DESIGN AND IMPLEMENTATION OF RECONFIGURABLE FLOATING POINT ARITHMETIC UNIT

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree of

MASTER OF TECHNOLOGY

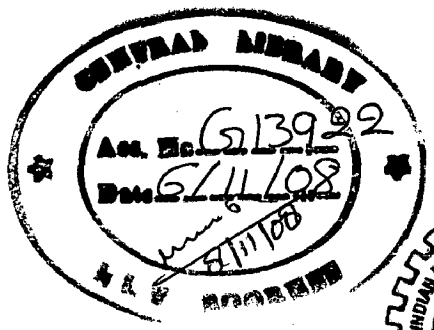
in

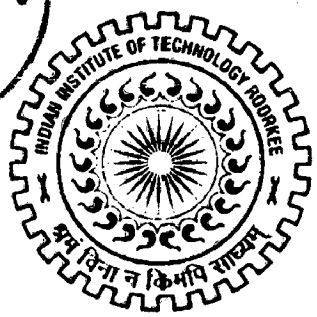
ELECTRONICS AND COMPUTER ENGINEERING

(With Specialization in Semiconductor Devices & VLSI Technology)

By

P.S.SUREKHA





DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE -247 667 (INDIA)

JUNE, 2008

Student Declaration

I hereby declare that the work being presented in the dissertation report titled " Design and Implementation of Reconfigurable Floating Point Arithmetic Unit" in partial fulfillment of the requirement for the award of the degree of Master of Technology in Semiconductor Devices and VLSI technology, submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, is an authenticate record of my own work carried out under the guidance of Dr. R.C.Joshi, Professor, and Dr.A.K.Saxena, Professor ,Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee.

I have not submitted the matter embodied in this dissertation report for the award of any other degree.

Surekha

Dated: 30/06/08 **(P. S.Surekha)**

Place: IIT Roorkee.

CERTIFICATE

This is to certify that above statements made by the candidate are correct to the best of my knowledge and belief.

Dr. R.C.Joshi 30/6

Professor

Dr.A.K.Saxena 30/6

Professor

Department of Electronics and Computer Engineering,

IIT Roorkee, Roorkee -247667 (India).

Dated: 30·6·08

Place: IIT Roorkee.

Acknowledgements

I am thankful to Indian Institute of Technology Roorkee for giving me this opportunity. It is my privilege to express thanks and my profound gratitude to my supervisor Dr.R.C.Joshi, Professor and Dr.A.K.Saxena for their invaluable guidance and constant encouragement throughout the dissertation. I was able to complete this dissertation in this time due to constant motivation and support obtained from Dr. R.C.Joshi.

I am also grateful to the staff of Sponsored Research Laboratory and VLSI Laboratory for their kind cooperation extended by them in the execution of this dissertation. I am also thankful to all my friends who helped me directly and indirectly in completing this dissertation.

I am thankful to Mr.Solomon Raju, scientist CEERI, Pilani for his constant encouragement in my work. I am grateful to Mr. Hari Krishna Boyapati and Mr. Babuji for being excellent peers and creating a congenial environment for work.

Most importantly, I would like to extend my deepest appreciation to my parents and brother for their love, encouragement and moral support. Finally I thank God for being kind to me and driving me through this journey.

Surekha

(P.S.SUREKHA)

Abstract

In recent years computer applications have increased in their computational complexity. The industry wide usage of performance benchmarks such as SPECmarks forces processor designers to pay particular attention to implementation of the floating point unit or FPU. Special purpose applications such as digital signal processing, audio processing and many real time applications placed further demands on processors with floating point unit. Unfortunately, the huge hardware resources occupied by these floating-point arithmetic units make it difficult to house a large number of units in a single FPGA. In this work we present the partial reconfiguration technique for the implementation of floating point arithmetic unit (FP-AU) which makes FP-AU with less resource utilization and flexible to operate in a rapidly changing environment. The hardware resources occupied by this unit have been reduced through time-sharing them between modules. Partial reconfiguration is the ability of certain devices (FPGAs) to reconfigure only selected portions of their programmable hardware while other portions continue to operate undisturbed. A FPGA can be partially reconfigured using a partial bitstream. We can use such a partial bitstream to change the structure of one part of an FPGA design as the rest of the device continues to operate and this reduces the reconfiguration time also.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

ABBREVATIONS

CTR	Compile Time reconfiguration
DSP	Digital Signal Processing
DR	Dynamic reconfiguration
EA	Early Access
FPGA	Field Programmable Gate Array
FP-AU	Floating Point Arithmetic Unit
FPU	Floating Point Unit
HDL	Hardware Descriptive Language
IEEE	Institute of Electical and Electonics Engineers
IOB	Input Output Blocks
ISE	Integrated Software Environment
ISDR	Ideal Software Defined Radio
JTAG	Joint Test Action Group
NCD	Native Circuit Description
NGD	Native Generic Database
PC	Personal Computer
PCI	Peripheral Component Interconnect
PR	Partial Reconfiguration
PRR	Partial reconfiguration Region
RTL	Register Transfer Logic
RTR	Run Time Reconfiguration
UCF	User Constraints File

VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language
XST	Xilinx Synthesis Technology
XUP	Xilinx University Programme

CHAPTER 1

INTRODUCTION

1.1 Introduction

Floating point numbers are used to represent very small to very large numbers with varying levels of precision. The mathematical operations such as addition, subtraction, multiplication and division are defined, but are slightly more complex than those for standard integer numbers. Floating point operations with high precision ranges requires more hardware resources. This requirement has to be reduced because of FP-AU real-time applications. Hence, the main objective of this thesis is to design a partially reconfigurable FP-AU, to decrease the hardware resources occupied by it and also to make it flexible to adapt to changing environment.

These partially reconfigurable modules are implemented on FPGA. The FPGAs can be configured to implement complex hardware flexible systems. FPGA reconfiguration [1] typically requires the whole chip to be reprogrammed even for the slightest circuit change and also some systems reconfiguration time adds delay to the application. Reconfiguration time of FPGA can be reduced by using partial dynamic reconfiguration. In partial reconfiguration only part of the circuit is reconfigured by time sharing the hardware resources at run-time, thus saving reconfiguration time and hardware resources.

The IEEE-754 standard defines floating-point number formats, operations, exceptions and their handling. A system conforming to IEEE-754 standard can be realized in software, hardware or combination of these. This standard has been chosen for representing floating-point numbers because of its use in wide range of real time applications. The IEEE floating point standard makes floating point unit implementation [2] portable and the precision of the results predictable. A variety of different circuit structures can be applied to the same number representations, offering flexibility. The floating point unit algorithm, architecture, and bit-width adaptation

offer significant potential for optimization. In this work Virtex-2pro (XC2VP50) FPGA device has been used to implement partially reconfigurable FP-AU.

1.2 Motivation

Embedded DSP applications demand more dynamic range and higher precision to prevent overflow and improve the quality respectively. The most straightforward way to satisfy both is to use the floating-point arithmetic, where the data samples are represented in the exponent and the mantissa parts and the data are normalized for every operation. Floating point arithmetic operations require huge hardware resources so it is difficult to embed them as a FPU Co-processor for DSP applications. This motivates to use partial reconfiguration techniques to implement FP-AU through which we can reduce the hardware resources needed for FP-AU without any loss of precision and accuracy. FPGAs, the research tool, for building custom circuits now has the feature of Run Time Reconfiguration supported, which allows to use the silicon in novel ways and also the tremendous increase in capacity of field-programmable gate-arrays (FPGAs) has enabled these devices for the implementation of hardware designs for basic arithmetic operations such as addition, multiplication and division in both single and double-precision IEEE-754 formats.

1.3 Statement of the problem

The analysis presented in this thesis has many attractive features and several contributions to the current state of knowledge. The general and specific contributions of this research include the following:

1. Modeling of floating point arithmetic modules in VHDL and simulating them using ModelSim6.0d.

2. Synthesizing and Optimizing the area and performance of modules using Xilinx ISE tools.

3. Implementation of FP-AU on FPGA.

4. Design and Implementation of Partial Reconfigurable FP-AU modules.

1.4 Outline of Thesis

This dissertation proposes a model for partial reconfiguration of FP-AU. The organization of the dissertation is as follows:

Chapter 2 discusses a briefly review of IEEE 754 single precision FP-AU, FPGA basics and types of reconfigurations on FPGA.

Chapter 3 describes the design flow of module based partial reconfiguration technique. This design flow has been followed in this work to do partial reconfiguration.

Chapter 4 describes algorithms used for floating point addition/subtraction, multiplication and division to write vhdl code, synthesis of the modules using Xilinx ISE 9.1i and implementation on Xilinx Virtex-2 Pro XC2VP50 kit.

Chapter 5 describes command line partial reconfiguration implementation flow and partial reconfiguration of FP-AU.

Chapter 6 shows the simulation, synthesis and implementation results of this work.

Chapter 7 concludes the dissertation work and gives suggestions for future work.

CHAPTER 2

A BRIEF REVIEW OF FLOATING POINT NUMBER SYSTEM, FPGA AND RECONFIGURABLE SYSTEMS

2.1 Floating Point Number System

Floating point number system is used to represent real numbers. The floating-point format needs slightly more storage, so when stored in the same space, floating-point numbers achieve their greater range at the expense of slightly less precision. The IEEE 754 Standard for Binary Floating-Point Arithmetic which is most widely-used standard for floating-point computation, and is followed by many CPU and FPU implementations. The standard defines formats for representing floating-point numbers and special values together with a set of floating-point operations that operate on these values.

General layout

IEEE 754 floating-point represents the number of bits using three fields. This representation is

$$V = (-1)^{sign} \times F \times 2^{E} \qquad (2.1)$$

Sign bit: This bit is used to indicate whether the number is positive or negative. '0' denotes a positive number, '1' denotes a negative number. Flipping the value of this bit flips the sign of the number.

Exponent (E): The exponent is that part 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. Bias is the value added to avoid storing of negative exponents. The exponent is biased by $(2^{e-1}) - 1$, where e is the number of bits used for the exponent field. In the case of single precision it is 127.

Biased-Exponent, e=E + bias *(2.2)*

Significand (F): It represents the fractional part of the number .It is the normalized because the exponent is adjusted so that the leading bit is always a 1. So, it does not have to be stored, and gives one more bit of precision. The length of the significand determines the precision to which numbers can be represented.

Special values

The IEEE 754 standard supports some special values like positive zero, negative zero, positive infinity, negative infinity and Not a Number (NaN) as given in Table 2.1

Table 2.1: Special values in IEEE 754 format

Name	Exponent	Fraction	Sign	Exp bits	Fractional 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

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.

Invalid operation

The invalid operation exception is signaled if an operand is invalid for the operation 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 $\pm \infty / \pm \infty$

Division by zero

When a nonzero number is divided by zero (the divisor must be *exactly* zero), a "zerodivide" event occurs, and the result is set to infinity of the appropriate sign.

Overflow

The overflow exception is signaled whenever the result exceeds the maximum value that can be represented due to 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. An overflow occurs as described previously, producing infinity.

Underflow

When the result of an operation has an exponent too small to represent properly, an "underflow" event occurs. The hardware responds to this by changing to a format in which the significand is not normalized, and there is no "hidden" bit i.e., all significand bits are represented. An underflow occurs as described previously, producing denormalized value or zero.

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.

2.2 FPGA

It is known that every application would be best served by custom circuitry targeted specifically for it; and, in fact, application-specific integrated circuits (ASICs) are often made in response to special needs. However custom chips are not affordable for every application. FPGAs are able to meet the above requirements by their ability to be reconfigured any number of times and also the growth in digital technology made FPGAs capable for implementing complex applications[3]. The word Field in the name FPGA refers to the ability of the gate array to be programmed for a particular function by the user instead of by the manufacturer of the device. The word array is used to denote a series of columns and rows of gates that can be configured by the end user. All FPGAs contain a regular structure of programmable basic logic cells surrounded by programmable interconnects and all these resources are configurable resources and its structure is shown in Fig 2.1

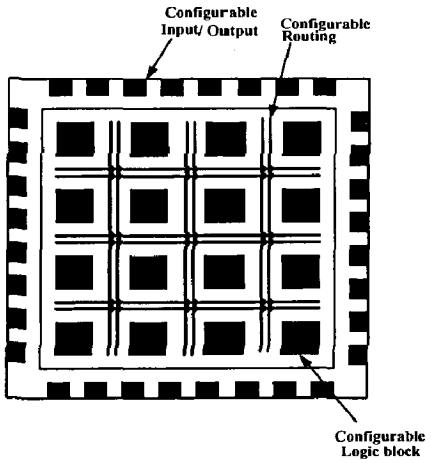


Figure 2.1: FPGA Structure

FPGAs are usually slower than their application-specific integrated circuit (ASIC) counterparts, cannot handle as complex a design, and draw more power for any given semiconductor process. But their advantages include a shorter time to market, ability to re-program in the field to fix bugs, and lower non-recurring engineering costs.

