CSE 30
Winter 2014
Midterm Exam

1. Number Systems ___________________  (15 points)
2. Binary Addition/Condition Code Bits/Overflow Detection ___________________  (12 points)
3. Branching ___________________  (24 points)
4. Bit Operations ___________________  (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)
8. Miscellaneous ___________________  (13 points)

Total ___________________  (120 points)

114 points = 100%
6 points = over 5% Extra Credit

This exam is to be taken by yourself with closed books, closed notes, no electronic devices.
You are allowed one side of an 8.5"x11" sheet of paper handwritten by you.
1. Number Systems

Convert \textcolor{red}{0xFBED} \textcolor{blue}{(2’s complement, 16-bit word)} to the following. (6 points)

\begin{itemize}
  \item binary \hspace{1cm} \underline{\hspace{5cm}} \textcolor{green}{(straight base conversion)}
  \item octal \hspace{1cm} \underline{\hspace{5cm}} \textcolor{green}{(straight base conversion)}
  \item decimal \hspace{1cm} \underline{\hspace{5cm}} \textcolor{green}{(convert to signed decimal)}
\end{itemize}

Convert \textcolor{red}{-327} to the following (assume 16-bit word). \textcolor{blue}{Express answers in hexadecimal.} (6 points)

\begin{itemize}
  \item sign-magnitude \underline{\hspace{5cm}}
  \item 1’s complement \underline{\hspace{5cm}}
  \item 2’s complement \underline{\hspace{5cm}}
\end{itemize}

Convert \textcolor{red}{+420} to the following (assume 16-bit word). \textcolor{blue}{Express answers in hexadecimal.} (3 points)

\begin{itemize}
  \item sign-magnitude \underline{\hspace{5cm}}
  \item 1’s complement \underline{\hspace{5cm}}
  \item 2’s complement \underline{\hspace{5cm}}
\end{itemize}

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{itemize}
  \item \begin{align*}
      01010111 & +00101001 \\
      \hspace{1cm} & \hspace{1cm} \\
      \hspace{1cm} & \hspace{1cm}
    \end{align*}
  \item \begin{align*}
      11101101 & +10111001 \\
      \hspace{1cm} & \hspace{1cm} \\
      \hspace{1cm} & \hspace{1cm}
    \end{align*}
  \item \begin{align*}
      11001101 & +00110011 \\
      \hspace{1cm} & \hspace{1cm} \\
      \hspace{1cm} & \hspace{1cm}
    \end{align*}
\end{itemize}

\begin{itemize}
  \item \begin{tabular}{|c|c|c|c|}
    \hline
    N & Z & V & C \\
    \hline
    \hline
  \end{tabular}
  \begin{tabular}{|c|c|c|c|}
    \hline
    N & Z & V & C \\
    \hline
    \hline
  \end{tabular}
  \begin{tabular}{|c|c|c|c|}
    \hline
    N & Z & V & C \\
    \hline
    \hline
  \end{tabular}
\end{itemize}
3. Branching (24 points)
Translate the C code below into the equivalent unoptimized SPARC Assembly code using the control flow specified in class. Just perform a direct translation – no optimizations. Use the local register mappings for the variables in assembly as specified.

C
/* Assume variables x and y have been
 properly defined as ints. */

if ( x <= 420 )
{
    /* x -= 8 same as x = x - 8 */
    for ( x = 9876; x > y; x -= 8 )
    {
        y = x + 37;
    }
    x = y;
}
else
{
    x = y * x;
}

SPARC ASSEMBLY
! x is mapped to %l3
! y is mapped to %l5
4. Bit Operations (17 points)

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

```
set 0x87654321, %l0
sra %l0, 11, %l0
Value in %l0 is 0x__________________________
```

```
set 0x87654321, %l0
sll %l0, 7, %l0
Value in %l0 is 0x__________________________
```

```
set 0x87654321, %l0
set 0x????????, %l1
xor %l0, %l1, %l0 ! Value in %l0 is now 0xCAFEBABE
Value set in %l1 must be this bit pattern 0x__________________________ (2 points)
```

In general, when performing bit-wise operations in C, which data type should you use? _________________

Complete the following expressions with the correct bit-wise operation.
Consider x a single bit - either 0 or 1.

Use the letters below.

