Integrating Instruction Set Simulator into a System Level Design Environment

A Thesis Presented
by

Akash Agarwal

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts

February 2013
To my family.
Contents

List of Figures iv
List of Tables vi
List of Acronyms vii
Acknowledgments viii
Abstract of the Thesis ix

1 Introduction 1
  1.1 Embedded System 1
  1.2 Growth and Challenges 3
  1.3 System Level Design 4
  1.4 Virtual Platform 6
  1.5 Problem Definition 9

2 Background 10
  2.1 System on Chip Environment 10
  2.2 Discrete Event Simulation 11
  2.3 Instruction Set Simulator 13
  2.4 Blackfin Family Processors 16
  2.5 Related Work 16

3 ISS Integration Guidelines 18
  3.1 Introduction 18
  3.2 ISS Integration 19
    3.2.1 Execution Model Integration 20
    3.2.2 Time Synchronization 23
    3.2.3 Bus System Integration 31
    3.2.4 Interrupt Propagation 34
    3.2.5 Support Debugging Environment on ISS 37
    3.2.6 Support Profiling Execution on ISS 37
# Integrating Blackfin GDB ISS

## 4.1 Introduction

## 4.2 GDBISS Integration

- 4.2.1 Execution Model Integration
- 4.2.2 Time Synchronization
- 4.2.3 Bus System
- 4.2.4 Interrupt Propagation
- 4.2.5 Support Debugging Environment on GDB ISS

# Experimental Results

## 5.1 Accuracy of BF527 GDB ISS

- 5.1.1 Introduction
- 5.1.2 Experimental Setup
- 5.1.3 Analysis and Results

## 5.2 Effect of Quantum-based Time Synchronization on Interrupt Latency

- 5.2.1 Introduction
- 5.2.2 Experimental Setup
- 5.2.3 Impact of Quantum Synchronization onto Quantum Latency and Response Time
- 5.2.4 Impact of Variable Quantum Length on Total Simulation Time and Quantum Latency

## 5.3 Design Space Exploration

- 5.3.1 Algorithm Under Test
- 5.3.2 Test Results

# Conclusion

# Bibliography

# Test Cases

## A.1 Specification

- A.1.1 Execution and Time Synchronization Test Case
- A.1.2 Bus Read and Write Test Case

## A.2 Architectural Mapping

- A.2.1 Execution and Time synchronization Test Case
- A.2.2 Bus Read and Write Test Case
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Embedded System Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>System On Chip Overview</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Productivity gap</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>ITRS Report 2011</td>
<td>4</td>
</tr>
<tr>
<td>1.5</td>
<td>Abstraction Levels</td>
<td>5</td>
</tr>
<tr>
<td>1.6</td>
<td>SoC Design Flow</td>
<td>7</td>
</tr>
<tr>
<td>1.7</td>
<td>Simplified Virtual Platform</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>System On Chip Environment</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>ISS Software Flow</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Core ISS: ISS with only Core support</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>System ISS: ISS with core &amp; other peripheral support</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>SoC Block Diagram showing only the Essential Components</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Simplified VP block Diagram</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>ISS Software Main loop</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>ISS Software Main Loop Modified</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>Sample Code Execution Sequence on SoC showing 3 different types of possible</td>
<td>24</td>
</tr>
<tr>
<td>Code Sequence on ISS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Update of VP time due to Execution of Component Models in Quantum</td>
<td>25</td>
</tr>
<tr>
<td>3.7</td>
<td>Time Synchronization at Quantum Expiration</td>
<td>25</td>
</tr>
<tr>
<td>3.8</td>
<td>Execution Sequence of the Specification till Bus Access</td>
<td>26</td>
</tr>
<tr>
<td>3.9</td>
<td>Time Synchronization at Bus Access</td>
<td>27</td>
</tr>
<tr>
<td>3.10</td>
<td>Execution Sequence at Interrupt Propagation</td>
<td>28</td>
</tr>
<tr>
<td>3.11</td>
<td>Time Synchronization at Interrupt</td>
<td>29</td>
</tr>
<tr>
<td>3.12</td>
<td>Core ISS VP, depicting the Bus Channel Integration</td>
<td>32</td>
</tr>
<tr>
<td>3.13</td>
<td>System ISS depicting the Bus Channel Integration</td>
<td>32</td>
</tr>
<tr>
<td>3.14</td>
<td>Core ISS VP depicting the Interrupt Integration</td>
<td>35</td>
</tr>
<tr>
<td>3.15</td>
<td>System ISS VP depicting the Interrupt Integration</td>
<td>36</td>
</tr>
<tr>
<td>4.1</td>
<td>Simplified Blackfin GDB ISS block diagram</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>Blackfin 52x Memory Map</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>BFIN based VP depicting the Bus System Integration</td>
<td>43</td>
</tr>
<tr>
<td>4.4</td>
<td>BFIN based VP depicting the Interrupt Connections</td>
<td>44</td>
</tr>
</tbody>
</table>
# List of Tables

| 5.1 | Core Clock Cycles on BF-527 HW and ISS for Benchmark | 51 |
| 5.2 | Memory Stall numbers for benchmark | 52 |
| 5.3 | Quantum Latency and Simulation Time for LOAD application | 58 |
| 5.4 | Table showing Quantum Latency variation over quantum length for BUSACCESS application | 59 |
| 5.5 | MP3 decoder exploration on ARM and BF516 based VP | 62 |
List of Acronyms

**ESL**  Electronic System Level A Electronic System Level design is an emerging electronic design and verification approach which focuses on higher abstraction level for design

**GPIO**  General Purpose Input and Output A General Purpose Input and Output is a generic pin in a chip, whose behaviour and functionality is controlled by the user at run time

**HDS**  Hardware Dependent Software Hardware Dependent Software is the part of the operating system, comprising mainly of device drivers and boot code. It serves the functionality of hardware initialization

**IC**  Integrated Circuit Integrated Circuit is a set of electronic circuit on one small semiconductor material

**ISS**  Instruction Set Simulator An Instruction Set Simulator is a software that simulates the target processor.

**PIC**  Programmable Interrupt Controller Programmable Interrupt Controller is a device that is used to combine several sources of interrupt onto one or more CPU lines.

**RTL**  Register Transfer Level Register transfer Level is a design abstraction which models synchronous digital circuit in term of flow of digital signal between hardware registers

**SoC**  System on Chip A Integrated Circuit component which integrates different components in a electronic device in a single chip

**SLDL**  System Level Design Languages System Level Design Languages is used to aid the design and develop specification of digital embedded system

**SCE**  System on Chip Environment System on Chip Environment is a Electronic System Level Design Tool developed at University of California, Irvine.

**TLM**  Transaction Level Modelling. Transaction Level modelling is an high level approach of modelling communication among modules as set of transactions

**VP**  Virtual Platform Virtual Platform is a software implemented abstraction of the hardware
Acknowledgments

I wish to thank my family, friends and a special thanks to Mike Frishenger for his guidance and developing Blackfin ISS under GNU, Rohan Kangarkalar for his work on RTOS integration, and Prof Gunar Schirner for his continuance guidance and support.
Abstract of the Thesis

Integrating Instruction Set Simulator into a System Level Design Environment

by

Akash Agarwal

Master of Science in Electrical and Computer Engineering
Northeastern University, February 2013

Dr. Gunar Schirner, Adviser

Design of Embedded System, which today comprises of both Hardware and Software (HW/SW) has become complex with the advancement of technology and with the ever increasing demand for complex and low cost features. Systems developed today should arrive must arrive at market quickly and at the same time meet the constraints of power, memory and cost.

As one solution, designers are moving to higher level of abstraction or System Level design which covers both Hardware (HW) and Software (SW). This is realized through a model based design methodology which utilizes virtual environment for early validation of the specification and at the same time also provides for verification of functionality of the system. It also augments the design process by providing a pre-silicon prototype of the final physical system for software development and allows for and helps in performance tuning of these complex systems. Furthermore, virtual platforms can also serve as inputs to Register Transfer Level (RTL) and software component synthesis for the system.

Modern Embedded Systems are heterogeneous as they have both a programmable processor core and dedicated hardware units that is the virtual platform which simulates the system. Instruction Set Simulator (ISS) provide a simulation environment for programmable processors and are utilized in a virtual environment to validate software running on programmable processor. Today there are many stand-alone ISS which are available in the open source community, however they are not integrated with a system design tool suite making it impossible to utilize them to validate SW. Manual effort to individually add them to a system-level design tool suite requires work for each addition and is prone to error.

In this thesis we focus on providing a generalized guidelines for integrating ISS to a system-level design tool. The guidelines are developed in order to simplify the process of adding an
ISS to a system level design tool. The developed guidelines takes into account the requirements to model communication between processor and other computing components in an System on Chip (SoC), and also provide to interrupt based synchronization capability between components. It also takes into account the requirements to reduce simulation time which is a major bottleneck and which limits the use of system simulation. The guidelines developed also take into account different types of ISS, one which simulate only the processor core and the other ones which simulate both the core and peripheral of the processor together.

We validate the generalized guidelines by integrating a Blackfin family processor ISS available under GNU tool chain with a system level design tool System on Chip Environment (SCE), which support modeling, synthesis and validation of Heterogeneous Embedded Systems.
Chapter 1

Introduction

1.1 Embedded System

An embedded system is a computer system which is designed to perform a particular functionality operating within constraints of memory, power and cost. Embedded systems today are pervasive and touch virtually every aspects of our life today as shown in Figure 1.1. They can be seen easily identified in Automotive, Avionics, Industrial Automation, IT/Hardware, Consumer Electronics, Telecommunication and Medical domains of industry. The development of these systems has made our life easier and brought technology to our doorstep. The development of TV set-top boxes, automobile engine management systems, mobile phones, personal digital assistant, MP3 players, and photocopiers has made it possible for the world to connect, learn and grow. It is hard to imagine our daily lives now without embedded systems.

Figure 1.1: Embedded System in our daily life [7]
CHAPTER 1. INTRODUCTION

Embedded systems today have grown from the age of simple micro controller based design to multi and heterogeneous processors on a single chip, known as a System on Chip Environment. System on Chip (SoC) is an integrated circuit Integrated Circuit (IC) integrate all digital components on a single chip. SoC have the capability to be interfaced with external memory chips (flash, RAM) and are interfaced with various external peripherals based on system requirements. These blocks are connected using proprietary or standard industry bus standards. SoC contains both hardware and software controlling the processors on the chip. An example SoC developed by NVIDIA [11], for mobile devices such as smart-phones, personal digital assistant and mobile devices is shown Figure 1.2, integrates a ARM [2] architecture processor, a graphics processing unit, and memory controller on a single package. The complexity of the system can be realized from the fact that it comprises of ARM processor which acts as a master, an image processor dedicated for image processing, an video processor dedicated to video processing, and an GPU for graphical processing and a rich set of peripherals for communication with the external world. Designing hardware and software configuration for systems like this is a huge challenge due to the number of processing components that they involve, and the available choices in selecting components. Developing software for this complex hardware is a major challenge facing the designers.

Figure 1.2: NVIDIA Tegra SoC (Source:[11])
CHAPTER 1. INTRODUCTION

1.2 Growth and Challenges

An Embedded System which comprises both hardware (integrated circuits and boards) and software continues to grow at an phenomenal rate. The worldwide market for embedded systems is expected to reach $158.6 billion by 2015 from $101.6 billion in 2009 as per BCC research [4] which is a compound annual growth rate (CAGR) of 7%. The highest growth rate as identified by [4] research comes in terms of revenues comes from the embedded software (operating systems, design automation and development tools).

The growth of semiconductor manufacturing technology from 800nm in 1989 to 22nm today has allowed for more transistors to be fitted in a small die area, thereby reducing cost and power while increasing the processing and memory capacity of [IC]. These advancement have given a tremendous boost to embedded system industry. The growth in [IC] technology has significantly increased the number of transistors available for design on a chip. The processing capacity of transistors doubles every 18 months following Moore’s law [34] but hardware designers are only able to utilize 1.6x over 18 months Figure 1.3. Unfortunately, the software productivity or the ability to utilize available capacity, has not been able to grow at the same pace, it grows to only 2x over a period of 5 years. This has created a huge productivity gap between the available technology at its utilization factor. A rise in software productivity will help in multiple ways on one hand it will increase the utilization factor of existing hardware and on the other hand it will allow for more features to be added to embedded systems.

Figure 1.3: Productivity Design Gap (Source: [9]).
CHAPTER 1. INTRODUCTION

The huge increase in the availability of transistors on a chip, along with the rising pressure to reduce the time to market and the cost as well as an ever increasing demand for complex features has put tremendous pressure on system designers. With this background system designers today are faced with dual challenges to match the needs and demands of consumer and at the same time design and develop products within the constraints of cost, memory and power. Embedded system constraints have now expanded now including time to market, memory footprint and ability to respond to external events which is the major criteria for a system to be a success or failure.

Software design cost continues to drive the cost of SoC. Various sources confirm that software development cost will out-pace the hardware development cost. International Technology road-map for Semiconductors (ITRS) [9], which has published a cost chart showing the hardware and software engineering costs for a SoC in its annual design report.

