Date:_____

Quiz for Chapter 2 Instructions: Language of the Computer

Not all questions are of equal difficulty. Please review the entire quiz first and then budget your time carefully.

Name:_____

Course:_____

Solutions in Red

1. [5 points] Prior to the early 1980s, machines were built with more and more complex instruction set. The MIPS is a RISC machine. Why has there been a move to RISC machines away from complex instruction machines?

There are number of reasons for the move towards RISC machines away from CISC. Some of them are:

- Since early computers had limited memory capacities and were expensive, having a CISC instruction set enabled performing complex operations with very few instructions (encoded within a smaller memory size compared to a corresponding RISC program). Since then memories have got cheaper and there has been a lot of advances in the design of cache hierarchies (for example, a dedicated on-chip instruction cache, prefetching techniques, etc.) that permit RISC machines work-around longer instruction sequences

- Writing a compiler to generate efficient code is easier for a RISC architecture than for a CISC architecture as the compiler can take advantage of a lot of registers provided by the RISC architecture than a CISC

- RISC instructions are easier to pipeline than CISC instructions

2. [5 points] Write the following sequence of code into MIPS assembler:

```
x = x + y + z - q;
```

Assume that `x, y, z, q` are stored in registers `$s1-$s4`.

The MIPS assembly sequence is as follows:

```
add $t0, $s1, $s2
add $t1, $t0, $s2
sub $s1, $t1, $s4
```

3. [10 points] In MIPS assembly, write an assembly language version of the following C code segment:

```
int A[100], B[100];
for (i=1; i < 100; i++) {
  A[i] = A[i-1] + B[i];
}
```

At the beginning of this code segment, the only values in registers are the base address of arrays `A` and `B` in registers `$a0` and `$a1`. Avoid the use of multiplication instructions–they are unnecessary.

Name:_____

The MIPS assembly sequence is as follows:

```
        li $t0, 1           # Starting index of i
        li $t5, 100         # Loop bound

loop:
        lw $t1, 0($a1)      # Load A[i-1]
        lw $t2, 4($a2)      # Load B[i]
        add $t3, $t1, $t2   # A[i-1] + B[i]
        sw $t3, 4($a1)      # A[i] = A[i-1] + B[i]
        addi $a1, 4         # Go to i+1
        addi $a2, 4         # Go to i+1
        addi $t0, 1         # Increment index variable
        bne $t0, $t5, loop  # Compare with Loop Bound

halt:
        nop
```

4. [6 points] Some machines have a special flag register which contains status bits. These bits often include the *carry* and *overflow* bits. Describe the difference between the functionality of these two bits and give an example of an arithmetic operation that would lead to them being set to different values.

The *carry* flag is set when arithmetic operation results in generating a carry bit out of the most significant bit position of its operand. The *overflow* flag is set when an arithmetic operation results in generating a carry bit out of the most significant bit position of the physical register which holds its operand. An overflow means that the register size is not big enough to hold the result of the current arithmetic operation while a carry just indicates that the resulting value's most significant bit position is higher (or lower in case of a borrow) than its operand by a bit position. When we add two integers: 0x0100 and 0x0110 which are held in 16-bit registers, the result 0x1010 generates a carry but not a overflow bit.

5. [6 points] The MIPS instruction set includes several shift instructions. They include logical-shift-left, logical-shift-right, and arithmetic-shift-right. Other architectures only provide an arithmetic-shift-right instruction.

a) Why doesn't MIPS offer an "arithmetic-shift-left" opcode?

The logical and arithmetic left shift operations are the same. That is why there is no need for a separate arithmetic left shift operation.

b) How would you implement in the assembler a logical-shift-left (LSL) pseudo-operation for a machine that didn't have this particular instruction? Be sure your LSL instruction can shift up to W-bits where W is the machine word size in bits.

Logical left shift operation corresponds to multiplication by 2. Implementing

```
        sll $s1, $s2, n
```

can be done via the following loop:

```
        li $t0, 0
        li $t1, n
        add $s1, $s2, 0
```

Name:_____

```
loop:
      mul $s1, $s1, 2
      sub $t0, 1
      bne $t0, $t1, loop
```

Here the `mul` instruction can easily be implemented using add operations (using a loop similar to above) if it is not provided natively in the instruction set.