<table>
<thead>
<tr>
<th>A) and</th>
<th>D) sll</th>
</tr>
</thead>
<tbody>
<tr>
<td>B) or</td>
<td>E) srl</td>
</tr>
<tr>
<td>C) xor</td>
<td>F) sra</td>
</tr>
</tbody>
</table>

For example:

x ____ 0 will always evaluate to 1 (The answer would be nand)
(x nand 0 will always be a 1)

Another way to look at this:
With two inputs and an output specified, what gate/operation on the two inputs will produce the specified output.

```
x ____ 1 will always evaluate to x
x ____ x will always evaluate to 0
x ____ 0 will always evaluate to 0
x ____ 1 will always evaluate to ~x
x ____ 1 will always evaluate to 1
0x80000000 ____ 31 bits will always evaluate to 0x1
0x1 ____ 31 bits will always evaluate to 0x80000000
0x80000000 ____ 31 bits will always evaluate to 0xFFFFFFFF
```

For the next 2, use 2 different operations (in any order, but not one of the shift operations) to get the same result:

```
x ____ 0 will always evaluate to x (op1)
x ____ 0 will always evaluate to x (op2)
```

A) and  D) sll
B) or  E) srl
C) xor  F) sra

Assume you have two 32-bit registers that are logically being used to implement 64-bit operations. %l0 holds the upper 32 bits of a 64-bit value and %l1 holds the lower 32 bits of this 64-bit value.

```
%l0 %l1
0x1234ABCD 0x56789EF0
```

If we perform a rotate right of 5 bits what hex values will be in the two registers?
5. Parameter Passing and Return Values (Local Stack Variables)

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 */
short foo( char, int, unsigned short );

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

/* Assume this local variable
   is declared appropriately
   and is the only local var. */

struct fubar {
    short          a;
    char           b[5];
    int            c[2];
    unsigned short d[2];
} fb;   /* Local variable fb */

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

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

fb.a = foo( fb.b[3], fb.c[1], fb.d[0] );
```

SPARC assembly

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

```

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[4];
    int *local_stack_var2;

    *local_stack_var2++ = y;               /* statement 1 */
    *local_stack_var2 = local_stack_var1[2]; /* statement 2 */
    local_stack_var2 = &local_stack_var1[1]; /* statement 3 */
    local_stack_var1[0] = 420420;   /* statement 4 */
    return ( local_stack_var1[3] + x );  /* 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 assembly
.globa fubar
.section  "text"
fubar:    /* Your unoptimized code goes below this point */
```
7. Load/Store/Memory Specify the full 32 bit hex values after each line has been fully executed. (11 points)

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

.c: .byte 0xAA
.align 2
s: .half 0x9753
.align 4
i1: .word 0xABCDEF1234
i2: .word 0xABCDEF1234
i3: .word 0xABCDEF1234
x: .word 0x22220000

.sectiton ".text"
main:

save %sp, -96, %sp

set i1, %l0
set s, %l1
ldsh [%l1], %l2  _____________________ Hex value in %l2

sth %l2, [%l0+2]  _____________________ Hex value in word labeled i1
sll %l2, 12, %l2  _____________________ Hex value in %l2
sth %l2, [%l0]  _____________________

set fmt, %o0
ld [%l0], %o1
call printf  _____________________ Hex value in word labeled i1
nop  _____________________ (same as output of this printf)

set x, %l0
set c, %l1
ldsb [%l1], %l2  _____________________ Hex value in %l2

sth %l2, [%l0]  _____________________ Hex value in word labeled x
stb %l2, [%l0+3]  _____________________

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

set i2, %l0
set i3, %l1
lduh [%l1], %l2  _____________________ Hex value in %l2

stb %l2, [%l0+1]  _____________________ Hex value in word labeled i2
sll %l2, 8, %l2  _____________________ Hex value in %l2
sth %l2, [%l0]  _____________________

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

ret
restore
```
8. Miscellaneous (13 points)

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

```c
#include <stdio.h>

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

    printf("%c", *p++ );
    printf("%c", ++*p );
    printf("%c", *++p );
    printf("%c", *(p+2) = *p);
    printf("%c", --*p++ );
    printf("%d", p - a );

    printf("\n%s\n", a );
    return 0;
}
```

What does BSS stand for? ________________________________________________________________

Using the Rt-Lt Rule, write the C variable definition for the variable named foo that is an array of 11 elements where each element is a pointer to a function that takes a pointer to short and returns a pointer to an int.