CPSC 313, 06w Term 1— Midterm 1

Date: October 4, 2006; Instructor: Norm Hutchinson

This is a closed book exam; no notes; you may use calculators only if you don't trust your own brain to perform simple arithmetic calculations. Answer in the space provided; use the backs of pages if needed. There are **6** questions on **5** pages, totaling **50** marks. You have **50 minutes** to complete the exam. On the last two pages you will find summaries of the x86 instructions and address modes. You may find it profitable to (carefully) remove these pages from the exam.

You should write this exam in pen - I will not consider requests to regrade solutions that are written in pencil.

NAME: ____ **SCORE:** _______ / 50

STUDENT NUMBER: __________________

_____ **1.** **(8 marks)** Short answers.

1a. Is the address of a local variable in a C function determined *statically* or *dynamically*? Briefly explain.

1b. Is the code address to which a procedure returns when it exits determined *statically* or *dynamically*? Briefly explain.

1c. Does the IA32 instruction-set architecture require that `%esp` be used as the stack pointer? Briefly explain.

1d. Give assembly-language code that computes `%eax = %eax * 9 + 7` as efficiently as possible.

_____ **2. (8 marks)** Consider the following C source file.

```
/* global variables */
int g, *gp, **gpp;

void foo (int *a1, int a2) {
    int l;
    /* consider each statement as if it were here */
}
```

Give an assembly-code implementation of each of the following statements of function `foo()`. Consider each statement in isolation (i.e., as if it were the only statement of foo). Do not assume that variables start out in registers. Be sure to write results to the appropriate location in memory. Assume that the local variable `l` is in memory (not in a register). A fully correct answer will use as few instructions as possible. **Comment your code.**

2a. `gp = &l;`

2b. `l = *a1 * a2;`

2c. `**gpp = 3;`

2d. `if (a2 > g) l = a2 else l = g;`

_____ **3.** **(8 marks)** Consider the following assembly language code.

```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %eax
    movl    12(%ebp), %edx
    cmpl    %edx, %eax
    je  L8
L6: cmpl    %edx, %eax
    jle L4
    subl    %edx, %eax
    jmp L2
L4: subl    %eax, %edx
L2: cmpl    %edx, %eax
    jne L6
L8: popl    %ebp
    ret
```

3a. Comment every line of this code carefully.

3b. Give an equivalent C-language function.

______ **4.** **(10 marks)** Consider this C procedure

```
int foo(int i)
{
    switch (i) {
        case 1:
            i = i * 2;
            break;
        case -1:
        case -4:
            i = i - 7;
            break;
        default:
            i = 0;
    }
    return i;
}
```

Give the assembly code that implements this switch statement using a jump table, in the most efficient manner possible. Include both `.text` and `.rodata` definitions. **Comment your code.**

5. **(6 marks)** Consider the procedure call to `callee()` in this C code.

```
void caller (int i, int j) {
    int l;

    l = callee (&j);
}
```

Assume that `callee` uses one callee-save register (i.e., `%esi`) and that `caller` has a value in one caller-save register (i.e., `%edx`) that must not be changed by the call to `callee`.

5a. Give the assembly code of the procedure call to `callee()`, including storing the result in local variable `l` in memory.

5b. Give the assembly code of `callee()`'s *prologue*.

_____ **6.** **(10 marks)** If you think about what we've been doing in this course so far, you may have been getting frustrated that it is all about doing again in assembly language things that you could already do in C. However, assembly code is strictly more powerful than C because you can access information that is not exposed to the C language programmer. This question hints at some of this information.

6a. Draw a picture of the runtime stack indicating three procedures, `A` which calls `B` which calls `C`. On your picture, very clearly indicate the locations of the saved frame pointers and return addresses. You should indicate the general location of parameters and local variables, but need not show them in detail. Clearly indicate exactly where in the stack the registers %esp and %ebp point.

6b. Write in assembler a function `fetch` that can be called by a function like C and fetches the program counter of C's caller and C's caller's caller. Its prototype is:

```
void fetch(int *callerspc, int *callerscallerspc);
```

Be careful to get the return address of `C`'s caller, and not `fetch`'s caller!

You may (carefully, so as to not destroy the staple) remove these last 2 pages from the exam and use them as a reference.

instruction	effect	description
leal s,d	d ← &s	load effective address
inc_ d	d ← d + 1	increment
dec_ d	d ← d - 1	decrement
neg_ d	d ← -d	negate
not_ d	d ← ~d	complement (bitwise)
add_ s,d	d ← d + s	add
sub_ s,d	d ← d - s	subtract
imul_ s,d	d ← d * s	multiply (32-bit)
xor_ s,d	d ← d ^ s	exclusive-or (bitwise)
or_ s,d	d ← d | s	or (bitwise)
and_ s,d	d ← d & s	and (bitwise)
sal_ k,d	d ← d << k	left shift
shl_ k,d	d ← d << k	left shift (same as sal_)
sar_ k,d	d ← d >> k	arithmetic right shift
shr_ k,d	d ← d >> k	logical right shift

type	gas form	operand value	addressing mode
immediate	$imm	imm	immediate
register	%r	R[r]	register
memory	imm	M[imm]	absolute
	(%r)	M[R[r]]	indirect
	imm(%r)	M[imm+R[r]]	base+displacement
	(%rb,%ri)	M[R[rb]+R[ri]]	indexed
	imm(%rb,%ri)	M[imm+R[rb]+R[ri]]	indexed
	(,%r,s)	M[R[r]*s]	scaled (by 1,2,4,8) indexed
	imm(,%r,s)	M[imm+R[r]*s]	scaled (by 1,2,4,8) indexed
	(%rb,%ri,s)	M[R[rb]+R[ri]*s]	scaled (by 1,2,4,8) indexed
	imm(%rb,%ri,s)	M[imm+R[rb]+R[ri]*s]	scaled (by 1,2,4,8) indexed

instruction		synonym	jump condition	description
jmp	label		1	direct jump
jmp	*operand		1	indirect jump
je	d	jz	zf	equal / zero
jne	d	jnz	~zf	not equal / not zero
js	d		sf	negative
jns	d		~sf	nonnegative
jg	d	jnle	~(sf ^ of) & ~zf	greater than (signed >)
jge	d	jnl	~(sf ^ of)	greater or equal (signed >=)
jl	d	jnge	sf ^ of	less than (signed <)
jle	d	jng	(sf ^ of) | zf	less or equal (signed <=)
ja	d	jnbe	~cf & ~zf	above (unsigned >)
jae	d	jnb	~cf	above or equal (unsigned >=)
jb	d	jnae	cf	below (unsigned <)
jbe	d	jna	cf | zf	below or equal (unsigned <=)