

# Implementing Functions at the Machine Level

(section 12.5 and 14.3)

Based on slides © McGraw-Hill  
Additional material © 2004/2005 Lewis/Martin  
Modified by Diana Palsetia

## Subroutines

Core operations:

1) callers passes parameters to callee and the control is transferred to callee

2) the callee does its task

3) a return value is passed back to caller and control returns to the caller

CIT 593

2/23

## Passing Arguments & Returns using Stack

In real world

- Arguments and returns values are passed using stacks

Steps:

- Calling routine pushes arguments on the stack and calls the subroutine
- Called routine
  - Uses passed arguments from the stack (pops the arguments)
  - Pushes any return value onto the stack
- Finally, the calling routine then retrieves (pops) the return value(s) from the stack and continues execution

CIT 593

3/23

## Function Call scenario

```
Main  ...
      JSR  Foo
Next  ...
      HALT

Foo   ST   R7, SaveR7
      AND  R0, R0, #0
      ...
      JSR  Foo
After ...
      LD   R7, SaveR7
      RET

Save7 .FILL #0
Counter .FILL #0

.END
```

First call to Foo

- SaveR7 contains address of Next

Second call to Foo

- SaveR7 contains address of After

First return from Foo

- Returns to After

Second return from Foo

- Returns to After again!!!

CIT 593

4/23

## Solution to Recursion problem

For each subroutine call, need a mechanism to distinguish its invocation

- This is known as *activation record*

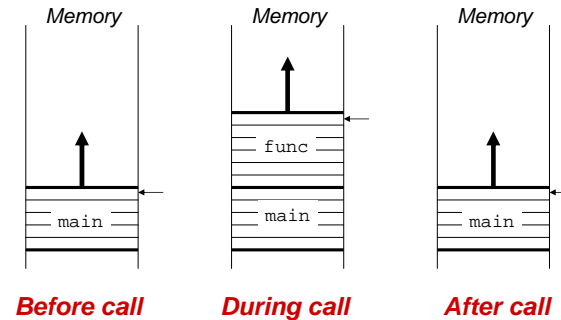
**Activation Record** contains

- Invocation-specific data (e.g., local variables, saved registers, arguments and returns)
- Need to store this information per function call and discard when the function call is over
- **Stack data** structure fits this description well
  - When function is called we store (push) record on the stack
  - When function call is over we discard (pop) record from the stack

CIT 593

5/23

## Big Picture



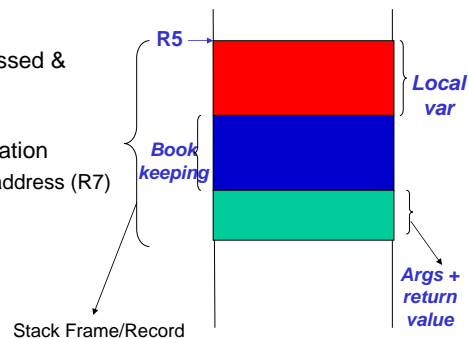
CIT 593

6/23

## Complete view of an activation record

**Activation Record of a called function:**

1. input parameters passed & return value
2. book keeping information
  - E.g. Callers Return address (R7)
3. local variables



CIT 593

7/23

## Frame Pointer

Each function gets a record (a.k.a frame), but we need to somehow

- Delineate one function's record from other, especially if we are in a recursive call
- Track within an activation record the local variables, book keeping info, and arguments

This tracking is done by what is called the **frame pointer**

- Frame pointer is the address which points **to the top of the record/frame**
- Once we know the frame pointer, all information can be accessed via frame pointer + offset

**Note:** By convention in LC3, R5 holds frame pointer

CIT 593

8/23

## Activation Record

### Arguments

- The calling routine places the arguments on top of the stack before calling a function
- The callee will use the arguments from the stack to do its work

### Book keeping

#### 1. Callers Frame Pointer

- Need to save callers frame pointer so that when the control returns to the caller, it will be able to access its own local variable, arguments etc.
  - If we destroy this value then we have trouble restarting the caller correctly when the callee finishes

CIT 593

9/23

## Activation Record contd..

### Book Keeping Record

#### 2. Return address

- Save pointer to next instruction in calling function
- Convenient location to store R7
  - Especially helpful during recursive calls

#### 3. Saved Registers

Save all registers that will be used for temporary work in the function

#### 4. Returns

- Every function always allocates a space for return value on the activation record, whether or not it returns anything
- Even if the function return is void

CIT 593

10/23

## Activation Record contd..

### Local Variables

- Local Variables are added after arguments and book keeping information
- They are added in the order they are declared
  - Therefore variable declared last is always on the top of the stack (LIFO)

