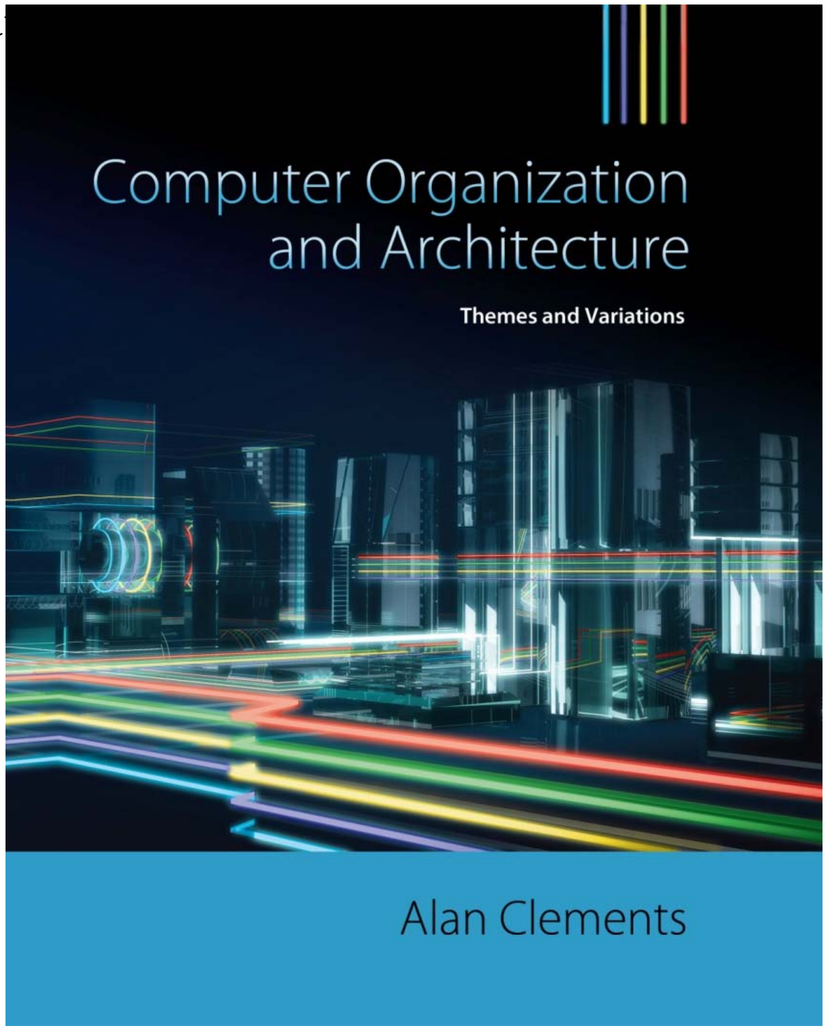
Part 1

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.

  create local variable for all values.

