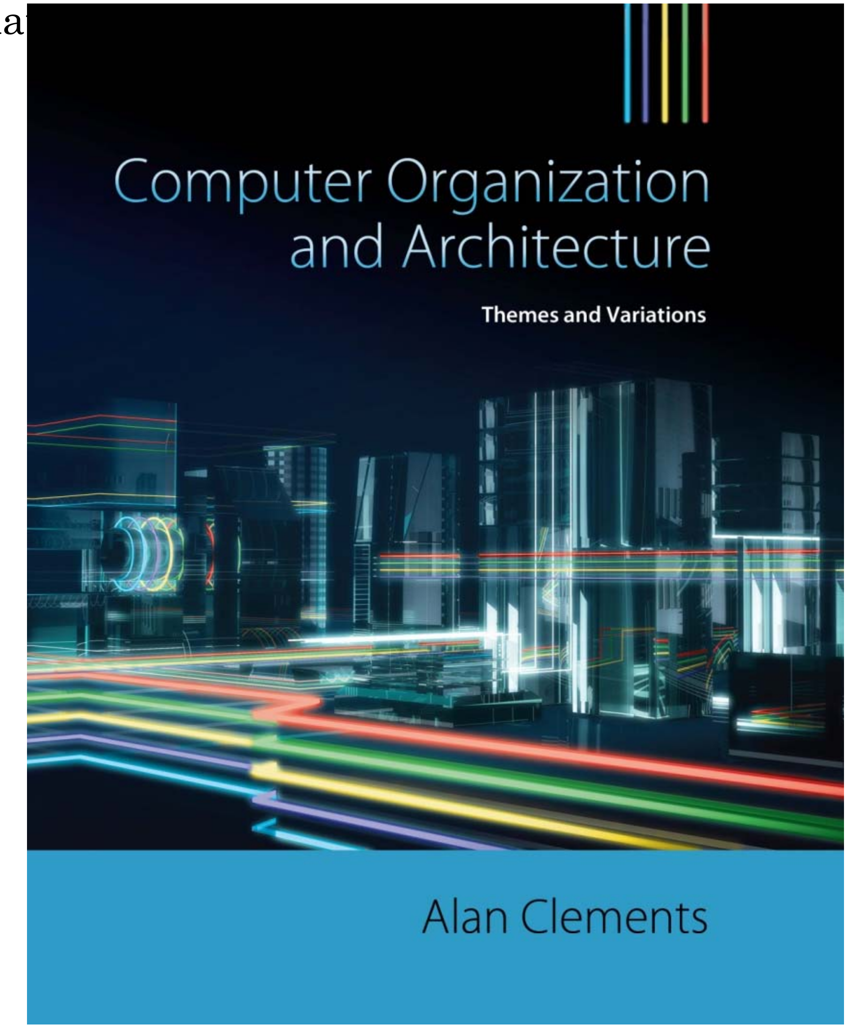
Part 2

CHAPTER 4

Computer Organization and Architecture



CENGAGE Learning

Music: *"Corporate Success"* by *Scott Holmes*, used under *Attribution-NonCommercial License*

Calling a Subroutine Step-by-Step

- To call a subroutine, the following steps need to be performed:
  - ***Parameters*** need to be *passed* from the caller to the subroutine.
    This can be performed via the stack.
  - The ***address*** of the instruction immediately after the calling instruction needs to be *saved in a safe place* ***BEFORE*** branching to the subroutine.
    This can be performed by using BL instruction or via the stack, or both.
  - *Inside the subroutine*, we need to:
    - Push the values of all registers to be used inside the subroutine, as well as the FP (R11) and LR (R14).
    - Make the FP (R11) point to the bottom of the frame by copying the value of the SP (R13) to the FP (R11).
    - Create a space inside the stack for local variables.
    - Perform the subroutine instructions.
      The addresses of parameters and local variables are calculated relative to the value of the FP (R11).
    - *At the end of the subroutine*, deallocate all created local variables.
    - Pop all pushed registers but use PC (R15) instead of LR (R14).
  - *At the caller program*, all pushed parameters need to be popped.

so that can return to the caller function

Passing Parameters via the Stack

- You can pass a parameter to a subroutine
  - ***by value***
  - ***by reference***

- When passed ***by value***, the subroutine receives a **copy** of the parameter.
  - Passing a parameter by value causes the *parameter to be cloned* and the *cloned version of the parameter* to be used by the subroutine.
  - If the parameter is modified by the subroutine, the new value does not affect the value of the parameter elsewhere in the program.

- When passed ***by reference***, the subroutine receives a **pointer**, (i.e., an **address**) to the parameter.
  - *There is only one copy of the parameter* and the subroutine can access this value because it knows the address of the parameter.
  - If the subroutine modifies the parameter, it is modified the original value.

Passing Parameters via the Stack

- The subroutine `swap(int a, int b)` *intends* to exchange two values.
- Let's examine how parameters are passed to this subroutine.

```
void swap(int a, int b)   /* swaps the value of a and b */
{  int temp;
   temp = a;              /* copy a      to temp       */
   a    = b               /* copy b      to a,    and  */
   b    = temp;           /* copy temp to b            */
}
```

copy of x copy of y.

```
void main(void)
{  int x = 2, y = 3;
   swap (x, y);           /* swap a and b */
}
```

by param

Will it work?

No.

Passing Parameters via the Stack

```
      AREA SwapVal, CODE, READONLY

      ENTRY
      ADR   sp,STACK              ;set up stack pointer
      MOV   fp,#0xFFFFFFFF        ;set up dummy fp for tracing
      B      main                 ;jump to the function main

      SPACE 0x20
STACK DCD 0
```

FD Stack

You need to re-do it yourself using the other stack types.

```
;     void swap (int a, int b)
;     Parameter a    is at [fp]+4
;     Parameter b    is at [fp]+8
;     Variable  temp is at [fp]-4
```

FD Stack

Passing Parameters via the Stack

You need to re-do it yourself using the other stack types.

```
;       {
swap    SUB   sp,sp,#4      ;Create stack frame: decrement sp
        STR   fp,[sp]       ;push the frame pointer onto the stack
        MOV   fp,sp         ;frame pointer points at the base
;       int temp;
        SUB   sp,sp,#4      ;move sp up 4 bytes for temp
;       temp = a;
        LDR   r0,[fp,#4]    ;get parameter a from the stack
        STR   r0,[fp,#-4]   ;copy a to temp onto the stack frame
;       a    = b;
        LDR   r0,[fp,#8]    ;get parameter b from the stack
        STR   r0,[fp,#4]    ;copy b to a
;       b    = temp;
        LDR   r0,[fp,#-4]   ;get temp from the stack frame
        STR   r0,[fp,#8]    ;copy temp to b
;       }
                            ;Collapse stack frame created for swap
        MOV   sp,fp         ;restore the stack pointer
        LDR   fp,[sp]       ;restore old frame pointer from stack
        ADD   sp,sp,#4      ;move stack pointer down 4 bytes
        MOV   pc,lr         ;return by loading LR into PC
```

pushing frame pointer

popping frame pointer

Passing Parameters via the Stack

```
;       void main(void)
;       {
main                        ;Create stack frame in main for x, y
        SUB   sp,sp,#4      ;move the stack pointer up
        STR   fp,[sp]       ;push the frame pointer onto the stack
        MOV   fp,sp         ;the frame pointer points at the base;
;       int x = 2, y = 3;
        SUB   sp,sp,#8      ;move sp up 8 bytes for 2 integers
        MOV   r0,#2         ;x = 2
        STR   r0,[fp,#-4]   ;put x in stack frame
        MOV   r0,#3         ;y = 3
        STR   r0,[fp,#-8]   ;put y in stack frame
;       swap(x, y);
        LDR   r0,[fp,#-8]   ;get y from stack frame
        STR   r0,[sp,#-4]!  ;push y on stack
        LDR   r0,[fp,#-4]   ;get x from stack frame
        STR   r0,[sp,#-4]!  ;push x on stack
        BL    swap          ;call swap, save return address in LR
        ADD   sp,sp,#8      ;Clean the stack from the parameters
;       }
        MOV   sp,fp         ;restore the stack pointer
        LDR   fp,[sp]       ;restore old frame pointer from stack
        ADD   sp,sp,#4      ;move stack pointer down 4 bytes
Loop    B     Loop          ;Stop
        END
```

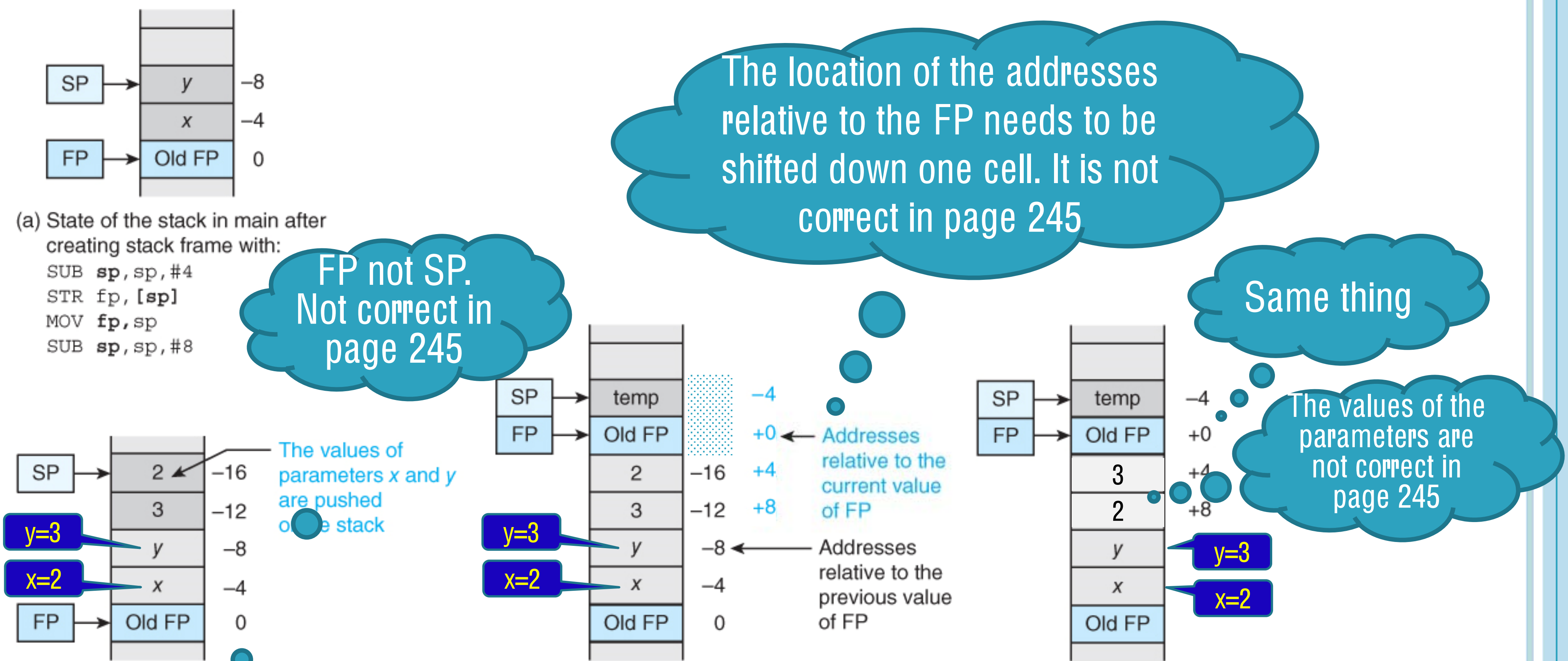
Bold is not correct in page 244

Passing Parameters via the Stack

- This code swaps the variables inside the stack frame
- When the return is made, the stack frame will be collapsed, and the effect of the swap will be lost.
- The variables in the calling environment are not affected.

