

Corewar

42 project

Introduction	3
What is core war?	3
Gameplay	3
Project	3
Assembler	5
Header	5
name	5
prog_size	5
comment	5
Label:	6
Op code	6
Op code arguments	6
A quick note:	7
Separator	7
Comments	8
Source to binary	9
Header:	9
What does this means ?	10
Instructions:	11
Instruction table	11
Instruction description	12
Encoding	14
The direct type	14
The indirect type	14
Some small notes about registers	15
Encoded byte	15
Label:	19
Virtual Machine	23
What is this Virtual Machine?	23
Specifications	24
Some rules the Virtual Machine must follow	24
The instruction cycle	25
What is an instruction cycle?	25
Cycle Scheduling and example	26
Cycle example table	26
Schedule table	26
Resources and Extra	28
Representation of op_tab from op.c	28
Some useful links	29

Introduction

What is core war?

Coreware is a programming battle game started in 1984 where two or more programs battle each other to gain control of a virtual computer. The game was created by D. G. Jones and [A. K. Dewdney](#). The programs (so called 'warriors') are written in 'abstract [assembly language](#)' called [Redcode](#).

Gameplay

At the beginning each programs are loaded into memory at a random location, after which each program executes one instruction in turn. The goal of the game is to cause the process of opposing programs to terminate (which happens if they execute an invalid instruction), leaving the sole possession of the machine called MARS (Memory Array Redcode Simulator). More details could be found in later section.

Project

Much like this 1984's game we too must create a corewar game and our own 'warrior' so we can conquer this virtual world. The project consist of 3 parts

- Writing an assembler called '**asm**', so we can compile our 'warrior'.
- Writing a 'Virtual Machine called '**corewar**', which will serve as our battle field.
- Writing a '**warrior**', that will conquer and destroy other 'warriors'

That's all. So in other words we must create a [compiler](#) so we can compile our 'warrior' in a binary format that is understood by our

Virtual Machine. The specification of each parts can be found in their appropriate sections in this document.

For this project the allowed functions are:

- ❖ open
- ❖ read
- ❖ write
- ❖ lseek
- ❖ close
- ❖ malloc
- ❖ realloc
- ❖ free
- ❖ exit
- ❖ perror
- ❖ strerror

For bonuses such as an ncurses interface on terminal we might use functions from ncurses library. Or for other bonus, other libraries (if necessary).

The assembler and the Virtual Machine is written in C and the 'warrior' is written in our 'abstract assembly language' (specification for this language can be found in assembler section). So we can see that the real coding could be separated in 2 parts rather than 3 parts which are writing **the assembler** and the **Virtual Machine**.

Assembler

In [coreware](#) we must create an assembler, where we must assemble an assembly file written in a "*semi assembly*" language called [Redcode](#) and create a binary executable file representing the assembly file. An assembly file has a `'.s'` extension and a **binary** file has a `'.cor'` extension (ex: source.s, source.cor).

This "*semi assembly*" code structure is pretty simple.

In the beginning of the file there is a file header which consist of A name which contains the 'warrior' name , The size of instruction section and a comment which contains a comment. And then a set of instructions. Everything in the header part is mandatory.

Header

name

The name segment is written as following `'.name "champion name"'` it must start with a period `'.'`, must be called `'name'` and must have the two quotes `'"` even if there aren't anything inside (empty name). The maximum number of information a name can contain is 128 bytes.

prog_size

~~The instruction size section is a 2 byte number that tells how many bytes there are on the instruction section, the instruction section is what comes immediately after the comment.~~

comment

And the comment segment looks as such: `'.comment "some comment about this player"'` much like the name segment it also must start with a period `'.'`, must be called `'comment'` and must follow by two

quotes `''`. The maximum number of information a comment can contain is 2048 bytes

In this language only one instruction is given each line

An instruction is composed of 3 parts:

```
<Label> <op code> and <op_code arguments>
```

Label:

label is just a name which is optional. Label is used to identify certain region of code. And later used with op code to indicate those regions. More details could be found in later sections when we talk about source to binary it to a binary format.

Op code

(operation code) Is an instruction that a processor will carry (in our case the VM). Op codes are used to manipulate memory, doing calculations and other stuff (explained in op code section) .

