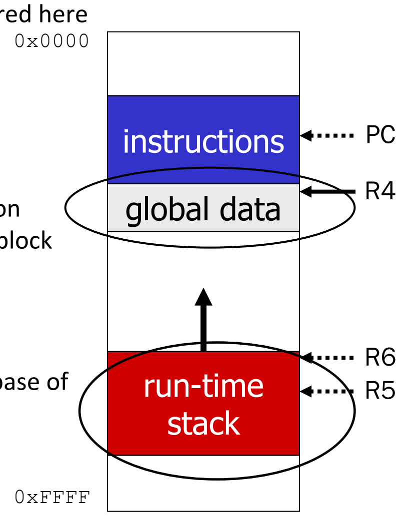
Functions (in C) & their implementation in Assembly

(Chapters 14,17)

1

Recap: LC3 Memory Allocation & Activation Records

- **Global data section:** global variables stored here
  - R4 points to beginning
- **Run-time stack:** for local variables
  - R6 points to top of stack
  - R5 points to top frame on stack
  - Local variables are stored in an activation record, i.e., stack frame, for each code block (function)
  - New frame for each block/function (goes away when block exited)
- symbol table "offset" gives distrance from base of frame (R5 for local var).
  - Address of local var = R5 + offset
  - Address of global var = R4 + offset
- return address from subroutines in R7



2

2

Implementing Functions (C to LC3)

- How to handle function calls ?
  - Where to store the data?
- implementation uses Run-time stack
  - Activation record for each function on stack
- recursion ? How is this implemented ?

Functions in C

- *Declaration* (also called prototype)
- `int Factorial(int n);`

type of return value | name of function | types of all arguments

- *Function call* -- used in expression
- `a = x + Factorial(y);`

1. execute function

2. use return value in expression

Example: Functions calling functions…

```
int mult(int a, int b) {
    int c=0 ;
    while (b > 0) {
      c=c+a ;
      b=b-1 ;
    }
    return c ;
}
int pow(int a, int p) {
    int c ;
    for (c = 1; p > 0; p--)
        c = mult(c, a) ; // performs: c=c*a
    return c ;
}
int main() {
    int a=2,b=3,c=0;
    c = pow (a, b) ; // performs: c=a^b
}
```

We'll trace these through the stack

5

5

Input Parameters/Arguments

```
int mult(int a, int b) {
    int c=0 ;
    while (b > 0) {
      c=c+a ;
      b=b-1 ;
    }
    return c ;
}
int pow(int a, int p) {
    int c ;
    for (c = 1; p > 0; p--)
        c = mult(c, a) ;
    return c ;
}
int main() {
    int a=2,b=3,c=0;
    c = pow (a, b) ; // performs: 2^3
}
```

Input Parameters

In MULT: 'a' and 'b' are input params

In POW: 'a' and 'p' are input params

'a' is not the same in both functions

6

6

Local Variables

```
int mult(int a, int b) {
   int c=0 ;
   while (b > 0) {
     c=c+a ;
     b=b-1 ;
   }
   return c ;
}
int pow(int a, int p) {
    int c ;
    for (c = 1; p > 0; p--)
       c = mult(c, a) ;
    return c ;
}
int main() {
    int a=2,b=3,c=0;
   c = pow (a, b) ; // performs: 2^3
}
```

Local Variables

`In MULT:` `'c'` local variable

`In POW:` `'c'` (a different one) is local var

`In MAIN: 'c' (also different) is local var`

7

7

Return Values

```
int mult(int a, int b) {
   int c=0 ;
   while (b > 0) {
     c=c+a ;
     b=b-1 ;
   }
   return c ;
}
int pow(int a, int p) {
    int c ;
    for (c = 1; p > 0; p--)
       c = mult(c, a) ;
    return c ;
}
int main() {
    int a=2,b=3,c=0;
   c = pow (a, b) ; // performs: 2^3
}
```

`mult` and `pow` return values of type `int`

8

8

Function Calls, Arguments, and Return Values

```
int mult(int a, int b) {
    int c=0 ;
   while (b > 0) {
     c=c+a ;
     b=b-1 ;
   }
   return c ;
}
int pow(int a, int p) {
     int c ;
     for (c = 1; p > 0; p--)
         c = mult(c, a) ;
     return c ;
}
int main() {
     int a=2,b=3,c=0;
    c = pow (a, b) ; // performs: 2^3
}
```

pow calls mult with arguments 'c' and 'a'

mult returns final value of 'c' to pow

9

9

Passing Parameters "By Value"

```
int mult(int a, int b) {
    int c=0 ;
   while (b > 0) {
     c=c+a ;
     b=b-1 ;
   }
   return c ;
}
int pow(int a, int p) {
     int c ;
     for (c = 1; p > 0; p--)
         c = mult(c, a) ;
     return c ;
}
int main() {
     int a=2,b=3,c=0;
    c = pow (a, b) ; // performs: 2^3
}
```

pow passes 'c' and 'a' to mult by value

Value of 'a' from pow is "bound" to local name 'b' in mult

In mult, 'b' is a local variable and can be modified (b = b-1)

When pow returns, 'a' in main is unaffected

10

10

Locals, Parameters, Arguments, and Return Values

```
int mult(int a, int b) {
    int c=0 ;
    while (b > 0) {
      c=c+a ;
      b=b-1 ;
    }
    return c ;
}
int pow(int a, int p) {
     int c ;
     for (c = 1; p > 0; p--)
         c = mult(c, a) ;
     return c ;
}
int main() {
     int a=2,b=3,c=0;
    c = pow (a, b) ; // performs: 2^3
}
```

One function's local variable is another's parameter

One function's return value is another's local variable

How do we organize all of these to maintain order?

