ECE 411, Final Exam

- This exam has 6 problems and a total of 17 pages. 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.

- 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 (12 pts): ___________
Problem 2 (18 pts): ___________
Problem 3 (24 pts): ___________
Problem 4 (6 pts): ___________
Problem 5 (12 pts): ___________
Problem 6 (8 pts): ___________

Total (80 points): ___________
1. Parallel Architectures (12 points)

a) Draw state transition diagram for MOSI (Modified, Owned, Shared, Invalid) cache coherence protocol for:

(i) Local processor actions (4 points)

(ii) Bus requests. (4 points)
b) Pack the following operations into 4-instruction VLIW words for a VLIW processor with 1 Load/Store unit and 3 ALUs. Each field of the three VLIW words should either be 1, 2, 3, 4, 5, or nop. (4 points)

1. LD R1, R2  /*R1 ← Mem[R2]*/
2. ADD R3, R1, R4  /*R3 ← R1+R4*/
3. OR R1, R5, R6  /*R1 ← R5 OR R6*/
4. ADD R2, R7, R0  /*R2 ← R7_R0*/
5. ST R5, R7  /*Mem[R7] ← R5*/

<table>
<thead>
<tr>
<th>VLIW 1</th>
<th>1. LD R1, R2</th>
<th>4. ADD R2, R7, R0</th>
<th>NOP</th>
<th>NOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLIW 2</td>
<td>5. ST R5, R7</td>
<td>2. ADD R3, R1, R4</td>
<td>3. OR R1, R5, R6</td>
<td>NOP</td>
</tr>
<tr>
<td>VLIW 3</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
</tbody>
</table>
2. Performance (18 points)

Consider the following piece of code:

```c
register int i, j;
register float sum1, sum2;
float a[64][64], b[64][64];

for (i=0; i<64; i++) {   \(*\) Line 1
    for (j=0; j<64; j++) {  \(*\) Line 2
        sum1 += a[i][j];  \(*\) Line 3
    }
    for (j=0; j<32; j++) {  \(*\) Line 4
        sum2 += b[i][2*j]; \(*\) Line 5
    }
}
```

Assume the following:
- There is a perfect instruction cache; i.e., do not worry about the time for any instruction accesses.
- Both int and float are of size 4 bytes.
- Assume that only the accesses to the array locations `a[i][j]` and `b[i][j*2]` generate loads to the data cache. The rest of the variables are all allocated in registers.
- Assume a fully associative, LRU data cache with 32 lines, where each line has 16 bytes.
- Initially, the data cache is empty.
- The arrays `a` and `b` are stored in row major form.
- To keep things simple, we will assume that statements in the above code are executed sequentially. The time to execute lines (1), (2) and (4) is 4 cycles for each invocation. Lines (3) and (5) take 10 cycles to execute and an additional 40 cycles to wait for the data if there is a data cache miss.
- There is a data prefetch instruction with the format `prefetch(array[index])`. This prefetches the entire block containing the word `array[index]` into the data cache. It takes 1 cycle for the processor to execute this instruction and send it to the data cache. The processor can then go ahead and execute subsequent instructions. If the prefetched data is not in the cache, it takes 40 cycles for the data to get loaded into the cache.

a) How many cycles does the above code fragment to execute if we do NOT use prefetching? Also calculate the average number of cycles per outer-loop iterations. (4 points)

**Solution 1:** Each line has 4 values, so every 4 accesses in line 3 will miss, and every 2 in line 5, for a total of $64*(16+16) = 2048$ misses.

- Line 1 executes 65 times, $65*4 = 260$
- Line 2 executes $64*65$ times, $64*65*4 = 16640$
- Line 3 executes $64*64$ times, $64*64*10 = 40960$
- Line 3 misses $64*64/4$ times, $64*64/4*40 = 40960$
- Line 4 executes $64*33$ times, $64*33*4 = 8668$

ECE 411, Final Exam
Line 5 executes 64*32 times, 64*32*10 = 20480
Line 5 misses 64*32/2 times, 64*32/2*40 = 40960
Total cycles = 168708

