16.482 / 16.561: Computer Architecture and Design

Spring 2015

Final Exam Solution

1. *(17 points)* ***Multiple issue and multithreading***

a. *(3 points) Explain why simply increasing the number of instructions a processor fetches and issues will not significantly improve its performance.*

Solution: The performance of a single-threaded processor is limited by the instruction-level parallelism within that thread. Fetching and issuing more instructions isn't likely to uncover significantly more independent instructions—it's actually likely to consume independent instructions more quickly, potentially leading to more stall cycles.

b. *(4 points) You are given a single program containing four separate threads of execution. Two of these threads experience a cache miss every 20 instructions, while the other two threads contain shorter but more frequent stalls. Would this program run faster on a processor using fine-grained or coarse-grained multithreading? Explain your answer for full credit.*

Solution: Given that two of the threads experience long latency stalls due to cache misses, the program is more likely to run faster using coarse-grained multithreading, which would allow the threads with shorter stalls to run without interruption whenever the threads with longer stalls are waiting.

c. *(10 points) This problem deals with the 3 threads below:*

Thread 1:
```
L.D    F0, 0(R1)
L.D    F2, 8(R1)
ADD.D  F4, F0, F8
SUB.D  F6, F2, F8
DADDUI R1, R1, 16
BNE    R1, 1600, L1
```

Thread 2:
```
SUB.D  F8, F0, F2
ADD.D  F6, F4, F8
S.D    F6, 8(R1)
DADDUI R1, R1, 16
L.D    F2, 0(R1)
SUB.D  F4, F6, F2
```

Thread 3:
```
DADDUI R3, R1, R2
SUB.D  F6, F8, F10
L.D    F4, 0(R3)
S.D    F6, 0(R1)
DADDUI R1, R1, 24
DADDUI R2, R2, -8
BNE    R1, R2, loop
```

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: 4 cycles (1 EX, 3 MEM)*
- *ADD.D/SUB.D: 3 cycles*
- *DADDUI: 2 cycles*
- *All other operations: 1 cycle*

1 (continued)

c (continued) Determine how long these threads 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 DADDUI operations*
- *Thread 1 is the preferred thread, followed by Thread 2 and Thread 3.*

Your solution should use the table on the next page, 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.***

Cycle	ALU1	ALU2	Mem1	Mem2	Branch
1	T2: SUB.D	T3: DADDUI	T1: L.D	T1: L.D	
2	T3: SUB.D				
3			T3: L.D		
4	T2: ADD.D				
5	T1: ADD.D	T1: ADD.D	T3: S.D		
6	T1: DADDUI	T3: DADDUI			
7	T2: DADDUI	T3: DADDUI	T2: S.D		
8					T1: BNE
9			T2: L.D		T2: BNE
10					
11					
12					
13	T2: SUB.D				

2. *(23 points)* ***Cache basics***

a. *(4 points) Explain the three different cache block placement strategies (in other words, how to determine where a block is located in the cache), listing one benefit of each.*

Solution:

- Direct mapped: each block can only be stored in one location. Direct mapped caches offer the fastest access times, since only one location needs to be searched for the data.
- Set associative: each block can be stored in one of a small group of locations, called a set. Set associative cache
- Fully associative: each block can be placed anywhere in the cache. Fully associative caches typically have the fewest conflicts and therefore the lowest miss rates.

b. *(3 points) Explain how and why LRU replacement is approximated—not implemented exactly—in set-associative caches with associativity greater than 2.*

Solution: LRU replacement is approximated in caches with associativity greater than 2 by keeping a single most recently used (MRU) bit for the entire set and replacing any block other than the MRU block when necessary. The approximation is used because it is much simpler than implementing LRU replacement exactly for higher associativity, and its performance is not significantly worse than true LRU replacement in these cases.

2 (continued)

c. *(16 points) You are given a system which has a 16-byte, write-back cache with 4-byte blocks. The cache is 2-way set-associative. The system uses 8-bit addresses, and the cache is initially empty. At first, \$t0 = 9 and \$t1 = 16.*

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	*8*	*9*	*49*	*17*	*5*	*25*	*24*
2	*35*	*10*	*9*	*18*	*14*	*26*	*27*
3	*10*	*11*	*21*	*19*	*13*	*27*	*7*
4	*99*	*12*	*3*	*20*	*49*	*28*	*18*
5	*17*	*13*	*0*	*21*	*77*	*29*	*8*
6	*64*	*14*	*90*	*22*	*15*	*30*	*55*
7	*1*	*15*	*4*	*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	MRU	Tag	Data				
`sb $t0,14($zero)`										
		1	**1**	**1**	**00001**	**3**	**0**	**9**	**4**	
				0						
`lb $t1,30($zero)`	\$t1 = 55									
		1	1	**0**	00001	3	0	9	4	
		1	**0**	**1**	**00011**	**18**	**8**	**55**	**99**	
`lb $t0,7($zero)`	\$t0 = 1									mem[12-15]= [3 0 9 4]
		1	**0**	**1**	**00000**	**99**	**17**	**64**	**1**	
		1	0	**0**	00011	18	8	55	99	
`sb $t0,28($zero)`										
		1	0	**0**	00000	99	17	64	1	
		1	**0**	**1**	00011	**1**	8	55	99	

