32-Bit Fully Pipelined MIPS

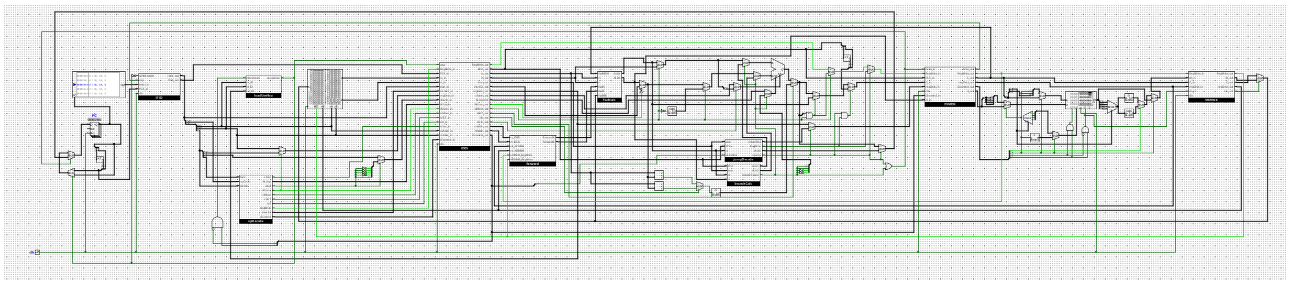
Rachel Nash and Jack Thompson

October 23, 2017

1. Introduction

This is an implementation of a subset of the MIPS32 architecture in Logisim that supports arithmetic/logic instructions, jumps and branches with one delay slot, and little endian memory loads/stores. It uses forwarding and stalling to avoid hazards. The five stages of the circuit are Instruction Fetch, Instruction Decode, Execute, Memory, and Write Back.

2. Overview



Pipeline registers separate every stage so that information can only go through one stage per clock cycle.

IF: Instruction Fetch

- Accesses the current instruction from memory
- PC (stored in a register) is incremented by 4
- Muxes check if there is a branch, jump, or stall

ID: Instruction Decode/Register File Read

- Instruction is decoded and control logic is passed on. Registers are read on falling clock edge
- Immediate numbers are sign-extended
- Load/use hazards are detected

EX: Execute/Address Calculated

- ALU computes the required operation for the instruction and outputs the result
- Forwarding logic for most hazards and branch/jump address calculation

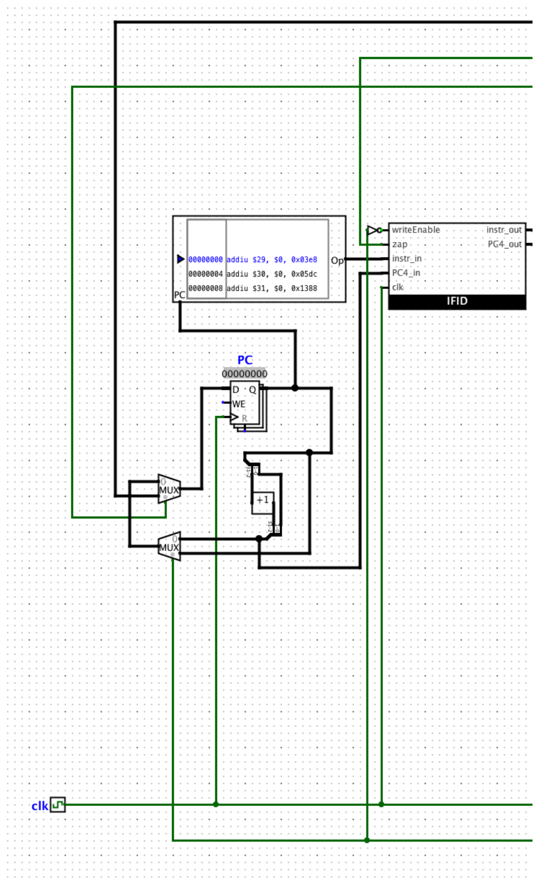
MEM: Data Memory Access

- If the instruction accesses memory, the data is read or written to memory
- Otherwise, ALU computation passes through

