Mini-RISC-V Processor Design Doc

Rahul Kindi (rkk59) and Daniel Stabile (dis52)

Overview

For this mini RISCV project the processor will be broken up into five stages; namely Fetch, Decode, Execute, Memory, and Write Back. Our design is heavily based off the *Computer Organization and Design* RISC-V Edition used for our CS 3410 course. Because we are using five stages, we will have four pipeline registers that will be referred to by the stages that they connect. The following documentation explains what we will be reading and writing from and to these pipeline registers for each of the operations that must be performed.

Table of Contents:

General Rules

- Writes of the register file occur in first half of a cycle; reads occur in the second half

Arithmetic Functions and Pipeline Registers

add

Form: add x4, x3, x2 # stores the sum of x3 and x2 in x4

Stage	P. PR	Need for Comp. (P. PR)	N. PR	Write to (N. PR)
IF		Directly Access: -PC Write: -PC set to PC + 4	IF/ID	-PC -Instruction
ID	IF/ID	Write Half: -Wait since the register might having something writing to it right now Read Half: -Get **instruction bits** from IF/ID.Instruction -Generate **control signals** from IF/ID.Instruction	ID/EX	Write Half: -Wait since the register might have something writing to it right now Read Half: -Use the instruction bits to put the specified **value of register A** in ID/EX.AValue -Use the instruction bits to put the specified **value of register B** in ID/EX.BValue -Put target register location in ID/EX.TargReg -Put instruction bits in ID/EX.InsBit -Put control signals in ID/EX.CtrlSig
EX	ID/EX	-Use ID/EX.AValue and ID/EX.BValue and take the sum (using ID/EX.CtrlSig)	EX/MEM	-Store sum in EX/MEM.ALUOut -Store destination register in EX/MEM.TargReg -Copy over: EX/MEM.CtrlSig, EX/MEM.Ctrl
MEM	EX/MEM	Do nothing.	MEM/WB	-Copy over EX/MEM

WB	MEM/WB			Write: -Write to MEM/WB.ALUOut to MEM/WB.TargReg

sub

Same as add, but in the EX step, we need to use ID/EX.CtrlSig to determine whether or not to set the carry-in for the ALU to 1 and whether or not to flip all the bits of B.

addi

Form: addi x2, x1, 20 # stores the sum of x1 and 20 in x3

Stage	P. PR	Need for Comp. (P. PR)	N. PR	Write to (N. PR)
IF		Directly Access: PC Write: PC set to PC + 4	IF/ID	-PC -Instruction
ID	IF/ID	Write Half: -Wait since the register might having something writing to it right now Read Half: -Get immediate from IF/ID.Instruction and sign extend -Get **instruction bits** from IF/ID.Instruction -Generate **control signals** from IF/ID.Instruction	ID/EX	Write Half: -Wait since the register might have something writing to it right now Read Half: -Use the instruction bits to put the specified **value of register A** in ID/EX.AValue -Store sign extended immediate in ID/EX.IMM -Put target register location in ID/EX.TargReg -Put instruction bits in ID/EX.InsBit -Put control signals in ID/EX.CtrlSig
EX	ID/EX	-Use ID/EX.CtrlSig to choose to sum ID/EX.AValue and ID/EX.IMM	EX/MEM	-Store sum in EX/MEM.ALUOut -Store destination register in EX/MEM.TargReg -Copy over: EX/MEM.CtrlSig, EX/MEM.Ctrl
MEM	EX/MEM	Do nothing.	MEM/WB	-Copy over EX/MEM
WB	MEM/WB			Write: -Write to MEM/WB.ALUOut to MEM/WB.TargReg

