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:	**SOLUTION SET**
SID number:	
Email address:	

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

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. **(3 points)**

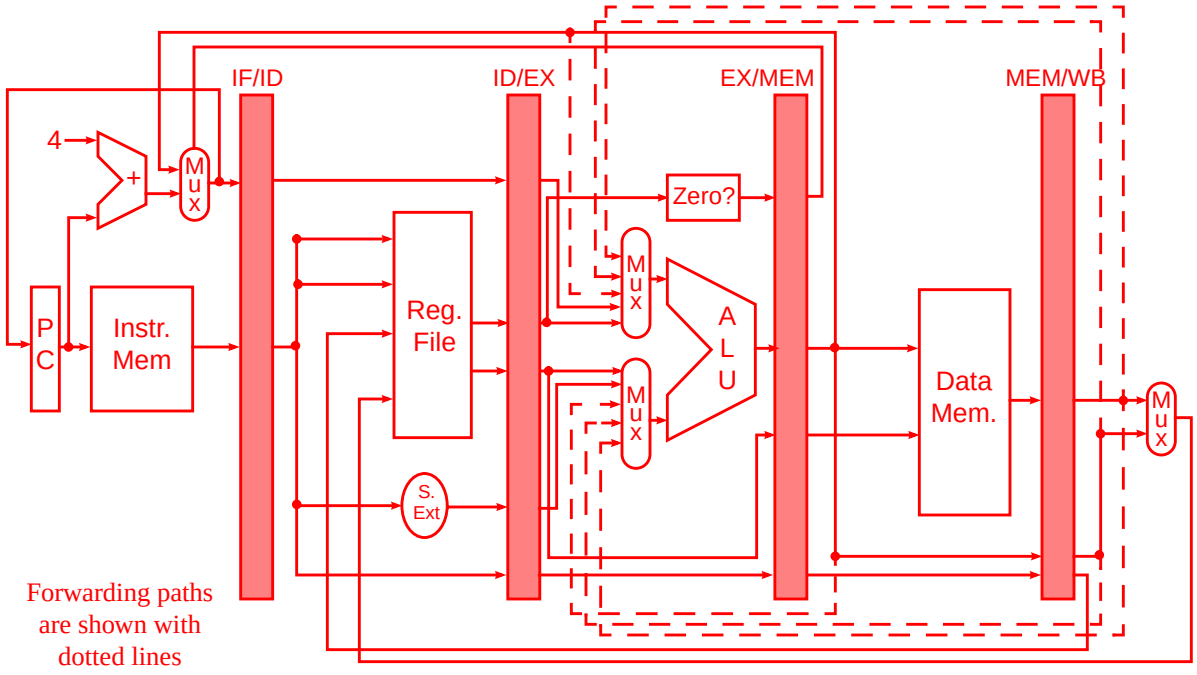
An interrupt/exception is considered precise if there is a single instruction (or interrupt point) for which all instructions preceding that instruction have committed their state and no following instructions, including the interrupting instruction, have modified any state.

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

The following are some of the many possibilities:

- to provide the restartability needed to implement virtual memory/paging
- to handle IEEE floating-point exceptions
- to simulate unimplemented instructions in software
- to allow fast interrupt handling/context switching
- to simplify debugging by providing a consistent view of the program's state
- to reduce the amount of state needed to be saved by the operating system

(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. **(5 points)**



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): **(1 point)**

TIME ⟶

```
lw    F    D    X    M    W
add       F    D    X    M    W
sub            F    D    X    M    W
```

Notice that I put time on the horizontal axis. This is the traditional way of making these time-evolution pipeline diagrams, as it makes it easy to see what's happening in a given cycle by taking a vertical slice of the diagram.

