

Part E

CHAPTER 3

Architecture and Organization



1

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

All downloaded copies of the slides are for personal use only.

Students must destroy these copies within 30 days after receiving the course's final assessment.

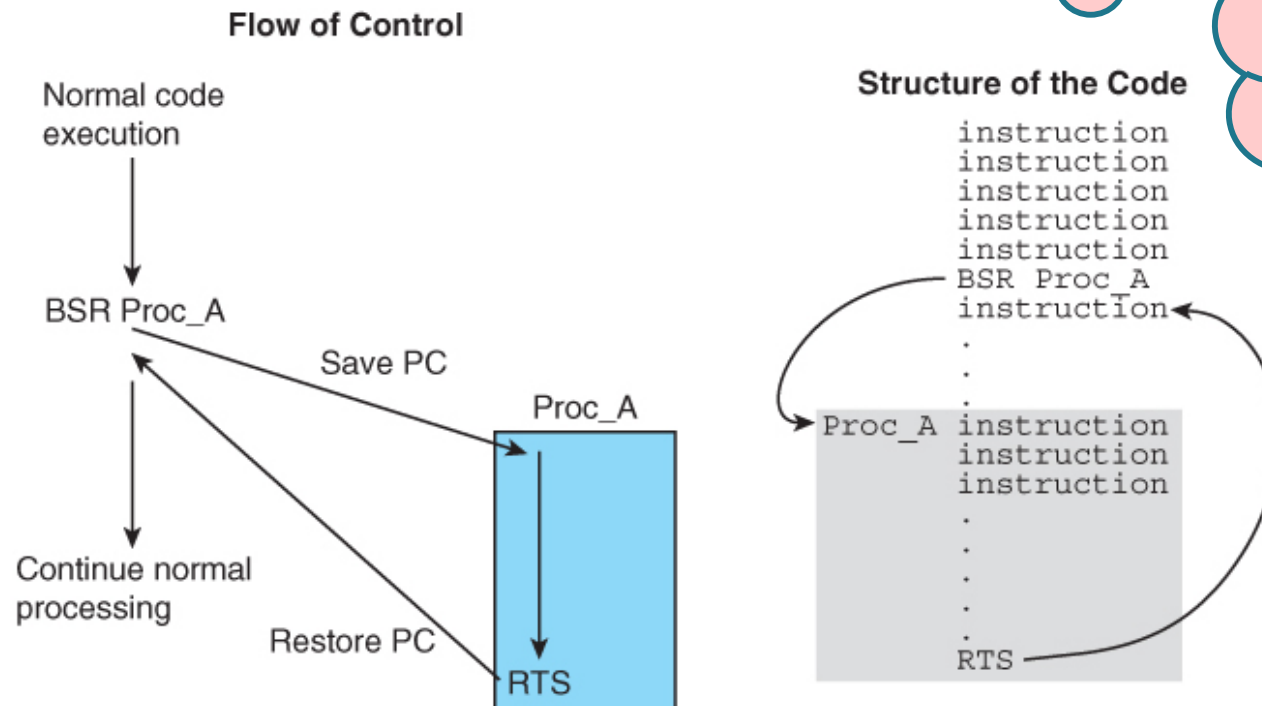
Subroutine Call and Return

- ❑ 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.