Passing Parameters via the Stack

FIGURE 4.8 Passing values to a subroutine by value



(a) State of the stack in main after creating stack frame with:

```
SUB sp,sp,#4
STR fp,[sp]
MOV fp,sp
SUB sp,sp,#8
```

(b) The stack in main after putting two parameters in the stack frame with:

```
MOV r0,#2
STR r0,[fp,#-4]
MOV r0,#3
STR r0,[fp,#-8]
```

Then pushing two parameters on the stack

```
LDR r0,[fp,#-8]
STR r0,[sp,#-4]!
LDR r0,[fp,#-4]
STR r0,[sp,#-4]!
```

(c) The stack after the creation of a stack frame in swap. The new stack frame is four bytes deep and holds the variable temp. The frame is created by:

```
SUB sp,sp,#4
STR fp,[sp]
MOV fp,sp
SUB sp,sp,#4
```

(d) The stack after executing the body of swap. Note that all data is referenced to FP.

```
LDR r0,[fp,#4]
STR r0,[fp,#-4]
LDR r0,[fp,#8]
STR r0,[fp,#4]
LDR r0,[fp,#-4]
STR r0,[fp,#8]
```

Passing Parameters via the Stack

- In the next example, we pass parameters by reference

```
void swap(int *a, int *b)    /* A function to swap two parameters
                                in calling program          */
{ int temp;                  /* copy *a    to temp          */
  temp = *a;                 /* copy *b    to *a,   and     */
  *a   = *b;                 /* copy temp to  *b            */
  *b   = temp;
}

void main(void)
{ int x = 2, y = 3;
  swap(&x, &y);              /* call swap and pass
       added                    addresses of parameters  */
}
```

Passing Parameters via the Stack

```
      AREA SwapVal, CODE, READONLY

      ENTRY
      ADR   sp, STACK          ;set up stack pointer
      MOV   fp,#0xFFFFFFFF     ;set up dummy fp for tracing
      B     main               ;jump to main function

      SPACE 0x20
STACK DCD 0

   ;   void swap (int *a, int *b)
   ;   Parameter *a   is at [fp]+4
   ;   Parameter *b   is at [fp]+8
   ;   Variable  temp is at [fp]-4
```

Passing Parameters via the Stack

```
;       {
swap    SUB   sp,sp,#4       ;Create stack frame: decrement sp
        STR   fp,[sp]        ;push the frame pointer onto the stack
        MOV   fp,sp          ;frame pointer points at the base
;       int temp;
        SUB   sp,sp,#4       ;move sp up 4 bytes for temp
;       temp = *a;
        LDR   r1,[fp,#4]     ;get address of parameter a
        LDR   r2,[r1]        ;get value of parameter a (i.e., *a)
        STR   r2,[fp,#-4]    ;store *a in temp in stack frame
;       *a = *b;
        LDR   r0,[fp,#8]     ;get address of parameter b
        LDR   r3,[r0]        ;get value of parameter b (i.e., *b)
        STR   r3,[r1]        ;store *b in *a
;       *b = temp;
        LDR   r3,[fp,#-4]     ;get temp
        STR   r3,[r0]         ;store temp in *b
;       }
                             ;Collapse stack frame created for swap
        MOV   sp,fp          ;restore the stack pointer
        LDR   fp,[sp]        ;restore old frame pointer from stack
        ADD   sp,sp,#4       ;move stack pointer down 4 bytes
        MOV   pc,lr          ;return by loading LR into PC
```

Missing the *
in page 247

Passing Parameters via the Stack

```
;      void main(void)
;      {
main                        ;Create stack frame in main for x, y
       SUB   sp,sp,#4       ;move the stack pointer up
       STR   fp,[sp].       ;push the frame pointer onto the stack
       MOV   fp,sp          ;the frame pointer points at the base;
;      int x = 2, y = 3;
       SUB   sp,sp,#8       ;move sp up 8 bytes for 2 integers
       MOV   r0,#2          ;x = 2
       STR   r0,[fp,#-4]    ;put x in stack frame
       MOV   r0,#3          ;y = 3
       STR   r0,[fp,#-8]    ;put y in stack frame
;      swap(&x, &y);
       SUB   r0,fp,#8       ;get  address of y in stack frame
       STR   r0,[sp,#-4]!   ;push address of y on stack
       SUB   r0,fp,#4       ;get  address of x in stack frame
       STR   r0,[sp,#-4]!   ;push address of x on stack
       BL    swap           ;call swap, save return address in LR
       ADD   sp,sp,#8       ;Clean the stack from the parameters
;      }
       MOV   sp,fp          ;restore the stack pointer
       LDR   fp,[sp]        ;restore old frame pointer from stack
       ADD   sp,sp,#4       ;move stack pointer down 4 bytes
Loop   B     Loop           ;Stop
       END
```

Bold is not correct in page 244

x, y

Passing Parameters via the Stack

- In the function main, the addresses of the *parameters are pushed onto the stack* by means of the following instructions:

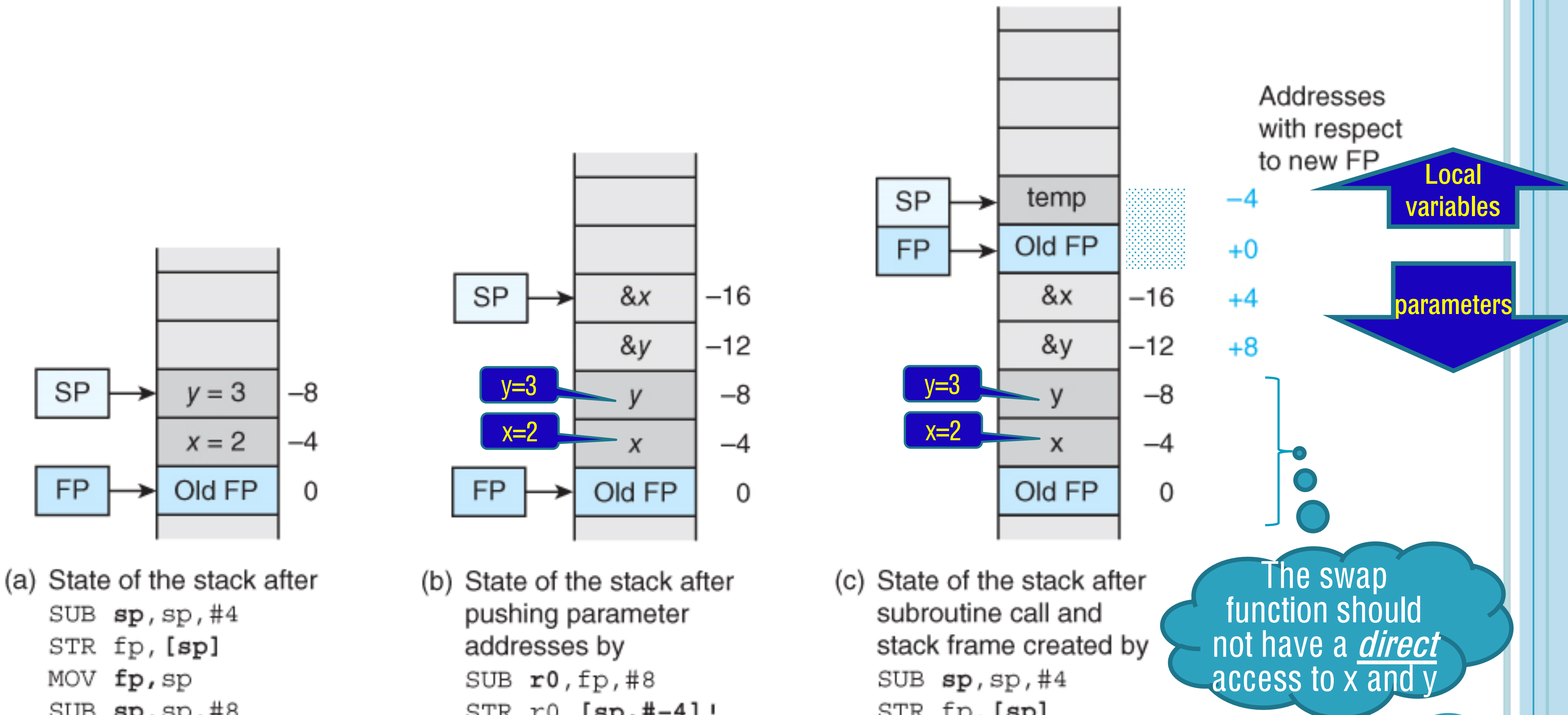
```
SUB   r0,fp,#8       ;get  address of y in stack frame
STR   r0,[sp,#-4]!   ;push address of y on stack
SUB   r0,fp,#4       ;get  address of x in stack frame
STR   r0,[sp,#-4]!   ;push address of x on stack
```

- In the function swap, the addresses of *parameters are read from the stack* by means of

```
;     temp = *a;
      LDR   r1,[fp,#4]   ;get address of parameter a
      LDR   r2,[r1]      ;get value of parameter a (i.e., *a)
      STR   r2,[fp,#-4]  ;store *a in temp in stack frame
;     *a = *b;
      LDR   r0,[fp,#8]   ;get address of parameter b
      LDR   r3,[r0]      ;get value of parameter b (i.e., *b)
      STR   r3,[r1]      ;store *b in *a
;     *b = temp;
      LDR   r3,[fp,#-4]   ;get temp
      STR   r3,[r0]       ;store temp in *b
```

Passing Parameters via the Stack

FIGURE 4.9 Passing values to a subroutine by reference



(a) State of the stack after

```
SUB sp,sp,#4
STR fp,[sp]
MOV fp,sp
SUB sp,sp,#8
MOV r0,#2
STR r0,[fp,#-4]
MOV r0,#3
STR r0,[fp,#-8]
```

in function main

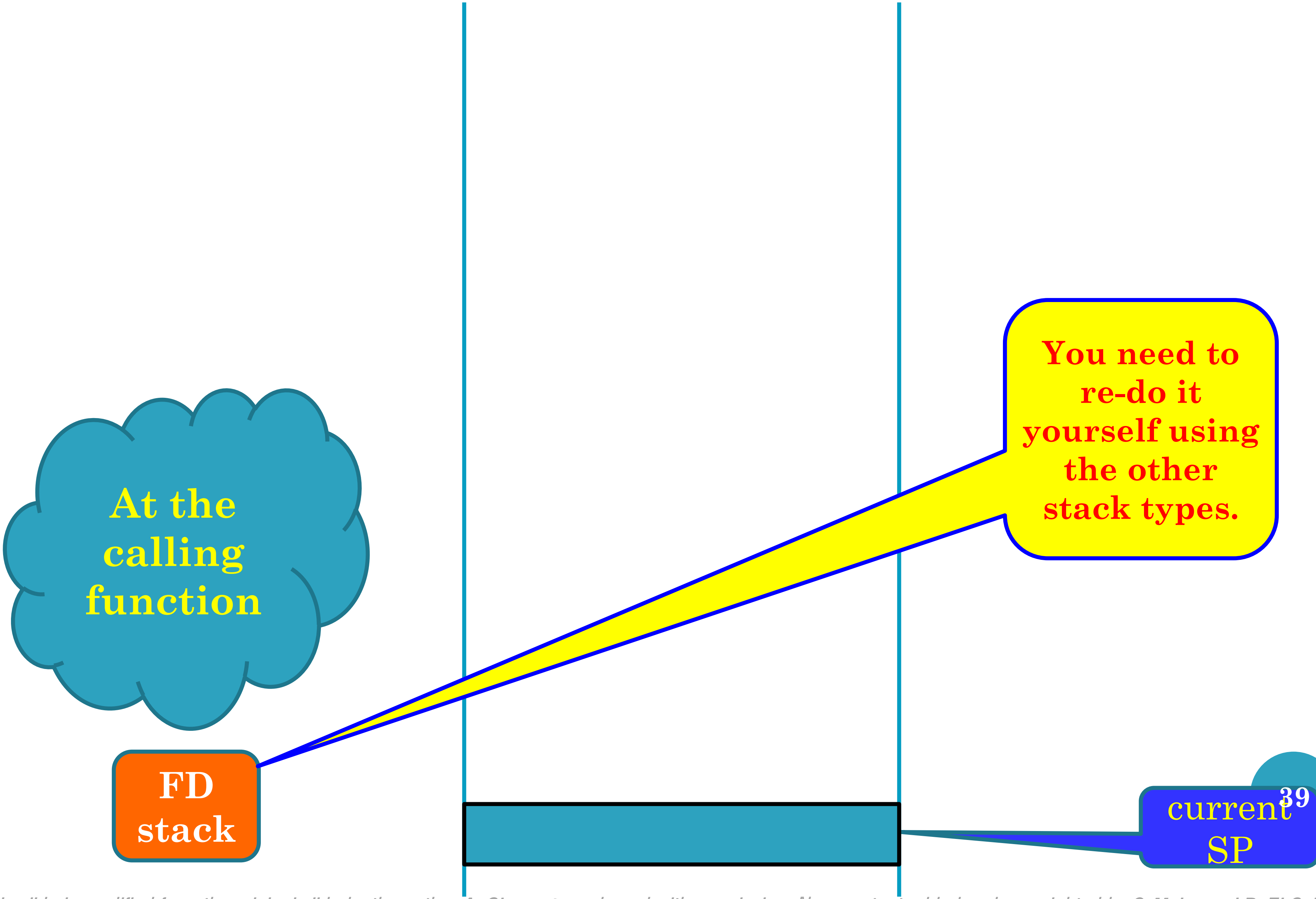
(b) State of the stack after pushing parameter addresses by

```
SUB r0,fp,#8
STR r0,[sp,#-4]!
SUB r0,fp,#4
STR r0,[sp,#-4]!
```