Average number of cycles per outer-loop iteration = 2636.0625

Solution 2: It was accepted for the case when the first line of the loop was considered 64 or 32 times.
Line 1 executes 64 times, 64*4 = 256
Line 2 executes 64*64 times, 64*64*4 = 16384
Line 3 executes 64*64 times, 64*64*10 = 40960
Line 3 misses 64*64/4 times, 64*64/4*40 = 40960
Line 4 executes 64*32 times, 64*32*4 = 8192
Line 5 executes 64*32 times, 64*32*10 = 20480
Line 5 misses 64*32/2 times, 64*32/2*40 = 40960
Total cycles = 168192

Average number of cycles per outer-loop iteration = 2628

b) Consider inserting prefetch instructions for the two inner loops for the arrays a and b respectively. Explain why we may want to unroll the loops to insert prefetches. What is the minimum number of times you would need to unroll for each of the two loops for this purpose? (5 points)

Solution: There is one miss every four iterations of the first loop, and every two iterations of the second loop. The latency of this miss covers the prefetch time. We only need to issue a prefetch instruction once per cache line accessed. If code size is not a problem, unrolling the loop is the most efficient way to do this (it avoids branches that test for the correct iteration count). The first loop would need to be unrolled 4 times, and the second two times.

c) Unroll the inner loops for the number of times identified in part b), and insert the minimum number of software prefetches to minimize execution time. The technique to insert prefetches is analogous to software pipelining. You do not need to worry about the startup and cleanup code and do not introduce any new loops. (5 points)

Solution:
Register int i, j;  //i, j are in the processor registers
Register float sum1, sum2, a[64][64], b[64][64];

for (i=0; i<64; i++) {    // Line 1
    for (j=0; j<64; j+=4) {   // Line 2
        prefetch(a[i][j+4]);  // Line P1
        sum1 += a[i][j];  // Line 3a
sum1 += a[i][j+1]; // Line 3b
sum1 += a[i][j+2]; // Line 3c
sum1 += a[i][j+3]; // Line 3d
}
for (j=0; j<32; j+=2) {   // Line 4
    prefetch(b[i][2*j+4]);  // Line P2
    sum2 += b[i][2*j];  // Line 5a
    sum2 += b[i][2*j+2];  // Line 5b
}

\[ \text{d) How many cycles does the code in part c) take to execute? Calculate the average speedup over the code without prefetching. Assume prefetches are not present in the startup code. Extra time needed by prefetches beyond the end of the loop execution time should not be counted. (4 points)} \]

Solution 1: Now only the only misses are on the very first execution of line 3a (row major ordering means prefetching is effective even across outer iterations), and the first two executions of line 5a (the prefetch is preparing for the j+2 iteration). There are 3 misses total.

Line 1 executes 65 times, \(65*4 = 260\)
Line 2 executes 64*17 times, \(64*17*4 = 4352\)
Line P1 executes 64*16 times, \(64*16*1 = 1024\)
Line 3a-3d each execute 64*16 times, \(64*16*4*10 = 40960\)
Line 3a misses only on its every first execution, \(40*1 = 40\)
Line 4 executes 64*17 times, \(64*17*4 = 4352\)
Line P2 executes 64*16 times, \(64*16*1 = 1024\)
Line 5a, 5b each execute 64*16 times, \(64*16*2*10 = 20480\)
Line 5a misses on the first two executions, \(40*2 = 80\)
Total cycles = 72572

The speedup over the code with no prefetching is \(168708/72572\), approximately 2.32.

Solution 2:
Line 1 executes 64 times, \(64*4 = 256\)
Line 2 executes 64*16 times, \(64*16*4 = 4096\)
Line P1 executes 64*16 times, \(64*16*1 = 1024\)
Line 3a-3d each execute 64*16 times, \(64*16*4*10 = 40960\)
Line 3a misses only on its every first execution, \(40*1 = 40\)
Line 4 executes 64*16 times, \(64*16*4 = 4096\)
Line P2 executes 64*16 times, \(64*16*1 = 1024\)
Line 5a, 5b each execute 64*16 times, \(64*16*2*10 = 20480\)
Line 5a misses on the first two executions, \(40*2 = 80\)
Total cycles = 72056

The speedup over the code with no prefetching is \(168192/72056\), approximately 2.334.
3. Data/Control Hazards (24 points)

The program stores the result of A[i] AND B[i] for i between 0 and N-1 to C[i], using the following code:

```
I1: LDR R1, R0, A
I2: LDR R2, R0, B
I3: LDR R3, R0, C
I4: LDR R7, R0, N
LOOP:
I5: LDR R4, R1, 0
I6: LDR R5, R2, 0
I7: ADD R1, R1, 2
I8: ADD R2, R2, 2
I9: AND R6, R4, R5
I10: STR R6, R3, 0
I11: ADD R3, R3, 2
I12: ADD R7, R7, -1
I13: BRzp LOOP
```

