Cassandra Sziklai
CS 3410
2 March 2024

P2 Final Design Document

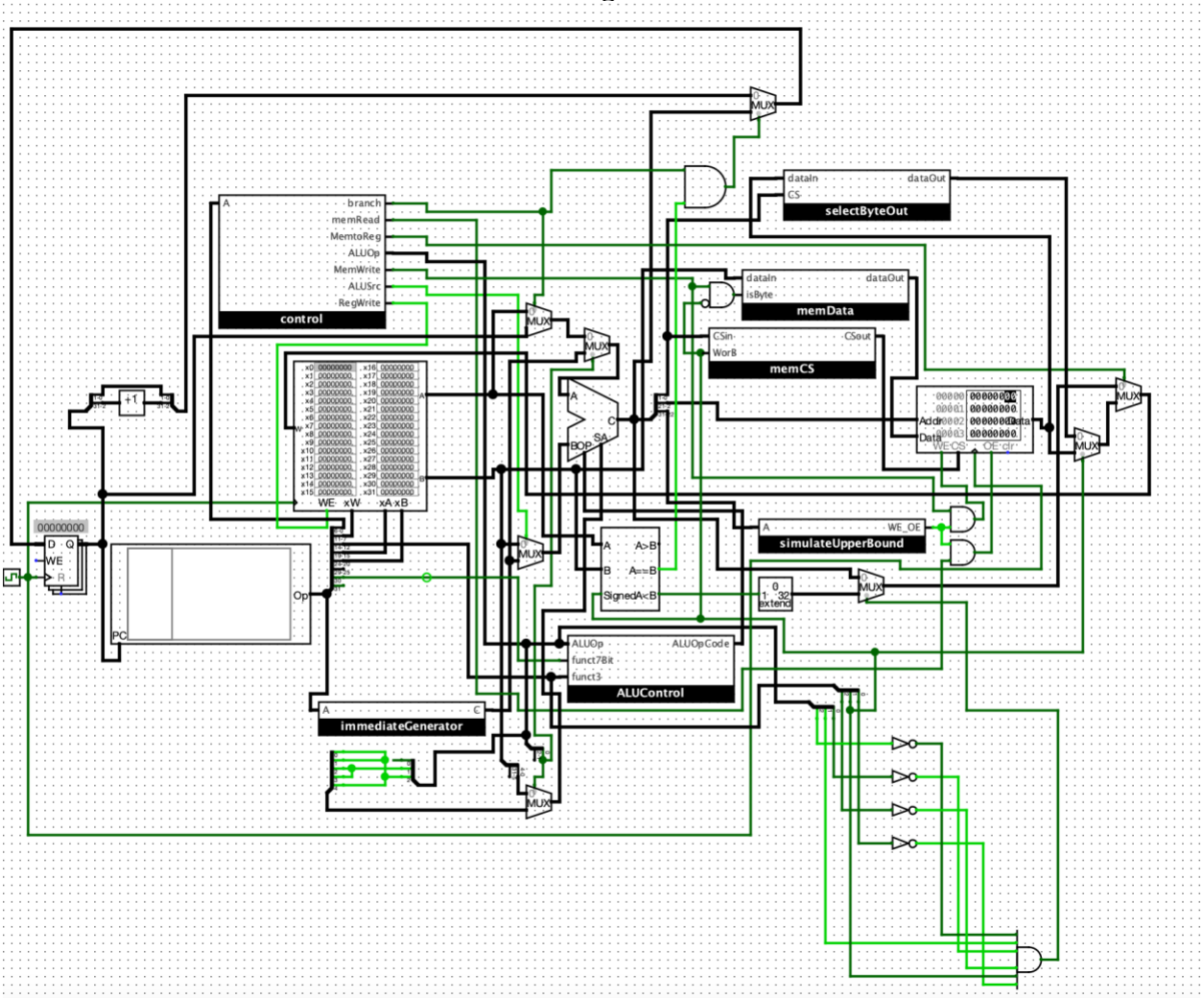


Fig 1.1: Block diagram of entire processor

The following describes how I implemented each stage of the processor.

Fetch

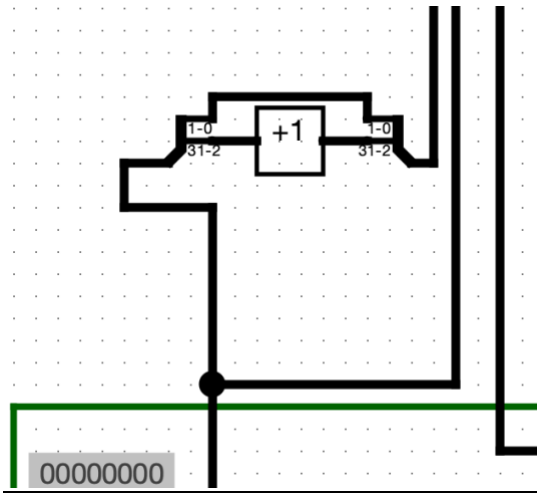


Fig 1.2: Incrementor used to compute PC + 4

To implement fetch, I used the incrementor on bits 2-31 to add 4. The first two bits stay the same, as shown in Fig. 1.2. For branches, the next instruction number is computed by using the ALU to add PC + imm.

