Architecture Overview

The machine consists of

- an array of instructions,
- an integer stack for memory,
- 3 special purpose registers,
- and a single general purpose register.

The Stack Machine

The structure of the virtual machine *might* be defined as follows (You are encouraged to explore as you understand):

```
class StackVirtualMachine
{
      int generalPurposeRegister; //General Purpose Register;

      //instruction model
      String[] instructions = new String[1024];
      int programCounter;     //next instruction to execute

      //stack memory model
      int[] memoryStack = new int [4096];
      int stackTop=0;         //next open slot in memory stack
      int stackPointer;       //current frame
}
```

You encouraged to improve or extend the model. For example rather than:

```
      private String[] instructions = new String[1024];
```

a more efficient model might be:

```
      private Command[] instructions = new Command[1024];
```

where abstract class Command is left to your imagination.

The Instruction Set

The instruction set consist of the following commands:

- call
- ret
- retv
- push
- pop
- dup
- load
- save
- store
- add
- sub
- mul
- div
- rem
- eq
- ne
- le
- ge
- lt
- gt
- brt
- brz
- jump
- print
- read
- end
- label

You are not required to implement this list exactly as presented. If you prefer “mult” to “mul” or “equal” to “eq” you may implement either or both. Maintain documentation of your extensions or changes.

You may augment/modify this list using synonyms. You may even extend this command set with any reasonable command or variation. Maintain documentation.

Comments are not part of the language but it is recommended that you allow a form of comment. A suggestion is to begin comments with a semi colon and let them carry to the end of the line.

Command Descriptions

Each instruction consist of a keyword and some may be followed by a register indicator, an integer literal, or a string literal. Each instruction may use up to two values from the memory stack and/or register values. When multiple values are referred to on a stack, rightmost values are on top (bottom, top). The general purpose register will most often reflect the top value on the memory stack with a few exceptions (notably pop). In some cases the effect on the General Purpose register may be left undefined. It is unsafe to assume the value after these instructions (As the register may be used to store temp values during execution).

call

Pre Stack: numParms, function location
Post Stack: return address
Side Effect: Stack pointer gets new frame reference.
Description: Executes subroutine. Return address is preserved on the stack and the stack pointer is set to indicate the new frame.

ret

Pre Stack: current frame
Post Stack: previous frame
Side Effect: Stack pointer gets new frame reference.
Description: returns from subroutine. Clears current frame. Stack pointer is reset to calling frame.

retv

Pre Stack: current frame
Post Stack: previous frame and return value
Side Effect: Stack pointer gets new frame reference. General purpose register is set to return value.
Description: Returns from subroutine with a value. Clears current frame. Stack pointer is reset to calling frame. Return value is pushed to the memory stack. General purpose register is set to return value.

push

Pre Stack:
Post Stack: value
Side Effect: General purpose register is set to pushed value.
Description: Pushes a value to memory stack.

pop

Pre Stack: value
Post Stack:
Side Effect: General purpose register is set to popped value.
Description: Removes value from memory stack.

dup

Pre Stack: value
Post Stack: value, value
Side Effect:General purpose register is set to dupped value.
Description: Duplicates top value on memory stack.

load

Pre Stack: address
Post Stack: value
Side Effect:General purpose register is set to loaded value.
Description: Fetches value from memory stack.

save

Pre Stack: value, address
Post Stack: value
Side Effect: General purpose register is set to saved value.
Description: Saves value to memory. Leaves value on stack.

store

Pre Stack: value, address
Post Stack:
Side Effect:General purpose register is set to stored value.
Description: Saves value to memory.

add

Pre Stack: left, right
Post Stack: value
Side Effect:General purpose register is set to sum value.
Description: Pops two values from the stack and pushes their sum.

sub

Pre Stack: left, right
Post Stack: value
Side Effect:General purpose register is set to difference value.
Description: Pops two values from the stack and pushes their difference.

mul

Pre Stack: left, right
Post Stack: value
Side Effect:General purpose register is set to product value.
Description: Pops two values from the stack and pushes their product.

div

Pre Stack: left, right
Post Stack: value
Side Effect:General purpose register is set to quotient value.
Description: Pops two values from the stack and pushes their quotient.

rem

Pre Stack: left, right
Post Stack: value
Side Effect:General purpose register is set to remainder value.
Description: Pops two values from the stack and pushes their quotient remainder.

eq

Pre Stack: left, right
Post Stack: value
Side Effect: General purpose register is set to result value.
Description: Pops two values from the stack and pushes one or zero. One is pushed if numerically left == right.

ne

Pre Stack: left, right
Post Stack: value
Side Effect: General purpose register is set to result value.
Description: Pops two values from the stack and pushes one or zero. One is pushed if numerically left <> right.

