16.482 / 16.561: Computer Architecture and Design

Spring 2014

Final Exam Solution

1. (19 points) ***Multithreading***

a. (4 points) Assume you have three threads concurrently executing on your system. Under what circumstances would coarse-grained multithreading give the lowest total execution time for all three threads? Under what circumstances would fine-grained multithreading give the lowest total execution time?

Solution: Coarse-grained multithreading has two major benefits: it hides long-latency stalls and allows threads with few stalls to run to completion. If we have a situation where at least one thread has long latency stalls and another thread can hide those stalls by running steadily, coarse-grained multithreading is the best bet.

Fine-grained multithreading maximizes fairness among threads. In a situation where all threads have stalls that are roughly the same length, fine-grained multithreading would be the best pick.

b. (15 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
  - Note: The ALUs can handle MUL.D operations
- 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 all branches are 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, but you do not have to write the full instruction—writing the opcode (i.e. L.D, ADD.D) is sufficient.**

Thread 1:

```
L.D F0, 0(R1)
L.D F2, 8(R1)
ADD.D F4, F0, F2
SUB.D F6, F4, F2
S.D F6, 16(R1)
DSUBUI R1, R1, #16
BNEZ R1, loop
```

Thread 2:

```
L.D F2, 0(R1)
MUL.D F2, F0, F2
DADDI R1, R1, #16
ADD.D F10, F8, F2
S.D F10, -8(R1)
DSUB R20, R4, R1
BNZ R20, Loop
```

Thread 3:

```
ADD.D F4, F0, F2
SUB.D F6, F2, F6
ADD.D F8, F0, F4
MUL.D F10, F4, F6
DADDUI R1, R1, #16
S.D F10, -16(R1)
BNE R1, R2, loop
```

QUESTION 1b SOLUTION

Cycle	ALU1	ALU2	Mem1	Branch	Stalls?
1	T3: ADD.D F4,F0,F2	T3: SUB.D F6,F2,F6	T1: L.D F0,0(R1)		
2			T1: L.D F2,8(R1		
3	T3: ADD.D F8,F0,F4	T3: MUL.D F10,F4,F6	T2: L.D F2, 0(R1)		
4	T3: DADDUI R1,R1,#16				
5	T1: ADD.D F4,F0,F2				
6	T2: MUL.D F2,F0,F2	T2: DADDI R1,R1,#16			
7	T1: SUB.D F6,F4,F2		T3: S.D F10,-16(R1)	T3: BNE R1,R2,loop	
8					Stall
9	T1: DSUBUI R1,R1,#16		T1: S.D F6,16(R1)		
10		T2: ADD.D F10,F8,F2		T1: BNEZ R1, loop	
11					
12	T2: DSUB R20,R4,R1		T2: S.D F10,-8(R1)		
13				T2: BNZ R20,Loop	
14					
15					
16					
17					
18					
19					
20					

2. (26 points) ***Caches***

You are given a system which has an 8-byte, write-back cache with 4-byte blocks. The cache is direct-mapped. All memory accesses use 7-bit addresses.

a. (18 points) Assume the initial memory state shown below for the first 16 bytes:

Address		Address	
0	20	8	15
1	8	9	67
2	27	10	78
3	3	11	19
4	12	12	26
5	44	13	99
6	34	14	9
7	5	15	4

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. Assume the cache is initially empty.

Solution: Changed state shown in bold; invalid lines left empty for clarity

Access	Modified register	Cache state							Modified mem. block
		V	D	Tag	Data				
`lb $t0,2($zero)`	*$t0 = 27*	***1***	***0***	***00***	***20***	***8***	***27***	***3***	*None*
		0							
`sb $t0,12($zero)`	*None*	*1*	*0*	*00*	*20*	*8*	*27*	*3*	*None*
		1	***1***	***01***	***27***	***99***	***9***	***4***	
`lb $t1,5($zero)`	*$t1 = 44*	*1*	*0*	*00*	*20*	*8*	*27*	*3*	*Addr: 12-15 data: 27 99 9 4*
		1	***0***	***00***	***12***	***44***	***34***	***5***	
`sb $t1,4($zero)`	*None*	*1*	*0*	*00*	*20*	*8*	*27*	*3*	*None*
		1	*1*	*00*	***44***	*44*	*34*	*5*	
`lb $t0,3($zero)`	*$t0 = 3*	*1*	*0*	*00*	*20*	*8*	*27*	*3*	*None*
		1	*1*	*00*	*44*	*44*	*34*	*5*	

2 (continued)

b. (8 points) Assume the access sequence above is part of a loop, which executes five times. The remainder of the program contains no data memory accesses. Calculate the average memory access time for this program and system if the cache takes 3 cycles to access and the memory takes 250 cycles to access. Assume those are the only two levels in the memory hierarchy.

