Introduction

In this lab, you will enhance your existing 3-stage RISC-V processor with branching and jump instructions. These will require some changes to parts of your pipeline, but will greatly improve the range of programs your CPU will be able to accept.

Proposed Approach

This lab builds on the work you have done in the previous lab, and will require you to extend several parts in order to support branch and jump type instructions. This section will outline several major modifications you will want to explore in completing this lab.

You will want to **extend your instruction decoder module** (or whatever other system you were using for decoding instructions) to support the two new instruction formats (B-type and J-type). For both types these should result in one or more sign-extended immediate values being produced (for the jump target offset).

You will also want to **extend your control unit** to support the new instructions. This will probably require you to add new cases to deal with B-type and J-type instructions. You will also want to add an extra I-type case to deal with the `jalr` instruction.

Your **fetching logic will need to change** as well, in order to support jumps and branches. We recommend the modifications suggested in the lecture 5 slides, which will change `PC_FETCH` from incrementing each cycle, to a more complex design.

You will need to handle the following cases in your updated fetch logic:

- `PC_FETCH + 1` :: Normal next instruction.
- `branch_addr_EX` :: Address to jump to if the branch is taken. (An offset from the current PC value.)
- `jal_addr_EX` :: Jump target of a `jal` instruction. (An offset from the current PC value.)
- `jalr_addr_EX` :: Jump to address specified in the `jalr` instruction. (`rs1` + offset)

We also recommend creating a `stall_FETCH` and `stall_EX` set of control signals (likely tied to the control unit) as a way of propagating stalls across your pipeline. You can use the stall signals to short out the regfile write- enable signal to ensure the instruction already executing is a no-op.

We recommend making these modifications incrementally – just one group of instructions at a time. This can make it easier to spot errors in your implementation, since you will be testing the updated fetching and branch control logic as you go. A good place to start would be the unconditional jump instruction `jal`.

Note that to support using the HEX displays, you should re-use your `hexdecoder` module from the previous lab.

Design Requirements

- (i) Your CPU must implement a three-stage pipeline
  - At a given clock cycle, your CPU should be fetching one instruction, stalling or executing another, and possibly (if not stalled) writing the result of a third instruction to the register file.
- (ii) Your CPU must implement all of the R-Type instructions listed in the previous lab.
- (iii) Your CPU must implement all of the I-Type instructions listed in in the previous lab, plus `jal` as noted in *Appendix A*
- (iv) Your CPU must implement all of the U-Type instructions listed in the previous lab.
- (v) Your CPU must implement all of the J-Type instructions listed in *Appendix A*.
- (vi) Your CPU must implement all of the B-Type instructions listed in *Appendix* .
- (vii) You must write a complete suite of test cases that validate your CPU works
  - To check if your test suite works, try choosing an instruction at random, and modify your control unit to produce incorrect values for that instruction, then run your test suite. Did your test suite throw an error? If not, it is incomplete. You should complete this process at least a couple of times for different instructions.
- (viii) The instruction `csrrw x0 io2 rs1` should cause the value of register `rs1` to be displayed on the hex displays.
- (ix) The instruction `csrrw rd io0 x0` should modify the value of register `rd` to be set to the current switch value.
- (x) You must write a RISC-V assembler program, which you should save as `sqrt.asm`. This program should read a value from the switches (`io0`), and should the display the square root of that value as a decimal number on the HEX displays in (8,5) base 10 fixed point.
  - For example, to display 3.75, your program would show `00375000` on the hex displays.
  - Another example: an input of `17` (`18'b10001`) on the switches should cause a value of approximately `00412311` to be displayed on the hex displays.
  - Your program should produce the correct result within a tolerance of $\frac{1}{100000}$.

  - **Tip**: you are encouraged to combine the square root code you wrote in lab 1 with your bin2dec code from lab 3.

The behavior of your CPU will be considered correct for a given instruction if any of the following conditions are met, with the exception of the CSRRW instruction which must work as described in *Apendix D*:

- Your CPU runs the instruction as described in *Appendix D* of this lab sheet.
- Your CPU runs the instruction identically to RARs.
- Your CPU runs the instruction identically to the linked RISC-V RV32I specification.

