Application Specific Instruction Set Processor Design For Embedded Application Using The CoWare Tool

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

Master of Technology

In

VLSI And Embedded Systems Design

By

Lopamudra Samal

210EC2314

Under the Guidance of

Prof. K. K. Mahapatra

and

Prof. A. K. Swain

Department of Electronics and Communication Engineering

National Institute Of Technology, Rourkela

Orissa 769008, India

2012

Application Specific Instruction Set Processor Design For Embedded Application Using The CoWare Tool

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

Master of Technology

In

VLSI And Embedded Systems Design

By

Lopamudra Samal

210EC2314

Under the Guidance of

Prof. K. K. Mahapatra

and

Prof. A. K. Swain

Department of Electronics and Communication Engineering

National Institute Of Technology, Rourkela

Orissa 769008, India

2012

Department of Electronics and Communication Engg
National Institute of Technology Rourkela
Rourkela-769008, Orissa, India.

June 4, 2012

Certificate

This is to certify that the work in the thesis entitled ***Application Specific Instruction Set Processor Design For Embedded Application Using The CoWare Tool*** by ***Lopamudra Samal*** is a record of an original research work carried out under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Master of Technology in Electronics and Communication Engineering. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Place: NIT Rourkela
Date: June 4, 2012

Prof. K. K. Mahapatra.
Department of EC
NIT Rourkela

Prof. A. K. Swain
Department of EC
NIT Rourkela

Acknowledgment

With the making of this thesis, I first express my sincere gratitude to my project guides **Professor Kamala Kanta Mahapatra** and **Professor Ayas Kanta Swain**, **Department of Electronics & Communication**, **NIT**, **Rourkela** for their persistent pioneer at every single step conducing to the successful completion of the thesis. At every stage, I encountered doubts and problems; they dispelled them away with their consistent guidance and advises. They were real source of inspiration for me throughout the making of the project.

I express my obligation to **Dr. Sukadev Meher**, **Head of Department**, **Electronics & Communication**, **NIT**, **Rourkela** for availing me all the facilities to accomplish the project in the department itself.

I am grateful to Jagarnath for his motivation for my thesis. His insightful feedback helped me improve the presentation of the thesis in many ways.

I would like to express my gratitude to all the research scholars whose works were referred to by us during the completion of this project work. A special mention about Mr. Vijay Sharma and Mr. Kanhu Charan Bhuyan for their timely and valuable guidance that helped us to finish the work in the stipulated period of time.

I am grateful to all faculty members and the supporting staff members of the department of Electronics & Communication for their constant support and motivation in the making of the thesis.

I thank my parents, husband & friends for being the eternal source of moral support & motivation and to almighty God for blessing me with the zeal and endeavor to work towards my goal.

Lopamudra Samal

Abstract

An Application Specific Instruction Set Processor (ASIP) is widely used as a System on a Chip(SoC) Component. ASIPs possess an instruction set which is tailored to benefit a specific application. Such specialization allows ASIPs to serve as an intermediate between two dominant processor design styles- ASICs which has high processing abilities at the cost of limited programmability and Programmable solutions such as FPGAs that provide programming flexibility at the cost of less energy efficiency. In this dissertation the goal is to design ASIP, keeping in mind a temperature sensor system. The platform used for processor design is LISA 2.0 description language and processor designing environment from CoWare. Coware processor designer allows processor architecture to be defined at an abstract level and automatic generation of chain of software tools like assembler, linker and simulator for functional verification followed by RTL level description. RTL level description is used to generate synthesized report of the design using RTL compiler and finally the layout is created using Cadence encounter.

***Keywords*:** 32-bit embedded processor, Language for Instruction Set Architecture(LISA), Coware, Register Transfer Level(RTL), XILINX, Cadence RTL compiler, Cadence Encounter,

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Motivation

Sensor Network

Types of processors

General Purpose Processors

Single Purpose Processors

Application Specific Instruction-set Processor

Literature Review

1.1 Motivation

Sensor network is one of the most interesting research area with a profound effect on technological developments. With the development of micromechanics, microelectronics and integrated circuits, it is possible to integrate sensors, processing units (microcontrollers), memories, wireless communication modules and power supply systems (e.g. batteries) in a single node , these sensor nodes are spread widely in a big area and it is very difficult (or sometimes impossible) to recharge their batteries. Therefore, battery life and power consumption are extremely important for single sensor nodes and whole networks.

A wireless sensor network is a collection of nodes organized into a cooperative network. Each node consists of processing capability (one or more microcontrollers, CPUs or DSP chips), may contain multiple types of memory (program, data and flash memories), have a RF transceiver (usually with a single omni-directional antenna), have a power source (e.g., batteries and solar cells), and accommodate various sensors and actuators. The nodes communicate wirelessly and often self-organize after being deployed in an ad hoc fashion. Systems of 1000s or even 10,000 nodes are anticipated. Such systems can revolutionize the way we live and work. Currently, wireless sensor networks are beginning to be deployed at an accelerated pace. It is not unreasonable to expect that in 10-15 years that the world will be covered with wireless sensor networks with access to them via the Internet. This can be considered as the Internet becoming a physical network. This new technology is exciting with unlimited potential for numerous application areas.

Generally in a sensor node the processor which is used there is like a general purpose processor its area and power is not optimizes according to the application of that sensor node where as an Application Specific Instruction Set Processor (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

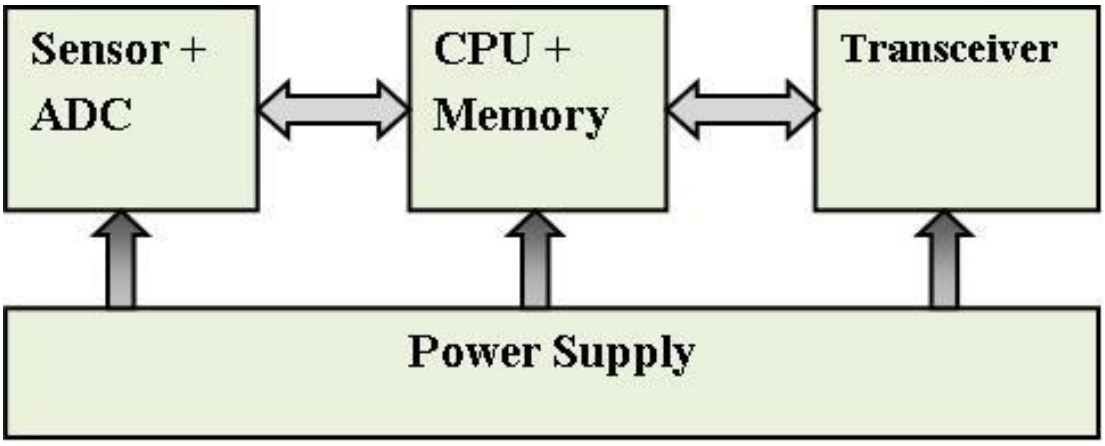

Figure 1.1: Block Diagram of a sensor node

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 Architecture Description Language (ADLs) like LISA, EXPRESSION etc..

1.2 Sensor network

Sensor networks consist of very small nodes that are deployed in some geographical area. These sensors are small, with limited processing and computing resources. These sensor nodes can sense, measure, and gather information from the environment and, based on some local decision process, they can transmit the data to the user. A typical sensor node consists of 4 main parts. Power supply, sensor and analog to digital converter (ADC), processor and storage memory, finally, transceiver to send and receive data. Fig.1 shows the block diagram of a sensor node.

The sensor of the node senses the changes in the environment and gathers the signal then the analog to digital converter converts the analog signal to digital data. Those data are send to the CPU, in the CPU the processor process the incoming data then send those data to the transceiver and also store those data in the memory. The transceiver transmit those data to the user end as well receive data from the user. For all these process a battery power supply is always there. battery life and power

consumption are extremely important for single sensor nodes and whole networks. Minimizing the power consumption of a sensor network is a holistic problem and needs cares from the whole design hierocracy, including low-power design efforts in sensors, wireless transmission modules and communication protocols. In the paper, we mainly focus on the techniques to minimize the power consumption of processing units (sensor network processors). The power consumption of a sensor network processor comes from two sides:

The standby power consumed when the processor is in idle states;

The active power consumed when the processor is executing codes processing samples from sensors, executing communication protocols, etc.

For conventional processors, standby power may be negligible. However, for a sensor network processor spending most (may be 99 percentage) of its time in idle states, standby power is very important. For example, a microprocessor with a 200 A standby current will have a maximum lifetime of 1 year when powered by an AA-size battery even if it never leaves the standby state. In contrast, the lifetime of a microprocessor that burns only few A of leakage current will be completely dominated by battery self-discharging and the active work to be done.

1.3 Types of processors

Processors mainly refer to the architecture of the computation mechanism employed to obtain the desired functionality of a system. The processors may be programmable or non-programmable, depending upon the application. They can be specialized and implement only a single function, or be general purpose and implement a wide range of functions. The main feature which governs the use of different types of processors for different applications are the design metrics. Some of the most commonly considered design metrics are NRE cost, flexibility, performance, power consumption, size, time-to-prototype and so on.

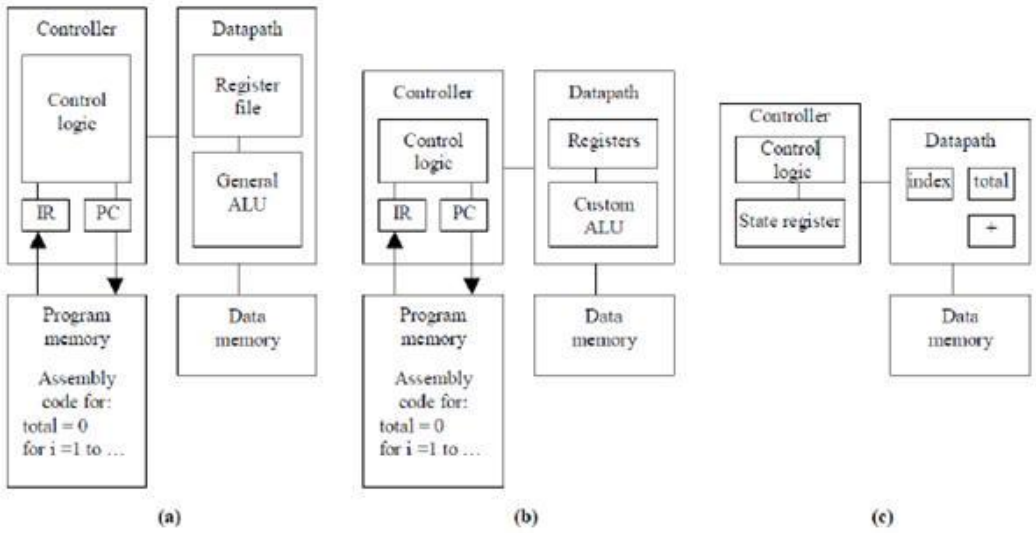

Figure 1.2: (a)General Purpose Processor,(b)Application Specific Instruction-set Processor,(c)Single Purpose Processor

1.3.1 General Purpose Processors

A General Purpose Processor (GPP) or microprocessor as it is generally called is a programmable device that has the aim of implementing a large number of applications such that the number of devices sold is maximized. The main features of this processor are that, the program memory is not built-in to the circuit, since it has to run different programs at different times and it has a general data path, with a large register file and one or more general purpose Arithmetic and Logical Units (ALUs).

It has good time-to-market and NRE costs since only the program has to be changed for the different applications without any change in hardware. Flexibility is also high due to the same reason. However, the performance is poor for certain applications and the size and power consumed are also high, because of the large hardware size.

