Implementation of 5 stage pipelined RISC V processor

A Project Report

submitted by

M Ravi Chandra
EE12B036

i*n Partial Fulfillment of the Requirements*
for the Degree of

Bachelor of Technology

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

April 2016

THESIS CERTIFICATE

This is to certify that the thesis entitled **Implementation of 5 stage pipelined RISC V processor**, submitted by **M Ravi Chandra, EE12B036** , to the Indian Institute of Technology Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. V Kamakoti
Research Guide
Professor
Dept. of Computer Science and Engineering
IIT Madras, 600 036

Place: Chennai

Date: **23th May, 2016**

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude towards several people who enabled me to reach this far with their timely guidance, support and motivation.

First and foremost, I offer my earnest gratitude to my guide, Dr. V. Kamakoti whose knowledge and dedication has inspired me to work efficiently on the project and I thank him for motivating me, and allowing me freedom and flexibility while working on the project.

My special thanks and deepest gratitude to Abhinaya Agrawal who has been very supportive with invaluable suggestions.

ABSTRACT

This project is about implementing 5 stage pipelined processor using RISC V Instruction set architecture realized in bluespec. With increased use of embedded systems we require a set of processors that use less power and area but yet achieve the main purpose of speed .

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

BSV	Bluespec System Verilog
MCP	Manager-Client pairing
TLM	Transaction Level Modelling
FIFO	First In First Out
RISC	Reduced Instruction Set Computer
HDL	Hardware Description Language
CPU	Central Processing Unit

CHAPTER 1

Introduction

1.1 Overview

The Processor design team of Reconfigurable and Intelligent Systems Engineering (RISE) Lab in the Computer Science Department of IIT Madras has been actively involved in research of The SHAKTI Processor project. My project targets E-Class processor which is 32 bit 5 stage in-order core aimed at 10 - 50 Mhz uC variants. This processor got an optional memory protection and very low power static design. The processor strictly follows the RISC-V Instruction Set Architecture (ISA). Entire design of the processor is done using a Hardware Description Language (HDL) named Bluespec System Verilog (BSV). This project describes the design and implementation of "5 stage pipelined RISC V processor". This work involves implementing this in Bluespec which is based on RISC-V ISA.

1.2 Organisation of thesis

Chapter 2 gives some insight about the Bluespec System Verilog, its key features, TLM module of BSV.

Chapter 3 discusses about RISC-V Instruction Set Architecture and different base encodings.

Chapter 4 gives us implementation of 5 stage processor with intermediate FIFOs and explains the flow of data in each cycle.

Chapter 5 contains information about hazards and operand forwarding.

Chapter 6 contains a conclusion and description on the future work.

CHAPTER 2

2.1 Bluespec System Verilog

The design of the blocks and their testing is written in Bluespec System Verilog (BSV). BSV is a high level Hardware Description Language. It expresses synthesizable behavior with rules, a rule can be viewed as a declarative assertion expressing a potential atomic state transition. The BSV compiler produces efficient RTL code that manages all the potential interactions between rules by inserting appropriate arbitration and scheduling logic, logic that would otherwise have to be designed and coded manually. BSV connects the modules by interfaces and methods. It also provides predefined library elements like FIFOs, BRAMs etc. which are modeled using BSV methods.

It has powerful static type checking which removes potential human errors which can't be detected at the stage of compilation normally but can be detected now during the compilation. BSV also has more general type parameterization (polymorphism) due to which modules and functions can be parameterized by other modules and functions, this enables the designer to reuse designs and glue them together in much more flexible ways. BSV's static elaboration helps to arrive at the design much faster than the other HDLs. The BSV compiler also can generate the synthesizable Verilog code of the written bluespec code which can be used later for synthesis purposes.

BSV has an inbuilt package called TLM (Transaction Level Modeling) which is used in this thesis.

