ECE 411 Exam 1

• 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 use one sheet of notes.

• You may use a calculator.

• Do not do anything that might be perceived as cheating. The minimum penalty for cheating 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>ISA</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>MP</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Cache</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Cache and VM</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Pipelining</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>70</strong></td>
<td></td>
</tr>
</tbody>
</table>
1. ISA (14 points)

Branch predication is a computer architecture design strategy that allows each instruction to either perform an operation or do nothing based on a condition. The condition, called a *predicate*, is determined based on the value of a general purpose register. For example, the following instruction executes when R1 is positive and does nothing if R1 is non-positive.

\[ (R1) \text{ADD R2, R3, 3} \]

Answer the questions below.

(a) (2 points) List one advantage and one disadvantage for implementing predication compared to using the traditional branch instruction.

Any answer that makes sense is accepted. There are many advantages/disadvantages you can talk about.

Example:
Advantage: More compact code, more spacial locality for instructions
Disadvantage: harder to implement, complex

(b) (5 points) Rewrite the following simple program using predicates. The resulting program should not have any branch or jump instructions. Explain any optimizations that you make and feel free to overwrite registers R1 and R2 as long as the result is not affected. *Note: Each instruction needs to have a predication register. Assume R3 is initially 0.*

```plaintext
1 ; R0 = 0  R5 = 1
2 ; VALUE1 and VALUE2
3 ; are both non-zero
4 LDR R1, R0, VALUE1
5 BRp FOO
6 CONT: LDR R2, R0, VALUE2
7 BRn BAR
8 END: ADD R3, R3, 0
9 BRp GOOD
10 ADD R3, R0, 0xBADD
11 HALT
12 GOOD: ADD R3, R0, 0x600D
13 HALT
14 FOO: ADD R3, R0, 1
15 LEA R7, CONT
16 JMP R7
17 BAR: AND R3, R0, 1
18 LEA R7, END
19 JMP R7
```
Assuming R3 is 0 as initial condition

1  (R5) LDR   R1,R0,VALUE1        
2  (R1) ADD   R3,R0,1             
3  (R5) LDR   R2,R0,VALUE2        
4  (R5) RSHFL R2,R2,15            //get sign bit  
5  (R2) AND   R3,R0,1             //THIS LINE SETS R3 to ZERO  
6  (R3) ADD   R3,R0,0x600D        
7  (R3) HALT                              
8  (R5) AND   R3,R0,0xBAAD         
9  (R5) HALT                              
(c) (2 points) What characteristics should VALUE1 and VALUE2 satisfy to make R3 end with value 0x600D/0xBADD?

Since value1 and value2 are both non zero we get 0x600D when value1 and value2 both positive, 0xBAAD else.

(d) (5 points) Given a system that is able to execute two parallel instructions (which do not have data dependencies on each other), rewrite the following code with predication implemented and explain how predication can help speed up the code.

If you need additional registers in your code, you are free to use registers T1, T2, and T3.

```assembly
1 ; Assume R0 has value 0,
2 ; R9 has value 1
3 LDR R2, R0, VALUE1
4 LDR R3, R0, VALUE2
5 ...;
6 ; Some instructions that
7 ; compute the value to
8 ; put in location ANS
9 ; based on R2 and R3.
10 ...
11 LDR R1, R0, ANS
12 BRp SEG1
13 BRn SEG2
14 END: ...
15 ; Wrapping up
16 ...
17 HALT
18 SEG1: ; Do multiply
19 MULT R4, R5, R6
20 STR R4, R0, ANS
21 LEA R7, END
22 JMP R7
23 SEG2: ; Do divide
24 DIV R8, R5, R6
25 STR R8, R0, ANS
26 LEA R7, END
27 JMP R7
```
There are two main points for this problem.