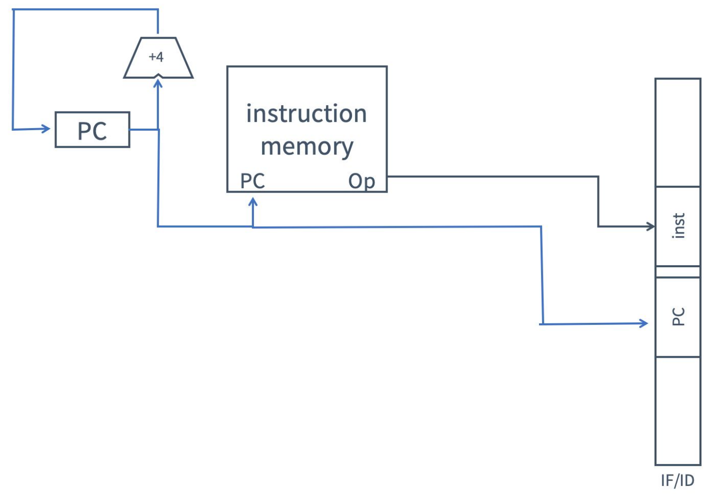
auipc

Form: auipc x2, 20 # stores the sum of PC and 20 in x2

Stage	P. PR	Need for Comp. (P. PR)	N. PR	Write to (N. PR)
IF		Directly Access: -PC Write: -PC set to PC + 4	IF/ID	-PC -Instruction
ID	IF/ID	Write Half: -Wait since the register might having something writing to it right now. (Not actually needed in this case). Read Half: -Get immediate from IF/ID.Instruction and sign extend -Get **instruction bits** from IF/ID.Instruction -Generate **control signals** from IF/ID.Instruction	ID/EX	Write Half: -Wait since the register might have something writing to it right now Read Half: -Store PC in ID/EX.PC -Store sign extended immediate in ID/EX.IMM -Put target register location in ID/EX.TargReg -Put instruction bits in ID/EX.InsBit -Put control signals in ID/EX.CtrlSig
EX	ID/EX	-Use ID/EX.CtrlSig to choose to sum ID/EX.PC and signext(ID/EX.IMM, 20)	EX/MEM	-Store sum in EX/MEM.ALUOut -Store destination register in EX/MEM.TargReg -Copy over: EX/MEM.CtrlSig, EX/MEM.Ctrl
MEM	EX/MEM	Do nothing.	MEM/WB	-Copy over EX/MEM
WB	MEM/WB			Write: -Write to MEM/WB.ALUOut to MEM/WB.TargReg

The Fetch Stage:

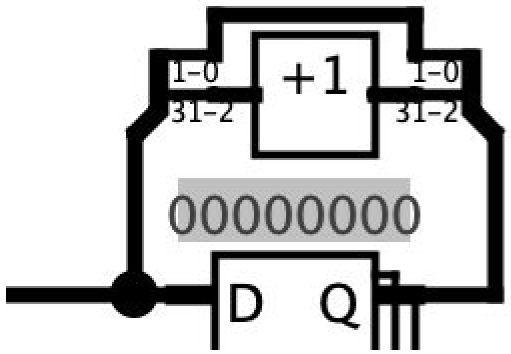
Circuit Diagram



All operations, including nops and the table B operations, require advancing the PC by 4 with certainty for each clock cycle. Therefore, we increment by 4 and store back in the PC. We pass the unmodified PC to the instruction memory ROM which produces the instruction we desire. Both the instruction and the unmodified PC (before the 4 is added) are stored in the IF/ID pipeline register.

