CS3410 Project 3 Design Document

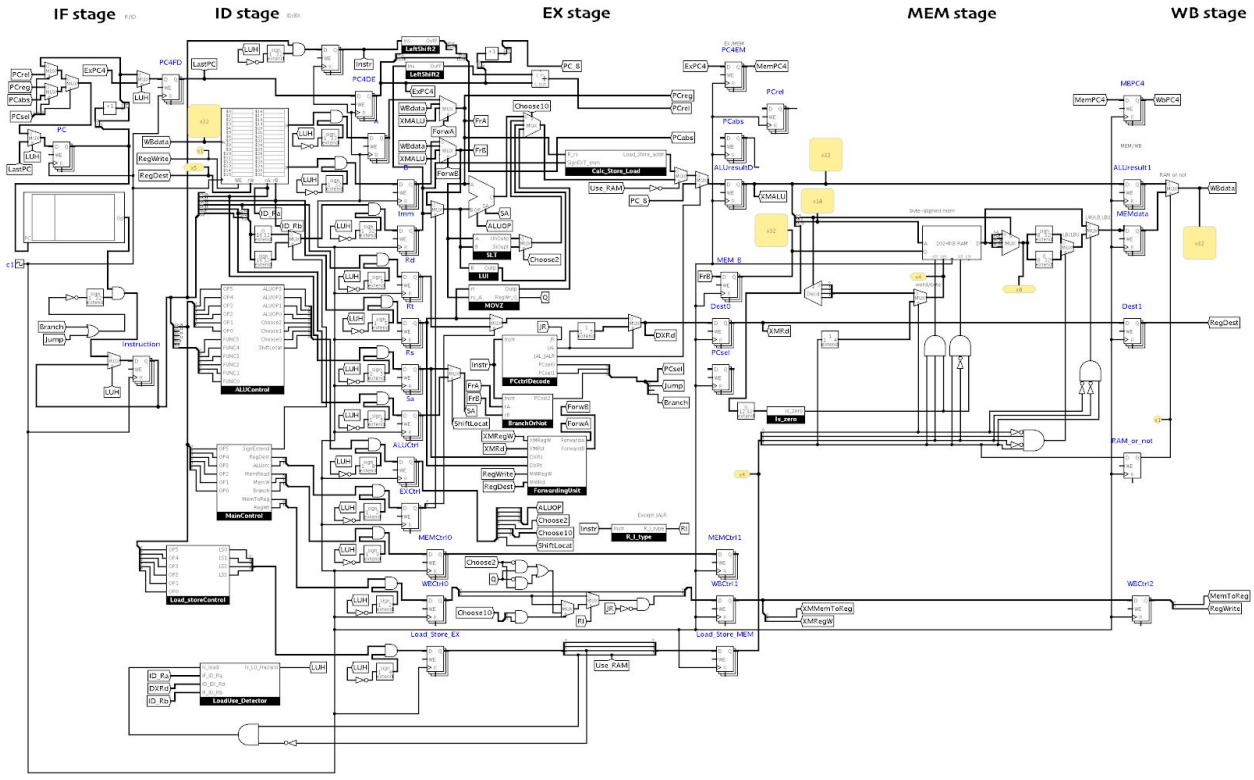
Pipelined Mini-MIPS

Bill Tang (bt294), Hao Rong (hr335)

March 28th, 2018

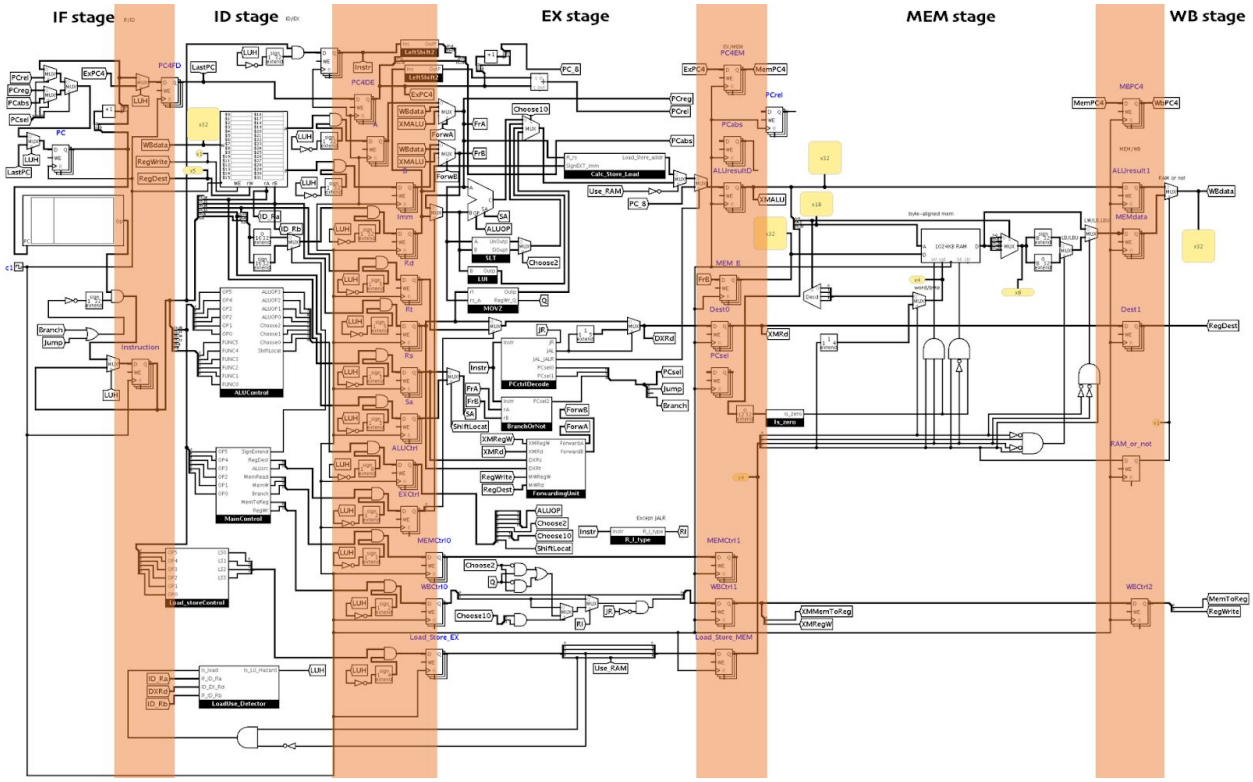
1 Overview

In this project we implemented the second subset of the MIPS32 table (Table B) using Logisim. We implemented a functioning outline of the pipelined processor for a small set of instructions, including: decoding instructions, implementing most of the MIPS pipeline, implementation of arithmetic and logic operations, and hazard avoidance. To make load and store instructions possible, RAM is used to load and store values; to make Jump and Branch instructions valid, PC is upgraded to allow more updating options.



2 Pipeline Stages Design

Our processor is consisted of five stages of pipeline including IF(Fetch), ID(Decode), EX(Execute), MEM(Memory), MEM (memory stage), WB(Writeback).



Between stages in the middle (namely IF/ID, ID/EX, EX/MEM, MEM/WB) are pipeline registers which are used to pass on either data or control signal to the next stage. The pipeline registers and PC register (which is used to select instruction) are updated on the rising edge of the clock while the register file are updated on the falling edge. This setting is to make sure that we write data after read.

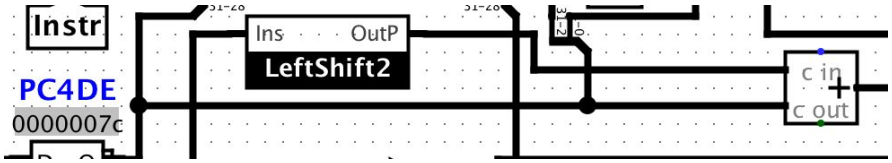
To accommodate more instructions such as jump and load, more pipeline registers are used. For instance, “Load_Store” register is added in ID_EX and EX_MEM stages to pass the load and store instructions. Another major change in project 3 about the pipeline registers is the “flushing” functionality which basically replaces all stored bits in pipeline registers with 0s when control hazard or load-use data hazard happens. More details will be provided in hazard section.

