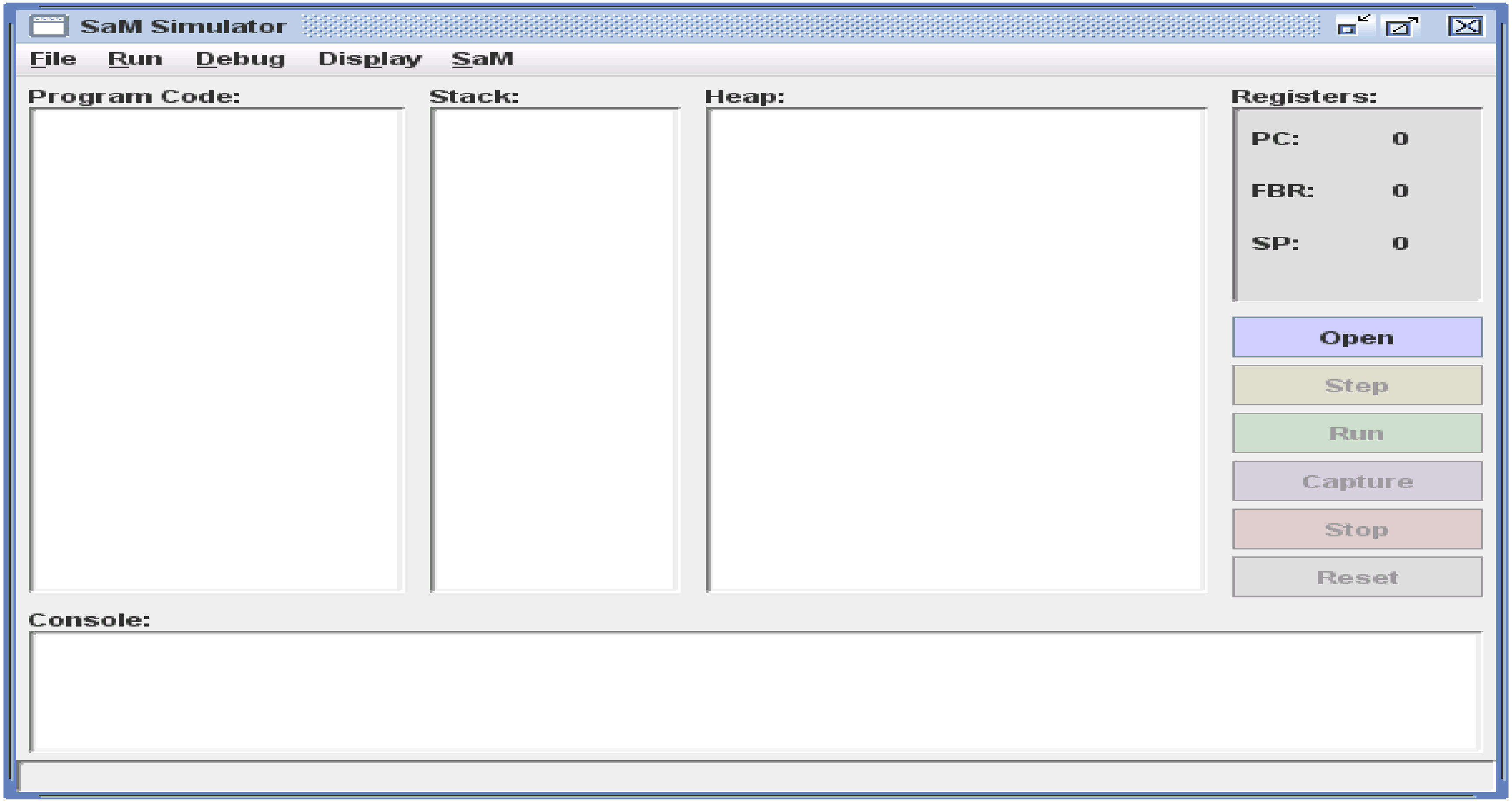
SaM I Am

What is SaM?

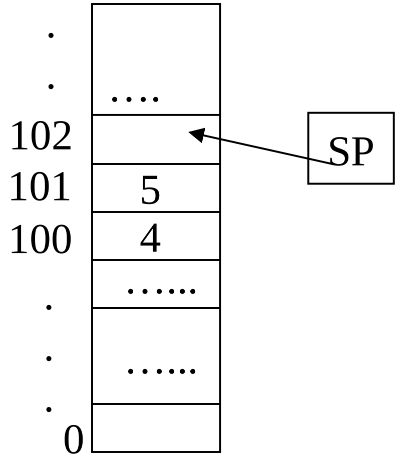
- SaM is a simple stack machine designed to introduce you to compilers in a few lectures
- SaM I: written by Dr. Keshav Pingali around 2000
  - modeled vaguely after JVM
- SaM II: complete reimplementation and major extensions by Ivan Gyurdiev and David Levitan (Cornell undergrads) around 2003
- Course homepage has
  - SaM jar file (interpreter)
  - SaM instruction set manual
  - SaM examples and source code

SaM Screenshot



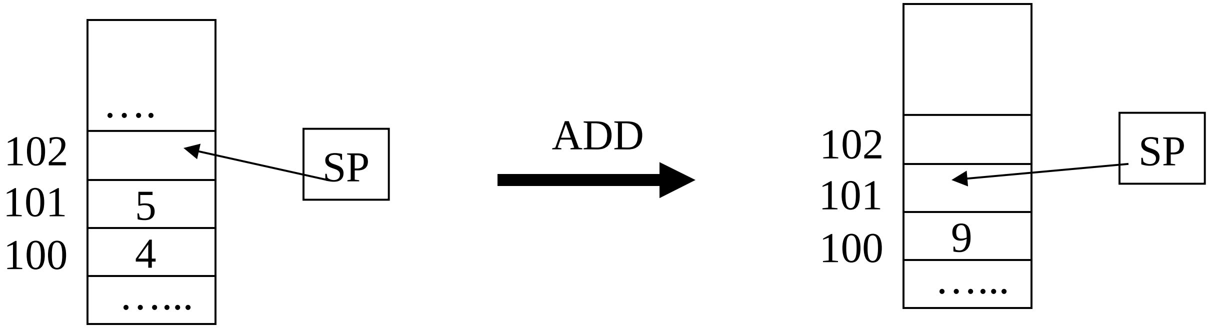
Stack machine

- All data is stored in stack (or heap)
  - no data registers although there might be control registers
- Stack also contains addresses
- Stack pointer (SP) points to the first free location on stack
- In SaM, stack addresses start at 0 and go up
- Int/float values take one stack location



Stack machine

- Stack machine is sometimes called a 0-address machine
  - arithmetic operations take operands from top of stack and push result(s) on stack



