Project 2: Design Documentation

Yogisha Dixit (yad4) Nicholas De Tullio (nrd24)

March 27, 2014

1 Introduction

This document is intended to describe the planned implementation of a MIPS pipeline processor with a simplified instruction set. There is information on design, testing, and the correctness constraints of each subcomponent and the overall circuit included. The intended audience is anyone familiar with the assignment and who has a level of understand comparable to the information learned in CS3410 at Cornell University. References include *Computer Organization and Design* by David Patterson and John Hennessy, *MIPS32 Architecture for Programmers*, Volumes 1,2, and 3 by MIPS Technologies, and slides and lectures from CS3410 at Cornell University, Spring 2013. This document contains information on the implementation of a mini MIPS pipeline processor, including diagrams, control logic, correctness constraints, and information on testing the processor.

Common abbreviations:
Mux: Multiplexer
PC: Program Counter
ALU: Arithmetic Logic Unit

2 Overview

We implement the pipeline processor by splitting it into 5 stages: Fetch, Decode, Execute, Memory, and Write Back. The Memory stage and Jump/Branch instructions are not going to be implemented during this stage of the project. The Fetch stage will get the instruction and update PC as necessary. The Decode stage will determine what action we will perform with the instruction, split it into different input values as appropriate, and pass relevent information onto the next stage. The execute stage will perform any non-Memory operations and pass the information on. The memory stage will, in the future, allow us to use instructions that require access to memory. For now, it is a dummy stage where information is just passed through unaltered. The Write Back stage will determine what we need to update in the register, if anything, and send the relevent register and output information back. Here is the overall circuit diagram. We provide more detail in each stage.

3 Transition Register Stages

There are four different substages that store the information between stages: ID/DE, DE/EX, EX/MEM, MEM/WB. They store the values in registers or in a d flip-flop if it is only 1 bit. Both kinds of memory storage devices use the rising edge as their trigger value. We include the diagram for only the EX/MEM stage in Figure 1, as the others are identical except for the values they store.

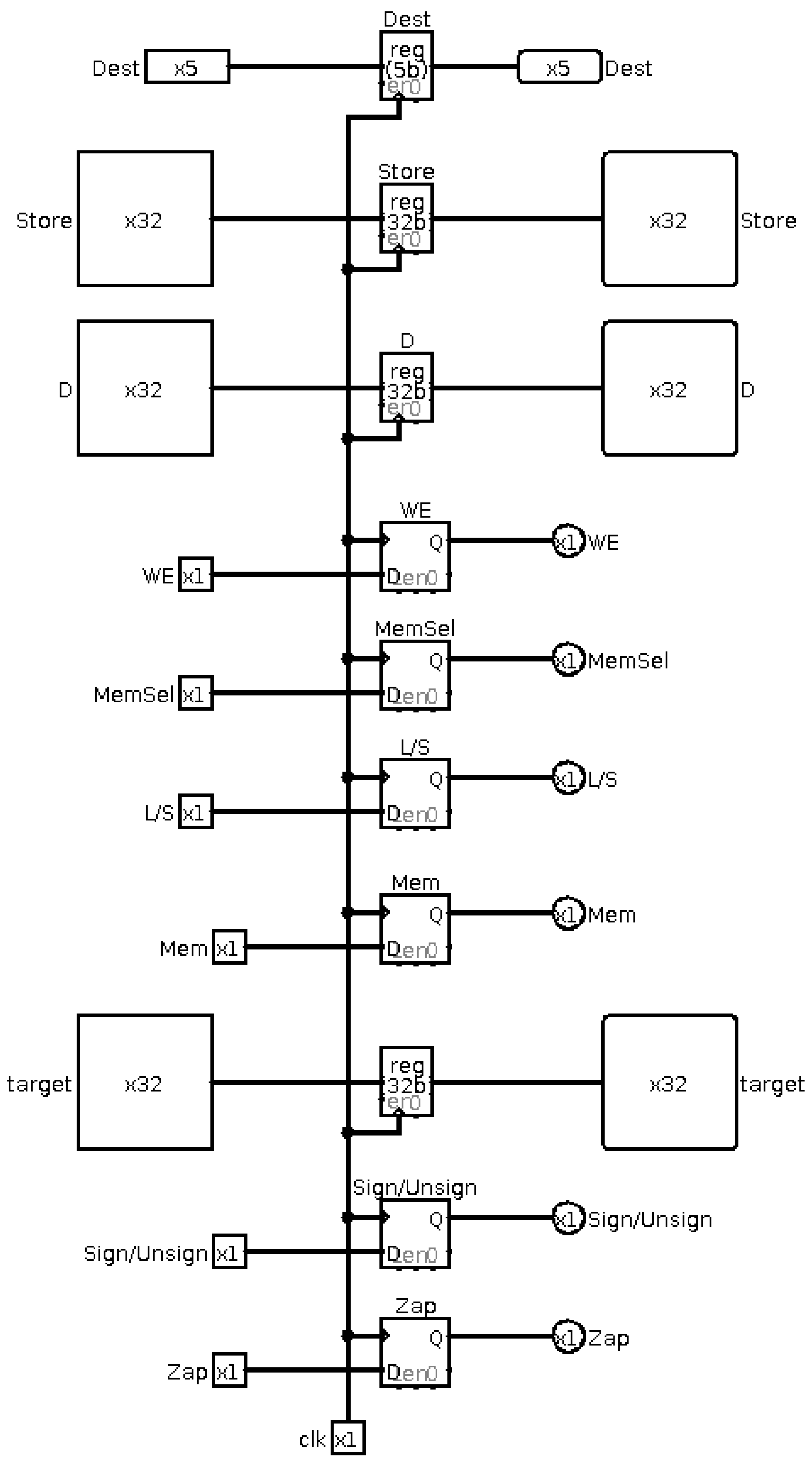


Figure 1: Ex/Mem Transition Register Stage

4 Circuit Diagram

5 The Fetch Stage

Inputs: PCNew
Outputs: ROMIn[32],PC+4[32]

The following defines what each of the input signals mean:

Is JI signals whether an immediate jump should take place
Is JR signals whether a register jump should take place
JI the new PC if an immediate jump is carried out
JR the new PC if a register jump is carried out
BranchTarget the new PC if a branch is carried out
IsBranch signals whether a branch should take place
NOP signals whether a NOP was inserted into the pipeline and thus, the PC should be prevented from incrementing

The following defines what each of the output signals mean:

ROMIn the address of the instruction to be fetched from the ROM
PC+4 the incremented value of the PC

The Fetch stage use the current value of PC to retrieve the instruction from the program ROM and updates the program counter. The logic that determines what the next program counter value should be depends on whether jumps/branches are being carried out and whether a NOP has been inserted into the pipeline. The truth table for this logic is shown in the truth table below.

Is JI	Is JR	Is Branch	NOP	New PC
0	0	0	0	PC+4
1	0	0	0	PC+4_{31-28}|JI_{25-0}|0^2
1	0	0	0	PC
0	1	0	0	PC+4
0	1	0	1	PC
0	0	1	0	PC+4
0	0	1	1	PC

The circuit for this stage is shown in Figure 2.

5.1 Submodule: PC+4

This submodule increments the program counter by 4. It uses the incrementer provided in the CS3410 jar file. Since the provided incrementer only increments the value it is provided by 1, we split the 32-bit input into the 30 most significant bits and the 2 least significant bits. We feed the 30 bits into the incrementer and combine the result with the 2 bits in the same order as before. The circuit is shown in Figure 3.

5.2 Correctness Constraints

The Fetch Stage must correctly get the instruction and update PC. If a Jump or Branch is specified, it should execute the instruction in the delay slot and update the program counter to the value specified by the Jump/Branch. It should retrieve the instruction at the index of PC in the ROM.

