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 3 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>Pipelining</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Control Hazards</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Data Hazards</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Hardware ILP</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Software ILP</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>66</td>
<td></td>
</tr>
</tbody>
</table>
1. Pipelining (10 points)
Assume we have a machine with a basic 5-stage pipeline with full forwarding. Assume the register file is not transparent.

(a) Show how the instructions in the sequence given below will proceed through the pipeline: We statically predict that the BEQ instruction is not taken. When the BEQ instruction is executed, the value in R10 is equal to the value in R20. Branches are resolved at the MEM stage. Denote the stages as (F, D, E, M, W) (7 points)

(b) Assume now the memory stage is broken up into 2 stages to improve the throughput in a pipelined datapath. The new pipeline stages are: F, D, E, M1, M2, W. Show how the instructions below would progress through this 6-stage pipeline. Assume we still have full forwarding and not-transparent register file. (3 points)
2. Control Hazards (13 points)

Given a loop that has two inner branches, B1 and B2, the results of the branches are given. Fill in the prediction made by different predictors.

Assume:
- T = Taken, NT = Not Taken.
- All counters, unless otherwise specified, are 2-bit saturating counters.
- Two bit counters are initialized as weakly taken.
- One bit counters are initialized as not taken.
- All history before the execution of the trace is not taken.

Branches are executed in the order below (left to right):

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
</tr>
</tbody>
</table>

- The PC address of the branches are PC(B1) = 0x28, PC(B2) = 0x34.
- History[2:0] is a 3-bit shift register. 0 indicates a prior branch is NT, 1 indicates a prior branch is T. The most recently resolved branch action is stored in History[0].

The predictors in the problem below are:
- Static-NT : Static Not Taken
- Static-T : Static Taken
- Bimodal : 1-bit saturating counter table indexed by PC[4:2]
- Global : 2-bit saturating counter table indexed by History[2:0]
- Gshare : Global correlation predictor. 2-bit saturating counter table indexed by XOR(PC[4:2], History[2:0])
(a) Complete the table of predictions and branch prediction accuracy. On each row, write the state of the corresponding predictor before it is updated by the branch. (9)

*Note: Two additional columns are given to you for intermediate computation.*

<table>
<thead>
<tr>
<th>Branch</th>
<th>Action</th>
<th>Static-NT</th>
<th>Static-T</th>
<th>Bimodal</th>
<th>Global</th>
<th>GShare</th>
<th>History</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>T</td>
<td>NT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>NT</td>
<td>NT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td>T</td>
<td>NT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>NT</td>
<td>NT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td>T</td>
<td>NT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>NT</td>
<td>NT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td>T</td>
<td>NT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td>NT</td>
<td>NT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Static-NT</th>
<th>Static-T</th>
<th>Bimodal</th>
<th>Global</th>
<th>GShare</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1 accuracy</td>
<td>1/5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2 accuracy</td>
<td>3/5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) From the results above, which branch prediction technique would you choose for both B1 and B2? Mark only one answer. (1)

<table>
<thead>
<tr>
<th>Static-NT</th>
<th>Static-T</th>
<th>Bimodal</th>
<th>Global</th>
<th>Gshare</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Note: question (c) is independent of previous questions.

(c) Rewrite the following LC-3 program to make use of two branch delay slots. (3)

Original:

LOOP:
ADD R0, R0, 3
AND R0, R0, -2
ADD R1, R1, -1
BRP LOOP
NOP
NOP

Your code:
3. Data Hazards (13 points)
Consider a 6 stage pipeline: IF, ID, EX, MEM1, MEM2, WB
Assume:
- The ISA is LC3-b.
- Always hit 1-cycle I-Cache.
- Always hit 1-cycle D-Cache.
- D-cache only has an 8-bit bus to the CPU. Therefore, it takes two cycles for a 16 bit read or a write to complete.
- EX calculates memory addresses.
- MEM1 retrieves the low byte of the word requested, MEM2 retrieves the high byte.
- As in your MP2 cache, the last bit of a memory address is ignored. e.g. 0x0000 and 0x0001 are treated as the same address by your cache.
- There is only 1 set of forwarding paths available: from MEM1 -> EX
- Transparent register file (Writes first half of cycle, reads second half).
- Only 1 instruction may be in MEM1/2 at a time.