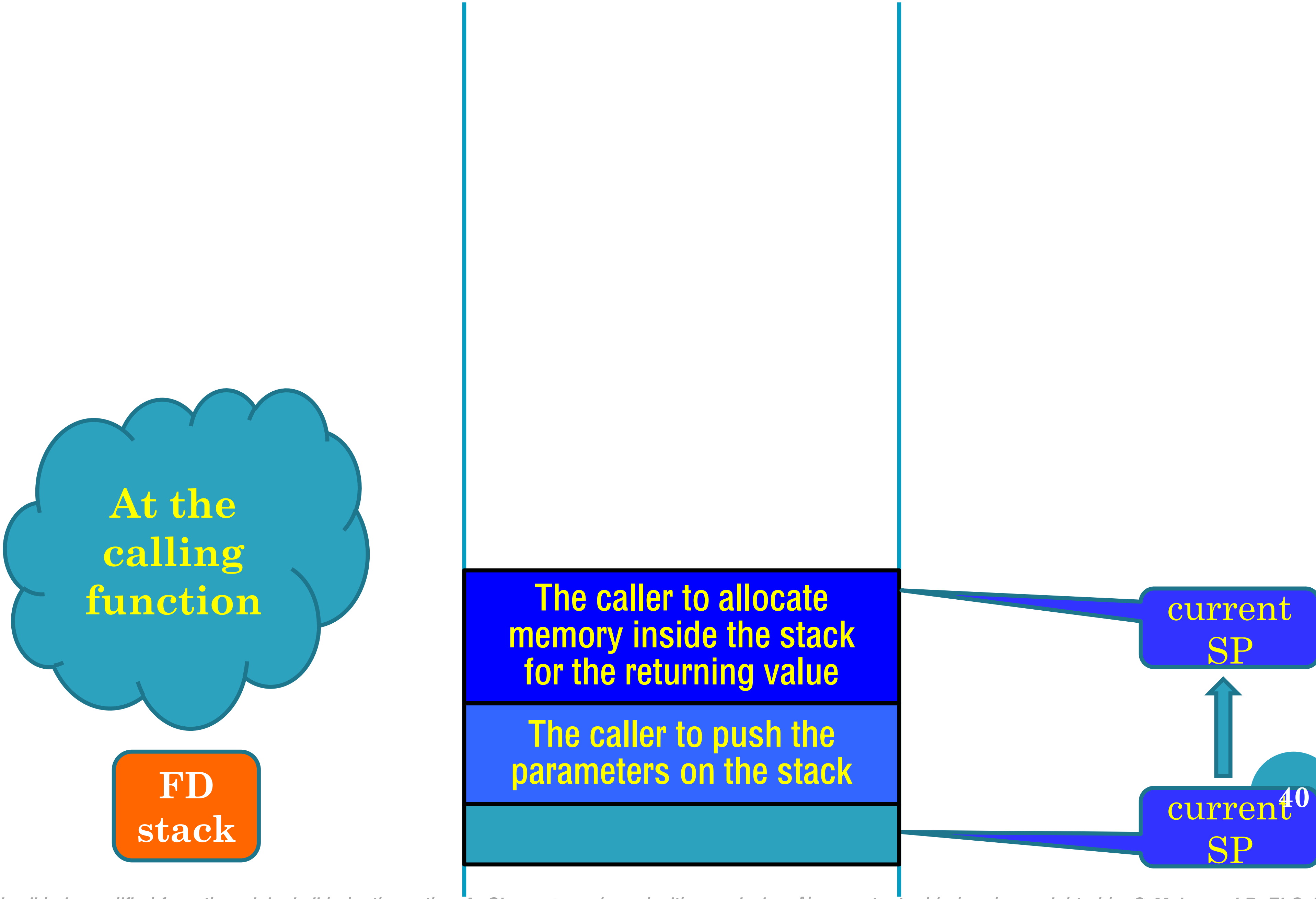
(c) State of the stack after subroutine call and stack frame created by

```
SUB sp,sp,#4
STR fp,[sp]
MOV fp,sp
SUB sp,sp,#4
```

