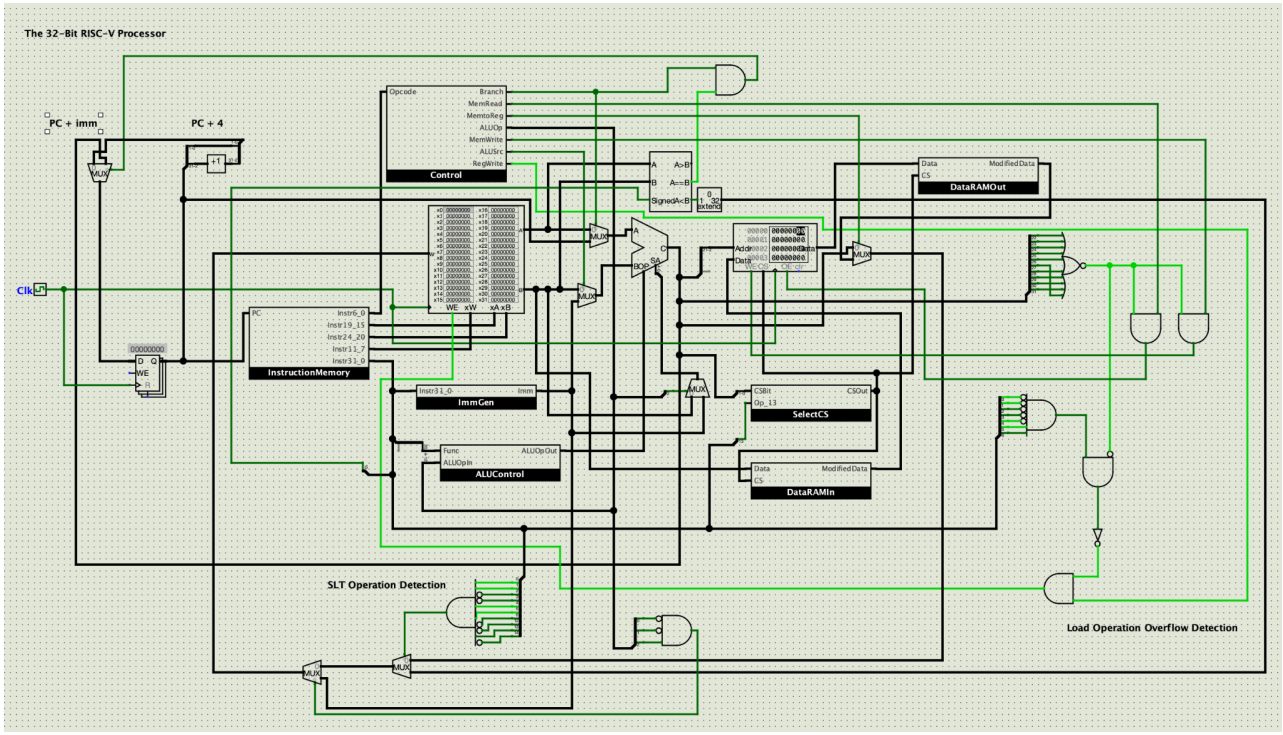
RISC-V Design Document

Overview

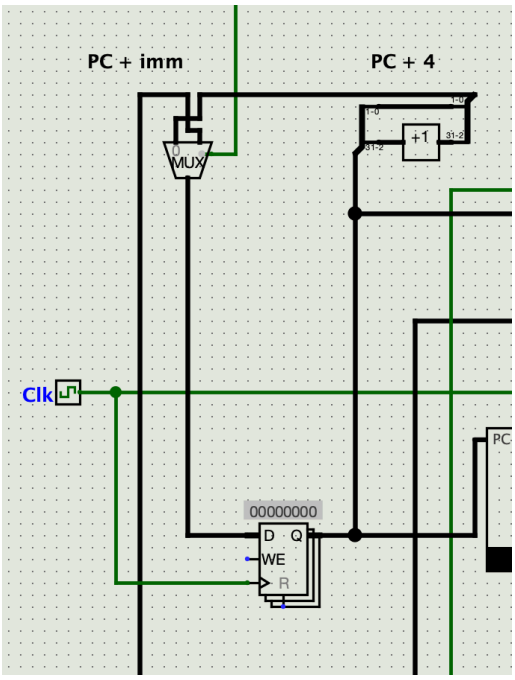
This document details the workings of a 32-bit RISC-V processor, designed in the Logisim software program. It is designed to process select R, I, S, U, and B-type instructions.