- Predication eliminates branch/jmp instructions to shorten code.
- Predication allows us to execute both SEG1 and SEG2 in parallel, each equipped with a different predication register.

```plaintext
1  /* opening stage is not important and is thus ignored */
2    P0       P1
3  (R9) LDR  R1,R0,ANS    No-op
4  (R9) RSHFL T1,R1,15     No-op
5  (R1) MULT R4,R5,R6      (T1) DIV R8,R5,R6
6  (R1) STR  R4,R0,ANS     (T1) STR,R8,R0,ANS
7  (R9) HALT             (R9) HALT
```
2. MP (15 points)

(a) (6 points) A copy of the MP 1 datapath is attached at the end of this exam. Implement the new instruction *Memory Increment* by modifying the datapath. The instruction definition is:

\[
\text{MEMINC } \text{BaseR, offset6}
\]

\[
\text{memWord[BaseR+SEXT(offset6)\ll 1]} \leftarrow \text{memWord[BaseR+SEXT(offset6)\ll 1]} + 1
\]

In the table below, list all the components of the given datapath that you need to change and then give the specific change for each component. You should not add any new components, just modify the original components and add signals. For example, you cannot add a MUX, but you can expand an existing MUX to be of larger size. *Only the table will be graded. No exceptions.*

<table>
<thead>
<tr>
<th>Component</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUMUX</td>
<td>Expand to take MDR output (mem_wdata) as input</td>
</tr>
<tr>
<td>ALU</td>
<td>Allow B + 1 operation</td>
</tr>
</tbody>
</table>

(b) (4 points) Complete the state machine below to implement the new instruction.

From Fetch ➔ Decode ➔
(c) (3 points) For the given C code, complete the equivalent LC-3b assembly programs below. One uses the original LC-3b ISA and the other uses the ISA with the Memory Increment instruction included. Assume N is positive.

```c
for (int i = 0; i < N; i++)
    vals[i]++;
```

**Original ISA**

```
LDR R1, R0, N
LEA R2, VALS
LOOP:
    LDR R3, R2, 0
    ADD R3, R3, 1
    STR R3, R2, 0
    ADD R2, R2, #2
    ADD R1, R1, #-1
    BRp LOOP
END: BRnzp END
N: DATA2 0x0040
VALS: DATA2 0x0002
```

**ISA w/ Memory Increment instruction**

```
LDR R1, R0, N
LEA R2, VALS
LOOP:
    MEMINC R2, 0
    ADD R2, R2, #2
    ADD R1, R1, #-1
    BRp LOOP
END: BRnzp END
N: DATA2 0x0040; Variable N
VALS: DATA2 0x0002
DATA2 0x0021
... ; Rest of Array Omitted
```

Give an expression for the number of instructions saved in the new ISA's program as a function of N.

2N

(d) (2 points) A student proposes putting a loop counter in memory and using the new memory increment instruction to increment the counter every iteration. What is a performance issue with this proposed method even if condition codes could be generated for memory locations?

Memory Latency
3. Cache (16 points)

(a) A 2-way set associative write back cache with true LRU replacement policy requires \(15 \times 2^{29} \times 2^9\) bits of storage to implement its tag store (including bits for valid, dirty, and LRU). The cache is virtually indexed and physically tagged. The virtual address space is 1 MB, the page size is 2 KB, and each cache block is 8 bytes.

i. (2 points) What is the size of the data store in bytes?

Ans. 8 KB.

The cache is 2-way set associative.
So each set has 2 tags each of size \(t\) valid bits, 2 dirty bits and 1 LRU bit (because a single bit is enough to implement perfect LRU for 2-way set associative cache).
Tag store size = \(2^i \times (2 \times t + 2 + 2 + 1) = 15 \times 2^9\)
Therefore, \(2t = 10 \Rightarrow t = 5, \ i = 9\)
Data store size = \(2^i \times (2 \times 8) \ \text{bytes} = 2^9 \times (2 \times 8) = 8 \ \text{KB}\)

ii. (2 points) How many bits of the virtual index come from the virtual page number?

Ans. 1 bit

Page size is 2 KB. Hence the page offset is 11 bits (bits 10:0).
The cache block offset is 3 bits (bits 2:0) and the virtual index is 9 bits (bits 11:3).
Therefore, one bit of the virtual index (bit 11) comes from the virtual page number.

iii. (2 points) What is the physical address space of this memory system?

