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

Date: October 2014; Instructor: Mike Feeley

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.

```
0x2000: 0x01
0x2001: 0x00
0x2002: 0x00
0x2003: 0x00
0x2004: 0x02
0x2005: 0x00
0x2006: 0x00
0x2007: 0x00
0x3000: 0x00
0x3001: 0x20
0x3002: 0x00
0x3003: 0x00
```

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.

It executes without error and computes the value 8.

```
*(a + ((&a[9] + 5) - b)) == *(a + (&a[14] - &a[6]))
                         == *(a + 8)
                         == a[8]
                         == 8
```

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;`

```
  ld $a, r2             # r2 = &a
  st r0, (r2, r1, 4)    # a[i] = 0
```

3b `b[i] = 0;`

```
  ld $b, r2             # r2 = &b
  ld (r2), r2           # r2 = b
  st r0, (r2, r1, 4)    # 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;`

```
  ld $s0, r2      # r2 = &s0
  st r0, 8(r2)    # s0.c = 0
```

4b `s1->c = 0;`

```
  ld $s1, r2      # r2 = &s1
  ld (r2), r2     # r2 = s1
  st r0, 8(r2)    # 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: `s0.s`
   Address: `s0 + 0 (offset to s)`
2. Variable: `s0.s->b[2]`
   Address: `s0.s + 8 (offset to b) + 2 (index) * 4 (size of int)`
3. Variable: `s0.s->a`

Address: `s0.s + 4 (offset to a)`

4. Variable: `s0.s->a[s0.s->b[2]]`

   Address: `s0.s->a + s0.s->b[2] * 4 (size of int)`

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

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

smaller instructions

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
```

`0x502 + 5 * 2 == 0x50c`

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.

If-then statements whose test condition is a dynamic value and loops that execute a bounded and dynamically determined number of times.

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`).

The `copy()` procedure allocates memory that is never freed. The simplest fix is to insert a `free(x)` statement as the last line of `doSomething()`; you get full marks for this answer. A better fix is to move the allocation to `doSomething()` and have it pass the target string to `copy()` as a parameter instead of having `copy()` return it; one bonus mark is available for a good explanation of the better solution.

You didn't have to give the code, but here is the code for the better solution.

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

  while (s [len] != 0 && len+1 < n)
    len++;
  for (i=0, i<len; i++)
    cpy [i] = s [i];
  cpy [len] = 0;
  return cpy;
}

void doSomething () {
  char x[1000];
  copy (x, "Hello World", sizeof (x));
  printf ("%s", x);
}
```

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
}
```

```
replace:      ld   $size, r0          # r0 = &size
              ld   0x0(r0), r0        # r0 = size = i
              ld   $a, r1             # r1 = &a
              ld   0x0(r1), r1        # r1 = a
              ld   $searchFor, r2     # r2 = &searchFor
              ld   0x0(r2), r2        # r2 = searchFor
              not  r2                 # r2 = !searchFor
              inc  r2                 # r2 = -searchFor
              ld   $replaceWith, r3   # r3 = &replaceWith
              ld   0x0(r3), r3        # r3 = replaceWith
loop:         beq  r0, done           # goto done if i==0
              dec  r0                 # i--
              ld   (r1, r0, 4), r4    # r4 = a[i]
              add  r2, r4             # r4 = a[i] - searchFor
              beq  r4, match          # goto match   if a[i]==searchFor
              br   nomatch            # goto nomatch if a[i]!=searchFor
match:        st   r3, (r1, r0, 4)    # a[i] = replaceWith
nomatch:      br   loop               # goto loop
done:         j    0x0(r6)            # return
callReplace:  gpc  $0x6, r6           # ra = pc + 6
              j    replace            # replace()
              halt
```