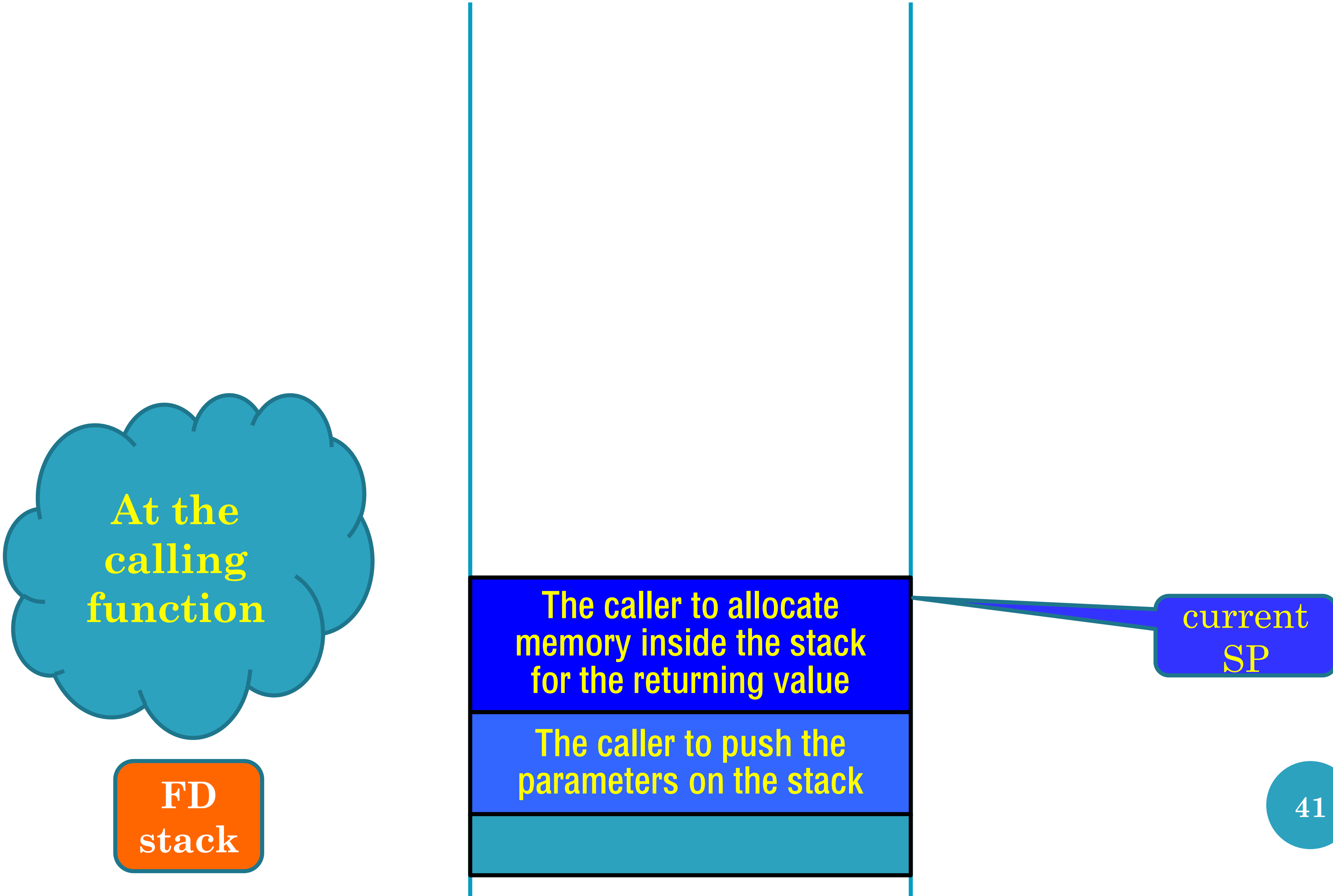
The Traditional Call/Return Mechanism



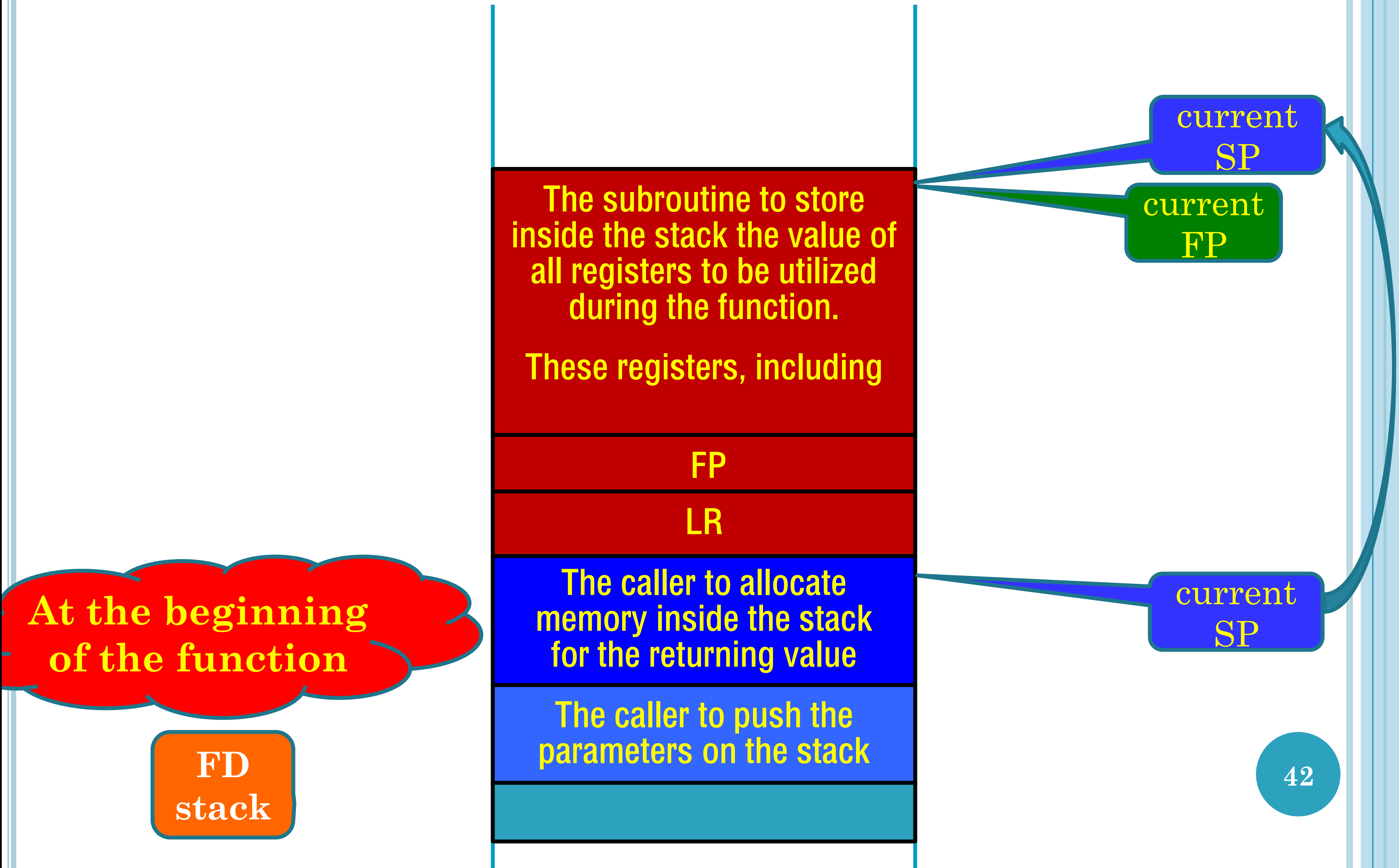
The Traditional Call/Return Mechanism



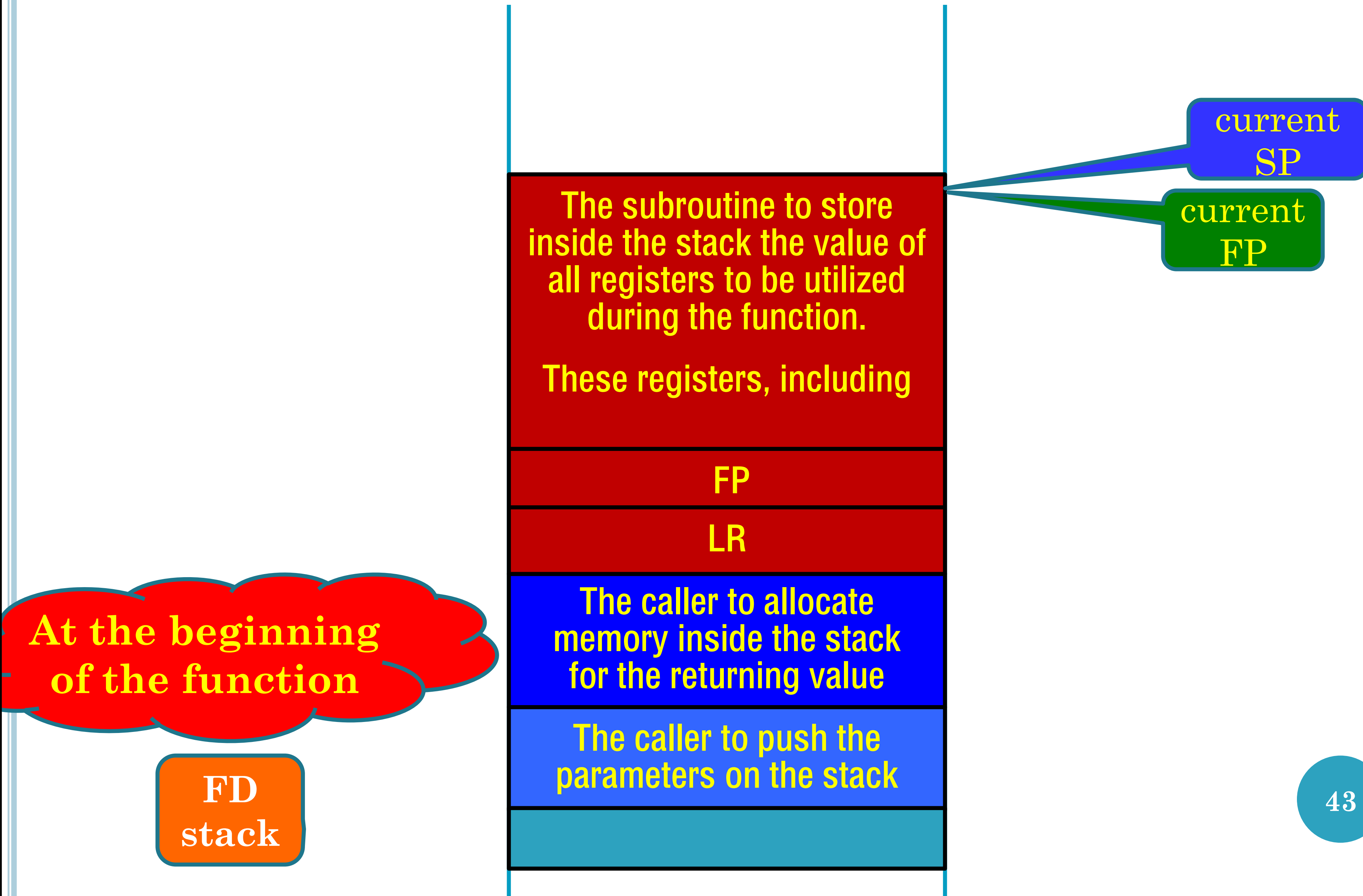
The Traditional Call/Return Mechanism



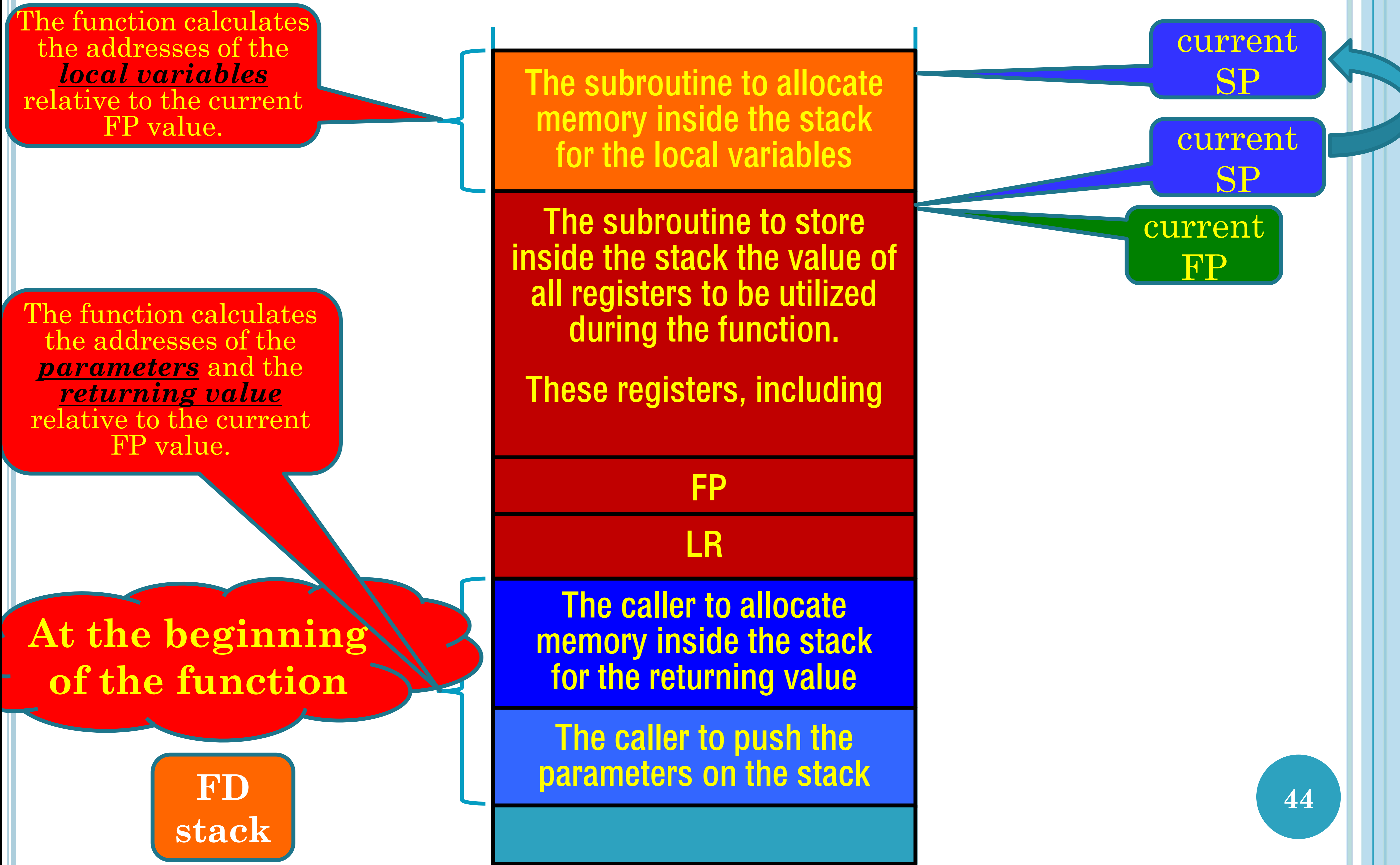
The Traditional Call/Return Mechanism



The Traditional Call/Return Mechanism



The Traditional Call/Return Mechanism



The Traditional Call/Return Mechanism

The function calculates the addresses of the ***local variables*** relative to the current FP value.

call by value
vs
call by reference

The function calculates the addresses of the ***parameters*** and the ***returning value*** relative to the current FP value.

At the beginning of the function

FD stack

The subroutine to allocate memory inside the stack for the local variables

The subroutine to store inside the stack the value of all registers to be utilized during the function.

These registers, including

FP

LR

The caller to allocate memory inside the stack for the returning value

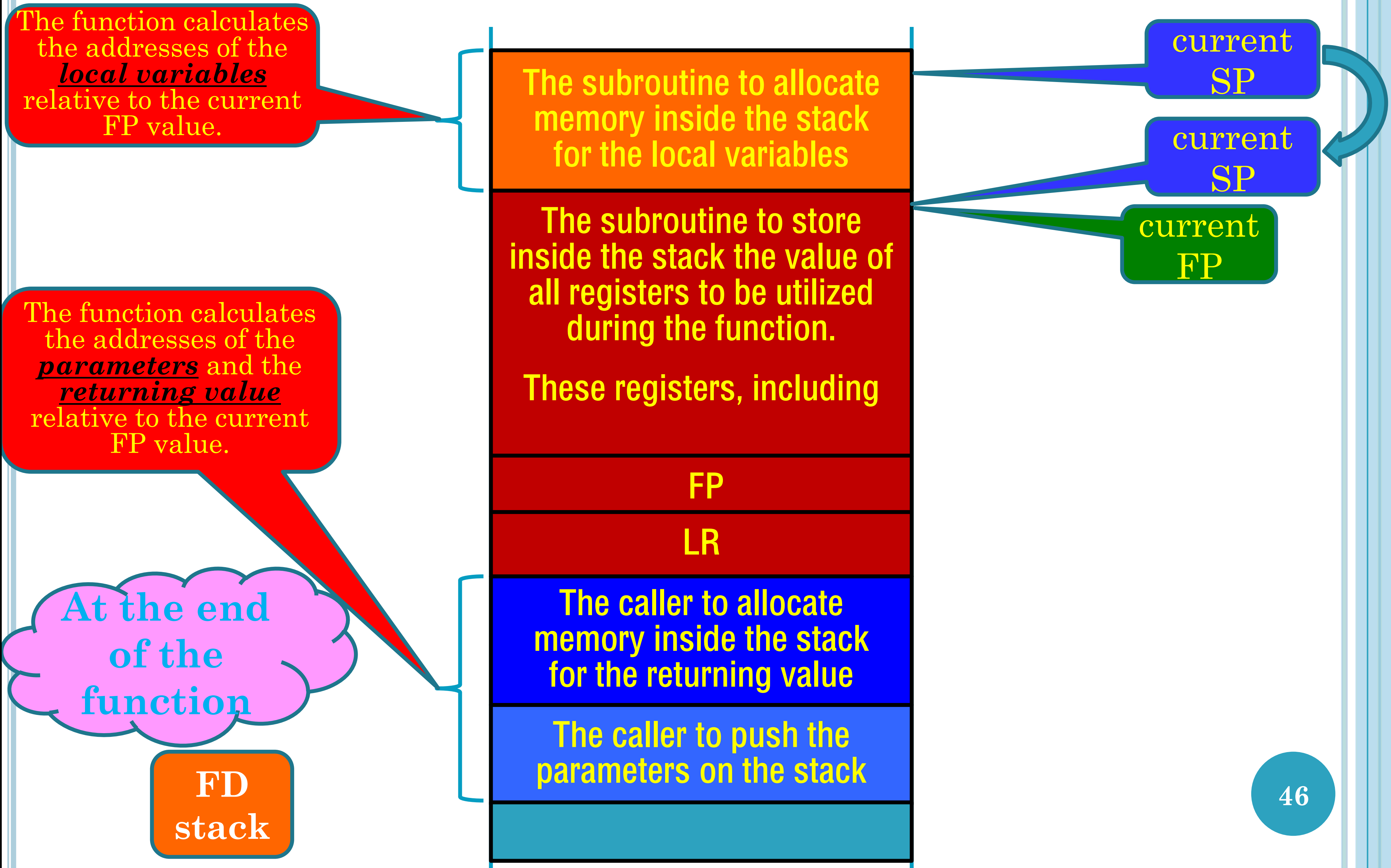
The caller to push the parameters on the stack