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

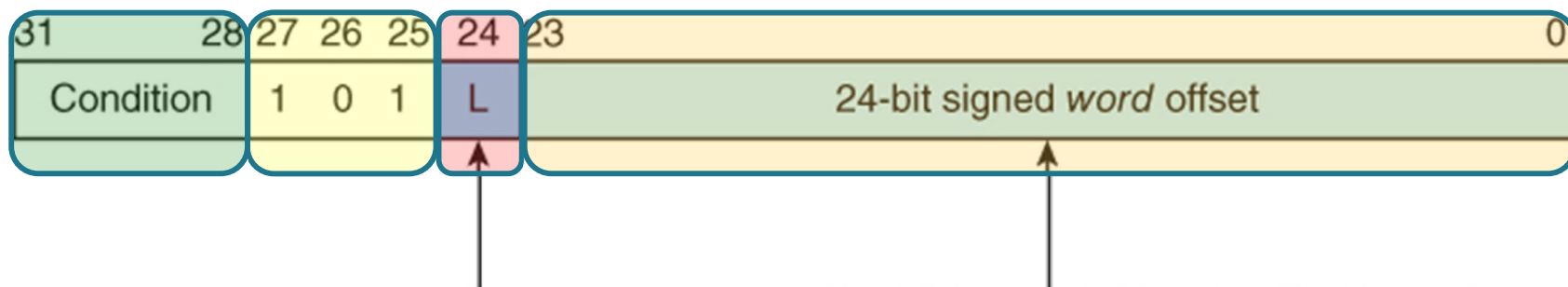
- ❑ **RISC** processors (including **ARM**) *do not provide* a *fully automatic* subroutine call/return mechanism like **CISC** processors.
- ❑ **ARM**'s *branch with link* instruction, **BL**,
 - automatically saves the return address in register **r14**.
- ❑ The branch instruction (Figure 3.41) has 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.

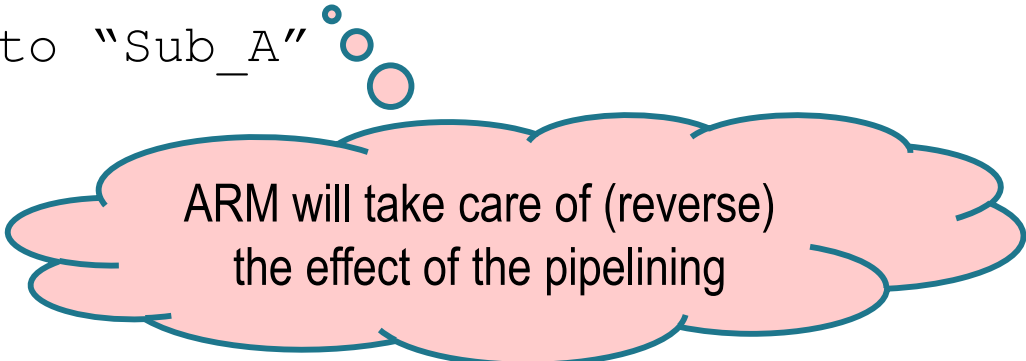
The 24-bit word offset is shifted left twice to create a 26-bit byte offset.

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"
```



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 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
```

