CIS501 – Computer Architecture	Final Exam Solutions
Prof. Martin	Monday, Dec. 20, 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) *IQ clog* can be remedied by eating more fiber.

   Answer: False, IQ Clog (on a multi-thread processor) can be mitigated with the ICOUNT fetch policy.

   (b) A fully-associative cache will never have conflict misses.

   Answer: True.

   (c) A store buffer (also called a write buffer) is primarily intended to hide store latency.

   Answer: True.

   (d) Translation lookaside buffers (TLBs) reduce the total size of the page table.

   Answer: False, TLBs reduce the latency of address translation by caching virtual to physical mappings.

   (e) Each application process has its own page table, and that page table is maintained by the operating system.

   Answer: True.

   (f) A multiplexed single core, a multicore, and a multithreaded core all provide the same shared-memory programming model.

   Answer: True

   (g) A memory consistency model specifies the "many readers" or "single writer" property for a single memory location.

   Answer: False, the above property describes coherence. In contrast, consistency is about ordering of operations to *different* memory locations.

   (h) Today's microprocessors use long vectors (64 32-bit elements or larger) to maximize the performance benefits of vectorization.

   Answer: False, today's microprocessors use short vectors of eight or fewer elements (commonly 128 bits or so) to simplify the implementation.

   (i) Power-related issues are relevant only when designing battery-powered devices.

   Answer: False, cooling issues limit the power dissipation (and thus the performance) of server and desktop processors.

2. **[8 Points] Pipelining and Clock Frequency**. Consider a workload with 40% branches and a 75% branch prediction accuracy (25% misprediction rate).

   (a) For a simple five-stage in-order scalar pipeline, what is the CPI of this workload assuming a three-cycle mis-prediction penalty and a single-cycle latency for all other instructions?

   Answer: [2 points] $1 + (40\% * 25\% * 3) = 1.3$ CPI

   (b) Consider a much deeper pipeline (but still scalar in-order) that has double the core clock frequency. Calculate under what condition does this deeper pipeline outperform the shallower pipeline?

   Answer: [2 points] At double the clock frequency, it could double the CPI (2.6 CPI) and still perform the same:
   $1 + (40\% * 25\% * N) = 2.6$ CPI
   Solving for N, give N = 16 cycles. Thus, only when the mis-prediction penalty is 16 cycles or less is the processor with the faster clock is faster.

   (c) Based on what you have learned about pipelines and calculated above, does this deeper pipeline seem like a reasonable design point? Why or why not?

   Answer: [2 points] Yes. In an in-order pipeline, the branch prediction penalty is usually around half the pipeline depth and never more than the entire pipeline depth. A well-designed 16-stage (or 32-stage) pipeline should have double the frequency of a 5-stage pipeline, making this a reasonable design point.

   (d) If cache misses and memory latency were taken into account, would the above calculation and conclusion change? If so, how? If not, why?

   Answer: [2 points] As memory latency doesn't magically shrink just because the processor core gets faster, the first order, there are no substantial impact on the calculations and conclusions drawn above. The overall speedup might be lower (as the memory latency time isn't reduced).

3. **[10 Points] Branch Direction Prediction**

(a) What is a *gshare* predictor and why does it generally increase prediction accuracy?

Answer: The gshare predictor uses a branch pattern register that records the outcomes of the most recent n branches. By hashing the branch pattern regiser with the PC before making a prediction, the predictor then examines a different two-bit counter based on the context of the prediction, increase its accuracy.

(b) For fixed branch predictor size, sometimes a *bimodal* outperforms *gshare*. Give two distinct reasons this can occur.

Answer: [2 points] (1) each static branch updates multiple different counters, placing significant pressure on the capacity of the predictor. (2) the predictor takes longer to learn all the patterns, thus has a longer warmup period (that is, a gshare predictor could have a lower prediction accuracy than bimodal even for a predictor of infinite size).

(c) What technique is used to mitigate these tensions between bimodal and gshare?

Answer: A hybrid predictor (which uses a "chooser" table to select between the bimodal or gshare predictor on each prediction).

