Project Application Development

Building an IJVM emulator

Sebastian Österlund
Vrije Universiteit Amsterdam

August 8, 2017

Contents

Preface

Welcome to the course Project Application Development!

Goal of this course

The main aim of this course is to give the student a more hands-on practical experience with programming. By implementing a larger project with a basis in material covered during the first year of the Computer Science bachelor, you will have the opportunity to gain more experience developing software, as well as showing off what you have learned during the first year.

Course in a nutshell

In this course you will implement an emulator capable of executing IJVM byte-code, as covered in the course Computer Systems. The implementation will be done in Java. The assignment is split into several smaller parts that build up to the final deliverable. During this course you will get individual guidance by a Teaching Assistant.

Structure of this reader

This reader is split into five units. Each chapter will build on the previous chapters and introduce some new tasks.

About the grading

- Your program has to pass all the basic automatic tests. (50%)
- Your grade will depend on the number of passed advanced tests. (20%)
- Your program will be graded on style and general impression. (20%)
- You can achieve a higher grade by implementing additional functionality (20%). Note: you are only eligible for these points if you pass the advanced tests.

- Naturally, your final grade is capped at a 10.

Based on this grading scheme it is, thus, possible to pass the course by passing the basic tests and getting a sufficent grade for the style.

Introductory Chapter

> *"However, as every parent of a small child knows, converting a large object into small fragments is considerably easier than the reverse process."*
>
> – Andy Tanenbaum, *Computer Networks, 4th ed*

0.1 The IJVM architecture

During this course we will work extensively with the IJVM Instruction Set Architecture as presented in [1]. The IJVM instruction set is a subset of the Java instruction set, containing only operations on integers. It, thus, removes much of the complexity added by typing. In this chapter we will try to introduce some of the concepts in a simple manner. For more in-depth details you have to read the book by Tanenbaum [1]. In the end of the course you will have a fully functional IJVM emulator that can execute IJVM bytecode. To achieve this we will split the project into five tasks:

1. Parsing the binary.
2. Implementing basic stack manipulation.
3. Implementing control flow.
4. Implementing operations on local variables.
5. Implementing method calls.

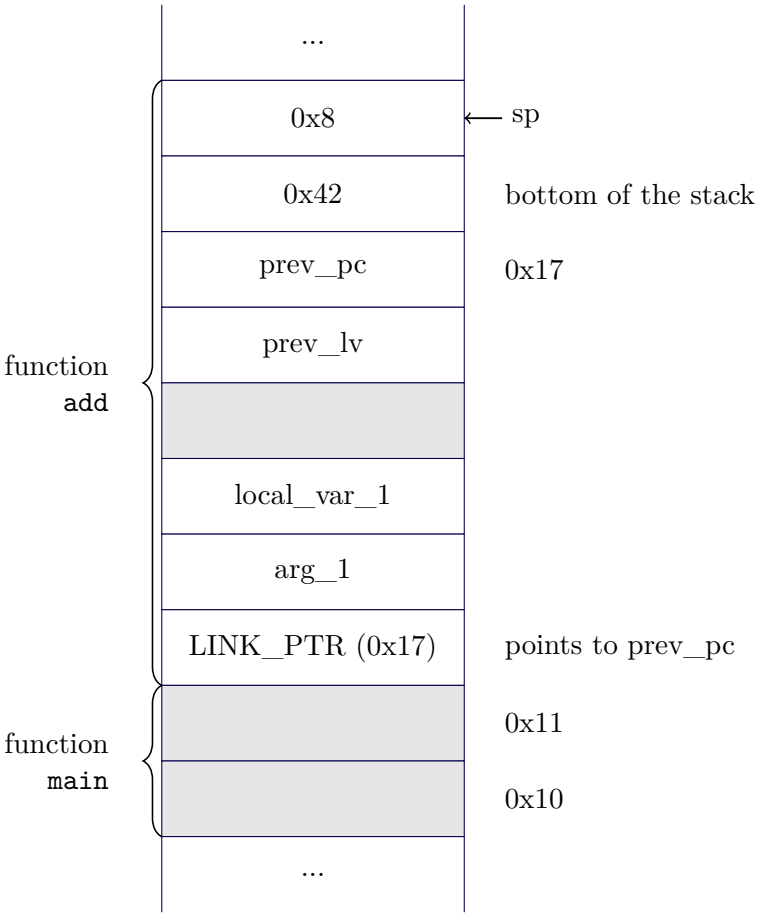
By implementing each task and thoroughly testing it, you will avoid common pitfalls that become hard to debug at a later stage[1]:

0.1.1 IJVM memory layout

The IJVM is a stack-based architecture, meaning that (almost) all operations are performed on a stack. This is different compared to architectures such as Intel x86 which performs its operations on registers. One main benefit of a stack-based architecture is that the instruction set becomes simple, since the operations don't need a source and destination operand.

