ECE 411 Exam 1

- 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!

Problem 1 (14 pts): __________
Problem 2 (18 pts): __________
Problem 3 (22 pts): __________
Problem 4 (12 pts): __________
Problem 5 (13 pts): __________
Problem 6 (14 pts): __________
Total (93 pts): __________
1. ISA

a) Consider the code in LC-3b assembly on the next page. Rewrite the given loop for an accumulator ISA with instructions given in the table below. (6 points)

- Assume the assembler will translate labels into the memory addresses or offsets
- All instructions take one operand
  - i. \(<mem>\) is a memory address
  - ii. \(<imm>\) is a signed immediate value
  - iii. \(<mem/imm>\) is either of the above
- Prefix immediate values with a pound sign (#)
- You may add prologue code or data before the loop if you deem necessary
- You may modify values in memory as needed

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD &lt;mem/imm&gt;</td>
<td>Add the operand to the accumulator value</td>
</tr>
<tr>
<td>AND &lt;mem/imm&gt;</td>
<td>And the operand with the accumulator value</td>
</tr>
<tr>
<td>LD &lt;mem&gt;</td>
<td>Load the memory at the given address to the accumulator</td>
</tr>
<tr>
<td>LDI &lt;mem&gt;</td>
<td>Perform an indirect load to the accumulator, i.e., (\text{Acc} \leftarrow \text{M}[\text{M}[&lt;mem&gt;]])</td>
</tr>
<tr>
<td>ST &lt;mem&gt;</td>
<td>Store the accumulator value to the given address</td>
</tr>
<tr>
<td>STI &lt;mem&gt;</td>
<td>Perform an indirect store to the given address, i.e., (\text{M}[\text{M}[&lt;mem&gt;]] \leftarrow \text{Acc})</td>
</tr>
<tr>
<td>BR[nzp] &lt;imm&gt;</td>
<td>Perform a branch based on sign/zeroness of accumulator value</td>
</tr>
</tbody>
</table>
Load-store ISA

ORIGIN 4x0000
SEGMENT
PROLOGUE:
  AND R4, R4, 0
  LDR R5, R4, A_PTR
  LDR R6, R4, B_PTR
  LDR R3, R4, N
  BR LOOP

A_PTR:
  DATA2 A
B_PTR:
  DATA2 B
A:
  DATA2[10]? 
B:
  DATA2[10]? 
N:
  DATA2 10

LOOP:
  LDR R1, R5, 0
  ADD R4, R4, R1
  STR R4, R6, 0
  ADD R5, R5, 2
  ADD R6, R6, 2
  ADD R3, R3, -1
  BRp LOOP

HALT:
  BR HALT

Accumulator ISA

ORIGIN 4x0000
SEGMENT
PROLOGUE:
  ; Prologue code (if any) goes here
  AND #0
  BR LOOP

SUM:
  DATA2 0
A_PTR:
  DATA2 A
B_PTR:
  DATA2 B
A:
  DATA2[10]? 
B:
  DATA2[10]? 
N:
  DATA2 10

LOOP:
  ; Loop code goes here
  LDI A_PTR ; Calculate running sum
  ADD SUM
  ST SUM
  STI B_PTR
  LD A_PTR ; Increment A pointer
  ADD #2
  ST A_PTR
  LD B_PTR ; Increment B pointer
  ADD #2
  ST B_PTR
  LD N ; Decrement N
  ADD #1
  ST N
  BRp LOOP

HALT:
  BR HALT
b) Assume that the load-store ISA belongs to a single-cycle processor running at 200 MHz. Assume that memory imposes no additional delay. How long does a single iteration of the loop take to run? Show your work. (4 points)

\[
200 \text{ MHz} = 5 \text{ ns / cycle} \\
(5 \text{ ns / cycle}) \times (1 \text{ cycle / 1 instruction}) \times (7 \text{ instructions}) = 35 \text{ ns}
\]


c) Suppose the accumulator ISA belongs to a multi-cycle processor on which instruction latencies are outlined below.