- Assume a standard five stage pipeline (IF, ID, EX, MEM, WB)
- All arithmetic instructions take 1 clock cycle in the EX stage
- Memory instructions take 1 clock cycle (i.e. 100% cache hits) in the MEM stage
- Ignore pipeline startup cycles in your cycle calculations
- The loop is called previously, meaning all branch predictor state bits have been initialized using the previous execution of the loop
- 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 instruction in WB writes to a register, a dependent instruction in ID can read that value during the same cycle
- Cache is Writeback

a) Data Forwarding

i) Without data forwarding, identify where and how many bubbles need to be inserted into the code above? (assume static branch prediction – predict Not Taken) (2 points)

Solution: Between I9 and I10, you need 2 bubbles. AND → STR (RAW)
In addition, you could have 3 bubbles after branch
ii) How many cycles (in terms of N) does this code take to run without any data forwarding? (assume static branch prediction – predict Not Taken) (2 points)

Solution: \(4 + 11N + 3(N-1) = 14N + 1\)

4 (setup) + 9N (instructions in loop) + 2N (no-ops from AND \(\rightarrow\) STR) + 3(N-1) (branch miss… Note: \(\rightarrow\) 3N if you listed 3 bubbles after branch above)

iii) How many cycles (in terms of N) can be saved with data forwarding (i.e., (total #cycles) - (total # cycles with forwarding))? (assume static branch prediction – predict Not Taken) (2 points)

Solution: \(4+9N+3(N-1) = 12N + 1\)

4 (setup) + 9N (instructions in loop) + 3(N-1) (branch miss…Note: \(\rightarrow\) 3N if you listed 3 bubbles after branch above)

b) Branch prediction (assume data forwarding for all of part b)

i) With a static predict taken scheme:

(1) How many cycles (in terms of N) does this code take to run without a BTB? (2 points)

Solution: \(10N + 7\) (or \(10N + 6\))

4 (setup) + 9N (instructions in loop) + N (extra instruction from lack of BTB) + 3

(1 branch miss)

(2) How many cycles (in terms of N) does this code take to run with a BTB? (assume that BTB lies in the Fetch stage and takes one cycle to access) (2 points)

Solution: \(9N + 7\)

4 (setup) + 9N (instructions in loop) + 3 (1 branch miss)

ii) With a 1-bit prediction scheme:

(1) How many cycles (in terms of N) does this code take to run without a BTB? (2 points)

Solution: \(10N + 10\) (or \(10N + 9\))

4 (setup) + 9N (instructions in loop) + N (extra instruction from lack of BTB) + 6

(2 branch miss)

(2) How many cycles (in terms of N) does this code take to run with a BTB? (2 points)

Solution: \(9N + 10\)

4 (setup) + 9N (instructions in loop) + 6 (2 branch miss)
iii) With a 2-bit prediction scheme:

1) How many cycles (in terms of N) does this code take to run without a BTB? (2 points)

Solution: \(10N + 7\) (or \(10N + 6\))
\[\text{4 (setup)} + 9N \text{ (instructions in loop)} + N \text{ (extra instruction from lack of BTB)} + 3 \text{ (1 branch miss)}\]

2) How many cycles (in terms of N) does this code take to run with a BTB? (2 points)

Solution: \(9N + 7\)
\[\text{4 (setup)} + 9N \text{ (instructions in loop)} + 3 \text{ (1 branch miss)}\]

c) Loop Unrolling

i) A software technique to reduce hazards is loop unrolling. Use loop unrolling to unroll the code above three times (i.e., three iteration worth of work in a new single iteration), optimizing out redundant instructions where possible. Assume N is a multiple of 3. 

\[\text{Hint: use the offset field of LDR/STR to help eliminate redundant instructions.} \ (4 \text{ points})\]