As shown in Figure 1.4 for year 2012, the estimates are $26M for hardware and $79M for software and for 2013, it is predicted [10] that the software development costs will drop and reach parity with hardware costs which can be augmented by the fact of many new core development tools and development of new design methodology.

![ITRS 2011 Cost Chart](image)

Figure 1.4: ITRS Report 2011 (Source:[9])

As pointed out in Figure 1.4, the software development cost will continue to out-pace the hardware costs in the years to come and keeping it down is one of the major challenges facing semiconductor industry in the years ahead.

1.3 System Level Design

Embedded system designers today are facing multi dimensional challenges, from the hardware side they have to reduce the productivity gap and prepare themselves for advancement
CHAPTER 1. INTRODUCTION

in fabrication technology, on the other side they face pressure from the market to reduce the time to market, cost, power requirements, and on the ever increasing demand of customers for advanced features in their devices. A new design methodology referred as Electronic System Level (ESL) has been proposed and developed to counter these challenges. ESL methodologies focus on designing at higher level of abstraction as shown in Figure 1.5. At higher level of abstraction the number of components in design decreases. A typical digital system will consists of million of transistors at its lowest level, which are then reduced to thousands when grouped at Register Transfer Level (RTL). Furthermore, the trend continues and finally at the system level the components are identified as processors, hardware units, memories and buses. It is to be noticed that moving up the abstraction level decreases the accuracy of design as shown in Figure 1.5 but designers at higher abstraction levels benefit by reduction in number of objects to be handled at one time this allow for more application specific development.

![Abstraction Levels](image)

Figure 1.5: Abstraction Levels

In this new era of HW/SW co-design, it has become important to have programming languages which have the ability to describe both hardware and software. Traditional programming languages such as C, C++ and Java have constructs to describe software aspects, but lack the ability to be able to describe hardware mainly due to missing constructs of time, concurrency and signal communication. On the other hand traditional hardware languages such VHDL/Verilog have the ability to describe hardware but do not have constructs to describe software design of a system. System Level design languages SLDL SystemC [30], SpecC [28] have been proposed in order to solve this problem. Both of these languages are based on discrete event driven model which allows for support for concurrency, synchronization, and timing and at the same time have a rich set of
abstract data types to describe the software.

Components of a system, mainly perform two major functions computing and other communication. In order to describe a system at higher levels of abstractions computation and communication have to be separated and abstracted. In order to abstract a digital system, mainly communication needs to be abstracted from wires, pins to channels which encapsulate lower level details of design. Transaction Level Modelling (TLM) proposes the concept of channel, which encapsulates the details of communication protocol, pins and wires. Both SystemC and SpecC System Level Design Languages (SLDL) provide with a module and a behavior construct to encapsulate the computation of a component. TLM along with SLDL provide with a set of interfaces for communication among modules or behaviors. In TLM defines the communication as set of data transfer between component. TLM have allowed designers to develop early platform better known as virtual platforms for pre-rtl software development and validation, architectural analysis and functional verification. ESL biggest contribution to SoC design flow is its ability to provide with Virtual Platform (VP). VP help to parallelize development of both hardware and software and provide scope for analysis to make system level design decisions at the pre-silicon stage.

1.4 Virtual Platform

Traditional embedded design methodologies which focussed on hardware to software or a software to hardware based design approaches are not valid for todays embedded system due the increase in complexity and time to market of the product. In order to meet the challenges of complexity of today embedded system a joint design approach encompassing both hardware and software better know as HW/SW co-design has been developed. A design flow for such an approach is shown in Figure 1.6. As shown in figure the design starts with a specification of the system, which is then divided among the hardware and software which is called hardware-software partitioning. Once the partitioning is done then TLM based virtual platform are used for concurrent hardware software development. TLM based VP allows for finding the best possible hardware/software configuration for the given application. Once the final configuration of hardware and software is decided a test chip is developed. The test chip is used for post silicon validation and system integration. Once the test chip meets the specification requirement then the design is sent for chip fabrication. As we can see VP plays a major role in this design flow, it allows for concurrent development of hardware and software, it helps designers evaluate performance for different hardware and software configuration. Hence we can see the VP play a major role in SoC design and development today.
CHAPTER 1. INTRODUCTION

Development of a virtual platform is one of the major contributions of the ESL design methodology. A VP is a TLM model of a SoC under after the specification and HW/SW partitioning phase in a design flow. VP can also act as input for the RTL model and aid in hardware synthesis, however research work in this area is in progress [32]. A VP will consist of modeled components of SoC as shown in Figure 3.1. These modeled components will simulate the functionality of SoC and help to develop software, perform architecture analysis and verification. Figure 1.7 shows a highly simplified view of a virtual platform, showing an ISS to simulate the processor, timer, PIC, memory, bus and an arbiter models.

Figure 1.6: SoC Design Flow, Source [29]

Figure 1.7: Simplified Virtual Platform
ISS which simulated the processor, is an integral piece in a virtual platform as it performs the same functionality of a processor in SoC, it communicates with other peripheral via a modeled communication component specific to processor. The modeled communication component is decided based on HW/SW partitioning decision and architecture decisions. The connectivity of communication for a multiprocessor SoC, will be different then a SoC with a single processor and hardware accelerator. The architecture configuration, depends on the specification of the SoC. Virtual platform accuracy levels depends on the accuracy level of the modeled components. The modeled components can be all at cycle accurate, some can be cycle accurate or none are cycle accurate. Cycle Accurate models of components are required to perform architecture verification, performance analysis, architectural exploration, tuning of Application code and Power estimation. It is to be noted that simulation speed decreases as the level of accuracy of VP increases.

The complexity of obtaining a virtual platform depends on the specification and the design level at which the SoC prototype is constructed as shown in Figure 1.6. If for example an SoC needs to be designed from the customer specification level which is agnostic to computation and communication demands then, the design space needs to be explored. Each decision needs to be verified and the model obtained needs to be validated against the specifications requirement. On the other hand if in SoC most of architectural components are fixed and only a particular component of a system needs to be explored for example adding a hardware accelerator for dumping a dsp fft operation then the complexity decreases.

The verification and validation at each level of design space exploration can be a tedious and error prone methodology and will depend on the expertise of designers. Using a tool for such a process would automate and reduce manual errors. There are both academic and commercial tools available which are able to provide a virtual platform. These tool cater to different segments of industry. The Cadence Virtual System Platform [5] provides creation and support of virtual prototypes with automated modeling, this mainly helps in early development of the SoC software side at the same time providing with limited architectural exploration. Synopsys Platform Architect [16] provides system designers with cycle accurate architecture models utilizing SystemC and TLM protocols and also provide the ability to integrate carbonized model which are abstracted from RTL models [6]. These model provide cycle accurate information on processor execution and interconnect fabric. These types of information helps in designing architecture configuration, performance analysis and optimization at architecture level. Similar tools is SCE which is developed at University of California(UCI). It provides for top-down refinement and validation using ISS. There has been significant research in the development of the tool. All the tools mentioned above utilize ISS.
CHAPTER 1. INTRODUCTION

to provide a co-simulation and validation of software.

1.5 Problem Definition

As discussed earlier ISS is an integral part of VP and ISS is a software which simulates a particular processor. Traditionally ISS have been developed as stand-alone program and they were never meant to be integrated or combined with and other system design tool. There are many ISS models available under [14], [8] and [12]. These ISS models differ mainly on account of accuracy (functional/cycle) and simulation time either (interpretive/compiled/JIT). A processor in an SoC performs the functionality of executing instruction, bus R/W, allow for interrupt and for clock synchronization. There have been approaches for integrating ISS in system level design framework. However these approaches are varied and suffer from discrepancies in modeling bus access, interrupt and time synchronization which makes integrating them in a ESL tool a challenge. In this thesis we formalize the integration of stand-alone ISS in system level design framework with ability to have bus and interrupt communications. We also study methods to reduce system simulation time and study its effect on accuracy. As a validation and demonstration of our approach we integrate a ADSP-BF527 processor [1] ISS under [8] in SCE [17] tool. We then perform design space exploration on a mp3 decoder for the newly integrated BF527 and already existing ARM[3] based virtual platform as a demonstration to enhance capability of the tool to perform multiprocessor design space exploration. This thesis is organized as follows: chapter[2] shows in detail the background to our work and also describes the other related work in the same area, chapter[3] shows our proposed guidelines for integration, chapter[4] shows a sample integration using the guidelines proposed in[3] and[5] contains the experimental results. Test cases developed as part of our generalized guidelines are discussed in the appendix.
Chapter 2

Background

2.1 System on Chip Environment

System on Chip Environment (SCE) is an Electronic System Level (ESL) tool. The tool is based on Specify-Explore-Refine methodology [27], in which the user is asked for the specification, the tool provides options to explore and make decision and then it refines the design based on the decision. The tool allows system-level designers to capture the system algorithm or design in a set of behaviors using the SpecC [38] which is System Level Design Languages (SLDL). The input design is then converted into a function model called the specification model. The specification model defines the desired application platform, framework and desired requirements. SCE then allows the user to make three major design decisions, allocating computation blocks, scheduling behaviors, communication connectivity among computational blocks. The tool utilizes a component database to provide users with different processors, hardware accelerators, operating system and buses for design space exploration at each design stage. After each design decision the tool generates a Transaction Level Modelling (TLM) model, which allows for validation and verification of design and the decision itself. After these major design decisions are augmented to initial specification, tools allow designer to generate a Bus Functional Model (BFM) and a Pin-accurate Model (PAM) for the design. BFM model is a TLM model which contains the bus model abstracted to atomic transactions in order to simplify the modelling of bus transactions, PAM model on the other hand is more accurate model which contains the bus model specific to the specific protocols.

The BFM or the PAM model can then be synthesized to RTL components using the hardware database or synthesized to VP using the Software (SW) database. SW database consists of ISS for generic processing element. During the refinement stages the the Hardware Dependent
CHAPTER 2. BACKGROUND

Software (HDS) along with drivers, and the target specific code partitioned on the processing element is compiled into an executable target binary code. The target binary code can be executed on ISS for target binary validation. The tool automates the process of creating a VP for a user specification, which reduces the time for design space exploration. VP for Pre-RTL software development, performance an architectural analysis. In this thesis we validate our approach by integrating a ADSP-BF527 processor [1] ISS available under [8] in the SCE tool.

2.2 Discrete Event Simulation

SLDL provide discrete event time model in order to support for concurrency, timing and synchronization. They provide these features enabling them to capture and model hardware behaviour. In a discrete event time model, the operation of a system is represented as sequence of events. These sequences of events are generated from each computation block represented as threads within the system simulation environment. These events occur at a particular instant of discrete time and cause a change in the system status. SLDL provide a run time environment (simulator) and in order to coordinate the simulation process provides for a distinguished kernel process.
CHAPTER 2. BACKGROUND

This kernel process supervises the execution of all threads and also takes care of incrementing the simulation time and time synchronization of all the threads. In order to assist kernel to decide when and which thread to simulate SLDL provides for specific constructs through which the kernel can run or suspend a particular threads within the simulation environment. During simulation an active thread can either be in a running state or in a suspended state. Threads are in running state when the event on which they have been waiting occurs in the system or if there are not waiting on any event and are in a suspended state while they are waiting for an event. Each event causes a change in the system, as they cause a thread to resume execution. In order for a suspended thread to resume execution, the current executing thread needs to be suspended. This causes a context switch on the host system. This context switch is important because the context of the current executing thread need to be preserved before the suspended thread is allowed to execute. This context switch causes a increase in simulation time. High simulation speed is desirable in order to effectively use VP in SoC design and development. In order to reduce the simulation time, the number of context switches have to be reduced.

A SLDL [VP] which consists of software model of each component in an SoC. These software models are represented as software threads which run under the supervision of simulation kernel. Simulation kernel is responsible to schedule execution of these software threads. In order to schedule these software threads the simulation kernel utilizes the discrete event time model. The time modes utilize a virtual clock in order to synchronize events. This clock incremented based on start of event as opposed to real time clock which increases linearly. In case of VP there are two virtual clocks, one is the VP virtual clock maintained by the simulation kernel and other is the ISS virtual clock maintained by the discrete model inside the stand alone ISS. These two clocks need to be time synchronized with each other. Virtual clocks in a discrete event model are synchronized by suspending the current executing discrete model for the virtual time it has executed and then allowing the other other discrete model to execute.

There are various mechanism which can be adopted through which it can be decided when these two discrete event clocks can be synchronized in order to reduce the simulation time. These mechanism can be varied in from the analyzing the software executing on the ISS [31] or using traces to divide the event generation and event alignment stages in simulation[37]. We in this thesis distinguish clocks synchronization in two general mechanisms:

1. Lock Step: ISS and other component models are synchronized every cycle. Synchronization every cycle has the advantage of added simulation accuracy but suffers from the disadvantage
CHAPTER 2. BACKGROUND

of high simulation time.

2. At Quantum Expiry: ISS is assigned a predefined quantum, based on the number of simulation steps (i.e. instructions or cycles) it can advance without synchronizing its clock. Quantum can be defined as a definite time in terms of processor clock for which the ISS executes before synchronizing the ISS clock and the virtual clock. The quantum can be adjusted during the initialization of ISS.

