SaM I Am

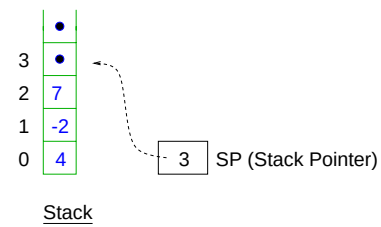
What is SaM?

SaM is a simple stack machine that we will use to

(i) understand virtual machines and Java portability, and

(ii) understand at the assembly language level how computers work.

Heart of SaM: a Stack and Stack Pointer (SP)



Stack: an array of integers
Stack grows when integer is "pushed" on top.
Stack shrinks when integer is "popped" from top.
Stack starts at address 0 and grows to larger addresses.

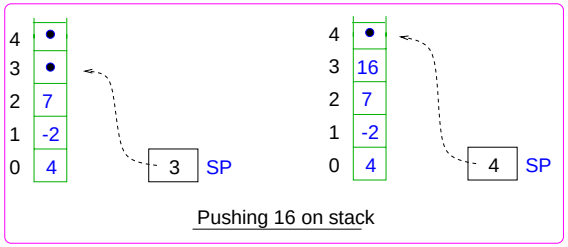
Stack pointer: first "free" address in stack
(initialized to 0)

SaM commands

ALL arithmetic/logical operations pop values from stack
perform operation and push result.

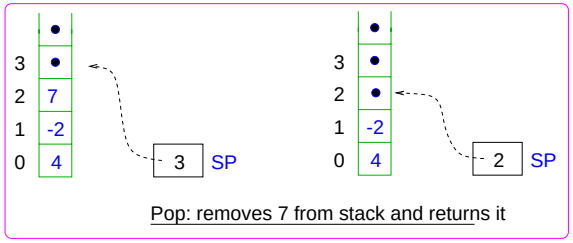
```
PUSHIMM      *some integer*
        //pushes that integer on stack
ADD
        //pops two values from top of stack
        //adds them and pushes result
SUB
        //pops two values (say top and below)
        //and pushes result of doing (below - top)
TIMES
GREATER
      // boolean values are simulated using 0/1 (false/true)
AND
      //logical AND
..........
STOP  //terminate execution of program
```


Operations on Stack



Pushing 16 on stack

Pushing an Integer

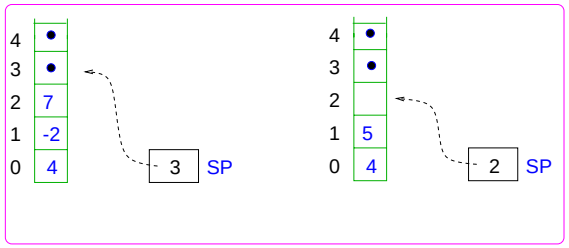


Pop: removes 7 from stack and returns it

Popping an Integer

Stack operations are used to implement SaM commands.
They are NOT SaM commands themselves.

SaM Commands

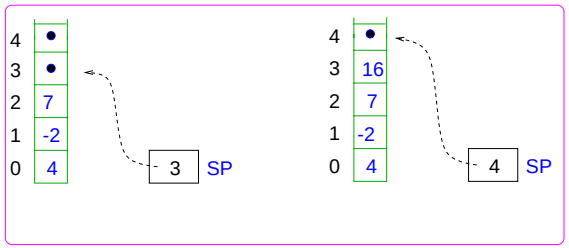


ADD

- pop two values from stack (7 and -2)
- add them (5)
- push result

SUB: similar; result would be (-2) - (7) = -9

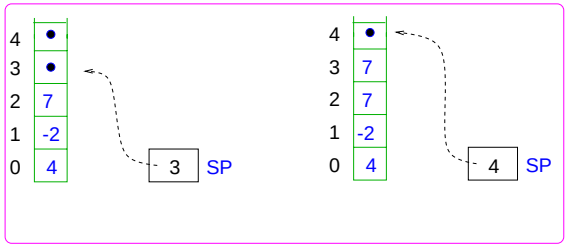
SaM Commands



PUSHIMM 16

- push the integer 16 on stack

SaM Commands

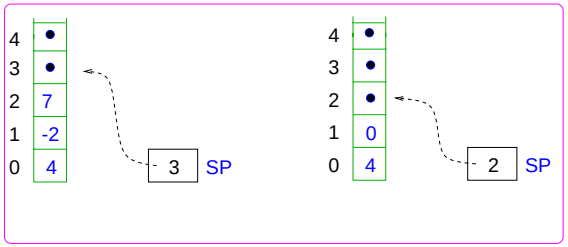


Booleans are simulated in SaM with integers
True -> 1, false -> 0

GREATER

- pop two values (Vtop and Vbelow) from stack
- in example, Vtop = 7 and Vbelow = -2
- if Vbelow > Vtop push 1 else push 0
- in example, we would push 0.

