DESIGN OF AN APPLICATION SPECIFIC INSTRUCTION SET PROCESSOR USING LISA

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Technology

in Electronics and Communication Engineering

by

Umakanta Nanda

Department of Electronics and Communication Engineering
National Institute of Technology
Rourkela, Orissa, 769 008, India

May 2010

DESIGN OF AN APPLICATION SPECIFIC INSTRUCTION SET PROCESSOR USING LISA

A Technical Report submitted in partial fulfillment
of the requirements for the degree of

Master of Technology

in

Electronics and Communication Engineering

by

Mr. Umakanta Nanda

under the guidance of

Prof. K.K.Mahapatra

Department of Electronics and Communication Engineering
National Institute of Technology
Rourkela-769 008, Orissa, India

May 2010

To my parents

Department of Electronics and Communication Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Orissa, India.

Certificate

This is to certify that the work done for the direction of thesis entitled **"Design of an Application Specific Instruction set Processor using LISA"** submitted by ***Mr.Umakanta Nanda*** in partial fulfillment of the requirements for the award of Master of Technology Degree in Electronics and Communication Engineering with specialization in VLSI Design and Embedded Systems at National Institute of Technology, Rourkela is an authentic work carried out by him under my supervision and guidance.

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

Place: NIT Rourkela
Date: 28th May, 2010

Dr.Kamala Kanta Mahapatra
Professor

Acknowledgement

My first thanks are to the Almighty God, without whose blessings I wouldn't have been writing this "acknowledgments".

I then would like to express my heartfelt thanks to my guide, Prof.Kamalakanta Mahapatra, for his guidance, support, and encouragement during the course of my master study at the National Institute of Technology, Rourkela. I am especially indebted to him for teaching me both research and writing skills, which have been proven beneficial for my current research and future career. Without his endless efforts, knowledge, patience, and answers to my numerous questions, this research would have never been possible. The experimental methods and results presented in this thesis have been influenced by him in one way or the other. It has been a great honor and pleasure for me to do research under supervision of Prof. Kamalakanta Mahapatra.

I am very much indebted to Prof. S. K. Patra, Head of the Department, Electronics and Communication Engineering, National Institute of Technology, Rourkela for his support during my work.

I am grateful to Dr. D. P. Acharya for teaching me the right way to present the motivation of my thesis. His insightful feedback helped me improve the presentation of the thesis in many ways.

I am also thankful to Prof. G. S. Rath, Prof. S. K. Das, Prof. S. K. Behera, Prof. NVLN Murty for giving encouragement during my thesis work.

I thank all the members of the Department of Electronics and Communication Engineering, and the Institute, who helped me by providing the necessary resources, and in various other ways, in the completion of my work.

Finally, I thank my parents and all my family member for their unlimited support and strength. Without their dedication and dependability, I could not have pursued my MTech. degree at the National Institute of Technology Rourkela.

Umakanta Nanda

Abstract

A Digital Signal Processor with specific instruction sets and meant for a specific application is called as Application Specific Instruction set Processor(ASIP). To design an ASIP many approaches are available. However optimization of an ASIP becomes handy if it is designed in a higher level of abstraction that is higher than Register Transfer Level (RTL). Application Description Languages (ADLs) are becoming popular recently because of its quick and optimal design convergence achievement capability during the design of ASIPs. Several stages are required to design a processor which are architecture design implementation, software development, instruction and system verification. Verification of such ASIPs at various design stages is a tedious job to do. This thesis presents the architecture description of a simple DSP processor using ADL based instruction set description. The design process is more consistent after allowing maximum flexibility here. Furthermore, it enables the design process in both instruction and cycle accurate modes. The design process of a three stage pipelined FIR Filter processor is demonstrated as a case study. Further optimization can be done with respect to resources, memory size and power consumption by changing the LISA code written in CoWare platform.

Contents

Certificate **i**

Acknowledgement **ii**

Abstract **iii**

List of Figures **vi**

List of Tables **viii**

1 Introduction **2**

1.1 Motivation 2

1.2 Related Work 4

1.3 Organization of this Thesis 5

2 Design Methodology Of ASIP **7**

2.1 Implementation of DSP Application 7

2.1.1 Implementation on General Purpose Processor (GPP) 7

2.1.2 Implementation on General Purpose DSP Processor 8

2.1.3 Implementation on Application Specific Integrated Circuit (ASIC) 8

2.1.4 Implementation on Application Specific Instruction Set Processor (ASIP) 9

2.2 ASIP Design Flow 10

2.2.1 Architecture Exploration 12

2.2.2 Architecture Implementation 13

2.2.3 Software Application Design 14

2.2.4 System Integration and Verification 14

2.3 Field of Application 15

3 Overview Of LISA **18**

3.1 Building a LISA model 18

3.1.1 Modeling Instructions 20

3.1.2 Operation Hierarchy Of a Processor in LISA model 21

3.2 ISS Design vs Processor Design 21

3.3 Instruction Accurate vs Cycle Accurate Modeling 22

3.4 The Instruction Set Designer 24

3.5 Processor Debugger 25

3.6 Major Benefits 26

4 Test Case: Two Processors Design Comparision **28**

4.1 The Instruction Set Designer 28

4.2 Implementation of General Purpose Processor 31

4.3 Operation profiling 31

4.4 Resource profiling 33

4.5 Memory profiling 35

4.6 Optimized implementation result 35

4.6.1 The generated HDL model structure 36

4.6.2 Comparison of the HDL codes generated 38

4.6.3 Synthesis Report collected from Cadence DC 39

4.7 Layout using MAGMA 40

5 Conclusion **46**

5.1 Main Contributions 47

5.2 Conclusion 47

5.3 Future Work 47

Reference **49**

Dissemination of Work **52**

List of Figures

2.1 DSP Processor Architecture 9
2.2 The ASIP Design flow 11
2.3 Phases of ASIP Design 12
2.4 Comparision of HDL And LISA model 13
2.5 Exploration and Implementation 15

3.1 Resource Section 19
3.2 Operation Section 20
3.3 Operation hierarchy 21
3.4 CoWare Processor Designer 22
3.5 Instruction Set Designer 24
3.6 Processor Debugger Window 25

4.1 Debugger Window 29
4.2 Instruction Set Designer 30
4.3 Part of LISA code of Processor-1 32
4.4 Operation Profiling Window 33
4.5 General Purpose Register Window 34
4.6 Optimized Implementation Result 36
4.7 Generated HDL Code Structure 37
4.8 Toplevel Schematic 39
4.9 Design Objects 40
4.10 Technology Schematic 41
4.11 Blast Create Layout Flow 42
4.12 Blast Fusion Flow 43

4.13 Layout 44

List of Tables

4.1 Comparision Between Two Processors 38
4.2 Synthesis Report 40

Chapter 1

INTRODUCTION

Motivation

Organization of this Thesis

Chapter 1

Introduction

1.1 Motivation

Today's market has a high demand on mobile and automotive devices due to robustness, performance, power, efficiency, flexibility, development time, and price of these systems. Unfortunately the decreasing structure size has the drawback of exponentially increasing non-recurring engineering (NRE) costs. The major factors of the NRE cost breakdown are chip design, chip verification and the development of the mask sets. Further more time to market is also an important factor for any processor. By taking into consideration of all these factors a better balance between the parameters like flexibility, efficiency and speed can be achieved by combining processor and ASIC technology. The result is an application specific instruction-set processor (ASIP). ASIPs [1,2] combine the two advantages of processors and ASICs: reconfigurability and efficiency with respect to performance, power and area. If we consider cost function ASIP design achieve an overhead factor of $10^3 - 10^7$ compared to ASIC implementation. They can be optimized in the form of instruction set, general purpose registers, memory size. Compared to digital signal processors (DSPs have ATE-costs of $10^7 - 10^8$) ASIPs are tailored to a smaller domain of applications. This allows for dedicated optimizations and significantly reduces ATE-costs. But ASIP design becomes challenging when we go for hardware model which is required for synthesis. So a software tool suite needs to be created and verified which include assembler, linker, simulator, and profiler [2]. Additionally the design process always involves an architecture ex-

ploration phase where architectural alternatives are evaluated and traded for best matching of the design constraints.

We describe an approach for application-specific processor design [3] based on an extendible microprocessor core. Core-based design allows to derive application-specific instruction processors from a common base architecture with low non-recurring engineering cost. The results of this application-specific customization of a common base architecture are families of related and largely compatible processor families. These families can share support tools and even binary compatible code which has been written for the common base architecture. Critical code portions are customized using the application-specific instruction set extensions. We describe a hardware/software co-design methodology which can be used with this design approach. The presented approach uses the processor core to allow early evaluation of ASIP design options using rapid prototyping techniques.

Application-Specific Instruction-set Processors (ASIP) can improve execution speed by using custom instructions. Several ASIP design automation flows have been proposed recently. One can investigate two techniques to improve these flows, so that ASIP can be efficiently applied to simple computer architectures in embedded applications. Firstly, we efficiently generate custom instructions with multi-cycle IO (which allows multi-outputs), thus removing the constraint imposed by the ports of the register file. Secondly, we allow identical portions of different custom instructions to be shared, thus allowing more custom instructions under the same area constraint. To handle the greatly increased exploration space, we propose several heuristics to keep the problem tractable. Experimental results show that we can achieve 3x speedup in some cases.

This thesis particularly focuses on the optimization of a digital signal processor with respect to instruction set, memory and general purpose registers. Then by using Coware tool the RTL file of the processor has been generated to compare the parameters like area, power and lines of HDL code [4] of two processors.

1.2 Related Work

The concept of instruction set oriented ASIPs is well known in the technical literature. In a concise overview of ASIP design issues [5] is given. The reviewed ASIP design flows are targeted at performance constraints and do not take into account the energy consumption of the implementation. Furthermore, the described design flows frequently separate ASIP architectural design space exploration from ASIP instruction set synthesis. In the currentwork, these design steps are combined, because the instruction set is viewed as an interface to the architecture with mutual dependencies. As a consequence, architecture and instruction set are jointly optimized in order to obtain optimum results.

There are various ASIP design tools for the complete ASIP design flow from application to implementation. In the PEAS [6] design environment is described which generates an instruction set simulation model and a synthesizable model from an architectural processor description. The MetaCore DSP development system [7] is an ASIP design tool which supports design space exploration and design generation. In the design flow, the development tools like C compiler, assembler, and ISA simulator as well as the HDL description of the processor are generated. In [8] the ISDL machine description language is used to generate a bit-true instruction level simulator and a synthesizable Verilog processor description.

There are also some design tools presented in the literature focusing on a subset of the ASIP design flow. A framework for Compiler-ASIP codesign with feedback from an optimizing compiler to the ASIP design is described in [9]. In [10] the RECORD compiler is presented which uses a structural RTL model of a DSP as a starting point of the compiler generation.

Furthermore, there are some commercial approaches to ASIP design. For instance, Tensilica, Improv and ARC Cores offer configurable processor cores together with design environments to generate the necessary development tools. For an overview refer to [11].

For the current case study, the processor description language LISA has been used to generate assembler and instruction set simulator as well as parts of the

HDL description. The underlying design methodology provides a power-conscious design flow. Power saving techniques similar to the ones that have been used for MCORE [12] but also ASIP-typical power saving techniques have been applied. Furthermore, by using the quantitative results of these optimizations, the important parameters computational performance, area, and energy consumption have been optimized simultaneously. This is especially challenging due to the large design space which is offered by ASIPs compared to systems using off-the-shelf processors.

1.3 Organization of this Thesis

Chapter 2 of this thesis lets us to know about the methodology of design an ASIP. Different approaches have been described and compared with each other. In chapter 3 the LISA (Language for Instruction Set Architecture) has been described and corresponding tool suits have been analyzed. In chapter 4 we have implemented the architecture of an Embedded DSP processor using LISA where the description for each instruction of the instruction set (of that specific architecture) is described properly in CoWare platform. At last in chapter 5 the main contribution with conclusion has been written. Then the future scope of work of this thesis have been described.

