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

```
        AREA    BL_instruction, CODE, READWRITE
        ENTRY

        ADR     r4,P              ;register r4 points at P
        ADR     r5,Q              ;register r5 points at Q

        LDR     r0,[r4]           ; get P
        BL      Func1             ; P = (if P > 0 then 16P + 1 else 32P)
        STR     r0,[r4,#8]        ; save P
;
; some code
;
        LDR     r0,[r5]           ; get Q
        BL      Func1             ; Q = (if Q > 0 then 16Q + 1 else 32Q)
        STR     r0,[r5,#8]        ; save P

        MOV     r0, #0x18         ; angel_SWIreason_ReportException
        LDR     r1, =0x20026      ; ADP_Stopped_ApplicationExit
        SVC     #0x123456         ; ARM semihosting (formerly SWI)


Func1   CMP     r0,#0             ;test for x > 0
        MOVGT   r0,r0, LSL #4     ;if x > 0 x = 16x
        ADDGT   r0,r0,#1          ;if x > 0 then x = 16x + 1
        MOVLT   r0,r0, LSL #5     ;ELSE if x < 0 THEN x = 32x
        MOV     pc,r14            ;return by restoring saved PC

        AREA    BL_instruction, DATA, READWRITE
P       DCD     0x00000003        ;P = 3
Q       DCD     0xFFFFFFFF        ;Q = -1
        SPACE   8
```

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

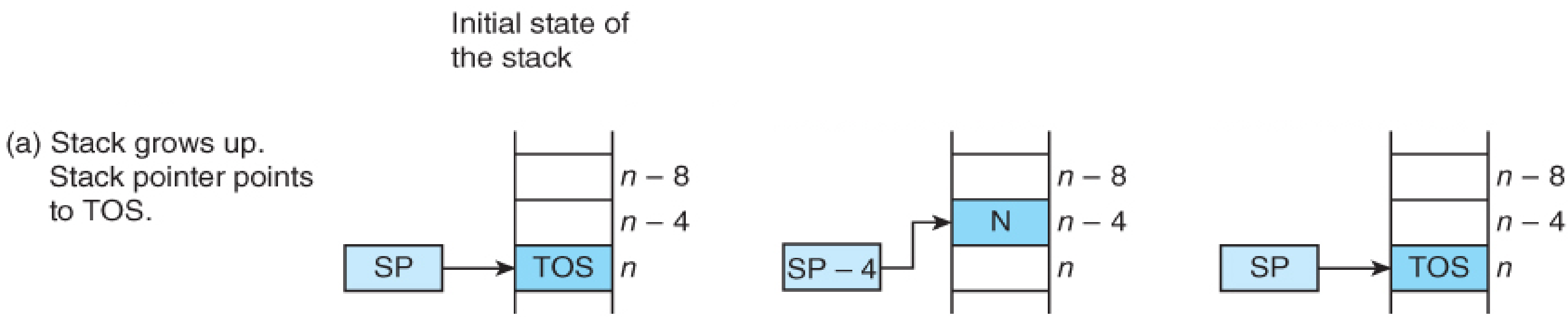
```
CMP r9,r4     ;if r9 < r4
BLLT ABC      ;then call subroutine ABC
```

- **BLLT** means
  - Branch
  - with Link
  - execute on condition Less Than

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




```
PUSH: [SP]   ← [SP] - 4  ;Adjust the stack pointer
      [[SP]] ← data      ;push data onto the stack
```

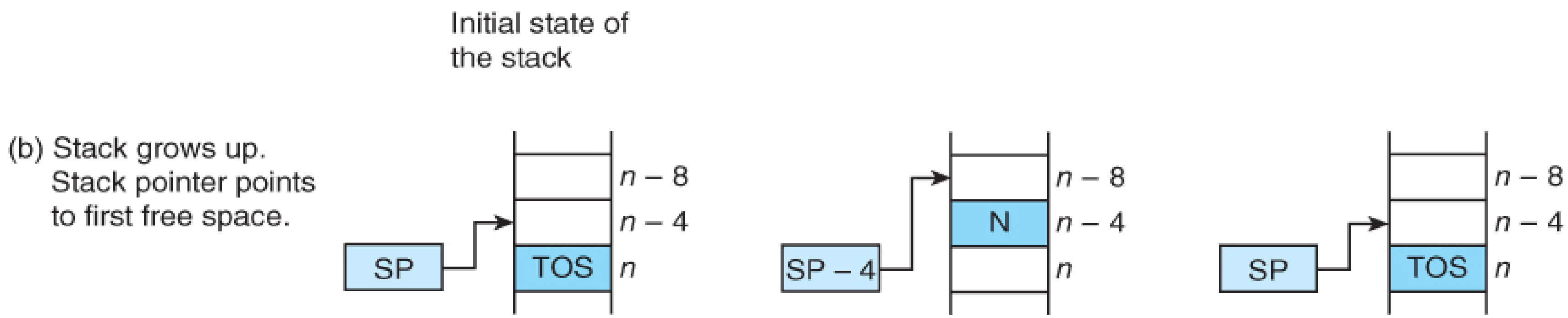
Pre-update

Post-update

```
POP:  data   ← [[SP]]    ;pull data off the stack
      [SP]   ← [SP] + 4  ;Adjust the stack pointer
```

TOS means ***top of stack***

The Stack



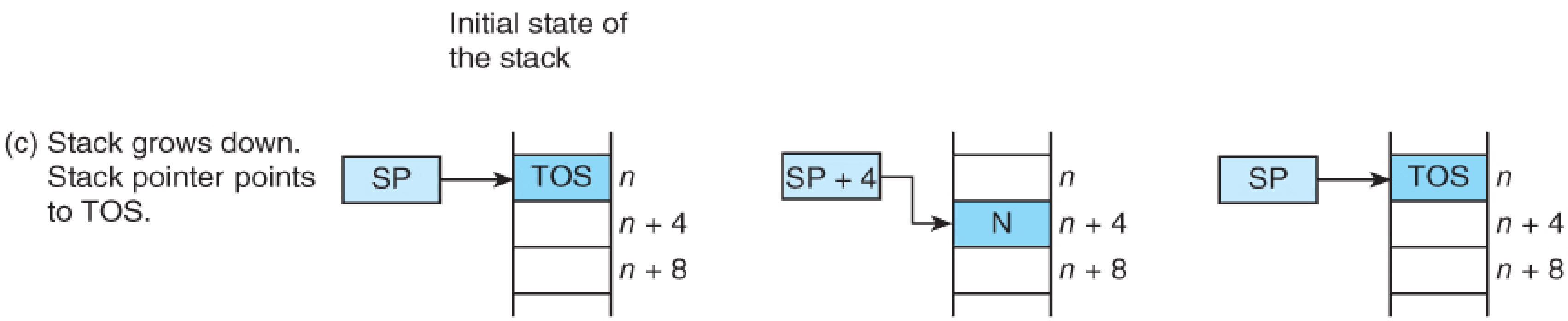

```
PUSH: [[SP]] ← data        ;push data onto the stack
      [SP]   ← [SP] - 4   ;Adjust the stack pointer

POP:  [SP]   ← [SP] + 4   ;Adjust the stack pointer
      data   ← [[SP]]     ;pull data off the stack
```

Post-update

Pre-update

TOS means ***top of stack***

The Stack




```
PUSH: [SP]   ← [SP] + 4  ;Adjust the stack pointer
      [[SP]] ← data      ;push data onto the stack
```

Pre-update

Post-update

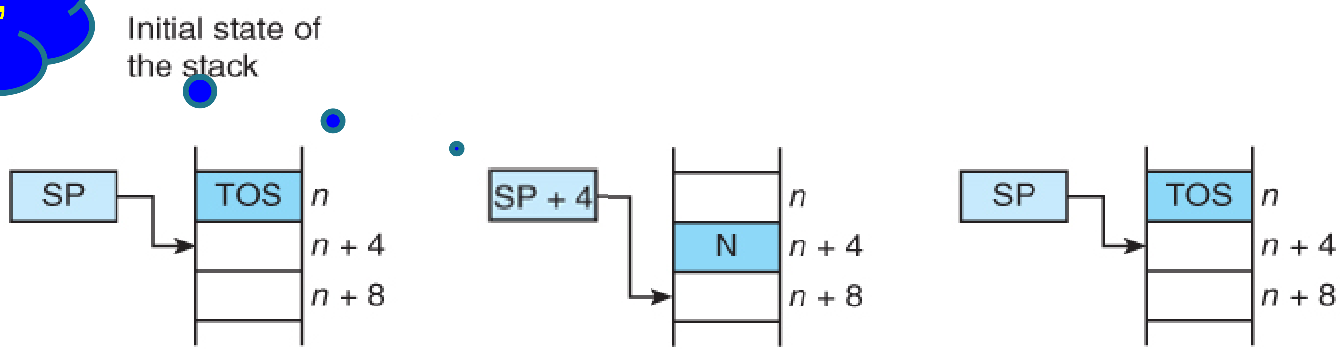
```
POP:  data   ← [[SP]]    ;pull data off the stack
      [SP]   ← [SP] - 4  ;Adjust the stack pointer
```