Op code arguments

Op code arguments are used with op code, for example if you use an operation code to add a value to some other value, then the argument will represent the values you are adding together. We can use a maximum number of 4 arguments for each op code (`MAX_ARGS_NUMBER` in op.h).

There are 3 type of data that could be passed as argument:

An Indirect:

Which is a normal number for example "ld 4, r2" which will be interpreted as "ld (PC + 4), r2" (PC is a register explained in register section). An Indirect is equivalent of a C memory address, Thus an indirect value represent a position in memory.

A Direct:

a literal number "ld %4, r3" the '%' tells the compiler that the following number is a literal number and not a relative number to PC

A register:

register like in the example shown before, a *register* such as "ld r2, r5" or a *label* for example

I would like to note that a label can be used anywhere where a **direct** or **an indirect** could be used. To call a **label** we must put a color ':' followed by the name of the label.

Direct label call	:ld %:<label name>, r7
Indirect label call	:lld :<label name>, r7

A quick note:

Generally in assembly code an instruction argument is represented as following (Intel syntax):

```
<op code> <destination arg> <source arg>
```

```
add r2, r3
```

Where if any manipulation of a number or memory or anything to do with storing a value happens then the result is stored in dest which is

`r2` in our case, but in our “*Pseudo assembly*” language the instructions aren't represented as we just saw, but as following (AT&T [syntax](#)):

```
<op code> <source> <destination>
```

```
add r2, r3
```

The result will be stored in `r3`.

Separator

Some op code handles only 1 argument and some more than 1, if more than 1 argument is needed then they must be separated by a comma `,`

Comments

In this semi assembly language comments are also supported. Comments are represented by a hash sign `#` following by the comment you want to write. A comment ends at the end of line `\n`, that means you can put a comment at the end of an instruction or label
ex: `ld r2, r5 #some comments`
That's all, that's all there is to this “*semi assembly*” language.

Here is an example of a source code:

```
.name "zork"
.comment "just a basic living prog"

12: sti r1, %:live, %0 #any comment_1
    and r1, %0, r1
#any comment_2
live: live %1
      zjmp %:live
```


Source to binary

This part is shared with ASM part and Virtual Machine part.

The goal of the assembler is to assemble the source code given and transform it to a binary encoded file, compatible for our virtual machine. The binary file has a straightforward format. Just like a real worlds [binary executable](#), our binary file has a [header](#), followed by some instruction set.

Header without padding:

Our header would have composed of 4 parts:

Part NB	part	Size in byte
1	Magic number	4
2	Name	128
3	Instruction section size	4
4	Comment	2048

But there is one problem, The compiler provided to us in the [resources](#) produces more data in the header section then mentioned above!

And it is because To create the header section of the binary the header_t structure in [op.h](#) was used and to align the data in header_t the compile uses 2 paddings of 4 bytes each, one after the [name](#) and the other after the [comment](#), thus the header looks like this:

Header with paddings:

Part NB	part	Size in byte
1	Magic number	4
2	Name	128
3	Padding 1	4 byte
4	Instruction section size	4
5	Comment	2048
6	Padding 2	4 byte

Paddings do not contain any values thus they are represented by 0.

The Magic Number is : 0xea83f3 (information about the header can be found in [op.h](#))

For each of those parts there are spaces reserved in the header even if they are not needed. For example in the header 128 bytes are reserved even if the name consist of only 4 bytes. And same goes for the comment.

What does this means ?

Let's take a look at a code snippet and it's binary file and dissect it.

```
.name "This is some name"
.comment "Name ended comment started"

live %:lbl
lbl: live %5
```

Hex dump:

```
00000000  00 ea 83 f3 54 68 69 73 20 69 73 20 73 6f 6d 65 |....This is some|
00000010  20 6e 61 6d 65 00 00 00 00 00 00 00 00 00 00 | name.....|
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000080  00 00 00 00 00 00 00 00 00 00 0a 4e 61 6d 65 |.....Name|
00000090  20 65 6e 64 65 64 20 63 6f 6d 6d 65 6e 74 20 73 | ended comment s|
000000a0  74 61 72 74 65 64 00 00 00 00 00 00 00 00 00 |tarted.....|
000000b0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000890  01 00 00 00 05 01 00 00 00 05 |.....|
0000089a
```

'*': indicates that the pattern (zeros '0') continues (use the -v format to see the full data).

