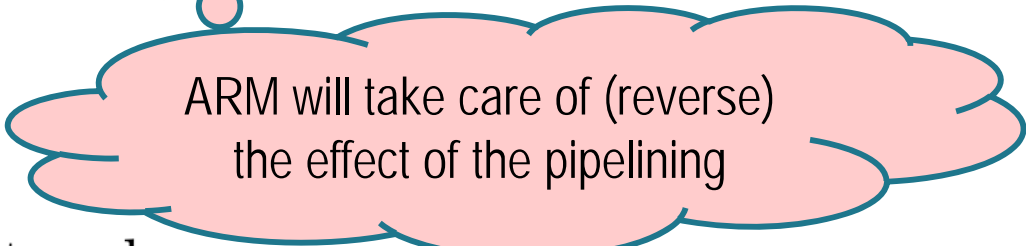


## ARM Support for Subroutines

- ❑ The *branch with link* instruction behaves like the branch instruction but the processor also copies the return address (i.e., the address of the next instruction to be executed following a return) into the link register **r14**.

- ❑ If you execute:

```
BL      Sub_A      ;branch to "Sub_A"  
                ;save return address in r14
```



ARM will take care of (reverse)  
the effect of the pipelining

- ❑ At the end of the subroutine you return by
  - *copying the return address* in r14 to the program counter by executing:

```
MOV pc,lr
```

or

```
MOV r15,r14
```

## ARM Support for Subroutines

- ❑ Suppose that you want to evaluate the following expression several times in a program.

**if  $x > 0$  then  $x = 16*x + 1$  else  $x = 32*x$**

Should it be LT  
or LE?

- ❑ Assuming that  $x$  is in  $r0$ , we can write :

```
Func1 CMP    r0,#0           ;test for x > 0
      MOVGT  r0,r0, LSL #4    ;if x > 0 x = 16*x
      ADDGT  r0,r0,#1         ;if x > 0 then x = 16*x + 1
      MOVLT  r0,r0, LSL #5    ;ELSE if x < 0 THEN x = 32*x
      MOV    pc,lr           ;return by restoring saved PC
```

- ❑ Consider the following invocation of the above subroutine.

```
LDR    r0,[r4]   ;get P
BL     Func1     ; First call
                ;P = (if P > 0 then 16*P + 1 else 32*P)
STR    r0,[r4]   ;save P
```

Later on ...

```
LDR    r0,[r5]   ;get Q
BL     Func1     ;Second call
                ;Q = (if Q > 0 then 16*Q + 1 else 32*Q)
STR    r0,[r5]   ; save Q
```

# ARM Support for Subroutines

```

01      AREA  BL_instruction, CODE, READWRITE
02      ENTRY
03
04      ADR    r4,P           ;register r4 points at P
05      ADR    r5,Q           ;register r5 points at Q
06
07      LDR    r0,[r4]        ; get P
08      BL     Func1          ; P = (if P > 0 then 16P + 1 else 32P)
09      STR    r0,[r4,#8]     ; save P
10      ;
11      ; some code
12      ;
13      LDR    r0,[r5]        ; get Q
14      BL     Func1          ; Q = (if Q > 0 then 16Q + 1 else 32Q)
15      STR    r0,[r5,#8]     ; save P
16
17      MOV    r0, #0x18       ; angel_SWIreason_ReportException
18      LDR    r1, =0x20026    ; ADP_Stopped_ApplicationExit
19      SVC    #0x123456       ; ARM semihosting (formerly SWI)
20
21
22      Func1  CMP    r0,#0     ;test for x > 0
23            MOVGT  r0,r0, LSL #4 ;if x > 0 x = 16x
24            ADDGT  r0,r0,#1    ;if x > 0 then x = 16x + 1
25            MOVLT  r0,r0, LSL #5 ;ELSE if x < 0 THEN x = 32x
26            MOV    pc,r14     ;return by restoring saved PC
27
28      AREA  BL_instruction, DATA, READWRITE
29      P
30      Q
31      DCD   0x00000003        ;P = 3
32      DCD   0xFFFFFFFF        ;Q = -1
33
34      SPACE 8

```

Register	Value
<b>Current</b>	
R0	0x00000018
R1	0x00020026
R2	0x00000000
R3	0x00000000
R4	0x00000044
R5	0x00000048
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x0000001C
R15 (PC)	0x00000028
CPSR	0xA00000D3
SPSR	0x00000000
User/System	
Fast Interrupt	
Interrupt	
Supervisor	
Abort	
Undefined	
Internal	
PC \$	0x00000028
Mode	Supervisor
States	36
Sec	0.00000000

Address	0x44
0x00000044:	00 00 00 03 FF FF FF FF
0x0000004C:	00 00 00 31 FF FF FF E0
0x00000054:	00 00 00 00 00 00 00 00

## Conditional Subroutine Calls

❑ **BL** instruction can be conditionally executed.

❑ **For example**

```
CMP r9,r4      ;if r9 < r4
```

```
BLLT ABC      ;then call subroutine ABC
```

❑ **BLLT** means

- **B**ranch
- with **L**ink
- execute on condition **L**ess **T**han

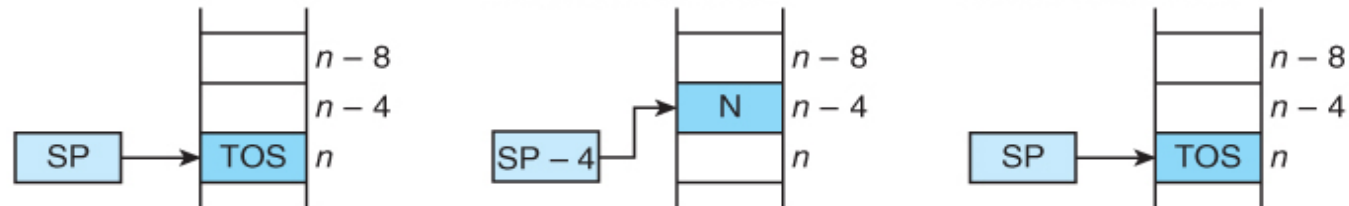
# The Stack

- ❑ The stack is a data structure, a *last in first out* queue, **LIFO**, in which items *enter at one end* and *leave from the same end* in a *reverse order*.
- ❑ Stacks in microprocessors are implemented by using a *stack pointer* to point to the *top of the stack (TOS)* in memory.
- ❑ As items are
  - added (*pushed*) to the stack, the stack pointer is moved *forward*, and
  - removed (*popped*) from the stack, the stack pointer is moved *backward*
- ❑ Figure 3.45 demonstrates four ways of constructing a stack.