Chapter 2

DESIGN METHODOLOGY OF ASIP

ASIP Design Flow
Architecture Exploration
Architecture Implementation
Software Application Design
System Integration and Verification
Field of Application

Chapter 2

Design Methodology Of ASIP

An ASIP [13] has a dedicated instruction set and dedicated data types. Functions are mapped to subroutines consisting of assembly instructions when designing an ASIP DSP [14]. But at the time of ASIC design, 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, it is becoming more popular to map DSP functions to an instruction set [14] because the challenge of complexity is handled in both software and hardware, and conquered separately.

2.1 Implementation of DSP Application

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

2.1.1 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.

2.1.2 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.

2.1.3 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.

2.1.4 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.

A simplified block diagram of DSP processor architecture is shown in figure 2.1.

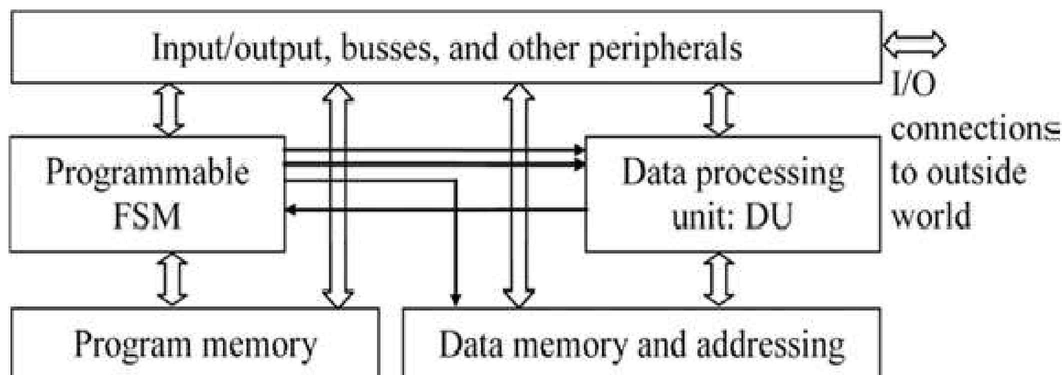

Figure 2.1: DSP Processor Architecture

A DSP processor contains five key components:

- Program memory (PM) is used to store programs (in binary machine code). PM is part of the control path.
- Programmable FSM block consists 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 (DM) stores information to be processed. Three types of data are stored in Data Memory. Those are 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.
- 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.
- 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.

2.2 ASIP Design Flow

Instruction set design is the first and most important step for the design of processor. There is tradeoff among a multiple parameters including performance, functional coverage, flexibility, power consumption, silicon cost and design time.

The complete design flow has been shown in the figure 2.2. It 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.

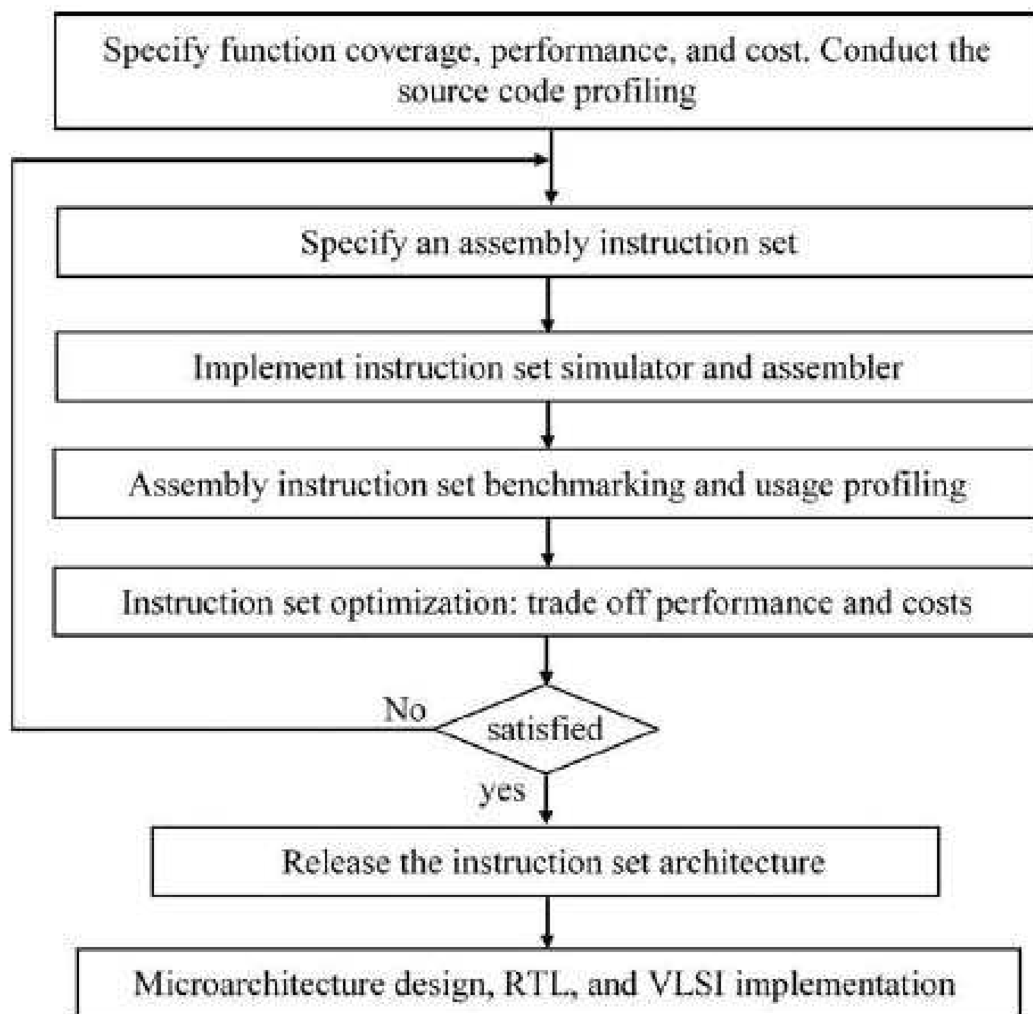

Figure 2.2: The ASIP Design flow

With some automation support by the vendors of embedded processors [15] and integrated circuits, Application Specific Instruction Set Processors design are being carried out traditionally. Four design phases [13, 16]are needed to describe the ASIP design which are shown in the figure 2.3. In all the design phases different development teams are required. So there should exist a good communication between the teams.

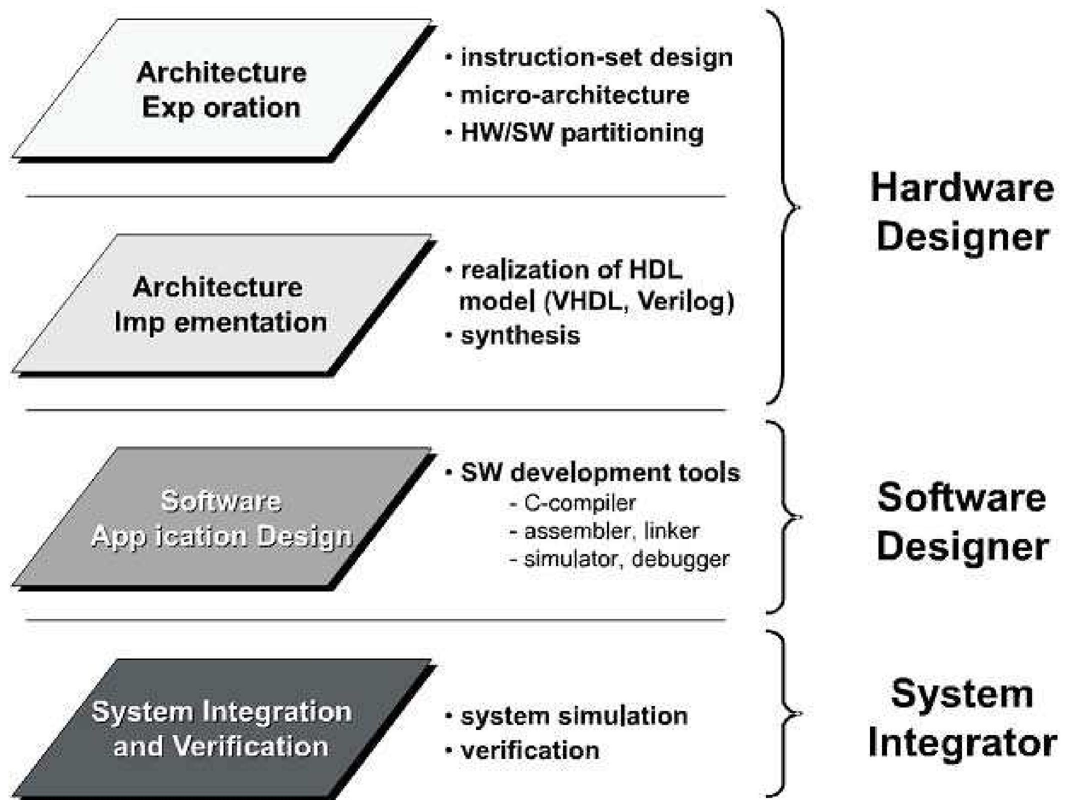

Figure 2.3: Phases of ASIP Design

2.2.1 Architecture Exploration

Architecture exploration phase is used to effectively map an application onto a dedicated processor architecture. Until a hardware implementation is found this process iteratively evaluates the alternatives. Hardware/soft-ware partitioning is also included here. Decisions are made to divide different parts of the application which will be executed either on dedicated hardware circuits or will be implemented in software. This phase has the central component which is the processor model. This is either specified in a low abstraction level that is in hardware description language or in the processor simulator which is in higher abstraction level. The complete micro architecture of the model is described in HDL where as the simulator tells only the architecture aspects of the processor resources, instruction coding, and the temporal behavior of operations.

2.2.2 Architecture Implementation

RTL processor model is created in this phase. Register Transfer Level is a Hardware Description Language (HDL) coding style that describes the processor in the form of registers and interconnected logic. The LISA compiler should derive all the necessary information from the given LISA description [13] since the generated HDL model does not have any predefined components. Then the generated HDL model can be compared to the LISA model [4, 17] components as shown in the figure 2.4.

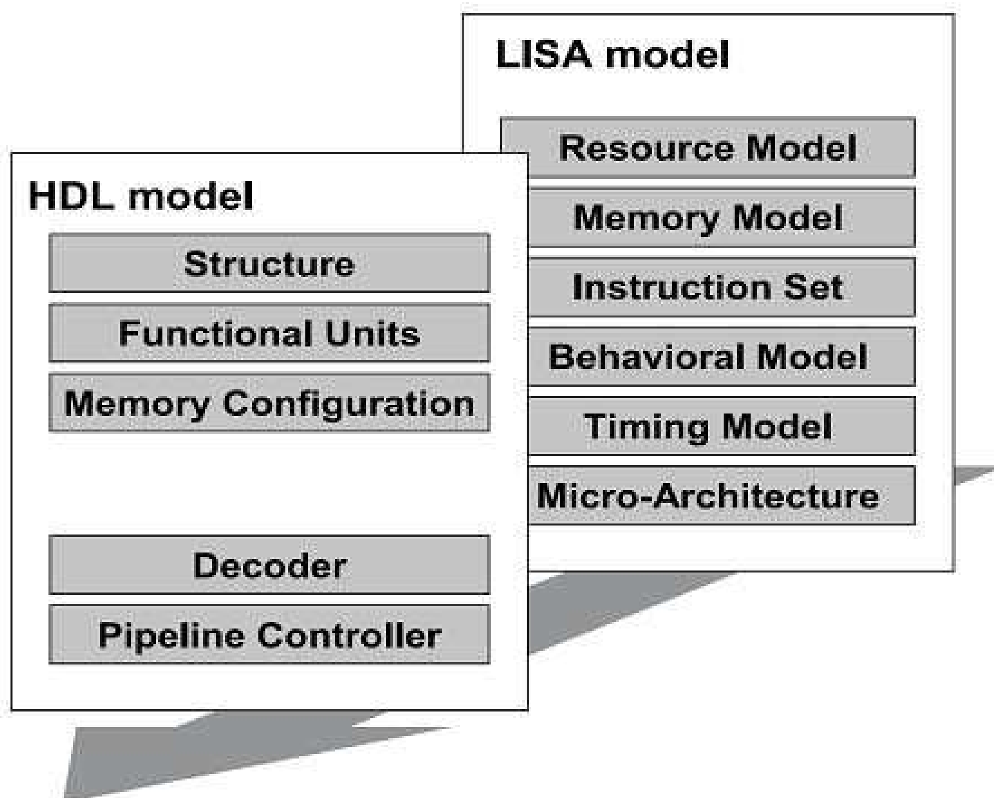

Figure 2.4: Comparision of HDL And LISA model

- LISA memory model derives the memory configuration which summarizes the registers and the memory sets
- Resource models gives the idea about the structure of the architecture such as pipeline stages and pipeline registers.

