ECE 411 Exam 2

- This exam has 6 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 use one sheet of notes.
- You may 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.
- The exam is meant to test your understanding. Ample time has been provided. So be patient and read the questions/problems carefully before you answer.
- *Good luck!*

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data hazards</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>Control hazards</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Software techniques for ILP</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Tomasulo</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>Speculation</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>Potpourri</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>71</strong></td>
<td></td>
</tr>
</tbody>
</table>
1. Data hazards (14 points)

A 5-stage basic pipeline with forwarding (EX → EX, MEM → EX, MEM → MEM, transparent register file) and hazard detection is lengthened to 6 stages to accommodate a new LAD instruction, shown below.

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

The address calculation and memory access occur in the EX and MEM stages, respectively. A new pipeline stage, AC, is added between MEM and WB to perform the addition between DR and the value from memory. The resulting pipeline contains six stages

\[
\text{IF - ID - EX - MEM - AC - WB}
\]

Consider the code segment below which performs some computations and places the results in R2 and R3. Assume the memory hierarchy always responds within one cycle, i.e., there are no stalls due to the cache or memory.

1  ADD R0, R0, R2
2  ADD R0, R0, R1
3  RSHFL R2, R0, 15
4  LSHF R0, R0, 1
5  ADD R3, R3, R0
6  LAD R2, R6, 0
7  LAD R3, R2, 2

Answer the questions below.

(a) (4 points) What forwarding paths must be added to eliminate avoidable hazards in the code above? Identify the source and target stages, as well as what information is needed by the forwarding unit to determine when data should be forwarded.

The forwarding path AC→EX should be added. Forwarding should occur when the instruction in AC writes to a source register of the instruction in EX.

(b) (4 points) How many cycles does the new code take to run on the 6-stage pipeline after additional forwarding paths and hazard detection are added? Do count the number of cycles needed for the pipeline to fill. Show your work.

5 (fill pipeline) + 7 (instructions) + 2 (stalls) = 14

The extra 2 cycles are due to load use hazards between the result of the first LAD and the address calculation in the second LAD.
(c) (6 points) Your teammate wants to implement a static 2-issue 6-stage pipeline. The pipeline will have full forwarding between different issue packets. The compiler for the processor schedules instructions to avoid hazards between instructions within the same issue packet. Each issue packet can contain up to two instructions with no limit on the type of instruction. What is the greatest number of cycles that your teammate’s scheme can save over (b) for the given code segment? Do count the number of cycles needed for the pipeline to fill. Explain your reasoning.

The optimal schedule is limited due to the dependencies between instructions. One way to schedule the instructions is below.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction A</th>
<th>Instruction B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADD R0, R0, R2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ADD R0, R0, R1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>RSHFL R2, R0, 15</td>
<td>LSHF R0, R0, 1</td>
</tr>
<tr>
<td>4</td>
<td>ADD R3, R3, R0</td>
<td>LAD R2, R6, 0</td>
</tr>
<tr>
<td>5</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>LAD R3, R2, 2</td>
<td></td>
</tr>
</tbody>
</table>

Cycles taken: 5 + 7 = 12

2 cycles are saved over part (b).
2. Control hazards (8 points)
A 5-stage pipelined processor with full forwarding has a 2 GHz clock frequency. A 2-bit bimodal branch prediction scheme is installed. Branches are predicted in the DECODE stage and resolved in the MEM stage. Assume that target address will be calculated in the DECODE stage AND the predicted instruction is fetched in the same cycle (i.e., we have zero stall cycles for a correctly predicted branch). Write LC-3b assembly programs (1 or more) whose execution times can be used to find out the initial state of the saturating counter. Explain how the execution times for these programs reveal the initial state of the counter. Assume counter is initialized to the same state before any program is run.

```
1  ADD R0, R0, 4
2   LOOP:
3  ADD R0, R0, -1
4   BRzp LOOP

SNT(M)→WNT(M)→WT→ST→ST(M)
WNT(M)→WT→ST→ST→ST(M)
WT→ST→ST→ST→ST(M)
ST→ST→ST→ST→ST(M)
  1  ADD R0, R0, 1
  2   BRnz LOOP
  3   BRnz LOOP
  4   NOP
  5   LOOP:

SNT→SNT
WNT→SNT
WT(M)→WNT
ST(M)→WT(M)
```
3. Software techniques for ILP (4 points)

(a) (4 points) How could we modify the following code to make use of a branch delay slot?

```c
loop:
lw $2, 100($3)
addi $3, $3, 4
beq $3, $4, loop
```

```c
loop:
addi $3, $3, 4
beq $3, $4, loop
lw $2, 96($3)
```

(b) (8 points (bonus)) Software pipelining is a technique analogous to hardware pipelining for increasing ILP by merging iterations of a loop and then reordering the instructions in them to allow parallel execution of parts of subsequent loop iterations. See the attached appendix if you don't know what software pipelining is.