2.1.1 TLM

The TLM package includes definitions of interfaces, data structures, and module constructors which allow users to create and modify bus-based designs in a manner that is independent of any one specific bus protocol. Designs created using the TLM package are thus more portable. In addition, since the specific signaling details of each bus protocol are encapsulated in pre-designed transactors, users are not required to learn, re-implement, and re-verify existing standard protocols.

The two basic data structures defined in the TLM package are TLMRequest and TLMResponse. By using these types in a design, the underlying bus protocol can be changed without having to modify the interactions with the TLM objects. TLM request contains either control information and data, or data alone. A TLMRequest is tagged as either a RequestDescriptor or RequestData. A RequestDescriptor contains control information and data while a RequestData contains only data. The table 2.1 describes the components of a RequestDescriptor and the valid values for each of its members.

The table 2.2 describes the components of a TLMResponse and the valid values for its members.

The TLM interfaces define how TLM blocks interconnect and communicate. The TLM package includes two basic interfaces: The TLMSendIFC interface and the TLMRecvIFC interface. These interfaces use basic Get and Put subinterfaces as the requests and responses. The TLMSendIFC interface generates (Get) request

and receives (Put) responses. The TLMRecvIFC interface receives (Put) requests and generates (Get) responses. These TLMSendIFC and TLMRecvIFC can be connected by mkConnection in the Connectable package of BSV.

The Data Structures Request Descriptor in TLMRequest, TLMResponse, and the interfaces provided TLMSendIFC and TLMRecvIFC are used extensively in this thesis.

TLM Data Structure:

```
typedef struct {

TLMCommand command;
TLMMode mode;
TLMAddr#(`TLM_PRM) addr;
TLMData#(`TLM_PRM) data;
TLMUInt#(`TLM_PRM) burst_length;
TLMByteEn#(`TLM_PRM) byte_enable;
TLMBurstMode burst_mode;
TLMBurstSize#(`TLM_PRM) burst_size;
TLMUInt#(`TLM_PRM) prty;
Bool lock;
TLMId#(`TLM_PRM) thread_id;
TLMId#(`TLM_PRM) transaction_id;
TLMId#(`TLM_PRM) export_id;
TLMCustom#(`TLM_PRM) custom;

} RequestDescriptor#(`TLM_PRM_DCL) deriving (Eq, Bits, Bounded);
```

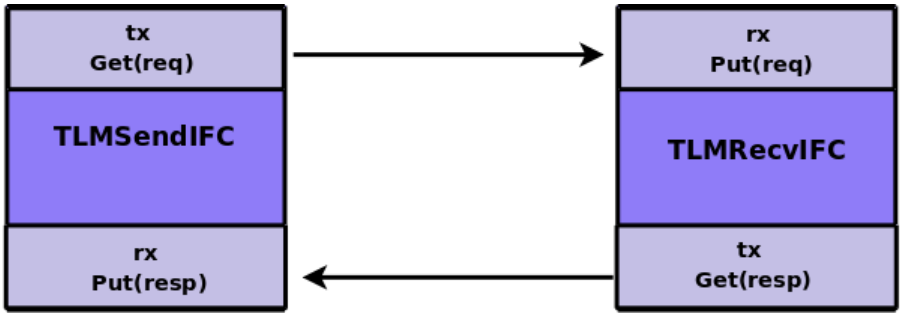
Table 2.1: Request Descriptor

RequestDescriptor		
Member Name	DataType	Valid Values
command	TLMCommand	READ, WRITE, UNKNOWN
mode	TLMMode	REGULAR, DEBUG, CONTROL, UNKNOWN
addr	TLMAddr#(`TLM_PRM)	Bit#(addr_size)
data	TLMData#(`TLM_PRM)	Bit#(data_size)
burst_length	TLMUint#(`TLM_PRM)	UInt#(uint_size)
byte_enable	TLMByteEn#(`TLM_PRM)	Bit#(TDiv#(data_size, 8))
burst_mode	TLMBurstMode	INCR, WRAP, CNST, UNKNOWN
burst_size	TLMBurstSize#(`TLM_PRM)	Bit#(TLog#(TDiv#(data_size, 8)))
prty	TLMUInt#(`TLM_PRM)	UInt#(uint_size)
lock	Bool	TRUE, FALSE
thread_id	TLMId#(`TLM_PRM)	Bit#(id_size)
transaction_id	TLMId#(`TLM_PRM)	Bit#(id_size)
export_id	TLMId#(`TLM_PRM)	Bit#(id_size)
custom	TLMCustom#(`TLM_PRM)	cstm_type

