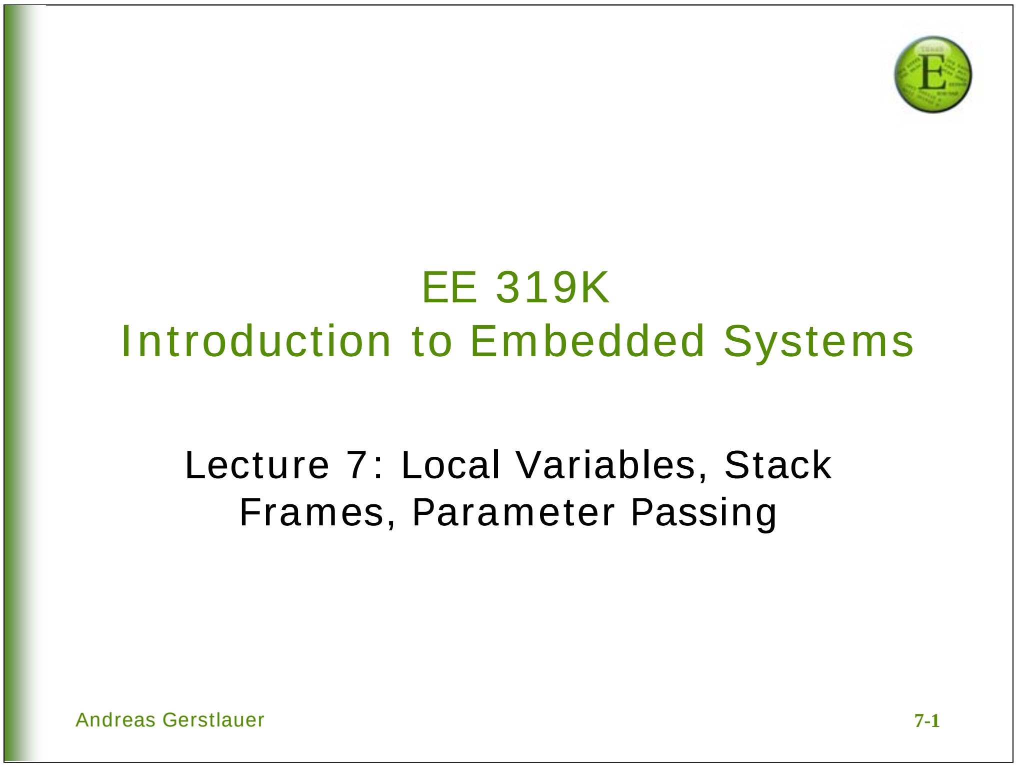
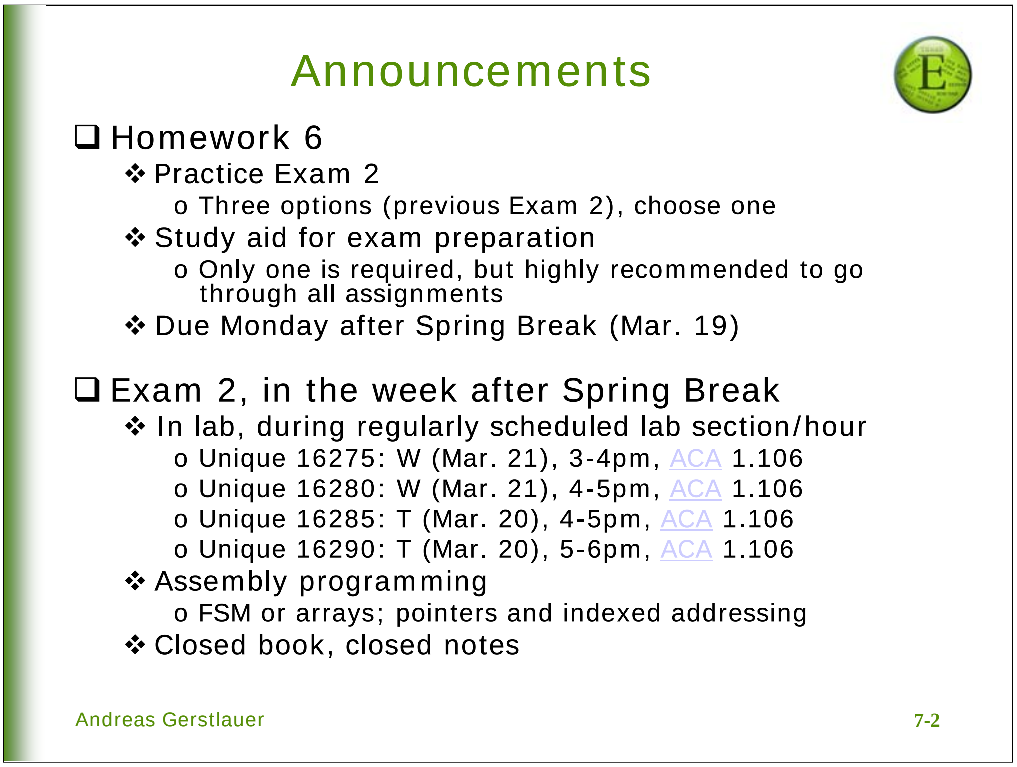


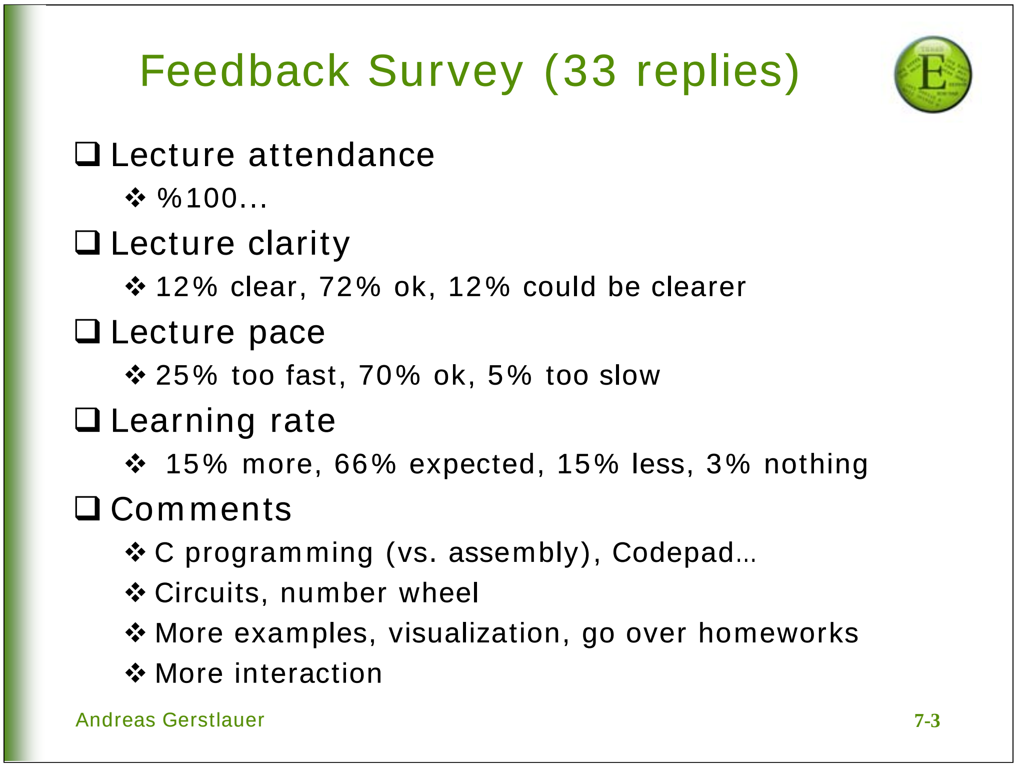
EE 319K
Introduction to Embedded Systems

