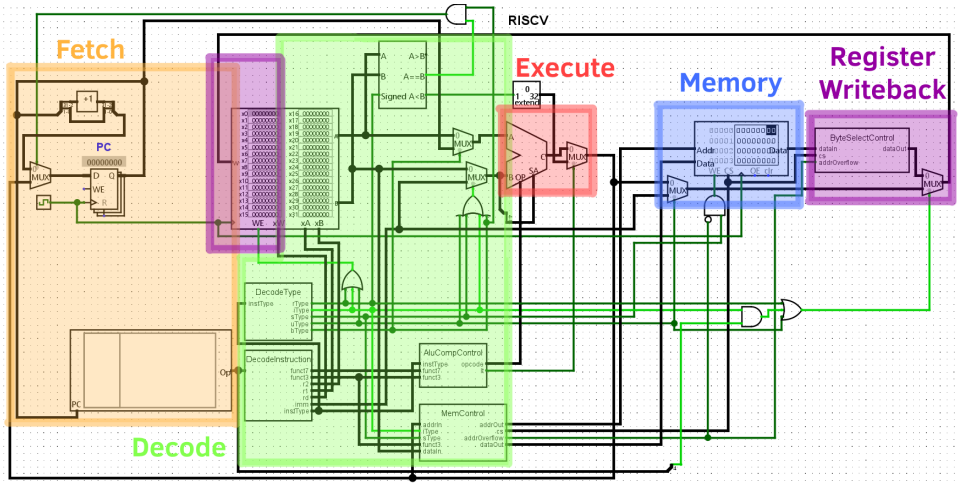
P1: RISC-V

Jack Ryan (jpr254)

March 6, 2024

1 Overview

The circuit shown in the block diagram below is a single-cycle RISC-V processor. This processor has been designed to accept instructions from the following table and execute them according to the RISC-V instruction set architecture. The block diagram breaks the main circuit into the five stages of the RISC pipeline, each of which will be explained in further detail.



2 Fetch

The RISC-V pipeline starts with fetch. This is the step where the next 32-bit instruction is grabbed and decoded. Fetch uses a one-bit incremented, a register, and the program ROM. Since all instructions are 32-bit (four

Format	Instructions
R-type	ADD, SUB, AND, SLT, SLL, SRA
I-type	ADDI, ANDI, LW, LB
S-type	SW, SB
U-type	LUI
B-type	BEQ

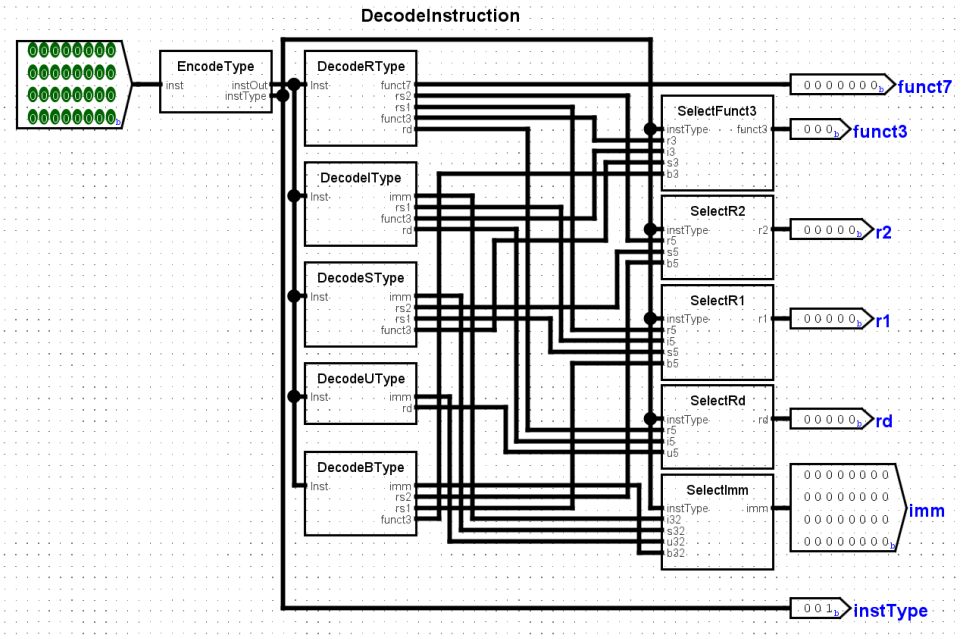
bytes), the program counter must always be a multiple of four bytes. To increment by multiples of four a splitter is used to break off the two least significant bits. The remaining 30 bits are incremented by one and then another splitter joins the bits back together. Due to how binary numbers work, this has the effect of incrementing by four instead of one. Since our processor handles a branch instruction, it is also possible that we want to jump forward or backward more than four bits instead of continuing linearly through the instructions in the program ROM. When a B-type instruction is decoded (discussed more in Section 2) and a jump is required, a control signal is sent a mux. This mux outputs PC + 4 when the signal is 0, and PC + imm when the signal is 1. The current program counter is sent to the program ROM, which outputs an instruction. There are no added subcircuits for this step, all discussed functionality can be observed in the block diagram.