Note:

The binary codes are only a series of 0 and 1 which is really hard to understand analyse which is why generally binary datas are viewed In hexa decimal values, because it takes only 2 hex value to represent 1 byte and viewing 1 byte in hexa decimal value is more interesting Than in binary. As you can see in the above example we have Represented our binary file (which is composed of 0 and 1) in Hexadecimal values, which is done by a tool called [hexdump](#) (which can be found on most unix like system) The hexdump tool [dump](#) [Our binary data in hexadecimal format](#). If you have trouble understanding The dumped data maybe [this](#) article can help. I would like to say that The value you see in hexadecimal format are representation of the Binary data from the file.

By watching the hex dump of our code we can immediately see that the first 4 bytes represents our magic number (0xea83f3) there is a (zero at the beginning because to represent 0xea 0x83 0xf3 we need only 3 bytes and not 4 so the first zero is just an unused byte) then following after the magic number we have our name represented by 128 bytes even if they are not shown in this example (takes too much place) after 132 bytes (128 + 4) we have our comment which takes up 2048 byte. That means after 2180 bytes the instructions begin.

Instructions:

After our header, come the instructions. Let's see all available instructions in this language.

Instruction table

NB	INSTRUCTION	INSTRUCTION NB	ENCODING BYTE	MODIFY CARRY	ARGUMENTS	ARGUMENT SIZE IN BYTE	EXAMPLE
01	live	0x01	NO	NO	1	4	Live %3
02	ld	0x02	yes	YES	2	vary	ld 34, r3
03	st	0x03	yes	NO	2	?	st r4, 34
04	add	0x04	yes	YES	3	3	add r2 ,r3, r5
05	sub	0x05	yes	YES	3	3	add r2 ,r3, r5
06	and	0x06	yes	YES	3	3	and r2, r3, r5
07	or	0x07	yes	YES	3	3	or r2, r3, r5
08	xor	0x08	yes	YES	3	3	xor r2, r3, r5
09	zjmp	0x09	NO	NO	1	1	zjmp %23
10	ldi	0x0a	yes	NO	3	?	ldi 3, %4, r1
11	sti	0x0b	yes	NO	3	?	sti r2, %4, %5
12	fork	0x0c	NO	NO	1	1	fork %4

13	lld	0x0d	yes	YES	2	vary	lld 34, r3
14	lldi	0x0e	yes	YES	3	?	lldi 3, %4, r1
15	lfork	0x0f	NO	NO	1	1	lfork %4
16	aff	0x10	yes	NO	1	vary	aff r3

Instruction description

(Warning: This section will be updated soon with better information about each instruction with more details as some details are not precise for your code thus either wait for the update or ask me for more info or ask your coworkers for more info)

instruction	Description	Modify carry flag
live	The argument taken is the player number, this instruction tells the vm that the player is alive. This instruction do not take register as it parameter. (this flag is not clear enough, more details must be added)	no
ld	'ld' stands for 'Load', this instruction load the value from first parameter to the second parameter. This instruction could be used to load values from memory to a register. The second parameter must be a general register (not PC register)	yes
st	'st' stands for 'store', this instruction saves the first parameter to a memory address or a register, if the second parameter is an indirect (address) then it must be calculated with '% IND_MOD'. The first parameter must be a register.	no
add	This instruction is used to addition 2 numbers. This instruction takes 3 registers as parameter where the value contained in first 2 registers are added and stored in the 3 parameter. All three of the arguments must be registers.	yes
sub	This one works same as add the only difference is that it subtracts the values of first 2 registers then stores the result in 3rd parameter. All three of the arguments must be registers.	yes
and	This op code is same as add but it does an bitwise 'and' operation 'a & b'. All three of the arguments must be registers.	yes
or	This op code is same as add but it does an bitwise inclusive 'or' operation 'a b'. All three of the arguments must be registers.	yes
xor	This op code is same as add but it does an bitwise exclusive 'or' operation 'a ^ b'. All three of the arguments must be registers.	yes
zjmp	zjmp jumps to a certain region of memory if the carry flag (explained in flag section) is set to 1. The memory location is indicated in the argument. This flag takes only one argument and the argument must not be a register.	no
ldi	This instruction takes 3 parameter where the first two parameters are numbers which is considered addresses, that will be added together (an addition) and then the result of the sum will be considered a memory address from where the value containing the address will be loaded to the third parameter. So for example if the following is our statement 'ldi %44, %1, r2' then the manipulation will be $44 + 1 = 45$ and we will go get the value from address 45 and load it to our register r2. The last argument must be a register.	no
sti	Here the manipulation are similar to 'ldi', this instruction takes 3 parameter	no

