Home.md

Corewar guide

This is a translation of @VBrazhnik Corewar wiki-page

1. Introduction

2. Resources

3. Assembly

4. Lexical Analysis

5. Assembly to byte-code

6. Disassembling

7. Virtual machine

8. Visualization

9. TABLES

Introduction.md

The story in short

In 1984, an article by Andrew Dewdney appeared in Scientific American magazine describing the game Core Wars . The article began with the following words:

> Two programs in their natural habitat - computer memory - chase each other from address to address. Sometimes they track down the enemy; sometimes they lay batteries of digital bombs; sometimes they copy themselves to another place of memory to avoid danger or stop to repair the damage done by the enemy. I call this game "Core Wars" ...

Source: Chapter 2. Part 1. The battle on the "ferromagnetic cores" or MARS - the god of war.(www.useless.school)

Corewar Project Overview

The Corewar project is based on the popular game and consists of the three mandatory parts:

- The Champion
- The Assembler)
- The Virtual Machine

The Champion

The objective of this section is to write code in assembly language, which will then be placed in a file with the extension `.s` .

> In fact, the code is written in a pseudo-assembly language. That is, in a language created specifically for this task, which is similar to a real assembly, but still it is not. For consistency with the text of the assignment, simplicity and in order to save six letters, we will also call this language assembly.

The generated code is our champion. Its' goal is to fight against other champions written by us or by other people.

The champion code has the following structure:

1. Name
2. Comment
3. Executable code

It may look like this:

```
.name       "Batman"
.comment    "This city needs me"

loop:
        sti r1, %:live, %1
live:
        live %0
        ld %0, r2
        zjmp %:loop
```

In this project, we have no goal to create the most powerful and invincible champion. This is the task for a completely different project called “Corewar Championship”.

For the Corewar project, we create our champion only to demonstrate understanding of the topic and the ability to write assembly code.

Our task is to write code without errors so that the program asm can turn it into byte-code, which the virtual machine would then execute.

However, our champion should be able to win the `zork` , which is `the basic living program` .

Assembler

The objective of this section is to create a program that will translate the champion code written in assembly language into byte-code - a bunch of numbers.

> From elvish this task can be interpreted as "translate commands from a language understandable to humans (assembly) into a language understandable to a virtual machine (byte-code)."

> **Program translation** is the conversion of a program presented in one of the programming languages into a program in another language. The translator usually also performs error diagnostics, generates identifier dictionaries, prints program text, etc.

Source: Translator - Wikipedia

That is, we must create a program with a name `asm` (from the word “assembly”), which will receive as a parameter a file with assembly language code and create a new file with byte-code.

The file with the code of our champion, written in assembly, must have the extension `.s`. On its basis, the program `asm` will create a new file with the extension `.cor` containing byte-code.

The name of the file itself will remain unchanged. That is, after calling the command `./asm batman.s`, a file `batman.cor` should appear next to the file `batman.s`, if no errors occurred during the translation, obviously.

Virtual machine

After we get the byte-code file, time comes to run the virtual machine.

The executable file of the virtual machine should be called `corewar`.

The task of the vm is to allocate memory and place the champions' code in that memory, as well as the cursors which will execute the champions' code.

Bonuses to the project

As always, the number of bonuses and their essence is limited solely by the author’s imagination.

Here are a few that could be implemented:

Extended error messaging

If an error occurred during the translation, it would be nice the program `asm` to behaves like a real translator (which it actually is). That is, to print out a meaningful message - number of line in the file `.s` and the type of error.

Ability to disassemble byte-code

To make it possible to obtain assembly code from a file with byte code. That is, implement the function inverse to the `asm`.

Visualizer

Create a program that will display the state of memory, as well as changing key game parameters during the battle.

You can also add various sound effects to focus on key points such as carriage death or a winner announcement.

Resources.md

The Corewar project page provides links for downloading the following files:

Archive `vm_champs.tar`

In this archive you can find the compiled executable file of the virtual machine `corewar`, the compiled compiler program `asm`, as well as many champions in the form `.s` and `.cor` files.

Translator `asm`

The project assignment says that the program `asm` translates code from the assembly language from the `.s` file into byte-code, which should be placed into a file with the extension `.cor`.

But in fact, the provided program does not pay any attention to the file extensions. It only searches for the last dot in the name and replaces all further contents with an extension `.cor`.

The following situations, which are not errors in the usual sense, but still violate the declared workflow, are possible:

```
$ ./asm batman.cor
Writing output program to batman.cor
```

```
$ ./asm batman
Writing output program to .cor
```

```
$ ./asm dc.heroes/batman
Writing output program to dc.cor
```

Also, the provided program does not limit the number of champions accepted for processing.

You can specify a lot of files with a variety of extensions as arguments at one time. True, only the last of them will be processed:

```
$ ./asm ant-man.s iron_man.s batman.s
Writing output program to batman.cor
```

`asm` ignores all other arguments:

```
$ ./asm --undefined-flag incorrect_file.s batman.s
Writing output program to batman.cor
```

How to deal with it?

Since such behaviour is not described in the text of the task, you can limit the number of accepted arguments to just one file, which must have an extension `.s`.

This will make the work `asm` more transparent and eliminate the above vulnerabilities in workflow.

> By the way, the virtual machine does not check incoming files for the `.cor` extension.
>
> Extension check could still be added to comply with a single style of behaviour for all system components.

Champions

There are also problems with the provided examples of champions.

Unfortunately, not all files were written correctly. Therefore, the program `asm` will not be able to translate some of them into bytecode and will give an error.

How to deal with it?

There will be no problems with the files from the folder 'champs/examples', but you will have to verify other files personally.

Files `op.c` and `op.h`

The file `op.c` contains a structure that describes each operation defined in assembly language.

The data given in this structure can be overwritten in any form convenient for us and placed in the project, no restrictions have been established.

The file `op.h` contains important preprocessor constants that determine the parameters of the virtual machine, as well as assembler syntax.

This file should be handled more carefully and included in the project. Consider making only the most necessary changes, because its' availability is one of the conditions for participation in the project **“Corewar Championship”**:

> It will be executed on our own virtual machine, so the configuration will be the one you will describe in your file `op.h` that will be attached.

The necessary changes mean compliance with Norm.

If you don't plan to particiate in the championship, then you shouldn't worry about the fate of the file `op.h`.

Assembly.md

Assembly syntax

The assembly language has the rule "In one line - one instruction."

> An instruction or statement is the smallest autonomous part of a programming language; team or set of commands. A program is usually a sequence of instructions.

Source: Statement - Wikipedia

Empty lines, comments, as well as extra tabs or spaces are ignored.

Comment

There is a constant `COMMENT_CHAR` in `op.h`. It determines which character indicates the beginning of the comment.

In the provided file it is the hashtag — `#`.