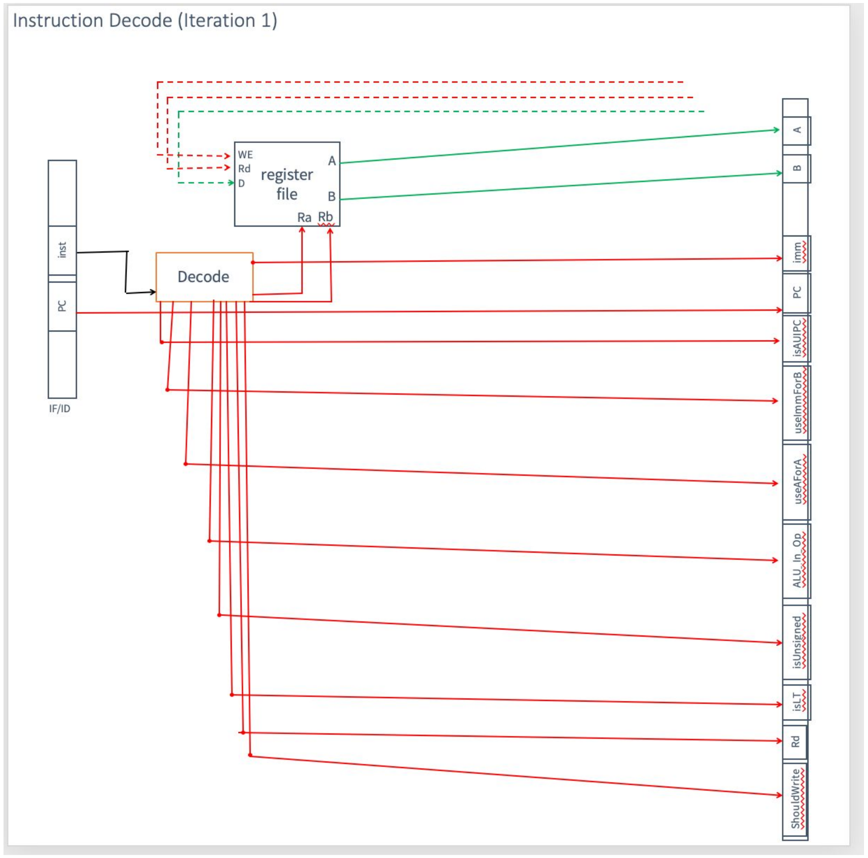
PC Incrementor:

Because we are only given a +1 incrementor to implement the +4 incrementor for the PC counter we will split our 32 bit PC signal into two. Bits 2-31 will be passed through the +1 incrementor while bits 0-1 will be unchanged. Fusing the two signals back together will give the desired effect of a +4 incrementor.



The Decode Stage:

