CS 231 Review for Test #1

1. Suppose you are writing a function in MIPS assembly that takes no parameters, and does not call any other function, but it does use registers $s6 and $s7 to perform its calculations. It is possible that the calling environment may also be using these same registers. Therefore, what instructions must be included at the beginning and end of your function to save and restore the calling environment's values of these registers?

2. For the following instruction, explain why the machine code requires more than one instruction. In other words, it's a pseudo-instruction. How could we rewrite it using only "true" 32-bit instructions? Hint: $65537 = 2^{16} + 1$.

```
li $s1, 65537
```

3. In MIPS assembly, how would you test to see if registers $s1, $s2 and $s3 are all equal?

4. Convert the following C program fragment into MIPS. State any assumptions you need to make regarding register allocation, etc. Assume that i, x and y have been declared as integers, that M is an array of integers, and that n is an integer containing the number of elements in the array M. M is indexed from 0. Upon completion, your code should store the computed value of y into a memory location labelled y.

```
x = 3;
i = n - 1;
y = M[i];
while (i > 0)
{
   i = i - 1;
   y = M[i] + x*y;
}
```

5. Consider the integer array declared as a[8][10] that is stored in row-major order. Compile this statement: a[i][j] = x;

6. How would the number 1024 appear as a word in memory on a big-endian machine? Assume the first byte is address 0x100.

7. Implement this pseudocode in MIPS: "if $t1 is between 70 and 79 inclusive then increment $s3 by one."

8. What is the effect of the instruction whose encoding is 0x1640ffee? Refer to the encodings given in the "TAL instruction set" handout.

9. What code do we write in a non-leaf procedure to preserve its return address?

10. Show how the following instruction is expressed in "true" assembly language using the slt instruction plus a branch instruction: `ble $s4, $s5, loop`.

11. Suppose that a function begins with instructions that save the return address register and the five registers $s0 through $s4 on the system stack. Assume that this function has one parameter, and it was saved to the system stack immediately before this function was called. If the $sp register currently points to the next available place on the system stack (where more data could be pushed), write the instruction that will copy the value of the function parameter into register $s0.

12. Suppose that in a particular computer program, 10% of the executing instructions are jumps. Jump instructions take 2 cycles each, and the other instructions take 5 cycles each.
    a. Calculate the CPI for this program.

    b. Suppose we could improve the compiler so that it could remove **half** of these jump instructions, but this will cause a slight increase in the number of non-jump instructions. What is the maximum percent increase in the number of non-jump instructions you would accept to ensure that the new compiler actually allows the program to take less time than the original version?

13. Suppose a is a 2-dimensional array of integers stored column major. The elements are numbered a[i][j], where i is the row number and j is the column number, and both indices start at 0. If the base address is 500 and the address of element a[2][1] is 600, then what are the dimensions of the array? If either or both of the dimensions cannot be determined, then briefly explain why.

14. Consider the following code to copy a string into a new location.

```
        la $s1, old_string
        la $s2, new_string

        move $t1, $s1
        move $t2, $s2
loop:
        lb $t3, 0($t1)
        sb $t3, 0($t2)
        addi $t1, $t1, 1
        addi $t2, $t2, 1
        bnez $t3, loop
```

    a. Give the hexadecimal encoding for the bnez instruction. Use the attached TAL instruction set. (Assume that $t3 is register number 11.)

b. Show what changes are necessary so that the code will correctly copy only the non-digits into the new string. For example if old_string points to “abc123xyz4”, then new_string should become “abcxyz”.

15. Suppose an array in C is declared as int a[8][9][10]; Assume that the array is stored in row-major order and each integer takes up 4 bytes. If &a[0][0][0] = 0x700, then which element is at address 0x1000?

16. Consider the instruction sw $s1, 0x45678($s2).
    a. Write a high-level language statement that could be compiled to generate the above instruction. (Hint: 0x45678 equals 284,280 in base 10.)

    b. Actually, the above instruction is not a “true” 32-bit instruction since it contains more than 32 bits of information. The instruction could be converted into the following three “true” instructions, but there are some mistakes. Identify what the mistakes are, and show the correct code:

```
ori $s2, $s2,  8
lui $s2, 0x4567
sw $s1, 0($s2)
```

17. Rewrite this loop so that the loop body has one fewer instruction in it (but do not change the effect of the loop).

```
        li $s0, 0
loop:
        bgt $s1, 100, endloop
        add $s0, $s0, $s1
        addi $s1, $s1, 1
        j loop
endloop:
```

