EECS 151/251A Homework 4 Solution

Problem 1: RISC-V ISA

For this part, it will be helpful to refer to the RISC-V Green Card. We will be using RV32I, the 32-bit RISC-V integer instruction format.

(a) RV32I instructions are stored in 32 bits in Instruction Memory. For the CPU Project, you will be implementing logic to decoding them. Here are some practice converting between instruction and their binary/hexadecimal encoding. Note while it is possible to do this exercise with venus or gcc, we encourage you to do this by hand to get more familiar with the instruction format.

i. What is the instruction encoded by 0x00219223?

ii. What is the instruction encoded by 0xEE151917?

iii. What is the encoding (in hexadecimal) of slt x5, x2, x3?

iv. What is the encoding (in hexadecimal) of lb x15, 8(x4)?

(b) Now let's take a look at some of the nuances of the ISA.

i. For branch instruction bltu x2, x3, 5. if current PC is 0x40 and that value in x2 $<$ x3, what is the PC value after the instruction is executed?

ii. There are a number of pseudo instructions in the ISA for programmer's convenience. For example, **la x1, symbol** loads the address represented by symbol into the x1 register. Suppose you are writing a RV32I assembler which replaces pseudo instruction with base instruction, would **addi x1, x2, symbol[11:0]** work for the case symbol is 0xABC (assuming x2 stores the PC value)? Why? If not, what base instruction(s) should we use to make **la x1, symbol** work here?

iii. Would using addi instruction still work if the symbol is 0xEEC5151? If not, what base instruction(s) should we use to make **la x1, symbol** work here?

iv. (251A Only) If the symbol is 0xEEC5951, what will be the base instruction(s) for **la x1, symbol**? Write out the value for the immediate explicitly here. *Hint:* is 0xEEC5951 equivalent of adding 0xEEC5000 and 0xFFFFF951?

v. **j label**; **jr rs1**; **ret** are common pseudo instructions used in program control flow. What are the corresponding base instruction for them?

(c) Now let's attempt to implement executing R type instructions in behavior verilog. For example, wire [31:0] ALU_out = a[31:0] + b[31:0]; implements the **add rd, rs1, rs2** instruction, assuming that a represents value from rs1, b represents value from rs2, and ALU_out will eventually be written back to rd. For the CPU project, you will implement all the R type instruction functionality in the ALU.

i. Implement bitwise xor **xor rd, rs1, rs2**.
wire[31:0] ALU_out = ____;

ii. Implement logical left shift **sll rd, rs1, rs2**.
Hint: How many bits of rs2 do we care about?
wire[31:0] ALU_out = ____;

iii. Implement logical right shift **srl rd, rs1, rs2**.
Hint: Is this sign extended or zero-extended? Remember in verilog, $signed denote whether to treat a number as signed or not
wire[31:0] ALU_out = ____;

iv. Implement arithmetic right shift **sra rd, rs1, rs2**.
Hint: Is this sign extended or zero-extended? Remember in verilog, $signed denote whether to treat a number as signed or not
wire[31:0] ALU_out = ____;

Solution:

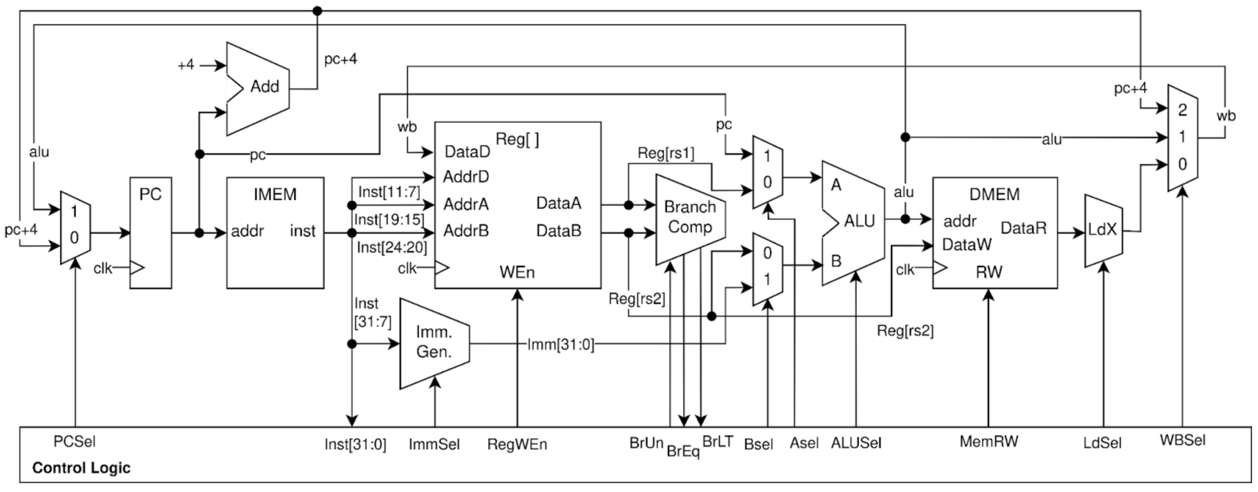
(a)
  i. sh x2, 4(x3)
  ii. auipc x18, 0xEE151
  iii. 0x003122B3
  iv. 0x00820783

