CPSC 213, Winter 2014, Term 1 — Midterm

Date: October 24, 2014; Instructor: Mike Feeley

This is a closed book exam. No notes. No electronic calculators.

Answer in the space provided. Show your work; use the backs of pages if needed. There are **9** questions on **8** pages, totaling **50** marks. You have **50 minutes** to complete the exam.

STUDENT NUMBER: ____________________

NAME: ____

SIGNATURE: ____

Q1	/ 7
Q2	/ 7
Q3	/ 6
Q4	/ 3
Q5	/ 6
Q6	/ 8
Q7	/ 7
Q8	/ 3
Q9	/ 3
Total	/ 50

1 **(7 marks)** **Variables and Memory.** Consider the following C code with three global variables, `a`, `b`, and `c`, that are stored at addresses `0x1000`, `0x2000`, `0x3000`, respectively, and a procedure `foo()` that accesses them.

```
int a[1];   // at address 0x1000
int b[1];   // at address 0x2000
int* c;     // at address 0x3000

void foo() {
    a[0] = 1;
    b[0] = 2;
    c = a;
    c[0] = 3;
    c = b;
    *c = 4;
}
```

Describe what you know about the content of memory following the execution of `foo()` on a 32-bit **Little Endian** processor. List only memory locations whose address and value you know. **List each byte of memory separately** using the form "`byte_address: byte_value`". List all numbers in hex.

2 **(7 marks)** **C Pointers.** Consider the following C code.

```
int  a[10] = {0,1,2,3,4,5,6,7,8,9};  // i.e., a[i] = i
int* b     = a+4;

int foo (int* x, int* y, int* z) {
    *x = *x + *y;
    *x = *x + *z;

    return *x;
}

int bar () {
    return foo (b - 2, a + (b - a) + (&a[7] - &a[6]), a + 2);
}
```

What value does `bar()` return? Justify your answer (1) by simplifying the description of the arguments to `foo()` as much as possible so that the relationship among them, if any, is clear and (2) by carefully explaining what happens when `foo()` executes.

3 **(6 marks)** **Global Arrays**. Consider the following C global variable declarations.

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

Give the SM213 assembly code the compiler might generate for the following statements that access these variables. You may use labels `a`, `b`, and `c` for addresses. You may not assume anything about the value of registers. **Comment every line.**

3a `b = a;`

3b `a[i] = i;`

4 **(3 marks)** **Instance Variables**. Consider the following C global variable declarations.

```
struct S {
    int   a;
    void* b;
    int   c;
};

struct S* s;
```

Give the SM213 assembly code the compiler might generate for the statement:

```
s->b = &s->c;
```

You may use the label `s`. You may not assume anything about the value of registers. **Comment every line.**

5 **(6 marks)** **Count Memory References.** Consider the following C global variable declarations.

```
struct S {
    int a[10];
};

struct S s;
```

```
struct T {
    int* x;
};

struct T* t;
```

For each question, count the number of memory **reads and writes** occur when the statement executes. Do not count the memory reads that fetch instructions. Justify your answer carefully by describing the reads and writes that occur.

5a `s.a[2] = s.a[3];`

5b `t->x[2] = t->x[3];`

6 **(8 marks)** **Loops and If.** The following assembly code computes `s = a[0]` where `a` is a global, static array of integers. Modify this code so that it computes the sum of all positive elements of the array where the size of the array is stored in a global int named `n`. Your solution should avoid unnecessary memory accesses where possible (e.g., inside of the loop). You may modify the code in place. Comment every line you add. Hint: notice that you have to add four things: (1) read the value of `n`, (2) turn part of this code into a loop, (3) exit the loop at the right time, and (4) only sum positive numbers; you might want to take these one at a time.

```
ld $a, r0               # r0 = &a = &[0]

ld $0, r1               # r1 = temp_i = 0

ld $0, r2               # r2 = temp_s = 0

ld (r0, r1, 4), r3      # r3 = a[temp_i]

add r3, r2              # temp_s = temp_s + a[temp_i]

ld $s, r4               # r4 = &s

st r2, (r4)             # s = temp_s
```

7 **(7 marks)** **Procedure Calls** Implement the following C code in assembly. Pass arguments on the stack. Assume that `r5` has already been initialized as the stack pointer and assume that some other procedure (not shown) calls `doit()`. You do not have to show the allocation of `x`; just use the label `x` to refer to its address. Comment every line.

```
int x;

void doit () {
    x = addOne (5);
}

int addOne (int a) {
    return a + 1;
}
```

8 **(3 marks)** **Programming in C.** Consider the following C code.

```
int* b;

void set (int i) {
    b [i] = i;
}
```

There is a dangerous bug in this code. Carefully describe what it is. Assume that `b` was assigned a value somewhere else in the program.