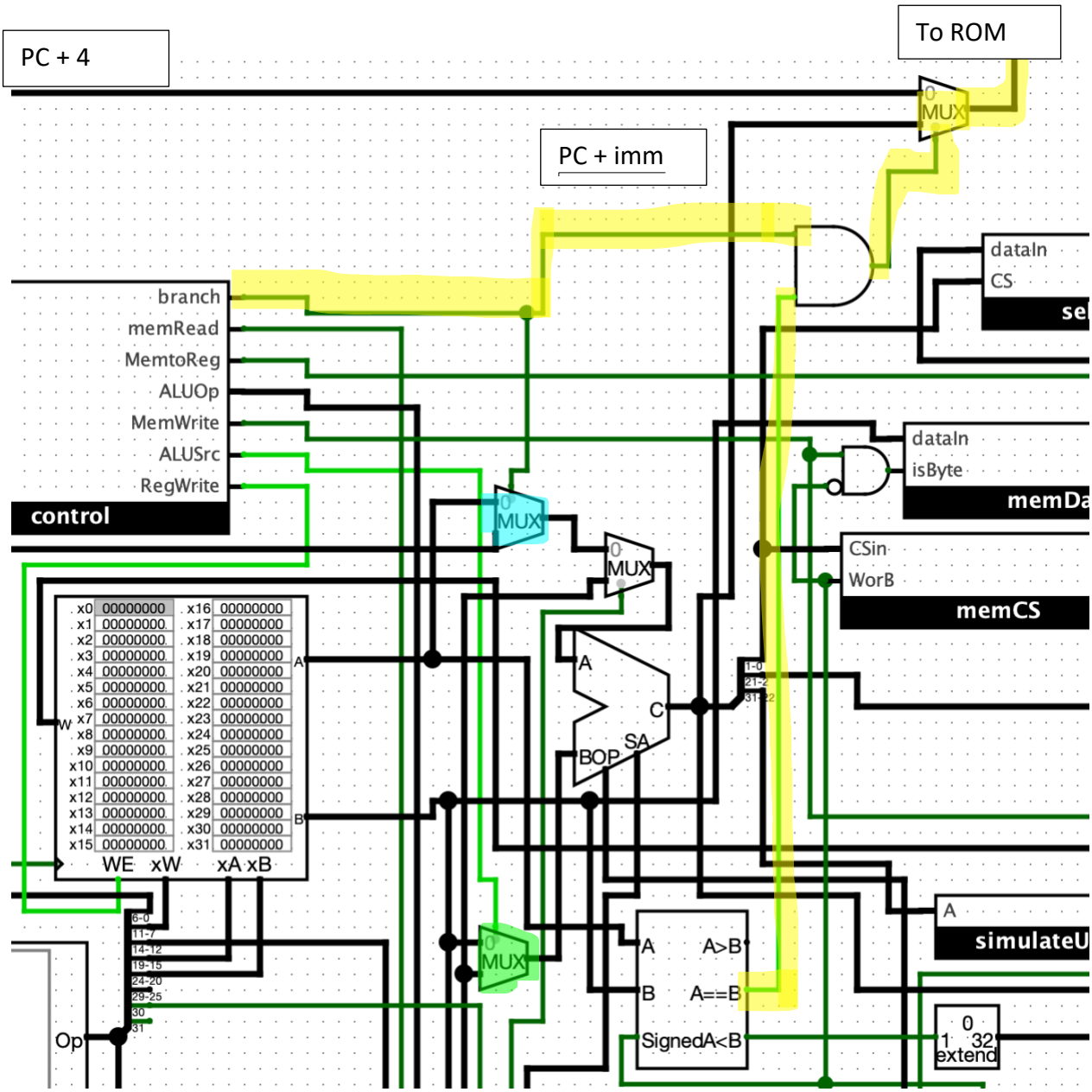
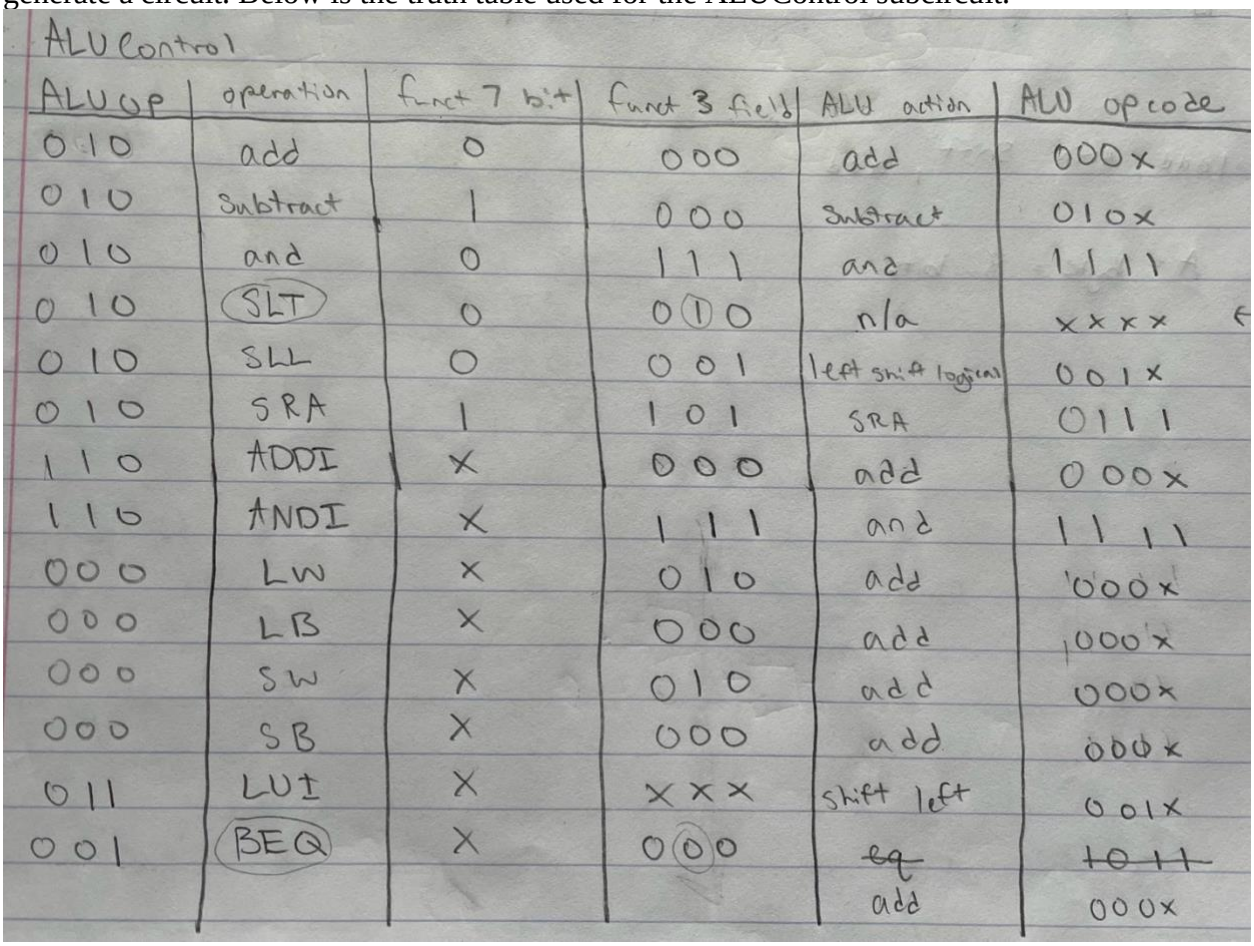