Circuit Diagram

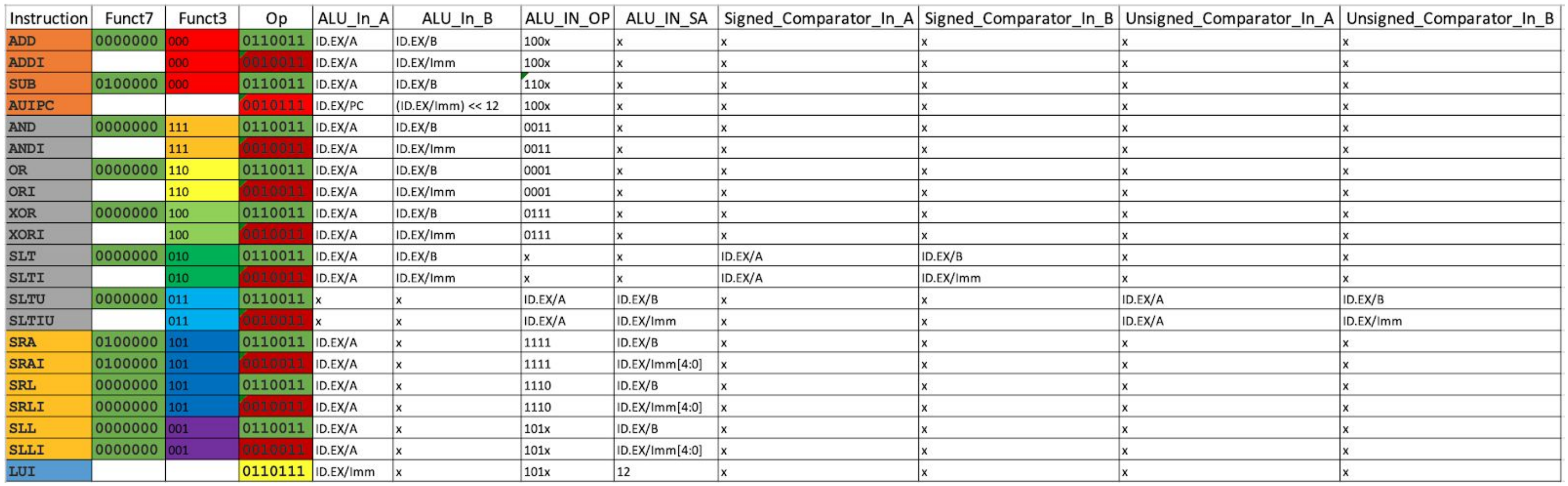


The instruction decode stage takes the contents of the IF/ID pipeline register and produces the following for the ID/EX pipeline register: A (the value at Ra), B (the value at Rb), imm (32 bit immediate value), PC, isAUIPC (whether or not the instruction is an

AUIPC command), useImmForB (set to 0 if what should be passed to the ALU B input is B; set to 1 otherwise - this happens as indicated in the ALU_In_B column in the table below), useAForA (set to 1 if what should be passed to ALU A input is A; 0 if something else should be passed - this happens with the immediate operations since the immediate should be passed to B - or in the case of AUIPC, the immediate shifted left by 12 should be passed to B), ALU_In_Op (what should be passed in as the op to the 3410 ALU component), isUnsigned (set to 1 if we are dealing with an unsigned lt comparison or unsigned lt immediate comparison), isLT (set to 1 if we are dealing with a less than comparison operator), Rd (the destination register - this can easily be found because for R, I, and U types, Rd is always the same set of bits in the instruction), and ShouldWrite (whether this operation needs to write back to the register file; this is always true for the operations in Table A).

The idea is that these pipeline registers will help when choosing amongst the computations generated in the EX stage. A bunch of computations are simultaneously done, but only one of those computations is correct based off of the instruction that is supplied. These signals in the pipeline register can be used to deliberate which computation to finally choose (more details in the EX stage section).

Further, we note that at this time we do not currently have all the logic set up to produce these pipeline registers, but the process to generate the minimal required logic to generate these signals is outlined here: we know that the signals can be fully be determined from the bits that correspond to the funct7, funct3, and Op spaces (we consider the bits in these spaces even for cases like AUIPC where there is no funct7). We build a table not dissimilar to the one included below. We enumerate all the functions in the leftmost column and then we write down the funct7, funct3, and Op values. We then discard all values that are the same across all the instructions that we would like to perform (because these will not factor into distinguishing which value to write to a signal because they are not different across different instructions). Once we do this, we enumerate a table of 1’s, 0’s, and x’s that describe the input output relationships between the retained bits of funct7, funct3, and Op and the desired values of the signals for these inputs. If we build this table in excel, we noticed that we can just paste the table into Logisim and have it minimize for us. This will generate a subcircuit for the desired signals we want to write to the pipeline register.

