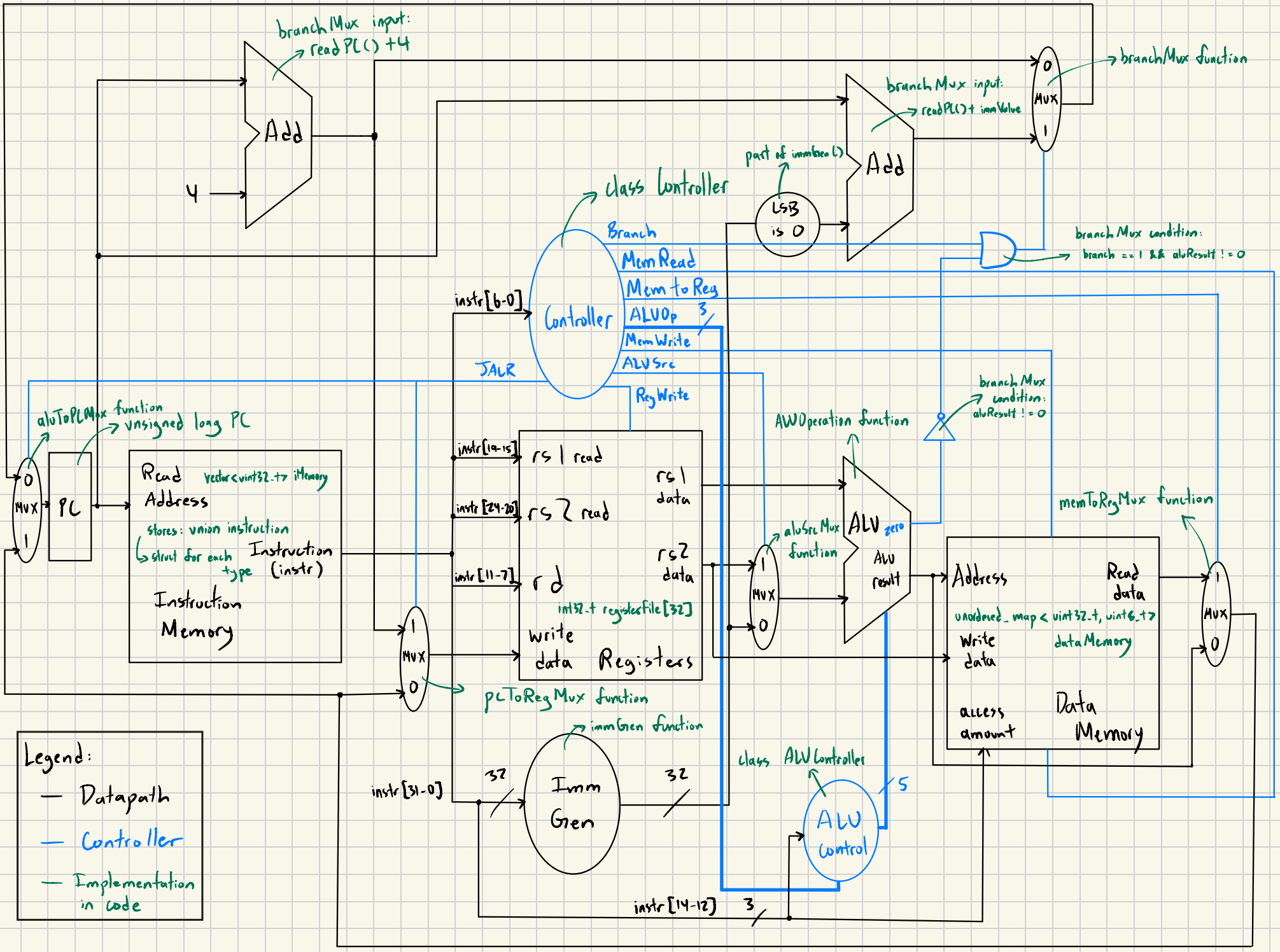
branch Mux input:
read PC() +4
Add
4
branch Mux input:
read PC() + immValue
Add
part of immGen()
LSB is 0
0
MUX
1
branch Mux function
branch Mux condition:
branch == 1 && aluResult != 0
class Controller
Branch
MemRead
MemtoReg
ALUOp
3
MemWrite
ALUSrc
RegWrite
JALR
Controller
instr[6-0]
aluToPCMux function
unsigned long PC
0
MUX
1
PC
Read Address
vector<uint32_t> iMemory
stores: union instruction
struct for each type
Instruction (instr)
Instruction Memory
instr[19-15]
instr[24-20]
instr[11-7]
rs1 read
rs2 read
rd
rs1 data
rs2 data
int32_t registersFile[32]
write data
Registers
1
MUX
0
pcToRegMux function
ALUOperation function
branch Mux condition:
aluResult != 0
aluSrc Mux function
1
MUX
0
ALU
zero
ALU result
memToRegMux function
Address
Read data
unordered_map<uint32_t, uint8_t>
Write data
data Memory
access amount
Data Memory
1
MUX
0
immGen function
instr[31-0]
32
Imm Gen
32
class ALUController
ALU control
5
instr[14-12]
3
Legend:
Datapath
Controller
Implementation in code

Table of Control Signals

Instruction	Opcode	RegWrite	ALUSrc	Branch	MemRead	MemWrite	MemToReg	JALR	ALUOp
R-type	0110011	1	0	0	0	0	0	0	2
I-type	0010011	1	1	0	0	0	0	0	3
LBU/LW	0000011	1	1	0	1	0	1	0	0
SH/SW	0100011	0	1	0	0	1	0	0	0
BNE	1100011	0	0	1	0	0	0	0	1
JALR	1100111	1	1	0	0	0	0	1	0
LUI	0110111	0	1	0	0	0	0	0	4

Summary of data path and controller design:

I kept the data path and controller mostly the same from the in-class example. To support the new instruction set, I made a few changes described below.

1. To implement BNE, I flipped the output of the zero gate from the ALU with a NOT gate to jump when the two operands are not equal.
2. To implement SH and LBU, I added an extra input to my data memory , passing in funct3. This enables the data memory to decide how many bytes to read/write from memory per instruction.
3. To implement JALR, I added an extra control signal (JALR). When JALR is 1, multiplexers aluToPCMux and pcToRegMux update PC to rs1 + imm and rd to PC + 4 respectively.

When implementing in C++, my goal was to keep the code as simple as possible while being accurate to the design. For example, I only made classes for components that store other components. I had 3 classes: CPU, Controller, and ALUController. The CPU class contains all of the datapath components (i.e. instruction memory, ALU, dataMemory, registers, etc.). The Controller class contains all of the multiplexers and control signals for the multiplexers. The ALUController class translates ALUOp and funct3 to the ALU control signal passed into the ALU.

RISC-V Reference

ALUOp key: 000 ADD
001 SUB
010 R-type
011 I-type
100 U-type

RISC-V Instruction Set

Core Instruction Formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Integer Instructions

ALUOp
ALU control sig : XXX XX

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	