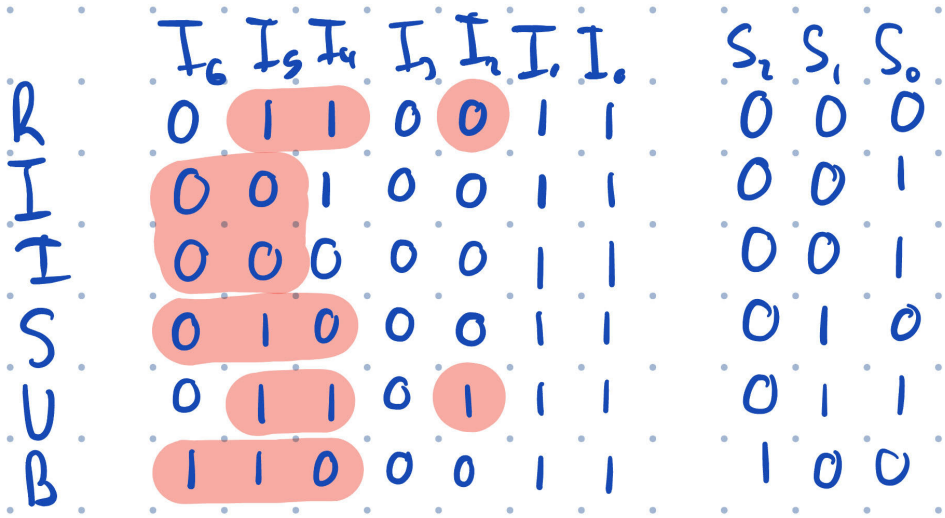
3 Decode

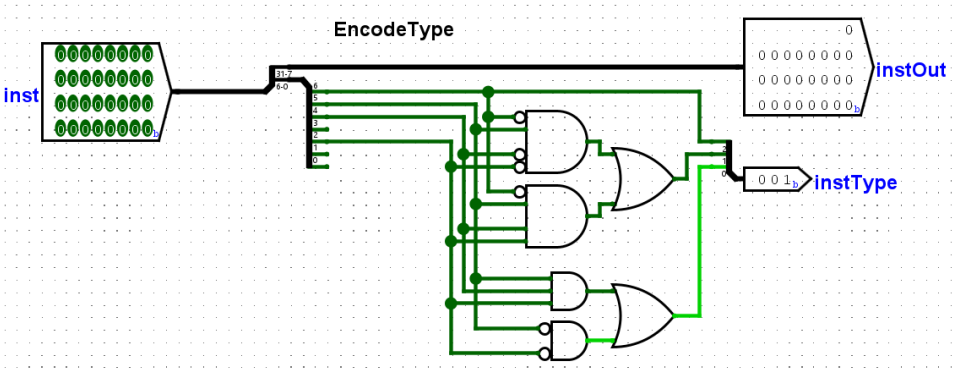
The second step in the pipeline, decode, is also the most complex. My implementation includes a 32-bit wide by 32-registers deep register file, along with a multitude of subcircuits to decode the instruction given by program ROM in fetch. The instruction is first passed into DecodeInstruction.

DecodeInstruction takes in a 32-bit instruction and splits it into fields (funct7, funct3, r2, r1, rd, imm) according to the type and specific function in the case of I-type. For any given type, not all of these fields will be used. In these cases, the unused fields are zeroed out so as not to access any unexpected registers and/or perform computations on unexpected values. The first step in decoding the instruction is to figure out what type of instruction we are dealing with. To avoid having subcircuits with an individual input for each type I chose to encode the type into a single 3-bit input.

