Name _________________________
Student ID __________________
Signature_______________________

CSE 30
Spring 2008
Midterm Exam

1. Number Systems ___________________  (15 points)
2. Binary Addition/Condition Code Bits/Overflow Detection ___________________  (12 points)
3. Branching ___________________  (23 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 ___________________  (15 points)
7. Load/Store/Memory ___________________  (11 points)

SubTotal ___________________  (105 points)

Extra Credit ___________________  (7 points)

Total ___________________
1. Number Systems

Convert 0xFB5C (2’s complement, 16-bit word) to the following. (6 points)

- **binary** ___________________________ (straight base conversion)
- **octal** 0___________________________ (straight base conversion)
- **decimal** ___________________________ (convert to signed decimal)

Convert -325 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 +462 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)

\[
\begin{array}{cccc}
11010111 & +10001001 & 00100101 & +00111001 \\
01000101 & +00111001 & 01011011 & +00100101 \\
\end{array}
\]

\[
\begin{array}{cccccc}
| & | & | & | & | & | & | & | & | & |
\end{array}
\]
3. Branching (23 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.

C
/* Assume variables a and b have been properly declared as ints. */

for ( b = 7575; a > b; a = a + b )
{
    if ( a <= 42 )
    {
        b = b - 23;
    }
    else
    {
        a = b % 49;
    }
}

SPARC ASSEMBLY
! a is mapped to %l3
! b is mapped to %l6
4. Bit Operations / C Runtime Environment

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

```c
set 0xACDC3579, %l0
sra %l0, 13, %l0
```

Value in %l0 is 0x________________________ (2 points)

```c
set 0xACDC3579, %l0
sll %l0, 11, %l0
```

Value in %l0 is 0x________________________ (2 points)

```c
set 0xACDC3579, %l0
set 0x????????, %l1
xor %l0, %l1, %l0
```

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

Value in %l0 is now 0xCAFEBABE

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>high memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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( char, int, unsigned short );

/* ... Other code ... */

struct fubar {
    char a;
    unsigned short b[3];
    char c;
    int d[4];
} fb;  /* Local variable fb */

/* ... Other code ... */

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

fb.c = foo( fb.a, fb.d[2], fb.b[0] );
```

**SPARC assembly**

```asm
/* ... Other code ... */

/* 
   Put your SPARC Assembly code in the box below. 
*/

/* 
   Put your SPARC Assembly code in the box below. 
*/

```

5. Parameter Passing and Return Values (Local Stack Variables)
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 x, int y ) {
    int local_stack_var1[4];
    int *local_stack_var2;

    *local_stack_var2 = 24680;                 /* statement 1 */
    x = local_stack_var1[2];                   /* statement 2 */
    *local_stack_var2++ = y;                   /* statement 3 */
    local_stack_var2 = &local_stack_var1[1];   /* statement 4 */
    return ( local_stack_var1[0] + y );        /* 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. (15 points)

```sparc
.globl fubar
.sect " .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 0x99
.s: .half 0xFACE
.il: .word 0x9ABCD123
.i2: .word 0x9ABCD123
.i3: .word 0x9ABCD123
.x: .word 0x55550000

.section ".text"
main:
    save  %sp, -96, %sp
    set   x, %l0
    set   s, %l1
    lduh  [%l1], %l2  ---------------------------------- Hex value in %l2
    std   %l2, [%l0+3]  ----------------------------------  Hex value in word labeled x

    srl   %l2, 4, %l2  ---------------------------------- Hex value in %l2
    std   %l2, [%l0+1]

    set   fmt, %o0
    ld    [%l0], %o1
    call  printf  ---------------------------------- Hex value in word labeled x (same as output of this printf)
    nop

    set   il, %l0
    set   c, %l1
    ldsb  [%l1], %l2  ---------------------------------- Hex value in %l2
    sth   %l2, [%l0]  ---------------------------------- Hex value in word labeled il
    std   %l2, [%l0+3]

    set   fmt, %o0
    ld    [%l0], %o1
    call  printf  ---------------------------------- Hex value in word labeled il (same as output of this printf)
    nop

    set   i2, %l0
    set   i3, %l1
    ld    [%l1], %l2  ---------------------------------- Hex value in %l2
    std   %l2, [%l0+1]

    sra   %l2, 12, %l2  ---------------------------------- Hex value in %l2
    sth   %l2, [%l0+2]

    set   fmt, %o0
    ld    [%l0], %o1
    call  printf  ---------------------------------- Hex value in word labeled i2 (same as output of this printf)
    nop

    ret
    restore
Extra Credit (7 points)

What gets printed at each printf() statement given the following C program?

```c
#include <stdio.h>

int main()
{
    char a[] = "CSE30";
    char *p = a;

    printf("%c", ++*p );
    printf("%c", *(p+3) = *p); 
    printf("%c", *++p );
    printf("%c", --*p++ );
    printf("%c", *p++ );
    printf("%d", ++p - a );
    printf("\n%s\n", a);
    return 0;
}
```

A portion of the Operator Precedence Table

<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>* 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</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>
</tbody>
</table>

= assignment R to L
Scratch Paper
Scratch Paper