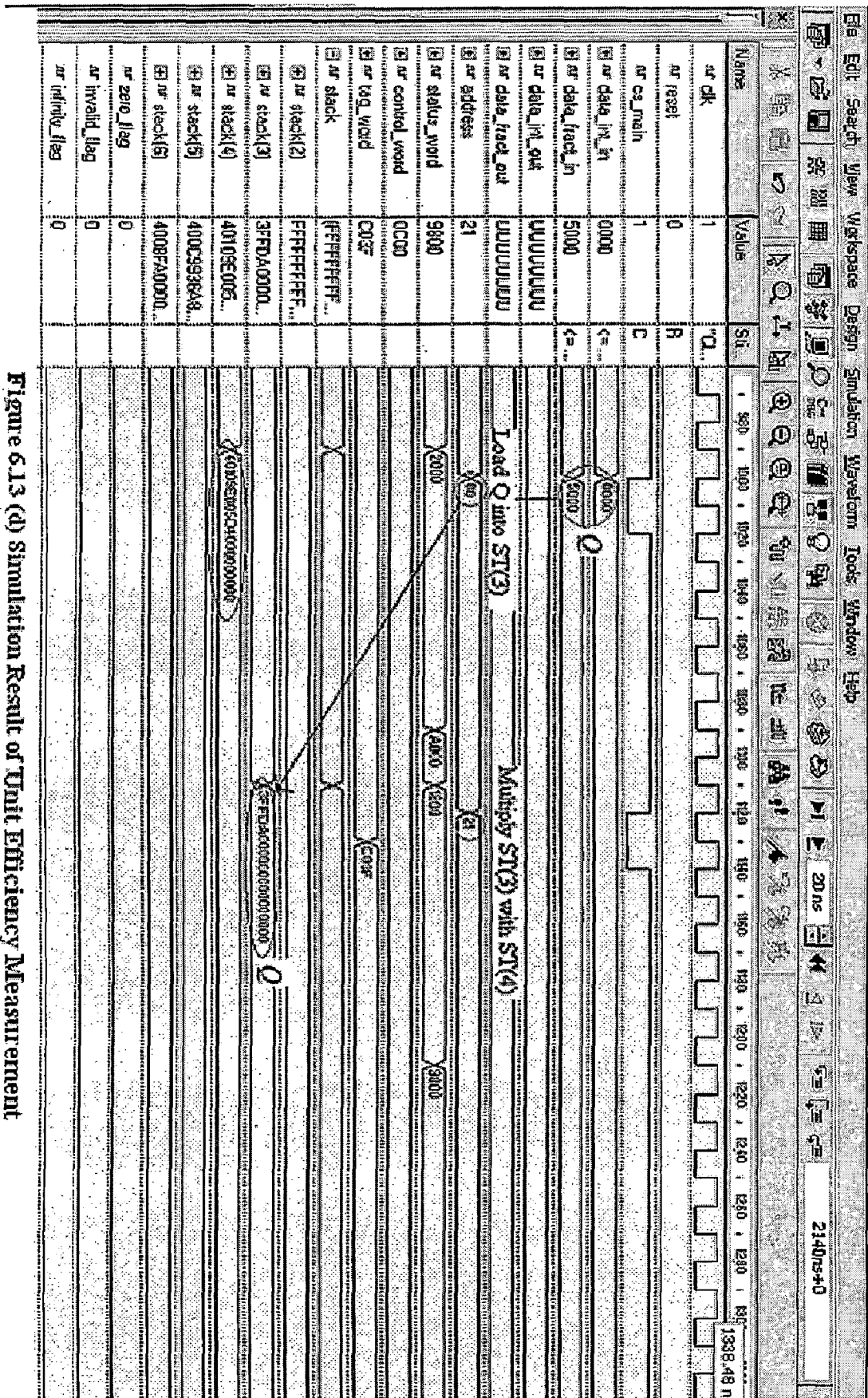


Figure 6.13 (d) Simulation Result of Unit Efficiency Measurement

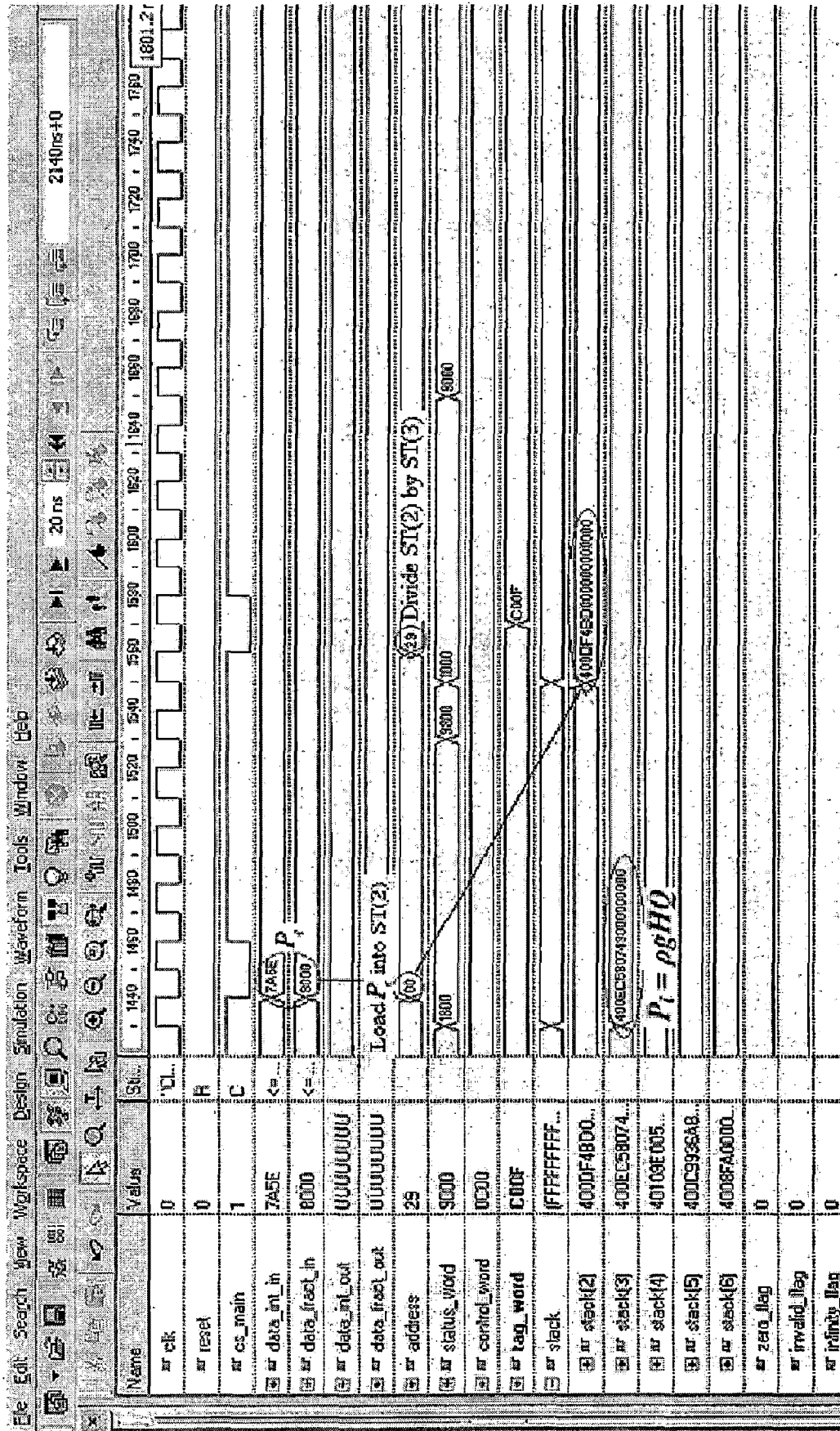


Figure 6.3 (e) Simulation Result of Unit Efficiency Measurement

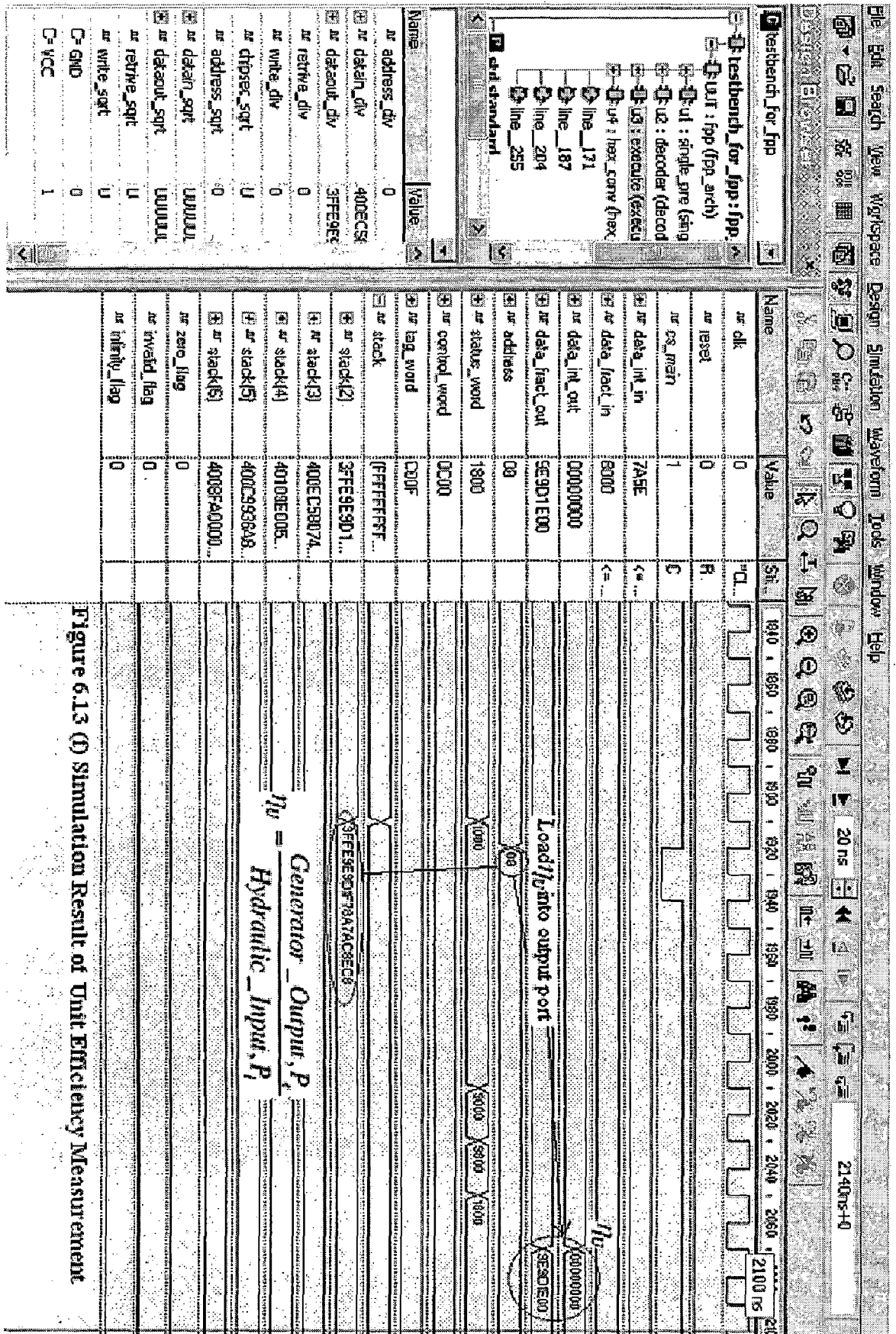


Figure 6.13 (D) Simulation Result of Unit Efficiency Measurement

Knowing the value of the generator efficiency, η_g , the turbine efficiency can be calculated by the following relation:

$$\text{TurbineEfficiency, } \eta_T = \frac{\text{Unit_Efficiency, } \eta_U}{\text{Generator_Efficiency, } \eta_g} \times 100\%$$

Explanation of turbine efficiency measurement waveform is similar to that of unit efficiency measurement waveform. Let the generator efficiency be 81.25%.

Program:

The following program gives the estimation of turbine efficiency.

```

FLOAD   $\eta_g$  (= 0.8125);           /*decrements top-of-stack and loads   $\eta_g$  in extended-
                                     double format into R6*/
FLOAD   $\eta_U$  (calculated above);  /*decrements top-of-stack and loads   $\eta_U$  in extended-
                                     double format into R5*/
FDIV ST(1);                       /*R2  $\leftarrow$  R2  $\div$  R3*/
FSTORE;                           /* stores   $\eta_T$  into output port and increments top-of-
                                     stack */

```

Machine language codes for the above programs are presented in Appendix B (Design Customized Instructions and their Usage).

