Institut für Techniche Informatik

Embedded Systems Department

Universität Stuttgart
Pfaffenwaldring 5B
D - 70569 Stuttgart

Master Project Nr. 3395

Modeling of a Multi-core MicroBlaze System at RTL and TLM Abstraction Levels in SystemC

Karim Eissa

Studiengang: Informatin Technology (INFOTECH)

Prüfer: Prof. Dr. Martin Radetzki

Betreuer: Dipl.-Ing Bastian Haetzer

begonnen am: September 21, 2012

beendet am: March 23, 2013

CR-Klassifikation: B.5.1, B.7.1, B.8.2, C.1.1, C.1.3, C.5.3

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.
Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.
Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Stuttgart, den 21. März 2013

Abstract

Transaction Level Modeling (TLM) has recently become a popular approach for modeling contemporary Systems-on-Chip (SoCs) on a higher abstraction level than Register Transfer Level (RTL). In this thesis a multi-core system based on the Xilinx MicroBlaze micro-processor is modeled at RTL and TLM abstraction levels in SystemC. Both implemented models have cycle accurate timing, and are verified against the reference VHDL model using a VHDL / SystemC mixed-language simulation with ModelSim. Finally, performance measurements are carried out to evaluate simulation speedup at the transaction level. Modeling of the MicroBlaze processor is based on a MicroBlaze Instruction Set Simulator (ISS) from SoCLib. A wrapper is therefore implemented to provide communication interfaces between the processor and the rest of the system, as well as control the timing of the ISS operation to reach cycle accurate models. Furthermore, a local memory module based on Block Random Access Memories (BRAMs) is modeled to simulate a complete system consisting of a processor and a local memory.

Contents

Abstract iii

List of Figures vi

List of Abbreviations vii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 1
- 1.3 Thesis Organization . . . 2

2 Background 3
- 2.1 Register Transfer Level (RTL) . . . 3
- 2.2 Transaction Level Modeling (TLM) . . . 3
  - 2.2.1 Cycle Accurate TLM . . . 3
- 2.3 SystemC . . . 4
  - 2.3.1 Channels, Ports and Processes . . . 4
  - 2.3.2 Data Types . . . 5
  - 2.3.3 Performance Measurements . . . 6
- 2.4 Software Tools . . . 6
  - 2.4.1 Modelsim . . . 6
  - 2.4.2 Xilinx Platform Studio (XPS) . . . 7
  - 2.4.3 Xilinx Software Development Kit (SDK) . . . 7
- 2.5 Xilinx MicroBlaze Micro-Processor . . . 7
  - 2.5.1 MicroBlaze Processor . . . 8
  - 2.5.2 Local Memory Bus (LMB) . . . 11
  - 2.5.3 Dual Port Block RAM (BRAM) . . . 12
- 2.6 SoCLib Instruction Set Simulator . . . 12
  - 2.6.1 Available Methods . . . 12
  - 2.6.2 Instruction Set Simulator Basic Untimed Usage . . . 13

3 Literature Review 15
- 3.1 ISS Usage . . . 15
- 3.2 RTL to TLM Transformation Techniques . . . 17
- 3.3 Testing . . . 18
  - 3.3.1 General Processor Testing . . . 18
  - 3.3.2 TLM Verification . . . 19

4 Design **21**
4.1 MicroBlaze System . . . 21
4.2 Data Types . . . 22
4.3 RTL Modeling . . . 23
4.3.1 RTL ISS Wrapper Module . . . 23
4.3.2 RTL Memory Module . . . 26
4.3.3 Multiple BRAMs . . . 27
4.4 TLM Modeling . . . 28
4.4.1 TLM Memory Module . . . 30
4.4.2 Local Memory Bus Interface (LMB_if) . . . 32

5 Implementation **33**
5.1 RTL ISS Wrapper . . . 33
5.1.1 Register Modeling . . . 33
5.1.2 ISS Basic Timed Implementation . . . 34
5.1.3 Adding Complexities . . . 37
5.2 RTL Memory Module . . . 48
5.2.1 BRAM Initialization . . . 49
5.3 TLM Memory Module . . . 50
5.4 TLM ISS Wrapper . . . 51
5.4.1 Instruction Fetch . . . 51
5.4.2 Memory Access . . . 52
5.4.3 Write Back . . . 53
5.5 Interface Adapters . . . 53
5.5.1 RTL-TLM Adapter . . . 53
5.5.2 TLM-RTL Adapter . . . 54
5.5.3 TLM-TLM Adapter . . . 55

6 Results **57**
6.1 Testing . . . 57
6.1.1 RTL Vs. VHDL Testing . . . 58
6.1.2 RTL Vs. TLM Testing . . . 59
6.2 Performance Measurements . . . 59

7 Conclusion **61**
7.1 Summary . . . 61
7.2 Outlook . . . 61

A Testing Applications **62**

Bibliography **67**

List of Figures

1.1 The simplified MicroBlaze micro-processor 2

2.1 Comparison between RTL , Untimed, CX and CA TLM timings . . 4
2.2 The MicroBlaze processor [10] 7
2.3 The MicroBlaze micro-processor system [11] 8
2.4 Type A and Type B instructions [10] 9
2.5 Basic structure of the pipeline 9

3.1 Triggered co-simulation approach [18] 16
3.2 I/O transformation rules [25] 18
3.3 Testbench approach using transactors [26] 19
3.4 TLM-based testbench [28] 20

4.1 A block diagram of the MicroBlaze system containing the processor, LMB, BRAM and the BRAM interface controllers 21
4.2 MicroBlaze system model containing the one processor and two BRAMs with the DLMB and BRAM interface controllers with RTL interconnects. For simplicity the figure only shows only the data memory (PORTB) signals 22
4.3 Simplified MicroBlaze system containing only the processor and the BRAM 22
4.4 RTL model of the MicroBlaze system containing only the processor and the BRAM 24
4.5 RTL ISS wrapper class diagram 24
4.6 RTL local memory class diagram 27
4.7 Simplified RTL Model including only the processor and the local memories 28
4.8 TLM system 29
4.9 TLM ISS wrapper class model 29
4.10 TLM memory class diagram 30
4.11 LMB_if class diagram 32

5.1 VHDL waveform of the double stall case 43
5.2 Pipeline scheduling for a double stall case showing the forwarding procedure 44
5.3 Pipeline scheduling for a double stall case showing a different forwarding handling technique 45

List of Abbreviations

ALU	Arithmetic and Logic Unit
BE	Byte Enable
BRAM	Block Random Access Memory
CA	Cycle Accurate
CPU	Central Processing Unit
CX	Cycle Approximate
DDD	Digital Data Display
DDR2	Double Data Rate Memory
DLMB	Data Local Memory Bus
DMA	Direct Memory Access
DUT	Device Under Test
DUV	Device Under Verification
EFSM	Extended Finite State Machine
EX	Instruction Execute
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
FSM	Finite State Machine
GDB	GNU Project Debugger
GPR	General Purpose Registers
GUI	Graphical User Interface
HW	Hardware
ID	Instruction Decode
IF	Instruction Fetch
ILMB	Instruction Local Memory Bus
IP	Intellectual Property

IPC	Inter Process Communication
ISS	Instruction Set Simulator
LMB	Local Memory Bus
MEM	Memory Access
MP-SoC	Multi-Processor System on a Chip
MUX	Multiplexer
PC	Program Counter
PLB	Processor Local Bus
RAM	Random Access Memory
RAMB	Random Access Memory Block
RTL	Register Transfer Level
SDK	Software Development Kit
SoC	System on a Chip
SW	Software
TLM	Transaction Level Modeling
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WB	Write Back
XPS	Xilinx Platform Studio

Chapter 1

Introduction

1.1 Motivation

Most of the electronic equipment that are currently used by millions of users around the world are run by embedded systems. This means that the development process of such systems is critical and in continuous need for improvement. System simulations are crucial for pre-silicon (early stage) development. One of the bottlenecks that face current developers is the massive simulation times that arise with such embedded systems, especially that the sizes and complexities of such circuits are in continuous increase.

A possible way to decrease simulation times is to change the abstraction level in which the system is defined, such that the system still performs the same function but with less simulation complexity. Simulating a complex system defined by Register Transfer Level (RTL) would mean that the simulator will have to monitor each internal register in the system at each clock cycle and compute how the register value should change. An abstraction level like Transaction Level Modeling (TLM) would offer some communication abstraction to the system such that the intermediate signals that connect different modules of the system can be removed. Those communication channels would then be replaced with method calls to carry out the required functionality. It is hypothesized [1, 2], that by applying such abstraction to an RTL system, simulation times of a System-on-a-Chip (SoC) with high communication rates between its modules would be reduced.

1.2 Objectives

The goal of this Master thesis is to model the Xilinx MicroBlaze system shown in Figure 1.1 at both RTL and TLM abstraction levels using SystemC. The system includes modules such as: MicroBlaze processor, LMB, BRAM and a LMB-BRAM interface controller. The target is therefore to model each of those components in both abstraction levels and reach a final system model for each level that is verified against the reference VHDL model of the system. At the end, the system will be used to run performance measuring tests to calculate the speedup achieved through the abstraction of the system from RTL to TLM. Modeling of the processor is to be done using the available MicroBlaze Instruction Set Simulator functionality. This means that a wrapper has to be implemented that guarantees the correct timing of the ISS operation, and the correct communication interfaces and protocols with external modules such as the LMB or the BRAM.

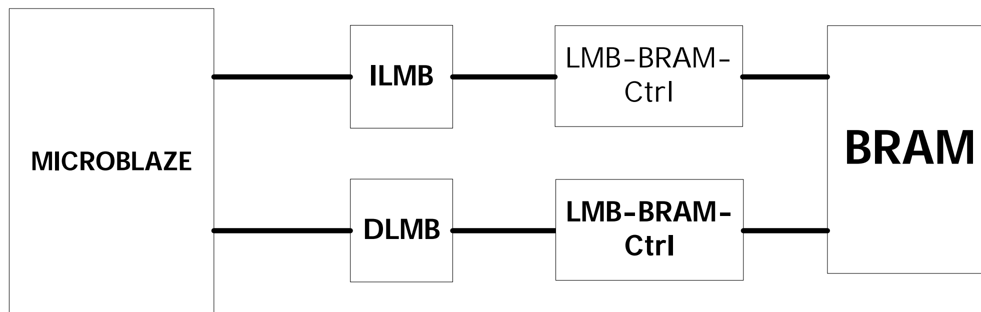

Figure 1.1: The simplified MicroBlaze micro-processor

1.3 Thesis Organization

This thesis is divided into seven chapters. After the current chapter, chapter two provides the background information which is needed as a basis for the work carried out. Chapter three contains reviews for some of the literature to show the current state-of-the-art of the targeted topics. Chapter four consists of the design phase of the task, which reviews the modules that will be created, their interfaces, properties, some brief details about the internal structures and different solutions to reach the targeted functionality. Chapter five contains the main implementation process with all the important details that show how the derived solution from the design chapter has actually been implemented. Chapter six provides the experimental results which includes testing and performance measurements. Finally, chapter 7 summarizes the work done, and provides an outlook for possible system improvements to be done in the future.

Chapter 2

Background

In this chapter, the background information needed throughout this work is reviewed. This includes concepts and definitions, as well as an overview about the target system and tools that are used during the implementation process.

2.1 Register Transfer Level (RTL)

RTL is an abstraction level that defines how the system behaves at every clock cycle (edge). It is used mainly to describe synchronous circuits whose functionality is defined in reference with a clock. RTL representation is achieved by describing a system in terms of registers and the data path components in between them, such that the transferring function between each register is defined [3].

2.2 Transaction Level Modeling (TLM)

TLM is a system-level abstraction level which describes the system in terms of initiators, targets and intermediate channels. It abstracts communication and models the system function in terms of interface methods such as Write(addr0, data0) called from a master to a slave [2].

2.2.1 Cycle Accurate TLM

Fixed definitions for TLM timing profiles varied in the literature reviewed for this work, and different levels of abstraction were further defined inside TLM. The work presented in [2] separated the computation and the communication such that their timings can be independent from each other. Three timing schemes were common in most of the literature: Un-timed TLM, Cycle Approximately Timed (CX) TLM and Cycle Accurate (CA) TLM [1, 2]. In some literature, the names were different, but the concept was essentially still the same; which is that untimed TLM takes no timings into consideration, and all methods are accomplished in a timeless manner. CX takes time into consideration but accuracy is not of importance, and finally CA TLM must abide to the strict timing constraints like an RTL model[1]. Figure 2.1 demonstrates the basic idea of the difference between these levels. In this work, we are interested to model a CA TLM model of the MicroBlaze system; CA TLM is a TLM model that shows similar timing profiles as an RTL model, such that at every clock edge, the internal register values of the TLM model are identical to those of the RTL model[1].

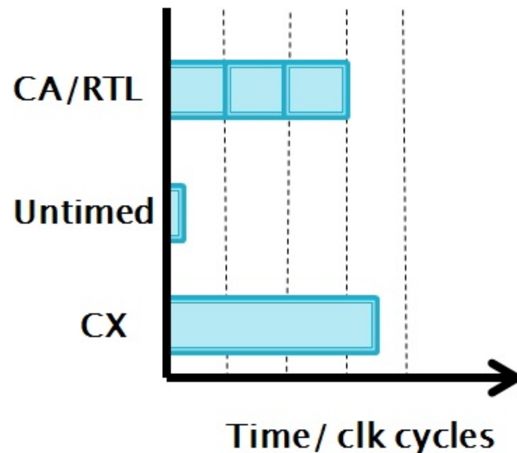

Figure 2.1: Comparison between RTL , Untimed, CX and CA TLM timings

2.3 SystemC

SystemC is a system level design language [4]. It is conceptually a C++ library containing classes and macros that enable the programmer to model concurrent processes, as well as communication means (such as channels and ports) that makes it possible to model a hardware system with C++ syntax. SystemC offers an environment that can conjoin Hardware (HW) and Software (SW) models.

2.3.1 Channels, Ports and Processes

SystemC offers a class definition *SC_MODULE* that resembles the building block of any SystemC design[5]. An *SC_MODULE* can communicate with other modules in the system with communication ports (*sc_port*) which are initialized using a variety of interface options. As an example, ports can be *sc_in*, *sc_out* or *sc_inout* that represent input, output or input/output signals respectively. A process is then used to define the actual functionality of the module where the variables can be modified and conditions can be checked. A process can either be an *SC_METHOD* or an *SC_THREAD*.

2.3.1.1 SC_METHOD

An *SC_METHOD* is like a function; it gets triggered by the occurrence of an event to carry out a specific function. *SC_METHODs* use a concept of sensitivity similar to VHDL processes sensitivity lists, in which whenever a member of that list is changed, the process is invoked. An example for an *SC_METHOD* is a sequential method that gets invoked at every positive clock edge.

2.3.1.2 SC_THREAD

An *SC_THREAD* is different from the *SC_METHOD* in a way that it only gets invoked once upon initialization of the module. Threads have the advantage of providing the capability to halt the execution using a *wait()* call, in which case the execution will only be resumed when an event occurs; this event can either be a member of the static sensitivity events which are those events defined in the sensitivity list of the thread, or the event can be a dynamic sensitivity event which is applicable when the thread calls *wait(some_event)*.

2.3.2 Data Types

Two data types can be used to store the internal state of a module. Either a channel like *sc_signal* can be used or the traditional C++ variables. An *sc_signal* acts like a wire; it is used for internal communications inside the module or from a parent module to an inherited child module[5]. An *sc_signal* can be of any legal data type such as boolean, unsigned, sc_uint, etc.. To write to a signal the write() method has to be called and to read, the read() method has to be called to access the internal value of that "wire".

On the other hand, a member variable like a pure boolean or unsigned type can be hazardous to use for inter-process communication. This is due to the fact that member variables get updated right at the time of assignment, which means that if it is used in two concurrent processes, it would be impossible to know which process updates the variable first which can cause conflicts to arise. For this reason, variables are usually confined inside a single process, or in several processes that are guaranteed not to run concurrently or else conflicts would occur[5].

Another difference between using *sc_signal* and member variables is that inside a process, an *sc_signal* updates its value only at the end of the process run, unlike the variables whose values change immediately at the time of assignment as shown in listing 2.1.

Throughout our design, a standard is established to avoid confusion, in which all input ports will have the prefix *pi_*, output ports will have *po_*, local sc_signals will have *si_*, local variables will have *m_* and finally connecting signals used in binding output and input ports of different modules in the system level modules will have *co_* prefix before the signal names.

Listing 2.1: Comparison between using sc_signal and normal variables

```
sc_signal<bool> si_reg1
bool m_reg1
int i = 0;
void seq_proc() // SC_METHOD sensitive<<clk.posedge()
{
  cout << si_reg1.read() << m_reg1 <<endl;
  if(i == 1){
    si_reg1.write(true);
    m_reg1=true;}
```

```
    cout << si_reg1.read() << m_reg1 <<endl;
    i++;
}
```

The output for such a code run for the first three clock cycles would be:

```
false false
false false

false false
false true

true true
true true
```

2.3.3 Performance Measurements

SystemC (C/C++ in general) includes some libraries that provide several ways to measure time inside the code in order to offer performance measurement options [6].

- time_t time(): returns the current calendar time.
- clock_t clock(): returns the process Central Processing Unit (CPU) time, which is the time passed on the CPU since the program has started.
- gettimeofday(): Native Linux Time Measurement method that returns the actual current time from the operating system.
- getrusage(): Also a Native Linux Time Measurement method, but it returns the time from the CPU clock instead of the operating system, which makes it more accurate in terms of calculating the resource (time) spent by a processor on a specific process, as is shown from its name: get resource usage.

All these time generation techniques can be used to measure how long the simulation takes by measuring the time before the simulation starts, and after it ends and then calculating the difference. In the performance measurement section of this work, the resource usage method will be used to get the real time spent by the processor to run the simulation.

2.4 Software Tools

2.4.1 Modelsim

Modelsim is a tool designed by Mentor Graphics [7] that can be used to simulate HW electronic systems and is considered to be a debugging environment that allows monitoring internal signals during simulations. Modelsim does not only simulate VHDL and Verilog defined systems, but also offers the possibility to simulate systems written in SystemC along with other languages. It is even possible to combine

modules from different languages in a single simulation in what is called Mixed-Level simulation [7]. This technique can be used in this thesis to facilitate the verification of our SystemC design with the reference VHDL system from Xilinx.