[1] We will provide a basic set of test cases for each stage of the assignment. Testing it more thoroughly is up to you!

The most essential part of the IJVM architecture is *the stack*. The stack is built up of *local frames*, one for each method invoked. The bottom of the frame contains a *link pointer* which points to the location at which the previous program counter is stored. Above the link pointer, you can find the method arguments and local variables (method arguments are treated in roughly the same manner as local variables). Above the local variables, you can find the previous *frame pointer* (lv) and *stack pointer* (sp). The values stored at these locations are restored appropriately when a method returns. For a more detailed explanation of what the IJVM memory model looks like, see [2].



The program code (also called *TEXT*) resides in its own location in memory. The *Program Counter* (pc) keeps track of which instruction should be executed next.

Besides the stack and the text, there is one other memory area, namely, the *constant pool*. This area contains immutable constant values that are loaded on program start-up.

[2] Structured Computer Organization, 4.2.2 p. 260

0.1.2 Instruction set

Table 1: The IJVM instruction set

Instruction	Args	OpCode	Description
BIPUSH	byte	0x10	Push a byte onto stack
DUP	N/A	0x59	Copy top word on stack and push onto stack
ERR	N/A	0xFE	Print an error message and halt the simulator
GOTO	short	0xA7	Unconditional jump
HALT	N/A	0xFF	Halt the simulator
IADD	N/A	0x60	Pop two words from stack; push their sum
IAND	N/A	0x7E	Pop two words from stack; push bit-wise AND
IFEQ	short	0x99	Pop word from stack and branch if it is zero
IFLT	short	0x9B	Pop word from stack and branch if it is less than zero
IF_ICMPEQ	short	0x9F	Pop two words from stack and branch if they are equal
IINC	byte byte	0x84	Add a constant value to a local variable. The first byte is the variable index. The second byte is the constant.
ILOAD	byte	0x15	Push local variable onto stack
IN	N/A	0xFC	Reads a character from the input and pushes it onto the stack. If no character is available, 0 is pushed
INVOKEVIRTUAL	short	0xB6	Invoke a method, pops object reference and pops arguments from stack.

Continued on next page

Table 1 – *Continued from previous page*

Instruction	**Args**	**OpCode**	**Description**
IOR	N/A	0xB0	Pop two words from stack; push bit-wise OR. **Note:** the book uses the opcode `0x80`.
IRETURN	N/A	0xAC	Return from method with a word value
ISTORE	byte	0x36	Pop word from stack and store in local variable
ISUB	N/A	0x64	Pop two words from stack; subtract the top word from the second to top word, push the answer;
LDC_W	short	0x13	Push constant from constant pool onto stack
NOP	N/A	0x00	Do nothing
OUT	N/A	0xFD	Pop word off stack and print it to standard out
POP	N/A	0x57	Delete word from top of stack
SWAP	N/A	0x5F	Swap the two top words on the stack
WIDE	N/A	0xC4	Prefix instruction; next instruction has a 16-bit index

0.2 Project skeleton

To make it easier to automatically test the submissions, we have provided a skeleton layout for the project. You are allowed to change this layout, as long as certain command still run successfully. The skeleton can be found at the following URL: `https://github.com/VU-Programming/IJVM-Skeleton`

0.2.1 Gradle

To build the project we will use *gradle*[3]. By executing `gradle build` in the top level directory of the project, the project will be built. Using `gradle run`, you can run your program. If you do not have gradle installed, do not worry! The template includes a gradle wrapper (`./gradlew`), that will download gradle on-the-fly when executed. You can use the command `./gradlew` instead of `gradle` in all subsequent examples.

If you want to run your program with command line arguments, you can build an executable *jar* using `gradle jar` and run it using `java -jar build/libs/ijvm.jar arg1`.

We have applied the *idea* (IntelliJ) and *eclipse* plugin to the gradle file. If you want to import the project into eclipse, execute `gradle eclipse` in the top-level directory. This will generate a project file that can be imported in Eclipse. Likewise, if you run `gradle idea`, gradle will generate an IntelliJ project for you.

0.2.2 Running the tests

We have provided a set of JUnit test cases for you to test your program with. By passing all these basic tests, and getting a satisfactory grade for style, **you will be able to pass the course**. You can run these tests using `./gradlew test`.

