ZHIYUAN COLLEGE, SHANGHAI JIAO TONG UNIVERSITY

Five-Stage MIPS Pipeline in Verilog HDL

Yurong You (ID: 5140519064)
ACM Honored Class

September 3, 2016

ABSTRACT

This report is a detailed summary on the five-stage MIPS pipeline project I have finish in this summer. The design is elaborated on both the whole and the part in this report. The pipeline supports all MIPS standard integer instructions except those related to coprocessors, and the forwarding, hazard control and branch control techniques are all fully implemented. The pipeline not only passes the basic Yamin Li's experiment code, but passes all self-designed test cases which are intended to test the pipeline's completeness on supporting every part of the MIPS integer instruction set. Source code in Verilog HDL and the pipeline blueprint are also provided.

Contents

1. Introduction and Design Philosophy

This report is for the final project of course MS108, computer system I. In this project, I implemented a comprehensive MIPS classical five-stage pipeline which supports all but two (co-processor instructions) MIPS standard integer instructions[1]. A summary on the supported instruction set are presented in appendix A.

On designing the structure of pipeline, I take [4] and [2] as reference, but I found that different functional units in Lei's design are coupling with each other, which will lead to inconvenience if we are to extend the functionality of our pipeline, and a large part of Li's design are low-level implementations on function units, which I think might be unnecessary. Thus I decided to design a pipeline for myself, follows the following principles,

1. Register-Transfer Level (RTL) modelling;
2. Separate different stages;
3. Separate control-path and data-path.

The blueprint of my design is presented in appendix B, note that due to the limited space, I do not circle what stage module consists of what sub-modules.

The branch control, hazard control and forwarding are inspired by [3]. Different functional units are implemented in different modules, the main part of which will be introduced in the next section. The modules are connected by `pipeline.v` to compose the whole pipeline. Source code is available on `https://github.com/YurongYou/MIPS_CPU` (will be set public after project deadline).

[1] `https://en.wikipedia.org/wiki/MIPS_instruction_set`

2. Main Modules Elaboration

2.1. Stage Modules

There are five stages in this five-stage MIPS pipeline.

2.1.1. Instruction Fetch

Implemented in `IF.v`. The main function of this module is to generate `PC`, the address of current instruction, which will be sent to instruction memory (`ROM`) in `pipeline.v` to access instruction. This module consists of merely a D-type flip-flop with enable port to store the address. It takes control signals and branch address from hazard control module and branch control module to implement those control.

2.1.2. Instruction Decode

Implemented in `ID.v`. This module is mainly a decoder, which takes instruction from IF module as the input and outputs several control signals. Those signals are

Signal	Meaning
WriteReg	whether or not is to write register
MemOrAlu	the source to write register is from memory or ALU
WriteMem	whether or not is to write memory
ReadMem	whether or not is to read register
AluType	the type of computation to perform in ALU in execution stage
AluOp	the subtype of computation to perform in ALU in execution stage
AluSrcA	the first source of ALU is (rs or shamt)
AluSrcB	the second source of ALU is (immediate or rt)
RegDes	which register is to be write
ImmSigned	use signed or unsigned extended immediate
is_jal	whether or not this instruction is a jal instruction

Table 2.1: A Summary on the decoder's output signals

Note that the "MemOrAlu" signal is indeed redundant (can be inferred from "WriteReg"); and the reason why there should be a "is_jal" signal is that this instruction is asking to put $PC+4$ into the 31th register and jump to a specific instruction address, which is too special to be

integrated effortlessly with other instructions in my implementation. These signal will go all the way down to the following stage to control the pipeline.
This module also output several (signed/unsigned extended) segmentations on the instruction such as shamt, imm, etc.

2.1.3. Execution

Implemented in `EX.v`. This module implements the forwarding on the source of ALU and performs the computation specified by “AluType” and “AluOp” in ALU. The forwarding is performed according to the signals sent by the forwarding module.