(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. **(1 point)**

TIME ⟶

```
lw    F    D    X    M    W
add       F    D    X    M    W
sub            F    D    X    M    W
```

(f) Which exception happens **first** in **time**? **(1 point)**

The ITLB fault for instruction 3 (`sub r5, r1, r10`)

(g) Which exception must be **taken** in order to have a precise exception? **(1 point)**

The DTLB fault for instruction 1 (`lw r2, 10(r1)`)

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. **(5 points)**

The simplest solution is to add an *exception bit*, an *exception type field*, and a *write-inhibit bit* to each of the pipeline latches. When an instruction generates an exception, it sets its exception bit to 1, and writes a bit pattern that identifies the type of exception into the exception type field of the pipeline register at the end of its current pipeline stage. It also sets the write-inhibit bit in this pipeline register AND in all pipeline registers corresponding to instructions issued *later* than the exception-causing instruction (*i.e.*, all instructions in earlier pipeline stages). The write-inhibit bit disables register and memory writes for those instructions that have it set. Note that the exception-generating instruction does NOT raise an exception at this point. Instruction fetch is disabled, but all instructions issued before the exception-generating instruction are allowed to continue unmodified.

Since all exceptions are generated and detected by the end of the MEM stage, we add logic at the end of that stage that checks the exception bit, and, if set, raises the exception (*i.e.*, sets whatever registers indicate the exception type, and restarts instruction fetch at the PC of the exception handler).

These modifications guarantee precise interrupts, since any instructions issued before an exception-causing instruction are allowed to complete, and all exceptions are raised in-order since exceptions are only raised in one pipeline stage (MEM). In addition, no inconsistent register or memory state is possible, since the write-inhibit bit prevents later instructions from writing to the register file or memory (essentially, turning these instructions into no-ops). One tricky case involves an instruction that generates an exception in the MEM stage. We must construct the hardware such that when the exception is detected, any modifications to memory are immediately halted and not allowed to commit. This is most important for stores, and easily solved by buffering the store until all possible exceptions are ruled out. Finally, note that by raising exceptions in MEM rather than in WB simplifies the case where an exception-generating instruction is followed by a store.

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. **(10 points)**

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

See Figure 4.34, page 311, in Hennessy & Patterson, *Computer Architecture: A Quantitative Approach, 2nd. Ed.* for the diagram.

Question 2 (continued)

(b) Explain how this architecture supports precise interrupts. **(4 points)**

Instructions are issued in program order into the reorder buffer (as well as to reservation stations, of course). When an exception occurs, the exception is flagged in the reorder buffer entry corresponding to the instruction that caused the exception. As reorder buffer entries reach the head of the buffer, they are checked to see if their exception flags are set. If so, the instruction is not allowed to graduate; instead, it and all instructions later than it in program order are flushed from the reorder buffer, and instruction fetch is redirected to the interrupt handler. Since instructions only modify state when they graduate from the reorder buffer, all instructions ahead of the exception-generating instruction are allowed to complete and commit their state changes before the exception is flagged, and similarly all instructions behind the exception-generating instruction (including that instruction) are flushed before graduating, and thus they never modify state.

One optimization is to flush the reorder buffer immediately upon detecting the exception, and then redirecting instruction fetch to the interrupt handler. This is effectively speculative execution of the interrupt handler.

(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. **(6 points)**

In order to speculate load-store dependencies, we first modify the issue unit to issue loads and stores without regard to dependencies between them. Having done this, we now must add hardware to detect mis-speculated dependencies and to roll-back the machine state when this happens.

We do this by having a queue that holds all memory operations in program order. Each entry contains the type of operation (ld/st), the calculated address, and a value. The issue unit issues memory operations to this queue at the same time it issues them to the reorder buffer. The queue is synchronized with the reorder buffer such that entries are removed (and stores are sent to memory) when they are committed by the reorder buffer.

Let's consider loads first. Loads are issued into the queue. When the effective address for the load has been calculated, all entries in the queue corresponding to instructions *prior to* the load are searched. If an entry corresponding to a store with the same address is *not* found, the load is allowed to issue to the memory system. If a store with the same address is found, the load takes the value of that store (if it is available), or takes the pointer to the functional unit/ reservation station producing the store's value (if the value is not available yet).

