1. Number Systems ___________________ (15 points)

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

3. Branching ___________________ (20 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 ___________________ (100 points)

Extra Credit ___________________ (6 points)

Total ___________________
1. Number Systems

Convert 0xF939 (2’s complement, 16-bit word) to the following. (6 points)

binary  

octal 0

decimal 

Convert -328 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 +477 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)

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. Branching (20 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 x and y have been properly declared as ints. */
for ( x = 5678; x >= 420; x = x - 70 )
{
    y = x;
    if ( y < 747 )
    {
        y = x - 15;
    } else {
        x = y + 95;
    }
}
```

```sparc
! x is mapped to %11
! y is mapped to %14
```
4. Bit Operations / C Runtime Environment

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

- `set 0x9B9C4321, %l0`
- `sra %l0, 13, %l0`

Value in %l0 is 0x_______________________________________ (2 points)

- `set 0x9B9C4321, %l0`
- `sll %l0, 11, %l0`

Value in %l0 is 0x_______________________________________ (2 points)

- `set 0x9B9C4321, %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)

<table>
<thead>
<tr>
<th>low memory</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>high memory</td>
<td></td>
</tr>
</tbody>
</table>
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
char foo( unsigned short, int, char );
/* ... Other code ... */

/* Local stack variables */
char           a;
unsigned short b[2];
char           c;
int            d[2];
/* ... Other code ... */

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

**SPARC assembly**

```assembly
/* Function Prototype */
char foo( unsigned short, int, char );
/* ... Other code ... */

/* Local stack variables */
char a;
unsigned short b[2];
char c;
int d[2];
/* ... Other code ... */

/* Write the code for just this function call saving the return value appropriately */
a = foo( b[1], d[0], c );
```
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 = 12345;                 /* statement 1 */
    y = *local_stack_var2++;                   /* statement 2 */
    local_stack_var1[0] = x;                   /* statement 3 */
    local_stack_var2 = &local_stack_var1[1];   /* statement 4 */
    return ( y - local_stack_var1[3] );        /* 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 "%x\n"          ! prints value as hex 0xXXXXXXXX

c: .byte 0xBB
.align 2
s: .half 0x9876
.align 4
i1: .word 0x12345678
i2: .word 0x12345678
i3: .word 0x12345678
x: .word 0

.section ".text"
main:
  save  %sp, -96, %sp
  set   i2, %l0
  set   c, %l1
  ldsb  [%l1], %l1
  sth  %l1, [%l0]
  set   fmt, %o0
  ld    [%l0], %o1
  call  printf _________________________________
  nop

  set   i3, %l0
  set   x, %l1
  ldub  [%l0+1], %l2
  sth  %l2, [%l1+2]
  ldsh  [%l0+2], %l2
  stb  %l2, [%l1]
  mov  %l1, %l0
  set   fmt, %o0
  ld    [%l0], %o1
  call  printf _________________________________
  nop

  set   i1, %l0
  set   s, %l1
  ldub  [%l1], %l1
  sth  %l1, [%l0+2]
  set   fmt, %o0
  ld    [%l0], %o1
  call  printf _________________________________
  nop

  ret
  restore
```
Extra Credit (6 points)

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

```c
#include <stdio.h>

int main()
{
    char s[] = "absolute";
    char *p = s;

    printf("%c\n", *p++);
    --*(p+4);
    printf("%c\n", *++p);
    p = p+1;
    *p = *(p-3) + 4;
    printf("%c\n", p[0]);
    *(p+1) = p[1] + 2;
    printf("%c\n", *++p);
    p++;
    printf("%c\n", *p++);
    p[0] = *(p+1);
    printf("%s\n", s);
    return 0;
}
```
Scratch Paper