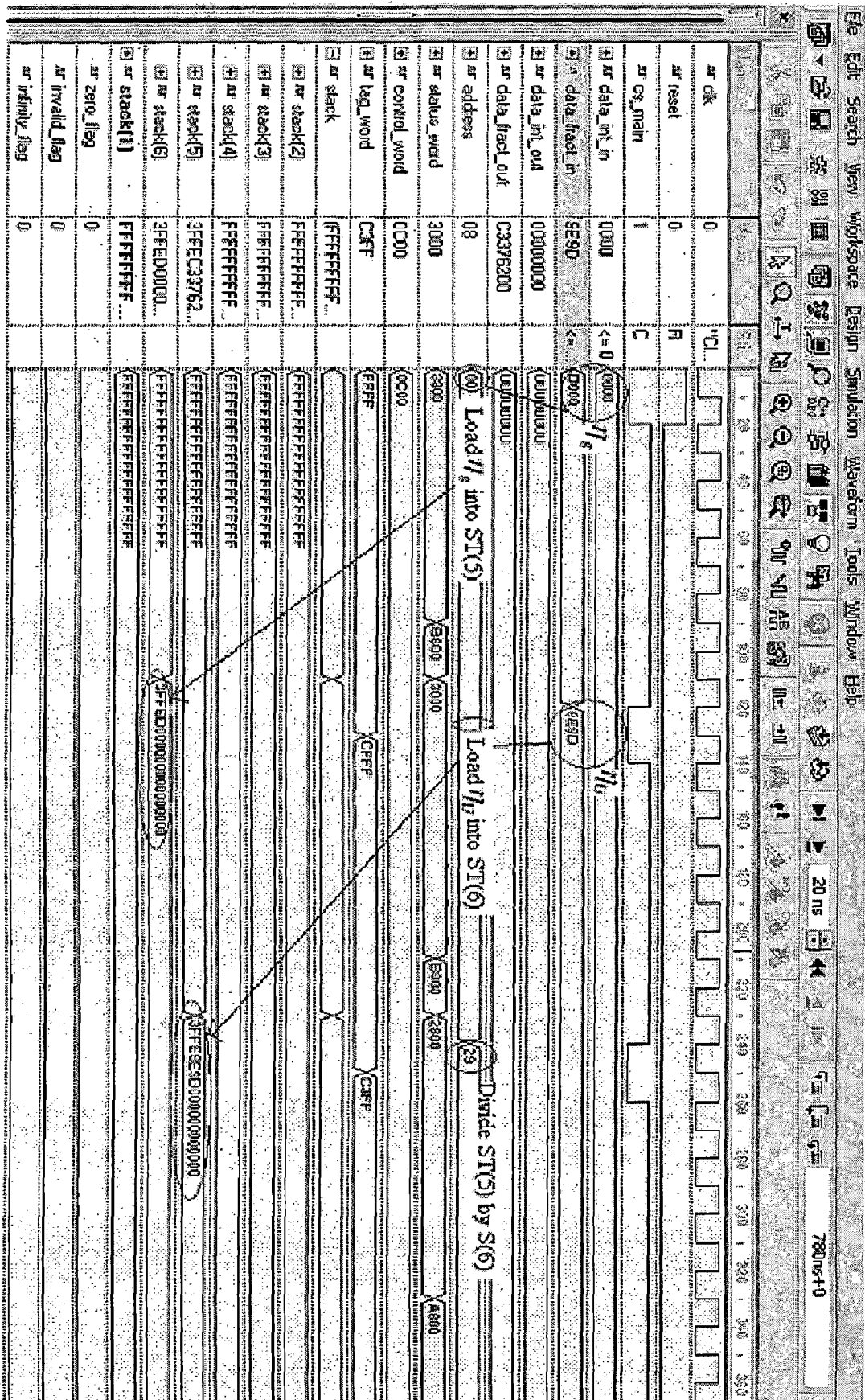


Figure 6.14 (a) Simulation Result of Turbine Efficiency Measurement

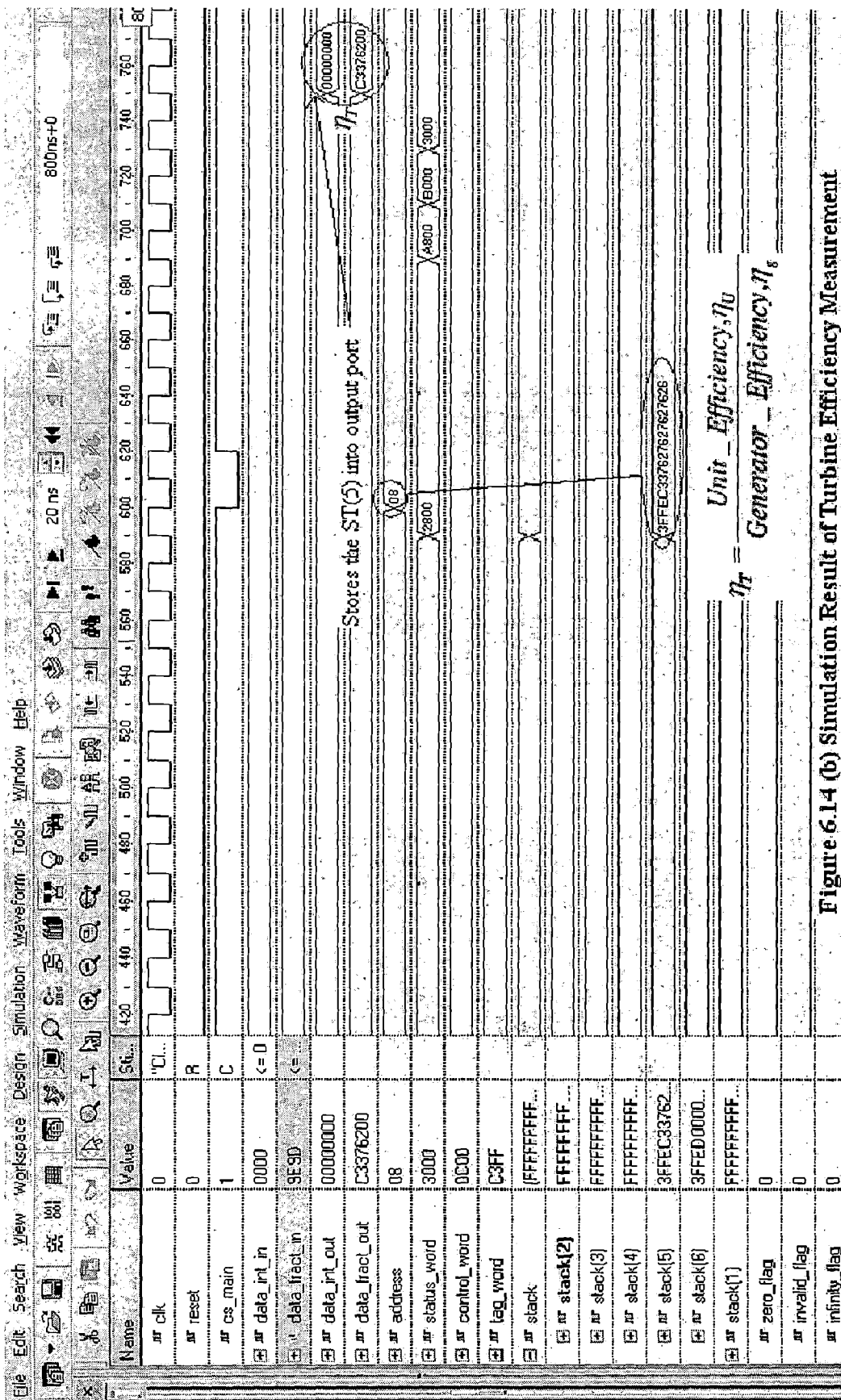


Figure 6.14 (b) Simulation Result of Turbine Efficiency Measurement

6.3 Verification of Simulation Results

Random binary test vectors were generated of length 32-bits, 16-bits for integer and 16-bits for fractional part, and were used to verify the output of each FP core element. The vectors used for testing were pre-selected such that it included all exceptions. The floating-point numbers included zero, maximum/minimum positive and negative numbers. Special values include positive and negative infinity and Not a Number. The output of the FP core elements are of IEEE-754 format. Appropriate flags are set for the special values and on an occurrence of overflow/underflow. The simulation results were verified both after the behavioral design and the structural design. The results for each fixed-point unit algorithms were also verified with the standard simulator. The standard simulator used for verification was Aldec's Active HDL 6.1. The following table will give idea of what kind of test vectors were generated.

