NAME:

Login name:

Computer Science 217
Final Exam
May 19, 2003
7:30-10:30PM

This test is 8 questions. Put your name on every page, and write out and sign the Honor Code pledge before turning in the test.

``I pledge my honor that I have not violated the Honor Code during this examination."

Question	Score
1	
2	
3	
4	
5	
6	
7	
8	
Total	

QUESTION 1 *(8 POINTS)*

(a) Convert the following decimal numbers to Binary, Octal, and Hexadecimal.
Use only as many digits as necessary to represent each number.

	Binary	Octal	Hexadecimal
30 =			
17 =			

(b) Convert the following decimal numbers to 2's Complement Binary, 1's Complement Binary, and Sign Magnitide Binary. Write 8 bit results.

	Signed Magnitude	1's Complement	2's Complement
31 =			
-31 =			
32 =			
-32 =			

(c) Show that the addition of two 32-bit numbers in twos complement format produces the correct result in twos complement format for all positive and negative inputs. You can ignore overflows.

QUESTION 2 *(12 POINTS)*

Convert the following C program into Sparc assembly language. Include terse comments to help with grading. You will get full credit even if your code is not optimized

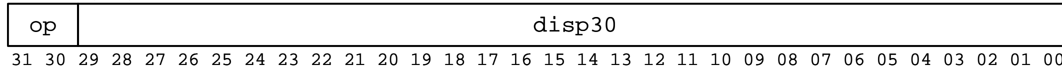
```
#include <stdio.h>

int a[4] = { 1, 2, 3, 4 };

int f (int x)
{
   if (x > 2)
     return f(x-1) + f(x-2);
   else {
     return a[x];
   }
}

int main ()
{
   printf("%d\n", f(4));
}
```

QUESTION 3 *(12 POINTS)*

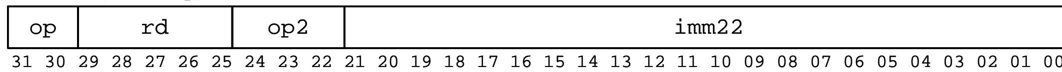
After taking CS217, Sun Microsystems hires you. Your first assignment is to modify the Sparc instruction set to handle 128 addressable registers. The instructions must still fit in 32 bits, and you must keep the same opcodes and operands for all five instruction formats (listed below). However, you are allowed to adjust the number of bits allocated for each opcode and operand in order to accommodate the increased number of addressable registers.

The Sparc instruction formats supporting 32 addressable registers are as follows:

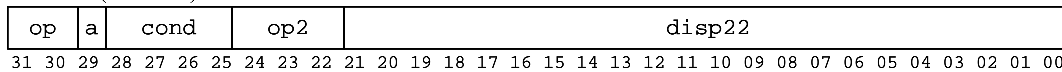
Format 1 (call)



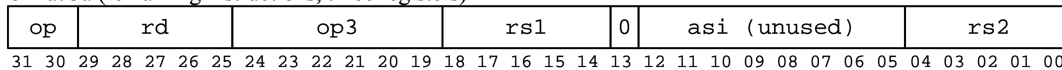
Format 2a (sethi, nop)



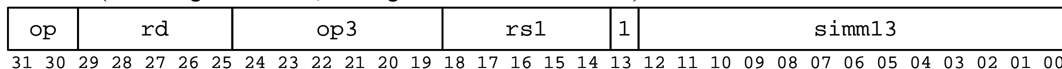
Format 2b (branches)



Format 3a (remaining instructions, three registers)

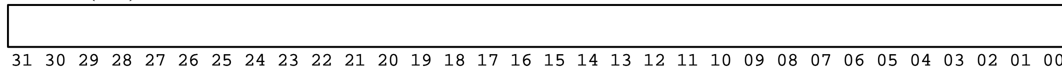


Format 3b (remaining instructions, two registers and one immediate)

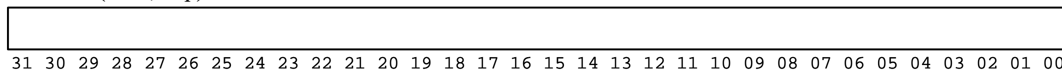


a) Draw the new allocation of bits for each of the five instruction formats in your new instruction set supporting 128 addressable registers (fill in the five boxes below with labeled opcodes and operands to form diagrams corresponding to the ones for 32 addressable registers shown above -- be careful to show how bits are allocated to each opcode and operand).