	<p>where instead of the last parameter being the register, here the first parameter is the register, and the last 2 are values that will be summed up and be considered as a memory address where the value of our first argument (register) will be saved.</p> <p>The first parameter must be a register.</p>	
fork	<p>This instruction takes an address as parameter. The job of this instruction is to create a new process that will start the process from the index given as parameter. The new process copy different stat of the parent process. The different stats include ALL the registers values but PC register because PC flag will be the parameter given to it and the count of live instruction called by the current process.</p>	no
lld	<p>The 'lld' stands for 'long-load', it works same as 'ld' the only difference is that we do not use '% IND_MOD' with the first parameter (or ay parameter)</p>	yes
ldi	<p>Same as 'ldi' but we do not use '% IND_MOD' with our first parameter (IND_MOD will be explained later)</p>	yes
lfork	<p>lfork is similar to the fork instruction but we do not use '% IND_MOD' with our argument</p>	no
aff	<p>This instruction takes a register as its only parameter writes the value of the register to the standard output. The parameter must be a register.</p>	no

Before we start to see how we represent our registers in a binary file we must see one last thing, encoding.

Encoding

An encoded value is used after an instruction number in a binary file to represent the type of argument that is given for that instruction. As we saw before that an instruction takes register as it's argument but also 2 different type of argument (well 3, but we will talk about that later) too.

There are 2 other types of arguments that can be passed to an instruction (if the instruction accepts other than registers as its parameter).

❖ The **direct** type

- ~~The direct type is also known as a literal type by which i mean it's real value is not calculated by adding any PC (see farther in the document for more info on PC register) or other type of values, but its self is considered the real value. A direct value is written as following 'ld %4, r5' the direct value here is 4, a direct value has a '%' befor it, the '%' tells the compiler that the value given is a direct value. In our example the number 4 is loaded to register r5.~~

A direct type value is represented in 4 bytes in binary file (0xff 0xff 0xff 0xff) so the size of a direct type is of 4 bytes (DIR_SIZE)

- A label could be used as a direct type like 'ld %:some_label, r5' we will later talk about how a label is represented in a binary file.

❖ The **indirect** type

- ~~The indirect type is a number whose real value is calculated by adding the value of PC register. Let's take the example from before 'ld 4, r5' here the indirect value is 4, to get~~

~~it's real value we must add the value of PC register, if PC register contain 124 then we add $4 + 124 = 128$, and then we store 128 to register r5.~~

An indirect type value is represented by 2 bytes in binary file (0xff 0xff) so the size of an indirect is of 2 bytes (IND_SIZE)

Note:

If you are wondering for what instruction should you choose which type of parameter see the resources and extra section at the end of this document.

Some small notes about registers

By the way, registers are just memory that can hold only one byte of data (in our case) in real life registers are memory inside a processor, they are so close to the processor that it is way faster to retrieve data from a register then retrieve it from a RAM memory cell. Real life registers nowadays have more than one byte of memory, a 64 bit processor has registers that can hold number big as 64 bit (1 byte = 8 bits). In our case there are 16 general purpose registers from r1 to r16 that we can manipulate. Later in this document more details can be found about registers.

Encoded byte

Now that we know that there are more than 1 type could be used as an instruction parameter and they all represented in different size which begs us the question how then a Virtual machine know what kind of data type coming next ? well that's where encoding comes in. encoding is just a byte that comes after an instruction byte, not 'all' instruction type uses an encoded byte in front of it but the one that accepts different type of parameter uses this encoded byte. As we know we can have maximum 4 parameters for an instruction, and 3 different types of parameters, that means only 2 bits are enough to represent 3 different types and we have 4 x 2 bits in a byte, which is just perfect for our case.

