CORE SOFTWARE DEVELOPER

JOINESET.COM

TASK

The task is writing program in C++, which will execute the provided EsetVm2 programs.

You'll be given:

- ✓ EsetVm2 bytecode description,
- ✓ EVM2 file format description,
- ✓ EVM2 compiler in Python,
- ✓ and sample programs written for the VM.

Program you will provide should execute .evm file. All provided correct sample programs should execute correctly and produce expected output. Execution environment must provide a way to specify console and binary inputs / outputs as required by instructions.

You will also have to provide source code.

ESETVM2

Architecture

VM has Harvard architecture and consist of:

- ✓ 16 signed 64-bit registers
- ✓ Fixed size linear memory addressed from 0
- ✓ Call stack (used by call and ret instructions)
- ✓ Multithreaded execution unit

Arithmetic

Arithmetic operations use two's complement representation.

Instruction Set

Instructions are placed in a stream of bits, starting at logical offset 0.

Instructions use variable-length encoding. Each instruction starts with a unique bit sequence, followed by arguments. Instructions are addressed using their bit address and are not necessary aligned to byte boundary.

Instruction bitstream encoding

Instructions and their arguments are stored in a single bit stream. Bytes that form bitstream are encoded in big endian – most significant bit of the byte is the first bit of stream and least significant last. For example, given a byte sequence A2 20, it will be converted to **1010 0010 0010 0000** sequence and processed in that order.

Instructions are not aligned and start right after each other – even if they don't fill whole byte.

Supported instructions

Instruction	Opcode bit sequence	Interpretation
mov arg1, arg2	000	arg2 <- arg1
loadConst constant, arg1	001	arg1 <- constant
add arg1, arg2, arg3	010001	arg3 <- arg1 + arg2
sub arg1, arg2, arg3	010010	arg3 <- arg1 - arg2
div arg1, arg2, arg3	010011	arg3 <- arg1 / arg2
mod arg1, arg2, arg3	010100	arg3 <- arg1 % arg2
mul arg1, arg2, arg3	010101	arg3 <- arg1 * arg2
compare arg1, arg2, arg3	01100	arg3 <- -1 if arg1 < arg2 arg3 <- 0 if arg1 == arg2 arg3 <- 1 if arg1 > arg2
jump address	01101	Move instruction pointer to address.
jumpEqual address, arg1, arg2	01110	Move instruction pointer to address if arg1 == arg2.
read arg1, arg2, arg3, arg4	10000	Read from binary input file using arg1 – offset in input file arg2 – number of bytes to read arg3 – memory address to which read bytes will be stored After read operation, arg4 receives amount of bytes actually read – may be less than arg2, if not enough data exists in input file.
write arg1, arg2, arg3	10001	Write to binary output file using arg1 – offset in output file arg2 – number of bytes to write arg3 – memory address from which bytes will be written If requested offset is larger than current output file size, file should be padded with zeroes.
consoleRead arg1	10010	Read hexadecimal value from console and store to arg1
consoleWrite arg1	10011	Write arg1 to console, as hexadecimal value.
createThread address,	10100	Create a new thread, starting at address and store it's

arg1		identifier to arg1. New thread starts with copy of current thread's registers.
joinThread arg1	10101	Wait till thread identified using arg1 ends and dispose its state. Threads will only be joined once.
hlt	10110	End current thread. If initial thread is ended, end whole program.
sleep arg1	10111	Delay execution of current thread by arg1 milliseconds.
call address	1100	Store address of instruction after the call to internal stack and continue execution at address.
ret	1101	Take address from internal stack and continue execution from it.
lock arg1	1110	Lock synchronization object identified by arg1.
unlock arg1	1111	Unlock synchronization object identified by arg1.
mov arg1, arg2	000	arg2 <- arg1

Argument encoding

Three types of arguments exist, depending on the instruction used.

- ✓ Code address is encoded using 32bit unsigned integer, little endian bitwise. Code address identifies offset in code bit stream.
- ✓ Constant argument is encoded as 64bit signed integer, little endian bitwise.
- ✓ Data access arguments are identified in table using argX. Argument is encoded using bit sequence:

Opcode	Interpretation
0 xxxx	Read xxxx as little endian register index, use that register's value or store to that register.
1 ss xxxx	Read xxxx as little endian register index, read its value as memory address. Decode ss as memory access size: 00 – one byte (byte) 01 – two bytes (word) 10 – four bytes (dword) 11 – eight bytes (qword) If argument is used in read context, read requested amount of bytes from memory and fill rest of register with zeros. If argument is used in write context, store requested amount of bytes, starting with least significant (little endian order).

Example

Given sequence bitsequence 0100010100010101000 1100

1. Read bits till a valid opcode is decoded, in this example it will be add, with sequence 010001
2. Parse arguments, as described in instruction table, in add case, it will be 3 data access arguments:
   a. First argument, starts with 0, so it will be a register argument 0-1000 1000 read a little endian is 1, so first argument is register1.
   b. Second argument starts with 1, so it will be a memory argument 1-01-0100 01 is a two byte access,
   c. Third argument starts with 0 again, so it’s a register argument: 0-1100 this encodes register3.
3. Decoded instruction is thus add register1, word[register2], register3 010001 0-1000 1-01-0100 0-1100

Memory

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

```
loadConst 1, reg1
mov qword[reg1], reg0
```

will load value of 0x5544332211ddccbb into register reg0.

Threading model

Evm2 supports threads and thread synchronization. Each thread has its own set of registers and its own call stack. Newly created thread starts with a copy of registers of creating thread, and empty call stack. Initial program thread starts with all registers set to zero and empty call stack.

- ✓ Memory access is not atomic and threads must use locks to ensure predictable execution state.
- ✓ All created threads must be joined before main thread exists.
- ✓ Locks are identified using numerical index (64 bit). Thread may hold any number of locks at once.
- ✓ It is an undefined behavior to lock same lock multiple times within same thread – locks are not guaranteed to bereentrant.

EVM2 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-VM2" 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-VM2"
- ✓ codeSize + initialDataSize + sizeof(Header) == size of file

Code

After header, instruction stream follow. Exactly codeSize bytes are specified. If instruction bit stream length does not divide by 8, it is padded with 0 bits. For example, if bitstream would end with 10110, 000 would be added to form 10110000 and 0xB0 stored as last byte. Padding is never executed by any valid programs. Instruction set encoding has been described in previous section.

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