Ans. 64 KB

The page offset is 11 bits.
The physical frame number, which is the same as the physical tag is 5 bits.
Therefore, the physical address space is \(2^{(11+5)} = 2^{16} \ \text{bytes} = 64 \ \text{KB}\)

(b) (2 points) Lyle is dreaming of a multicore LC-3s processor with virtual memory support. The processor will have 10 KB pages, a block size of 256 bytes, and a shared L3 cache. Lyle wants to make the L3 cache as large as 128 MB, but is concerned about the synonym problem and thinks it will complicate the design. What solution would you suggest to Lyle to solve the synonym problem?

Ans. L3 cache should have no synonym problem as it is physically addressed
(c) (8 points) Below are four different sequences of memory addresses generated by a program running on a processor with a cache. The *cache hit ratio* for each sequence is also shown.

<table>
<thead>
<tr>
<th>Sequence No.</th>
<th>Address Sequence</th>
<th>Hit Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0, 512, 1024, 1536, 2048, 1536, 1024, 512, 0</td>
<td>0.33</td>
</tr>
<tr>
<td>2</td>
<td>0, 2, 4, 8, 16, 32</td>
<td>0.33</td>
</tr>
<tr>
<td>3</td>
<td>0, 512, 1024, 0, 1536, 0, 2048, 512</td>
<td>0.25</td>
</tr>
<tr>
<td>4</td>
<td>0, 64, 128, 256, 512, 256, 128, 64, 0</td>
<td>0.33</td>
</tr>
</tbody>
</table>

Assume that
• the cache is *initially empty* at the beginning of each sequence,
• all memory accesses are one byte accesses,
• all addresses are byte addresses.

Find parameters below such that a cache with the discovered parameters would behave according to the above table.

i. Associativity
   
   Ans. **4**

   For sequence 1, blocks 0, 512, 1024 and 1536 are the only ones that are reused and could potentially result in cache hits when they are accessed the second time. Three of these four blocks should hit in the cache when accessed for the second time to give a hit rate of 0.33 i.e. \( \frac{3}{9} \). Given that the block size is 8 and for either cache size (256 B or 512 B), all of these blocks map to set 0. Hence, an associativity of 1 or 2 would cause at most one or two of these four blocks to be present in the cache when they are accessed for the second time, resulting in a maximum possible hit rate of less than \( \frac{3}{9} \). However the hit rate for this sequence is \( \frac{3}{9} \). Therefore an associativity of 4 is the only one that could potentially give a hit ratio of 0.33 or \( \frac{3}{9} \).

ii. Block size
   
   Ans. **8 bytes**

   For sequence 2, only 2 out of the 6 accesses (specifically those to addresses 2 and 4) can hit in the cache, as the hit ratio is 0.33. With any other cach blocks size but 8 bytes, the hit ratio is either smaller or larger than 0.33. Therefore the cache block size is 8 bytes.

iii. Total cache size
   
   Ans. **256 B.**

   For sequence 3, a total cache size of 512 B will give a hit rate of \( \frac{4}{9} \) with a 4-way associative...
cache and 8 byte blocks regardless of the replacement policy which is higher than 0.33. Therefore the total cache size is 256 bytes.

iv. Replacement policy (LRU, Pseudo LRU, or FIFO)
Ans. **LRU**
For the aforementioned cache parameters, all cache lines in sequence 4 map to set 0. If a FIFO replacement policy were used, the hit ratio would be \( \frac{3}{8} \), whereas if LRU replacement policy were used, the hit ratio would be \( \frac{1}{4} \). Therefore the replacement policy is LRU.
(d) (8 points (bonus)) You have three processors D, E, and F each with only one level of cache. The caches have the following parameters:

- All caches have
  - a total size of 128 bytes
  - a block size of 32 bytes
  - LRU replacement policy
- D uses a direct mapped cache
- E uses a 2-way set associative cache
- F uses a fully associative cache

A benchmark was run to evaluate the processors which tests memory read performance by issuing read requests to the cache. Assume the caches are empty at the beginning of the benchmark.

