

University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Fall 2016

Instructors: Randy Katz, Bernhard Boser

2016-12-16



CS61C FINAL



After the exam, indicate on the line above where you fall in the emotion spectrum between “sad” & “smiley”...

<i>Last Name</i>	
<i>First Name</i>	
<i>Student ID Number</i>	
<i>CS61C Login</i>	cs61c-
<i>The name of your SECTION TA and time</i>	
<i>Name of the person to your LEFT</i>	
<i>Name of the person to your RIGHT</i>	
<i>All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61C who have not taken it yet. (please sign)</i>	

Instructions

- This booklet contains 10 numbered pages of text including the cover page.
- Turn off all cell phones, smartwatches, and other mobile devices and place out of reach. Remove hats & headphones. Place your backpacks, laptops and jackets on the sides of the room.
- You have **170 minutes** to complete this exam. The exam is closed book; no computers, phones, or calculators are allowed. You may use **three** handwritten 8.5"x11" page (front and back) crib sheet in addition to the MIPS Green Card, which we will provide.
- There may be partial credit for incomplete answers; write as much of the solution as you can. Keep in mind though, we will deduct points if your solution is far more complicated than necessary or interspersed with incorrect answers. When we provide a blank, please fit your answer within the space provided.
- Points are assigned by the approximate time to answer the question, 1 point = 1 minute. Pace yourself, and attempt every question for partial credit.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Total
Max Points	20	12	10	18	20	20	20	120

Q1: Potpourri (20 points)

1.1 PUE (Power Usage Effectiveness) (4 points)

In a Warehouse-Scale Computer, what happens to the PUE (Power Usage Effectiveness) when we make the IT equipment more efficient by cutting its power in half, but leave the infrastructure power unchanged? Circle the best answer choice below.

- a. The PUE is doubled
- b. The PUE is halved
- c. The PUE is unchanged
- d. The PUE is increased by 50%
- e. The PUE is decreased by 50%
- f. None of the above

1.2 Amdahl's Law (6 points)

Using deep learning, you programmed a neural network that is a perfect predictor of the stock market. One small problem: on your single-core laptop, the code takes so long that you get your predictions a week late. Benefitting from your innovation requires a 100x speedup.

Your program spends 20% of its time for setup and the remaining 80% number crunching. A superscalar can speed up the former, while a SIMD unit speeds up the latter. Furthermore, the execution speed is proportional to the number of cores, with negligible parallel processing overhead.

Calculate the speedup for each of the processor choices listed in the table below and the number of cores require to achieve the necessary 100x speedup.

Processor	Speedup of		Overall Speedup	# Cores
	setup	number crunching		
Laptop	1x	1x	1x	100
A	1x	4x	$100/40 = 2.5x$	40
B	4x	1x	$100/85 = 1.18x$	85
C	2x	8x	$100/20 = 5x$	20

1.3 Test and Set (10 points)

A new implementation of MIPS uses the `tas` instruction (“test and set”) for synchronization, rather than `ll` and `sc`.

`tas $rt off($rs)`

writes a 1 to memory location `off($rs)` and sets `$rt` to the old contents of `off($rs)`.

Finish writing the following code (marked by the black arrows) to grant exclusive access to a “critical section” in MIPS assembly. You may choose to first write pseudo-C-code, and then translate that code to assembly. Only the assembly will be graded. Use `tas`, not `ll` and `sc` in your code! Fill in the code below at the places marked by arrows. You may not need all lines. You may use labels in MIPS assembly.

Initialization (executed once at program start):

Pseudo C code	MIPS assembly
<code>// declaration of lock</code>	<code># declaration of lock</code>
<code>// (program initialization)</code>	<code># \$s1 points to lock</code>
<code>int lock = 0</code>	<code>sw \$zero, 0(\$s1)</code>

Critical Section:

<code>// acquire lock</code>	<code># acquire lock</code>	
<code>while (tas(lock) == 1);</code>	<code>addi \$t1, \$zero, 1</code>	←
	<code>Try: tas \$t2, 0(\$s1)</code>	
	<code>beq \$t1, \$t2, Try</code>	
<code>// critical section</code>	<code># critical section</code>	
<code>// exclusive access</code>	<code># exclusive access</code>	
<code>// release lock</code>	<code># release lock</code>	
<code>lock = 0</code>	<code>sw \$zero, 0(\$s1)</code>	←