2.2: TLMResponse

TLMResponse		
Member Name	DataType	Valid Values
command	TLMCommand	READ, WRITE, UNKNOWN
data	TLMData#(‘TLM_PRM)	Bit#(data_size)
status	TLMStatus	SUCCESS, ERROR, NO_RESPONSE, UNKNOWN
prty	TLMUInt#(‘TLM_PRM)	UInt#(uint_size)
thread_id	TLMId#(‘TLM_PRM)	Bit#(id_size)
transaction_id	TLMId#(‘TLM_PRM)	Bit#(id_size)
export_id	TLMId#(‘TLM_PRM)	Bit#(id_size)
custom	TLMCustom#(‘TLM_PRM)	cstm_type

Figure 2.1: Connecting TLM Send And Receive Interface



CHAPTER 3

RISC-V ARCHITECTURE

RISC-V is a new instruction set architecture (ISA) that was originally designed to support computer architecture research and education, but which we now hope will become a standard open architecture for industry implementations. Main goals of RISC-V include:

- A completely open ISA that is freely available to academia and industry.
- A real ISA suitable for direct native hardware implementation, not just simulation or binary translation.
- An ISA separated into a small base integer ISA, usable by itself as a base for customized accelerators or for educational purposes, and optional standard extensions, to support general purpose software development.
- Support for the revised 2008 IEEE-754 floating-point standard.
- Both 32-bit and 64-bit address space variants for applications, operating system kernels, and hardware implementations.
- An ISA with support for highly-parallel multicore or many core implementations, including heterogeneous multiprocessors.
- Optional variable-length instructions to both expand available instruction encoding space and to support an optional dense instruction encoding for improved performance, static code size, and energy efficiency.
- A fully virtualizable ISA to ease hypervisor development.

3.1 RISC-V ISA Overview

The RISC-V ISA is defined as a base integer ISA, which must be present in any implementation, plus optional extensions to the base ISA. Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the user address space. The base RISC-V ISA has fixed-length 32-bit instructions that must be naturally aligned on 32-bit boundaries.

