CIS501 – Computer Architecture **Midterm Exam Solutions**
Prof. Martin **Thursday, Nov. 2nd, 2010**

1. **[9 Points] True or False**. If a statement is "false," briefly explain how so by describing how the statement may most simply be made true. Unjustified (or poorly justified) "false" answers will be marked wrong. Simply stating the negation of the false statement is *not* sufficient justification. *Please be specific!*

   (a) Integrated circuits can lead to such wonders as home computer, automatic controls for automobiles, and personal portable communications equipment.

   Answer: True

   (b) Moore's 1965 paper predicts an exponential improvement in transistor switching speeds over time.

   Answer: False. Moore's paper predicts an exponential increase in the number of transistors per chip.

   (c) The cost to manufacture a chip is proportional to the area of the chip.

   Answer: False. Because yields decrease with chip size (due to defects), the cost of a chip is super-linear in the area of the chip.

   (d) DRAM capacity is improving faster than its access latency.

   Answer: True.

   (e) The SPEC CPU benchmarks are a reasonable way to compare the performance of two processors with **different ISAs** and **different compilers**.

   Answer: True.

   (f) A compiler optimization can increase performance yet hurt CPI.

   Answer: True.

   (g) The ARM ISA is an example of a RISC ISA.

   Answer: True.

   (h) The principle of spatial locality states that recently referenced data is likely to be referenced again soon.

   Answer: False, the statement describes temporal locality.

   (i) A loop stream detector mitigates the dependence cross-checking problem in a superscalar processors.

   Answer: False. A loop stream detector increases superscalar fetch rate.

2. **[6 Points] Processor Performance.**

   (a) Assume a typical program has the following instruction type breakdown:

   - 30% loads
   - 15% stores
   - 50% adds
   - 4% multiplies
   - 1% divides

   Assume the current-generation processor has the following instruction latencies:

- loads: 2 cycles
- stores: 5 cycles
- adds: 1 cycle
- multiplies: 14 cycles
- divides: 50 cycles

If for the next-generation design you could pick one type of instruction to make twice as fast (half the latency), which instruction type would you pick? Show your work.

Answer:[3 points] First, we calculate the CPI adder of each type of instruction:
loads: 0.6
stores: 0.75
adds: 0.5
multiplies: 0.56
divides: 0.5
As **stores** contribute the most to the CPI, we should half the latency of store operations.

(b) Consider a simple in-order five-stage pipeline with full bypassing, a two-cycle branch mis-prediction penalty, and a single-cycle load-use delay penalty. For a specific program, 40% of the instructions are loads, 20% are branches, the remaining 40% of instructions are simple single-cycle ALU operations. Half of the load instructions are followed immediately by a dependent instruction. 75% of branches are predicted correctly. What is the average CPI of this program on this processor?

Answer: [3 Points] The CPI would be 1 plus an extra cycle for dependent loads (20% of instructions), an extra two cycles for mis-predicted branches (10% of 20%). Thus, the CPI is $1 + (1 * 0.20) + (2 * 0.25 * 0.20) = 1.3$.

3. **[6 Points] Virtual Memory**.

(a) What are two benefits of virtual memory?

Answer: [2 points]
Potential answers:

i. Protection/isolation: programs are prevented from accessing each other's data.
ii. Virtualization: each program has a single contiguous (and unbounded) view of memory.
iii. Paging: the operating system can swap out pages to disk when physical memory runs low. This is also what give each program the appearance that it has an unbounded amount of memory (independent of the size of the physical memory).

(b) What decides upon the mapping of virtual pages to physical pages?

Answer: [1 points] The operating system.

(c) How does the hardware avoid walking the page table at every memory access?

Answer: [1 points] It caches virtual to physical translations in the translation buffer (TB).

(d) What is the primary advantage of a multi-level page table? What is the primary disadvantage?

Answer: [2 points] Advantage: smaller when not all of virtual memory is being used. Disadvantage: misses in the translation buffer are slower because a miss must perform multiple memory accesses to walk the multi-level page table.

4. **[8 Points] Instruction Set Architectures**.

(a) ISAs have different numbers of registers, and there is no consensus on what is the "right" number of registers. Give two reasons not to have too many registers (e.g., 1000 registers) in an ISA:

Answer: [2 points] (1) Requires more bits in the instructions (leading to larger instructions). (2) A larger register file will be slower to access.

