Princeton University
COS 217: Introduction to Programming Systems
Spring 2002 Midterm Exam 2 Answers

Question 1

	Binary	**Octal**	**Hexadecimal**
82	1010010	122	52
31	11111	37	1F

	Signed Magnitude	**1’s Complement**	**2’s Complement**
22	00010110	00010110	00010110
-22	10010110	11101001	11101010
64	01000000	01000000	01000000
-64	11000000	10111111	11000000

```
Advantages of Signed Magnitude:
       Easy to negate a number.
       Easy for people to interpret negative numbers.
       Symmetric:  Range of negative numbers that can be represented is equal to range of
       positive numbers that can be represented.
Disadvantages of Signed Magnitude:
       The number 0 has two representations.
       Need different hardware for addition and subtraction.
Advantages of 1’s Complement:
       Easy to negate a number.
       Same hardware can perform addition and subtraction.
       Symmetric:  Range of negative numbers that can be represented is equal to range of
       positive numbers that can be represented.
Disadvantages of 1’s Complement
       The number 0 has two representations.
       Relatively difficult for people to interpret negative numbers.
Advantages of 2’s Complement:
       The number 0 has only one representation.
       Easy to negate a number.
       Same hardware can perform addition and subtraction.
Disadvantages of 2’s Complement
       Relatively difficult for people to interpret negative numbers.
       Asymmetric:  Range of negative numbers that can be represented is one greater than
       range of positive numbers that can be represented.
```

Question 2

```
a) add reg, reg, reg

b) sub %g0, reg, reg

c) sethi %hi(0xFFFFFFFF), reg
   or reg, %lo(0xFFFFFFFF), reg

   Alternate:
   sub %g0, 1, reg

d) jmpl reg, %o7

f) .byte 0

g) .skip 4 * N
```

<u>Question 3</u>

a) (i) is used to return from a leaf subroutine. (ii) is used to return from a non-leaf subroutine.

b) Store the address of the calling instruction in %o7. Jump to the address specified in the calling instruction. (But first, execute the instruction in the calling instruction's delay slot.)

c) Store the address of the calling instruction in %o7. Jump to the address specified in the calling instruction. (But first, execute the instruction in the calling instruction's delay slot.) Execute a "save" instruction to shift the register window forward; thus the register formerly known as %o7 is now known as %i7.

d) When a subroutine is finished executing, control should return to the instruction following the instruction following the calling instruction.

e) The jmpl instruction stores the address of the current instruction into the register specified as its second operand. When returning from a subroutine, that address is irrelevant. Storing the address to register %g0 discards it.

f) Different calls to a subroutine may need to return to different instructions.

<u>Question 4</u>

In the original C source code, the expression "i<N" should be "i<=N".

Expanded C Source Code:

```
int factorial(int N)
{
   int i;
   int fact;
   fact = 1;
   i = 1;
loop:
   if (i > N) goto loopend;
   fact *= i;
   i++;
   goto loop;
loopend:
   return fact;
}
```

SPARC Assembly Code:

```
!=======================================================
    .section ".text"
!=======================================================

!-------------------------------------------------------
! int factorial(int N)
!
! Return N factorial.
!
! Register map:
! %i0  int N
! %l0  int fact
! %l1  int i
!-------------------------------------------------------

    .align 4
    .global factorial

factorial:

    save %sp, -96, %sp
```

```
! fact = 1;
   mov  1, %l0

! i = 1;
   mov  1, %l1

loop:

! if (i > N) goto loopend;
   cmp  %l1, %i0
   bg   loopend
   nop

! fact *= i;
   mov  %l0, %o0
   mov  %l1, %o1
   call .mul
   nop
   mov  %o0, %l0

! i++;
   inc  %l1

! goto loop;
   ba   loop
   nop

loopend:

! return fact;
   mov  %l0, %i0
   ret
   restore
```

Question 5

a) In the SPARC architecture, each branch instruction has an annul bit. If the bit is set, then the CPU will execute the branch instruction's delay instruction if and only if the branch is taken. (For the ba instruction, if the annul bit is set, then the CPU will not execute the branch instruction's delay instruction.) Assembly language programmers use the annul bit to optimize control structures.