# The Stack

Initial state of the stack

(a) Stack grows up.  
Stack pointer points to TOS.



PUSH:  $[SP] \leftarrow [SP] - 4$  ;Adjust the stack pointer  
 $[[SP]] \leftarrow \text{data}$  ;push data onto the stack

*Pre-update*

POP:  $\text{data} \leftarrow [[SP]]$  ;pull data off the stack  
 $[SP] \leftarrow [SP] + 4$  ;Adjust the stack pointer

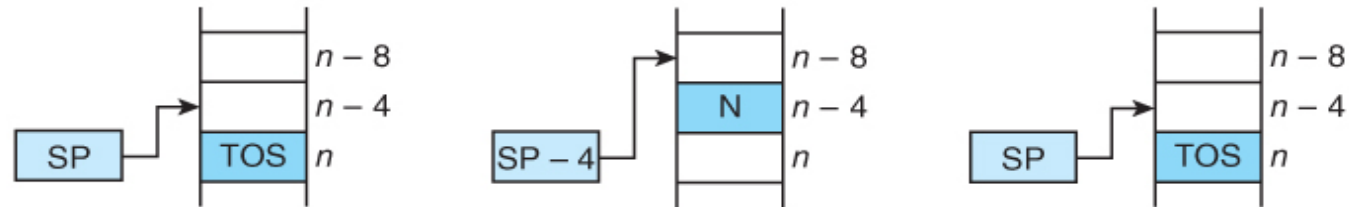
*Post-update*

**TOS** means *top of stack*

# The Stack

Initial state of the stack

(b) Stack grows up.  
Stack pointer points to first free space.



PUSH:  $[[SP]] \leftarrow \text{data}$  ;push data onto the stack  
 $[SP] \leftarrow [SP] - 4$  ;Adjust the stack pointer

*Post-update*

POP:  $[SP] \leftarrow [SP] + 4$  ;Adjust the stack pointer  
 $\text{data} \leftarrow [[SP]]$  ;pull data off the stack

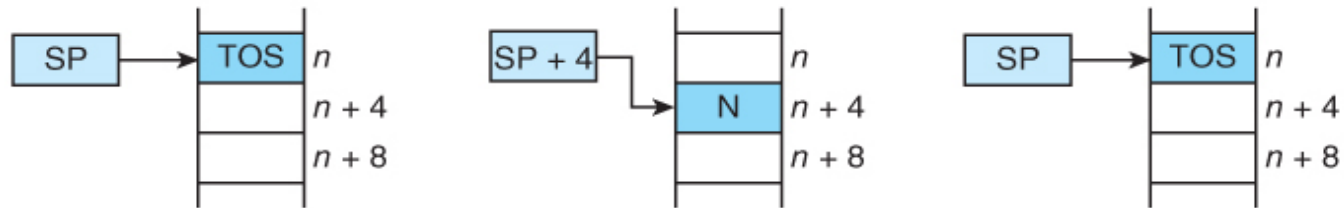
*Pre-update*

**TOS** means *top of stack*

# The Stack

Initial state of the stack

(c) Stack grows down.  
Stack pointer points to TOS.



PUSH: [SP]  $\leftarrow$  [SP] + 4 ;Adjust the stack pointer  
[[SP]]  $\leftarrow$  data ;push data onto the stack

*Pre-update*

POP: data  $\leftarrow$  [[SP]] ;pull data off the stack  
[SP]  $\leftarrow$  [SP] - 4 ;Adjust the stack pointer

*Post-update*

**TOS** means *top of stack*

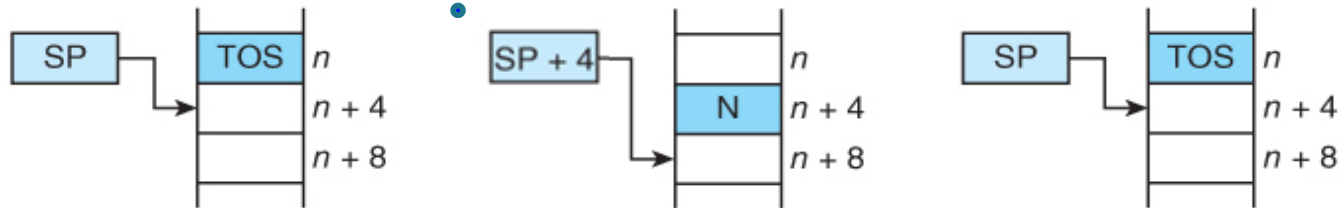


# The Stack

It is  $SP+4$ ,  
not  $SP+8$

Initial state of  
the stack

(d) Stack grows down.  
Stack pointer points  
to first free space.



PUSH:  $[[SP]] \leftarrow \text{data}$  ;push data onto the stack  
 $[SP] \leftarrow [SP] + 4$  ;Adjust the stack pointer

Post-update

POP:  $[SP] \leftarrow [SP] - 4$  ;Adjust the stack pointer  
 $\text{data} \leftarrow [[SP]]$  ;pull data off the stack

Pre-update

TOS means *top of stack*

# The Stack

- ❑ The *two design decisions* need to be made when implementing a stack are
  - whether the stack grows
    - *up toward low memory addresses* as items are pushed or
    - *down toward high memory addresses* as items are pushed.
  - whether the stack pointer points to
    - the *top item* on the stack or
    - the *first free empty space* on the stake.

# The Stack

- ❑ **CISC** processors maintain the stack automatically.
- ❑ **RISC** processors force the programmer to maintain the stack.

