

NAME _____
First Last

Student ID# _____

STONY BROOK UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
MIDTERM #2 EXAMINATION

CSE 220
Spring Semester 2008
April 10, 2008

This is a closed-book exam. (80 minutes)
Use this form for your work and return it.

The exam has 10 problems.

It is crucial to show all work done in order to obtain full/partial credit

# 1 _____ (10 pts)	# 6 _____ (8 pts)
# 2 _____ (5 pts)	# 7 _____ (4 pts)
# 3 _____ (3 pts)	# 8 _____ (4 pts)
# 4 _____ (5 pts)	# 9 _____ (5 pts)
# 5 _____ (5 pts)	#10 _____ (3 pts)

TOTAL _____ (50 max)

1

[10 pts] Multiple Choice. Identify and fill in the correct answer.

Each part has just one correct answer - you will get no credit if you choose more than one answer.

- (d) — Register \$at
- (a) always contains value 0
 - (b) points to the top of the stack
 - (c) is reserved for the O/S
 - (d) is reserved for the assembler
- (d) — The MIPS register or registers for returning function values are:
- (a) \$fp
 - (b) \$a0 - \$a4
 - (c) \$v0
 - (d) \$v0 & \$v1
 - (e) there are no such registers
- (e) — The R instruction format does not contain which field:
- (a) Target register
 - (b) Source register
 - (c) Function
 - (d) Opcode
 - (e) Immediate
- (c) — The jump instruction j
- (a) uses direct addressing
 - (b) uses the immediate (I) instruction format
 - (c) uses a pseudodirect addressing
 - (d) none of the above
- (c) — The jump-and-link instruction ('jal') has which instruction format:
- (a) Register (R)
 - (b) Immediate (I)
 - (c) Jump (J)
 - (d) none of the above
-

- (d) — Which instruction uses base addressing
- (a) bitwise or, `or`
 - (b) adding, `add`
 - (c) set less than, `slt`
 - (d) store word, `sw`
 - (e) jump register, `jr`
- (c) — Which instruction stores the offset as words and not as bytes
- (a) load word, `lw`
 - (b) set less than immediate, `slti`
 - (c) branch not equal, `bne`
 - (d) load upper immediate, `lui`
- (b) — Which of the following is not a MIPS addressing mode
- (a) Immediate Addressing
 - (b) Direct Addressing
 - (c) PC-Relative Addressing
 - (d) Register Addressing
 - (e) Pseudodirect Addressing
- (d) — Consider linking three modules with text segments T1, T2, and T3. The assembler start address is 0. The linker prepares an executable program with start address 0x00400000, where T1 is placed first, followed by T2 and then T3. The sizes of T1, T2 and T3 are 0x8000, 0x3000 and 0x1000 bytes respectively. What is the start address of text segment T3?
- (a) 0x00408000
 - (b) 0x00409000
 - (c) 0x0040A000
 - (d) 0x0040B000
 - (e) 0x0040C000
- (a) — Which statement is TRUE for Virtual memory
- (a) Allows execution of programs larger than physical memory
 - (b) All addresses generated by a program during execution are physical memory addresses
 - (c) Instruction cycle requires the same amount of time as running without virtual memory
 - (d) Every program is executed in its own virtual space
 - (e) None are true

I would also accept an answer of (d). Strictly speaking, a program is, of course, executed in *physical* memory space, which is shared with many other programs. However, to the extent that the executing program generates addresses from its own virtual space (which are then translated into physical memory addresses by the virtual memory mechanism), there is something of an ambiguity here which makes the choice of (d) plausible.

- 2 [5 pts] Suppose two 8-bit numbers $A = 1001\ 0101$ and $B = 0110\ 0110$ are in 8-bit registers, $\$r1$ and $\$r2$ respectively.
- (a) [1 pt] What are the contents of $\$r0$ after: `or $r0 $r1 $r2`? **1111 0111**
- (b) [1 pt] What are the contents of $\$r0$ after: `andi $r0 $r1 19`? **0001 0001**
- (c) [1 pt] What are the contents of $\$r0$ after: `sra $r0 $r1 3`? **1111 0010**
- (d) [2 pts] Consider the shift right arithmetic ('sra') instruction and its purpose, in contrast to the shift right logical ('srl') instruction. Briefly explain why a 'sla' ("shift left arithmetic") instruction is not needed.

Shifting right is equivalent to integer division by 2. Shift right arithmetic does sign extension so as to preserve the sign of the result when the original value is negative. Shifting left is equivalent to multiplication by 2. The sign of the result is automatically preserved when we do a shift left logical. Thus there is no nothing to distinguish a logical from an arithmetic shift in this case.

