University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Fall 1999 John Kubiatowicz

Midterm II

November 17, 1999
CS152 Computer Architecture and Engineering

Your Name:	
SID Number:	
Discussion Section:	

Problem	**Possible**	**Score**
1	25	
2	25	
3	25	
4	25	
Total		

[This page left for π]

3.141592653589793238462643383279502884197169399375105820974944

Problem 1: Memory Hierarchy

Problem 1a: Assume that we have a 32-bit processor (with 32-bit words) and that this processor is byte-addressed (i.e. addresses specify bytes). Suppose that it has a 512-byte cache that is two-way set-associative, has 4-word cache lines, and uses LRU replacement. Split the 32-bit address into "tag", "index", and "cache-line offset" pieces. Which address bits comprise each piece?

tag:
index:
cache-line offset:

Problem 1b: How many sets does this cache have? Explain.

Problem 1c: Draw a block diagram for this cache. Show a 32-bit address coming into the diagram and a 32-bit data result and "Hit" signal coming out. Include, all of the comparators in the system and any muxes as well. Include the data storage memories (indexed by the "Index"), the tag matching logic, and any muxes. You can indicate RAM with a simple block, but make sure to label address widths and data widths. Make sure to label the function of various blocks and the width of any buses.

[This page left for scratch]

Problem 1d: Below is a series of memory read references set to the cache from part (a). Assume that the cache is initially empty and classify each memory references as a hit or a miss. Identify each miss as either **compulsory, conflict, or capacity.** One example is shown. *Hint: start by splitting the address into components. Show your work.*

Address	Hit/Miss?	Miss Type?
0x300	Miss	Compulsory
0x1BC		
0x206		
0x109		
0x308		
0x1A1		
0x1B1		
0x2AE		
0x3B2		
0x10C		
0x205		
0x301		
0x3AE		
0x1A8		
0x3A1		
0x1BA		

Problem 1e: Calculate the miss rate and hit rate.

[This page intentionally left blank]

Problem 1f: You have a 500 MHz processor with 2-levels of cache, 1 level of DRAM, and a DISK for virtual memory. Assume that it has a Harvard architecture (separate instruction and data cache at level 1). Assume that the memory system has the following parameters:

Component	Hit Time	Miss Rate	Block Size
First-Level Cache	1 cycle	4% Data 1% Instructions	64 bytes
Second-Level Cache	20 cycles + 1 cycle/64bits	2%	128 bytes
DRAM	100ns+ 25ns/8 bytes	1%	16K bytes
DISK	50ms + 20ns/byte	0%	16K bytes

Finally, assume that there is a TLB that misses 0.1% of the time on data (doesn't miss on instructions) and which has a fill penalty of 40 cycles. What is the average memory access time (AMAT) for Instructions? For Data (assume all reads)?

Problem 1g: Suppose that we measure the following instruction mix for benchmark "X":

Loads: 20%, Stores: 15%, Integer: 30%, Floating-Point: 15% Branches: 20%

Assume that we have a single-issue processor with a minimum CPI of 1.0. Assume that we have a branch predictor that is correct 95% of the time, and that an incorrect prediction costs 3 cycles. Finally, assume that data hazards cause an average penalty of 0.7 cycles for floating point operations. Integer operations run at maximum throughput. What is the average CPI of Benchmark X, including memory misses (from part g)?

Problem #2: Superpipelining

Suppose that we have single-issue, *in-order* pipeline with one fetch stage, one decode stage, multiple execution stages (which include memory access) and a singe write-back stage. Assume that it has the following execution latencies (i.e. the number of stages that it takes to compute a value): **multf** (5 cycles), **addf** (3 cycles), **divf** (2 cycles), integer ops (1 cycle). Assume full bypassing and two cycles to perform memory accesses, i.e. loads and stores take a total of 3 cycles to execute (including address computation). Finally, branch conditions are computed by the first execution stage (integer execution unit).