3. *(12 points)* ***Virtual memory***

a. *(3 points) Explain why a translation lookaside buffer (TLB) is implemented as a fully associative structure.*

Solution: TLB misses require a memory access to perform address translation, thus slowing down the actual data access, so preventing TLB misses is more important than having extremely fast TLB accesses. Using a fully associative structure will minimize the miss rate of the TLB.

b. *(9 points) This problem involves a process using the page table below:*

PAGE TABLE STATE:

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

Assume the system uses 16-bit addresses and 2 KB pages. The process accesses four addresses: 0x07FE, 0x1978, 0x26AA, and 0x2C33.

Determine (i) which address causes a page fault, (ii) which one sets a previously cleared reference bit for a page that is currently in physical memory, and (iii) which one accesses a page that has been modified since it was brought into memory. ***For full credit, show all work.***

Solution: First of all, note that 2 KB = 2^{11} B pages require an 11-bit page offset, and therefore allow for a 5-bit virtual page number within the 16 bit address.

The next step is to identify the virtual page numbers within each of the given addresses:

- 0x07FE = $\underline{0000\ 0}111\ 1111\ 1110_2$ → page 0
- 0x1978 = $\underline{0001\ 1}001\ 0111\ 1000_2$ → page 3
- 0x26AA = $\underline{0010\ 0}110\ 1010\ 1010_2$ → page 4
- 0x2C33 = $\underline{0010\ 1}100\ 0011\ 0011_2$ → page 5

By referring back to the page table, we can see that the answers are as follows:

(i) Address causing a page fault: access to an invalid page → page 3 → 0x1978
(ii) Address setting a previously cleared reference bit: page must be valid but reference bit must be 0 → page 5 → 0x2C33
(iii) Address accessing a modified page: valid page with dirty bit = 1 → page 4 → 0x26AA

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

a. *(9 points) Answer three of the following four questions about cache optimizations:*

i. *Explain why it is possible to have multiple copies of the same instruction stored in a trace cache*

Solution: Instructions can potentially be part of multiple traces that depend on different branch outcomes—one copy of the instruction might be in a trace in which a branch is taken, while another copy might be in a different trace in which the same branch is not taken.

ii. *In a multi-banked cache, why should data be sequentially interleaved across the banks?*

Solution: Since most programs display spatial locality, they are likely to access nearby blocks within a short time period. Sequential interleaving splits blocks with consecutive addresses across different banks to minimize the likelihood that multiple accesses will be made to the same bank in a short period of time, thus preventing those accesses from proceeding in parallel.

iii. *Why does critical word first offer no performance benefit without being combined with early restart?*

Solution: Critical word first allows a cache block to be filled starting with the requested data, rather than starting with the lowest addressable byte or word within the cache block. However, without early restart, which allows the requested data to be supplied to the processor as soon as it is present in the cache, the entire block must fill before the processor gets the data. Therefore, the order in which data is brought into the cache does not matter without using early restart to get the requested data to the processor as quickly as possible.

iv. *Under what circumstances would next sequential prefetching actually increase the miss rate of a cache?*

Solution: If the prefetched data displaces a block that the processor accesses, then prefetching would cause a miss that would not have occurred otherwise. If prefetching causes more misses in this manner than it prevents by bringing data in early, then the cache would have a higher miss rate with prefetching than without.

4 (continued)

b. *(10 points) A designer is considering two cache designs for a memory subsystem:*

- *A direct-mapped cache with a 90% hit rate that takes 2 ns to access.*
- *A set-associative cache with a 93% hit rate that takes 10 ns to access.*

To improve the performance of the set-associative cache, the designer considers adding a way predictor to that cache. If the way predictor is correct, cache hits would take only 3 ns, while way misses that still produce cache hits would take 12 ns. Assume that the cache miss penalty is 200 ns, regardless of the type of cache being used.

How accurate must the way predictor be for the system to have the same overall performance with the set-associative cache as it would with the direct-mapped cache? ***For full credit, show all work.***

Solution: To solve this problem, you'll have to work with the average memory access time (AMAT) of the memory subsystem. Recall that, in general:

$$AMAT = (hit\ time) + (miss\ rate \times miss\ penalty)$$

The given information allows us to write specific formulas for the AMAT of each setup as shown below (AMAT_{DM} = AMAT with direct mapping; AMAT_{SA} = AMAT with set associative; HR_{WP} = hit rate of way predictor). In each case, we show the general formula before simplifying as much as possible:

$$AMAT_{DM} = (2\ ns) + (0.1)\times(200\ ns) = 22\ ns$$

$$\begin{aligned} AMAT_{SA} &= ((3\ ns \times HR_{WP}) + \big(12\ ns \times (1 - HR_{WP})\big)) + (0.07)\times(200\ ns) \\ &= ((3ns \times HR_{WP}) + 12\ ns - \ (12ns \times HR_{WP}) + 14\ ns) \\ &= (-9ns \times HR_{WP}) + \ 26\ ns \end{aligned}$$

