CIS501– Computer Architecture **Final Exam Solutions**
Prof. Martin **Tuesday, Dec. 22, 2009**

1. **[7 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) "The Alpha 21264 architecture is very dynamic."

   Answer: True

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

   Answer: False. More's law 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) A compiler optimization can increase performance yet hurt CPI.

   Answer: True

   (e) Clustering (such as that used by the Alpha 21264) is one approach for tackling the n^2 problem of dependence cross checking logic.

   Answer: False. Clustering tackles the n^2 bypassing and register file port problems, but does not address the dependence checking issue.

   (f) Assuming similar pipeline depths, high branch prediction accuracy is generally more important in a superscalar (multiple-issue) processor than in a scalar (single-issue) processor.

   Answer: True.

   (g) An easy way to exploit data parallelism in your programs is to call library code that has been optimized to use vector instructions.

   Answer: True

2. **[8 Points] Multiplying Performance**.

   (a) Consider a computation that consists of calculating the sum of thousands of integers. Using a simple single-cycle datapath as a starting point, what are four techniques or approaches we discussed that *each* can increase performance by a factor of four or more (that is, altogether these four techniques could result in a 256x speedup!)

   (Single word answers are sufficient.)

   Answer: (1) pipelining (2) instruction-level parallelism via superscalar execution, (3) data-level parallelism via vector instructions, and (4) coarse-grained parallelism using multicore. If just "multithreading" was given, it was worth one point (rather than two points), as hardware multithreading is unlikely to provide a 4x performance improvement.

3. **[11 Points] Performance & ISAs (Part 2)**.

An alternative way of accelerating the computation from the previous question is introducing a new three-input `ADD3` instruction to your favorite ISA. This new instruction operates on normal 64-bit registers only and performs the computation $A = B + C + D$.

(a) What impact–*if any*–would adding the `ADD3` instruction have on the following aspects of a pipelined datapath:

Answer: [6 Points]
- Instruction fetch: None
- Stall logic: Need to check for additional input register
- Bypassing: Additional bypassing datapaths for more input registers
- Register file: Additional read port
- Execution units: Extra logic to perform multiple additions
- Data cache: None

(b) What impact–*if any*–would adding this `ADD3` instruction have on the following aspects of a dynamically-scheduled pipeline?

Answer:
- The register rename logic would need to support renaming an additional register each cycle.
- The instruction wakeup logic would need to track more input registers for checking for instruction wakeup.
- None.

(c) An alternative way of implementing `ADD3` would be by using micro-ops. What would be the key advantage and disadvantage of implementing `ADD3` in this way?

Answer:[2 Points] Advantage: trades all the modifications throughout the pipeline with simpler changes to the decode stage. Disadvantage: the instruction wouldn't be any faster than just doing multiple normal two-input `ADD` instructions.

4. **[9 Points] Virtual Memory**.

Consider a system with a **42-bit virtual address** space and **4KB pages**. Consider a single-level, two-level, and three-level page table organization (with the indexing bits evenly divided among the levels). Each entry in the page table nodes is **4 bytes**. For a program that uses the entire first **8GB** (2^{33} **bytes)** of the virtual memory space, how many bytes are allocated for each level of the page table under each organization?

(a) How many bits in the page offset:

Answer: 12 bits (2^{12} is 4KB)

(b) How many bits in the virtual page number:

Answer: 30 bits, because $30 + 12 = 42$

(c) Single-level page table:

i. Bytes for first (only) level:

Answer: Independent of the address space actually used, a sigle-level page table has one 4-byte entry for all 2^{30} pages. As each entry is four bytes, this is 2^{32} bytes, which is 4GB!

(d) Two-level page table:

i. Bytes for first (root) level:

Answer: Each level is indexed using 15 bits, so independent of the address space actually used, the root has 2^{15} entries, which is 2^{17} bytes or 128KB.

ii. Bytes for second level:

Answer: For 8GB there are 2^{21} entries ($2^{33}/2^{12}$), which is 2^{23} bytes or 8MB.

