**Pipelined MIPS Datapath with Forwarding and Hazards**

# Samuel Lichtenheld

12/12/15

## **TABLE OF CONTENTS**

## Introduction…………………………………………………………………………………………………………………………....3

Instruction Set………………………………………………………………………………………………………………………….4

## Datapaths from Textbook…………………………………………………………………………………………………………5

## Implemented Datapath.................................................................................................................6

Alterations to Datapath…………………………………………………………………………………………………………...7

Forwarding……………………………………………………………………………………………………………………………….8

## ALU Forwarding……………………………………………………………………………………………………………..8

Branch Forwarding………………………………………………………………………………………………………...9

Memory Forwarding………………………………………………………………………………………………………9

Hazard Detections (Stalls)...........................................................................................................10

Branch False-Prediction Data Invalidate Logic............................................................................11

Sample Test Program……………………………………………………………………………………………………………..12

RAM Memory Controller…………………………………………………………………………………………………

### Introduction

This was a project for my Computer Architecture class at the University of Florida. I created a fully-pipelined datapath with hazard-detection and forwarding. My design was based on the MIPS implantation outlined in the textbook Computer Organization and Design, 5th Ed. I wrote everything in VHDL and simulated in ModelSim-Altera 10.3d.

# **Implemented Instruction Set**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Short** | **Operation** | **Type** | **ALU action** |
| Add | add | R[rd] = R[rs] + R[rt] | R | add |
| add imm. | addi | R[rt] = R[rs] + SignExtImm | I | add |
| add imm. uns. | addiu | R[rt] = R[rs] + ZeroExtImm | I | add |
| add uns. | addu | R[rd] = R[rs] + R[rt] | R | add |
| and | and | R[rd] = R[rs] & R[rt] | R | and |
| and imm. | andi | R[rt] = R[rs] & ZeroExtImm | I | and |
| branch eq | beq | if (R[rs]==R[rt]) PC=PC+4+BranchAddr | I | subtract |
| branch not eq | bne | if (R[rs]!=R[rt]) PC=PC+4+BranchAddr | I | subtract |
| jump | j | PC=JumpAddr | J | add |
| jump and link | jal | R[31]=PC+4;PC=JumpAddr | J | add |
| jump reg | jr | PC=R[rs] | R | - |
| load byte uns. | lbu | R[rt] = {24’b0,M[R[rs]+SignExtImm](7:0)} | I | add |
| load halfword uns. | lhu | R[rt] = {16’b0,M[R[rs]+SignExtImm](15:0)} | I | add |
| load upper imm. | lui | R[rt] = {imm, 16’b0} | I | - |
| load word | lw | R[rt] = M[R[rs]+SignExtImm] | I | add |
| Nor | nor | R[rd] = ~(R[rs] | R[rt]) | R | NOR |
| Or | or | R[rd] = R[rs] | R[rt] | R | OR |
| Or imm. | ori | R[rt] = R[rs] | ZeroExtImm | I | OR |
| set less than | slt | R[rd] = (R[rs]<R[rt]) ? 1:0 | R | slt |
| set less than imm. | slti | R[rt] = (R[rs]<SignExtImm) ? 1:0 | I | slt |
| set less than imm. uns. | sltiu | R[rt] = (R[rs]<SignExtImm) ? 1:0 | I | slt |
| set less than uns. | sltu | R[rd] = (R[rs]<R[rt]) ? 1:0 | R | slt |
| shift left logical | sll | R[rd] = R[rt] << shamt | R | sll |
| shift right logical | srl | R[rd] = R[rt] >>> shamt | R | srl |
| store byte | sb | M[R[rs]+SignExtImm}(7:0) = R[rt](7:0) | I | add |
| store halfword | sh | M[R[rs]+SignExtImm}(15:0) = R[rt](15:0) | I | add |
| store word | sw | M[R[rs]+SignExtImm} = R[rt] | I | add |
| subtract | sub | R[rd] = R[rs] -R[rt] | R | subtract |
| subtract uns. | subu | R[rd] = R[rs]= R[rt] | R | subtract |

