CHAPTER

1 Introduction

1.1 Abstract

In this thesis, we have designed a core model of microprocessor that can be used for performance evaluation of any communication architecture. In its outer form, this model is an entity in VHDL. To do performance evaluation for any communication architecture, several instances of this entity can be made depending upon the requirement and are then connected to the memory according to the configuration of that communication architecture. These models then generate traffic on the bus to communicate with the memory. At the end, these models generate report about their performance. Most important in that report is the effective CPI (Cycles per Instruction) under the given communication architecture.

The main feature of this model is that it performs simulation very fast as compared to the behavior models of microprocessors. For e.g., it can simulate 1 million instructions in nearly 1-2 minutes on SUN machines. Whereas, the complex models of microprocessors require 2-3 hours or even more for the same number of instructions. Moreover, it doesn't require any software to run i.e. to perform simulation it does not require that some software should be loaded into the memory. It can perform simulation without any original software to generate some performance statistics.

Although in its outer form it is a core model of a microprocessor (an entity in VHDL), it can also be termed as a 'tool' to analyze performance of a system. Because it can only be used for performance estimation purposes and is not a hardware design.

1.2 Background

Systems on chips (SOC) are becoming increasingly more complex and dense. A single SOC design may consist of a processor, memory, some dedicated hardware and Input/output interface. SOC designs that consist of more than 1 processor are common these days. To deal with the complexity, we rely on use of intellectual Property (IP) cores. With increasing number of IP cores, it is important to connect these in a structured and efficient way. The communication architecture becomes essential in providing a flexible platform, and it is essential for the overall SOC performance. Decisions about the communication architecture should be made as early as possible in the design process.

These decisions have been made by simulating the system and evaluating the performance of the system. By performance, we mean how fast the system completes its task. However simulating the system with the real cores is too slow and time consuming. It is not possible to simulate a system in a short period of time that consists of many complex IP cores. At the early phase of the design, which is likely to undergo lot of changes it is not a good idea to spend too much time on extensive simulation. A general rough estimation is quite enough at the start of the design. Therefore methods for faster performance simulation are always attractive and a tradeoff between faster execution and accurate performance estimation is justified. Hence at the early phase of design process, we can replace the real cores with their simple core models. These core models are much faster to simulate than the real cores.

When dealing with performance evaluation we are not interested in the functionality of the system. For example, if we have to make a performance estimation of a system comprising of two processors sharing a single memory, we are only interested in knowing how frequently the processors communicate with the memory, how many times we have a conflict between processors to access the memory, how much time the processor remains stalled waiting for its turn to access the memory, is the memory has been efficiently shared or not etc.

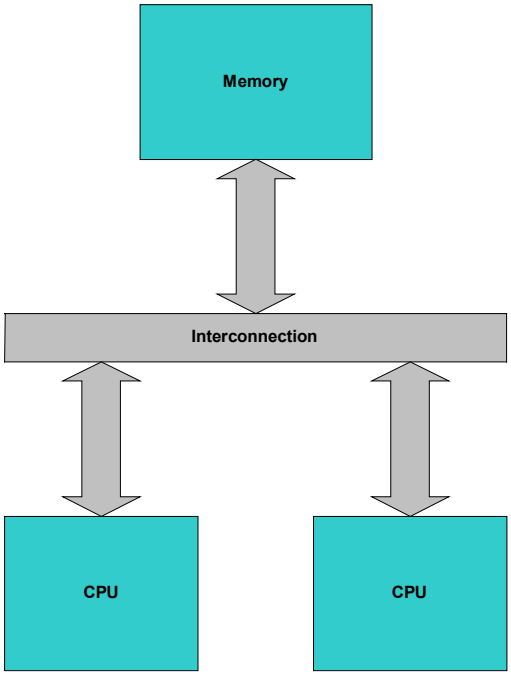


Figure 1.1 Two CPUs sharing the interconnection network to access the single memory

For simulation we also need software that is going to be executed on the processor. The processor communicates with the memory architectures when it needs instructions which are not in its cache or when it requires some data which are not in its cache as well. This behavior largely depends on the software and different softwares generate different traffic. It can be a case that we don't know which software would be executed on this given architecture. It is also difficult to get a significant number of softwares that are written for performance estimation.

Hence the objective is that we don't know which software is going to be executed on a given architecture and so we don't want to make performance estimation by running few self-made programs

In this project, we have targeted on these two different problems:

- If we don't know what software would be executed on a given architecture then how can we make the performance estimation?
- How can we make the simulation much faster than the simulation that involves real cores or the behavior models of the cores?

To tackle the first problem, we will make some models of the different program behaviors and a core model of microprocessor uses these behaviors to generate traffic that involves communication with the outside environment. Since we are not executing any real program, the job of the core model is to generate traffic only. This means that the model would not do any thing for the instructions, which do not involve communication with the memory. The communication with the memory takes place when the processor needs instruction to execute which is not found in the instruction cache or when the

processor needs to read or write some data, which is not in the data cache. The former case would happen only when the instruction is not found in the instruction cache and the latter case would happen only when the given instruction is either load or store and the processor faces a data miss in the data cache. Thus whenever there is a miss in either instruction or data cache the processor communicates with the memory and there is traffic on the bus connecting the processor and the memory. The job of the model is to make decisions at which time traffic is generated i.e. when there is a miss in the cache. As there is no real software executing on the model so it has to make this decision by its own. This requires some input parameters from the user of this model. Some of these parameters are:

- Number of instructions
- Load/store instructions
- Cycles per instruction (CPI)
- Instruction miss rate
- Data miss rate

Based on these parameters the microprocessor model would communicate with the memory from time to time. Our job is to study different program behaviors that cause communication between the CPU and the memory and characterize these behaviors in the CPU model. Thus the CPU model will generate traffic that looks like a normal program executing on the processor.

In most CAD tools, which are used for performance evaluation, we provide them some software to execute along with some cache configuration parameters (cache size, block size etc.). These tools then calculate the effective instruction and data miss rate. Our model, nearly works in opposite direction. It takes total number of instructions, instruction miss rate and data miss rate as an input and then generate different possible sequences of these misses. In other words, by using these inputs it generates different possible traffic patterns that may result due to different types of softwares.

Hence we will develop a CPU model, which can generate traffic by using some input parameters that statically corresponds to the real cores and also do some performance measurements like execution time of given number of instructions, no of memory accesses, effective CPI, number of clock cycles CPU remains stalled waiting for the memory, number of clock cycles during which CPU is perfectly in execution etc. We can connect this CPU model in different configurations and then evaluate the traffic statistics between different components. In a multiprocessing environment the performance is highly dependent on the communication architecture and different configurations may result in different performance results.

The second requirement is to make it much faster than normal simulation. This simulation should be very fast because in the absence of real software the performance estimation would not be very accurate. The aim is to get some rough approximation at the very early stage of design process.

In fact, the steps taken to solve the first requirement have also solved the second requirement. When simulating with real cores the simulation of every instruction consumes some time. The simulator needs to record every event that happened inside or outside the CPU. However in our model we only do some processing in the case of a miss. So in every clock cycle we just need to make decision whether we have a miss or not. The simulation time for making such decisions is much smaller than the time for simulating real processor core that involve lot of signals and millions of gates. Off course the high-level behavior models of the processors can do the performance simulation. But still it doesn't provide that much speedup and also they require software to be executed on these models.

Therefore the simulation with this behavior model of a microprocessor would be much faster than executing any program on the processor as well as it doesn't contain any effect introduce by a particular program. As mentioned earlier, since we are only interested in the performance it is possible to make some rough estimate of the system without executing the software.

The CPU model is designed to be used in different architectures; therefore it is necessary that it should follow some standard interface. The CPU model follows the Open Core Protocol (OCP) interface i.e. it can be connected to any component that follows the OCP interface. Or we can say that the communication between the processor and the memory would be through some interconnect following the OCP. The relevant details of OCP are given in Chapter 2.

Developing CPU model requires a lot of programming features, which are hard to find in a Hardware Description Language like VHDL. Therefore the model is written in C language. However most of the IP cores currently used in the industry are in VHDL. As a result of this, we need to work in a mixed language environment with our processor model in C language and the other cores which are connected with the model are in VHDL. This requires an interface between C and VHDL. So that when seen from outside, the model looks like an entity written in VHDL but from inside, its all functionality is written in C language. We have used Foreign Language Interface (FLI) provided by the tool ModelSim. This means that our simulation environment consists of our model written in C language and the other cores written in VHDL all running in ModelSim. It may happen that in a given test bench we have more than one copies of our model along with other IP cores (written in VHDL) all connected to each other through OCP. The relevant details of FLI and its use in the thesis are given in Chapter 3.

1.3 Tasks of the project

Based on the above discussion we can say that we need to develop a core model of the general-purpose microprocessor that must have the following characteristics:

- It must follow the OCP and FLI interface.
- It must be able to generate traffic according to the input parameters discussed above.

- The simulation with the model should be much faster than the simulation with the real core.
- The statistics generated by the model should be reasonably accurate when compared with real program execution.

The next two chapters give an overview of OCP and FLI and the features, which are used in the model. Chapters 4 and 5 include the discussion how the misses have been introduced or we can say how the traffic has been generated. Chapter 6 deals with the discussion related to the testing of the model.

1.4 SUMMARY

In this chapter, we have introduced our project. We have described the motivation behind this project and our proposed steps that we will take to complete the requirements of the project. At the end, we have set the tasks that have to be met in this project.

We will design a CPU model, which is following OCP interface. The purpose of the model is to do faster simulation to make some performance measurements. It has to do these performance calculations without using any software. In the absence of real software, the task of the model is to behave in a manner as some real software is executing on it. This model is supposed to be used at the very early phase of design process where the tradeoff between accuracy and speed can be justified to a larger extent.

CHAPTER

2 Open Core Protocol

When an instruction or data miss occurs in a CPU, the CPU needs to communicate with the memory. This communication is brought through some protocol, which defines how the request to read or write data will be generated, how the memory responds to the request, how many minimum clock cycles are required to complete the data transfer, etc. The interface of the CPU and the memory (or any component which needs to communicate with the outside world) is designed according to that protocol. It means that any two or more components that follow the same protocol can be connected to each other without any modifications in the design.

We have used Open Core Protocol (OCP) in our CPU model. Since our CPU model can be used only for simulation there is no particular advantage of using any kind of protocol. We have used OCP because it is the most fast emerging protocol. To make simulation in any environment the CPU model needs to be connected with some other entities. Since OCP is the most widely used protocol these days, so it is very likely that the simulation would be done in an environment where the entities are following the OCP protocol.

The OCP defines complete standard from the basic data flow signals to the signals that are used for test purposes. Broadly, OCP signals can be divided into two main categories, the basic OCP signals and the optional OCP signals. The presence of basic OCP signals in any core is necessary if it is following the OCP interface. The optional OCP signals can be included according to the requirement. In our model, we have used only the basic OCP signals and very few optional signals. These signals are included both in our CPU model and the memory, which we have designed for testing purposes.

This chapter covers the very basic introduction about the OCP. Only the features used in the thesis are described here. Detailed information can be found from the online manual of OCP at www.ocp-ip.com.

2.1 An Overview of Open Core Protocol

The Open Core Protocol (OCP) defines a high-performance, bus-independent interface between IP cores that reduces design time, design risk, and manufacturing costs for SOC designs.

An IP core can be a simple peripheral core, a high-performance microprocessor, or an on-chip communication subsystem such as a wrapped on-chip bus. The Open Core Protocol:

- Achieves the goal of IP design reuse. The OCP transforms IP cores making them independent of the architecture and design of the systems in which they are used.
- Optimizes die area by configuring into the OCP only those features needed by the communicating cores.
- Simplifies system verification and testing by providing a firm boundary around each IP core that can be observed, controlled, and validated.

2.1.1 OCP Characteristics

The OCP defines a point-to-point interface between two communicating entities, such as IP cores and bus interface modules (bus wrappers). One entity acts as the master of the OCP instance, and the other as the slave. Only the master can present commands and is the controlling entity. The slave responds to commands presented to it, either by accepting data from the master, or presenting data to the master. For two entities to communicate in a peer-to-peer fashion, there need to be two instances of the OCP connecting them - one where the first entity is a master, and one where the first entity is a slave. *In our case, the CPU is the master and the memory is a slave entity.*

Figure 2.1 shows a simple system containing a wrapped bus and three IP core entities: one that is a system target, one that is a system initiator, and an entity that is both.

The characteristics of the IP core determine whether the core needs master, slave, or both sides of the OCP; the wrapper interface modules must act as the complementary side of the OCP for each connected entity. A transfer across this system occurs as follows. A system initiator (as the OCP master) presents command, control, and possibly data to its connected slave (a bus wrapper interface module). The interface module plays the request across the on-chip bus system. The OCP does not specify the embedded bus functionality. Instead, the interface designer converts the OCP request into an embedded bus transfer. The receiving bus wrapper interface module (as the OCP master) converts the embedded bus operation into a legal OCP command. The system target (OCP slave) receives the command and takes the requested action. Each instance of the OCP is configured (by choosing signals or bit widths of a particular signal) based on the requirements of the connected entities and is independent of the others.

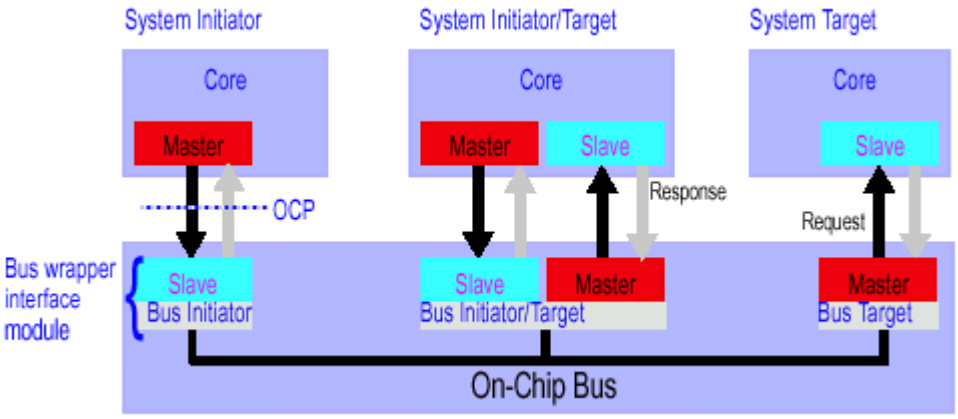


Figure 2.1 System Showing Wrapped Bus and OCP instances

For instance, system initiators may require more address bits in their OCP instances than do the system targets; the extra address bits might be used by the embedded bus to select which bus target is addressed by the system initiator.

The OCP is flexible. There are several useful models for how existing IP cores communicate with one another. Some employ pipelining to improve bandwidth and latency characteristics. Others use multiple-cycle access models, where signals are held static for several clock cycles to simplify timing analysis and reduce implementation area. Support for this wide range of behavior is possible through the use of synchronous handshaking signals that allow both the master and slave to control when signals are allowed to change.

2.1.2 OCP Interface Signals

OCP interface signals are grouped into dataflow, sideband, and test signals. A small set of the signals from the dataflow group called the basic OCP, is required in all OCP configurations. Optional signals can be configured to support additional core communication requirements. The optional dataflow signals are divided into simple and complex extensions. All sideband and test signals are optional. We will restrict our discussion to only basic OCP signals and signal used in burst access, which is in simple OCP extension category.

The OCP is a synchronous interface with a single clock signal. All OCP signals are driven with respect to and sampled by the rising edge of the OCP clock. *Except for clock and reset, OCP signals are strictly point-to-point and uni-directional.*

Dataflow Signals

The dataflow signals consist of a small set of required signals called the basic OCP and optional signals that can be configured to support additional core communication requirements. The optional dataflow signals are grouped into simple and complex extensions.

The naming conventions for dataflow signals use the prefix M for signals driven by the OCP master and S for signals driven by the OCP slave.

Basic Signals

Table 2.1 lists the basic OCP signals that must be present in any OCP interface.

Name	Width	Driver	Function
Clk	1	Varies	OCP clock
MAddr	1-32	Master	Transfer address
MCmd	3	Master	Transfer command
MData	8/16/32/64/128	Master	Write data
SCmdAccept	1	Slave	Slave accepts transfer
SData	8/16/32/64/128	Slave	Read data
SResp	2	Slave	Transfer response

Table 2.1: Basic OCP Signals

Clk

Clock signal for the OCP. All interface signals are synchronous to the rising edge of Clk.

MAddr

The Transfer address, MAddr specifies the slave-dependent address of the resource targeted by the current transfer.

MCmd

Transfer command. This signal indicates the type of transfer at the OCP. Commands are encoded as follows in table 2.2.

MCmd[2:0]	Transaction Type	Mnemonic
000	Idle	IDLE
001	Write	WR
010	Read	RD
011	ReadEx	RDEX
100	Reserved	
101	Reserved	
110	Reserved	
111	Broadcast	BCST

Table 2.2 : Command Encoding

MData

Write data. This field carries data from the master to the slave.

SCmdAccept

Slave accepts transfer. A value of 1 on the SCmdAccept signal indicates that the slave accepts the master's transfer request.

SData
Read data. This field carries data from the slave to the master.

SResp
Response field from the slave to a transfer request from the master. Response encoding is as follows in table 2.3.

SResp[1:0]	Response	Mnemonic
00	No response	NULL
01	Data valid/ accept	DVA
10	Reserved	
11	Response error	ERR

Table 2.3 : Response Encoding

MBurst
Burst type. This signal allows linking related transfers into a burst transaction. It is configured into the OCP using the burst parameter. Mburst is not a basic OCP signal. It encodes both the burst type and the burst code, as shown in the Table 2.4.

MBurst[2:0]	Burst Type	Burst Code
000	All	LAST
001	Incrementing	TWO
010	Incrementing	FOUR
011	Incrementing	EIGHT
100	Custom (packed)	DFLT1
101	Custom (not packed)	DFLT2
110	Streaming	STRM
111	Incrementing	CONT

Table 2.4 : Burst Encoding

All these interface signals are used in CPU (Master) and (Memory). The VHDL declarations of CPU and Memory are shown in fig 2.2 and 2.3. It should be noted that in OCP the width of the address and data signals could be varied according to the requirement. Therefore their widths are declared as ‘generic’.

```
entity cpu is
  generic (
    addr_width : integer ;
    data_width : integer );

  port (
    nreset          : in  bit;
    clk             : in  bit;                                  -- OCP signal
    SCmdAccept      : in  bit;                                  -- OCP signal
    SResp           : in  bit_vector(1 downto 0);               -- OCP signal
    SData           : in  bit_vector(data_width-1 downto 0);    -- OCP signal
    MCmd            : out bit_vector(2 downto 0);               -- OCP signal
    MBurst          : out bit_vector(2 downto 0);               -- OCP signal
    MAddr           : out bit_vector(addr_width-1 downto 0);    -- OCP signal
    MData           : out bit_vector(data_width-1 downto 0);    -- OCP signal
    cpu_stall       : out bit;
    cpu_out         : out bit);

end cpu;
```

Figure 2.2 VHDL declaration of CPU model following the OCP interface

```
entity memory is
  generic (
    addr_width : integer ;
    data_width : integer );

  port (
    nreset          : in   bit;
    clk             : in   bit;                                 -- OCP signal
    MCmd            : in   bit_vector(2 downto 0);              -- OCP signal
    MBurst          : in   bit_vector(2 downto 0);              -- OCP signal
    MAddr           : in   bit_vector(addr_width-1 downto 0);   -- OCP signal
    Mdata           : in   bit_vector(data_width-1 downto 0);   -- OCP signal
    SCmdAccept      : out  bit;                                 -- OCP signal
    SResp           : out  bit_vector(1 downto 0);              -- OCP signal
    SData           : out  bit_vector(data_width-1 downto 0)    -- OCP signal
    );

end memory;
```

Figure 2.3 VHDL declaration of Memory model following the OCP interface

2.1.3 Timing Diagrams

The following timing diagrams show the data transfer in its various forms between the Master and the Slave using OCP basic signals. The data transfer would become more versatile by the addition of simple and complex extensions. In our case, consider the Master as a CPU, which is originating communication with the memory, which is acting as a slave.

Simple Write and Read Transfer

Figure 2.4 illustrates a simple write and read transfer on a basic OCP interface. This diagram shows typical behavior for a synchronous SRAM or for the control and status registers of a core.

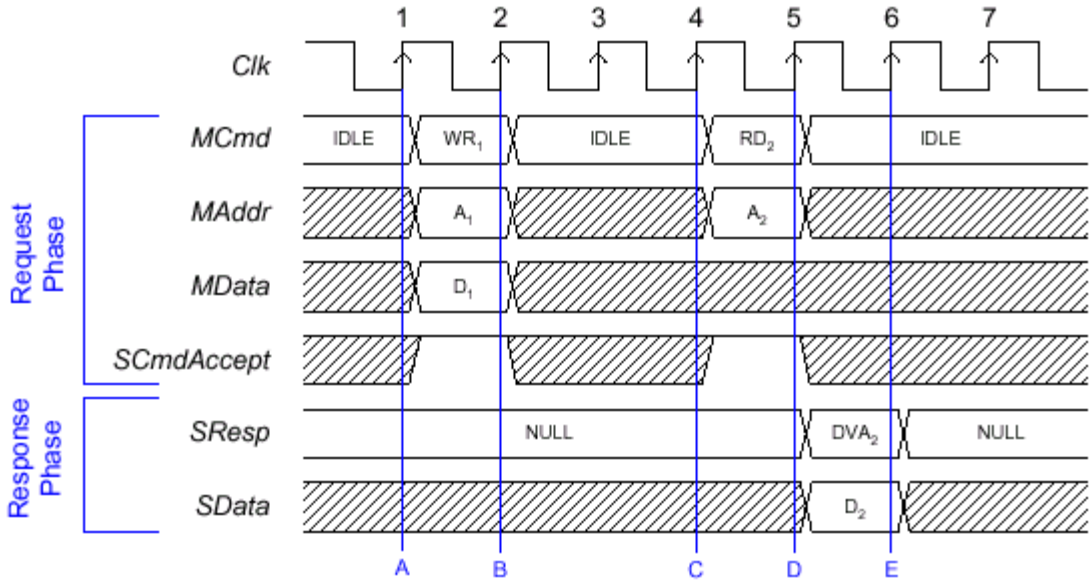


Figure 2.4 Simple Write and Read Transfer

Sequence

A. The master starts a request phase on clock 1 by switching the MCmd field from IDLE to WR. At the same time, it presents a valid address (A_1) on Addr and valid data (D_1) on MData. The slave asserts SCmdAccept in the same cycle, making this a 0-latency transfer.

B. The slave captures the values from MAddr and MData and uses them internally to perform the write. Since SCmdAccept is asserted, the request phase ends.

C. The master starts a read request by driving RD on MCmd. At the same time, it presents a valid address on MAddr. The slave asserts SCmdAccept in the same cycle for a request-accept latency of 0.

D. The slave captures the value from MAddr and uses it internally to determine what data to present. The slave starts the response phase by switching SResp from NULL to DVA.

The slave also drives the selected data on SData. Since SCmdAccept is asserted, the request phase ends.

E. The master recognizes that SResp indicates data valid and captures the read data from SData, completing the response phase. This transfer has request-to-response latency of 1.

Request Handshake

Figure 2.5 illustrates the basic flow-control mechanism for the request phase using SCmdAccept. There are three write transfers, each with a different request accept latency.

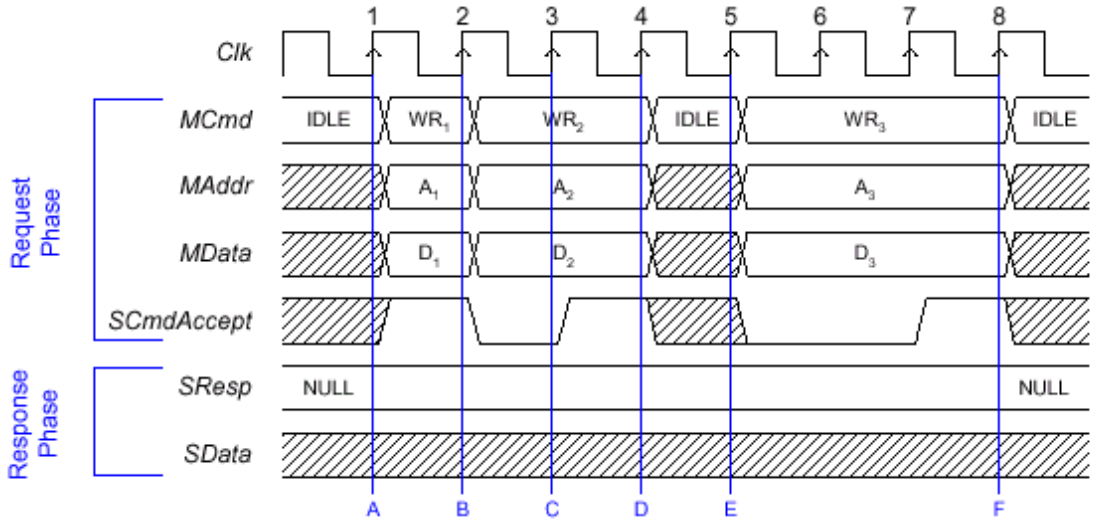


Figure 2.5 Request Handshake

Sequence

A. The master starts a write request by driving WR on MCmd and valid address and data on MAddr and MData, respectively. The slave asserts SCmdAccept in the same cycle, for a request accept latency of 0.

