## 2. Global Variables and Arrays [9 marks]

Consider the following code where the value of the variable `i` is already loaded into `r0` and where the assembly-code labels `a`, `b`, and `c` represent the address of the variables `a`, `b`, and `c`, respectively. Treat each subquestion separately; do not use values computed in prior subquestions. No comments needed.

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

a) Give assembly code for the statement `c = *a;`

b) Give assembly code for the statement `a = b;`

c) Give assembly code for the statement `b[i] = *(a + i) + i;`

**2** **[10 marks]** **Global Variables and Arrays.** Answer the following questions about global variables.

```
int    i;
int a[5];
int   *b;
```

**2a** Give assembly code for the C statement: `i = a[2]`.

**2b** Give assembly code for the C statement: `b[0] = a[b[i]];`.

**2c** How many memory reads are required to execute the statement? Fill in a single multiple choice option below.

```
b[a[3]] = a[3] + a[i];
```

0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8 ○

**Exercise 4**
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.

**Exercise 5**
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.

**Exercise 6**
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.

**Exercise 7**
What is the value of the register `r0` after the following program executes?

```
ld $0x2004, r0  # r0 = 0x2004
ld (r0), r0     # r0 = m[r0]

.pos 0x2000
.long 0         # value at address 0x2000
.long 1         # value at address 0x2004
.long 2         # value at address 0x2008
.long 3         # value at address 0x200c
```

**2** **(6 marks)** **Global Variables.** Consider the following C declaration of global variables. Give the SM213 assembly code for each of the following C statements. Use labels such as `a`, `b` etc. for statically know values. Comments are not required. You can treat your answers as a sequence and thus use register values loaded in one subquestion in subsequent ones to avoid duplication.

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

**2a** `b = &a;`

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

**2c** `a = *c[a];`

**2 (6 marks) Static Scalars and Arrays.** Consider the following C code containing global variables `s` and `d`.

```
int  s[2];
int* d;
```

Use `r0` for the variable `i` (i.e., do not read or write memory for `i`) and use labels for static values. Give assembly code that is equivalent to each of the following C statements. Treat each question separately; i.e., do not use register values assigned in a previous answer.

**2a** `i = s[i];`

**2b** `i = d[i];`

**2c** `d = &s[10];`

**3 (6 marks) Structs and Instance Variables** Consider the following C code containing the global variable `a`.

```
struct S {
    int       x;
    int       y;
    struct S* z;
};

struct S* a;
```

Once again use `r0` for the variable `i` (i.e., do not read or write memory for `i`) and use labels for static values. Give assembly code that is equivalent to each of the following C statements. Treat each question separately; i.e., do not use register values assigned in a previous answer.

**3a** `i = a->y`

**3b** `i = a->z->y;`

**3c** `a->z->z = a;`

**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;`

**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;`

```
}
```

**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;
```

**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;`

**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.

**4 (12 marks) Global Arrays and Writing Assembly Code.** In the context of the following C declarations:

```
int *b;
int a[10];
int i = 3;
a[i] = 2;
b = &a[5];
b[i] = 4;
```

Provide the SM213 assembly code for the C code above, with comments. Be as concise as possible. You may assume labels `$i`, `$a`, `$b` have been created pointing to appropriate memory locations for storage, so there is no need to write any assembly for the first two lines of C code.

## 2. Global Variables and Arrays [9 marks]

Consider the following code where the value of the variable `i` is already loaded into `r0` and where the assembly-code labels `a`, `b`, and `c` represent the address of the variables `a`, `b`, and `c`, respectively. Treat each subquestion separately; do not use values computed in prior subquestions. No comments needed.

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

a) Give assembly code for the statement `c = *a;`

```
ld $a, r1       # r1 = &a
ld (r1), r1     # r1 = a
ld (r1), r1     # r1 = *a
ld $c, r2       # r2 = &c
st r1, (r2)     # c = *a;
```

b) Give assembly code for the statement `a = b;`

```
ld $b, r1       # r1 = b
ld $a, r2       # r2 = &a
st r1, (r2)     # a = b
```

c) Give assembly code for the statement `b[i] = *(a + i) + i;`

```
ld $a, r1            # r1 = &a
ld (r1), r1          # r1 = a
ld (r1, r0, 4), r2   # r2 = *(a + i) = a[i]
add r0, r2           # r2 = *(a + i) + i
ld $b, r3            # r3 = b
st r2, (r3, r0, 4)   # b[i] = *(a + 1) + i
```

```
int    i;
int a[5];
int   *b;
```

**2a** Give assembly code for the C statement: `i = a[2]`.