TOS means ***top of stack***

The Stack

It is SP+4, not SP+8

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




```
PUSH: [[SP]] ← data         ;push data onto the stack
      [SP]   ← [SP] + 4     ;Adjust the stack pointer
```

Post-update

```
POP:  [SP]   ← [SP] - 4     ;Adjust the stack pointer
      data   ← [[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:

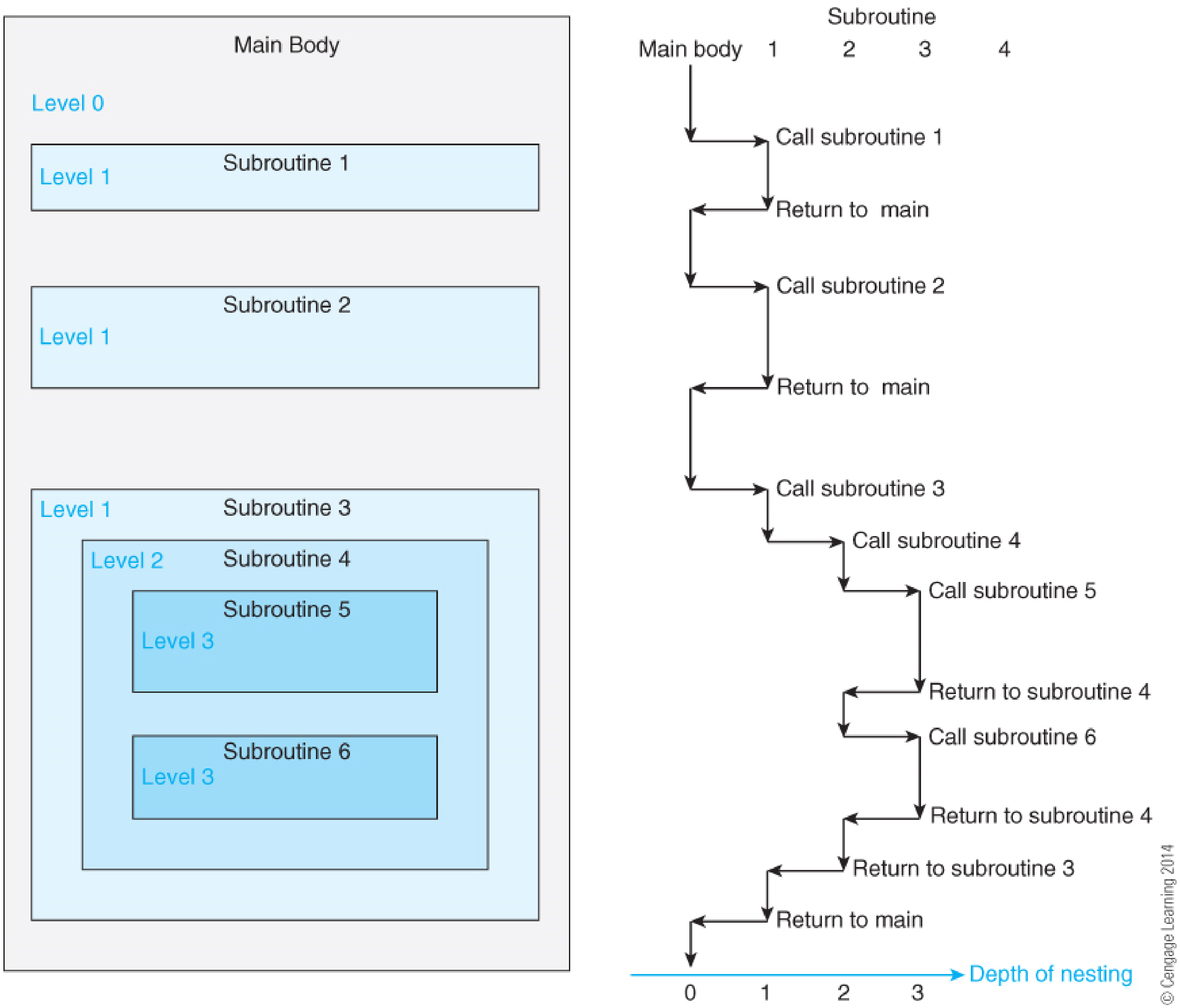
```
LDR r12,[r13],#+4    ;get saved PC and post-increment
                     ;stack pointer
SUB r15,r12,#4       ;fix PC and load into r15 to return
```

Why did we subtract 4?

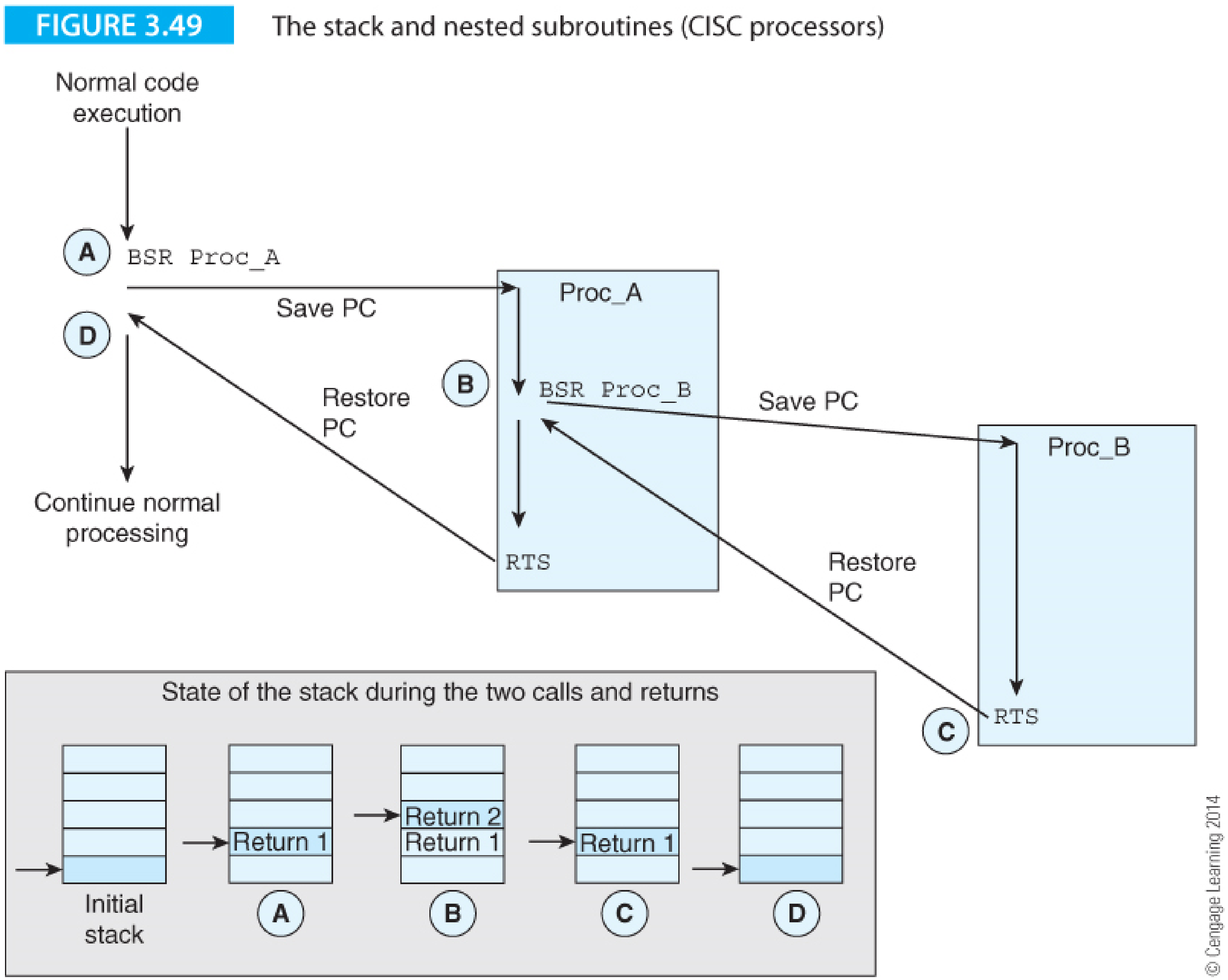