FPGA design and programming

To define the behavior of the FPGA the user provides a hardware description language (HDL) or a schematic design. Common HDLs are VHDL and Verilog. Then, using an electronic design automation tool, a technology-mapped netlist is generated. The netlist can then be fitted to the actual FPGA architecture using a process called place-and-route, usually performed by the FPGA company's proprietary place-and-route software. The designed hardware will be validated through map, place and route results and timing analysis, simulation, and other verification methodologies. Once the design and validation process is complete, the binary file generated is used to (re)configure the FPGA.

FPGA for FP-AU

With gate counts approaching ten million gates, FPGAs are quickly becoming suitable for major floating point computations. Despite their clock rate overhead with respect to contemporary general-purpose processors, these devices can be field-programmable to meet the precision requirements and operator-level parallelism of a specific computation. The balance between FPGA floating point unit resources and performance is influenced by subtle context and design requirements. Generally, implementation requirements are characterized by throughput, latency and area [4]:

- FPGAs are often used in place of software to take advantage of inherent parallelism and specialization.
- If floating point computation is in a dependent loop, computation latency could be an overall performance drawback.
- In typical FPGA designs, only a few floating point units will be on the critical path. For non-critical path units, it may be possible to trade off unit performance for reduced resource area.

FPGA for PR

PR can be implemented on devices whose behavior can be changed after fabrication i.e., on the devices which can be programmed any number of times. All user-programmable features inside a Virtex FPGA are controlled by memory cells that are volatile and must be configured on power-up. These memory cells are known as the configuration memory, and define the look-up table (LUT) equations, signal routing, input/output block (IOB) voltage standards, and all other aspects of the design. To program configuration memory, instructions for the configuration control logic and data for the configuration memory are provided in the form of a bitstream, which is delivered to the device through the JTAG, SelectMAP, serial, or ICAP configuration interface.

A programmed Virtex FPGA can be partially reconfigured using a partial bitstream. You can use partial reconfiguration to change the structure of one part of an FPGA design as the rest of the device continues to operate. Partial reconfiguration is a process of device configuration that allows a limited, predefined portion of an FPGA to be reconfigured while the remainder of the device continues to operate. This is especially valuable where devices operate in a mission-critical environment that can't be disrupted while some subsystems are being redefined.

2.3 Reconfigurable systems

2.3.1 Reconfiguration

Reconfiguration is a post-fabrication process in which processing elements are programmed spatially and temporally i.e., computation in space and time, using hardware that can adapt at the logic level to solve specific problems.

The term "reconfiguration" refers to reprogramming an FPGA after its configuration is complete. Reconfiguration can be initiated by pulsing the full chip reset pin (this method is identical to configuration), or by resynchronizing the device and sending configuration data. The latter method is only available in SelectMAP and JTAG configuration modes. To reconfigure a device in SelectMAP mode without pulsing

full chip reset pin, the BitGen persist option must be set otherwise, the Data pins becomes user I/O after configuration. Reconfiguration must be enabled in BitGen. Reconfiguration begins when the synchronization word is clocked into the SelectMAP port.

Terms used in Reconfiguration

a) ***Granularity:*** The granularity of the reconfigurable logic is defined as the size of the smallest functional unit that is addressed by the mapping tools.

Fine Grained: Fine-gained reconfigurable devices are bit-level programmable. Because of the configurability at bit-level, the configuration overhead is large. Flexibility of these devices is high. Fine-grained reconfigurable devices are perfectly suited for prototyping and implementing encryption algorithms. Example of fine grained architecture is FPGA.

Coarse Grained: Coarse-grained reconfigurable devices are flexible at word-level. Multipliers, adders, etc., are hardwired in these devices. Because only coarse functional blocks have to be configured, the configuration overhead is small. Flexibility of these devices is less compared to fine grained architectures. These coarse-grained devices achieve high performance for DSP [5] as each cell performs 16-bit or 32-bit operations.

Medium Grained: Medium-grained reconfigurable devices works at 4-bit or 8-bit data, so word-length modules require a group of cells. These devices attempts to balance performance and flexibility.

b) ***Rate of reconfiguration***

In a typical reconfigurable system, a bit stream is used to program the device at deployment time (i.e., between execution phases or during execution). A fine grained system by their own nature requires greater configuration time than more coarse-grained architectures due to more elements needing to be addressed and programmed. Therefore more coarse-grained architectures gain from potential lower energy requirements, as less information is transferred and utilized. Intuitively, the slower the

rate of reconfiguration the smaller the energy consumption as the associated energy cost of reconfiguration are amortized over a longer period of time. Partial reconfiguration aims to allow part of the device to be reprogrammed while another part is still performing active computation. Partial reconfiguration allows smaller reconfigurable bit streams thus not wasting energy on transmitting redundant information in the bit stream.

2.3.2 Types of Reconfiguration

There are two types of reconfiguration mechanisms, depending on the way they make use of dynamic nature of the reconfigurable device.

2.3.2.1 Compile-Time Reconfiguration

CTR [6] is the simplest and most commonly used approach for implementing applications with reconfigurable logic. The most important feature of CTR applications is that they consist of a single system-wide configuration for all the system (Fig 2.2). The FPGAs are loaded with their respective configurations before the execution of the operation, and once execution of the application starts, they remain in this configuration till the end of execution.

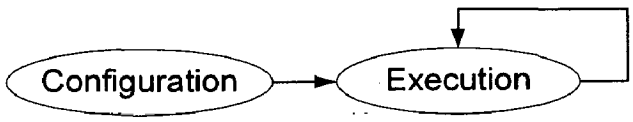


Figure 2.2: Compile Time Reconfiguration

This approach is similar to using an ASIC because the hardware does not change during the execution of the application.

2.3.2.2 Run-Time Reconfiguration

Run-Time reconfigurable applications consist of a set of time-exclusive tasks that can be downloaded into the FPGA (one at each time, or several simultaneously) using a dynamic allocation scheme. In contrast to CTR, the FPGA will probably be reconfigured more than once during the execution of an application (Fig 2.3).

Developing dynamic reconfiguration [6] is difficult because of the need for both software and hardware expertise to determine how best to partition a computation into sections to implement in hardware, how to sequence these circuit sections, and how to tie them together to produce an efficient computation. This overhead can be reduced to some extent by using dynamic partial reconfiguration which is described below.

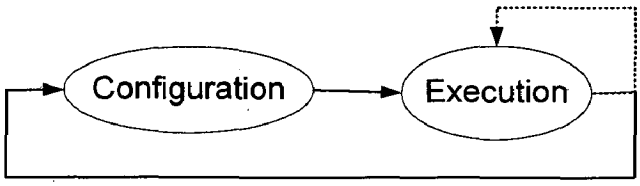


Figure 2.3: Run Time Reconfiguration.

The main advantage of RTR in front of CTR is that it allows reusing the reconfigurable device several times for the same application. To be able to do that it is necessary to partition the application into a set of configurations, but instead of using spatial exclusiveness as a criterion, this method uses time exclusiveness.

We can distinguish two classes of run-time reconfiguration schemes Global reconfiguration and Local reconfiguration which are described below.

a) Global Run-Time Reconfiguration

Here application is divided into distinct temporal phases where each phase is implemented as a single system wide configuration that occupies all system FPGA resources. In this case, reconfiguration time is more critical than in a CTR application. In this the reconfiguration of the FPGA is not only performed during the set-up of the system, but several times during the execution of the application.

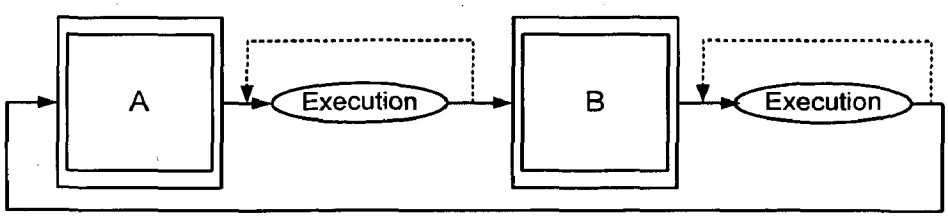


Figure 2.4: Global Run-Time Reconfiguration

Fig 2.4 shows the execution of a Global RTR application which is mapped into 2 configurations.

b) Local Run-Time Reconfiguration

It is also possible to reconfigure only subsets of the reconfigurable circuit. This approach is called partial reconfiguration or Local RTR. In this case important time-savings are made compared with a complete reconfiguration of the components, as reconfiguration is quite a time-consuming operation and with Local RTR not all the circuitry must be reconfigured to carry out changes.

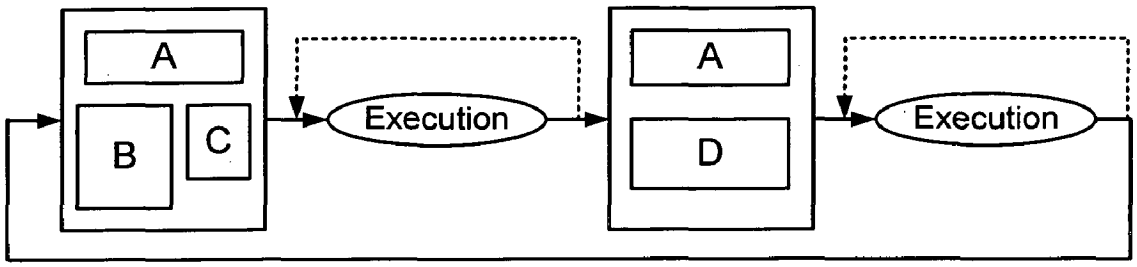


Figure 2.5: Local Run-Time Reconfiguration

Fig 2.5 shows an example of Local RTR where the application to implement consists of 4 partitions A, B, C and D. In a first step, partitions "A", "B" and "C" are loaded into the FPGA and then executed. In a second step, partitions "B" and "C" are removed and partition "D" is loaded into the FPGA, which is followed by the execution of the application.

CHAPTER 3

PARTIAL RECONFIGURATION TECHNIQUES

Reconfiguring the whole system is complicated costly in terms of overhead and may also be redundant in cases when desired functionality can be implemented by changing only a part of the circuit. The solution is to use partial reconfiguration which proves to be more efficient. Partial reconfiguration involves partitioning the hardware [7] within the platform to reduce the reconfiguration overhead.

Base and Partially Reconfigurable Regions (PRR)

Partial Reconfiguration is the ability to reconfigure a portion of an FPGA while the remainder of the design is still operational. Certain areas of a device can be reconfigured while other areas remain operational and unaffected by reprogramming.

The base region is the portion of the design that does not change during partial reconfiguration and may include logic that controls the partial reconfiguration process. PRRs contain logic that can be reconfigured independently of the base region and other PRRs. The shape and size of each PRR is defined by the user through a range constraint. Each PRR has at least one, and usually multiple, partially reconfigurable modules (PRM) that can be loaded into the PRR. Fig 3.1 illustrates a design with a single partial reconfiguration region PRR A. PRR A can be loaded with PRMs A1, A2, or A3. Each of the PRMs contains different logic for processing data passed from the static logic in the base region to the dynamic logic programmed in PRR A.

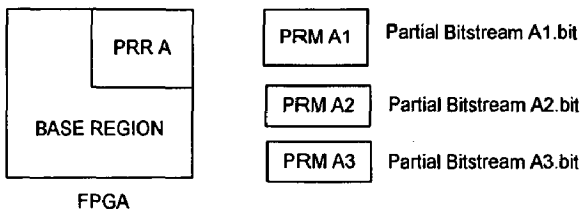


Figure 3.1 Partially Reconfiguration process

Partial Reconfiguration can be carried out in two different ways. It is possible to reconfigure part of the circuit while the operation of the other parts is interrupted. This kind of reconfiguration is called Passive Partial Reconfiguration. It is also possible in some cases, when partial reconfiguration is applied, to leave the non-reconfigured parts of the circuit in operation while other parts are being reconfigured. This method is called Active Partial Reconfiguration. By using Active Partial Reconfiguration time savings are made by lowering the reconfiguration time compared with a complete reconfiguration of the components.

Partial reconfiguration can be carried out by using Early Access (EA) Partial Reconfiguration design flow. The EA PR design flow requires several steps that are not found in the normal FPGA design flow. The normal FPGA flow involves a single pass through the implementation tools, while the PR design flow involves implementing the base design and each PRM separately, followed by a final merge step to generate the full and partial configuration bitstreams.

3.1 Partial Reconfiguration Design Flow

Fig 3.2 describes the Early Access Partial Reconfiguration Design Flow. First four Steps are similar to the non-PR design flow and remaining steps are unique to the PR design flow [8].

