EMBEDDED DSP PROCESSOR DESIGN USING COWARE PROCESSOR DESIGNER AND MAGMA LAYOUT TOOL

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology
In
Electronics and Communication Engineering
and
Electronics and Instrumentation Engineering

By
Dodani Vicky Rameshlal
and
Nikhil Kumar

Department of Electronics and Communication Engineering

National Institute of Technology Rourkela

May, 2010

EMBEDDED DSP PROCESSOR DESIGN USING COWARE PROCESSOR DESIGNER AND MAGMA LAYOUT TOOL

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology
In
Electronics and Communication Engineering
and
Electronics and Instrumentation Engineering

By

Dodani Vicky Rameshlal
and
Nikhil Kumar

Under the guidance of

Prof. K. K. Mahapatra

Department of Electronics and Communication Engineering

National Institute of Technology Rourkela

May, 2010

National Institute of Technology Rourkela

CERTIFICATE

This is to certify that the thesis entitled "Embedded DSP Processor Design using CoWare Processor Designer and Magma Layout Tool" submitted by Dodani Vicky Rameshlal (Roll No.-10609008) and Nikhil Kumar (Roll No.-10607026) in partial fulfillment of the requirements for the award of Bachelor of Technology Degree in Electronics and Communication Engineering and Electronics and Instrumentation Engineering respectively at National Institute of Technology, Rourkela is an authentic work carried out by them under my supervision and guidance.

To the best of my knowledge, the matter embodied in thesis has not been submitted to any other university/ Institute for the award of any degree or Diploma.

Prof. K. K. Mahapatra

Date:

Department of E.C.E

National Institute of Technology

Rourkela-769008

ACKNOWLEDGEMENT

This is a research project and the fact that we have been able to complete it successfully owes a lot to a number of persons associated with us during this project.

First of all, we would like to thank **Prof. K. K. Mahapatra** for giving us a golden opportunity to work on such an interesting topic and providing a thoroughly professional and research oriented environment. He also guided us nicely throughout the project period and helped us time to time with his vast experience and innovative ideas.

We wish to extend our sincere thanks to Prof. S. K. Patra, Head of our Department, for approving our project work with great interest.

Also we need to thank Mr. Sudeendra Kumar K., Mr. Ayaskanta Swain, Mr. Jagannath Mohanty and Mr. Sushant Kr. Pattanayak , whose contribution in this project has been quite significant.

Finally, a word of thanks to all of them who have been associated with us and directly or indirectly helped us during this project.

Dodani Vicky Rameshlal

Roll No.-10609008

Nikhil Kumar

Roll NO.-10607026

Department of Electronics and Communication Engineering

National Institute of Technology Rourkela

TABLE OF CONTENTS

CERTIFICATE ii

ACKNOWLEDGEMENT iii

ABSTRACT vi

LIST OF FIGURES vii

LIST OF TABLES ix

INTRODUCTION 1

1.1. Implementation of DSP Application 2

1.1.1 Implementation on General Purpose Processor (GPP) 2

1.1.2. Implementation on General Purpose DSP Processor 2

1.1.3. Implementation on Application Specific Integrated Circuit (ASIC) 2

1.1.4. Implementation on Application Specific Instruction Set Processor (ASIP) 3

1.2. DSP Processor Architecture 3

1.3. Embedded System Overview 5

1.4. DSP in an Embedded System 5

1.5. ASIP Design Flow 6

APPLICATION DESCRIPTION LANGUAGE 8

2.1. Introduction 9

2.2. LISA Modelling Fundamentals. 9

2.3. Modelling Processor Resources. 10

2.4. Modelling Instructions. 10

2.4.1. The Instruction Behavior. 11

2.4.2. The Instruction Syntax. 11

2.4.3. The Instruction Coding. 12

PROCESSOR DESIGN PLATFORM 14

3.1. Traditional Embedded System Design Flow 15

3.2. Hardware Software Co-Design Flow 16

3.3. CoWare Design Flow 17

3.4. CoWare Processor Designer 18

3.5. The Instruction Set Designer. 19

3.6. CoWare Processor Debugger 21

MAGMA LAYOUT TOOL 22
4.1. Magma Design Flow 23
4.2. Introduction to Magma Blast Create Tool 23
4.3. Introduction to Magma Blast Fusion Tool 24
Virtex-II Pro FPGA 26
5.1. Introduction. 27
5.2. XUP Virtex-II Pro Development System. 28
5.3. Chipscope Pro Tools. 30
IMPLEMENTATION OF PROCESSOR 31
6.1. Implementation of General Purpose Processor 32
6.2. Architecture Profiling and Debugging 38
SIMULATION & SYNTHESIS RESULTS 41
7.1. Introduction 42
7.2. Simulation Results 43
7.3. Synthesis Results 44
LAYOUT & FPGA IMPLEMENTATION OF THE PROCESSOR 45
8.1. Layout using Magma Tool 46
8.2. Configuring the Virtex-II Pro FPGA. 47
CONCLUSION 50
References 52

ABSTRACT

A Digital Signal Processing (DSP) application can be implemented in a variety of ways. The objective of this project is to design an Embedded DSP Processor. The desired processor is run by an instruction set. Such a processor is called an Application Specific Instruction Set Processor (ASIP). ASIP is becoming essential to convergent System on Chip (SoC) Design. Usually there are two approaches to design an ASIP. One of them is at Register Transfer Level (RTL) and another is at just higher level than RTL and is known as Electronic System Level (ESL). Application Description Languages (ADLs) are becoming popular recently because of its quick and optimal design convergence achievement capability during the design of ASIPs.

In this project we first concentrate on the implementation and optimization of an ASIP using an ADL known as Language for Instruction Set Architecture (LISA) and CoWare Processor Designer environment. We have written a LISA 2.0 description of the processor. Given a LISA code, the CoWare Processor Designer (PD) then generates Software Development tools like assembler, disassembler, linker and compiler. A particular application in assembly language to find out the convolution using FIR filter is then run on the processor. Provided that the functionality of the processor is correct, synthesizable RTL for the processor can be generated using Coware Processor Generator.

Using the RTL generated, we implemented our processor in the following IC Design technologies:

- Semi-Custom IC Design Technology

  Here, the RTL is synthesized using Magma Blast Create Tool and the final Layout is drawn using Magma Blast Fusion Tool

- Programmable Logic Device IC Design Technology

  Here, the processor is dumped to a Field Programmable Gate Array (FPGA). The FPGA used for this purpose is Xilinx Virtex II Pro.

LIST OF FIGURES

Figure 1.1 DSP Processor Architecture 4

Figure 1.2 DSP Processor in an Embedded System 6

Figure 1.3 ASIP Design Flow 7

Figure 3.1 Traditional Embedded System Design Flow 15

Figure 3.2 Hardware Software Co-Design Flow 16

Figure 3.3 CoWare Design Flow 17

Figure 3.4 CoWare Processor Designer Main Window 18

Figure 3.5 Instruction Set Designer Window 20

Figure 3.6 Processor Debugger Window 21

Figure 4.1 Magma Design Flow 23

Figure 4.2 Magma Blast Create Flow and Commands 24

Figure 4.3 Floorplanning within Magma Blast Fusion Flow 25

Figure 5.1 FPGA Block Structure 27

Figure 5.2 XUP Virtex-II Pro Development System Board Photo 28

Figure 5.3 XUP Virtex-II Pro Development System Block Diagram 29

Figure 5.4 Chipscope Pro Tools Design Flow 30

Figure 6.1 Operation Hierarchy of Implemented GPP 37

Figure 6.2 Processor Debugger for the Implemented GPP 38

Figure 6.3 Operation Profiling for the Desired Application 39

Figure 6.4 Instruction Set Designer Window of the Designed ASIP 40

Figure 7.1 The Generated HDL Code Structure 42

Figure 7.2 Simulation Results 43

Figure 8.1 Layout of the Processor 46

Figure 8.2 FPGA Design Flow 47

Figure 8.3 Chipscope Analyzer Waveform for the current Design 49

LIST OF TABLES

Table 5.1 XC2VP30 Device Features 29
Table 6.1 Modes of Implementation in CoWare Design 32
Table 6.2 Instructions of implemented GPP 35
Table 7.1 Synthesis Results 44
Table 8.1 Target Device (XC2VP30) Utilization 48

CHAPTER 1

INTRODUCTION

1.1. Implementation of DSP Application

There are various ways of implementing a DSP application. They are:

1. 1. 1 Implementation on General Purpose Processor (GPP)

Many DSP applications, with or without real-time requirements, can be implemented on a general-purpose processor (GPP). There are two reasons for implementing a DSP application on a general-purpose computer:

- To quickly supply the application to the final user within the shortest possible time.
- To use this implementation as a reference model for the design of an embedded system.

1. 1. 2. Implementation on General Purpose DSP Processor

Many DSP applications are implemented using a general-purpose DSP (off-the shelf processor). Here, general-purpose DSP stands for a DSP available from a semi-conductor supplier and not targeted for a specific class of DSP applications. A general purpose DSP has a general assembly instruction set that provides good flexibility for many applications. However, high flexibility usually means fewer application specific features or less acceleration of both arithmetic and control operations. Therefore, a general-purpose DSP is not suitable for applications with very high performance requirements. High flexibility also means that the chip area will be large. A general-purpose DSP processor can be used for initializing a product because the system design time will be short. When the volume has gone up, a DSP ASIP could replace the general-purpose processor in order to reduce the component cost.

1. 1. 3. Implementation on Application Specific Integrated Circuit (ASIC)

There are two cases when an ASIC is needed for digital signal processing. The first is to meet extreme performance requirements. In this case, a programmable device would not be able to handle the processing load. The second case is to meet ultralow power or ultra-

low silicon area, when the algorithm is stable and simple. In this case, there is no requirement on flexibility, and a programmable solution is not needed.

ASIC implementation is to map algorithms directly to an integrated circuit. Comparing a programmable device supplying the flexibility at every clock cycle, an ASIC has very limited flexibility. It can be configurable to some extent in order to accommodate very similar algorithms, but typically it cannot be updated in every clock cycle.

1. 1. 4. Implementation on Application Specific Instruction Set Processor (ASIP)

A DSP ASIP has an instruction set optimized for a single application or a class of applications. On one hand, a DSP ASIP is a programmable machine with a certain level of flexibility, which allows it to run different software programs. On the other hand, its instruction set is designed based on specific application requirements making the processor very suitable for these applications. Low power consumption, high performance, and low cost by manufacturing in high volume can be achieved. The specialization of an ASIP provides a tradeoff between the flexibility of a general purpose CPU and the performance of an ASIC. The flexibility of these processors can be achieved by many ADLs like LISA, EXPRESSION, MIMOLA etc.

An ASIP DSP has a dedicated instruction set and dedicated data types. When designing an ASIP DSP, functions are mapped to subroutines consisting of assembly instructions. When designing an ASIC, the algorithms are directly mapped to circuits. However, most DSP applications are so complicated that mapping functions to circuits is becoming increasingly difficult. On the other hand, mapping DSP functions to an instruction set is becoming more popular because the challenge of complexity is handled in both software and hardware, and conquered separately.

1.2. DSP Processor Architecture

Figure 1.1 shows a simplified block diagram of DSP processor architecture:

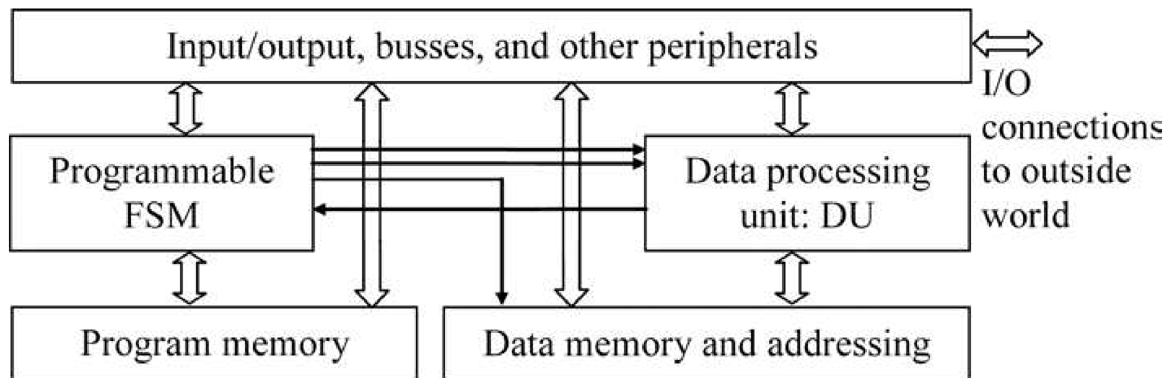

Figure 1.1 DSP Processor Architecture

As shown in the figure, a DSP processor contains five key components:

- **Program memory (PM)**: PM is used for storing programs (in binary machine code). PM is part of the control path.

- **Programmable FSM:** It is a programmable finite state machine consisting of a program counter (PC) and an instruction decoder (ID). It supplies addresses to the program memory for fetching instructions. Meanwhile, it also performs instruction decoding and supplies control signals to the data processing unit and data addressing unit.

- **Data memory and data memory addressing:** DM stores information to be processed. Three types of data are stored in DM: input/output data, intermediate data in a computing buffer (a part of the data memory), and parameters or coefficients. The data memory addressing unit is controlled by programmable FSM and supplies addresses to data memories.

