# CPSC 313, 04w Term 2— Midterm Exam 2 — Solutions

Date: March 11, 2005; Instructor: Mike Feeley

**1. (10 marks)** Short answers.

**1a.** Give an example of one important CISC feature that is normally not part of a RISC instruction set architecture. Very briefly explain why by referring back to the y86 pipeline implementation.

> In a RISC, arithmetic instructions can't access memory. The reason is that the ALU is needed to calculate the effective address of memory instructions and so isn't available to do anything else. Also okay: in RISC instructions are uniform size and conditional branches don't use op codes.

**1b.** Write a sequence of two C statements that are causally dependent on each other and show what causes the dependency.

**1c.** Define temporal locality and say why it important for caching?

> Temporal locality exists when the same memory address is access multiple times in a short interval of time. Its important for caching, because if a cache stores recently accessed items, subsequent accesses to those same items will hit in the cache.

**1d.** Define instruction-level parallelism. Is it important for pipelining or caching? Why?

> Instruction-level parallelism exists when instructions can be run in parallel or out of order due to the fact that the dependencies among them do not impose an order. It is important for pipelining, because for a pipeline to be effective, it must work on multiple instructions at once, partially in parallel.

**1e.** Give one advantage and one disadvantage of a direct-mapped cache compared to an associative cache (i.e., fully-associative or set-associative).

> Advantage: faster. Disadvantage: more conflict misses.

**2. (6 marks)** Pipelines.

**2a.** Carefully explain why one would expect that changing a processor implementation from sequential to pipelined would improve the processor's performance.

> The sequential circuitry in each pipeline stage will have less gate-propagation delay than the original sequential circuit and thus transistors will be used more effectively and the clock can run faster. In a sequential circuit with large propagational delay, the transistors close to the inputs spend must of their time just holding a value waiting for signal to get through the rest of the circuit. Similarly transistors distant from the inputs spend most of their time waiting for the signal to arrive. This time the transistors spend waiting is wasted time and the more waste there is the lower the overall throughput.

**2b.** List all of the reasons you can think of why adding more stages to a pipelined implementation might not necessarily improve performance.

> There are two main reasons. First, adding a pipeline stage adds a fixed propagational delay for the registers that sit between stages. When the delay of each stage is close to this register delay, then adding more stages provides little benefit. Second, deeper pipelines are hard to keep filled. They may require more instruction level parallelism than exists in most programs and thus the number of pipeline stalls starts to increase with pipeline depth (i.e., the CPI increases).

**2c.** Does it make sense for Intel to report a single throughput measurement for the P4 Prescott? Why or why not?

> No, because the actual throughput of a pipelined processor depends on the workload. Some workloads will have little instruction-level parallelism and thus many pipeline stalls. Others will do better. Throughput depends not just on the cycle time, but also on the number of pipeline stalls per instruction.

**3. (3 marks)** Object files and linking.

**3a.** What is the benefit of separate compilation and what problem does it cause that linking solves?

> The benefit is that it makes it easier to write larger programs. It allows you to compile just the modules that change during the software development cycle, not everything, and is thus faster. It also allows you to use modules for which you do not have the source code and thus aids software resuse (e.g., libraries). The problem it creates is that the compiler will be unable to resolve references to symbols defined in other modules.

**3b.** What is the difference between a *relocatable* and an *executable* object file? How do they relate to the first part of this question?

> The compiler generates *relocatable* object files. These files have unresolved references and do not yet have a runtime virtual-memory address assigned. The linker generates *executable* object files from collections of relocatable files. Executable files have no unresolved references, have runtime virtual-memory addresses assigned to all code and data and are thus read to load into memory and execute in the CPU.

**4. (8 marks)** You are responsible for designing the next-generation y86 cpu. The design challenge you are confronted with is that the execute phase has roughly twice the delay as any of the other pipeline phases. You decided to split this stage in two, so that the pipeline now has six stages and none of the results of the execute phase are available until the end of the second execute stage. Answer the following questions.

**4a.** What problem did the slow execute phase cause and how did splitting it in two solve this problem?

> The problem is that the clock can only be as fast as the slowest pipeline stage and thus the slow execute stage was constraining throughput. By splitting it into two stages, the clock speed can be doubled and thus throughput may double.

**4b.** Recall that a data hazard exists in the following scenario (among others):

```
addl    %eax, %ebx
rrmovl  %ebx, %ecx
```

Briefly explain how the current, 5-stage, y86 implementation handles this particular hazard and state whether doing so requires any pipeline stalls or bubbles.

> Register values are forwarded from the output of the execute phase to the decode phase. In this case, the second instruction will read the value of `%ebx` from the forwarding signals instead of from the register file. No pipeline stall is required in the 5-stage pipeline to handle this hazard.

**4c.** Your 6-stage pipeline can not handle the hazard illustrated by this example in the same way as the 5-stage pipeline. Carefully explain why not. Are additional pipeline stalls or bubbles needed to correctly handle the two-instruction sequence above (assuming no other change to the design)? How many?

In the 6-stage pipeline, the new value of %ebx isn't known until the end of the second execute stage, which is two stages after the decode stage. Thus the second instruction will be in the first execute stage, not the decode stage, when the new value becomes available on the forwarding path. If we do nothing else, a single-cycle stall, and corresponding bubble, is required between the two instructions to keep the second instruction in the decode phase until the first has finished the second execute phase.