SaM Commands



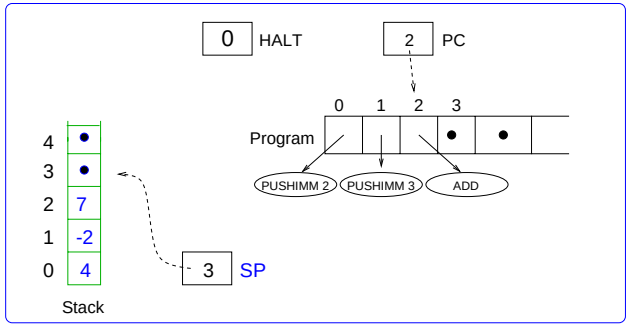
DUP :
duplicate the top element of the stack
note: nothing is popped from stack

Here are two simple SaM programs:

```
PUSHIMM 5
PUSHIMM 4
PUSHIMM 3
PUSHIMM 2
TIMES
TIMES
TIMES
STOP //should leave 120 on top of stack
```

```
PUSHIMM 5
PUSHIMM 4
GREATER
STOP  //should leave 1 on top of stack
```

How do commands get into Program array?

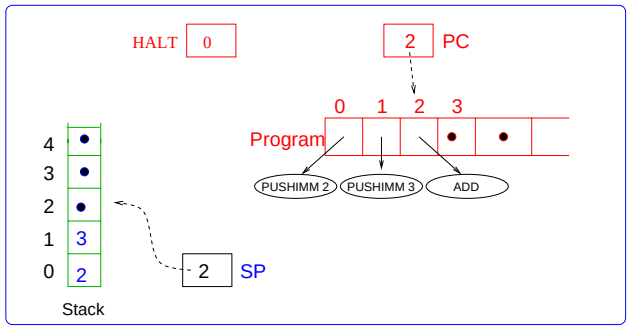


Loader:

method that is given a filename
reads in commands from file and writes them into Program array

We will assume all programs are loaded into program memory starting at address 0.

Where does program reside in SaM?



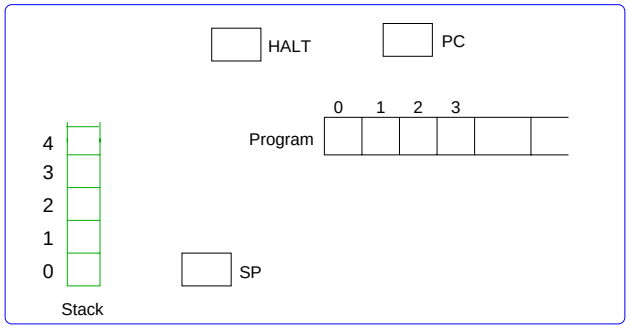
Program: an area of memory that contains commands

Program Counter (PC): address of command to be executed

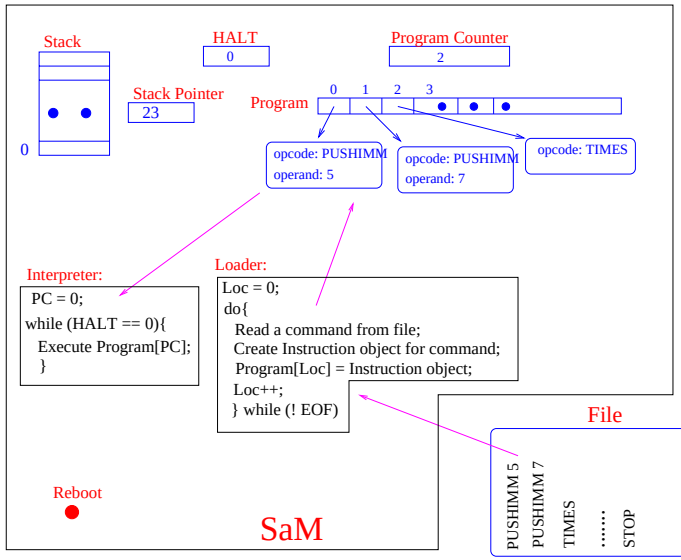
Program execution:

```
PC = 0;
while (HALT == 0)  //STOP command sets HALT to 1
    Execute command Program[PC];
        //arithmetic and logical commands increment PC
```


Sample program execution:



PUSHIMM 5
PUSHIMM 4
GREATER
STOP



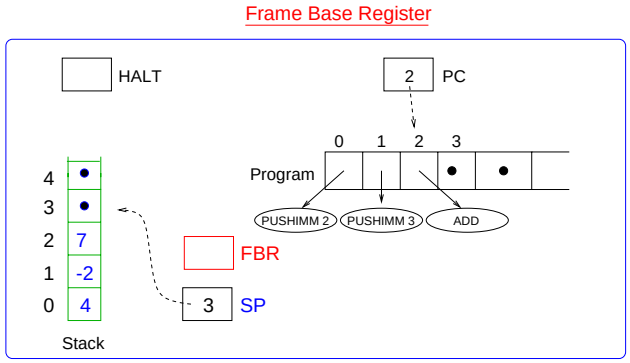
Jumps and Conditional Jumps

So far, SaM commands were executed sequentially: each command incremented PC when it was done.

Need more flexibility to translate conditionals and loops:

- ability to skip some commands
- ability to repeatedly execute some sequence of commands

Solution: JUMP/JUMPC commands



Frame Base Register: used to implement method invocations

for now, assume it contains 0


```
PUSHIMM 5
PUSHIMM 4
GREATER     //is 5 > 4
JUMPC 5     //if so, jump to 5
PUSHIMM 10
PUSHIMM 12  // this command will be stored at Program[5]
STOP
```

To understand this program, let us look at where these commands are stored in the Program area.

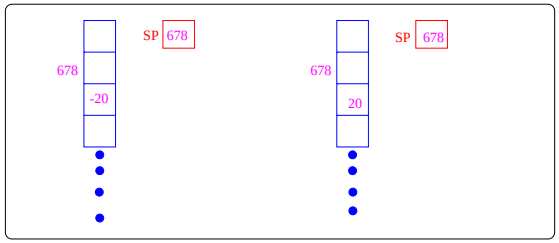
JUMP/JUMPC: like GOTO's in Pascal

- JUMP t: jump to instruction at `Program[t]` and start executing instructions from there on. Implementation: $PC \leftarrow t$
- JUMPC t: pop boolean from stack; if it is true, execute commands starting at `Program[t]`; otherwise, this command has no effect and execution continues with command after this one.

  Implementation: pop top of stack (V_{top}). If V_{top} is true, $PC \leftarrow t$; otherwise, $PC++$.

Write a program to find absolute value of element on top of stack.

Program




```
DUP //duplicate top of stack
PUSHIMM 0
GREATER  //n > 0 ?
JUMPC 6   //forward jump to STOP
PUSHIMM -1
TIMES //(n)*(-1)
STOP
```



```
Program
Address
-------
0:        PUSHIMM 5
1:        PUSHIMM 4
2:        GREATER     //is 5 > 4
3:        JUMPC 5     //if so, jump to 5
4:        PUSHIMM 10
5:        PUSHIMM 12
6:        STOP
```



```
 //work-horses
 void reboot();
 int load(String s); //input:file name, returns number of commands lo
 void start(); //start execution
 void dumpState();//print out registers and stack contents
 ......//other methods we will see later
}
```



```
interface SaM{
 //getter and setter methods
 void setFBR(int v);
 int  getFBR();
 void setPC(int v);
 int  getPC();
 void setSP(int v);
 int  getSP();
 void setStackElement(int a, int v); //Stack[a] = v;
 int  getStackElement(int a); //return Stack[a]

 //convenient setter and getter methods
 void push(int v);
 int  pop();
 void incPC();
```