Table 6.1 Example of Test Vectors

<i>FP operation</i>	<i>Result</i>	<i>Value or flagset</i>
ZERO - NaN	NaN	NaN flag
ZERO - NEG_INF	POS_INF	Infinity flag
MIN_POS - MAX_POS	MAX_NEG	dfeffff ffffff
MIN_POS - NaN	NaN	NaN flag
MAX_NEG - MAX_POS	NEG_INF	Infinity flag
POS_INF - NEG_INF	POS_INF	Infinity flag
7.5 - POS_INF	NEG_INF	Infinity flag
NEG_INF + POS_INF	ZERO	Zero flag
NaN - ZERO	NaN	NaN flag
ZERO * 7.5	ZERO	Zero flag
POS_INF * ZERO	ZERO	Zero flag
NEG_INF * 3.25	ZERO	Zero flag
NaN * 0.45	NaN	NaN flag
7.5 / ZERO	Infinity	Infinity flag

CHAPTER 7***CONCLUSION AND FUTURE SCOPE***

This chapter concludes and suggests the future work which can be done in this area.

7.1 Conclusion

I presented the design of the Floating-Point Arithmetic unit for FPGAs based on IEEE 754 standard. Performing the arithmetic operations on IEEE Floating-point numbers imposed challenges beyond the challenges of Fixed-Point arithmetic. These challenges particularly include the task of normalization and IEEE compliant rounding. I implemented both normalization unit and rounding unit (capable of performing all four rounding modes). Based on simulation and synthesis results, it is concluded that the design is performing in desired fashion and the purposed design is very suitable for FPGAs (see Appendix C *Synthesis Report*, less than 17% of available resources in Virtex II Pro is used). The Floating-point arithmetic unit is simulated using Aldec's Active HDL 6.1 and synthesized using Xilinx ISE 7.1i supported by ModelSim 5.7. The design is targeted for Xilinx Virtex II Pro FPGA. The input/output number format confirms IEEE-754 standard single precision real numbers. Internally, calculations are performed according to IEEE-754 standard double-extended precision real numbers (as incorporated in Intel Pentium4 processor). This inherited feature assists floating-point arithmetic unit in enhancing the accuracy. Besides implementing the addition, subtraction, multiplication, division, square root, and absolute unit, some other supporting units like general purpose registers, control registers, tag register, status register etc are also implemented to make it independent programmable chip and the FPU works in stand-alone mode. With that limited exception handling has also been implemented. Although most of the features are taken into account from Intel's Pentium4 but new things have been added to Pentium4's FPU and successfully implemented. One of the most exciting such thing is the Tag register. Tag register available in Pentium4 needs to be taken care

of by the end user to check the validity of a number, but here in my design, it is automatic. The FPU checks, itself, the validity of a number before and after the computation. Thereby, it also saves the large number of clock cycles whenever the data register is empty or contains some invalid number. In essence, my work is superset of all previous works related to this area. Since the target application was turbine efficiency measurement which may require lot of computation on number of variables, special attention has been paid to all kinds of arithmetic algorithms to design best possible core units for FPU.

7.2 Suggestions for Future Work

Some suggestions are presented in this section which can be considered for future work.

- ❖ Apply pipelining to all units/subunits. It helps in reducing latency.
- ❖ A few more complex instruction related to trigonometric and logarithmic can be integrated into the FPU to perk up flexibility.
- ❖ I implemented Normalization unit within the floating-point arithmetic modules. Area consumed will decrease if a separate hardware module is designed for Normalization.
- ❖ Denormalization unit can also be implemented [26][28][32]. I have discussed advantages of denormalization in Chapter 3.
- ❖ The work can be extended for quad precision and dual double precision format [38].
- ❖ Power reduction techniques can also be implemented.

REFERENCES

- [1] Altera Corporation, <http://www.altera.com>
- [2] Dr. H. K. Verma and Dr. Arun Kumar, "Instrument Networking for Efficiency Measurement in Small Hydro-Power Stations", *IIT Roorkee*.
- [3] IEC-60041 (1991): Field acceptance test to determine the hydraulic performance of hydraulic turbines, storage pumps and storage turbines.
- [4] B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating-Point Arithmetic", *IEEE Transactions on VLSI systems*, 2(3), September 1994.
- [5] N. Shirazi, A. Walters, and P. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA based Custom Computing Machines", *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, April 1995.
- [6] L. Louca, T. A. Cook, and W. H. Johnson, "Implementation of Single Precision Floating Point addition and multiplication on FPGAs", K. L. Pocek and J. Arnold, editors, *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 107-116, April 1996.
- [7] W. B. Ligon III, S. Mcmillan, G. Monn, K. Schonover, F. Stivers, and K. D. Underwood, "A Re-evaluation of the Practicality of Floating Point Operations on FPGAs", *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.
- [8] I. Stamoulis, M. White, and P. F. Lister, "Pipelined Floating-Point Arithmetic optimized for FPGA architectures", *9th International Workshop on Field Programmable Logic and Applications*, volume 1673 of *LNCS*, pages 365-370, August-September 1999.

- [9] I. Sahin, C. S. Gloster, and C. Doss, "Feasibility of Floating Point Arithmetic in reconfigurable computing systems", *2000 MAPLD International Conference*, 2000.
- [10] J. Dido, N. Geraudie, L. Loiseau, O. Payeur, y. Savaria, and D. Poirier, "A flexible Floating Point Format for Optimizing Data Paths and operators in FPGA based DSPs", *International Symposium on Filed Programmable Gate Arrays*, pages 50-55, ACM Press, February 2002.
- [11] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna, "Analysis of High-Performance Floating-Point Arithmetic on FPGAs," *International Parallel and Distributed Processing Symp.*, pp. 149b, April 2004.
- [12] A. Malik and S. Ko, "Efficient Implementation of Floating Point Adder using pipelined LOP in FPGAs," *IEEE Canadian Conference on Electrical and Computer Engineering*, pp. 688-691, May 2005.
- [13] *IEEE Standard Board and ANSI*, "IEEE Standard for Binary Floating-Point Arithmetic," 1985, IEEE Std 754-1985.
- [14] G. Hinton, et al., "*The Microarchitecture of the Pentium 4 Processor*," Intel Technology J., 1st quarter 2001 at.
<http://www.intel.com/technology/itj/q12001.htm>
- [15] Mentor Graphics corporation, <http://www.mentor.com>
- [16] John L Henessy and David A Patterson, "*Computer Arithmetic – A Quantitative Approach*", Morgan Kaufmann, 2003.
- [17] Xilinx corporation, <http://www.xilinx.com>

- [18] Wakerly, J. F. 2000, "*Digital Design: Principles and Practices*", 3rd ed. Upper Saddle River, NJ: Printice Hall.
- [19] Perry, D. L. 2004, "*VHDL: Programming By Example*", 4th ed. New York: McGraw-Hill Companies inc.
- [20] Bhasker, J. 1997, "*A VHDL Primer*", Allentown, PA: Star Galaxy Press.
- [21] T. S. Hall and J. O. Hamblen, "System-on-a-Programmable-Chip Development Platforms in the Classroom", To appear in *IEEE Transactions on Education*, 2004, [Online Document, Cited 2004 February 29], Available HTTP: http://www.ece.gatech.edu/~hamblen/papers/SOC_top.pdf
- [22] Y. Li and W. Chu, "A New Non-restoring Square root algorithm and its VLSI implementations", *Proc. of 1996 IEEE international conference on computer design: VLSI in computers and Processors*, Austin, Texas, USA, October 1996, pp 538-544.
- [23] Xiaojun Wang and Brent E. Nelson, "Tradeoffs of Designing Floating-Point Division and Square Root on Virtex FPGAs", *Proceeding of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.
- [24] Taek-Jun Kwon, Joong-Seok Moon, Jeff Sondeen and Jeff Draper, "A 0.18 μ m Implementation of Floating-Point Unit for a Processing-In-Memory system", *IEEE*, 2003, pp. 453-456.
- [25] Kyung-Nam Han, Sang-Wook Han and Euisik Yoon, "A New Floating-Point Normalization Scheme by Bit-Parallel Operation of Leading One Position Value", *IEEE*, 2002, pp. 221-224.
- [26] Hu He, Zheng Li and Yihe Sun, "Multiply-Add fused Float Point Unit with On-Fly Denormalized Number Processing", *IEEE*, 2005, pp.1466-1468.

