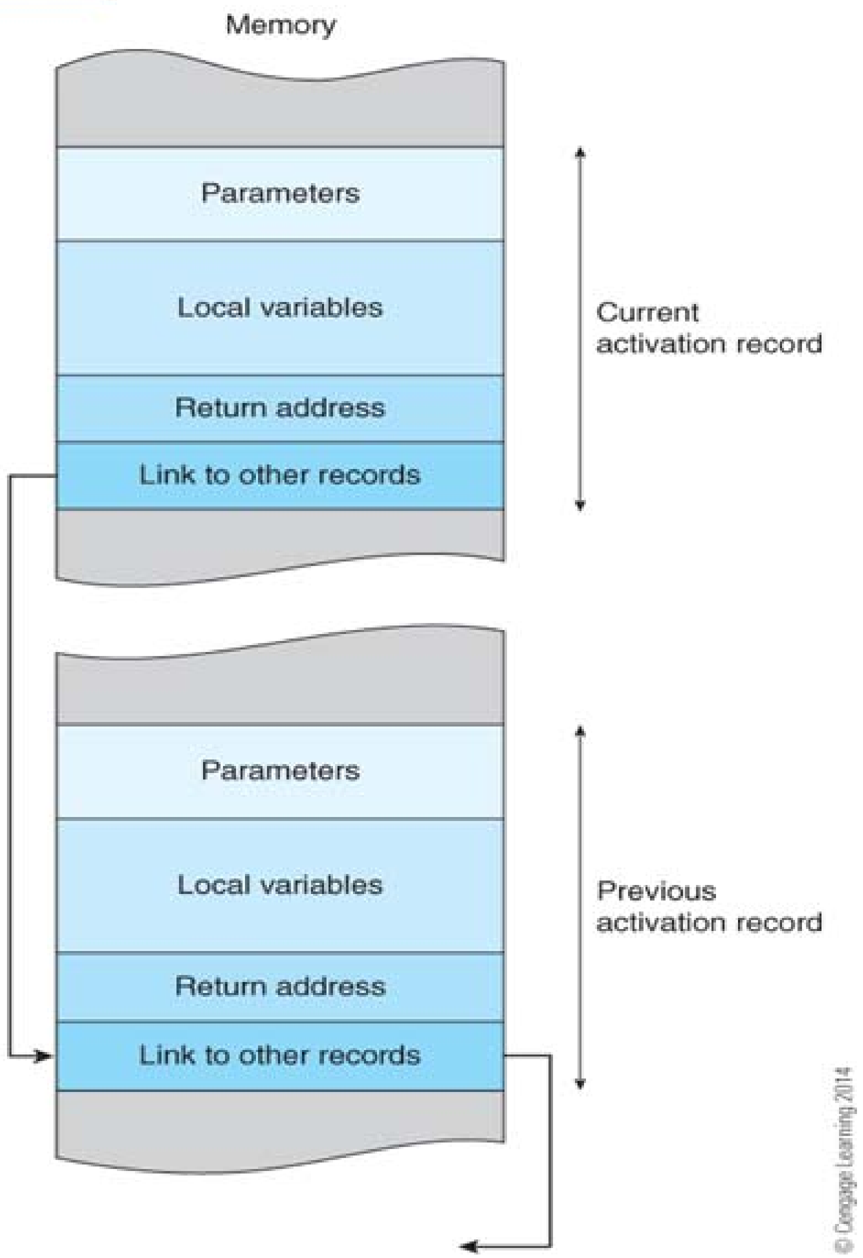
Storage and the Stack

- 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

- We now 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 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 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.
- Data in the stack frame may be accessed with respect to either the stack pointer or the stack frame.