- **Data processing unit (DU):** The data processing unit, or datapath, performs arithmetic and logic computing. A DU includes at least a register file (RF), a multiplication and accumulation unit (MAC), and an arithmetic logic unit (ALU). A data processing unit may also include some special or accelerated functions.

- **Input/output unit (I/O):** I/O serves as an interface for functional units connected to the outside world. I/O also handles the synchronization of external signals. Memory buses and peripherals are also included.

1.3. Embedded System Overview

An embedded system is a special-purpose computer system designed to perform one or a class of dedicated functions. In contrast, a general-purpose computer, such as a personal computer, can do many different tasks, depending on programming. An embedded system could be a component of a personal computer such as a keyboard controller, mouse controller, or a wireless modem. An embedded system could also be a digital subsystem inside a mobile phone, a digital camera, a digital TV, or in medical equipment. Except for general computers, most microelectronic systems are embedded systems. Within the specific application domain, the embedded system may have much higher performance or much lower power consumption compared to a general computer system.

1.4. DSP in an Embedded System

DSP processors are essential components in many embedded systems. One or several DSP processors consist of a DSP subsystem in an embedded system. A general embedded system, including a DSP subsystem, is shown in Figure 1.2. Such a system is also called a system on a chip (SoC) platform for embedded applications.

The system in Figure 1.2 can be divided into four parts:

- The first part is the microcontroller (MCU), which is the master of the chip or the system. The MCU is responsible for handling miscellaneous tasks, except computing for real-time algorithms.

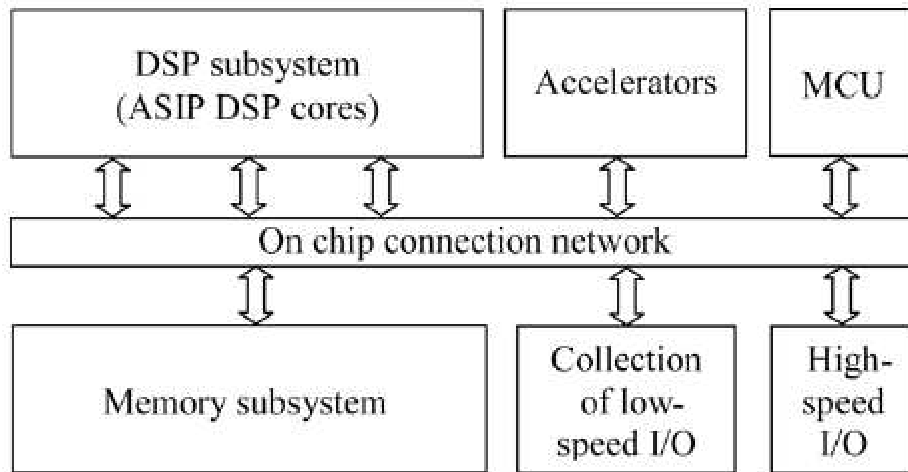

Figure1.2 DSP Processor in an Embedded System

- The second part is the ASIP DSP subsystem, which is the main computing engine of the system. All heavy computing tasks should be allocated to this subsystem.

- The third part is the memory subsystem, which supports data and program storage for the DSP subsystem and the MCU.

- The fourth part consists of peripherals including high-speed and low-speed I/Os.

1.5 ASIP Design Flow

The first and most important step in the design of a processor is the instruction set design. The instruction set design is a trade-off among a multitude of parameters including performance, functional coverage, flexibility, power consumption, silicon cost, and design time. Figure 1.3 shows the general ASIP design flow.

The ASIP design flow starts from the requirement specification and finishes after the microarchitecture design. The design of an ASIP is based mostly on experience, and it is essential to minimize the cost of design iteration.

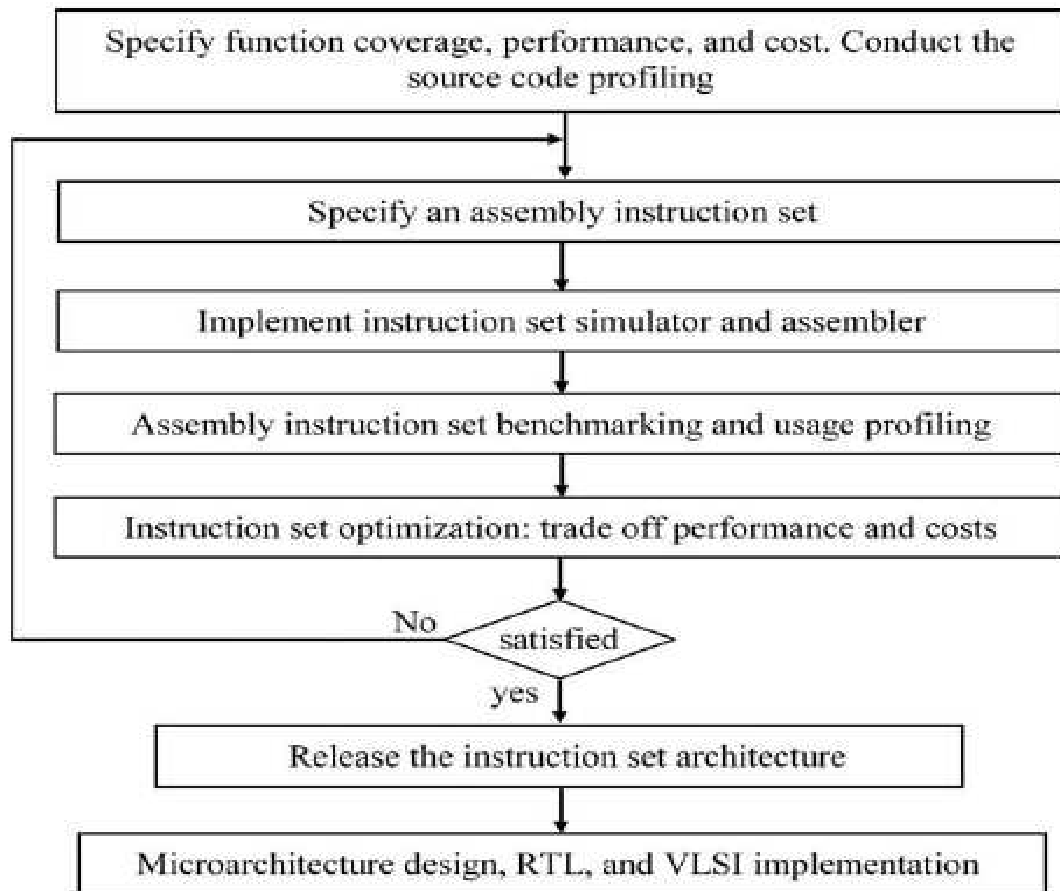

Figure1.3 ASIP Design Flow

.

CHAPTER 2

APPLICATION DESCRIPTION LANGUAGE

2.1. Introduction

The architecture design languages (ADLs) nowadays are offering promising avenues for fast design-space exploration with enough room for optimization for target-specific architectures. The advantages offered by the ADL-based design are as follows:

- Faster design space exploration;
- Seamless integration of the components through the automatic generation of the software tool-chain (simulator, high level language compiler, assembler etc.) as well as the RTL description of the processor;
- The higher abstraction level which helps in doing away with the details of the implementation and thereby, the designer can manage the increasingly complex processor design.

These merits coupled with the narrowing application-domains for the processors, encourage modelling of the application specific instruction-set processor (ASIP) using the ADLs. Examples of ADLs like LISA, EXPRESSION, MIMOLA, nML. The ADL used in this project is LISA.

2.2 LISA Modelling Fundamentals.

The scope of LISA is perfectly reflected by the meaning of its acronym, that is, "**L**anguage for **I**nstruction-**S**et **A**rchitectures". LISA is suited to model any architecture that is driven by an instruction set, in other words, any architecture whose behavior is steered by the content of a dedicated resource, which we call the instruction resource. The language elements are generic enough to cover any kind of target architectures like GP processors, RISC processors, DSPs, ASIPs, special purpose co-processors, and so on.

Resources and operations are the basic objects of a LISA processor model. Resources describe the storage elements of the processor. Operations are the basic language elements that describe the complete transition functions of the processor, including both the instructions and the instruction-independent functions such as the fetch mechanism.

2.3. Modelling Processor Resources.

Processor resources include the internal storage elements of the processor as well as dedicated input/output pins and global variables. The internal storage elements of the processor are represented by its registers and its internal memories.

While in cycle-accurate models there are other types of processor resources, like pipeline registers and interconnect signals, this manual confines itself to registers, memories, and pins.

Processor resources are declared in the resource section, which is indicated by the keyword *RESOURCE*, followed by the section body limited by braces as shown below:

```
RESOURCE {
... // section body, containing resource declarations
}
```

A resource declaration typically consists of an identifier, a data type specifier, and an optional keyword defining the semantic type of the resource. Few examples are register, ram, pin, and so on. All resources that are declared in a resource section are global to the entire LISA model. A LISA model may contain multiple resource sections; however, this is just an aesthetic aspect. Due to the global validity of all resources, merging or splitting resource sections have no effect on the function of the model. Consequently, the resource identifiers must be unique in the whole LISA model.

2.4. Modelling Instructions.

An *operation* is the basic building block that describes the state transitions of the processor, namely the instructions. The instruction is represented as an OPERATION with three sections that are labelled as CODING, SYNTAX, and BEHAVIOR.

Operations are declared by the keyword OPERATION, followed by a unique identifier and a body limited by braces as shown below:

```
OPERATION <operation_identifier> {
... // operation body, containing one or more sections.
}
```

Like a resource section, an operation has no super ordinate context and is globally valid in the LISA model. Therefore the operation identifier must be unique in the whole model. The definition of an instruction comprises the three properties: behavior, syntax, and coding. Each of these properties is modelled in a dedicated section within the body of an operation.

2.4.1. The Instruction Behavior.

The instruction behavior is modelled in the behavior section of an operation. This section consists of the keyword BEHAVIOR, followed by a body limited by braces. This body contains, in principle, arbitrary C block code to describe the instruction behavior as shown below:

```
OPERATION ... {
...
BEHAVIOR {
... // arbitrary C code
}
...
}
```

2.4.2 The Instruction Syntax.

The assembly syntax of an instruction is modelled in the syntax section of an operation. The syntax section consists of the keyword SYNTAX, followed by a body limited by braces.

```
OPERATION ... {
```

```
...
SYNTAX {
// sequence of syntax string elements
}
...
}
```

In the simplest case, the syntax body contains a string or a sequence of strings that represent the assembly syntax. Literal strings are enclosed in double quotes.

2.4.3 The Instruction Coding.

The binary image or coding of an instruction is modelled in the coding section of an operation. The coding section consists of the keyword CODING, followed by a body limited by braces as shown below:

```
OPERATION ... {
...
CODING {
... // sequence of bit fields
}
...
}
```

The coding body consists of a sequence of bit fields. A terminal bit pattern consists of the prefix "0b", followed by an arbitrary number of 0, 1, or X (don't-care) digits. Repetitions of the same digit can be alternatively represented by 0b, the digit, and the repetition count of the digit enclosed in brackets. For example, the term 0b1[5] is equivalent to 0b11111. Generally, the coding section may also contain non terminal bit fields represented by identifiers.

The most important property of the coding section is that it fully implies the instruction decoder. All tools of the Processor Designer, which require the generation of an instruction decoder, extract the necessary information from the coding sections of all operations in the LISA model.

CHAPTER 3

PROCESSOR DESIGN PLATFORM

3.1. Traditional Embedded System Design Flow

The traditional embedded system design flow is given in Figure 3.1. To simplify a design, system design inputs are partitioned and assigned to the HW design team and the SW design team at the beginning of the system function design. The SW design team and HW design team design their SW and HW independently without much interwork. In the SW design team, functions mapped to SW will be implemented in programs using high-level behavior language. The implemented functions will be verified during simulations, and finally the simulated programs are translated to machine language during the SW implementation.

From the HW design perspective, functions assigned to the HW team are allocated to HW modules. Modules could be either processors or functional circuits. The behaviors of programmable HW modules are described by an assembly language simulator. The behaviors of nonprogrammable HW modules are described by hardware description languages.

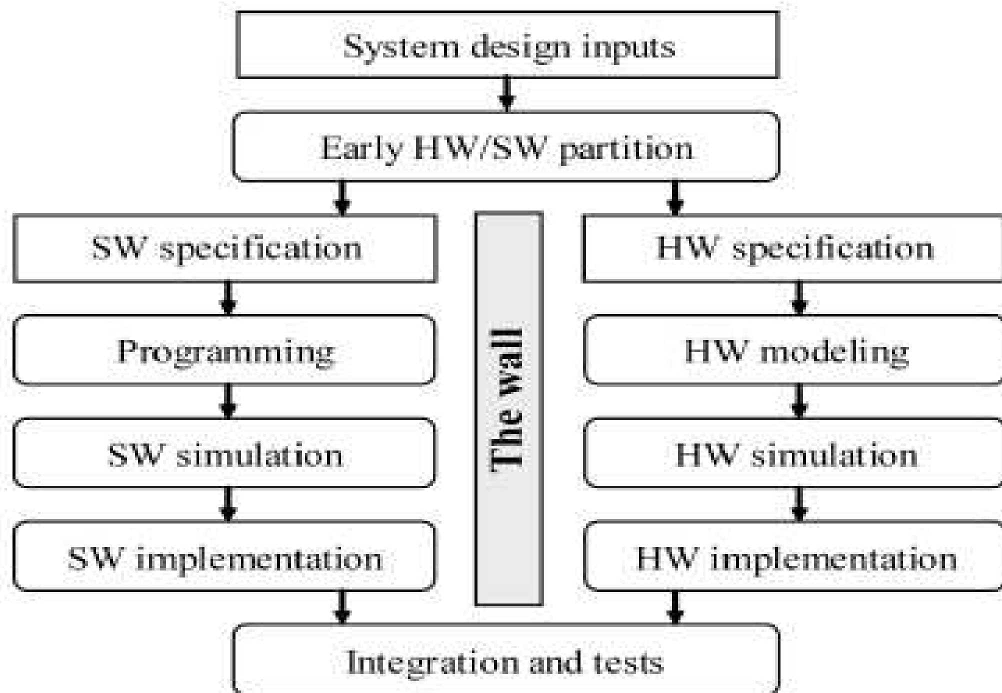

Figure 3.1 Traditional Embedded System Design Flow

Finally, the implemented SW and HW will be integrated. The implemented binary code of the assembly programs will be executed on the implemented HW. The design flow in Figure 3.1 is correct because it follows the golden rule of design: divide-and-conquer. However, when designing a high-quality embedded system, we actually do not know how to make a correct or optimized early partition. In other words, the early partition usually is not good enough without iterative optimizations through the design.