Fig 1.3: Control logic for branch instructions. Pay special attention to the highlighted parts of the diagram as they relate to the BEQ logic. As one can see, 'branch' and A=B must be true in order for the control signal to be 1. The yellow highlighted mux selected between PC +4 or PC + imm and outputs the result back to the ROM (not depicted).

For BEQ, two muxes control the inputs A and B to the ALU. For A, the mux (highlighted in blue in Fig 1.3) decides between the PC or A from the register based on the branch signal. For B, another mux (highlighted in green) decides between the generated immediate from the subcircuit immGen or B from the register based on the ALUsrc signal. Then after computing the sum of PC + imm, another mux (highlighted in yellow) decides whether to use this as the next PC vs the PC + 4 based on both the branch control signal and the equal signal from the comparator.

<u>Decode</u>

Decoding the instruction involves the subcircuits control and ALUControl. Control takes in the 7-bit opcode and creates signals for whether to write to a register, what the second ALU operand will be, whether to read from memory, etc. This is explained more in depth in question 3 at the end. From there, the ALUOp combined with the func3 and func7 bits are used to generate a 4-bit ALU opcode. I created truth tables for my desired functionality and fed them to Logism to generate a circuit. Below is the truth table used for the ALUControl subcircuit.



ALU Control

ALUOp	operation	funct 7 bit	funct 3 field	ALU action	ALU opcode
010	add	0	000	add	000x
010	subtract	1	000	subtract	010x
010	and	0	111	and	1111
010	SLT	0	010	n/a	xxxx
010	SLL	0	001	left shift logical	001x
010	SRA	1	101	SRA	0111
110	ADDI	x	000	add	000x
110	ANDI	x	111	and	1111
000	LW	x	010	add	000x
000	LB	x	000	add	000x
000	SW	x	010	add	000x
000	SB	x	000	add	000x
011	LUI	x	xxx	shift left	001x
001	BEQ	x	000	~~eq~~	~~1011~~
				add	000x

