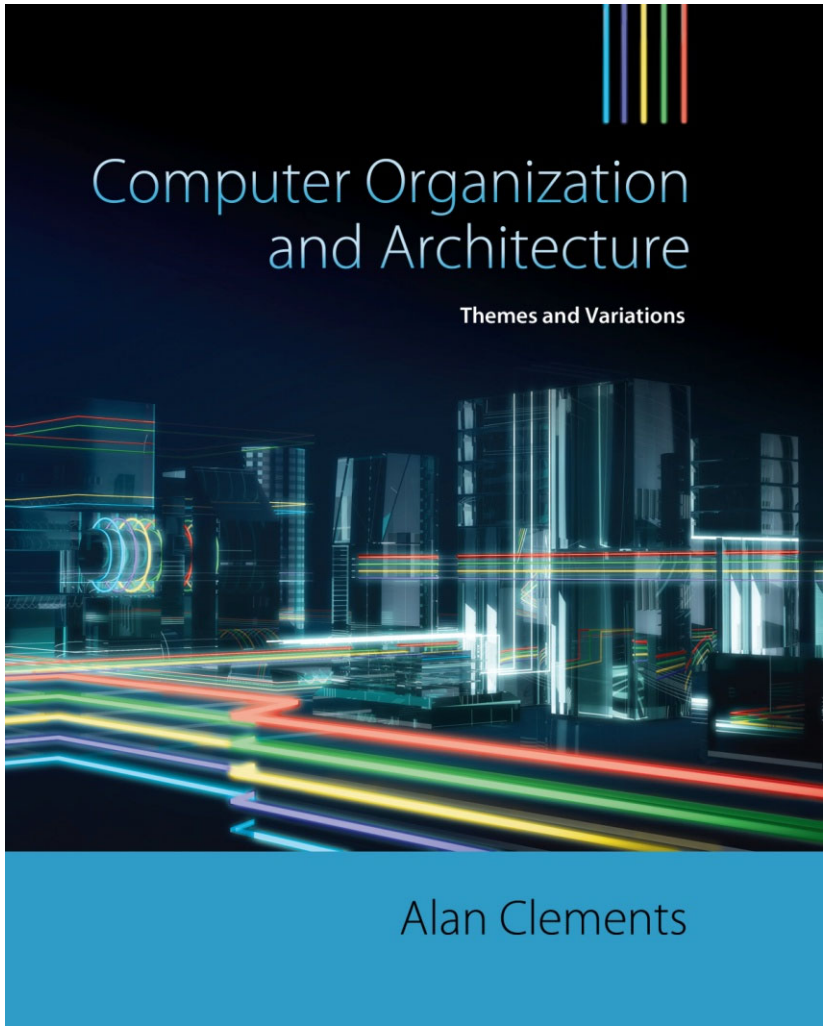
# Part C

# Chapter 3

# Architecture and Organization







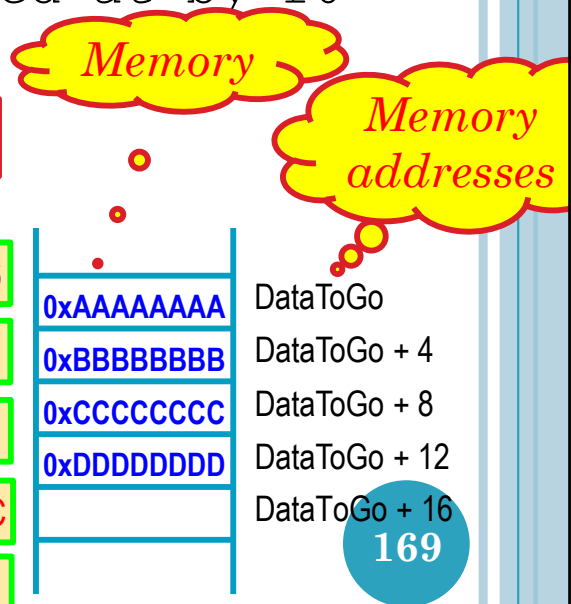

CENGAGE Learning

# Block Move Instructions

- The following conventional ARM code demonstrates how to load four registers from ***consecutive*** memory-locations.

```
1st  ADR r0,DataToGo   ;load r0 with the address of the data area
3rd  LDR r1,[r0]       ;load r1 with the word pointed at by r0
     ADD r0,#4         ;update the pointer
5th  LDR r2,[r0]       ;load r2 with the word pointed at by r0
     ADD r0,#4         ;update the pointer
7th  LDR r3,[r0]       ;load r3 with the word pointed at by r0
     ADD r0,#4         ;update the pointer
9th  LDR r5,[r0]       ;load r5 with the word pointed at by r0
     ADD r0,#4         ;update the pointer
```