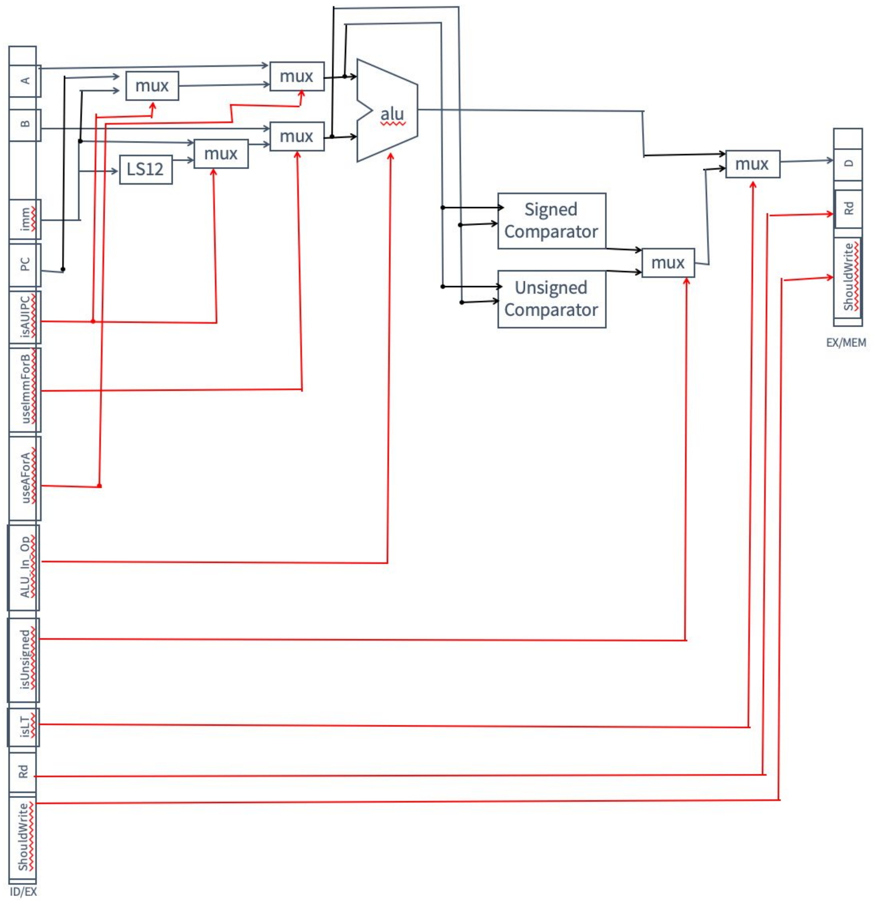


Instruction	Funct7	Funct3	Op	ALU_In_A	ALU_In_B	ALU_IN_OP	ALU_IN_SA	Signed_Comparator_In_A	Signed_Comparator_In_B	Unsigned_Comparator_In_A	Unsigned_Comparator_In_B
ADD	0000000	000	0110011	ID.EX/A	ID.EX/B	100x	x	x	x	x	x
ADDI		000	0010011	ID.EX/A	ID.EX/Imm	100x	x	x	x	x	x
SUB	0100000	000	0110011	ID.EX/A	ID.EX/B	110x	x	x	x	x	x
AUIPC			0010111	ID.EX/PC	(ID.EX/Imm) << 12	100x	x	x	x	x	x
AND	0000000	111	0110011	ID.EX/A	ID.EX/B	0011	x	x	x	x	x
ANDI		111	0010011	ID.EX/A	ID.EX/Imm	0011	x	x	x	x	x
OR	0000000	110	0110011	ID.EX/A	ID.EX/B	0001	x	x	x	x	x
ORI		110	0010011	ID.EX/A	ID.EX/Imm	0001	x	x	x	x	x
XOR	0000000	100	0110011	ID.EX/A	ID.EX/B	0111	x	x	x	x	x
XORI		100	0010011	ID.EX/A	ID.EX/Imm	0111	x	x	x	x	x
SLT	0000000	010	0110011	ID.EX/A	ID.EX/B	x	x	ID.EX/A	ID.EX/B	x	x
SLTI		010	0010011	ID.EX/A	ID.EX/Imm	x	x	ID.EX/A	ID.EX/Imm	x	x
SLTU	0000000	011	0110011	x	x	ID.EX/A	ID.EX/B	x	x	ID.EX/A	ID.EX/B
SLTIU		011	0010011	x	x	ID.EX/A	ID.EX/Imm	x	x	ID.EX/A	ID.EX/Imm
SRA	0100000	101	0110011	ID.EX/A	x	1111	ID.EX/B	x	x	x	x
SRAI	0100000	101	0010011	ID.EX/A	x	1111	ID.EX/Imm[4:0]	x	x	x	x
SRL	0000000	101	0110011	ID.EX/A	x	1110	ID.EX/B	x	x	x	x
SRLI	0000000	101	0010011	ID.EX/A	x	1110	ID.EX/Imm[4:0]	x	x	x	x
SLL	0000000	001	0110011	ID.EX/A	x	101x	ID.EX/B	x	x	x	x
SLLI	0000000	001	0010011	ID.EX/A	x	101x	ID.EX/Imm[4:0]	x	x	x	x
LUI			0110111	ID.EX/Imm	x	101x	12	x	x	x	x