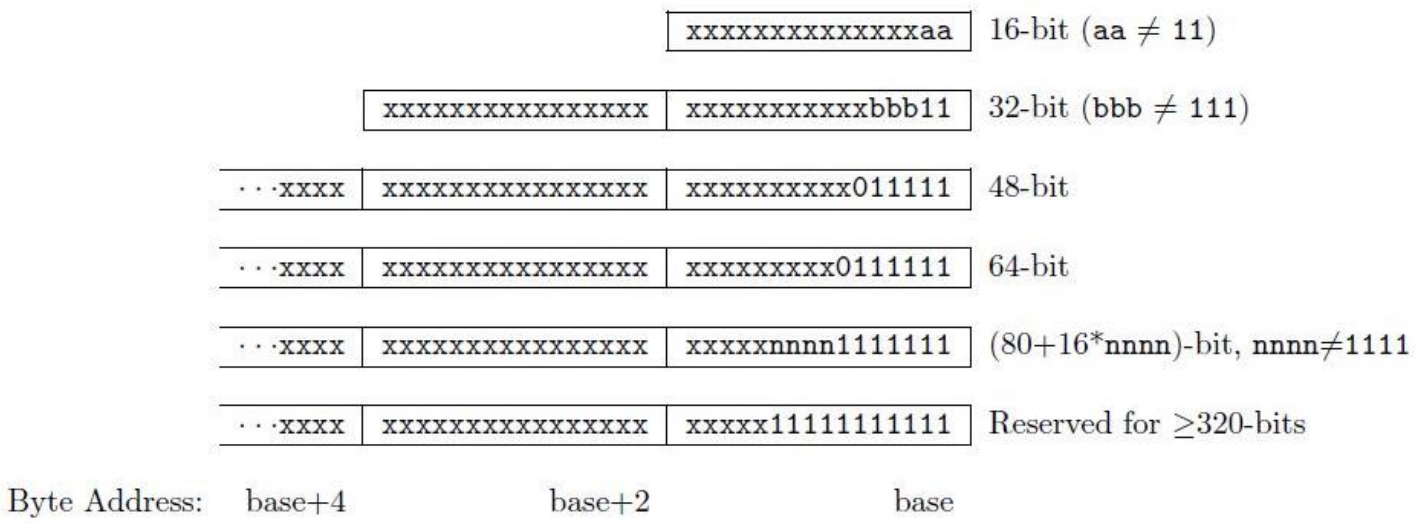


Figure 3.1: RISC-V instruction length encoding

3.2 Base Instruction Format

In the base ISA, there are four core instruction formats (R/I/S/U), as shown in Figure 3.2.1. All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction address misaligned exception is generated if the pc is not four-byte aligned on an instruction fetch. The RISC-V ISA keeps the source (rs1 and rs2) and destination (rd) registers at the same position in all formats to simplify decoding. Immediate are packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediate is always in bit 31 of the instruction to speed sign-extension circuitry.

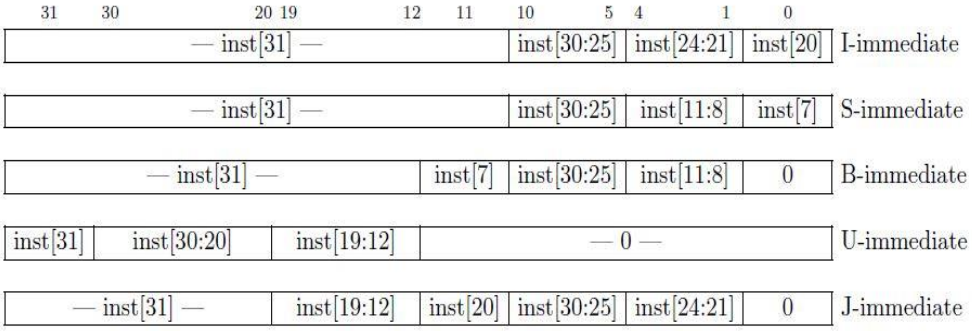


Figure 3.2.1: RISC-V instruction length encoding

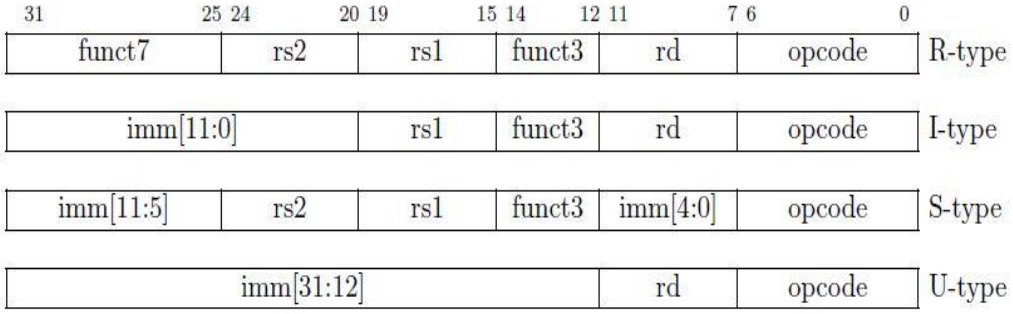


Figure 3.2.2: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

CHAPTER 4

5 Stage pipeline

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Multiple tasks operating simultaneously using different resources. Pipelining doesn't help latency of single task, it helps throughput of entire workload Pipeline rate is limited by slowest pipeline stage. There are five stages in RISC pipeline. So we use 4 FIFOs to control the flow of data from each stage to next till the present task is executed in that cycle.

