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

Fall 1998 J. D. Kubiatowicz

Quiz 1
October 7, 1998
CS252 Graduate Computer Architecture

You are allowed to use a calculator and one 8.5" x 11" double-sided page of notes. Show your work on all problems. If you find a problem unclear or underspecified, please ask for clarification of the assumptions. If you make assumptions not listed in the problem, please state them clearly in your solutions. Good luck!

Your name:	
SID number:	
Email address:	

1	/20
2	/20
3	/40
4	/20
Total	**/100**

Question 1: Being Precise

This problem explores some of the issues involved in supporting precise interrupts in a simple 5-stage DLX-like pipeline.

(a) Give a simple definition of precise interrupts/exceptions.

(b) Why are precise interrupts/exceptions useful? Give 3 examples.

(c) Draw a simple block diagram of a 5-stage DLX pipeline, including bypass paths to the execution unit (ALU). Do not include the sign-extension hardware or the hardware needed to optimize conditional branches (as presented in class). Be sure to label the pipeline stages.

Question 1 (continued)

(d) Consider the following simple instruction sequence:

```
(1)  lw    r2, 10(r1)        ; instruction 1
(2)  add   r4, r3, r1        ; instruction 2
(3)  sub   r5, r1, r10       ; instruction 3
```

Assume that this sequence completes correctly. Using single letters to represent each pipeline stage (F, D, X, M, W), show the time evolution of this sequence (lining up the phases of each instruction):

(e) Now assume the following exceptions occur: instruction 1 gets a data TLB fault; instruction 2 gets an overflow; and instruction 3 causes an instruction TLB fault. Produce a diagram as in part (d), with the faulting stages labeled.

(f) Which exception happens **first** in **time**?

(g) Which exception must be **taken** in order to have a precise exception?

Question 1 (continued)

(h) What sort of hardware support would be necessary to reorder the exceptions so as to get precise interrupts in the 5-stage pipeline (that is, so that the exception identified in part (g) would happen instead of the exception in part (f))?

Be sure to explain how your hardware prevents inconsistent register and memory state caused by instructions that started their execution before the exception, but that must be squashed after the exception happens.

Question 2: Being Precise with Tomasulo

(a) Draw a block diagram of the major parts of the datapath of an **out-of-order** processor using **Tomasulo's algorithm**, that supports **precise** interrupts.

Assume that the following are true:

1. There is an adder that can support 3 simultaneous requests
2. There is a multiplier that can support 2 simultaneous requests
3. There can be 6 outstanding loads

Question 2 (continued)

(b) Explain how this architecture supports precise interrupts.

(c) Describe the hardware needed to speculate load-store dependencies (hint: this is required to figure out when a load-store dependency is violated). Describe briefly how this hardware works and what it means to speculate dependencies. Contrast this with the "original" solution of no speculation. Do not assume the sophistication of store-sets to selectively block requests.

Question 3: Speeding up the Loops

For the following problem, assume an in-order DLX-style pipelined architecture that has functional units that take the following number of execution cycles:

1. Floating-point multiplier: **4 cycles**
2. Floating-point adder: **2 cycles**
3. Integer operations: **1 cycle**

Assume as well there is **one branch delay slot**, that there is no delay between integer operations and dependent branch instructions, and that the load-use latency is **2 cycles**. Assume all functional units are fully pipelined and fully bypassed.

The following code computes a dot product. Assume that `r1` and `r2` contain addresses of arrays of floating-point numbers, and `r3` contains the length of the arrays (in elements). Assume that `r4` is initialized to zero. Then, the dot product can be computed as follows:

```
loop:  ld    f5, 0(r1)    ; load element from first array
       ld    f6, 0(r2)    ; load element from second array
       multd f7, f5, f6   ; multiply elements
       addd  f4, f4, f7   ; add elements to accumulator in f4
       addi  r1, r1, #8   ; increment pointers
       addi  r2, r2, #8
       subi  r3, r3, #1   ; decrement element count
       bnez  r3, loop     ; continue until all elements done
        nop               ; (branch delay slot)
```

(a) How many cycles does each iteration take, without arranging the code?

(b) What is the lowest number of cycles per iteration you can achieve by only rearranging code (no unrolling)? You do not need to show the scheduled code.

(c) Unroll the given loop once, and schedule it to avoid stalls.

Question 3 (continued)

(d) How many cycles per iteration does this unrolled loop achieve?

(e) If you were to unroll the loop 8 times, how many cycles per iteration would this achieve? (*hint: you do not need to actually perform the unrolling to answer this question)*

(f) Software pipeline the given loop so that it has **three iterations** overlapped simultaneously, and so that it has **no stalls**. Give the code for your solution below. Do not show start-up or clean-up code.

(g) How many cycles does your software-pipelined loop get per iteration?