Give one reason not to have too few registers (e.g., 4 registers):

Answer: [1 point] Too few registers limits the compilers ability to keep values in registers, increasing the number of loads and stores of temporary results, too few registers makes it difficult to perform coding scheduling.

(b) The IBM 801 project moved from a 16-register ISA to a 32-register ISA. What technological shift was the driving force behind this change?

Answer: [1 points] The advances in compiler technology that allowed for the transition from hand-written assembly code to compiler-generated code.

(c) The IBM 801 project abandoned a variable-length instruction format in favor of a fixed-length instruction format. Give two advantages and one disadvantage of this change.

Answer: [3 points] Advantages: (1) easier to decode and fetch next instruction and (2) instructions don't cross page boundaries. Disadvantages: code density.

(d) The IBM RS/6000 and PowerPC ISAs are both direct decedents of the IBM 801 project. What was the most notable addition to these ISAs (and other modern ISAs) over and beyond what was present in the IBM 801?

Answer: [1 points] Floating point instructions (and a dedicated floating point register file).

5. **[8 Points] Branch Prediction Conflicts and Tagged Predictors**.

You're a microprocessor designer, and your trace-based simulations of an important workload indicate that branch instructions at the following two 32-bit addresses (in binary) are executed frequently:

- Address `A`: `1000 1101 1100 0011 0110 1011 0100 1001`
- Address `B`: `1000 1101 0110 1001 1110 1011 0100 1001`

(a) The above two instructions are likely to conflict (hash to the same entry) in a branch predictor. For a simple "bimodal" predictor of two-bit saturating counters, how many entries must the predictor have to prevent these two branches from interfering (conflicting) with each other? How many total bytes must the predictor be?

Answer: These two addresses have the lower 15 bits in common. Thus, the index bits would need to at least 16 bits to map these two addresses to different entries in the branch predictor. That would require a 64k-entry predictor, which is **16KBs**.

(b) You have a brilliant idea: "Why not create a set-associative branch direction predictor?" Your simulations indicate that a predictor with just 2048 entries (512 bytes) would be sufficient if it wasn't for these two trouble branches. Consider a two-way set-associative predictor that uses a straightforward tagging strategy (one full tag for each two-bit counter, instructions have 32-bit addresses). How large (in KBs) is a two-way set-associative 2048-entry tagged predictor?

Answer: The 2048-entry predictor has 1024 sets in each of its two ways, so the lower 10 bits are used to index into the table. Thus, the tags are each $32 - 10 = 22$ bits. Adding the two-bit counter, each entry is 24 bits or 3 bytes. 2K entries at 3 bytes each is **6KBs**. (Actually, unless random replacement was used, a LRU bit is also needed, which would increase the size of each entry to 25 bits, for 6.25KBs.)

(c) You have *another* brilliant idea: "Because this is just a predictor, it can be wrong, so it doesn't actually need the full tags, just enough of a tag to avoid this particular conflict". With this new insight, (1) how large should the tags be and (2) what is the total size in KBs of this predictor?

Answer: To avoid the conflict, we only need to "tag" the lower 16 bits of the address. 10 bits are already covered by indexing into the 2k-entry predictor (1024 sets). Thus, only 6 tag bits are need. Six bits plus the 2-bit saturating counter is a total of 8 bits (1 byte) per entry. 2k entries at 1 byte is **2KB**. (As above, an LRU bit would add a bit to each entry (9 bits) for a total of 2.25KBs.)

(d) How might a predictor capture both the conflict-mitigating benefits of a tagged set-associative predictor and most of the area efficiency of a tag-less predictor?

Answer: Some sort of "hybrid" predictor. Perhaps something similar in concept to how a small "victim buffer" can mitigate conflicts in direct-mapped caches: form a hybrid of a small set-associative predictor and a large tag-less predictor. Access the predictors in parallel, if the tag matches, use the prediction from the set-associative predictor. Otherwise, use the prediction from the large tag-less predictor. To make most efficient use of the small predictor, insert a new entry (new tag) into it only when a branch is mis-predicted (if the large table is getting a branch correct, there is no need to put that branch in the small predictor).

6. **[4 Points] Superscalar Simulation.** In the second homework assignment you simulated a single-issue pipelined processor with varying load-use latency using a simple "scoreboard" algorithm.