18. Consider the following program.

```
        .data
blank:      .asciiz " "
newline:    .asciiz "\n"
        .text
main:
        li $s0, 1
outer:
        move $s1, $s0
inner:
        li $v0, 1                   # print the integer in $s1
        move $a0, $s1
        syscall

        li $v0, 4                   # print a blank
        la $a0, blank
        syscall

        addi $s1, $s1, _____
        _____ $s1, _____, inner

        li $v0, 4                   # print newline
        la $a0, newline
        syscall

        addi $s0, $s0, _____
        _____ $s0, _____, outer

        li $v0, 10                  # end of program
        syscall
```

The above program contains some instructions that are incomplete. Fill in the blanks with the appropriate mnemonic or operand so that the overall effect of the program is to print the following numbers when it executes:

1
3 1
5 3 1
7 5 3 1
9 7 5 3 1

19. Suppose that an assembly program has a main function, and a recursive function A. Each time we call A, we need to pass two integer parameters onto the system stack, and function A itself needs to use register $s0, so A needs to save this to the stack as well. Assume that when we run the program, the input is such that A will call itself exactly twice. Show what the system stack looks like (including the position of the stack pointer), at the point just before A returns from its innermost call.

20. Consider two machines, A and B, where machine A costs $10,000 and machine B costs $5000. Machine A can run program 1 in 5 seconds and program 2 in 4 seconds. Machine B can run program 1 in 7 seconds and program 2 in 10 seconds. It turns out that program 1 is an essential operation that must run 200 times per hour. Any remaining time on a computer can be spent running program 2, so that we can define a machine's throughput as how many times per hour it can perform program 2.
    a. Each hour, how much time can machine A devote to program 2?

    b. How many times per hour can machine A run program 2?

    c. Each hour, how much time can machine B devote to program 2?

    d. How many times per hour can machine B run program 2?

    e. Let us define "cost effectiveness" as the amount of work that can be done per hour divided by the cost of a machine. Its unit is jobs per hour per dollar. What is the cost effectiveness of each machine?

    f. Finally, we can conclude, that machine _____ is more cost effective than machine _____ by a factor of _____.

Answers to Review #1

1. At the beginning of the function:

```
        sw $s6, 0($sp)
        sw $s7, -4($sp)
        addi $sp, $sp, -8
```

   At the end of the function:

```
        addi $sp, $sp, 8
        lw $s6, 0($sp)
        lw $s7, -4($sp)
```

2. The constant cannot fit into 16 bits. We use lui for the upper 16 bits and ori for the lower 16 bits.

```
        lui $s1, 1
        ori $s1, $s1, 1
```

3. Use two consecutive branch instructions. First, compare $s1 and $s2. If they are equal, then proceed (fall through) to the second branch instruction. Therefore, we need to write bne instructions.

```
        bne $s1, $s2, not_equal
        bne $s2, $s3, not_equal
        # the code to perform if they are all equal
        j end_if
not_equal:
        # the code to perform if they are not all equal
end_if:
```

4. Let's allocate the registers as follows:
   $s0 = x
   $s1 = y
   $s2 = i
   $s3 = n
   $s4 = base address of M
   $s5 = address of a cell from M we wish to access
   $s6 = value from a cell from M
   $s7 = address of y
   $t0 = a temporary value, intermediate arithmetical result

```
        li $s0, 3
        addi $s2, $s3, -1
        la $s4, M
        mul $s5, $s2, 4
        add $s5, $s4, $s5
        lw $s1, 0($s5)

loop:
        ble $s2, $zero, end_loop
        addi $s2, $s2, -1
        mul $s5, $s2, 4
        add $s5, $s4, $s5
        lw $s6, 0($s5)
        mul $t0, $s0, $s1
        add $s1, $s6, $t0
        j loop

end_loop:
        la $s7, y
        sw $s1, 0($s7)
```

5. The array has 8 rows and 10 columns. To access an element in row i, column j in row-major order, we need to determine how many cells we are from the beginning of the array. Because we are in row i, there are i rows above the cell. How many cells are there per row? The number of columns in the array (10). On the same row, there are j cells to the left. Therefore, the cell at row i, column j is located 10i + j cells from the beginning of the array. Since integers occupy 4 bytes each, the byte offset is 4(10i + j) from the base address.

```
        # Let $s0 = base address of a
        # $s1 = i
        # $s2 = j
        # $s3 = address of a[i,j]
        # $s4 = address of x
        # $s5 = value of x

        la $s0, a
        mul $s3, $s1, 10        # 10i
        add $s3, $s3, $s2       # 10i + j
        mul $s3, $s3, 4         # 4(10i + j)
        add $s3, $s0, $s3       # total address
```

```
        la $s4, x
        lw $s5, 0($s4)
        sw $s5, 0($s3)
```

