

Name _____

Signature _____

cs30x _____

Student ID _____

**CSE 30
Winter 2008
Midterm Exam**

1. Number Systems _____ **(15 points)**

2. Binary Addition/Condition Code Bits/Overflow Detection _____ **(12 points)**

3. Branching _____ **(21 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 _____ **(9 points)**

SubTotal _____ **(101 points)**

Extra Credit _____ **(7 points)**

Total _____

1. Number Systems

Convert **0xFA4A** (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 **-317** 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 **+488** 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)

$$\begin{array}{r} 11010111 \\ +00001001 \\ \hline \end{array}$$

N Z V C

--	--	--	--

$$\begin{array}{r} 11000101 \\ +10111001 \\ \hline \end{array}$$

N Z V C

--	--	--	--

$$\begin{array}{r} 01111111 \\ +10000001 \\ \hline \end{array}$$

N Z V C

--	--	--	--

3. Branching (21 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. */

if ( a <= 249 )
{
    while ( b <= 37 )
    {
        b = a * 13;
    }
} else {
    b = b + a;
}
```

SPARC ASSEMBLY

```
! a is mapped to %12
! b is mapped to %15
```

4. Bit Operations / C Runtime Environment

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

```
set 0xABCD5678, %l0
sra %l0, 11, %l0
```

Value in %l0 is **0x**_____ (2 points)

```
set 0xABCD5678, %l0
sll %l0, 13, %l0
```

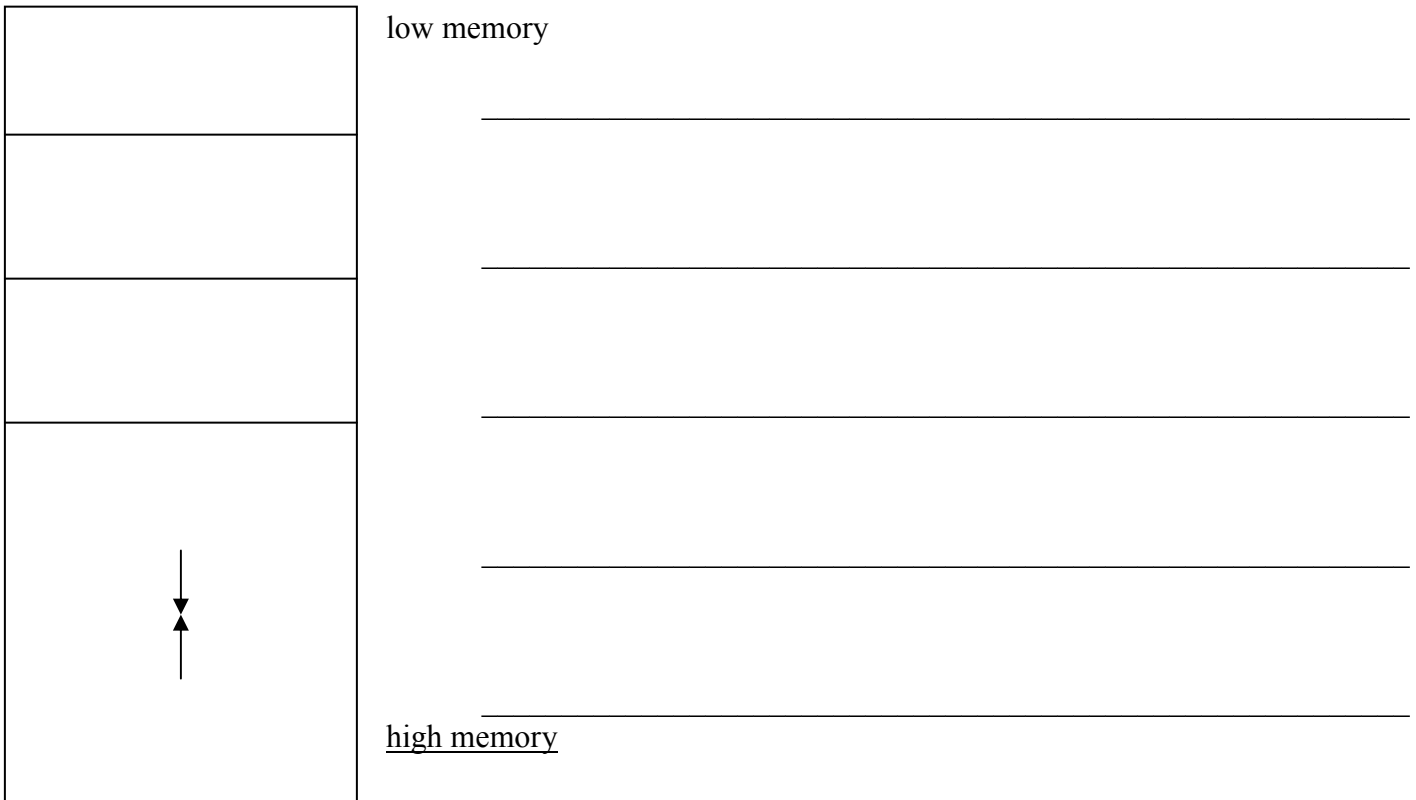
Value in %l0 is **0x**_____ (2 points)