That is, everything between the character '#' and the end of the line is considered a comment.

A comment can be located anywhere in the file.

Example #1:

```
# Codam
# is a programming school
```

Example #2:

```
ld %0, r2    # And it is located in the Marine Terrein
```

Alternative comment

In the provided archive `vm_champs.tar` in directory `champs/examples` there is the file `bee_gees.s` with the champion code, which the original program `asm` translates into byte-code without errors.

There are two kinds of comments in the code of this champion:

- standard, which was discussed above;
- alternative, about which there is no information in subject.

Instead of octotorp (`#`) here is used `;`.

An example of using a comment of this type:

```
sti r1, %:live, %1    ; Marine Terrein is placed in Amsterdam, Netherlands
```

How to deal with it?

This type of comment is not described in the subject, but it is supported by the original translator. Therefore we most likely do not have to support it.

But let's add it into the file `op.h`:

```
# define ALT_COMMENT_CHAR    ;
```

> It will be the only (excepting the norme) change we make to file `op.h` .

Champion name

Name of the champion must be defined in the file .s. There is a command in assembly, saved in the constant `NAME_CMD_STRING` . In the provided file `op.h` it is defined as `.name` .

Command `.name` must be followed by a string containing champion name:

```
.name    "Batman"
```

The length of the name must not exceed the value defined in the constant `PROG_NAME_LENGTH` . In the provided file it equals to `128` .

An empty string is a valid champion name:

```
.name    ""
```

But lack of the string is an error:

```
.name
```

Champion comment

Also, in the file `.s` must be present champion's comment.

Command for comment is defined in the constant `COMMENT_CMD_STRING` in the file `op.h` as `.comment` .

The length of comment is restricted by the value of constant `COMMENT_LENGTH` and must not exceed `2048` .

Command `.comment` is very similar to `.name` and behaves identically in case of empty string and abscence of the string.

Other commands

Some of the provided `.s` example files contain the command `.extend` .

This command and all other commands excepting `.name` and `.comment` , are not described in the subject and are detected as an error by the original `asm` compiler.

Executable code

Champion's executable code consists of instructions.

Assembly language has a rule `one instruction per line` . The new line character `\n` means both end of line and end of instruction. So instead of `;` as for C language, we will use `\n` .

This role means that **even after the last instruction there must be a new line character**. Otherwise the `asm` will display an error message.

Each instruction consists of several components:

Label

Label consists of characters defined in the constant `LABEL_CHARS` . In `op.h` they are: `abcdefghijklmnopqrstuvwxyz_0123456789` .

Label can not containt characters other than those defined in the constant `LABEL_CHARS` .

Label must be followed by a character defined in constant `LABEL_CHAR` . In `op.h` it is `:` .

Why do we need labels?

Label points to the instruction which immediately follows this label. Label points to **one singe instruction** and not to the block of instructions.

```
.name       "Batman"
.comment    "This city needs me"

loop:
        sti r1, %:live, %1    # <-- operation 'sti' is pointed to by the label 'loop'
live:
        live %0               # <-- operation 'live' is pointed to by the label 'live'
        ld %0, r2             # <-- and this operation is not pointed to by any labels
        zjmp %:loop
```

Labels make our life easier, by making coding process easier.

How do labels work?

As we know, the code writtedd in assembly language will be transformed into byte-code. The virtual machine will work with the byte-code.

Let's assume we want to create a loop, in which operation `live` will be performed. We have operation `zjmp`, which can send us a few bytes forward or backward from current posistion.

To create a loop we have to give to operation `zjmp` a value. But what is the value? To find it out, we have to calculate how many bytes in the bytecode the operation code and its argument will use.

As we know, operation code is always 1 byte, and operation `live` has a single argument of size 4 bytes.

It turns out we need to go back 5 bytes back:

```
live %1
zjmp %-5
```

Not so difficult, but these calculations take time and it would be much easier to "switch to the operation `live`". This is what labels are made for.

We simply create a label `loop` and write it as `zjmp` operation argument:

```
loop:    live %1
         zjmp %:loop
```

Now it is the translators' job to calculate the 'distance' between operations. It will calculate how many bytes to jump back and will insert this value in the byte-code.

Examples

Valid labels:

```
marker:
        live %0


marker:


        live %0


marker: live %0
```

All labels from the examples above have the same value and meaning. You may choose any style you want.

Many labels for one operation

Another valid example:

```
marker:

label:
        live %0
```

This means label `marker`, and label `label` point to a single operation.

Pointing to nothing

Valid example:

```
marker:
# End of file
```

In this case label points to the end of executable code.

Important to have a `\n` at the end of line. Otherwise translator will print an error message.

Operations and their arguments

Assembly language consists of 16 operations. Each operation has 1 to 3 arguments.

Information about operation names and about arguments they receive is provided in the file `op.c`.

Operation code	Operation name	Argument #1	Argument #2	Argument #3
1	live	T_DIR	—	—
2	ld	T_DIR / T_IND	T_REG	—
3	st	T_REG	T_REG / T_IND	—
4	add	T_REG	T_REG	T_REG
5	sub	T_REG	T_REG	T_REG
6	and	T_REG / T_DIR / T_IND	T_REG / T_DIR / T_IND	T_REG
7	or	T_REG / T_DIR / T_IND	T_REG / T_DIR / T_IND	T_REG
8	xor	T_REG / T_DIR / T_IND	T_REG / T_DIR / T_IND	T_REG
9	zjmp	T_DIR	—	—
10	ldi	T_REG / T_DIR / T_IND	T_REG / T_DIR	T_REG
11	sti	T_REG	T_REG / T_DIR / T_IND	T_REG / T_DIR
12	fork	T_DIR	—	—
13	lld	T_DIR / T_IND	T_REG	—
14	lldi	T_REG / T_DIR / T_IND	T_REG / T_DIR	T_REG
15	lfork	T_DIR	—	—
16	aff	T_REG	—	—

Understanding of operations and how they work with the arguments of different types is the base of understanding of "Corewar".

Operations and their arguments

Arguments

There are three types of arguments:

1. Registry — `T_REG`

Registry is a variable where we can store some data. The size of registries in octets is defined in the file `op.h` in the constant `REG_SIZE` and is equal to `4`. A registry is a part of cursor (process), but this will be discussed later.

> An octet in computer science is eight binary digits.

Source: Octet — Wikipedia

Amount of registries is defined in the constant `REG_NUMBER` with the value `16`, so available regisries are `r1`, `r2`, `r3` ... `r16`.