2.1.4. Memory Access

Implemented in `MEM.v`. Memory access follows the standard of Wishbone Bus [2], which has a brief usage guide on page 257 of book [4]. I use a write memory controller and a read memory control to modify the data in order to get the desired form. This module also implements the forwarding on the source of memory write according to the signals sent by the forwarding module.

2.1.5. Write Back

Not implemented as a single module, since there is just one switch on the source of data, from the ALU or the memory, to write back into the register file.

2.2. Stage Sandwich Modules

Also called “pipeline registers”, but I prefer this name. These modules are merely collections of D-type flip-flops with enable port, which can update data on the rising edge of clock. See source code `if_id.v`, `id_ex.v` and `ex_mem.v`.

2.3. Control Modules

[2]https://en.wikipedia.org/wiki/Wishbone_(computer_bus)

2.3.1. Forwarding

Forwarding unit takes signals from various stages and output the forwarding control signals. There are 7 kinds of forwarding, and the control logic is presented as follows,

1. Forward Alu result in MEM stage to EX stage

   ```
   data_addr_EX == target_MEM && WriteReg_MEM == ‘WriteEnable &&
   target_MEM != 0
   ```

2. Forward Alu result in WB stage to EX stage

   ```
   data_addr_EX == target_WB && WriteReg_WB == ‘WriteEnable &&
   target_WB != 0
   ```

3. Forward Mem result in WB stage to MEM stage (A load followed by a store)

   ```
   data_addr_MEM == target_WB && WriteReg_WB == ‘WriteEnable &&
   target_WB != 0
   ```

4. Forward Alu result in EX stage to ID stage (for branch control)

   ```
   data_addr_ID == target_EX && WriteReg_EX == ‘WriteEnable &&
   target_EX != 0
   ```

5. Forward Alu result in MEM stage to ID stage (for branch control)

   ```
   data_addr_ID == target_MEM && WriteReg_MEM == ‘WriteEnable &&
   target_MEM != 0
   ```

6. Forward Hi/Lo in MEM stage to EX stage

   ```
   we_hi(lo)_MEM == ‘WriteEnable
   ```

7. Forward Hi/Lo in WB stage to EX stage

   ```
   we_hi(lo)_WB == ‘WriteEnable
   ```

2.3.2. Hazard Control

There will be only one kind of hazard in my implementation: a load instruction immediately followed by an not-store operation which has data dependency on it. In this case, we need to stall the pipeline for one cycle — keep PC register and IF/ID sandwich register remain unchanged and reset ID/EX sandwich register to zero in the next clock cycle.

2.3.3. Branch Control

Branch control can be viewed as performed in the ID stage. Since in my implementation there is no delay slot, we have to abandon the previous fetched instruction in IF stage if branch is taken. In practice, just reset the If/ID sandwich register in the next clock cycle and sent the branch address to the PC register.

2.4. Register File

Note that the register file writes at the falling edge. The reason of doing so is that if the register writes at the rising edge, the same with sandwich register, it will write the data in the next clock cycle, which leads to some additional code to support forwarding inside register file.

3. Test Summary

All test code is available on `https://github.com/YurongYou/MIPS-CPU-Test-Cases`. There are 8 tests in total, 6 of them are respectively intended to test a major part of the MIPS integer instructions. And there is a Li's experiment code test. On each test folder, there are instruction data (`.data`), MIPS source code (`.s`) and my waveform simulation screenshot (`.png`). The waveform is captured on Scansion[3].

To run the tests,

1. modify the `SOPC.v` file, include the pipeline, fill in the corresponding instruction data location;

[3] `http://www.logicpoet.com/scansion/`

2. compile the `SOPC.v` (I use iverlog[4] on mac).