Lecture 7: Local Variables, Stack Frames, Parameter Passing



Announcements

- ❑ Homework 6
  - ❖ Practice Exam 2
    - o Three options (previous Exam 2), choose one
  - ❖ Study aid for exam preparation
    - o Only one is required, but highly recommended to go through all assignments
  - ❖ Due Monday after Spring Break (Mar. 19)
- ❑ Exam 2, in the week after Spring Break
  - ❖ In lab, during regularly scheduled lab section/hour
    - o Unique 16275: W (Mar. 21), 3-4pm, ACA 1.106
    - o Unique 16280: W (Mar. 21), 4-5pm, ACA 1.106
    - o Unique 16285: T (Mar. 20), 4-5pm, ACA 1.106
    - o Unique 16290: T (Mar. 20), 5-6pm, ACA 1.106
  - ❖ Assembly programming
    - o FSM or arrays; pointers and indexed addressing
  - ❖ Closed book, closed notes



Feedback Survey (33 replies)

❑ Lecture attendance

- ❖ %100…

❑ Lecture clarity

- ❖ 12% clear, 72% ok, 12% could be clearer

❑ Lecture pace

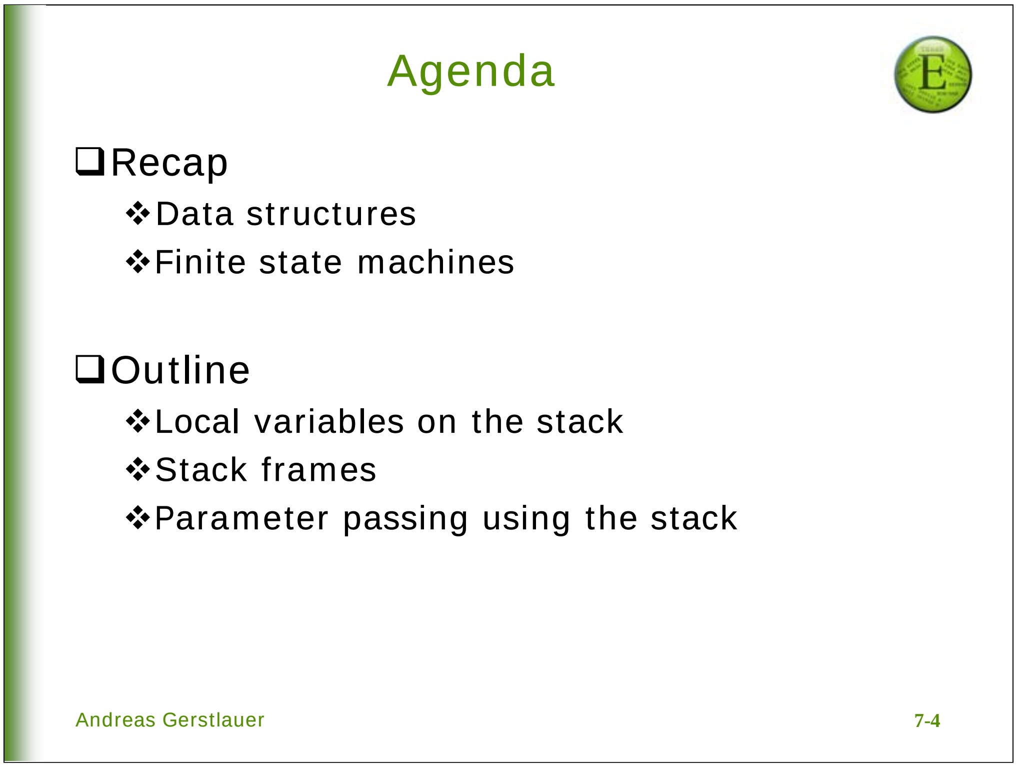
- ❖ 25% too fast, 70% ok, 5% too slow

❑ Learning rate

- ❖ 15% more, 66% expected, 15% less, 3% nothing

❑ Comments

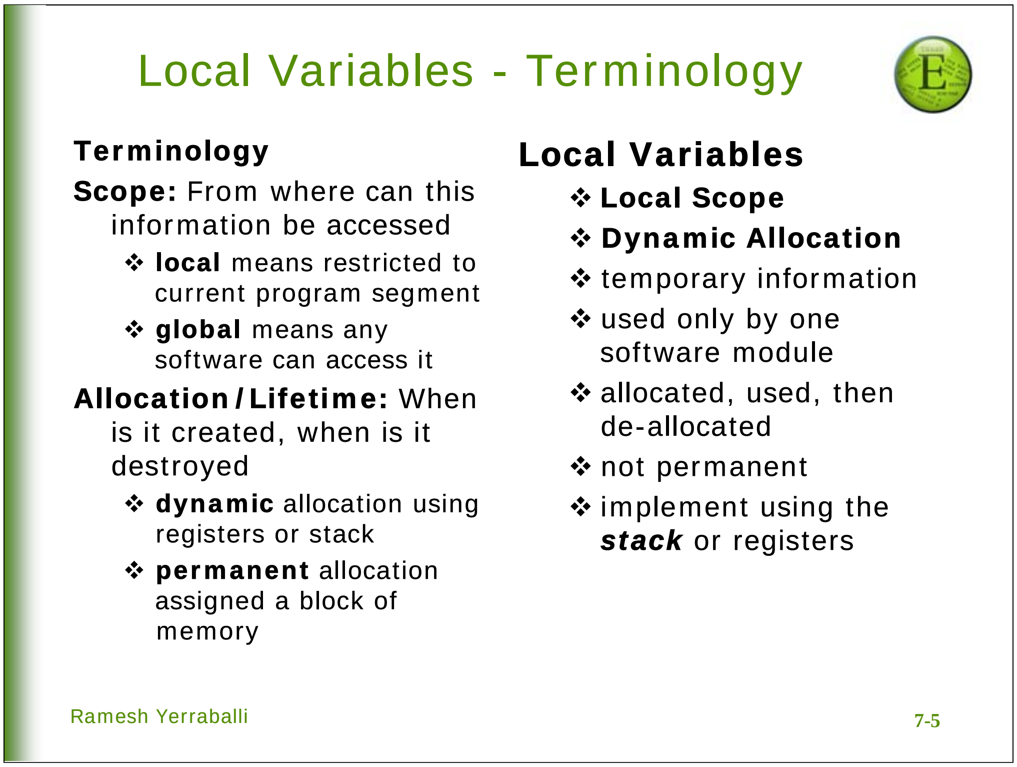
- ❖ C programming (vs. assembly), Codepad…
- ❖ Circuits, number wheel
- ❖ More examples, visualization, go over homeworks
- ❖ More interaction