- Functional units are either generated as empty frames or with fully functionality depending on the HDL language used.
- Coding information in the instruction set model and the timing model results the decoders.
- Pipeline controller is also generated from the above.

The designer will have full control over the generated HDL model with all its components. The generated HDL model can be analyzed with respect to power, area and time constraints and the optimized HDL model can be replaced with the handwritten HDL code written by the experienced designers.

A synthesis tool can be used to generate a gate level netlist automatically which specifies all logic gates and interconnects that are part of the processor model. In an automatic place and route step the location of the gates and the conducting paths are determined. The result of this step is a geometric description of the processor hardware. In this phase no further addition is allowed in the architecture of the programmer's model. Only the architecture can be optimized wrt. Instructions and addressing modes etc. Verification is the major focus here.

2.2.3 Software Application Design

In this phase the software development tools like assembler, linker and debugger are developed those are used to create the application's binary code. Ultimately it is clear that after the architecture exploration phase C compiler is created. Furthermore support libraries (e.g. standard library, floating point emulation) need to be created. Additionally the operation system (e.g. Windows/Unix) needs to be considered. The complete toolchain is usually driven by a graphical user interface - an integrated design environment (IDE), that needs to be developed, too.

2.2.4 System Integration and Verification

A processor simulator without the simulation environment of the entire SOC is not very useful. Through this approach we can interact with other processors, co

processors, ASICs, busses and other peripherals.

2.3 Field of Application

A consistent design flow for system level, processor architecture and software architecture is needed which can be done at LISA processor design platform (LPDP) environment. CoWare Inc. has the commercial version of the above platform. LISA describes the behavior, structure and the input/output interfaces of a processor architecture in a hierarchical manner. Different types of processors are supported by this environment including ARM7, C62, C54x and ASIPs.

Out of the above said four phases mainly two phases are taken into consideration in this project for architecture design. However for implementation purpose hardware description languages are used to model the underlying hardware [17] as shown in the figure 2.5.

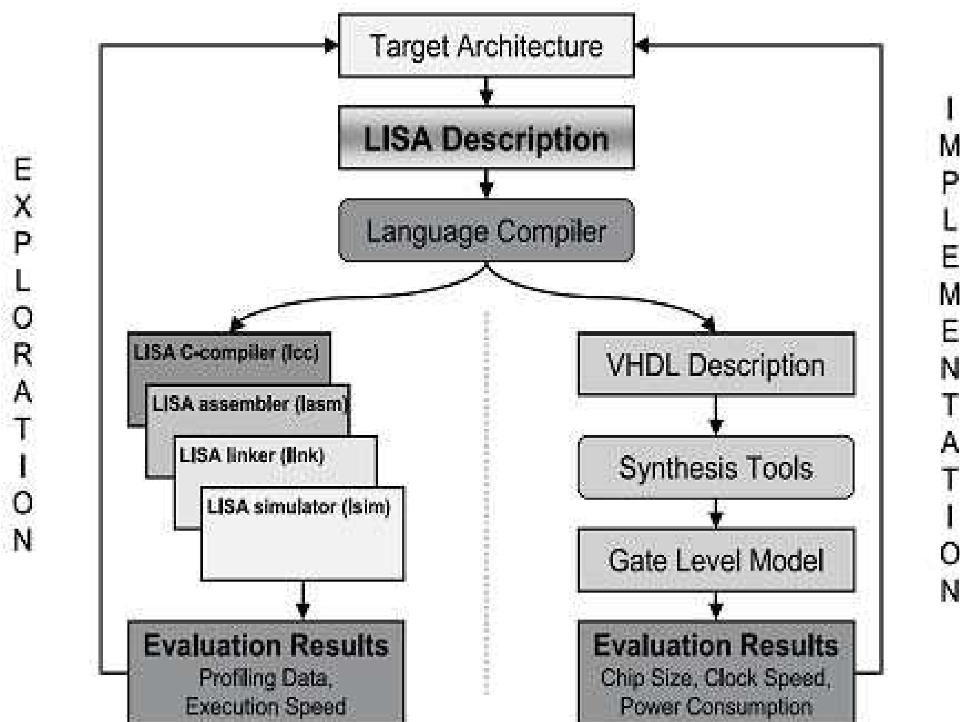

Figure 2.5: Exploration and Implementation

It is very advantageous to combine both of the development process and the

HDL description. Here the LISA compiler can generate both of these. After design exploration and application design the target architecture needs to be implemented.

Chapter 3

OVERVIEW OF LISA

Building a LISA model

Modeling Instructions

ISS Design vs Processor Design

Instruction Accurate vs Cycle Accurate Modeling

The Instruction Set Designer

Processor Debugger

Chapter 3

Overview Of LISA

The acronym of LISA [4] that is "Language for Instruction Set Architecture" give a clear idea that it is a language by which we can model any architecture that is driven by an instruction set. LISA is a mixed behavioral/structural modeling language for the formalized description of programmable processor architectures, their peripherals and interfaces. LISA is having so much flexibility that the elements of this language are generic enough to build any kind of target architectures like general purpose processors, RISC processor, DSPs, ASIPs, and so on. The instruction resource is often a register that is referred as IR (Instruction Register). Instruction resource in LISA can be a memory location, an input pin array, or a concatenation of multiple storage elements.

3.1 Building a LISA model

Generally a processor model written in LISA has two sections those are Resource and Operation section [18]. Again the Operation section contains three subsections those are Coding, Syntax and Behavior.

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.

But in cycle accurate models there are other types of processor resources, like pipeline registers and internal signals [19]. Processor resources are generally declared in the resource section, indicated by the keyword RESOURCE. An example

is shown in figure 3.1.

```
RESOURCE
{
MEMORY_MAP
  {
   RANGE(0x0000, 0x0fff) -> prog_mem[(31..0)];
   RANGE(0x1000, 0x1fff) -> data_mem[(31..0)];
  }
 /* 0x1000 32bit words of data memory   */
 /* FLAGS are set to R/W meaning that data_mem is readable and writable */
 MEMORY uint32 data_mem
  {
   SIZE(0x1000);
   BLOCKSIZE(32);
   FLAGS(R/W);
  };
 /* 0x1000 32bit words of program memory */
 /* FLAGS are set to R/X meaning that prog_mem is readable and executable */
 MEMORY uint32 prog_mem
  {
   SIZE(0x1000);
   BLOCKSIZE(32);
   FLAGS(R/X);
  };
 /* Register file with 32 registers    */
 REGISTER int32 GPR[0..31];
 /* Fetch program counter register*/
 PROGRAM_COUNTER uint32 FPC;
 /* 32 bit instruction register */
 REGISTER uint32 IR;
}
```

Figure 3.1: Resource Section

As shown in the above example a resource declaration typically consists of an identifier, a data type specifier, and an optional keyword defining the semantic type of the resource. All resources that are declared in a resource section are global to the entire LISA model. The resource identifier must be unique in the whole LISA model.

Registers are declared within the resource section using the keyword REGISTER. Here there are 32 general purpose registers having 32 bits each. Every LISA model needs to have a unique resource that is labeled as program counter, using the keyword PROGRAM COUNTER. This information is used by the Processor Debugger [14]. Here memory of the processor has been declared using the keyword MEMORY MAP and different subordinate keywords are evaluated by the

Processor Designer tools that include Processor Debugger, Compiler Generator, and Processor Generator to extract the information which memories are present in the model, and what are their parameters.

3.1.1 Modeling Instructions

The concept of a LISA operation is explained in this section with the help of a flat example.

```
OPERATION addi
{
 DECLARE
 {
  REFERENCE dest;
 }
 CODING  { 0b001000 }
 SYNTAX  { "addi" }
 BEHAVIOR
 {
  /* Compute and write-back the value into the destination register */
  GPR[dest] = operand1 + operand2;
 }
}
```

Figure 3.2: Operation Section

In this flat example. Assume that we would like to model an instruction that adds two particular values and writes the result to a particular general-purpose register.

Operation section [14] usually uses the the keyword OPERATION to initialize any operation which describes about the particular instruction. In the BEHAVIOR section of the model the instruction behavior is described which is in C block code. In the syntax part the assembly syntax of an instruction is modeled. This section starts from the keyword SYNTAX. The binary image or coding of an instruction is modeled in the coding section of an operation. The coding section consists of the keyword CODING. The coding consists of sequence of bit fields or terminal bit patterns which consists of the prefix "0b" followed by "0", "1" or "X" (Don't care).

3.1.2 Operation Hierarchy Of a Processor in LISA model

According to the LISA 2.0 description of the processor, the processor designer generates the software development tools. In figure 3.3 the hierarchy of the instructions [18] has been shown for an assembly language code to find out the convolution of two sequences using FIR filter.

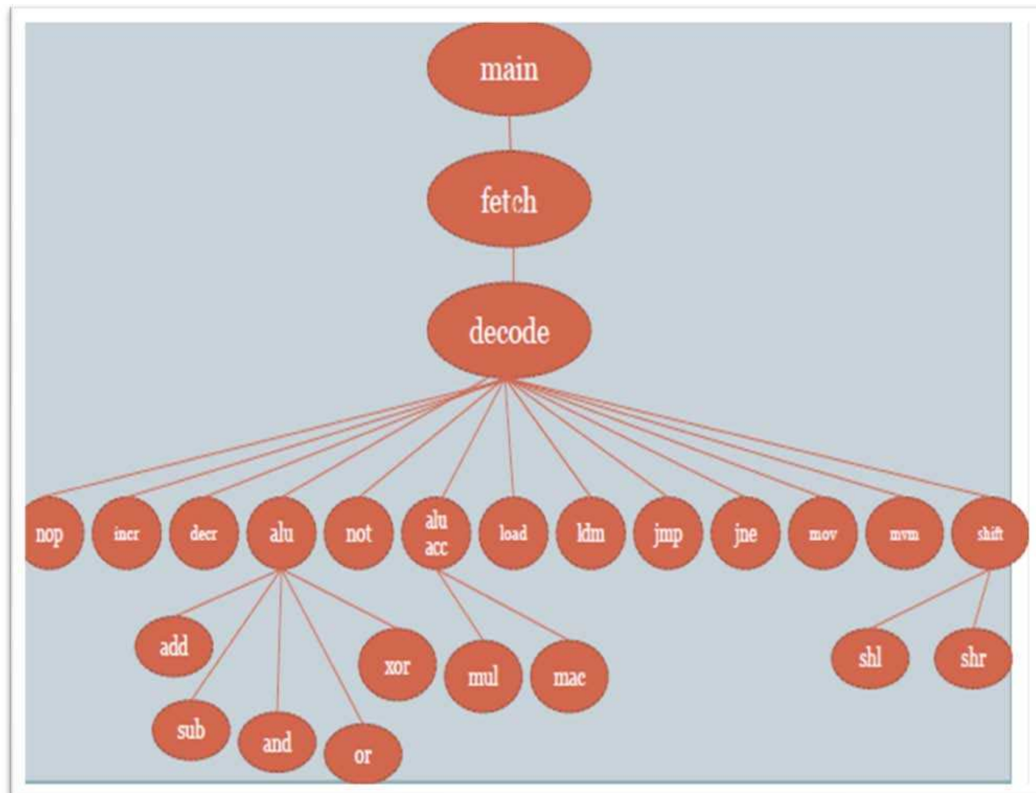

Figure 3.3: Operation hierarchy

3.2 ISS Design vs Processor Design

LISA can be utilized as a unique language by designers with very different intentions. Depending on the intention, different use models of the LISA language and the Processor Designer product family are distinguished. The two main used models [18] are processor design and instruction set simulator (ISS) design.