3. At Bus Access: In order to ensure that external data access happens at correct time and in order to main, the ISS clock and the time need to be synchronized at bus access.

Simulation time increase with a small quantum and decrease with a large quantum. The accuracy of simulation also increase with a small quantum as it allows for other component model to execute more frequently, while a large quantum suspends other component model for a longer time.

2.3 Instruction Set Simulator

An Instruction Set Simulator (ISS) is typically a stand alone application which simulates the functionality of a processor instruction set of a particular processor. The figure 2.2 below shows the sequence of simulation of the target application on the ISS. As shown, a target application is compiled with a cross compiler specific to the processor. The binary generated is then passed as an input to the ISS. ISS then simulates the target application by interpreting and simulating each individual instruction. Its resulting functionality, is as if it would run on the target architecture.

ISS can simulate a processor at either function-level or cycle-accurate level. In the case of function accurate models focus is only on simulation of the instructions behaviour rather then modeling the data-path. The processor pipeline in this case is coarsely modeled and more focus is on the functionality of processor. Functional accurate ISS have advantage of high simulation speed and give an idea of capability of the processor. In case of cycle-accurate model the data-path of the processor is accurately modeled and each instruction is modeled as it would execute on real processor. The cycle accurate simulation of processor gives the advantage of accurate simulation of target application but suffers from the disadvantage of low simulation speed. ISS’s can be classified based on how they simulate processor execution:

1. Interpretive: These simulators interpret the target processor instruction and store state of target processor in host memory and then follow the fetch decode and execute cycle in the
serial order. These models are simple to construct and are flexible but the limitation with these models is that these simulators are slow, primarily due the overhead of decoding and executing the target instruction. Some interpretive ISS increase the simulation speed by storing decoded instructions into a decode buffer for later reuse. Examples of such simulator are the ISS available under GDB [8], TRAP [18].

2. Compilation Based: These simulators translate each target processor instruction into a series of host instruction. The translation is done at compile time (static compiled simulation), where the fetch-decode-execute overhead of the interpretative simulation is eliminated. This simulator suffers from the disadvantage that this does not conceptually model the processor as in case of an interpretive simulator [21].

3. Dynamic Compilation (Just In Time): The slow simulation speed of interpretive simulation is mainly due the time consuming process of decoding the target instruction. This can be eliminated by using compiled simulation operation approach of translating target machine instruction into a series of host instruction. This significantly remove time consuming process of decoding target processor instruction QEMU [14].

ISS’s can also be varied with the number of supporting peripheral apart from the core they support in there software model. Based on this classification the ISS’s can be broadly classified as follows:
CHAPTER 2. BACKGROUND

1. Core ISS: These ISS provide support for only the core of processor as shown in Figure 2.3. They just model an internal memory and the core of the processor. The main focus of these ISS is to model instruction of the processor. Examples of these are ISS available under TRAP [18]. QEMU also has set of only a core simulating ISS.

2. System ISS: These ISS provide support for the core and also a set of other supporting peripheral along with the processor core as shown in Figure 2.4. The number of supported peripherals can be different for different ISS, and it depends on the purpose for which the ISS has been designed. Examples of these ISS are available under GDB [8] and some under the QEMU [14].

The differentiation between core and system ISS is important for integration into a VP. When integrated into a VP the ISS needs to provide for interaction between the ISS and VP. The
way these interaction are to be extracted from an ISS hugely depends on the type of ISS. In the next chapter discuss the interaction in details on how they are to be extracted for different types of ISS.

2.4 Blackfin Family Processors

In order to validate our guidelines for integration of ISS in a system level design tool, we have chosen Blackfin family processor from Analog Devices [1]. The Blackfin digital signal processors are designed specifically to meet the computational demands and power constraints of embedded systems. They are widely used for embedded audio, video and communications applications. Blackfin processors utilize a RISC programming model, and provide functionally for advanced signal processing. This combination of processing capabilities allow Blackfin processors to perform equally well in both signal and control processing applications. This also eliminates the need for a heterogeneous processing processor, thereby simplifying hardware and software design implementation.

The Blackfin family of processor offer a dual MAC and single instructions multiple data instructions capability in its instruction set. The Blackin core supports for parallel issue with some limitations. The processor supports a wide range of peripheral which make it a suitable of system on chip solutions.

We in this thesis demonstrate our approach by integrating a Blackfin family processor simulator provided under the GDB framework [8] with System on Chip Environment [SCE].

2.5 Related Work

ISS have been used for a long time for analyzing, generating traces for machine instruction of a target processor. ISS allows for a methodology for analysis and optimization of target binary, as it simulates the behavior of each instruction. There are different types and categories of ISS as explained in section [22]. Utilizing ISS for a system on chip simulation is also has been a topic of research and there has been significant research going on in this area.

Wang et.al. and Monton et.al. [33] have utilized QEMU [22] using systemC wrapper to develop a framework to test HW/SW partitioning and parallelize development of SW, Operating system and HW. The work is focussed on utilizing the QEMU ISS and can not be generalized to be used with another ISS’s, also the work is more focussed on modelling the data communication and does not model the interrupt based communication between processor and hardware modules.
Imperas provides Open Virtual Platform (OVP) which allows to construct Virtual platforms using the OVP API’s. The OVP models provide for SystemC wrapper in order to use them under the SystemC environment. Our work is closely resembles by the work done by Imperas in providing generic wrapper for OVP platforms, but in our generic guidelines clearly define methods to reduce the overhead of time synchronization between the simulating models which are not very intuitive. Open cores which supports ork1sim, which is a generic OpenRISC 1000 architecture simulator capable of emulating OpenRISC based computer system, also provides for mechanism to extend ork1sim for integration into a SystemC based platform. The method proposed are one of case and are not generalized enough to be used with other ISS’s.

Schirner et.al. have also integrated SWARM a cycle accurate simulator for ARM7tdmi inside the SCE framework. In their work they have laid down the foundation for integration of ISS in the SCE framework. We in our thesis further extend the work in order to generalize the integration wrapper and simplify the integration of other ISS in the SCE framework. Pablo has also integrated an OVP based ARM-9JES processor inside the SCE framework but the work is not generalized enough in order to simplify the integrations of ISS.

Significant prior work has been invested into extension of ISS of use in system level design tools. Different researchers have followed different approaches in order to extend the use of ISS within a VP. However, these approaches have only focussed on the particular instance and no general guidelines for integration were given. This hinders extending the work and integrating additional ISS. In our work we aim to solve this problem and provide a generalized set of guidelines for any future integration of ISS into another discrete event simulation tool.
Chapter 3

ISS Integration Guidelines

3.1 Introduction

In this chapter we show our generalized guidelines for integration of Instruction Set Simulator (ISS) into a Virtual Platform (VP). We follow a two step approach for developing our integration guidelines. First we develop our generalized guidelines based on the procedure for integration of an processor IP with an physical SoC, and then we add on the additional steps that are required for integration of discrete time model of processor IP that is an ISS with another discrete time model of an SoC that is VP. We also show our generic System Level Design Languages (SLDL) wrapper function which encapsulates these functionalities and is used to integrate the ISS within a VP.

A SoC comprises of a processor, computing blocks, and set of peripherals as shown in Figure 3.1. SoC provide a bus system and a common clock in order to provide data communication and execution of the components. In an SoC all the components run concurrently and are time synchronized with each other. A SLDL VP replicates physical platform and provides a way of doing pre-silicon software development, performance optimization and hardware and software configuration. It provides for a TLM model of bus for data communication and provides a run time environment and a distinguished kernel process which co-ordinates the simulation process. ISS which is typically a stand alone simulator of the processor is developed in either 'C' or ‘C++’ programming language. These languages do not provide for support for discrete event simulation hence the ISS can not be directly utilized within a SLDL VP. In order to utilize the ISS it is required to wrap the ISS within an SLDL wrapper. The SLDL wrapper allows for co-simulation of ISS within a VP. This SLDL wrapper should be able to connect the SLDL VP bus model with the ISS internal
bus model in order to allow for data communication between them. It should also be able to forward interrupts generated from other component model to the ISS in order to allow for interrupt based synchronization. The SLDL wrapper also needs to utilize the discrete event simulation semantics of the SLDL in order to allow itself to be run under the SLDL run time environment. Internally the SLDL wrapper should be able to initialize and run the ISS. A SLDL VP provides for performance optimization through a profiling the software execution. The SLDL wrapper should be able to provide these features. A SLDL VP provides for pre-silicon software development, and in order to provide a software development it is important to have a debugging environment. Hence the SLDL wrapper should be able to provide for a debugging environment.

In summary in order for an ISS to be integrated into a VP, the SLDL wrapper should provide the following capabilities:

1. Provide function to initialize and run ISS within the VP
2. Provide capability to connect bus traffic to the processor
3. Provide capability for interrupts propagation from system to processor
4. Provide capability for time synchronization with other peripheral components on the SoC
5. Provide features for debugging target application on processor
6. Provide ability to profile instructions and event execution on processor

The next sections describe how to realize the above mentioned capabilities for integrating an ISS into an SLDL environment.

### 3.2 ISS Integration

Figure 3.1 shows an simplified diagram of an SoC showing the essential components of an SoC. This is a simplified diagram of an SoC and it does not represent the modern SoC as shown in Figure 1.2. The diagram shows a processor core which communicates with an internal memory using the internal bus and external memory using external bus. Processor peripherals the UART, Timer and PIC communicate to the processor using the peripheral bus. Timer is an important component in an SoC as it is required to provide clock ticks for the operating system. Programmable Interrupt Controller (PIC) is responsible for forwarding interrupts generated from the peripheral to
CHAPTER 3. ISS INTEGRATION GUIDELINES

the core. UART is mainly required to provide support for system Input/Output within the SoC. As shown in Figure 3.2, a SLDL VP for this SoC, it consists of SLDL model for each of the component. Each individual hardware component is modeled through software. A TLM channel is used to model the bus. A TLM channel models bus communication as a set of transactions, separating the details of implementation from the details of communication. Transaction request takes place by calling an interface function of the bus channel model. As shown in the diagram Figure 3.2, the TLM bus exposes calling interfaces to the HW (External Hardware) component and PE (Processing Element) component.

ISS which are designed as stand alone simulator of the processor are developed in either ’C’ or ’C++’ programming language. These languages do not provide for support for discrete event simulation hence can not be directly utilized within a virtual platform which are build using SLDL which inherently provides support for discrete event simulation time model. In order to utilize the ISS we need to provide with a wrapper which is written in SLDL and allows for running the ISS within the context of system simulation and internally utilizes the ISS functionality. The wrapper provides for connection between the SLDL framework and ISS and allows for communication between them.

![SoC Block Diagram showing only the Essential Components](image)

Figure 3.1: SoC Block Diagram showing only the Essential Components

3.2.1 Execution Model Integration

Traditionally ISS are not constructed for system level design exploration and are supposed to run as stand alone application which take run time arguments and simulate target processor. Utilizing them in system level design exploration was not the goals during designing these application, in order to utilize them in the system level design tools ISS software need modification to all them
to execute with other models in a VP. In order to understand the ISS software in details we first show the internals software execution of a typical ISS.

As shown in Figure 3.3, ISS software works as follows it parses the command line arguments that the user initializes with the ISS, it then initializes the internal data structures which maintain the state of simulation. It then load the binary to an address defined the linker address. The loading creates the base environment for simulation and it simulates as if executing as in a processor. Once the binary is loaded then the ISS executes in a loop fetching, decoding and executing
instruction. After each instruction it accumulates the time of execution of the instruction in internal
time model. Figure 3.4 shows the two main routines one is initialization and the other is run. The
initialization block captures the initialization of data structure based on the configuration parameters
passed to the ISS. The initialization parameter can either passed via command line parameters or
can be passed as function arguments. The other functionality which can be attributed to the initial-
ization block is to load the target binary at the predefined address. The loading of the target binary at
the address creates the base environment for simulation of the processor and can be seen analogous
to loading a binary on the real physical platform. The other main functionality of ISS is to simulate
the instruction which works as follows. Once the target binary is loaded then as a real processor
will execute the ISS runs in a loop fetching, decoding and executing each instruction. In order to
increase the time of simulation instruction decoded are stored in a decoding buffer analogous to an
instruction cache in a real processor. This functionality can be captured in a run function as shown.

![Figure 3.4: ISS Software Main Loop Modified](image)

The main routine of an ISS is divided into two functions. One initialization and another
run function. These function will be able to provide functionality to the SLDL wrapper to start and
simulate ISS for fixed number of cycles. In order to expose these two functions to the VP and allow
for a single binary generation. We need to compile ISS into an archive library and link it with other
CHAPTER 3. ISS INTEGRATION GUIDELINES

component models in VP.

```c
/* Main loop for running the ISS */
void main(void)
{
    /* Create an instance of the simulator */
    ret = sim_init(&IssOut, ISS_PE_NAME, argc, argv);

    /* Register the bus read/write callbacks */
    ret = sim_registerbuscpp_callback(&IssOut, this);

    /* Register the time synchronization callbacks */
    ret = sim_registersynccpp_callback(&IssOut, this, cycles);

    /* Run the simulator */
    ret = sim_run(&IssOut);
}
```

list/genericIntegration.c

The above code expert shows the API's called from the wrapper for software integration. These API's initialize the ISS with the predefined parameters, registers the bus read/write and time synchronization callbacks, and then runs the ISS. These API's need to be supported by the ISS in order for them to be integrated with a system level design tool. In this section we have shown how an ISS can be integrated for execution with a system level design tool.