(a) Identify all of the data dependencies (both true dependencies and anti-dependencies) in this program. Specify in the following format:

Type: I_m->I_n (R_x) e.g. RAW I_8->I_7 (R_9)

I1 - ADD R1, R2, R3
I2 - ADD R2, R1, R4
I3 - LDR R2, R2, R4
I4 - LDB R6, R7, 25
I5 - STR R7, R6, 12
I6 - ADD R3, R5, R7
R1 = R2 = R3 = R4 = R5 = R6 = R7 = 0 at beginning of program

(3 points)
(b) Which of the above dependencies will cause stalls? Specify in the following format:

Type: $I_m \rightarrow I_n (R_x)$, # cycles.

(2 points)

Note: part (c) on the next page.
(d) Consider instructions LBI, SBI added to the LC3 ISA.

- LBI is load byte indirect, which loads a single byte from a byte array in memory.
- SBI is store byte indirect, which stores a single byte to a byte array in memory.
- Full specification is given in Appendix A.

**Question:** in the figure on the next page, **draw forwarding paths** and **stalling logic** so that LBI and SBI cause minimal stalling but provide correct operation even when data dependencies exist for these instructions. Fully specify the control signals. Control signals and forwarded values should originate from stage latches and terminate at forwarding MUXes. Label the destination of each MUX that you use.

**Requirements:**
- You may use only basic logic gates and comparators. You must show all paths related to LBI/SBI. Do not rely on existing forwarding paths.
- You need not consider control hazards or structural hazards.
- You must show your stall logic for all data hazards involving LBI/SBI but you need not show your stall mechanism i.e., an arrow captioned “stall” is sufficient.
- To simplify your logic, you may assume that the execute stage of SBI uses the value stored in the stage register SR2 as DR.
- Enumerate each of the forwarding and stall conditions to receive partial credit in case of an incorrect implementation.

(8 points)
4. **Hardware ILP (Tomasulo) (20 points)**

Consider a Tomasulo machine with the following characteristics, see the picture for a visual of the process on the following page:

The instructions are LC3-b instruction set with a multiply instruction (MUL) implemented.

There are 3 execution units, each with 2 reservation stations. A Memory reservation station, a multiplication reservation station, and an integer reservation station.

The machine is pipelined after the instruction queue with the following characteristics:

- An instruction takes one cycle to leave the instruction queue and enter its corresponding reservation station.
- An instruction must be in a reservation station for at least one cycle.
- The cycle an instruction leaves the reservation station (issue), it starts executing.
  - i.e. an add (1 cycle) will finish the same cycle it leaves the reservation station.
- When an instruction is executing, the execution unit is busy and other ready instructions waiting for the same execution unit cannot issue.
- The cycle after the instruction finishes executing, the instruction will forward its result on the common data bus.
- If multiple instructions units want to access the common data bus on the same cycle, the instruction that will use the common data bus will be in the order of execution units: multiplication, load and store, add. Other instructions will stall that cycle.
- A reservation station entry and an ROB entry are filled by the instruction until it is committed. The next instruction filling the slot can be allotted on the same cycle as the previous instruction committed.

The memory hierarchy and multiplier are NOT pipelined.

The structures (ROB, D-Cache, and Reservation Stations) are initially empty. The instruction queue is already filled with all the instructions.

Data cache line size is 16 bytes long.

There is no load store queue (treat loads and stores as normal instructions).
The following latencies for each instruction are as follows:
  STR:  1 cycle for address calculation, 1 cycle for L1 hit, 10 cycles for miss
  LDR:  1 cycle for address calculation, 1 cycle for L1 hit, 10 cycles for miss
  MUL:   10 cycles
  ADD, ADDI, and LEA:  1 cycle: these go to the adder reservation stations

Initial values of all registers (0-7) are 0.

SEGMENT    DataSegment:
DATA_one:   DATA2  5
DATA_two:   DATA2  3
DATA_three: DATA2 10
DATA_four:  DATA2  0

LEA R0, DataSegment
LDR R1, R0, DATA_one
LDR R2, R0, DATA_two
ADDI R6, R0, 6
MUL R3, R1, R1
MUL R4, R2, R2
ADD R5, R3, R4
STR R5, R0, DATA_four
ADDI R7, R7, 1
LDR R6, R6, 0
Cycle 0

Instruction Queue

ROB

MUL
ADDI
LDR
LDR
LEA

R0 0
R1 0
R2 0
R3 0
R4 0

Loads and Stores

Cache
Adder
Multiplier
(a) Walk through Tomasulo for the code above, fill in the tables below for two ROB sizes. (9+9 points)

ROB size = 4

<table>
<thead>
<tr>
<th>Instruction</th>
<th>In Reservation Station</th>
<th>Executing</th>
<th>CDB</th>
<th>Committed</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA</td>
<td>1:1</td>
<td>2:2</td>
<td>3:3</td>
<td>4:4</td>
</tr>
<tr>
<td>LDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Write as <Starting cycle> : <Last cycle>
ROBsize = 8