2.1 PC & Instruction Fetch logic

2.1.1 PC calculation

PC (Program Count) is incremented through a +1 incrementer. We use this +1 incrementer on the third bit of PC to realize the effect of +4. The PC register will pass the current value to program memory in the current cycle, ready to update +4 PC in the next cycle. Also there are PC register that stores PC +4 value in the pipeline register which will be used for jump in next project.

PCrel represents for relative address for the Branch type instruction. When Branch type condition is confirmed, next PC will be PCrel. The PCrel is calculated through adding PC+4 and the offset.



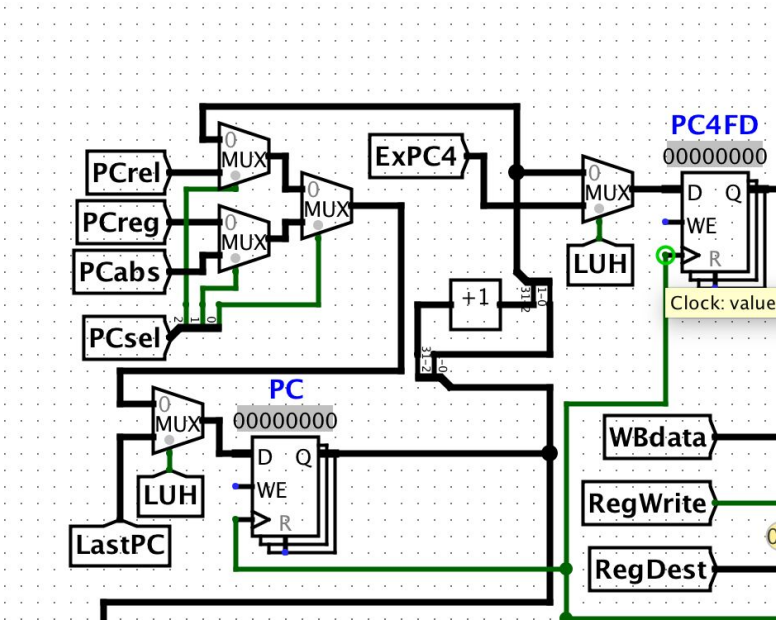
PCreg represents for address from file register. When JR or JALR instruction runs, next PC will be PCreg. PCabs represents the absolute address when J or JAL instruction runs, next PC will be PCabs. PCabs is calculated through combining current PC with part of the immediate field value.

All these PC address is calculated in Execution stage and are passed to Instruction Fetch stage.

2.1.2 PC selection

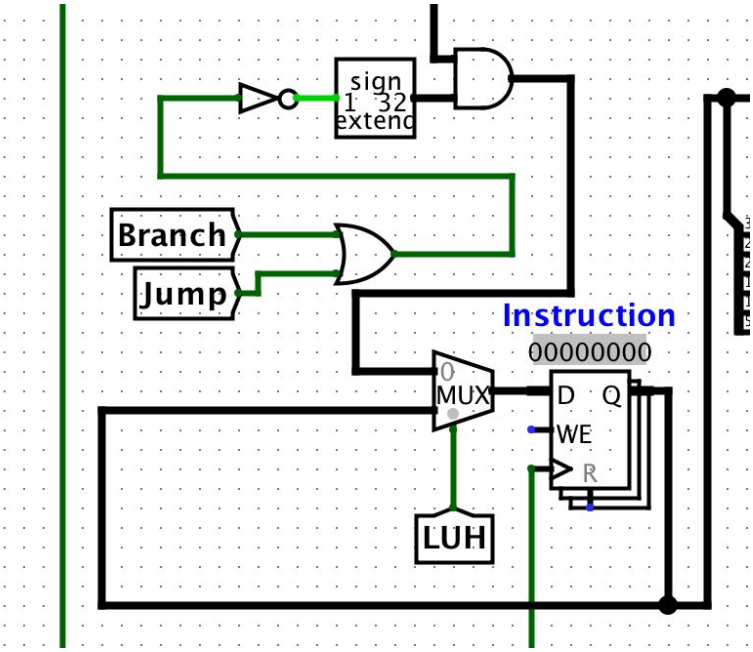
What type of PC we are going to use is controlled by the PCsel control signal passing from Execution stage to Fetch stage. PCsel [0] represents whether we are using [Jump type] or [Branch or normal] type. PCsel [1] represent whether we are using [J or JAL] or [JR or JALR] type. PCsel[2] represent whether [non-Jump-Branch-type or Branch but fail] or [Branch success].

Meanwhile the LUH (Load Use Hazard) control signal control whether we are going to stall PC for a cycle by passing the PC from last cycle to the current one.



2.1.2 Instruction Fetch

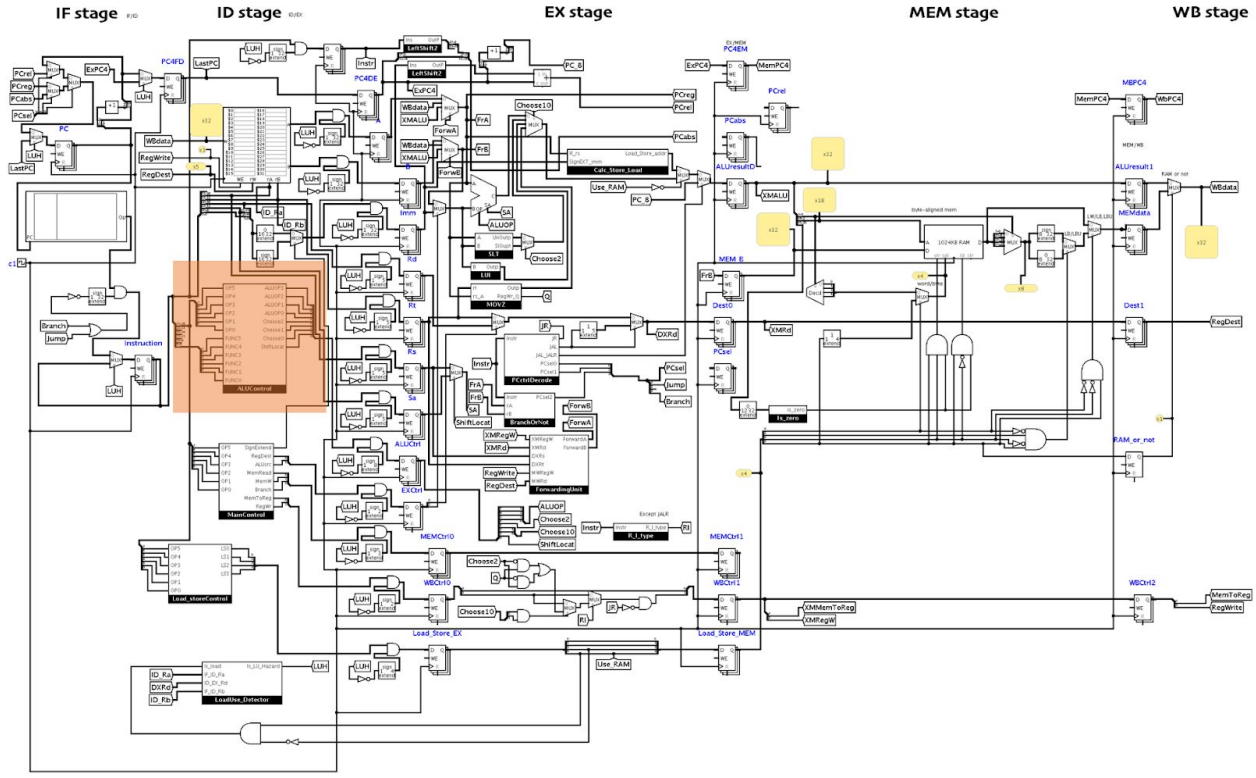
While PC is bypassing by PCsel and stalling by LUH (Load Use Hazard). Instruction Fetch follows the similar logic. When there are branch, jump condition, we flush the instruction by passing all zero instruction into instruction register so it will work as NOP. When LUH happened, we stall the cycle by passing the instruction from the last cycle to the current cycle.



2.2 Instruction Decode

In this stage, the 32 bits instruction is read from IF/ID pipeline register into the register file and various decoders for different instruction types. Among them are ALUControl unit, MainControl unit, Load_store_Control for further decode. Register file takes in data from Write Back stage, “Regwrite” which enables the write, “RegDest” which gives the destination register of the data, “rA” and “rB” from instruction’s 25-21 and 20-16 bits (5 bits location codes which tells the locations of chosen registers) as in inputs. Then the register file outputs wanted 32 bits data in A and B from chosen registers. For ALUcontrol unit and MainControl unit, we have detailed them in the design document for project 2. In the following subsection, we will specify the designing logic behind the newly added decoder: Load_StoreControl unit.