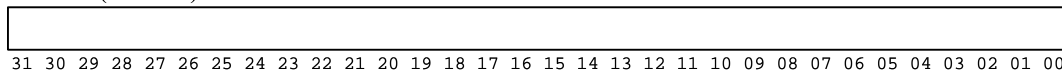
Format 1 (call)



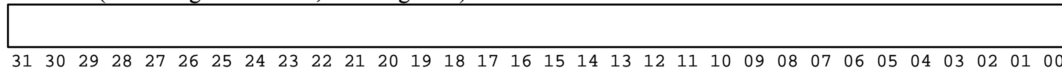
Format 2a (sethi, nop)



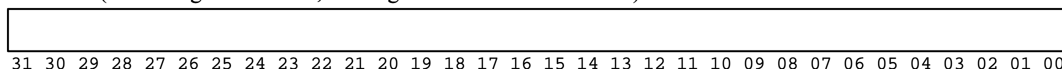
Format 2b (branches)



Format 3a (remaining instructions, three registers)



Format 3b (remaining instructions, two registers and one immediate)



QUESTION 3 *(continued)*

b) For your new instruction set, what is the range of values allowed for the constant (N) in the following instruction: **`add %g1, N, %g2`**

_______ ≤ N ≤ _______

c) For your new instruction set, how many bits are available for the target address of a `call` instruction? Please describe which memory addresses can be specified with these bits.

d) For your new instruction set, how many bits are available for the target address of a `branch` instruction? Please describe which memory addresses can be specified with these bits.

e) For your new instruction set, show the assembly code and machine language (in binary) for assigning the value `0x12345678` in hex into the 32-bit register %r33. Please do not use synthetic instructions or the %hi and %lo macros. *Hint: Use a sequence of `sethi`, `shift`, and `or` instructions.*

<u>Assembly Code</u> <u>Machine language</u>

QUESTION 4 *(8 POINTS)*

(a) Why do Sparc instructions act on values stored in registers and not directly on values stored in memory? (one sentence)

(b) Give a sequence of Sparc assembly language instructions (with operand values) that will set the overflow condition code when executed.

(c) Translate the following C language code into Sparc assembly language:

```
struct employee {
  int id;
  int code[3];
}

struct employee *e = malloc(sizeof(employee));
e->id = 4321;
e->code[0] = 1;
e->code[1] = 2;
e->code[2] = 3;
```

QUESTION 5 *(16 POINTS)*

(a) Fill in the boxes (on the following two pages) showing the text and data sections that the `as` assembler will produce when given this Sparc assembly language program. Fill any bit that cannot be determined by the assembler with a question mark (?). As an example, an appropriate answer for the first instruction of each section has been provided in the first line of each table. You may include comments if you wish to ensure partial credit, but they are not required for a correct answer.

```
          .section ".data"
          .ascii "COS"
          .skip 2
          .align 4
var1:     .word 1

          .section ".text"
          add %r8, %r9, %r10
label1:   set var2, %r8
          call doit
          bg label1
          set var1, %r17
          ret
          ld [%r10 + 10], %r16

          .align 4
doit:     call printf
          retl

          .section ".data"
          .align 2
var2:     .ascii "COT"
          .align 2
var3:     .half 0, 3, 4
          .byte 5
          .asciz "TON"
```

DATA SECTION FOR QUESTION 5a

Offset	Machine Code in Hexadecimal	Comment
0	0x43	C .ascii "COS"
1	0x4F	O
2	0x53	S
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		
25		
26		
27		
28		
29		
30		

TEXT SECTION FOR QUESTION 5a

Offset	**Machine Instruction in Binary**
0	10 01010 000000 01000 0 00000000 01001
4	
8	
12	
16	
20	
24	
28	
32	
36	
40	
44	
48	
52	
56	
60	

QUESTION 5 *(continued)*

(b) Draw the symbol table that the SPARC assembler would produce when given the SPARC assembly language program from (a). Be sure to title each column.

(c) Draw a table showing the relocation records that the SPARC assembler would produce when given the SPARC assembly language program from (a). Be sure to title each column.

QUESTION 6 *(16 POINTS)*

In this question, you will provide an implementation for the stdio functions `fopen` and `fread`. They provide an interface whereby a client can open a file and read data from it. As we discussed in class, the main feature of the stdio module is that it provides buffering of data. It relies upon an abstract data type (FILE) and system calls (e.g., `open` and `read`) to perform its functions. *Note: manual pages for fopen, fread, open, and read are provided on a page attached to the back of the exam.*

(a) How is a system call (e.g., `read`) different from a user-level call (e.g., `fread`)?
(one sentence)

(b) Why does the operating system provide a special mechanism for system calls?
(one sentence)

(c) Why does the stdio module buffer data? (one sentence)

(d) Write a plausible definition for the C structure implementing the FILE abstract data type suitable for inclusion in stdio.h.