WB: Write back

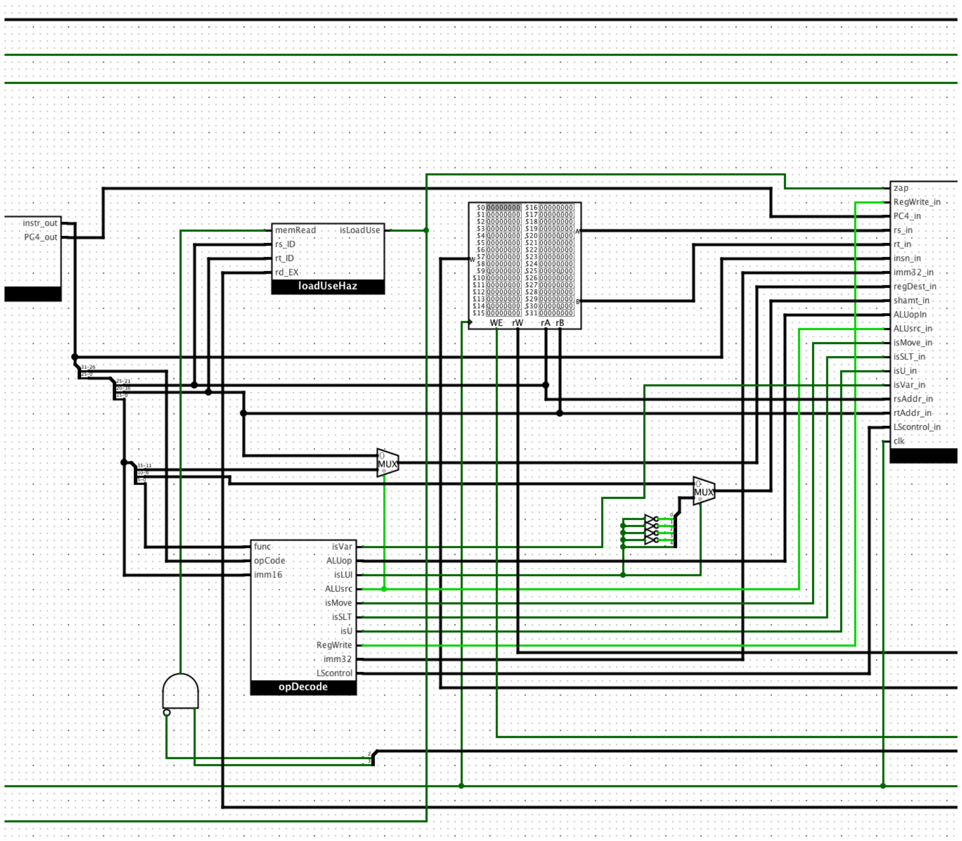
- Determines which data to write to a register (the data from Memory or from Execute stage)
- Writes this data to the designated register on rising clock edge

3. Instruction Fetch (IF)


00000000 addiu $29, $0, 0x03e8
00000004 addiu $30, $0, 0x05dc
00000008 addiu $31, $0, 0x1388
PC
Op
writeEnable
zap
instr_in
PC4_in
clk
instr_out
PC4_out
IFID
PC
00000000
D
Q
WE
R
MUX
+1
MUX
clk

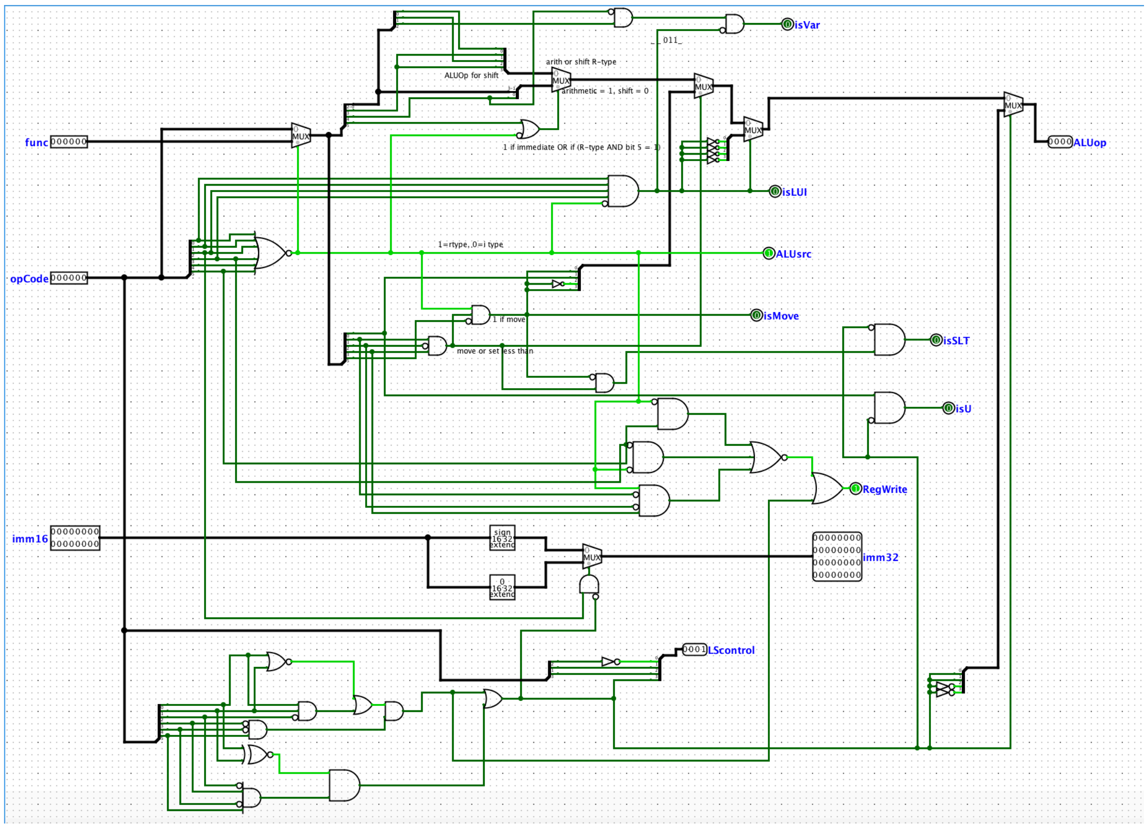
In this first stage, the PC (stored in a register) is used to fetch the next 32-bit instruction in the program ROM, which is then passed on. The PC prepares to read the next instruction at the next rising edge of the clock by adding 4. If there is a taken branch or a jump, we zap the instruction in IFID to create a nop and increment the PC accordingly. If there is a stall, we disable write enable to IFID and prevent PC from updating to PC+4.