11

11

Function calls.. What needs to be done?

- Caller can pass parameters to the function
- Function returns a value
- Function needs to return to caller
  - PC needs to be stored
  - "pointer" to variables used by caller needs to be restored
- Function uses local variables, so allocate space for these variables
  - New scope (i.e., new frame pointer)
- Function can be called from another function…
- capture all this information in an **Activation Record**

12

12

Activation Record/Stack Frame

- Activation record: Place to keep
  - Parameters, Local (auto) variables, Register spillage
  - Return address
  - Return value
  - Old frame pointer
- Frame pointer R5 points to beginning of a region of activation record for the function

13

13

Run-Time Stack

• local variables are stored on the run-time stack in an *activation record (i.e., stack frame)*

•Frame pointer (R5) points to the beginning of a region of activation record that stores local variables for the current function

•When a new function is called, its activation record is pushed on the stack;

•when it returns, its activation record is popped off of the stack

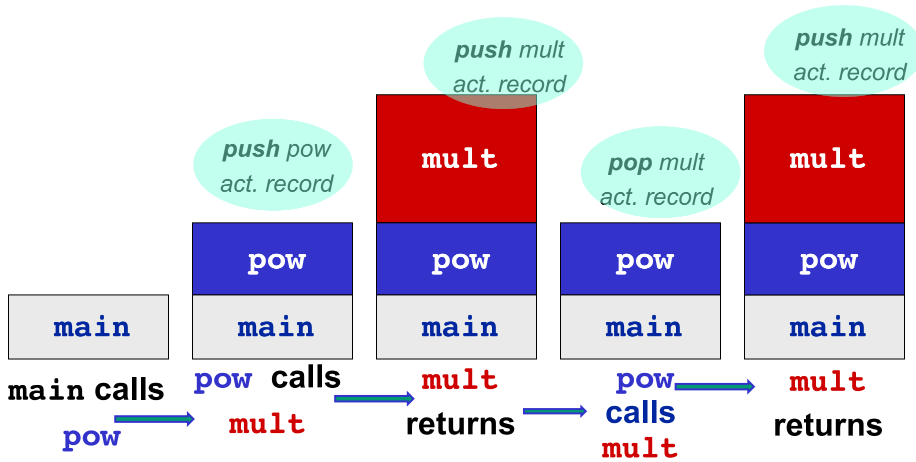
- Allows recursion

14

14

Function Calls & Stack Frames (Activation Records)

- Stack managed in function-sized chunks called frames or activation records
  - This all happens at run-time



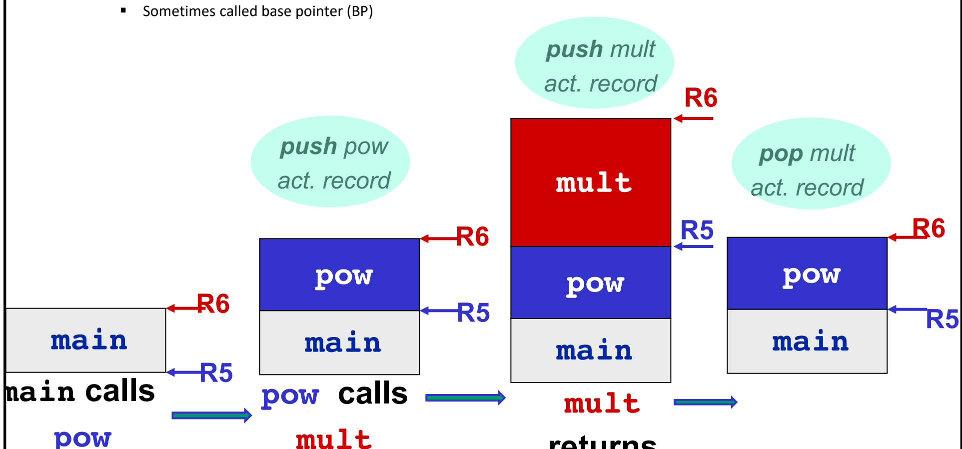
In C, top-level user function is always ‘main’

15

15

Frame Pointer (R5) and Stack Pointer (R6)

- LC3 uses two more registers as part of calling convention
  - R6 is the stack pointer (SP), “points to” current “top” of stack
  - R5 is the frame pointer (FP), “points to” bottom of current frame
    - Sometimes called base pointer (BP)



In x86, esp and ebp are the stack and base pointer registers

16

16

Some info to keep in Activation Record: Bookkeeping records

•Return value

- space for value returned by function
- allocated even if function does not return a value

