CPSC 313, 05w Term 1— Midterm 2 — Solutions

Date: November 9, 2005; Instructor: Mike Feeley

1. (10 marks) Short answers.

1a. Briefly explain how an n-channel MOSFET transistor closes the circuit between its source and drain when its gate voltage is high.

> The high gate voltage produces a field in the positive-doped silicon substrate that draws electrons toward the gate. The electrons pile up along the glass insulator that separates the gate from the substrate and they form an electron-rich channel between the negative-doped source and drain wells. There is now an electron rich path between source and drain and thus current can flow between them.

1b. Can an general-purpose processor (just the processor itself) be design using only combinational logic? Why or why not?

> No, because combination logic along is finite state and thus is unable to express all forms of computation, some of which are require infinite state (i.e., memories and unbounded time). Combination logic can implement any Boolean function or any finite automata, but you need a sequential circuit to implement a Turing machine or a general-purpose program.

1c. Give one advantage the y86 (and/or a RISC processor) gets from confining memory access to special instructions that do nothing else but load from and store to memory (i.e., `mrmovl` and `rmmovl`.

> ALU instructions have predictable performance, because they do not access memory; memory access has variable performance depending on whether target is in cache or not. **OR** Can use ALU to compute memory address for mrmovl and rmmovl, because they don't need the ALU to compute anything else, unlike the ALU instructions (e.g., addl etc.).

1d. Explain the difference between a pipeline stall and a pipeline bubble?

> A pipeline stall is when a pipeline stage's input memory remains the same from one cycle to the next. A pipeline bubble is when a pipeline stage's input memory changes to that of a *nop* instruction, i.e., an instruction that does nothing. On any cycle when an instruction stalls in one stage, a bubble is injected into the subsequent stage.

1e. Briefly explain what a control hazard is, what stage of the y86 pipeline it affects and how.

> A control hazard exists when the fetch stage is unable to determine the address of the next instruction to execute. In y86 this problem occurs for jumps and procedure return.

2. (6 marks) Pipelines.

2a. Carefully explain how pipelines improve performance.

> By dividing a combination circuit into multiple smaller pipeline-stage circuits connected in sequence, pipelining seeks to reduce the maximum gate delay through any pipeline stage. The clock frequency is limited by the maximum delay of the slowest stage. Reducing the delay thus allows the clock to pulse more frequently and, if the pipeline is able to retire one instruction each cycle, the throughput (i.e, instructions completed per unit time) improves.

2b. Give one reason why adding pipeline stages may **reduce** processor performance (i.e., make it worse).

If increasing the number of stages also increases the number of pipeline bubbles.

2c. Give a formula that computes the maximum clock frequency (units: millions of cycles / second) of a five-stage pipeline with pipeline-stage delays (including memory delays) of 50 ps, 75 ps, 100 ps, 50 ps and 25 ps (ps = 1 trillionth of a second).

$tp = \frac{1\ cycle}{100\ ps} \times \frac{10^{12}\ ps}{s} = \frac{10^{10}\ cycles}{s} = 10\ GHz$

3. (7 marks) Each of the following pieces of code may contain a dependency. Indicate whether it does. If there is a dependency, give the **name of the dependency** (i.e., causal, anti or output) and **the register(s) involved**. Then indicate whether the code can be re-written to eliminate the dependency (without changing its meaning) and if so, modify the code to do so. You can assume that the remainder of the program will also be modified appropriately.

3a.
```
addl    %eax, %ebx
addl    %ebx, %ecx
```

Dependency ("NO" or name and register(s)): Yes; causal dependency with `%ebx`

Re-write possible ("YES" or "NO"; if YES, modify the code): No

3b.
```
addl    %eax, %ebx
irmovl  $1, %ebx
```

Dependency ("NO" or name and register(s)): Yes; output dependency with `%ebx`

Re-write possible ("YES" or "NO"; if YES, modify the code):

Yes; rename one of the destination registers.