Now consider stores. Stores are issued directly into the queue. When the effective address for the store has been calculated, all entries in the queue corresponding to *loads* issued *after* the

Question 2 (continued)

store are searched. Any loads found with matching addresses are incorrectly speculated loads, and must be flushed; all entries in the memory queue and reorder buffer corresponding to that load and all subsequent instructions are flushed. The load then is reissued, and will detect the dependency with the store, as described in the previous paragraph. Note that this address check happens *as soon as the store address is available*. If we wait until the store reaches the head of the memory queue or reorder buffer, we may end up squashing subsequent loads that were not in fact misspeculated (loads issued after the store that finished their address computation *after* the store address was available, and thus got the correct dependence*)*.

This hardware avoids violating dependencies as follows. Let's imagine we have a store followed by a dependent load. If the store finishes its address calculation first, the load will see the matching address when it (the load) completes its own address calculation, and will either take the store's value or will wait for the store. On the other hand, if the load finishes its address calculation first, it will be issued to memory incorrectly. However, at some point the store will complete its address calculation and will check the queue. Since entries from the queue are retired in program order, the load will still be in the queue, and the store will see that it has a matching address. It can then roll the reorder buffer and queue back to the point of that load, and reissue it. Since the store address is available, the newly-issued load will see the dependence.

This solution differs from the "original" solution of no speculation by allowing loads to go to memory in the common case where there is no dependence. The original solution requires newly-issued loads to wait for all prior stores that have not completed their address calculation (since there is no mechanism to detect violations and roll-back state). In our new solution, loads can go ahead without waiting, effectively assuming that the addresses of any pending stores will be different. In the case where this assumption is false, the load is squashed and the state is restored to the point where the speculation began.

Finally, note that in practice the single memory queue is often split into separate load and store queues; other solutions involve using the reorder buffer as one or both of the memory queues.

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 that all functional units are fully pipelined and 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? **(4 points)**

Each iteration takes **14 cycles**, which can be determined by simply counting the number of unfilled execute slots between instructions. There are two stall cycles between the second `ld` and the `multd` (due to a load-use hazard on `f6`); there are three more between the `multd` and the `addd` (since the `addd` uses `f7`, the result of the multiply); and finally there is one due to the `nop`. So there are 6 stalls, and 8 computational instructions (not counting the `nop`), so that makes 14 cycles.

(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. **(3 points)**

The idea here is to move the independent integer operations up into the floating-point stall cycles. Here's the scheduled code:

```
loop:  ld    f5, 0(r1)
       ld    f6, 0(r2)
       addi  r1, r1, #8
       addi  r2, r2, #8
       multd f7, f5, f6
       subi  r3, r3, #1
       bnez  r3, loop
        addd f4, f4, f7   ; one stall on f7 before completes
```

Question 3 (continued)

In this code sequence, all of the stall slots have been filled with integer instructions except for one stall due to the fact that there are only two instructions between the `multd` and the `addd`. The `nop` has been removed. There are 8 instructions and one stall slot, for a total of **9 cycles.**

(c) Unroll the given loop once, and schedule it to avoid stalls. **(6 points)**

We unroll the loop once and schedule it below. Note that we have renamed `f5` to `f8`, `f6` to `f9`, and `f7` to `f10` in the second iteration. We've also assumed that the requested number of iterations (the starting value of `r3`) is even, for simplicity. Notice the offset adjustments in the second set of loads, and in the `addi`'s and `subi`:

```
loop:  ld    f5, 0(r1)    ; load first element from array #1
       ld    f6, 0(r2)    ; load first element from array #2
       ld    f8, 8(r1)    ; load second element from array #1
       ld    f9, 8(r2)    ; load second element from array #2
       multd f7, f5, f6   ; multiply first set of elements
       addi  r1, r1, #16  ; update array 1 pointer by 2 elts.
       multd f10, f8, f9  ; multiply second set of elements
       addi  r2, r2, #16  ; update array 2 pointer by 2 elts.
       addd  f4, f4, f7   ; update sum with first set of elts.
       subi  r3, r3, #2   ; decrease loop counter by two
       bnez  r3, loop
        addd f4, f4, f10  ; update sum w/second set of elts.
```

There are no stalls in this code, and we've removed the `nop`.

