16.482 / 16.561: Computer Architecture and Design

Fall 2013

Final Exam Solution

1. (16 points) ***Dynamic scheduling and speculation***

a. (4 points) Explain why instruction results in a speculative, dynamically scheduled processor are stored in the reorder buffer rather than directly being written to the register file or memory. How does this setup impact register renaming (in other words, how is register renaming handled differently in a dynamically scheduled processor with speculation than it is without speculation)?

Solution: *In order to allow the ability to recover from branch mispredictions, instructions cannot update the permanent state of the processor until it is known that they are correct. However, for performance reasons, instructions should execute and complete out of order where possible. Instruction results therefore must be stored between the write back (WB) and commit (C) stages; the ROB is used to hold these results during that time.*

Because results may be read from the ROB between WB and C, registers are renamed based on their ROB entry, not the reservation station to which the instruction is issued.

b. (12 points) You are given the following piece of code to run on a dynamically scheduled machine that allows speculation:

```
Loop:   L.D    F2, 0(R1)
        MUL.D  F4, F4, F2
        DADDUI R1, R1, #8
        BNE    R1, R2, Loop
        S.D    F4, 8(R1)
```

Assume the following latencies:

- L.D and S.D have 2 cycle latencies (1 cycle execution, 1 cycle in memory)
- MUL.D has a 5 cycle latency
- All other operations have 1 cycle latencies

Also, assume:

- The loop executes three times.
- The branch at the end of the loop is always predicted taken.
    - If there is a misprediction, you should accurately show how it would be handled. Unknown instructions should be shown as "?" in the Inst. column of your diagram
- Integer and floating point operations are handled in separate functional units.
- You only have one common data bus available—if two instructions need to use the common data bus, the earliest instruction in terms of program order has priority.
- You only have the ability to commit one instruction per cycle.

How many cycles will this sequence take, from the time the first instruction fetches to the time the last instruction commits? Complete in the pipeline diagram on the following page to support your answer. Use the space below and the back of this page for any additional work.

QUESTION 1b SOLUTION

Inst.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
`L.D F2,0(R1)`	IF	IS	EX	M	WB	C																		
`MUL.D F4,F4,F2`		IF	IS	**S**	E1	E2	E3	E4	E5	WB	C													
`DADDUI R1,R1,#8`			IF	IS	EX	WB	--	--	--	--	--	C												
`BNE R1,R2,Loop`				IF	IS	EX	--	--	--	--	--	--	C											
`L.D F2,0(R1)`					IF	IS	EX	M	WB	--	--	--	--	C										
`MUL.D F4,F4,F2`						IF	IS	**S**	**S**	E1	E2	E3	E4	E5	WB	C								
`DADDUI R1,R1,#8`							IF	IS	EX	**S**	WB	--	--	--	--	--	C							
`BNE R1,R2,Loop`								IF	IS	EX	--	--	--	--	--	--	--	C						
`L.D F2,0(R1)`									IF	IS	EX	M	WB	--	--	--	--	--	C					
`MUL.D F4,F4,F2`										IF	IS	**S**	**S**	**S**	E1	E2	E3	E4	E5	WB	C			
`DADDUI R1,R1,#8`											IF	IS	EX	WB	--	--	--	--	--	--	--	C		
`BNE R1,R2,Loop`												IF	IS	EX	--	--	--	--	--	--	--	--	C	
`L.D F2,0(R1)`													**IF**	**IS**										
`MUL.D F4,F4,F2`														**IF**										
`S.D F4,8(R1)`															IF	IS	EX	--	--	--	--	--	--	C

Note: *Stalls due to data dependences shown in **red**, as are squashed instructions. Stalls due to CDB conflicts shown in **blue**. Cycles spent waiting between WB and C are indicated by "--".*

2. (16 points) ***Multithreading***

a. (4 points) List one benefit of fine-grained multithreading over coarse-grained multithreading and one benefit of coarse-grained multithreading over fine-grained multithreading.

