16.482 / 16.561: Computer Architecture and Design

Fall 2014

Final Exam Solution

1. (18 points) ***Multiple issue and multithreading***

This problem deals with the 3 threads below (which are also shown on your extra handout):

Thread 1:

```
ADD.D F0, F2, F4
L.D   F6, 0(R1)
SUB.D F8, F0, F8
ADD.D F10, F0, F10
MUL.D F4, F8, F6
DADDUI R1, R1, #16
BNE   R1, R2, loop
```

Thread 2:

```
MUL.D F2, F6, F6
S.D   F2, 8(R1)
SUB.D F6, F6, F0
DADDUI R1, R1, #-1
BNEZ  R1, labelA
MUL.D F0, F4, F2
ADD.D F0, F6, F0
J     labelB
```

Thread 3:

```
L.D   F0, 0(R1)
ADD.D F4, F6, F8
ADD.D F10, F0, F4
L.D   F2, 8(R1)
S.D   F10, 16(R1)
L.D   F4, 24(R1)
SUB.D F6, F4, F2
JR    R31
```

In both parts of the problem, the processor executes instructions in order, and you should assume all branches are not taken. Each instruction has the latency given below:

- *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*

a. *(6 points) Will these threads run faster on a processor that uses fine-grained or coarse-grained multithreading? Assume the coarse-grained processor would switch threads on stalls greater than 3 cycles (i.e., 4 or more cycles). Explain your answer without calculating the total time required for each case. (Hint: look at the number and length of stalls in each thread.)*

Solution: We can see from the instruction latencies that no instruction should have to stall for more than 3 cycles, since the longest-latency instruction is 4 cycles. We can go through the threads and look at the dependences in detail, but all three of these threads will have some "short" stalls that can be hidden by regularly switching threads. If run with coarse-grained multithreading, the processor would have to wait for every stall cycle. Therefore, these threads would run faster on a processor using fine-grained multithreading.

1 (continued)

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

- *Five functional units: 2 ALUs, 2 memory ports (load/store), 1 branch*
  - *Note: The ALUs can handle MUL.D and DADDUI operations*
- *Thread 1 is the preferred thread, followed by Thread 2 and Thread 3.*

Your solution should use the table below, which contains columns to show each cycle and the functional units being used during that cycle. ***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.***

Remember, your extra handout contains a copy of the threads and latencies.

Cycle	ALU1	ALU2	Mem1	Mem2	Branch
1	T1: ADD.D	T2: MUL.D	T1: L.D	T3: L.D	
2	T3: ADD.D				
3	T1: SUB.D	T1: ADD.D			
4	T3: ADD.D		T3: L.D		
5	T1: MUL.D	T1: DADDUI	T2: S.D		
6	T2: SUB.D	T2: DADDUI	T3: S.D	T3: L.D	T1: BNE
7	T2: MUL.D				T2: BNEZ
8					
9	T3: SUB.D				T3: JR
10					
11	T2: ADD.D				T2: J

2. *(23 points)* ***Cache basics*** *(4 points) Identify and explain the two types of locality, giving an example of each.*

Solution:
Spatial locality: if you access a memory location, you are more likely to access a location near it than some random location. Example: array accesses typically proceed from one element to the next, not randomly throughout the array.

Temporal locality: if you access a memory location, you are more likely to access that location again than some random location. Example: index variable for a loop will be accessed in every loop iteration.

a. (3 points) Explain how LRU replacement works and why it is effective.

Solution: LRU (least recently used) replacement is a policy used in set-associative caches, in which the block that is evicted from a full set on a cache miss is the one that was accessed least recently. This policy is based on temporal locality—since the block that was most recently accessed is most likely to be accessed again, the one that was least recently accessed is least likely to be accessed again, making it the best candidate for eviction.

2 (continued)

b. (16 points) 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.

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 $t0,8($zero)`	*$t0 = 19*								
		1	***0***	***0000***	***19***	***78***	***9***	***12***	
`sb $t0,9($zero)`									
		1	***1***	*0000*	*19*	***19***	*9*	*12*	
`lb $t1,15($zero)`	*$t1 = 98*								
		1	*1*	*0000*	*19*	*19*	*9*	*12*	
		1	***0***	***0000***	***1***	***0***	***63***	***98***	
`sb $t1,24($zero)`									*mem[8-11] = [19 19 9 12]*
		1	***1***	***0001***	***98***	***24***	***21***	***7***	
		1	*0*	*0000*	*1*	*0*	*63*	*98*	

3. *(11 points)* ***Virtual memory***

a. *(3 points) Explain the purpose of a translation lookaside buffer (TLB).*

Solution: The TLB functions as a cache for page table entries. On a cache miss, the TLB is checked to see if a valid translation for the given virtual page has been recently performed. If so, the time required to perform address translation is much faster than it would be if a full page table access was required.