  - 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

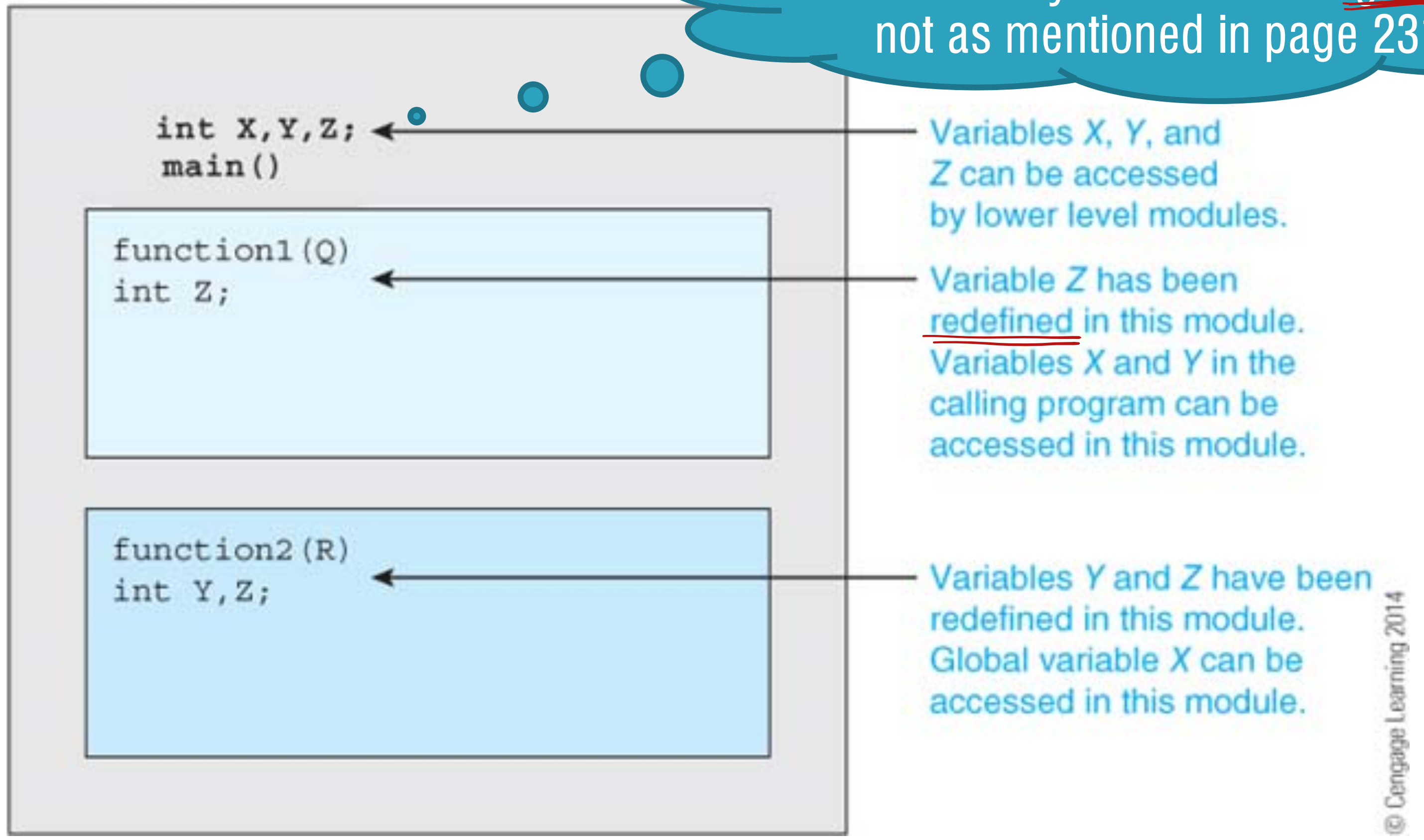
- Figure 4.1 illustrates the scope of variables

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



```
int X,Y,Z;
main()

function1(Q)
int Z;

function2(R)
int Y,Z;
```