B. The master starts a new transfer in the next cycle. The slave captures the write address and data. It deasserts SCmdAccept, indicating that it is not yet ready for a new request.

C. Recognizing that SCmdAccept is not asserted, the master holds all request phase signals (MCmd, MAddr, and MData). The slave asserts SCmdAccept in the next cycle, for a request-accept latency of 1.

D. The slave captures the write address and data.

E. After 1 idle cycle, the master starts a new write request. The slave deasserts SCmdAccept.

F. Since SCmdAccept is asserted, the request phase ends. SCmdAccept was low for 2 cycles, so the request-accept latency for this transfer is 2. The slave captures the write address and data.

Request Handshake and Separate Response

Figure illustrates a single read transfer in which a slave introduces delays in the request and response phases. The request accept latency 2, corresponds to the number of clock cycles that SCmdAccept was deasserted. The request to response latency 3, corresponds to the number of clock cycles from the end of the request phase (D) to the end of the response phase (F).

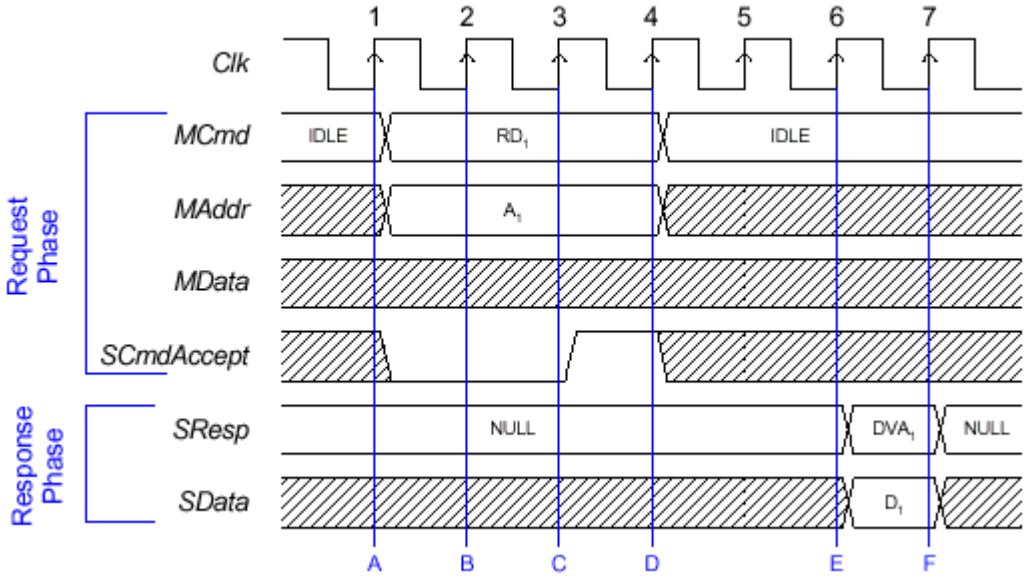


Figure 2.6 Request Handshake and Separate Response

Sequence

A. The master starts a request phase by issuing the RD command on the MCmd field. At the same time, it presents a valid address on MAddr. The slave is not ready to accept the command yet, so it deasserts SCmdAccept.

B. The master sees that SCmdAccept is not asserted, so it keeps all request phase signals steady. The slave may be using this information for a long decode operation, and it expects the master to hold everything steady until it asserts SCmdAccept.

C. The slave asserts SCmdAccept. The master continues to hold the request phase signals.

D. Since SCmdAccept is asserted, the request phase ends. The slave captures the address, and although the request phase is complete, it is not ready to provide the response, so it continues to drive NULL on the SResp field.

E. The slave is ready to present the response, so it issues DVA on the SResp field, and drives the read data on SData.
F. The master sees the DVA response and captures the read data.

Burst Read

Figure 2.7 illustrates a burst read transaction that is composed of four-pipelined burst read transfers. An additional field, MBurst, is added to the request phase, indicating the type of the burst and the number of transfers that the master expects. In this diagram, MData and SData are assumed to be 32 bits.

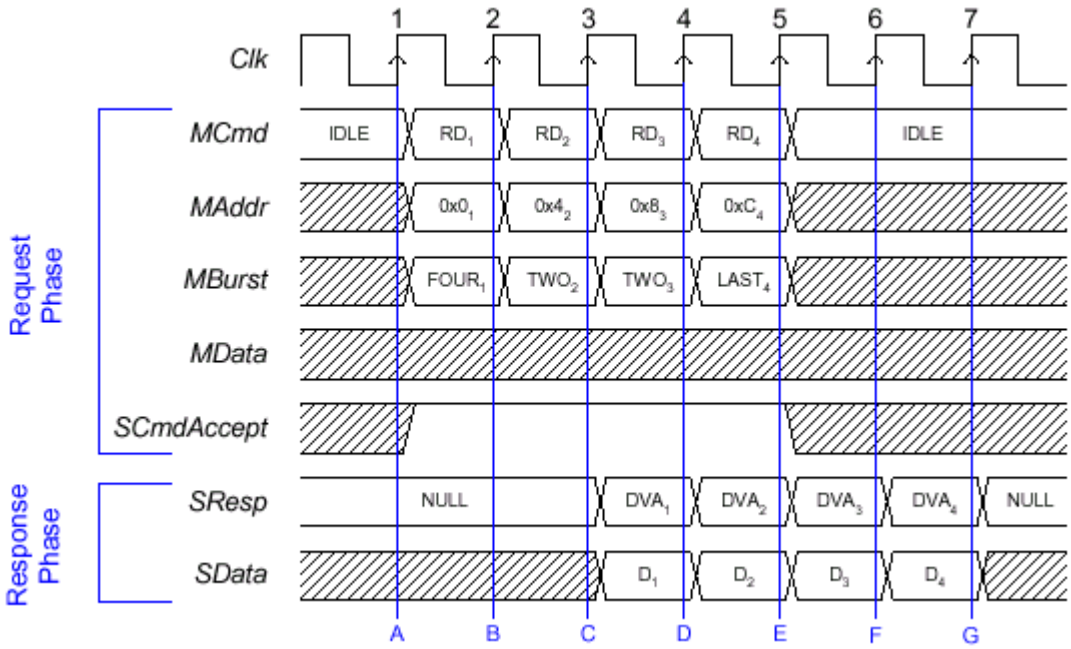


Figure 2.7 Burst Read

Sequence

A. The master starts the burst read by driving RD on MCmd, the first address of the burst on MAddr, and the burst code FOUR on MBurst. The burst code indicates that this is an incrementing burst and that four or more transfers are expected. The slave is ready for anything, so it asserts SCmdAccept.

B. The master issues the next read in the burst. MAddr is set to the next word-aligned address. For 32-bit words, the address is incremented by 4.The master also changes MBurst to TWO, meaning that two or more transfers remain in the transaction.

C. The master issues the next read in the burst, incrementing MAddr and leaving MBurst set to TWO, because there are still two or more transfers remaining. The slave is now ready to respond to the first read in the burst, so it drives DVA on SResp and valid data on SData. The request-to-response latency for this transfer is 2.

D. The master issues the final read in the burst, incrementing MAddr and setting MBurst to LAST. The master also captures the data for the first read from the slave. The slave

responds to the second transfer. The request-to-response latency for this transfer is 2, although it is possible for the slave to introduce more latency for each response in a burst transaction. (In OCP, bursts do not impose any additional constraints on protocol timing.)

E. The master captures the data for the second read from the slave. The slave responds to the third transfer.

F. The master captures the data for the third read from the slave. The slave responds to the fourth and last transfer.

G. The master captures the data for the last read from the slave.

2.2 OCP Implementation

The OCP interface is implemented in the CPU model according to the requirements discussed in the previous sections. As described in the first chapter, the CPU model is written in C. The OCP handshake protocol is implemented in a C function bus_interface_process(). When a miss occurs, this function is called to initiate communication with the memory. The total clock cycles required to complete the data transfer depend on the memory latency and the OCP protocol overhead. After the start of the handshake, the function bus_interface_process() is executed on every clock cycle until the end of the handshake. All the variables that store different transitions of the OCP signals during the handshake are declared globally in the C program so that their values would not be lost at the end of the clock cycle. This function is capable of handling both normal and burst OCP accesses. The C source code is given in the appendix.

2.3 Summary

In this chapter, we have discussed various features of OCP that are used in our CPU model. This includes the description of basic OCP signals and their signaling requirements. Later, their use has been demonstrated through timing diagrams, which show different types of handshakes. As told earlier, this chapter covers very basic information about OCP. Detailed information can be found from the online manual of OCP at www.ocp-ip.com. The important thing to remember is that the detection of a miss causes the CPU model to initiate communication with the memory. This communication is done through OCP and is implemented in the model by a C function bus_interface_process().

REFERENCES

Online manual of OCP at www.ocp-ip.com.

CHAPTER

3 Foreign Language Interface

As described in the first chapter, our CPU model is written in C language. However, most existing IP cores today are written either in VHDL or Verilog. In Europe, VHDL is the widely used Hardware Description Language. Therefore to make the simulation in a mixed language environment we need some interface between C and VHDL. The VHDL Foreign Language Interface (FLI) by 'Mentors Graphics' provides this interface.

The VHDL FLI allows us to replace VHDL architectures and subprogram bodies with code written in C. The FLI also provides a number of C functions to allow the VHDL database to be accessed and manipulated.

FLI has lot of features and can be used for many different tasks. This chapter only covers the portion of FLI, which is used in the thesis. Detailed information can be found from the ModelSim technical manual. However, it is our experience that the chapter concerning FLI in the ModelSim technical manual lacks some practical information that should be included for the beginner. In this chapter, we have tried to provide this information. We hope that this chapter would not only serve to understand the simulation environment of our thesis but also as a quick start guide for the beginner. It is assumed that the reader is familiar with VHDL.

3.1 Using VHDL FLI with Foreign Architectures

The purpose of the following discussion is not to explain FLI but to make it clear how to use FLI i.e. how we can integrate our C and VHDL code so that we can simulate in a mixed language environment.

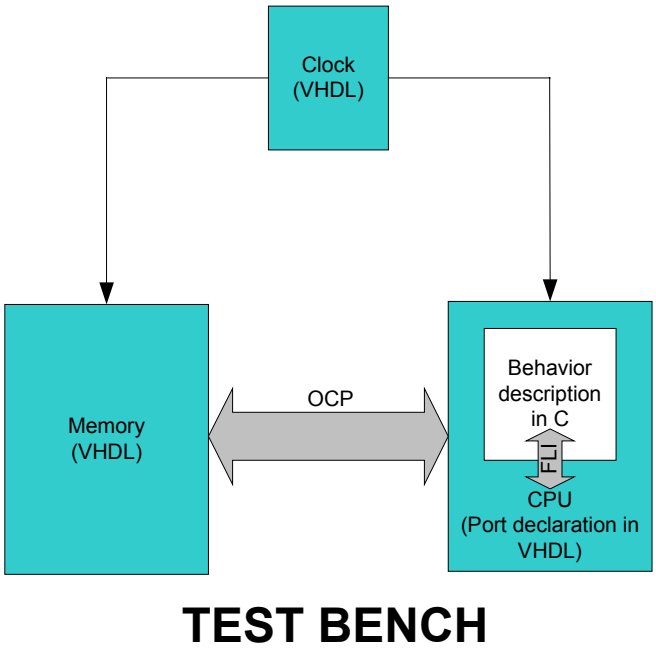


Figure 3.1

We start with the example of our test bench, which is our simulation environment. Our test bench consists of following entities:

- Clock
- CPU
- Memory

CPU is the entity whose whole behavior is described in C language. The Clock and Memory are purely in VHDL. We start with the VHDL description of the CPU. The test bench is shown in figure 3.1. The VHDL description of the CPU in which its ports and generics are declared, is shown in figure 3.2.

```
entity cpu is
  generic (
    input_file_C  :string :="";
    output_file_C :string :="";
    addr_width : integer ;
    data_width : integer );

  port (
    nreset              : in  bit;
    clk                 : in  bit;                                  -- OCP signal
    SCmdAccept          : in  bit;                                  -- OCP signal
    SResp               : in  bit_vector(1 downto 0);               -- OCP signal
    SData               : in  bit_vector(data_width-1 downto 0);    -- OCP signal
    MCmd                : out bit_vector(2 downto 0);               -- OCP signal
    MBurst              : out bit_vector(2 downto 0);               -- OCP signal
    MAddr               : out bit_vector(addr_width-1 downto 0);    -- OCP signal
    MData               : out bit_vector(data_width-1 downto 0);    -- OCP signal
    cpu_stall           : out bit;
    cpu_out             : out bit);

end cpu;


architecture behaviour of cpu is
    attribute foreign : string;
    attribute foreign of behaviour : architecture is "cpu_init ./ptool.sl ;";
begin
end;
```

Figure 3.2 VHDL description of CPU entity

This is all what we need to specify in VHDL.

To use the foreign language interface with C models, you first create and compile architecture with the FOREIGN attribute. The string value of the attribute is used to specify the name of a C initialization function and the name of an object file to load. When ModelSim elaborates the architecture, the initialization function is called. Parameters to the function include a list of ports and a list of generics.

Starting with VHDL93, the FOREIGN language attribute is declared in package STANDARD. With the 1987 version, you need to declare the attribute yourself. You can declare it in a separate package, or you can declare it in the architecture that you are replacing. (This will also work with VHDL93).

The value of the FOREIGN attribute is a string containing two parts. For the following declaration:
attribute foreign of behaviour : architecture is "cpu_init ./ptool.sl";

The attribute string parses this way:

cpu_init
The name of the initialization function for this architecture. This part is required.

ptool.sl
The path to the shared object file to load. This part is required.

In our example we have used only two parameters and these are both required. Some optional parameters can also be used whose details can be found from the ModelSim manual.

In our example the cpu_init() function is written in the C file ptool.c. The file ptool.sl is generated by compiling and linking ptool.c using GNU compiler. So when the ModelSim loads the CPU entity it searches for the file ptool.sl where all the behavior of the CPU entity is elaborated. Normally the .so extension is used but on HP machines (which we are using) .sl extension is used.

If the initialization function has a leading '+' or '-', the VHDL architecture body will be elaborated in addition to the foreign module. If '+' is used (as in the example below), the VHDL will be elaborated first. If '-' is used, the VHDL will be elaborated after the foreign initialization function is called.

3.1.1 The C initialization function

This is the entry point into the foreign C model. The initialization function typically:

- Allocates memory to hold variables for the instance
- Registers a callback function to free the memory when ModelSim is restarted
- Saves the handles to the signals in the port list
- Creates drivers on the ports that will be driven
- Creates one or more processes (a C function that can be called when a signal changes)
- Sensitizes each process to a list of signals

The ModelSim FLI has provided a lot of library functions and types, which are declared in the header file **mti.h**. The type mtiSignalIdt is used for the input signals of the entity (in VHDL) and the type mtiDriverIdt is used for the output signals. In the VHDL description of the CPU entity the signals nreset, clk, SCmdAccept, SResp and SData are the input signals so they are declared as type mtiSignalIdT and the output signals of the cpu entity are MCmd, MBurst, MAddr MData, cpu_stall and cpu_out and they are declared as types mtiDriverIdT in the C program of the CPU model. So these are declared

as shown below in figure 3.3. *Note that names of the signals should be the same as in VHDL.*

```
typedef struct {

   mtiSignalIdT nreset;
   mtiSignalIdT clk;
   mtiSignalIdT SCmdAccept;
   mtiSignalIdT SResp;
   mtiSignalIdT SData;
   mtiDriverIdT MCmd;
   mtiDriverIdT MBurst;
   mtiDriverIdT MAddr;
   mtiDriverIdT MData;
   mtiDriverIdT cpu_stall;
   mtiDriverIdT cpu_out;

} inst_rec;
```

Figure 3.3 : Declaration in C

Now we look at the `cpu_init()` function. This function is executed when we load the CPU entity in ModelSim .In addition to specifying inputs, outputs and sensitivity we can also use that function to initialize other variables which has to be used later in the code. We can also add file read/write functions and anything, which we need to use just like a normal C program. Only the part of the `cpu_init()` function, which is used to interface with VHDL, is shown in figure 3.4.

```
void cpu_init(
  mtiRegionIdT       region,
  char               *param,
  mtiInterfaceListT *generics,
  mtiInterfaceListT *ports
)
{
  inst_rec     *cpu_ip;
  mtiProcessIdT cpu_proc;
  mtiSignalIdT  outp;
  cpu_ip = (inst_rec *)mti_Malloc(sizeof(inst_rec));
  mti_AddRestartCB(mti_Free, cpu_ip);

  /* Here we can add our additional code. This can be found in the Appendix. */
    cpu_ip->clk    = mti_FindPort(ports, "clk");
    cpu_ip->nreset = mti_FindPort(ports, "nreset");
    cpu_ip->SCmdAccept = mti_FindPort(ports, "SCmdAccept");
    cpu_ip->SResp = mti_FindPort(ports, "SResp");
    cpu_ip->SData = mti_FindPort(ports, "SData");
    outp = mti_FindPort( ports, "cpu_out" );
    cpu_ip->cpu_out = mti_CreateDriver( outp );

    cpu_ip->MCmd   = mti_CreateDriver(mti_FindPort(ports, "MCmd"));
    cpu_ip->MBurst = mti_CreateDriver(mti_FindPort(ports, "MBurst"));
    cpu_ip->MAddr  = mti_CreateDriver(mti_FindPort(ports, "MAddr"));
    cpu_ip->MData  = mti_CreateDriver(mti_FindPort(ports, "MData"));
    cpu_ip->cpu_stall = mti_CreateDriver(mti_FindPort(ports, "cpu_stall"));

    cpu_proc = mti_CreateProcess("cpu_process", cpu_process, cpu_ip);
    mti_Sensitize(cpu_proc, cpu_ip->clk, MTI_EVENT);
    mti_Sensitize(cpu_proc, cpu_ip->nreset, MTI_EVENT);
}
```

Figure 3.4 : C initialization function cpu_init()

We have declared a pointer *cpu_ip of type inst_rec. This pointer is used to point the variables of inst_rec, which are basically the input and output signals of the CPU entity. Then we have used ModelSim FLI library functions whose detail is given below:

void mti_AddRestartCB(mtiVoidFuncPtrT func, void *param)

Causes the specified function to be called before the simulator is restarted. The function is passed the parameter specified by “param”, and it should free any memory that was allocated.

void *mti_Malloc(unsigned long size)

Allocates a block of memory of the specified size and returns a pointer to it. The memory is initialized to zero. On restore, the memory block is guaranteed to be restored to the same location with the values contained at the time of the checkpoint. This memory can be freed by mti_Free(). It cannot be freed by a call to the free() C-library function.

mtiSignalIdT mti_FindPort(mtiInterfaceListT *list, char *name)

This function searches linearly through the specified interface list and returns the signal ID of the port whose name matches the one specified. It returns NULL if it does not find the port. The search is not case-sensitive. So this function is used for all the input and output ports of the CPU entity. As shown in the code of cpu_init() function, all the input and output signals nreset, clk, SCmdAccept, SResp, SData, MCmd, Mburst, Maddr, Mdata, cpu_stall and cpu_out are specified using mti_FindPort() function.

mtiDriverIdT mti_CreateDriver(mtiSignalIdT sig)

Creates a driver on a signal. A driver must be created for a resolved signal in order to be able to drive values onto that signal and have the values be resolved. Multiple drivers can be created for a resolved signal, but no more than one driver can be created for an unresolved signal. This function is used for all the output signals of the entity. In our example the output signals are MCmd, MBurst, MAddr, MData, cpu_stall and cpu_out, which are used alongwith mti_FindPort() function.

mtiProcessIdT mti_CreateProcess(char *name, mtiVoidFuncPtrT func, void *param)

Creates a new process. The parameter "name" is the name that will appear in the Simulator's process window, which in our case is cpu_process. If the process is created during elaboration, the specified function will be called at time 0 after all the signals have been initialized. The mti_Sensitize() and mti_ScheduleWakeup( ) functions can be used to cause the function to be called at other times. When the function is called, it is passed the parameter specified by "param".

void mti_Sensitize(mtiProcessIdT proc, mtiSignalIdT sig, mtiProcessTriggerT when)

Causes the specified process to be called when the specified signal is updated. If the *when* parameter is MTI_EVENT, then the process is called when the signal changes value. If the *when* parameter is MTI_ACTIVE, then the process is called whenever the signal is active. *Since the CPU is strictly sequential so it is only sensitive to nreset (reset signal) and clk (clock) signal. So whenever the nreset or clk signal changes this process cpu_process() function is called. The nreset signal is only used at the beginning for initialization.*

Input sensitive function

So cpu_init() function is used for initialization and cpu_process() function is used to describe the functionality of the CPU on every clock cycle. Again, only the portion of cpu_process() related to FLI is shown here.

```
static void cpu_process( inst_rec *cpu_ip )
{
    int clk,nreset;
    int SCmdAccept;
    int count,stall;

    char SData[DATA_WIDTH],MData[DATA_WIDTH],MAddr[ADDR_WIDTH];
    char MCmd[3];
    char SResp[2];

    clk      = mti_GetSignalValue( cpu_ip->clk );
    nreset   = mti_GetSignalValue( cpu_ip->nreset );
    SCmdAccept = mti_GetSignalValue( cpu_ip->SCmdAccept);

    mti_GetArraySignalValue(cpu_ip->SResp,SResp);
    mti_GetArraySignalValue(cpu_ip->SData,SData);

    convert(&clk,1);
    convert(&nreset,1);
    convert(&SCmdAccept,1);

  count=clk;
  if(nreset==0)
    { count=0;
      stall=0;
      bus_active=0;
      convert(&count,0);
      convert(&stall,0);
      MCmd[0]=MCmd[1]=MCmd[2]= BIT_0;
      mti_ScheduleDriver( cpu_ip->MCmd,(long)MCmd, 0, MTI_INERTIAL );
      mti_ScheduleDriver( cpu_ip->cpu_out,count, 0, MTI_INERTIAL );
      mti_ScheduleDriver( cpu_ip->cpu_stall,stall, 0, MTI_INERTIAL );
    }
  else if(clk==1)
    {convert(&count,0);
     convert(&stall,0);

/* the functionality of cpu is described here. Complete version is
given in  Appendix*/

    mti_ScheduleDriver( cpu_ip->cpu_out,count, 0, MTI_INERTIAL );
    mti_ScheduleDriver( cpu_ip->cpu_stall,stall, 0, MTI_INERTIAL );

    mti_ScheduleDriver( cpu_ip->MCmd,(long)MCmd, 0, MTI_INERTIAL );
    mti_ScheduleDriver( cpu_ip->MAddr,(long)MAddr, 0, MTI_INERTIAL );
    mti_ScheduleDriver( cpu_ip->MData,(long)MData, 0, MTI_INERTIAL );
      }
  }
}
```

Figure 3.5 cpu_process()

As evident the process cpu_process only runs when there is a change in clock signal. To make it sensitive only with the positive edge of the clock, all the code is written under the "if" condition: clk=1. For all the input and output signals the corresponding variables have been declared. Hence the signals nreset, clk and SCmdAccept, which are single bit signals, are declared as integers of type int. And the signals having more than one bit SData, MData, MAddr, MCmd and SResp as arrays of type char having length equal to their bit width. The input variables get the values from the VHDL environment from the FLI functions mti_GetSignalValue() and mti_GetArraySignalValue(). These variables have been updated every time when the process runs and are then assigned to their corresponding signals by the function mti_ScheduleDriver(). All the input signals to CPU entity are generated by the entities, which are in VHDL. By using the FLI library functions (which are used in cpu_init() function and cpu_process() function) these values of these signals are assigned to their corresponding variables in cpu_process() function. Based on these values we update the output variables whose values are in turn assigned to the output signals of CPU entity. These output signals are inputs to memory, which is again an entity fully in VHDL.

