# CSE 378 Final Exam 3/14/11 Sample Solution

Name _____

There are 8 questions worth a total of 100 points. Please budget your time so you get to all of the questions – don't miss the short questions at the end. 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 _____ / 18

2 _____ / 8

3 _____ / 10

4 _____ / 10

5 _____ / 18

6 _____ / 14

7 _____ / 12

8 _____ / 10

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.** (18 points) To warm up, write a MIPS assembly language function that is equivalent to the following C function that computes the sum of the sequence of numbers x + (x+1) + … + y:

```
int sum(int x, int y) {
  if (x > y)
    return 0;
  else
    return (x + sum(x+1,y));
}
```

You should follow the standard MIPS conventions for register usage, function calls, and stack frames. Your assembly language code must be equivalent to the given code – you cannot rearrange the computation to calculate the result differently (for example, you cannot replace the recursive function calls with a loop). You may use regular MIPS assembly language pseudo-instructions in your code if you wish.

```
      # $a0 = x, $a1 = y
sum:
      ble  $a0,$a1,general   # jump if x <= y
      li   $v0,0             # return 0 since x > y
      jr   $ra

      # general case: recursive call
general:
      addi $sp,$sp,-8        # allocate stack frame
      sw   $ra,0($sp)        # save return address, x
      sw   $a0,4($sp)
      addi $a0,$a0,1         # call sum(x+1,y)
      jal  sum               # recursive result in $v0
      lw   $t0,4($sp)        # reload x
      add  $v0,$v0,$t0       # add x to recursive call
      lw   $ra,0($sp)        # restore return address
      addi $sp,$sp,8         # deallocate stack frame
      jr   $ra               # return
```

**As usual there are many possible solutions. In particular, most MIPS C compilers would use `addiu` and `addu` for addition, but either was acceptable for this problem.**

**Question 2.** (8 points) We have a C program that does extensive matrix manipulation and we're trying to improve the execution speed of the code. One part of the code contains the following nested loop to transpose a N*N matrix:

```
for (j = 0; j < N; j++) {
   for (i = 0; i < j; i++) {
      temp = a[i][j];
      a[i][j] = a[j][i];
      a[j][i] = temp;
   }
}
```

This code is heavily used, so one of the summer interns who has recently taken CSE 378 suggests that it would run faster if it were written this way instead:

```
for (i = 0; i < N; i++) {
   for (j = 0; j < i; j++) {
      temp = a[i][j];
      a[i][j] = a[j][i];
      a[j][i] = temp;
   }
}
```

The intern claims that this will make better use of the cache and run faster. Is this correct? Give a brief technical justification for your answer. In particular, what effect will this change have on cache miss (or hit) rates, if any?

You should assume that the variables `i`, `j`, `N`, and `temp` are stored in registers and not in memory, and that instruction fetches do not interfere with the data cache(s). Also, remember that two-dimensional C arrays are stored in memory in row-major order, with row 0 appearing first, followed by row 1, then row 2, etc.

**The change won't make any significant difference in the cache miss rates. In this particular problem, the inner loop simultaneously references the array in both row-major and column-major order. The memory reference pattern of the nested loops will be the same in either case.**

**Question 3.** (10 points) The following MIPS code could be used to increment the value of two integer variables (similar to "`i++; j++`" in Java).

```
(a)  li   $t0,1              # load 1 into $t0
(b)  lw   $t1,100($t5)       # load first variable
(c)  add  $t1,$t0,$t1        # increment
(d)  sw   $t1,100($t5)       # store
(e)  lw   $t2,104($t5)       # load 2nd variable
(f)  add  $t2,$t0,$t2        # increment
(g)  sw   $t2,104($t5)       # store
```

(a) Suppose these instructions are executed on a 5-stage pipelined MIPS processor with the usual hazard detection and forwarding implemented. Do any hardware stall cycles (pipeline bubbles) need to be inserted by the processor to avoid incorrect results during execution? If so, how many and where are they needed? (i.e., n cycles inserted between (x) and (y).)

