UNIVERSITY OF CALIFORNIA, LOS ANGELES UCLA

BERKELEY • DAVIS • IRVINE • LOS ANGELES • RIVERSIDE • SAN DIEGO • SAN FRANCISCO SANTA BARBARA • SANTA CRUZ

CS M151B / EE M116C

Final Exam

Before you start, make sure you have all 14 pages attached to this cover sheet.

Please put your name at the top of each page.

All work and answers should be written directly on these pages, use the backs of pages if needed.

This is an open book, open notes final – but you cannot share books, notes, or calculators.

I will uphold the university policy on cheating – so please do not cheat on this exam. Keep your eyes on your own exam – and show all of your work.

NAME: Key

ID: _____

Problem 1 (14 points): _____

Problem 2 (20 points): _____

Problem 3 (20 points): _____

Problem 4 (20 points): _____

Problem 5 (30 points): _____

Problem 6 (21 points): _____

Problem 7 (25 points): _____

Total: _____ (out of 150)

Fall 2004 NAME _____

1. ***Déjà Vu (14 points):*** Consider a processor with a BCPI of 2.5. The instruction cache has a 10% miss rate and the data cache has a 15% miss rate. The miss latency for both caches is 12 cycles. Assume that 25% of all instructions are loads and that store misses do not cause stalls and calculate the TCPI. Show your work.

$MCPI = (0.1)(12) + (0.15)(0.25)(12)$

$= 1.2 + 0.45$

$= 1.65$

$TCPI = BCPI + MCPI$

$= 2.5 + 1.65 = 4.15$

TCPI: 4.15

You are considering increasing the size of the data cache to reduce the miss rate. However, this will impact the clock rate of the processor. The new data cache you intend to use will have a miss rate of 10%, but will decrease the clock rate by 5%. The instruction cache will not be impacted, and we will assume here that the miss latency for both caches will remain 12 cycles. Calculate the performance improvement (or reduction) for this modification in MIPS relative to the original configuration. Show your work.

$MCPI_{NEW} = (0.1)(12) + (0.1)(0.25)(12)$

$= 1.2 + .3$

$= 1.5$

$TCPI_{NEW} = 4.0$

$MIPS_{NEW} = \frac{CR}{4.0 \times 10^{-9}} = \frac{(.95)(CR)}{4.0 \times 10^{-9}}$

$MIPS_{OLD} = \frac{CR}{4.15 \times 10^{-9}}$

Speedup in MIPS when increasing the data cache size: -1.4%

$\frac{MIPS_{NEW}}{MIPS_{OLD}} = \frac{\frac{(.95)(CR)}{4.0 \times 10^{-9}}}{\frac{CR}{4.15 \times 10^{-9}}} = 0.986$

Fall 2004 NAME _____

2. ***Cycles of Pain (20 points):*** Consider the following instruction sequence:

SCH:
```
lw $t0, 0($s0)
add $t1, $s2, $t0
lw $t3, 8($t1)
bne $t3, $s3, SCH
add $s0, $s0, $t3
nop
```

These instructions are executed on the 5-stage pipelined MIPS processor, using full forwarding and hazard detection. The branch penalty is two cycles. There are two branch delay slots indicated by the boxes at the bottom – it is filled with the final add and a nop. Show your work in the form of a pipeline diagram – a table is provided (use IF, ID, EX, M, and WB in the appropriate slots).

a) How many cycles will it take to completely execute two iterations of this loop? 20

	Clock Cycle																														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
lw $t0, 0($s0)	IF	ID	EX	M	WB																									
add $t1, $s2, $t0		IF	ID	EX	M	WB																								
lw $t3, 8($t1)			IF	ID	EX	M	WB																							
bne $t3, $s3, SCH				IF	ID	EX	M	WB																						
add $s0, $s0, $s1					IF	ID	EX	M	WB																					
nop						IF	ID	EX	M	WB																				
lw $t0, 0($s0)							IF	ID	EX	M	WB																			
add $t1, $s2, $t0								IF	ID	EX	M	WB																		
lw $t3, 8($t1)									IF	ID	EX	M	WB																	
bne $t3, $s3, SCH										IF	ID	EX	M	WB																
add $s0, $s0, $s1											IF	ID	EX	M	WB															
nop												IF	ID	EX	M	WB														

b) Now suppose that we use a data cache in our 5-stage pipeline. In the previous section we ideally assumed that memory would take a single cycle. But with a cache, the cache latency is 2 cycles and it does not miss during the two iterations of the loop. This means our pipeline will now have 6-stages that will be seen by all instructions (i.e. memory will take two stages). Show your work.