3.1.2 Some important issues

1. The logic level '1' in VHDL is equivalent to 3 in C and logic level '0' is equivalent to We use our own simple function convert() which convert the input signal values from integer 3 to integer 1, integer 2 to integer 0 and vice versa for the output signals. This is done just to make the code more readable.

2. As described earlier that the signals having more than one bit have been used as characters. These signals are of type bit_vector in VHDL. To deal with these signals in C, we have declared a type of enumerated data.

```
typedef enum {BIT_0, BIT_1} bit;
```

If we want to assign logic level '0' to a particular bit we assign it BIT_0 and for logic level '1' we assign BIT_1. Thus if we want to assign "001" to the three bit signal MCmd, it would be as follows:

```
MCmd[0]=BIT_0;
MCmd[1]=BIT_1;
MCmd[2]=BIT_1;
```

3. Since cpu_process() is basically a C function so all the variables declared inside are actually local variables i.e. they lost their values at the end of the function. Therefore if we want for some variables to retain their values after the function, they must be declared globally.

4. To get the values of generics specified in VHDL, the pointer *generics is used as shown in C initialization function cpu_init(). The pointer *generics is of type mtiInterfaceListT which is declared in the header file mti.h. Consider the example of our model in which the two generics specified in VHDL are input_file_C and output_file_C

and they are of type string. As evident from the name, the user provides the names of the input and output files through these generics. The name of the files can be retrieved in the C program by using the *generics pointer as shown below:

```
if( (fptr1= fopen(generics->u.generic_array_value,"r")) ==NULL)
    {
     printf("\n Couldn't open the file %s",generics->u.generic_array_value);
    }

if( (fptr2= fopen(generics->nxt->u.generic_array_value,"w")) ==NULL)
    {

     printf("\n Couldn't open the file %s",generics->nxt->u.generic_array_value);
    }
```

A linked list is created in header file mti.h and successive generic elements can be retrieved by using the pointers shown in the above example.

3.2 SUMMARY

The FLI is used to make communication between entities written in C and in VHDL. The entity whose behavior is described in C is still has its port declaration in VHDL. But in its architecture description in VHDL, the name of the 'C initialization function' and 'C shared file' containing this function is specified. In the 'C shared file' the complete behavior of the entity is elaborated. When the entity is loaded in the simulator the control is transferred to that 'C initialization function'. This function uses FLI library functions to transfer the information about the ports and generics of the entity, which are specified in its VHDL declaration. It also specifies different functions, which are sensitive to the input signals of this entity. Whenever these inputs change, the corresponding functions are executed. Whenever these functions are executed they update some variables, which are basically the updated values of the output signals of the entity. The FLI library functions then convert these values according to VHDL standards so that can be understood by the entities written purely in VHDL and vice versa.

In this chapter we have described, how we can simulate in a mixed language environment by using FLI. The FLI has lot of features and options available and there is lot of ways to do same things. We have used which is most simple and we have covered only those features, which are required in our task.

REFERENCES

Foreign Language Interface (FLI), Mentors Graphics' ModelSim Technical Manual.

CHAPTER

4 An Overview of the CPU Model

Now we know about OCP and FLI and also how they are used in the model. In this chapter, we will describe the overall structure and working of our CPU model. We start with some features of general-purpose microprocessors and then we will explain how these features are included in our model.

4.1 An overview of the General-purpose Microprocessor

To make a model of the processor we need to study the features and functionality of the processors used today. All processors today are pipelined and have separate instruction and data cache. In a given clock cycle more than one instruction are in progress. Their number depends on the depth of the pipeline. The modern processors used today are very complex. To increase the throughput the parallelism among instructions is exploited by out of order execution [1] and more than one instruction may be issued in every clock cycle [2]. However, in our model we haven't targeted these architectures. Our model is focused on the processors used in embedded applications. These processors are not as much complex like the processors used in Personal Computers, Workstations or High end Servers. Their architecture is much simpler than that and the pipeline scheme is also not that much complex. The instruction execution is also in order and in a given pipeline stage only one instruction is active. In most embedded processors the pipeline depth is between 5 to 8 stages [3]. ARM processor is one of them, which is largely used, in embedded applications. Therefore we can focus ourselves to study the behavior of these processors and try to include this behavior in our model.

Let's have a quick look on a five stage pipelined processor [4] with separate instruction and data cache. The first stage of the pipeline is called Instruction Fetch stage in which the processor fetches instruction from the instruction cache by generating the instruction address. In some processors this Instruction Fetch stage is further divided into two stages to decrease the clock period. The next step is to decode the instruction, which is normally

called Instruction Decode (ID) stage. The instruction is decoded and the specified registers are accessed. In most architectures if the instruction is Jump or Branch some action has been taken in the ID stage that involve change in the next instruction target address. The next stage is called the Execution stage where the instruction is treated according to its type. If it is an R-Format instruction (instruction that require arithmetic operation by ALU on its operands) then some arithmetic or logical operation is performed in the ALU (Arithmetic Logic Unit) and in the case of a load or store instruction the memory address has been calculated. The next stage is the memory stage where the data cache is accessed for either data read or data write. And the last stage is Write Back stage where the output of the ALU or the data from the memory is written into the register file.

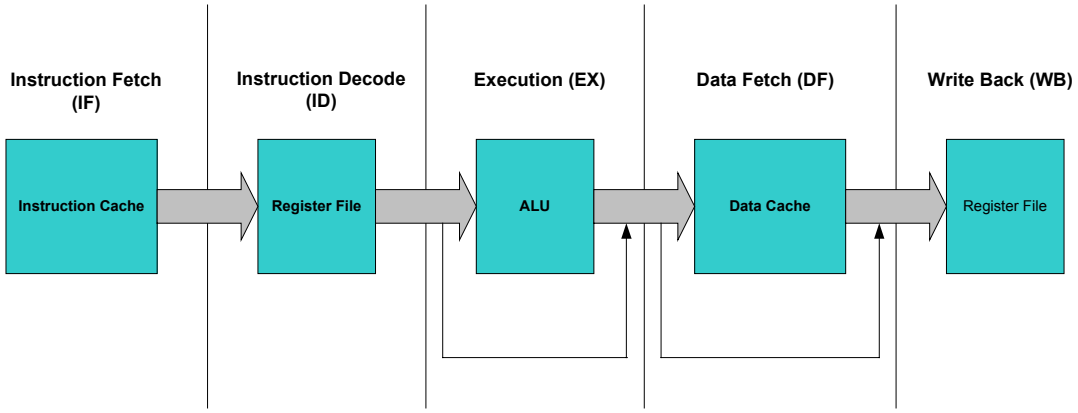


Figure 4.1 Five-stage Pipeline MIPS R1000

In the ideal case, every instruction consumes one clock cycle and the CPI (cycles per instruction) is 1. In Superscalar machines where more than one instruction can be issued in a clock cycle the ideal CPI can also be decreased from 1. However, these machines are not the targets in our model. The actual CPI is always greater than 1 due to structural, data and control hazards. There are also instructions like multiply/divide that consume several clock cycles, and depending upon the frequency of these instructions the CPI increases. But the most dominant factor is the misses produced during the execution.

There are only two cases when the processor needs to communicate with the memory:

- When the instruction miss occurs
- When the data miss occurs

The instruction cache is accessed in every clock cycle. So there is always a probability that an instruction miss can occur. Thus in the first stage if the instruction is not found in the instruction cache, instruction miss occurs and the CPU has to fetch the instruction from the memory. This involves consumption of many cycles depending on the latency of the memory. When the instruction is brought in the cache the normal execution continues.

Data miss can only occur in the case of load or store instruction and also when these instructions result in a data miss. All other types of instructions that do not access data cache cannot generate data misses. Because of these reasons, instruction misses are higher than data misses in most programs. However, the opposite is also true in many programs. In the case of load or store instruction, when the miss arises in the data cache, the pipeline is stalled and remains stalled until the required data reaches the data cache from the memory. There are two types of data misses: read misses and write misses. In the case of a read miss the CPU initiates communication with the memory to read some data elements while in the case of a write miss the CPU initiates communication to write on some particular location in the memory.

The instruction miss and the data read miss are similar in terms of communication with the memory. Whenever any of these misses occur, the access to the memory is 'burst access'. An access to the memory brings the next four instructions or data items from the memory into the cache to exploit spatial locality. However, write accesses to the memory are not burst accesses. When memory is accessed for write operation, only one location in the memory is updated. The 'burst access' occupies the bus for a greater number of clock cycles as compared to normal access as shown in section 2.1.3 where both the normal and burst OCP accesses are shown.

There are two kinds of write policies in cache designs: write through and write back. In write through scheme, the information is written to both the block in the cache and to the block in the memory. So in this scheme every store instruction results in a write miss. In write back scheme, the information is written only to the block in the cache. The modified block is written to the memory only when it is replaced.

So whenever a miss occurs, the pipeline is stalled and remains stalled until the corresponding miss is handled. It may happen that the instruction and data miss occurs at the same time then both the misses are handled one by one with the data miss to be handled first so that the pipeline can continue. These factors increase the completion time of an individual instruction, which in ideal case is 1 clock cycle. *We can say that if there is no miss or no other hazard then at the end of every clock cycle one instruction is completed other wise it is not.*

Some processors also have second-level cache that is bigger than the first-level caches. It can be a unified cache that is used for both data and instruction or it may be a case that there are separate instruction and data second-level caches. When there is a miss in the first-level cache then before accessing the memory, second-level cache is accessed. If the required instruction or data is found in the second-level cache it is supplied to the first-level cache otherwise the memory is accessed. In the case of a hit in the second-level cache the penalty for accessing the second-level cache is lesser than accessing the memory. But in the worst case when there is a miss in the second-level cache the miss penalty is the number of clock cycles to access the second-level cache plus the number of clock cycles to access the memory. The effective miss rate in the presence of second-level cache is the product of the individual miss rates of both first-level and second-level caches.

As described in the first chapter, our aim is to generate traffic on the bus in a realistic manner. Therefore, in the above discussion we have focused ourselves on the factors, which generate traffic on the bus connecting the CPU and memory. Following are the important observations from the above discussion:

- In ideal case, every instruction requires 1 clock cycle in a pipelined processor.
- In every clock cycle, the number of active instructions is usually equal to the depth of the pipeline.
- In every clock cycle, there is a probability that an instruction miss or data miss or both can occur.
- When a miss occurs, memory is accessed and the pipeline is stalled. The pipeline remains stalled until the required data is brought into the cache.
- Instruction misses and data read misses generate burst access to the memory while the write misses generate single access.

Based on this information we made our model which include the behavior of all these phenomenon discussed above

4.2 Structure and working of the model

We will start the discussion of our model with the explanation of the basic input parameters of the CPU model and then we will show how the necessary data is collected from these inputs. Later, we will show how the model is organized into different parts and then we will describe the overall sequence of operation during the whole simulation.

4.2.1 Basic inputs of the model

The basic input parameters are supplied by the user to carry out the simulation. These inputs are written in a specially designed file, which is made for this model. These basic inputs are the required inputs and are necessary to carry out the simulation. In addition to these basic inputs there are also some optional inputs, which will be discussed in the next chapter. The basic inputs are given below:

- Total number of instructions
- Percentage of load instructions
- Percentage of store instructions
- Instruction miss rate
- Data miss rate
- CPI (including the effect of structural, data and control hazards)
- Memory Access startup latency (Number of cycles required to initiate memory transfer after detecting a miss)
- Memory Accesses end latency (Number of cycles required to restore the normal execution after the completion of memory transfer)
- Seed for the random functions

The ‘total number of instructions’ tells us about the total number of instructions which we have to simulate. These ‘total number of instructions’ are not the total number of instructions, which we can found in the assembly program. They are the total number of instructions, which are actually executed. For e.g., if there is a loop consisting of 10 instructions and that loop runs for 100 times then the total number of instructions that are executed are 1000 not 10.

‘Percentage of load instructions’ and ‘Percentage of store instructions’ tell us how many instructions are load/store from the total number of instructions. These are the instructions that may cause a data miss. These two parameters and ‘Data miss rate’ also help us to distinguish between read and write misses as these two misses have different behaviors on the bus.

‘Instruction miss rate’ and ‘Data miss rate’ tell us about how many data and instruction misses are generated during the execution of the total number of instructions. We have to generate these misses in a manner that resembles to a real traffic.

The basic input CPI (cycles per instruction) tells the CPI without taking into account the effect of the misses. However, the CPU model demands that this input CPI includes the effect of different data, control and structural hazards. The reason for this demand is that the CPU model can’t include the effect of these hazards by itself because of the absence of real software. However, even if the user provides the ideal CPI it doesn’t have a very remarkable affect on the simulation. Therefore, the ideal CPI can also serves the purpose.

‘Memory Access startup latency’ is the number of clock cycles required to initiate memory transfer after detecting a miss. Nearly all processors, after detecting a miss can’t initiate communication with the memory in the same cycle. Generally it is started at the start of the next cycle and in some processors depending upon their pipeline technique more than one clock cycle is required. Similarly, ‘Memory Accesses end latency’ is the number of clock cycles required to restore the normal execution after the completion of the memory transfer and is different for different processors.

The input ‘Seed for random functions’ is any integer number, which is used as a seed in different random functions in the C program of the model.

4.2.2 Execution clock cycles

By using the values of these inputs the CPU model generates traffic on the bus. Based on these parameters we can approximate the total number of clock cycles required to complete the given number of instructions. The total number of clock cycles in which the CPU does some useful work is given by (let’s call it Execution clock cycles):

$$\text{Execution clock cycles} = \text{CPI} \times \text{Total number of instructions} \tag{4.1}$$

The parameter ‘Execution clock cycles’ is one of the most important parameters of our model. The ‘Execution clock cycles’ to complete the given number of instructions remain

constant and is independent of memory latency and bus arbitration. In this chapter and in the next chapter, whenever we use the term 'Execution clock cycles', we mean the effective clock cycles in which the CPU is perfectly in execution i.e. the number of clock cycles during which the CPU is not stalled.

If we take into account the misses occurred during the execution, the total number of clock cycles will be:

$$\textit{Total number of clock cycles} = \textit{Execution clock cycles} + \textit{Total number of misses} \times \textit{Miss penalty} \qquad (4.2)$$

Where,

Total number of misses = instruction misses + data misses

$$\textit{Total Execution time} = \textit{Total number of clock cycles} \times \textit{Clock period} \qquad (4.3)$$

Thus before the start of the simulation, we can calculate the Execution clock cycles from the CPI and the total number of instructions, which are the user-supplied inputs. We also know the total number of misses, which have to be introduced within these Execution clock cycles. However, we can't calculate the Total number of clock cycles by using equation 4.2 because we don't know about the Miss penalty i.e. the number of clock cycles required to access the memory. Different memories have different miss penalties and the miss penalty for the same memory may be different in the same simulation depending upon the traffic on the bus. It may happen that when CPU initiates communication with the memory, the memory is busy handling request from some other component of the SOC design and causes the CPU to stall for the clock cycles more than the latency of the memory. Hence we can only know about the total number of consumed clock cycles (to execute the given number of instructions) only at the end of the simulation. However, we can calculate the Total execution cycles and we can use it in our model.

As already described, no real software is running on our model, so during the simulation we have to make some guess whether in a given cycle we have any kind of miss or not. If we don't have a miss we can say that one instruction is completed other wise instruction is not completed. This process is repeated until the given number of instructions has been completed i.e. the Total execution cycles are completed. Chapter 5 is dedicated to the discussion about how we make the decision about a miss in a given cycle.

Before the start of the actual simulation, the CPU model is initialized with the total number of execution cycles and the total number of misses and with lots of other data elements that will be used during the simulation. A variable is initialized that counts the execution clock cycles. This counter increment by one in every useful clock cycle. When this count becomes equal to the Total execution clock cycles, the simulation is ended. In every clock cycle separate functions are called to make decision about the misses. If there is no miss then the normal execution continues. In the other case, when a miss has been found the model initiates communication with memory. During this whole memory

access, the CPU remains stalled i.e. not doing useful work. So the execution clock cycle counter does not increment during this period. When the memory transfer is completed the normal execution continues i.e. execution clock cycle counter starts incrementing again after every useful clock cycle.

4.2.3 Three major parts of the model

The whole model is divided into three major parts. Each part consists of a group of functions to carry out a particular task.

- A set of initialization functions that takes control from VHDL through FLI. This part is explained in chapter 3.

- A set of functions that communicates with the outside world like memory through OCP. This involves updating of the signals that communicates with the memory and the variables that keep track of the transitions occurred during the handshaking. The signals transitions are according to OCP as explained in chapter 2.

- A set of functions that takes care of the generation of misses during the simulation. They are divided into parts. The first part consists of initialization functions that initializes the model with the input parameters and makes necessary calculation that will help in the rest of the simulation. The second part consists of functions that make decision about a miss in a given cycle

The third part contributes the major part of the model and is explained in Chapter 5.

4.2.4 Sequence of Operations

The whole simulation is carried out in the following sequence. All these steps are shown in figure 4.2.

1. Initialize the CPU model with the variables that contain the values of the input parameters i.e. total number of instructions, load/store instructions, instruction miss rate, data miss rate etc and make some necessary calculations. The inputs are read from the file by using C file read functions. The initialization takes place when we load the design in the simulator (Modelsim) window. This step is called 'Initialization' and is shown as the grey shaded region in figure 4.2. This part is explained in detail in Chapter 5.

All the remaining steps are performed during the simulation i.e. in each clock cycle the model undergoes the following steps.

2. Check at the start of each cycle if the CPU is in some stage of memory transfer. This can be seen by checking the variable, which shows whether CPU is stalled or

not. If the CPU is stalled then it is already in the communication phase i.e. communicating with the memory through OCP, which requires some cycles to complete. In that case call the function bus_interface_process(), which is responsible for the memory transfer. The function also checks the current status of the handshaking and takes necessary actions accordingly. This step is repeated in every clock cycle until the memory transfer is completed. If the memory transfer is completed then clear the variables that contain information about the misses that needs to be handled i.e. sets them to 0. If the CPU is not currently involved in memory transfer or the memory transfer is completed then go to the next step.

3. Increment the Execution clock cycle count.

4. Call the functions that make decision about the introduction of instruction and data misses in a given cycle. Update these variables according to the decision i.e. set them to 1 if a miss has been found or 0 if there is no miss. This step is also indicated as a grey shaded region in figure 4.2 and is explained in detail in the next chapter.

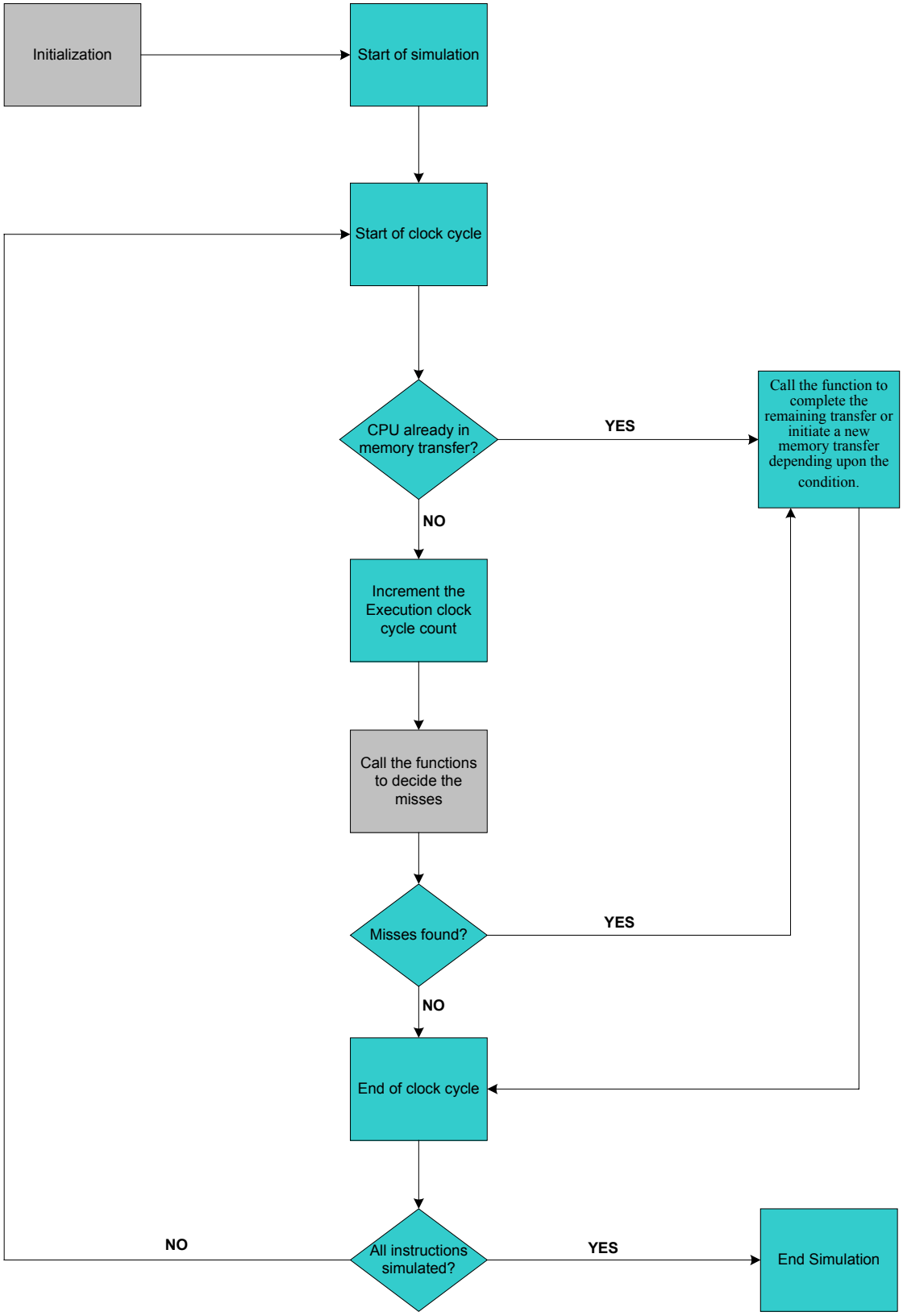
5. Check the variables that contain information about the instruction and data misses. If there is no miss then go to the next step otherwise go to step 8.

6. The clock cycle has ended. Save the values of all the variables that will be used in the next cycle. All necessary variables are declared globally so that their values will not be lost at the end of the function. Go to the next step.

7. Compare the value of the execution clock cycle count variable with the Total execution clock cycles value. If it is less than the total value then it means that the simulation is not ended yet and thus go back to step 2. If it is equal or greater than the total value then it means that all the instructions have been simulated so go to the step 9.

8. If a miss has been found then stall the processor model. Call the function that starts communication with the memory through OCP. Keep the processor stalled until the miss has been handled. This has been checked at the start of every cycle as shown in step2. If there is a miss in both the instruction and data then handle the data miss first and then handle the instruction miss. Go to step 6.

9. All the instructions have been simulated. Generate report about the total execution time of the instructions, the number of cycles in which the CPU remain stalled waiting for the memory response, the number of cycles the CPU is executing instructions, the number of clock cycles consumed in the longest handshake as well as in the shortest handshake etc. Terminate the simulation.



Flowchart of the complete process

Figure 4.2

4.3 SUMMARY

In this chapter, we have discussed those features of general-purpose processors, which are used in our CPU model. We have described what happens to the pipeline when a miss occurs. We have described the basic input parameters of the model and their purpose in the model. We have shown how the model is divided into three different parts and then at the end we have described the flowchart of the whole process.

From the basic parameters, the CPU calculates the total Execution clock cycles and the total number of instruction and data (read and write) misses. These misses have to be introduced by the model within these Execution clock cycles. During the simulation, the CPU model decides in every clock cycle whether a miss has to be introduced or not. The variable that counts the Execution clock cycles increments by one after every Execution clock cycle except when the CPU is stalled. When this count reaches the limit i.e. the total number of Execution clock cycles, it implies that all the instructions are completed. The model ensures that at this point all the misses have already been introduced. The simulation is then completed and the model generates the report, which tells the effective CPI that includes the effect of the misses. It also calculates the total number of clock cycles that are required to complete the execution of the given number of instructions, the number of clock cycles during which the CPU remains stalled and the longest and shortest time to complete the data transfer with the memory.