```
ld $2, r0                # r0 = 2
ld $a, r1                # r1 = &a
ld (r2, r0, 4), r1       # r1 = a[2]
ld $i, r2                # r2 = &i
st r1, (r2)              # i = a[2]
```

**2b** Give assembly code for the C statement: `b[0] = a[b[i]];`.

```
ld $i, r0                # r0 = &i
ld (r0), r0              # r0 = i
ld $b, r1                # r1 = &b
ld (r1), r1              # r1 = b
ld (r1, r0, 4), r2       # r2 = b[i]
ld $a, r3                # r3 = &a
ld (r3, r2, 4), r3       # r3 = a[b[i]]
st r3, (r1)              # b[0] = a[b[i]]
```

**2c** How many memory reads are required to execute the statement? Fill in a single multiple choice option below.

```
b[a[3]] = a[3] + a[i];
```

```
4: i, b, a[3], a[i]
```

0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8 ○

**3** **[10 marks]** **Structs and Instance Variables.** Answer the following questions about the global variables `a` and `b`, declared as follows. Assume that `int`'s and pointers are **4-bytes** long.

```
struct A {
    char x;
    int y[3];
    struct B* z;
};

struct A* a;
struct B  b;
```

```
struct B {
    struct A j;
    int k;
};
```

Treat each sub-question separately (i.e., do not use values computed in prior sub-questions). Comments are not required, but they will probably help you.

**3a** Give assembly code for the C statement: `b.k = a->z[2].k;`.

```
ld $a, r0            # r0 = &a
ld (r0), r0          # r0 = a
ld 16(r0), r0        # r0 = a->z
ld 68(r0), r0        # r0 = a->z[2].k
ld $b, r1            # r1 = &b
st r0, 20(r1)        # b.k = a->z[2].k
```

**3b** How many memory reads are required to execute the statement? Fill in a single multiple choice option below.

```
b.k = a->z->j.z->j.y[2];
```

```
4: a, a->z, a->z->j.z, a->z->j.z->j.y[2]
```

0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8 ○

**4** **[10 marks]** **Static Control Flow.** Answer the following questions on static control flow.

**Exercise 5**
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;**

```
ld $a, r0        # r0 = &a
ld $b, r1        # r1 = &b
st r0, (r1)      # b = a
```

**3b. a[i] = i;**

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

**Exercise 6**
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.

There's a potential array overflow. Need to check that i is in range (0 .. size of b - 1) before writing to b[i] and thus this size, which is dynamically determined, should be a parameter to set or a global variable.

**Exercise 7**
What is the value of the register `r0` after the following program executes?

```
ld $0x2004, r0  # r0 = 0x2004
ld (r0), r0     # r0 = m[r0]

.pos 0x2000
.long 0         # value at address 0x2000
.long 1         # value at address 0x2004
.long 2         # value at address 0x2008
.long 3         # value at address 0x200c
```

r0 = 1

# CPSC 213, Winter 2016, Term 2 — Final Exam **Solution**

Date: April 19, 2017; Instructor: Mike Feeley and Alan Wagner

**1** **(4 marks)** **Variables and Memory.** Consider the following code running on 32-bit, big-endian architecture.

```
struct S {
    char a[4];
};

int       i;
struct S* s;
```

```
char foo() {
    i = 0x12345678;
    s = (struct S*) &i;
    return s->a[1];
}
```

Does the procedure `foo()` compile and execute without error? If not, what is the error?

Yes, it compiles

If so, can you determine what value it returns?

Yes you can.

If so, what is that value?

4 marks, In big endian it is 34 and in little endian 56

**2** **(6 marks)** **Global Variables.** Consider the following C declaration of global variables. Give the SM213 assembly code for each of the following C statements. Use labels such as `a`, `b` etc. for statically know values. Comments are not required. You can treat your answers as a sequence and thus use register values loaded in one subquestion in subsequent ones to avoid duplication.

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

**2a** `b = &a;`