4. Instruction Decode (ID)



This stage takes the 32-bit instruction from IF and breaks it up to find the read registers, destination register, shift amount, op code, and function code, depending on the instruction type. It reads from rA and rB in the register file and passes everything else through to the execute stage. Instructions are decoded and control bits are assigned. LScontrol is a 4-bit number that contains information about loads and stores. The sub-circuit loadUseHaz determines if there needs to be a stall, and if so, it zaps IDEX, prevents writing to IFID and stops PC from updating.

4.1 opDecode



This sub-circuit contains all of the logic to decode instructions and assign control bits. The outputs of this subcircuit are described in the table below.

Output	**Description**
ALUsrc	1 if R-type, 0 if I-type, used as control later
ALUop	4-bit op code to go into ALU
isSLT	1 if any type of 'set less than' instruction
isU	1 if SLT unsigned
isVar	1 if the shift is variable, 0 if not variable

RegWrite	1 if writing to register in the register file, 0 if instruction not supported yet
isMove	1 if a move instruction
isLUI	1 if LUI instruction
LScontrol	bit3: 1 if a load or store, bit 2: 1 if store, bit1: 1 if full word (not byte), bit0: 1 if signed

One of the most important controls is ALUsrc, which is 1 if the instruction is an R-type and 0 if it is I-type.

Logic is implemented to detect correspondences between the instructions and the 4-bit ALU operations for said instructions, producing the ALUop output.

For I-type instructions, this control input comes from the opcode field of the instruction. For R-type instructions, this control input comes from the function field of the instruction. R-type and I-type instructions are differentiated by the presence of 000000 in the opcode field. The 1-bit control line ALUSrc differentiates this, and is used in a mux to determine the 2nd ALU input.

In many cases, various bits in these 6-bit fields correspond to the same ALU operation. The table below shows the ALU operation inputs compared to the instructions that use these operations.

ALU Operation	Function	R-type (func field)	I-type (opcode field)
000x	Shift left	SLL: 000000 SLLV: 000100 LUI: 001111	
001x	Add	ADDU: 100**001**	ADDIU: 001**001**
0100	Shift right logical	SRL: 000010 SRLV: 000110	
0101	Shift right arith.	SRA: 000011 SRAV: 000111	
011x	Subtract	SUBU: 100**011**	
1000	And	AND: 100**100**	ANDI: 001**100**
1010	Or	OR: 100**101**	ORI: 001**101**
1100	Xor	XOR: 100**110**	XORI: 001**110**

1110	Nor	NOR: 100**111**	
1001	Eq	MOVZ: 001010	
1011	Ne	MOVN: 001011	
1101	Gtz		
1111	Lez		
		SLT: 101010 SLTU: 101011	SLTI: 001010 SLTIU: 001011

For R-type instructions starting with a 1 in their function field, bits 0-3 in the function field correspond to bits 1-3 in the ALU operation input. For all I-type instructions, the same rule applies with bits 28-26 in the opcode field. This is displayed in bold on the table. A 0 was appended to these corresponding bits in the instruction to produce the ALU operation.