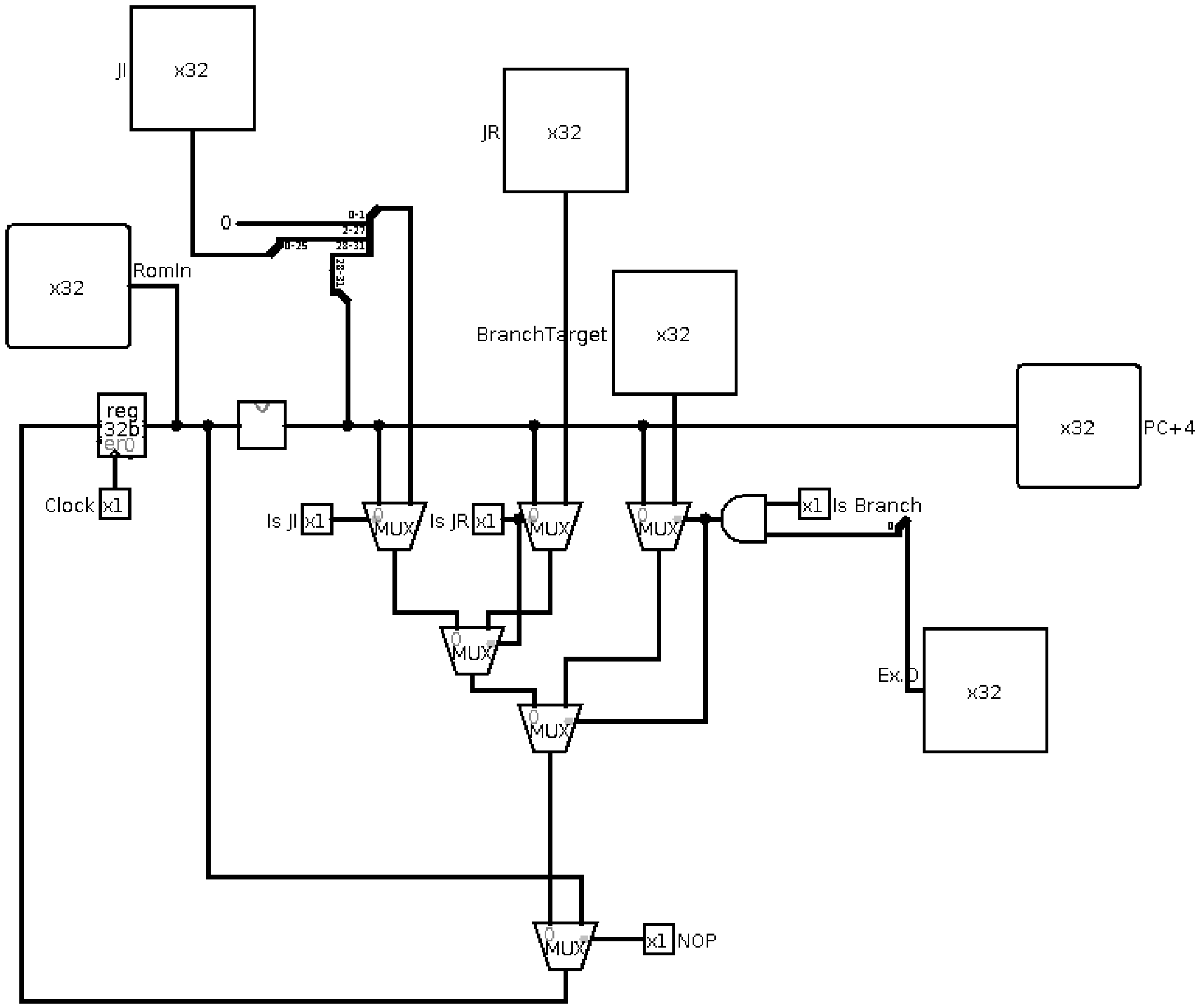


Figure 2: Fetch Stage

6 The Decode Stage

Inputs: Op[32], Zap[1]

Outputs: Is JR[1], WhichSa[1], WriteEn[1], Is JALR[1], LT/GE[1], Rs[5], Rt[5], Rd[5], ALU Op[4], Is R[1], Is Branch[1], Is JAL[1], B or Immediate[1], Immediate[1], WeirdBranch[1], Is JI[1], MemSel[1], Mem[1], Sign/Unsign[1], L/S[1]

The decode stage determines which register addresses we wish to access as well as if we wish to write to the register. This is where much of the control logic necessary for the execute stage is determined as well. It also extends the immediate value of I instructions here as necessary. The following defines what each of the input signals mean:

Op the instruction retrieved from the program ROM at the previous stage
Zap if branch is taken, zap the second instruction following it

The following defines what each of the output signals mean:

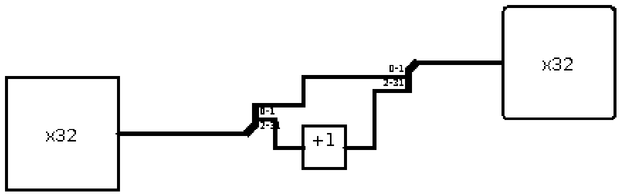


Figure 3: +4 Incrementer

Is JR signals whether Op is a R-type jump instruction
WhichSa signals whether the shift amount to use is the one provided or in a register
WriteEn signals whether we should enable writing to the Rd register
Is JALR signals whether Op is a JALR instruction
LT/GE signals whether Op is a $<$ or $\geq$ branch command
Rs the first adress to be read from the register file
Rt the second address to be read from the register file
Rd the address to which the output should be written
ALU Op the op code to feed into the ALU
Is R signals whether Op is a R-type instruction
Is Branch signals whether Op is a branch instruction
Is JAL signals whether Op is a JAL instruction
B or Immediate signals whether to use the B value or the Immediate value as input to the ALU
WeirdBranch signals whether Op is a BLTZ or BGEZ command
Is JI signals whether Op is a I-type jump instruction
MemSel signals whether a memory instruction requires us to load/store an entire word or only a byte
Mem signals whether Op is a memory instruction
Sign/Unsign signals whether Op is a signed or unsigned memory load instruction
L/S signals whether Op is a load or store memory operation

A register is used to store the previous Op code. In case of a NOP is inserted into the pipeline, this previous instruction is retrieved and decoded. Furthermore, if the instruction is a JAL command, then the decoded value for Rd is scrapped and replaced with 31 instead. The rest of the calculations are done in the two sub-modules. The top-level decode circuit is shown in Figure 4.

6.1 Submodule: Immediate Decode

Inputs: Op[32]
Outputs: Immediate[32], B or Immediate[1]
This submodule determines how we should extend the immediate value as well as if we should use the B or the Immediate value in the execute stage. The circuit is shown in Figure 5.

6.1.1 Extend Mux Logic

We only care about how we extend Immediate when we have I type instructions, as they are the only ones to use this value. The value that we choose to extend by depends on the specification of the instruction. We present a table with the significant values below, all other values are treated as don't cares. The Mux chooses zero extend if 1 and sign extend if 0. Each instruction is an I-type presented by its Op value (it's first 5 most significant bits).