```
ld   $a, r0          # r0 = &a
ld   $b, r1          # r0 = &b
st   r0,(r1)         # b = &a
```

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

```
ld   $a, r0          # r0 = &a
ld   $b, r1          # r0 = &b
## new part
ld   0(r0), r2       # r1 = a
ld   0(r1),r3        # r1 = b
ld   (r3,r2,4),r4    # r4 = b[a]
st   r4,(r0)         # a = b[a]
```

**2c** `a = *c[a];`

```
ld   $a, r0          # r0 = &a
ld   0(r0), r2       # r1 = a
## new part
ld   $c,r4           # r4 = &c[0]
ld   (r4,r2,4),r5    # r5 = c[a]
ld   0(r5),r6        # r6 = *c[a]
st   r6,(r0)         # a = *c[a]
```

**3** **(6 marks)** **Instance and Local Variables.** Give the SM213 assembly code for the following statements that access elements of global and local structs. Consider each statement as if it were the last line of `foo()` at the location indicated by the comment. Use labels such as `a`, `b` etc. for statically known values. Assume that `r5` stores the value of the stack pointer. Comments are not required. You can treat your answers as a sequence and thus use register values loaded in one subquestion in subsequent ones to avoid duplication.

# CPSC 213, Winter 2015, Term 2 — Midterm Exam **Solution**

Date: February 29, 2016; Instructor: Mike Feeley

**1 (8 marks)** **Memory and Numbers.** Consider the following C code containing global variables `a`, `b`, and `c` that is executing on a *little endian*, 32-bit processor. Assume that the address of `a[0]` is `0x1000` and that the compiler allocates global variables in the order they appear in the program without *unnecessarily* wasting space between them. With this information, you can determine the value of certain bytes of memory following the execution of `foo()`.

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

```
void foo() {
    a[1] = 0x10;
    b[1] = 0x30405060;
    c    = b;
}
```

List the address and value of every memory location whose address and value you know. Use the form "`address: value`". **List every byte on a separate line** and list all numbers in hex.

```
0x1001: 0x10
0x1008: 0x60
0x1009: 0x50
0x100a: 0x40
0x100b: 0x30
0x100c: 0x04
0x100d: 0x10
0x100e: 0x00
0x100f: 0x00
```

**2 (6 marks)** **Static Scalars and Arrays.** Consider the following C code containing global variables `s` and `d`.

```
int  s[2];
int* d;
```

Use `r0` for the variable `i` (i.e., do not read or write memory for `i`) and use labels for static values. Give assembly code that is equivalent to each of the following C statements. Treat each question separately; i.e., do not use register values assigned in a previous answer.

**2a** `i = s[i];`

```
ld $s, r1             # r1 = s
ld (r1, r0, 4), r0    # i = s[i]
```

**2b** `i = d[i];`

```
ld $d, r1             # r1 = &d
ld (r1), r1           # r1 = d
ld (r1, r0, 4), r0    # i = d[i]
```

**2c** `d = &s[10];`

```
ld  $s, r1     # r1 = s
ld  $40, r2    # r2 = 10*4
add r2, r1     # r1 = &s[10]
ld  $d, r2     # r2 = &d
st  r1, (r2)   # d = &s[10]
```

**3 (6 marks)** **Structs and Instance Variables** Consider the following C code containing the global variable `a`.

```
struct S {
    int       x;
    int       y;
    struct S* z;
};
```

```
struct S* a;
```

```
b - 2                          = a + 4 - 2
                               = a + 2
a + (b - a) + (&a[7] - &a[6]) = a + ((a+4) - a) + ((a+7) - (a+6))
                               = a + 4 + 1
                               = a + 5
```

So the call to `foo` simplifies to `foo(a+2, a+5, a+2)`. Thus when `foo()` runs we have:

```
*(a+2) = *(a+2) + *(a+5);
       = 2 + 5
       = 7
*(a+2) = *(a+2) + *(a+2)
       = 7 + 7
       = 14
```

Thus `foo()` returns 14.

**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;`

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

**3b** `a[i] = i;`

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