•Return address

- save pointer to next instruction in calling function
- convenient location to store R7 in case another function (JSR) is called

•Dynamic link

- caller’s frame pointer
- used to pop this activation record from stack

17

17

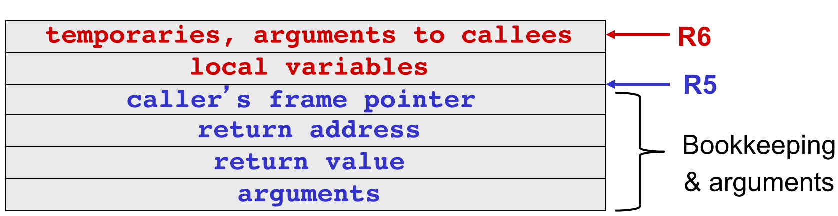
The Stack Frame Layout (Activation Records)

•In caller’s stack frame: addresses > R5

- Caller’s saved frame pointer
- return address, return value
- arguments

•In running function’s stack frame: addresses <= R5

- Local variables
- temporaries
- arguments to running function’s callees



18

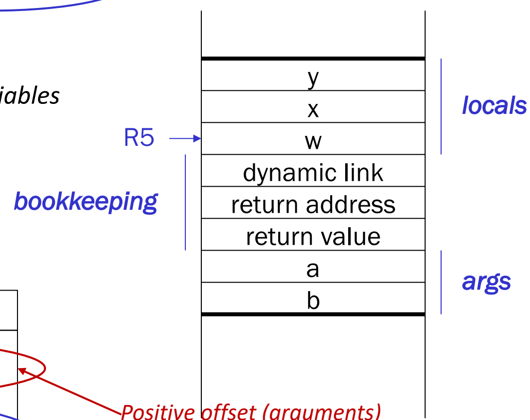
18

Activation Record Example

```
•int NoName(int a, int b)
{
   int w, x, y;
   .
   .
   .
   return y;
}
```

arguments

Local variables



Name	Type	Offset	Scope
a	int	4	NoName
b	int	5	NoName
w	int	0	NoName
x	int	-1	NoName
y	int	-2	NoName

Positive offset (arguments)

Local vars – 0 or negative offset

19

Recursion

•A recursive function is one that solves its task by calling itself on smaller pieces of data.

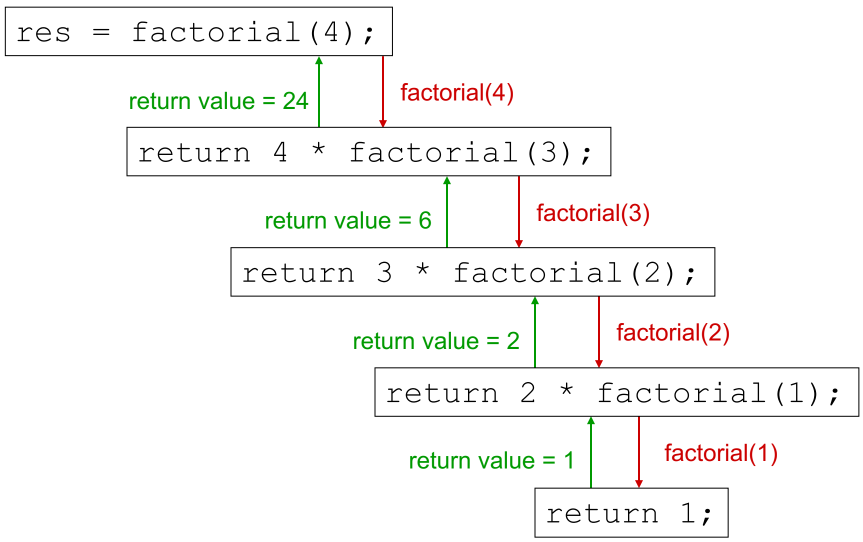
- recurrence function in mathematics – use this to prove correctness of recurrence functions (induction !!)
- Like iteration -- can be used interchangeably; sometimes recursion results in a simpler solution.

• Example: Factorial(n) = n*(n-1)*(n-2)*..*2*1

```
int factorial(n){
   if (n >1) return n*factorial(n-1)
     else return 1;
}
/* call from main */
 res=factorial(n);
```

20

Executing Factorial



21

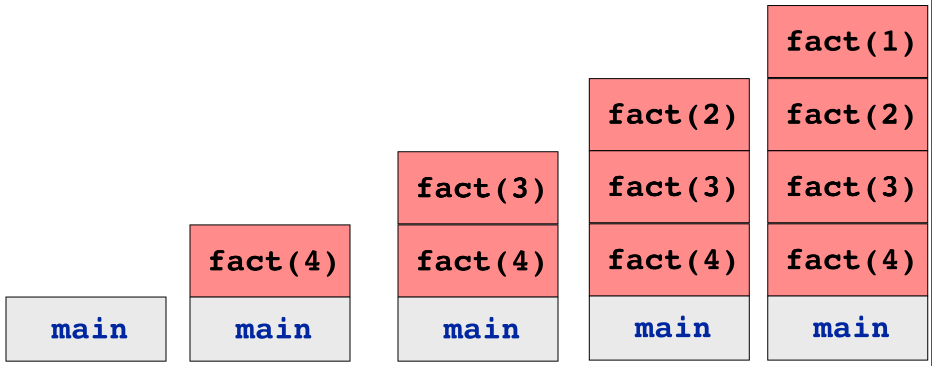
How is recursion implemented ?