2.2.1 ALU Control

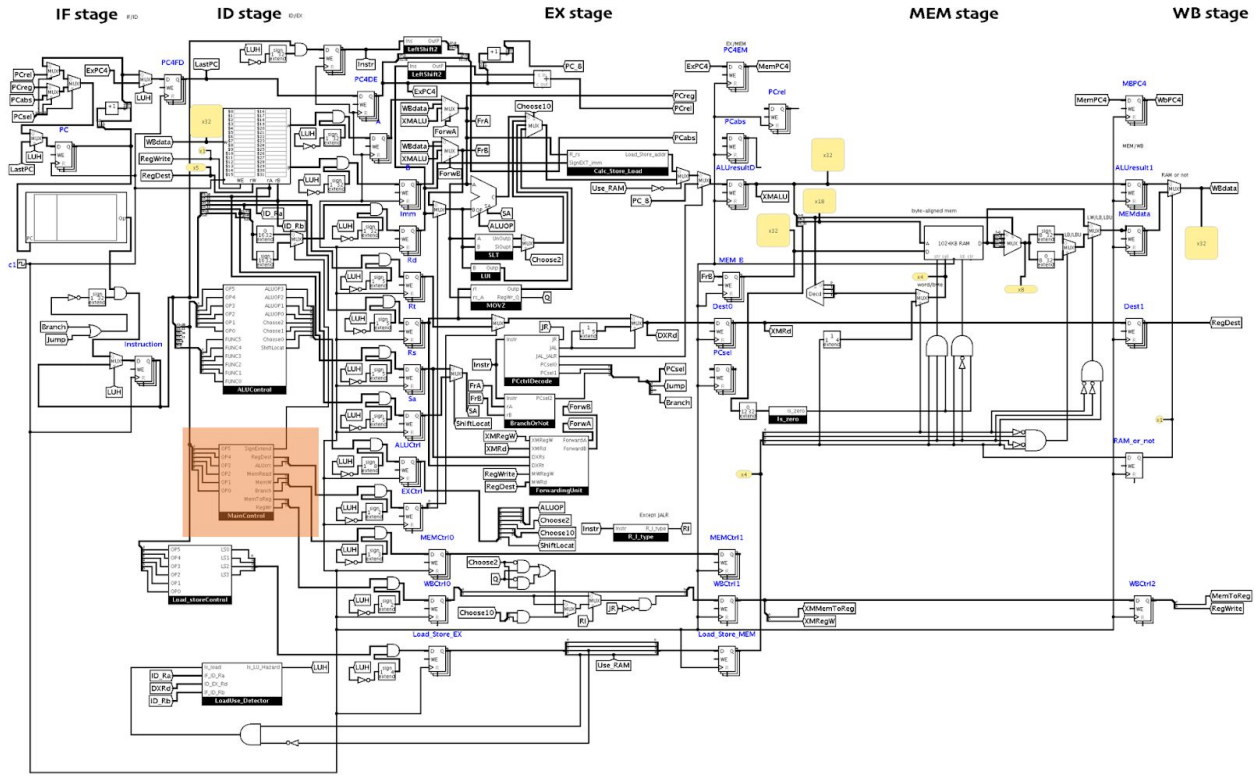


The ALU Control, marked in red in the above graph, is used to decode a 32 bits instruction. It takes in 6 bits opcodes and 6 bits function codes and outputs 8 bits instruction code to be stored in the “ALUCtrl” pipeline register. The first bit “Shiftocal” which stands for shift location will be the control bit for the MUX deciding the shift result location for Shift Logical and Shift Logical variable. “Choose 0 - 2” are 3 control bits used to choose the execution result among the ALU, “SLT”, “LUI”, and “MOVZ” computing units which will be addressed later. “ALUOP 0 - 3” are 4 bits control codes to ALU to choose among all instructions allowed such as addition or subtraction.

We used Logism’s build-in helper function to build this unit after we thoroughly thought through the inputs and desired output of such unit. The truth table is attached below:

ALU-control								
instruction opcode	Instruction operation	OPcode	function filed	ALU control input	ChooseALU	ShiftLocat (1=sa;0=rs)		desired ALU action
	load		next	next				add
	store		next	next				add
	branch		next	next				subtract
	jump		next	next				
I type	addIU	001001	xxxxxx	001x	000	x		add
	SLTI	001010	xxxxxx	x	101	x		<
	SLTIU	001011	xxxxxx	x	001	x		<
	andI	001100	xxxxxx	1000	000	x		and
	orI	001101	xxxxxx	1010	000	x		or
	XORI	001110	xxxxxx	1100	000	x		xor
	LUI	001111	xxxxxx	x	010	x		
R type	SLL	000000	000000	000x	000	1		<<
	SRL	000000	000010	0100	000	1		>>
	SLLV	000000	000100	000x	000	0		<<
	SRLV	000000	000110	0100	000	0		>>
	SRA	000000	000011	0101	000	1		>>
	SRAV	000000	000111	0101	000	0		>>
	addu	000000	100001	001x	000	x		add
	subu	000000	100011	011x	000	x		sub
	and	000000	100100	1000	000	x		and
	or	000000	100101	1010	000	x		or
	XOR	000000	100110	1100	000	x		xor
	NOR	000000	100111	1110	000	x		nor
	SLT	000000	101010	x	101	x		<
	SLTU	000000	101011	x	001	x		<
	Movn	000000	001011	x	111	x		ne
	Movz	000000	001010	x	011	x		eq

2.2.2 Main Control



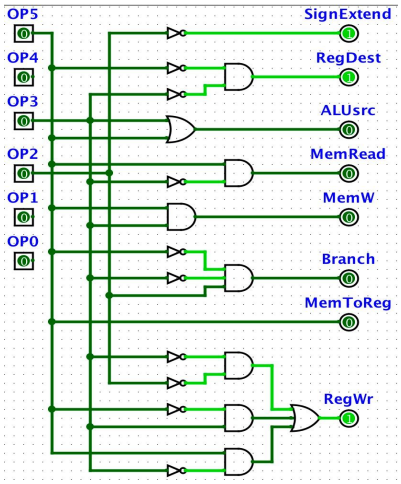
The Main Control, marked by orange above, takes in 6 bits Opcodes from the 32 bits instruction code and outputs 8 bits instruction code that will be used as control bits for later stage. The “SignExtend” bit is used to control the MUX for choosing the result between sign extend and zero extend. The designing of other bits functionalities are enlightened by book. Their meaning is given by the table below:

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

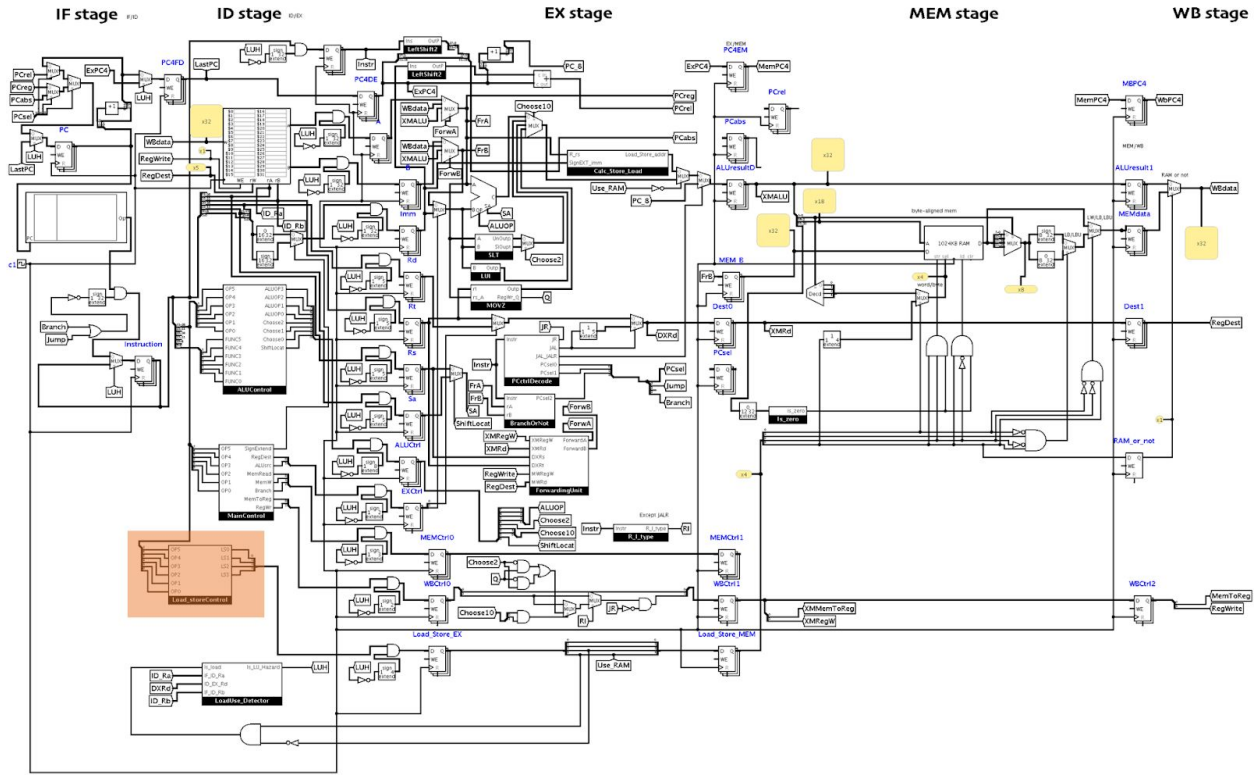
We used Logism's build-in helper function to build this unit after we thoroughly thought through the inputs and desired output of such unit. The truth table is attached below:

		Main Control						
			R-type	I-type	Load	Store	Jump	Branch
inputs		OP5	0	0	1	1	0	0
		OP4	0	0	0	0	0	0
		OP3	0	1	0	1	0	0
		OP2	0	x	x	x	0	1
		OP1	0	x	x	x	1	x
		OP0	0	x	x	x	x	x
outputs	bit location							
no-pass		SignExtend?	x	dfferent	x	x	x	x
To-Ex	0	RegDest	1	0	0	x	x	x
	1	ALUSrc	0	1	1	1	next time	0
To-M	0	MemRead	0	0	1	0	next time	0
	1	MemW	0	0	0	1	next time	0
	2	Branch	0	0	0	0	next time	1
To-WB	0	MemToReg	0	0	1	x	next time	x
	1	RegWr	1	1	1	0	next time	0

The result circuit from the above truth table is displayed below:



2.2.3 Load_StoreControl Unit



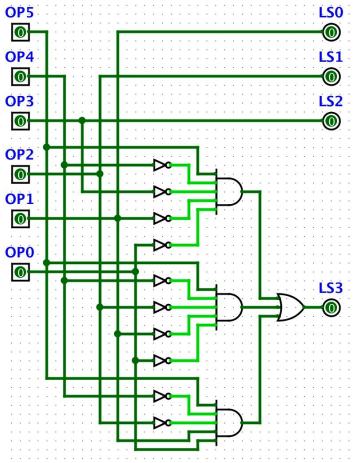
The Load_StoreControl unit, marked in red in the above graph, is used to decode a 32 bits load instruction . It takes in 6 bits opcodes and outputs 4 bits LoadStore control codes (LS0 - LS3). The logic and meaning behind each bit is summarized in the following table:

meaning		activated (1)	deactivated (0)
	LS3	RAM data passed	ALU data passed
	LS2	store	load
	LS1	load unsign	load sign
	LS0	word	byte

We used Logism's build-in helper function to build this unit after we thoroughly thought through the inputs and desired output of such unit. The truth table is attached below:

load_store setup	Instruction operation	OPcode	LS0	LS1	LS2	LS3
	load byte	100000	0	0	0	1
	load byte unsigned	100100	0	1	0	1
	load word	100011	1	0	0	1
	store byte	101000	0	x	1	1
	store word	101011	1	x	1	1
	others	xxxxxx	x	x	x	0
						cannot contanminate RAM

The result circuit from the above truth table is displayed below:



2.3 Execute

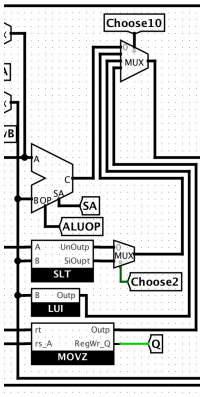
In Ex stage, we perform ALU operation and select proper result. We also save spot for branch. This is also where forwarding control (explained in the next section) make its effect to influence the input of ALU or other calculation units.

2.3.1 Input & Destination control

The input of calculation units is controlled by forwarding control with ALUsrc, which is used to select input from B or Imm through MUX. The register destination is also selected in the execution stage by RegDst, which choose destination between Rs register or Rd register through MUX.

2.3.2 Calculation Units control

The OP control signal is defined by us to control the output of which calculation unit to use. The OP code is decoded in the main control to pass on to a MUX to do the selection job. The calculation units include the original ALU (provided in the class folder), SLT unit, LUI unit, and MOV unit by the signal “Choose10”. The last three unit is created for specific genre of instruction can be easily handled outside the original ALU.



2.3.2 ALU

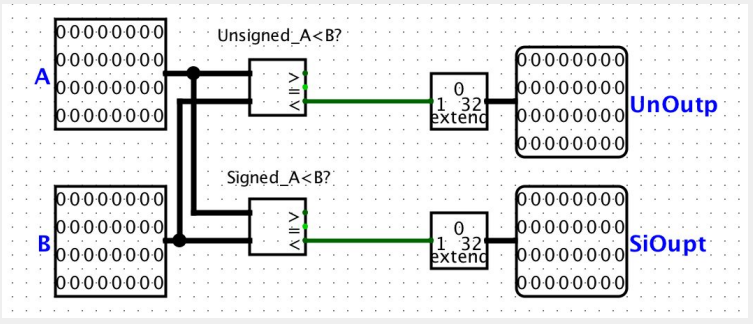
The ALU just the original ALU unit functioning as described in class documentation. The ALU 4-bit OP code and SA (shift amount, different in common shift or value shift) are decoded in ID stage by ALU decode. The ALU takes care of following instruction:

`ADDIU, ANDI, ORI, XORI, ADDU, SUBU, AND, OR, XOR, NOR, SLL, SRL, SRA, SLLV, SRLV, SRAV`

2.3.3 SLT unit

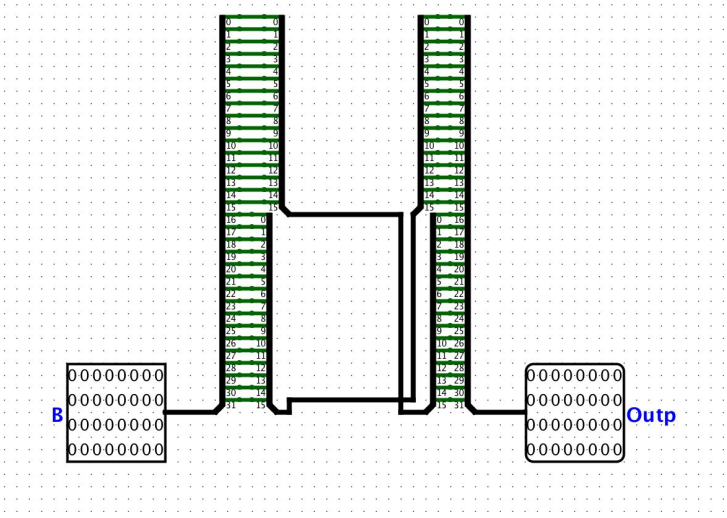
The SLT unit use the comparator to compare input both in signed or unsigned number. The SLT unit takes care of following instruction:

`SLTI, SLTIU, SLT, SLTU`



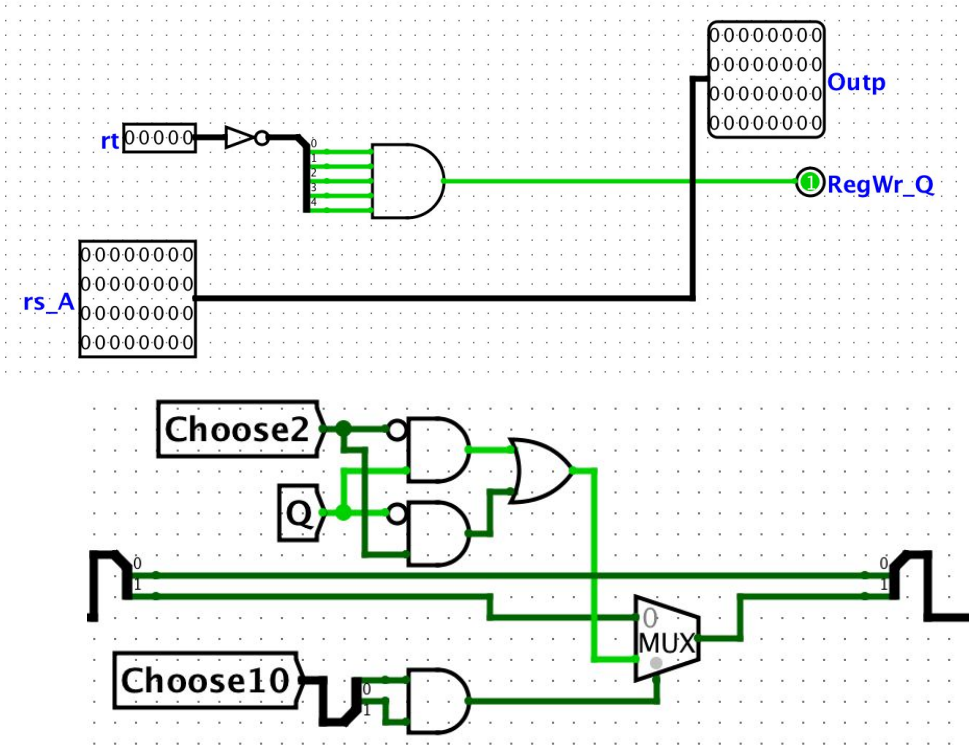
2.3.4 LUI unit

The input of LUI is previous zero-extend in the ID stage. So we just need to flip the upper 16 bit with the lower 16 bit using wire. The LUI unit only takes care of `LUI` instruction:



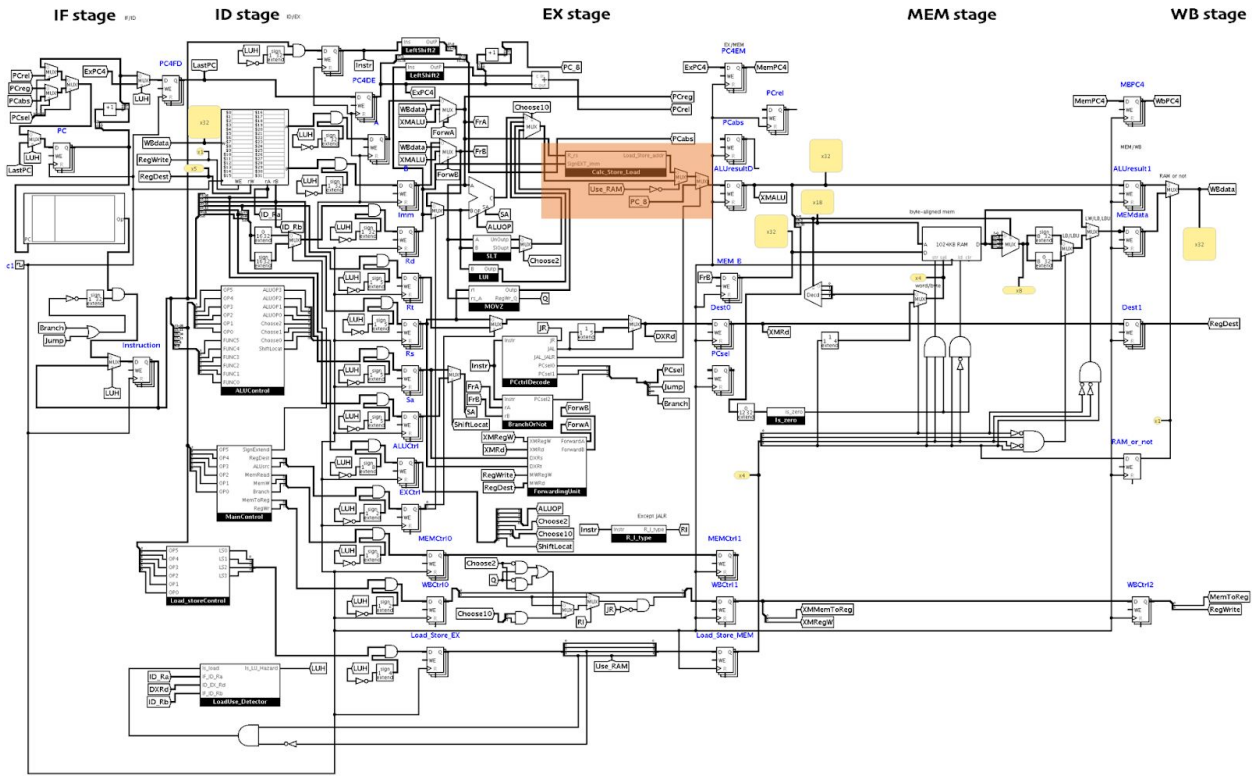
2.3.5 MOV unit

The MOV unit takes care of both MOVZ, MOVN instruction by inverting one result to get another. The inverting unit is controlled by the control signal “Choose2”. The MOV unit might change the “RegWrite” from 1 to 0 because the value is not desired, we do not want to write. This is controlled by the signal “Choose10” ().



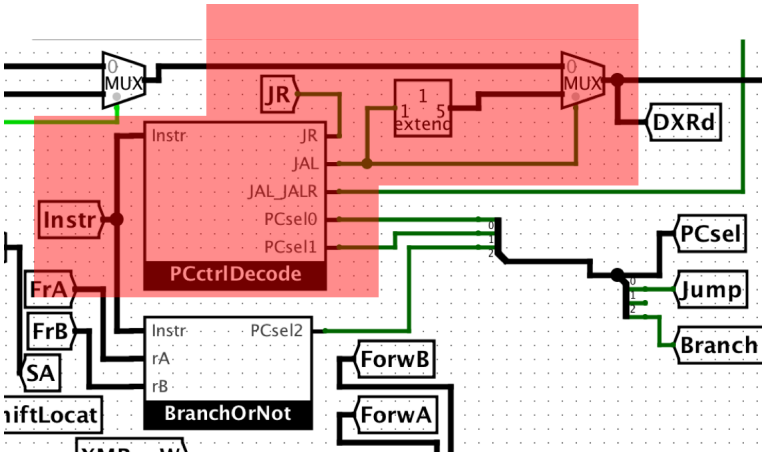
2.3.6 Calculate for Load or Store offset address

The calculation unit “Calc_Store_Load” for load and store is implemented by simply using an adder to add R[$Rs] and SignEXT(imm). A mux is then used to choose between the original ALU result and the result for Load or Store by using LS3, a control bit defined earlier and labeled as “Use_RAM” here.