## **Datapaths Given in Textbook**

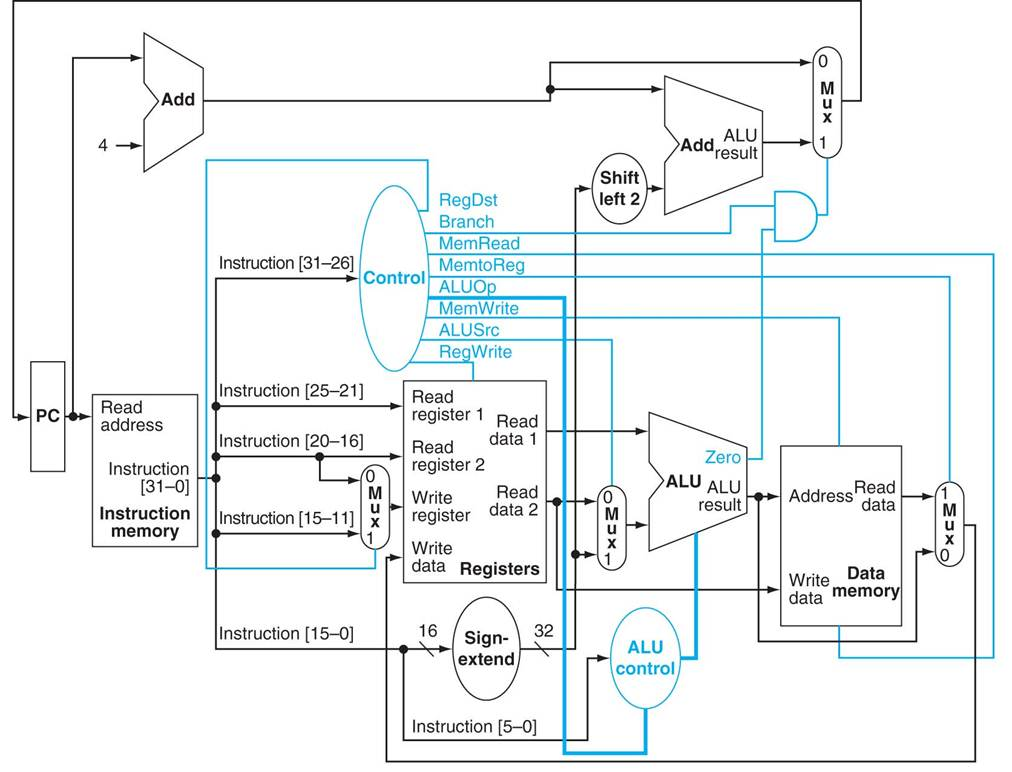


Figure : Single Cycle MIPS datapth

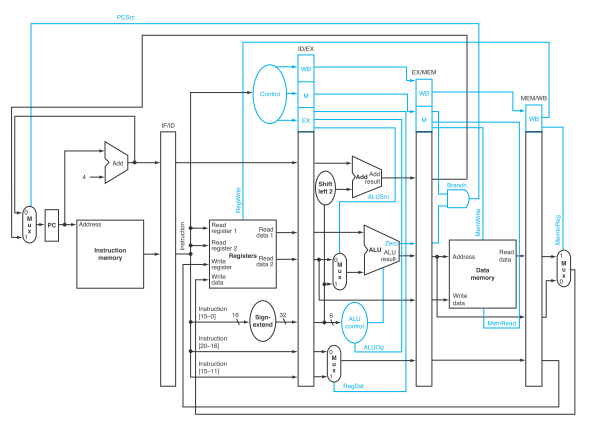


Figure : Pipelined MIPS datapath

**My Datapath**

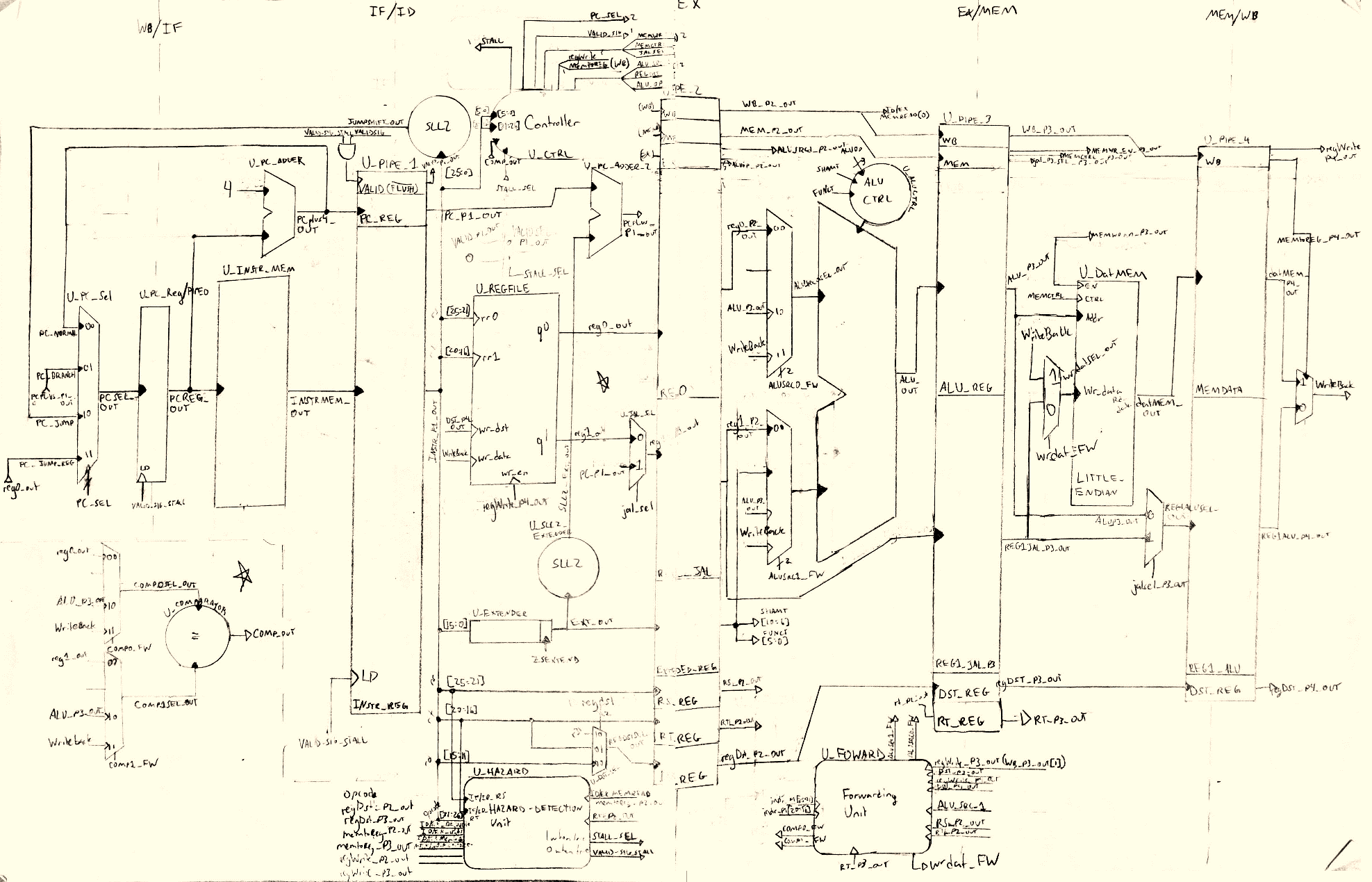


Figure 3: My Datapath Implementation

### My Datapath Alterations Explained

1. Move comparator for branching to ID stage.
   1. If branch is predicted correctly, don’t have to invalidate any data
   2. If prediction is false, only have to invalidate 1 cycle
   3. Requires forwarding from the MEM and WB stages
   4. Requires additional Hazard Detection logic
      1. Must stall if dependent on result of ALU from previous instruction
      2. Must stall if dependent on result of MEM load from previous two instructions
         1. Stall twice if mem load is immediately prior to branch
         2. Stall once if mem load is two instructions prior to branch
2. Move Register Destination Selection ID stage
   1. Necessary for Hazard Detection Logic (for branch conditions described above)
3. Provide Forwarding to the ID and MEM stages
   1. Forwarding to ID stage necessary for branches as described above
   2. Forwarding for MEM stage necessary for writes to data memory
      1. If previous instruction’s result is being stored in Memory, must be forwarded from WB stage as the registers haven’t yet been updated