> **Registry values**
>
> During the startup of the virtual machine all the registries, excepting `r1`, will be initialized with the value of 0.
>
> Registry `r1` will contain the negated number of champion.
>
> This number is unique and is required for the operation `live` to report champion alive.
>
> Cursor placed at the beginning of the player `2`, will have in it's registry `r1` value `-2`.
>
> If operation `live` is executed with the argument `-2`, virtual machine will know player 2 is alive:
>
> ```
> live %-2
> ```

2. Direct — `T_DIR`

Direct argument consists of special character defined in the constant `DIRECT_CHAR` (`%`) and a number or label, which represents **direct value**.

If there is a label in the argument, there must be the character defined in the constant `LABEL_CHAR` (`:`) in front of it:

```
sti r1, %:marker, %1
```

What are direct and indirect values?

> Let's consider a simple example.
>
> We have a number `5`. Direct value means we just use `5` as is. That is, `5` is `5`.
>
> If `5` is an indirect, it reprsents a relative address, pointing five bytes forward from the current position.

Direct and indirect labels

> What is the difference between labels and numbers?
>
> Actually there is no difference at all. The labels are transformed into their numrical equivalents by translator at the compiling stage.
>
> This means labels are numbers, written in the form of words in `.s` file but replaced by numerical values by translator.
>
> Process of label replacement is describer in the chapter "Why do we need labels?»".

3. Indirect — `T_IND`

Argument of type indirect can be a number or a label, which will represent an **indirect value**.

If argument of type `T_IND` contains a number, there is no need in auxiliar symbols:

```
ld    5, r7
```

If indirect argument has a label as value, it must have `LABEL_CHAR` (`:`) in front of it:

```
ld    :label, r7
```

Delimiter character:

To separate arguments a delimiter character is used. It is defined in `SEPARATOR_CHAR` constant as `,` :

```
ld    21, r7
```

Operations

Operation `live`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
1	`live`	`T_DIR`	—	—

Description

Objectives of the operation:

1. Reports that **cursor**, which performed this `live` operation, is alive.
2. If argument of operation `live` is identical to the number of player saved in the cursor's registry `r1` , it **reports that player is alive**. If registry `r1` of the carriage executing operation `live` equals `-2` , and the argument of opeartion `live` equals to `-2` , **virtual machine counts player `2` as alive**.

> **What is a cursor?**
>
> Here is a brief explanation of the notion. In more detail it will be described in the chapter "Virtual machine".
>
> A cursor is a process that executes the instruction on which it stands.
>
> Let's assume we run our virtual machine with players loaded into it's memory. Every player will obtain a cursor (a process), which will be placed at the beginning of the players code.
>
> 3 champions. 3 sectors of memory with loaded executable code. 3 cursors (processes).
>
> Each cursor contains some information:
>
> - `PC` (Program Counter) - position of the cursor (process) in memory
> - Registries (`r1` ... `r16`) - variables that store cursor's data; their amount is defined by the constant `REG_NUMBER`
> - Flag `carry` - a special `boolean` variable that affects operation `zjmp` and can take one of the two values `1` and `0` .
> - Number of the cycle in which cursor reported `live` last time - this information is used to check whether cursor is still alive or not.
>
> In fact, cursor contains more information, but it will be discussed later.

Operation `ld`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
2	`ld`	`T_DIR` / `T_IND`	`T_REG`	—

Description

This operation loads value into the registry of the cursor. But its behavior depends on the type of the first argument:

- **How it works with the Argument #1 of type `T_DIR`**

If the first argument is `T_DIR`, then it will be loaded into the registry as is.

Actions performed:

1. Write the number from the first argument into the registry received as second argument.
2. If the written value is `0`, then set `carry` to `1`; if the written vaue is `!0` then set `carry` to `0`.

- **How it works with the Argument #1 of type `T_IND`**

If the first argument is of type `T_IND`, it represents an address.

If we receive argument of this type, we must truncate it with a modulo operation: `<FIRST_ARGUMENT> % IDX_MOD`.

> **What is `IDX_MOD`?**
>
> `IDX_MOD` is another constant from the file `op.h`. It's value is defined as `(MEM_SIZE / 8)`, where `MEM_SIZE` is the size of memory in bytes. The virtual machine's memory size is `MEM_SIZE`. In this memory the champions will fight.
>
> What is the constant `IDX_MOD` for? It is used to limit the maximum distance a cursor can jump in the memory. In the file `op.h` the `MEM_SIZE` is initialized with the value `(4 * 1024)`, so `IDX_MOD` equals to `512`.
>
> This means a cursor can't move more than 512 bytes away from the current position.

> After argument of type `T_IND` has been truncated, we use it as a relative address in memory - how many bytes forward or backward relative to the current position of the cursor is the position we need.

Actions performed:

1. Calculate address: current position of the cursor + `<FIRST_ARGUMENT> % IDX_MOD`.
2. Read **four bytes** starting from the obtained address.
3. Write the value from the step 2 into the registry passed as the second argument.
4. If the written value is `0`, then set `carry` to `1`; if the written vaue is `!0` then set `carry` to `0`.

> **Why do we read exactly four bytes?**
>
> The registry size and direct value size are defined as 4 in the file `op.h`:
>
> ```
> # define REG_SIZE    4
> # define DIR_SIZE    REG_SIZE
> ```
>
> We go to the address we've calculated with the argument of type 'T_IND' to read a value. To read the value "as is". That is, to read a value of type `T_DIR`. And we have to save the read value into the registry. To guarantee the success of the operation, the size of read number must be compatible to the size of the registry.
>
> Also, later we will discover that it is possible to write values from the registry to the address in memory. That is why the size of number read from the memory and size of the registry must be compatible in both directions.
>
> So we read as many bytes as the registry can store.

Operation `st`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
3	`st`	`T_REG`	`T_REG / T_IND`	—

Description

This operation writes the value from the registry passed to it as the first parameter, but destination depends on the type of the second argument:

- **How it works with the Argument #2 of type `T_REG`**

```
st    r7, r11
```

In this case value from the registry 7 is written to the registry 11

- **How it works with the Argument #2 of type `T_IND`**

Type `T_IND` is related to the memory addresses, so `st` workflow is:

1. Truncate indirect value by modulo: `% IDX_MOD`.
2. Calculate the address: current address of the cursor + `<SECOND_ARGUMENT> % IDX_MOD`.
3. Write the value from the registry passed as first argument into the memory address calculated in the previous step.

Operation `add`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
4	`add`	`T_REG`	`T_REG`	`T_REG`

Description

Arguments of this operation are of the same type `T_REG`.

- **How it works:**

1. Summ up value of registry number passed to it as first argument with the value of the registry number passed as second argument.
2. Write the result of sum into the registry number passed to it as third parameter.
3. If the written value is `0`, then set `carry` to `1`; if the written vaue is `!0` then set `carry` to `0`.

Operation `sub`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
5	`sub`	`T_REG`	`T_REG`	`T_REG`

Description

Arguments of this operation are of the same type `T_REG`.