One approach to writing a SaM simulator:

```
class Simulator { //main class
 ....
 }
class SaMmy implements SaMInterface{
 ...
 }
class Instruction implements InstructionInterface{
 ...
 }
```



```
interface InstructionInterface {
 void execute(SaM machine);
}
```



```
class Instruction implements InstructionInterface{
  private String opcode;
  private int operand; //optional
 public Instruction(CS211In f) {
   //read one command from file and make Instruction
  }
 public String toString() {
   //return string representing command
 }
 public void execute(SaM machine) {
   //execute instruction
 }
```



```
class Simulator {
 public static void main(String[] arg) {
   if (arg.length == 0) {
     System.out.println("sam file must be command line argument");
     return;}
   SaM Sumatra = new SaMmy(); //My machine is called Sumatra
   int pSize = Sumatra.load(arg[0]);
   Sumatra.start();
   Sumatra.dumpState();
 }
}
```

```
public void execute(SaM machine){
    //for executing the instruction

   if (opcode.equals("ADD")){
          int op2 = machine.pop();
          int op1 = machine.pop();
          machine.push(op2 + op1);
          machine.incPC();
 }
   else
   if (opcode.equals("PUSHIMM"){
    ....
    ....//other opcode
```



```
public Instruction(CS211In inf) {
    opcode = inf.GetWord();
    if (opcode.equals("PUSHIMM"))//only command with an explicit operand
       operand = inf.GetInt();

}
public String toString(){
    if (opcode == PUSHIMM)
        return ("PUSHIMM " + operand);
    else
        return opcode;}
}
```



```
class SaMmy implements SaM{
 ...
    public void start() {
      //Execute instructions starting at Program[0]
        HALT = 0;
        PC = 0;
        do {
            Program[PC].execute(this);
           }
        while(HALT == 0);
     }
 ...
}
```



```
class SaMmy implements SaM{
    protected int PC,FBR,SP,HALT;
    protected InstructionInterface[] Program = new InstructionInterface[300]
    protected int[] Stack  = new int[300];

    public int load(String ProgramFile) {
       //Load in instructions from ProgramFile and store in Program
       CS211In inf = new CS211In(ProgramFile);//create input manager
       int LC = 0;//LC is where next instruction should be put
       while (inf.peekAtKind() != inf.EOF)
           { Program[LC] = new Instruction(inf);
             LC++;
           }
       inf.close();
       return LC; }
    ...
}
```

Modular solution: USE SUBTYPING

Represent each opcode by its own class.

Adding new opcode = adding new class to program!

How can we still store different types of instructions into Program array in SaM?

Answer: all classes implement InstructionInterface!

Happily, type of Program is `InstructionInterface[]`!

Critique of solution:

Adding new command => changing a bunch of variables and methods in the `Instruction` class:

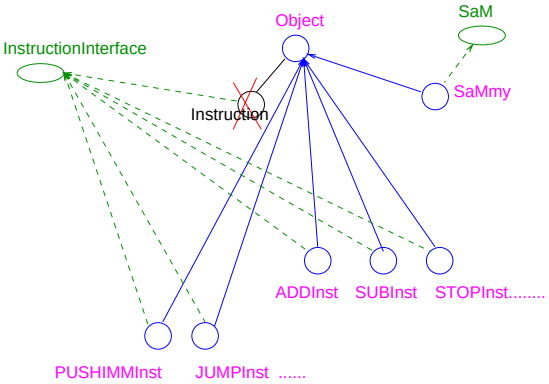
opcodeTable, constructor, toString, execute,

No easy way to extend the code to accommodate more commands.

If you wrote this sort of code, you might as well program in a non-OO language!!


```
class ADDInst implements InstructionInterface {
    public String toString() {
       return "ADD";
     }
    public void execute(SaM machine) {
      int op2 =  machine.pop();
      int op1 =  machine.pop();
      machine.push(op2 + op1);
      machine.incPC();
     }
   }
```


Type Picture



How does this change `SaM` class?

No change needed to `start` method!!

```
class SaMmy implements SaM{
 ...
    public void start() {
      //Execute instructions starting at Program[0]
        HALT = 0;
        PC = 0;
        do {
            Program[PC].execute(this); //type dispatch!!
           }
        while(HALT == 0);
     }
 ...
}
```



```
class SUBInst implements InstructionInterface {
    public String toString() {
       return "SUB";
     }
    public void Execute(SaM machine) {
      int op2 =  machine.pop();
      int op1 =  machine.pop();
      machine.push(op1 - op2);
      machine.incPC();
     }
   }
```


Solution: Let `load` read in opcode and create an instance of the right class.

```
class SaMmy implements SaM{
    public int load(String ProgramFile) {//override Load in SaM

       CS211In inf = new CS211In(ProgramFile);//create input manager
       int LC = 0;//LC is where next instruction should be put

       ReadLoop:
       while (inf.peekAtKind() != inf.EOF)
           { String s = inf.getWord();
             InstructionInterface i;
             if (s.equals("ADD")) i = (InstructionInterface)new ADDInst();
             else
             if (s.equals("SUB")) i = (InstructionInterface)new SUBInst();
             else
             ....
```



```
class SaMmy implements SaM{
 ...
    public int load(String ProgramFile) {
       //Load in instructions from ProgramFile and store in Program
       CS211In inf = new CS211In(ProgramFile);
       int LC = 0;
       while (inf.peekAtKind() != inf.EOF)
           {
             Program[LC] = new Instruction(inf); <--- does not make sense
             LC++;
           }
       inf.close();
       return LC;
      }
 ...
}
```

There is no `Instruction` class anymore to instatiate!

