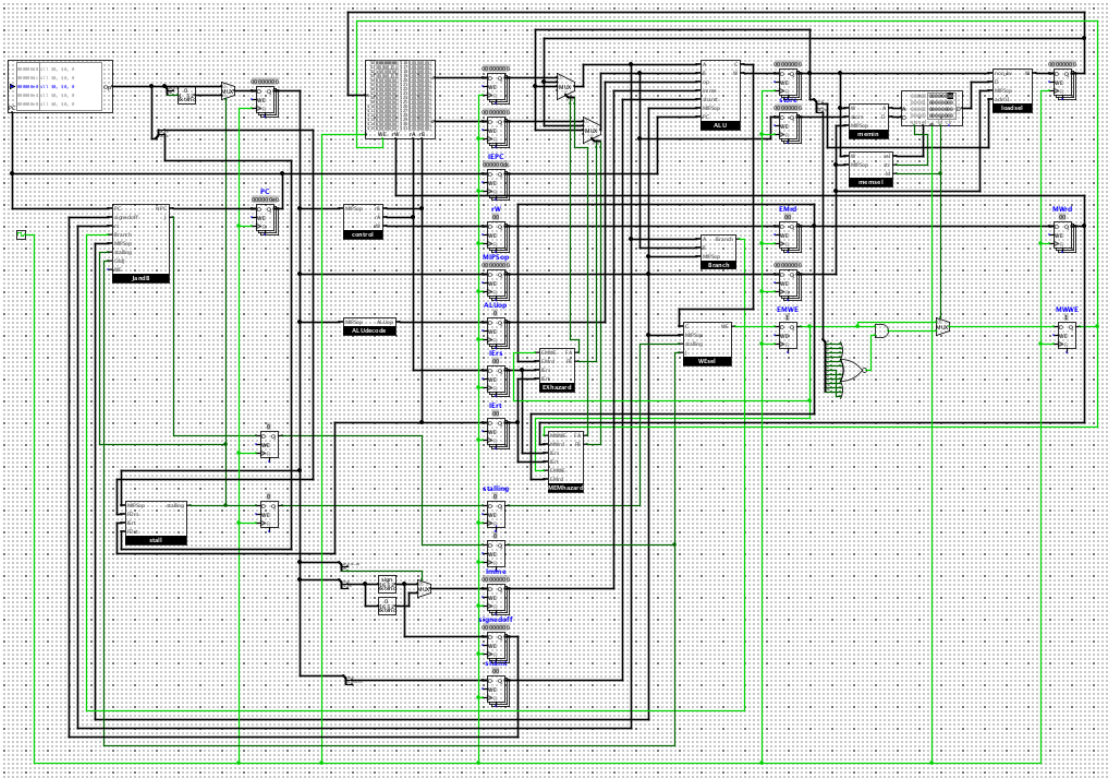
Design Documentation for MIPS32

1. Introduction

This Logisim circuit builds a fully-pipelined MIPS32 processor. MIPS is a commonly-seen processor architecture.

This design document will discuss how we build up the processor in detail, including description of sub-circuits and the logic for building each part. Our design makes reference to the example MIPS processor given in lectures and Zybook.

2. Overview



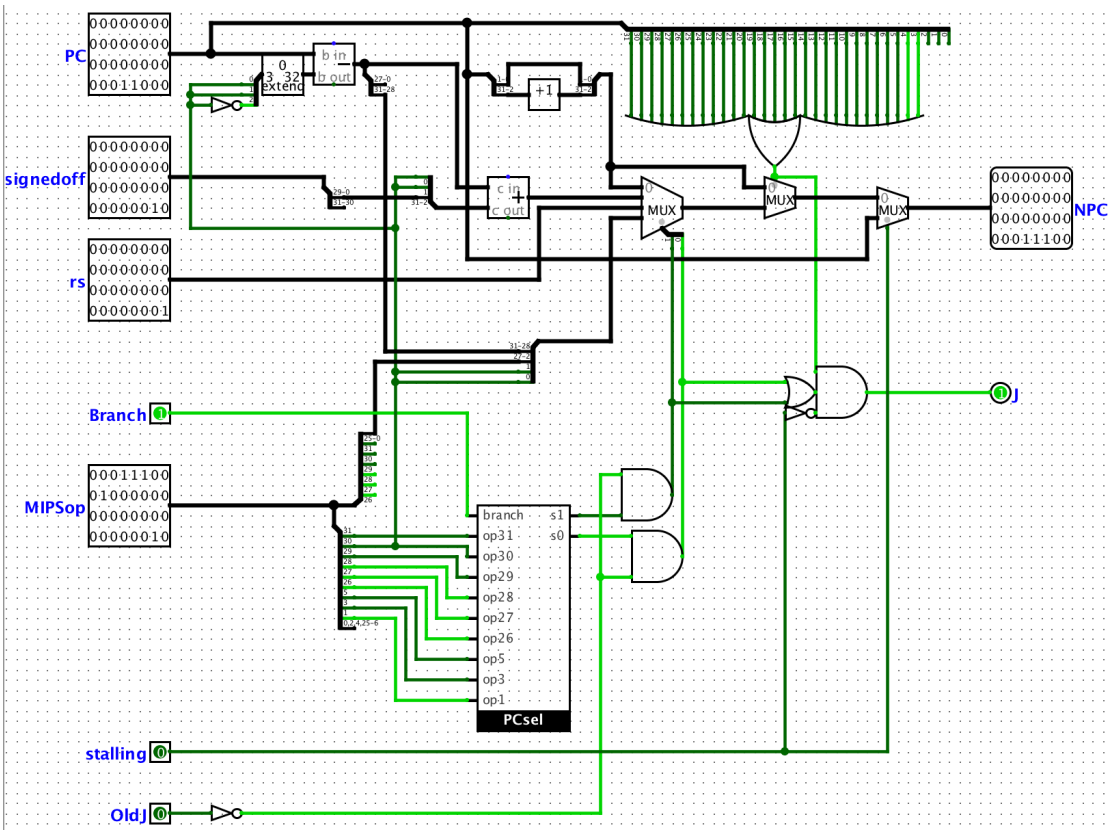
The MIPS32 has several major parts. The program ROM on the top left corner loads the 32-bit opcode and computes the program counter address. Each instruction adds PC by 4, as achieved by shifting the 3rd bit with an incrementer. The 32-bit opcode is then delivered to the next few parts: control, which is used to read different register addresses from the opcode, ALUdecode, which is used to convert the funct code of the 32-bit value to the 4-bit ALU opcode. The sign extenders are used to extend zeroes for immediate instruction, and the mux connecting shamt is used to get the shift amount. The next stage is execution, as encoded in the sub-circuit ALU. This circuit gets in all the inputs decoded from the MIPS opcode and computes the corresponding results using ALU. The result is then written back to the register file, depending on the states of WE. The multiple registers achieve the purposes of a pipelined-processor. EXhazard

and MEMhazard are data hazard detectors.

