CSE 30
Spring 2010
Midterm Exam

1. Number Systems ___________________ (15 points)
2. Binary Addition/Condition Code Bits/Overflow Detection ___________________ (12 points)
3. Branching ___________________ (22 points)
4. Bit Operations / C Runtime Environment ___________________ (17 points)
5. Parameter Passing and Return Values (Stack Variables) ___________________ (12 points)
6. Local Variables, The Stack and Return Values ___________________ (16 points)
7. Load/Store/Memory ___________________ (11 points)

SubTotal ___________________ (105 points)
Extra Credit ___________________ (12 points)
Total ___________________
1. Number Systems

Convert 0xFB99 (2’s complement, 16-bit word) to the following. (6 points)

- **binary** 
  
- **octal** 
  0

- **decimal** 
  (convert to signed decimal)

Convert -341 to the following (assume 16-bit word). **Express answers in hexadecimal.** (6 points)

- **sign-magnitude** 0x

- **1’s complement** 0x

- **2’s complement** 0x

Convert +448 to the following (assume 16-bit word). **Express answers in hexadecimal.** (3 points)

- **sign-magnitude** 0x

- **1’s complement** 0x

- **2’s complement** 0x

2. Binary Addition/Condition Code Bits/Overflow Detection

Indicate what the condition code bits are when adding the following 8-bit 2’s complement numbers. (12 points)

```
  11010111
+10101001
-------
```

```
  00100001
+01001001
-------
```

```
  10000000
+10000000
-------
```

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. Branching (22 points)
Translate the C code below into the equivalent unoptimized SPARC Assembly code. Just perform a direct translation – no optimizations. Use the local register mappings for the variables in assembly as specified.

\[
\begin{align*}
\text{C} & \quad \text{SPARC ASSEMBLY} \\
/* Assume variables a and b have been 
\quad \text{properly declared as ints. */} & \quad \text{! a is mapped to %l3} \\
\text{if ( (a > b) || (a < 22) )} & \quad \text{! b is mapped to %l6} \\
\quad \{ & \\
\quad \quad a = b + 44; & \\
\quad \}\ & \\
\text{else} & \\
\quad \{ & \\
\quad \quad --b; & \\
\quad \quad \text{do} & \\
\quad \quad \quad \{ & \\
\quad \quad \quad \quad a = a \% b; & \\
\quad \quad \quad \} & \text{while ( a >= 88 );} \\
\quad \} & \\
\end{align*}
\]
4. Bit Operations / C Runtime Environment

What is the value of %l0 after each statement is executed? **Express your answers in hexadecimal.**

- set 0xDEADBEEF, %l0
- sra %l0, 9, %l0

Value in %l0 is 0x______________________________ (2 points)

- set 0xDEADBEEF, %l0
- sll %l0, 11, %l0

Value in %l0 is 0x______________________________ (2 points)

- set 0xDEADBEEF, %l0
- set 0x????????, %l1
- xor %l0, %l1, %l0 ! Value in %l0 is now 0xCAFEBABE

Value set in %l1 must be this bit pattern 0x______________________________ (3 points)

---

Fill in the names of the 5 areas of the C Runtime Environment as laid out by the SPARC architecture. Then state what parts of a C program are in each area. (10 points)

<table>
<thead>
<tr>
<th>Low Memory</th>
<th>Stack</th>
<th>Global Memory</th>
<th>Low Stack Frame</th>
<th>High Memory</th>
</tr>
</thead>
</table>

- ________________________________
- ________________________________
- ________________________________
- ________________________________
- ________________________________
5. Parameter Passing and Return Values (Local Stack Struct Variable)

Write the equivalent **unoptimized** SPARC assembly language instructions to perform the following C code fragment. You can assume just this one local variable. (12 points)

```c
/* Function Prototype */
char foo(int, short, unsigned short);
/* ... Other code ... */
/* Assume this local variable is declared appropriately and is the only local var. */

struct fubar {
    char a[5];
    short b;
    unsigned short c;
    int d;
} fb;  /* Local variable fb */
/* ... Other code ... */

/*
Write the code for just this function call, saving the return value appropriately
*/

fb.a[3] = foo(fb.d, fb.b, fb.c);
```

```sparc
/*...*/
```