(e) Three-level page table:

i. Bytes for first (root) level:

Answer: Each level is indexed using 10 bits, so independent of the address space actually used, the root has 2^{10} entries, which is 2^{12} or 4KB.

ii. Bytes for second level:

Answer: For 8GB there are 2^{11} entries ($2^{33}/2^{12}/2^{10}$) at the second level needed to point to the next layer, which is 2^{13} bytes or 8KB.

iii. Bytes for third level:

Answer: For 8GB there are 2^{21} entries ($2^{33}/2^{12}$), which is 2^{23} bytes or 8MB.

(f) In addition to a page table per process (which maps virtual to physical pages), the operating system also keeps a single "inverse" page table (which maps physical to virtual pages). Why?

Answer: When swapping out a physical page, the operating system needs to know which entry in the page tables it needs to invalidate. [In fact, as a single physical page can be used by multiple virtual addresses spaces, the OS data structure actually maps a single physical pages to zero or more virtual page numbers with their associated address space identifiers.]

5. **[9 Points] Caching**.

Consider the following progression of on-chip caches for three generations of chips (loosely modeled after the Alpha 21064, 21164, and 21264).

(a) Assume the first generation chip had a direct-mapped 8KB single-cycle first-level data cache. Assume t_{hit} for this cache is one cycle, t_{miss} is 100 cycles (to accesses off-chip memory), and the miss rate is 20%. What is the average memory access latency?

Answer: [2 points] $1 + (100 * 20\%) = 21$ cycles

(b) The second generation chip used the same direct-mapped 8KB single-cycle first-level data cache, but added a 96KB second-level cache on the chip. Assume the second-level cache has a 10-cycle hit latency. If an access misses in the second-level cache, it takes an additional 100 cycles to fetch the block from the off-chip memory. The second-level cache has a global miss rate of 4% of memory operations (which corresponds to a local miss rate of 20%). What is the average memory access latency?

Answer: [3 points]
$1 + (20\% * 10) + (4\% * 100) = 1 + 2 + 4 = 7$ cycles
— or —
$1 + (20\% * (10 + (20\% * 100))) =$
$1 + (20\% * (10 + 20)) =$
$1 + 20\% * 30 = 7$ cycles

(c) The third generation chip replaced the two-levels of on-chip caches with a single-level of cache: a set-associative 64KB cache. Assume this cache has a 5% miss rate and the same 100 cycle miss latency as above. Under what conditions does this new cache configuration have an average memory latency lower than the second generation configuration?

Answer: [2 points] $t_{hit} + (5\% * 100) = 7$ cycles
$t_{hit} <= 2$ cycles

(d) Notably, the second generation chip was an in-order pipeline, whereas the third generation was an out-of-order pipeline design. How might this difference have influenced the design of the memory hierarchies described above?

Answer: [2 points] In-order cores have a difficult time tolerating large load-to-use penalties. In contrast, an out-of-order core can look further down the instruction stream to find independent work, making it less sensitive to a higher load-to-use penalty. Thus, the third generation has a two-cycle load-to-use penalty, allowing for a larger set-associative cache.

6. **[10 Points] Pipelining**.

The standard pipeline discussed in class has five stages: fetch (F), decode (D), execute (X), memory (M), and writeback (W). The pipeline is fully bypassed and branches resolve at the end of the X stage. Consider the following pipelines, formed either by spreading the logic of a single pipeline stage across two stages (six-stage pipelines) or by combining pipeline stages (four-stage pipelines).

(a) What impact would splitting `F` into two stages (`F1` and `F2`) have on CPI?

Answer: Increase the branch misprediction penalty from two to three.

(b) What impact would splitting `D` into two stages (`D1` and `D2`) have on CPI?

Answer: Increase the branch misprediction penalty from two to three.

(c) What three distinct impacts would splitting `X` into two stages (`X1` and `X2`) have on CPI?

