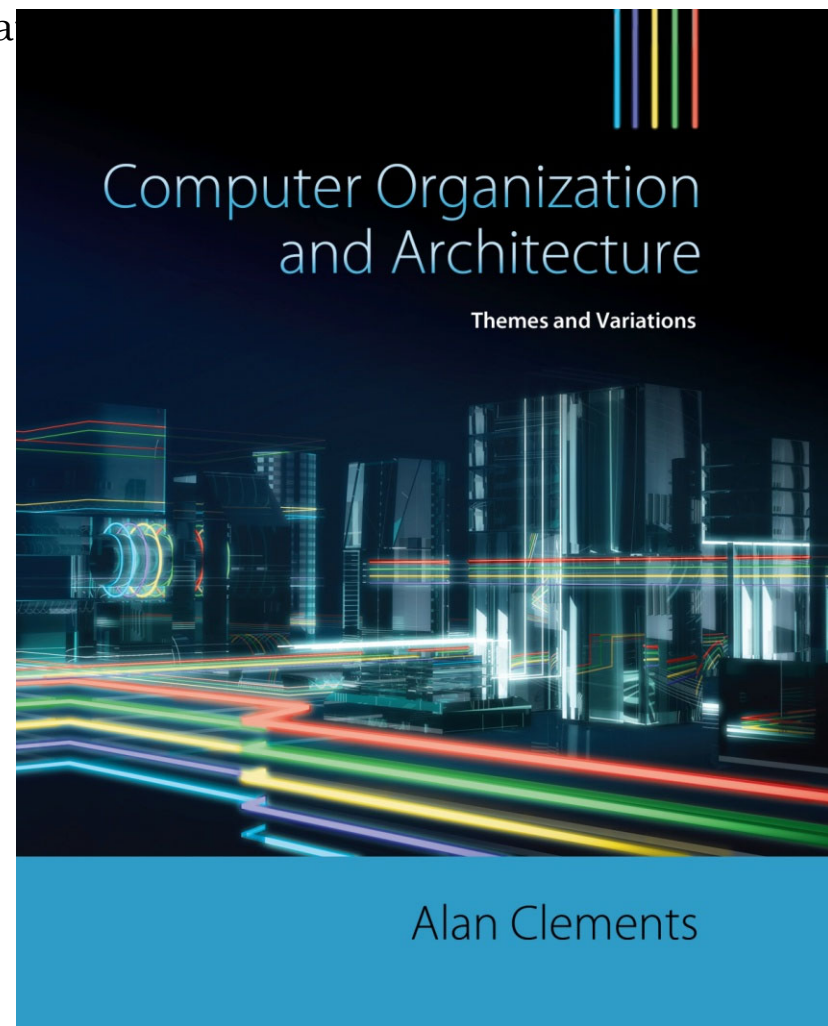


# Part 1

## CHAPTER 4

# Computer Organization and Architecture

1



These slides are being provided with permission from the copyright for CS2208 use only. The slides must not be reproduced or provided to anyone outside of the class.

All download copies of the slides are for personal use only. Students must destroy these copies within 30 days after receipt of final course evaluations.

# 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 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:
  - *reserving* storage for them and *assigning* names to 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 (*block scope*) and **parameters** (*function scope*) 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*