3. observe the waveform, take branch test as example in figure 3.1

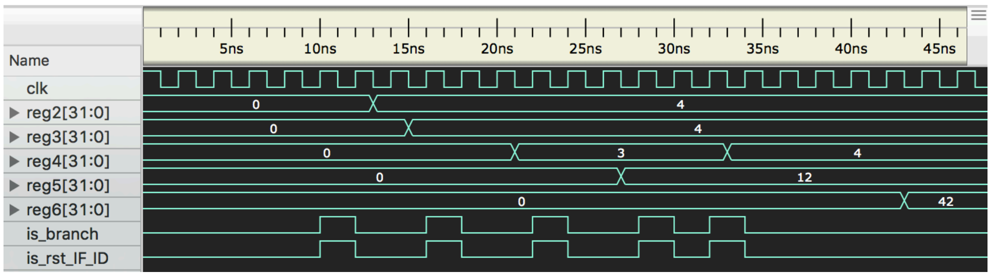


Figure 3.1: waveform for branch test

It is easy to check that the pipeline is running as expected, which finally writes the answer to life, the universe and everything into register 6.

4. Further Enhancement

1. Out of order execution. I think classical five-stage pipeline is quite difficult to improve performance. Straight forward ways to speed up such as multiple issue and multiple ALUs will be hard to be implemented under in-order execution pipeline structure. Thus maybe dynamic scheduling structure will have better promise on the performance.

2. Cache. I have not implement cache by now since all test is functional simulation and there is no memory delay. If we are to run the pipeline on the FPGA using the provided RAM, data cache and instruction cache are indispensable.

3. Exception and interrupt. To run an operation system, the CPU must be able to handle internal exception such as overflow, system call, etc. and external interrupt such as I/O.

4. Synthesis on FPGA. This is the ultimate goal of the project, but there still a lot to tackle on it.

[4]http://iverilog.icarus.com

5. ACKNOWLEDGEMENT

Special thanks to Zihao Ye, Zhijian Liu and Tianyao Chen for effective discussion.

REFERENCES

[1] J.L. Hennessy, D.A. Patterson, and K. Asanović. *Computer Architecture: A Quantitative Approach*. Computer Architecture: A Quantitative Approach. Morgan Kaufmann/Elsevier, 2012.

[2] Yamin Li. *Computer Principles and Design in Verilog HDL*. Tsinghua University Press, 2011.

[3] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2013.

[4] 雷思磊. *自己动手写 CPU*. 电子工业出版社, 2014.