Program area in SaM

- Program area:
  - contains SaM code
  - one instruction per location
- Program Counter (PC):
  - address of instruction to be executed
  - initialized to 0 when SaM is booted up
- HALT:
  - Initialized to false (0) when SaM is booted up
  - Set to true (1) by the STOP command
  - Program execution terminates when HALT is set to true (1)

Program Execution

Command interpreter:

PC = 0;

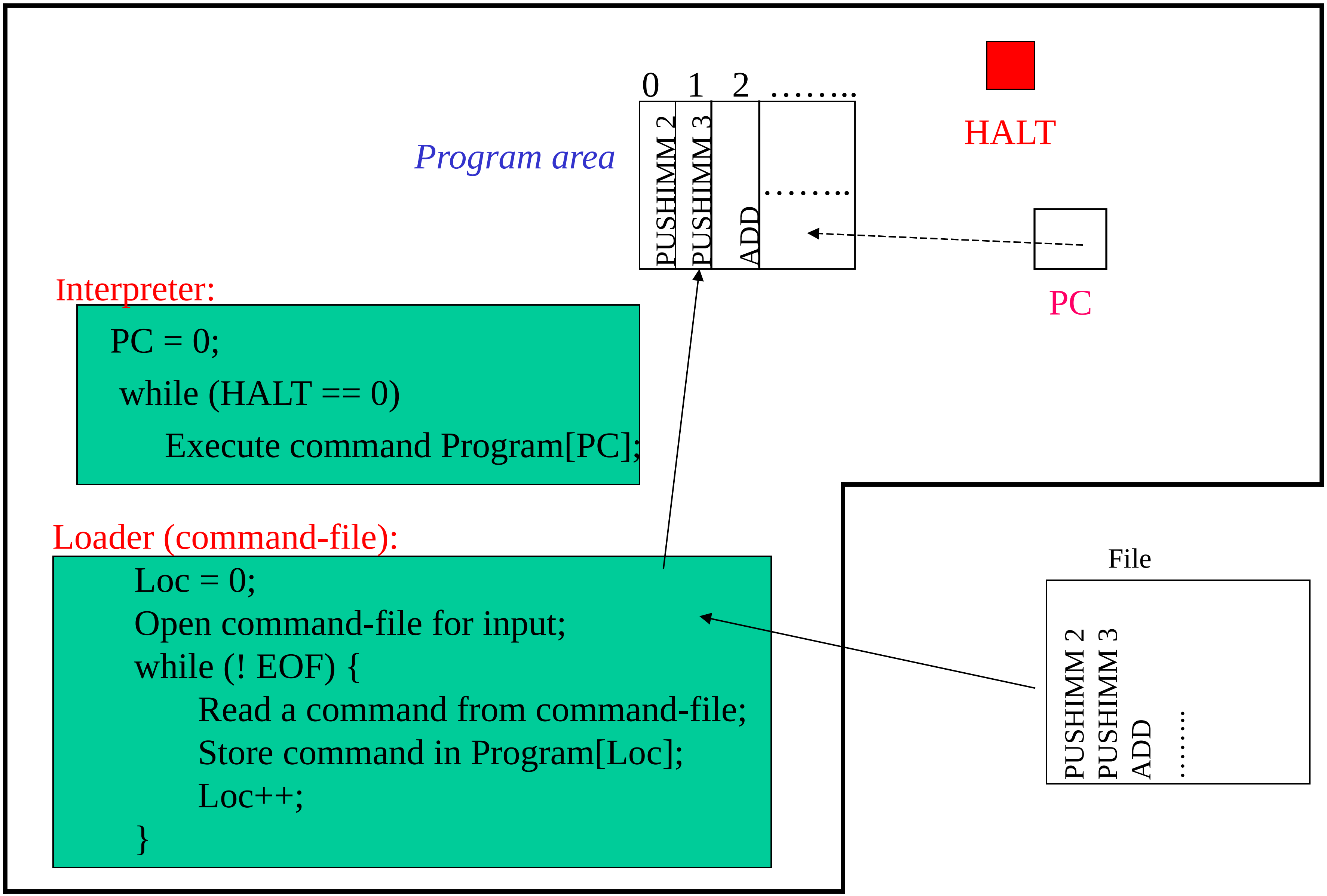
while (HALT == 0) *//STOP command sets HALT to 1*

Execute command Program[PC]; *//ADD etc increment PC*

Loader

- How do commands get into the Program area of SaM?
- Loader: a program that can open an input file of SaM commands, and read them into the Program area.