One discrepancy from this trend is the shift instructions. However, there are some trends for these functions as well:

Bit 29: 1=LUI (shift 16 bits) 0=other shifts

Bit 30: 1=variable shift amount, 0=shift amount from shamt field

Bit 31: 1=right shift, 0=left shift

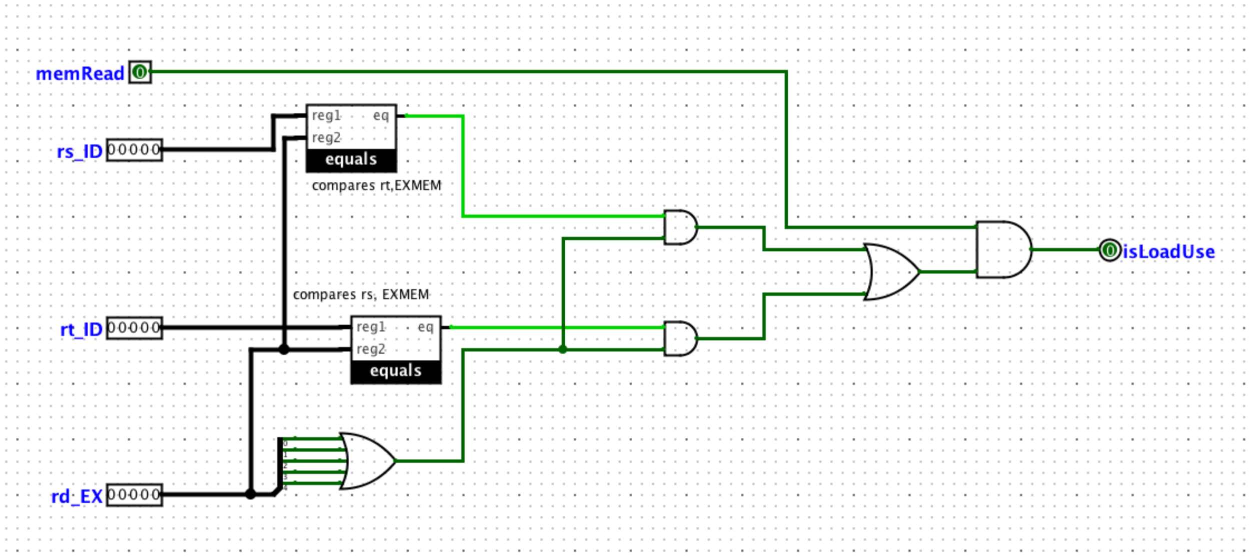
Bit 32: 1=arithmetic, 0=logical,

Additional logic is implemented for move instructions, which use the eq and ne operations of the ALU.

Combining these values in a control circuit allows us to control the ALU to perform the desired operation.

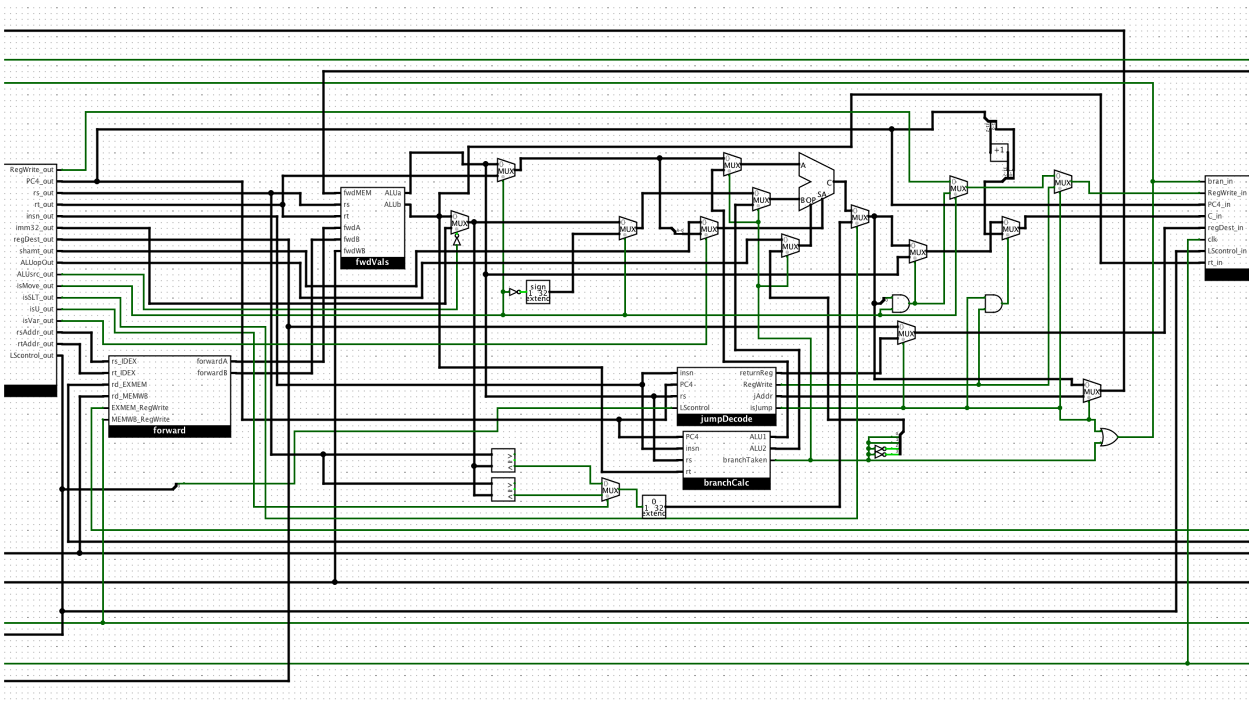
The “set less than” functions do not access the ALU, but instead are compared via either a signed or unsigned comparator. So, an isSLT control signal is extracted from these instructions as well as isU to determine which comparator to use.

