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 C, the command line arguments are passed as an array of strings to the `main` function of your program. For example, invoking the program in Listing 4 with `./helloworld myname lastname 1337` would print the following:
`Hello, myname lastname. You entered 1337 as your favorite number.`

```c
#include <stdio.h>
int main(int argc, char **argv)
{
  printf("Hello, %s %s, You entered %s as your favorite
  ↪  number", argv[0], argv[1], argv[2]);
}
```

Listing 4: Example of reading command line arguments.

1.2 Reading files in C

In the course *Computer Programming* you have learned how to parse input from the standard input. Here we will open an actual file on the file-system, using the `fopen` system call wrapper in `libc`. See [1] for a description on

[1] `http://pubs.opengroup.org/onlinepubs/009695399/functions/fopen.html`

`fopen`. Below is a code snippet detailing how to open a file for reading in C.

```c
#include <stdio.h>
#include <unistd.h>

char *binary = "path/to/binary.ijvm"
FILE *fp;
char buffer[128];
fp = fopen(binary, "rb"); // Open read-only

// Read 128 bytes into buffer
fread(buffer, sizeof(char), 128, fp);
```

Listing 5: Example of reading a file.

1.3 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 calling `fprintf`, with the argument `stderr` provided as the file parameter. It can also be useful to use a precompiler macro for debug printing, allowing you to easily toggle debug output. We have included a simple debug print mechanism in our skeleton (in the `include/util.h` header).

1.4 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. 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
```

Listing 6: The layout of an IJVM binary

Note that the x86 architecture (that you are probably using) uses *little-endian* integers, while Java, and thus IJVM, uses *big-endian* integers.

```
static uint32_t swap_uint32(uint32_t num)
{
  return ((num>>24)&0xff) | ((num<<8)&0xff0000) |
  ↪  ((num>>8)&0xff00) |
     ((num<<24)&0xff000000);
}


// When reading integer from IJVM binary
FILE *fp;
uint32_t size;
fread(&size, sizeof(uint32_t), 1, fp);
size = swap_uint32(size);
```

Listing 7: Example of changing endianess for a 32 bit unsigned integer.

1.5 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. running one by one) through all the instructions.

A good starting point for your implementation is 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 program must implement the** functions defined in the `ijvm.h` header file (this is required so that we can run some automatic tests against your implementation). When calling `init_ijvm`, you should

initialize your ijvm instance, but it should not start executing the binary yet. Be ready to execute when calling `run()` or `step()`. After implementing everything from this task your program should pass the provided basic tests for this task.

1.5.1 Functions to implement

Relevant ijvm functions to implement (or start implementing) for this task:

- `init_ijvm()`
- `destroy_ijvm()`
- `step()`
- `run()`
- `get_text()`
- `get_program_counter()`
- `get_instruction()`

The implementation of the other functions can be left empty (e.g., return 0).

```
BIPUSH
BIPUSH
IADD
OUT
```

Listing 8: The expected output for the binary task1/program1.ijvm.

1.6 Suggested approach

Start by having a look at the files given in the skeleton. The c file `main.c` is invoked when your program starts. In this program you should call the function `init_ijvm()` to create a new instance of your machine.

In the function `init_ijvm()` you need to load the specified binary into your IJVM instance. Parse the binary in a structured way; store the contents of the *constants* and *text* blocks. **HINT:** blocks have the same layout, thus you can create a re-usable function that reads a block and stores its data in a buffer.

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*.

HINT: Remember to split modules of your program into separate files; no one want to look at thousand-lines-long C files.

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`, `IAND`, `IOR`, 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

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

on words. **In the context of the IJVM, a word is equal to a 32 bit integer.** In certain cases you need to convert an array of bytes into an integer. We suggest you write some utility functions as early as possible, such as to keep your code clean.

Be aware that the x86 architecture uses *little-endianness* for integers, while Java (and thus the IJVM), uses big-endian integers. You, thus, may need to convert some data when you perform operations on them.

When switching endianness, you have to do something like the following to get a correct result:

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

Listing 9: Example of converting a byte array to an int.