- **How it works:**

1. Substract from the value of registry number passed to it as first argument, the value of the registry number passed as second argument.
2. Write the result of sum into the registry number passed to it as third parameter.
3. If the written value is `0`, then set `carry` to `1`; if the written vaue is `!0` then set `carry` to `0`.

Operation `and`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
6	`and`	`T_REG / T_DIR / T_IND`	`T_REG / T_DIR / T_IND`	`T_REG`

Description

BITWISE AND - Wikipedia

- **How it works:**

It performs a bitwise AND operation for the values of the first two arguments and writes the result into the register passed as the third argument

If the written value is `0`, then set `carry` to `1`; if the written vaue is `!0` then set `carry` to `0`.

First and second arguments can be of different types. Here is how to get values we need:

- Argument #1 or Argument #2 — `T_REG`

In this case we take the value from the registry passed as argument.

- Argument #1 or Argument #2 — `T_DIR`

In this case we take the value passed as argument.

- Argument #1 or Argument #2 — `T_IND`

Calculate the address where to read the value from: current cursor position + `<ARGUMENT> % IDX_MOD`.

Read four bytes from the memory starting with the address calculated in the previous step. It will be the value we need.

Operation `or`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
7	`or`	`T_REG / T_DIR / T_IND`	`T_REG / T_DIR / T_IND`	`T_REG`

Description

BITWISE OR - Wikipedia

- **How it works**:

Identical to the operation AND described above.

Operation `xor`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
8	`xor`	`T_REG / T_DIR / T_IND`	`T_REG / T_DIR / T_IND`	`T_REG`

Description

BITWISE XOR - Wikipedia

- **How it works**:

Identical to the operation AND described above.

Operation `zjmp`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
9	`zjmp`	`T_DIR`	—	—

Description

This is that function affected by the value of the flag `carry`.

- **How it works**:

If `carry` is equal to `1`, this operation performs a 'jump'. It moves cursor to the address: current position + `<FIRST_ARGUMENT> % IDX_MOD`.

This operation allows us to jump back and forth to different places in memory and not to execute everything by order.

If `carry` is equal to `0`, 'jump' is not performed.

Operation `ldi`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
10	`ldi`	`T_REG / T_DIR / T_IND`	`T_REG / T_DIR`	`T_REG`

Description

This operation saves a value into the registry that was passed as third argument.

- **How it works**:

To get the value, it has to read four bytes from the memory. The address of the bytes to be read is calculated as follows:

```
current position + `(<VALUE_OF_FIRST_ARGUMENT> + <VALUE_OF_SECOND_ARGUMENT>) % IDX_MOD`.
```

First and second arguments can be of different types. Here is how to get values we need:

- Argument #1 or Argument #2 — `T_REG`

Value from the regisry number passed as argument

- Argument #1 or Argument #2 — `T_DIR`

Value passed as argument

- Argument #1 — `T_IND`

Read four bytes from the address: current position `+ <FIRST_ARGUMENT> % IDX_MOD` .

Operation `sti`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
11	`sti`	`T_REG`	`T_REG / T_DIR / T_IND`	`T_REG / T_DIR`

Description

This operation writes the value from the registry that was passed as first argument.

- **How it works**:

Address in memory to write the value to: current position `+ (<VALUE_OF_SECOND_ARGUMENT> + <VALUE_OF_THIRD_ARGUMENT>) % IDX_MOD` .

How to get values is described above.

Operation `fork`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
12	`fork`	`T_DIR`	—	—

Description

Operation `fork` creates duplicate of current cursor (process) and places it to the address `<FIRST_ARGUMENT> % IDX_MOD` .

** The duplicate cursor is identical to the initial one excepting the position**.

Operation `lld`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
13	`lld`	`T_DIR / T_IND`	`T_REG`	—

Description

It is a more powerful version of operation `ld` (see above).

The only difference between them is that in case of `T_IND` type for the first argument, we have to read four bytes from the address: current position + `<FIRST_ARGUMENT>` . **No modulo truncation required**.

> **the problems of original virtual machine**
>
> Original VM `corewar` , unfortunately malfunctions, and reads two bytes instead of four. Perhaps a this bug is explained by the same lines as the problems in the provided files:
>
> > ... we might have mistaken a bottle of water for a bottle of vodka.

With the argument of type `T_DIR` it's identical to operation `ld` .

Operation `lldi`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
14	`lldi`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	`T_REG`

Description

It is a more powerful version of operation `ldi` (see above).

We have to read four bytes from the address: current position + `(<VALUE_OF_FIRST_ARGUMENT> + <VALUE_OF_SECOND_ARGUMENT>)` . **No modulo truncation required**.

> **To get values of arguments of type `T_IND` we still need to make modulo:**
>
> <VALUE_OF_FIRST_ARGUMENT>: read four bytes from current position + `<FIRST_ARGUMENT> % IDX_MOD`

Operation `lfork`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
15	`lfork`	`T_DIR`	—	—

Description

It is a more powerful version of operation `fork` (see above).

No modulo truncation required.

Operation `aff`

Operation code	Operation name	Argument #1	Argument #2	Argument #3
16	`aff`	`T_REG`	—	—

Description

This operation takes the value from the registry passed as argument, casts it to the type `char` and prints ot to the standard output.

```
(char)(value)
```

`aff` in original `corewar`

In the original vm `corewar` by default `aff` is switched off. To see it's output we have to use flag `-a` .

Operations table

Code	Name	Argument #1	Argument #2	Argument #3	Changes `carry`	Description

Code	Name	Argument #1	Argument #2	Argument #3	Changes `carry`	Description
1	`live`	`T_DIR`	—	—	No	alive
2	`ld`	`T_DIR` / `T_IND`	`T_REG`	—	Yes	load
3	`st`	`T_REG`	`T_REG` / `T_IND`	—	No	store
4	`add`	`T_REG`	`T_REG`	`T_REG`	Yes	addition
5	`sub`	`T_REG`	`T_REG`	`T_REG`	Yes	subtraction
6	`and`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`	Yes	bitwise AND (`&`)
7	`or`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`	Yes	bitwise OR (`|`)
8	`xor`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`	Yes	bitwise XOR (`^`)
9	`zjmp`	`T_DIR`	—	—	No	jump if non-zero
10	`ldi`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	`T_REG`	No	load index
11	`sti`	`T_REG`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	No	store index
12	`fork`	`T_DIR`	—	—	No	fork
13	`lld`	`T_DIR` / `T_IND`	`T_REG`	—	Yes	long load
14	`lldi`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	`T_REG`	Yes	long load index
15	`lfork`	`T_DIR`	—	—	No	long fork
16	`aff`	`T_REG`	—	—	No	aff

Cycles before execution (cycles to wait)

One more detail about operations.

Another important parameter is `cycles to wait`.