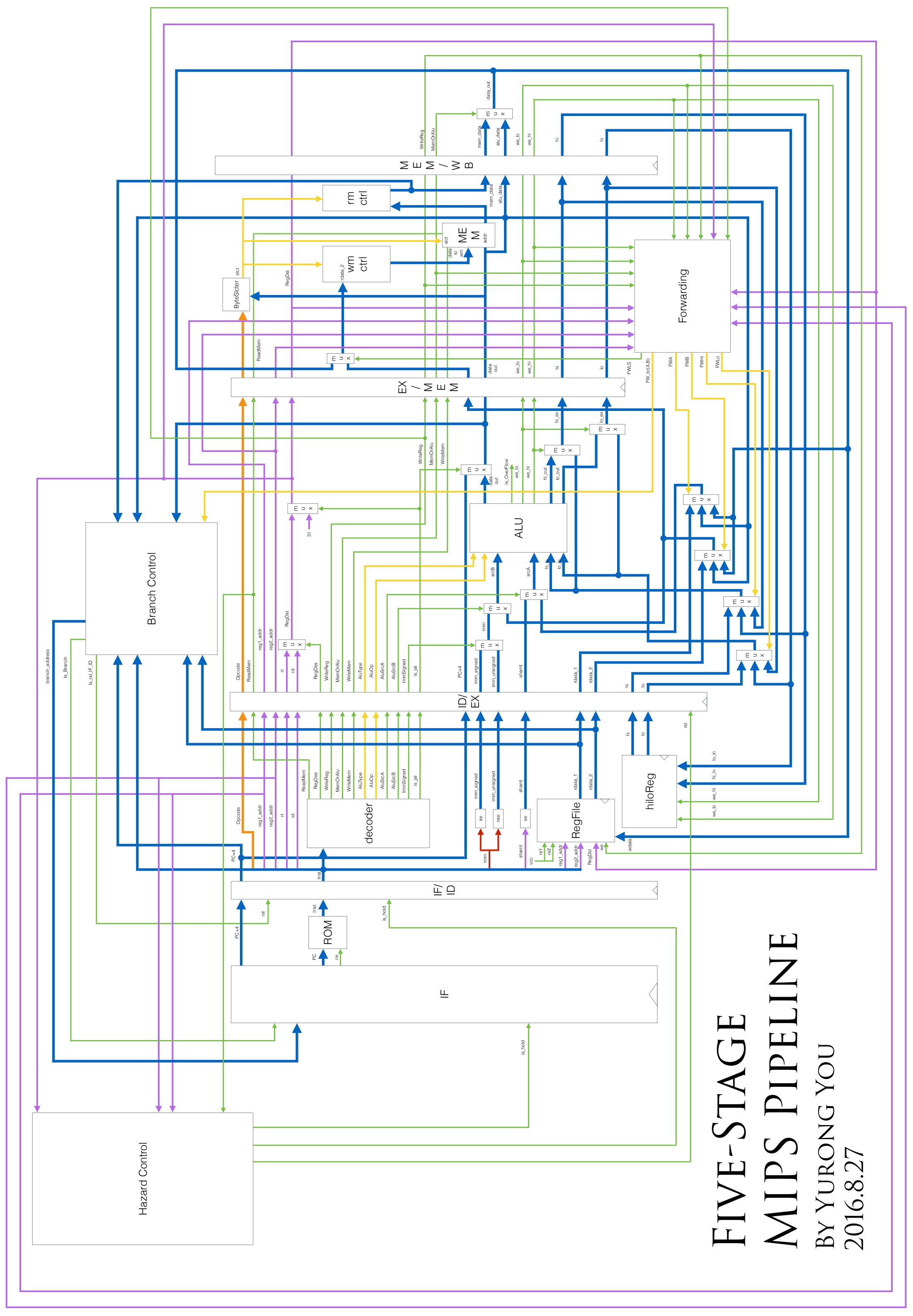
Appendices

Appendix A Instruction Summary

OpType	Type	Syntax	Binary	Remark
Arithmetic	R	`add $d,$s,$t`	`000000 sssss ttttt ddddd 00000 100000`	
	R	`addu $d,$s,$t`	`000000 sssss ttttt ddddd 00000 100001`	
	R	`sub $d,$s,$t`	`000000 sssss ttttt ddddd 00000 100010`	
	R	`subu $d,$s,$t`	`000000 sssss ttttt ddddd 00000 100011`	
	I	`addi $t,$s,C`	`001000 sssss ttttt CCCCC CCCCC CCCCCC`	
	I	`addiu $t,$s,C`	`001001 sssss ttttt CCCCC CCCCC CCCCCC`	
	R	`mult $s,$t`	`000000 sssss ttttt ddddd 00000 011000`	`(HI,LO) = (64-bit) $s * $t`
	R	`multu $s,$t`	`000000 sssss ttttt ddddd 00000 011001`	`(HI,LO) = (64-bit) $s * $t`
	R	`div $s,$t`	`000000 sssss ttttt ddddd 00000 011010`	`LO = $s / $t, HI = $s % $t`
	R	`divu $s,$t`	`000000 sssss ttttt ddddd 00000 011011`	`LO = $s / $t, HI = $s % $t`
Logical	R	`and $d,$s,$t`	`000000 sssss ttttt ddddd 00000 100100`	
	I	`andi $t,$s,C`	`001100 sssss ttttt CCCCC CCCCC CCCCCC`	
	R	`or $d,$s,$t`	`000000 sssss ttttt ddddd 00000 100101`	
	I	`ori $t,$s,C`	`001101 sssss ttttt CCCCC CCCCC CCCCCC`	
	R	`xor $d,$s,$t`	`000000 sssss ttttt ddddd 00000 100110`	
	R	`nor $d,$s,$t`	`000000 sssss ttttt ddddd 00000 100111`	
	R	`slt $d,$s,$t`	`000000 sssss ttttt ddddd 00000 101010`	
	R	`sltu $d,$s,$t`	`000000 sssss ttttt ddddd 00000 101011`	
	I	`slti $t,$s,C`	`001010 sssss ttttt CCCCC CCCCC CCCCCC`	
Bitwise Shift	R	`sll $d,$t,shamt`	`000000 sssss ttttt ddddd 00000 000000`	`$d = $t << shamt`
	R	`srl $d,$t,shamt`	`000000 sssss ttttt ddddd 00000 000010`	`$d = {16’b0, $t >> shamt}`