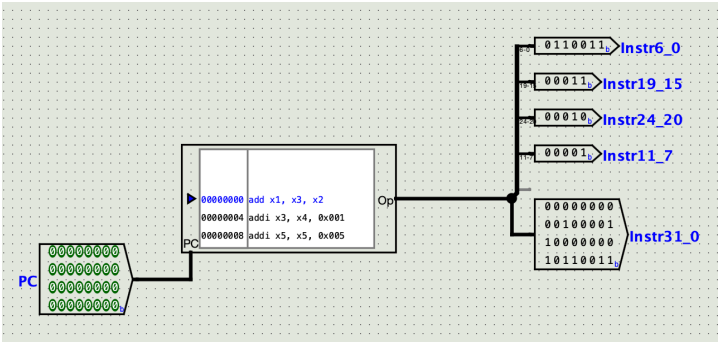
Block Diagram

Below is the block diagram for my processor's main circuit, labeled "RISCV32" in Logisim. It employs 7 subcircuits that perform operations in the different stages of instruction processing. The clock, as depicted below, is connected to the program register, the program ROM, and the RAM component so that all three of these components are always at the same clock tick value.



Implementing the Fetch Stage





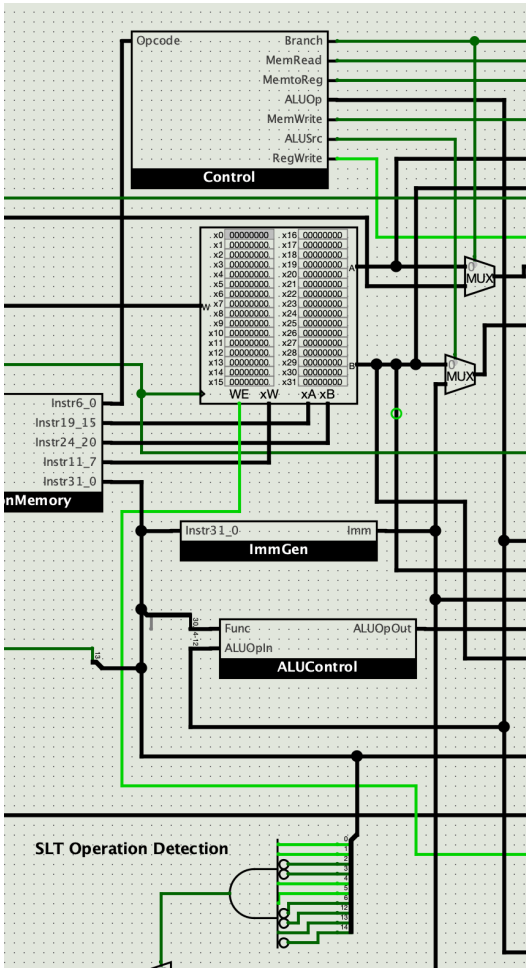
The first stage of the processor's instruction evaluation is the fetch stage. In my circuit, I fetch an instruction back from the main memory (ROM) component based on the address specified by the Program Counter (PC) at a given time. Shown above is a closer analysis of this stage.

The figure on the left here calculates the increment value for the next instruction, which can either be PC + 1 (a 4-byte increment since the input address must be a multiple of 4) or PC + imm (incrementing by a certain 12-bit value if the previous instruction was a Branch Equality instruction). Based on the signal received by the BEQ evaluation logic, which I pass into a multiplexor, if it is 0 I simply perform a normal PC increment; if it is 1, then I perform a PC custom increment based on what was specified in the branch instruction immediately prior. The normal 4-byte PC increment is performed by using a single incrementor to increment the third least-significant bit of the 32-bit PC address (bit 2). By incrementing this bit and leaving bits 0 and 1 the same, I can effectively increment the PC by 4 bytes using efficient logic.

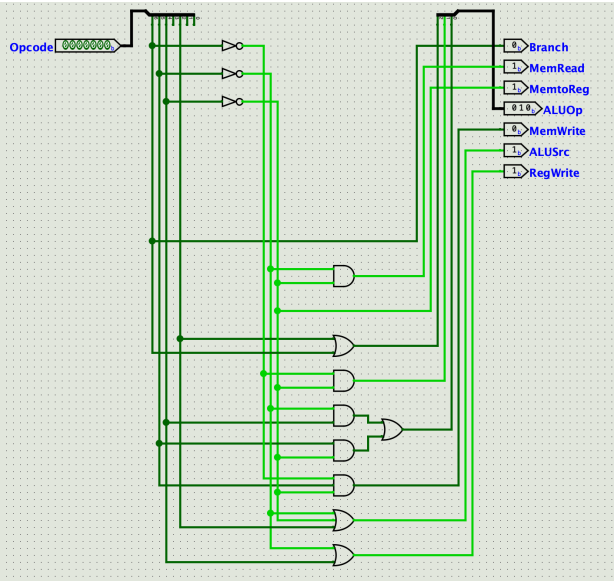
The figure on the right here enters this PC value into the program ROM, where it reads the address specified by the PC and outputs a 32-bit instruction corresponding to the value at the address.

