Your task is write an emulator for a hypothetical stack machine operating on 32 bit integers with the following instructions:

PUSH	v	-- push v (an integer constant) on the stack
RVALUE	l	-- push the contents of variable l
LVALUE	l	-- push the address of the variable l
POP		-- throw away the top value on the stack
STO		-- the rvalue on top of the stack is place in the lvalue below it and both are popped
COPY		-- push a copy of the top value on the stack
ADD		-- pop the top two values off the stack, add them, and push the result
SUB		-- pop the top two values off the stack, subtract them, and push the result
MPY		-- pop the top two values off the stack, multiply them, and push the result
DIV		-- pop the top two values off the stack, divide them, and push the result
MOD		-- pop the top two values off the stack, compute the modulus, and push the result
NEG		-- pop the top value off the stack, negate it, and push the result
NOT		-- pop the top value off the stack, invert all the bits, and push the result
OR		-- pop the top two values off the stack, compute the logical OR, and push the result
AND		-- pop the top two values off the stack, compute the logical AND, and push the result
EQ		-- pop the top two values off the stack, compare them, and push a 1 if they are equal, and a 0 if they are not
NE		-- pop the top two values off the stack, compare them, and push a 1 if they are not equal, and a 0 if they are equal
GT		-- pop the top two values off the stack, compare them, and push a 1 if the first operand is greater than the second, and a 0 if it is not
GE		-- pop the top two values off the stack, compare them, and push a 1 if the first operand is greater than or equal to the second, and a 0 if it is not
LT		-- pop the top two values off the stack, compare them, and push a 1 if the first operand is less than the second, and a 0 if it is not
LE		-- pop the top two values off the stack, compare them, and push a 1 if the first operand is less than or equal to the second, and a 0 if it is not
LABEL	n	-- serves as the target of jumps to n; has no other effect
GOTO	n	-- the next instruction is taken from statement with label n
GOFALSE	n	-- pop the top value; jump if it is zero
GOTRUE	n	-- pop the top value; jump if it is nonzero
PRINT		-- pop the top value off the stack and display it as a base 10 integer
READ		-- read a base 10 integer from the keyboard and push its value on the stack
GOSUB	l	-- push the current value of the program counter on the call stack and transfer control to the statement with label l
RET		-- pop the top value off the call stack and store it in the program counter
ORB		-- pop the top two values off the stack, compute the bitwise OR, and push the result
ANDB		-- pop the top two values off the stack, compute the bitwise AND, and push the result
XORB		-- pop the top two values off the stack, compute the bitwise XOR, and push the result
SHL		-- pop the top value off the stack, logical shift the bits left by 1 bit, and push the result
SHR		-- pop the top value off the stack, logical shift the bits right by 1 bit, and push the result
SAR		-- pop the top value off the stack, arithmetic shift the bits right by 1 bit, and push the result
HALT		-- stop execution

For two operand instructions, the operand on top of the stack is the second operand, and the one immediately below it is the first operand

The numeric opcodes are as follows:

HALT	**0**
PUSH	**1**
RVALUE	**2**
LVALUE	**3**
POP	**4**
STO	**5**
COPY	**6**
ADD	**7**
SUB	**8**
MPY	**9**
DIV	**10**
MOD	**11**
NEG	**12**
NOT	**13**
OR	**14**
AND	**15**
EQ	**16**
NE	**17**
GT	**18**
GE	**19**
LT	**20**
LE	**21**
LABEL	**22**
GOTO	**23**
GOFALSE	**24**
GOTRUE	**25**
PRINT	**26**
READ	**27**
GOSUB	**28**
RET	**29**
ORB	**30**
ANDB	**31**
XORB	**32**
SHL	**33**
SHR	**34**
SAR	**35**

All instructions for this machine are 32 bits (4 bytes) long, with the following format: Bits 32-22 are ignored, bits 21-16 hold the opcode, and bits 15-0 hold the operand. (If there is no operand, those bits are filled with zeroes, but otherwise ignored.)

Your interpreter should read a set of machine instructions from a binary file (in big-Endian format) whose name is passed as a command-line argument and load those instructions into your code memory, stopping when detecting the end-of-file. Then your program counter should be initialized to 0 and the interpreter should run until a HALT instruction is detected.

For example, if your C program is called vm and your code is named myprog.bin, then use the command line:

> **`vm myprog.bin`**

Similary, for a java program, the command line would be

> **`java Vm myprog.bin`**

(In the above example, for C, argv[0] = “vm” and argv[1] = “myprog.bin”; for Java, args[0] = “myprog.bin”.)

We will use the Harvard memory model, with two separate 256KB (65,536 32-bit words) for instructions and data. The memory will be word-addressable, that is, a 32 bit word is the smallest addressable memory location. Note that implication of having word addressability means that the location counter will be incremented by 1 rather than by 4 at each step.

The stack implementation is up to you. You can use dynamic memory allocation, but it would be more “realistic” if you use a third memory segment of fixed size as your stack segment. It would be even more realistic if you allocated your stack in the data segment, letting in grow downward from the highest memory locations, of course. For the subprogram call/return mechanism, use a dedicated call stack, instead of the operand stack the other instructions use.

You may use any language you wish to implement your emulator, but it must interpret the appropriate binary code in Big Endian format. If you use Java, you may use any platform you wish. If you use any other language, you must either provide a makefile or complete compilation instructions for generating an executable under Linux.

Hand in your source code and compilation instructions via Blackboard.