(b)
  i. PC = 0x44. Although the immediate here is 5, which is 0b0000_0000_0101, when the type SB instruction is encoded, the last bit of the immediate is ignored and 11 of the 12 bits are stored. When it is decoded, the encoded immediated will be decoded as 0b0000_0000_010 < < 1 which is 0b0000_0000_0100 = 4. Hence, PC becomes 0x44 once the branch is taken. The reason this one bit bit-shift exists in the ISA is an optimization to store less information given PC in RISC-V must be incremented at least every 2 bytes (RV16 instructions are 16 bits).
  ii. Yes, it would work. Since in this case we only care about the last 12 bits of 0xABC and addi allow for 12 bit immediate.
  iii. No, using addi won't work here since we have more than 12 bits. We will need to use 2 base instructions: **auipc rd, symbol[31:12]** to load the upper 20 bits and then **addi rd, rd, symbol[11:0]** for the lower 12 bits. Here it would be **auipc x1, 0xEEC5** and **addi x1, x1, 0x151**.
  iv. Here clearly we need both auipc and addi as we shown in the previous part. However, note that last 12 bits 0x951 = 0b1001_0101_0001 start with a 1 in its MSB. Note that the lower immediate is sign extended before adding, thus the upper 20 bit immediate must account for the equivalent -1 lower 12 bit sign extended is causing. Hence, here the pseudo instruction is broken into **auipc x1, 0xEEC6** and **addi x1, x1, 0x951**.
  v. j label -> jal x0, label. jr rs -> jalr x0, rs, 0. ret -> jalr x0, x1, 0

(c)
  i. wire[31:0] ALU_out = a[31:0] ^ b[31:0];
  ii. wire[31:0] ALU_out = a[31:0] << b[4:0]; Note we only use the last 5 bits of b as we can only shift at most 32 bits.
  iii. wire[31:0] ALU_out = a[31:0] >> b[4:0];
  iv. wire[31:0] ALU_out = $signed(a[31:0]) >>> b[4:0]; we need to keep the sign for sign extension since it is an arthematic shift.

Problem 2: Single Cycle Datapath

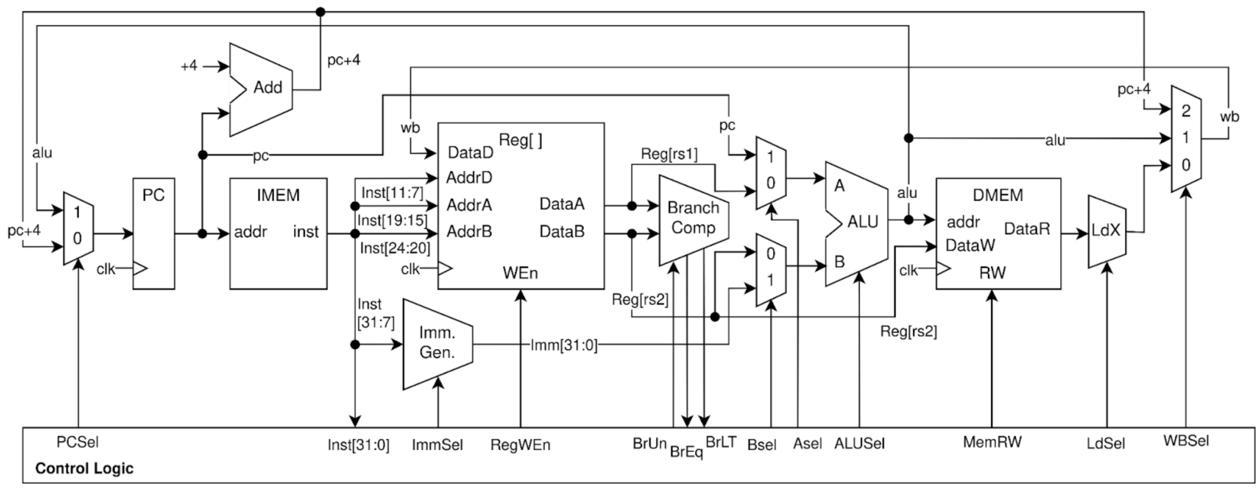
The diagram here shows an implementation of single-cycle RV32I datapath.



(a) Fill in the control signals when executing the following instructions. Use * to denote don't care.

Instruction	PCSel	ImmSel	RegWEn	Asel	BSel	ALUSel	MemRW	WBSel
add rd, rs1, rs2	0	*	1	0	0	ADD	READ	1
ori rd rs1 imm								
beq rs1 rs2 label (taken)								
jalr rd rs1 imm								
lhu rd imm(rs1)								
sb rs2 imm(rs1)								
auipc rd imm								

(b) Suppose we want to add support for the following instruction that describes: PC = rs2 + M[rs1+imm]. How might we modify our datapath to realize that? It might be helpful to draw it out.



(c) (251A Only) Can our datapath be modified to support CISC-style instruction that performs M[rd] = M[rs1] + M[rs2]? Or would it be better to break it down into smaller RISC instruction(s) and how would you do it? Justify your answer.

Solution:

(a) See table below.

Instruction	PCSel	ImmSel	RegWEn	Asel	BSel	ALUSel	MemRW	WBSel
add	0	*	1	0	0	ADD	READ	1
ori	0	I	1	0	1	OR	READ	1
beq (taken)	1	B	0	1	1	ADD	READ	*
jalr	1	I	1	0	1	ADD	READ	2
lhu	0	I	1	0	1	ADD	READ	0
sb	0	S	0	0	1	ADD	WRITE	*
auipc	0	U	1	1	1	ADD	READ	1

(b) You will need to extend rs2 and put an adder to add output of memory read with it, then feedback to the PCSel mux as a new input option.

(c) The overhead to modify this pipeline this Mem -> Mem CISC operation is too significant. We can break this complex instruction into two loads, one add, and one store without modifying our datapath. This example shows the power of RISC: able to break complex instruction / memory access sequence into a few simple instructions that can be executed on a smaller and simpler datapath.