THE STACK FRAME – AN EXAMPLE

- In this example we are going to demonstrate how a subroutine is called and a stack frame used to store temporary variables.
- We also demonstrate the passing of parameters by reference and by value.
- Assume that we wish to use a subroutine to calculate $C = A^2 + B^2$.
- The structure of this program in memory will be:
  - Body of program
  - Subroutine
  - Stack area
  - Data area A,B,C

We can express this operation in C as:

```
// C function using both pass by value and reference
void SumSq(int P, int Q, int *R)
{
*R = P*P + Q*Q;
}


// Here’s where we set up the variables and call
SumSq
int main()
{
int A = 5; int B = 7;
SumSq( A,B,&C);
}
```

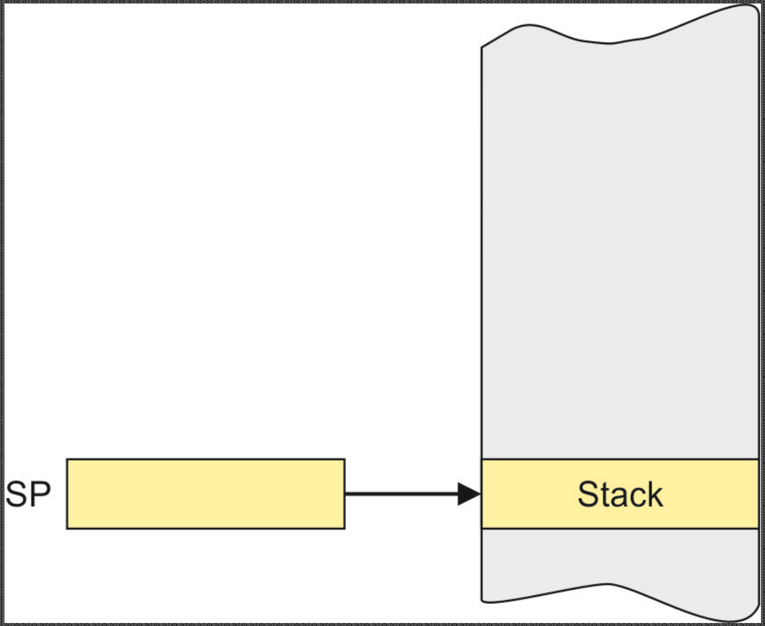
The assembly language code is given below. In this presentation we are going to walk through the code. The code is not optimum and is for demonstration purposes only.

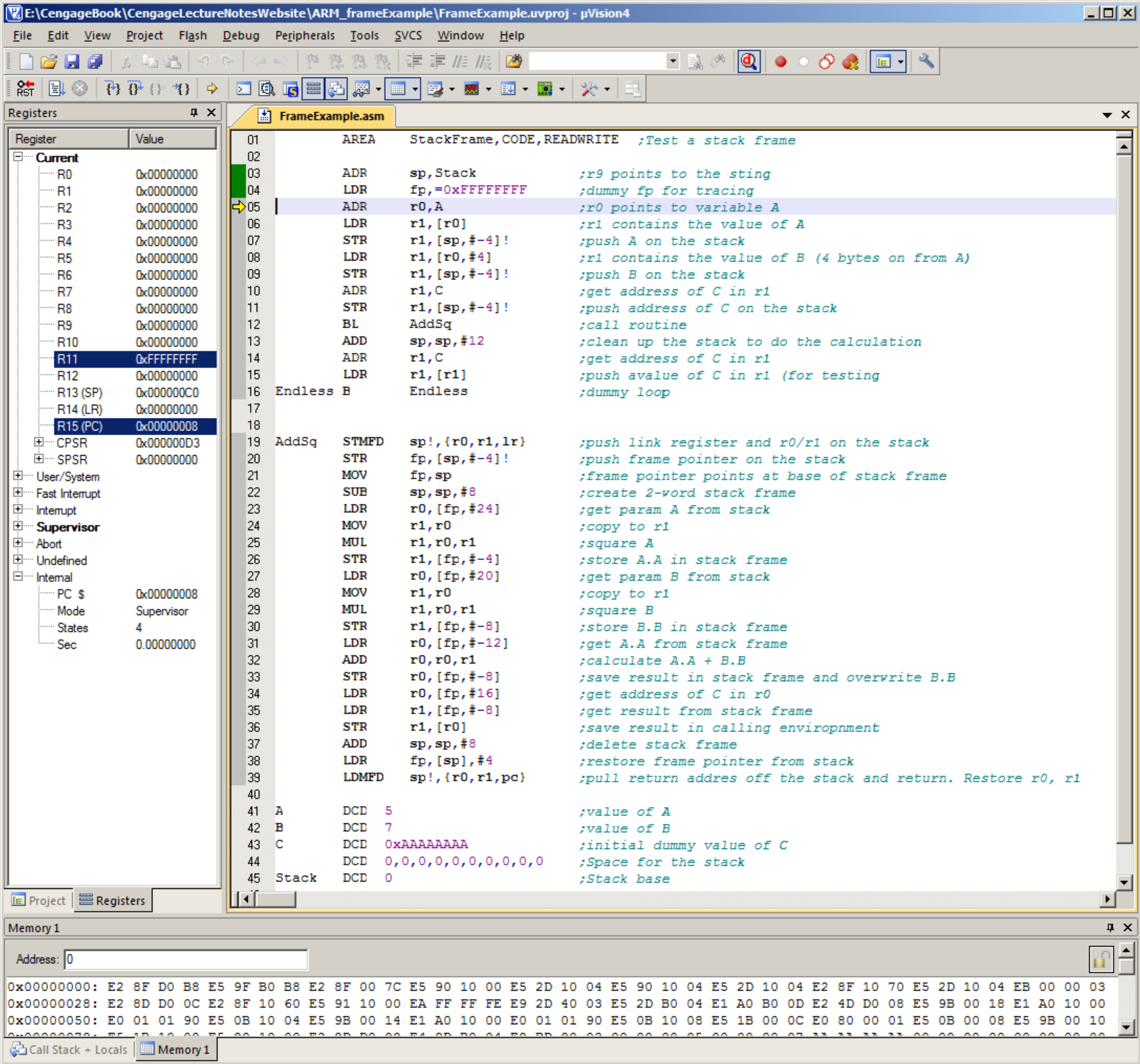
```
         AREA  SF,CODE,READWRITE                    ;Test a stack frame
         ADR   sp,Stack             ;r9 points to the stack
         LDR   fp,=0xFFFFFFFF       ;dummy fp for tracing
         ADR   r0,A                 ;r0 points to variable A
         LDR   r1,[r0]              ;r1 contains the value of A
         STR   r1,[sp,#-4]!         ;push A on the stack
         LDR   r1,[r0,#4]           ;r1 contains the value of B (4 bytes on from A)
         STR   r1,[sp,#-4]!         ;push B on the stack
         ADR   r1,C                 ;get address of C in r1
         STR   r1,[sp,#-4]!         ;push address of C on the stack
         BL    AddSq                ;call routine
         ADD   sp,sp,#12            ;clean up the stack to do the calculation
         ADR   r1,C                 ;get address of C in r1
         LDR   r1,[r1]              ;push a value of C in r1 (for testing)
Endless  B     Endless              ;dummy loop
 AddSq   STMFD sp!,{r0,r1,lr}       ;push link register and r0/r1 on the stack
         STR   fp,[sp,#-4]!         ;push frame pointer on the stack
         MOV   fp,sp                ;frame pointer points at base of stack frame
         SUB   sp,sp,#8             ;create 2-word stack frame
         LDR   r0,[fp,#24]          ;get param A from stack
         MOV   r1,r0                ;copy to r1
         MUL   r1,r0,r1             ;square A
         STR   r1,[fp,#-4]          ;store A.A in stack frame
         LDR   r0,[fp,#20]          ;get param B from stack
         MOV   r1,r0                ;copy to r1
         MUL   r1,r0,r1             ;square B
         STR   r1,[fp,#-8]          ;store B.B in stack frame
         LDR   r0,[fp,#-12]         ;get A.A from stack frame
         ADD   r0,r0,r1             ;calculate A.A + B.B
         STR   r0,[fp,#-8]          ;save result in stack frame and overwrite B.B
         LDR   r0,[fp,#16]          ;get address of C in r0
         LDR   r1,[fp,#-8]          ;get result from stack frame
         STR   r1,[r0]              ;save result in calling environment
         ADD   sp,sp,#8             ;delete stack frame
         LDR   fp,[sp],#4           ;restore frame pointer from stack
         LDMFD sp!,{r0,r1,pc}       ;pull return address off the stack. Return. Restore r0, r1
A        DCD  5 ;value of A
B        DCD  7 ;value of B
C        DCD  0xAAAAAAAA            ;initial dummy value of C
         DCD  0,0,0,0,0,0,0,0,0     ;Space for the stack
Stack    DCD  0                     ;Stack base
         END
```

The first three lines define the storage area and set up the stack. We do this by loading the address of the stack area with the pseudoinstruction ADR. We also set up the frame pointer with the dummy value 0xFFFFFFFF.

```
AREA  SF,CODE,READWRITE         ;Test a stack frame
ADR   sp,Stack                  ;r9 points to the stack
LDR   fp,=0xFFFFFFFF            ;dummy fp for tracing
```

Why is the frame pointer loaded with 0xFFFFFFFF? We don't need to do this, but we will be able to see it in memory when we debug the program. It's a marker.





