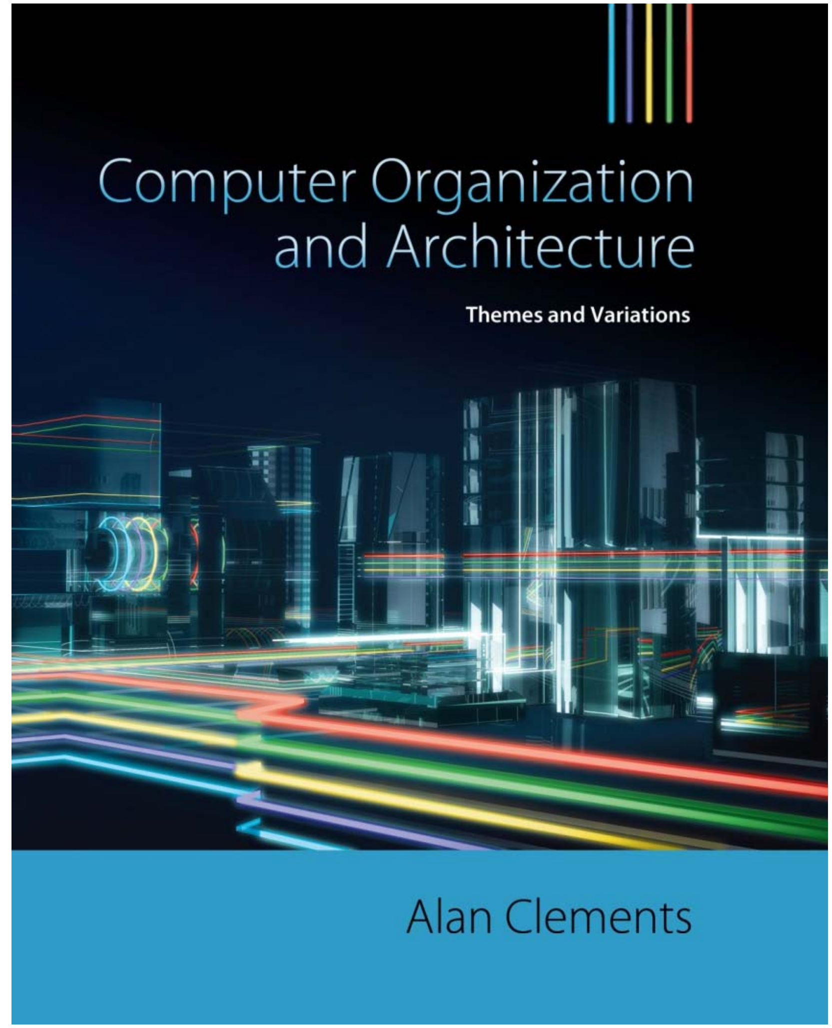
Part 0xE

CHAPTER 3

Architecture and Organization



CENGAGE Learning

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 serve a given function.

- A ***subroutine*** gives the simplest form of program abstraction.

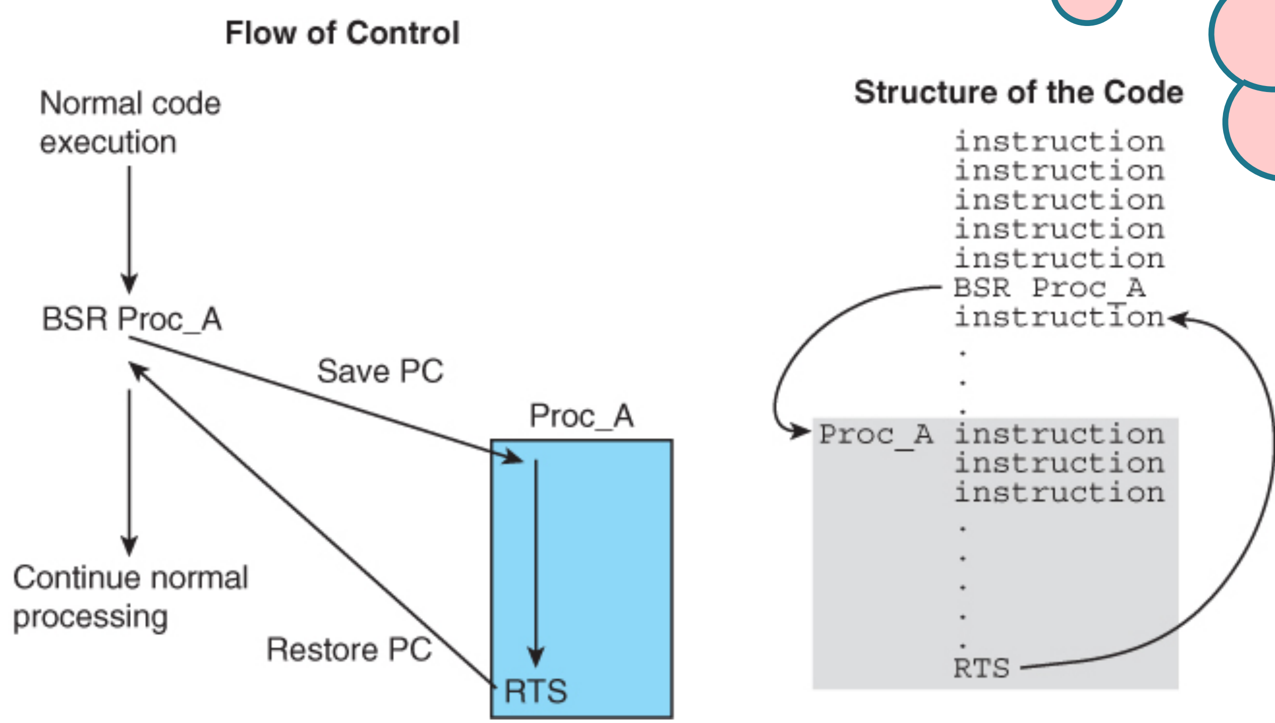
- There are two main characteristics in any subroutine.

  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 hypocritical 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.

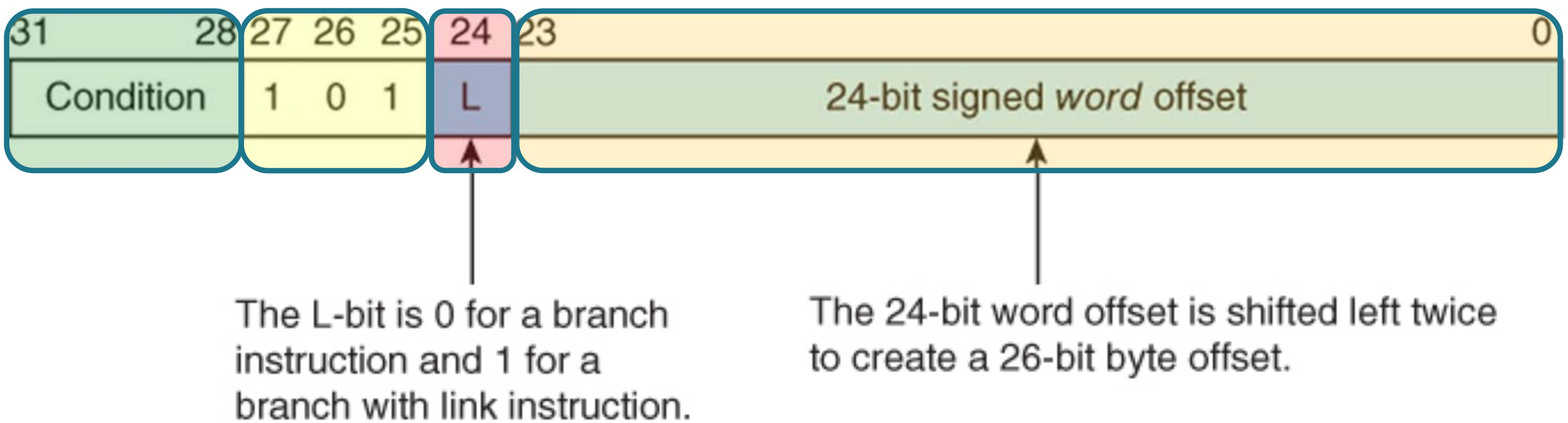
This is the main difference between B and BL

- **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*).