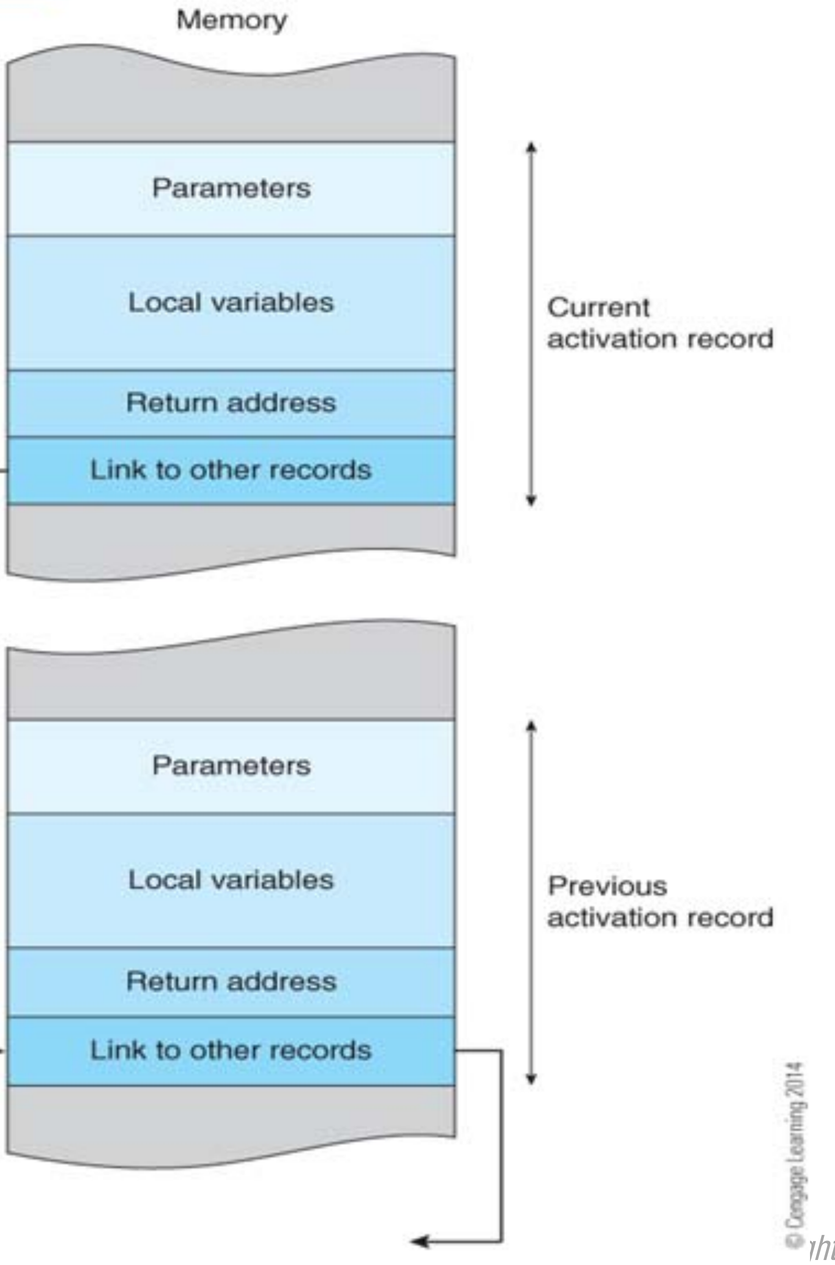
Variables X, Y, and Z can be accessed by lower level modules.
Variable Z has been redefined in this module. Variables X and Y in the calling program can be accessed in this module.
Variables Y and Z have been redefined in this module. Global variable X can be accessed in this module.
© Cengage Learning 2014

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



return to the activation of the function.

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
    programmer processor take care of this.

- 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`**).

r15 → PC
r14 → LR

processor.

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

programmer.

- 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

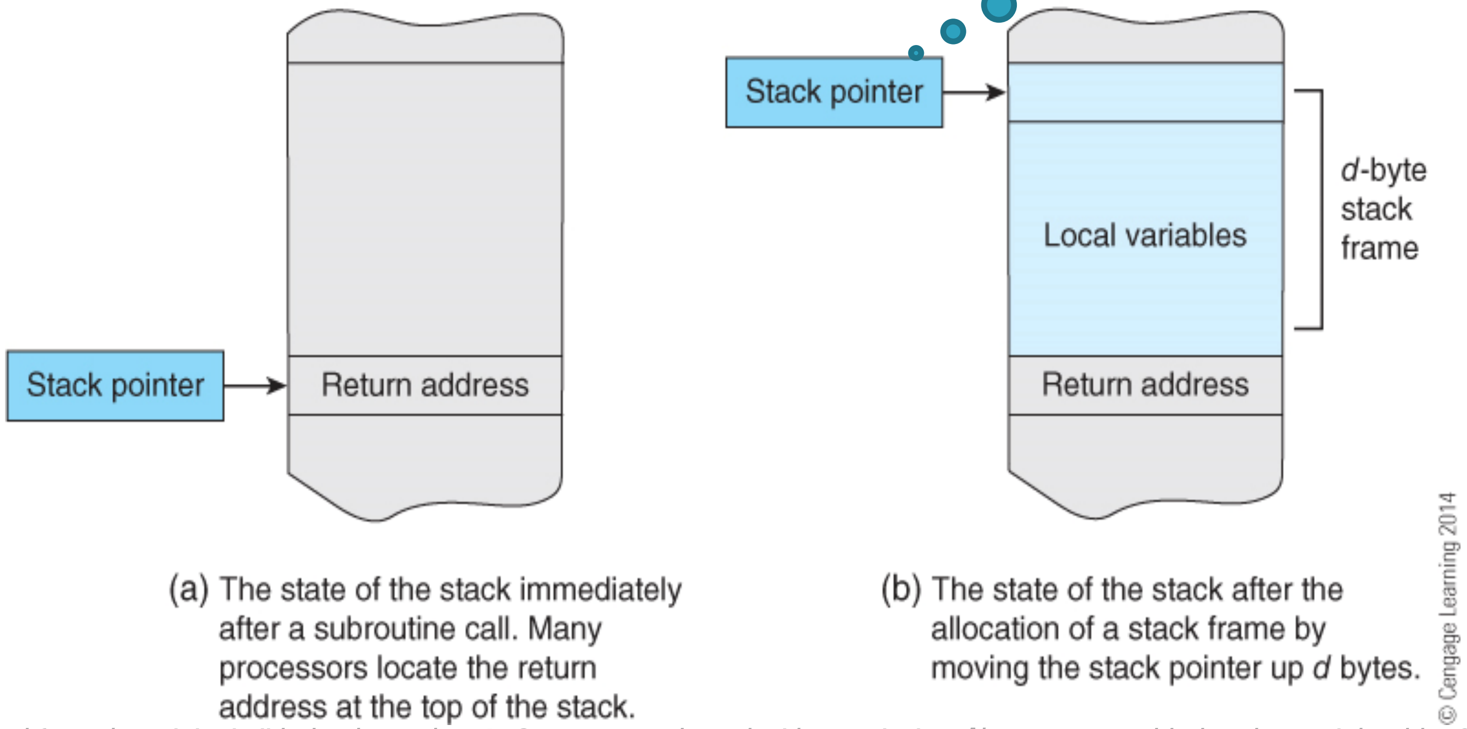
- Assume that the stack that we use grows up towards low addresses and that the stack pointer is always pointing at the item currently at the top of the stack (i.e., FD).

