Name: _____

CS 33003 **Exam #3** **CompOrg**

Friday 13 November 1998

1. Given the following C code fragment:

```
if (a < b)
    a = b + 2;
else
    a = b + 4;
```

In the space below, translate this code fragment into the book's LOAD / STORE format. Assume that space has already been allocated for variables *a* and *b* earlier in the program, and they have been given initial values. (20 points)

```
        LOAD    R0,a
        LOAD    R1,b
        CMP     R0,R1
        BRLT    then
        JUMP    else
then:   ADD     R0,R1,#2
        JUMP    end
else:   ADD     R0,R1,#4
end:
```

2. Given a 32-bit value in register R3, write a one-instruction code fragment in the book's LOAD / STORE format that will set the 8 most significant bits to all zeros, and leave the remaining 24 bits untouched. Indicate a binary value by preceding it with "b", as in "#b10000000" for 128_{10}. (5 points)

```
        AND     R3,R3,#b00000000111111111111111111111111
or      BCLR    R3,31,24
```

3. In the space below each of the following 3 statements, explain what the statement does, making it clear to me that you understand the difference between them. (5 points each = 15 points)

a. `.equate x 100`

Defines the symbol "x" to stand for the value "100"; does not allocate any memory space.

b. `x: .reserve 100`

Allocates 100 bytes of unitialized memory space, which is referred to as "x".

c. `x: .word 100`

Name: _____

Allocates 1 word (4 byte) of memory space, initialized to the value "100"; this word is referred to as "x".

4. Given the following C code fragment, where *arr* is an array of 5 ints:

```
for (i=0; i<5; i++)
    arr[i] = i;
```

In the space below, translate this code fragment into the book's LOAD / STORE format, using indexed addressing to access the array *arr*. Assume that uninitialized space has already been allocated for variable *arr* earlier in the program, and *i* can be held in register R0. (20 points)

```
        LOAD    R0,#0       ; i
        LOAD    R1,#0       ; index
test:   CMP     R0,#5       ; if (i<5)
        BRLT    for
        JUMP    end
for:    STORE   arr[R1],R0  ; arr[index] = i
        ADD     R0,R0,#1    ; i++
        ADD     R1,R1,#4    ; index += 4
        JUMP    test
end:
```

5. Given the following assembly language code fragment, which begins execution at label "main":

```
sub1: …
      JSR   sub2,R31
      …
      JUMP@R31

sub2: …
      …
      JUMP@R31

main: …
      …
      JSR   sub1,R31
      …
      …
```

a. What problems will occur when the code executes? Be specific. (10 points)

As sub1 starts to execute, R31 contains the return address — the address of the instruction following the JSR in main. However, when sub1 runs, the JUMP to sub2 will put a new return value into R31— the address of the instruction following the JSR in sub1. This destroys the previous value, so once sub1 finishes, when it tries to return to the line following the JSR in main, it will go instead to the line following the JSR in sub1 — essentially stuck forever in an infinite loop.

Name: _____

b. How can this problem be avoided? (5 points)

Save the R31 value at the beginning of sub1, and restore it before returning from sub1.

6. Show the 5-bit representation of each of the decimal values below in each of the specified formats. If it is not possible to represent a particular value in a particular format, write "not possible" in that location. (15 points)

Value	Signed Magnitude	Excess 16	2's Complement
16	not possible	not possible	not possible
15	01111	11111	01111
2	00010	10010	00010
–15	11111	00001	10001
–16	not possible	00000	10000

7. This question concerns the SPARC architecture.

a. What does the instruction "ld [%l3+%l4],%l5" do? (5 points)

Adds the values in local registers %l3 and %l4 to produce an address, gets the value at that address in memory, and stores the result in local register %l5.

b. What does the instruction "set arr,%l5" do, and what is unusual about this instruction? (5 points)

Stores the address of "arr" into local register %l5. It is unusual in that it is a synthetic instruction, known only to the assembler; it is actually implemented in machine language using the "sethi" and "or" instructions.