```
static uint32_t swap_uint32(uint32_t num)
{
  return ((num>>24)&0xff) | ((num<<8)&0xff0000) |
↪  ((num>>8)&0xff00) |
    ((num<<24)&0xff000000);
}


// When reading integer from IJVM binary
FILE *fp;
uint32_t size;
fread(&size, sizeof(uint32_t), 1, fp);
size = swap_uint32(size);
```

Listing 10: Example of changing endianess for a 32 bit unsigned integer.

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

For the `OUT` instruction, we want you to output the word at the top of the stack interpreted as ASCII to the output specified by the method `set_ouptut()`. You can, by default, set the output to the standard output. Some methods from the interface are not yet relevant (e.g. `get_local_variable()`,

```
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 11: An example of what the output could look like.

and can thus return a value of your choosing (common practise is to return `0` or `NULL`)).

2.2.1 About the functions

Relevant functions from the interface to implement for this task:

- `tos()`
- `get_stack()`
- `set_output()`
- `set_input()`

```
1  FILE *out;
2
3  void set_output(FILE *file) {
4          out = file;
5  }
6  // ...
7
8  // Prints to the specified file descriptor
9  fprintf(out, "My Output here");
```

Listing 12: Example of setting the standard output.

2.3 Suggested approach

Since you are going to use a stack, it makes sense to implement a stack datatype (i.e., a struct). This stack contains words (integers) as elements.

Implementing the `get_stack()` can be a bit tricky, depending on your implementation of the stack. The function `get_stack()` should return a pointer to the first stack value of the current frame. Since you have not

implemented frames yet, you can simply let it point to the beginning of the stack.

When implementing `IN` you can use the `getc(FILE *)` function (found in `stdio.h`) on the specified input stream (`stdin` by default). Similarly, you can use `fwrite` to write to the specified output stream.

HINT: Have a look at the example C code snippet `calc.c` in the C code examples folder. Also, remember to add some debugging code and error handling (i.e., assert that the stack pointer does not go beyond the allocated area of the stack). Adding these kinds of checks will help you a lot down the line.

3

Controlling the Flow: the GOTO solution!

> **TASK:** 1) Write a function 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 13 you can see how the `GOTO` instruction should behave.

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 bytes 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. In Figure 3.1 the different kinds of IJVM instruction formats are shown.

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 13: 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

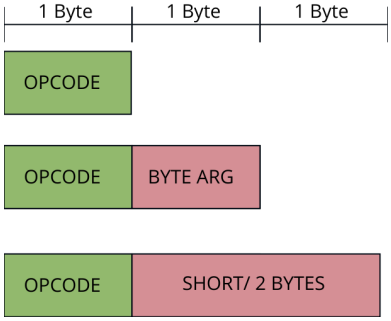
GOTO	short	0xA7	Unconditional jump
IFEQ	short	0x99	Pop word from stack and branch if it is zero
IFLT	short	0x9B	Pop word from stack and branch if it is less than zero
IF_ICMPEQ	short	0x9F	Pop two words from stack and branch if they are equal

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 function that reads a short at a certain program address. Start by implementing the `GOTO`, as this is the easiest instruction to implement.

HINT: some instructions (such as the branching instructions) have arguments that are signed, while other instructions have an unsigned argument. In general, instructions that take an index have an unsigned argument, while the other instructions have signed arguments.

Figure 3.1: Layout of different IJVM instruction formats.




```
1  .main
2
3  L1:
4        BIPUSH 0xA    // push 10 to stack
5  L2:
6        BIPUSH 0x1
7        ISUB          // Subtract 1
8        DUP
9        IFEQ END      // Jump to end if zero
10       BIPUSH 0x31
11       OUT           // Print 1
12       GOTO L2       // Repeat loop
13
14 END:
15       HALT
16 .end-main
```

Listing 14: 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 1.4 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.

