UNIVERSITY OF BIRMINGHAM

School of Computer Science

First Year – MSc Computer Science

06 02525

Fundamentals: Introduction to Computer Science

Summer Examinations 2013

Time Allowed: 1:30 hours

[Answer Question ONE and Three Other Questions]

For reference: In some of the questions you may find the following reminder of Java bytecodes useful.

Many bytecode instructions start with a letter that indicates the type of the data being used. `i`,`d`,`a` indicate int, double and object reference (address) respectively. We omit the type letter in what follows.

"#" followed by a number is an entry in the run-time constant pool, used to refer for example to a class or a method. A comment in the disassembled bytecode will explain what that entry is.

const pushes a constant value onto the operand stack.

ldc2_w pushes a long or double constant from the constant pool onto the operand stack.

load pushes the contents of a local variable onto the operand stack.

store pops the top of the operand stack into a local variable.

putfield sets a field in an object: the value to store is top of the operand stack, the object reference is second from top, and the field name is a "#" reference (see above).

getfield gets field from an object: similar to `putfield`, but the object reference is top of the operand stack and is replaced by the value got from the field.

add, sub, mult add, subtract or multiply the top two entries on the operand stack.

cmpl compares the second from top (v_1) on the operand stack with the top (v_2), pops them both, and pushes an integer result defined as

$$\left\{ \begin{array}{r} 1 \\ 0 \\ -1 \end{array} \right\} \text{ if } v_1 \left\{ \begin{array}{c} > \\ = \\ < \end{array} \right\} v_2$$

ifeq, ifne, iflt etc. are conditional jumps: the jump takes place if the top of the operand stack is equal to 0, not equal, less than, etc.

invokestatic, invokevirtual, invokespecial call methods of various kinds.

[Answer THIS question – 40%]

1. (a) (i) Suppose a CPU uses 8 bytes for its memory addresses. How many bytes of memory can it address? Give your answer both as a power of 2 and (approximately) in decimal. **[5%]**

   (ii) Write the 1-byte hex number E0 in binary, in decimal as an unsigned number, and in decimal as a signed number. **[5%]**

   (b) What are the *Internet* and the *World Wide Web*? Give a technical answer with just enough detail to explain why they are not the same. Also mention the protocols that they use. **[10%]**

   (c) What is an *instance invariant* for a class in Java? Explain the role of (i) private instance variables and (ii) defensive method definitions in helping to ensure that invariants are always kept true. **[10%]**

   (d) (i) What does it mean to say that an algorithm executes in *polynomial time*? **[5%]**

   (ii) Briefly explain the classes P and NP of problems, and the "P = NP" question. **[5%]**

[Answer THREE out of Questions 2, 3, 4 and 5 – 20% each]

2. (a) Explain how the *program counter* (pc) is used in the fetch-execute cycle, and how jumps are executed. **[3%]**

Suppose a particular CPU (not a real one) has a 256-byte address space. Its registers are 1-byte each, and one of them is referred to as the a-register. Each instruction is either one or two bytes, an opcode byte and possibly an extra byte whose value we shall write as N. Three of the instructions are described as follows. Hex notation is used throughout.

Number of bytes	Opcode byte	Mnemonic	Action
2	01	`st [a]` N	Writes value N to memory at the address held in the a-register.
2	02	`jpnz a` N	If the a-register is non-zero, jumps to address N.
1	03	`dec a`	Subtracts 1 from the a-register.

At the start, the first 8 memory locations have the following contents:

Contents	00	00	00	01	FF	03	02	03	...
Address	00	01	02	03	04	05	06	07	08

(b) Memory locations 03 and 04 contain the instruction "`st [a] FF`". Suppose the pc has value 03 and the a-register has value 02. Describe in detail how the instruction is fetched from memory and executed. List *all* the memory reads and writes involved in this, stating the memory addresses and the values read or written. **[7%]**

(c) Write the mnemonics for the instructions in locations 05 to 07. **[3%]**

(d) Starting as before, with pc and a-register having value 03 and 02, write a table showing the following for each instruction executed: the pc for that instruction, the values of the a-register and memory locations 00, 01, 02 after the instruction has executed, and the pc for the next instruction to be executed.