Why did not we copy the stack content directory to r15?

Nested subroutines

FIGURE 3.48 An example of nested subroutines



Example of nested subroutine



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

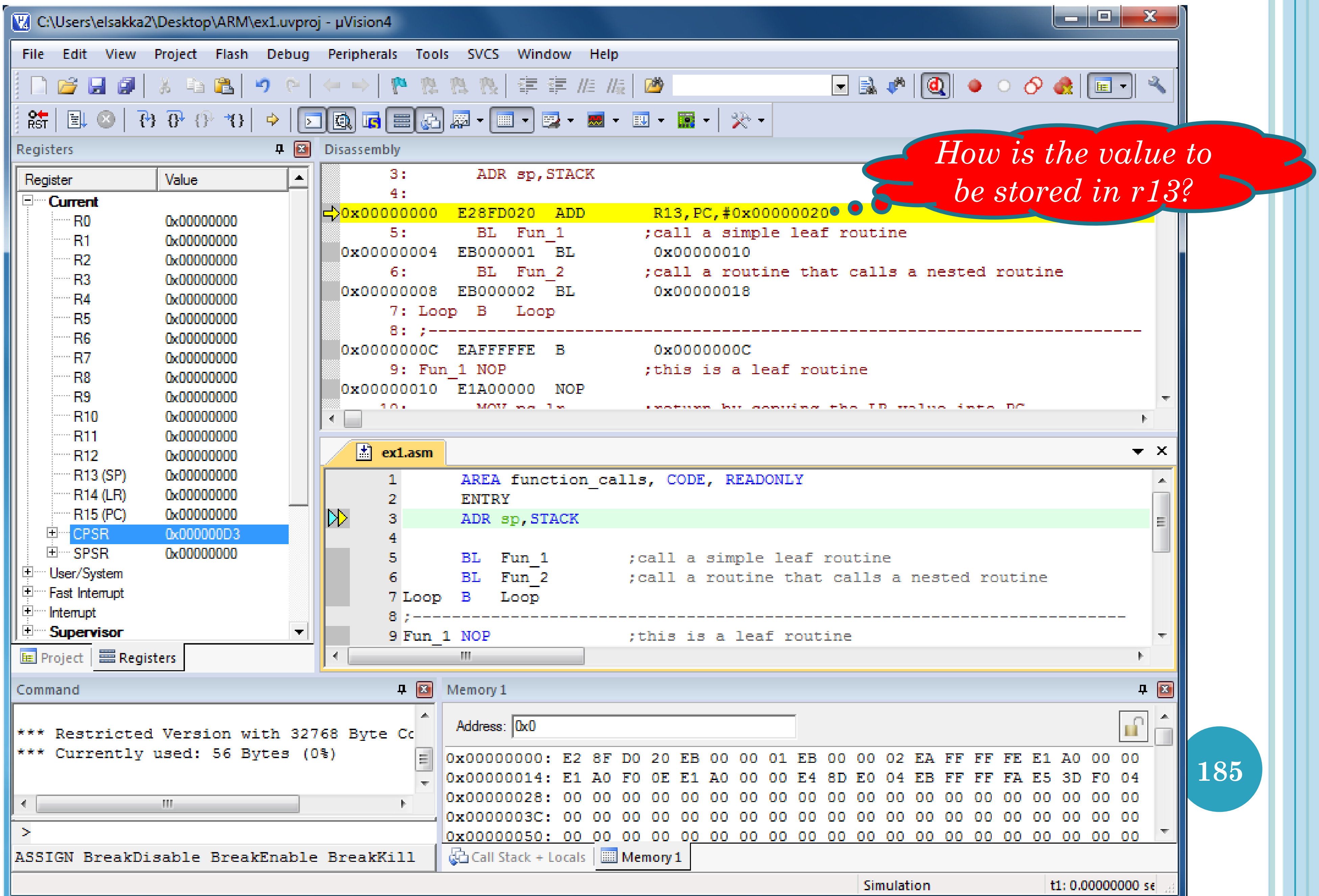