Note that if we consider the original unshifted value as a signed integer and the sign 'flips' from 0 to 1 (or from 1 to 0) in the result of a left shift, this indicates that an overflow (underflow) has occurred. Similarly, if we consider the original unshifted value as an unsigned integer, and the shifted result has all bits 0, that too is an indication of overflow.

- 3 (a) (1 pt) "`la $t0, A`", where A is a label in `.data`, is a pseudoinstruction. Briefly explain how you can tell that (*i.e.*, how you can tell that it is a pseudoinstruction and not an actual machine instruction).

Taken at face value, the instruction refers to a memory address (for label A), which is 32 bits. It would, furthermore, have to include a 6-bit opcode (for 'la') and a 5-bit register number (for '\$t0'), which makes a total of 43 bits. But an instruction in MIPS is only 32 bits wide. So "`la $t0, A`" has to be a pseudoinstruction.

- (b) (2 pts) Consider the pseudoinstruction “`lw $t0, A($t1)`”, where A is a label in `.data` with address `0x10010000`. The assembler will generate the following machine instruction code for it:

```

lui    $at, 4097
addu   $at, $at, $t1
lw     $t0, ($at)

```

What would be generated if the address of A were `0x10000023` instead of `0x10010000`? Your answer should consist of three instructions as above, appropriately modified. No new instructions are needed.

The top half (top 16 bits) of the 32-bit address for A was `0x1001 = 409710` and is now `0x1000 = 409610`.

The bottom half was `0x0000 = 010` and is now `0x0023 = 3510`. Thus, the assembler would generate:

```

lui    $at, 4096
addu   $at, $at, $t1
lw     $t0, 35($at)

```

Note that the assembler generates exactly the same three instructions in all cases, irrespective of the address for A , adjusting the numerical values in line with the actual value of the address. It does not analyze the value of the address and generate sequences of instructions with different operations for different values (as some of your answers seemed to imply).

- 4 (a) [2 pts] The global data area in MIPS memory starts from `0x10000000`. What is the size of global area that is accessed efficiently using `$gp`? Give your answer in bytes, expressed in decimal.

2^{16} bytes = 64 KB = (65,536)₁₀, in the range (`$gp - 215`) to (`$gp + (215 - 1)`).

- (b) [1 pt] Suppose we decide to have the global area starting from `0x10020000`. What would be the size of our new global area that we can access efficiently using `$gp`? Again, give your answer in bytes, expressed in decimal.

Same answer as for part (a) above (assuming `$gp` is appropriately initialized – see the answer to part (c) below).

The area that can be accessed depends on the range of values we can have in the *immediate* field of an I-type instruction, and not on where in memory that area is.

- (c) [2 pts] To access the new global area (which starts at 0x10020000), \$gp must be initialized to some value before execution begins. What is that value, expressed in hexadecimal?

We want \$gp to be initialized to the middle of the 2^{16} -byte area starting at address 0x10020000 that we can potentially access using *base-index* addressing, with base value \$gp and both positive and negative *immediate-value* displacements. Note that $2^{16} \div 2 = 2^{15} = 0x8000$.

Thus we want $\$gp = 0x100208000$.

- 5 [5 pts] The MIPS instruction set that we have studied can be classified into the five following groups: (i) Arithmetic (ii) Logical (iii) Conditional Branch (iv) Unconditional Branch/Jumps (v) Data Transfer .

- (a) [1 pt] Instructions from which of these groups are allowed to access memory during the execution phase of an instruction? List one or more groups.

(v) Data Transfer.

There are no other groups that access memory in the MIPS *load-store* architecture.

- (b) [1 pt] Name one group such that all the instructions in the group use the same instruction format (be it R or I or J).

**(iii) Conditional Branch (all instructions are I type), and
(v) Data Transfer (all instructions are I type).**

- (c) [1 pt] Name one group such that the instructions in the group use more than one format (R, I or J).

**(i) Arithmetic (an instruction is either R or I type), and
(ii) Logical (an instruction is either R or I type), and
(iv) Unconditional Branch/Jumps (an instruction is R, I or J type).**

- (d) [2 pts] During linking, text and data segments from different modules are placed in some order. Depending on the start address for text and data segments, some instructions need modification or “readjustment” and some do not. State which instruction groups may require modification, and which groups would never require any changes (no matter what the start address). Briefly explain your answer.