1.3.2 Single Purpose Processors

A Single Purpose Processor (SPP) is a processor or a digital circuit which is designed to execute only a single program. For example, the circuit used for image processing in a digital camera is a SPP which has the single function of processing the input image and storing it for subsequent retrievation. It has almost the opposite features of a GPP, since it has a small register file, a dedicated data path with an ALU performing only a limited number of operations and no provision of altering the program memory.

It has several design benefits, since the performance may be fast, power consumption less and also small size. However, it has the disadvantages of having very high NRE costs, low flexibility and longer design time.

1.3.3 Application Specific Instruction-set Processor

An Application Specific Instruction-set Processor (ASIP) serves as a compromise between a GPP and a SPP. It is a programmable processor which has an optimized data path for implementing only a particular class of operations. Several special functionalities may be added while unnecessary ones eliminated. Microcontrollers and Digital Signal Processors (DSPs) are some of the most common types of ASIPs in use. They have a program memory that can be changed for different applications and limited register-memory file depending upon the type of application and memory use.

It has the advantages of having flexibility, at the same time achieving good performance, low power consumption and optimum size. The drawback is that it requires large NRE cost to manufacture, especially to design the compiler. Certain design environments such as CoWare offer the benefit of automatically generating the compiler which has greatly reduced the cost and time of manufacturing the device.

1.4 Literature Review

Wireless sensor networks (WSN) consist of a large number of wireless sensor nodes deployed randomly in the area. The nodes collect the environmental data and send them through the network towards the sink node. The nodes are constructed to be

operational for a long time without replacing the batteries. Therefore, one of the primary goals when designing sensor nodes is to reduce the power consumption. To minimize the power of a sensor node, researchers tend to combine novel architecture solutions with advanced power saving techniques. In [1], authors proposed an application specific architecture that integrates an event processor that assists main microcontroller executing required system tasks. The presented approach promises good power optimization, but no real world implementation results have been presented. The approach in [2] utilizes hardware acceleration and optimized radio in a highly integrated single-chip solution. It applies an 8- bit data, 16-bit instruction CPU with reported size of 0.381 mm2 in a 0.25um process. The reduction of power in sensor node radio is also investigated. One novel approach to low power radio is presented in Pico Radio project [3]. Some researchers propose use of wake-up radio in order to reduce the radio power. Wakeup radio serves as a low-power switch to the node transceiver [4]. The implementation of advanced power-saving techniques such as dynamic voltage scaling [5] and power gating [6] promises to deliver additional reduction of node power consumption. The energy harvesting is also considered as a feasible solution to extend the battery life [7]. Using an asynchronous processor in the design of a sensor node is proposed in [8]. Another solution is an asynchronous architecture of a sensor node presented in [9]. The later solution includes additionally an energy harvesting circuit. However, no implementation results for those solutions have been presented. Fully asynchronous architecture is difficult to implement and it requires all peripherals to have a dedicated asynchronous interface.

The concept of instruction set oriented ASIPs is well known in the technical literature. In a concise overview of ASIP design issues [10] 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 [11] 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 [12] 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 [13] 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 [14]. In [15] the RECORD compiler is presented which uses a structural RTL model of a DSP as a starting point of the compiler generation. In [16] authors present a highly efficient processor design methodology based on the LISA 2.0 language. Typically the architecture design phase is dominated by an iterative processor model refinement based on the results of hardware and software simulation and profiling.

Chapter 2

ASIP Design Methogology

Implementation of Application Specific Instruction Set Processor

ASIP Design Flow

Architecture Exploration

Field of Application

2.1 Implementation of Application Specific Instruction Set Processor (ASIP)

An ASIP has an instruction set optimized for a single application or a class of applications. On one hand, an 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.

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

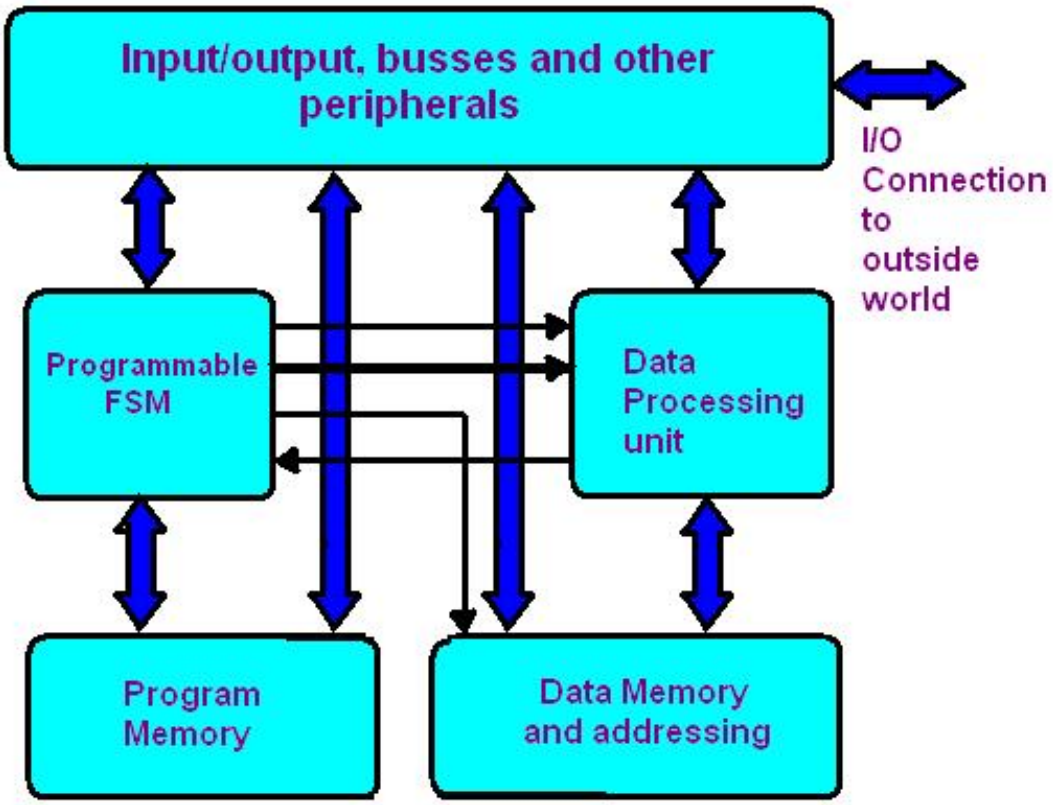

Figure 2.1: DSP Processor Architecture

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

Processor design is a complicated process. Without an advanced design flow, a processor cannot be designed in time and the quality of the design will not be high. The design flow is therefore essential for complicated systems such as ASIP. The ASIP design flow is introduced briefly here. The ASIP design flow is divided into three parts: architecture design, design of programming tools, and firmware design, as depicted in Figure 2.2.The first and most important step in the design of a processor is the instruction set design. This design step is complicated, and no one can really claim that a certain instruction set is the best. 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. In Figure 2.2, a simplified design flow is described including the basic flow for the design of an instruction set architecture. The starting point of the design of anASIP is the application analysis.Application coverage should be specified first and then translated to functional (algorithm) coverage. Application coverage is the process of reading and understanding specifications and standards of the relevant applications. Functional coverage of an ASIP is decided based on both the current standard specifications and carefully collected knowledge features for future usage. Performance and cost should also be specified as design constraints.

After the functional coverage is determined, the partitioning of hardware and software should be decided through profiling of the source code. Hardware/software partitioning for an ASIP is to meet the performance constraint by defining what functions should be accelerated by application-specific instructions and what functions should be implemented as software routines using conventional instructions. This is an important design step of an instruction set, which is called the 10 percent to 90 percent code locality. The locality rule means that 10 percent of the instructions run 90 percent of the time and 90 percent of the instructions appear only 10 percent of the time during execution. In other words, ASIP design is to find the best instruction set architecture optimized for the 10 percent most frequently used instructions and to select those among the 90 percent of the not often used instructions in order to

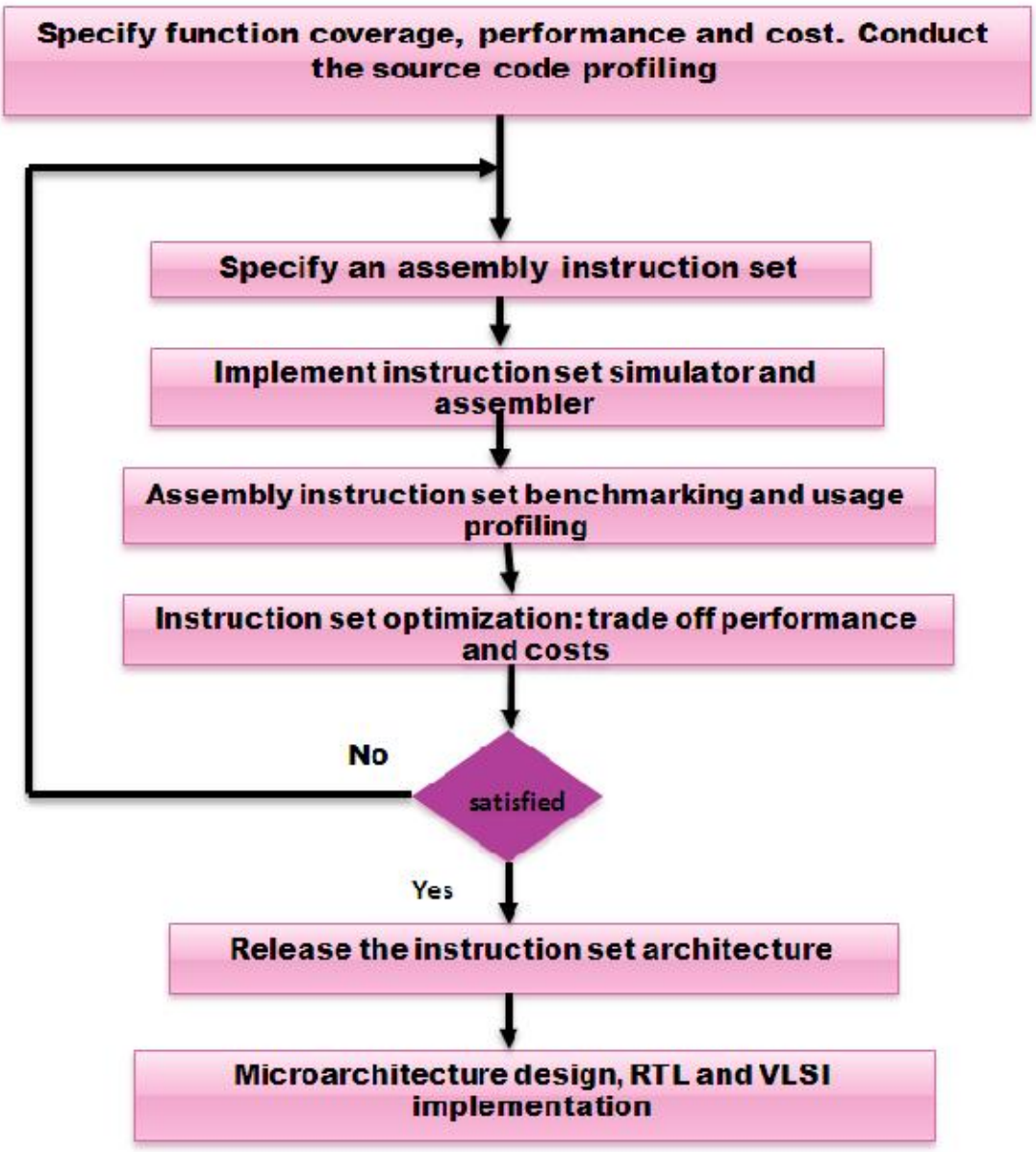

Figure 2.2: The ASIP Design flow

guarantee the functional coverage.