Implementing the Decode Stage

This stage involves decoding the 32-bit instruction read from the program ROM in the fetch stage. To do so, I took the 7 least significant bits from the instruction, and based on the type of operation being performed, I added control signals to access different parts of the processor. Here is the circuit logic for the decode stage:



As shown here, the Control subcircuit outputs 6 different control signals: Branch, MemRead, MemtoReg, MemWrite, ALUSrc, and RegWrite. These signals enable or disable different parts of memory and register reading/writing, as well as passing in an immediate or a full register value. ALUOp is used to pass into ALUControl so that the ALU can process operations that need it.



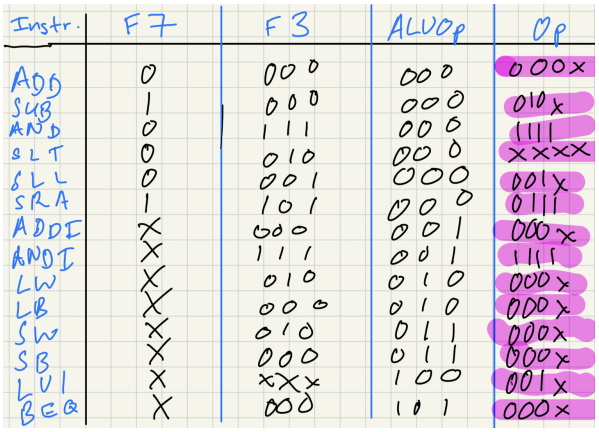
I created a truth table and used the Logisim "Analyze Circuit" tool to input my table and generate the Control subcircuit shown above.

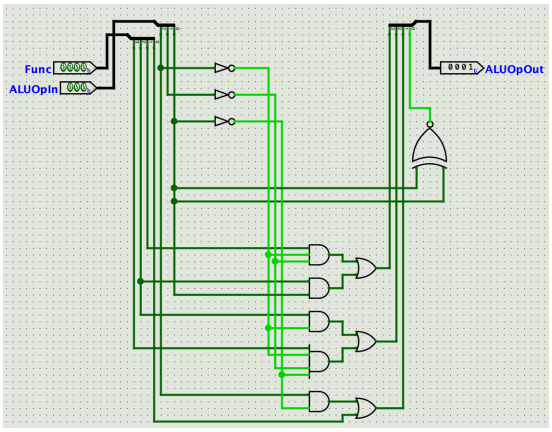
Op	B	MR	MtoR	ALUOp	MW	ALUSrc	RW
0110011 R	0	0	0	000	0	0	1
0 0100 11 I	0	0	0	001	0	1	1
0 0000 11 LW/LB	0	1	1	010	0	1	1
0 10 00 11 SW/SB	0	0	X	011	1	1	0
0110111 LUI	0	0	0	100	0	1	1
1 1 0 00 11 BEQ	1	0	X	101	0	0	0

I separated my values based on the opcode, and there are 6 unique opcodes to consider for this processor. After this, I also included an "ALUOp" 3-bit column here to pass in each unique opcode into the ALUOp control signal to convert the opcode values into 4-bit values that the Logisim ALU component can process. I used the truth table below to create this subcircuit:

R-TYPE	FUNC7	RS2	RS1	FUNC3	RD	OPCODE	ALU	
ADD	0 0 0 0 0 0 0	RS2	RS1	0 0 0	RD	0 1 1 0 0 1 1	0 0 0 X	add
SUB	0 1 0 0 0 0 0	RS2	RS1	0 0 0	RD	0 1 1 0 0 1 1	0 1 0 X	sub
AND	0 0 0 0 0 0 0	RS2	RS1	1 1 1	RD	0 1 1 0 0 1 1	1 1 1 1	and
SLT	0 0 0 0 0 0 0	RS2	RS1	0 1 0	RD	0 1 1 0 0 1 1		
SLL	0 0 0 0 0 0 0	RS2	RS1	0 0 1	RD	0 1 1 0 0 1 1	0 0 1 X	left logical shfit
SRA	0 1 0 0 0 0 0	RS2	RS1	1 0 1	RD	0 1 1 0 0 1 1	0 1 1 1	right arith. shift