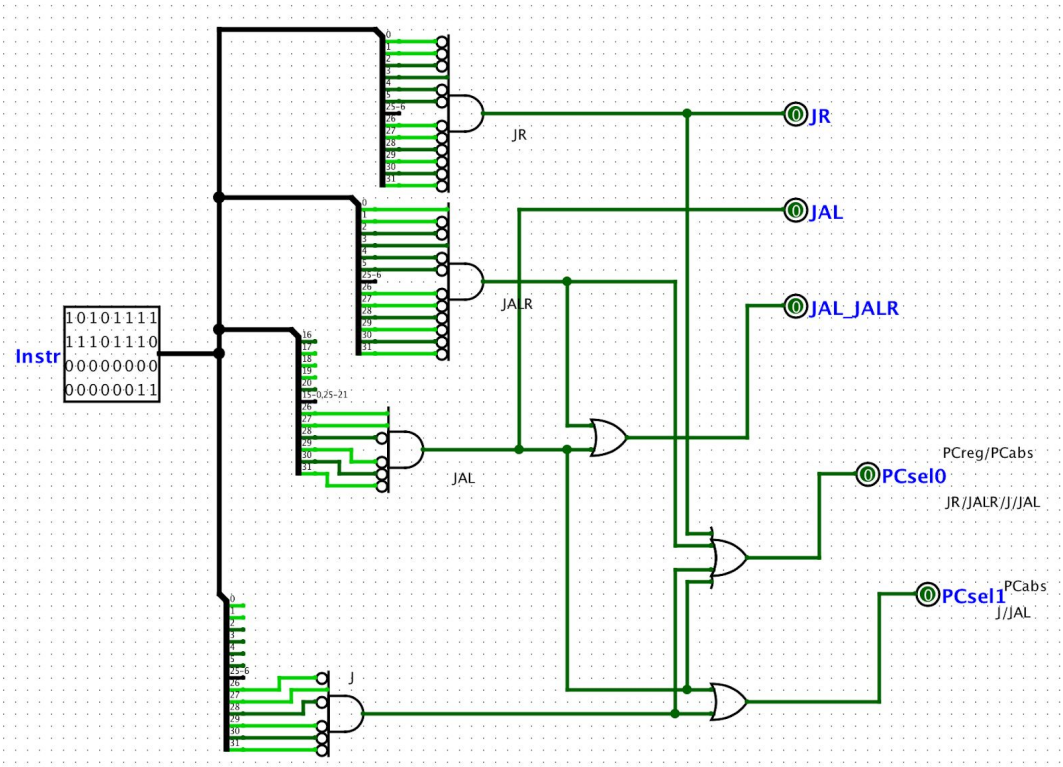


2.3.3 Jump Signal Control

The jump signal is passed from the Execution stage to the Fetch stage. The control signal involving jump includes PCsel [0] and PCsel [1]. (PCsel [0] represents whether we are using [Jump type] or [Branch or normal] type. PCsel [1] represent whether we are using [J or JAL] or [JR or JALR] type.) So detecting process is simply inspecting if instruction match. When jump is valid, these control signal will zap the current fetching instruction to NOP.

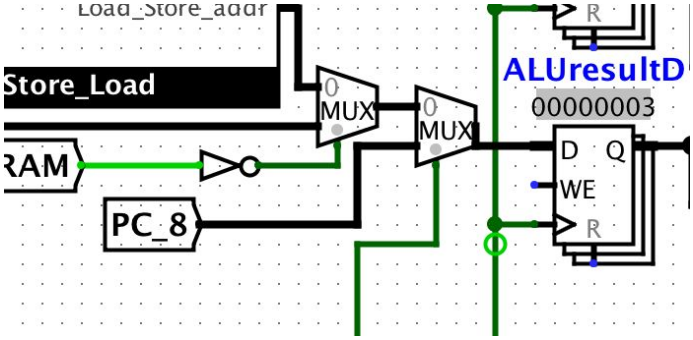


The jump signal decode is just detecting which jump category current instruction falls in. So we can connect proper PC address and decide whether to execute link step or not.



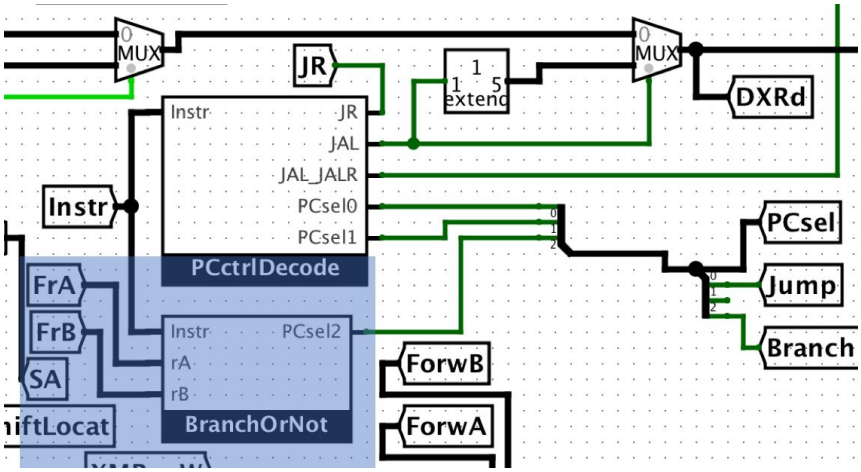
2.3.4 Save JAL and JALR Link Address

When executing JAL and JALR we need to save the PC + 8 address to a designated address. To do so, we just replace the ALUresult which is the data to write with our PC + 8 data. Meanwhile we replace the write destination to 31 when JAL runs. We use the Rd field when JALR runs.

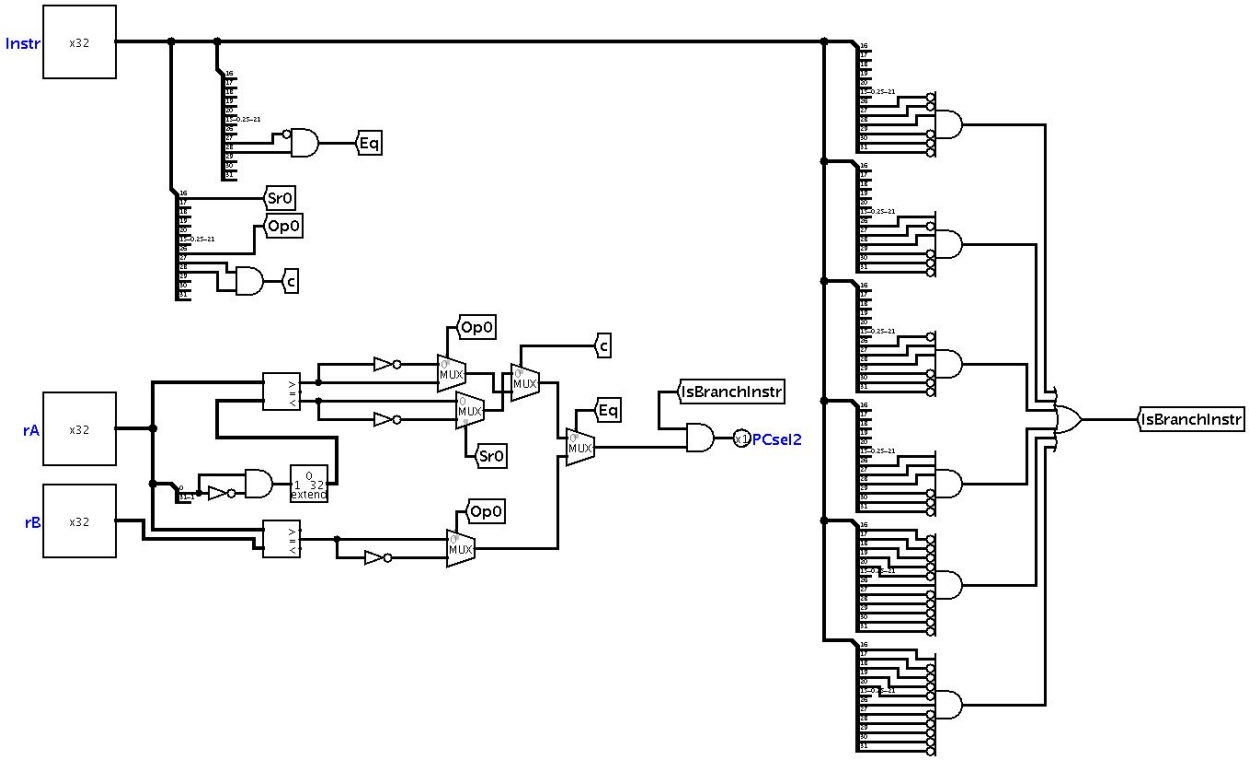


2.3.5 Branch Signal Control

The branching signal is also passed from the Execution stage to the Fetch stage. The control signal involves PCsel[2], which represent whether [non-Jump-Branch-type or Branch but fail] or [Branch success]. In the normal non-Jump-Branch-type and Branch but fail cases, we just need to keep passing PC + 4 as normal. When branch is valid, this control signal will zap the current fetching instruction to NOP.



The control signal decode has two parts. One is detecting whether this is a branch instruction or not. The other is detecting whether the branch condition is true or not.