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.

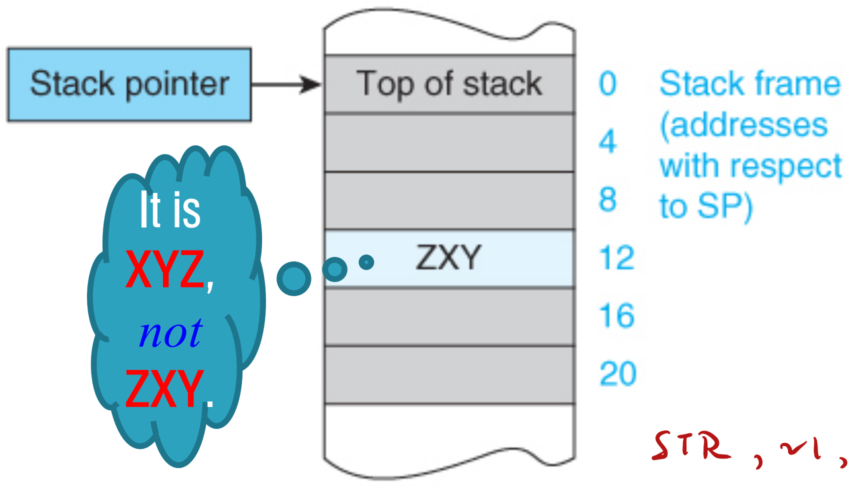
The problem here is that if anything is pushed onto the stack, you have to manually recalculate the position of the stack frame relative to the SP.

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



STR, r1, [r13,#12].