Handing Table B Operations:

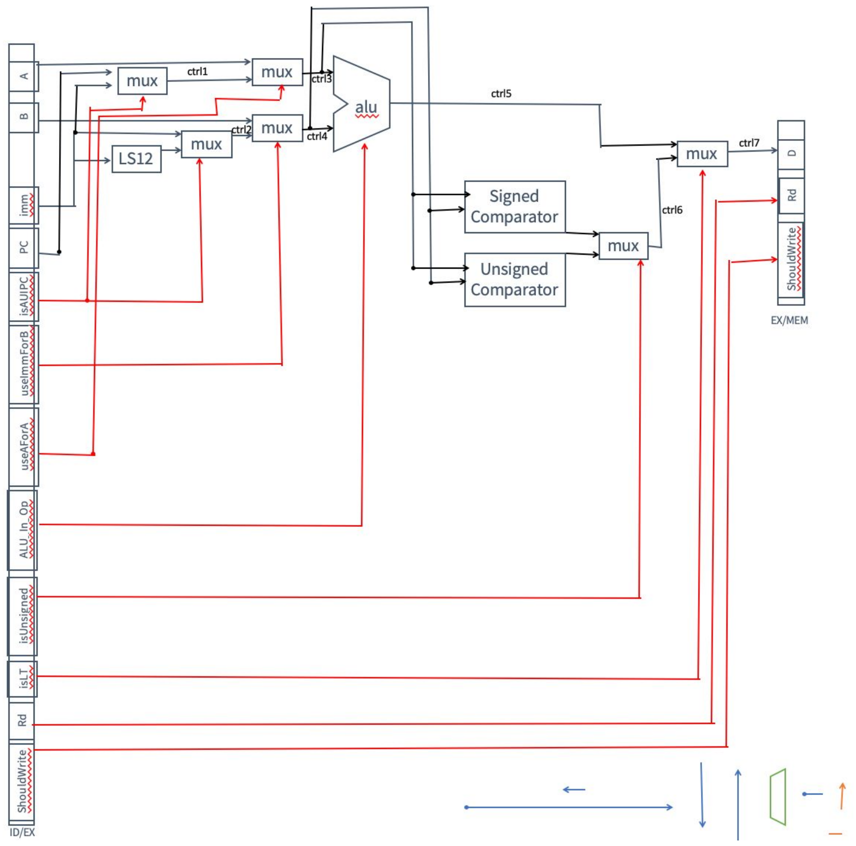
We plan to handle Table B operations by setting the values of the ID/EX pipeline register to be those equivalent to what would be generated if the operation were instead add x0, x0, x0. This will prevent those operations from having any effect on the output of the processor since x0 is always supposed to be 0. This is accounted for in the above description of the “Decode” block logic when building the excel input/output map.

The Execute Stage:



In this stage, we utilized the pipeline register values produced in the ID stage (and now stored in the ID/EX pipeline register) to simultaneously do computations and filter then down using a sequence of muxes. TODO: label the signals going to and out of the muxes. Describe them for the meeting. Note that we don't need to worry about Table B values here because they are converted to an add x0, x0, x0 operation in the instruction decode stage. We only need to store the data to write back to the register file (D), where to write it back to (Rd), and the fact that our operation does need to write back to the register file (ShouldWrite - this is also set to 0 if the destination register is equal to 0). This value is set to 1 for all of the instructions in Table A.

The image below labels certain wires with ctrl + a number. Below the image is a textual description of what the values on these lines are. A table with all the values for these wires given all the various pipeline register inputs is omitted due to the sheer size of what the table would be and to retain simplicity in this preliminary document.