- Do we need to do anything different from how we handled function calls ?
- No!
  - Activation record for each instance/call of Fibonacci !

22

22

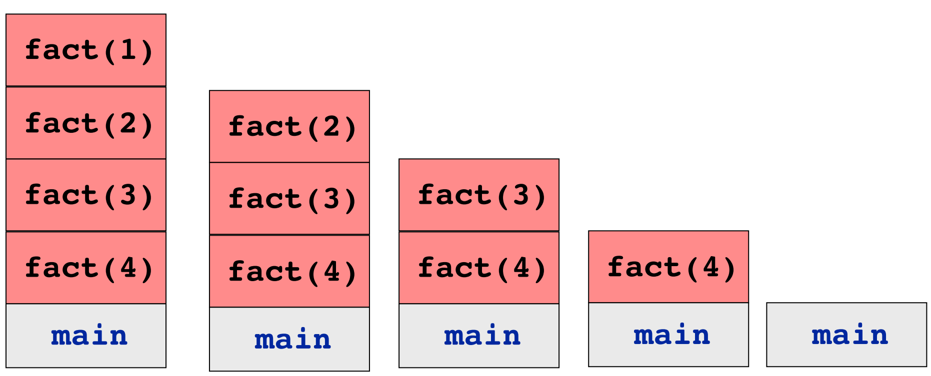
Sequence of stack frames during factorial(4) execution



23

23

Returning from each instance of factorial

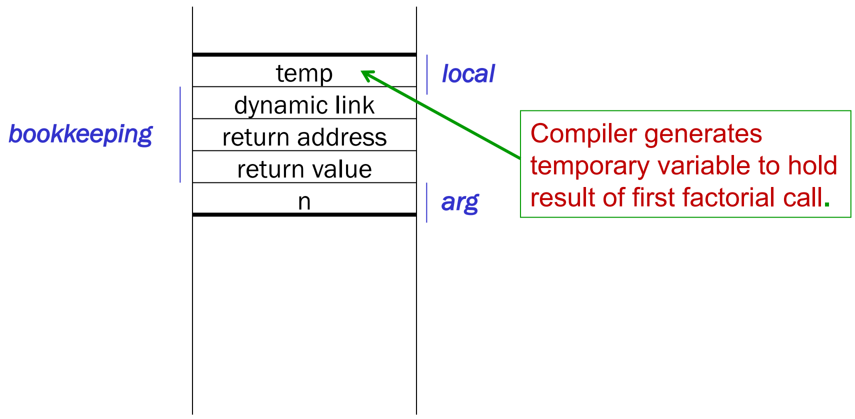


24

24

Factorial: LC-3 Code

- Activation Record
-



25

25

Calling Convention

- Compilers typically compile functions separately
  - Generate assembly for main(), `mult()`, and `pow()` independently
  - Why? They may be in different files, `mult` may be in a library, etc.

- This necessitates use of calling convention
  - Some standard format for arguments and return values
  - Allows separately compiled functions to call each other properly
  - In LC-3, we've seen part of its calling convention:
    - R7=return address

- Calling convention is function of HLL, ISA, and compiler
  - Why code compiled by different compilers may not inter-operate

26

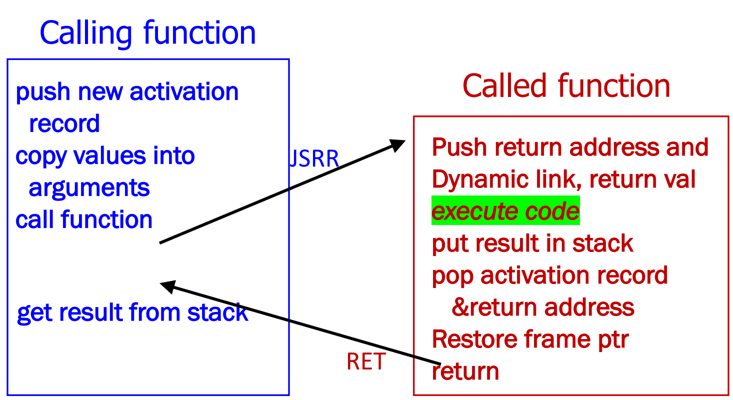
26

Functions in C & Translation to Assembly: Part 2 – Memory Layout during Function Call and Return

27

Implementing Functions: Overview

- Activation record
  - information about each function, including arguments and local variables
  - stored on run-time stack



28

28

Caller and Callee: Who does what?

