Functions

CS212
Fall 2006

Announcements

Enhanced Bali--

```
main() {
   { int x, int y; }
   { x = 10; y = 20;
     return add(x,y); }
}

add(int p1, int p2) {
   { }
   { return (p1+p2); }
}
```

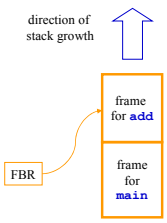

Samcode

```
main:    PUSHIMM 0    // return slot for main and program
         ADDSP 2      // allocate 2 local vars
         PUSHIMM 10   // push 10
         STOREOFF 1   // store val 10 in address 1 (x<-10)
         PUSHIMM 20   // push 20
         STOREOFF 2   // store val 20 in address 1 (y<-20)
         PUSHIMM 0    // allocate return value for add
         PUSHOFF 1    // push value of x for p1
         PUSHOFF 2    // push value of y for p2
         LINK         // save old FBR (0) and update FBR (6)
         JSR add      // jump to function "add"
         UNLINK       // restore FBR after returning from "add"
         ADDSP -2     // pop parameters (p1, p2) of "add"
         JUMP mainEnd // prepare to end main
mainEnd: STOREOFF 0   // store program's rv
         ADDSP -2     // remove x,y
         STOP         // end program
add:     PUSHOFF -2   // get p1
         PUSHOFF -1   // get p2
         ADD          // push x+y
         JUMP addEnd  // begin to end add
addEnd:  STOREOFF -3  // store x+y as rv
         RST          // return to main
```


Frames

Some terms:

- The (Call) ***Stack***
- ***Frame***
- ***FBR*** (more later)



Samcode and Frame Structure

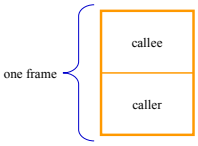
- Compiling process:
  - Bali code→Samcode
  - Stored in array
    - Gives addresses
    - See SaM session
  - You can provide addresses with ***labels***

Program Code:

```
0: PUSHIMM 0  (<= main )
1: ADDSP 2
2: PUSHIMM 10
3: STOREOFF 1
4: PUSHIMM 20
5: STOREOFF 2
6: PUSHIMM 0
7: PUSHOFF 1
8: PUSHOFF 2
9: LINK
10: JSR add
11: UNLINK
12: ADDSP -2
13: JUMP mainEnd
14: STOREOFF 0  (<= mainEnd )
15: ADDSP -2
16: STOP
17: PUSHOFF -2  (<= add )
18: PUSHOFF -1
19: ADD
20: JUMP addEnd
21: STOREOFF -3  (<= addEnd )
22: RST
```

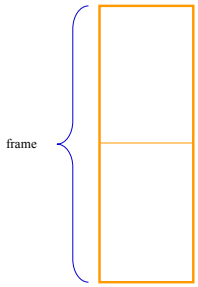
Creating a Frame

- Creation of a frame is shared:
  - ***caller*** (calling code)
  - ***callee*** (function's code)



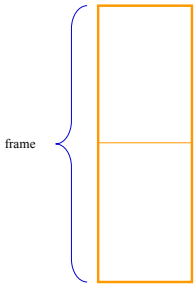
Caller's Responsibilities

- Push space for rv
- Push arguments
- Create new frame
- Jump to callee
- Demolish callee's frame (when done)



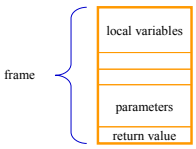
Callee's Responsibilities

- Push space for local variables
- Continue with callee's code
- Deallocate memory allocated during callee
- Return control back to caller



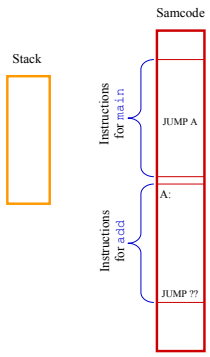
Variables/Parameters

- Return variable:
  - Must remain after frame demolished by caller
  - Caller uses in its code
- Local variables:
  - In function scope
  - So, keep function's local variables in its frame
- Parameters:
  - Also local variables
  - So, in frame, too



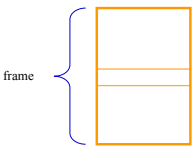
Jumping

- How to connect caller/callee:
  - caller calls callee
  - callee returns to caller
  - both destroy frame
- Solution:
  - Addresses and labels



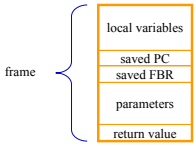
Frame-Based Register (FBR)