Problem 2a:
Assume that this pipeline consists of a *single linear sequence* of stages in which later stages serve as no-ops for shorter operations. Draw each stage of the pipeline as a box (no internal details) and name each of the stages. Describe what is computed in each stage and show *all* of the bypass paths (as arrows between stages). *Your goal is to design a pipeline which never stalls unless a value is not ready.* Label each of these arrows with the types of instructions that will forward their results along these paths (i.e. use “M” for **multf,** “D” for **divf,** “A” for **addf,** “I” for integer operations). [*Hint:* be careful to optimize for information feeding into store instructions!]

Problem 2b:
How many extra instructions are required between each of these instruction combinations to avoid stalls (i.e. assume that the second instruction uses a value from the first). Be careful!

Between a **divf** and an **store:**

Between a **load** and a **multf:**

Between two integer instructions:

Between a **multf** and an **addf:**

Between an **addf** and a **divf:**

Between an integer op and a **store:**

Problem 2c:
How many branch delay slots does this machine have? Explain.

Probem 2d:
Could branch prediction increase the performance of this pipeline? Why or why not?

Problem 2e:
In the 5-stage pipeline that we discussed in class, a load into a register followed by an immediate store of that register to memory would not require any stalls, i.e. the following sequence could run *without* stalls:

```
lw   r4, 0(r2)
sw   r4, 0(r3)
```

Explain why this was true for the 5-stage pipeline.

Problem 2f:
Is this still true for the superpipelined processor? Explain.

Problem #3: Fixing the loops

For this problem, assume that we have a superpipelined architecture like that in problem (2) with the following use latencies (these are not the right answers for problem #2b!):

Between a **multf** and an **addf:**	**3 insts**	Between a **load** and a **multf/divf:**	**2 insts**
Between an **addf** and a **divf:**	**1 insts**	Between a **divf** and a **store:**	**7 insts**
Between an int op and a **store:**	**0 insts**	Number of branch delay slots:	**1 insts**

Consider the following loop which performs a restricted rotation and projection operation. The array based at register **r10** contains pairs of double-precision (64-bit) values which represent x,z coordinates. The array based at register **r20** receives a projected coordinate along the observer's horizontal direction:

```
loop:   ldf     $F20, 0($r10)
        multf   $F6,  $F20, $F1
        addf    $F12, $F6, $F2
        ldf     $F10, 8($r10)
        divf    $F13, $F12, $F10
        stf     0($r20), $F13
        addi    $r10, $r10,#16
        addi    $r20, $r20, #8
        subi    $r1, $r1, #1
        bne     $r1, $zero, loop
          nop
```

Problem 3a: How many cycles does this loop take per iteration? Indicate stalls in the above code by labeling each of them with a number of cycles of stall:

Problem 3b: Reschedule this code to run with as few cycles per iteration as possible. Do not unroll it or software pipeline it. How many cycles do you get per iteration of the loop now?

Problem 3c: Unroll the loop once and schedule it to run with as few cycles as possible per iteration of the original loop. How many cycles do you get per iteration now?

Problem 3d: Your loop in (3c) will not run without stalls. Without going to the trouble to unroll further, what is the minimum number of times that you would have to unroll this loop to avoid stalls? Explain. How many cycles would you get per iteration then?

Problem 3e: Software pipeline the original loop to avoid stalls. Overlap 5 different iterations. What is the average number of cycles per iteration? Your code should have no more than one copy of the original instructions. Ignore startup and exit code.