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



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 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
```

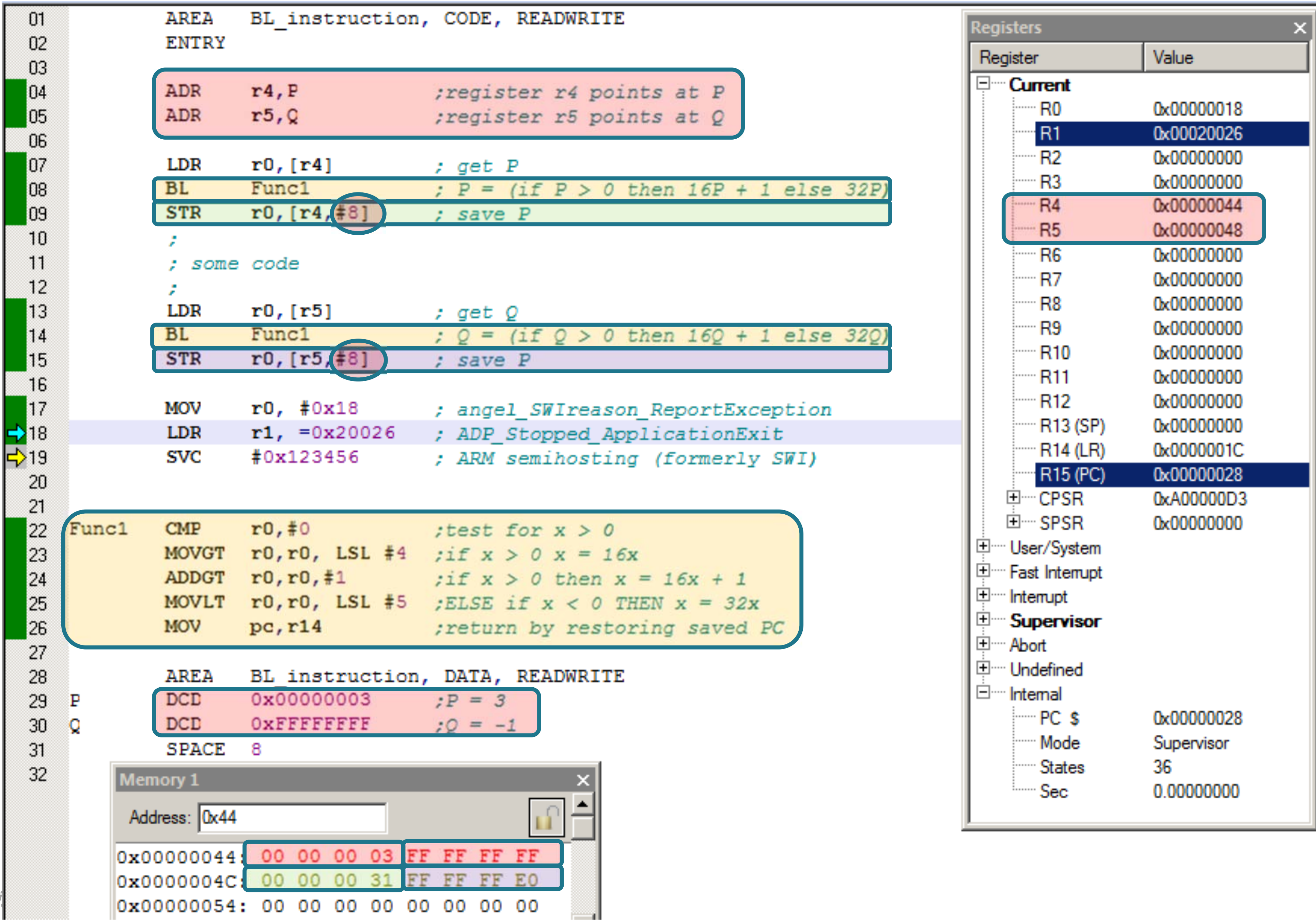
- Consider the following invocation of the above subroutine.

```
      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



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.
  - Typically, this operation is implemented in **CISC** processors.
  - Once the execution of the subroutine code is completed, a *return from subroutine* instruction is executed
    - Popping the return address from the stake
    - Copy the return address to the PC

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

Grows up

Occupied memory

Subroutine Call and Return

- *Example*

This is B. It is NOT BL

```
...                  ;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
...
```

The proper return address

The address pushed onto the stake.

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:

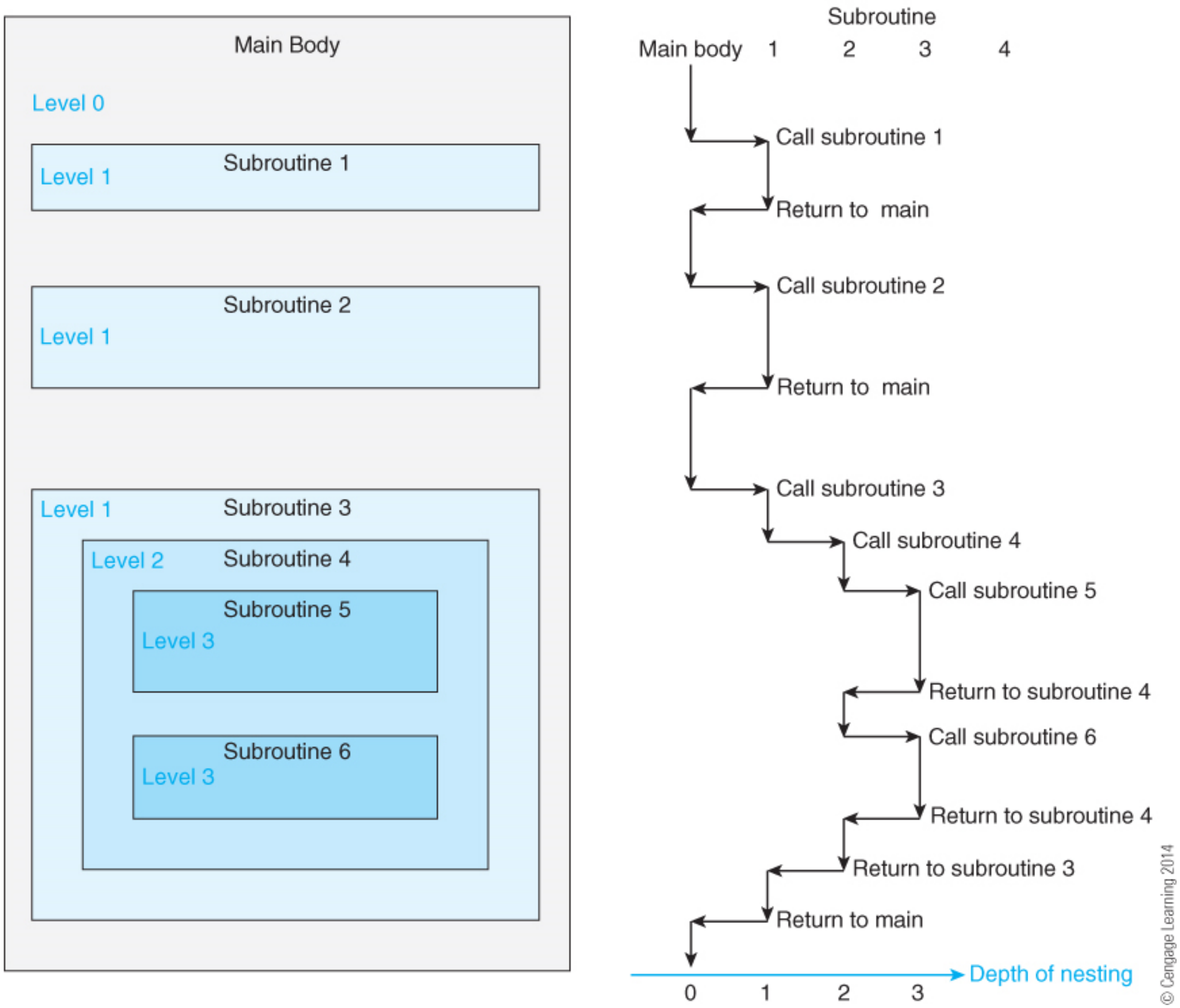
```
LDR r12,[r13],#+4     ;get saved PC and post-increment
                      ;stack pointer
SUB r15,r12,#4        ;fix PC and load into r15 to return
```

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

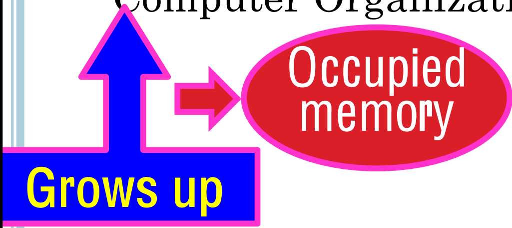
The 4 is subtracted to make the popped address pointing to the proper return address.