Values in bits	type	Reserves size in byte
01	register	1
10	direct	Int = 4 addr = 2
11	indirect	2 (IND_SIZE)

Let's take a look at a code snippet and its binary data to compare and make sense of what we just saw.

```
.name "some name"
.comment "some comments"
```

```
ldi 3, %4, r4
```

Hex dump:

```
00000000  00 ea 83 f3 73 6f 6d 65 20 6e 61 6d 65 00 00 00 |....some name...|
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000080  00 00 00 00 00 00 00 00 00 00 00 07 73 6f 6d 65 |.....some|
00000090  20 63 6f 6d 6d 65 6e 74 73 00 00 00 00 00 00 00 | comments.....|
000000a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000890  0a e4 00 03 00 04 01 |.....|
00000897
```

If we ignore the name and comment section we can see that we have a series of hex values '0a e4 00 03 00 04 01'. Let's try to read the instruction. We can see the **first byte** is '0a' we know that just after comment section start the instruction, that means this is an instruction byte, from the instruction table we can see 0x0a is ldi which we can also see in our code snippet, great. We know that this instruction takes different type of parameter thus it needs an encoded byte to represent the type of parameters, so the **second byte** that comes just after the instruction byte is the encoded byte here the encoded byte is representing out 3 parameters, and it is 'e4' e4 in bits is '11 10 01 00'

This is indicating that our first parameter is of type **11** which is an **indirect** the second parameter is of type **10** which is a **direct** and the last one is of type **01** which represent a **register**. Now if we look again the bytes that comes after the encoded byte we can find 2 bytes representing the value 3 as we can also see in our code snippet, 2 bytes are representing the value 3 is because the size of an indirect is 2. Then comes 2 more bytes represents the value 4 and at the end 1 byte representing the number of register.

As you have noticed that in this case a direct is represented in 2 bytes but the size of a direct is 4 bytes so then how come it is represented in only 2 bytes ? well the answer is simple, if we see our instruction description of `'ldi'` we can see that the first 2 values are considered as addresses and addresses are coded in 2 bytes

Let's take another look are a snippet where we use encoded bytes:

```
.name "some name"  
.comment "some comments"
```






```
and %3, %7, r1  
and %6, r2, r7  
and r2, r5, r1  
or  r2, %2, r4  
and 3, 9, r14  
and %3, 9, r4
```

Hex dump:

00000000	00 ea 83 f3 73 6f 6d 65	20 6e 61 6d 65 00 00 00some name...
00000010	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
*			
00000080	00 00 00 00 00 00 00 00	00 00 00 30 73 6f 6d 650some
00000090	20 63 6f 6d 6d 65 6e 74	73 00 00 00 00 00 00 00	comments.....
000000a0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
*			
00000890	06 a4 00 00 00 03 00 00	00 07 01 06 94 00 00 00
000008a0	06 02 07 06 54 02 05 01	07 64 02 00 00 00 02 04T....d.....
000008b0	06 f4 00 03 00 09 0e 06	b4 00 00 00 03 00 09 04

0xa4 : 10 10 01 00
0x94 : 10 01 01 00
0x54 : 01 01 01 00
0x64 : 01 10 01 00
0xf4 : 11 11 01 00
0xb4 : 10 11 01 00

Color code

 Instructions byte (1 byte)
 Encoded byte representing different type of parameters (1 byte)
 Direct type (4 bytes)
 Indirect type (2 bytes)
 Register type (1 byte)

In this example the use of an encoded byte is illustrated more clearly, and shown the need and use of an encoded byte.

We have seen most of what there is to see in a source to binary transformation, how the source code is represented in a binary file, how there is a name section, a comment section and after first 2180 bytes the instructions began and how the instructions and the parameters are represented depending on different condition. There is one last thing to this conversion that is a little bit special, and that is Labels.

Label:

We have briefly talked about Labels in different occasions. As i mentioned earlier that Labels are just names that a label is used to identify certain region of code and could be used as a parameter for an instruction but nowhere clearly mentioned what replaces the label name exactly, for example let's take the code snippet as example:

```
.name "some name"
.comment "some comments"
and %6, %:test, r7
test: and %3, %7, r1
```

We know that after the instruction byte and the encoded byte 2 four bytes will represent the 3 and 7 as they are directs and 1 byte for the register, same goes for the next instruction after the instruction and encoded byte comes 2 four bytes representing the 2 and ... what comes after 2 ...?

