CPSC 213, Winter 2010, Term 1 — Midterm Exam

Date: October 27, 2010; Instructor: Tamara Munzner

This is a closed book exam. No notes. Electronic calculators are permitted. You may detach the last page with SM213 assembly/machine formats for easier reference. You may fill in this front page while waiting, but do not open this booklet until you are told to do so.

Answer in the space provided. Show your work; use the backs of pages if needed. There are **7** questions on **8** pages, totaling **48** marks. You have **50 minutes** to complete the exam.

STUDENT NUMBER: _____

NAME: _____

SIGNATURE: _____

LAB DAY/TIME: _____

Q1	/ 2
Q2	/ 8
Q3	/ 5
Q4	/ 12
Q5	/ 5
Q6	/ 12
Q7	/ 4
Total	/ 48

1 (2 marks) **Memory Alignment.** Consider the memory address 0x92. List all power-of-two sizes for which aligned memory access is possible and carefully justify your answer.

2 (8 marks) **Pointer Arithmetic.** Consider the following lines of C code. For the assignments to `i` , `j`, `k`, and `m` say (a) whether the code generates an error and why or (b) what value the variables have after the code executes. If one line generates an error but a later one does not, give the value of the later ones. Show your work.

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

2a `i`:

2b `j`:

2c `k`:

2d `m`:

3 (5 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. Answer true or false to the following questions:

- Dangling pointers can occur in C.
- Dangling pointers can occur in Java.
- Memory leaks can occur in C.
- Memory leaks can occur in Java.
- Garbage collection is a partial solution to the dangling pointer problem.
- Garbage collection is a partial solution to the memory leak problem.
- Garbage collection is a full solution to the dangling pointer problem.
- Garbage collection is a full solution to the memory leak problem.
- The stack is a partial solution to the dangling pointer problem.
- Having memory allocation and deallocation in separate functions is a partial solution to the dangling pointer problem.

4 (12 marks) Global Arrays and Writing Assembly Code. In the context of the following C declarations:

```
int *b;
int a[10];
int i = 3;
a[i] = 2;
b = &a[5];
b[i] = 4;
```

Provide the SM213 assembly code for the C code above, with comments. Be as concise as possible. You may assume labels `$i`, `$a`, `$b` have been created pointing to appropriate memory locations for storage, so there is no need to write any assembly for the first two lines of C code.

5 **(5 marks)** **Instance Variables.** In the context of the following C declarations:

```
struct S {
    int i[3];
    int j[4];
    int k;
};
struct S  a;
struct S* b;
```

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

- `&(a.i[2])`
- `&(b->j[2])`
- `&(b->k)`
- `(&(b->j[1]) - &(b->j[2]))`

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

6 (12 marks) Procedures and Writing Assembly Code. Consider the following procedure in SM213 assembly code. Assume that arguments have been passed in on the stack, not in registers.

```
int sum (int *a, int i, int* b, int j) {
    return a[i] + b[j];
}
```

6a Why would we use the stack to store arguments instead of just using registers? Explain carefully.

6b What is known statically vs dynamically?

- The offset between the stack pointer for sum and the address of a?
- The offset between the stack pointer for sum and the address of i?
- The address of a?
- The address of i?

6c Implement the C procedure above in SM213 assembly code. Assume that arguments have been passed in on the stack, not in registers. Comment your code.

7 (4 marks) **Static Control Flow.** Consider the following procedure in C.

```
if (a > 2) b = 3;
else if (a < -4) b = 5;
```

In SM213 assembly, how many branch/jump statements are needed to implement this code? How many of these are conditional and how many are unconditional? Justify your answer.

SM213 machine language instructions are 2 bytes or 6 bytes. The first 2 bytes are split into 4 hex digits of 4 bits each for the opcode and the three operands: OpCode, Op0, Op1, Op2. There are 16 possible opcodes, numbered '0' through 'e' in hex. The meaning and use of the three operands is different for each opcode, and is given in the table using these mnemonics: for registers 0-7, 's' for source, 'd' for destination, 'i' for index. The mnemonic 'o' for offset represents an actual number (not a register). Sometimes two hex digits are used to encode this number, sometimes just one. In assembly, 'p' is a multiple of 'o': $o * 2$ or $o * 4$. The placeholder '-' means the hex digit is ignored.