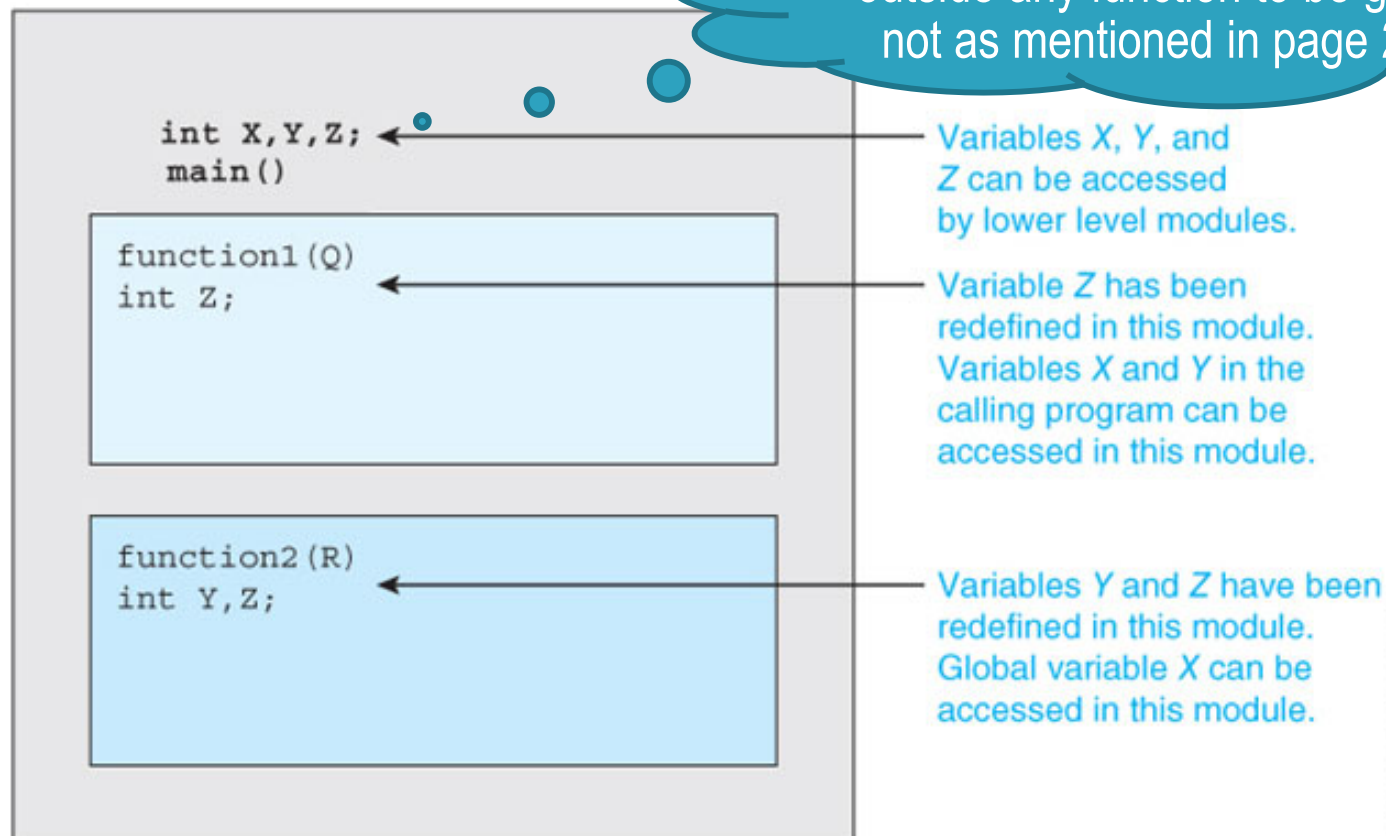
# The Stack and Data Storage

❑ 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*

FIGURE 4.1

The concept of scope



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

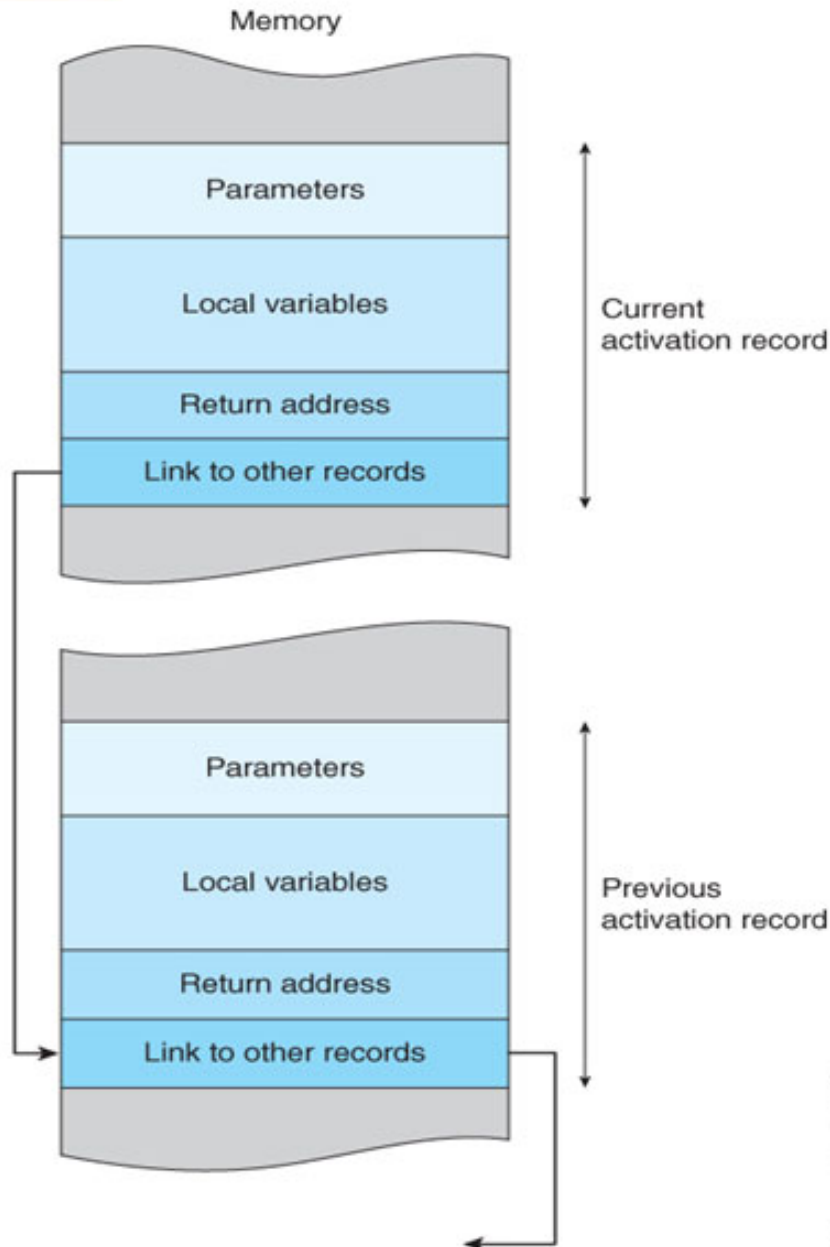
# Storage and the Stack

- ❑ 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

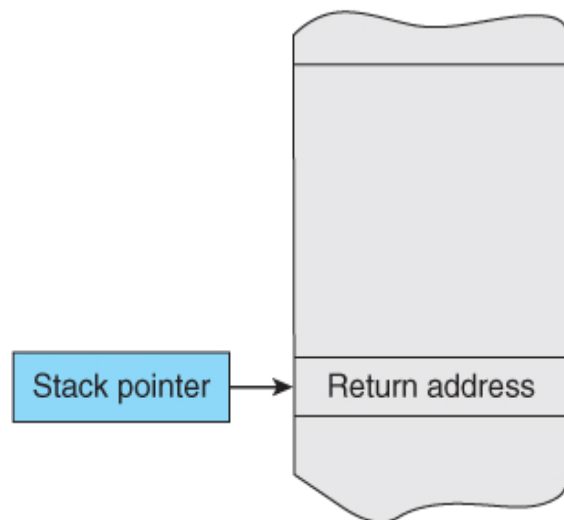
- ❑ The **stack pointer** always points to the **top** of the stack.
- ❑ The **frame pointer** always points to the **base** of the **current** stack frame.
- ❑ The **stack pointer** may change during the execution of the procedure, but the **frame pointer** will not change.
- ❑ While the data in the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

✓!!!