2.4.2 Xilinx Platform Studio (XPS)

XPS[8] is a software tool developed by Xilinx that provides a Graphical User Interface (GUI) that enables the user to design different kinds of digital systems starting from simple Finite State Machines (FSMs) to complex micro-processor designs such as the MicroBlaze system. XPS is used throughout this work to design the MicroBlaze system in VHDL which acts as the reference to the designed RTL and TLM models.

2.4.3 Xilinx Software Development Kit (SDK)

SDK is another software tool developed by Xilinx that can be used for editing, compiling and debugging C/C++ based on the GNU compiler. One of the important properties of SDK is the availability of the Data2MEM [9] tool which is used to transform executable and linkable files (.elf) into different kinds of files that can be used to initialize the micro-processor memory blocks.

2.5 Xilinx MicroBlaze Micro-Processor

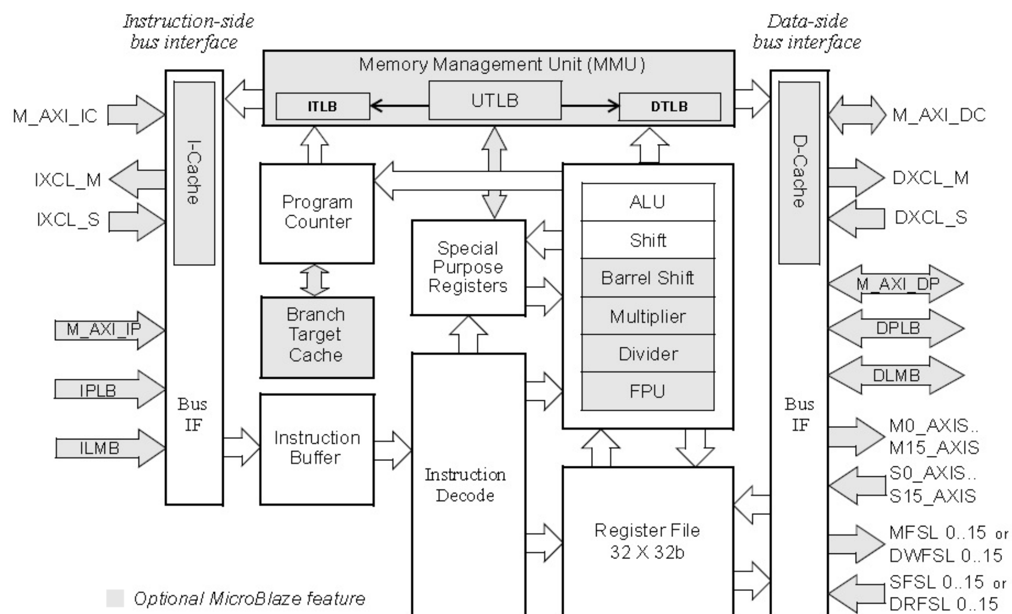

Figure 2.2: The MicroBlaze processor [10]

The MicroBlaze is a micro controller system originally designed to work with the Xilinx Field Programmable Gate Array (FPGA) to run different kinds of applications [11]. Figure 2.2 demonstrates the core of the micro-processor system, which is the MicroBlaze processor, and shows the internal structure of the processor, as well as communication options with external modules. Figure 2.3 on the other hand demonstrate a typical example of the MicroBlaze micro-processor structure. It consists of four main blocks, aside from the debugging module which is usually an optional block which is unnecessary for the core operation.

2.5.1 MicroBlaze Processor

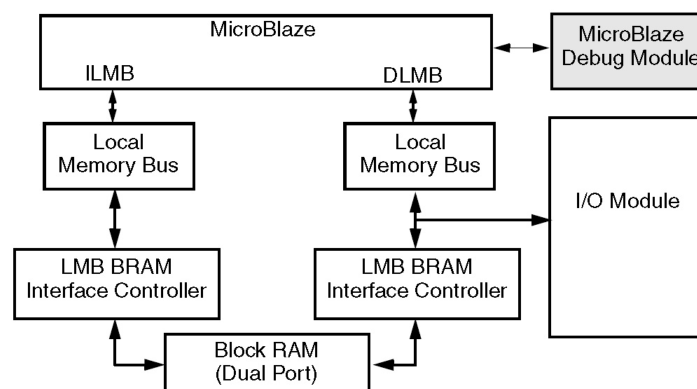

Figure 2.3: The MicroBlaze micro-processor system [11]

The MicroBlaze processor, demonstrated in Figure 2.2 is an embedded soft core processor based on the reduced instruction set computer (RISC) architecture[10]. It includes several features, most of which can be configured by the user during the design phase of the system. The full list of features and configuration options are demonstrated in [10]. Below, an overview of the most interesting points, and points relevant to the work presented in this work is presented:

- 32-bit width for the General Purpose Registers (GPR) and address bus.
- Instruction word is 32 bits with three operands and two addressing modes (Type-A and Type-B) as shown in Figure 2.4.
- An instruction prefetch buffer in the instruction fetch pipeline stage.
- Pipeline depth can be chosen to be either three or five stages.
- Option to include Instruction and Data Caches for faster fetching.
- HW ALU extensions such as barrel shifter, multiplier, divider (32 or 64 bit) or a floating point unit.

– Fast Simplex Link (FSL) bus for communication with external modules.

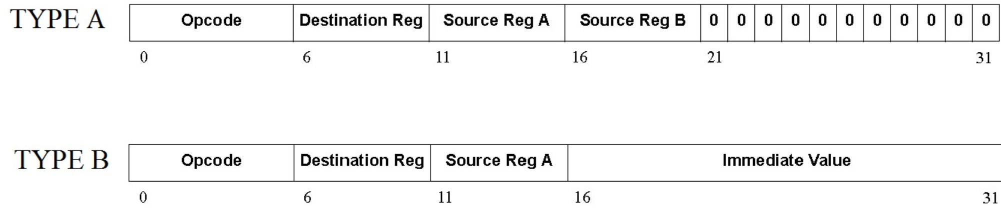

Figure 2.4: Type A and Type B instructions [10]

2.5.1.1 5-Stage Pipeline Structure

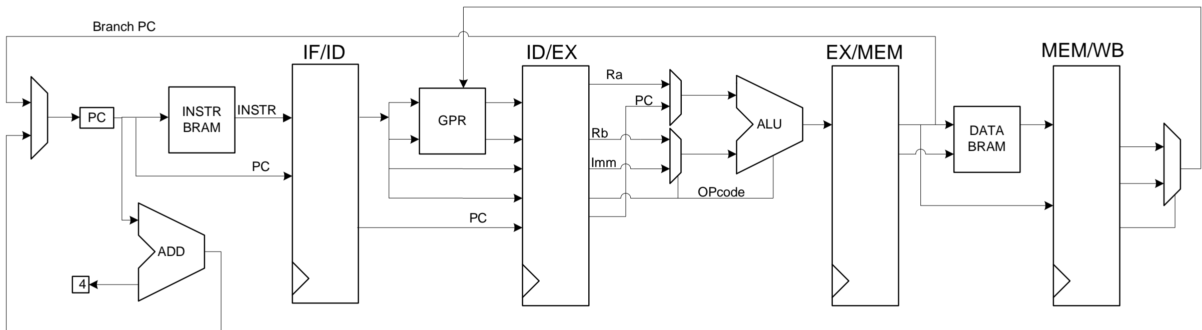

Figure 2.5: Basic structure of the pipeline

The work presented in this thesis is only concerned with the MicoBlaze 5-stage pipeline. The MicroBlaze internal 5-stage pipeline is similar to the structure presented in [12]. Figure 2.5 shows the basic diagram for a typical 5-stage pipeline structure. It is simply defined in terms of register blocks and combinational logic in between them which define the pipeline stage functionality. The pipeline is divided into five stages: Instruction Fetch (IF), Instruction Decode (ID), Instruction Execution (EX), Memory Access (MEM) and Write Back (WB).

– Instruction Fetch Unit:

  IF unit is the first stage of the pipeline which is responsible for providing the current Program Counter (PC) to the instruction memory to fetch a new instruction and feed it into the ID unit. IF is also responsible for incrementing

the PC by four after each fetch in order to have the correct PC for the next clock cycle.

– Instruction Decode Unit:

  ID unit is responsible for dividing the instruction coming from the IF unit into: Opcode, Destination Register (Rd), source registers (Ra and Rb) and an immediate operand (IMM) . Moreover, the ID unit accesses the GPR to get the values of the source registers Ra and Rb. ID then provides this information to the Instruction Execution (EX) unit.

– Instruction Execution Unit:

  EX is the stage responsible for the actual execution of the instruction, it mainly consists of an Arithmetic Logical Unit (ALU) that performs the required function. Each of the two inputs of the ALU is selected by a multiplexer (MUX); the first MUX selects between Ra and PC, whereas the second one selects between Rb and IMM. The ALU then performs an arithmetic operation on those two values and produces an output to be forwarded to the MEM unit. The select signals for the MUXs and the ALU are generated from the Opcodes; for example if the instruction is *ADD R3, R1, R2* then the selects will be such that the first MUX passes the value of Ra (R1) to the ALU, the second MUX will pass the value of Rb (R2), and the ALU will be set to perform an add operation on both values. However, if the instruction is for example *BRAI 24*, MUX1 will pass the PC, MUX2 will pass the immediate value of 24 and the ALU will also perform an addition to calculate the new PC and provide it back to the IF. In any case, the output of the ALU gets forwarded to the MEM unit.

– Memory Access Unit:

  MEM stage is responsible for writing to and reading from the data memory. It receives the required information (like memory address and data to be written) from the EX stage, and upon finalizing the memory access it provides the data read from the memory to the WB stage.

– Write Back Unit:

  WB is responsible for updating the GPR with the new register value resulting from the instruction execution, whether it is a local instruction like *ADD R2, R1, R0* or a memory access instruction like *LW R2, R1, R0*.

2.5.1.2 Basic Implementation Hazards

This basic implementation of the instruction pipeline would face two types of hazards while running an assembly program; the hazards can either be control hazards or data hazards.

A control hazard occurs whenever a branch is taken. By taking a look at the basic implementation idea of the pipeline, it is obvious that when a branch is taken,

two extra unwanted instructions will have been already in the pipeline in IF and ID stages. To overcome this hazard, NOPs have to be inserted in the pipeline to replace the unwanted instructions such that the erroneous instructions would not be executed. However, it has to be taken into consideration that some branch instructions make use of what is called as a “delay slot” where the instruction following the branch instruction should be normally executed. This delay slot reduces the overhead caused when a branch is taken.

A data hazard occurs when two or more consecutive instructions are dependent on each other. For example:

ADD R2, R1, R0
ADD R4, R3, R2

In this example R2 is updated by the first instruction, and used by the second one. In the basic implementation, R2 would only be updated during the WB stage of the first instruction, this means that the second instruction cannot perform the EX stage unless the WB for R2 has been completed. Therefore, the EX has to be ”stalled” for two clock cycles until the WB has been finished. Another way to overcome this stalling is to apply the forwarding technique. This means that the ALU output from the EX stage would be stored in a temporary forwarding register to be used also in the EX stage during the next cycle. The output of the the forwarding register is then fed into the two MUXs responsible for the ALU inputs. A hazard detection unit would then be used to detect such a dependency and set the select signals for both MUXs such that the value from the forwarding register would be used instead of the value read from the GPR.
Another data hazard might occur in such a case:

ADD R2, R1, R0
ADD R10, R9, R8
ADD R4, R3, R2

This case is different from the previous one, since the required R2 value will not be coming from the EX unit, but it will be from the WB, therefore there needs to be another path from the WB output to the ALU input MUXs. The hazard detection unit should then be able to differentiate between these two cases of forwarding and provide the correct MUX signals accordingly.

2.5.2 Local Memory Bus (LMB)

LMB is a fast single Master - multiple Slave local bus that is used to connect the instruction and data ports of the MicroBlaze processor to high speed peripherals such as the BRAM block[13]. It is also a 32-bit module and can allow up to 16 memory slaves. The main function of the LMB is forwarding the signals from the processor to the memory and vice versa.

2.5.3 Dual Port Block RAM (BRAM)

The BRAM is an on chip Random Access Memory (RAM) block that internally instantiates a number of smaller memory blocks (RAMB) primitives in order to reach the required configurable memory size [14]. It can be divided into separate ports in order to distribute the memory accessing capabilities. For example, in the MicroBlaze micro-processor typical setup demonstrated in Figure 2.3, the BRAM is divided into two separate ports (PORTA and PORTB) such that one port is for instruction communication, and the other is for data. The size of the BRAM can be configured by setting the value of *C_MEMSIZE* to some value between 4094 (4kB) and 524288 (512kB) depending on the internal primitive memory blocks used [14].

The BRAM and the LMB cannot be directly connected. For this reason the LMB-BRAM interface controller [15] is required, as shown in Figure 2.3. It translates the signals coming from the LMB into signals for the BRAM. The LMB-BRAM interface controller is also responsible for address decoding; meaning that it is responsible for selecting whether to forward the LMB signals to the BRAM or not, based upon the address mapping.

2.6 SoCLib Instruction Set Simulator

System On a Chip Library (SoCLib) is an open platform for virtual prototyping of multi-processors SoC (MP-SoC) [16]. From their many offered designs, they offer a MicroBlaze based Instruction Set Simulator (ISS) that is used to provide the core processor functionality for this work.

The ISS is a C++ class containing the basic functionality of the MicroBlaze processor. It contains some basic internal registers such as the GPR (an array of 32-bit unsigned variables) and different method declarations, each of which carry out a separate pipeline stage functionality. Timings however are not taken into consideration in the ISS, it simply provides the interface methods for the pipeline stages. The ISS uses mainly two data types: unsigned int which is “type defined” into the name *uint32_t* which resembles a 32-bit unsigned template, and the other type is boolean to model the single bits and flags of the system.

2.6.1 Available Methods

- getInstrucionRequest(bool &req, uint32_t &addr): It is responsible for returning the current PC from the ISS. The method takes two references as arguments; a boolean to notify the caller that there is a valid request, and an unsigned value containing the required PC. This method always returns true to the *req* reference, along with the current PC from the *r_pc* register to the *addr* reference.

- setInstruction(bool error, uint32_t insn): It is responsible for feeding the new instruction into the processor. The method takes as arguments a boolean that shows whether there had been an error with the IF, and an unsigned value

with the fetched instruction. The method then sets the local variable *m_ir* to the instruction *insn*, and sets the *m_ibe* to the error flag in order to declare an exception if an error had occurred.

- step(): This is the longest and the most complex method of the ISS, and is responsible for actually executing the instruction that has been fed into the processor. First, the 32-bit instruction has to be decoded (separated) into opcode, destination register address (rd), operand A address (ra), operand B address (rb), and the immediate operand (imm). This is done using the local method call IDecode(m_ir, &ins_opcode, &ins_rd, &ins_ra, &ins_rb, &ins_imm). IDecode method first determines whether the instruction is Type A or Type B from the Opcode using a predefined lookup table matching each known MicroBlaze instruction to its respective type, and then it performs some shifting to the instruction to extract each field on its own.

  After decoding the instruction, *step()* method then has a switch statement over the *Opcode* for every instruction possibility. A LOAD statement is defined at the top of the class that assigns the memory access signals whenever a load instruction is being executed for example. A similar definition is available for the STORE. At the end of the *step()*, the current and next PCs are updated.

- getDataRequest(bool &valid, enum DataAccessType &type, uint32_t &address, uint32_t &wdata): It is a simple method that takes four references as arguments, and assigns to them the memory access signals such as the address, the data to be written (in case of store instruction), the type of access (read half word, write byte, etc..), and a boolean that states whether a memory access is actually needed or not.

- setDataResponse(bool error, uint32_t data): It returns to the ISS the data that has been requested from the external memory. This method is similar to the *setInstruction()* method; it has an error argument to show if an error had occurred during memory access, and an unsigned variable containing the requested data. Internally, the method has a switch statement which decides what to do with the incoming data according to the memory type that had been requested.

2.6.2 Instruction Set Simulator Basic Untimed Usage

To use the ISS, a wrapper class needs to be defined that instantiates the ISS and performs the methods consecutively as shown in listing 2.2. The wrapper should also contain sources for instructions and data memories, and this can either be from local arrays like the example in the listing, or from external memory modules. This would then make the code more complex due to the inclusion of some communication techniques with external modules.

Listing 2.2: Basic ISS Wrapper Implementation

```
//class ISSWrapper with an ISS instance iss
while(true)
{
  iss.getInstructionRequest(&m_pc);                    //IF
  iss.setInstruction(false, instrMEM[m_pc] );          //ID
  iss.step();                                          //EX
  iss.getDataRequest(&m_Wdata, &m_addr, &m_DType);     //MEM
  if(m_DType == WRITE)
    dataMEM[m_addr] = m_Wdata;
  iss.setDataResponse(false, dataMEM[m_addr]);         //WB
}
```

Chapter 3

Literature Review

The task of this thesis is mainly divided into two parts, the first part is to use an ISS to reach an RTL model of the MicroBlaze processor, and the other part is to convert this RTL model into TLM and verify that they are consistent.

3.1 ISS Usage

Over the past few years, it has become extremely popular to use an ISS to model the functionality of a processor [1, 17, 18, 19, 20]. Using such an approach makes it easier to model complex hardware and software systems in a co-simulation environment. Moreover, it saves a lot of time spent on designing the RTL model of the system for simulation or testing in the initial phases of the design if the RTL is not already available.

In the literature there are different approaches on how to use an ISS. In the first example [1] an ISS similar to the one used in this thesis was used. They used the ISS from SoCLib to model a Multi-Processor System on a Chip MP2-SoC in three types of TLM abstraction levels: Untimed, Approximately Timed (AT) and Cycle Accurate Bit Accurate models (CABA). To achieve this, they designed three separate wrappers; one for each abstraction level. They considered the ISS to be an FSM with almost infinite states where each state resembles a unique combination of the internal registers of the processor and the memory. Each time an instruction is executed the FSM state changes. Their idea of a timed model of the system would then be achieved if the ISS changed state at every system cycle not necessarily at every instruction execution[1]. For that paper, all the TLM models were designed according to the OSCI TLM 2.0 standard [21]. For the untimed model, there was no synchronization for the processor or for the memories which means that the memory and cache delays for example are not taken into consideration. For this reason, the untimed ISS wrapper simply contained an *SC_THREAD* with an infinite loop that continuously got the instruction and data requests from the ISS, performed the memory accesses and then performed the execution in a single iteration of the loop. On the other hand, the AT-TLM wrapper has an approximate time for the memory access by calculating the difference between the time before and after the memory access. Then the ISS is informed with that time using *iss.executeNCycles(dt)* so that the ISS is delayed for "approximately" the same time as the memory access. Finally, the CABA was done differently, it was designed in an "RTL" fashion, where the instruction requests were translated into different sc_signals, and instead of having an infinite loop that keeps getting requests and setting responses, the CABA had

two processes; one of them was sensitive to the positive clock edge, and the other was to the negative one. The negative (clock falling) one set the request signals to communicate with the cache, and the clock rising read the data from the cache to the ISS. By this they accomplished a cycle accurate design since the system is always synchronized with the clock edges[1].