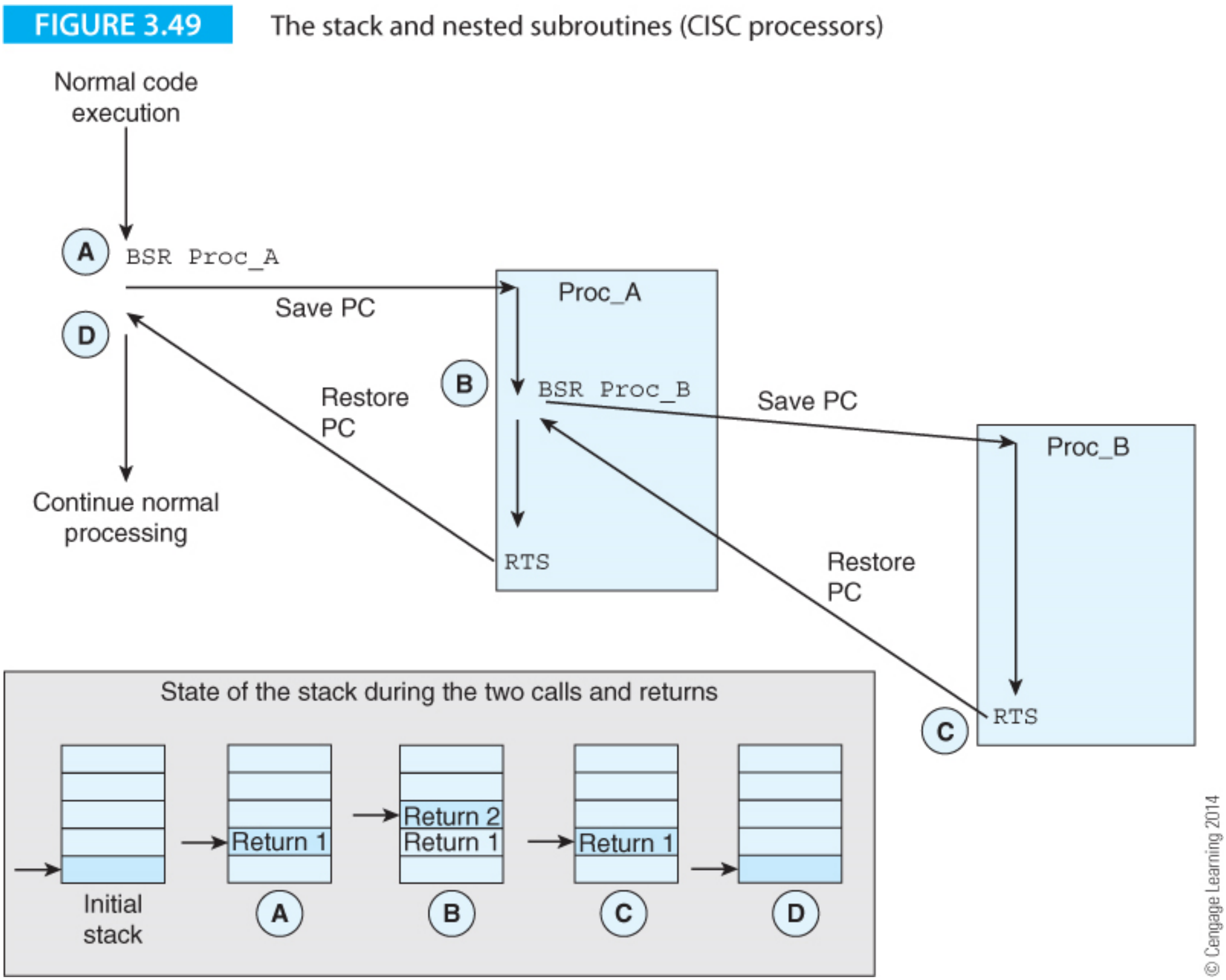
Nested subroutines

FIGURE 3.48 An example of nested subroutines



Example of nested subroutine





Grows down

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?

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

Leaf routines

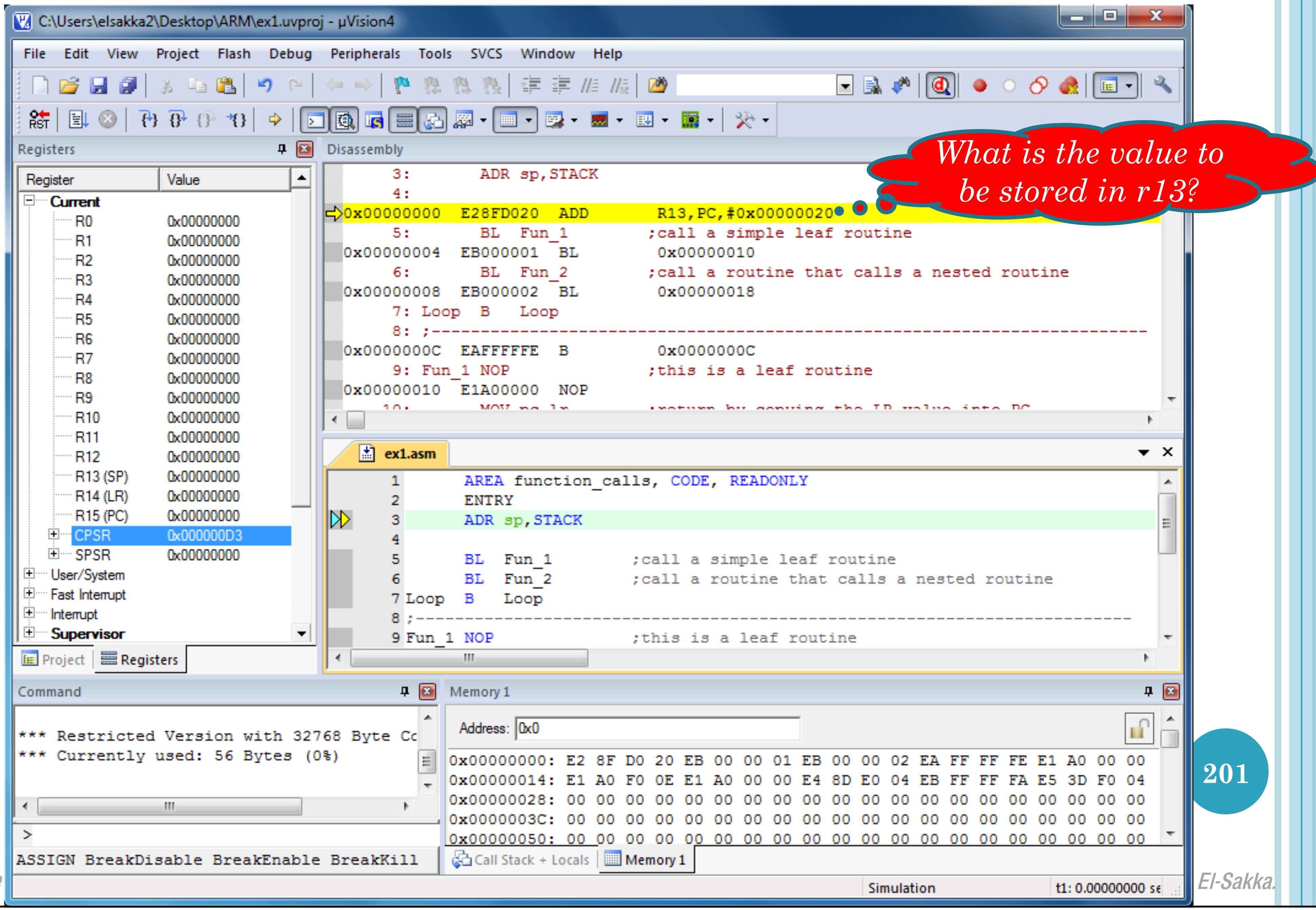
- Subroutine `Fun_1` is a leaf subroutine that does not call any other subroutine and, therefore, we don't have to worry about saving the link register, **`r14`**, and we can return by executing `MOV` **`pc`**`,lr`.

- 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`.

