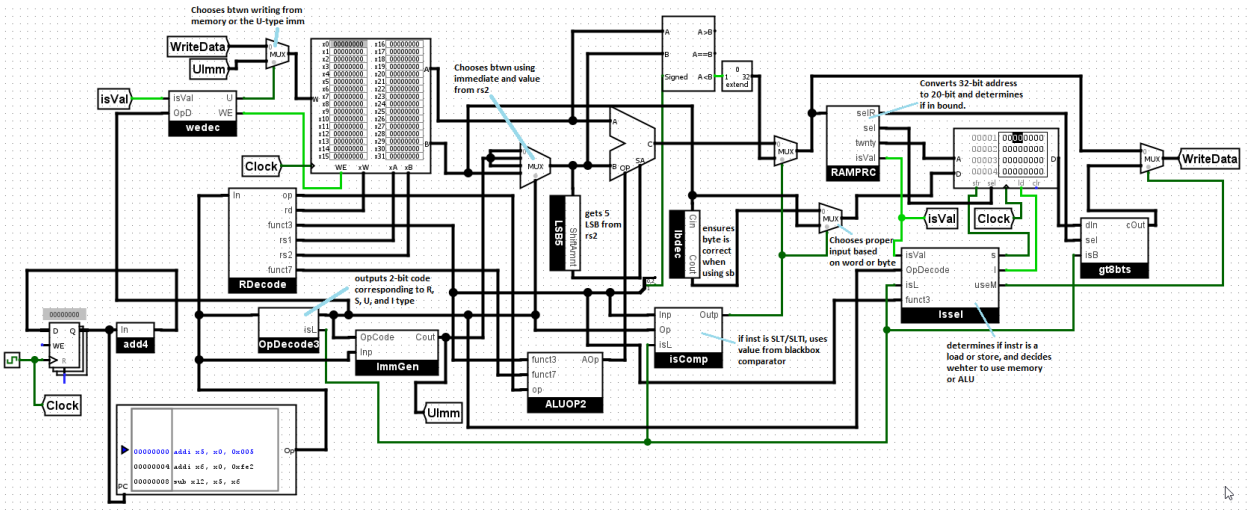
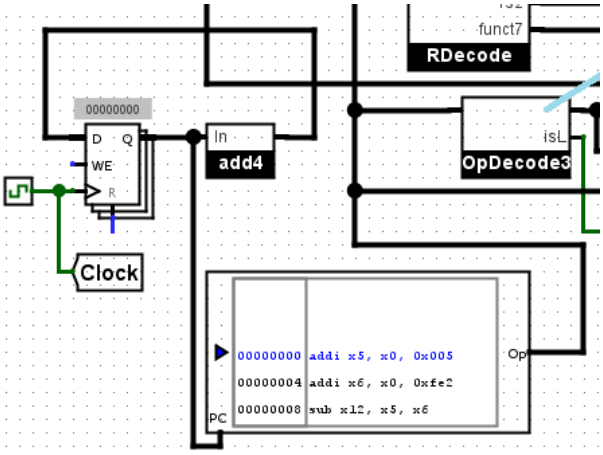