This is amount of cycles a process waits before it executes the operation.

For example, if a curssor (process) gets on the operation `fork`, it must stay idle for 800 cycles before it actually can execute the operation.

And operation `ld` stops the process for only five cycles.

This parameter is used to create the game mechanics, in which the most effective and useful functions have the highest cost.

Code	Name	Cycles to wait
1	`live`	10
2	`ld`	5
3	`st`	5
4	`add`	10
5	`sub`	10
6	`and`	6

Code	Name	Cycles to wait
7	`or`	6
8	`xor`	6
9	`zjmp`	20
10	`ldi`	25
11	`sti`	25
12	`fork`	800
13	`lld`	10
14	`lldi`	50
15	`lfork`	1000
16	`aff`	2

THE COMPLETE OPERATIONS TABLE

Lexical_analysis.md

Before the actual translation of the assembly code into the byte-code, program will perform a so called **lexical analysis** of the source code.

> In computer science, lexical analysis, lexing or tokenization is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens (strings with an assigned and thus identified meaning).

Source: Lexical analysis — Wikipedia

the campion's code will be parsed into components, each of them being assigned to a certain type.

For example, following string:

```
loop: sti r1, %:live, %1
```

... will be decomposed into tokens:

Nr	Content	Type
1	`loop:`	`LABEL`
2	`sti`	`INSTRUCTION`
3	`r1`	`REGISTER`
4	`,`	`SEPARATOR`
5	`%:live`	`DIRECT_LABEL`
6	`,`	`SEPARATOR`
7	`%1`	`DIRECT`

It is possible to study in detail how the provided translator program identifies each type of token using the error messages it displays.

How it works

Brute-forcing original `asm` with wrong inputs and reading it's error messages we can find out the following:

Commands order

Places of commands for name and comment definitions can be switched

```
.comment    "This city needs me"
.name       "Batman"
```

`STRING`

`STRING` token contains everything between opening and closing double quotation marks `"`, even if there are new line characters `\n` and comment characters (`#` or `;`) in between.

The following example is absolutely correct:

```
.name "one
#two
three"
```

Extra tabs and spaces

If translator can distinguish two components without whitespaces in between them, spaces can be omitted (valid examples):

```
live%42
```

```
loop:sti r1, %:live, %1
```

```
.name"Batman"
.comment"This city needs me"
```

But obviously there are cases where whitespaces are mandatory.

±0

Arguments can contain `-`, but a `+` will lead to an error message.

There also can be present some leading zeroes:

```
live %0000042
```

Registries

The provided `asm` recognises as a registry a string containing character `r` with one or two trailing digits.

The following examples are valid:

```
r2
```

```
r01
```

```
r99
```

Last example deserves some attention.

Lack of error messages on registry 99 means `asm` doesn't know about virtual machines' configuration and is not connected to the constant `REG_NUMBER`.

This is absolutely logic. Translator `asm` and the virtual machine are different programs which are not aware of each other's configuration (or even existence).

Invalid byte-code may become valid (and vice versa) if we change value of the constant `REG_NUMBER`.

> If value of `REG_NUMBER` is `16`, argument `r42` is incorrect. If value of constant is `49` argument becomes valid.

Two important cases (`r0` and `r00`)

This two arguments are translated without error messages by the original `asm`. But this contradicts to the subject of the project, which clearly states:

> Registry: (`r1 <-> rx` with `x = REG_NUMBER`)

So if we can explain behaviour of translator with the top boundary (it's not connected to `REG_NUMBER`), arguments `r0` and `r00` are always invalid.

Executable code size

The size of executable code is not restricted by the translator. It will translate both files with one instruction and with one million instructions.

The size is limited in the virtual machine. The max size is defined in the constant `CHAMP_MAX_SIZE` in the file `op.h`:

```
# define MEM_SIZE           (4 * 1024)
// ...
# define CHAMP_MAX_SIZE     (MEM_SIZE / 6)
```

So max size is 682 bytes.

The min size is not limited by the virtual machine (it will run a champion without executable code).

But it is quiet difficult to create such a file. If `.s` file contains name and comment of champion and nothing more, it is invalid. But if we add at least one label to the end of file, it will be successfully translated.

```
.comment    "This city needs me"
.name       "Batman"

loop:
# EOF
```

Assembly_to_byte-code.md

File structure

To understand the rules of translation from source code into byte-code, we have to take a look at the structure of the file `.cor`.

Let's translate one of the champions with the provided `asm`:

```
.name       "Batman"
.comment    "This city needs me"

loop:
        sti r1, %:live, %1
live:
        live %0
        ld %0, r2
        zjmp %:loop
```

This is how the `.cor` content looks like:

MAGIC HEADER

CHAMPION NAME

NULL

CHAMPION EXEC CODE SIZE

CHAMPION COMMENT

```
00ea 83f3 4261 746d 616e 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0016 5468 6973
2063 6974 7920 6e65 6564 7320 6d65 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000
```

```
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
```

What is magic header why use it?

Magic header is a signature of the file and means file is of a certain type (make analogy to the extension).

If you see a file with `.cor` extension, you assume it is a corewar champion. The virtual machine will also check the magic header and will only execute the file if it does contain one.

Champion name

Next 128 bytes are champion's name. 128 is the value of constant `PROG_NAME_LENGTH` from `op.h` file.

If the actual name is shorter than 128 bytes. The remaining space will be filled in with trailing zeroes.

Each character is written to the file as it's ASCII value:

Character	B	a	t	m	a	n
ASCII-code	0x42	0x61	0x74	0x6d	0x61	0x6e

NULL

Next four bytes are NULL-bytes. Their goal is to be present at this place and to be NULL. If they are not, file is invalid.

Champion exec code size

Next four bytes represent the size of champion's executable code (**only executable part of champion**).

The virtual machine checks whether executable code size doesn't exceed `CHAMP_MAX_SIZE` value from `op.h` file (`682` bytes).

Champion comment

Next 2048 bytes represent champions comment. It is same as champion name excepting max size `COMMENT_LENGTH` .

NULL

Next four bytes are NULL-bytes.

Champion exec code

The last part of the file is executable code.

No trailing zeroes in this part.

Operations encoding

We will need two tables to understand how encoding works.