3.2. Hardware Software Co-Design Flow

The HW/SW co-design flow is depicted in Figure 3.2. Following the figure, the idea of the new design flow is to optimize the partition of HW and SW functions cooperatively at each design step during the embedded system design. HW/SW co-design trades off function partition and implementation using either SW or HW during all design steps through the embedded system design. Eventually, the results will be optimized following certain goals.

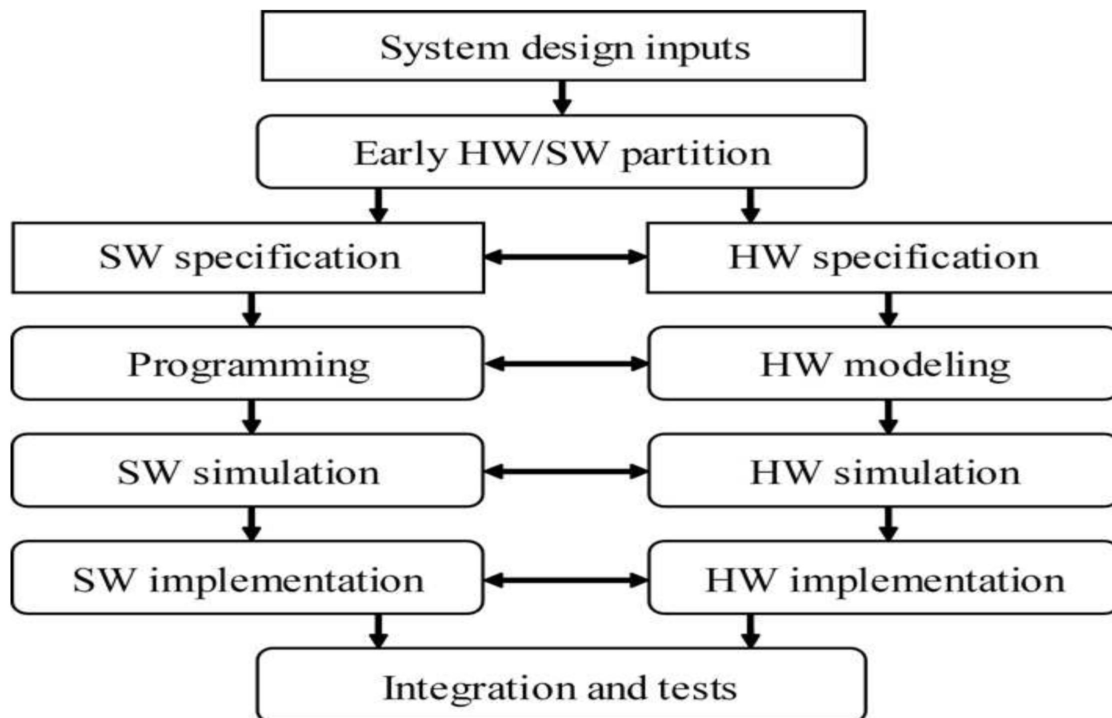

Figure 3.2 Hardware Software Co-Design Flow

3.3. CoWare Design Flow

Figure 3.2 shows the flow for Coware Processor Designer Platform. The design flow concentrates on Hardware Software Co-Design. As Shown in the figure, a LISA 2.0 description of the processor is written. The Coware Processor Designer then generates software development tools. Any particular application can then be fed to these software development tools. The executable file is then analyzed using the Processor Debugger. Once the design goals are met, the synthesizable RTL can be generated. The advantage of this flow is that if the design goals are not met, we just have to change the LISA description of the processor. The processor generator does the appropriate changes in the software development tools and the RTL

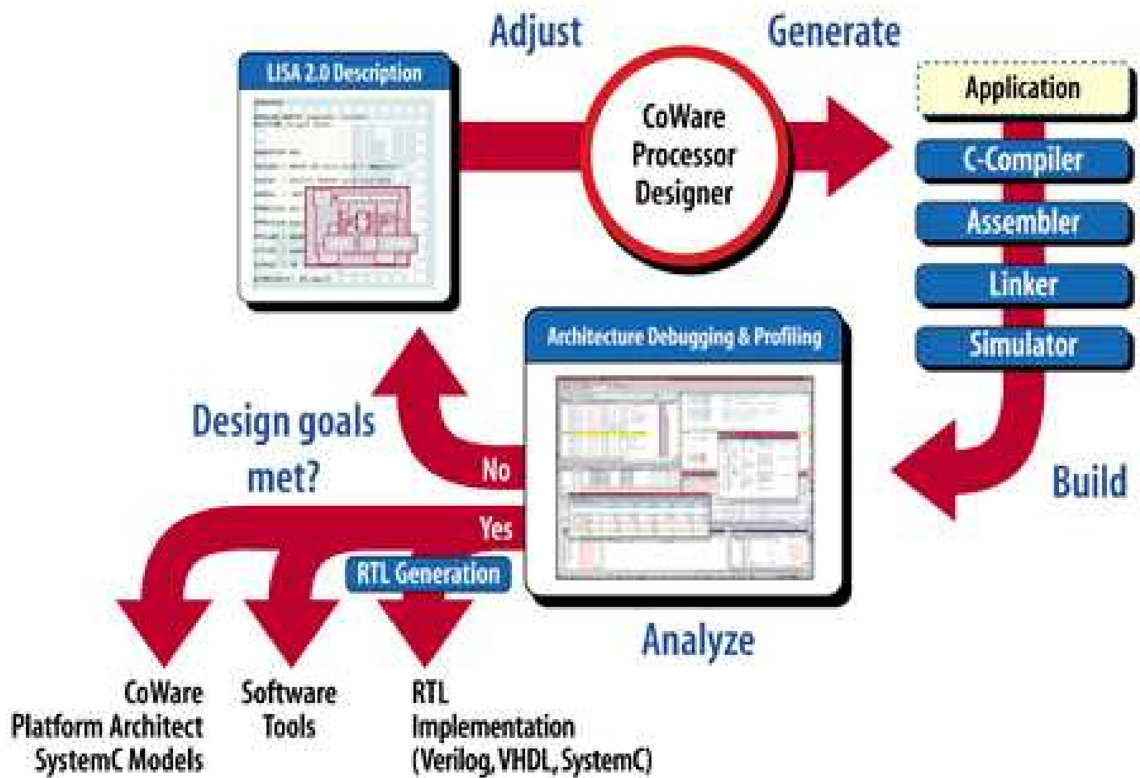

Figure 3.3 CoWare Design Flow

3.4. CoWare Processor Designer

CoWare Processor Designer is an automated, application-specific embedded processor design and optimization environment. The Processor Designer is the top-level model managing tool of the Processor Designer product family and is intended to facilitate designing LISA 2.0 models of processor architectures in the LISA 2.0 language. Figure 3.1 shows the Processor Designer Main Window:

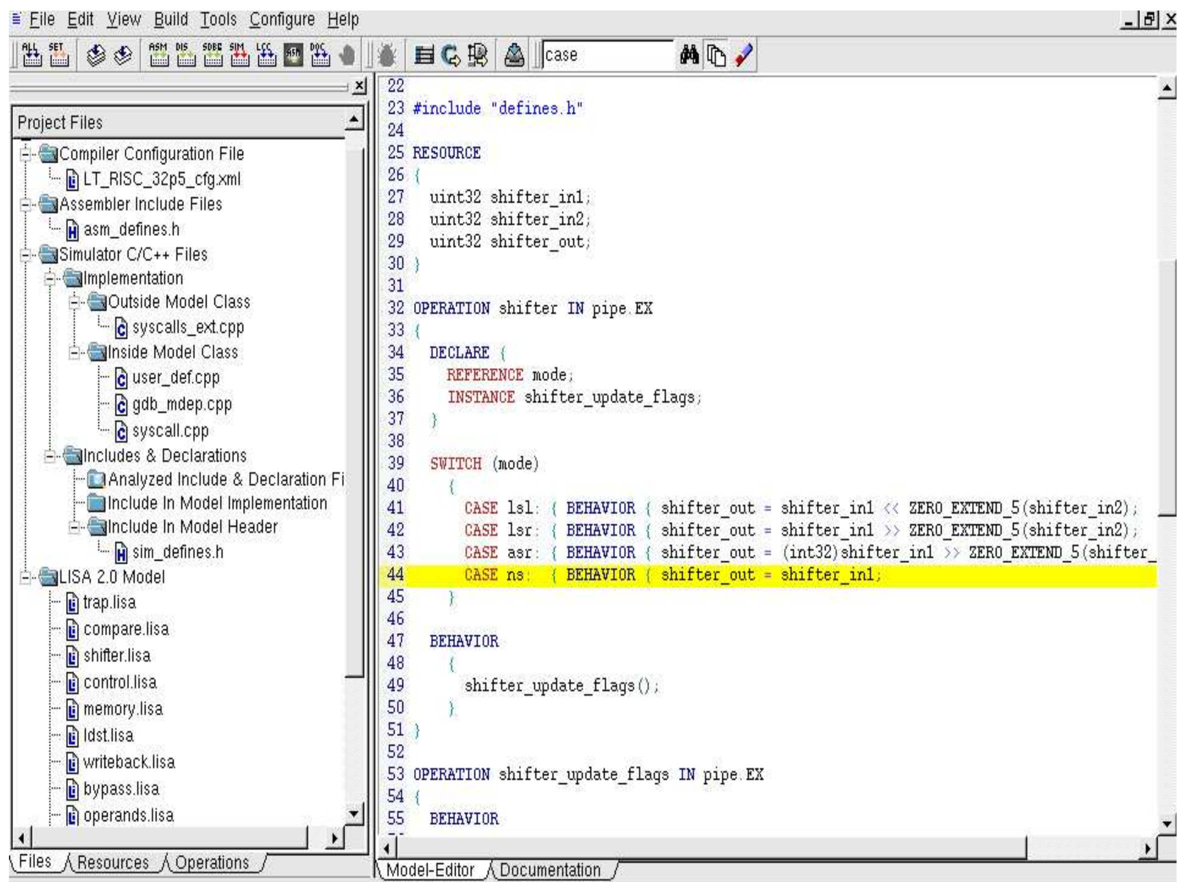

Figure 3.4 CoWare Processor Designer Main Window

CoWare Processor Designer has following advanced and flexible features:

- Automatic generation of synthesizable RTL with both control and datapath.
- Accurate profiling capabilities for high speed instruction set simulator.

- Compatible with extensively used synthesis tools like Synopsys Design Compiler and Cadence Encounter.
- Software development tool generation like assembler, linker, debugger, C-compiler.
- Ensures compatibility of instruction set simulator (ISS), software development tools and RTL implementation.
- Integrated profiling helps to optimize instructions for the target architecture.
- Enables the design team to develop flexible and reusable ASIPs rapidly.

The key to Processor Designer's automation is its Language for Instruction Set Architectures, LISA 2.0. In contrast to SystemC, which has been developed for efficient specification of systems, LISA 2.0 is a processor description language that incorporates all necessary processor-specific components such as register files, pipelines, pins, memory and caches, and instructions. It enables the efficient creation of a single "golden" processor specification as the source for the automatic generation of the instruction set simulator (ISS) and the complete suite of software development tools, like Assembler, Linker, Archiver and C-Compiler, and synthesizable RTL code.

The development tools, together with the extensive profiling capabilities of the debugger, enable rapid analysis and exploration of the application-specific processor's instruction set architecture to determine the optimal instruction set for the target application domain. Processor Designer enables the designer to optimize instruction set design, processor micro-architecture and memory sub-systems, including caches.

Processor Designer's use of a single high-level processor specification ensures the consistency of the ISS, software development tools and RTL implementation, eliminating the verification and debug effort necessitated by multiple, independently-created models.

3.5. The Instruction Set Designer.

The Instruction-Set Designer is a GUI for viewing, editing, and creating LISA processor models. Having a graphical representation of a processor model rather than just the

source code makes it much easier to get an overview and understand its hierarchy. Instruction sets can be designed and maintained in an intuitive way without having to cope with all the details of the syntax of the LISA language. Figure 3.5 shows the Instruction Set Designer Window.