Agenda

- ❑ Recap
  - ❖ Data structures
  - ❖ Finite state machines

- ❑ Outline
  - ❖ Local variables on the stack
  - ❖ Stack frames
  - ❖ Parameter passing using the stack



Local Variables - Terminology

Terminology

Scope: From where can this information be accessed

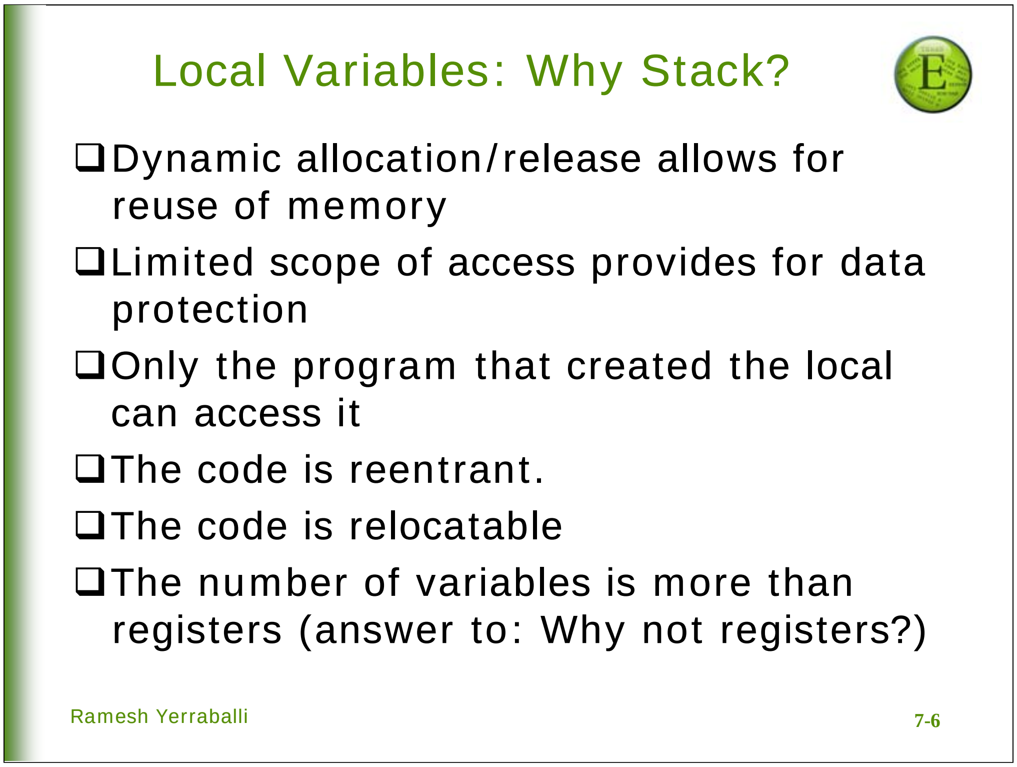
- **local** means restricted to current program segment
- **global** means any software can access it

Allocation/Lifetime: When is it created, when is it destroyed

- **dynamic** allocation using registers or stack
- **permanent** allocation assigned a block of memory

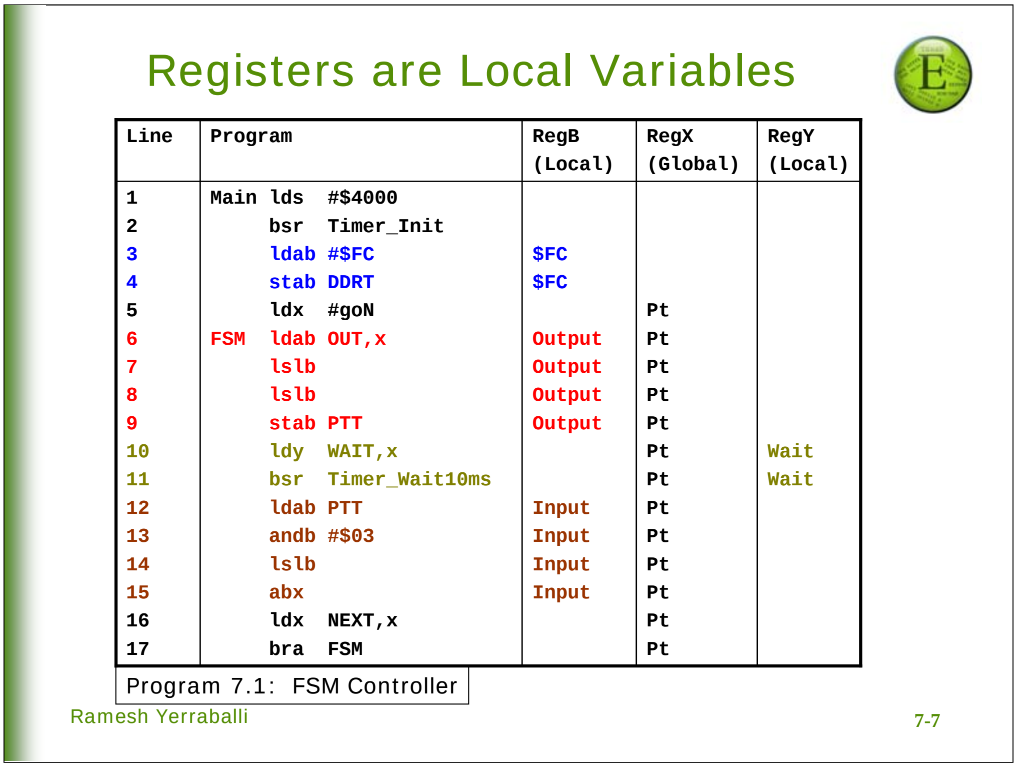
Local Variables

- **Local Scope**
- **Dynamic Allocation**
- temporary information
- used only by one software module
- allocated, used, then de-allocated
- not permanent
- implement using the ***stack*** or registers



Local Variables: Why Stack?

- ❑ Dynamic allocation/release allows for reuse of memory
- ❑ Limited scope of access provides for data protection
- ❑ Only the program that created the local can access it
- ❑ The code is reentrant.
- ❑ The code is relocatable
- ❑ The number of variables is more than registers (answer to: Why not registers?)