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

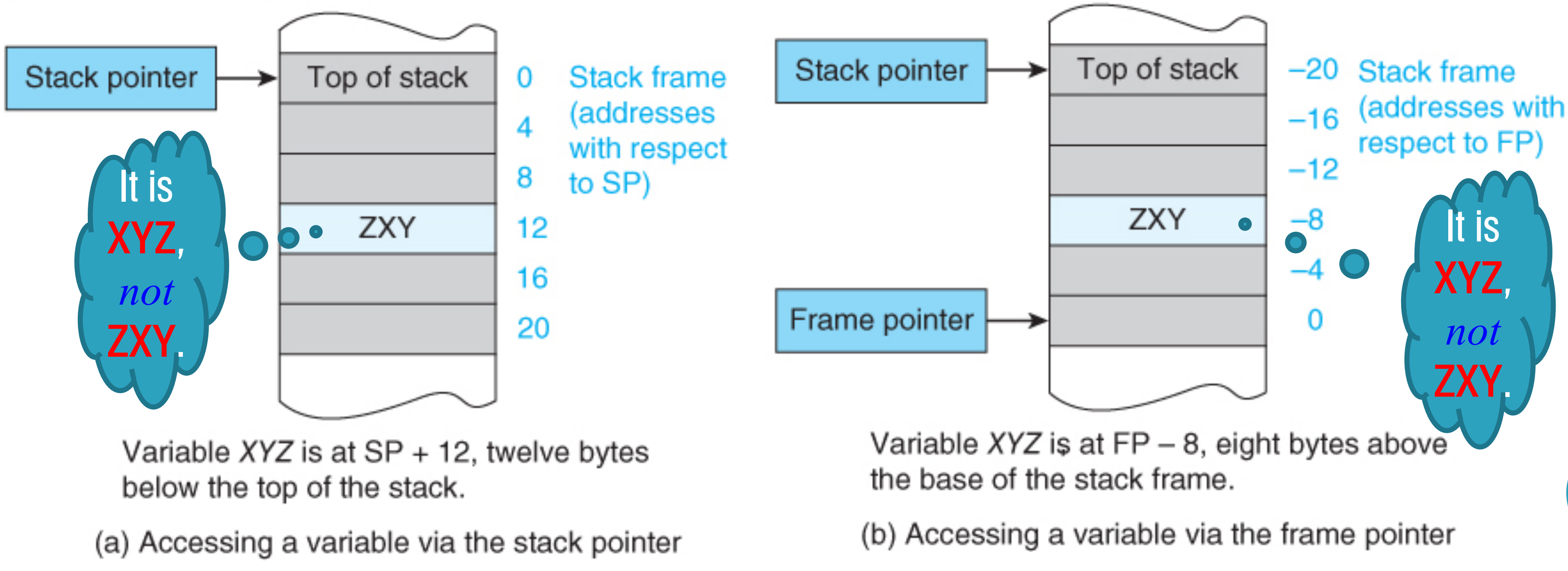
(a) Accessing a variable via the stack pointer

make pointer point to the base of the

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


Stack pointer
Top of stack
ZXY
0
4
8
12
16
20
Stack frame (addresses with respect to SP)
It is XYZ, not ZXY.
Variable XYZ is at SP + 12, twelve bytes below the top of the stack.
(a) Accessing a variable via the stack pointer
Stack pointer
Top of stack
ZXY
Frame pointer
–20
–16
–12
–8
–4
0
Stack frame (addresses with respect to FP)
It is XYZ, not ZXY.
Variable XYZ is at FP – 8, eight bytes above the base of the stack frame.
(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* 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)
```

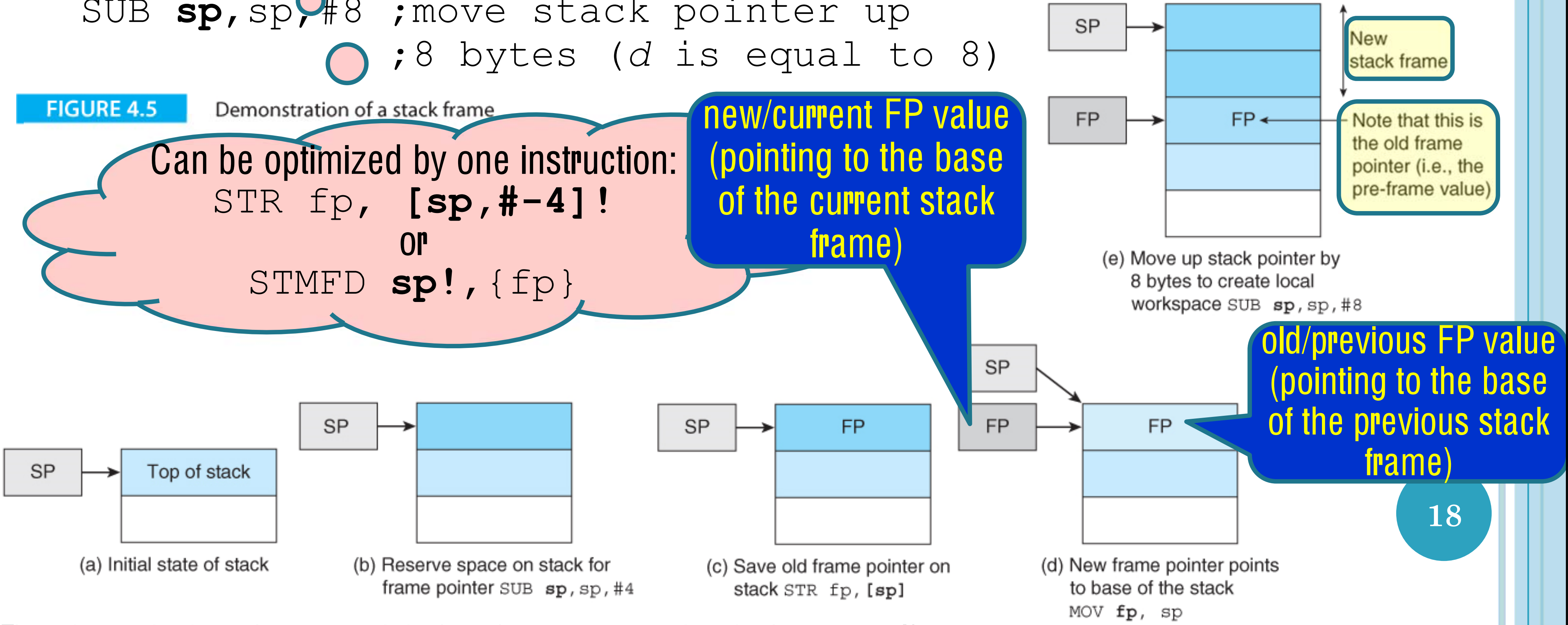
Can be optimized by one instruction:

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

or