How many cycles will it take to completely execute two iterations of this loop now? 25

	Clock Cycle																														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
lw $t0, 0($s0)	IF	ID	EX	M	M2	WB																								
add $t1, $s2, $t0		IF	ID	ID	ID	EX	M1	M2	WB																					
lw $t3, 8($t1)			IF	IF	IF	ID	EX	M1	M2	WB																				
bne $t3, $s3, SCH						IF	ID	ID	ID	EX	M1	M2	WB																	
add $s0, $s0, $s1							IF	IF	IF	ID	EX	M1	M2	WB																
nop										IF	ID	EX	M1	M2	WB															
lw $t0, 0($s0)											IF	ID	EX	M1	M2	WB														
add $t1, $s2, $t0												IF	ID	ID	ID	EX	M1	M2	WB											
lw $t3, 8($t1)													IF	IF	IF	ID	EX	M1	M2	WB										
bne $t3, $s3, SCH																IF	ID	ID	ID	EX	M1	M2	WB							
add $s0, $s0, $s1																	IF	IF	IF	ID	EX	M1	M2	WB						
nop																				IF	ID	EX	M1	M2	WB					

Fall 2004 NAME _____

3. ***Too Many 2's (20 points):*** For this problem, we will look at a 2-level page table. The first level of the page table provides a physical address for the second level page table containing the desired translation if that second level table is in physical memory (otherwise a page fault is raised and the table is loaded from disk). The virtual address is still split into two components: the virtual page number and the virtual page offset. But now the virtual page number will be split into two components: a page table number and a page table offset. The page table number will be used to index into the first level page table. The page table offset will be used to index into the second level page table pointed to by the first level page table. This is exactly the idea we discussed in class – and in case you need it, there is a diagram on the next page that shows this idea. We will assume that each second level page table occupies exactly one page of virtual memory. And we will further assume that each translation is stored along with only 2 extra protection bits (no other bits – i.e. dirty bits – are required). Consider memory with a 64-bit virtual address space, 128KB pages, 2 GB of physical memory, and a 512B 4-way set associative TLB. Show your work.

2^{17} 2^{31} 2^{9}

Fill in the following:

128kB in page, 2 byte entries

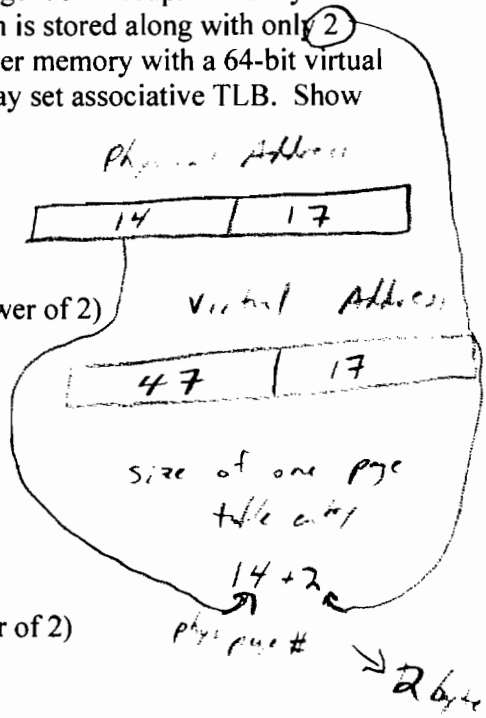
a) # of entries in ***each*** 2nd level page table: 2^{16}
(express this as a power of 2)

2^{47} entries total

b) # of 2nd level page tables : 2^{31}
(express this as a power of 2)

$\frac{2^{47}}{2^{16}} = 2^{31}$

c) # of entries in the first level page table: 2^{31}
(express this as a power of 2)



d) Circle the bits of the following 64-bit virtual address that are used to find an index into the TLB (i.e. the bits that select one of the indices of the TLB – not the tag or offset bits):