The below table describes the encoding done by EncodeType. Each type of instruction has a unique 7-bit opcode (the seven least significant bits). By identifying unique patterns in each opcode (highlighted in red), these are


DecodeInstruction
EncodeType
inst
instOut
instType
DecodeRType
DecodeIType
DecodeSType
DecodeUType
DecodeBType
SelectFunct3
SelectR2
SelectR1
SelectRd
SelectImm
funct7
funct3
r2
r1
rd
imm
instType


I6 I5 I4 I3 I2 I1 I0 S2 S1 S0
R 0 1 1 0 0 1 1 0 0 0
I 0 0 1 0 0 1 1 0 0 1
I 0 0 0 0 0 1 1 0 0 1
S 0 1 0 0 0 1 1 0 1 0
U 0 1 1 0 1 1 1 0 1 1
B 1 1 0 0 0 1 1 1 0 0

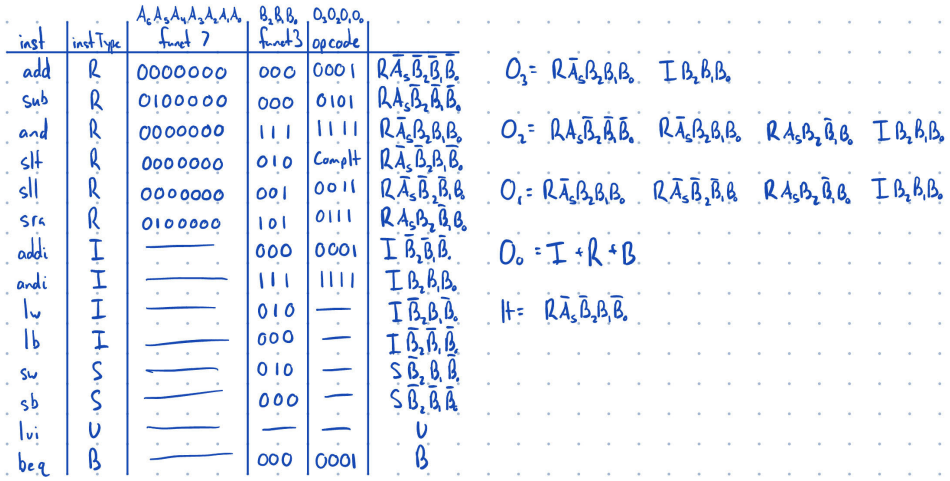


used to figure out what instruction the circuit is dealing with and encode it accordingly. EncodeType takes in a 32-bit instruction and outputs a 3-bit representation of the type, as well as the original instruction minus the seven least significant bits for the type opcode.

Once the type is known, the circuit must split up the instruction according to the RISC-V ISA. DecodeInstruction sends the now 25-bit instruction into a decoding subcircuit for each of the five types. These decoders will split, join, and extend the given instruction and output values according to specifications. The next step is to select the correct values for each of the possible fields (funct7, funct3, r2, r1, rd, imm), dependent on the type. Other than function7, which exists in every type this processor deals with, DecodeInstruction needs to select the correct fields according to their type. If a given type uses a field, say r2 for R-type, it will be outputted by SelectR2 correctly. If a field is not used it is zeroed out, as discussed earlier. At this point, the instruction is separated into the correct fields, which are outputted along with the instruction type for ease of use. (To limit total page size DecodeXType and SelectX are not pictured, know that they are relatively simple and can be understood based on the RISC-V ISA)