Additionally, your program will also be tested against a number of advanced tests. These advanced tests can also be found in the project skeleton. Passing these tests allows you to obtain a higher grade proportional to the number of passed tests.

To only run a subset of the tests (e.g. to speed stuff up when debugging), you can use the `--tests` flag. For example, to only run the tests for the first module, execute `./gradlew test --tests pad.ijvm.Task1`.

0.2.3 Folder structure

In the provided template we include a number of folders. All your code should be placed in the `src/main/java/pad/ijvm` folder. **Please do not alter this folder structure!**. You are, however, allowed to add extra

[3] `http://www.gradle.org`

packages in this folder. As long as the program compiles with `gradle build` we are happy! In the `interfaces`-folder you can find the `IJVMInterface`. Your main IJVM class should implement this interface, so that we can test your program easily.

0.2.4 Requirements

1. The program should compile, using `gradle assemble`
2. The program should execute the binary `binary.ijvm` when executing the command `gradle jar` followed by
   `java -jar build/libs/ijvm.jar binary.ijvm`.
3. The tests in the folder `src/test` should execute successfully when running `gradle test`.

0.3 Useful Tools

0.3.1 UNIX/ CLI

During this course you will make use of some kind of command line interface. Although we strongly recommend you familiarize yourself with UNIX-like systems (like Linux/BSD/Mac OS X), it is still possible (albeit a bit harder) to use a CLI on Windows. Be sure that you know how to compile Java programs using the command line. If you are not familiar with compiling programs using the command line in Windows have a look at the URL in the footnote.[4]

0.3.2 xxd/ hexdump/ hexedit

Since you will be working with binary files, there are some tools that come in handy when debugging and/ or creating test inputs. One utility, `xxd`, creates a dump of a given file in hexadecimal (if you are not familiar with hexadecimal notation, you should also read up on that a bit[5]).

```
root@machine: xxd -g 1 -c 8 test.ijvm
00000000: 1d ea df ad 00 01 00 00  ........
00000008: 00 00 00 00 00 00 00 00  ........
00000010: 00 00 00 0d 10 31 fd a7  .....1..
00000018: 00 06 10 32 fd 10 33 fd  ...2..3.
00000020: ff                       .
```

Listing 1: Output of xxd on an ijvm binary.

[4] `http://introcs.cs.princeton.edu/java/15inout/windows-cmd.html`
[5] `https://en.wikipedia.org/wiki/Hexadecimal`

Modifying binary files can be done using an editor such as `hexedit`.

0.3.3 MIC-1 Emulator/ IJVM Assembler

Using the provided MIC-1 emulator[6] from Tanenbaum's book, you can assemble programs to test your own IJVM emulator. The assembled files have a .ijvm extension, and follow the same format that we will use in this course. You can also use the emulator to verify the behavior of your implementation.

For your convenience we (or rather *Jur van den Berg*) have written our custom IJVM assembler with a command-line interface, called *goJASM*[7], which gives some more detailed error messages when assembly of a program fails. See the documentation of goJASM for further instructions on how to assemble IJVM programs.

0.3.4 GIT

Using version control is a must for larger projects. During this course you will, incrementally, build a final application. Sometimes you might make a design decision that at a later stage seems detrimental. In such cases it is really useful to be able to revert back to a certain earlier stage. This is where version control systems, such as SVN, Mercurial, and GIT come in handy.

There exist numerous online tutorials on how to use GIT[8][9]. Furthermore, it might be a good idea to use a service like GitHub[10] as a backup for your code (but please *do not* make your code public).

[6] `http://media.pearsoncmg.com/ph/esm/ecs_tanenbaum_sco_6/tanenbaum_sco6.zip`

[7] `https://git.practool.xyz/nova/goJASM`

[8] `http://rogerdudler.github.io/git-guide/`

[9] `https://help.github.com/articles/good-resources-for-learning-git-and-github/`

[10] `https://education.github.com`

1

Binaries, dreaded Binaries!

> **TASK:** 1) Read a filename from the command line arguments. 2) Read that file into a byte array, and extract the relevant IJVM headers from it. 3) Finally, starting at the first instruction, print the names (e.g. POP) of the instructions in the binary to the standard output.

1.1 Command line arguments

In Java, the command line arguments are passed as a `String` array to the `main` method of the called class. For example, invoking the program in Listing 2 with `java HelloWorld myname lastname 1337` would print the following:
`Hello, myname lastname. You entered 1337 as your favorite number.`

```
1 public class HelloWorld {
2
3     public static void main(String argv[]) {
4         System.out.printf("Hello, %s %s. You entered %s as
  ↪ your favorite number.",
5         argv[0], argv[1], argv[2]);
6     }
7 }
```

Listing 2: Example of reading command line arguments.