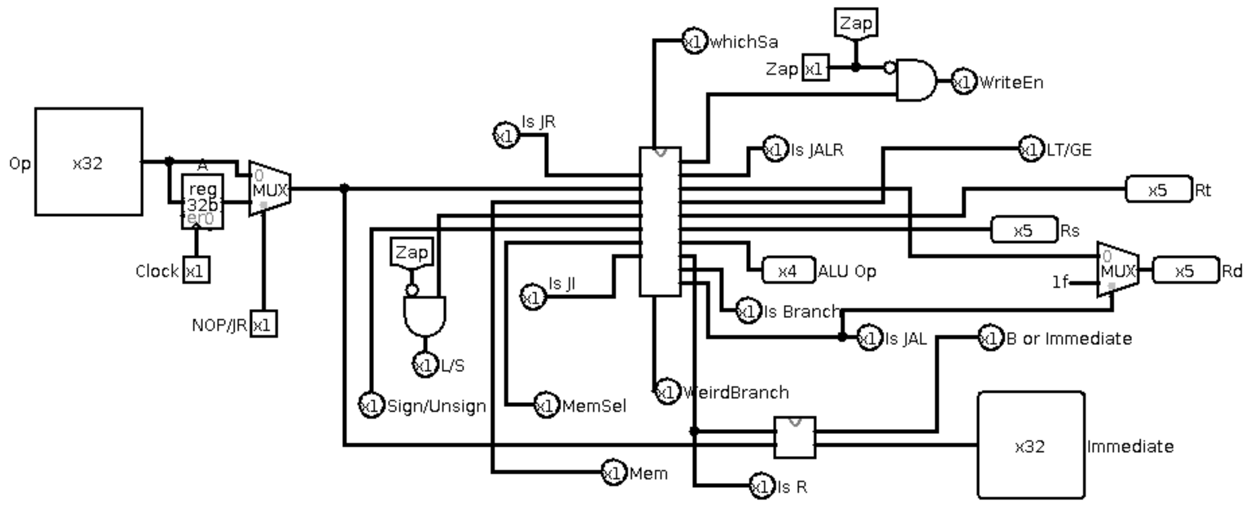


Figure 4: Top level decode stage

Op5	Op4	Op3	Op2	Op1	Op0	Mux
0	0	1	0	0	1	0
0	0	1	1	0	0	1
0	0	1	1	0	1	1
0	0	1	0	1	0	0
0	0	1	0	1	1	0
0	0	1	1	1	0	1
0	0	1	1	1	1	0
0	0	0	1	0	0	0
0	0	0	1	0	1	0
0	0	0	1	1	0	0
0	0	0	0	0	1	0
0	0	0	1	1	1	0
0	0	0	0	0	1	0
0	0	0	0	0	1	0
1	0	0	0	1	1	0
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	1	0
1	0	1	0	0	0	0

So the logic equation is $mux = Op_4 Op_4$.

6.1.2 B or Immediate logic

B or Immediate is simply a marker for if we should use the B value of the Rt register or the immediate value of the instruction, as specified by the instruction itself. We encode B or Immediate as follows: B is 1 and Immediate is 0.

The following truth table indicates what this value should be set to based on the instruction. Note that all R-types use the B value, while most, but not all, I-types use the immediate value. As a result, the truth table only contains op codes for immediate instructions.

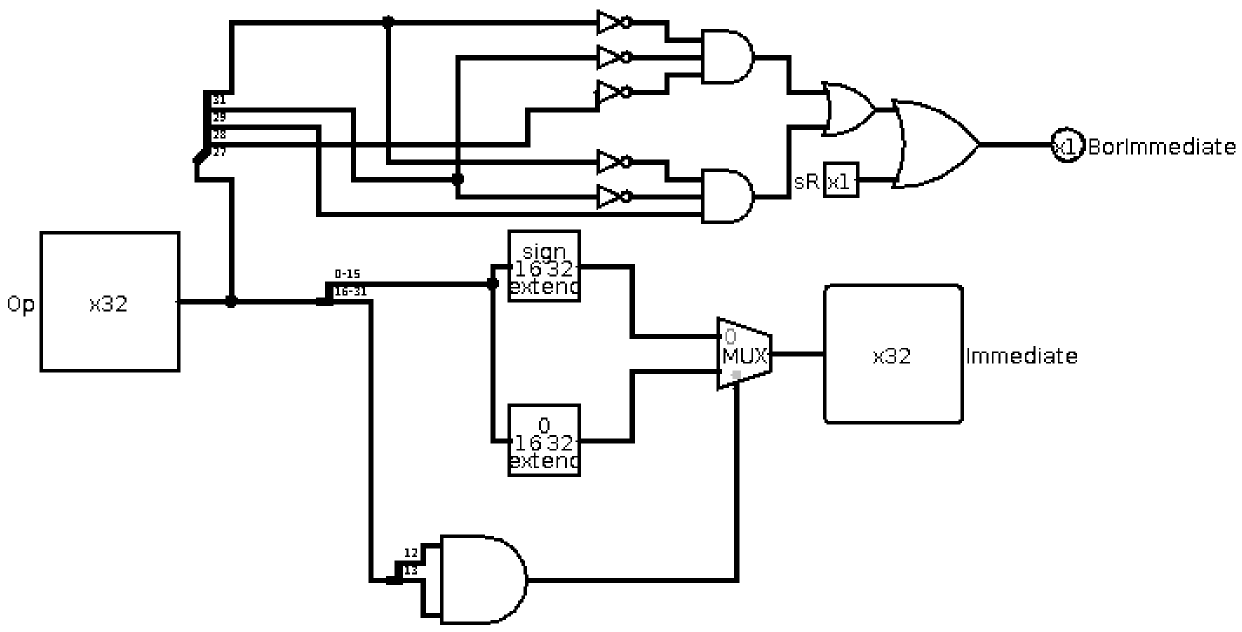


Figure 5: Immediate decode stage

Op5	Op4	Op3	Op2	Op1	Op0	B or Immediate
0	0	1	0	0	1	0
0	0	1	1	0	0	0
0	0	1	1	0	1	0
0	0	1	0	1	0	0
0	0	1	0	1	1	0
0	0	1	1	1	0	0
0	0	1	1	1	1	0
0	0	0	1	0	0	1
0	0	0	1	0	1	1
0	0	0	1	1	0	1
0	0	0	0	0	1	1
0	0	0	1	1	1	1
1	0	0	0	1	1	0
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	1	0
1	0	1	0	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	0	1
0	0	0	0	1	1	0

Since we have th input value isR to tell us if an instruction is R-type or not (this is calculated in the non immediate decode stage that will be explained below), we can just look at the I-type instructions that use B instead of immediate to finish the logic. Then the logic equation is then $BorImmediate = IsR + \bar{(}Op_5Op_3)$.

6.2 Submodule: Non Immediate Decode

Non Immediate Decode deals with the logic that isn't related to the immediate value. It parses the register locations from the instruction, which is done with a splitters.

6.2.1 Control for ALU

It also determines the control value for the ALU. This is done with two ROMS – one for R-type instructions and one for I-type instructions. The R-instruction ROM is addressed by $Func_5, Func_3, Func_2, Func_1, Func_0$, where Func is the function code of the R-type instructions, as this determines them. We noticed $Func_4$ is always 0 in our instruction set, and so we don't need it. The I-type ROM is the same, except it uses the Op part of the instruction.

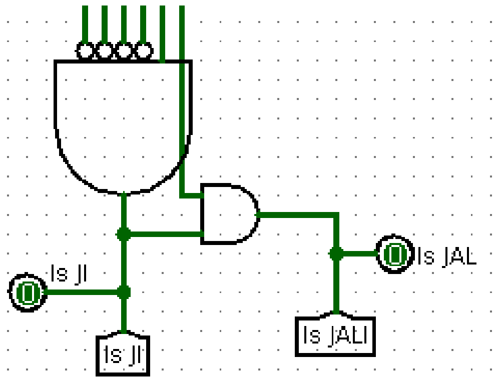
The mux to control that chooses between these ROMs is based on whether or not the instruction is an R-type. Note that isR=$Op5Op4Op\bar{3}Op2Op1Op0$ since all of our R type instructions start with all 0's in their Op code.

6.2.2 Jump Instructions

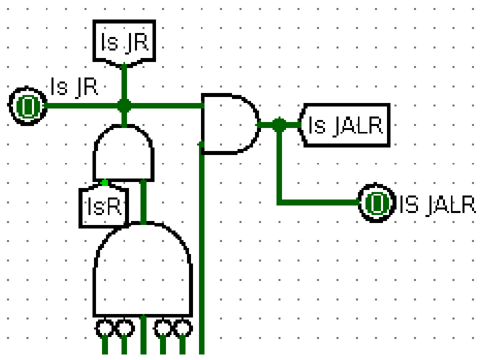
There are two parts to decoding jump instructions. Based on whether they are I-type instructions or R-type instructions, we utilize the Op code or the function code in the 32 bit instruction. As a result, the instruction is a register jump if $Func\bar{5}Func4Func3Func\bar{2}Func1$. On the other hand, it is an immediate jump if $Op5Op\bar{4}Op3Op2Op1$. We also care about if the instruction requires us to link to a return address. The instruction is a JALR instruction if $IsJR\&Func_0$. The instruction is a JAL isntruction if $IsJI\&Op_0$. Both parts are shown in Figure 6

6.2.3 Memory Instructions

There are several steps to decoding memory instructions. First, we need a signal to determine whether Op is a memory instruction. By looking at the table of Op codes below, we can determine that Mem = Op_5. Further, we need to be able to tell if the instruction is a load or store operation. Again, by looking at the table of Op codes below, we can tell that L/S = Op_3. We set L/S to 1 when Op is a store operation and 0 when it is a load operation. Lastly, we need to be able to tell if a load operation should be signed or unsigned. Sign/Unsign = Op_2 and therefore equals 0 when loading a signed byte and 1 when loading an unsigned byte.