(d) Consider a predictor with a single saturating counter which is either 1-bit or 2-bits. The 1-bit counter encodes either "Taken (T)" or "Not-taken (N)", and it is initialized to "Not-taken". The 2-bit counter encodes "Strongly Taken (T)", "Weakly Taken (t)", "Weakly Not-taken (n)" and "Strongly Not-taken (N)", and it is initialized to "Weakly Not-taken (n)".

i. Give a short sequence (4 or fewer) of branch directions (T or N) in which a 2-bit saturating counter is better than a 1-bit saturating counter. What is the prediction accuracy for *both* predictors?

Answer: [3 points] N T N — 33% vs 66% (answers may vary)

ii. Give a short sequence (4 or fewer) of branch directions (T or N) in which a 1-bit saturating counter is better than a 2-bit saturating counter. What is the prediction accuracy for *both* predictors?

Answer: [3 points] N T T — 66% vs 33% (answers may vary)

4. **[9 Points] Branch Target Prediction**

(a) What are the two purposes of a branch target buffer (BTB)?

Answer: [2 points] (1) determine if an instruction is a branch or not just based on the PC of the branch, and (2) if the branch is predicted taken, the addresses to which the branch jumps.

(b) Why is a BTB generally tagged whereas the branch history table (BHT) used in the bimodal and gshare predictors is not?

Answer: A miss in the BTBs is used to indicate that the instruction isn't a branch. Thus without tags it would likely indicate that every instruction was a branch.

(c) Even though the BTB is typically tagged, it often contains only a partial tag (just 8 or 16 bits) rather than a full tag (up to 64 bits on a 64-bit architecture).

Answer: [3 points] Advantage: fewer bits (smaller predictor). Disadvantage: aliasing will reduce the predictor accuracy. It is still correct because the BTB is just a predictor; it doesn't have to be "correct"

(d) The BTB is used for even for unconditional branches, whereas a branch direction predictor is used for only conditional branches. Why?

Answer: BTBs are used to predict the target of a branch with just the PC of the branch to allow fetching of the next instruction to proceed in parallel with instruction decode. Thus, this is helpful even for unconditional branches. In contrast, unconditional branches are always taken, so there is no need to make a branch direction prediction for them.

(e) What is the purpose of a *return address stack*? Assuming a processor with a BTB, why might it also have a *return address stack* in addition to the BTB?

Answer: [2 points] The return address stack predicts the target of function call return instructions. As the target of a return statement depends on when function called it (and the same function can be called from many places), a BTB is as effective at predicting return addresses as a return address stack.

5. **[8 Points] Memory Hierarchy Performance**

Your job is to decided on the cache configuration of your companies next-generation chip. Two of your options are: (1) a direct-mapped cache with a **two-cycle hit latency** or (2) a set-associative cache with a **three-cycle hit latency**. In both cases, the miss latency is an *additional* 10 cycles.

(a) If the miss rate of the direct-mapped cache is 15%, what is its average latency of a memory operation?

Answer: [2 points]
$2 + (15\% * 10) = 2 + 1.5 = 3.5$ cycles

(b) What miss rate must the set-associative cache obtain to have a lower average latency than the direct-mapped cache with a 15% miss rate?

Answer: [2 points]
$3 + (x * 10) = 3.5$
$(x * 10) = 0.5$
$x = 0.05 = 5\%$

(c) Alternatively, a victim cache could be used to reduce the miss latency of the cache to just 5 cycles when a miss hits in the victim cache (the victim cache and next-level of the hierarchy are accessed in parallel, so it is still just a 10 cycle miss penalty if it doesn't hit in the victim buffer). Assuming the direct-mapped cache miss rate is 15% and the victim cache has a local hit rate of 50% (that is, half of all misses find the block in the victim cache), what is the new average latency?

Answer: [2 points]
$2 + (15\% * 10 * 50\%) + (15\% * 5 * 50\%) = 2 + 0.75 + 0.375 = 3.125$ cycles

(d) Another alternative is to use use way prediction to obtain the miss rate of the set-associative cache but the hit latency of the direct-mapped cache. Assume that a way mis-prediction adds 5 cycles to the cache hit latency but doesn't impact the latency of misses.