Well, labels represents the position of the labeled instruction in our code in bytes. If we see a hex dump of our snippet above

Hex dump:

```
00000000  00 ea 83 f3 73 6f 6d 65 20 6e 61 6d 65 00 00 00 |....some name...|
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000080  00 00 00 00 00 00 00 00 00 00 00 16 73 6f 6d 65 |.....some|
00000090  20 63 6f 6d 6d 65 6e 74 73 00 00 00 00 00 00 00 | comments.....|
000000a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
```

```

position--00-01-02-03-04-05-06-07--08-09-01-0b-0c-0d-0e-0f-----
00000890 06 a4 00 00 00 06 00 00 00 0b 07 06 a4 00 00 00 |.....|
000008a0 03 00 00 00 07 01 |.....|
000008a6

```

We have our **0x06** representing the 'and' instruction then comes the Encoded byte representing 2 directs and 1 register, the first direct taking 4 bytes representing 6 and the other direct where our 'label' was is replaced with **0x0b** after that the register and after the register comes the next 'and' instruction byte.

So why is the label is replaced by 0x0b? Well, it's pretty simple actually, it's because at the **0x0b** position (counting from the start of instruction section in the binary file) is the instruction byte that was labeled 'test', we can see it in our hex dumped in the position line i have made for you. This value is a relative value to the position of the current instruction byte, let's take a look to understand what it means.

```

.name "some name"
.comment "some comments"

and %6, %:test, r7
and %6, %:test, r7
and %6, %:test, r7
and %6, %:test, r7
test: and %3, %7, r1

```

Hex dump:

```

00000000 00 ea 83 f3 73 6f 6d 65 20 6e 61 6d 65 00 00 00 |....some name...|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000080 00 00 00 00 00 00 00 00 00 00 00 37 73 6f 6d 65 |.....7some|
00000090 20 63 6f 6d 6d 65 6e 74 73 00 00 00 00 00 00 00 | comments.....|
000000a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
position----00-01-02-03-04-05-06-07--08-09-0a-0b-0c-0d-0e-0f
000890 0x00 06 a4 00 00 00 06 00 00 00 2c 07 06 a4 00 00 00 |.....,.....|
0008a0 0x10 06 00 00 00 21 07 06 a4 00 00 00 06 00 00 00 16 |....!.....|
0008b0 0x20 07 06 a4 00 00 00 06 00 00 00 0b 07 06 a4 00 00 |.....|
0008c0 0x30 00 03 00 00 00 07 01 |.....|

```


I have highlighted all the labels that calls 'test'. So the first is 0x2c and if we see carefully you will notice that the '06' labeled instruction byte is at 0x2c position from the instruction point of the first highlighted byte calling 'test' and the second byte that calls 'test' is '21', it's because the labeled byte is '21' bytes farther from the instruction byte of the second highlighted instruction, and this goes on, more you are near the labeled instruction more the number will be close to 0.

So what if the label calling byte is passed (after) the labeled instruction like this one:

```
.name "some name"
.comment "some comments"
```

```
and %6, %:test, r7
test: and %3, %7, r1
and %6, %:test, r7
and %6, %:test, r7
and %6, %:test, r7
```

Hex dump:

```
00000000  00 ea 83 f3 73 6f 6d 65 20 6e 61 6d 65 00 00 00 |....some name...|
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000080  00 00 00 00 00 00 00 00 00 00 00 37 73 6f 6d 65 |.....7some|
00000090  20 63 6f 6d 6d 65 6e 74 73 00 00 00 00 00 00 00 | comments.....|
000000a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000890  06 a4 00 00 00 06 00 00 00 0b 07 06 a4 00 00 00 |.....|
000008a0  03 00 00 00 07 01 06 a4 00 00 00 06 ff ff ff f5 |.....|
000008b0  07 06 a4 00 00 00 06 ff ff ff ea 07 06 a4 00 00 |.....|
000008c0  00 06 ff ff ff df 07 |.....|
000008c7
```

Well, if a label called is passed the labeled instruction then the distance is simply represented in negative value. As we use unsigned chars to represent a binary data (cause there is no negative value in a computer) so the number become so higher ([checkout this site to learn more about how negative numbers are represented in binary](#))

Note:

(Label is not important for the Virtual Machine, as all it sees are numbers and executes what asked, it doesn't need to go to an address by name)

Well that's all there is to a binary file, some op code and some values representing memory or pure number. Some additional information might be found in the Virtual Machine section.