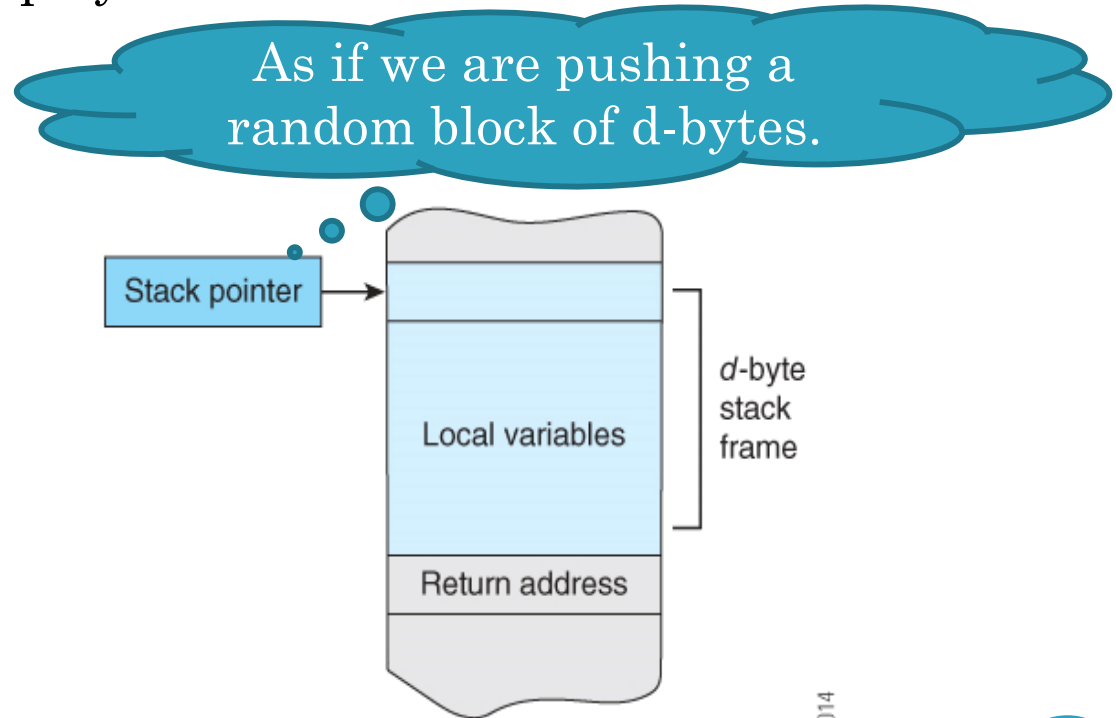
# The Stack Frame and Local Variables

- ❑ 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.

**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
Code memory add ;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

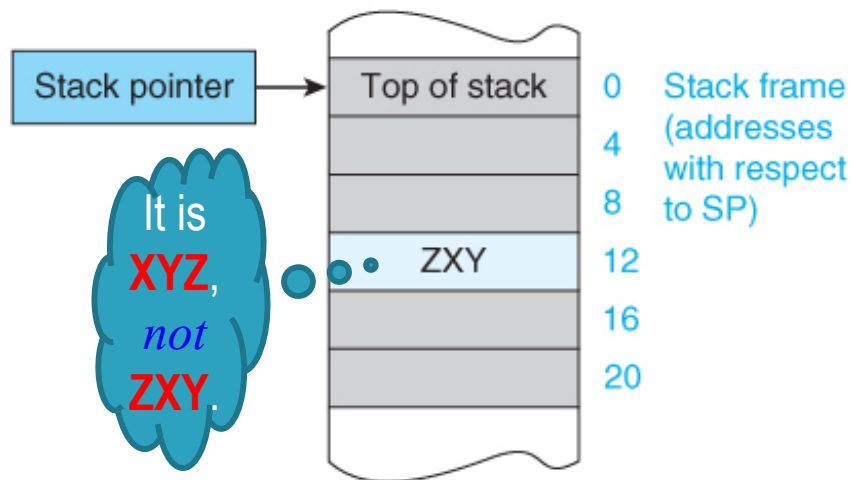
In this example, FP, i.e., R11, is not used.

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.

