COREWAR

1. Introduction

1.1. Champion

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

In fact, we are writing code in a pseudo-assembler language. That is, in a language created specifically for this task, which is similar to a real assembler, but still it is not. But for consistency with the text of the assignment, simplicity and in order to save six letters, we will also call this language assembler.
The generated code is our champion, whose goal is to fight with other champions written by us or other people.

The code of each champion has the following structure:

1. Name
2. Comment
3. Executable code

It may look, for example, 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 a task for a completely different project called **“Corewar Championship”** . At **Corewar,** we create our champion only to demonstrate an understanding of the topic and the ability to write assembly code. And not so that he could defeat someone. Our task is to write code without errors so that the program `asm` can turn it into bytecode, which the virtual machine would then execute. The goals to win the battle or to demonstrate at least some worthy results in the battle are not worth us.

1.2. Assembler

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

From elvish this task can be interpreted as "translate commands from a language understandable to humans (assembler) 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, issues program text for printing, etc.

That is, we must create a program with a name `asm` (from the word “assembler”), which will receive as a parameter a file with code in assembly language and create a new file with byte code on its basis. The file with the code of our champion, written in assembler, must have the extension `.s`. On its basis, the program `asm` will create a new file with the extension `.cor`, where the created bytecode will be located. The name of the file itself will remain unchanged. That is, after calling the command `./asm batman.s`, a file `batman.s` should appear next to the file `batman.cor`. Of course, if during the broadcast there is no error.

1.3. Virtual Machine

After we received the bytecode file, the virtual machine’s running time comes. A virtual machine is also a program whose executive file should be called `corewar`. Her task is to allocate a specific piece of memory, place the champions code and the carriage that will execute it on this section. And then follow the progress of the battle to declare the champion champion after its completion.

1.4. Bonuses

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

1.4.1. Extended error message

If an error occurred during the generation of the bytecode, then it is desirable that in this case the program `asm` behaves like a real translator, which it actually is. That is, I got a meaningful message - in which line when working with the file `.s` an error occurred and what its type is.

1.4.2. Ability to disassemble bytecode

To make it possible to obtain source code in assembler by having a file with byte code. That is, implement the function inverse to the one for which the program is intended `asm`.

1.4.3. 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 this to focus on key points such as carriage death or a winner announcement.

2. Files provided

2.1. Archive `vm_champs.tar`

2.1.1. Translator `asm`

The project assignment says that the program `asm` translates code from the assembly language written in the `.s` file into byte code, which should be placed in a file with the extension `.cor`. But in fact, the provided program does not pay any attention to the extension of incoming files. She only searches for the last dot in the name and replaces all further contents with an extension `.cor`. The question “How can such an approach be called optimal?” Remains open. After all, this makes possible the following situations, which, although they are not errors in the usual sense, still violate the declared workflow:

```
$ ./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. Therefore, at one time, you can specify a lot of files with a variety of extensions as arguments. 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 live with it?

Since such behavior is not described in the text of the task, in your own implementation 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 also does not check incoming files for the extension `.cor`. And although `corewar` such behavior is not a source of vulnerabilities for a program , in its implementation this check should still be added to comply with a single style of behavior for all system components.

2.1.2. Champion

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 live with it?

In this situation, it is worth considering that `champs/examples` there will be no problems with the files from the folder , but you will have to verify the operability of the others personally.

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

The file contains a `op.c` 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 in this case.

We are also provided with a header file `op.h`. It 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 as it is. Making only the most necessary changes.

As for participation in the project **“Corewar Championship”** its availability is one of the conditions:

> 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 that must be performed on this file is to bring it into compliance with Norm. Since the file provided does not comply with the established formatting rules, Norminette will display a number of warnings about errors found when checking it.

If participation in the Corewar Championship competition is not planned, then `op.h` you should not worry about the fate of the file . In this case, any, even much more serious, changes can be made with him than a simple reduction to Norm.

3. Assembler

3.1. Assembler Syntax

The assembly language obeys 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. Empty lines, comments, as well as extra tabs or spaces are ignored.

3.2. Comment

There `op.h`is a constant in the header `COMMENT_CHAR`. It determines which character indicates the beginning of the comment. In the file provided, it is octotorp - `#`. That is, everything between the character `#`and the end of the line will be taken as a comment . A comment can be located anywhere in the file.

Example # 1:

```
# UNIT Factory
# is a programming school
```

Example # 2:

```
ld %0, r2    # And it is located in UNIT City
```

3.3. Alternative comment

In the provided archive `vm_champs.tar`along the way `champs/examples`you can find 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 and alternative, about which there is no information on the subject. This alternative view differs from the standard and described in subject only in the symbol of the beginning of the comment. Instead of octotorp (`#`) stands here `;`. An example of using a comment of this type:

```
sti r1, %:live, %1    ; UNIT City is placed in Kyiv, Ukraine
```

How to live with it?

This type of comment is not described in subject, but is supported by the original translator. Therefore, we most likely do not have to process it. But still add his support to our project. To do this, the `op.h` following line will be added to the header file :

```
# define ALT_COMMENT_CHAR    ';'
```

3.4. Champion name

In the file with the code of the champion his name must be defined. To do this, in assembler there is a command whose name is defined in a constant `NAME_CMD_STRING`. In the file provided, `op.h` this `.name`. That is, after the command `.name` should follow the line with the name of our champion:

```
.name    "Batman"
```

The line length must not exceed the number specified in the constant `PROG_NAME_LENGTH`. In the file provided, it is equal `128`. By the way, an empty string can also be used as a champion name:

```
.name    ""
```

But the complete absence of a line is already an error:

```
.name
```

3.5. Champion comment

Also in the file with the extension `.s` must be defined comment champion. A command to help you do this is contained in the `COMMENT_CMD_STRING` file constant `op.h`. In the file provided, this `.comment`. The length of the comment line is limited by a constant `COMMENT_LENGTH`. In the file provided, `op.h` its value is equal `2048`. At its core, a command is `.comment` very similar to `.name` and behaves similarly in cases with an empty string and in cases with its complete absence.

3.6. Other teams

In some files with the extension `.s` that were provided to us as an example, there was such a command as `.extend`. This command, like any others other than `.name` and `.comment`, is not described in subject'e and the original translator is defined as erroneous. In the same way we will handle similar teams.

3.7. Executable code

The champion executable code consists of instructions. For the assembler language, the “One line - one instruction” rule applies. And the symbol for ending an instruction in this language is a line feed. That is, instead of the symbol familiar in the language C, the symbol `;`appears here `\n`.

Based on this rule, we must remember that **even after the last instruction, a line feed should follow** . Otherwise, it `asm` will display an error message. Each instruction consists of several components:

- Labels.
- Operations and their arguments.

3.7.1. Label

A label consists of characters that have been defined in a constant `LABEL_CHARS`. In the sample file, this `abcdefghijklmnopqrstuvwxyz_0123456789`. That is, the label cannot contain characters that are not specified in `LABEL_CHARS`. And after the label itself, the symbol defined in the constant should follow `LABEL_CHAR`. In the sample file, this is a symbol `:`. Why are tags needed? A label indicates an operation that immediately follows. It is for one operation, and not for their unit.

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

loop:
        sti r1, %:live, %1    # <-- На эту операцию указывает метка loop
live:
        live %0               # <-- На эту операцию указывает метка live
        ld %0, r2             # <-- А на эту операцию никакая метка не указыва
ет
        zjmp %:loop
```