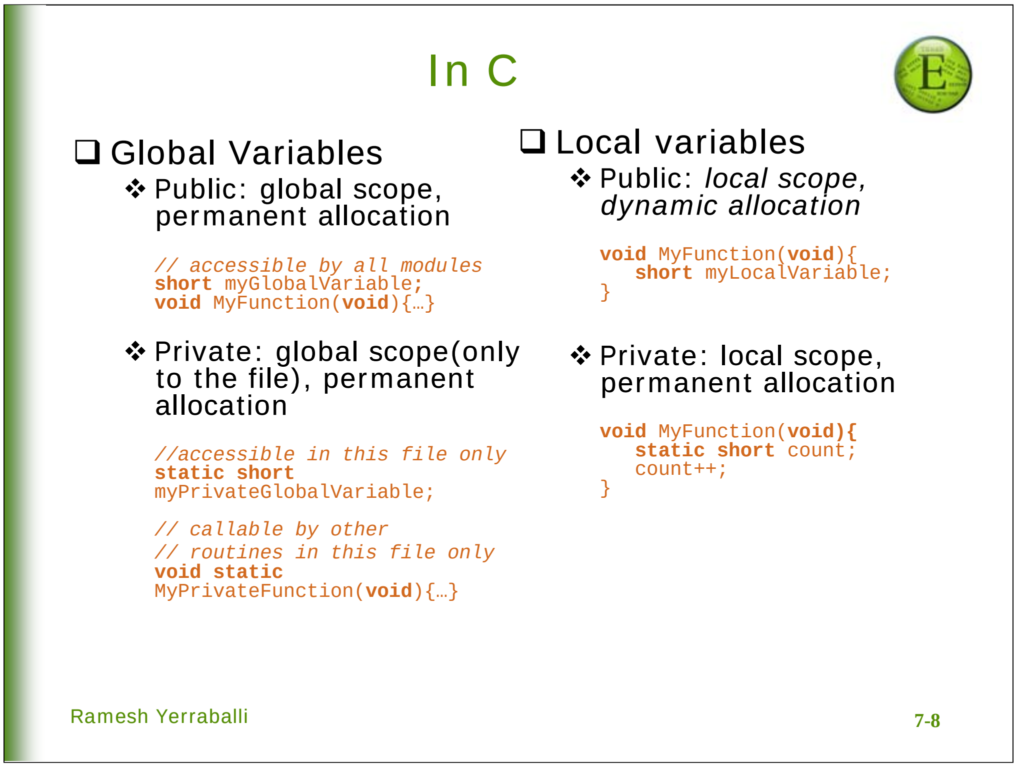


Registers are Local Variables

Line	Program	RegB (Local)	RegX (Global)	RegY (Local)
1	Main lds #$4000			
2	bsr Timer_Init			
3	ldab #$FC	$FC		
4	stab DDRT	$FC		
5	ldx #goN		Pt	
6	FSM ldab OUT,x	Output	Pt	
7	lslb	Output	Pt	
8	lslb	Output	Pt	
9	stab PTT	Output	Pt	
10	ldy WAIT,x		Pt	Wait
11	bsr Timer_Wait10ms		Pt	Wait
12	ldab PTT	Input	Pt	
13	andb #$03	Input	Pt	
14	lslb	Input	Pt	
15	abx	Input	Pt	
16	ldx NEXT,x		Pt	
17	bra FSM		Pt	

Program 7.1: FSM Controller

Ramesh Yerraballi



In C

- ❑ Global Variables
  - ❖ Public: global scope, permanent allocation

```
// accessible by all modules
short myGlobalVariable;
void MyFunction(void){…}
```

  - ❖ Private: global scope(only to the file), permanent allocation

```
//accessible in this file only
static short
myPrivateGlobalVariable;

// callable by other
// routines in this file only
void static
MyPrivateFunction(void){…}
```

- ❑ Local variables
  - ❖ Public: *local scope, dynamic allocation*

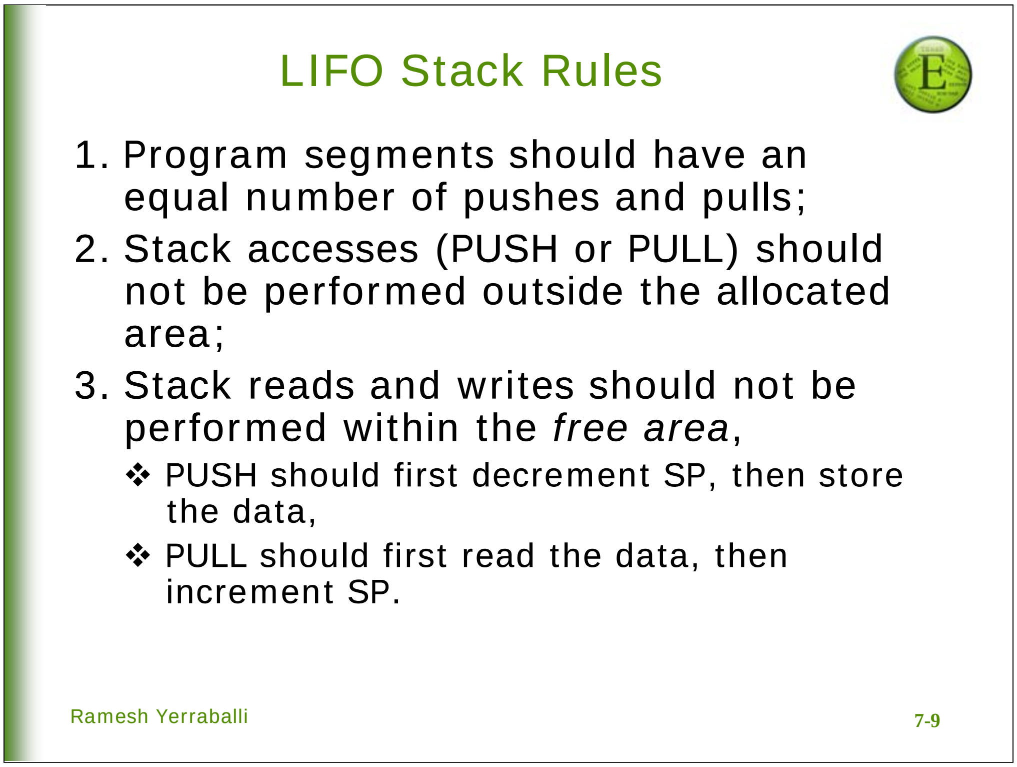
```
void MyFunction(void){
   short myLocalVariable;
}
```

  - ❖ Private: local scope, permanent allocation

```
void MyFunction(void){
   static short count;
   count++;
}
```