Solution:

Fine-grained benefits: maximum fairness among all threads

Coarse-grained benefits: best single thread performance for thread that can run without long-latency stalls.

b. (12 points) Given the 3 threads below, determine how long they take to execute using simultaneous multithreading on a processor with the following characteristics:

- 4 functional units: 2 ALUs, 1 memory port (load/store), 1 branch
  - The ALUs can handle MUL.D operations
  - The branch unit can handle jumps
- In-order execution
- The following instruction latencies:
  - L.D/S.D: 3 cycles (1 EX, 2 MEM)
  - MUL.D: 4 cycles
  - ADD.D/SUB.D: 2 cycles
  - All other operations: 1 cycle
- Thread 1 is the preferred thread, followed by Thread 2 and Thread 3.

Assume the BEQ in Thread 3 is not taken.

Your solution should use the table on the next page, which contains columns to show each cycle, the functional units being used during that cycle, and space to indicate stall cycles. Note that you only need to label a cycle as a stall if all active threads are stalled. **Clearly indicate which thread contains each instruction when completing the table.**

NOTE: The last page of the exam contains an extra copy of the latencies and threads.

Thread 1:

```
L.D   F0, 0(R1)
MUL.D F4, F0, F2
ADD.D F6, F4, F10
S.D   F6, 16(R1)
SUB.D F10, F6, F2
S.D   F10, 32(R1)
DADDUI R1, R1, #-48
BNEZ  R1, loop
```

Thread 2:

```
L.D   F2, 0(R1)
ADD.D F4, F4, F0
SUB.D F6, F6, F0
MUL.D F8, F2, F6
MUL.D F10, F2, F4
ADD.D F0, F10, F8
DSUB  R1, R1, R2
BNEZ  R1, loop
```

Thread 3:

```
LW    R1, 0(R2)
BEQ   R1, R3, t1
ADD.D F0, F2, F4
MUL.D F6, F2, F4
S.D   F0, 0(R1)
S.D   F6, 8(R1)
JR    R31
```

QUESTION 2b SOLUTION

Cycle	ALU1	ALU2	Mem1	Branch
1			T1: L.D	
2	T2: ADD.D	T2: SUB.D	T2: L.D	
3			T3: LW	
4	T1: MUL.D			
5	T2: MUL.D	T2: MUL.D		
6	T3: ADD.D	T3: MUL.D		T3: BEQ
7				
8	T1: ADD.D		T3: S.D	
9	T2: ADD.D	T2: DSUB		
10	T1: SUB.D		T1: S.D	T2: BNEZ
11			T3: S.D	T3: JR
12	T1: DADDUI		T1: S.D	
13				T1: BNEZ

3. (16 points) ***Cache operation***

You are given a system which has a 16-byte, write-back cache with 4-byte blocks. The cache is direct-mapped. The system uses 8-bit addresses, and the cache is initially empty.

a. (12 points) Assume the initial memory state shown below for the first 32 bytes:

Address		Address		Address		Address	
0	27	8	19	16	22	24	13
1	3	9	78	17	5	25	24
2	20	10	9	18	15	26	21
3	11	11	12	19	13	27	7
4	5	12	1	20	49	28	18
5	12	13	0	21	77	29	8
6	14	14	63	22	15	30	55
7	2	15	98	23	44	31	99

For each access in the sequence listed below, fill in the cache state, indicate what register (if any) changes, and indicate if any memory blocks are written back and if so, what addresses and values are written. The cache state should carry over from one access to the next.

Access	Modified register	Cache state							Modified mem. block
		V	D	Tag	Data				
`lb $t0,6($zero)`	*$t0 = 14*								*None*
		1	***0***	***0000***	***5***	***12***	***14***	***2***	
`sb $t0,4($zero)`	*None*								*None*
		1	***1***	*0000*	***14***	*12*	*14*	*2*	
`lb $t1,23($zero)`	*$t1 = 44*								*Bytes 4-7 = [14 12 14 2]*
		1	***0***	***0001***	***49***	***77***	***15***	***4***	
`sb $t1,14($zero)`	*None*								
		1	*0*	*0001*	*49*	*77*	*15*	*4*	
		1	***1***	***0000***	***1***	***0***	***44***	***98***	