Criticism of load code: still need a conditional that must be modified when opcode is added!

Here's some cool code: given a String representing a class name, create an instance of that class!

```
public int load(String ProgramFile) {
      CS211In inf = new CS211In(ProgramFile);//create input manager
      int LC = 0;//LC is where next instruction should be put

      ReadLoop:
      while (inf.peekAtKind() != inf.EOF)
          { String s = inf.getWord();
            InstructionInterface i;
            try  { i = (InstructionInterface)(Class.forName(s + "Inst")).newInstance();
                  } catch (Exception e)
                    {System.out.println("Funny opcode " + s);
                     break ReadLoop;
            }
```



```
            //now read in operands
            i.readOperand(inf); //must extend InstructionInterface to
                                //require readOperand method
            Program[LC] = i;
            LC++;
          }
      return LC;
     }
}
```


Java note: Why can't reading of operands be performed by constructors in the ADDInst, SUBInst etc. classes?

Problem: if you use reflection to generate a class name on the fly, you can only call the default constructor that takes no arguments. So we cannot pass *inf* to it.


```
            i.readOperand(inf);
            Program[LC] = i;
            LC++;
          }
      return LC;}}
```

Reflection: interpret a string as a class name and create an instance of that class!

```
class PUSHIMMInst implements InstructionInterface {
    protected int operand;
    public void readOperand(CS211In inf) {
     operand = inf.getInt(); }
    public String toString() {
       return "PUSHIMM " + operand;
     }
    public void execute(SaM machine) {
      machine.push(operand);
      machine.incPC();
     }
   }
```


Here's the definition of the new instruction interface:

```
interface InstructionInterface {
 void readOperand(CS211In inf);
 void execute(SaM machine); }
.....
class ADDInst implements InstructionInterface {
    public void readOperand(CS211In inf) { //no-op
     }
    public String toString() {
       return "ADD";}
    public void execute(SaM machine) {
      int op2 =  machine.pop();
      int op1 =  machine.pop();
      machine.push(op2 + op1);
      machine.incPC();
     }}
```



```
class JUMPCInst implements InstructionInterface {
    protected int operand;
    public void readOperand(CS211In inf) {
     operand = inf.getInt(); }
    public String toString() {
       return "JUMPC " + operand;
     }
    public void execute(SaM machine) {
      if (machine.pop() == 1) machine.setPC(operand);
      else machine.incPC();
     }
   }
```



```
class JUMPInst implements InstructionInterface {
    protected int operand;
    public void readOperand(CS211In inf) {
     operand = inf.getInt(); }
    public String toString() {
       return "JUMP " + operand;
     }
    public void execute(SaM machine) {
      machine.setPC(operand);
     }
   }
```

OK, so this is cool, but there is a fair amount of code repeated.

All instructions with no operands (ADD,SUB,TIMES,GREATER,NOT,...) will have exactly the same `readOperand` method.

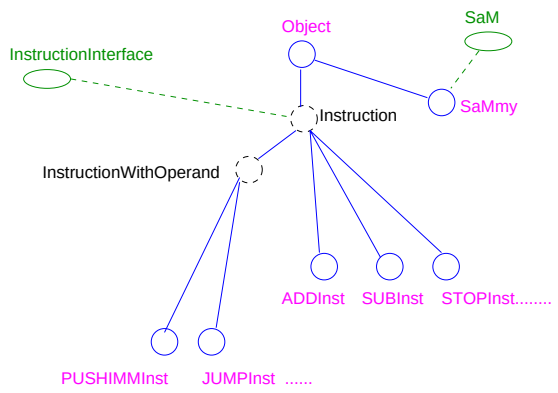
All instructions with one operand (PUSHIMM,JUMP,...) will have exactly the same `readOperand` method.

Is there a way to "share" this code between different classes?

Answer: use inheritance

Define two abstract classes:

- Instruction: contains code common to all instructions (don't confuse this with old Instruction class we defined at the start, I just ran out of meanginful names)
- InstructionWithOperand extends Instruction: contains code common to all instructions with an operand.




```
abstract class Instruction implements InstructionInterface{
    abstract public void execute(SaM machine);
    public void readOperand(CS211In inf) {//do nothing
      }
    protected String mnemonic() { //another use of reflection
      String s = this.getClass().getName();
      return s.substring(0, s.length() - ("Inst".length()));
      }
    public String toString() {
      return mnemonic();
    }
 }
```

```
class ADDInst extends Instruction { //inherit toString and readOperand!
   public void execute(SaM machine) {
      int op2 =  machine.pop();
      int op1 =  machine.pop();
      machine.push(op2 + op1);
      machine.setPC(machine.getPC() + 1);
     }
   }
```



```
//This is the class for all instructions with one operand.
//The operand can be a word or an integer.
abstract class InstructionWithOperand extends Instruction{
    protected int operand;
    public String toString() {
      return  mnemonic() + " " + operand;
    }
    public void readOperand(CS211In inf) {
      if(inf.peekAtKind() == inf.INTEGER)
         operand = inf.getInt();
      //else should report error
    }
}
```


Final twist to SaM saga: *symbolic labels* in SaM commands

So far, JUMP and JUMPC commands (in file) took an integer operand.