pc	a	00	01	02	next pc
:	:	:	:	:	:

Continue the table for as long as the CPU is executing instructions from memory locations in the address range 03 to 07. **[7%]**

3. A Java class `Temperature` is written for objects to record maximum temperature. Although it works in Celsius, it has convenience methods that take Fahrenheit parameters. Here is part of its definition.

```
public class Temperature {
  private double maxC; //current maximum in Celsius

  public void updateF(double newF){
    double newC = 5.0/9.0*(newF-32); //new temperature in Celsius
    if (newC > maxC){
      maxC = newC;
    }
  }
```

Here is the disassembled bytecode for the method `updateF`.

```
 0: ldc2_w        #3   // double 0.5555555555555556d
 3: dload_1
 4: ldc2_w        #5   // double 32.0d
 7: dsub
 8: dmul
 9: dstore_3
10: dload_3
11: aload_0
12: getfield      #2   // Field maxC:D
15: dcmpl
16: ifle          24
19: aload_0
20: dload_3
21: putfield      #2   // Field maxC:D
24: return
```

(a) The stack frame for `updateF` has space for five local variables, but the only indexes used are 0, 1 and 3. Explain what these correspond to in the Java source and why indexes 2 and 4 are not used. **[3%]**

(b) Suppose the `main` method constructs an instance `t` of `Temperature` with value 0 for its instance variable (field) maxC, and calls `t.updateF(50)`. (50°F is 10°C.)

(i) Explain how this call is achieved, with reference to both the operand stack in the stack frame for `main` and the new stack frame for `updateF`. **[7%]**

(ii) For each instruction in the disassembled bytecode for `updateF`, list all the entries on the operand stack when that instruction has executed. Indicate which stack entries are double length (i.e. `long` or `double`). **[10%]**

4. "Russian" multiplication works by replacing `x*m` by `(2*x)*(m/2)` and repeating. Because `m/2` is an integer division, there needs to be a correction if `m` is odd, and this is seen in `z` in the following Java implementation.

   Although there is little point in doing multiplication this way in Java, the same idea is very useful for calculating powers – see part (e).

```
/** Multiply two numbers by the Russian method.
 *    There are no actual multiplications or divisions except by 2.
 *  requires: m >= 0
 *  @param x one number to multiply
 *  @param m other number to multiply
 *  @return x*m
 */
public static double multRussian(double x, int m) {
  double y =     ;
  int n =        ;
  double z =     ;
  /* loop invariant:
   *   n >= 0 and y*n + z = x*m
   */
  while (n > 0) {
    if (n%2 == 1) {
      z = z+y;
    }
    n = n/2;
    y = y*2;
  }
  return        ;
}
```

   (a) Is this *defensive* or *non-defensive* specification? What does the "requires" line in the Javadoc signify? **[2%]**

   (b) Complete the `return` statement. How does the loop invariant help you to know that you are returning the correct answer? **[3%]**

   (c) How should `y`, `n` and `z` be initialized? (Do not use any arithmetic to calculate the initial values.) **[3%]**

   (d) Explain why the loop invariant is true initially and after each iteration. **[10%]**

   (e) Why would the same idea give an efficient way to calculate powers x^m? Approximately how many multiplications would it need (as a function of m)? **[2%]**

5. (a) (i) In a multiprocessing operating system, explain each of the the following three possible states of a process: *running*, *ready* and *waiting*. **[3%]**

   (ii) Given two reasons, apart from ones that arise in the process's own code, why a running process might become ready. **[2%]**

   (b) Suppose two processes P_1 and P_2 both access a variable `x` that they share. No other processes have access to `x`. The processes are running machine code, but at a high level their next actions are:

   P_1: `x = x+1;`

   P_2: `x = x+2;`

   (The machine code for each process loads the value of `x` into a register `a`, adds 1 or 2 to `a` and then stores the result back in `x`. Also, each process has its own copies of registers, which are saved when the process is not running.)

   Process P_1 is running, P_2 is ready. Together they should add 3 to `x`. Explain with an example how this may fail to happen: both processes perform their updates, but `x` changes by a value other than 3. **[10%]**

   (c) What facilities can an operating system provide to enable processes such as those in part (b) to run safely so that the variable `x` increases by 3? **[5%]**