Register	Value
Current	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0xFFFFFFFF
R12	0x00000000
R13 (SP)	0x000000C0
R14 (LR)	0x00000000
R15 (PC)	0x00000008
CPSR	0x000000D3
SPSR	0x00000000
User/System	
Fast Interrupt	
Interrupt	
Supervisor	
Abort	
Undefined	
Internal	
PC $	0x00000008
Mode	Supervisor
States	4
Sec	0.00000000

```
        AREA    StackFrame,CODE,READWRITE  ;Test a stack frame

        ADR     sp,Stack            ;r9 points to the sting
        LDR     fp,=0xFFFFFFFF      ;dummy fp for tracing
        ADR     r0,A                ;r0 points to variable A
        LDR     r1,[r0]             ;r1 contains the value of A
        STR     r1,[sp,#-4]!        ;push A on the stack
        LDR     r1,[r0,#4]          ;r1 contains the value of B (4 bytes on from A)
        STR     r1,[sp,#-4]!        ;push B on the stack
        ADR     r1,C                ;get address of C in r1
        STR     r1,[sp,#-4]!        ;push address of C on the stack
        BL      AddSq               ;call routine
        ADD     sp,sp,#12           ;clean up the stack to do the calculation
        ADR     r1,C                ;get address of C in r1
        LDR     r1,[r1]             ;push avalue of C in r1 (for testing
Endless B       Endless             ;dummy loop


AddSq   STMFD   sp!,{r0,r1,lr}      ;push link register and r0/r1 on the stack
        STR     fp,[sp,#-4]!        ;push frame pointer on the stack
        MOV     fp,sp               ;frame pointer points at base of stack frame
        SUB     sp,sp,#8            ;create 2-word stack frame
        LDR     r0,[fp,#24]         ;get param A from stack
        MOV     r1,r0               ;copy to r1
        MUL     r1,r0,r1            ;square A
        STR     r1,[fp,#-4]         ;store A.A in stack frame
        LDR     r0,[fp,#20]         ;get param B from stack
        MOV     r1,r0               ;copy to r1
        MUL     r1,r0,r1            ;square B
        STR     r1,[fp,#-8]         ;store B.B in stack frame
        LDR     r0,[fp,#-12]        ;get A.A from stack frame
        ADD     r0,r0,r1            ;calculate A.A + B.B
        STR     r0,[fp,#-8]         ;save result in stack frame and overwrite B.B
        LDR     r0,[fp,#16]         ;get address of C in r0
        LDR     r1,[fp,#-8]         ;get result from stack frame
        STR     r1,[r0]             ;save result in calling enviropnment
        ADD     sp,sp,#8            ;delete stack frame
        LDR     fp,[sp],#4          ;restore frame pointer from stack
        LDMFD   sp!,{r0,r1,pc}      ;pull return addres off the stack and return. Restore r0, r1

A       DCD     5                   ;value of A
B       DCD     7                   ;value of B
C       DCD     0xAAAAAAAA          ;initial dummy value of C
        DCD     0,0,0,0,0,0,0,0,0,0 ;Space for the stack
Stack   DCD     0                   ;Stack base
```

Address: 0

```
0x00000000: E2 8F D0 B8 E5 9F B0 B8 E2 8F 00 7C E5 90 10 00 E5 2D 10 04 E5 90 10 04 E5 2D 10 04 E2 8F 10 70 E5 2D 10 04 EB 00 00 03
0x00000028: E2 8D D0 0C E2 8F 10 60 E5 91 10 00 EA FF FF FE E9 2D 40 03 E5 2D B0 04 E1 A0 B0 0D E2 4D D0 08 E5 9B 00 18 E1 A0 10 00
0x00000050: E0 01 01 90 E5 0B 10 04 E5 9B 00 14 E1 A0 10 00 E0 01 01 90 E5 0B 10 08 E5 1B 00 0C E0 80 00 01 E5 0B 00 08 E5 9B 00 10
```

This is the state of the system after the frame pointer has been loaded with a dummy value.

Here’s some background before we continue.

The store operation `STR reg,[pointer]` stores a register in the memory location defined by pointer (which is also a register).

`STR r0,[r1]` stores the contents of register r0 in the memory location pointed at by register r1.

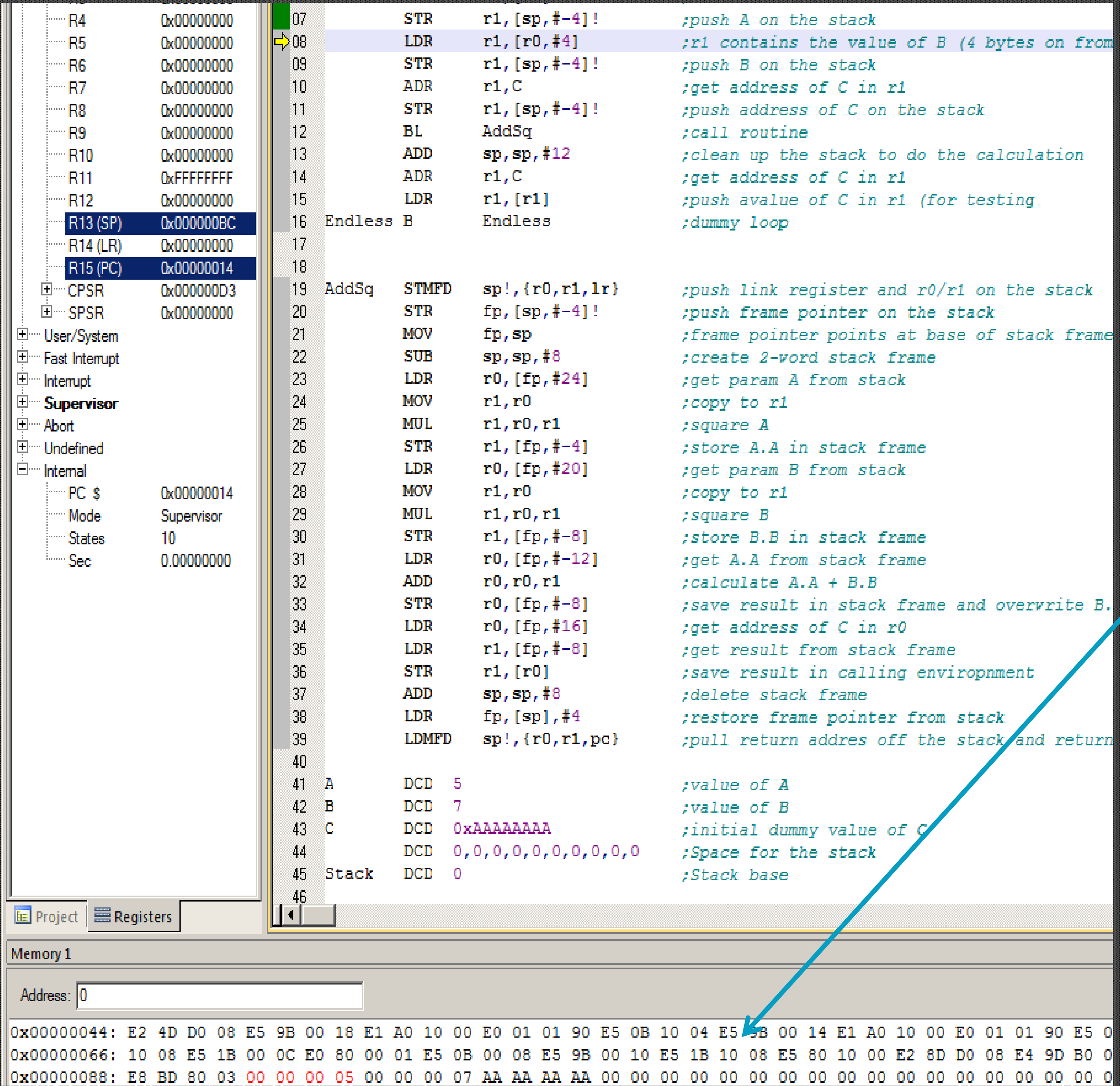
`STR r0,[sp]` stores the contents of register r0 in the memory location pointed at by the stack pointer. The stack pointer can be written r13 or sp.