- ❑ 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      SPACE 8
32

```

Registers

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

Memory 1

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

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
 - Copy the return address to the **PC** register

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

Subroutine Call and Return

Occupied memory

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 stack.

; 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

You need to re-map the memory to make the stack space read/write enabled (Debug/Memory Map).
The other option is to use a .ini file
You may want to review tutorial 7, slides 93-106.

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

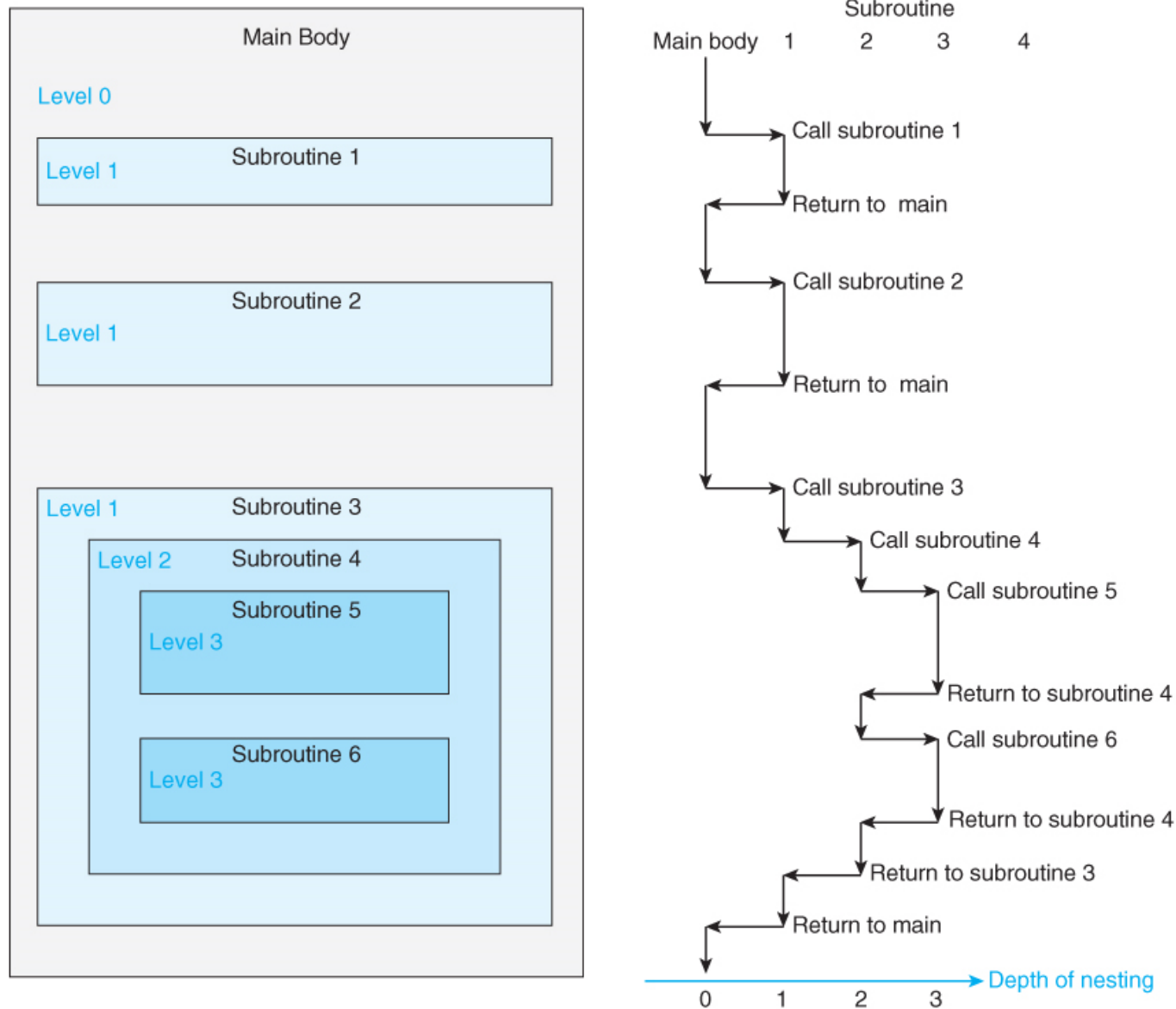
; fix PC and load into r15 to return

Why did 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