- Subroutine `Fun_1` is a leaf subroutine that does not call a nested subroutine and, therefore, we don't have to worry about saving the link register, **`r14`**, and we can return by executing `MOV pc, lr`.

- 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`.

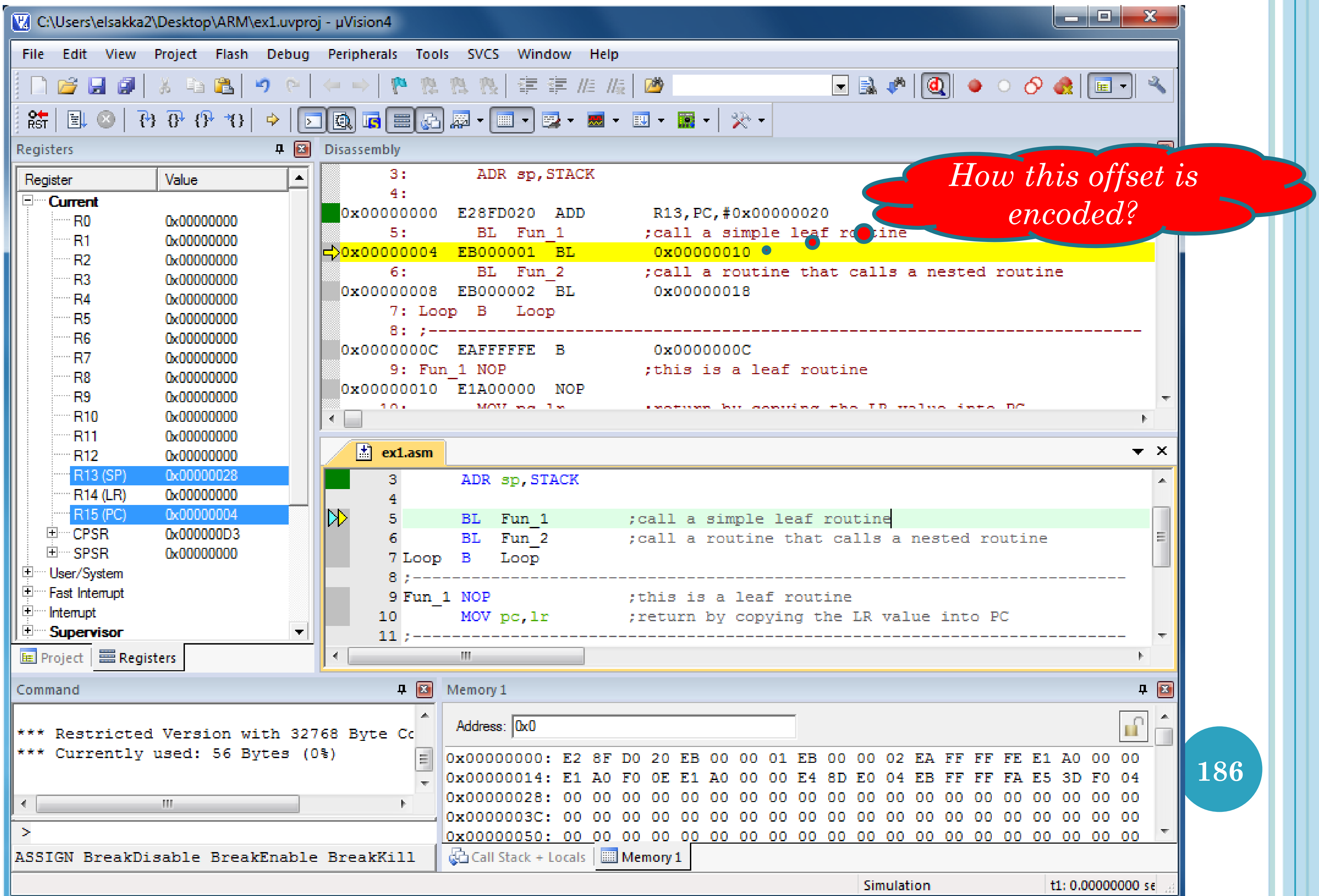
- The simplest way of ***saving*** the link register is to ***push*** it on the stack.