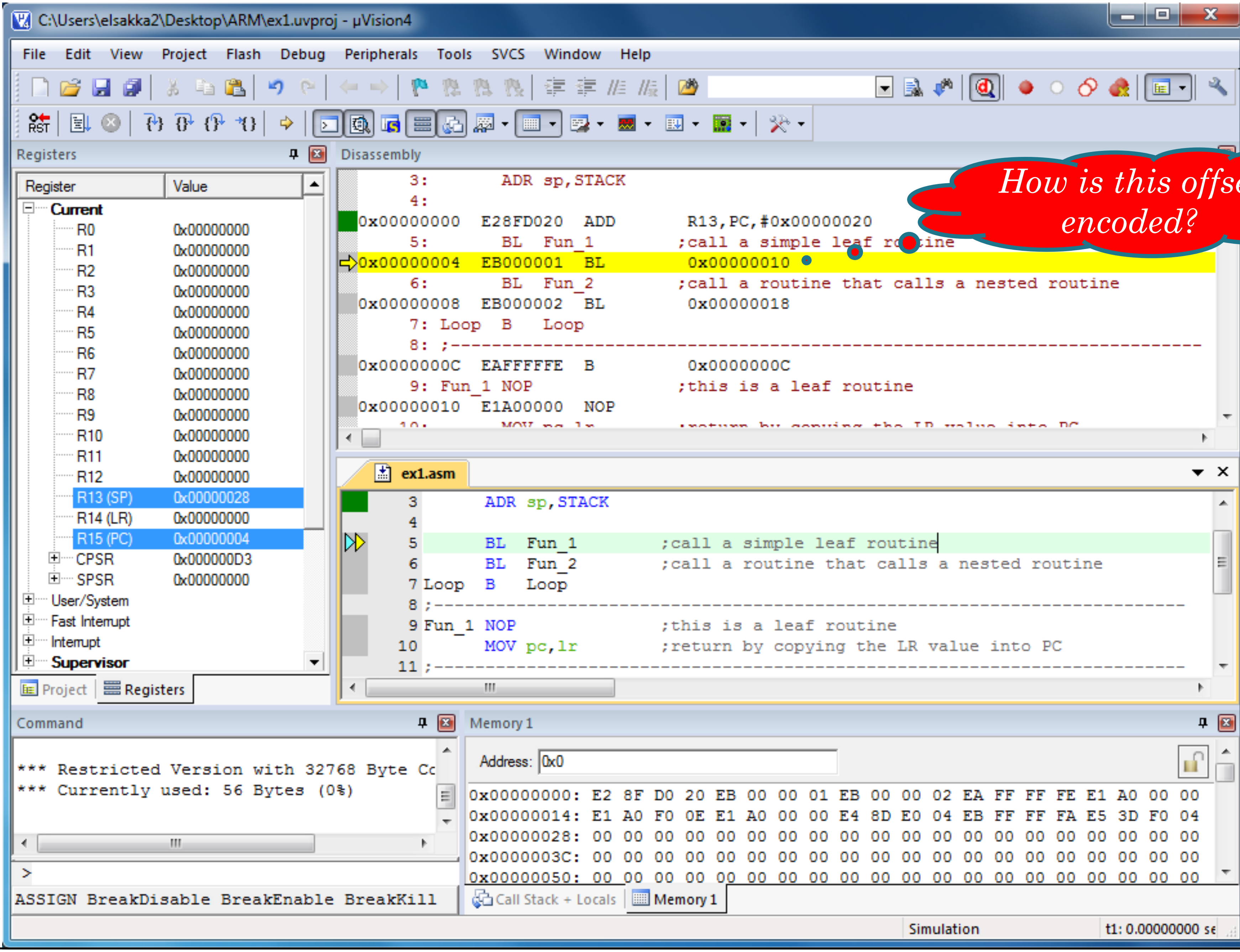
- The simplest way of ***saving*** the link register is to ***push*** it onto the stack.