3.2.2 Time Synchronization

An SLDL VP is composed of multiple components, an VP can support both interrupt and polling based of synchronization options among the components. It also model bus transaction. In order to maintain an execution order there needs to be time synchronization among the components. In case of SLDL VP with an integrated ISS, there are two event clocks; one increments time due to execution of instruction inside ISS, and the other VP clock which increments time annotated for execution of SLDL component models. These two clocks can either be synchronized at every cycle or can be synchronized at a number of cycles referred or quantum. Synchronization at every cycle is the most accurate, but adds simulation overhead and increases the total simulation time. On the other hand synchronization at pre-defined quantum length reduces the accuracy of simulation and at
the same time reducing the simulation time. Hence a trade-off exists between the length of quantum and timing accuracy of simulation.

In order to better understand the discrete event clock synchronization between the ISS clock and VP clock, and study their effects on timing accuracy in an SLDL VP. We take the following simplified code execution sequence in an SoC. As shown in Figure 3.5, the execution sequence is a simple send and receive model. The send code runs on HW, and the receive code is running on a ISS and there is interrupt based synchronization between them. The execution of code on the ISS can be divided into three sections:

1. Uninterrupted Execution: Processor executes without any external access
2. Bus Access: Processor executes and then updates a variable mapped on the Hardware
3. Interrupted Execution: Processes is executing and it receives an interrupt.

Figure 3.5: Sample Code Execution Sequence on SoC showing 3 different types of possible Code Sequence on ISS

The Figure 3.5 shows these 3 segments of execution on the processor. It is an highly simplified example of communication between a processor and another components. In the following sections we show how are the discrete event clock synchronized for the above three identified code sequence executing on the ISS.

3.2.2.1 At Quantum Expiry

When the ISS executes without any external interaction, it accumulates time due to simulation of instructions. This time needs to be periodically synchronized with VP time. For this a update interval or quantum is defined. The figure below shows how the VP time is updated on quantum expiry.
Figure 3.6: Update of VP time due to Execution of Component Models in Quantum

Figure 3.6 shows the increment of VP time, due to execution of its component models mainly Wrapped ISS, PIC and HW. The Wrapped ISS thread is scheduled for execution for a quantum by the SLDL run time environment at virtual time $t_0$. Wrapped ISS causes the VP time to increment from $t_0$ to $t_1$. At $t_1$, again the Wrapped ISS thread is scheduled to run, as there are no other active component inside the VP. The wrapped ISS thread again causes the time to increment from $t_1$ to $t_2$. In this figure we have see how the wrapped ISS, increments the VP time due to ISS execution. In the figure Figure 3.6, we show how the same update of time co-relates to real time.

Figure 3.7: Time Synchronization at Quantum Expiration

Figure 3.7 shows the virtual time on the y-axis and real clock time on x-axis. ISS time is indicate by the blue line, and VP time is indicated by brown line. Continuous line means that the thread is executing and dotted line indicates non-increment of time for the particular thread. At real time 1 sec, ISS thread is scheduled for execution. ISS executes and increases the ISS time,
indicated by the continuous blue line, during this time VP time remains consonant indicated by the
dotted dark brown line. At real time 2 sec, ISS finishes execution for its quantum. Now VP time
is incremented from \( t_0 \) to \( t_1 \). In the figure the ISS time is shown to increase linearly with the host
time, but in reality the time increments in step much smaller the ISS execution steps. Again from
real time 2 sec to 3 sec, the VP time is constant but the ISS time increases due to simulation of
instruction and at real time 3 sec the virtual time is increased from \( t_1 \) to \( t_2 \). In the above sequence of
execution, we have simplified the executions where only the ISS the active component executing
inside the VP. If there are other active components executing inside the ISS, they will also cause the
VP time to increment. In the next section, we see how VP time updates when there are other active
component inside the VP.

### 3.2.2.2 Synchronization at Bus Access

When the ISS executes an external bus access, the two clocks needs to be synchronized
before the bus access is made. This is because the ISS model executes ahead of other component
models inside the VP, so if the time is not synchronized the bus access will occur on the component
model before the virtual time it has to occur. In the section we show how this update happens.

![Figure 3.8: Execution Sequence of the Specification till Bus Access](image)

26
CHAPTER 3. ISS INTEGRATION GUIDELINES

Figure 3.8 shows the increment of VP time, due to execution of its component models mainly Wrapped ISS, PIC and HW. The Wrapped ISS thread is scheduled for execution for a quantum by the SLDL run time environment at virtual time \( t_2 \). With the course of simulation of instruction, Wrapped ISS generates a bus access at time \( t_3 \). At \( t_3 \), the bus component model executes and increments the VP time from \( t_3 \) to \( t_4 \). At \( t_4 \), again the Wrapped ISS thread is scheduled to run, as there are no other active component inside the VP. The wrapped ISS thread again causes the time to increment from \( t_4 \) to \( t_5 \). In this figure we have see how the wrapped ISS, increments the VP time due to ISS execution and also at bus access. In the figure Figure 3.9 we show how the same update of time co-relates to real time.

![Figure 3.9: Time Synchronization at Bus Access](image)

Figure 3.9 shows the virtual time on the y-axis and real clock time on x-axis. ISS time is indicate by the blue line, and VP time is indicated by brown line. Continuous line means that the update of time continuously and dotted line indicates non-increment of time. At real time 3 sec, ISS thread is scheduled for execution. ISS executes and increases the ISS time, indicated by the continuous blue line, during this time VP time remains consonant indicated by the dotted dark brown line. At real time 3.5 sec, ISS generates a bus access within its execution quantum. Now VP time is incremented from \( t_2 \) to \( t_3 \). After the VP time is synchronized with ISS time at \( t_3 \), the bus model is scheduled for execution at real time 3.5 and it also increment the VP time as shown in the figure by the small steps of increments. Again at real time 4 sec, the ISS is scheduled for execution and it accumulate time due to simulation of instruction and at real time 5 sec the virtual time is increased from \( t_4 \) to \( t_5 \). In this section we have shown how the update of VP time happens if
there are 2 active component inside the VP.

3.2.2.3 Synchronization with Interrupts

The previous two sections have shown how time synchronization between ISS time and SLDL time happen at the boundary of ISS execution quantum and at bus accesses. During the time the SLDL wrapper executes an SLDL waitfor, other behaviors in the SLDL can be scheduled. During the their execution, interrupts maybe generated which have to be forwarded to the ISS itself. This section discusses how interrupts are forwarded and their influence onto the time synchronization.

Figure 3.10: Execution Sequence at Interrupt Propagation

Figure [3.10] shows the execution sequence with a interrupt which happens at virtual time $t_6$. ISS thread is scheduled by the scheduler at virtual time $t_5$, for a quantum which $(t - 7 - t_5)$. ISS
accumulates time due to the execution and then synchronizes time. SLDL run time environment schedules at time $t_6$ HW thread for execution. The HW thread generates an interrupt, which wakes up the PIC thread. SLDL run time environment then schedules the PIC thread which then forwards the interrupt to ISS. The interrupt is seen by the ISS only at $t_7$, and not at $t_6$ when it was generated. This difference of virtual time is termed as quantum latency. The quantum latency depends on the quantum length. Larger the quantum length, greater will be the latency as ISS will be further ahead of other component models in the VP. In the figure Figure 3.11 we show how the same update of time co-relates to real time.

Figure 3.11: Time Synchronization at Interrupt

Figure 3.11 shows the virtual time on the y-axis and real clock time on x-axis. ISS time is indicate by the blue line, and VP time is indicated by brown line. Continuous line means that the update of time continuously and dotted line indicates non-increment of time. At real time 5 sec, ISS thread is scheduled for execution. ISS executes and increases the ISS time, indicated by the continuous blue line, during this time VP time remains constant indicated by the dotted dark brown line. At real time 6 sec, ISS completes executing for a quantum, and the VP time is incremented from $t_5$ to $t_6$. At virtual time $t_6$ and real time 6 sec, HW thread is ready for execution and it generates an interrupt. In an idle scenario, the interrupt generated at virtual time $t_6$, should be serviced by the ISS at $t_6$, but ISS due to execution in quantum is at $t_7$. This difference of HW generating interrupt
The above code excerpt shows the API’s in the SLDL wrapper for time synchronization. As shown earlier time synchronization among components in an SoC can happen at two instances namely at quantum expiration, at bus access. In order to support time synchronization at quantum expiration we need a quantum for which the ISS runs before it synchronizes with other components in an SoC. This quantum is predefined at the time of initialization of ISS and a callback functions is registered which is invoked each time the ISS runs for the predefined quantum. In case of time synchronization at bus access which is invoked from the bus read and bus write callbacks. The bus access need to be made after the time synchronization is done and if there are any pending interrupt it needs to prorogated before the bus access is made. This needs to be done in order for the bus
access to be made in the correct time order. At each synchronization point either at bus access or at quantum expiry the function simulatedtime_update is called. This function updates the VP time, by the time ISS has executed in the time quantum.

### 3.2.3 Bus System Integration

Figure 3.1 shows the different types of bus system in an SoC. The processor connects with an internal memory on the internal memory bus. The internal memory bus connects all the internal components with the core. It provides for dedicated communication channel between the core and the internal memory. The peripheral bus connects the external peripheral with the core such as timer, PIC, and UART. Another type of bus system supported in an SoC is external bus. The external bus provides for memory expansion and can also be used for communication with other dedicated processing units such as an hardware accelerator. Each types of bus system has its own standard protocol, and each component when connected on the bus system is required to communicate on that particular bus protocol.

In case of SLDL VP, as shown in Figure 3.2, the bus is modelled using a TLM channel model. A TLM channel models bus communication as a set of transactions, separating the details of implementation from the details of communication. Transaction request takes place by calling an interface function of the bus channel model. Bus access inside an ISS can be modelled by a set of data structures. There data structures are updated each time a processor tries to access a memory location. The SLDL wrapper needs to intercept the bus access inside the ISS and then call the TLM bus interface. In order to intercept and forward this bus transaction from the ISS to SLDL VP, SLDL wrapper utilizes callback function. The callback functions are registered during the initialization of ISS by the SLDL wrapper. This call back functions is invoked when a bus access happens inside the ISS. The callback functions forwards the current transaction on the TLM channel model by calling the interface with the details of the current transaction such as memory transfer size and address.

Figure 3.12 shows a SLDL VP with a Core ISS. This type of ISS only provides for core support which simulates the instruction and uses a modeled internal memory for fetching the code for execution. This type of ISS are mainly developed with goal to simulate functional behaviour of instructions. To utilize core ISS in a VP supporting SLDL models of timer, PIC and a bus model to connect these peripheral with the core needs to be developed and integrated. A processor specific external bus TLM model and peripheral bus TLM model also needs to be developed in order to model communication with the peripheral and external component. The SLDL wrapper in this case
has to intercept both the peripheral bus access and the external bus access from the ISS and then forward them to the respective TLM bus models.

Figure 3.13 shows a SLDL VP with a System ISS. This type of ISS provides peripheral model support along with the core. It also models the internal communication among them. This type of ISS is developed with goal to simulate the processor with supporting peripheral models. In this case a processor specific external bus model needs to be developed. The SLDL wrapper in this case has to intercept only the external bus access from the ISS and then forward them to TLM bus model.

```c
1 // CallBack registration function
2 int sim_registerbus_callback (tIss *pSGdbSim, tCbReadExt read_callback,
3                              tCbWriteExt write_callback, void *pIFC);
```

32
CHAPTER 3. ISS INTEGRATION GUIDELINES

```c
/* bus write callback */
void busWrite(unsigned int data, unsigned long address, unsigned int wr)
{
    /* Update the simulation time */
    synchronization_cleanup(); //

    switch(wr)
    {
    case 4: // write word
    {
        myaddress = address & 0xFFFFFFFFCu;
        ifDbgBus_printf("%s : Write myword to addr = %x, data = %x\n", __FILE__, (int)myaddress, myword);
        bus.StoreWord(myaddress[31:2], myword);
    }
    break;
        case 1: // write byte
        ....
        ....
        break;
    case 2: // write halfword
        ....
        ....
        break;
    }
}

/* bus read callback */
void busRead(void *value, unsigned long address, unsigned int rd)
{
    /* Update the simulation time */
    synchronization_cleanup();

    switch(rd)
    {
    case 4: // read word
    {
        myaddress = address & 0xFFFFFFFFCu;
        word = AhbVect32ToBfinInt(bus.LoadWord(address[31:2]));
        *(unsigned int*)value = word;
    }
    ....
    ....
    ....
    break;
    }
}
```
The above code excerpt shows the API’s for bus system integration. TLM bus channel is connected with the SLDL wrapper through a half channel. In order to get the bus access from the ISS which is written either in C or C++, we register bus read and write callback function from the wrapper at the time of initializing the ISS. These callback functions are invoked when a bus access happens inside the ISS. As the bus access can happen for different data size, we transfer data in this callback functions based on the size of access made in the bus. The above excerpt show the registration functions which is called along with the ISS initialization function it registers the bus callbacks. The other API’s are bus read/write callbacks which are invoked when a bus access happens inside the ISS. In this callback function we first check for the size of data accessed on the bus and then we store it in the TLM channel interfaces for the particular modelled bus.