- To return from `Fun_2`, we ***restore the pushed*** **`r14`** into the program counter.

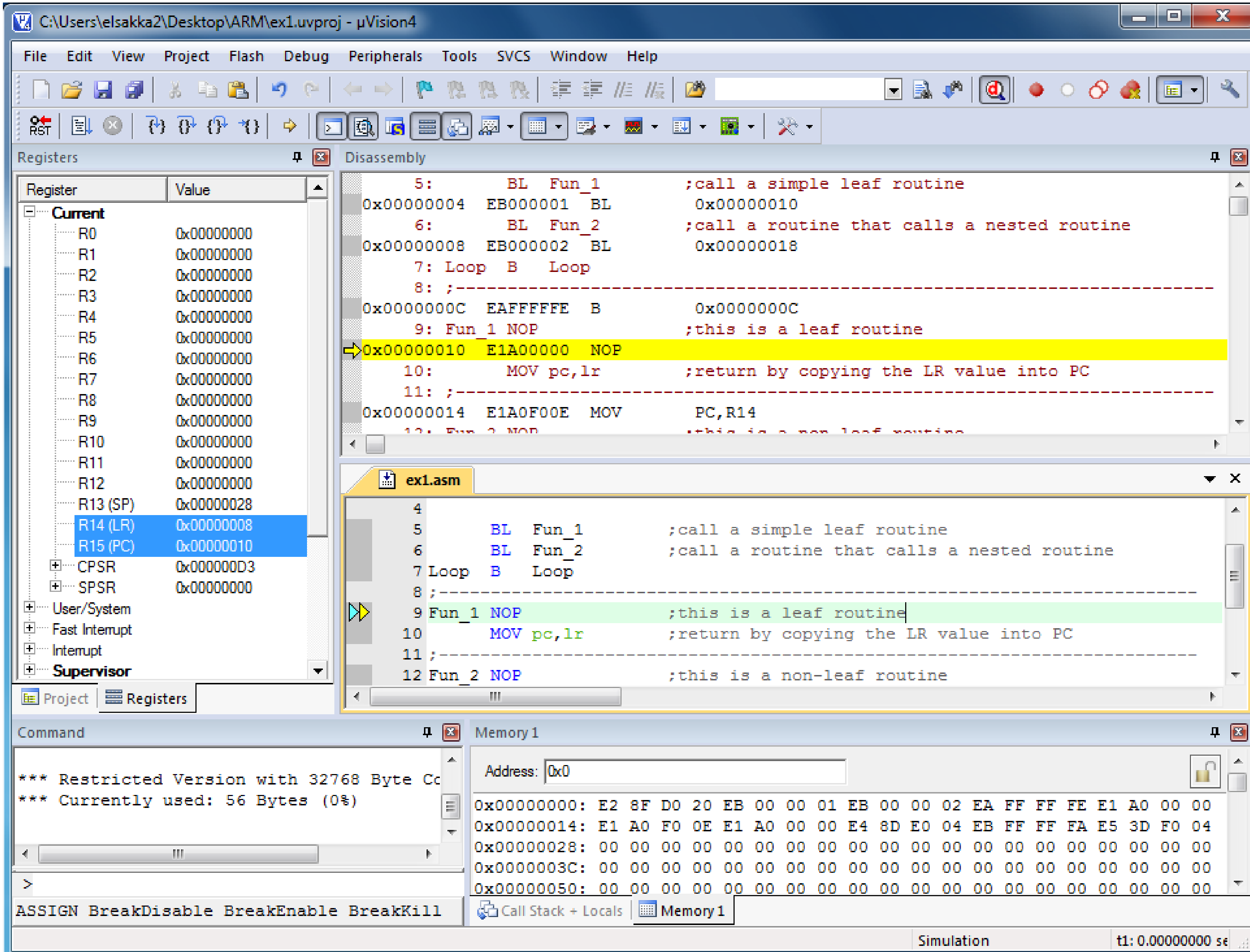
Leaf routines



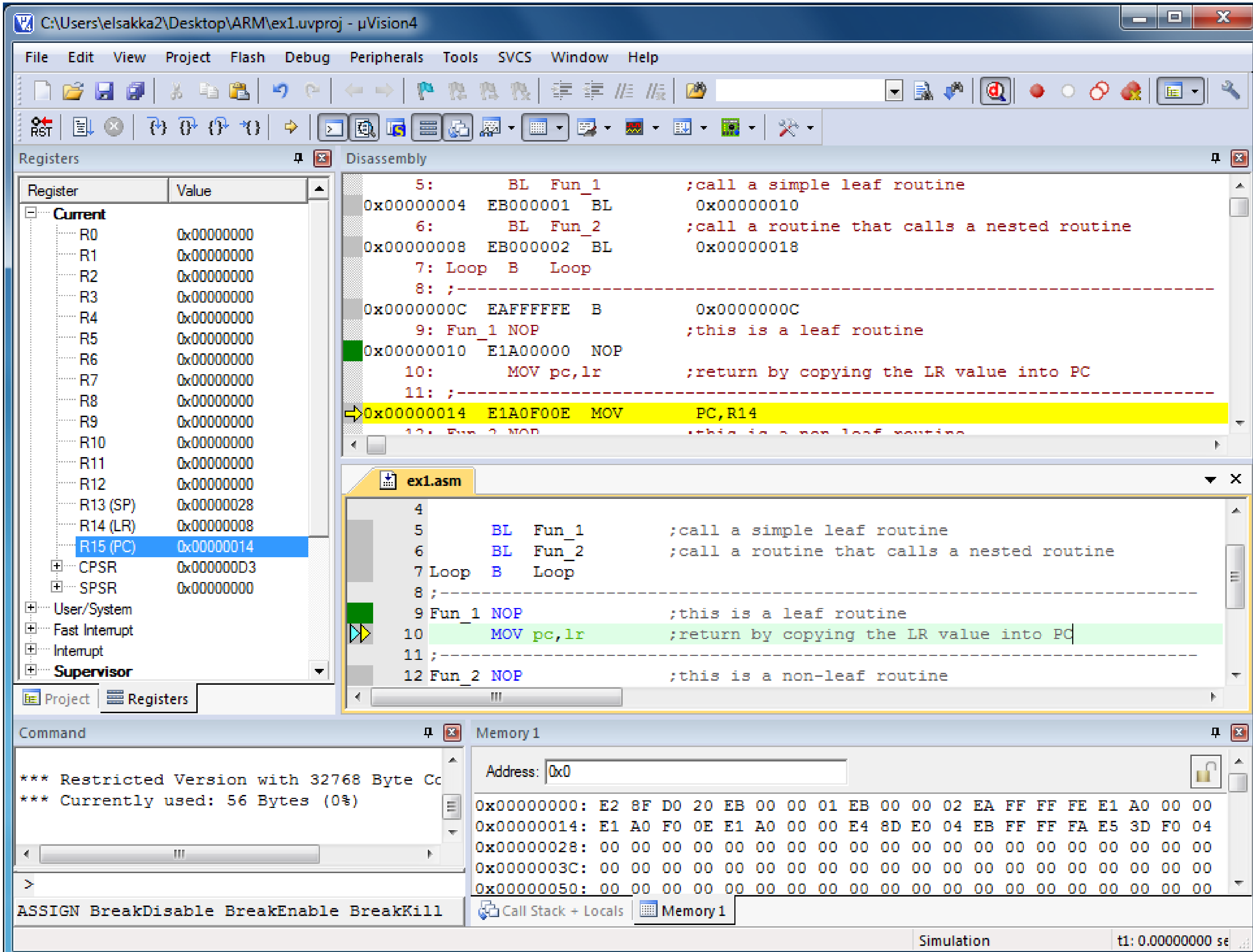
Leaf routines



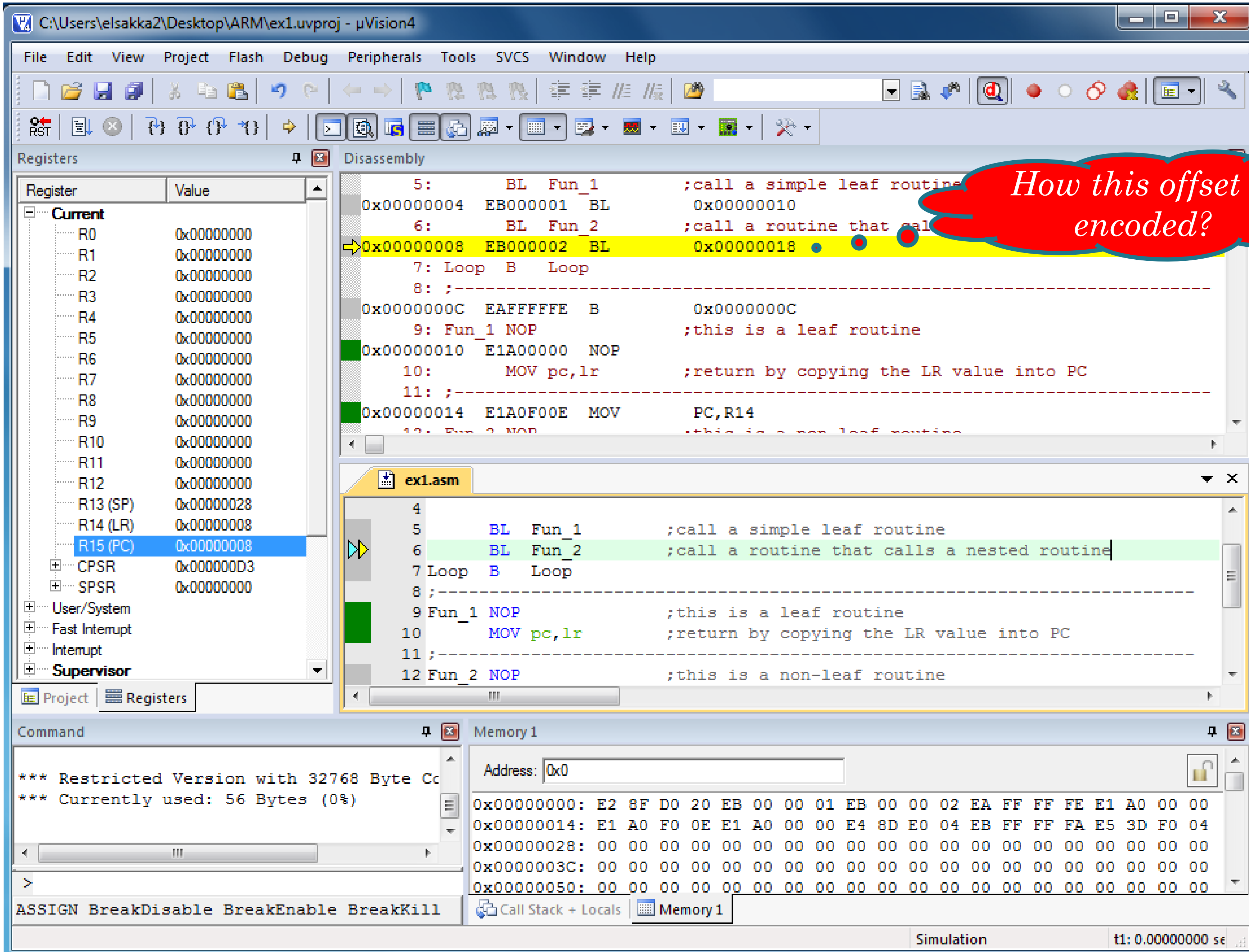
Leaf routines