3 (continued)

b. (4 points) Assume this cache is the Level 1 cache in a hierarchy with the following characteristics:
    - Level 1 cache: 4 ns access time, 92% hit rate
    - Level 2 cache: 12 ns access time, 96% hit rate
    - Memory: 100 ns access time, 99% hit rate
    - Disk: 1000 ns access time

Assume that the processor cycle time is 2 ns. How many cycles will an average memory access take? **Show all work for full credit.**

Solution: *Simply plug into the AMAT equation. Note that this solution gives you the time in nanoseconds and then converts to cycles; you could recognize that the L1 cache takes 2 cycles, L2 takes 6, memory takes 50, and disk takes 500 and just use those numbers:*

$$
\begin{aligned}
AMAT &= (\text{Hit time}) + (\text{Miss rate}) * (\text{Miss penalty}) \\
&= 4 + (.08) * AMAT_{L2} \\
&= 4 + (.08) * (12 + (.04) * AMAT_{Memory}) \\
&= 4 + (.08) * (12 + (.04) * (100 + (.01) * AMAT_{disk})) \\
&= 4 + (.08) * (12 + (.04) * (100 + (.01) * (1000))) \\
&= 4 + (.08) * (12 + (.04) * (100 + 10)) \\
&= 4 + (.08) * (12 + 4.4) \\
&= 4 + 1.312 = 5.312 \text{ ns}
\end{aligned}
$$

5.312 ns / 2 ns per cycle → *2.656 cycles* → ***must round up to 3 cycles***

4. (15 points) ***Virtual memory***
Answer the following questions about a process using the page table below:

Virtual page #	Valid bit	Reference bit	Dirty bit	Frame #
0	0	0	0	--
1	1	1	1	2
2	1	0	1	3
3	0	0	0	--
4	1	0	0	1
5	0	0	0	--

a. (3 points) Which pages are candidates to be evicted on a page fault? Under what conditions would a page fault cause one of these pages to be evicted?

Solution: *Any valid page with a reference bit equal to 0 is a candidate for eviction—in this case, pages 2 and 4. Evictions only occur, however, if there are no free frames in memory. If a page fault occurs when a free frame is available, the free frame is filled without evicting anything.*

b. (6 points) Assuming 2 KB pages, what physical addresses would the virtual addresses below map to? Note that some virtual addresses may not have a valid translation, in which case you should note that address causes a page fault.

Note: 2 KB = 2^{11} byte pages have an 11-bit page offset and a (16-11) = 5-bit page number.

- 0x2010

Solution: *0x2010 = **0010 0**000 0001 0000_2 → page # = 00100_2 = 4*
*Frame # = 1 = 00001 → physical address = **0000 1**000 0001 0000_2 = **0x0810***

- 0x1FFE

Solution: *0x1FFE = **0001 1**111 1111 1110_2 → page # = 00011_2 = 3 → **page fault***

- 0x0A1B

Solution: *0x0A1B = **0000 1**010 0001 1011_2 → page # = 00001_2 = 1*
*Frame # = 2 = 00010_2 → physical address = **0001 0**010 0001 1011_2 = **0x121B***

4 (continued)

c. (6 points) Fill in the table at the bottom of the page to show the **final** state of the page table after the following sequence of accesses. Assume main memory has 4 frames, numbered 0-3, and frame 0 is initially free. The initial state of the page table is repeated below for your reference.

ACCESS SEQUENCE

- Read page 3 → *Page fault; page 3 allocated to frame 0; Valid = Ref = 1; Dirty = 0*
- Write page 4 → *Dirty = Ref = 1; all other bits unchanged*
- Write page 5 → *Page fault; page 2 evicted (only valid page with Ref = 0)*
  Page 5 allocated to frame 3; Valid = Ref = Dirty = 1
- Read page 4 → *No changes—page 4 is valid and reference bit already = 1*

INITIAL PAGE TABLE STATE:

Virtual page #	Valid bit	Reference bit	Dirty bit	Frame #
0	0	0	0	--
1	1	1	1	2
2	1	0	1	3
3	0	0	0	--
4	1	0	0	1
5	0	0	0	--