ISS design is required to model the processor that allows to simulate its instruction set at very high speed. The simulation speed in terms of instructions per second is the main metric for the model quality.

On the other side, processor design is useful for them whose intention is mainly to design a processor, or any feature around the processor that needs to be aware of the processor architecture. Here LISA works as an Architecture Description Language.

3.3 Instruction Accurate vs Cycle Accurate Modeling

Instruction accurate modeling is a synonym for ISS model [18] because in every simulation control step, the complete behavior of an instruction is executed instantly. Instruction Set Simulator has no notion of pipeline.

In pipelined architectures [14], a single instruction is executed in the span of multiple clock cycles. Thus multiple instructions are simultaneously active. In processor design we do need a cycle accurate description of the processor architecture [20] in order to reflect the effect of the pipeline effects. So the cycle accurate modeling is used as a synonym for ISS design.

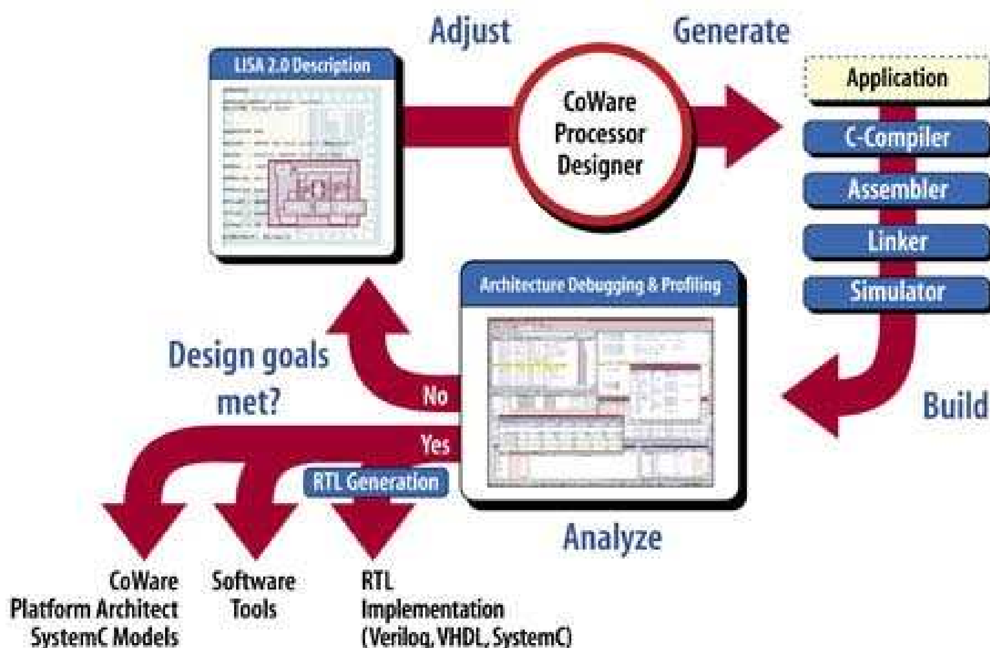

Figure 3.4: CoWare Processor Designer

This language is more suitable with the processor designer tool called CoWare

[21] for its advanced and flexible features such as,

- Automatic generation of synthesizable RTL with both control and datapath.
- Accurate profiling capabilities for high speed instruction set simulator.
- Compatible with extensively used synthesis tool like SYNOPSYS [22]and physical design tool like MAGMA [23].
- Software development tool generation like assembler, linker, debugger, C-compiler.
- Integrated profiling [16] helps to optimize instructions for the target architecture.
- Enables the design team to develop flexible and reusable ASIPs rapidly.

The design flow of an ASIP [21] is shown in the figure 3.4. As illustrated in figure 3.4 LISA 2.0 is a language for processor description which incorporates all processor-specific components such as pipelines, pins, register files, memory and caches, and instructions. The efficient automatic generation of ISS(Instruction Set Simulator) is generated as well as the complete suite of software development tools, like Linker, Archiver, Assembler, and C-Compiler, and synthesizable RTL code. Having extensive profiling capabilities [14] the development tools 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. Furthermore Processor Designer enables the designer to optimize processor micro-architecture [16], instruction set design and memory sub-systems including caches.

Operating at a high level of abstraction [24], Processor Designer not only eliminates the time and cost inherent in HDL-based processor design and manual tool development, but also enables hardware and software designers to customize the instruction set to their needs.

3.4 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.

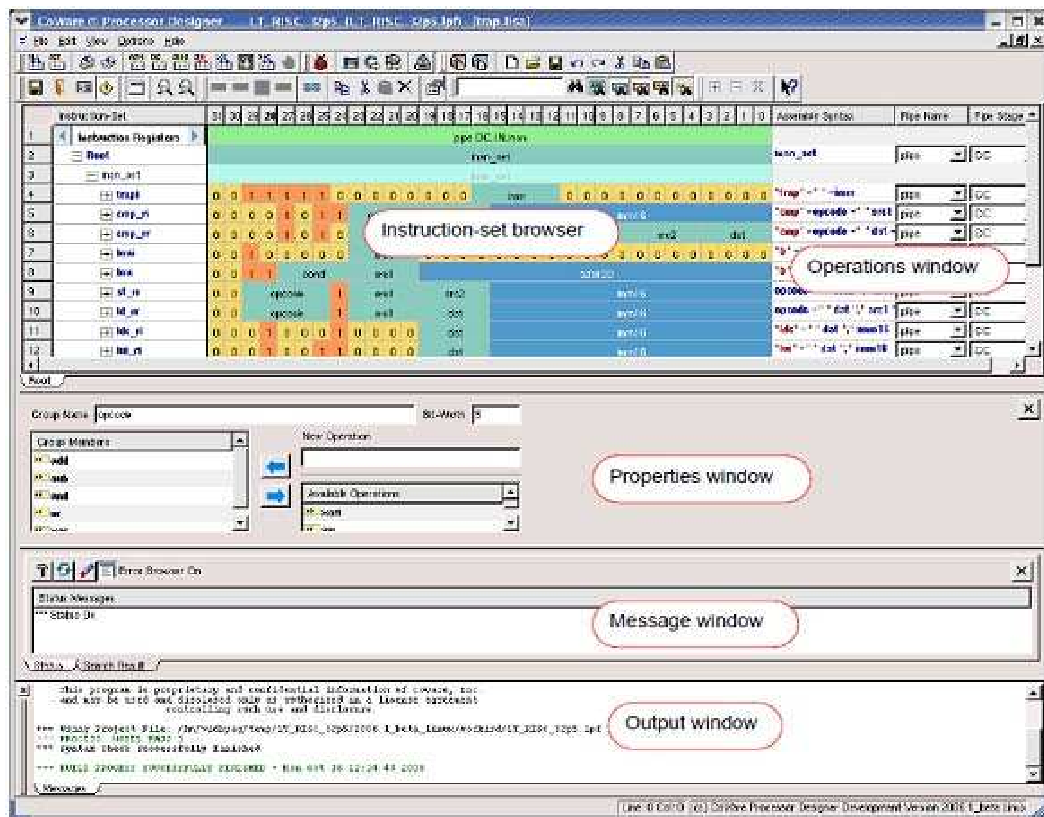

Figure 3.5: Instruction Set Designer

The Instruction-Set Designer [18] 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.

3.5 Processor Debugger

The Processor Debugger GUI allows us 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.

Furthermore, this GUI [18] 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 debugger GUI can either be connected to a single Processor Designer simulator back-end or to an embedded simulator in a SOC simulation. The underlying ISS is derived from the LISA 2.0 model of the processor architecture. This simulator may be run either as a stand alone application, or alternatively it may be attached to the graphical Processor Debugger.

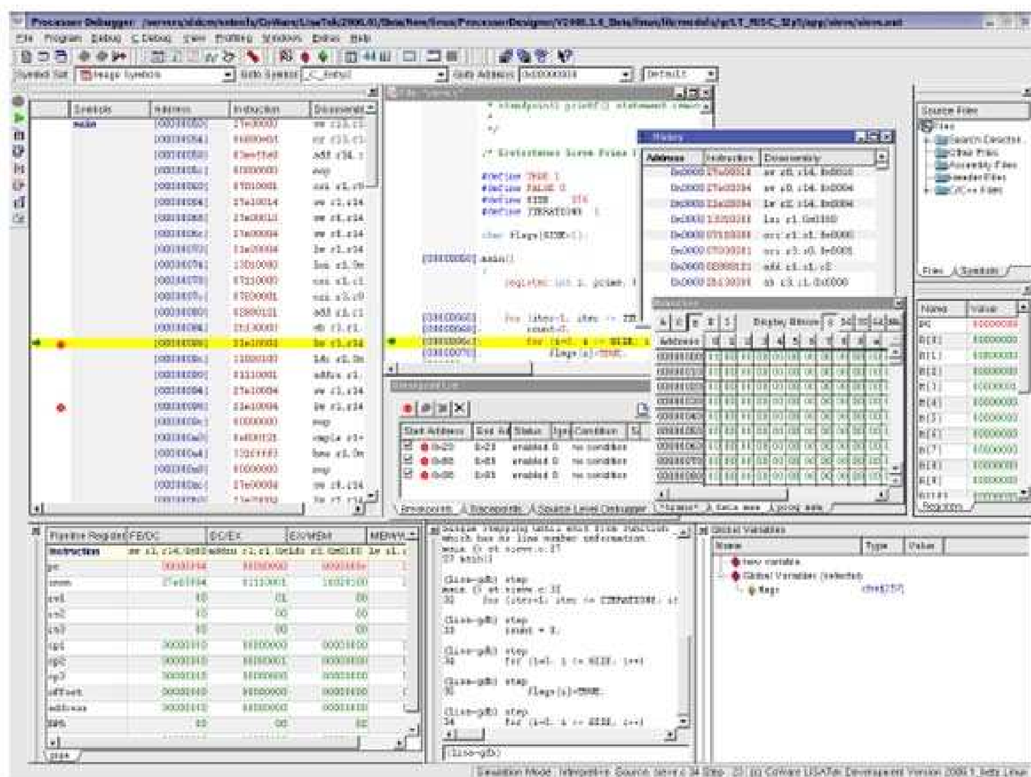

Figure 3.6: Processor Debugger Window

3.6 Major Benefits

- Design teams can rapidly develop flexible and re-usable application specific embedded processors section [20] which include essential SoC functionality [19], through:
  - Rapid architecture design with LISA 2.0 by any designer conversant with C/C++
  - Automatic generation of software development tools and simulator [21]
  - Instruction set profiling and optimization are easy to meet or beat performance objectives
  - Synthesizable RTL for both control and datapath hardware can automatically generated, with robust links to established RTL simulation and synthesis tools
  - An automated , unified methodology that ensures consistency of hardware implementation, simulation model and software development tools implementations with the high level design specification
- Enables embedded software application development and debug with greatly reduced time to market through:
  - Early commencement of software development
  - Reduced software application design and development time
  - Fast and accurate instruction set simulator

Chapter 4

TEST CASE: TWO PROCESSORS DESIGN COMPARISION

The Instruction Set Designer

Implementation of General Purpose Processor

Operation profiling

Resource profiling

Memory profiling

Optimized implementation result

Chapter 4

Test Case: Two Processors Design Comparision

A simple FIR filter with three stage pipelining is implemented here with the help of LISA in Coware platform [21]. Then the resource section of this model has been optimized. A major decrease in total architecture design time can be seen, as the LISA model results from the design exploration phase.

The software development tool suit includes assembler, linker and simulator as well as a graphical debugger frontend. The tools are the enhanced version of those tools used for architecture exploration. The enhancements for the software simulate the ability to graphically visualize the debugging process of the application under test. The LISA debugger frontend [18] is a generic graphical user interface for the generated LISA simulator as shown in the figure 4.1.