6. The number 1024 = 2^{10}, so its binary representation is 100 0000 0000. In hexadecimal, this becomes 0x00000400. The bytes of this number are: 00 00 04 00 (in hexadecimal). On a big endian machine, the first (lowest address) byte of a word contains the big end of the number, the byte of the number containing the largest powers of 2. Therefore:
   Byte 0x100 = 00
   Byte 0x101 = 00
   Byte 0x102 = 04
   Byte 0x103 = 00

7. We should use two consecutive branch instructions to avoid the incrementing instruction. We need to branch if $t1 is less than 70 or greater than 79.

```
        blt $t1, 70, end_if
        bgt $t1, 79, end_if
        addi $s3, $s3, 1
end_if:
```

8. Let's convert from hex to binary:
   0x1640ffee = 0001 0110 0100 0000 1111 1111 1110 1110

   Scan the instruction encodings to see what type of instruction is compatible with these bits. The fact that our first four bits are 0001 narrows the possibilities to blez, bgtz, beq and bne. If you look at the 5th and 6th bits (01), you see that we have a bne instruction. Its format is:
   `000101 sssss ttttt iiiiiiiiiiiiiiii`
   If we regroup the 32 bits we are given in this fashion, we obtain:
   `000101 10010 00000 1111111111101110`
   Therefore, the s register is register number: 10010 (equals 18)
   And the t register is register number: 00000 (equals 0)
   The immediate value is 1111 1111 1110 1110. This is a negative number. Let's find the opposite by inverting all the bits until the last 1: 0000 0000 0001 0010. This is the binary representation of 16 + 2 = 18. Therefore, the immediate value is –18. We are told that the format of the bne instruction is "bne Rs, Rt, I". This means the instruction is:
   bne $18, $0, –18

The meaning of the "–18" is that we have to adjust the program counter 18 instructions backwards from where it would have been by default. By default, we would have executed the next instruction. So, the effect of the instruction is to say:
"If register 18 is not equal to zero, then go to the instruction located 17 instructions before this branch instruction."

9. We need to push the $ra register on the stack at the beginning of the function, and then pop it at the end.

```
sw $ra, 0($sp)
addi $sp, $sp, -4

# body of function

addi $sp, $sp, 4
lw $ra, 0($sp)
```

10. The instruction slt means "set if less than." "le" means "not gt". And "gt" is the same as "lt" except for the order of the operands.

```
slt $t0, $s5, $s4
beqz $t0, loop
```

11. Here is what memory looks like in the vicinity of the run-time stack:

	Rest of RAM with lower addresses
$sp →	
	Value of $s4
	Value of $s3
	Value of $s2
	Value of $s1
	Value of $s0
	Value of $ra
	The parameter passed to the function
	Rest of RAM with higher addresses

We notice that the parameter is located 7 words deeper in the stack from where the stack pointer is pointing.

```
lw $s0, 28($sp)
```

12. A question of computer performance:
    a. The CPI is determined by a weighted average.
       10% (2) + 90% (5) = 4.7 cycles per instruction

    b. Let n = number of instructions in the original program.
       Let x = proportional increase in non-jump instructions (i.e. 0.1 would be 10%)

       The original execution time = CPI * (number of instructions) = 4.7n cycles.
       Note that the clock rate makes no difference in our comparison.

       The "new" number of cycles = # cycles from jump + # cycles from non-jump
       = CPI_j * # jump + CPI_nj * # non-jump
       = 2 * 0.05n + 5 * 0.9n * (1 + x)
       = 0.1n + 4.5n + 4.5nx
       = (4.6 + 4.5x) n cycles
       We want 4.6 + 4.5x < 4.7, which works out to 4.5x < 0.1, or x < 1/45 or x < 2.22%.

13. When accessing an array element:
    total_address = base_address + offset
    = base_address + (size of integer) * (column # * size of column + row #)
    Plugging in the numbers we are given:
    600 = 500 + 4 (1 * numRows + 2)
    100 = 4 (numRows + 2)
    So, numRows = 23.

    We don't know the number of columns because the address of a[i, j] only depends on the columns to the left, not the right. There could be any number of columns, but certainly at least 2 because our element resides in column 2.