```
STMFD sp!,{fp}
```

FIGURE 4.5 Demonstration of a stack frame


SP
Top of stack
(a) Initial state of stack
SP
(b) Reserve space on stack for frame pointer SUB sp,sp,#4
SP
FP
(c) Save old frame pointer on stack STR fp,[sp]
SP
FP
FP
(d) New frame pointer points to base of the stack MOV fp, sp
SP
FP
FP
New stack frame
Note that this is the old frame pointer (i.e., the pre-frame value)
(e) Move up stack pointer by 8 bytes to create local workspace SUB sp,sp,#8
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)

The Stack Frame and Local Variables

- The figure below demonstrates how the stack frame collapses.

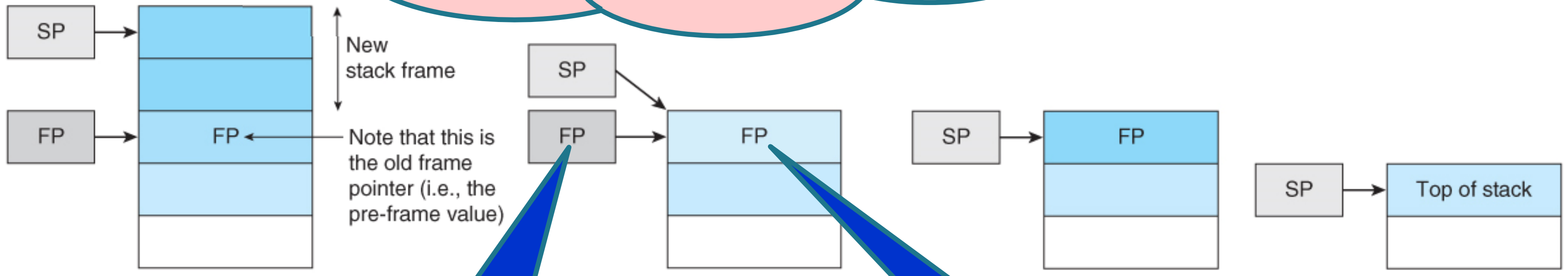
```
MOV sp,fp      ;restore the stack pointer
LDR fp,[sp]    ;restore old frame pointer from the stack
ADD sp,sp,#4   ;move stack pointer down 4 bytes to
               ;restore stack
```

Can be optimized by one instruction:

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

or

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

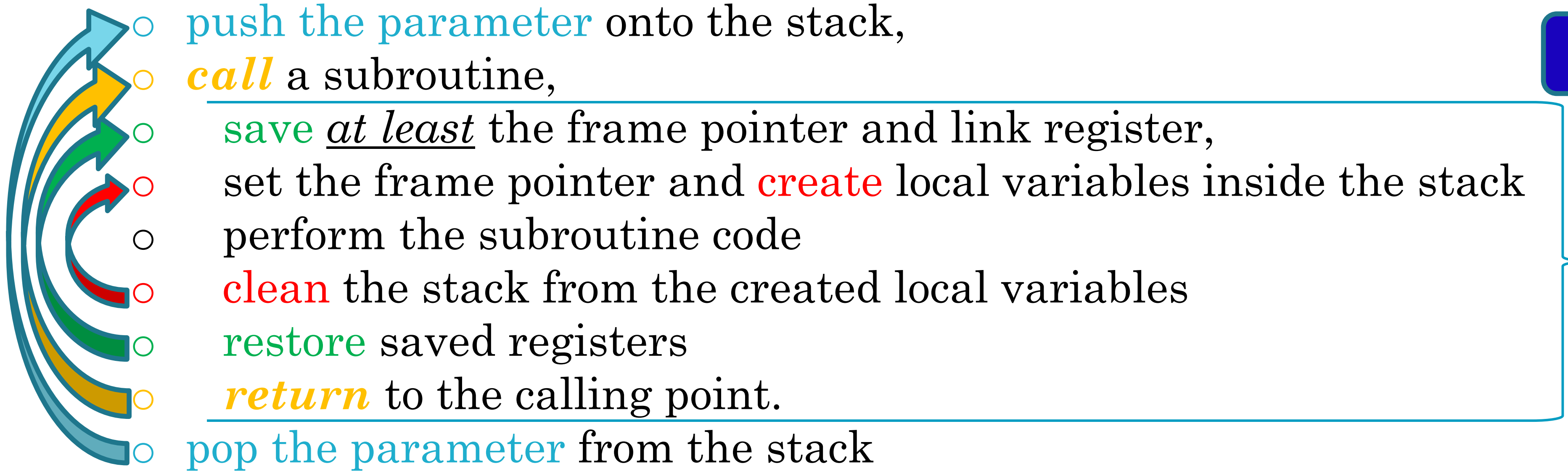

SP
FP
New stack frame
FP
Note that this is the old frame pointer (i.e., the pre-frame value)
SP
FP
FP
SP
FP
SP
Top of stack

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



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
```