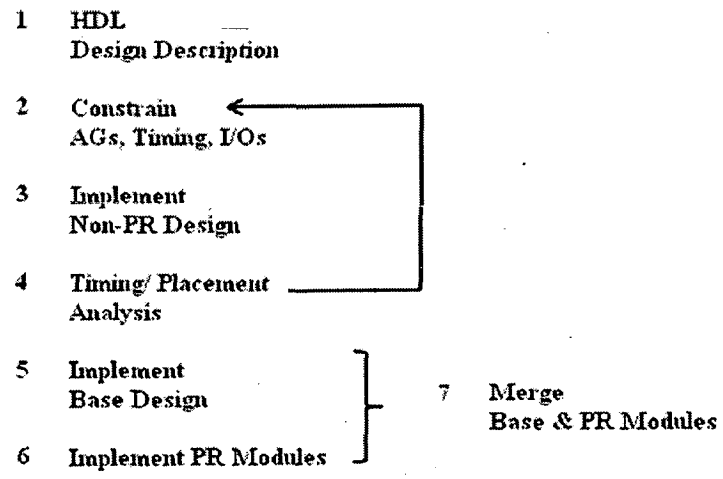


Figure 3.2 EA PR DesignFlow

- **HDL design description and synthesis:** The first step of the EA PR design flow is defining the HDL description of the design and then synthesizing that description. The EA PR design flow supports design descriptions in either VHDL or Verilog, and synthesis with any tool currently supported by ISE software designed by Xilinx.

- **Set Design Constraints:** After the HDL design description is synthesized, the next step is to place constraints on the design for place and route.

- **Implement the Non-PR Design:** While not a required step, implementing the design using the non-PR ISE implementation flow before moving to PR design implementation is recommended. This step is crucial for design debug and this gives an idea about initial timing and placement analysis, and helps in determining the best bus macro locations.

- **Timing/Placement Analysis**: The next step is to analyse both the timing and placement of the design. Timing and placement analysis is critical in establishing the best PR region shape, size, and location. During this step, designers determine whether the bus macros are placed effectively, and whether the PR region shape and location allow the tools to meet timing requirements.

- **Implement the Base Design:** After timing and placement analysis is complete, the base design must be implemented.

- **Implement PR Modules**: After the base design is implemented, each PRM must also be implemented.

- **Merge**: The final step in the PR design flow is to merge the top, base, and PR modules. During the merge step, a complete design is built from each PRM and the base design. As many complete bitstreams and partial bitstreams are created as there are PRMs (i.e. one partial bitstream for each PRM and one full bitstream for the PRM merged with the base design).

3.2 Types of Partial Reconfiguration

Partial reconfiguration is divided into two types [9]

1) Module-Based Partial Reconfiguration

2) Difference-Based Partial Reconfiguration.

1. Module-Based Partial Reconfiguration: In this method entire reconfigurable module is modified while leaving base region unaffected. Modular Design is best used for large designs that can easily be partitioned into self-contained modules. It is also used when communication is needed between modules.

2. Difference-Based Partial Reconfiguration: This method of Partial Reconfiguration is accomplished by making a small change to a design (normally done in FPGA_Editor), and then by generating a bitstream based on only the differences in the two designs. Switching the configuration of a module from one implementation to another is very quick, as the bitstream differences can be extremely smaller than the entire device bitstream. This method is very useful for implementing modules which differ by only little changes in their coding.

In this work we have followed module based partial reconfiguration and it is described in the following section.

3.2.1 Module Based Partial reconfiguration

In this the modules which are to be reconfigured are called reconfigurable modules while rest of the device remains in active operation. Implementation Using Modular Design: The partial reconfiguration implementation process is broken down into three main phases:

- **Initial Budgeting Phase** - Creating the floor-plan and constraints for the overall design.
- **Active Module Phase** - Implementing each module through the place and route process.

- **Final Assembly Phase** - Assembling individual modules together into a complete design.

Initial Budgeting Phase Details

Initial budgeting operations should be done in the top or initial folder of the recommended project directory structure. The initial budgeting phase has the following main steps:

a) The floor-planning of module areas:

- Have a set width that is always a multiple of four slices .
- Is always the full height of the device.
- Are always placed on an even four-slice boundary.
- Attach partial reconfiguration flow-specific properties to the area groups in the .ucf file

b) The floor-planning of all IOBs:

- Shall be wholly contained within the "columnar space" of their associated reconfigurable module. No intermixing between columnar regions is allowed.
- All IOBs must be locked down to exact sites.

c) The floor-planning of all global logic:

- Logic that is not part of a lower level module must be constrained to specific sites in the device via LOC constraints. Typically the Floor-planner tool can be used to create these constraints.
- There must be no unconstrained top-level logic.

d) LOC constraints are manually inserted for each bus macro into the .ucf file: Locate the bus macro to exactly straddle the boundary between the modules forming the communication bridge.

e) Global-level timing constraints: These are created for the overall design, using the Constraints Editor, if desired.

The output of the initial budgeting phase is a .ucf file containing all placement and timing constraints. Each module is implemented using this .ucf file, in addition to any module-specific constraint requirements.

In partial reconfiguration designs, all reconfigurable module inputs and outputs connect either to primary I/Os, global logic, or bus macros. No signals going to or from a reconfigurable module will load or source any element in another module without first passing through a bus macro. Unlike a standard modular design, a partial reconfiguration design does not have intermodule ports. In fact, if pseudo-drivers or pseudo-loads are found when viewing the design in the floorplanner, the design violates the criteria that all intermodule signals must utilize a bus macro channel. If this occurs, re-examine the HDL source and correct the problem. This .ucf file generated in this step is used during the active implementation of each module.

Active Module Phase Details

Up to this point, the design has been synthesized, floorplanned, and constrained. Now implementation (place and route) of all modules (both fixed and partially reconfigurable) can be carried out. Each module will be implemented separately, but always in the context of the top-level logic and constraints. Bitstreams will be generated for all reconfigurable modules. This section describes the overview of how to independently implement each module. Copy the .ucf file created during the initial budgeting phase (top or initial folder) to the active implementation directories for each module.

- In each active module working directory, augment the local copy of the .ucf file with any module level timing constraints required to specify the performance requirements for that module.

- The Constraints Editor can be used to create module-level timing constraints. Run NGDBUILD, MAP, PAR BITGEN, and PimCreate for each module. This will result in a placed and routed module as well as a module-specific bitstream.

- The PimCreate process "publishes" the routed design (and associated files) to the Pims folder. This will be used during the final assembly phase later in the implementation process.

- Optionally, run ngdanno, and ngd2ver/ngd2vhdl if module-level simulation is to be done.

- Using FPGA_Editor, visually inspect the routed design to verify that routing does not expand beyond the module boundary except, of course, for signals traversing to other modules via the bus macro structures.

Final Assembly Overview

The final assembly phase is the process of combining each of the individual modules back into a complete FPGA design. The placement and routing achieved during the active implementation phase for each module will be preserved, thereby, maintaining the performance of each module.

The following section gives brief explanation of Bus macros which are used for communication between partially reconfigurable modules or between base module and partially reconfigurable modules.

Bus Macros

Bus macros provide a means of locking the routing between PRMs and the base design making the PRMs pin compatible with the base design. As a result, all

connections between the PRMs and the base design must pass through a bus macro, with the exception of the clock signal (global signals, GND and VCC).

The bus macro naming convention is as follows:

busmacro_device_direction_synchronicity_width.nmc

Where:

device = xc2vp - Virtex-II Pro

xc2v - Virtex-II

xc4v - Virtex-4

direction = r2l - right-to-left

l2r - left-to-right

b2t - bottom-to-top (Virtex-4 only)

t2b - top-to-bottom (Virtex-4 only)

synchronicity = sync - synchronous

async - asynchronous

width = wide - wide bus macro

narrow - narrow bus macro

CHAPTER 4

IMPLEMENTATION OF FPGA BASED FP-AU

The tremendous increase in capacity of field-programmable gate-arrays (FPGAs) has enabled them to implement hardware designs for basic arithmetic operations such as addition, multiplication and division in both single and double-precision IEEE-754 compliant formats. The design flow of FPGA based FP-AU will be as shown in Fig 4.1

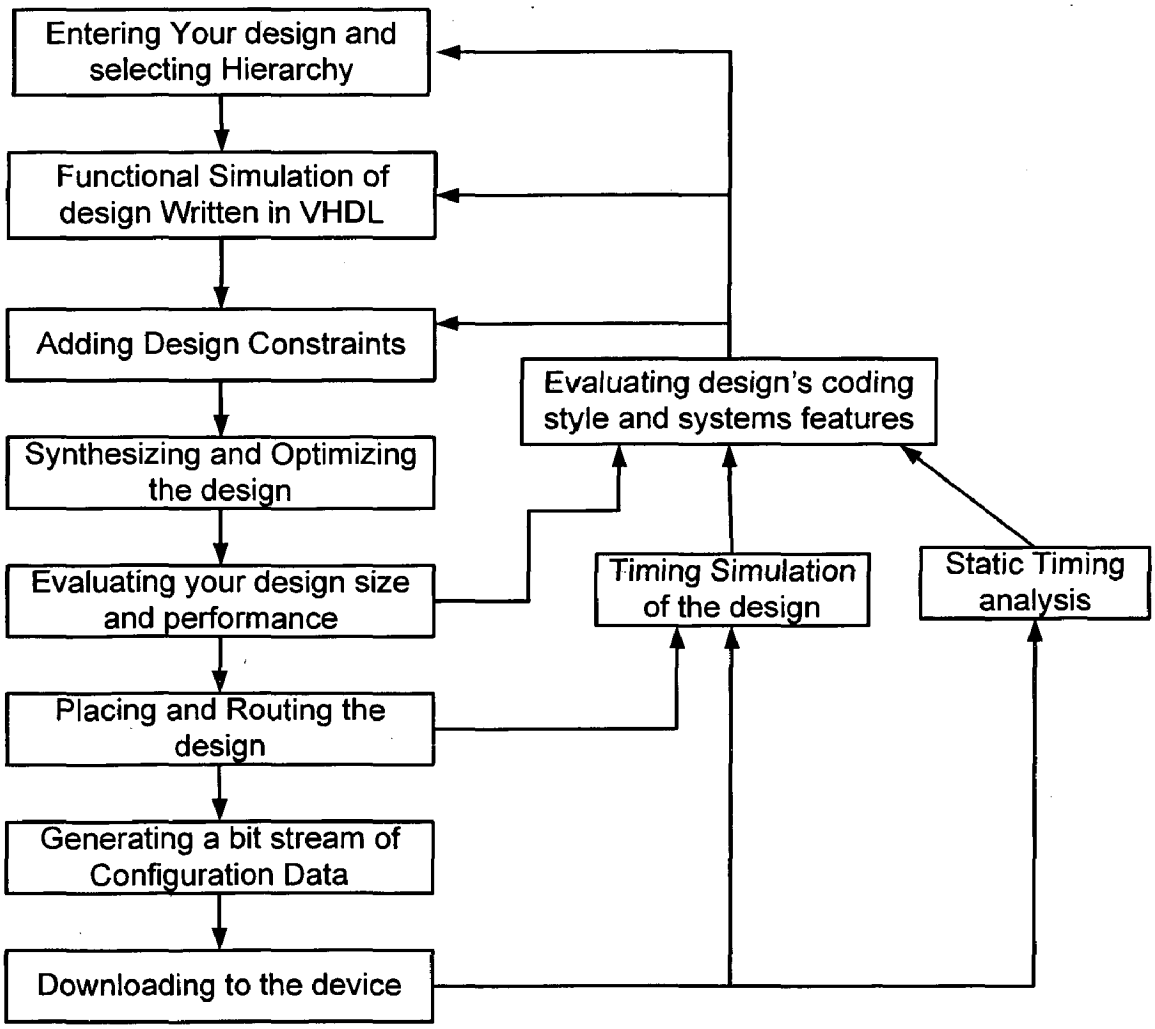


Figure 4.1: FPGA Design Flow

Design of floating point arithmetic [10] modules has been carried out using VHDL. The modules require two 32-bit inputs and give a 32-bit output. All the operations are carried out in *IEEE 754* single precision format.

Tools used

The following softwares and hardwares were used in this implementation

- Modelsim 6.0d (Simulation Tool)
- Turbo C++
- Xilinx ISE 9.1i and 9.2i with SP4 (Synthesis Tool)
- XUP Virtex-2 Pro FPGA kit.
- XUP Virtex-4 FPGA kit.

4.1 Simulation of FP-AU

4.1.1 Floating point Addition and Subtraction

For the implementation of this module, B's compliment addition and subtraction algorithm has been used. Addition of binary floating point numbers is given as follows.

- Align radix points
- Add
- *Normalize the result.*
- Rounding the result.

The above terms can be described as follows

1. Alignment: If given inputs differ in their exponent values then alignment is done to make exponents equal. The primary component in alignment is shifter. This can be described as follows: If mantissa is shifted left by one bit then decrease the exponent

by one. If mantissa is shifted right by one bit then increase the exponent by one. It de-normalizes the significand of the smaller operand such that both operand exponents are identical.