The benchmark generates the following cache accesses:

\[ \text{A B A H B G H H A E H D H G C C G C A B H D E C C B A D E F} \]

Each letter is a unique cache block and all eight cache blocks are contiguous in memory. However, the ordering of letters does not correspond to the cache block ordering in memory.

i. The benchmark running on processor D generates the following sequence of cache misses:

\[ A B A H B G A E D H C G C B D A F \]

Identify which cache blocks belong in the same set (for the cache in processor D).

**Ans.**

A and B

C and G

H and D

E and F

ii. For processor E, the benchmark crashes after the following sequence of cache misses:

\[ A B H G E \]

Can you identify which cache blocks are in the same set for the cache of processor E? Explain your answer.

**Ans.** If h was in the same set as A and B then the B right after H would have missed. Similarly if G was in the same set as A and B, the A right before E have missed. Using this information the sets are respectively,

A, B, E, and F

H, G, C, and D

The simulation data of the cache up to the point it breaks is shown below.

- **Req:** A B A H B G H H A E
- **LRU0:** - A B B A A A A B A
- **LRU1:** - - - - - H G G G G
iii. Write down, in order of generation, the sequence of cache misses for the benchmark running on processor F.

Ans. By simulating the cache and using the requests:

| Req  | A | B | A | H | B | G | H | H | A | E | H | D | H | G | C | C | G | C | A | B | H | D | E | C | C | B | A | D | E | F |
| Miss | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| LRU  | - | - | - | - | A | A | B | G | G | A | A | E | D | D | D | H | G | C | A | B | H | H | D | E | C | B | A |

iv. What is the cache miss rate for the benchmark running on processor F?

Ans. \( \frac{19}{30} \)
4. Cache and VM (16 points)

Consider a memory system with the following parameters and components:

- Byte addressable
- 256 byte \(2^8\) byte page size

**Cache:**

- Virtually-indexed and physically-tagged
- 4-way set-associative with 6 4 index bits
- 4 KB \(2^{12}\) byte) 1 KB \(2^{10}\) byte) data storage (excluding bits for dirty, valid, tag and LRU)
- Read allocate policy
- Indexing the data array takes 10 ns
- Indexing the tag array takes 8 ns
- Tag comparison takes 4 ns
- Multiplexing the output data takes 3 ns
- A cache miss takes 100 ns to access the main memory and allocate to the cache line
- Assume a hit or miss is detected immediately after the tag comparison
- Initially empty (all lines are invalid)

**TLB:**

- Fully-associative
- A TLB access takes 5 ns
- Read allocate policy
- TLB is updated on a TLB miss
- All entries are listed below

<table>
<thead>
<tr>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0001</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>0100</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1010</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0100</td>
</tr>
</tbody>
</table>

**Page Table:**

- Single level page table
- A page table access takes 80 ns
- Some of the entries are listed below

<table>
<thead>
<tr>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0000</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0000</td>
</tr>
</tbody>
</table>
(a) (12 points) Fill in the blanks and calculate the cache access times for the following actions in sequence (the second action follows immediately after the first one). Show your calculations for full credit. Write the address in hex and circle hit or miss.

i. Read virtual address 4x00107
   - Cache access time: 207 ns
   - Physical address: x3307
   - TLB hit / miss
   
   VA: 00 0000 0001 0000 0111 TLB miss on VPN = 00 0000 0001
   PA: 0011 0011 0000 0111 Cache Compulsory Miss
   
   5 + 80 + 5 = 90 ns to get the physical tag
   4 ns to detect a cache miss by tag comparison
   100 ns to access the main memory
   10 + 3 = 13 ns to access data (tag access can be done in parallel)

ii. Read virtual address 4x34500
   - Cache access time: 13 ns
   - Physical address: x3300
   - TLB hit / miss
   
   VA: 11 0100 0101 0000 0000 TLB hit on VPN = 11 0100 0101
   PA: 0011 0011 0000 0000 Cache hit
   
   Parallelism:
   Indexing tag + tag comparison = 8 + 4 = 12 ns
   Indexing data + multiplexing data = 10 + 3 = 13 ns

(b) (2 points) Write a virtual address in hex that will cause a page fault. How will it be handled when this address is accessed?
   x000XX/x103XX/x3FFXX
   A page fault exception must be handled by OS to find missing page in secondary storage (disk).

