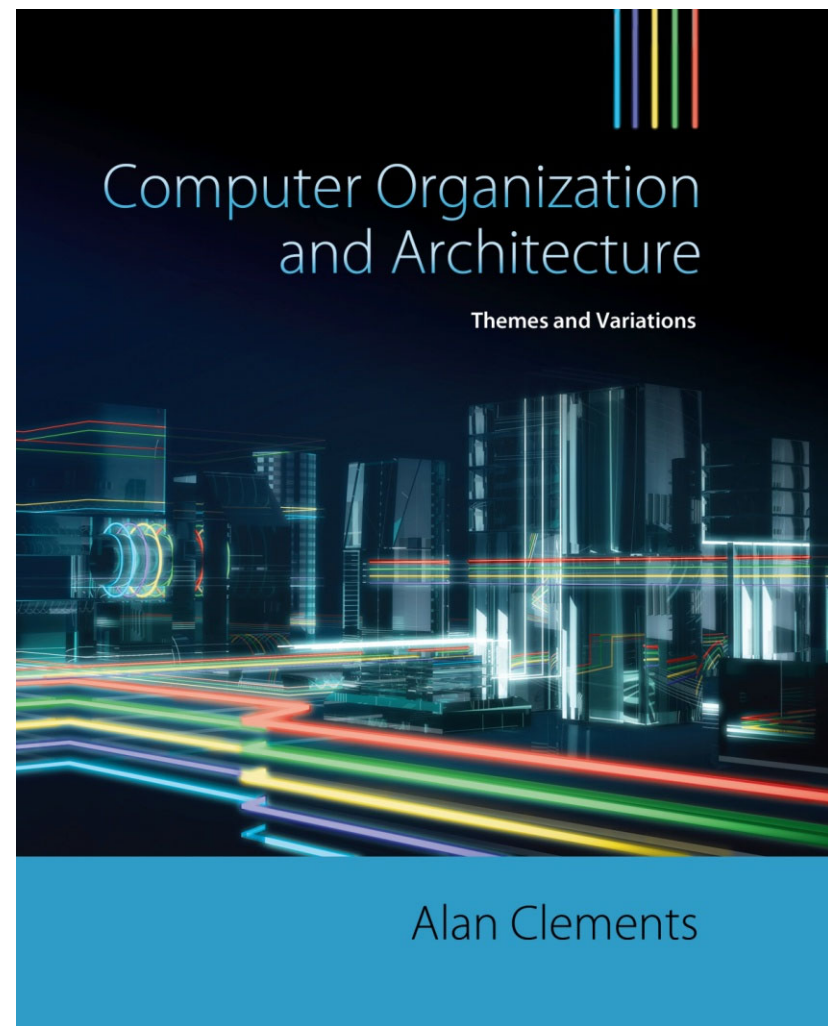


Part E

CHAPTER 3

Architecture and Organization

1



These slides are being provided with permission from the copyright for CS2208 use only. The slides must not be reproduced or provided to anyone outside of the class.

All download copies of the slides are for personal use only.
Students must destroy these copies within 30 days after receipt of final course evaluations.