If the above three conditions are in conflict, prefer the lab sheet, but we will accept any. If you know there is a conflict, this is probably an error on our part, and you should let us know.

Rubric

- (A) 30 points – demonstration of sqrt program operating successfully.
- (B) 20 points – sqrt program deployed onto the DE2-115 board.
- (C) 30 points – demonstration of your testing suite.
- (D) 20 points – explanation of your testing methodology to the TA.

Maximum score: 100 points.

All grades will be assigned by demoing your code to the TAs. You may demo in any lab session, during office hours, or by appointment. Late penalty will be calculated from the time you turn your code in via Moodle, so it's OK if you don't get to demo before the lab is due.

When you demo your code to the TA, please be prepared to:

- Show your sqrt program running on the DE2-115 board.
- Show your testbench(es), test vectors, test programs, and any other scripts or materials you use to test your design.
- Briefly explain your testing strategy, and justify why you believe it is a complete and adequate evaluation of your design's functionality. Be prepared to answer questions.

Optional: for rubric category (D), if you feel uncomfortable explaining and justifying your test suite in an oral, in-person setting, you may choose to instead write a textual report, not to exceed 3 pages, to be submitted as a PDF via email to the TAs. You will still need to demonstrate the test suite and bin2dec running however.

Maximum score: 100 points.

Note that the following may cause you to lose points:

- Academic honesty violation, such as turning in another student's code.
- Code which does not compile. difficult to grade your project.

What to Turn In

When you are satisfied that your design works correctly, generate your submission file with `make pack`. Your submission file must be named `CSCE611_<Semester>_jb_<Your USC Username>.7z`. Your "USC username" is whatever you log into your university computer accounts with. For example, a student with the email address `jsmith@email.sc.edu` might turn in a file named `CSCE611_Fall2021_jb_jsmith.7z`. Please use the provided Make target, **do not pack your submission manually**.

Appendix A - RV32I I/J/B Instruction Reference

Note: In some cases, instructions are grouped with types other than their actual encoding for logical clarity. Such cases are specifically noted.

Note: All immediate values should be sign-extended. The sign bit of all immediate values is in the most-significant bit (bit 31) of the instruction.

I-Type

`jalr`

Note: this instruction deals with control-flow, but is encoded as an I-type.

`jalr rd, offset`: t=PC+4; PC += (R[rs1] + sext(offset))&~1; R[rd] = t

31-20	19-15	14-12	11-7	6-0
`imm[11:0]`	`rs1`	`000`	`rd`	`1100111`

B-Type

These instructions prevent the cpu from executing the next instruction in the program and instead cause it to begin executing a sequence of instructions in another memory location.

b

`b offset`: PC += sext(offset)

31	30-25	24-20	19-15	14-12	11-8	7	6-0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

beq

`beq rs1, rs2, offset`: if (R[rs1]==R[rs2]) PC += sext(offset)

31	30-25	24-20	19-15	14-12	11-8	7	6-0
imm[12]	imm[10:5]	rs2	rs1	000	imm[4:1]	imm[11]	1100011

bge

`bge rs1, rs2, offset`: if (R[rs1]>=R[rs2]) PC += sext(offset)

31	30-25	24-20	19-15	14-12	11-8	7	6-0
imm[12]	imm[10:5]	rs2	rs1	101	imm[4:1]	imm[11]	1100011

bgeu

`bgeu rs1, rs2, offset`: if (R[rs1]>=uR[rs2]) PC += sext(offset)

31	30-25	24-20	19-15	14-12	11-8	7	6-0
imm[12]	imm[10:5]	rs2	rs1	101	imm[4:1]	imm[11]	1100011

blt

`blt rs1, rs2, offset`: if (R[rs1]<R[rs2]) PC += sext(offset)

31	30-25	24-20	19-15	14-12	11-8	7	6-0
imm[12]	imm[10:5]	rs2	rs1	100	imm[4:1]	imm[11]	1100011

bltu

bltu rs1, rs2, offset: if (R[rs1]<uR[rs2]) PC += sext(offset)