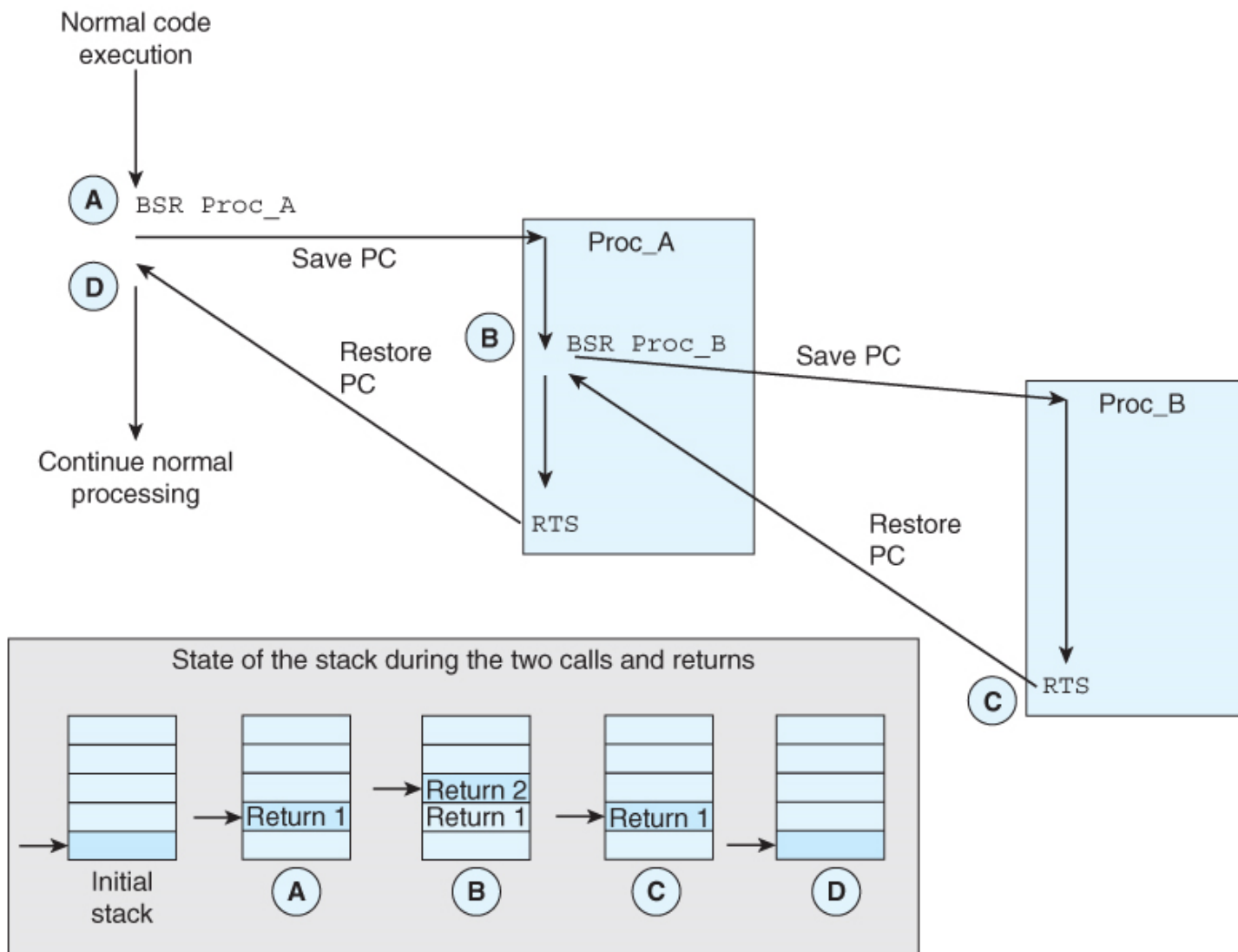


Occupied
memory

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
```

```
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?

200

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

You need to re-map the memory to make the stack space read/write enabled (Debug/Memory Map).
The other option is to use a .ini file
You may want to review tutorial 7, slides 93-106.

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.
- ❑ 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.

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 ASCII representations.
- Bottom Bar:** Includes 'ASSIGN BreakDisable BreakEnable BreakKill', 'Call Stack + Locals', 'Memory 1', 'Simulation', and a timer 't1: 0.00000000 s'.

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

Leaf routines

How is this offset encoded?