During the process of hardware and software partitioning, the instruction set of the ASIP is gradually specified.The next design step is to implement the instruction set, which includes instruction coding, design of the instruction set simulator, and benchmarking.The coding of the instruction set includes the design of the assembly syntax and the design of the binary machine codes. The instruction set simulator must

be implemented after the instruction set has been coded. Finally, the instruction set must be evaluated by benchmarking. The performance of the instruction set and the usage of each instruction will be exposed as inputs for further optimization.

The ASIP architecture can be specified when the assembly instruction set is released. The microarchitecture design is a refinement of the architecture design including fine-grained function allocation and hardware pipeline scheduling, specifying hardware modules,and interconnections between modules.

TheASIP 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.

2.3 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.

2.3.1 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 since the generated HDL model does not have any predefined components. Then the generated HDL model can be compared

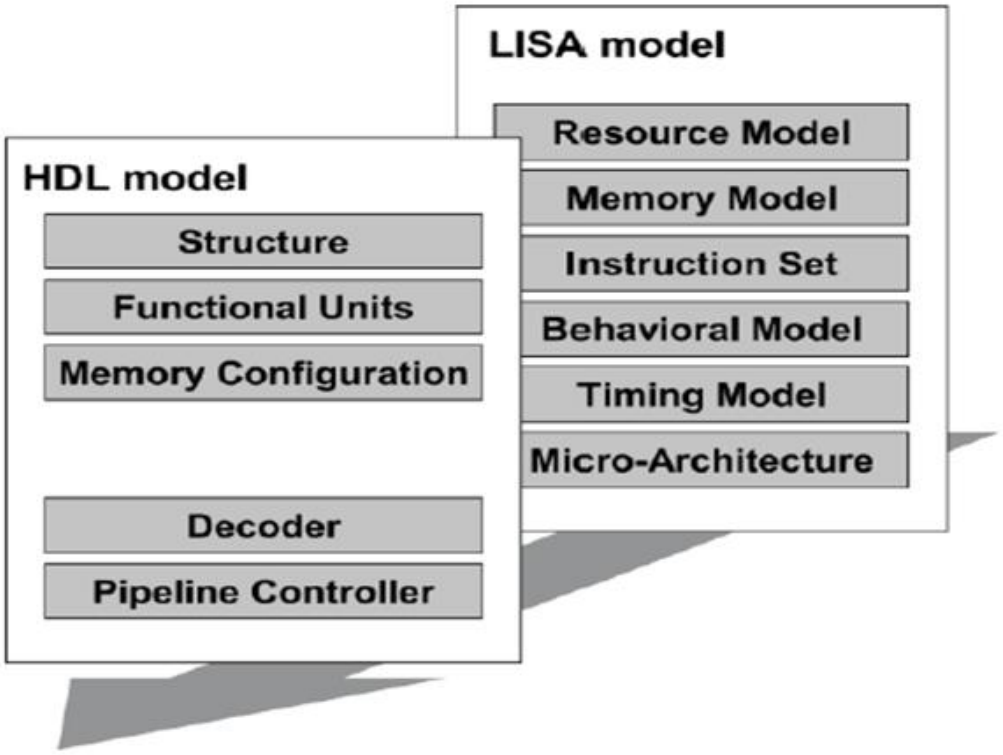

Figure 2.3: Comparision of HDL And LISA model

to the LISA model components as shown in the figure 2.4.

- 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.

2.3.2 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.

2.3.3 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.

2.4 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 ar-

chitecture 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 as shown in the figure 2.5.

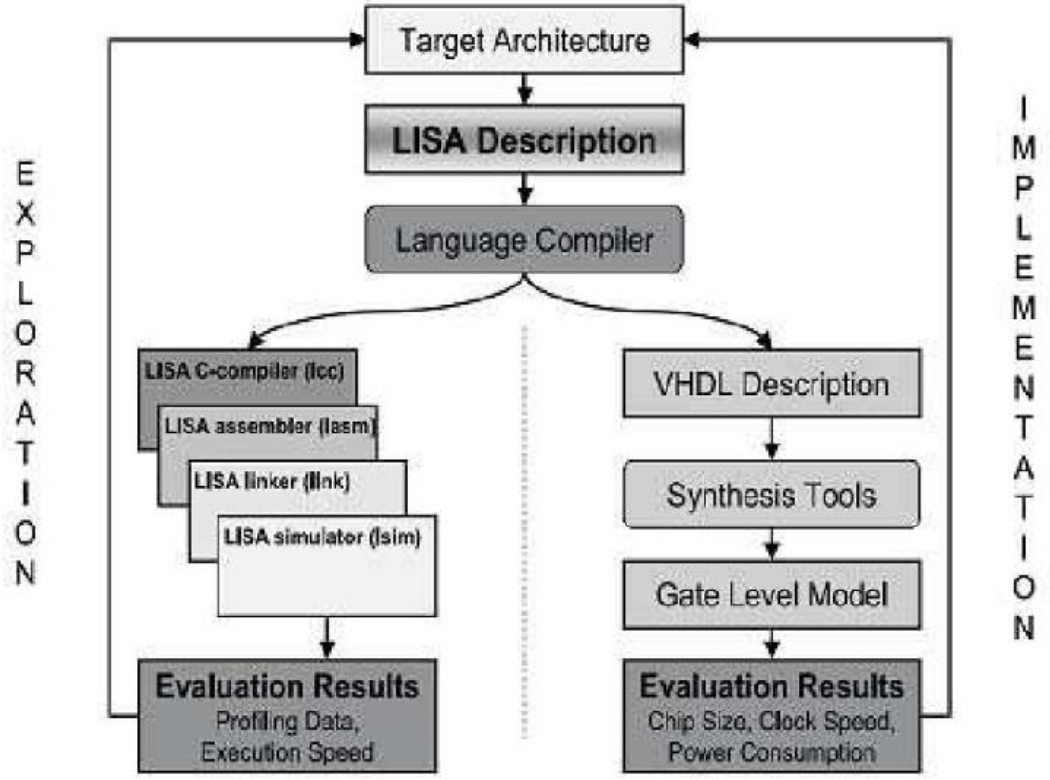

Figure 2.4: Exploration and Implementation

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

CoWare Design Flow

CoWare Design Flow

Designing Application-Specific Processors with ADLs

The LISA processor models

CoWare Processor Designer

The Instruction Set Designer

CoWare Processor Debugger

Major Benefits

3.1 CoWare Design Flow

Figure 3.1 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.

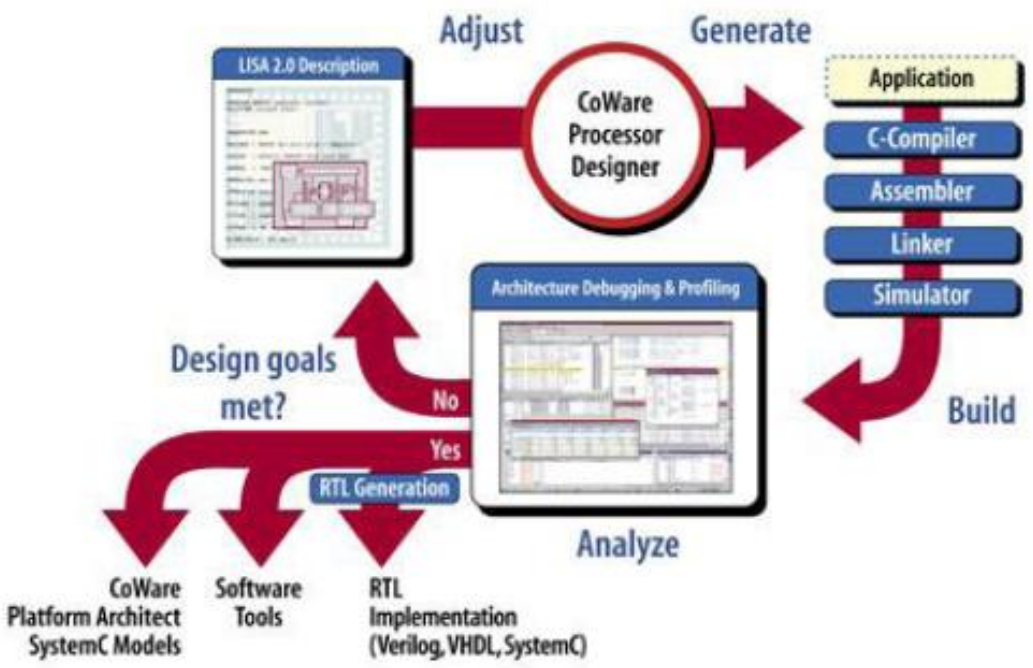

Figure 3.1: CoWare design flow block diagram

Its advanced and flexible features are

- 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 and physical design tool like Cadence Encounter.
- Software development tool generation like assembler, linker, debugger, C- compiler.
- Integrated profiling helps to optimize instructions for the target architecture.
- Enables the design team to develop flexible and reusable ASIPs rapidly.

3.2 Designing Application-Specific Processors with ADLs

The typical design flow of a microprocessor and the associated tools was introduced in Fig.3.3. In the classical approach, we start with an architecture description and then develop the instruction set and architecture. We then write the HDL code for the processor and write, based on this developed architecture, the development tools (e.g., the instruction set simulator (ISS), the C compiler, the assembler, et cetera). While this hand-coded HDL may allow us to obtain an extremely small core-size by taking advantage of the underlying logic blocks (e.g., PicoBlaze by Xilinx used the 32x1 LUT to implement the processor registers and save many resources), the disadvantage is that any changes in hardware also need to be coded in all the development tools. This is considered a major source of cost and inefficiency in embedded processor design when using HDL. Architecture Description Languages (ADLs) allow a microprocessor to be modeled in all levels of the design [5,6] with just one consistent description. With ADLs the microprocessor design process becomes efficient and reliable. Let us have a brief look at available ADLs and their features. Early ADLs were either structure-oriented (MIMOLA, UDL/I) or behavior-orientated (Valen-C or ISDL). Later, mixed ADLs such as nML, LISA, HMDES, ASIP meister, Flexware, TDL, and EXPRESSION adopted an integrated approach: the language captures both the structural as well as the behavioral design of the embedded processor, often called an Application-Specific Integrated Processor (ASIP).

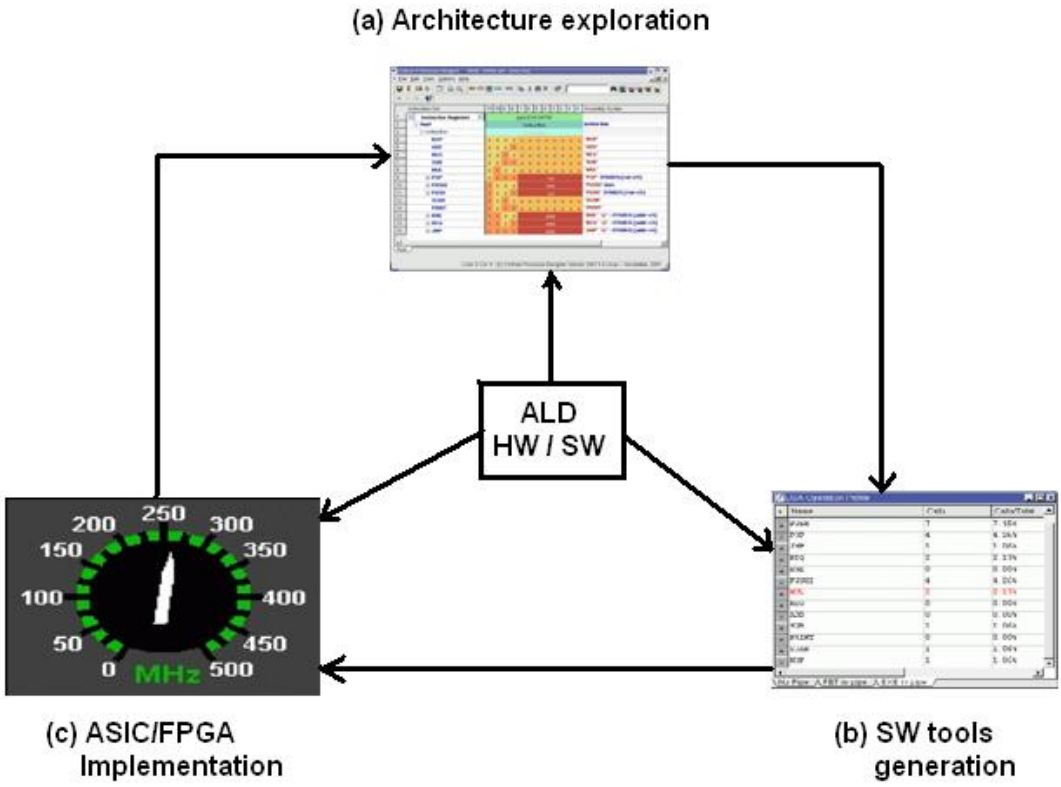

Figure 3.2: Processor Design using LISA. (a) Architecture explorations: ISA, caches, co-processor. (b) SW tools: C-compiler, assembler, linker, instruction set simulator, profiler. (c) Implementation: size, power, speed.

