Name: ____

IB Computer Science 2
Program translation worksheet

Compiling to assembly code

Assembly code is a human-readable representation of compiled code. Each processor architecture has its own assembly language; here we will examine that of the Java Virtual Machine (JVM). To do so, we will use the Java compiler together with a *disassembler* that shows the compiled program's *instructions* in text form.

1. Place the following Java code in a file (for example, `Translation.java`):

```
if(Integer.parseInt(System.console().readLine("int? "))==0) {
    System.out.println("It is zero");
}
```

   Compile it to bytecode by running a command of the form: `javac Translation.java`
   A file whose name is of the form `Translation.class` should appear in the folder.
   Disassemble it with a command of the form: `javap -c Translation`

   a. Each line of output corresponds to a single instruction telling the JVM to perform some action. Find the four consecutive instructions that correspond to the `if` control-flow structure in the source code. Copy them below, without comments.

   b. The `javap` command shows a number to the left of each line of assembly code. What is the purpose of these numbers? Briefly discuss why there are gaps between them, but move on if you cannot come up with a reason.

   c. It can take many assembly instructions to complete a single source code statement. Looking at the comments, determine which of the lines you copied above correspond to the print statement. Draw a labeled box around them.
   d. Find the instruction that corresponds to the `if` statement itself. Based on its name, what do you think it does?

e. The instruction you just identified expects two operands. In this case, one of them is the integer that the user input, but what is the other? Looking through the earlier instructions, copy down the one that reveals the answer.

f. What do you find surprising about the instruction that `javac` chose for the `if`?

g. What does the number that immediately follows this instruction mean? Use it to explain why the compiler changed the program logic.

2. Change the == to a < and the output to `It is negative`. Do not compile it yet!

   a. Predict what type of comparison the compiler will choose now.

   b. Recompile and disassemble the program. Copy down the instruction that corresponds to the `if` statement, including the number that follows it.

   c. Was your prediction correct? If not, what was your mistake?

3. In the Java code, add an `else` case that prints `It is nonnegative`. Recompile and disassemble it.

   a. Copy down the instruction that corresponds to the `if` statement, including the number that follows it.

   b. Compare this instruction against the one from the previous version of the program. What has changed and why?

   c. Compare the structure of the assembly code against the instructions you copied down at the beginning of the activity. Besides the body of the `else`, what other instruction has the compiler added and why?

Name: ____

d. Write rough pseudocode for the process of translating a conditional from Java to JVM assembly.

4. Place the following Java code in a new file. Then, compile and disassemble it.

```
while(Integer.parseInt(System.console().readLine("int? ")) < 0)
{
        System.out.println("Input cannot be negative");
}
```

a. Find the instruction that corresponds to the `while` statement's condition. How does it compare to the one from the previous version of the program?

b. Find the instruction that causes the program to repeat and copy it down, including the number that follows it.

c. We have seen this instruction before. What is different about this use of it that creates a loop?

d. Write rough pseudocode for the process of translating a `while` loop from Java to JVM assembly.

If you have extra time…

5. In the Java code, convert the `while` loop into a `do..while` loop. Compile but do not disassemble it yet.

    a. Predict what the compiler will do differently now.

    b. Disassemble the bytecode and check your prediction. If your prediction did not match the compiler's translation, would your approach also have worked? If you predicted correctly, can you come up with another possible translation?

6. Place the following Java code in a file. Then, compile and disassemble it.

```
for(int counter = 0; counter != 3; ++counter) {
     System.out.println(counter);
}
```

    a. Copy all of the instructions that make up `main()`, without comments.

    b. Draw a labeled box around each set of instructions that corresponds to one of the four statements in the Java code.

7. In the Java code, rewrite the `for` loop as a `while` loop. Compile but do not disassemble it yet.

    a. Predict what the compiler will do differently now. Then disassemble the bytecode and check your prediction.

Name: ____

Assembling to machine code

Machine code is the machine-readable representation of compiled code. It encodes the exact same sequence of instructions as the corresponding assembly, but as raw bytes representing numbers rather than printable text characters. For this reason, machine code for a virtual machine such as the JVM is also called *bytecode*. The class files produced by the Java compiler contain JVM bytecode.

8. Compile and disassemble your final conditionals program (the one with the `if..else`). Make sure you can find a passage like the below. If any of the numbers differ in your version, change them here now.

```
15: ifge          29
18: getstatic     #27
21: ldc           #31
23: invokevirtual #33
26: goto          37
29: getstatic     #27
32: ldc           #39
34: invokevirtual #33
37: return
```

   a. Run `hexdump -C` on the class file to display its binary contents as hexadecimal numbers. On the right side, you will see the ASCII character equivalent of each byte, at least for the part of the file that contains text data.
   b. Look up the hexadecimal number (`0x`##) corresponding to the `ifge` instruction in the "Opcode Mnemonics by Opcode" chapter of *The Java Virtual Machine Specification*. Looking after the text portion of the file in the hexdump, find the beginning of this instruction. Write it and the two bytes that follow above, next to the `ifge` line.
   c. What is the decimal equivalent of the two additional numbers you copied? How does it relate to the number to the *right* of the instruction in the above assembly?

   d. Now look up the hexadecimal number for `getstatic` in the specification. Where does it appear in the hexdump? Why are there gaps between the numbers to the *left* of each instruction in the above assembly?

   e. Write all the bytes of the `getstatic` instruction next to its line above. What is each byte for? Now run `javap -v` on the class file to see its constant pool, the text portion from the hexdump. What is the number to the *right* of the instruction?

If you have extra time…

f. Write rough pseudocode for the process of translating assembly to bytecode.