<table>
<thead>
<tr>
<th>Instruction</th>
<th>In Reservation Station</th>
<th>Executing</th>
<th>CDB</th>
<th>Committed</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEA</td>
<td>1:1</td>
<td>2:2</td>
<td>3:3</td>
<td>4:4</td>
</tr>
<tr>
<td>LDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Write as $<\text{Starting cycle}> : <\text{Last cycle}>$

(b) Why may R7 not have the correct “value”? (2 points)
5. Software ILP

Assume:
- The data arrays in the cache have size 4KiB
- Block size is 64B
- Load/store instructions have 6 bit sign extended offsets using 2’s complement format (for example, an offset of -8 would be 111000)
- Memory is byte addressable, address offsets are given in terms of bytes
- Floating point multiplication takes 10 cycles
- Floating point add takes 3 cycles
- Data cache misses take 50 cycles
- Ignore latency of other operations.

```
#define N 512
double *F0 = F0_global;
double *F1 = F1_global;
double *F2 = F2_global;

for (int R3 = 0; R3 < N; R3++) {
    *F0 = *F0*(F1);
    F0++;
    F1++;
}

F0 = F0_global;

for (int R4 = 0; R4 < N; R4++) {
    *F0 = *F0 + *F2;
    F0++;
    F2++;
}
```
(a) To make the above code run faster, fuse the loops together. That is, rewrite the code with the same behavior, but only using one loop. As a simplification, consider only the dominating costs: the latency given above for data cache misses, floating point multiplication, and floating point add. Assume that due to loop iteration and pointer increment overhead, there is no pipelining across iterations. (3 points)

(b) Rewrite the code using loop unrolling (see appendix for example) to take advantage of pipelined arithmetic units (assume no pipelining across iterations). Make sure to only update pointers once per iteration, and make sure the compiled code will not need to do additional pointer arithmetic. Within this constraint, maximize the amount of pipelining being done. To save space, you may inline intermediate values (for example “a = b-c; f(a);” becomes “f(b-c);”). (4 points)
(c) How many cycles does the above code take to execute? As in (a), consider only the three dominating latencies. Assume a perfect out of order machine:

- Each execution unit is perfectly pipelined (one set of operands consumed each cycle).
- There is only one of each type of execution unit (memory unit has its own adder).
- Infinite issue width.
- No speculative execution across a branch.
- Dependencies must still be obeyed.
- Ignore commit latency including data bus contention.
- Ignore reservation station limitations.

(3 points)
Appendix A

<table>
<thead>
<tr>
<th>LBI/SBI</th>
<th>1000</th>
<th>DR/SR</th>
<th>BaseR</th>
<th>L</th>
<th>offset5</th>
</tr>
</thead>
</table>
| LBI     | DR, BaseR, offset5
| SBI     | SR, BaseR, offset5

If L == 1:

\[ \text{DR} = \text{ZEXT}[\text{memWord}[\text{baseR} + \text{SEXT}(\text{offset5}) \ll 1]] \]

Else

\[ \text{mem}[\text{memWord}[\text{BaseR} + (\text{SEXT}(\text{offset5}) \ll 1)] = \text{SR}; \]

**Description:** An address is computed by sign-extending bits [4:0] to 16 bits and adding this value to the contents of the register specified by bits [8:6]. For an LBI, the word content of memory at this address is the address of the 8-bit word data to be loaded into DR. The byte contents of memory at this address are zero-extended to 16 bits and loaded into DR. The condition codes are set, based on whether the value loaded is negative, zero, or positive. For an SBI, the word content of memory at this address is the location where SR data will be stored.

The memory address specified by \text{Base} + \text{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. The LSB is significant for the indirect address, however.
Appendix B

Loop unrolling is a transformation where statements from multiple iterations of a loop are executed in a single iteration of the transformed loop. For example:

```java
for(int i = 0; i < N; i++) {
    big_array[i] = i;
}
```

can be transformed to the following if using an unrolling factor of 4 (this is an example, the unrolling factor in problem 5 may be different):

```java
for(int i = 0; i < N-4; i+=4) {
    big_array[i] = i;
    big_array[i+1] = i+1;
    big_array[i+2] = i+2;
    big_array[i+3] = i+3;
}
```

To ensure correctness, you must also handle the case where N is not a multiple of the unrolling factor, but that is not needed for this exam. To save time while writing the body of the loop, you can just write the things that change in each line, for example:

```java
for(int i = 0; i < N-4; i+=4) {
    big_array[i] = i;
    i+1   i+1
    i+2   i+2
    i+3   i+3
}
```