(d) How many cycles per iteration does this unrolled loop achieve? **(2 points)**

There are no stalls in the above loop, and 12 instructions, so there are 12 cycles for each iteration of the *unrolled* loop. However, each of these unrolled iterations produces **two** results, so there are effectively 12 cycles for two iterations of the original loop, or 12/2 = **6 cycles per iteration.**

(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)* **(3 points)**

Since we've removed all the stalls from the loop, unrolling more times won't help eliminate stalls. It will, however, help amortize the overhead code (the `addi`'s, `subi`, and `bnez`) across multiple iterations. There are 4 real computational instructions per iteration of the original loop (`ld`, `ld`, `multd`, `addd`), so if we unroll 8 times we have 4*8 = 32 compute instructions plus the 4 overhead instructions, or 36 cycles. We produce 8 results in those 36 cycles, so the cycles per iteration is 36/8 = **4.5.**

Question 3 (continued)

(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. **(7 points)**

The idea here is to take instructions from independent iterations, then combine them into one "rolled" loop that is doing part of each iteration at once. Note that the problem asks for three iterations to be overlapped, despite the fact that there are four compute instructions per result; it is possible to do this because the loads are independent and their results are used simultaneously, so they can be issued in the same iteration. Here's the software-pipelined loop, omitting startup/cleanup code:

```
loop:  addd  f4, f4, f7   ; update the sum (iteration 3)
       multd f7, f5, f6   ; compute the product (iter. 2)
       ld    f5, 0(r1)    ; load the two values (iter. 1)
       ld    f6, 0(r2)
       addi  r1, r1, #8   ; increment pointers (iter. 1)
       subi  r3, r3, #1   ; decrement element count (iter. 1)
       bnez  r3, loop
        addi r2, r2, #8   ; increment second pointer (iter. 1)
```

(g) How many cycles does your software-pipelined loop get per iteration? **(2 points)**

There are no stalls in the above code, and there are 8 instructions, so each iteration of the software-pipelined loop takes 8 cycles. Note that one result is produced per iteration, although it takes 3 iterations for a particular result to be generated. In steady-state, there is one result per iteration, so there are **8 cycles/result**. Note that this is *slower* than the unrolled loop. However, it uses fewer registers, and can tolerate much greater functional unit latency. It also does not increase code size as much.

(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? **(3 points)**

Note: execution latency is the number of execute cycles, not the operation latency (the visible latency between the instruction and the time its result is available).

In the above code, there are 6 instructions between the `multd` and the `addd` (which consumes the `multd` results in the next iteration). Since our pipeline is fully bypassed, this means that the `multd` can have up to **7 cycles** of execution latency. Similarly, the `addd` result is not consumed until 7 instructions have passed; thus `addd` has **8 cycles** of execution latency. Finally, there are 5 instructions between the `ld` and the result use (`multd`), so the load-use latency is **5 cycles**.

Question 3 (continued)