current SP

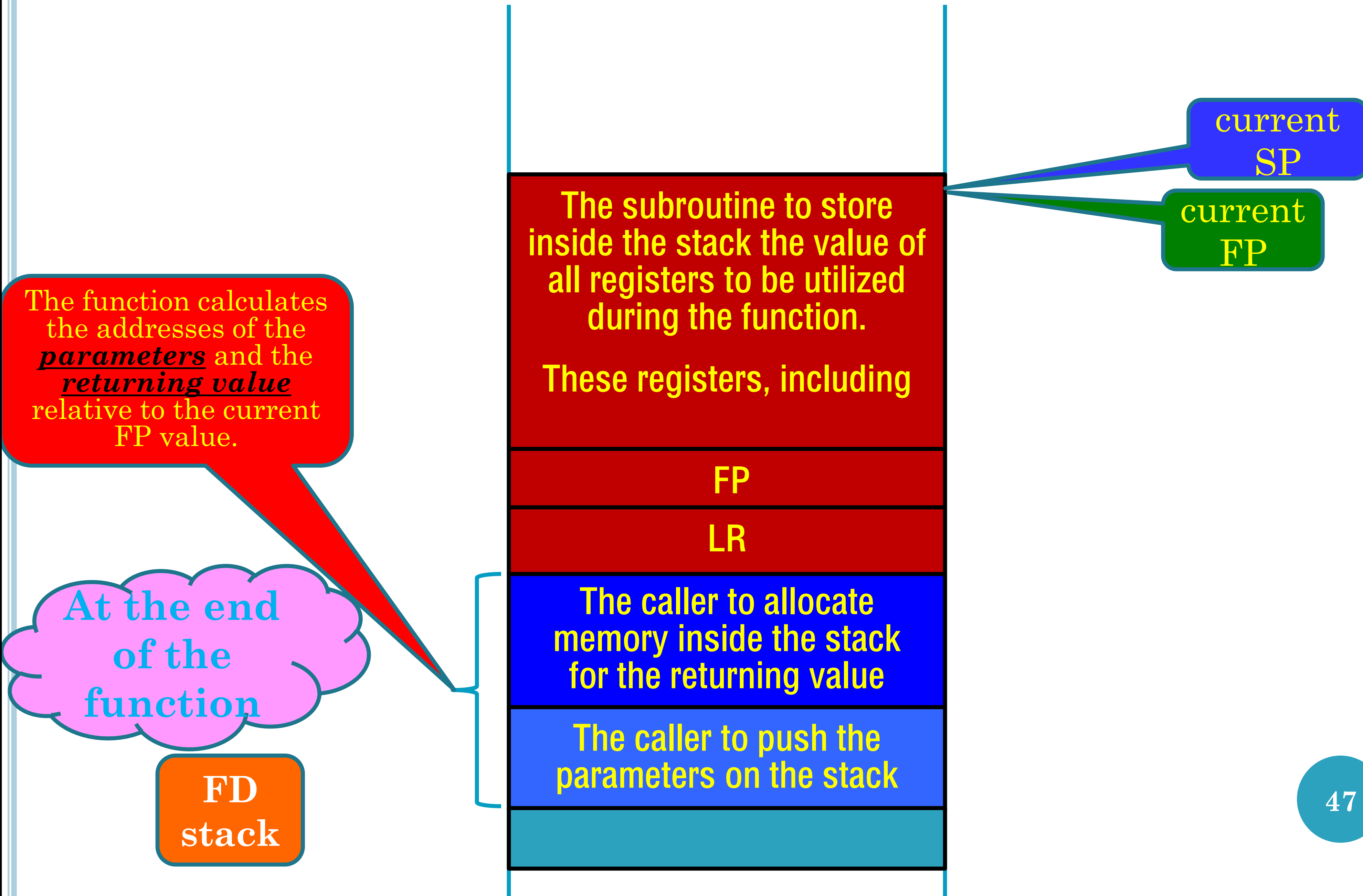
current FP

What will happen if the function called itself? i.e., recursion

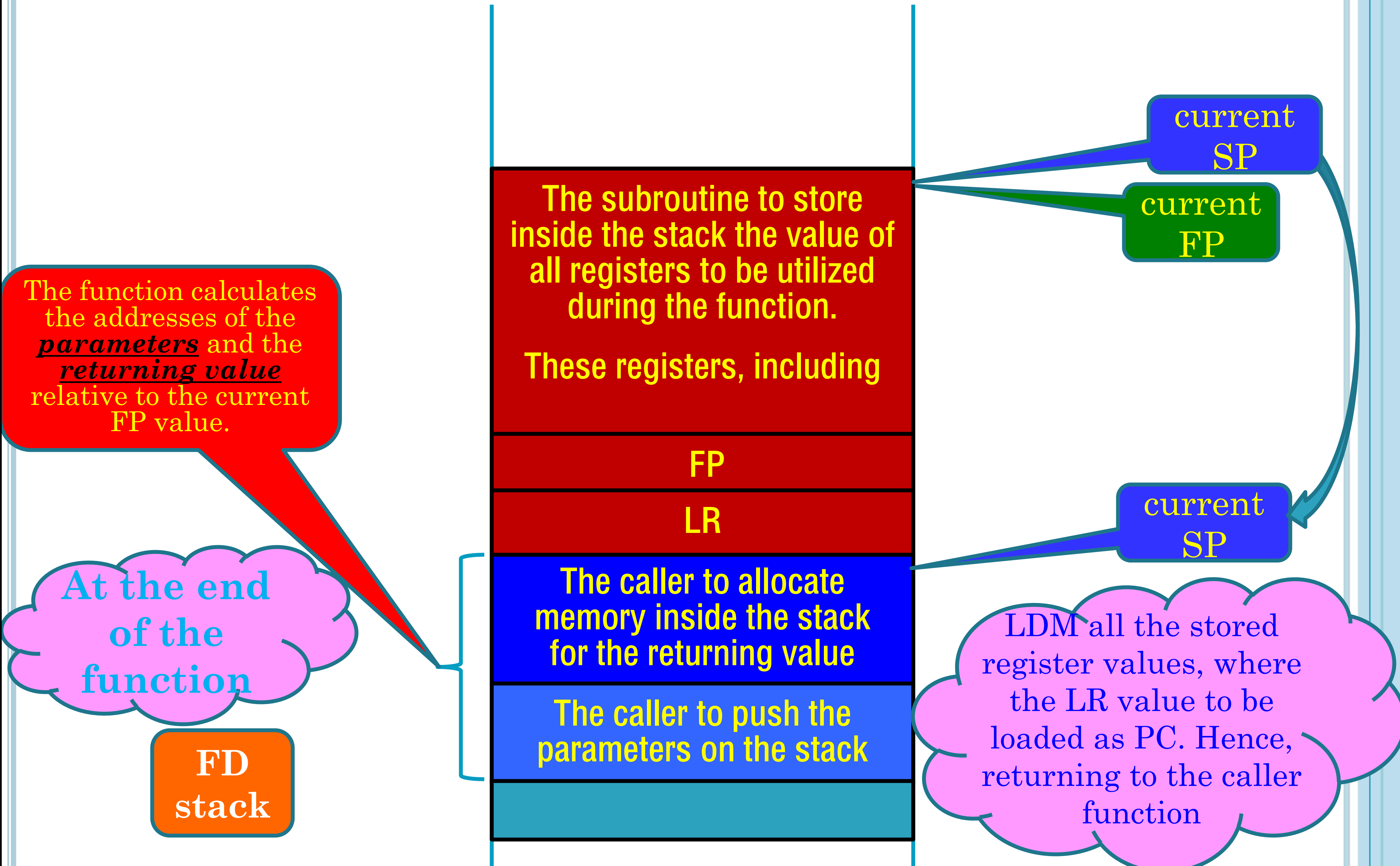
The Traditional Call/Return Mechanism



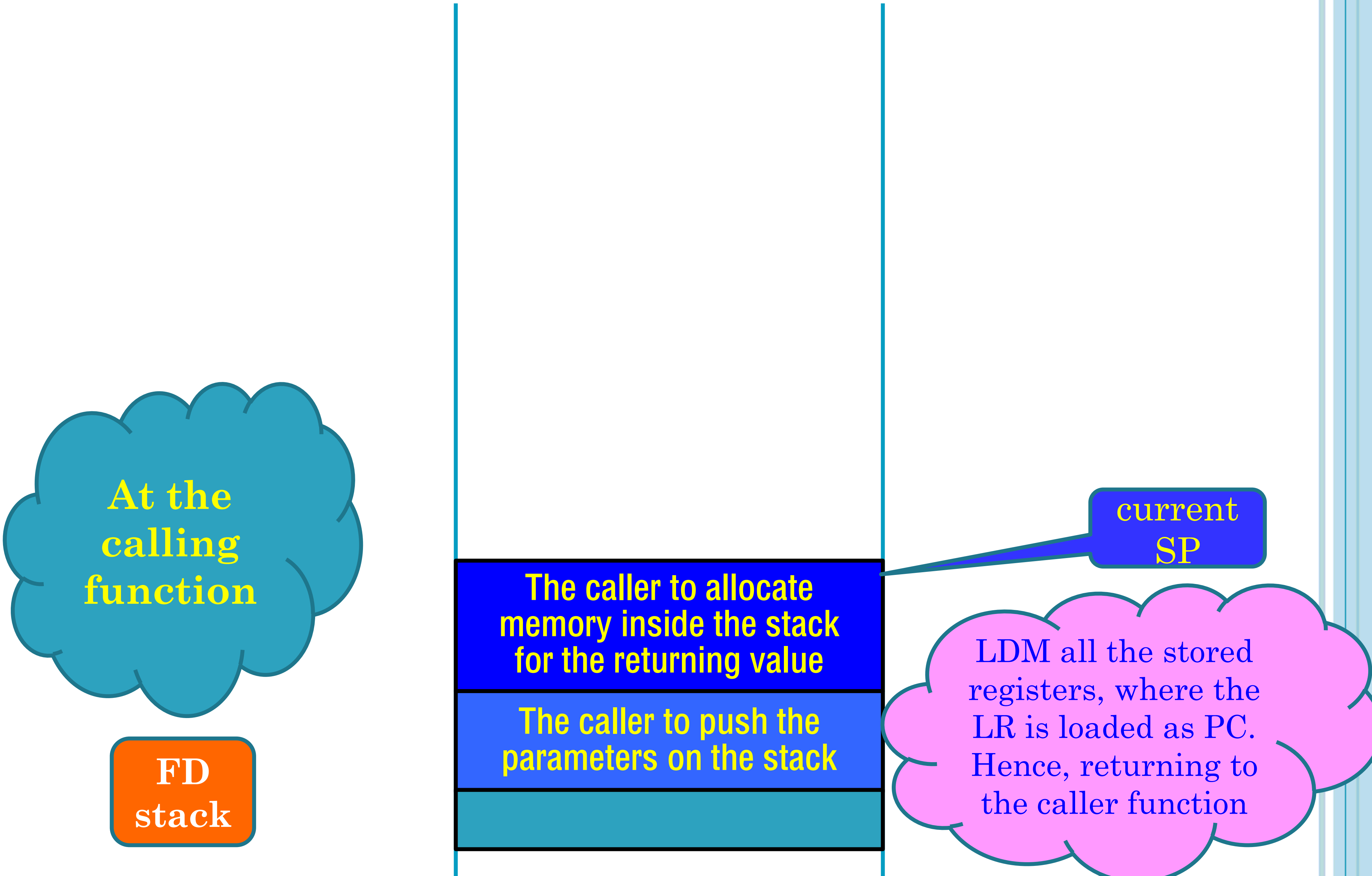
The Traditional Call/Return Mechanism



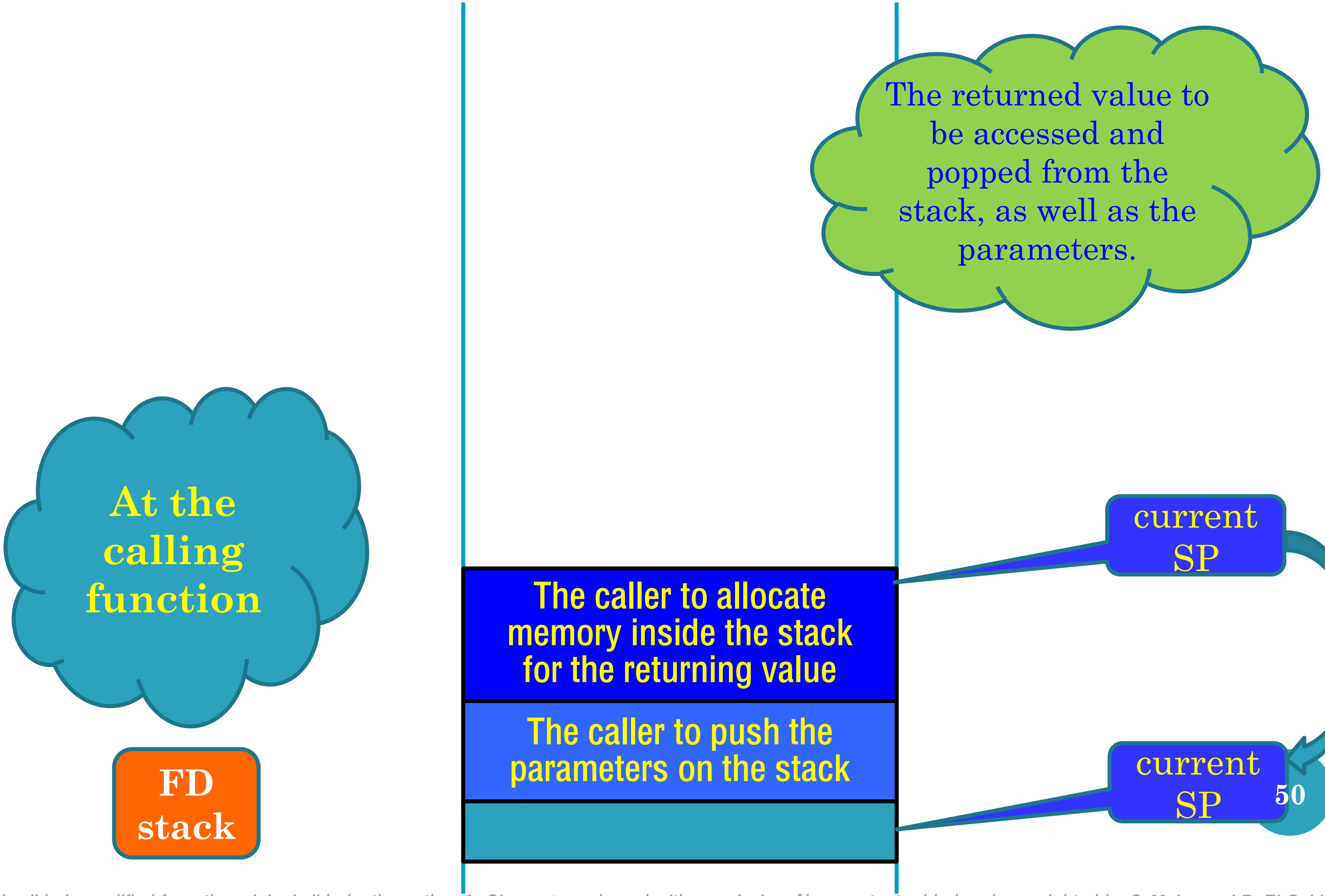
The Traditional Call/Return Mechanism



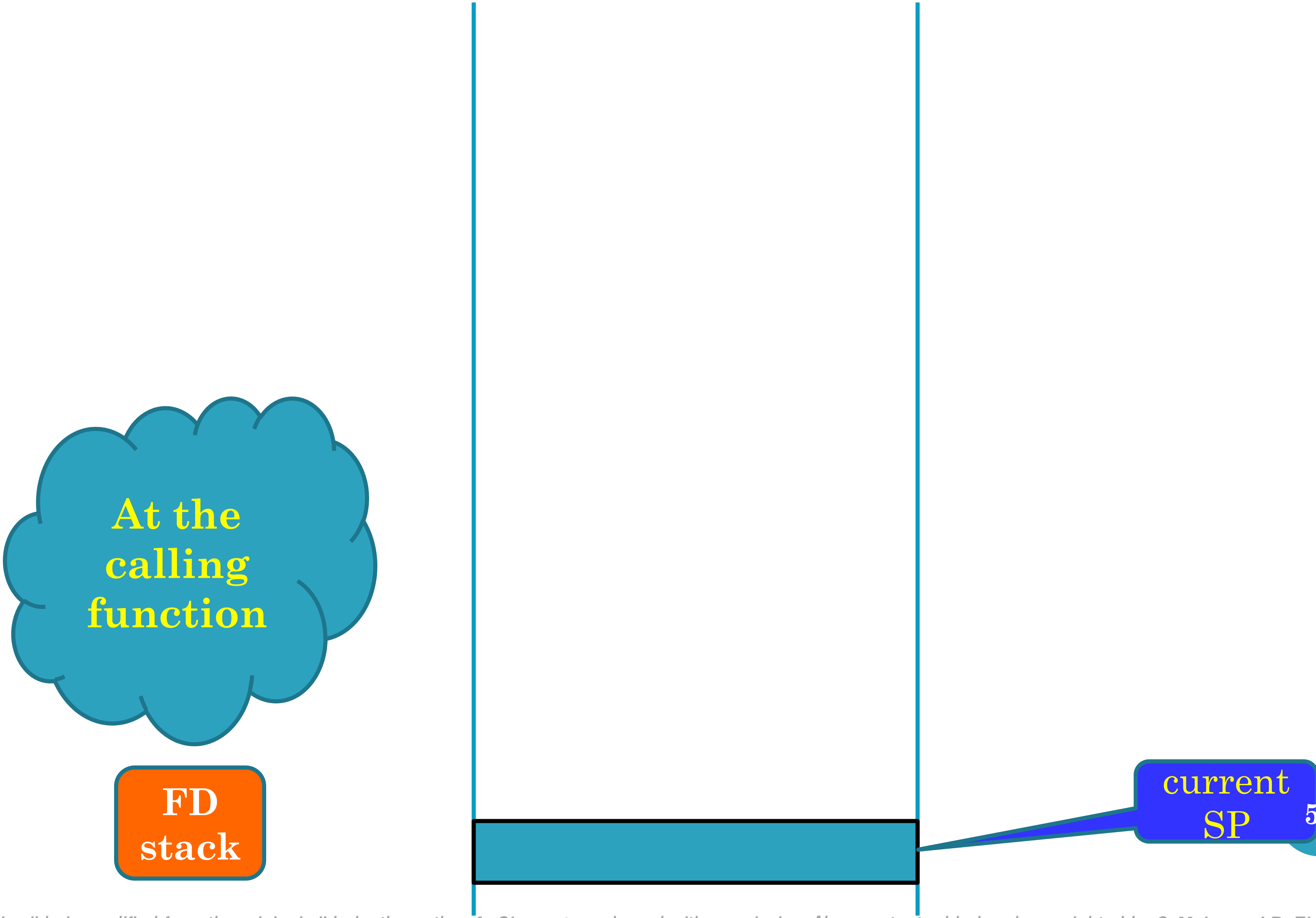
The Traditional Call/Return Mechanism



The Traditional Call/Return Mechanism



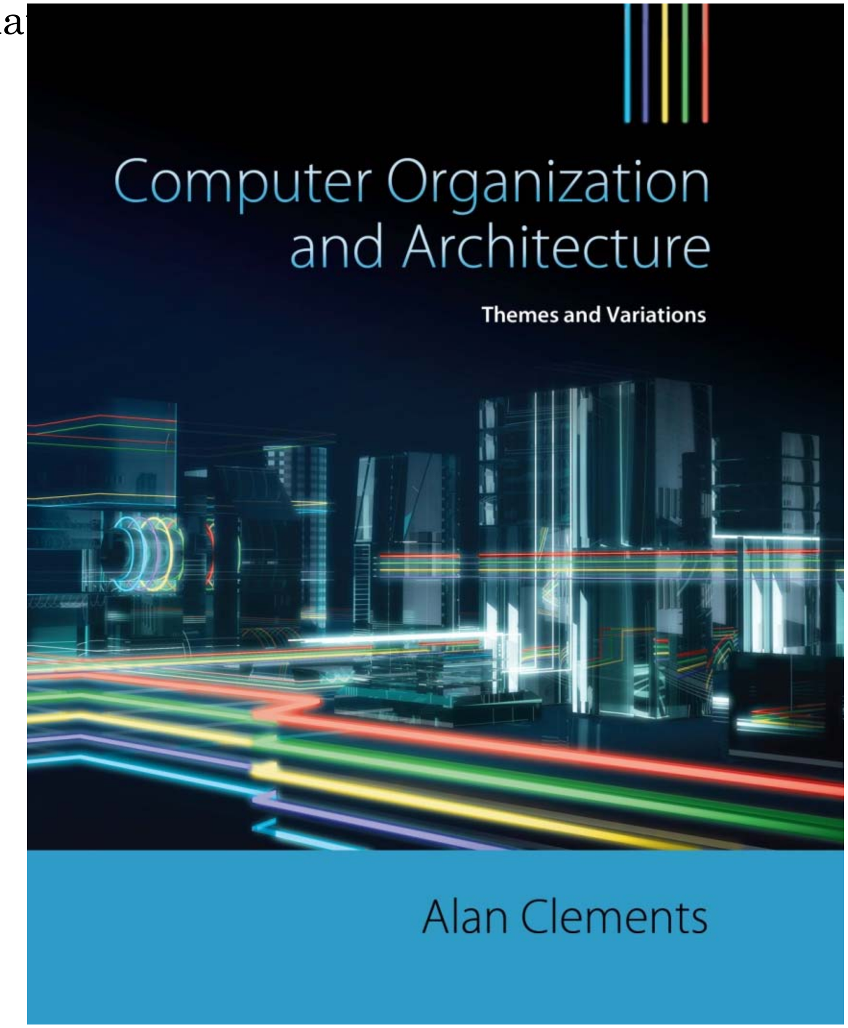
The Traditional Call/Return Mechanism



Part 1

Chapter 4

Computer Organization and Architecture



CENGAGE Learning

ISAs Breadth and Depth

- This chapter extends the overview of ISAs in both breadth and depth.
  - *Yet, we will only cover the depth part in lectures this term*

- In particular, we will look at the role of the stack and architectural support for subroutines and parameter passing.

The Stack and Data Storage

- Let's begin by looking at some background issues concerning *data storage*, *procedures*, and *parameter passing*.

- ***Computer programs*** and ***subroutines*** consist of
  - ***data elements*** and
  - ***procedures*** which operate on these ***data elements***

- High-level language programmers use variables to represent *data elements*
- Variables are declared by:
  - *assigning* names to them and by *reserving* storage for them.
- *Reserving* memory storage for variables can be performed
  - at compilation time (***static memory allocation***)
  - at runtime (***dynamic memory allocation***)

- *Statically allocated variables* will have *static addresses* which *will not* be changed during execution
- *Dynamically allocated variables* will have *dynamic addresses* which *will* be changed during execution, as they will be allocated at runtime

The Stack and Data Storage

- Procedures often require *local workspace* for their temporary variables.