QUESTION 6 *(continued)*

(e) Write C code implementing the `fopen` and `fread` functions that allow a client to open a file and read data with buffering. You should use your FILE data structure defined in (b) and the `open` and `read` system calls in your implementation of `fopen` and `fread`. Your `fopen` implementation can ignore the "type" argument.

QUESTION 7 *(12 POINTS)*

(a) What is virtual memory? Describe the key idea that makes it work. (one or two sentences)

(b) Name three reasons for an operating system to provide virtual memory. (list of three phrases)

(c) What is swapping? (one sentence)

(d) What sequence of steps does the operating system take when a program references memory not available in physical memory? (list of phrases)

(e) What is a working set? (one sentence)

QUESTION 7 *(continued)*

(f) What is locality of reference? (one sentence)

(g) What is thrashing? (one sentence)

(h) In the C programming language, two-dimensional arrays (a[M][N]) are stored in row-major order, which means that values within the first row (a[0][[0..N-1]) appear first, then values within the second row a[1][[0..N-1]) , etc. Based on this observation, which of the two following code fragments is likely to execute faster (circle the faster one)? Note: the two code fragments differ only in the order of the for loops.

Fragment #1:

```
extern int a[128][128];
int i, j, sum = 0;
for (i = 0; i < 128; i++)
  for (j = 0; j < 128; j++)
    sum += a[i][j];
```

Fragment #2:

```
extern int a[128][128];
int i, j, sum = 0;
for (j = 0; j < 128; j++)
  for (i = 0; i < 128; i++)
    sum += a[i][j];
```

Please explain your answer (one or two sentences):

QUESTION 8 *(16 POINTS)*

(a) Write a C program named "power" that accepts two command-line arguments (x and y) and prints x to the y power to standard output. You may assume that x and y are integers greater than 0. You need not be concerned with error checking.

Example executions:

```
--> power 2 4
16
--> power 5 3
125
--> power 12 2
144
```

Hint: you may want to use "`int atoi(char *s)`" for converting a string to an integer (e.g., `atoi("123")` returns an integer with value 123).

QUESTION 8 *(continued)*

(b) Write a C program named "power2" that reads two integers (x and y) from standard input (after prompting for each) and prints x to the y power to standard output. Your program should fork a new process that invokes the "power" program from part (a) to compute x to the y power and return the result to the main program via a pipe. Again, you may assume that x and y are integers that are greater than 0, and you need not be concerned with error checking.

Example executions:

```
--> power2
Enter x:  2
Enter y:  4
x to the y power:  16
--> power2
Enter x:  5
Enter y:  3
x to the y power:  125
--> power2
Enter x:  12
Enter y:  2
x to the y power:  144
```

##
The following are manual pages for two functions in the stdio module
##

`FILE *fopen (const char *filename, const char *type)`

`fopen` opens the file named by *filename* and associates a stream with it. `fopen` returns a pointer to the `FILE` structure associated with the stream. The *filename* argument points to a character string that contains the name of the file to be opened. *type* is a character string specifying whether the file is to be opened for reading/writing/both, which you can ignore.

`size_t fread (void *ptr, size_t size, size_t nitems, FILE *stream)`

`fread` copies, into an array pointed to by *ptr*, up to *nitems* items of data from the named input stream, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. `fread` stops reading bytes if an end-of-file or error condition is encountered while reading stream, or if *nitems* items have been read. The file pointer associated with stream is positioned following the last byte read, which may be at end-of-file. `fread` returns the number of items read.

##
The following are manual pages for two system calls
##

`int open (const char *path, int oflag)`

`open` opens a file specified in the character string *path* and returns a file descriptor that can be used to read and write data with the `read` and `write` system calls. The *oflag* argument specifies whether the file is open for reading/writing/both, which you can ignore.

`size_t read(int fildes, void *buf, size_t nbyte);`

`read` attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*. fildes is a file descriptor obtained from a `creat`, `open`, `dup`, `fcntl`, `pipe`, or `ioctl` system call. `read` returns the number of bytes read. If *nbyte* is zero, `read` returns zero and has no other results.