Modify the pseudo-code below to simulate idealized four-wide superscalar execution of independent instructions (assume perfect fetch, perfect branch prediction, and a maximum of four instructions per cycle with any mixture of operations).

```
next_instruction = true

while (true) {

  if (next_instruction) {

      read next micro-op from trace

      next_instruction = false

  }

  if instruction is "ready" based on scoreboard entries for register inputs {

      if the instruction writes a register {

          "execute" the instruction by recording when its register will be ready
      }

      next_instruction = true

  }

  advance by one cycle by decrementing non-zero entries in the scoreboard

  increment total cycles counter

}
```

Answer:
The algorithm can be adjusted to simulate a four-wide superscalar processor by just adding a loop that executes four times before advancing to the next cycle. For efficiency, the programmer would probably want to add a `break` statement once any instruction is found to be "not ready", but it isn't required.

```
next_instruction = true
while (true) {
  for (i=0; i<4; i++) {   // *** ADDED ***
    if (next_instruction) {
        read next micro-op from trace
        next_instruction = false
    }
    if instruction is "ready" based on scoreboard entries for register inputs {
        if the instruction writes a register {
            "execute" the instruction by recording when its register will be ready
        }
        next_instruction = true
    }
  }  // *** ADDED ***
  advance by one cycle by decrementing non-zero entries in the scoreboard
  increment total cycles counter
}
```

7. **[12 Points] Predication Performance.** Consider the code below and two versions of it in x86 assembly:

```
int sum_square_max(int x[], int y[], int size)
{
  int sum = 0;
  for(int i = 0; i < size; i++) {
    int val = 0;
    if (x[i] > y[i]) {
      val = (x[i] * x[i]);
    } else {
      val = (y[i] * y[i]);
    }
    sum = sum + val;
  }
  return sum;
}
```

Code A:

```
.L14:  movl  (%rdi,%rdx), %r8d
       movl  (%rsi,%rdx), %ecx
       cmpl  %ecx, %r8d
       jg .L16   // difficult to predict branch (50% taken, 50% not-taken)
.L15:  imull %ecx, %ecx
       addq  $4, %rdx
       addl  %ecx, %eax
       cmpq  %r9, %rdx
       jne .L14  // always predicted correctly
       // loop exit


.L16:  movl  %r8d, %ecx
       jmp .L15  // unconditional branch
```

Code B:

```
.L14:  movl  (%rdi,%rdx), %r9d
       movl  (%rsi,%rdx), %r8d
       movl  %r9d, %ecx
       movl  %r8d, %r11d
       imull %r9d, %ecx
       imull %r8d, %r11d
       cmpl  %r8d, %r9d
       cmovle   %r11d, %ecx
       addq  $4, %rdx
       addl  %ecx, %eax
       cmpq  %r9, %rdx
       jne .L14  // always predicted correctly
       // loop exit
```

In the first assembly translation, assume the branch labeled "difficult to predict branch" is taken/not-taken 50%/50%. The second assembly translation uses `cmov` to avoid the branch using predication.

Consider a simple single-issue pipelined processor with a **CPI of 1** when all branches are predicted correctly. The branch mis-prediction penalty is **4 cycles**.

(a) If the "difficult to predict branch" is predicted with 100% accuracy (always predicted correctly), which of the two codes is faster and how much faster is it (for example, X is 50% faster than Y)?

Answer: Code A will take either 9 or 11 cycles, with an average of 10 cycles. Code B will always take 12 cycles. Thus, Code A is 20% faster than Code B.

(b) If the “difficult to predict branch” is predicted with 0% accuracy (never predicted correctly), which of the two codes is faster and how much faster is it (for example, X is 50% faster than Y)?

Answer: Code A will take either 9+4 or 11+4 cycles, with an average of 14 cycles. Code B will still always take 12 cycles. Thus, Code B is 17% faster than Code A.

(c) For what range of prediction accuracies for the “difficult to predict” branch will Code B (the predicated code) be faster than Code A (the non-predicate code)?

Answer: The break-even point is when 10 + (4*mispred) = 12 which implies that the prediction accuracy for this bad branch must be 50% or worse for Code B to be faster.

Now consider a superscalar pipeline. Assume for the moment that its **CPI is 0.5** (2 IPC) on both of these codes. The branch mis-prediction penalty remains **4 cycles**.