Several of these tools have been developed in academia, and some have become commercial tools. Currently, three professional packages are available that have a full tool set including VHDL/Verilog code generation: nML, ASIP Meister, and LISA. The nML comes with a Chess/Checkers retargetable C compiler, an RTL synthesis generator GO, and RISK a test-program generator. These commercial tools have been used in a wide variety of products, such as portable audio and hearing instruments by Cool-Flux, Wireline modems ADSL2+ by STMicroelectronics, wireless modems HS-DPA by Nokia, and TI video accelerators and network processors [5]. ASIP Meister is a GUI-based processor system and has been used by 180 academic institutions in 37 countries since 2002. Since 2006 ASIP Solution Inc. has taken over the maintenance and further development of ASIP Meister. The GUI-based platform is somehow

more restrictive that nML or LISA, but allows a much improved development time. MIPS processor for instance could be developed in 8 h, and DLX in only 3.5 h [5]. The Language for Instruction Set Architecture (LISA) [6,17] allows to specify first an instruction- or cycle-accurate lP using a few LISA operations, then to create architecture exploration using a tool generator and profiler, and to finally determine speed/size/power parameters via automatically synthesized VHDL or Verilog code. The LISA tools developed at ISS RWTH Aachen are now the Processor Designer product of Coware/Synopsys Inc. (CA, US)[17,6]. The LISA tool environment is shown in Fig.3.3. The design flow (see Fig.3.2) is similar to the classic approach, the only difference is that one LISA 2.0 based processor description is used to specify the behavior of the microprocessor as well as the generated development tools. We decided to use the LISA tool sets since they are available at University pricing, provide HDL production quality code, and offer many different ASIP starting point models. The authors have developed tutorials for a iterative LISA-based processor refinement which are used at FSU for embedded lP design course and have been posted online recently [18,19]. Although LISA is used in this study similar hardware acceleration results are expected with other tools such as nML,ASIP Meister, or EXPRESSION.

3.3 The LISA processor models

The LISA processor designer (PD) design flow was used to develop the embedded processors in this study. The LISA language supports a profile-based and stepwise refinement of processor models down to cycle-accurate and even VHDL or Verilog RTL synthesis models for fast custom VLSI implementation. Microprocessors from simple RISC to highly complex VLIW processors have been described and successfully implemented using the Processor Designer for FPGAs and cell-based ASICs. There are more than 40 LISA models in both industry and academia from different architectural categories (RISC, PDSP, and ASIP) available [2023]. These include different ARM and MIPS models: PDSP from TI and StarCore, as well as ASIPs from Infineon (ICORE), STMicroelectronis, etc. LISA has been adopted by several leading ASIC houses and has over 40 members in the CoWare PD University program.

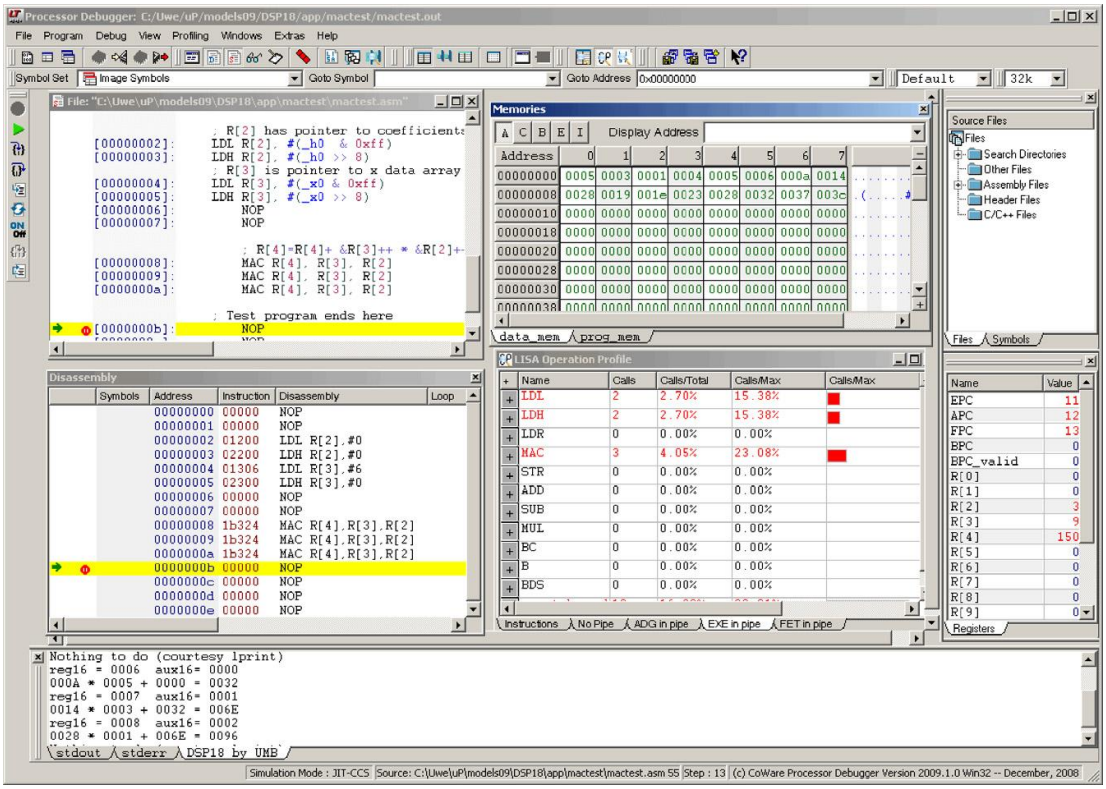

Figure 3.3: LISA Development tools. Disassembler (Left). Memory monitor and Pipeline profiles (Center). Source files and register window (Right)

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 3.4 shows the Processor Designer Main Window

The key to Processor Designers 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,

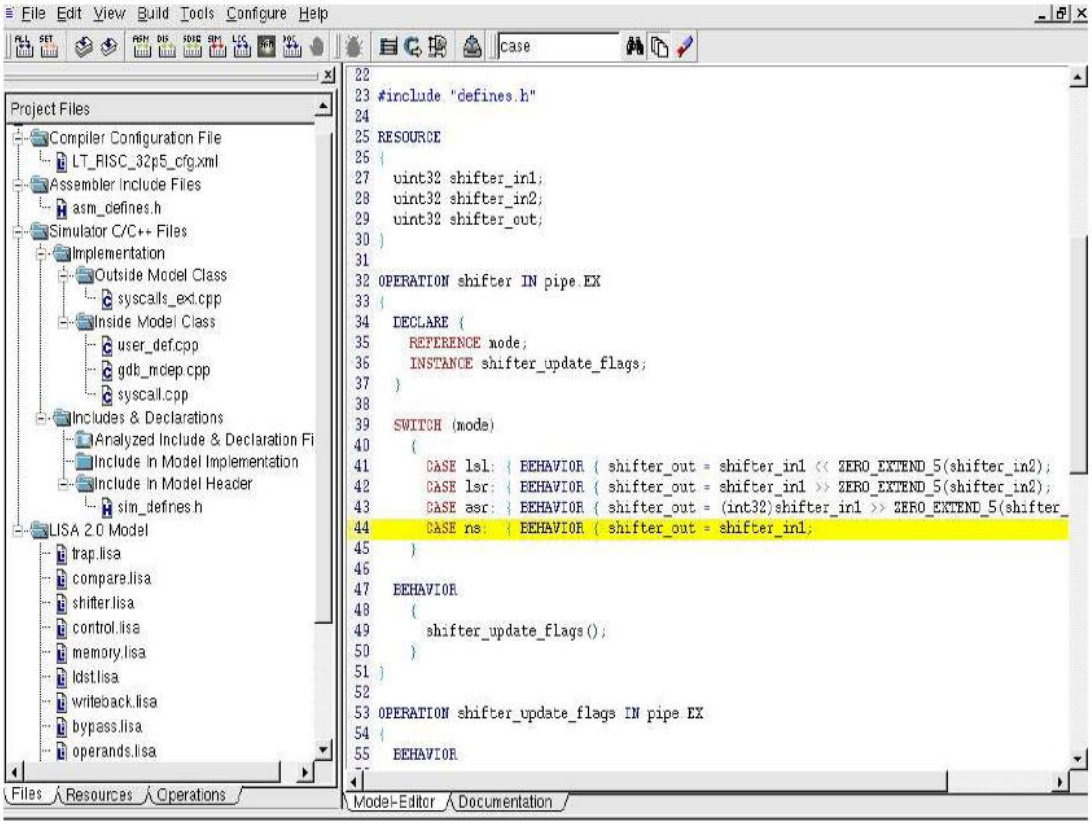

Figure 3.4: CoWare Processor Designer Main Window

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 processors 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 Designers 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 graphical user interface (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. The Instruction-Set

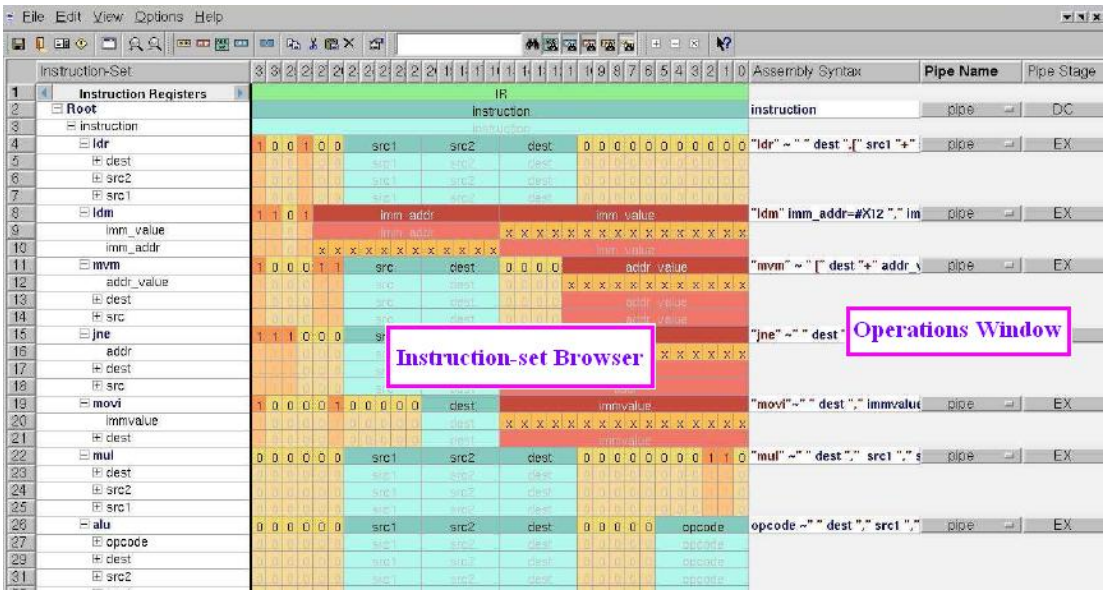

Figure 3.5: Instruction Set Designer Window

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 processors resources and the hardware behavior is still manually written as LISA code.