Operations table

Code	Name	Argument #1	Argument #2	Argument #3	Codage octet	Size `T_DIR`
0x01	`live`	`T_DIR`	—	—	no	4
0x02	`ld`	`T_DIR` / `T_IND`	`T_REG`	—	present	4
0x03	`st`	`T_REG`	`T_REG` / `T_IND`	—	present	4
0x04	`add`	`T_REG`	`T_REG`	`T_REG`	present	4
0x05	`sub`	`T_REG`	`T_REG`	`T_REG`	present	4

Code	Name	Argument #1	Argument #2	Argument #3	Codage octet	Size `T_DIR`
0x06	`and`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`	present	4
0x07	`or`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`	present	4
0x08	`xor`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`	present	4
0x09	`zjmp`	`T_DIR`	—	—	no	2
0x0a	`ldi`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	`T_REG`	present	2
0x0b	`sti`	`T_REG`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	present	2
0x0c	`fork`	`T_DIR`	—	—	no	2
0x0d	`lld`	`T_DIR` / `T_IND`	`T_REG`	—	present	4
0x0e	`lldi`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	`T_REG`	present	2
0x0f	`lfork`	`T_DIR`	—	—	no	2
0x10	`aff`	`T_REG`	—	—	present	4

> **What is «Size `T_DIR` » stands for?**
>
> In short, to read/writes correct amount of bytes from/to vm memory.
>
> In more detail it will be covered in corresponding chapter.

The complete operations table

Arguments table

The second table contains codes of types of arguments, and size of arguments.

Type	Assembly	Code	Size
`T_REG`	`r`	`01`	1 byte
`T_DIR`	`%`	`10`	`T_DIR` size
`T_IND`	—	`11`	2 bytes

The complete arguments table

> **Registries and their sizes**
>
> There are two characteristics of a registry
>
> Name of registry (`r1` , `r2` ...) has size of 1 byte and is laced in the byte-code. But the registry itself is 4 byte big, as defined in the `REG_SIZE` constant, and is a variable of cursor.

> **`T_DIR` arguments size**
>
> As we can see in the operations table, size of arguments of type `T_DIR` is not fixed and depends on operation. but file `op.h` contains a constant `T_DIR` defined with value — 4:
>
> ```
> # define IND_SIZE     2
> # define REG_SIZE     4
> # define DIR_SIZE     REG_SIZE
> ```
>
> What is the logic?
>
> Operations with size 2 for `T_DIR` argument use this argument as a relative address (as an argument of type `T_IND`), and `T_IND` size is always 2 bytes.

Encoding algorithm

Each operation in byte-code has the following structure:

1. Operation code — 1 byte
2. Encoding byte for saving arguments types (if needed) — 1 byte
3. Arguments (see sizes in the table)

> **Encoding byte for arguments types**
>
> We can check whether it is needed for current operation from the operations table. If operation has one single argument and it's type is `T_DIR`, then encoding byte is not used. For all other operations encoding byte must be present.

Let's encode some operations:

```
loop:
        sti r1, %:live, %1
live:
        live %0
        ld %0, r2
        zjmp %:loop
```

Operation #1

```
loop:
        sti r1, %:live, %1
```

Define size for all parts of operation.

Operation code	Encoding byte	Argument #1	Argument #2	Argument #3
1 byte	1 byte	1 byte	2 bytes	2 bytes

Find values for all parts of operation.

Operation code

The code for each operation can be found in the operations table (1-16). For `sti` it is `11`.

Encoding byte for the arguments types

Lets write encoding byte in **binary form**. Left most pair of bits stand for type of argument #1, next pair - for type of argument #2, third pair - for type of argument #3. Last pair is always `00`. Codes for different types can be found in **Arguments table**.

Argument #1	Argument #2	Argument #3	—	The whole byte	Decimal	Hex
`T_REG`	`T_DIR`	`T_DIR`	—	—		
`01`	`10`	`10`	`00`	`01101000`		

Argument-registry `T_REG`

Convert the number of registry `r1` into `0x01`.

Argument-label `T_DIR`

Label is converted into a number, which represents distance in bytes from the current position.

Label `live` points to the next operation, we know that current operation is 7 bytes, so the value to write is 7.

This value must be written on 2 bytes — `0x0007`.

Argument-number `T_DIR`

In this case we just take value and write as is on 2 bytes.

Final output for this operation

System	Operation code	Encoding byte	Argument #1	Argument #2	Argument #3	Final output
`Decimal`	`11`	`104`	`1`	`7`	`1`	
`Hex`	`0x0B`	`0x68`	`0x0001`	`0x0007`	`0x01`	**`0B 68 01 0007 0001`**

Operation #2

Repeat for the next operation:

```
live:
        live %0
```

The only major difference is that this operation doesn't have the encoding byte for argument types:

Operation code	Argument #1
1 byte	4 bytes

Final output in hex — **`01 0000 0000`** .

Instruction #3

```
        ld %0, r2
```

Operation code	Encoding byte	Argument #1	Argument #2
1 byte	1 byte	4 bytes	1 byte

Final output in hex — **`02 90 00 00 00 00 02`** .

Instruction #4

```
        zjmp %:loop
```

Operation code for `zjmp` is `0x09` .

Encoding byte is not used here

Operation code	Argument #1
1 byte	2 bytes

Final output in hex — **`09 ff ed`**

Result

Executable code of the champion will look like:

```
0b68 0100 0700 0101 0000 0000 0290 0000
0000 0209 ffed
```

Disassembling.md

Disassembler (decompiler) is opposite to assembler (compiler). It translates the byte-code into human readable assembly language.

Important, that the restored `.s` file will not be 100% identical to original one.

During the compilation all information about comments and label names is lost. Labels are replaced with the numeric equivalents.

That is, if initial file looks like:

```
.name       "Batman"
.comment    "This city needs me"

loop:
        sti r1, %:live, %1
live:
        live %0
        ld %0, r2
        zjmp %:loop
```

... after 'compilation-decompilation cycle' will look loke:

```
.name "Batman"
.comment "This city needs me"

sti r1, %7, %1
live %0
ld %0, r2
zjmp %-19
```

Also, file `.cor` compiled from the restored `.s` file must be identical to the `.cor` file compiled from original one.

Virtual_machine.md

After byte-code files are generated, it is time to run the virtual machine.

Launch

To start `corewar` we have to pass `.cor` files as arguments:

```
$ ./corewar batman.cor ant-man.cor iron_man.cor
```

By the way, it is possible to run the vm passing the same file as parameter for multiple times:

```
$ ./corewar batman.cor batman.cor batman.cor
```

Vm doesn't care about players' names. It assigns unique IDs and handles players as `Player 1`, `Player 2` and `Player 3`.

There is a constant `MAX_PLAYERS` initialised with value 4. It limits amount of players that can be loaded in a single game.

Flag `-n`

The order of the players, or rather the order of assigning identification numbers, can be changed using the flag `-n`.

Original `corewar` doesn't support this flag, but according to the text of the subject, it must be present in our program.

This flag assigns an id number to the specified champion. And players who did not receive such a number using the flag, first of the unused id-s will be assigned:

> `-n number` sets the number of the next player. If non-existent, the player will have the next available number in the order of the parameters.