Answer: (1) Simple instructions like `adds` would now take two cycles to execute, introducing a use stall if the next instruction uses that register. (2) The branch prediction penalty would increase by one. (3) The load-use stall of loads would increase from one to two. Altogether, this would likely have a significant negative impact on CPI.

(d) What impact would splitting `M` into two stages (`M1` and `M2`) have on CPI?

Answer: Increase the load-use stall from one to two cycles.

(e) How would combining the `D` and `X` stages impact CPI?

Answer: Reduce the branch misprediction penalty from two to one.

(f) How would combining the `X` and `M` stages impact CPI?

Answer: Eliminate the load-to-use stall cycle.

(g) How could combining the `M` and `W` stages positively impact the clock frequency of the pipeline?

Answer: By combining these stages, it removes a bypassing path, reducing the size of the bypass muxes in the “X” stage. If the “X” stage was the slowest stage, this could improve the clock frequency of the pipeline.

(h) How might combining the `F` and `D` stages impact the implementation of branch prediction?

Answer: As decode is part of the fetch cycle, a branch target buffer (BTB) wouldn’t be needed to determine which instructions are branches or what their target would be.

7. **[10 Points] 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 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 addresses cause frequent and repeated conflicts in a small direct-mapped instruction cache. Instead of implementing a set-associative cache, you instead decide to increase the size of the direct-mapped cache. How large must the direct-mapped cache be before these two addresses no longer conflict with each other?

Answer: These two addresses have the lower 15 bits in common. Thus, the index+offset bits would need to at least 16 bits to map these two addresses to different sets. That would require a 64KB cache.

(b) The branch direction predictors that we discussed are all implicitly direct-mapped because they don't have tags. Your simulations of a simple "bimodal" predictor of two-bit saturating counters indicate that the branches at addresses A and B also map to the same entry in the branch predictor (and have conflicting biases). How many entries must the predictor contain to prevent these two branches from interfering (conflicting) with each other? How many total bytes?

Answer: As before, 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**.

(c) You have a brilliant idea: "Why not create a set-associative branch direction predictor?". Your simulations indicate that a predictor with just 2k 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 tag for each two-bit counter, instructions have 32-bit addresses). How large (in KBs) would a two-way set-associative 2k-entry tagged predictor be?

Answer: The 2k-entry predictor has 1024 sets, 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.)

(d) 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 and 1 byte is **2KB**. (As above, an LRU bit would add a bit to each entry (9 bits) for a total of 2.25KBs.)

(e) 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 omething 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 mispredicted (if the large table is getting a branch correct, there is no need to put that branch in the small predictor).

8. **[15 Points] Dynamic Scheduling**.

Consider the following seven-stage dynamically scheduled pipeline. It uses aggressive load scheduling, and it is similar to both the Alpha 21264 and the pipelined used in the lecture notes, but for simplicity the various execute, memory, and writeback stages have been coalesced into a single "X" stage. The stages are:

F	Fetch
R	Rename
D	Dispatch
I	Issue
RR	Register Read
X	Execute, Memory, and Writeback
C	Commit

(a) In the table below, fill in each empty box with the shorthand name of the stage (that is, `F`, `R`, `D`, `I`, `RR`, `X`, or `C`) in which the operation (column) is performed on the structure (rows):
[8 points]

Structure	Entry Allocated	Entry Read	Entries Searched	Entry Written	Entries Updated	Entry Deallocated
Register File	R	RR	—	X	—	C
Instruction Queue	D	—	I	D	I	X
Store Queue	D	C	X	X	—	C
Load Queue	D	—	X	X	—	C
Data Cache	—	X	—	C	—	—

(b) *Store queue writes* are triggered by (circle one): loads stores both

Answer: Stores

(c) *Store queue searches* are triggered by (circle one): loads stores both

Answer: Loads

(d) *Load queue writes* are triggered by (circle one): loads stores both

Answer: Loads

(e) *Load queue searches* are triggered by (circle one): loads stores both

Answer: Stores

(f) When using *conservative memory scheduling*, is the store queue necessary? Why or why not?