Consider the following code:

```c
// n guaranteed to be at least 10
void computeInArray(short *a, int n) {
    for (int i = 0; i < n - 2; ++i)
        a[i] = a[i + 1] + a[i + 2];
}
```

Rewrite this code to benefit from software pipelining on a superscalar processor with two arithmetic and two load/store units. Assume that an instruction exists to copy data between two registers in negligible time.

```c
void computeInArray (short* a, int n) {
    a[0] = a[1] + a[2];
    short atPlusNone;
    short atPlusOne;
    short atPlusTwo = a[2];
    short atPlusThree = a[3];
    for (size_t i = 2; i < n - 2; i += 2) {
        atPlusNone = atPlusTwo;
        atPlusOne = atPlusThree;
        atPlusTwo = a[i + 2];
        atPlusThree = a[i + 3];
        short b = atPlusOne * (atPlusNone + 74) / atPlusTwo;
        short c = atPlusTwo * (atPlusOne + 74) / atPlusThree;
        a[i] = b;
        a[i + 1] = c;
    }
}
```
4. **Tomasulo (19 points)**

Consider a Tomasulo machine with the following characteristics:

- There is no speculation
- All instructions are already in the instruction queue
- There are 3 stages after the instruction queue: Issue (IS), Execute (EX), and Writeback (WB)

**Issue**
- The first instruction issues at cycle 0, i.e., the instruction in the IS stage fills the next available entry in the reservation station
- All instructions issue in order
- One instruction issues per cycle
- There are 3 entries in the unified reservation station
- If an instruction moves to its WB stage in cycle $n$, then the entry in the reservation station is available for use in cycle $n+1$

**Execute**
- There is one ALU, which takes 3 cycles in the EX stage for each instruction
- There is one load-store unit, which takes 5 cycles in the EX stage for a direct load-store instruction and 10 cycles for an indirect load-store instruction
- If an instruction moves to its WB stage in cycle $n$, then an instruction that is waiting on the same functional unit can start its EX stage in cycle $n$
- If an instruction moves to its WB stage in cycle $n$, then its dependent instruction that is waiting for data on the common data bus (CDB) can get the CDB broadcast and start its EX stage in cycle $n$
- CDB broadcast is in order of the issue of the instructions in case multiple instructions finish executing in the same cycle, i.e., the instruction that issued first gets to broadcast first if there is a conflict

**Writeback**
- WB takes one cycle
- Only one register can be written at a time
Run the following sequence of instructions on this machine.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IS</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR R1, R0, DATA1</td>
<td>0</td>
<td>1-5</td>
<td>6</td>
</tr>
<tr>
<td>NOT R2, R1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R2, R2, 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDI R3, R1, DATA2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R4, R1, R5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND R5, R3, R4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STR R5, R0 DATA1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R7, R5, 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(a) (10½ points) Complete the table on the previous page with cycle numbers.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IS</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR R1, R0, DATA1</td>
<td>0</td>
<td>1-5</td>
<td>6</td>
</tr>
<tr>
<td>NOT R2, R1</td>
<td>1</td>
<td>6-8</td>
<td>9</td>
</tr>
<tr>
<td>ADD R2, R2, 1</td>
<td>2</td>
<td>9-11</td>
<td>12</td>
</tr>
<tr>
<td>LDI R3, R1, DATA2</td>
<td>7</td>
<td>8-17</td>
<td>18</td>
</tr>
<tr>
<td>ADD R4, R1, R5</td>
<td>10</td>
<td>12-14</td>
<td>15</td>
</tr>
<tr>
<td>AND R5, R3, R4</td>
<td>13</td>
<td>18-20</td>
<td>21</td>
</tr>
<tr>
<td>STR R5, R0, DATA1</td>
<td>16</td>
<td>21-25</td>
<td>26</td>
</tr>
<tr>
<td>ADD R7, R5, 2</td>
<td>19</td>
<td>21-23</td>
<td>24</td>
</tr>
</tbody>
</table>

(b) (3 points) If the machine is modified to issue 2 instructions in the same cycle, will the performance on the sequence of instructions increase? Why or why not?
No. The second instruction is dependent on the first one, and all the other instructions cannot be issued at the same time because there are only 3 entries in the reservation station.