(d) For what range of prediction accuracies for the “difficult to predict” branch will Code B (the predicated code) be faster than Code A (the non-predicate code)?

Answer: The break-even point is when (10*0.5) + (4*mispred) = (12*0.5) which implies that the mis-prediction rate must be 25% or lower (prediction accuracy of 75% or better) for Code B to be faster.

(e) What does this say about the value of the predicated code as the processor’s base CPI improves?

Answer: The better the core CPI, the more likely the predicated code will be beneficial.

(f) In reality, there is at least one more additional aspect of wide-issue superscalar execution that favors Code B over Code A.

Answer: The taken branch inside the loop of Code A will make it difficult to sustain high instruction fetch rates. In addition, although Code B has more instructions, it also has more independent instructions. Thus, overall, Code B should have more ILP (better CPI) than Code A in practice.

(g) What hardware feature included in recent Intel microprocessors (for example, the Core 2 and Core i7) would benefit Code B but not benefit Code A?

Answer: The loop stream detector—which prevents needing to re-fetch the loop body every time—works only when there are no mis-predicted branches inside the loop body. Code B would be the best case for such a loop stream detector.

8. **[12 Points] Memory Hierarchy Performance**

Consider the following C code:

```
double sum_square_diff(double x[], double y[], int size)
{
  double sum = 0.0;
  for(int i = 0; i < size; i++) {
    double diff = x[i] - y[i];
    sum = sum + (diff * diff);
  }
  return sum;
}
```

The loop body can be translated into eight x86 assembly instructions, two of which are loads:

```
.L2: movsd (%rdi,%rax), %xmm1      // Load
     movsd (%rsi,%rax), %xmm2      // Load
     subsd %xmm2, %xmm1
     mulsd %xmm1, %xmm1
     addsd %xmm1, %xmm0
     addq  $8, %rax
     cmpq  %rdx, %rax
     jne   .L2
```

In this question, the loop executes an extremely large number of iterations (`size` is extremely large), but the function is called only once. Each array element is a 64-bit (8-byte) floating point value.

(a) Consider a simple non-pipelined processor core with a 1Ghz clock frequency (1ns clock period) and a blocking data cache. All non-memory instructions and loads that hit in the cache have a single-cycle latency. The data cache has 8-byte blocks, is two-way set associative, and has a miss penalty of 256 nanoseconds (256 cycles at 1Ghz). The cache is initially empty. What is the CPI?

Answer: With a 8-byte block, there will be no spatial locality. As the function is called only once, there is no temporal locality either. Thus, the loop has two cache misses each iteration. $8 + 256 + 256$ = 520 cycles for 8 instructions, which is a CPI of 65!

(b) **Increasing block size.** What is the CPI if the cache block size is increased to be 64 bytes?

Answer: The 64-byte blocks capture spatial locality reducing the miss rate by 8x (12.5% miss rate versus 100%) because each element of the array is 8 bytes. On average each loop iteration will have a cache miss 25% of the time, thus the new cycles is $8 + (256 * 25\%) = 72$ cycles per iteration. Thus, the CPI is $72/8 = 9$ CPI.

(c) **Adding stream buffers.** To further reduce the cache miss penalty, consider the addition of a pair of stream buffers to the system (one stream buffer for each of the two arrays). How many 64-byte entries must each stream buffer have to entirely hide the memory latency?

Answer: In the best case the loop executes in 8 cycles and we need to prefetch 256 cycles ahead to hide all the memory latency. Thus, we need to prefetch 32 loop iterations ahead. As each iteration uses 8 bytes of data, we need to prefetch 256 bytes ahead in the data stream. With a 64-byte block size, that implies that the our stream buffers must have at least four 64-byte entries each.

(d) **Bandwidth demands.** With deep enough stream buffers and enough memory bandwidth, the system can completely hide the memory latency in this program. What is the minimum amount of bandwidth that the main memory must sustain to keep up with the processor? Give your answer in gigabytes per second (GB/second). Hint: 1 GB/second is one byte per nanosecond.

Answer: The loop executes in 8 cycles and consumes 16 bytes of data. Thus, the memory system would need to supply 2 bytes of data per cycle, which is 2 GB/second of memory bandwidth.