## Subroutine Call and Return

- ❑ An important application of the stack is to save return addresses after a subroutine call.
- A subroutine call can be implemented by
  - pushing the return address on the stack and then
  - jumping to the branch target address.
- Typically, this operation is implemented automatically by *BSR target* in **CISC** processor.
- Once the execution of the subroutine code is completed, a *return from subroutine* instruction is executed, i.e.,
  - the program counter to be restored to the point it was at after the *BSR Proc\_A* instruction had been fetched.

This is another method to implement a subroutine call, other than using R14.

## Subroutine Call and Return

- ❑ Because **ARM** does not implement *BSR* operations, you could synthesize this instruction by:

```

;assume that the stack grows towards
;low addresses and the SP points at
;the top item on the stack.
STR r15,[r13,#-4]! ;pre-decrement the stack pointer AND
;push the return address on the stack
B Target ;jump to the target address (B not BL)
... ;to return here
...
```

Due to the pipeline effect, the PC value will not be the address of the current instruction. Instead, it will be current address +12. Yes, it is +12, not +8, as it is STR instruction

- ❑ Because **ARM** does not support a stack-based subroutine return mechanism, you would have to write:

```

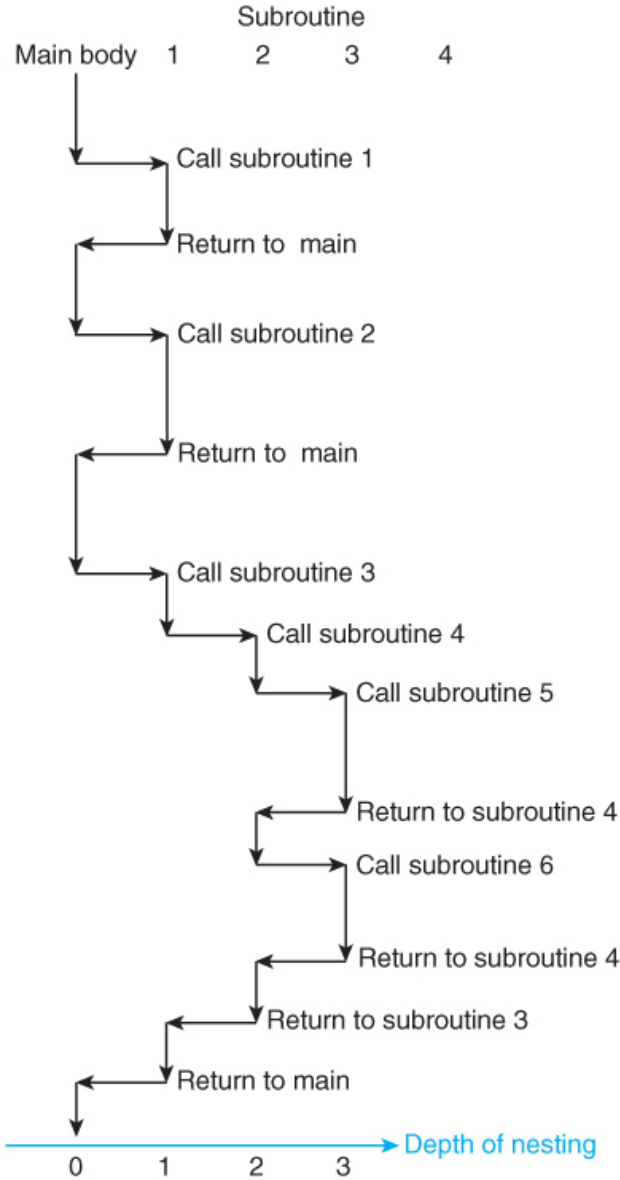
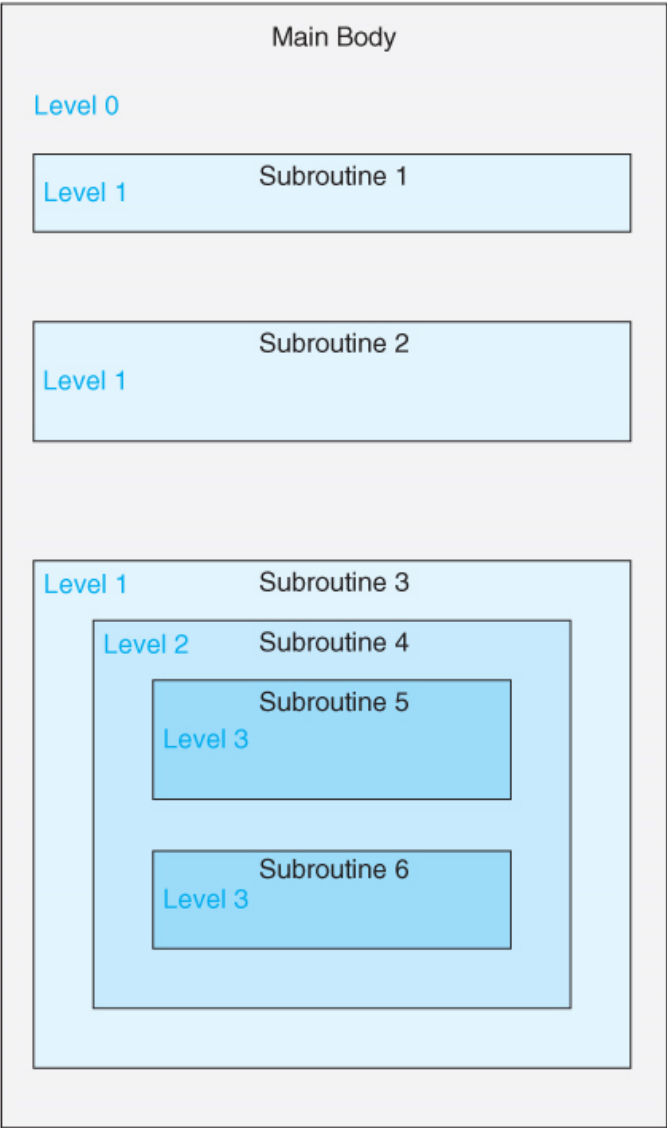
LDR r12,[r13],#4 ;get saved PC and post-increment
;stack pointer
SUB r15,r12,#4 ;fix PC and load into r15 to return
```

Why did we subtract 4?