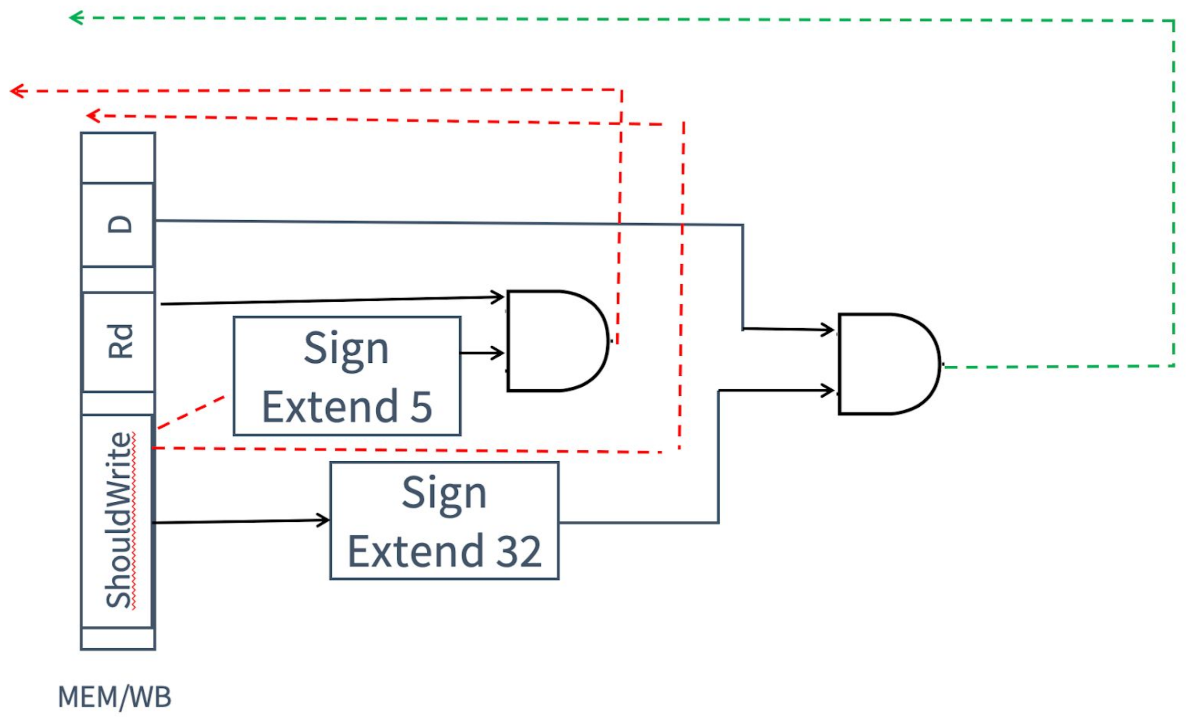
Ctrl1 - imm if isAUIPC else PC
Ctrl2 - (imm << 12) if isAUIPC else imm
Ctrl3 - A if useAForA else Ctrl1

Ctrl4 - Ctrl2 if useImmForB else B
Ctrl5 - the output of the ALU given the appropriate opcode. This output is best described by the color coded table in the Decode stage section.
Ctrl6 - Result of Unsigned Comparator if isUnsigned else Result of Signed Comparator
Ctrl7 - Ctrl6 if isLT else Ctrl5

The Memory Stage:

This is a pass through so we just connect the EX/MEM and MEM/WB pipeline register values together.

The Write Back Stage:



In the write-back stage, we simply take the results of the pipeline registers MEM/WB and put the results back into the register file. The AND gates and the sign extends ensure that if the ShouldWrite bit is set to 0 (in the case we are dealing with an operation that should not write back to the register - for example, if the operation is a Table 2 operation) that the value that gets written back is set to 0 and the place where it is being put is the 0th register x0. Since ShouldWrite accounts for the case where Rd=0, this prevents a non-zero value from being written into x0.

