CS233 Exam 6
November 29th, 2016

Name: ____________________________________________________________
NETID: __________________________________________________________

Circle the section that you attend (so we can hand back your exam).

<table>
<thead>
<tr>
<th>Monday</th>
<th>Tuesday</th>
</tr>
</thead>
<tbody>
<tr>
<td>(11am-noon) Craig</td>
<td>(11am-noon) Chamila</td>
</tr>
<tr>
<td>(noon-1pm) Litian</td>
<td>(noon-1pm) Vishaal</td>
</tr>
<tr>
<td>(1-2pm) Litian</td>
<td>(1-2pm) Chamila</td>
</tr>
<tr>
<td>(2-3pm) Ed</td>
<td>(2-3pm) Geoffrey</td>
</tr>
<tr>
<td>(3-4pm) Vishaal</td>
<td>(3-4pm) Dustyn</td>
</tr>
<tr>
<td>(4-5pm) Ed</td>
<td>(4-5pm) Dustyn</td>
</tr>
</tbody>
</table>

- This exam has 12 pages; the final sheets are provided as reference and scratch paper, but must be handed in with the exam.
- You have 120 minutes.
- No calculators or other electronics are allowed. You may bring one 8.5” x 11” sheet of handwritten notes that must be handed in with the exam.
- To make sure you receive credit, please write clearly and show your work.
- We will not answer questions regarding course material.

<table>
<thead>
<tr>
<th>Question</th>
<th>Maximum</th>
<th>Your Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>
Question 1: Pipelining (40 points)

Above is a pipelined architecture that is similar to the one discussed in class except that operations associated with the ALU have been spread out over two pipelined execution stages: EX1 and EX2. In addition, there is no forwarding from EX1 to any stage. Branches are predicted as not-taken and are resolved in the MEM stage. Unconditional branches are resolved in the ID stage.

### Part(a) (10 points)

Given the structure latencies (shown at the left), what is the speed up achieved by this pipelining compared to a non-pipelined version (you may assume a large number of instructions have executed and that there are no stalls or flushes). **Explain.**

#### Note: the pipeline registers have a 2ns write latency!

For a large number of instructions the pipeline fill time is negligible. With no stalls & flushes, the performance difference will be just the cycle time difference.

Non-pipelined: \( 5 + 7 + 15 + 8 + 18 + 10 = 63 \text{ ns} \)

Pipelined: \( \text{max}(5+2, 7+2, 15+2, 8+2, 18+2, 10) = 20 \text{ ns} \)

**Speedup** = \( \frac{\text{old latency}}{\text{new latency}} = \frac{63}{20} \)
Part (b) (10 points)
Annotate the following code to identify as many true data dependencies as you can.

```assembly
add $t0, $zero, 21
sub $t1, $t0, $t0  # stall x1
loop: sub $t0, $t0, 1
    beq $t0, 0, done  # stall x21
    and $t2, $t0, 1
    beq $t2, 1, odd  # stall x20
    j even
odd: add $t3, $zero, $t0
    j continue
even: add $t3, $zero, $zero
continue: add $t1, $t1, $t3  # stall x10 (when from even)
    j loop
done: sub $t3, $t3, $t3
```

Part (c) (15 points)
Calculate the total number of cycles the above code takes to execute on the 6-stage pipeline on the previous page, from the IF stage of the first instruction to the WB stage of the last instruction. Explain. We must understand your computation to give you points.

#fills = 5 (for a 6-stage pipeline)
#instructions = 2 prologue + 20*4 loop + 10*1 j even + 10*2 odd + 10*1 even + 20*2 continue + 2 final loop + 1 done = 2 + 80 + 10 + 20 + 10 + 40 + 3 = 165
#flushes = 1 * #jumps + 4 * #taken_branches
= 1*(10 j even + 10 j continue + 20 j loop) + 4*(1 beq done + 10 beq odd) = 1*40 + 4*11 = 84
#stalls = 1 + 21 + 20 + 10 = 52
Cycles = #fills + #instructions + #flushes + #stalls = 5 + 165 + 84 + 52 = 306

Part (d) (5 points)
Could the running time of the above code be improved by removing one of the pipeline registers to make the machine into a 5-stage pipeline? If so, indicate which one and explain why/how performance would improve. If not, explain why not.

