

NAME _____
First Last

Student ID# _____

UNIVERSITY AT STONY BROOK
COMPUTER SCIENCE DEPARTMENT
MIDTERM #2 EXAMINATION

CSE 220
Spring Semester 2009
April 2, 2009

This is a closed-book exam (80 minutes). It has 9 problems.

You are NOT allowed to use calculators

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

TOTAL _____ (50 max)

1 (5 points) Mark Yes or No.

- (a) For the MIPS computer, when an executable file is created by the linker, relocation information is retained. **N**
- (b) (i) Logical right shift by 1 bit, and (ii) divide by 2, are essentially equivalent assuming positive integers and integer division. **Y**
- (c) Pseudo-instructions are expanded in the second pass of the assembler. **N**
- (d) When virtual memory is being used, not all addresses generated by a program during its execution are virtual addresses. **N**
- (e) The instruction `div $t4, 20, 5` has a syntax error. **Y**

2 (5 points) Suppose a 16-bit binary number 1100 0000 0001 1111 is in a 16-bit register.

- (a) (1 point) Its contents after 2 arithmetic right shifts are: **1111 0000 0000 0111**
- (b) (1 point) How many rotate right shifts are required to make the number positive? **6**
- (c) (3 points) MIPS has a pseudo-instruction `neg`. It obtains the 2's complement of an integer. For example, `neg $t0, $t1` calculates the 2's complement of `$t1` and stores it in `$t0`. How would you expand the pseudo-instruction `neg $t0, $t1` (*i.e.*, write out its equivalent in MIPS instructions)? Remember that in the expansion, you should use the instruction `nor` to obtain the 1's complement first.

We need to invert the bits of `$t1` to obtain its 1's complement. (Instruction `not` could do this, but it is a pseudo-instruction: it is implemented using instruction `nor` in which we 'nor' a register with itself.)

```
nor $t0, $t1, $t1      # obtain 1's complement in $t0
addi $t0, $t0, 1      # add 1, to get 2's complement
```

An easier way, that does not make use of the instruction `nor`, is:

```
sub $t0, $zero, $t1   # subtract $t1 from $zero
```

3 (3 points)

- (a) (1 point) What are the contents of \$gp (the global pointer register)? Give your answer in hexadecimal. If you cannot remember or work out the value, at least give a **careful** and **precise** description of what that value is.

The contents of \$gp are 0x10008000.

This is the midway point of the MIPS static (or ‘global’) data area which has size 0x10000 bytes (64KBytes), starting at byte address 0x10000000.

- (b) (2 points) What is the main advantage of having variables stored in the global area? Explain using an example such as `lw $t0, X`, where X is a variable (*i.e.*, a label in `.data`).

The main advantage is that the assembler would need to generate only one instruction, using base/displacement addressing centered on \$gp, to load (or store) X: *e.g.*, ‘`lw $t0, Δ($gp)`’, where Δ is some immediate (constant) value that gives the offset of X from the middle of the static data area. If X were not in the static data area (remember that variables declared in `.data` go into the static data area), but rather somewhere else (such as the dynamic data area, which starts at 0x1001000), then we would need two instructions for a load or a store.

4 (6 points)

- (a) (1 point) Name a MIPS instruction format that has fields (*i.e.*, space/room) for storing only two registers.

I format

- (b) (2 points) Name two MIPS instructions (not pseudo-instructions) that store offsets as bytes and not as words.

Any two of : *lb, lbu, lh, lhu, lw, sb, sh, sw.*

- (c) (1 point) What is the main advantage of the immediate addressing mode used in instructions such as *load immediate (li)*?

It saves a memory access, since the value to be loaded is fetched as part of the instruction itself.

(d) (2 points) The instruction `jr` (*jump register*) does not use the J format. Briefly explain why.

The instruction ‘jump register’ is used as in `jr $ra`; *i.e.*, it targets a jump address that is in a register.

The jump instruction format J is used when we need to store the target address directly in the instruction itself, in the form of a 26-bit pseudo-direct address. This is not the case for the instruction `jr`: we just need space for the opcode (`jr`) and for one register; but the J instruction format does not have a field for a register. The `jr` instruction actually uses the R instruction format.