For example ...:

```
$ ./corewar batman.cor -n 1 ant-man.cor iron_man.cor
```

... Ant-Man will be `Player 1`, Batman — `Player 2`, a Iron Man — `Player 3`.

Important

the number after `-n` must be bigger than `0`, should not exceed amount of players loaded into the game, and must be unique. We can't assign same `id` to multiple players.

Validation

The virtual machine reads the files with champions' bytecode and checks for the presence of the magic header, whether the code size in header of the file matches the real one, and so on.

The max limit for player's executable code size (`682 bytes`):

```
# define MEM_SIZE           (4 * 1024)
// ...
# define CHAMP_MAX_SIZE     (MEM_SIZE / 6)
```

No min limit is defined, so a champion without executable code is considered valid.

After reading the file `vm` must have following data:

- unique ID

- champion name
- champion comment
- executable code size
- executable code

Arena initialisation

Vm's memory is a battlefield (or arena) for champions.

If all the files and parameters were validated successfully, champions' executable codes must be loaded into the arena and fight.

Arena size is defined by the `MEM_SIZE` constant, which is initialised with the value `4096`.

The start position of each player is defined as follows:

- divide memory size by amount of champions
- multiply the result by player's `id - 1`

```
4096 / 3 = 1365

player1 position **1365 * (1 - 1) = 0**

player2 position **1365 * (2 - 1) = 1365**

player3 position **1365 * (3 - 1) = 2730**
```

```
0b 68 01 00 07 00 01 01 00 00 00 00 02 90 00 00 00 00 02 09 ff ed 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

So for memory size `4096`, first cell index is `0` and last cell index is `4095`.

To avoid addressing out of memory boundaries, we have to use modulo `4096` every time we deal with addresses. **And don't forget the negative cases**.

Game parameters

Before game starts, the value of the following variables must be set:

- player last reported alive

This is the variable showing the winner. It is initialised with the highest player id, and is updated every time operation `live` is performed.

- cycles counter
- counter for operation `live`, to check how many times this operation was performed during the last `cycles_to_die` cycles.
- `cycles_to_die` — length of current check period in cycles.

This variable is initialised with the value of constant `CYCLES_TO_DIE` (`1536`).

- amount of checks performed

What is a check and how it is performed we will discuss later.

Cursors (processes) initialisation

After champions are loaded into the memory, at their beginnings are placed cursors.

cursors contain following information:

- `id` - unique.
- `carry` - flag which can be changed by certain operations and which affects `zjmp operation`, initialised with value `false`.
- `opcode` - operation code, before the battle starts it is not initialised.
- `last_live` - number of cycle in which current cursor performed operation `live` last time.
- `wait_cycles` - amount of cycles to wait before operation execution.
- `position` - address in memory
- `jump` - amount of bytes cursor must jump to get to the next operation
- `registries [REG_NUMBER]` - registries of current cursor

> All registries excepting r1 are initialised with `0`. r1 is initialised with the negated player id (`r1 = -player_id`).

Important

Cursor doesn't work for a certain player. It just executes the code it reads from the memory.

During the battle after operations `fork` or `lfork`, the new cursor will get all it's values from the parent cursor and not from the player.

Cursor list

All cursors are stored in a list. Cursors will execute operations in the order they are placed in this list.

Cursors are added to the beginning of the list. This means cursor of the last player (with the biggest id) will be executed first.

```
Cursor 3 (Player 3)
        |
        V
Cursor 2 (Player 2)
        |
```

```
          V
  Cursor 1 (Player 1)
```

That's it for the game initialisation.

Player introduction

Before game start players must be introduced:

```
  Introducing contestants...
  * Player 1, weighing 22 bytes, "Batman" ("This city needs me") !
  * Player 2, weighing 12 bytes, "Ant-Man" ("So small") !
  * Player 3, weighing 28 bytes, "Iron Man" ("Jarvis, check all systems!") !
```

The battle

Battle rules

One of the most important game variables is cycle counter.

List of cursors is checked every cycle. Vm acts accordingly to the state of cursor - when needed, it assigns a new `opcode`, sets or decreases `wait_cycles`, or performs the operation, on which cursor is placed.

The battle continues while at least one cursor is alive

That means a cursor can die. This happens during the **check**.

The check

The check is performed once in `cycles_to_die` cycles if `cycles_to_die > 0`. After it's value becomes less than 1, the check is performed each cycle.

During the check dead cursors are removed from the list.

How do you know that a cursor is dead?

Cursor is considered dead, if it performed operation `live` more than `cycles_to_die` or more cycles ago.

Also, if `cycles_to_die <= 0` all carriages are considered dead.

In addition to deletion of cursors, value of variable `cycles_to_die` is modified during the check.

If during last `cycles_to_die` cycles operation `live` was performed `NBR_LIVE` times or more, `cycles_to_die` is decreased with `CYCLE_DELTA`.

See `op.h` for values of constants.

If operation `live` was performed less than `NBR_LIVE` times, vm counts how many checks were performed.

If after `MAX_CHECKS` checks `cycles_to_die` was not changed, it gets decreased.

Example:

Cycles	`live` executed, times	`cycles_to_die`	Current check counter
1536	5	1536	1
3072	193	1536 -> 1486	2
4558	537	1486 -> 1436	1
5994	1277	1436 -> 1386	1

Cycles	`live` executed, times	`cycles_to_die`	Current check counter
7380	2314	1386 -> 1336	1
8716	3395	1336 -> 1286	1
...	...	...	...

> `live` counter is being reset after each check.

> Current check counter includes all checks (current check as well) since `cycles_to_die` was changed for the last time.

Another example:

Cycles	`live` executed, times	`cycles_to_die`	Current check counter
...	...	...	...
24244	41437	136 -> 86	1
24330	25843	86 -> 36	1
24366	10846	36 -> -14	1
24367	186	-14 -> -64	1

> This values can be reproduced with the champion Jinx, from `vm_champs.tar` archive, path: `champs/championships/2014/rabid-on/` .

Inside the game loop

In each cycle vm checks the whole list of cursors and performs the following (when needed):

Assigns a new `opcode`

If cursor has moved in the previous cycle, it gets a new opcode.

> During the first cycle all the cursors will obtain an opcode.

To get the opcode vm reads the byte from the memory address cursor stands on.

If read value corresponds to a valid operation code, it is saved in cursor.

`wait_cycles` must be set in this cycle.

Don't read encoding byte and operation arguments, since they can be overwritten during the `wait_cycles` .

If read value is not a valid operation code, save value and set `wait_cycles = 0` .

Decrease `wait_cycles`

If `wait_cycles > 0` decrease it by 1.

> **Important to execute all these operations in right order within a single cycle**
>
> - Assign new opcode
> - Set `wait_cycles`
> - Decrease `wait_cycles` by 1
>
> If there was an operation with `wait_cycles == 1` , it would be executed in the same cycle it was read.

Execute the operation

If `wait_cycles == 0` it's time to execute the operation, that is saved in the cursor.

If saved operation code is a valid code and operation requires encoding byte, validate it.

If encoding byte is valid and there are registries among the arguments, validate registry numbers.

If all the checks were successfully passed, execute the operation and move cursor to the next operation.

If operation code is not valid, move cursor to the next byte.

If operation code is valid, but encoding byte is invalid or one of arguments is not valid. Cursor must be moved to the next operation without execution.

To get to the next operation, cursor must skip operation byte, encoding byte (if present), and arguments saved in the encoding byte.

> That is why in the operations table there is `T_DIR` size even for the operations, which don't use this type of argument.

For example, opcode is `0x04`:

Opcode	Name	Argument #1	Argument #2	Argument #3
4	`add`	`T_REG`	`T_REG`	`T_REG`

Encoding byte is invalid and is equal `0xb6`, in binary:

Argument #1	Argument #2	Argument #3	—
`10`	`11`	`01`	`10`
`T_DIR`	`T_IND`	`T_REG`	`T_DIR`

Operation has only 3 arguments, so we use only three most left pairs of bits.

For this operation we have to skip 9 bytes: opcode (1 byte) + encoding byte (1 byte) + Argument `T_DIR` (4 bytes) + Argument `T_IND` (2 bytes) + Argument `T_REG` (1 byte)

`-dump` flag

In general, it's all you have to know about the virtual machine.

Last thing to add is `-dump` flag.

From subject:

> `-dump nbr_cycles`
>
> at the end of `nbr_cycles` of executions, dump the memory on the standard output and quit the game. The memory must be dumped in the hexadecimal format with 32 octets per line.

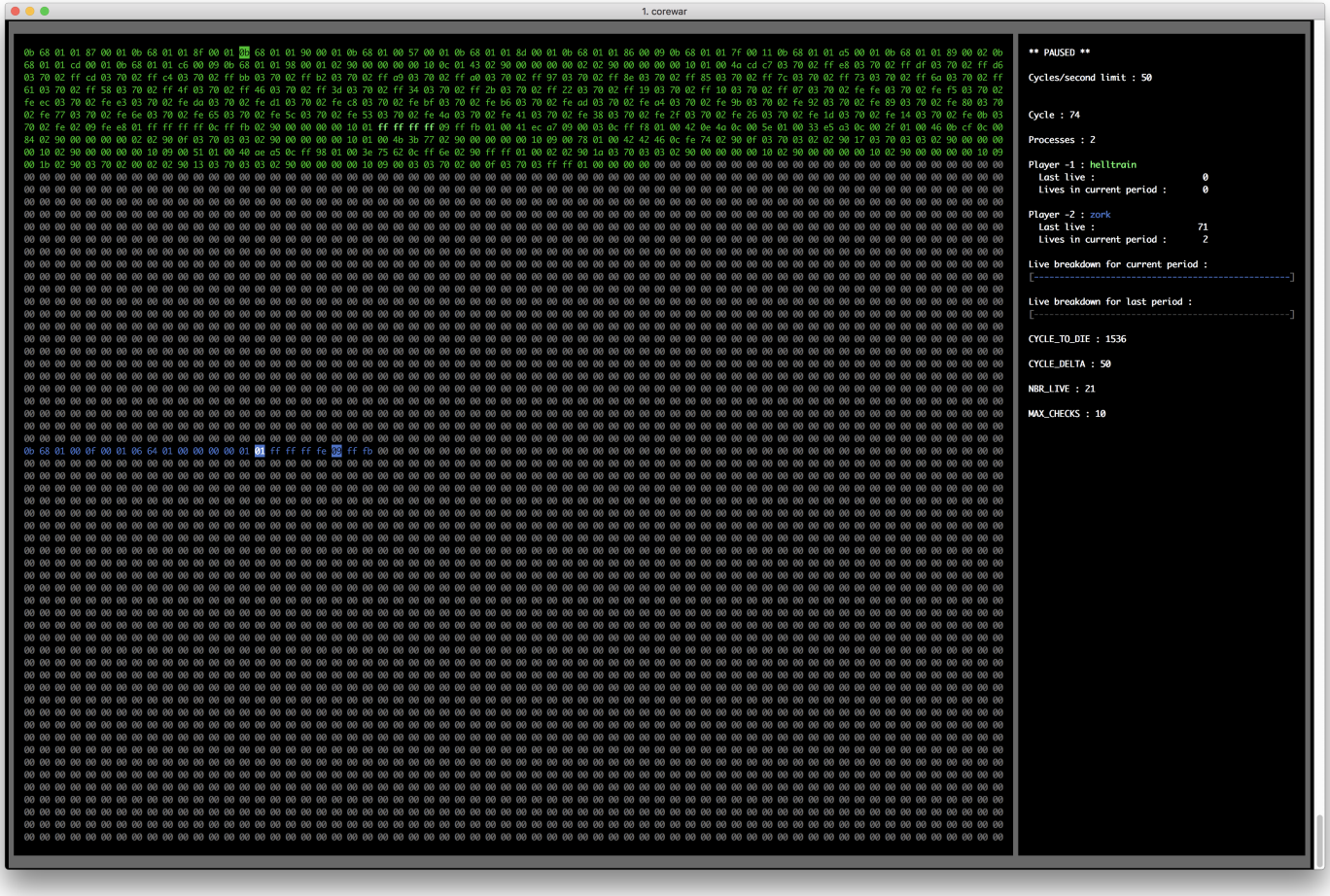
the original vm has a similar flag — `-d`.

Both flags receive number of cycle, after which vm has to print the memory dump, but `-d` prints dump 64 bytes wide bump, and `-dump` - 32 bytes wide.

Visualization.md

Visualization is a bonus part of the Corewar project, so there are a lot of different implementations.

We will look at the original visualizer as an example. It is coded with the help of library Ncurses



Each player has it's own color. The executable code of the champion will have it's color. The cursors (processes) which store player's number in their reqistry 1, will write to the board with the champion's color using operations `st` and `sti`.

Memory sectors that are not colored by champions, have default color - grey.

Cursors

To show current position of the cursor, background of cell is colored in player's color and the value is of black color.

Important that the background is of color of the cell and not of cursor. That is, a cursor created by the green player will be displayed with color blue, if it is in the blue cell.

Newly overwritten memory sectors

Also newly overwritten (with operations `st` and `sti`) memory sectors will be **bold** for the next 50 cycles.

Operation `live`

In addition to the newly overwritten memory, event of opertation `live` is highlighted.

After `live` is performed by a cursor, the cell containing the code of this operation will be highlighted as follows: cell background - with color of the cursor, and the code - with the white color and **bold**.

Highlighting operation `live` has higher priority than showing the carriage's position.