May Require modification: (iv) **Unconditional Branch/Jumps**
(v) **Data Transfer**

Some instructions in these groups use either pseudodirect addressing (in which case the 26-bit pseudodirect address field might require readjustment); or, in the case of *load* and *store* pseudoinstructions, are translated into instructions that use base-index addressing (in which case the base address and/or index/offset values might require readjustment).

Never require modification: (i) **Arithmetic**
(ii) **Logical**
(iii) **Conditional Branch**

These instructions use either register addressing with or without *immediate* values; or, in the case of conditional branching, PC-relative immediate offset values. No adjustment is needed in either case.

6

[8 pts] Consider the following MIPS program fragment which is assembled starting at address-byte 600. The address of *main* is byte 600 (or word 150) in decimal.

Do not worry about what the program does. Some pseudoinstructions will generate more than one MIPS instruction, as noted in the code comments below.

```

main:   li $t0, 10
        li $t3, 20
        la $t4, Input # la generates 2 instructions. Input is a string in .data
        move $t2, $t4
loop:   lb $t1, ($t4)
        addi $t1, 48
        bgt $t1, $zero, out # this form of bgt generates 2 instructions
        addi $t2, 20
        sb $t3, 1($t2)
        j loop             # <--- part (d)
out:    sub $t0, 1
        beqz $t0, loop     # <--- part (c)
        addi $t1, $t2, 1   # ignore rest of the code.

```

(a) [2 pts] After the program has been assembled, what would be the addresses (in bytes, expressed in decimal) of labels:

(i) *loop* = (620)₁₀ (ii) *out* = (648)₁₀

(b) [2 pts] For the above program, give all jump/branch instructions that have a backward reference.

– (i) **j loop** – (ii) **beqz \$t0, loop**

(c) [2 pts] What is the offset value in the **beqz** instruction after assembly, in decimal? (Use actual MIPS implementation and not SPIM.)

-9 words/instructions. The offset value is stored in words (not bytes) in the *immediate* field of the instruction, relative to the instruction which follows the ‘beqz’.

(d) [2 pts] What is the value in the 26-bit field of the ‘j loop’ instruction, in decimal?

(155)₁₀ (= byte address 620). The value is stored in words (not bytes) in the 26-bit *address/target* field of the ‘j’ instruction.

7

[4 pts] Consider the following program that obtains a ‘3-4-swap’ of a string. A 3-4-swap is possible when the length of a string is exactly divisible by 4; this length does not include the terminating null character, `\0`. The 3-4-swap program takes a group of 4 bytes at a time, and swaps the third and the fourth byte/character. For string *ABCD1234*, for example, its 3-4-swap would be *ABDC1243*.

```

        .data
m1:     .ascii "Execution-begins-here\n"
        .align 2                # align to address divisible by 4.
Str:    .ascii "WE-LOVE-CSE220-!" # string for a 3-4 swap.

        .text
        .globl main

main:   li $v0, 4                # print message m1.
        la $a0, m1
        syscall

        la $t0, Str            # pointer to Str in $t0.
chk:    beqz $t0, done          # check end of Str.

--->

--->
        addi $t0, 4             # increment pointer by 4.
        j chk

done:   li $v0, 4                # print Str.
        la $a0, Str
        syscall
        ...                     # code for exit, omitted.

```

(a) [1 pt] Is `.align` really necessary? Give a straight ‘Yes’ or ‘No’ answer.

No.

(b) [1 pt] Briefly explain your answer to part (a) above.

‘.align 2’ means align on a 2^2 byte boundary; *i.e.*, on a word boundary. But none of the data that follows the ‘.align’ requires alignment on word boundaries. There is, in fact, just one data item that follows the ‘.align’, and that is the null-terminated ASCII string *Str*. ASCII strings can start at any byte address in memory.

(c) [2 pts] Fill in code (where the arrows are in the code above) that swaps the third and fourth characters of a group of 4 bytes. You may use additional registers, but your code you is not allowed to change the pointer in \$t0.

```

lb $t1, 2($t0)    # load third byte into $t1
lb $t2, 3($t0)    # load fourth byte into $t2
sb $t2, 2($t0)    # store fourth byte as the third character of the substring
sb $t1, 3($t0)    # store third byte as the fourth character of the substring

```

8 [4 pts] Suppose we want to add a new instruction to MIPS. This is a pseudoinstruction called double (*dbl* for short). Its syntax is:

```
dbl $t1, $t0
```

Here the value in \$t0 is multiplied by 2 and stored in \$t1, ignoring any overflow. An easy way to translate *dbl* is to use the following sequence of MIPS instructions:

```

li $at, 2        # $at stores 2.
mult $t0, $at    # multiply $t0 by $at
mflo $t1         # copy product to $t1

```

(a) [2 pts] This is really not efficient. Briefly explain why.