FINAL PAGE TABLE STATE: *Changes are in **bold***

Virtual page #	Valid bit	Reference bit	Dirty bit	Frame #
0	*0*	*0*	*0*	--
1	*1*	*1*	*1*	*2*
2	***0***	***0***	***0***	***--***
3	***1***	***1***	***0***	***0***
4	***1***	***1***	***1***	*1*
5	***1***	***1***	***1***	***3***

5. (13 points) ***Cache optimizations***

Use the following page to answer **only 1 of the following 2 questions—part (a) or part (b)**. **Clearly indicate** which question you have chosen to answer at the top of the page.

a. ***(Multi-banked/non-blocking caches)*** Assume we have a system containing 16 blocks of memory, numbered 0-15. This system has an 8-line, direct-mapped cache that is initially empty. Assume we have a program that accesses 10 of these blocks in the following order, with one access initiated per cycle unless a stall occurs:

0, 1, 4, 14, 7, 5, 9, 8, 15, 6

Note that all of these accesses are misses; each miss takes 20 cycles to handle.

i. (3 points) Calculate the total time for these 10 accesses if the cache is not split into banks and is therefore a blocking cache (i.e., only one miss can be handled at a time).

Solution: *Since each access causes the cache to block, they cannot be overlapped. The total time is therefore: (20 cycles per access) * (10 accesses) =* ***200 cycles.***

ii. (5 points) Calculate the total time for these 10 accesses if the cache is divided evenly into four banks. Assume the blocks are not interleaved sequentially, so blocks are mapped to cache lines using normal direct mapping. In other words, B0 maps to cache line 0, B1 to cache line 1, and so on.

Solution: *Remember, accesses to different banks can be overlapped and will start in consecutive cycles (for example, if an access to one bank starts in cycle i, the next access can start in cycle i+1). Since we are not using sequential interleaving, the blocks are mapped to cache lines—and therefore to banks—as shown in the table below:*

Line #	Blocks mapped to this line	
0	0, 8	Bank 0
1	1, 9	
2	2, 10	Bank 1
3	3, 11	
4	4, 12	Bank 2
5	5, 13	
6	6, 14	Bank 3
7	7, 15	

Now, we can determine the total access time by looking at each access and determining which ones can be overlapped. Note that up to 4 accesses can be overlapped—1 per bank.

5.a.ii (continued)

*The table below shows each access, the bank it accesses, and the start and end time of those accesses. In total, the sequence takes **105 cycles** given this cache organization.*

Block #	Bank	Start cycle	End cycle
0	0	1	20
1	0	21	40
4	2	22	41
14	3	23	42
7	3	43	62
5	2	44	63
9	0	45	64
8	0	65	84
15	3	66	85
6	3	86	105

iii. (5 points) Calculate the total time for these 10 accesses if the cache is divided evenly into four banks and the blocks are interleaved sequentially across those four banks.

Solution: *With sequential interleaving, we have the following mapping:*

Line #	Blocks mapped to this line	
0	0, 8	Bank 0
1	4, 12	
2	1, 9	Bank 1
3	5, 13	
4	2, 10	Bank 2
5	6, 14	
6	3, 11	Bank 3
7	7, 15	

*As shown in the table below, the sequence will take **66 cycles** given this cache organization.*

Block #	Bank	Start cycle	End cycle
0	0	1	20
1	1	2	21
4	0	21	40
14	2	22	41
7	3	23	42
5	1	24	43
9	1	44	63
8	0	45	64
15	3	46	65
6	2	47	66

b. ***(Early restart/critical word first)*** Say we have a program running on a 32-bit processor in which a certain cache block is evicted from the cache before every access, making every access to this block a cache miss. The block contains 512 bytes. Assume every word (i.e., every 4 bytes of data) in the block is accessed exactly once.