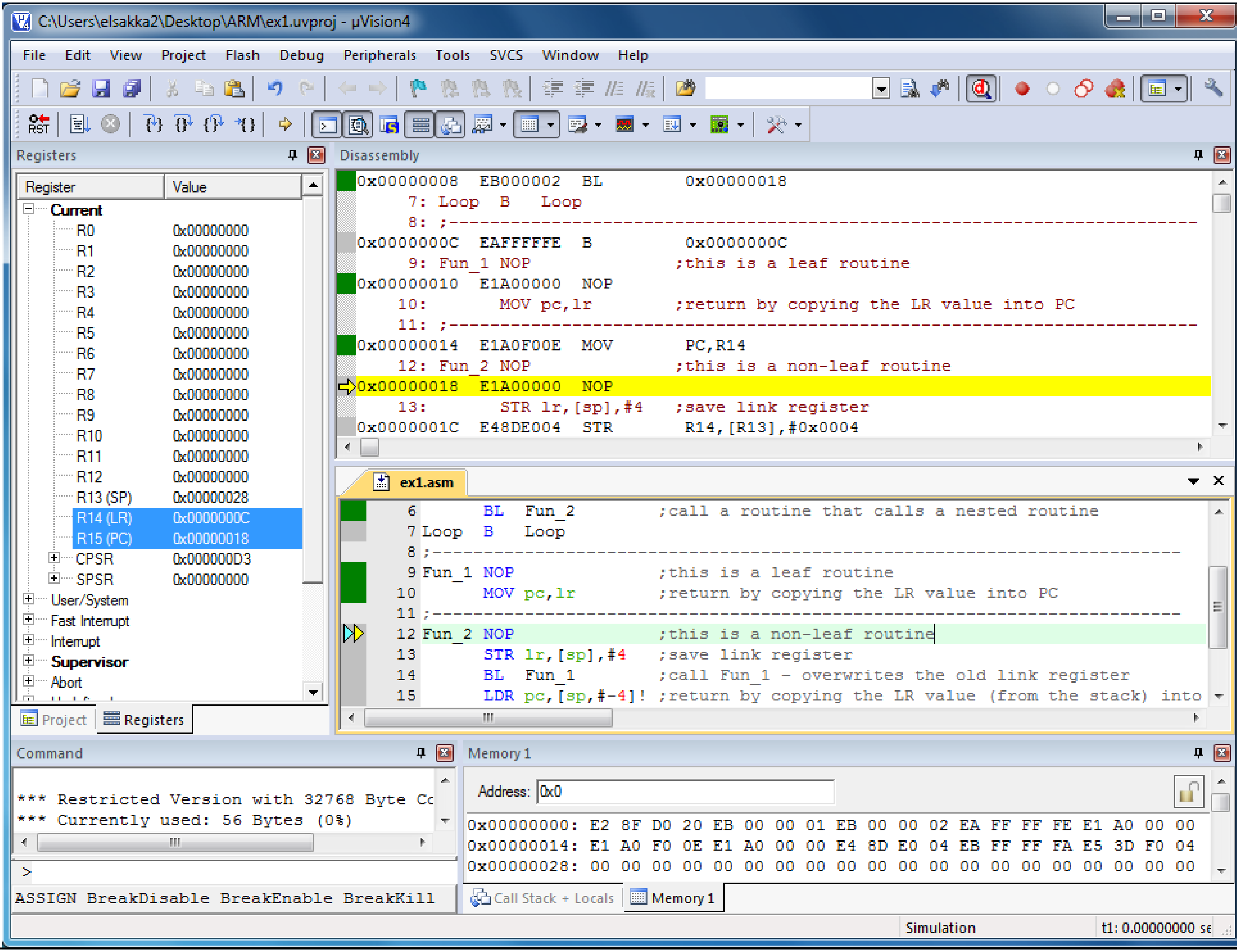
Leaf routines



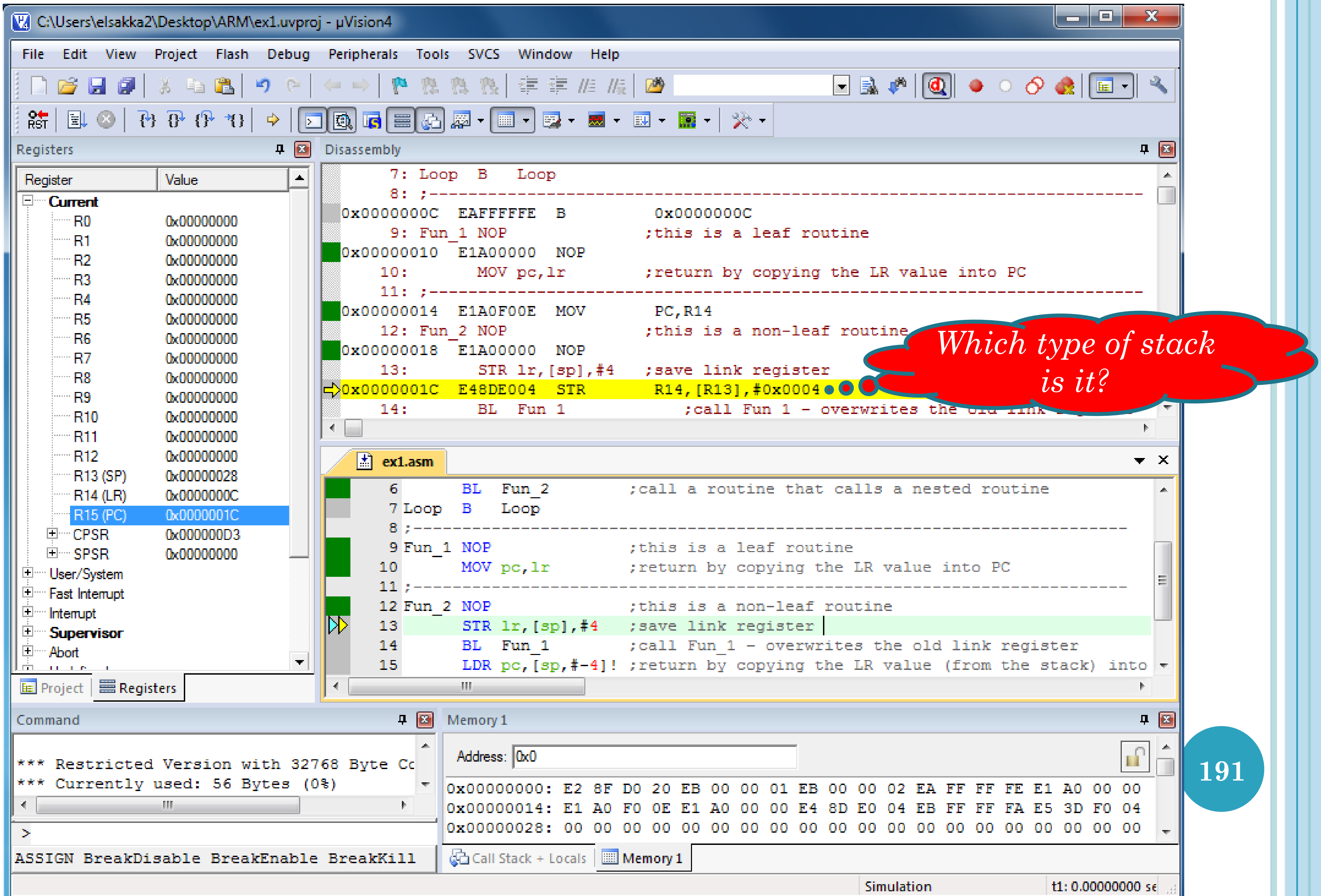
Leaf routines



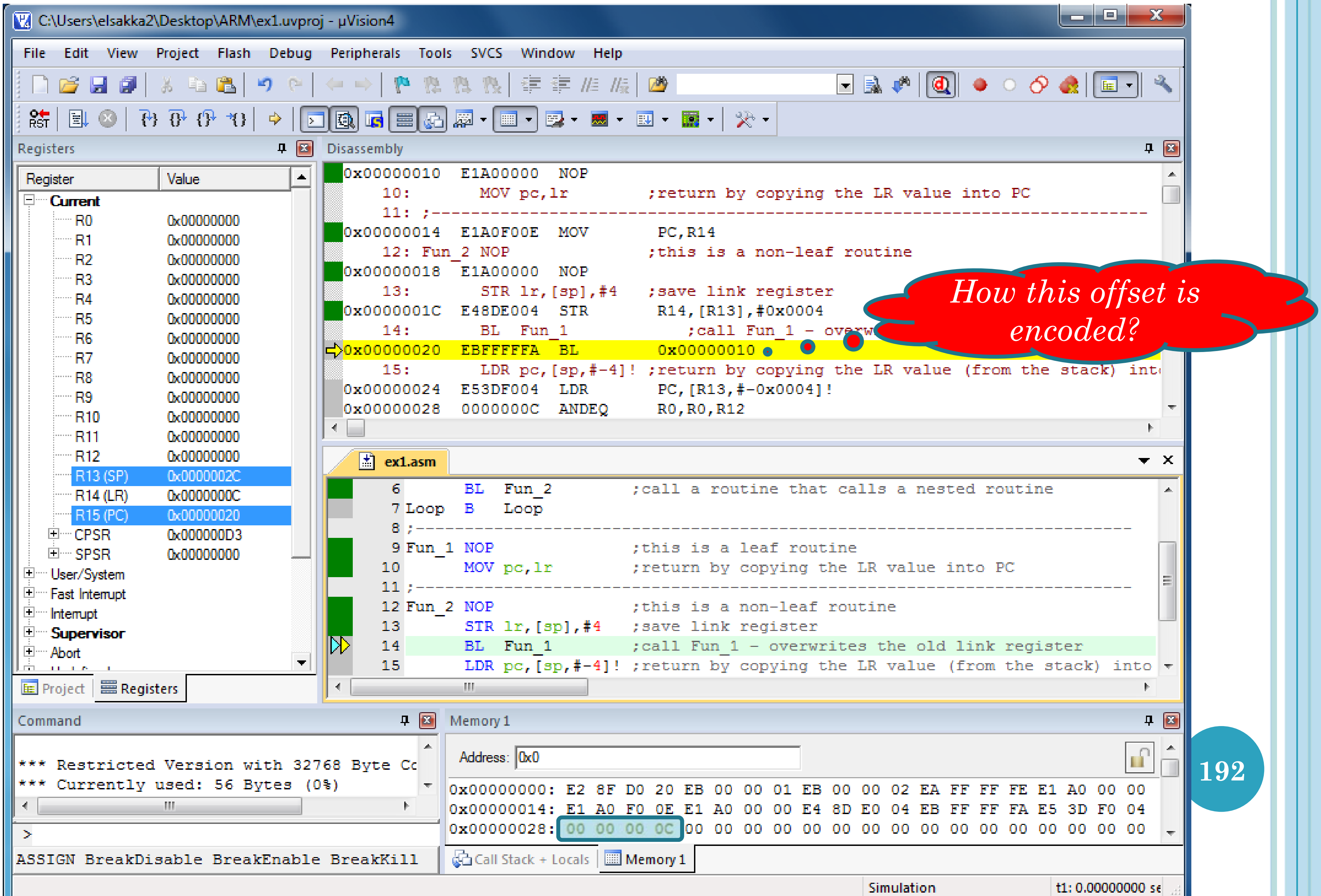
Leaf routines



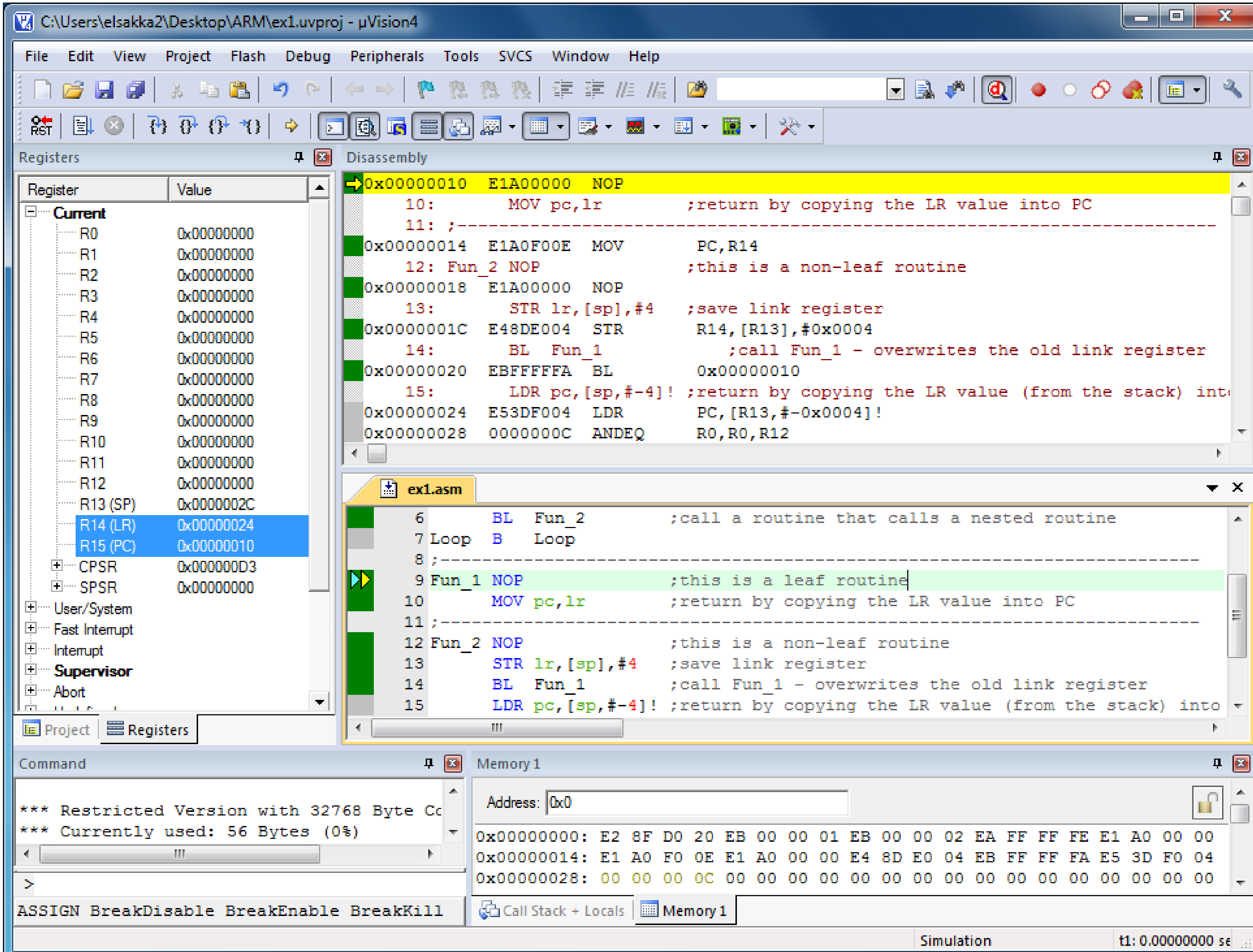
Leaf routines



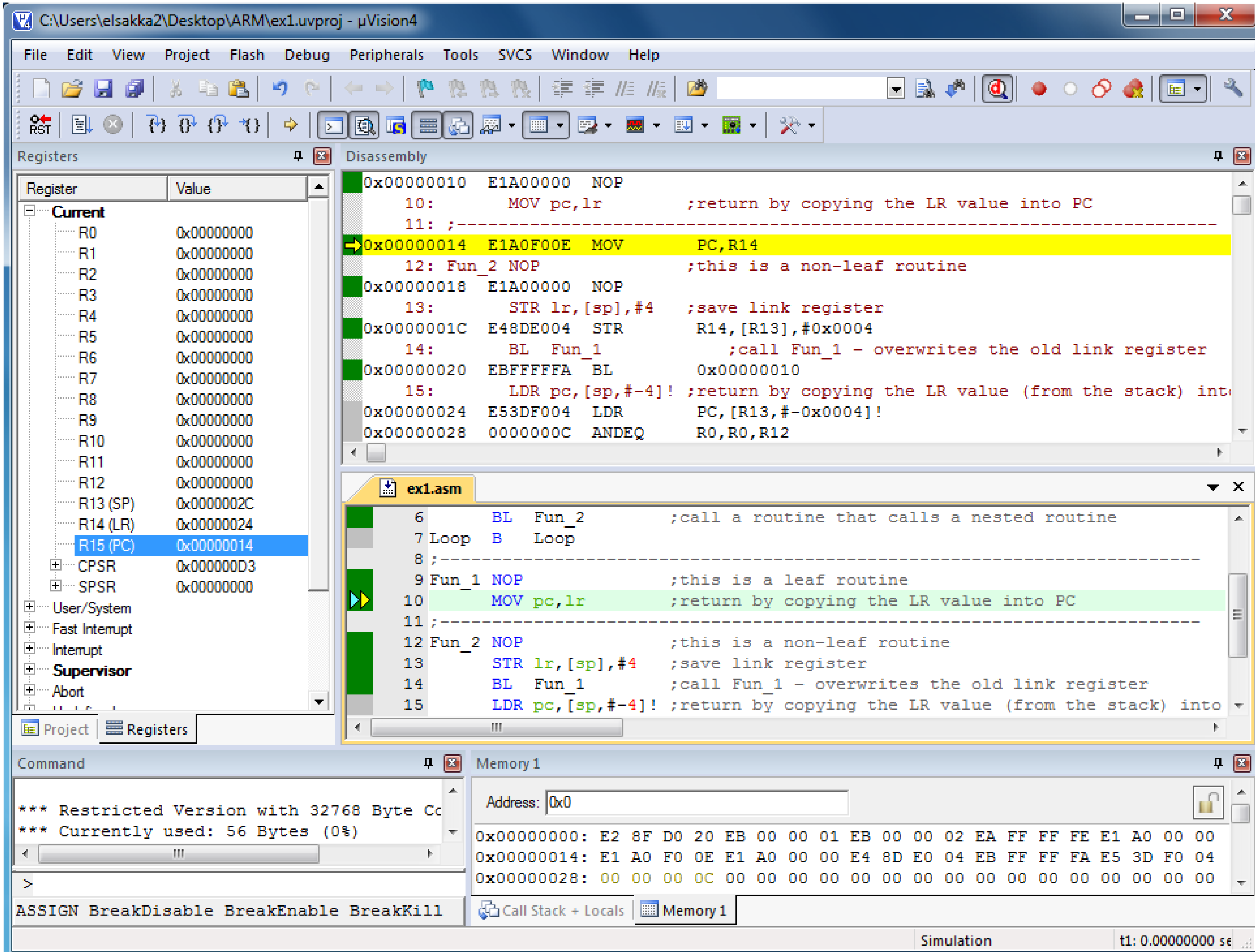
Leaf routines



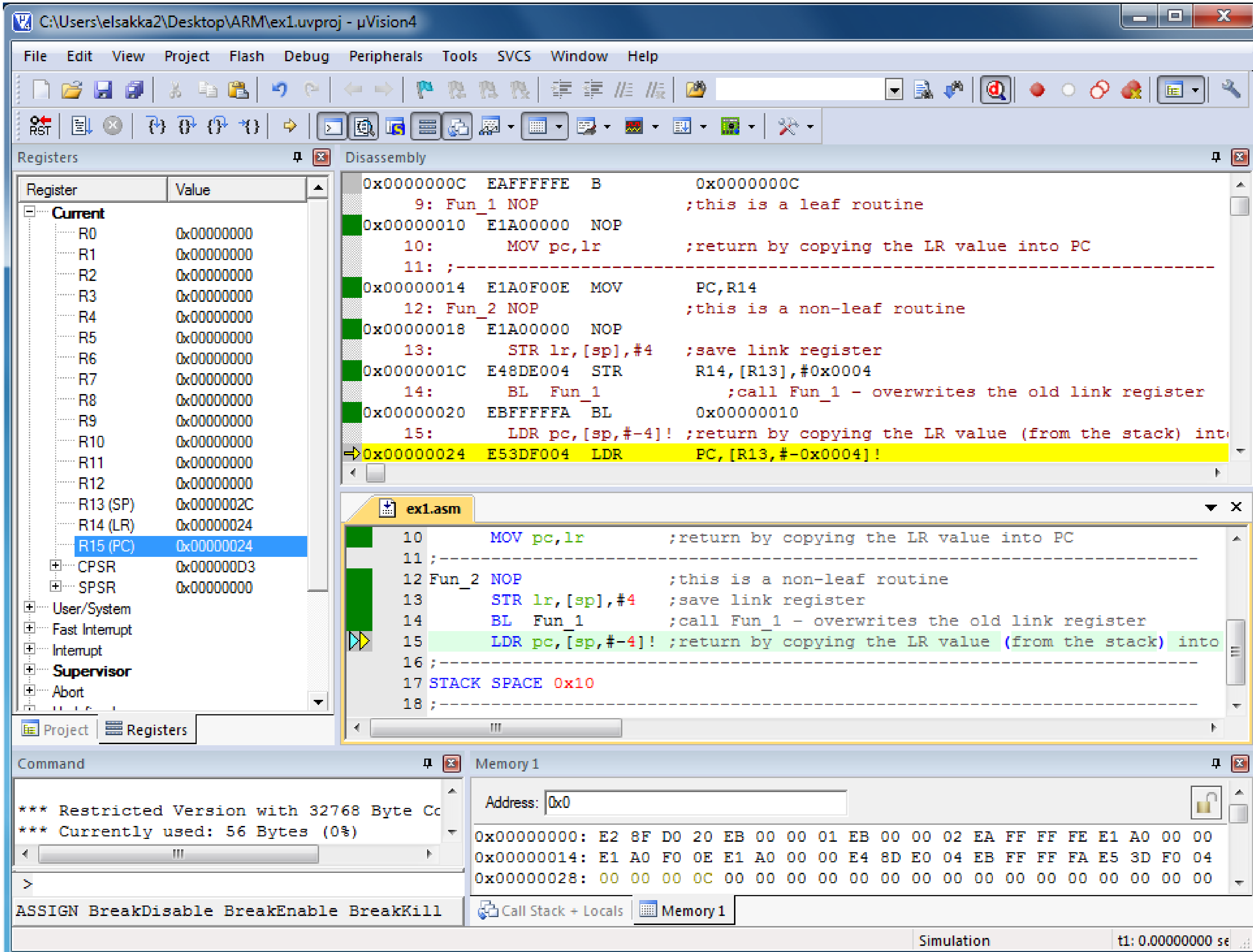
