00:     00100093   addi x1 x0 1
00001111222233334444555566660000
00000000000100000000000010010011
<-imm[11:0]->< rs-><f><rd -><- opc->

04:     00800113   addi x2 x0 8
08:     00202223   sw x2 4 x0
00001111222233334444555566660000
0000000000100000001000100**0100011**

(1) OPCODE: 0100011
candidates: SB, SH, SW
Use funct3, funct7 to determine the specific instruction
They are all S-type instructions so we know how to split the instruction.

00001111222233334444555566660000
00000000001000000**010**00100**0100011**
(2) funct3: 010
instruction: SW
So we know now that the instruction of interest is SW
00000000001000000**010**00100**0100011**
<-im -><rs2 ><rs1 >010<-im -><opcode>
missing upper bits are sign-extended before computation (since it is a 32-bit machine, we can address 2^32 bytes)

If funct3 are also identical, func7[5] tells the difference.

So in general, we can create an enum for OPCODE with six different types.

____________________Psuedocode____________________

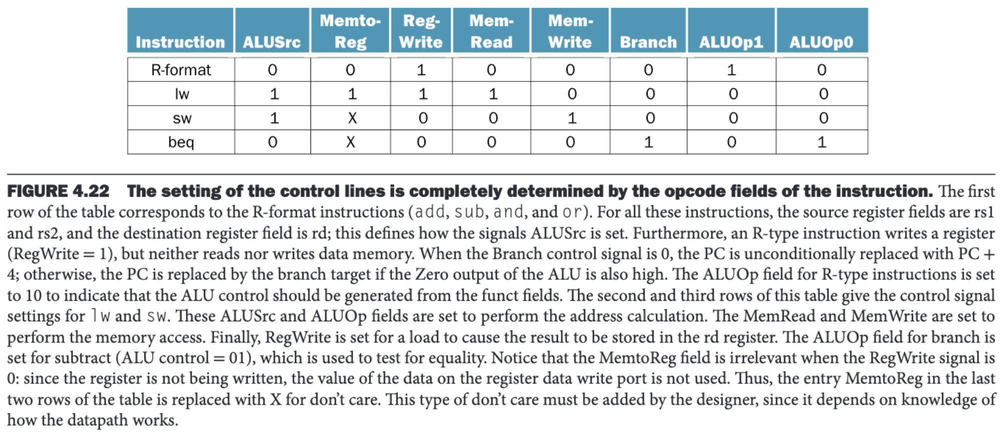
```
enum OPCODE
type I-type
type J-type
type B-type
type S-type
type U-type
type R-type

check instruction[-7:] with OPCODE:
        get type
```

```
func split_instruction(get_type):
        switch (get_type) {
        case R-type:
                r-type-splitter(get_type)
                break
        case I-type:
                i-type-splitter(get_type)
                break
        // do the same for all cases
        default:
                throw ValueError
        }
```

```
Use funct3 further narrows down the instruction
if funct3 is equal:
        use funct7[5]
```

____________________End____________________

That was the general idea of my program.



Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
lw	1	1	1	1	0	0	0	0
sw	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

FIGURE 4.22 The setting of the control lines is completely determined by the opcode fields of the instruction. The first row of the table corresponds to the R-format instructions (add, sub, and, and or). For all these instructions, the source register fields are rs1 and rs2, and the destination register field is rd; this defines how the signals ALUSrc is set. Furthermore, an R-type instruction writes a register (RegWrite = 1), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high. The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct fields. The second and third rows of this table give the control signal settings for lw and sw. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegWrite is set for a load to cause the result to be stored in the rd register. The ALUOp field for branch is set for subtract (ALU control = 01), which is used to test for equality. Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

I modeled my Control Table to be something similar to the one above provided by the textbook as well as those covered during week 2 discussion.

This is the controller table I used for the program, ensuring that different opcodes yield different controller outputs.

Here, func denotes that I am referring to the funct3 and funct7 (ADD vs. SUB) regions to determine the exact control function to send out.

instr	opc	regW	aluSrc (imm)	branch (or pcSrc)	memR	memW	memToReg	AluOp	Jump
ADD	`0110011`	1	0	0	0	0	0	func → ADD	0
SUB	^	1	0	0	0	0	0	func → SUB	0
XOR	^	1	0	0	0	0	0	func → XOR	0
ANDI	`0010011`	1	1	0	0	0	0	func → AND	0
ADDI	^	1	1	0	0	0	0	func → ADD	0
SRA	`0110011`	1	0	0	0	0	0	func → SHIFT	0
LW	`0000011`	1	1	0	1	0	1	ADD (PC + offset)	0
SW	`0100011`	0	1	0	0	1	0	ADD (PC + offset)	0

BLT	1100 011	0	1	1	0	0	0	SUB (rs1 - rs2)	0
JALR	1100 111	0	1	0	0	0	0	ADD (rs1 + offset)	1

In this project, I created separate classes for controller and ALU since those two units are distinct from the existing CPU structures. In reality, each component would become a module (in HDL), but in C++, I decided that I could emulate the effect by using enums to set the number of possible ALU operation codes and possible combinations of control signals as described by the table above.