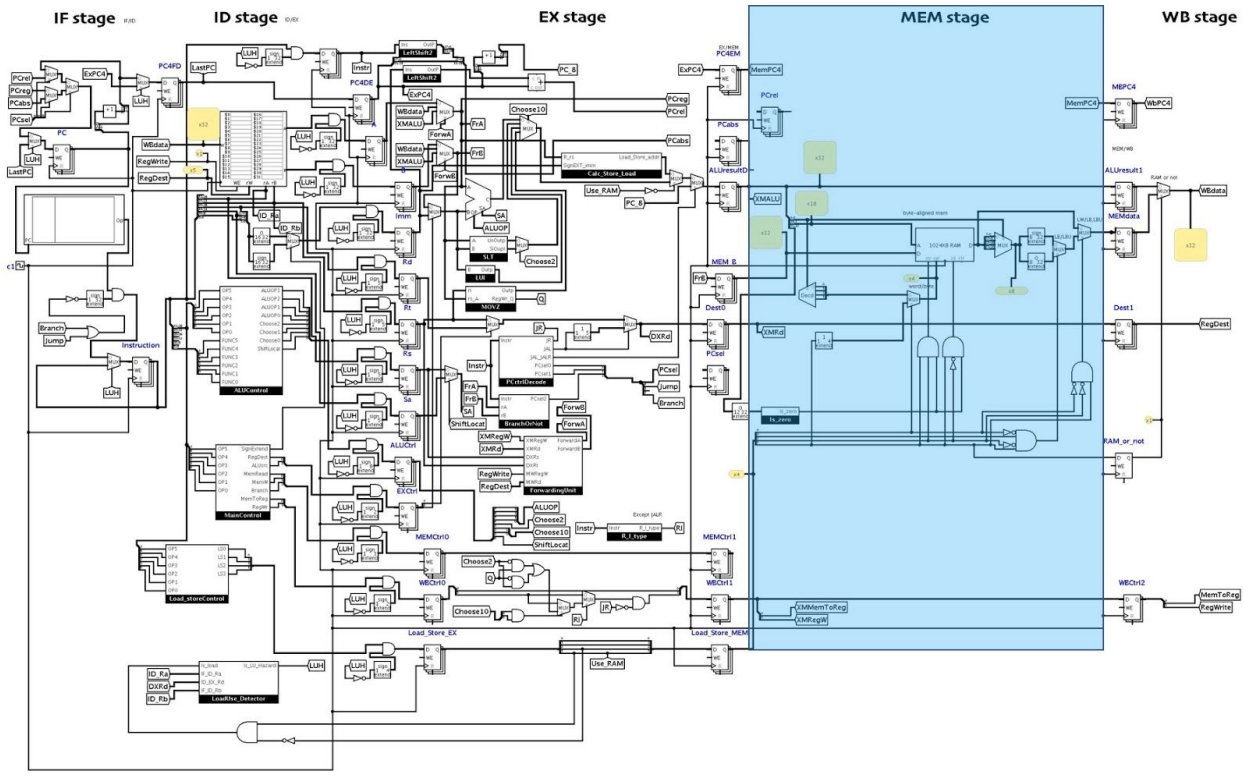


2.3.6 Calculation Units control

The OP control signal is defined by us to control the output of which calculation unit to use. The OP code is decoded in the main control to pass on to a MUX to do the selection job. The calculation units include the original ALU (provided in the class folder), SLT unit, LUI unit, and MOV unit by the signal “Choose10”. The last three unit is created for specific genre of instruction can be easily handled outside the original ALU as before.

2.4 Memory

A MIPS RAM unit is used at memory area for this stage. The RAM takes in 18 bits memory address (divided by four for byte-aligned addressing by right shifting by four), 32 bits memory contents from “MEM_B” pipeline register, 1 bit “store enable”, 1 bit “load enable”, and 4 bits selection control bits. It outputs 32 bits memory content for further selection dependent on the types of instruction. Store_enable and Load_enable are calculated and determined by the four bits Load_Store control defined earlier, passed from Load_Store_MEM pipeline register. The selection bits for load byte and load word is controlled by a 2x1MUX with selection bit from the third Load_Store control bits. When “word” is chosen (1111), the MUX will select output from extender. When “byte” is chosen instead, four bits output from decoder is selected. At the same time, since the output of RAM is 32 bits, the correspondent selected byte has to be “screened out” to be further extended. A 2x1MUX is used for this so that the selected position of the byte in a word is synchronized with output byte. In the end, the output is selected by using a MUX for LB and LW with selection bit calculated by using previously defined 4 bit load_store control bits.

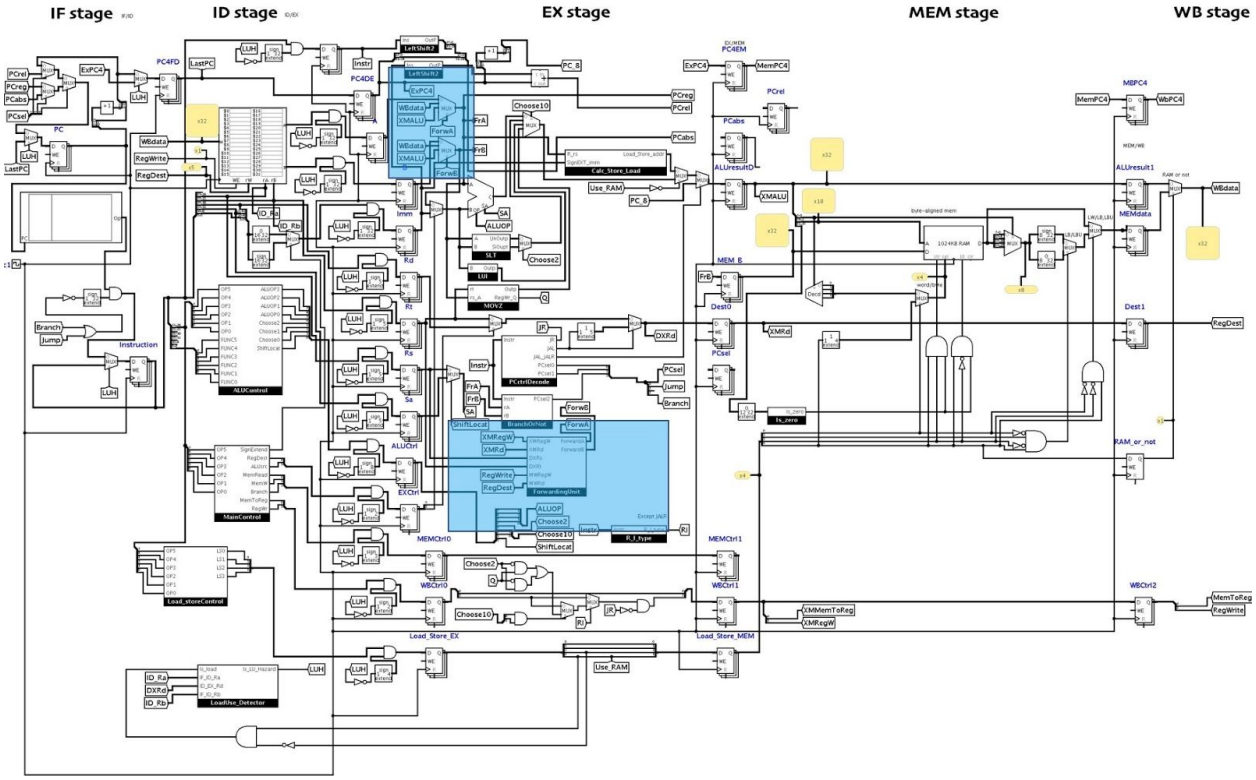


2.5 Write Back

In WB stage, we select which value to pass between ALU_result and MEM_data pipeline registers in MEM/WB. A mux with selection bit from RAM_or_not register is used. Whether to write to the register file or not (using “RegWrite”), and write to a certain register (controlled by “RegDest”) in the register file are used as projected 2 to pass signals back to regester file.