Why did not we copy the stack content directory to r15?

# Nested subroutines

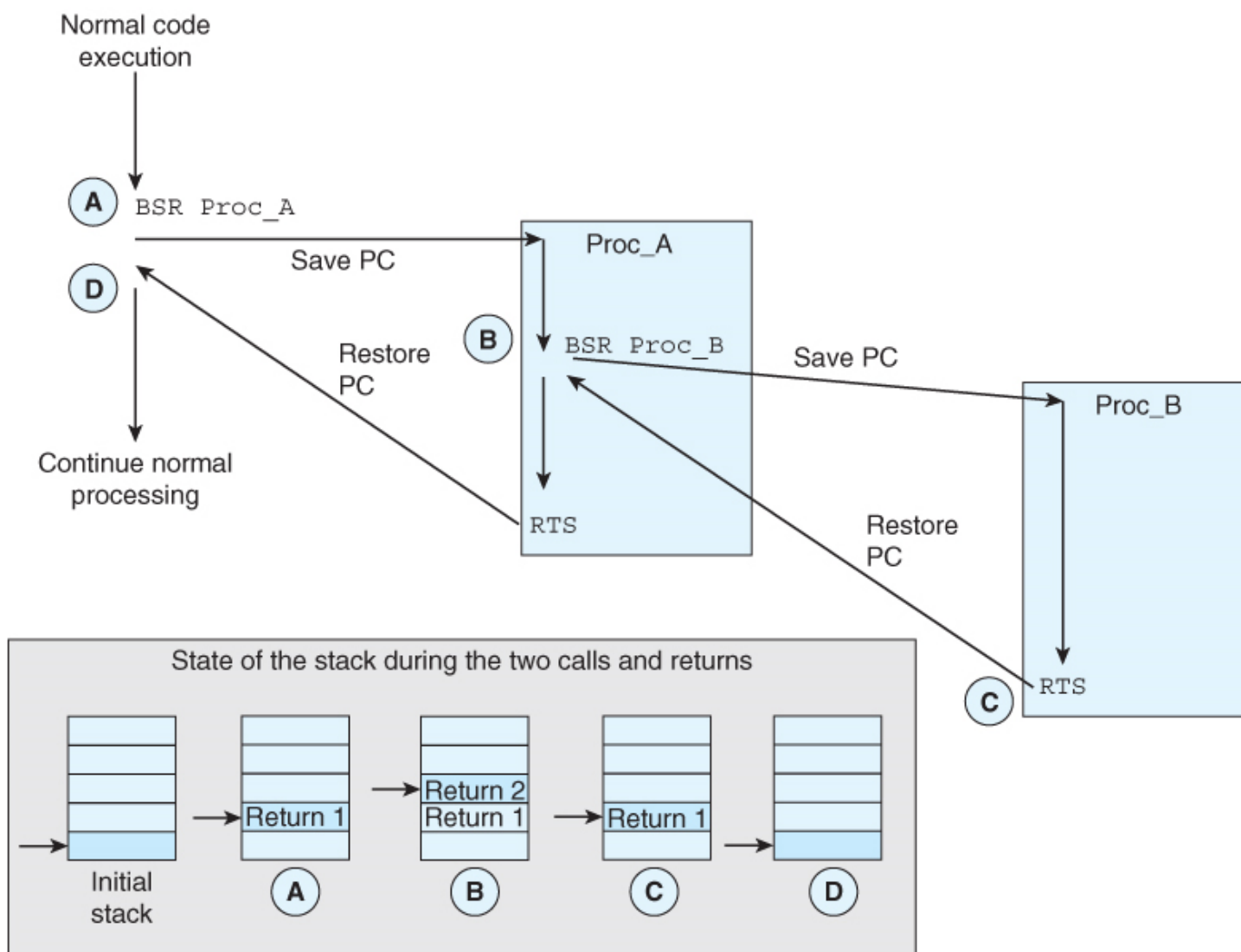
**FIGURE 3.48** An example of nested subroutines



© Cengage Learning 2014

## Example of nested subroutine

**FIGURE 3.49** The stack and nested subroutines (CISC processors)



## Leaf routines

- ❑ A *leaf routine* doesn't call another routine; it's at the end of the tree.
- ❑ If you call a *leaf routine* with **BL**,
  - the return address is saved in link register **r14**.
- ❑ A return to the calling point is made with a **MOV pc, lr**.
- ❑ If the routine is *not a leaf routine*, you cannot call another routine without first saving the link register.

```
ADR sp, STACK
```

```
BL Fun_1 ;call a simple leaf routine
```

```
BL Fun_2 ;call a routine that calls a nested routine
```

```
Loop B Loop
```

```
Fun_1 NOP ;this is a leaf routine
      MOV pc, lr ;return by copying the LR value into PC
```

```
Fun_2 NOP ;this is a non-leaf routine
      STR lr, [sp], #4 ;save link register
      BL Fun_1 ;call Fun_1 - overwrites the old LR
      LDR pc, [sp, #-4]! ;return by copying the LR value (from
                        ;the stack) into PC
```

```
STACK SPACE 0x10
```

What kind of stack is used here?

What is the maximum depth that can be called using this stack?



## Leaf routines

- ❑ Subroutine Fun\_1 is a leaf subroutine that does not call a nested subroutine and, therefore, we don't have to worry about saving the link register, **r14**, and we can return by executing `MOV pc, lr`.
- ❑ Subroutine Fun\_2 contains a call to a nested subroutine and we have to save the link register in order to return from Fun\_2.
- ❑ The simplest way of *saving* the link register is to *push* it on the stack.
- ❑ To return from Fun\_2, we *restore the pushed* **r14** into the program counter.