It visualizes the internal state of simulation process. Here the C source code, the disassembly of the application as well as all the configured memories and registers (pipeline) are displayed. In this frontend all contents can be changed at the run time of the application. Tools like assembler and linker can be enhanced in functionality as well. More than 30 assembler directives, labels and symbols are supported by the assembler.

4.1 The Instruction Set Designer

Through this Graphical User Interface [18] we can view, edit and create any processor model. By understanding its hierarchy it is much easier to design any

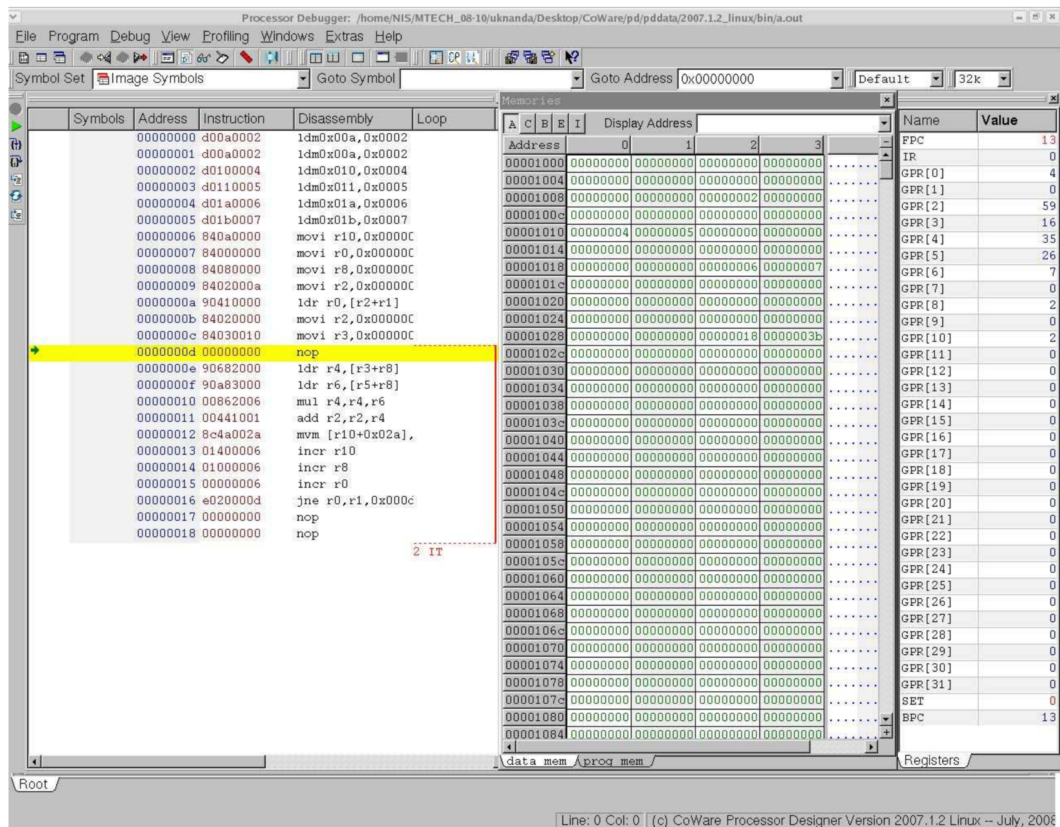

Figure 4.1: Debugger Window

processor resource section. 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. The figure 4.2 shows our optimized processor's ISD window.

The Instruction-Set Designer [18] 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.

The processor debugger provides extensive hardware and software profiling capabilities. Operation profiling gives us the information about Calls/Total which shows the proportion of operation executions for a specific operation to all executed operations.

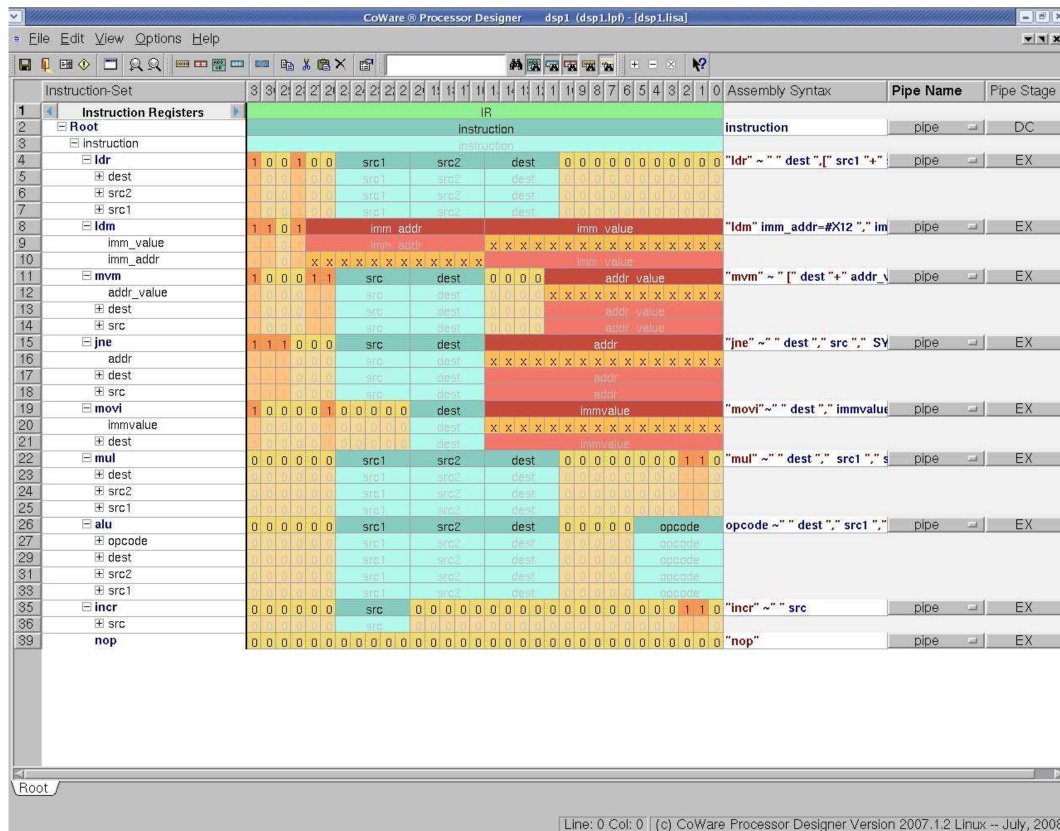

Figure 4.2: Instruction Set Designer

Here in the above debugger window we can see that our processor can understand the assembly code written for the FIR filter with 2 coefficients. Those are A1=4, A2=5 and X1=6, X2=7.

So the output should come as 59(Decimal) and we can see it got the result as GPR[2]=59. Now we can conclude that our processor is correct by giving correct result. The resources we have taken here are:

- General purpose registers: 32
- Instruction set having number of instructions = 15
- Memory(data and program) allocation =0x0000 to 0xffff

4.2 Implementation of General Purpose Processor

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 and having 19 instructions. As stated before, LISA code consists of processor resources and operations. Part of the LISA code of the processor is given in figure 4.2.

To increase the design efficiency and in order to exploit common properties of instructions, operation hierarchy is defined. Figure 3.3 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.

4.3 Operation profiling

The operation profiling window of our processor has been shown in the figure 4.3. For each of the pipelining stages a separate field exists at the bottom of the window. It shows the operations that are located in the respective pipeline stage.

The operations which are not assigned to any pipeline stage are under a separate folder called as main(no pipe).

For each LISA 2.0 operation the following aspects are shown in the Profiling window.

- Name contains the operation name as it is specified in the underlying LISA 2.0 model.
- Calls contains the total number of operation calls (executed operations) for each of the visualized operations.
- Calls/Total shows the proportion of operation executions for a specific op-

```
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{
...
}
```

Figure 4.3: Part of LISA code of Processor-1

eration to all executed operations. As an equation this looks as follows:

$$\frac{calls}{Total} = \frac{Number of specific operations execution}{Number of all opeartions execution}$$

- Similarly Calls/Max contains information containing the proportion of the

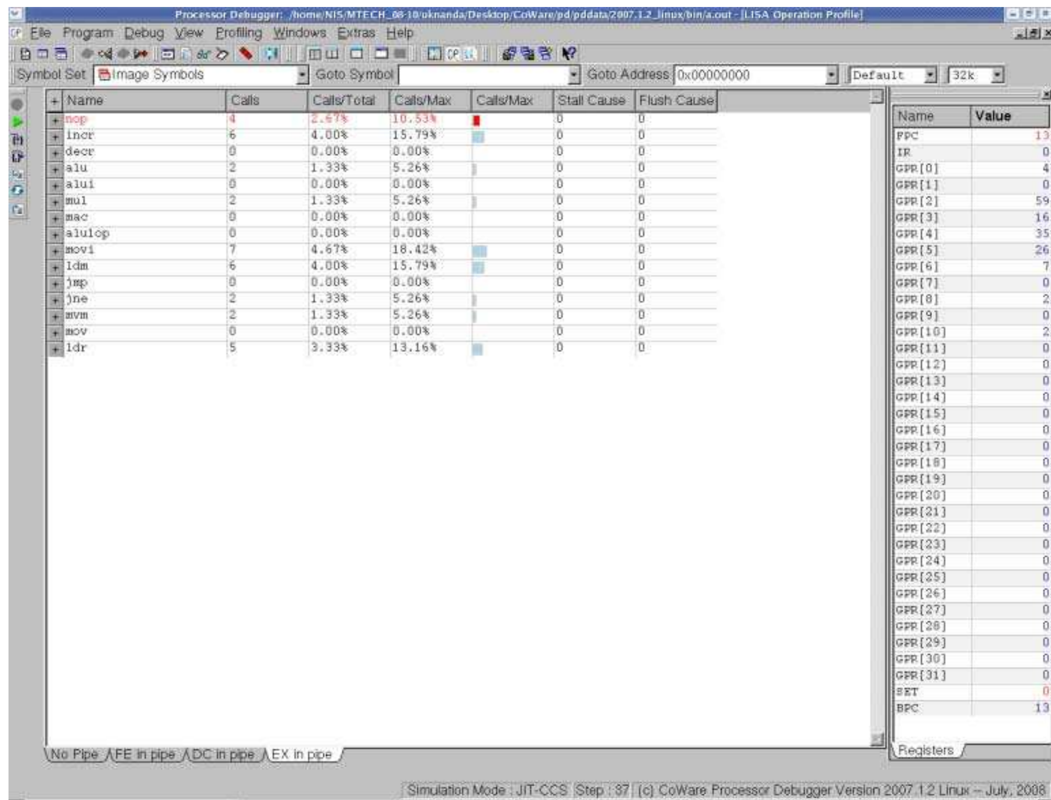

Figure 4.4: Operation Profiling Window

execution of a specific operation to the execution of the LISA operation which has been executed the highest number of times.

$$\frac{calls}{Max} = \frac{Number of specific operations execution}{Maximum number of specific opeartions execution}$$

- These information can be shown graphically also.
- Stall cause shows the total number of stalls invoked by the respective LISA 2.0 operation.
- Flush cause shows the total number of flushes invoked by the respective LISA 2.0 operation.

4.4 Resource profiling

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 as shown in the figure 4.5.

