CPSC 213, Winter 2014, Term 1 — Some More Sample Midterm Questions

Date: October 2014; Instructor: Mike Feeley

This was a closed book exam. No notes. No electronic calculators. This sample combines questions from multiple exams and so a real exam will have fewer total questions.

Answer in the space provided. Show your work; use the backs of pages if needed. There are **15** questions on **8** pages, totaling **79** marks. You have **50 minutes** to complete the exam.

STUDENT NUMBER: _____

NAME: _____

SIGNATURE: _____

Q1	/ 2
Q2	/ 4
Q3	/ 4
Q4	/ 6
Q5	/ 6
Q6	/ 3
Q7	/ 8
Q8	/ 10
Q9	/ 5
Q10	/ 3
Q11	/ 3
Q12	/ 3
Q13	/ 3
Q14	/ 9
Q15	/ 10
Total	/ 79

1 (2 marks) **Memory Alignment.** The memory address 0x2018 is aligned for certain byte size accesses but not to others. List all power-of-two sizes for which it is aligned and carefully justify your answer.

2 (4 marks) **Pointer Arithmetic.** Without using the `[]` array syntax (i.e., using only pointer arithmetic) and without introducing **any** additional variables, finish implementing this function that copies the first `n` integers of array `from` into array `to`.

```
void copy (int* from, int* to, int n) {
```

```
}
```

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 Carefully explain the most serious symptom of a dangling-pointer bug.

3b Carefully explain the most serious symptom of a memory-leak bug.

3c Does Java's garbage collector completely eliminate the dangling-pointer bug? Justify your answer carefully.

3d Does Java's garbage collector completely eliminate the memory-leak bug? Justify your answer carefully.

4 **(6 marks)** **Global Arrays.** In the context of the following C declarations:

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

Indicate what, if anything, about the following expressions can not be computed statically. Be precise.

4a `a[3]`

4b `&a[3]`

4c `b[3]`

5 **(6 marks)** **Instance Variables.** In the context of the following C declarations:

```
struct S {                              struct S  a;
    int i,j;                            struct S* b;
};
```

Indicate what, if anything, about the following expressions can not be computed statically. Be precise.

5a `&a.i`

5b `&b->i`

5c `(&b->j)-(&b->i)`

6 **(3 marks)** **Memory Endianness** Examine this C code.

```
char a[4];
*((int*) (&a[0])) = 1;
```

Carefully explain how this code can be used to determine the *endianness* of the machine on which it runs, recalling that on a *Big Endian* machine, the high-order (most-significant) byte of a number has the lowest address.

7 **(8 marks)** Consider the following C declarations.

```
struct S {                              int       a[10];
  int       i;                          int*      b;
  int       j[10];                      int       c;
  struct S* k;                          struct S  d;
};                                      struct S* e;
```

For each of the following questions indicate the total number of memory references (i.e., distinct loads and/or stores) required to execute the listed statement. Justify your answers carefully.

7a `a[3] = 0;`

7b `a[c] = 0;`

7c `b[c] = 0;`

7d `d.i = 0;`

7e `d.j[3] = 0;`

7f `e->i = 0;`

7g `d.k->i = 0;`

7h `e->k->i = 0;`

8 (10 marks) **Reading Assembly Code.** Consider the following snippet of SM213 assembly code.

```
foo: ld $s,    r0          #
     ld 0(r0), r1          #
     ld 4(r0), r2          #
     ld 8(r0), r3          #
     ld $0,    r0          #
     not       r1          #
     inc       r1          #
L0:  bgt       r3, L1      #
     br        L9          #
L1:  ld        (r2), r4    #
     add       r1, r4      #
     beq       r4, L2      #
     br        L3          #
L2:  inc       r0          #
L3:  dec       r3          #
     inca      r2          #
     br        L0          #
L9:  j         (r6)        #
```

8a Carefully comment every line of code above.

8b Give precisely-equivalent C code.

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

9 (5 marks) **Pointers in C** Consider the follow declarations in C.

```
int  a[10] = 0,2,4,6,8,10,12,14,16,18; // a[i] = 2*i;
int* b     = &a[4];
int* c     = a+4;
```

Answer the following questions. Show your work for the last question.

9a What is the *type* of the variable `a`?

9b What is the value of `b[4]`?

9c What is the value of `c[4]`?

9d What is the value of `*(a+4)`?

9e What is the value of `b-a`?

9f What is the value of `*(&a[3] + *(a+(&a[3]-&a[2])))`?

10 (3 marks) **Mystery Variable 1** This code stores 0 in a variable.

```
ld $0, r0
st r0, 8(r5)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

11 (3 marks) **Mystery Variable 2** This code stores 0 in a variable.

```
ld $0, r0
ld $3, r1
ld $0x1000, r2
st r0, (r2, r1, 4)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

12 (3 marks) **Mystery Variable 3** This code stores 0 in a variable.

```
ld $0, r0
ld $3, r1
ld $0x1000, r2
ld (r2), r2
st r0, (r2, r1, 4)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

13 (3 marks) **Mystery Variable 4** This code stores 0 in a variable.

```
ld $0, r0
ld $0x1000, r2
ld (r2), r2
st r0, 8(r2)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

14 (9 marks) Dynamic Storage

14a Carefully explain how a C program can create a *dangling pointer* and what bad thing might happen if it does.

14b Carefully explain how a C program can create a *memory leak* and what bad thing might happen if it does.

14c Can either or both of these two problems occur in a Java program? Briefly explain.

15 (10 marks) Implement the following in SM213 assembly. You can use a register for `c` instead of a local variable. **Comment every line.**

```
int  len;
int* a;
int countNotZero () {
    int c=0;
    while (len>0) {
        len=len-1;
        if (a[len]!=0)
            c=c+1;
    }
    return c;
}
```

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)`