- FBR:
  - Keeps track of frames
  - Points to old FBR
  - First FBR at 0
- Caller:
  - Save old FBR
  - Set new FBR
- Samcode: **`LINK`**
  - Push currently value of FBR onto Stack
  - Set FBR to SP-1



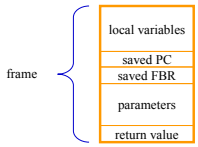
Program Counter (PC)

- PC:
  - Keeps track of current Samcode instruction
  - Not location in The Stack!
- Caller:
  - Save *next* Samcode instruction address
  - Jump to first Samcode instruction of callee
  - When done, next Samcode instruction ready to go!
- Samcode:
  `JSR address`
  - ***Jump to SubRoutine***
  - Push PC+1 onto stack
  - Set PC to **`address`**



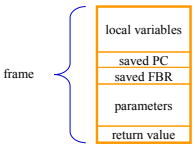
Clearing Frame

- Clear frame:
  - *callee* and *caller* share in destruction
  - rv remains on Stack
- Callee:
  - Clear local variables
  - Return to caller (reset PC) with **`RST`** (***Return from SubrouTine***):
    - Set PC to value on top of stack
    - SP– (remove saved PC)
- Caller:
  - Restore FBR with **`UNLINK`**:
    - Set value of FBR to value on top of stack
    - SP–
  - Clear params



Access to Frame's Data

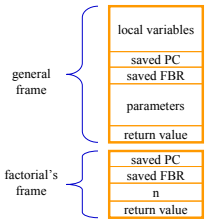
- Why bother with FBR?
- Data stored in the frame are accessed via offset from the FBR
  - Let **`p`** be the number of parameters



- The first local variable
  - **`STOREOFF 2`**
- The second local variable
  - **`STOREOFF 3`**
- The first parameter
  - **`STOREOFF -p`**
- The second parameter
  - **`STOREOFF -p + 1`**
- The return value
  - **`STOREOFF -p - 1`**

An Example (see also slide 3)

```
fact (int n) { (){
   if ((n < 2)) return 1;
   else return (n * fact(n-1));
   } }
```




```
factorial: PUSHOFF -1
PUSHIMM 2
LESS
JUMPC true
JUMP false
true: PUSHIMM 1
STOREOFF -2        // Store return value
RST                // Return
false: PUSHOFF -1
ADDSP 1            // Space for return value
PUSHOFF -1
PUSHIMM 1
SUB                // Argument is now on stack
LINK               // Create new stack frame
JSR factorial      // Call the function
UNLINK             // Restore FBR
ADDSP -1           // Clear the argument
TIMES
STOREOFF -2        // Store return value
RST                // Return
```


Example Calling Code

```
program:
ADDSP 1            // Space for return value
PUSHIMM 5          // The argument
LINK               // Create new stack frame
JSR factorial      // Call the function
UNLINK             // Restore FBR
ADDSP -1           // Clear the argument
STOP
```

We need this "calling code" to help create factorial's initial frame

Code Pattern for Caller

```
func(exp1,exp2,exp3)

ADDSP 1            // Return value
code for exp1      // Push args
code for exp2
code for exp3
LINK               // Create new frame
JSR func           // Call func
UNLINK             // Restore FBR
ADDSP -3           // Remove args
```

- Creating frame:
  - Push space for rv
  - Push params
  - Create new frame (**`LINK`**)
  - **`JSR`** to callee (push PC+1 and jump to callee)
- Destroying frame:
  - Restore the FBR (**`UNLINK`**)
  - Clear the params from stack

Code Pattern for Callee

```
func(t1 e1, t2 e2, t3 e3) {
   {localvars}
   {statements; return exp}
}

ADDSP v          // Space for v
                 //    local variables
code for statements
code for exp     // Compute return value
JUMP endfunc     // Jump to clean-up
endfunc:
STOREOFF -4      // Store return value
ADDSP -v         // Clear local variables
RST              // Return to caller
```

- Creating frame:
  - Push space for local variables
- Destroying frame:
  - Clear local variables from Stack
  - **RST** to caller (clear the saved PC and jump back to calling code)

What about the `main` Function?

- Two choices:
  - **main** and Program share rv
  - Program calls **main**

```
program:
ADDSP 1          // Return value for main
LINK             // Create new frame
JSR main         // Call main
UNLINK           // Restore FBR

main:
ADDSP v          // Space for main's
                 // local variables
code for statements
code for exp     // Compute return value
JUMP endmain     // Jump to clean-up
endmain:
STOREOFF -1      // Store return value
ADDSP -v         // Clear localvariables
RST              // Return to caller
```