In another example, [18] implemented an ISS wrapper with an interface which was based upon the standard GDB remote debugging interface[22]. The paper studied two different approaches to use an ISS to reach a Co-Simulation environment with an ISS using SystemC in order to simulate a multi-processor system containing both HW processor models, as well as SystemC ISS cores. The first approach (Triggered Co-Simulation) proposed implementing a wrapper (gdbAgent) that is based on the Digital Data Display (DDD) package [23] to control the ISS. Figure 3.1 shows the general structure of the triggered approach, it shows the two pipes by which the wrapper controls the ISS using the gdb commands. This approach had the advantage of offering granularity to the system simulation because it used the gdb instructions such as *Next, Run and Quit*; this means that there were distinctive states between each instruction execution that would facilitate testing or monitoring of the system.

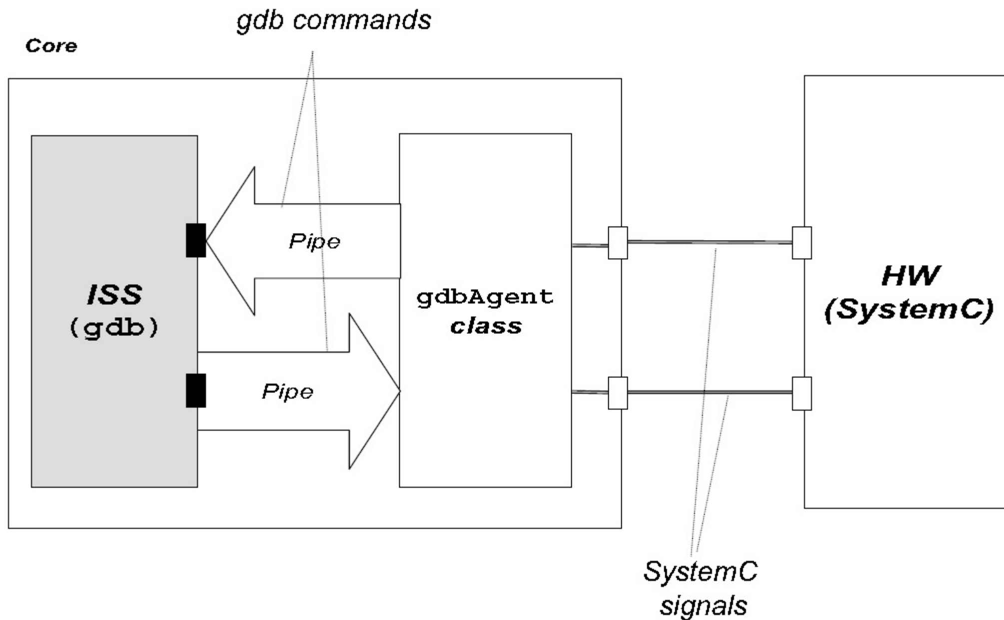

Figure 3.1: Triggered co-simulation approach [18]

The approach given in 3.1 had a disadvantage that it caused too much Inter-Process Communication (IPC) due to having a separate module for the wrapper and for the ISS itself. This meant that multiple commands needed to be called for each instruction, therefore they proposed another solution which they called Legacy

Co-Simulation. In this approach they embedded the ISS as an sc_module inside the wrapper and thus minimized the IPC. The legacy co-simulation however did not provide much simulation granularity as the triggered one so both solutions were presented as a tradeoff between simulation times and simulation granularity [18].

By studying the reviewed approaches, it is possible to match some of this work's requirements with the ideas implemented in the literature for the ISS usage. For example, the CABA TLM approach [1] is similar to the CA TLM concept used in this work. However, the ISS used in that case was not based on the MicroBlaze system. Moreover, *SC_THREADs* were used to develop their wrappers, while in our case it would be possible to use the faster *SC_METHODs* since blocking operations can be avoided. Another disadvantage (from the point of view of the requirements) is that part of their system is triggered by the negative clock edge, which cannot be used to model the MicroBlaze processor. Another interesting concept for the topic of this work is the Legacy Co-Simulation concept provided in [18], the idea now is to try to combine both to design the models required for this work.

3.2 RTL to TLM Transformation Techniques

A lot of researchers are now interested in taking the pre-silicon testing and verification phases of SoCs to a higher abstraction level such as TLM. This would save simulation time that become massive when complex systems are simulated using a low abstraction level such as RTL [24]. For this reason, different researchers proposed different techniques to transform RTL Intellectual Properties (IPs) into TLM models[25, 26, 24, 27, 28, 29]. Some of the approaches proposed an algorithm for automatic transformation from RTL to TLM models like [25]; the proposed technique was divided into three steps:

1. Identification of the computational phases of the RTL model.
2. Generation of TLM functions through merging the states
3. Generation of the TLM communication protocol.

They started by defining the RTL system in terms of an Extended Finite State Machine (EFSM) sequence, with the EFSM divided into three computational phases:

i. The input phase, where the IP gets the inputs.

ii. The elaboration phase, where the actual computation of the inputs and internal registers is carried out.

iii. The output phase where the output ports are assigned. After defining the computational phases.

They merged all the elaboration phases together to generate the actual TLM functionality of the system. They used this merging concept to transform the RTL elaboration phases into a single TLM method call. Finally they implemented the

TLM communication protocol from the input and output computation phases in accordance with the OSCI TLM 2.0[21] standard to give the system a global interface to enable reuse as shown in Figure 3.2.

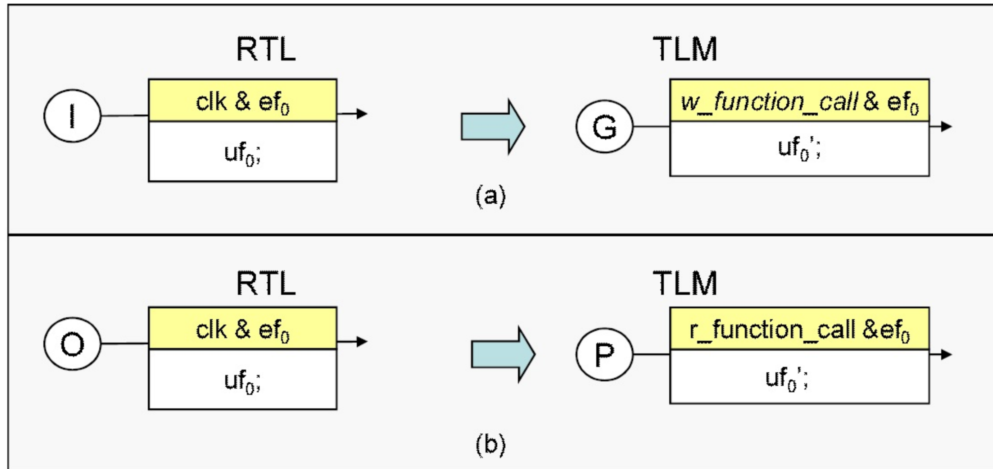

Figure 3.2: I/O transformation rules [25]

3.3 Testing

In the RTL-TLM mixed level research field, testing can be divided into two main parts. The first part is to test and verify the functionality of the RTL model (for example a processor model like the aim of this thesis), and the second part is to perform TLM-RTL verification, where the TLM is tested to prove that it carries out the exact same functionality as the RTL model (which had already been verified).

3.3.1 General Processor Testing

The traditional way of testing any digital electronic system, is to simply derive test cases that would iterate over all the inputs and internal line values to provide all the possible combinations for them, and assign these test cases to the system while monitoring the outputs and comparing them to the expected values that should occur if the system is correct. While such a technique would work well for a small digital system such as a collection of gates and registers such as a simple Finite State Machine (FSM), computing all the possible combinations and simulating them would be almost impossible to apply on a complex structure such as a processor. Therefore new approaches were researched to enable testing complex electronic structures.

According to the literature, different approaches for testing the functionality of a processor are available. One way to do this is to use the processor to run an operating system on an emulator which is the approach used in [19, 20], where they

used a cycle accurate ISS as a core for a SoC in accordance with the Quick Emulator (QEMU) tool to run a full Linux kernel. However, as much reliable as this testing technique might be, it would be too complex to be adopted in this work. It would be more convenient to follow the conventional techniques of providing the processor with test instructions and monitoring the response, especially because the models designed for this work need to be verified against VHDL RTL models, which is a concept that was not addressed in the reviewed literature.

3.3.2 TLM Verification

The second stage of testing is to verify the functionality of the TLM model. In [26] they were interested in designing a reusable testbench that can be used on models with different abstraction levels including RTL and TLM. The main point addressed in that paper, is the use of transactors. Transactors are adapters that can be connected to the Device Under Test (DUT) to transform the communication technique (abstraction level) into a unified level that can communicate with the rest of the testbench as shown in Figure 3.3. The testbench then simulates some inputs to the DUT through the transactor, and compares the outputs of the system with predefined test results to verify the system functionality.

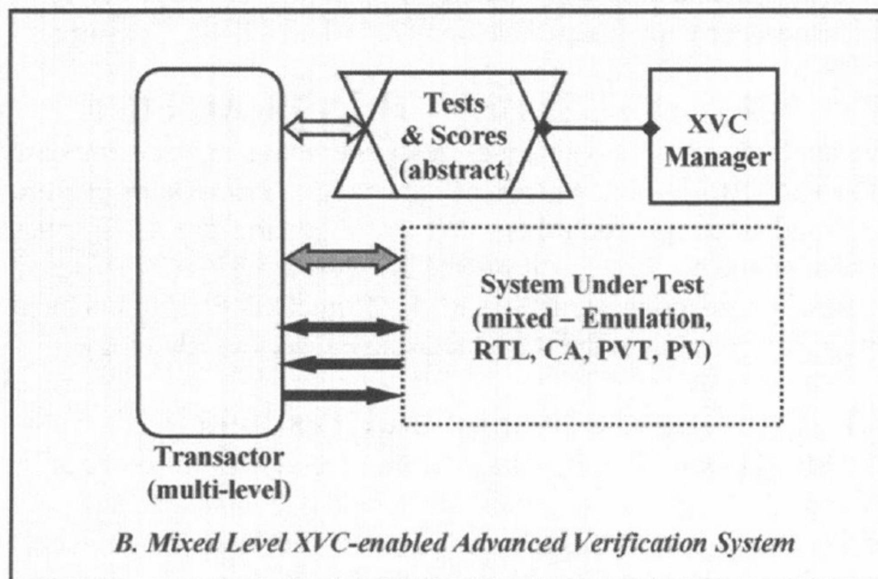

Figure 3.3: Testbench approach using transactors [26]

Another testing approach is given in [28], where they defined a TLM-based testbench that verifies an RTL or a TLM model against a "golden" RTL model as demonstrated in Figure 3.4. Since the testbench itself is defined in TLM, transactors were needed to connect RTL models to the rest of the system for both the reference RTL model, as well as the Device Under Verification (DUV). The verification starts by the generation of TLM input stimuli by the automatic test generator. These

stimuli are then forwarded to the golden model, as well the DUV. If the target model is in RTL, then a transactor would be needed to transform the TLM function calls to RTL sequence of statements. Outputs of both models are then forwarded to a checker tool that performs the actual verification of the DUV.

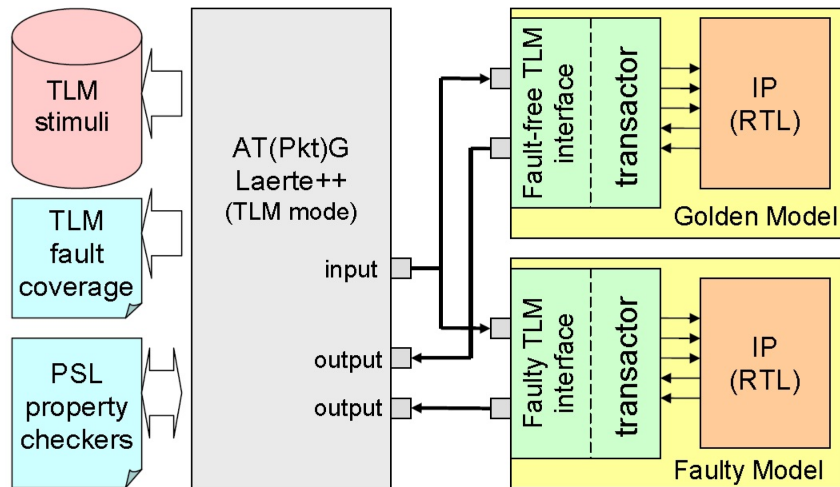

Figure 3.4: TLM-based testbench [28]

As shown in the demonstrated approaches, transactors (or adapters) are usually used when different level models need to be verified. This concept can be used during the testing process for the designed moels presented in this work.

Chapter 4

Design

In this chapter, different approaches will be reviewed, about how to transform the MicroBlaze system into the RTL and TLM SystemC models.

4.1 MicroBlaze System

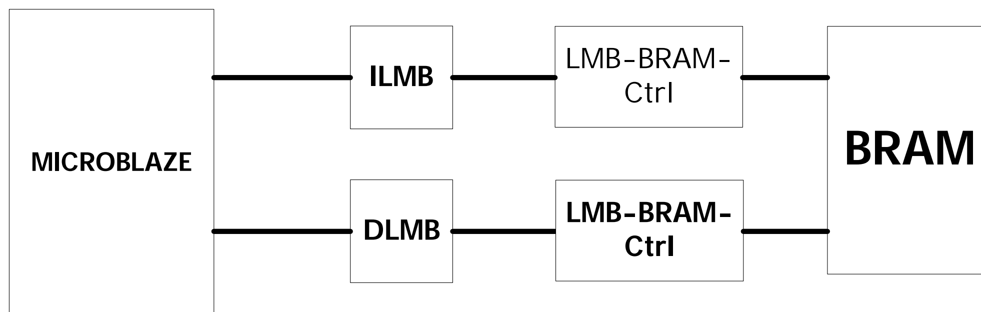

Figure 4.1: A block diagram of the MicroBlaze system containing the processor, LMB, BRAM and the BRAM interface controllers

The simplest MicroBlaze system should contain at least one processor, and one local memory block as shown in Figure 4.1. An LMB is needed to act as the interface between the processor and the BRAM, and it makes it possible for one processor to be connected to more than one BRAM. Each BRAM requires an LMB-BRAM interface controller to be connected to the LMB. The first approach to model such a system, would be modeling each component in a separate module, and form a system that looks exactly like Figure 4.1. Figure 4.2 shows an example of a system containing two BRAMs with the RTL intermediate signals, the figure shows how complex and crowded in terms of intermediate signals it becomes due to the intermediate modules. It would be much simpler if the LMB and BRAM interface controllers could be somehow embedded into the other main modules to save the unnecessary connection complexities.

The LMB is a rather simple module, and is responsible for connecting the MicroBlaze processor to the BRAM. Since it is such a simple module, it would be pretty easy to simplify the system and embed the address decoding functionality inside the processor and save an extra module declaration. Another simple module is the LMB-BRAM interface controller. This module is an interface that enables a BRAM to be connected to an LMB. It performs basic forwarding of memory access

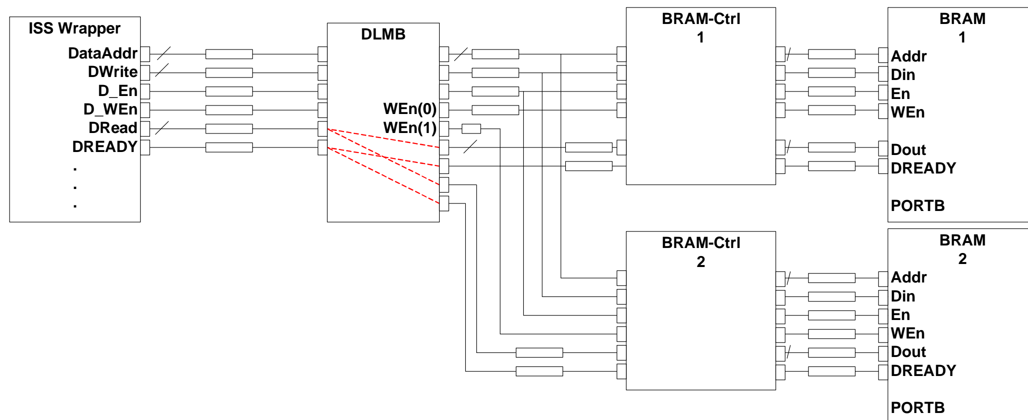

Figure 4.2: MicroBlaze system model containing the one processor and two BRAMs with the DLMB and BRAM interface controllers with RTL interconnects. For simplicity the figure only shows only the data memory (PORTB) signals

signals, and it is also responsible for address decoding and forwarding the memory requests to the correct BRAM in case more than one BRAM is connected to the processor. Therefore it can also be abstracted inside the BRAM itself. This way the whole local system could be modeled using only two modules as shown in Figure 4.3.

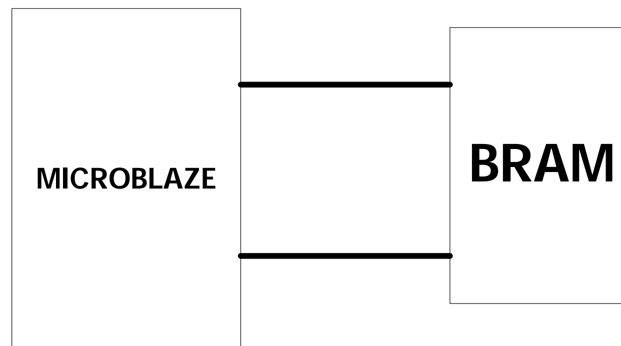

Figure 4.3: Simplified MicroBlaze system containing only the processor and the BRAM

4.2 Data Types

The MicroBlaze is a purely digital system and most of the registers and communication signals are 32-bit wide. Therefore there are mainly two data types needed throughout the whole system; the first type is the one used to define the single bit

registers such as intermediate flags. Such a data type can either be defined using the sc_logic type which is used in the real MicroBlaze system, or it can be defined using the simpler and less memory consuming boolean or sc_bit types. The difference between the resolved logic and the bit is that the bit only has two options for each variable which are either '0' or '1'. On the other hand, the resolved logic type is used to resolve signals that are driven by different sources and resolve the signal to a known value in case there is a conflict in the driving signals, the possibilities are: '0', '1', 'Z' or 'X'.