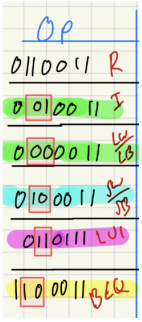
I-TYPE	IMM12	RS1	FUNC3	RD	OPCODE	ALU	
ADDI	IMM12	RS1	0 0 0	RD	0 0 1 0 0 1 1	0 0 0 X	add
ANDI	IMM12	RS1	1 1 1	RD	0 0 1 0 0 1 1	1 1 1 1	and
LW	IMM12	RS1	0 1 0	RD	0 0 0 0 0 1 1		
LB	IMM12	RS1	0 0 0	RD	0 0 0 0 0 1 1		

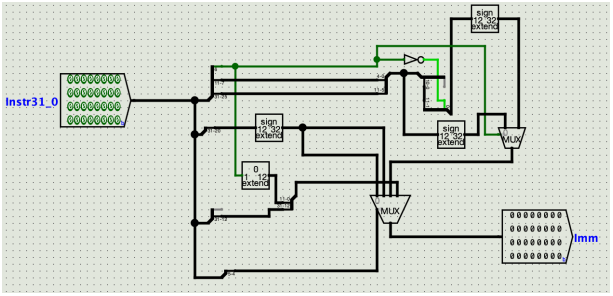




The truth table I imported into Logisim here uses bit 30 from the func7 instruction segment and the 3 bits of the func3 segment to distinctly identify each type of operation. Then, based on what kind of ALU arithmetic I would need, such as addition for the load and store instructions that add values from registers, I outputted a 4-bit opcode from it, and the subcircuit for ALUControl is shown above to the right.

I also needed to distinctly identify when I needed to use an immediate instead of a second register when decoding an instruction, since some instructions like ADDI require an immediate to be passed into the ALU instead. The following ImmGen subcircuit achieves this:

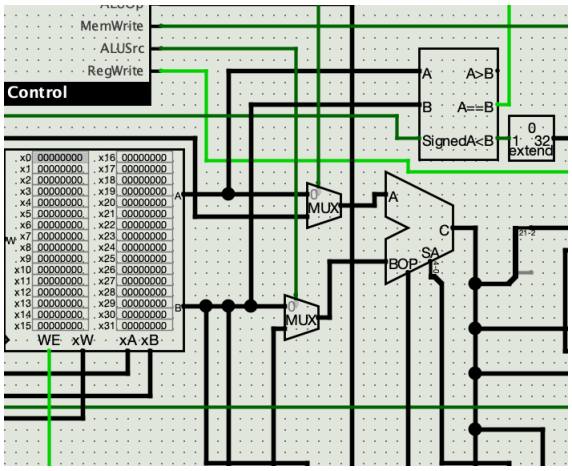




There are 5 different types of processing for opcodes with immediates (R-types don't use immediate values), and as such I created the circuitry and used a 4-input mux to select between

the 5 types. Circled in orange are the two bits (bits 4-5) of the opcode I used to differentiate the types. BEQ and SW/SB instructions had a bit value of 10, so I used another mux and chose bit 6 to separate these two since bit 6 is 1 for BEQ and 0 for SW/SB operations.