1.2 Try-catch statement

Some operations, like reading files, may fail if the file cannot be found. In Java, these kinds of situations usually *throw an exception*. The Java

compiler will usually complain if the exception is uncaught. Depending on the situation, different problems raise different exceptions. For example, reading a file can throw the following exceptions: `FileNotFoundException` and `IOException`.

```
try {
    File file =  new File("non-existing.txt"); // read file
} catch (FileNotFoundException e) {
    System.err.printf("%s\n", e.getMessage());
}
```

Listing 3: Example of a try-catch statement.

1.3 Reading files in Java

In the course *Introduction to Programming* you have learned how to parse files using the `Scanner` class in Java. Since the `Scanner` is mainly suitable for parsing text, we will not use it for this course.

Furthermore, since the final program only has to be able to work on the command-line, we will also not use `libUI` (as used in the course Programming).

A file can be read manually in Java by creating a new `File` object and then reading from that file using a `FileInputStream`.

```
File file = new File("/path/to/file.txt");

byte[] bytes = new byte[(int) file.length()];
FileInputStream fileInputStream = new FileInputStream(file);
fileInputStream.read(bytes);
fileInputStream.close();

// The array 'bytes' now contains the file
```

Listing 4: Example of reading a file.

1.4 Debug prints

It can be useful to print some debug information to the console while developing your program. In the next tasks we will test you program against the output printed to the standard output, thus your program might fail tests if you have some debug prints. We suggest you print debug messages

to `stderr` instead of `stdout`. You can achieve this by creating an new `PrintStream` that prints to `System.err`.

1.5 IJVM binaries

In this course we will follow the binary layout of the IJVM binaries generated by the emulator provided with the book [1]. The generated binaries consist of a 32-bit magic number followed by a number of blocks. During this course, you can safely assume that there are only two blocks: the first block is the block containing *constants*, the second block contains the *text* (executable code).

Every block starts with a 32-bit number signifying where to load it in memory. Depending on your implementation, **this can probably be ignored**. After this there is a second 32-bit number, describing the size of the data in the block (the size is denoted in bytes). The rest of the block contains the actual data.

```
binary file = <32-bit magic number> [block]* // 2 blocks: 1) Constants, 2) TEXT
block       = <32-bit origin> <32-bit byte size> <data>
32-bit magic number = 1D EA DF AD
```

1.6 Information about the task

After reading the file, your IJVM should be ready to execute the binary. When calling `run` your IJVM should start *stepping* (i.e. parsing one by one) through all the instructions.

You can test your program by implementing the following: loop through the `TEXT` section of the binary. Print the name of every instruction encountered (i.e. read the text byte by byte). If you encounter a byte that is not an instruction (e.g. an argument to an instruction), skip that byte. Do not worry about arguments that have the same value as an instruction, just print the name of the corresponding instruction. Or, simply put: read one byte, determine instruction, read one byte, determine instruction, etc. Also start planning ahead a bit. For example, now is the perfect time to implement a *program counter* mechanism!

Furthermore, **your main IJVM class must implement the** `IJVMInterface` (this is required so that we can run some automatic tests against your implementation). You must also implement the `MachineFactory` class in such a manner that it returns an instance of your `IJVM` class. The returned IJVM-instance should not have executed yet, but be ready to execute when calling `run()`. After implementing everything from this task you program should pass the provided basic tests for this task.

1.6.1 About the IJVMInterface

Relevant IJVMInterface methods to implement (or start implementing) for this task:

- `step()`
- `run()`
- `getText()`
- `getProgramCounter()`
- `getInstruction()`

The implementation of the other methods can be left empty (e.g. return `null`).

```
BIPUSH
BIPUSH
IADD
OUT
```

Listing 5: The expected output for the binary task1/program1.ijvm.

1.7 Suggested approach

Start by having a look at the classes given in the skeleton. The class `Main` is invoked when your program starts. In this program you should call the method `MachineFactory.createIJVMInstance()` to create a new instance of your `IJVM` class.

In the method `MachineFactory.createIJVMInstance()` you need to load the specified binary into your IJVM instance. For this purpose you can use a separate class (e.g. `BinaryLoader`), which does all the parsing of the input file.

In the `IJVM` you should implement the main functionality of the IJVM interpreter. In the method `run()` you should run the loaded program as long as there is a next instruction to read. You can parse the current instruction using a `switch`-statement.

Finally, to keep track of at which instruction you currently are, you will need a *program counter*, which is a property of the IJVM instance.

2

Stack up!