(a) Immediate Jump Decode



(b) Register Jump Decode

Figure 6: Jump Decoding

Op5	Op4	Op3	Op2	Op1	Op0	Memory
0	0	1	0	0	1	0
0	0	1	1	0	0	
0	0	1	1	0	1	0
0	0	1	0	1	0	0
0	0	1	0	1	1	0
0	0	1	1	1	0	0
0	0	1	1	1	1	0
0	0	0	1	0	0	1
0	0	0	1	0	1	1
0	0	0	1	1	0	1
0	0	0	0	0	1	1
0	0	0	1	1	1	0
0	0	0	0	0	1	0
0	0	0	0	0	1	0
1	0	0	0	1	1	1
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	1	1
1	0	1	0	0	0	1

6.2.4 WhichSa

WhichSa is based on the fact that sometimes we need the value in a register to use as the shift amount for certain commands. The truth table for these commands is presented below, based on their function code, as only R-types are shift instructions. We set WhichSa to 0 when we use the immediate Sa value and 1 if we have to grab it from a register.

Func5	Func4	Func3	Func2	Func1	Func0	WhichSa
0	0	0	0	0	0	0
0	0	0	0	1	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	1
0	0	0	1	1	0	1
0	0	0	1	1	1	1

As a result, WhichSa=$func2$.

6.2.5 Write Enable

This value determines if Write Enable is activated or not. Writing to the register file is disabled for non jump and link jump commands, branch commands, and store to memory commands. As a result, we set Enable to 0 when we get any of the above instructions and 1 otherwise. A circuit diagram is given in Figure 7

The WriteEn value for every instruction is given in the table below.

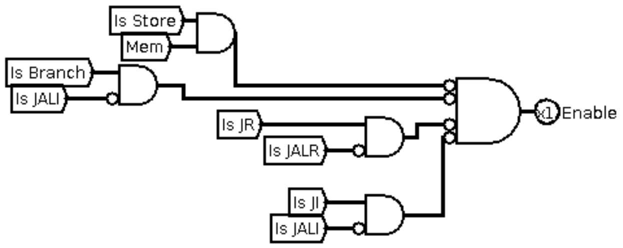


Figure 7: Write Enable Circuit

Op5	Op4	Op3	Op2	Op1	Op0		func5	func4	func3	func2	func1	func0	WriteEn
0	0	0	0	0	0	. . .	1	0	0	0	0	1	1
0	0	0	0	0	0	. . .	1	0	0	0	1	1	1
0	0	0	0	0	0	. . .	1	0	0	1	0	0	1
0	0	0	0	0	0	. . .	1	0	0	1	0	1	1
0	0	0	0	0	0	. . .	1	0	0	1	1	0	1
0	0	0	0	0	0	. . .	1	0	0	1	1	1	1
0	0	0	0	0	0	. . .	1	0	1	0	1	0	1
0	0	0	0	0	0	. . .	1	0	1	0	1	1	1
0	0	0	0	0	0	. . .	0	0	1	0	1	1	1
0	0	0	0	0	0	. . .	0	0	1	0	1	0	1
0	0	0	0	0	0	. . .	0	0	0	0	0	0	1
0	0	0	0	0	0	. . .	0	0	0	0	1	0	1
0	0	0	0	0	0	. . .	0	0	0	0	1	1	1
0	0	0	0	0	0	. . .	0	0	0	1	0	0	1
0	0	0	0	0	0	. . .	0	0	0	1	1	0	1
0	0	0	0	0	0	. . .	0	0	0	1	1	1	1
0	0	0	0	0	0	. . .	0	0	1	0	0	0	0
0	0	0	0	0	0	. . .	0	0	1	0	0	1	0
0	0	1	0	0	1	. . .	x	x	x	x	x	x	1
0	0	1	1	0	0	. . .	x	x	x	x	x	x	1
0	0	1	1	0	1	. . .	x	x	x	x	x	x	1
0	0	1	0	1	0	. . .	x	x	x	x	x	x	1
0	0	1	0	1	1	. . .	x	x	x	x	x	x	1
0	0	1	1	1	0	. . .	x	x	x	x	x	x	1
0	0	1	1	1	1	. . .	x	x	x	x	x	x	1
0	0	0	1	0	0	. . .	x	x	x	x	x	x	0
0	0	0	1	0	1	. . .	x	x	x	x	x	x	0
0	0	0	1	1	0	. . .	x	x	x	x	x	x	0
0	0	0	0	0	1	. . .	x	x	x	x	x	x	0
1	0	0	0	1	1	. . .	x	x	x	x	x	x	0
1	0	0	0	0	0	. . .	x	x	x	x	x	x	0
1	0	0	1	0	0	. . .	x	x	x	x	x	x	0
1	0	1	0	1	1	. . .	x	x	x	x	x	x	0
1	0	1	0	0	0	. . .	x	x	x	x	x	x	0
0	0	0	0	1	0	. . .	x	x	x	x	x	x	0
0	0	0	0	1	1	. . .	x	x	x	x	x	x	0

6.3 Correctness Constraints

The Decode stage must correctly interpret the instruction into its relevent values and operations. This includes finding the correct vales for Rb, Ra, Rb and write enable in the register and correctly extending the immediate value of an I type instrution. It should also correctly determine important decode variables used in the execute stage, such as whether the instruction is an R-type and what we should use for the shift amount.

7 ID/EX Stage

Inputs: A[32], B[32], Ex.D[32], WB.D[32], Ex.Rd[5], M.Rd[5], WB.Rd[5], whichSA[1], WE[1], Rs[5], Rt[5], Op[4], Rd[5], MemSel[1], B or Immediate[1], Imm[32], isR[1], Mem[1], L/S[1], Sign/Unsign[1], PC+4[32], IS JAL[1], IS JALR[1], LT/GE[1], Is Branch[1], WeirdBranch[1]

Outputs: A[32], B[32], whichSA[1], WE[1], Rs[5], Rt[5], Op[4], Rd[5], MemSel[1], B or Imm[1], Imm[32], isR[1], Mem[1], L/S[1], Sign/Unsign[1], target[32], Is JAL[1], Is JALR[1], LT/GE[1], Is Branch[1], WeirdBranch[1], NOP[1], NOP/JR[1]

A the A value passed in by the instruction
B the B value passed in by the instruction
WB.D the value to be written
Ex.Rd the address to which the output should be written (pulled from Execute)
M.Rd the address to which the output should be written (pulled from Memory)
WB.Rd the address to which the output should be written (pulled from Write Back)
whichSA signals whether the shift amount to be used is the one provided on in a register
WE signals whether we should enable writing to the Rd register
Rs the first address to be read from the register file
Rt the second address to be read from the register file
Op the op code to feed into the ALU
Rd the address to which the output should be written
MemSel signals whether a memory instruction requires us to load/store an entire word or only a byte
B or Immediate signals whether to use the B value or the immediate value as input to the ALU
Imm the immediate value passed in by the instruction
isR signals whether or not it is an R-Type instruction
Mem signals whether Op is a memory instruction
L/S signals whether Op is a load or a store memory instruction
Sign/Unsign signals whether Op is a signed or unsigned instruction
PC+4 the value of the program counter incremented by 4
target the PC value (+4 if NOP/JR —— Is Branch) (+8 otherwise)
Is JAL signals whether Op is a JAL instruction
Is JALR signals whether Op is a JALR instruction
LT/GE signals whether Op is a $<$ or $\geq$ branch command
Is Branch signals whether Op is a branch instruction
WeirdBranch indicates whether Op is a BLTZ or BGEZ command
NOP a no-op for internal logic
NOP/JR passed into Decode to help with reading the same instruction again

Unlike the other transition stages which contain only registers with values passing through, this particular stage contains some important control logic. This is where forwarding for jumps occurs and where NOPs, if necessary, are inserted into the pipeline.