b) A leaf subroutine that modifies register %o7 corrupts its return address.

c) An assembler translates assembly language code into machine language code, storing the result in an object (.o) file.

d) Relocation: An assembler assumes each .o file begins at address 0; when multiple object files are linked together a linker must relocate some addresses. Resolution: A linker must resolve all external references, searching libraries to find any undefined symbols.

e) A user-level process performs functions that execute privileged instructions by executing a "system call." System calls are often implemented using traps. Through a trap, the operating system gains control, switches to supervisor mode, performs a service, switches back to user mode, and gives control back to user.

f) Modern operating systems implement "context switching" faster than older operating systems. (Context switching is the switching of the processor from one process to another.)

Question 6

a)
```
    ld [%l0], %l1
    nop
    add  %l1, 1, %l2
```

b) Suppose the nop instruction is missing. Then the CPU attempts to perform the "execute" cycle of the add instruction at the same time as the "memory fetch" cycle of the ld instruction. But that is impossible, because the "execute" cycle of the add instruction uses the data that the ld instruction fetches from memory. Thus the CPU automatically causes a delay between the ld and add instructions. (Note that the assembly language programmer need not explicitly indicate the delay via a nop instruction; the CPU causes the delay automatically.)

c)
```
    clr  i
    ba   test
    clr  sum
loop:
    ld   [array+offset], value
    inc  i
    add  sum, value, sum
test:
    cmp  i, N
    bl,a loop
    sll  i, 2, offset
done:
    ...
```

<u>Question 7</u>

a) How modified: Through execution of an *X*cc instruction (for example, addcc). How used: Through execution of a conditional branch instruction.

b) How modified: Through execution of a trap instruction. How used: To determine if the CPU is in supervisor mode. A program running in supervisor mode can execute privileged instructions and access privileged memory.

c) How modified: Cleared when a trap occurs; set by the rett (return from trap) instruction. How used: (paraphrasing the Paul book p. 331) The machine can handle only one trap at a time. If a trap occurs when the ET bit is cleared, the machine resets and current execution halts.

d) How modified: Decremented through execution of a save instruction, and incremented through execution of a restore instruction. How used: Selects the current register window, thus defining the current set of input (i), local (l), and output (o) registers.

<u>Question 8</u>

SPARC Assembly Code for "f":

```
   .section ".text"
    .align 4
    .global f
f:
    save %sp, (-92 - 12) & -8, %sp
    mov  0, %o0
    mov  1, %o1
    mov  2, %o2
    mov  3, %o3
    mov  4, %o4
    mov  5, %o5
    mov  6, %l0
    st   %l0, [%sp + 92]
    mov  7, %l0
    st   %l0, [%sp + 96]
    mov  8, %l0
    st   %l0, [%sp + 100]
    call g
    nop
    mov  %o0, %i0
    ret
    restore
```

SPARC Assembly Code for "g":

```
    .section ".text"
    .align 4
   .global g
g:
    save %sp, (-92 - 8) & -8, %sp
    mov  1, %l0
    st   %l0, [%fp – 4]
    mov  2, %l0
    st   %l0, [%fp – 8]
    ld   [%fp – 4], %l0
    mov  %l0, %l1
    ld   [%fp – 8], %l0
    add  %l1, %l0, %l1
    add  %l1, %i0, %l1
    ld   [%fp + 100], %l0
    add  %l1, %l0, %l1
    mov  %l1, %i0
    ret
    restore
```

