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;
```

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`

```
ld $a, r1      # r1 = &a
ld (r1), r1    # r1 = a
ld 4(r1), r0   # i = s->y
```

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

```
ld $a, r1      # r1 = &a
ld (r1), r1    # r1 = a
ld 8(r1), r2   # r2 = a->z
ld 4(r2), r0   # i = a->z->y
```

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

```
ld $a, r1      # r1 = &a
ld (r1), r1    # r1 = a
ld 8(r1), r2   # r2 = a->z
st r1, 8(r2)   # a->z->z = a
```

4 **(6 marks)** **Static Control Flow.** Answer these questions using the register `r0` for `x` and `r1` for `y`.

4a Write commented assembly code equivalent to the following.

```
if (x <= 0)
    x = x + 1;
else
    x = x - 1;
```

```
      bgt r0, else  # goto else if x > 0
      inc r0        # x++ if a <= 0
      br   done
else: dec r0        # x-- if a > 0
done:
```

4b Write commented assembly code equivalent to the following.

```
for (x=0; x<y; x++)
   y--;
```

```
      ld $0, r0       # x = 0
loop: mov r1, r2      # r2 = y
      not r2
      inc r2          # r2 = -y
      add r0, r2      # r2 = x-y
      bgt r2, done    # goto done if x > y
      beq r2, done    # goto done if x == y
      dec r1          # y--
      inc r0          # x++
      br  loop        # goto loop
done:
```

5 **(6 marks)** **C Pointers.** Consider the following C procedure `copy()` and global variable `a`.

```
void copy (char* s, char* d, int n) {
    for (int i=0; i<n; i++)
        d[i] = s[i];
}

char a[9] = {1,2,3,4,5,6,7,8,9};
```

And this procedure call:

```
copy (a, a+3, 6);
```

List the value of the elements of the array a (in order), following the execution of this procedure call.

```
{1,2,3,1,2,3,1,2,3}
```

6 **(6 marks)** **Dynamic Allocation.** The following four pieces of code are identical except for the their use of `free()`. Each of them may be correct or they may have a memory leak, dangling pointer or both. In each case, determine whether these bugs exists and if so, briefly describe the bug(s); do not describe how to fix the bug.

6a

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  *dst = *src;
  return dst;
}
```

```
int foo() {
  int  a = 3;
  int* b = copy (&a);
  return *b;
}
```

Memory leak, because object allocated in `copy` is not freed in the shown code and when `foo` returns it is unreachable.

6b

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  *dst = *src;
  free (dst);
  return dst;
}
```

```
int foo() {
  int  a = 3;
  int* b = copy (&a);
  return *b;
}
```

Dangling pointer. After `free` in `copy`, `dst` is a dangling pointer. This value is returned by copy and so `b` in `foo` is also a dangling pointer. The last statement of `foo`, `return *b` dereferences this dangling pointer.

6c

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  *dst = *src;
  return dst;
}
```

```
int foo() {
  int  a = 3;
  int* b = copy (&a);
  free (b);
  return *b;
}
```

Dangling pointer. After `free` in `foo`, `b` becomes and dangling pointer and it is then dereferenced in the last statement.

6d

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  *dst = *src;
  free (dst);
  return dst;
}
```

```
int foo() {
  int  a = 3;
  int* b = copy (&a);
  free (b);
  return *b;
}
```

Dangling pointer. After `free` in `copy`, `dst` becomes a dangling pointer. This value is returned by `copy` and so `b` in `foo` is also a dangling pointer. The third statement of `foo` then calls `free` again on this value, attempting to free an object that has already been freed, which results in an error. If the program where to proceed it would then dereference the dandling pointer in the `return` statement.

7 **(8 marks)** **Reference Counting.** The following extends the code from the previous question by adding a procedure `saveIfMax` that is implemented in a separate module. Add calls to `inc_ref` and `dec_ref` to use referencing counting to eliminate all dangling pointers and memory leaks in this code while creating no *coupling* between `saveIfMax` and the rest of the code (i.e., `saveIfMax` can not know about what the rest of the code does and neither can the rest of the code know what `saveIfMax` does). Do not implement reference counting nor worry about storing the reference count itself; just add calls to `inc_ref` and `dec_ref` in the right places, **which may require slightly rewriting portions of the code**.

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  inc_ref (dst);
  *dst = *src;
  return dst;
}

int foo() {
  int  a = 3;
  int* b = copy (&a);
  saveIfMax (b);
  int t = *b;
  dec_ref (b)
  return t;
  return *b;

}
```

```
int* max;
void saveIfMax (int* x) {
  if (max==NULL || *x > *max) {
    if (max != NULL)
      dec_ref(max);
    max = x;
    inc_ref(max);
  }
}
```