| | After 1st instruction | After 3rd instruction | After 5th instruction | After 7th instruction | After 9th instruction |
|---|---|---|---|---|---|
| r0 | DataToGo | DataToGo + 4 | DataToGo + 8 | DataToGo + 12 | DataToGo + 16 |
| r1 | 0x00000000 | 0xAAAAAAAA | 0xAAAAAAAA | 0xAAAAAAAA | 0xAAAAAAAA |
| r2 | 0x00000000 | 0x00000000 | 0xBBBBBBBB | 0xBBBBBBBB | 0xBBBBBBBB |
| r3 | 0x00000000 | 0x00000000 | 0x00000000 | 0xCCCCCCCC | 0xCCCCCCCC |
| r4 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| r5 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0xDDDDDDDD |



# Block Move Instructions

- *A more efficient way* to load four registers from ***consecutive*** memory-locations.

```
ADR r0,DataToGo  ;load r0 with the address of the data area
LDR r1,[r0],#4   ;load r1 with the word pointed at by r0
                 ;and post-update the pointer
LDR r2,[r0],#4   ;load r2 with the word pointed at by r0
                 ;and post-update the pointer
LDR r3,[r0],#4   ;load r3 with the word pointed at by r0
                 ;and post-update the pointer
LDR r5,[r0],#4   ;load r5 with the word pointed at by r0
                 ;and post-update the pointer
```

- ARM has
  - a ***block move from memory to registers*** instruction, **LDM**, and
  - a ***block move from registers to memory*** instruction, **STM**

  that can copy ***group of registers*** from and to memory.
  *This is even more efficient than the above solution*.

- Both block move instructions take a **suffix** to describe ***how*** the data is accessed.

# Block Move Instructions

- Think of block move instructions as if they are stack operations
  - ***STM:*** *to* ***push*** *a group of registers' content to memory*
  - ***LDM:*** *to* ***pop*** *values from memory and load them to a group of registers*

- Let's start by copying the contents of registers r1, r2, r3, and r5, into ***sequential*** memory-locations with

*This is a different example. It is an **STM** and the example in the previous two slides was **LDM**.*

```
ADR r0,DataToGo .
STMIA r0!,{r1-r3,r5} ;note the syntax of this instruction
                     ;the register list is put between
                     ;curly braces and it can have a range
```