(c) (3 points) Suppose you can add entries in the reservation station. What is the minimum number you can add to achieve the best performance of running the sequence of instructions?
One. Adding one entry reduces the latency by 2 cycles and adding any more entries will not improve the performance.

(d) (2½ points) Does adding an additional ALU reduce the latency of running the sequence of instructions? Why or why not?
No. Adding an ALU will make some ALU instructions finish earlier, but will not help speed up the critical path LDR→LDI→AND→STR.
5. Speculation (18 points)

You are to build an out-of-order execution processor using a reorder buffer for in-order commit. Prior to the implementation, you want to analyze the behavior of the design based on the size of the hardware.

Hardware specification

- Pipeline: \( p \) stages
- Instruction queue (IQ): \( q \) entries
- Reservation station (RS): \( r \) entries/stations
- Reorder buffer (ROB): \( b \) entries
- Functional unit (FU): \( r \) units, pipelined, no differentiation between adder/int/mem units
- No data forwarding

Pipeline stages

![Diagram of pipeline stages](image-url)

Branch resolution

IQ

RS

ROB

Hardware window

out-of-order
Test code A: Loop with full dependency

1 loop:
2   instr R1, R0, R0
3   instr R2, R1, R1
4   instr R3, R2, R2
5   ...
6   instr Rn, Rn-1, Rn-1
7   BR loop

(a) (6 points) Assume you are using a test code that is a loop with many iterations and every instruction is dependent to its previous instruction (refer to the sample test code above). At cycle \( c \), the branch was resolved in pipeline stage \( k \), \( s_k \), for \( 1 < k < p \), but was mispredicted. For \( b, r, q > p \), how many entries for each instruction queue, reservation station and reorder buffer should be flushed? Explain the answer in terms of \( p, q, r, b, \) and \( k \).

Note: As illustrated in the diagram on the previous page, instruction queue will manage the instruction in the first stage of the pipeline, whereas the reservation stations will handle the instructions from second stage to the second from last stage, and the reorder buffer will be tracking the entire pipeline.

I apologize that this question had some missing information.

Assumption: similar to (b) and (c), branch was executed for the first time in the first iteration of the loop when it was predicted “not taken,” but actual branch behavior was “taken.” Also, the instructions after the loop are composed of independent instructions that will not create any stalls or bubbles. Throughput of the instructions to fetch is 1, so not multiple instructions can be fetched in one clock cycle. Then, the following number of entries will be flushed.

- Instruction queue: flush either all q entries or 1 entry
- Reservation station: \( k-2 \) entries
- Reorder buffer: \( k-1 \) entries (You can also say b-(q-k) entries but this will contain empty entries to be flushed)
Now we want to understand the design by using the test code provided below.

**Test code B**

```assembly
1    ADD R1, R0, 2
2    ADD R2, R0, -1
3 loop:
4    LDR R3, R0, 4
5    NOT R4, R3
6    AND R5, R4, 3
7    ADD R1, R1, R2
8    BRzp loop
9    ; Multiple independent instructions follow
```

(b) (6 points) Assume \( p, q, r = 6, b = 8, k = 5 \) and the latency for every instruction for each pipeline stage is 1 cycle. On the first loop branch execution/resolution, it was also mispredicted. Subsequent instructions after the loop are independent of loop instructions and independent to each other. Now, how many entries for each instruction queue, reservation station and reorder buffer should be flushed? Note that out-of-order execution starts right after the second stage of the pipeline.

- Instruction queue: 1 entry or 6 entries
- Reservation station: 3 entries
- Reorder buffer: 4 entries

(c) (6 points) Based on above test code B, what are the possible scenarios such that it will not require a flush in the reservation station on the first branch execution on mispredict? On each scenario, what is the maximum size of the bottleneck hardware? You may assume other hardwares have infinite number of entries.