3 Hazard Control

3.1 Forwarding logic



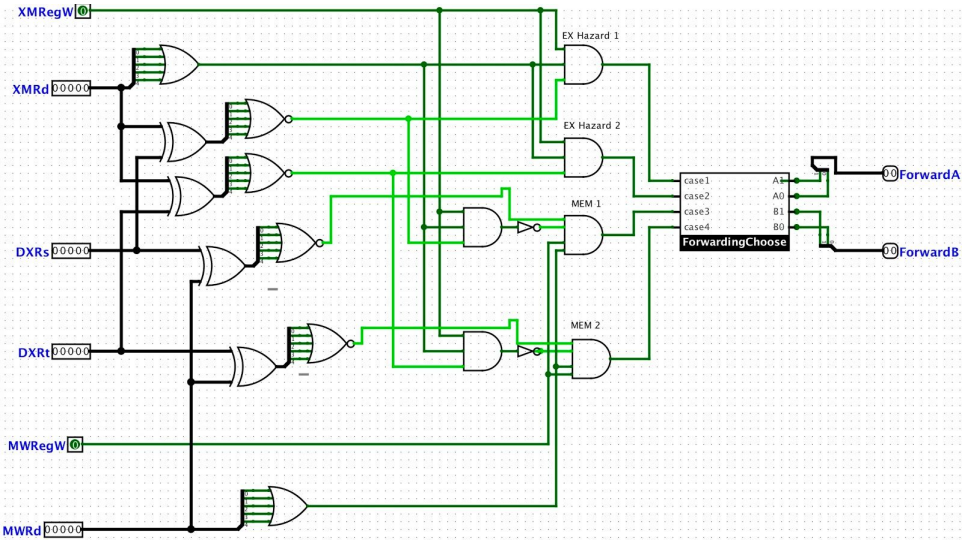
The forwarding logic is used to prevent data hazade by “forwarding” data, which has not been officially written into register file, from pipeline registers into needed places. In this case, the needed places will be the inputs for ALU or other computation units in execution stage.

To achieve this, a forwarding unit is implemented. The forwarding unit, marked red in above picture, takes in “Register Write” from EX/MEM pipeline register and MEM/WB pipeline register, Rd from EX/MEM pipeline register, and Register Destination from MEM/WB pipeline register. It output 2 bits “Forward A” and 2 bits “Forward B”. These four bits are used as control bits for two MUXs in the yellow area. These two MUXs decide whether A input and B input of the ALU and other computing units come from the register file, prior ALU result, or data memory or an earlier ALU result. The meanings of Forward bits are listed below:

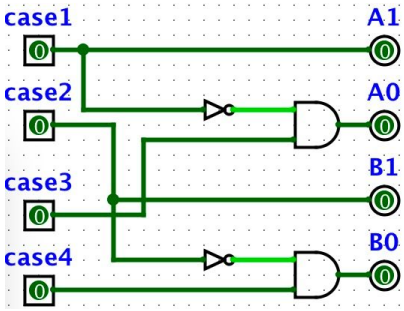
Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

The logic used to implement the forwarding unit is given from the project handout. There are four cases: two cases for EX/MEM pipeline stage, and two cases for MEM/WB stage. The boolean equations

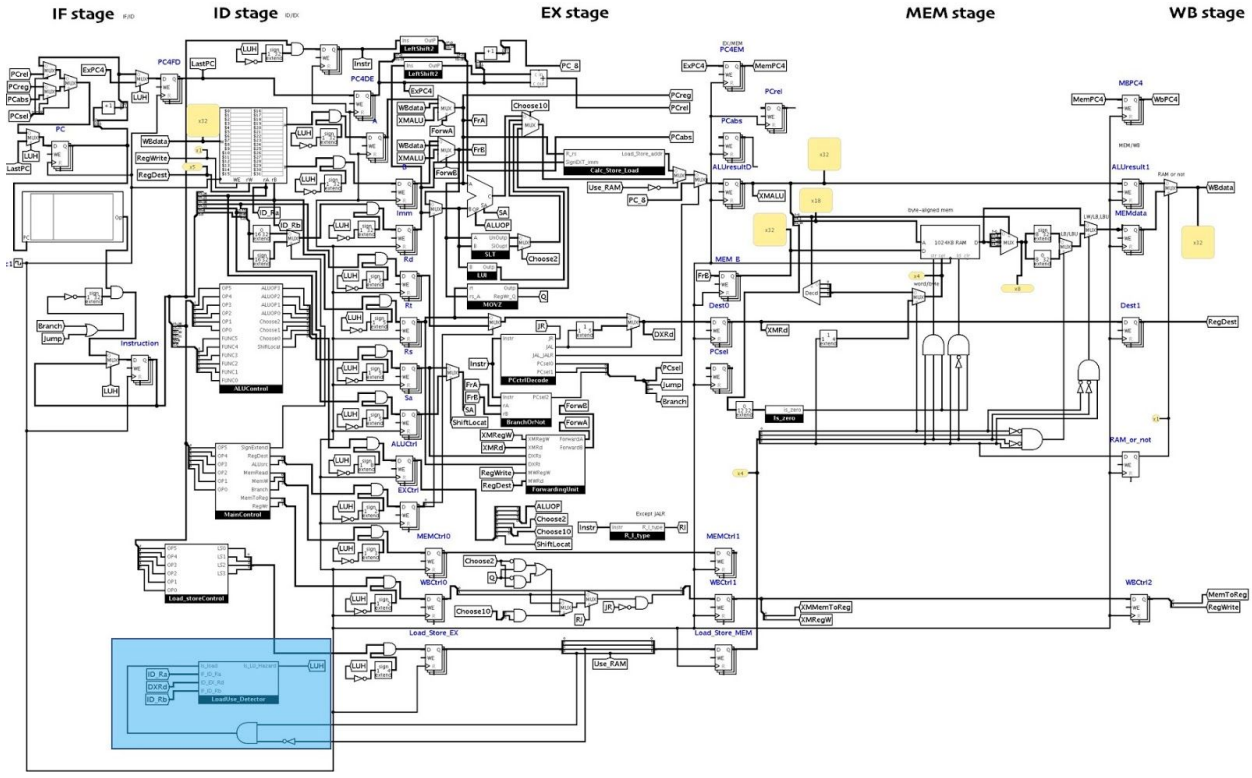
are listed in the project website, for the sake of space, we will not list it here. By using those equations, we can come up with the following circuit :



The subcircuit “Forwarding Choose” is simply taking the outcomes of each condition and maps them to two 2-bits desired outcomes. Truth table and Logism’s build-in function is used to implement this unit. The result circuit is the following:



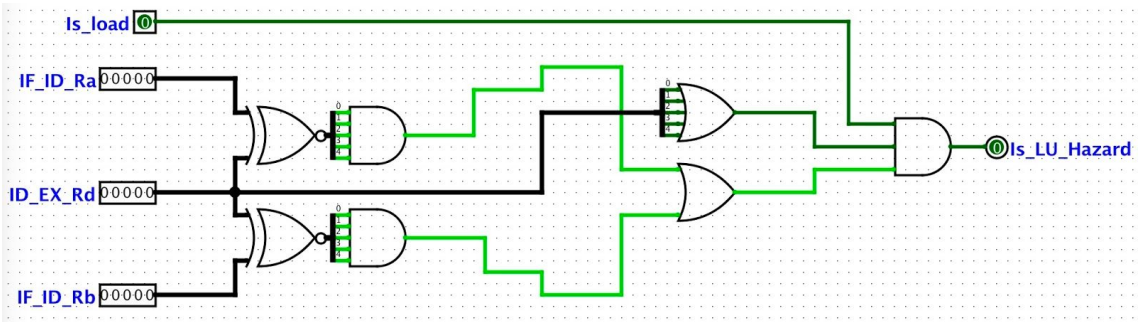
3.2 Load-Use Hazard (Stalling Logic)



Stalling are implemented to prevent load use hazard. The load use hazard detection applies the following logic:

if (ID/EX.Load == 1 AND (IF/ID.Ra == ID/EX. Rd OR IF/ID.Rb == ID/EX.Rd) AND ID/EX.Rd != 0)

Based on this formula, a subcircuit unit LoadUse_Detecter is implemented:



3.3 Zap/Flush and Delay Slot

When the above condition is satisfied, this subcircuit will output 1. Consequently, all pipeline registers in ID/EX stages will be flushed and PC will be stalled. Zap and Flush are implemented by muxing the original instruction with a nop. Whenever a branch is taken, the instruction after the delay slot will be zapped.

4 Testing

To ensure the correctness of our processor, we tested all the instructions and possible hazards holistically. Among them are all instructions in Table A and Table B, data hazard, load-use hazard, and control hazard. In the end, we used all three different implementations of fibonacci program to test mixed possible situations for hazards.

All the test cases are labeled and commented. For more detailed information, please refer to test file.