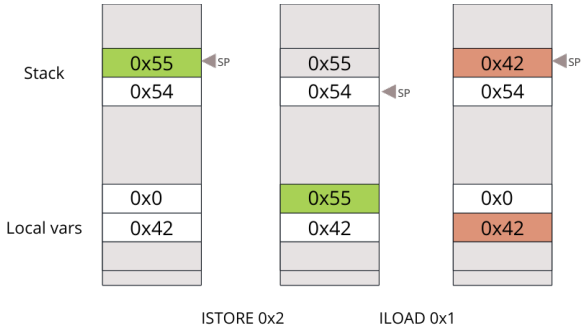


Figure 4.1: Example of an `ISTORE` followed by an `ILOAD`. Note that local variable number 0 is usually a dummy item (`OBJREF`).

As you will see in the next chapter, the local frame contains some more info than local variables. 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 `ijvm.h` interface 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 What to implement

Relevant interface functions to implement for this task:

- `get_local_variable(int i)`.
- `get_constant(int i)`.

4.4 Suggested approach

As mentioned the local variables reside in a local frame. Carefully read how you have to set up your memory layout!

You could, for example, implement frames using a *linked-list* of frames, or you can take a more machine-centric approach by saving the frame in main memory of your machine and utilizing a *frame-pointer* variable (as described in [1]/ see Figure 5.2).

Regarding the `WIDE` instruction, our tests assume that the function `step()` executes the whole wide-prefixed instruction in one step. You can view the `WIDE`-prefixed instructions as one instruction (e.g., `WIDE ILOAD`, `WIDE ISTORE`, or `WIDE IINC`).

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 5.2 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 5.2, however, it is a nice starting point.

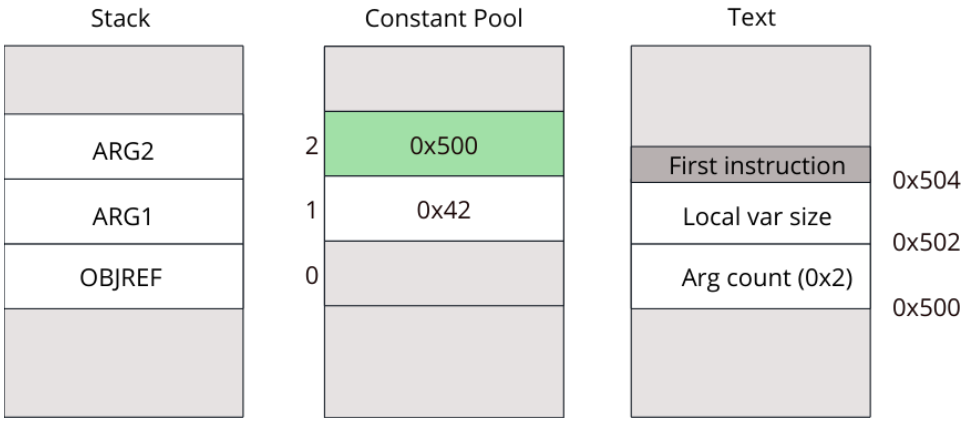


Figure 5.1: Example of executing `INVOKEVIRTUAL 0x2`. The pointer to the method is looked up at constant index 0x2 (with value 0x500). The method area is found at offset 0x500 in the text area. The number of arguments and the local variable size are read. After this the frame is set up, and the first instruction of the method starts executing.

5.3 Returning from a method

At the end of a method, the `IRETURN` instruction is called. When this happens, the current 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.