**The processor needs to stall for a cycle between (b) and (c) and again between (e) and (f). In both of these cases the result of a load is needed as an input value for the next instruction, and even with forwarding, the value is not available from the memory unit in time to forward it back to the execution cycle without waiting.**

(b) Without changing what the code does, can the instructions be rearranged to reduce the number of stalls needed if there are any? If so, describe the rearrangement and explain how many stall cycles (if any) are needed in your revised code.

**Any rearrangement that pushes the load instructions early so their values are not needed on the very next instruction will eliminate the need for any stalls. An easy way to do this is to order the instructions (b) (e), (a), (c), (d), (f), (g).**

**Question 4.** (10 points) Suppose we have a processor that has the following CPI figures for different kinds of instructions:

| Kind | CPI |
|---|---|
| memory | 4 |
| ALU | 1 |
| cond. branch | 2 |
| jump | 1 |

Our customers mainly execute two types of operating systems that have the following mix of instructions:

| OS | % memory | % ALU | % branch | % jump |
|---|---|---|---|---|
| Loonix | 30 | 25 | 15 | 30 |
| Playdows | 20 | 30 | 25 | 25 |

Given the available budget we can afford to make one of these two possible changes in the next generation of the processor:

- Reduce the number of cycles needed by memory instructions from 4 to 2, or
- Reduce the number of cycles needed by conditional branches from 2 to 1.

Which change should we make? Give a quantitative argument to support your decision.

**Reduce the number of cycles needed by memory instructions from 4 to 2.**

**An easy way to see this is to look at the contribution of each type of instruction to the CPI before and after the proposed change. For the memory instructions:**

| OS | CPI contribution was | New CPI contribution | CPI improvement |
|---|---|---|---|
| Loonix | 4 * 0.3 = 1.2 | 0.6 | 0.6 |
| Playdows | 4 * 0.2 = 0.8 | 0.4 | 0.4 |

**The change in branch instructions would have the following effect:**

| OS | CPI contribution was | New CPI contribution | CPI improvement |
|---|---|---|---|
| Loonix | 2 * 0.15 = 0.3 | 0.15 | 0.15 |
| Playdows | 2 * 0.25 = 0.5 | 0.25 | 0.25 |

**For both systems, the reduction in CPI is significantly better if we double the speed of the memory instructions.**

**Question 5.** (18 points) On the midterm exam we added a set of conditional branch instructions to the single-cycle MIPS datapath. These branches were similar to the branch instructions already supported by the processor (BEQ, BNE, BLEZ, and BGTZ), except they compared a value in memory to a register instead of comparing two registers. For example:

```
MBEQ  $rs,$rt,label      branch to label if M[R[$rs]] = R[$rt]
MBLEZ $rs,$zero,label    branch to label if M[R[$rs]] <= 0
```

(a) On the diagram on the next page, add support for these instructions to the 5-cycle pipelined datapath. All values required to implement the instructions must be pipelined as necessary, and all other instructions supported by the processor should continue to work. Add anything you need: wires, busses, ports, muxes, and/or logical units; however you may not add additional pipeline stages (it still must remain a 5-cycle processor). Don't worry at this point about timing issues or interactions with later instructions in the pipeline.

(b) Given your solution to part (a), will your changes have any effect on the timing or clock speed of the processor datapath? Why or why not?