REFERENCES

[1] DLX using Tomasulo's algorithm, Computer Architecture A Quantitative Approach by John L Hennessy & David A Patterson

[2] Superscalar version of DLX, Computer Architecture A Quantitative Approach by John L Hennessy & David A Patterson

[3] ARM9 Embedded Trace Macrocell (ETM9) Technical Reference Manual

[4] MIPS R1000, Computer Organization and Design by John L Hennessy & David A Patterson

CHAPTER

5 Miss Generation Techniques

In the previous chapter we have explained the overall structure and working of the model. The model is divided into three parts. The first part consists of set of functions that are used to transform different signals and variables' values from C to VHDL and vice versa. The second part is responsible to communicate with the memory. The third part, which is the most important part, makes necessary calculations about the misses and introduces them during the simulation. The third part is the two grey shaded regions in fig 4.2. The first two parts have already been explained in chapters 2,3 and 4. In this chapter we will explain the third part i.e. the two grey shaded regions in fig 4.2. So this chapter includes the discussion about different techniques that are used to generate the misses in the simulation.

When a miss occurs, it generates traffic on the bus. Our main aim is to generate traffic that resembles to a real traffic that occurs as a result of execution of real software on real IP cores. And to accomplish this, we have to generate misses in a manner that resembles to the misses generated in a real program.

We start with a very brief overview of caches. As evident from the above discussion, it is assumed that you have read chapter 4.

5.1 An Overview of Caches

Following is a quick review of some cache related concepts. The intention here is not to describe caches but to show later how these particular cache-related concepts are used in our model and also why some of these concepts are not used in our model. So we assume that the reader has enough knowledge of caches and the related concepts. The interested reader can read more about caches from [1] and [2]. Most information in this section is also taken from [1] and [2].

5.1.1 Cache Basics

Cache is the first level of memory hierarchy. The next instruction (to be executed) or the data item is first searched into the cache. When the CPU finds a requested data item in the cache, it is called a *cache hit.* When the CPU does not find a data item it needs in the cache, a *cache miss* occurs. A fixed-size collection of data containing the requested word, called a *block,* is retrieved from the main memory and placed into the cache. *Temporal locality* tells us that the same word is likely to be needed again in the near future, so it is useful to place it in the cache where it can be accessed quickly. Because of *spatial locality*, there is high probability that the other data in the block will be needed soon.

The time required for the cache miss depends on both the latency and bandwidth of the memory. Latency determines the time to retrieve the first word of the block, and bandwidth determines the time to retrieve the rest of this block. Bandwidth can also be taken as the width of the bus connecting the cache or CPU to the memory. A cache miss is handled by hardware and causes processors following in-order execution to pause, or stall, until the data are available.

5.1.2 Division of address

Caches have an address tag on each block frame that gives the block address. The tag of every cache block that might contain the desired information is checked to see if it matches the block address from the CPU. As a rule, all possible tags are searched in parallel because speed is critical.

There must be a way to know that a cache block does not have valid information. The most common procedure is to add a *valid bit* to the tag to say whether or not this entry contains a valid address. If the bit is not set, there cannot be a match on this address.

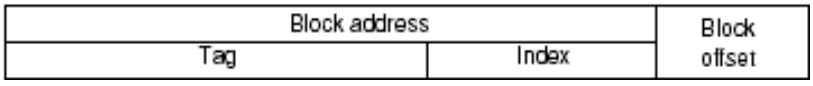


Figure 5.1 : Three portions of address in a set-associative or direct-mapped cache

Figure 5.1 shows how an address is divided. The first division is between the *block address* and the *block offset.* The block frame address can be further divided into the *tag* field and the *index* field. The block-offset field selects the desired data from the block, the index field selects the set, and the tag field is compared against it for a hit.

If the total cache size is kept the same, increasing associativity increases the number of blocks per set, thereby decreasing the size of the index and increasing the size of the tag. That is, the tag-index boundary in Figure 5.1 moves to the right with increasing associativity, with the end point of fully associative caches having no index field.

5.1.3 Miss Rate

Miss rate can be defined as the fraction of cache accesses that result in a miss (i.e., number of accesses that miss divided by number of accesses).

$$\text{Miss rate} = \frac{\text{Number of accesses that miss}}{\text{Total number of accesses}}$$

$$\text{Instruction miss rate} = \frac{\text{Number of instruction cache accesses that miss}}{\text{Total number of instructions}}$$

$$\text{Data miss rate} = \frac{\text{Number of data cache accesses that miss}}{\text{Total number of load/store instructions}}$$

5.1.4 Different Cache Configurations

There are three different types of cache configuration:

- Direct mapped
- Set Associative
- Fully Associative

If each block can appear at only one place in the cache, the cache is said to be *direct mapped*. The mapping is done according to the formula:

(Block address) MOD *(Number of blocks in cache)*

If a block can be placed anywhere in the cache, the cache is said to be *fully associative*.

If a block can be placed in a restricted set of places in the cache, the cache is *set associative*. A *set* is a group of blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within that set. The set is usually chosen by *bit selection*; that is,

(Block address) MOD *(Number of sets in cache)*

If there are *n* blocks in a set, the cache placement is called *n-way set associative*.

The range of caches from direct mapped to fully associative is really a continuum of levels of set associativity. Direct mapped is simply one-way set associative and a fully associative cache with *m* blocks could be called *m*-way set associative. Equivalently, direct mapped can be thought of as having *m* sets and fully associative as having one set.

5.1.5 Types of Misses

Compulsory: The very first access to a block *cannot be* in the cache, so the block must be brought into the cache. These are also called *cold start misses* or *first reference misses*.

Moving from direct mapped to full associativity has no impact on the compulsory misses. Larger block size can reduce the compulsory misses.

Capacity: If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur because of blocks being discarded and later retrieved.

A little can be done to improve the capacity misses except to increase the size of the cache.

Conflict: If the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if too many blocks map to its set. These misses are also called *collision misses* or *interference misses*. The idea is that hits in a fully associative cache which become misses in an N-way set associative cache are due to more than N requests on some popular sets.

Our objective here is not to study the ways by which these three kinds of misses can be improved. Our objective is to model these misses.

5.2 Modeling and distribution of the misses

Based on the issues and the concepts, which were discussed in the previous section, we will develop our model that is used to generate misses. The modeling of the misses consists of two parts.

- Initialization
- Simulation

The ‘Initialization’ part is done before the start of the actual simulation i.e. when the design containing the CPU core model is loaded in the simulator. The ‘Simulation’ part takes place during the actual simulation. During every clock cycle a decision is made about the introduction of a miss according to some rule described in the later section. The Initialization and the Simulation part can be identified in figure 4.2 as the grey shaded regions.

We start with the discussion of the ‘Initialization’ part, which include all the initial calculation of the model that takes place before the start of the simulation.

5.2.1 Initialization

In this part the total number of Execution clock cycles and the total number of instruction and data misses are calculated. The total number of Execution cycles is divided into many parts and the misses are distributed among these parts according to some procedure, which is described in this section. We have made several assumptions and decisions in this part. Our goal is to build some information that will be used to make decisions about the introduction of the misses during the 'Simulation' part. We will show all the steps of our calculations in the same order as they are performed in the model. The discussion of these steps is coupled with the reasons and the background theories, which enable us to make these steps and decisions.

5.2.1.1 Initial calculation of the basic input parameters

The model starts by reading the basic and optional inputs from the input file. Details about the input file are given in the appendix. As described in the previous chapter, our basic inputs include:

- Total number of instructions
- Percentage of load instructions
- Percentage of store instructions
- Instruction miss rate
- Data miss rate
- CPI (including the effect of structural, data and control hazards)
- Memory Access startup latency (Number of cycles required to initiate memory transfer after detecting a miss)
- Memory Accesses end latency (Number of cycles required to restore the normal execution after the completion of memory transfer)
- Seed for the random functions

We can obtain the necessary data from these basic inputs that can help in the rest of the simulation.

As described earlier, the total Execution clock cycles can be calculated as:

$$Execution\ clock\ cycles = CPI \times Total\ number\ of\ instructions \qquad (5.1)$$

These are the total number of clock cycles in which the CPU does some useful work i.e. the cycles, which are consumed in executing the instructions during which the CPU is not stalled. The execution clock cycles are independent of the memory latency and bus arbitration.

The total number of load/store instructions is given by:

$$\textit{Percentage of load/store instructions} = \textit{Percentage of load instructions} + \textit{Percentage of store instructions} \quad (5.2)$$

$$\textit{Total number of load/store instructions} = \textit{Percentage of load/store instructions} \times \textit{Total number of instructions} \quad (5.3)$$

Similarly, we can find:

$$\textit{Total number of load instructions} = \textit{Percentage of load instructions} \times \textit{Total number of instructions} \quad (5.4)$$

$$\textit{Total number of store instructions} = \textit{Percentage of store instructions} \times \textit{Total number of instructions} \quad (5.5)$$

We can calculate the total number of instruction and data misses which are generated during the execution of the given number of instructions:

$$\textit{Instruction misses} = \textit{Instruction miss rate} \times \textit{Total number of instructions} \quad (5.6)$$

$$\textit{Data misses} = \textit{Data miss rate} \times \textit{Total number of load/store instructions} \quad (5.7)$$

$$\textit{Write misses} = \textit{Total number of store instructions} \times \textit{data miss rate} \quad (5.8)$$

$$\textit{Percentage of write misses} = \textit{Write misses Data misses} \quad (5.9)$$

$$\textit{Read misses} = \textit{Data misses} - \textit{Write misses} \quad (5.10)$$

The Data cache Access rate can be calculated by dividing the total number of load/store instructions with the total number of instructions.

$$\textit{Data cache Access rate} = \frac{\textit{Total number of load/store instructions}}{\textit{Total number of instructions}} \quad (5.11)$$

$$\textit{Total misses} = \textit{Instruction misses} + \textit{Data misses} \quad (5.12)$$

Now we have the following parameters, which will be used during the rest of the calculations and during the simulation.

- Execution clock cycles
- Instruction miss rate
- Data miss rate
- Data cache Access rate
- Instruction misses
- Data misses

- Read misses
- Write misses
- Seed for different random functions

From the basic inputs, we have calculated the Execution clock cycles and the total number of instruction and data misses. Now we have to distribute these misses within these Execution clock cycles. *No matter how we distribute these misses in these execution clock cycles; the miss rate would remain the same. Our job is to introduce these misses in such a way that they generate traffic on the bus that looks similar to the traffic that arises as a result of the execution of real softwares on real IP cores of the processors.*

Before proceeding to the next steps of our initial calculations, we need to discuss some issues, which forces us to make some assumptions.

5.2.1.2 Some Basic Issues

The misses that are generated during the program execution depends on two factors.

- Cache configuration (associativity, cache size, block size, number of blocks)
- Software

To introduce the misses in a way that looks like the misses due to the real program, we need to study the effect of the different cache configurations and different software routines running on the CPU.

If we want to calculate a miss in a given cycle by using the block placement formulas of the direct mapped or set associative caches given in section 5.1.4, we must know the Block Address and the number of cache blocks in direct mapped cache while in the case of set associative or fully associative caches we need to know about the Block address and the number of sets in the cache. Number of blocks or the number of sets can be made user supplied inputs but we can't know the block address because we are not running any real software. Same software generates different number of misses on different cache configurations. Therefore we can't use the conventional methods which are used in most simulators that take an address trace of the instruction and data references and cache configuration, simulate the cache behavior to determine which references hit and which miss, and then report the hit and miss totals.

Direct mapped caches have higher miss rates than set associative and fully associative caches. Increasing associativity of the cache always result in decrease in miss rate. However set associative caches are little bit slow than the direct mapped caches resulting in an increase in the clock period.

The cache configuration has a direct impact on the miss rate, which is also one of the basic input parameter of our model. Since miss rate is itself an input parameter, there is no need to include cache configuration as an input parameter. We can't make use of it any way without executing the real software. Therefore when the user enters the instruction miss rate and the data miss rate, we assume that it includes the effect of

different cache configurations. This miss rate is the most important parameter and the whole behavior of the model depends on its value.

Similarly, the second-level cache decreases the total accesses to the memory. In the presence of second-level cache the global miss rate is the actual miss rate, which is the product of first-level cache miss rate and the second-level cache miss rate. Therefore it is assumed that a small value of instruction and data miss rate as an input parameter can approximate the effect of second-level cache.

The second factor that affects on the number of misses are the software routines. All software routines are combinations of loops, conditional statements, functions, arrays, pointers and structures. Although we don't know the software but still it is possible to model the behavior due to these basic programming constructs. We have already discussed the three C's of misses (compulsory, capacity, conflict) in section 5.1.5. The misses introduced by these software routines are the combination of these three different kinds of misses. For example, an instruction miss occurs when an instruction is not found in the cache. There can be many reasons of an instruction miss. Either it is the first time that this instruction is accessed (compulsory miss). The program reaches to this instruction either through sequential execution or because of a jump (either conditional or non-conditional). If it is the first time that this part of code is accessed then the bunch of misses would occur when the next sequential instructions are accessed (compulsory misses). It may happen that this part of code was in the cache before but due to conflict it was replaced by some other instructions (conflict misses). The important point to note here is that this sort of behavior is present in most programs and we can introduce some misses in our model that resembles to the misses originated because of the above-mentioned behavior. In the same manner most data misses are produced in loops and when arrays are accessed. We can also introduce data misses in our model, which resemble to the misses originated when different parts of the arrays are accessed and when they are accessed in a loop. These two cases are not the only behaviors that can be modeled approximately. There are many that can be modeled and they will be discussed in the later sections. Since we don't know the software we can't say which behavior is mostly present in a program unless specified by the user. What we can do is to introduce these behaviors in significant proportions in the given number of available Execution clock cycles. It is also not possible to model all different behaviors but they are present in the programs and we can't just eliminate them. Summarizing, we can say that there are many software behaviors that can be modeled approximately, but there are many of them, which can't be modeled. Remember that the total number of instructions and the total number of instruction and data misses are constant. We can distribute these misses to represent different behaviors. We have to decide that in a given number of Execution clock cycles how we introduce these special behaviors and in which proportions and what we have to do about the behaviors, which can't be modeled.

5.2.1.3 Three modes of simulation

Based on what we can model and what we can't model, we divide the whole execution sequence into three major modes. In other words, we can say that the total Execution cycles and total number of instruction and data misses are divided into three parts and are assigned to these modes. However, the distribution among the three modes is not even. The three modes are:

- Compulsory mode
- Random mode
- Special mode

The *compulsory mode* is used to model the compulsory misses. This mode is introduced at the beginning of the simulation.

Random mode is used for the behavior, which we can't model. Therefore when this mode is in execution (during simulation) it makes decisions about the misses based on randomness. A full detail of this mode is described in the next section. The results have shown that the traffic generated by this mode resembles like a real traffic.

Special mode introduces a behavior due to specific software routines. It is further divided into many small modes each representing a special kind of behavior.

During the simulation, after the completion of Compulsory mode, the execution sequence switches between Random and Special modes (within different modes of Special mode each representing some particular behavior) till the end of the total Execution cycles.

During this whole 5.2.1.3 section, we will describe how we divide this total number of Execution clock cycles and total instruction and data misses among these modes. As written many times, our goal is to generate traffic on the bus in a realistic manner. And to achieve this, we have made some rules for this division of Execution clock cycles and the misses. Therefore in this whole section, we will describe how we are doing this division and why it is necessary to make such division.

C Random function

During the calculation and the simulation C random function is used many times. These functions generate pseudo random numbers in the range 0 to RAND_MAX. There are many C random functions and are classified on the basis of the range of pseudo random numbers they produce, number of arguments and whether they are thread safe or not. We have used C rand_r() function. Its prototype declaration is:

rand_r(int &,int &)

And can be used as:

rand_r(&seed,&randomvalue);

Every time this function is called the integer variable 'randomvalue' gets some pseudo random number. The sequence of the pseudo random numbers depends on the value of the seed. This integer variable seed gets its value from the user and is also one of the basic input parameter. A different value of seed every time when the program runs, means that every time different sequence of pseudo random number will be generated. If the same value of seed is used again, the same sequence will be generated. For this reason, the 'seed' is user-defined parameter.

Now we will calculate the Execution clock cycles, instruction and data misses assigned to each mode.

5.2.1.3.1 Calculations of Compulsory mode

The compulsory misses contribute 1-25% of the overall misses [1]. In a perfect ideal cache, all the misses are compulsory misses. In most caches the compulsory misses contribute a very little percentage of the overall misses. But in better caches, compulsory misses contribute a significant portion of the overall misses because very little can be done to improve these misses.
The compulsory misses have been initialized according to the method described below. It is based on some calculations, which have been done during the initialization i.e. before the start of the simulation.

First we have to decide about the percentage of the compulsory misses. It has been calculated by using the random function. The seed for this random function is calculated from the user supplied input seed. The random function is called to get some random value and then its modulo with some limit (which in this case is 12) is calculated:

$$\textit{Percentage of compulsory instruction misses} = \textit{(random value) MOD (12)}$$
$$\textit{Percentage of compulsory data misses} = \textit{(random value) MOD (12)}$$

As described earlier, compulsory misses contribute 1-25% of the overall misses. We have divided this percentage equally between instruction and data misses as 12. So the percentage of both instruction and data misses can range between 1 to 12 percent.

Proceeding this way, we can now calculate the number of compulsory instruction and data misses:

$$\textit{Compulsory instruction misses} = \textit{Percentage of compulsory instruction misses} \times \textit{Instruction misses} \qquad (5.13)$$

$$\textit{Compulsory data misses} = \textit{Percentage of compulsory data misses} \times \textit{Data misses} \qquad (5.14)$$

Generally, most of the compulsory misses occur at the beginning of the program execution. Because when the program starts, the instruction and data cache are both empty. It is reasonable to assume that most misses at the beginning of the program

execution are compulsory misses. So this special behavior can be introduced at the start of the simulation. We can calculate the approximate number of compulsory misses that can occur at the start of the program execution.

$$\textit{Compulsory instruction misses at startup} = \textit{Percentage of compulsory instruction misses at startup} \times \textit{Compulsory instruction misses} \quad (5.15)$$

$$\textit{Compulsory data misses at startup} = \textit{Percentage of compulsory data misses at startup} \times \textit{Compulsory data misses} \quad (5.16)$$

This percentage at startup is user defined. The default value is 0.4 or 40%. So if the user doesn't provide any value for that parameter it is taken as 0.4. In other words, we have assumed that 40% of the compulsory misses of both types occur at the beginning of the program execution. The reason for making this default percentage as 40% is that it may happen that by random calculations the percentage of compulsory misses come up to be in the range 20-25%. If we assume that all compulsory misses occur at the startup then nearly one fourth of the whole simulation would be dominated by one behavior, which is unlikely in most programs. Conversely a very low percentage of compulsory misses at startup fail to give any impression of compulsory misses. The percentage 40% seems to be most optimum.

The compulsory data misses at startup are divided into read and write misses at startup.

$$\textit{Write misses at startup} = \textit{Compulsory data misses at startup} \times \textit{Percentage of write misses} \quad (5.17)$$

$$\textit{Read misses at startup} = \textit{Compulsory data misses at startup} - \textit{Write misses at startup} \quad (5.18)$$

Now we have to calculate the number of Execution clock cycles during which the CPU model remains in this mode. This will be the execution cycles during which we need to introduce these compulsory instruction and data misses at startup. We proceed as follows:

We assumed that the miss rate during the given number of execution cycles in the compulsory mode is higher than the overall miss rate. It is because that at the start of the program execution the instruction and data cache are both empty so most accesses to the caches result in a miss. Or we can say that it is one of the regions of the whole program trace that contain more misses as compared to other regions. We have scaled the miss rate during this mode as 3 times higher than the overall miss rate, which is our default value.

$$\textit{Compulsory miss rate at startup} = 3 \times \textit{Instruction miss rate} \quad (5.19)$$

$$\textit{Compulsory miss rate} = \frac{\textit{Compulsory instruction misses at startup}}{\textit{No. of instructions in compulsory mode}}$$

$$\text{No. of instructions in compulsory mode} = \frac{\text{Compulsory instruction misses at startup}}{\text{Compulsory miss rate}} \quad (5.20)$$

$$\text{No of execution cycles in compulsory mode} \approx \text{No of instructions in compulsory mode} \times CPI$$

$$\text{No of execution cycles in compulsory mode} \approx \frac{\text{Compulsory instruction misses at startup} \times CPI}{\text{Compulsory miss rate}} \quad (5.21)$$

Now we know the total number of Execution clock cycles of this mode and the compulsory instruction and data misses which we have to introduce within these Execution clock cycles. The order in which these misses are introduced will be described in the next section when we discuss simulation part. The compulsory mode appears only once and it appears at the beginning of the simulation.

All the above calculation has been done before the start of the simulation.

5.2.1.3.2 Calculations of Random and Special modes

Now we have to calculate the same parameters for the Random and Special modes. The remaining number of Execution clock cycles and misses are distributed between these two modes according to the procedure, which we will discuss now.

The Random and Special modes are further divided into smaller parts. During the simulation, after the completion of Compulsory mode these smaller modes appear in random order till the end of the total number of Execution clock cycles. Recall that the Random mode represent the general behavior of the software on misses, which is not dominated by some particular software routine or the behavior which can't be modeled and the Special mode represent the behavior which is introduced as a result of some specific software routine. Therefore the simulation is a combination of different behaviors appearing in random order one after the other.

First we will calculate the total number of Random and Special modes. In other words, we can say that we will calculate the total number of parts in which the total Execution clock cycles are divided. Each part will represent some particular behavior.

Calculation of number of modes

The total number of modes has been divided on the basis of Execution clock cycles. The number of modes increases by 20 (by default) depending upon the range of Execution cycles. If it is within 50000 cycles the number of modes will be 20. If it is between 50000 to 100000 the number of modes become 40 and so on. This has been done so that the simulation would not be dominated by some particular mode for a long period unless specified by the user. So if we use the default value of this 'increasing factor', which is 20, in this case then each mode (except the compulsory mode) would consist 2500

execution clock cycles. However, if the user changes this value then the division would be different. For example, if the user enters the value of 'increasing factor' as 10 then each mode (except the compulsory mode) would consist of 5000 execution clock cycles. This 'increasing factor' is one of the optional inputs of the model. If the user doesn't provide any value for this, the default value of 20 would be taken.

After calculating the total number of modes, the individual number of Random and Special modes has been calculated. The percentages of these modes are also user configurable. User can use the optional input parameter "percentage of Random mode" to modify the number of modes of each type. It is possible to run the simulation with only Random mode or the Special mode or a combination of both. By default, the execution sequence is equally divided between these two modes i.e. 50% of the available Execution cycles are assigned to the Random mode and 50% to the Special mode. So by default, both Random and Special have equal number of modes and their quantity depends on the total Execution cycles. However, the distributions of misses are different which we will see shortly.

The Special mode introduce misses that are originated due to specific software routines like loops accessing the arrays, conditional statements etc. These routines do not remain active during whole execution of the program. They occur for some definite number of clock cycles and then vanish. Therefore the default percentage for both these major categories is set to 50% so that simulation is not dominated by some behaviors. However, some particular behavior can be introduced in greater number according to the will of the user.

Now we can calculate the individual number of Random and Special modes.

$$\textit{Number of Random modes} = \textit{Total number of modes} \times \textit{percentage of random mode} \tag{5.22}$$

$$\textit{Number of Special modes} = \textit{Total number of modes} - \textit{Number of Random modes} \tag{5.23}$$

The execution cycles for these two major modes can be calculated as:

$$\textit{Number of execution cycles of the Random mode} = (\textit{Total number of Execution cycles} - \textit{Execution cycles of compulsory mode}) \times \textit{percentage of Random mode} \tag{5.24}$$

$$\textit{Number of execution cycles of the Special mode} = \textit{Total number of execution cycles} - \textit{Execution cycles of compulsory mode} - \textit{Number of execution cycles of the Random mode} \tag{5.25}$$

The execution cycles are divided equally between each individual mode of the Random and Special mode.

$$\textit{Execution cycles in each individual mode of Random mode} = \frac{\textit{Number of execution cycles of the Random mode}}{\textit{Number of Random modes}} \quad (5.26)$$

$$\textit{Execution cycles in each individual mode of Special mode} = \frac{\textit{Number of execution cycles of the Special mode}}{\textit{Number of Special modes}} \quad (5.27)$$

We have calculated the total number of modes and then we have divided these modes into two categories the Random and the Special mode. We have divided the available Execution clock cycles between these two categories. Then we have calculated the individual number of modes of each category and then divide the execution clock cycles assigned to each category among their individual modes.