- Caller
  - Puts arguments onto stack (R→L)
  - Does a JSR (or JSRR) to function
- Callee
  - Makes space for Return Value and Return Address (and saves Return address. i.e., R7)
  - makes space for and saves old FP (Frame Pointer)
    - Why ?
  - Makes FP point to next space
  - Moves SP enough for all local variables
  - Starts execution of "work" of function

29

29

Who does what?

- Callee (continued)
  - As registers are needed their current contents can be spilled onto stack
  - When computation done...
  - Bring Stack Pointer SP back to base
  - Restore Frame Pointer FP (adjust SP)
  - Restore Return Address RA (adjust SP)
  - Leave SP pointing at return value
  - RET

  (Bring Stack Pointer … Leave SP pointing at return value, RET:) ***Prepare to return to caller***
- Caller (after RET)
  - Grabs return value and uses it

- Observe the steps needed to support function call and return
  - These steps do not do any of the function's work

30

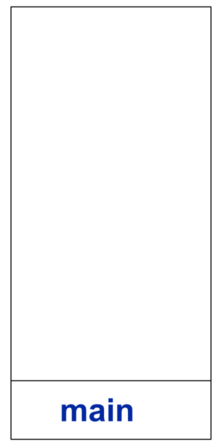
30

Example : Function Call

Show contents of stack/memory at each time:

```
int main{
        int a,b;
                                  ← Time 1
        a=5
        b=foo(a);   /* assume b=foo(a) is at address 2100 */
                                  ← Time 5
        …}
int bar(int q, int r){
        int k, m;
                                  ← Time 3
        k=q+r;
        return k;
}                                 ← Time 4a
int foo(int a){
        int w;
        w=8;
        w = bar(w,10);            ← Time 2
                    /* w=bar(w,10) is at address 2200 */
        return w;                 ← Time 4b
}
```

R5=#3000 (for main)



First construct Symbol Table

Identifier	Type	Offset	Scope
a	int	0	main
b	int	-1	main
w	int	0	foo
a	int	4	foo
k	int	0	bar
m	Int	-1	bar
q	int	4	bar
r	int	5	bar

main (a, b) · **foo** (w, a) · **bar** (k, m, q, r)

arguments to function: positive offset

Remember: 3 places in Activation record

for Ret.Addr, old FP, Return value => arguments start at offset 4+

Memory Contents at Time 1 (Run time stack): when main starts

```
int main {
      int a,b; /*time 1 */
      a=5;
      b=foo(a); @2100
      ..} /*time 5*/
int bar(int q, int r){
      int k,m;
      /*Time 3 */
      k=q+r;
      return k; /*time 4a */
      }
int foo(int a){
      int w; /*Time 2*/
      w=8;
      w=bar(w,10);/*time 4b*/
      return w;}
```

Address	Content/Identifier	Value	
2996			
2997			
2998			
2999	b (local var)	?(not initialized)	main
3000	a (local var)	?(not initialized)	main

R6:2999 TOS → 2999

R5=3000 frame pointer → 3000

33

33

Memory contents at Time 2: after foo is called and just before foo executes its first instruction

```
int main {
      int a,b; /*time 1 */
      a=5;
      b=foo(a);
      ..} /*time 5*/
int bar(int q, int r){
      int k,m;
      /*Time 3 */
      k=q+r;
      return k; /*time 4a */
      }
int foo(int a){
      int w; /*Time 2*/
      w=8;
      w=bar(w,10);/*time 4b*/
      return w;}
```

Address	Content	Value	
2989			
2990			
2991			
2992			
2993			
2994	w (local var)	?	foo
2995	dynamic link (for main)	3000	foo
2996	return addr (to main)	2101	foo
2997	return value		foo
2998	argument: a	5	foo
2999	b (local var)	?	main
3000	a (local var)	5	main

R6:2994 → 2994

R5=2994 frame pointer → 2994

34

34

Memory Contents at Time 3: when foo calls bar, just before bar executes first instruction.

R6:2987

R5=2988
frame ptr

Address	Content	Value
2987	m (local var)	?
2988	k (local var)	?
2989	dyn.link (to foo)	2994
2990	ret. addr. (to foo)	2201
2991	ret.value	
2992	q (argument)	8
2993	r (argument)	10
2994	w (local var)	8
2995	dynamic link	3000
2996	return addr	2101
2997	return value	
2998	argument: a	5
2999	b (local var)	?
3000	a (local var)	5

bar (2987–2993)

foo (2994–2998)

main (2999–3000)

```
int main {
      int a,b; /*time 1 */
      a=5;
      b=foo(a);
      ..} /*time 5*/
int bar(int q, int r){
      int k,m;
      /*Time 3 */
      k=q+r;
      return k; /*time 4a */
      }
int foo(int a){
      int w; /*Time 2*/
      w=8;
      w=bar(w,10);/*time 4b*/
      return w;}
```