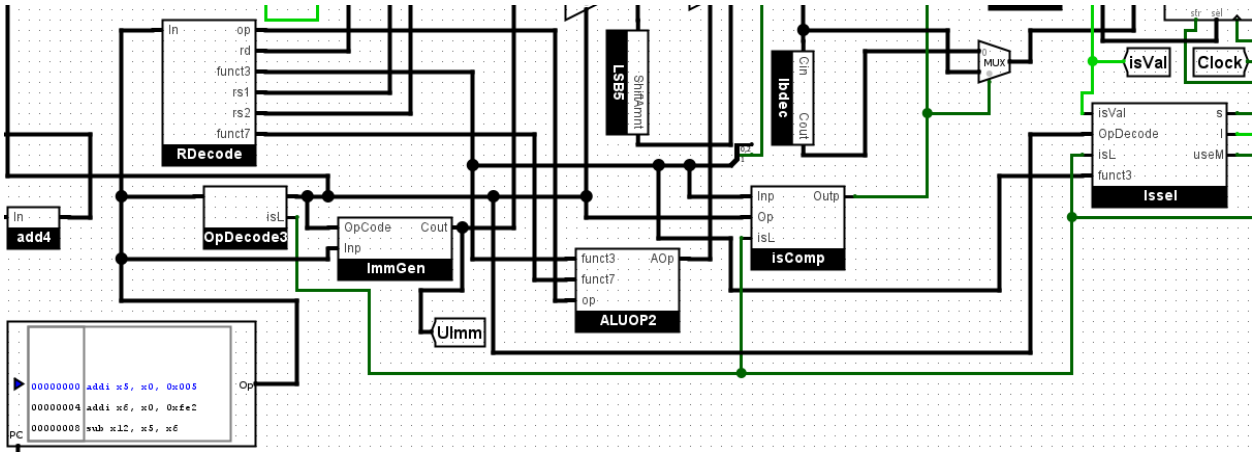
Implementation:

- Fetch:



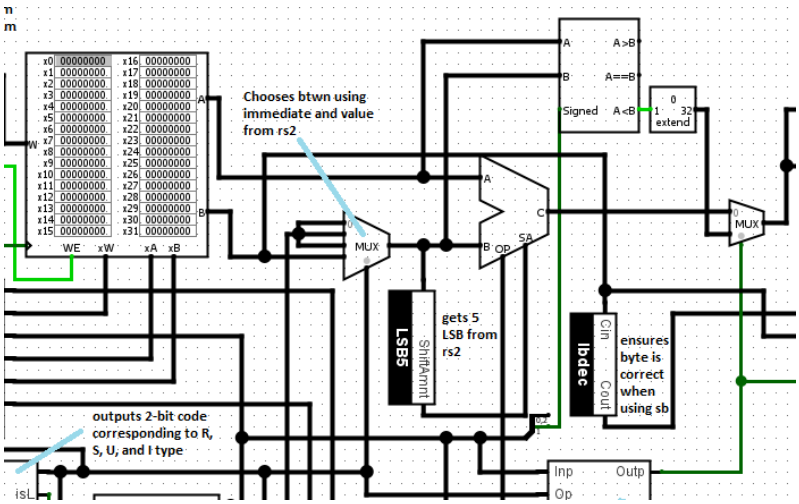
  - Fetch was easy enough to implement. First, we took the PC and connected it to the memory. 4 is then added to the output of the PC, which is wired back into its input. The clock is connected to the PC so that the 4 is added at each cycle.

- Decode:



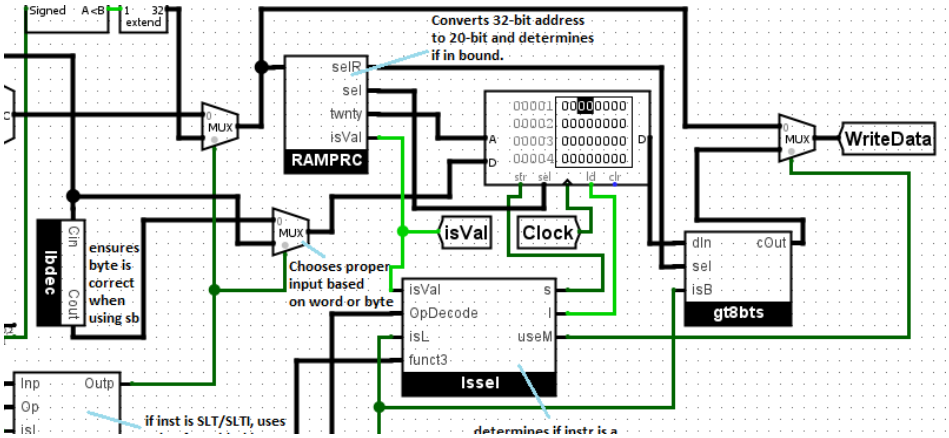
- The decode step was implemented at various stages, and there are a multitude of components to it, depending on what is to be decoded.
- First, the output from memory is taken and passed through OpDecode3, which outputs a 2-bit type code which corresponds to I, S, R, or U type. It also outputs whether the instruction is a load.
- Next, the 32-bit immediate is generated based on the 2-bit type code and the instruction.
- The instruction is decoded into all of the parts used by R-type instructions (and most others) in the RDecode subcircuit.
- Based on funct3, funct7, and the 7-bit opcode, the 4-bit opcode for the ALU is decoded.
- We also decode whether or not the instruction is a Comparator (SLT/SLTI) in isComp.
- in ISsel, we determine whether to enable store, load, and whether to use the output of the memory or ALU, based on the address being valid, the 2-bit type opcode, and funct3.
- Finally, we read from the register using all of the decoded information from RDecode.