If the set-associative cache and way-prediction cache both have a miss rate of 10%, how accurate must the way predictor be for the way prediction cache to out perform the original set-associative cache? That is, what percent of cache hits must be predict correctly?

Answer: [2 points]
Set-associative cache latency: $3 + (10\% * 10) = 4$ cycles
Way-prediction cache latency: $2 + (10\% * 10) + (90\% * x * 5) = 3 + (x * 4.5)$ cycles
Solve for x: $3 + (x * 4.5) = 4$
$x * 4.5 = 1$
$x = 0.222 = 22\%$
22% is the mis-prediction rate, so the prediction rate must be 78% or more accurate to achieve a speedup over the set-associative cache.

6. **[10 Points] Register Renaming**

(a) What is the purpose of register renaming?

Answer: [2 points] Removes all false dependencies (WAW and WAR), leaving only true data dependencies.

Consider the following code for a processor with four logical registers (r0-r3):

```
add r0, r1 -> r2
ld [r2] -> r1
add r0, r1 -> r1
ld [r0] -> r3
```

(b) Rename the above instructions assuming **eight** physical registers (p0-p7) (by replacing all the logical registers with the appropriate physical registers). Assume that r0 is initially mapped to p0, r1 to p1, r2 to p2, and r3 to p3. The remainder of the physical registers are on the free list in the order p4, p5, p6, p7.

Answer: [4 points]

```
add p0, p1 -> p4
ld [p4] -> p5
add p0, p5 -> p6
ld [p0] -> p7
```

(c) Rename the above instructions assuming a **six** physical registers (p0-p5). Assume the same initial mapping as above; the free list contains p4 and p5 (in that order).

Answer: [2 points]

```
add p0, p1 -> p4
ld [p4] -> p5
add p0, p5 -> p2
ld [p0] -> p1
```

(d) How does this fewer number of physical registers concretely impact the execution of this code?

Answer: [2 points] The second `add` instruction cannot dispatch until the first `add` has committed. Likewise, the second `ld` instruction cannot dispatch until the first `ld` has committed.

7. **[9 Points] Store Sets**

(a) What problem is addressed by the Chrysos and Emer's *store sets* paper?

Answer: Memory scheduling (predicting when it is safe to issue a load instruction).

(b) In the paper, the predictor can be "wrong" in two different ways. What are those ways and what is the consequence of each?

Answer: [4 points] (1) It can be overly conservative by unnecessarily predicting a dependence, which delays execution, and (2) it can be overly aggressive by fail to predict an actual dependence, which causes a pipeline squash.

(c) As defined in the paper, what is a load's *store set*?

Answer: A static store is in a load's store set if any instance of that store ever bypassed a value (via the store queue) to any instance of the load.

(d) What key idea does the paper introduce to simplify the implementation of store sets? How does this simplify things?

Answer: [2 points] Each store can be in exactly one "store set" but to allow two different loads to depend on the same static store, a "store set" can have multiple loads in it. That is a "set" is really a merged set of loads and stores.
This simplifies things such that a load or store can do a simple direct lookup in the *Store Set ID Table* (SSIT) to obtain its store set ID. This ID is then used to find the most recent (if any) in-flight store in that set, so the load knows which store wait until it has executed.

(e) As described in the paper, the stores sets approach also constrains the execution order of stores. Why?

Answer: A load needs to wait until all stores in its store set have begun execution, so if those stores could execute in any order, it would need to explicitly need to know the identify of all those stores and remember it in the instruction queue. However, by forcing all the stores in a store set to execute in order, a load can just be set to depend on the most recent store in the store set.

8. **[9 Points] Cache Coherence**

(a) What is the main advantage of the "MSI" protocol over the "VI" cache coherence protocol?

Answer: "MSI" allows multiple processor to share read access to a block, reducing the number of coherence-induced cache misses.

(b) What is the main advantage of the "MESI" protocol over the "MSI" cache coherence protocol?

Answer: "MESI" reduces upgrade misses by giving non-shared blocks to processors in the write-able "exclusive clean" state.

(c) What is *false sharing*?