# Leaf routines

The screenshot shows the uVision4 IDE with the following components:

- Registers Panel:** Lists registers R0 through R15, CPSR, and SPSR. R13 (SP) is highlighted with a value of 0x00000000.
- Disassembly Panel:** Shows assembly code with comments:
 

```

3:      ADR sp,STACK
4:
0x00000000 E28FD020 ADD    R13,PC,#0x00000020
5:      BL  Fun_1      ;call a simple leaf routine
0x00000004 EB000001 BL     0x00000010
6:      BL  Fun_2      ;call a routine that calls a nested routine
0x00000008 EB000002 BL     0x00000018
7: Loop B  Loop
8: ;-----
0x0000000C EAffffFE B      0x0000000C
9: Fun_1 NOP           ;this is a leaf routine
0x00000010 E1A00000 NOP
10:      MOV pc,lr      ;return by moving the LR value into PC
      
```
- Source Panel (ex1.asm):** Shows the corresponding assembly source code with similar comments. Line 3, 'ADR sp,STACK', is highlighted in green.
- Command Panel:** Displays memory usage: '\*\*\* Restricted Version with 32768 Byte Cc' and '\*\*\* Currently used: 56 Bytes (0%)'.
- Memory Panel:** Shows a memory dump starting at address 0x0, with hex values and their corresponding assembly instructions (e.g., E2 8F D0 20 EB 00 00 01).

A red callout bubble with the text "How is the value to be stored in r13?" points to the instruction `ADD R13, PC, #0x00000020` in the disassembly panel.

# Leaf routines

The screenshot shows the uVision4 IDE with the following components:

- Registers Panel:** Shows the current state of registers. R13 (SP) is 0x00000028, R14 (LR) is 0x00000000, and R15 (PC) is 0x00000004.
- Disassembly Panel:** Shows the assembly code for the current address range. The code includes:
 

```

3:      ADR sp,STACK
4:
0x00000000 E28FD020 ADD    R13,PC,#0x00000020
5:      BL Fun_1          ;call a simple leaf routine
0x00000004 EB000001 BL     0x00000010
6:      BL Fun_2          ;call a routine that calls a nested routine
0x00000008 EB000002 BL     0x00000018
7: Loop B Loop
8: ;-----
0x0000000C EAfffffe B      0x0000000C
9: Fun_1 NOP              ;this is a leaf routine
0x00000010 E1A00000 NOP
10:     MOV pc,lr          ;return by copying the LR value into PC
      
```
- Source Panel (ex1.asm):** Shows the corresponding assembly source code:
 

```

3      ADR sp,STACK
4
5      BL Fun_1          ;call a simple leaf routine
6      BL Fun_2          ;call a routine that calls a nested routine
7 Loop B Loop
8 ;-----
9 Fun_1 NOP              ;this is a leaf routine
10     MOV pc,lr          ;return by copying the LR value into PC
11 ;-----
      
```
- Command Panel:** Displays the message: "Restricted Version with 32768 Byte Cc" and "Currently used: 56 Bytes (0%)".
- Memory Panel:** Shows the memory dump starting at address 0x0. The dump includes hexadecimal values and their corresponding ASCII representations.

A red callout bubble with the text "How this offset is encoded?" points to the BL instruction at address 0x00000004, which has an offset of 0x00000010.

# Leaf routines

The screenshot displays the uVision4 IDE interface with the following components:

- Registers:** A list of registers (R0-R15, CPSR, SPSR) with their current values. R14 (LR) is highlighted with a value of 0x00000008, and R15 (PC) is highlighted with a value of 0x00000010.
- Disassembly:** A window showing the disassembled code. The instruction at address 0x00000010 is highlighted in yellow: `0x00000010 E1A00000 NOP`. The comment for this instruction is `;this is a leaf routine`. The instruction at address 0x00000014 is `0x00000014 E1A0F00E MOV PC,R14`, with the comment `;return by copying the LR value into PC`.
- ex1.asm:** A window showing the source code for the assembly file. The instruction `9 Fun_1 NOP` is highlighted in green, with the comment `;this is a leaf routine`. The instruction `12 Fun_2 NOP` is also visible, with the comment `;this is a non-leaf routine`.
- Command:** A window showing a message: `*** Restricted Version with 32768 Byte Cc` and `*** Currently used: 56 Bytes (0%)`.
- Memory:** A window showing the memory dump starting at address 0x0. The memory dump shows the following values: `0x00000000: E2 8F D0 20 EB 00 00 01 EB 00 00 02 EA FF FF FE E1 A0 00 00`, `0x00000014: E1 A0 F0 0E E1 A0 00 00 E4 8D E0 04 EB FF FF FA E5 3D F0 04`, `0x00000028: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`, `0x0000003C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`, and `0x00000050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`.

# Leaf routines

The screenshot displays the uVision4 IDE interface for a project named 'ex1.uvproj'. The main window shows the disassembly of assembly code. The registers window on the left shows the current state of registers, with R15 (PC) highlighted at address 0x00000014. The disassembly window shows the following code:

```

5:      BL  Fun_1      ;call a simple leaf routine
0x00000004 EB000001 BL      0x00000010
6:      BL  Fun_2      ;call a routine that calls a nested routine
0x00000008 EB000002 BL      0x00000018
7:      Loop B      Loop
8:      ;-----
0x0000000C EAffffFE B      0x0000000C
9:      Fun_1 NOP      ;this is a leaf routine
0x00000010 E1A00000 NOP
10:     MOV pc,lr      ;return by copying the LR value into PC
11:     ;-----
0x00000014 E1A0F00E MOV      PC,R14
12:     Fun_2 NOP      ;this is a non-leaf routine

```

The instruction at address 0x00000014, `MOV PC, LR`, is highlighted in yellow. Below the disassembly window, the 'ex1.asm' source file is shown with the corresponding assembly code:

```

4
5      BL  Fun_1      ;call a simple leaf routine
6      BL  Fun_2      ;call a routine that calls a nested routine
7      Loop B      Loop
8      ;-----
9      Fun_1 NOP      ;this is a leaf routine
10     MOV pc,lr      ;return by copying the LR value into PC
11     ;-----
12     Fun_2 NOP      ;this is a non-leaf routine