In our system it would not be necessary to use the more complex resolved logic types since conflicts should not be expected so we can simplify the system in terms of variable complexity. Therefore, the sc_bit or the boolean type would be used for our single bit variables in our system. Furthermore, [30] suggests a 20% performance speedup when using boolean over sc_bit data type, therefore the boolean data type is used for our single bit variables throughout our system. The other data type which is mostly used throughout the system, is the 32-bit variable that is used for most of the system registers like the instructions, operands and addresses registers. For this type, the same concept of single bit data type applies, either the sc_lv (logic vector) can be used, or the simpler unsigned or the sc_bv (bit vector), again for the sake of simplicity and boosting the performance, the unsigned data type is used during the system modeling.

The other important concern with data types is that they need to be compatible with the VHDL data types that present the same function because our models shall later be combined with the VHDL system for mixed level simulation. Fortunately, according to [7] unsigned data types are compatible with VHDL's std_logic_vector and bool is compatible with std_logic which are the data types used in the MicroBlaze VHDL system.

4.3 RTL Modeling

The RTL model consists mainly of two classes of type sc_modules; one representing the MicroBlaze processor called *ISS_wrapper_RTL* and another one for the memory block called *local_mem_RTL* as shown in Figure 4.4. The names and properties for the ports used for communication between both modules are all inherited from the real system shown in the reference guides [31] and [14].

4.3.1 RTL ISS Wrapper Module

There are several ways to model the MicroBlaze processor; the most straight forward approach would be to model all the internal components starting from registers and logic gates until complex structures like memories and ALUs. Then all those components would be used to create a structural model that looks exactly like the internal structure of the MicroBlaze processor. However, the task of this work is to reach the same MicroBlaze processor functionality using the available ISS, therefore

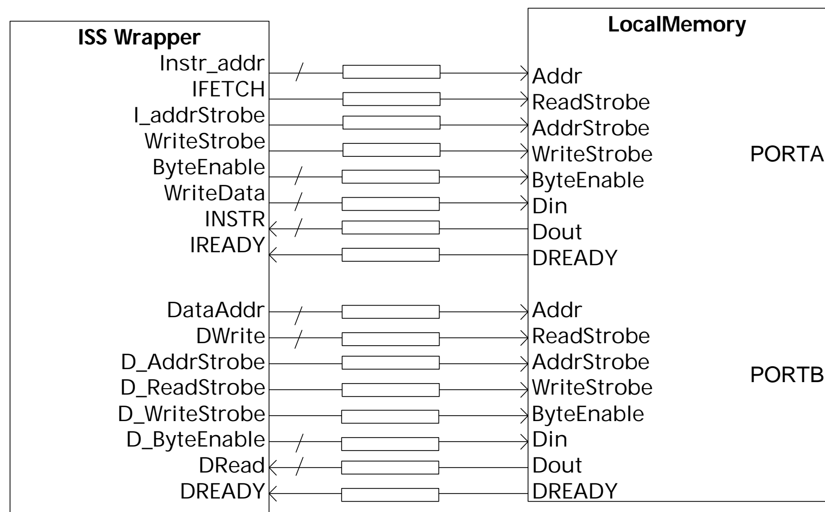

Figure 4.4: RTL model of the MicroBlaze system containing only the processor and the BRAM

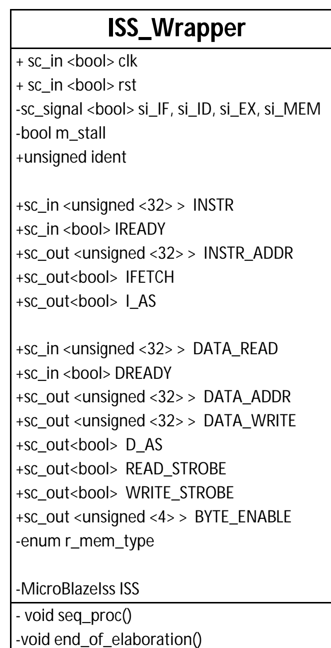

Figure 4.5: RTL ISS wrapper class diagram

another approach to model the micro-processor has to be found. The idea is to find a way to achieve the same functionality using the methods offered by the ISS.

Since an ISS will be used for the core functionality of the processor, another module has to be defined to act as a wrapper for the ISS. The ISS wrapper module would control all the timings of the processor, as well as the communication with other modules such as the BRAM. So the ISS wrapper is a module class that has an internal ISS instance, and some input and output ports to communicate with external modules as shown in the class diagram in Figure 4.5. For initialization, the ISS_wrapper_RTL class constructor needs some important details to facilitate the communication with the rest of the system such as:

- ident: provides an identity for the processor that is used for determining the processor priority in multi-processor systems.
- num_slaves: provides the number of memory blocks connected to the processor.
- high_addresses[]: an array with size number of slaves, it provides the high addresses of each of the slaves.
- low_addresses[]: an array with size number of slaves, it provides the low addresses of each of the slaves.

The MicroBlaze processor mainly consists of the 5-stage pipeline. Modeling of such a system, can be achieved by dividing the code into a separate section for each pipeline stage. One idea to achieve this, is to have a separate process for each pipeline stage, each of those processes would be sensitive to the positive clock edge, this would guarantee the concurrency of all the pipeline stages as is the case in the real HW model. However, having five sequential processes in a single class would have unnecessary overhead on the memory usage as well as simulation times. For this reason, it would be more efficient to define the whole functionality in one sequential process. A way to achieve this would be to have a code segment for each of the pipeline stages, with sc_signals to function as the interconnecting registers between each stage.

An example for such a design is shown in the following code segment, where in the first clock cycle, IF will be done for an arbitrary instruction x and si_IF_valid will be set to true at the end of the process run, therefore ID for x can only be initiated one clock cycle after x has undergone IF, and so on for the rest of the stages, this way the correct timing of the pipeline can be accomplished.

```
void seq_proc() //sensitive << clock positive edge
{
  //perform IF
  si_IF_valid.write(true);
  if(si_IF_valid.read())
  {
    //perform ID
```

```
    si_ID_valid.write(true);
  }
  else
    si_ID_valid.write(false);

  if(si_ID_valid.read())
  {
    //perform EX
    si_EX_valid.write(true);
  }
  else
    si_EX_valid.write(false);

  if(si_EX_valid.read())
  {
    //perform MEM
    si_MEM_valid.write(true);
  }
  else
    si_MEM_valid.write(false);

  if(si_MEM_valid.read())
    //perform WB
}
```

4.3.2 RTL Memory Module

As for the memory, the main idea is to have a sequential module that checks for the memory access strobes (address, read and write) at each positive clock edge, and perform the action accordingly. If there is a high read strobe and address strobe, then the BRAM replies with the memory content inside the specified address along with a high data ready signal that would stay high for 1 clock cycle. On the other hand if there is a high write strobe and address strobe, then the BRAM should write the input data in the specified address, and return the content of that address back in the data read signal also along with a high data ready signal that would stay high for one clock cycle.

Figure 4.6 shows the class diagram of the RTL memory module. The design is done for the Write-First write mode shown in the BRAM reference guide [14] which means that the reply from the BRAM after a write request will have the up-to-date memory content (the data that had just been written) and not the old data like the Read-First mode [14]. Apart from the sequential process, the BRAM contains another method *load_BRAM()* that is called during the initialization of the BRAM; it is responsible for filling the instruction program into the BRAM, for this it needs a path to a memory file that is generated by the data2MEM command. The information needed to initialize an RTL memory instance is:

– HighAddr: defines the highest address mapped to the BRAM.

– LowAddr: defines the lowest address mapped to the BRAM.

– mem_size: that defines the size of the memory given in number of words inside the memory array.

– mem_path: a string that points to the (.mem) file with the program to be loaded into the BRAM.

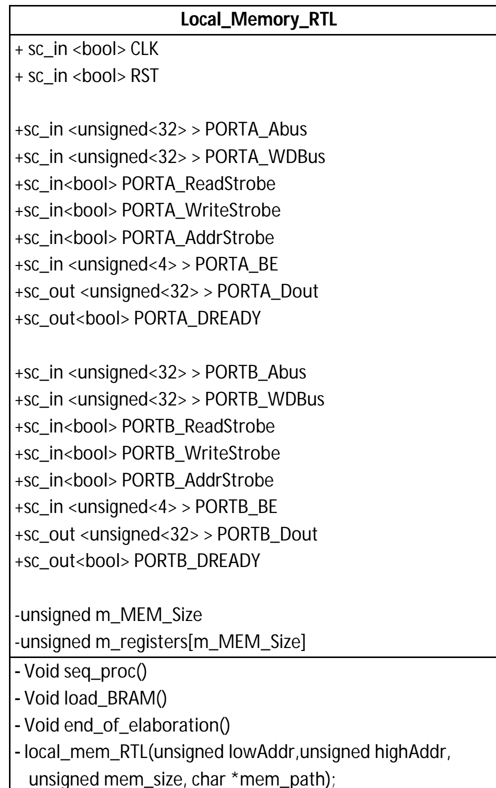

Figure 4.6: RTL local memory class diagram

4.3.3 Multiple BRAMs

One way to make it possible for multiple BRAMs to be connected to the processor, is to simply share all the communication signals with all the slaves. However, this would mean that there would be some signals with multiple drivers, and that the wrapper will not be able to select a specific BRAM to communicate with. At the same time, since the BRAM only acts when an address strobe is detected, then it would be possible to share all the other channels going into the BRAM and have exclusive address strobes. The required address strobe would then be chosen according to some sort of address decoding to address only the correct BRAM. The reply signals from the BRAMs also cannot be shared because this will mean

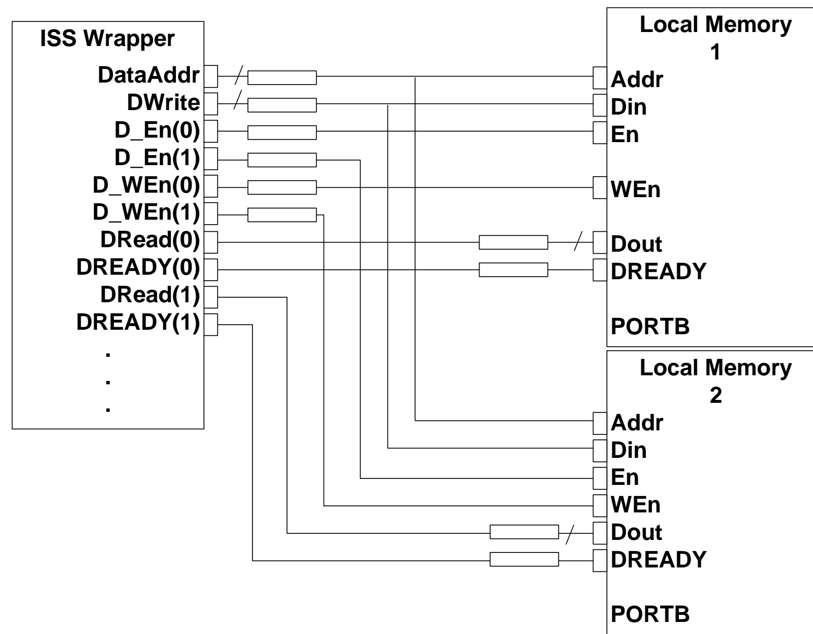

Figure 4.7: Simplified RTL Model including only the processor and the local memories

that they would have multiple drivers which would cause conflicts at every memory access. Therefore the ISS wrapper would have to have separate inputs for each BRAM. The wrapper then waits for the reply from the input selected from the address decoding process done in the previous stage as demonstrated in Figure 4.7.

4.4 TLM Modeling

To transfer to TLM, only the communication techniques has to be altered. Instead of counting on the input and output signals in connection with the BRAM to read and write data to the memory, the wrapper (Master) needs to call internal methods in the BRAM (Slave) to carry out the required function. The processor uses sc_ports that implement an sc_interface class *LMB_if* to communicate with the BRAM. The BRAM on the other hand has sc_exports that also implement the interface *LMB_if*. BRAM exports are bound to the processor ports to enable the method calls inside the BRAM class. The transformation is then completed by replacing all the output port assignments and input port readings of the RTL system to some method calls. For example the code for an RTL request to write some data (wData) to a specific memory address (addr) would be as follows:

```
po_addrStrobe.write(true);
po_writeStrobe.write(true);
po_dataWrite.write(wData);
po_addr.write(addr);
```

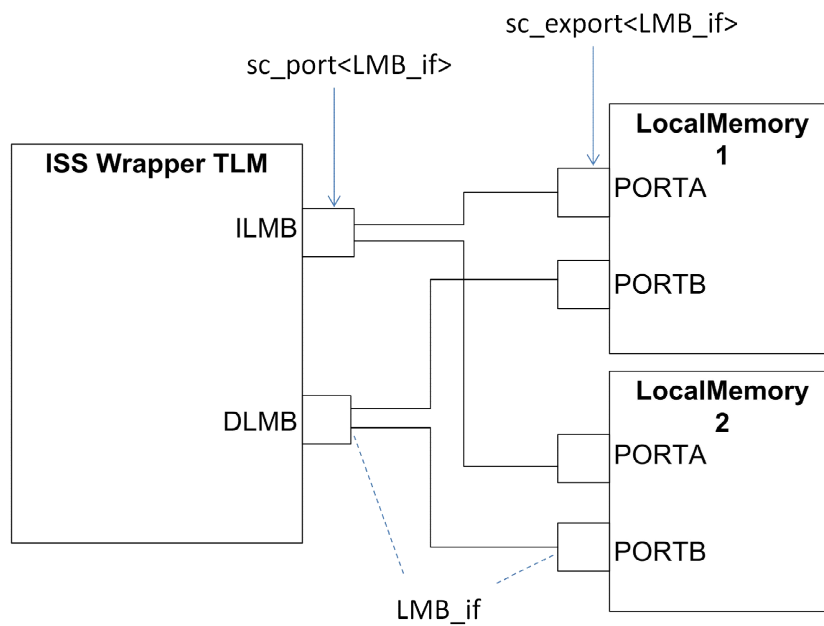

Figure 4.8: TLM system

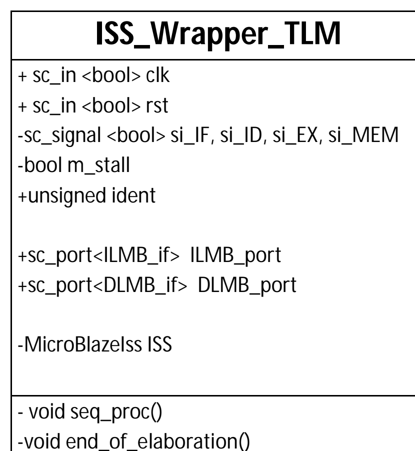

Figure 4.9: TLM ISS wrapper class model

The idea is to find a way to replace this whole code segment with a simple method call that carries out the same function such as:

```
PORTX->Write(wData, addr);
```

The method Write(data, addr) would then be defined inside the BRAM module in order to carry out the required function. However, this only solves half of the problem since the write transaction does not only consist of the the actual writing of the data in an address, but it also has a reply phase. After the BRAM accomplishes the required data access it assigns the "DReady" and "Data_Read" signals as a sign that the memory access is successful. So the other part for the transformation would be to find a way to translate such a code segment into TLM:

```
if(pi_DReady.read())
  iss.setDataResponse(pi_DRead.read());
```

4.4.1 TLM Memory Module

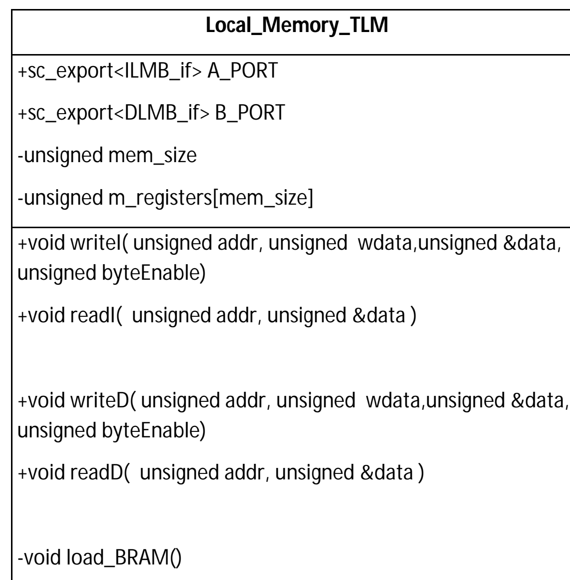

Figure 4.10: TLM memory class diagram

The TLM memory module mainly consists of the memory array, and some method definitions to carry out the read and write functions. There are two main points to take into consideration while designing the TLM communication system. The first point is to make sure that the same functionality of the RTL system is achieved, and the second point is to ensure the correct timing of that action.

A simple way to model the correct read and write functionality would be to provide a reference to a member variable inside the method call such as:

```
void read(unsigned address, unsigned &reply)
{
  reply = MEM[address];
}
void write(unsigned data, unsigned address, unsigned &reply)
{
  MEM[address] = data;
  reply = MEM[address];
}
```

The memory would then set the required value to the reply such that the ISS wrapper can read it. Such a solution would perform the correct function, however the time that the wrapper receives the reply would be earlier than the case of RTL modeling. Therefore a different implementation would be needed to achieve the correct timing, one idea to achieve this is to create a sequential process in the TLM memory module that checks if the methods have been accessed, and somehow return the reply to the wrapper by calling other methods such as void *Acknowledge(unsigned &reply)* that would be implemented in the wrapper class.

There is a property that can be used to reach a more efficient solution for modeling the correct TLM timing; at any TLM memory access, it is always certain that the method call has been done at a positive clock edge since it is called from the wrapper sequential process. This means that it is possible to model a memory without any sequential processes and have the methods abide with the timing requirements of the memory access. The idea is to store the reply in a temporary member variable during the method call for the memory access itself, and during the next method call, the reply would be provided for the ISS as follows:

```
void read(unsigned address, unsigned &reply)
{
  reply = read_tmp_reply;
  read_tmp_reply = MEM[address];
}
void write(unsigned data, unsigned address, unsigned &reply↩
   )
{
  MEM[address] = data;
  reply = write_tmp_reply;
  write_tmp_reply = MEM[address];
}
```

This however means that the wrapper must guarantee that another method call is made in the clock cycle following the memory request stage to achieve a complete memory access.