- [27] Ramyanshu Datta and Jacob A. Abraham, "A Low Latency and Low Power Dynamic Carry Save Adder", *IEEE*, 2004, pp. 477-480.
- [28] Eric M. Schwarz, "FPU Implementations with Denormalized Numbers", *IEEE Computer Society*, May 2005, pp. 825-836.
- [29] Ling Zhuo and Viktor K. Prasanna, "High-Performance and Area-Efficient Reduction Circuits on FPGAs", *Proceedings of the 17th International Symposium on Computer Architecture and High Performance Computing*, 2005.
- [30] Guenter Gerwig, Holger Wetter, Eric M. Schwarz and Juergen Haess, "High Performance Floating-Point Unit with 116-bit wide divider", *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, 2003.
- [31] Jian Liang and Russell Tessier and Oskar Mencer, "Floating Point Unit Generation and Evaluation for FPGAs", *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.
- [32] Li Zheng, He Hu and Sun Yihe, "Floating-Point Unit Processing Denormalized Numbers", 2003, pp. 90-93.
- [33] Alex Panato, Sandro Silva, Flávio Wagner, Marcelo Johann, Ricardo Reis and Sergio Bampi, "Design of Very Deep Pipelined Multipliers for FPGAs", *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition Designers' Forum, IEEE*, 2004.
- [34] Peter-Michael Seidel, "High-radix Implementation of Floating-point Addition", *Proceeding of the 17th IEEE Symposium on Computer Arithmetic*, 2005.

[35] Haiping Sun and Minglun Gao, "Unified Bit for Leading Zero Anticipatory Logic for High Speed Floating-Point Addition", *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, 2003, pp 786-789.

[36] Steven D. Krueger and Peter-Michael Seidel, "Design of an On-Line IEEE Floating-Point Addition for FPGAs", *Proceeding of the 12th IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004.

[37] Peter-Michael Seidel, "Design of an On-Line IEEE Floating-Point Multiplication and Division fro Reduced Power Dissipation", *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, 2004, pp 498-502.

[38] Ahmet Akkas and Michael J. Schulte, "A Quadruple Precision and Dual Double Precision Floating-Point Multiplier", *Proceedings of the Euromicro Symposium on Digital System Design*, 2003.

[39] He Jing and Han Yue-qiu, "A Pipelined Multiplication Unit", *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, 2003, pp. 1247-1250.

[40] C. Chen, L. A. Chen and J. R. Cheng, "Architectural design of a fast floating-point multiplication-add fused unit using signed-digit addition", *Proceedings of IEE Symposium on Comput. Digit. Tech., Vol. 149, No. 4, July 2002*, pp. 113-120.

[41] Nicolas Brissibarre and Jean-Michel Muller, "Accelerating Correctly Rounded Floating-Point Division when the Divisor is known in advance", *IEEE Computing Society*, December 2003, pp. 1069-1072.

[42] Irvin Ortiz and Manuel Jimenez, "Scalable Pipeline Insertion in Floating-Point Division and Square root units", *IEEE*, 2004, pp. 225-228.

- [43] Yamin Li and Wanming Chu, "Implementation of Single Precision Floating-point Square root on FPGAs", *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, 1997, pp 226-232.
- [44] Liang-Kai Wang and Michael J. Schulte, "Decimal Floating-Point Square root using Newton-Raphson iteration", *Proceedings of the 16th International Conference on Application-Specific Systems, Architecture and Processors*, 2005.
- [45] Jose-Alejandro Pineiro, "High-Speed Double Precision computation of Reciprocal, Division, Square root, and Inverse Square root", *IEEE*, 2002.
- [46] Luo Min, Bai Yong-Qiang, Shen Xu-Bang and Gao-De-Yuan, "The Implementation of an Out-of-Order Execution Floating-Point unit", *IEEE*, 2004, pp. 1384-1387.
- [47] Claudio Brunelli, Fabio Campi, Jari Nurmi and Juha Kylliainen, "Reconfigurable FPU as IP component for SoCs", *IEEE*, 2004, pp. 103-106.
- [48] Neil Burgess, "Prenormalization Rounding in IEEE Floating-Point Operations using a Flagged Prefix Adder", *IEEE Transactions on Very Large Scale Integration Systems (VLSI)*, Vol. 13, No. 2, February 2005, pp. 266-277.
- [49] Gokul Govindu, Ling Zhuo, Seonil Choi and Viktor Prasanna, "Analysis of High-performance Floating-point Arithmetic on FPGAs", *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, *IEEE*, 2004.
- [50] <http://www.howstuffworks.com/hydro>

APPENDIX A***GLOSSARY***

A**ASICs**

An ASIC (application-specific integrated circuit) is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use.

Aldec's Active HDL

The Active-HDL from Aldec Inc. suite is a comprehensive and totally integrated environment for digital IC design and verification that employs hardware description languages and C/C++ solutions. It provides tools for efficient and vendor independent design implementation and testing for engineers and design teams. Active-HDL supports even the most complex FPGA and ASIC designs.

Altera

Altera Corporation is a manufacturer of programmable logic devices.

ASCII

ASCII (*American Standard Code for Information Interchange*), generally pronounced [æski], is a character encoding based on the English alphabet. ASCII codes represent text in computers, communications equipment, and other devices that work with text.

C

CPLD

CPLD stands for Complex Programmable Logic Device. It is a programmable logic device with complexity between that of FPGAs and PALs, and architectural features from both. The building block of a CPLD is the macro cell, which contains logic implementing disjunctive normal form expressions and more specialized logic operations.

CAD

Computer-aided design (CAD) is the use of a wide range of computer-based tools that assist engineers, architects and other design professionals in their design activities. It is the main geometry authoring tool within the Product Lifecycle Management process and involves both software and sometimes special-purpose hardware.

Compile-Time

In computer science, compile time, as opposed to runtime, is the time when a compiler compiles code written in a programming language into an executable form.

CLBs, IOBs & Interconnects

The FPGA has three major configurable elements: configurable logic blocks (CLBs), input/output blocks, and interconnects. The CLBs provide the functional elements for constructing user's logic. The IOBs provide the interface between the package pins and internal signal lines. The programmable interconnect resources provide routing paths to connect the inputs and outputs of the CLBs and IOBs onto the appropriate networks.