The Stack Frame and Local Variables

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

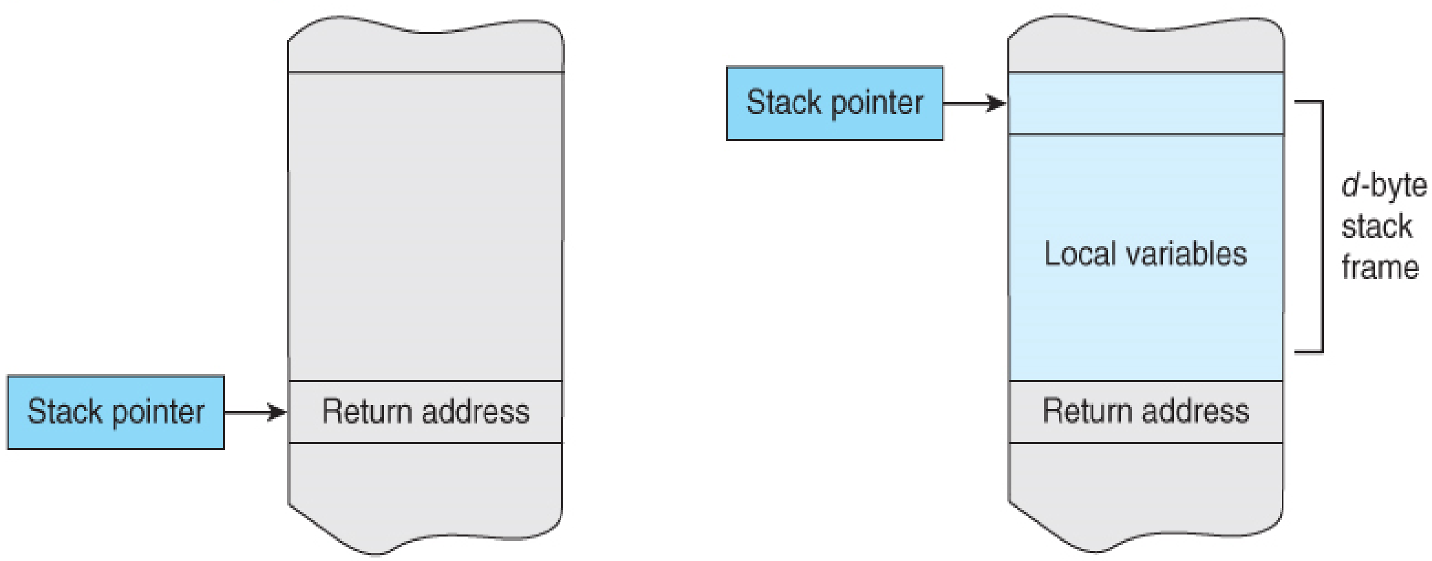
- Two concepts associated with dynamic storage techniques are
  - the Stack Pointer, SP (**`r13`**), and
  - the Frame Pointer, FP (**`r11`**).

- The stack-frame is a region of temporary storage at the top of the current stack.

The Stack Frame and Local Variables

- 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.
- We assume that the stack pointer 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).

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

- Reserving 100 bytes of memory is achieved by

```
SUB r13,r13,#100 ;move the stack pointer up 100 bytes
```

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

ADD r13,r13,#100.

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

The Stack Frame and Local Variables

- Consider the following simple example of a procedure.

```
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
```