Q2: RAID and ECC (12 points)

A RAID 2 Disk Array organization interleaves data on a fine-grain basis and protects it using a Hamming Code with additional parity bits mixed in among the data bits. As a reminder, a Hamming ECC Code with three parity bits and four data bits redundantly encodes data as follows:

	1	2	3	4	5	6	7
	p1	p2	d1	p4	d2	d3	d4
p1	x		x		x		x
p2		x	x			x	x
p4				x	x	x	x

For the purposes of this question, **assume that parity is calculated using ODD rather than EVEN parity.**

Four disks worth of data is nibble (4-bits at a time) interleaved across four disks with an additional three parity bits. Disk #4 (p4) suffers a hard crash and cannot be read. The data on the remaining disks for the first nibble looks like this, with ? indicating the bits that cannot be read:

0	0	0	?	1	1	0:	1	(2 points)
1	0	0	?	1	0	1:	1	(2 points)
0	0	1	?	0	0	0:	1	(2 points)
1	1	1	?	0	0	1:	0	(2 points)

Fill in the correct missing bits in the underlined spaces at the right (8 points).

If a single disk can do 50 small (i.e., sector-sized) random read or write operations per second, how many read/write operations can the 4 D + 3 P RAID 2 disk array perform per second? Circle the correct answer below:

50 150 200 350 (2 points)

A single disk can transfer large contiguous blocks of data at the rate of 100 Mbps. Circle the correct **effective** large contiguous block **real** data transfer rate of the 4 D + 3 P disk array below (in Mbps):

100 300 **400** 700 (2 points)

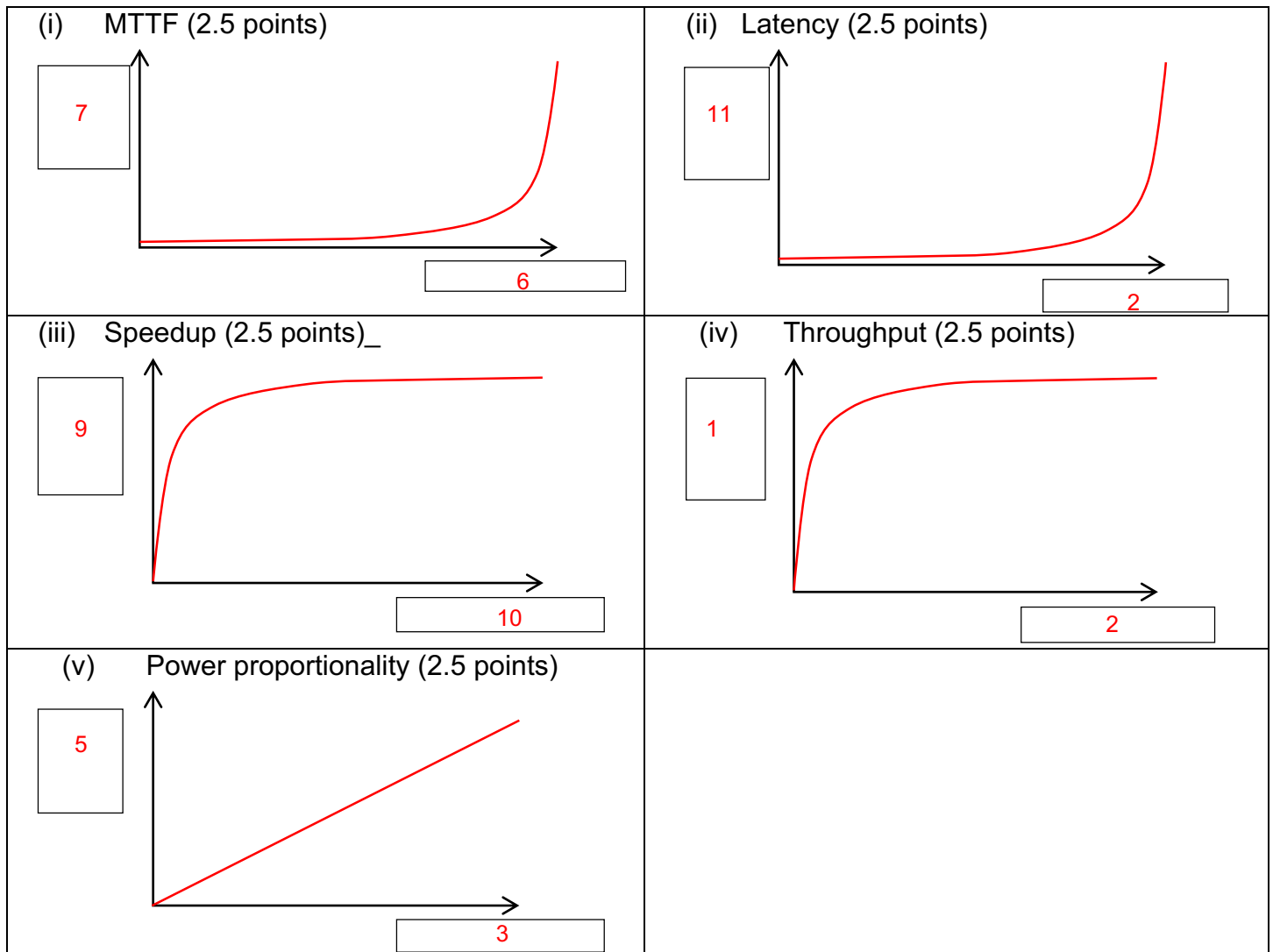
Q3: Always label your axes (12.5 points)