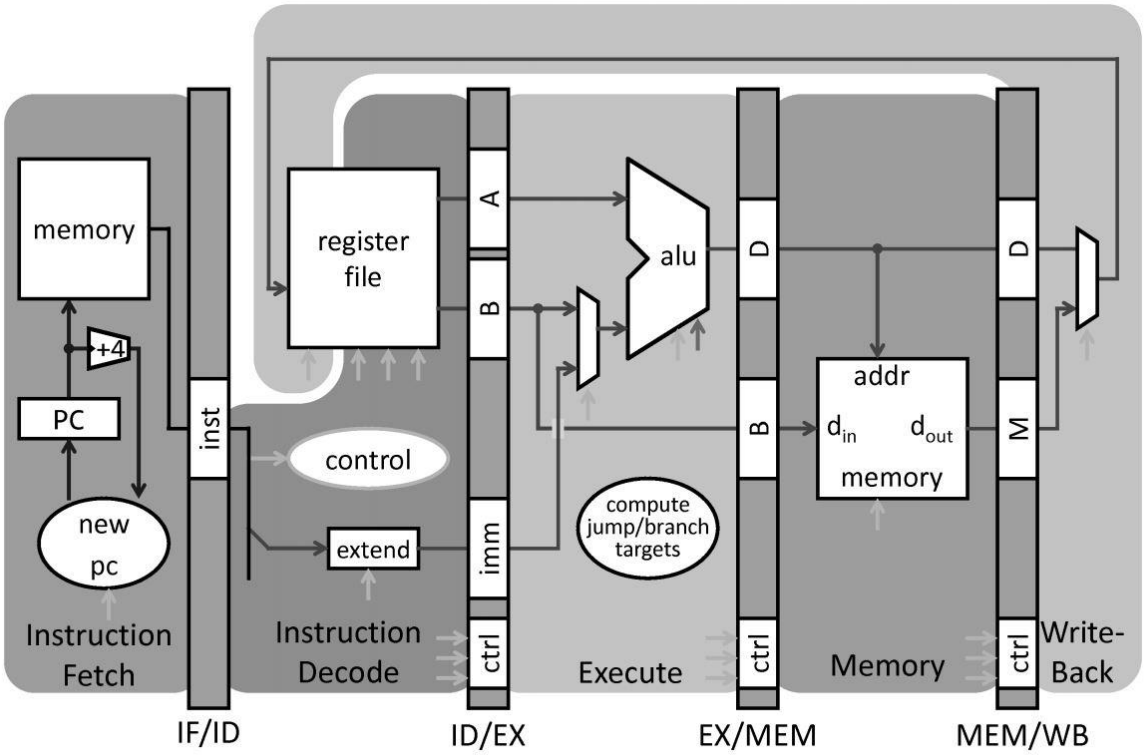
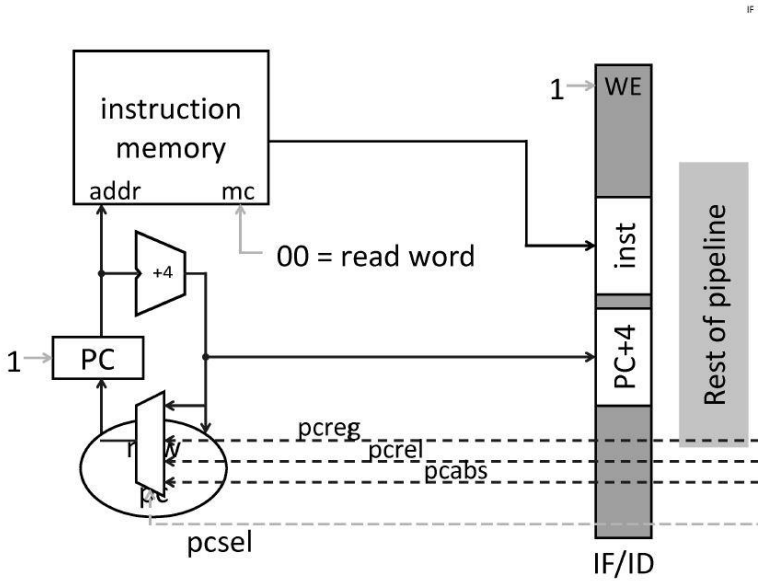