The sub-circuits will be discussed in detail below.

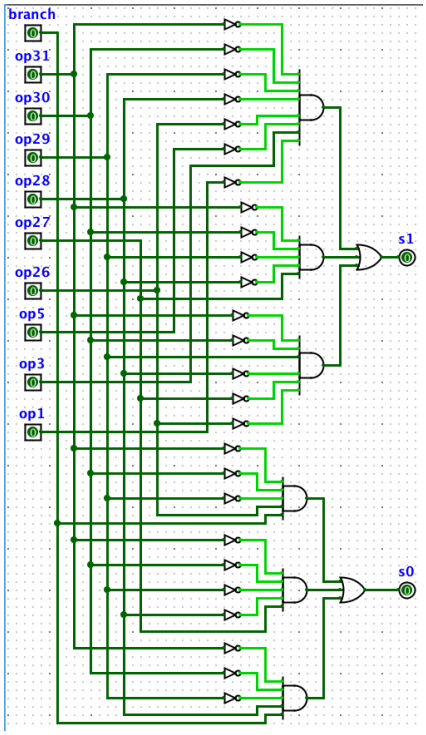
3. Stage I: Fetch

3.1 JandB



JandB determines what the next PC address it. Normally, there are four options: PC+4, target for J and JAL, register value for JR and JALR, PC+4+offset for branches. Based on MIPS opcode, the correct PC is chosen. This selecting signal is done by the subcircuit PCsel. However, there are 2 special cases to consider. The first one is when both the current and the second previous instruction are jump/branch. If the PC jumps elsewhere, the current jump/branch should not be executed, so PC+4 should be selected. This is done using the OldJ input, indicating the second previous instruction made the PC jump elsewhere. The second special case is when stalling due to load-use hazard. When there is a hazard, not matter what the operation is, PC should stay PC instead of PC+4 or other special PCs. This is done through the stalling input. When stalling signal a 1, the next PC stays the same.

3.1.1 PCsel



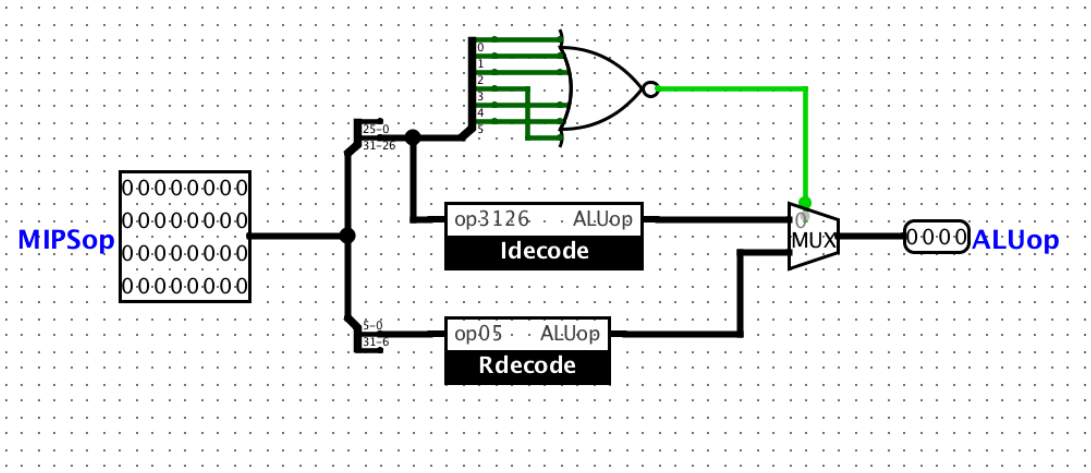
This is an automatically generated circuit. When the instruction is branch and the branch signal indicates that branch condition is met, s1 output 0 and s0 outputs 1, letting JandB selecting the correct PC. Similar results for regular instructions, jump, and jump register.

4. Stage II: Decode

The MIPS 32-bit opcode is delivered to the IF/ID stage and processed.

4.1 ALUdecode

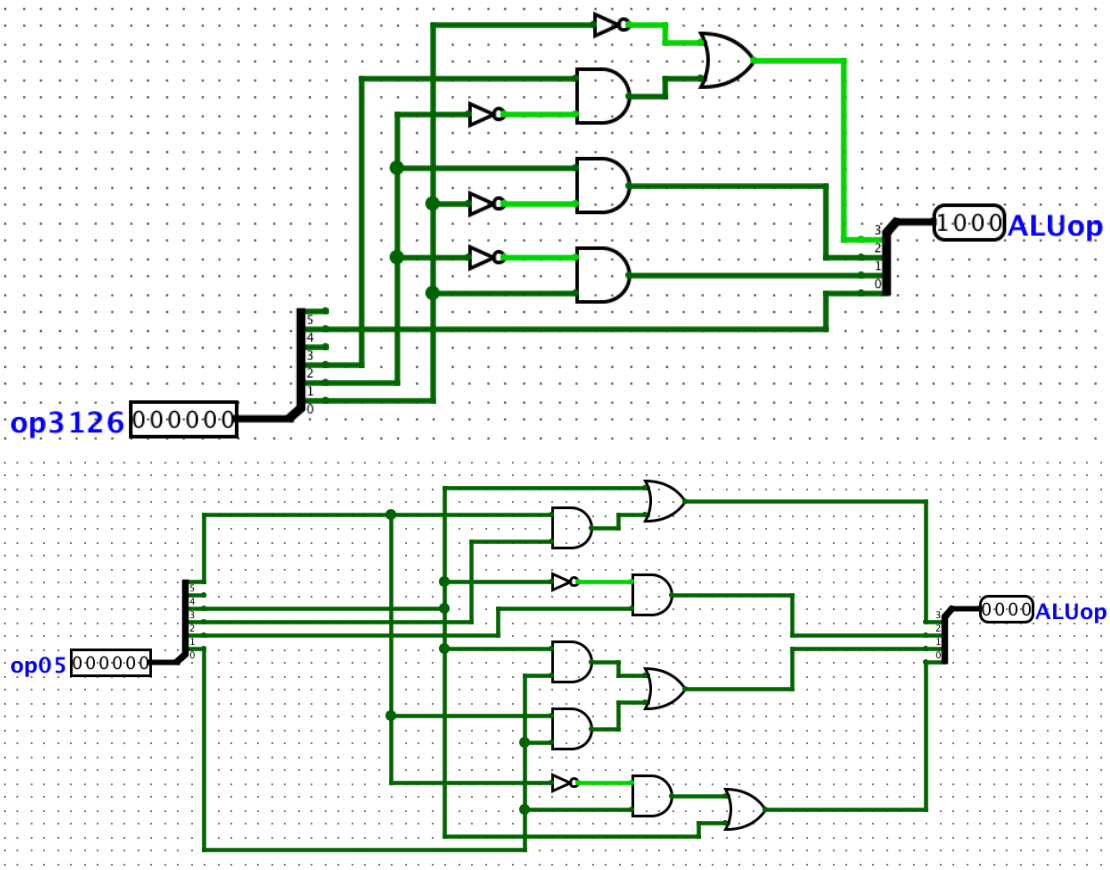
The ALUdecode takes in a 32-bit MIPS opcode and converts it to the 4-bit ALU opcode. Whether the six bits convert to R-type instruction code (Op[0-5]) or I-type instruction code (Op[26-31]) depends on Op[26-31]. If Op[26-31] are all zeros then it indicates that this is an R-type instruction, otherwise Op[26-31] are the 6-bit Opcode of an I-type instruction. This is detected by a 6-input NOR gate, and chosen with a mux.