Solution:

\[
\begin{align*}
\text{LDR R1, R0, A} & \quad \# \text{load pointer to A} \\
\text{LDR R2, R0, B} & \quad \# \text{load pointer to B} \\
\text{LDR R3, R0, C} & \quad \# \text{load pointer to C} \\
\text{LDR R7, R0, N} & \quad \# \text{load iteration counter} \\
\text{LOOP:} \\
\text{LDR R4, R1, 0} & \quad \# \text{load element of A} \\
\text{LDR R5, R2, 0} & \quad \# \text{load element of B} \\
\text{ADD R1, R1, 6} & \quad \# \text{increment pointer to A (avoid load-use)} \\
\text{AND R6, R4, R5} & \quad \# \text{execute AND statement} \\
\text{STR R6, R3, 0} & \quad \# \text{store result at C} \\
\text{LDR R4, R1, -4} & \quad \# \text{load element of A} \\
\text{LDR R5, R2, 2} & \quad \# \text{load element of B} \\
\text{ADD R2, R2, 6} & \quad \# \text{increment pointer to B (avoid load-use)} \\
\text{AND R6, R4, R5} & \quad \# \text{execute AND statement} \\
\text{STR R6, R3, 2} & \quad \# \text{store result at C} \\
\text{LDR R4, R1, -2} & \quad \# \text{load element of A} \\
\text{LDR R5, R2, -2} & \quad \# \text{load element of B} \\
\text{ADD R2, R3, 6} & \quad \# \text{increment pointer to C (avoid load-use)} \\
\text{AND R6, R4, R5} & \quad \# \text{execute AND statement} \\
\text{STR R6, R3, 0} & \quad \# \text{store result at C} \\
\text{LDR R3, R1, -2} & \quad \# \text{load element of A} \\
\text{ADD R7, R7, -3} & \quad \# \text{decrement iteration counter} \\
\text{BRzp LOOP} \\
\end{align*}
\]
ii) How many cycles (in terms of N) does this code take to run? Assume we have no branch prediction or data forwarding. (2 points)

Solution: $4 + 26N - 3 = 1 + 26N$, where $N/3$ is actually number of iterations
4 (setup) + 17 (N) (instructions in loop) + 6 (N) (no-ops from AND $\rightarrow$ STR) + 3(N - 1)
(extra instruction from lack of BTB)
4. Instruction Set Architecture (6 points)

An unaligned memory access is a memory access that crosses the boundary of the word size supported by the memory hierarchy. In the LC3b, all accesses are aligned: a single load can fetch bytes 0 and 1 or bytes 2 and 3, but there is no single instruction to fetch both bytes 1 and 2.

Out of your love for computer architecture, you decide to modify the LDR instruction in a pipelined LC3b processor to support unaligned accesses.

(a) What changes must be made to the implementation of the instruction? Assume that the memory hierarchy cannot be changed to support unaligned accesses, and avoid decreasing performance for aligned loads. (2 points)

Solution: The major change is to change LDR to fetch two words instead of one on an unaligned access. An aligned access will still require just one memory access. You can determine whether the access is aligned or unaligned by looking at the low bit of the (SR + offset<<1).

An unaligned access requires two memory accesses (similar to LDI or STI). You then have to pick and choose the correct byte out of each access and combine them to create the result.

(b) The memory hierarchy of your processor contains the following:

- 256 B 2-way set associative L1 cache with 128-bit line size; 30ns access time
- 1KB 8-way set associative L2 cache with 256-bit line size; 60 ns access time (256 bit memory bandwidth; 600ns access time).

Give an address that, when loaded, will cause a worst-case unaligned load in this system? How long does it take (consider only the delay from memory)?

* For simplicity, assume that the access times are cumulative (e.g. an L1 miss that can be resolved by the L2 takes a total of 30+60 = 90 ns). (4 points)

Solution: The address has to essentially cause an L1 and L2 miss for both accesses. So, (30ns + 60ns + 600ns)*2 = 1380ns.
An address that would cause this would be 0x00ff.
5. Caches and Virtual Memory (12 points)

A computer system has the following specification:

- Data cache size is 1KB and 4-way associative. Block size is 16 bytes. It is physically indexed and physically tagged. It uses LRU replacement algorithm. It uses write-allocate and write-back policy.
- Physical address has 24 bits. Byte-addressable.
- Virtual address has 32 bits.
- Page size is 64KB.
- TLB is 2-way set-associative, has 32 entries and uses LRU replacement policy.

a) For each of the following question, indicate the bits of the virtual address that correspond to: (4 points)

Solution: We can think of a virtual address as (page number, page offset).