Put your SPARC Assembly code in the box below.
6. Local Variables, The Stack, and Return Values
Here is a C function that doesn’t do much but allocate local variables, perform statements, and returns a value:

```c
int fubar( int a, int b ) {
    int *local_stack_var1;
    int local_stack_var2[6];

    *local_stack_var1++ = a; /* statement 1 */
    local_stack_var1 = &local_stack_var2[2]; /* statement 2 */
    b = local_stack_var2[4]; /* statement 3 */
    *local_stack_var1 = 9990999; /* statement 4 */
    return ( ++a + local_stack_var2[1] ); /* statement 5 */
}
```

Now write the equivalent unoptimized SPARC assembly language instructions to perform the equivalent. **You must allocate all local variables on the Stack.** Perform each instruction literally. **No short-cuts.** Draw a line between groups of instructions to indicate which instructions are associated with each C statement. (16 points)

```sparc
.global fubar
.section "".text"
fubar: /* Your unoptimized code goes below this point */
```
7. Load/Store/Memory Specify the hex values requested after those lines have been fully executed. (11 points)

.global main

.section ".data"
fmt: .asciz "0x%08X\n" ! prints value as hex 0xXXXXXXXX

c: .byte 0xDD

.align 2
s: .half 0xCafe

.align 4
i1: .word 0x12345678
i2: .word 0x12345678
i3: .word 0x12345678
x: .word 0xAAAA0000

.section ".text"
main:
save %sp, -96, %sp
set x, %l0
set s, %l1
lduh [%l1], %l2   _____________________ Hex value in %l2
stb %l2, [%l0+1]  _____________________ Hex value in word labeled x
srl %l2, 8, %l2  _____________________ Hex value in %l2
stb %l2, [%l0+2]
set fmt, %o0
ld [%l0], %o1
call printf   _____________________ Hex value in word labeled x
(none as output of this printf)
set i1, %l0
set c, %l1
ldub [%l1], %l2   _____________________ Hex value in %l2
sth %l2, [%l0+2]
stb %l2, [%l0+1]
set fmt, %o0
ld [%l0], %o1
call printf   _____________________ Hex value in word labeled i1
(none as output of this printf)
set i2, %l0
set i3, %l1
ld [%l1], %l2   _____________________ Hex value in %l2
sth %l2, [%l0]
set fmt, %o0
ld [%l0], %o1
call printf   _____________________ Hex value in word labeled i2
(none as output of this printf)

ret
restore
Extra Credit (12 points)
What gets printed at each printf() statement given the following C program? (8 pts)

```c
#include <stdio.h>

int main()
{
    char a[] = "SPARC";
    char *p = a;

    printf("%c", p[2] + 1);
    printf("%c", *(a+4) = *p);
    printf("%c", *p++);
    printf("%c", *p--);
    printf("%c", *++p = p[2] + 2);
    printf("%c", *(a+4) = *p);
    printf("\n%s\n", a);
    return 0;
}
```

Using the Rt-Lt Rule, write the C variable definition for the variable named fubar that is a pointer to a function that takes a single argument of type pointer to int and returns a pointer to a double. (4 pts)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>++ postfix increment</td>
<td>L to R</td>
</tr>
<tr>
<td>-- postfix decrement</td>
<td></td>
</tr>
<tr>
<td>[] array element</td>
<td></td>
</tr>
<tr>
<td>() function call</td>
<td></td>
</tr>
<tr>
<td>* indirection</td>
<td>R to L</td>
</tr>
<tr>
<td>++ prefix increment</td>
<td></td>
</tr>
<tr>
<td>-- prefix decrement</td>
<td></td>
</tr>
<tr>
<td>&amp; address-of type or object</td>
<td></td>
</tr>
<tr>
<td>(type) type cast</td>
<td></td>
</tr>
<tr>
<td>* multiplication</td>
<td>L to R</td>
</tr>
<tr>
<td>/ division</td>
<td></td>
</tr>
<tr>
<td>% modulus</td>
<td></td>
</tr>
<tr>
<td>+ addition</td>
<td>L to R</td>
</tr>
<tr>
<td>- subtraction</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>= assignment</td>
<td>R to L</td>
</tr>
</tbody>
</table>
Scratch Paper