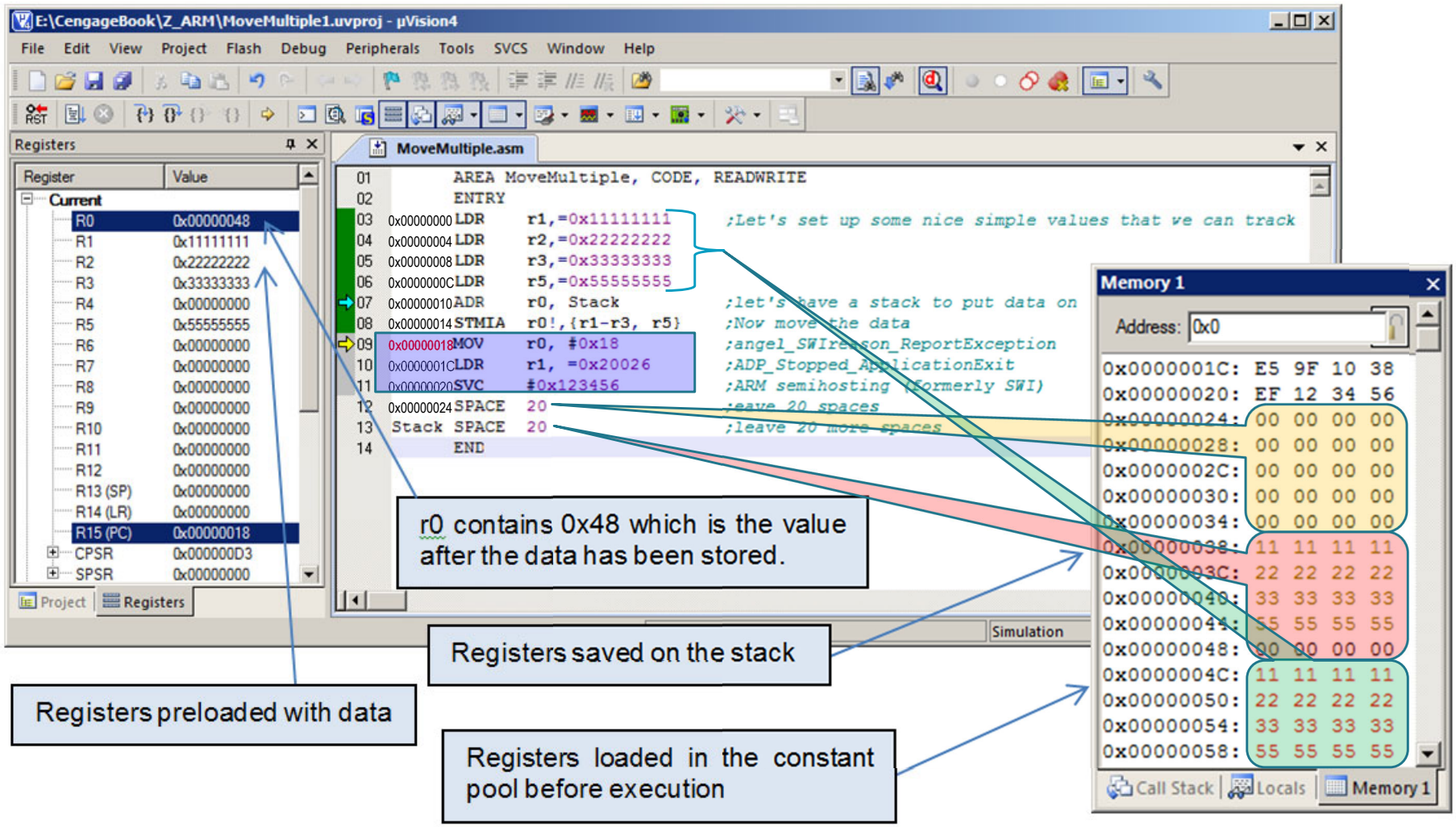
the value that the address in r0 hold.

- This instruction copies registers r1 to r3, and r5, into sequential memory-locations, using r0 as a pointer with *auto-indexing* (indicated by the **!** suffix).

- The suffix **IA** indicates that index register r0 is *incremented after* the transfer, with data transfer in the order of increasing addresses.

- Although ARM's block move instructions have several variations, *ARM always stores the lowest numbered register first at the lowest memory address*, followed by the next lowest numbered register, and so on, regardless of the order in the instruction.
  For example, "`STMIA r0!,{r5,r1-r3}`" and "`STMIA r0!,{r2,r3,r5, r1}`" have the same effect as "`STMIA r0!,{r1-r3,r5}`"

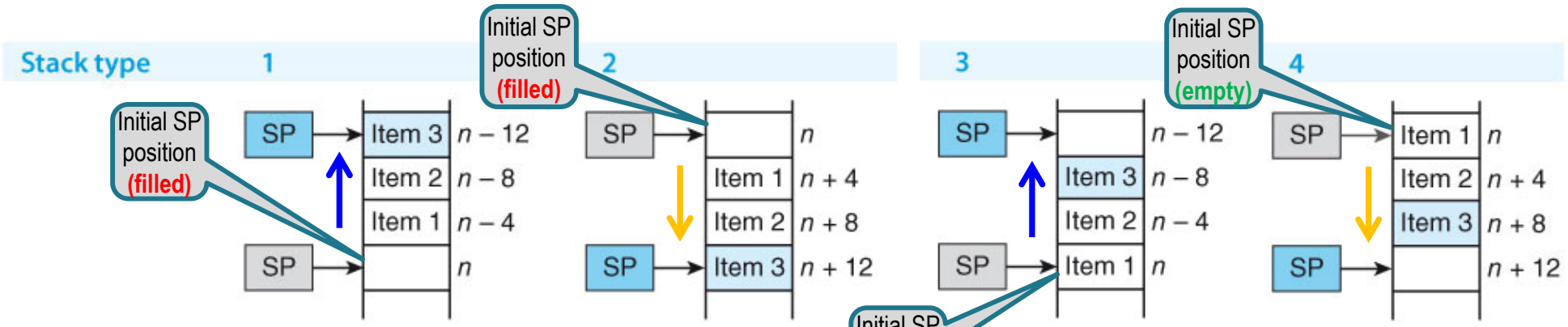
# Block Move Instructions




- In LDM/STM, the access happens in the order of increasing register numbers,
  - the lowest numbered register occupies the lowest memory address and
  - the highest numbered register occupies the highest memory address