Label the X and Y axes of the following graphs.

The possible options are:

1. bytes per second
2. operations per second
3. percent utilization
4. power consumed
5. processor power
6. time
7. probability of failure
8. ratio of ideal to actual performance
9. ratio of actual to ideal performance
10. thread count
11. time per operation

Some options can be used more than once and some options may never be used.



Q4: Spark (15.5 points)

Given a set of documents, fill in the following functions so the code will do the following sequentially:

1. Count the total number of times each word shows up in a document
2. Finds the document that the words show up in the most times. If there is a tie between two documents, choose the document with the lower document ID.
3. Finally returns a set of key value pairs where the **key** is the document ID and the **value** is a list of all the words that showed up the most frequently in that given document.

```
def find_all_words(document):
    # Returns a list of all the words in a document. Words are converted
    # to lower-case, represented as strings. Assume this is implemented.

def flatmap_func(pair):
    document_id = pair[0] # This is an integer
    document = pair[1]
    all_words = find_all_words(document)
    return [(word, document_id), 1) for word in all_words]

def count(v1,v2):
    return v1 + v2

def map_func(pair):
    key = pair[0][0]
    value = (pair[1],pair[0][1])
    return (key, value)

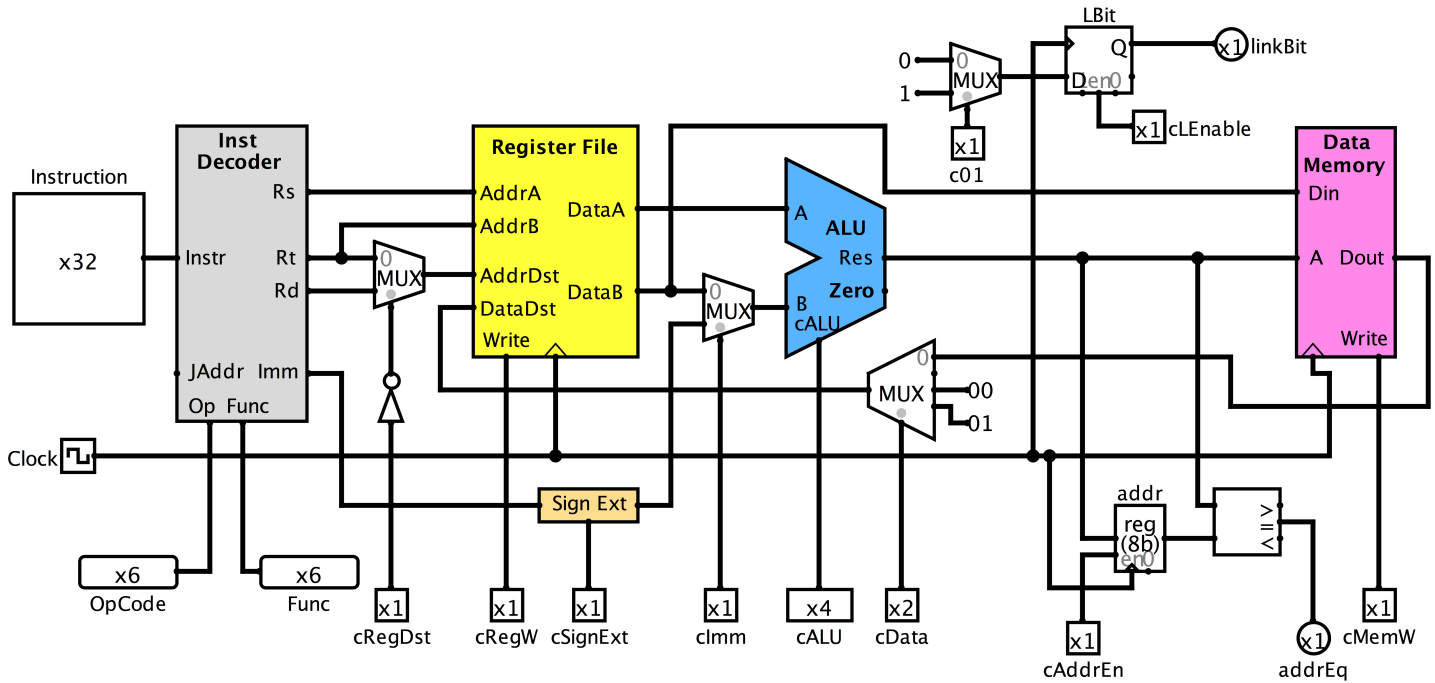
def compare(v1,v2):
    if v1[0] > v2[0]:
        return v1
    if v2[0] > v1[0]:
        return v2
    if v1[1] > v2[1]:
        return v2
    else:
        return v1

def transform_func(pair):
    key = pair[1][1]
    value = pair[0]
    return (key, value)

if __name__ == "main":
    rdd = sc.parallelize(documents) \
        .flatMap(flatmap_func) \
        .reduceByKey(count) \
        .map(map_func) \
        .reduceByKey(compare) \
        .map(transform_func) \
        .groupByKey().collect()
```

Q5: MIPS Datapath (20 points)

Shown below is a simplified single-cycle MIPS datapath with circuitry added to implement the **ll** and **sc** instructions. Lbit is a 1-Bit D-flip-flop that can be set and reset with **c01** when **cLEnable=1**. A 32-Bit address latch, enabled by **cAddrEn**, and an address comparator have been added also.



Complete the MIPS control table on the next page for instructions **lw**, **ll**, and **sc**. You may not need all rows.

Hint: Consider what is the purpose of **ll** and **sc**, namely to guarantee exclusive access to a critical region. Any action that could potentially jeopardize this guarantee must cause **sc** to fail, while still accepting all valid attempts to acquire exclusive access.