**This depends on what was added to the processor in your solution to part (a).**

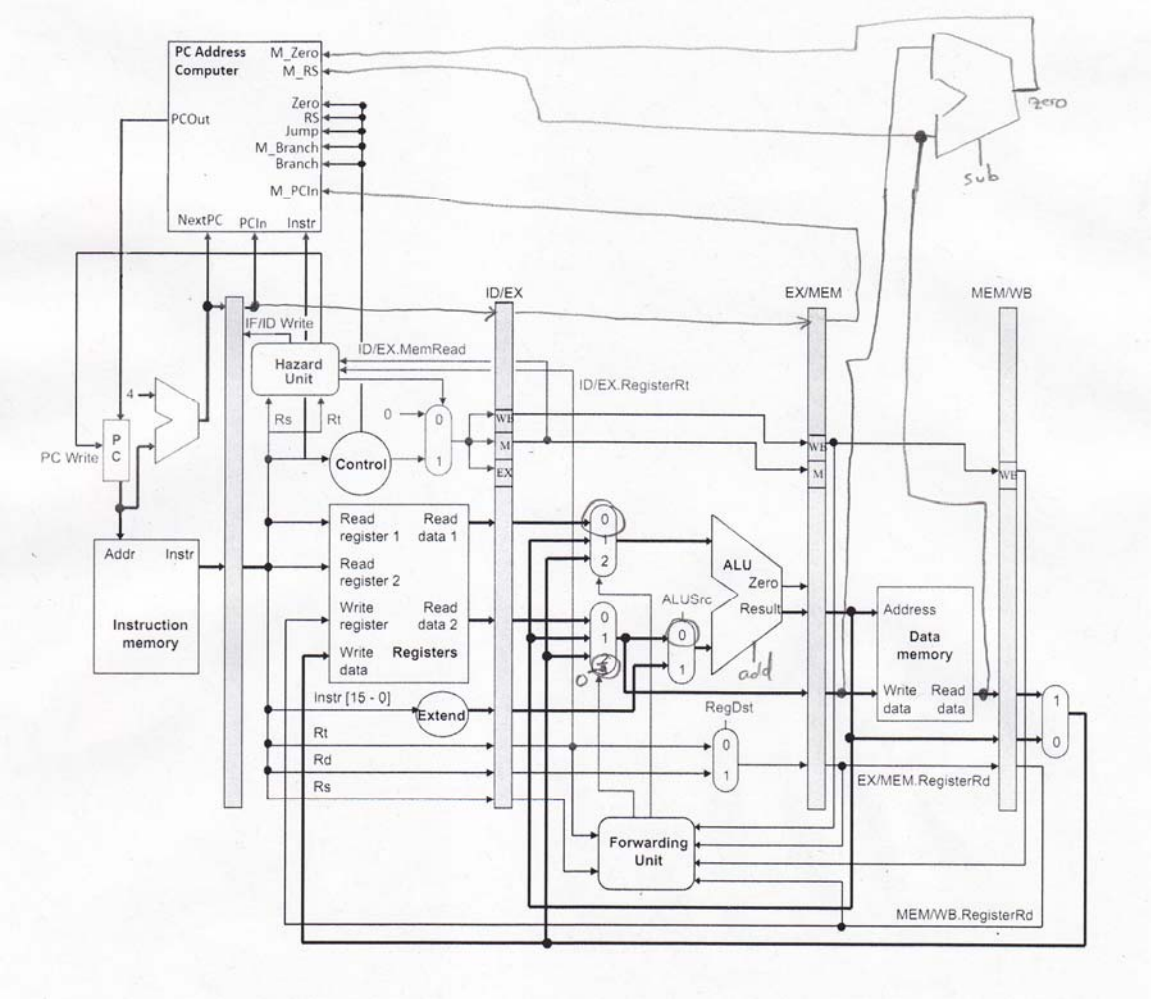
**In the sample solution on the next page an ALU is added to the memory stage. This means there will be an extra delay in that stage and that will limit the possible clock frequency.**

**If your solution makes different changes to the processor your answer to this part of the question was evaluated based on any timing changes that are implied by your solution.**