# Block Moves and Stack Operations

**TABLE 3.5** Stack Types and the ARM Block Move Instruction Suffixes




| Stack type | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Stack growth | Descending | Ascending | Descending | Ascending |
| Class | Full | Full | Empty | Empty |
| Stack suffix | FD | FA | ED | EA |
| Load suffix | IA (increment after) | DA (decrement after) | IB (increment before) | DB (decrement before) |
| Store suffix | DB (decrement before) | IB (increment before) | DA (decrement after) | IA (increment after) |



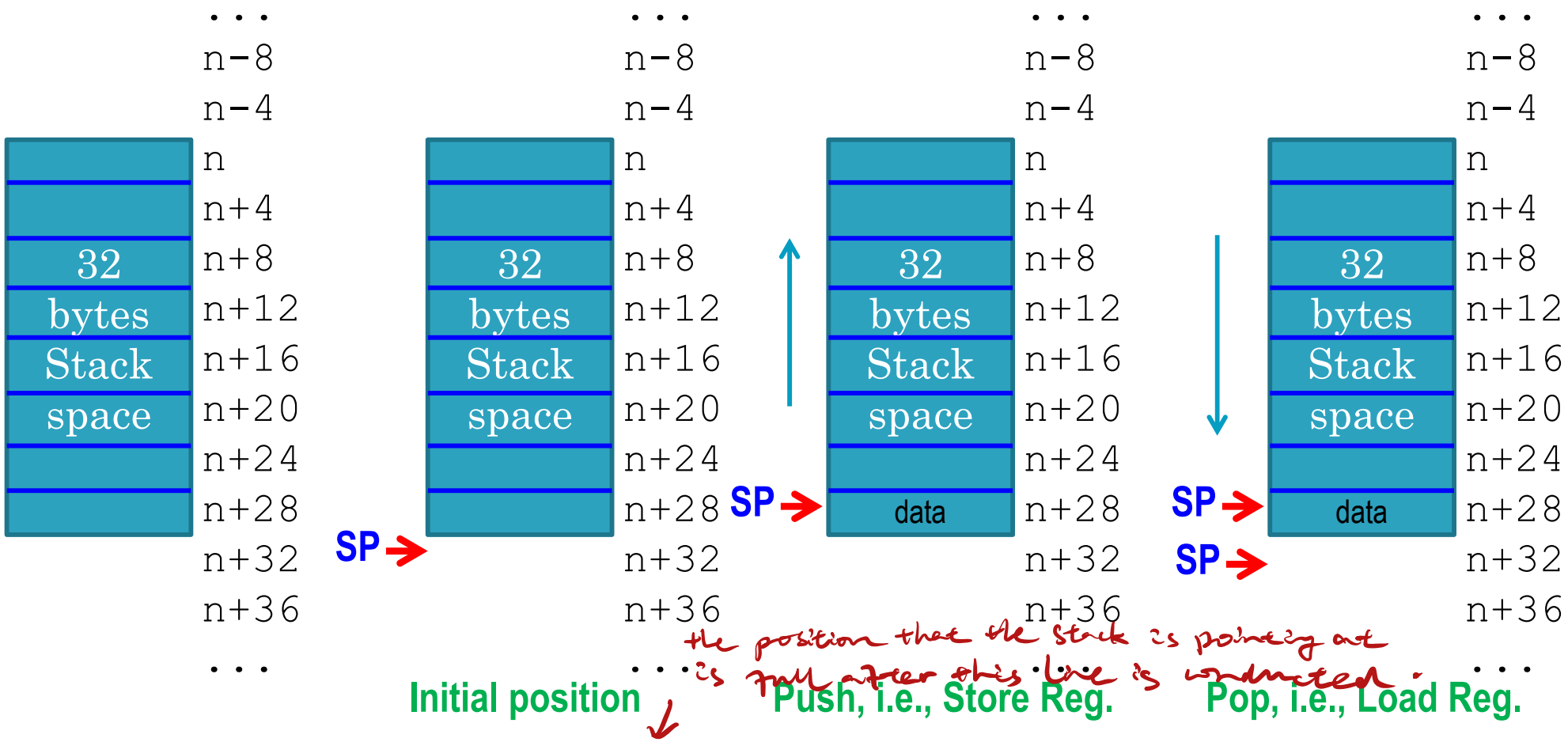
Similar to PUSH, i.e., PUSH from registers to memory