Stack

Location	Contents
%sp	g register window
%sp + 4	g register window
%sp + 8	g register window
%sp + 12	g register window
%sp + 16	g register window
%sp + 20	g register window
%sp + 24	g register window
%sp + 28	g register window
%sp + 32	g register window
%sp + 36	g register window
%sp + 40	g register window
%sp + 44	g register window
%sp + 48	g register window
%sp + 52	g register window
%sp + 56	g register window
%sp + 60	g register window
%sp + 64	g structure pointer
%sp + 68	g actual parameter 1
%sp + 72	g actual parameter 2
%sp + 76	g actual parameter 3
%sp + 80	g actual parameter 4
%sp + 84	g actual parameter 5
%sp + 88	g actual parameter 6
%fp – 8	g local variable - **b2 – the number 2**
%fp – 4	g local variable - **b1 – the number 1**
%fp	f register window
%fp + 4	f register window
%fp + 8	f register window
%fp + 12	f register window
%fp + 16	f register window
%fp + 20	f register window
%fp + 24	f register window
%fp + 28	f register window
%fp + 32	f register window
%fp + 36	f register window
%fp + 40	f register window
%fp + 44	f register window
%fp + 48	f register window
%fp + 52	f register window
%fp + 56	f register window
%fp + 60	f register window
%fp + 64	f structure pointer
%fp + 68	f actual parameter 1
%fp + 72	f actual parameter 2
%fp + 76	f actual parameter 3
%fp + 80	f actual parameter 4
%fp + 84	f actual parameter 5

%fp + 88 f actual parameter 6
%fp + 92 f actual parameter 7 – **the number 6**
%fp + 96 f actual parameter 8 – **the number 7**
%fp + 100 f actual parameter 9 – **the number 8**

<u>Question 9</u>

Data Section

Offset	**Machine Code in Hexadecimal**	**Comment**
0	0x43	C .ascii “COS”
1	0x4F	O
2	0x53	S
3	UNDEFINED	.align 2
4	0x01	.byte 1
5	UNDEFINED	.skip 7
6	UNDEFINED	
7	UNDEFINED	
8	UNDEFINED	
9	UNDEFINED	
10	UNDEFINED	
11	UNDEFINED	
12	0x49	I .asciz “IS”
13	0x53	S
14	0x00	\0
15	0x47	G .ascii “GR”
16	0x52	R
17	0x00	.word 8
18	0x00	
19	0x00	
20	0x08	

Text Section

Offset	**Machine Instruction in Binary** **(comment for each opcode and operand)**
0	10 01010 000000 01000 0 00000000 01001 Format 3a: (op=2)(rd=10)(op3=0)(rs1=8)(i=0)(rs2=9)
4	10 01010 000000 01010 1 0000000010110 Format 3b: (op=2)(rd=10)(op3=0)(rs1=10)(i=1)(simm13=22)
8	01 000000000000000000000000001001 Format 1: (op=1)(disp30=9)
12	00 0 1010 010 0000000000000000000100 Format 2: (op=0)(a=0)(cond=10)(op2=2)(disp22=4)
16	00 01000 100 ?????????????????????? Format 2: (op=0)(rd=8)(op2=4)(imm22=?)
20	10 01000 000010 01000 1 ????????????? Format 3b: (op=2)(rd=8)(op3=2)(rs1=8)(i=1)(simm13=?)
24	01 ?????????????????????????????? Format 1: (op=1)(disp30=?)
28	11 00111 000000 00101 0 00000000 00000 Format 3a: (op=3)(rd=7)(op3=0)(rs1=5)(i=0)(rs2=0)
32	11 01000 000000 00101 0 00000000 00110 Format 3a: (op=3)(rd=8)(op3=0)(rs1=5)(i=0)(rs2=6)
36	11 01001 000000 00101 1 0000000000011 Format 3b: (op=3)(rd=9)(op3=0)(rs1=5)(i=1)(simm13=3)
40	10 00000 111000 11111 1 0000000001000 Format 3b: (op=2)(rd=0)(op3=56)(rs1=31)(i=1)(simm13=8)
44	10 01000 000000 01000 0 00000000 01001 Format 3a: (op=2)(rd=8)(op3=0)(rs1=8)(i=0)(rs2=9)
48	10 00000 111000 01111 1 0000000001000 Format 3b: (op=2)(rd=0)(op3=56)(rs1=15)(i=1)(simm13=8)
52	10 01000 000000 01000 0 00000000 01010 Format 3a: (op=2)(rd=8)(op3=0)(rs1=8)(i=0)(rs2=10)