0000100001000010000100001000010000100001000010000100001000010000

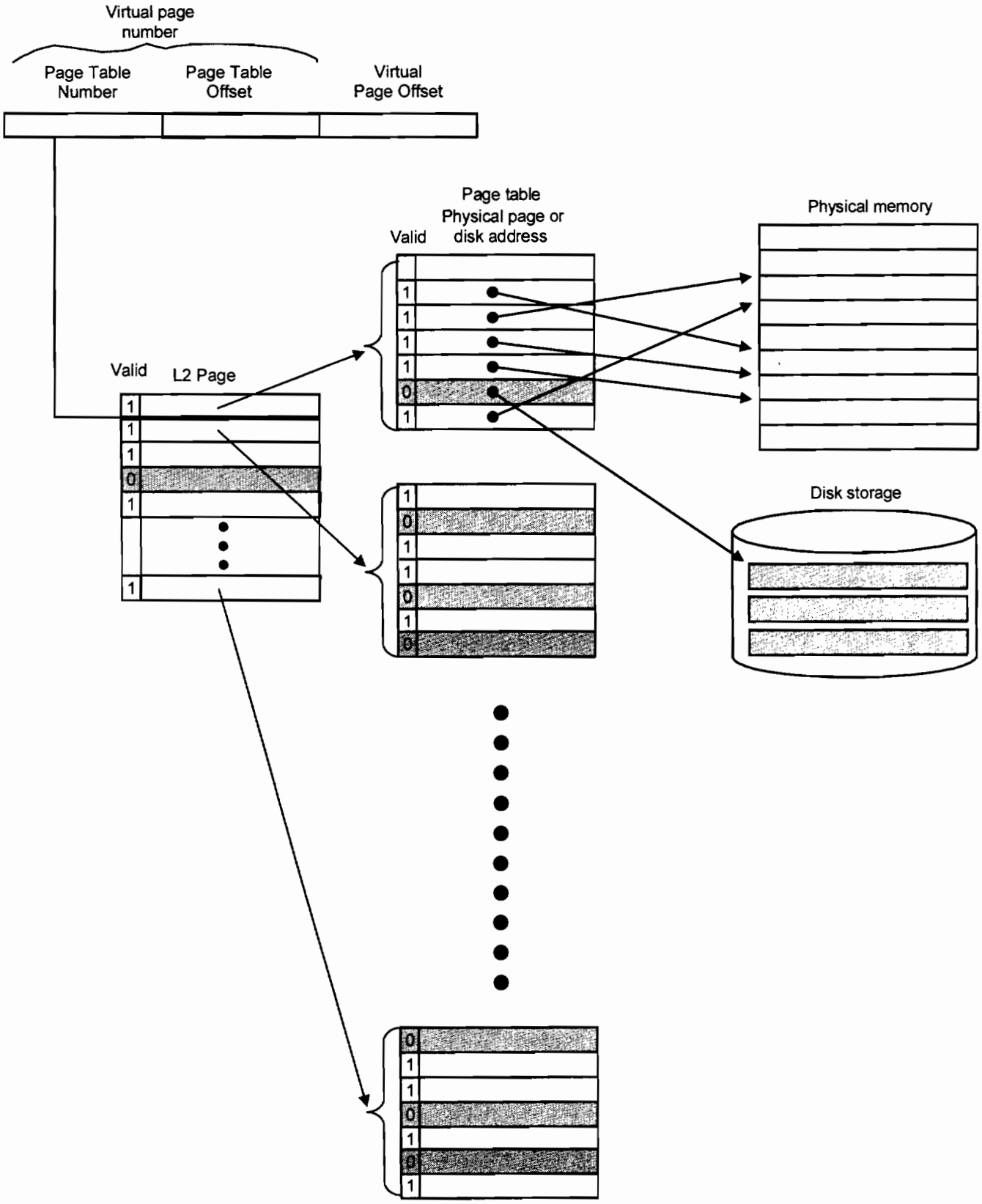
TLB size

$\frac{2^9 B}{4 \times 2B} = 2^7$

TLB index

x size

blocks


Virtual page number
Page Table Number
Page Table Offset
Virtual Page Offset
Page table
Physical page or disk address
Valid
Physical memory
Valid
L2 Page
Disk storage