Solution: On its initial pass, shown above, the access sequence produces three cache misses (the first three accesses) and two cache hits (the last two accesses). However, to solve this problem correctly, you have to note that the first load is actually a hit on subsequent loop iterations, because that block (addresses 0-3) will remain in the cache for the duration of the loop. Therefore, each of the remaining four loop iterations produces two misses and three hits, for a total of 3 + (2×4) = 11 misses and 2 + (3×4) = 14 hits in the 25 total accesses.

You can now find the average memory access time in two different ways:

1. Calculate the total time required for all accesses, then divide by 25: Each hit takes 3 cycles; each miss takes 3 + 250 = 253 cycles, so the total time required is:

   14×3 + 11×253 = 2825 cycles

   and the AMAT is 2825 / 25 = **113 cycles**

2. Plug in the appropriate values to the AMAT equation: Since we have just a two-level hierarchy, using this equation is simple. The hit time is 3 cycles, the miss rate is (11/25) = 0.44, and the miss penalty is 250 cycles—the time to access memory. Therefore:

   AMAT = (hit time) + (miss rate) × (miss penalty)
   = 3 + (0.44) × (250) = 3 + 110 = **113 cycles**

3. (18 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	*1*	*1*	*1*	*12*
1	*0*	*0*	*0*	*--*
2	*1*	*0*	*1*	*5*
3	*1*	*0*	*0*	*8*
4	*0*	*0*	*0*	*--*
5	*1*	*1*	*0*	*2*

a. (3 points) Explain the purpose of the reference bits, including conditions under which those bits are set and cleared.

Solution: Reference bits indicate whether a given page has been accessed relatively recently; if the reference bit for a page is 0 when an eviction is required, that page is a candidate to be evicted from main memory.

b. (9 points) Assuming 16-bit addresses and 512 byte 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: 512 B = 2^9 byte pages have an 9-bit page offset and a (16-9) = 7-bit page number.

- 0x0A16

Solution: *0x0A16 =* ***0000 101****0 0001 0110$_2$ → page # = 0000101$_2$ = 5*
Frame # = 2 = 0000010 → physical address = ***0000 010****0 0001 0110$_2$ =* ***0x0416***

- 0x02FE

Solution: *0x02FE =* ***0000 001****0 1111 1110$_2$ → page # = 0000001$_2$ = 1 →* ***page fault***

- 0x011B

Solution: *0x011B =* ***0000 000****1 0001 1011$_2$ → page # = 0000000$_2$ = 0*
Frame # = 12 = 0001100 → physical address = ***0001 100****1 0001 1011$_2$ =* ***0x191B***

3 (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 16 frames, numbered 0-15, and frame 15 is the only frame that is free before this sequence executes. The initial state of the page table is repeated below for your reference.

ACCESS SEQUENCE

- Read page 1 → *Page fault; page 15 allocated to frame 1; Valid = Ref = 1; Dirty = 0*
- Write page 3 → *Dirty = Ref = 1; all other bits unchanged*
- Write page 5 → *Dirty = 1; all other bits unchanged*
- Read page 4 → *Page fault; page 2 evicted (only valid page with Ref = 0)*
  Frame 5 allocated to Page 4; Valid = Ref = 0

INITIAL PAGE TABLE STATE:

Virtual page #	Valid bit	Reference bit	Dirty bit	Frame #
0	1	1	1	12
1	0	0	0	--
2	1	0	1	5
3	1	0	0	8
4	0	0	0	--
5	1	1	0	2

FINAL PAGE TABLE STATE:

Virtual page #	Valid bit	Reference bit	Dirty bit	Frame #
0	1	1	1	12
1	***1***	***1***	***0***	***15***
2	***0***	***0***	***0***	***--***
3	***1***	***1***	***1***	***8***
4	***1***	***1***	***0***	***5***
5	***1***	***1***	***1***	***2***

4. (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. ***(Way prediction)*** *Consider the following two caches for use in a system:*

- *A direct mapped cache with an access time of 2 ns and an average hit rate of 90%*
- *A four-way set associative cache with an access time of 6 ns and an average hit rate of 95%.*

Assume the cache miss penalty for the system is 100 ns.

We want to use way prediction to improve the performance of the set associative cache. If a way prediction hit takes as long as a direct-mapped access, and a way-prediction miss adds an additional 6 ns, how accurate must the way predictor be to match the average access time of the direct-mapped cache?