D

DSP

Digital signal processing (DSP) is the study of signals in a digital representation and the processing methods of these signals. DSP and analog signal processing are

subfields of signal processing. DSP has three major subfields: audio signal processing, digital image processing and speech processing. The microprocessor class of digital signal processor (DSP) is a specialized microprocessor designed specifically for digital signal processing, generally in real-time.

E

EDA

Electronic design automation (EDA) is the category of tools for designing and producing electronic systems ranging from printed circuit boards (PCBs) to integrated circuits. This is sometimes referred to as ECAD (electronic computer-aided design) or just CAD.

F

FPGA

Field-programmable gate array or FPGA is a semiconductor device containing programmable logic components and programmable interconnects. The programmable logic components can be programmed to duplicate the functionality of basic logic gates (such as AND, OR, XOR, NOT) or more complex combinatorial functions such as decoders or simple math functions. In most FPGAs, these programmable logic components (or logic blocks, in FPGA parlance) also include memory elements, which may be simple flip-flops or more complete blocks of memories.

Finite State Machine (FSM)

A finite state machine or finite automaton is a model of behavior composed of states, transitions and actions. A state stores information about the past, i.e. it reflects the input changes from the system start to the present moment. A transition indicates a state change and is described by a condition that would need to be fulfilled to enable the transition. An action is a description of an activity that is to be performed at a given moment.

Front-end and back-end

In their most general meanings, the terms front end and back end refer to the initial and the end stages of a process flow.

In **software design**, the front-end is the part of a software system that deals with the user, and the back-end is the part that processes the input from the front-end. The separation of software systems into "front ends" and "back ends" is a kind of abstraction that helps to keep different parts of the system separated.

In **compilers**, the front-end translates the source language into an intermediate representation, and the back-end works with the internal representation to produce code in the output language.

In **electronic design automation**, front-end stages of the design cycle are logical and electrical design (e.g., schematic capture, logic synthesis). Sometimes floor planning is also considered front-end. Back-end are place and route, custom layout design and physical verification (design rule checking, layout versus schematic, parasitic extraction).

Many **programs** are divided conceptually into front and back ends, but in most cases, the "back-end" is hidden from the user. However, sometimes programs are written which serve simply as a front-end to another, already existing program, such as a graphical user interface (GUI) which is built on top of a command-line interface. This type of front-end is common in Unix GUIs, where individual programs are developed on the design philosophy of many small, tested programs, able to run independently or together.

G

Graphical User Interface

A graphical user interface (or GUI, sometimes pronounced "gooey") is a method of interacting with a computer through a metaphor of direct manipulation of graphical images and widgets in addition to text. GUIs display visual elements such as icons, windows and other gadgets.

H

High-Level Language (HLL)

A programming language such as C, FORTRAN, or Pascal that enables a programmer to write programs that are more or less independent of a particular type of computer. Such languages are considered high-level because they are closer to human languages and further from machine languages. In contrast, assembly languages are considered low-level because they are very close to machine languages.

Hardware Description Languages (HDLs)

In electronics, a hardware description language or HDL is any language from a class of computer languages for formal description of electronic circuits. It can describe the circuit's operation, its design, and tests to verify its operation by means of simulation.

I

IP (Intellectual Property)

In electronic design a semiconductor intellectual property core, IP block, IP core, or core is a reusable unit of logic, cell, or chip layout design. Cores that are the property of one party may be licensed to another party though cores can also be owned and used by a single party alone. The term is derived from the licensing of the patent and source code copyright intellectual property rights that subsist in the design. An uncommon alternative expansion is "integrated processor block". IP cores can be used as building blocks within ASIC chip designs or FPGA logic designs.

IDE

An integrated development environment (IDE), also known as integrated design environment and integrated debugging environment, is a type of computer software that assists computer programmers to develop software. IDEs normally consist of a source code editor, a compiler and/or interpreter, build-automation tools, and (usually) a debugger.

Integrated Circuit

Another name for a chip, an integrated circuit (IC) is a small electronic device made out of a semiconductor material. Integrated circuits are often classified by the number of transistors and other electronic components they contain:

SSI (small-scale integration): Up to 100 electronic components per chip

MSI (medium-scale integration): From 100 to 3,000 electronic components per chip

LSI (large-scale integration): From 3,000 to 100,000 electronic components per chip

VLSI (very large-scale integration): From 100,000 to 1,000,000 electronic components per chip

ULSI (ultra large-scale integration): More than 1 million electronic components per chip

IA-32

IA-32, sometimes generically called x86-32, is the instruction set architecture of Intel's most successful microprocessors. Within various programming language directives it is also referred to as "i386". The term may be used to refer to the 32-bit extensions to the original x86 architecture, or to the architecture as a whole. The term means Intel Architecture, 32-bit, which distinguishes it from the 16-bit versions of the architecture that preceded it.

IEEE

Institute of Electrical and Electronics Engineering.

IEEE-754

IEEE Standard for Binary Floating-point Arithmetic is the most widely used standard for floating-point computation, and is followed by many CPU and FPU implementations.

J

JTAG

JTAG, an acronym for Joint Test Action Group, is the usual name used for the IEEE 1149.1 standard entitled Standard Test Access Port and Boundary-Scan Architecture for test access ports used for testing printed circuit boards using boundary scan.

L

LUT

In digital logic, an n-bit lookup table can be implemented with a multiplexer whose select lines are the inputs of the LUT and whose inputs are constants. This is an efficient way of encoding Boolean logic functions, and 4-bit LUTs are in fact the key component of modern FPGAs.

LE

Logic Elements, FPGA are defined in terms of Les. . The array of logic cells and interconnects form a fabric of basic building blocks for logic circuits. Complex designs are formed by combining these Logic elements to build the desired circuit.

M

MicroBlaze

The MicroBlaze is a soft processor core from Xilinx for use in Xilinx FPGAs. The MicroBlaze is based on a RISC architecture very similar to the DLX architecture described in a popular computer architecture book by Patterson and Hennessy. It features a 3-stage pipeline, with most instructions completing in a single cycle. Both instruction and data words are 32 bits.

N

Netlist

In Electronic Design domain, a "netlist" describes the connectivity of an electronic design. Netlists usually convey connectivity information and provide nothing more than instances, nets, and perhaps some attributes. Netlists can be either *physical* or *logical*; either *instance-based* or *net-based*; and *flat* or *hierarchical*.

O

On-Chip Peripheral Bus (OPB)

This is a part of the IBM CoreConnect architecture. A processor (hard or soft) core accesses low speed and low performance system resources through On-chip Peripheral Bus (OPB). The OPB is a fully synchronous bus that functions independently at a separate level of bus hierarchy.

On-Chip/Off-Chip Memory

On-Chip memory refers to a memory tightly coupled to processor, generally on the same silicon chip. Off-Chip memory refers to a memory which resides outside the silicon chip where the processor actually resides.

P

PROM

A programmable read-only memory (PROM) or field programmable read-only memory (FPRM) is a form of digital memory where the setting of each bit is locked by a fuse or antifuse. Such PROMs are used to store programs permanently. Some of its types are EPROM, EEPROM etc.,

Programmable Logic Devices (PLDs)

Programmable Logic Device or PLD is an electronic component used to build digital circuits. Unlike a logic gate, which has a fixed function, a PLD has an undefined function at the time of manufacture. Before the PLD can be used in a circuit it must be programmed.