35

35

Memory Contents at Time 4 (a): at RET instruction in bar

R6:2991
Points to return val

R5=2994
frame pointer

Address	Content	Value
2987		
2988		
2989		
2990		
2991	ret.value	[k]value of k **18**
2992	q (argument)	8
2993	r (argument)	10
2994	w (local var)	8
2995	dynamic link	3000
2996	return addr	2101
2997	return value	
2998	argument: a	5
2999	b (local var)	?
3000	a (local var)	5

foo (2994–2998)

main (2999–3000)

```
int main {
      int a,b; /*time 1 */
      a=5;
      b=foo(a);
      ..} /*time 5*/
int bar(int q, int r){
      int k,m;
      /*Time 3 */
      k=q+r;
      return k; /*time 4a */
      }
int foo(int a){
      int w; /*Time 2*/
      w=8;
      w=bar(w,10);/*time 4b*/
      return w;}
```

36

36

Memory Contents at Time 4 (b) (after foo resumes)

Address	Content	Value
2987		
2988		
2989		
2990		
2991		
2992		
2993		
2994	w (local var)	[k] **18**
2995	dynamic link	3000
2996	return addr	2101
2997	return value	
2998	argument: a	5
2999	b (local var)	?
3000	a (local var)	5

R6:2994 → 2994

R5=2994 frame ptr → 2994

foo: 2994–2998

main: 2999–3000

```
int main {
      int a,b; /*time 1 */
      a=5;
      b=foo(a);
      ..} /*time 5*/
int bar(int q, int r){
      int k,m;
      /*Time 3 */
      return k; /*time 4a */
      }
int foo(int a){
      int w; /*Time 2*/
      w=8;
      w=bar(w,10);/*time 4b*/
      return w;}
```

37

37

Time 5: Memory contents after foo returns and before main completes

Address	Content/Identifier	Value
2996		
2997		
2998		
2999	b (local var)	5 **18**
3000	a (local var)	[foo(5)] **5**

R6:2999 → 2999

R5=3000 frame ptr → 3000

main: 2999–3000

```
int main {
      int a,b; /*time 1 */
      a=5;
      b=foo(a);
      ..} /*time 5*/
int bar(int q, int r){
      int k,m;
      /*Time 3 */
      return k; /*time 4a */
      }
int foo(int a){
      int w; /*Time 2*/
      w=8;
      w=bar(w,10);/*time 4b*/
      return w;}
```

38

38

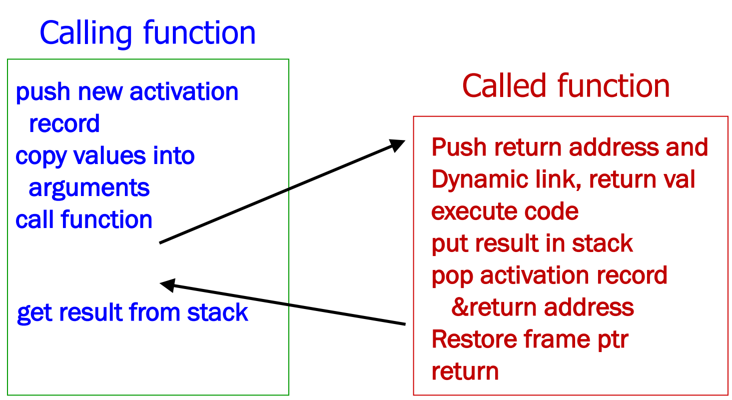
Functions in C : Part 3 – LC3 Instructions to implement function call and return

Details in the textbook (some in notes)….
Required reading (hint: Exam 2!)

39

Implementing Functions: Overview

- Activation record
  - information about each function, including arguments and local variables
  - stored on run-time stack



40

40

Example: Calling the Function

```
int main()
 {int a,b;
       a=5;
       b=foo(a);
}
int bar(int q, int r)
{
  int k;
  int m;
  ...
       return k;
}
int foo(int a)
{
  int w;
       w=8;
       ...
       w = bar(w,10);
       ...
       return w;
}
```

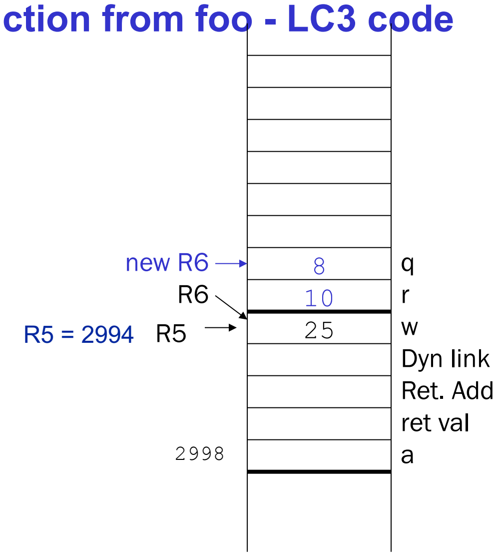
41

41

Step 1: Calling the Function from foo - LC3 code

- w = bar(w,10);
- ; push arguments

```
; call subroutine
JSR  bar
```



Note: Caller needs to know number and type of arguments,
doesn't know about local variables. It needs to push the arguments and then JSR