b. *(8 points) Fill in the table at the bottom of the page to show the* ***final*** *state of the given page table after the following sequence of accesses. Assume main memory has 8 frames, numbered 0-7, and frame 5 is the only frame that is free before this sequence executes.*

ACCESS SEQUENCE

- Read page 3 → *Page fault; frame 5 allocated to page 3; Valid = Ref = 1; Dirty = 0*
- Write page 5 → *Dirty = 1; all other bits unchanged*
- Write page 1 → *Page fault; page 2 evicted (only valid page with Ref = 0); Frame 4 allocated to page 1; Valid = Ref = Dirty = 1*
- Read page 0 → *No bits change*

INITIAL PAGE TABLE STATE:

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

FINAL PAGE TABLE STATE:

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

4. *(19 points)* ***Cache optimizations***

a. *(9 points) Briefly explain three of the following four cache optimizations:*

i. *Trace caches*

Solution: A cache in which dynamic instruction traces are stored. Dynamic instruction traces are sequences of instructions in the order they are actually executed, not necessarily how they are stored in memory—for example, jump instructions and their targets are stored consecutively in dynamic traces.

ii. *Way prediction*

Solution: An optimization for set-associative caches in which a predictor is used to identify the line within a set that is most likely to have the desired data. If the prediction is accurate, the access time is comparable to that of a direct mapped cache, while the cache retains the lower miss rate of a set-associative cache.

iii. *Early restart/critical word first*

Solution: Optimizations to reduce the time required to fill a cache line on a miss. Early restart allows the cache to supply data to the processor as soon as the desired word is loaded, rather than waiting for the entire block to fill. Critical word first ensures that the desired word is loaded first, rather than always starting with the lowest addressed word in the block.

iv. *Hardware prefetching*

Solution: An optimization intended to reduce miss rate by anticipating future misses. Hardware prefetchers fetch two cache blocks on a miss—the block that was accessed and a block that is determined to be most likely accessed next. The simplest and often most effective form of hardware prefetching is next sequential prefetching, in which the next consecutive block is fetched on a miss.

4 (continued)

b. *(10 points) Assume we have a system containing 32 blocks of memory, numbered 0-31. The processor's L1 data cache in this system contains four banks in which the blocks are sequentially interleaved. This system runs a program in which it must access the following ten blocks of data, but may do so in any order:*

1, 2, 10, 12, 15, 16, 24, 27, 28, 30

Assume all ten accesses are misses, each of which takes 20 cycles to handle, and also assume that only one new access can be started each cycle.

Determine the minimum number of cycles required to access all ten blocks, as well as a sequence of accesses that will take that amount of time. (Note: while there is only one correct value for the minimum number of cycles, there are multiple ways to access all ten blocks in that amount of time.) ***For full credit, show all work.***

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 using sequential interleaving, the blocks are mapped to banks as shown in the table below—note that the number of cache lines do not matter. Blocks that are accessed in the given sequence are marked in red, and the number of accesses per bank is also given:

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

To minimize the total access time for this sequence, consider the following:

- Bank 0 has the most accesses, so accesses to that bank should start as early as possible. Those four accesses alone will take a total of 80 cycles.
- You want to maximize the number of simultaneous accesses at all times to ensure that no other accesses are ongoing when bank 0's accesses finish.

One possible sequence that completes in **80 cycles** is below. Any sequence starting with an access to bank 0 that does not delay accesses to other banks will take the same amount of time.

Block #	Bank	Start cycle	End cycle
12	0	1	20
2	2	2	21
15	3	3	22
1	1	4	23
16	0	21	40
10	2	22	41
27	3	23	42
24	0	41	60
30	2	42	61
28	0	61	80

5. (14 points) ***RAID***

Both parts of this problem involve a six-disk RAID array containing a total of 18 sectors; the exact sector configuration depends on the RAID level used. In all cases, fifteen of the eighteen sectors (S0-S14) hold data, while the remaining three sectors (P0-P2) hold parity information.

a. (6 points) Determine how many of the six disks would be used in each of the following operations in this array, and briefly explain why:

i. Large read

Solution: A large read is a read that uses all disks in the array—the disks that do not hold the desired data are read to check that the parity disk is correct. So, all six disks would be used in this array.

ii. Large write

Solution: Similar to a large read, a large write would use all six disks in this array. The data disks that are not being written are used to compute the new value for the appropriate sector on the parity disk.

iii. Small read

Solution: A small read requires only one disk, regardless of the array size. Small reads use the per-disk error correction codes to check for errors on a read, rather than reading all other disks.

iv. Small write

Solution: A small write requires only two disks, regardless of the array size. In a small write, one data disk and the parity disk are written. Small writes use the fact that the difference between the new and old parity sector will be the same as the difference between the new and old data sector, so no additional information is required from other disks.