31	30-25	24-20	19-15	14-12	11-8	7	6-0
imm[12]	imm[10:5]	rs2	rs1	110	imm[4:1]	imm[11]	1100011

J-Type

Note: This instruction is the only J-type we will implement this semster.

jal

jal rd, offset: R[rd] = PC+4; PC += sext(offset)

31	30-21	20	19-12	11-7	6-0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	1101111

Appendix B - Full Instruction Table

For your convenience, the entire table of RV32I and M instructions is given below. Note that includes some instructions we will not use in this course.

mnemonic	spec	funct7	funct3	opcode	encoding
LUI	RV32I			0110111	U
AUIPC	RV32I			0010111	U
JAL	RV32I			1101111	J

mnemonic	spec	funct7	funct3	opcode	encoding
JALR	RV32I		000	1100111	I
BEQ	RV32I		000	1100011	B
BNE	RV32I		001	1100011	B
BLT	RV32I		100	1100011	B
BGE	RV32I		101	1100011	B
BLTU	RV32I		110	1100011	B
BGEU	RV32I		111	1100011	B
LB	RV32I		000	0000011	I
LH	RV32I		001	0000011	I
LW	RV32I		010	0000011	I
LBU	RV32I		100	0000011	I
LHU	RV32I		101	0000011	I
SB	RV32I		000	0100011	S
SH	RV32I		001	0100011	S
SW	RV32I		010	0100011	S
ADDI	RV32I		000	0010011	I
SLTI	RV32I		010	0010011	I
SLTIU	RV32I		011	0010011	I
XORI	RV32I		100	0010011	I
ORI	RV32I		110	0010011	I
ANDI	RV32I		111	0010011	I
SLLI	RV32I	0000000	001	0010011	R
SRLI	RV32I	0000000	101	0010011	R
SRAI	RV32I	0100000	101	0010011	R
ADD	RV32I	0000000	000	0110011	R
SUB	RV32I	0100000	000	0110011	R
SLL	RV32I	0000000	001	0110011	R

mnemonic	spec	funct7	funct3	opcode	encoding
SLT	RV32I	0000000	010	0110011	R
SLTU	RV32I	0000000	011	0110011	R
XOR	RV32I	0000000	100	0110011	R
SRL	RV32I	0000000	101	0110011	R
SRA	RV32I	0100000	101	0110011	R
OR	RV32I	0000000	110	0110011	R
AND	RV32I	0000000	111	0110011	R
FENCE	RV32I		000	0001111	I
FENCE.I	RV32I		001	0001111	I
ECALL	RV32I		000	1110011	I
EBREAK	RV32I		000	1110011	I
CSRRW	RV32I		001	1110011	I
CSRRS	RV32I		010	1110011	I
CSRRC	RV32I		011	1110011	I
CSRRWI	RV32I		101	1110011	I
CSRRSI	RV32I		110	1110011	I
CSRRCI	RV32I		111	1110011	I
MUL	RV32M	0000001	000	0110011	R
MULH	RV32M	0000001	001	0110011	R
MULHSU	RV32M	0000001	010	0110011	R
MULHU	RV32M	0000001	011	0110011	R
DIV	RV32M	0000001	100	0110011	R
DIVU	RV32M	0000001	101	0110011	R
REM	RV32M	0000001	110	0110011	R
REMU	RV32M	0000001	111	0110011	R

Appendix C – ALU Operation Table

This table is also given in the lecture slides, but is reproduced here for convenience.

Note that the ALU B input is used as the shift amount (shamt) for shift instructions, since it would otherwise be unused.

4-bit opcode	function
0000	A and B
0001	A or B
0010	A xor B
0011	A + B
0100	A - B
0101	A * B (low, signed)
0110	A * B (high, signed)
0111	A * B (high, unsigned)
1000	A << shamt
1001	A >> shamt
1010	A >>> shamt
1100	A < B (signed)
1101	A < B (unsigned)
1110	A < B (unsigned)
1111	A < B (unsigned)

