Problem 1: RISC-V Practice

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.

When inputting RISC-V instructions into Gradescope, please follow the following guidelines:

- Use registers \( x0, x1, \ldots, x31 \) instead of \( ra, s1, t1, a0, \) and other special register names
- Include commas between registers and immediate values (\( \text{addi} x0, x0, 0 \))
- Use decimal for any immediate values
- For memory accesses, follow the format of \( \text{sw} \ rs2, \text{offset}(rs1) \) (e.g. \( \text{sw} x1, -4(x2) \))
- For hexadecimal values be sure to include the prefix \( 0x \)

For any questions that ask for **32-bit hexadecimal solutions**, please follow these formatting guidelines:

- For hexadecimal values be sure to include the prefix \( 0x \) (e.g. \( 0x\text{deadbeef} \))
- Be sure to include any leading 0s (e.g. \( 0x00000000 \))
- Each response should consist of \( 0x + 8 \) characters of hexadecimal
- Both upper and lowercase characters are acceptable

(a) RV32I instructions are stored in 32 bits in Instruction Memory. For the CPU Project, you will be implementing logic to decoding them. Let’s practice converting between instruction and their binary/hexadecimal encoding.

i. What is the instruction encoded by \( 0xFF4AAA23 \)?

ii. What is the instruction encoded by \( 0x403FD293 \)?
iii. What is the encoding (in 32-bit hexadecimal) of \texttt{bltu x2, x3, 102}?

iv. What is the encoding (in 32-bit hexadecimal) of \texttt{add x0, x5, x31}?

(b) Now, let’s investigate some interesting cases of the RISC-V ISA. (These may be helpful to consider later on when implementing the logic for your CPU)

i. Suppose you executed the instruction (iv) from part (a) \texttt{add x0, x5, x31}, what will the 32-bit hexadecimal value in \texttt{x0} be after its execution? Assume that \texttt{x5} and \texttt{x31} begin with the values \texttt{0x000000ee} and \texttt{0x00000151}, respectively.

ii. We want to take advantage of the upper immediate instructions provided by the ISA to store a 32-bit immediate into a register. A friend writes the following instruction sequence

\begin{verbatim}
addi x5, x0, 0xeef
lui x5, 0xdeadb
\end{verbatim}

What is the resulting value of the \texttt{x5} register in 32-bit hexadecimal?

iii. Another friend slightly modifies the instructions, so now we have:

\begin{verbatim}
lui x5, 0xdeadb
addi x5, x5, 0xeef
\end{verbatim}

What is the resulting value of the \texttt{x5} register in 32-bit hexadecimal?

iv. We decide to try out other other upper immediate instruction \texttt{auipc}. Our PC happens to be at \texttt{0x00010eed}. If the next instruction is \texttt{auipc x5, 0xdeadb}, what should we expect the contents of \texttt{x5} to be after this instruction is executed in 32-bit hexadecimal?

v. Let’s look at the branch instruction from part (a): \texttt{bltu x2, x3, 102}. If current PC is \texttt{0x100} and that value in \texttt{x2} < \texttt{x3}, what is the PC value after the instruction is executed in 32-bit hexadecimal?
vi. \texttt{j label; jr rs1; ret} are common pseudo instructions used in program control flow. What are the corresponding base instruction for them?
Problem 2: Datapath

(a) Given the single cycle RISC-V datapath shown below, fill in the control signals when executing the following instructions. Be sure to keep in mind which signals are ‘don’t care’.

The options for ImmSel are: I, S, B, U, J for the different instruction types.

The options for ALUSel are: ADD, AND, OR, XOR, SLL, SRL, SRA, SLT, SLTU.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>PCSel</th>
<th>ImmSel</th>
<th>RegWEn</th>
<th>Asel</th>
<th>BSel</th>
<th>ALUSel</th>
<th>MemRW</th>
<th>WBSelect</th>
</tr>
</thead>
<tbody>
<tr>
<td>and rd, rs1, rs2</td>
<td>0</td>
<td>*</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>AND</td>
<td>READ</td>
<td>1</td>
</tr>
<tr>
<td>slli rd rs1 imm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq rs1 rs2 label (not taken)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jalr rd rs1 imm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lhu rd imm(rs1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sb rs2 imm(rs1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>auipc rd imm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
If we want to add some new instructions, what modifications would be required? Select all that apply.

The only allowed Datapath modifications are adding wires connecting existing units, and adding muxes and adders.

(b) Description: \( R[rd] = \neg (R[rs1] \& R[rs2]) \)

\texttt{nand rd rs1 rs2}

- Control Logic
- Datapath (adding wires, muxes, adders)
- ALU
- Immediate Generator

(c) Description: \( M[rs1 + \text{imm2}] = \text{imm1} \)

\texttt{sw\_immediate rs1 imm1 imm2}

- Control Logic
- Datapath (adding wires, muxes, adders)
- ALU
- Immediate Generator
Problem 3: Pipelining and Hazard Practice

Suppose we have a small RISC-V program that reads:

```
1    addi t1, x0, 0x151
2    bnez t1, normal
3 fault:
4    addi t1, x0, 0x251
5 normal:
6    lw  t2, 0(t1)
7    xori t4, t2, 1
8    lw  t3, 0(t2)
9    slli t3, t3, 1
10   xor  t5, t3, t4
```

(a) If we ran this program on the single cycle datapath provided in the previous problem, how many stalls would need to be introduced for the code to execute correctly? What would the cycles per instruction (CPI) of this program be?

(b) Now, assume we have implemented the standard 5-stage RISC-V pipeline provided in lecture and also shown above without any forwarding logic added. How many hazards of each
type (structural, data, control) are introduced if we executed the above program on the pipelined datapath?

Assume an asynchronous-read, synchronous-write register file (capable of reading from 2 registers and writing to 1 register in the same cycle) and an asynchronous-read, synchronous write DMEM/IMEM. Assume IMEM and DMEM are also separated.

(c) Let’s say we implemented the **ALU-to-ALU** and **MEM-to-ALU** data forwarding as shown above. Would this eliminate all stalls needed to address the data hazards present in this program? If not, specify the line number(s) after which some stalls would need still to be inserted.

(d) Assume that our branch comparator is in the execute stage as shown. Which stages of the pipeline need to get flushed on a branch mispredict?
(e) (251A Only) Now let’s evaluate the performance of the 5-stage pipelined design in comparison to the single stage datapath. Assume that branches are always predicted not-taken and mispredictions are handled by pipeline flushes. Assume all memory accesses are valid.

Consider that we have implemented both ALU-to-ALU and MEM-to-ALU forwarding, for both ALU and branch comparator, for both rs1 and rs2. Use the table on the next page to help you determine how many cycles this program takes to run. Count all cycles from Fetch stage of the first instruction to Writeback stage of the last instruction. For the instruction column, you only need to write the operator. You may not use all the rows and columns.

(f) (251A Only) Show how you can reorder instructions after the normal label in the program in a way that minimizes the data hazard stalls, without changing the final values of the registers and data memory. How many cycles does this new reordering take?