CS 211	Computers and Programming Fall 1997	Handout 10
	SaM: A Stack Machine	November 5th, 1998

1 Introduction

SaM is a simple stack machine. It has only one data type: integer. Boolean values are simulated using integers — 0 stands for false and 1 stands for true. There are no floating point numbers, characters etc. although these are easy to add.

SaM has two areas of memory: *program* and *stack*. The program area is where the user program is loaded prior to execution. The stack is where data is stored during the execution of the program. There are four registers: PC, SP, FBR and HALT. The PC register is the *Program Counter* — it contains the address of the instruction that is currently being executed. The SP register is the *Stack Pointer* — it contains the address of the first free location on the stack. FBR is the *Frame Base Register*. Its role will become clear later; for now, assume it holds 0. In addition, there is a HALT register. SaM executes instructions while HALT has a 0 value in it. An instruction that wishes to terminate program execution does so by storing 1 into the HALT register.

2 SaM Instructions

SaM instructions are divided into the following classes:

ALU instructions: These instructions perform arithmetic and logical operations like addition, subtraction etc. on integers. The operands are popped from the stack and the result is pushed back to the stack. After the execution of an ALU instruction, control is always transferred to the instruction that follows that ALU instruction in the program.

Stack manipulation instructions: These instructions are used to copy a value from one location in the stack to another.

Register save/restore instructions: These instructions permit the values of the registers to be pushed and popped from the stack.

Control instructions : These instructions are used to implement conditional and unconditional transfer of control in the program.

In the descriptions of the instructions given below, we assume V_{top} and V_{below} are the topmost element of the stack and the element below it respectively *before*

the instruction is executed. If a command needs these operands, they are popped before any results are pushed on the stack. Some commands may need only V_{top}, in which case only V_{top} is popped.

2.1 ALU instructions

- ADD // Push $V_{below} + V_{top}$
- SUB // Push $V_{below} - V_{top}$
- TIMES // Push $V_{below} * V_{top}$
- DIV // Push V_{below}/V_{top}
- EQUAL // Push $V_{below} == V_{top}$
- GREATER // Push $V_{below} > V_{top}$
- LESS // Push $V_{below} < V_{top}$
- AND // Push $V_{below} \&\& V_{top}$
- OR // Push $V_{below} || V_{top}$
- NOT // Push the negation of V_{top}

2.2 Stack manipulation instructions

- PUSHIMM c // c is an integer; push the value c
- DUP //Duplicate top element:Stack[SP] = Stack[SP-1];SP++
- SWAP //Exchange the top two elements on stack
- PUSHIND //push $Stack[V_{top}]$
- STOREIND $//Stack[V_{below}] \leftarrow V_{top}$
- PUSHOFF o //push $Stack[o + FBR]$
- STOREOFF o $//Stack[o + FBR] \leftarrow V_{top}$

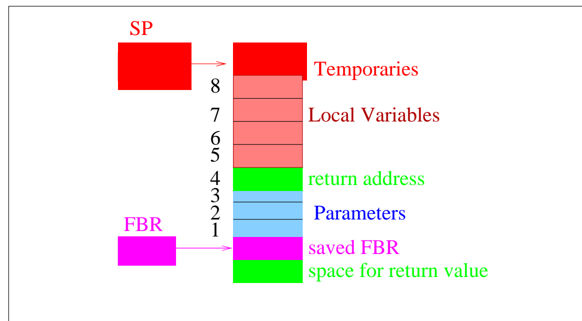
2.3 Register save/restore instructions

- PUSHSP $//Stack[SP] \leftarrow SP; SP++$
- POPSP $//SP \leftarrow V_{top}$
- ADDSP c //c is integer;$SP \leftarrow SP + c$
- PUSHFBR //push FBR
- POPFBR $//FBR \leftarrow V_{top}$

2.4 Control operations

Assume that t is a non-negative integer.