Interpreter and Loader



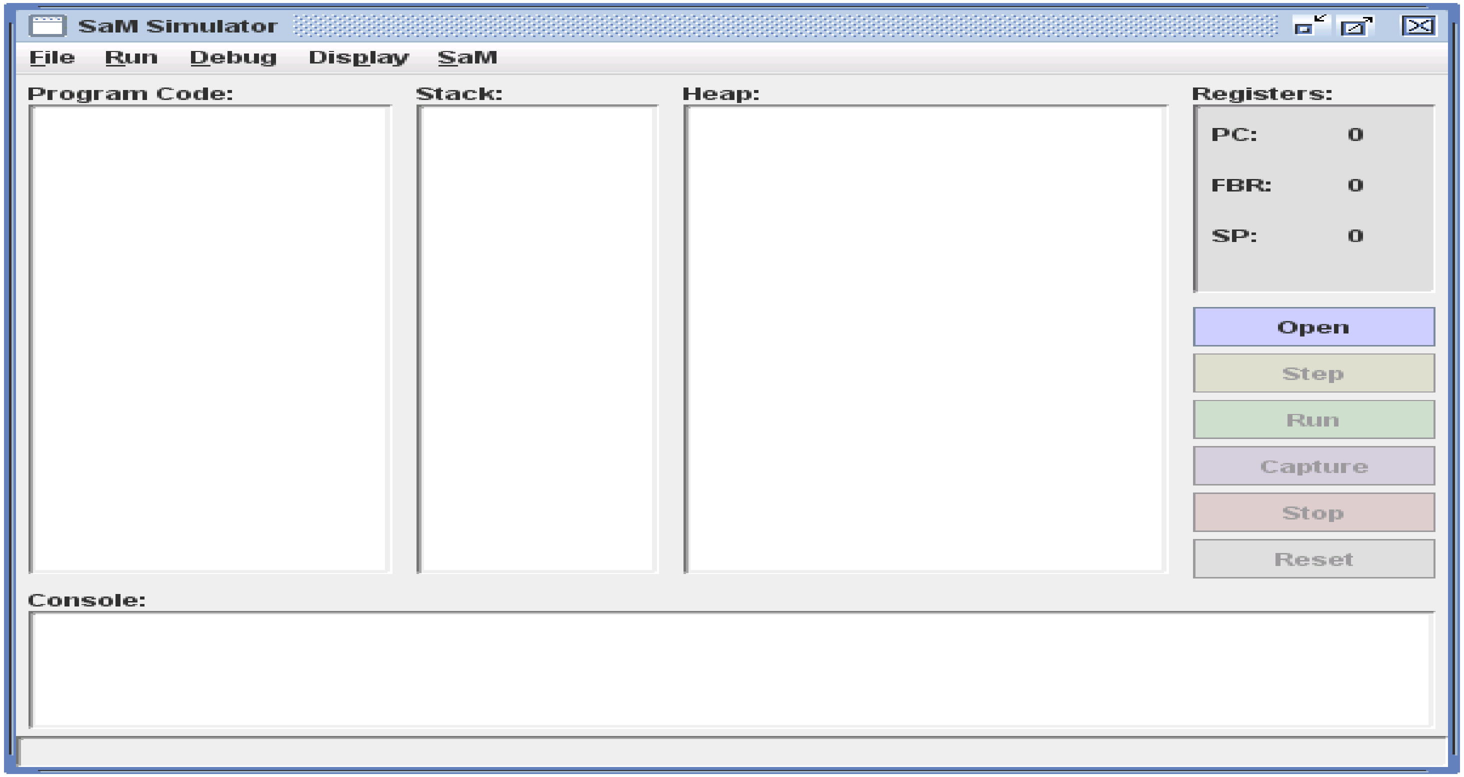
Labels

- SaM assembly instructions in program file can be given labels

```
foo: PUSHIMM 1
        .......
        JUMP foo
```

- SaM loader resolves labels and replaces jump targets with addresses

Other SaM areas



- FBR: Frame Base Register (see later)
- Heap: for dynamic storage allocation (malloc and free)
  SaM uses a version of Doug Lea's allocator

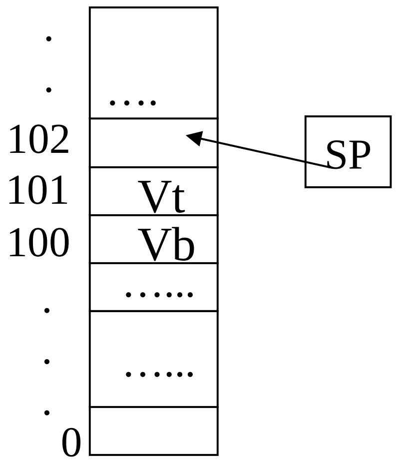
Some SaM commands

Classification of SaM commands

- Arithmetic/logic commands:
  - ADD,SUB,..
- Load/store commands:
  - PUSHIMM,PUSHIND,STOREIND,…
- Register←→Stack commands:
  - PUSHFBR,POPFBR, LINK,PUSHSP,…
- Control commands:
  - JUMP, JUMPC, JSR, JUMPIND,…

ALU commands

- ADD,SUB,TIMES,…: pop two values Vt and Vb from stack;
  - Vb op Vt
- DUP: duplicate top of stack (TOS)
- ISPOS:
  - Pop stack; let popped value be Vt
  - If Vt is positive, push true (1);otherwise push false (0)
- ISNEG: same as above but tests for negative value on top of stack
- ISNIL: same as above but tests for zero value on top of stack
- CMP: pop two values Vt and Vb from stack;
  - If (Vb < Vt) push 1
  - If (Vb = Vt) push 0
  - If (Vb > Vt) push -1



Pushing values on stack

- PUSHIMM c
  - "push immediate": value to be pushed is in the instruction itself
  - will push c on the stack

  (eg) PUSHIMM 4

  PUSHIMM -7

Example

SaM code to compute (2 + 3)

```
PUSHIMM 2
PUSHIMM 3
ADD
```

SaM code to compute (2 – 3) * (4 + 7)

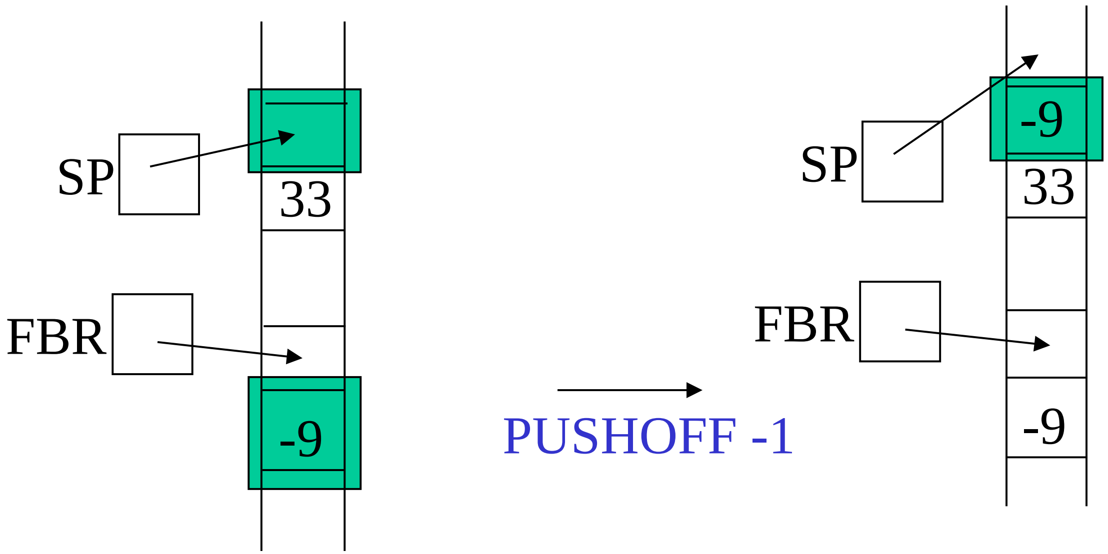
```
PUSHIMM 2
PUSHIMM 3
SUB
PUSHIMM 4
PUSHIMM 7
ADD
TIMES
```

← Compare with postfix notation (reverse Polish)