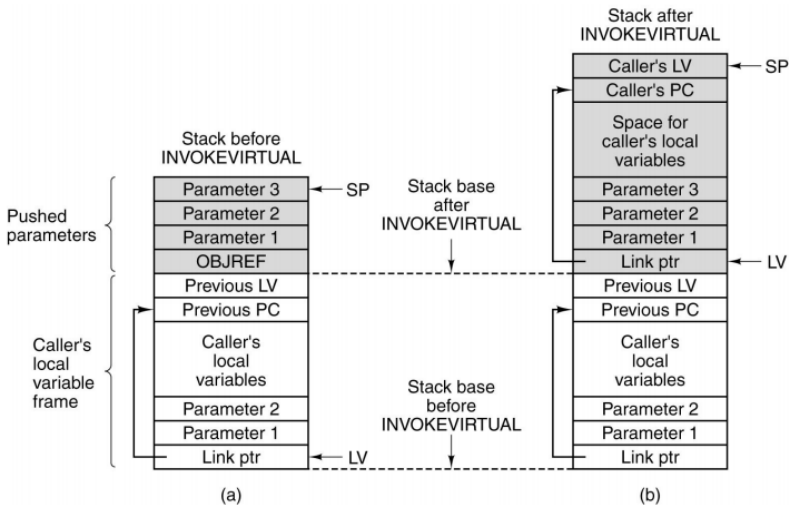
5.4 Information about the task

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! **Note:** the memory layout of your implementation does not exactly need to correspond to the layout presented in the figures in this chapter. You can choose to allocate separate memory objects for each frame, rather than placing everything in one contiguous array.

5.5 Suggested approach

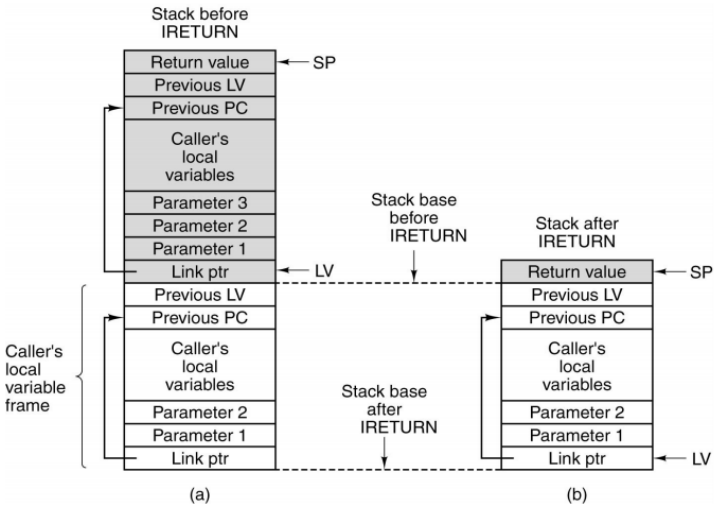
To make it easier to return from a method, you should always keep track of the previous frame. When returning from a method, simple set the current

Figure 5.2: The stack before and after INVOKEVIRTUAL.



frame pointer to the previous frame pointer, while altering the stack of the previous frame slightly by pushing the return values of the method to the stack.

Figure 5.3: 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 (passing basic tests and *preferrably* all of the advanced tests) before you start on the additional functionality. You will not be able to get additional points if your IJVM implementation does not work correctly.

You can get a total of 30% on your final grade by implementing additional functionality. You can pick and choose what features you want to implement, but keep in mind that some functionality is harder to implement 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 (5%)

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 if 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.

You should also implement **bounds checking** for the array instructions. For example, if you use an index for the `IALOAD` instruction that goes beyond the length of the referenced array, you should print an error and halt your virtual machine.

6.1.1 Garbage collection (additional 10%)

Implement garbage collection [1] for your allocated heap data. You are free to choose how to implement this. You can, for example, keep track of existing references to arrays or implement a simple *mark and sweep* mechanism. The garbage collection should be transparent to the user, but also include an instruction `GC` (`0xD4`) that manually triggers garbage collection. Also include some documentation (in a readme file) on how you implemented your garbage collection strategy.

[1] `https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)`

6.2 Debugger (10%)

For this task you have to write an interactive memory debugger for your virtual machine with roughly the same functionality as GDB[2]. 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`.
- `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.

Note: you are allowed to use the `readline` library for command line parsing.