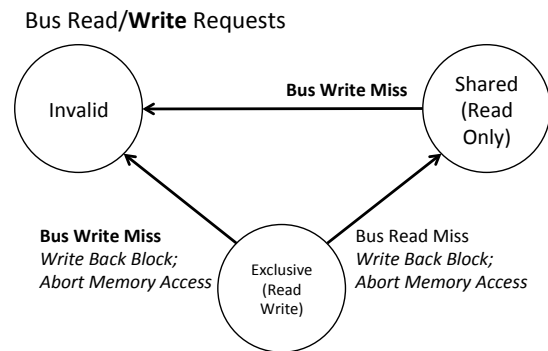
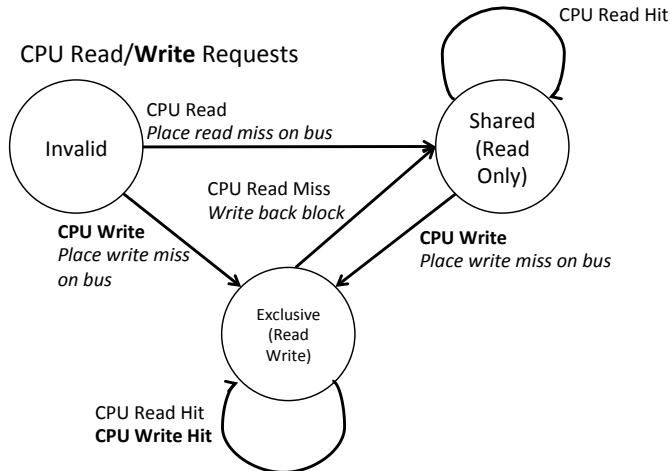
OpCode	addrEq	linkBit	cData	cAddrEn	cLEnable	c01
lw	0	X	00	0	0	X
lw	1	X	00	0, 1, X*	1	0
ll	X	0	00	1	1	1
ll	X	1	00	0	1	0
sc	1	1	11	0, 1, X*	1	0
sc	0	X	10	X	1	0
sc	X	0	10	X	1	0

* accept either

For this problem, all that was necessary to receive full credit were the lines marked above. Extra lines added with these signal combinations were permitted as long as no contradictions were made.

Q6: Cache-Coherency (20 points)

The two state diagrams below show a three-state cache coherency protocol based on **Invalid**, **Shared**, and **Exclusive** states. The first diagram on the left shows the cache controller from the perspective of the processor and the second on the right from the perspective of the snooping memory bus.



From the CPU side, a memory block not in the cache starts in the **Invalid** state. A *CPU read* causes a memory access and a transition to the **Shared** state. You should interpret this state as implying that the addressed memory block could be in another cache. Subsequent reads hit in the cache. A *CPU write* causes a transition from **Invalid** or **Shared** to the **Exclusive** state, and a write miss transaction is placed on the memory bus.

This causes other caches in the **Shared** state for this block to move to the **Invalid** state, and for a cache in the **Exclusive** state to trigger a write back to memory as it transitions to the **Invalid** state. In this latter case, the original memory access is aborted, causing the cache with the write miss to restart its memory access. This is shown in the Bus side state diagram, described next.

From the Bus side, a write miss moves either the **Shared** or **Exclusive** states to Invalid. If the latter, the memory block must first be written back to memory and the original CPU that generated the write miss must restart its memory transaction. If the bus signals a read miss, the cache block transitions from **Exclusive** to **Shared** after the block is written back to memory. Again, the reading cache must restart its memory access.

Invalid State: No memory block in the cache block.

Shared State: The memory block in the cache block could also be in other caches.

Exclusive State: The memory block in the cache block is not in any other cache

Complete the following table.

Possible states are Invalid (I), Shared (S), and Exclusive (X).

Possible addresses are A1 and A2, which happen to map to the same cache block.

Possible values are 10, 20, and 40.

Possible (Bus) Actions are Write Miss (WM), Read Miss (RM), Read Data (RD), Write Data (WD).

Possible Processors are P1 and P2 for CPU1 and CPU2 respectively.

Note:

- Before the first write operation, all cache entries are I (for Invalid) and all memory data is all 0.
- Not every row and column needs to have an entry
- Some operations may trigger multiple state and bus action; fill in the boxes with the correct sequence from top to bottom.

Step	P1			P2			Bus				Memory	
	State	Addr	Value	State	Addr	Value	Action	Proc	Addr	Value	Addr	Value
P1: write 10 to A1	X	A1	10				WM	P1	A1			
P1: read A1	X	A1*	10*									
P2: read A1	S	A1*	10*	S	A1		RM	P2	A1			
				S	A1	10	WD	P1	A1	10	A1	10
				S	A1	10	RD	P2	A1	10	A1*	10*
P2: write 20 to A1	I	A1*	10*	X	A1	20	WM	P2	A1		A1*	10*
P2: write 40 to A2				X	A2	40	WM	P2	A2		A1*	10*
							WD	P2	A1	20	A1	20

P1 read A1: Trivial.