4. ***Taken for a Loop (20 points):*** In this problem, we will schedule code to execute on a 2-way superscalar VLIW pipelined processor. For this processor, assume that **ANY** two independent instructions can be executed in each cycle – and that full bypassing is provided. Assume that there is a single-cycle branch penalty, and that the processor uses branch delay slots to resolve this single cycle penalty. Consider the following MIPS fragment:

```
loop:  lw $t0, 0($s1)
       lw $t1, 4($s1)
       add $t2, $t0, $t1
       sw $t2, 0($s1)
       addi $s1, $s1, 4
       bne $s1, $t3, loop
```

Assume that $s0, $s1, and $t3 are initialized before the loop is entered, and that the loop will always be taken a number of times that is a multiple of four. Unroll the loop three times (i.e. to make **four** copies of the loop body) and optimize the instructions for scheduling. Hint – you can reduce the number of loads to 5. List the new code sequence here:

```
lw $t0, 0($s1)
lw $t1, 4($s1)
add $t2, $t0, $t1
sw $t2, 0($s1)

lw $t0, 4($s1)
lw $t1, 8($s1)
add $t2, $t0, $t1
sw $t2, 4($s1)

lw $t0, 8($s1)
lw $t1, 12($s1)
add $t2, $t0, $t1
sw $t2, 8($s1)
lw $t0, 12($s1)
lw $t1, 16($s1)
add $t2, $t0, $t1
sw $t2, 12($s1)
addi $s1, $s1, 16
bne $s1, $t3, loop
```

⇒

```
lw $t0, 0($s1)
lw $t1, 4($s1)
add $t2, $t0, $t1
sw $t2, 0($s1)
lw $t3, 8($s1)
add $t4, $t1, $t3
sw $t4, 4($s1)
lw $t5, 12($s1)
add $t6, $t3, $t5
sw $t6, 8($s1)
lw $t7, 16($s1)
add $t8, $t5, $t7
sw $t8, 12($s1)
addi $s1, $s1, 16
bne $s1, $t3, loop
```

Fall 2004 NAME _____

Now that you have an optimized sequence of instructions, schedule these instructions in the following slots. Remember to fill the branch delay slots.

Cycle	1st Issue Slot (for ANY instruction)	2nd Issue Slot (for ANY instruction)
1	lw $t0, 0($s1)	lw $t1, 4($s1)
2	lw $t3, 8($s1)	lw $t5, 4($s1)
3	lw $t7, 16($s1)	add $t2, $t0, $t1
4	add $t4, $t1, $t3	add $t6, $t3, $t5
5	add $t8, $t5, $t7	sw $t2, 0($s1)
6	addi $s1, $s1, 16	sw $t4, 4($s1)
7	bne $s1, $t3, loop	sw $t6, -8($s1)
8	NOP	sw $t8, -4($s1)
9		
10		
11		
12		

Fall 2004 NAME _____

5. ***Cache Me If You Can (30 points)***: You are designing the data cache for an embedded processor. Power is critical, so you do not want something too associative or too large. You consider the following two alternatives:

DM - A 2KB direct mapped cache with 16 byte block size
SA - A 2KB 2-way set associative cache with 16 byte block size (uses LRU replacement within a set)

a. Consider the performance of these caches on the given stream of byte addresses. Note that there are 6 unique byte addresses here – and that the sequence of six addresses is repeated to make 12 total addresses. Mark whether the DM and SA cache has a "hit" or "miss" for each address – i.e. whether or not the desired memory address is found in the cache. For the addresses – assume that "..." means all leading 0's. Assume that both caches are completely empty (all entries invalid) at the start of the stream. For the DM cache only, classify each miss as capacity, compulsory, or conflict (Miss Type).

Address in Binary	Address in Decimal	DM Hit or Miss	Miss Type	SA Hit or Miss
...01100101001010010000	828704	M	Comp	M
...01100101001010000 1000	828680	M	Comp	M
...01100101001010000 1100	828684	H	—	H
...01100101111010000 1000	834824	M	Comp	M
...01101001000101111 0000	860912	M	Compulsory	M
...01101001010101111 0000	862960	M	Compulsory	M
...01100101001010010000	828704	H	—	H
...01100101001010000 1000	828680	M	Conflict	H
...01100101001010000 1100	828684	H	—	H
...01100101111010000 1000	834824	M	Conflict	H
...01101001000101111 0000	860912	M	Conflict	H
...01101001010101111 0000	862960	M	Conflict	H

b. You have an idea to try a compromise between the SA and DM caches: a pseudoassociative cache (PA cache). The PA cache will look exactly like a direct mapped cache, but if you do not find the block you are looking for at the index specified by your address, you will just check the next contiguous index for a hit in the next cycle. For example, if your address demands that you check index 12 of the PA cache, then you will check 12 first, then 13 in the next cycle. But you will *only* check 13 if the block you wanted is not in 12. This means that hits in the first location take 1 cycle, and hits in the second location take 2 cycles. If you miss in the second location, then it is a cache miss. If your index is the last entry of the cache, then your second access goes back to the first entry of the cache.