3c.
```
addl    %eax, %ecx
addl    %ebx, %eax
```

Dependency ("NO" or name and register(s)): Yes; anti dependency with `%eax`

Re-write possible ("YES" or "NO"; if YES, modify the code):

Yes; rename either destination of first or source of second instruction.

4. (7 marks) Consider the y86 PIPE discussed in class. For each of the following indicate whether a hazard exists, how it is handled (very briefly) and under what circumstances y86 introduces a pipeline bubble.

4a.
```
addl    %eax, %ebx
addl    %ebx, %ecx
```

Hazard ("YES" or "NO") and how it is handled:

Yes a data hazard exists with `%ebx`. Handled by *data forwarding* of destination-register values from execute, memory and write-back stages to the decode stage.

Under what circumstances, if any, does y-86 introduce pipeline bubbles and how many:

Never a bubble.

4b.
```
jle     target
```

Hazard ("YES" or "NO") and how it is handled:

Yes a control hazard exists. Handled by predicting branch is taken.

Under what circumstances, if any, does y-86 introduce pipeline bubbles and how many:

> If execute phase determines branch should not be taken, instructions in decode and fetch phase that should never have executed turn into bubbles on next clock pulse.

4c. `ret`

Hazard ("YES" or "NO") and how it is handled:

> Yes a control hazard exists. Handled by stalling three cycles until `ret` reaches memory phase, when the address of the next instruction is read from memory.

Under what circumstances, if any, does y-86 introduce pipeline bubbles and how many:

> Always three bubbles following `ret`.

5. **(7 marks)** Consider the y86 PIPE implementation discussed in class (diagram attached to exam). The load-use hazard occurs when an instruction that reads a value from memory is immediately followed by an instruction uses that value.

5a. Give an example of the load-use hazard.

5b. Carefully explain how y86-PIPE resolves this dependency and why this solution is different from that of causal dependencies that don't involve memory.

> If *use* immediately follows the *load*, y86-PIPE stalls the second instruction (the use) in the decode phase one cycle (so it spends two cycles there in total), until the first instruction (the load) reaches memory phase. At this point the value the first instruction reads from memory (in `m_valM`) is forwarded back to the decode phase. If load and use are separated by one other instruction, there is no stall/bubble, but `W_valM` is forwarded to decode from write-back. This stall is not required, however, if the first instruction sets a register value in any other way, because in all of those cases, the new value of the register is know by the end of the *execute* phase, in time for it to be forwarded to *decode* with no stall.

5c. Notice that `pushl` (*IPUSHL*) and `rmmovl` (*IRMMOVL*) do not use the register value they read in the execute phase; their first use of this value is in the memory phase. It should thus be possible to handle load-use hazards involving these instructions without stalling. Explain why and very briefly outline a solution (the next question asks for more details)

.

> These two instructions (and `mmmovl`) do not need the value of their source operand in the execute phase. It should thus be possible to allow them to proceed into the execute phase, without stalling, even if they immediately follow a memory read that loads a new value into their source operand register. In this case, the use instructions will have the wrong value for its source operand during the execute phase, but the correct value can be forward from the memory phase at the end of the execute phase.

6. **(7 marks)** This question continues the previous one. One solution that avoids stalling IPUSHL and IRMMOVL is to introduce a forwarding circuit in the execute stage that picks the correct `valA` to send to the memory stage for these two instructions. Give the HCL description of that circuit (refer to the diagram on last page of exam).

```
int e_valA = [
```

7. **(6 marks)** Memory.

7a. By referencing properties of SRAM and DRAM describe why a memory hierarchy is necessary.

SRAM is much faster to access than DRAM, but SRAM bits use more transistors and power than DRAM. Thus, the hierarchy uses relatively small caches implemented using SRAM and large main-memories implemented using DRAM.

7b. Carefully explain the relationship between locality and caching.

Caching is a technique designed to exploit locality to improve performance. If a workload has not locality it will receive no benefit from caching.

7c. Carefully explain the difference between temporal and spatial locality.

Temporal locality is when a program accesses the same memory location multiple times within a relatively brief interval. Spatial locality is when a program accesses memory locations with nearby addresses within a relatively brief interval.