```

The memory window at the bottom shows the hex dump of the code, with the instruction at address 0x00000014 highlighted in green:

```

0x00000000: E2 8F D0 20 EB 00 00 01 EB 00 00 02 EA FF FF FE E1 A0 00 00
0x00000014: E1 A0 F0 0E E1 A0 00 00 E4 8D E0 04 EB FF FF FA E5 3D F0 04
0x00000028: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000003C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00000050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

The command window at the bottom left shows the status: '\*\*\* Restricted Version with 32768 Byte Cc' and '\*\*\* Currently used: 56 Bytes (0%)'. The bottom status bar indicates 'Simulation' and 't1: 0.00000000 s'.



# Leaf routines

The screenshot shows the uVision4 IDE with the following components:

- Registers Panel:** Shows the current state of registers. R15 (PC) is highlighted with a value of 0x00000008.
- Disassembly Panel:** Shows the assembly code. Line 6 is highlighted: `0x00000008 EB000002 BL 0x00000018`. A red callout bubble points to the offset 0x00000018, asking "How this offset is encoded?".
- Source Panel (ex1.asm):** Shows the corresponding assembly source code. Line 6 is highlighted: `BL Fun_2 ;call a routine that calls a nested routine`.
- Command Panel:** Shows the command prompt with the text: `*** Restricted Version with 32768 Byte Cc` and `*** Currently used: 56 Bytes (0%)`.
- Memory Panel:** Shows the memory dump starting at address 0x0. The first two lines are: `0x00000000: E2 8F D0 20 EB 00 00 01 EB 00 00 02 EA FF FF FE E1 A0 00 00` and `0x00000014: E1 A0 F0 0E E1 A0 00 00 E4 8D E0 04 EB FF FF FA E5 3D F0 04`.

# Leaf routines

The screenshot displays the uVision4 IDE interface for an ARM project. The main window shows the disassembly of the code, with the following instructions visible:

```

0x00000008 EB000002 BL      0x00000018
7: Loop B Loop
8: ;-----
0x0000000C EAffffff B      0x0000000C
9: Fun_1 NOP ;this is a leaf routine
0x00000010 E1A00000 NOP
10: MOV pc,lr ;return by copying the LR value into PC
11: ;-----
0x00000014 E1A0F00E MOV     PC,R14
12: Fun_2 NOP ;this is a non-leaf routine
0x00000018 E1A00000 NOP
13: STR lr,[sp],#4 ;save link register
0x0000001C E48DE004 STR     R14,[R13],#0x0004
  
```

The registers window on the left shows the current state of the registers, with R14 (LR) and R15 (PC) highlighted. The PC is at 0x00000018.

The source code window (ex1.asm) shows the corresponding assembly code:

```

6 BL Fun_2 ;call a routine that calls a nested routine
7 Loop B Loop
8 ;-----
9 Fun_1 NOP ;this is a leaf routine
10 MOV pc,lr ;return by copying the LR value into PC
11 ;-----
12 Fun_2 NOP ;this is a non-leaf routine
13 STR lr,[sp],#4 ;save link register
14 BL Fun_1 ;call Fun_1 - overwrites the old link register
15 LDR pc,[sp,#-4]! ;return by copying the LR value (from the stack) into
  
```

The command window at the bottom shows the status of the simulation, including the memory address 0x0 and the memory contents at 0x00000000, 0x00000014, and 0x00000028.

# Leaf routines

The screenshot shows the uVision4 IDE with the following components:

- Registers Panel:** Lists registers R0 through R15, CPSR, and SPSR. R15 (PC) is highlighted with a value of 0x0000001C.
- Disassembly Panel:** Shows assembly code for a routine. Line 13, `STR lr, [sp], #4`, is highlighted in yellow. A red callout bubble points to this line with the text "Which type of stack is it?".
- Source Panel (ex1.asm):** Shows the corresponding assembly source code. Line 13, `STR lr, [sp], #4`, is also highlighted in green.
- Command Panel:** Displays the command: `*** Restricted Version with 32768 Byte Cc` and `*** Currently used: 56 Bytes (0%)`.
- Memory Panel:** Shows memory addresses and their contents. Address 0x00000000 contains `E2 8F D0 20 EB 00 00 01 EB 00 00 02 EA FF FF FE E1 A0 00 00`. Address 0x00000014 contains `E1 A0 F0 0E E1 A0 00 00 E4 8D E0 04 EB FF FF FA E5 3D F0 04`. Address 0x00000028 contains `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`.



# Leaf routines

The screenshot shows the uVision4 IDE with the following components:

- Registers Panel:**

Register	Value
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x0000002C
R14 (LR)	0x0000000C
R15 (PC)	0x00000020
CPSR	0x000000D3
SPSR	0x00000000
- Disassembly Panel:**

```

0x00000010 E1A00000 NOP
10:      MOV pc,lr           ;return by copying the LR value into PC
11: ;-----
0x00000014 E1A0F00E MOV      PC,R14
12: Fun_2 NOP               ;this is a non-leaf routine
0x00000018 E1A00000 NOP
13:      STR lr,[sp],#4      ;save link register
0x0000001C E48DE004 STR      R14,[R13],#0x0004
14:      BL  Fun_1           ;call Fun_1 - overw
->0x00000020 EBFFFFFFA BL      0x00000010
15:      LDR pc,[sp,#-4]!    ;return by copying the LR value (from the stack) into
0x00000024 E53DF004 LDR      PC,[R13,#-0x0004]!
0x00000028 0000000C ANDEQ    R0,R0,R12
  
```
- Source Panel (ex1.asm):**

```

6      BL  Fun_2           ;call a routine that calls a nested routine
7 Loop B      Loop
8 ;-----
9 Fun_1 NOP               ;this is a leaf routine
10     MOV pc,lr           ;return by copying the LR value into PC
11 ;-----
12 Fun_2 NOP               ;this is a non-leaf routine
13     STR lr,[sp],#4      ;save link register
14     BL  Fun_1           ;call Fun_1 - overwrites the old link register
15     LDR pc,[sp,#-4]!    ;return by copying the LR value (from the stack) into
  
```
- Memory Panel:**

Address: 0x0