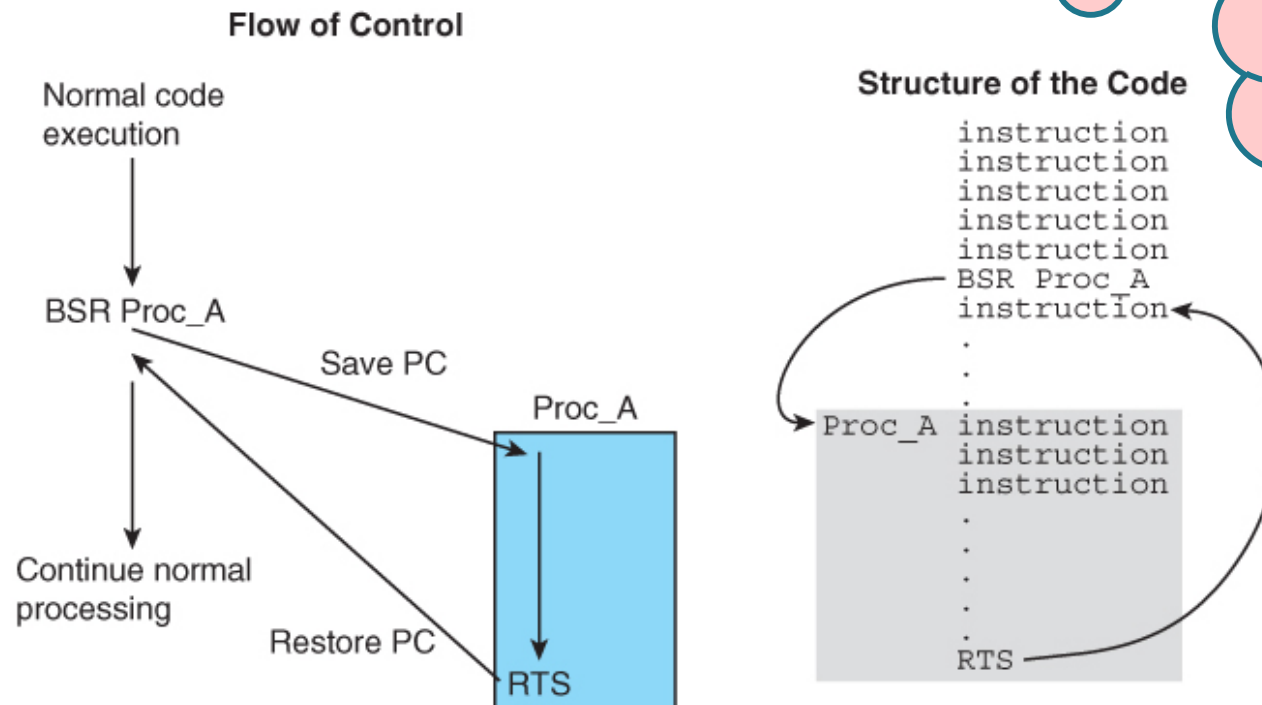
Subroutine Call and Return

- A *subroutine* (a.k.a. *function*, *procedure*, and *subprogram*) is *a set of instructions* that *may be repeatedly called* by a program to do a given function.
- A *subroutine* gives the simplest form of program abstraction.
- There are two main characteristics in any subroutine.
 1. A subroutine can be called from anywhere in the program.
 2. Once the subroutine is completed, it should return to the instruction directly after the subroutine calling location.
goes back to the main func

Subroutine Call and Return

- ❑ A *hypothetical* instruction *BSR Proc_A* calls subroutine *Proc_A*.
 - The processor saves the address of the next instruction to be executed in a safe place, and
 - loads the program counter with the address of the first instruction in the subroutine.
- ❑ At the end of the subroutine a *return from subroutine instruction*, *RTS*,
 - causes the processor to return to the point immediately following the subroutine call.

FIGURE 3.40 The subroutine call and return



BSR and *RTS*
are not ARM
instructions

ARM Support for Subroutines

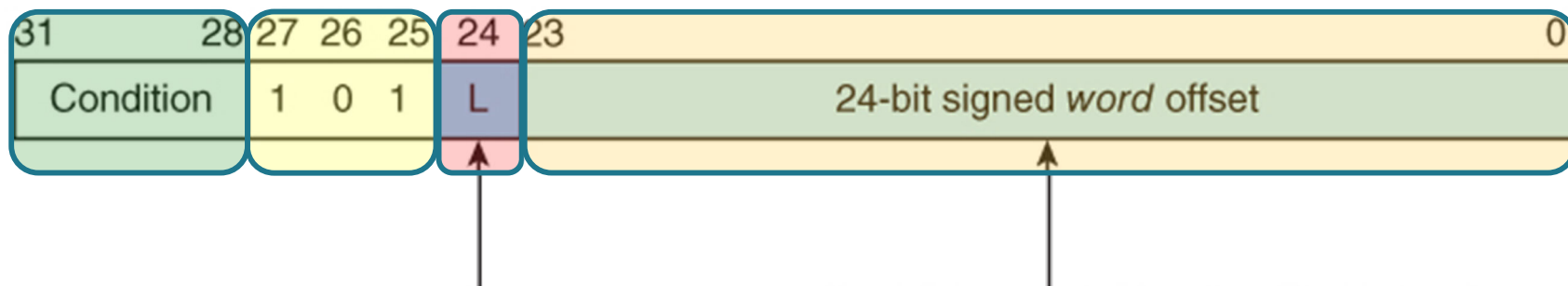
- ❑ **RISC** processors (including **ARM**) *do not provide* a *fully automatic* subroutine call/return mechanism like **CISC** processors.
So you have to manually add the return
- ❑ **ARM**'s *branch with link* instruction, **BL**,
 - automatically saves the return address in register **r14**.
- ❑ The branch instruction (Figure 3.41) has a 24-bit *signed* program counter relative offset (*word address offset*).