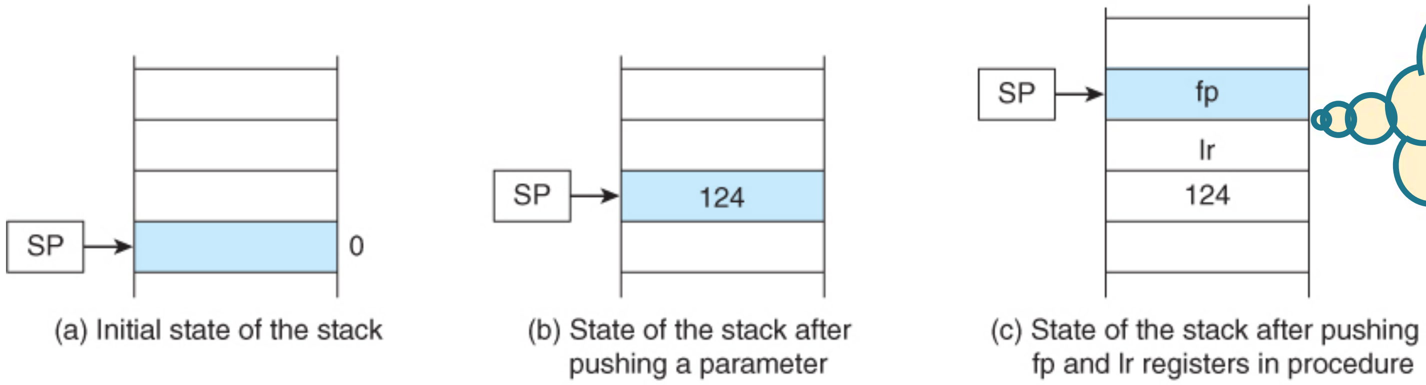
Bold is not correct in page 238

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}`.