```

0x00000000: E2 8F D0 20 EB 00 00 01 EB 00 00 02 EA FF FF FE E1 A0 00 00
0x00000014: E1 A0 F0 0E E1 A0 00 00 E4 8D E0 04 EB FF FF FA E5 3D F0 04
0x00000028: 00 00 00 0C 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  
```

A red callout bubble with the text "How this offset is encoded?" points to the instruction `BL 0x00000010` in the Disassembly panel.

# Leaf routines

The screenshot displays the µVision4 IDE interface with the following components:

- Registers Window:**

Register	Value
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x0000002C
R14 (LR)	0x00000024
R15 (PC)	0x00000010
CPSR	0x000000D3
SPSR	0x00000000
- Disassembly Window:**

```

0x00000010 E1A00000 NOP
10:      MOV pc,lr      ;return by copying the LR value into PC
11:      ;-----
0x00000014 E1A0F00E MOV      PC,R14
12: Fun_2 NOP          ;this is a non-leaf routine
0x00000018 E1A00000 NOP
13:      STR lr,[sp],#4  ;save link register
0x0000001C E48DE004 STR      R14,[R13],#0x0004
14:      BL  Fun_1       ;call Fun_1 - overwrites the old link register
0x00000020 EBFFFFFFA BL      0x00000010
15:      LDR pc,[sp,#-4]! ;return by copying the LR value (from the stack) into
0x00000024 E53DF004 LDR      PC,[R13,#-0x0004]!
0x00000028 0000000C ANDEQ    R0,R0,R12
  
```
- ex1.asm Source Window:**

```

6      BL  Fun_2       ;call a routine that calls a nested routine
7 Loop B      Loop
8      ;-----
9 Fun_1 NOP          ;this is a leaf routine
10     MOV pc,lr      ;return by copying the LR value into PC
11     ;-----
12 Fun_2 NOP          ;this is a non-leaf routine
13     STR lr,[sp],#4  ;save link register
14     BL  Fun_1       ;call Fun_1 - overwrites the old link register
15     LDR pc,[sp,#-4]! ;return by copying the LR value (from the stack) into
  
```
- Command Window:**

```

*** Restricted Version with 32768 Byte Co
*** Currently used: 56 Bytes (0%)
  
```
- Memory Window:**

Address: 0x0

```

0x00000000: E2 8F D0 20 EB 00 00 01 EB 00 00 02 EA FF FF FE E1 A0 00 00
0x00000014: E1 A0 F0 0E E1 A0 00 00 E4 8D E0 04 EB FF FF FA E5 3D F0 04
0x00000028: 00 00 00 0C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  
```
- Bottom Panel:**

ASSIGN BreakDisable BreakEnable BreakKill

Simulation t1: 0.00000000 s

# Leaf routines

The screenshot displays the uVision4 IDE interface for a project named 'ex1.uvproj'. The main window shows the disassembly of the code, with the following instructions visible:

```

0x00000010 E1A00000 NOP
10:      MOV pc,lr      ;return by copying the LR value into PC
11: ;-----
0x00000014 E1A0F00E MOV    PC,R14
12: Fun_2 NOP          ;this is a non-leaf routine
0x00000018 E1A00000 NOP
13:      STR lr,[sp],#4  ;save link register
0x0000001C E48DE004 STR    R14,[R13],#0x0004
14:      BL  Fun_1        ;call Fun_1 - overwrites the old link register
0x00000020 EBFFFFFFA BL    0x00000010
15:      LDR pc,[sp,#-4]! ;return by copying the LR value (from the stack) into PC
0x00000024 E53DF004 LDR    PC,[R13,#-0x0004]!
0x00000028 0000000C ANDEQ  R0,R0,R12
  
```

The Registers window on the left shows the current state of the registers, with R15 (PC) highlighted at 0x00000014. The ex1.asm source window at the bottom shows the assembly code for Fun\_1 and Fun\_2, with Fun\_1 being a leaf routine and Fun\_2 being a non-leaf routine.

The Command window at the bottom left shows the status of the simulation, indicating a restricted version with 32768 Byte Code and currently used 56 Bytes (0%). The Memory window at the bottom right shows the memory address 0x0 and the corresponding memory contents.

# Leaf routines

The screenshot displays the uVision4 IDE with the following components:

- Registers Window:** Shows the current state of registers. R15 (PC) is highlighted with a value of 0x00000024.
- Disassembly Window:** Shows the assembly code being executed. The current instruction is `LDR PC, [R13, #-0x0004]!` at address 0x00000024.
- Source Window (ex1.asm):** Shows the corresponding assembly code. The current instruction is `LDR pc, [sp, #-4]!` at line 15.
- Command Window:** Displays the status of the simulation, including the restricted version and current usage.
- Memory Window:** Shows the memory contents at address 0x0, displaying hexadecimal values.

**Registers Window Data:**

Register	Value
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x0000002C
R14 (LR)	0x00000024
R15 (PC)	0x00000024
CPSR	0x000000D3
SPSR	0x00000000

**Disassembly Window Data:**

Address	Instruction	Comment
0x0000000C	EAF FFF E B	0x0000000C
9:	Fun_1 NOP	;this is a leaf routine
0x00000010	E1A00000	NOP
10:	MOV pc,lr	;return by copying the LR value into PC
11:	;	
0x00000014	E1A0F00E	MOV PC,R14
12:	Fun_2 NOP	;this is a non-leaf routine
0x00000018	E1A00000	NOP
13:	STR lr,[sp],#4	;save link register
0x0000001C	E48DE004	STR R14,[R13],#0x0004
14:	BL Fun_1	;call Fun_1 - overwrites the old link register
0x00000020	EBFF FFA	BL 0x00000010
15:	LDR pc,[sp,#-4]!	;return by copying the LR value (from the stack) into
0x00000024	E53DF004	LDR PC,[R13,#-0x0004]!

**Source Window Data (ex1.asm):**

```

10      MOV pc,lr      ;return by copying the LR value into PC
11      ;-----
12 Fun_2 NOP          ;this is a non-leaf routine
13      STR lr,[sp],#4 ;save link register
14      BL  Fun_1      ;call Fun_1 - overwrites the old link register
15      LDR pc,[sp,#-4]! ;return by copying the LR value (from the stack) into
16      ;-----
17      STACK SPACE 0x10
18      ;-----
  
```

**Command Window:**