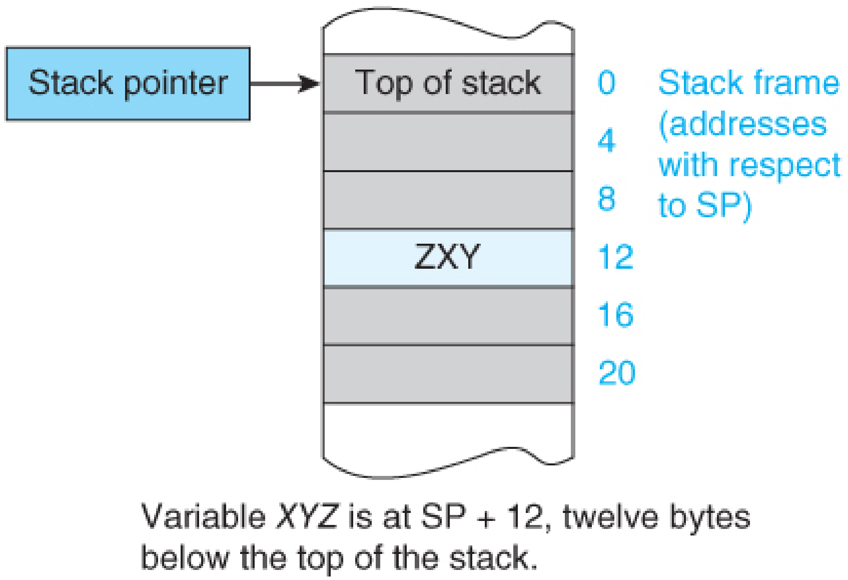
Bold is not correct in page 235

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



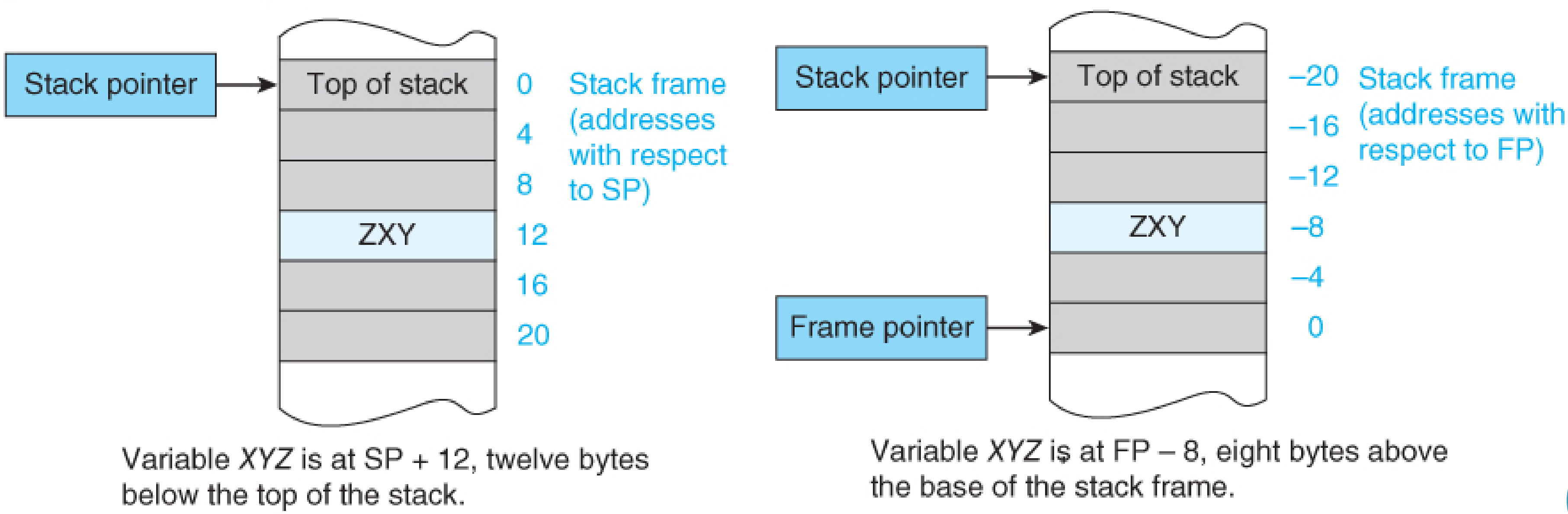
(a) Accessing a variable via the stack pointer

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



(a) Accessing a variable via the stack pointer

(b) Accessing a variable via the frame pointer

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* on 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 on 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