Table 1 Operations General Information:

Operation	Instr. Fmt.	Operation	Instr. Fmt.
AND	R	ADD	R
ANDI	I	ADDI	I
OR	R	SUB	R
ORI	I	AUIPC	U
XOR	R	SRA	R
XORI	I	SRAI	I
SLT	R	SRL	R
SLTI	I	SRLI	I
SLTU	R	SLL	R
SLTIU	I	SLLI	I
NOP	Pseudo	LUI	U

Instr. Fmt.	
R	ADD, SUB, AND, OR, XOR, SLT, SLTU, SRA, SRL, SLL
I	ADDI, ANDI, ORI, XORI, SLTI, SLTIU, SRAI, SRLI, SLLI
U	AUIPC, LUI

Arithmetic Operations
Logic Operations
Shift Operations
Immediate Load

Instr. Fmt.	EX/Address Calculation Stage Control Lines		Write Back Stage Control Lines	
	ALUOp	ALUSrc	RegWrite	MemtoReg
R	10	0	1	0
I	00			
U	01			

Hazards:

Because our memory stage in the pipeline is empty, the results of all previous computation are generated by the end of the EX stage (stage 3/5). Since the register pipeline in the instruction decode stage (stage 2/5) looks for the falling edge while the other pipeline registers are looking for the rising edge, we can forward from the end of the EX stage to the beginning of the ID stage. This means that none of the operations we have to support require stalling to avoid hazards. **Therefore, we only need to implement forwarding (from EX/MEM to EX and MEM/WB to EX).**

EX/MEM to EX forward = (Ex/M.WE && EX/M.Rd != 0 && ID/Ex.Rs1 == Ex/M.Rd) || (same for Rs2)

MEM/WB to EX forward = (M/WB.WE && M/WB.Rd != 0 && ID/Ex.Rs1 == M/WB.Rd && not (ID/Ex.WE && Ex/M.Rd != 0 && ID/Ex.Rs1 == Ex/M.Rd) || (same for Rs2)

Testing:

We will be testing via a combination of manual edge cases and randomly generated cases to test our implementation. The randomly generated cases will be made by simulating the running of a sequence of instructions in Java. The Java code we write will automatically generate the corresponding assembly and the desired outputs. The manual edge cases will focus on making sure that the hazard handling is correct. We will explicitly write cases to test that the various forwarding cases we have.

Control Signals:

First, we describe how the control signals look like assuming that we are only supporting the table A operations and assuming that our model is not incorporating any hazard detection, stalling, etc.

Instruction Fetch

- Don't increment by 4.
- WE for the IF/ID pipeline registers.

Instruction Decode

- Register File Inputs
    - xA
        - Bits 19-15 (for both R and I type)
    - xB
        - Bits 24-20 (for R type)
    - xW
        - ~~Bits 11-7 (for R, I, and U type)~~
        - Comes from the WB phase
    - W
        - Comes from the WB phase

- WE
  - Comes from the WB phase
- 2 Immediate Values
  - I_Imm (for I Type)
    - Bits 31-20 => Sign extend
  - U_Imm (for U Type)
    - Bits 31-12 => Sign extend
- Control
  - Funct7
    - Bits 31-25
  - Funct3
    - Bits 14-12
  - Op
    - Bits 6-0

Magic Decoder Cell

- Getting the immediate
- Relevant: (11)
  - 31-20
    - ADDI
    - ANDI
    - ORI
    - XORI
    - SLTI
    - SLTIU
  - 31-25
    - SRAI

        - SRLI
        - SLLI
    - 31-12
        - LUI
        - AUIPC
- Irrelevant: ADD, SUB, AND, OR, XOR, SLT, SLTU, SRA, SRL, SLL (10)
    - Table B: JAL, JALR, BEQ, BNE, BLT, BGE, BLTU, BGEU, LW, LB, SW, SB
-