2. Addition: Depending on the respective signs of the aligned operands, one of the following operations must be executed:

- If they have the same sign, the sum *aligned-s1+aligned-s2* must be computed;
- If they have different signs, the difference *aligned-s1-aligned-s2* is computed, and if the difference is negative, the alternative difference *aligned-s2-aligned-s1* is computed.

$$result = (aligned - s1) \pm (aligned - s2) \qquad (3.1)$$

3. Normalization: It means having a non-zero MSB. In the case of binary floating point numbers the MSB is one. By normalizing the floating point values an extra bit of precision can be used in IEEE-754 format.

4. Rounding: In order to take care of the rounding precision, the round to the nearest, tie to even method is used.

4.1.2 Floating Point Multiplication

Basically, multiplication is a very simple operation as it most often reduces to multi-operand addition.

Floating point multiplication for binary numbers:

- Multiplication of operands
- Add exponents: Always add true exponents (otherwise the bias gets added in twice)
- Normalize the result: Moving the radix point one place to the left increases the exponent by 1.

In floating point Base-B numbers, a simple parallel multiplier is used [11]. Given two floating-point numbers

$$(-1)^{sign1} . s1 . B^{e1} \qquad (3.4)$$

and

$$(-1)^{sign2} . s2 . B^{e2} \qquad (3.5)$$

The floating point multiplication algorithm below corresponds to a (p+1)-by-(p+1) multiplier, an adder, and a XOR gate and generates an exact value of the product. Any type of multiplier can be used.

Algorithm:

sign: =sign1 xor sign2;

*s:=s1*s2;*

e:=e1+e2;

if s>=B then e: =e+1;

s:=s/B;

end if;

s:=round(s);

if s>=B then

e:=e+1;

s:=s/B;

end if;

In multiplication, rounding and normalization components are present which are to minimize the errors and to get accurate result. Arithmetic operations on floating point values compute results that cannot be represented in the given amount of precision. So, we must round results. There are many ways of rounding. They each have correct

uses, and exist for different reasons. The goal in a computation is to have the computer round such that the end result is as correct as possible.

4.1.3 Floating Point Division

Division is the most complex of the four basic arithmetic operations. In this work a restoring algorithm is used. In the restoring division, we compare the remainder with B. If the remainder is greater than B, we set the quotient bit to 1 and subtract B from the remainder. Otherwise, we simply shift the remainder to the left. It is called restoring, because to compare the remainder with B, we have to do a subtraction, and we restore the value to that before the subtraction if the result of the subtraction is negative.

$$q(i) = 1 \quad \text{if} \quad 2r\text{-}1 \geq B \qquad (3.6)$$

$$= 0 \quad \text{if} \quad 2r\text{-}1 < B \qquad (3.7)$$

$$r(i) = 2.r(i\text{-}1) - q(i).B \qquad (3.8)$$

Advantage of restoring algorithm is that we never have to deal with negative numbers. We always only add 1 to the LSB of the quotient bit which is always 0, which means we only need an OR gate and there will not be any carry.

Algorithm:

```
sign: = sign1 xor sign2;
s:=s1/s2;
e:=e12 e2;
if s<1 then e:=e21;
s: =s*B; end if;
```

The floating point division algorithm is almost similar to the floating point multiplication. The whole functionality of FP-AU has been described in the Fig 4.2. From the Fig 4.2 we can also say operations on any floating point numbers involve normalization and rounding. VHDL code [12] for floating point adder/subtractor,

multiplier and divider is written using algorithms described above and verified their functionality by simulating them using ModelSim 6.0d.

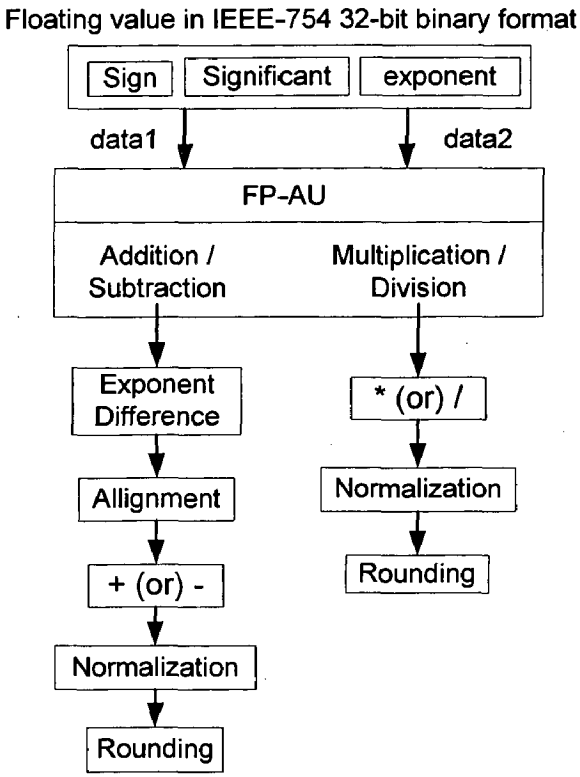


Figure 4.2: FP-AU Design Flow

4.2 Synthesis of FP-AU:

Synthesis

Using Synthesis abstract design descriptions are reduced into a lower level circuit representation, such as netlists or equations. HDLs provide the input and output of hardware synthesizers. Floating point arithmetic modules are synthesized and their netlists has been generated.

Implementation

Using Implementation logical design is converted into a physical file format that can be downloaded to the selected target device. Floating point arithmetic modules are implemented. Implementation involves following steps:

1. Translate

The Translate process merges all of the input netlists and design constraints and outputs to a Xilinx native generic database (NGD) file, which describes the logical design reduced to Xilinx primitives.

2. Map

The Map process maps the logic defined by an NGD file into FPGA elements, such as CLBs and IOBs. The output design is a native circuit description (NCD) file that physically represents the design mapped to the components in the Xilinx FPGA.

3. Place and Route

The Place and Route process takes a mapped NCD file, places and routes the design, and produces an NCD file that is used as input for bitstream generation.

Programming File Generation

This step will generate a bitstream of configuration data and it is in .bit format which can be loaded onto FPGA using iMPACT tool. The .bit file configures the FPGA to our desired functionality. Fig 4.3 shows the Xilinx ISE 9.1i [13] Project Navigator Window on which synthesis process has been carried out.

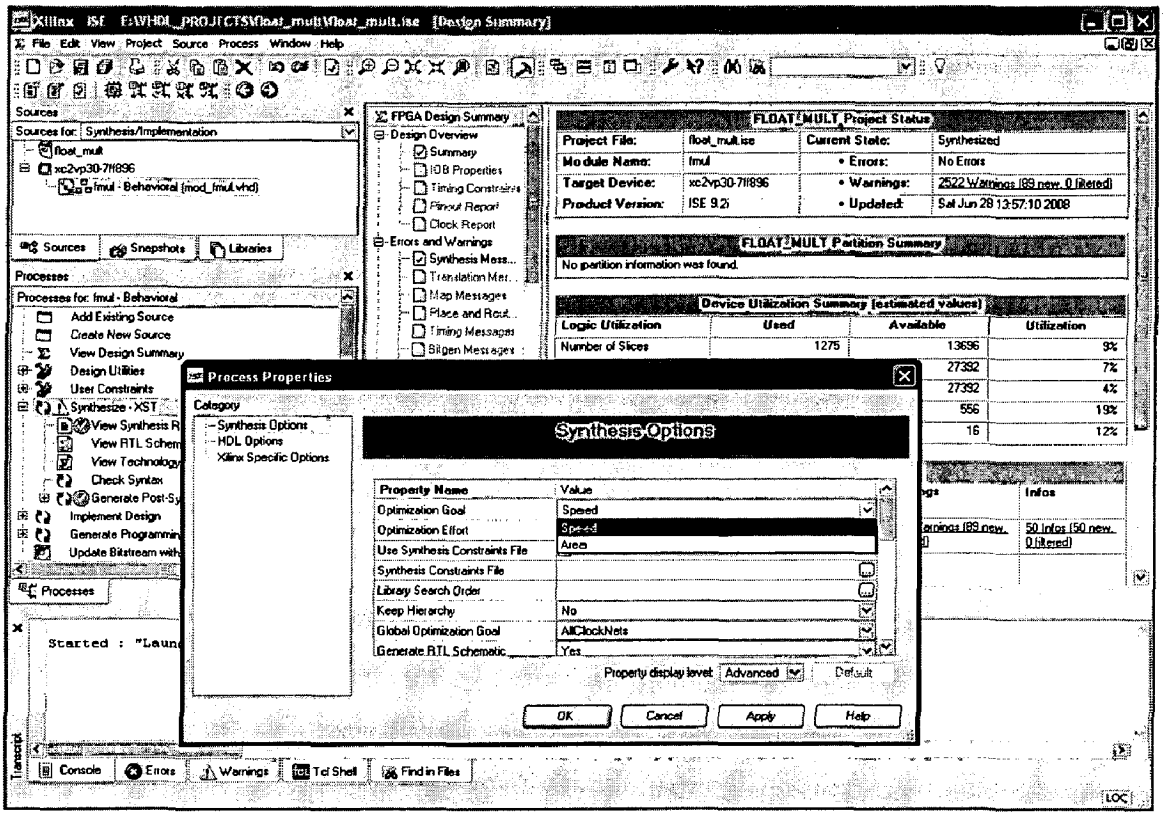


Figure 4.3: Xilinx ISE 9.2i Window

Floating point arithmetic module has been Synthesized successfully and the required files .ngo, .ncd, .ngd, .bit etc files are generated.

4.3 Implementation of FP-AU

The floating point arithmetic modules require two 32-bit inputs and one 32-output. To give 32-bit input to the floating point module, pci interface has been used. For implementing the floating point arithmetic modules on FPGA Virtex-2 Pro XC2VP50 kit [14] is used. Fig 4.4 gives the Front View of Virtex-2 Pro Board with PCI Interface.

The PCI bus uses either 32 or 64 bits of parallel data. With each clock rising edge, 32 data is transferred over the bus. Transferring 32 bits at a time translates to a very large parallel bus, using a minimum of 32 lines in addition to all the required control and

signal lines. The application FPGA communicates with the PCI controller using Rocket IO connections that support full data rate to the PCI bus.

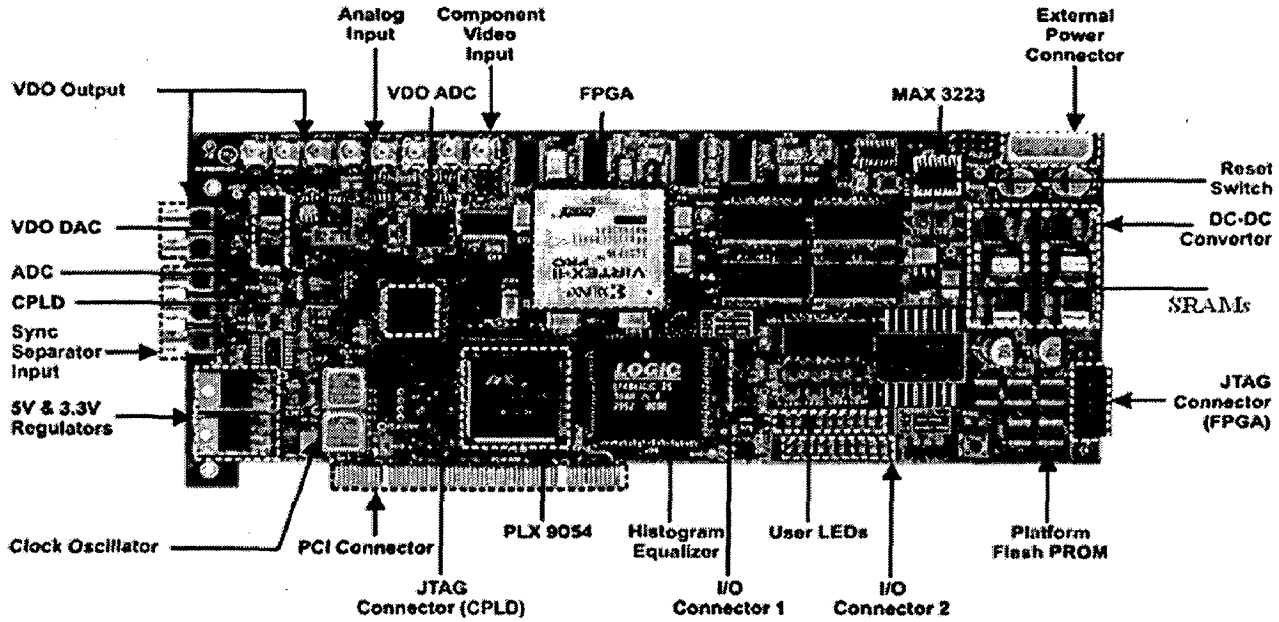


Figure 4.4 Front View of Virtex-2 Pro Board with PCI Interface

In this work we have used two coding parts. One is software coding part which is done in C++ and other is hardware coding part which is written in VHDL. These two programs combined together transfers 32-bit or 64-bit data between pci interface of FPGA and pci bus of FPGA. For doing pci interfacing with a Personal Computer, it should be loaded with drivers. These are again available from the vendor from whom the FPGA development kit is procured. Software code gives 32-bit input from PC to FPGA and giving 32-bit output from FPGA to PC. The inputs are read from files and outputs are written into files. In the hardware coding part we have combined the adder/subtractor, multiplier and divider coding part to implement them on FPGA.