6. [6 points] Consider the following assembly code for parts 1 and 2.

```
r1 = 99
```

Loop:

```
  r1 = r1 - 1
  branch r1 > 0, Loop
  halt
```

(a) During the execution of the above code, how many dynamic instructions are executed?

The Loop instructions execute for a total of 100 times. The number of dynamic instructions in the code is 102

(b) Assuming a standard unicycle machine running at 100 KHz, how long will the above code take to complete?

Execution Time = $102*1/(100*10^3) = 10.2$ microseconds

7. [15 points] Convert the C function below to MIPS assembly language. Make sure that your assembly language code could be called from a standard C program (that is to say, make sure you follow the MIPS calling conventions).

```
unsigned int sum(unsigned int n)
{
       if (n == 0) return 0;
       else return n + sum(n-1);
}
```

This machine has no delay slots. The stack grows downward (toward lower memory addresses). The following registers are used in the calling convention:

Register Name	Register Number	Usage
$zero	0	Constant 0
$at	1	Reserved for assembler
$v0, $v1	2, 3	Function return values
$a0 - $a3	4 – 7	Function argument values
$t0 - $t7	8 – 15	Temporary (caller saved)
$s0 - $s7	16 – 23	Temporary (callee saved)
$t8, $t9	24, 25	Temporary (caller saved)
$k0, $k1	26, 27	Reserved for OS Kernel
$gp	28	Pointer to Global Area
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

Name:_____

The MIPS code is as follows:

```
sum:
        addi $sp, $sp, -8           # Set up the stack
        sw $ra, 4($sp)              # Save return address
        addi $t0, $a0, 0            # Initialize the sum
        li $v0, 0                   # Initialize return value
        beq $t0, 0, return  # If argument is 0 then return
        subi $t1, $t0, 1            # Compute n-1
        sw $t0, 8($sp)              # Save caller saved regs
        addi $a0, $t1, 0            # Move n-1 into argument register
        jal sum                     # Call sum
        lw $t0, 8($sp)              # Restore caller saved reg
        add $v0, $t0, $v0   # Add return value to $t0
        lw $ra, 4($sp)              # Get the return address

return:
        jr $ra                      # Return
```

8. [5 points] In the snippet of MIPS assembler code below, how many times is instruction memory accessed? How many times is data memory accessed? (Count only accesses to memory, not registers.)

```
lw $v1, 0($a0)
addi $v0, $v0, 1
sw $v1, 0($a1)
addi $a0, $a0, 1
```

The instruction memory is accessed four times (as there are four instructions) and the data memory is accessed twice (once for the `lw` instruction and another time for the `sw` instruction).

9. [6 points] Use the register and memory values in the table below for the next questions. Assume a 32-bit machine. Assume each of the following questions starts from the table values; that is, DO NOT use value changes from one question as propagating into future parts of the question.

Register	Value	Memory Location	Value
R1	12	12	16
R2	16	16	20
R3	20	20	24
R4	24	24	28

a) Give the values of `R1`, `R2`, and `R3` after this instruction: `add R3, R2, R1`

After `add R3, R2, R1`:

`R1 = 12, R2 = 16 and R3 = 20`

b) What values will be in `R1` and `R3` after this instruction is executed: `load R3, 12(R1)`

After load `R3, 12(R1)`:

`R3 = 16 and R1 = 12`

Name:_____

c) What values will be in the registers after this instruction is executed: `addi R2, R3, #16`

After `addi R2, R3, 16`:

`R2 = 16 and R3 = 20`

10. [20 points] Loop Unrolling and Fibonacci: Consider the following pseudo-C code to compute the fifth Fibonacci number (F(5)).

```
1 int a,b,i,t;
2 a=b=1; /* Set a and b to F(2) and F(1) respectively */
3 for(i=0;i<2;i++)
4 {
5 t=a; /* save F(n-1) to a temporary location */
6 a+=b; /* F(n) = F(n-1) + F(n-2) */
7 b=t; /* set b to F(n-1) */
8 }
```

One observation that a compiler might make is that the loop construction is somewhat unnecessary. Since the the range of the loop indices is fixed, one can unroll the loop by simply writing three iterations of the loop one after the other without the intervening increment/comparison on i. For example, the above could be written as:

```
1 int a,b,t;
2 a=b=1;
3 t=a;
4 a+=b;
5 b=t;
6 t=a;
7 a+=b;
8 b=t;
```

(a) Convert the pseudo-C code for both of the snippets above into reasonably efficient MIPS code. Represent each variable of the pseudo-C program with a register. Try to follow the pseudo-C code as closely as possible (i.e. the first snippet should have a loop in it, while the second should not).