4.2 loadUseHaz



This simple sub-circuit determines if there is a load-use hazard (if the instruction currently in execute stage is going to load a value from memory that is needed for the instruction currently in decode). When isLoadUse is 1, the instruction in decode is stalled once and a nop bubble is passed through to the next stage.

5. Execute (EX)



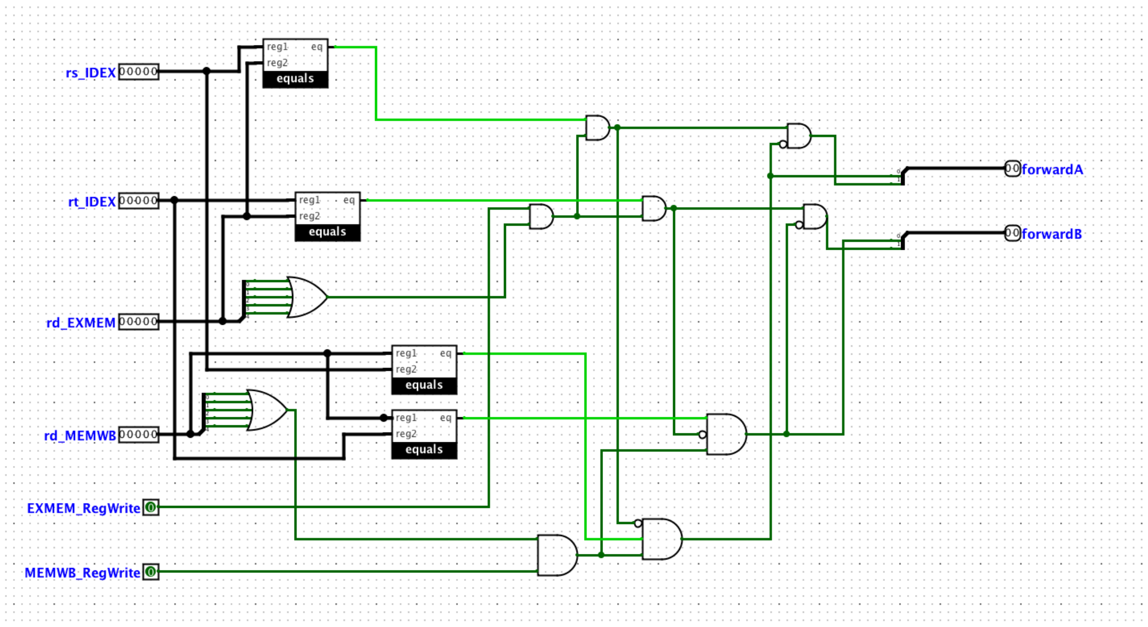
This stage performs any necessary calculations, primarily using the ALU. We pass in the ALU opcode determined in opDecode, and either put target register values into the ALU, load an immediate into the upper 16 bits, or run register values through either a signed or unsigned comparator (for SLT). For moves, a 0 is sign-extended to 32 bits and put in the ALU to be compared to rt and regWrite is turned off if the condition is not met. The jumpDecode and branchCalc sub-circuits find the address the PC is set to for these instructions.