Bit file has been generated, which when downloaded to the FPGA determines the FPGA's behavior. The software code transfers all the operand data for the application to the card's SRAM banks through PCI bus, Once the data transfer is complete the FPGA design is ready to start. After execution of the program we can see 32-bit output stored in the specified location in pc and it is written in text file. In this

application 32-bit binary data has been given in hex format and output is also obtained in hex format. The simulation, synthesis and implementation results of this section are shown in chapter 6.

CHAPTER 5

DESIGN OF PARTIALLY RECONFIGURABLE FP-AU

Partial reconfiguration is defined as procedure of configuring the part of the device or one of the partitioned blocks without disturbing other part operation or rest of the partitioned blocks. Partial reconfiguration is useful for systems with multiple functions that can time-share the same FPGA device resources. In such systems, one section of the FPGA continues to operate, while other sections of the FPGA are disabled and partially reconfigured to provide new functionality. Partial Reconfiguration is supported by the devices which are programmable like FPGA. So we implement partial reconfiguration on FPGA. Further the large gate count of FPGAs made them suitable for design of floating point arithmetic unit. Before designing the partially reconfigurable FP-AU, a simple design of integer adder/subtractor and LEDs right/left shift operations has been partially reconfigured, this has been extended to implement the FP-AU with partial reconfiguration techniques.

5.1 Partial Reconfiguration of Simple Integer Addition-Subtraction and LEDs shifting

In this design process first of all an integer adder/subtractor operation and LEDs right/left operations have been partially reconfigured using command line PR flow [15]. The design shown in figure below consists of two PRR, each having two RMs. The two PRR are math and led. The math processor consists of two functions addition and subtraction where as the led PRR consists of right shifting and left shifting LEDs. Interaction of math PRR has been done through HyperTerminal whereas interaction of LED PRR is achieved using push–buttons. The dynamic modules are downloaded using iMPACT software. The complete design along with bus macros for communication is shown in Fig 5.1.

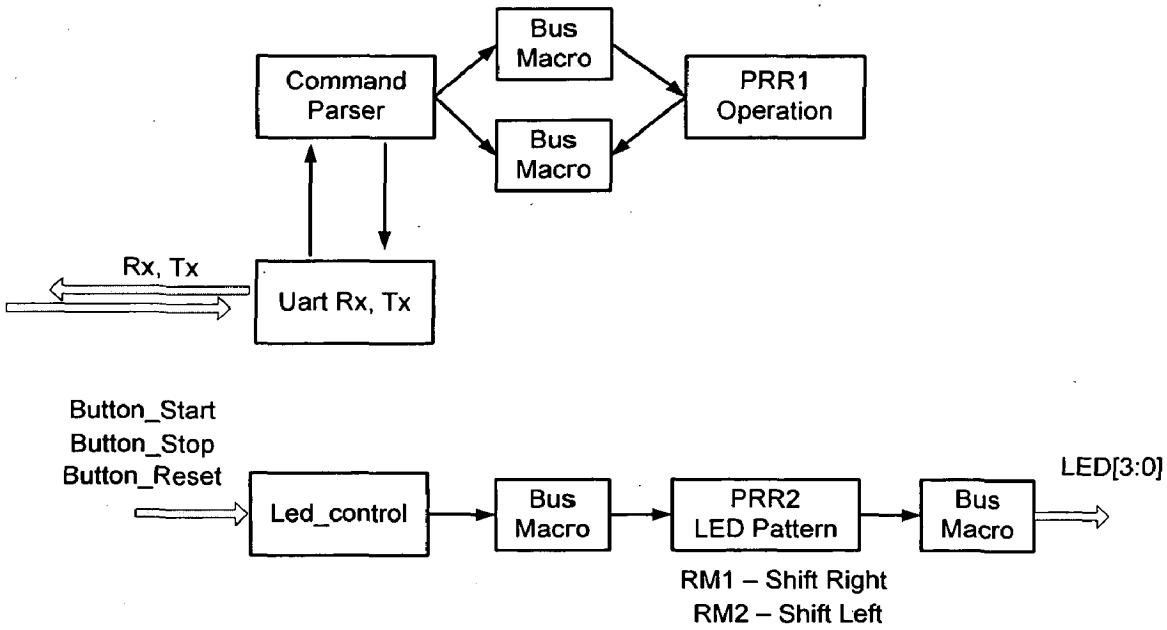


Figure 5.1: Complete System

Creating a partial reconfiguration design requires the creation and implementation of the design within a set of specific guidelines. The partial reconfiguration flow utilizes a modified form of the Xilinx Modular Design process.

Coding Part has been divided as:

1. Writing the synthesizable VHDL code for top level module. This coding should include global logic, clock buffers, black box instantiations of all reconfigurable modules(in this case adder/subtractor, left shift/right shift), bus macros (if communication is there between modules) etc.

2. VHDL code for static logic that runs on FPGA continuously and may or may not control PR modules.

3. Partially Reconfigurable modules which are to be partially reconfigured.

4. top.ucf file which contains area group constraints, mode constraints and slice allocation of PR regions etc.

All the logic that should be partially reconfigured should be arranged in folders. Recommended directory structure is as shown in figure below. Fig 5.2 shows the

folders that should be present in PR design and also the files that should be included in them.

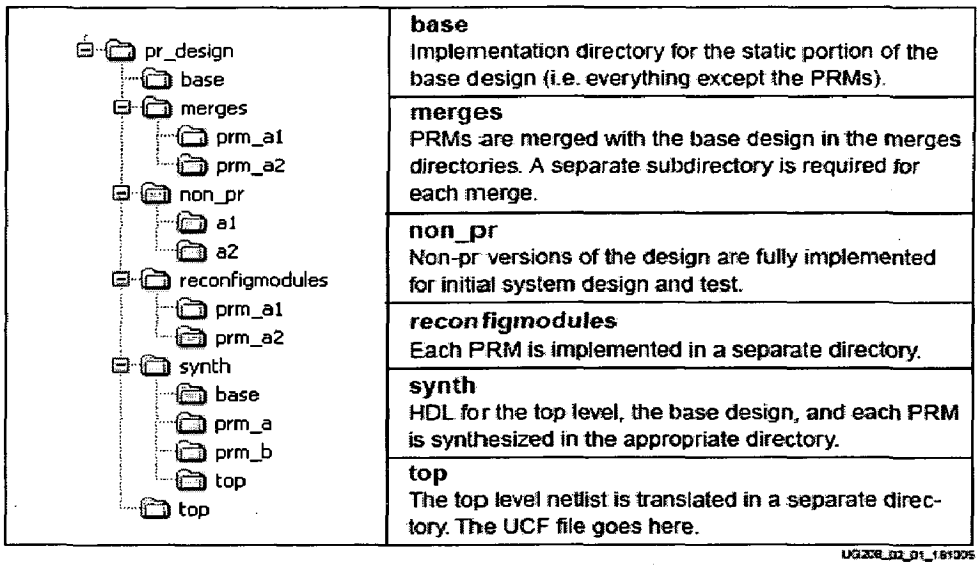


Figure 5.2: PR Directory Structure

5.1.1 Partial Reconfiguration Implementation flow in command line using ISE 9.2.4i

1. Build Flat Design: In this step we have verified the functionality of the design.
2. Synthesize all modules including RM: In this step all lower-level modules are synthesized with I/O buffers insertion OFF and the top-level with I/O buffers insertion selected.
3. Build Top-Level Design: In this step top.ngd file which will be used in next step is generated.
4. Build Static Design: In this step top_routed.ncd, static.used files are generated. The top_routed.ncd file contains the implemented static design. The static.used file contains routes used by static logic in PRR, the information needed during the RM implementation step.

5. Build RM Design: In this step top_routed.ncd file has been generated for reconfigurable modules.

6. Assemble Static Design: This step is used to assemble the static modules as well as desired one RM for each PRR into a design that will be loaded when the FPGA is configured. Static_full.bit file is generated which contains adder and right shift operations. Ag_reconfig_LEDs_blank.bit and ag_reconfig_addsub_blank.bit files which can replace the LEDs and addsub PRR with blank logic is also generated in this step.

7. Generate Partial Bitstreams: In this step partial bitstreams are generated and the PRR has been reconfigured instead of the entire FPGA with bit-streams of individual reconfigurable modules one at a time i.e dataadder_partial.bit, subtractor_partial.bit, rightshift_partial.bit and leftshift_partial.bit files.

8. Testing: In this step use static_full.bit file to program FPGA and then verify the functionality using partial bitstreams. We can notice that programming is very quick reducing the reconfiguration time.

Fig 5.3 shows how partially reconfigurable regions are described on FPGA using planahead tool [16]. It also shows the bus macro placement over right corners of partially reconfigurable regions.

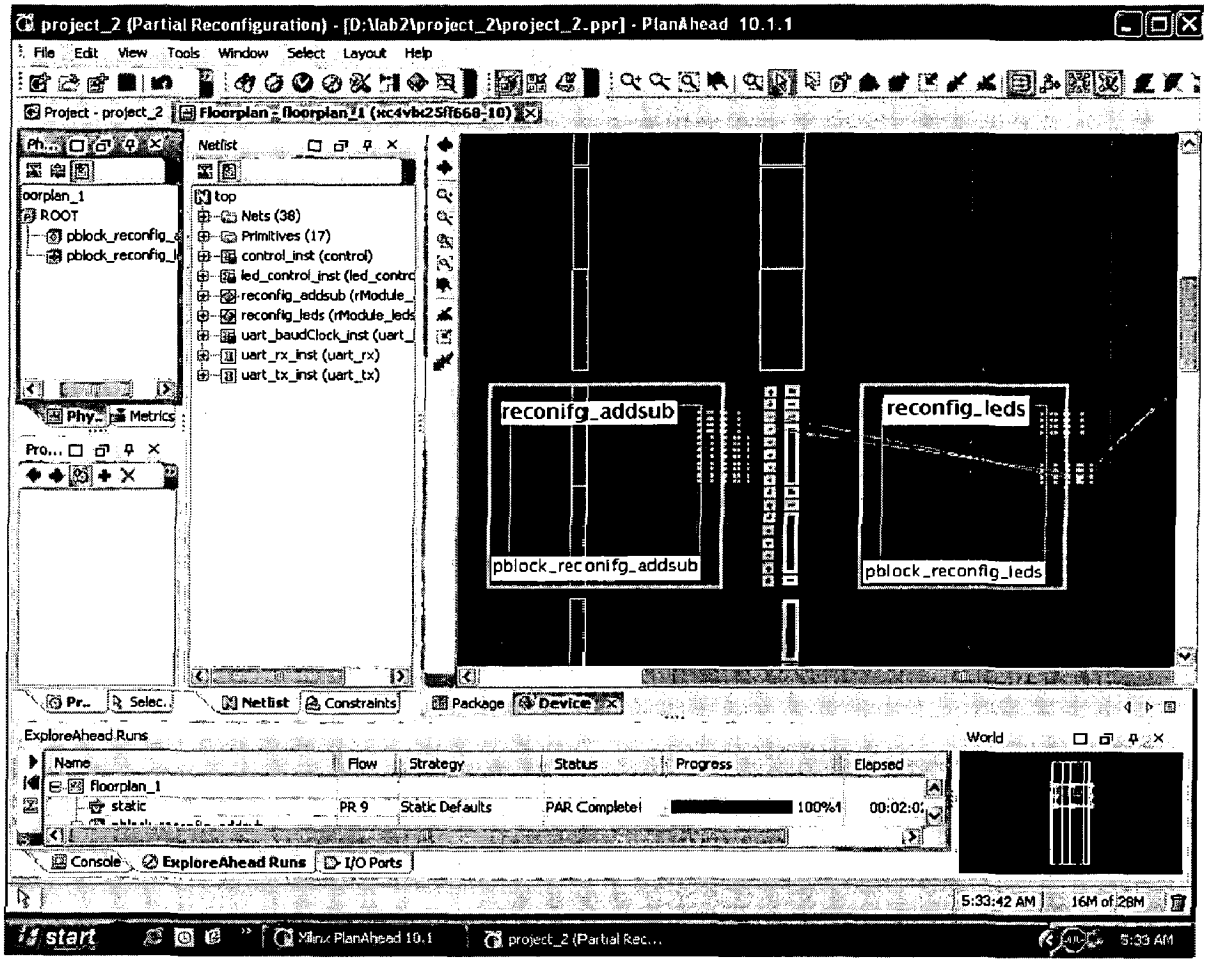


Figure 5.3: Plan Ahead window showing two Partially Reconfigurable Regions

5.2 Design of Partially Reconfigurable FP-AU

Using the example case described in previous section we have designed a partially reconfigurable system which includes one partial reconfigurable region. In this partial reconfigurable region hardware resources are time-shared between floating point arithmetic modules like adder/subtractor, multiplier and divider thus saving the area occupied by FP-AU.