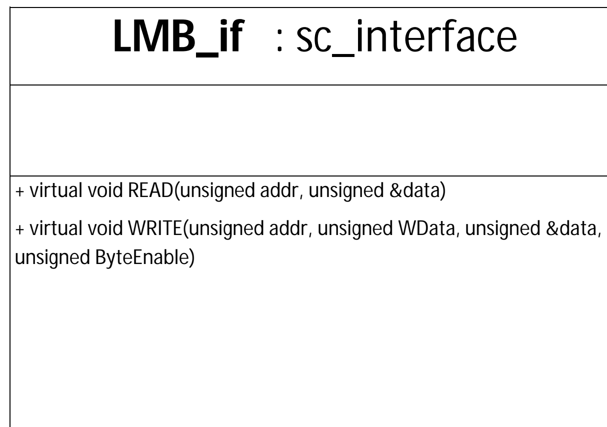

Figure 4.11: LMB_if class diagram

4.4.2 Local Memory Bus Interface (LMB_if)

The LMB interface class is responsible for providing the abstract methods that will be called by the processor and implemented inside the BRAM. It simply defines the method signatures for the required methods as shown in the class diagram in figure 4.11.

Chapter 5

Implementation

In this chapter, the final look of the designed system is demonstrated. It is divided into: basic implementation then adding complexities such as optimizing the pipeline structure, adding prefetch buffer, forwarding, stalling, branching and floating point operations.

5.1 RTL ISS Wrapper

As shown in the design chapter, the ISS wrapper sequential process is divided into a separate section for each pipeline stage, and each of those stages calls the respective ISS method to carry out the pipeline function in the correct timing with the correct inputs/outputs.

Listing 2.2 shows how to use the ISS in an untimed manner, the goal now is to somehow transform this basic implementation, such that each instruction undergoes each of the pipeline stages with a difference of one clock cycle between each of them. In other words, this means that registers can be placed between each ISS call such that those registers control the data-flow of the pipeline and provide a clock cycle difference between each pipeline stage.

5.1.1 Register Modeling

A register is the most important component in an RTL model, there are several ways to model a register in SystemC. The most straight forward approach would be to define a module *Register* that consists of *pi_enable* and *pi_input* input signals and *po_output* output signal. Internally there would be a process that gets invoked at every positive clock edge, inside this process an output signal can be assigned to an input value if the enable is true. This way the value of the output only changes at the clock positive edges; thus modeling a register.

```
//class Register : sc_module
void seq_proc() // SC_METHOD sensitive<<clk.posedge()
{
  if(pi_enable)
    po_output.write(pi_input);
}
```

This idea however can be inherited and used inside the other classes of the system to enable the usage of the concept of a register without having to have a separate entity for each intermediate register value needed. This is achieved using

sc_signals that are invoked inside a sequential process as shown in the background chapter.

Another way to model a register is to replace the sc_signals with the (simpler in terms of memory and simulation time) local member variables. Members variables get updated directly at the time of assignment, this means that modeling of the register will require a little more complexity in coding to make sure that the assignment occurs only at the end of the process after all the variable usages have been performed.

5.1.2 ISS Basic Timed Implementation

This section will show how to transform the untimed ISS operation into a correctly timed 5-stage pipeline. Each stage will be discussed separately and will later be interconnected together as blocks for simplicity.

5.1.2.1 Instruction Fetch

During IF, the target is to set the output signals to the instruction BRAM with the current PC. The PC is already available inside the ISS therefore the wrapper should first call the *getInstructionRequest(m_IValid, m_InstrAddr)* method in the ISS to find out whether there is a valid instruction request, and the required PC to be fetched. If the request is valid, the wrapper then sets the output ports accordingly. Otherwise, the address strobe and the IFetch signals should be set to false to indicate that no instruction is currently requested as shown in Listing 5.1.

Listing 5.1: Basic instruction fetch code

```
iss.getInstructionRequest(m_IValid, m_InstrAddr);
po_addrStr.write(m_IValid);
po_IFetch.write(m_IValid);
po_DataAddr.write(m_InstrAddr);
```

However, to apply this implementation, modifications should be made to the ISS; because the *getInstructionRequest()* originally just forwards the internal PC without incrementing it. This means that for the first three cycles of an application run, it will keep returning the same PC until the first instruction reaches the EX stage to update the PC. A way to solve this would be to define a new variable *fetch_pc* that is independent from the normal PC *r_pc*, and modify the *getInstructionRequest()* method so that it provides the wrapper with that *fetch_pc* and increment it by four at each IF call as shown in Listing 5.2 as is the case in any typical pipelined processor. In case a branch has been taken, the method detects that and returns the branch destination *r_npc* calculated inside the *step()* method. At the same time, it also synchronizes the *fetch_pc* to the same value and increments it by four again for the next instruction fetch.

Listing 5.2: Modified getInstructionRequest method code

```
void MicroBlazeIss::getInstructionRequest(bool &req, ↩
   uint32_t &address)
{
  req = true;
  if(m_branch)
  {
    address = r_npc;
    fetch_pc = r_npc;
  }
  else
    address = fetch_pc;
  fetch_pc = fetch_pc+4;
}
```

When a stall occurs, the IF should be stopped. Therefore, further conditions for actually accessing the IF stage are present, and will be discussed with the stall functionality in the optimizations section.

5.1.2.2 Instruction Decode

In any typical pipelined processor, ID occurs in a unit that is separate from the EX. However, the ISS performs the real decoding process inside the EX method call *step()*. For this reason, an identical modeling of the ID stage will not be possible using the given ISS, it would not be feasible to store the intermediate operands and opcodes in actual intermediate registers to be forwarded to the EX like the HW processor structure. The operands will be read from the GPR directly while the instruction is being executed, and the ALU result is also directly written into the GPR while EX, which is also different from the HW structure. Therefore, another way has to be found to model a one cycle delay between the IF and EX in the pipeline. Another method is offered by the ISS that can fit in this stage; *setInstruction(bool IFerror, unsigned Instr)* is used to provide this intermediate stage between IF and EX and will be considered to be the ID stage of our system, where we simply feed a new instruction into the ISS.

ID therefore checks for *pi_IReady* signal, which notifies the processor that there is a new reply from the instruction memory with a new instruction to be decoded. In the case where *pi_IReady* is true, ID feeds the instruction in *pi_Instr* to the ISS by calling *iss.setInstruction(false, pi_Instr.read())*.

Listing 5.3: Instruction Decode code segment

```
if(pi_IReady.read())
{
  iss.setInstruction(false,pi_Instr.read());
  si_ID.write(true);
}
else
  si_ID.write(false);
```

5.1.2.3 Instruction Execution

On the ISS wrapper side, EX is quite simple. The whole idea is to check if there is a valid decoded instruction, and if this is true, iss.step() is called and the flag *m_EX* is set to true to declare that a valid EX stage has occurred so the MEM can start. Otherwise both flags are set to false.

Listing 5.4: Basic EX stage code segment

```
if(si_ID.read())
{
  m_EX = true;
  iss.step();
}
else
  m_EX = false;
```

5.1.2.4 Memory Access

MEM stage first checks if a valid EX had occurred, if so it calls the *getDataRequest()* method to see if a memory access is required and then sets the memory access signals accordingly. The *m_DType* defines whether the instruction request is a read or a write, and whether it is for a word, byte or a half word to select the Byte Enable(BE) signals. BE is a four bit value that selects which parts of the memory word are accessible, there are three options for the BE setting: 1000 means that the target is the most significant byte in the destination address, 1100 targets the most significant half word (two bytes) and 1111 targets the whole word.

```
  iss.getDataRequest(m_DReqValid, m_DType, m_DAddr, m_WData↩
     , m_r_mem_dest);
  if(m_DReqValid)
  {
    po_D_AS.write(true);
    po_Data_Addr->write(m_DAddr);
    po_Data_Write->write(m_WData);
    switch (m_DType)
    {
      case WRITE_BYTE :
        po_Byte_Enable->write(0x8); //0xC for WRITE_HALF, ↩
            and 0xF for WRITE_WORD
        po_Read_Strobe->write(false);
        po_Write_Strobe->write(true);
        break;
      .
      .
      case READ_BYTE:
        po_Byte_Enable->write(0x8); //0xC for WRITE_HALF, ↩
            and 0xF for WRITE_WORD
        po_Read_Strobe->write(true);
```

```
        po_Write_Strobe->write(false);
        break;
      .
      .
    }
  }
  else
  {
    po_D_AS.write(false);
  }
}
```

5.1.2.5 Write Back

WB continuously checks for a *pi_DReady* signal from the memory. When it is true, the *setDataResponse()* method is called using the data from the *pi_DRead* signal which contains the actual data returned from the BRAM.

```
if(pi_DReady.read())
{
  iss.setDataResponse(pi_DError.read(), pi_Data_Read.read()↩
      , si_DType.read(), si_r_mem_dest.read());
}
```

5.1.3 Adding Complexities

5.1.3.1 Address Decoding

The address decoding idea is quite simple, the goal is to check the address that should be accessed, and determine which slave (BRAM) to forward the correct memory access signals to. To achieve this, the wrapper can get the important details like the number of slaves connected to it, and the starting and ending addresses of each of those slaves. Then, whenever a memory access is requested, the wrapper compares the address with the list of high and low addresses of each slave to see where it should be forwarded. The chosen slave number has to be somehow be stored and maintained for at least 1 clock cycle because the slave will be replying in the following clock cycle, and the wrapper needs to know which slave it should be reading from. As shown in listing 5.5, the wrapper iterates on all the slaves to match the current requested address *m_InstrAddr* with one of the slaves, and sets the required slave to a variable *m_req_slave* and a signal *si_req_slave*, the variable is used right away to set the address strobe signal to the respective BRAM, and the signal will be used in the following clock cycle during ID to select the correct "Data Ready" to wait for. This means that all the output address strobes and input data ready's and data read's are not single ports but are array of ports (multiports) to enable simple address decoding.

Listing 5.5: Address decoding for instruction memory accesses

```
for(int i= 0; i< m_Num_Slaves ; i++)
{
  if(m_InstrAddr <= m_HighAddr[i] && m_InstrAddr >= ↩
     m_LowAddr[i])
  {
    m_req_Islave = i;
    si_req_Islave.write(i);
  }
}
po_IFetch->write(true);
po_I_AS[m_req_Islave]->write(true);
po_Instr_Addr->write(m_InstrAddr);
```

5.1.3.2 Pipeline Optimization

Typically, the wrapper would normally be in the traditional ordering of the pipeline with the IF at the beginning and the WB at the end. However, if this is done, the processor will not function correctly. The reason for this is that for example if the ID is done before the EX, then at each clock cycle, the ISS will get the instruction decoded and executed in the same clock cycle while the previous instruction might not be executed at all. Therefore for correct functionality of the pipeline using the given ISS, the pipeline has to be in a reversed order: WB → MEM → EX → ID → IF. This way, the instruction decoded in a clock cycle, will be executed in the following clock cycle, and so on for all the pipeline stages. However, another modification is needed because by monitoring the waveforms of the VHDL systems, it turns out that the EX and MEM for each instruction occur in the same clock cycle, i.e. the memory access signals are activated at the clock edge where the instruction is actually executed. For this reason, the link between EX and MEM stages would have to be altered to match the VHDL behavior, in this case the signal declaring that a valid instruction has been executed can be a member variable *m_EX* instead of a signal *si_EX*, and the MEM stage would be placed at the end of the code for simplicity. Therefore, the correct order is actually WB → EX → ID → IF → MEM

5.1.3.3 Prefetch Buffer

The MicroBlaze processor contains a prefetch buffer, which simply resembles a 16-Byte (4-instruction) wide FIFO channel. This FIFO channel can be located in the ID stage, which means that the ID will be divided into two separate stages: pushing the fetched instruction into the prefetched bufffer, and the other stage will perform the actual decoding; this stage will check if the FIFO has any instructions, if so then the instruction at the beginning of the FIFO is decoded using the *iss.setInstruction* method. Both stages need to be completely separable because they will need to be placed at different parts of the code when the stalling mechanism is introduced.

```
if(pi_IReady[si_req_Islave]->read())
{
  fifo.push(pi_Instr[si_req_Islave]->read());
}
if(!fifo.isEmpty())
{
  iss.setInstruction(pi_IError[si_req_Islave]->read(),↩
     m_Instr);
  si_ID.write(true);
}
else
  si_ID.write(false);
```

5.1.3.4 Branch Handling

The decision of whether a branch is taken or not is done when the branch instruction is in the EX stage. This means that in the case where the branch is taken, two unwanted instructions are already in the pipeline stages ID and IF. The instruction inside the ID is automatically flushed inside the ISS because the step function is designed to detect that a branch (with no delay slot) is taken, and sets a flag *m_cancel* that cancels the EX of the following instruction by not returning from the step function before the actual execution as shown in Listing 5.6.

Listing 5.6: Part of the step() function responsible for flushing the instruction following a taken branch

```
void MicroBlazeIss::step(void)
{
  if (m_cancel) {
    m_cancel = false;
    r_pc   = r_npc;
    r_npc  = r_pc + 4;
    return;
  }//rest of the step() method
}
```

The other instruction that needs special handling is therefore the instruction that is already in the IF and waiting to be decoded. The goal is to find a way to detect that this instruction is an unwanted instruction and therefore disable the ID at that clock cycle; thus flushing the instruction. One way to detect that this is an unwanted instruction is to use the address of each instruction; typically (with no branches) the addresses should be consecutive which means that if the instruction currently in the EX stage has address x, then the instruction in ID should have address $x + 4$ and that in the IF will have address $x + 8$. However, when a branch occurs, the order of addresses is altered. This means that the ID will for example contain an instruction with address x and the current PC that needs to be fetched will show the branch target which is something that is not $x + 4$. This technique

can therefore be used to detect that a branch had occurred, the idea is therefore to keep track of the PC from the last clock cycle in a variable *m_old_pc* for example, and compare it with the current PC *m_InstrAddr* that is currently used by the IF and see if *m_InstrAddr* is not equal to *m_old_pc + 4*. If this is true, a flag is declared to notify the ID to not accept the instruction that is currently being forwarded from the IF stage. The code would then be :

```
// in ISS_Wrapper class
   m_flush_branched = (m_InstrAddr!=(m_old_pc + 4));
```

This approach however has a flaw that could cause it to fail in some cases; namely when a branch is taken, but the branch target points to 3 instructions after the branch instruction itself such as *Bri 12*. Such an instruction would cause consecutive instruction fetches to occur, because assuming the *Bri* instruction is at address x, instructions $x + 4$ and $x + 8$ will be fetched like the case in any other branch instruction. The difference here is that the next instruction to be fetched will be the branch target itself which is $x + 12$. This means that *m_old_pc* will be equual to $x + 8$ and *m_InstrAddr* will be equal to $x + 12$ so *m_flush_branched* would remain false. This would lead to the instruction at $x + 8$ mistakenly being forwarded to the ID and the rest of the pipeline stages, although it should have been flushed. A possible solution to work around this single case exception would be to introduce a new flag to the ISS which is set to true when a branch is taken with a branch target *next_pc* equal to *r_pc + 12*.

```
// in ISS class
   p_m_branch_anyway = branch && (next_pc == r_pc + 12);
```

Finally, the ID checks for these two flags (*m_branch_anyway* and *m_flush_branched*) and if either of them is true, ID is not accessed and *si_ID* is set to false so that a bubble inside the pipeline is modeled.

5.1.3.5 Forwarding and Stalling

Normally, forwarding occurs in a processor when the up-to-date value of one of the operands for an instruction is in one of the pipeline registers instead of the GPR. In this case the EX has to take this up-to-date value instead of the value provided by the ID. A typical example that would cause this case to occur is:

ADD R2, R1, R0
ADD R3, R2, R5

Due to the architecture of the ISS, forwarding does not need any extra complexity since the GPR gets updated directly after each instruction execution. Therefore, a destination operand can be accessed using the following instruction or any other one afterwards.

The other data dependency that does need special handling is the data dependency of a Load instruction destination for example:

LWi R1, R0, 0
ADD R3, R1, R2

In this case, there is no way the ADD instruction can get the correct value for operand R1 in the cycle following the LW EX cycle. Therefore, a stall has to occur; a stall is when the initial pipeline stages (IF, ID and EX) are halted until a data memory access has been completed. The straight forward approach to implement the stall technique would be to detect the dependency during the ID phase (of the ADD instruction in this example). At this clock cycle, the LW will have been executed, and the ADD will have been already decoded. In the next clock cycle however, the IF, ID and EX stages should be halted. Therefore, an if statement can combine the three stages and check for the *si_Dependency* signal, if it is true, then those stages would not be accessed to give a chance for the memory access to be completed. However, a problem arises when this technique is implemented in comparison with the VHDL reference system. For some reason, the stall functionality occurs just as discussed here, but the IF halt is delayed by one clock cycle. When the dependency is detected, ID and EX are stalled at the following clock cycle, but the IF is stalled two cycles after that. This needed to be handled manually, so another sc_signal (*si_was_stalled*) was defined, which is a one-cycle delayed version of the *si_Dependency* signal. The IF then uses this *si_was_stalled* to check for stalls.