9 **(3 marks)** **Programming in C.** Consider the following C code.

```
int* one () {
    int loc = 1;
    return &loc;
}

void two () {
    int zot = 2;
}
```

```
void three () {
    int* ret = one();
    two();
}
```

There is a dangerous bug in this code. Carefully describe what it is.

Hint: what is the value of `*ret` just before and just after `two()` is called? Look carefully at the implementation of `one()`, what it returns, and when variables are allocated and deallocated.

You may remove this page. These two tables describe the SM213 ISA. The first gives a template for instruction machine and assembly language and describes instruction semantics. It uses 's' and 'd' to refer to source and destination register numbers and 'p' and 'i' to refer to compressed-offset and index values. Each character of the machine template corresponds to a 4-bit, hexit. Offsets in assembly use 'o' while machine code stores this as 'p' such that 'o' is either 2 or 4 times 'p' as indicated in the semantics column. The second table gives an example of each instruction.

Operation	Machine Language	Semantics / RTL	Assembly
load immediate	`0d-- vvvvvvvv`	$r[d] \leftarrow vvvvvvvv$	`ld $vvvvvvvv,rd`
load base+offset	`1psd`	$r[d] \leftarrow m[(o = p \times 4) + r[s]]$	`ld o(rs),rd`
load indexed	`2bid`	$r[d] \leftarrow m[r[b] + r[i] \times 4]$	`ld (rb,ri,4),rd`
store base+offset	`3spd`	$m[(o = p \times 4) + r[d]] \leftarrow r[s]$	`st rs,o(rd)`
store indexed	`4sdi`	$m[r[b] + r[i] \times 4] \leftarrow r[s]$	`st rs,(rb,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	`7dss`	$r[d] \leftarrow r[d] << ss$	`shl ss, rd`
		(if ss is negative)	`shr -ss, rd`
branch	`8-pp`	$pc \leftarrow (aaaaaaaa = pc + pp \times 2)$	`br aaaaaaaa`
branch if equal	`9rpp`	if $r[r] == 0 :\ pc \leftarrow (aaaaaaaa = pc + pp \times 2)$	`beq rr, aaaaaaaa`
branch if greater	`Arpp`	if $r[r] > 0 :\ pc \leftarrow (aaaaaaaa = pc + pp \times 2)$	`bgt rr, aaaaaaaa`
jump	`B--- aaaaaaaa`	pc $\leftarrow aaaaaaaa$	`j aaaaaaaa`
get program counter	`6Fpd`	$r[d] \leftarrow$ pc $+ (o = 2 \times p)$	`gpc $o, rd`
jump indirect	`Cdpp`	$pc \leftarrow r[d] + (o = 2 \times pp)$	`j o(rd)`
jump double ind, b+off	`Cdpp`	$pc \leftarrow m[(o = 4 \times pp) + r[d]]$	`j *o(rd)`
jump double ind, index	`Edi-`	$pc \leftarrow m[4 \times r[i] + r[d]]$	`j *(rd,ri,4)`

Operation	Machine Language Example	Assembly Language Example
load immediate	`0100 00001000`	`ld $0x1000,r1`
load base+offset	`1123`	`ld 4(r2),r3`
load indexed	`2123`	`ld (r1,r2,4),r3`
store base+offset	`3123`	`st r1,8(r3)`
store indexed	`4123`	`st r1,(r2,r3,4)`
halt	`f000`	`halt`
nop	`ff00`	`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:  8003`	`br 0x1008`
branch if equal	`1000:  9103`	`beq r1, 0x1008`
branch if greater	`1000:  a103`	`bgt r1, 0x1008`
jump	`b000 00001000`	`j 0x1000`
get program counter	`6f31`	`gpc $6, r1`
jump indirect	`c104`	`j 8(r1)`
jump double ind, b+off	`d102`	`j *8(r1)`
jump double ind, index	`e120`	`j *(r1,r2,4)`