P2 read A1: P2 transitions to the shared state to read A1. From the bus side, we see this memory access gets aborted until P1 writes back the data block to memory. After doing so, the memory access is attempted again.

P2 writes 20 to A1: A write miss is placed on the bus. The shared state of the cache line that the A1 block in P1 has is invalidated.

P2 writes 40 to A2: "From the bus side, a write miss moves either the **shared** or **exclusive** states to invalid. If the latter, the memory block must first be written back to memory and the original CPU that generated the write miss must restart its memory transaction."
Keep in mind that A1 and A2 happen to "map to the same cache block."

*: Elements not necessary (must be left blank if not included) but students still received points if they included or left these out. Some students included memory values in each cell whereas some only indicated when there were updates to values.

Q7: Virtual Memory and Caches (20 points)

A hypothetical processor has 1-MiByte physical memory organized in 16 pages and separate (single-level) instruction and data caches with 16-Byte blocks addressed with 4-bit index fields. Virtual addresses are 19 bits wide. The tables below show partial contents. The data stored in the TLB, caches and memory is consistent. On miss, the TLB fills from the D-Cache.

TLB

Valid	Tag/VPN	PPN/DPN
1	0x3	0x7
1	0x7	0xb

Memory

VPN	Address	Content (32b)
	0x20224	0x1000,0001
	0x20220	0x0000,0010
7	0x2021c	0x1000,000b
6	0x20218	0x0000,000a
5	0x20214	0x1100,0006
4	0x20210	0x1000,0002
3	0x2020c	0x1000,0007
2	0x20208	0x0000,0004
1	0x20204	0x1100,0000
0	0x20200	0x1000,0004
	0x201fc	0x1000,0005

Caches (only valid entries shown)

I-Cache		D-Cache	
Index	Tag	Index	Tag
0x0	0x3b5	0x1	0x202
0x0	0x400	0x0	0xb50
0x0	0xbb5	0x5	0x7b5
0x0	0x700	0x1	0x011

The page table is at address 0x20200. Each 32-bit entry is encoded as follows:

Bit 28	Bit 24	Bits 23 ... 0
Valid bit	Swap bit	PPN/DPN

A Valid bit = 1 indicates that the page can be read or written to by the program. Accessing a page that is not valid results in a segmentation fault.

A Swap bit = 1 indicates that the page has been swapped out to disk.

Now let's assume that the processor executes load instructions **lw \$rd, 0(\$rs)** where the instruction is at location I-addr and the the value of **\$rs** is D-addr. I-addr and D-addr are specified in the table below. For each **lw** instruction calculate the physical addresses of the instruction and data, the number of TLB-, instruction-, and data-cache misses, and the number of page faults.

Add comments describing the outcome of each load instruction. Only leave the comments field blank in the case of 0 misses and 0 faults.

In the case of an error, you should specify possible errors (e.g. segmentation fault). Feel free to leave the other fields blank in the case of an error as well.

Note that each instruction can suffer several faults. Treat each instruction in isolation, i.e. assume that the contents of the memory and caches are as indicated in the tables above and have been not changed by prior instructions.

lw		I-paddr	D-paddr	# misses			page faults	Errors / Comments
I-addr	D-addr			TLB	I\$	D\$		
0x30000	0x75004	0x70000	0xb5004	0	0	0	0	
0x70000	0x66544	0xb0000	seg fault					segmentation fault
0x35008	0x7500c	0x75008	0xb500c	0	1	0	0	
0x00200	0x70008	0x40200*	0xb0008	1	1	2	0	PTE and D not in D\$
0x00203	0x7000a	unaligned						unaligned access
0x4000c	0x7b550	0x2000c*	0xbb550	1	1	1	0	instr TLB miss
0x14000	0x3b500	0x04000	0x7b500	1	1	2	1	instr page fault

Some students rightfully realized that I-addr's interaction with the TLB would potentially affect the TLB hit/miss for D-addr that took place immediately upon executing the instruction. Students did not have to acknowledge/observe this to get full credit.