Listing 5.7: Stalling functionality

```
if(!si_Dependency.read() )
{
  // Normal INSTRUCTION EXECUTE as in listing 5.4
  //INSTRUCTION DECODE
  m_flush_branched = (m_InstrAddr!=(m_old_pc + 4));
  si_ID.write(false);
  if(!fifo.isEmpty() && !m_flush_branched && !↩
     m_local_branch_anyway)
  {
    m_Instr = fifo.view();
    if(((m_Instr & 0xD0000000) == 0xC0000000) // Load ↩
       Instruction
    {
      si_Load.write(true);
      si_Load_destination.write((m_Instr & 0x03E00000)>>21)↩
        ; // put Rd in the least signif.
    }
    else
      si_Load.write(false);
    if(si_Load.read())
    {
      RaDep =  (((m_Instr & 0x001F0000) >> 16) == ↩
        si_Load_destination);
      RbDep = ((((m_Instr & 0x0000F800) >> 11) == ↩
        si_Load_destination) && !(m_Instr & 0x20000000));
      SwDep = ((((m_Instr & 0x03E00000) >> 21) == ↩
        si_Load_destination) &&  ((m_Instr & 0xD0000000) ↩
```

```
          == 0xD0000000));
      if(RaDep || RbDep || SwDep)
        si_Dependency.write(true);
    }
  }
}
else
  si_Dependency.write(false);

//Instruction Fetch
if(!si_was_stalled.read())
{
  iss.getInstructionRequest(m_IValid, m_InstrAddr);
  if(m_IValid)
  {
    po_DataAddr.write(m_InstrAddr);
    po_addrStr.write(true);
    po_IFetch.write(true);
  }
  else
  {
    po_addrStr.write(false);
    po_IFetch.write(false);
  }
}
si_was_stalled.write(si_Dependency.read());
```

The other exception that needs to be manually handled is the case of double stalls, where two consecutive Load instructions are dependent on each other followed by another dependency such as:

LWi R1, R0, 0
LWi R2, R1, 4
ADD R4, R2, R3

Handling this case gets rather complex because the VHDL system response to that case is rather strange. As an example, Figure 5.1 demonstrates a waveform segment of the ILMB and DLMB signals of the VHDL system while executing the following instructions:

```
1b4: e82006b4  lwi   r1, r0, 1716
1b8: e8410000  lwi   r2, r1, 0
1bc: 20620001  addi  r3, r2, 1
1c0: 10600000  addk  r3, r0, r0
1c4: 10330000  addk  r1, r19, r0
1c8: ea610004  lwi   r19, r1, 4
```

When applying typical forwarding and stalling concepts such as those discussed in [12], handling of such a sequence of instructions is expected to be different than what is shown in Figure 5.1. Figure 5.2 shows a typical scheduling technique for

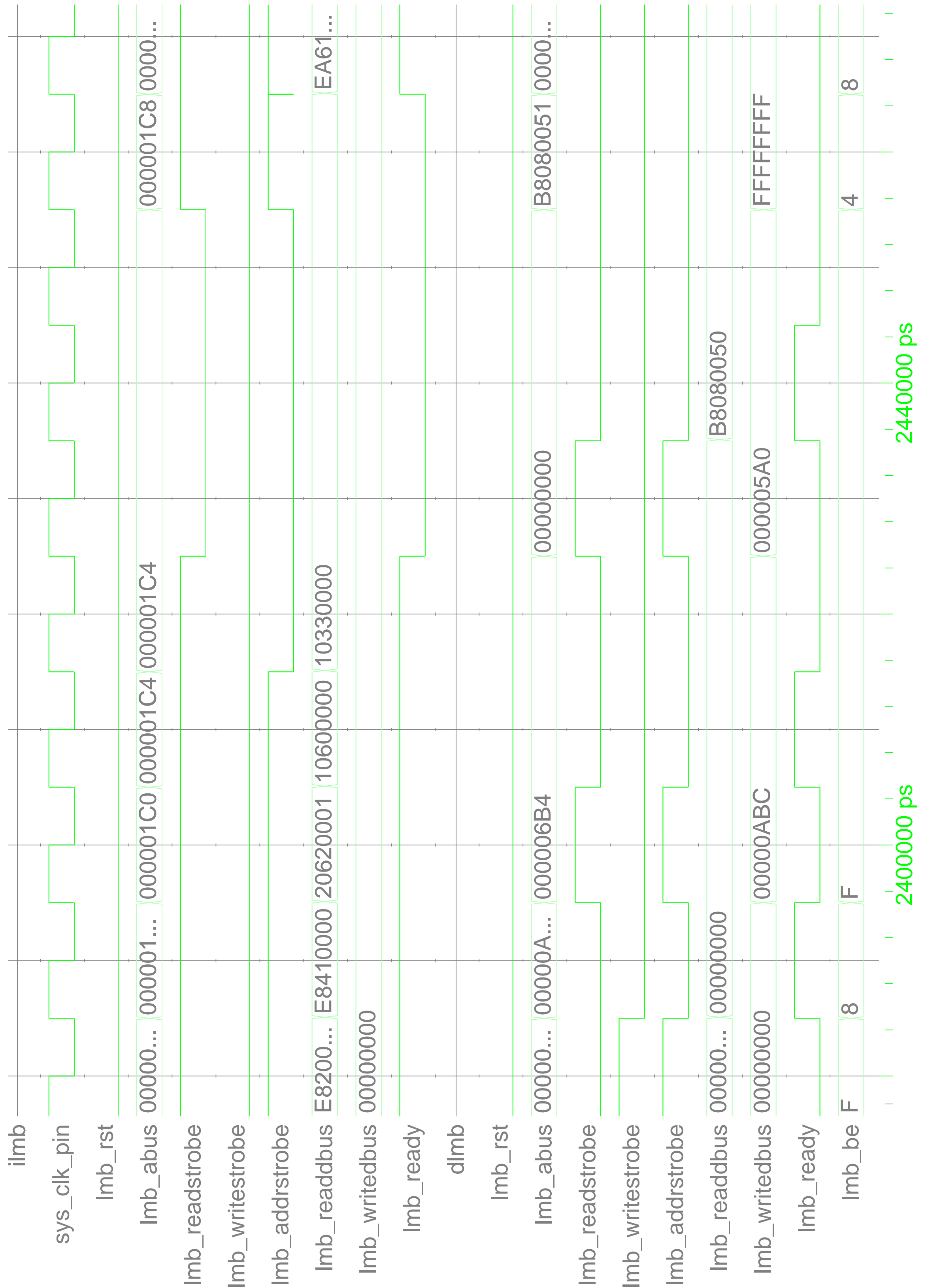

Figure 5.1: VHDL waveform of the double stall case

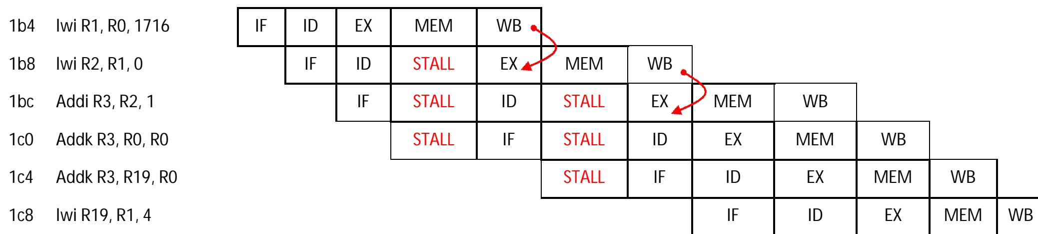

Figure 5.2: Pipeline scheduling for a double stall case showing the forwarding procedure

dealing with the double stall case, where each dependent instruction delays its EX until the WB of the previous instruction is complete. This scheduling however is inconsistent with the waveforms shown in Figure 5.1. The inconsistencies are:

1. First, the start of the stall is shown by canceling the *lmb_addrstrobe* of the ILMB (top section) at around 2410 ns after fetching instruction 1C4, unlike Figure 5.2 where the stall is declared after fetching instruction 1CO. This case has been mentioned before in the single stall case, where it was shown that the IF gets stalled one clock cycle after the ID and EX have been stalled.

2. The second difference is that the stall is continuous; for four clock cycles there has been no instruction fetched, although it should be expected that sometime during the stall, the first dependent instruction *lwi r2, r1, 0* at 1B0 has been executed so the EX unit becomes free. This means that the pipeline can proceed for at least one clock cycle meaning that a new instruction can be fetched as shown in Figure 5.2. However, this is not the case, and the stall state is continuous.

3. The third difference is that the stall takes four clock cycles. This can be justified from the waveforms by observing the DLMB (bottom section) signals. It is shown in the waveforms that the first (independent) memory access is initiated at around 2390 ns by assigning the address 6B4 to the DLMB *lmb_addrbus* and setting the read strobe and address strobe to true. In the following clock cycle, the response arrives (shown by true *lmb_ready*). In the typical case, forwarding should now occur to provide the EX of instruction 1B8 with the result of WB of instruction 1B4 to be used as an address for the new memory access as demonstrated in 5.2. Nevertheless, the waveforms show a cycle difference between the response of instruction 1B4 and the request of 1B8. This can be explained by a different concept of forwarding, where the WB result of 1B4 is forwarded to the ID of 1B8. Applying this approach can explain this behavior as demonstrated in Figure 5.3 which performs the forwarding to ID instead of EX. Even though using this forwarding explains this cycle difference, inconsistencies 1 and 2 are still available between the scheduling in Figure 5.3 and the waveforms in Figure 5.1.

Therefore, since it is obviously quite complex to reach a scientific solution to match the VHDL behavior, the behavior of the double stall case is modeled as a special case with some extra flags to explicitly carry out the same behavior as the VHDL model. This is achieved in several stages. First, a new signal *si_was_stalled2*

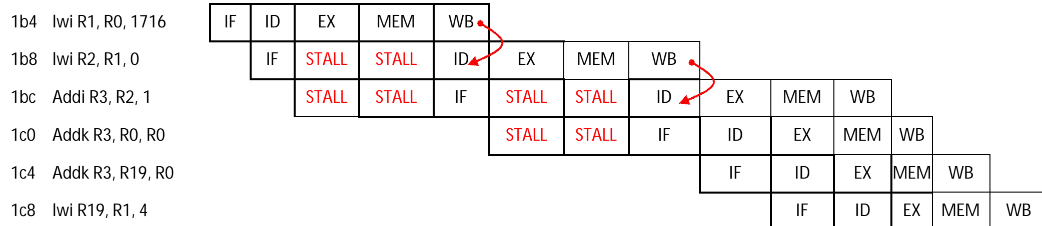

Figure 5.3: Pipeline scheduling for a double stall case showing a different forwarding handling technique

is defined that gets its value from *si_was_stalled*. This means that *si_was_stalled2* signal is a 2-cycle delayed version of *si_Dependency* stalling signal. Whenever a new dependency is detected, *si_was_stalled2* is checked and if it is true, this would mean that this is a double stall case so a flag *m_double_stall* is set to true aside from the normal stall procedure. This *m_double_stall* is then used to extend the stall with one extra clock cycle, and at the same time disable the intermediate fetching process discussed above in order to match the same VHDL behavior.

5.1.3.6 Floating Point Unit (FPU)

The main problem faced while modeling a floating point unit is how to handle the timing properties. Most of the floating point instructions take more than one clock cycle to be executed. This means that the pipeline structure needs to modified to accommodate this difference in execution. The other problem, is that the available ISS does not handle floating point instructions, so they first need to be added to the ISS before dealing with the wrapper to fix the pipeline operation.

The MicroBlaze FPU is based on the IEEE 754-1985 standard [31]. It uses the single precision floating point format; which consists of:

1. One sign bit (bit 0).
2. Eight bits for a biased exponent (bits 1 to 8).
3. 23 bits for the fraction (mantissa) part (bits 9 to 31).

The FPU register values can define infinity, Not-a-Number(NaN), zero and normal floating point values. A register value v can be interpreted as follows:

- If *exponent* = 255 and *fraction* <>0, then v= NaN, regardless of the *sign* bit.
- If *exponent* = 255 and *fraction* = 0, then v= $(-1)^{sign} * \infty$.

- If 0 <*exponent* <255, then v = $(-1)^{sign}$ * 2(*exponent*-127) * (1.*fraction*).
- If *exponent* = 0 and *fraction* <>0, then v = $(-1)^{sign}$ * 2^{-126} * (0.*fraction*).
- If *exponent* = 0 and *fraction* = 0, then v = $(-1)^{sign}$ * 0.

The instructions supported by the MicroBlaze FPU are: addition, subtraction, multiplication, division, comparison, conversion and square root instructions. A pseudo code is provided in [31] for each instruction separately and is simply translated to SystemC and placed in the switch statement inside the *ISS::step()* function. The actual operation (addition for example) is easier to be done in the C/C++ *float* data type. Therefore, the operands from the GPR are first converted to *float* data types, then the operation is performed to produce another *float* type result which is finally converted back to the single precision *unsigned* value to be stored in the GPR. Transformation between the unsigned data type of the GPR to the floating point values is done using the type casting *reinterpret_cast* command as follows:

```
float flt_Rb = *reinterpret_cast <float*> (&r_gpr[ins_rb]);
unsigned uint_Rb = *reinterpret_cast <unsigned*> (&flt_Rb);
```

As mentioned above, execution times vary from one floating point instruction to the other. The ISS wrapper needs be notified with the instruction being executed in order to fix the number of cycles which the pipeline will be halted for. The idea is to set a public variable *unsigned floatIns* inside the ISS, which can be checked by the ISS wrapper after each call for the *step()* function. The value of *floatIns* is normally set to zero, which resembles that a normal single-cycle instruction has been executed. Otherwise a value other than zero is set. Execution times for floating point instructions can be divided into four collective groups:

1. 4-cycle instructions: Includes most of the instructions like: FADD, FRSUB, FMUL and FLT.
2. 5-cycle instructions: Includes FINT instruction.
3. 27-cycle instructions: Includes FSQRT instruction.
4. 28-cycle instructions: Includes FDIV instruction.
5. 1-cycle instructions: Includes FCMP instruction.

Group five needs no special assignment to *floatIns* since it can be considered as normal non-floating point instruction, so the *floatIns* remains zero. For the rest of the groups, *floatIns* is set to a different value for each group (one to four for example) to notify the wrapper which delay time to enforce.

By this the ISS part for handling the FPU is complete, the rest of the functionality is achieved by the wrapper. The idea now is check the *iss.floatIns* at the end of each EX stage and act accordingly. Listing 5.8 demonstrates how it is done, an extra case is added other than the four groups shown above, this case is for the integer divide (IDIV) instruction that takes 32 cycles for execution and is handled

the same way as a floating point instruction. The goal of this code segment is to set a flag *float_stall* to true that performs a similar function as *si_Dependency* which is stopping the ID and EX stages. An unsigned variable *m_float_delay* is also set (based on the type of the instruction), which acts as a counter for the stalling process. This means that every clock cycle the pipeline is stalled due to the *float_stall* flag, this counter is decreased by one. When the counter finally reaches zero, the pipeline is resumed by setting *float_stall* to false as demonstrated in Listing 5.9.

Listing 5.8: Detection of the floating point instructions in the ISS wrapper

```
//normal EX stage code
iss.step();
if(iss.floatIns>0)
{
  float_from_stall = m_was_stalled2.read();
  float_dest = iss.m_ins_rd;
  float_stall = true;
  switch(iss.floatIns)
  {
  case 1 :
    m_float_delay = 4;
    break;
  case 2 :
    m_float_delay = 28;
    break;
  case 3 :
    m_float_delay = 5;
    break;
  case 4 :
    m_float_delay = 27;
    break;
  case 5 : //IDIV instruction
    m_float_delay = 32;
    break;
}
```

Typically, no modifications should be done to the IF stage since the prefetch buffer should keep fetching until it is full. However, as is the case with the normal stalls discussed earlier, the IF in the VHDL behaves in a strange way when the stall is over, and this is handled explicitly using the *m_neglect_fetch_float* flags that control when the IF starts fetching again to exactly match the VHDL behavior.

Listing 5.9: Handling the float stall case

```
if(!si_Dependency.read() && !float_stall )
{
  // EX stage
  // ID stage
}
else
{
```

```
  if(si_Dependency)
    si_Dependency.write(false);

  if(float_stall)
  {
    m_float_delay --;
    float_stall = (m_float_delay > 0);
    if(m_float_delay == 0)
    {
      float_stall = false;
      m_neglect_fetch_float = true;
      if(!float_from_stall)
      m_neglect_fetch_float2 = true;
    }
  }
}
```

5.2 RTL Memory Module

The memory contains two almost identical ports to separate between instruction memory access (PORT A) and data memory access (PORT B). At each clock edge, the memory checks for address strobes. If a high address strobe is detected, it checks the read and write strobes for that port. If it is a read request, the memory then returns the value of the memory register given by the address signal. On the other hand if it is a write request, then the memory writes into the memory address given by pi_addr the value given by pi_WData, then it gives as an output the content of that same memory address.

```
#define enable_8  0xFF000000
#define enable_16 0xFFFF0000
// sequential process
if(PORTA_AddrStrobe->read())
{
  if(PORTA_WriteStrobe->read())
  {
    tmp_reg = m_registers[PORTA_Abus->read()];
    switch(PORTA_BE->read())
    {
      case(0x8):
        tmp_reg = (tmp_reg & ~enable_8) | (PORTA_WDbus->↩
            read() & enable_8);
        break;
      case(0xC):
        tmp_reg = (tmp_reg & ~enable_16) | (PORTA_WDbus->↩
            read() & enable_16);
        break;
      case(0xF):
        tmp_reg = PORTA_WDbus->read();
```

```
        break;
      }
      m_registers[PORTA_Abus->read()] = tmp_reg;
    }
    PORTA_Dout->write(m_registers[PORTA_Abus->read()]);
    PORTA_DREADY->write(true);
}
else
    PORTA_DREADY->write(false);
//exact code repeated for PORTB
```

5.2.1 BRAM Initialization

For BRAM initialization, a systematic way to transform the (.elf) file application written using the EDK tool into a SystemC array of instructions is to be found. Upon compilation, EDK automatically produces (.elf) files from the *main.c* C-file containing the application. XPS then uses that (.elf) file to develop a VHDL initiation file that initializes the VHDL BRAMs. A similar approach can be made to transform the (.elf) file into a simple series of instructions and corresponding addresses using the Data2MEM command. This command has the capability to produce a (.mem) file looking like the output shown in Listing 5.10. As shown in the listing, the (.mem) file is divided into several records. At the beginning of each record is the symbol '@' followed by the first address in that record. In the following line the instructions are written consecutively in hexadecimal format with a space between each byte.

Listing 5.10: (.mem) file used for BRAM initialization

```
// Program header record #0, Size = 0x4, at 0x00000000 to 0↩
    x00000003.
@00000000
    B8 08 00 50
// Program header record #1, Size = 0x4, at 0x00000008 to 0↩
    x0000000B.
@00000008
    B8 08 01 D4
.
.
.
// Program header record #4, Size = 0x54A, at 0x00000050 to↩
     0x00000599.

@00000050
    31 A0 06 B8 30 40 05 A0 30 20 0A D0 .....
```

The goal now is to read this (.mem) file, separate each instruction on its own and calculate the respective address of that instruction to store in in the BRAM memory array. What also needs to be taken into consideration is that all the information

extracted from such a file are strings, so they need to be converted accordingly to unsigned data types.