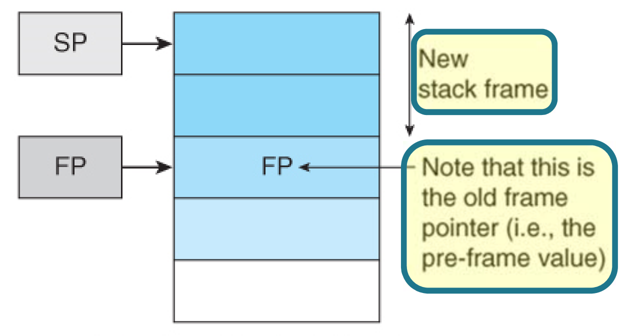
- Figure 4.5 demonstrates how the stack frame grows.
- Note that, the FP appears ***twice***;
  - as the old/previous stack frame on the stack and
  - as the current stack frame pointing to the base of the stack frame.

```
SUB sp,sp,#4  ;move the stack pointer up by a 32-bit word
STR fp,[sp]   ;push the frame pointer on 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

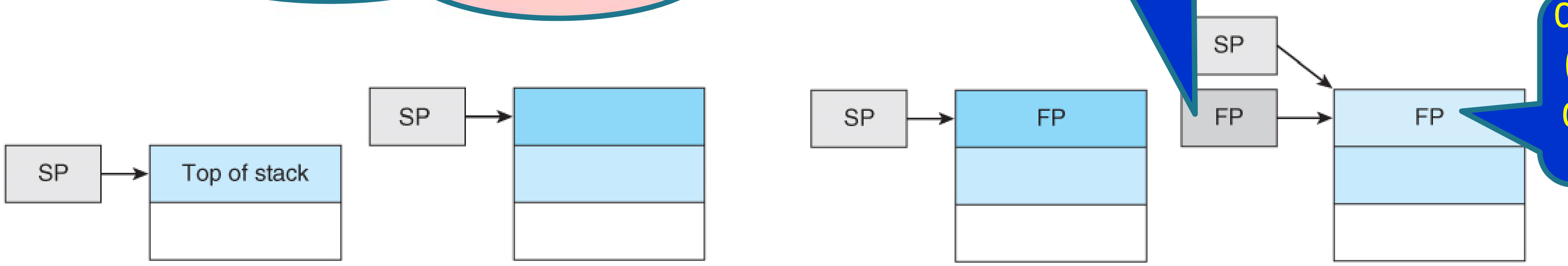
Can be optimized by one instruction:
`STR fp, [sp,#-4]!`
or
`STMFD sp!,{fp}`

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.

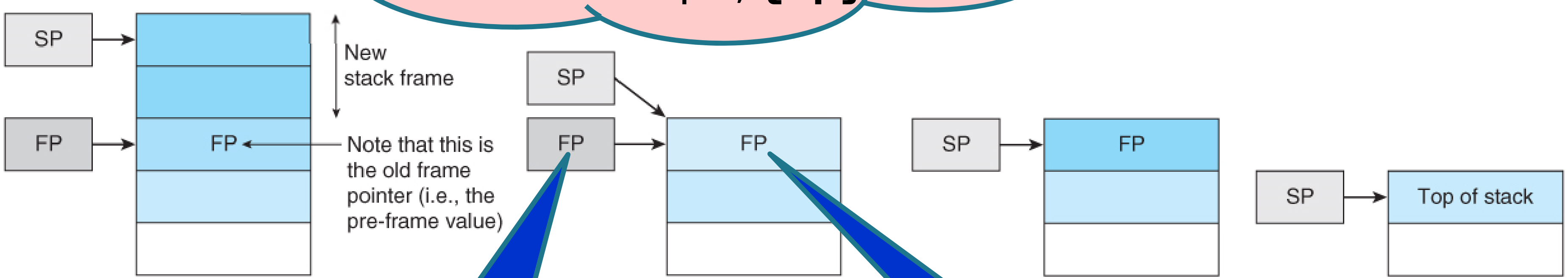
```
MOV sp,fp      ;restore the stack pointer
LDR fp,[sp]    ;restore old frame pointer from the stack
ADD sp,sp,#4   ;move stack pointer down 4 bytes to
               ;restore stack
```

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)

Example of an ARM processor Stack Frame

- The following demonstrates how you might set up a stack frame:
  - push the parameter on 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

Example of an ARM processor 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)
```