In this section we have shown how the SLDL wrapper provides for data communication between the ISS and other SLDL component model in SLDL VP. I have also shown that SLDL wrapper needs to intercept peripheral and an external bus access in case of core ISS and only external bus access in case of system ISS.

### 3.2.4 Interrupt Propagation

An interrupt is an asynchronous signal which can cause an interruption to the execution on a processor. Interrupts based on there generating source can be classified as either software
interrupts or hardware interrupts. Software interrupts are caused by some exceptional condition or by execution of interrupts generating specific instructions they are handled on the processor itself. Hardware interrupts are asynchronous hardware signals caused by external peripherals. They are generated when a programmed condition happens on the peripheral. For example a timer can be programmed to generate an interrupt when the count reaches zero. Processors have specific pins to receive an interrupt request. Each interrupt pin in a processor can either be shared between interrupt sources or can be dedicated to a particular source. Due to the limitations of the number of pins on the processor each individual peripheral may not be assigned a unique pin on the processor. PIC are used as intermediary device to solve this problem, they combine multiple sources of interrupts on-to a single or multiple processor pins. A PIC contain a set of registers which allow for programming the priority of the interrupts, it also latches on to the interrupt and stores it even if the generating source clears the interrupt. Interrupts offer an low latency of operation and low overhead of processing as compared to the polling mode in which the processor runs in a busy loop.

In case of SLDL VP, interrupts lines are modelled using shared variable among the component models, and interrupts are triggered high or low by updating the variable. The variable is configured as out in case of interrupt source model and is configured as input in case of the interrupt servicing model. For example in the Figure 3.14, the interrupt line between the HW and the PIC is modelled using a shared variable between the HW and PIC SLDL model. The variable is configured as input in PIC model and as output in the HW model. In order to generate a interrupt from the HW model, it toggles a values on the variable and seen by the PIC model and it model the subsequent interrupt behaviour.

**Figure 3.14:** Core ISS VP depicting the Interrupt Integration

Figure 3.14 shows the interrupt connection in case of a core ISS. The SLDL wrapper
in this case needs to share a variable with a PIC model and a change in the variable needs to be forwarded to the core in order for the interrupts to be recognized by the ISS.

Figure 3.15 shows the interrupt connection in case of a system ISS. The SLDL wrapper in this case needs to share a variable with a HW model and a change in the variable needs to be forwarded to the PIC model inside the ISS in order for the interrupts to be recognized by the core. The interrupt propagation between the PIC and the processor core is internally modelled by the processor.

In this section we have shown how the SLDL wrapper provides for interrupt propagation capabilities between the ISS and other SLDL component model in SLDL VP. I have also shown the difference ways the SLDL wrapper needs to connect either to an external PIC Model or an external HW model in case of core ISS and system ISS respectively.

```c
/* Clear the interrupts line on bfinproxy connected to gpio on the ISS */
void pinLow(unsigned int intNr) {
    sim_interrupts(&IssOut,intNr,0);
}

/* Set the interrupts line on bfinproxy connected to gpio on the ISS */
void pinHigh(unsigned int intNr) {
    sim_interrupts(&IssOut,intNr,1);
}
```

list/genericIntegration.c
CHAPTER 3. ISS INTEGRATION GUIDELINES

The above code excerpt shows the API’s in the wrapper for interrupt integration. The ISS in order to allow for interrupts based communication need to support these API’s which allow for interrupt to be set and cleared on a particular pin. The PIC model which is wrapped under an SLDL Wrapper extends the interrupt capability to other component in an SoC.

3.2.5 Support Debugging Environment on ISS

In order to develop the software on a processor a debugging environment is desirable. Same is the case of VP where there are processing element. The debugging support on the processor can be provided through either using remote server protocol or local debug session. A remote protocol works by running a server program along with the target software. It allows for client to connect to it and transfer commands between the client and the server using remote server protocol. In case of local debug session the client and the host are on the same machine. Cross platform debugging is provided using the GDB remote server protocol, as the clients can be running on any architecture and the server runs on the target processor. In case of VP same support is desired as we have a different simulation host environment and we have a target processor which is being simulated using the VP. In order to start ISS in a debug mode, there are certain command line options. These command line options need to be passed to the ISS during the ISS initialization stage, in order to debug the ISS inside the SLDL VP.

3.2.6 Support Profiling Execution on ISS

In order to provide profiling result of the software execution on the ISS. ISS provides with a level of runtime environment setting and command line options. The level of profiling support depends on the ISS model. General target software profile support provided by the ISS are tracing instructions, tracing hardware events and to capture total simulation time. In order to enable the profile support in the ISS, profiling command line options have to phased to the ISS during the initialization stage options.

In this chapter we have introduced the ISS integration SLDL wrapper and showed the API’s that need to be supported by the ISS in order to allow them for to be integrated with a system level design tool. The integration of ISS using this API’s with tools supporting is agnostic to type of SLDL languages. In case of SystemC these API’s need to be encapsulated in a module, and in case of SpecC the need to be encapsulated in a behavior. The way these API’s are supported by the ISS depends on the organization of the software code of ISS.
CHAPTER 3. ISS INTEGRATION GUIDELINES

ISS like the OVP [12] provide the integration API’s mentioned in this section and can be directly used, while in some cases like the GDB ISS [8] or QEMU [22] the integration API’s are not provided. In the cases when these API’s are not provided we need to modify the ISS software in order for them to expose these API’s. We in the next chapter show how this is done for [8]. In order to verify the integration of ISS with an VP, we propose certain basic test cases which validates that newly integrated ISS is able to run within the VP these test scenarios are explained in the appendix of the thesis.
Chapter 4

Integrating Blackfin GDB ISS

4.1 Introduction

In this chapter we show a sample integration of Blackfin family processor ISS available within the GNU tool chain. We integrate this ISS into a system level design tool System on Chip Environment (SCE) following the guidelines introduced in the chapter 3. The GDB provides a host of simulators are available for ARM, Blackfin, D10V, D30V, FR30, H8/300, H8/500, i960, M32R, MIPS, MN10200, MN10300, PowerPC, SH, Sparc, V850, W65, and Z8000. These simulators were developed to provide an alternative to debug the program in case of non-availability of hardware. The stage of completion and accuracy of these models have to be individually confirmed. Nonetheless the GDB framework provides for development of any processor simulator. The support for Blackfin family was recently added and it is fairly accurate as per our experiment. Please see section 5.1 for detailed analysis and results. The GDB Blackfin ISS provides functional accurate. It is however not cycle accurate. This makes the GDB Blackfin ISS usable for functional validation and software development in a VP, but cannot be used for cycle accurate estimation. In the next section we show how do we make the Blackfin family processor simulator under GDB satisfy the requirements as laid in the chapter 3.

4.2 GDBISS Integration

The Blackfin family of processors are specifically designed to meet the requirements for low-power embedded system. They support a rich set of peripherals. For example timers, real-time clock, TWI, Ethernet MAC (ADSP-BF527), USB 2.0, GPIOs, UARTs, SPORTs, PPI, watchdog
CHAPTER 4. INTEGRATING BLACKFIN GDB ISS

timer, and SPI. The Blackfin ISS available under the GDB framework also contains a model of these peripherals along with the core. As the simulator supports the peripheral model along with the core model it is considered in the category of an ISS with system simulation. Figure 4.1 shows a simplified model of the component modelled in GDB ISS. While GDB ISS supports a rich set of peripherals along with the core, in our diagram we only show those models which were used in the integration of ISS.

![Figure 4.1: Simplified Blackfin GDB ISS block diagram](image)

Integration of system ISS with a system level design tool of poses a lot of different challenges as compared to integration of only a core simulator ISS. As system ISS have peripheral models along with the core, which is well integrated into there simulation framework and separating them from the simulation framework will effect the entire simulation. For the case of GDB ISS of the blackfin processor, the System ISS contains a own discrete event simulation environment with own concept of time, events and concurrency. Hence in order to integrate the GDB ISS without effecting the entire simulation functionality, we decided to extend the GDB ISS with a proxy peripheral. This proxy peripheral is modelled as GDB ISS modelled peripheral. This proxy peripheral runs under the GDB simulation framework and is responsible for synchronizing time with the SLDL VP, forwarding bus access and support interrupt propagation from the SLDL VP to the ISS. In order to provide support for time synchronization we use the proxy peripheral to schedule event periodically inside the ISS. This periodic event on expiry trigger a callback in SLDL wrapper which allows for time to be synchronized between the SLDL VP and ISS. In order to provide bus access capability we connect the proxy peripheral on the modelled asynchronous memory bank 0-3 inside the GDB ISS. Any read or write access on this memory bank is forwarded to SLDL VP through callback functions. Last in order to provide for interrupt prorogation capability we connect the proxy peripheral to the GPIO peripheral model inside the ISS. In the following section we show in details
CHAPTER 4. INTEGRATING BLACKFIN GDB ISS

how we integrated the BF-527 GDB ISS model with [SCE] System Level Design Tool following the guidelines outlined in chapter [3]. For this experiments, we selected the SpecC SLDL for modelling of virtual platforms and integrate the GDB ISS into an SLDL model. In result of our integration, the GDB ISS becomes available in the System-on-Chip Environment (SCE) which is used for automatic construction of virtual platforms.

4.2.1 Execution Model Integration

GDB ISS is not designed for integration with a SLDL VP. GDB ISS is usually executed inside the GDB, but also available as a stand alone program "bfin-elf-run". In order to integrate into the SLDL model, the stand alone binary has to be broken up in to library functions, when then can be called inside the SLDL wrapper. The library is then linked with the VP in order to generate the final executable binary.

```c
/* Create an instance of the simulator */
int sim_init (tGdbBfinIss *pSGdbSim, char *str, int argc, char **argv);

/* Run the simulator */
int sim_run (tGdbBfinIss *pSGdbSim);
```

In order to support the initialization, and execution of the GDB ISS from the SLDL wrapper we divide the GDB ISS main loop in two functions sim_init and sim_run. The function sim_init takes in argument as the GDB ISS simulator instance, a set of run time arguments required to initialized the ISS. The function sim_run takes the instance of the GDB ISS. By calling these two functions from the SLDL wrapper we can initialize and run the ISS.

GDB ISS comes with a set of profiling and instrumentation flags which can be enabled at initialization. The most prominent among them is the tracing hardware and profiling instructions, these can be enabled from the integration wrapper and passed along with other command line arguments during the GDB ISS initialization stage. In this section, we have shown how the GDB ISS was converted into a library in order to have the execution of the ISS within the SCE VP. In the following sections we have also shown how we achieved the time synchronization, bus model integration and interrupt prorogation capabilities from the GDB ISS.
CHAPTER 4. INTEGRATING BLACKFIN GDB ISS

4.2.2 Time Synchronization

As discussed earlier in section 3.2.2 in SLDL VP, there are two event clocks, one increments time due to execution of instruction inside ISS, and the other VP clock which increment time over the execution of SLDL component models. These two clocks needs to be synchronized. The synchronization can be done at either every cycle or in number of cycles referred as quantum. We have decided to choose a quantum based approach for the implementation. In order to allow for GDB ISS to be synchronized at every quantum, we utilize the proxy peripheral. The proxy peripheral schedules an event every quantum. On expiry of this event a callback functions is triggered. The callback executes the SLDL \texttt{waitfor} to delay execution of the ISS wrapper behavior according to the time passed in the ISS. This in turn causes the VP time to increment and may suspend the wrapper (if other behaviors are active during the time).

```
/** Function to get the timing information from the gdb simulator */
int sim_gettime (tGdbBfinIss *pSGdbSim, unsigned long long *gdb_time);

typedef void (*tCbSyncExt)(void *ptr, unsigned int cycles, unsigned int gdb_clock,
                              unsigned int sce_clock);
int sim_registerSyncCallback(tGdbBfinIss *pSGdbSim, tCbSyncExt sync_callback,
                              void *pIFC, unsigned int cycles);
```

The function sim_gettime takes the GDB ISS instance and returns the accumulated time inside the ISS. The function sim_registerSyncCallback registers a callback function of type tCbSyncExt to the proxy device. It takes in argument cycle. This cycles refers to the number of cycles ISS executes before the quantum expires inside the ISS. Hence on every quantum expiry a callback function is triggered inside the SLDL wrapper which updates the VP time. In this section we have shown how the time synchronization capability was added to the SLDL wrapper for the GDB ISS.

4.2.3 Bus System

The Blackfin family supports a bus hierarchy system. The GDB ISS replicates the same hierarchy of bus internally. Blackfin memory map as shown in Figure 4.2 provides four asynchronous memory banks of 1 Mega Byte each. We utilize this memory space to connect the proxy peripheral. The peripheral is then able to access any read and write activity on this memory space.
CHAPTER 4. INTEGRATING BLACKFIN GDB ISS

Figure 4.2: Blackfin 52x Memory Map

Figure 4.3 shows the block diagram of the BFIN ISS VP. The block diagram shows the proxy device added to ISS and connected to the modelled external memory interface.

