List of Problems

1 Microcoded TinyRV1 Processor 2
   1.A Implementing Conditional Move Instruction 3
   1.B Implementing Memory-Memory Swap Instruction 3
   1.C Implementing Memory-Memory Increment Instruction 4
   1.D Implementing String Length Instruction 4
Problem 1. Microcoded TinyRV1 Processor

Consider the TinyRV1 FSM processor with a microcoded control unit described in lecture. In this problem, we explore adding two new instructions by using the same datapath and just adding new microcode sequences to the control store.

Your solutions should be elegant and efficient; minimize the number of new states needed. Figure 1 shows the encoding you should use for all of the control signal fields that are not just 0 or 1. Note that we have added one new operations to our ALU compared to lecture: increment the A input by one. You cannot add new datapath components or modify the datapath components beyond this one change. If you use any new pseudo-control-signal syntax please clearly explain what this syntax means. When filling in microcode, use don’t cares (marked with an x or –) for fields where it is safe to use don’t cares. Study the processor described in lecture well, and make sure all of your microinstructions are legal. Please comment your code clearly. Your code should exhibit “clean” behavior and not modify any architectural registers in the course of the execution. Finally, make sure that each new macroinstruction fetches the next macroinstruction with a microjump to F0.

---

<table>
<thead>
<tr>
<th>B/C Mux Select Encoding</th>
<th>b</th>
<th>select mux input from bus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s</td>
<td>select mux input from shifter</td>
</tr>
</tbody>
</table>

| Immediate Generation Unit (IG) | i | sext( IR[31:20] ) |
| Types                        | s | sext( IR[31:25], IR[11:7] ) |
|                             | b | sext( IR[31], IR[7], IR[30:25], IR[11:8], 0 ) |
|                             | j | sext( IR[31], IR[19:12], IR[20], IR[30:21], 0 ) |

| Arithmetic/Logic Unit (ALU) Functions | +4 | A + 4 |
|                                      | +  | A + B |
|                                      | +? | C[0] ? A + B : copy A |
|                                      | cmp | A == B |
|                                      | -4 | A - 4 |
|                                      | +1 | A + 1 |

<table>
<thead>
<tr>
<th>Register File Select Encoding</th>
<th>x0</th>
<th>select x0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rs1</td>
<td>select register based on rs1 field in IR</td>
</tr>
<tr>
<td></td>
<td>rs2</td>
<td>select register based on rs2 field in IR</td>
</tr>
<tr>
<td></td>
<td>rd</td>
<td>select register based on rd field in IR</td>
</tr>
</tbody>
</table>

| Memory Request Op Encoding | r | read memory request |
|                          | w | write memory request |

| Next State Encoding | n | goto next state by incrementing µPC by one |
|                    | d | dispatch to instruction sequence based on opcode |
|                    | f | goto state F0 |
|                    | b | goto state F0 if A == B |

---

**Figure 1: Control Signal Encodings**
Part 1.A Implementing Conditional Move Instruction

For this part, you are to add a new conditional move (movz) instruction. This instruction only copies the source register to the destination register if a second source register is zero. The assembly format and semantics for the new RISC-V instruction are as follows:

\[
\text{movz } rd, rs1, rs2 \\
\text{if ( } R[rs2] == 0 \text{ ) } R[rd] \leftarrow R[rs1]
\]

To reiterate, the instruction only copies the source register if the second register is equal to zero! This is a much easier problem if the instruction copies the source register when the second register is not equal to zero. Create a table like the one shown in Figure 2 to represent the contents of the control store, and fill in the state, pseudo-control-signal syntax, actual control signals, and next state fields for a microinstruction fragment that implements the movz instruction. The fetch fragment has already been provided for you.

Part 1.B Implementing Memory-Memory Swap Instruction

For this part, you are to add a new memory-memory swap (swap.mm) instruction. The assembly format and semantics for the new RISC-V instruction are as follows:

\[
\text{swap.mm } rd, rs1 \\
\text{ temp } \leftarrow M[R[rs1]]; \ M[R[rs1]] \leftarrow M[R[rd]]; \ M[R[rd]] \leftarrow temp
\]

Note that this instruction should not modify any architectural state other than what is indicated by the semantics (i.e., you must carefully select what you use for temporary state to avoid corrupting architectural state). Create a table like the one shown in Figure 2 to represent the contents of the control store, and fill in the state and pseudo-control-signal syntax for a microinstruction fragment that implements the swap.mm instruction. You do not need to fill in the actual control signals or the next state fields! The fetch fragment has already been provided for you. Minimize the number of microinstructions in your sequence. Note that a naive implementation will always do the swap even when the values being swapped are identical. Optimize your microinstruction sequence to reduce the execution time when the values being swapped identical, but avoid increasing the execution time in the common case when the values being swapped are not identical.
Part 1.C Implementing Memory-Memory Increment Instruction

For this part, you are to implement a new instruction that is similar in spirit to the x86 memory-memory inc instruction discussed in a practice problem for the previous topic. This instruction will use a relatively complicated addressing mode to read a value from memory, increment that value by one, and then write the value back to the same location in memory. The assembly format and semantics for the new RISC-V instruction are as follows:

\[
\text{inc rd, rs1, imm} \quad \text{addr} \leftarrow R[\text{rd}] + (R[\text{rs1}] \times \text{imm}[3:0]); \quad M[\text{addr}] \leftarrow M[\text{addr}] + 1
\]

Note that \( \text{addr} \) is simply a temporary to simplify the instruction semantics. It is not architectural state. Note that this instruction only uses the least significant four bits of the immediate when calculating the effective address. Create a table like the one in Figure 2 to represent the contents of the control store, and fill in the state, pseudo-control-signal syntax, actual control signals, and next state fields for a microinstruction fragment that implements the inc instruction. The fetch fragment has already been provided for you.

Part 1.D Implementing String Length Instruction

For this part, you are to implement a new string length (strlen) instruction. The assembly format and semantics for the new RISC-V instruction are as follows:

\[
\text{strlen rd, rs1} \quad R[\text{rd}] \leftarrow 0; \quad \text{while} \ (M[R[\text{rs1}]] \neq 0) \ \{ \ R[\text{rs1}] \leftarrow R[\text{rs1}] + 1; \ R[\text{rd}] \leftarrow R[\text{rd}] + 1 \}\n\]

In other words, the strlen instruction should count the number of characters in a string pointed to by rs1 and return the final count in the rd register. After the instruction has finished, the rs1 register will contain a pointer to the null character at the end of the string. Note that we will need to assume we can do byte reads from memory. Create a table like the one shown in Figure 2 to represent the contents of the control store, and fill in the state and pseudo-control-signal syntax for a microinstruction fragment that implements the strlen instruction. You do not need to fill in the actual control signals or the next state fields! The fetch fragment has already been provided for you.