To determine the necessary way predictor hit rate for these two subsystems to have the same performance, we set these equations equal to one another:

$$\begin{aligned} 22ns &= (-9ns \times HR_{WP}) + \ 26\ ns \\ -4\ ns &= (-9\ ns \times HR_{WP}) \\ HR_{WP} &= \left(\frac{-4ns}{-9ns}\right) = \frac{4}{9} \approx 0.44 \end{aligned}$$

Therefore, for the two subsystems to have the same performance, the way predictor would need a hit rate of approximately **44%.**

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

a. *(6 points) Identify which types of operations—reads and/or writes—can be overlapped in each of the following RAID levels, and briefly explain why:*

i. *RAID 3*

Solution: RAID 3 uses both large reads and large writes—operations which use an entire stripe—thus preventing any accesses from being overlapped.

ii. *RAID 4*

Solution: RAID 4 uses small reads (reads to a single disk) and large writes, so read operations can be overlapped if they access different disks, but nothing can overlap with a write.

iii. *RAID 5*

Solution: RAID 5 uses small reads and small writes (writes involving one data disk + one parity disk), so any operations can be overlapped as long as they do not share the same disk.

5 (continued)

b. *(8 points) This part of the problem involves a five-disk RAID 5 array containing a total of 20 sectors. Sixteen of the twenty sectors (S0-S15) hold data, while the remaining four sectors (P0-P3) hold parity information. You should also assume the following:*

- *Large reads/writes take 250 ms, small reads take 75 ms, and small writes take 125 ms.*
- *The number of disks in the array is the only limit on performance—any number of consecutive operations may proceed simultaneously if they do not share any disk within the array. Multiple accesses in the same stripe are allowed.*
- *All operations must be performed in order.*
- *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 S5*
2. *write S10*
3. *read S13*
4. *write S4*
5. *write S15*
6. *read S3*
7. *write S7*
8. *write S14*

Solution: In RAID 5, both small reads and writes are allowed; we can therefore overlap any consecutive operations that do not share the same disk. Remember RAID 5 also involves interleaved parity to make small writes possible. This solution assumes the organization below:

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
S12	P3	S13	S14	S15

We can overlap any transactions that do not share a disk. Remember that the small writes enabled in RAID 5 require two disks—the data and parity disks.

Operation	Start	End	Notes
read S5	1	75	Use disks 2, 3, 4
write S10	1	125	
read S13	126	200	Read to S13 uses disk 3—can't overlap with write S10. However, these three operations are split across all 5 disks and can therefore overlap
write S4	126	250	
write S15	126	250	
read S3	251	325	Write to S7 also uses disk 4—can't overlap
write S7	326	450	Write to S14 also uses disk 4—can't overlap
write S14	451	575	

This sequence takes **575 ms.**

6. *(15 points)* ***<u>Coherence protocols</u>***

a. *(4 points) In a snooping coherence protocol, why must each processor transmit read or write misses to all other processors? What potential performance problems arise in systems using this type of coherence protocol?*

<u>Solution:</u> In a snooping protocol, the sharing state for a block is maintained in each cache with a copy of that block, with no way for one processor to know if other processors have a copy. A transaction that can potentially change the block's state in other processors' caches must therefore be transmitted to every processor. These protocols therefore do not scale well as the number of processors increases—increasing the number of processors only increases the amount of information being broadcast.

b. *(3 points) Explain when a Write Back message would be sent in a directory protocol.*

<u>Solution:</u> Write Back messages are sent in response to Fetch or Fetch/Invalidate messages—messages that request the most up-to-date copy of a block. Those messages are used when (i) a block is transitioning from the exclusive state to the shared state, so multiple caches will now have copies of the block, or (ii) ownership of an exclusive block is changing from one processor to another, so the new owner needs the most up-to-date data within that block

c. *(3 points) How can false sharing misses be avoided in multiprocessor systems?*

<u>Solution:</u> False sharing misses occur two caches share different data that reside within the same cache block, and one processor writes that block, thus invalidating the other processor's data. False sharing misses would not occur if the data were in different blocks, so any solution that allocates the unshared data to different blocks would help avoid the miss. The simplest solution would be to use smaller cache block sizes, thus decreasing the likelihood of false sharing.

6 (continued)

d. *(5 points) Say we have a multi-processor system with three nodes, P0, P1, and P2, 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*	*P2*	*Dirty*
A	*1*	*1*	*0*	*0*

If P2 now attempts to write block A, what messages are sent between the nodes and directory to ensure P2 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 shared state—P0 and P1 both have copies of the block.

The messages involved in this transaction are therefore:

- Write Miss(P2, A): P2 indicates to the directory that it wishes to write this block.
- Invalidate(A): The directory sends invalidate requests to both P0 and P1, indicating that another processor is modifying the block, thus making any other copies out of date.
- Data Value Reply(<data>): The directory can now send the up-to-date values for the block to P2; once this reply is received, P2 will proceed with its write.