Fig 2.1: truth table for ALUControl subcircuit

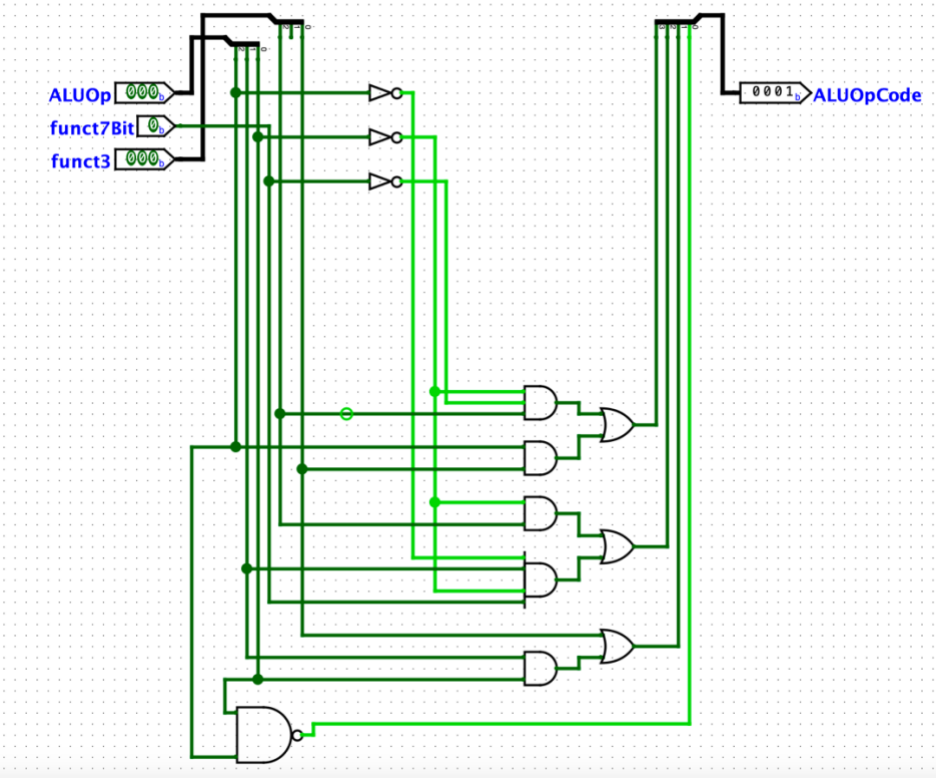


Fig 2.2: Subcircuit for ALUControl

Execute

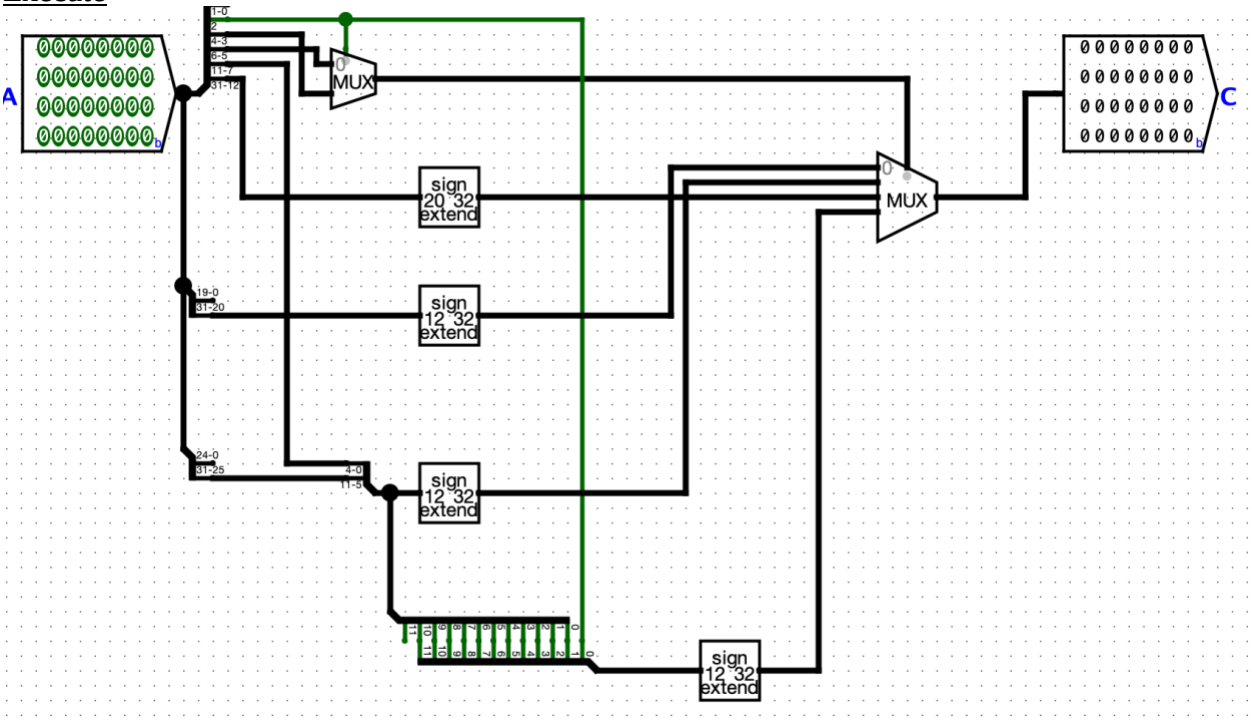


Fig 3.1: Subcircuit for immediateGenerator

Part of the logic for execute is carried out in the subcircuit immediateGenerator, which generates the necessary intermediates for all instruction types, outputting a 32-bit number which can be used by the ALU. First, the mux selects based on bit 2 of the instruction. Bit 2 is only 1 when it is a U-type instruction. If it is, we want bits 4-3 (which will be 10 in U-type instructions) to be the selector bits for the second mux. Else, the selector bits come from bits 6-5 of the instruction. This adequately groups the pattern of where the intermediates are located for each of the four groups (R-type and S-type are 01, I-type is 00, U-type is 10, B-type is 11). This subcircuit appropriately combines intermediate fields if necessary and sign extends to make the output 32 bits.