5.3 TLM Memory Module

As mentioned in the Design chapter, the idea is to replace the sequential process and in/out ports with method calls to carry out the same functionality with the same timing. It is also mentioned that some kind of register is modeled inside those methods, such that when a request is made, the correct reply is only available at the following method call (following clock cycle) in order to model the exact timing of the RTL model. What also needs to be correctly timed, is when exactly the data is written in the memory array by the *write()* function. In the RTL model, the wrapper sets the write strobe and write data signals, and they are received and used by the BRAM only in the following cycle. Therefore, to achieve a CA TLM model, the actual writing to the BRAM content in TLM cannot be done inside the *write()* method itself, but in the clock cycle following the *write()* call. A way to do this would be to set a flag inside the *write()* that declares that a write has been requested, and keep the memory access details (address and write data) in some shared variables. In the following cycle when *read()* is called, the actual update of the memory array is done, and the result is forwarded back as a reply to the ISS wrapper to have the exact timing behavior as the RTL model.

```
void local_mem_TLM :: writeD( uint32_t addr, uint32_t  ↩
    wdata,uint32_t &data, uint32_t byteEnable)
{
  m_writingD = true;
  m_writingDAddr = addr;
  data = m_D_data;
  switch(byteEnable)
  {
    case(0x8):
      m_writingDData = (m_registers[addr] & ~enable_8) | (↩
          wdata & enable_8);
      break;
    case(0xC):
      m_writingDData = (m_registers[addr] & ~enable_16) | (↩
          wdata & enable_16);
      break;
    case(0xF):
      m_writingDData=wdata;
      break;
  }
  m_D_data = m_writingDData;
};

void local_mem_TLM :: readD( uint32_t addr, uint32_t  &data↩
    )
```

```
{
  if(m_writingD)
  {
    m_writingD = false;
    m_registers[m_writingDAddr] = m_writingDData;
  }
  data = m_D_data;
  m_D_data = m_registers[addr];
};
```

5.4 TLM ISS Wrapper

The main difference between TLM and RTL systems is changing how the processor communicates with the memory. So the goal to accomplish the transformation is to maintain the same functionality and timing while replacing the output port signal assignments with read and write method calls.

5.4.1 Instruction Fetch

The RTL IF is needs three clock edges to be completed, in the first clock edge, the wrapper gets the PC, and forwards it to the memory with the correct strobe signals. In the second clock edge, the BRAM gets the strobes and address, and sets the reply signals containing the requested instruction. Finally, in the third clock edge, the wrapper can read that instruction and use it for ID. To model this in TLM we will need to explicitly define the three stages. In the first stage the *readI()* method is called to notify the BRAM of the requested address. In the second stage, simulation of the RTL BRAM reply is achieved by calling *readI()* again with a dummy address just to get the reply from the request of the previous cycle. This reply is then stored in an sc_signal *si_tlm_instr* which would act as an exact model for RTL's *pi_Instr*. In the third stage, this instruction will be fed into the prefetch buffer to be used in that same clock cycle for ID.

```
// IF stage 3 : pushing the instruction into the prefetch ↩
   buffer
if(m_tlm_instr_requested)
{
  fifo.push(si_tlm_instr.read());
  m_tlm_instr_requested = false;
}

//IF stage 2 : getting the reply
if(si_IF.read())
{
  ILMB_port->readI(0,m_tlm_instr);
  si_tlm_instr.write(m_tlm_instr);
  m_tlm_instr_requested = true;
}
```

```
else
  m_tlm_instr_requested = false;

//IF stage
iss.getInstructionRequest(m_IValid, m_InstrAddr);
if(m_IValid)
{
  PORTA.readI(m_InstrAddr, m_tlm_Instr);
  si_IF.write(true);
}
else
  si_IF.write(false);
```

5.4.2 Memory Access

The idea of MEM stage transformation is similar to the IF, the goal is to divide the procedure into different stages. Stage one gets the memory request properties from the ISS and calls *readD()* or *writeD()* with the respective memory access details. The second stage calls *readD()* again using a dummy address to notify the BRAM to actually do the required action requested in the previous call, and then get the reply. The reply is stored in an sc_signal to be used in the WB stage.

```
// MEM stage 2
if(si_MEM.read() )
{
  DLMB_port[si_req_Dslave]->readD(0,m_tlm_data);
  si_tlm_data.write(m_tlm_data);
  m_tlm_data_requested = true;
}
else
  m_tlm_data_requested = false;


// MEM stage 1
iss.getDataRequest(m_DReqValid, m_DType, m_DAddr, m_WData, ↩
   m_r_mem_dest);
if(m_DReqValid)
{
  si_MEM.write(true);
  switch (m_DType)
  {
    case WRITE_BYTE :
      DLMB_port[m_req_Dslave].writeD(m_DAddr, m_WData, ↩
         m_tlm_data, 0x8);
      break;
    .
    .
    case READ_BYTE:
      DLMB_port[m_req_Dslave].readD(m_DAddr,m_tlm_data);
```

```
        break;
      .
      .
    }
  }
```

5.4.3 Write Back

WB simply checks *m_tlm_data_requested*, and if it is true the data in *si_tlm_data* signal is fed to the ISS along with the memory request details.

```
if(m_tlm_data_requested)
  iss.setDataResponse(false, si_tlm_data.read(), si_DType.↩
     read(), si_r_mem_dest.read());
```

5.5 Interface Adapters

As shown in the literature review section, one way to verify TLM models is to connect them to Transactors that transform the TLM interface into RTL signals to facilitate the comparison with a reference RTL system. This concept can be adopted in our system to enable more accurate testing capabilities as well as offering more usability for both RTL and TLM systems by making it possible to connect different modules from different abstraction levels in a single system. To achieve this, adapters can be designed that are capable of transferring the TLM methods into RTL signals, and also RTL signals into TLM methods. Therefore the possible adapter modules that can be created would be:

- RTL-TLM: to connect an RTL wrapper instance to a TLM BRAM.
- TLM-RTL: to connect a TLM wrapper instance to an RTL BRAM.
- TLM-TLM: this adapter is only used for testing purposes as a mean to regenerate RTL signals from TLM systems.

5.5.1 RTL-TLM Adapter

The function of this adapter is to translate the memory access requests provided by the ISS wrapper in the form of signal ports into method calls at the TLM memory side. The code demonstrated in Listing 5.11 is similar to the RTL memory module functionality. The difference is that instead of accessing its own memory array, it calls TLM methods to the TLM memory module.

Listing 5.11: RTL-TLM adapter code

```
//sequential process
if(PORTB_AddrStrobe->read())
  if(PORTB_WriteStrobe->read())
  {
```

```
    BRAM_B_PORT->writeD(PORTB_Abus->read(), PORTB_WDbus->↩
       read(), tmp_RData, PORTB_BE->read());
    BRAM_B_PORT->readD(PORTB_Abus->read(), tmp_RData);
    PORTB_Dout->write(tmp_RData);
    PORTB_DREADY->write(true);
  }
  else
    if(PORTB_ReadStrobe->read())
    {
      BRAM_B_PORT->readD(PORTB_Abus->read(), tmp_RData);
      BRAM_B_PORT->readD(PORTB_Abus->read(), tmp_RData);
      PORTB_Dout->write(tmp_RData);
      PORTB_DREADY->write(true);
    }
```

5.5.2 TLM-RTL Adapter

This adapter translates TLM method calls from the ISS wrapper, to RTL signal assignments for the RTL memory. Listing 5.12 is similar to the TLM memory module, but instead of accessing local registers, it forwards the requests in the form of RTL signals to the RTL memory module.

Listing 5.12: TLM-RTL adapter code

```
void writeD( uint32_t addr, uint32_t  wdata,uint32_t &data,↩
    uint32_t byteEnable)
{
  returningD = true;
  po_Data_Addr->write(addr);
  po_Byte_Enable->write(byteEnable);
  po_Data_Write->write(wdata);
  po_D_AS->write(true);
  po_Write_Strobe->write(true);
  po_Read_Strobe->write(false);
}

void readD(  uint32_t addr, uint32_t &data )
{
  if(returningD)
  {
    returningD = false;
    po_D_AS->write(false);
    po_Write_Strobe->write(false);
    po_Read_Strobe->write(false);
    data = pi_Data_Read->read();
  }
  else
  {
    returningD = true;
    po_Data_Addr->write(addr);
```

```
    po_D_AS->write(true);
    po_Write_Strobe->write(false);
    po_Read_Strobe->write(true);
  }
}
```

5.5.3 TLM-TLM Adapter

As mentioned above, the TLM-TLM Adapter is only used for testing purposes to verify the TLM system against the RTL system by regenerating the RTL signals between the TLM processor and TLM memory. The code for this adapter is quite simple; whenever a TLM method is called by the wrapper, it is forwarded exactly the same way to the memory. Moreover, local dummy RTL registers are assigned the same way as the TLM-RTL adapter. However, an extra problem needs to be handled in this adapter that was not available in the previous ones; since there is no sequential process inside this adapter, the *TLM_DReady* response signals get assigned to true whenever a memory response is available, and typically they should remain true for only one clock cycle. However, having only TLM methods means that there is no way that the adapter can know when to set the *TLM_DReady* signals back to false. Therefore an extra method *cancelReadys()* is defined to cancel both *TLM_IReady* and *TLM_DReady*. This method should then be called at every clock cycle by the wrapper in the beginning of the code. This way the *ready* signals will only be set to true when a response occurs, and will be automatically set back to false in the following clock cycle.

```
void writeD( uint32_t addr, uint32_t  wdata,uint32_t &data,↩
    uint32_t byteEnable)
{
  BRAM_B_PORT->writeD(addr, wdata, data, byteEnable);
  tlm_DataAddr = addr;
  tlm_ReadStrobe = false;
  tlm_WriteStrobe = true;
  tlm_BE = byteEnable;
  tlm_Data_Write = wdata;
  tlm_D_AS = true;
  returningD = true;
}
void readD(  uint32_t addr, uint32_t &data )
{
  BRAM_B_PORT->readD(addr, data);
  if(returningD)
  {
    tlm_Data_Read = data;
    tlm_DReady = true;
    tlm_D_AS = false;
    tlm_ReadStrobe = false;
    tlm_WriteStrobe = false;
    returningD = false;
```

```
  }
  else
  {
    tlm_DataAddr = addr;
    tlm_ReadStrobe = true;
    tlm_WriteStrobe = false;
    tlm_D_AS = true;
    returningD = true;
  }
}

void cancelReadys()
{
  tlm_DReady = false;
  tlm_IReady = false;
}
```

Chapter 6

Results

6.1 Testing

Testing of any module or system can be done using two approaches. The first one is to test the system on its own by feeding it some inputs and comparing the internal values and the outputs to the expected responses for such an input. This is usually a rather complex procedure because the programmer then has to design test applications, and provide full tracing for the internal and output signals to act as a validation reference. The second approach that is used for system testing is comparing the DUT to a reference system that carries out the same function, and is guaranteed to be working correctly.

It would be hard to fully test a processor functionality, especially when test applications are generated manually as is the case in this work. For this reason test applications used are confined to target specific internal details inside the processor that are known to be relatively complex as those areas would be more likely to have errors. Therefore the test applications tried on our system would mainly be addressing:

- Some basic instructions like add, multiply, load and store instructions to test the general functionality and timings of the system.
- Branching instructions with and without delay slots, including the “branch anyway” case discussed in the implementation chapter.
- Forwarding on all levels.
- Stalls and double stalls.
- FPU functionality, including providing faulty operands (denormalized for example).
- Byte access functionality.

From these requirements, two test applications were developed. The first one (referred to as full test application) demonstrated in Listing A.1 is a long test that lists all the possible intructions that are defined in the MicroBlaze processor. The second test (referred to as the formal test application) is developed using the requirements discussed above. The formal test aims to tackle the critical design details such as stalling and branching.

6.1.1 RTL Vs. VHDL Testing

Unfortunately, the VHDL models designed by Xilinx are encoded in a way that they can be simulated, but the internal signals cannot be viewed during simulations. Therefore the only signals that can be viewed and compared to our models for testing are the communication signals of the entity wrappers like the signals between the processor and the LMB. Therefore, the RTL model can only depend on the communication signals. A testbench system is then defined, that initializes a SystemC RTL system consisting of a single processor and a single BRAM, along with a VHDL system containing a processor, LMB, BRAM and a BRAM-LMB interface controller. The testbench includes a sequential process that performs the actual comparing of the internal signals of both systems. A problem might arise if the process is sensitive to the positive clock edge like both MicroBlaze models, because it is not guaranteed which process would run before the other therefore the test might not work correctly. Therefore it would be better to make the testbench process sensitive to the negative clock edge so that it is guaranteed that no uncertainty shall occur with the signal assignment orderings.

Upon running the full test, some unexpected observations were made. For example, some instructions (like Multiply and Barrel shift instructions) when they are followed by any other instruction that is dependent on the result of those instructions, no forwarding occurs. This means that the pipeline gets stalled as if it is a Load instruction followed by another dependent instruction. As a workaround for this special case, the wrapper is modified so that it handles Multiply and Barrel shift registers (with all their different versions) as Load instructions. The new code is as follows:

```
if(((m_Instr & 0xD0000000) == 0xC0000000)||((m_Instr & 0↩
    xDC000000) == 0x40000000)||((m_Instr & 0xFC000780) == 0↩
    x58000200)) // Load Instruction OR Multiply instruction ↩
    OR Barrel Shift
{
  si_Load.write(true);
  si_Load_destination.write((m_Instr & 0x03E00000)>>21); //↩
      put Rd in the least signif.
}
else
  si_Load.write(false);
```

Another problem was discovered in the ISS, where it was detected that the carry calculation and keeping was wrong, so the Add instruction handlings (Add, Addk, Addc, Addi, etc..) were modified in the ISS to have correct functionality. Another observed problem was that there was another forwarding issue, in the case where a Load instruction, has a dependent instruction that is two instructions away from it. For example:

```
LWi R3, R5, 0
NOP
Add R7, R3, R0
```

In this case, it was observed that the ID of the Add instruction becomes delayed by 1 clock cycle, which was unexpected.

Upon implementing some workaround techniques for those errors, both tests were finally passed for the RTL-VHDL testing.

6.1.2 RTL Vs. TLM Testing

Verifying the TLM system against the RTL one has an advantage over the test with the VHDL system, since all the internal signals for both abstraction levels are accessible. This means that at every clock cycle, the testbench can compare registers located inside the ISS such as PC and GPRs, as well as memory contents inside the BRAM which gives more coverage to the tests performed. Therefore the testbench in this case also includes a sequential process that compares communication signals from the RTL model with the re-generated signals from the TLM-TLM adapter in the TLM system, and also compares the internal registers including GPRs and status registers of each ISS, and memory contents of both BRAMs.

The TLM wrapper was also modified with the workarounds discussed above to have the same performance as the RTL model, and upon this updating process, the tests were passed.

6.2 Performance Measurements

In this section we are interested to measure how much time it would take both SystemC abstraction levels (RTL and TLM) to simulate the same application for the same number of clock cycles. Conceptually, TLM saves time during memory accesses. In order to prove this point, different applications with different memory accessing frequencies can be used to show how much TLM really affects the simulation times. To create the performance measurement applications, also the EDK tool is used, which means that all applications have some mandatory instructions that the EDK uses to correctly initialize the system and the runtime environment before running any programs. These initial instructions do not matter to our measurements, especially when the simulations are run for long enough, such that the initial contribution to the measurement results is negligible.

The idea now is to reach some applications that offer the best case to show the TLM advantage, and the worst case for that in order to highlight the effect of using TLM. One way to achieve this would be to have an application with nothing but continuous memory accesses, and another with the least memory accesses possible. An example for the high-rate memory accessing application would be an infinite loop with random load and store instructions inside, while the other application would also be an infinite loop with the same number of instructions inside, but those instructions are simple ones like add, subtract or shift instructions. As a control experiment, a third application is used that is a simple branch instruction that loops on itself which simply resembles an infinite loop with no instructions inside. The three applications are referred to as inf_mem, inf_simple and inf_loop,

respectively. Another point that would make it an unfair comparison is system initialization, which is automatically done at the beginning of the simulation when sc_start() is called. The idea is that the RTL system contains more sc_signals and much more ports than the TLM system. This means that it might need more time to initialize the modules themselves and establish the communication signals between them than in the case of TLM. Therefore it would be more fair if the measurements are taken after both systems have been initialized, which is why the SC_INITIALIZE() command is called before any time measurements are made. Another command that achieves the same function is sc_start(SC_ZERO_TIME) so either one can be used.

The measurements are performed consecutively not in parallel so as not to affect each other. Each application is run on each model ten times, each run is a 5-second simulation that is initiated using sc_start(5, SC_SEC) command. Time of the simulation is measured using the Native Linux Resource Usage command *getrusage()* [6]. The results of the measurement are summarized in Table 6.1. As expected, TLM is faster in all the tested applications which proves the hypothesis of this thesis. The least speedup (24.5%) occurred in the case of the simple instructions loop, and the maximum speedup (28%) was achieved with the high-rate memory accesses application which was also expected.

Application	RTL				TLM				Speedup
	Min	Max	Avg	St. dev	Min	Max	Avg	St. dev	
inf_loop	587.2	607.4	594.5	8.7	431.4	437.9	434	1.9	26.9%
inf_mem	859.8	871.6	865.8	4.2	613.5	632.4	624.1	5.4	28.0%
inf_simple	608.8	622.6	615.5	4.4	457.6	474.9	464.5	4.4	24.5%

Table 6.1: Performance measurements summary

Chapter 7

Conclusion

7.1 Summary

This work presented the modeling of the muti-core Xilinx MicroBlaze micro-processor system in RTL and TLM abstraction levels in SystemC. The system included several modules such as: MicroBlaze processor, LMB, BRAM and a LMB-BRAM interface controller. The system was abstracted into a processor and BRAM block for simplicity, and this abstracted system was modeled twice in RTL and TLM abstraction levels. Modeling of the processor was achieved by implementing a wrapper for the untimed ISS; the wrapper provided both the timing requirements as well as the communication means for the ISS to communicate with external modules such as the local memory. The task was divided into two stages, first the RTL model was implemented, and then it was used as a basis for TLM transformation simply by modifying the communication protocols. The TLM model was designed to be cycle-accurate, which means it has the same timing properties as the RTL model. Adapters were also implemented to transform between RTL and TLM abstraction levels to enable the use of different level modules in a single system. After the implementation was completed, tests were made to verify the modeled systems against the reference VHDL model of the MicroBlaze micro-processor using SystemC mixed-level simulation. Finally, the speedup of using TLM to simulate the system instead of RTL was calculated by running some applications on both systems and measuring the simulation times. Speedup varied from 24.5% to 28% depending on the rate of memory accesses in the application.

7.2 Outlook

Further properties can still be added to the system in the future. The first field of modification opportunities is internal improvements to the processor, such as: adding interrupt handling, implementation of Fast Simplex Links (FSL) or adding configuration options to allow the user to choose the capabilities of the system instead of having a single version of the processor. The other field of modification is the external field, where it is possible to model other new modules that are included in the MicroBlaze micro-processor system such as: the Processor Local Bus (PLB), Double Data Rate (DDR2) memory interface module, XPS-BRAM interface or the XPS Direct Memory Access (DMA) controller.