So a PA cache is just a direct mapped cache where you look in two places for a desired cache block.

The PA cache will be 2KB with a 16 byte block size.

Consider the performance of this cache on the same address stream. Mark whether the PA cache has a "hit" or "miss" for each address – i.e. whether or not the desired memory address is found in the cache. For the addresses – assume that "..." means all leading 0's.

Address in Binary	Address in Decimal	PA Hit or Miss
...01100101001010010000	828704	M
...01100101001010000100	828680	M
...01100101001010000110	828684	H
...01100101111010000100	834824	M
...01101001001011110000	860912	M
...01101001010101111000	862960	M
...01100101001010010000	828704	H
...01100101001010000100	828680	H
...01100101001010000110	828684	H
...01100101111010000100	834824	H
...01101001001011110000	860912	H
...01101001010101111000	862960	H

c. Assume that for a given workload, a hit in the PA cache will be in the first location 80% of the time, and in the second location 20% of the time. A hit in the first location takes 1 cycle and a hit in the second location takes 2 cycles. The PA cache has a 10% miss rate (so of the remaining 90% - 80% are in the first location and 20% are in the second location). The next level of the memory hierarchy is an L2 cache with a 5% miss rate and an access time of 10 cycles. The time to access main memory is 150 cycles. Calculate the average memory access time **in cycles** for this memory hierarchy. Show your work.

AMAT = 3.03

$(.9)\Big((.8)(1) + (.2)(2)\Big) + (.1)\Big(2 + \big(10 + (0.05)(150)\big)\Big)$

$(.9)(.8 + .4) + (.1)(2 + 17.5)$

$1.08 + 1.95$

3.03

6. ***Problem Solver (21 points):*** Given the following problems, suggest one solution and give one benefit and one drawback of the solution.

EXAMPLE
Problem: long memory latencies
Solution: *Caches*
benefit: *low latency when the data is in the cache*
drawback: *when the cache misses, the latency becomes worse due to the cache access latency*
We would not accept solutions like: *"do not use memory", "use a slower CPU", etc*

Problem: too many conflict misses in the data cache
Solution: increase cache associativity
benefit: less conflict misses → may be
drawback: cache access latency and power will increase

Problem: too much traffic/contention on the bus to memory
Solution:
benefit:
drawback:

Problem: too many control hazards
Solution: branch prediction in hardware
benefit: can avoid some hazards
drawback: extra hardware/complexity

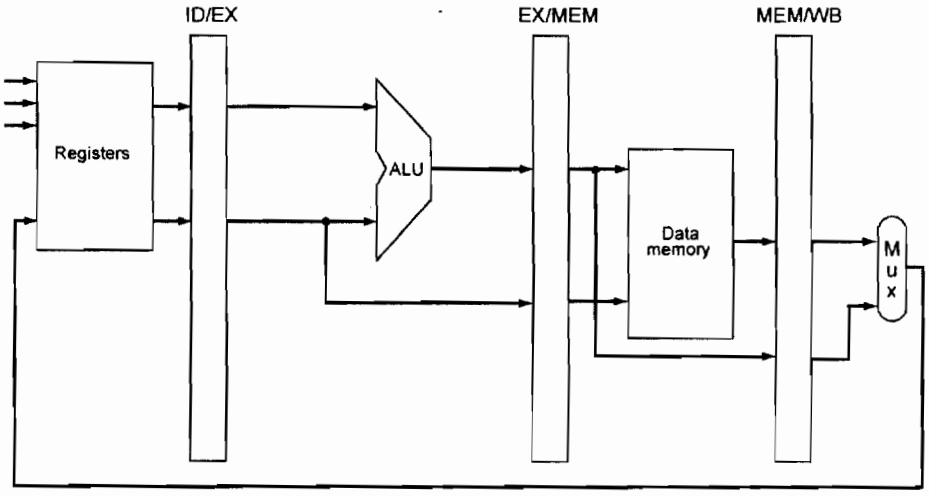
Problem: our use of daisy chaining to select a bus master is starving low priority devices
Solution:
benefit:
drawback:

Problem: our ripple carry adder is too slow
Solution: carry look ahead adder
benefit: faster computation of Cin's
drawback: more logic required, greater complexity

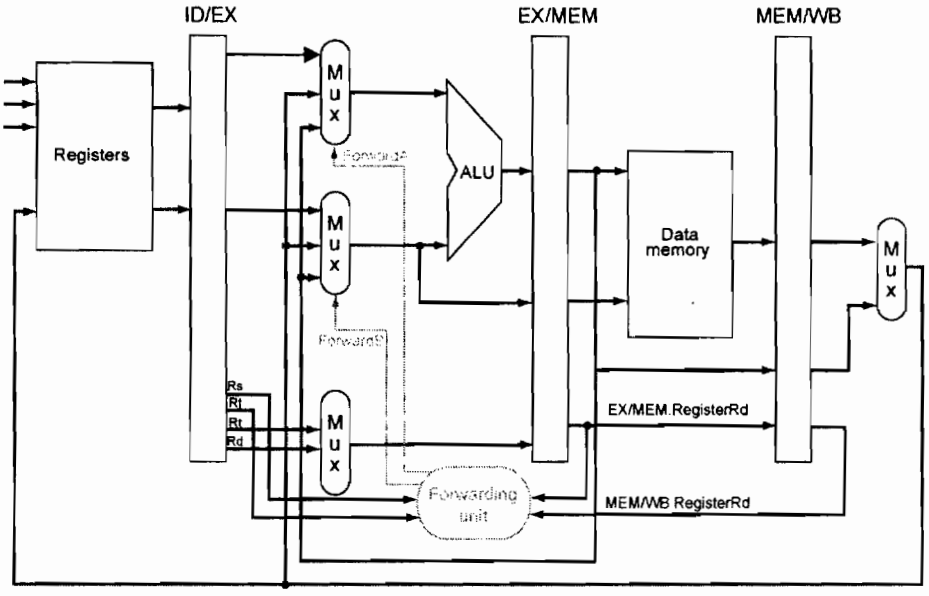
Problem: we want more instructions in the MIPS ISA
Solution: variable length ISA
benefit: more instructions bits, but instruction footprint can remain small
drawback: complex decoding

Problem: our page table takes up too much space in memory
Solution: 2 level page table
benefit: we can store parts of page table to disk
drawback: page faults are more expensive

7. ***A Staggering Blow (25 points):*** Consider the 5-stage pipelined architecture – we use forwarding to avoid pipeline stalls. The following figure demonstrates the addition of forwarding paths as we examined in class.



