ECE 411, Exam 2

- This exam has 5 problems. Make sure you have a complete exam before you begin.
- Write your name on every page in case pages become separated during grading.
- You will have three hours to complete this exam.
- Write all of your answers on the exam itself. If you need more space to answer a given problem, continue on the back of the page, but clearly indicate that you have done so.
- This exam is closed-book. You may also use a calculator.
- DO NOT do anything that might be perceived as cheating. The minimum penalty will be a grade of zero.
- Show all of your work on all problems. Correct answers that do not include work demonstrating how they were generated may not receive full credit, and answers that show no work cannot receive partial credit. If a dinosaur is drawn as an answer to a non-bonus question, it will not receive credit. But we will think kindly of those that make us smile.

- The exam is meant to test your understanding. Ample time has been provided. So, be patient. Read the questions/problems carefully before you answer.

- **Good luck!**

Problem 1 (16 pts): ___________
Problem 2 (25 pts): ___________
Problem 3 (8 pts): ___________
Problem 4 (18 pts): ___________
Problem 5 (17 pts): ___________

Total: ___________
1. Pipeline Design (16 points)

The basic 5-stage pipeline we talked about in class has stages in this order:
IF ID EX MEM WB. We will call this basic machine FOO

Suppose now we change the architecture so that the 5 stages take the order:
IF ID MEM EX WB. We will call the modified machine BAR

Machine BAR is a register-memory machine. This means that memory can be an operand for an ALU instruction, but no offset can be used for these instructions. For example, ADD R0, R1, (R2) is valid. This means add the contents of R1 to the content in the memory at the address pointed to by R2. You then store the result in R0. In RTL, this would be R0 ← R1 + M[R2].

a) The forwarding paths for machine FOO were discussed in class. On machine BAR, state the forwarding paths needed and the instruction sequences that exploit each path (indicate any stalling needed). (4 points)

ADD R1, R0, (R5) if R1 is used in X: no problem
ADD X, Y, (Z) if R1 is used in Y: EX-->EX fwd
if R1 is used in Z: EX-->MEM fwd, 1 cycle stall

LDR R0, R1, 0 MEM-->MEM fwd
LDR R2, R0, 0

b) Give a sequence of instructions that will execute faster on machine FOO. Give a second sequence of instructions that will execute faster on machine BAR. (6 points)

ADD R2, R1, R3 no stall on machine FOO, 1 cycle stall on machine BAR
LDR R5, (R2) no stall on machine FOO, 1 cycle stall on machine BAR
LDR R1, (R0)
ADD R2, R1, R3 no stall on machine BAR, 1 cycle stall on machine FOO
c) Now you realize that there are basic differences on the two machines. Give an example of a short function for which machine BAR needs more instructions than machine FOO to accomplish the same functionality. Then give a different example where machine FOO needs more instructions than machine BAR does to accomplish the same task. (Hint: remember the restrictions of machine BAR). (6 points)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Machine FOO</th>
<th>Machine BAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R1 \leftarrow M[R0+4] )</td>
<td><code>LDR R1, R0, #4</code></td>
<td><code>ADD R0, R0, #4</code> <code>LDR R1, (R0)</code></td>
</tr>
<tr>
<td>( R1 \leftarrow R0 + M[R2] )</td>
<td><code>LDR R2, (R2)</code></td>
<td><code>ADD R1, R0, R2</code></td>
</tr>
</tbody>
</table>
2. Data Hazards (25 points)

Consider the following assembly code:

```
ADD R1, R1, R2
ADD R3, R1, 3
LDR R4, R3, 0
SUB R2, R4, R5
AND R5, R4, R1
STR R5, R1, 5
LDR R3, R1, 0
AND R4, R5, R3
ADD R2, R3, 4
ADD R3, R3, 3
STR R2, R1, 8
ADD R6, R0, 0
ADD R6, R6, 1
ADD R7, R0, 3
```

a) Assume you have the 5 stage pipelined processor discussed in class and no data forwarding or hazard detection. Insert NOPs into the above assembly code to ensure that the program runs as intended. Assume a perfect cache. Also, assume that a value written into the register file becomes valid on the next clock cycle. (4 points)