Appendix A

Testing Applications

Listing A.1 demonstrates the exhaustive testing application, while Listing A.2 demonstrates the formally generated test application used in Chapter 6.

Listing A.1: Full test application

```
000001a8 <main>:
 1a8: 3021fff8  addik r1, r1, -8
 1ac: fa610004  swi r19, r1, 4
 1b0: 12610000  addk  r19, r1, r0
 1b4: 20200064  addi  r1, r0, 100
 1b8: 204000c8  addi  r2, r0, 200
 1bc: 00611000  add r3, r1, r2
 1c0: 04811000  rsub  r4, r1, r2
 1c4: 08a11000  addc  r5, r1, r2
 1c8: 0cc11000  rsubc r6, r1, r2
 1cc: 10e11000  addk  r7, r1, r2
 1d0: 15011000  rsubk r8, r1, r2
 1d4: 19211000  addkc r9, r1, r2
 1d8: 1d420800  rsubkc  r10, r2, r1
 1dc: 1d611000  rsubkc  r11, r1, r2
 1e0: 15811001  cmp r12, r1, r2
 1e4: 15a11003  cmpu  r13, r1, r2
 1e8: 21c10017  addi  r14, r1, 23
 1ec: 25e10034  rsubi r15, r1, 52
 1f0: 2a010022  addic r16, r1, 34
 1f4: 2e21004b  rsubic  r17, r1, 75
 1f8: 32410041  addik r18, r1, 65
 1fc: 3661ffec  rsubik  r19, r1, -20
 200: 3a810063  addikc  r20, r1, 99
 204: 3ea1002b  rsubikc r21, r1, 43
 208: 42c11000  mul r22, r1, r2
 20c: 42e11001  mulh  r23, r1, r2
 210: 43011003  mulhu r24, r1, r2
 214: 43211002  mulhsu  r25, r1, r2
 218: 63410005  muli  r26, r1, 5
 21c: 47611a00  bsra  r27, r1, r3
 220: 03811800  add r28, r1, r3
 224: 47a11c00  bsll  r29, r1, r3
 228: 67c10003  bsrli r30, r1, 3
 22c: 67e10205  bsrai r31, r1, 5
 230: f86007bc  swi r3, r0, 1980
 234: f88007bc  swi r4, r0, 1980
```

```
   238:  f8a007bc   swi r5, r0, 1980
   23c:  f8c007bc   swi r6, r0, 1980
   240:  f8e007bc   swi r7, r0, 1980
   244:  f90007bc   swi r8, r0, 1980
42 248:  f92007bc   swi r9, r0, 1980
   24c:  f94007bc   swi r10, r0, 1980
   250:  f96007bc   swi r11, r0, 1980
   254:  f98007bc   swi r12, r0, 1980
   258:  f9a007bc   swi r13, r0, 1980
47 25c:  f9c007bc   swi r14, r0, 1980
   260:  f9e007bc   swi r15, r0, 1980
   264:  fa0007bc   swi r16, r0, 1980
   268:  fa2007bc   swi r17, r0, 1980
   26c:  fa4007bc   swi r18, r0, 1980
52 270:  fa6007bc   swi r19, r0, 1980
   274:  fa8007bc   swi r20, r0, 1980
   278:  faa007bc   swi r21, r0, 1980
   27c:  fac007bc   swi r22, r0, 1980
   280:  fae007bc   swi r23, r0, 1980
57 284:  fb0007bc   swi r24, r0, 1980
   288:  fb2007bc   swi r25, r0, 1980
   28c:  fb4007bc   swi r26, r0, 1980
   290:  fb6007bc   swi r27, r0, 1980
   294:  fb8007bc   swi r28, r0, 1980
62 298:  fba007bc   swi r29, r0, 1980
   29c:  fbc007bc   swi r30, r0, 1980
   2a0:  fbe007bc   swi r31, r0, 1980
   2a4:  48811000   idiv  r4, r1, r2
   2a8:  48a11002   idivu r5, r1, r2
67 2ac:  80c11000   or  r6, r1, r2
   2b0:  84e11000   and r7, r1, r2
   2b4:  89011000   xor r8, r1, r2
   2b8:  8d211000   andn  r9, r1, r2
   2bc:  a141007f   ori r10, r1, 127
72 2c0:  a56100ff   andi  r11, r1, 255
   2c4:  a9810000   xori  r12, r1, 0
   2c8:  ada100ff   andni r13, r1, 255
   2cc:  81c11400   pcmpbf  r14, r1, r2
   2d0:  89e11400   pcmpeq  r15, r1, r2
77 2d4:  8e011400   pcmpne  r16, r1, r2
   2d8:  92210001   sra r17, r1
   2dc:  92410021   src r18, r1
   2e0:  92610041   srl r19, r1
   2e4:  92810060   sext8 r20, r1
82 2e8:  92a10061   sext16  r21, r1
   2ec:  f86007bc   swi r3, r0, 1980
   2f0:  f88007bc   swi r4, r0, 1980
   2f4:  f8a007bc   swi r5, r0, 1980
   2f8:  f8c007bc   swi r6, r0, 1980
```

```
 87   2fc: f8e007bc  swi r7, r0, 1980
      300: f90007bc  swi r8, r0, 1980
      304: f92007bc  swi r9, r0, 1980
      308: f94007bc  swi r10, r0, 1980
      30c: f96007bc  swi r11, r0, 1980
 92   310: f98007bc  swi r12, r0, 1980
      314: f9a007bc  swi r13, r0, 1980
      318: f9c007bc  swi r14, r0, 1980
      31c: f9e007bc  swi r15, r0, 1980
      320: fa0007bc  swi r16, r0, 1980
 97   324: fa2007bc  swi r17, r0, 1980
      328: fa4007bc  swi r18, r0, 1980
      32c: fa6007bc  swi r19, r0, 1980
      330: fa8007bc  swi r20, r0, 1980
      334: faa007bc  swi r21, r0, 1980
102   338: 58611000  fadd  r3, r1, r2
      33c: 58811080  frsub r4, r1, r2
      340: 58a11100  fmul  r5, r1, r2
      344: 58c11180  fdiv  r6, r1, r2
      348: 58e11200  fcmp.un r7, r1, r2
107   34c: 59011210  fcmp.lt r8, r1, r2
      350: 59211220  fcmp.eq r9, r1, r2
      354: 59411230  fcmp.le r10, r1, r2
      358: 59611240  fcmp.gt r11, r1, r2
      35c: 59811250  fcmp.ne r12, r1, r2
112   360: 59a11260  fcmp.ge r13, r1, r2
      364: 59c10280  flt r14, r1
      368: 59e10300  fint  r15, r1
      36c: 5a010380  fsqrt r16, r1
      370: f86007bc  swi r3, r0, 1980
117   374: f88007bc  swi r4, r0, 1980
      378: f8a007bc  swi r5, r0, 1980
      37c: f8c007bc  swi r6, r0, 1980
      380: f8e007bc  swi r7, r0, 1980
      384: f90007bc  swi r8, r0, 1980
122   388: f92007bc  swi r9, r0, 1980
      38c: f94007bc  swi r10, r0, 1980
      390: f96007bc  swi r11, r0, 1980
      394: f98007bc  swi r12, r0, 1980
      398: f9a007bc  swi r13, r0, 1980
127   39c: f9c007bc  swi r14, r0, 1980
      3a0: f9e007bc  swi r15, r0, 1980
      3a4: fa0007bc  swi r16, r0, 1980
      3a8: 10600000  addk  r3, r0, r0
      3ac: 10330000  addk  r1, r19, r0
132   3b0: ea610004  lwi r19, r1, 4
      3b4: 30210008  addik r1, r1, 8
      3b8: b60f0008  rtsd  r15, 8
      3bc: 80000000  or  r0, r0, r0
```

Listing A.2: Formal test application

```
000001a8 <main>:
 1a8: 3021fff8  addik r1, r1, -8
 1ac: fa610004  swi r19, r1, 4
 1b0: 12610000  addk  r19, r1, r0
 1b4: 20a00abc  addi  r5, r0, 2748
 1b8: b8000014  bri 20     // 1cc
 1bc: 20a50001  addi  r5, r5, 1
 1c0: 20a50002  addi  r5, r5, 2
 1c4: 80000000  or  r0, r0, r0
 1c8: 80000000  or  r0, r0, r0
 1cc: f8a006b4  swi r5, r0, 1716   // 6b4 <__dso_handle>
 1d0: 80000000  or  r0, r0, r0
 1d4: 80000000  or  r0, r0, r0
 1d8: 80000000  or  r0, r0, r0
 1dc: 20a00abb  addi  r5, r0, 2747
 1e0: b8100014  brid  20     // 1f4
 1e4: 20a50001  addi  r5, r5, 1
 1e8: 20a50002  addi  r5, r5, 2
 1ec: 20a50003  addi  r5, r5, 3
 1f0: 80000000  or  r0, r0, r0
 1f4: f8a006b4  swi r5, r0, 1716   // 6b4 <__dso_handle>
 1f8: 80000000  or  r0, r0, r0
 1fc: 80000000  or  r0, r0, r0
 200: 80000000  or  r0, r0, r0
 204: 20a00abc  addi  r5, r0, 2748
 208: b800000c  bri 12     // 214
 20c: 20a50001  addi  r5, r5, 1
 210: 20a50002  addi  r5, r5, 2
 214: f8a006b4  swi r5, r0, 1716   // 6b4 <__dso_handle>
 218: 80000000  or  r0, r0, r0
 21c: 80000000  or  r0, r0, r0
 220: 80000000  or  r0, r0, r0
 224: 20a00abb  addi  r5, r0, 2747
 228: b800000c  bri 12     // 234
 22c: 20a50001  addi  r5, r5, 1
 230: 20a50002  addi  r5, r5, 2
 234: f8a006b4  swi r5, r0, 1716   // 6b4 <__dso_handle>
 238: 80000000  or  r0, r0, r0
 23c: 80000000  or  r0, r0, r0
 240: 80000000  or  r0, r0, r0
 244: 20a00abb  addi  r5, r0, 2747
 248: 20a50001  addi  r5, r5, 1
 24c: f8a006b4  swi r5, r0, 1716   // 6b4 <__dso_handle>
 250: 80000000  or  r0, r0, r0
 254: 80000000  or  r0, r0, r0
 258: 80000000  or  r0, r0, r0
 25c: 20a00abb  addi  r5, r0, 2747
 260: 80000000  or  r0, r0, r0
```

```
264:  20a50001   addi  r5, r5, 1
268:  80000000   or  r0, r0, r0
26c:  80000000   or  r0, r0, r0
270:  f8a006b4   swi r5, r0, 1716  // 6b4 <__dso_handle>
274:  20a00abb   addi  r5, r0, 2747
278:  f8a006b4   swi r5, r0, 1716  // 6b4 <__dso_handle>
27c:  20a00000   addi  r5, r0, 0
280:  e8a006b4   lwi r5, r0, 1716  // 6b4 <__dso_handle>
284:  20a50001   addi  r5, r5, 1
288:  f8a006b8   swi r5, r0, 1720  // 6b8 <p.2214>
28c:  80000000   or  r0, r0, r0
290:  80000000   or  r0, r0, r0
294:  80000000   or  r0, r0, r0
298:  e8a006b4   lwi r5, r0, 1716  // 6b4 <__dso_handle>
29c:  80000000   or  r0, r0, r0
2a0:  20a50001   addi  r5, r5, 1
2a4:  f8a006b8   swi r5, r0, 1720  // 6b8 <p.2214>
2a8:  e8a006b4   lwi r5, r0, 1716  // 6b4 <__dso_handle>
2ac:  20c00abc   addi  r6, r0, 2748
2b0:  f8c50000   swi r6, r5, 0
2b4:  20c00000   addi  r6, r0, 0
2b8:  20a00000   addi  r5, r0, 0
2bc:  e8a006b4   lwi r5, r0, 1716  // 6b4 <__dso_handle>
2c0:  e8c50000   lwi r6, r5, 0
2c4:  20c60001   addi  r6, r6, 1
2c8:  f8c006b8   swi r6, r0, 1720  // 6b8 <p.2214>
2cc:  10600000   addk  r3, r0, r0
2d0:  10330000   addk  r1, r19, r0
2d4:  ea610004   lwi r19, r1, 4
2d8:  30210008   addik r1, r1, 8
2dc:  b60f0008   rtsd  r15, 8
2e0:  80000000   or  r0, r0, r0
```

Bibliography

[1] N. Pouillon, A. Becoulet, A. V. de Mello, F. Pecheux, and A. Greiner, "A generic instruction set simulator api for timed and untimed simulation and debug of mp2-socs," in *Rapid System Prototyping, 2009. RSP'09. IEEE/IFIP International Symposium on.* IEEE, 2009, pp. 116–122. (Cited on pages 1, 3, 15, 16 and 17.)

[2] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis.* ACM, 2003, pp. 19–24. (Cited on pages 1 and 3.)

[3] F. Vahid, *Digital Design with RTL Design, Verilog and VHDL.* Wiley, 2010. (Cited on page 3.)

[4] Accellera Systems Initiative. (2002) About systemc. [Online]. Available: http://www.accellera.org/downloads/standards/systemc/about_systemc/ (Cited on page 4.)

[5] D. K. Tala. (2013, Feb) Systemc modules. [Online]. Available: http://www.asic-world.com/systemc/modules1.html (Cited on pages 4 and 5.)

[6] K. Iglberger, B. Heubeck, and C. Jandl, "Time Measurement in C/C++," DEC 2008. (Cited on pages 6 and 60.)

[7] Mentor Graphics, "Modelsim pe user's manual," 2011. (Cited on pages 6, 7 and 23.)

[8] Xilinx Inc. Xilinx platform studio. [Online]. Available: http://www.xilinx.com/tools/xps.htm (Cited on page 7.)

[9] Xilinx Inc., "Data2mem user guide," *Reference Manual*, Jun 2009. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/data2mem.pdf (Cited on page 7.)

[10] Xilinx, "Microblaze processor reference guide," APR 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf (Cited on pages vi, 7, 8 and 9.)

[11] Xilinx, "Logicore ip microblaze micro controller system (v1.1)," APR 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ds865_microblaze_mcs.pdf (Cited on pages vi and 8.)

[12] D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface.* Morgan Kaufmann, 2009. (Cited on pages 9 and 42.)

[13] Xilinx, "Local memory bus (lmb) v1.0 (v1.00a)," APR 2005. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/lmb.pdf (Cited on page 11.)

[14] Xilinx, "Ip processor block ram (bram) block (v1.00a)," MAR 2011. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/bram_block.pdf (Cited on pages 12, 23 and 26.)

[15] Xilinx, "Lmb bram interface controller (v2.10b)," DEC 2009. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/lmb_bram_if_cntlr.pdf (Cited on page 12.)

[16] SoCLib Consortium and others, "The soclib project: An integrated system-on-chip modelling and simulation platform," Technical report, CNRS, 2003. http://www. soclib. fr, Tech. Rep. [Online]. Available: http://www.soclib.fr/ (Cited on page 12.)

[17] C. Mucci, F. Campi, A. Deledda, A. Fazzi, M. Ferri, and M. Bocchi, "A cycle-ac curate iss for a dynamically reconfigurable processor architecture," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. IEEE, 2005, pp. 8–pp. (Cited on page 15.)

[18] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino, "Legacy systemc co-simulation of multi-processor systems-on-chip," in *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*. IEEE, 2002, pp. 494–499. (Cited on pages vi, 15, 16 and 17.)

[19] T.-C. Yeh, G.-F. Tseng, and M.-C. Chiang, "A fast cycle-accurate instruction set simulator based on qemu and systemc for soc development," in *MELECON 2010-2010 15th IEEE Mediterranean Electrotechnical Conference*. IEEE, 2010, pp. 1033–1038. (Cited on pages 15 and 18.)

[20] M.-C. Chiang, T.-C. Yeh, and G.-F. Tseng, "A qemu and systemc-based cycle-accurate iss for performance estimation on soc development," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 593–606, 2011. (Cited on pages 15 and 18.)

[21] J. Aynsley, "Osci tlm-2.0 language reference manual," *Open SystemC Initiative (OSCI)*, 2009. (Cited on pages 15 and 18.)

[22] R. Stallman and R. H. Pesch, *The GDB Manual: The GNU Source-level Debugger*. Free Software Foundation, 1992. (Cited on page 16.)

[23] A. Zeller and D. Luetkehaus, "Ddd-a free graphical front-end for unix debuggers," *ACM Sigplan Notices*, vol. 31, no. 1, pp. 22–27, 1996. (Cited on page 16.)

[24] N. Bombieri, F. Fummi, and V. Guarnieri, "Accelerating rtl fault simulation through rtl-to-tlm abstraction," in *European Test Symposium (ETS), 2011 16th IEEE*. IEEE, 2011, pp. 117–122. (Cited on page 17.)

[25] N. Bombieri, F. Fummi, and G. Pravadelli, "Automatic abstraction of rtl ips into equivalent tlm descriptions," *Computers, IEEE Transactions on*, vol. 60, no. 12, pp. 1730–1743, 2011. (Cited on pages vi, 17 and 18.)

[26] A. Bruce, M. Kamal Hashmi, A. Nightingale, S. Beavis, N. Romdhane, and C. Lennard, "Maintaining consistency between systemc and rtl system designs," in *Proceedings of the 43rd annual Design Automation Conference*. ACM, 2006, pp. 85–89. (Cited on pages vi, 17 and 19.)

[27] W. Klingauf, "Systematic transaction level modeling of embedded systems with systemc," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*. IEEE Computer Society, 2005, pp. 566–567. (Cited on page 17.)

[28] N. Bombieri, F. Fummi, and G. Pravadelli, "RTL-TLM equivalence checking based on simulation," in *Design & Test Symposium (EWDTS), 2008 East-West*. IEEE, 2008, pp. 214–217. (Cited on pages vi, 17, 19 and 20.)

[29] A. Habibi and S. Tahar, "Design for verification of systemc transaction level models," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*. IEEE Computer Society, 2005, pp. 560–565. (Cited on page 17.)

[30] W. Ecker and L. Schönberg, "Impact of SystemC data types on execution speed," Infineon Technologies AG, Apr. 2007. (Cited on page 23.)

[31] I. Xilinx, "Microblaze processor reference guide," *reference manual*, 2006. (Cited on pages 23, 45 and 46.)