4.1.1 Idecode and Rdecode

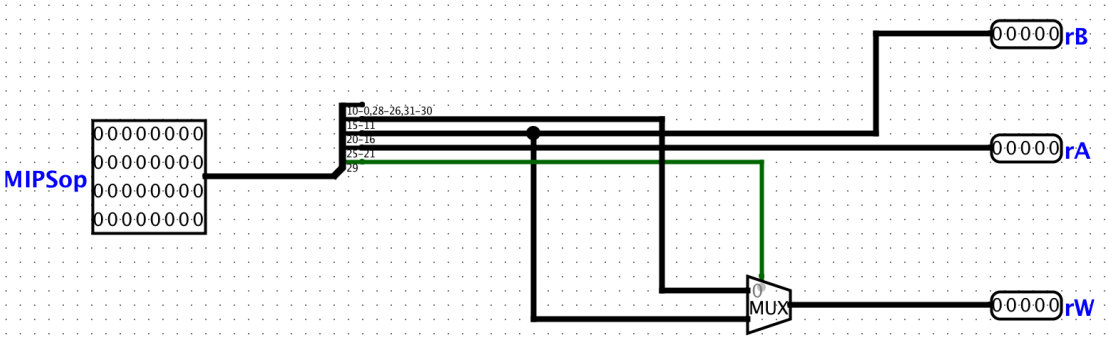
These two subcircuits are created using Logisim's combinational logic functionality. Given six inputs, four outputs and the corresponding truth table, the two circuits are auto-generated. They convert the 6-bit opcode to the 4-bit ALU opcode.

Note: ALUdecode does not handle instructions SLT, SLTU, SLTI, SLTIU as they are computed using comparators.



4.2 Control

This part further decodes the 32-bit opcode to compute the addresses of the three registers used in each instruction. rs[21-25], rt[16-20] are fixed and wired to rA, rB, respectively. rW depends on the types of operation. For I-type, it is [16-20], and for R-type, it is [11-15]. A MUX is used to select from these two values with the 29th bit of opcode as the select bit, for 1 indicates I-type and 2 indicates R-type.



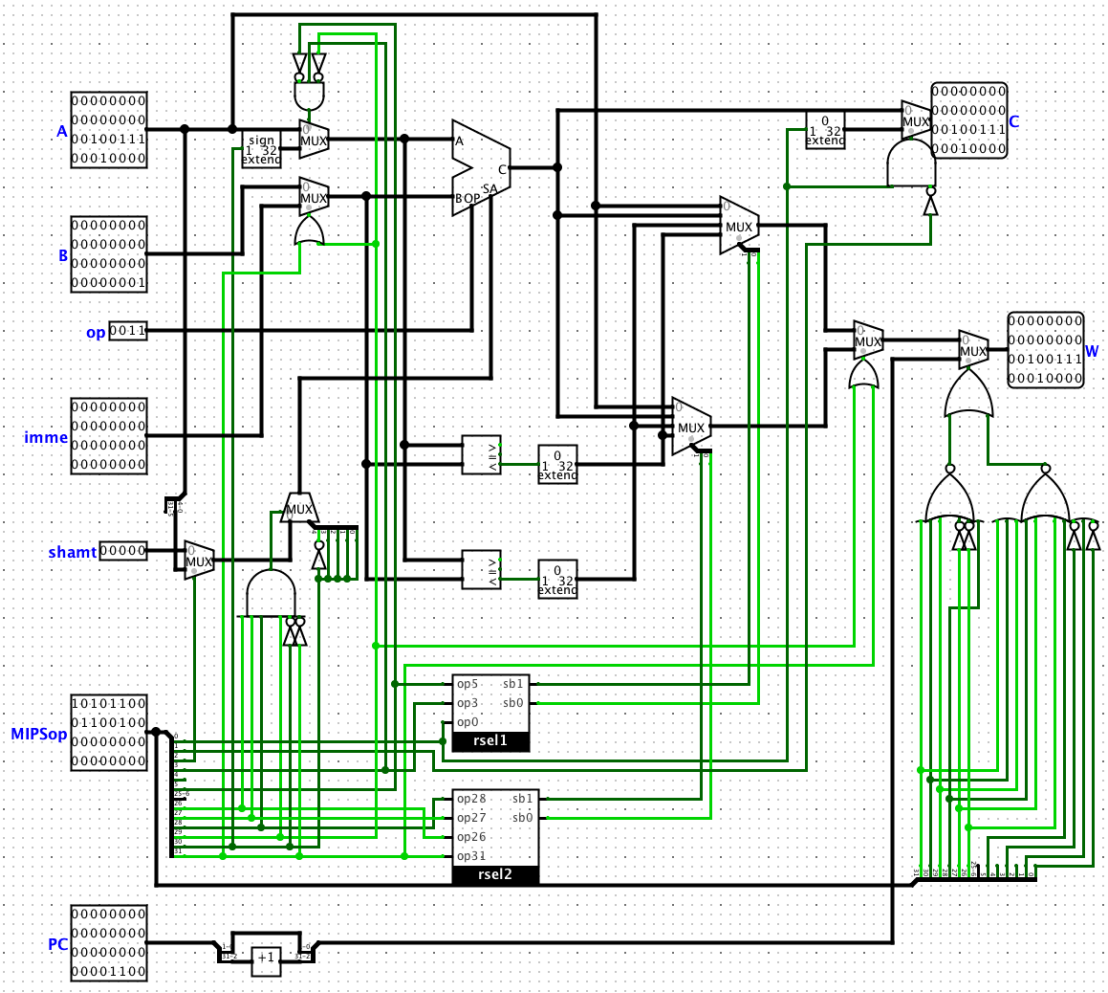
5. Stage III: Execution

The subcircuit Stage III performs the execution stage in a MIPS processor.

