CSE 378 Final 3/18/10 Sample Solution

Name _____

There are 10 questions worth a total of 100 points. Please budget your time so you get to all of the questions. Keep your answers brief and to the point.

Copies of the MIPS reference card ("green card") have been handed out separately and you can refer to that. Otherwise the exam is closed book, closed notes, closed neighbors, closed electronics, closed telepathy, etc., but open mind… You won't need a calculator either, so please keep that stowed in the overhead compartment or under the seat in front of you – or whatever is equivalent in this room.

Please wait to turn the page until everyone is told to begin.

Score _____

1 _____ / 20

2 _____ / 14

3 _____ / 6

4 _____ / 5

5 _____ / 7

6 _____ / 12

7 _____ / 12

8 _____ / 12

9 _____ / 6

10 _____ / 6

Some powers of 2 (in case you need them)

n	2^n
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1 024
11	2 048
12	4 096
13	8 192
14	16 384
15	32 768
16	65 536
20	1 048 576
24	16 777 216
30	1 073 741 824
31	2 147 483 648
32	4 294 967 296

Question 1. (20 points) To warm up, write a MIPS assembly language function that is equivalent to the following C function:

```
int thing(int x, int y) {
   int a, b;
   b = x+y;
   a = compute(x);
   if (a > 0)
      return a;
   else;
      return b;
}
```

You should follow the standard MIPS conventions for register usage, function calls, and stack frames. Assume that function `compute` is an integer-valued function that is written elsewhere and can be called with an appropriate jump to the label `compute`.

```
# on entry: $a0 = x, $a1 = y

thing:  addi    $sp, $sp, -8      # allocate stack frame
        sw      $ra, 0($sp)       # save return address
        add     $t0, $a0, $a1     # b = x+y
        sw      $t0, 4($sp)       # save b
        jal     compute           # a = compute(x)
        slt     $t0, $0, $v0      # t1 = "0 < a"
        bne     $t0, $0, exit     # return a if 0 < a
        lw      $v0, 4($sp)       # return b if not
exit:   lw      $ra, 0($sp)       # restore return address
        addi    $sp, $sp, 8       # release stack frame
        jr      $ra               # return
```

There are, of course, many possible solutions. There were, however, a couple of unusual things we found while reading through the exams.

- **A surprising number of solutions initialized registers to 0 before using them with instructions like `add $t0,$0,$0`. That makes no more sense than initializing a variable to 0 in a C or Java program before assigning a different value to it. But since it doesn't make the code fail, there was no penalty.**
- **Most solutions allocated stack frames to save values, but the code to allocate and free the stack frame was in all sorts of odd places. It's much more straightforward to do it at the beginning and end of the function, and that's how compiled code almost always works.**

Question 2. (14 points) Suppose we have a cache that has 16-word (64-byte) blocks. The total size of the cache is 64 blocks (= 4,096 bytes), and the cache is direct mapped. Now, suppose we have a C program that contains a 32x32 array of doubles. Each double-precision number occupies 2 words (8 bytes). C arrays are stored in row-major order: row 0 is followed in memory by row 1, then row 2, etc.

```
double matrix[32][32];
```

(a) What is the cache miss rate if we use the following code to store 0's in the array? You can give an appropriate formula or brief explanation and do not need to calculate the final numeric answer.

```
for (r = 0; r < 32; r++)
   for (c = 0; c < 32; c++)
      matrix[r][c] = 0.0;
```

Miss rate = 1/8.

This version goes through the memory locations assigned to `matrix` sequentially. There will be a miss when the first variable in each cache block is accessed. The remaining variables in each block will found in the cache. Since each block holds 8 doubles, 1/8 of the array elements will cause a miss.

(b) What is the cache miss rate if we use the following code to store 0's in the array? (same code, except the order of the two outer loops is reversed) Again, it's ok to just give a formula.

```
for (c = 0; c < 32; c++)
   for (r = 0; r < 32; r++)
      matrix[r][c] = 0.0;
```

100%

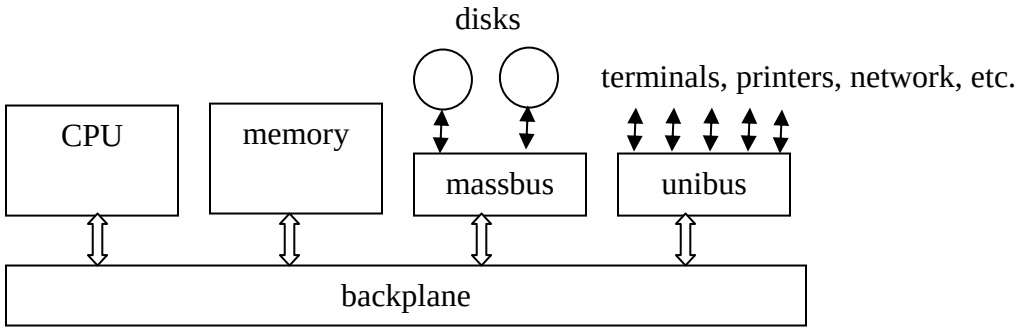
In this version, each reference to an array element causes a cache miss. The first time through the outer loop we only reference the first column of each row. Since all of these are in different cache blocks, there will be a fault on each one. Further, because the cache can only hold half of the array, by the time we reference a cache block for the second time on the next iteration of the loop it will have been flushed from the cache by earlier references to different parts of the array.