(c) (2 points) When should the TLB be flushed? Explain why.
   On context switch, so one program cannot access memory of a different program (security).
5. Pipelining (9 points)

(a) (3 points) What is highest speedup possible through pipelining for a 6 instruction program if latch delay is 2 ns and total combinational logic delay of a non-pipelined design is 10 ns?

The highest speedup can be obtained when dividing the total combinational logic into $N$ stages, which each stage taking $10/N$ ns.

On the non-pipelined processor, the program takes $6 \times 10 = 60$ ns to run.

On a pipelined processor, each stage takes $10/N + 2$ ns. The program in total takes

$$\frac{(N-1) + 6}{N} \times \left(\frac{10}{N} + 2\right)$$

$$= \frac{(N+5)}{N} \left(\frac{10}{N} + 2\right) \text{ ns}$$

The total time is minimized for $N = 5$. The speedup is $\frac{60}{40} = 1.5$.

(b) (2 points) Give a specific example of a sequence of assembly instructions that include a data dependency (hazard) that cannot be resolved by forwarding alone.

1. LDR R1, R0, 0
2. ADD R0, R1, R1

(c) (4 points) Consider the code below.

1. add $t0, $s0, $s1
2. xor $t1, $t0, $s2
3. lw $s0, -12($a0)
4. sub $s5, $s0, $s1

Is it possible to resolve any of the hazards in the above code by reordering the instructions so that forwarding would be unnecessary? If yes, show how. If not, explain why not.

It's possible to resolve one, but not both of the hazards in the above code by moving the xor after the sub.

1. add $t0, $s0, $s1
2. lw $s0, -12($a0)
3. sub $s5, $s0, $s1
4. xor $t1, $t0, $s2
Appendix: MP 0 Datapath
Appendix: LC-3b State Diagram
## Appendix: LC-3b ISA

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD*</td>
<td>0001</td>
<td>DR SR1 0 00 SR2</td>
</tr>
<tr>
<td>ADD*</td>
<td>0001</td>
<td>DR SR1 1 imm5</td>
</tr>
<tr>
<td>AND*</td>
<td>0101</td>
<td>DR SR1 0 00 SR2</td>
</tr>
<tr>
<td>AND*</td>
<td>0101</td>
<td>DR SR1 1 imm5</td>
</tr>
<tr>
<td>BR</td>
<td>0000</td>
<td>n z p PCoffset9</td>
</tr>
<tr>
<td>JMP</td>
<td>1100</td>
<td>000 BaseR 000000</td>
</tr>
<tr>
<td>JSR</td>
<td>0100</td>
<td>1 PCoffset11</td>
</tr>
<tr>
<td>JSRR</td>
<td>0100</td>
<td>0 00 BaseR 000000</td>
</tr>
<tr>
<td>LDB*</td>
<td>0010</td>
<td>DR BaseR offset6</td>
</tr>
<tr>
<td>LDI*</td>
<td>1010</td>
<td>DR BaseR offset6</td>
</tr>
<tr>
<td>LDR*</td>
<td>0110</td>
<td>DR BaseR offset6</td>
</tr>
<tr>
<td>LEA*</td>
<td>1110</td>
<td>DR PCoffset9</td>
</tr>
<tr>
<td>NOT*</td>
<td>1001</td>
<td>DR SR 111111</td>
</tr>
<tr>
<td>RET</td>
<td>1100</td>
<td>000 111 000000</td>
</tr>
<tr>
<td>RTI</td>
<td>1000</td>
<td>000000000000</td>
</tr>
<tr>
<td>SHF*</td>
<td>1101</td>
<td>DR SR A D imm4</td>
</tr>
<tr>
<td>STB</td>
<td>0011</td>
<td>SR BaseR offset6</td>
</tr>
<tr>
<td>STI</td>
<td>1011</td>
<td>SR BaseR offset6</td>
</tr>
<tr>
<td>STR</td>
<td>0111</td>
<td>SR BaseR offset6</td>
</tr>
<tr>
<td>TRAP</td>
<td>1111</td>
<td>0000 trapvect8</td>
</tr>
</tbody>
</table>