This is a full length practice midterm exam. If you want to take it at exam pace, give yourself 75 minutes to take the entire test. Just like the real exam, each question has a point value. There are 75 points in the exam, so that you can pace yourself to average 1 point per minute (some parts will be faster, some slower).

Questions:

1. Combinatorial Logic: 15 pts
2. Sequential Logic: 10 pts
3. Finite State Machines: 10 pts
4. MIPS Assembly: 20 pts
5. Datapaths: 10 pts
6. Memory Hierarchy: 10 pts

This is the solution set to the practice exam. The solutions appear in blue boxes.
Question 1: Combinatorial Logic [15 pts]

Given the following truth-table:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

1. Write the sum-of-product formula

   **Answer:**
   
   (NOT a AND b AND NOT c) or (a AND NOT b AND NOT c) or (a AND b AND NOT c)

2. Simplify the formula

   **Answer:**
   
   (a OR b) AND (NOT c)

3. Write VHDL that implements the formula

   ```vhdl
   entity q1 is
   port (a : in std_logic;
          b : in std_logic;
          c : in std_logic;
          x : out std_logic);
   end q1;
   architecture basic of q1 is
   begin
   x <= (a or b) and (not c);
   end basic;
   ```
Question 2: Sequential Logic [10 pts]
Consider the following VHDL fragment, in which \( x \), and \( y \) are inputs, \( z \) is an output, and there are two DFFs (\( a \) and \( b \)) whose \( d \) inputs are \( a_d \) and \( b_d \) respectively, and whose \( q \) outputs are \( a_q \) and \( b_q \) respectively:

\[
\begin{align*}
  b_d &\leftarrow a_q \ XOR x; \\
  a_d &\leftarrow (b_q \ AND a_q) \ XOR y; \\
  z &\leftarrow a_q \ XOR b_q;
\end{align*}
\]

Complete the waveform below (assume that the DFFs are triggered by the rising edge of \( \text{clk} \)): Note that \( a_q \) is initially 1, and \( b_q \) is initially 0:
Question 3: FSMs [10 pts]

Draw a state machine diagram for a finite state machine which accepts a single bit input (either 0 or 1—you can just label each edge with 0 or 1). This state machine also has a single bit of output, which is initially 0.

- Whenever the FSM receives an input bit of 1 followed by anything other than two 0s (so 11, or 101) the output goes to 1.
- The output remains at 1 until 3 consecutive 0s are received by the FSM.
- Once three consecutive 0s are received by the FSM, the output returns to 0.
- The output now remains 0 until it again sees a 1 which is not followed (immediately) by two 0s.

Note that when the FSM sees a 1 not followed by two 0s, the output goes to 1 as soon as the situation is detected (e.g., on the second 1 of 11 or 101). Label each state with the bit it outputs. Be sure to indicate your start state (with an arrow to it from nowhere).
Question 4: Asm Programming [20 pts]
Write MIPS assembly for the following C function.

```c
int countMatching(int * ptr, int n) {
    int count = 0;
    while (n > 0) {
        int x = testFn(*ptr);
        if (x != 0) {
            count++;
        }
        ptr++;  
        n--;
    }
    return count;
}
```

Answer on next page
.globl countMatching
.ent countMatching
.text
countMatching:

    addiu $sp, $sp, -32  # make space for stack frame
    sw $fp, 0($sp)    # save regs to stack
    sw $ra, 4($sp)
    sw $s0, 8($sp)
    sw $s1, 12($sp)
    sw $s2, 16($sp)
    addiu $fp, $sp, 28  # setup new fp
    move $s0, $a0  # move ptr into s0
    move $s1, $a1  # move n into s1
    li $s2, 0  # count in s2
  .L_cm_while:  # while (n > 0) {
    blez $s1, .L_cm_end_while
    lw $a0, 0($s0)  # put *ptr in $a0
    jal testFn    # call testFn
    # x returned in v0
    beqz $v0, .L_cm endif  # if (x != 0) {
    addi $s2, $s2, 1  # count++
  .L_cm endif:  # } of if
    addiu $s0, $s0, 4  # ptr++ (add 4 b/c int *)
    addi $s1, $s1, -1  # n--
    b .L_cm_while  # } of while loop
.L_cm endwhile:
    move $v0, $s2  # put return result in v0
    lw $s2, 16($sp)  # restore regs from stack
    lw $s1, 12($sp)
    lw $s0, 8($sp)
    lw $ra, 4($sp)
    lw $fp, 0($sp)
    addiu $sp, $sp, 32  # put sp back (release space)
    jr $ra  # return to caller

.end countMatching
Question 5: Datapaths [10 pts]
Consider the following single-cycle data-path:

Fill in the following table of control signals to correctly control the datapath for each of these instructions. For any control signal that does not matter, write X for “don’t care.” RW is the register file write-enable. BS and V are mux selectors (0 = top input, 1 = bottom input), DW is the data-memory write-enable. Br is the second input of the AND-gate pictured.

<table>
<thead>
<tr>
<th>Insn</th>
<th>RW</th>
<th>BS</th>
<th>DW</th>
<th>V</th>
<th>Br</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>
1. SRAM access latency is roughly proportional to \( \#\text{bits} \times \#\text{ports}^2 \). Explain why.

**Answer:**
The primary factor in this is the capacitance of the bit lines and word lines. The capacitance of these wires is proportional to their length. Increasing the number of bits in the RAM lengthens these wires in one direction—e.g., adding more entries lengthens the bit lines, while adding more bits per word lengthens the word lines. Adding 1 port, however, causes longer bit lines and word lines. The reason for this growth in both directions is that adding the port requires adding a set of bit-lines and a set of word-lines. Adding the extra bit-lines makes the RAM wider, lengthening the word-lines, and vice-versa for the word-lines/bit-lines.

2. Explain the motivation for a memory hierarchy (as opposed to just one single memory element).

**Answer:**
The motivation for a memory hierarchy comes from the tension between fast access time (low \( t_{hit} \)) and large capacity (low \( \%\text{miss} \))—a large RAM can hold a lot of data, but is slow to access; a small RAM is fast, but cannot hold much data. To resolve this tension, we build a hierarchy of memory elements—those memory elements closest to the processor are small and fast. A miss in a small/fast element of the memory-hierarchy is serviced by the next level which is larger and slower. The net effect is a system in which the average memory access time is typically quite fast.