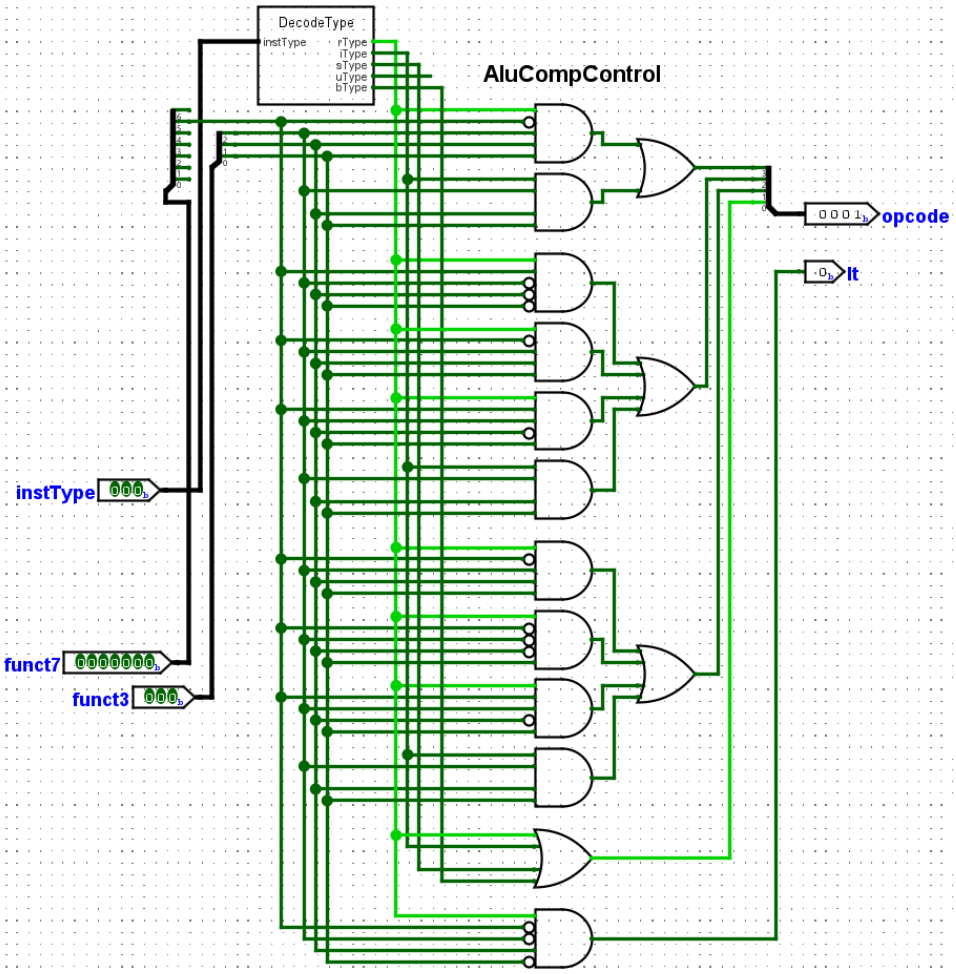
At this point rd, r2, and r1 are passed directly into the register. A subcircuit, DecodeType, which does the opposite of EncodeType based on the same table, is used to easily decode the type when needed. Per specification, writing is enabled for types R, I, and U. Branching off DecodeType is an or gate that acts as a control signal (RegWrite) to decide when to write data to the register. Function7, fucntion3, and the type are passed into AluCompControl with acts off the following table to generate an opcode to be used by the ALU and a control signal to designate when the output of the A $<$ B from the black-box comparator is used (needed because ALU does not

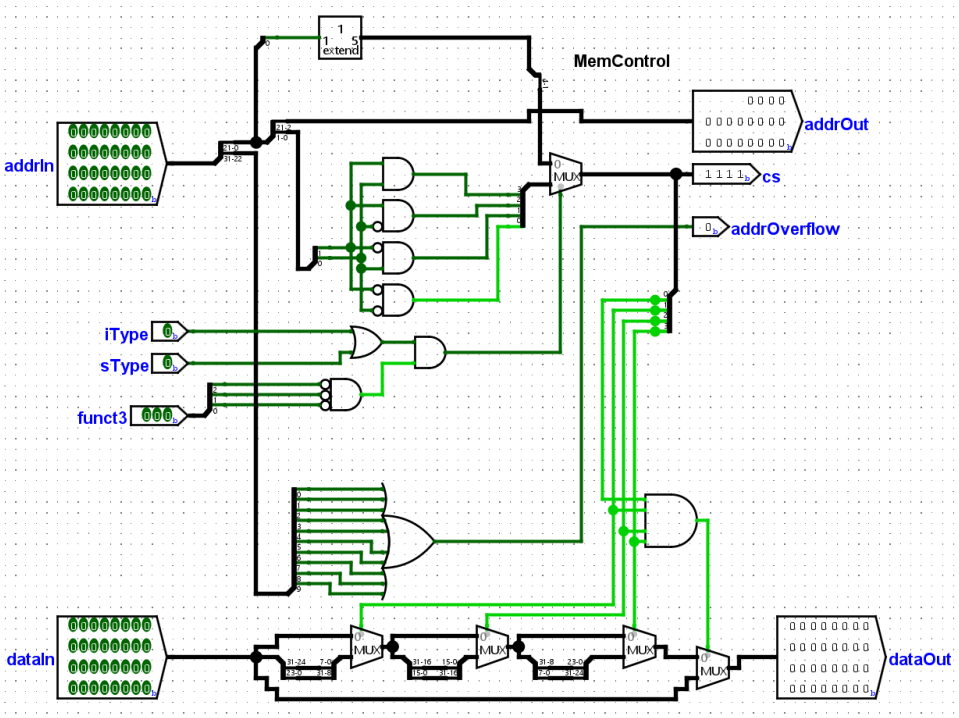
have access to correct functionality). An intermediate circuit to generate the 4-bit ALU opcode is necessary because it is not inherently a part of the instruction. AluCompControl only needs funct3, funct7, and mistype because as seen in the table, these fields hold enough unique information to generate the correct ALU opcode. Since at least one of the types (R-type) uses both funct3 and funct7, they are both needed in AluCompControl. As can be seen in the table without funct7 there would be an overlap of funct3 values of R-type.