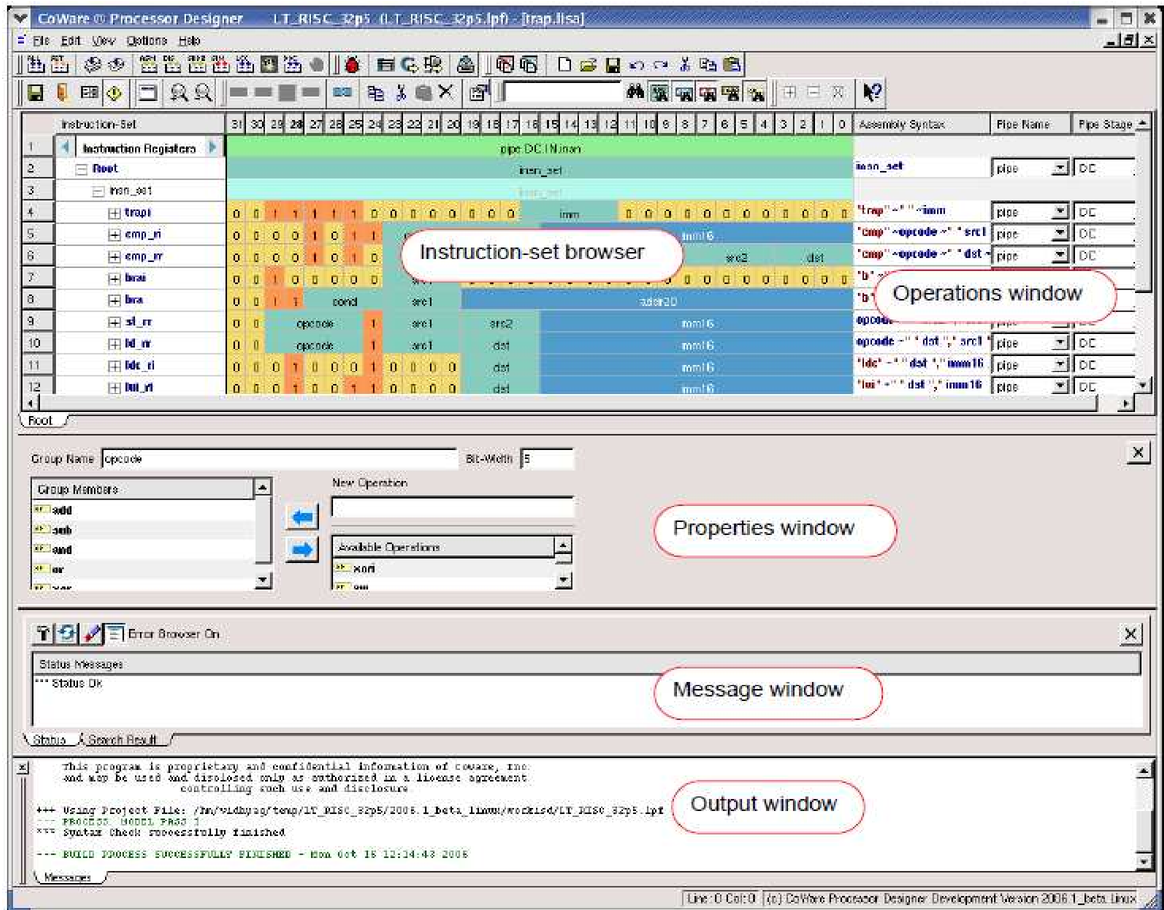

Figure 3.5 Instruction Set Designer Window

The Instruction-Set Designer does not replace the text editor; rather complements it. You can arbitrarily switch between the graphical and the textual representation. Changes made to the model in the GUI only result in minimal changes to the LISA code. All comments and formatted code are preserved. While the LISA hierarchy and the encoding of the instruction set is most efficiently designed with the GUI, the processor's resources and the hardware behavior is still manually written as LISA code.

3.6. CoWare Processor Debugger

The Processor Debugger GUI allows you to observe, debug, and profile the executed application source code and the state of the processor by visualizing all processor resources and the output which is produced by the executed application. Figure 3.6 shows the Processor Debugger Window for a loaded application.

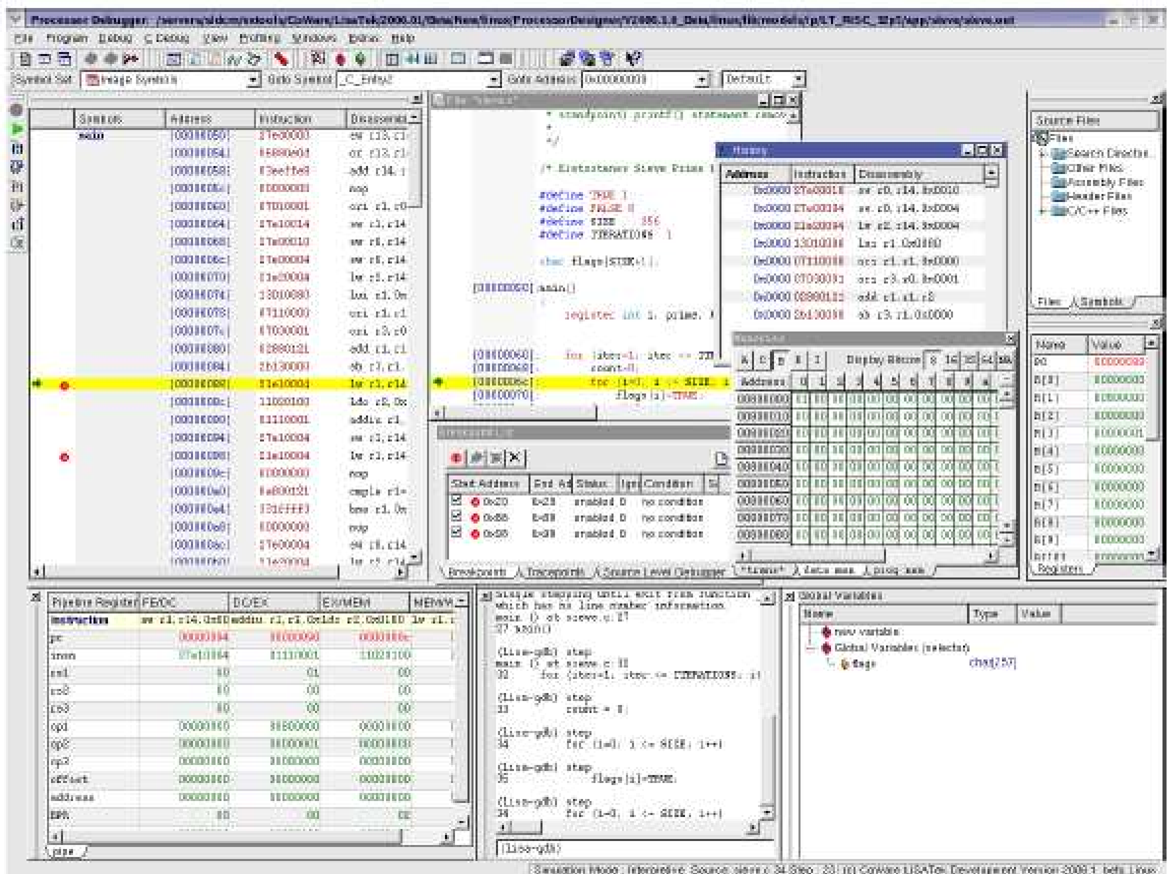

Figure 3.6 Processor Debugger Window

Furthermore, this GUI is intended to analyze and debug the LISA 2.0 processor model with special regard to the hardware behavior, instruction set, micro-architecture, and memory subsystem. The underlying ISS is derived from the LISA 2.0 model of the processor architecture.

CHAPTER 4

MAGMA LAYOUT TOOL

4.1. Magma Design Flow

Figure 4.1 shows the general Magma Design Flow. The uses of different tools are given below:

- **Blast Create:** RTL synthesis, DFT Timing Analysis
- **Blast Plan:** Floorplanning
- **Blast Fusion APX:** Physical Synthesis and Implementation

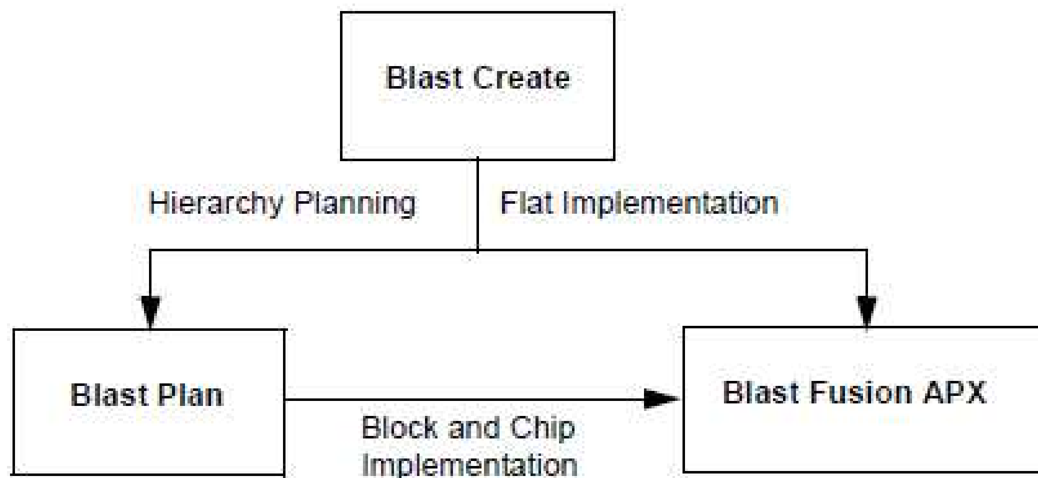

Figure 4.1 Magma Design Flow

4.2. Introduction to Magma Blast Create Tool

Blast Create is a gain-based RTL synthesis tool that provides fast, high-capacity synthesis, integrated into an RTL-to-GDSII design flow. Blast Create performs logic synthesis, data-path synthesis, physical synthesis, power optimization, scan-based DFT, and static-timing analysis.

Blast Create provides fast and early predictability of results before handing off to a back-end tool. Blast Create streamlines chip planning and design by eliminating the numerous, cumbersome, and error-prone data transfers between point tools in traditional flows. Blast

Create outputs a design that is a placed, timing-correct physical design, with DFT structures inserted and that is ready for routing. Figure 4.2 shows the flow and commands for the Blast Create tool.

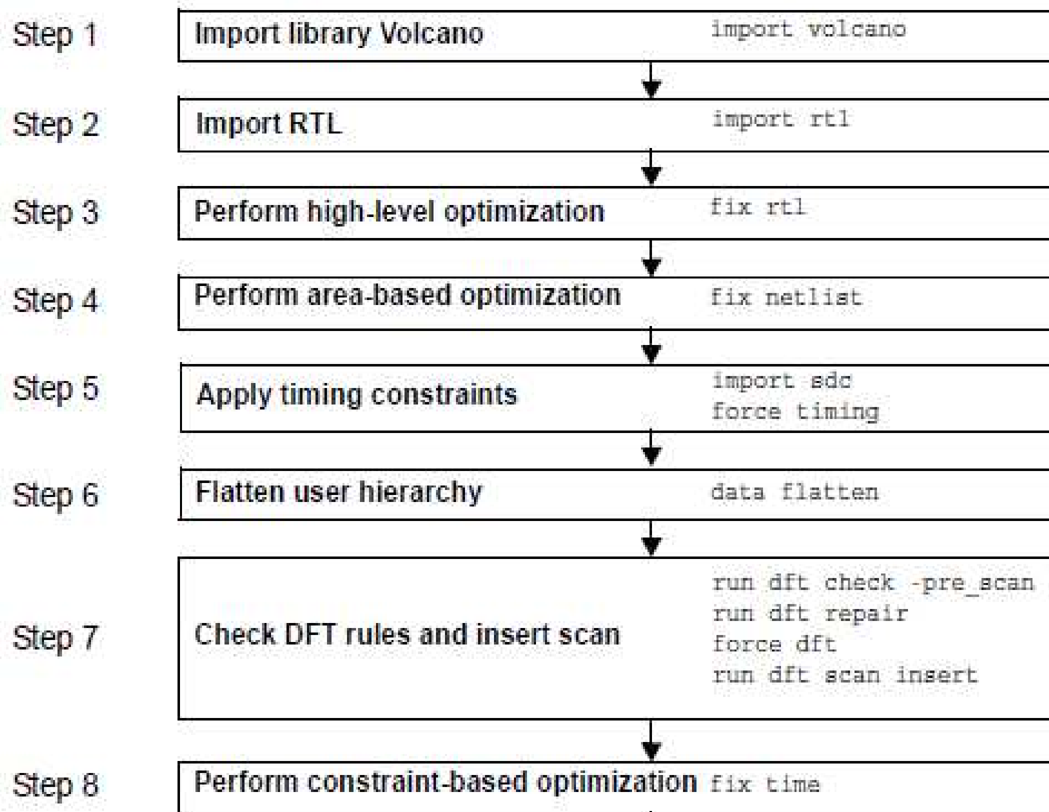

Figure 4.2 Magma Blast Create Flow and Commands

4.3. Introduction to Magma Blast Fusion Tool

Floorplanning, analyzing and refining the floorplan, power routing, physical implementation and synthesis are possible in the Blast Fusion Environment. Floorplanning is the process of:

• Positioning blocks on the die or within another block, thereby defining routing areas between them.

• Creating and developing a physical model of the design in the form of an initial optimized layout.