5 (8 points) Consider the following MIPS program fragment which is assembled starting at byte-location 1000_{10} ; *i.e.*, the address of label `main` is byte 1000 (or, equivalently, word 250) in decimal. Do not worry about what this program does.

```

main:  li $t0, 4000
      li $t4, 0
top:   lb $t1, ($t0)
      beqz $t1, exit      # <----- part(b)
      addi $t0, 1
      j bot               # <----- part (c)
exit:  li $t2, 3000
      sub $t0, 1
      beq $t1, $t0, top   # <----- part(b)
bot:   lb $t3, ($t0)
      sb $t3, ($t2)
      addi $t2, $t2, 1    # Ignore rest of the code.
```

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

`bot` : 1036 and `exit` : 1024

Start assigning addresses to the instructions. An instruction is a 4-byte word. There are no pseudo-instructions in the code, so each assembly language instruction will generate one MIPS machine code instruction. The address of `main` is byte 1000, and that of `top` byte 1008 . . . , *etc.*

(b) (1 point) Usually, offsets in branch instructions are calculated in the second pass. But sometimes an offset can be calculated during the first pass as well. In the program above, for which conditional branch instruction could the offset be calculated in the first pass?

```

beq $t1, $t0, top   # top comes before this instruction, so the assembler
                   # will already know where it is when it reaches this
                   # instruction during the first assembly pass
```

- (c) (2 points) What is the content of the address field in the instruction **j bot**, in decimal?

The content of address field for *j bot* has to be stored as a word address in which we lop off the rightmost 2 bits. We also lop off the leftmost 4 bits, so as to make it into a 26-bit pseudo-direct address.

The address of label *bot* is: $1036_{10} = 10000001100_2$. Viewed as a 32-bit quantity, this has 21 leading zeros. Lopping off the bits to get the 26-bit pseudo-direct address gives: $100000011_2 = 259_{10}$.

Note that in this particular case, this is equivalent to: $1036_{10} \div 4 = 259_{10}$. But simply dividing the byte address in decimal by 4 will not work if its value is not sufficiently small that its 32-bit binary equivalent does not have at least four leading zeros.

- (d) (3 points) MIPS provides several conditional branch instructions. But many are actually pseudo-instructions and are assembled using **slt** (*set on less than*). Its syntax is as follows:

```
slt $Rd, $Rs1, $Rs2    # Set destination register Rd to 1 if $Rs1 is less than $Rs2;
                          # otherwise set it is to 0.
```

How would the SPIM assembler translate the **blt** (*branch on less than*) pseudo-instruction using **slt** and **bne**? Write the exact sequence of MIPS instructions that translate:

blt \$t0, \$t1, less.

Note that **bne** is not a pseudo-instruction. Your answer must use \$t0, \$t1 and the label *less*, and any other registers that may be required.

```
slt $1, $t0, $t1    # set register $1 to 1, if $t0 < $t1
bne $1, $zero, less # if $t0 < $t1, then $1 will have been set
                   # to 1, so it will not be equal to zero
```

- 6 (4 points) Consider data segments D1 and D2. Each is assembled assuming start address at byte 0. The size of D1 is 0x40 bytes (note that this is in hex). During linking, D1 is placed at 0x10001000. D2 is placed immediately after D1.

What is the size of data segment D2? What are addresses of *x*, *y* and *z* after linking? Fill your answers in below in hexadecimal.

First assign addresses to *x*, *y* and *z*. Then calculate the size of D2.

D1 is placed at 0x10010000, and its size is 0x40 bytes. D2 starts after D1, therefore D2 starts at 0x10010040. This is also the address of *x*.

The address of *y* is 0x10010048 (8 bytes after *x*).

The address of *str* is 0x10010049. Now allocate space for 5 bytes for *str*: the last

byte of *str* is therefore at 0x1001004D.

z must start on the next word boundary, which is 0x10010050 after skipping the two bytes 0x1001004E and 0x1001004F.

To obtain the size of D2, add all sizes: $(8 + 1 + 5 + 2 + 4) = 20$ bytes in decimal (we skipped two bytes in order to get *z* on the next word boundary), or 0x14 in hex.

	.data	# D2	Size of D2 = 0x14
x:	.double 1.0		Address of x = 0x10010040
y:	.byte 10		Address of y = 0x10010048
str:	.space 5		
z:	.word 100		Address of z = 0x10010050

7 (8 points) Complete the following program that swaps the last and the second from last (*i.e.*, the neighbour of last) elements of array *A*. You need to add code at the two arrows in the program that starts below and goes on to the following page:

(a) (4 points) Add code to obtain the address of the last element of array *A* in \$t2. You must use the **mult** (multiplication) instruction. Remember that **mult** is not a pseudo-instruction. Its syntax is:

```
mult $Rs1, $Rs2    # product is stored in the registers hi & lo.
```

You may use additional registers. **Put your code at the place indicated below.**

(b) (4 points) Insert code for the swap. For full credit, use as few instructions as possible. **Put your code at the place indicated below.**

```
.data
A:    .space 100    # an integer array of size 25
size: .word 25     # size of A

.text
.globl main
main: . . . . .    # code for initializing array A to some values; it has been omitted

    la $t1, A      # address of A (first element) in $t1
    lw $t3, size   # load size in $t3
    li $t2, 0      # initialize t2 to 0
```

Code continues on next page

Continuation of code from preceding page

-----> [part(a)] # Fill in code to obtain address into \$t2 of last element, using mult.