3.6 Simulation in CoWare Processor debugger

The simulation in the CoWare processor debugger can be described by following three step process Figure 3.6 represents that . The assembler and the linker are the standard

software tools generated by the LISATek product family. An assembly file is written corresponding to the instruction set defined for the processor.

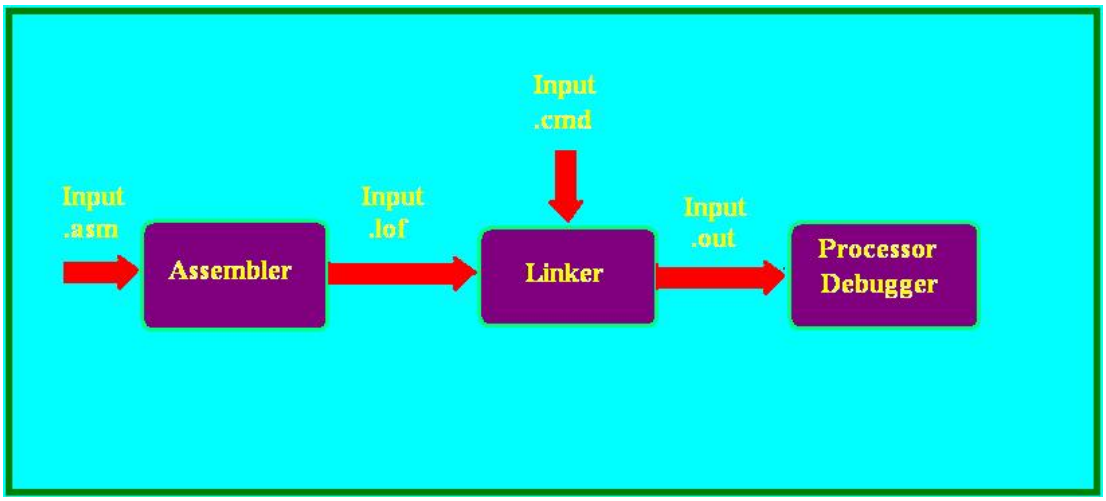

Figure 3.6: Simulation in CoWare Processor debugger

The assembler takes this file as input and outputs a ".lof" file, among other files. The linker takes in the ".lof" file as input and together with ".cmd" file having specified format, it generates the executable ".out", which can be used to run on the virtual QSIP architecture.

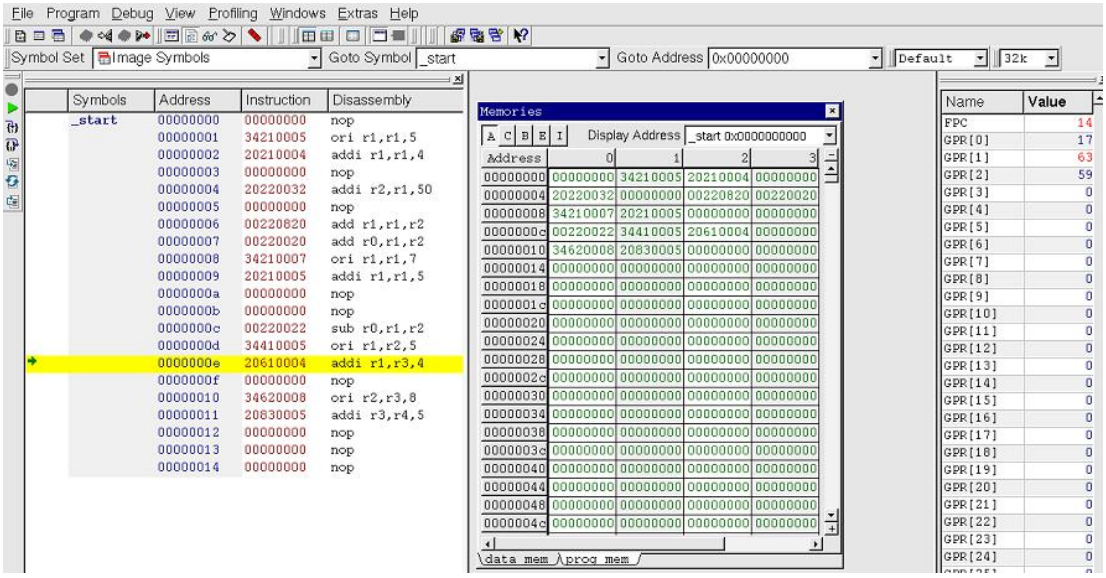

Figure 3.7: Processor Debugger Window

3.7 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 3.7 shows the Processor Debugger Window for a loaded application.

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.

3.8 Major Benefits

- Design teams can rapidly develop flexible and re-usable application specific embedded processors section which include essential SoC functionality, through:
  - Rapid architecture design with LISA 2.0 by any designer conversant with C/C++
  - Automatic generation of software development tools and simulator
  - 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

LISATEK Design Methodology

LISA Description

LISA Description

Hardware Modelling (RESOURCE Section)

Software Modelling

4.1 LISATEK Design Methodology

The idea of the LISATek design flow is to define a programmable platform tailored to a specific application domain. This puts a heavy burden on the ASIP designer to compose a capable platform from a huge design space for the target application. The goal of the LISA 2.0 based processor design flow is to guide the designer from the algorithmic specification of the application down to the implementation of the micro-architecture. In every phase of the processor design the designer maintains an abstract model of the target architecture written in the LISA 2.0 language. The language LISA 2.0 is aiming at the formalized description of programmable architectures, their peripherals and interfaces. LISA 2.0 is not a completely new language it is an extension to C. The hardware behavior as well as processor resources like registers are modeled in pure C, whereas LISA 2.0 adds on top of the Clanguage capabilities to describe an instruction-set with its binary encoding and assembly syntax. Also, LISA 2.0 allows to express timing in processors. An example is a pipelined architecture where instruction execution is spread over multiple cycles. LISA 2.0 is very easy to learn so that a couple of days is sufficient to become familiar with this language.

From such a model, a working software development tool set supporting the evaluation tasks for the current development phase can be generated automatically. The first stage of the design process is concerned with the examination of the application to be mapped onto the processor architecture. Critical portions of the application need to be identified that will later require parallelization and specific hardware acceleration. For this reason the design space exploration starts with the definition of the processors instructionset.

The two main ASIP development phases of the LISA 2.0 based design flow are shown in figure 4.1. On the left hand side the architecture exploration phase with the software development tool generation is visualized, on the right hand side the implementation phase which starts with the automatic creation of an RTL model of the ASIP.

As a starting point for model creation CoWare LISATek provides a library of sample models which contains processors that are already tailored to specific applications.

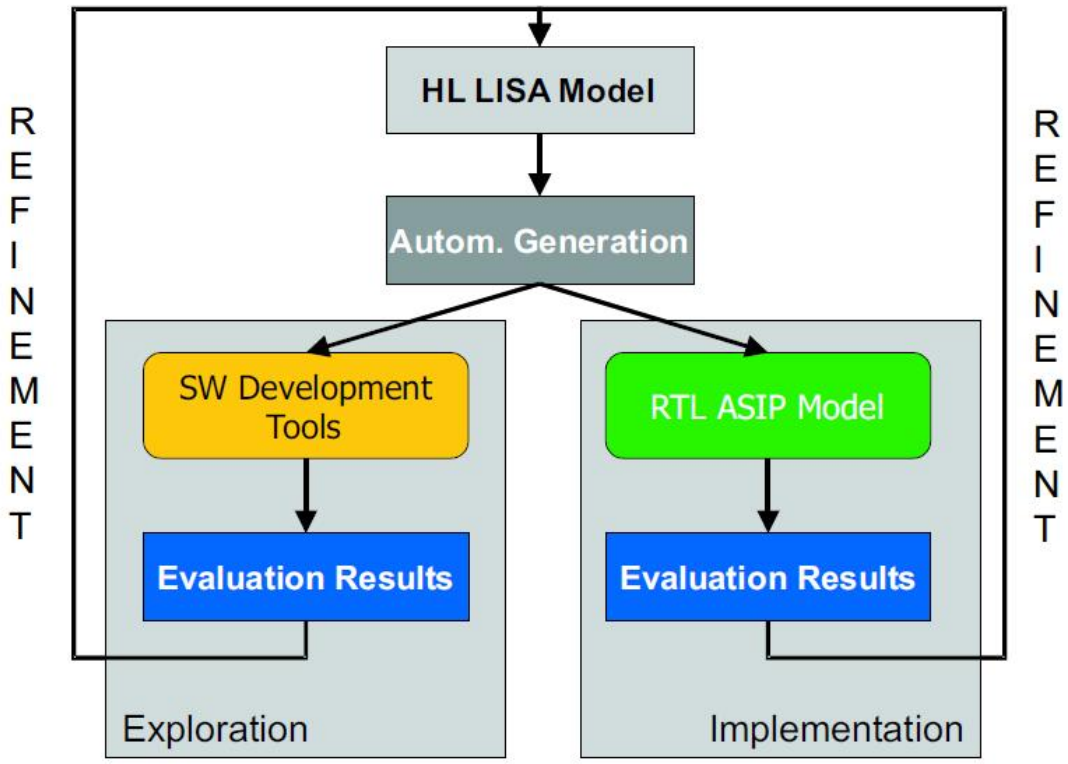

Figure 4.1: CoWare design flow block diagram

These processors efficiently implement algorithms like turbo decoding and the FFT. Also, there are sample models for different architecture categories available which cover DSPs, micro controllers with specific features like SIMD (single instruction multiple data) which is popular in the multimedia domain as well as the increasingly popular VLIW architectures which comprises massively parallel functionality. It is important to distinguish these sample models from configurable template models where only some parameters may be changed. Taking such models as a basis has the major advantage to directly have compiler support for the architecture due to the existence of an instruction-set. This makes the C- and instruction profiling of the application possible from the very beginning of the architecture development. The simulator which is derived from this model constitutes a virtual machine executing the application directly. The profiling capabilities of the simulator are used to generate execution statistics of the application code. Once the profiling information is gathered, critical portions which require parallelization are identified. Based on the

profiling results the instruction-set is adapted until the application profiling meets the given criteria.

At this point in the design phase the designer has to consider different aspects of the micro architecture. The major exploration and optimization point is the pipeline. The designer has to decide how many pipeline stages are required with respect to control flow instructions and the efficient implementation of hardware loops and interrupts. For the performance of the architecture it is important to avoid data hazards during program execution. For this reason data bypassing may be implemented when specifying the pipeline of the processor. This mechanism serves already calculated results to pipeline stages prior in the pipeline. The intention of this mechanism is to bypass data storage in registers or memories. Having an optimal pipeline in an ASIP requires a memory subsystem to support this pipeline with memory data fast enough because otherwise the pipeline has to be stalled working on the application while waiting for the memory data. The memory hierarchy directly contributes to the performance of the memory subsystem, thus, the developer has to consider it carefully. Caches with varying parameters are widely used to enhance the performance of the memory subsystem. Here the cache parameters i.e. the cache size and the cache read and write policies must be determined with respect to the target application. Additionally, the designer has to evaluate the role of a memory managing unit (MMU) and has to check the performance of the utilized bus to ensure the optimal configuration of the utilized memories. A very powerful capability of the LISA 2.0 language besides its ability to model arbitrarily complex processors is a special template library with memory modules which can be easily parameterized from within the LISA 2.0 model. Using these library elements, caches, MMUs and buses can be easily modeled.

