CPSC 213, Winter 2015, Term 2 — MIDTERM SAMPLE QUESTIONS
Solution

Date: Feb 18, 2016; Instructor: Mike Feeley

1 **(7 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, and a procedure `foo()` that accesses them.

```
int a[1];    // at address 0x1000
int b[1];    // at address 0x2000
int* c;      // at address 0x3000

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

Describe what you know about the content of memory following the execution of `foo()` on a 32-bit **Little 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.

```
0x1000: 0x03
0x1001: 0x00
0x1002: 0x00
0x1003: 0x00
0x2000: 0x04
0x2001: 0x00
0x2002: 0x00
0x2003: 0x00
0x3000: 0x00
0x3001: 0x20
0x3002: 0x00
0x3003: 0x00
```

2 **(7 marks)** **C Pointers.** 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.

```
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.

```
int x;

void doit () {
    x = addOne (5);
}

int addOne (int a) {
    return a + 1;
}
```

```
doit:
    deca r5         # allocate space for ra on stack
    st r6, (r5)     # save ra on stack
    deca r5         # make room for argument on stack
    ld $5, r0       # r0 = 5
    st r0, (r5)     # arg0 = 5
    gpc $6, r6      # get return address
    j add           # call addOne (5)
    inca r5         # remove argument area
    ld $x, r1       # r1 = &x
    st r0, (r1)     # x = addOne (5)
    ld (r5), r6     # restore ra from stack
    inca r5         # remove ra space from stack
    j (r6)          # return
addOne:
    ld (r5), r0     # r0 = a
    inc r0          # r0 = a + b
    j (r6)          # return a + b
```

8 (3 marks) **Programming in C.** 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.

9 (3 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();
}
```

There is a dangerous bug in this code. Carefully describe what it is.

Hint: what is the value of `*ret` just before and just after `two()` is called? Look carefully at the implementation of `one()`, what it returns, and when variables are allocated and deallocated.

Yes; there's a dangling pointer. The procedure `one()` returns a pointer to a local variable, but that local variable is deallocated when the procedure returns. Just before `three()` calls `two()` the value of `*ret` is 1, but after calling `two()` it changes to `2` because `two()`'s local variable `zot` will be allocated in the same location as `one()`'s `loc`, and `*ret` is a dangling pointer pointing to that location.

10 (4 marks) **Branch and Jump Instructions**.

10a What is one important benefit that *PC-relative* branches have over *absolute-address* jumps.

smaller instructions

10b 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
```

11 (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.

12 (4 marks) **Procedure Call and Return**.

12a Is a procedure call a static or dynamic jump? Justify your answer.

Static. The compiler knows the address of every procedure.

12b Is a procedure return a static or dynamic jump? Justify your answer.

Dynamic. A procedure can be called from multiple statements and each of these will have different return addresses. The same return statement must thus be able to jump to many different addresses, depending on which statement called it.

13 (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
```

14 **(20 marks)** The following SM213 assembly code implements a simple procedure. Carefully comment every line, give an equivalent C program that would compile into this assembly, and explain in plan English what this procedure does.

14a

```
X:  ld    0(r5), r0          # { r0 = item}
    ld    4(r5), r1          # { r1 = list}
    ld    8(r5), r2          # { r2 = i = n}
    dec   r2                 # { i = i - 1}
L0: bgt   r2, L1             # { goto L1(cont) if i > 0}
    beq   r2, L1             # { goto L1(cont) if i >= 0}
    br    L2                 # { goto L2(done) if i < 0}
L1: ld    (r1, r2, 4), r3    # { r3 = list [i]}
    mov   r3, r4             # { r4 = list [i]}
    not   r4                 # { r4 = ~ list [i]}
    inc   r4                 # { r4 = - list [i]}
    add   r0, r4             # { r4 = item - list [i]}
    bgt   r4, L2             # { goto L2(done) if (item > list[i])}
    inc   r2                 # { r2 = i + 1}
    st    r3, (r1, r2, 4)    # { list [i + 1] = list [i] if (item <= list [i])}
    dec   r2                 # { r2 = i}
    dec   r2                 # { i = i - 1}
    j     L0                 # { goto L0(loop)}
L2: inc   r2                 # { r2 = i + 1}
    st    r0, (r1, r2, 4)    # { list [i + 1] = item}
    j     (r6)               # { return}
```

14b Equivalent C program that would compile into this assembly:

```
void insertIntoSortedList (int item, int* list, int n) {
    for (int i = n - 1; i >= 0 && item <= list [i]; i --)
        list [i + 1] = list [i];
    list [i + 1] = item;
}
```

Or:

```
void insertIntoSortedList (int item, int* list, int n) {
    for (int i = n - 1; i >= 0; i --) {
        if (item > list [i])
            break;
        list [i + 1] = list [i];
    }
    list [i + 1] = item;
}
```

14c Plain English explanation of what this procedures does.

It inserts an integer into a sorted, ascending list of integers, maintaining sort order.

15 (10 marks) **Reading Assembly Code.** Consider the following snippet of SM213 assembly code.

```
foo: ld $s,     r0          # { r0 = &s}
     ld 0(r0),  r1          # { r1 = s.a}
     ld 4(r0),  r2          # { r2 = s.b}
     ld 8(r0),  r3          # { r3 = s.c}
     ld $0,     r0          # { r0 = 0}
     not        r1          # { }
     inc        r1          # { r1 = -a}
L0:  bgt        r3, L1      # { goto L1 if c>0}
     br         L9          # { goto L9 if c<=0}
L1:  ld         (r2), r4    # { r4 = *b}
     add        r1, r4      # { r4 = *b-a}
     beq        r4, L2      # { goto L2 if *b==a}
     br         L3          # { goto L3 if *b!=a}
L2:  inc        r0          # { r0 = r0 +1 if *b==a}
L3:  dec        r3          # { c--}
     inca       r2          # { a++}
     br         L0          # { goto L0}
L9:  j          (r6)        # { return}
```

15a Carefully comment every line of code above.

15b Give precisely-equivalent C code.

```
struct S {
    int  a;
    int* b;
    int  c;
};
S s;
int foo () {
    int  i=0;
    int* b=s.b;

    while (s.c>0) {
        if (s.a==*b)
            i++;
        s.c--;
        b++;
    }
    return i;
}
```

Or

```
int foo () {
    int i=0,j;

    for (j=0; j<s.c; j++)
        if (s.a==s.b[j])
            i++;
    return i;
}
```

15c The code implements a simple function. What is it? Give the simplest, plain English description you can.

It counts the number of elements in the integer array `s.b` whose size is `s.c` that have the value `s.a` and returns this number.

16 **(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;
}
```

```
countZero: ld   $len, r1          # r1 = &len
           ld   0(r1), r1         # r1 = len
           ld   $a, r2            # r2 = &a
           ld   0(r2), r2         # r2 = a
           ld   $0, r0            # r0 = c
loop:      bgt  r1, cont          # goto cont if len>0
           br   done              # goto done if len<=0
cont:      dec  r1                # len = len - 1
           ld   (r2, r1, 4), r3   # r3 = a[len]
           beq  r3, loop          # goto skip if a[len]==0
           inc  r0                # c=c+1 if a[len]!=0
           br   loop              # goto loop
done:      j    (r6)              # return c
```