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Cycles per instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD immediate / AND immediate / BR</td>
<td>3</td>
</tr>
<tr>
<td>LDI / STI</td>
<td>6</td>
</tr>
<tr>
<td>All others</td>
<td>4</td>
</tr>
</tbody>
</table>

What must the frequency be for one iteration of the loop to finish in the same time as the processor in b)? Show your work. (4 points)

\[
(4 \text{ inst} \times 3 \text{ cycles/inst}) + (2 \text{ inst} \times 6 \text{ cycles/inst}) + (8 \text{ inst} \times 4 \text{ cycles/inst}) = 56 \text{ cycles} \\
56 \text{ cycles} / 35 \text{ ns} = 1.6 \text{ GHz}
\]
2. MPs

Below is the datapath diagram of MP0. Now a new instruction is added to the instruction set. This instruction adds the contents in memory locations specified in the two source registers, stores the result in the destination register and sets the condition code.

```
ADDM R1, R2, R3;
R1 <- Mem[R2] + Mem[R3]; SetCC;
```

a) Draw clearly and neatly on the diagram how you would modify the datapath to implement this instruction. (6 points)

- Label all additional blocks
- Name all additional signals between the control and datapath
- Give a short description of any new blocks (except for muxes)
- Do not modify existing components (instead of expanding a mux, add a new mux to an input of the existing mux)
- All muxes select input 0 as default
- `alu_pass` passes the first input (from `sr1_out`) of the ALU
- Add as few components as possible for optimization
- `mem_read <= 0, mem_write <= 0, mem_byte_enable <= 11` by default

* see page 9 for the grading rubrics
b) Draw the state diagram for this new instruction. **Use as few states as possible.** (4 points)
c) Translate the following code from C to LC-3b assembly with and without ADDM and calculate the number of cycles needed for each version. Assume the memory is byte addressable and a short is 2 bytes. Assume all cache accesses are hits and take 1 cycle to complete. (6 points)

    short h[3];
    short x[3];
    short y[3];
    short i;

    for (i = 0; i < 3; i++)
        y[i] = h[i] + x[i];

Given:
    R0 contains 0
    R1 contains the address of h
    R2 contains the address of x
    R3 contains the address of the output array y

With ADDM
(Number of cycles = 5+42*3 = 131)

ADD    R4, R0, 3;     (5)

LOOP:

ADD    R5, R1, R2;     (9)
STR    R5, R3, 0;      (7)
ADD    R1, R1, 2;      (5)
ADD    R2, R2, 2;      (5)
ADD    R3, R3, 2;      (5)
ADD    R4, R4, -1;     (5)
BRp    LOOP            (6)

HALT:
BRnzp HALT

Without ADDM
(Number of cycles = 5+52*3 = 161)

ADD    R4, R0, 3;     (5)

LOOP:

LDR    R5, R1, 0;      (7)
LDR    R6, R2, 0;      (7)
ADD    R7, R5, R6;     (5)
STR    R7, R3, 0;      (7)
ADD    R1, R1, 2;      (5)
ADD    R2, R2, 2;      (5)
ADD    R3, R3, 2;      (5)
ADD    R4, R4, -1;     (5)
BRp    LOOP            (6)

HALT:
BRnzp HALT
d) Why is ADDM a good choice for dealing with large arrays? (2 points)

- ADDM only takes 9 cycles. Fewer instructions and therefore fewer fetches are needed if ADDM is used. From part c we see ADDM reduces the number of cycles by 10 for each iteration of the for loop. The performance will significantly increase when dealing with large arrays.
- Fewer registers needed

