CPSC 213, Winter 2013, Term 2 — Final Exam

Date: April 14, 2014; Instructor: Mike Feeley

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

Answer in the space provided. Show your work; use the backs of pages if needed. There are **12** questions on **12** pages, totaling **105** marks. You have **2.5 hours** to complete the exam.

STUDENT NUMBER: _____

NAME: _____

SIGNATURE: _____

Q1	/ 5
Q2	/ 8
Q3	/ 8
Q4	/ 10
Q5	/ 12
Q6	/ 3
Q7	/ 6
Q8	/ 12
Q9	/ 6
Q10	/ 9
Q11	/ 10
Q12	/ 16
Total	/ 105

1 (5 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.

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

```
int a[1];
int b[1];
int* c;
```

Describe what you know about the content of memory following the execution of `foo()` on a 32-bit **Big 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 **(8 marks)** **Global Variables.** Consider the following C declaration of global variables.

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

Recalling that `&` is the C *get address* operator, which of the following can be computed statically? Justify your answers.

2a `&b`

2b `&b[4]`

2c `&c[4]`

Now answer this question.

2f Give the assembly code the compiler would generate for "`c[a] = *b;`". Use labels for static values.

3 **(8 marks)** **Instance Variables and Local Variables.** Consider the following C declaration of global variables.

```
struct X {
    int a;
    int b;
};
struct X c;
struct X* d;
```

Which of the following can be computed statically? Justify your answers.

3a `&c.a`

3b `&d->a`

3c `(&d->a) - (&d->b)`

Now answer this question.

3d Give the assembly code the compiler would generate for "`d->b = c.b;`".

4 (10 marks) **Write Assembly Code.** Give the assembly code the compiler would generate to implement the following C procedure, assuming that arguments are passed on the stack. Just this procedure. Use labels for static values. Comment every line of your code.

```
int computeSum (int* a) {
    int sum=0;
    while (*a>0) {
        sum = add (sum, *a);
        a++;
    }
    return sum;
}
```

5 **(12 marks)** **Read Assembly Code.** Consider the following SM213 code.

```
X:  deca  r5              #
    deca  r5              #
    st    r6, 4(r5)       #
    ld    $0, r1          #
    st    r1, 0(r5)       #
    ld    12(r5), r2      #
    ld    16(r5), r3      #
    not   r3              #
    inc   r3              #
L0: mov   r1, r4          #
    add   r3, r4          #
    beq   r4, L2          #
    bgt   r4, L2          #
    ld    (r2,r1,4), r4   #
    deca  r5              #
    st    r4, 0(r5)       #
    gpc   $2, r6          #
    j     *12(r5)         #
    inca  r5              #
    ld    $1, r4          #
    and   r0, r4          #
    beq   r4, L1          #
    ld    0(r5), r4       #
    add   r0, r4          #
    st    r4, 0(r5)       #
L1: inc   r1              #
    br    L0              #
L2: ld    0(r5), r0       #
    ld    4(r5), r6       #
    inca  r5              #
    inca  r5              #
    j     (r6)            #
```

5a Add a comment to every line of code. Where possible use variables names and C pseudo code in your comments to clarify the connection between the assembly code and corresponding C statements.

5b Give an equivalent C procedure (i.e., a procedure that may have compiled to this assembly code).

6 (3 marks) **Programming in C.** Consider the following C code.

```
int* b;

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

Is there a bug in this code? If so, carefully describe what it is.

7 (6 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();
}
```

7a Is there a bug in this code? If so, carefully describe what it is.

7b What is the value of "`*ret`" at the end of `three`? Explain carefully.

8 (12 marks) Static and Dynamic Procedure Calls.

8a Procedure calls in C are normally static. Method invocations in Java are normally dynamic. Carefully explain the reason why Java uses dynamic method invocation and what benefit this provides to Java programs.

8b Carefully explain an important disadvantage of dynamic invocation in Java or other languages.

8c Demonstrate the use of function pointers in C by writing a procedure called `compute` that:

1. has three arguments: a non-empty array of integers, the size of the array, and a function pointer;
2. computes either the array min or max depending only on the value of the function pointer argument;
3. contains a `for` loop, no `if` statements, and one procedure call (per loop).