Extra Credit (Problem 3X):
Assume that you have a Tomasulo architecture with functional units of the same execution latency (number of cycles) as our deeply pipelined processor (*be careful to adjust use latencies to get number of execution cycles!).* Assume that it issues one instruction per cycle and has an unpipelined divider with a small number of reservation stations. Suppose the other functional units are duplicated with many reservation stations and that there are many CDBs. . What is the minimum number of divide reservation stations to achieve one instruction per cycle with the optimized code of (3b)? Show your work. *[hint: assume that the maximum issue rate is sustained and look at the scheduling of a single iteration]*

Problem 4: Short Answers

Problem 4a: Give a simple definition of precise interrupts/exceptions:

Problem 4b: Explain how the presence of delayed branches complicates the description of a precise exception point (*Hint: what if there is a divide instruction in a delay slot that gets a divide by zero exception)?*

Problem 4c: Explain the relationship between support for precise exceptions and support for branch prediction. What hardware structure supports both of these mechanisms in a modern out-of-order pipeline?

Problem 4d: Explain how pipelining can save power (and energy) for multimedia (streaming) applications:

Problem 4e: A PalmPilot is a portable computing device that holds calendars and addresses. It has a micro-power mode that stops the clock and shuts down power to the processor when it is idle. Suppose that it also recognized when the battery was getting low and ran the clock at lower than normal speed during busy periods. Would this extend battery life? Why or why not?

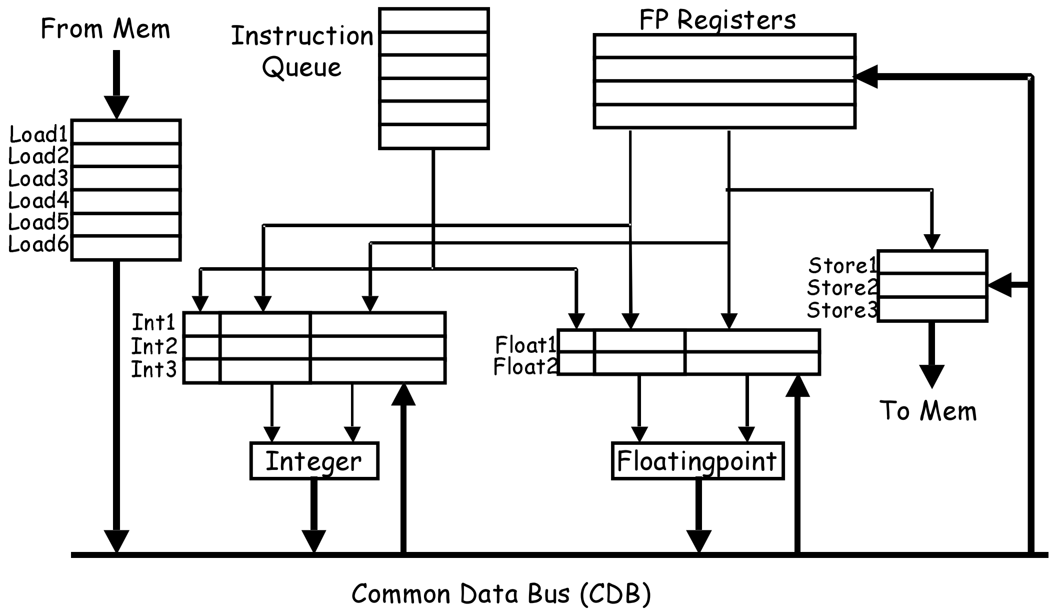

Figure 1: A basic Tomasulo architecture

Problem 4f: The Tomasulo architecture (shown above) replaces a normal 5-stage pipeline with 4 stages: *Fetch, Issue, Execute, and Writeback.* One of its strengths is that it is able to *Execute* instructions in a different order than the programmer originally specified. The simplest version of this architecture also performs Writeback out-of-order as well. However, the Fetch and Issue stages of the Tomasulo architecture are always handled in program order. Why?

Problem 4g: Pipelined architectures have three different types of data hazards with respect to registers. Name and define them. For *each* type, give a short code sequence that illustrates the hazard and describe how a Tomasulo architecture removes this hazard.

Problem 4h: What is register renaming, why is it desirable, and how is it accomplished in the Tomasulo architecture?

Problem 4i: Why does a Tomasulo architecture need branch prediction?