```
ld $s, r0     # r0 = &s
ld (r0), r1   # r1 = s = &s->a
ld $8, r2     # r2 = 8
add r1, r2    # r2 = &s1->c
st r2, 4(r1)  # s1->b = &s1->c
```

**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];`

1 read: `s.a[3]`; 1 write: `s.a[2]`

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

3 reads: `t`, `t->x`, `t->x[3]`; 1 write: `t->x[2]`

**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
```

Added lines are numbered

```
            ld $a, r0              # r0 = &a = &[0]
            ld $0, r1              # r1 = temp_i = 0
            ld $0, r2              # r2 = temp_s = 0
[1]         ld $n, r5              # r5 = &n
[2]         ld (r5), r5            # r5 = n = temp_n
[3]     loop:
[4]         bgt r5, cont           # continue if temp_n > 0
[5]         br done                # exit look if temp_n <= 0
[6]     cont:
            ld (r0, r1, 4), r3     # r3 = a[temp_i]
[7]         dec r5                 # temp_n --
[8]         inc r1                 # temp_i ++
[9]         bgt r3, add            # goto add if a[temp_i] > 0
[10]        br  loop               # skip add & goto loop if a[temp_i] <= 0
[11]    add:
            add r3, r2             # temp_s += a[temp_i] if a[temp_i] < 0
[12]        br loop                # start next iteration of loop
[13]    done:
            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.

```
  b - 2                          = a + 4 - 2
                                 = a + 2
  a + (b - a) + (&a[7] - &a[6]) = a + ((a+4) - a) + ((a+7) - (a+6))
                                 = a + 4 + 1
                                 = a + 5
```

So the call to `foo` simplifies to `foo(a+2, a+5, a+2)`. Thus when `foo()` runs we have:

```
  *(a+2) = *(a+2) + *(a+5);
         = 2 + 5
         = 7
  *(a+2) = *(a+2) + *(a+2)
         = 7 + 7
         = 14
```

Thus `foo()` returns 14.

**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;`

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

**3b** `a[i] = i;`

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

```
  ld $s, r0     # r0 = &s
  ld (r0), r1   # r1 = s = &s->a
  ld $8, r2     # r2 = 8
  add r1, r2    # r2 = &s1->c
  st r2, 4(r1)  # s1->b = &s1->c
```

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

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

struct S s;
```

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

Date: October 2014; Instructor: Mike Feeley

**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.

The lower 3 bits are zero, but the 4th bit is 1 and so its aligned for 2, 4, and 8 byte access, but not for anything more than that.

**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) {
    while (n--)
        *to++ = *from++;
}
```

**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.

The memory pointed to by the dangling-pointer may be re-allocated for some other use and then the program might use the dangling-pointer to erroneously update this newly allocated thing, thinking its updated the old (but freed) thing.

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

The program could eventually exhaust memory and slow itself, and the entire system it runs on, horribly.

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

Yes. It will only free memory when it is unreachable via any pointer in the program.

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

No. Sometimes program's retain references to objects that they never intend to use again. The GC can't free these things since they are still reachable.

**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]`

The value of the variable.

**4b** `&a[3]`

Nothing.

**4c** `b[3]`

The address and value of the variable.

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

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

Date: October 2014; Instructor: Mike Feeley

**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.

The lower 3 bits are zero, but the 4th bit is 1 and so its aligned for 2, 4, and 8 byte access, but not for anything more than that.

**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) {
    while (n--)
        *to++ = *from++;
}
```

**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.

The memory pointed to by the dangling-pointer may be re-allocated for some other use and then the program might use the dangling-pointer to erroneously update this newly allocated thing, thinking its updated the old (but freed) thing.

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

The program could eventually exhaust memory and slow itself, and the entire system it runs on, horribly.

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

Yes. It will only free memory when it is unreachable via any pointer in the program.

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

No. Sometimes program's retain references to objects that they never intend to use again. The GC can't free these things since they are still reachable.

**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]`

The value of the variable.

**4b** `&a[3]`

Nothing.

**4c** `b[3]`

The address and value of the variable.

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

# 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;
```

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

Date: October 2014; Instructor: Mike Feeley

**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.

The lower 3 bits are zero, but the 4th bit is 1 and so its aligned for 2, 4, and 8 byte access, but not for anything more than that.

**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) {
    while (n--)
        *to++ = *from++;
}
```

**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.

The memory pointed to by the dangling-pointer may be re-allocated for some other use and then the program might use the dangling-pointer to erroneously update this newly allocated thing, thinking its updated the old (but freed) thing.