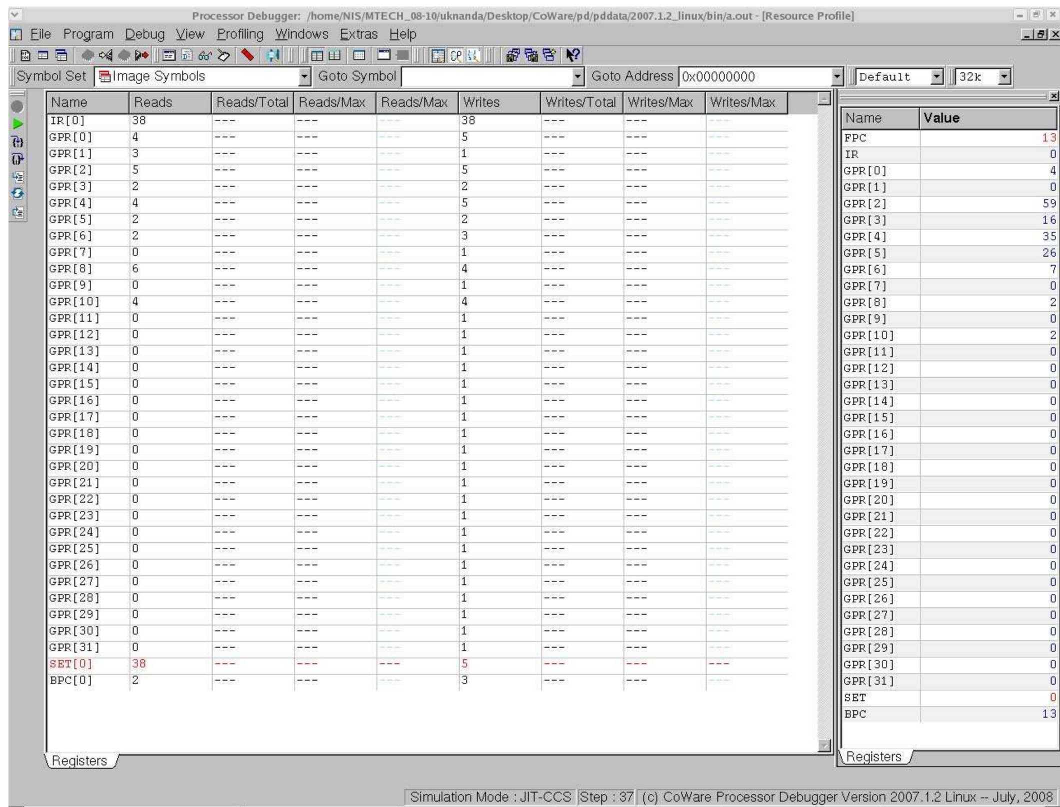

Figure 4.5: General Purpose Register Window

The following information were gathered from the above model:

- Name tells about the name of the resource.
- Reads shows the absolute number of reads on the respective resource.
- Reads/Total Contains the proportion of the reads of the specific resource to the number of total reads of all labeled resources. The equation looks as follows:

$$\frac{Reads}{Total} = \frac{Numberofspecificresourcereads}{Totalnumberofallresourcereads}$$

- Reads/Max: It tells about the proportion of reads of specific register resource to the maximal number of register resource reads is given in this column. The equation looks as follows:

$$\frac{Reads}{Max} = \frac{Numberofspecificresourcereads}{Maximumnumberofspecificresourcereads}$$

- Writes shows the absolute number of writes on the respective resource.

- Writes/Total contains the proportion of the writes of the specific resource to the number of total writes of all labeled resources. The equation looks as follows:
$$\frac{Writes}{Total} = \frac{Number of specific resource writes}{Total number of all resource writes}$$
- Writes/Max shows the proportion of writes to a specific register resource to the maximal number of register resource writes. The equation looks as follows:
$$\frac{Writes}{Max} = \frac{Number of resource writes}{Total number of all resource writes}$$
- These information are visualized graphically also.

The values in the different columns may be sorted by a simple click with the mouse on the top of the column (where the criterion of the respective column is visualized). With one click, the values are sorted in an ascending sequence, with further click in a descending sequence.

4.5 Memory profiling

Similarly memory profiling tells about the access statistics for the memories contained in the processor model. This model has the program memory range 0x0000 to 0x1111 and data memory range 0x1111 to 0xffff.

These profiling information is very much required to optimize our design. This architecture was designed on the respective abstraction level with LISA and software development tools [14] were generated successfully.

4.6 Optimized implementation result

Here in the operation profiling window we can see that the instruction resources like decr, alui, mac, alu1op, jmp, mov have not been called yet. So writing the behavioral code for these instructions is not required. And if we remove these resources from our specific model we can reduce the area without affecting the result. To reduce the area further we can remove the descriptions of the instructions like sub, and, or also.

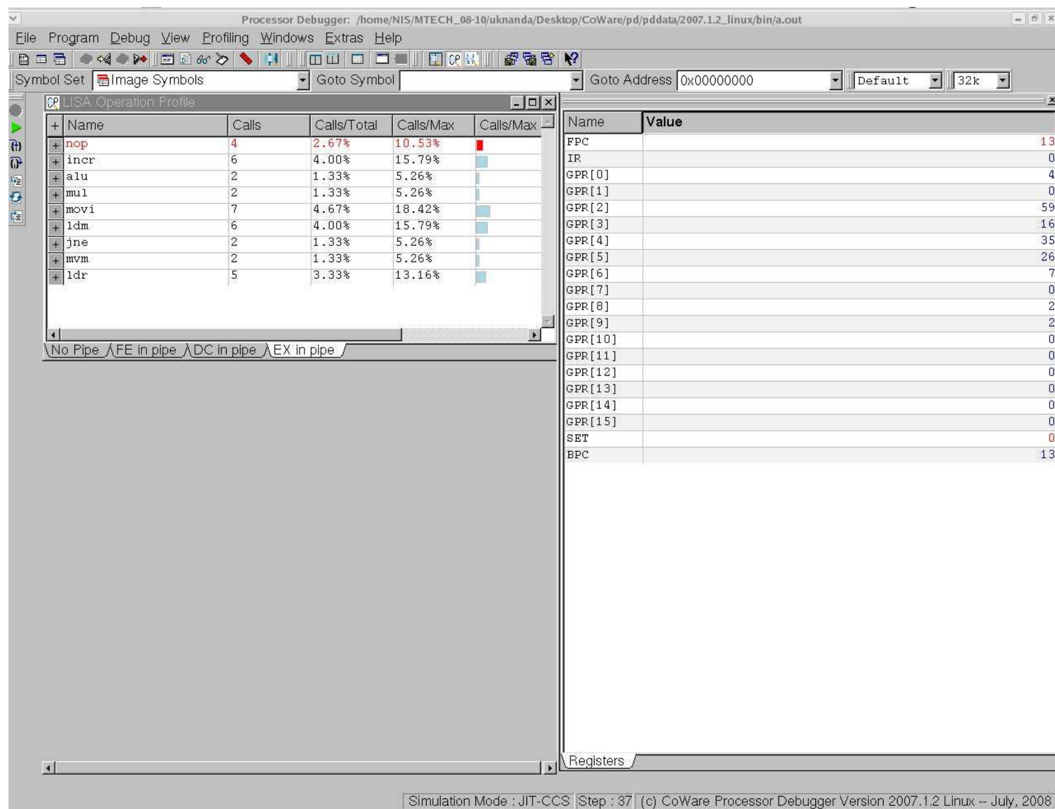

Figure 4.6: Optimized Implementation Result

In the optimized model we have less space allocated for data and program memory. Program memory starts from 0x0000 to 0x0015 and Data memory starts from 0x0016 to 0x0042 reducing the area further.

To reduce the resource section further we can take 16 general purpose registers (GPR) instead of 32 which will reduce the area of our model.It has been shown in the figure 4.6.

4.6.1 The generated HDL model structure

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 4.7.

Resource model and memory model of LISA tells the information about register, memory configuration, pipeline sets and pipeline registers. To generate the base structure of a HDL model this information is used. Different entities are there in the base structure for the register resources, memory resources and the pipeline.

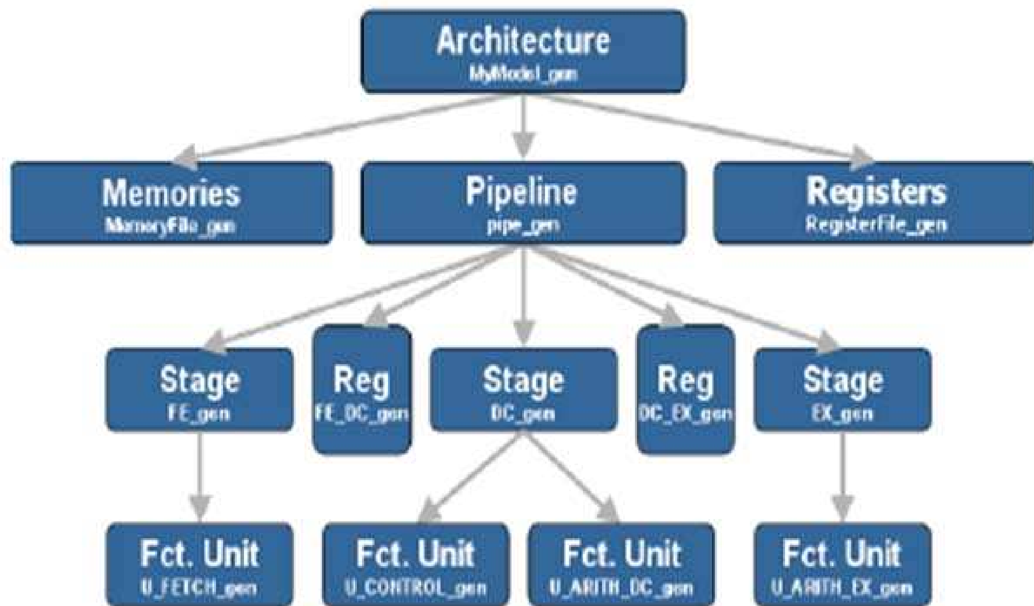

Figure 4.7: Generated HDL Code Structure

To model the register behavior the register resources are completely generated at RTL level. As the memory entity is left empty the designer has the freedom to place any desired memory model into this entity.

In the pipeline there are several entities representing the pipeline registers and stages. Further the pipeline has the controller which has been derived from the LISA model. LISA has the ability to provide a formalized way to initiate several pipeline functions like stall, flush. So the HDL generator can use these information. The pipeline decoder which is placed in the pipeline stage entities drives the pipeline controller. The entities having the functional units are contained in the pipeline stages. More precisely, the functional units implement the data path and will be discussed in detail later. Besides decoder, multiplexers are generated to avoid driver conflicts. the information about the exclusiveness from the coding information included in the LISA instruction set model is derived by the HDL generator. The RTL schematic and the technology schematic of our optimized model are shown in figure 4.8, 4.9 and 4.10 respectivly.

4.6.2 Comparison of the HDL codes generated

The next work in this project is to compare the HDL codes generated from the two different processors. This gives the idea about the number of lines of code of the HDL models it has been observed that the HDL code of our optimized model has very less number of lines compared with that of the previous processor(without optimization). Then both the processors have been compared with respect different parameters like area, power, memory used and number of lines of HDL code.

Table 4.1: Comparision Between Two Processors

Processor	Area(μm^2)	Power(watt)	Memory used(kb)	Lines of HDL code
Processor-1	78122	0.15568	222468	6716
ASIP	30339	0.14122	176268	5070

The RTL was synthesized using Cadence Encounter [25] and the results are tabulated as shown in Table 4.1. The library used for the synthesis was TSMC (65nm). Thus we can see a drastic reduction in the area and power requirement.

The HDL code generated was synthesized using Xilinx ISE 10.1.03 [26]and the RTL Schematics are shown in the figures 4.8, 4.9. Thechnology schematics has been shown in figure 4.10.

In top level schematic which has been shown in figure 4.8 we can see that it has 10 terminals those are:

- Program memory (input)
- Program memory (output)
- Data memory (input)
- Data memory (output)
- Program memory address
- Data memory address
- Clock main

- Reset main
- Data memory
- Program memory

In design objects schematic we can see the internal parts of each and every blocks of the entire architecture. Further we can observe all the interconnects as shown in figure 4.9. Here except 3 blocks all other blocks have not been shown. Lastly in the technology schematic all the blocks have been combined and shown in one window as shown in figure 4.10.