Load/store commands

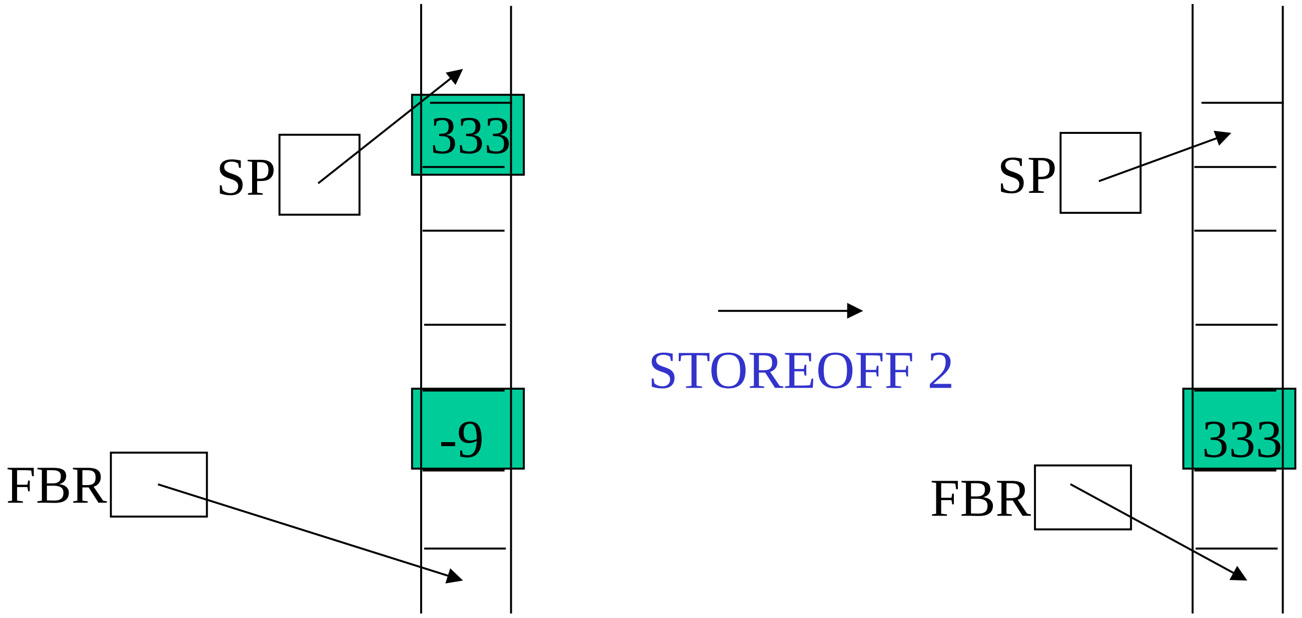
- SaM ALU commands operate with values on top of stack.
- What if values we want to compute with are somewhere inside the stack?
- Need to copy these values to top of stack, and store them back inside stack when we are done.
- Specifying address of location: two ways
  - address specified in command as some offset from FBR (offset mode)
  - address on top of stack (indirect mode)

- PUSHOFF n: push value contained in location Stack[FBR+n]
  - v = Stack[FBR + n]
  - Push v on Stack



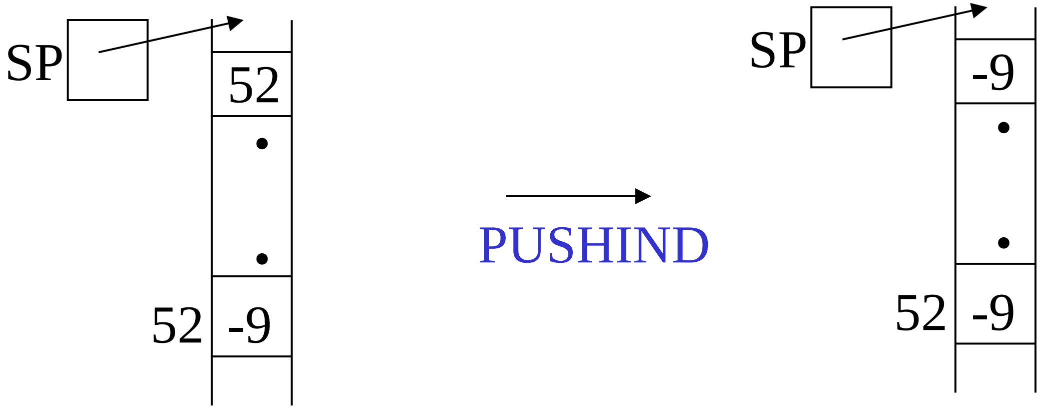
Stack[FBR –1] contains -9

- STOREOFF n: Pop TOS and write value into location Stack[FBR+n]
  - TOS has a value v
  - Pop it and write v into Stack[FBR + n].



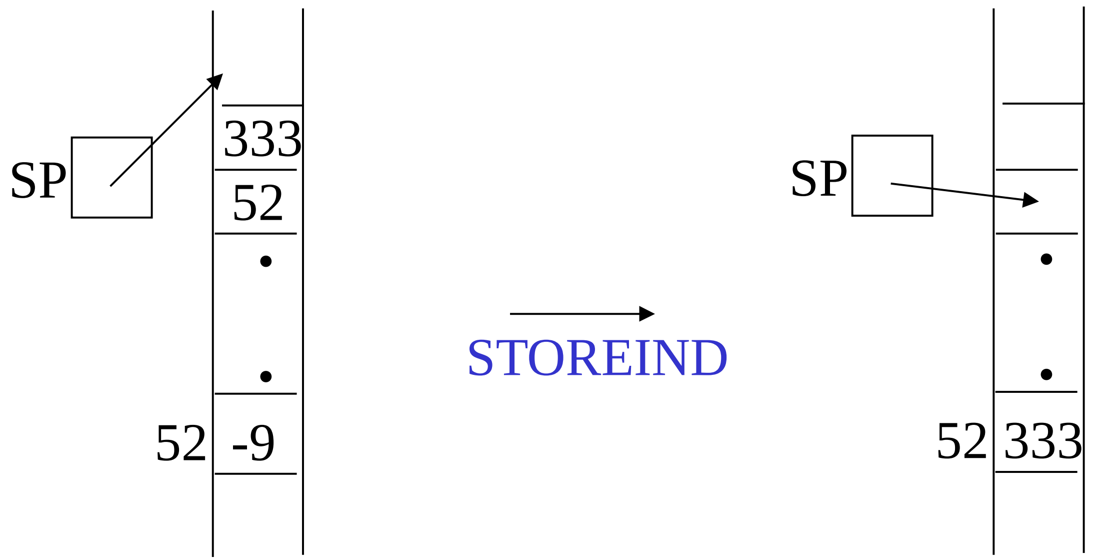
Store 333 into Stack[FBR+2]

- PUSHIND:
  - TOS has an address
  - Pop that address, read contents of that address and push contents on stack



TOS is 52
Contents of location 52 is -9

- STOREIND:
  - TOS has a value v; below it is address s
  - Pop both and write v into Stack[s].



