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, ..., x31 instead of ra, s1, t1, a0, and other special register names
- Include commas between registers and immediate values (addi x0, x0, 0)
- Use decimal for any immediate values
- For memory accesses, follow the format of sw rs2, offset(rs1) (e.g. 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. 0xdeadbeef)
- 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?
   \( \text{sw x20, -12(x21)} \)

ii. What is the instruction encoded by 0x403FD293?
   \( \text{srai x5, x31, 3} \)

iii. What is the encoding (in 32-bit hexadecimal) of bltu x2, x3, 102?
   \( 0x06316363 \)
iv. What is the encoding (in 32-bit hexadecimal) of `add x0, x5, x31`?

0x01f28033

(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) `add x0, x5, x31`, what will the 32-bit hexadecimal value in `x0` be after its execution? Assume that `x5` and `x31` begin with the values 0x000000ee and 0x000000151, respectively.

0x00000000. `x0` always stores the value 0 and cannot be overwritten. It is still a valid instruction, but will not make any changes to the state of the registers.

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

```
addi x5, x0, 0xeef
lui x5, 0xdeadb
```

What is the resulting value of the `x5` register in 32-bit hexadecimal?

0xdeadb000. `lui` fills the bottom 12 bits of the immediate with 0s so the `lui` instruction will zero out the bottom 12 bits from the previous `addi`

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

```
lui x5, 0xdeadb
addi x5, x5, 0xeef
```

What is the resulting value of the `x5` register in 32-bit hexadecimal?

0xdeadaeef. We first load the upper immediate bits before attempting to add the lower 12 bits so we don’t run into the same issue as before. However, remember that by default the immediates are treated as signed values, so the `0xeef` gets sign extended, leading to our final value to 0xdeadaeef rather than 0xdeadbeef

iv. We decide to try out other other upper immediate instruction `auipc`. Our PC happens to be at 0x000010eed. If the next instruction is `auipc x5, 0xdeadb`, what should we expect the contents of `x5` to be after this instruction is executed in 32-bit hexadecimal?

0x0deaeb0ed. `auipc` will add the upper immediate bits to the current PC.

v. Let’s look at the branch instruction from part (a): `bltu x2, x3, 102`. If current PC is 0x100 and that value in `x2` < `x3`, what is the PC value after the instruction is executed in 32-bit hexadecimal?

0x00000166. The first thing to realize that the branch offset was provided in decimal so converting to hexadecimal gets you a relative offset of 0x66. Since the branch condition is true, we jump to the current PC + the relative address offset which is 0x00000166.

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

- `j label` -> `jal x0, label`
- `jr rs` -> `jalr x0, rs, 0`
- `ret` -> `jalr x0, x1, 0`
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.

![Datapath Diagram]

<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>WBSel</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: \( R[rd] = \sim (R[rs1] \& R[rs2]) \)

\textbf{nand rd rs1 rs2}

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

\textbf{ALU, Control Logic. We want to have the ALU deal with another type of instruction.}

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

\textbf{sw_immediate rs1 imm1 imm2}

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

\textbf{Control Logic, Datapath, Immediate Generator. Edit the immediate generator to output 2 immediates and add a new mux right before the input to DMEM.}
Problem 3: Pipelining and Hazard Practice

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

```assembly
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?

0 Stalls. CPI of 1. Single cycle datapaths do not have any hazards because each instruction will complete before the next instruction is fetched.

(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.
Structural Hazards: 0. The regfile is specified to allow for 2 reads and a write at a time and the IMEM and DMEM can be accessed simultaneously which removes any structural hazards introduced by the 5-stage pipeline.

Data Hazards: 5. The hazards introduced are:

- addi t1, x0, 0x151 → bnez t1, normal
- lw t2, 0(t1) → xori t4, t2, 1
- lw t2, 0(t1) → lw t3, 0(t2)
- lw t3, 0(t2) → slli t3, t3, 1
- slli t3, t3, 1 → xor t5, t3, t4

Depending on whether you take the branch or not, you may or may not encounter the dependency between addi t1, x0, 0x251 → lw t2, 0(t1). However that branch will never be taken since we have hardcoded the immediate values for the branch instruction.

Control Hazards: 1. The branch instruction in line 2 will introduce a control hazard because we cannot determine which instructions to execute afterwards until the branch is resolved at the end of the X stage.

Figure 2: Datapath with ALU-to-ALU and MEM-to-ALU forwarding logic. (borrowed from EECS151 Fall 2022)

(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.

No, lines 6 and 8 still require 1 stall. Although we have MEM-to-ALU forwarding, this does not allow us to directly forward the output of the DMEM directly back into the ALU on the same cycle.
You could argue that this is possible, but this path is not shown in the provided datapath and would most likely greatly increase the critical path and clock period.

(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?

F/D/X stages need to get flushed, because the new branch PC only gets forwarded back to the PC when when the branch is in the M stage.

(e) (251 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. What is the resulting CPI? 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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

16 cycles.

The catch is the load-to-use sequences (lw-addi and lw-slli), where even with MEM-to-ALU forwarding we cannot avoid one stall cycle.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bnez</td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xori</td>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xori</td>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>slli</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(f) (251 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?

We start from the fact that we cannot avoid one stall cycle for back-to-back load-to-use chains. So the trick is to keep the load-to-use distance longer by “prefetching” t2:

\[
\begin{align*}
\text{lw} & \quad t2, 0(t1) \\
\text{lw} & \quad t3, 0(t2) \\
\text{xori} & \quad t4, t2, 1 \\
\text{slli} & \quad t3, t3, 1 \\
\end{align*}
\]

Then we can eliminate the two load stalls. However, we introduce one new stall between \text{lw}s because there is still an immediate load-to-use dependency on \text{t2}. The total number of cycles becomes 15.