**Grading Rubrics for Part A:**
6 points – correct design (not necessarily the same as the solution)
5 points – design is not optimized (e.g. adding an adder, adding a register that is not needed)
5 points – storing Mem[SRI] to DR before reading SR2 (This does not work if DR and SR2 are the same, e.g. ADDM R1, R2, R1)
3 points – changing the value in one of the source registers
2 points – changing the values in both source registers
1 point – design is wrong but one mux is in the right place
3. Cache

A 2-way set associative write back cache with true LRU replacement requires 15 bits to implement its tag store per set (including bits for valid, dirty and LRU). The cache is virtually indexed, physically tagged. The virtual address space is 1 MB, page size is 2 KB, cache block size is 8 bytes and is byte-addressable.

a) What is the maximum total size of the data stores of the cache in bytes? (4 points)

It is a VIPT cache. So, Cache index+offset bits = Page offset bits

Total Cache size = Page size*No. of ways = 2^{11}*2 = 2^{12} bytes

b) What is the physical address space of this memory system? (4 points)

15 bits for tag store; 2 ways
⇒ 1 bit for LRU, 2 for Dirty and 2 for Valid
⇒ Total no. of Tag bits = 15 – 2 – 2 – 1 = 10
⇒ Tag bits per way = 10/2 = 5
⇒ No. of bits in physical address = Tag + Index + offset = Tag+Page offset = 5+11 = 16
⇒ Physical address space = 2^{16} bytes
c) Consider the following code.

```c
// A is an array of integers (contiguous in memory)
int i;
For (i = 0; i < 100; i++)
    A[i] *= 2;
```

An integer is 4 bytes and the cache is initially empty. What is the miss rate of the cache when running this code? Assume the beginning of the array is aligned on a block boundary. (5 points)

Each cache block can hold 2 integers. Each array element is accessed twice (Read and Write). So, 1 miss for every 4 accesses. Miss rate = ¼ = 25%

d) L1 hit takes 2 cycles. Physical memory response takes 100 cycles. What is the average memory access time? (4 points)

\[
AMAT = 2 + 0.25(100) = 27 \text{ cycles}
\]

e) You suggest to add an L2 cache to improve the performance. L2 cache hit takes 4 cycles. What is the required hit rate of the L2 cache to increase performance by 50%? (5 points)

\[
\text{Speed up} = 1.5 = \frac{\text{Old \#cycles}}{\text{New \#cycles}} = \frac{27}{x} \Rightarrow x = 18 \text{ cycles}
\]
Let L2 miss rate be y
\[
18 = 2 + 0.25(4+y(100)) \Rightarrow y = 15/25 = 0.6 = 60%
\]
Required L2 hit rate = 40%
4. Cache/VM Interaction

Let's assume a cache design very similar to what you are currently building on your MPs but smaller with the specification below. Now you want to expand the memory architecture by implementing virtual memory with TLB and page table.

Cache specification

- 8 lines per way with 2 words per line
  - 4 bytes × 8 lines per way × 2 ways = 64B
- Write-back and write allocate policy
- LRU replacement policy
- Cache hits take exactly one cycle to complete

Advanced memory specification

- 12-bit virtual addresses
- 10-bit physical addresses
- Fully associative TLB with 2 entries
  - True LRU replacement policy
- 512B pages
- Single level page table
- Virtually indexed, physically tagged
- Byte-addressable

### TLB

Note: LRU bit set to 1 indicates the true LRU entry to be replaced

<table>
<thead>
<tr>
<th>Entry</th>
<th>Valid</th>
<th>Virtual Page Number</th>
<th>Physical Page Number</th>
<th>LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Page Table

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virtual Page Number</th>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
a) Find the cache hit/miss, TLB hit/miss behavior of the memory access sequence in the order below. (8 points)