To push a register on the stack using a **full descending** stack, we have to first predecrement the stack pointer by a word (4 bytes) before performing the move. We can do this with

```
STR r0,[sp,#-4]!
```

The #-4 means subtract 4 from the stack pointer before using it, and the ! indicates that the change in the stack pointer is to be kept.

This operation is equivalent to `PUSH r0`.



```
R4          0x00000000
R5          0x00000000
R6          0x00000000
R7          0x00000000
R8          0x00000000
R9          0x00000000
R10         0x00000000
R11         0xFFFFFFFF
R12         0x00000000
R13 (SP)    0x000000BC
R14 (LR)    0x00000000
R15 (PC)    0x00000014
CPSR        0x000000D3
SPSR        0x00000000
User/System
Fast Interrupt
Interrupt
Supervisor
Abort
Undefined
Internal
PC  $       0x00000014
Mode        Supervisor
States      10
Sec         0.00000000
```

```
         STR    r1,[sp,#-4]!           ;push A on the stack
         LDR    r1,[r0,#4]             ;r1 contains the value of B (4 bytes on from
         STR    r1,[sp,#-4]!           ;push B on the stack
         ADR    r1,C                   ;get address of C in r1
         STR    r1,[sp,#-4]!           ;push address of C on the stack
         BL     AddSq                  ;call routine
         ADD    sp,sp,#12              ;clean up the stack to do the calculation
         ADR    r1,C                   ;get address of C in r1
         LDR    r1,[r1]                ;push avalue of C in r1 (for testing
Endless  B      Endless                ;dummy loop


AddSq    STMFD  sp!,{r0,r1,lr}         ;push link register and r0/r1 on the stack
         STR    fp,[sp,#-4]!           ;push frame pointer on the stack
         MOV    fp,sp                  ;frame pointer points at base of stack frame
         SUB    sp,sp,#8               ;create 2-word stack frame
         LDR    r0,[fp,#24]            ;get param A from stack
         MOV    r1,r0                  ;copy to r1
         MUL    r1,r0,r1               ;square A
         STR    r1,[fp,#-4]            ;store A.A in stack frame
         LDR    r0,[fp,#20]            ;get param B from stack
         MOV    r1,r0                  ;copy to r1
         MUL    r1,r0,r1               ;square B
         STR    r1,[fp,#-8]            ;store B.B in stack frame
         LDR    r0,[fp,#-12]           ;get A.A from stack frame
         ADD    r0,r0,r1               ;calculate A.A + B.B
         STR    r0,[fp,#-8]            ;save result in stack frame and overwrite B.
         LDR    r0,[fp,#16]            ;get address of C in r0
         LDR    r1,[fp,#-8]            ;get result from stack frame
         STR    r1,[r0]                ;save result in calling environpnment
         ADD    sp,sp,#8               ;delete stack frame
         LDR    fp,[sp],#4             ;restore frame pointer from stack
         LDMFD  sp!,{r0,r1,pc}         ;pull return addres off the stack and return

A        DCD  5                        ;value of A
B        DCD  7                        ;value of B
C        DCD  0xAAAAAAAA               ;initial dummy value of C
         DCD  0,0,0,0,0,0,0,0,0,0      ;Space for the stack
Stack    DCD  0                        ;Stack base
```

Memory 1

Address: 0

```
0x00000044: E2 4D D0 08 E5 9B 00 18 E1 A0 10 00 E0 01 01 90 E5 0B 10 04 E5 9B 00 14 E1 A0 10 00 E0 01 01 90 E5 0
0x00000066: 10 08 E5 1B 00 0C E0 80 00 01 E5 0B 00 08 E5 9B 00 10 E5 1B 10 08 E5 80 10 00 E2 8D D0 08 E4 9D B0 0
0x00000088: E8 BD 80 03 00 00 00 05 00 00 00 07 AA AA AA AA 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
```

The stack has been set up and the value of A pushed on the stack.

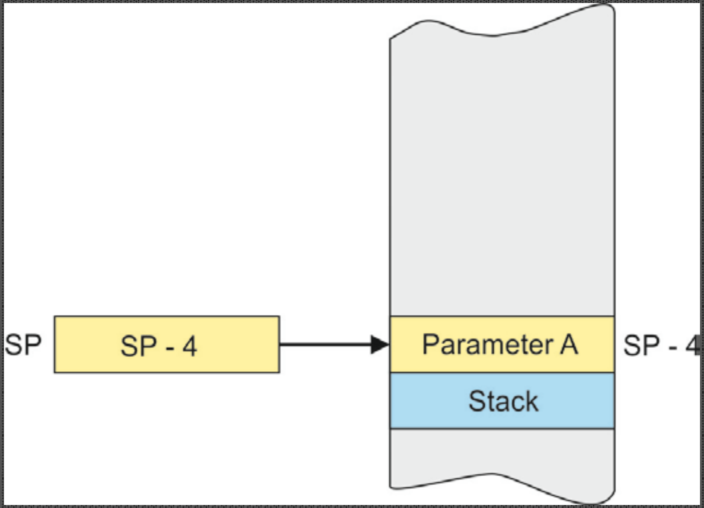
This can be seen in the memory may (i.e., 5).

The next step is to push the three parameters on the stack, A, B and the address of C. Let’s begin with A.

```
ADR   r0,A          ;r0 points to variable A
LDR   r1,[r0]       ;r1 contains the value of A
STR   r1,[sp,#-4]! ;push A on the stack
```

Note that we first have to load r0 with the address of A, then load the value of A into register r1, and then finally push the contents of r1 on the stack with `STR   r1,[sp,#-4]!`.

The figure below shows the state of the memory at the end of this sequence.



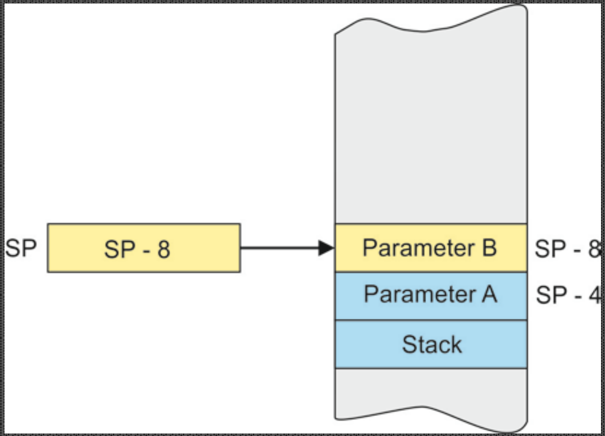
The next step is to push parameter B on the stack. We could load the address of B into a register and use it as a pointer.

