CPSC 213, Winter 2009, Term 2 — Midterm Exam **Solution**

Date: March 12, 2010; Instructor: Mike Feeley

1 **(2 marks)** **Memory Addresses.** Give an example of a memory address that is *aligned* to an four-byte boundary, but not to an eight-byte boundary.

4

2 **(4 marks)** **Pointer Arithmetic.** Consider the following three lines of C code. For the assignments to `i` and `j`, say (a) whether the code generates a runtime error and why or (b) what value the variables have after the code executes. If the first generates an error and the second does not, give the value of the second ignoring the first. Show your work.

```
int a[10] = { 0,1,2,3,4,5,6,7,8,9 };
int i = *(a +  ((&a[7])-a)  + *(a+2));
int j = *(a + *((&a[7])-a)  + *(a+2));
```

2a `i`:

No error. Value of i is 9.

```
int i = *(a +  ((&a[7])-a)  + *(a+2));
int i = *(a +  ((a+7)-a)  + *(&a[2]);
int i = *(a +  (7 + a[2]);
int i = *(a +  9);
int i = a[9];
```

2b `j`:

This statement will attempt to de-reference the address `((&a[7]-a)  == 7`, which would probably generate a runtime error. *On a non-Intel architecture it will be an error, because 7 is an unaligned address. The Intel ISA allows unaligned addresses, but on most operating systems, page 0 is protected and so attempting to read address 7 generates an invalid address error. To answer this question correctly, you can say either "it will generate an error" or "it will read memory at address 7, which it outside of the array" or something like that.*

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 Are *dangling pointers* possible in C, Java, both or either? Why or why not?

Possible in C, but not Java. The Java garbage collector ensures that an object is not freed if there are any references to it. In C, the programmers calls free explicitly and thus no such check is performed.

3b Are *memory leaks* possible in C, Java, both or neither? Why or why not?

Possible in both. Java programs can unintentionally retain references to dynamic memory and thus prevent it from being reclaimed by the garbage collector, which results in a memory leak.

4 **(11 marks)** **Global Arrays.** In C, global arrays can be declared in two ways, exemplified by `a` and `b` below.

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

4a Indicate which of the following the compiler knows statically and which is determined dynamically.

- the address of `a`? static
- the address of `b`? static
- the address of `a[0]`? dynamic
- the address of `b[0]`? static
- the number of bytes between `b[0]` and `b[i]`? dynamic

4b Give C code that assigns a value to `a` so that `a[0]==b[4]`, `a[1]==b[5]` and so on (don't copy the entire array, just assign a value to `a`).

```
a = &b[4];
```

4c Give SM213 assembly code that reads the value of `a[i]` into register `r0`; assume the value of `i` is in `r1`. Comment your code.

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

4d Give SM213 assembly code that reads the value of `b[i]` into register `r0`; assume the value of `i` is in `r1`. Comment your code.

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

4e Give SM213 assembly code that stores the value in `r0` in `a[i]`; assume the value of `i` is in `r1`. Comment your code.

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

5 **(6 marks)** **Instance Variables.** A C struct is a bit like a Java object, but without methods. Variables stored in an object or struct are called *instance variables*. Now consider this C declaration of two global variables, `a` and `b`.

```
typedef struct {
    int i, j, k;
} A;

A  a;
A* b;
```

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

- the address of `a`? static
- the address of `b`? static
- the address of `a.k`? static
- the address of `b->k`? dynamic
- the number of bytes between `b->i` and `b->k`? static
- the difference between the value of `b` and the address of `b->k`? static

5b Give SM213 assembly code that reads the value of `b->k` into `r0`. Comment your code.

```
ld   $b, r0          # r0 = &b
ld   (r0), r0        # r0 = b
ld   8(r0), r0       # r0 = r->b
```

6 **(8 marks)** **Procedures.** Consider the local variables and arguments declared in the following C code.

```
void foo (int a, int b, int c) {
    int i, j, k;
}
```

6a Indicate which of the following the compiler knows statically and which is determined dynamically.

- the address of `c`? dynamic
- the address of `k`? dynamic
- the offset to `k` from the value of the stack pointer? static

- the number of bytes between `i` and `k`? static

6b Give machine/assembly code that implements the procedure call `foo (1, 2, 3)`. Be sure to include every part of the procedure call statement (but just this statement). Pass the arguments on the stack.

```
add  $-12, r5        # make room for 3 args on the stack
ld   $1, r0
st   r0, 0(r5)       # arg0 = 1
ld   $2, r0
st   r0, 4(r5)       # arg1 = 2
ld   $3, r0
st   r0, 8(r5)       # arg2 = 3
ld   $foo, r0        # r0 = address of foo
gpc  r6              # r6 = pc
inca r6              # r6 = return address
j    (r0)            # foo (1, 2, 3)
add  $12, r5         # remove argument area from stack
```

7 **(7 marks)** **Dynamic Procedure Calls.** Procedure calls in C are normally static. Dynamic calls, like Java's polymorphic dispatch, however, have many benefits and can be implemented in C with *jump tables* (i.e., arrays of function pointers). Consider the following C code that declares (a) an array of pointers to four procedures and (b) a procedure that uses this array to invoke the ith one of them (starting with 0).

```
void procA () { printf ("A"); }
void procB () { printf ("B"); }
void procC () { printf ("C"); }
void procD () { printf ("D"); }
void (*proc[4])() = { procA, procB, procC, procD };
int i;
void foo () {
  proc[i] ();
}
```

7a Indicate which of these the compiler knows statically and which is determined dynamically.

- the address of proc[2]? static
- the value of proc[2]? static
- the address of the procedure that `foo` calls? dynamic

7b Give SM213 assembly code that implements the procedure call "`proc[i]`" using as few instructions as possible. Comment your code.

```
ld      $i, r0        # r0 = &i
ld      (r0), r0      # r0 = i
ld      $proc, r1     # r1 = &proc
gpc     r6            # r6 = pc
inca    r6            # r6 = return address
j       *(r1,r0,4)    # goto m[proc+i*4]
```

8 **(8 marks)** **Reading Assembly Code.** Consider the following snippet of SM213 assembly code.

```
     ld   $0, r0          # r0 = 0
     ld   0(r5), r1       # r1 = arg0
     ld   4(r5), r2       # r2 = arg1
L0:  bgt  r2, L1          # goto L1 if arg1>0
     br   L9              # goto L9 if arg1<=0
L1:  ld   (r1), r3        # r3 = *arg0
     shl  $31, r3         # r3 = *arg0 >> 31
     beq  r3, L2          # goto L2 if *arg0 is even
     inc  r0              # r0++ if *arg0 is odd
L2:  inca r1              # arg0++
     dec  r2              # arg1--
     br   L0              # goto L0
L9:  j    *(r6)           # return
```

8a Carefully comment every line of code above.

8b The code implements a simple function. What is it? Give the simplest, plain English description you can.

It computes `countOdd(int* arg0, int arg1)` that returns a count of the number of values in the integer array `arg0[0] ... arg0[arg1-1]` that are odd.