Execute:



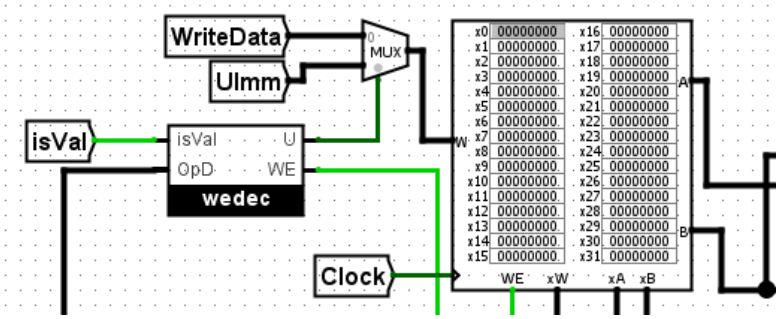
- Next, we wire the outputs of the first register into both the ALU and the blackbox comparator, since both use at least one register.
- We mux between the output of the second register or the immediate based on the 2-bit type opcode, and pass the muxed result into both the ALU and the comparator. (We take only the 5 least significant bits of the result when we input into SA).
- The ALU op-code decoded earlier is passed as OP into the ALU.
- The output of the comparator is 0-extended to be 32 bits instead of 1 bit.
- Now that we have a value from both the comparator and the ALU, we choose which one to keep using with a mux, which decides based on the output of isComp.

Access Memory:



- We pass the output from the mux of the ALU and comparator into a RAM Processor. This subcircuit takes the 32 bit input and generates the select bits for the RAM, the 20 bits that represent the desired memory address, and whether or not the input is a valid memory address. We pass the address and select bits into memory.
- If the memory address is invalid, a nop is created by disabling storing and loading.
- We Pass in isVal, the 2-bit type opcode, whether or not the instruction is a load, and funct3 into LSsel, which determines if store and load should be enabled. These values are then passed into the memory.
- We choose the input D using a Mux based on if the instruction is loading a byte or a word. If a word, it simply takes the value from rs2, but if a byte, we ensure that the memory processes it correctly by converting it in lbdec.
- We connect a clock to the memory so it is synchronized with the rest of the circuit.
- If we are loading a value from memory, we make sure to get the correct value based on if we are loading a word or a byte. We convert the output of D to ensure the correct value is returned. Finally, we mux the value retrieved from memory, and the output of ALU/Comparator based on whether a memory or ALU/Comparator instruction had been executed. The muxed output is then routed to the register file.