Figure 4.3 shows the floorplanning within the Blast Fusion flow.

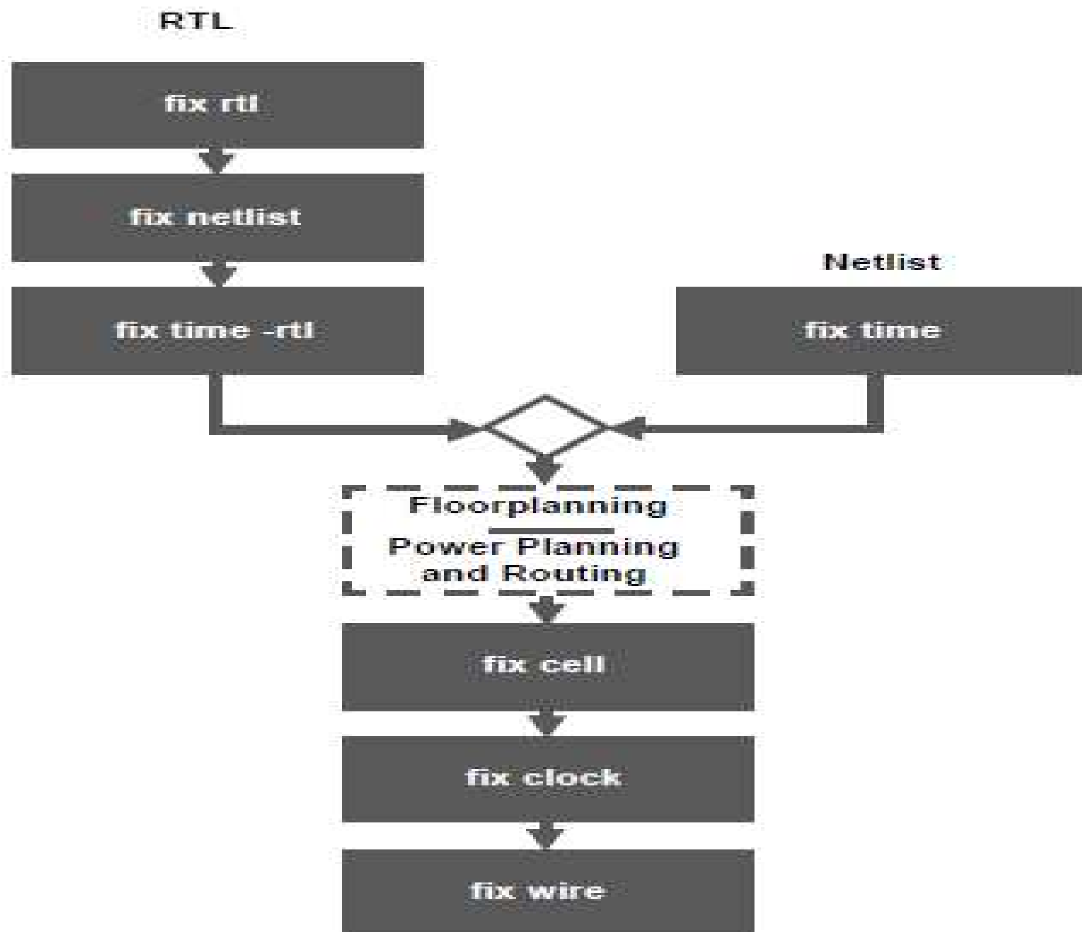

Figure 4.3 Floorplanning within Magma Blast Fusion Flow

CHAPTER 5

Virtex-II Pro FPGA

5.1 Introduction.

FP**GA** stands for **F**ield **P**rogrammable **G**ate **A**rray. These are programmable semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnect. There are many forms of devices which are field programmable. These are PAL, PLD, CPLD, and FPGA. These devices differ on their granularity, how the programming is accomplished etc. PAL, PLA and CPLD devices are usually smaller in capacity but more predictable in timing and they can be implemented with Sum-of-Products, Product-of-Sums or both. FPGA devices can be based on Flash, SRAM, EEPROM or Anti-Fuse connectivity. As opposed to Application Specific Integrated Circuits (ASICs) where the device is custom built for the particular design, FPGAs can be programmed to the desired application or functionality requirements. Although one-time programmable (OTP) FPGAs are available, the dominant types are SRAM based which can be reprogrammed as the design evolves. The FPGA block structure is shown in Figure 5.1.

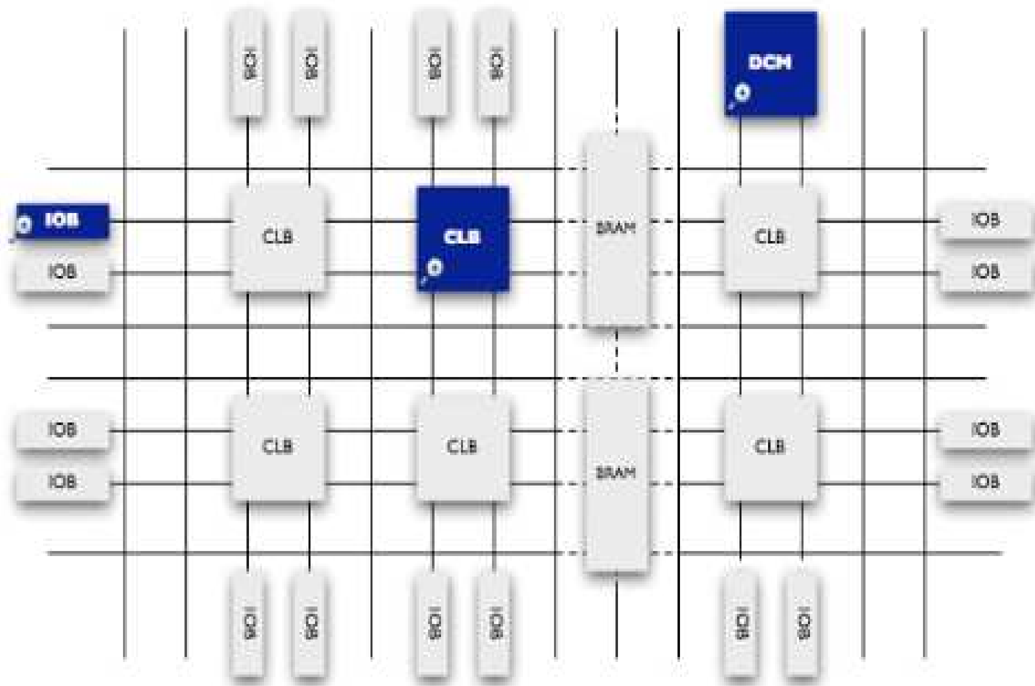

Figure 5.1 FPGA Block Structure.

The current design implementation uses Xilinx University Program Virtex-II Pro Development System (XUP VP2).

5.2 XUP Virtex-II Pro Development System.

The XUP Virtex-II Pro Development System provides an advanced hardware platform that consists of a high performance Virtex-II Pro Platform FPGA surrounded by a comprehensive collection of peripheral components that can be used to create a complex system and to demonstrate the capability of the Virtex-II Pro Platform FPGA. Figure 5.2 shows the XUP Virtex-II Pro system development photo.

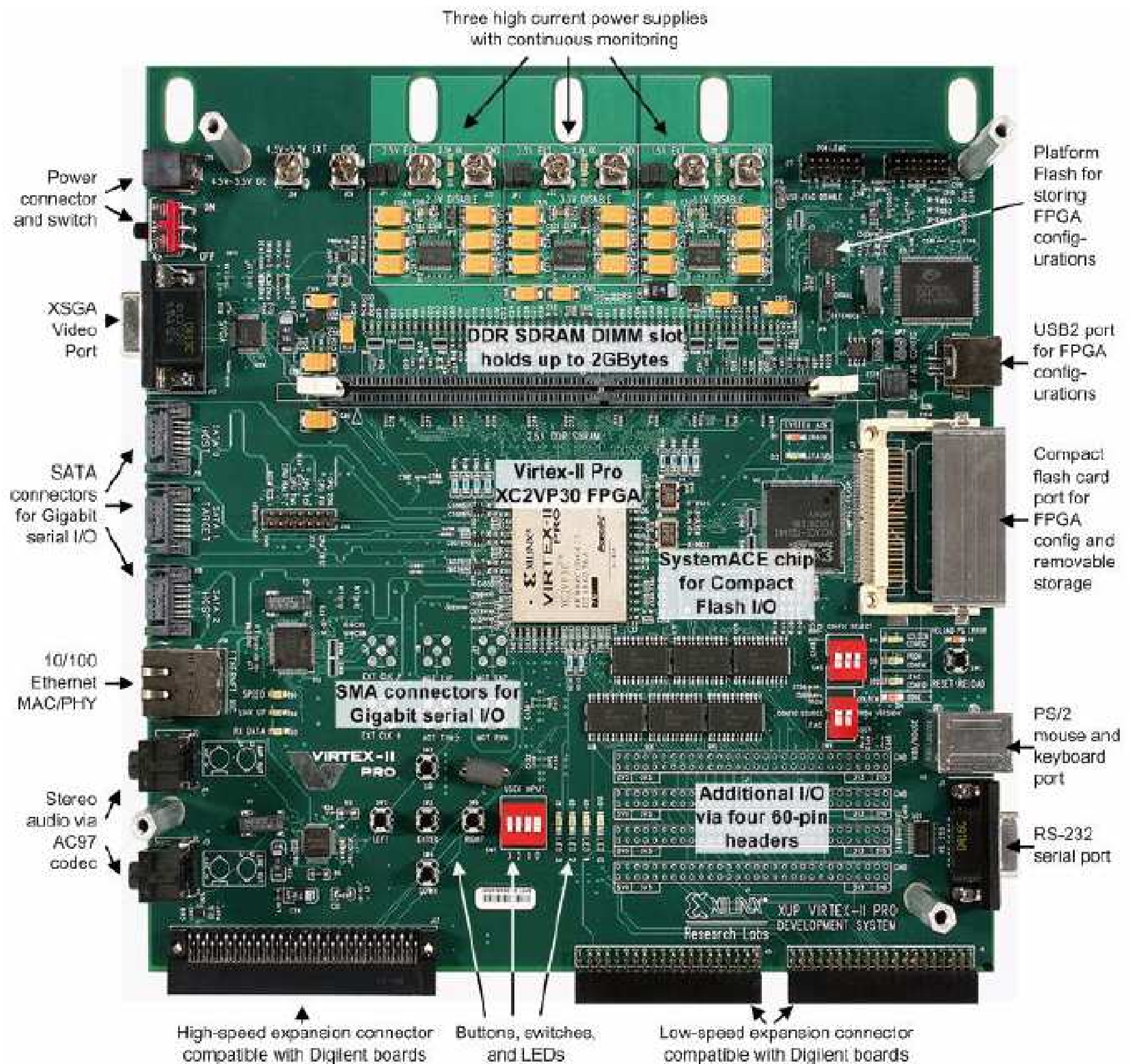

Figure 5.2 XUP Virtex-II Pro Development System Board Photo.

Figure 5.3 shows a block diagram of the XUP Virtex-II Pro Development System.

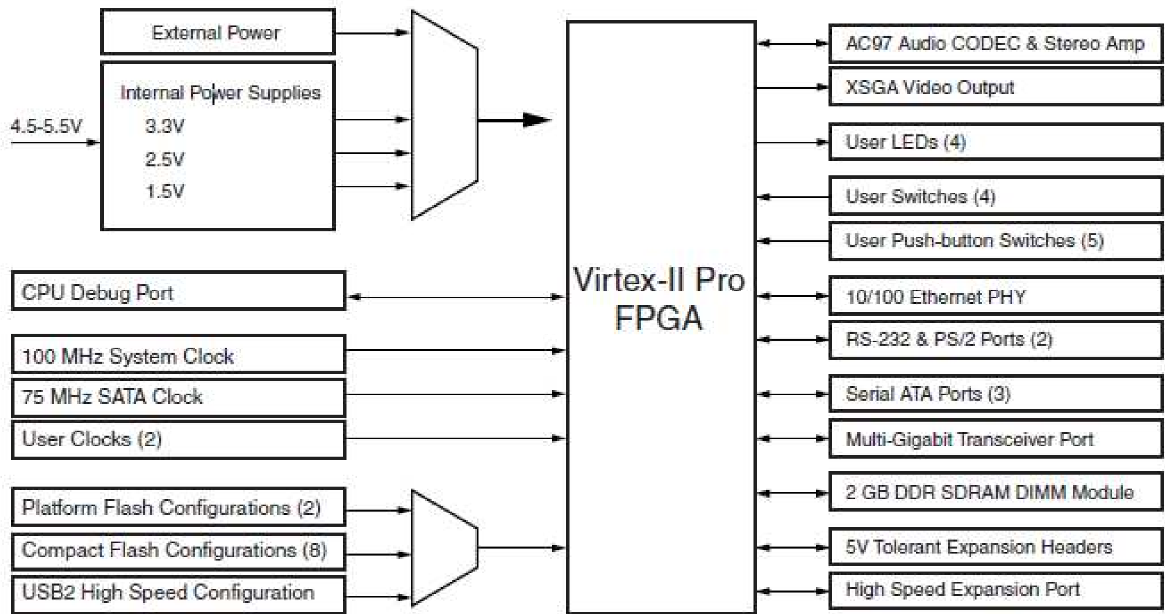

Figure 5.3 XUP Virtex-II Pro Development System Block Diagram.

