NAME:

Answer all questions. Do the work on these pages, using the back of the pages if necessary. Show all your work for partial credit. State your assumptions if you think there is any ambiguity. And please be concise -- no core dumps.

If you have any questions on these questions (other than what is the answer), ask us NOW.

And please write legibly and big.

Question	Part	Max	Grade
1	a	2	
	b	4	
2	a	2	
	b	2	
	c	6	
3	a	2	
	b	2	
4	a	3	
	b	4	
5	a	2	
	b	4	
	c	7	
	d	1	
	e	1	
6	a	2	
	b	1	
	c	1	
	d	3	
7		8	
Total		57	

1) a) [2 points] What are sub-blocks?

A sub-block is the portion of a cache block that is transferred from memory on a cache miss.

You could have said something else about sub-blocks, like a valid bit per sub-block. But you needed to say something other than they are portions of a block, which you can tell just from the name, even if you know nothing about them.

b) [4 points] Using sub-blocks combines advantages of both large and small cache blocks and therefore is often a good compromise design. What are these advantages? Name at least one that applies to each size (large/small).

Sub-blocks have (at least) three advantages:

They allow you to have the spatial locality and the fewer tags of a large cache block, but the low latency (low transfer time) and low false sharing overhead of a small cache block.

2) a) [2 points] In a cache hierarchy of two caches, what is the goal of the first-level cache?

The goal of the first-level cache (in a two-cache hierarchy, the one closest to the CPU) is to keep have a small cycle time, preferably that of the CPU.

b) [2 points] What is the goal of the second-level cache?

The goal of the second-level cache (in a two-cache hierarchy, the one closest to the bus) is to keep traffic off the processor-to-memory (or system) bus.

c) [6 points] Given this goal, how would you design the second-level cache? Limit your answer to three configuration parameters. For each one, say:

- what it is and
- how does it meet the goal in part b)?

You could have said any three of the following:

A large cache size, to keep the miss rate down.

A set associative cache, for the same reason.

A large, direct mapped cache, because at very large sized caches, the difference in miss rate between a direct-mapped cache and a set-associative cache is extremely tiny. So there is less justification in spending the extra hardware or increasing the access time.

A large block size, because it will exploit spatial locality and the cache is large enough to avoid conflict misses.

A physically-addressed cache. Since fast access time is not a concern, there is time to do address translation. Secondly, you can avoid flushing a physically-addressed cache on a context switch. Since the cache is large, flushing will cause a lot of bus traffic. Third, snooping must be done on physical addresses.

A write-back memory update policy, to reduce writes to memory.

Associate a victim cache with it, to reduce bus traffic.

Use sub-block placement, so you increase the probability that you bring in from memory only the words you actually access.

A unified cache, because it has a hit ratio that is higher than the sum of the hit ratios of two caches (separate instruction and data) half its size.

3) a) [2 points] In the `regfile` module of our R2000 infrastructure, a rising clock edge triggers a register write (the Verilog code sequence is below). Why?

```
always @(posedge clk)
   if (we2 && index2 != 0)
      rf[index2] <= #5 value2;
```

```
So that two register file accesses (to arbitrary registers)
can be done in the same clock cycle. The write is done in the
first phase to reduce the hazard penalty when the write and
read are to the same register.
```

b) [2 points] In the `regfile` module of our R2000 infrastructure, a change in the value of `rf[index0]` triggers a register read (the Verilog code sequence is below). Why?

```
always @(index or rf[index0])
   value0 <= #5 rf[index0];
```

```
So that a particular register can be both written and read in
the same clock cycle.
```

4) a) [3 points] Dynamically scheduled processors execute instructions out-of-order. Explain how they do this.

Dynamically scheduled processors issue instructions when their operands have been computed or loaded from memory. This means they can execute instructions that are behind (in the compiler-generated program order), but independent of, other instructions that have stalled. They do this by dispatching instructions to reservation stations (that may or may not be associated with particular functional units), where they sit until their operands are ready. Then they issue them to the functional units. To break output and anti-dependences, all architecture registers are renamed to the processor's physical registers.

b) [4 points] Even though dynamically scheduled processors execute instructions our-of-order, they retire them in-order. What are two reasons they do this?