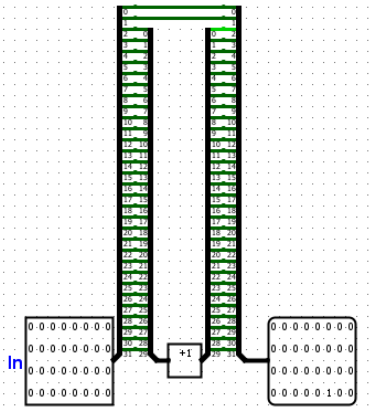
Write Back:



- Using a mux, we decide whether the output from memory/ALU/comparator should be passed back into the register, or the immediate in the case that the instruction is U-Type.
- Based on the 2-bit type opcode and whether the address is valid, we determine wether to enable writing back to the register file. We connect our rd from RDecode to xW to ensure the correct register would be written to.

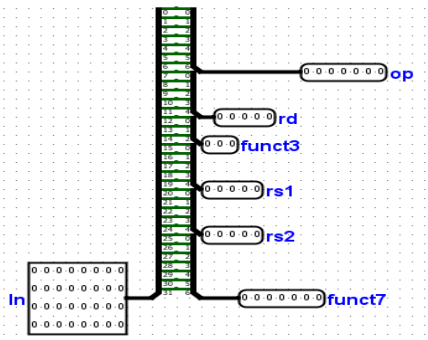
Subcircuits:

Add4:



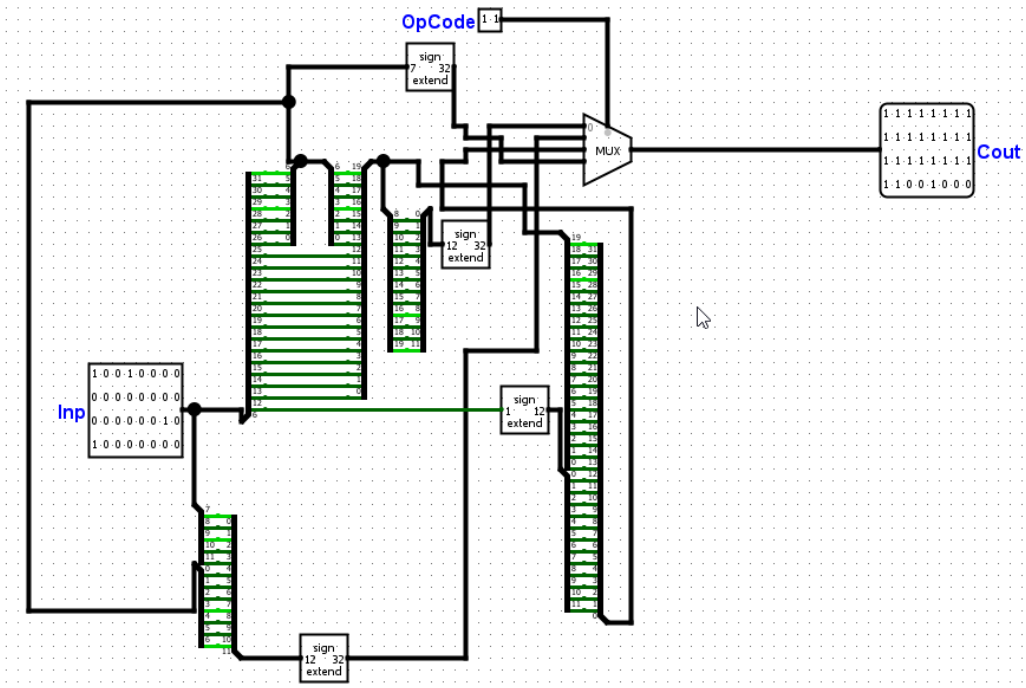
Add4 is used by the PC to add 4 to the 32-bit input. To add 4, we simply split the 32-bits into the first 2 bits and the other 30. We have the first two bits remain the same, but we increment the last 30 bits. By “adding 1” to the 3rd bit, we effectively add 4 in decimal.