Figure 5: RISC-V pipeline stages with FIFOs.

4.1 Stages of pipeline

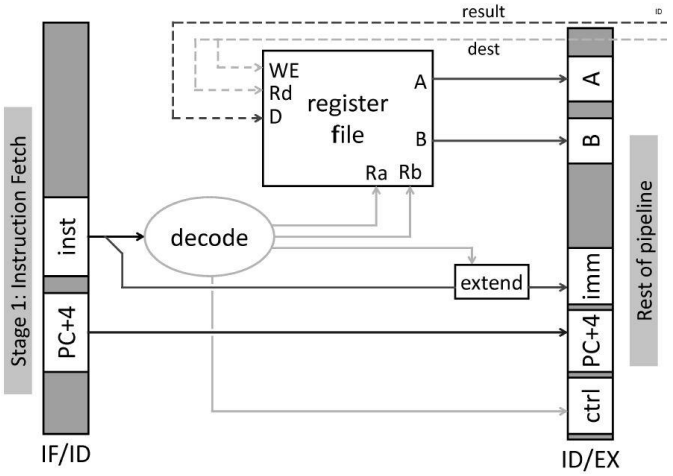
1. **Instruction Fetch (IF):**

   Current Program counter (PC) is index to instruction memory. Increment the PC at the end of cycle. Fetch the data from input and write the values of interest to Pipeline FIFO (IF/ID) between IF and Instruction decode stage.



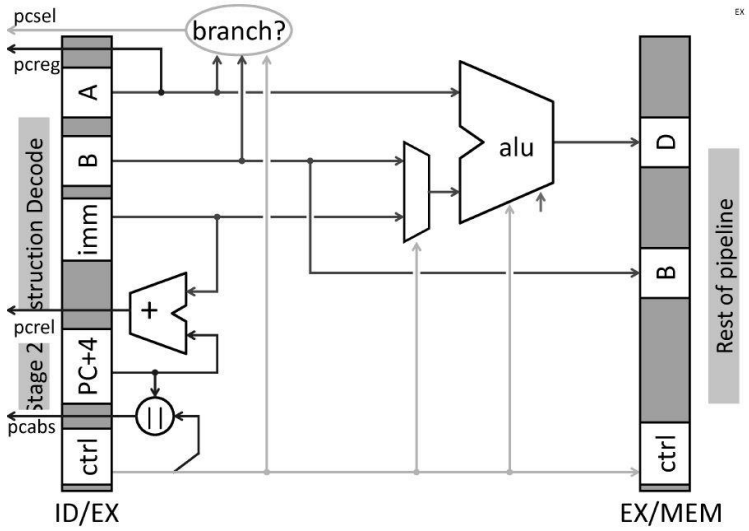
2. **Instruction Decode (ID)**:

   Read from IF/ID FIFO to get instruction bits. Decode instruction, generate control signals and then read from register file. Write values of interest to next pipeline FIFO (ID/EX). Control information, Rd index, immediates, offsets, contents of Ra, Rb are sent to next stage.