- The term *local* means that the workspace is private to the procedure and is never accessed by the calling program or by other subroutines.
- If a procedure is to be made re-entrant or to be used recursively, its local variables must be bounded up not only with the procedure itself, but with the occasion of its use.
  - Each time the procedure is called, a new workspace must be assigned to it.

The Stack and Data Storage

- A variable has a *scope* associated with it.
  - The scope of a variable defines the range of its *visibility* or *accessibility* within a program.
    - ***Global*** variables are *visible* (accessible) from the moment they are loaded into memory to the moment when the program stops running ***(static memory allocation)***
    - ***Local*** variables and ***parameters*** are *visible* (accessible) *within* that procedure but *invisible* (inaccessible) *outside* the procedure ***(dynamic memory allocation)***

- Here, we are interested to learn more about ***dynamic memory allocation***

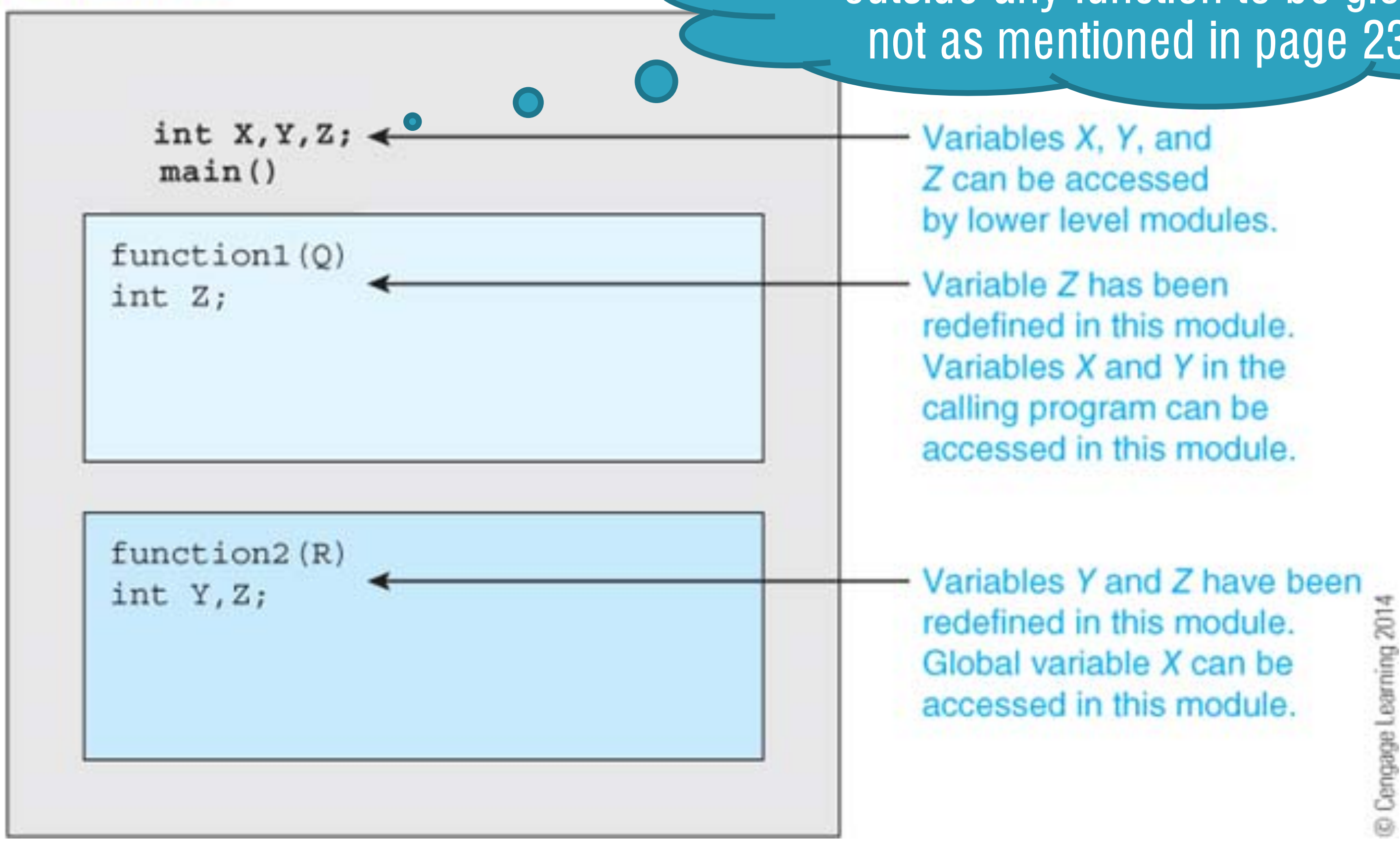
The Stack and Data Storage

- Figure 4.1 illustrates the scope of variables

The *duration* of local variables and parameters are "*automatically*"
- *allocated* when the enclosing *function is called*
  and
- *deallocated* when the *function returns*

The `int X,Y,Z;` should be outside any function to be global, not as mentioned in page 231.

FIGURE 4.1 The concept of scope

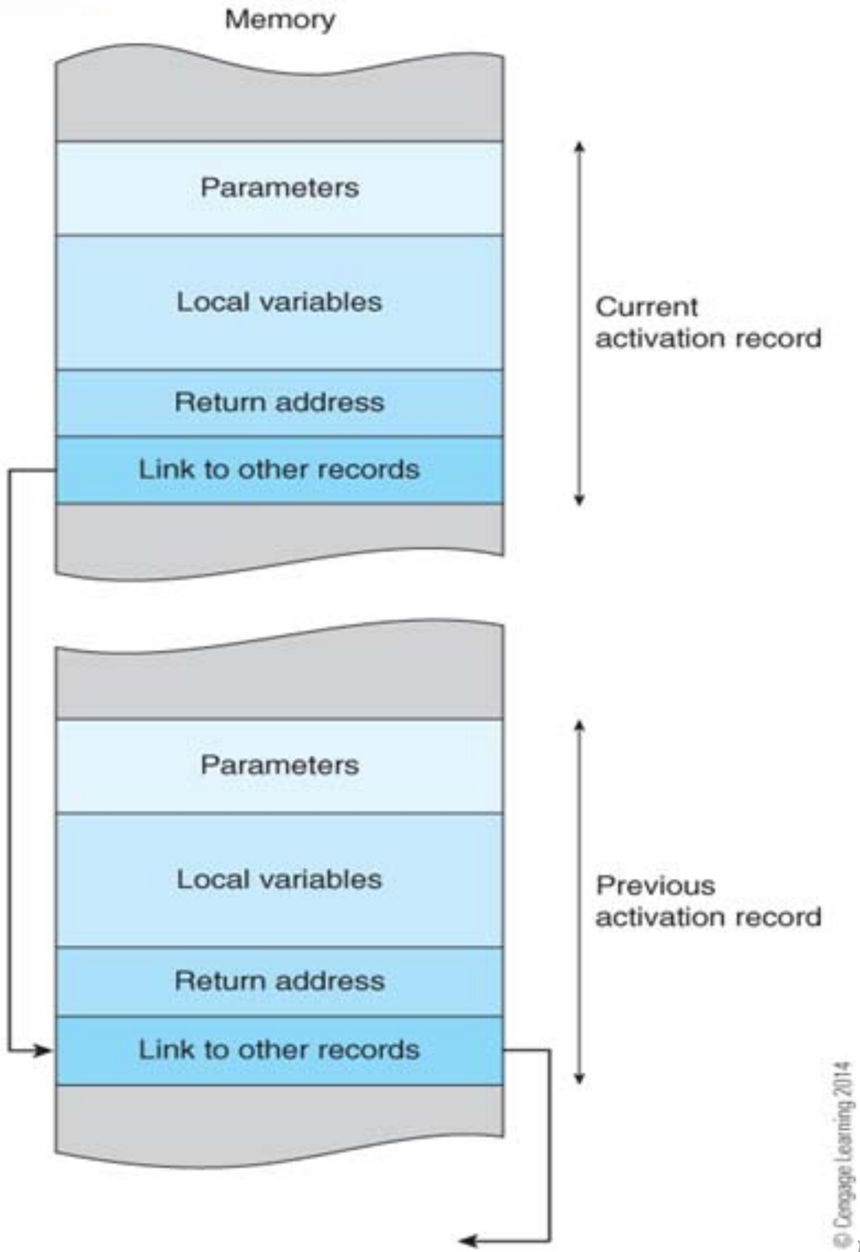


Storage and the Stack

- When a language invokes a procedure, it is said to *activate* the procedure.
- Associated with each invocation (activation) of a procedure, there is an ***activation record*** containing all the information necessary to execute the procedure, including
  - parameters,
  - local variables, and
  - return address,

Storage and the Stack

FIGURE 4.2 The activation record



The elements inside this activation record are not in the correct order.