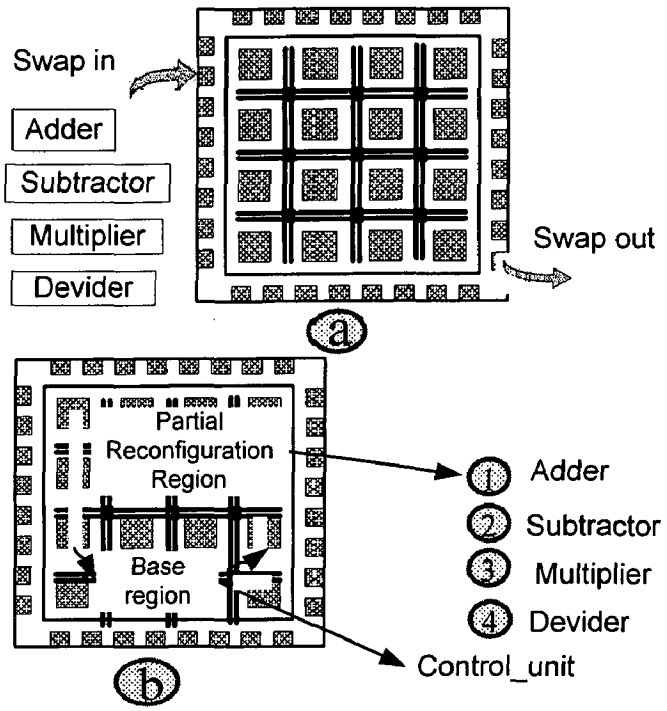


Figure 5.4: Run-time and Partial reconfigurations of Floating point modules

Partial reconfiguration design has been divided into 2 parts as shown in the Fig 5.4 above. One is Static Region or Base Region which contains the common logic which will not change between the PR modules and another is Partially Reconfigurable Region which will time-share the hardware resources among the partial reconfigurable modules.

5.2.1 Partial Reconfiguration Methodology

Design of floating point arithmetic modules is done using Command Line. In this we have used module based partial reconfiguration approach i.e., if we need to change functionality of a PR module then we have to replace the entire running module bitstream with another module bitstream. In our approach we have divided entire design into three parts Top level Design, Base design and Partially Reconfigurable modules.

1. Top level design includes:

- Black box instation of PCI interface program as static part.
- Buffers to delay the clock
- Black box instation of partially reconfigurable modules which are implemented in previous section. Give same entity name to all PR modules.
- Bus macros for communication between static logic and PR logic.

2. Base design includes:

PCI program because it is the one required to give 32-bit input and get 32-bit output from PR modules. So, this logic will run continuously on the FPGA.

3. Partial reconfigurable modules design includes:

Floating point adder/subtractor, multiplier and divider. These modules are swapped in or swapped out depending on application into the partial reconfigurable region without stopping the functionality of base region i.e., pci interface. The modules which are implemented in the previous section serve as the partially reconfigurable modules in this section.

Along with this it is important to note that we have to mention area group ranges with area constraints and modes with reconfigurable regions are assigned with proper values in the UCF file. After designing these files and putting them in appropriate folders we can follow the same procedure as that of example case to partially reconfigure FP-AU.

Partial Reconfiguration of FP-AU is partially implemented due to the unavailability of area constraints file and bus macro files for Virtex-2 Pro XC2VP50. This implementation is not possible in Virtex-4 also as it do not have external Pci-Interface to give 32-bit input and get 32- bit or LEDs to show 32-bit output. But approximately we can say that Partial Reconfiguration has reduced the number of slices required to implement floating point arithmetic unit on FPGA from 2400 slices to 1600 slices thus reducing the hardware overhead.

CHAPTER 6

RESULTS

Simulation, synthesis and implementation details of floating point adder/subtractor, multiplier and divider are presented in this section.

Simulation results:

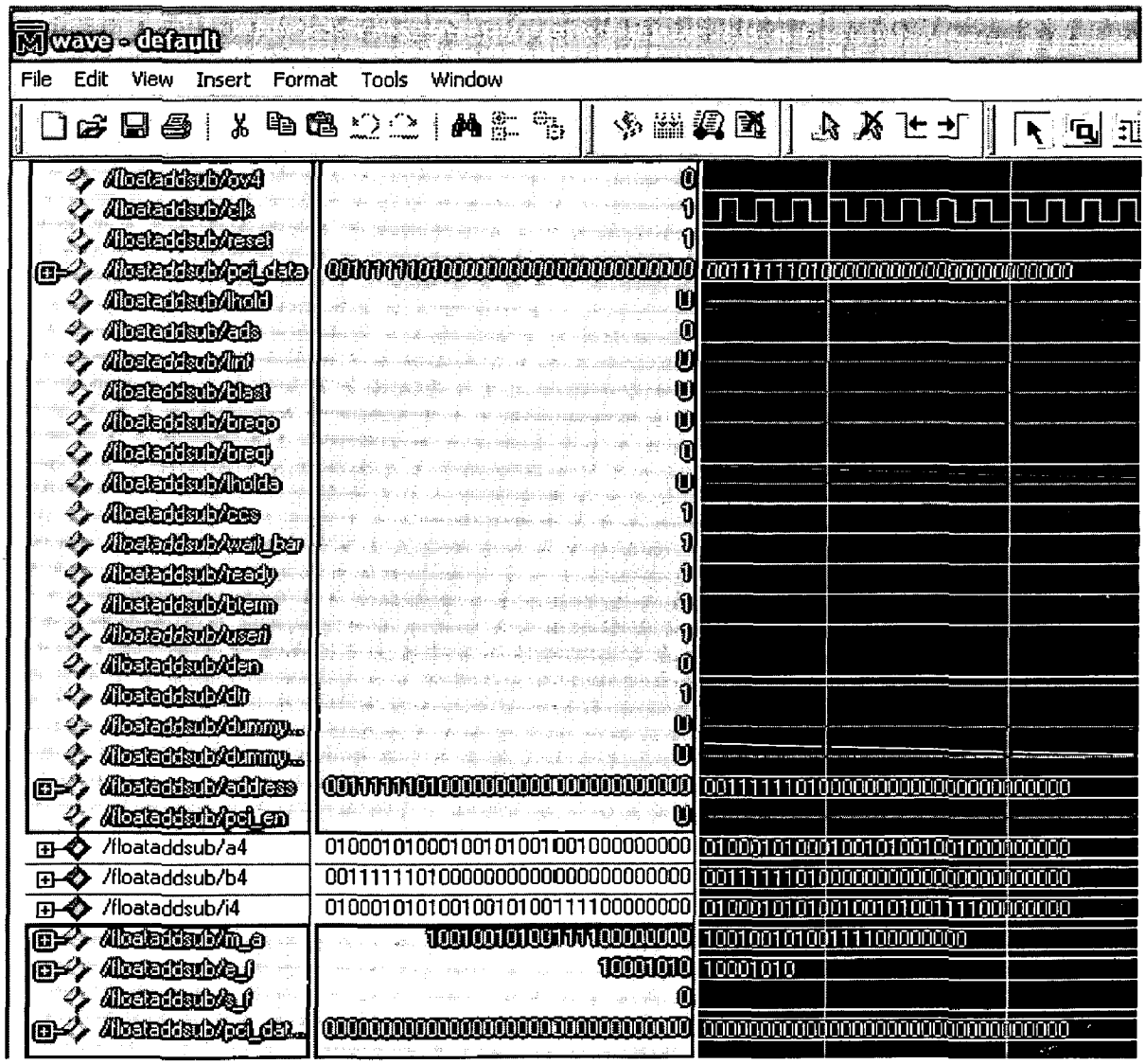


Figure 6.1: Simulation Result of Floating Point Adder/Subtractor

Simulation is done for the verifying the functionality of the modules. Firstly, the compilation is done. Then the simulation task is performed. The compilation and the simulation are done using ModelSim SE 6.0d. After the simulation is completed, we analyzed the output data of the simulation to determine if the design is correct or not. If not, we had fixed the errors and made changes in VHDL code and repeat the above steps. This process continues till all the errors are removed. The simulation results for the floating point arithmetic operations are given in the following Figs 6.1, 6.2 and 6.3.