(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. **(5 points)**

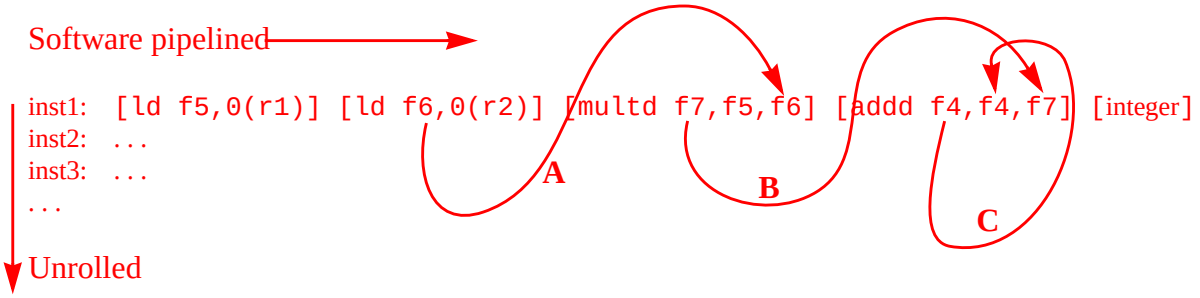
Assuming that software pipelining and loop unrolling can get rid of all dependencies, then we just need to have all pieces of an iteration in one VLIW instruction. Thus, we need two `ld`s, one `multd`, one `addd`, and integer slots for the overhead instructions (two `addi`'s, one `subi`, and one branch). Knowing nothing more, you might assume that four integer units would be needed to handle these overhead instructions. However, it is probably less, depending on how much unrolling you do—if a loop has 2 VLIW instructions, then you only need 4/2 = 2 integer units. If a loop has 4 VLIW instructions, then you only need 4/4 = 1 integer unit. The answer to part (j), below, tells us that there are in fact 4 instructions, so only one integer unit is needed. Thus, we need:
- 2 load/store units
- 1 floating-point multiplier
- 1 floating-point adder
- 1 integer unit

(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. **(5 points)**

Imagine that you use software pipelining across a VLIW instruction, getting three overlapped iterations (just like in (g), but all in parallel). By having as many accumulators as the number of VLIW instructions (the number of times unrolled), then all VLIW instructions could be independent. So, the only "stalls" that we would have to get rid of would be from one iteration to the next in a single VLIW instruction:

Question 3 (continued)



For arrow **A**, we would need enough VLIW instructions so that the load-use delay is satisfied, *i.e.*, two more. For arrow **B**, we would need three more instructions, and for arrow **C**, two more instructions. So, arrow **B** is the critical path. This means that we need **4 total VLIW instructions**, or an unrolling of 4.

Thinking back on part (i), we see that with the functional unit mix specified there, the hardware would be 100% utilized. If you want to see the actual VLIW code, first, ignore integer operations. Put down four VLIW instructions spread over twelve iterations of the original loop:

```
lp: [ld f1,0(r1)]   [ld f2,0(r2)]   [multd f3,f1,f2]   [addd f4,f4,f3]   [int]
    [ld f5,8(r1)]   [ld f6,8(r2)]   [multd f7,f5,f6]   [addd f8,f8,f7]   [int]
    [ld f9,16(r1)]  [ld f10,16(r2)] [multd f11,f9,f10] [addd f12,f12,f11][int]
    [ld f13,24(r1)] [ld f14,24(r2)] [multd f15,f13,f14][addd f16,f16,f15][int]
```

Now we need to add the integer instructions. Scheduling them into the rest of the instructions, and adjusting offsets accordingly, gives:

```
lp:[ld f1,0(r1)]  [ld f2,0(r2)]   [multd f3,f1,f2]   [addd f4,f4,f3]   [addi r1,r1,#32]
   [ld f5,-24(r1)][ld f6,8(r2)]   [multd f7,f5,f6]   [addd f8,f8,f7]   [addi r2,r2,#32]
   [ld f9,-16(r1)][ld f10,-16(r2)][multd f11,f9,f10] [addd f12,f12,f11][subi r3,r3,#4]
   [ld f13,-8(r1)][ld f14,-8(r2)] [multd f15,f13,f14][addd f16,f16,f15][bnez r3, lp]
```

Part of the following code would have to add `f8`, `f12`, `f16` and `f4` together to get the final dot product in `f4`.

Finally, note that we wouldn't have been able to generate this code if we hadn't added four separate accumulators (`f4`, `f8`, `f12`, and `f16`), since otherwise all of the VLIW instructions would have depended on each other.