42

42

Step 1: Calling the Function

- **w = bar(w, 10);**
- **; push second arg**

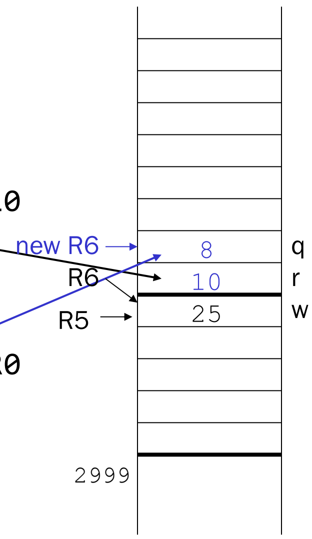
```
AND  R0, R0, #0
ADD  R0, R0, #10 ;  set R0 to10
ADD  R6, R6, #-1
STR  R0, R6, #0; push R0
```

- **; push first argument**

```
LDR  R0, R5, #0   ;read w to R0
ADD  R6, R6, #-1 ;
STR  R0, R6, #0   ;push R0
```

- **; call subroutine**

JSR bar ; or JSRR



Note: Caller needs to know number and type of arguments, doesn't know about local variables. It needs to push the arguments and then JSR

43

43

Next steps: starting Callee function

- Create space for return value
- Store/push return address
- Store/push frame pointer
- Set/push new frame pointer
- Set space for local variables

44

44

Starting the Callee Function

- ; leave space for return value
- ; push return address

…

- ….

; push dyn link (caller's frame ptr)

…

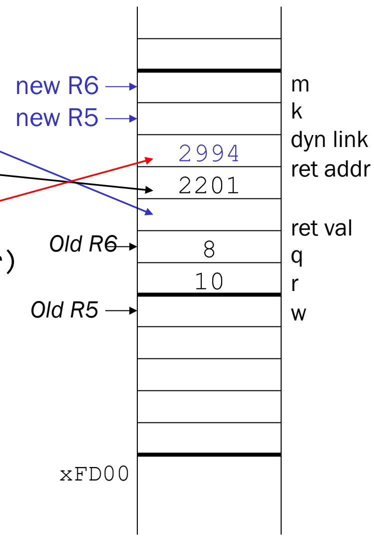
- ; set new frame pointer

….

; allocate space for locals

….

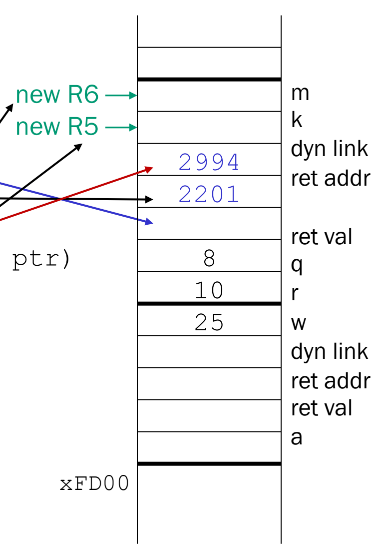
```
int bar(int q, int r)
{
  int k;
  int m;
  …
  return k;
}
```



Starting the Callee Function

- ; leave space for return value
  ADD R6, R6, #-1
- ; push return address
  ADD R6, R6, #-1
  STR R7, R6, #0
- ; push dyn link (caller's frame ptr)
  ADD R6, R6, #-1
  STR R5, R6, #0
- ; set new frame pointer
  ADD R5, R6, #-1
- ; allocate space for locals
  ADD R6, R6, #-2

```
int bar(int q, int r)
{
  int k;
  int m;
  …
  return k;
}
```



Returning from function…steps

- Write return value
- return address into R7
- Restore old frame pointer
- Pop local variables
- Where should top of stack point to after RET?
  - To the return value

- Go through the notes/textbook for remaining steps….

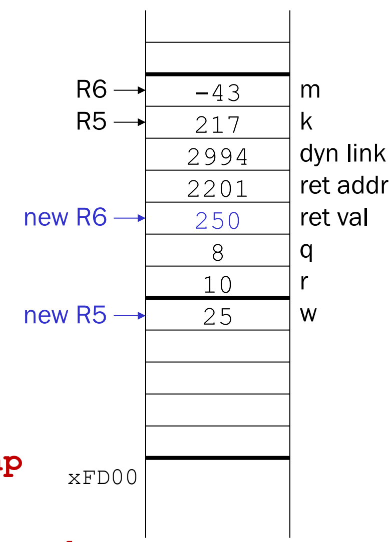
Ending the Callee Function

```
•return k;
•; copy k into return value
; pop local variables

; pop dynamic link (into R5)

; pop return addr (into R7)

; return control to caller
RET
```



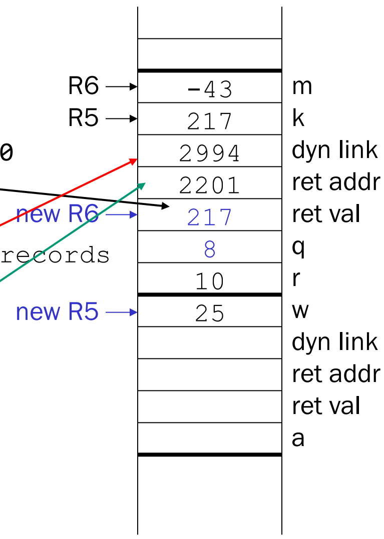
- **Now we understand why swap Did not work!**

  In C arguments are passed by value

Ending the Callee Function

• return k;

```
•; copy k into return value
LDR  R0, R5, #0   } ; load k into R0
STR  R0, R5, #3     ; store R0
•; pop local variables
ADD  R6, R5, #1; TOS=bookkeeping records
•; pop dynamic link (into R5)
LDR  R5, R6, #0   } ; pop
ADD  R6, R6, #1
•; pop return addr (into R7)
LDR  R7, R6, #0   }
ADD  R6, R6, #1
; return control to caller
RET
```



49

49

Back to caller…steps

- What should caller do now?