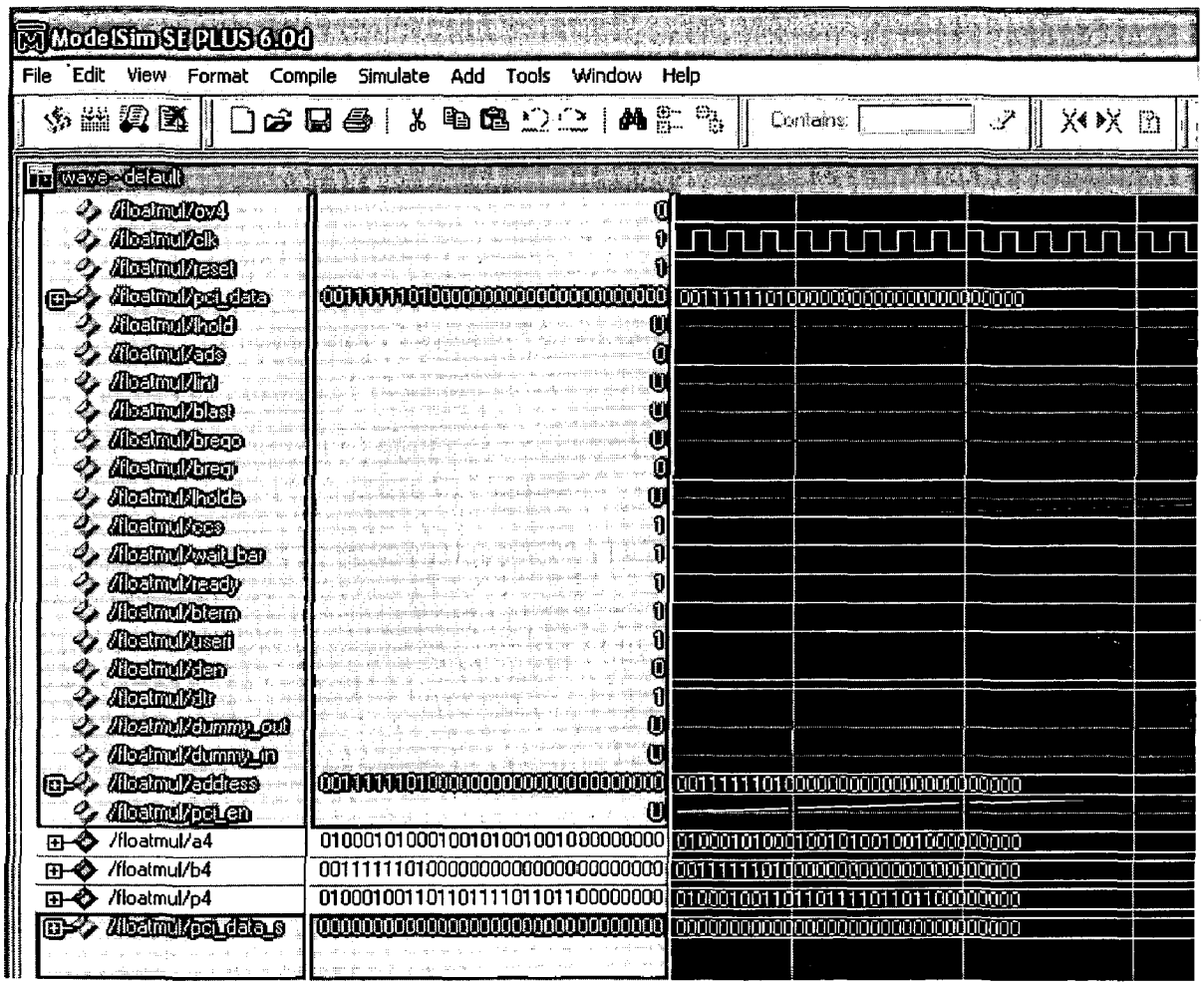


Figure 6.2 Simulation Result of Floating Point Multiplier

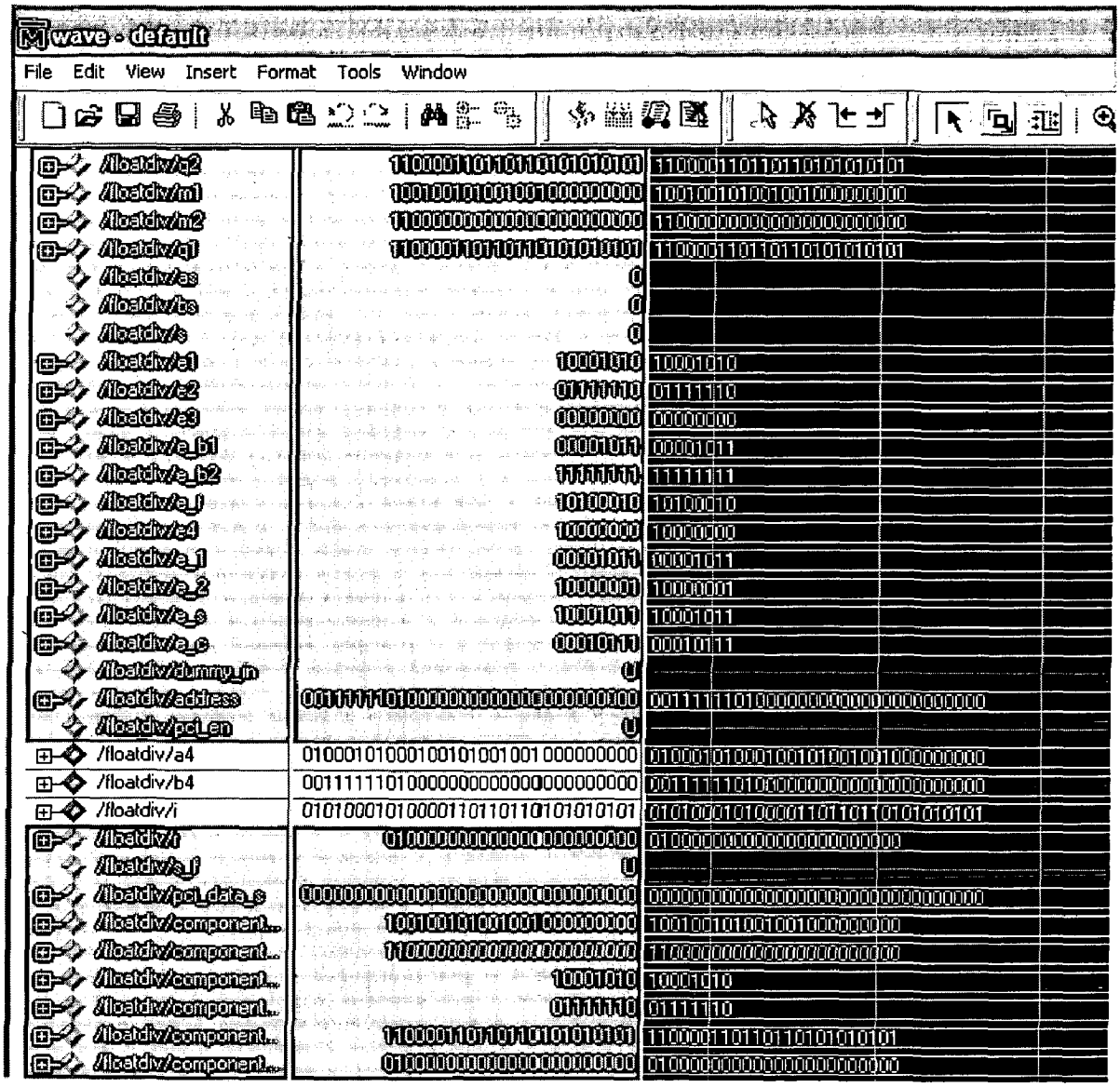


Figure 6.3 Simulation Result of Floating Point Divider

Synthesis results:

Synthesis has been carried out on Xilinx ISE 9.1i tool. Table 6.1 gives the number of hardware resources that should be allotted for floating point arithmetic modules on FPGA.

Selected Device: XC2VP50-ff1152-5

Table 6.1 Synthesis Results of Floating Point Arithmetic Unit

Name of the Module	Adder/Subtractor	Multiplier	Divider
Number of Slices (23616)	350	975	1119
Number of Slice Flip Flops (47232)	160	496	1273
Number of 4 input LUTs (47232)	610	610	1815
Number of bonded IOBs (692)	50	150	52
Number of GCLKs (16)	16	2	2

Implementation results:

Some of the results obtained by implementing floating point modules on Virtex-2 Pro XC2VP50 FPGA are given in Table 6.2. Inputs are given in Hex format and outputs are in Hex format.

Table 6.2 Results obtained from implementing on FPGA

Name of the Module	Input1	Input2	Output
Addition-Subtraction	12121231 98571234 091a0470	31310016 02091a04 abcdef10	3158800b 986b8919 abe6f787
Multiplication	12121231 98571234 091a0470	31310016 02091a04 abcdef10	03c9fd40 ffe65d32 fff7cac2
Division	12121231 98571234 091a0470	31310016 02091a04 abcdef10	6c534426 e980000 e8bf7630

Fig 6.4 shows a model through which we have given input to FPGA and read output from FPGA. As shown in the figure data1 = 0x45129200 and data2 = 0x3f400000 are the inputs given to FPGA through software program which acts as a interface between

pci bus of FPGA and local bus of pc. We can also observe that output files are stored in the specified location as given by the software program. The two .txt files shown in the Fig 6.4 are the outputs of 32-bit single precision floating point addition.

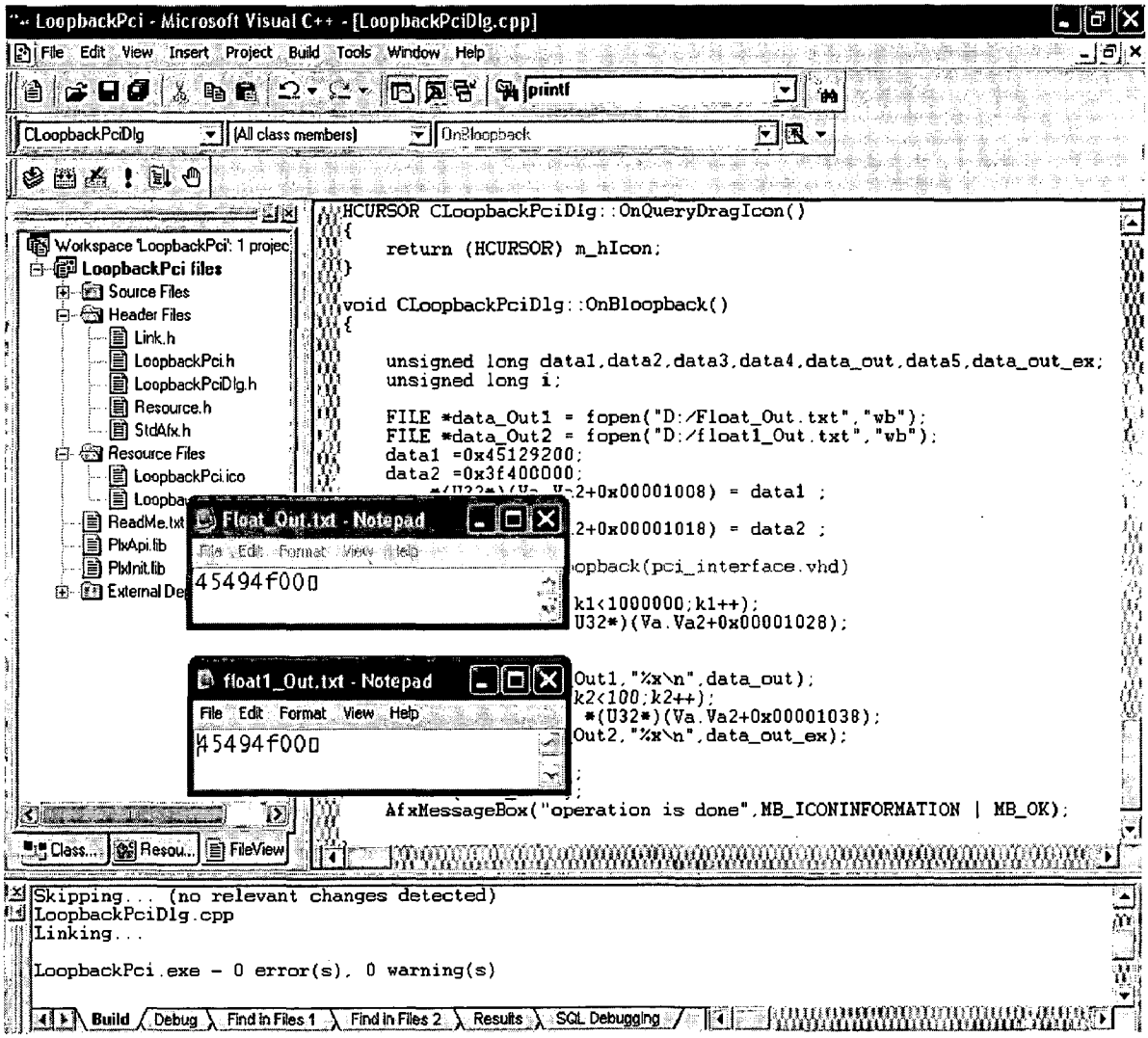


Figure 6.4 Implementation of Floating point adder on Virtex-2 Pro XC2VP50 FPGA

Partial reconfiguration Implementation results of pci-interface (static module) and partially reconfigurable modules of floating point arithmetic unit are shown in table 6.4.Total number of slices required to implement floating point arithmetic unit is approximately 2400 slices and partial reconfiguration reduced it to 1600 slices.

Table 6.4 Implementation details of partially reconfigurable FP-AU

Name of the Module	Adder/ Subtractor	Multiplier	Divider	Pci-Interface (static Module)
Number of Slices (23616)	350	975	1119	237
Number of Slice Flip Flops (47232)	160	496	1273	126
Number of 4 input LUTs (47232)	610	610	1815	133
Number of bonded IOBs (692)	50	150	52	34
Number of GCLKs (16)	16	2	2	1

CHAPTER 7

CONCLUSIONS AND FUTURE SCOPE

Conclusions:

Partial Reconfiguration of floating point arithmetic unit has been done. The contribution in this thesis is as follows:

1. Floating point arithmetic modules are implemented on Xilinx Virtex-2 Pro XC2VP50 device using PCI interface and results are shown in Table 6.2. It can be observed that the modules are working satisfactory.

2. Partial reconfiguration of integer adder and subtractor in one partially reconfigurable region and LEDs left shift and right shift in another partially reconfigurable region has been implemented on Virtex-4 ML401 device.

3. Technique developed in 2 has been extended for floating point arithmetic unit which is implemented on Virtex-2 Pro XC2VP50 and results are shown in Table 6.4.

Partial Reconfiguration has reduced the number of slices required to implement floating point arithmetic unit on FPGA from 2400 slices (approx) to 1600 slices thus reducing the hardware overhead.

However there are some limitations in the implementation. Due to the unavailability of area constraints file and bus macro files for Virtex-2 Pro XC2VP50 partial reconfiguration of floating point arithmetic unit is partially implemented. This implementation is not possible in Virtex-4 also as it do not have external PCI-Interface to give 32-bit input or to get 32- bit output or 32 LEDs to show 32-bit output. There is a small delay produced due to reconfiguration time during the computations of floating point arithmetic modules because of partial reconfiguration.

But this can be neglected when compared to area reduction achieved due to partial reconfiguration.

Future Scope:

Further area reduction can be achieved if more redundancy is followed in the coding part of floating point arithmetic modules. This redundancy based coding approach can also be used to follow difference based partial reconfiguration which almost completely reduces the reconfiguration delay and enhances the performance.

REFERENCES

[1] Yusuf. S, Luk. W, Sloman. M, Dulay. N, Lupu. E.C, Brown. G, "Reconfigurable Architecture for Network Flow Analysis," IEEE Transactions on Very Large Scale Integration (VLSI) Systems ,Volume 16, Issue 1, PP. 57 – 65, Jan. 2008.

[2] Gerardo Leyva, Gabriel Caffarena, Carlos Carreras, Octavio Nieto-Taladriz, "A Generator of High-speed Floating-point Modules," IEEE Symposium on Field-Programmable Custom Computing Machines PP. 306-307, 2004.

[3] Pedro C. Diniz, Gokul Govindu, " Design of a field-programmable dual-precision floating point arithmetic unit," International Conference in Field Programmable Logic and Applications, On PP.1-4,Aug.2006.

[4] Jian Liang, Tessier R, Mencer. O, "Floating point unit generation and evaluation for FPGAs ," in Proceedings of 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines,9-11 April 2003, PP. 185 – 194.

[5] Hung-Yueh Lin, Tay-Jyi Lin, Chie-Min Chao, Yen-Chin Liao, Chih-Wei Liu, Chein-Wei Jen, "Static floating-point unit with implicit exponent tracking for embedded DSP," in the Proceedings of the 2004 International Symposium on Circuits and Systems, ISCAS '04, Volume 2, PP.821-4 Vol.2, 23-26 May 2004.

[6] Antoni.L, Leveugle.R, Feher.M, "Using run-time reconfiguration for fault injection in hardware prototypes," in proceedings of 17th IEEE International Symposium, PP. 245 – 256, November 2002.

[7] Xiaoyao Liang, Athalye.A, Sangjin Hong, "Dynamic coarse grain dataflow reconfiguration technique for real-time systems design", in proceedings of IEEE International Symposium, Vol. 4, PP. 3511 - 3514, May 2005.

[8] Xilinx Early Access Partial Reconfiguration User Guide [online] http://www.xilinx.com/support/documentation/user_guides/ug208.pdf.

