

Announcement: The final exam will be given as scheduled on Tuesday, May 15, from 10:30 AM to 12:30 PM. It will be given with alternate seating in two rooms: LC-6(for even ids: 0,2,4,6,8) and LC-22(for odd ids: 1,3,5,7,9).

The final exam format will be similar to that of the midterms: closed book with 2 sheets of notes permitted (which could be double sided.) It will be comprehensive: 3/4 of the points will be on topics covered in the other 2 exams and 1/4 of the points will be on the later topics: System interconnection using buses, interrupts, pipelining, and memory hierarchies (with caches and virtual memories).

Reading: Patterson and Hennessy Chapter 7: Sections 7.1 through 7.4; concentrate on topics covered in the lectures.

Problem 1: Patterson and Hennessy problem 6.3: First, copy the code, write a NOP instruction under the beq and explain why that NOP instruction is necessary. Then, answer the question by modifying the code so the instruction in the delay slot (the NOP) is changed to one that does useful work. (You **will** have to make one or more additional modifications.) Finally, under the assumption that the loop runs 5 times (5 loads and adds) figure out (1) the number of clock steps used to execute all steps for the original code and (2) the number of clock steps used to execute your new, improved code. No stalls will occur because the ALU result is forwarded to the branch comparator in time for the branch decision to be made (see fig. 6.51).

```
Loop: lw    $2, 100($3)
      addi  $3, $3, 4
      beq  $3, $4, Loop
      nop
```

The nop instruction is necessary so the assembler/loader puts the well-defined instruction nop right after the delayed branch, so errors due to a “garbage” instruction in delay slot are guaranteed not to occur.

```
;Make sure this code is entered through the top, not a jump to Loop:
      addi  $3, $3, 4
Loop: lw    $2, 96($3)
      beq  $3, $4, Loop
      addi  $3, $3, 4    ;delay slot instruction, always executed.
;Register 3 has a DIFFERENT value at the end compared to the original!
      addi  $3, $3, -4
;This instruction may be needed to make revised code do exactly what
; the original did.
```

The answer below is based on the assumption (written on the assignment sheet) that no stalls result from dependencies of a branch upon an ALU instruction. This assumption is feasible since the ALU calculation and the branch decision can be made in the same clock step if the combinational circuitry is fast enough and enough forwarding hardware is used. However, PH’s pipelined CPU diagrams do not show all the necessary forwarding hardware.

Instruction	Clock Steps										
	1	2	3	4	5	6	7	8	9	10	11
lw \$2, 100(\$3)	IF	ID	EX	MEM	WB						
addi \$3, \$3, 4		IF	ID	EX	MEM	WB					
beq \$3, \$4, Loop			IF	ID	EX	MEM	WB				
nop				IF	ID	EX	MEM	WB			
lw \$2, 100(\$3)					IF	ID	EX	MEM	WB		

Notice the loop iterations begin on clock steps 1, 5, 9, 13, 17, etc., so the 5th and last iteration begins on step 17. We show how the loop terminates and the pipeline empties:

Instruction	Clock Steps							
	17	18	19	20	21	22	23	24
lw \$2, 100(\$3)	IF	ID	EX	MEM	WB			
addi \$3, \$3, 4		IF	ID	EX	MEM	WB		
beq \$3, \$4, Loop			IF	ID	EX	MEM	WB	
nop (fall through instr)				IF	ID	EX	MEM	WB
					IF	ID	EX	MEM WB

Hence the instruction right after the loop is fetched during step 21, so the loop itself cost a total of 20 clock steps. If the loop were removed, the program would have used 20 fewer clock steps. Note that 24 clock steps are needed for all the stages of the loop instructions to finish.

ANALYSIS OF PIPELINE RUNNING THE REVISED CODE:

Instruction	Clock Steps										
	1	2	3	4	5	6	7	8	9	10	11
addi \$3, \$3, 4	IF	ID	EX	MEM	WB						
lw \$2, 96(\$3)		IF	ID	EX	MEM	WB					
beq \$3, \$4, Loop			IF	ID	EX	MEM	WB				
addi \$3, \$3, 4				IF	ID	EX	MEM	WB			
lw \$2, 100(\$3)					IF	ID	EX	MEM	WB		

The lw instructions began executing at steps 2, 5, 8, 11, and 14. The pipeline sequence near the end of the loop is:

Instruction	Clock Steps										
	14	15	16	17	18	19	20	21	22	23	24
lw \$2, 96(\$3)	IF	ID	EX	MEM	WB						
beq \$3, \$4, Loop		IF	ID	EX	MEM	WB					
addi \$3, \$3, 4			IF	ID	EX	MEM	WB				
addi \$3, \$3, -4 (next instruction)				IF	ID	EX	MEM	WB			
					IF	ID	EX	MEM	WB		

Thus, the revised code consumes only 17 clock steps, even though extra instructions were added before and after the loop. Note the 4 clock steps 18, 19, 20 and 21 needed for the last addi are also used for later instructions.