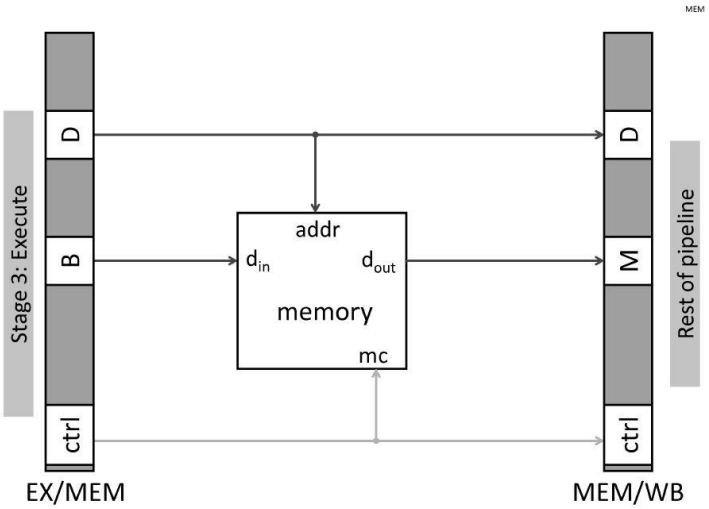
3. **Execute (EX):**

Read ID/EX pipeline FIFO to get value and control bits and then perform ALU operations. Compute targets (PC+4+offset) n case this a branch statement. Then write values of interest to next pipeline FIFO EX/MEM. Send Rd index, result of ALU operation, value in case this is memory store instruction.



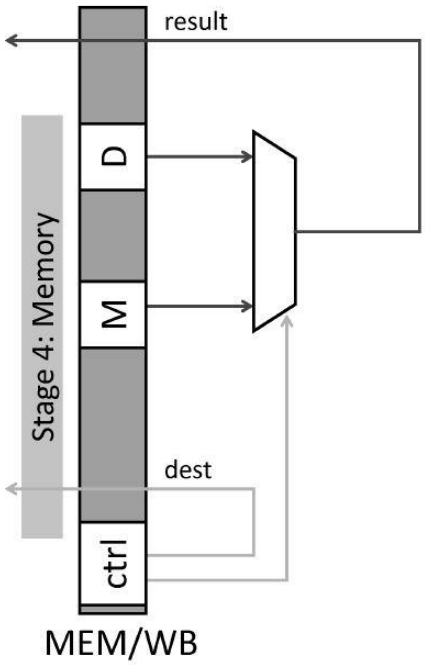
4. **Memory Access (MEM):**

Read EX/MEM pipeline FIFO to get values and control bits and then perform memory load/store if needed and the address is ALU result. Then write values of interest to next pipeline FIFO MEM/WB. Send control information, Rd index, result of memory operation, and pass result of ALU operation.



5. **Write Back (WB):**

On every cycle read from last pipeline FIFO MEM/WB to get values and control bits. Select the value and write it back to register file.



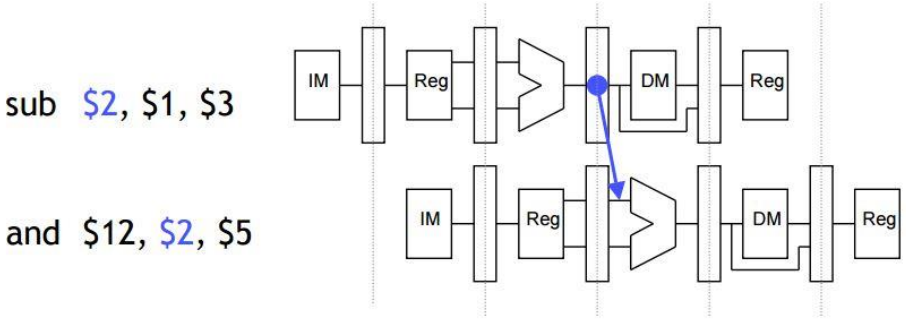
CHAPTER 5

Overcoming hazards

This chapter discusses about hazards and how to overcome it. Hazards are problems with the instruction pipeline in processor architectures when the next instruction cannot execute in the following clock cycle and can potentially lead to incorrect computation results. One of the solution to it is operand forwarding and we discuss about it in here.