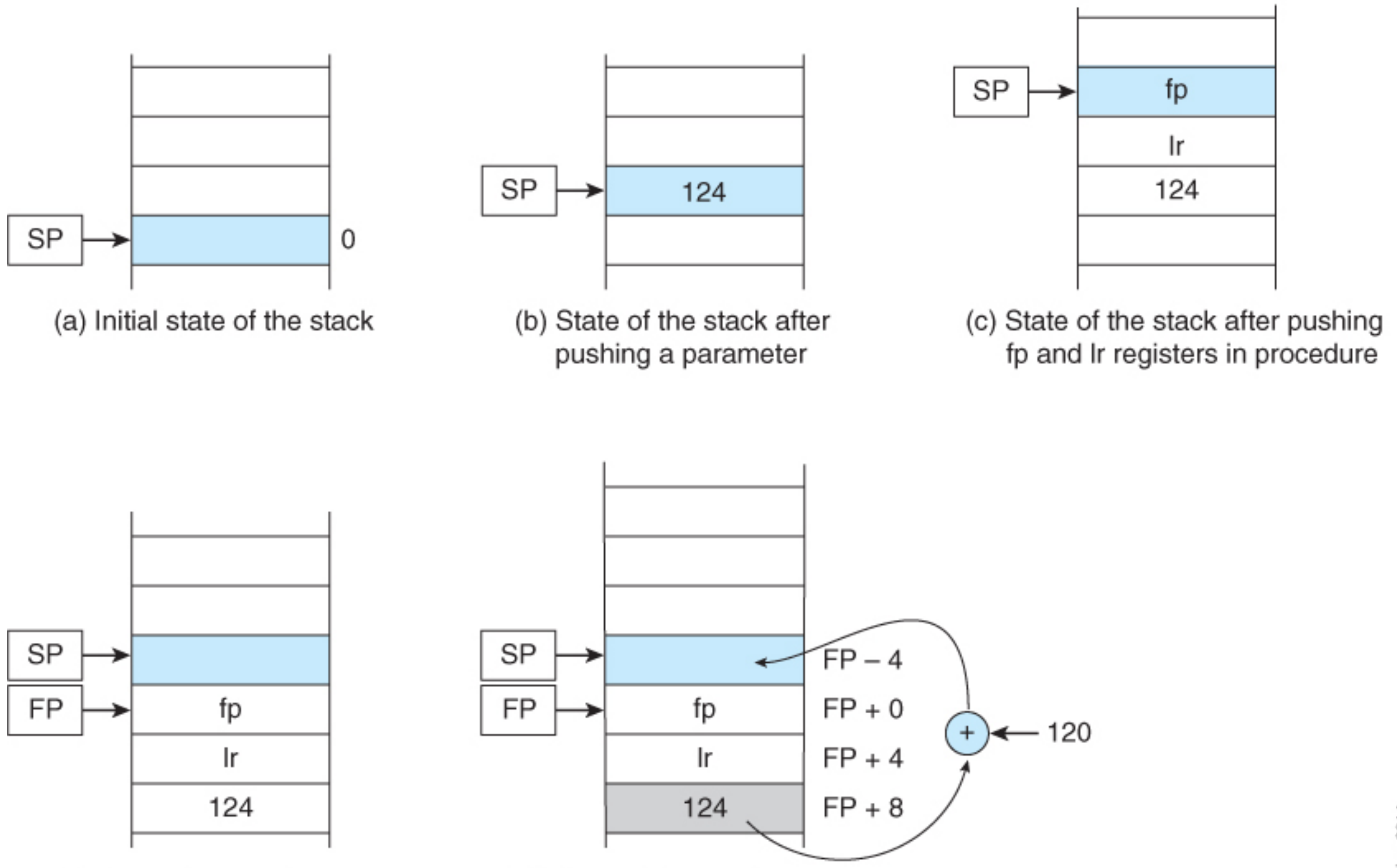
FIGURE 4.6 The behavior of the stack during the execution of the code


SP
0
(a) Initial state of the stack
SP
124
(b) State of the stack after pushing a parameter
SP
fp
lr
124
(c) State of the stack after pushing fp and lr registers in procedure

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
```