When assigning different parts of the instruction execution to the already defined pipeline stages the developer must care about resource sharing and the length of the critical path in the emerging architecture. This is important for an efficient hardware implementation of the ASIP. Here the required chip area and the gate count are commonly used constraints which directly refer to the power consumption of the resulting hardware. The critical path is important, since its length limits the clock

speed which is another important criteria when designing an ASIP.

When the architecture meets the design criteria and efficiently implements the application, in a final architecture development step hardware implementation is done. Synthesizable HDL RTL code (currently VHDL and Verilog) for the control and data path of the processor can be derived from the abstract processor model automatically. This includes the entire hardware model structure such as the pipeline, pipeline controller including complex interlocking mechanisms, forwarding, etc. to steer the architectures behavior and an implementation of the data path which is directly derived from the behavioral specification in the LISA 2.0 model. Having the RTL generation capabilities in the processor exploration loop allows to easily explore the trade off area vs. timing (clock speed) vs. flexibility. Based on the simulation and synthesis results of the hardware, the abstract LISA 2.0 model might be modified to meet power, area and frequency constraints. Due to the fact that RTL and ISS simulator are derived from a sole processor model, they are automatically consistent. It is obvious that deriving both software tools and hardware implementation model from the same architecture specification in LISA 2.0 has significant advantages. Only one model needs to be maintained, even if changes to the micro-architecture or the behavior in the hardware model must be realized.

Once the processor design is finished, a set of production quality software development tools is generated from the LISA 2.0 model. These software tools (C-compiler, assembler, linker) can compete well in terms of functionality and feature richness with state-of-theart tools from Greenhills, ARM, etc. The generated C-Compiler is an optimizing compiler which is capable of generating code which is close to handwritten assembly code. In addition to these generated software development tools a macro assembler and an archiver are provided for the LISATek product family.

In order to be able to integrate into system simulation environments (SoC) to gather realistic stimuli from the system, LISATek generates processor simulators which couple directly with the following tools: CoWare ConvergenSC, Cadence Incisive, Mentor Graphics Seamless CVE, OSCI reference simulator as well as with any Cbased environments. The generated simulators automatically interface with popular

busses like AMBA AHB and can be extended to work with proprietary buses easily. The generated instructionset simulators (ISS) support system simulation on different levels of abstraction from cycle accurate to untimed system simulations. Utilizing the patent pending Just-In-Time-Cache-Compiled (JITCC) simulation technology LISATek simulators run at a very high speed.

Moreover, LISATek tools support multi-processor debugging in such a system. Here, the designer can debug one ore more processors with a single graphical debugger. The verification of the ISS vs. the RTL model can be performed by using the IBM Genesys [17] test-generation tool. Genesys is a test-generator which has been exclusively developed for validating processors. It works based on a test plan and generates test programs automatically which are run directly on an ISS. Finally the test-program together with the expected result values in the processor are given. A major benefit of the LISATek approach is the fact that the designer has neither to be a software nor a hardware expert. So a single person can cover a broad spectrum of development tasks which cannot be covered by the traditional ASIP design approach.

4.2 LISA Description

A LISA 2.0 processor description mainly consists of hardware and software model of the architecture. The hardware model comprises the definition of processor resources like registers, memories, pipeline and buses for accessing memories. The software model comprises the definition of the processor instruction set, their binary instruction coding, assembly syntax and response of the processor hardware to instruction.

The acronym of LISA 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.

The idea of the LISATek design flow is to define a programmable platform tailored to a specific application domain. This puts a heavy burden on the ASIP designer to compose a capable platform from a huge design space for the target application. The goal of the LISA 2.0 based processor design flow is to guide the designer from the algorithmic specification of the application down to the implementation of the micro-architecture. In every phase of the processor design the designer maintains an abstract model of the target architecture written in the LISA 2.0 language. The language LISA 2.0 is aiming at the formalized description of programmable architectures, their peripherals and interfaces. LISA 2.0 is not a completely new language it is an extension to C. The hardware behavior as well as processor resources like registers are modeled in pure C, whereas LISA 2.0 adds on top of the Clanguage capabilities to describe an instruction-set with its binary encoding and assembly syntax. Also, LISA 2.0 allows to express timing in processors. An example is a pipelined architecture where instruction execution is spread over multiple cycles.

4.3 Hardware Modelling (RESOURCE Section)

The RESOURCE section lists the definitions of all objects which are required to build the memory model and the resource model. These resources represent the current state of the processor. Each time the processor performs one control step - this can be an instruction, cycle, phase - the processor is driven into a new state according to the behavior of the functional units.The Resource declarations follow the style of variable declaration in C and data values of resources are treated like variables in C. However, they can only be declared outside the scope of operations. Consequently, all resources are defined globally and visible for all operations. This resembles the properties of hardware components that are global by their nature. The RESOURCE section allows the declaration of the following types of objects:

Simple Resources:

- Register and Register Flags
- Ideal Memory Arrays
- Signals and flags
- Other resources that are not visible in the architecture

Memory Maps:

- The mapping of memories into processor addresses space.
- The connectivity between memory and bus modules.

The Pipeline structure for instruction and data paths. Pipeline registers storing data on its way from one pipeline stage to the next. And some no ideal memories such as Caches, Buses etc. as a part of the memory subsystem.

4.4 Software Modelling

The software model of the processor consists of processor instructions that are implemented using operations. An operation represents a basic entity in LISA 2.0 model. They represent behavior, structure and instruction set of the programmable architecture. The execution of any instruction ultimately leads to the execution of corresponding instruction. An operation possesses several attributes which are described below.

- The DECLARE element is used to reference elements from other operations and to define used resources.
- The CODING includes the binary coding of the instructions as a sequence of coding fields. The value can be "0", "1", or "X", all equally interchangeable. Alternatively, it can reference the coding field of other LISA operations.
- The SYNTAX describes the assembler syntax that references variables via labels.

- The BEHAVIOR description is included in the processors data path function. Coding is sequential, just as in regular C coding.Resources such as registers, memories, flags, and pins can be accessed, modified, and stored.
- The ACTIVATION allows a LISA operation to activate another operation or a group of operations typically for the next pipeline stage and also includes the coding root tree specification with CODING AT.
- The DOCUMENTATION allows specification of a description that will be included in the instruction set manual generated automatically by the PD.

Each tool derived from LISA models needs a subset of these sections to perform its particular task. For example, the Assembler utilizes the information contained in the CODING and SYNTAX sections, since this tool maps the instruction syntax onto a binary coding word, while the De-assembler works the other way round. For this reason, it also makes use of the CODINGand SYNTAX sections. The Simulator depends on the information of the BEHAVIOR sections, while the operation scheduler relies on the ACTIVATION section.

Chapter 5

Processor Specification

Processor Specification

Processor Specification Description

5.1 Processor Specification

The designed processor is based on a 32-bit architecture with the following specifications:

Instruction Length	32 bits
Opcode Length	8 bits
GPR	32 bits
Program Counter	32 bits
Pipeline Stages	3
Program Memory	32 bits
Program Memory Range	0x0000-0x0FFF
Data Memory	19 bits
Data Memory Range	0x0000-0x0FFF

Figure 5.1: Processor Specification

5.2 Processor Specification Description

5.2.1 Instruction Length

The total number of bits used to represent an instruction including the opcode and operands. The current instruction being executed is stored in the instruction register (IR).

5.2.2 Opcode Length

The number of bits reserved for the opcode segment of the instruction. Since the opcode length is 8-bits, the total number of instructions that can be defined is 28 = 256.

5.2.3 GPR (General Purpose Register)

Eight GPRs each 32 bits in length have been defined to store any transient data required by the program.

5.2.4 Program Counter (PC)

The Program Counter keeps track of the next instruction to be executed. It stores the address of the program memory location corresponding to the next instruction. The IR in turn is loaded with the instruction from the memory location pointed to by the PC at the start of the next clock cycle. Since the PC is 32 bits in length, it can point to a program memory with 232 locations.

5.2.5 Program Memory

The program memory stores a sequence of instructions comprising the program that implements the application functionality. The address space of the program memory for the designed architecture is 232 as the PC length is 32-bits. However, due to resource constraints of the final implementation platform, the range is kept limited to 0x1000 memory locations (0x0000-0x0FFF). Also, each memory location is 32-bits long in accordance with the IR length which is also 32-bits long.

5.2.6 Data Memory

The data memory stores the input data on which the program operates as well as the final results computed from the operation. Each memory location is 19-bits in length i.e., same as that of a GPR and hence data can be transferred from memory to GPR in just one clock cycle. The defined range is 0x0000-0x8FFF (0x9000 locations).

5.2.7 Pipeline

A computer program is, in essence, a stream of instructions executed by a processor. These instructions can be re-ordered and combined into groups which are then executed in parallel without changing the functionality of the program. This is known as

instruction-level parallelism and implemented through pipelining. The designed architecture implements a 3-stage pipeline. The three stages comprise of the following stages:

- FE (Fetch and Decode): Fetching the instruction from the IR and decoding its functionality.
- DC (Address Generation): Generating the address of the data memory locations that store the required operands. Also, the branch address in the program memory as specified by the flow control (branch) instructions is generated in this stage.
- EX (Execution): Actual execution of the fetched instruction occurs in this stage.

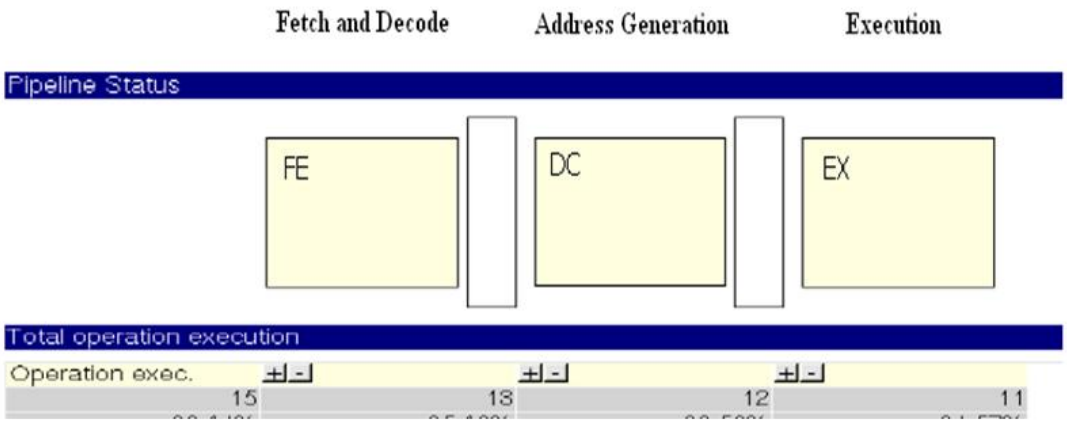

Figure 5.2: Pipeline Stages

FE and EX stages occur for every instruction. AG stages occur only for those instructions that require data memory access or involve branching in program memory (Flow control instructions).

Chapter 6

The generated HDL model structure

The generated HDL model structure

HDL Modules Overview

Comparison of the HDL codes generated

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.6.2 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 esources, memory resources and the pipeline.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.

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.

6.2 HDL Modules Overview

This section provides a closer look at the generated HDL code structure. Consider the following LISA model with the RESOURCE section.