5.1 ALU

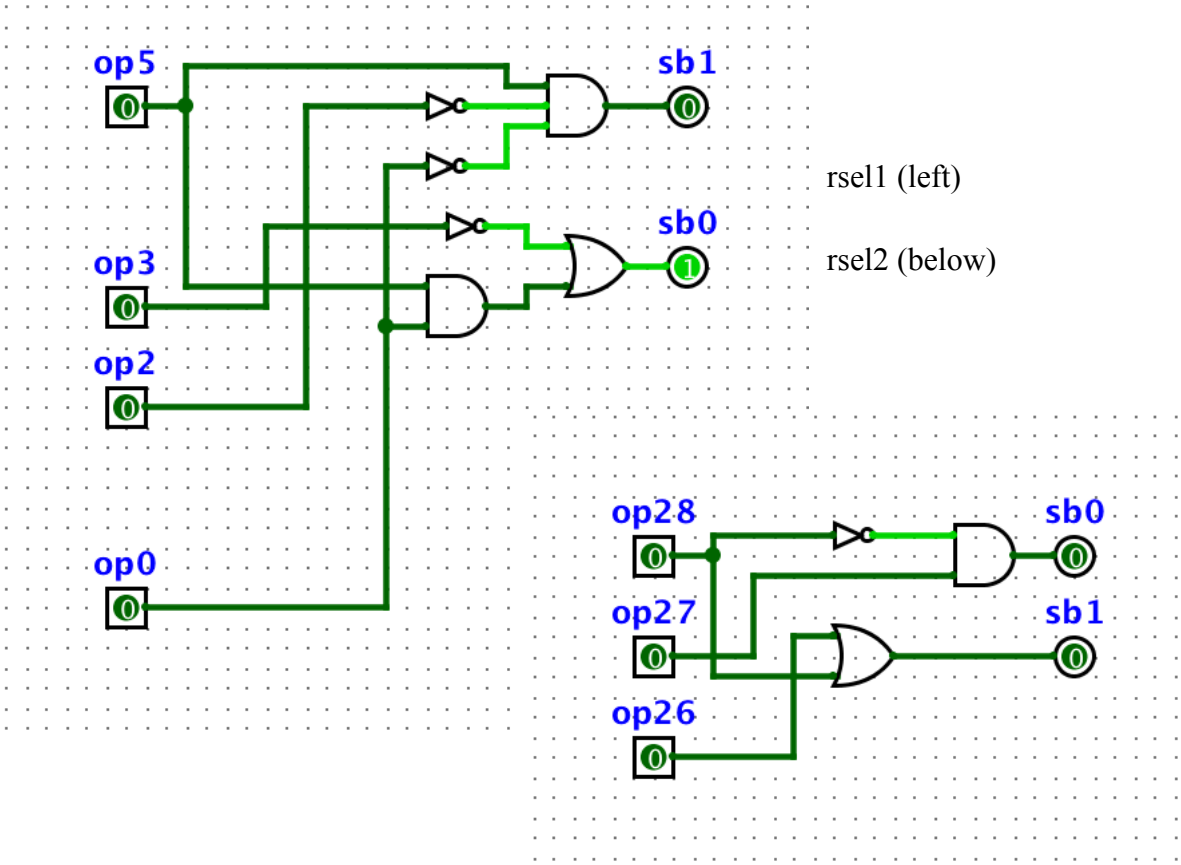
If ignore all the green (1-bit) signals which act as select bit for the MUXs, this subcircuit has three main parts: ALU, signed comparator, and unsigned comparator. All the operations have one of value A, ALU, signed comparator, or unsigned comparator as the final result, which will be written back to the registers. The value A going into the ALU and the comparators is chosen from input A (for operations other than MOVN, MOVZ) and hardcoded constants 0^{32} (for MOVN and MOVZ). The value B going into the ALU and the comparators is chosen from input B (for R-type operations) and immediate (for I-type operations). The op for ALU comes from the decoded opcode using subcircuit 3.1 ALUdecode from the previous stage. Shift amount SA is either from shamt (for operations other than LUI) or is the hardcoded constant 16 (for LUI). These choices are made by the left three MUXs using certain bits from the MIPS opcode. The two hardcoded constants also come from the 30th bit of MIPS opcode, as it is always 0 for the operations concerned in this project.

After the three main calculation units, three more MUXs are used to select where the right result is from. Details are explained in 4.1.

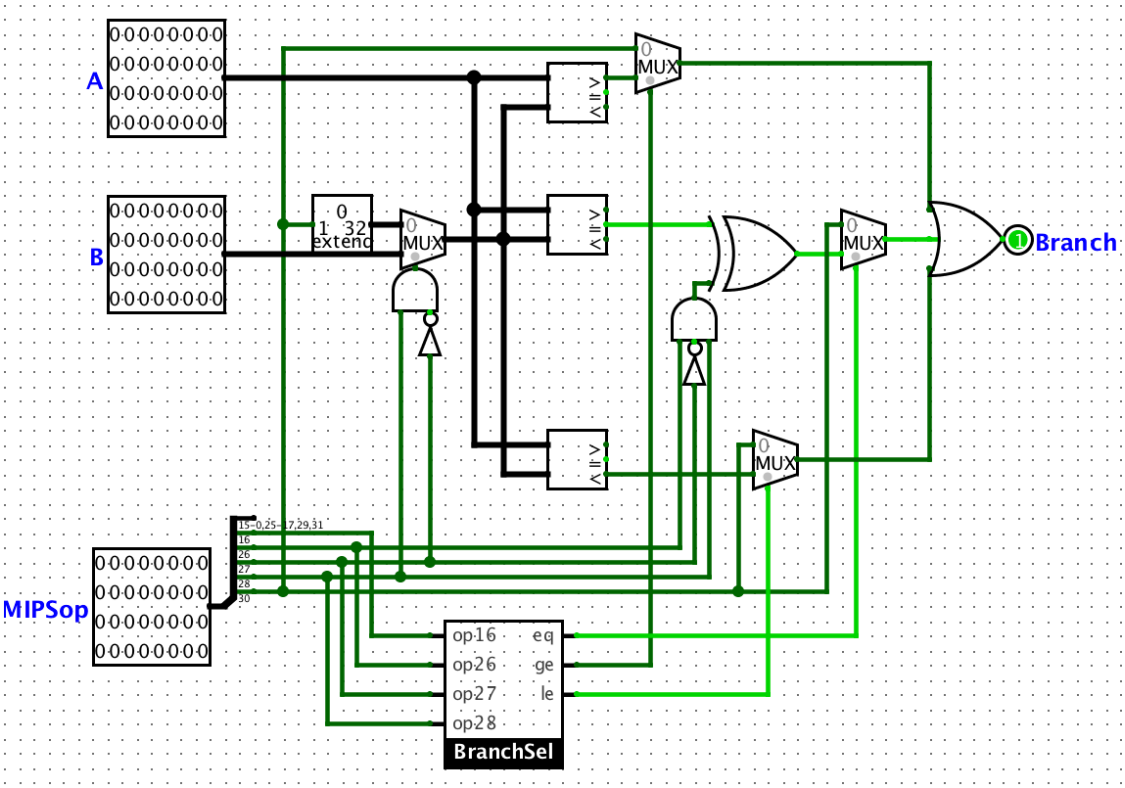


5.1.1 rsel1 and rsel2

rsel1 and rsel2 are Logisim-generated circuits used for selecting the final results from one of A, ALU, signed comparator, or unsigned comparator. Rsel1 is used to select inputs for the R-type instruction while Rsel2 is used for distinguishing I-type instruction.