Solution: To solve this problem, we have to calculate average memory access times. Recall that the formula for AMAT is:

AMAT = (hit time) + (miss rate)(miss penalty)

For the direct-mapped case, this value is simple to calculate—our hit time is 2 ns, the miss rate is (100%-90%) = 10% = 0.1, and the miss penalty is 100 ns:

AMAT = (2 ns) + (0.1)(100 ns) = 2 ns + 10 ns = 12 ns

For the way-predicted case, the miss rate is (100%-95%) = 5% = 0.05 and the miss penalty remains the same. The hit time has become more complex. Way predictor hits take 2 ns; way predictor misses take 2 + 6 = 8 ns. If we call HR_{way} the success rate of the way predictor (the unknown for which we need to solve), we can then write the hit time as:

$(HR_{way})(2\text{ ns}) + (1 - HR_{way})(8\text{ ns})$

The question asks you to find the way predictor success rate that will give the same access time—12 ns—as the direct-mapped case, so we must solve the following equation to get HR_{way}:

$((HR_{way})(2\text{ ns}) + (1 - HR_{way})(8\text{ ns})) + (0.05)(100\text{ ns}) = 12\text{ ns}$

$2HR_{way} + 8 - 8HR_{way} + 5 = 12$
$-6HR_{way} + 13 = 12$
$-6HR_{way} = -1$

➔ **HR_{way} = -1 / -6 = 1/6 = 16.666…%**

b. ***(Multi-banked/non-blocking caches)*** *Assume we have a system containing 32 blocks of memory, numbered 0-31. 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:*

4, 5, 1, 2, 24, 25, 26, 27, 16, 12

Note that all of these accesses are misses; each miss takes 10 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: (10 cycles per access) * (10 accesses) = **100 cycles.**

ii. (5 points) Calculate the total time for these 10 accesses if the cache is divided evenly into two 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, 16, 24	Bank 0
1	1, 9, 17, 25	
2	2, 10, 18, 26	
3	3, 11, 19, 27	
4	4, 12, 20, 28	Bank 2
5	5, 13, 21, 29	
6	6, 14, 22, 30	
7	7, 15, 23, 31	

Now, we can determine the total access time by looking at each access and determining which ones can be overlapped. Note that up to 2 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 **82 cycles** given this cache organization.

Block #	Bank	Start cycle	End cycle
4	1	1	10
5	1	11	20
1	0	12	21
2	0	22	31
24	0	32	41
25	0	42	51
26	0	52	61
27	0	62	71
16	0	72	81
12	1	73	82

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

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

Line #	Blocks mapped to this line	
0	0, 8, 16, 24	Bank 0
1	2, 10, 18, 26	
2	4, 12, 20, 28	
3	6, 14, 22, 30	
4	1, 9, 17, 25	Bank 2
5	3, 11, 19, 27	
6	5, 13, 21, 29	
7	7, 15, 23, 31	

As shown in the table below, the sequence will take **60 cycles** given this cache organization.

Block #	Bank	Start cycle	End cycle
4	0	1	10
5	1	2	11
1	1	12	21
2	0	11	20
24	0	21	30
25	1	22	31
26	0	31	40
27	1	32	41
16	0	41	50
12	0	51	60

5. (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 take 200 ms, small reads take 50 ms, and small writes take 100 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.*

1. *read S2*
2. *read S10*
3. *write S9*
4. *read S4*
5. *read S8*
6. *write S7*
7. *write S9*
8. *read S2*

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

Solution: Taking each of these one at a time:

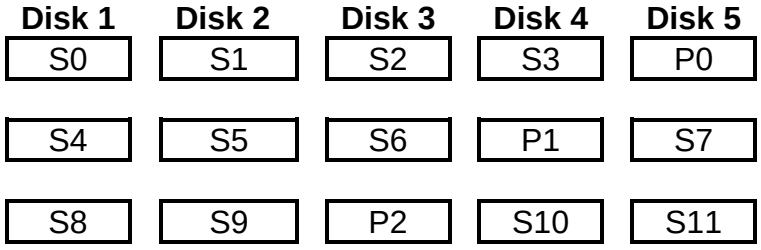
- In RAID 3, only large reads and writes are allowed. Since each operations involve every disk, no operations may be overlapped, and the total time for 8 large reads and writes is 8 x 200 ms = **600 ms**.
- In RAID 4, you may perform small reads, but only large writes. Therefore, any two consecutive reads that do not use the same disk can proceed in parallel. 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

In this particular problem, all pairs of consecutive reads (S2/S10, S4/S8) involve the same disk and cannot be overlapped. They will, however, take less time than in RAID 3 because these operations can be performed as small reads. The time required for this sequence is therefore:

	Time to read S2 (small read)	50 ms
	Time to read S10 (small read)	50 ms
	Time to write S9 (large write)	200 ms
	Time to read S4 (small read)	50 ms
	Time to read S8 (small read)	50 ms
	Time to write S7 (large write)	200 ms
	Time to write S9 (large write)	200 ms
+	Time to read S2 (small read)	50 ms
	TOTAL	**850 ms**