The above code sample shows a three-stage pipeline, some global registers, a program and data memory. It also includes some UNIT declarations to group the op-

```
RESOURCE {
    REGISTER TClocked<int32> GPR[0..31];
    REGISTER TClocked<int32> BPC;
    ...

    MEMORY  uint32 prog_mem { ... };
    RAM     uint32 data_mem { ... };

    PIPELINE pipe = { FE; DC; EX };
    PIPELINE_REGISTER IN pipe {
        uint32 pc;
        uint32 insn;
        uint32 operand1;
        uint32 operand2;
        ...
}


    UNIT U_FETCH IN pipe.FE { ... };
    UNIT U_CONTORL IN pipe.DC { ... };
    UNIT U_ARITH_DC IN pipe.DC { ... };
    UNIT U_ARITH_EX IN pipe.EX { ... };
}
```

Figure 6.1: LISA Code

erations of the model into functional units. This leads to the following HDL code structure showen in Figure 6.1.

All generated files for the entities VHDL respective modules Verilog have the suffix-gen, which allows you to store unique hand-written HDL modules in the same source directory without the danger of accidentally overwriting them. In general, the entity/module name is the same as the file name, without the suffix-gen. The following

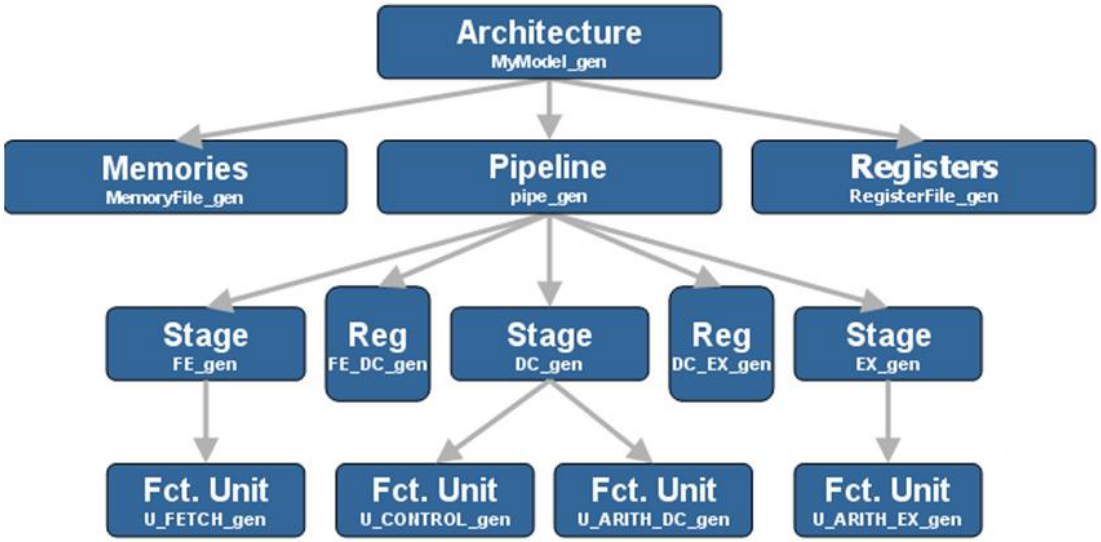

Figure 6.2: HDL Code Structure

modules can be identified in the above figure:

6.2.1 Architecture (My Model-gen):

This is the top-level module. The module name is the name of the LISA model, defined as Architecture Name in the Processor Compiler Setup dialog box of the Processor Designer. It instantiates three submodules: MemoryFile, pipe (named after the pipeline name) and RegisterFile. Beside this, all global signals in the design (declared as nonclocked registers) are implemented in this module. As this module is the top-level module of the generated HDL code, its interface are the main input and output ports, as there are:

- Clock and reset input Their names, defined in the Processor Generator configuration are retained throughout the whole design.
- Input and output ports of the LISA model (PINs), if present.
- Input and output ports for a memory, if configured to be external.

6.2.2 Memories (MemoryFile-gen):

This module implements the interface to all memories, in the example the interface to the program memory prog-mem and data memory data-mem. If a memory is configured to be internal, it is also instantiated in this module; if a memory is configured to be external, the interface is routed to the architecture module as input/output ports. Basically, the implementation of the memory interface consists of the following parts:

- The mapping of all accesses (read/write) in the design to the available ports of the specified memory. If for example, there are three write accesses from three different operations to a memory with only one write port (single port memory), it is necessary to implement a multiplexer for the addresses and data. Processor Generator performs some analysis to detect if the three operations are exclusive to each other, meaning that they cannot be executed at the same time. During the generation process, Processor Generator informs about this analysis; if exclusivity cannot be guaranteed a warning is given. In this case, you need to check carefully, if the given memory will meet the actual design structure.

- The mapping of the port accesses to the signals for the specified memory. In case of the use of the generic memory interface, these signals are specified in the Memory Interface Description File (MIDF).

6.2.3 Pipeline (pipe-gen):

This module implements the whole pipeline. The name of the module is taken from the pipeline resource declaration. This declaration is shown as pipe. It instantiates modules for each pipeline stage (the stages FE, DC and EX in the example) and for each pipeline register between the stages (FE-DC for the registers between the FE and DC stage, and DC-EX for the registers between the DC and EX stage). Thus, the names directly correspond to the names specified in the RESOURCE section. If stalls (stall()) and flushes (flush()) are used in the LISA model, then a pipeline controller is also implemented.

6.2.4 Registers (RegisterFile-gen):

This module implements all global clocked register resources. It is either explicitly marked as TClocked in the RESOURCE declaration, or implicitly set to TClocked by configured to Automatic assignment of cycle-accurate behavior to processor resources in the Processor Compiler Setup dialog box of the Processor Designer.

6.2.5 Stage XX (FE-gen, DC-gen, EX-gen):

These are the modules that correspond to the pipeline stages defined in the RESOURCE section. A pipeline stage module holds all operations assigned to that stage. These can be explicitly assigned to functional units defined. If not explicitly assigned to a functional unit, all operations with a nonempty BEHAVIOR section are collected in a default functional unit (unit-pipeName-stageName). Besides the functional units (and potentially a default unit) that basically implement all BEHAVIOR code of the model, each stage also consists of an instruction decoder (pattern matcher) for each of the operations in this stage, and a part for the activation signal generation of operations in this and the following stages. Each of this is implemented in a separate combinational process.

6.2.6 Pipeline Register XX (FE-DC-gen, DC-EX-gen):

The pipeline register modules hold all pipeline registers that are required to pass information from one stage to the following stage. Thus, only the required registers are implemented. For example, consider the following instruction: OUT.insn = code-from-ROM; If this assignment to the instruction pipeline register is in a fetch operation (in FE), and it is read in the DC stage, but not used in the further pipeline stages, then the insn pipeline register will be present only in the FE-DC module. Similarly if, for example, the pipeline registers operand1 and operand2 are only written in the DC stage and read in the EX stage, they will be present only in the DC-EX module. As a consequence of this,even a complete pipeline register stage may be omitted if no pipeline registers are used between two pipeline stages. On the other hand, if a

pipeline register is written in a certain stage and read in some other pipeline stage that occur later, then all pipeline registers that lie between these stages shall automatically be implemented and the value gets automatically shifted through these stages.

6.2.7 Functional units (U-FETCH-gen, U-CONTROL-gen, U-ARITH-DC-gen, U-ARITH-EX-gen):

These modules hold all operations that are grouped together in the RESOURCE section. Within a functional unit, each operation is implemented in a separate (combinational) process.

6.3 Comparison of the HDL codes generated

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 there desigine summery estimated value as The HDL code generated was synthesized using Xilinx ISE10.1.03. Figure 6.3 show the difference.

Device Utilization Summary (estimated values)		
Logic Utilization	**Processor 1**	**ASIP**
Number of Slices	8528	2573
Number of Slice Flip Flops	1200	887
Number of 4 input LUTs	16412	4983
Number of bonded IOBs	2	214
Number of MULT18X18SIOs	6	3
Number of GCLKs	1	1

Figure 6.3: LISA Code

Chapter 7

Simulation and Results

Simulation Results using Xilinx ISE

Synthesized report using Synopsys

Power Analyses Report

Area Analyses Report

Layout of the ASIP

Statistics for net list

Complete Global Routing

7.1 Simulation Results using Xilinx ISE

All the blocks of the processor are simulated individually using Xilinx ISE Figure.7.1 represents the top-module simulation result.

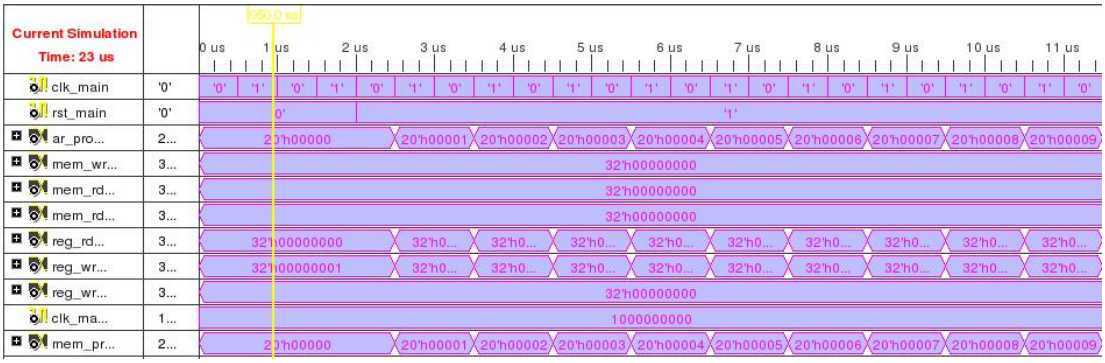

Figure 7.1: simulation result using Xilinx ISE

7.2 Synthesized report using Synopsys

After simulating the final processor Figure.7.2 represents synthesixed report of the processor and also the internal structure of the designed Processor. To find out the area and power of the circuit the logic is again synthesized using Synopsys tool.

7.3 Power Analyses Report

Power is find out from the power analyses report of the circuit and Figure.7.3 represents that from the report it is found out that the power consumption by the circuit is 86.5202 W.

7.4 Area Analyses Report

Area is find out from the area analyses report of the circuit and Figure.7.4 represents that, from the report it is found out that the area of the circuit is 43765 square micron.