Operation	Machine Language	Semantics/RTL	Assembly
load immediate	`0d-- aaaaaaaa`	$r[d] \leftarrow v$	`ld aaaaaaaa,r1`
load base+dis	`1osd`	$r[d] \leftarrow m[o \times 4 + r[s]]$	`ld p(rs),rd`
load indexed	`2sid`	$r[d] \leftarrow m[r[i] \times 4 + r[s]]$	`ld (rs,ri,4),rd`
store base+dis	`3sod`	$m[o \times 4 + r[d]] \leftarrow r[s]$	`st rs,p(rd)`
store indexed	`4sdi`	$m[r[i] \times 4 + r[d]] \leftarrow r[s]$	`st rs,(rd,ri,4)`
halt	`f000`	(stop execution)	`halt`
nop	`ff00`	(do nothing)	`nop`
rr move	`60sd`	$r[d] \leftarrow r[s]$	`mov rs, rd`
add	`61sd`	$r[d] \leftarrow r[d] + r[s]$	`add rs, rd`
and	`62sd`	$r[d] \leftarrow r[d]$ & $r[s]$	`and rs, rd`
inc	`63-d`	$r[d] \leftarrow r[d] + 1$	`inc rd`
inc addr	`64-d`	$r[d] \leftarrow r[d] + 4$	`inca rd`
dec	`65-d`	$r[d] \leftarrow r[d] - 1$	`dec rd`
dec addr	`66-d`	$r[d] \leftarrow r[d] - 4$	`deca rd`
not	`67-d`	$r[d] \leftarrow !r[d]$	`not rd`
shift	`7doo`	$r[d] \leftarrow r[d] << oo$	`shl oo, rd`
		(if oo is negative)	`shr -oo, rd`
branch	`8-oo`	$\text{pc} \leftarrow \text{pc} + 2 \times o$	`br oo`
branch if equal	`9doo`	if $r[r] == 0$, $\text{pc} \leftarrow \text{pc} + 2 \times o$	`beq rd, oo`
branch if greater	`adoo`	if $r[r] > 0$, $\text{pc} \leftarrow \text{pc} + 2 \times o$	`bgt rd, oo`
jump	`b--- aaaaaaaa`	$\text{pc} \leftarrow a$	`jmp aaaaaaaa`
get program counter	`6f-d`	$r[d] \leftarrow \text{pc}$	`gpc rd`
jump indirect	`cdoo`	$\text{pc} \leftarrow r[r] + 2 \times o$	`jmp pp(rd)`
jump double ind, b+disp	`ddoo`	$\text{pc} \leftarrow m[4 \times o + r[r]]$	`jmp *pp(rd)`
jump double ind, index	`edi-`	$\text{pc} \leftarrow m[4 \times r[i] + r[r]]$	`jmp *(rd,ri,4)`

Operation	Machine Language Example	Assembly Example
load immediate	`0100 00001000`	`ld $0x1000,r1`
load base+dis	`1123`	`ld 4(r2),r3`
load indexed	`2123`	`ld (r1,r2,4),r3`
store base+dis	`3123`	`st r1,8(r3)`
store indexed	`4123`	`st r1,(r2,r3,4)`
halt	`f000`	`halt`
nop	`ff00`	`do nothing (nop)`
rr move	`6012`	`mov r1, r2`
add	`6112`	`add r1, r2`
and	`6212`	`and r1, r2`
inc	`6301`	`inc r1`
inc addr	`6401`	`inca r1`
dec	`6501`	`dec r1`
dec addr	`6601`	`deca r1`
not	`6701`	`not r1`
shift	`7102`	`shl $2, r1`
	`71fe`	`shr $2, r1`
branch	`1000:  8004`	`br 0x1008`
branch if equal	`1000:  9104`	`beq r1, 0x1008`
branch if greater	`1000:  a104`	`bgt r1, 0x1008`
jump	`b000 00001000`	`jmp 0x1000`
get program counter	`6f01`	`gpc r1`
jump indirect	`c102`	`jmp 8(r1)`
jump double ind, b+disp	`d102`	`jmp *8(r1)`
jump double ind, index	`e120`	`jmp *(r1,r2,4)`