(e) **Impact of limited bandwidth.** If the memory system provides only 1 GB/second of bandwidth, what is the new best-case CPI?

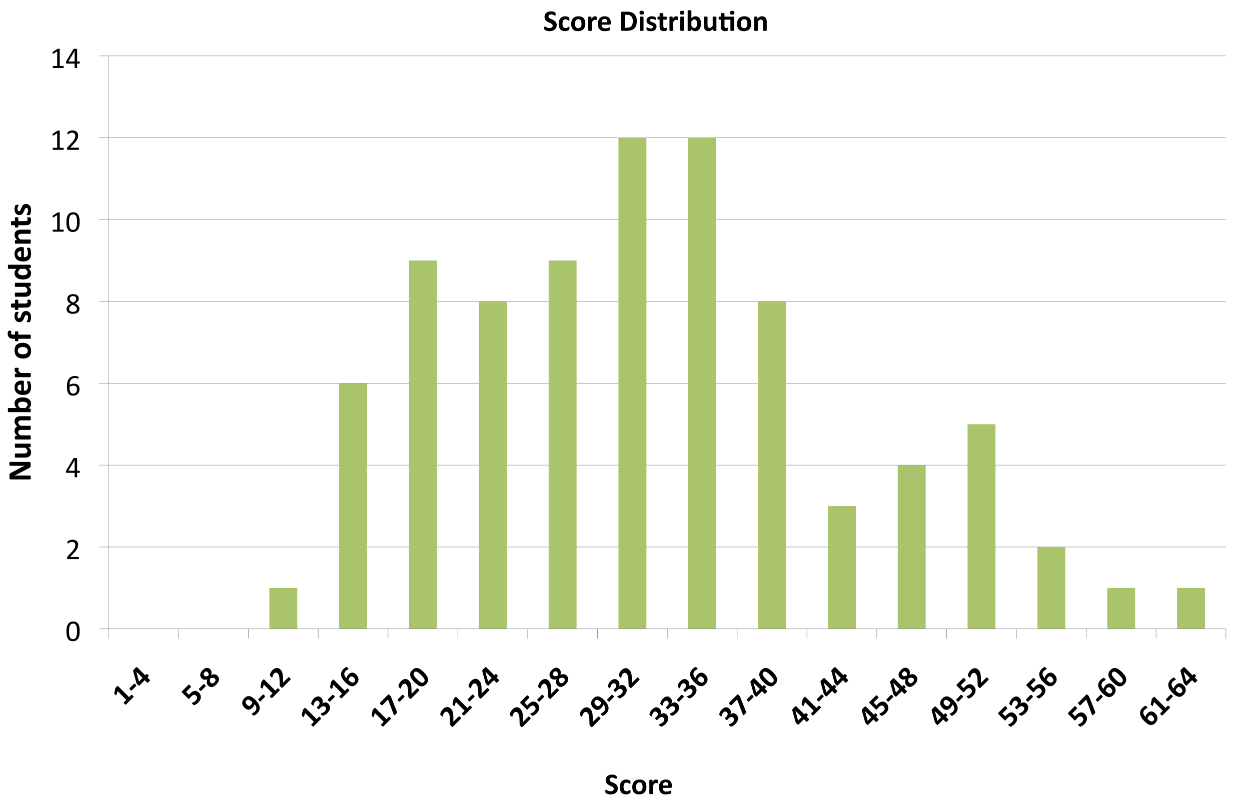
Answer: The system can now only execute one loop iteration every 16 cycles (as it requires 16 bytes of data and they system only supplies 1 byte per cycle). Thus, the new CPI is $16/8 = 2$ CPI in the best case.

(f) **Impact of faster processor.** Consider replacing the simple processor core with a pipelined superscalar processor that improves the CPI by 2x (the new CPI is 0.5) and the core clock frequency by 2x (the new clock frequency is 2Ghz). Would this change the number of entries per stream buffer needed to hide memory latency? If so, approximately how much larger or smaller? (Assume sufficient bandwidth.)

Answer: The faster processor will increase the number of entries by 4x. (The memory latency stays the same at 256ns, so making the processor 4x faster means the stream buffers need to be deeper to hide the relatively longer memory latency. The bandwidth demands on the memory system also increase by 4x.)

— End of exam —

Distribution



Mean: 31.7 points — Median: 31 points — High: 63