4. Additional muxes for JAL instruction

### Forwarding Logic

* regWrite signal is passed along each stage for each instruction – it is true if the instruction will be updating the contents of a register
* rs and rt signals are the 5 bit signals that determine which registers are going to be read for the instruction
* regDst is the 5 bit signal that determines which register will be written to for the instruction

**ALU FORWARDING PSEUDOCODE**

if ((EX/MEM.regDst = ID/EX.rs) && (EX/MEM.regWrite = '1'))

*Forward from EX/MEM to ALU input A;*

else if ((MEM/WB.regDst = ID/EX.rs) && (MEM/WB.regWrite = '1'))

*Forward from MEM/WB to ALU input A;*

if ((EX/MEM.regDst = ID/EX.rt) && (EX/MEM.regWrite = '1'))

*Forward from EX/MEM to ALU input B;*

else if ((MEM/WB.regDst = ID/EX.rt) && (MEM/WB.regWrite = '1'))

*Forward from MEM/WB to ALU input B;*

**NOTES:**

* Priority very important in forwarding logic
  + Need to make sure forwarding most recent data
  + Thus, EX/MEM stage gets priority since it is happening after the MEM/WB stage
* RS and RT signals need to be passed from the IF/ID stage to the ID/EX stage for this logic

**BRANCH COMPARATOR FORWARDING PSEUDOCODE**

if ((EX/MEM.regWrite = '1') && (EX/MEM.regDst = IF/ID.rs))