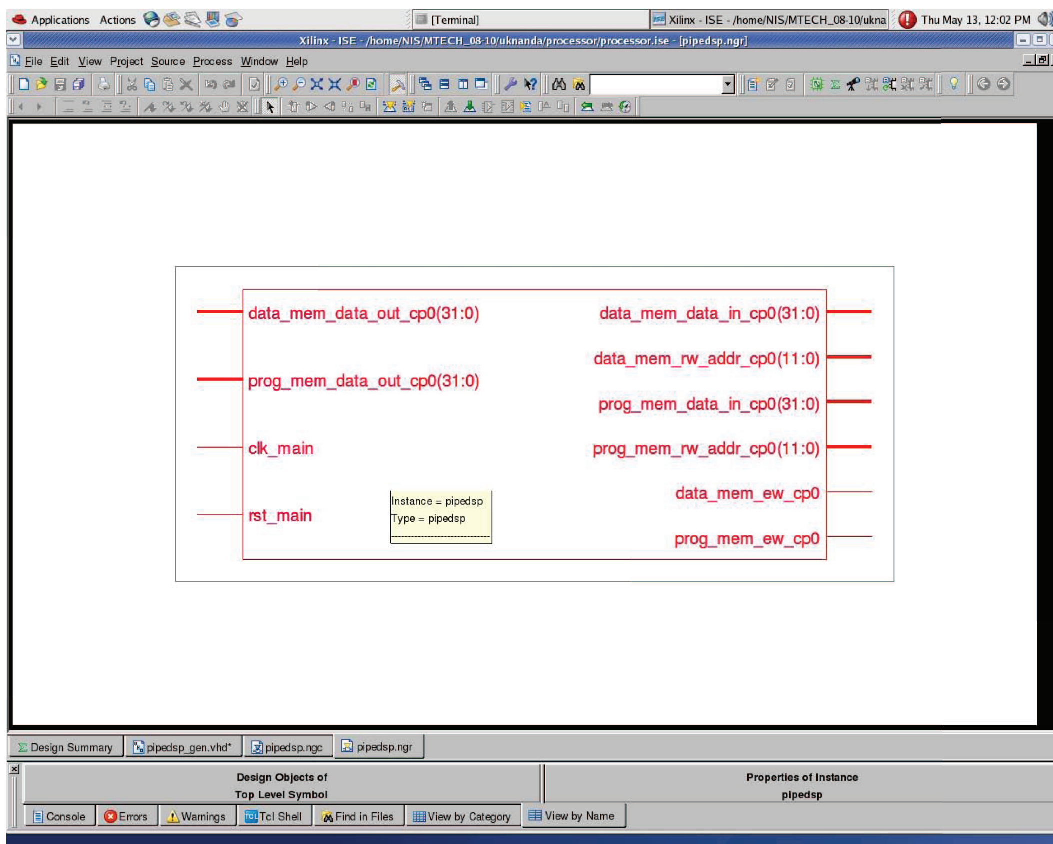

Figure 4.8: Toplevel Schematic

4.6.3 Synthesis Report collected from Cadence DC

Coware supports the universally used synthesis tool like Cadence. So using Cadence DC we have observed the following parameters.

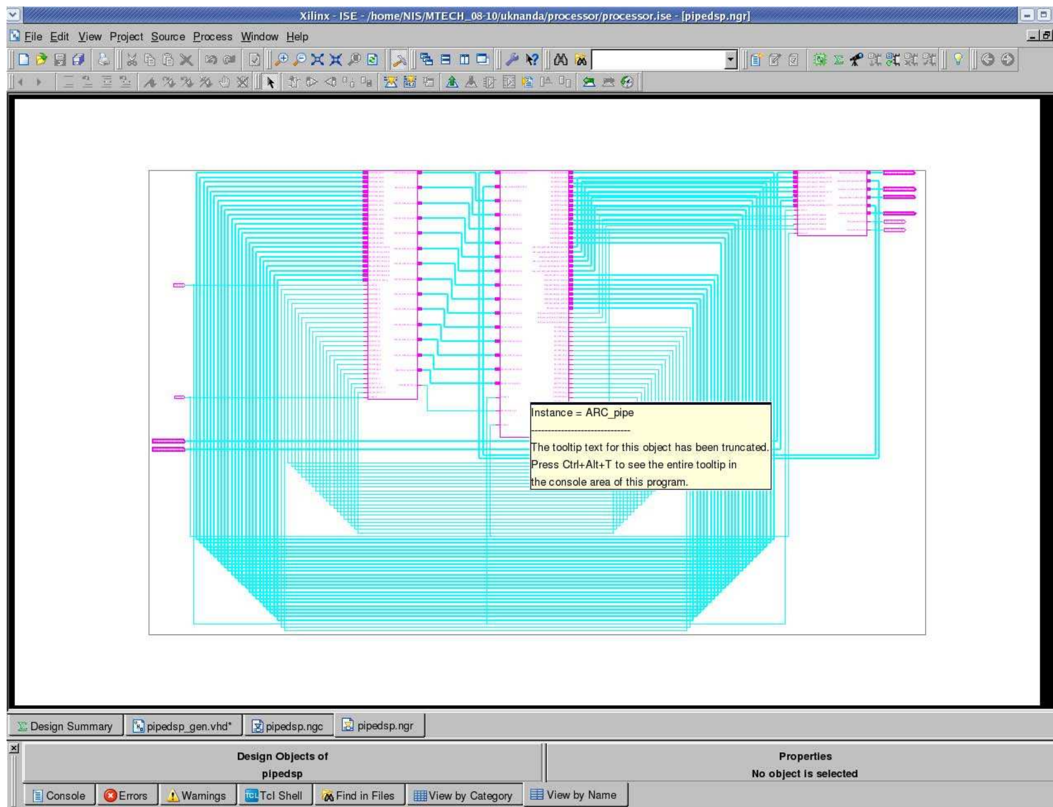

Figure 4.9: Design Objects

Table 4.2: Synthesis Report

Parameters	Used	Total	Percentage
Number of slices	2611	4656	56
Number of slice FFs	640	9312	6
Number of 4 i/p LUTs	5096	9312	54

Here the number of I/Os used is 156 and the Clock period is 20.909ns (frequency: 47.825MHz). Further more we can see that the total memory used here is 576760 kilobytes.

4.7 Layout using MAGMA

The final layout was extracted using MAGMA Blastcreat and Blastfusion tool [23]. 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-

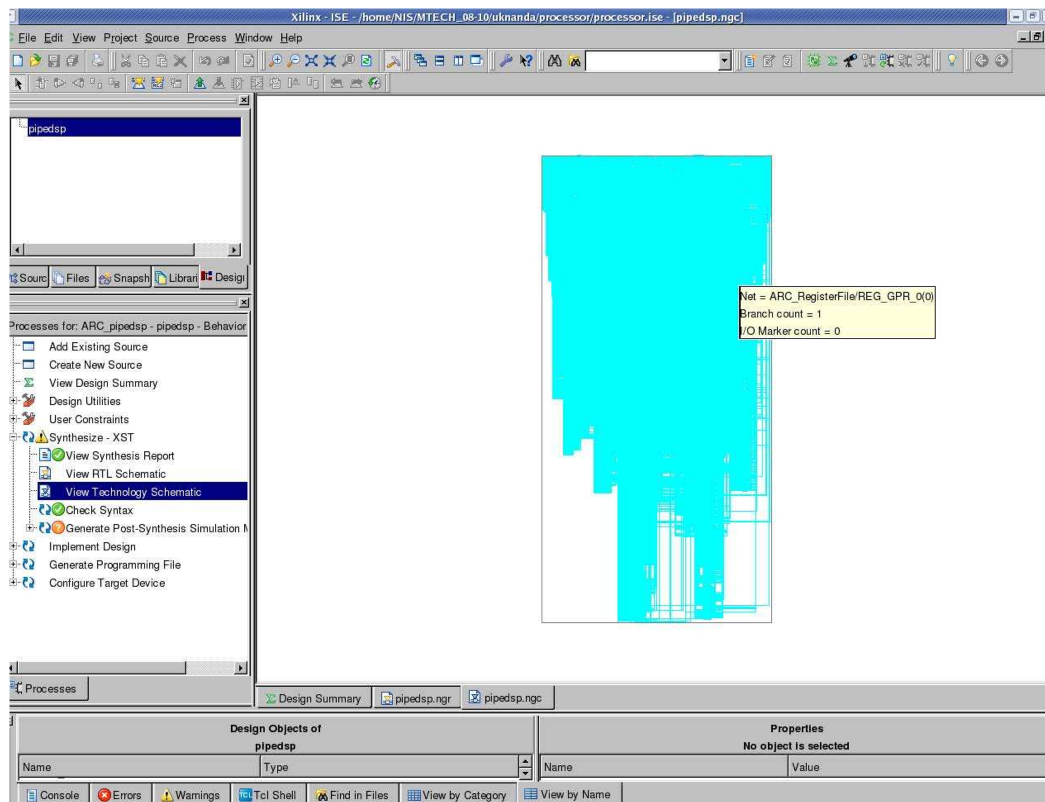

Figure 4.10: Technology Schematic

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 4.11 shows the complete flow of layout and figure 4.13 shows the complete layout of our optimized processor model.

Floorplanning, analyzing and refining the floorplan, power routing, physical implementation and synthesis are possible in the Blast Fusion Environment shown in figure 4.12. 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.