If main memory is capable of supplying a word to the cache every 100 µs (1 µs = 10^{-6} s), calculate the total time required to access **all** words in the cache block using each of the following policies to fill the cache block on a miss:

- Sequential cache block fill (i.e., start with word 0 and fill all words in block)
- Early restart (without critical word first)
- Critical word first with early restart

Solution: *First, note that the processor word size is 32 bits = 4 bytes. Since we're given the amount of time required to supply a word to the cache, it's important to know that each cache block contains 512 / 4 = 128 words. Since we're told that every word is accessed exactly once, that means this sequence contains 128 accesses.*

Considering each of the fill policies:

- *Sequential cache block fill: Every access requires the entire block to be filled before the processor starts; the total time for the sequence is therefore:*

  *(100 µs/word) * (128 words/access) * (128 accesses) =* ***1638400 µs = 1.6384 s***

- *Early restart (without critical word first): This organization still uses sequential block fill; however, the processor can be restarted as soon as the desired word has been fetched. Accessing the first word takes just 100 µs; accessing the last word takes 12800 µs. The total time for the sequence is therefore:*

  100 µs + 200 µs + … + 12700 µs + 12800 µs = ***825600 µs = 825.6 ms***

  *(Note: I found a shortcut to solve this sum; namely, that the combined time required to fetch word i and word (129-i) = 12900 µs. In other words, fetching words 1 and 128 (the first & last words), 2 and 127, 3 and 126, and so on. If you recognize that you have 64 such pairs, you can solve the problem by multiplying (12900 µs / pair) * (64 pairs).)*

- *Critical word first with early restart: In this case, every desired word is fetched immediately, and the processor is restarted when the word is retrieved. The total time for the sequence is therefore:*

  *(100 µs / word) * (1 word/access) * (128 accesses) =* ***12800 µs = 12.8 ms***

6. (10 points) ***RAID***

You are working with a 5-disk RAID array that contains a total of 15 sectors; the exact sector configuration depends on the RAID level used. In all cases, twelve of the fifteen sectors (S0-S11) will hold data, while the remaining three sectors (P0-P2) hold parity information. Large reads and writes (reads/writes that access an entire stripe in the array) take 300 ms, small reads (reads involving only a single disk) take 150 ms, and small writes (writes involving 1 data disk + 1 parity disk) take 200 ms.

Given the following sequence of sector reads and writes, determine the time required if the array is configured with RAID 3, RAID 4, and RAID 5. Assume the following:

- Requests are queued in such a manner that two consecutive operations may proceed simultaneously if they do not share any disk within the array.
- If two disks, D_x and D_y, are in use, and the access to D_x finishes before the access to D_y, a new operation may start immediately assuming it does not involve D_y.
- Multiple accesses to the same stripe may overlap if they don't use the same disk.

1. read S0
2. write S5
3. write S8
4. read S9
5. read S3
6. write S7
7. read S11
8. write S1

In each case, show the organization of the array to support your answer. The next page contains additional space to solve this problem.

Solution:

- *In RAID 3, only large reads and writes are allowed. Since each operation involves every disk, no operations may be overlapped, and the total time for 8 large reads and writes is 8 x 300 ms =* ***2400 ms = 2.4 s****.*
- *In RAID 4, you may perform small reads, but only large writes. Therefore, any two consecutive reads that do not use the same disk can proceed in parallel. Nothing may be overlapped with a write. We assume the following organization:*

Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
S0	S1	S2	S3	P0
S4	S5	S6	S7	P1
S8	S9	S10	S11	P2

Operation	Start time	End time	Notes
read S0	1	150	Small read followed by write—no overlap
write S5	151	450	
write S8	451	750	
read S9	751	900	Sectors are on different disks, so small reads can be overlapped
read S3	751	900	
write S7	901	1200	
read S11	1201	1350	Small read followed by write—no overlap
write S1	1351	1650	

*In RAID 4, this sequence takes **1650 ms = 1.65 s.***

- *In RAID 5, both small reads and writes are allowed; we can therefore overlap any two consecutive operations that do not share the same disk. Remember RAID 5 also involves interleaved parity to make small writes possible, so the organization changes as follows:*

Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
S0	S1	S2	S3	P0
S4	S5	S6	P1	S7
S8	S9	P2	S10	S11