Figure 4.3: BFIN based VP depicting the Bus System Integration

```c
typedef void (*tCbRead_ext)(void *ptr, const void *pSrc, unsigned long addr, int bytes, unsigned int gdb_clock, unsigned int sce_clock);
typedef void (*tCbWrite_ext)(void *ptr, void *pDest, unsigned long addr, int bytes, unsigned int gdb_clock, unsigned int sce_clock);
int sim_register_bus_callback(tGdbBfinIss *pSDbSim, tCbRead_ext read_callback, tCbWrite_ext write_callback, void *pIFC);
```

list/bf52x_gdbsim.h
In order to connect the TLM bus channel interface to the ISS bus model, we utilize the proxy peripheral. As shown in the diagram Figure 4.3, the proxy peripheral is connected to the modelled external memory inside the ISS. Hence access to this memory can be made using the proxy peripheral. Next we register read and write callback functions inside the proxy peripheral during the initialization stage using the API’s sim_registerbus_callback. These callback functions (tCbRead_ext, tCbWrite_ext) execute in the SLDL wrapper and call the bus interface inside the VP. In this way we are able to connect the bus model in the ISS with the TLM bus model.

4.2.4 Interrupt Propagation

In order to be able to forward an interrupt from external device to the core we utilize the proxy peripheral inside the ISS. The proxy peripheral is connected to GPIO port F pin 0-15 as shown in Figure 4.4. The proxy devices runs in the context of the ISS and forwards the interrupt to the GPIO port F. The GPIO Port F pins are configured as inputs, and interrupts are enabled at the GPIO port F in order to allow the interrupts from the proxy to be processed at the core.

![Figure 4.4: BFIN based VP depicting the Interrupt Connections](image)

```c
/* Function to set and clean interrupts level (1=set 0=clear */
int sim_interrupts(tGdbBfinIss *pGdbSim ,int pin ,int level);
```

In order to generate interrupt at the proxy device, we expose an API sim_interrupts to the SLDL wrapper. This API takes in arguments the instance of the GDB ISS inside the SLDL wrapper, the pin at which the interrupts is to be triggered and the third argument being the interrupt level. By calling this API from the SLDL wrapper, it is able to generate interrupt inside the ISS.
**CHAPTER 4. INTEGRATING BLACKFIN GDB ISS**

Figure 4.5: Interrupt Prorogation Sequence Inside the VP (When quantum ISS then width of interrupt)

Figure [4.5] shows the interrupt propagation sequence from the external hardware to ISS. The interrupt is generated at the HW which is then forwarded to the proxy device. The proxy devices pins are connected to GPIO pins. The GPIO pins configured to generate an interrupt on rising edge. Any interrupt at the GPIO level is forwarded through the SEC to the Core for further processing. Interrupts generated at the hardware are supposed to be captured by the ISS wrapper and then forwarded to the core for further processing. This type of connection works reliably in case of lock step synchronization approach, because all the components are synchronized every clock and any signal can be forwarded to the components in each cycle.

Interrupt propagation into a system ISS in which the PIC (or our GPIO) is inside the ISS poses challenges for integration. The challenges stem from the limited visibility in time of the ISS. SLDL VP changes are only visible in the ISS at synchronization points. In results, a short pulse in the SLDL VP may not be visible in the ISS. In order to illustrate this let us see the high level code, executing on the HW and ISS as shown in Figure 4.6. As shown, the hardware generating an interrupt is simulated by the hardware behavior in the SLDL toggling a signal HINT0. This signal represents a interrupt line. The interrupt signal is kept high by the HW component for a time $H_{time}$ and after that HW component toggles back as shown in Figure 4.7. With the SLDL VP
any component waiting for the change of the signal will be immediately scheduled. However the
ISS executes in its own time, and only synchronizes at coarse points. Hence the ISS will only see
the pulse if it synchronizes right when the pulse happens. In all other events it will never see the
pulse. As shown in Figure 4.7 ISS scheduled to execute for $n_{time}$ does not see the interrupt high
when it resumes execution. The coarse synchronization points in my implementation happen at
two instance, one at regular time quantum expiry and the other at bus access. The bus access are
unpredictable as they depend on the software running on the ISS. The regular time quantum expiry
is determined by the pre-decided quantum length. Hence there is no guarantee that the ISS will be
scheduled at the time a change of pulse happens at the HW component inside the SLDL.

![Figure 4.6: High level Code Sequence Execution on Hardware and ISS](image)

In order to make the interrupt generated at the HW component visible to the ISS, we
add an additional latch model inside the VP. This latch model, runs as shown in Figure 4.8 runs
concurrently with the hardware and ISS model, and waits on signal change by the HW model.
Hence is immediately scheduled by the SLDL run time environment on change of signal. This latch
model stores the interrupt request from the HW component model. This allows the interrupt to be seen by the ISS when the SLDL waitfor has finished and the ISS continues execution. This makes the interrupt propagation from the HW component to the ISS.

Figure 4.8: High Level Code Sequence Execution on Hardware, Latch and ISS

Figure 4.9 shows how the additional latch model forwards the interrupt to the wrapped ISS. The Latch behavior which is waiting on a signal change from the HW model, is immediately scheduled when the change happens. The latch stores on the signal state as high till the time ISS behaviour is scheduled for execution by the SLDL run time environment.

Figure 4.9: Execution Sequence of Hardware, ISS and Latch showing Interrupts being captured

Figure 4.10 shows the block diagram of the BFIN VP with an additional latch model. The additional latch model running concurrent with the SLDL wrapper allows for interrupts generated at the HW component model to be propagated to the ISS.

4.2.5 Support Debugging Environment on GDB ISS

In order to have support for target software development using the GDB ISS, the availability of debugging support is important. GDB ISS being part of GDB framework had debugging support
but through a local host. Local host debugging is suitable for a stand-alone simulator but in order to effectively debug the program running on the VP remote debugging support is required. We added the remote debugging support on BFIN GDB ISS using the GDB remote server protocol support. In order to add the remote debugging support for GDB server was added. The GDB server waits for command from the GDB client. When it receives a command it interprets and forwards to the GDB ISS. In order for GDB ISS to be run under the supervision of GDB server, the ISS is initialized in a debug environment. In order to support the debug functionality of viewing memory and registers, the GDB server API’s were connected to GDB simulation framework API’s.
Chapter 5

Experimental Results

In this chapter we show our experimental results. The experimental section is divided into three main aspects. Our first experiment is to analyze accuracy of the integrated BF-527 GDB ISS. In this experiment we aim to estimate the cycle accurateness of a functional accurate of an ISS. Our next set of experiments aim to study the overall effect of size of quantum on the simulation time and interrupt latency. This analysis helps us understand the trade-off between simulation speed on one hand and accuracy on the other. Finally we perform a design space exploration of a mp3 decoder application using the SCE tool using the newly intergraded BF-527 and ARM92JES. In the following sections we first introduce and then discuss in details the experimental setup and the discuss results obtained for each experiment individually.

5.1 Accuracy of BF527 GDB ISS

5.1.1 Introduction

The aim of this experiment is to measure the accuracy of the instruction accurate Blackfin ISS under the GDB. The ISS is an instruction accurate and is not cycle accurate but it provides with a approximate number of processor cycles required for simulating each instruction. We have used the same ISS global time in our integration for time synchronization with the VP clock.

When executing on an actual processor, the Blackfin processor provides support for measuring the performance of the processor through cycle counter registers. These registers can be read at any given point during execution to get the the count of core clock cycles. For analyzing the accuracy of ISS, we select a set of MiBench and DSPstone benchmarks. We select this set
CHAPTER 5. EXPERIMENTAL RESULTS

of benchmarks as Blackfin is a DSP processor widely used in the area of control applications. We execute this set of benchmarks on both the hardware and ISS and then estimate the inaccuracy of GDB ISS.

5.1.2 Experimental Setup

In order to get the core cycle count from the ISS and from the HW, we run the same binary on both the hardware and the ISS. We extend the benchmarks to print core cycle count at the end of the execution both in the case of HW and as well in the case of ISS. All benchmarks where compiled with a bfin-elf-gcc version 4.3.5 cross compiler for Blackfin and the simulation were run on Intel(R) Core(TM)2 Duo CPU E8500 @ 3.16GHz. As physical hardware, we use the TLL6527 [13], developed by The Learning Labs. The platform is a versatile platform containing a BF-527 processor and XILINX FPGA. In order to calculate the percentage inaccuracy of ISS, we use the below formula. The formula first calculates the absolute difference of cycle count between the HW and the ISS and in order to calculate the percentage inaccuracy it divides by the Hardware cycle count.

\[
\text{ISS\ %\ Inaccuracy} = \frac{\text{abs(Hardware Cycle Count - ISS Cycle Count)}}{\text{Hardware Cycle Count}} \times 100
\]

5.1.3 Analysis and Results

BF-527 GDB ISS is a functional accurate it does not contain the model the multi level of memory, pipeline as in a real processor. The core cycle incurred for each instruction is statically added for each instruction program. This is not based on real execution inside the pipeline. In result, a pipeline stall in case of physical processor will increase the cycles of the physical processor. The cycle count inside the ISS, on the other hand, remains as if no stalls have occurred. The ISS also does not model the memory access latency. In result, a binary executed from SRAM or SDRAM will yield the same core cycle count on the ISS. However, there will be additional core cycles in case of a physical platform. With this background the numbers of the core cycle counts for the benchmarks will be higher in case of physical processor compared to the ISS.

Table 5.1 and Figure 5.1 shows the result in tabular and graphical form respectively. As we see from the table 5.1 that for all the benchmarks expect the fib, gamma the core cycle count from the Hardware is higher then the ISS. In case of fib and gamma ISS reports a higher number of core cycle count number then the hardware. It might be due to effect of inaccurate number of core cycles accounted inside the ISS for some instructions which are more prominent in these two benchmark as
### CHAPTER 5. EXPERIMENTAL RESULTS

<table>
<thead>
<tr>
<th>Serial No</th>
<th>Benchmark</th>
<th>Hardware</th>
<th>ISS</th>
<th>ISS % Inaccuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>dhry</td>
<td>82,398,292</td>
<td>66,150,799</td>
<td>19.72</td>
</tr>
<tr>
<td>2</td>
<td>AES</td>
<td>129,888</td>
<td>87,027</td>
<td>33.00</td>
</tr>
<tr>
<td>3</td>
<td>crc</td>
<td>53,621,931</td>
<td>42,667,896</td>
<td>20.43</td>
</tr>
<tr>
<td>4</td>
<td>engine</td>
<td>88,743,268</td>
<td>76,638,313</td>
<td>13.64</td>
</tr>
<tr>
<td>5</td>
<td>fir2</td>
<td>58,847,771</td>
<td>40,290,499</td>
<td>31.53</td>
</tr>
<tr>
<td>6</td>
<td>hanoi</td>
<td>1,062,983</td>
<td>958,393</td>
<td>9.84</td>
</tr>
<tr>
<td>7</td>
<td>pocsag</td>
<td>18,454,219</td>
<td>15,312,066</td>
<td>17.03</td>
</tr>
<tr>
<td>8</td>
<td>bcnt</td>
<td>28,599,192</td>
<td>19,850,025</td>
<td>30.59</td>
</tr>
<tr>
<td>9</td>
<td>des</td>
<td>14,832,190</td>
<td>11,161,976</td>
<td>24.74</td>
</tr>
<tr>
<td>10</td>
<td>fft</td>
<td>72,823,944</td>
<td>74,157,572</td>
<td>1.83</td>
</tr>
<tr>
<td>11</td>
<td>queens</td>
<td>23,920,182</td>
<td>17,891,302</td>
<td>25.20</td>
</tr>
<tr>
<td>12</td>
<td>blit</td>
<td>89,047,121</td>
<td>74,321,716</td>
<td>16.54</td>
</tr>
<tr>
<td>13</td>
<td>fib</td>
<td>63,235,422</td>
<td>93,188,426</td>
<td>47.37</td>
</tr>
<tr>
<td>14</td>
<td>gamma</td>
<td>89,963,117</td>
<td>99,059,995</td>
<td>10.11</td>
</tr>
<tr>
<td>15</td>
<td>matrix</td>
<td>196,181,738</td>
<td>115,339,479</td>
<td>41.21</td>
</tr>
<tr>
<td>16</td>
<td>mib_off_strsearch</td>
<td>29,153,300</td>
<td>23,717,810</td>
<td>18.64</td>
</tr>
<tr>
<td>17</td>
<td>quicksort</td>
<td>190,515</td>
<td>153,717</td>
<td>19.32</td>
</tr>
<tr>
<td>18</td>
<td>mib_auto_basicmath</td>
<td>55,309,848</td>
<td>51,415,108</td>
<td>7.04</td>
</tr>
<tr>
<td>19</td>
<td>whets</td>
<td>33,197,891</td>
<td>30,139,002</td>
<td>9.21</td>
</tr>
<tr>
<td>20</td>
<td>heapsort</td>
<td>157,844,656</td>
<td>120,984,013</td>
<td>23.35</td>
</tr>
<tr>
<td>21</td>
<td>jpeg</td>
<td>14,044,025</td>
<td>10,908,968</td>
<td>22.32</td>
</tr>
<tr>
<td>22</td>
<td>mdrAll</td>
<td>10,732,424</td>
<td>9,671,642</td>
<td>9.88</td>
</tr>
<tr>
<td>23</td>
<td>v42</td>
<td>49,671,832</td>
<td>39,557,748</td>
<td>20.36</td>
</tr>
</tbody>
</table>

Table 5.1: Core Clock Cycles on BF-527 HW and ISS for Benchmark

![ISS Inaccuracy % Graph](image)

Figure 5.1: Graph for Percentage Inaccuracy of ISS
CHAPTER 5. EXPERIMENTAL RESULTS

compared to others. Figure 5.1 shows that for some benchmarks like `bcnt`, `matrix` the % inaccuracy is around 40%, for some benchmarks like `dhrystone`, `engine`, `jpeg`, `v42`, `quicksort` the percentage inaccuracy is around 20%, and for some benchmarks like `whets`, `hanoi`, `fft`, `mib auto basicmath`, `mdr_all` the percentage inaccuracy is around 10%. This variation inaccuracy is expected due to the missing memory and pipeline model inside the ISS. One of the effect is the benchmarks which cause a high number of memory stalls in case of physical processor will have higher core cycle counts as compared to ISS, this is due to the fact that ISS does not have a memory model.