- JUMP t // $PC \leftarrow t$
- JUMPC t // if V_{top} is true, $PC \leftarrow t$ else $PC++$
- JUMPIND // $PC \leftarrow V_{top}$
- JSR t // push (PC+1), $PC \leftarrow t$
- JSRIND // push (PC+1);$PC \leftarrow V_{top}$
- STOP // $HALT \leftarrow 1$



- frame contains all information required to execute and return from a method call
- FBR: Frame Base Register (all addressing of locals and parameters is relative to FBR)
- one slot for each parameter (values computed and pushed by caller)
- one slot for return address (where do we return to when method call is done?)
- one slot for each local variable (allocated by callee)
- temporaries are pushed and popped on top of frame for intermediate computations

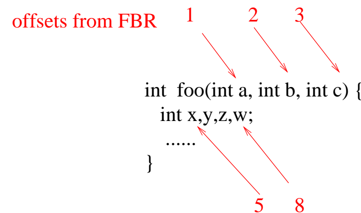


Figure 1: A Frame for a Method Invocation

3 Method invocation code sequence

The method invocation code sequence is not a part of the definition of SaM but we include it here to show how methods can be implemented in SaM. Each method invocation gets a *frame* that contains all the parameters, local variables and book-keeping information required for that invocation. If a method A invokes a method B (or itself), the frame for the invocation of B is pushed on top of the frame for the invocation of A. A frame is not pushed at one shot - rather it is built collaboratively by the caller and the callee.

The Frame Base Register always points to the bottom of the topmost frame on the stack. The location in the frame that the FBR points to contains the contents of the FBR before the topmost frame was constructed (so intuitively, it is where the old FBR contents are saved). When the method invocation returns, the value of the FBR is restored from that location.

Figure 1 shows a picture of a frame for an invocation of a method that has 3 parameters and 4 local variables.

Here is the code sequence that sets up the frame. Assume that the method

call is $f(e_1, e_2)$.

```
Invocation/cleanup sequence: executed by caller

PUSHIMM 0 //leave slot for return value, initialized to 0
          //we could also have made the previous instruction ADDSP 1
PUSHFBR   //save FBR
code for evaluating e1
code for evaluating e2
PUSHSP          //setup FBR as follows
PUSHIMM 3       //number of paramters of parameters + 1
SUB
POPFBR          //FBR set correctly for recursive call
JSR f   //push return address and jump to address f
        //the following code is executed on return
ADDSP -2 //pop parameters
POPFBR  //restore FBR

Invocation/return sequence: executed by callee
ADDSP n   //n is actual number of local variables
....
ADDSP -n   //pop off local vars
JUMPIND   //pop return address into PC
```

4 Sample programs

Here is a simple SaM program that does some silly calculation:

```
PUSHIMM 5
PUSHIMM 4
TIMES
PUSHIMM 3
TIMES
PUSHIMM 2
TIMES
STOP
```

Here is a much longer program that computes factorial. The code is an edited version of the code produced by our compiler from the following Bali program:

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

fact(n)
 if (n > 2) return fact(n - 1)*n;
 else return n;
```

```
PUSHIMM 0  //slot for return value from program
PUSHFBR   //save FBR
PUSHIMM 1 //set FBR to 1
POPFBR    //
JSR main  //now call main
POPFBR    //restore FBR
STOP     //terminate program
main:
ADDSP 0   //leave space for local variables (none)
          //invocation fact(5)
PUSHIMM 0 //space for return value
PUSHFBR   //save FBR
PUSHIMM 5 //push parameter
PUSHSP    //set FBR
PUSHIMM 2
SUB
POPFBR
JSR fact   //call fact
ADDSP -1  //get rid of parameter
POPFBR     //restore FBR
           //set up return avlue
STOREOFF -1 //save return value
JUMPIND     //return from fact

fact:
ADDSP 0     //leave space for local variables (none)
PUSHOFF 1   //push value of parameter (n)
PUSHIMM 2   //push 2
GREATER     // n > 2 ??
JUMPC factLabel0 //if so, jump to true side of conditional
                 //false side of conditional
PUSHOFF 1        //store n in return value slot
STOREOFF -1
JUMPIND          //return

factLabel0:   //true side of conditional
              //set up call to fact(n-1)
PUSHIMM 0     //space for return value
PUSHFBR       //save FBR
PUSHOFF 1     //compute n-1 as follows
PUSHIMM 1
SUB           //TOS now contains n-1
PUSHSP        //set FBR as follows
PUSHIMM 2
SUB
POPFBR
```

```
JSR fact       //call fact
ADDSP -1      //pop parameter
POPFBR         //restore FBR
PUSHOFF 1      //push n
TIMES          //fact(n-1) * n
STOREOFF -1    //save in return value slot
JUMPIND        //return
```

A SaM simulator and a GUI have been put on the home page. Download the code and play with it to get a feel for SaM.