TOS is value 333.
Below it is address 52.
Contents of location 52 is -9

Value 333 is written into location 52

Using PUSHOFF/STOREOFF

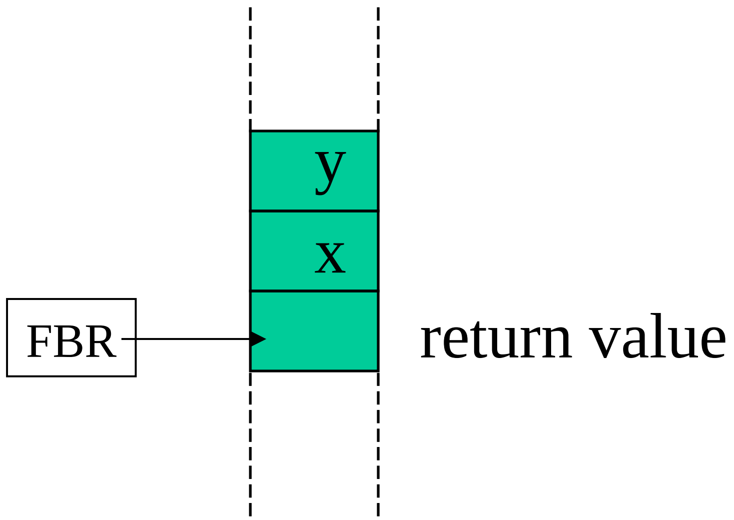
- Consider simple language SL
  - only one method called main
  - only assignment statements

```
main( ){
   int x,y;
   x = 5;
   y = (x + 6);
   return (x*y);
}
```

We need to assign stack locations for “x” and “y” and read/write from/to these locations to/from TOS

Stack frame

- Sequence of stack locations for holding local variables of procedure
  - “x” and “y”
- In addition, frame will have a location for return value
- Code for procedure must leave return value in return value slot
- Use offsets from FBR to address “x” and “y”
- Where should FBR point to
  - let’s make it point to “return value” slot
  - we’ll change this later

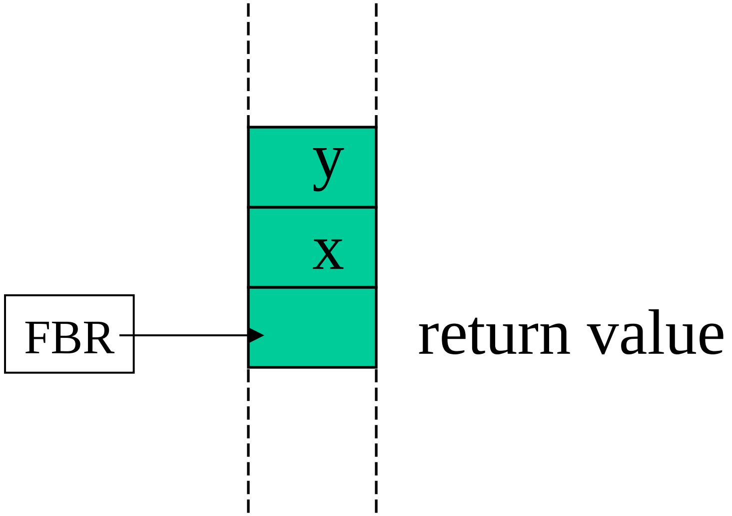


frame for main

SaM code (attempt 1)

```
main: PUSHIMM 0 //allocate space for return value
      PUSHIMM 0//allocate space for x
      PUSHIMM 0//allocate space for y
      //code for x = 5;
      PUSHIMM 5
      STOREOFF 1
      //code for y = (x+6);
      PUSHOFF 1
      PUSHIMM 6
      ADD
      STOREOFF 2
      //compute (x*y) and store in rv
      PUSHOFF 1
      PUSHOFF 2
      TIMES
      STOREOFF 0
      ADDSP -2   //pop x and y
      STOP
```

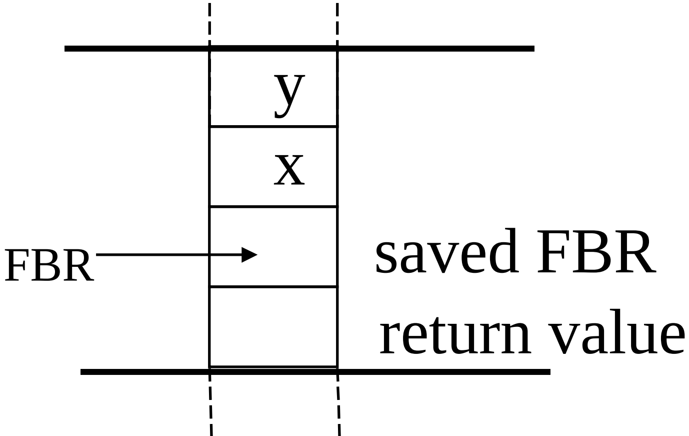
ADDSP 3



frame for main

Problem with SaM code

- How do we know FBR is pointing to the base of the frame when we start execution?
- Need commands to save FBR, set it to base of frame for execution, and restore FBR when method execution is done.
- Where do we save FBR?
  - Save it in a special location in the frame

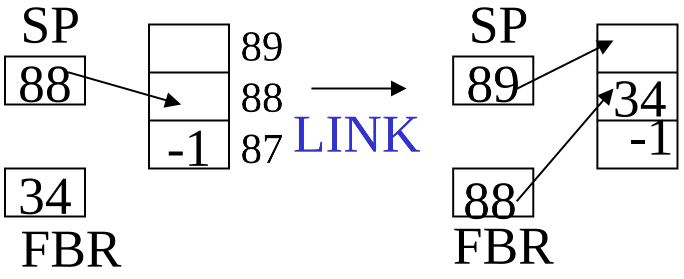


Register←→Stack Commands

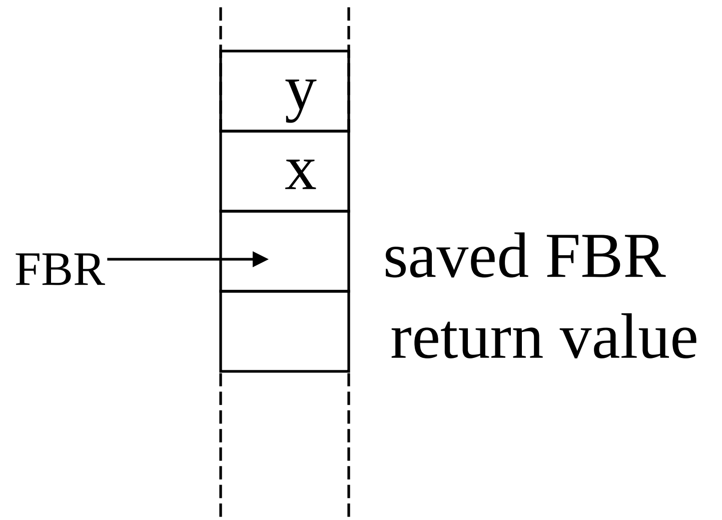
- Commands for moving contents of SP, FBR to stack, and vice versa.
- Used mainly in invoking/returning from methods
- Convenient to custom-craft some commands to make method invocation/return easier to implement.