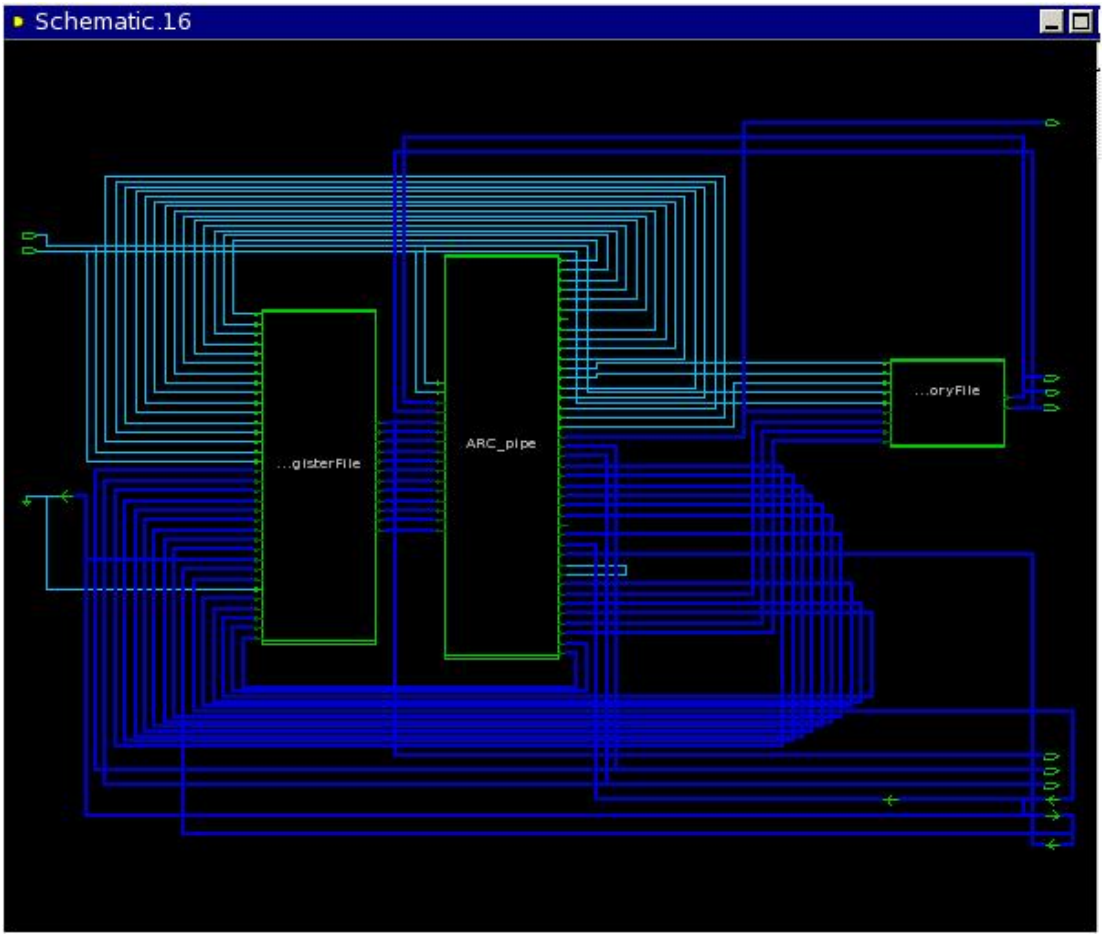

Figure 7.2: RTL synthesis

```
Global Operating Voltage = 5
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 0.100000ff
    Time Units = 1ns
    Dynamic Power Units = 100nW    (derived from V,C,T units)
    Leakage Power Units = Unitless


  Net Switching Power  =  86.5202 uW  (100%)
                          ---------
Total Dynamic Power    =  86.5202 uW  (100%)
```

Figure 7.3: Power Analyses Report

```
Number of ports:                214
Number of nets:                 732
Number of cells:                  3
Number of references:             3

Combinational area:       35116.000000
Noncombinational area:     8649.000000


Total cell area:          43765.000000
```

Figure 7.4: Power Analyses Report

7.5 Layout of the ASIP

The lay out of the processor is generated Cadence Encounter(Cadence-IC5141 UMC180nm technology) and Statistics for net list and Complete Global Routing report is gener-

ated. The Figure.7.5 show the layout.

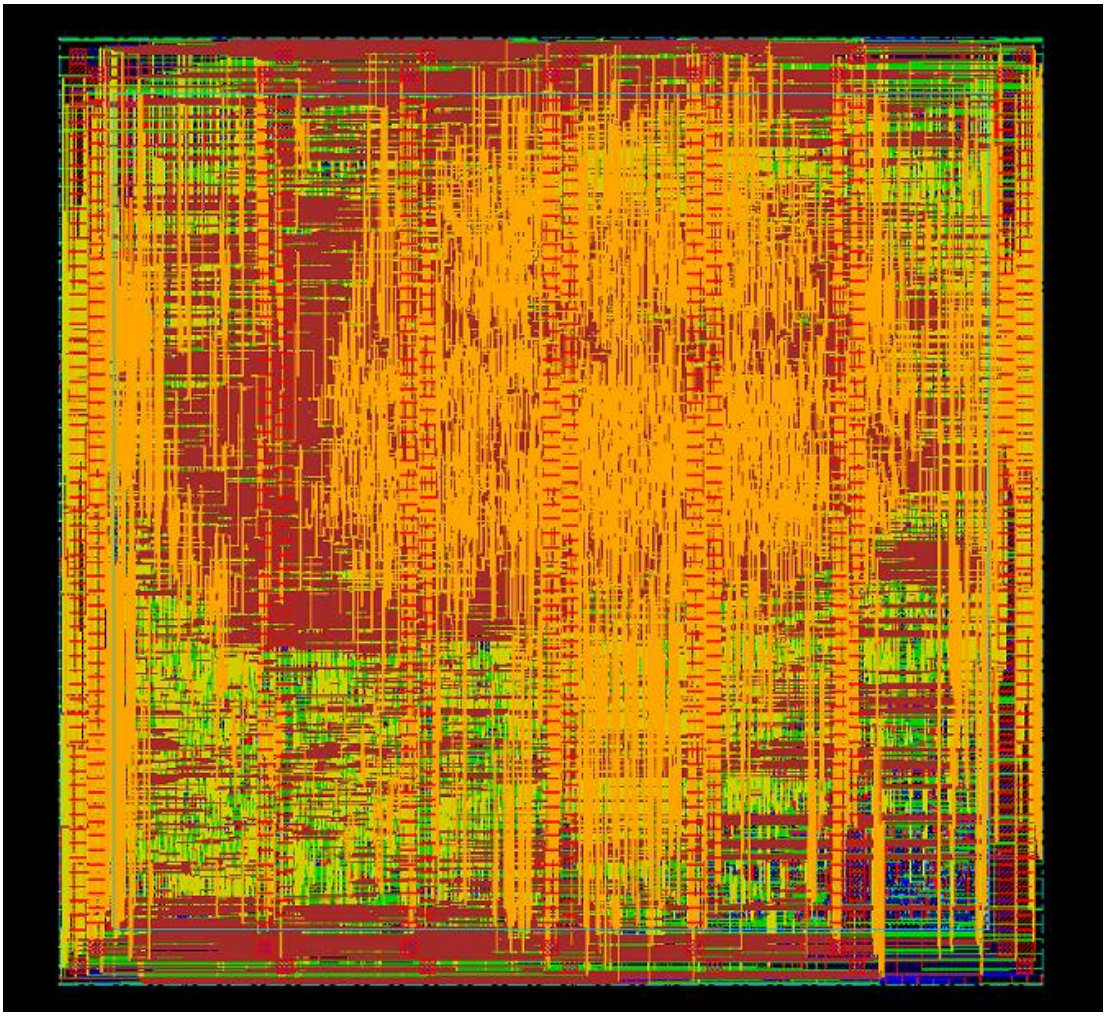

Figure 7.5: Layout of the ASIP

7.6 Statistics for net list

- Number of cells = 15232
- Number of nets = 11866
- Number of design nets = 228
- Number of pins = 48587
- Number of inputs/outputs = 214

7.7 Complete Global Routing

- Total wire length = 1500892 um.
- Total half perimeter of net bounding box = 762721 um.
- Total wire length on LAYER Metal1 = 19207 um.
- Total wire length on LAYER Metal2 = 195779 um.
- Total wire length on LAYER Metal3 = 403413 um.
- Total wire length on LAYER Metal4 = 360563 um.
- Total wire length on LAYER Metal5 = 338683 um.
- Total wire length on LAYER Metal6 = 183247 um.
- Total number of vias = 103657 um.

Chapter 8

Conclusion

Conclusion

Main Contributions

Future Work

8.1 Conclusion

Application Specific Instruction-set Processors (ASIPs) are a type of processor that serve as a compromise between General Purpose Processors (GPPs) and Single Purpose Processors (SPP). Their data-path can be optimized for a particular class of operations such as embedded control, Digital Signal Processing (DSP) applications etc. This project was based on designing an Application Specific Instruction-Set Processor whose instructions were tailored made for a specific application. This processor processed different temperature data of different areas and show the highest temperate with the location of that area. The simulation of this design was carried out on CoWare Processor Debugger platform. LISA code used to describe the architecture of this processor and CoWare tool generated HDL structure. Following the simulation work, the entire model was synthesized using Synopsys design vision(synopsys TSMC 65nm Technology) and layout was generated in Cadence encounter. This design has a three stage pipelined.

8.2 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, 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.

8.3 Future Work

In future we can go for designing a complex five stage pipelined processor 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.Further optimization can be done with respect to resources, memory size and power consumption by changing the LISA code written in CoWare platform.

Bibliography

[1] M. Hempstead, N. Tripathi, P. Mauro, G.-Y. Wei, and D. Brooks, "An ultra low power system architecture for sensor network applications," *P*roc. 32nd Annual International Symposium on Computer Architecture, Madison (USA) 2005, pp. 208-219.1.

[2] J. L. Hill, System Architecture for Wireless Sensor Networks, PhD Thesis. Berkeley, CA: University of California, 2003.

[3] J. M. Rabaey, J. Ammer, T. Karalar, S. Li, B. Otis, M. Sheets, and T.Tuan, "PicoRadios for wireless sensor networks: The next challenge in ultra-low-power design," Digest Tech. Papers IEEE International SolidState Circuits Conference, San Francisco (USA) 2002, pp. 200-201.Y.

[4] L. Gu and J. A. Stankovic, "Radio-triggered wake-up capability for sensor networks," Proc. JOth IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto (Canada) 2004, pp. 27-36.

[5] T. Burd, T. Pering, A. Stratakos, and R. Brodersen, "A dynamic voltagescaled microprocessor system," IEEE J. Solid-State Circuits, vol. 35, pp. 157 1- 1580,2000.

[6] G. Panic, D. Dietterle, Z. Stamenkovic, "Architecture of a Power-Gated Wireless Sensor Node," dsd, pp.844-849, 2008 lith EUROMICRO Conference on Digital System Design Architectures, Methods and Tools, 2008

[7] R. Amirtharajah and A. P. Chandrakasan, "Self-powered signal processing using vibration-based power generation," IEEE J. Solid-State Circuits, vol. 33, pp. 687-695, 1998.

[8] C. Kelly, IV , V. Ekanayake , R. Manohar, SNAP: A Sensor-Network Asynchronous Processor, Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems, p.24, May 12- 15, 2003

[9] Y. Ammar , A. Buhrig , M. Marzencki , 8. Charlot , S. Basrour , K. Matou , M. Renaudin, "Wireless sensor network node with asynchronous architecture and vibration harvesting micro power generator," Proceedings of the 2005 joint conference on Smart objects and ambient intelligence: innovative context-aware services: usages and technologies, October 12- 14,2005, Grenoble, France.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] Hoffmann, A.; Fiedler, F.; Nohl, A.; Parupalli, S.; , "A methodology and tooling enabling application specific processor design," *V*LSI Design, 2005. 18th International Conference on , vol., no., pp. 399- 404, 3-7 Jan. 2005 doi: 10.1109/ICVD.2005.20

[17] A. Hoffman, F. Friedler, A. Nohl and Surender Parupalli. A Methodology and Tooling Enabling Application Specific Processor Design. In Proc. of the VLSID, 2005

[18] Li Zhang, Shuangfei Li, Zan Yin and Wenyuan Zhao. A Research on an ASIP Processing Element Architecture Suitable for FPGA Implementation.In Proc. of the International Conference on Computer Science and Software Engineering, 2008.

[19] V. Ekanayake, C. Kelly, et al., An Ultra Low-Power Processor for Sensor Networks., ASPLOS04, October 2004.

[20] LISA Language Reference Manual supplied by CoWare.

[21] HDL Code Generation Guide supplied by CoWare.

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

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

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

[25] A. Hoffmann, H. Meyr, R. Leupers, Architecture Exploration for Embedded Processors with LISA, first ed., Kluwer Academic Publishers, Boston, 2002.

[26] U. Meyer-Baese, LISA Online Resource, 2010, ¡http://www.eng.fsu.edu/umb/lisa¿.

[27] IBM http://www.ibm.com.

List of Publications

[1] Lopamudra Samal, K. K. Mahapatra, "Designing a Low power 8-bit Application Specific Processor in VHDL," Embedded System(Paper accepted)