A forwarding sub-circuit (described in detail next) determines if there is a hazard requiring values to be forwarded from MEM or WB. Control of forwardA is shown below and forwardB is the same. Two muxes in the main EX stage are used to choose between 3 possible inputs.

forwardA	A input in ALU

00	rs
01	forward from WB
10	forward from MEM

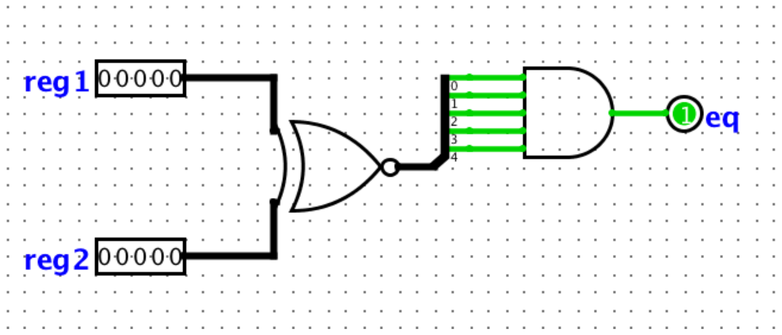
5.1 Forward



Within this sub-circuit of Execute, the destination registers in the Memory and Write Back stage are compared to registers being used in current operations to see if there are any hazards. We use a simple equals sub-circuit to check if they are the same. Outputs are used as control signals to choose ALU inputs. The output forwardA is for the A input of ALU and forwardB is for the B input. The logic is as follows:

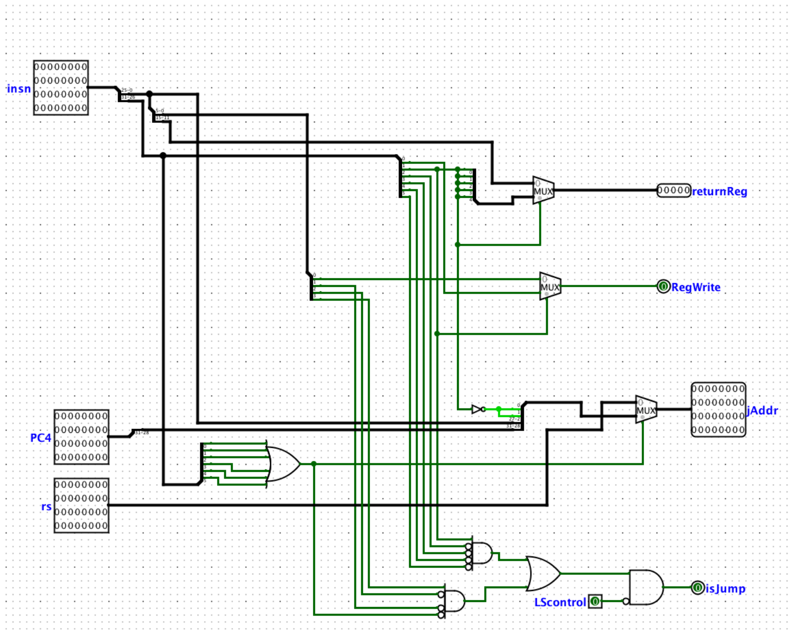
```
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd == ID
/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd == ID
/EX.RegisterRt)) ForwardB = 10

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)

  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)

    and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
  and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
    ForwardA = 01

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
  and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
    ForwardB = 01
```

5.1.1 Equals



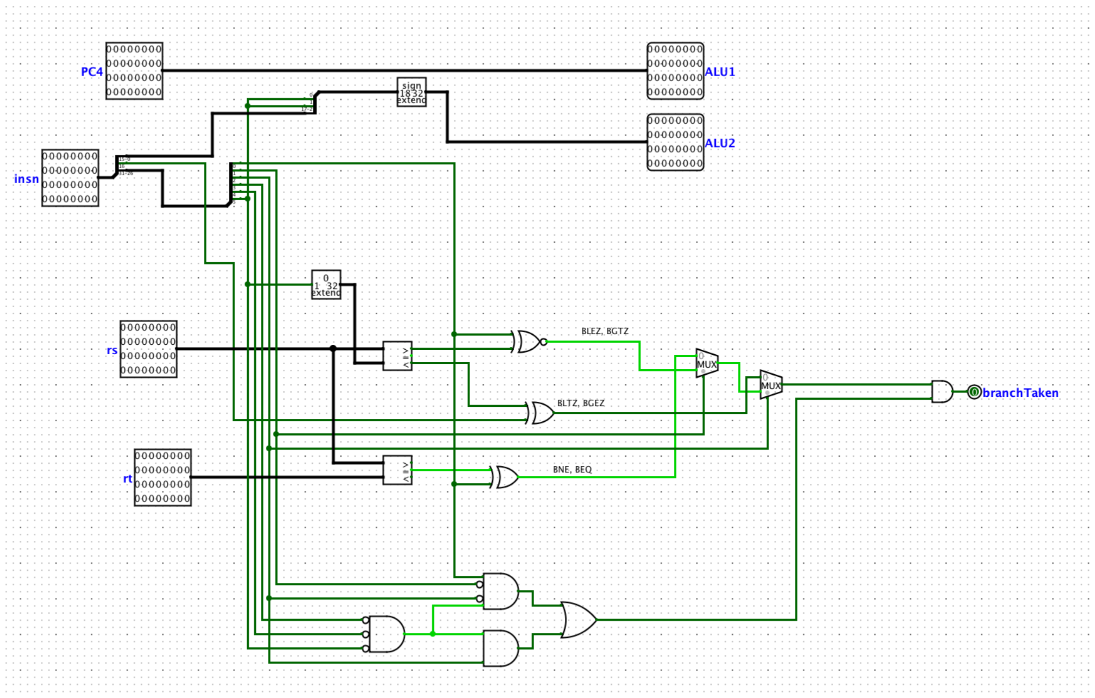
Shown is the simple subcircuit to compare 2 registers in order to detect a data hazard.