	R	`sra $d,$t,shamt`	`000000 sssss ttttt ddddd 00000 000011`	`$d = {{16{t[31]}}, $t >> shamt}`
	R	`sllv $d,$t,$s`	`000000 sssss ttttt ddddd 00000 000100`	`$d = $t << $s`
	R	`srlv $d,$t,$s`	`000000 sssss ttttt ddddd 00000 000110`	`$d = {16’b0, $t >> $s}`
	R	`srav $d,$t,$s`	`000000 sssss ttttt ddddd 00000 000111`	`$d = {{16{t[31]}}, $t >> $s}`
Data Transfer	I	`lw $t,C($s)`	`100011 sssss ttttt CCCCC CCCCC CCCCCC`	
	I	`lh $t,C($s)`	`100001 sssss ttttt CCCCC CCCCC CCCCCC`	
	I	`lhu $t,C($s)`	`100101 sssss ttttt CCCCC CCCCC CCCCCC`	
	I	`lb $t,C($s)`	`100000 sssss ttttt CCCCC CCCCC CCCCCC`	
	I	`lbu $t,C($s)`	`100100 sssss ttttt CCCCC CCCCC CCCCCC`	
	I	`sw $t,C($s)`	`101011 sssss ttttt CCCCC CCCCC CCCCCC`	
	I	`sh $t,C($s)`	`101001 sssss ttttt CCCCC CCCCC CCCCCC`	
	I	`sb $t,C($s)`	`101000 sssss ttttt CCCCC CCCCC CCCCCC`	
	I	`lui $t,C`	`001111 00000 ttttt CCCCC CCCCC CCCCCC`	
	R	`mfhi $d`	`000000 00000 00000 ddddd 00000 010000`	
	R	`mflo $d`	`000000 00000 00000 ddddd 00000 010010`	
Conditional branch	I	`beq $s,$t,C`	`000100 sssss ttttt CCCCC CCCCC CCCCCC`	
	I	`bne $s,$t,C`	`000101 sssss ttttt CCCCC CCCCC CCCCCC`	
Unconditiona l jump	J	`j C`	`000010 CCCCC CCCCC CCCCC CCCCC CCCCCC`	`addr = {PC_plus_4[31:28], C << 2}`
	R	`jr $s`	`000000 sssss 00000 00000 00000 001000`	
	J	`jal C`	`000011 CCCCC CCCCC CCCCC CCCCC CCCCCC`	`addr = {PC_plus_4[31:28], C << 2}`

Table A.1: A summary on supported MIPS instructions

Appendix B Pipeline Blueprint


Five-Stage
MIPS Pipeline
By Yurong You
2016.8.27
IF
ROM
IF/ID
decoder
RegFile
hiloReg
ID/EX
ALU
EX/MEM
MEM
wm ctrl
rm ctrl
ByteSlicer
MEM/WB
Forwarding
Branch Control
Hazard Control