Note also that the problem states we can start a new transaction any time an existing transaction ends, provided the new transaction does not use the same disk as a currently executing transaction. Note that we must be careful. Although RAID 5 does enable small writes, these operations use two disks—the disk being written and the parity disk for that stripe. The solution to this part of the problem therefore becomes more complex and can best be described by tracking start and end times for each operation:

Operation	Start time	End time	Notes
read S0	1	150	Different disks—overlap allowed.
write S5	1	200	
write S8	151	350	Can start when read to S0 finishes.
read S9	201	350	Can start when write to S5 finishes. Allow multiple accesses in same stripe if different disks involved.
read S3	351	500	Cannot overlap with next write—both use Disk 4.
write S7	501	700	Cannot overlap with next read—both use Disk 5.
read S11	701	850	Cannot overlap with next write—both use Disk 5.
write S1	851	1050	

*In RAID 5, this sequence takes **1050 ms = 1.05 s.***

7. (14 points) ***Multiprocessors***
Note: To complete this problem, you must solve part (a) + either part (b) or (c).

a. (4 points) Say you have a dual processor system running two separate programs that share data but use different variable names. The following pieces of code are executed in the order shown, with the side effects listed:
   - P0: `x = 12;` (P0 writes x, invalidates copy of that block in P1's cache)
   - P1: `a = b;` (P1 experiences cache miss while reading b)

Assume that the cache miss for P1 is a result of the invalidation from P0. Under what conditions is the cache miss for P1 a true sharing miss? Under what conditions is that miss a false sharing miss?

Solution: *We know from the problem that x and b share a cache block—that condition must be true for the invalidation by P0 to cause a cache miss in P1 while reading b. If x and b share the exact same address—they both use the same word in the cache block—then the miss is a true sharing miss. If they have different addresses, the miss is a false sharing miss.*

b. (10 points) ***Solve either part (b) or part (c)—not both.***
Say we have a four-processor system that uses a write-invalidate, directory coherence protocol. The system contains a total of 8 memory blocks, as shown in the initial directory state below:

Block #	P0	P1	P2	P3	Dirty
0	0	0	0	0	0
1	1	0	1	1	0
2	1	0	0	0	1
3	0	1	0	1	0
4	0	0	1	1	0
5	0	1	0	0	1
6	0	1	0	0	0
7	0	0	0	0	0

For all sequences of transactions shown on the next page, list all messages sent as well as the final directory state for the block(s) in question. You should assume that each sequence of accesses is independent—your answer to part 2 does not depend on part 1—but accesses within a sequence are dependent on one another—your answer for part 1, access (ii) **does** depend on what happens in part 1, access (i).

Note: Your second handout contains an extra copy of the directory state above.

Solution: *See the table on the next page. Note that the "final directory state column" shows only the directory entry or entries for blocks that are referenced.*

QUESTION 7b SOLUTION

Transaction(s)	Messages sent	Final directory state *(shown as [P0,P1,P2,P3,Dirty])*
1. (i) P1: read block 2 (ii) P2: read block 2 (iii) P3: read block 2	*1. ReadMiss(P1,2) to directory* *Fetch(2) from directory to P0* *DataWriteBack(2,mem[2]) from P0 to directory* *DataValueReply(mem[2]) from directory to P1* *(ii) ReadMiss(P2,2) to directory* *DataValueReply(mem[2]) from directory to P2* *(iii) ReadMiss(P3,2) to directory* *DataValueReply(mem[2]) from directory to P3*	*Block 2:* *[1,1,1,1,0]*
2. (i) P2: write block 5 (ii) P2: write block 6 (iii) P2: write block 7	*(i) WriteMiss(P2,5) to directory* *FetchAndInvalidate(5) from directory to P1* *DataWriteBack(5,mem[5]) from P1 to directory* *DataValueReply(mem[5]) from directory to P2* *(ii) WriteMiss(P2,6) to directory* *Invalidate(6) from directory to P1* *DataValueReply(mem[6]) from directory to P2* *(iii) WriteMiss(P2,7) to directory* *DataValueReply(mem[7]) from directory to P2*	*Block 5:* *[0,0,1,0,1]* *Block 6:* *[0,0,1,0,1]* *Block 7:* *[0,0,1,0,1]*
3. (i) P3: write block 2 (ii) P0: write block 2 (iii) P3: read block 2	*(i) WriteMiss(P3,2) to directory* *FetchAndInvalidate(2) from directory to P0* *DataWriteBack(2,mem[2]) from P0 to directory* *DataValueReply(mem[2]) from directory to P3* *(ii) WriteMiss(P0,2) to directory* *FetchAndInvalidate(2) from directory to P3* *DataWriteBack(2,mem[2]) from P3 to directory* *DataValueReply(mem[2]) from directory to P0* *(i) ReadMiss(P3,2) to directory* *Fetch(2) from directory to P0* *DataWriteBack(2,mem[2]) from P0 to directory* *DataValueReply(mem[2]) from directory to P3*	*Block 2:* *[1,0,0,1,0]*