Now we will distribute the instruction and data misses between these two major categories and then further distribute them among their individual modes. These are the misses, which will be introduced during simulation within the execution clock cycles assigned to each mode.

Distribution of instruction and data misses

The distribution of misses can be divided into parts.

- When the distribution is done according to default values.
- When the distribution is done according to user-defined values.

In the first case, it means that the user hasn't provided relative percentages of Random and Special modes. Therefore simulation will be done according to default values and the default percentages of these two major modes are 50%. It implies that the Execution cycles are divided equally between Random and Special mode and both have equal number of modes. In that case, 70% of the data misses and 30% of the available instruction misses are assigned to the Special mode. This also means that the 30% of the data misses and 70% of the instruction misses are assigned to the "Random mode".

As described before, that the execution sequence is divided on the basis of what we can model (Compulsory & Special mode) and what we can't model (Random mode). Data misses are easier to model than instruction misses. Most data misses occurred when the program is in some kind of loop and it accesses different portions of data in each iteration. During the loops, same instructions are executed again and again but they access different data elements most of the time. During the execution of these routines, data misses are much higher than instruction misses. This is a behavior, which can be easily modeled. Most modes of the "Special mode" represent this kind of behavior. Because of these reasons, 70% of the data misses are assigned to the Special mode by default. On the other hand, it is hardly to find any behavior of the instruction misses due

to software except the conditional and unconditional jumps. One individual mode of the "Special mode" represents this behavior. Therefore 70% of the instruction misses is assigned to the Random mode by default.

Total data misses of the Special mode = (Total data misses – Compulsory data misses) × percentage of data misses assigned to the Special mode (5.28)

Total instruction misses of the Special mode = (Total instruction misses – Compulsory instruction misses) ×percentage of instruction misses assigned to the Special mode (5.29)

Total data misses of the Random mode = Total data misses – Compulsory data misses – Total data misses of the Special mode (5.30)

Total instruction misses of the Random mode = Total instruction misses – Compulsory instruction misses – Total instruction misses of the Special mode (5.31)

The above equations are used to calculate the misses when the default percentages of the Random and Special mode are used. In that case, the default percentages of instruction and data misses for each mode are used. If the user has changed the percentages of Random and Special mode, then the distribution of misses should also change accordingly i.e. the percentage of these misses allotted to each mode should increase or decrease in the same proportion as the increase or decrease of the corresponding mode. For example, the default percentage of data misses assigned to Special mode is 70%. This percentage is used when the default percentage of Special mode is used which is 50%. Now if the user changes this percentage from 50% to 70% then percentage of data and instruction misses assigned to Special mode should also increase from 70% and 30% respectively to some value in the same ratio. In the same way, the percentage of data and instruction misses of the Random mode should also decrease from 30% and 70% to some new value in the same ratio.

If the user-defined percentage of the Special mode is greater than the default percentage, then

Percentage of instruction misses of the Special mode =

Default percentage of the instruction misses + Scale factor × (user-defined percentage of Special mode – default percentage of Special mode) (5.32)

Where,

$$\textit{Scale factor} = \frac{\textit{1.0 – default percentage of the instruction misses}}{\textit{Default percentage of the Special mode}}$$

Percentage of data misses of the Special mode =
Default percentage of the data misses + Scale factor ×
(user-defined percentage of Special mode – default percentage of Special mode)

(5.33)

Where

$$\textit{Scale factor} = \frac{\textit{1.0 – default percentage of the data misses}}{\textit{Default percentage of the Special mode}}$$

If the user-defined percentage of the Special mode is less than the default percentage, then

Percentage of instruction misses of Special mode =
Default percentage of the instruction misses - Scale factor ×
(default percentage of Special mode – user-defined percentage of Special mode)

(5.34)

Where,

$$\textit{Scale factor} = \frac{\textit{default percentage of the instruction misses}}{\textit{Default percentage of Special mode}}$$

Percentage of data misses of Special mode =
Default percentage of the data misses – Scale factor ×
(default percentage of Special mode – user-defined percentage of Special mode)

(5.35)

Where,

$$\textit{Scale factor} = \frac{\textit{default percentage of the data misses}}{\textit{Default percentage of Special mode}}$$

After calculating the percentages of instruction and data misses according to the above formulas, equations 5.28,5.29,5.30 and 5.31 can be used to calculate the number of instruction and data misses. But this time, the percentages of instruction and data misses would be according to the formulas given above (5.32-5.35) and not the default percentages. The percentages for the Random mode can be found simply by subtracting the instruction and data misses assigned to the Special mode from the available total misses.

It should be noted that in the above formulas only the parameter ***"percentage of Special mode"*** is user defined. If the user changes this value, the percentage of the misses varies according to the scale factor. This scale factor depends on two values; the default percentage of the Special mode and the default percentage of the instruction and data

misses assigned to this mode. These parameters are declared as **`#define`** in the C code. Therefore if the user doesn't agree with this distribution and also has the access to the source code, the user can change them easily and all the calculations would be adjusted accordingly with the new default distribution.

We have distributed the instruction and data misses between Random and Special modes. Now we will distribute them among their individual modes.

Distribution of misses within individual modes

The number of instruction and data misses in each individual Random mode is calculated as:

$$\textit{Instruction misses in each Random mode} = \frac{\textit{Total number of instruction misses in the Random mode}}{\textit{Total number of Random modes}} \quad (5.36)$$

Similarly,

$$\textit{Data misses in each Random mode} = \frac{\textit{Total number of data misses in the Random mode}}{\textit{Total number of Random modes}} \quad (5.37)$$

$$\textit{Write misses in each Random mode} = \textit{Data misses in each Random mode} \times \textit{Percentage of write misses} \quad (5.38)$$

$$\textit{Read misses in each Random mode} = \textit{Data misses in each Random mode} - \textit{Write misses in each Random mode} \quad (5.39)$$

As evident, the instruction and data misses in each individual Random mode is the same. But during the simulation they don't look like the same because the decision about the miss is taken randomly in each cycle. Also the order in which these random modes appear is also based on randomness. We will explain this in detail in the later section.

We have already calculated the total number of misses for the Special mode. Now we have to distribute these misses within different types of Special mode. As told earlier, these modes are used to model the misses due to some special software routines. These are divided into the following categories according to their behavior:

- Routines that generate lot of data misses and few instruction misses.
- Routines that generate many instruction misses but few data misses.
- Routines that generate significant amount of both instruction and data misses.
- Routines that do not generate misses.

These following modes are used to represent these behaviors respectively:

- Loop Array data misses mode
- Instruction misses mode
- Zero misses mode
- Instruction data misses mode1
- Instruction data misses mode2

We will explain these modes in the later section. All these above mentioned modes are in Special category. In this section, we will restrict our discussion about how the misses have been distributed among them and the reasons behind these distributions.

The individual number of each of these modes can be found by multiplying its percentage with the total number of Special modes.

$$\textit{Number of modes} = \textit{Percentage of that mode} \times \textit{Total number of Special modes} \qquad (5.40)$$

This percentage in 5.40 is a user-defined perimeter i.e. the user can change the percentage of any individual mode of the Special mode. However, if the user doesn't make any changes default values will be used. The default percentage of each of these modes and the default percentages of the misses allotted to them are summarized in the table 5.1.

"Loop Array data misses mode" represent the behavior when the program is in a loop accessing different locations of the arrays, which result in misses. As most of the data misses are produced due to these pieces of code, most of the data misses are assigned to this mode as shown in table 5.1. Off course, not all loops which access arrays result in many misses but here we represent those loops, which access arrays in a manner that result in many data misses. At the same time, very few instruction misses are assigned to this mode because in a loop same instructions are repeated most of the times so it is very likely that they are found in the cache.

Mode	Default percentage of the mode Within Special mode	Default percentage of Instruction misses	Default percentage of data misses
Loop Array data misses mode	20	5	40
Instruction misses mode	20	40	5
Zero misses mode	10	0	0
Instruction data misses mode1	30	30	25
Instruction data misses mode2	20	25	30

Table 5.1

"Instruction misses mode" represent the behavior when the program jumps in a piece of code, which is not in a cache, and result into many instruction misses. And also the instruction misses, which occur because of many conditional jumps. Therefore it is assigned with many instruction misses and few data misses. It has assigned very few data misses because the mode represent the behavior of those conditional statements which operate differently on the same data elements based on which conditions become true. Although it seems that a huge percentage of instruction misses are assigned to this mode but actually it is not. The total percentage of instruction misses assigned to the Special mode by default is 30%. And out of these instruction misses 40% are assigned to the modes of this type.

"Zero misses mode" represents the behavior when the CPU is perfectly in execution and do not produce any misses. So no miss is assigned to this mode. This may happen when the CPU is performing some long arithmetic operations (multiplication/division) on some data that consumes many cycles and cache accesses result in hits.

"Instruction data misses mode1" and *"Instruction data misses mode2"* represent the behavior when we have both instruction and data misses in a significant number at the same time. These modes are basically the combinations of "Loop Array Data misses mode" and "Instruction misses mode" but introduce instruction and data misses in a different fashion during the simulation. Because of these reasons, a significant percentage of both types of misses are assigned to these modes.

Again the distribution of instruction and data misses among these different types of Special modes depends on whether the percentages of these modes are modified by the user or not. If they are not modified, then the default percentages of instruction and data misses will be taken and are calculated by multiplying these percentages with the total number of instruction and data misses of the Special mode.

Instruction misses of any individual mode of Special mode =
Number of instruction misses of the Special mode × Percentage of instruction misses of that mode *(5.41)*

Data misses of any individual mode of Special mode =
Number of Data misses of the Special mode × Percentage of data misses of that mode *(5.42)*

If the user changes the percentages of these modes, then percentages of the misses should also be scaled accordingly.

If the user-defined percentage of the mode is greater than the default percentage, then

Percentage of instruction misses of the mode =
Default percentage of the instruction misses + Scale factor ×
(user-defined percentage of the mode – default percentage of the mode)
(5.43)

Where,

$$Scale\ factor = \frac{1.0 - default\ percentage\ of\ the\ instruction\ misses}{Default\ percentage\ of\ the\ mode}$$

Percentage of data misses of the mode =
Default percentage of the data misses + Scale factor ×
(user-defined percentage of the mode – default percentage of the mode) *(5.44)*

Where,

$$Scale\ factor = \frac{1.0 - default\ percentage\ of\ the\ data\ misses}{Default\ percentage\ of\ the\ mode}$$

If the user-defined percentage of the mode is less than the default percentage, then

Percentage of instruction misses of the mode =
Default percentage of the instruction misses – Scale factor ×
(default percentage of the mode – user-defined percentage of the mode)
(5.45)

Where,

$$Scale\ factor = \frac{default\ percentage\ of\ the\ instruction\ misses}{Default\ percentage\ of\ the\ mode}$$

Percentage of data misses of the mode =
Default percentage of the data misses – Scale factor ×
(default percentage of the mode – percentage of the mode) *(5.46)*

Where,

$$Scale\ factor = \frac{default\ percentage\ of\ the\ data\ misses}{Default\ percentage\ of\ the\ mode}$$

Again, the parameters default percentage of the mode and the default percentage of instruction and data misses of each mode are declared as #define in the C code and can be changed if required.

After the above calculation, these misses are divided equally among the modes of the same type. For e.g., the instruction and data misses of the 'Loop Array data misses mode' can be calculated as:

Instruction misses in each 'Loop Array data misses mode' =

$$\frac{\text{Total instruction misses of the 'Loop Array data misses mode'}}{\text{Total number of 'Loop Array data misses mode'}} \quad (5.47)$$

Data misses in each 'Loop Array data misses mode' =

$$\frac{\text{Total data misses of the 'Loop Array data misses mode'}}{\text{Total number of 'Loop Array data misses mode'}} \quad (5.48)$$

Write misses in each 'Loop Array data misses mode' =
Data misses in each 'Loop Array data misses mode' × Percentage of write misses (5.49)

Read misses in each 'Loop Array data misses mode' =
(Data misses in each 'Loop Array data misses mode' – Write misses in each 'Loop Array data misses mode') (5.50)

5.2.1.3.3 Scheduling the simulation

From the above calculations, we have calculated the following parameters:

- Total Execution cycles
- Total number of instruction and data misses
- Total number of read and write misses
- Execution cycles for the Compulsory mode
- Total number of Instruction and data misses of the Compulsory mode
- Total number of read and write misses in the Compulsory mode
- Total number of modes
- Total number of Random modes
- Total number of Special modes
- Total number of instruction and data misses of the Random mode
- Total number of instruction and data misses of the Special mode
- Execution cycles in each individual mode of Random mode
- Instruction and data misses in each individual mode of Random mode
- Read and write misses in each individual mode of Random mode
- Total number of modes of each type of the Special mode
- Total number of Instruction and data misses of each type of Special mode
- Execution cycles in each individual mode of the Special mode
- Instruction and data misses in each individual mode of each type of the Special mode
- Read and write misses in each individual mode of each type of the Special mode

From these parameters we can schedule the simulation. For this purpose an array of structures is created which contain all the necessary data. This array is called schedule array in the code. *The length of the array is equal to the total number of modes.* The first location of the array is dedicated to the compulsory mode. The remaining locations are

filled based on randomness i.e. the order in which these modes occupy each location in the array is determined randomly. Each location of the array contains the following data:

Type of mode		
Total Execution cycles	Execution cycles count	Execution cycles left
Total instruction misses of the mode	Instruction misses count	Instruction misses left
Total Data misses of the mode	Data misses count	Data misses left
Total Write misses of the mode	Write misses count	Write misses left
Total Read misses of the mode	Read misses count	Read misses left

Figure 5.2

The 'type of mode' tells the ID of the mode. Each mode is assigned an ID. At the start of the clock cycle, the ID is checked to know which mode is currently active in the simulation and then the program jumps to the function, which represent the corresponding mode. For e.g., after accessing some location of the schedule array, the mode ID comes out to be 2. Then it means that 'Loop Array data misses mode' is the currently active mode, so the program jumps to the function representing 'Loop Array data misses mode'. The modes with their corresponding ID are given in table 5.2.

Mode Type	ID
Compulsory mode	0
Random mode	1
Loop Array data misses mode	2
Instruction misses mode	3
Zero misses mode	4
Instruction data misses mode1	5
Instruction data misses mode2	6

Table 5.2

Every location of the array is initialized as follows:

```
Type of mode = ID of the mode

Total Execution cycles   = Total Execution cycles in each mode of that
type
Execution cycles count  = 0
Execution cycles left   = Total Execution cycles

Total instruction misses of the mode  = Total instruction misses in
each mode of that type
Instruction misses count     = 0
Instruction misses left      = Total instruction misses of the mode
```

```
Total data misses of the mode = Total data misses in each mode of that
type
Data misses count  = 0
Data misses left   = Total data misses of the mode

Total Write misses of the mode  = Total Write misses in each mode of
that type
Write misses count = 0
Write misses left  = Total Write misses of the mode

Total Read misses of the mode  = Total Read misses in each mode of that
type
Read misses count = 0
Read misses left  = Total Read misses of the mode
```

During the simulation, at the end of every Execution clock cycle the 'Execution cycles count' increments by 1 and 'Execution cycles left' decrements by 1.When the 'Execution cycles count' becomes equal to 'Total Execution cycles' the mode is ended and from the next cycle the next location of the array is accessed which contains the same data for some other mode.

In the same way, 'Instruction misses count', 'data misses count' increment by 1 and 'Instruction misses left', 'data misses left' decrement by 1 after the introduction of instruction or data miss respectively. The parameter 'Execution cycles left' is compared with 'Instruction misses left' and 'data misses left' to ensure that enough Execution cycles are available to introduce the given number of misses. If the 'Execution cycles left' is found equal to either 'instruction misses left' or 'data misses left' then the corresponding misses have to be introduced in all the remaining Execution cycles of that mode to ensure that all the given misses have been introduced completely. Certainly, this is a deviation from the behavior which we are trying to introduce but we have to do it to ensure that all the misses allotted to the mode are introduced. Simulations have shown that this condition rarely became true and when it became true only one or two misses have been left.

When any mode is ended, the 'Execution cycles left', 'instruction misses left', 'data misses left', 'read misses left' and 'write misses left' become equal to zero and 'Execution cycles count', 'instruction misses count', 'data misses count',' read misses count',' write misses count' become equal to 'Total Execution cycles', 'Total instruction misses of the mode',' Total data misses of the mode', 'Total read misses of the mode' and 'Total write misses of the mode' respectively.

Example

Now we will illustrate all these steps through the following example. For the sake of simplicity lets assume that the user has provided only the basic inputs (no optional input), so the distribution would be done using the default values. The inputs are:

```
Total number of instructions     =  96,178
Percentage of load instructions  =  20%
```

```
Percentage of store instructions =  5%
Instruction miss rate            = 0.101
Data miss rate                   = 0.144
CPI (including the effect of structural, data and control hazards) =
1.1
Memory Access startup latency (Number of cycles required to initiate
memory transfer after detecting a miss)  = 1
Memory Accesses end latency (Number of cycles required to restore the
normal execution after the completion of memory transfer) = 1
Seed for the random functions  = 10
```

Now we will use all the above equations (5.1-5.50) to calculate the necessary data for the simulation. By using these equations we calculate the following parameters:

Execution clock cycles = CPI × Total number of instruction

⇒ 1.1 × 96178 = 105795

Percentage of load/store instructions = Percentage of load instructions + Percentage of store instructions

⇒ 20% + 5% = 25%

Total number of load/store instructions = Percentage of load/store instructions × Total number of instructions

⇒ 0.25 × 96178 = 24044

Total number of load instructions = Percentage of load instructions × Total number of instructions

⇒ 0.20 × 96178 = 19235

Total number of store instructions = Percentage of store instructions × Total number of instructions

⇒ 0.05 × 96178 = 4808

We can calculate the total number of instruction and data misses which are generated during the execution of the given number of instructions:

Instruction misses = Instruction miss rate × Total number of instructions

⇒ 0.101× 96178 = 9713

Data misses = Data miss rate × Total number of load/store instructions

⇒ 0.144 × 24044 = 3462

Write misses = Total number of store instructions × data miss rate

⇒ 4808 × 0.144 = 692

$$\textit{Percentage of write misses} = \frac{\textit{Write misses}}{\textit{Data misses}}$$

⇒ 692/3462 = 0.2000

Read misses = Data misses – Write misses
⇒ 3462 – 692 = 3370

Data cache Access rate = $\frac{\textit{Total number of load/store instructions}}{\textit{Total number of instructions}}$
⇒ 24044/96178 = 0.2499

Total misses = Instruction misses + Data misses
⇒ 9713 + 3462 = 13175

Now, we will calculate all the necessary data for the Compulsory mode:

Compulsory instruction misses = Percentage of compulsory instruction misses × Instruction misses
⇒ 0.06 × 9713 = 582

Compulsory data misses = Percentage of compulsory data misses × Data misses
⇒ 0.04 × 3462 = 138

Compulsory instruction misses at startup = Percentage of compulsory instruction misses at startup × Compulsory instruction misses
⇒ 0.40 × 582 = 232

Compulsory data misses at startup = Percentage of compulsory data misses at startup × Compulsory data misses
⇒ 0.40 × 138 = 55

Write misses at startup = Compulsory data misses at startup × Percentage of write misses
⇒ 55 × 0.2000 = 11

Read misses at startup = Compulsory data misses at startup - Write misses at startup
⇒ 55 – 11 = 44

Compulsory miss rate at startup = 3 × Instruction miss rate
⇒ 3 × 0.101 = 0.303

No of execution cycles in compulsory mode ≈ $\frac{\textit{Compulsory instruction misses at startup} \times \textit{CPI}}{\textit{Compulsory miss rate}}$
⇒ 842

Now, we will calculate the total number of Random and Special modes:

Assume that the 'increasing factor' of the modes is 14 per 50,000 execution clock cycles.

Total number of modes = 42 (Execution cycles lay in the range 100,000 to 150,000)

Number of Random modes = Total number of modes ×percentage of random mode
$\Rightarrow$ 42 × 0.5 = 21

Number of Special modes = Total number of modes – Number of Random modes
$\Rightarrow$ 42 – 21 = 21

The execution cycles for these two major modes can be calculated as:

Number of execution cycles of the Random mode =
(Total number of Execution cycles – Execution cycles of compulsory mode)
×percentage of Random mode

$\Rightarrow$ (105795 – 842) × 0.5 = 52476

Number of execution cycles of the Special mode =
Total number of execution cycles – Execution cycles of compulsory mode –
Number of execution cycles of the Random mode

$\Rightarrow$ 105795 – 842 – 52476 = 52477

The execution cycles are divided equally between each individual mode of the Random and Special mode.

Execution cycles in each individual mode of Random mode =

$$\frac{\text{Number of execution cycles of the Random mode}}{\text{Number of Random modes}}$$

$\Rightarrow$ 52476/21 = 2498

Execution cycles in each individual mode of Special mode =

$$\frac{\text{Number of execution cycles of the Special mode}}{\text{Number of Special modes}}$$

= 52477/21 = 2498

The total instruction and data misses for these two major modes can be calculated as:

Total data misses of the Special mode =
(Total data misses – Compulsory data misses) ×
percentage of data misses assigned to the Special mode

$\Rightarrow$ (3462 – 55) × 0.70 = 2384

Total instruction misses of the Special mode =
(Total instruction misses – Compulsory instruction misses) ×
percentage of instruction misses assigned to the Special mode

$\Rightarrow$ (9713 – 232) × 0.30 = 2844

Total data misses of the Random mode =
Total data misses – Compulsory data misses – Total data misses of the Special mode

$\Rightarrow$ 3462 – 55 – 2384 = 1023

Total instruction misses of the Random mode =
Total instruction misses – Compulsory instruction misses – Total instruction misses of the Special mode

$\Rightarrow$ 9713 – 232 – 2844 = 6637

Instruction misses in each Random mode =

$$\frac{\text{Total number of instruction misses in the Random mode}}{\text{Total number of Random modes}}$$

$\Rightarrow$ 6637/21 = 316

Similarly,
Data misses in each Random mode =

$$\frac{\text{Total number of data misses in the Random mode}}{\text{Total number of Random modes}}$$

$\Rightarrow$ 1023/21 = 48

Write misses in each Random mode =
Data misses in each Random mode × Percentage of write misses
$\Rightarrow$ 48 × 0.2000 = 9

Read misses in each Random mode =
Data misses in each Random mode – Write misses in each Random mode
$\Rightarrow$ 48 – 9 = 39

Now we will calculate the misses for the individual modes of Special mode:

The individual number of each mode can be found by multiplying its percentage with the total number of Special modes. Similarly, instruction and data misses can be found by using the equations 5.47-5.50. Following data is collected for each mode by using the values in Table 5.1.

'Loop Array data misses mode':

Number of 'Loop Array data misses mode' = 0.20 × 21 = 4
Instruction misses = 0.05 × 2844 = 142
Data misses = 0.40 × 2384 = 953
Instruction misses in each mode = 142/4 = 35
Data misses in each mode = 953/4 = 238
Write misses in each mode= 238 × 0.2000 = 47
Read misses in each mode= 238 – 27 = 191

'Instruction misses mode':

Number of 'Instruction misses mode'= 0.20 × 21 = 4
Instruction misses = 0.40 × 2844 = 1137
Data misses = 0.05 × 2384 = 119
Instruction misses in each mode = 1137/4 = 284
Data misses in each mode = 119/4 = 29
Write misses in each mode= 29 × 0.2000 = 5
Read misses in each mode = 29 – 5 = 24

'Zero misses mode':

Number of 'Zero misses mode'= 0.10 × 21 = 2
Instruction misses = 0
Data misses = 0
Instruction misses in each mode = 0
Data misses in each mode = 953/4 = 0
Write misses in each mode = 0
Read misses in each mode = 0

'Instruction data misses mode1':

Number of 'Instruction data misses mode1'= 0.30 × 21 = 6
Instruction misses = 0.30 × 2844 = 853
Data misses = 0.25 × 2384 = 596
Instruction misses in each mode = 853/6 = 142
Data misses in each mode = 596/6 = 99
Write misses in each mode = 99 × 0.2000 = 19
Read misses in each mode = 99 – 19 = 80

'Instruction data misses mode2':

```
Number of 'Instruction data misses mode2'= 21 – 4 – 4 – 2 – 6  =
5
Instruction misses = 0.25 × 2844 = 711
Data misses = 0.30 × 2384 = 715
Instruction misses in each mode = 711/5 = 142
Data misses in each mode = 715/5 = 143
Write misses in each mode = 143 × 0.2000 = 28
Read misses in each mode = 143 – 28 = 115
```

All the necessary calculated data for all the modes is summarized in the following table.