#### Compiler tracks each variable

- Just like assembler, maintains symbol table for labels
- Compiler stores for very variable
  - 1.Name (identifier)
  - 2.Type
  - 3.Location in memory
  - 4.Scope

Name	Type	Offset	Scope

Compiler only stores offset and uses R5 as the base address

CIT 593

11/23

## Example local variables of main()

```

int main()
{
    double amt = 0;
    int p = 0;
    int t = 0;
    int r = 0;
    ...
    amt = p * t * r;
}
    
```

LC3 initialization code:

```

ADD R0, R0, #0
STR R0, R5, #3
STR R0, R5, #2
STR R0, R5, #1
STR R0, R5, #0
    
```

R5 →

r = 0
t = 0
p = 0
amt = 0

Compiler's Symbol Table

Name	Type	Offset	Scope
amt	double	3	main
p	int	2	main
t	int	1	main
r	int	0	main

We know that R5 contains the frame pointer of function main(). Now variables in main can accessed by :

Frame Pointer + Offset

CIT 593

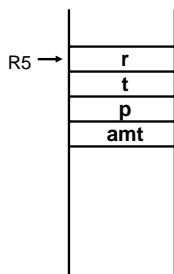
12/23

## Example local variables of main() (contd..)

To calculate `amt`, we can now easily access local variables `p`, `t`, and `r` using the frame pointer `R5`

■ Assigning `amt = p * t * r`;

```
LDR R1, R5, #2 ;R1 = p
LDR R2, R5, #1 ;R2 = t;
LDR R3, R5, #0 ;R3 = r
MUL R1, R1, R2 ;R1 = p * t
MUL R1, R1, R3 ;R1 = p * t * r
STR R1, R5, #3 ; Store R1 = amt
```



Compiler's Symbol Table

Name	Type	Offset	Scope
amt	double	3	main
p	int	2	main
t	int	1	main
r	int	0	main

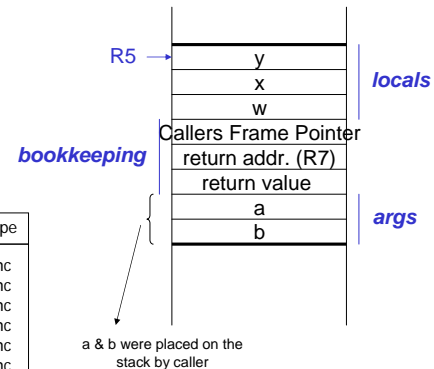
Note: (explanation deviates from book section 12.5)

CIT 593

13/23

## Example of a Complete Activation Record

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



Name	Type	Offset	Scope
b	int	7	func
a	int	6	func
"ret. value"	int	5	func
w	int	2	func
x	int	1	func
y	int	0	func

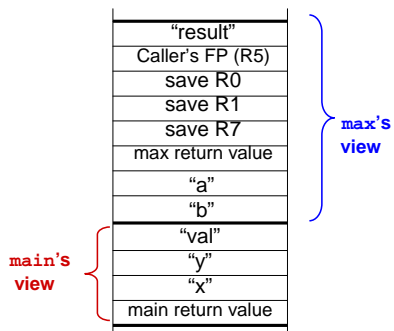
CIT 593

14/23

## Function Call Example

```
int main()
{
    int x, y, val;
    x = 10;
    y = 11;
    val = max(x + 10, y);
    printf("%d", val);
    return 0;
}

int max(int a, int b)
{
    int result = 0;
    if (b > a) {
        result = b;
    }
    else {
        result = a;
    }
    return result;
}
```



Why haven't I shown save R7 and frame pointer R5 for main() ?

CIT 593

15/23

## Main Function (1 of 2)

```
MAIN: ;R6 is initialized STACKBASE by OS code in LC3
      ADD R6, R6, #-1 ; Space for return value

Local variables of main()
      AND R0, R0, #0 ; x = 10
      ADD R0, R0, #10
      ADD R6, R6, #-1
      STR R0, R6, #0

      AND R0, R0, #0 ; y = 11
      ADD R0, R0, #11
      ADD R6, R6, #-1
      STR R0, R6, #0

      ADD R6, R6, #-1 ; space for variable "val"

;Now we have complete record of main() on the stack
      ADD R5, R6, #0 ; R5 = Main's Frame Pointer

Args for callee (i.e. MAX())
      LDR R0, R5, #1 ; load y into R0
      ADD R6, R6, #-1
      STR R0, R6, #0 ;arg for max() is put on stack

      LDR R1, R5, #2 ; load x into R1
      ADD R1, R1, #10 ; R1 = x + 10
      ADD R6, R6, #-1
      STR R1, R6, #0 ;arg for max() is put on stack

      JSR MAX ; call max function

... ; more here (Main Function (2 of 2))
```