However, we set up the data by means of the following directive

```
A        DCD  5  ;value of A
B        DCD  7  ;value of B
C        DCD  0xAAAAAAAA       ;initial dummy value of C
```

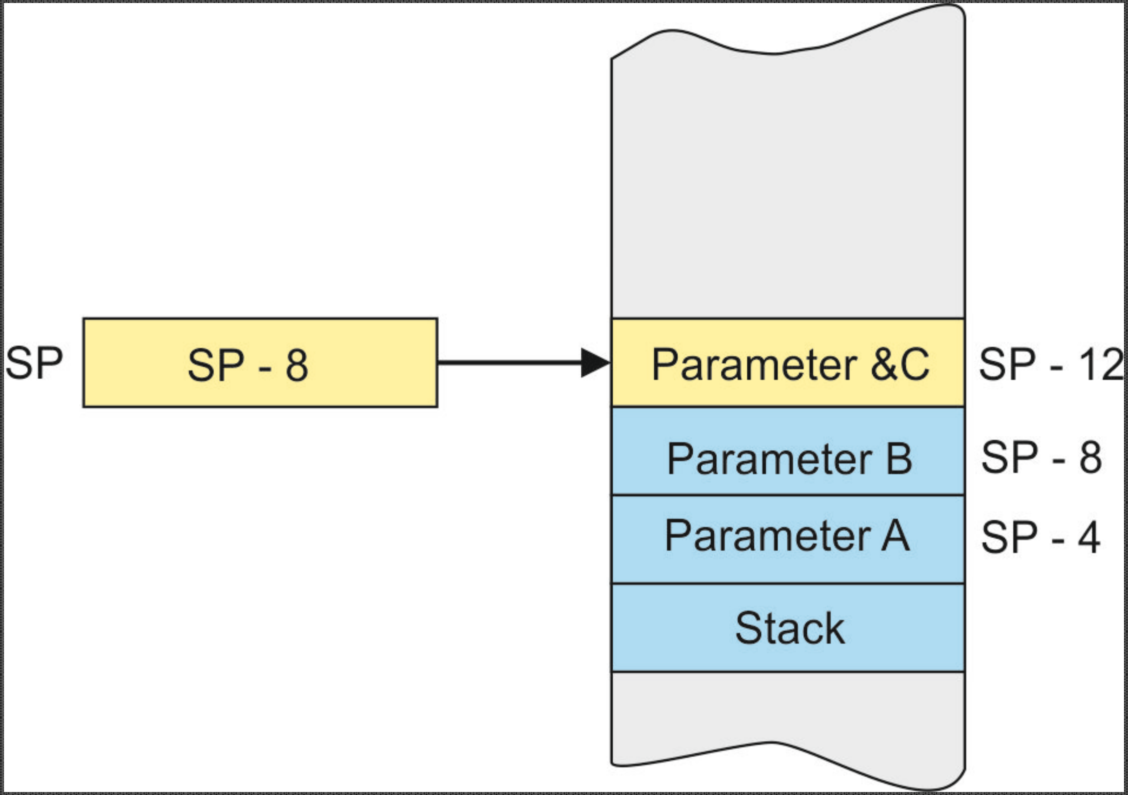
The location of B is 4 bytes from A, so were can use the address of A as a pointer to B by adding 4 (i.e., using the address [r0,#4]). Note that we have given C the initial value 0xAAAAAAAA. As before this makes it easy to trace the program. The code to push B is as follows.

```
        LDR   r1,[r0,#4]  ;r1 contains the value of B (4 bytes on from A)
        STR   r1,[sp,#-4]!   ;push B on the stack
```



The next step is to push the address of parameter C. We can get it by using the ADR (load address) pseudoinstruction to put the address of C in r0 and then push that address on the stack as follows.

```
ADR   r1,C            ;get address of C in r1
STR   r1,[sp,#-4]! ;push address of C on the stack
```



This is the data area in the program.

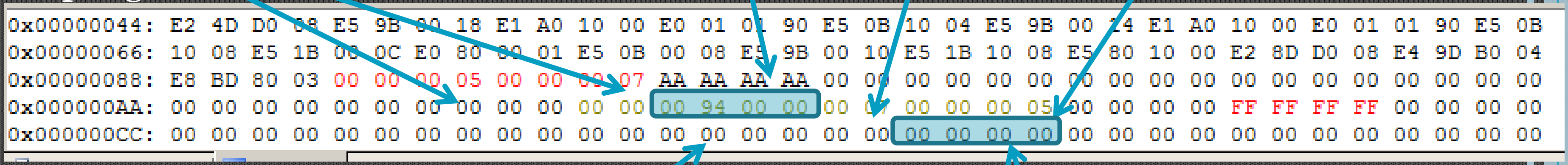
```
A       DCD  5                    ;value of A
B       DCD  7                    ;value of B
C       DCD  0xAAAAAAAA           ;dummy value of C
        DCD  0,0,0,0,0,0,0,0,0,0  ;space for the stack
Stack   DCD                       ;stack base
```

The value of B on the stack (7).

The value of A on the stack (5).

These are the values of A and B in memory at the end of the program.

The 0xAAAAAAA marker (i.e., the dummy value of C).



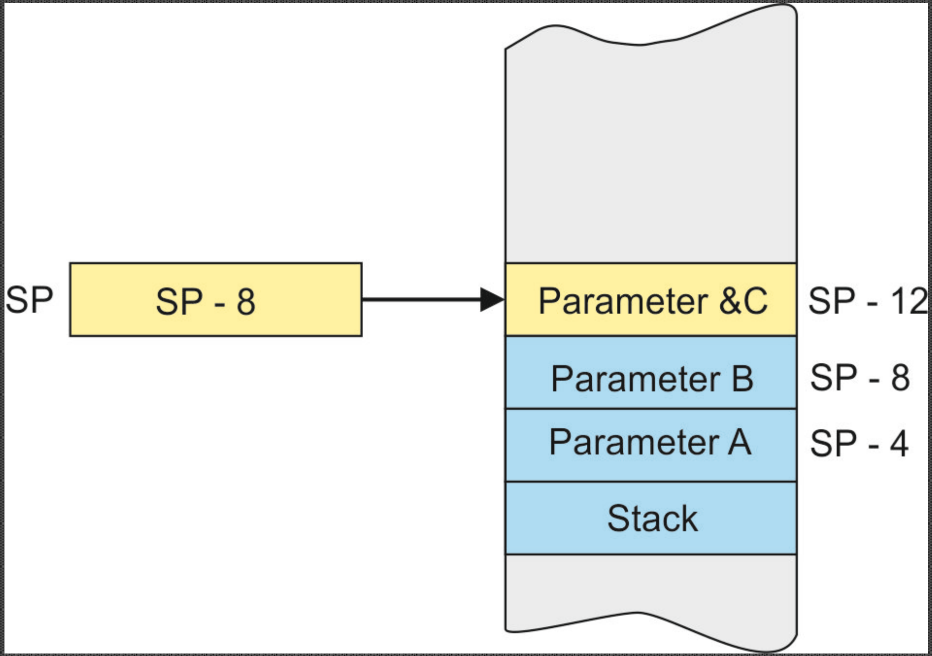
```
0x00000044: E2 4D D0 08 E5 9B 00 18 E1 A0 10 00 E0 01 01 90 E5 0B 10 04 E5 9B 00 14 E1 A0 10 00 E0 01 01 90 E5 0B
0x00000066: 10 08 E5 1B 00 0C E0 80 00 01 E5 0B 00 08 E5 9B 00 10 E5 1B 10 08 E5 80 10 00 E2 8D D0 08 E4 9D B0 04
0x00000088: E8 BD 80 03 00 00 00 05 00 00 00 07 AA AA AA AA 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000000AA: 00 00 00 00 00 00 00 00 00 00 00 00 00 94 00 00 00 07 00 00 00 05 00 00 00 00 FF FF FF FF 00 00 00 00
0x000000CC: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The address of C on the stack (0x00000094).

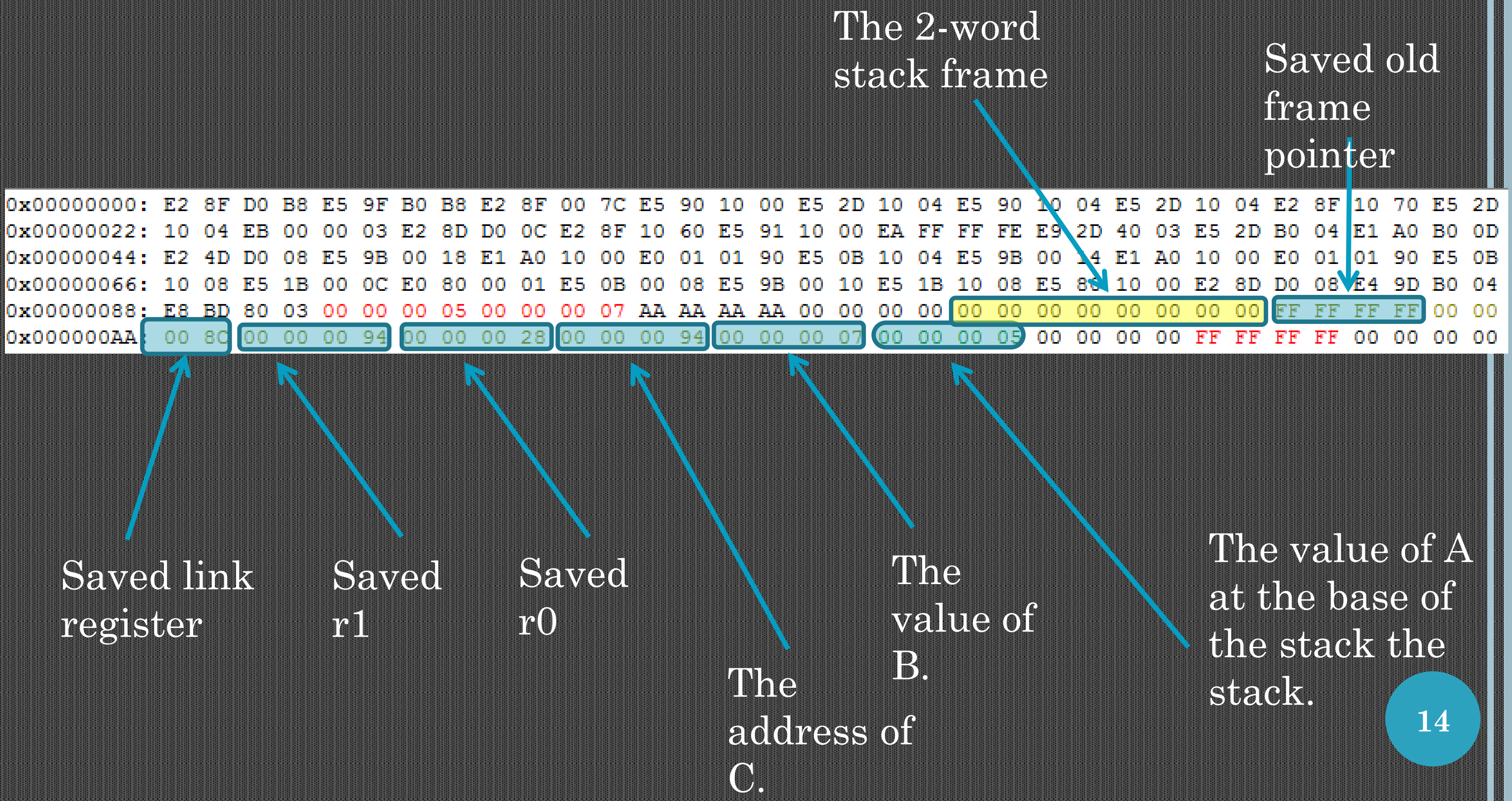
The first word at the bottom of the stack (the value of A)

We have now used the following block of code to push A, B, and &C on the stack. The next step is to call the function.