RDecode:



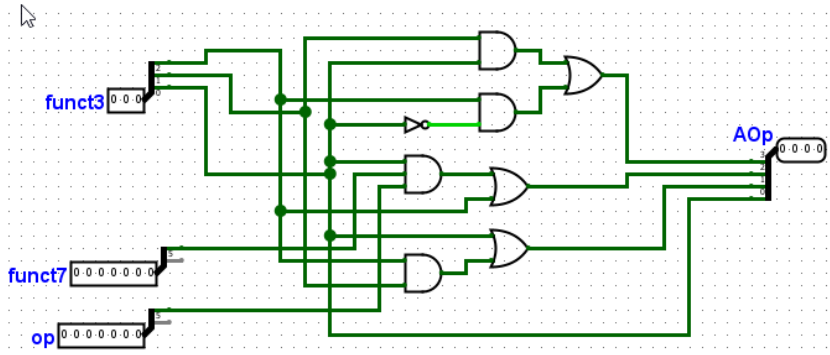
- RDecode takes the 32-bit instruction as an input. It splits the instruction into the various portions of an R-type instruction using multiple splitters, including op, rd, funct3, rs1, rs2, and funct7.

ImmGen:



ImmGen takes the 32-bit instruction input and splits it into individual bits: the first 7 bits R-type, the first 12 for I-type, 7+5 bits for S-type, and the 12 bits for U-type. All the immediates are then sign-extended to 32-bits. Then, based on the 2-bit type opcode (which tells us whether the instruction is I, U, S, or R-type), the correct immediate is chosen and outputted.

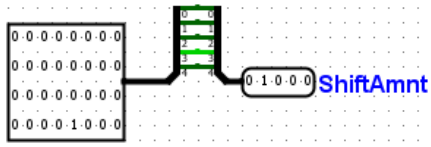
ALUOP:



ALUOP takes funct7, the 7-bit instruction opcode, and funct3, and outputs the correct ALU Op-code based off of that. The only 5 bits actually necessary to fully distinguish between all operations are the 3 bits from funct3, the 2nd bit of funct7, and the 2nd bit of op. This circuit was created using the Circuit Analyzer.

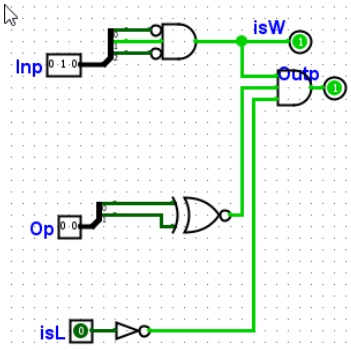
a1	a2	a3	a7	op	b1	b2	b3	b4
0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	0	1	1	0	1	0	x
0	0	1	0	0	0	0	1	1
0	0	1	0	1	0	0	1	1
0	0	1	1	0	0	0	1	1
0	0	1	1	1	0	1	1	1
0	1	0	0	0	0	0	0	0
0	1	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0
0	1	0	1	1	0	0	0	0
0	1	1	0	0	1	0	1	1
0	1	1	0	1	1	0	1	1
0	1	1	1	0	1	0	1	1
0	1	1	1	1	1	1	1	1
1	0	0	0	0	1	1	0	0
1	0	0	0	1	1	1	0	0
1	0	0	1	0	1	1	0	0
1	0	0	1	1	1	1	0	0
1	0	1	0	0	0	1	1	1
1	0	1	0	1	0	1	1	1
1	0	1	1	0	0	1	1	1
1	0	1	1	1	0	1	1	1
1	1	0	0	0	1	1	1	0
1	1	0	0	1	1	1	1	0
1	1	0	1	0	1	1	1	0
1	1	0	1	1	1	1	1	0
1	1	1	0	0	1	1	1	1
1	1	1	0	1	1	1	1	1
1	1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1	1

LSB5:



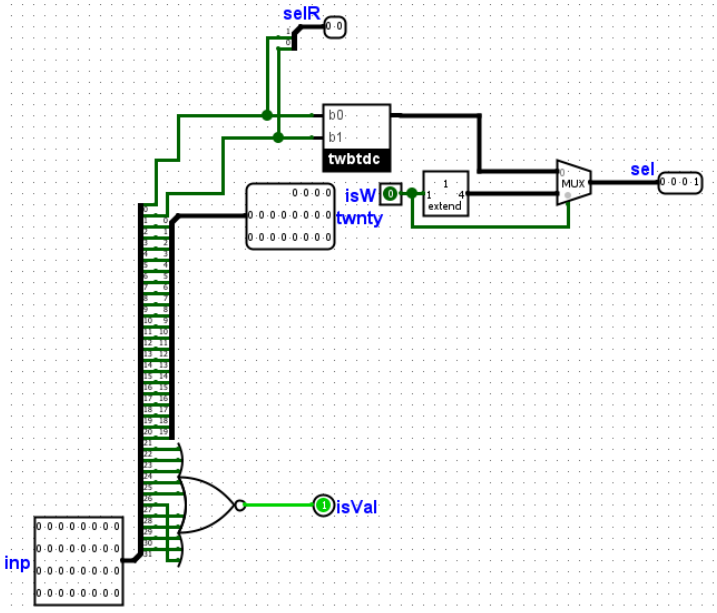
LSB5 takes a 32-bit input and returns the 5 least significant bits using a splitter.

IsComp:



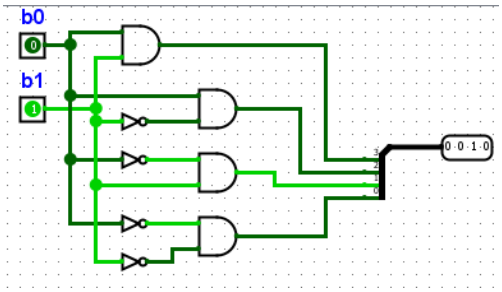
IsComp takes as inputs the 2-bit type opcode, funct3, and an indicator for if the instruction is a load or not. If the op-code is I or R-type, and funct3 is 010, and the instruction is not a load, then the instruction is a comparator, and the output is 1.
We also check the funct3 to determine if the instruction might be a word (010), or a byte (000).

RAMPRC:



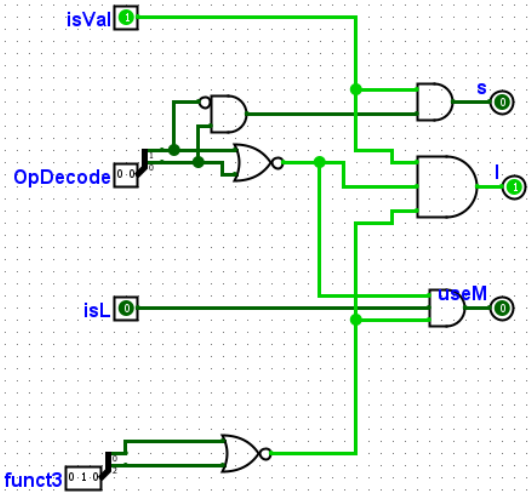
RAMPRC takes a 32-bit address as input. First, we split the address into the first 12 bits, the next 20, and the last 2. First, it checks the first 12 bits of the address to ensure it is in the range of the memory, and therefore a valid address. Next, we output the next 20 bits as the 20-bit address.We pass the final 2 bits into twbtdc (two-bit decoder) for it to generate the proper 4-bit selection code. If the instruction is uses a word, we want to use 1111 as a 4-bit selection code, so we mux the value from the two-bit-decoder with the value 1111 based on if it's a word. We also output those last two bits to use later.