The only pair of stages that we can merge without increasing the cycle time is IF and ID. Merging IF and ID will allow us to resolve jumps in the first stage of the pipeline and reduce the pipeline length by 1 cycle, allowing us to reduce the number of flushes by 1 for both jumps and branches. As such, it would save us 51 cycles.
Question 2: Cache Analysis (30 points)

For a 8KB 2-way set associative L1 cache with LRU replacement and with blocks of size 16 bytes on a machine with 24-bit address space (both virtual and physical) with no hardware prefetching, consider the following code snippet. Given that the cache is initially empty, the memory is byte addressable, and that data for multi-dimensional arrays is stored in row-major order, answer the following questions. A is stored starting at address 0x110fe0 and B is stored starting at address 0xa80fe0. Assume that the function ceil makes no memory accesses.

#define N 1024
int A[N][N]; // integers are 4B  each row is 4KB; total is 4MB
double B[N];  // doubles are 8B  total is 8KB
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        A[i][j] += ceil(B[i]);
    }
}

Part (a) (5 points)
Compute the address trace for the first 6 memory accesses and also mark if each access is a HIT or a MISS.

<table>
<thead>
<tr>
<th>Address</th>
<th>HIT/MISS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110fe0</td>
<td>miss</td>
</tr>
<tr>
<td>0xa80fe0</td>
<td>miss</td>
</tr>
<tr>
<td>0x110fe0</td>
<td>hit</td>
</tr>
<tr>
<td>0x110fe4</td>
<td>hit</td>
</tr>
<tr>
<td>0xa80fe0</td>
<td>hit</td>
</tr>
<tr>
<td>0x110fe4</td>
<td>hit</td>
</tr>
</tbody>
</table>

Part (b) (10 points)
Compute the total number of accesses and the total number of misses for the above code for the specified cache configuration. Explain.

Accesses = (2 loads + 1 store)/iteration * N*N iterations = 3 N^2 accesses = 3 * 1024 * 1024 accesses

Accesses to both arrays are along the rows, which means we get the full benefit of spatial locality (4 elements of A, 2 elements of B in each cache block). There is no re-use of A, so there is no temporal locality. Accesses to B are almost entirely temporal locality, but as soon as we’re done with an element, we never return to it. Because there is no non-trivial reuse, we only care about 1 cache block from each array at a time; because of this and because the cache is two-way set associative, we don’t need to worry about cache conflicts. Instead, we need just the number of misses to bring the data in.

Misses (on A) = (4 * 1024 * 1024 B) / (16B / miss) = 256 * 1024 misses
Misses (on B) = (8 * 1024 B) / (16B / miss) = 512 misses
Misses (total) = 256 * 1024 + 512 misses
**Question 2: Cache Analysis**

**Part (c) (5 points)**
If we added a 64KB 2-way set associative L2 cache with 32B blocks, what would be the number of accesses and misses for the L2 cache? Explain.

The number of access of L2 cache would be equal to the number of misses of L1 cache: $256.5 \times 2^{10}$ accesses.

The block size is double to the L1 cache, thus would fetch more information from memory, make half of the accesses become hits. So the number of misses would be: $256.5 \times 2^9$ misses.

**Part (d) (5 points)**
For the L1 cache on the previous page, compute the number of misses if the code was changed to the following: Explain.

```c
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        A[j][i] += ceil(B[i]);
    }
}
```

The B traversal doesn’t change, but the A traversal is now column-wise, meaning that there is no longer spatial locality unless we can fit a whole column into the cache. A column consists of 1024 16B cache blocks (or 16KB), so it is too big. Furthermore, because each row is 4KB and our cache is (effectively) 2 4KB caches put together, A[1][0] is going to have the same cache index as A[0][0]. For both of these reasons there will be no reuse of the cache blocks of A across iterations (there will still be hits on all of the stores because the loads will fetch the data for the stores).