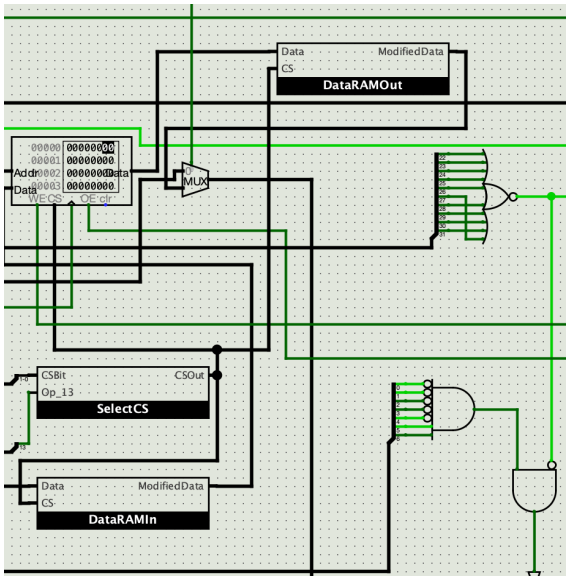
Implementing the Execute Stage



The image to the right here is my execution stage. I pass in A (rs1) and B (rs2) into two muxes here, which select from either an immediate value from ImmGen in place of B (using the ALUSrc control signal; 1 if an immediate, and 0 if not an immediate), or the B value from rs2. R-type operations have ALUSrc = 0, so I pass in both rs1 and rs2 for these operations. For Branch, I take the PC value instead of rs1 and use that to add an immediate from ImmGen to increment the PC by the immediate. This is dependent on the comparator component I used, which when the boolean value from rs1 == rs2 and the Branch control signal is enabled I perform the BEQ operation. SLT also uses the comparator here and performs an rs1 < rs2 boolean comparison. I sign-extend this resultant value to 32 bits and write that value back directly into the register (explained further in the writeback stage below).

Implementing the Memory Stage

This stage involves using a SelectCS subcircuit to map the two CS bits (which are the two least significant bits from the ALU's result, and mapping these values to 4-bit CS values that the RAM component can process. Shown below is the stage's circuit logic and the truth table for SelectCS:



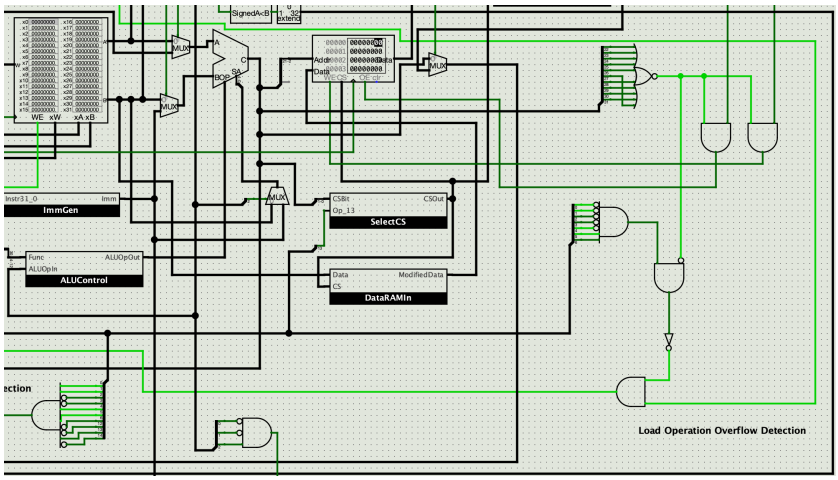
CSBit[1..0]	Op_13	CSOut[3..0]
00	0	0001
00	1	1111
01	0	0010
01	1	1111
10	0	0100
10	1	1111
11	0	1000
11	1	1111

For the LW and SW instruction, we need to load the entire length of bits, so I set CSOut to 1111 for those cases since all 4 bytes in memory need to be stored to or loaded from. For LB and SB, on the other hand, I use the middle bit from the func3 segment of the instruction (bit 13), since it is 1 when the instruction is loading or storing an entire word, and 0 otherwise. This, in conjunction with CSBit input, let me output a 4-bit CSOut value that I inputted into the RAM component.

DataRAMIn and DataRAMOut deal with modifying memory. Depending on how many CS bits of the 4 bits are a value of 1, I take the least significant 8 bits from the value in B (rs2), since this is where the final data is, and copy them to the 4 byte sections of a 32-bit data value. This, in turn, then uses DataRAMOut to shift down any bytes to their least significant positions before writing back to the register (if necessary, depending on if control signal RegWrite is enabled).

To account for the upper bound of memory, I evaluate the 10 most significant bits of the ALU result. If any of them equal a 1 (in the 10-input NOR gate), this signifies a 32-bit value greater than the maximum value this Logisim RAM component can hold, so I immediately disable writeback to the register, turn off reading from an invalid memory location (OE disabled), and disable WE to memory. These are the AND gates shown above. If it is a valid address, however, I allow the ALU result to be split into bits 2-21 for the address.

Implementing the Writeback Stage



The AND gates here all represent the processing of writeback. Writeback is disabled when the instruction does not require updating the register file, such as branch or store instructions that do not need to modify any registers. It is also disabled for invalid addresses for LB and LW, which are detected through an AND gate that detects a load-type opcode.

Testing Strategy

To test my processor's circuit, I conducted rigorous edge cases and general random cases for each of the operations I implemented here. Depending on the type of instruction, such as R-types, I would also use the maximum and minimum bit values possible to check for corner cases where applicable.