Storage and the Stack

- The activation record described by Figure 4.2 is known as a *frame*.
- After an activation record has been used, executing a ***return*** *from procedure* *deallocates* or frees the storage taken up by the record.
  - Who should perform this *freeing* process? RISC versus CISC
- Coming next, we will look at how frames are created and managed at the machine level and demonstrate how two pointer registers are used to efficiently implement the activation record creation and deallocation.

Stack Pointer and Frame Pointer

- The stack provides a mechanism for implementing the dynamic memory allocation.

- The stack-frame is a region of temporary storage
  - At the beginning of the subroutine, it will be pushed onto the stack.
  - At the end of the subroutine, it will be popped from the stack.

- The two pointers associated with stack frames are
  - the Stack Pointer, SP (**`r13`**), and
  - the Frame Pointer, FP (**`r11`**).

- A CISC processor maintain a hardware SP that is automatically adjusted when a BSR or RTS is executed.

- RISC processors, like ARM, do not have an explicit SP, although **`r13`** is used as the *ARM's programmer-maintained stack pointer* by convention.
- By convention, **`r11`** is used as a ***frame pointer*** in ARM environments.

The Stack Frame and Local Variables

- The stack pointer always points to the top of the stack.
- The frame pointer always points to the *base of the **current** stack frame*.

- The stack pointer may change during the execution of the procedure, but the frame pointer will not change.
- While the data in the stack frame might be accessed with respect to the stack pointer, it is ***strongly recommended*** to access the data in the stack frame via the stack frame.

The Stack Frame and Local Variables

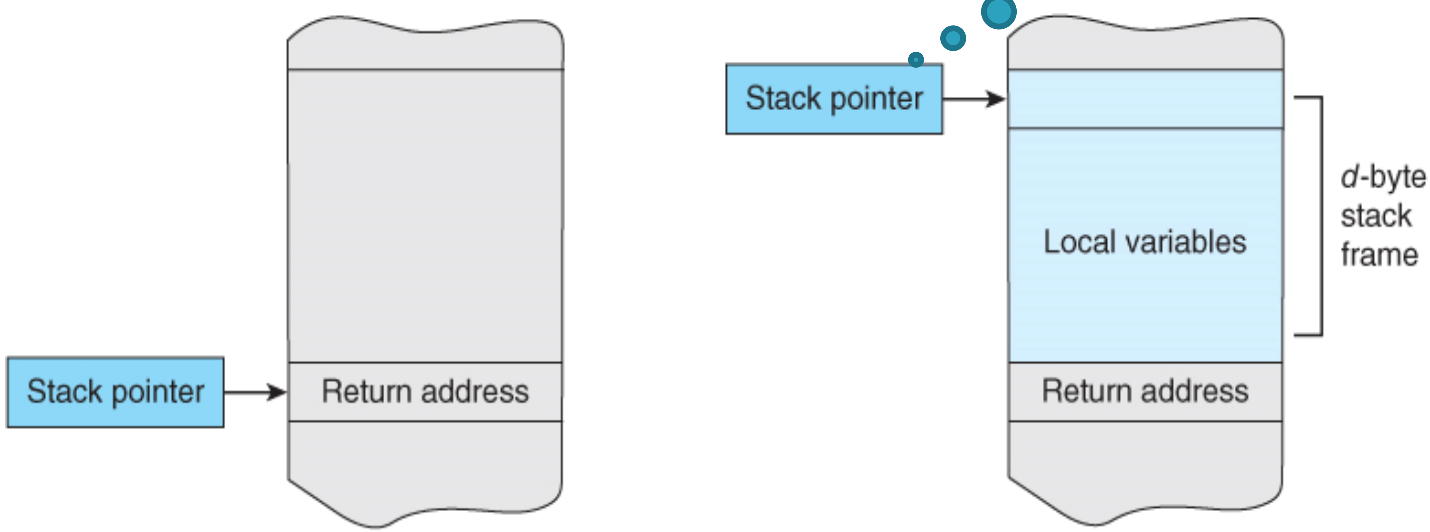
- Assume that the stack that we use grows up towards low addresses and that the stack pointer is always pointing at the item currently at the top of the stack (i.e., FD).

You need to re-do it yourself using the other stack types.

- Figure 4.3 demonstrates how a *d*-byte stack-frame is created by
  - moving the stack pointer up by *d* locations at the start of a subroutine.

As if we are pushing a random block of d-bytes.

FIGURE 4.3 The stack frame



(a) The state of the stack immediately after a subroutine call. Many processors locate the return address at the top of the stack.

(b) The state of the stack after the allocation of a stack frame by moving the stack pointer up *d* bytes.

The Stack Frame and Local Variables

- Because the FD stack grows towards the low end of memory, the stack pointer is decremented to create a stack frame

You need to re-do it yourself using the other stack types.

- Reserving 16 bytes of memory is achieved by

```
SUB r13,r13,#16  ;move the stack pointer up 16 bytes
```

- Before a return from subroutine is made, the stack-frame is collapsed by restoring the stack pointer with

```
ADD r13,r13,#16
```

- In general, operations on the stack are *balanced*; that is, if you put something onto the stack you have to remove it.

The Stack Frame and Local Variables

- Consider the following simple example of a subroutine, where it is called using BL.

```
Proc  SUB r13,r13,#16  ;move the stack pointer up 16 bytes
      Code             ;some code
      STR r1,[r13,#8]  ;store something in the frame 8 bytes
                       ;below TOS
      Code             ;some more code
      ADD r13,r13,#16  ;collapse stack frame
      MOV pc,r14       ;restore the PC to return
```

Bold is not correct in page 235

In this example, FP, i.e., R11, is not used.

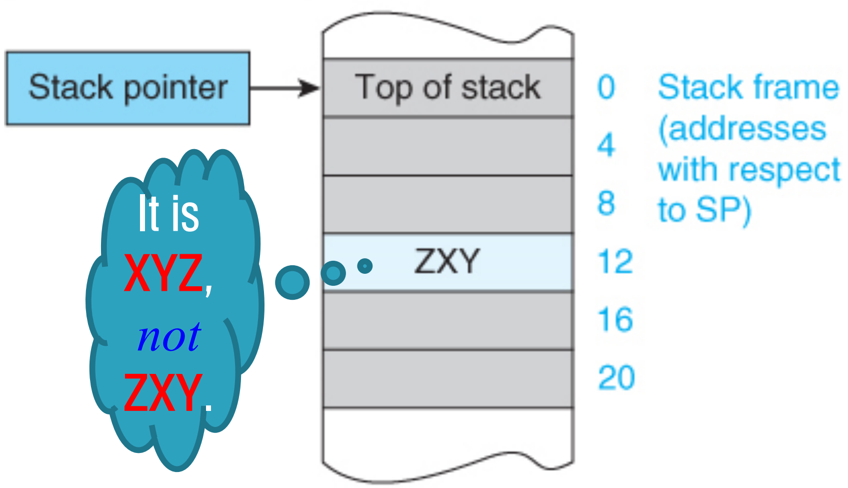
The problem here is that if anything is pushed onto the stack, you have to manually recalculate the position of the stack frame relative to the SP.

The Stack Frame and Local Variables

- In Figure 4.4a variable XYZ is 12 bytes below the stack pointer
  - we access XYZ via address `[r13,#12]`.

- Because the stack pointer is free to move as other information is added to the stack, it is better to construct a stack frame with a pointer independent of the stack pointer.

FIGURE 4.4 Accessing variables in the stack frame



Variable *XYZ* is at SP + 12, twelve bytes below the top of the stack.

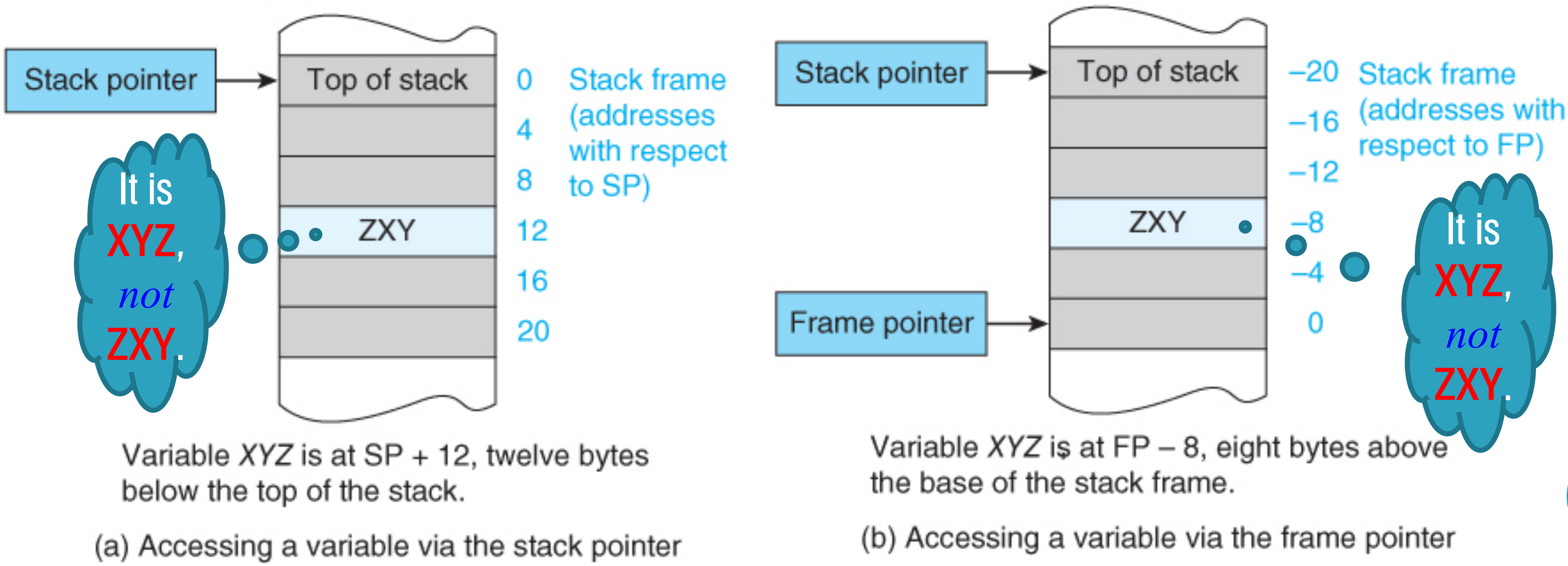
(a) Accessing a variable via the stack pointer

The Stack Frame and Local Variables

- Figure 4.4b illustrates a stack frame with a *frame pointer*, FP, that points to the bottom of the stack frame and is independent of the stack pointer.

- The XYZ variable can be accessed via the frame pointer at **`[r11,#-8]`**, assuming that **`r11`** is the frame pointer.

FIGURE 4.4 Accessing variables in the stack frame



The Stack Frame and Local Variables

- In CISC architecture, a *link* instruction creates a stack frame and an *unlink* instruction collapses it.
- ARM lacks such link and unlink instructions

- To create a stack frame you could
  - push the old *frame pointer* onto the stack (*to save its value*)
  - Make the *frame pointer* to *point to the **bottom of the stack frame***
  - move up the *stack pointer* by *d* bytes (*to create a local workplace*)

```
SUB sp,sp,#4  ;move the stack pointer up by a 32-bit word
STR fp,[sp]   ;push the frame pointer onto the stack
MOV fp,sp     ;move the stack pointer to the frame pointer
SUB sp,sp,#8  ;move stack pointer up 8 bytes
              ;(d is equal to 8)
```

- The *frame pointer*, **fp**, points at the base of the frame and can be used to access local variables in the frame.
- By convention, register **r11** is used as the *frame pointer*.