The address of the last element of the array is the base address of A (address of the first element of A) plus $4*(size - 1)$, where we multiply by 4 to get the byte quantity.

```
li    $t4, 4           # load 4 in $t4
addi  $t3, $t3, -1     # size = size -1
mult  $t3, $t4         # product = 4*(size-1) is in register lo
mflo  $t2              # product is moved to $t2 from register lo
add   $t2, $t2, $t1    # add (base) address of A to $t2
```

-----> [part(b)] # Fill in code for swapping last element and its neighbour.

```
lw  $s1, ($t2)        # load last element of A in $s1
lw  $s2, -4($t2)      # load its neighbour in $s2
sw  $s1, -4($t2)      # store $s1 and $s2 in reverse order
sw  $s2, ($t2)        # swap is complete
```

To get full credit, you must use offset -4 in the lw and sw instructions. If you subtract 4 and later add 4 instead, your code will have two or more extra instructions -- deduction 1 point.

8 (3 points)

- (a) (1 point) Conditional branch instructions use the I instruction format in which the 16-bit immediate field is used to store the offset. State what this offset is calculated relative to.

The offset is relative to the program counter value (PC) for the branch instruction in SPIM; more precisely, it is relative $PC+4$ in the real MIPS architecture.

- (b) (2 points) Suppose we store this offset as bytes, and not as words. What would be the range (*i.e.*, how far up or down) we could branch from the given conditional branch instruction? Give your answer in terms of words/instructions.

If we store this offset as bytes, the range would be reduced by a factor of 4. The width of the offset field in the I instruction format is 16 bits, and the value stored there is in 2's complement representation. The leftmost bit is for the sign. So we have 15 bits for the 'magnitude'. When we store this as words, our range is -2^{15} to $+(2^{15} - 1)$. But if we store this offset as bytes, the range would be -2^{13} to $+(2^{13} - 1) = -8192_{10}$ to $+8191_{10}$ words/instructions.

9 (8 points) Complete the following MIPS program, using MIPS conventions for: function parameters, returning values, making use of the stack, and saving registers. The program counts lower case letters in a given string. The actual check is done by a function *lower* which takes one parameter (a character). *lower* returns 1 if the character given as parameter is a lower case letter, otherwise it returns 0. The function has been written by somebody else, and we do not know which registers it makes use of. So it is possible, for example, that *all* of the ‘temporary’ registers (t0 – t9) are used in *lower*. Note that *main* makes use of some t registers after the call to *lower*. Some of these will need to be saved before the call to *lower*, and then restored after we return from the call.

(a) (6 points) Write code at the two places indicated by arrow (---->). For full credit, save and restore **only** those registers that need saving for the correct functioning of the program.

```

        .text
        .globl main

main:   move $s1, $zero           # initialize s1, contains count of lower
        la $t0, str              # address of str in t0
chk:    lb $t1, ($t0)            # character in $t1
        beqz $t1, done           # check for end of string
---->
        # prepare for function call

```

Here you are supposed to save and restore register \$t0 only, because the value that was in it before the call to *lower* is used after the call returns. By MIPS convention, \$t0 is to be saved by making use of the stack.

If you saved \$t1 (not required for the logic), and \$s1 or \$a0 (not the responsibility of the calling function) then you may lose 1 point for each. Similarly, there is no need to save \$ra.

Worth 4 points:

```

        move $a0, $t1           # parameter in $a0 ----- 2 points
        sub $sp, $sp, 4         # allocate space for $t1 ----- 2 points
        sw $t0, ($sp)          # save $t0 --- (part of the 2 points immediately above)

        jal lower              # function call to lower
---->
        # after function call

```

Worth 2 points:

```

        lw $t0, ($sp)          # restore $t0.
        addi $sp, $sp, 4        # deallocate stack space

```

Code continues on next page

Continuation of code from preceding page

```
        add $s1, $s1, $v0      # v0 returns 1 for a lower case letter
        addi $t0, $t0, 2
        j chk
done:    . . . .              # print count from $s1
        . . . .              # exit from SPIM

        .data
str:    .asciiz "I love CSE 220"
```

- (b) (2 points) There is a small error in the code that has been given. What is the error and how would you fix it?

The error is in the instruction ‘addi \$t0, \$t0, 2’: we should increment the address in \$t0 by 1 and not 2 . Change it to: ‘addi \$t0, \$t0, 1’.