```
ADR   r0,A             ;r0 points to variable A
LDR   r1,[r0]          ;r1 contains the value of A
STR   r1,[sp,#-4]!     ;push A on the stack
LDR   r1,[r0,#4]       ;r1 contains the value of B (4 bytes on from A)
STR   r1,[sp,#-4]!     ;push B on the stack
ADR   r1,C             ;get address of C in r1
STR   r1,[sp,#-4]!     ;push address of C on the stack
```



This memory map shows the situation after A, B, and &C have been pushed on the stack, registers r0,r1, and the link register saved. The old frame pointer 0xFFFFFFFF at the base of the stack acts as a marker.



```
0x00000000: E2 8F D0 B8 E5 9F B0 B8 E2 8F 00 7C E5 90 10 00 E5 2D 10 04 E5 90 10 04 E5 2D 10 04 E2 8F 10 70 E5 2D
0x00000022: 10 04 EB 00 00 03 E2 8D D0 0C E2 8F 10 60 E5 91 10 00 EA FF FF FE E9 2D 40 03 E5 2D B0 04 E1 A0 B0 0D
0x00000044: E2 4D D0 08 E5 9B 00 18 E1 A0 10 00 E0 01 01 90 E5 0B 10 04 E5 9B 00 14 E1 A0 10 00 E0 01 01 90 E5 0B
0x00000066: 10 08 E5 1B 00 0C E0 80 00 01 E5 0B 00 08 E5 9B 00 10 E5 1B 10 08 E5 8B 10 00 E2 8D D0 08 E4 9D B0 04
0x00000088: E8 BD 80 03 00 00 00 05 00 00 00 07 AA AA AA AA 00 00 00 00 00 00 00 00 00 00 00 00 FF FF FF FF 00 00
0x000000AA: 00 8C 00 00 00 94 00 00 00 28 00 00 00 94 00 00 00 07 00 00 00 05 00 00 00 00 FF FF FF FF 00 00 00 00
```

The 2-word stack frame

Saved old frame pointer

Saved link register

Saved r1

Saved r0

The address of C.

The value of B.

The value of A at the base of the stack the stack.

We have now used the following block of code to push A, B, and &C on the stack. The next step is to call the function with:

```
        BL    AddSq        ;call routine
```

This operation saves the return address in the link register, rl (i.e., r14).

The first thing we do in the function is to save the link register on the stack and any working registers we are going to be using.

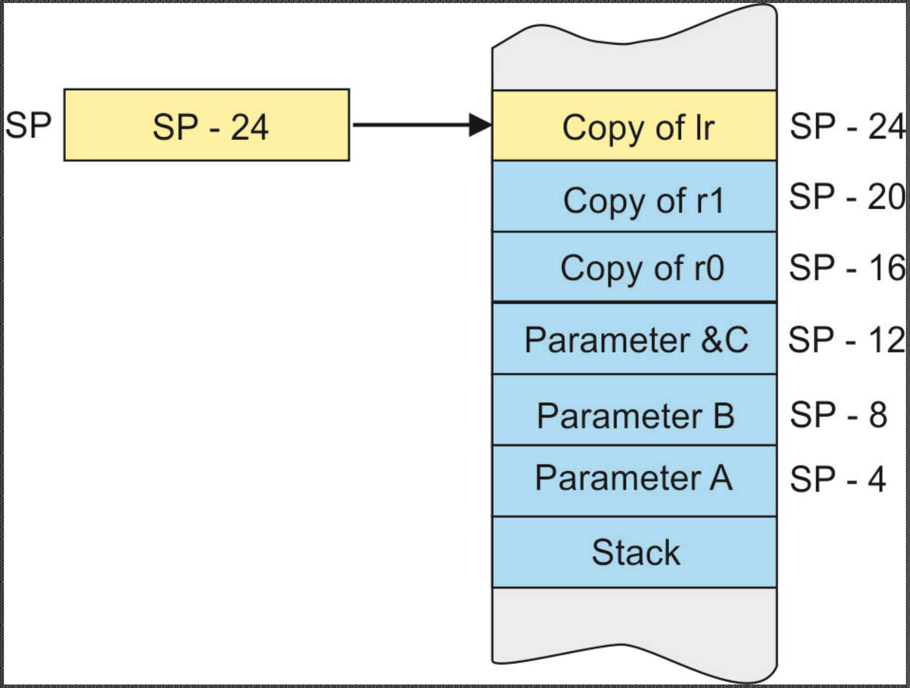
In this case we will be using two registers in the function, r0 and r1, so these will also be pushed on the stack along with the link register.

This is the start of the function.