Inserting a NOP essentially involves transforming the current operation into a SLL \$0, \$0, 0 and preventing the PC from incrementing. Using the following two subcircuits, we generate a signal called NOP (which equals 1 if and only if a NOP needs to be inserted into the pipeline) and use it to select between the current values of all the inputs and values that would transform the input into a SLL \$0, \$0, 0.

7.1 Jump Forwarding

Inputs: Id/Ex.Ra[5], Ex.WE[1], Ex.Rd[5], M.WE[1], M.Rd[5], WB.WE[1], WB.Rd[5], Is JR[1]
Outputs: Forward[2], NOP[1]

The jump forwarding logic, shown in Figure 8, acts similarly to the forwarding logic that takes place in the execute stage. If the destination register of any of the previous commands matches the register in which

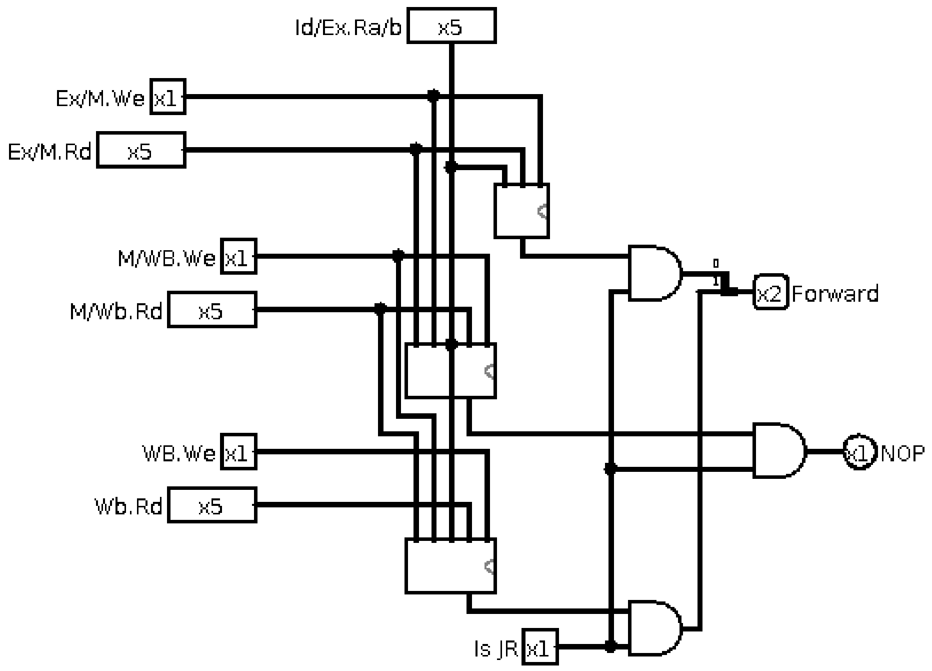


Figure 8: Jump Forwarding Logic

the jump target is stored and writing is enabled to that destination register, the value is forwarded. The latest value to be stored in that register is always forwarded. The Forward output consists of two bits. The truth table below illustrates which value is chosen depending on the Forward bits.

Forward_1	Forward_0	Output
0	0	A
0	1	Ex.D
1	0	WB.D
1	1	Ex.D

Lastly, if the jump target value is currently in the memory stage, a NOP is introduced into the pipeline, and the value is read when it reaches the writeback stage. If a NOP is to be introduced, the NOP output is set to 1. Otherwise, it is set to 0.

7.2 Load Use Hazard Detection

A load use hazard occurs when a byte or word to be read from memory is needed in the next instruction, but is not available until the end of the memory stage. In this case, a NOP needs to be inserted into the pipeline.

If the previous instruction was a load from memory instruction, we need to check if its destination register matches any of the registers the current instruction is reading from. Furthermore, we want to ensure that the destination register of the current instruction is not 0. The circuit for this stage is shown in Figure 9.

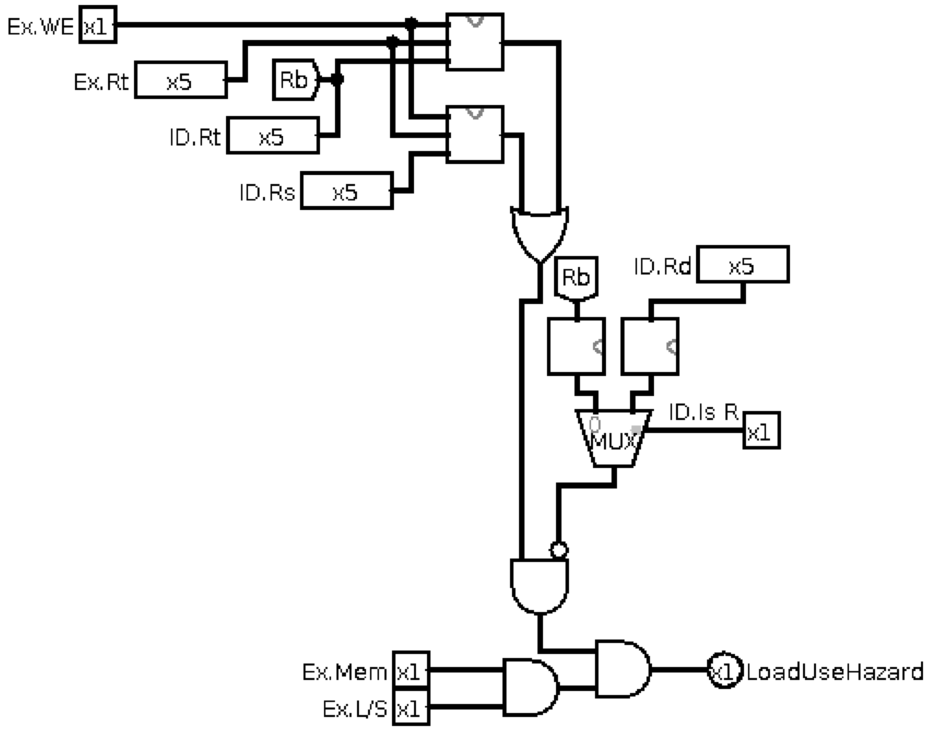


Figure 9: Load Use Hazard Detection Unit

8 The Execute Stage

Inputs: A[32], Immediate[32], B[32], B or Immediate[1], Mem in[1], PC+4[32], WE[1], WhichSa[1]

Outputs: D[32], WE2[1], target[32], Dest[5]

A the A value passed in by the instruction
Immediate the immediate value passed in by the instruction
B the B value passed in by the instruction
B or Immediate signals whether to use the B value or the Immediate value as input to the ALU
Mem in signals whether Op is a memory instruction
PC+4 the value of the program counter incremented by 4
WE/WE2 signals whether we should enable writing to the Rd register
WhichSa signals whether the shift amouunt to use is the one provided or in a register
D the value to be written by the instruction
target the target instruction for the PC
Dest the address to which the output should be written

The execute stage performs any necessary computation for the commands that do not access memory. It contains an ALU to do this, as well as some comparators. A and B are the values at the register locations given by Rs and Rt, respectively, Immediate is 16 least significant bits of the instruction(its immediate field if I-type), B or Immediate determines whether we use B or Immediate in our calculation, Mem in describes whether or not the instruction uses memory, WE defines whether write enable should be turned off from information we parsed in the last stage, WhichSa defines what we use for the shamt in the ALU, and PC+4 is the possible address of the next instruction. D is then the output of the calculation, WE2 is whether or not we should allow writes to the register after doing our calculations in this stage, and Dest is the address of the register we wish to write to. Target is the location of a branch instruction, but it is simply PC+4 in this case since we are not implementing branch instructions.