14. This question has two parts.
    a. The bnez instruction is equivalent to: bne $t3, $0, loop

       The format is
       `000101 sssss ttttt iiiiiiiiiiiiiiii`

       We need to figure out the numbers s, t and i, and write them in binary.
       The number s refers to register number 11, so sssss is the binary for 11 = 01011.
       The number t refers to register number 0, so ttttt is the binary for 0 = 00000.

The 16-bit number i is the instruction offset. If we take the branch, how many instructions away is the destination compared to the default fall-through instruction? It is 5 instructions back, so i = –5. The number 5 in 16-bit binary is 0000 0000 0000 0101. To negate this number, we invert all the bits from left to right until the last one: 1111 1111 1111 1011.

Therefore the binary encoding of the instruction is:
000101 01011 00000 1111 1111 1111 1011
And if we group in fours to facilitate converting to hexadecimal, we obtain:
0001 0101 0110 0000 1111 1111 1111 1011
The hexadecimal equivalent is: 0x1560fffb.

There is an alternative correct answer. You could have swapped the order of the $t3 and $0 in the bne instruction: bne $0, $t3, loop. In this case, the binary encoding would have been:
000101 00000 01011 1111 1111 1111 1011
And if we group in fours to facilitate converting to hexadecimal:
0001 0100 0000 1011 1111 1111 1111 1011
The hexadecimal equivalent is 0x140dfffb.

b. When loading the character into $t3, if it is less than '0' or greater than '9', then go ahead and copy it into the new_string. Otherwise skip this character. The code before the loop is unchanged. The loop becomes:

```
loop:
        lb $t3, 0($t1)
        blt $t3, '0', copy
        bgt $t3, '9', copy
        j next
copy:
        sb $t3, 0($t2)
        addi $t2, $t2, 1    # only increment $t2 if copying
next:
        addi $t1, $t1, 1
        bnez $t3, loop
```

15. First, be careful with the arithmetic: 0x1000 – 0x700 is 0x900, not 0x300. This is because 0x1000 = 16 * (0x100) and 0x700 is 7 * (0x100).

    0x900 bytes = 9 * 2^8 bytes = 9 * 64 words = 576 words.
    Each page has 90 integers.

576 / 90 = 6, and this gives us our page number.
576 mod 90 = 36, and this tells us where on the page we are, but this needs to be interpreted as a row and column.
Each row has 10 integers because there are 10 columns.
36 / 10 = 3, and this gives us our row number.
36 mod 10 = 6, and this gives us the column number.

The array reference is a[6][3][6].

16. Two questions about a store instruction.
    a. This store instruction intends to put a number into an array. Note that the number 284280 that is given to us is a byte offset. Divide it by 4 to obtain the word offset: 71070. The statement is: a[71070] = x;

    b. The mistakes are:
       The lui instruction must appear before the ori instruction.
       The 32-bit number 0x45678 was not split correctly into 16-bit halves.
       Finally, we should not overwrite the base address in $s2.

       The corrected code is:

```
        lui $t0, 4
        ori $t0, $t0, 0x5678
        add $t0, $t0, $s2
        sw $s1, 0($t0)
```

17. After we eliminate the j instruction, the code becomes:

```
        li $s0, 0
        bgt $s1, 100, endloop
loop:
        add $s0, $s0, $s1
        addi $s1, $s1, 1
        ble $s1, 100, loop
endloop:
```

18. The blanks can be filled in as follows:
    –2
    bge
    1 (0 would also work)
    2
    ble
    9 (10 would also work)

    Note that bge could be replaced with bgt, and ble could be replaced with blt, but this would affect the choice of constant being compared on the next line.

19. Here is a diagram of RAM in the vicinity of the run-time stack:

	Area of RAM with lower addresses
$sp →	
	Value of $s0
	Return address to A
	Parameter #2
	Parameter #1
	Value of $s0
	Return address to A
	Parameter #2
	Parameter #1
	Value of $s0
	Return address to main
	Parameter #2
	Parameter #1
	Area of RAM with higher addresses

20. A question comparing two machines A and B.
    a. A runs program 1 for 1000 seconds, so 2600 seconds are available for program 2.
    b. Since program 2 takes 4 seconds to run, we can run it 2600/4 = 650 times per hour.
    c. B runs program 1 for 1400 seconds, so 2200 seconds are available for program 2.
    d. Since program 2 takes 10 seconds, we can run it 2200/10 = 220 times per hour.

    e. The cost effectiveness of A is 650 / 10000 = 0.065 jobs per hour per dollar.
       The cost effectiveness of B is 220 / 5000 = 0.044 jobs per hour per dollar.
    f. A is better than B by a factor of 65/44, which is approximately 1.48.