5.1 Operand Forwarding

Operand forwarding (or data forwarding) is an optimization in pipelined processors to limit performance deficits which occur due to pipeline stalls. A data hazard can lead to a pipeline stall when the current operation has to wait for the results of an earlier operation which has not yet finished. So how can the hardware determine if a hazard exists?
An EX/MEM hazard occurs between the instruction currently in its Execute stage and the previous instruction if the previous instruction will write to the register file, and the destination is one of the ALU source registers in the Execute stage. There is an EX/MEM hazard between the two instructions below



Forwarding eliminates data hazards involving arithmetic instructions. The forwarding unit detects hazards by comparing the destination registers of previous instructions to the source registers of the current instruction. Hazards are avoided by grabbing results from the pipeline registers before they are written back to the register file

Hence the first ALU source comes from the pipeline register when necessary.

if (EX/MEM.RegWrite = 1 and EX/MEM.RegisterRd = ID/EX.RegisterRs)

then ForwardA = 2

The second ALU source is similar.

if (EX/MEM.RegWrite = 1 and EX/MEM.RegisterRd = ID/EX.RegisterRt)

then ForwardB = 2

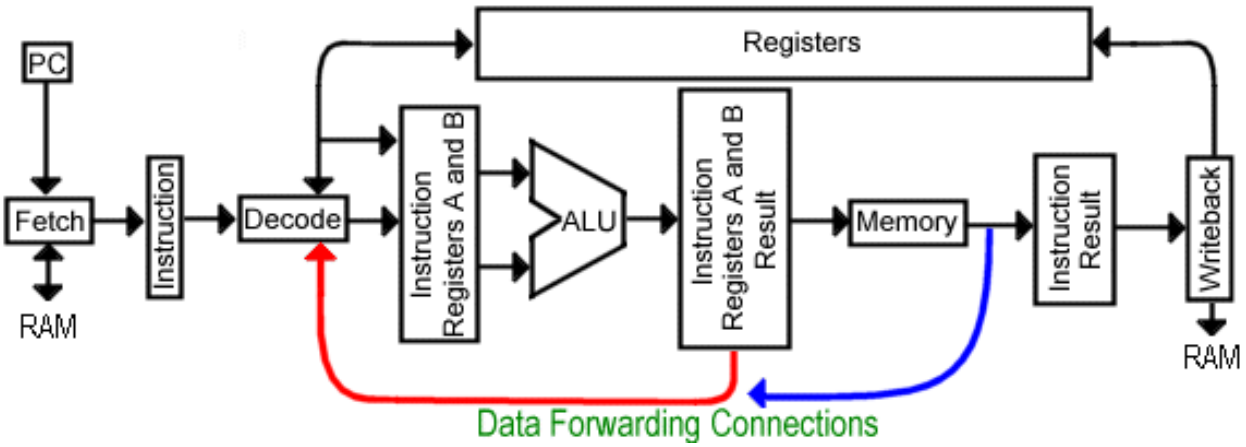


Figure 5.1: Operand forwarding

CHAPTER 6

Conclusion and Future work

Implemented 5 stage pipelined processor using RISC V Instruction set architecture realized in bluespec. This design is realized in Bluespec System Verilog (BSV) which provides module and configuration flexibility.

This thesis shows the implementing 5 stage pipelined processor using RISC V Instruction set architecture with Operand forwarding. We can also go further and implement branch predictors to improve the flow in the instruction pipeline. Branch predictors play a critical role in achieving high effective performance.

REFERENCES

[1] http://www.cs.cornell.edu/courses/cs3410/2010sp/lecture/topic10-pipelined-cpu-w-g.pdf.

[2] https://courses.cs.washington.edu/courses/cse378/07au/lectures/L12-Forwarding.pdf

[3] Bluespec,Inc.Bluespec System Verilog Reference Guide, Revision 30 July 2014.