Question 3 (continued)

(h) For the software-pipelined version of the loop, what is

- the maximum execution latency for `multd`
- the maximum execution latency for `addd`, and
- the maximum load-use latency for `ld`

such that the loop runs without stalls?

(i) Let's say that computing the dot product is particularly important to your company. You are allowed to design a VLIW architecture especially for this purpose. What is the **minimal mix** of functional units that you would need to get close to one clock cycle per iteration? Assume that you will use a combination of software pipelining and loop unrolling.

You may choose from

- load/store units
- floating-point multipliers
- floating-point adders
- integer units (which also do branches)

Hint: you should be able to answer this question without too much analysis.

(j) What is the minimal number of times that you need to unroll the dot-product loop before software pipelining in order to get cycles per iteration close to 1 on your VLIW machine? *(you do not have to give the unrolled code; you should be able to answer this question without actually doing the unrolling/scheduling)*

Hint: as you consider a software pipelined version of the loop, assume that you will break the sum into multiple pieces that you will sum at the end.

Question 4: Branching Out

In this question, you will examine several different schemes for branch prediction, using the following code sequence for a DLX-like ISA with **no** branch delay slot:

```
           addi  r2, r0, #45    ; initialize r2 to 101101, binary
           addi  r3, r0, #6     ; initialize r3 to 6, decimal
           addi  r4, r0, #10000 ; initialize r4 to a big number
   top:    andi  r1, r2, #1     ; extract the low-order bit of r2
PC1-->     bnez  r1, skip1      ; branch if the bit is set
           xor   r0, r0, r0     ; dummy instruction
   skip1:  srli  r2, r2, #1     ; shift the pattern in r2
           subi  r3, r3, #1     ; decrement r3
PC2-->     bnez  r3, skip2
           addi  r2, r0, #45    ; reinitialize pattern
           addi  r3, r0, #6
   skip2:  subi  r4, r4, #1     ; decrement loop counter
PC3-->     bnez  r4, top
```

This sequence contains 3 branches, labeled by **PC1**, **PC2**, and **PC3**. The following are the steady-state taken/not-taken patterns for each of these branches (**T** indicates taken, **N** indicates not-taken):

PC1: `TNTTNT TNTTNT ...`
PC2: `TTTTTN TTTTTN ...`
PC3: `TTTTTT TTTTTT ...`

please turn to next page . . .

Question 4 (continued)

Here is one branch predictor structure:

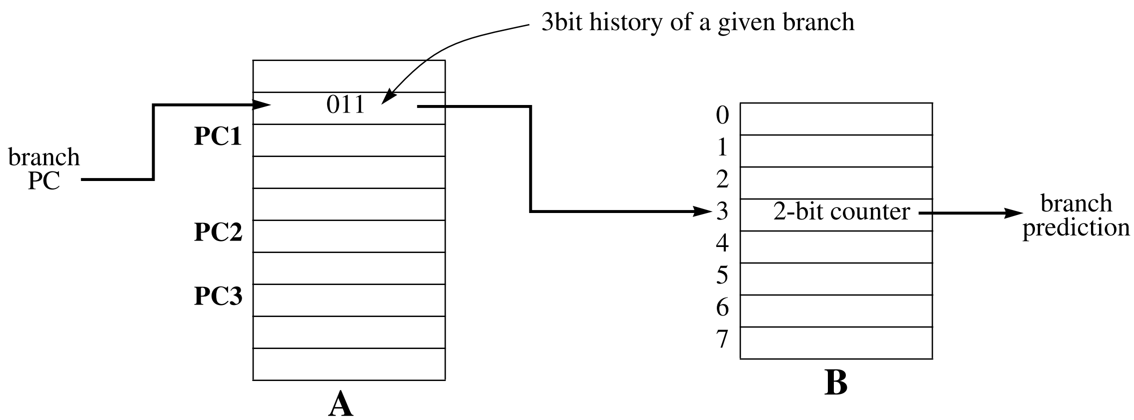

Table A is indexed by the program counter. Table B contains two-bit saturating up-down counters as described in class; they predict taken when the counter value is 2 or 3. You can assume that **PC1**, **PC2**, and **PC3** map to the *distinct* slots in the table shown above.

(a) Classify the above predictor using the Yeh & Patt scheme presented in class.

(b) What is the prediction success rate (that is, the ratio of correctly predicted branches to total branches) for each of the three branches (**PC1**, **PC2**, and **PC3**) using this predictor, **in steady-state**, after the loop has been running for many iterations? If you want, you may assume that all counters and histories start initialized to zero, but this should not impact your answer.

(c) What is the **overall** prediction rate for the given code sequence on this predictor?

Question 4 (continued)

(d) Let's now consider another approach to branch prediction. Draw the structure of a **GAg** predictor that uses 3 global history bits and a table of 2-bit saturating up-down counters.

(e) What is the **overall** prediction rate for the given code sequence on this predictor?

(f) How many global history bits are required to get 100% prediction accuracy with this type of predictor (GAg)?