Answer: Yes, to allow forwarding of values from stores to loads within the window. Without a store queue, loads couldn't obtain values from earlier stores until all prior stores had committed (rather than just having executed).

(g) When using *conservative memory scheduling*, is the load queue necessary? Why or why not?

Answer: Not needed. The load queue is used to detect violations, which can't occur when loads are scheduled conservatively.

(h) How does the Alpha 21264 (which uses aggressive memory scheduling) avoid repeated memory ordering violations?

Answer: It uses a wait table to mark loads (as identified by their PC) that have caused violations in the past. The table is used to subsequent identify loads to schedule conservatively. The table is cleared periodically to allow it to forget and relearn (in case the dynamic behavior of the instructions have changed).

9. **[9 Points] Thread-Level Parallelism and Multicore**.

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

(c) What is the primary advantage of the "MSI" protocol over the simpler "VI" cache coherence protocol?

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

(d) What is the primary advantage of the "MESI" protocol over the simpler "MSI" cache coherence protocol?

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

(e) Describe *false sharing*. How can software help avoid false sharing?

Answer: [2 points] False sharing is when two processors are accessing different parts of the same cache block. Software can help by placing shared data in different cache blocks.

(f) What is a *memory fence* (also known as *memory barrier*)? Where are they typically used?

Answer: [2 points] Memory fences are instructions used to ensure ordering among instructions in systems with relaxed memory consistency models. They are used when writing synchronization operations, such as after a lock acquire and before a lock release.

10. **[8 Points] Limits of Performance**.

(a) Give two key reasons not to build a processor with a **1000-stage pipeline**:

Answer: (1) Implementation reason: pipeline latch overhead will dominate (beyond some point more stages doesn't help clock frequency) and it isn't realistic to chop up the execution of a single instruction into so many parts. (2) More fundamental reason: branch mispredictions penalties would be enormous, hurting CPI even with a state-of-the-art branch predictor.

(b) Give two key reasons not to build **1000-issue superscalar** processor:

Answer: (1) Implementation reason: the various n^2 operations (dependence checking, bypassing, register file ports) will hurt the clock frequency. (2) There isn't enough instruction-level parallelism (ILP) in applications to support such wide execution.

(c) Give two key reasons not to build a processor with a **1000-instruction scheduling window**:

Answer: (1) Implementation reason: scaling the size of the register file, issue queue, and load/store queues will all negatively impact the clock frequency of the processor. (2) More fundamental reason: branch mispredictions (even with a 99% branch prediction accuracy, branch mispredictions will occur every few hundred instructions on average).

(d) Give two key reasons not to build a **1000-core multicore**:

Answer: (1) Implementation reason: fitting 1000 cores on chip today would be infeasible (or require overly stripped down cores). (2) More fundamental reason: few algorithms can exploit such extreme levels of parallelism.

11. **[14 Points] General Themes & Concepts**.

(a) We have seen the general idea of "learning from the past to predict future behavior" employed in several different contexts. Give three distinct examples of prediction and the problem it is used to attack in each case:

Answer: [6 points] (1) Branch prediction to dramatically reduce control penalties, (2) way prediction to reduce the access latency of set-associative caches, (3) memory dependence prediction in dynamic scheduling for handling out-of-order memory operations (avoiding over-stalling or too much squashing), and (4) hardware prefetching to guess which blocks should be fetched to reduce memory stalls.