5.2 jumpDecode



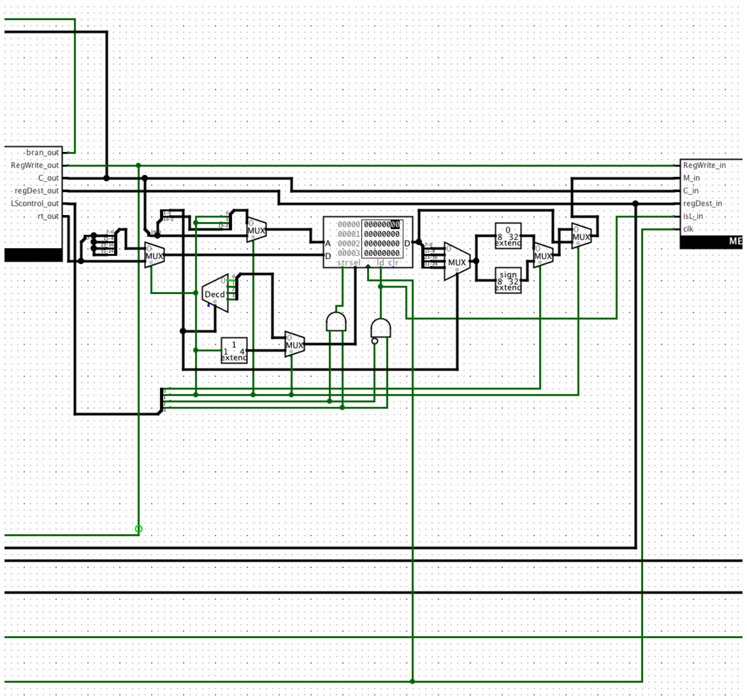
This sub-circuit works with jump instructions. It outputs the address to jump to (jAddr), decides whether or not a return address is saved in a register (with RegWrite), and determines if the instruction is a jump (with isJump).

5.2 branchCalc



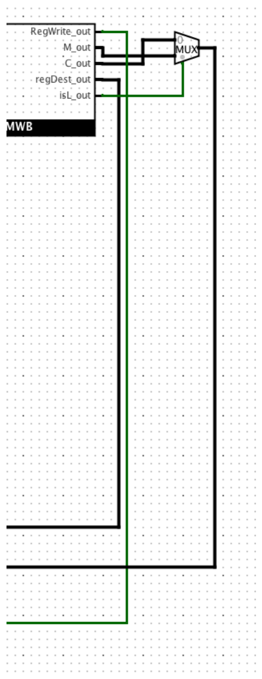
If there is a branch instruction, we use comparators to check if it is taken and we find the inputs to the ALU which are PC+4 and the 16-bit signed offset shifted left 2.