Table 5.1 lists the Virtex-II Pro device features.

Features	XC2VP30
Slices	13969
Array Size	80 x 46
Distributed RAM	428 Kb
Multiplier Blocks	136
Block RAMs	2448 Kb
DCMs	8

Table 5.1 XC2VP30 Device Features.

5.3. Chipscope Pro Tools.

As the density of FPGA devices increases, so does the impracticality of attaching test equipment probes to these devices under test. The ChipScope Pro tools integrate key logic analyzer and other test and measurement hardware components with the target design. The tools communicate with these components and provide the designer with a robust logic analyzer solution.

The tools design flow (Figure 5.4) merges easily with any standard FPGA design flow that uses a standard HDL synthesis tool and the ISE 10.1 implementation tools.

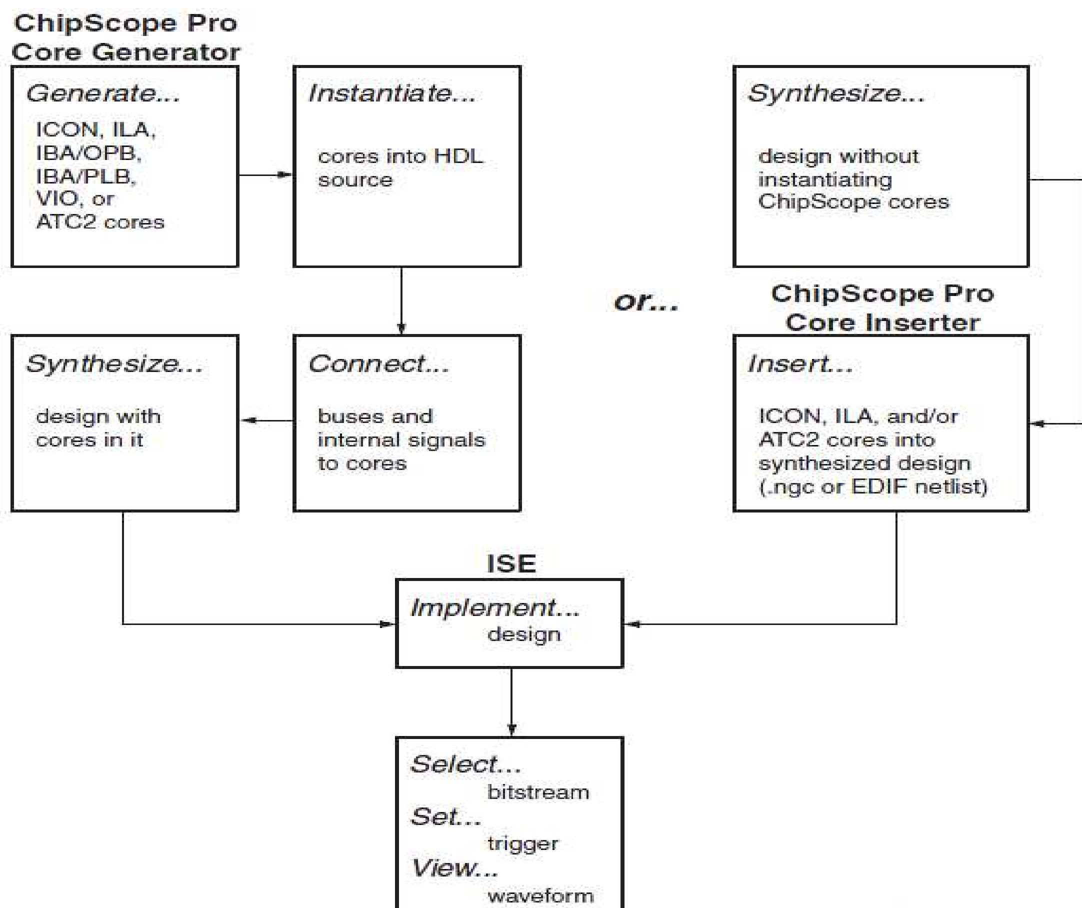

Figure 5.4 Chipscope Pro Tools Design Flow.

Chipscope Analyzer provides device configuration, trigger setup, and trace display for the ILA and other cores. The various cores provide the trigger, control, and trace capture capability.

CHAPTER 6

IMPLEMENTATION OF PROCESSOR

6.1 Implementation of General Purpose Processor

The CoWare Processor Designer allows two modes of implementation. They are discussed below:

ISS Design	Processor Design
• Instruction Accurate Modeling is used. • The objective is to simulate instruction set at very high speed. • It has no notion of a pipeline.	• Cycle Accurate Modeling is used. • The objective is to design a processor. • Pipeline is the integral part of this design.

Table 6.1 Modes of Implementation in CoWare Design

A General Purpose Processor is first implemented in CoWare Processor Designer Platform. The instruction set of this processor is selected so as to cover the most recurring instructions. The instructions implemented here are given in Table 6.2:

Sl. No.	Mnemonic	Syntax	Operation
1.	nop	nop	It performs no operation
2.	incr	incr rx	It increments the contents of a general purpose register rx
3.	decr	decr rx	It decrements the contents of a general purpose register rx

4.	add	add rd,rs1,rs2	It adds the contents of GPR rs1 and rs2 and save it to GPR rd
5.	sub	sub rd,rs1,rs2	It subtracts the contents of GPR rs2 from rs1 and save it to GPR rd
6.	and	and rd,rs1,rs2	It performs logical AND operation on the contents of GPR rs1 and rs2 and save it to GPR rd
7.	or	or rd,rs1,rs2	It performs logical OR operation on the contents of GPR rs1 and rs2 and save it to GPR rd
8.	xor	xor rd,rs1,rs2	It performs logical XOR operation on the contents of GPR rs1 and rs2 and save it to GPR rd
9.	not	not rd,rs	It saves the 1's complement of the contents of the GPR rs in GPR rd
10.	mul	mul rs1,rs2	It multiplies the contents of GPR rs1 and rs2 and save it to GPR r0
11.	mac	mac rs1,rs2	It multiplies and accumulates the contents of GPR rs1 and rs2 and save

			it to GPR r0
12.	load	load rd,value	It saves an immediate value in a GPR rd.
13.	ldm	ldm addr,value	It saves an immediate value in a location of data memory
14.	jmp	jmp addr	It causes program counter to jump unconditionally
15.	jne	jne rd,rs,addr	It compares the contents of GPR rd and rs and ca causes program counter to jump if they are not equal
16.	mvm	mvm [rd+addr],rs	It moves the contents of GPR rs to a location in data memory
17.	mov	mov rd,[rs]	It moves the contents of a location in data memory to a GPR rd
18.	shl	shl rd,rs,count	It performs left shift operation on the contents of GPR rs by the count specified and saves it to GPR rd

19	shr	shr rd,rs,count	It performs right shift operation on the contents of GPR rs by the count specified and saves it to GPR rd

Table 6.2 Instructions of Implemented GPP

The LISA 2.0 description of the general purpose processor is written for the instructions given above. As stated in chapter 2, LISA code consists of processor resources and operations. Part of the LISA code of the processor is given next:

```
RESOURCE {
        MEMORY_MAP    {
        RANGE(0x0000, 0x0fff) -> prog_mem[(31..0)];
        RANGE(0x1000, 0x1fff) -> data_mem[(31..0)];
        }
        MEMORY uint32 prog_mem    {
        …
         };
        MEMORY uint32 data_mem    {
        …
         };
        REGISTER  int32 GPR[0..31];
        PROGRAM_COUNTER  uint32  FPC;
         REGISTER  uint32 IR;
        PIPELINE_REGISTER IN pipe{
        …
        }
}
```

```
OPERATION reset {
	BEHAVIOR {
	…
	}
}

OPERATION fetch IN pipe.FE  {
…
}

OPERATION decode IN pipe.DC {
…
}

OPERATION alu IN pipe.DC{
…
}

OPERATION add IN pipe.EX{
...
}
```

To increase the design efficiency and in order to exploit common properties of instructions, operation hierarchy is defined. Figure 6.1 shows the operation hierarchy of the processor implemented. Operation main activates operation fetch which is in the stage 'FE' of the pipeline. Operation fetch activates operation decode which is in the stage 'DC' of the pipeline. The operation decode activates all other operations in the stage 'EX' of the pipeline.

According to the LISA 2.0 description of the processor, the processor designer generates the software development tools. An assembly language code to find out the convolution of two sequences using FIR filter was then written.

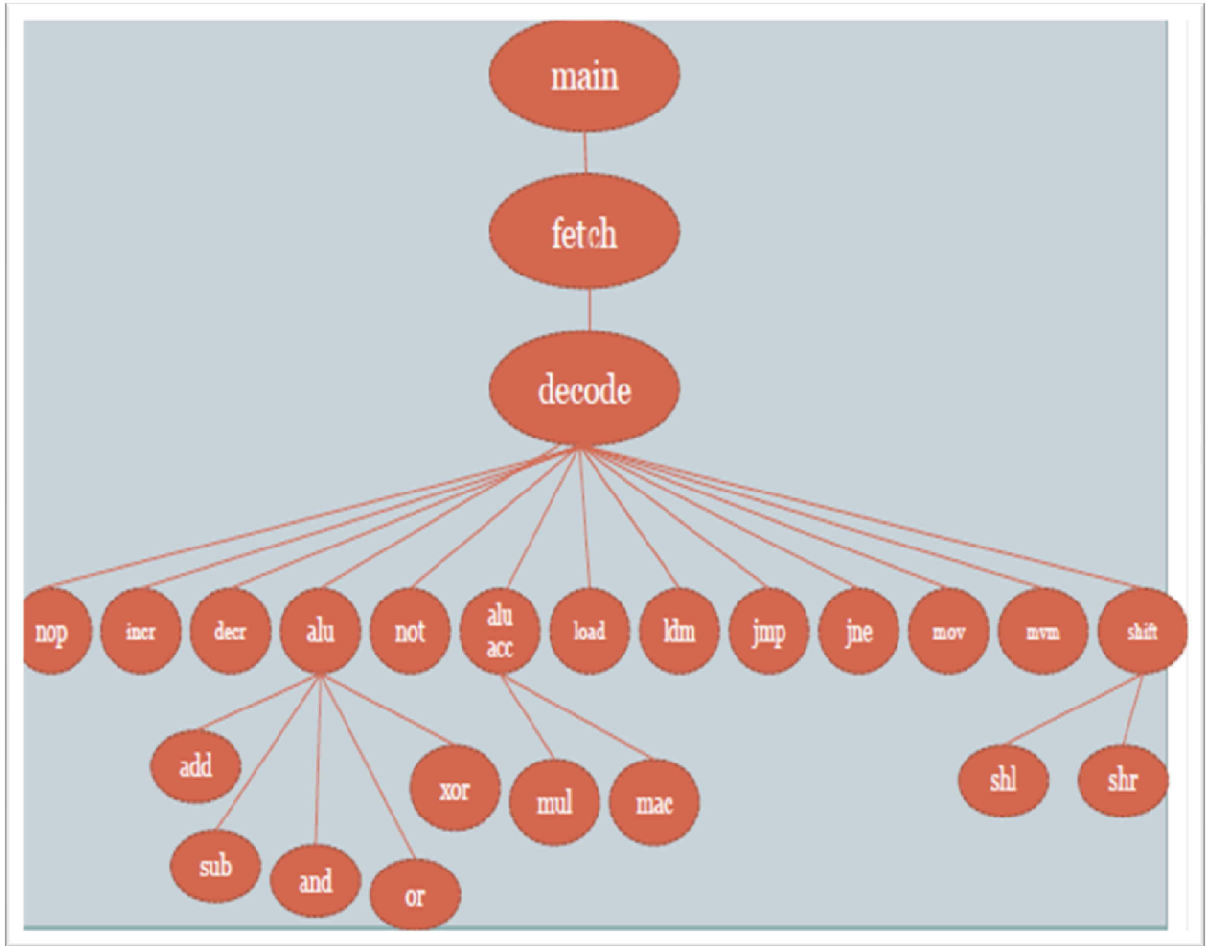

Figure 6.1 Operation Hierarchy of implemented GPP

The application can be assembled using the following command at command prompt in the specific folder:

$. /lasm fir.asm

This command generates fir.lof file. Now this file can be linked by typing the command:

$. /llnk fir.lof

This will create an executable file a.out. This executable file can then be analyzed in the processor debugger.

6.2 Architecture Profiling and Debugging

The LISA debugger frontend is a generic graphical user interface for the generated LISA simulator as shown in the figure 6.2. It visualizes the internal state of simulation process.