FBR ←→ Stack commands

- PUSHFBR: push contents of FBR on stack
  - Stack[SP] = FBR;
  - SP++;
- POPFBR: inverse of PUSHFBR
  - SP--;
  - FBR = Stack[SP];
- LINK : convenient for method invocation
  - Similar to PUSHFBR but also updates FBR so it points to location where FBR was saved
  - Stack[SP] = FBR;
  - FBR = SP;
  - SP++;



```
main:   PUSHIMM 0//space for rv
        LINK//save and update FBR
        ADDSP 2//space for x and y
        //code for x = 5;
        PUSHIMM 5
        STOREOFF 1
        //code for y = (x+6);
        PUSHOFF 1
        PUSHIMM 6
        ADD
        STOREOFF 2
        //compute (x+y) and store in rv
        PUSHOFF 1
        PUSHOFF 2
        TIMES
        STOREOFF –1
        ADDSP –2//pop locals
        POPFBR//restore FBR
        STOP
```



frame for main

SP ←→ Stack commands

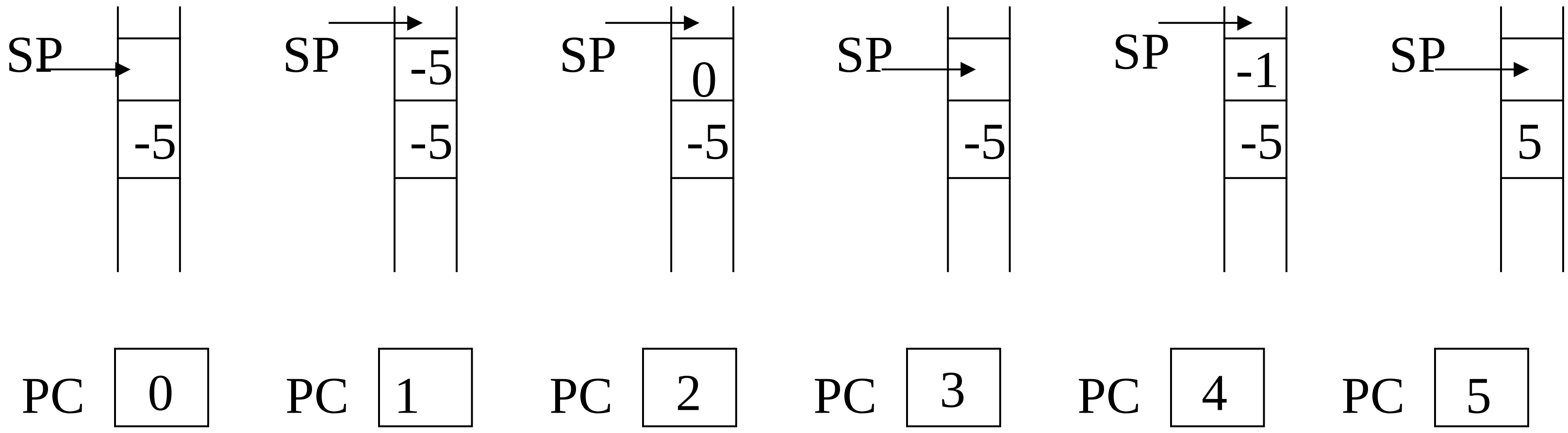
- PUSHSP: push value of SP on stack
  - Stack[SP] = SP;
  - SP++
- POPSP: inverse of POPSP
  - SP--;
  - SP = Stack[SP];
- ADDSP n: convenient for method invocation
  - SP = SP + n
  - For example, ADDSP –5 will subtract 5 from SP.
  - ADDSP n can be implemented as follows:
    - PUSHSP
    - PUSHIMM n
    - ADD
    - POPSP

Control Commands

- So far, command execution is sequential
  - execute command in Program[0]
  - execute command in Program[1]
  - …..
- For implementing conditionals and loops, we need the ability to
  - skip over some commands
  - execute some commands repeatedly
- In SaM, this is done using
  - JUMP: unconditional jump
  - JUMPC: conditional jump
- JUMP/JUMPC: like GOTO in PASCAL

- JUMP t: *//t is an integer*
  - Jump to command at Program[t] and execute commands from there on.
  - Implementation: PC ← t
- JUMPC t:
  - same as JUMP except that JUMP is taken only if the topmost value on stack is true; otherwise, execution continues with command after this one.
  - note: in either case, stack is popped.
  - Implementation:
    - pop top of stack (Vt);
    - if Vt is true, PC ← t else PC++

Example



Program to find absolute value of TOS:

0: DUP
1: ISPOS
2: JUMPC 5
3: PUSHIMM –1
4: TIMES
5: STOP

If jump is taken, sequence of PC values is 0,1,2,5

Symbolic Labels

- It is tedious to figure out the location/line numbers of commands that are jump targets (such as STOP in example).
- SaM loader allows you to specify jump targets using a symbolic label such as DONE in example above.
- When loading program, SaM figures out the addresses of all jump targets and replaces symbolic names with those addresses.

```
DUP
ISPOS
JUMPC   5
PUSHIMM –1
TIMES
STOP
```

```
       DUP
       ISPOS
       JUMPC   DONE
       PUSHIMM –1
       TIMES
DONE:  STOP
```

Using JUMPC for conditionals

- Translating if e then B1 else B2

```
    code for e
    JUMPC newLabel1
    code for B2
    JUMP newLabel2
newLabel1:
    code for B1
newLabel2:
    ……
```

PC ←→ Stack Commands

- Obvious solution: something like
  - PUSHPC: save PC on stack // not a SaM command
    - Stack[SP] = PC;
    - SP++;
- Better solution for method call/return:
  - JSR xxx: save value of PC + 1 on stack and jump to xxx
    - Stack[SP] = PC +1;
    - SP++;
    - PC = xxx;

  - JUMPIND: like "POPPC" (use for return from method call)
    - SP--;
    - PC = Stack[SP];