The task of labels is to simplify our life, or rather the process of writing code. To fully appreciate their role, let's imagine a world without labels. As we know, the champion code written in assembly language after the translator program will turn into a lot of bytes represented in the hexadecimal system. And it is just such a bytecode that the virtual machine will execute. Suppose we need to organize a cycle in which an operation would be performed over and over again `live`. To do this, we have an operation `zjmp` that can transfer us to the Nth number of bytes forward or backward. In this case, we need to return to the operation again after each iteration of the loop `live`. But how many bytes are it back? To find out, you need to find out how many bytes in the bytecode the operation code and its argument will occupy. As we learn later, the operation code `live`takes 1 byte, and it's only argument needs 4 bytes. It turns out we need to go back 5 bytes back:

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

Not so difficult, but still, such calculations take time. And it would be much simpler to just write “go to surgery `live`”. For this, there are labels. We simply create a label for the operation we need `live` and transfer it to the operation `zjmp`:

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

From the point of view of translating assembly code into bytecode, both examples are absolutely identical. During operation, the program `asm`will calculate how many bytes the label indicates back `loop`and replace it with a number `-5`. So, for the final result, it does not matter at all what was used. But writing code using labels is much more convenient.

Recording Forms

There are several approaches to writing a label:

```
marker:
        live %0
```

```
marker:


        live %0
```

```
marker: live %0
```

All the above examples for the translator program mean the same thing. Therefore, you can choose any of the options presented. Many marks for one operation. This form of recording is also possible:

```
marker:

label:
        live %0
```

This means that both the label `marker`and the label `label`point to the same operation.

No operation

Or a situation may arise when the label does not have an operation to which it could indicate:

```
marker:
# Конец файла
```

In this case, the label indicates the place immediately after the champion's executable code. The main thing is that at the end of the line on which it is located `\n`. Otherwise, the compiler will report an error.

3.7.2. Operations and their arguments

Assembly language has a specific set of 16 operations. Each of which takes from one to three arguments. Information about the name of the operation, its code, as well as about the arguments that it takes, is given in the file provided by the task `op.c`.

Operation code	Operation name	Argument # 1	Argument # 2	Argument # 3
one	live	T_DIR	-	-
2	ld	T_DIR / T_IND	T_REG	-
3	st	T_REG	T_REG / T_IND	-
four	add	T_REG	T_REG	T_REG
five	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
eight	xor	T_REG / T_DIR / T_IND	T_REG / T_DIR / T_IND	T_REG
9	zjmp	T_DIR	-	-
ten	ldi	T_REG / T_DIR / T_IND	T_REG / T_DIR	T_REG
eleven	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	-	-
sixteen	aff	T_REG	-	-

3.8. Operations and their arguments

3.8.1. Arguments

3.8.1.1. Register - Registry - `T_REG`

Register is such a variable where we can store any data. The size of this variable in octets is indicated in a constant `REG_SIZE`, which is `op.h` initialized in the example file with a value `4`. An octet in computer science - eight binary digits. In Russian, an octet is usually called a byte. The number of registers is limited by the number specified in the constant `REG_NUMBER`. In the example - `16`. That is available to us it registers `r1`, `r2`, `r3`... `r16`.

Register Values: During startup of the virtual machine, all registers, except `r1`, will be initialized to zeros. The `r1`number of the champion player will be recorded. Only with a minus sign. This number is unique within the game and needs operations `live` to inform that a particular player is alive. That is, the carriage, which will be placed at the beginning of the player's code under the number `2`, will receive a value `r1`equal to `-2`. If the operation `live` is performed with an argument `-2`, the virtual machine will consider that this player is alive:

```
live %-2
```

3.8.1.2. Direct - Direct - `T_DIR`

The direct argument consists of two parts: the character that is specified in the constant `DIRECT_CHAR`(`%`) + the number or label that represents the **direct value**. In the case of a label, then the symbol from the variable `LABEL_CHAR`(`:`) must also be indicated before its name :

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

What is direct and indirect meaning?

To understand the difference between direct and indirect value, it is worth considering one very simple example. Suppose we have a number `5`. In its direct meaning, it represents itself. That is, a number `5`is a number `5`. But in an indirect sense, this is no longer a number, but a relative address that points 5 bytes in advance.

Direct and indirect label

If everything is clear with numbers in the direct and indirect meaning, then what about the labels? What is the difference? Everything is pretty simple. As we know, the virtual machine will perform these operations. And for her it is important what argument was received. Direct or indirect? But the tags simply won't reach the virtual machine. At the stage of translation into bytecode, they will all be replaced by their numerical equivalents. Therefore, labels are the same numbers. Only recorded in another form. The process of replacing labels with numbers is described in the chapter "Why do we need labels?".