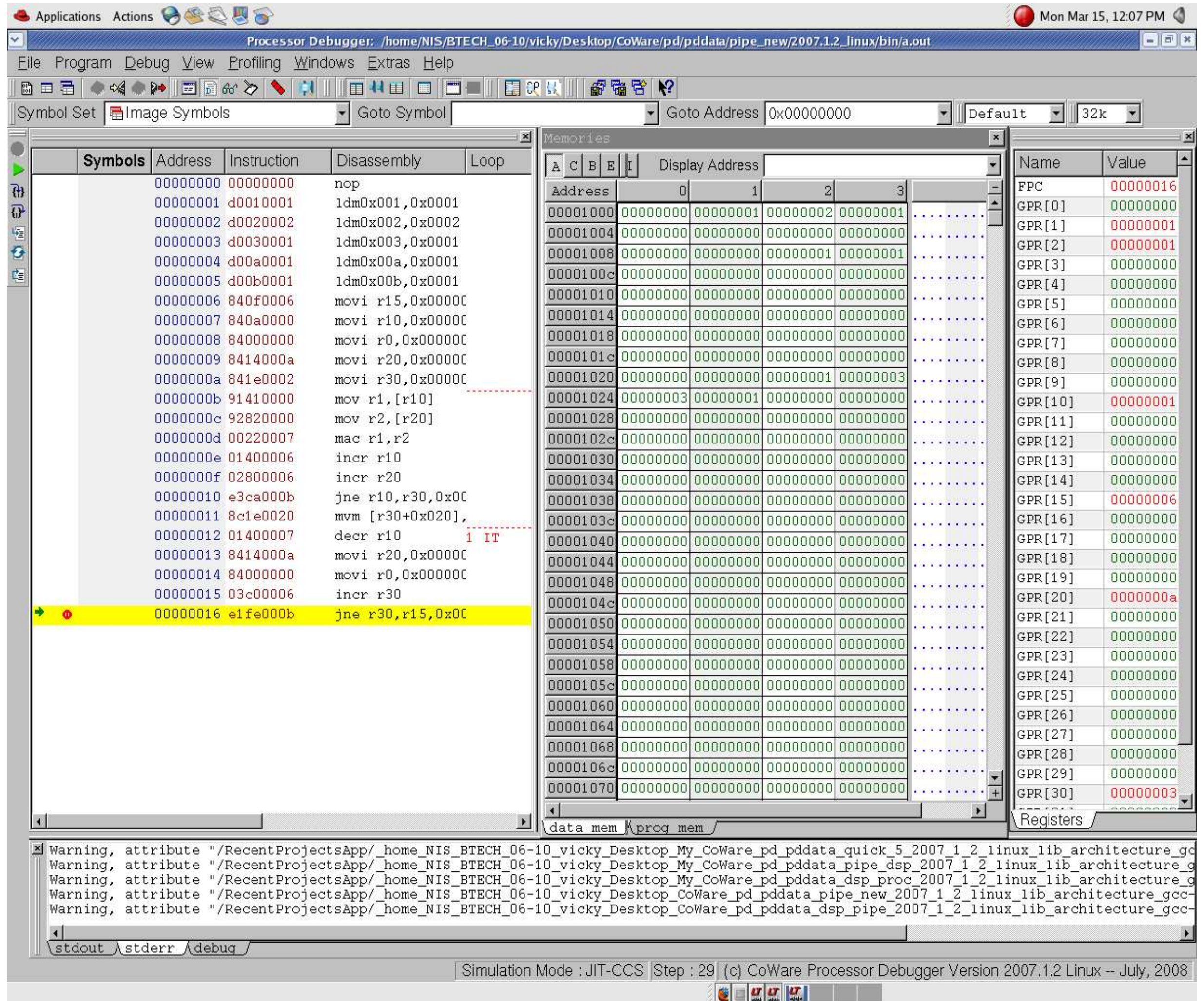

Figure 6.2 Processor Debugger for the Implemented GPP

The processor debugger provides extensive hardware and software profiling capabilities like register profiling, memory profiling, resource profiling and operation profiling. Memory profiling tells about the access statistics for the memories contained in the processor model. Similarly resource profiling shows the access statistics for all resources modeled with the resource specifier as one of register, program counter and control register in the LISA model. Operation profiling gives us the information about executions for all the operations divided among the pipeline stages.

Applications Actions

LISA Operation Profile

Name	Calls	Calls/Total	Calls/Max	Calls/Max	Stall Cause	Flush Cause
decode	30	25.21%	100.00%		0	0
nop	1	0.84%	3.33%		0	0
incr	5	4.20%	16.67%		0	0
decr	1	0.84%	3.33%		0	0
alu	0	0.00%	0.00%		0	0
mul	0	0.00%	0.00%		0	0
mac	2	1.68%	6.67%		0	0
alu1op	0	0.00%	0.00%		0	0
load	7	5.88%	23.33%		0	0
ldm	6	5.04%	20.00%		0	0
jmp	0	0.00%	0.00%		0	0
jne	2	1.68%	6.67%		0	0
mvm	2	1.68%	6.67%		0	0
mov	4	3.36%	13.33%		0	0

Figure 6.3 Operation Profiling for the desired application

The figure 6.3 shows the profiling results for an assembly code written to calculate convolution using FIR filter. This profiling information is very much required to optimize our design. Based on the profiling results, the processor was optimized with regards to resources, memory and operations. The new processor included only those operations required to calculate the convolution using FIR. Thus, the development tools, together with the extensive profiling capabilities of the debugger, enabled analysis and exploration of the application specific processor's instruction set architecture to determine the optimal instruction set for the target application domain i.e. convolution using FIR.

Figure 6.4 shows the Instruction Set Designer window of the designed ASIP.

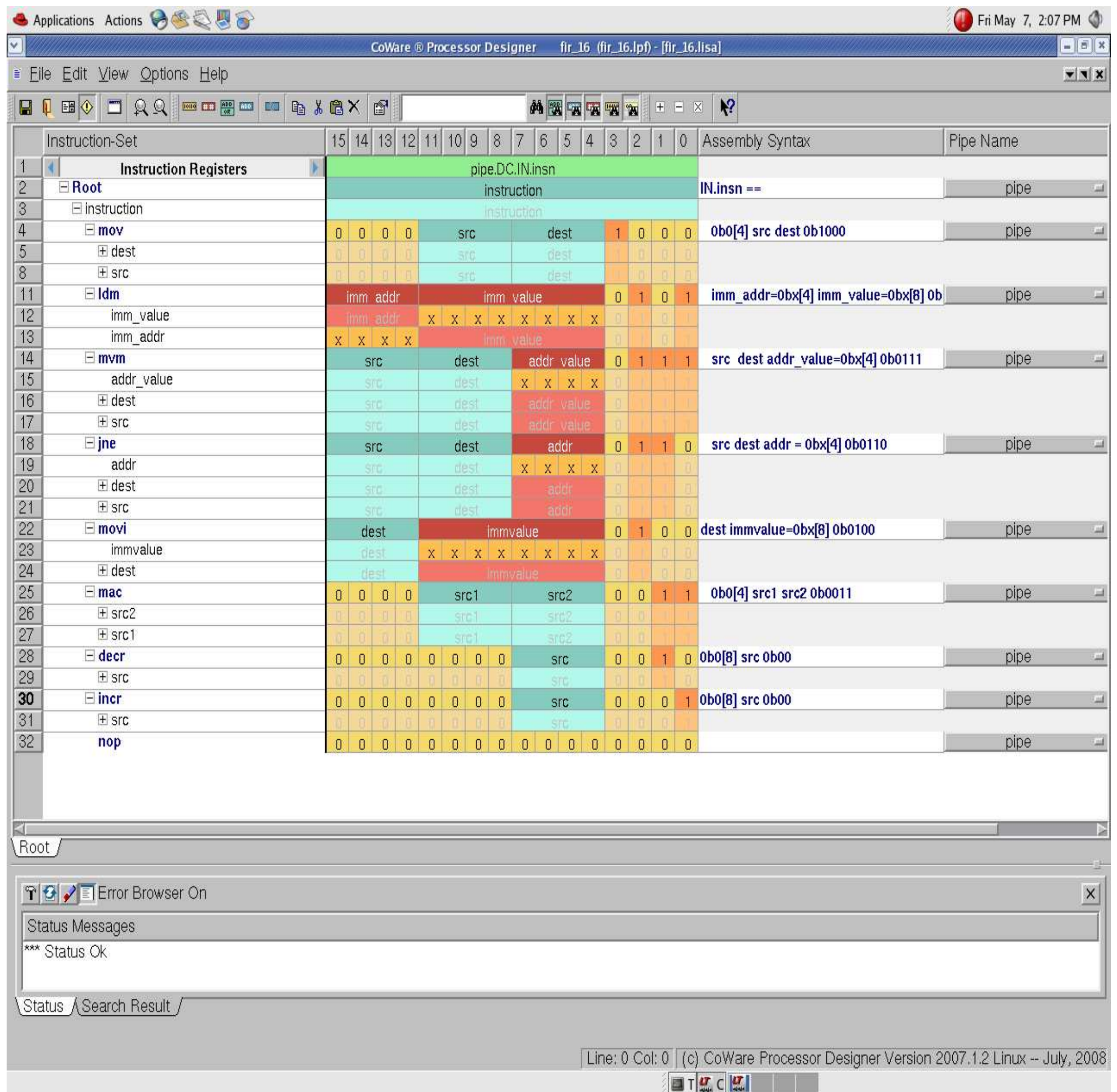

Figure 6.4. Instruction Set Designer Window of the Designed ASIP

CHAPTER 7

SIMULATION & SYNTHESIS RESULTS

7.1 Introduction

We developed a General Purpose Processor with possible 19 instructions. Later on we targeted an application named FIR filter, for which only 9 out of the 19 instructions are being used. Therefore in the process a novel ASIP is developed. Both these processors execute the same FIR filtering algorithm and a comparative assessment is brought out. In this way we have implemented an ASIP optimizing the previous processor with regards to:

- Data and program memory
- Instruction set
- Number of general purpose registers

The Processor Generator tool provided in the Processor Designer generated the synthesizable RTL for both the processors. The structure of the generated HDL is given in the figure 7.1

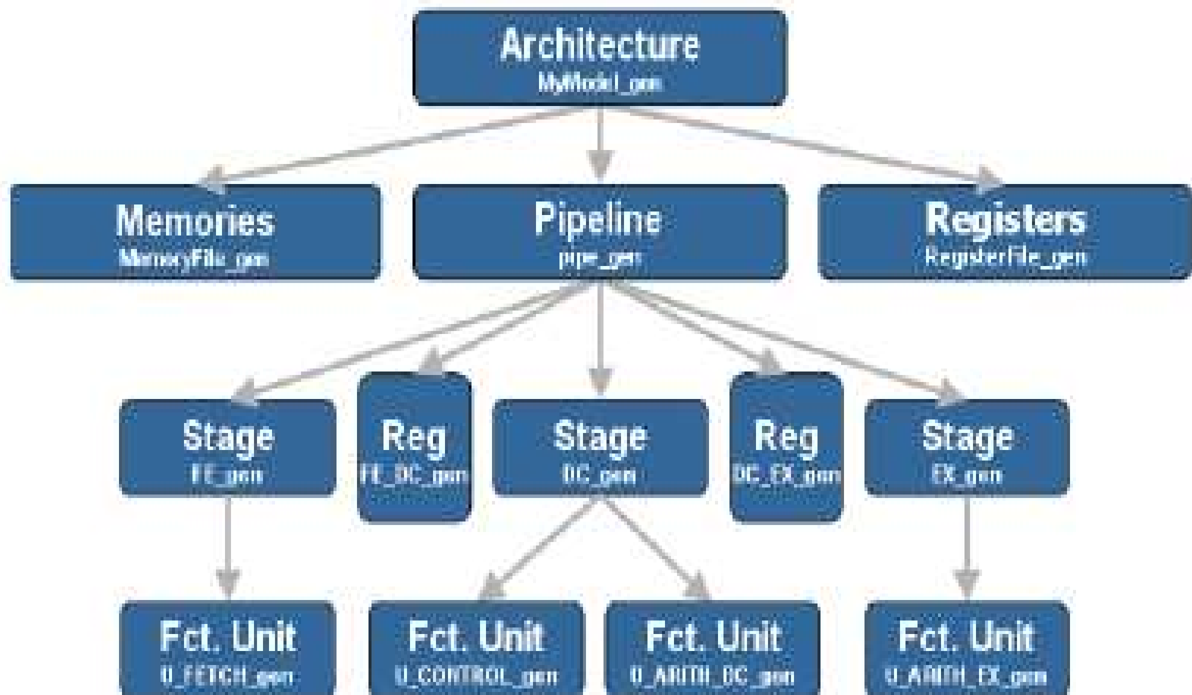

Figure 7.1 The Generated HDL Code Structure

The generated RTL was then simulated using ModelSim and synthesized using Cadence Encounter.

7.2. Simulation Results

Processor Generator can be configured to automatically generate scripts for running a RTL simulation and a gate-level synthesis. The generated RTL of the processor is simulated using the same executable file of our application. Figure 7.2 shows the simulation results (the contents of the data memory.)