> **TASK:** 1) Implement the stack memory, and 2) the common operations on the stack. 3) Implement the basic IJVM stack manipulation instructions, such as `IADD`, `ISUB`, and `BIPUSH`. 4) Finally, read a simple provided binary, execute the instructions, and print the whole contents of the stack (in hexadecimal) to the standard output. Also implement the `IN` and `OUT` instructions.

2.1 The stack abstract data type

The *stack* is a data-structure that is essential in Computer Science. Without the notion of a stack many concepts (like recursion) would be impossible. The *abstract data type* of the stack describes four essential operations on a stack:

- **PUSH**. Add an element to the top of the stack.
- **POP**. Remove one element from the stack, and return it.
- **TOP**. Return the element at the top of the stack without removing it.
- **SIZE**. Return the size of the stack.

For more info in the stack abstract data type see the wikipedia page[1].

2.2 Information about the task

Since the IJVM instruction only addresses words (of 4 bytes), or rather all stack operations operate on words, the implemented stack should operate on words. **You, thus, need to implement the `Word` class.** In this class

[1] `https://en.wikipedia.org/wiki/Stack_(abstract_data_type)`

you can add methods and constructors that make the conversion from an array of bytes to a word easier. It is also a good idea to make a method that converts a `Word` to an integer.

When implementing this you should be aware that the `byte` primitive type in Java is a signed byte. When converting a byte array to an integer, you have to do something like the following to get a correct result:

```
1 int result = ((bytes[0] & 0xFF) << 24) | ((bytes[1] & 0xFF) <<
  ↪  16) | ((bytes[2] & 0xFF) << 8) | (bytes[3] & 0xFF);
```

Listing 6: Example of converting a byte array to an int.

It is also very handy to make a utility method, that given a `byte` array, prints it as hexadecimal to the the standard output. It is usually nice to write 4 or 8 bytes per line.

```
0x00 0x00 0x00 0x42 0x00 0x00 0x00 0x42
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

Listing 7: An example of what the output could look like.

For the `OUT` instruction, we want you to output the word at the top of the stack interpreted as ASCII to the `PrintStream` specified by the method `setOutput`. You can, by default, set the output to the standard output. Also, **do not forget to let your Machine class implement the `IJVMInterface`**. Some methods from the interface are not yet relevant (e.g. `getLocalVariable`, and can thus return a value of your choosing (common practise is to return `0` or `null`)).

2.2.1 About the IJVMInterface

Relevant IJVMInterface methods to implement for this task:

- `topOfStack()`
- `getStackContents()`
- `setOutput()`
- `setInput()`

```java
public class Machine implements IJVMInterface {

    private PrintStream out;

    Machine() {
        out = System.out;
    }

    //...

    void setOutput(PrintStream out) {
        this.out = out;
    }

    // ...
}
```

Listing 8: Example of setting the standard output.

2.3 Suggested approach

Since you are going to use a stack, it makes sense to implement a class `Stack`. This stack contains elements of the type `Word`. In the word class it is essential to make methods to convert a `Word` to an integer (so that you can actually perform arithmetic operations on it). Also, it is a good idea to implement a constructor for `Word` that creates a word given a `byte`-array.

Implementing the `getStackContents()` can be a bit tricky, depending on your implementation of the `Stack`. The easiest solution is to add a method `toIntArray()` to the `Stack` class. Another way of implementing it is by making a *copy* of your stack object, and pop off the elements one by one, adding them to the resulting int-array.

3

Controlling the Flow: the GOTO solution!

> **TASK:** 1) Write a method to convert a byte array to an integer so that you can read instruction operands. 2) Implement the GOTO instruction. 3) Finally, implement the other control flow instructions.

3.1 Basic branching

The idea behind the *goto*-instruction is really simple: add an offset to the *program counter*, and continue executing the program from that address. Other instructions such as `IFEQ` only branch if a certain condition is met, otherwise the program just continues to the following instruction. In the example in Listing 9 you can see how the `GOTO` should work.

3.2 Information about the task

Be sure to test your program carefully! There are some small caveats that can cause errors in edge cases. For example, the byte following the `GOTO`-instruction should be interpreted as a signed short (i.e. a signed number consisting of 2 bytes). Also note that the offset is calculated from the beginning of the branching instruction.

Something that should also be taken into consideration, is that the addressing for instructions is done on a *byte-granularity-level*, while all other memory addressing is done on a *word-level*. In other words, the offset after the `GOTO` is an offset in *bytes*, while, for example the *index* argument to `LDC_W` is an offset into the constant pool given in the unit of *words* (4 bytes). Thus, `LDC_W 0x2` pushes the third constant (thus at an offset $0x8$ from the beginning of the constant pool) to the stack.