<table>
<thead>
<tr>
<th>Access #</th>
<th>Virtual Address</th>
<th>Read/Write</th>
<th>Cache Hit/Miss?</th>
<th>TLB Hit/Miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4x6B5</td>
<td>Read</td>
<td>Hit</td>
<td>Hit</td>
</tr>
<tr>
<td>2</td>
<td>4x983</td>
<td>Write</td>
<td>Miss</td>
<td>Hit</td>
</tr>
<tr>
<td>3</td>
<td>4xE46</td>
<td>Read</td>
<td>Miss</td>
<td>Miss</td>
</tr>
<tr>
<td>4</td>
<td>4x6A0</td>
<td>Read</td>
<td>Hit</td>
<td>Miss</td>
</tr>
</tbody>
</table>

Solution:
Address bits - VA: [11:9] VPN, [8:0] page offset,
PA: [9] PPN, [8:0] page offset (after translated from VA)
Access 1: Virtual Address = 011010110101
  Physical Address = 1010110101
  During translation, TLB hits on entry 0 where bold bit is mapped to, which is followed by cache hit on set 5 way 0.
Access 2: Virtual Address = 100110000011
  Physical Address = 0110000011
  TLB hits on entry 1 and sets LRU to be directed to entry 0. Then, misses on cache set 0 and replaces way 0 as LRU is set to 0 then update LRU with way 1.
Access 3: Virtual Address = 111001000110
  Physical Address = 0001000110
  TLB misses and replaces entry 0 with virtual page number 7. Cache misses on set 1 and fill in way 1 with new tag 00010.
Access 4: Virtual Address = 011010100000
  Physical Address = 1010100000
  Since the virtual page number 3 is replaced in access 3, it will be TLB miss and replace entry 1. Then, hits on cache set 0 way 1 as way 0 was replaced on access 2.
b) If TLB is increased with 4 entries but newly created entries are initially empty, will this affect the hit/miss behavior of the answer in a)? If so, how? (4 points)

Solution: Yes, TLB behavior for access 4 will be hit instead of miss. Entry 0 with virtual page of 3 wouldn’t have been evicted due to the miss from access 3, and this is accessed again at 4.
5. **Pipelining**  

a) Consider two implementations A and B of the MIPS instruction set, both built using the same technology, but using different pipelines. Both machines have a base CPI of 1.0, but have different cycle times and different stalls for control and data hazards. In particular, the pipelines stall differently for taken and not-taken branches, for loads followed by dependent instructions, and for stores.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle time</td>
<td>400 ps</td>
<td>240 ps</td>
</tr>
<tr>
<td>Taken branch stalls</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Not-taken branch stalls</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Load-use stalls</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Store stalls</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

A workload that is to be run on the processors has the instruction mix given below. Assume that 60% of branches are taken and 45% of loads are followed by a dependent instruction.

<table>
<thead>
<tr>
<th>Branches</th>
<th>Loads</th>
<th>Stores</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>15%</td>
<td>30%</td>
<td>15%</td>
<td>40%</td>
</tr>
</tbody>
</table>

i. Compute the overall CPI for the two datapaths for the workload. (4 points)

\[
\text{CPI}_A = \text{base CPI} + \% \text{branches} \times \% \text{taken branches} \times \text{stall cycles per taken branch} + \% \text{branches} \times (1 - \% \text{taken branches}) \times \text{stall cycles per not-taken branch} + \% \text{loads} \times \% \text{instructions dependent on loads} \times \text{stall cycles per dependent load} + \% \text{stores} \times \text{stall cycles per store} = 1.225
\]

\[
\text{CPI}_B = 1.975
\]
ii. The slower machine would perform better with a faster clock. How fast would the slower machine’s clock need to be to have the same performance as the faster machine (3 points)?

\[ \text{CPI}_{\text{slow}} \times \text{Cycles}_{\text{slow}} = \text{CPI}_{\text{fast}} \times \text{Cycle}_{\text{fast}} \]

\[ 1.225 \times C = 1.975 \times 240 \]

\[ C = 387 \text{ps} \]

b) Identify the RAW, WAW and WAR dependences (potential data hazards) in the code below. State whether the dependence will cause a stall. Consider a five stage pipeline with Fetch (IF), Decode (ID), Execute (EX), Memory (MEM), and Writeback (WB) stages. Branches are resolved in the ID stage. All stages take 1 cycle. Assume full forwarding. (6 points)

