CSE 30
Winter 2009
Midterm Exam

1. Number Systems ___________________  (15 points)
2. Binary Addition/Condition Code Bits/Overflow Detection ___________________  (12 points)
3. Branching ___________________  (23 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 ___________________  (16 points)
7. Load/Store/Memory ___________________  (11 points)

SubTotal ___________________  (106 points)
Extra Credit ___________________  (7 points)

Total ___________________
1. Number Systems

Convert 0xF65 (2’s complement, 16-bit word) to the following. (6 points)

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

Convert -424 to the following (assume 16-bit word). **Express answers in hexadecimal.** (6 points)

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

Convert +361 to the following (assume 16-bit word). **Express answers in hexadecimal.** (3 points)

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

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Z</td>
<td>V</td>
<td>C</td>
<td>N</td>
<td>Z</td>
<td>V</td>
</tr>
<tr>
<td>11010111</td>
<td></td>
<td></td>
<td></td>
<td>+10101001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+10101001</td>
<td></td>
<td></td>
<td></td>
<td>+01111001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000101</td>
<td></td>
<td></td>
<td></td>
<td>+10100101</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Z</td>
<td>V</td>
<td>C</td>
<td>N</td>
<td>Z</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. Branching (23 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 )
{
    for ( a = 9951; b > a; ++a )
    {
        b = a % 71;
    }
}
else
{
    a = b - 35;
}
```

```sparc
! a is mapped to %l3
! b is mapped to %l6
```
4. Bit Operations / C Runtime Environment

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

set 0x9375ACD1, %l0
sra  %l0, 15, %l0

Value in %l0 is 0x_______________________________________ (2 points)

set 0x9375ACD1, %l0
sll  %l0, 9, %l0

Value in %l0 is 0x_______________________________________ (2 points)

set 0x9375ACD1, %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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>high memory</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)

```
short foo( char, int, unsigned short );

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

fb.a = foo( fb.b[2], fb.c, fb.d[0] );
```

C

SPARC assembly

/* Function Prototype */

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

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

```
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[3];
    int *local_stack_var2;
    local_stack_var2 = &local_stack_var1[1]; /* statement 1 */
    *local_stack_var2 = 8675309; /* statement 2 */
    y = local_stack_var1[2]; /* statement 3 */
    *local_stack_var2++ = x; /* statement 4 */
    return ( ++y + local_stack_var1[0] ); /* 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 l fubar
.section " .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)

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

.c: .byte 0xAA
    .align 2
.s: .half 0xFADE
    .align 4
i1: .word 0x87654321
i2: .word 0x87654321
i3: .word 0x87654321
x: .word 0x66660000

.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+1]
    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]              Hex value in word labeled i1
    stb    %l2, [%l0+3]
    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
    stb    %l2, [%l0+1]           Hex value in word labeled i2
    sra    %l2, 12, %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 (7 points)

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

```c
#include <stdio.h>

int main()
{
    char a[] = "DAT30";
    char *p = a;

    printf("%c", *p++ );  ______
    printf("%c", *(p+3) = *p); ______
    printf("%c", ++*p );  ______
    printf("%c", *++p );  ______
    printf("%c", *++p );  ______
    printf("%c", +++p++ ); ______
    printf("%d", ++p - a );  ______
    printf("\n%s\n", a ); _______________________
    return 0;
}
```

A portion of the Operator Precedence Table

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>++ postfix increment</td>
<td>L to R</td>
</tr>
<tr>
<td>-- postfix decrement</td>
<td></td>
</tr>
<tr>
<td>* indirection</td>
<td>R to L</td>
</tr>
<tr>
<td>++ prefix increment</td>
<td></td>
</tr>
<tr>
<td>-- prefix decrement</td>
<td></td>
</tr>
<tr>
<td>&amp; address-of</td>
<td></td>
</tr>
<tr>
<td>* multiplication</td>
<td>L to R</td>
</tr>
<tr>
<td>/ division</td>
<td></td>
</tr>
<tr>
<td>% modulus</td>
<td></td>
</tr>
<tr>
<td>+ addition</td>
<td>L to R</td>
</tr>
<tr>
<td>- subtraction</td>
<td></td>
</tr>
<tr>
<td>= assignment</td>
<td>R to L</td>
</tr>
</tbody>
</table>
Scratch Paper
Scratch Paper