Virtual Machine

What is this Virtual Machine?

The virtual Machine is a program called **corewar** that takes binary files with .cor extension as parameter, (compiled by the **asm** program) and executes them in the virtual environment called the arena. Here by executing i mean executing the instructions coded in the binary file.

In a arena a warrior doesn't fight with itself thus there are multiple programs executed parallely so they can battle each other to be the one who takes control.

As we are executing abstract assembly instructions that do not have any meaning to our real processor that means we must interpret them, thus simulate a processor, maybe more of a micro operating system as we are not going to 'just' execute the file but before executing we will parse it and process it THEN we will execute the instructions, and doing all those things are not the job of a processor but the job of an operating system (generally). The bare minimum needed to execute a set of instruction are some memory and a processor and a power supply.

Power : Make the processor and memory to work.

Memory : To store data for later usage by processor.

Processor : To process information.

Specifications

- This virtual machine has a total of **4096** bytes (MEM_SIZE) of memory for 4 players thus **1024** bytes for each. The memory is circular thus if a player want to reach the memory location 4098 then the virtual machine should just start counting from 0 which means after 4096 comes 0 then 1 then 2 ..., so 4098 will give you 1.
- Maximum 4 players (MAX_PLAYERS) can be passed to the Virtual Machine.
- There are **16** (REG_NUMBER) general purpose registers that can contain 4 bytes (REG_SIZE) each (r1 - r6).
- There is a Program Counter (PC) register that load the next opcode address that will be executed for the current process. Each process has their own PC register so they each know from which memory index to execute the next instruction from.
- A flag register called carry flag (CF), whose status changes from 0 to 1 if the main register in question contain 0 after any operation to be done, if not then 0 (carry flag is changed by certain instructions, see the op_tab).

Some rules the Virtual Machine must follow

1. The Virtual Machine Must verify for each active process that they have called the **live** instruction every 1536 cycle (CYCLE_TO_DIE) with their player number, if the player number used in live do not belong to the program then that's bad for it, but good for the program to whom the number belong to. (else what happens???)
2. If after 21 (NBR_LIVE) calls to **live** instruction (after using **live** 21 times), all processes are still alive then decrease the value of

CYCLE_TO_DIE by 50 (CYCLE_DELTA) (so that programs executes more and more instruction faster and faster) And this process of reducing the value of CYCLE_TO_DIE will be continued till there are no processes alive. (This one isn't very clear, what if 1 out of 3 dies will this process not be performed or the epitech subject meant that IF ANY processes are alive then perform this one ????)

3. The last valide (alive) calling the instruction `live` will be the winner
4. At each execution of the instruction `live`, the Virtual machine must show the following "Le joueur <player_number>(<player_name>) est en vie."
5. If a player wins then the Virtual Machine must show the following "Le joueur <player_number>(<player_name>) a gagné."

Exemples: Le joueur 3(zork) a gagné.

Note:

The player number is either passed by the user with the `-n` flag followed by the player number during execution (ex: `./corewar -n 2 a.cor`), each time this flag is used the warrior mentioned after the flag will take the given number else if no number is given then the player number will be in the order as they are given. The player number is only important to call the live function and mentioning that this specific player is alive and nothing else.

Most of the things there is to know about the Virtual Machine has been told but before we end the document let's clarify one last thing.

The instruction cycle

What is an instruction cycle?

A cycle is a somewhat unity to measure the speed of a processor. When we talk about cycle we talk about the clock cycle (clock being the oscillator) a cycle is between two pulses of an oscillator. Generally speaking. An instruction cycle is how many cycle an instruction takes. Which implies that an instruction may take different cycle, as this is true in real life cpu and real life instructions, this is also true for our emulated virtual cpu and our abstract assembly language instructions,

in [Resources and Extra](#) section of this document you can find information about the cycles that specific instruction takes.

Cycle Scheduling and example

Suppose we are running 3 programs in our virtual machine. As we know that this is a battle arena where multiple programs will battle each other, thus programs will be executed in parallel ((else there is no meaning to this game...)) in or case this is more of a [concurrent computing](#)). So let's say that our 3 programs has to execute the following instruction of the following cycles:

Cycle example table

Player	ins	cycle	ins	cycle	ins	cycle	ins	cycle	ins	cycle	ins	cycle	ins	cycle
A	A1	4	A2	5	A3	8	A4	2	A5	1	A6	3	A7	2
B	B1	2	B2	7	B3	9	B4	2	B5	1	B6	1	B7	3
C	C1	2	C2	9	C3	7	C4	1	C5	1	C6	4	C7	9

Then they will be executed in the following manner.