(continued next page)

Question 2. (cont.) (c) If the cache were 2-way set associative instead of direct mapped, but had the same size, would the miss rate for part (b) change significantly? Why or why not?

No change – still 100%. The cache still does not have enough space to hold the entire array, and since we are still accessing it in column-major order, each cache block will be flushed by the time we reference it again.

Question 3. (6 points) Some things never change. The machine in the Allen Center lobby is a Digital Equipment Corp. VAX computer. The basic architecture included two main busses for connecting external devices: a Massbus for disks and tape drives, and a Unibus to connect terminals, networks, printers, and similar things.



Why do you think they had two different busses? Why not attach everything to one bus, or have several identical busses to connect the various I/O devices?

The fast bus used to service the high-speed devices was more expensive and complex than the slower bus. Using the Massbus for everything would have required expensive interfaces for the slower devices, and it would have been more difficult to handle the fast, streaming devices while also dealing with the slower ones. The Unibus was cheaper and provided adequate service for the slower deices, but wouldn't have been able to keep up with the faster disks and tapes.

Question 4. (5 points) For each of the following, put an X in the box that is closest to the correct order of magnitude for the speed of that operation on a typical current, consumer desktop or laptop computer. (You may assume "typical" ≈ 2GHz processor)

What	100ms	10ms	1ms	100µs	10µs	1µs	100ns	10ns	1ns	100ps	10 ps
ALU add operation									X		
Read CPU register									X		
Read cache memory									X		
Read main memory							X	X			
Read from disk		X									

Main memory cycle time is on the order of dozens of ns, so we gave credit for either 100ns or 10ns (even though 10ns is closer to the right order of magnitude).

Question 5. (7 points) All modern processors have two modes: system mode, where all operations are permitted, and user mode, where certain operations are disabled. Ordinary programs run in user mode so they cannot make changes to parts of the system that would allow them to bypass protections, read or write unauthorized data, or otherwise interfere with the operation of the system. For each of the following, check the box in the "user mode" column if it is safe to allow these operations in user mode. Otherwise check system mode.

Operation	OK in user mode	System mode only
Store a value in a general register like $t0	X	
Disable interrupts so nothing will interrupt the current process while it does something important		X
Change the page table register that points to the current program's page tables		X
Change the program counter	X	
Initiate an I/O operation on a raw disk device using memory-mapped I/O		X
Execute a compare-and-swap instruction to grab an exclusive lock for some concurrent data structure	X	
Switch from user to system mode	X	*

***A user program can't be allowed to switch to system mode and continue execution, but it does need to be able to switch to system mode as part of a call to get operating system services. The question wasn't clear as to which was meant, so we gave credit for either answer.**

Question 6. (12 points) Suppose we have a processor that has the following CPI figures for different kinds of instructions:

Kind	CPI
load/store	5
ALU	1
branch	2

Now, suppose we have two programs that have the following percentage breakdown of instructions executed:

Application	% load/store	% ALU	% branch
A	25	60	15
B	20	70	10

(a) What is the CPI for Application A? (As before, it's ok to just give the formula and not crank out the final answer.)

$CPI_A = 5 * 0.25 + 1 * 0.6 + 2 * 0.15 = 2.15$

(b) Now suppose we design a much better memory system that reduces the CPI for load/store instructions on our processor to 2. How much does application B speed up? (Formula is ok here too.)

Original $CPI_B = 5 * 0.2 + 1 * 0.7 + 2 * 0.1 = 1.9$

New $CPI_B = 2 * 0.2 + 1 * 0.7 + 2 * 0.1 = 1.3$

The speedup is: change / old = 0.6 / 1.9 = 31.6%. We gave credit for any answer that showed a reasonable comparison between the old and new CPI values, even if it wasn't expressed exactly like this.

Question 7. (12 points) The Round Number Disk Company manufacturers a disk with the following characteristics:

- Rotation speed: 6000 rpm
- Seek time (average): 10 ms
- 500 bytes per disk sector
- 200 sectors per track
- Overhead time for each I/O request: 2 ms.
- Data can be transferred as fast as it moves under the disk read/write heads

Give equations for each of the following. You do not need to compute the final answer.

(a) Time needed to read one disk sector at a random location on the disk:

The rotation time of the disk is 60/6000 sec = 10 ms.

T = seek + rotation delay + transfer + overhead

= 10 ms + 10 ms/2 + 10 ms/200 + 2ms = 17.05 ms.