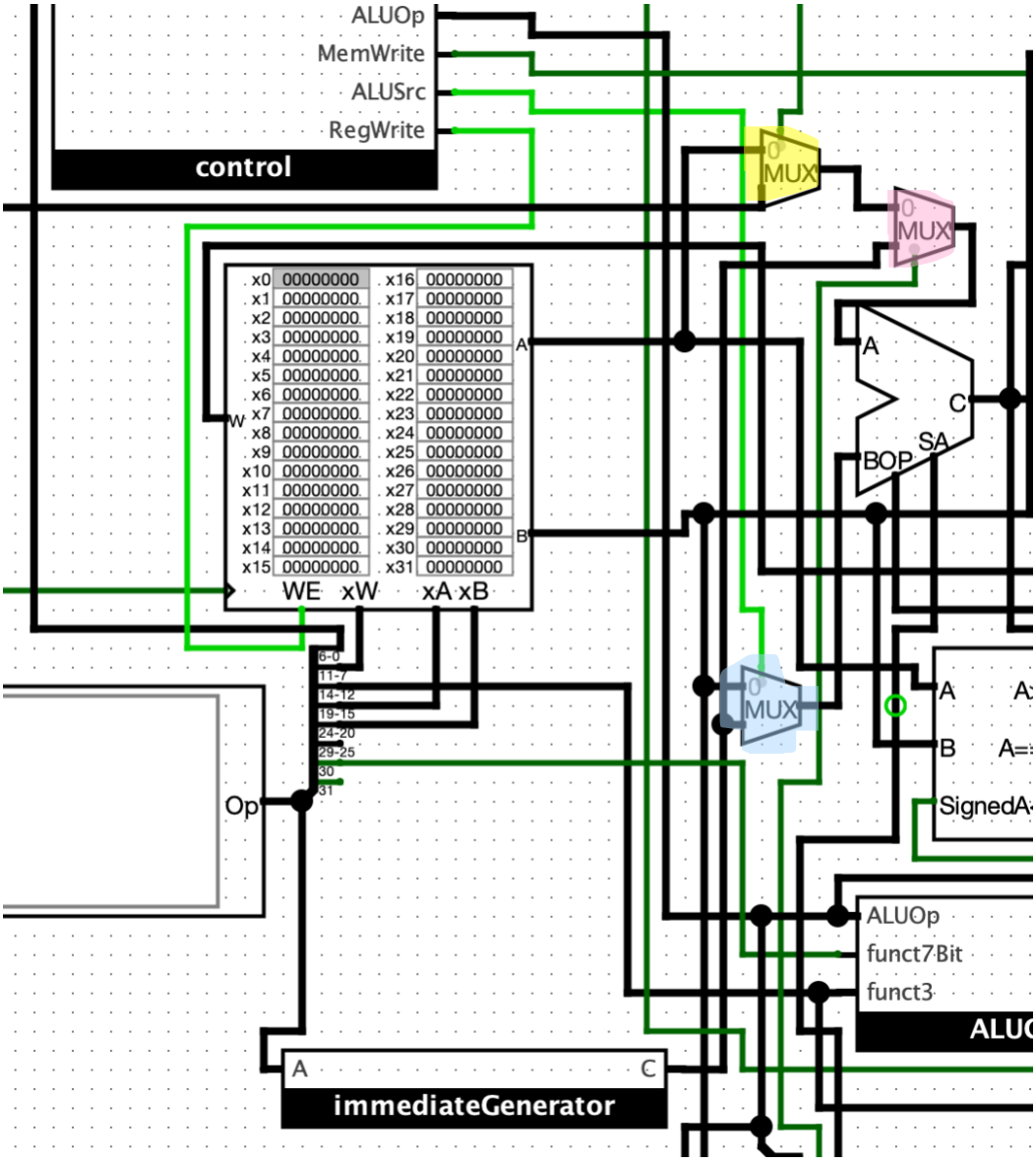


Fig 3.2: Choosing inputs for the ALU

When deciding what the input to the ALU will be, there are a few muxes involved. The yellow mux decides between the PC and A from the register (discussed in Fetch section). The pink mux then decides between A and the result from immediateGenerator. This is based on bit 0 of ALUOp, which distinguishes LUI instructions from everything else besides BEQ. The blue mux

chooses between B from the register and the output of immediateGenerator. The signal is ALUsrc, which comes from control.