Solution:

```
ADD R1, R1, R2
NOP
NOP
NOP
ADD R3, R1, 3
NOP
NOP
NOP
LDR R4, R3, 0
NOP
NOP
NOP
SUB R2, R4, R5
AND R5, R4, R1
NOP
NOP
NOP
STR R5, R1, 5
LDR R3, R1, 0
NOP
NOP
NOP
AND R4, R5, R3
ADD R2, R3, 4
```
b) What is the speedup of running this program over an un-pipelined processor that takes 5 cycles to complete each instruction (assuming the same clock speed)? (Remember, the pipelined processor still takes 5 cycles per instruction, but because multiple instructions can be performed at the same time, one instruction can complete per cycle.) (3 points)

Solution:  
14 inst * 5 cyc/inst = 70 cycles for unpipelined. 
34 + 4 cycles for part a) 
so Speedup = 70 / 38 = 1.84 x

c) Suppose your compiler can reorder the instructions to eliminate stalls. Write the newly reordered code. Insert NOP’s wherever needed and do not rename registers. Calculate the speedup over your answer for part a). (7 points)

Solution:  
ADD R1, R1, R2 
ADD R6, R0, 0 
ADD R7, R0, 3 
NOP 
ADD R3, R1, 3 
ADD R6, R6, 1 
NOP 
NOP 
LDR R4, R3, 0 
LDR R3, R1, 0 
NOP 
NOP 
SUB R2, R4, R5 
AND R5, R4, R1 
AND R4, R5, R3 
ADD R2, R3, 4
d) Consider another case where data forwarding has been added to the datapath but there is no instruction reordering. Rewrite the original code with any necessary NOPs and calculate the speedup over part a). (7 points)

Solution:

```
ADD R1, R1, R2
ADD R3, R1, 3
LDR R4, R3, 0
NOP
SUB R2, R4, R5
AND R5, R4, R1
STR R5, R1, 5
LDR R3, R1, 0
NOP
AND R4, R5, R3
ADD R2, R3, 4
ADD R3, R3, 3
STR R2, R1, 8
ADD R6, R0, 0
ADD R6, R6, 1
ADD R7, R0, 3
```

16 + 4 cycles, so Speedup = 70 / 20 = 3.5 x

e) Suppose that you want to run the code on an out-of-order design that includes 16 physical registers. Perform register renaming and instruction reordering on the original code to eliminate WAW and WAR hazards. You can use registers R0-R15. (4 points)

```
ADD R8, R1, R2
ADD R3, R8, 3
LDR R4, R3, 0
SUB R9, R4, R5
AND R10, R4, R8
STR R11, R8, 5
LDR R12, R8, 0
AND R13, R10, R12
ADD R14, R12, 4
ADD R15, R12, 3
STR R2, R8, 8
ADD R6, R0, 0
ADD R1, R6, 1
ADD R7, R0, 3
```
3. Tomasulo (8 points)

LDB R3, R0, Tatooine
ADD R1, R2, R3
LDR R4, R1, 4
AND R2, R4, R3
ADD R2, R2, 2
NOT R5, R3
ADD R6, R5, 1
LEA R7, Vulcan
STI R6, R7, 0

Suppose you run this code on a Tomasulo machine with the following characteristics:
- The machine has 2 ALUs and 1 load-store unit.
- There are 8 registers.
- All instructions are already in the instruction queue.
- An ALU instruction takes 2 cycles to complete.
- A load or store instruction takes 3 cycles to complete.
- A dependent instruction issues 1 cycle after its preceding instruction completes.