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:

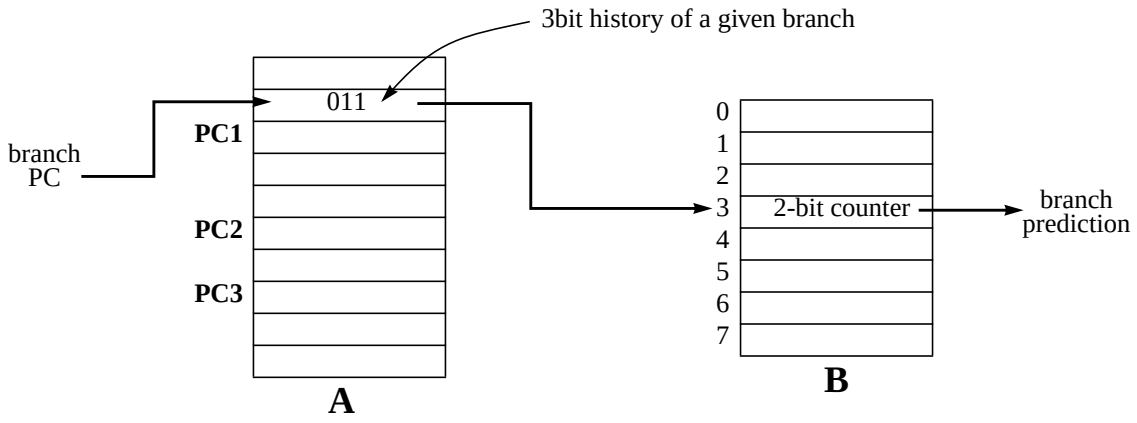


Table A is indexed by the program counter. Table B contains two-bit saturating up-down counters as described in class; they predict taken when the counter value is 2 or 3. You can assume that **PC1**, **PC2**, and **PC3** map to the *distinct* slots in the table shown above.

(a) Classify the above predictor using the Yeh & Patt scheme presented in class. **(3 points)**

This is a **PAg** predictor: per-address history, global predictor table.

(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.

The correct answer is:

- PC1: 100%
- PC2: 66.7%
- PC3: 100%

To do this problem, we first determine the steady-state prediction of the relevant counters in table B. The following is the history of the branches:

```
PC1: T N T T N T T N T T N T T N T T N T
PC2: T T T T T N T T T T T N T T T T T N
PC3: T T T T T T T T T T T T T T T T T T
```

Imagine a sliding window, illustrated by the box above. At some point during steady-state operation, the branch history for PC1 will be TNT, for PC2 TTT, and for PC3 TTT. This is illustrated by the solid box, above. At this point, the next set of branch patterns is given by the next column of the chart, or T for PC1, T for PC2, and T for PC3. So we build a table that

holds the branch behavior for each entry in table B. For example, PC1's history is TNT, and the branch is taken, so we put a T in entry 101 (decimal 5) of the table, etc.:

000	
001	
010	
011	
100	
101	T
110	
111	TT

We now slide the window one branch to the right, indicated by the dotted box, and update the table again. This continues until we hit a repeating pattern (after we enter 18 branches into the table). The result is shown below, with the repeat pattern in italics:

000	
001	
010	
011	NNT *NNT* . . .
100	
101	TTT *TTT* . . .
110	TTT *TTT* . . .
111	TTTTNTTTT *TTTTNTTTT* . . .

We now imagine each of these patterns being fed into a 2-bit saturating counter, and we observe that counter 011 will always predict not-taken, counter 101 will predict taken, counter 110 will predict taken, and counter 111 will predict taken in steady-state.

Going back to the original branch patterns, we see that PC1 goes through the patterns TNT, NTT, TTN, with next-branch behavior T, N, T, respectively. These correspond to counters 101, 011, and 110, which predict T, N, and T, respectively. These predictions match the branch behavior exactly, so PC1 is predicted with 100% accuracy.

PC2 goes through the states TTT, TTT, TTT, TTN, TNT, and NTT, with next-branch behavior T, T, N, T, T, and T. The states correspond to entries 111, 111, 111, 110, 101, and 011, and thus predictions T, T, T, T, T, and N respectively. The third and sixth predictions don't match the actual branch behavior, so 2 out 6 branches are mispredicted, so PC2 has a prediction accuracy of 4/6, or 66.7%.