8.0.1 Rd Mux

This determines what value of Rd we send back to the register. It is the rt section if the command is an I-type, in which case the mux chooses register encoded by the the 21-17 bits, and the rd section of the instruction if it is an r-type or a jump command, in which case the mux chooses the register enoded by 16-12 bits. Thus we OR the previously calcuated value is R with Is JAL, which gives us 1 if Rd should be chosen and 0 if Rt should be chosen. The circuit diagram is shown in Figure 10.

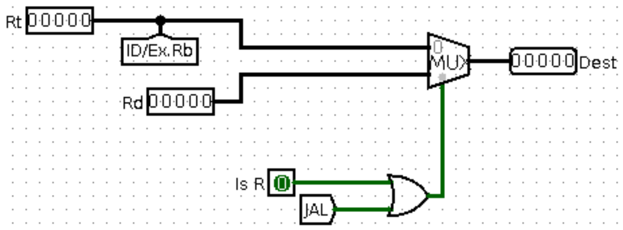


Figure 10: Destination Register Selection Circuit

8.0.2 Immediate or B to ALU Mux

This determines if we send value B from the register or the Immediate value we computed from the instruction to the ALU. It depends on the Opcode of the instruction, in the sense that only I types have an immediate value and for the branch I-type instructions we do not send the immediate value to the ALU. The select bit for the mux is the B or Immediate value we calculated in the decode stage. A circuit diagram is shown in Figure 11.

8.0.3 Shift Ammount Muxes

This mux chooses what value we use for the shift amount. For SLL, SRL, and SRA use the shamt that is encoded in the R-type instructions, while SLLV, SRLV, and SRAV use the 5 least significant bits of the value in the register at address Rs and for some Immediate instructions, we shift the bit extended value by 16. We first choose between the 6-10 bits of the immediate value and 16 based on $\bar{Op}2Op0$ of the ALU op code (the ALU02 tunnel in the circuit diagram). Then, we choose between the value generated by the previous mux, and the 0-4 bits of A, based on WhichSa. We determined this value in the previous stage. The circuit diagram is shown in Figure 12.

8.0.4 ALU Control Logic

This value is determined solely by the ALU Op code we decoded from the previous stage.

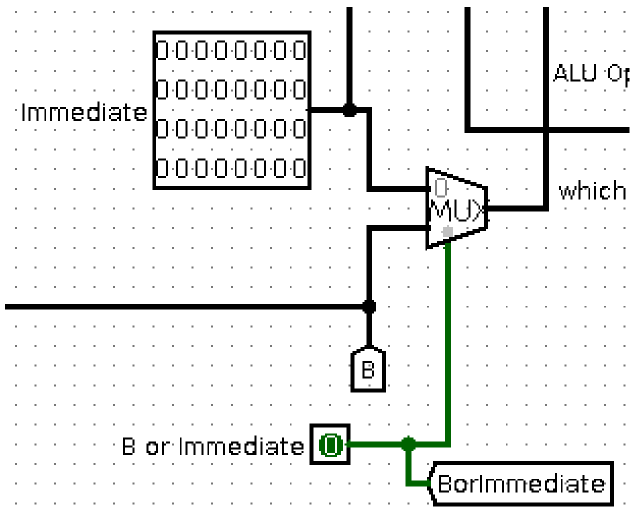


Figure 11: B or Immediate Selection Circuit

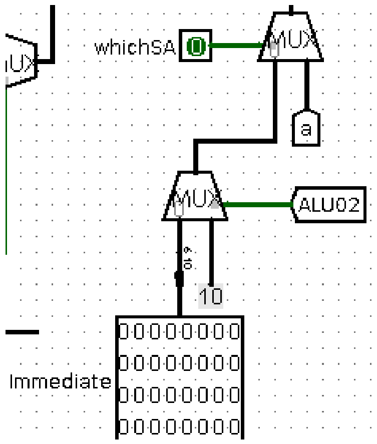


Figure 12: Shift Amount Selection Circuit

8.0.5 Comparators

For the SLTxx commands, we need two comparators – one for unsigned comparison and one for two's complement. We also need logic to choose between these two, the ALU output, and A (for load commands). We use some logic on the ALU Op code to choose between these values. The circuit diagram is shown in Figure 13.

8.0.6 WE2 Control Logic

We partially determined whether nor not we should permit writing to the register in the previous stage. We now account for the cases that result from MOVN and MOVZ, as we only want to write to the register for those commands if B!=0 and B==0, respectively. As a result, we look at whether it is a equal/not equal command, and whether B equals zero. The circuit diagram is shown in Figure 14 .

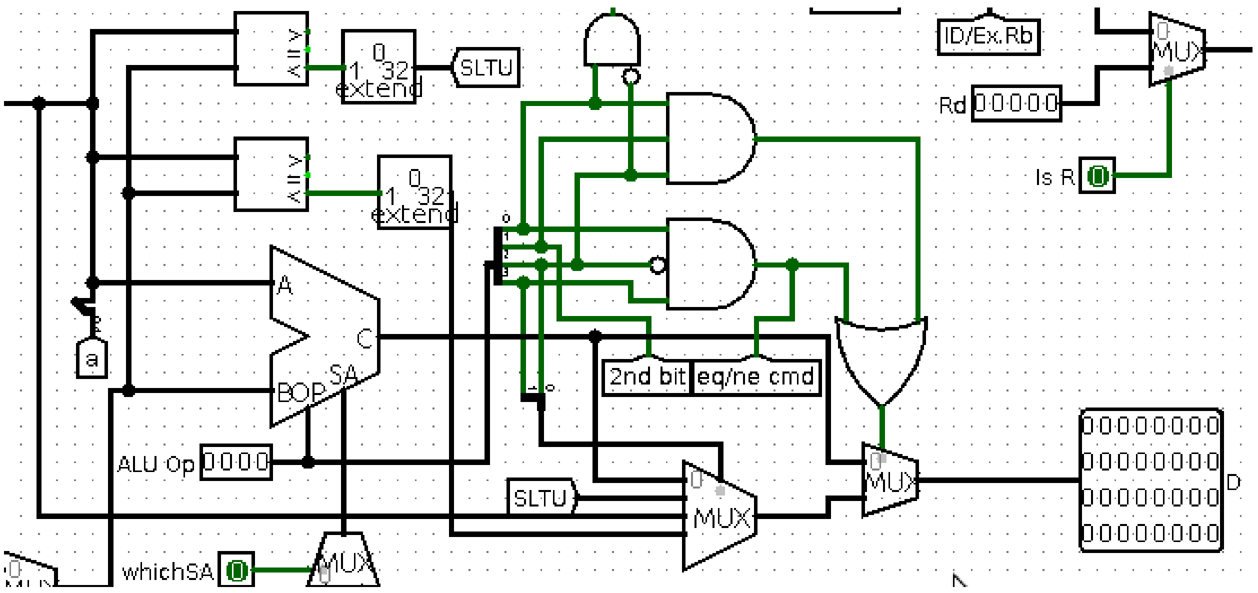


Figure 13: Comparator Logic Circuit

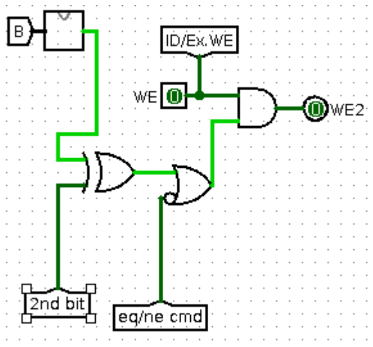


Figure 14: WE2 Control Logic Circuit

8.1 Branch Target Calculation

If a branch is to be taken, then the new PC, hereafter known as the target, requires some calculation. It equals the least 16 bits of the Immediate value with two zeroes concatenated to the end added to the PC of the branch instruction +4. The circuit for this calculation is shown in Figure 15

8.2 LT/GE calculation

The BGTZ and BLEZ branch commands can use the comparison operator present in the ALU. However, the BLTZ and BGEZ branch commands require some outside logic. We know that a two's complement number is < 0 if its most significant bit is 1 and ≥ 0 otherwise. The above is determined based on the most significant bit of A and bit extended to get the output for these weird branch commands. The circuit for this calculation is shown in Figure 16