Mode	Total number of appearances	Total number of instruction misses	Total number of data misses	Instruction misses in each mode	Data misses in each mode	Execution clock cycles in each mode
Compulsory mode	1	232	55	232	55	842
Random mode	21	6637	1023	316	48	2498
Loop Array data misses mode	4	142	953	35	238	2498
Instruction misses mode	4	1137	119	284	29	2498
Zero misses mode	2	0	0	0	0	2498
Instruction data misses mode1	6	853	596	142	99	2498
Instruction data misses mode2	5	711	715	142	143	2498

Table 5.3

Now we have all the necessary data to schedule the simulation. We will place all this data in an array. The total number of modes is 43, so the length of the array will be 43. The first location is fixed for the compulsory mode and the remaining locations are filled randomly depending upon the value of the seed i.e. different value of seed fills the array in different order. During the simulation, the array is accessed sequentially from location 0 to 43. A possible sequence of the order through random calculation is shown in figure 5.3.

Location of the Array	Mode Occupied
0	Compulsory mode
1	Random mode
2	Random mode
3	Instruction misses mode
4	Instruction misses mode
5	Zero misses mode
6	Random mode
7	Random mode
8	Loop Array data misses mode
9	Instruction data misses mode1
10	Random mode
11	Instruction data misses mode2
12	Random mode
13	Random mode
14	Random mode
15	Instruction data misses mode1
16	Instruction data misses mode1
17	Random mode
18	Random mode
19	Random mode
20	Random mode
21	Instruction misses mode
22	Instruction data misses mode1
23	Loop Array data misses mode
24	Random mode
25	Random mode
26	Loop Array data misses mode
27	Instruction misses mode
28	Random mode
29	Instruction data misses mode1
30	Random mode
31	Instruction data misses mode2
32	Instruction data misses mode2
33	Random mode
34	Random mode
35	Instruction data misses mode1
36	Random mode
37	Instruction data misses mode2
38	Loop Array data misses mode
39	Random mode
40	Instruction data misses mode2
41	Random mode
42	Zero misses mode
43	Random mode

Figure 5.3 A possible order of execution of different modes

5.2.2 Simulation

This part starts when the user begins the simulation. All the necessary data that is required to carry out the simulation is calculated in the Initialization part. All this necessary data is stored in an array. During the simulation each location of the array is accessed one by one. Each location represent one mode and in that location all necessary information for that mode is stored, which tells about the total number of Execution clock cycles during which the simulation remains in this mode and also the total number of misses which have to be introduced in this mode. During each clock cycle, the function representing the mode has to decide whether a miss has to be introduced or not. These decisions have been made according to some rules. In this section we will describe the reasons behind these rules. The introduction of a miss is as simple as updating the variables that contain information about the misses and then calling the functions, which initiates communication with the memory. The important factor is the reasons behind these rules. Therefore in this section we will focus ourselves on the reasons, which enable us to set some rules for each mode.

Before starting this discussion, refer to figure 5.2 which shows the data elements of each location of the schedule array. The count variables 'instruction misses count', 'data misses count' increment after the introduction of their corresponding misses. Read misses count and write misses count variables increment depending upon the whether the corresponding data miss is a read miss or a write miss respectively. Before introducing any kind of miss, the corresponding count variables are checked and the miss can only be introduced if the value of these count variables is lesser than their maximum value which is stored in variables Total instruction misses of the mode, Total data misses of the mode, Total read misses of the mode and Total write misses of the mode. When this limit is reached, no further miss of that type is introduced during that mode.

5.2.2.1 Compulsory mode

When the simulation is started, the first location i.e. 0 location of the array is accessed which contains the data for the compulsory mode. Every simulation starts with the compulsory mode.

In this mode we have to introduce a behavior, which occurs at the beginning of the program. At the start of the program both instruction and data caches are empty so lot of misses occur at the beginning of the program. Most accesses to the memory are burst accesses. So when an instruction miss occurred an access to memory bring next 4 instructions in the cache. If there is no jump or branch instruction in these next 4 instructions then there will be no instruction miss in the next 4 cycles but if it is not, an instruction miss may occur. After 4 clock cycles an instruction miss will occur again. Many of these instructions are load/store instructions. Most load/store instructions at the beginning are initializing some data elements. So data misses occur frequently at the beginning. These misses are may be due to the initialization of an array or due to the initialization of some individual variables.

It is not possible to model all the above-mentioned behavior. We can only model the most general behavior that occurs at the beginning of the program. So we again follow our old rule i.e. model the most optimum behavior and the rest should be characterize by the use of Random functions.

Based on the above discussion, the simulation of the compulsory mode is divided into two parts each having equal number of execution clock cycles i.e. (Total execution cycles of the mode)/2 in each part. Instruction and data misses have also been divided equally between these two parts.

During the first part, the misses have been introduced in the following manner:

- Introduce one instruction miss after every 4 clock cycles.
- Introduce one data miss after every 7 clock cycles.

During the second part we use our random calculation method to decide about the misses. This is explained in the description of the Random model. However the comparison has been made with the miss rate of the compulsory mode, which is three times higher than the overall actual miss rate. So the misses generated during this mode through random calculations is higher than those produced in the Random mode. (Read this paragraph again after reading the next topic 'Random mode').

5.2.2.2 Random mode

In this mode, instruction and data misses are produced by using random function. It represents the behavior, which is not easier to model.

The question arises, are the origination of misses are random? The answer is yes if we look at all the misses that produce during the whole execution of a program. Their generation does not follow any rule and can be originated in any cycle. This is because there are many factors (cache configuration, software) that decide about a miss in a given cycle and it is difficult to say which limitation becomes the cause of a miss. However when we start observing the misses of a program in a shorter duration, they don't look like originating in a random fashion at some places. This may be due to some particular piece of code. We have tried to model these behaviors in the Special mode. However, the basic programming constructs can appear in the programs in an infinite number of combinations and it is not possible to model all of them. Therefore the misses that arises because of that pieces of code or where they don't follow any pattern can be considered as originating in a random fashion. There is no way to model these misses except by the use of random calculations.

We start with the observation that miss rate can also be taken as a probability that in a given cycle the cache access can result in a miss. Recall that the probability of any event is equal to the total number of occurrences of that event divided by the total number of all probable events.

$$\textit{Probability of a miss in a given cycle} = \frac{\textit{Number of accesses that miss}}{\textit{Total number of accesses}}$$

During the simulation, when the random mode is in execution, undergoing the following steps makes the decision about a miss in a given cycle.

The decision about the instruction miss is taken as follows:

1. Call the C random function (rand_r()) to get some random value.
2. Divide this random value with the maximum random value (termed as RAND_MAX). It means that after division we will get a value that ranges from 0 to 1.
3. Compare this value with the instruction miss rate, which is also the probability of an instruction miss in a given cycle.
4. If it is less than the instruction miss rate then it will be taken as a miss otherwise a hit.

These steps are illustrated with the following example.

Suppose the miss rate is 0.21 means the probability of a miss in a given cycle is 0.21. Then by following the above procedure, if we get some value less than 0.21 say 0.16 then it can be taken as a miss because it lies in the shaded region (as shown in figure 5.4). On the other hand, if it is greater than 0.21 then it can be taken as a hit. This shaded region can also be taken as the PDF (probability density function) of misses. So all the returned values that fall in the PDF would be taken as misses.
The decision about the data miss is taken as follows:

1. Call the C random function (rand_r()) to get some random value.
2. Divide this random value with the maximum random value (termed as RAND_MAX). It means that after division we will get a value that ranges from 0 to 1.
3. Compare this value with the data cache access rate, which is the probability that the given instruction is either load or store.
4. If it is less than the data cache access rate then it can be taken as a load/store instruction otherwise not. If it is not a load/store instruction then this process ends here, otherwise it continues to the next step.
5. Repeat the first two steps again.
6. Compare this value with the data miss rate, which is also the probability of a data miss in a given cycle.
7. If it is less than the data miss rate then it can be taken as a miss otherwise a hit.
8. If it is a miss then repeat the first two steps again.
9. Compare this value with the percentage of write misses, which is the probability of write miss in total number of data misses.
10. If it is less than the percentage of write misses then it will be taken as a write miss otherwise a read miss.

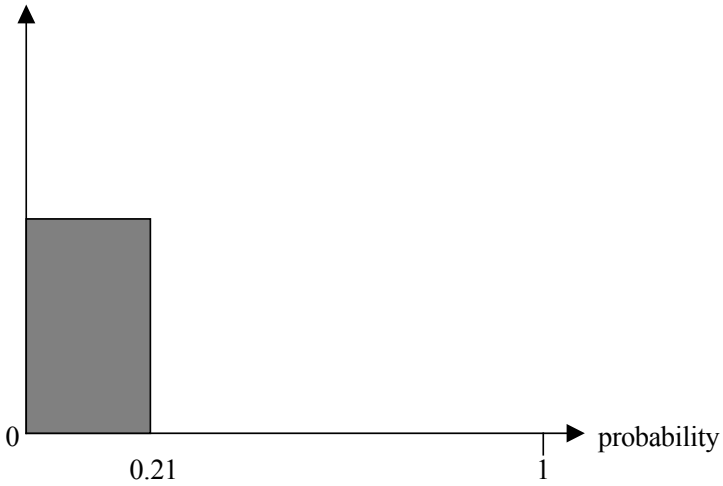


Figure 5.4 PDF of misses

By using the random model, it is found that when the miss rates are low, then most of the resulting misses are spaced apart by many clock cycles (more than 50) and when the miss rates are normal or high most misses are close to each other i.e. the time difference between two successive misses is normally less than 10-15 clock cycles most of the times. This is the same kind of behavior, which we can find in actual programs execution. When the miss rates are low which implies that a cache is big and after each memory access a large amount of data comes in the cache. There would also be less conflict and capacity misses and so it is very likely that the number of clock cycles between two successive memory accesses are large. In the same way, a high miss rate means a small cache or a bad software that result in many misses which are close to each other in terms of number of clock cycles among them. There are many behaviors where random mode doesn't provide enough approximation and we have tried to model these behaviors in Special mode.

5.2.2.3 Loop Array Data misses mode during simulation

In this mode we model the misses that are originated when the program is in a loop accessing the arrays. As shown in table 5.1, this mode has been assigned the highest percentage of data misses. Therefore lots of misses are introduced when this mode is active during the simulation. In most programs, most data misses are produced when the program is accessing arrays in the loops. It doesn't mean that all loops generate lot of misses. Our aim in this mode is to target those loops, which generate many data misses. The loops that do not generate many misses can be approximated by the Random mode. To demonstrate our model, we begin with the examples of few loops that generate many misses. Consider the example of following nested loop, which accesses two two-dimensional arrays.

```
for ( i = 0; i<3;i++)
      for (j=0;j<100;j++)
            a[i][j] = b[j][0]+b[j+1][0];
```

The number of misses produced during the whole execution of this loop depends on the size of the cache, block size and the degree of associativity, but as discussed earlier, we can't make use of these parameters. If we consider the body of the loop, we can observe that in each iteration of the inner loop for j, at least two load instructions are required to access b[j][0] and b[j+1][0] and one store instruction to store the result of the addition in a[i][j]. If we consider the worst case, there will be three data misses in each iteration of the inner loop. The array a can be benefited from spatial locality but the array b couldn't since the accesses to the array b are not in order in which they are stored in the memory. However, array b can be benefited from temporal locality since the same elements of the array b are accessed in each iteration of the loop. Remember that nearly in all microprocessors used today, all read accesses to the memory are burst accesses. This means each read access to the memory brings the next four data items or instructions in the cache. Therefore if we ignore the conflict misses for a moment then during the first complete execution of the inner loop when i=0, there will two data misses due to array b and one data miss for array a after every fourth iteration. During i=1 and i=2 there would be one data miss due to array a after every fourth iteration and there would be few data misses due to array b because most of the elements would already be in the data cache. So the first complete execution of the inner loop i=0 is of particular interest for us. There may be few instruction misses at i=0,j=0 during the very first iteration but after that there would be no instruction miss unless the instruction cache is very small. This is because same instructions are executed again and again. Each iteration would approximately consists of 8-10 assembly instructions (2 load, 1 store, 4 add, 1 branch) and require 8-10 clock cycles plus clock cycles consumed to access the memory in case of a miss, to complete each iteration in a pipelined machine.

Now consider another loop,

```
for (j=0;j<100;j++)
      for (i=0;i<100;i++)
            x[i][j]=x[i][j]+s;
```

The above loop would generate two data misses (one read and one write) in each iteration of the inner loop since the accesses to the memory are not in order in which they are stored. Off course, the above loop couldn't be written by a good programmer but our aim here is to study only the effect of different codes on the misses and not to improve them. In MIPS assembly, the inner loop may look like as follows:

```
Loop:  LD F0,0(R1);         F0=array element
       ADDD F4,F0,F2;       add scalar in F2
       SD 0(R1),F4;         store result
       SUBI R1,R1,#800;     decrement pointer
       BNE R1,Loop;         branch R1!=zero
```

The above loop requires at least five clock cycles to complete each iteration (ignoring data hazards) and most probably would generate two data misses in each iteration. The improved version of this loop is as follows:

```
for (i=0;i<100;i++)
    for (j=0;j<100;j++)
        x[i][j]=x[i][j]+s;
```

Most probably the above loop would generate data misses after every fourth iteration of the inner loop since the memory accesses are in order and loop would benefit from spatial locality. This means that we will see some misses after 20-25 clock cycles. Definitely, modeling such a loop is not our target in this mode. Random mode can represent such kinds of loops.

As evident from the above discussion, our focus is on the loops that generate lots of data misses. The above two examples are few among them. We couldn't just take these few examples and study them thoroughly and include their behavior exactly in our model. There can be many loops that have nearly the same behavior (number of misses in each iteration and clock cycles required to complete the iterations) with some variations. But there can be many loops of these types with different behaviors. It is impossible to model all of them. We can only generalize them and by generalizing we can say that when these loops (which generate many data misses) are in execution there are at least two data misses within every 10 clock cycles.

There are very few instruction misses that occur in a loop because same instructions are executed again and again. Most of these instruction misses occur during the first and the second iteration of the loop. The main reasons for this are that during the first iteration the instructions are not in the loop and the branches are also miss predicted.

The number of Execution clock cycles, instruction and data misses for this mode are already calculated in the Initialization part. So when this mode appears in the simulation two data misses (one read and one write) are introduced after every 8 clock cycles. This will continue until the data misses count reaches to the values of the total misses assigned to this mode. Some instruction misses are also introduced at the beginning of the mode. One instruction miss is introduced after every 4 clock cycles during the first 20 clock cycles. After that only the data misses are introduced.

5.2.2.4 Instruction misses mode

In this mode we represent the behavior when instruction misses occur in quick succession. In terms of software, the most obvious reasons for these misses are conditional and unconditional jumps. The conditional jumps occur because of ‘if and if else’ statements and the test conditions of the loops. The unconditional jumps occur when some function is called and when the function is returned. In this mode, we have focused on the conditional statements that occur due to ‘if and if else’ statements or switch statements.

If the program is in a part of a code, which is run for the first time, or which is not in the cache, an instruction miss is generated after every 4 clock cycles of sequential execution. Each access to the memory brings the next 4 sequential statements. If there are branches or jumps within these 4 instructions then an instruction miss can occur before 4 clock cycles (if the branch target instructions are also not in the cache). Consider the following example:

```
if (a = =1)
      {
            if (b= =1){
            do some thing on variables a and b;}
            else
            do some thing;
      }
else if (a = = 2)
      {
            if (b= =1){
            do some thing on variables a and b;}
            else
            do some thing;
      }
```

and so on.

Imagine if this code is run for the first time or if this code is not in the cache. Suppose there are lot of conditional if else statements in the above code, which become true depending on the value of variable a. Consider the worst case, when there are even nested if statements under each condition. In that case there will be lot conditional jumps and the code will not execute sequentially. Assuming the code is not in the cache, this particular software behavior result in instruction misses and the clock cycles between two consecutive instruction misses would be less than 4 clock cycles. Data misses may occur but in this particular scenario the data misses would be less than instruction misses because these statements operate on the same data elements but in a different manner.

In this mode, the codes that resemble with the above example are the targets. The aim is to generate some instruction misses in a manner that the difference between two consecutive instruction misses is sometimes less than 4 clock cycles and sometimes greater than 4 clock cycles. The generation of these instruction misses should also be coupled with some data misses but with a lesser frequency than the instruction misses.

Based on these requirements, when this mode is active during the simulation, an instruction miss is introduced after every 5^{th}, 7^{th} and 9^{th} clock cycles and a data miss is introduced after every 20 clock cycles until they reach limits assigned to the mode.

As described earlier, at first glance it seems that this mode introduces unreasonably large number of instruction misses in quick succession. But actually, it is not. Only 30% of the instruction misses is assigned to all the modes of Special type. Out of this 30%, 40% of the instruction misses is assigned to this mode, which is further subdivided according to the total number of 'instruction misses modes'. When this mode is active in simulation, the maximum limit of instruction misses usually reaches very early (usually one fourth of the execution clock cycles).

5.2.2.5 Zero misses mode

As evident from the name this mode doesn't introduce any kind of misses. The basic purpose of this mode is to produce a gap between regions that have lots of misses. In real programs, generally misses occur in quick succession and then there are no misses for a quite a number of clock cycles. This mode basically represents these clock cycles when the CPU is perfectly in execution. When this mode is active in simulation it counts the number of Execution clock cycles that are executed in this mode. When this count reaches to the limit assigned to it, it ends.

5.2.2.6 Instruction data misses mode1

The 'Loop Array data misses mode' and 'Instruction misses mode' represent the extreme behavior i.e. when a burst of data misses and burst of instruction misses occur. In many pieces of code both instruction and data misses occur frequently. The 'Instruction data misses mode1' and the 'Instruction data misses mode2' generate both instruction and data misses in significant number. Also the misses generated in these modes are not representing some particular software routines all the time. They also represent some combination of misses, which are difficult to visualize in terms of software. They model the behavior when both instruction and data misses occur in quick succession and when both instruction and data miss occur at the same time i.e. in the same clock cycle.

During the simulation, this mode is further divided into three parts. In the first part it represent the approximate behavior of the misses that occur in a sequential code, which is not in the cache. Therefore it introduces one instruction miss after every 4 clock cycles. Data misses can also occur in a straight sequential (not in the cache) but usually less than instruction misses because all instructions are not load/store and all load/store instructions doesn't result into a miss. Therefore one data miss is introduced after every 10 clock cycles. This data miss toggles as a read and write miss, every time it is introduced. This behavior continues until half of the instruction misses allotted to this mode are introduced. After that it switches to another behavior.

In the second part, it represents the behavior when data misses are produced in greater number than the instruction misses. This may be due to a loop but not of that types which are modeled in 'Loop Array data misses mode'. In this part, we assume loops that have a big body consisting of many instructions and each iteration requires many clock cycles. Therefore instruction misses may occur in every iteration but to a lesser extent than the data misses. Therefore in this part, one data miss is introduced after every 6 clock cycles and one instruction miss is introduced after every 10 clock cycles. Again the introduction of data misses is divided into read and write misses. One write miss is introduced after every two read misses. This behavior continues until ¾ of the data misses allotted to this mode are introduced.

At the end it represents the behavior when an instruction and data miss occurs very closely. In a pipeline processor, the instruction cache is accessed in every clock cycle and when some load/store instruction reaches in a pipeline stage where it has to access the data cache, the instruction and data caches are both simultaneously accessed in that particular clock cycle. It may happen that both of these accesses result in a miss. In that case, data miss is handled first followed by the handling of instruction miss. This causes the processor to stall for a larger number of clock cycles. In the outside world, it may mean that it will occupy the bus for a larger number of clock cycles.

So in the third part of this mode, this behavior is introduced after every 6 clock cycles. One instruction and one data miss is introduced in the same cycle after every 6 clock cycles. The data miss is handled first and then instruction miss is handled. This third part is very small (it should be small) and it is found by simulation that it normally occurs less than 10 times. It is because; instruction misses allotted to this mode is already small. And nearly 90% of the instruction misses are introduced in the first two parts of the mode. Remember that an instruction or data miss can only be introduced if the current values of their respective counters are lesser than the maximum value of the misses allotted to the mode. Therefore in the third part, after few simultaneous instruction and data misses, instruction misses count reaches the maximum value and there will be no more instruction misses. However, data misses are continuously introduced until they also reach their limit.

5.2.2.7 Instruction data misses mode2

The behavior represented by this mode resemble to the behavior introduced in 'Instruction misses mode'. In 'Instruction misses mode' the instruction misses are introduced in greater number than data misses. In that mode we have represented the behavior when the time difference between two successive instruction misses is less than 4 clock cycles. This effect is due to the conditional statements. In that mode, we have focused on the conditional statements, which operate on the same data elements resulting in lesser number of data misses.

It may be a case that these conditional statements operate on different data elements, which are also not in the cache and hence result in data misses. Also the generation of instruction misses in 'Instruction misses mode' represent the extreme behavior i.e. lot of

instruction misses are introduced in that mode. It may be a case that the generation of instruction misses due to these conditional statements is not that much high as represented in 'Instruction misses mode'.

In this mode, we have represented the similar behavior of conditional statements with the difference that they operate on many data elements and these statements result in lesser number of instruction misses. Therefore it generates both instruction and data misses in significant proportions.

Therefore when this mode is active in the simulation, two instruction and two data misses are introduced in every 15 clock cycles. The two instruction misses are introduced consecutively one after the other. The data misses are also introduced one after the other. One of the data miss is read miss and the other is write miss.