- At the end of the subroutine, the stack frame is collapsed by:

```
MOV sp,fp     ;restore the stack pointer
LDR fp,[sp]   ;restore old frame pointer from the stack
ADD sp,sp,#4  ;move stack pointer down 4 bytes to
              ;restore stack
```

The Stack Frame and Local Variables

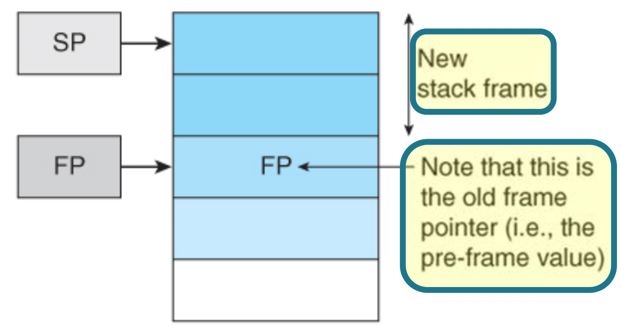
- Figure 4.5 demonstrates how the stack frame grows.
- Note that, the FP appears *twice*;
  - as the old/previous stack frame onto the stack and
  - as the current stack frame pointing to the base of the stack frame.

```
SUB  sp,sp,#4   ;move the stack pointer up by a 32-bit word
STR  fp,[sp]    ;push the frame pointer onto the stack
MOV  fp,sp      ;move the stack pointer to the frame pointer
SUB  sp,sp,#8   ;move stack pointer up
                ;8 bytes (d is equal to 8)
```

FIGURE 4.5 Demonstration of a stack frame

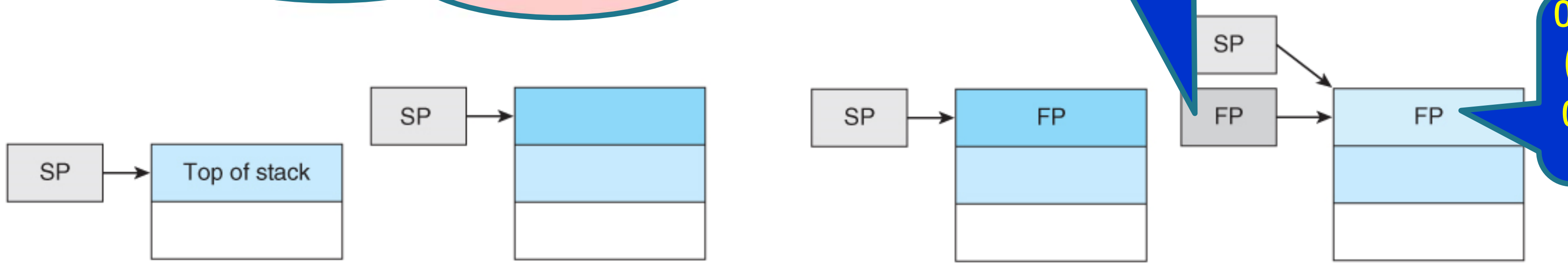
Can be optimized by one instruction:
`STR fp, [sp,#-4]!`
or
`STMFD sp!,{fp}`

new/current FP value (pointing to the base of the current stack frame)



(e) Move up stack pointer by 8 bytes to create local workspace `SUB sp,sp,#8`

old/previous FP value (pointing to the base of the previous stack frame)



(a) Initial state of stack

(b) Reserve space on stack for frame pointer `SUB sp,sp,#4`

(c) Save old frame pointer on stack `STR fp,[sp]`

(d) New frame pointer points to base of the stack `MOV fp,sp`

The Stack Frame and Local Variables

- The figure below demonstrates how the stack frame collapses.

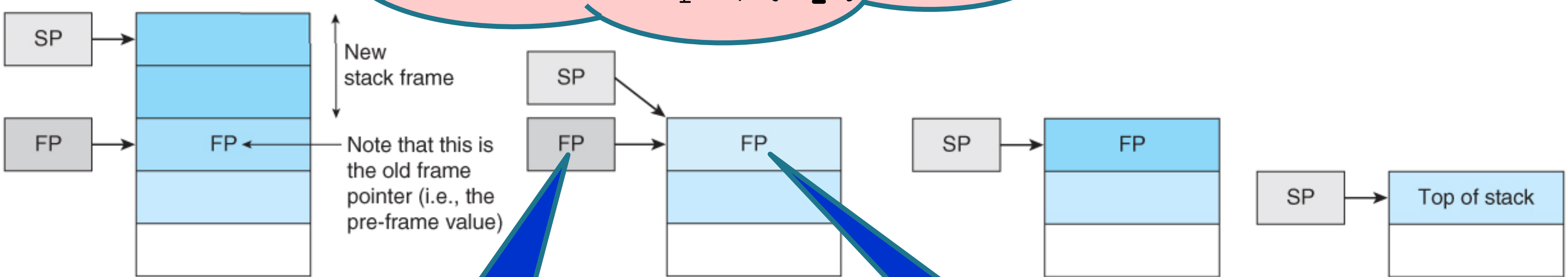
```
MOV sp,fp        ;restore the stack pointer
LDR fp,[sp]      ;restore old frame pointer from the stack
ADD sp,sp,#4     ;move stack pointer down 4 bytes to
                 ;restore stack
```

Can be optimized by one instruction:

```
LDR fp,[sp],#4
```

or

```
LDMFD sp!,{fp}
```



new/current FP value (pointing to the base of the current stack frame)

old/previous FP value (pointing to the base of the previous stack frame)

ARM's Subroutine Example with Stack Frame

- The following demonstrates how you might set up your program.
  - push the parameter onto the stack,
  - ***call*** a subroutine,
    - save ***at least*** the frame pointer and link register,
    - set the frame pointer and create local variables inside the stack
    - perform the subroutine code
    - clean the stack from the created local variables
    - restore saved registers
    - ***return*** to the calling point.
  - pop the parameter from the stack

subroutine

ARM's Subroutine Example with Stack Frame

```
    AREA TestProg, CODE, READONLY
    ENTRY     ;This is the calling environment.
              ;subroutine code is on the next slide.
```

```
Main ADR   sp,Stack       ;set up r13 as the stack pointer
     MOV   r0,#124        ;set up a dummy parameter in r0
     MOV   fp,#123        ;set up dummy frame pointer
     STR   r0,[sp,#-4]!   ;push the parameter
     BL    Sub            ;call the subroutine
     LDR   r1,[sp],#4     ;pop the parameter
Loop B     Loop           ;wait here (endless loop)
```

Missing the post update value in page 237

Bold is not correct in page 237

ARM's Subroutine Example with Stack Frame

```
Sub   STMFD sp!,{fp,lr}   ;push frame-pointer and link-register
      MOV   fp,sp         ;frame pointer at the bottom of
                          ;the frame
      SUB   sp,sp,#4      ;create the stack frame (one word)
      LDR   r2,[fp,#8]    ;get the pushed parameter
      ADD   r2,r2,#120    ;do a dummy operation on
                          ;the parameter
      STR   r2,[fp,#-4]   ;store it in the stack frame
      ADD   sp,sp,#4      ;clean up the stack frame
      LDMFD sp!,{fp,pc}   ;restore frame pointer and return

       DCD   0x0000       ;clear memory
       DCD   0x0000
       DCD   0x0000
       DCD   0x0000
Stack  DCD   0x0000       ;start of the stack

      END
```

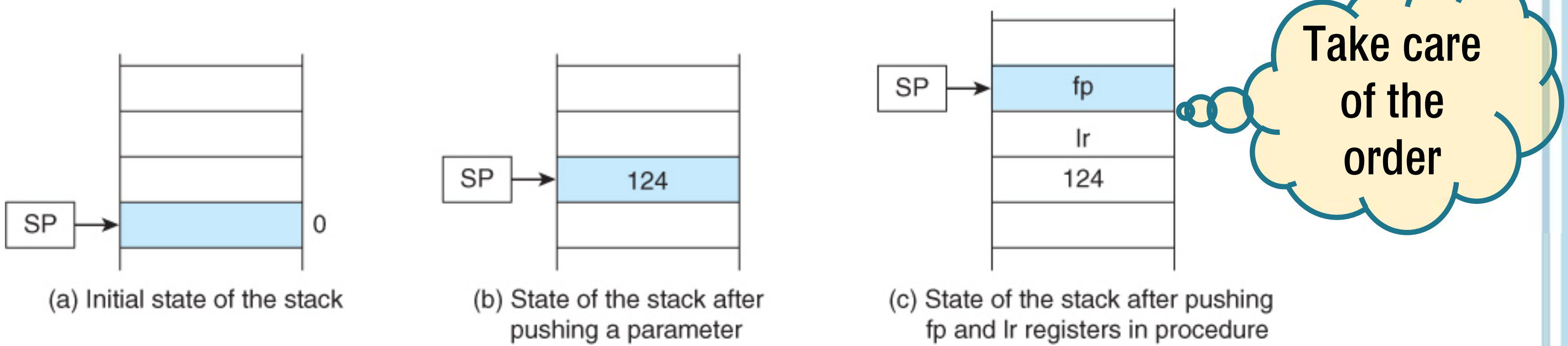
Bold is not correct in page 238

R2 has been changed inside the subroutine. Hence, it should be saved at the beginning of the subroutine and restored at the end.

ARM's Subroutine Example with Stack Frame

- Figure 4.6 demonstrates the behavior of the stack during the code's execution. Figure 4.6a depicts the stack's initial state. In Figure 4.6b the parameter has been pushed onto the stack. In Figure 4.6c the frame pointer and link register have been stacked by `STMFD sp!,{fp,lr}`.

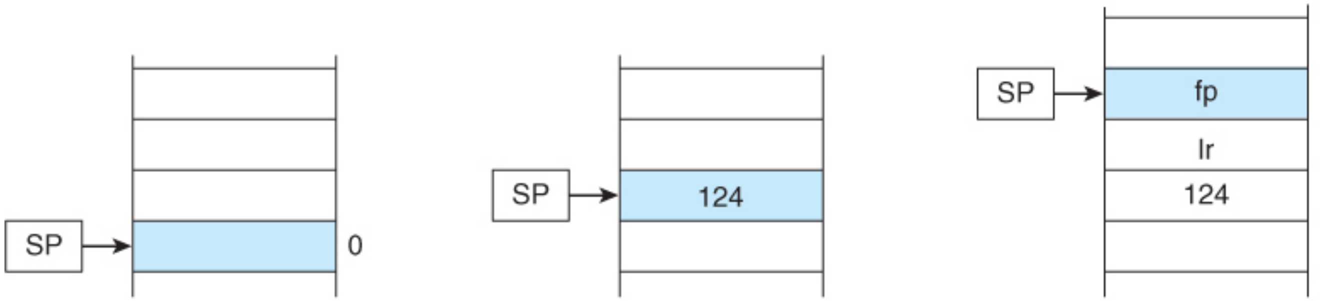
FIGURE 4.6 The behavior of the stack during the execution of the code



ARM's Subroutine Example with Stack Frame

- In Figure 4.6d a 4-byte word has been created at the top of the stack. Finally, Figure 4.6e demonstrates how the pushed parameter is accessed and moved to the new stack frame using register indirect addressing with the frame pointer.

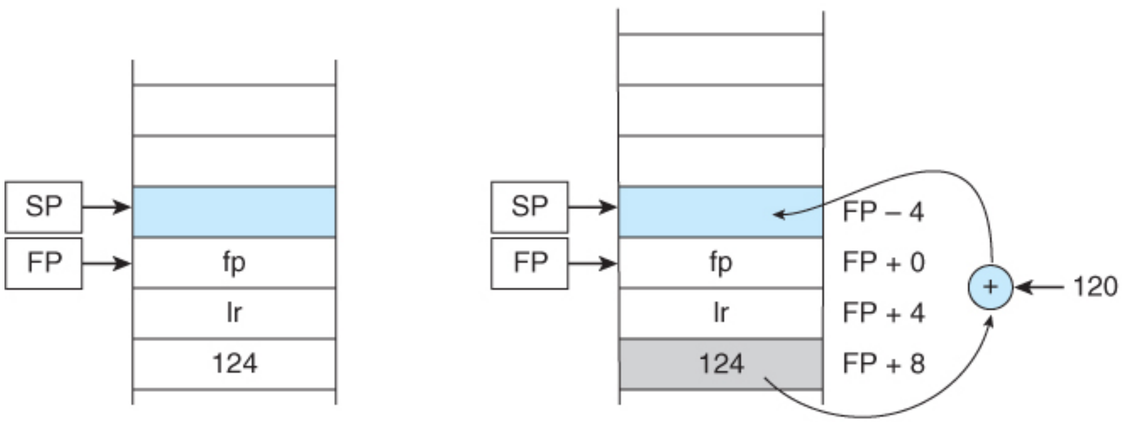
FIGURE 4.6 The behavior of the stack during the execution of the code



(a) Initial state of the stack

(b) State of the stack after pushing a parameter

(c) State of the stack after pushing fp and lr registers in procedure



(d) State of stack after creating 4-byte space on the stack

(e) State of stack after the sequence

```
LDR r2,[fp,#8]  ;get parameter
ADD r2,r2,#120  ;add 120
STR r2,[fp,#-4] ;store sum in stack frame
```