- Get return value
- Clear arguments

50

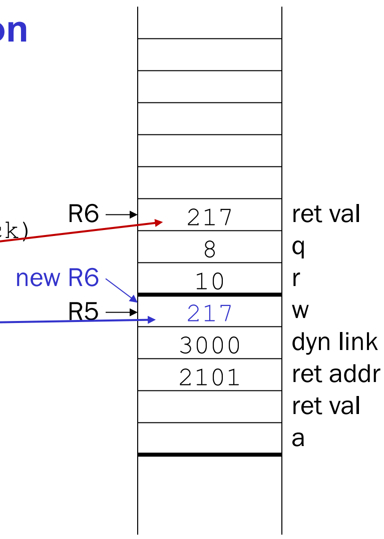
50

Resuming the Caller Function

• **w = bar(w,10);**

```
•JSR   bar

; load return value (top of stack)
LDR   R0, R6, #0
; perform assignment
STR   R0, R5, #0
; pop return value
ADD   R6, R6, #1
; pop arguments
ADD   R6, R6, #2
```



And We're Done…

- with that call/return sequence
  - Stack frame back to where it was

- A lot of extra instructions involved in function call
  - Many compilers try to inline functions
    - Expand function body at call site
  - Removes most of call overhead
  - Introduces other overheads
    - Multiple static copies of same function

Prologue, Body, Epilogue

- Steps at start of function that we saw are called function prologue
  - Setup code compiler generates automatically
    - One of the (few) abstractions C provides over assembly
  - More sophisticated compilers can generate tighter prologues

- Code that follows is translation of function body
  - lcc does this statement-by-statement
    - Results in many inefficiencies
  - More sophisticated compilers view entire function (at least)
    - Gives us opportunity to *optimize* the code

- When explicit body finishes, need function epilogue
  - Cleanup code compiler generates automatically
  - epilogue (unwinding/popping of the stack)
- Observation: a lot of extra instruction involved in function call
  - If we *inline* the function, we can eliminate the call overhead…..
  - Remember this when we get to our last topic of code optimization

53

53

Things to notice

- 1) Arguments are pushed onto stack right-to-left
  - So that first argument from left is closest to callee
  - This is called C convention (left-to-right is called PASCAL)
  - Needed for functions with *variable* argument counts (e.g., printf)
- 2) C is pass-by-value (not pass-by-reference)
  - Functions receive "copies" of local variables
    - Recall, arguments to functions were copies of local vars
  - Protects local variables from being modified accidentally

3) We see why variables must be declared at start of function
  - Size of static/automatic variables are known at compile time:
    - ADD R6, R6, #-1 ; allocate space for local vars
  - Also, compiler may compile line-by-line, hence right up front!

54

54

Caller and Callee Saved Registers

- R5, R6, and R7 actively participate in call/return sequence
  - What happens to R0–R4 across call?
  - "Callee saved": callee saves/restores if it wants to use them
  - "Caller saved": caller saves/restores if it cares about values
- Turns out… LCC doesn't have a convention for these???
  - Doesn't have to (because it compiles statement-by-statement)
  - At the end of every statement, all local variables are on stack
  - R0–R4 are used just as "temporary" storage within expressions
    - Highly inefficient
  - Register allocation: assign locals to registers too
    - Avoid many unnecessary loads and stores to stack
    - All real compilers do this

55

55

Summary of LC-3 Function Call Implementation

1. Caller pushes arguments (last to first).
2. Caller invokes subroutine (JSR).
3. Callee allocates return value, pushes R7 and R5.
4. Callee allocates space for local variables.
5. Callee executes function code.
6. Callee stores result into return value slot.
7. Callee pops local vars, pops R5, pops R7.
8. Callee returns (JMP R7).
9. Caller loads return value and pops arguments.
10. Caller resumes computation…

56

56

For details ..

- Read Chapter 14 – section 14.3, Figure 14.8 for full implementation of the function call process
- Check out the lcc cross compiler

57

57

Question…What can go wrong?

- What if the return address was overwritten: Where does program return to ?
- returns to whatever was written in that location on the run-time stack!
- Buffer overflow attack/stack smashing attack
  - Return to this after discussion of arrays

58

58

Function Calls -- Summary

- Activation records keep track of caller and callee variables
  - Stack structure
- What happens if we “accidentally” overwrite the return address ?

- Next: Pointers, Arrays, Dynamic data structures and the heap