le

Pre Stack: left, right
Post Stack: value
Side Effect: General purpose register is set to result value.
Description: Pops two values from the stack and pushes one or zero. One is pushed if numerically left <= right.

ge

Pre Stack: left, right
Post Stack: value
Side Effect: General purpose register is set to result value.
Description: Pops two values from the stack and pushes one or zero. One is pushed if numerically left >= right.

lt

Pre Stack: left, right
Post Stack: value
Side Effect: General purpose register is set to result value.
Description: Pops two values from the stack and pushes one or zero. One is pushed if numerically left < right.

gt

Pre Stack: left, right
Post Stack: value
Side Effect: General purpose register is set to result value.
Description: Pops two values from the stack and pushes one or zero. One is pushed if numerically left > right.

brt

Pre Stack: value, location
Post Stack:
Side Effect: program counter may be updated with location
Description: Sets program counter if value <> 0

brz

Pre Stack: value, location
Post Stack:
Side Effect: program counter may be updated with location
Description: Sets program counter if value == 0

goto

Pre Stack: location
Post Stack:
Side Effect: program counter is updated with location
Description: Sets program counter to location.

print

Pre Stack: value
Post Stack:
Side Effect:
Description: Prints top value from stack

read

Pre Stack:
Post Stack: value
Side Effect: General purpose register is set to read value.
Description: reads a value(int) from std-in.

end

Pre Stack:
Post Stack: value
Side Effect:
Description: program execution halts.

label

Pre Stack:
Post Stack:
Side Effect:
Description: Defines a location. A label statement consist of a label name on a line by itself. A label name may not be a command keyword and may otherwise consist of any legal identifier names. Labels may be used as jump points and substituted anywhere a pc value is expected. The line on which the label occurs is a considered a nop instruction. Labels should be unique.

Usage Variations.

Some instructions may require or have optional parameters on the command line. These parameter values may be indicate string, integer, or register content.
String values are indicated in quotes and only appear in the print instruction. Registers are named pc, sp, top. The general purpose register is not named. Comman usages are grouped.

push value

- value may be an integer literal, a register name, or blank to indicate the general purpose register.
- examples:
  - push 0
  - push 1
  - push sp
  - push top
  - push

pop register

- register may be a register name or blank to indicate the general purpose register.
- examples:
  - pop sp
  - pop
  - pop top

load register
save register
store register

- register may be sp or top to indicate frame or relative address. If the register is absent zero(global frame) is assumed.
- value and offset will always be stack values
- examples:
  - load sp
  - save sp
  - store top
  - load top
  - save
  - store
-

call label
brt label
brz label
jump label

- label may be an int literal or a label name. If present this is the location of the branch. If not present the branch location will be found on the stack. Label names may be used to indicate label statements. If the label is an integer the pc is set to that value. If the pc is a label name the value of the label name is used to set the pc.
- Examples:
  - brt 100
  - brz 50
  - jump 20
  - jump L0001
  - call foo

print string

- string if preset is printed. If the string is not present the top value on the stack is printed.
- Examples:
    - print
    - print “hello world”

end value

- value may be an integer literal, a register name, or blank to indicate general purpose register.
- examples:
    - push 0
    - push 1
    - push sp
    - push top
    - push

Execution Model

Psuedo-Code:

```
The program is read into instructions[];
pc is set to zero;
top is set to zero;
gpr is set to zero;
while instruction[pc] <> end
    //because some instructions modify pc, we increment pc first
    exec = pc;
    pc++
    execute (instructions[exec])
execute(instruction[pc]) //the end statement is executed.
```

Sample Program

We present an example program that computes a Greatest Common Divisor between two numbers using Euclid's method. We presented first the program a high-level pseudo-code.

```
int gcd(int u, int v){
      if (v == 0)
            return u;
      else
            return gcd(v, u-u/v*v);
}

void main(void){
      int x;
      int y;
      output(“\nEuclid's GCD”);
      output(“\nEnter a number:\t”);
      x = input();
      output(“\nEnter a second number:\t”);
      y = input();
      output(“\nRESULT:\t”+gcd(x,y))+”\n”);
}
```

```
      goto MAIN
GCD
      push 1
      load SP
      push 0
      eq
      brz LABEL1
      push 0
      load SP
      ret
      jump LABEL2
LABEL1
      push 1
      load SP
      push 0
      load SP
      push 0
      load SP
      push 1
      load SP
      div
      push 1
      load SP
      mul
      sub
      push 2
      call GCD
      ret
LABEL2
      ret
MAIN
      print “\nEuclid's GCD”
      push 0
      push 0
      print “\nEnter a number:\t”
      read
      push 0
      store SP
      print “\nEnter a second number:\t”
      read
      push 1
      store SP
      push 0
      load SP
      push 1
      load SP
      push 2
      call GCD
      print “\nRESULT:\t”
      print
      print ”\n”
      end
```