RAT

<table>
<thead>
<tr>
<th>Physical Reg</th>
<th>Arch. Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R0</td>
</tr>
<tr>
<td>1</td>
<td>R2</td>
</tr>
<tr>
<td>2</td>
<td>R3</td>
</tr>
<tr>
<td>3</td>
<td>R7</td>
</tr>
<tr>
<td>4</td>
<td>R1</td>
</tr>
<tr>
<td>5</td>
<td>R5</td>
</tr>
<tr>
<td>6</td>
<td>R4</td>
</tr>
<tr>
<td>7</td>
<td>R6</td>
</tr>
<tr>
<td>8</td>
<td>R2</td>
</tr>
</tbody>
</table>
Instruction Queue

<table>
<thead>
<tr>
<th>Opcode</th>
<th>DR</th>
<th>SR1</th>
<th>SR2</th>
<th>FU</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDB</td>
<td>R3</td>
<td>R0</td>
<td>-</td>
<td>1: LS *</td>
</tr>
<tr>
<td>ADD</td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>4: ALU1 *</td>
</tr>
<tr>
<td>LDR</td>
<td>R4</td>
<td>R1</td>
<td>-</td>
<td>6: ALU1 *</td>
</tr>
<tr>
<td>AND</td>
<td>R2</td>
<td>R4</td>
<td>R3</td>
<td>8: ALU1 *</td>
</tr>
<tr>
<td>STR</td>
<td>R2</td>
<td>R0</td>
<td>-</td>
<td>10: LS *</td>
</tr>
<tr>
<td>ADD</td>
<td>R2</td>
<td>R2</td>
<td>-</td>
<td>10: ALU1 *</td>
</tr>
<tr>
<td>NOT</td>
<td>R5</td>
<td>R3</td>
<td>-</td>
<td>4: ALU2 *</td>
</tr>
<tr>
<td>ADD</td>
<td>R6</td>
<td>R5</td>
<td>-</td>
<td>6: ALU2 *</td>
</tr>
<tr>
<td>LEA</td>
<td>R7</td>
<td>-</td>
<td>-</td>
<td>8: ALU2 *</td>
</tr>
<tr>
<td>STI</td>
<td>-</td>
<td>R6</td>
<td>R7</td>
<td>10: LS *</td>
</tr>
</tbody>
</table>

How many cycles does it take for all instructions to complete? Show your work in the above tables (8 points)

13 cycles
4. Control Hazards (18 points)

In this question, you will examine a piece of assembly code for hazards and identify its performance given various conditions. Given the following C code:

```c
void translate(int8 *buf, int16 len, int8 translate_table[256])
{
    int8 temp;
    do
    {
        temp = translate_table[*buf];
        *buf = temp;
        buf++;
        len--;
    } while (len);
}
```

which translates to the following piece of LC3B assembly:

```
Loop:
I1:  LDB R6, R1, 0
I2:  ADD R4, R6, R3
I3:  LDB R5, R4, 0
I4:  STB R5, R1, 0
I5:  ADD R1, R1, 1
I6:  ADD R2, R2, -1
I7:  BRp Loop
I8:  RET
```

Hint: `translate_table` will be treated as a pointer; the compiler will put `buf` in R1, `len` in R2, `translate_table` in R3, and use JSR to call the routine so that the return address is in R7.

Answer the following questions given the assumptions below:

- Assume a standard five stage pipeline (IF, ID, EX, MEM, WB).
- All arithmetic instructions take 1 clock cycle.
- Assume full data forwarding.
- Memory instructions take 1 clock cycle (i.e. 100% Cache Hits).
- You may ignore pipeline startup cycles in your cycle calculations.
- The loop is called frequently, meaning all branch prediction hardware has stabilized.
- Assume branches are resolved in the MEM stage (3-stage penalty for misprediction).
- Assume branch destinations are calculated in the ID stage.
- Assume a 1-bit predictor is initialized to not taken.
- Assume a 2-bit predictor is initialized to weakly taken.
- If an instruction in WB writes to a register, a dependent instruction in ID can read that value during the same cycle.
a) With a static predict taken scheme:
   i.) How many cycles (in terms of len) does this code take to run without a BTB? (3 points)
       \[9*\text{len} + 4\]
   ii.) How many cycles (in terms of len) does this code take to run with a BTB? (3 points)
       \[8*\text{len} + 4\]

b) With a 1-bit prediction scheme:
   i.) How many cycles (in terms of len) does this code take to run without a BTB? (3 points)
       \[9*\text{len} + 7\]
   ii.) How many cycles (in terms of len) does this code take to run with a BTB? (3 points)
       \[8*\text{len} + 7\]