5 (continued)

b. (8 points) Given the six-disk array described on the previous page (sectors S0-S14 hold data; P0-P2 hold parity information), also assume the following for this part of the problem:

- *The array is configured with RAID 5.*
- *Large reads/writes take 200 ms, small reads take 50 ms, and small writes take 100 ms.*
- *Requests are queued in such a manner that up to three consecutive operations may proceed simultaneously if they do not share any disk within the array. Multiple accesses in the same stripe are allowed.*
- *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.*

Determine the time required for this array to complete the following sequence of accesses. For full credit, show all work, including the organization of the array:

1. *read S6*
2. *write S12*
3. *read S8*
4. *write S2*
5. *write S5*
6. *write S9*
7. *write S11*
8. *read S14*

Solution: In RAID 5, both small reads and writes are allowed; we can therefore overlap any three 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	Disk 6
S0	S1	S2	S3	S4	P0
S5	S6	S7	S8	P1	S9
S10	S11	S12	P2	S13	S14

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. Remember that the small writes enabled in RAID 5 require two disks—the data and parity disks. The solution is best described by tracking start and end times for each operation:

Operation	Start time	End time	Notes
read S6	1	50	Different disks—overlap allowed.
write S12	1	100	
read S8	101	150	Read to S8 cannot overlap with write to S12, as S8 is on same disk as P1. However, all three of these operations use different disks and can overlap.
write S2	101	200	
write S5	101	200	
write S9	201	300	These operations use disks 5/6 (write S9) and 2/4 (write S11) and can therefore overlap.
write S11	201	300	
read S14	301	350	Read to S14 cannot start until write to S9 is done

This sequence takes **350 ms = 0.35 s.**

6. (15 points) ***Coherence protocols***

a. (3 points) In a snooping coherence protocol, what is a relevant transaction?

Solution: Relevant transactions are identified by the cache controller for a processor in a multiprocessor system, which monitors the interconnect between processors for read or write misses from other processors. If one of those misses accesses data that is stored in that processor's cache, then it is a relevant transaction.

b. (4 points) Explain the differences between the Fetch and Fetch/Invalidate messages used in a directory coherence protocol and describe a situation in which each message would be used.

Solution: Both of these messages are used when a block is in the exclusive state and a processor that does not hold the lone copy of the block requests access to that data.

A "Fetch" message is used to initiate a write back from a remote node when another node requests a read only copy of the block. The data will be written back to memory, and two shared copies of the block will exist once the transaction is complete.

A "Fetch/Invalidate" message initiates a write back from a remote node when another node requests an exclusive, writeable copy of the block. The data will be written back to memory, and the remote node will invalidate its copy, as the node requesting the data intends to change it. The block therefore remains in the exclusive state—it is just stored in a different node.

c. (3 points) Explain the difference between true and false sharing misses.

Solution: A true sharing miss occurs when multiple processors share the same piece of data. When one of those processors writes the datum, the other processors' copies will be invalidated and their next accesses to that datum will be cache misses. False sharing misses are similar, but these occur when the processors share separate pieces of data that happen to reside in the same cache block.

For example, in a block with 4 words, numbered 0 to 3, assume two processors use the block. If P0 and P1 both access word 0 and P0 writes that word, P1's next access will be a miss—a true sharing miss, because both P0 and P1 use word 0 from the block.

If, however, P0 uses word 0 and P1 uses word 1, a write from P0 to word 0 will cause P1 to miss on its next access to word 1. That miss is a false sharing miss—the processors share the same block, but share different data within the block.

6 (continued)

d. (5 points) Say we have a dual-processor system with two nodes, P0 and P1, which share a block at address A. The system uses a write-invalidate, directory coherence protocol. Initially, the directory entry for the block reads:

	P0	P1	Dirty
A	1	0	1

If P1 now attempts to read block A, what messages are sent between the nodes and directory to ensure P1 gets the most up-to-date block copy and the directory holds the appropriate state? You may want to draw a diagram to support your answer.

Solution: First, note that the state shown above means that the block is in the modified state—P0 has the only copy, and that copy has yet to be written back to memory. This information means that any access from P1 involving block A must access P0 to get the most up-to-date copy of the block.

The messages involved in this transaction are therefore:

- Write Miss(P1, A): P1 indicates to the directory that it wishes to write this block.
- Fetch/Invalidate(A): The directory sends a fetch & invalidate request to P0. This message indicates that P0 should provide the most up-to-date copy of the block and then invalidate its own copy, since P1 will now be modifying the block.
- Data Write Back(A, <data>): P0 responds to the previous message with the most up-to-date values for block A, so that those values may be stored in memory and sent to the cache requesting the data.
- Data Value Reply(<data>): The directory can now send the up-to-date values for the block to P1; once this reply is received, P1 will proceed with its write.