Example: JUMP 678

It is more convenient to use *symbolic labels* for commands.

- Label: a name that starts with a letter and contains letters and numbers
- Any command can be given a label.
- A command may have multiple labels.


```
class JUMPInst extends InstructionWithOperand{//inherit toString,readOperand
    public void execute(SaM machine) {
      machine.setPC(operand);
    }
}
```

```
        PUSHIMM 22  //some value n
        DUP //duplicate top of stack
        PUSHIMM 0
        GREATER  //n > 0 ?
        JUMPC BOND   //forward jump to instruction labeled BOND
        PUSHIMM -1
        TIMES //(n)*(-1)
JAMES:
 BOND:  STOP
```


Question:

- Commands like JUMP take symbolic labels as operands
- Within machine, instructions like JUMP require integer addresses as operands.
- Who does the conversion from labels to integers?

Answer: loader.

Additional duty of loader:

- While loading the program, create a directory of (symbolic name, address) pairs (like a telephone directory) (eg. JAMES = 7, BOND = 7, etc.).
- Directory is called a *symbol table*.
- After loading is done, walk over instructions in Program and replace each symbolic name by address, using symbol table (eg, replace `BOND` in instruction `JUMPC BOND` by 7.

This is called a *two-pass loader*.

- Symbol table is an example of a more general data structure called *hash table*.
- Hash table maintains a set of (key,value) pairs. In our example, keys are symbolic labels and values are integers (addresses corresponding to the symbolic labels).
- Two main operations:
  - insert a new (key,value) pair
  - look up the value corresponding to a key
- Sometimes, we may also want to delete a particular (key,value) pair from the hash table.
- Designing a hash table for fast look up: see later.

```
interface InstructionInterface{
  void readOperand(CS211In inf); //reads one integer or word operand
  void execute(SaM machine);
  void translate(SymbolTable ST, SaM m); //translate symbolic addresses into integers
}
```


This requires the following changes:

- change loader so it keeps track of (name, address) pairs
- change InstructionWithOperand so it can read in and store a symbolic operand
- add a `translate` method to each instruction: given symbol table, look up symbolic operand and translate to integer if necessary.


```
  //two pass loader: first pass reads in instructions into Program array
  //but does not translate symbolic operands. second pass performs translatio
  //returns number of instructions read in
    public int load(String ProgramFile) {
       CS211In inf = new CS211In(ProgramFile);//create input manager
       return load(inf);
    }
  public int load(CS211In inf) {
       SymbolTable ST = new SymbolTable();
       int LC = 0;//LC is where next instruction should be put

       //First Pass
        LC = 0;
FirstLoop:
        while (inf.peekAtKind() != inf.EOF)
              {
               String s = inf.getWord();
               //eat up labels
```



```
            while(s.endsWith(":"))
            {String tmpstr = s.substring(0,s.length()-1);
             boolean success = ST.insert(tmpstr,LC);
                     if (! success)
                        println("Multiply defined label " + tmpstr );
                     //check for dangling label at end of file
                     if (inf.peekAtKind() == inf.EOF) break FirstLoop;
                     else s = inf.getWord();
                    }
              //at this point, s should contain the opcode
             InstructionInterface i;
               try  { i = (InstructionInterface)(Class.forName(s + "Inst")).n
                   } catch (Exception e)
                     {println("Funny opcode " + s);
                      return -1;
             }
```

```
abstract class Instruction implements InstructionInterface{
    abstract public void execute(SaM machine);

    public void readOperand(CS211In inf)  {}

    public void translate(SymbolTable ST, SaM m) {}

    protected String mnemonic() { //the opcode
      String s = this.getClass().getName();
      return s.substring(0, s.length() - ("Inst".length()));
    }

    public String toString() {
      return mnemonic();
    }
}
```



```
                i.readOperand(inf);
                Program[LC] = i;
                LC++;
              }
        inf.close();

       //Second Pass
for (int i = 0; i < LC; i++) {
  if (Program[i] == null)
     println("??? Null instruction at location " + i);
  else Program[i].translate(ST, this);
        }
return LC;
  }
```