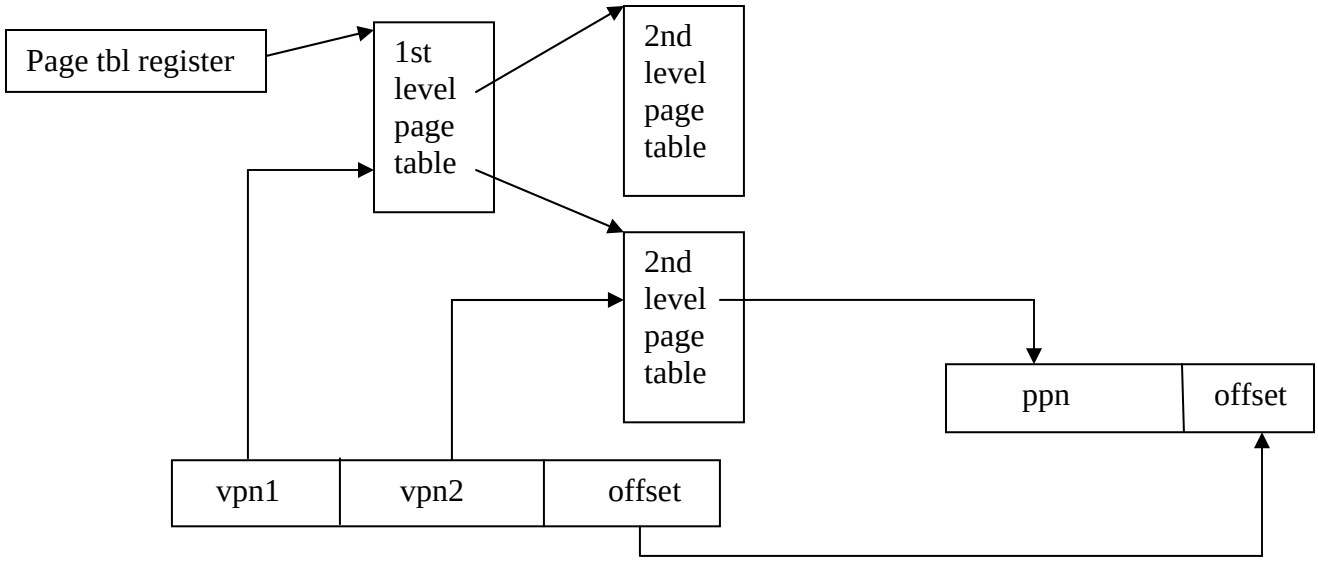
(c) These new conditional branch instructions store a new value into the PC if the branch is taken. Does your solution to part (a) create any new hazards in the pipeline requiring additional forwarding, stalls, or branch delays? Briefly describe any new issues introduced by your design and how they could be handled. You can use either hardware or software solutions, or a combination of both.

**Yes. There are now three branch delay slots rather than one. These must be accounted for in hardware (stalls or flushing instructions that will not be used) or in software (compiler inserted nops or other instructions in the delay slots). There is also a new possible control hazard if a normal control instruction appears after one of the new memory-branch instructions. This either requires a stall in the hardware or care in the compiler to avoid generating instruction sequences that encounter this problem.**

**Question 5. (cont.)** Draw your answer to part (a) on this diagram:



**Question 6.** (14 points) We are writing a simulator for a processor with a conventional 32-bit virtual memory system that has 2-level page tables:




The memory system has the following characteristics:

- 4 GB virtual address space (32-bit addresses)
- 4 GB physical address space
- 4 KB pages
- Each page table occupies exactly one (1) physical page.

Given these parameters, the address offsets are 12 bits, and the virtual page number parts (vpn1 and vpn2) are 10 bits each. There are 1024 entries in each page table.

Each page table entry in both the 1st and 2nd level page tables occupies 32 bits and is formatted as follows:

| 1 | 1 | 10 bits | 20 bits |
|---|---|---|---|
| V | D | unused | physical page number |

The V and D bits indicate valid (1 if present, 0 if not) and dirty (1 if modified, 0 if not).

As we said before, we're writing a simulator for this processor. On the next page, implement in MIPS assembly language a function that will translate a virtual address to a physical address, given the contents of the page table register (the address of the 1st level page table) and the virtual address to be translated. Your function should return the physical address if the translation is possible, or return -1 (`0xffffffff`) if a page fault occurs because of an invalid (V=0) page table entry is encountered.

Feel free to remove this page for reference while working on your solution to the problem.

**Question 6.** (cont) Give your implementation of the virtual-to-physical address translation algorithm using MIPS assembly language below.