```
AddSq    STMFD sp!,{r0,r1,lr}          ;push link register and r0/r1 on the stack
```

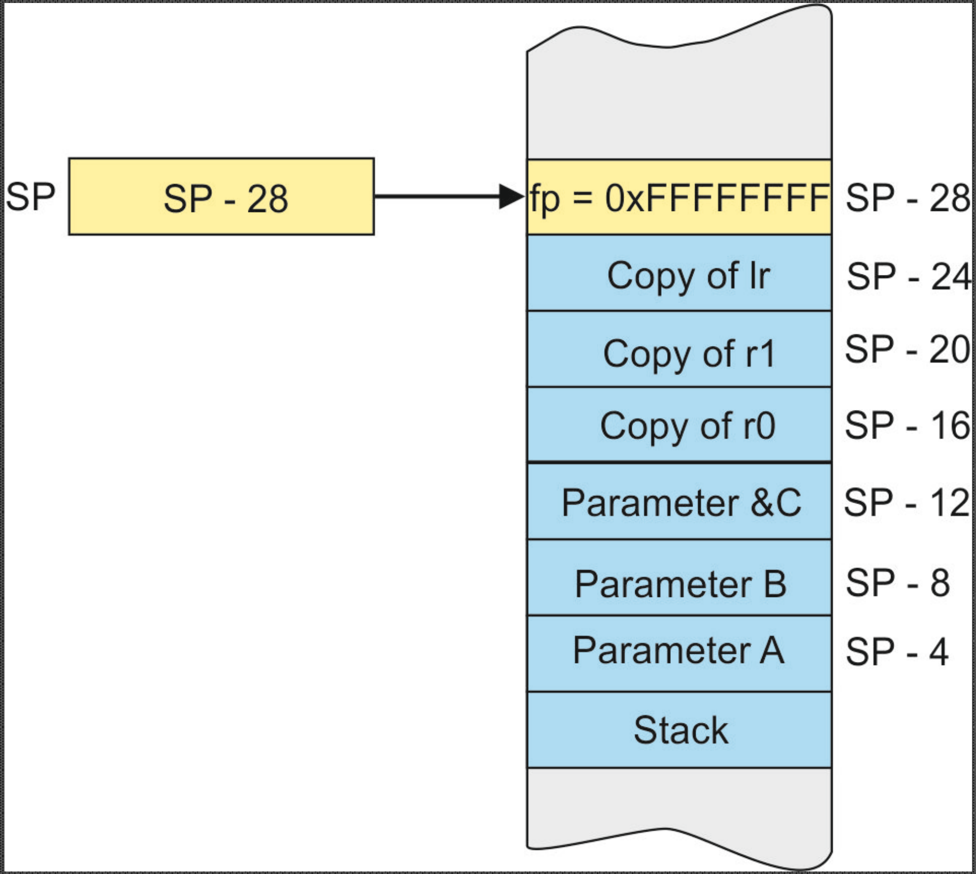
We have used the STMFD instruction (store multiple registers using a full descending stack) to push the link register r0, and r1. Registers are always stacked with the lowest numbered register at the lowest numbered address.

We now have the situation below.



To create a stack frame we first push the old frame pointer on the stack.

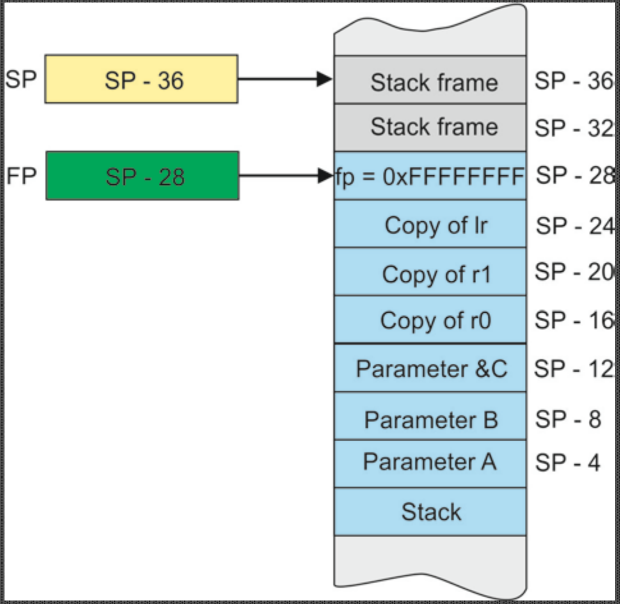
```
AddSq  STMFD sp!,{r0,r1,lr}       ;push link register and r0/r1 on the stack
       STR   fp,[sp,#-4]!         ;push frame pointer on the stack
```



In the next step we copy the stack pointer to the frame pointer. This means that the frame pointer is now pointing to the base of the current frame (i.e., where the previous value of the frame pointer has been saved).

By subtracting 8 (two words) from the stack pointer, we move the stack pointer up to leave a two-word stack frame.

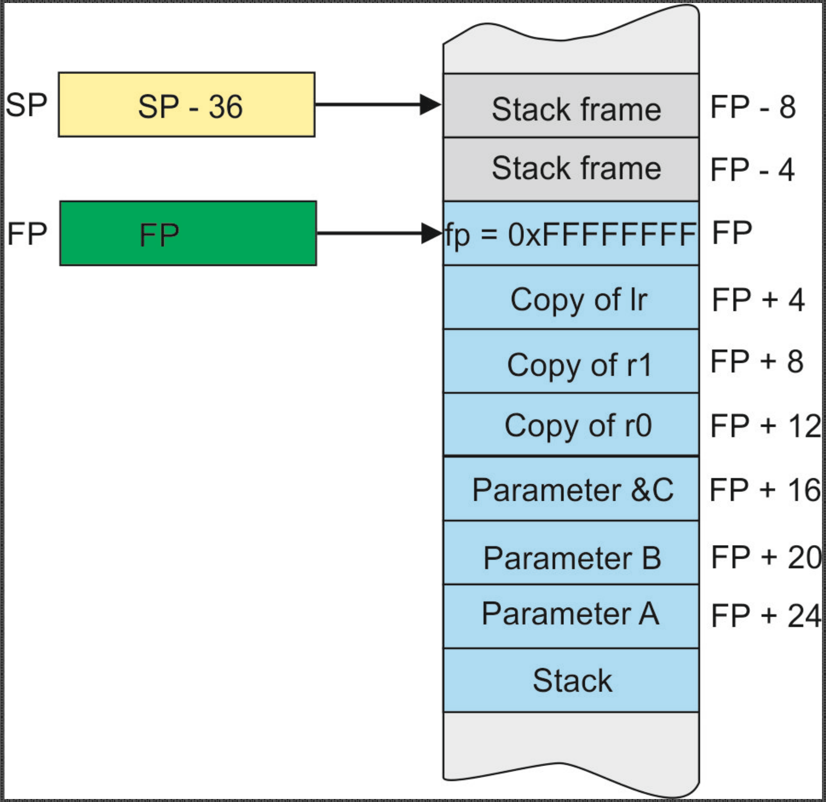
```
AddSq   STMFD sp!,{r0,r1,lr} ;push link register and r0/r1 on the stack
        STR   fp,[sp,#-4]!    ;push frame pointer on the stack
        MOV   fp,sp           ;frame pointer points at base of stack frame
        SUB   sp,sp,#8        ;create 2-word stack frame
```



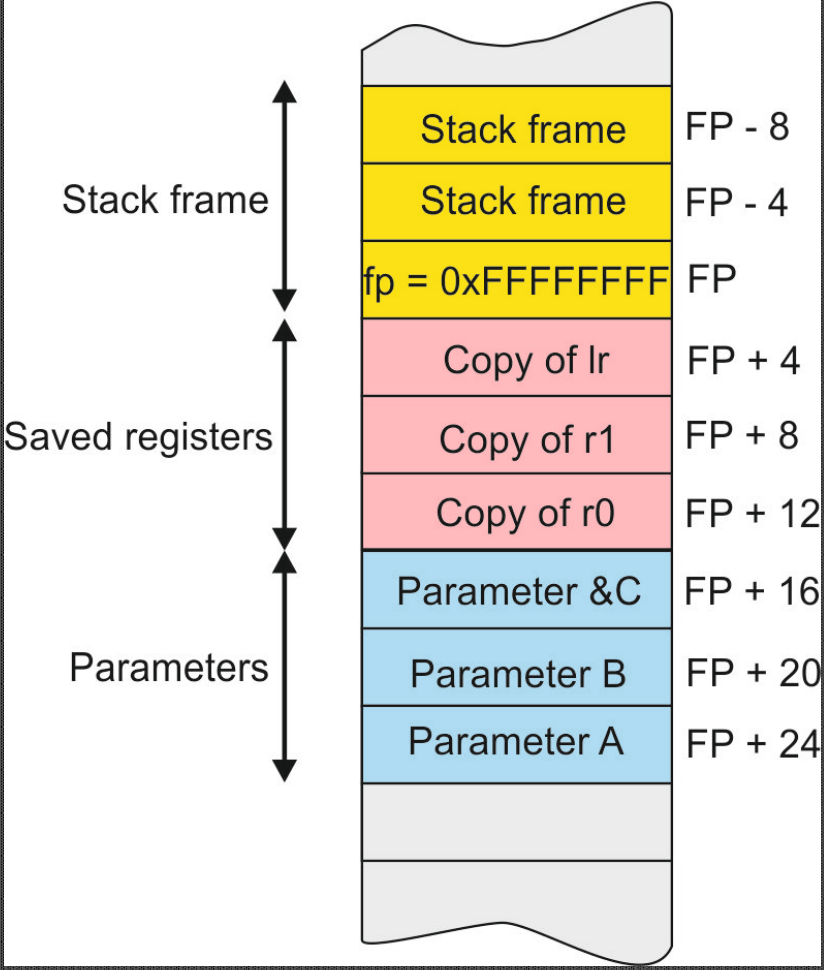
Here we have exactly the same situation as in the previous figure.

The only difference is that all memory addresses are now labelled with respect to the current value of the frame pointer.

That is, the frame pointer will be used to make all futures accesses during the evaluation of the function.

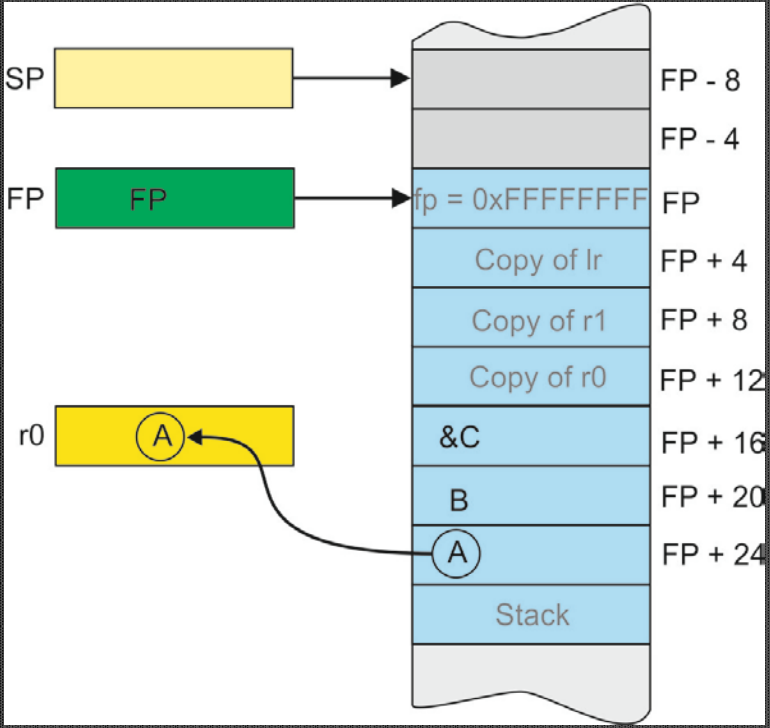


This is simply a repetition of the previous figure. We have used different shadings to show the three components of the stack: parameters, saved registers, and stack frame.



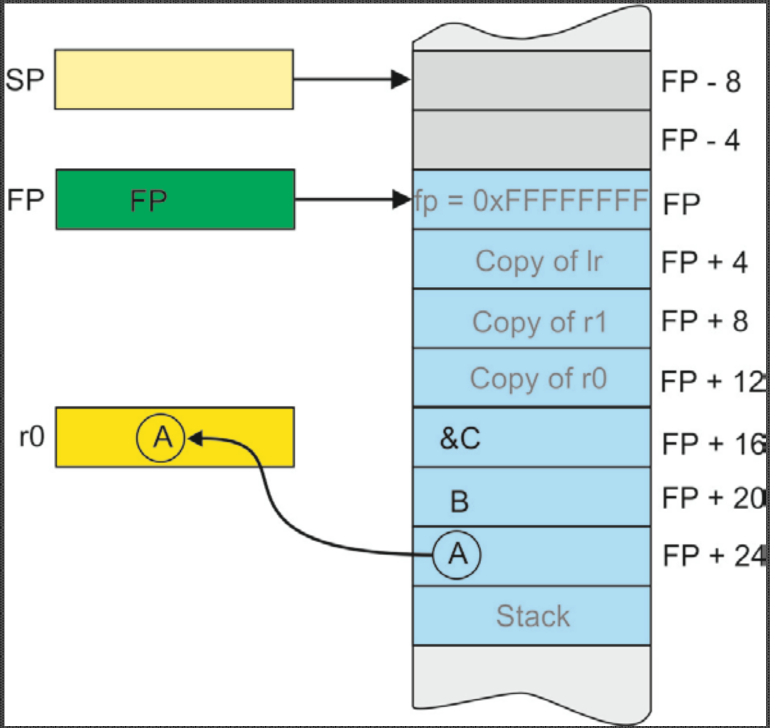
Now we can begin data processing. The following code shows how we read the value of A from the stack 24 bytes (6 words) below the frame pointer and copy it to register r0.