```

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
11: ;-----
  
```

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

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	Hex
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

Simulation

Leaf routines

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

- Registers:** A list of registers with their current values. R14 (LR) is highlighted with a value of 0x00000008, and R15 (PC) has a value of 0x00000010.
- Disassembly:** Shows the disassembled instructions for Fun_1 and Fun_2. Instruction 9 is highlighted: `0x00000010 E1A00000 NOP` with the comment `;this is a leaf routine`.
- ex1.asm:** Shows the source code for the routines. Instruction 9 is highlighted: `9 Fun_1 NOP ;this is a leaf routine`.
- Command:** Displays a message: `*** Restricted Version with 32768 Byte Cc` and `*** Currently used: 56 Bytes (0%)`.
- Memory:** Shows a memory dump starting at address 0x0. The first few lines of memory contain instructions: `0x00000000: E2 8F D0 20 EB 00 00 01 EB 00 00 02 EA FF FF FE E1 A0 00 00`.

Leaf routines

The screenshot displays the uVision4 IDE interface for a project named 'ex1.uvproj'. The main window is divided into several panes:

- Registers:** A list of registers (R0-R15, CPSR, SPSR) with their current values. R15 (PC) is highlighted with a value of 0x00000014.
- Disassembly:** A list of instructions with their addresses and values. Instruction 11 is highlighted: `0x00000014 E1A0F00E MOV PC, R14`. Comments indicate that this instruction returns by copying the LR value into the PC.
- ex1.asm:** The source assembly code. Instruction 11 is highlighted: `MOV pc,lr`. Comments indicate that this instruction returns by copying the LR value into the PC.
- Command:** A message box indicating a restricted version of the tool with 32768 Byte Code and 56 Bytes (0%) currently used.
- Memory:** A memory dump starting at address 0x00000000. The dump shows the instruction at 0x00000014: `E1 A0 F0 0E E1 A0 00 00 E4 8D E0 04 EB FF FF FA E5 3D F0 04`.

The status bar at the bottom indicates the simulation is running, with a timer showing 0.00000000 seconds.

Leaf routines

The screenshot shows the µVision4 IDE interface. The 'Registers' window on the left lists registers R0 through R15, with R15 (PC) highlighted at address 0x00000008. The 'Disassembly' window shows the following assembly 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 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

```

A red callout bubble with the text "How is this offset encoded?" points to the instruction at address 0x00000008: `BL Fun_2` with offset `0x00000018`.

The 'ex1.asm' window shows the source 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 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 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 shows the status: "Restricted Version with 32768 Byte Code Memory" and "Currently used: 56 Bytes (0%)". The 'Memory' window also shows the 'Call Stack + Locals' and 'Memory 1' tabs.

Leaf routines

The screenshot displays the uVision4 IDE interface for an ARM project. The **Registers** window on the left shows the current state of registers, with R14 (LR) at 0x0000000C and R15 (PC) at 0x00000018. The **Disassembly** window shows the following instructions:

```

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 **ex1.asm** source code window shows the following assembly with comments:

```

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 memory address 0x0 and the following hex values:

```

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

The **Memory 1** window shows the address 0x0 and the following hex values:

```

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
  
```

The **Command** window also shows the text: **ASSIGN BreakDisable BreakEnable BreakKill**

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 00 00

Simulation t1: 0.00000000 s

Which type of stack
is it?

Leaf routines

The screenshot shows the uVision4 IDE 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)	0x0000000C
R15 (PC)	0x00000020
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 - 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
  
```
- Assembly Source Window (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 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 00
  
```

A red callout bubble points to the instruction at address 0x00000020: *How is this offset encoded?*

A blue arrow points from the R15 (PC) register value 0x00000020 to the memory location at address 0x00000020, which contains the value 00 00 00 0C.

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 0x00000024, and R15 (PC) is highlighted with a value of 0x00000010.
- Disassembly Window:** Shows the assembly code being executed. The current instruction is at address 0x00000010: `E1A00000 NOP`. Subsequent instructions include `MOV pc,lr` (return by copying the LR value into PC), `Fun_2 NOP` (non-leaf routine), `STR lr,[sp],#4` (save link register), `BL Fun_1` (call Fun_1 - overwrites the old link register), and `LDR pc,[sp,#-4]!` (return by copying the LR value (from the stack) into PC).
- Source Code Window (ex1.asm):** Shows the assembly code with comments. 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`. Line 12 is `Fun_2 NOP ;this is a non-leaf routine`. Line 13 is `STR lr,[sp],#4 ;save link register`. Line 14 is `BL Fun_1 ;call Fun_1 - overwrites the old link register`. Line 15 is `LDR pc,[sp,#-4]! ;return by copying the LR value (from the stack) into PC`.
- Command Window:** Displays the message: `*** Restricted Version with 32768 Byte Code Memory *** Currently used: 56 Bytes (0%)`.
- Memory Window:** Shows the memory address 0x0 and the corresponding memory contents: `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 0C 00 00 00 00 00 00 00 00 00 00 00 00`.

Leaf routines

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

- Registers Window:** Shows the current state of registers. R15 (PC) is highlighted with a value of 0x00000014.
- Disassembly Window:** Shows 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 Window:** Shows the source assembly code. The instruction `MOV pc,lr` at address 10 is highlighted, corresponding to the disassembly.
- Command Window:** Shows the status of the simulation, including the number of bytes used (56 Bytes) and the current instruction address (0x00000014).

The assembly code for Fun_2 is as follows:

```

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

```

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 PC

```

Leaf routines

The screenshot shows the uVision4 IDE with the following components:

- Registers Panel:** Lists registers R0 through R15 (PC), CPSR, and SPSR. R15 (PC) is currently set to 0x00000024.
- Disassembly Panel:** Shows the assembly code for the current routine. The code includes:
 - 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`
 - Line 15: `LDR pc,[sp,#-4]! ;return by copying the LR value (from the stack) into`
 - Line 16: `E53DF004 LDR PC,[R13,#-0x0004]!` (highlighted in yellow)
- Source Panel (ex1.asm):** Shows the corresponding assembly code with comments:
 - Line 10: `MOV pc,lr ;return by copying the LR value into PC`
 - Line 11: `;`
 - 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`
 - Line 15: `LDR pc,[sp,#-4]! ;return by copying the LR value (from the stack) into`
 - Line 16: `;`
 - Line 17: `STACK SPACE 0x10`
 - Line 18: `;`
- Command Panel:** Displays the message: `*** Restricted Version with 32768 Byte Cc` and `*** Currently used: 56 Bytes (0%)`.
- Memory Panel:** Shows the memory address 0x0 and the corresponding memory dump:
 - 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
- Bottom Panel:** Includes the `ASSIGN BreakDisable BreakEnable BreakKill` command, a `Simulation` status indicator, and a timer showing `t1: 0.00000000 s`.

Leaf routines

The screenshot shows the uVision4 IDE 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)	0x00000028
R14 (LR)	0x00000024
R15 (PC)	0x0000000C
CPSR	0x000000D3
SPSR	0x00000000
- Disassembly Window:**

```

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
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]!
  
```
- Source Code Window (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
  
```
- 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
  
```
- Command Window:**

```

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

A blue arrow points from the R15 (PC) register value (0x0000000C) to the instruction at address 0x00000028 in the source code window, which is `BL Fun_1`.

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:

```

0x0000000C  EAEFFFFE 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  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]!
  
```

The Registers window on the left shows the current state of the registers, with R13 (SP) at 0x00000028 and R14 (LR) at 0x00000024. The Command window at the bottom shows the status: "Restricted Version with 32768 Byte Code Limit" and "Currently used: 56 Bytes (0%)". The Memory window at the bottom right shows the memory dump for address 0x0, displaying the hex values for the instructions shown in the disassembly window.

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:

```
BL      test                ;call test, save return
                                ;address in r14

...

test STMFD r13!, {r0-r4,r10} ;subroutine test, save working
                                ;registers

. body of code

.
LDMFD r13!, {r0-r4,r10}      ;subroutine completes,
                                ;restore the registers

MOV     pc, r14              ;copy the return address in
                                ;r14 to the PC
```

You need to re-map the memory to make the stack space read/write enabled (Debug/Memory Map).
The other option is to use a .ini file
You may want to review tutorial 7, slides 93-106.

Subroutines and Block Move Instructions

- ❑ If you are using block move STM/LDM instructions to store/load multiple registers to/from the stack, you may also want to store the link register (R14) as well and then load its value directly into the program counter (R15) to save an instruction.

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 register values, including the value of the return address which is placed into the *PC*, to make the return.
 - By doing so, we reduced the size of this code by one instruction

You need to re-map the memory to make the stack space read/write enabled (Debug/Memory Map).
The other option is to use a .ini file
You may want to review tutorial 7, slides 93-106.