3.8.1.3. Indirect - Indirect - `T_IND`

An argument of this type can be either a number or a label, which represent an **indirect value**. If the argument `T_IND` is a number, then no additional characters are needed:

```
ld    5, r7
```

If the label is such an argument, then the symbol from the variable `LABEL_CHAR`(`:`) must be indicated before its name :

```
ld    :label, r7
```

3.8.2. Delimiter character

In order to separate one argument from another within one operation, the assembler uses a special delimiter character. It is determined by the constant of the preprocessor `SEPARATOR_CHAR`and in the example file `op.h`it is a symbol `,`:

```
ld    21, r7
```

3.8.3. Operations

3.8.3.1. Operation `live`
3.8.3.2. Operation `ld`
3.8.3.3. Operation `st`
3.8.3.4. Operation `add`
3.8.3.5. Operation `sub`
3.8.3.6. Operation `and`
3.8.3.7. Operation `or`
3.8.3.8. Operation `xor`
3.8.3.9. Operation `zjmp`
3.8.3.10. Operation `ldi`
3.8.3.11. Operation `sti`
3.8.3.12. Operation `fork`
3.8.3.13. Operation `lld`
3.8.3.14. Operation `lldi`
3.8.3.15. Operation `lfork`
3.8.3.16. Operation `aff`

3.8.4. Updated Operation Table

Code	Name	Argument # 1	Argument # 2	Argument # 3	Changes `carry`	Description
one	`live`	`T_DIR`	-	-	Not	alive
2	`ld`	`T_DIR` / `T_IND`	`T_REG`	-	Yes	load
3	`st`	`T_REG`	`T_REG` / `T_IND`	-	Not	store
four	`add`	`T_REG`	`T_REG`	`T_REG`	Yes	addition
five	`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 (`|`)
eight	`xor`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR` / `T_IND`	`T_REG`	Yes	bitwise XOR (`^`)
9	`zjmp`	`T_DIR`	-	-	Not	jump if non-zero
ten	`ldi`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	`T_REG`	Not	load index
eleven	`sti`	`T_REG`	`T_REG` / `T_DIR` / `T_IND`	`T_REG` / `T_DIR`	Not	store index
12	`fork`	`T_DIR`	-	-	Not	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`	-	-	Not	long fork
sixteen	`aff`	`T_REG`	-	-	Not	aff

3.8.5. Cycles before execution

But that's not all you need to know about operations. There is another important parameter - loops before execution. This is the number of cycles that the carriage must wait before starting an operation. An example, having appeared on the operation for the `fork` next 800 cycles, she will hold

on hold. And for the operation, the `ld` wait is only 5 cycles. This parameter was introduced to create game mechanics in which the most effective and useful functions have the highest cost.

Code	Name	Cycles Before Execution
one	`live`	ten
2	`ld`	five
3	`st`	five
four	`add`	ten
five	`sub`	ten
6	`and`	6
7	`or`	6
eight	`xor`	6
9	`zjmp`	20
ten	`ldi`	25
eleven	`sti`	25
12	`fork`	800
13	`lld`	ten
14	`lldi`	50
15	`lfork`	1000
sixteen	`aff`	2

4. Lexical analysis

Before starting the translation of code written in assembly language into bytecode, the translator program performs the so-called **lexical analysis of the** source code. In computer science, lexical analysis (“tokenization”, from the English tokenizing) is the process of analytically parsing an input sequence of characters into recognized groups - tokens, in order to obtain identified sequences called “tokens” (like grouping letters in words). This process consists in parsing the champion code into separate components, each of which will be assigned to one of the types. For example, the following line after the lexical analysis turned into ...:

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

... token list:

roomContent	Type of
one	loop: LABEL
2	sti INSTRUCTION
3	r1 REGISTER
four	, SEPARATOR
five	%: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.

4.1. Features of work

It is thanks to these reports and the "method of scientific enumeration" that you can find out the following:

4.1.1. Order of commands

Commands for setting the name and comment can be interchanged:

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

4.1.2. Line

A type token `STRING` includes everything from opening to closing quotes. And neither line breaks nor comment characters stop him on this path. Therefore, the example below is absolutely correct:

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

4.1.3. Optional spaces and tabs

In cases where the translator can unambiguously separate the components from each other, white spaces between them can be omitted:

```
live%42
```

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

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

But there are cases when you cannot do without at least one space or tab.

4.1.4. ± 0

Operation arguments may contain minus signs, but using the plus sign will cause an error. It cannot be in the number. But leading zeros may be present:

```
live %0000042
```

4.1.5. Registers

For the provided program, the `asm` register is a string consisting of a character `r` and one or two digits. Therefore, all of the following examples will be successfully translated into bytecode:

```
r2
```

```
r01
```

```
r99
```

Particular attention should be paid to the last of them.

The fact that its translation does not cause any errors indicates that `asm` it knows nothing about the configuration of the virtual machine and the value of the constant `REG_NUMBER`.

This behavior of the program is absolutely logical. After all, the translator and the virtual machine are separate entities that are not aware of each other's configuration (or even existence).

And if the constant value changes `REG_NUMBER`, the same bytecode can become valid for the virtual machine from invalid, and vice versa.

> If the value is `REG_NUMBER` equal, the `16` operation with the register argument `r42` will be incorrect. And with the value, `49` it will be executed without problems.

Two special cases

But two special cases of the program work `asm` raise questions - `r0` and `r00`.

The original translator will also handle these two examples without any problems. Although this contradicts the logic of the assignment, which says:

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

And if the program's ignorance of the value of the upper bound (`REG_NUMBER`) is logical and easily explainable. Indeed, when changing the configuration of a virtual machine, the same section of bytecode can become invalid from invalid. The translation results `r0`, and `r00` will not be correct at all times.

Therefore, when implementing your own, `asm` this problem is worth fixing.

4.1.6. Executable Code Size

The size of the executable code is in no way limited to the translator program. That is, it `asm` will work equally well with the source code consisting of one instruction, and with the code, where these instructions are hundreds of thousands.

With a virtual machine, everything is a little different. For her, a restriction is set on the maximum amount of executable code in bytes using a constant `CHAMP_MAX_SIZE`.

In the returned file, `op.h` this constant is initialized with the following lines:

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

That is, in this case, the size of the champion's executable code should not exceed 682 bytes.

With a minimal border, everything is a little more interesting.

The virtual machine will easily accept a `.cor` file in which the size of the executable code will be zero. But to produce such a file using the provided translator is not so simple.

If, apart from setting the name and comment of the champion, nothing else is written to the `.s` -file, then it will not be possible to convert it to bytecode.