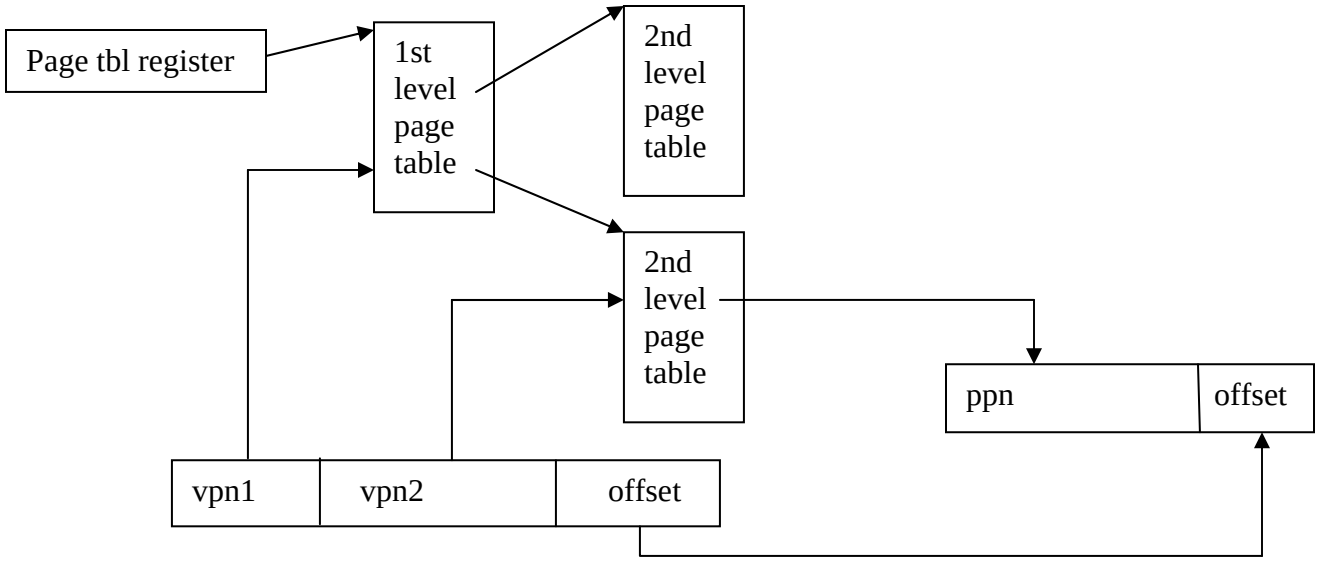
(b) Time needed to read all of the sectors on a single track sequentially (i.e., time to read a full 100,000 byte track in order):

There were a couple of ways to answer this question (alas). If we assume that transfers need to start at the beginning of a specific sector, then the rotational latency stays the same and the transfer time becomes 10 ms, or 1 revolution of the disk. In that case the total time is 27 ms.

If we assume that we can start the transfer anywhere on the track and reassemble the data after it is transferred (which is a reasonable assumption), then the rotational latency is 0 and the total time is 22 ms.

On a real disk, the transfer would probably have to start at the beginning of a sector, even if we start reading anywhere, so there would be a small latency term equal to the time for 1/2 sector to move under the heads.

Question 8. (12 points) We would like to figure out the details of a virtual memory system. We have a 2-level page table:



The memory system has the following characteristics:

- 2 GB virtual address space
- 8 GB physical address space
- 16 KB pages
- Each 2nd level page table occupies exactly one (1) physical page.

Fill in the following:

Number of bits in the offset part of the address _____**14**_____

Number of physical page number (ppn) bits _____**33-14 = 19**_____

Number of bytes in each page table entry (smallest power of 2 needed to hold ppn, valid, and dirty bits) ___**4 (21 bits)**_____

Number of entries in each 2nd level page table _____**16K/4 = 4096**_____

Number of bits in vpn2 part of virtual address used to index 2nd level page table __**12**___

Number of bits in vpn1 part of virtual address used to index 1st level page table ___**5**____

Number of entries in 1st level page table ____**32**_____

Number of bytes in 1st level page table _____**128**_____

Question 9. (6 points) Why do systems have 2- and 3-level page table designs like the one in the previous question? After all, the design and implementation of a virtual memory system with multiple levels of page tables is significantly more complex than using a single page table. What problem or problems are solved by using a multiple level page table?

To save space. For example, in a typical system with 32-bit addresses and 4K pages, a flat page table would need 2^{20} or roughly 1 million entries, even though most programs use only a tiny fraction of the available address space.

Question 10. (6 points) Both the cache and the TLB perform a similar function – they are small, fast, expensive memories that hold currently active data that normally resides in larger, slower, cheaper storage. One significant difference is that TLBs are often fully associative memories, while caches almost never are. Explain why this is. Why is a fully associative memory an appropriate design choice for a TLB, yet this is rarely a good idea for a cache?

TLBs are typically small, containing far fewer entries than a cache. A fully associative memory is reasonable for a TLB, while for a cache the logic needed for a fully associative memory would be too complex, expensive, and slow.