c) With a 2-bit prediction scheme:
   i.) How many cycles (in terms of len) does this code take to run without a BTB? (3 points)
       \[9*\text{len} + 4\]
   ii.) How many cycles (in terms of len) does this code take to run with a BTB? (3 points)
       \[8*\text{len} + 4\]
5. Potpourri (17 points)
a) Ben Bitdiddle wants to accelerate matrix multiplication using vector instructions that allow him to perform 4 multiplications in parallel. However, since the inner loop currently operates on only one element at a time, he realizes that he will need to unroll it 4 times. Write the following code in C after the inner loop has been unrolled. Note that the matrices are square but they may have an arbitrary size. Your code will need to have the same functionality as the original in order to receive full credit. (4 points)

```c
for(i = 0; i < MATRIX_WIDTH; i++)
{
    for(j = 0; j < MATRIX_WIDTH; j++)
    {
        for(k = 0; k < MATRIX_WIDTH; k++)
        {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
rem = MATRIX_WIDTH % 4;
for(i = 0; i < MATRIX_WIDTH; i++)
{
    for(j = 0; j < MATRIX_WIDTH; j++)
    {
        for(k = 0; k < MATRIX_WIDTH; k+=4)
        {
            C[i][j] += A[i][k] * B[k][j];
            C[i][j] += A[i][k+1] * B[k+1][j];
            C[i][j] += A[i][k+2] * B[k+2][j];
            C[i][j] += A[i][k+3] * B[k+3][j];
        }
        if(rem)
        {
            for(k = MATRIX_WIDTH - rem; k < MATRIX_WIDTH; k++)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```
b) Will the optimal number of pipeline stages increase or decrease if the metric of evaluation is not performance, but energy? Why? (4 points)

b) Decreases. The performance overheads of decreasing the number of stages is, in general, more than outweighed by the power benefits of fewer stages.

c) What is the number of read ports and write ports required for a register file to be used with a 4-issue OOO processor? (4 points)

8 read ports, 4 write ports

d) List all comparisons performed in the dependence check logic in the register renaming stage of an OOO pipeline. (4 points)

sources registers for the instructions in the current rename group against the destination registers of the older instructions in the current rename group.

e) While working on your MP3 design, you inadvertently create an evil artificial intelligence (like SkyNet, or GLaDOS, or Apple) that takes control of the world’s electronics in a matter of seconds (congrats on that, I guess). The AI then goes on a destructive rampage against humanity, as evil AIs are wont to do.

Choose one option. Circle your choice and write your answer below:

i.) You join your fellow ECE 411 students in a rag-tag resistance to free the world from the AI’s clutches. Give your resistance movement a name and a flag.

ii.) Alternatively, you convince your AI to trust you and use it to conquer the world. Give your evil overlord a name and a flag.

iii.) You wake up and realize it was all just a VHDL-induced nightmare. Draw a flag for your ECE 411 MP3 group.
**LDR**

**Load Word**

**Assembler Format**

\[
\text{LDR} \quad DR, \text{BaseR}, \text{offset6}
\]

**Encoding**

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>offset6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0110</td>
<td>DR</td>
<td>BaseR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

\[
\text{DR} = \text{memWord[BaseR + (SEXT(offset6) << 1)]};
\text{setcc}();
\]

**Description**

An address is computed by sign-extending bits [5:0] to 16 bits, left-shifting this value by 1 bit, and adding this result to the contents of the register specified by bits [8:6]. The 16-bit word at this address is loaded into DR. The condition codes are set, based on whether the value loaded is negative, zero, or positive. The memory address specified by Base+offset will be treated as a word-aligned address. In other words, bit [0] of the address will be treated as if it is 0.

**Example**

\[
\text{LDR} \quad R4, R2, \#-5 \quad ; R4 \leftarrow \text{memWord[R2 - 10]}
\]
1.3 The Instruction Set

The LC-3b supports a rich, but lean, instruction set. Each 16-bit instruction consists of an opcode (bits[15:12]) plus 12 additional bits to specify the other information which is needed to carry out the work of that instruction. Figure 1.3 summarizes the 16 different opcodes in the LC-3b and the specification of the remaining bits of each instruction. In the following pages, the instructions are described in greater detail. For each instruction, we show the assembly language representation, the format of the 16-bit instruction, the operation of the instruction, an English-language description of its operation, and one or more examples of the instruction. Where relevant, additional notes about the instruction are also provided.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD*</td>
<td>0001</td>
<td>DR, SR1, 0, 0, SR2</td>
<td></td>
</tr>
<tr>
<td>ADD*</td>
<td>0001</td>
<td>DR, SR1, 1, imm5</td>
<td></td>
</tr>
<tr>
<td>AND*</td>
<td>0010</td>
<td>DR, SR1, 0, 0, SR2</td>
<td></td>
</tr>
<tr>
<td>AND*</td>
<td>0101</td>
<td>DR, SR1, 1, imm5</td>
<td></td>
</tr>
<tr>
<td>BR</td>
<td>0000</td>
<td>n, z, p, PC + offset</td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td>1100</td>
<td>0, 0, BaseR, 000000</td>
<td></td>
</tr>
<tr>
<td>JSR</td>
<td>0100</td>
<td>0, 0, BaseR, 000000</td>
<td></td>
</tr>
<tr>
<td>JSRR</td>
<td>0100</td>
<td>1, BaseR, 000000</td>
<td></td>
</tr>
<tr>
<td>LDB*</td>
<td>0010</td>
<td>DR, BaseR, offset</td>
<td></td>
</tr>
<tr>
<td>LDI*</td>
<td>1010</td>
<td>DR, BaseR, offset</td>
<td></td>
</tr>
<tr>
<td>LDR*</td>
<td>0110</td>
<td>DR, BaseR, offset</td>
<td></td>
</tr>
<tr>
<td>LEA*</td>
<td>1110</td>
<td>DR, PC + offset</td>
<td></td>
</tr>
<tr>
<td>NOT*</td>
<td>1001</td>
<td>DR, SR, 111111</td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td>1100</td>
<td>0, 111, 000000</td>
<td></td>
</tr>
<tr>
<td>RTI</td>
<td>1000</td>
<td>0000000000000000</td>
<td></td>
</tr>
<tr>
<td>SHF*</td>
<td>1101</td>
<td>DR, SR, A, D, imm4</td>
<td></td>
</tr>
<tr>
<td>STB</td>
<td>0011</td>
<td>SR, BaseR, offset</td>
<td></td>
</tr>
<tr>
<td>STI</td>
<td>1011</td>
<td>SR, BaseR, offset</td>
<td></td>
</tr>
<tr>
<td>STR</td>
<td>0111</td>
<td>SR, BaseR, offset</td>
<td></td>
</tr>
<tr>
<td>TRAP</td>
<td>1111</td>
<td>0, 00, trapvec18</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.2: LC-3b Instruction Formats. NOTE: + indicates instructions that modify condition codes.