```
set 0xABCD5678, %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)



5. Parameter Passing and Return Values (Local Stack Variables)

Write the equivalent **unoptimized** SPARC assembly language instructions to perform the following C code fragment. **All local variables must be allocated on the run time stack.** (12 points)

C

```
/* Function Prototype */

short foo( short, int, char );

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

/* Local stack variables */

short      a;
char       b[5];
short      c;
int        d[3];

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

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

c = foo( a, d[2], b[1] );
```

SPARC assembly

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 x, int y ) {
    int *local_stack_var1;
    int local_stack_var2[3];

    local_stack_var2[0] = y;          /* statement 1 */
    local_stack_var1 = &local_stack_var2[1]; /* statement 2 */
    *local_stack_var1 = 54321;        /* statement 3 */
    x = *local_stack_var1++;          /* statement 4 */
    return ( x - local_stack_var2[2] ); /* 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 assembly

```
.global      fubar
.section     ".text"
fubar:      /* Your unoptimized code goes below this point */
```

7. Load/Store/Memory

What gets printed in the following program? (9 points)

```
.global main

.section ".data"
fmt: .asciz "0x%X\n"      ! hex 0XXXXXXXXX

c:   .byte 0xEE

    .align 2
s:   .half 0x89AB

    .align 4
i1:  .word 0x87654321
i2:  .word 0x87654321
i3:  .word 0x87654321
x:   .word 0

.section ".text"
main:
    save    %sp, -96, %sp

    set    i1, %10
    set    s, %11
    ldsh   [%11], %12
    sth    %12, [%10+2]

    set    fmt, %0
    ld     [%10], %01
    call   printf
    nop

    set    i2, %10
    set    c, %11
    ldsh   [%11], %12
    sth    %12, [%10]
    ldsh   [%10+3], %13
    stb    %13, [%10+2]

    set    fmt, %0
    ld     [%10], %01
    call   printf
    nop

    set    i3, %10
    set    x, %11
    ldsh   [%10], %12
    sth    %12, [%11+2]
    ldub   [%10+3], %13
    stb    %13, [%11]
    ldub   [%10+2], %14
    stb    %14, [%11+1]
    mov    %11, %10

    set    fmt, %0
    ld     [%10], %01
    call   printf
    nop

    ret
    restore
```

Extra Credit (7 points)

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

```
#include <stdio.h>

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

    printf( "%c", *p++ );           _____
    printf( "%c", ++*p );           _____
    printf( "%c", **p + 1 );        _____
    p++;

    printf( "%c", *p = *p + 3);      _____
    printf( "%c", --*p++ );          _____
    printf( "%d", p - a );           _____
    printf( "%s", a );               _____

    return 0;
}
```

A portion of the Operator Precedence Table

<u>Operator</u>	<u>Associativity</u>
++ postfix increment	L to R
-- postfix decrement	

* indirection	R to L
++ prefix increment	
-- prefix decrement	
& address-of	

* multiplication	L to R
/ division	
% modulus	

+ addition	L to R
- subtraction	

.	
.	
.	

= assignment	R to L

Scratch Paper

Scratch Paper