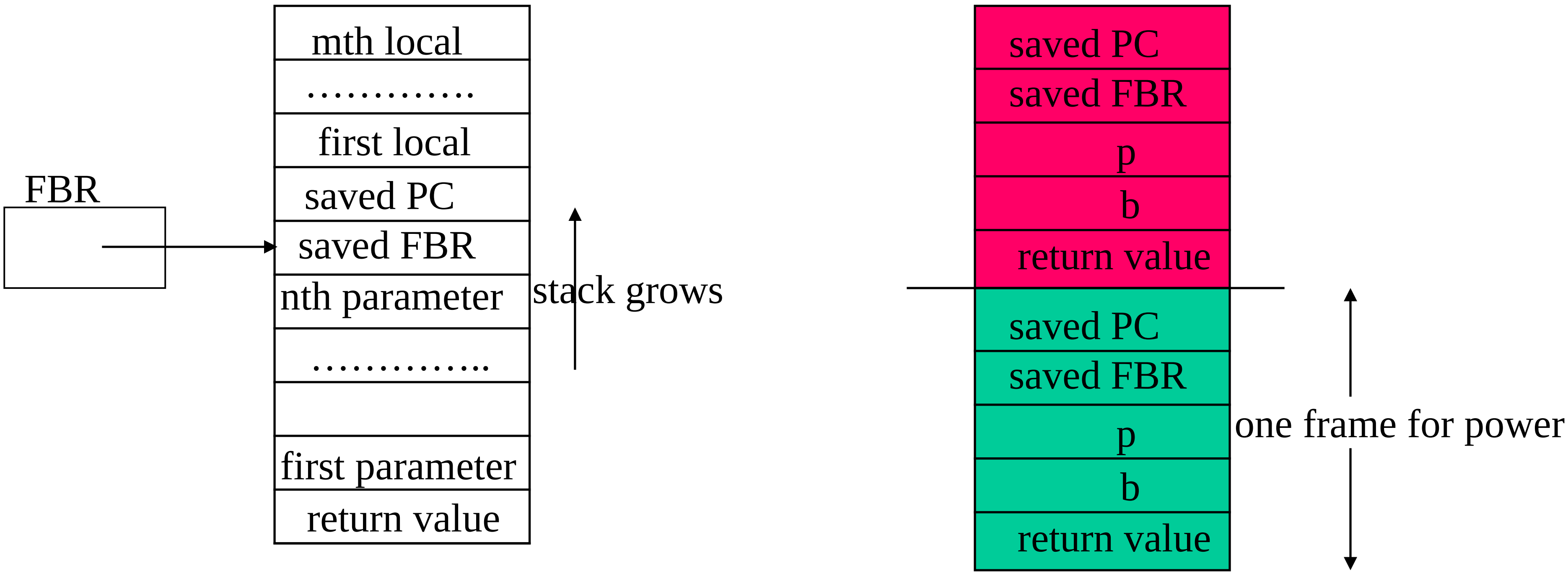
  - JSRIND: like JSR but address of method is on stack
    - temp = Stack[SP];
    - Stack[SP] = PC + 1;
    - SP++;
    - PC = temp;

Example

```
       …….
       JSR foo //suppose this command is in Program[32]
       ADD
       …..
foo: ADDSP 5 //suppose this command is in Program[98]
      …….
       JUMPIND//suppose this command is in Program[200]
       …..
```

Sequence of PC values: ….,32,98,99,…,200,33,34,…..,
assuming stack just before JUMPIND is executed is same as it was just after JSR was executed

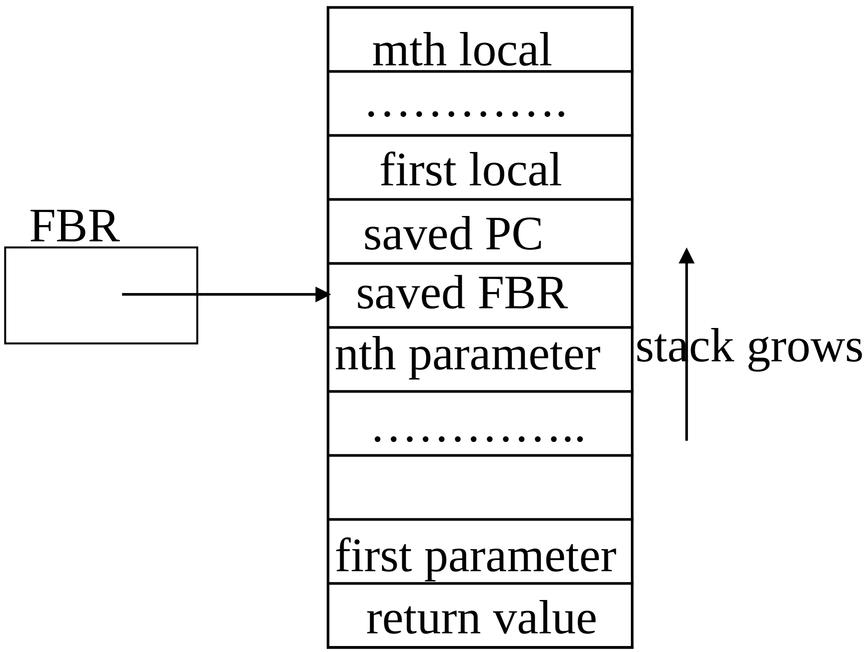
SaM stack frame for CS 380C




```
power(b,p){
        if (p = = 0) return 1;
        else return b*power(b,p-1);
}
```

Protocol for call/return

- Caller:
  - creates return value slot
  - evaluates parameters from first to last, leaving values on stack
  - LINK
  - JSR to callee
  - POPFBR //executed on return
  - pop parameters
- Callee:
  - create space for local variables
  - execute code of callee
- Return from callee:
  - Evaluate return value and write into rv slot
  - Pop off local variables
  - JUMPIND //return to caller



Writing SaM code

- Start by drawing stack frames for each method in your code.
- Write down the FBR offsets for each variable and return value slot for that method.
- Translate Bali code into SaM code in a compositional way. Think mechanically.

Recursive code generation

Construct	Code
integer	PUSHIMM xxx
x	PUSHOFF yy //yy is offset for x
(e1 + e2)	code for e1 code for e2 ADD
x = e;	code for e STOREOFF yy
{S1 S2 … Sn}	code for S1 code for S2 …. code of Sn

Recursive code generation(contd)

Construct	Code
if e then B1 else B2	code for e JUMPC newLabel1 code for B2 JUMP newLabel2 newLabel1: code for B1 newLabel2:
while e do B;	newLabel1: code for e ISNIL JUMPC newLabel2 code for B JUMP newLabel1 newLabel2:

Better code:

```
JUMP newLabel1
newLabel2:
    code for B
newLabel1:
    code for e
    JUMPC newLabel2
```

Better code

Recursive code generation(contd)

Construct	Code
f(e1,e2,…en)	PUSHIMM 0//return value slot Code for e1 … Code for en LINK//save FBR and update it JSR f POPFBR//restore FBR ADDSP –n//pop parameters

Recursive code generation(contd)

Construct	Code

f(p1,p2,…,pn){
int x,…,z;//locals
B}
	ADDSP c // c is number of locals
	code for B
	fEnd:
	STOREOFF r//r is offset of rv slot
	ADDSP –c//pop locals off
	JUMPIND//return to callee

return e;	code for e JUMP fEnd//go to end of method

OS code for SaM

- On a real machine
  - OS would transfer control to **main** procedure
  - control returns to OS when **main** terminates
- In SaM, it is convenient to begin execution with code that sets up stack frame for main and calls main
  - this allows us to treat main like any other procedure

```
//OS code to set up call to main

PUSHIMM 0 //rv slot for main
LINK  //save FBR
JSR main //call main
POPFBR
STOP
```

Symbol tables

- When generating code for a procedure, it is convenient to have a map from variable names to frame offsets
- This is called a "symbol table"
- For now, we will have
  - one symbol table per procedure
  - each table is a map from variable names to offsets
- Symbol tables will also contain information like types from type declarations (see later)

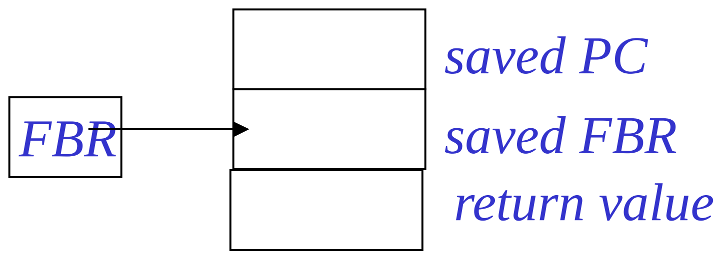
Example

Let us write a program to compute absolute value of an integer.