```
1  .main
2
3  L1:
4        BIPUSH 0x31 // Push '1' to stack
5        OUT         // Print '1'
6        GOTO L3     // Jump to L3
7  L2:
8        BIPUSH 0x32
9        OUT          // Print '2'
10
11 L3:
12       BIPUSH 0x33
13       OUT          // Print '3'
14       HALT
15
16 .end-main
```

Listing 9: Simple GOTO test code. The output of this example should be `13`. The GOTO should skip over label L2, thus jumping to L3.

Table 3.1: Instructions to be implemented for this task

GOTO	short	0xA7	Unconditional jump
IFEQ	short	0x99	Pop word from stack and branch if it is zero
IFLT	short	0x9B	Pop word from stack and branch if it is less than zero
IF_ICMPEQ	short	0x9F	Pop two words from stack and branch if they are equal

3.3 Suggested approach

Since the branching instructions all have as argument a signed short (16 bit integer), it is a good idea to create a method that reads a short at a certain program address. Start by implementing the `GOTO`, as this is the easiest instruction to implement.

```
.main

L1:
      BIPUSH 0xA    // push 10 to stack
L2:
      BIPUSH 0x1
      ISUB          // Subtract 1
      DUP
      IFEQ END      // Jump to end if zero
      BIPUSH 0x31
      OUT           // Print 1
      GOTO L2       // Repeat loop