(b) Another general technique we have seen is "adding a level of indirection". Give two distinct examples of indirection and the problem it is used to attack in each case:

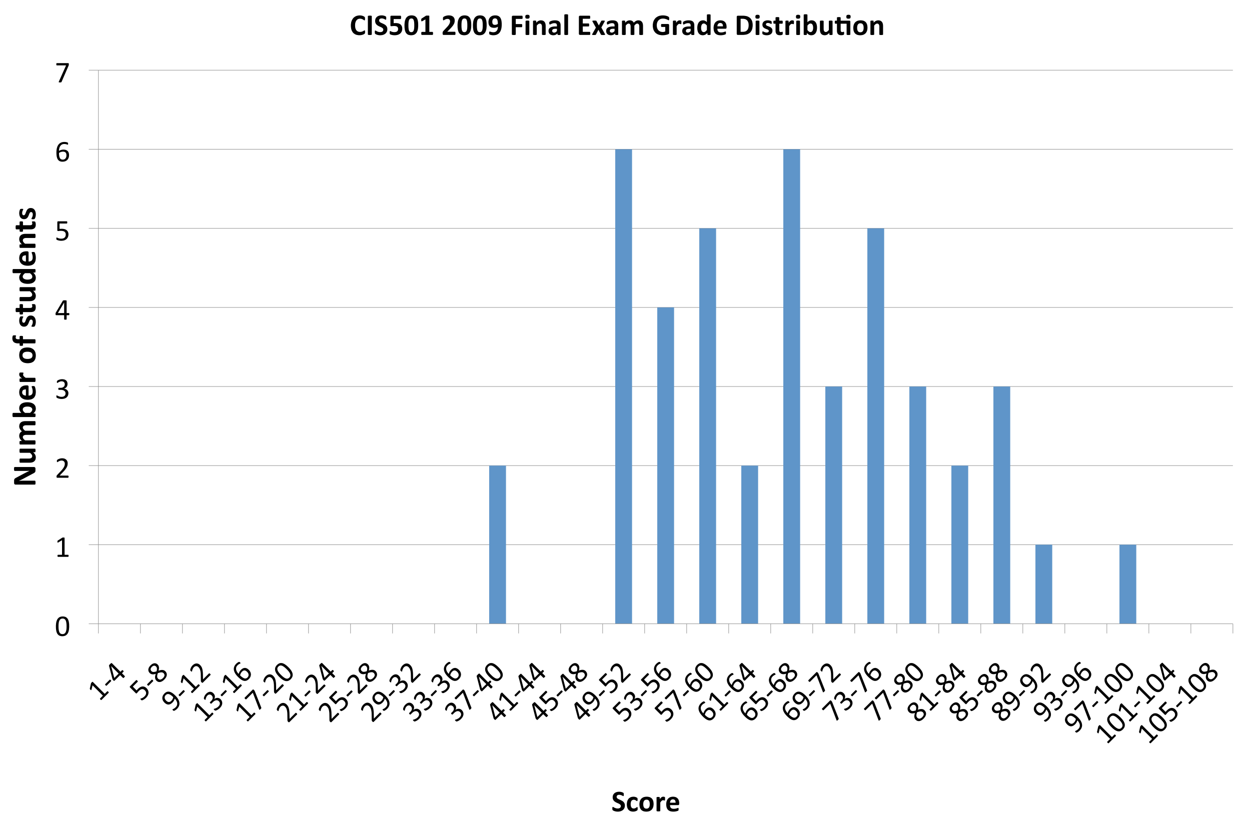
Answer: [4 points] (1) Virtual memory provides a level of indirection between virtual address and physical address to allow process isolation, protection, paging, etc. (2) Register renaming remaps architectural registers into physical registers to break false dependencies in dynamic scheduling. (3) Directory-based cache coherence avoids the scalability bottlenecks of broadcast snooping based protocols.

(c) Caching is another general technique. *Besides processor caches* for data and instructions, we discussed several other applications of the concept of caching or locality (spatial or temporal). Give two examples.

Answer: [4 points] (1) TLB, (2) virtual memory's swap space, (3) two-level page tables exploit spatial locality of the memory layout in the virtual address space. Mentioning next-block prefetching or stream buffers was also accepted.

— End of exam —

Distribution



Mean: 66 points (60%) — Median: 65 points (59%) — High: 100 (91%)