Hint: Logical operations (and, or, etc.) and shift instructions are particularly useful for isolating and combining the bits needed to perform the translation.

```
# Translate a virtual address to a physical address
# Input:
#    $a0   address of first-level page table
#    $a1   virtual address to be translated
# Output:
#    $v0   physical address corresponding to virtual
#          address if translation succeeds or 0xFFFFFFFF
#          if a page fault occurs

vir2phys:
     srl  $t0,$a1,20
     andi $t0,$t0,0xFFC     # $t0 = vpn1*4
     or   $t0,$t0,$a0       # $t0 = addr 1st level pte
     lw   $t0,0($t0)        # $t0 = 1st level pte
     bge  $t0,$zero,fault   # page fault if sign bit = 0
     sll  $t0,$t0,12        # $t0 = addr 2nd level pg tbl
     srl  $t1,$a1,10
     andi $t1,$t1,0xFFC     # $t1 = vpn2*4
     or   $t1,$t1,$t0       # $t1 = addr 2nd level pte
     lw   $t1,0($t1)        # $t1 = 2nd level pte
     bge  $t1,$zero,fault   # page fault if sign bit = 0
     sll  $t1,$t1,12        # $t1 = ppn part of address
     andi $v0,$a1,0xFFF     # $v0 = offset bits
     or   $v0,$v0,$t1       # $v0 = physical address
     jr   $ra               # return

fault:                      # return -1 = 0xFFFFFFFF
     li   $v0,-1            # after page fault
     jr   $ra
```

**Note: The above code uses logical "or" operations to combine page frame addresses with offsets and indices. It should work just as well to use add instructions in this case, but since we really are concatenating bits to form addresses the logical operations might make the intent a little clearer.**

**Question 7.** (12 points) A few short-answer questions.

(a) Caches can use either a write-back or a write-through policy. What are these policies and how do they differ? Which one is most likely to improve cache performance and why?

**Write-back = writes go only to the cache, and new values are only written to main memory when the block is evicted from the cache.**

**Write-through = new values are immediately written directly to main memory.**

**Write-back is normally better for performance since multiple writes to a single cache block (which are likely because of both temporal and spatial locality) can be done using only the cache and do not require the time or bandwidth needed to write to main memory for every update.**

(b) Suppose we are designing a 32K cache with 64-byte blocks. We have a choice between a direct-mapped cache with 512 rows, or a 2-way set associative cache with 256 rows (sets), but two blocks per row. Which would be the better choice generally and why? (Give a brief technical justification for your answer).

**The 2-way associative cache is more likely to improve performance. With a direct-mapped cache, each main memory block can only be held in a single location in the cache. If two main memory blocks are in active use and, unfortunately, require the same cache location, they will force each other out of the cache repeatedly. With a 2-way associative cache both of the blocks can more likely be retained in the cache at the same time.**

**Question 7 (cont.)** (c) Many of the I/O devices connected to computers use a Direct Memory Access (DMA) interface. What does this mean? What is the major difference between how DMA and non-DMA devices perform I/O?

**Once an I/O operation is initiated with a DMA device, the device handles transfers of data between the device and memory by grabbing control of the bus when needed to transfer the next piece of data. Without DMA the processor needs to control the data transfer piece-by-piece, which decreases the amount of processor cycles available for other work.**

**Question 8.** (10 points) Finally, to finish up, here are some true/false questions. Circle the correct answer.

a) True / False. MIPS is a load-store architecture.

b) True / False. The decimal number 127 as a 32-bit two's complement hexadecimal number is 0xFFFFFF83.

c) True / False. Pipelining takes advantage of instruction level parallelism.

d) True / False. A CPU with a faster clock frequency always has higher performance than one with a slower clock.

e) True / False. The MIPS `move` instruction is a pseduo-instruction.

f) True / False. Temporal locality refers to the idea that data in memory which is close in address to memory data which has recently been accessed is likely to be accessed soon.

g) True / False. For a virtual memory system with 48-bit virtual addresses, a single page table and 4 KB physical page size, the width of the virtual page number portion of an address is 34 bits.

h) True / False. Page faults are handled by the operating system.

i) True / False. The opcode field of all J-format MIPS instructions is 0.

j) True / False. I enjoyed this class. (All honest answers receive credit for this part.)