In order to further analysis we show the results of percentage of memory stalls in core clock cycles for some benchmarks in Table 5.2. The tables shows the for `bcnt`, `matrix` the percentage inaccuracy is higher at 30.59% and 41.21% respectively and the percentage memory stall is 13.81% 11.02% on the physical hardware. `whets` and `fft` the percentage inaccuracy is lower 9.21% and 1.83% so is the the memory stall cycles is 3.58% and 2.07% respectively. This only shows the effect of the missing memory model and its contribution to the percentage inaccuracy of ISS. Other effects attribute such as pipeline stalls due to data hazards, which is not modelled in the ISS also attributes to the inaccuracy.

<table>
<thead>
<tr>
<th>Serial No</th>
<th>Benchmark</th>
<th>ISS % Inaccuracy</th>
<th>Memory Stalls Cycles</th>
<th>% Memory Stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bcnt</td>
<td>30.59</td>
<td>3,950,659</td>
<td>13.81</td>
</tr>
<tr>
<td>2</td>
<td>matrix</td>
<td>41.21</td>
<td>6,971,275</td>
<td>11.02</td>
</tr>
<tr>
<td>3</td>
<td>blit</td>
<td>16.54</td>
<td>5,255,315</td>
<td>5.90</td>
</tr>
<tr>
<td>4</td>
<td>v42</td>
<td>20.36</td>
<td>3,736,582</td>
<td>7.52</td>
</tr>
<tr>
<td>5</td>
<td>whets</td>
<td>9.21</td>
<td>1,186,880</td>
<td>3.58</td>
</tr>
<tr>
<td>6</td>
<td>fft</td>
<td>1.83</td>
<td>1,507,750</td>
<td>2.07</td>
</tr>
</tbody>
</table>

Table 5.2: Memory Stall numbers for benchmark

In this experiment we have seen that the ISS core cycles numbers are not accurate in comparison to a physical hardware, which is acceptable as the ISS is only instruction accurate. Our experiment shows that the ISS core cycle counts are approximately 20%-40% off from the real hardware. As we have used these core cycle counts to time synchronize the ISS clock with the VP clock, the time delay due to execution on the simulated processor is 20%-40% off from the real SoC.
CHAPTER 5. EXPERIMENTAL RESULTS

5.2 Effect of Quantum-based Time Synchronization on Interrupt Latency

5.2.1 Introduction

In the section 3.2.2.3 it was shown that due to synchronization in quantum, we add an additional interrupt latency. Interrupt latency is defined as the time difference between the time at which an interrupt is generated and the time at which it is being actually serviced. In case of system simulation we have an additional latency due to time synchronization among the components in a simulation, we term this as quantum latency. For the purpose of analysis we define quantum latency as the difference of time between the interrupt being generated and the time at which the interrupt is sent to the processor model. In a simulation environment, interrupts can be forwarded only when the SLDL run time environment suspends the processor model. SLDL run time environment suspends the processor model, at time synchronization points. These time synchronization points in our implementation happen either at periodic quantum expiry or at bus access. Bus access are dependent on the application running on the processor, and the quantum expiry is dependent on quantum length the ISS is scheduled to run by the SLDL run time environment.

A small quantum length will increase the simulation overhead increasing the simulation time at the same time reduce the quantum latency. On the other hand a large quantum will reduce the simulation overhead thereby reducing the simulation time but increase the quantum latency. So a trade-off exists between quantum latency and total simulation time. We in this experiment aim to evaluate the trade-off for an application without bus access, and one with bus access. We also aim to show the effect of quantum expiry based time synchronization on quantum latency and interrupt latency. We also show the combined effect of quantum expiry and bus access based time synchronization on quantum latency and interrupt latency. We use the System on Chip Environment (SCE) tool for generating our experiment. The SCE tool 2.1 automates the process of generating the software code running on processor along with the generation of virtual platform.

5.2.2 Experimental Setup

Figure 5.2 shows the experimental setup for our experiment. It comprises of two behaviors which are refined to execute on a hardware and on processor (BF-527 ISS). The HW behavior is programmed to send an interrupt and then wait on for the response for that interrupt from the processor. On the other hand the processor behaviour is composed of multiple tasks. One task
which is waiting to receive a request from hardware and send a response back. The other is a variable JPEG encoder task. The JPEG encoder is a compute intensive DSP application. The encoder task internally is composed of three main behaviors encoder, stimulus and monitor. The encoder behavior reads a stripe of the bitmap from the stimulus, encodes the JPEG image and write the JPEG image in byte format in the monitor behavior. HW and the response task on processor, runs for 10,000 times in a loop. Thereby generating 10,000 interrupts and responses at random interval of time with a range of 0 cycles to 10*10,000 cycles. We measure the interrupt progression at 3 instances (1) when the interrupt is generated at hardware, (2) when the interrupt is delivered to the ISS for processing and (3) the response (ie. the processor reads data from hardware) received at the hardware. Difference between time stamps 1, and 2 yields the quantum latency and the difference between time stamp 1, and 3 yields the response time which is a summation of quantum latency, interrupt latency at the processor and the latency of bus transfer. The total simulation time is the time when the response of all the 10,000 interrupts generated from the HW behavior is received.

We divide our analysis and the result section in the following two parts. First we vary the application on processer and see the effect on quantum latency and response for a fixed quantum size. Secondly we vary the quantum length and see its effects on quantum latency for a JPEG encoder without bus access and JPEG encoder with bus access running on the processor.

5.2.3 Impact of Quantum Synchronization onto Quantum Latency and Response Time

In this section we show the effect of varying application running on processor on quantum latency and response time. We use the experimental setup above, and vary the task running on the processor. The different applications for this experiment are:

1. Response Task (NOOS): The response task is refined to run on the processor without an
CHAPTER 5. EXPERIMENTAL RESULTS

operating system.

2. OS + Response Task (OS): The response task is refined to run on an RTEMS operating system.

3. OS + Response Task + JPEG encoder Task (LOAD): The response task is refined as running on an RTEMS operating system and another JPEG encoder task is refined to run along along with it. Thereby creating a multitasking environment on the processor. The JPEG encoder task is refined to run the stimulus, encoder and monitor on the processor.

4. OS + Response Task + JPEG encoder Task with bus access (BUS ACCESS): In this application, we refine the JPEG encoder to run on the processor and the stimulus and monitor behavior run on the hardware thereby generating bus access.

In order to analyze the effect of different application on quantum latency and response time we fix the quantum length of the processor at 10,000 cycles. We expect to have less response time from an application running without an OS, then by an application running with an OS. This is due to the fact that with an OS we add the extra overhead of context switch in order to service the response task. Thereby increasing the response time for an application running on an OS. However quantum latency which is defined as the time difference between interrupt generated at the hardware and forwarded to the processor for service, is independent of an application running with an OS and without an OS. The quantum latency should not change with an application running with an OS or without an OS. Quantum latency is however dependent on bus accesses by the application. A bus access by the ISS triggers synchronizing ISS and VP time before performing the bus access. This in turn offers an earlier chance for forwarding interrupts to the ISS, thus should reduce the quantum latency. So we expect to see different trends for an application with bus and with an application without bus access. The response time for these applications is also expected to follow the same trend as in our set up response time is a summation of quantum latency, interrupt latency and bus access delay.

Figure 5.3 shows the observed quantum latency for the four applications described above. Using our set up, we measure the quantum latency for the 10,000 interrupts generated at the hardware. Then we use them to calculate the cumulative probability. In the figures we plot quantum latency in cycles on X-axis using there cumulative probabilities indicated on Y-axis. Figure 5.3 shows the quantum latency for the three applications: NOOS indicated in pink, OS indicated in blue and LOAD indicated in light green. These three applications follows the same trend. This was expected as quantum latency is independent for an application running with or without an OS.
CHAPTER 5. EXPERIMENTAL RESULTS

Figure 5.3: Simulation Latency for ISS @ [Quantum 10000] with different workload

However we see that the application BUSACCESS, shows a significant deviation from others. We see that for this application there is 70% probability that the quantum latency will be less than 300 cycles, for the other three application without bus access is around 7000 cycles.

Figure 5.4: Response Time for ISS @ [Quantum 10000] with different workload

Figure 5.4 shows the observed response time for the four application described above. Using our set up we measure the response time for the 10,000 interrupts generated at the hardware.

56
CHAPTER 5. EXPERIMENTAL RESULTS

Then we use them to calculate the cumulative probability. In the figures we plot response time in cycles on X-axis using there cumulative probabilities indicated on Y-axis. In Figure 5.4 as shown the quantum latency for the three applications OS indicated in blue, LOAD indicated in light green, BUSACCESS indicated in red have the same minimum response time of 2810 cycles, and a maximum of 12540 cycles. However, we see that the for NOOS indicated in pink, shown a minimum latency of 1010 cycles and and maximum of 10650 cycles. This matches the expectation as response time of the application is dependent on whether the application is running or an OS or without an OS. We see different trends for LOAD application indicated in light green, BUSACCESS application indicated in red because the response time is the summation of quantum latency, interrupt latency and bus access delay.

In this section, we have validated that quantum latency varies only for an applications running with bus access and without bus access and is independent whether an application is refined with an operating system or without an operating system. In the next section we evaluate the trade-off between quantum latency and total simulation time.

5.2.4 Impact of Variable Quantum Length on Total Simulation Time and Quantum Latency

In this section we see the effect of varying quantum length on quantum latency and total response time for the LOAD, and BUSACCESS application. We choose these application as we found that quantum latency varies within application having bus access and one not having bus access.

In order to analyze the effect of different quantum length on total simulation time and quantum latency, we vary the quantum length of the processor from 1,000 cycles to 10,000 cycles. We first analyze the effect for LOAD application and then we analyze the effect for the BUSACCESS application. We measure the quantum latency and simulation time at different quantum length. We expect the quantum latency to increase linearly as we increases the quantum length. This is due to reason that as quantum length increases we increase the time after which the with other SLDL component model are synchronized with ISS. The simulation time is expected to reduce as we increase the quantum length. This happens as with increase in quantum length we reduce the number of synchronization points thereby reducing the overhead of simulation on the SLDL run time environment.

Table 5.3 shows the obtained result in tabular form and Figure 5.5 shows the result in
CHAPTER 5. EXPERIMENTAL RESULTS

<table>
<thead>
<tr>
<th>Quantum Size (cycles)</th>
<th>Simulation Time (secs)</th>
<th>50 percentile (cycles)</th>
<th>max (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>32.70</td>
<td>0</td>
<td>4980</td>
</tr>
<tr>
<td>9000</td>
<td>32.84</td>
<td>0</td>
<td>4536</td>
</tr>
<tr>
<td>8000</td>
<td>32.98</td>
<td>0</td>
<td>3908</td>
</tr>
<tr>
<td>7000</td>
<td>33.12</td>
<td>0</td>
<td>3504</td>
</tr>
<tr>
<td>6000</td>
<td>33.24</td>
<td>0</td>
<td>3019</td>
</tr>
<tr>
<td>5000</td>
<td>33.31</td>
<td>0</td>
<td>2505</td>
</tr>
<tr>
<td>4000</td>
<td>33.42</td>
<td>0</td>
<td>2017</td>
</tr>
<tr>
<td>3000</td>
<td>34.16</td>
<td>0</td>
<td>1534</td>
</tr>
<tr>
<td>2000</td>
<td>34.50</td>
<td>0</td>
<td>995</td>
</tr>
<tr>
<td>1000</td>
<td>34.80</td>
<td>0</td>
<td>506</td>
</tr>
</tbody>
</table>

Table 5.3: Quantum Latency and Simulation Time for LOAD application

Figure 5.5: Quantum latency and Simulation time for different quantum length for LOAD application
CHAPTER 5. EXPERIMENTAL RESULTS

graphical form. On X-axis we have quantum size in cycles, on the Y-axis we denote the simulation time in seconds. On the other right hand y-axis we have the quantum latency in cycles. The simulation time plotted in red linearly decreases with increasing quantum. At the same time we see quantum latency linearly increasing with increasing quantum. This matches expectation as with an increasing quantum length we are reducing overhead of synchronization. It can also be concluded that if the users cares less about interrupt accuracy, then a larger quantum length will be sufficient. But if interrupt latency is a major factor, then a smaller quantum meeting the time requirements is needed. In this experiment, we have only seen the effect of regular time synchronization effect on quantum latency and simulation time, in the next experiment we evaluate the effect of regular as well as bus access based time synchronization effect on quantum latency and simulation time.

