CPSC 213, Winter 2009, Term 2 — Midterm Exam

Date: March 12, 2010; Instructor: Mike Feeley

This is a closed book exam. No notes. Electronic calculators are permitted.

Answer in the space provided. Show your work; use the backs of pages if needed. There are **8** questions on **4** pages, totaling **50** marks. You have **50 minutes** to complete the exam.

STUDENT NUMBER: ____________________

NAME: ____

SIGNATURE: ____

Q1	/ 2
Q2	/ 4
Q3	/ 4
Q4	/ 11
Q5	/ 6
Q6	/ 8
Q7	/ 7
Q8	/ 8
Total	/ 50

1 **(2 marks)** **Memory Addresses.** Give an example of a memory address that is *aligned* to an four-byte boundary, but not to an eight-byte boundary.

2 **(4 marks)** **Pointer Arithmetic.** Consider the following three lines of C code. For the assignments to `i` and `j`, say (a) whether the code generates a runtime error and why or (b) what value the variables have after the code executes. If the first generates an error and the second does not, give the value of the second ignoring the first. Show your work.

```
int a[10] = { 0,1,2,3,4,5,6,7,8,9 };
int i = *(a +  ((&a[7])-a) + *(a+2));
int j = *(a + *((&a[7])-a) + *(a+2));
```

2a `i`:

2b `j`:

3 **(4 marks)** **Dynamic Allocation.** A *dangling pointer* exits when a program retains a reference to dynamically allocated memory after it has been freed/reclaimed. A *memory leak* exists when a program fails to free/reclaim dynamically allocated memory that it is no longer needed by the program.

3a Are *dangling pointers* possible in C, Java, both or either? Why or why not?

3b Are *memory leaks* possible in C, Java, both or neither? Why or why not?

4 **(11 marks)** **Global Arrays.** In C, global arrays can be declared in two ways, exemplified by `a` and `b` below.

```
int i;
int* a;
int b[10];
```

4a Indicate which of the following the compiler knows statically and which is determined dynamically.

- the address of `a`?
- the address of `b`?
- the address of `a[0]`?
- the address of `b[0]`?
- the number of bytes between `b[0]` and `b[i]`?

4b Give C code that assigns a value to `a` so that `a[0]==b[4]`, `a[1]==b[5]` and so on (don't copy the entire array, just assign a value to `a`).

4c Give SM213 assembly code that reads the value of `a[i]` into register `r0`; assume the value of `i` is in `r1`. Comment your code.

4d Give SM213 assembly code that reads the value of `b[i]` into register `r0`; assume the value of `i` is in `r1`. Comment your code.

4e Give SM213 assembly code that stores the value in `r0` in `a[i]`; assume the value of `i` is in `r1`. Comment your code.

5 **(6 marks)** **Instance Variables.** A C struct is a bit like a Java object, but without methods. Variables stored in an object or struct are called *instance variables*. Now consider this C declaration of two global variables, `a` and `b`.

```
typedef struct {
    int i, j, k;
} A;

A  a;
A* b;
```

5a Indicate which of the following the compiler knows statically and which is determined dynamically.

- the address of `a`?
- the address of `b`?
- the address of `a.k`?
- the address of `b->k`?
- the number of bytes between `b->i` and `b->k`?
- the difference between the value of `b` and the address of `b->k`?

5b Give SM213 assembly code that reads the value of `b->k` into `r0`. Comment your code.

6 **(8 marks)** **Procedures.** Consider the local variables and arguments declared in the following C code.

```
void foo (int a, int b, int c) {
    int i, j, k;
}
```

6a Indicate which of the following the compiler knows statically and which is determined dynamically.

- the address of `c`?
- the address of `k`?
- the offset to `k` from the value of the stack pointer?
- the number of bytes between `i` and `k`?

6b Give machine/assembly code that implements the procedure call `foo (1, 2, 3)`. Be sure to include every part of the procedure call statement (but just this statement). Pass the arguments on the stack.

7 **(7 marks)** **Dynamic Procedure Calls.** Procedure calls in C are normally static. Dynamic calls, like Java's polymorphic dispatch, however, have many benefits and can be implemented in C with *jump tables* (i.e., arrays of function pointers). Consider the following C code that declares (a) an array of pointers to four procedures and (b) a procedure that uses this array to invoke the ith one of them (starting with 0).

```
void procA () { printf ("A"); }
void procB () { printf ("B"); }
void procC () { printf ("C"); }
void procD () { printf ("D"); }
void (*proc[4])() = { procA, procB, procC, procD };
int i;
void foo () {
  proc[i] ();
}
```

7a Indicate which of these the compiler knows statically and which is determined dynamically.