1: \( \text{ADD R1, R2, R3} \)

RAW:
- 1\( \rightarrow \)2 (R1)
- 2\( \rightarrow \)3 (R4) (stall)
- 2\( \rightarrow \)5 (R4)
- 4\( \rightarrow \)5 (R4) (stall)

2: \( \text{LD R4, 0(R1)} \)

WAW:
- 1\( \rightarrow \)3 (R1)
- 2\( \rightarrow \)4 (R4)

WAR:
- 2\( \rightarrow \)3 (R1)
- 3\( \rightarrow \)4 (R4)

3: \( \text{ADD R1, R4, R5} \)

4: \( \text{SUB R4, R6, R7} \)

5: \( \text{BEQZ R4, DONE} \)
6. **Potpourri**
   
a) Give one reason why the pipeline performance may decrease beyond a certain number of stages. (3 points)

b) A multicycle design was observed to perform worse than the corresponding single cycle design for some workloads. What could be the reason? (2.5 points)

c) Reducing the frequency of a processor appeared to affect the CPI. Give one possible reason. (2.5 points)

d) Write C code for a program for which random replacement policy will work better than LRU for a fully associative cache. (4 points)

e) Person riding a bike while wearing a sombrero (a type of wide brimmed Mexican hat)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD*</td>
<td>0 001</td>
<td>DR SR1 0 00 SR2</td>
</tr>
<tr>
<td>ADD*</td>
<td>0 001</td>
<td>DR SR1 1 imm5</td>
</tr>
<tr>
<td>AND*</td>
<td>0 010</td>
<td>DR SR1 0 00 SR2</td>
</tr>
<tr>
<td>AND*</td>
<td>0 010</td>
<td>DR SR1 1 imm5</td>
</tr>
<tr>
<td>BR</td>
<td>0 000</td>
<td>n z p PCoffset9</td>
</tr>
<tr>
<td>JMP</td>
<td>1 100</td>
<td>0 00 BaseR 000000</td>
</tr>
<tr>
<td>JSR</td>
<td>0 010</td>
<td>1 PCoffset11</td>
</tr>
<tr>
<td>JSRR</td>
<td>0 010</td>
<td>0 00 BaseR 000000</td>
</tr>
<tr>
<td>LDB*</td>
<td>0 010</td>
<td>DR BaseR offset6</td>
</tr>
<tr>
<td>LDI*</td>
<td>1 010</td>
<td>DR BaseR offset6</td>
</tr>
<tr>
<td>LDR*</td>
<td>0 010</td>
<td>DR BaseR offset6</td>
</tr>
<tr>
<td>LEA*</td>
<td>1 110</td>
<td>DR PCoffset9</td>
</tr>
<tr>
<td>NOT*</td>
<td>0 011</td>
<td>DR SR 111111</td>
</tr>
<tr>
<td>RET</td>
<td>1 100</td>
<td>0 00 111 000000</td>
</tr>
<tr>
<td>RTI</td>
<td>1 000</td>
<td>000000000000</td>
</tr>
<tr>
<td>SHF*</td>
<td>1 101</td>
<td>DR SR A D imm4</td>
</tr>
<tr>
<td>STB</td>
<td>0 011</td>
<td>SR BaseR offset6</td>
</tr>
<tr>
<td>STI</td>
<td>1 011</td>
<td>SR BaseR offset6</td>
</tr>
<tr>
<td>STR</td>
<td>0 011</td>
<td>SR BaseR offset6</td>
</tr>
<tr>
<td>TRAP</td>
<td>1 111</td>
<td>0 000 trapvec8</td>
</tr>
</tbody>
</table>

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