8 (8 marks) **Procedures and the Stack.** Answer the following questions about this assembly code.

```
[01]         ld   $-12, r0
[02]         add  r0, r5
[03]         ld   $2, r0
[04]         st   r0, 0(r5)
[05]         st   r0, 4(r5)
[06]         st   r0, 8(r5)
[07]         gpc  $6, r6
[08]         j    foo
[09]         ld   $12, r1
[10]         add  r1, r5
[11]         ld   $0x1000, r1
[12]         st   r0, (r1)
[13]         halt

[14]   foo:  ld   $-8, r0
[15]         add  r0, r5
[16]         st   r6, 4(r5)
[17]         ld   8(r5), r0
[18]         ld   12(r5), r1
[19]         ld   16(r6), r2
[20]         add  r1, r0
[21]         add  r2, r0
[22]         ld   $8, r1
[23]         add  r1, r5
[24]         j (r6)
```

8a How many arguments, if any, does `foo()` have?

3

8b How many local variables, if any, does `foo()` have (count them even if they are not used)?

1

8c Is `foo()`'s return address saved on the stack at any point. If so, which line saves it?

Yes; line 16.

8d If you can determine the integer value in memory at address `0x1000` following the execution of this code, give its value.

6

9 **(10 marks)** **Reading Assembly.** Comment the following assembly code and then translate it into C. *Use the back of the preceding page for extra space if you need it.*

```
foo:    ld   $-12, r0           # r0 = stack space for ra and 2 locals
        add r0, r5              # allocate stack space for ra and 2 locals
        st   r6, 8(r5)          # save ra to stack
        ld   $0, r1             # i = 0
        st   r1, 0(r5)          # loc0 = 0
        st   r1, 4(r5)          # loc1 = 0 (later realized that this is i)
        ld   20(r5), r2         # r2 = arg2
        not r2                  #
        inc r2                  # r2 = -arg2
L0:     mov r2, r3              # r3 = -arg2
        add r1, r3              # r3 = i-arg2
        beq r3, L3              # goto L3 if i > arg2
        bgt r3, L3              # goto L3 if i == arg2
        ld   12(r5), r3         # r3 = arg0
        ld   (r3, r1, 4), r3    # r3 = arg0[i]
        ld   16(r5), r4         # r4 = arg1
        ld   (r4, r1, 4), r4    # r1 = arg1[i]
        ld   $-8, r0            # r0 = space for 2 arguments
        add r0, r5              # allocate stack space for 2 arguments
        st   r3, 0(r5)          # save bar_arg0 = arg0[i] to stack
        st   r4, 4(r5)          # save bar_arg1 = arg1[i] to stack
        gpc $6, r6              # r6 = return address for call to bar
        j    bar                # t = bar (arg0[i], arg1[i])
        ld   $8, r3             # r3 = space for 2 arguments
        add r3, r5              # deallocate stack space for 2 arguments
        beq r0, L2              # goto L2 if t==0
        ld   0(r5), r3          # r3=loc0 if t
        inc r3                  # r3++ if t
        st   r3, 0(r5)          # loc0++ if t
L2:     inc r1                  # i++
        br   L0                 # goto L0 (top of loop)
L3:     ld   0(r5), r0          # r0=loc0
        ld   8(r5), r6          # restore ra from stack
        ld   $12, r1            # r1 = stack space for ra and 2 locals
        add r1, r5              # deallocate stack space for ra and 2 locals
        j    (r6)               # return loc0
```

Translate into C:

```
int foo (int* arg0, int* arg1, int arg2) {
    int loc0 = 0;
    int i    = 0;
    for (i=0; i<a2; i++)
        if (bar (arg0[i], arg1[i]))
            loc0++;
    return loc0;
}
```