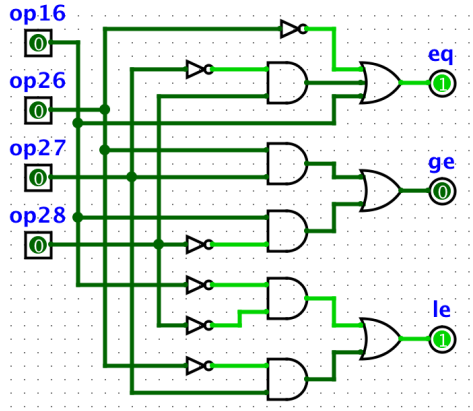
5.2 Branch



The branch circuit outputs whether the branch condition is met. This signal is sent back to JandB circuit to select the next PC. Based on each instruction, the results from >, =,

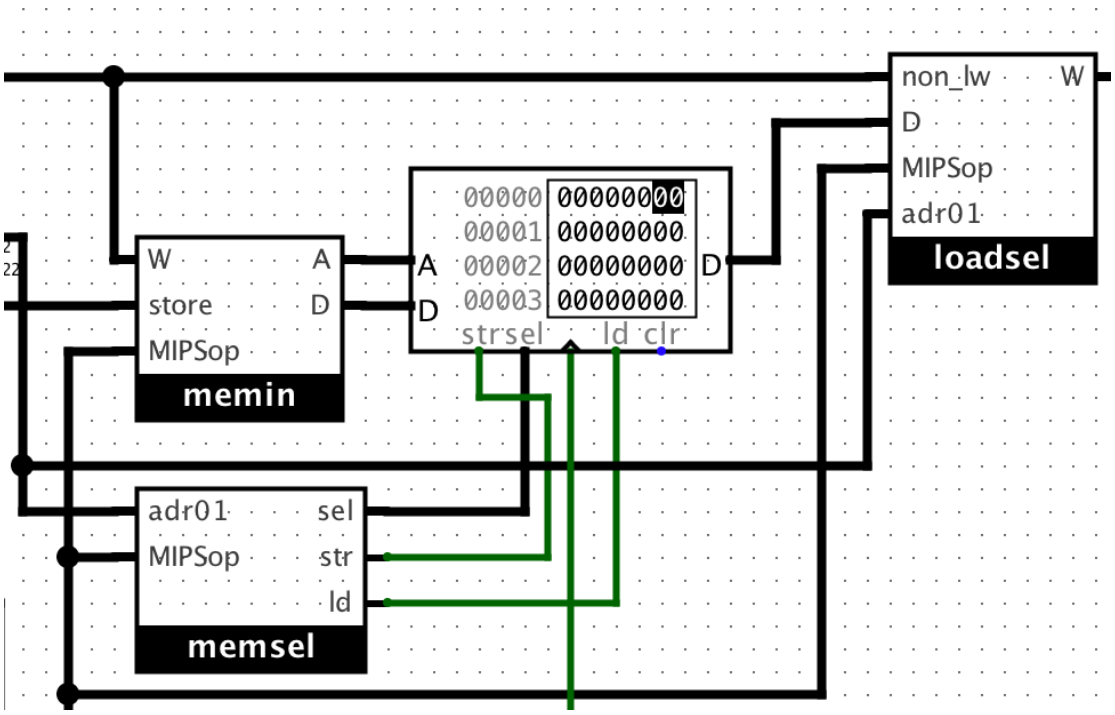
< comparator are selected into the OR gate. For example, is the branch instruction is BLEZ, < and = results are chosen into the OR gate. Which instruction to choose is decided by the subcircuit Branchsel.

5.2.1 Branchsel



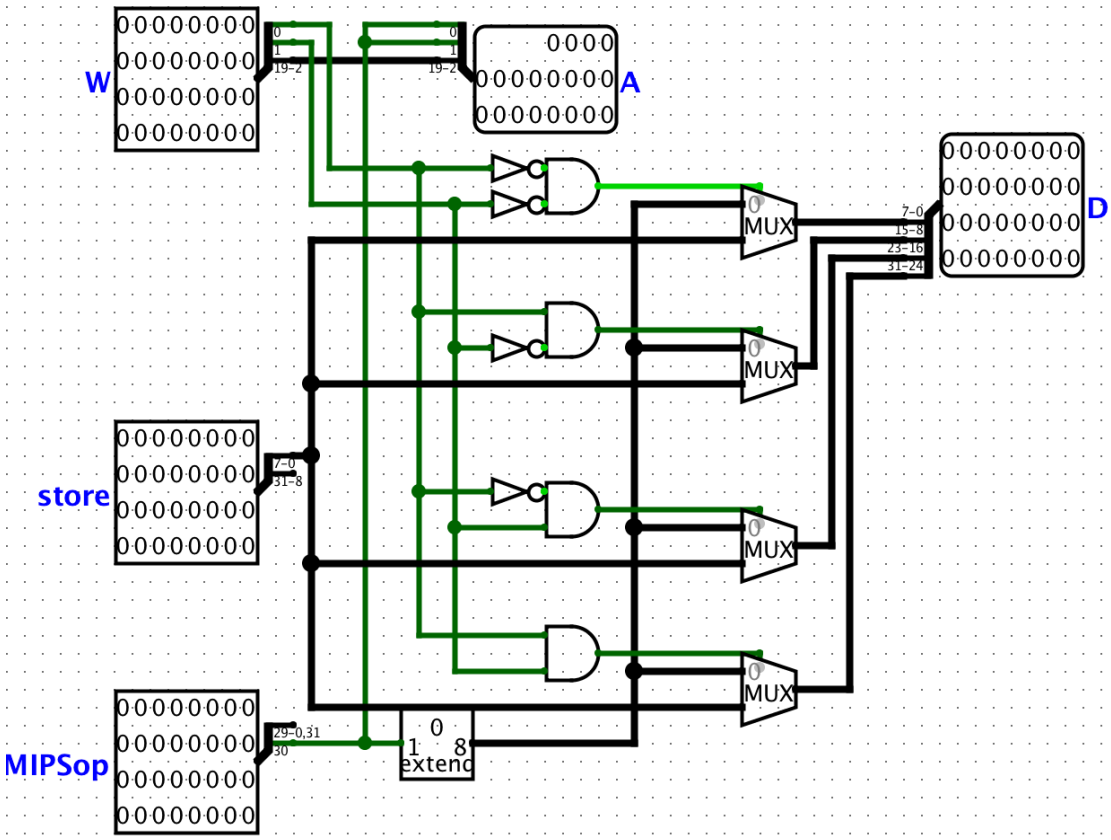
Branchsel outputs signals for >, =, < conditions separately. Depending on which branch instruction it is, eq, ge, and le are either 1 or 0.