```

*** Restricted Version with 32768 Byte Cc
*** Currently used: 56 Bytes (0%)
  
```

**Memory Window:**

Address	Value
0x00000000	E2 8F D0 20 EB 00 00 01 EB 00 00 02 EA FF FF FE E1 A0 00 00
0x00000014	E1 A0 F0 0E E1 A0 00 00 E4 8D E0 04 EB FF FF FA E5 3D F0 04
0x00000028	00 00 00 0C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

**Simulation Status:** Simulation, t1: 0.00000000 s



# Leaf routines

The screenshot shows the uVision4 IDE with the following components:

- Registers Panel:** Lists registers R0 through R15, CPSR, and SPSR. R13 (SP) is at 0x00000028, R14 (LR) is at 0x00000024, and R15 (PC) is at 0x0000000C.
- Disassembly Panel:** Shows assembly code for two routines:
  - Fun\_1 (Leaf Routine):**

```

9: Fun_1 NOP ;this is a leaf routine
10: MOV pc,lr ;return by copying the LR value into PC
11: ;-----
12: Fun_2 NOP ;this is a non-leaf routine
13: STR lr,[sp],#4 ;save link register
14: BL Fun_1 ;call Fun_1 - overwrites the old link register
15: LDR pc,[sp,#-4]! ;return by copying the LR value (from the stack) into PC
16: LDR PC,[R13,#-0x0004]!

```
  - Fun\_2 (Non-leaf Routine):**

```

6: BL Fun_2 ;call a routine that calls a nested routine
7: Loop B Loop
8: ;-----
9: Fun_1 NOP ;this is a leaf routine
10: MOV pc,lr ;return by copying the LR value into PC
11: ;-----
12: Fun_2 NOP ;this is a non-leaf routine
13: STR lr,[sp],#4 ;save link register
14: BL Fun_1 ;call Fun_1 - overwrites the old link register

```
- Memory Panel:** Shows memory addresses and their corresponding hex values:
  - 0x00000000: E2 8F D0 20 EB 00 00 01 EB 00 00 02 EA FF FF FE E1 A0 00 00
  - 0x00000014: E1 A0 F0 0E E1 A0 00 00 E4 8D E0 04 EB FF FF FA E5 3D F0 04
  - 0x00000028: 00 00 00 0C 00 00 00 00 00 00 00 00 00 00 00 00 00 00
- Command Panel:** Displays a message: "\*\*\* Restricted Version with 32768 Byte Code Memory \*\*\* Currently used: 56 Bytes (0%)".
- Simulation Status:** Shows "Simulation" and "t1: 0.00000000 sec".

# Leaf routines

The screenshot displays the uVision4 IDE interface. The main window shows the disassembly of an ARM binary file. The left pane lists registers (R0-R15, CPSR, SPSR) and their current values. The right pane shows the disassembly of the binary, with instructions and their corresponding assembly code. The bottom pane shows the source code (ex1.asm) with comments explaining the instructions.

**Registers:**

Register	Value
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000028
R14 (LR)	0x00000024
R15 (PC)	0x0000000C
CPSR	0x000000D3
SPSR	0x00000000

**Disassembly:**

```

0x0000000C  EAfffffe B 0x0000000C
9: Fun_1 NOP ;this is a leaf routine
0x00000010  E1A00000 NOP
10: MOV pc,lr ;return by copying the LR value into PC
11: ;-----
0x00000014  E1A0F00E MOV PC,R14
12: Fun_2 NOP ;this is a non-leaf routine
0x00000018  E1A00000 NOP
13: STR lr,[sp],#4 ;save link register
0x0000001C  E48DE004 STR R14,[R13],#0x0004
14: BL Fun_1 ;call Fun_1 - overwrites the old link register
0x00000020  EBfffffa BL 0x00000010
15: LDR pc,[sp,#-4]! ;return by copying the LR value (from the stack) into PC
0x00000024  E53DF004 LDR PC,[R13,#-0x0004]!
  
```

**Source Code (ex1.asm):**

```

6 BL Fun_2 ;call a routine that calls a nested routine
7 Loop B Loop
8 ;-----
9 Fun_1 NOP ;this is a leaf routine
10 MOV pc,lr ;return by copying the LR value into PC
11 ;-----
12 Fun_2 NOP ;this is a non-leaf routine
13 STR lr,[sp],#4 ;save link register
14 BL Fun_1 ;call Fun_1 - overwrites the old link register
  
```

**Command Window:**

```

*** Restricted Version with 32768 Byte Code Memory
*** Currently used: 56 Bytes (0%)
>
ASSIGN BreakDisable BreakEnable BreakKill
  
```

**Memory Window:**

Address: 0x0

```

0x00000000: E2 8F D0 20 EB 00 00 01 EB 00 00 02 EA FF FF FE E1 A0 00 00
0x00000014: E1 A0 F0 0E E1 A0 00 00 E4 8D E0 04 EB FF FF FA E5 3D F0 04
0x00000028: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  
```

Simulation t1: 0.00000000 sec

## Data Organization and Endianism

- ❑ The way in which numbers are stored in memory is not a trivial matter.
  - It can lead to incompatibilities between microprocessor families that store data in different ways.

## Data Organization and Endianism

- ❑ *Bit numbering* can vary between processors.
- ❑ Figure 3.51a shows *right-to-left* numbering, with *the least-significant bit on the right*.
  - Microprocessors (e.g., Intel) number the bits of a word from the *least-significant bit (lsb)* which is bit 0, to the *most-significant bit (msb)* which is bit  $m - 1$
- ❑ Some microprocessors, (PowerPC) reverse this scheme, as illustrated in Figure 3.51(b).

**FIGURE 3.51**

Numbering the bits of a byte



(a) Bit numbering with the least-significant bit at the right



(b) Bit numbering with the least-significant bit at the left



# Data Organization and Endianism

- ❑ As well as the way in which we *organize the bits of a byte*, we have to consider the way in which we *organize the individual bytes of a word*.
- ❑ The figures below demonstrates that we can number the bytes of a word in two ways. We can either
  - Put the *most-significant byte* at the *lowest byte address* of the word (*big endian*), or
  - Put the *least-significant byte* at the *lowest byte address* of the word (*little endian*).

