1) 20pts. Interpretation.
   (a) 10pts. Describe what direct threading is in an interpreter, why its used (what does it save?), and one disadvantage of using it.
   (b) 5pts. What are superinstructions (give an example)?
   (c) 5pts. How does using superinstructions improve performance of the interpreter (what is it optimizing)? What about replication (what is it optimizing)?
2) **25pts. Compilation**

(A) 3pts. How do we identify the basic blocks in a method exactly (which instructions are leaders)?

(B) 6pts. What are dynamic basic blocks? What is one disadvantage of using dynamic basic blocks (give an example)?

(C) 6pts. What code does a compiler generate if a method being compiled accesses a field in a class that has not yet been loaded? For table-based linking, what does this code do when executed the first time? For table-based and patch-based linking, what does the code do when executed the second time?
(D) 10pts. For the code below, perform linear scan register allocation. You have 2 registers to work with. Give the Live Interval for each register (Rx for x=1,2,3,4,7) and the Active List (with entries in the correct order) at each step in the algorithm. State the reason behind all spills, if any. Point out why the algorithm in not optimal in this example.

<table>
<thead>
<tr>
<th>Live Interval</th>
<th>Active List</th>
<th>Spill</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) R3 = a //param</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2) R2 = c //param</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3) R1 = R3 + R2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4) R4 = R2 + 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5) foo(R3,R4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6) R7 = R4 * R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7) return R7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extra Credit (2pts): Describe one key difference in the greedy register allocation strategy used in TraceMonkey (the tracing Javascript paper) vs Jalapeno’s (JikesRVM’s) linear scan register allocation.
3) 20pts. Internal Runtime Representations

For the code below, draw (on the next page) the runtime statics table and its map, the internal class representations of each class and their maps, and the object layouts for the two local variables in main.

Assume that we are executing at the end of B.foo() and about to return and that we got here by executing main in class A the first time. That is, figure out which classes are loaded and which methods are compiled (draw bodies and give addresses of), using this execution path. Assume that the header of an object is 2 words wide (word=4 bytes).

Use these addresses:
0x789 compiler stub method
0x999 class A’s internal representation (VMT)
0x888 class A’s class map
0x777 class B’s internal representation (VMT)
0x666 class B’s class map
0x555 the VM’s statics table
0x444 the VM’s internal map

For each entry, specify the following:
index, name, value, one of: sm, im, o
sm = static method
im = instance method
o = something other than sm or im

Note that this is Java: among other things all Java fields are given default values by the VM if not initialized.

Assume that the object header size is two words (8 bytes) in total. Specify the class name of the member if ambiguous. Use the tables below and add any other memory objects, as needed. Number of entries in tables may not match the solution (there may be more/less).

class A {
    static int field1 = 4;
    int field2 = 3;
    int field3 = 7;
    static int field4;
    static void m1() {...}
    void m2() {...}
    void m3() {...}
    A() {...}
    static void main() { B.foo(); }
}

class B extends A {
    static int fielda = 4;
    int field2;
    int fieldc = 2;
    void m2() {...}
    static void m4() {...}
    void m5() {...}
    B() { ... }
    static void foo() {
        A tmpA1 = new A();
        A tmpA2 = new B();
        tmpA2.m2();
    }
}


Extra Credit (3pts): Use the memory addresses in the picture above to describe the steps during execution of compiled code that occur when the line `tmpA2.m2();` is invoked in foo, assuming that this m2 is compiled. How is this different from if `A.m1()` was invoked?