Week 7
Implementing Functions

Paul Chew
CS 212 – Spring 2004

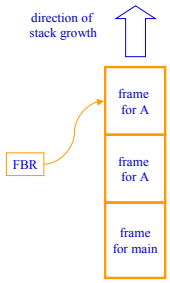
Announcements

- Sections this week (primary purpose: answer questions about Part 2B)
  - W evening, Mar 10
  - M afternoon, Mar 15
  - M evening, Mar 15
- Some of the "helper" files for Part 2B have been changed due to small errors
  - The new versions are on the Web
- The CMS for CS 212 has been cleaned
  - All names that have turned in *no HW at all* have been eliminated
  - If you have been eliminated, but are still in the course, you need to let me know

Basic Idea for Functions

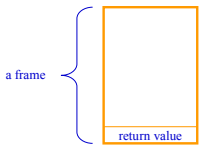
- A new *frame* (on the stack) is created for each function call
  - We use the FBR (Frame Base Register) to indicate the current frame
  - When a function returns it should "clean up" its frame

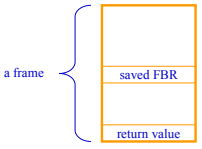
```
int main ( )
    { int i, j; } { ...;  i = A( ); ... }

int A ( )
    { int x, y; } { …; x = A( ); … }
```



What's Kept in a Frame?

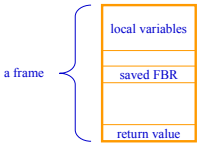
- We already have this principle:
  - When an expression is evaluated, the result is left on top of the stack
- What should be left on the stack after a function call?
- We know we have to change the FBR for each new frame
  - What do we do with the old FBR?

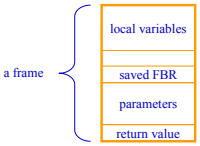




What Else is Kept in a Frame?

- Another principle:
  - Every time a function is called, it has its own local variables
- Thus it makes sense to keep a function's local variables in its frame
- The parameters of a function are also "local variables"
  - They can be kept in the frame, too

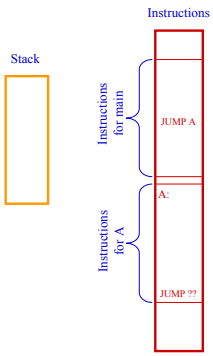




Is That It? Nothing Else in a Frame?

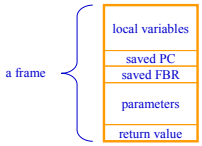
- Well, no; there's one more thing…
- We are using assembly language
  - If we want to jump somewhere and then come back then we must *remember* where to come back to

```
int main ( )
    { int i, j; } { ...;  i = A( ); ... }

int A ( )
    { int x, y; } { …; x = A( ); … }
```



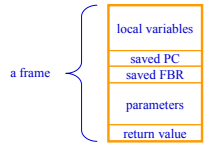
How Do We Jump Back?

- We can store the *return address* (i.e., a saved PC value) in the frame, too
- We have provided SAM instructions to store and restore the PC
  - JSR *address*
    - push PC+1 onto stack; set PC to *address*
    - Jump to SubRoutine
  - JUMPIND
    - set PC to value on top of stack
    - JUMP INDirect
- We also have instructions to save and restore the FBR
  - LINK
    - push value of FBR onto stack; set FBR to SP-1
  - POPFBR
    - set value of FBR to value on top of stack



Creating a Frame

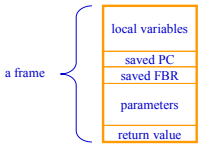
- Creation of a frame is shared by the *caller* (calling code) and the *callee* (the function's code)



- Caller's responsibilities
  - Push space for return value
  - Push arguments
  - Create new frame (use LINK = push current FBR and set FBR to SP–1)
  - JSR to callee (push PC+1 and jump to callee)
- Callee's responsibilities
  - Push space for local variables
  - Continue with callee's code

Clearing a Frame (Clean-up)

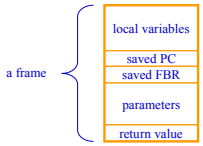
- Clearing of a frame is shared by the *callee* (the function's code) and the *caller* (calling code)



- Callee's responsibilities
  - Clear local variables from stack
  - JUMPIND to caller (clear the saved PC and jump back to calling code)
- Caller's responsibilities
  - Restore the FBR (POPFBR)
  - Clear the arguments from stack
  - Note: return value remains on stack

Access to Frame's Data

- 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

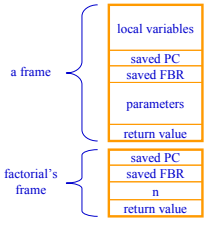
The first parameter
- STOREOFF –p

The second parameter
- STOREOFF –p + 1

The return value
- STOREOFF –p – 1

An Example

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




```
factorial: PUSHOFF -1
PUSHIMM 2
LESS
JUMPC true
JUMP false
true: PUSHIMM 1
STOREOFF -2        // Store return value
JUMPIND            // 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
POPFBR             // Restore FBR
ADDSP -1           // Clear the argument
TIMES
STOREOFF -2        // Store return value
JUMPIND            // 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
POPFBR             // Restore FBR
ADDSP -1           // Clear the argument
WRITE              // Write result
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 arguments
code for exp2
code for exp3
LINK             // Create new frame
JSR func         // Call func
POPFBR           // Restore FBR
ADDSP –3         // Remove arguments
```

- Caller's responsibilities (frame creation)
  - Push space for return value
  - Push arguments
  - Create new frame (LINK)
  - JSR to callee (push PC+1 and jump to callee)
- Caller's responsibilities (frame clean-up)
  - Restore the FBR (POPFBR)
  - Clear the arguments from stack

Code Pattern for Callee

```
retType func (type1 exp1,
    type2 exp2, type3 exp3)
    {variables}
    {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
JUMPIND          // Return to caller
```

- Callee's responsibilities (frame creation)
  - Push space for local variables
- Callee's responsibilities (frame clean-up)
  - Clear local variables from stack
  - JUMPIND to caller (clear the saved PC and jump back to calling code)

What about the "main" Function?

- The mainFunction can be called by other functions
  - Thus, it needs to behave as a callee (i.e., it participates in building a frame)
  - We need initial code to build the rest of main's frame

```
program:
ADDSP 1          // Return value for main
LINK             // Create new frame
JSR main         // Call main
POPFBR           // 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 local variables
JUMPIND          // Return to caller
```