Without modifying the number of pipeline stages or the stage where the branch is being resolved, reservation station or reorder buffer size can be controlled to act as a bottleneck. The maximum size for reservation station would be 3 (=6-3) entries, whereas reorder buffer can be 4 (=8-4) entries to be the bottleneck.
6. Potpourri (8 points)

(a) (2 points) Give one scenario where the amount of state needed to implement a RAM-based rename table would be the same as the amount of state needed to implement a CAM-based table. Number of physical registers is N. Number of architectural registers is M. The number of checkpoints is 0.

(b) (2 points) What is the minimum number of physical registers needed for an out-of-order processor that supports the execution of four simultaneous LC-3 threads using the same pipeline?

(c) (2 points) What is the primary advantage of a processor that can execute two 4-wide VLIW words simultaneously over a processor that can execute one 8-wide VLIW word? What is the primary disadvantage?

(d) (2 points) A typical haiku is a three-line quirky observation about a fleeting moment involving nature. The second line tends to be the longest. For example,

Haikus are easy.
But sometimes they don’t make sense.
Refrigerator.

Write a haiku about ECE 411.
Appendix: Software pipelining

Excerpted from Wikipedia:

Consider the following loop:

```
1  for (i = 1) to bignumber
2      A(i)
3      B(i)
4      C(i)
5  end
```

In this example, let \( A(i), B(i), C(i), \) be instructions, each operating on data \( i, \) that are dependent on each other. In other words, \( A(i) \) must complete before \( B(i) \) can start. For example, \( A \) could load data from memory into a register, \( B \) could perform some arithmetic operation on the data, and \( C \) could store the data back into memory. However, let there be no dependence between operations for different values of \( i. \) In other words, \( A(2) \) can begin before \( A(1) \) finishes.

Without software pipelining, the operations execute in the following sequence:

\[ A(1) B(1) C(1) A(2) B(2) C(2) A(3) B(3) C(3) \ldots \]

Assume that each instruction takes 3 clock cycles to complete (ignore for the moment the cost of the looping control flow). Also assume (as is the case on most modern systems) that an instruction can be dispatched every cycle, as long as it has no dependencies on an instruction that is already executing. In the unpipelined case, each iteration thus takes 7 cycles to complete \((3 + 3 + 1, \text{because } A(i+1) \text{ does not have to wait for } C(i))\)

Now consider the following sequence of instructions (with software pipelining):

\[ A(1) A(2) A(3) B(1) B(2) B(3) C(1) C(2) C(3) \ldots \]

It can be easily verified that an instruction can be dispatched each cycle, which means that the same 3 iterations can be executed in a total of 9 cycles, giving an average of 3 cycles per iteration.
## Appendix: LC-3b ISA

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD*</td>
<td>0001 DR SR1 0000 SR2</td>
</tr>
<tr>
<td>ADD*</td>
<td>0001 DR SR1 1 imm5</td>
</tr>
<tr>
<td>AND*</td>
<td>0101 DR SR1 0000 SR2</td>
</tr>
<tr>
<td>AND*</td>
<td>0101 DR SR1 1 imm5</td>
</tr>
<tr>
<td>BR</td>
<td>0000 n z p PCoffset9</td>
</tr>
<tr>
<td>JMP</td>
<td>1100 000 BaseR 000000</td>
</tr>
<tr>
<td>JSR</td>
<td>0100 1 PCoffset11</td>
</tr>
<tr>
<td>JSRR</td>
<td>0100 00 BaseR 000000</td>
</tr>
<tr>
<td>LDB*</td>
<td>0010 DR BaseR offset6</td>
</tr>
<tr>
<td>LDI*</td>
<td>1010 DR BaseR offset6</td>
</tr>
<tr>
<td>LDR*</td>
<td>0110 DR BaseR offset6</td>
</tr>
<tr>
<td>LEA*</td>
<td>1110 DR PCoffset9</td>
</tr>
<tr>
<td>NOT*</td>
<td>1001 DR SR 11111</td>
</tr>
<tr>
<td>RET</td>
<td>1100 000 111 000000</td>
</tr>
<tr>
<td>RTI</td>
<td>1000 00000000000</td>
</tr>
<tr>
<td>SHF*</td>
<td>1101 DR SR A D imm4</td>
</tr>
<tr>
<td>STB</td>
<td>0011 SR BaseR offset6</td>
</tr>
<tr>
<td>STI</td>
<td>1011 SR BaseR offset6</td>
</tr>
<tr>
<td>STR</td>
<td>0111 SR BaseR offset6</td>
</tr>
<tr>
<td>TRAP</td>
<td>1111 0000 trapvect8</td>
</tr>
</tbody>
</table>