MIPS code for Loop:

```
        li $t0, 1       # a = 1
        li $t1, 1       # b = 1
        li $t3, 3       #  Loop bound
        li $t4, 0
loop:
        addi $t2, $t0, 0     # t = a
        add $t0, $t0, $t1    # a += b
        addi $t1, $t2, 0     # b = t
        subi $t3, $t3, 1     # Loop Index decrement
        bne $t3, $t4, loop
```

MIPS code for unrolled version:

```
        li $t0, 1           # a = 1
        li $t1, 1           # b = 1
        addi $t2, $t0, 0    # t = a
```

Name:_____

```
add $t0, $t0, $t1   # a += b
addi $t1, $t2, 0    # b = t
addi $t2, $t0, 0    # t = a
add $t0, $t0, $t1   # a += b
addi $t1, $t2, 0    # b = t
addi $t2, $t0, 0    # t = a
add $t0, $t0, $t1   # a += b
addi $t1, $t2, 0    # b = t
```

(b) Now suppose that instead of the fifth Fibonacci number we decided to compute the 20th. How many static instructions would there be in the first version and how many would there be in the unrolled version? What about dynamic instructions? You do not need to write out the assembly for this part.

The number of static instructions in the first case would still remain the same, which is 9. In case of the unrolled version, the number of static instructions would be $2 + 3*(20 - 2) = 56$. The number of dynamic instructions in the first case would be $4 + 5*(20 - 2) = 94$ while in the unrolled case, the number of dynamic instructions is same as the number of static instructions, which is 56.

11. [10 points] In MIPS assembly, write an assembly language version of the following C code segment:

```
for (i = 0; i < 98; i ++) {
  C[i] = A[i + 1] - A[i] * B[i + 2]
}
```

Arrays A, B and C start at memory location *A*000*hex*, *B*000*hex* and *C*000*hex* respectively. Try to reduce the total number of instructions and the number of expensive instructions such as multiplies.

The MIPS assembly sequence is as follows:

```
       li $s0, 0xA000          # Load Address of A
       li $s1, 0xB000          # Load Address of B
       li $s2, 0xC000          # Load Address of C
       li $t0, 0               # Starting index of i
       li $t5, 98              # Loop bound
loop:
       lw $t1, 0($s1)          # Load A[i]
       lw $t2, 8($s2)          # Load B[i+2]
       mul $t3, $t1, $t2       # A[i] * B[i+2]
       lw $t1, 4($s1)          # Load A[i+1]
       add $t2, $t1, $t3       # A[i+1] + A[i]*B[i+2]
       sw $t2, 4($s3)          # C[i] = A[i+1] + A[i]*B[i+2]
       addi $s1, 4             # Go to A[i+1]
       addi $s2, 4             # Go to B[i+1]
       addi $s3, 4             # Go to C[i+1]
       addi $t0, 1             # Increment index variable
       bne $t0, $t5, loop      # Compare with Loop Bound
halt:
       nop
```

Name:_____

12. [6 points] Suppose that a new MIPS instruction, called `bcp`, was designed to copy a block of words from one address to another. Assume that this instruction requires that the starting address of the source block be in register `$t1` and that the destination address be in `$t2`. The instruction also requires that the number of words to copy be in `$t3` (which is > 0). Furthermore, assume that the values of these registers as well as register `$t4` can be destroyed in executing this instruction (so that the registers can be used as temporaries to execute the instruction).

Do the following: Write the MIPS assembly code to implement a block copy without this instruction. Write the MIPS assembly code to implement a block copy with this instruction. Estimate the total cycles necessary for each realization to copy 100-words on the multicycle machine.

The MIPS code to implement block copy without the bcp instruction is as follows:

```
loop:
        lw $t4, 0($t1)
        sw $t4, 0($t2)
        addi $t1, $t1, 4
        addi $t2, $t2, 4
        subi $t3, $t3, 1
        bne  $t3, $zero, loop
```

To implement block copy with this instruction:

```
        li $t1, src
        li $t2, dst
        li $t3, count
        bcp
```

Assuming each instruction in the MIPS code of 'loop' takes 1 cycle, for doing a 100-word copy the total number of cycles taken is 6*100 = 600 cycles.