c. (10 points) ***Solve either part (b) or part (c)—not both.***

You are given a four-processor system that uses a write-invalidate, snooping coherence protocol. Each direct-mapped, write-back cache has four lines, each of which holds eight bytes; in the diagram below, only the least-significant byte of each word is shown. The cache states are I (invalid), S (shared), and M (modified/exclusive).

The caches and memory have the following initial state; please note that all addresses and tags are shown in hexadecimal:

P0

	State	Tag	Data	
B0	I	0x100	02	80
B1	S	0x108	00	88
B2	M	0x110	30	09
B3	I	0x118	00	10

P1

	State	Tag	Data	
B0	I	0x100	02	80
B1	M	0x128	AB	CD
B2	I	0x110	20	08
B3	S	0x118	14	92

P2

	State	Tag	Data	
B0	S	0x120	13	31
B1	S	0x108	00	88
B2	I	0x130	14	12
B3	I	0x138	01	38

P3

	State	Tag	Data	
B0	M	0x100	13	31
B1	S	0x108	00	88
B2	S	0x130	14	12
B3	S	0x118	14	92

Memory

Address	Data	
0x100	02	80
0x108	00	88
0x110	20	08
0x118	14	92
0x120	13	31
0x128	FF	FE
0x130	14	12
0x138	AB	BA

For each of the transactions listed on the next page, use the table to list all cache blocks modified and their final state, as well as all memory blocks modified and their final state. Assume each set of transactions starts with the same initial state—in other words, your answer to part (b) does not depend on your answer to part (a). However, you should track the state transitions of each block throughout the problem.

NOTE: Your second handout contains an extra copy of the tables above.

QUESTION 7c SOLUTION

Transaction(s)	Cache blocks modified	Memory blocks modified
1. (i) P0: write 0x114 ← 99 (ii) P2: read 0x110 (iii) P3: write 0x110 ← 99	*(i) P0.B2: (M, 0x110, 30 99)* *(ii) P0.B2: (S, 0x110, 30 99)* *P2.B2: (S, 0x110, 30 99)* *(iii) P0.B2: (I, 0x110, 30 99)* *P2.B2: (I, 0x110, 30 99)* *P3.B2: (M, 0x110, 99 99)*	*(ii) M[0x110]: 30 99*
2. (i) P2: write 0x138 ← 12 (ii) P2: read 0x118 (iii) P0: read 0x138	*(i) P2.B3: (M, 0x138, 12 BA)* *(ii) P2.B3: (S, 0x118, 14 92)* *(iii) P0.B3: (S, 0x138, 12 BA)*	*(ii) M[0x138]: 12 BA*
3. (i) P3: write 0x10C ← FF (ii) P3: write 0x134 ← 41 (iii) P3: write 0x104 ← AB	*(i) P0.B1: (I, 0x108, 00 88)* *P2.B1: (I, 0x108, 00 88)* *P3.B1: (M, 0x108, 00 FF)* *(ii) P3.B2: (M, 0x130, 14 41)* *(iii) P3.B0: (M, 0x100, 13 AB)*	