- **The translation uses three instructions: we can surely make do with less.**
- **One of these three instructions is, furthermore, a multiply, which is an expensive operation.**

(b) [2 pts] Give two additional ways of translating pseudoinstruction double which are much better than what is given above. Give actual MIPS instructions - each additional way you give should consist of just one instruction.

- (i) `sll $t1, $t0, 1`
- (ii) `addu $t1, $t0, $t0` # ‘addu’ rather than ‘add’ because we ignore overflow
(‘add’ generates an exception on overflow; ‘addu’ does not)

9

[5 pts] The following MIPS program consists of *main*, a function *X*, and a function *Y*. Both functions *X* and *Y* have one parameter each. Main calls function *X*, and *X* calls function *Y*. Function *Y* does not call any other function.

Do not worry about what the program does. Most instructions have been omitted (indicated by ‘...’). Only relevant instructions that deal with function calls are given. Focus on the call to *X* (*i.e.*, before the call to, and after the return from, *X*), the entry into and exit from *X*, and the call to *Y* (*i.e.*, before the call to, and after the return from, *Y*).

```
.text
.globl main
main:  ...
      ...
      ...
      li $t1, -1
      move $a0, $t3      # parameter in $a0.
      jal X             # call X
      beq $t1, $s1, equal
      ...
      ...
equal: ...
      ...
done:  ...             # code for exit.
      ...

X:     addi $sp, $sp, -8 # function X.
      sw $s0, 4($sp)
      sw $s1, ($sp)

      li $s0, 50
      li $s1, 10
      li $s2, 5
      ...
      ...
      li $a0, -1       # parameter in $a0.
      jal Y           # call Y
      add $s2, $s1, $a0
chk:   beqz $s2, ret
      ...
      ...
ret:   lw $s0, 4($sp)  # prepare for return
      lw $s1, ($sp)
      addi $sp, $sp, 8
      jr $ra          ***** CODE CONTINUES ON THE NEXT PAGE *****
```

```
Y:      ...           # function Y.  
        ...  
        ...
```

- (a) [1 pt] Which of X or Y is a leaf function? **Y is a leaf function; X is not.**
- (b) [3 pts] This program violates several MIPS conventions regarding what should be done when:
(i) calling a function; and (ii) implementing the code for a function. List and briefly explain each such violation.
Note that a violation of MIPS conventions is not always necessarily an error: a program may still work even if some conventions are violated.
- $main$ does not save $\$t1$ on the stack before calling X , although it uses $\$t1$ after the call to X returns.**
 - X does not save the parameter register $\$a0$ on the stack before calling Y , although it uses $\$a0$ after the call to Y returns.**
 - X uses $\$s2$ but does not first save it on the stack then restore it just before returning, as it does with $\$s0$ & $\$s1$ which it also uses.**
- (c) [1 pt] In addition to violations of some conventions, this program has a serious error which will cause it not to work correctly (this error is in itself also a violation of MIPS conventions). State what the error is and briefly explain it.

X does not save $\$ra$, its return address back to $main$, on the stack before issuing the ‘jal Y ’ call to Y . This ‘jal’ will cause $\$ra$ to be over-written with the return address for Y back to X (*i.e.*, with the address of the instruction that follows the ‘jal Y ’). Thus, when X comes to return to $main$, it will go instead to this return address for Y .

- 10 [3 pts] Consider the following `.data` segment. It has exactly one variable declaration that contains an error.

```
.data
X:    .word 300
Y:    .byte 300
Z:    .space 300
Str:  .asciiz "I-LIKE-PIZZA"
End:  .ascii "Hi"
```

- (a) [2 pts] Name the variable, and explain the error.

***Y.* Label *Y* declares a byte and initializes it to the (decimal) value 300. But the largest value that can be stored in 8 bits is 255.**

- (b) [1 pt] Now assume that the error has been fixed, and all variables are retained in the same order. What would the size of data segment be? Give your answer in bytes, in decimal.

(320)₁₀.

- *X* declares a word (4 bytes) and initializes it to value (300)₁₀
- *Y* declares a byte and initializes it to some suitably corrected value
- *Z* reserves a space of (300)₁₀ bytes
- *Str* declares a null-terminated string of total length 13 bytes (including the ‘\0’ terminator)
- *End* declares a string with no null terminator of total length 2 bytes

The total is thus (320)₁₀ bytes.