This is the main difference between B and BL

You may want to review slides 89 to 91 to remember how to encode and decode this 24-bit offset.

FIGURE 3.41

Encoding ARM's branch and branch-with-link instructions



The L-bit is 0 for a branch instruction and 1 for a branch with link instruction.

The 24-bit word offset is shifted left twice to create a 26-bit byte offset.

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

*Store the address
of the next instruction
in LR
To continue the program,
just read LR.*

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, r14
```

the executing instruction

or

```
MOV r15, r14
```

ARM Support for Subroutines

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

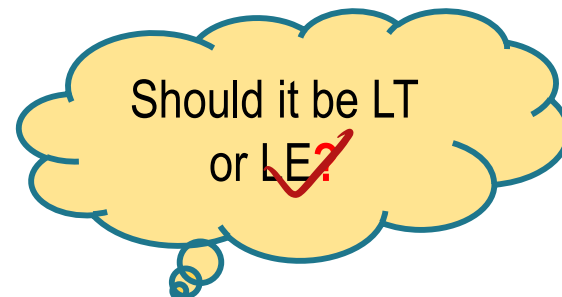
if $x > 0$ then $x = 16*x + 1$ else $x = 32*x$

update flags.

- ❑ Assuming that **x** is loaded into **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
```

= 16x *lr*



- ❑ 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      ; [r0]=r4+8
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      DCD    0x00000003 ;P = 3
30 Q      DCD    0xFFFFFFFF ;Q = -1
31
32 SPACE 8

```

Register	Value
Current	
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

$4 \times 16 + 4$
 68
 $68/4 = 17$

R13: stack pointer

R14: link register

when using B/BRL, return add will be store in R14.

B → branch to the target add, and not update R14.

BL → store the next instruction.

BLS → switch to smv.

BX → jump to the target add.

LDR: memory → register

STR: register → memory

MOV: register → register

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