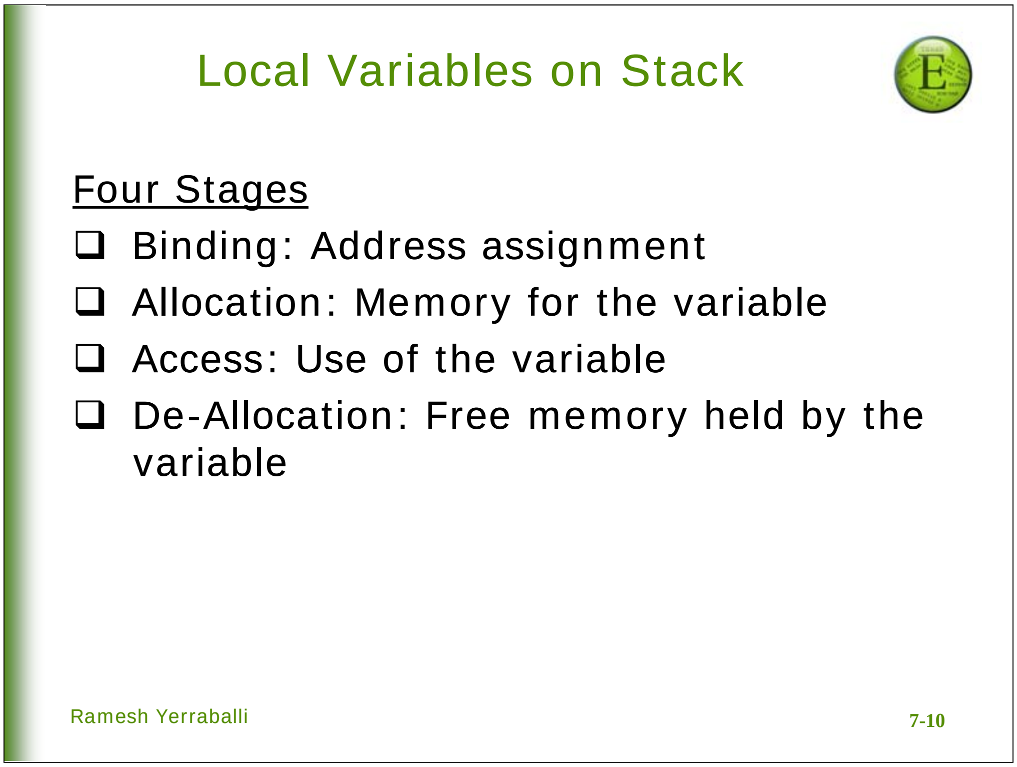
Ramesh Yerraballi

7-8



LIFO Stack Rules

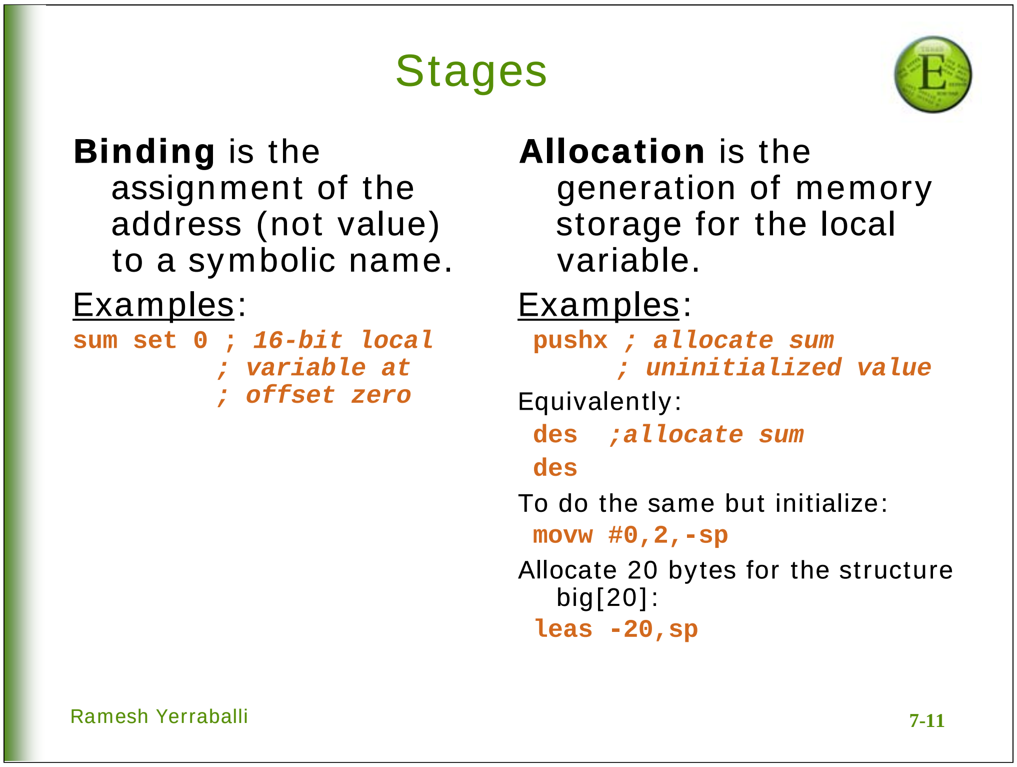
1. Program segments should have an equal number of pushes and pulls;
2. Stack accesses (PUSH or PULL) should not be performed outside the allocated area;
3. Stack reads and writes should not be performed within the *free area*,
   - ❖ PUSH should first decrement SP, then store the data,
   - ❖ PULL should first read the data, then increment SP.



Local Variables on Stack

<u>Four Stages</u>

- ❑ Binding: Address assignment
- ❑ Allocation: Memory for the variable
- ❑ Access: Use of the variable
- ❑ De-Allocation: Free memory held by the variable



Stages

Binding is the assignment of the address (not value) to a symbolic name.

Examples:

```
sum set 0 ; 16-bit local
          ; variable at
          ; offset zero
```

Allocation is the generation of memory storage for the local variable.

Examples:

```
pushx ; allocate sum
      ; uninitialized value
```

Equivalently:

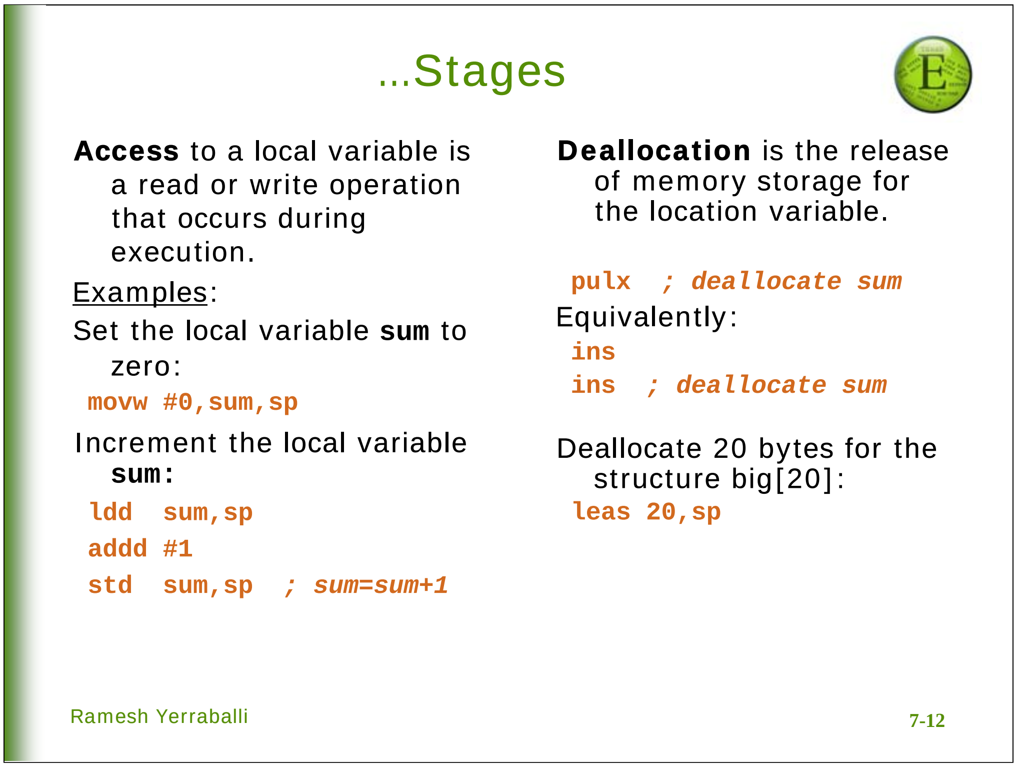
```
des  ;allocate sum
des
```

To do the same but initialize:

```
movw #0,2,-sp
```

Allocate 20 bytes for the structure big[20]:

```
leas -20,sp
```



...Stages

Access to a local variable is a read or write operation that occurs during execution.

Examples:

Set the local variable **sum** to zero:

```
movw #0,sum,sp
```

Increment the local variable **sum:**

```
ldd   sum,sp
addd #1
std   sum,sp   ; sum=sum+1
```

Deallocation is the release of memory storage for the location variable.

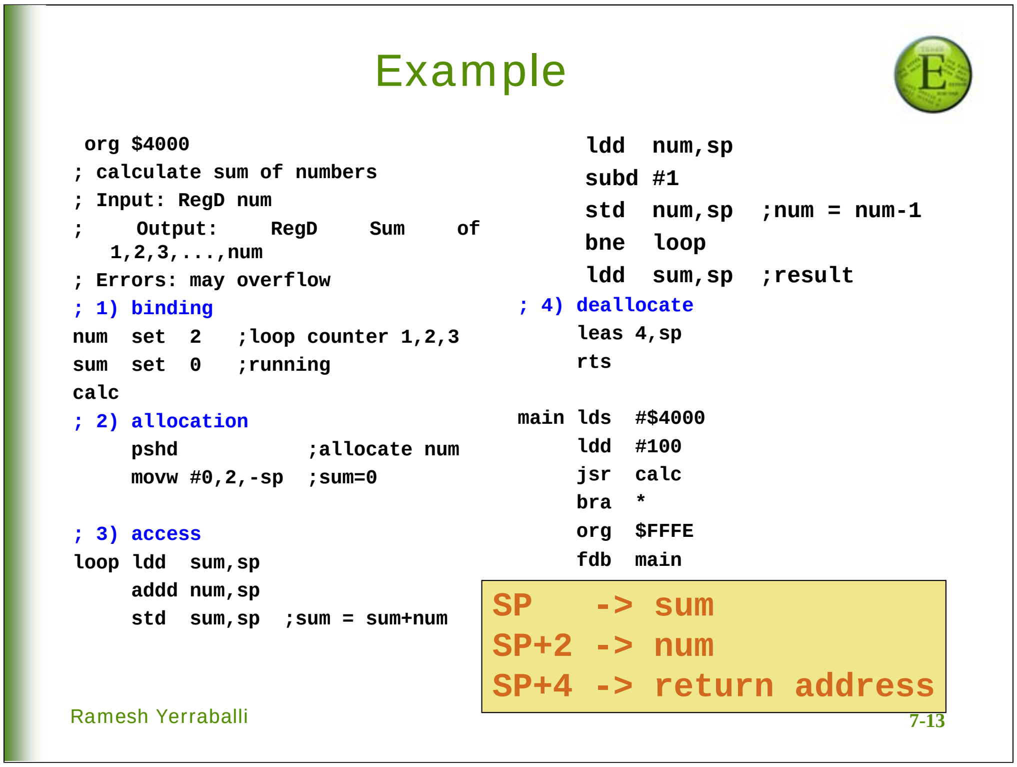
```
pulx   ; deallocate sum
```

Equivalently:

```
ins
ins   ; deallocate sum
```

Deallocate 20 bytes for the structure big[20]:

```
leas 20,sp
```



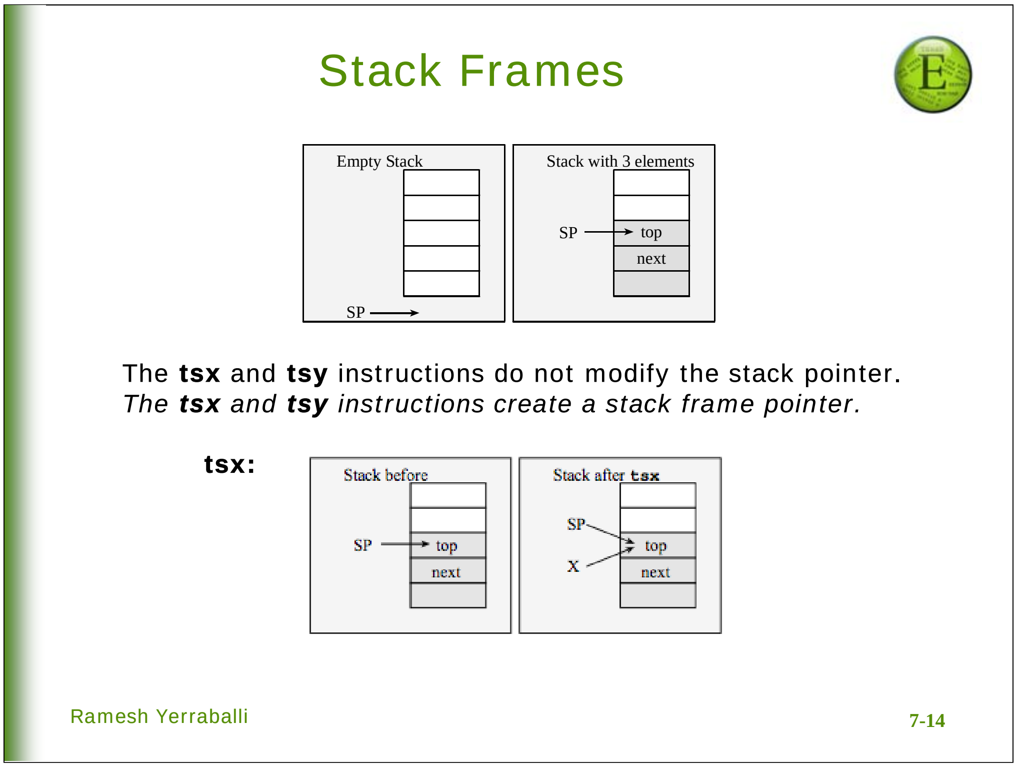
Example

```
 org $4000
; calculate sum of numbers
; Input: RegD num
;    Output:    RegD    Sum    of
   1,2,3,...,num
; Errors: may overflow
; 1) binding
num  set  2   ;loop counter 1,2,3
sum  set  0   ;running
calc
; 2) allocation
     pshd            ;allocate num
     movw #0,2,-sp   ;sum=0

; 3) access
loop ldd  sum,sp
     addd num,sp
     std  sum,sp  ;sum = sum+num
```

```
     ldd  num,sp
     subd #1
     std  num,sp  ;num = num-1
     bne  loop
     ldd  sum,sp  ;result
; 4) deallocate
     leas 4,sp
     rts

main lds  #$4000
     ldd  #100
     jsr  calc
     bra  *
     org  $FFFE
     fdb  main
```

```
SP   -> sum
SP+2 -> num
SP+4 -> return address
```