- In RAID 5, both small reads and writes are allowed. This means that we can overlap any two consecutive operations that do not share the same disk. Remember that RAID 5 also involves interleaved parity, which makes small writes more feasible, so the organization would change as follows:



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 S2	1	50	Overlap reads on different disks
read S10	1	50	
write S9	51	150	S8 on same disk as S4, so no overlap for those reads. Both reads can overlap write to S9, which uses disks 2 and 3.
read S4	51	100	
read S8	101	150	
write S7	151	250	These two can overlap because both transactions involve different data and parity disks—the first write uses Disks 4 and 5, while the second uses Disks 2 and 3
write S9	151	250	
read S2	251	300	

In RAID 5, this sequence takes **300 ms.**

6. **(14 points)** *Coherence protocols*

a. ***Solve either part (a) or part (b)—NOT BOTH***

(Snooping coherence protocols) 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	01	23
B1	S	0x108	00	88
B2	M	0x110	00	30
B3	I	0x118	00	10

P1

	State	Tag	Data	
B0	I	0x100	01	23
B1	M	0x128	AB	CD
B2	S	0x130	14	12
B3	S	0x118	14	92

P2

	State	Tag	Data	
B0	S	0x120	13	31
B1	S	0x108	00	88
B2	I	0x130	51	55
B3	I	0x138	01	38

P3

	State	Tag	Data	
B0	S	0x120	13	31
B1	S	0x108	00	88
B2	I	0x110	00	30
B3	S	0x118	14	92

Memory

Address	Data	
0x100	00	00
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 (2) does not depend on your answer to part (1). 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 6a SOLUTION

Transaction(s)	Cache blocks modified	Memory blocks modified
1. (i) P1: read 0x110 (ii) P2: read 0x110 (iii) P3: read 0x110 (iv) P0: write 0x110 ←55	***(i) P0.B2: (S,0x110, 00 30)*** ***P1.B2: (S, 0x110, 00 30)*** ***(ii) P2.B2: (S, 0x110, 00 30)*** ***(iii) P3.B2: (S, 0x110, 00 30)*** ***(iv) P0.B2: (M, 0x110, 55 30)*** ***P1.B2: (I, 0x110, 00 30)*** ***P2.B2: (I, 0x110, 00 30)*** ***P3.B2: (I, 0x110, 00 30)***	***(i) M[0x110]: (00 30)***
2. (i) P3: write 0x110←45 (ii) P0: write 0x110←67 (iii) P3: read 0x114 (iv) P0: read 0x114	***(i) P0.B2: (I, 0x110, 00 30)*** ***P3.B2: (M, 0x110, 45 30)*** ***(ii) P0.B2: (M, 0x110, 67 30)*** ***P3.B2: (I, 0x110, 45 30)*** ***(iii) P0.B2: (S, 0x110, 67 30)*** ***P3.B2: (S, 0x110, 67 30)*** ***(iv) No change***	***(i) M[0x110]: (00 30)*** ***(ii) M[0x110]: (45 30)*** ***(iii) M[0x110]: (67 30)***

6 (continued)

b. ***Solve either part (a) or part (b)—NOT BOTH***

(Directory protocols) 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	1	0	1	0	0
1	0	1	0	0	1
2	0	0	0	0	0
3	0	1	1	1	0
4	0	0	0	1	1
5	1	1	0	0	0
6	0	1	0	0	1
7	1	1	1	1	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.

QUESTION 6b SOLUTION

Transaction(s)	Messages sent	Final directory state *(shown as [P0,P1,P2,P3,Dirty])*
1. P0: read block 1 P2: read block 1 P3: write block 1 P3: read block 1	***(i) ReadMiss(P0,1) to directory*** ***Fetch(1) from directory to P1*** ***DataWriteBack(1,mem[1]) from P1 to directory*** ***DataValueReply(mem[1]) from directory to P0*** ***(ii) ReadMiss(P2,1) to directory*** ***DataValueReply(mem[1]) from directory to P2*** ***(iii) WriteMiss(P3,1) to directory*** ***Invalidate(1) from directory to P0, P1, and P2*** ***DataValueReply(mem[1]) from directory to P3*** ***(iv) No messages—access is a hit***	***Block 1:*** ***[0, 0, 0, 1, 1]***
2. P0: write block 2 P1: read block 2 P2: read block 2 P3: write block 2	***(i) WriteMiss(P0,2) to directory*** ***DataValueReply(mem[2]) from directory to P0*** ***(ii) 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*** ***(iii) ReadMiss(P2,2) to directory*** ***DataValueReply(mem[2]) from directory to P2*** ***(iv) WriteMiss(P3,2) to directory*** ***Invalidate(2) from directory to P0, P1, and P2*** ***DataValueReply(mem[2]) from directory to P3***	***Block 2:*** ***[0, 0, 0, 1, 1]***