Indian Institute of Technology, Delhi

FALL, 2013

CSL 211: COMPUTER ARCHITECTURE

Major Exam

Two Hours

NOTE:
- All answers need to be brief and to the point.
- Please make any assumptions that you deem to be reasonable.
- We need exquisite detail, and clarity in every answer.
- You need to write your answer in the answer box (Ans:). We will **NOT GRADE** your answer, if it is not written in the box.
- Every answer needs to be written neatly and cleanly in the space provided for it.
- Use proper handwriting, and do not write anything in the margins.
- Note that in most questions, there is no part marking.
- You can use rough sheets. Do not attach them with this paper.
- You are not allowed to carry any electronic gadget including calculators.
- This question paper **NEEDS TO BE SUBMITTED**. Do not take it with you.

Total Marks: 60
Total Number of Pages : 10

Name:	Group No:	Entry No:

Marks:	1	2	3	4	5	6	7	8	9	Total

Easy

1. *NO PART MARKING*

Answer the following: (15 marks)

i) The two Z series instructions that we add in our ISA are $\boxed{\text{Ans: } \textit{movz} \text{ and } \textit{retz}}$

ii) Interrupts are typically (precise/imprecise). $\boxed{\text{Ans: precise}}$

iii) The TLB is required to ensure correctness. (Yes/No) $\boxed{\text{Ans: no}}$

iv) Tripling the size of a cache, typically decreases its miss rate by a factor of $\boxed{\sqrt{3}\ (1.73)}$

v) For temporal locality, we use the (address distance/ stack distance) $\boxed{\text{Ans: stack distance}}$

vi) For spatial locality, we use the (address distance/ stack distance) $\boxed{\text{Ans: address distance}}$

vii) We require fences for which consistency model? $\boxed{\text{Ans: weak consistency}}$

viii) Which I/O signalling protocol has transitions on both 0s and 1s. $\boxed{\text{Ans: Manchester encoding}}$

ix) We set the value of a shared variable in a function, and then we return from the function. At this point of time is another thread guaranteed to see the updated value of the variable in a weak consistency model? $\boxed{\text{Ans: no}}$

x) In large multicore processors, we divide the set of processors and cache banks into $\boxed{\text{Ans: tiles}}$

xi) Coherent caches that do not use a shared bus, use a $\boxed{\text{Ans: directory}}$ instead.

xii) How many devices (including hubs) can be connected to a single USB host? $\boxed{\text{Ans: 127}}$

xiii) The PCI express protocol is (half duplex/ full duplex) $\boxed{\text{Ans: full duplex}}$

xiv) A platter has $\boxed{\text{Ans: 2}}$ recording surfaces.

xv) In a floating gate transistor, the threshold voltage (increases/decreases) $\boxed{\text{Ans: increases}}$ when it is programmed.

Medium

2. You have a cache with the following parameters: size (n bytes), associativity (k), block size (b) bytes. Assume a 64 bit memory system. (5 marks)

i) What is the size of the tag in bits? $\boxed{\text{Ans: 64 - log(n) + log(k)}}$ (3 marks)

ii) What is the size of the set index in bits? $\boxed{\text{Ans: log(n) - log(b) - log(k)}}$ (2 marks)

Note to TAs Let us have some amount of part marking. Use your judgement, and use a consistent scheme.

3. Consider a 8-entry fully associative cache that follows the LRU replacement scheme. Now, we run a program that has the following sequence of memory accesses (the numbers denote the word address):
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 22, 30, 21, 23, 31
Assume that initially the cache is empty. What are the contents of the cache after the end of the execution of the program? (3 marks)

$\boxed{\text{Ans: 28,29,22,30,21,23,31,27}}$

Note to TAs: Per mistake (deduct 1 mark), Order of entries not important

4. We have a producer-consumer interaction between processes A and B. A writes data into a shared address space that B reads. However, B should never be allowed to write anything into that shared space. How can we implement this using paging? How do we ensure that B will never be able to write into the shared space? (5 marks)

- Map the same physical page to the virtual memory in both the processes. (2)
- Mark the page as read-only in B's TLB (1), and page table (2).
- (1) $\rightarrow$ 1 mark, (2) $\rightarrow$ 2 marks

5. Answer the following: (7 marks)

NO PART MARKING

i) How is the angular velocity (ω) related to the linear velocity (v) in a hard disk? $\boxed{v = \omega r}$ (1 mark)

ii) How many Program/Erase cycles can a flash device typically tolerate? $\boxed{\text{Ans:}\quad\quad\quad}$100,000 (1 mark)

iii) A NAND flash device can erase data at the granularity of $\boxed{\text{Ans:}\quad\quad\quad}$blocks. (1 mark)

iv) In the Snoopy protocol, why do we write back data to the main memory upon a M to S transition? (2 marks)

Answer: To ensure seamless eviction from the S state. Otherwise, it is possible that modified data might get lost, and never get written to the lower level. This violates the first axiom of cache coherence.

v) Why is NAND flash preferred over NOR flash? (2 marks)

Answer: NAND flash is much more compact because we try to reduce the number of wires as much as possible. Specifically, in a NOR flash device, every transistor is connected to a bit line. However, in a NAND flash device, we group 16 transistors, and they have only one connection to the bit line. This strategy ensures that the wiring overhead is minimised, and we can store more data per area.