- The virtual page offset:
  
  The virtual page offset: 
  The page size is 64 KB, so 64 KB = 2^16. 16 bits are needed to indicate the page offset. The last 16 bits of the virtual address (virtual address bits 15:0) correspond to the page offset.

- The virtual page number:
  
  The virtual page number: 
  Since the last 16 bits are for page offset, we have 32-16 = 16 bits for virtual page number. Virtual address bits 31:16 correspond to the virtual page number.

Solution: We can also think of the virtual address as (tag, index, page offset).

- The TLB index:
  
  The TLB index: 
  The TLB has 32 entries total and is 2-set way associative, so we will need 32/2 = 16 = 2^4 sets. We need 4 bits to indicate the TLB index. We know the page offset takes the last 16 bits of the virtual address and the index field is before the offset field, so 19:16 correspond to the index.

- The TLB tag
  
  The TLB tag: 
  32 – 4 index bits 4 – 16 offset bits = 12 bits are left for TLB tag, which are the first 12 bits of the virtual address. 31:20 correspond to the tag field.
b) For each of the following question, indicate the bits of the physical address that correspond to: (4 points)

Solution: We can think of a physical address as (page frame number, page offset)

- The physical page offset:
  The physical page offset:
  Same as virtual page offset, which is 16 bits. The last 16 bits of the physical address (15:0) correspond to this field.

- The physical page number:
  The physical page number:
  The remaining bits of physical address are 24 – 16 = 8 bits. The first 8 bits of the physical address (23:16) correspond to this field.

Solution: We can also think of the physical address as (tag, index, block offset).

- The cache index:
  The cache index:
  The cache is 4-way set-associative and the cache size is 1 KB, so we have 1 KB/ (4 * 16) = 2^4 sets. We will need 4 bits to indicate this field, so 7:4 bits of the physical address correspond to this field.

- The cache tag:
  The cache tag:
  24 – 4 index bits – 4 block offset bits = 16, so the first 16 bits of the physical address correspond to tag.
c) The following table gives the state of the TLB, providing the address translations for the relevant virtual addresses (using hexadecimal notation). Valid bit of 1 implies the entry is valid. (4 points)

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Physical page number</th>
<th>Valid</th>
<th>Index</th>
<th>Tag</th>
<th>Physical page number</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>FFC</td>
<td>EF</td>
<td>1</td>
<td>7</td>
<td>560</td>
<td>6F</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>FF9</td>
<td>28</td>
<td>1</td>
<td>6</td>
<td>A31</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>F55</td>
<td>CD</td>
<td>1</td>
<td>5</td>
<td>9E1</td>
<td>55</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>446</td>
<td>1F</td>
<td>1</td>
<td>4</td>
<td>151</td>
<td>48</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>6AC</td>
<td>D5</td>
<td>1</td>
<td>3</td>
<td>0A0</td>
<td>6A</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>31A</td>
<td>D4</td>
<td>1</td>
<td>2</td>
<td>68E</td>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>39E</td>
<td>5A</td>
<td>1</td>
<td>1</td>
<td>A34</td>
<td>EE</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>732</td>
<td>DE</td>
<td>1</td>
<td>0</td>
<td>7A0</td>
<td>99</td>
<td>0</td>
</tr>
</tbody>
</table>

The following table lists a stream of virtual address accesses by the processor (all addresses are hexadecimal). Complete the rest of the entries in the table using the above information and your solutions to parts A and B. For the TLB hit and Cache hit, specify “yes” or “no.” Assume initially the cache is empty.

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Corresponding Physical Address</th>
<th>Part of Physical Address that indexes Cache</th>
<th>TLB hit?</th>
<th>Cache hit?</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFF ABCD</td>
<td>EF ABCD</td>
<td>C</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>446C CEBA</td>
<td>1F CEBA</td>
<td>B</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>48F8 ABCD</td>
<td>B2 ABCD</td>
<td>C</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>446C CEAB</td>
<td>1F CEAB</td>
<td>A</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>
6. Potpourri (8 points)

a) What is the primary difference between a GPU and a multi-core architecture? (2 points)

b) How does clock gating affect static power? Explain. (2 points)

c) What is the advantage of a crossbar interconnect over a bus interconnect? (2 points)

d) You have to architect a processor for Google search. Would you build a multi-core processor, a simultaneously multithreaded processor, or a hybrid? Why? (2 points)
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.

![Instruction Set Table]

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