Subroutine Call and Return

- ❑ An important application of the stack is to save the address to return to after executing the subroutine.
- A subroutine call can be implemented by
 - Pushing the return address onto the stack
 - Branching to the target address.
- Once the execution of the subroutine code is completed, a *return from subroutine* instruction is executed
 - Popping the return address from the ~~stack~~^{stack}
 - Copy the return address to the **PC** register

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

Occupied
memory

Subroutine Call and Return

Grows up

Example

This is B. It is NOT BL

...
...
...

STR r15, [r13, #-4] !

B Target

...
...

The proper return address

The address pushed onto the stake.

; assume that the stack grows towards
; low addresses and the SP points at
; the top item on the stack.
; pre-decrement the stack pointer AND
; push the return address on the stack
; 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

SUB r15, r12, #4

; get saved PC and post-increment

; stack pointer

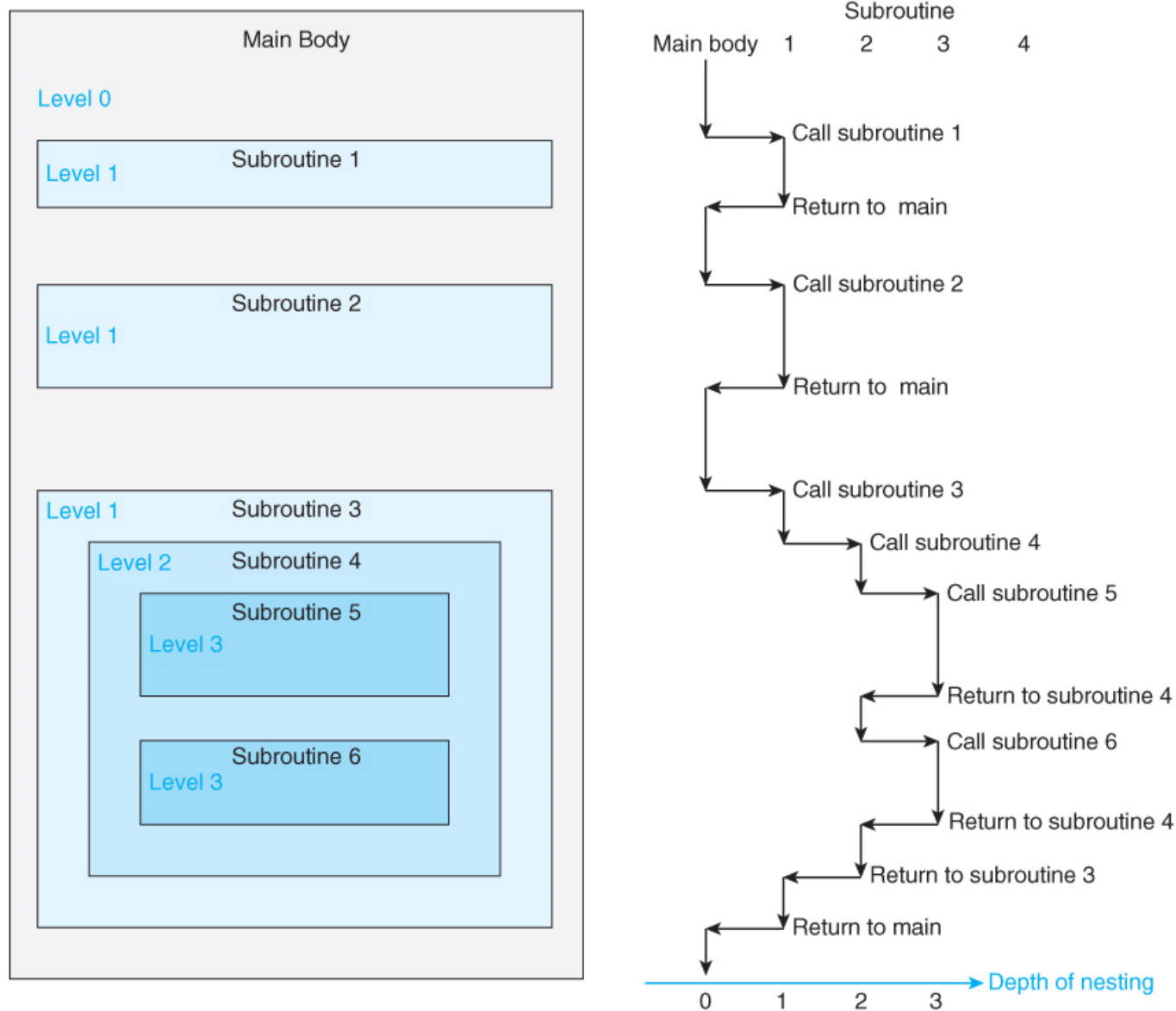
; fix PC and load into r15 to return

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

The 4 is subtracted to make the popped address pointing to the proper return address.

Nested subroutines

FIGURE 3.48 An example of nested subroutines

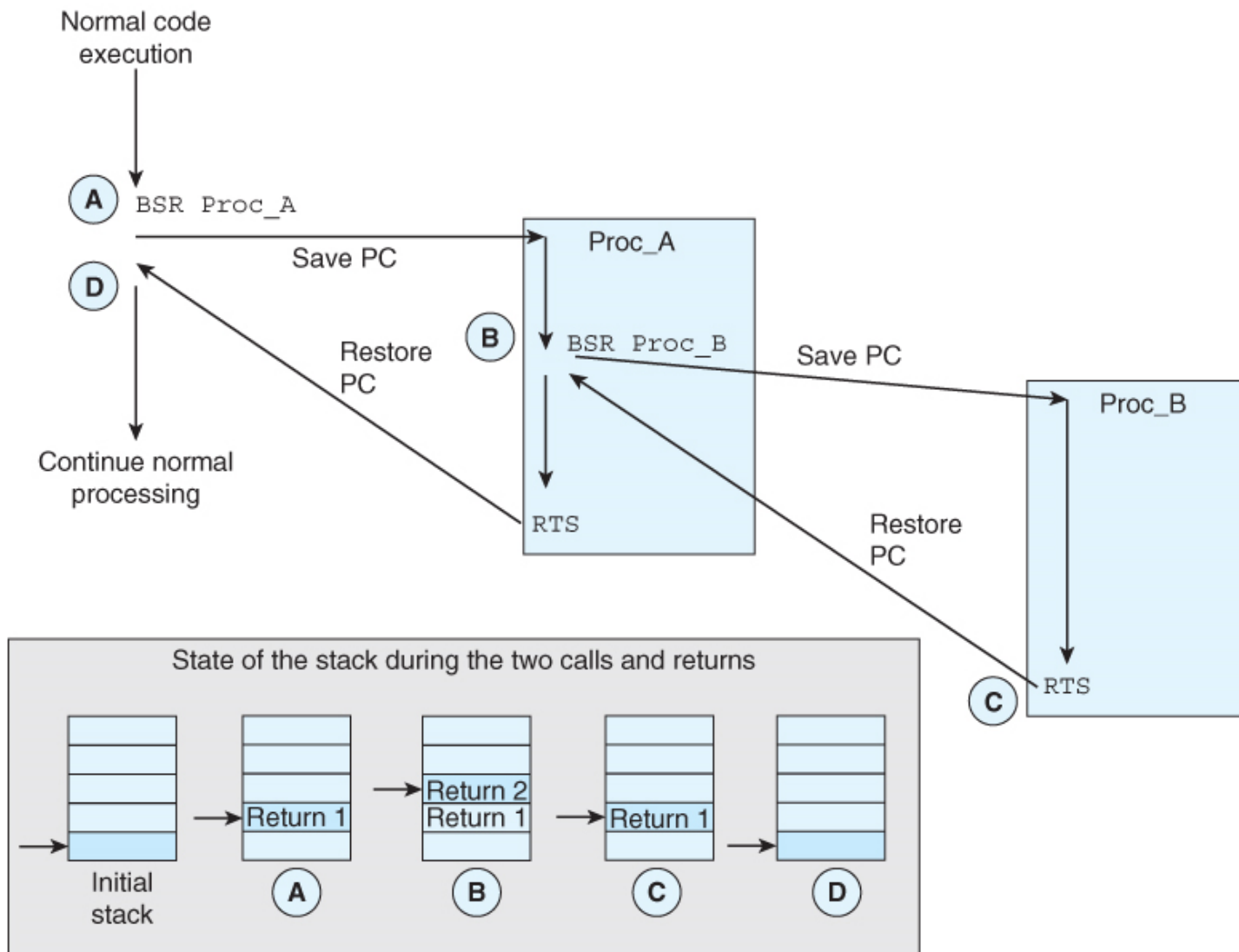


© Cengage Learning 2014

Occupied
memory

Example of nested subroutine

FIGURE 3.49 The stack and nested subroutines (CISC processors)



Empty
memory

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
```