6. Stage IV: Memory



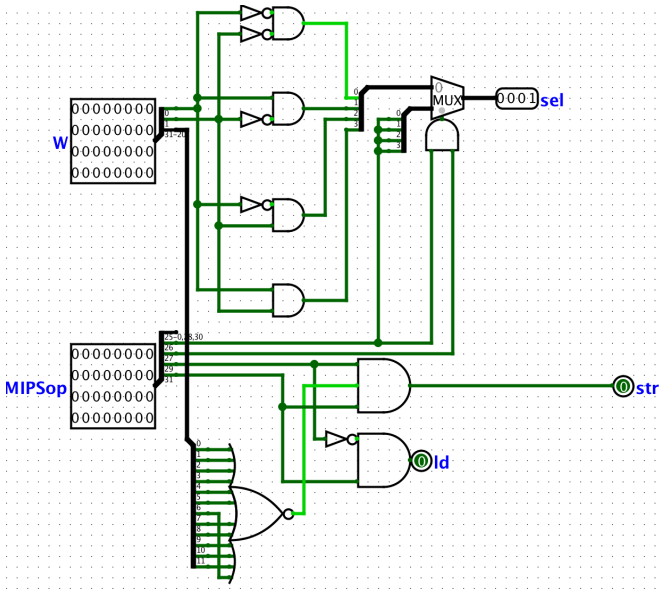
Memory stages have three subcircuits. Memin decides the inputs to RAM, namely the address and load value. Memsel decides the signals for read and write, as well as which byte to read/write if it is a byte instruction. Loadsel selects from the results from ALU and the value loaded from memory.

6.1 memin



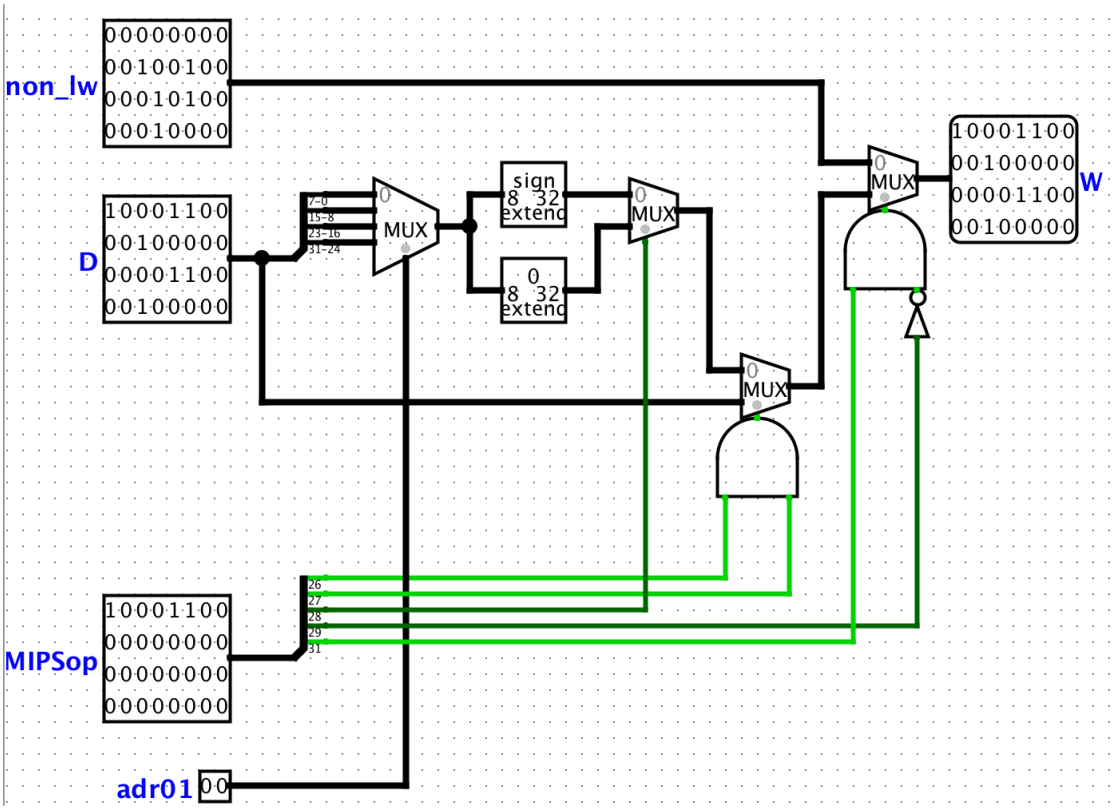
Subcircuit memin select the right inputs to RAM. The address A is the 19-2 bit from the 32-bit address calculated by ALU, with two 0s added to the least two significant bits. The value to be stored, D, is calculated based on the last two bits of the address. If the last two bits are 00, then the 7-0 bit to be stored will be placed at 7-0 of D. If the last two bits are 01, then it will be placed at 15-8, and so on.

6.2 memsel



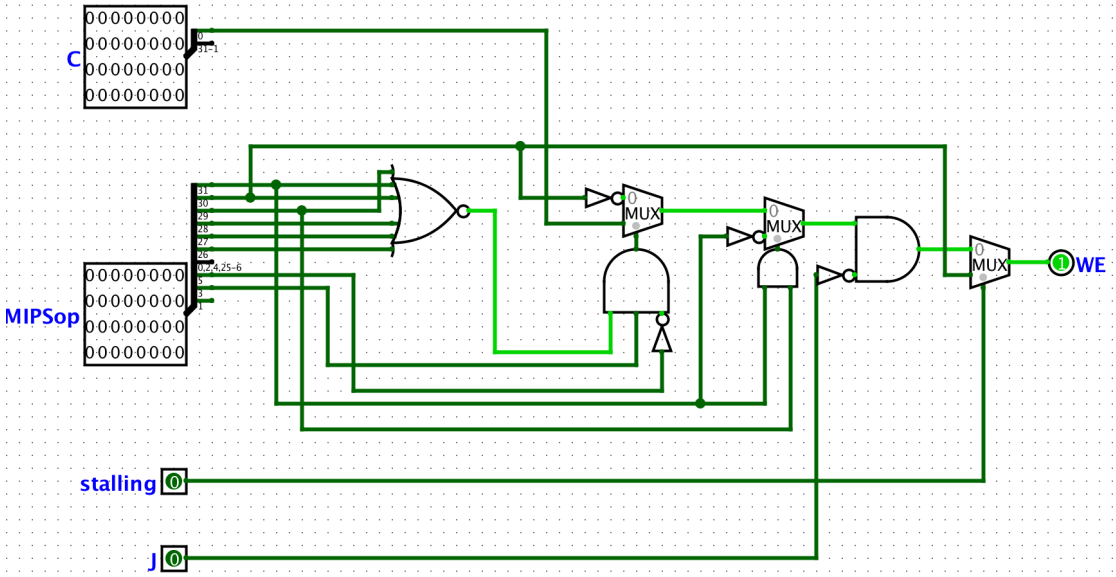
memsel outputs the selecting signal for RAM. If the instruction is load, then str is 0 and ld is1, and vice versa for store. Sel is calculated depending on the last 2 bits of address. 00 corresponds to 0001, 01 to 0010, 10 to 0100, and 11 to 1000.