Twbtdc:



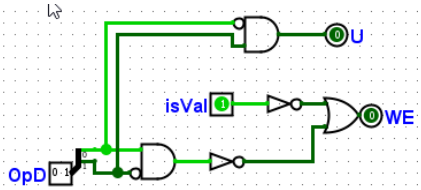
TwBtDc takes two bits as input and outputs 4 bits which correspond to which bytes of the memory should be used. The circuit was made using the circuit analyzer tool using the following conversion:
00 -> 0001
01 -> 0010
10 -> 0100
11 -> 1000

LSsel:



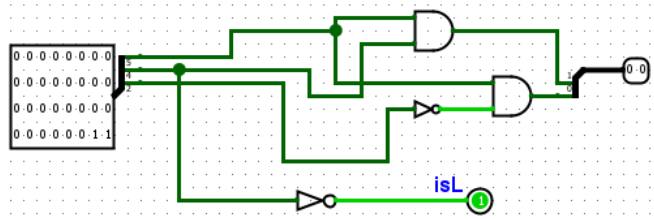
LSsel takes as input funct3, the 2-bit type opcode, an indicator for if the address is valid, and an indicator for if the instruction is a load. If funct3 is 010 or 000, and the OpCode corresponds an I-type (00), and the address is valid, then load is enabled. If the opcode corresponds to an S-type (01), and the address is valid, then store is enabled. Finally, if the instruction is a load, and the opcode corresponds to an I-type, useM becomes 1 (which we will route to a mux to decide wether to use memory or ALU output).

WEdec:



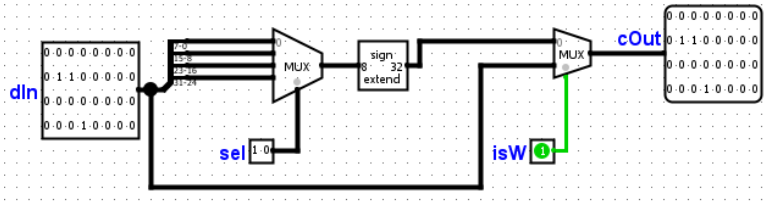
Based on the 2-bit type opcode and wether the address is valid, we decide on if writing is enabled. Writing is enabled if the opcode corresponds to an R-type (11), and the address is valid. We also check if the op-code corresponds to a U-type (10).

OpDecode3



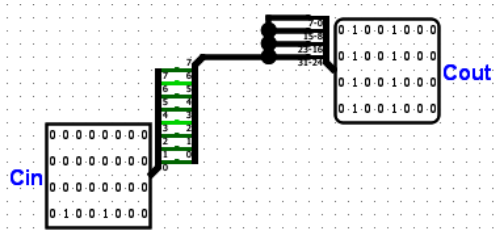
Using a 32-bit input (the instruction) we decode what type of instruction it is. To differentiate instruction types, we must simply know bits 2,4, and 5 of the instruction. We also differentiate based on whether the instruction is a load or not (based on bit 4).

Gt8bts:



Based on the 32-bit input (which is the output from memory), we must process the input so that the correct information is written to the register. We split the 32-bit input into 4 8-bit inputs, then choose which one to process based on the 2 selector bits that we separated earlier from the 20-bit address. This tells us where in the instruction the 8 written bits are. We sign extend the output. Finally, we mux the output with the original 32-bit input, in case we are loading a word (when we always process the entire 32 bits). We send the muxed value as an output.

Lbdec:



Lbdec takes a 32-bit input which consists most importantly of the last 8 bits (where a byte is stored). We split these 8-bits and then copy them into the 3 other 8-bit groups of the 32-bits. This way, we will always store the proper 8-bits into memory.

Testing Strategy

- My strategy when writing test cases was to use the maximum and minimum values of the various components, and to test based on those. For example, the maximum value of a 12-bit immediate is 2047, while the minimum is -2048. Therefore, for any operations that use a 12-bit immediate, I tested with those two values. Alternatively, operations that used two registers would be tested by putting the maximum 32-bit value in a register, and then testing with that maximum or minimum value.