You could have said any two of the following:

to preserve precise interrupts.

to preserve all types of data hazards.

to prevent speculative instructions on a mispredicted path from completing.

5) a) [2 points] What is the cache coherency problem?

The cache coherency problem occurs when a reading processor does not get the most current copy for an address -- or -- a writing processor does not communicate the write to the other processors.

b) [4 points] Name and describe a cache coherency protocol for preventing the cache coherency problem from occurring.

You could have said either:

Write-invalidate: coherency is maintained by making sure the writing processor has the only cached copy of a block. When a processor does a write, it invalidates all other caches copies. Or you could have described MESI in these terms, saying it was a write-invalidate protocol.

Write-update: coherency is maintained by broadcasting all writes to shared data to all processors. When a processor does a write, it broadcasts the write on the bus if some other processor raised the shared bus line on the last operation for this block.

c) [7 points] Assuming that you are using the four states of the MESI cache coherency protocol, fill in the state for a cache block in each of the processors (P0, P1, P2) below for each of the memory operations.

Operations	P0	P1	P2
Initially	invalid	invalid	invalid
P1: loads B	invalid	private clean	invalid
P2: loads B	invalid	shared clean	shared clean
P0: stores B	private dirty	invalid	invalid
P1: loads B	$\boxed{\text{shared clean}}$	shared clean	invalid
P1: stores B	invalid	private dirty	invalid

d) [1 point] Does memory have to be updated during these memory operations? If so, **circle the operation** during which this update should take place. If not, ignore this question.

e) [1 point] Suppose that the MESI protocol had a fifth state, called Shared Dirty. Shared Dirty is the state that a private dirty block goes into after its snoop hears a bus read for the block. The block is dirty, because memory is not updated; it is shared, because now two caches have copies. (Memory is updated when the block is replaced.) The state of the reader's block is shared, as usual. **Draw a square around the state values** above that would be Shared Dirty, if we were using the new version of MESI.

6) a) [2 points] What are invalidation (or coherency) misses?

An invalidation miss is a miss on a block that previously resided in one of the caches in a shared memory multiprocessor, but was invalidated by a writing processor. They occur with write-invalidate cache coherency protocols.

b) [1 point] Can they occur with true sharing?

yes

c) [1 point] Can they occur with false sharing?

yes

d) [3 points] What is a design/technique that can reduce invalidation misses?

You could have said any two of the following:

Coherency misses can be reduced by decreasing the block size. If the block size is only one word, the invalidations of one processor will not cause coherency misses by another unless they are accessing the same word.

Coherency misses can be reduced by implementing a cache coherency protocol that invalidates sub-blocks. If the sub-blocks are one word, invalidation misses will only occur for true sharing.

Coherency misses can be reduced by grouping together (either by the compiler or the applications programmer) data that is accessed by individual processors. Then coherency misses will only occur to data that is true shared.

Although coherency misses will decrease with smaller cache sizes, since there will be less opportunity for data to be invalidated. This is not a suggested solution, since there are more compelling reasons for having large caches.

Use a write-update cache coherency protocol.

7) [8 points] Consider the following memory sub-system:

- 48 bit virtual address
- 32 bit physical address
- 4KB page size
- 4-way set-associative TLB
- 128 page table entries in the TLB, each of which only contains a page frame number
- 512 KB, direct-mapped, physically-addressed cache with 16 byte blocks.

Show the size of the labeled fields in the following diagram of the TLB and the cache (the next page). Show your formulas for calculating the answers (on the page after that). An incorrect answer will be counted as correct, if your formula for obtaining it is correct. Correct numeric answers will not be counted if the formula is incorrect, i.e., don't just guess. Place your answers in the appropriate slots below.