- the address of proc[2]?
- the value of proc[2]?
- the address of the procedure that `foo` calls?

7b Give SM213 assembly code that implements the procedure call "`proc[i]`" using as few instructions as possible. Comment your code.

8 **(8 marks)** **Reading Assembly Code.** Consider the following snippet of SM213 assembly code.

```
    ld   $0, r0         #
    ld   0(r5), r1      #
    ld   4(r5), r2      #
L0: bgt  r2, L1         #
    br   L9             #
L1: ld   (r1), r3       #
    shl  $31, r3        #
    beq  r3, L2         #
    inc  r0             #
L2: inca r1             #
    dec  r2             #
    br   L0             #
L9: j    *(r6)          #
```

8a Carefully comment every line of code above.

8b The code implements a simple function. What is it? Give the simplest, plain English description you can.

OpCode	Format	Semantics	Eg Machine	Eg Assembly
load immediate	`0d--` `vvvvvvvv`	$r[d] \leftarrow v$	`0100` `00000100`	`ld $0x1000,r1`
load base	`1osd`	$r[d] \leftarrow m[o \times 4 + r[s]]$	`1123`	`ld 4(r2),r3`
load indexed	`2sid`	$r[d] \leftarrow m[r[i] \times 4 + r[s]]$	`2123`	`ld (r1,r2,4),r3`
store base+dis	`3sod`	$m[o \times 4 + r[d]] \leftarrow r[s]$	`3123`	`st r1,8(r3)`
store indexed	`4sdi`	$m[r[i] \times 4 + r[d]] \leftarrow r[s]$	`4123`	`st r1,(r2,r3,4)`
halt	`f000`		`f000`	`halt`
nop	`ff00`		`ff00`	`do nothing (nop)`
rr move	`60sd`	$r[d] \leftarrow r[s]$	`6012`	`mov r1, r2`
add	`61sd`	$r[d] \leftarrow r[d] + r[s]$	`6112`	`add r1, r2`
and	`62sd`	$r[d] \leftarrow r[d] \ \& \ r[s]$	`6212`	`and r1, r2`
inc	`63-d`	$r[d] \leftarrow r[d] + 1$	`6301`	`inc r1`
inc addr	`64-d`	$r[d] \leftarrow r[d] + 4$	`6401`	`inca r1`
dec	`65-d`	$r[d] \leftarrow r[d] - 1$	`6501`	`dec r1`
dec addr	`66-d`	$r[d] \leftarrow r[d] - 4$	`6601`	`deca r1`
not	`67-d`	$r[d] \leftarrow !r[d]$	`6701`	`not r1`
shift	`7dss`	$r[d] \leftarrow r[d] << s$	`7102` `71fe`	`shl $2, r1` `shr $2, r1`
branch	`8-oo`	$\text{pc} \leftarrow \text{pc} + 2 \times o$	`1000: 8004`	`br 0x1008`
branch if equal	`9roo`	if $r[r] == 0,\ \text{pc} \leftarrow \text{pc} + 2 \times o$	`1000: 9104`	`beq r1, 0x1008`
branch if greater	`aroo`	if $r[r] > 0,\ \text{pc} \leftarrow \text{pc} + 2 \times o$	`1000: a104`	`bgt r1, 0x1008`
jump	`b---` `aaaaaaaa`	$\text{pc} \leftarrow a$	`b000` `00001000`	`jmp 0x1000`
get program counter	`6f-d`	$r[d] \leftarrow \text{pc}$	`6f01`	`gpc r1`
jump indirect	`croo`	$\text{pc} \leftarrow r[r] + 2 \times o$	`c102`	`jmp 8(r1)`
jump double ind, b+disp	`droo`	$\text{pc} \leftarrow m[4 \times o + r[r]]$	`d102`	`jmp *8(r1)`
jump double ind, index	`eri-`	$\text{pc} \leftarrow m[4 \times r[i] + r[r]]$	`e120`	`jmp *(r1,r2,4)`