```
LDR     r0,[fp,#24]     ;get parameter A from stack
MOV      r1,r0          ;copy to r1
MUL     r1,r0,r1        ;square A
STR     r1,[fp,#-4]     ;store A² in stack frame
```



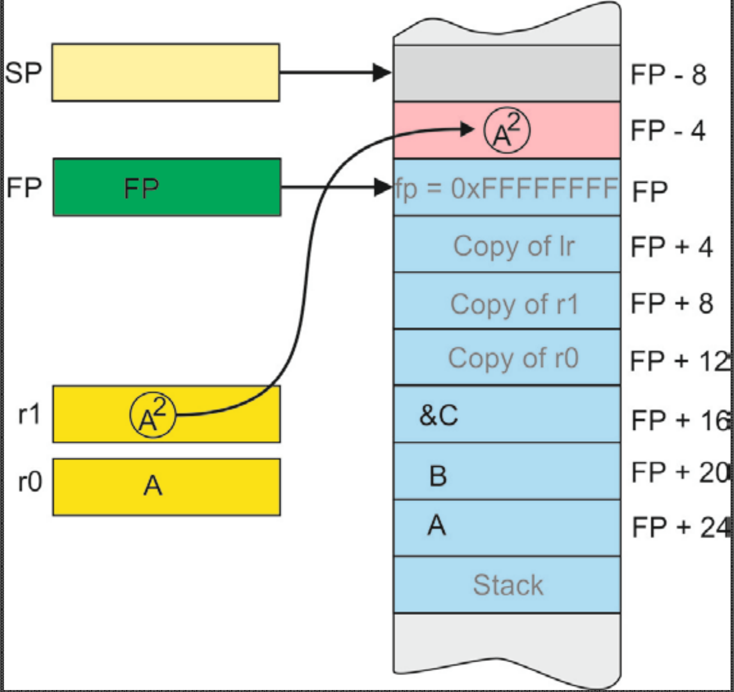
Now we can begin data processing. The following code shows how we read the value of A from the stack 24 bytes (6 words) below the frame pointer and copy it to register r0.