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

The program could eventually exhaust memory and slow itself, and the entire system it runs on, horribly.

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

Yes. It will only free memory when it is unreachable via any pointer in the program.

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

No. Sometimes program's retain references to objects that they never intend to use again. The GC can't free these things since they are still reachable.

**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]`

The value of the variable.

**4b** `&a[3]`

Nothing.

**4c** `b[3]`

The address and value of the variable.

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

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`

**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`

Statically known. The compiler decides the address of global variables.

**2b** `&b[4]`

Not statically known. The compiler doesn't know the address of an element in a dynamic array.

`&b[4] = &*(b + 4) = b + 4`. The compiler doesn't know the value of `b`

**2c** `&c[4]`

Statically known. The compiler chooses where a globally declared array is located and thus knows the address of any of its elements.

Now answer this question.

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

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

# static data area (not required in answer)
.pos 0x1000
a: .long 0
b: .long 0
c: .long 0 # c[0]
   .long 0 # c[1]
   .long 0 # c[2]
   .long 0 # c[3]
   .long 0 # c[4]
   .long 0 # c[5]
   .long 0 # c[6]
   .long 0 # c[7]
   .long 0 # c[8]
   .long 0 # c[9]
```

**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`

Address of member of a statically allocated is statically known.

- Garbage collection is a partial solution to the dangling pointer problem. False
- Garbage collection is a partial solution to the memory leak problem. True
- Garbage collection is a full solution to the dangling pointer problem. True
- Garbage collection is a full solution to the memory leak problem. False
- The stack is a partial solution to the dangling pointer problem. False
- Having memory allocation and deallocation in separate functions is a partial solution to the dangling pointer problem. False

**4** **(12 marks)** **Global Arrays and Writing Assembly Code.** In the context of the following C declarations:

```
int *b;
int a[10];
int i = 3;
a[i] = 2;
b = &a[5];
b[i] = 4;
```

Provide the SM213 assembly code for the C code above, with comments. Be as concise as possible. You may assume labels `$i`, `$a`, `$b` have been created pointing to appropriate memory locations for storage, so there is no need to write any assembly for the first two lines of C code.

```
      ld $i, r0            # r0 = &i
      ld $0x3, r1          # r1 = 3
      st r1, 0x0(r0)       # i = 3
      ld $a, r2            # r2 = &a
      ld $0x2, r3          # r3 = 2
      st r3, (r2, r1, 4)   # a[i] = 2
      ld $b, r4            # r4 = &b
      ld $20, r6           # r6 = 20 (5*4)
      add r2, r6       # r6 = &a[5]
      st r6, 0x0(r4)       # b = &a[5]
      ld $0x4, r7          # r7 = 4
      st r7, (r6, r1, 4)   # b[i] = 4
```

(12 marks, one per line. 1/2 mark off for verbose/unnecessary instructions. 1/2 mark off for manual add instead of using an indexed instruction. No penalty for leaving off the 0 offset when using base/displacement load or store.)

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

```
struct S {
    int i[3];
    int j[4];
    int k;
};
struct S  a;
struct S* b;
```

**5a** Indicate which of the following the compiler knows statically and which is determined dynamically.

- `&(a.i[2])` static
- `&(b->j[2])` dynamic
- `&(b->k)` dynamic
- `(&(b->j[1]) - &(b->j[2]))` static

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

Date: October 2014; Instructor: Mike Feeley

**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.

The lower 3 bits are zero, but the 4th bit is 1 and so its aligned for 2, 4, and 8 byte access, but not for anything more than that.

**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) {
    while (n--)
        *to++ = *from++;
}
```

**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.

The memory pointed to by the dangling-pointer may be re-allocated for some other use and then the program might use the dangling-pointer to erroneously update this newly allocated thing, thinking its updated the old (but freed) thing.

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

The program could eventually exhaust memory and slow itself, and the entire system it runs on, horribly.

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

Yes. It will only free memory when it is unreachable via any pointer in the program.

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

No. Sometimes program's retain references to objects that they never intend to use again. The GC can't free these things since they are still reachable.

**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]`

The value of the variable.

**4b** `&a[3]`

Nothing.

**4c** `b[3]`

The address and value of the variable.

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