1. Number Systems  
2. Binary Addition/Condition Code Bits/Overflow Detection  
3. Branching  
4. Bit Operations / C Runtime Environment  
5. Parameter Passing and Return Values (Structures)  
6. Local Variables, The Stack and Return Values  

SubTotal  
Extra Credit  
Total
1. Number Systems

Convert $DADA_{16}$ (2’s complement, 16-bit word) to the following. (6 points)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal</td>
<td></td>
</tr>
<tr>
<td>octal</td>
<td>0</td>
</tr>
<tr>
<td>binary</td>
<td></td>
</tr>
</tbody>
</table>

Convert $+431_{10}$ to the following (assume 16-bit word). **Express answers in hexadecimal.** (3 points)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sign-magnitude</td>
<td>0x--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1’s complement</td>
<td>0x--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2’s complement</td>
<td>0x--------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>

Convert $-274_{10}$ to the following (assume 16-bit word). **Express answers in hexadecimal.** (6 points)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sign-magnitude</td>
<td>0x--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1’s complement</td>
<td>0x--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2’s complement</td>
<td>0x--------------------------------------------------------------------------------------------------</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)

\[
\begin{array}{c}
01010101 \\
+10101011 \\
\hline
\end{array} \quad \begin{array}{c}
11010101 \\
+00111001 \\
\hline
\end{array} \quad \begin{array}{c}
10111001 \\
+10010110 \\
\hline
\end{array}
\]

\[
\begin{array}{cccccc}
N & Z & V & C \\
\hline
| & | & | & |
\end{array} \quad \begin{array}{cccccc}
N & Z & V & C \\
\hline
| & | & | & |
\end{array} \quad \begin{array}{cccccc}
N & Z & V & C \\
\hline
| & | & | & |
\end{array}
\]

3. Branching (15 points)

There are a few different ways to write checkRange() from PA1. Write checkRange with these restrictions:

- The only instructions you can use are: `save`, `cmp`, `nop`, `ble`, `bge`, `ba`, `restore`, `mov`, `ret`
- The following instructions can be used only once: `save`, `ble`, `bge`, `restore`, `ret`
- The other instructions can be used more than once: `cmp`, `nop`, `ba`, `mov`
- The only registers you can use are: `%i0`, `%i1`, `%i2`
- No optimizations allowed (no filling delay slots with anything other than nops)

/* int checkRange( long theValue, long minRange, long maxRange ); */
/* Return value of 0 means No, theValue was not in the range; 1 otherwise. */
4. Bit Operations / C Runtime Environment

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

```
set  0xFEEDFACE, %l0
sll  %l0, 11, %l0
Value in %l0 is 0x_____________________________ (2 points)

set  0xFEEDFACE, %l0
sra  %l0, 9, %l0
Value in %l0 is 0x_____________________________ (2 points)

set  0xFEEDFACE, %l0
set  0xCDCAB072, %l1
xor  %l0, %l1, %l0
Value in %l0 is 0x_____________________________ (2 points)
```

Fill in the names of the 5 areas of the C Runtime Environment as layed out by most Unix operating systems. Then state what parts of a C program are in each area. (10 points)

low memory

__________________________________________________________
__________________________________________________________
__________________________________________________________
__________________________________________________________
__________________________________________________________

high memory
Write the equivalent unoptimized SPARC assembly language instructions to perform the following C code fragment. You can assume just this one local variable. (10 points)

```
C
/* Function Prototype */
int foo( short, char );

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

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

struct fubar {
    int   a;
    char  b;
    int   c;
    short d;
    long  e;
} fb;   /* Local variable fb */

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

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

    fb.c = foo( fb.d, fb.b );
```
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 ) {
    long *local_stack_var1;
    long  local_stack_var2[5];

    local_stack_var1 = local_stack_var2; /* statement 1 */
    y = local_stack_var2[3];              /* statement 2 */
    *local_stack_var1 = 23775;            /* statement 3 */
    local_stack_var1 = local_stack_var1 + 3; /* statement 4 */
    return ( x + 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. (14 points)

```
SPARC assembly
.global fubar

.section " .text"
fubar: /* Your unoptimized code goes below this point */
```
Extra Credit (4 points)

Optimize your solution to the checkRange() problem #3 earlier in this exam. The restrictions now are:

   The only instructions you can use are: save, cmp, ble, bge, ba, restore, mov, ret
   The following instructions can be used only once: save, ble, bge, ba, restore, ret
   The other instructions can be used more than once: cmp, mov
   The only registers you can use are: %i0, %i1, %i2
   Note: there can be no nops
   It is up to you whether you use annulled branches or not (it really doesn’t matter if you do or not).

/* int checkRange( long value, long minRange, long maxRange ); */
/* Return value of 0 means No, theValue was not in the range; 1 otherwise. */
Scratch Paper
Scratch Paper