Give the C code for `compute`, the two procedures that it uses (i.e., that are passed to it as the value of the function-pointer argument), and two calls to `compute`, one that computes min and the other that computes max (be sure to indicate which is which).

9 **(6 marks)** **Switch Statements.** There are two ways to implement `switch` statements in machine code. For purposes of this question, lets call them A and B.

9a Describe A, very briefly.

9b Describe B, very briefly.

9c State precisely one situation where A would be preferred over B and why.

9d State precisely one situation where B would be preferred over A and why.

10 **(9 marks)** **IO Devices.** Three key hardware features used to incorporate IO Devices with the CPU and memory are Programmed IO (PIO), Direct Memory Access (DMA) and interrupts.

10a Carefully explain the difference between PIO and DMA; give one advantage of DMA.

10b Demonstrate why interrupts are needed by carefully explaining what programs would have to do differently to perform IO if interrupts didn't exist and what disadvantages this approach would have.

10c Explain how interrupts would be added to the Simple Machine simulator by indicating where the interrupt-handling logic would be added and saying roughly what it would do.

11 (10 marks) Threads.

11a Threads can be used to manage the asynchrony inherent in I/O-device access (e.g., reading from disk). Carefully explain how threads help.

11b Carefully describe in plain English the sequence of steps a user-level thread system such as *uthreads* follows to switch from one thread to another. Ensure that your answer explains the role of the *ready queue* and explains how the hardware switches from running one thread to the other.

11c What is the role of the *thread scheduler*?

11d Explain priority-based, round-robin scheduling.

11e Explain what else is needed to ensure that threads of equal priority get an equal share of the CPU?

12 (16 marks) Synchronization

12a Explain the difference between busy-waiting and blocking. Give one advantage of blocking.

12b Consider the following program in which `inc` and `dec` can run concurrently.

```
spinlock_t s;
int        c;
void dec() {
    int success = 0;
    while (success==0) {
        while (c==0) {}
        spinlock_lock (s);
        if (c>0) {
            c = c - 1;
            success = 1;
        }
        spinlock_unlock (s);
    }
}
void inc() {
    spinlock_lock (s);
    c = c + 1;
    spinlock_unlock (s);
}
```

Re-implement the program to eliminate all busy waiting using *monitors* and *condition variables*. You may make the changes in place above or re-write some or all of the code below.

12c Assume that monitors are implemented in such a way that a thread inside of a monitor is permitted to re-enter that monitor repeatedly without blocking (e.g., when `bar` calls `zot`, which calls `foo`, `foo` is permitted to enter monitor `x`). Indicate whether the following procedures could cause deadlock in multi-threaded program that contained them (and other procedures as well). Explain why or why not. If they could, say whether you could eliminate this deadlock by only adding additional monitors or additional monitor enter or exits (you may not remove monitors). If so, show how.

```
void foo () {
    monitor_enter (x);
    monitor_exit (x);
}
void bar () {
    monitor_enter (x);
    zot ();
    monitor_exit (x);
}
void zot () {
    monitor_enter (y);
    foo ();
    monitor_exit (y);
}
```

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,r1`
load base+dis	`1psd`	$r[d] \leftarrow m[(o = p \times 4) + r[s]]$	`ld o(rs),rd`
load indexed	`2sid`	$r[d] \leftarrow m[r[s] + r[i] \times 4]$	`ld (rs,ri,4),rd`
store base+dis	`3spd`	$m[(o = p \times 4) + r[d]] \leftarrow r[s]$	`st rs,o(rd)`
store indexed	`4sdi`	$m[r[d] + r[i] \times 4] \leftarrow r[s]$	`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-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`	$pc \leftarrow r[r] + (o = 2 \times pp)$	`j o(rd)`
jump double ind, b+disp	`ddpp`	$pc \leftarrow m[(o = 4 \times pp) + r[r]]$	`j *o(rd)`
jump double ind, index	`edi-`	$pc \leftarrow m[4 \times r[i] + r[r]]$	`j *(rd,ri,4)`

Operation	Machine Language Example	Assembly Language 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`	`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+disp	`d102`	`j *8(r1)`
jump double ind, index	`e120`	`j *(r1,r2,4)`