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

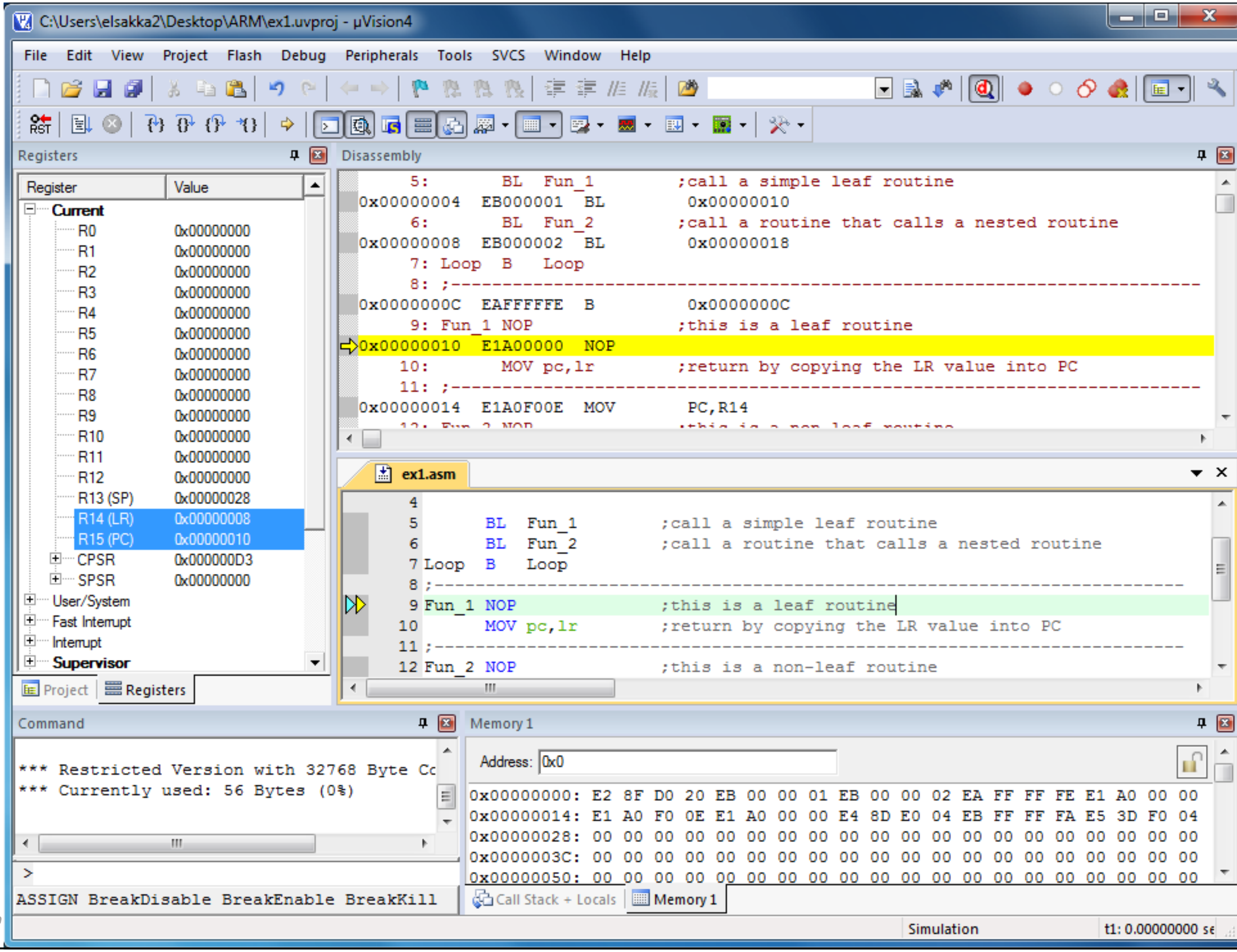
Leaf routines



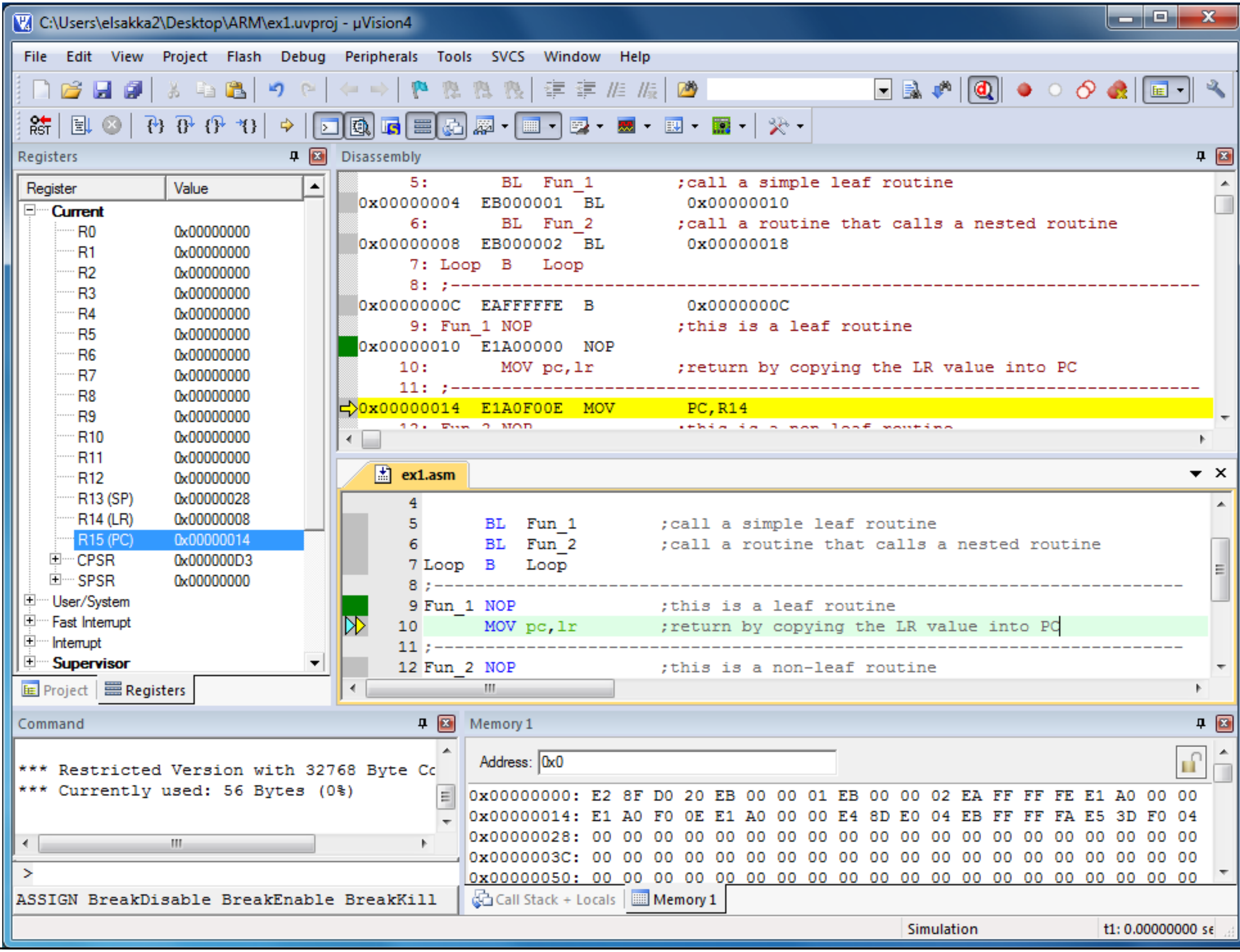
Leaf routines



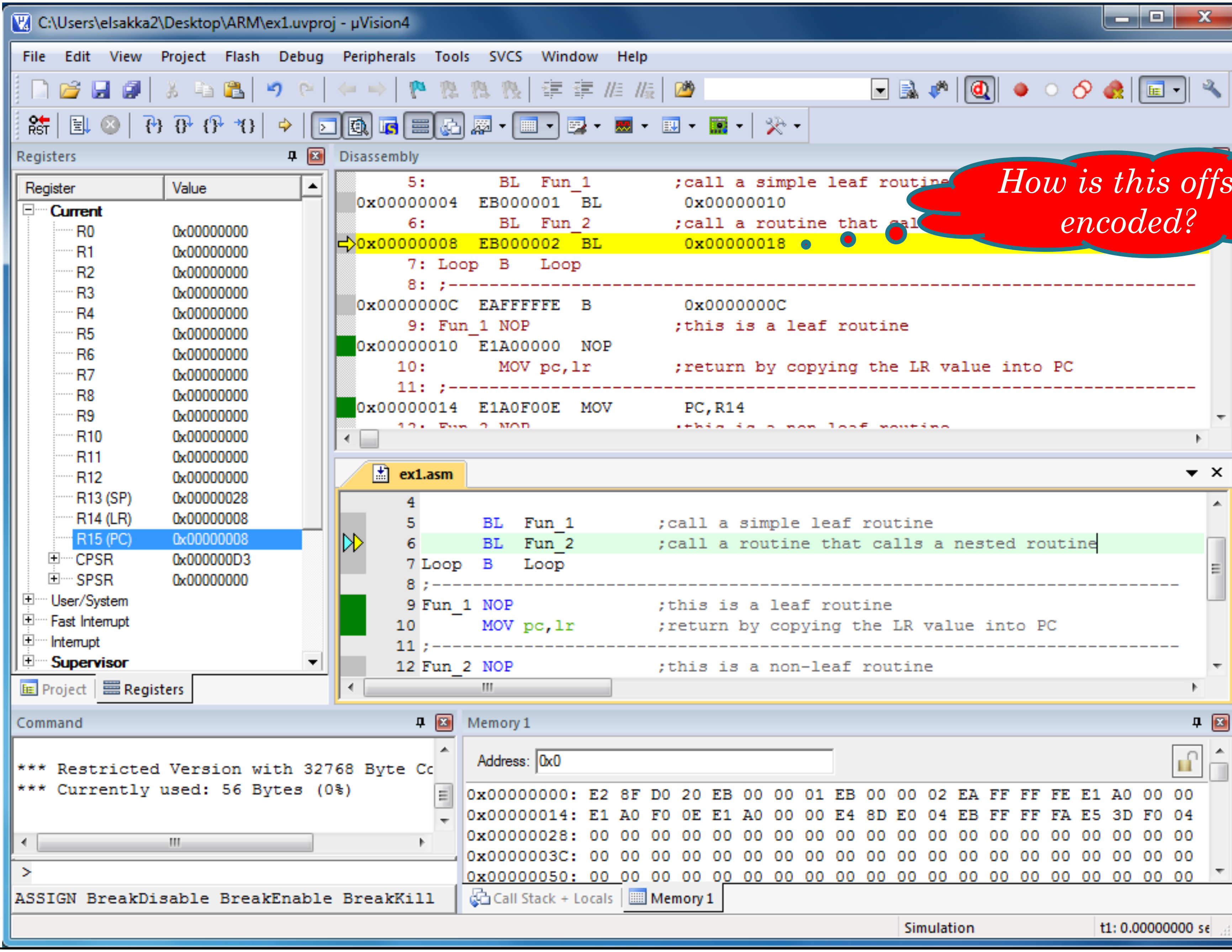
Leaf routines



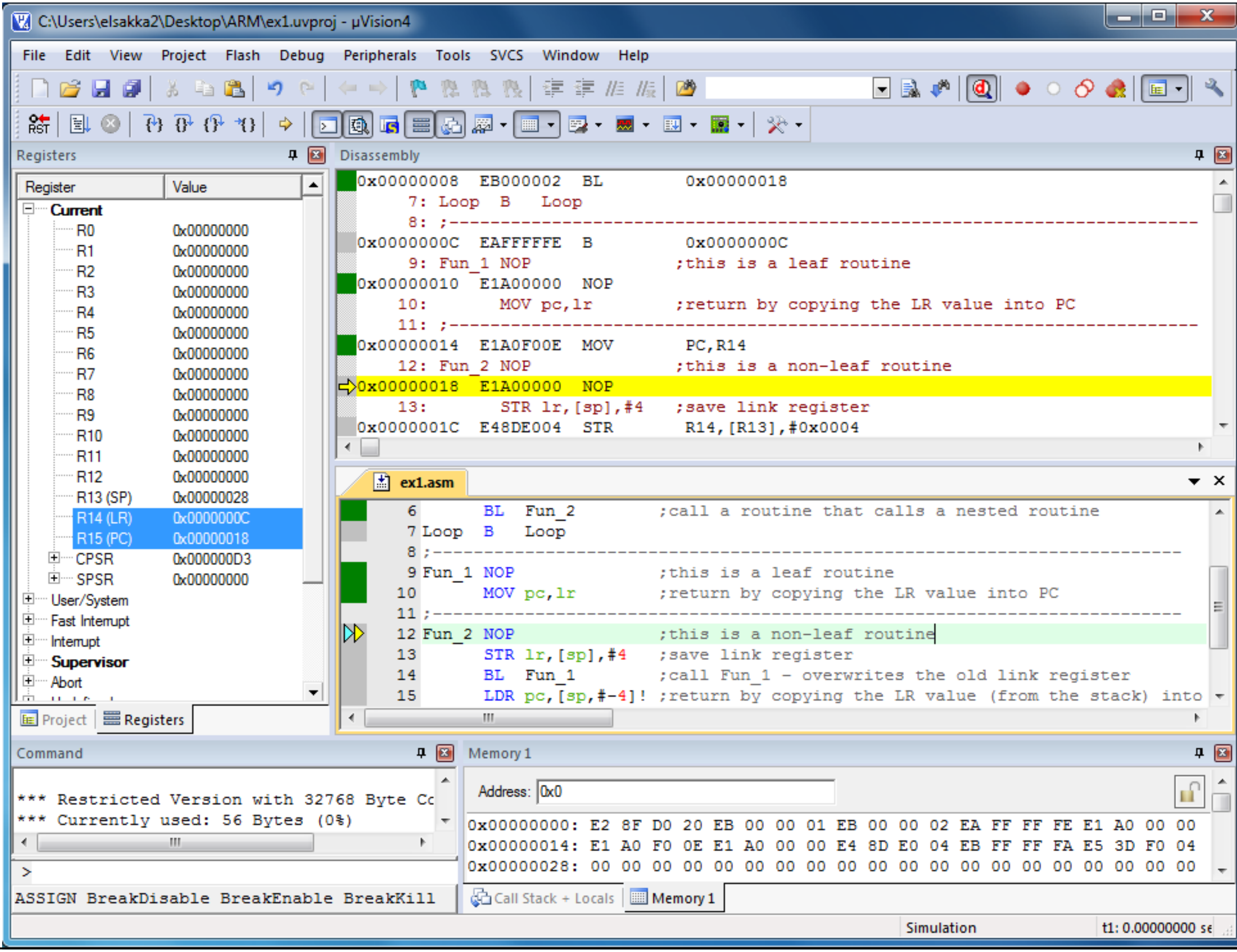