LDm
EA →

```
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
```

return add next instruction
↓ to be read.

```
Fun_2 NOP ;this is a non-leaf routine
      STR lr, [sp], #4 ;save link register → save the address of
      BL Fun_1 ;call Fun_1 - overwrites the old LR return
      LDR pc, [sp, #-4]! ;return by copying the LR value (from r14)
                        ;the stack) into PC
```

push
lr to
the stack.

```
STACK SPACE 0x10
```


What kind of stack is used here?

200

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

2

Leaf routines

- ❑ Subroutine Fun_1 is a leaf subroutine that does not call any other 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 another subroutine (i.e., nested subroutine) and we have to save the link register in order to return from Fun_2.
`STR lr [sp], #4.`
- ❑ The simplest way of *saving* the link register is to push it onto the stack.

- ❑ To return from Fun_2, we *restore the pushed* **r14** into the program counter.
`LDR pc [sp, #-4]`

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 for a function. The instruction at address 0x00000000 is highlighted: `ADD R13, PC, #0x00000020`. Other instructions include `BL Fun_1`, `BL Fun_2`, `Loop B Loop`, `NOP`, and `MOV PC, R13`.
- Source Panel (ex1.asm):** Shows the corresponding assembly source code with comments: `AREA function_calls, CODE, READONLY`, `ENTRY`, `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`, `NOP ; this is a leaf routine`, and `MOV PC, R13 ; return by saving the LR value into PC`.
- 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, displaying hexadecimal values and their corresponding ASCII characters.

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

Leaf routines

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)	0x00000000
R15 (PC)	0x00000004
CPSR	0x000000D3
SPSR	0x00000000
User/System	
Fast Interrupt	
Interrupt	
Supervisor	

Disassembly

```

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

```

ex1.asm

```

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 ;-----

```

Memory 1

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 00
0x0000003C:	00 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 00

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

ASSIGN BreakDisable BreakEnable BreakKill

Simulation t1: 0.00000000 s

How is this offset encoded?

