### CSE 30
#### Spring 2009
#### Midterm Exam

<table>
<thead>
<tr>
<th>Section</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Number Systems</td>
<td>(15 points)</td>
</tr>
<tr>
<td>2. Binary Addition/Condition Code Bits/Overflow Detection</td>
<td>(12 points)</td>
</tr>
<tr>
<td>3. Branching</td>
<td>(21 points)</td>
</tr>
<tr>
<td>4. Bit Operations / C Runtime Environment</td>
<td>(17 points)</td>
</tr>
<tr>
<td>5. Parameter Passing and Return Values (Stack Variables)</td>
<td>(12 points)</td>
</tr>
<tr>
<td>6. Local Variables, The Stack and Return Values</td>
<td>(16 points)</td>
</tr>
<tr>
<td>7. Load/Store/Memory</td>
<td>(11 points)</td>
</tr>
<tr>
<td><strong>SubTotal</strong></td>
<td>(104 points)</td>
</tr>
<tr>
<td><strong>Extra Credit</strong></td>
<td>(12 points)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
</tr>
</tbody>
</table>
1. Number Systems

Convert $0x$FC16 (2’s complement, 16-bit word) to the following. (6 points)

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

Convert $-444$ to the following (assume 16-bit word). Express answers in hexadecimal. (6 points)

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

Convert $+341$ to the following (assume 16-bit word). Express answers in hexadecimal. (3 points)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sign-magnitude</td>
<td>1’s complement</td>
<td>2’s complement</td>
</tr>
<tr>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>$01010111$</td>
<td>$01100001$</td>
<td>$00011011$</td>
</tr>
<tr>
<td>$+10101001$</td>
<td>$+01001001$</td>
<td>$+00100101$</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>
</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>
</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
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, 15, %l0
```

Value in %l0 is \(0x\) \underline{________________________} \hspace{1cm} (2 points)

```
set 0xCAFE1234, %l0
sll %l0, 9, %l0
```

Value in %l0 is \(0x\) \underline{________________________} \hspace{1cm} (2 points)

```
set 0xCAFE1234, %l0
set 0x????????, %l1
xor %l0, %l1, %l0 ! Value in %l0 is now \(0x1234CAFE\)
```

Value set in %l1 must be this bit pattern \(0x\) \underline{________________________} \hspace{1cm} (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>high memory</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>
</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)

```
C

/* Function Prototype */

char foo( short, unsigned short, int );

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

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

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[2] = foo( fb.b, fb.c[1], fb.d );

SPARC assembly

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 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)

```
SPARC assembly
 .global    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)

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

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

.section ".text"
main:
    save  %sp, -96, %sp
    set   x, %l0
    set   s, %l1
    ld uh  [%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   il, %l0
    set   c, %l1
    ldsb  [%l1], %l2               Hex value in %l2
    stb   %l2, [%l0+2]            Hex value in word labeled il
    stb   %l2, [%l0]
    set   fmt, %o0
    ld    [%l0], %o1
    call  printf                    Hex value in word labeled il
    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 (12 points)
What gets printed at each printf() statement given the following C program? (8 pts)

```c
#include <stdio.h>

int main()
{
    char a[] = "SPARC";
    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( "%c", ++*p++ );
    printf( "%d", ++p - a );
    printf( "\n%s\n", a );

    return 0;
}
```

Using the Rt-Lt Rule, write the C variable definition for the variable named foo that is a pointer to an array of 7 elements where each element is a pointer to a char. (4 pts)

---

A portion of the Operator Precedence Table

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