The result of the ALU operation is fed to three different places. One is a mux for BEQ (discussed in Fetch section above). In addition, it is fed to RAM (discussed next in memory section). Additionally, it is fed to a mux that decides between it and the output of the comparator. The control signal for this is based on the 3 bit ALUOp and func3 fields, which are both 010 for SLT, which uses the comparator.

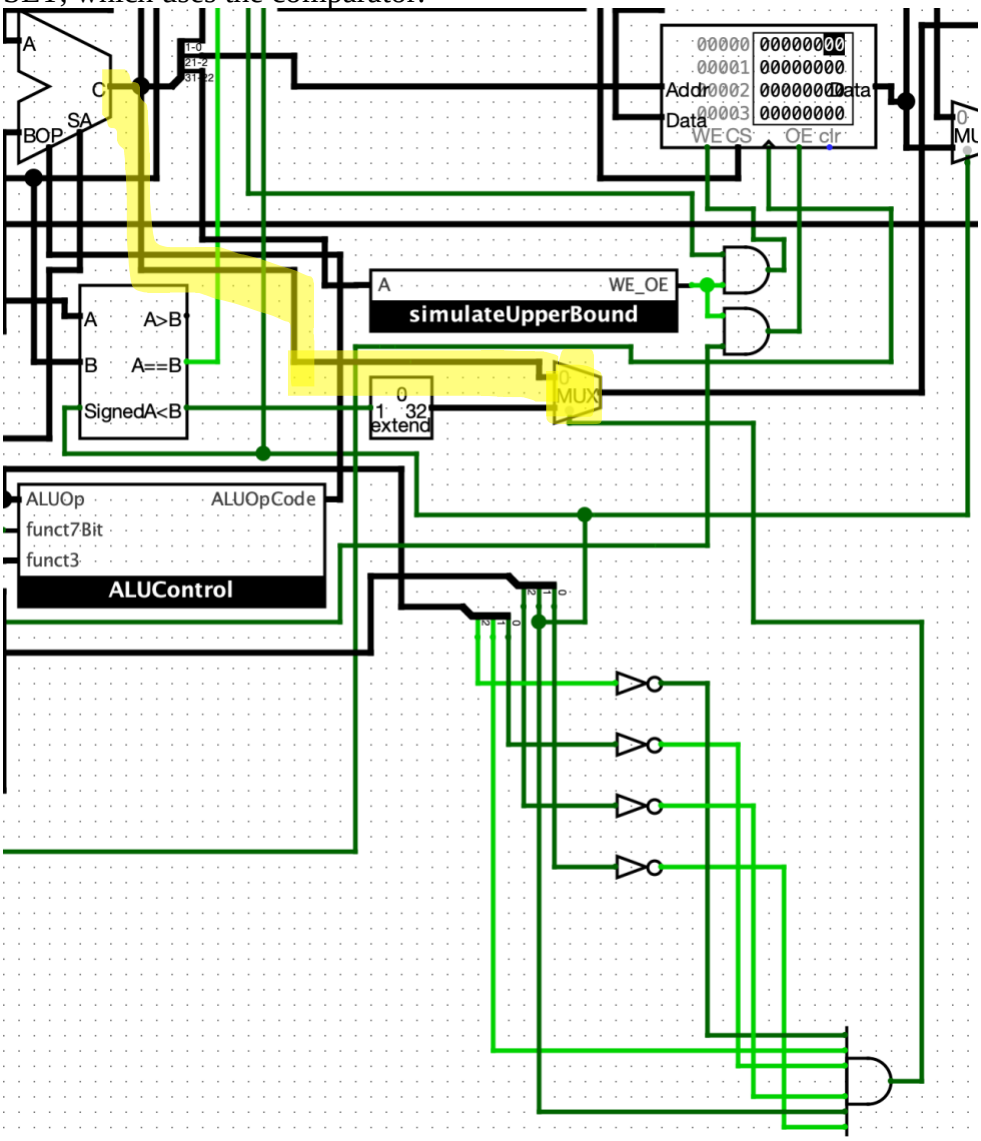


Fig 3.3: Choosing between result from SLT or comparator.

