Task

The task is writing program, which will **translate** binary files for EsetVm into fully working 64-bit Portable Executables.

You'll be given:

- EsetVm description,
- EVM file format description,
- sample program files written for the vm (with .evm extension),
- and an interpreter in which you'll be able to run the .evm files

Program you will provide should convert .evm file, to working .exe file.

You will also have to provide source code.

EsetVm

Architecture

VM has Harvard architecture and consist of:

- 32 signed 64-bit registers,
- linear memory addressed from 0,X
- call stack (used by call and ret instructions)

Arithmetic

Arithmetic operations use two's complement representation. Two's complement representation description.

Instruction Set

Instructions are placed in a consecutive block of memory, addressed from 0.

Execution starts at instruction 0, after every instruction **"instruction pointer"** (IP) is incremented.
Jump instructions might affect IP.

Every instruction is 3 bytes long, most instructions have format.

Opcode	Destination Argument	Source Argument
8 bits	8 bits	8 bits

instructions taking imm16 have format:

Opcode	Argument
8 bits	16 bits

Instruction	Instruction Opcode		
NOP	32	no operation	
IN reg	40	read **hexadecimal** value from standard input, and store in registry *reg*	`reg <- stdin`
OUT reg	41	write **hexadecimal** value in registry *reg* to standard output	`stdout <- reg`
STORE reg1, reg2	48	store value of *reg2* in memory cell pointed by *reg1*	`[reg1] = reg2`
LOAD reg1, reg2	49	load value from memory cell pointed by *reg1* into register *reg2*	`reg1 = [reg2]`
LDC reg, imm8	50	load 8-bit immediate value to *reg*	*reg* `= imm8`
MOV reg1, reg2	64	copy value from register2 to register1	`reg1 = reg2`
ADD reg1, reg2	65	add value of *reg2* to *reg1*, and save result in *reg1*	`reg1 = reg1 + reg2`
SUB reg1, reg2	66	subtract value of *reg2* from *reg1*, and save result in *reg1*	`reg1 = reg1 - reg2`
MUL reg1, reg2	67	multiplies value of *reg1* by value of *reg2* and save result in *reg1*	`reg1 = reg1 * reg2`
DIV reg1, reg2	68	divides value of *reg1* by value of *reg2* and save result in *reg1*	`reg1 = reg1 / reg2`
MOD reg1, reg2	69	calculates reminder of division of *reg1* by *reg2* and save result in *reg1*	`reg1 = reg1 % reg2`
JZ reg, imm8	97	if value of *reg* is zero, does relative jump. Behavior of jumps is described bellow	`if reg1 == 0:` `  IP = IP + imm8`
JL reg, imm8	98	if value of *reg* is less then zero, does relative jump. Behavior of jumps is described bellow	`if reg1 < 0:` `  IP = IP + imm8`
JUMP imm16	99	unconditional relative jump by imm16. Behavior of jumps is described bellow	`IP = IP + imm16`
CALL imm16	100	stores next instruction pointer on internal stack and jumps by imm16. Behavior of jumps is described bellow	`save IP of next instruction to call stack` `JMP imm16`
RET	101	reads absolute instruction pointer from stack and jumps to it (returns to next instruction after corresponding CALL)	`restore IP from call stack` `IP = saved_IP`
HLT	126	ends program, terminates execution	

Memory

Memory is linear, addressing starts from 0. Memory has byte-level addressing.

Instructions accessing memory, always access 64-bit values.

Data in stored and read from memory in little-endian format (see Endianness).

Example:

Let's assume we have following bytes in memory (hex values):

```
aa bb cc dd  11 22 33 44  55 66 77 88  99 00 ee ff
```

The following program

```
LDC reg0, 1
LOAD reg1, reg0
```

Will load value of 0x5544332211ddccbb into register *reg1*

EVM file format

File format consists of three segments: header, code section and initial data section values

Header

Header consist of 8 byte magic value "ESET-VM1" followed by 3 32-bit values: size of code (in instructions), size of whole data section (in bytes) and size of initialized data size (in bytes) and can be described using C structure like:

```
struct Header
{
    char magic[8];
    uint32_t codeSize;
    uint32_t dataSize;
    uint32_t initialDataSize;
};
```

All values in header are stored in little endian.

Valid file format has:

- *dataSize* >= *initialDataSize*
- *magic* == "ESET-VM1"
- *codeSize* * 3 + *initialDataSize* + 20 == size of file

Code

After header, instruction block follow. Exactly *codeSize* instructions are specified, giving this section *3 * codeSize* bytes length.

Each instruction starts with opcode byte (see "Instruction Opcode" column in instruction table). Two bytes follow and their interpretation depends on the argument type:

- Register reference (reg or regN in table) is stored as single byte where 0 marks first register and 31 last. Any value above 31 is invalid.
- Immediate (imm8) is stored as single byte
- Long immediate (imm16) is stored as two bytes in little endian.

For example LDC reg5, 33 will be encoded as 0x32 0x05, 0x21

Some instructions might NOT use argument bytes (i.e. OUT), in such case excessive byte(s) are ignored.

Data section

Data section may be initialized with data loaded from file. If *initialDataSize > 0*, then *initialDataSize* bytes are read from file and copied to the beginning of the memory. Any non-initialized data (*dataSize* - *initialDataSize* bytes) is then **initialized to zero**.

Jump encoding

Jump/call target offset is always decoded as relative to current instruction pointer. Immediate of jump instruction is interpreted as signed integer of proper size (signed char or signed short) and added to next instruction pointer to get target instruction pointer.

- Given absolute jump at 4th instruction, jumping to 5th instruction, it will be encoded as 0x63, 0x00, 0x00
- Given absolute jump at 4th instruction, jumping to 6th instruction, it will be encoded as 0x63, 0x01, 0x00
- Given absolute jump at 4th instruction, jumping to 4th instruction (looping in place), it will be encoded as 0x63, 0xFF, 0xFF
- Given absolute jump at 4th instruction, jumping to 3th instruction, it will be encoded as 0x63, 0xFE, 0xFF

Interpreter

You are provided with interpreter for the EVM files. Interpreter is a console application that takes EVM file as first argument and executes it. Standard input and standard output is used for IN / OUT opcodes.