[9] Two Flows for Partial Reconfiguration: Module Based or Difference Based, Xilinx website [online] http://china.xilinx.com/support/documentation/application_notes/xapp290.pdf.

[10] Jean-Pierre, Deschamps, Gery, Jean Antoine Bioul, Gustavo D.Sutter, "Synthesis of arithmetic circuits FPGA ASIC and Embedded Systems," John Wiley and Sons publishers 2006.

[11] Israel Koren "Computer Arithmetic Algorithms" second edition, A K Peters, Ltd publication, Natick, Massachusetts, 2001.

[12] Douglas L. Perry "VHDL Programming by Example" Fourth Edition, Tata McGraw- Hill Publication, New York, 2002.

[13] Xilinx Corporation. *ISE Logic Design Tools*, 2007 http://www.xilinx.com/.

[14] Virtex II Pro Complete data sheet Xilinx website [online] http://www.xilinx.com/support/documentation/ data_sheets/ds083.pdf.

[15] Xilinx Development System Reference Guide Xilinxwebsite [online] www.xilinx.com/docsan/xilinx8/books/docs/dev/dev.pdf.

[16] Xilinx PlanAhead Methadology Guide Xilinxwebsite [online] www.xilinx.com/ise/optional_prod/planahead.htm.

APPENDIX –A

RTL diagrams

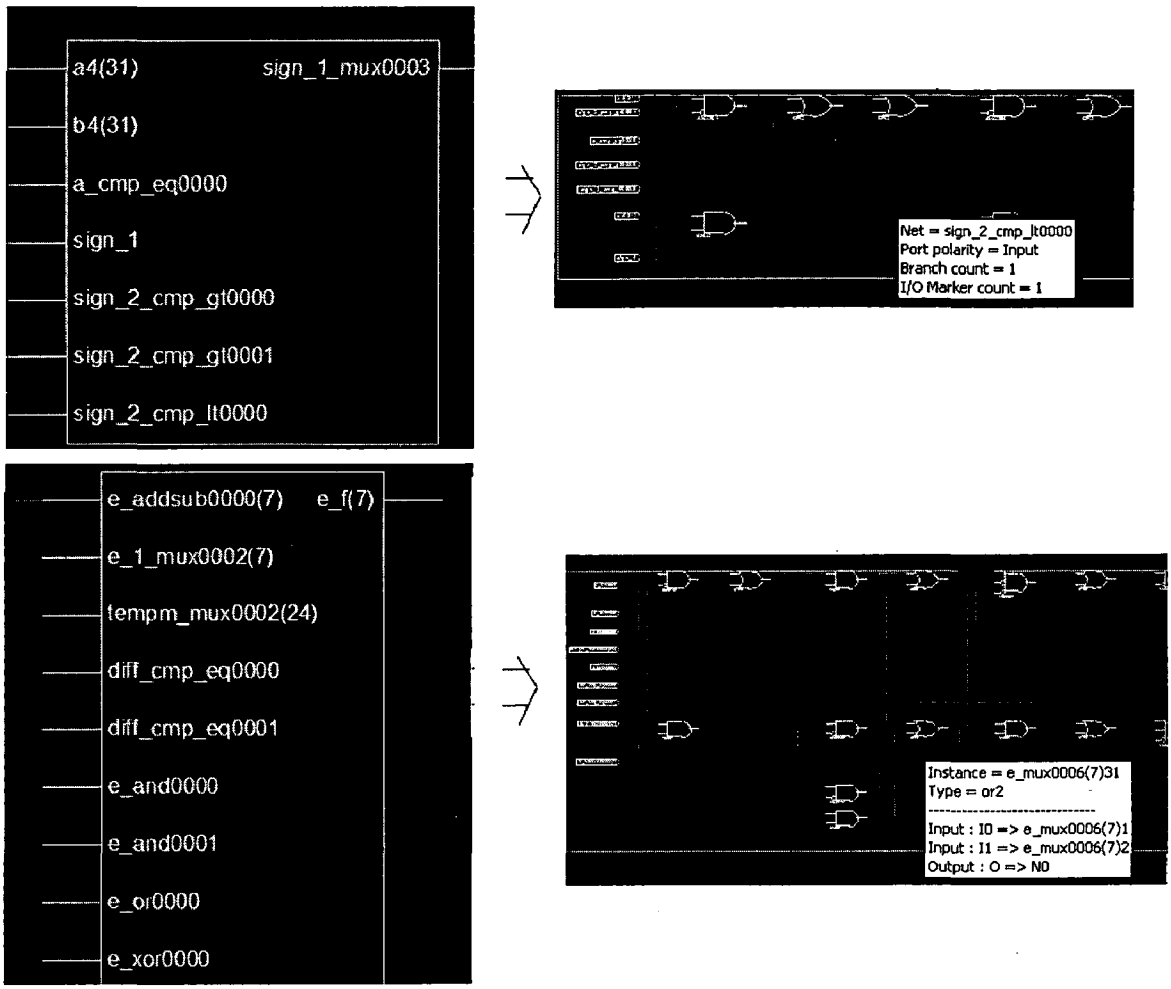


Figure A.1 RTL diagram of floating point adder/subtractor.

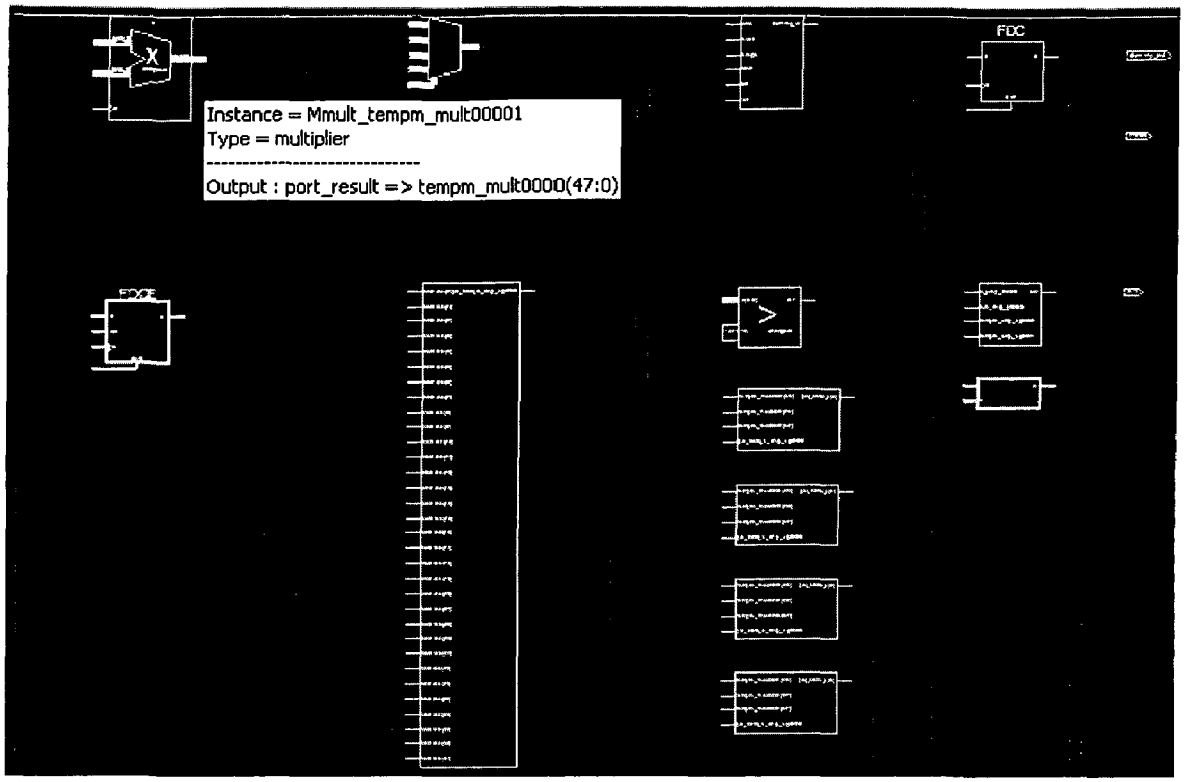
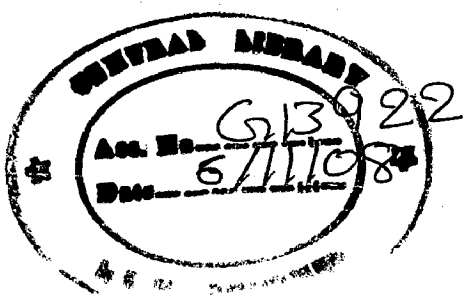


Figure A.2 RTL diagram of floating point multiplier



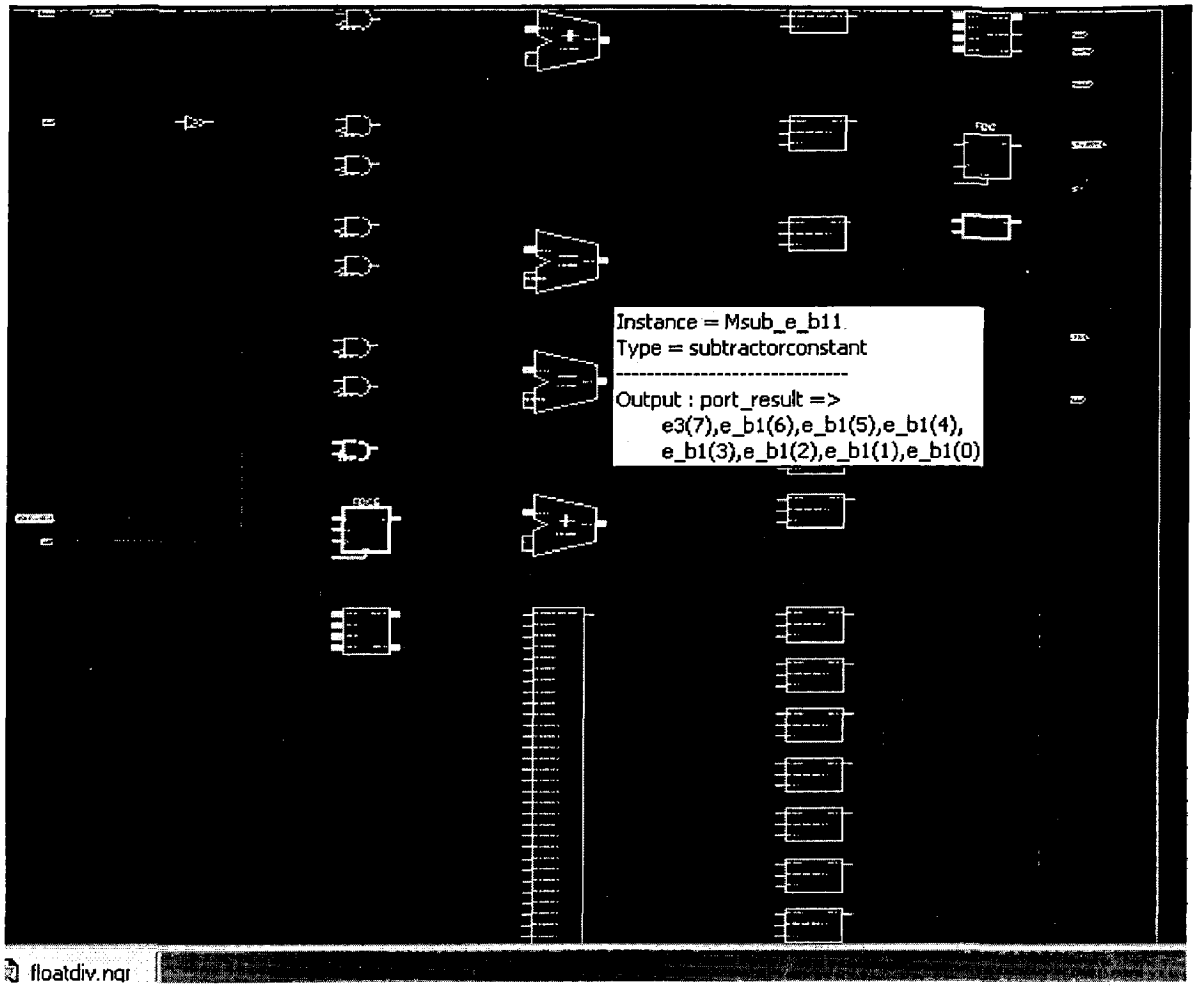


Figure A.3 RTL diagram of floating point divider

LIST OF PUBLICATIONS

- P.S.Surekha, R.C.Joshi, A.K.Saxena, "Design and Implementation of Reconfigurable FP-AU" 3rdInternational conference on Advanced Computing and Communication Technologies, Panipat, Haryana, November-2008 (Communicated).
- P.S.Surekha, B.Harikrishna, A.K.Saxena, "Design and Analysis of Dynamic Reconfigurable Systems" 4th National Conference on Machine Intelligence, Jagadhri, Haryana, August-2008 (Accepted).