Process (in VHDL)

Behavioral descriptions are supported with the process statement. The process statement can appear in the body of an architecture declaration just as the signal assignment statement does. The contents of the process statement can include sequential statements like those found in software programming languages. These statements are used to compute the outputs of the process from its inputs. Sequential statements are often more powerful, but sometimes have no direct correspondence to a hardware implementation. The process statement can also contain signal assignments in order to specify the outputs of the process. The body of the process appear between the *begin* and *end* keywords. Example of a process is shown below.

```
compute_xor: process (b,c)
begin
  a<=b xor c;
end process;
```

Sensitivity List

Next to the keyword *process* (in *VHDL*), which starts the definition of a process there is a list of signals in parenthesis, called the *sensitivity list*. The signal sensitivity list is used to specify which signals should cause the process to be re-evaluated. Whenever any event occurs on one of the signals in the sensitivity list, the process is re-evaluated.

Run-Time

In computer science, runtime or run time describes the operation of a computer program, the duration of its execution, from beginning to termination.

Reconfigurable Computing (RC)

Reconfigurable computing is computer processing with highly flexible computing fabrics. The principal difference when compared to using ordinary microprocessors is the ability to make substantial changes to the data path itself in addition to the control flow.

Ex: FPGA, CPLD etc.

RTL

Register Transfer Language (RTL) has two meanings in computer science. The first is an intermediate representation used by the GCC compiler. *Register Transfer Language* also refers to a language that defines precisely what each instruction in a processor does, to a level of detail that allows synthesis of the hardware. The acronym RTL is also used for register transfer level, an attribute of a hardware description language.

RAM

Random-access memory (commonly known by its acronym RAM) refers to data storage formats and equipment that allow the stored data to be accessed in any order -- that is, at random, not just in sequence.

Static Random Access Memory (SRAM) is a type of semiconductor memory. The word "static" indicates that the memory retains its contents as long as power remains applied,

Dynamic random access memory (DRAM) is a type of random access memory that stores each bit of data in a separate capacitor. As real-world capacitors are not ideal and hence leak electrons, the information eventually fades unless the capacitor charge is refreshed periodically. Different variations of DRAMS are: PSRAM, DDR SDRAM, DRDRAM, SDRAM, QDR SDRAM, SGRAM, MDRAM, BEDO DRAM, EDO DRAM, WRAM, and VRAM.

RISC (Reduced Instruction Set Computer)

The reduced instruction set computer, or RISC, is a microprocessor CPU design philosophy that favors a smaller and simpler set of instructions that all take about the same amount of time to execute. The most common RISC microprocessors are ARM, DEC Alpha, PA-RISC, SPARC, MIPS, and IBM's PowerPC.

RTOS

A real-time operating system (RTOS) is a class of operating system intended for real-time applications. Examples include embedded applications (programmable thermostats, household appliance controllers, mobile telephones), industrial robots, industrial control (SCADA), and scientific research equipment.

S

SOC

System-on-a-chip (SoC or SOC) is an idea of integrating all components of a computer or other electronic system into a single chip. It may contain digital, analog, mixed-signal, and often radio-frequency functions – all on one chip. A typical application is in the area of embedded systems.

SOPC

System-on-a-Programmable-chip, Similar to SOC, only difference it is build in programmable hardware

SOPC Builder

SOPC Builder is a powerful system development tool developed by Altera for creating systems based on processors, peripherals, and memories. SOPC Builder enables you to define and generate a complete SOPC in much less time than using traditional, manual integration methods

Soft Processor/Soft-core

A soft processor is a processor created out of the configurable logic in a FPGA.

Synthesis

In the world of electronic design automation, synthesis is the process of converting a digital design written in a hardware description language (HDL) into a low-level implementation consisting of primitive logic gates. Most large integrated circuits designed today are written in an HDL and "compiled" using a synthesis product.

Serial Port

In computing, a serial port is an interface on a computer system through which information transfers in or out one bit at a time

U

UCF file

The User Constraints File is an ASCII file that you create. You can create this file by hand or by using the Constraints Editor. The UCF file contains timing and layout constraints that affect how the logical design is implemented in the target device. The constraints in the file are added to the information in the output NGD file.

User-defined

User defined items are the one which are developed and implemented by the users or developers as applicable.

V

VHDL

VHDL or VHSIC Hardware Description Language, is commonly used as a design-entry language for field-programmable gate arrays and application-specific integrated circuits in electronic design automation of digital circuits.

Verilog

Verilog is a hardware description language (HDL) used to model electronic systems. The language (sometimes called *Verilog HDL*) supports the design, testing, and implementation of analog, digital, and mixed-signal circuits at various levels of abstraction.

X

Xilinx, Inc.

It is the world's largest developer and manufacturer of the class of reconfigurable hardware chips known as Field-Programmable Gate Arrays (FPGAs). Xilinx is a developer of FPGA and CPLD devices that are used in numerous applications within telecommunications, automotive, consumer, defense, and other fields. Xilinx offers device families for glue logic (CoolRunner, CoolRunner II), low-cost (Spartan), and high-end (Virtex) applications in addition to supporting devices such as PROMs.

Xilinx ISE & EDK

Xilinx offers electronic design automation (EDA) tools for use with its devices. Chief among these is ISE, which offers a complete EDA flow. The other being Xilinx's Embedded Developer's Kit (EDK), which is aimed primarily at designers wishing to use the embedded PowerPC 405 core in the Virtex-II Pro and Virtex-4, or Xilinx's own soft microprocessor/microcontroller (MicroBlaze) in their designs. Other domain-specific tools include System Generator for DSP, which provides seamless simulation and implementation of high-performance DSP designs on Xilinx's FPGAs.

X86

x86 or 80x86 is the generic name of a microprocessor architecture first developed and manufactured by Intel. The x86 architecture currently dominates the desktop computer, portable computer, and small server markets.

XS Board

FPGA development board series.

APPENDIX B

DESIGN CUSTOMIZED INSTRUCTION AND THEIR USAGE

1 FLOAD instruction

Load real number from the input port into the stack.

Opcode: 0x00

Clock cycles: 5

Description:

It merges the integer and fraction part of the real number and then converts the number into extended double precision format and decrements the data register stack by one and stores the converted data into top of stack.

2 FSTORE instruction

Store real number from the stack into the output port.

Opcode: 0x08.

Clock cycles: 6

Description :

It retrieves the extended double precision number from the top of stack, converts it into equivalent real number, and then stores the converted data into output port

3 FADD instruction

Add TOP of data register stack to the TOP + X and store the result in TOP of data register stack.

Opcode: 0x10 + X. (X → 0 to 7)

Clock cycles: 18 (if both numbers are valid)

7 (if one of the number is invalid, zero or infinite)

Description:

It adds the content (extended double precision number) of the top of data register stack to the content of the top of stack plus X mentioned in instruction—0

to 7 (extended double precision number) and stores the result (extended double precision number) into the top of data register stack.

4 FSUB instruction

Subtract TOP + X of data register stack from the TOP and store the result in TOP of data register stack.