Bold is not correct in page 237

Missing the post update value in page 237

Example of an ARM processor Stack Frame

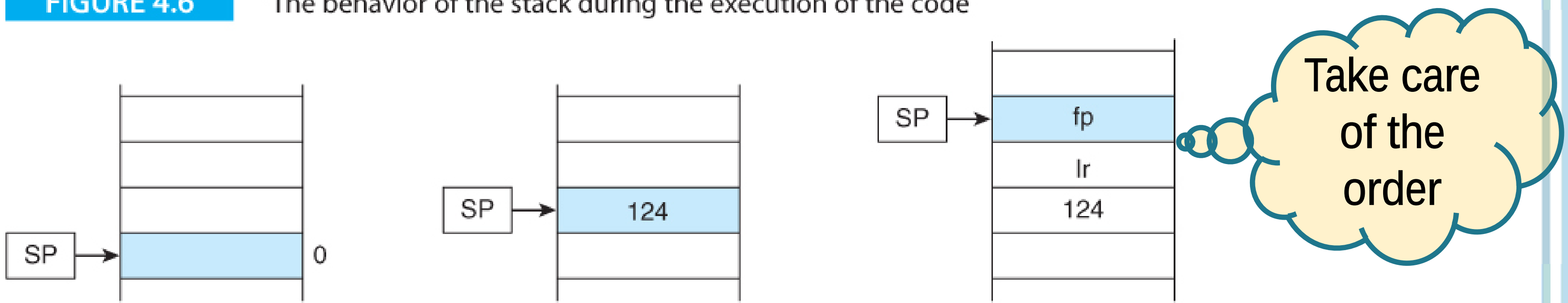
```
Sub   STMFD sp!,{fp,lr}   ;push frame-pointer and link-register
      MOV   fp,sp         ;frame pointer at the bottom of
                          ;the frame
      SUB   sp,sp,#4      ;create the stack frame (one word)
      LDR   r2,[fp,#8]    ;get the pushed parameter
      ADD   r2,r2,#120    ;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

Example of an ARM processor 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 on 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



(a) Initial state of the stack

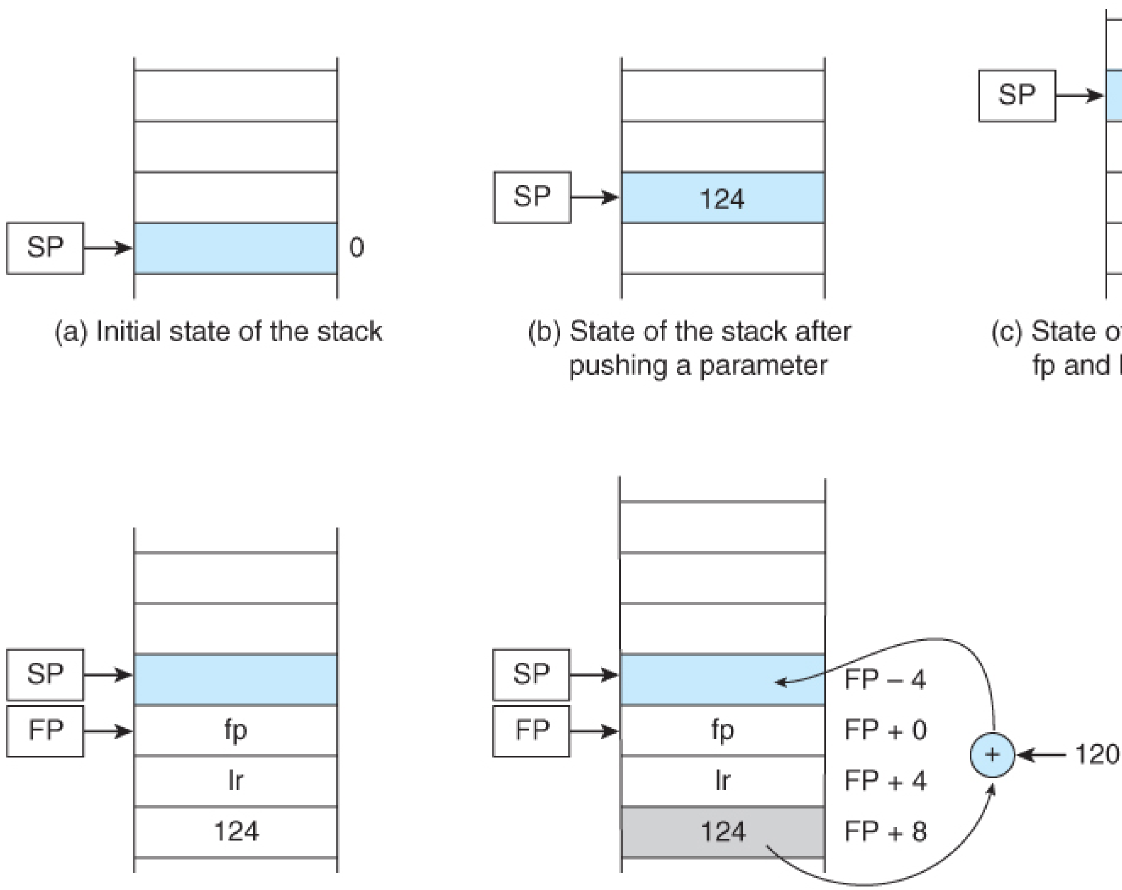
(b) State of the stack after pushing a parameter

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

Example of an ARM processor 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
```