Leaf routines

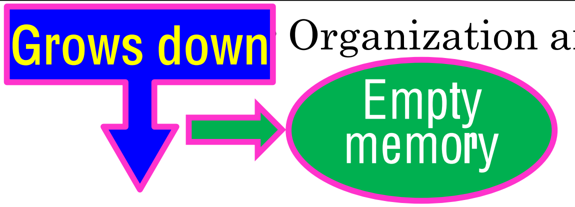


Leaf routines

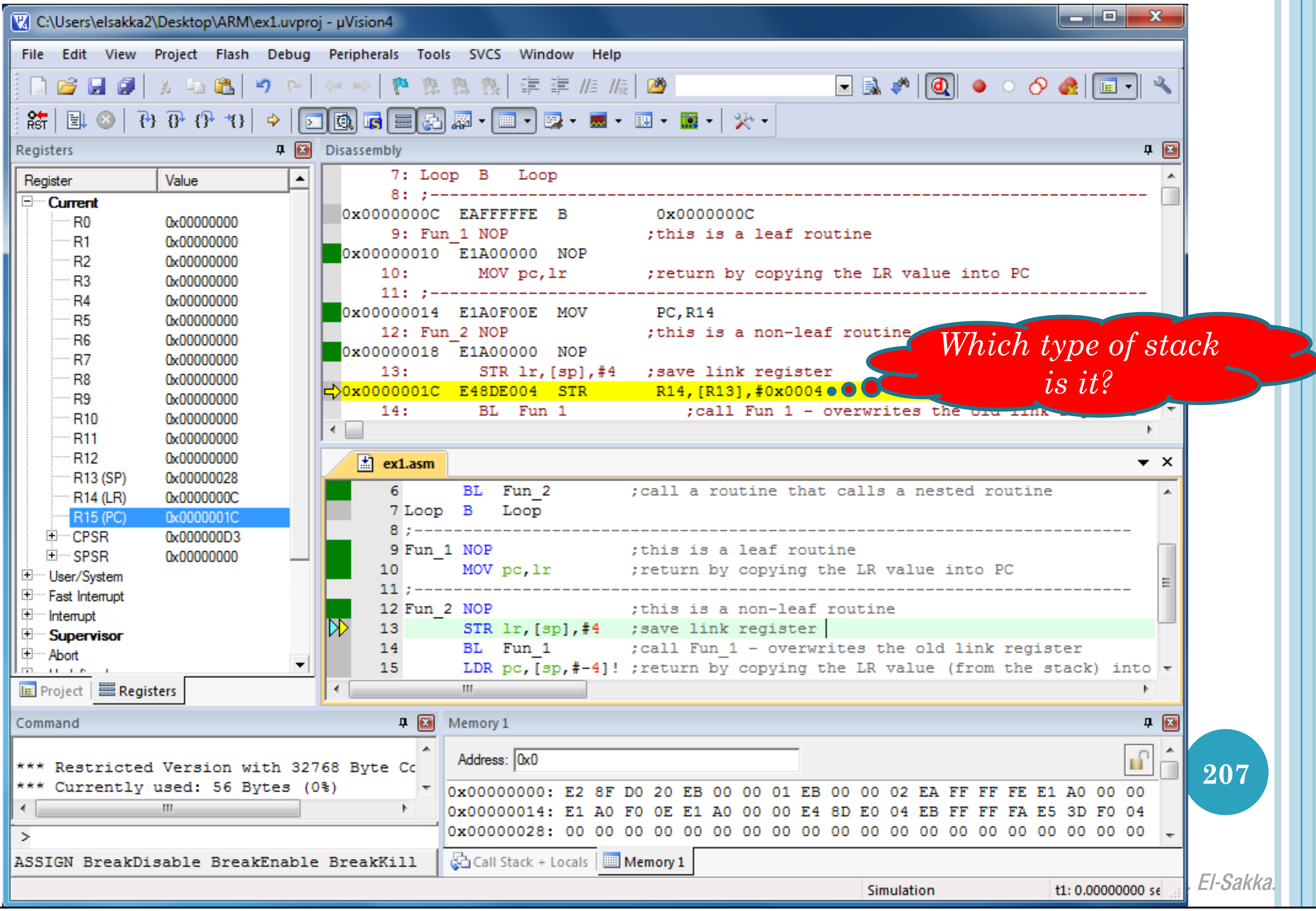


Leaf routines

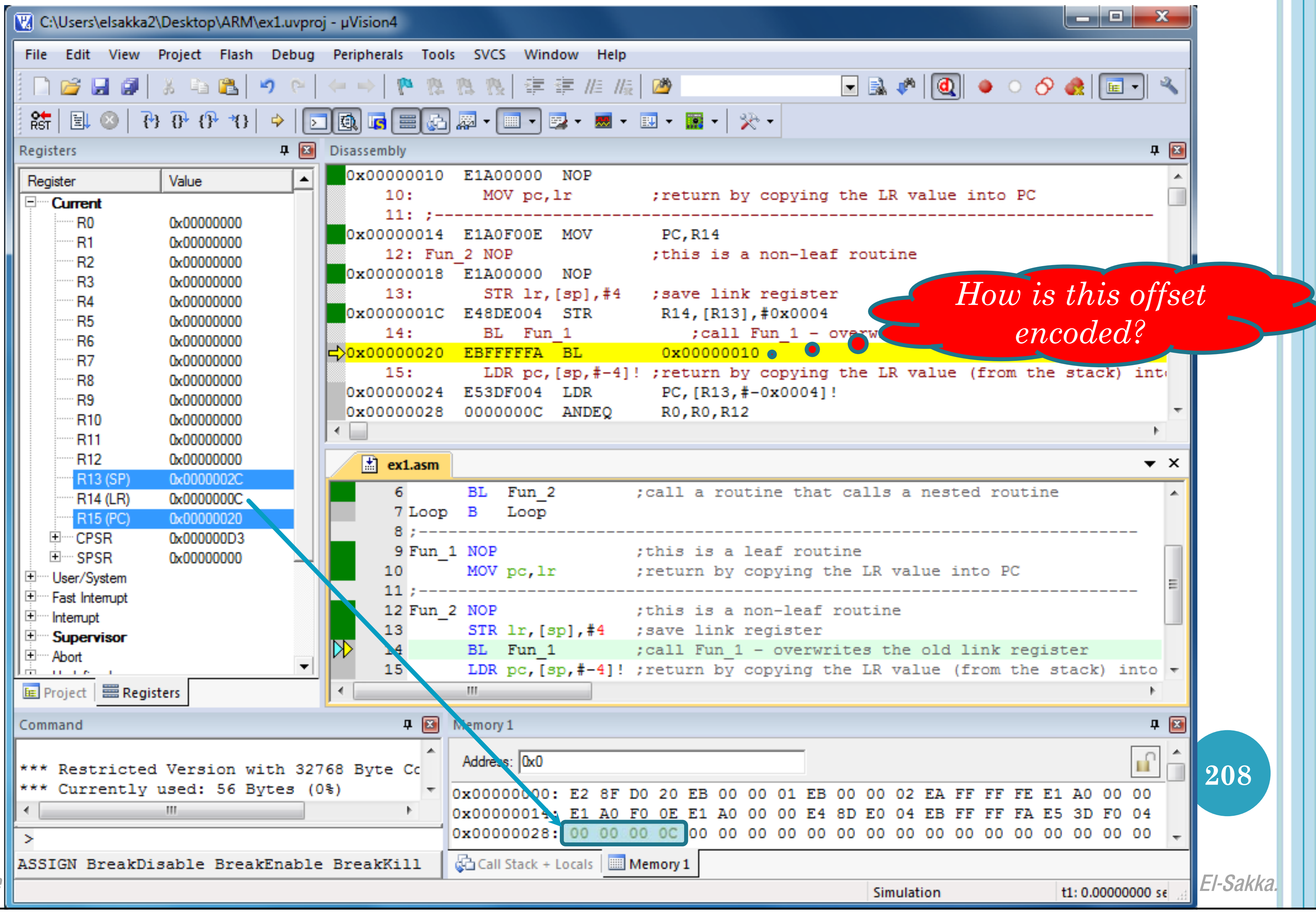




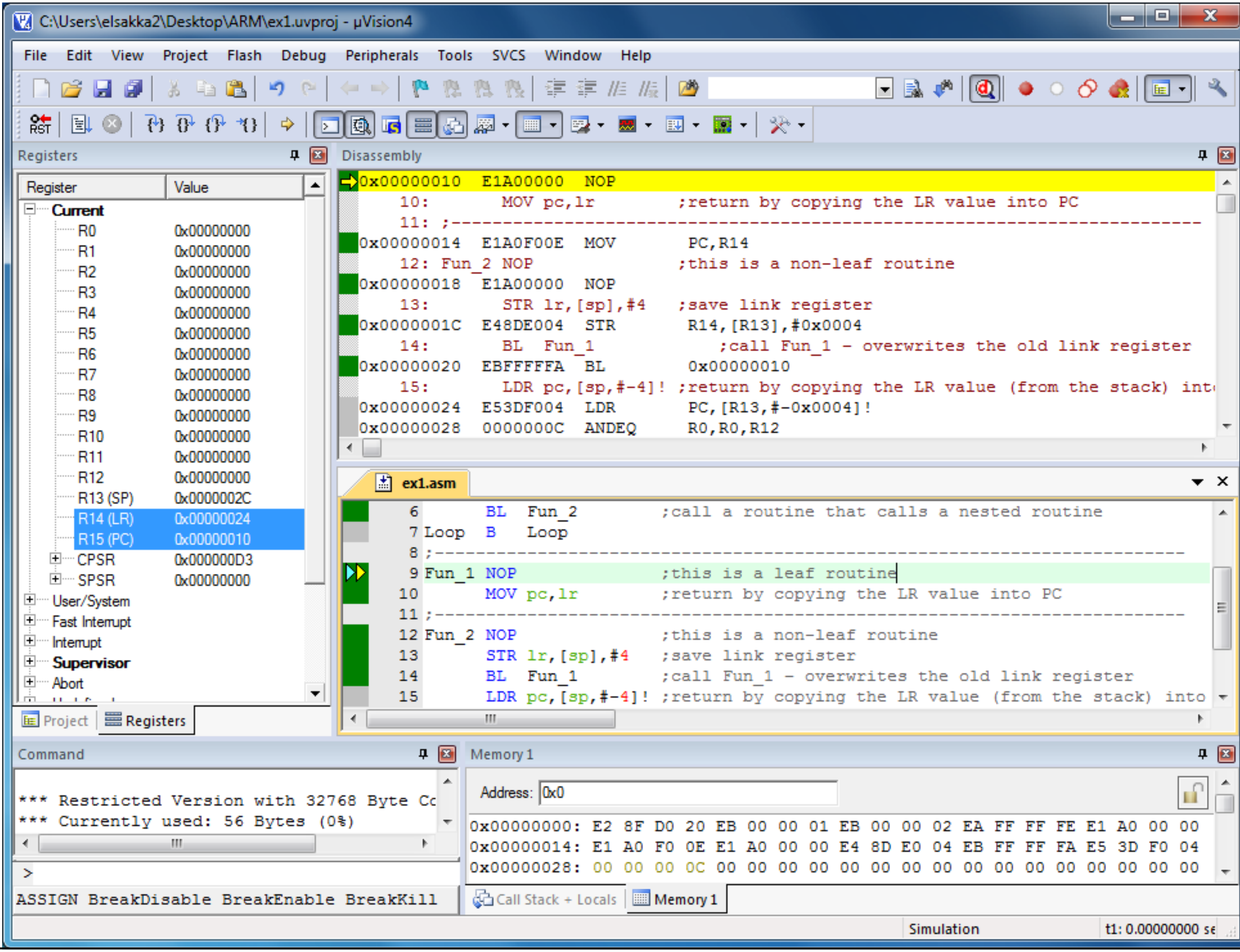
Leaf routines



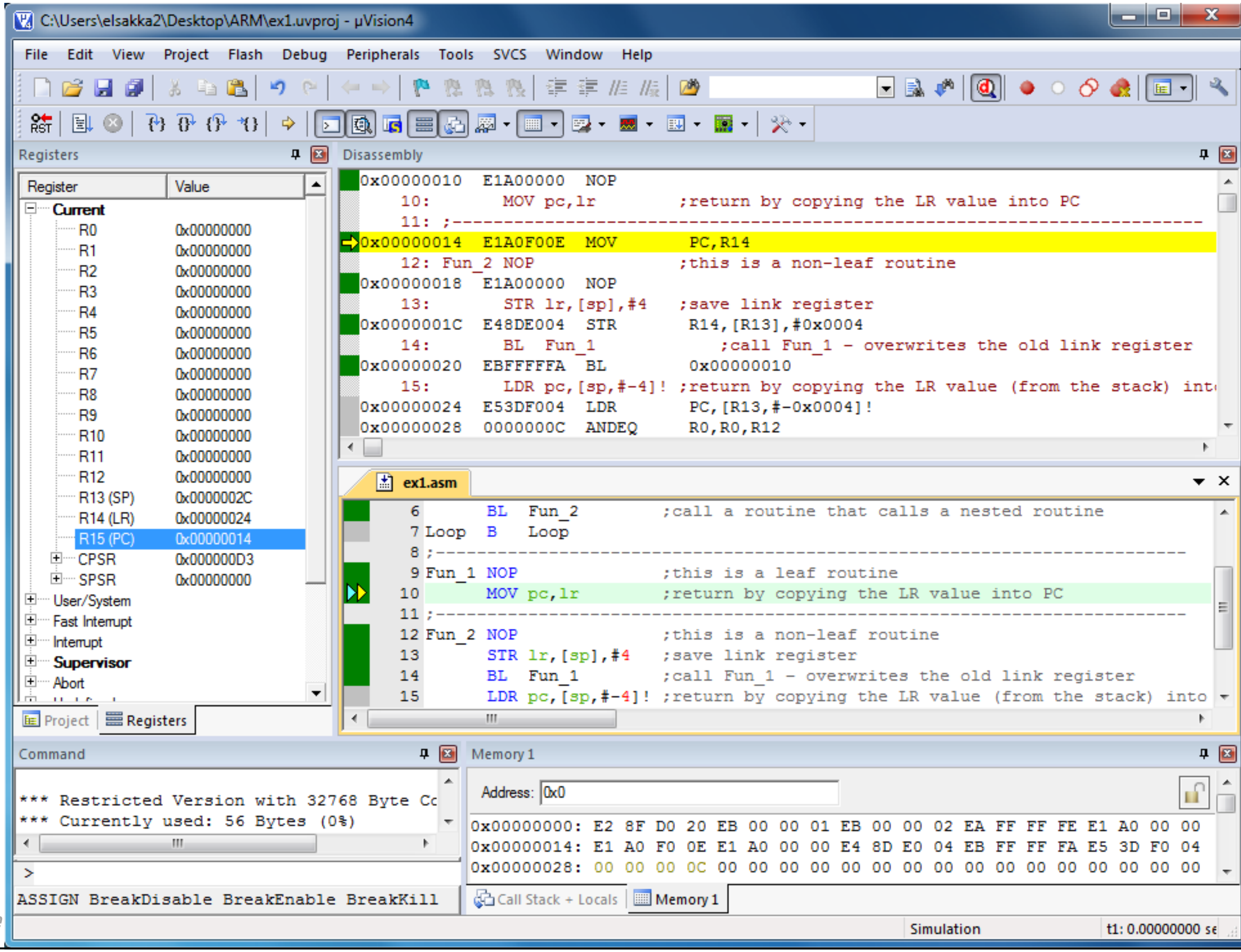