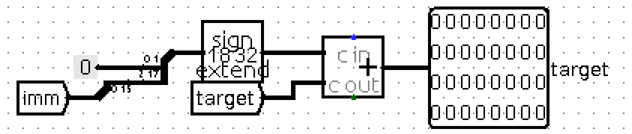


Figure 15: Branch Target Calculation

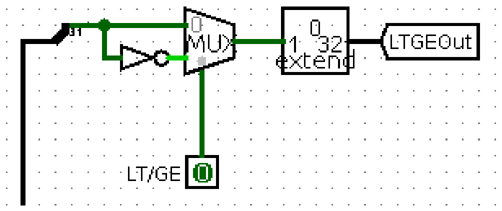


Figure 16: LT/GE Calculation

8.3 Choosing the proper output

The value stored in D, the output of the execute stage, does not necessarily equal the output of the ALU. Due to commands such as jumps and branches, more logic is necessary to determine D. If a branch instruction is currently in the execute stage, then the output is set to either the output of the ALU or, if it is a weird branch instruction, the output of the LT/GE calculation. If a jump and link instruction is currently in the execute stage, then D is set to the PC+8 value. Lastly, if a branch is taken, the the command currently in the fetch stage needs to be zapped. In that case, the output Zap is set to 1. Otherwise, it is set to 0. The circuit diagram is shown in Figure 17.

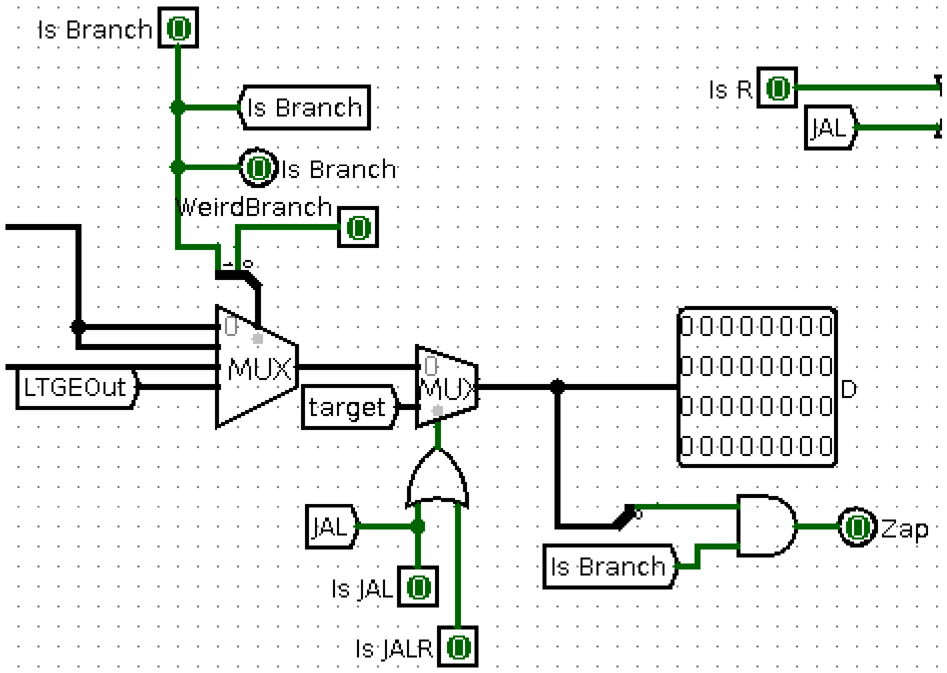


Figure 17: Choosing proper D

8.4 Submodule: Forwarding Unit

In this submodule, we determine if any sort of forwarding is needed to bypass a data hazard. We check if an Ex/Mem bypass or an Mem/WB bypass is necessary using the following subcircuits and also check to ensure that the immediate value is not needed instead. A circuit diagram is shown in Figure .

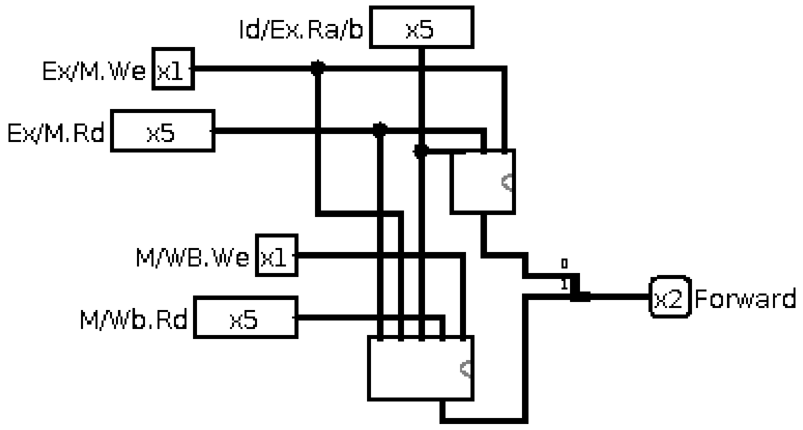


Figure 18: Forwarding Top Level Circuit

8.4.1 Ex/Mem Bypass

It checks if the value from the previous instruction currently entering the memory stage needs to be forwarded to the execute stage of the next instruction. The logic is as follows:
forward = (Ex/M.WE && EX/M.Rd != 0 && ID/Ex.Ra/b == Ex/M.Rd)
The circuit diagram is shown in Figure 19 14.

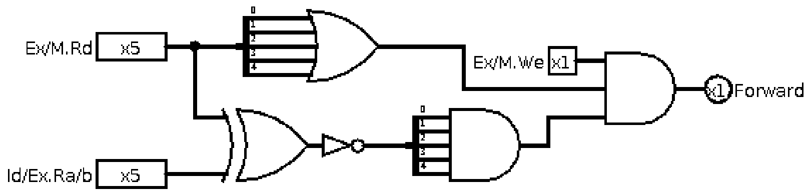


Figure 19: Ex/Mem Bypass Circuit

8.4.2 Mem/Wb Bypass

It checks if the value from two previous instructions ago currently entering the writeback stage needs to be forwarded to the execute stage of the next instruction. The logic is as follows:
forward = (M/WB.WE && M/WB.Rd != 0 && ID/Ex.Ra/b == M/WB.Rd) && not Ex/Mem Bypass
The circuit diagram is shown in Figure 20.

8.5 Submodule: EqZero and EqZero5

This submodule determines if a 32 or 5 bit input is equal to 0. We have made it a submodule primarily for space constraints in the execute stage. The circuit diagram is shown in Figure 21.

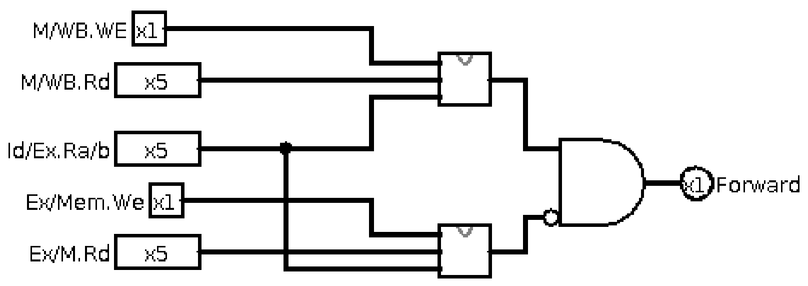


Figure 20: Mem/WB Bypass Circuit

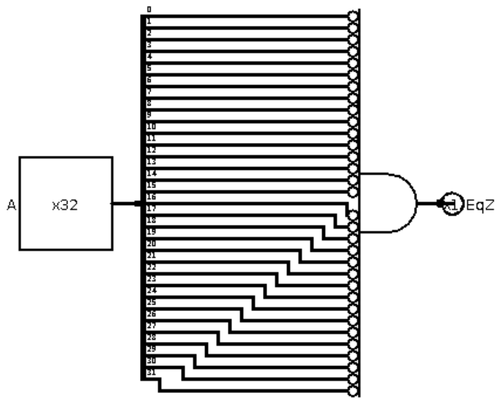


Figure 21: Equals Zero Circuit

8.6 Correctness Constraints

This subsection should be able to int erpret the control logic to select the correct operation in the ALU, as well as determing which values we want to input into the ALU. This all depends on the specifications of each instruciton.