But if you add only one label to the end, you can get the desired file as a result without executable code:

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

loop:
# Конец файла
```

Perhaps the creators of the original `asm` wanted to ban the broadcast of the champion without an executable code. So that at least one operation in the file is still present.

But in this case, they did not take into account some points and you can still create such a file.

How to live with it?

In your work, you can, as well as improve this protection, to guarantee the presence of at least one operation. So remove it completely, which looks like a more logical solution. After all, the original virtual machine works without problems with `.cor` files without executable code.

5. From assembler to bytecode

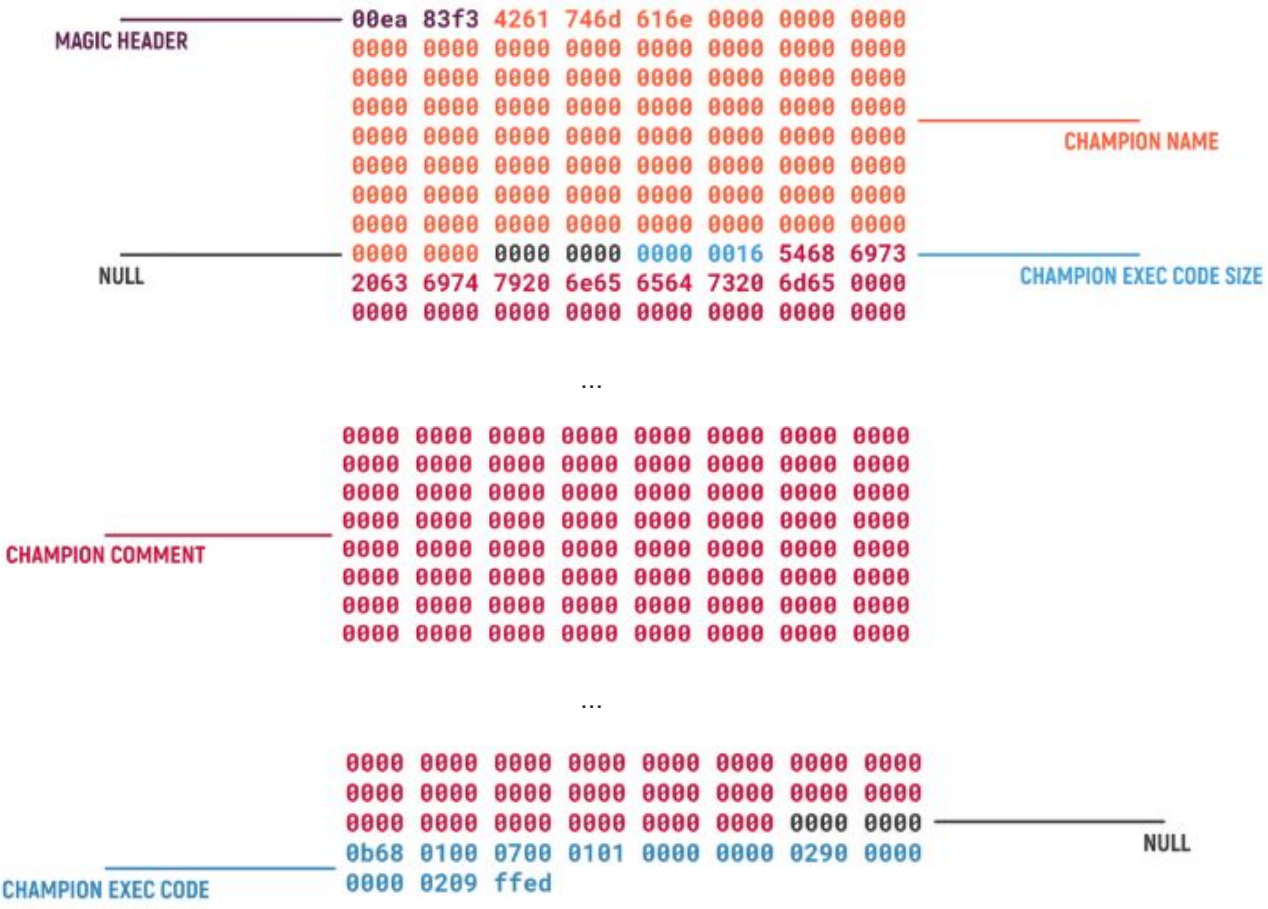
5.1. File structure

To understand what rules the process of translating assembly code into bytecode follows, you need to consider the file structure with the extension `.cor`. To do this, we broadcast this champion using the program provided by the task `asm`:

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

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

The resulting file will have the following structure:



5.1.1. Magic header

The first 4 bytes in the file are the “magic number”. It is defined using a constant `COREWAR_EXEC_MAGIC` whose value in the sample file `op.h` is equal `0xea83f3`.

What is magic header and why is it needed? Magic header is such a special number, the task of which is to inform that the given file is binary. If there is no such “message” in the file, it can be interpreted as text.

5.1.2. Champion name

The next 128 bytes in the file takes the champion name. Why 128? This is a constant value `PROG_NAME_LENGTH` that defines the maximum length of a string with a name. The name of our champion is much shorter. But in bytecode, it still takes up all 128 bytes. Because according to the rules of translation, if the length of a string with a name is less than the established limit, then the missing characters are compensated by zero bytes. In general, each character of the name is converted into an ASCII code of 1 byte size, written in hexadecimal notation:

Symbol	`B`	`a`	`t`	`m`	`a`	`n`
ASCII code	`0x42`	`0x61`	`0x74`	`0x6d`	`0x61`	`0x6e`

And instead of the missing characters, we write zero bytes.

5.1.3. Null

The next 4 bytes in the file structure are reserved for a certain control point - four zero octets. They do not carry any information load. Their task is simply to be in the right place.

5.1.4. Champion exec code size

These 4 bytes contain quite important information - the size of the champion's executable code in bytes. As we recall, the virtual machine must make sure that the size of the source code does not exceed the limit specified in the constant `CHAMP_MAX_SIZE`. In the file provided, `op.h`it is equal `682`.

5.1.5. Champion comment

The next 2048 bytes are occupied by the champion comment. And at its core, this part is completely analogous to the Champion name part. True except that now the limit on the maximum length is set by a constant `COMMENT_LENGTH`.

5.1.6. Null

And again 4 zero octets.

5.1.7. Champion exec code

The last part of the file is the executable code of the champion. Unlike a name or comment, it is not padded with null bytes.

5.2. Operations Coding

5.2.1. Operation table

First of all, we need an updated table of operations with the columns "Code of argument types" and "Size `T_DIR`".

Code	Name	Argument # 1	Argument # 2	Argument # 3	Argument Type Code	The size `T_DIR`
0x01	live	T_DIR	-	-	Not	four
0x02	ld	T_DIR / T_IND	T_REG	-	there is	four
0x03	st	T_REG	T_REG / T_IND	-	there is	four
0x04	add	T_REG	T_REG	T_REG	there is	four
0x05	sub	T_REG	T_REG	T_REG	there is	four
0x06	and	T_REG / T_DIR / T_IND	T_REG / T_DIR / T_IND	T_REG	there is	four
0x07	or	T_REG / T_DIR / T_IND	T_REG / T_DIR / T_IND	T_REG	there is	four
0x08	xor	T_REG / T_DIR / T_IND	T_REG / T_DIR / T_IND	T_REG	there is	four
0x09	zjmp	T_DIR	-	-	Not	2
0x0a	ldi	T_REG / T_DIR / T_IND	T_REG / T_DIR	T_REG	there is	2
0x0b	sti	T_REG	T_REG / T_DIR / T_IND	T_REG / T_DIR	there is	2
0x0c	fork	T_DIR	-	-	Not	2
0x0d	lld	T_DIR / T_IND	T_REG	-	there is	four
0x0e	lldi	T_REG / T_DIR / T_IND	T_REG / T_DIR	T_REG	there is	2
0x0f	lfork	T_DIR	-	-	Not	2
0x10	aff	T_REG	-	-	there is	four

Why do we need "Size `T_DIR`" for operations that do not accept arguments of this type?

At this stage, this is really useless information. But they will be needed while the virtual machine is running. Which will be considered in the corresponding section.

5.2.2. Argument table

The second table we need contains information about the codes of argument types and their sizes.

Type of	Sign	Code	The size
T_REG	r	01	1 byte
T_DIR	%	10	The size `T_DIR`
T_IND	-	11	2 bytes

About registers and its sizes: It is important to distinguish between two size characteristics regarding registers. The name of the register (`r1`, `r2`...) in the bytecode is 1 byte. But the register itself contains 4 bytes, as indicated in the constant `REG_SIZE`.

About the size of type arguments `T_DIR`: As you can see in the table of operations, the size of type arguments is `T_DIR`not fixed and depends on the operation. But `op.h` there is a preprocessor constant in the file , which says that the size `T_DIR`is equal to the size of the register - 4 bytes:

```
# define IND_SIZE      2
# define REG_SIZE      4
# define DIR_SIZE      REG_SIZE
```

Where is the logic here?

There is logic here, but rather ghostly.

As we have already said, to write a type number `T_DIR` to a register and to unload this number into memory, the sizes of the registers and type arguments `T_DIR` must match. With this statement, all is well. It is correct. If you look at the operations that load the value into the register, for them the size `T_DIR` is equal `4`. Also the size `T_DIR` is equal `4` to the operation `live`. As for operations for which the size `T_DIR` is equal `2`, in these cases an argument of this type takes part only in the formation of the address. And for such a task, 4 bytes are redundant. After all, the amount of memory is only 4096 bytes, as indicated in the constant `MEM_SIZE`. And any address number that exceeds this limit will be truncated modulo `MEM_SIZE` if it has not already been done with `IDX_MOD`. In this situation, the type argument `T_DIR`plays the role of a relative address. That is, an argument of type `T_IND`. And so its size is `IND_SIZE`(2 bytes).

5.2.3. Coding algorithm

Each operation presented in bytecode has the following structure:

1. Operation code - 1 byte
2. Argument type code (Not required for all operations) - 1 byte
3. Arguments

Argument Type Code

As indicated above, the second component may be absent in the structure of the encoded operation — the argument type code.

Its availability depends on the specific operation. If it takes only one argument and its type is uniquely defined as `T_DIR`, then code with information about the types of arguments is not needed. For all other operations, this component is required.

You can check whether this code is needed for a specific operation in the "Type of Argument Code" column of the operation table.

Let's look at how we code executable code instructions:

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

Instruction # 1

The first instruction to broadcast is:

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

First, let's set the dimensions of each component.

Operation code	Argument Type Code	Argument # 1	Argument # 2	Argument # 3
1 byte	1 byte	1 byte	2 bytes	2 bytes

Now let's look at how we recognize the bytecode of each part of the instruction.

Operation code

The code for each operation is indicated in the operation table. For `sti`it is equal `0x0b`.

Argument Type Code

In order to generate this code, you need to represent 1 byte **in the binary system** . The first two bits on the left will be occupied by code like argument # 1. The next two will go to code like the second argument. And so on. The last fourth pair will always be equal `00`.

Codes for each type are listed in the argument table.

Argument # 1	Argument # 2	Argument # 3	-	Final code
`T_REG`	`T_DIR`	`T_DIR`	-	-
`01`	`10`	`10`	`00`	`0x68`

Type argument `T_REG`

In this case, the register number is translated into the hexadecimal code. To register `r1`it `0x01`.

Type Label Argument `T_DIR`

As we already know, the label should turn into a number that contains the relative address in bytes.

Since the label `live` indicates the next instruction, and the size in bytes of the current instruction is already known to us, you can easily calculate the required distance - 7 bytes.

The resulting address number should be placed on 2 bytes - `0x0007`.

Type argument `T_DIR`

In this case, it's still easier. You just need to write down the number obtained in the decimal system, in the form of a hexadecimal bytecode - `0x0001`.

The final bytecode of the instruction is `0b 68 01 0007 0001`.

Instruction # 2

For the following instructions, everything is similar:

```
live:
        live %0
```

The only significant change is that the argument type code is not needed for this operation:

Operation code	Argument # 1
1 byte	4 bytes

The resulting bytecode for the second instruction is `01 00000000`.

Instruction # 3

The third instruction:

```
        ld %0, r2
```

Operation code	Argument Type Code	Argument # 1	Argument # 2
1 byte	1 byte	4 bytes	1 byte

There are also no surprises here `02 90 00000000 02`.

Instruction # 4

```
        zjmp %:loop
```

The operation code `zjmp`is `0x09`. The argument type code is not needed for this operation.

Operation code	Argument # 1
1 byte	2 bytes

The label `loop` indicates 19 bytes back. But how do we represent a number in the hexadecimal system `-19`? To do this, write the number `19` **in the binary system** in the direct code:

```
0000 0000 0001 0011
```

From the number `19` written in the direct code, we can get the number `-19` in the additional code.

It is in the additional code that it is customary to represent negative integers in computers. In order to get an additional number code `-19`, we must perform the following steps:

1. Invert all digits. That is, change the unit to zero, and vice versa:

```
1111 1111 1110 1100
```

2. Add one unit to the number

```
1111 1111 1110 1101
```

Done. So we got the number `-19`in the additional code. Convert it from binary to hexadecimal - `0xffed`. Final bytecode for instruction # 5 - `09 ffed`.

Total

The entire executable code of the champion in question will look like this:

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

6. Disassembly

A disassembler performs the opposite task `asm`- translates the bytecode back into assembly language. It is important to understand that the “restored” file will not be 100% identical to the original. After all, the `.cor` file does not contain information about comments that are ignored during translation, as well as label names, which are replaced by numerical equivalents during processing. Therefore, the file that originally looked like this ...:

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

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

... after "recovery" from the bytecode it will look a little different:

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

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

7. Virtual machine - Launch

After receiving the Champions bytecode files, the virtual machine's operating time comes.

When starting the program `corewar` as arguments, we indicate the `.cor` files of the champions who will take part in the battle:

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

By the way, this does not have to be three different files with three different champions. We could start the virtual machine by specifying the same file in the parameters three times. Ej `batman.cor`.

The program is absolutely no difference. It distinguishes players not by their names, but by using unique numbers that they assign to each of them. For her, three Batman's `Player 1`, `Player 2`and `Player 3`.

The maximum number of champions who can simultaneously fight in memory is monitored by a constant `MAX_PLAYERS`. In the sample file, it is initialized with a value `4`. That is, no more than 4 players can be loaded into the virtual machine at a time.

7.1. Flag `-n`

The order of the players, or rather the order of the identification numbers, can be changed using the flag `-n`. In the original virtual machine, `corewar` support for such a flag is not implemented, but according to the text of the task, it should be present in our program. This flag assigns an identification number to the specified champion player. And for those players who did not receive such a number using the flag, the first of the unoccupied 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.

That is, if you run this command ...:

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

... Ant-Man will become `Player 1`, Batman - `Player 2`, and Iron Man - `Player 3`.

The only thing you need to monitor in this case is the number that is indicated after `-n`. It should be more or equal `1`, but not exceed the total number of players who take part in the battle. Also, this number must be unique within the battle, so several players cannot have the same number.

7.2. Validation

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

By the way, if the translator program did not attach importance to the maximum size of the executable code, then for the virtual machine this parameter has a value and cannot exceed 682 bytes:

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

The virtual machine does not have a minimum limit on the size of executable code, so it may be completely absent. After processing the received data about each of the players, the virtual machine should know the following:

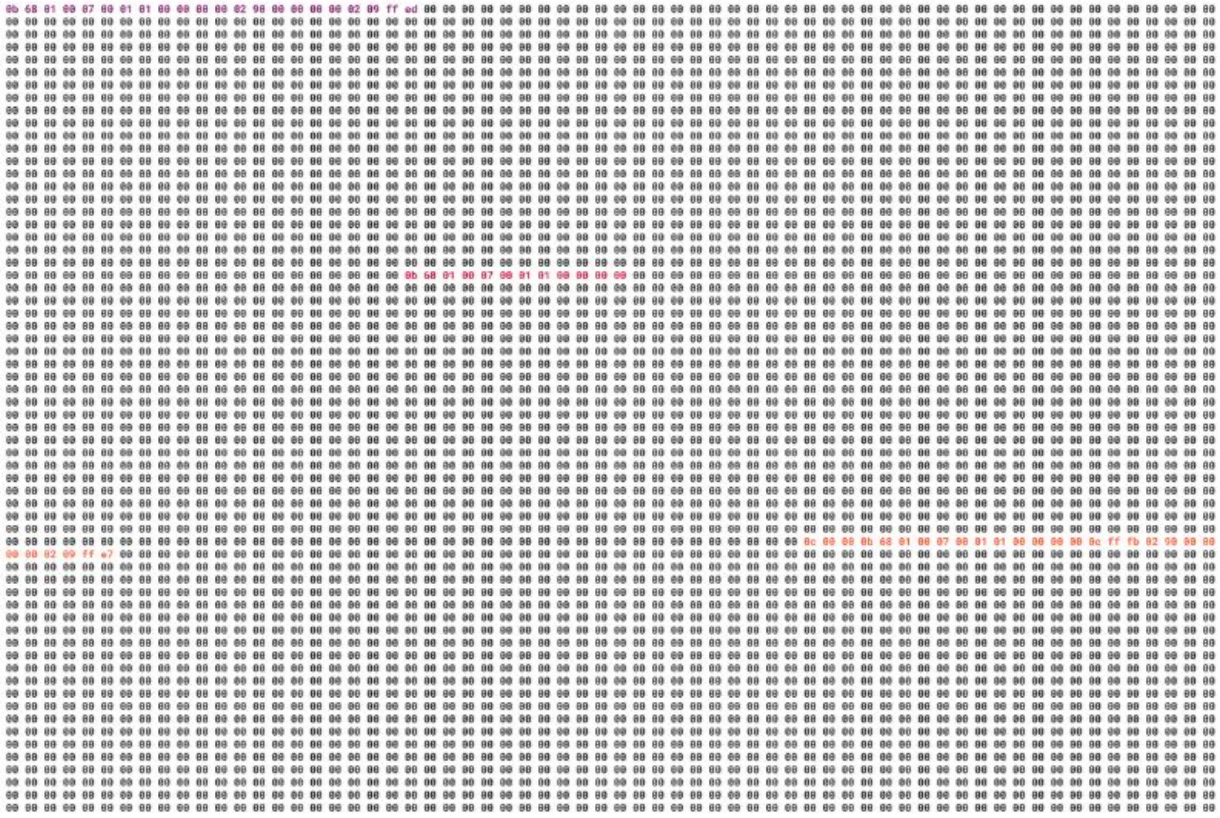
- unique identification number
- champion name
- champion comment
- size of executable code in bytes
- executable code

7.3. Arena initialization

After the program is convinced that everything is fine with the provided files of the players, it is necessary to initialize the arena where the champions' executable codes will be placed. The size of the arena in bytes is determined by a constant `MEM_SIZE`whose value in the example is equal `4096`. To understand where the executable code of a particular player will be located, it is necessary to divide the total amount of memory by the number of participants in the battle. The result will be equal to the amount of memory in bytes, at the beginning of each of which will be located the executable code of the corresponding champion:

```
4096 / 3 = 1365
```

That is, the code of the first player will be located starting from the zero memory cell and beyond. The code of the second is from the 1365th cell. And the third - from the 2730th.



7.3.1. Memory organization

The memory in the virtual machine is organized on the principle of a ring or loop. That is, the last cell follows the first. And if the total memory capacity is 4096 cells, then the number of the first cell will be equal `0`, and the last - `4095`. In the case of access to the cell with a number, more `4095`, the remainder of the division modulo will be taken as a valid number `4096`. Thus, out of memory limits.

7.3.2. Game Options

Also, before starting work, you need to set the value of the following variables:

- The last player to say he was alive: It is with this variable that the winner of the battle will be announced. This variable is initialized with a pointer to the player with the highest identification number, and is updated in the operation `live`.
- The number of cycles since the beginning of the game.
- The number of operations performed `live` over the last period, long `cycles_to_die.`
- `cycles_to_die` - the length of the period before verification: Initialized by a constant value `CYCLES_TO_DIE`, which is equal to `1536`.
- Number of checks performed: What is verification and what is its role in the operation of a virtual machine will be discussed later.

7.4. Carriage Initialization

After the champions executable codes were placed in the arena, a carriage is placed at the beginning of each of them. Inside it contains the following data:

- Unique carriage number
- `Carry`: A flag that some operations may change. Initially, its value is equal `false`.
- The operation code on which the carriage stands. Prior to the battle, the value of this variable was not set.
- Cycle in which the operation was last performed `live`
- The number of cycles remaining until the operation on which the carriage stands
- Current carriage position
- The number of bytes that will need to be "crossed" to be on the next operation
- Registers, the number of which is set in a constant `REG_NUMBER`

  When the carriage is initialized, the values of its registers are set at the beginning of the game. The `r1`player's identification number will be written on the code of which is the carriage. Only with a minus sign. And all other registers will be initialized with zeros.

It is important to understand that the carriage does not work for a specific player. It simply executes the code it appears on. Regardless of who it belongs to:

From the point of view of the visualizer provided, the carriage knows which player it came from. Therefore, if at the beginning of the game it was located on the executable code of the "green" player or if such a carriage gave birth to it, then it will also write in memory using operations `st`/ `sti`it will also be green. Regardless of what color code it is currently on.

But from the point of view of the virtual machine, the only moment when the carriage and the player are somehow connected is the beginning of the game. At this moment, the player's identification number is written in the first carriage register, only with a minus sign.

Subsequently, during operations `fork` or the `lfork` value of all registers, the "newborn" carriage will receive from the parent carriage, and not the player.

7.4.1. Carriage list

All carriages form a list. And it is precisely in the order in this list that they will be executed. Adding new items to the list is done by pasting to the beginning. Therefore, before the start of the battle, the top will be a carriage that is on the code of the last player (the player with the highest identification number):

```
Cursor 3 (Player 3)
        |
        V
Cursor 2 (Player 2)
        |
        V
Cursor 1 (Player 1)
```

On this, all the preparatory work can be considered completed.

7.5. Player introduction

Before the start of the battle, the received champion players must declare:

```
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!") !
```

7.6. Battle

7.6.1. Battle rules

One of the most important variables in a battle is the number of cycles that have passed since the start.

During each cycle, the entire list of carriages is scanned. And depending on the state of each of them, certain actions are performed - a new operation code is set, the number of cycles before execution is reduced, or the operation itself is performed on which the carriage stands.

The battle continues until at least one living carriage remains.

That is, the carriage can die, or can someone kill it? Yes, the carriage may die. This happens during an event such as **verification** .

7.6.1.1. Check

The check occurs every `cycles_to_die`cycle until the value is `cycles_to_die`greater than zero. And after its value becomes less than or equal to zero, the check begins to be carried out after each cycle.

During this check, carriages that are dead are removed from the list.

How to determine if a carriage is dead?

A carriage that performed an operation of `live` `cycles_to_die`cycles backward or more is considered dead.

Also, any carriage is considered dead if `cycles_to_die <= 0`.

In addition to deleting carriages, the value is modified during the check `cycles_to_die`.

If the number of `cycles_to_die` operations performed during a period is `live` **greater than or equal to** `NBR_LIVE` , the value `cycles_to_die`decreases by `CYCLE_DELTA`.

The constant value `NBR_LIVE`in the provided file is equal `21`, and the value `CYCLE_DELTA`is `50`.

If the number of operations performed is `live`less than the set limit, then the virtual machine simply remembers that the check was performed.

If the `MAX_CHECKS`checks after the value `cycles_to_die`does not change, then it will be forcibly reduced by the value `CYCLE_DELTA`.

The value `MAX_CHECKS`in the sample file `op.h`is equal `10`.

To better understand when a check occurs and what it changes, consider an example:

Cycles	Number of operations `live`	`cycles_to_die`	Current number of checks
1536	five	1536	one
3072	193	1536 -> 1486	2
4558	537	1486 -> 1436	one
5994	1277	1436 -> 1386	one
7380	2314	1386 -> 1336	one
8716	3395	1336 -> 1286	one
...	...	...	...

The number of operations `live`is reset after each check, regardless of its results.

The “current number of inspections” includes the ongoing audit.

It would also be useful to consider another possible scenario for the development of the battle. Only in this case we are only interested in the latest battle cycles:

Cycles	Number of operations `live`	`cycles_to_die`	Current number of checks
...	...	...	...
24244	41437	136 -> 86	one
24330	25843	86 -> 36	one
24366	10846	36 -> -14	one
24367	186	-14 -> -64	one

Such data can be obtained during the execution of the Jinx champion, who is in the archive `vm_champs.tar` along the way `champs/championships/2014/rabid-on/`.

7.6.2. Inside the loop

Now let's take a closer look at what happens inside the loop. In each cycle, the virtual machine scans the list of carriages and performs the necessary actions on each of them:

7.6.2.1. Sets the opcode

If during the last cycle the carriage moved, then it is necessary to establish on the code of what operation it is now.

By the way, a variable storing the number of cycles until the operation is executed can indicate that the carriage was moving on the previous execution cycle. If its value is zero, then the movement took place and the carriage needs to set the code of the operation on which it is now located.

During the very first execution cycle, all carriages will receive operation code values without exception. Since during initialization it is not installed. To find out the operation code, it is necessary

to read the byte on which the carriage is located. If the received number corresponds to the code of the real operation, then it must be remembered. You also need to set the value of a variable that stores the number of cycles until the operation is completed. No other operation data (argument type code, arguments) need to be read and stored. They must be received at the time of the operation. If you remember them earlier, the virtual machine will not work correctly. Indeed, during the waiting time, the corresponding cells in the memory can be overwritten with new values. But if the read number does not fall into the range [`0x01`; `0x10`], that is, the resulting code indicates a non-existent operation? How to be in this situation? In this case, it is necessary to remember the read code, and leave the value of the variable storing the number of cycles before execution equal to zero.

7.6.2.2. Reduce the number of cycles before execution

If the number of loops before execution, which is stored by the corresponding variable in the caret, is greater than zero, it is necessary to reduce it by 1.

> It is important that during the cycle all the checks and actions described are carried out strictly in the indicated sequence.
>
> Since the same carriage can receive a new operation code in one cycle and set the number of cycles before it is executed. And also reduce this amount by one.
>
> If there was an operation with only one wait cycle, then it would also be performed during this one cycle.

7.6.2.3. Perform an operation

If the number of cycles before execution is zero, then it is time to perform the operation whose code the carriage stores.

If the stored code corresponds to an existing operation, then it is necessary to check the validity of the code containing the types of arguments.

If this code is correct and indicates that there is a register among the arguments of the operation, you must also verify that the register number is correct.

If all the necessary checks have been successfully completed, you need to perform the operation and move the carriage to the next position.

If the operation code is wrong, you just need to move the carriage to the next byte.

If everything is normal with the code itself, but the argument type code or the register number is incorrect, you must skip this operation together with the argument type code and the arguments themselves.

But if everything is easy with skipping the incorrect operation code, you just need to move to the next byte. With moving after the operation is completed, it is still transparent - the types of arguments are

known, as well as their sizes. In the case of an incorrect argument type code, one important question arises: "How to skip operation arguments if the argument type code is incorrect?"

You need to skip the operation code, the argument type code, as well as the arguments specified in the type code.

> That is why in the table of operations the sizes of the arguments were indicated `T_DIR`even for those operations that do not accept such arguments.

Let's say the code of our operations is `0x04`:

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

But the value of the byte containing the argument types is `0xb6`:

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

Since this operation takes three arguments, we consider the values of only the first three pairs of bits in the type code. The values of the remaining pairs do not interest us.

After a short calculation, we find out that in this situation the following 9 bytes must be skipped: operation code (1 byte) + byte with types (1 byte) + Type argument `T_DIR`(4 bytes) + Type argument `T_IND`(2 bytes) + Type argument `T_REG`(1 byte).

7.6.3. Flag `-dump`

In general, this is all you need to know about the operation of a virtual machine. But there is another important flag that can stop the process of its implementation. This is the flag `-dump`.

The essence of his work is described in the text of the task in the following lines:

> `-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.

An analogue of this flag is present in the original virtual machine. Only under a different name - `-d`.

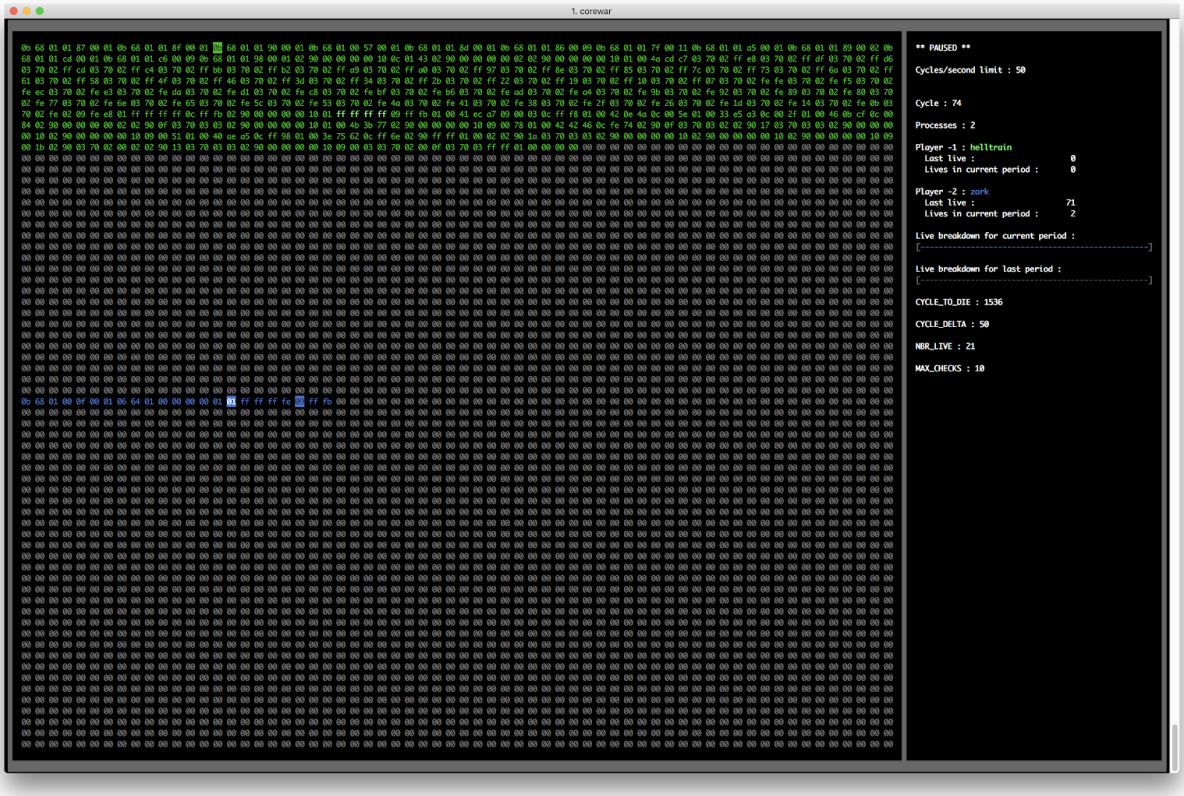
Both flags receive a loop number, after which it is necessary to display the memory status on the screen and terminate the program `corewar`. But the display modes of the arena contents for these two flags are slightly different.

The flag `-d` displays 64 octets in a row. And the flag `-dump`is only 32 octets.

8. Visualization

Visualization is a bonus part of the Corewar project, so there are a lot of options for its implementation.

In this section, we will look at the visualization approach that was provided to us as an example.



It is based on the presence of each of the players who participate in the battle, its own color.

The champion's executable code will be painted over with this color during the initialization of the arena, as well as those areas of memory that will be recorded by the carriages belonging to him using the `st` and operations `sti`.

The player's list includes a carriage, which is placed at the beginning of his source code before the battle, as well as all carriages that she then spawns using operations `fork` and `lfork`.

Pieces of memory that are not painted in the colors of one of the players will have a default color of gray.

8.1. Carriage

Also, carriages stand out in a special way in the arena. The cell on which the carriage is currently located is highlighted as follows: background - the color in which the code in the cell is painted; contents are black.

It is important that the background displays the color code of the cell itself, not the caret. That is, the carriage generated by the "green" player, once on the blue field, will glow blue.

8.2. Updated sites

Also, the fate of the memory that was recorded by the operations `st`and `sti`. The contents of these cells will be bold for the next 50 cycles.

8.3. Operation `live`

In addition to updated sections of memory, execution of the operation is also allocated `live`. The next 50 cycles, the cell that contains the code for this operation will be highlighted in the following way: background - color from the carriage that performed this operation; the contents of the cell will be white and highlighted in bold. By the way, the display of the performed operation `live` has a higher priority than the carriage display. Therefore, when superimposing these two entities, the carriage will not be visible.