END:
      HALT
.end-main
```

Listing 10: Slightly more advanced branching test code. The output of this example should be `111111111`.

4

Local variables: Artisan and Organic!

> **TASK:** 1) Implement the constant pool (`LDC_W` instruction). 2) Implement local variables (`ILOAD` and `ISTORE` instructions), 3) the `IINC` instruction. 4) Finally, implement the `WIDE` instruction.

We are now close to having an emulator that can execute simple binaries! After implementing local variables, you should be able to run most simple programs.

4.1 The constant pool

The constant pool is the location in memory which contains read-only constants. These constants are loaded into the constant pool at load-time, and are never changed thereafter. Using the `LDC_W` instruction, a constant from the constant pool can be pushed onto the stack. If you have not yet loaded the constant pool from the binary (see Section 1.5 for the layout of the binaries), please do so now!

Implementing the `LDC_W` instruction is rather straight-forward:

1. Read the argument of the instruction (hereafter called the *index*). This index is represented using an unsigned *short*.
2. Load the word at offset *index* from the constant pool. Note: addressing in the constant pool is always done on a *word-granularity* level.
3. Push the loaded word to the stack.

4.2 The local frame

Implementing local variables is a bit trickier than the constant pool. The first thing to notice is that the local variables reside in the *local frame* of the

current method while the constant pool stays the same, no matter in which local frame you are. When defining a variable in the JAS-file, the assembler gives every local variable a unique label, which happens to be the offset into the local variable frame. For example, if a method has two variables *a* and *b*, variable *a* could get the label 0, while *b* could get the label 1.

When encountering an operation on a local variable, the instruction parameter is the numeric *label*. E.g. when encountering `0x15 0x0` (`ILOAD 0x0`), simply load the first local variable, and push it to the stack.

As you will see in the next chapter, the local frame contains some more info besides local variables. If you have chosen for an object-oriented approach for you implementation, it may be a good idea to make a class `Frame`. By keeping in mind that the memory layout of the IJVM can be viewed as a *stack of local frames*, the implementation of method invocation will be much less work.

4.3 Information about the task

While this task may not seem that hard, a bad design may come back to haunt you when implementing methods. Think ahead! How does your design fit together with method invocations?

Also make sure that you have implemented all the methods from the `IJVMInterface` after finishing this task. It is much easier for you if you can test that your program is correct before continuing to the next stage.

4.3.1 About the IJVMInterface

Relevant IJVMInterface methods to implement for this task:

- `getLocalVariable(int i)`.
- `getConstant(int i)`.

4.4 Suggested approach

As mentioned the local variables reside in a local frame. The obvious choice is, thus, to create a class `Frame` which contains the local variable of the current frame. Furthermore, you can add your program counter as well as your stack to the frame. This will make it much easier to implement method invocations. You could view the frame as the state of your IJVM instance.

5

Call yourself a method!

> **TASK:** Implement the `INVOKEVIRTUAL`, `IRETURN`, and any other instructions that you haven't implemented yet.

5.1 IJVM method invocation

Method invocation on the IJVM can be a bit tricky. It has some strange behavior that are left-overs from the Java Virtual Machine. For example, when invoking a method, the caller has to push an Object-reference to the stack as the first parameter. Since the IJVM does not support different objects, pushing this reference has no meaningful functionality.

Before invoking a method, the caller also pushes the method arguments to the stack. Thereafter `INVOKEVIRTUAL` is called with one argument, which is a reference to a pointer in the constant pool. The pointer in the constant pool in turn points to the first address of the *method area*. The method area first contains two *shorts* (2 byte numbers), the first one signifying **the number of arguments** the method expects, and the second one being **the local variable area size**. The fifth byte in the method area is the actual first instruction to be executed.

5.2 Setting up a local frame

When a method is invoked, a new local frame is created. In this local frame one can find local variables, as well as the value of the previous program counter. Have a look at Figure 5.1 to see what the memory should look like after a method invocation. Since you are building an emulator, the exact memory layout does not need to be the same as in Figure 5.1, however, it is a nice starting point.

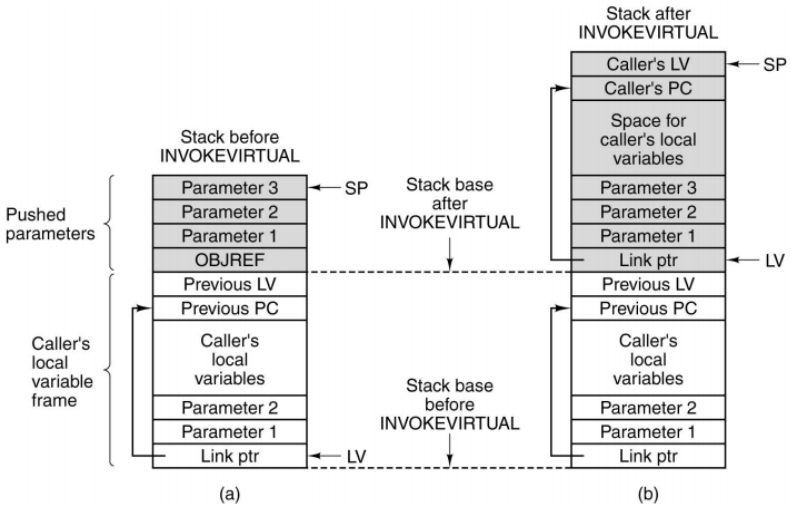
5.3 Returning from a method

At the end of a method, the `IRETURN` instruction is called. When this happens, the currents stack pointer as well as the program counter are restored to the previous value. Finally, the top of the stack of the current frame is returned by overwriting the pushed `OBJREF` with the value, and then pointing the *stack pointer* to this location. Another way of looking at this is that the pushed arguments and the `OBJREF` are removed from the stack of the previous frame, and then a return value is placed at the top of the stack.

5.4 Information about the task

This is by far the hardest task until now. You may have to re-write some of your previous code because of incompatible design decisions. First get a good overview of how the method invocation mechanism actually works, then start implementing it!

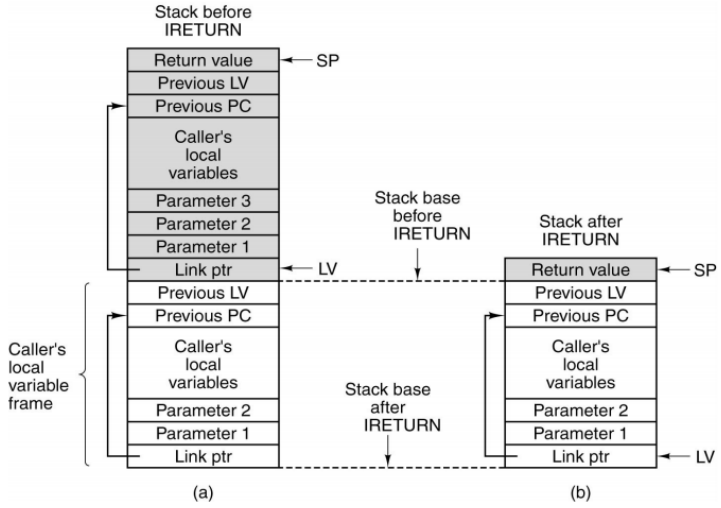
Figure 5.1: The stack before and after INVOKEVIRTUAL.



5.5 Suggested approach

Since you have the `Frame`-class, which represents the current state, invoking a method should set up a new frame. To make it easier to return from a method, you should always keep track of the previous frame. When returning from a method, simple set the current frame to the previous frame, while altering the stack of the previous frame slightly.

Figure 5.2: The stack before and after IRETURN.



6

Even more stuff?! Because why not?

TASK: Implement additional functionality for the IJVM.

Please note: the rest of your program should be working correctly before you start on the bonus. You will not be able to get bonus points if your IJVM implementation does not work correctly.

You can get a total of 20% on your final grade by implementing additional functionality. You can pick and choose what features you want to implement. Some functionality is harder than others (as can be seen by how many points you get for them).

For some of these additional features, we have provided some tests to test your implementation, while others are more open-ended. Please discuss your plan with your TA before starting to implement any of these additional functionalities.

6.1 Heap memory (10%)

Currently all information in the IJVM is saved on the local stack. Sometimes it is necessary to have persistent memory between method calls (this is really necessary of you want to implement something like a database).

For this task you have to implement heap memory by implementing three new instructions that work with heap-allocated arrays. The arguments of these instructions should be placed on the stack in the order in which they appear (i.e. the last argument is at the top of the stack when the instruction is executed).

Table 6.1: IJVM Heap instructions. All the arguments are placed on the stack.

Instruction	OpCode	Args	Description
NEWARRAY	0xD1	count	Create new array on the heap. The count must be of type int. The count is popped of the stack and replaced by an array reference that can be used to refer to the newly created array.
IALOAD	0xD2	index, arrayref	Load the value stored and location index in the array referenced by arrayref.
IASTORE	0xD3	value, index, arrayref	Store value at location index in the array referenced by arrayref.

6.2 Debugger (10%)

For this task you have to write an interactive memory debugger with roughly the same functionality as GDB[1]. The program should start on the command line and give the user a prompt.

Your debugger should have the following commands:

- `help` prints help on how to use the debugger.
- `file <binary>` loads the specified binary.
- `run` run until the next breakpoint is encountered. If the program has already started, stop and start again.
- `input <file>` sets the IJVM standard input to contain contents of specified file.
- `break <addr>` sets a breakpoint for instruction at address `addr`.

[1] `https://www.gnu.org/software/gdb/`

- `step` should perform one instruction.
- `continue` should continue executing until the next breakpoint.
- `info frame` should show the local stack (in hexadecimal) and variables of the current frame.
- `backtrace` should show a call-stack of all frames (i.e. in which order methods have called each other and with what arguments).

Create a new program `IJDB` that reads commands from the command line, and starts up a new IJVM instance that can be debugged when the `run` command is executed.

6.2.1 Debug symbols (additional 10%)

To earn an additional 10% you have to implement handling of debug symbols. The goJASM assembler can add debug symbols to a binary. The debug symbols allows you to break the execution at a certain method or at a certain label. Extend the `break` command such that besides an address, you could also provide a label or a method to break at. E.g. `break myfunc`, should break the execution when the method `myfunc` starts executing.

The debug symbols for methods and labels can be found in the third and fourth block of the binary respectively. The format of these sections is as follows:

```
debugblock = [entry]*
entry = <32-bit instruction address> <symbol name> <null terminator>
symbol name = [char]+
char = <any ASCII letter>
null terminator = '\0'
```

For example, suppose you have a debug section with an entry with address `0x1337` and symbol name `l33tfunction`. If you execute `break l33tfunction`, you should break at the address `0x1337`. The debug symbols for *labels* should be handled in the same manner. Since different methods can have labels with the same identifier, the assembler automatically prepends the method name to the identifier of labels (e.g. label `L1` in `myfunc` becomes `myfunc#L1`).

6.3 Network communication (10%)

As seen in the course *Computer Networks*, most modern applications depend on networked access to other devices. For this assignment you have to extend your IJVM to support *one* network connection. Implement the following instructions:

Table 6.2: IJVM network instruction set. All instruction arguments are placed on the stack

Instruction	**OpCode**	**Args**	**Description**
NETBIND	0xE1	port	Pops port of the stack and starts a network connection on specified port.
NETCONNECT	0xE2	host,port	Opens a network connection to the specified host (ipv4 address encoded in one word) and specified port. This instruction places a boolean on the stack, indicating wether the connection succeeded.
NETIN	0xE3		Reads a character from the active network connection.
NETOUT	0xE4	char	Pops a word from the stack and writes that character to the network.
NETCLOSE	0xE5		C loses the current network connection.

You are allowed to use the built in network stack of Java[2], so you do not have to worry about implementing TCP. You also only need to support one network connection at a given time (so the IJVM has a global network connection, which you can view as an output device). We have provided you with some test-cases that show how the network communication is supposed to work. You can enable these tests by modifying the `build.gradle` file in your project folder.

[2] `https://docs.oracle.com/javase/8/docs/technotes/guides/net/`

Bibliography

[1] Andrew S Tanenbaum. *Structured computer organization*. Pearson, 2006.