Memory

For SW and SB, my subcircuit memCS determines what CS should be based on the least two significant bits of the output of the ALU and bit 1 of func3 tells whether we're dealing with a word or a byte. If it is a word, CS will be 1111 in order to capture the entire word. As for the data that I want to write to memory, the subcircuit memData determines whether to use write the entire word or a byte. Since we always want to write the least significant byte to memory, my memData subcircuit duplicate the least significant byte. Then there is a mux to decide whether to use that repeated byte or the original word as the data to write to memory.

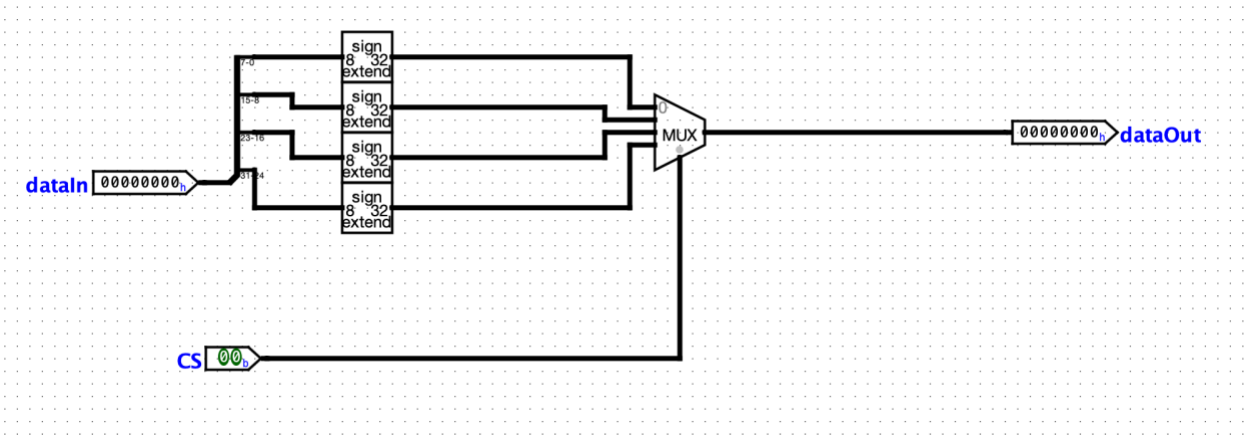


Fig 4.1: selectByteOut subcircuit

For load byte instructions, I have a subcircuit selectByteOut, which appropriately sign extends each byte and feeds it to a mux which selects which to use based on the 2-bit original CS code. For load word instructions, the output from the 'Data' field in RAM can simply be written back to the register. Thus, there is a mux to select which output to use based on bit 1 of func3.

Finally, to simulate the upper bound of memory, I detect if there are any ones in the upper 10 bits of the output of the ALU. This is done using a simple 10-input NOR gate in my simulateUpperBound subcircuit. If not all the 10 most significant bits are zero, then we don't want to load/store. Thus, WE and OE should be zero. This is handled using 2 AND gates; one for WE and another for OE, that take the output of simulateUpperBound and the memWrite and memRead from control.

The yellow mux in fig 4.2 below chooses between the unaltered data output from the RAM and the sign extended version from selectByteOut. The former is for LW, the latter is for LB, and the control is bit 1 of func3.