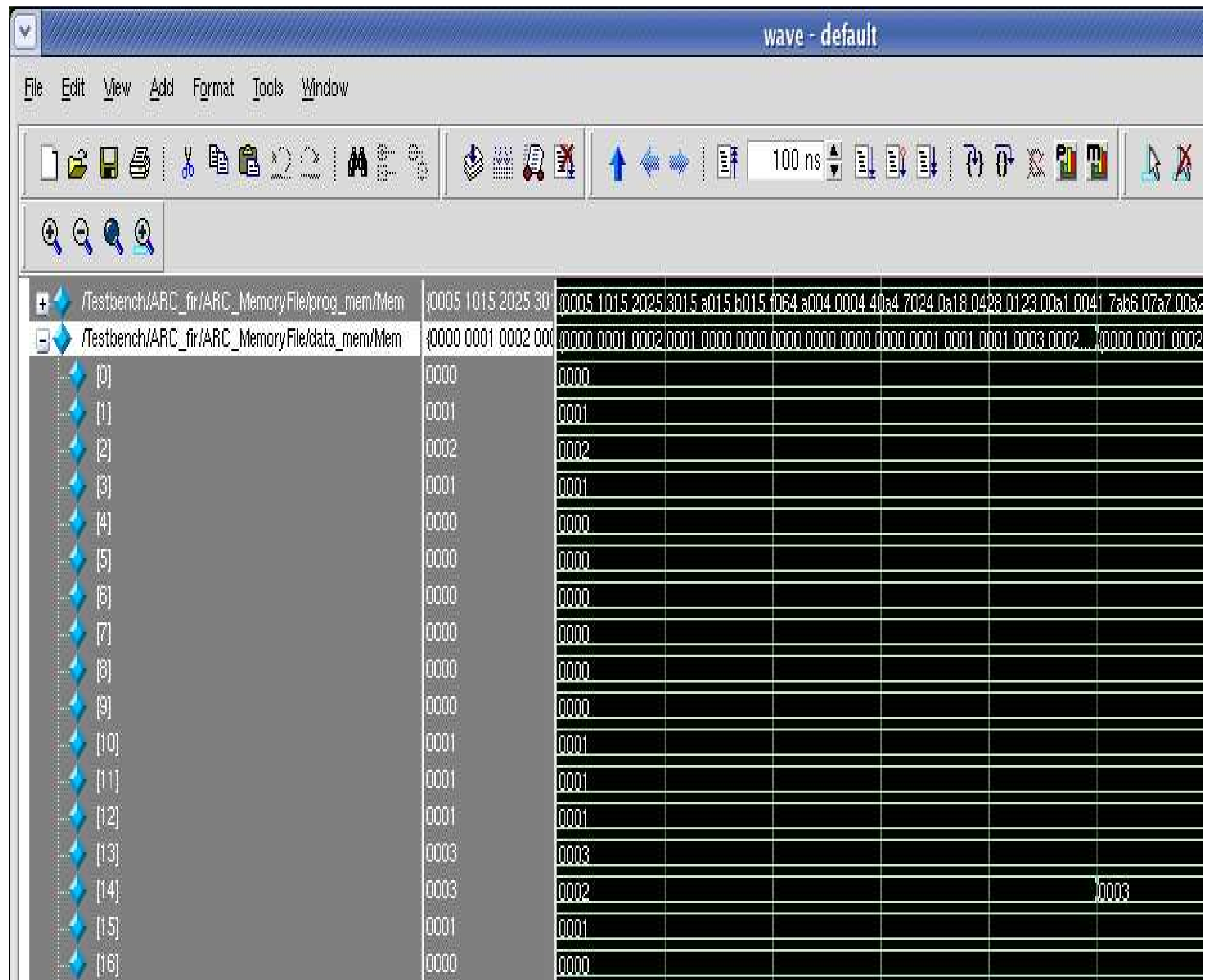

Figure 7.2 Simulation Results

In the above simulation, both the program and data memory are shown. The memory locations of the data memory show the result of the convolution of two sequences. The program memory locations contain the binary image of the instruction of the assembly

language code written for the application. The results shown here are same as shown in the Processor Debugger window; this verifies the CoWare design flow.

7.3. Synthesis Results

The RTL was synthesized using Cadence Encounter and the results are tabulated as follows:

Parameters / Processor	Number of Cells	Area (in μm^2)	Power (in nW)
GPP	24682	75489	3870366.052
ASIP	6856	23611	1847776.752

Table 7.1 Synthesis Results

The library used for the synthesis was TSMC (65nm). Thus we can see a drastic reduction in the area and power requirement.

CHAPTER 8

LAYOUT & FPGA IMPLEMENTATION OF THE PROCESSOR

8.1. Layout using Magma Tool

The final layout of the ASIP was drawn using Magma Blast Create and Blast Fusion Tool. Figure 8.1 shows the layout of the processor. The processor has in all 67 pins.

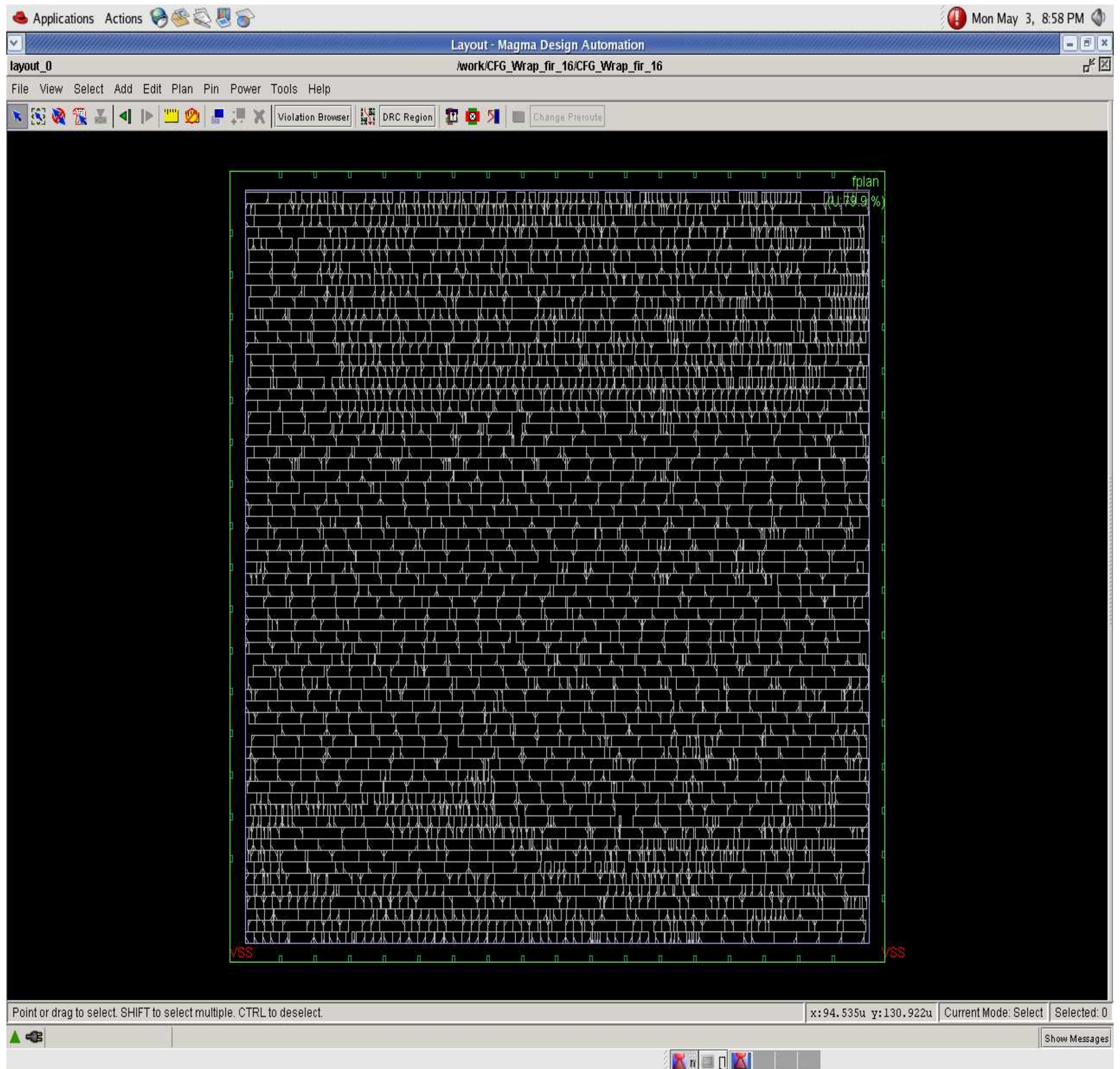

Figure 8.1 Layout of the Processor

8.2. Configuring the Virtex-II Pro FPGA.

The Processor Generator embedded in the Processor Designer Environment generates the RTL both in VHDL and VERILOG. For the FPGA implementation Verilog files are used to build the project in the Xilinx ISE 10.1.03.

The design consists of Top module and other modules that are instantiated in the top module. The data and program memory are generated using the Xilinx IP core Block Memory generator. The program memory is loaded with the binary image of the instruction at the memory locations using the *"coe"* file initialization. The FPGA design flow is shown in figure 9.1.

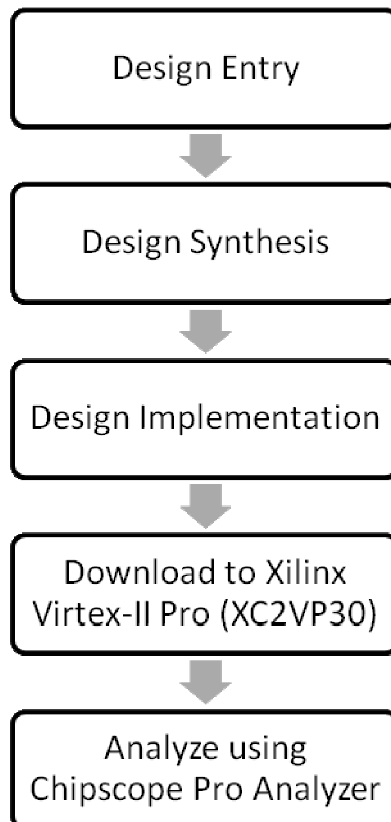

Figure 8.2.FPGA Design Flow

The device utilization under the current design is shown in table 9.1 below:

Device (XC2VP30) Utilization Summary

Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	676	27392	2%
Number of 4 input LUTs	2661	27392	9%
Logic Distribution			
Number of occupied Slices	1571	13696	11%
Total number of 4 input LUTs			
Number used as Logic	2534		
Number used as rout-thru	85		
Number used as shift registers	127		
Number of RAMB 16s	4	136	2%
Number of MULT18X18s	1	136	1%

Table 8.1 Target Device (XC2VP30) Utilization

After the configuration of the FPGA the analysis is done using the Chipscope Pro Analyzer. Figure 8.3.shows the waveform window in the analyzer.

Fig 8.3 Chipscope Analyzer Waveform for the Current Design.

In the analyzer waveform, the address and data bus of the program memory and data memory are shown respectively. The data corresponding to the address are shown in the data bus. The binary image of the instruction of the assembly language code is shown in the memory location. These data are same as the simulation and the Processor Debugger result. This verifies the functionality of the processor in the FPGA.

CHAPTER 9

CONCLUSION

In this project, a processor model was implemented using LISA and the CoWare Processor Designer Platform. The instruction set of the processor included arithmetic, branch, logical and data transfer instructions. The functionality of all the instructions was checked and found to be correct using Processor Debugger. The same model was then optimized to an ASIP, an FIR filter in our case.

According to the profiling results, the optimization was with respect to resources like data memory, program memory, instruction set and number of general purpose registers. The RTL for both the processors was generated and synthesized. The synthesis results were compared and ASIP was found to be much better than the general purpose processor in terms of power and area. Thus the CoWare design flow was explored. By considering the profiling any ASIP can be implemented and optimized taking our general purpose processor as a reference.

The generated RTL was simulated using ModelSim. The functionality of the processor implemented in LISA and its corresponding generated RTL was exactly same.

The final Layout was drawn using Magma Blast Create and Blast Fusion Tool. Finally, the processor was dumped to an FPGA board, (Xilinx Virtex-II Pro). The FPGA implementation of the processor was found to be accurate.

References

1. “Embedded DSP Processor Design” by Dake Liu
2. Anupam Chattopadhaya, Arnab Sinha, Dandian Zhang, Rainer Leupers, Gerd Ascheid, Henrich Meyr, ”Integrated Verification approach during ADL driven processor design”, Microelectronics journal 40(2009), page 1111-1123.
3. Welhua Sheng, Jianjiang Ceng, Manuel Hohenauer, Hanno Scharwachter, Rainer Leupers, Henrich Meyr, Institute for Integrated systems, Achen, Germany, ”A novel approach for fexible and consistent ADL driven ASIP design”, DAC’04, June 7-11, 2004, San Diego, California, USA.
4. Andreas Hoffman, Member IEEE, Tim Kogel, Achim Nohl, Gunnar Braun, Oliver Schliebush, Oliver Wahlen, Andreas Wieferink and Henrich Meyr, Fello, IEEE, “A novel methodology for the design of application specific instruction set processors (ASIPs) using a machine description language”. IEEE transaction on Computer Aided Design of integrated circuits and systems, vol-20, number 11, Nov.-2001.
5. O. Schliebusch, A. Chattopadhayay, E M Witte, D Kammler, G. Ascheid, R Leupers, H Meyr, ”Optimization techniques for ADL driven RTL processor synthesis” in IEEE workshop on rapid system prototyping(RSP), Montreal, Canada, June 2005.
6. U K Nanda, K K Mahapatra, “Design of a pipelined FIR filter using Architecture Description Language”, National Conference on Wireless Communication and VLSI Design 2010, GEC, Gwalior, India
7. CoWare, The ESL design Leader reference manuals, Product version V2007.1.2, June-2008. “Embedded System Design, A Unified Hardware/Software Introduction” by Frank Vahid, Tony Givargis
8. “Digital Integrated Circuits, A Design Perspective” by Jan Rabaey, Anantha Chandrakasan, Borivoje Nikolic
9. CoWare, inc, http://www.coware.com
10. MAGMA Blast Create and Blast Fusion Manuals.
11. http://www.xilinx.com/company/gettingstarted/index.htm#CLB

12. www.xilinx.com/ise/.../chipscope_pro_sw_cores_10_1_ug029.pdf
13. http://www.xilinx.com/univ/XUPV2P/Documentation/ug069.pdf