Leaf routines



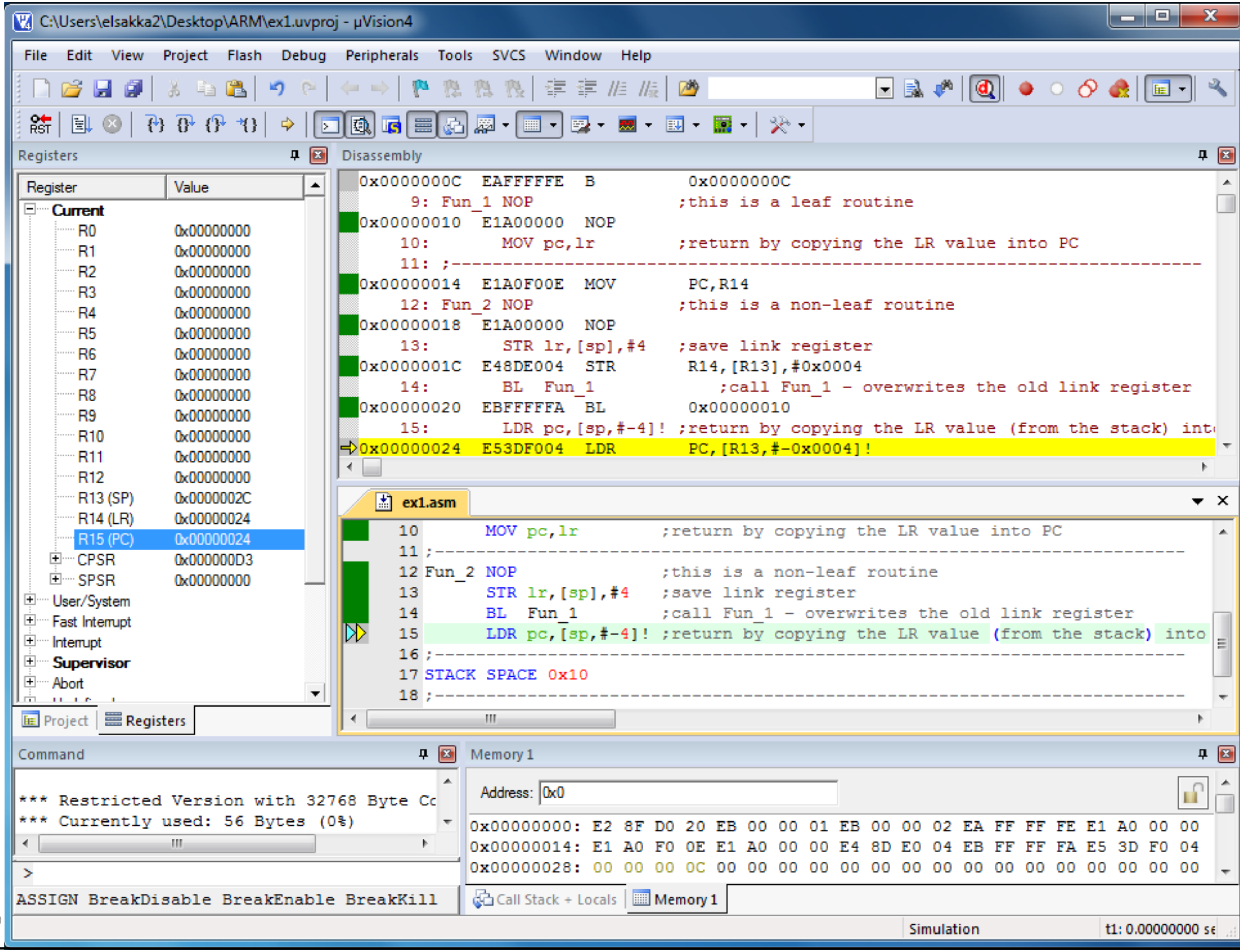
Leaf routines



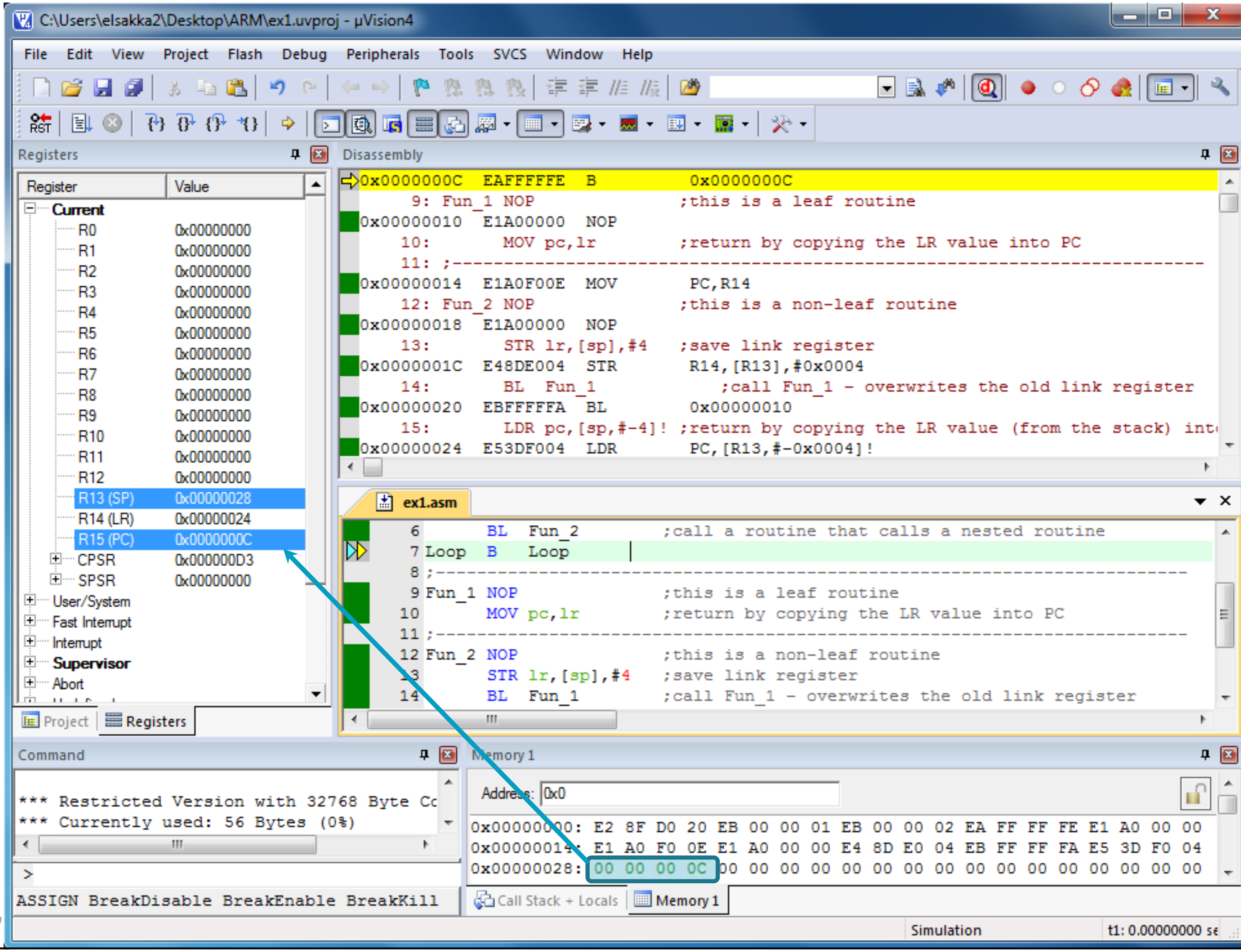
Leaf routines



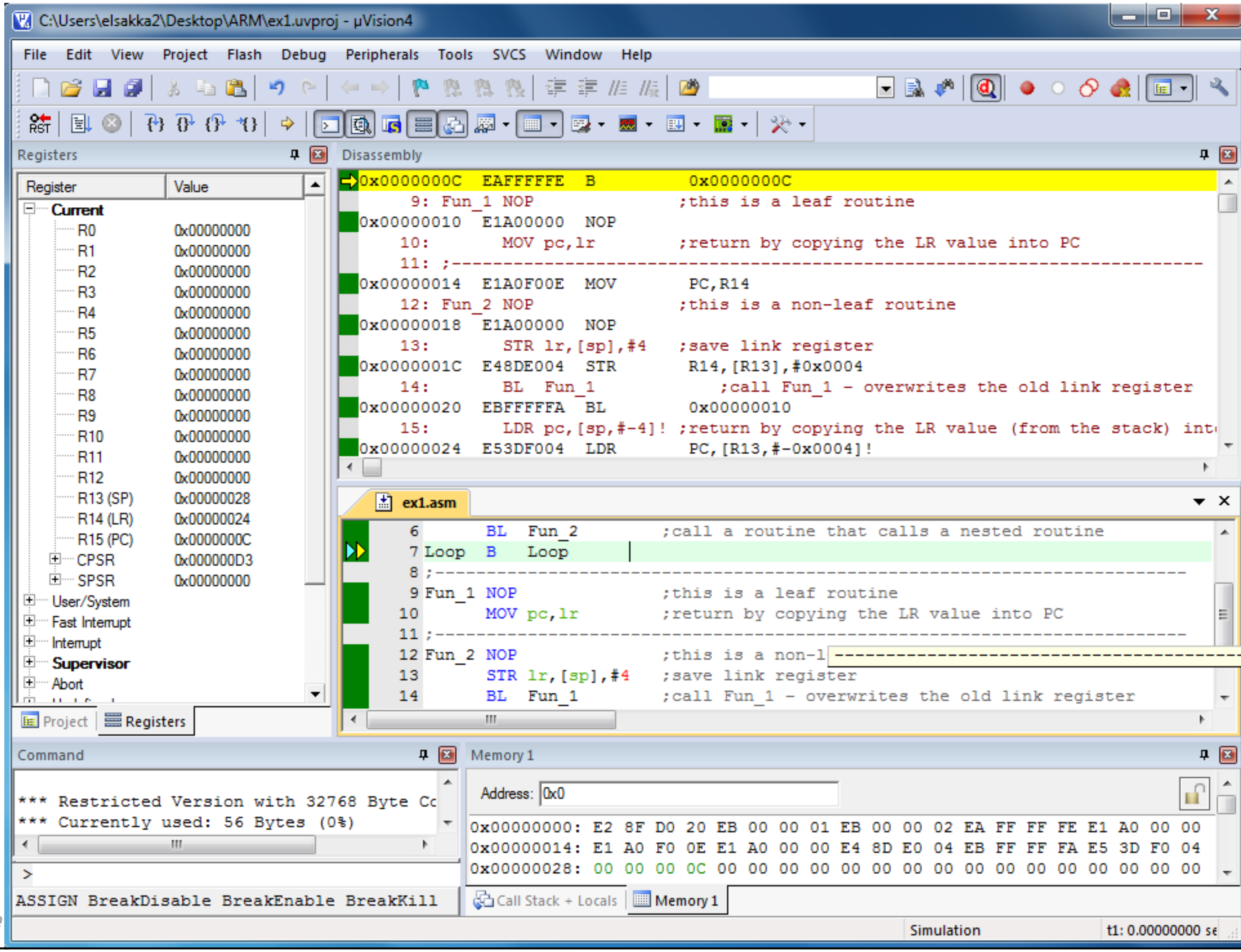
Leaf routines



Leaf routines



Leaf routines



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 on entering a subroutine and
  - restore registers 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
```

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 effect the return.
  - By doing so, we reduced the size of this code by one instruction