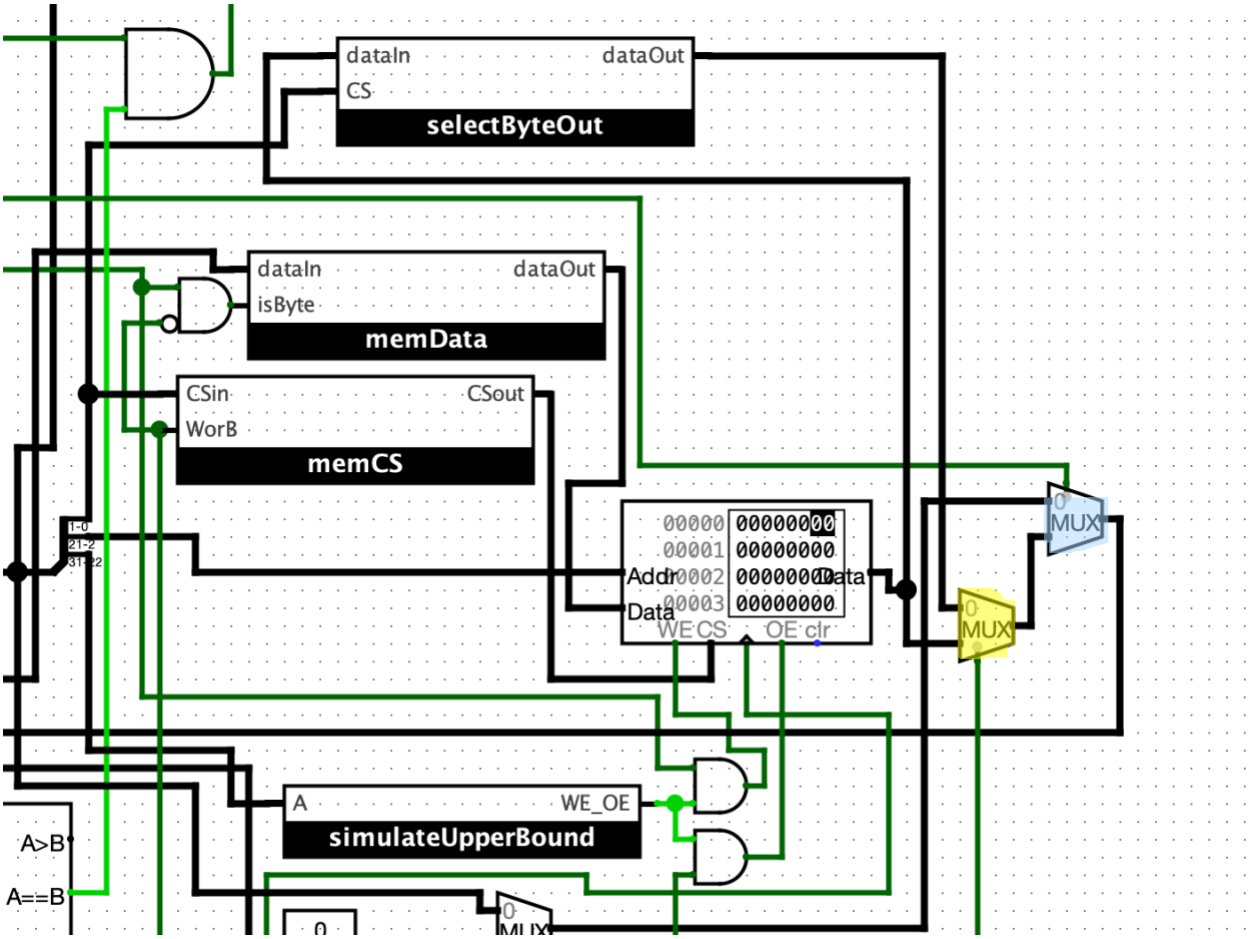


Fig 4.2: Overview of Memory logic

Writeback

To write back the result of the instruction, the RegWrite signal from control is fed into the register to control whether a write to the register is required or not. The blue mux from figure 4.2 above decides what finally gets written back to the register. That is, either something from memory or the mux output of ALU vs. comparator (see fig 3.3).

1. This choice doesn't affect our circuit because we have control bits in place that make sure that whatever value we read from rs2 is disregarded in our circuit. In other words, after processing the op code in the control subcircuit, we can control what we do with each output from different parts of the circuit and only use the outputs that we need to execute the instruction.
2. Feeding random data to the 'write register' field doesn't affect the store operation because as long as we have WE disabled, nothing will be written.
3. MemtoReg = 0 means the value to be written to the register comes from the ALU. When MemtoReg = 1, this means that the value to be written to the register comes from RAM. When RegWrite = 1, this means that the register will be written to using the value from the write data input. MemRead = 1 tells us that we want to read from the specified memory address. MemWrite = 1 means we want to write to the specified address in memory. When Branch = 1 and the comparator outputs a 1 after testing the equality of

the two values, PC + imm is chosen for the next instruction. If Branch = 0, then PC + 4 is used to fetch the next instruction.

4. We use an intermediate ALUOp in order to differentiate between the different types of instructions. For mine, I split it up into different categories: an ALUOp of 000 was for loads and stores because they needed to do add, 001 was for B-type testing equality, 010 was for R-type and ANDI and ADDI, and 011 was for LUI. This makes more sense that using the 7-bit opcode because the categories I picked don't exactly align with the opcode. For instance, LW, SW, SB, LB want to have add for the ALU, but they do not have the same 7-bit opcode.
5. ALUOp, funct3, and funct7 are enough for input to ALUControl because they have just enough to differentiate the different functionality we want. This is all we need to know in order to determine which ALU opcode to produce. For example, the funct3 field in R-type tells which opcode is needed. But add and subtract both have it as 000. However, they differ in the second to last bit of func7, so we can use that bit to determine whether to add or subtract. Because types other than R-type don't have a funct7, we can know to disregard it or not based on the ALUOp value.

Explanation of Testing Strategy:

For testing, I first started out by writing basic test cases for all the operations excluding the ones involving memory. I wanted to test first that my logic for fetch, decode, execute, and writeback were correct before I moved on to the memory part. After I found that it worked, I implemented memory and wrote test cases for that. Next, I wrote two other test files, making sure to cover all cases as well as add a robust amount of edge cases. Some edge cases included testing the upper bound of memory, having immediates that were large enough to be stored in both the imm7 and imm5 fields, and storing/loading bytes at different locations within a word.