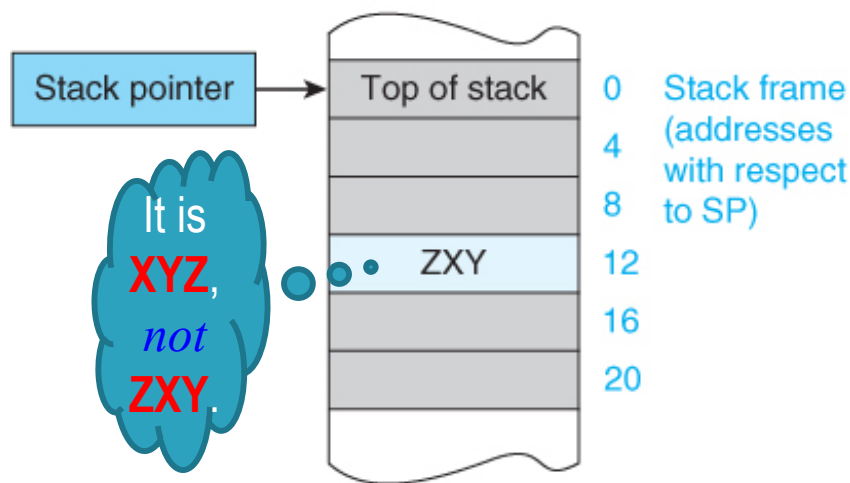
(a) Accessing a variable via the stack pointer

# 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 <sup>ram add.</sup> **[r11, #-8]**, assuming that **r11** is the frame pointer.

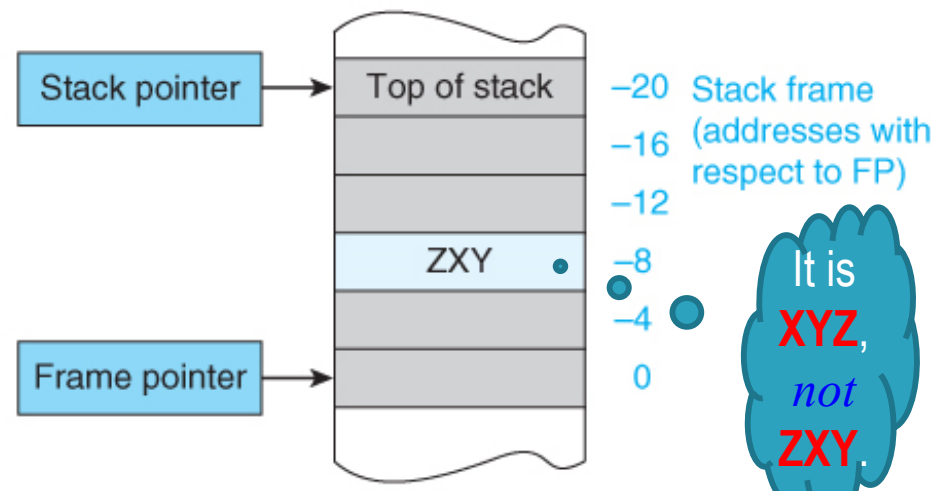
FIGURE 4.4

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



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)

*1 byte: 8 bits.*

*register register.*

- ❑ 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

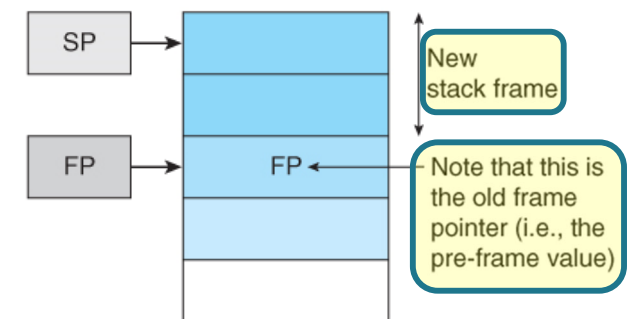
Can be optimized by one instruction:

STR fp, [sp, #-4]!

or

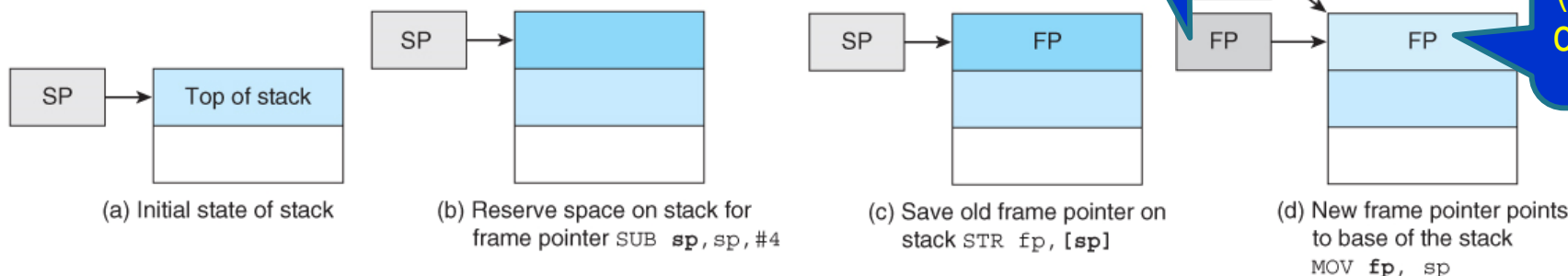
STMFD sp!, {fp}

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)