```
LDR   r0,[fp,#24]       ;get parameter A from stack
MOV   r1,r0             ;copy to r1
MUL   r1,r0,r1          ;square A
STR   r1,[fp,#-4]       ;store A^2 in stack frame
```



We copy r0 to r1 and then use MUL to square the number. Note that we have to use two different source registers; this is a requirement of MUL. Then we copy A^2 to the stack frame we have created. Note that it’s location is 4 bytes above the frame pointer.

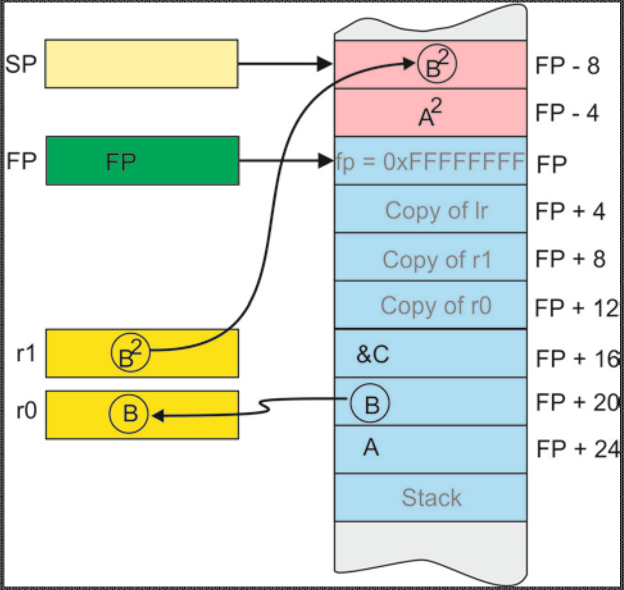
```
LDR   r0,[fp,#24]      ;get parameter A from stack
MOV   r1,r0            ;copy to r1
MUL   r1,r0,r1         ;square A
STR   r1,[fp,#-4]      ;store A² in stack frame
```



We continue with the calculation. Parameter B is read from the stack, squared and loaded into the second slot on the stack frame as shown.

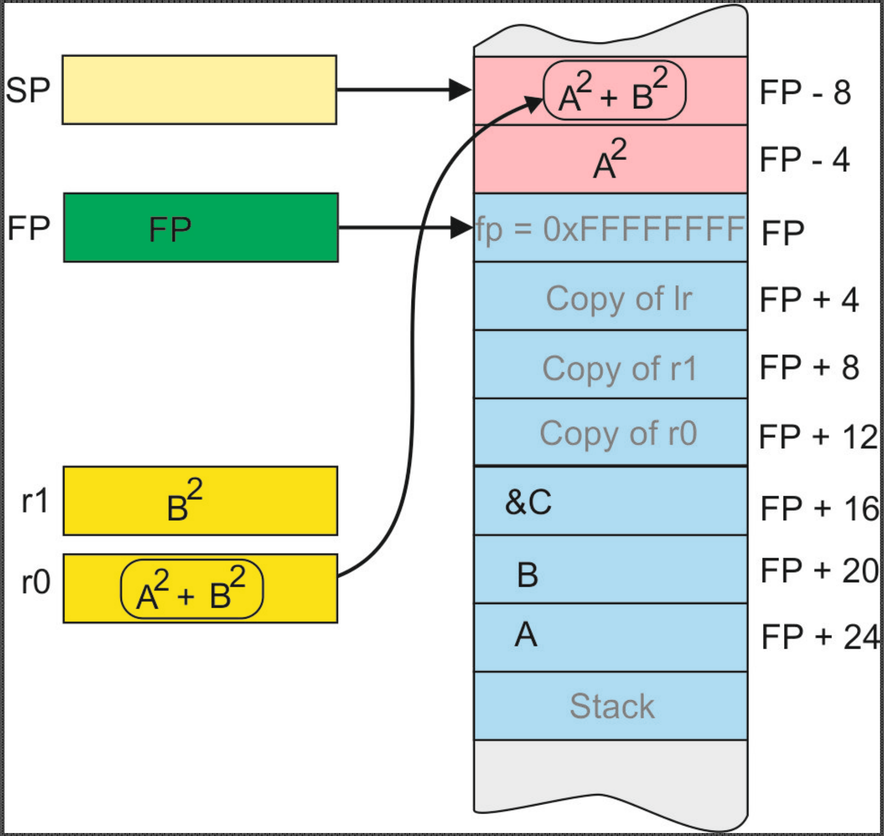
We now have a stack frame that contains our two temporary variables.

```
LDR  r0,[fp,#20]   ;get parameter B from stack
MOV  r1,r0                 ;copy to r1
MUL  r1,r0,r1      ;square B
STR  r1,[fp,#-8]   ;store B^2 in stack frame
LDR  r0,[fp,#-12]  ;get A^2 from stack frame
ADD  r0,r0,r1      ;calculate A^2 + B^2
```



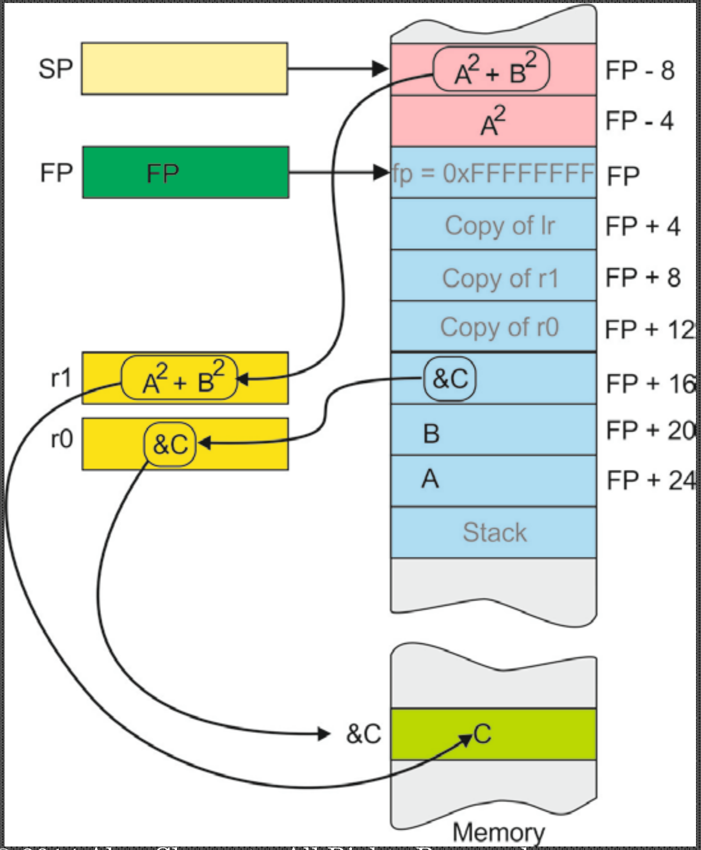
We now read A2 back from the stack frame and add B^2 to it to get $A^2 + B^2$. This final result is saved in the stack frame overwriting the old B^2.

```
LDR r0,[fp,#-12]   ;get A² from stack frame
ADD r0,r0,r1       ;calculate A² + B²
STR  r0,[fp,#-8]   ;save result in stack frame and overwrite B²
```



The next step is to return the result in memory location C. The address of C, &C, is loaded in register r0 from [fp] + 16. Then the result in the stack frame is loaded into register r1. This is at [fp] – 8. Finally, the result is passed to the calling program by STR r1,[r0].

```
LDR           r0,[fp,#16]  ;get address of C in r0
LDR           r1,[fp,#-8]   ;get result from stack frame
STR r1,[r0]   ;save result in calling environment
```

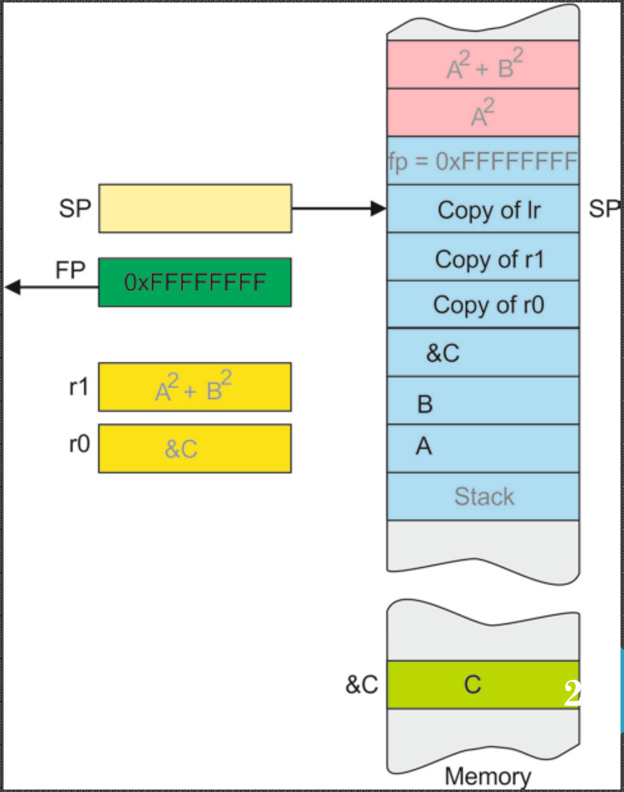


All that now remains is to return from the function. We have to collapse the stack frame, restore registers, and return to the calling point.

```
ADD    sp,sp,#8          ;delete stack frame
LDR    fp,[sp],#4        ;restore frame pointer from stack
LDMFD  sp!,{r0,r1,pc}   ;pull return address off the stack, return, restore r0, r1
```

The figure shows the state of the stack after the first two instructions have been executed to collapse the stack frame.

Note that the frame pointed has been restored to its previous value.



The final fragment of code demonstrates the sequence of events after the return.

```
AddSq    ;call routine
         ADD   sp,sp,#12  ;clean up the stack to do the calculation
         ADR   r1,C       ;get address of C in r1
         LDR   r1,[r1]    ;push a value of C in r1 (for testing)
Endless  B     Endless    ;dummy loop
```

We clean up the stack by moving it down three words (12 bytes) to release the space taken by the three parameters A, B, and &C.

Finally, we load the address of C in r1 and then retrieve its value. This is done simply to debug the program.

At the end we enter an infinite parking loop.