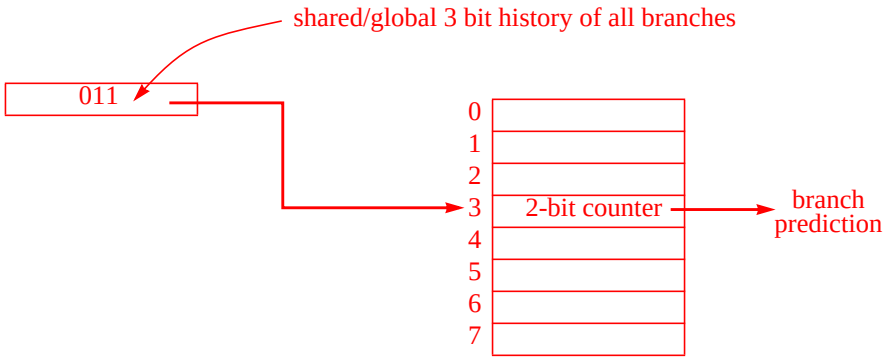
Question 4 (continued)

Finally, PC3 only has one pattern, TTT, and the next-branch behavior is always T. It is thus predicted correctly 100% of the time, since entry 111 predicts T.

(c) What is the **overall** prediction rate for the given code sequence on this predictor? **(2 points)**

Above, we saw that the branch pattern repeats every eighteen branches. Of those eighteen, two are mispredicted (the two mispredicts of PC2). Thus, the overall prediction rate is 16/18, or **88.9%**.

(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. **(3 pts)**



(e) What is the **overall** prediction rate for the given code sequence on this predictor? **(5 points)**

We proceed as in part (b), but now we look at a global window. Here is the global branch pattern:

T T T N T T T T T T T T N T T T N T

This time, our sliding history window (the solid box) only covers three branches. In this case, with a global pattern of TTT, the next branch encountered is not taken. So we can build another table as in part (b); the dotted box shows how the window advances from the first position. After the pattern repeats (18 branches), the table looks like:

Question 4 (continued)

000	
001	
010	
011	TTT *TTT* . . .
100	
101	TTT *TTT* . . .
110	TTT *TTT* . . .
111	NTTTTTNNT *NTTTTTNNT* . . .

Counters 011, 101, and 110 will saturate to T. Counter 111 is more tricky. Let's assume that the counter starts off with a value of 0 (*i.e.*, strong predict-not-taken). Other starting values will produce the same steady-state prediction, but verification of this is left as an exercise for the reader. Then, the counter evolves as follows:

counter value:	0	0	1	2	3	3	3	2	1		2	1	2	3	3	3	3	2	1		2	1	2	...
prediction:	N	N	N	T	T	T	T	T	N		T	N	T	T	T	T	T	T	N		T	N	T	...
branch behavior:	N	T	T	T	T	T	N	N	T		N	T	T	T	T	T	N	N	T		N	T	T	...

The pattern in the box will repeat, and thus it is the steady-state pattern.

So, what's the overall prediction rate? Of the 18 branches in the above table, those in entries 011, 101, and 110 will always predict correctly. This accounts for 9 of the branches. The other 9 branches correspond to those in the box above (for entry 111). We see that only four of those 9 entries are predicted correctly (the third, fourth, fifth, and sixth). So, out of the eighteen branches, 9+4=13 are predicted correctly, giving us a **72.2%** overall prediction rate.

(f) How many global history bits are required to get 100% prediction accuracy with this type of predictor (GAg)? **(2 points)**

Nine bits are required. There are two maximal-length repeating patterns, each with 8 bits: TTTTTTTT and TTTTNTTT. The first pattern is always followed by N, so it only needs 8 bits of state. The second pattern can be followed by either N or T, so we need an extra ninth bit of state to distinguish these. All other patterns are shorter than 8 bits, and so are handled correctly with 9 history bits.