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:

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

```
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 Graphical user interface (10%)

Extend your IJVM with a graphical user interface, which allows a user to load a binary, reset the IJVM instance, and allows the user to give enter input and read output. See the WebIJVM [3] for inspiration.

There exist many libraries that allow you to write Graphical User Interfaces in C. One example of a simple library is Nuklear [4]. Remember to document (in the README) what libraries you used and how to install them.

Add a new rule in your Makefile that builds a binary called `ijvm-gui`, that should build with `make ijvm-gui`. Make sure that your binary works on Linux (you will get more points if you make it portable to other operating systems).

Possible features (pick at least 3):

- Allow the user to select a binary to be executed
- Allow the user to reset the state if the IJVM instance and restart the execution
- Allow the user to enter input in a text field and read the output
- Allow the user to select an input file to be used for input instead of waiting for keyboard strokes
- Allow saving the IJVM output to a file of choice
- Display information, such as the program counter and stack contents
- Make your GUI portable, so that it works on other operating systems

[3] `https://apps.blacknova.io/webijvm/`
[4] `https://github.com/vurtun/nuklear`

6.4 Network communication (5%)

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 listening for a network connection on specified port. Places the value `0` on the stack on failure, otherwise places a value greater than `0` on the stack.
NETCONNECT	0xE2	host,port	Opens a network connection to the specified host (ipv4 address encoded in one word) and specified port. Places the value `0` on the stack on failure, otherwise places a value greater than `0` on the stack.
NETIN	0xE3	netref	Pops netref from the stack and reads a character from the active network connection.
NETOUT	0xE4	netref,char	Pops netref and a word from the stack and writes that character to the network.
NETCLOSE	0xE5	netref	Closes the current network connection.

You are allowed to use the built in network functions in glibc[5], 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

[5] `https://www.gnu.org/software/libc/manual/html_node/Sockets.html`

with some test-cases that show how the network communication is supposed to work. You can run these tests by looking at the `Makefile` in your project folder. Make sure that your network functionality works on Linux.

6.4.1 Multiple connections (additional 5%)

Modify your network instruction implementation to support multiple concurrent open connections. Use the `netref` argument as an identifier for the network connection. Modify `NETBIND` and `NETCONNECT` to place a `netref` on the stack instead of the success value (while still using 0 as a failure status code). Modify `NETIN` and `NETOUT` to make use of the `netref` to distinguish between sending data over different network connections.

6.5 Hardening your code (10%)

By default – mainly due to limited time – we do not expect you to catch all possible errors in the program. Our standard policy is you should make a best-effort to check for errors. To get additional points on your grade, you can choose to harden your implementation. Make sure your program does not crash due to memory errors (i.e., segmentation faults) even when given an invalid/ malicious binary.

An example of an easily exploitable part is the offsets used in the binary. If the index value of the 'ISTORE' is not checked, this could be used to diver the control flow of the program.

Make sure to check all sizes of values that you read, check for overflows, etc. Make use of tools such as fuzzers[6] to find bugs in your implementation. Include some text in your README, documenting what you did to harden your code. To obtain these points your program needs to compile with 'make pedantic' and should also pass the 'make testsanitizers' tests. Also describe your development process (i.e., how did you find the errors in your code, what inputs where able to crash your program, etc.) and how your changes might affect performance (if applicable).

Requirements:

1. Set up a *fuzzer* (e.g., AFL [7]) to find bugs. Let it run for about an hour or so.

2. Write a *README* on what you did to mitigate bugs. Did you apply any preemptive mitigations against such bugs? What is the tradeoff of these mitigations? The main weight of the additional feature lies with analysing what the attack surface is and how to mitigate possible exploitation.

6.6 Snapshots (10%)

One benefit of virtual machines is that the machine is (suprise) *virtualized*. This allows for nice features, like migration and snapshots. For this additional functionality you have to save the state of your IJVM instance to a file when receiving a `SIGINT` signal (Ctrl + c). You then have to be able to start up a new IJVM instance (add a command-line option to your ijvm program) from the serialized file, continuing at the state where you saved the snaphsot.

It is up to you to decide the format of the serialized file. Since dumping all of the memory takes up a lot of space (up to 4 GB), you might want to introduce some compression tricks.

[6] `https://en.wikipedia.org/wiki/Fuzzing`
[7] `http://lcamtuf.coredump.cx/afl/`