CPSC 213, Winter 2010, Term 1 — Midterm Exam **Solution**

Date: October 27, 2010; Instructor: Tamara Munzner

1 (2 marks) **Memory Alignment.** Consider the memory address 0x92. List all power-of-two sizes for which aligned memory access is possible and carefully justify your answer.

2. The lowest bit is zero, but the 2nd lowest bit is 1 so it is aligned only for 2-byte access. Alternately: the decimal equivalent 146 divides evenly by 2, but not by 4 or any larger power of 2.

2 marks: 1 for correct answer, 1 for correct justification. Thus mark of 1/2 if forgot to convert from hex and did computation for decimal 92 getting answer 2 and 4, or had correct logic but computational mistake.

2 (8 marks) **Pointer Arithmetic.** Consider the following lines of C code. For the assignments to `i` , `j`, `k`, and `m` say (a) whether the code generates an error and why or (b) what value the variables have after the code executes. If one line generates an error but a later one does not, give the value of the later ones. Show your work.

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

2a `i`:

No error. Value of i is 5.

```
int i = *(a+4);
int i = *(&a[4]);
int i = a[4];
int i = 5;
```

2b `j`:

No error. Value of j is 2.

```
int j = &a[3] - &a[1];
int j = 2;
```

2c `k`:

No error. Value of i is 5.

```
int k = *(a+*(a+6);
int k = *(a+*(&a[6]));
int k = *(a+ 3);
int k = *(&a[3]);
int k = 6;
```

2d `m`:

This attempt to de-reference the address `((&a[5]-a) == (&a[5]-&a[0]) == 5`, statement will generate an error (from the compiler or at runtime). The address 5 is not unaligned. It is also a protected location on most architectures.

3 (5 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. Answer true or false to the following questions:

- Dangling pointers can occur in C. True
- Dangling pointers can occur in Java. False
- Memory leaks can occur in C. True
- Memory leaks can occur in Java. True

- 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

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

```
  ld   $b, r0              # r0 = &b (pointer to struct)
  ld   0x0(r0), r0         # r0 = b (address of struct itself)
  ld   0x1c(r0), r0        # r0 = b->k (content of address of field in struct)
```

3 marks. Full credit also given for alternate answer using indexed load. Subtle point: while the offset has to fit into one hex digit of machine language, what's actually stored is in the machine language instruction is the number from the assembly language instruction divided by 4. Hex 1c is decimal 28. $28/4 = 7$, and 0x7 will indeed fit into that digit. But full credit also given for slightly more verbose solution:

```
  ld  $b, r0       # r0 = &b (pointer to struct)
  ld  0x0(r0), r0  # r0 = b (address of struct itself)
  ld $0x1c, r1     # r1 = decimal 28 (7 integers * 4 bytes each)
  add r1, r0       # r0 = address of struct field b->k
  ld 0x0(r0), r0   # r0 = value of b->k field
```

6 **(12 marks)** **Procedures and Writing Assembly Code.** Consider the following procedure in SM213 assembly code. Assume that arguments have been passed in on the stack, not in registers.

```
int sum (int *a, int i, int* b, int j) {
    return a[i] + b[j];
}
```

6a Why would we use the stack to store arguments instead of just using registers? Explain carefully.

There are a limited number of registers, so procedures with many arguments would not be able to store all arguments on the stack. Also, the convention in SM213 is to use specific registers to store information like the return address and the stack pointer, so the number of registers available for free use inside a procedure is even more limited. Using the stack in the first place avoids the need to save registers to the stack in subsequent procedure calls.

2 marks. The answer "the stack is better than registers because it's too hard to keep track of which register is used for which parameter" is not correct: the compiler can indeed figure that information out at compile time. The answer "the stack is more efficient than using registers" is not correct: the stack is slower since it requires memory accesses. Most answers involving dangling pointers or polymorphism were incorrect.

6b What is known statically vs dynamically?

- The offset between the stack pointer for sum and the address of a? static
- The offset between the stack pointer for sum and the address of i? static
- The address of a? dynamic
- The address of i? dynamic

6c Implement the C procedure above in SM213 assembly code. Assume that arguments have been passed in on the stack, not in registers. Comment your code.

```
sum:    ld  0(r5), r1       # r1 = a
        ld  4(r5), r2       # r2 = i
        ld  8(r5), r3       # r3 = b
        ld  12(r5), r4      # r4 = j
        ld  (r1,r2,4), r0   # r0 = a[i]
        ld  (r3,r4,4), r1   # r1 = b[j]
        add r1, r0          # r0 = a[i] + b[j]
        j   (r6)            # return
```

8 marks.

- +1 for using parameters on the stack that were placed by the caller, as opposed to pushing parameters explicitly in this code or using registers.
- +1 for having the correct offsets for stack arguments
- +1 for leaving the teardown to the caller (not doing it explicitly here)
- +1 for correct calculation for a/b
- +1 for correct calculation for i/j
- +1 for add
- +1 for having return value in r0
- +1 for jump to r6 value
- -1/2 for moving result to r0 instead of computing it there in the first place (verbose)
- -1/2 for wrong order of arguments
- -1/2 for changing the stack pointer instead of using offsets from it
- -1/2 for jump to r6 instead of (r6)

7 **(4 marks)** **Static Control Flow.** Consider the following procedure in C.

```
if (a > 2) b = 3;
else if (a < -4) b = 5;
```

In SM213 assembly, how many branch/jump statements are needed to implement this code? How many of these are conditional and how many are unconditional? Justify your answer.

Three total, two conditional and one unconditional. One conditional is needed for each of the two tests in the C code. One unconditional is needed to skip past the second code block. A second unconditional is not needed at the end of that block, because control will simply flow through to the next line.

4 marks: 1 for correct answer of 3 total, 1 for its justification. 1 for correct answer of 2 cond + 1 uncond, 1 for its justification.