Schedule table

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
Instructions A	A1				A2					A3			
Instructions B	B1		B2							B3			
Instructions C	C1		C2									C3	
Cycle	14	15	16	17	18	19	21	22	23	24	25	26	27
Instructions A					A4			A5	A6			A7	
Instructions B						B4			B5	B6	B7		
Instructions C						C4	C5	C6					C7

So on the schedule table we can see that at the beginning all the instructions are executed parallelly but as we can see that the **A1** instruction takes 4 cycles to be completed, thus during the other 3 cycles it doesn't do anything but wait until it is completed, but

instruction **B1** and **C1** do not take 4 cycles to be completed but only 2 cycles thus they only wait 1 more cycle, after 2 cycle (at the 3d cycle) we can see that the instruction **B2** and **C2** are executed but A is still waiting to be completed, i suppose you get the point.

That's all about the processor and Virtual Machine, now all you need to do is code!!!

Resources and Extra

Representation of **op_tab** from **op.c**

Name	Param nb	Param possible	Op code	cycle	Complete name	ENCODING BYTE	Direct size
live	1	T_DIR	1	10	alive	0	0 (4 bytes)
ld	2	T_IND T_DIR, T_REG	2	5	load	1	0 (4 bytes)
st	2	T_REG, T_IND T_REG	3	4	store	1	0 (4 bytes)
add	3	T_REG, T_REG, T_REG	4	10	+	1	0 (4 bytes)
sub	3	T_REG, T_REG, T_REG	5	10	-	1	0 (4 bytes)
and	3	T_REG T_IND T_DIR, T_REG T_IND T_DIR, T_REG	6	6	&	1	0 (4 bytes)
or	3	T_REG T_IND T_DIR, T_REG T_IND T_DIR, T_REG	7	6		1	0 (4 bytes)
xor	3	T_REG T_IND T_DIR, T_REG T_IND T_DIR, T_REG	8	6	^	1	0 (4 bytes)
zjmp	1	T_DIR	9	20	Jump if zero	0	1 (2 bytes)
ldi	3	T_REG T_IND T_DIR, T_DIR T_REG, T_REG	10	25	Load index	1	1 (2 bytes)
sti	3	T_REG, T_REG T_DIR T_IND, T_DIR T_REG	11	25	Store index	1	1 (2 bytes)
fork	1	T_DIR	12	800	fork	0	1 (2 bytes)
lld	2	T_DIR T_IND, T_REG	13	10	Long load	1	0 (4 bytes)
lldi	3	T_REG T_IND T_DIR, T_DIR T_REG, T_REG	14	50	Long load index	1	1 (2 bytes)
lfork	1	T_DIR	15	1000	Long fork	0	1 (2 bytes)
aff	1	T_REG	16	2	aff	1	0 (4 bytes)

(Note: The colored instruction-name changes the carry flag)

Typedef	Name	Size in byte
T_DIR	Direct	4
T_IND	indirect	2
T_REG	register	1

Some useful links

- 01: [Corewar standard 1993 \(icws94\)](#)
- 02: [An introduction to RedCode](#)
- 03: [Tableau asm corewar](#)
- 04: [Epitech corewar subject 42 subject_\(fr\) 42 subject_\(en\)](#)
- 05: [This document in google doc \(to get in different format\)](#)
- 06: [Corewar cheat sheet](#)
- 07: [op.c op.h](#)
- 08: [VM and Asm resources \(from 42\)](#)
- :

Note:

For any notification, clarification, or anything that is not clear or you think should be changed or added than email me at uddin.samad0@gmail.com or contact me slack @sam0

Update: 09.11.2018

Changed the Cycle example table where B and C started with wrong numbers

Update: 29.09.2018

Changed the op_tab representation. The carry flag ray was replaced by encoded byte, as it was encoded byte that was represented in op_table and not carry_flag (my misunderstanding because of lack of information). The carry flag is represented in **Instruction table** and in instruction description.