5.3 An Overview of sequence of Operations

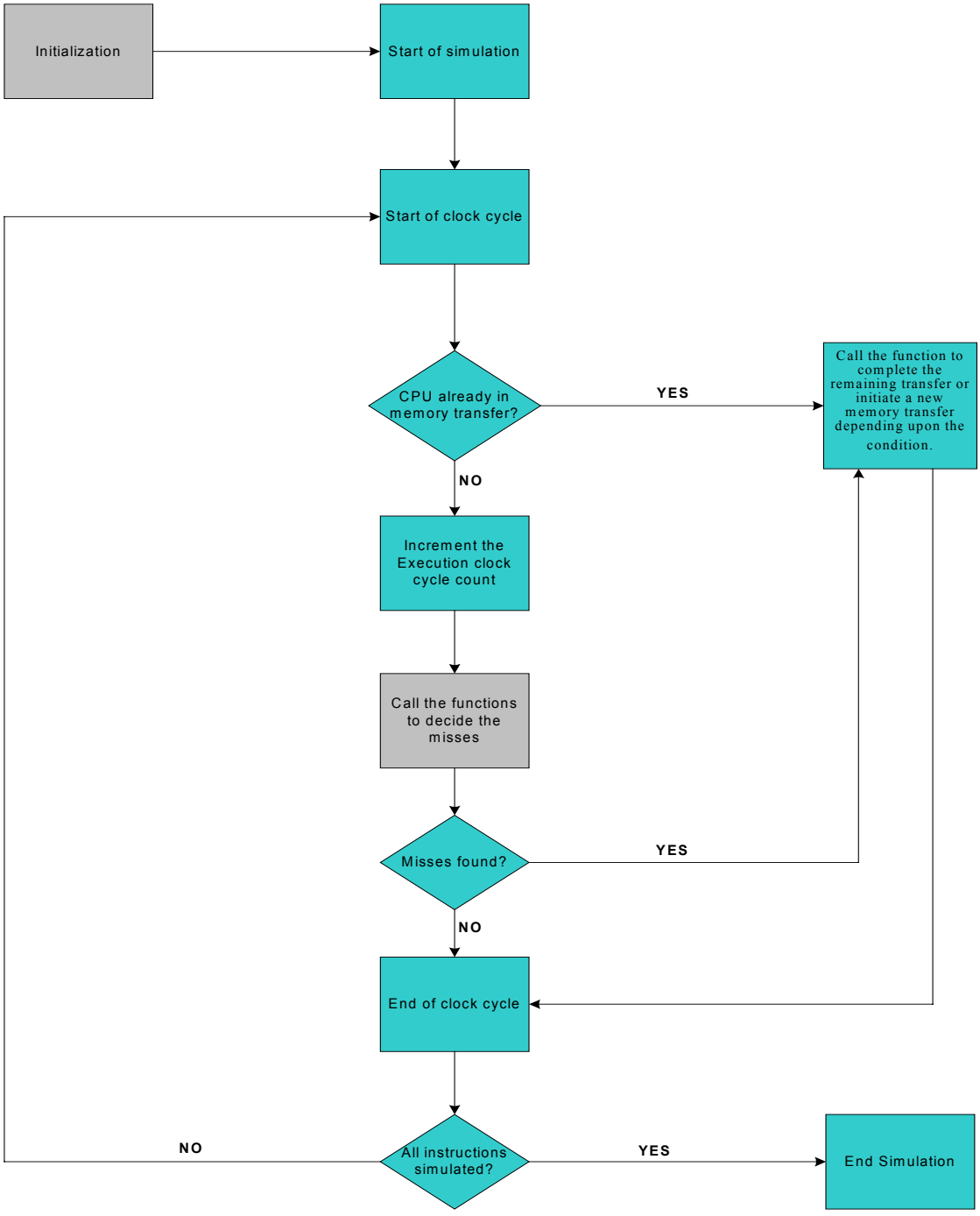
In this section we will describe the sequence in which different steps are carried out. The whole simulation sequence is described in section 4.2.4 of chapter 4. For ease the figure 4.2 is reproduced here as figure 5.5. In section 4.2.4 all the steps have been explained except the two grey shaded blocks labeled as 'Initialization' and 'Call the functions to decide the misses'. The first block 'Initialization' is basically the Initialization part discussed in section 5.2.1 and the second block 'Call the functions to decide the misses' is the simulation part discussed in section 5.2.2. These two blocks in figure 5.5 can be replaced by figure 5.6 and fig 5.7. Now we will describe the whole execution sequence again that also include the detailed steps which were missing in section 4.2.4.We start with the 'Initialization' part which takes place before the start of the simulation. All these steps are shown in figure 5.6

1. Read the input parameters from the file, which include both the basic and optional input parameters.
2. Calculate the total number of Execution clock cycles, total number of instruction and data misses (read and write misses) for the whole simulation.
3. Calculate the Execution clock cycles, instruction and data misses for the Compulsory mode.
4. Calculate the total number of modes from the Execution clock cycles by using the procedure described in section 5.2.1. Calculate the individual number of Random and Special modes and divide the remaining Execution clock cycles among these modes according to their percentages.
5. Check whether the user has changed the default percentages of the Random and Special modes. If the user hasn't changed them, then use the default percentages of instruction and data misses to calculate the total number of instruction and data misses assigned to each mode. On the other hand, if the user has changed the default percentages of Random and Special modes, then first scale the percentages of instruction and data misses according to the relative percentages of Random

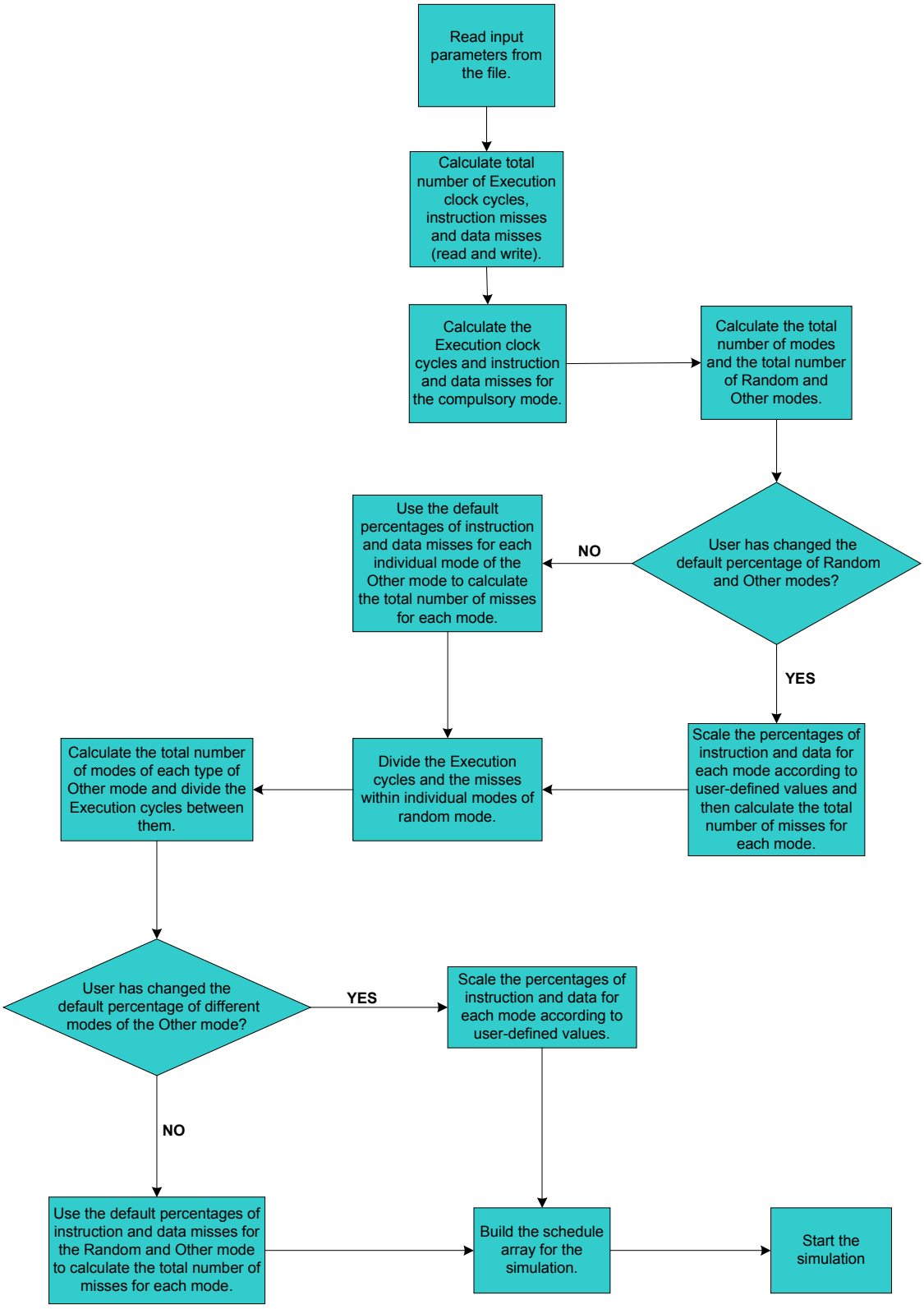
and Special modes and then calculate the total number of instruction and data misses for each mode.

6. Divide the total Execution clock cycles, instruction and data misses which are assigned to the Random mode, equally among individual small modes of Random mode.

Figure 5.5



Flowchart of the complete process



Flowchart of the initialization part

Figure 5.6

7. Divide the total Execution clock cycles assigned to the Special mode among individual modes of different types of the Special mode.

8. Check whether the user has changed the default percentages of the different types of Special modes. If the user hasn't changed them, then use the default percentages of instruction and data misses to calculate the total number of instruction and data misses assigned to each mode. On the other hand, if the user has changed the default percentages of these modes, then first scale the percentages of instruction and data misses according to their relative percentages and then calculate the total number of instruction and data misses for each mode.
9. Build the schedule array that contains all the necessary information to carry out the simulation. Each location of the array represents one individual mode. The length of the array is equal to the total number of modes.

This completes the 'initialization' part. All the remaining steps are performed during the simulation. In each clock cycle, the following steps are carried out until the end of the simulation. These steps are shown in figure 5.7.

10. Check at the start of each cycle if the CPU is in some stage of memory transfer. This can be seen by checking the variable, which shows whether CPU is stalled, or not. If the CPU is stalled then it is already in the communication phase i.e. communicating with the memory through OCP, which requires some cycles to complete. In this case call the function bus_interface_process(), which is responsible for the memory transfer. The function also checks the current status of the handshaking and takes necessary actions accordingly. This step is repeated in every clock cycle until the memory transfer is completed. If the CPU is not currently involved in memory transfer or the memory transfer is completed then go to the next step.
11. Increment the 'Execution clock cycles count' variable, which is the counter for the total Execution clock cycles for the whole simulation including all the modes.
12. Access the schedule array using schedule_count variable to know which mode is currently active. This can be found by checking the mode ID. Jump to the function, which represent this mode.
13. Increment the 'Execution clock cycles count' variable (this variable is different from the variable used in step 11 and is different for each mode) for that mode and decrement the 'Execution clock cycles left' variable for that mode.
14. Decide whether an instruction miss or data miss or both have to be introduced in this cycle or not. The decision procedure is different depending upon which mode is active in a given cycle.
15. Based on the decision procedure, check whether any miss or misses have been found or not.
16. If a miss or misses have been found then set the variables to 1 that contain information about the misses. Increment the 'instruction misses count' or 'data misses count' variables depending upon whether instruction miss or data miss have been found. Also decrement the 'instruction misses left' or 'data misses left' variables depending upon the type of the miss.

17. If no miss has been found then check whether enough Execution cycles are available to introduce the given number of instruction and data misses for that mode. This has been done by comparing the 'Execution cycles left' variable with the 'instruction misses left' and 'data misses left' variables.
18. If the 'Execution cycles left' variable becomes equal to either 'instruction misses left' or 'data misses left'(enough cycles are not available) then go to step 16 , otherwise (enough cycles are available) go to the next step.
19. Compare the 'Execution clock cycle count' variable with the 'Execution clock cycles' variable for that mode. If the 'Execution clock cycle count' variable becomes equal or greater than the 'Execution clock cycles' for the mode then increment the 'schedule_count' variable so that from the next cycle the next location of the schedule array will be accessed which would contain some other mode.

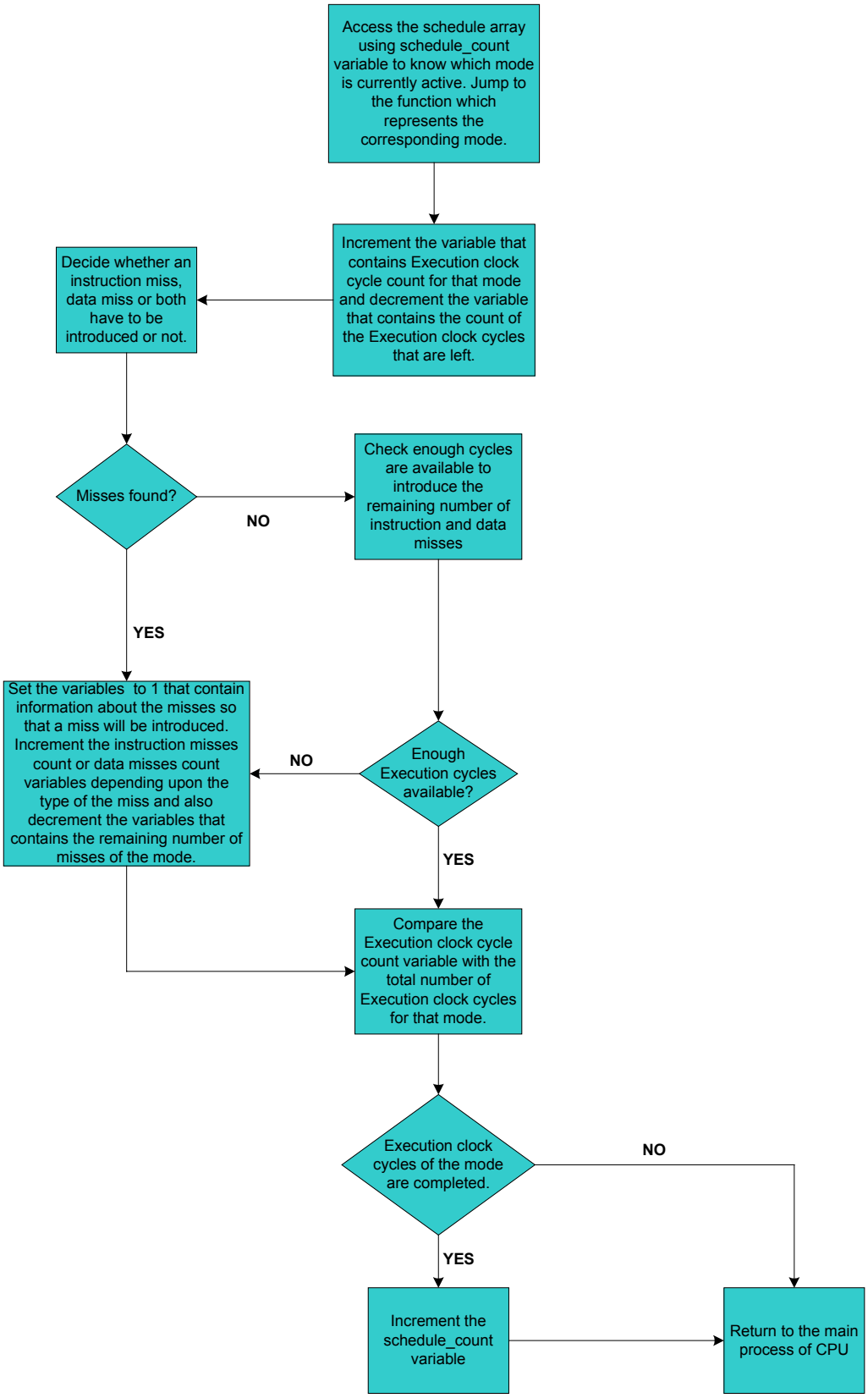
This completes the steps which are used to take decision about the introduction of a miss in a given cycle during simulation.

20. Check the variables that contain information about the instruction and data misses. If there is no miss then go to step 21 otherwise go to the step 23.

21. The clock cycle has ended. Save the values of all the variables that will be used in the next cycle. All necessary variables are declared globally so that their values will not be lost at the end of the function.

22. Compare the value of the 'Execution clock cycle count variable' with the Total execution clock cycles value. If it is less than the total value then go back to step 10. If it is equal or greater than the total value then it means that all the instructions have been simulated so go to the step 24.
23. If a miss has been found then stall the processor model. Call the function that starts communication with the memory through OCP. Keep the processor stalled until the miss has been handled. This has been checked at the start of every cycle as shown in step 10. If there is a miss in both the instruction and data then handle the data miss first and then handle the instruction miss. Clear the variables that contain information about the misses that needs to be handled i.e. sets them to 0. Go to step 21.

24. All the instructions have been simulated. Generate report about the total execution time of the instructions, the number of cycles in which the CPU remain stalled waiting for the memory response, the number of cycles the CPU is executing instructions, shortest and longest memory access, effective CPI etc. Terminate the simulation.



Flowchart of the simulation part

Figure 5.7

5.4 SUMMARY

In this chapter we have described the methods which we have used to generate misses. We have started with the brief overview of the caches. Then we have described the method used to generate misses which consists of two parts the Initialization (that takes place before the start of the simulation) and the Simulation (that takes place during the simulation). We have described these two parts in detail in section 5.2.In the end, we have described the complete sequence of the whole process. The overall objective of all of these steps is to generate the misses in a manner that they look like the misses of a real program.

From the input parameters, the model calculate the total number of misses and the Execution clock cycles in which it has to generate these misses. To introduce these misses in a realistic manner, it does some calculations on the available data before the start of the simulation. It divides the total Execution clock cycles into three parts. These parts represent different modes of behaviors of program execution and are called Compulsory mode, Random mode and Special mode. It then divides these available number of instruction and data misses among these three modes. The distribution of misses has been done according to some rules. These rules have been made based on the study of different behaviors of software routines and their affects on the misses. The Execution clock cycles assigned to the Random and Special mode are further divided into small parts or modes. The total number of these parts or modes increases with the increase of total number of Execution cycles. The instruction and data misses assigned to Random and Special mode are then further distributed within these smaller parts of each category. This distribution is also done with some rules which have been made with some theoretical background. In all the steps of the distributions, the user has the ability to change the rules of distribution of misses. At the end a number of different modes are created. Each mode represent some software behavior. Each has assigned some Execution clock cycles and the misses which have to be introduced within these clock cycles. The total number of Execution clock cycles and the total number of misses are thus distributed among these small parts or modes. All this information is then placed in an array called schedule array. Each mode occupy one location. The length of the array is equal to the total number of modes. When the simulation is started this array is accessed in order i.e. from the first location to the last one. These modes starts appearing in the simulation generating misses according to the procedure representing the mode. Each mode stays for a fixed number of clock cycles assigned to it , introduces its misses and then vanishes. In this way all the locations of the arrays are accessed. When the mode in the last location of the array completes its clock cycles, the simulation is ended. At that time, the total execution cycles have completed and all the required number of misses have also been introduced as well. The model then generate the report about the effective CPI which include the effect of all the misses.

REFERENCES

[1] Computer Architecture A Quantitative Approach by John L Hennessy & David A Patterson.

[2] Computer Organization and Design by John L Hennessy & David A Patterson.

CHAPTER

6

Testing the CPU Model

In the first five chapters we have described the complete functionality of the model. The last task is to test its accuracy. In this chapter we will discuss all the issues related to it.

The first question in this regard is: Can we really test it? If yes then how can we test it? This is not a conventional software or hardware design. Normally to test any software/hardware we apply different combinations of inputs and then compare the output of the system with the desired output. In that cases, we know the desired output but in our case we don't completely know the desired output.

This chapter starts with the discussion about different approaches used for testing. We will describe different steps, which we have taken to do this testing process. This discussion is coupled with the description of different problems, which encountered during this process and the tasks, which haven't been completed because of these problems.

6.1 Different Approaches for testing

First we have to see in which patterns misses originate in most programs. In other words, how the generation of misses looks like when viewed with respect to the time i.e. during the complete execution. Figure 6.1 shows some general traces of the programs, in which the generation of misses are plotted with the elapsed time. This is the general behavior, which we can find in most programs. The dark grey regions show the instants during program execution when significant number of misses are produced. The light grey regions represent the instants when misses are produced but not in very large number. Finally, the white regions show where very few misses are produced or when most accesses are hits. A deviation from this behavior means that the whole trace is occupied by same region, which can be either dark grey, light grey or white. Many programs deviate from the general behavior shown in figure 6.1 but we are interested in the most general behavior which we can find in most programs i.e. according to figure 6.1.

Therefore in the complete execution trace of most programs, at some points the concentration of misses is high and at some points the concentration is low. Generally, misses do not originate in regular order. Due to some special piece of code, large number of misses occur in quick succession. When the program leaves this part of code and enters some other part, which is in the cache, the generation of misses decreases.

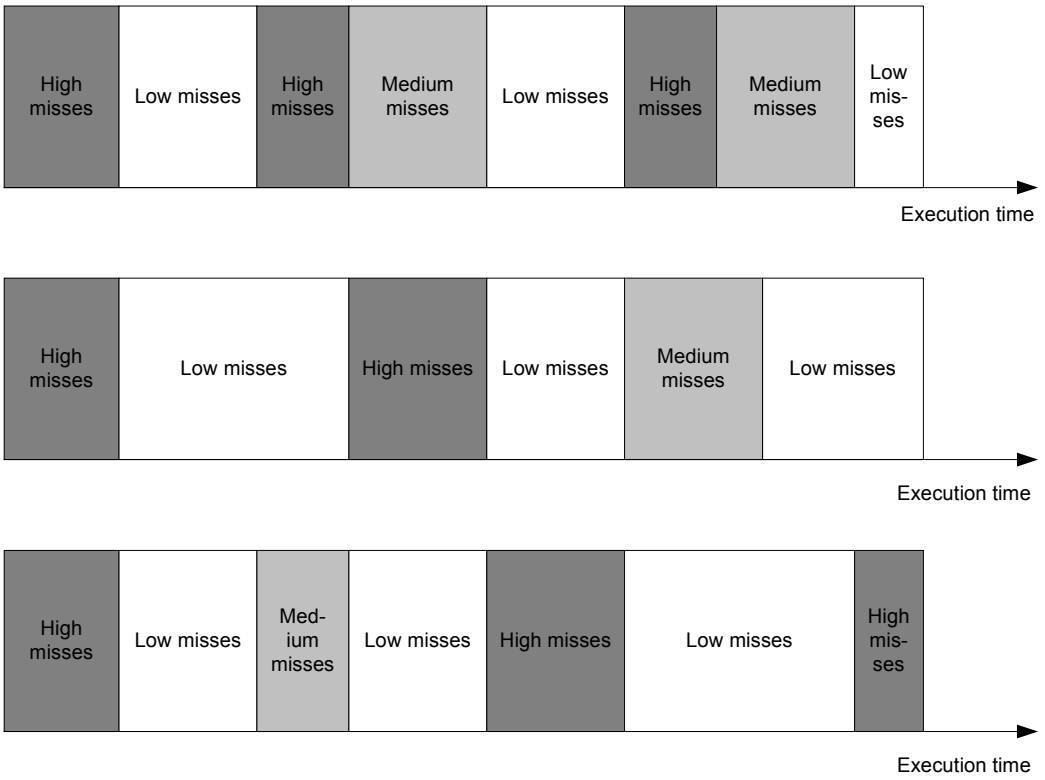


Figure 6.1 Generation of misses with respect to time

This is the behavior, which we want in our model: the irregularity in the generation of misses. We get the total number of available misses as an input and our job is to distribute them in different parts and then originate them in a manner that they look like appearing in irregular fashion. So for some particular number of clock cycles we have to introduce large number of misses. Then for some number of clock cycles, we have to introduce very little number of misses and then for some number of clock cycles we have to originate some optimum number of misses. To achieve this, we have developed some techniques, which are described in detail in Chapter 5 and it is assumed in the later discussion that you have read Chapter 5.

To test our techniques, we need to evaluate the traces generated by our CPU model as a result of some user supplied inputs and then see whether they are similar to the one shown in figure 6.1. There is one important question in this approach: how we can set the length of the region which can be termed as a region of high, medium or low misses. For e.g., consider the first trace of figure 6.1. The first region is dark grey (high misses) and

the second region is white (low misses). If we combine them into one region, it will become light grey (medium misses).

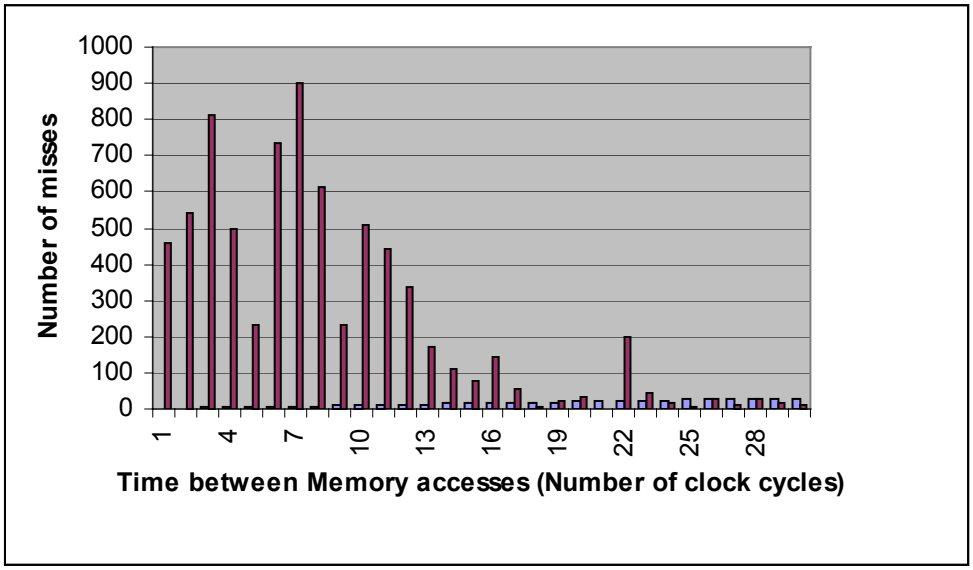


Figure 6.2 Generation of misses viewed in another way

Figure 6.2 shows an alternative. It shows how a bar graph between 'number of misses generated during program execution ' and 'time between memory accesses' in terms of number of clock cycles looks like. To understand this figure, consider the bar at 1, which shows the value of 'Number of misses' between 400 and 500, say 450. It means that 450 times it happens that the time difference between two successive memory accesses is 1 clock cycle. This graph represents the scenario, when a single CPU is connected to the memory and there is no sharing. Also the latency of the memory is eliminated. If the memory latency is included, then the minimum time difference between two successive accesses would be memory latency (number of clock cycles to complete memory access) +1. So if the memory latency is 20 clock cycles we will see the first non-zero value at 21.