Opcode: $0x18 + X$. ($X \rightarrow 0$ to 7)

Clock cycles: 18 (if both numbers are valid)

7 (if one of the number is invalid, zero or infinite)

Description:

It subtracts the content (extended double precision number) of the top of data register stack plus X mentioned in instruction—0 to 7 from the content of the top of stack (extended double precision number) and stores the result (extended double precision number) into the top of data register stack.

5 FMUL instruction

Multiply TOP of data register stack to the TOP + X and store the result in TOP of data register stack.

Opcode: $0x20 + X$. ($X \rightarrow 0$ to 7)

Clock cycles: 11 (if both numbers are valid)

7 (if one of the number is invalid, zero or infinite)

Description:

It multiplies the content (extended double precision number) of the top of data register stack to the content of the top of stack plus X mentioned in instruction—0 to 7 (extended double precision number) and stores the result (extended double precision number) into the top of data register stack.

6 FDIV instruction

Divides TOP + X of data register stack from the TOP and stores the result in TOP of data register stack.

Opcode: $0x28 + X$. ($X \rightarrow 0$ to 7)

Clock cycles: 18 (if both numbers are valid)

7 (if one of the number is invalid, zero or infinite)

Description:

It divides the content (extended double precision number) of the top of data register stack plus X mentioned in instruction—0 to 7 from the content of the top of stack (extended double precision number) and stores the result (extended double precision number) into the top of data register stack.

7 FSQRT instruction

Computes square-root of TOP of data register stack and store the result in TOP of data register stack.

Opcode: $0x38$.

Clock cycles: 66 (if number is valid)

7 (if number is invalid, zero infinite or negative)

Description:

It computes square root of the content (extended double precision number) of the top of data register stack and stores the result (extended double precision number) into the top of data register stack.

8 FCHS instruction

Change sign of TOP of data register stack and store the result in TOP of data register stack.

Opcode: $0x40$.

Clock cycles: 5

Description:

It changes the sign of the content of the top of data register stack (extended double precision number) and stores the result (extended double precision number) into the top of data register stack.

9 FABS instruction

Computes the Absolute value of TOP of data register stack and store the result in TOP of data register stack.

Opcode: 0x30.

Clock cycles: 5

Description:

It computes the absolute value of the content of the top of data register stack (extended double precision number) and stores the result (extended double precision number) into the top of data register stack.

10 RTI instruction

Round to nearest integer TOP of data register stack and store the result in TOP of data register stack.

Opcode: 0x58.

Clock cycles: 5

Description:

It rounds the content of the top of data register stack (extended double precision number) to the nearest integer and stores the result (extended double precision number) into the top of data register stack.

11 LDCW instruction

Load FPU control word

Opcode: 0x60.

Clock cycles: 5

Description:

Load the immediate input data into the control word register.

12 FRSW instruction

Store FPU status word into the output port.

Opcode: 0x68

Clock cycles: 5

Description:

Store the status word (different states) into the output port.

13 FCLEX instruction

Clear floating-point exception flags.

Opcode: 0x70

Clock cycles: 5

Description:

Clears the exception flags in the status word.

14 NOP instruction

Do nothing.

Opcode: 0x78

Clock cycles: 4

Description:

Do nothing for 4 clock cycles.

APPENDIX C

SYNTHESIS REPORT

Presented work is simulated using Aldec's Active HDL and the simulation results obtained in Active HDL simulation environment are available in Chapter 6. Xilinx ISE 7.1i supported by Modelsim [15] is used to synthesize the Floating-point Arithmetic unit. The target FPGA was Xilinx's Virtex II Pro [17]. This appendix provides the crucial part of synthesis results.

Device utilization summary:

The following table gives utilization summary of Floating-Point Arithmetic unit which is synthesized on Xilinx Virtex II Pro FPGA kit using Xilinx ISE 7.1i.

Target Device : xc2vp100

Table C.1 Device utilization for FPU

Logic Utilization	Used	Available	Utilization
Number of Slices:	7687	44096	17%
Number of Slice Flip Flops:	6712	88192	7%
Number of 4 input LUTs:	14145	88192	16%
Number of bonded IOBs:	163	1040	15%
Number of MULT18X18s:	16	444	3%
Number of GCLKs:	8	16	50%

Timing Summary:

Speed Grade: -5

Minimum period: 107.329ns (Maximum Frequency: 9.317MHz)

Minimum input arrival time before clock: 16.416ns

Maximum output required time after clock: 15.594ns

Maximum combinational path delay: 6.407ns

Rest of the tables shows utilization summary of sub units of FPU.

Table C.2 Device utilization for Stack Register

Logic Utilization	Used	Available	Utilization
Number of Slices:	775	44096	1%
Number of Slice Flip Flops:	776	88192	0%
Number of 4 input LUTs:	828	88192	0%
Number of bonded IOBs:	193	1040	18%
Number of GCLKs:	3	16	18%

Table C.3 Device utilization for Store (part of stack)

Logic Utilization	Used	Available	Utilization
Number of Slices:	42	44096	0%
Number of Slice Flip Flops:	38	88192	0%
Number of 4 input LUTs:	48	88192	0%
Number of bonded IOBs:	122	1040	11%
Number of GCLKs:	1	16	6%

Table C.4 Device utilization for Load (part of stack)

Logic Utilization	Used	Available	Utilization
Number of Slices:	24	44096	0%
Number of Slice Flip Flops:	41	88192	0%
Number of 4 input LUTs:	5	88192	0%
Number of bonded IOBs:	118	1040	11%
Number of GCLKs:	1	16	6%

Table C.5 Device utilization for Decoder Unit

Logic Utilization	Used	Available	Utilization
Number of Slices:	18	44096	0%
Number of Slice Flip Flops:	16	88192	0%
Number of 4 input LUTs:	32	88192	0%
Number of bonded IOBs:	26	1040	2%
Number of GCLKs:	1	16	6%

Table C.6 Device utilization for Addition Unit

Logic Utilization	Used	Available	Utilization
Number of Slices:	1330	44096	3%
Number of Slice Flip Flops:	1709	88192	1%
Number of 4 input LUTs:	2249	88192	2%
Number of bonded IOBs:	200	1040	19%
Number of GCLKs:	1	16	6%

Table C.7 Device utilization for Multiplication Unit

Logic Utilization	Used	Available	Utilization
Number of Slices:	615	44096	1%
Number of Slice Flip Flops:	664	88192	0%
Number of 4 input LUTs:	1060	88192	1%
Number of bonded IOBs:	200	1040	19%
Number of MULT18x18s:	16	444	3%
Number of GCLKs:	1	16	6%

Table C.8 Device utilization for Division Unit

Logic Utilization	Used	Available	Utilization
Number of Slices:	2916	44096	6%
Number of Slice Flip Flops:	913	88192	1%
Number of 4 input LUTs:	5571	88192	6%
Number of bonded IOBs:	201	1040	19%
Number of GCLKs:	1	16	6%

Table C.9 Device utilization for Absolute Unit

Logic Utilization	Used	Available	Utilization
Number of Slices:	55	44096	0%
Number of Slice Flip Flops:	96	88192	0%
Number of 4 input LUTs:	4	88192	0%
Number of bonded IOBs:	167	1040	16%
Number of GCLKs:	1	16	6%