The final subcircuit in decode is MemControl. The purpose of MemControl is to recognize addresses that are out of bounds (above 0xFFFFF bytes), to generate the correct value of cs according to specifications when executing lb and sb, and to adjust the data to be stored in memory to make sure it works correctly when executing sb. This adjustment consists of a shift in bits to make sure the two least significant bytes are always stored in the correct address. AddrOverflow is 1 when an invalid address is inputted, else 0.

There are a few more control signals to touch on before finishing up with decode. The first of which controls when the immediate value is used vs the value of xB. Per spec know that I, S, U, and B types all use the immediate, so when any of these types are used the mux signal is 1 and the immediate value is passed to the ALU. There is also a control for when to use xA vs the PC. This is necessary to use the ALU to calculate jumps when branching. As such, when the instruction is of type B this mux has a signal of 1 and


DecodeType
instType
rType
iType
sType
uType
bType
AluCompControl
opcode
lt
instType
funct7
funct3


MemControl
addrIn
addrOut
cs
addrOverflow
iType
sType
funct3
dataIn
dataOut
extend
MUX

takes the value of PC, when 0 takes xA. There is a final signal into a mux that appears in the bottom left of the memory block, but is actually part of decode. This controls a mux that passes in the output of the ALU when 0, and the 12-bit shifted immediate value when 1 which happens when we have U-type.

4 Execute

The third stage in the pipeline is Execute. Overall there is little (that can be seen from this circuit, the ALU itself is complex) happening. The ALU receives an opcode generated by AluCompControl and a shift amount, taken from the five least significant bytes of xB. Since the ALU does not handle less than, there is a mux that takes in the lt control signal, also from AluCompControl. When this signal is 0 the output of the ALU is taken, when it is 1 the output of the comparator is taken.

5 Memory

Like with Execute, in the Memory stage, there is relatively little going on that we can see. Here there are a few more control signals to discuss. MemWrite connects to WE to control when data is written to memory, this only happens when we have S-type and addrOverflow is not 1. MemRead connects to OE to control when data is read from memory, this only happens when we have instruction lw or lb.

6 Write-back

The final stage, Writeback, controls when data is written back to the register. The subcircuit ByteSelectControl is used to handle selecting the correct bytes when lb is executed according to the value of cs. It also takes in addrOverflow, which prevents a register out of range from being read. There is a mux that decides to take the value of execute when 1 or memory when 0. This signal, MemtoReg, is 1 when R-type, U-type, or I-type but not lb or lw. The output of this mux is sent to the register.

7 Testing

While making this processor I was constantly testing by running my GDC algorithm and a test file containing a multitude of tests and edge cases for each instruction. I would run these files on both my circuit and the RISC-V interpreter and manually compare the results. Most of these edge cases consisted of doing computations with zero, 1, INT_MAX, or INT_MIN. Other notable edge cases I checked for were overflow/underflow for any instruction that was added or subtracted, and ensuring that memory addresses out of bounds were not written to or read from. After finishing the processor I started trying to use a test script. I ran into some problems doing this but eventually got everything converted. Finally, I added my GDC algorithm to the test script because it was a great random test for all but four (and, sra, and, and sb). I added a few random tests for these four, then ran the entire random test in the interpreter. I took the final values in the registers and used them as the expected values. Together these tests convinced me that I had a fully functioning RISC-V processor.