*Forward from EX/MEM to Comparator input A;*

else if ((MEM/WB.regWrite = '1') && (MEM/WB = IF/ID.rs))

*Forward from MEM/WB to Comparator input A;*

if ((EX/MEM.regWrite = '1') && (EX/MEM.regDst = IF/ID.rt))

*Forward from EX/MEM to Comparator input B;*

else if ((MEM/WB.regWrite = '1') && (MEM/WB = IF/ID.rt))

*Forward from MEM/WB to Comparator input B;*

**NOTES:**

* Once again, priority important
* Comparator in ID stage, hence why checking RT and RS signals from the IF/ID stage

**MEMORY FORWARDING PSEUDOCODE**

if ((MEM/WB.regWrite = '1') && (EX/MEM.rt = MEM/WB.regDst))

*Forward from MEM/WB to EX/MEM for data memory input;*

**NOTES:**

* This only checks if the previous instruction is updating a register whose contents are now being written to memory
* Need to pass RT signal from the ID/EX stage to the EX/MEM stage
  + Only need to check the RT signal because the RS signal is used to calculate the address in the ALU – the forwarding would have happened already in the previous stage
  + The RT signal is used to see which register’s content to store, and is not checked for in the previous stage

**Hazard Detection Logic**

* If a load instruction is immediately followed by an instruction which uses this data, it won’t be able to be forwarded until one cycle later, and thus a stall is needed

if (ID/EX.load = '1') {

if ((ID/EX.rt = IF/ID.rt) || (ID/EX.rt = IF/ID.rs))

*STALL;*

}

* If when the instruction is decoded it is found to be a branch instruction, certain conditions are checked
  + If the previous instruction is writing to the registers, or if one of the previous two instructions were loads, and if in any of these cases the registers being updated are used for the branch compare -> stall

if ((IF/ID.instr = beq) || (IF/ID.instr = bne)) {

if ((ID/EX.regWrite = '1') && ( (ID/EX.regDst = IF/ID.rs) || (ID/EX.regDst = IF/ID.rt)))

*STALL;*

else if ((ID/EX.load = '1') && ((ID/EX.regDst = IF/ID.rs) || (ID/EX.regDst = IF/ID.rt)))

*STALL;*

else if ((EX/MEM.load = '1') and ((EX/MEM.regDst = IF/ID.rs) or (EX/MEM.regDst = IF/ID.rt)))

*STALL;*

}

**NOTES:**

* Priority is not important for hazard detection, all condition results would be ORed together

**Branch False-Prediction Data Invalidate Logic**

* If the CPU branches and predicts wrong, the instruction loaded afterwards would be invalid data. If the regWrite and memWrite control signals for this instruction are set to false, however, the instruction can still propogate through the pipeline while not making any unwanted changes to data.
* My datapath will always predict branch equals false; thus if branch = true, the following instructions data will need to be invalidated

if ((IF/ID.instr = beq) && (comparator\_out = '1')

*invalidate next loaded instructions data*

*calculate new PC value based on branch instruction*

if ((IF/ID.instr = bne) && (comparator\_out = '0')

*invalidate next loaded instructions data*

*calculate new PC value based on branch instruction*

**SAMPLE PROGRAM ASSEMBLY CODE**

00: lui $5, 0x0302

01: ori $5, $5, 0x0100

02: ori $6, $0, 0x0908

03: lui $29, 0x1000

04: sw $5, 0($29)

05: sh $6, 4($29)

06: ori $6, $0, 0x0b0a

07: sh $6, 6($29)

08: lui $30, 0x1000

09: ori $30, $30, 0x0008

0a: add $15, $30, $0

BRANCH\_3:

0b: lbu $18, 0($29)

0c: lbu $17, 4($29)

0d: add $16, $17, $18

0e: slti $10, $16, 0x000b

0f: beq $10, $0, 0x0002

10: add $17, $16, $18

11: j 0x0013 (JUMP\_1)

BRANCH\_1:

12: sub $17, $16, $18

JUMP\_1:

13: slti $10, $17, 0x000b

14: beq $10, $0, 0x0002 (BRANCH\_2)

15: add $18, $17, $16

16: j 0x0018 (JUMP\_2)

BRANCH\_2:

17: sub $18, $16, $17

JUMP\_2:

18: sb $18, 0($30) --0x10 @0x08

19: addi $30, $30, 0x0001

1a: addi $29, $29, 0x0001

1b: bne $29, $15, 0xffef (BRANCH\_3)

1c: ori $19, $0, 0x0008

1d: subu $29, $29, $19

1e: lw $1, 0($29)

1f: lw $2, 4($29)

20: lw $3, 8($29)

21: lw $4, 12($29)

END:

22: j 0x0022 (END)