# The Stack Frame and Local Variables

- ❑ The figure below demonstrates how the stack frame collapses.

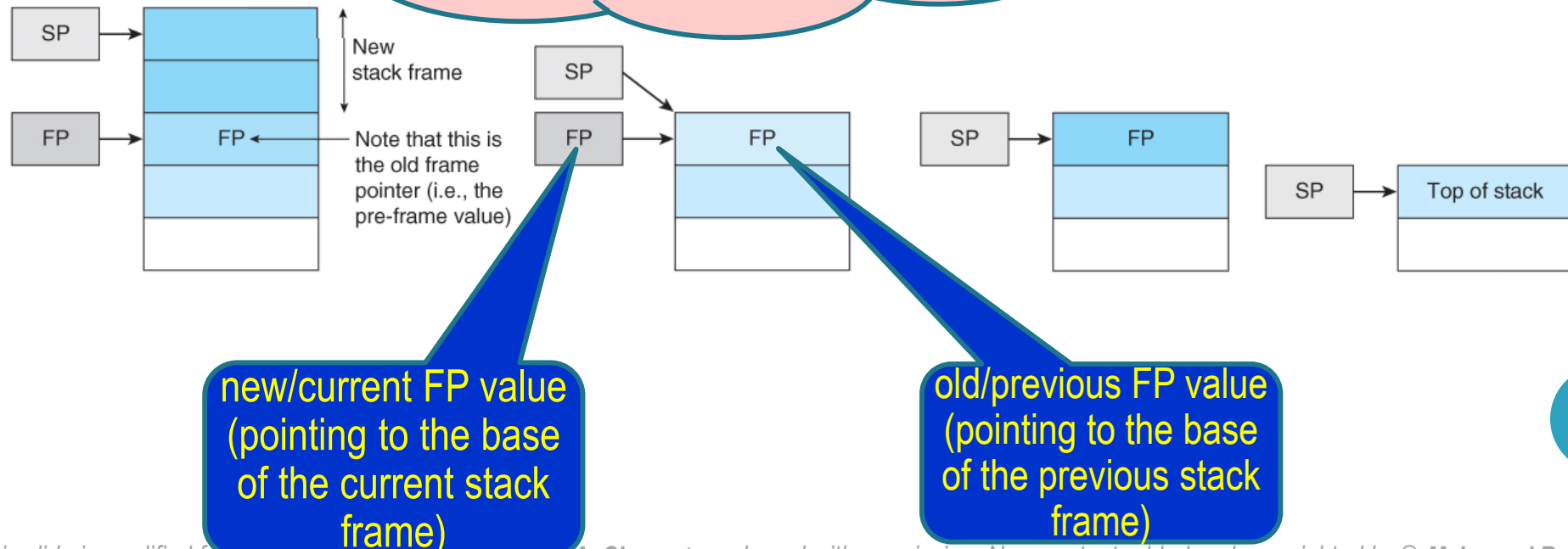
```
MOV sp, fp      ;restore the stack pointer
LDR fp, [sp]     ;restore old frame pointer from the stack
ADD sp, sp, #4   ;move stack pointer down 4 bytes to
                ;restore stack
```

Can be optimized by one instruction:

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

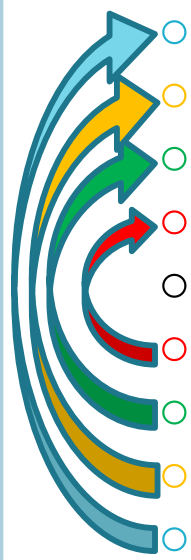
or

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



# 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