**4d.** Can you modify the design to improve the situation for the two-instruction sequence above? Give a brief, high-level outline of your solution (implementation details are not necessary).

The rrmovl instruction doesn't actually need the value of %ebx until the write-back stage, at which point the first instruction has been retired. So, the solution is to add forwarding signals from the end the second execute phase back to the end of the first execute phase. The second instruction will thus read the wrong value from the register file in the decode phase, but then get the right value in the execute phase, in plenty of time for it to write the correct value back to %ecx in the register file.

**4e.** Can you use the same technique for the following two-instruction sequence?

```
addl    %eax, %ebx
addl    %ebx, %ecx
```

Carefully explain why or why not.

In this case, the second instruction needs the value of %ebx at the beginning of the first execute phase, but the value isn't generated until the first instruction finishes the second execute stage (i.e., when the second instruction is at the *end* of the first stage: too late to be useful). A stall is thus required in this case.

**5. (7 marks)** Consider the following y86 program and the standard y86 pipeline implementation (i.e., PIPE) described in class and in the textbook.

```
start:      rrmovl  %eax, %ebx
            rmmovl  %ebx, 8(%ebp)
            mrmovl  12(%ebp), %ebx
            addl    %ebx, %ecx          # [1]
            call    foo
            addl    %eax, %ecx
            jne     notzero             # [2]
            irmovl  $1, %ecx
notzero:    mrmovl  %ecx, 12(%ebp)
            ...
            hlt
foo:        irmovl  $100, %eax
            ret                         # [3]
```

**5a.** Annotate this code to indicate every place that a pipeline stall occurs and explain why it occurs in each case.

1. Load-use hazard stalls this instruction in the decode phase for one cycle, introducing one bubble.

2. Possible two bubbles if branch is not taken. However, branch probably is usually taken (it is only not taken when %exc is zero), so there's probably not a problem here.

3. Return hazard stalls two cycles in fetch phase waiting to read return address from memory.

**5b.** Modify this piece of code to eliminate as many pipeline stalls as you can, but without changing its semantics (i.e., its meaning). You can view this code in complete isolation from any other code; that is, you can assume that no other code calls into this code. Write your modified code below, **using comments to highlight and explain every change you have made**.

```
start:      rrmovl  %eax, %edx          # [1]
            mrmovl  12(%ebp), %ebx
            rmmovl  %edx, 8(%ebp)       # [2]
            addl    %ebx, %ecx          # [3]
            irmovl  $100, %eax          # [4]
            addl    %eax, %ecx
            jne     notzero             # [5]
            irmovl  $1, %ecx
notzero:    mrmovl  %ecx, 12(%ebp)
            ...
            hlt
```

1. Renamed `%ebx` to `%edx` to avoid anti-dependency between next two instructions so that they can be swapped.
2. Swapped with previous instruction to fill the load-use delay slot and thus eliminate the pipeline stall that used to occur here.
3. Forwarding logic can now void a stall here, because this use of `%ecx` is two instructions away from the load of `%ecx` from memory.
4. Inline the procedure `foo` to avoid the return hazard.
5. Nothing to do here, assuming the `%ecx` is usually not zero.

**6. (4 marks)** When caches help.

**6a.** What feature of a cache is designed to exploit temporal locality?

The fact that the cache stores recently accessed items for a while.

**6b.** What feature of a cache is designed to exploit spatial locality?

The fact that blocks are bigger than 4 bytes.

**6c.** Can a cache provide any benefit to a program that has neither temporal nor spatial locality? How?

Yes if the hardware provides a mechanism for prefetching and either the workload has a predictable access pattern or the application program can predict future accesses sufficiently far in the future to hide memory-access latency.

**7. (6 marks)** You work for Intel's cache design group and are having a bad day. You've just finished the design of the cache you are responsible for. Now your boss is pushing you to make one more change and you don't want to. For each of the following suggested changes give a good reason why the change might make things worse, assuming that the total size of the cache (i.e., the total amount of data it can store) must remain the same.

**7a.** Increase the block size.

Decreases cache's ability to capture temporal locality, because the cache stores fewer total blocks.

**7b.** Decrease the block size.

Decreases cache's ability to capture spatial locality.

**7c.** Decrease the set associativity (e.g., from 8-way to 2-way).

Increases potential for conflict misses.

**7d.** Increase the set associativity (e.g., from 8-way to 32-way).

Slows cache-hit time.

**7e.** Changing from write-through to write-back.

Slows write-miss handling, because the write must delay until the accessed block is fetched from memory into the cache.

**8. (6 marks)** Consider a small 64-B, 2-way set associative, write through, write allocate cache with 8-B blocks and using LRU replacement. The highest-order address bits are assigned to the tag.

**8a.** If addresses are 32 bits long, how many address bits are needed for each of the following:

- block offset (b) = 3
- set index (s) = 1
- tag (t) = 28

**8b.** The cache is initially empty and then it sees the following sequence of accesses. For each access, indicate whether it is a hit or miss. If its a miss, give the type of miss.

- read 0x10 = 01 0 000: cold miss
- read 0x04 = 00 0 100: cold miss
- read 0x14 = 01 0 100: hit
- write 0x08 = 00 1 000: cold miss
- read 0x0c = 00 1 100: hit
- read 0x24 = 10 0 100: cold miss
- read 0x10 = 01 0 000: conflict miss