Answer: False sharing is when two or more processors are accessing different parts of the same cache block (and one of the processors is doing a write).

(d) How can software help prevent false sharing?

Answer: Software can help by aligning data structures and placing shared data in different cache blocks.

(e) What two issues with a *snooping* protocol does a *directory* cache coherence protocol fix? How does it fix each issue?

Answer: [4 points] (1) avoids a bus interconnect by using a point-to-point (switched) interconnect and using the directory as the point of ordering. (2) it avoids broadcast by tracking which processors are caching a block, and only sending requests to those actually caching the block.

(f) When considering the future of multicore systems, what is the biggest technical challenge?

Answer: Parallel programming (making them easier to program).

9. **[9 Points] Synchronization & Consistency**

(a) What is the primary disadvantage of coarse-grained locking?

Answer: Limits the parallelism because of lock contention.

(b) What are two disadvantages of employing fine-grained locking?

Answer: [2 points] (1) More difficult programming and (2) each lock acquire and release adds runtime overhead.

(c) What research proposal is aimed at mitigating the above issues with locking granularity?

Answer: Transactional memory

(d) Assuming a system with a standard coherence protocol, describe the advantage of a "test-and-test-and-set" spin lock versus a simple "test-and-set" spin lock implementation?

Answer: [2 points] A test-and-set lock repeatedly writes the block, causing it to repeatedly be fetched and invalidated (lots of traffic). A test-and-test-and-set lock adds a load before the "test-and-set" operation, allowing processors waiting for the lock to spin on a local read-only copy.

(e) Give two specific implementations reasons that a hardware implementer might want to design a chip that is not *sequentially consistent* (that is, allows the unexpected behaviors we discussed).

Answer: [2 points] (1) The hardware designers wants to avoid store miss latency via a store buffer, and (2) the designer might want to have a simpler out-of-order core by avoiding scanning the load queue (LQ) anytime a block is evicted from the cache.

(f) Even if the hardware does not reorder operations (that is, is sequential consistent), a programmer may still observe outcomes that are not sequentially consistent. Why?

Answer: The compiler can also reorder operations, causing non-SC executions.

10. **[8 Points] Dynamic Scheduling & Limits of ILP**. Consider the following C code:

```
double sum_square(double x[], int size) {
  double sum = 0.0;
  for(int i = 0; i < size; i++) {
    double value = x[i];
    double square = value * value;
    sum = sum + square;
  }
  return sum;
}
```

The loop body can be translated into six x86-like assembly instructions, one of which is a load:

```
.L2: movsd (%rdi,%rax) -> %xmm1    // load (-> value)
     mulsd %xmm1, %xmm1 -> %xmm1   // multiply (-> square)
     addsd %xmm1, %xmm0 -> %xmm0   // add (-> sum)
     addq  $8, %rax -> %rax        // add (-> i)
     cmpq  %rdx, %rax              // reads both registers, writes the flags
     jne   .L2                     // reads the flags
```

In this question, the loop executes an extremely large number of iterations (`size` is extremely large). Assume a fully idealized dynamically scheduled superscalar (perfect branch prediction, perfect infinitely wide fetch, rename, execute, commit and unbounded reorder buffer and instruction queues).

(a) If all instructions have a single-cycle latency, what is the maximum IPC? Why?

Answer: [2 points] 6 IPC (There is a serial dependency that limits us to one loop iteration per cycle, as the loop is 6 instructions, that is 6 IPC.)

(b) If the `mulsd` and `addsd` have a four-cycle latency and all other instructions have a single-cycle latency, what is the maximum IPC? Why?

Answer: [2 points] 1.5 IPC. (There is a serial dependency between loop iterations on the `sum` variable, which is calculated via the `addsd`. If this has a four-cycle latency, we can only do a loop iteration every four cycles, so $6/4 = 1.5$.

(c) If the `movsd` (the load) repeatedly misses in the cache, and thus has a 16-cycle latency and **all other instructions have a single-cycle latency**, what would the maximum IPC be? Why?