Leaf routines

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

- Registers Window:** Shows the current state of registers. R14 (LR) is highlighted with a value of 0x00000008. R15 (PC) has a value of 0x00000010.
- Disassembly Window:** Shows the disassembled code. Instruction 9 is highlighted: `0x00000010 E1A00000 NOP` with a comment `;this is a leaf routine`. Instruction 10 is `MOV pc,lr` with a comment `;return by copying the LR value into PC`.
- ex1.asm Source Window:** Shows the source assembly code. Line 9 is highlighted: `Fun_1 NOP ;this is a leaf routine`. Line 10 is `MOV pc,lr ;return by copying the LR value into PC`.
- Command Window:** Displays the message: `*** Restricted Version with 32768 Byte Cc` and `*** Currently used: 56 Bytes (0%)`.
- Memory Window:** Shows a memory dump starting at address 0x0. The first few lines of memory contain the instruction bytes for the routines.

Leaf routines

The screenshot displays the uVision4 IDE interface for an ARM project. 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. This instruction returns by copying the value in the LR register into the PC register.

The registers window shows the following values:

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)	0x00000008
R15 (PC)	0x00000014
CPSR	0x000000D3
SPSR	0x00000000

The memory window shows the address 0x00000014 containing the instruction `E1 A0 F0 0E`.

The command window shows the following text:

```

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

```

The status bar at the bottom indicates the simulation is running at time 0.00000000 seconds.

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 the value 0x00000008.
- Disassembly Panel:** Shows the assembly code. The instruction at address 0x00000008 is highlighted in yellow:


```
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 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
```
- Source Panel (ex1.asm):** Shows the corresponding assembly source code. The instruction at address 6 is highlighted in green:


```
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
```
- Memory Panel:** Shows the memory dump starting at address 0x0. The first two lines of memory are 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
```
- Command Panel:** Shows the command prompt with the text:


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

A red callout bubble with the text "How is this offset encoded?" points to the offset 0x00000018 in the assembly code.

Leaf routines

The screenshot shows the uVision4 IDE with the following components:

- Registers Panel:** Lists registers R0 through R15, CPSR, and SPSR. R14 (LR) is highlighted with a value of 0x0000000C, and R15 (PC) is highlighted with a value of 0x00000018.
- Disassembly Panel:** Shows assembly code with comments. Line 9 is highlighted in green: `0x0000000C EAfffffe B 0x0000000C ;this is a leaf routine`. Line 10 is highlighted in yellow: `0x00000010 E1A00000 NOP ;return by copying the LR value into PC`. Line 12 is highlighted in green: `0x00000014 E1A0F00E MOV PC,R14 ;this is a non-leaf routine`. Line 13 is highlighted in yellow: `0x00000018 E1A00000 NOP ;save link register`. Line 14 is highlighted in green: `0x0000001C E48DE004 STR R14,[R13],#0x0004`.
- Source Panel (ex1.asm):** Shows the corresponding assembly source code. Line 9 is highlighted in green: `9 Fun_1 NOP ;this is a leaf routine`. Line 10 is highlighted in yellow: `10 MOV pc,lr ;return by copying the LR value into PC`. Line 12 is highlighted in green: `12 Fun_2 NOP ;this is a non-leaf routine`. Line 13 is highlighted in yellow: `13 STR lr,[sp],#4 ;save link register`. Line 14 is highlighted in green: `14 BL Fun_1 ;call Fun_1 - overwrites the old link register`. Line 15 is highlighted in yellow: `15 LDR pc,[sp,#-4]! ;return by copying the LR value (from the stack) into`.
- Command Panel:** Shows the command `*** Restricted Version with 32768 Byte Cc` and `*** Currently used: 56 Bytes (0%)`.
- Memory Panel:** Shows memory addresses and their corresponding values. Address 0x0 is highlighted. The memory dump shows: `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`, and `0x00000028: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`.
- Bottom Panel:** Shows the status bar with `Simulation` and `tl: 0.00000000 s`.

Grows down

Empty
memory

Leaf routines

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)	0x0000000C
R15 (PC)	0x0000001C
CPSR	0x000000D3
SPSR	0x00000000
User/System	
Fast Interrupt	
Interrupt	
Supervisor	
Abort	

Disassembly