6.3 loadsel

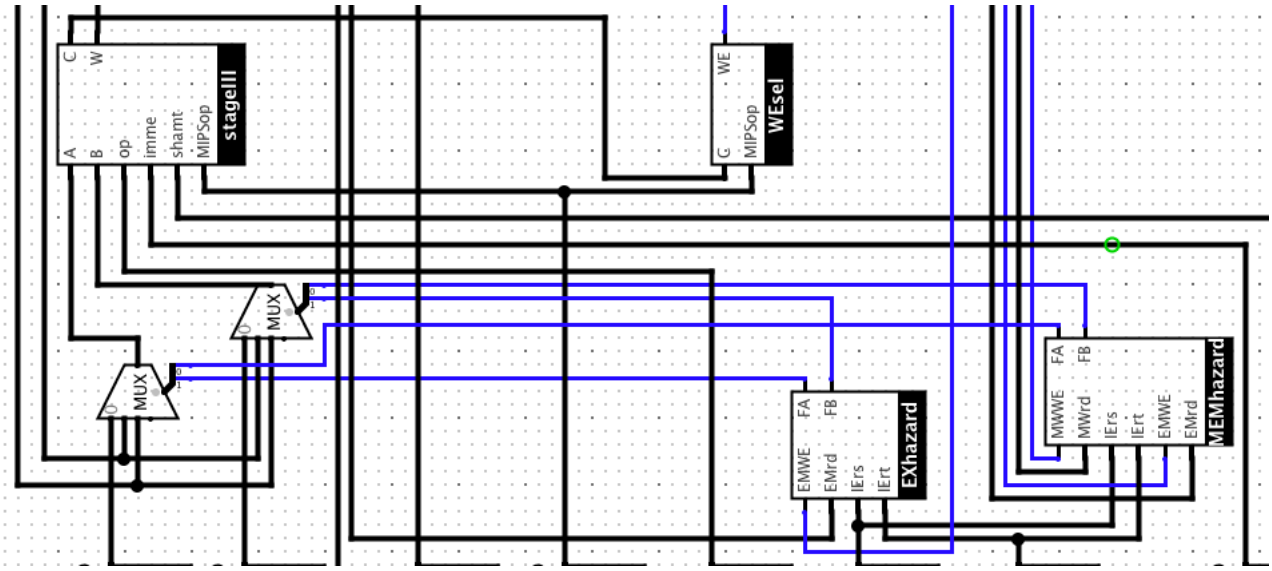


loadsel make sures the right output is written into register. For load instruction, W is from memory. For all other instructions, W is the result directly from ALU. For load instruction, if it is LW, then the entire 32 bits are selected. If it is LB or LBU, then the corrected byte is selected then is either sign-extended or zero-extended.

7. Stage V: write back
   The WEsel subcircuit calculated what the WE signal to the register should be. The logic is that if op[31-29] is 001, or op[31-26] is 000000 and op[5,3] is 0,1 (so JR and JALR cannot change register values), WE is 1. In addition, if the operations are MOVN or MOVZ, WE is the LSB of C. This is implemented through the MUX.



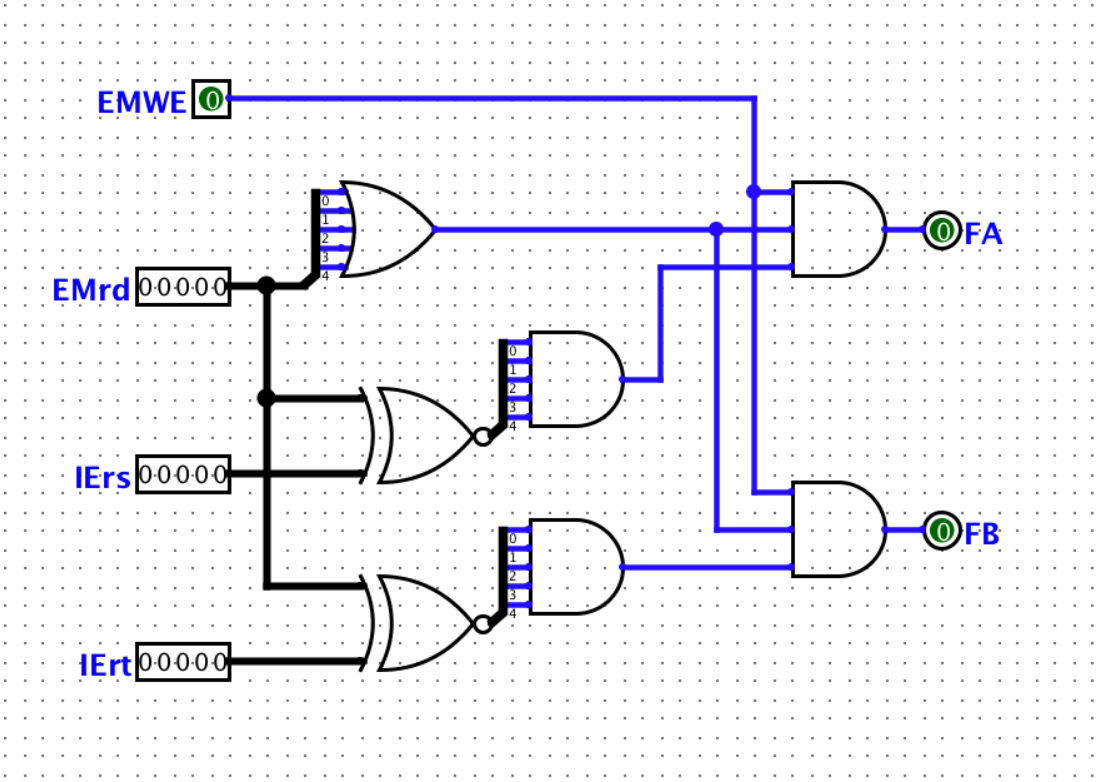
8. Data Hazard



Data hazard will largely affect the correctness of our processor. Luckily we have established mechanism for handling two types of hazards: EX/MEM and MEM/WE hazard. We acquire the desired register values by forwarding/bypassing the correct values before they are written back to the register file.

The two detectors read the register values from both the ID stage and the EX/MEM/WB stage to examine whether values should be forwarded. Their outputs, FA and FB will be used to select the mux values above. The mux selects the correct value to be given to the execution stage.