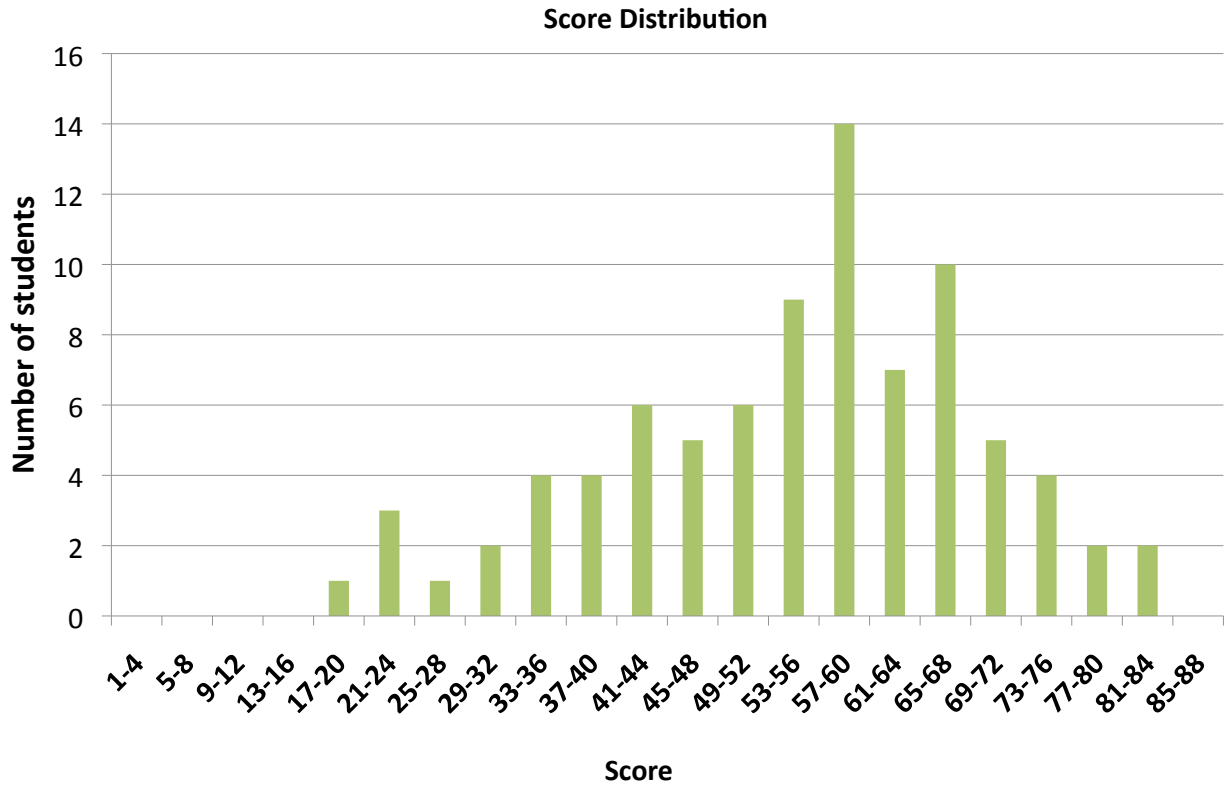
Answer: [2 points] 6 IPC (as all the loads are independent of each other, all of the load latency can be hidden with a large enough window!)

(d) For the 16-cycle load latency configuration above, **estimate** how large must the "instruction window" (say, size of the re-order buffer) be to achieve this IPC? Assume that instructions are magically fetched, decoded, and dispatched in a single cycle. Assume that instructions are cleared out of the reorder buffer (ROB) the same cycle the complete execution (assuming all older instructions have also completed, of course). **Explain your answer**. (Note: your reasoning is more important than the exact number of entries.)

Answer: [2 points] The exact size would likely need to be determined by simulation, but an estimate can be made. Even with a single-cycle load, to obtain the 6 IPC the processor probably needs around six loop iterations or so in the window (36 instructions). With a 16-cycle load, it needs to initiate the load 16 cycles before being "needed". At IPC of 6, that is a lead time of $16 * 6 = 96$ cycles.

— End of exam —

Distribution



Mean: 55 points (62%) — Median: 57 points (64%) — High: 82 (92%)