```
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
14: BL Fun_1 ;call Fun_1 - overwrites the old link register
```

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 1

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

Command

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

ASSIGN BreakDisable BreakEnable BreakKill

Simulation t1: 0.00000000 se

Which type of stack
is it?

Leaf routines

The screenshot shows the uVision4 IDE with the following components:

- Registers Window:** Shows the current state of registers. R13 (SP) is 0x0000002C, R14 (LR) is 0x0000000C, and R15 (PC) is 0x00000020.
- Disassembly Window:** Shows the assembly code. The instruction at address 0x00000020 is `BL 0x00000010`, which is highlighted in yellow. The comment for this instruction is `;return by copying the LR value (from the stack) into PC`.
- Source Window (ex1.asm):** Shows the assembly source code. The instruction `BL Fun_1` at address 0x00000020 is highlighted in green. The comment for this instruction is `;call Fun_1 - overwrites the old link register`.
- Memory Window:** Shows the memory contents. The address 0x00000020 is selected, and the memory value is `00 00 00 0C`, which is highlighted in blue. This value represents the offset for the BL instruction.

A red callout bubble with the text "How is this offset encoded?" points to the BL instruction in the Disassembly window. A blue arrow points from the R14 (LR) register value in the Registers window to the memory location 0x00000020 in the Memory window, indicating that the offset is the value stored in the link register.

Leaf routines

The screenshot displays the µVision4 IDE interface. The **Registers** window on the left shows the current state of registers: R0-R15, CPSR, SPSR, and Supervisor. R14 (LR) is highlighted with a value of 0x00000024, and R15 (PC) is 0x00000010.

The **Disassembly** window shows the following instructions:

```

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
  
```

The **ex1.asm** source window shows the following 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 shows the status of the simulation:

```

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

The **Memory** window shows the memory dump at 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 00
  
```

The **Simulation** status bar at the bottom shows the time: t1: 0.00000000 s.

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. R15 (PC) is highlighted with a value of 0x00000014.
- Disassembly:** A window showing the disassembled code for Fun_2. The instruction at address 0x00000014 is highlighted: `MOV PC, R14`. Comments indicate this is a non-leaf routine.
- ex1.asm:** A window showing the source assembly code. The instruction `MOV pc,lr` at address 10 is highlighted, corresponding to the disassembly.
- Command:** A window showing memory usage: `*** Restricted Version with 32768 Byte Cc` and `*** Currently used: 56 Bytes (0%)`.
- Memory:** A window showing the memory dump for address 0x0. The dump shows the instruction bytes for Fun_2.

The assembly code for Fun_2 is as follows:

```

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
  
```

The source code for ex1.asm is as follows:

```

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 memory dump shows the instruction bytes for Fun_2:

```

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
  
```

Leaf routines

The screenshot displays the uVision4 IDE interface for an ARM project. The main window shows the disassembly of the code, highlighting the leaf routine Fun_1 and the non-leaf routine Fun_2. The Registers window on the left shows the current state of the registers, with R15 (PC) at 0x00000024. The ex1.asm source window shows the assembly code for Fun_1 and Fun_2. The Command window at the bottom shows a restricted version with 32768 Byte Cc and 56 Bytes (0%) currently used. The Memory window shows the memory address 0x00000000 and its contents.

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)	0x0000002C
R14 (LR)	0x00000024
R15 (PC)	0x00000024
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
0x00000024 E53DF004 LDR PC,[R13,#-0x0004]!
  
```

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