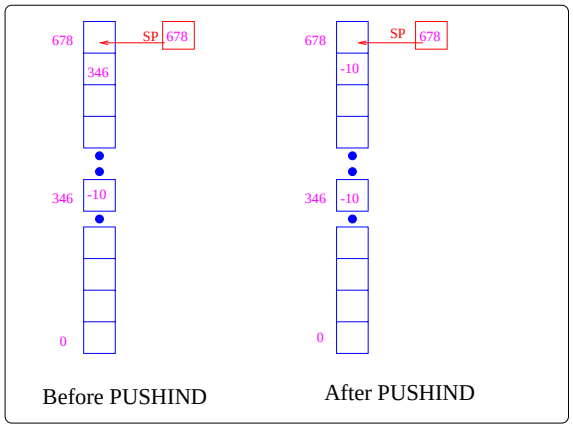


```
    public void translate(SymbolTable ST, SaM m) {
      //if we had a symbolic operand
      if (! dest.equals("")) {
        //translate
        operand = ST.lookup(dest);
        if (operand == -1)  m.println("Undefined name " + dest);
        //change dest so that we can print both dest and operand
        dest = dest + " //address: ";}}}
```



```
//This is the class for all instructions with one operand.
//The operand can be a word or an integer.
abstract class InstructionWithOperand extends Instruction{
    protected String dest = new String("");
    protected int operand;
    public String toString() {
      return  mnemonic() + " " + dest + operand;}
    public void readOperand(CS211In inf) {
      if(inf.peekAtKind() == inf.WORD)
        dest = new String(inf.getWord());
      else if(inf.peekAtKind() == inf.INTEGER)
        operand = inf.getInt();}
```

PUSHIND Command



PUSHIND : TOS contains a Stack address.

Pop that address, read contents of that address, and push.

SaM continued

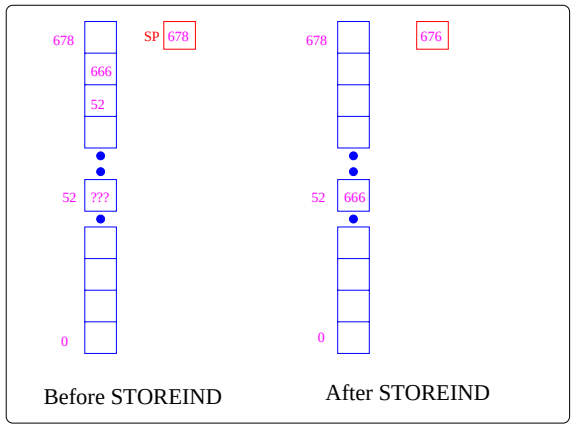
Let's give SaM a few more commands to make it possible to compile real programs to SaM commands.

So far, we have only pushed and popped values from stack.

In "real" programs, we often need to read and write stack locations other than topmost stack location.

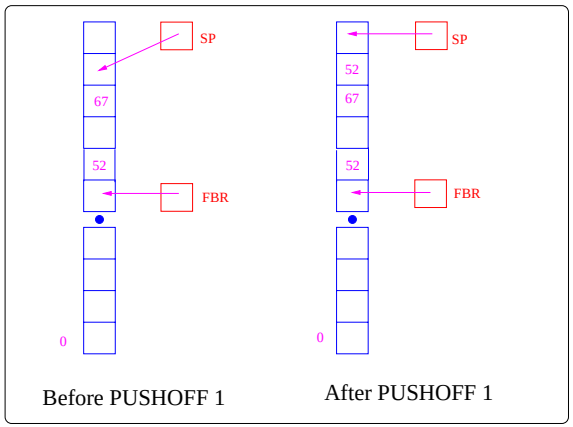
Intuitively, this corresponds to treating Stack area as if it is a conventional memory.

STOREIND Command



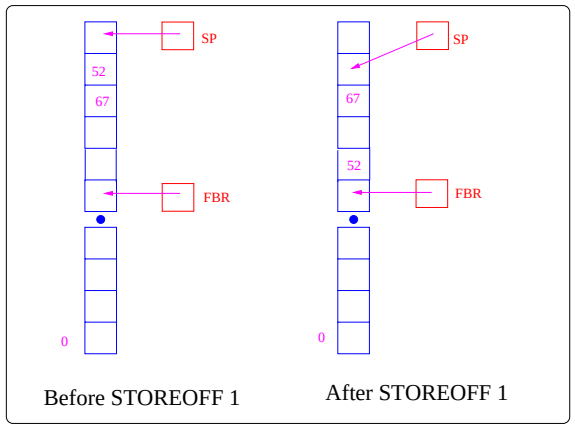
STOREIND : TOS contains a value. Below that is a stack address.
Pop these from the stack and
Write value into that address.

