CPSC 213, Winter 2014, Term 1 — Sample Midterm (Winter 2013 Term 2)

Date: October 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 **7** pages, totaling **50** marks. You have **50 minutes** to complete the exam.

STUDENT NUMBER: _____

NAME: _____

SIGNATURE: _____

Q1	/ 8
Q2	/ 4
Q3	/ 4
Q4	/ 4
Q5	/ 8
Q6	/ 4
Q7	/ 4
Q8	/ 4
Q9	/ 10
Total	/ 50

1 (8 marks) **Memory and Numbers**. Consider the following C code with global variables `a` and `b`.

```
int  a[2];
int* b;

void foo() {
    b    = a;
    a[0] = 1;
    b[1] = 2;
}

void checkGlobalVariableAddressesAndSizes() {
    if ((&a==0x2000) && (&b==0x3000) && sizeof(int)==4  && sizeof(int*)==4)
        printf ("OKAY");
}
```

When `checkGlobalVariableAddressesAndSizes()` executes it prints "`OKAY`". Recall that `sizeof(t)` returns the number of bytes in variables of type `t`.

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

2 **(4 marks)** **Pointers in C**. Consider the following declaration of C global variables.

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

And the following expression that accesses them found in some procedure.

```
*(a + ((&a[9] + 5) - b))
```

When this expression is evaluated at runtime does it cause an error? If not, what value does it compute?

Briefly explain your answer as follows: if there is a runtime error, clearly explain what causes it; if there is not an error, show at least 3 lines of work with intermediate values to explain your answer, step by step.

3 **(4 marks)** **Global Arrays**. Consider the following C global variable declarations.

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

Give the SM213 assembly code the compiler might generate for the following statements that access these variables. Assume the value `0` is in `r0` and the value of `i` is in `r1`. Use labels `a` and `b` for variable addresses.

3a `a[i] = 0;`

3b `b[i] = 0;`

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

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

struct S  s0;
struct S* s1;
```

Give the SM213 assembly code the compiler might generate for the following statements that access these variables. Assume the value `0` is in `r0`. Use labels `s0` and `s1` for variable addresses.

4a `s0.c = 0;`

4b `s1->c = 0;`

5 (8 marks) Count Memory References. Consider the following C global variable declarations.

```
struct S {
    struct S* s;
    int*      a;
    int       b[10];
};

struct S s0;
```

And this statement found in some procedure.

```
int v = s0.s->a[s0.s->b[2]];
```

List every memory read that will occur when this statement executes in the SM213 machine. List only memory reads and list each read separately. For each read give the name of the variable being read and an expression that computes the target memory address. This expression may only contain the label `s0`, numbers, the name of any variable previously read and arithmetic operators such as for addition and multiplication. Numbers must be immediately followed by a description in parentheses (e.g., "...4 (size of int)"). There are more lines listed below than you need:

1. Variable:

   Address:

2. Variable:

   Address:

3. Variable:

   Address:

4. Variable:

   Address:

5. Variable:

   Address:

6. Variable:

   Address:

6 **(4 marks)** **Branch and Jump Instructions**.

6a What is one important benefit that *PC-relative* branches have over *absolute-address* jumps.

6b What is the value of the program-counter register (i.e., the PC) following the execution of this unconditional branch instruction at address `0x500`. Justify your answer.

```
0x500: 8005
```

7 **(4 marks)** **Loops**. Consider the consequences of eliminating the two conditional branch instructions from SM213 (and not adding any other instructions). Would compilers still be able to compile C programs to run on this modified machine? If so, explain how. If not, carefully explain what feature or features of C would be impossible to execute on the modified machine.

8 **(4 marks)** **Dynamic Allocation**. The following code contains a procedure that creates a copy of a null-terminated string (the standard representation for strings in C). It contains a serious bug related to dynamic memory allocation.

```
char* copy (char* s) {
  int   i, len = 0;
  char* cpy;

  while (s [len] != 0)
    len++;
  cpy = (char*) malloc (len+1);
  for (i=0, i<len; i++)
    cpy [i] = s [i];
  cpy [len] = 0;
  return cpy;
}
```

```
void doSomething () {
  char* x;
  x = copy ("Hello World");
  printf ("%s", x);
}
```

Explain in plan English what the bug is and how you would fix it (without changing the semantics of `copy`).

9 (10 marks) **Writing Assembly Code**. Write SM213 assembly code that implements the following C program. Use labels for static addresses but do not include variable label declarations (i.e. "`.long`" lines). Show only the code for these two procedures. Do not implement a return from `callReplace()`; simply halt at the end of that procedure. Do not use the stack. Comment every line.

```
int* a;
int  searchFor, replaceWith, size, i;

void replace() {
    for (i=0; i<size; i++)
        if (a[i]==searchFor)
            a[i]=replaceWith;
}
```

```
void callReplace() {
    replace();
    // halt; do not return
}
```

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)`