9 The Memory Stage

Inputs: Store[32], Dest[5], D[32], MemSel[1], Clock[1], L/S[1], WE[1], Mem in[1], Sign/Unsign[1]

Outputs: D[32], Dest[5], Mem out[1], WE[1], M[32]

Store the value to be store in an store instruction
Dest the address to which the output should be written
D the value to be written
MemSel determines which memory value to output
Clock a clock input for the RAM
L/S signals whether Op is a load or store memory operation
WE signals whether we should enable writing to the Rd register
Mem in/Mem out signals whether Op is a memory instruction
Sign/Unsign signals whether Op is a signed or unsigned memory instruction
M the memory value to be stored by the instruction

For the memory stage we pass in three identification bits, Mem/Sel which determines the select bits for the RAM, L/S which determines whether the instruction is a load or a store (0 when load, 1 when store), and Mem in which is treated as a passthrough to later be used in the Writeback Stage.

9.1 Byte Select Logic

Depending on if we want to read/write an entire word or just a byte, we need to set the the byte select input to the RAM. The MemSel input is 1 when the entire word should be considered and 0 when only a byte should be. In the case that a byte must be read, the 2 least significant bits of D specify the index. The truth table for this logic is shown below. The circuit diagram is shown in Figure 22

$MemSel$	D_1	D_2	Sel_3	Sel_2	Sel_1	Sel_0
0	0	0	0	0	0	1
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	1	1	1	0	0	0
1	0	0	1	1	1	1

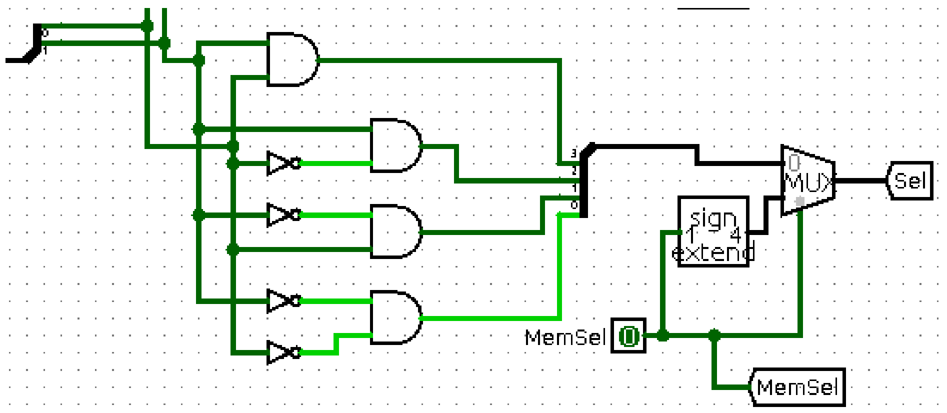


Figure 22: Byte Select Logic

9.2 Loading Byte Logic

Loading a byte provides its own set of challenges. The 8 bits to be loaded need to be placed in the 8 least significant bits of the output and either sign extended or 0 extended. The logic that determines this is shown in Figure 23

9.3 Storing Byte Logic

When storing a byte, we must take the least 8 significant bits of the Store input and shift them left by 8 times the byte in which the value is to be stored. The logic that determines this is shown in Figure 24

10 The Writeback Stage

Inputs: Dest[5], Mem[1], WE[1], D[32], M[32]

Outputs: rW[5], WE[1], D[32]

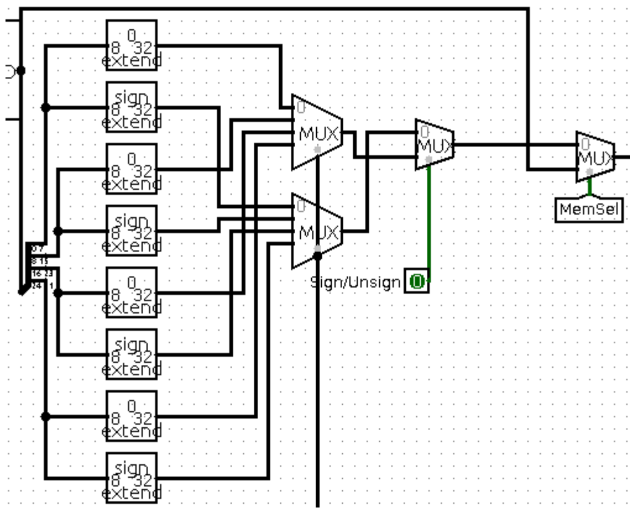


Figure 23: Loading Byte Logic

Dest/rW the register to be written to
Mem signals whether Op is a memory operation
WE signals whether we should enable writing to the Rd register
D the value to be written
M the value stored in memory

The write back stage sends the final computed values back to the register. The circuit diagram is shown in Figure 25.

10.0.1 M or D to D Mux

This multiplexer determines if we send the output value from memory stage or the output value from the execute back to the execute stage as the D value. The Mux selects the 0 value if we want to use the current D and 1 if we want to use Memory output. This is based on the value of Mem, which we have determined earlier (it is 1 when the instruction accesses memory). The mux select bit is then simply the value of Mem. WE, whether or not write is enabled on the register, and Dest, the address of the register to potentially write a value to have also been calculated previously and are just passed through here.

10.0.2 Correctness Constraints

The Writeback stage should write back the correct value to the correct register if it is an instruction that modifies a value in a register. It should return a value from memory if it is a memory instruction and the value we computed in the execute stage otherwise.

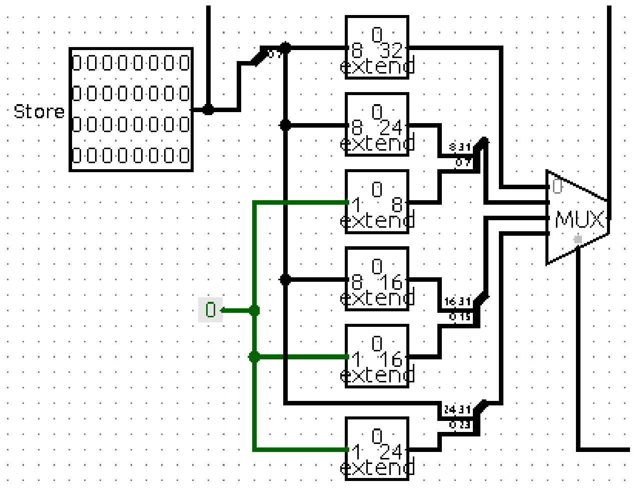


Figure 24: Storing Byte Logic

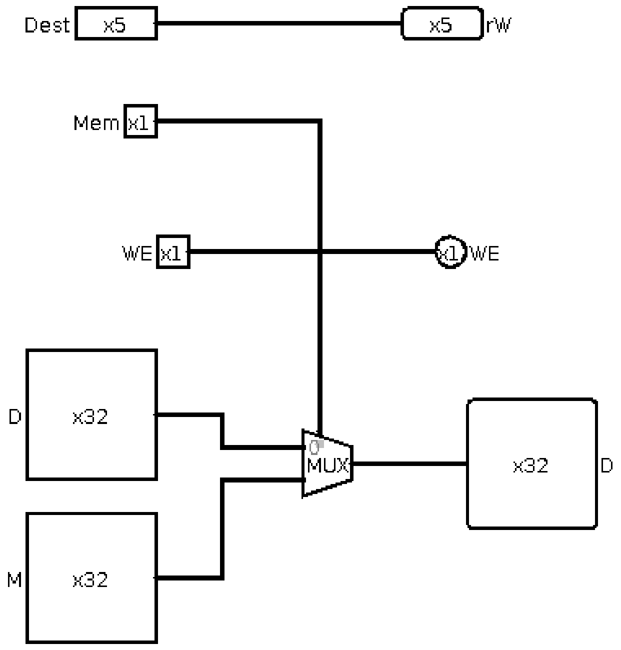


Figure 25: Writeback Stage