We now analyze the effect for a BUSACCESS application. The JPEG encoder behavior running on the processor reads a stripe of bitmap image from the stimulus behavior, and write the encode JPEG image in byte format in the monitor behavior. The stimulus and the monitor behavior are executed on Hardware. We measure the quantum latency and simulation time at different quantum length. Compared to the previous experiment However, the total simulation time will be significantly larger due to large image size and additional bus access. The additional bus access are due to the mapping of stimulus and the monitor behavior on the hardware. We increase the encoding delay inside the JPEG encoder behavior thereby increasing simulation time.

<table>
<thead>
<tr>
<th>Quantum (cycles)</th>
<th>Simulation Time (secs)</th>
<th>Quantum Latency(cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>309.00</td>
<td>333</td>
</tr>
<tr>
<td>9000</td>
<td>310.40</td>
<td>319</td>
</tr>
<tr>
<td>8000</td>
<td>311.00</td>
<td>302</td>
</tr>
<tr>
<td>7000</td>
<td>312.51</td>
<td>284</td>
</tr>
<tr>
<td>6000</td>
<td>313.33</td>
<td>262</td>
</tr>
<tr>
<td>5000</td>
<td>314.25</td>
<td>245</td>
</tr>
<tr>
<td>4000</td>
<td>315.00</td>
<td>223</td>
</tr>
<tr>
<td>3000</td>
<td>317.00</td>
<td>205</td>
</tr>
<tr>
<td>2000</td>
<td>318.66</td>
<td>175</td>
</tr>
<tr>
<td>1000</td>
<td>321.31</td>
<td>147</td>
</tr>
</tbody>
</table>

Table 5.4: Table showing Quantum Latency variation over quantum length for BUSACCESS application

Table 5.4 shows the obtained result in tabular form and Figure 5.6 shows the result in graphical form. On X-axis we have quantum size in cycles, on the Y-axis we denote the simulation
time in seconds, and on the other right hand y axis we have the quantum latency in cycles. The simulation time plotted in red linearly decreases with increasing quantum. At the same time, quantum latency linearly increases with increasing quantum. This was expected as with an increasing quantum length we are reducing the overhead of synchronization points and increasing the quantum at which the SLDL components can synchronize. But the effect of change of quantum length is not as pronounced as was seen in the LOAD application. We see the quantum latency for quantum length of 10,000, the quantum latency is reduced from 4980 cycles in case of the LOAD application to 333 cycles in case of BUSACCESS application. Based on the observation it is clear that a trade-off exists between quantum latency and total simulation. However in case of the BUSACCESS application the total simulation time is effected much more by the application running on the hardware and its effect on the part of the application running on SW, that the effect of change of quantum is negligible.
CHAPTER 5. EXPERIMENTAL RESULTS

5.3 Design Space Exploration

5.3.1 Algorithm Under Test

To demonstrate the gained capabilities of integrating the BF-527 ISS into the SCE toolchain, we discuss in this section an example application of mapping an MP3 decoder on BF-527 ISS. MP3 decoder application is a widely used application. The MP3 decoder application has been converted to specc model \(^{[26]}\). In the project the authors utilized the SCE tool. The authors have used the original MAD C code from \(^{[19]}\) which is a new implementation of the ISO/IEC standards.

Figure 5.7 shows the top level of MP3 decoder specification model. At the top-level, the MP3 decoder Main behavior executes the MP3decoder application supplying the parsed command line arguments like input and output file names via interface methods calls and ports. As its testbench setup the stimulus and monitor run concurrently with the actual design. Stimulus and monitor are supplied with names of input and output files to read from and to write. The design communicates with testbench through abstract channels for incoming MP3 bytes (stream_in) outgoing PCM samples (pcm_out) and asynchronous error condition (decode_error). The abstract channels are FIFO queues for buffering of frame data and decoupling of threads in order to improve performance. With the newly added BFIN DSP processor we increase the design space by providing a DSP based processor. We exploration for the different configuration shown below:

1. A pure software solution on the ARM92JES
CHAPTER 5. EXPERIMENTAL RESULTS

2. A pure software solution on the BFIN-527

5.3.2 Test Results

<table>
<thead>
<tr>
<th>Model</th>
<th>LOC</th>
<th>Behaviors</th>
<th>Leaf</th>
<th>Channels</th>
<th>Simulation Time(sec)</th>
<th>Frame Delay(msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifications</td>
<td>14182</td>
<td>46</td>
<td>30</td>
<td>3</td>
<td>0.08</td>
<td>26.12</td>
</tr>
<tr>
<td>SW (ARM)</td>
<td>25270</td>
<td>113</td>
<td>70</td>
<td>54</td>
<td>480.85</td>
<td>24.87</td>
</tr>
<tr>
<td>SW (BFIN)</td>
<td>24915</td>
<td>111</td>
<td>68</td>
<td>53</td>
<td>10131.8</td>
<td>23.17</td>
</tr>
</tbody>
</table>

Table 5.5: MP3 decoder exploration on ARM and BFIN based VP

Table 5.5 shows the results of refining the MP3 decoder application using the SCE tool chain on ARM92JES and BFIN-527. The frame delay in case of using BFIN is less than the ARM. This is due to higher operating frequency of the BFIN-527 which runs at 500MHz as compared to 100MHz of ARM92JES also due to the fact that BFIN-527 is a DSP processor and supports special dsp instruction set. However we see that the simulation time for the ARM92JES based VP is 480.85 sec as compared to BFIN at 10131.8 sec. The ARM92JES ISS is provided by the OVP [12]. The OVP simulator’s uses the dynamic binary translation to simulate instruction set of the target processor. While GDB uses an interpretive based method to simulate the target instruction set, which is more slower than the binary translation method. These results also validate our ISS integration approach, and guideline.
Chapter 6

Conclusion

Embedded system designers today face enormous challenges with the rising software content in modern SoC. With the rising software content the software design cost increases and drives the cost of SoC. New design methodology referred as Electronic System Level (ESL) has been proposed and developed which focuses on higher levels of abstraction to counter these challenges. Development using virtual platforms is one of major contribution of the ESL design methodology. A VP consist of modeled components of SoC and simulates the functionality of SoC. A VP replicates aspects of physical platform and provides for pre-silicon software development, performance optimization and hardware and software configuration. ISS are an integral piece in a processor centric VP as it performs the processor functionality and by that enables early binary validation. Traditionally ISSs have been developed as stand-alone program running independent of a VP environment.

In this thesis we have designed and developed generic guidelines on how to integrate an ISS with SLDL VP environment. These generic guidelines outline the integration of ISS-in four steps. (1) Execution model integration: Integration of the ISS execution model with the VP execution model. (2) Time synchronization: Synchronization of the ISS clocks which increments time due to execution of instruction inside ISS, and the VP clock which increments time annotated for execution of SLDL component models. (3) Bus system integration: Integration of ISS internal bus model, with the SLDL TLM bus model (4) Interrupt Propagation: Propagation of interrupt from the HW SLDL model to the ISS.

We have applied these guidelines when integrating the BF-527 ISS GDB with the SCE ESL environment. Quantum based time synchronization was chosen for the integration. In quantum based time synchronization, ISS is allowed to execute for a quantum (number of cycles) before the ISS clock and VP clock are synchronized. This reduces the total simulation time. We
further studied the effect of quantum based time synchronization on interrupt latency, and found that there is latency added due to synchronization in quantum interval. As the ISS synchronizes in quantum steps, it only sees an interrupt at the end of the quantum. Thus an additional delay is generated between the interrupt generated at the HW and being prorogated to the ISS. We termed this latency as quantum latency. Our experiments shows that the quantum latency increases with increasing quantum length, as with increasing quantum length the time interval at which the ISS sees the event is also increased. The experiments also show that the simulation time decreases with the increasing quantum. Hence a trade-off exists between the quantum latency and total simulation time. However, for application with bus access the quantum latency is reduced. Since ISS time and VP times are synchronized before each access, the ISS detects interrupts earlier (before expiry of the quantum step). Our experiments also show the quantum latency is independent of the application if running entirely on the processor without external bus access.

We have also measured the relative inaccuracy of the GDB ISS using a set of MiBench and DSPstone benchmarks[39] and found the GDB ISS to 20%-40% inaccurate as compared to physical platform. We have also demonstrated the gained capabilities of integrating the BF-527 ISS into the SCE tool-chain, by performing a design space exploration using an MP3 decoder.

Our generalized guidelines in this thesis will aid other researchers in integrating new ISS into SLDL VP and understanding the issues of time synchronization among components in a SLDL VP.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY

Appendix A

Test Cases

In this section we show the test cases for integration. As highlighted in the chapter the ISS needs to be able expose the following capability.

1. Provide capability to connect bus traffic to the processor
2. Provide capability for interrupts propagation from system to processor
3. Provide capability for time synchronization with other peripheral components on the SoC
4. Provide function to initialize and run ISS within the VP
5. Provide features for debugging target application on processor
6. Provide ability to profile instructions and event execution on processor

We have shown in the chapter how these capabilities can be achieved from ISS. In order to verify that the we have been successfully able to obtain these functionality we are proposing the following basic test cases. These test cases are developed in specC and have been tested using System on Chip Environment (SCE). As described in section 2.1 SCE tool, works on the methodology of specify, explore and refine. The designers develop a specification model, which is then refined and explored making design decision at each stage of refinement. In the following sections we show the specification and the architectural model for the test cases.
APPENDIX A. TEST CASES

Figure A.1: Specification of Execution and Time Synchronization Test Case

A.1 Specification

A.1.1 Execution and Time Synchronization Test Case

The purpose of this specification is to check the basic functionality of execution of ISS, with other component model in VP. As shown in Figure A.1 the test case comprises of send and receive behaviour, connected through a single handshake channel. The single handshake channel provides for one-way synchronization with storage for one event. This is a simple send and receive model.

A.1.2 Bus Read and Write Test Case

The purpose of this specification is to check the bus read and write capability of the integrated ISS, with other component model in VP. As shown in Figure A.2 the test case comprises of send and receive behaviour, connected through a single handshake channel and share a data variable to test data transfer cpa. This is a simple send and receive model which shares a data. The send behaviour updates on the shared data variable, and then sends a signal to the receive behavior which reads the data and prints it.
APPENDIX A. TEST CASES

A.2 Architectural Mapping

A.2.1 Execution and Time synchronization Test Case

A.2.1.1 PE based Synchronization Test

Figure A.3 show the architectural mapping for the PE based time synchronization. In this test case we check the basic functionality of ISS being executed within the VP and provide for execution to other component models in the SoC. This is a basic test of integration where we check execution. The expected results for this test case is that the processor executes sends a notification to the hardware, and then hardware executes and terminates simulation.

A.2.1.2 HW based Synchronization with Polling Test

Figure A.4 show the architectural mapping for the HW based synchronization with the processor waiting in busy loop. The aim of the test case is to check the time synchronization between the ISS and other components in a VP.

A.2.1.3 HW Based Synchronization with Interrupts Test

Figure A.5 show the architectural mapping for the HW based synchronization with interrupts test. In this test case we check the capability of the SoC support of interrupts.
APPENDIX A. TEST CASES

Figure A.5: HW based Synchronization with Interrupts based Synchronization

A.2.2 Bus Read and Write Test Case

A.2.2.1 Bus Write Test

Figure A.6: Bus Write Test

Figure [A.6] show the architectural mapping for the bus write test. In this test case we test the functionality of the integrated ISS being able to write to the interconnected bus and the connected component seeing the same value. This test case is intended to test the bus write interconnection and can be extended further to test the various type of data transfer across the bus.

A.2.2.2 Bus Read with Polling Test

Figure A.7: Bus Read with Polling Test

Figure [A.7] show the architectural mapping for the bus read test. In this test case we test the functionality of the integrated ISS being able to read from the interconnected bus and when an
external component writes on the bus. This test case tests the bus read interconnection and can be extended further to test the various type of data transfer across the bus.

A.2.2.3 Bus Read with Interrupt Test

Figure A.8: Bus Read with Interrupt Test

Figure A.8 show the architectural mapping for the bus read test. In this test case we test the functionality of the integrated ISS being able to read from the interconnected bus and when an external component writes on the bus. This test case tests the bus read interconnection and can be extended further to test the various type of data transfer across the bus. In this test case we also check the interrupt propagation capability of the SoC, and test the modelling of interrupts.