Practice Problem A: Patterson and Hennessy problem 6.4

Practice Problem B: Patterson and Hennessy problem 6.5

Practice Problem C: Patterson and Hennessy problem 6.11: (Draw the pipeline timing diagram. When the clock strikes at the end of the 5th clock step, one general purpose register will get a new value and the Read Data 1 and Read Data 2 ID/EX pipeline registers will get new values read from two general purpose registers. Figure out which three of the general purpose registers these are.)

Problem 2: Patterson and Hennessy 6.23.

```
lw    $3, 0($5)
add   $7, $7, $3 ;one stall
lw    $4, 4($5)
add   $8, $8, $4 ;one stall
sw    $6, 0($5)
add   $10, $7, $8
beq   $10, $11, Loop ;one stall if Fig. 6.51 HW is used.
```

The hardware of Fig. 6.51 shows that an ALU result must be stored in an EX/MEM pipeline register before it can be forwarded to register file for use to calculate a branch decision. If we make the assumption that enough forwarding hardware is built to forward the ALU result to the “=” tester of figure 6.51, only one stall is caused by the dependency of the beq on the add through \$10.

Problem 3: Patterson and Hennessy page 628, problems 7.7 and 7.8. (These are both problems about 16 word direct mapped caches.)

First, we express the given address reference sequence, the same for both problems, in binary. After each address, we write its decomposition into tag, index and offset field, first for the problem 7.7 cache and second for the problem 7.8 cache.

Time	Address	binary	tag,index,offset	tag,index,offset
1	1	000001	00,0001=1,0	00,00=0,01
2	4	000100	00,0100=4,0	00,01=1,00
3	8	001000	00,1000=8,0	00,10=2,00
4	5	000101	00,0101=5,0	00,01=1,01
5	20	010100	01,0100=4,0	01,01=1,00
6	17	010001	01,0001=1,0	01,00=0,01
7	19	010011	01,0011=3,0	01,00=0,11
8	56	111000	11,1000=8,0	11,10=2,00
9	9	001001	00,1001=9,0	00,10=2,01
10	11	001011	00,1011=11,0	00,10=2,11
11	4	000100	00,0100=4,0	00,01=1,00
12	43	101011	10,1011=11,0	10,10=2,11
13	5	000101	00,0101=5,0	00,01=1,01
14	6	000110	00,0110=6,0	00,01=1,10
15	9	001001	00,1001=8,0	00,10=2,01
16	17	010001	01,0001=1,0	01,00=0,01

Next, we trace the activity in the cache when the references are made to the addresses in the above sequence. Each action is written next to the cache block affected by that action.

For each action, we record the time, the address of the reference, the tag, whether it's a hit or a miss, and address range that the block holds now. First for the 16-block cache:

Block	Action Log
0	
1	(1,1,00,M,1-1)(6,17,01,M,17-17)(16,17,01,Hit,17-17)
2	
3	(7,19,01,M,19-19)
4	(2,4,00,M,4-4)(5,20,01,M,20-20)(11,4,00,M,4-4)
5	(4,5,00,M,5-5)(13,5,00,Hit,5-5)
6	(14,6,00,M,6-6)
7	
8	(3,8,00,M,8-8)(8,56,11,M,56-56)
9	(9,9,00,M,9-9)(15,9,00,Hit,9-9)
10	
11	(10,11,00,M,11-11)(12,43,10,M,43-43)
12	
13	
14	
15	

Now for the 4 block cache; same address sequence and notation:

Block	Action Log
0	(1,1,00,M,0-3)(6,17,01,M,16-19)(7,19,01,H,16-19)(16,17,01,H,16-19)
1	(2,4,00,M,4-7)(4,5,00,H,4-7)(5,20,01,M,20-23)(11,4,01,M,4-7)(13,5,01,H,4-7)(14,6,01,H,4-7)
2	(3,8,00,M,8-11)(8,56,11,M,56-59)(9,9,00,M,8-11)(10,11,00,H,8-11)(12,43,10,M,40-43)(15,9,00,M,9-12)
3	

Problem 4: Patterson and Hennessy problem 7.32

Since the virtual address is a *byte* address, and the page size of 16-KB is $2^{10+4} = 2^{14}$ bytes, 14 of the 40 virtual address bits comprise the offset field. Therefore, the page table must accomodate as many page table entries as can be counted by the remaining $40 - 14 = 26$ virtual address bits. This number of page

table entries is 2^{26} .

The physical address uses 36 bits to index bytes. 14 of these are comprise the byte offset since the page size is 2^{14} . Therefore the number of physical address bits used to locate a page is $36 - 14 = 22$

Each page table entry requires the 22 bits to locate the physical page (meaningful if the page table entry is *valid*), plus, following the assumption given in the problem, 4 bits for the valid, protection, dirty and use bits. Therefore, each page table entry is $26 = 22 + 4$ bits.

Answer: The total page table size is $2^{26} * 26$ bits.