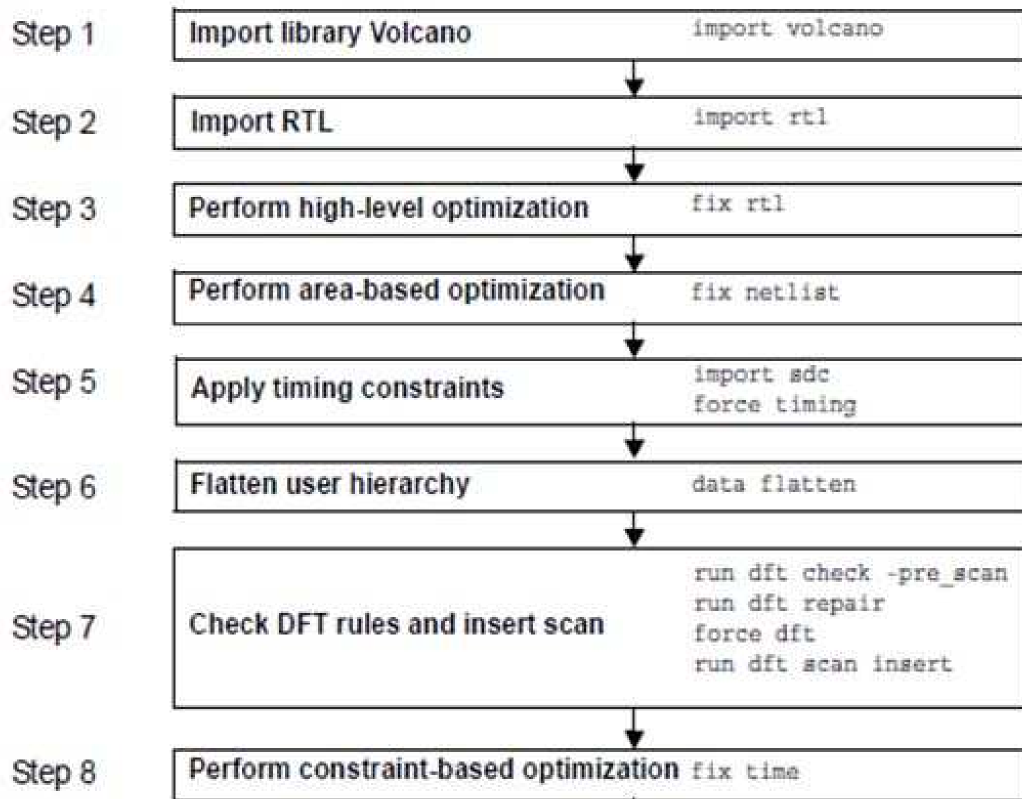

Figure 4.11: Blast Create Layout Flow

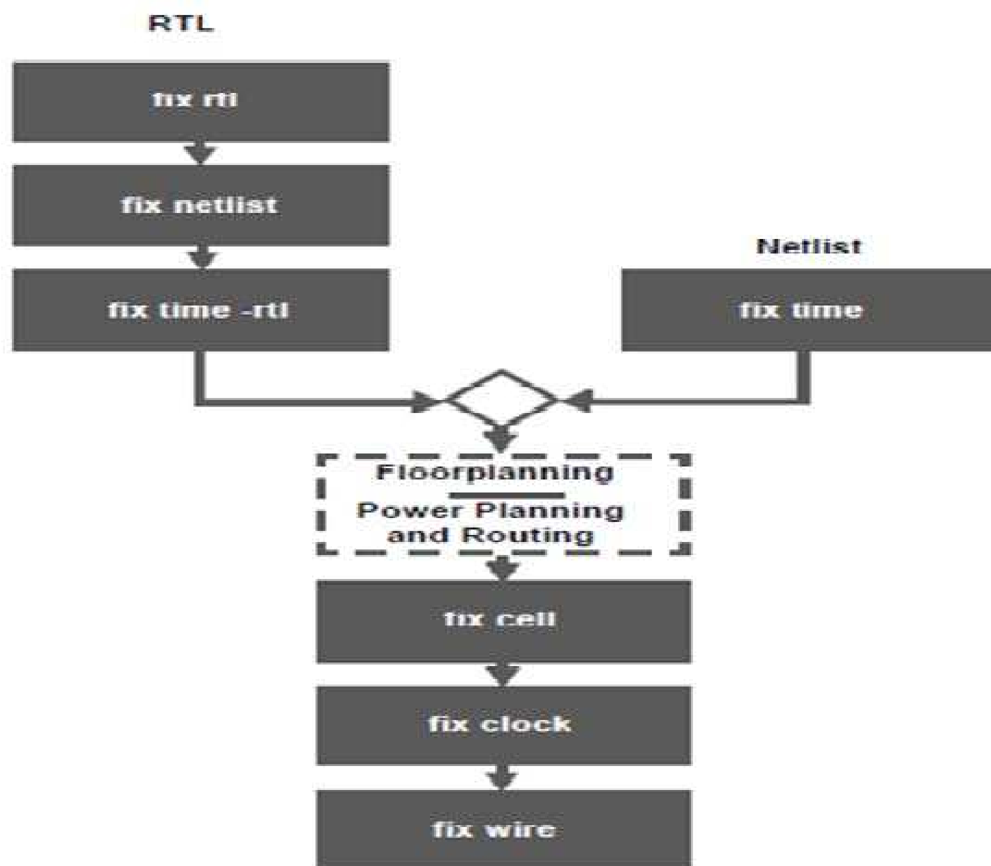

Figure 4.12: Blast Fusion Flow

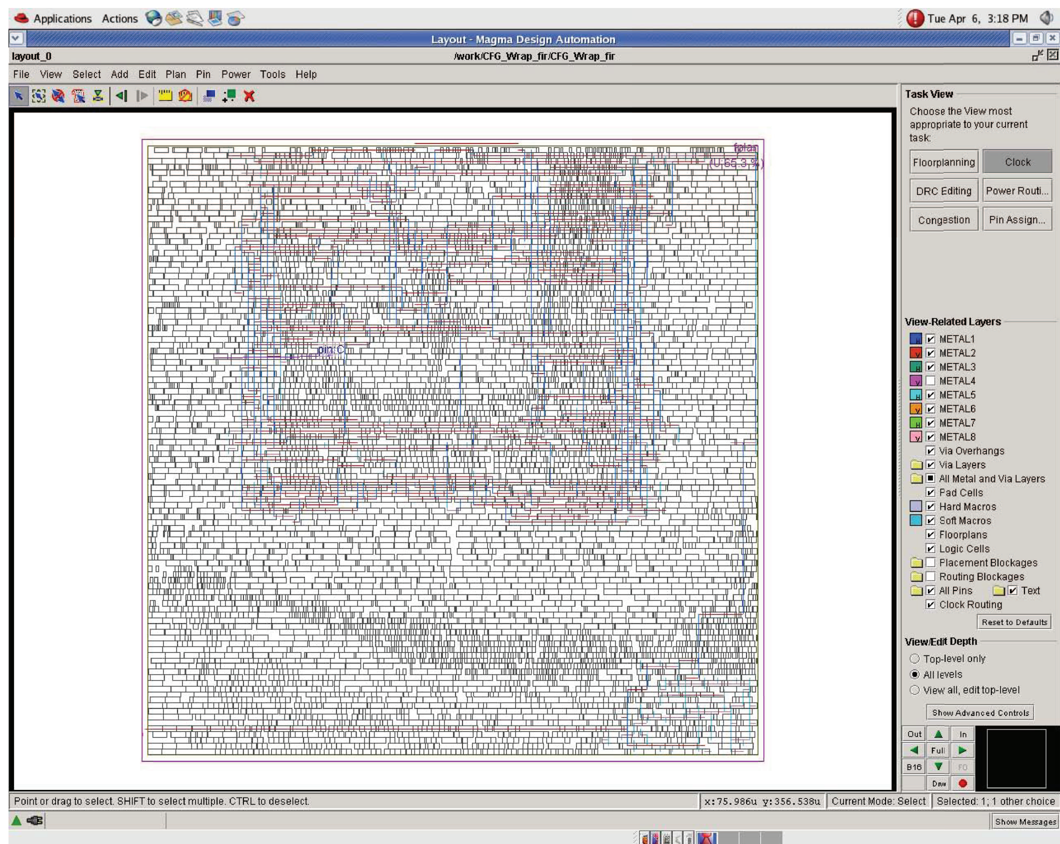

Figure 4.13: Layout

Chapter 5

Summary and Conclusion

Main Contributions
Conclusion
Future Work

Chapter 5

Conclusion

Especially in the mobile and automotive application domain robustness, performance, power eficiency, flexibility, development time, and price per device are opposing design goals that can only be reached with specialized (i.e. application specific) and highly integrated circuits [13]. These goals are the drivers for system on chips (SOCs) that mainly contain fast and power eficient hard wired parts with little flexibility (ASICs) in combination with highly flexible but slow and power hungry programmable parts (Microcontrollers, DSPs).

On the other hand it is quite hard to face the recent trend of applications becoming more versatile and multimedia oriented with this kind of architectures. A more economic compromise between flexibility and power eficiency can be achieved by incorporating application specific instruction-set processors (ASIPs) in the SOC. In this thesis, we have developed a processor with 19 possible instructions. Afterwards we have taken the initiative to design an ASIP(FIR filter). Then we have compared both the processors.

Applications that are becoming more and more complex make an assembly programmers model for the ASIP very tedious and error prone. Thus the utilization of compiler technology - as it is already common in the domain of general purpose processors - is becoming an important productivity factor in ASIP design [13]. A state of the art approach is to implement a C compiler relatively late in the ASIP design process. This chapter concludes the thesis by summarizing the contributions and describing future directions.

The chapter is organized as follows: Section 5.1 highlights the main con-

tributions of the thesis. Finally, Section 5.3 summarizes the results and their implications.

5.1 Main Contributions

In this thesis, using LISA and the CoWare Processor Designer Platform a processor model was implemented. The processor includes 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, area, memory used and lines of HDL code generated. 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.

5.2 Conclusion

This thesis has presented an optimized design of an Application Specific Instruction set Processor. The experimental results reported in the thesis have shown that the proposed ASIP design is better than the general purpose processor with respect to area, power and memory size. Further more we can see that the lines of HDL code of ASIP, generated from CoWare processor designer tool are very much less than the General purpose processor.

5.3 Future Work

In future we can go for designing a complex five stage pipelined FIR filter and we can compare that with a hand written HDL coded design of the same. Further we

can explore our design process by modeling more and more real world processor architectures. How ever the optimized generation of data path, considering the resource sharing issue, is another area of research.

Bibliography

[1] Dandian Zhang Rainer Leupers Gerd Ascheid A Chattopadhaya, A Sinha and Henrich Meyr. Integrated verification approach during adl driven processor design. *Microelectronics journal 40*, 2009.

[2] Manuel Hohenauer Welhua Sheng, Jianjiang Ceng and Hanno Scharwachter. A novel approach for fexible and consistent adl driven asip design. *DAC'04*, June 2004.

[3] Michael Gschwind. Instruction set selection for asip design. In *Yorktown Heights, NY 10598*. Technische University at Wien,Vienna, Austria.

[4] Achim Nohl Gunnar Braun Oliver Schliebush Oliver Wahlen Andreas Hoffman, Tim Kogel and Andreas Wieferink. 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*, 20(11), November 2001.

[5] A Hoffmann T Glo Kler and H Meyr. Methodical low-power asip design space exploration. pages 229–246. Journal of VLSI Signal Processing, Kluwer Academic Publishers, 2003.

[6] et al. M. Itoh. Peas-iii: An asip design environment. pages 430–436. IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors, 2000.

[7] J.-H. Yang et al. Metacore: An application-specific programmable dsp development system. pages vol.8 no.2,173–183. IEEE Transactions on Very Large Scale Integration Systems, April 2000.

[8] P. Russo G. Hadjiyiannis and S. Devadas. A methodology for accurate performance evaluation in architecture exploration. New Orleans, 36th Design Automation Conference, June 1999.

[9] A. Nicolau F. Onion and N. Dutt. Incorporating compiler feedback into the design of asips. pages 508–513. Proc. of European Design and Test Conference, 1995.

[10] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, 1997.

[11] http://www.eetimes.com/story/OEG20001120S-0028 M. Santarini. 2000.

[12] www.mot.com/SPS/MCORE/pdf container/lowpower.pdf. 2001.

[13] Oliver Wahlen. *C Compiler Aided Design of Application-Specific Instruction-Set Processors Using the Machine Description Language LISA*. PhD thesis, Shaker Verlag, 2004.

[14] Y. Bajot and H. Mehrez. Customizable. Dsp architecture for asip core design. *Proc. of the IEEE Int. Symposium on Circuits and Systems (ISCAS)*, May 2001.

[15] Wayne Wolf. *Computers as Components*. Morgan Kaufmann, first edition, 2005.

[16] D Kammler O. Schliebusch, E M Witte and G. Ascheid. Optimization techniques for adl driven rtl processor synthesis. *IEEE workshop on rapid system prototyping(RSP), Montreal, Canada*, June 2005.

[17] A Hoffmann Oliver Schliebusch and Achim Nohl. Architecture implementation using machine description language lisa. In *Proceedings of 15th International Conference on VLSI Design (VLSID02)*. Computer Society IEEE, 2002.

[18] CoWare. *CoWare, The ESL design Leader reference manuals*, v2007.1.2 edition, June 2008.

[19] Anantha Chandrakasan Jan Rabaey and Borivoje Nikolic. *Digital Integrated Circuits, A Design Perspective*. Pearson, Prentice Hall, second edition.

[20] J G Mazidi M A Mazidi. *Microcontroller and Embedded Systems*. Pearson Education, fourth edition, 2002.

[21] Coware,inc,http://www.coware.com.

[22] Synopsis. http://www.synopsis.com.

[23] *MAGMA Blast Create and Blast Fusion Manuals*.

[24] T Givargis F Vahid. *Embedded System Design*. Wiley India, 2008.

[25] Cadence. http://www.cadence.com.

[26] Xilinx. http://www.xilinx.com.

Dissemination of Work

1. U K Nanda, K K Mahapatra "Design of an Application Specific Instruction set Processor using LISA", *First International Conference on Advanced Computing and Communication*, pages 206-209, 3-4 May 2010, AJCE, Kanjirapally, Kerala, India.

2. U K Nanda, K K Mahapatra "Design of a FIR filter using Application Description Language ", *National Conference on Wireless Communication and VLSI Design*, 27-28 March 2010, Gwalior, India.
PAPER ACCEPTED:

3. V Dodani Nikhil Kumar, Umakanta Nanda and K K Mahapatra "Optimization of an Application Specific Instruction Set Processor using Application Description Language", *IEEE International Conference on Industrial and Information Systems - 2010 (ICIIS 2010)*, July 29th-Aug 1st 2010, NIT, Suratkal, Karnataka.