CSE 30
Fall 2012
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) ___________________ (16 points)
6. Local Variables, The Stack, and Return Values ___________________ (16 points)
7. Load/Store/Memory ___________________ (11 points)

SubTotal ___________________ (108 points)

Extra Credit ___________________ (8 points)

Total ___________________
1. Number Systems

Convert 0xFF27 (2’s complement, 16-bit word) to the following. (6 points)

<table>
<thead>
<tr>
<th>Base</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>0xFFFF</td>
</tr>
<tr>
<td>Octal</td>
<td>0</td>
</tr>
<tr>
<td>Decimal</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Base</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign-magnitude</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>1’s complement</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>2’s complement</td>
<td>0xFFFFFFFF</td>
</tr>
</tbody>
</table>

Convert +444 to the following (assume 16-bit word). Express answers in hexadecimal. (3 points)

<table>
<thead>
<tr>
<th>Base</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign-magnitude</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>1’s complement</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>2’s complement</td>
<td>0xFFFFFFFF</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>Number 1</th>
<th>Number 2</th>
<th>Number 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>00011011</td>
<td>01010111</td>
<td>01100001</td>
</tr>
<tr>
<td>+00100101</td>
<td>+10101001</td>
<td>+01001001</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>---------</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 <= b) && (b > 99) )
{
    do
    {
        a = a * b;
    } while ( a < 55 );

    ++b;
}
else
{
    b = b - 22;
}
```

```sparc-assembly
! a is mapped to %l3
! b is mapped to %l6

if ( (a <= b) && (b > 99) )
{
    do
    {
        a = a * b;
    } while ( a < 55 );

    ++b;
}
else
{
    b = b - 22;
}
```
4. Bit Operations / C Runtime Environment

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

set 0xCAFE1234, %l0
sra %l0, 13, %l0

Value in %l0 is 0x_______________________________________ (2 points)

set 0xCAFE1234, %l0
sll %l0, 10, %l0

Value in %l0 is 0x_______________________________________ (2 points)

set 0xCAFE1234, %l0
set 0x????????, %l1
xor %l0, %l1, %l0 ! Value in %l0 is now 0xD8CAD8CA

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></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>high memory</td>
<td></td>
<td></td>
</tr>
</tbody>
</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)

```
struct fubar {
    char a[3];
    short b;
    unsigned short c[2];
    int d;
} fb; /* Local variable fb */
/* ... Other code ... */

/*
   Write the code for just this function call, saving the return value appropriately
*/
fb.a[1] = foo( fb.b, fb.c[0], fb.d );
/* ... Other code ... */
```

Using the Rt-Lt Rule, write the C variable definition for the variable named `bar` that is a pointer to a function that takes a single argument of type pointer to short and returns a pointer to a char. (4 pts)
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[5];

    local_stack_var1 = &local_stack_var2[2];   /* statement 1 */
    *local_stack_var1 = 5343362;               /* statement 2 */
    local_stack_var2[0] = a;                   /* statement 3 */
    b = *local_stack_var1++;                   /* 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)

```
.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)

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

c:     .byte   0x99
.s:    .half   0xBEAD
.i1:   .word   0x12345678
.i2:   .word   0x12345678
.i3:   .word   0x12345678
.x:    .word   0x77770000

.section ".text"
main:
  save   %sp, -96, %sp
  set    x, %l0
  set    s, %l1
  lduh   [%l1], %l2                  Hex value in %l2
  stb    %l2, [%l0+3]               Hex value in word labeled x
  srl    %l2, 4, %l2                Hex value in %l2
  stb    %l2, [%l0]
  set    fmt, %o0
  ld     [%l0], %o1
  call   printf                      Hex value in word labeled x
  nop
  set    i1, %l0
  set    c, %l1
  ldsb   [%l1], %l2                 Hex value in %l2
  sth   %l2, [%l0+2]                Hex value in word labeled i1
  stb   %l2, [%l0]
  set    fmt, %o0
  ld    [%l0], %o1
  call   printf                     Hex value in word labeled i1
  nop
  set    i2, %l0
  set    i3, %l1
  ld   [%l1], %l2                  Hex value in %l2
  sth  %l2, [%l0]                   Hex value in word labeled i2
  sra  %l2, 16, %l2                Hex value in %l2
  sth  %l2, [%l0+2]
  set    fmt, %o0
  ld   [%l0], %o1
  call   printf                     Hex value in word labeled i2
  nop
ret
restore
```
Extra Credit

What gets printed at each printf() statement given the following C program? (8 pts)

```c
#include <stdio.h>

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

    printf( "%c", *p++ );
    printf( "%c", *(p+2) = p[3] );
    printf( "%c", *(p+3) = p[-1] );
    printf( "%c", *++p );
    printf( "%c", *p++ = *a + 2 );
    printf( "%d", ++p - a );
    printf( "\n%s\n", a );

    return 0;
}
```

A portion of the C Operator Precedence Table

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

Hexadecimal - Character

| 00 NUL | 01 SOH | 02 STX | 03 ETX | 04 EOT | 05 ENQ | 06 ACK | 07 BEL |
| 08 BS  | 09 HT  | 0A NL  | 0B VT  | 0C NF  | 0D CR  | 0E SO  | 0F SI  |
| 10 DC1 | 11 DC2 | 12 DC3 | 13 DC4 | 14 NAK | 15 SYN | 16 ETB |
| 17 CAN | 18 EM  | 19 SUB | 1A ESC | 1B FS  | 1C GS  | 1D RS  | 1E US  |
| 20 SP  | 21 !   | 22 "   | 23 #   | 24 $   | 25 %   | 26 &   | 27 '   |
| 28 (    | 29 )   | 2A *   | 2B +   | 2C ,   | 2D -   | 2E .   | 2F /   |
| 30 0   | 31 1   | 32 2   | 33 3   | 34 4   | 35 5   | 36 6   | 37 7   |
| 38 8   | 39 9   | 3A :   | 3B ;   | 3C <   | 3D =   | 3E >   | 3F ?   |
| 40 @   | 41 A   | 42 B   | 43 C   | 44 D   | 45 E   | 46 F   | 47 G   |
| 48 H   | 49 I   | 4A J   | 4B K   | 4C L   | 4D M   | 4E N   | 4F O   |
| 50 P   | 51 Q   | 52 R   | 53 S   | 54 T   | 55 U   | 56 V   | 57 W   |
| 58 X   | 59 Y   | 5A Z   | 5B [   | 5C \   | 5D ]   | 5E ^   | 5F _   |
| 60 `   | 61 a   | 62 b   | 63 c   | 64 d   | 65 e   | 66 f   | 67 g   |
| 68 h   | 69 i   | 6A j   | 6B k   | 6C l   | 6D m   | 6E n   | 6F o   |
| 70 p   | 71 q   | 72 r   | 73 s   | 74 t   | 75 u   | 76 v   | 77 w   |
| 78 x   | 79 y   | 7A z   | 7B {   | 7C }   | 7D ~   | 7E `   | 7F DEL |