6. Memory



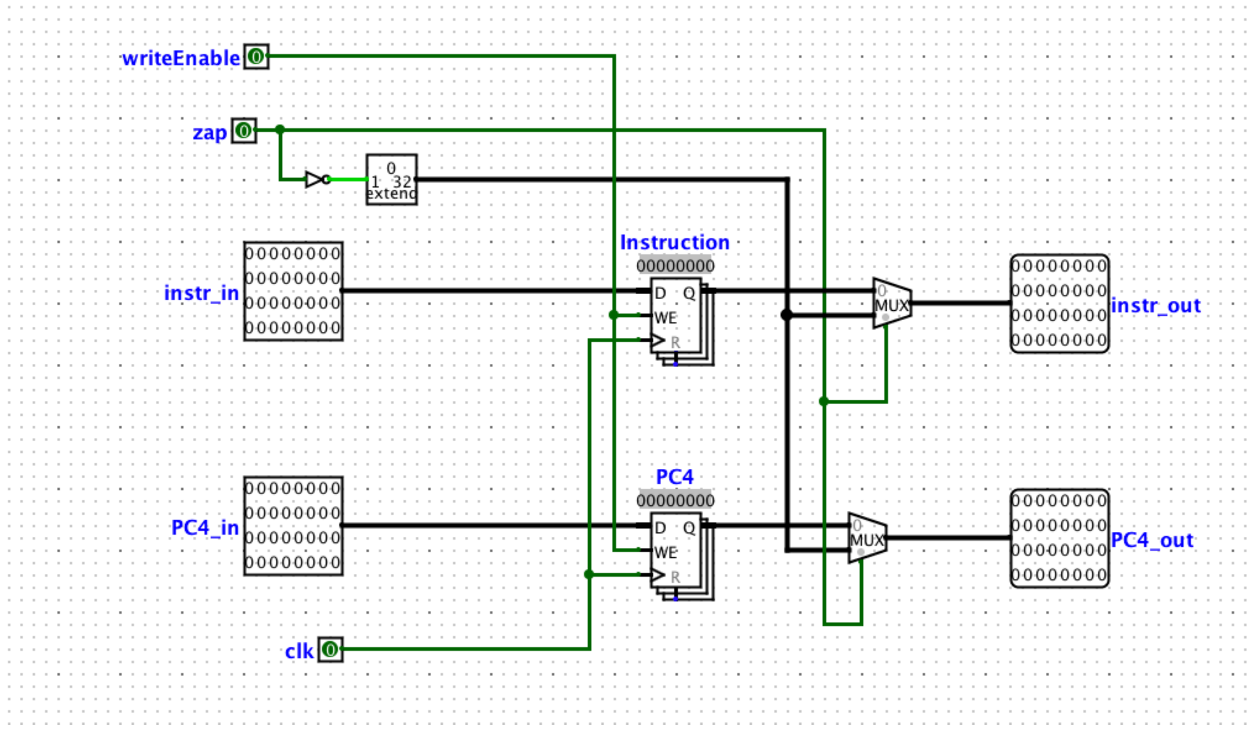
The LScontrol that has been passed through (see opDecode sub-circuit) can now be used to either load or store in memory. For accessing bytes, the address input is shifted right 2 to find the word address and bits 0 and 1 from the original input (I will call these the byteControl) are used to select the byte using a decoder. Full word access turns every bit of RAM's sel input on (selecting all bytes). Then after going through RAM, for a load, muxes choose between the full word value (LW) and a sign-extended (LB) or zero-extended byte (LBU). A splitter and mux determines which byte is put through using the same byteControl from before.

7. Write Back (WB)



The simplest stage, write back writes either the data from memory (M) or output of the ALU (C) to the register file in the ID stage on the falling edge of the clock if RegWrite is on.

8. Pipeline Registers (Example: IF/ID)



This is an example of one of our pipeline registers, IF/ID. The others (ID/EX, EX/MEM, and MEM/WB) are all structurally similar, with different inputs and outputs, but they all write useful values to registers under control of the clock. IF/ID is slightly different because it can stall and zap instructions. When a stall is detected, write enable to IF/ID pipeline registers is disabled in order to prevent the instruction from being passed forward. When we detect that we need to perform a zap, we set the instruction to all 0s using a mux.

A table explaining each value in the registers is here for clarity.

ID/EX			**EX/MEM**	
zap	1 if there is a stall, causes nop to be passed through		bran	1 branch/jump is taken
RegWrite	1 if writing to register		RegWrite	1 if writing to register
PC4	Value of PC of instruction +4		PC4	PC+4
rs	value in register designated by rs field of instruction		C	result from execute
rt	value in register designated by rt field of instruction		regDest	destination register
insn	32-bit instruction		LScontrol	4-bit control signal for loads/stores
imm32	32-bit immediate		rt	the value in rt
regDest	destination register		**MEM/WB**	
shamt	shift amount		RegWrite	1 if writing to register, 0 if instruction not supported yet
ALUop	4 bit op for ALU		C	result from execute
ALUsrc	1 if R-type, 0 if I-type		M	result from memory
isMove	1 if move instr		regDest	destination register
isSLT	1 if any type of set less than		isL	1 if a load
isU	1 if SLT is unsigned			

isVar	1 if Shift left variable			
rsAddr	the address of the rs register			
rtAddr	the address of the rt register			
LScontrol	4-bit control signal for loads/stores			

10. Testing

We manually generated assembly code tests for our mini-MIPS. We started with testing every instruction type in Table A with no dependencies. Then we tested our Fibonacci Iterative, Recursive, and Memoized programs. We tested all branches when they are both taken and not taken, all load and store instructions with different offsets, and all jump instructions going forwards and backwards in the program. Then, we checked the following hazards: EX/MEM, MEM/WB, Double, Triple, Single False, Double False, Triple False, 1-cycle False, 2-cycle False, 3-cycle False, and Load-Use Hazard.