a. No forwarding

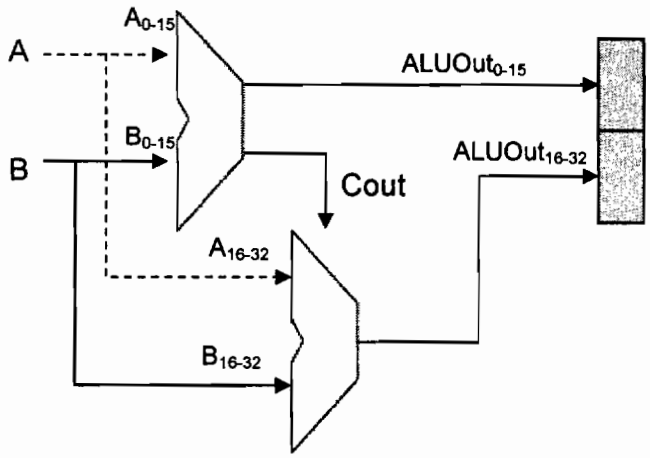


b. With forwarding

FIGURE 7.1

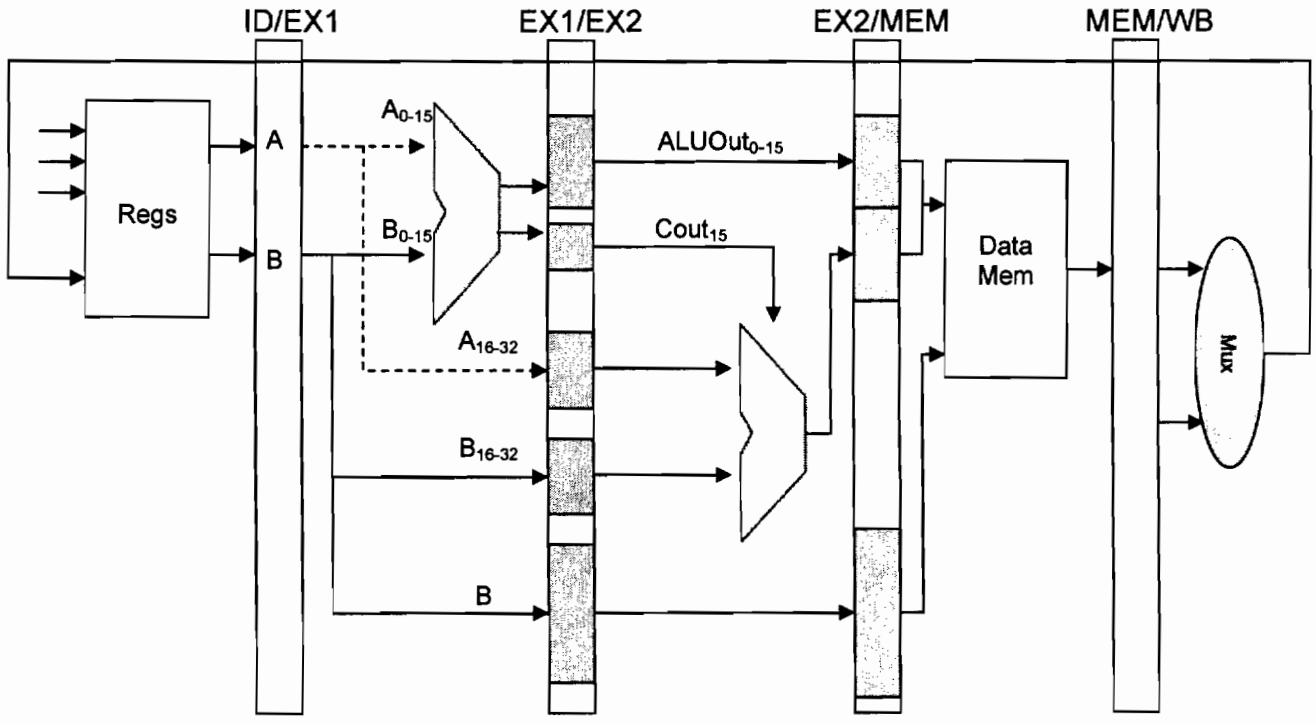
Suppose that the ALU is the bottleneck to increasing your clock rate. One solution (used on the Pentium 4) is to use a ***staggered*** ALU. This is where the ALU is pipelined over two stages. In the first stage, the first half of the computation is done. In the second stage, the second half of the computation is done. As in the above diagram, you will need to add forwarding hardware to this new 6-stage pipeline.

First, let's look at the staggered ALU:



There are two ALUs in this figure – both are 16-bit ALUs. Suppose for the purposes of this problem that we are **not** supporting *slt*. The first ALU computes the desired operation on the lower 16 bits of the 32 bit registers A and B. The carry out (Cout) from this ALU is used as the Cin for the second ALU, which computes the desired operation on the upper 16 bits of the 32 bit registers A and B. The results of both ALUs together make up the 32 result of the operation on A and B.

The following diagram demonstrates how these ALUs will fit into the pipeline.



Note that there is no forwarding shown here. You will want to be able to support back to back instructions – like the following sequence:

add $t0, $t1, $t2
and $t3, $t0, $t4

If each 16-bit ALU is placed in a separate pipeline stage, the value of $t0 should be forwarded from the *add* to the *and* without any stalling. First, the result of the lower 16 bit sum is forwarded to the *and* as it enters the first adder, and then the result of the upper 16 bit sum is forwarded to the *and* as it enters the second adder. We have started the datapath and forwarding logic for the diagram below, **just as was done for the 5 stage pipeline in Figure 7.1 above**. Complete this modification by adding muxes and wires – label all wires that you add. You do not need to design the internals of the forwarding units – this should be just as was done in Figure 7.1.