CIT 593

16/23

### Max Function (1 of 3)

```
;NOTE that R6 always points to top of the stack
MAX:  ADD R6, R6, #-1 ;allocate spot for return var

      ADD R6, R6, #-1 ;
      STR R7, R6, #0  ; save R7 (link register)

      ;we will be using R1 and R0
      ;within func so we must save them
      ADD R6, R6, #-1
      STR R1, R6, #0  ; save R1
      ADD R6, R6, #-1
      STR R0, R6, #0  ; save R0

      ADD R6, R6, #-1
      STR R5, R6, #0  ; Save Main's Frame Pointer

      AND R0, R0, #0
      ADD R6, R6, #-1
      STR R0, R6, #0  ;Max's local var (result = 0)

      ADD R5, R6, #0  ; Max's Frame Pointer

      LDR R0, R5, #6   ; load "a"
      LDR R1, R5, #7   ; load "b"
```

CIT 593

17/23

### Max Function (2 of 3)

```
      NOT R0, R0      ; calculate -a
      ADD R0, R0, #1

      ADD R0, R1, R0   ; compare (i.e. b - a > 0)
      BRP BGRTA

      LDR R0, R5, #6   ; load "a"
      STR R0, R5, #0   ; store "a" into "result"
BGRTA: STR R1, R5, #0   ; store "b" into "result"

      LDR R1, R5, #0   ;load "result"
      STR R1, R5, #5   ;store "result" into return
                        ;value

      ;start deactivating the record
      ;pop of local variable of max()
      ADD R6, R6, #1
```

CIT 593

18/23

### Max Function (3 of 3)

```
;restore Main's frame pointer
LDR R5, R6, #0   ; restore R5 (main's frame pointer)
ADD R6, R6, #1   ; pop the SAVE R5 (Mains' FP)

;restore other register
LDR R0, R6, #0   ; restore R0
ADD R6, R6, #1   ; pop SAVE R0

LDR R1, R6, #0   ; restore R1
ADD R6, R6, #1   ; pop SAVE R1

LDR R7, R6, #0   ; restore R7 (link register)
ADD R6, R6, #1   ;pop the return address (SAVE R7)

RET
```

CIT 593

19/23

### Main Function (2 of 2)

```
; previous code here

JSR MAX          ; call max function

LDR R0, R6, #0   ; read return value of max
STR R0, R5, #0   ; put value into local "val"

;Do code for printf()

AND R0, R0, #0
STR R0, R5, #3   ;"return 0" for main's exit

ADD R6, R5, #3   ;pop of main's local variables

HALT
```

CIT 593

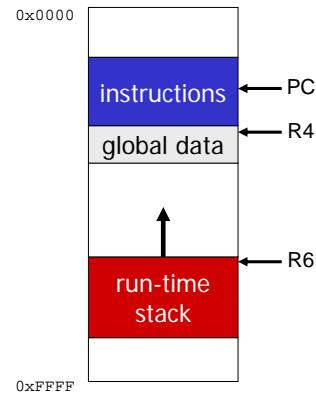
20/23

## Scope of Local Variable & Pass by Value

- Once the function does its work, the activation record is popped of the stack

1. Hence the variable ceases to exist after the method body i.e. { }

2. When arguments are passed they are passed by value i.e. only the copy of their value is passed



CIT 593

21/23

## Allocating Space for Global Variables

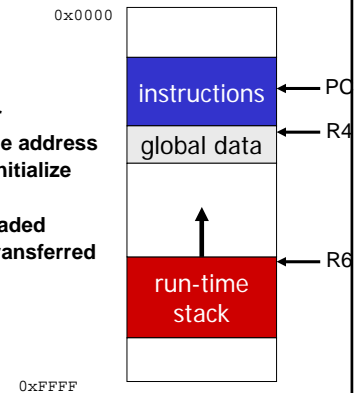
### Global data section

- All global variables stored here (actually all static variables)

- The memory address where global data section starts is kept a register

- In LC3, we assume R4 holds the address
- So programmer access R4 to initialize and assign global variables
- The initial reference of R4 is loaded by the OS before the control is transferred to user code

Note:  
R4 is just a convention in LC3, could be different for different ISAs



CIT 593

22/23

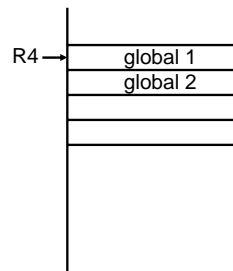
## Example of Global Variable

- Example of C program:

```
int global1 = 0;
int global2 = 0;
int main(){...}
```

- LC3 Equivalent code

```
AND R0, R0, #0;
LDR R0, R4, #0;
LDR R0, R4, #1;
```



Compiler's Symbol table

Identifier	Type	Location (offset)	Scope
global1	int	0	global
global2	int	1	global

CIT 593

23/23