Bali:

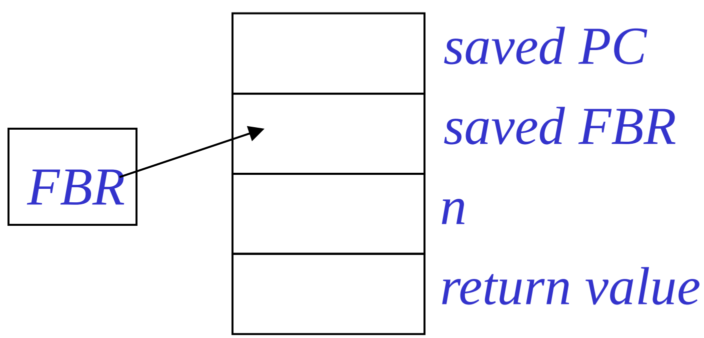
```
main() {
    return abs(-5);
}

abs (n) {
    if ((n > 0)) return n;
    else return (n*-1);
}
```



Symbol Table

Variable	Offset
rv	-1



Symbol Table

Variable	Offset
n	-1
rv	-2

```
main() {

 return abs(-5);
  }
```

```
main:
     ADDSP 0 // 0 is number of locals
      code for  "return abs(-5)"
mainEnd:
     STOREOFF -1//-1 is offset of rv slot
     ADDSP -0  //pop locals off
     JUMPIND//return to callee
```

(1)

```
main:
     ADDSP 0 // 0 is number of locals
      code for  "abs(-5)"
      JUMP mainEnd
mainEnd:
      STOREOFF -1//-1 is offset of rv
     ADDSP -0  //pop locals off
     JUMPIND//return to callee
```

(2)

```
main:
     ADDSP 0 // 0 is number of locals
      PUSHIMM 0
      code for "-5"
      LINK
      JSR abs
      POPFBR
      ADDSP -1
      JUMP mainEnd
mainEnd:
     STOREOFF -1//-1 is offset of rv slot
     ADDSP -0  //pop locals off
     JUMPIND//return to callee
```

(3)

```
main:
     ADDSP 0 // 0 is number of locals
      PUSHIMM 0
      PUSHIMM -5
      LINK
      JSR abs
      POPFBR
      ADDSP -1
      JUMP mainEnd
mainEnd:
     STOREOFF -1//-1 is offset of rv slot
     ADDSP -0  //pop locals off
     JUMPIND//return to callee
```

(4)

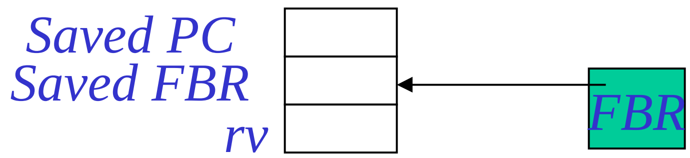
Complete code

```
//OS code to set up call to main
PUSHIMM 0 //rv slot for main
LINK  //save FBR
JSR main //call main
POPFBR
STOP
main:
    //set up call to abs
    PUSHIMM 0//return value slot for abs
    PUSHIMM –5//parameter to abs
    LINK//save FBR and update FBR
    JSR abs//call abs
    POPFBR //restore FBR
    ADDSP –1//pop off parameter
    //from code for return
    JUMP mainEnd
mainEnd:
    STOREOFF -1//store result of call
    JUMPIND

abs:  PUSHOFF –1//get n
      ISPOS //is it positive
      JUMPC pos//if so, jump to pos
      PUSHOFF –1//get n
      PUSHIMM –1//push -1
      TIMES//compute -n
      JUMP absEnd//go to end
pos:  PUSHOFF –1//get n
      JUMP absEnd
absEnd:
      STOREOFF –2//store into r.v.
      JUMPIND//return
```

Factorial

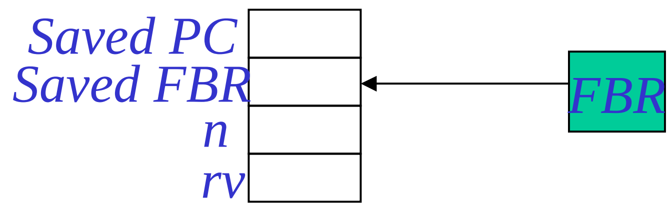
```
main() {
    return fact(5);
}
```



Symbol Table

Variable	Offset
rv	-1

```
fact(n) {
  if ((n ==0) return 1;
  else return (n*fact(n-1));
}
```



Symbol Table

Variable	Offset
n	-1
rv	-2

```
//OS code to set up call to main
PUSHIMM 0 //rv slot for main
LINK  //save FBR
JSR main //call main
POPFBR
STOP

main:
    //code for call to fact(10)
     PUSHIMM 0
     PUSHIMM 10
     LINK
     JSR fact
     POPFBR
     ADDSP -1
     //from code for return
     JUMP mainEnd
     //from code for function def
mainEnd:
     STOREOFF -1
     JUMPIND
```

```
fact:
     PUSHOFF -1 //get n
     PUSHIMM 0
     EQUAL
     JUMPC zer
     PUSHOFF -1  //get n
     PUSHIMM 0 // fact(n-1)
     PUSHOFF -1
     PUSHIMM 1
     SUB
     LINK
     JSR fact
     POPFBR
     ADDSP -1
     TIMES          //n*fact(n-1)
     JUMP factEnd
zer: PUSHIMM 1
      JUMP factEnd
 factEnd:
     STOREOFF -2
     JUMPIND
```

Running SaM code

- Download the SaM interpreter and run these examples.
- Step through each command and see how the computations are done.
- Write a method with some local variables, generate code by hand for it, and run it.

Introduction to Parsing

- Please read the lecture notes titled "Parsing" on the course website before next class (prerequisite):

http://www.cs.utexas.edu/%7Epingali/CS380C/2016/lectures/parsingIntro.pdf