Leaf routines



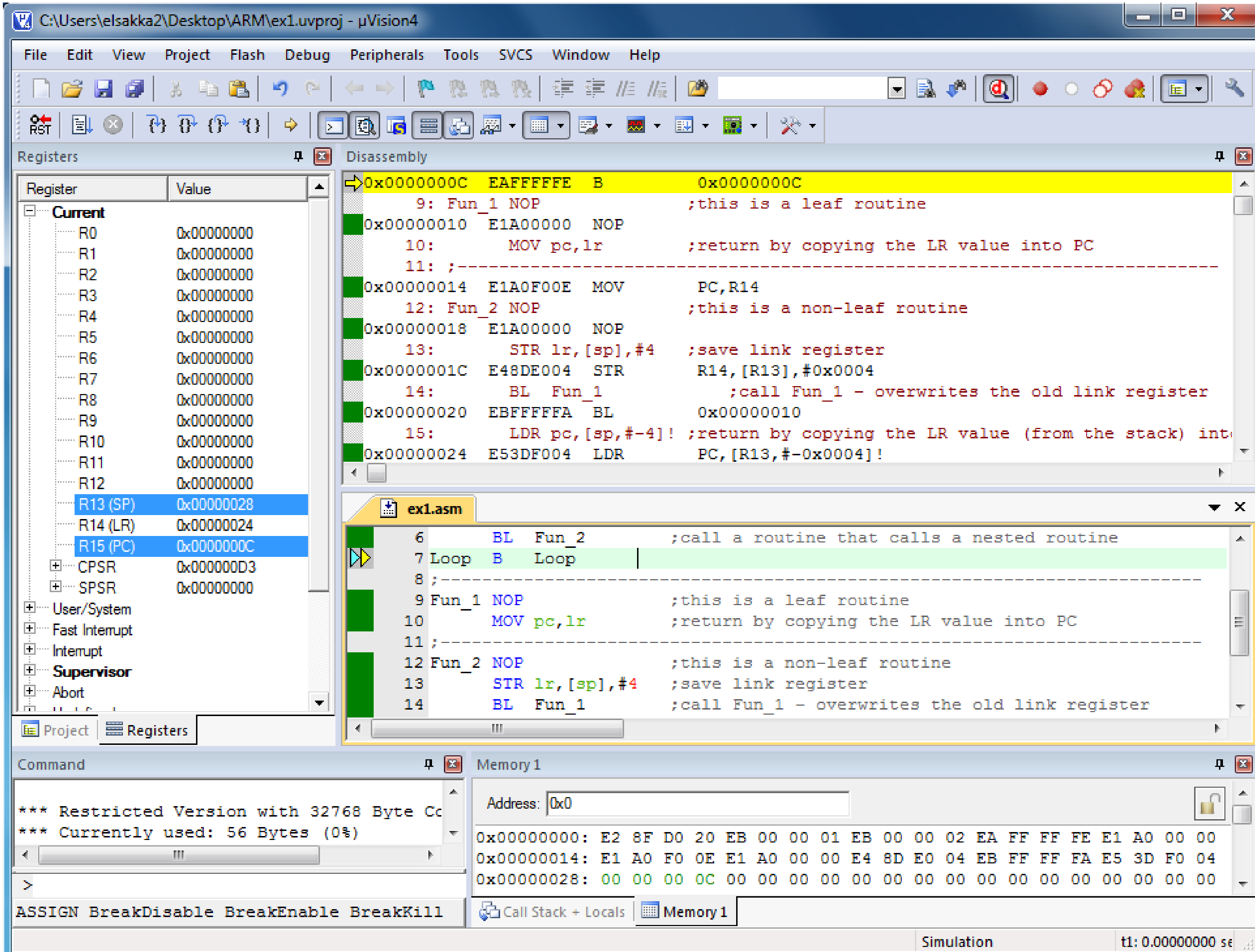
Leaf routines



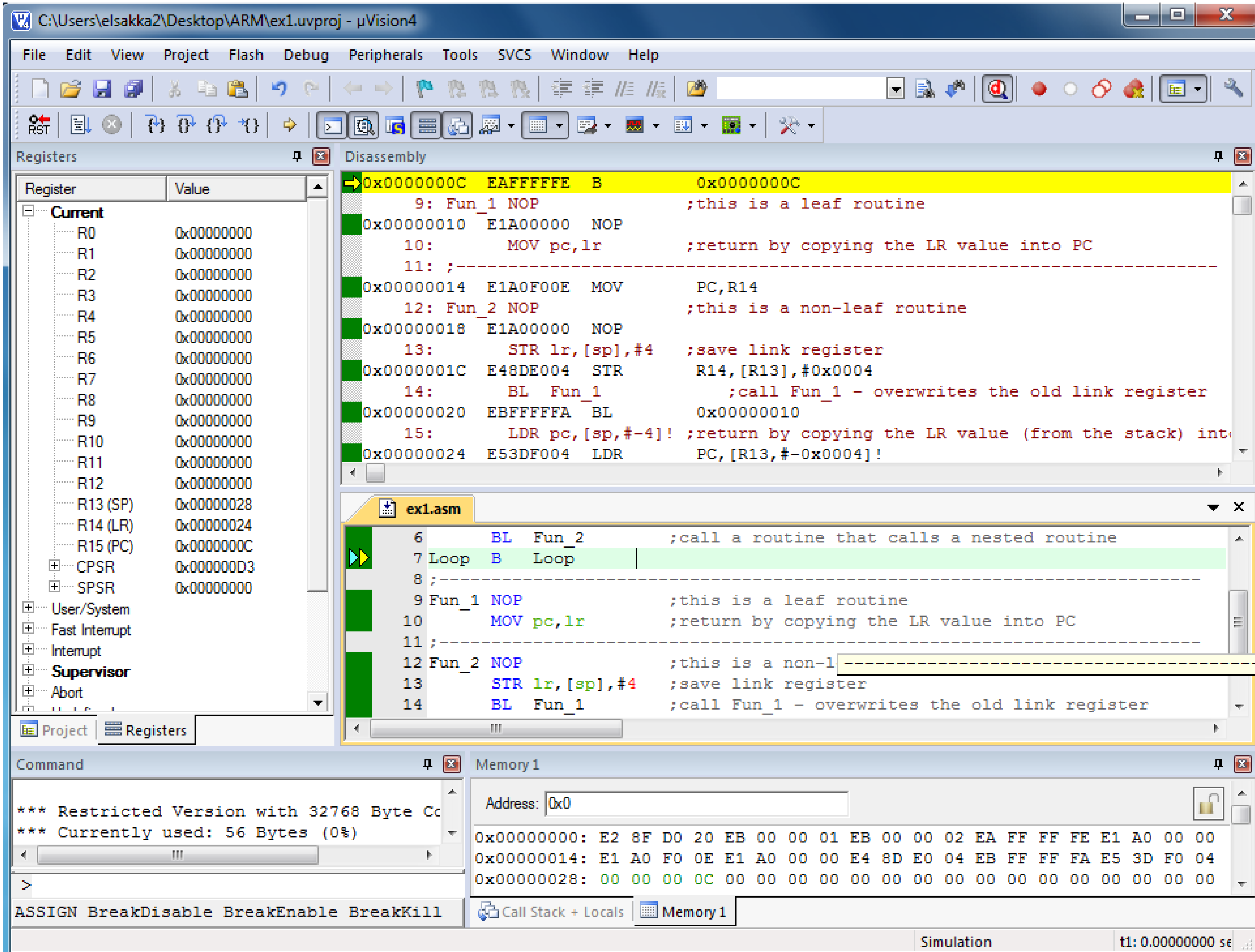
Leaf routines



Leaf routines



Leaf routines



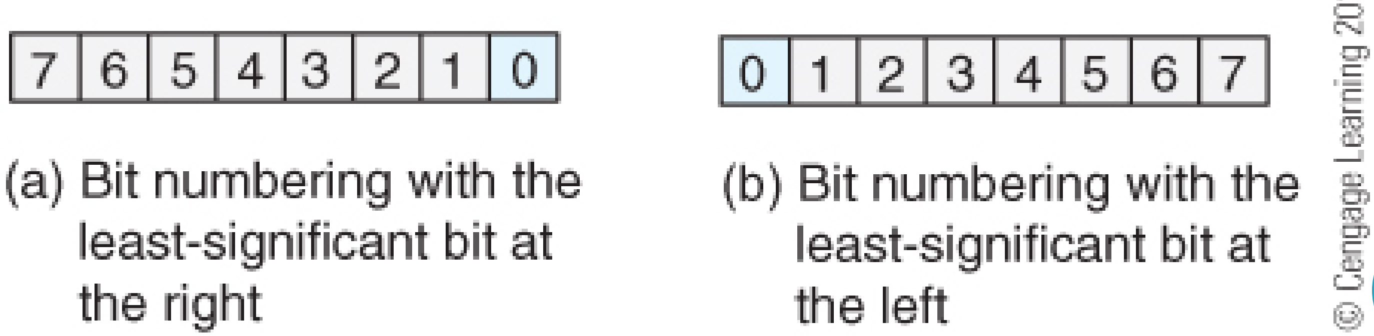
Data Organization and Endianism

- 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*).