8.1 EX/MEM hazard

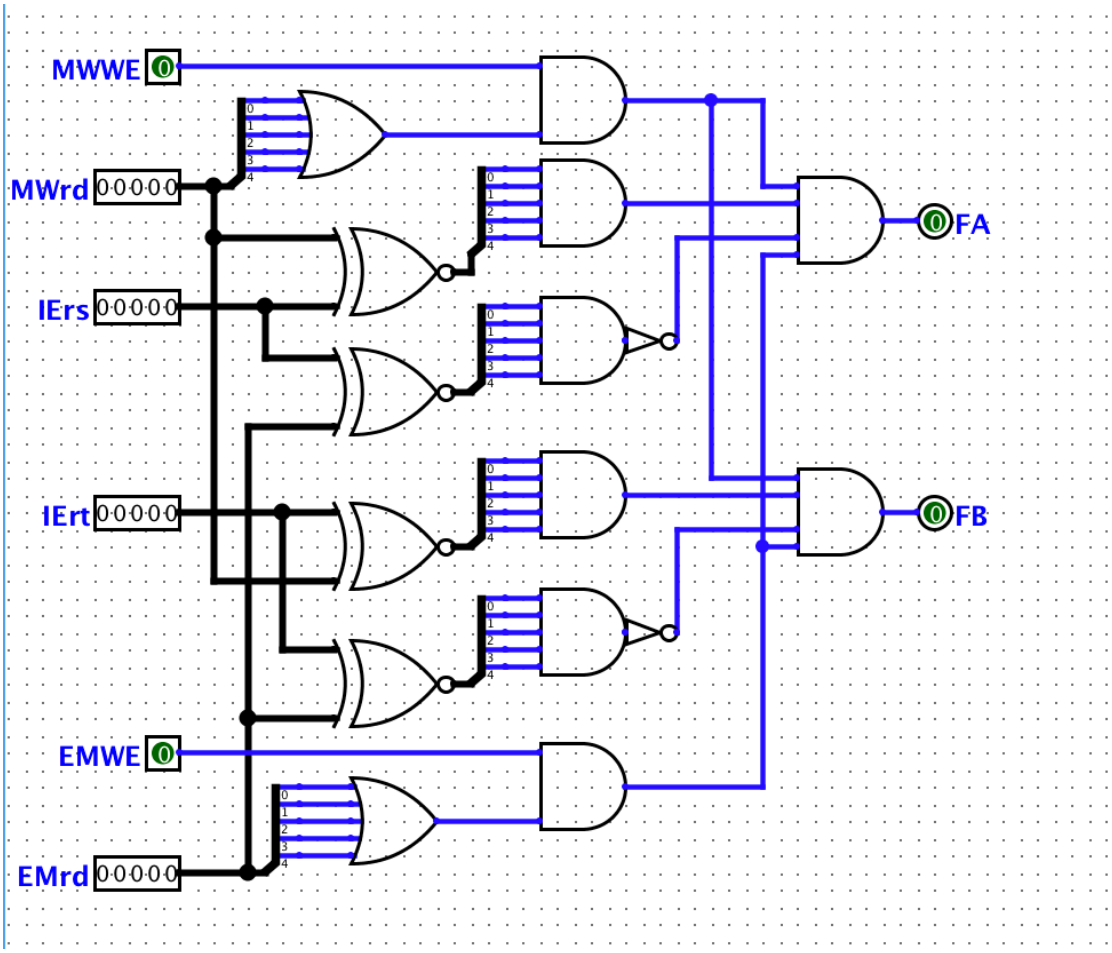


This sub-circuit handles the situation where the register value read from the 32-bit instruction during the decode stage is wrong because a register's value may have been changed but not written to the register file. Thus we have to correctly forward the correct result from the execution stage to decode stage. We closely follow the forwarding logic from the writeup:

```
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
```

and we build this sub-circuit. FA and FB will output one if hazard is detected given the above logic.

8.2 MEM/WB hazard



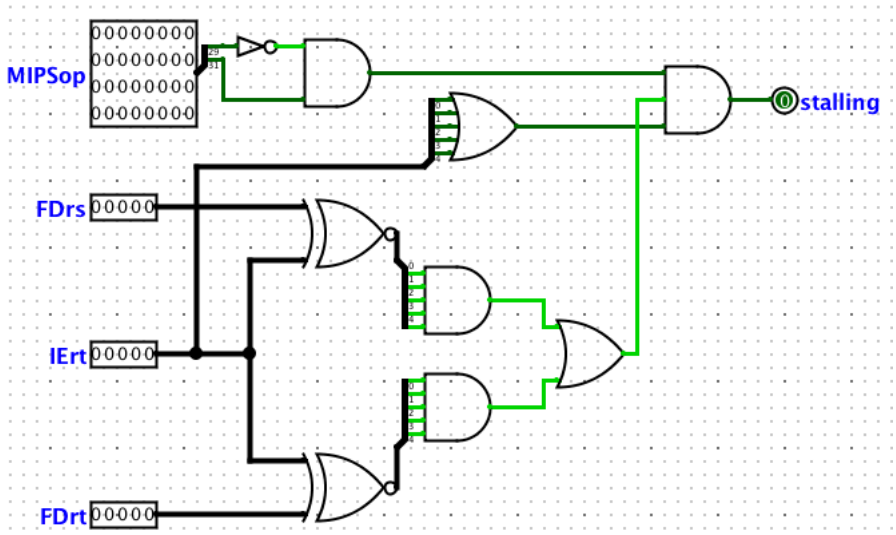
The functionality well resemble the EM/MEM hazard detector, except the forwarding logic is a bit more complicated.

```
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
```

9. Stalling

Stalling signal depends on three separate conditions: the operation is load, the register written to is not $zero, and the register is used as a source for the next instruction. This is three signals going into the AND gate. If the stalling signal is 1, then the PC keeps unchanged, and WE is set to 0.



10. Test

We have written individual tests to test each instructions, and compare their values to the MIPS interpreter: http://dannyqiu.me/mips-interpreter/ as created by Danny Qiu, CS3410 TA. We also mix up the instructions from both table A and table B to make sure that table B instructions will not affect any register value, yet.

We also test the processor using the hard-coded hailstone(5) program. It can be used to test upon data-hazard.

An example list of tests is given:

SRA $2, $3, 8
XORI $22, $28, 90
XORI $18, $9, 38
ADDIU $4, $4, 5
MOVN $21, $22, $24
MOVZ $17, $7, $2
SLTI $2, $4, -327
ADDIU $4, $4, 4
LUI $4, 383
SLTI $7, $9, 5
SLTIU $20, $2, 77

SLTI $25, $5, 4

Hailstone(5):

```
SUBU $8, $8, $8
ADDIU $8, $8, 5
SUBU $9, $9, $9
ADDIU $9, $9, 1
SUBU $7, $7, $7
ADDU $7, $7, $8
ADDU $8, $8, $7
ADDU $8, $8, $7
ADDIU $8, $8, 1
ADDIU $9, $9, 1
SRL $8, $8, 1
ADDIU $9, $9, 1
SRL $8, $8, 1
ADDIU $9, $9, 1
SRL $8, $8, 1
ADDIU $9, $9, 1
SRL $8, $8, 1
SUBU $2, $2, $2
ADDIU $2, $9, 0
```

We test upon each individual instruction from table A and B. We test load-use hazard with stalling combining instructions from both tables. We also include data-hazard and forwarding test. A comprehensive list of test cases can be found in our submission.