Individual Exercises - Lecture 15

1. Read the articles under additional references.
   The lean, mean, virtual machine:
   https://www.infoworld.com/article/2077184/the-lean--mean--virtual-machine.html
   The Jasmin User Guide:
   http://jasmin.sourceforge.net/guide.html
   Technical Overview of the Common Language Runtime (or why the JVM is not my favorite execution environment):
   https://www.semanticscholar.org/paper/Technical-Overview-of-the-Common-Language-Runtime-Meijer-Miller/60d90adf79acd100704ccc5d476982569272251c

2. Do Fischer et al exercise 4, 7, 8, 9 on pages 414-415 (exercise 4, 5, 8, 10 on pages 446-447 in GE)

   4. Why are there two instructions (ldc and ldc_w) for pushing a constant value on top of the stack?
   In summary, ldc selects what to push onto the stack using one byte for the index while ldc_w uses two bytes (wide index). The wide index is constructed as indexbyte1 << 8 + indexbyte2.

   7. The JVM has two instructions for unconditional jumps: goto and goto_w. Determine the appropriate conditions for using each instruction.
   The goto instruction forms a jump offset using two bytes, while the goto_w forms the jump offset using four bytes. Thus, if the intended target of the jump can be reached by an offset o, $-32768 \le o \le 32767$, then the goto instruction suffices. Otherwise, the goto_w instruction must be used. It turns out, however, that the size of a JVM method cannot exceed 65535 bytes, so the goto_w instruction is rarely needed.

   8. The ifeq instruction provisions only 2 bytes for the branch offset. Unlike goto, there is no corresponding goto_w for instruction ifeq. Explain how you would generate code so that a successful outcome of ifeq could reach a target that is too far to be reached by a 16 bit offset.
   The ifeq instruction is a conditional jump using two bytes for the offset, to jump further one can simply jump to another jump instruction that supports a larger offset like goto_w.

9. Consider the C or Java ternary expression: (a > b) ? c : d, which leaves either c or d on TOS depending on the outcome of the comparison. Assume all variables are type int, but do not assume that any of them are 0. Explain how you would generate code for the comparison that uses only the ifne instruction for branching (no other if or goto instructions are allowed).

The ifne instruction can be used to compute a > b by checking that all other conditions are not True, for example checking for equality by subtracting the two numbers and checking if the result is not 0.
If a > b then a - b would be negative. Then we can use ifne to check the sign bit by shifting right 31 bits (Given that the int is represented as 32 bit signed).
For a more thorough explanation see Solutions Manual for Crafting a Compiler available on Moodle (Page 35)

3. (optional) Follow the studio associated with Crafting a Compiler Chapter 10: Intermediate Representations
The materials for this exercise can be found in the virtual machine available on Moodle.

4. (optional) Follow the Lab associated with Crafting a Compiler Chapter 10: Intermediate Representations
The materials for this exercise can be found in the virtual machine available on Moodle.

Group Exercises - Lecture 15

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. Do Fischer et al exercise 5, 6, 10 on pages 414-415 (exercise 6, 7, 9 on pages 446-447 in GE)

   5. The form of the getstatic instruction is described in Section 10.2.3 as having both a *name* and a *type*. While the name is certainly necessary to access the desired static field, is the type information really necessary? Recall that the accessed field field has declared type in the class defining the field
   The JVM is designed to support separate compilation of Java classes. A getstatic instruction may be issued for a static element of a class that has been modified and recompiled out-of-view of the code that is fetching the static element.
   We therefore could have two different views of the type of some static element: one at the instruction fetching the value, created when the fetching code was compiled, and another at the class providing the value, created when that code was compiled.
   The type of the static field is included in the getstatic instruction so that the actual and supposed type of the static element can be checked for compatibility.

   6. The getstatic instruction described in Section 10.2.3 requires that the static field's name be specified as an immediate operand of the instruction.

   Suppose the JVM instead had a getarbitrary instruction that could load a value from some arbitrary location. Instead of specifying the field by its name in an immediate operand, the location's address would be found at TOS. What are the implications of such an instruction on the performance and security of the JVM?
   If JVM had a getarbitrary instruction without any checks it could be used like a pointer in C and, e.g., read parts of a value such as the second half of a 64-bit long as a 32-bit int. But if the instruction has to be made secure JVM would need to check that the address is the start of a value, that the value matches the expected type, and that value is allowed to be read. So to keep guarantees to the getstatic instruction, the implementation of getarbitrary would have a higher complexity.

   10. A constructor call consumes the TOS reference to the class instance it should initialize. All constructors are void, so they do not return any kind of result. However, Java programs expect to obtain the result of the constructor call on TOS after the constructor has finished. By what JVM instruction sequence can this be accomplished?
   As constructors consume the reference put on the top of the stack by *new instruction,* the reference should be duplicated by dup before the constructor is executed.
   For more information see Stack Operations in Fisher Section 10.2.

3. Discuss the advantages and disadvantages of types in the JVM instruction set.
   Might be easier to implement statically typed languages on top of the JVM, since the JVM bytecode verifier provides type checking before executing code. This means that certain program errors that could otherwise lead to unintended behavior can be detected by the JVM at runtime. Implementing type-safe interpreters are made simpler by having typed instructions.

   On the other hand the JVM provides no way of encoding type-unsafe features of typical programming languages, such as pointers, immediate descriptors (tagged pointers), and unsafe type conversions. As such, it may be slightly more difficult to implement dynamically typed languages in the JVM(although "Da vinci Machine" project extended the JVM to better support dynamic languages. This fx. added instructions that allowed method invocation without type checking).
   https://www.cin.ufpe.br/~haskell/papers/Technical_Overview_of_the_Common_Language_Runtime-Meijer&Miller.pdf
   https://openjdk.java.net/projects/mlvm/

   Typed instructions introduce a large number of instructions that have identical behavior and differ only in their type. Furthermore, the JVM uses only one byte for opcodes. If every typed operation supported every type in jvm, the amount of instructions there would exceed what we could represent in a single byte. Instead the JVM instructions have a reduced level of support for certain types(less orthogonality). For a more thorough explanation, and a table showing what combination of operations and types are available see:
   https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.11.1

4. MS IL has similar properties to JVM instruction, but they are not all typed. Discuss why that is the case and what are the consequences.
   Having a single untyped instruction (e.g., add) instead of one instruction per type (e.g., iadd ladd fadd dadd) significantly reduces the number of instructions in the instruction set. However, having untyped instructions makes the implementation of an interpreter more difficult, as it must determine what operation to perform from its operands. This is not a problem for the CLR as all methods are compiled when executed, however, Oracle's HotSpot JVM starts by interpreting the bytecode and then compiles code executed repeatedly.
   For more information see:
   https://blog.overops.com/clr-vs-jvm-how-the-battle-between-net-and-java-extends-to-the-vm-level/
   https://www.oracle.com/technical-resources/articles/java/architect-evans-pt1.html
   https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process