Misses (on A) = $N^2$ misses = $1024 \times 1024 = 2^{20}$ misses
Misses (on B) = unchanged from part (b) = 512 misses

**Part (e) (5 points)**
For the original code, compute the number of misses that would occur if the L1 cache was changed to be direct mapped with all the other parameters unchanged. Explain. Feel free to explain qualitatively (for partial credit) even if you can’t compute a result.

This would create one set of conflicts every two rows of the A matrix, because it takes 2 rows = 8KB to fill the cache. Each time a conflict occurs, we’ll get 8 additional misses. Assume A[x][y] through A[x][y+3] are in the same cache block at it alias with B[x]. Presumably, B[x] is already in the cache, because of its high degree of temporal locality. As such, we’d expect access patterns like:

We’d get 1024/2 such conflicts for a total of $8 \times 1024 / 2 = 4096$ additional misses.
Question 3: Cache-aware Programming (30 points)

Part (a) (10 points)
What must be true about the original version of any piece of code for tiling to be a profitable transformation? Explain. (You might find it helpful to write a loop where tiling would be helpful on the scratch paper to answer this question.)

To benefit from tiling, a piece of code must have data reuse, but not enough temporal locality to benefit from that reuse. Tiling increases temporal locality by shortening the amount of other data that is accessed between subsequent uses of a given piece of data. If a piece of code has no data reuse or if the data is already in the cache when reuse occurs, the piece of code can’t benefit from tiling.

Part (b) (20 points)
Rewrite the following code to optimize its cache performance on a system with software prefetching and a single-level cache. The cache is a 2-way set-associative 32KB cache with 64B blocks. Ints and floats are 4 byte data types. Software prefetch syntax is shown on the right. Assume that the cache is initially empty and that the presents array starts at an address divisible by 4096. Explain the transformations you applied and why. Your score will depend both on how much you optimize and how well you explain.

```c
#define NUM_PRESENTS 12000
#define NUM_STORES 7
struct item {
    float best_price;
    float prices[NUM_STORES];
}

float total_cost = 0;
struct item presents[NUM_PRESENTS];  // initialized elsewhere

for (int i = 0; i < NUM_STORES; i++) {
    for (int j = 0; j < NUM_PRESENTS; j++) {
        presents[j].best_price =
            MIN(presents[j].prices[i], presents[j].best_price);
    }
}

for (int j = 0; j < NUM_PRESENTS; j++) {
    total_cost += presents[j].best_price;
}
return total_cost;
```

void __builtin_prefetch (const void *addr,
    unsigned rw, unsigned locality);

addr: the address of the memory to prefetch.
rw: (optional) a compile-time constant 1 or 0; 1: the program anticipates writing the data soon, 0 (default) in the near term, the program expects to only read the data.
locality: (optional) a compile-time constant from 0 to 3. 0: data has no temporal locality, so need not be left in the cache after the access, 3: (default) data has a high degree of temporal locality and should be retained in all levels of cache if possible. Use 0 or 3 based on the expected reuse.
Question 3: Cache-aware Programming (cont.)

This program suffers from three problems:

1. Bad data locality in accesses to the presents array, because we’re not accessing all of the elements of one present (which are likely in the same cache block) before moving on to the next. We fix this through loop inversion.

2. Even after we fix the traversal order, we’d still have to stall on demand misses for our data. We fix this by adding software prefetch instructions. We prefetch 16 iterations ahead in the code below, because each present fills a cache line and that should be enough cache lines to tolerate the memory latency and/or keep the memory system occupied. (Any number in the large single digits to small double digits should be reasonable.)

3. With two separate loops, we need to do 2 traversals through the presents array. We fix this by loop fusion.

```c
for (int j = 0; j < NUM_PRESENTS; j++) {
    __builtin_prefetch(&presents[j+16], 1, 0); // read/write, no locality
    float current_best_price = presents[j].best_price;
    for (int i = 0; i < NUM_STORES; i++) {
        current_best_price =
            MIN(presents[j].prices[i], current_best_price);
    }
    presents[j].best_price = current_best_price;
    total_cost += current_best_price;
}
return total_cost;
```