Figure 6.2 shows that there are peaks on the left side of the graph. As we move from left to right these peaks start decreasing and nearly vanish after 60-70 (not shown in figure). This is what we can find in most programs. Most misses occur in quick succession i.e. the time difference between two successive memory accesses is small. For some period of time either zero or very few misses occur and then for some period of time a bunch of misses occur. Therefore most programs execution generates peaks on the left side of the graph. If we compare figure 6.1 with figure 6.2,it can be clearly seen that the peaks on the left side of the graph in figure 6.2 is due to these dark grey regions. Figure 6.2 shows the same behavior, which is shown in figure 6.1. But the method use in figure 6.2 is a more clear way and provides better judgment to make comparison.

So we have to perform simulations with our CPU model using different values of input parameters, record the misses and then generate bar graph like in figure 6.2 and then see

whether we get the peaks or not and if we get them then whether they are on the left side, right side or in the middle of the graph.

6.2 Recording Activity on the bus

To follow the approach of section 6.1, we need to record all the events on the bus connecting the CPU and the memory. An activity on the bus means that a miss is generated.

We have shown our test bench in figure 3.1 in chapter 3. The test bench consists of our CPU model, memory and the clock. We have added another entity in the test bench known as 'Log entity'. The 'Log entity' records activity on the bus. Whenever CPU communicates with the memory, it records that event i.e. write it in a file. It records whether it is a read access or a write access. If it is a read access then whether it is an instruction read or a data read. It also records the cycle number at which the access is initiated. The variable that contains the value of cycle number increments by one at the end of every clock cycle. At the end of the simulation, it generates two files. The first file contains the complete data i.e. a record of all the accesses that contains the cycle number at which the access was initiated as well as type of access. The second file is used to generate a graph shown in figure 6.2. The time difference between every two successive memory accesses is calculated and is stored in an array. Therefore array[7] = 123 means that 123 times it has happened that the time difference between two successive accesses is 7 clock cycles. Therefore every time when it is found that the difference between two successive accesses is 7 clock cycles the value stored in location array[7] increments by 1.

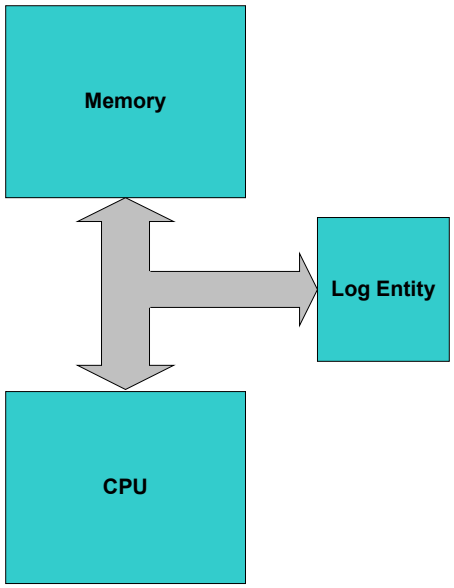


Figure 6.3 Log Entity recording the activity on the bus

6.3 Testing the Simulation of the model

As described in the previous chapter, the simulation is divided among three major modes: Compulsory mode, Random mode and Special mode. The Compulsory mode is a small mode and appears only once during simulation. Random and Special modes are the most prominent one. It is also possible to run the simulation with either Random mode or Special mode.

We have performed number of experimentations with our model. We have used different values of input parameters (number of instructions, miss rates, seeds). Since we are making many decisions in our model based on randomness, therefore value of seed is also important. We have tried different values of seeds by keeping all other parameters as constant. It is found that the different values of seeds doesn't affect on the output. Off course, every new value of seed produce different combination but the bar graph remains the same i.e. the peaks on the left side.

Consider the following example. The following input data is used to perform simulation. Figures 6.4,6.5 and 6.6 show the outputs when the simulation is performed with only Random mode, with only Special mode and then equal number of Random and Special mode. However, Compulsory mode is present in all these simulations but its appearance is very small as compared to the other two modes.

The input data is:

```
Total number of instructions     =  313917
Percentage of load instructions  =  20%
Percentage of store instructions =  5%
Instruction miss rate            = 0.0993
Data miss rate                   = 0.1445
CPI (including the effect of structural, data and control hazards) = 1
Memory Access startup latency (Number of cycles required to initiate
memory transfer after detecting a miss)  = 1
Memory Accesses end latency (Number of cycles required to restore the
normal execution after the completion of memory transfer) = 1
Seed for the random functions  = 25
```

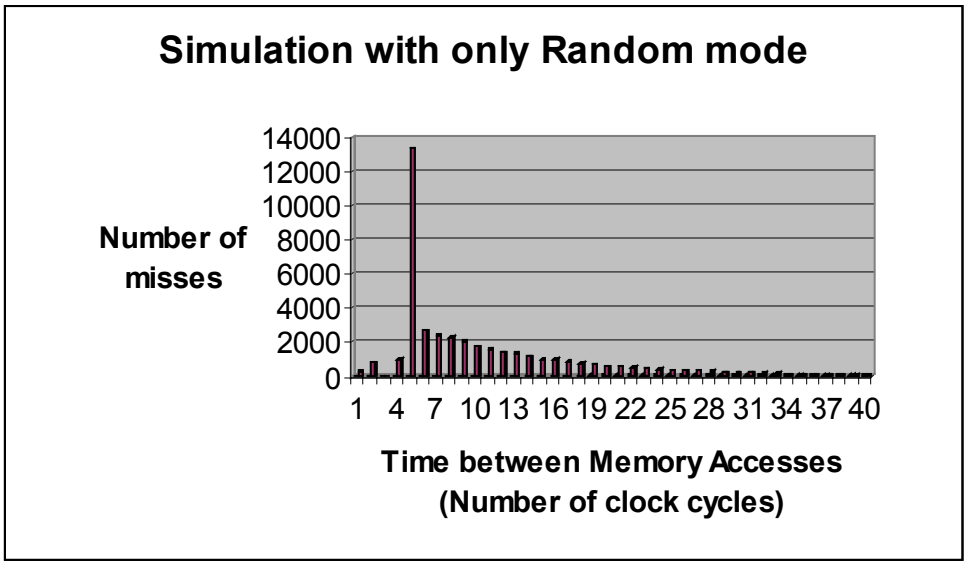


Figure 6.4

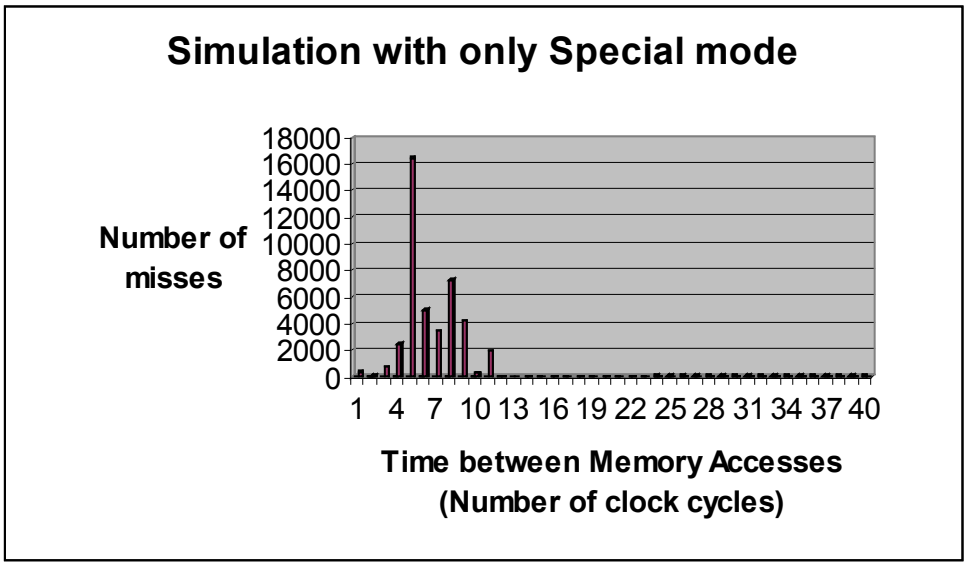


Figure 6.5

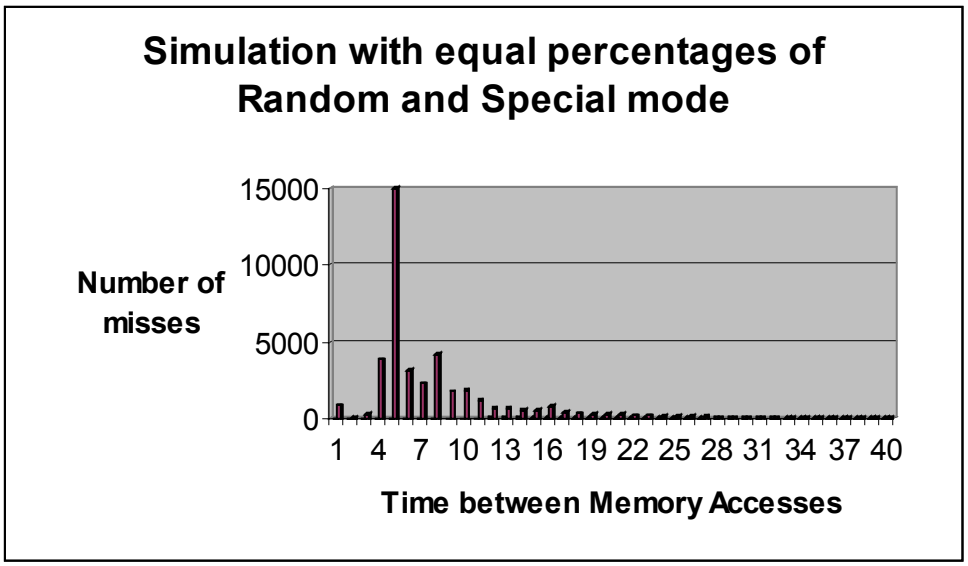


Figure 6.6

As shown from the figures, all combinations generate peaks on the left side. The heights of the peaks are not important, the variation with respect to each other is important. In all the figures, as we go from left to right they starts decaying and this is our desired behavior. The special mode is used to introduce region of high misses of figure 6.1 and Random mode is used to produce region of medium misses. Therefore the peaks generated by Special mode are higher than Random mode. When all these modes are combined together they approximate the behavior shown in figure 6.1.

6.4 Comparison with Real Programs

Initially, it was decided that the simulation with CPU model is compared with the simulation of real cores to test the accuracy of our CPU model. This means that we will take some original software and run it in a system consisting of a memory and real CPU core. We will record the activity on the bus in the same manner as we have done in our test bench i.e. creating a log entity and connect it with the bus connecting the CPU core and the memory. Then we will generate the same bar graph shown in figure 6.2 from this real simulation and then we will compare it with the graph generated by our CPU model.

The first question about this approach is: Is this approach correct?
Suppose we have followed the above-mentioned procedure and at the end we have two bar graphs that have to be compared with each other. If we find that these two graphs are not similar then what should we do? Does it mean that our CPU model is not correct? Conversely, if we have found that the two bar graphs are very similar to each other. Does it mean that our model is very accurate?

We have designed our CPU model by keeping in mind how general software behaves? The bar graph generated by the real simulation is due to that specific single software. If the two bar graphs are not similar then should we start modifying our model so that it generates the graph similar to that one? We shouldn't do that. If the two graphs are not similar it doesn't mean that the CPU model is not correct and if they are similar we shouldn't finish our testing process by calling it a success.

We should collect number of softwares at least more than dozen, which are different from each other in terms of routines written in it. Then we should repeat the same procedure with the output of all of these programs. Calculate the total number of instructions of each program, instruction miss rate, data miss rate etc. Then we use this data as input in our CPU model and then collect the statistics generated by our program. Then with the help of these numbers of comparisons, we will be confident to comment on the accuracy of the CPU model.

6.4.1 Design of testing system

Unfortunately, in most projects a significant amount of time is consumed in doing things, which are not the actual part of the project. This thesis was also not an exception. The following discussion of testing process will illustrate this.

The first step of this testing process is to build a test environment for the real simulation. For this we need an IP core of real CPU and the memory. We have found a behavioral model of ARM processor, designed for simulation purposes. The simulation of this ARM model is equivalent to the simulation of synthesized version of IP core.

Before the simulation this processor model needs to be configured. A software has to be written to configure it i.e. setting the TLB tables, enable or disable the caches etc. This software has to be combined with the software that has to be run on the processor. Both these programs are compiled by a Make file, which combine the output binary files of both of these programs into a single file. The combined software is then loaded into the memory.

When the simulation starts, the processor starts accessing the memory from the first location. Therefore the configuration software starts executing i.e. the configuration of the processor starts. After the completion of this configuration, the normal software starts executing. The processor has 16 KB of separate instruction and data caches. It is possible to enable or disable the caches with the help of configuration software [1].

A memory has been designed to interface with this ARM processor. The memory, which we have used in our test bench, cannot be used because this model of the ARM processor doesn't follow the OCP interface. The interfacing with the processor didn't prove to be a small task because it requires knowing about architecture of the processor i.e. transitions of different signals, when the input and output data is latched etc and how the data is aligned in memory.

As told earlier, this processor model has 16 KB of separate instruction and data cache. Also the memory with which the processor communicates consists of only software i.e. the configuration software that executes at the beginning and then the normal software, which is our testing software. This situation is different from the usual execution i.e. when the given software is executed under the supervision of an operating system. The operating system allocates memory to execute the software and also handles the exceptions, which occur during program execution. In nearly every program, we use input/output functions (printf(),scanf() etc) and memory allocation functions (malloc(),pointers, linked lists). Absence of operating systems means these programming features should not be present in the code. Summarizing the above discussion, this situation sets two major restrictions on the programs, which have to be used for testing purposes:

- The program should be large enough in size, so that it will not fit in 16 KB cache.
- The program should not contain any piece of code, which requires operating system.

The first condition implies that significant number of misses should be generated during the execution of the program. The whole comparison and in fact the whole project is based on misses. If the program is small it will fit into the caches and so the misses produced would be very small. The conflict misses (which contribute the major part of misses in most programs) would be almost zero. Most misses would be compulsory. Therefore to make some real evaluation, we need big programs, which have a code size much larger than 16 KB. So that during its execution a significant number of conflict misses would occur.

As explained, the second condition implies that the software code shouldn't contain any operating system dependent programming. It means that the program should be free of all input /output functions, memory allocation functions.

The required softwares are basically the benchmarks specially written for the performance analysis of different systems. Therefore for testing, we need benchmarks with large code size and they shouldn't contain any operating system dependent code. Due of this restriction we can't use benchmarks, which are used in normal PCs because they have lot of input/output functions as well as operating system functions. It is possible to clean the code with the input/output functions by putting an effort. But its nearly impossible to remove operating system dependent piece of code, which include complex data structures. Therefore, we need benchmarks that are used for embedded applications and our CPU model is also focused on processors used in embedded applications. The embedded benchmarks use much simpler programming constructs than other conventional benchmarks. Thus to make a good analysis of our model we need some embedded benchmarks which follows the above-mentioned conditions so that we would be able to run them in our test system containing the ARM processor model.

The design of the test system was started even before the completion of the CPU model. The intention was that the detailed traffic analysis of embedded benchmarks also helps us

in making right decisions in the development of the model. However, at that time these benchmarks were not available and we were hoping that we would get them soon. Unfortunately, this wasn't happened. A lot of effort has been made but we haven't been able to find any benchmark, which also fulfill the above-mentioned conditions. Because of the absence of the benchmarks, we can't able to make any comparison with the real programs, although every other necessary requirement was fulfilled.

Another Approach

During the last month of the project, we have tried another approach. As mentioned earlier, that simulation with this ARM processor model can be done by either enabling or disabling caches. We have changed the configuration software to disable the caches. It means that now the processor does not have any cache. In other words, we can say that every access to a cache result in a miss. We have written some small programs keeping in mind the above limitations (operating system conditions). Although, these programs are small but now we can find some traffic on the bus because the caches are disabled. We recorded the activity on the bus with the help of our 'Log entity' but this time we have recorded the memory addresses as well. Whenever the memory is accessed the corresponding address is also recorded.

We wrote a C program in which different cache formulas given in section 5.1.4 or in [2] and [3] are implemented. These formulas are used to find cache hits or miss with the help of block address, tag size, number of blocks in the cache etc. Block address is obtained from the address trace. For e.g. in a direct mapped cache the block placement is done by the formula:

(Block address) MOD *(Number of blocks in cache)*

We have implemented these formulas and made some arrays, which represent the instruction and data caches. In short we have developed a model of the cache in a C program. The program reads the log file generated by the 'Log entity'. Takes all the information about type of access, clock cycle at which it is accessed, address of the access etc. And then with the help of these cache formulas it calculates that in a presence of a cache whether that access would result in a hit or a miss. We assumed a small cache i.e. small size with less number of blocks. Thus proceeding in this way we have generated the bar graph from the executions of these small programs.

We have made some comparisons between the output of the test system and the output of our CPU model. We have found similarities in few cases and dissimilarities as well. However, we have rejected our results i.e. we haven't felt confident to comment on the accuracy because of the presence of following doubt in the approach:

The CPU model is designed keeping in view a pipeline machine when more than one instructions are active in a cycle and we can expect misses closer to each other in the order of occurrence. If the caches are disabled, the processor is not really working as a

pipeline machine. In a pipeline machine, instruction cache is accessed in every clock cycle. Therefore every instruction access should be a miss. This stalls the processor. To prevent the pipeline from permanent stall, it is required that the instructions in the next pipeline stages should remain continue executing. When the memory access is completed then at that time the previously issued instruction is already completed. After issuing this new instruction the processor starts accessing memory again and thus stalls again. So in practical, the processor would not work as a pipelined processor. Therefore, it is not safe to come to any conclusion based on the results of these test systems.

6.5 Concluding Remarks

We have designed our model according to the general behavior of the CPU i.e. how it behaves during the execution of most programs. The relationship between misses and the software is carefully studied and implemented in the best possible way. We have shown through our test approaches that we have correctly implemented the concepts, which we have developed in our study of misses. We got the behavior, which we wanted to see in our CPU model.

Special mode is designed to introduce lot of misses therefore it is targeted towards smaller caches and thus high miss rates. Random mode is designed to produce medium number of misses based on the miss rate. It is our observation that Special mode represent high miss rate behavior better than Random mode and similarly Random mode represent low miss rates better than Special mode.

We have done a reasonable effort to test our model against the real program execution. We have developed every necessary item that can help us in this process. Unfortunately, due to the unavailability of embedded benchmarks we can't complete this process. We have every necessary item except the benchmarks. In future, if we get them we can do the testing with much lesser effort.

The model is made in a very a flexible manner. It provides lot of options to the user to run the simulation in different flavors. The traffic analysis generated by the model depends on two major factors. The first factor is the rules of distribution of Execution clock cycles and the misses among different modes. The working depends on two parts: the distribution of execution clock cycles and the misses and the rules with which these misses are originated in each cycle. These rules of distribution are very much user controlled. And the parameters which are not user controlled, their values are #define in the code. If these values are modified, changes at the beginning will adjust all the formulas in the code accordingly. The second factor is the rules of generation of misses in each cycle. These rules are different for each mode. Separate functions are written to represent each mode. Therefore its also very easy to modify them.

Consider the worst case, if in future some serious bugs are found in the model, it will be easy to remove them. All the rules can be modified very easily. But remember we are talking about the very worst case. We are very hopeful that this will never happen and

even if some thing goes wrong the user-supplied options will be enough to control. We feel that we have completed the tasks given section 1.3.

6.6 SUMMARY

In this chapter, we have described our approach for testing. First we have shown that what kind of results we are looking for and then we have shown some simulation outputs of our model. We have shown that we became successful in implementing the desired behavior in our model. Then we described, our different efforts to do testing with the real programs, which haven't been completed due to the unavailability of embedded benchmarks.

REFERENCES

[1] ARM9 Embedded Trace Macrocell (ETM9) Technical Reference Manual

[2] Computer Architecture A Quantitative Approach by John L Hennessy & David A Patterson.

[3] Computer Organization and Design by John L Hennessy & David A Patterson.

CHAPTER

7 Conclusion

Finally we came to the last chapter. In this chapter we will briefly describe our experiences during the whole project, possible future expansions and our feelings at the end.

7.1 Possible usage of CPU model

During the whole project, we have designed and experimented in an environment of one CPU and one memory. And it is because we have to make solo model of CPU i.e. how it behaves in isolation. But definitely it will not be used for performance estimation in such an environment. It will be used in an environment when a bus is shared by more than one CPU to access the memory. Therefore the most common use of the CPU model is to make two or more instances of it and then connect it in a given communication architecture and then simulate. At the end, we are basically interested in estimating the effective CPI for each instance of the CPU model. All the effort we have done to generate the misses is basically used to estimate this value. In an environment of more than one CPU, when the misses are generated in two CPUs in the same cycle, only one would be able to get access of the bus, therefore the latency for the other CPU would be more than what it has in isolation. Therefore depending upon the configuration of the communication architecture, the effective CPI of each CPU instance would be different than what it has in isolation. The communication architecture, which gives the smallest CPI for most CPUs, is the best communication architecture.

Although we are not able to comment on the accuracy of the model with confidence but it should be noted that expectations were not very high even before the start of the project. For us even the 50% accuracy is enough. We should keep in mind what we are doing: Running simulation without any software. Imagine the ease what this tool is providing. We can simulate millions of instructions in five minutes (in fact we are not running the instructions). Running same number of instructions on real cores require a full day. It is also possible that we can make use of all the options of the model, collect different

figures (CPI) and then take the average to increase the accuracy. Even if we need to run it several times with different settings: it will not take more than an hour (in fact even less). And most important we don't need to look and search for the softwares (like we did for the benchmarks) that need to be run for simulation. If more work has to be done on it (to increase its accuracy through testing and adding more behaviors) and its credibility has been proven then it will prove to be an excellent tool for performance evaluation.

7.2 Future Expansions

After reading chapter 6, it is clear that one of the most important future works is to get some embedded benchmarks and then follow the testing procedure given in chapter 6 and try to uncover the bugs (if they are) present in the model.

We have done our best to make the model as modular and flexible as possible. We have divided our simulation into many modes. In each mode we have represented some behavior. It is possible to add more behaviors in the model by adding more modes. Table 5.2 shows the ID of each mode, which is used to go to the function representing the particular mode. The mode with highest ID is 6. Therefore numbers greater than 7 can be used for future modes. If some new behavior needs to be added, we can write a separate function representing it, assigned it an ID along with some percentage of Execution clock cycles and misses. And then that mode will start appearing in the simulation like others.

The Special mode is added to introduce greater number of misses in quick succession i.e. it is focused towards high miss rates or we can say small caches. One possible expansion would be to add some modes which are focused towards low miss rates i.e. they introduce gaps between misses. In other words, they generate peaks on the right side of the bar graph in figure 6.2.

7.3 Experiences

The programming in this project was quiet straightforward except few stuff of FLI. The challenging task in this project was to come up with some good ideas that also have strong theoretical background. In this project we have shown many new ideas. What you have read in this project are only those ideas, which remains valid until the end. Many ideas have been made, implemented and then rejected. The technique, which we have used in this model, is a result of continuous thinking and discussions consisting of many weeks.

It has been found during the initial phase, that no work has been done on this topic before. When we have started this work we have tried to gather the maximum information that can help us. We have read more than 50 papers related to caches. The intention was to get some mathematical formulas about the misses. If we get or able to develop some mathematical formulas for the misses then we can straightaway implement

them in the model. Some papers have been found that present some mathematical analysis of the misses. But all these formulas were based on the memory addresses. And as written many times, without software we can't know the address. Therefore we realized that we have to do something completely at our own and this is what we have made.

Although the implementation of FLI is as straightforward as shown in Chapter 3. This straightforward way was not known at the beginning. For even a small minor error, the ModelSim crashed every time leaving no clue about the error. The job was then to find this error, which consumed significant time. What we have learnt from this FLI implementation is: try to do as simple as possible. Fancy programming approaches always result in problems for this FLI at least in our case.

Although the model was written in C but at many places in the C code (OCP protocol implementation) we have to think like writing code in VHDL because the functions written in the C program runs with the rising transitions of the clock, which was a new experience. The study and implementation of OCP was also worthwhile.

A lot of effort has been made in the testing process. Unfortunately, it went unrewarded at the end.

Project as learning point of view

As a student, I have learned a lot in this project. This thesis was an excellent learning opportunity to know more about Computer Architecture. The task assigned in the thesis was very unique. Since it was totally a new kind of work and enough help from the reading materials were not available therefore it forced me a lot to think. I haven't worked on such kind of project before where you have to come up with some new idea of your own and then test whether it is right or not. At the end, I am satisfied with my work but still feel that more could be done.