$$c = \log_2(\text{page size}) = \log_2(4K) = 12$$

$$b = \log_2\left(\frac{\text{entries}}{\text{associativity}}\right) = \log_2\left(\frac{128}{4}\right) = 5$$

$$a = 48 - b - c = 31$$

$$d = 32 - c = 20$$

$$e = c = 12$$

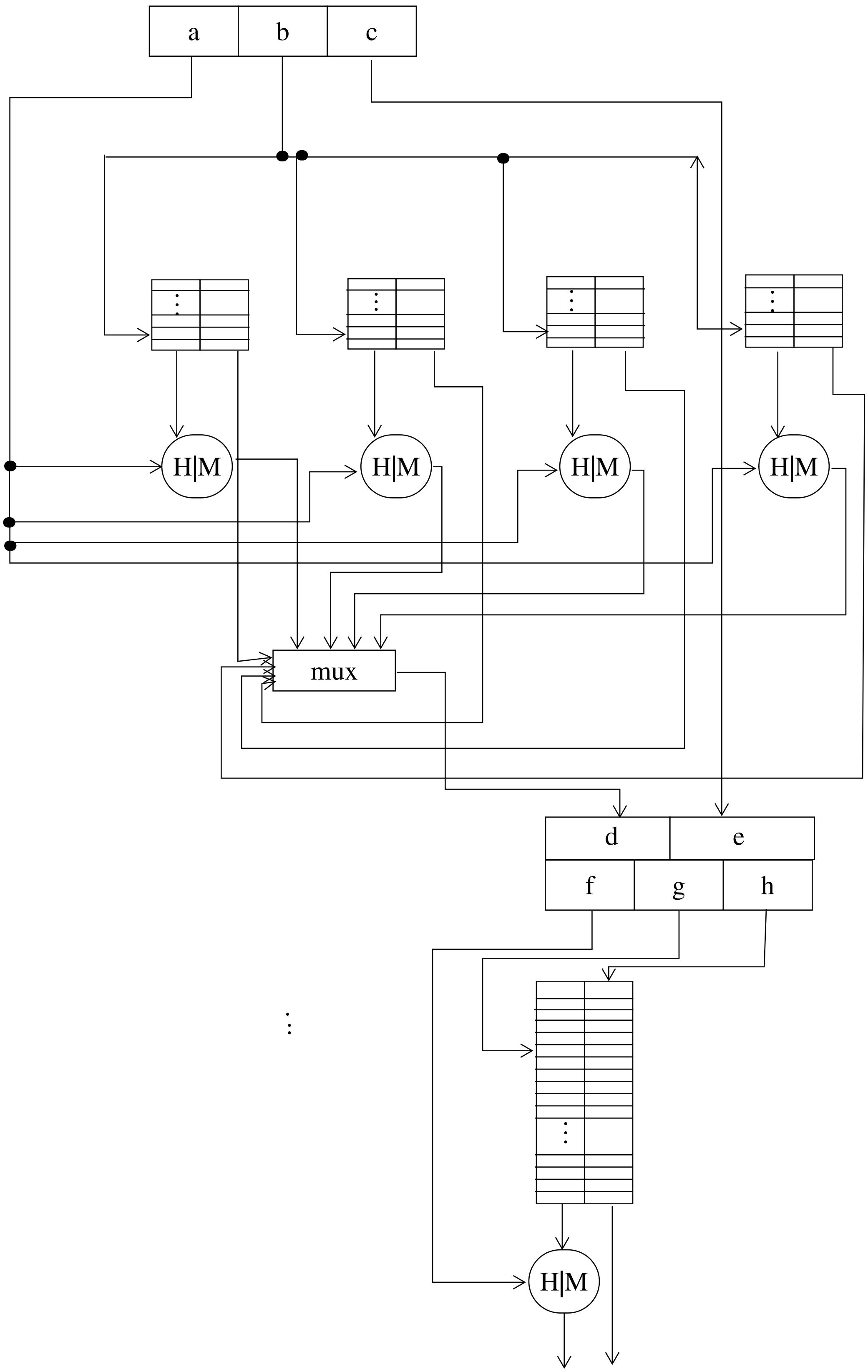
$$h = \log_2(\text{block size}) = \log_2(16) = 4$$

$$g = \log_2\left(\frac{\text{cache size}}{\text{associativity} \cdot \text{block size}}\right) = \log_2\left(\frac{512KB}{1 \cdot 16}\right) = 15$$

$$f = 32 - f - g = 13$$

a 31 b 5 c 12 d 20

e 12 f 13 g 15 h 4


a
b
c
H|M
H|M
H|M
H|M
mux
d
e
f
g
h
H|M