Hard

6. Assume a cache that has n levels. For each level, the hit time is x cycles, and the local miss rate is y per cycle. (5 marks)

(a) What is the recursive formula for the average memory access time?

(b) What is the average memory access time as n tends to ∞?

Answer:

(a) $T(n) = x + y \times T(n-1)$

(b) $T(n) = \frac{x}{1-y}$

7. Draw the receiver circuit diagram (as shown in the slides) for the mesochronous, source synchronous, and asynchronous buses. (7 marks)

8. Assume that we want to implement an instruction called *MCAS*. The *MCAS* instruction takes k (known and bounded) memory locations as arguments, a set of k old values, and a set of k new values. Its pseudo-code is shown below. We assume here that *mem* is a hypothetical array representing the entire memory space. (6 marks)

```
/* multiple compare and set */
boolean MCAS(int memLocs[], int oldValues[], int newValues[]){
        /* compare */
        for(i=0; i < k; i++) {
                if(mem[memLocs[i]] != oldValues[i]) {
                        return false;
                }
        }

        /* set */
        for(i=0; i < k; i++) {
                mem[memLocs[i]] = newValues[i];
        }

        return true;
}
```

The challenge is to implement this instruction such that it appears to execute instantaneously. Let us look at some subtle cases. Assume that we want to write (4,5,6) to three memory locations if their previous contents are (1,2,3). It is possible that after writing 4, and 5, there is a small delay. During this time another thread reads the three memory locations, and concludes that their values are 4,5, and 3 respectively. This result is incorrect because it violates our assumption that *MCAS* executes instantaneously. We should either read (1,2,3) or (4,5,6).

Now, let us look at the case of reading the three memory locations. Let us say that their initial values are 1,2, and 0. Our *MCAS* instruction reads the first two locations and since they are equal to the old values, proceeds to the third location. Before reading it, a store operation from another thread changes the values of the three locations as follows. (1,2,0) $\rightarrow$ (5,2,0) $\rightarrow$ (5,2,3). Subsequently, the *MCAS* instruction takes a look at the third memory location and finds it to be 3. Note that the three memory locations were never equal to (1,2,3). We thus arrive at a wrong conclusion.

How should we fix these problems? We want to implement a *MCAS* instruction purely in hardware, which provides an illusion of instantaneous execution. It should be free of deadlocks, and should complete in a finite amount of time. How can we extend our coherence protocols to implement it?

Research

9. Assume a processor with a weak consistency model. Let us run a "properly labelled" program on it. A *properly labelled*(PL) program does not allow conflicting accesses (read-write, write-read, or write-write) to a shared variable at the same time. For example, the following code sequence is not properly labeled because it allows x to be modified concurrently. (7 marks)

```
Thread 1
----------
x = 0

Thread 2
---------
x = 1
```

In reality, the coherence protocol orders one write access before the other. Nevertheless, both the threads **try** to modify x concurrently at the programmer's level. This is precisely the behaviour that we wish to avoid.

In a PL program, two threads do not **try** to modify x at the same time. This is achieved by having two magic instructions known as *lock* and *unlock*. Only one thread can lock a memory location at any point of time. If another thread tries to lock the location before it is unlocked, then it stalls till the lock is free. If multiple threads are waiting on the same lock, only one of them is given the lock after an *unlock* instruction. Secondly, both the *lock* and *unlock* instructions have a built in $fence$ operation, and all the *lock* and *unlock* instructions execute in program order. The PL version of our program is as follows:

```
Thread 1
-----------
lock(x)
x = 0
unlock(x)

Thread 2
-----------
lock(x)
x = 1
unlock(x)
```

We can thus think of a lock-unlock block as sequential block that can only be executed by one thread at a given time. Now, prove that all PL programs running on a weakly consistent machine have a sequentially consistent execution. In other words we can interleave the memory access of all the threads such that they appear to be executed by a single cycle processor that switches among the threads.

Suggested methodology: (need not be followed in totality, details are sketchy)

- Draw a circle(node) for each dynamic instruction.
- Draw a circle(node) for each lock-unlock block.
- Connect two nodes across threads if they have a read-write dependence. (write $\rightarrow$ read)
- Connect the lock-unlock blocks of each thread with edges because they execute in program order.
- Since each lock-unlock block contains fences, add additional edges.
- The edges represent (**happens before**) relationships. $A \rightarrow B$ means that instruction B started its execution after A finished.
- Now, prove that it is possible to arrange the nodes in a sequence that respects program order without violating any happens before relationships.

HINT : Connect consecutive nodes in a thread with an edge (for signifying program order). If you can show that the graph does not have any cycles, we are done.