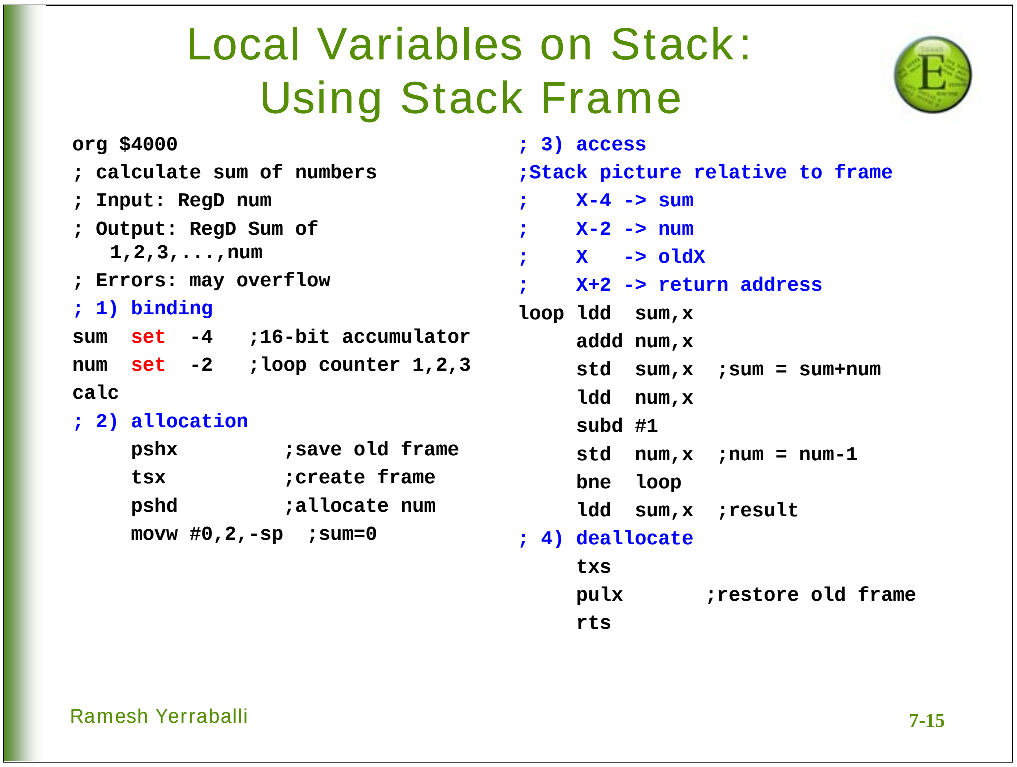
Ramesh Yerraballi



Stack Frames

The **tsx** and **tsy** instructions do not modify the stack pointer.
*The **tsx** and **tsy** instructions create a stack frame pointer.*

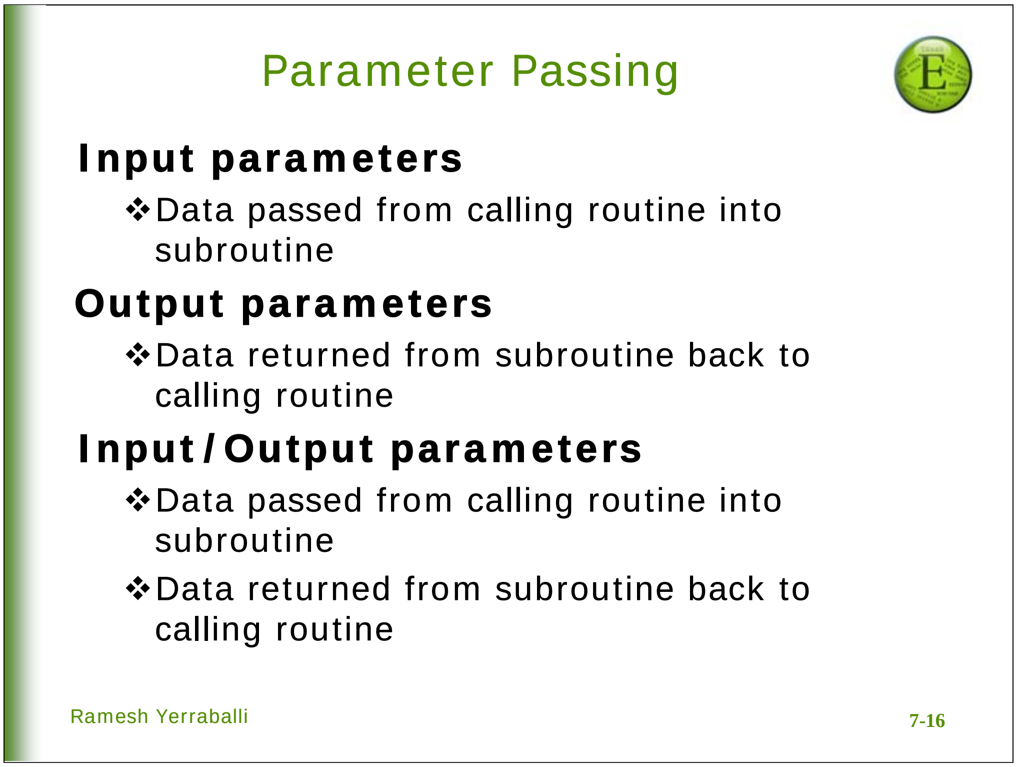
tsx:



Local Variables on Stack: Using Stack Frame

```
org $4000
; calculate sum of numbers
; Input: RegD num
; Output: RegD Sum of
   1,2,3,...,num
; Errors: may overflow
; 1) binding
sum  set  -4   ;16-bit accumulator
num  set  -2   ;loop counter 1,2,3
calc
; 2) allocation
     pshx         ;save old frame
     tsx          ;create frame
     pshd         ;allocate num
     movw #0,2,-sp  ;sum=0
```

```
; 3) access
;Stack picture relative to frame
;    X-4 -> sum
;    X-2 -> num
;    X   -> oldX
;    X+2 -> return address
loop ldd  sum,x
     addd num,x
     std  sum,x  ;sum = sum+num
     ldd  num,x
     subd #1
     std  num,x  ;num = num-1
     bne  loop
     ldd  sum,x  ;result
; 4) deallocate
     txs
     pulx       ;restore old frame
     rts
```



Parameter Passing

Input parameters

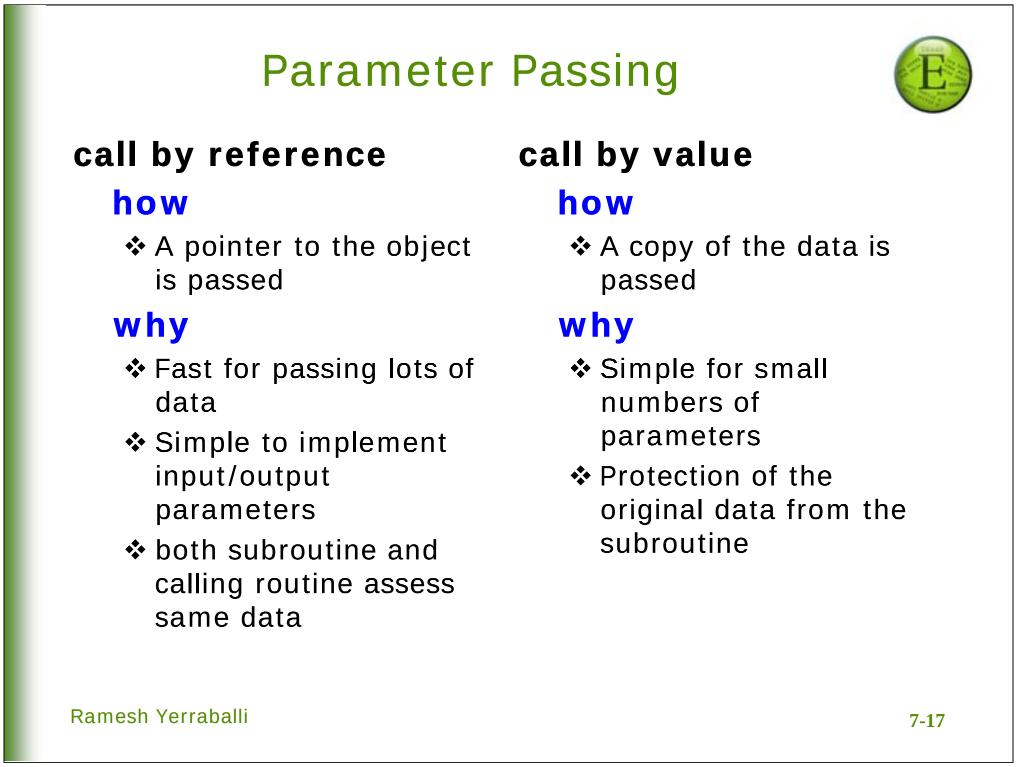
- ❖Data passed from calling routine into subroutine

Output parameters

- ❖Data returned from subroutine back to calling routine

Input/Output parameters

- ❖Data passed from calling routine into subroutine
- ❖Data returned from subroutine back to calling routine



Parameter Passing

call by reference

how

- A pointer to the object is passed

why

- Fast for passing lots of data
- Simple to implement input/output parameters
- both subroutine and calling routine assess same data

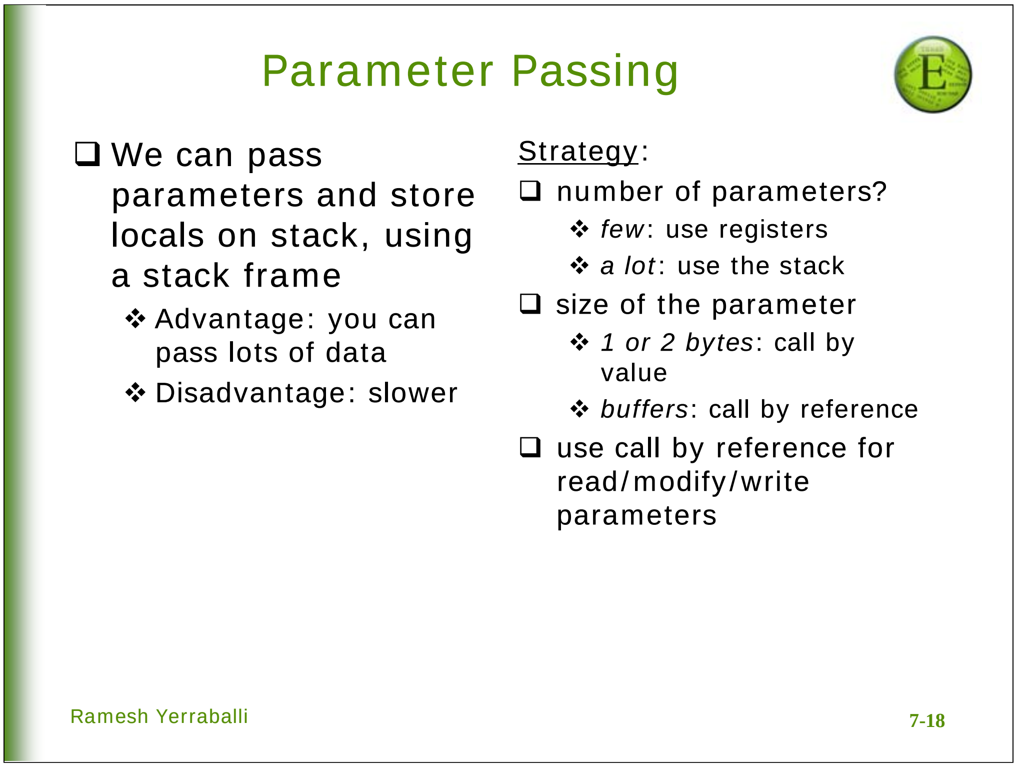
call by value

how

- A copy of the data is passed

why

- Simple for small numbers of parameters
- Protection of the original data from the subroutine



Parameter Passing

- ❑ We can pass parameters and store locals on stack, using a stack frame
  - ❖ Advantage: you can pass lots of data
  - ❖ Disadvantage: slower

Strategy:

- ❑ number of parameters?
  - ❖ *few*: use registers
  - ❖ *a lot*: use the stack
- ❑ size of the parameter
  - ❖ *1 or 2 bytes*: call by value
  - ❖ *buffers*: call by reference
- ❑ use call by reference for read/modify/write parameters