PUSHOFF Command



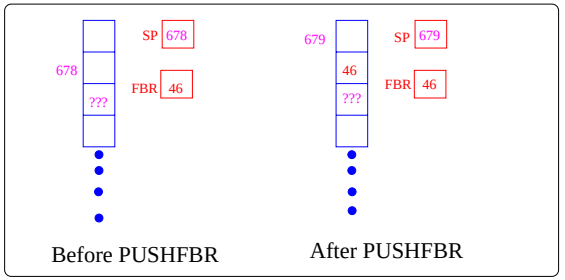
PUSHOFF x : let v be value stored in address (FBR + x)
push v on top of stack
FBR is unchanged

STOREOFF Command



STOREOFF x : Say TOS contains value v.
Pop it from the stack and
write v into address (FBR + x)
FBR is unchanged

PUSHFBR/POPFBR Commands



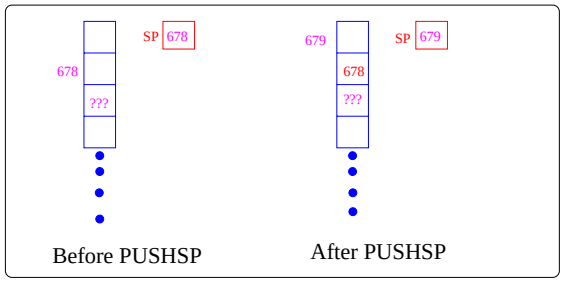
PUSHFBR: Push contents of FBR on stack.

Stack[SP] = FBR;
SP++;

POPFBR: Inverse operation

SP--;
FBR = Stack[SP];

PUSHSP/POPSP Commands



PUSHSP: Push contents of SP on stack.

Stack[SP] = SP;
SP++;

POPSP: Inverse operation

SP--;
SP = Stack[SP];

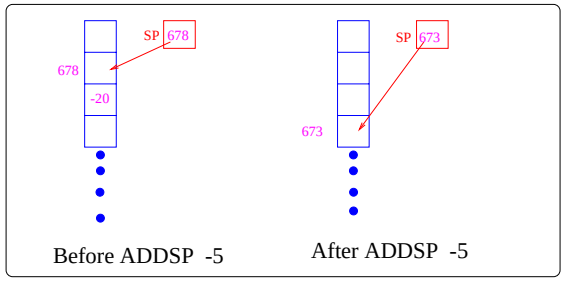
Two instructions for control transfer in method call/return:

```
JSR xxxx: Push address of next instruction on stack and jump to xxxx
JUMPIND: Pop an address from stack, and jump to that address

Example:
     ....
     JSR foo  //assume this instruction is stored in Program[32]
     ADD
     ....
foo: ADDSP 5  //assume foo is 98
     ....
     JUMPIND  //assume this instruction is stored at Program[200]
     ....
```

Sequence of PC values: 32, 98, 99, ..., 200, 33, 34,.....

ADDSP Command



ADDSP: takes one integer as explicit operand
Add that integer to SP

ADDSP n can be written in terms of other instructions:

PUSHSP
PUSHIMM n
ADD
POPSP

Convenient to have just one command.

Interfaces and inheritance are powerful tools for writing reusable programs.

Central ideas:

- Subtyping: permits you to define an extension of an existing type (inheritance,interfaces).
- Dynamic method binding: permits existing code to splice in the new type without modification.

Useful idea:

Inheritance (of implementation) Permits subtype to be defined *incrementally*

SaM Simulator brings all these ideas together.

Final Notes on SaM

Portability of SaM code:

- SaM standard: everyone agrees on SaM commands, labels etc.
- On all Macs/PCs/etc, provide SaM simulator written in your favorite language (BASIC, FORTRAN,...).
- SaM code is portable - write SaM code (or compile SaM code from say a Java program) and ship SaM code around the Internet.
- Java achieves portability the same way - Java Virtual Machine (JVM) is like SaM.
- JVM is much more complicated than SaM - need to support threads, multiple data types, need to worry about security etc.

Interfaces in SaM simulator:

- Interfaces document specifications.

  What methods must SaM support to permit instructions to execute? What methods do instructions support?

- Interfaces permit independent development of implementation code and client code.

- Interfaces permit objects of different types to be stored in one data structure. Example: instructions of different types in Program array