```

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

Memory 1

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
  
```

Simulation t1: 0.00000000 s

Leaf routines

The screenshot shows the uVision4 IDE with the following components:

- Registers Window:** Lists registers R0 through R15. R13 (SP) is at 0x00000028, R14 (LR) is at 0x00000024, and R15 (PC) is at 0x0000000C. A blue arrow points from R15 to the memory window.
- Disassembly Window:** Shows assembly code for a leaf routine (Fun_1) and a non-leaf routine (Fun_2). Fun_1 consists of a NOP instruction followed by MOV pc, lr. Fun_2 consists of a NOP instruction, STR lr, [sp], #4, and BL Fun_1.
- Memory Window:** Shows memory addresses 0x00000000 through 0x00000028. The value at 0x00000028 is 00 00 00 0C, which is highlighted with a blue box and a blue arrow pointing from the R15 register.
- Command Window:** Displays the message: "Restricted Version with 32768 Byte Code Memory. Currently used: 56 Bytes (0%)."
- Simulation Status:** Shows "Simulation" and "t1: 0.00000000 s".

Leaf routines

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

- Registers Panel:** Shows the current state of 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 the disassembled code for the current address range.
 - Address 0x0000000C: `EAF FFF FE B 0x0000000C` (NOP) - Comment: `;this is a leaf routine`
 - Address 0x00000010: `E1A 00 00 0 NOP` (NOP)
 - Address 0x00000014: `E1A 0F 00 E MOV PC,R14` (NOP) - Comment: `;this is a non-leaf routine`
 - Address 0x00000018: `E1A 00 00 0 NOP` (NOP)
 - Address 0x0000001C: `E48 DE 0 0 4 STR R14,[R13],#0x0004` (BL Fun_1) - Comment: `;call Fun_1 - overwrites the old link register`
 - Address 0x00000020: `EB FFF F FA BL 0x00000010` (LDR pc,[sp,#-4]!) - Comment: `;return by copying the LR value (from the stack) into PC`
 - Address 0x00000024: `E53 DF 0 0 4 LDR PC,[R13,#-0x0004]!`
- Source Panel (ex1.asm):** Shows the corresponding assembly source code.
 - Line 6: `BL Fun_2 ;call a routine that calls a nested routine`
 - Line 7: `Loop B Loop`
 - Line 9: `Fun_1 NOP ;this is a leaf routine`
 - Line 10: `MOV pc,lr ;return by copying the LR value into PC`
 - Line 12: `Fun_2 NOP ;this is a non-leaf routine`
 - Line 13: `STR lr,[sp],#4 ;save link register`
 - Line 14: `BL Fun_1 ;call Fun_1 - overwrites the old link register`
- Command Panel:** Displays memory usage: `*** Restricted Version with 32768 Byte Code Space` and `*** Currently used: 56 Bytes (0%)`.
- Memory Panel:** Shows the memory dump starting at 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`
- Bottom Panel:** Includes a command line with `ASSIGN BreakDisable BreakEnable BreakKill`, a status bar showing `Simulation` and `t1: 0.00000000 sec`, and a page number `214`.

Subroutines and Block Move Instructions

- ❑ All subroutines commonly use the same set of registers to save values, and this might cause problems.
 - Assume that a program used **R1** to store a temporary value.
 - Later, this program called a function.
 - The function also used **R1** to store a different value.
 - After returning from the function, the program will not have access to the original **R1** value that was there before calling the function.

- ❑ To solve this issue, the followings need to be done:
 - At the beginning of the function, the values of all registers that will be used in the function must be pushed onto a stack.
 - Just before returning from the function, all pushed values must be popped and loaded to the same registers.

Subroutines and Block Move Instructions

- ❑ The **ARM**'s block move instructions can be used to
 - save register values once entering a subroutine and
 - restore registers just before returning from a subroutine.
- ❑ Consider the following ARM code:

<pre> BL test ... test STMFD r13!, {r0-r4, r10} . body of code . LDMFD r13!, {r0-r4, r10} MOV pc, r14 </pre>	<pre> ;call test, save return ;address in r14 ;subroutine test, save working ;registers ;subroutine completes, ;restore the registers ;copy the return address in ;r14 to the PC </pre>
--	---

store ST multiple full descending

load LD M F D

STMPD SP!, {R0~R7, LR}

$\leftarrow R0-R7,$
start-add = $SP - 9 \cdot 4$ 9 registers are push into the stack.

end-add = $SP - 4 \leftarrow$ top of the stack: 1 position.

LDMPD SP!, {R0~R7, LR}

start-add = SP

end-add = $SP + 9 \cdot 4$.

Subroutines and Block Move Instructions

- ❑ If you are using a block move to restore registers from the stack, you can also include the program counter.

We can write:

```
test STMFD r13!, {r0-r4, r10, r14} ;save working registers
                                     ;and return address in r14
:
LDMFD r13!, {r0-r4, r10, r15} ;restore working registers
                              ;and put r14 in the PC
```

- ❑ At the beginning of the subroutine, we push the *link register r14* containing the return address onto the stack, and then at the end we pull the saved registers, including the value of the return address which is placed in the *PC*, to make the return.
 - By doing so, we reduced the size of this code by one instruction