For the ALU operation code, I used the following:

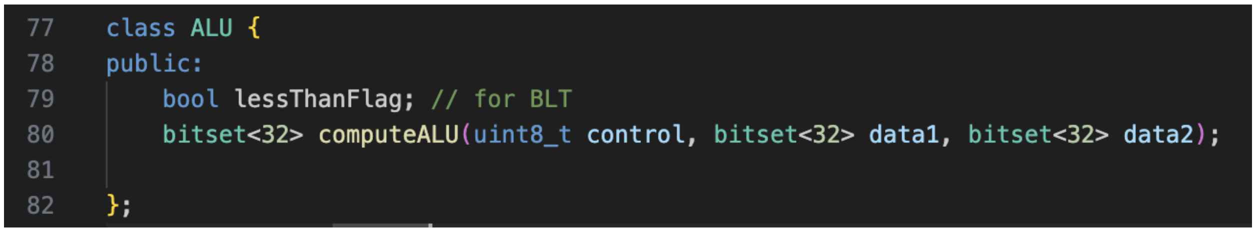
IDLE = 0b0000,
AND = 0b1111,
XOR = 0b0001,
ADD = 0b0010,
SUB = 0b0110,
SHIFT = 0b1001

For the operation types (for generating distinct sets of control signals), I used:

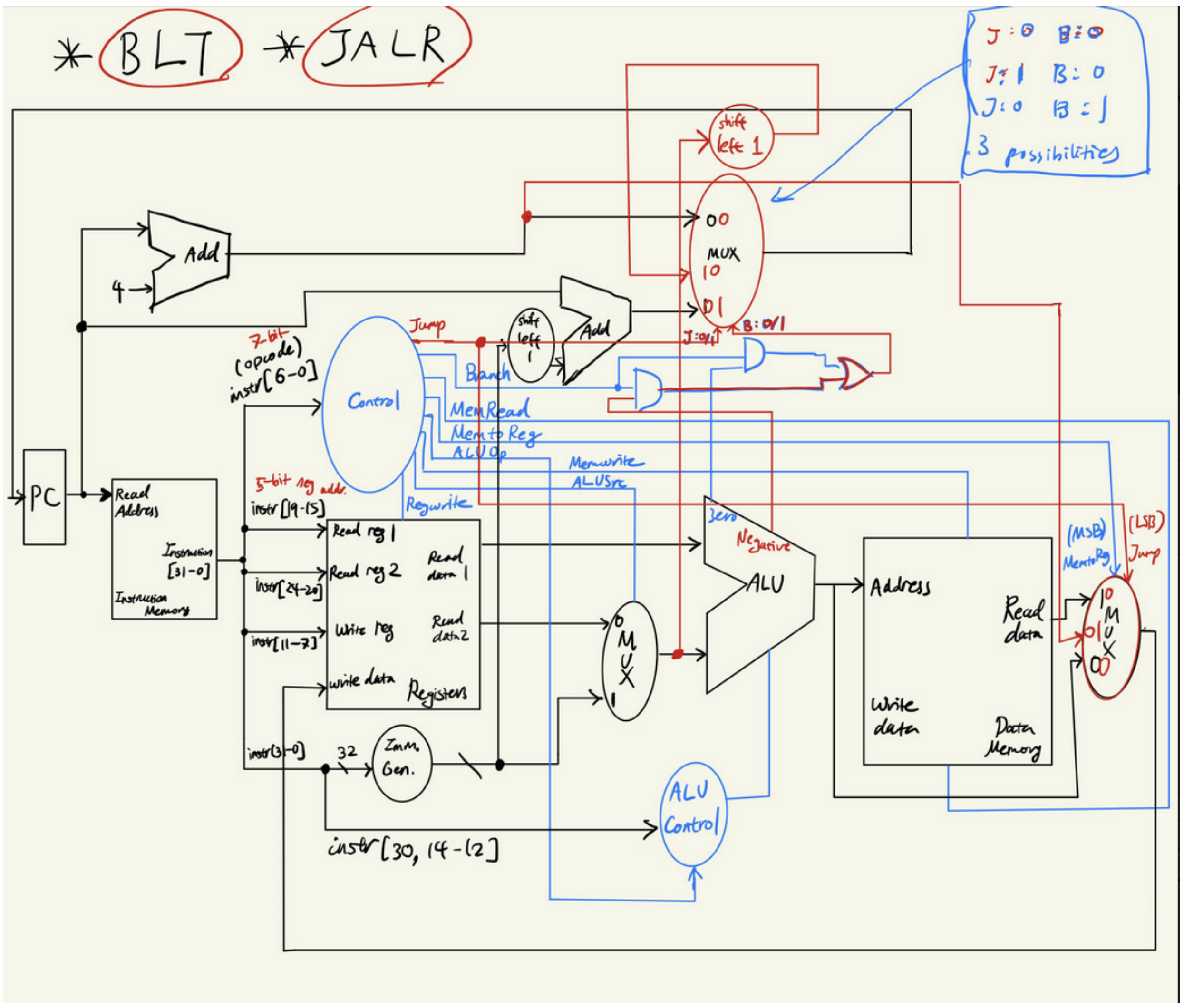
- R-type
- I-type
- Load
- Store
- B-type
- J-type

I separated load and store although they follow I-type and R-type format because I had to implement separate operations for them during the Memory stage (CPU::Memory()).

In ALU class, I included lessThanFlag to implement BLT instruction:



```
class ALU {
public:
    bool lessThanFlag; // for BLT
    bitset<32> computeALU(uint8_t control, bitset<32> data1, bitset<32> data2);

};
```



This datapath incorporates all the necessary features for R- and I-type instructions as well as LW, SW, BLT, and JALR instructions. I used if-else if-else statements to express multiplexors. Fetch, Execute, Decode, and Memory stages were written as functions in CPU class. Writeback is incorporated in the Run() function, in which I call Execute, Decode, and Memory to emulate a single-cycle processor.