Similar to POP, i.e., POP from memory to these registers

Grows up

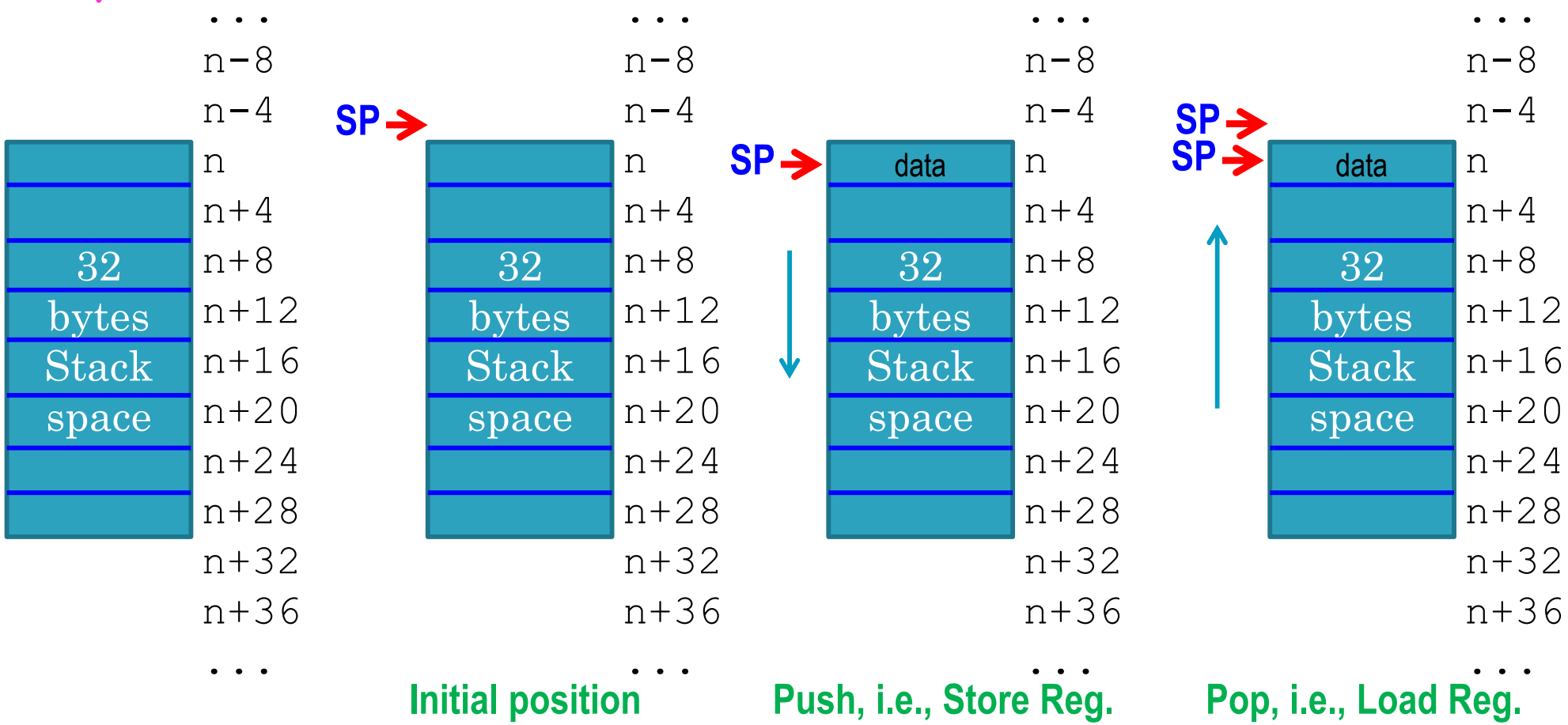
Occupied memory

# Block Moves and Stack Operations

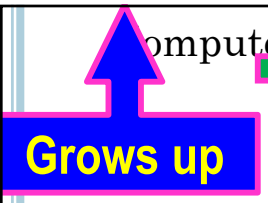



- Stack type: **FD** (i.e., Class=**F**ull and Stack growth=**D**escending)
  (**STMFD** and **LDMFD**)
  - Empty stack ➔ SP points to just after the stack space
  - Pushing on the stack ➔ SP to be **D**ecremented **B**efore (**STMDB**)
  - Popping off the stack ➔ SP to be **I**ncremented **A**fter (**LDMIA**)

# Block Moves and Stack Operations

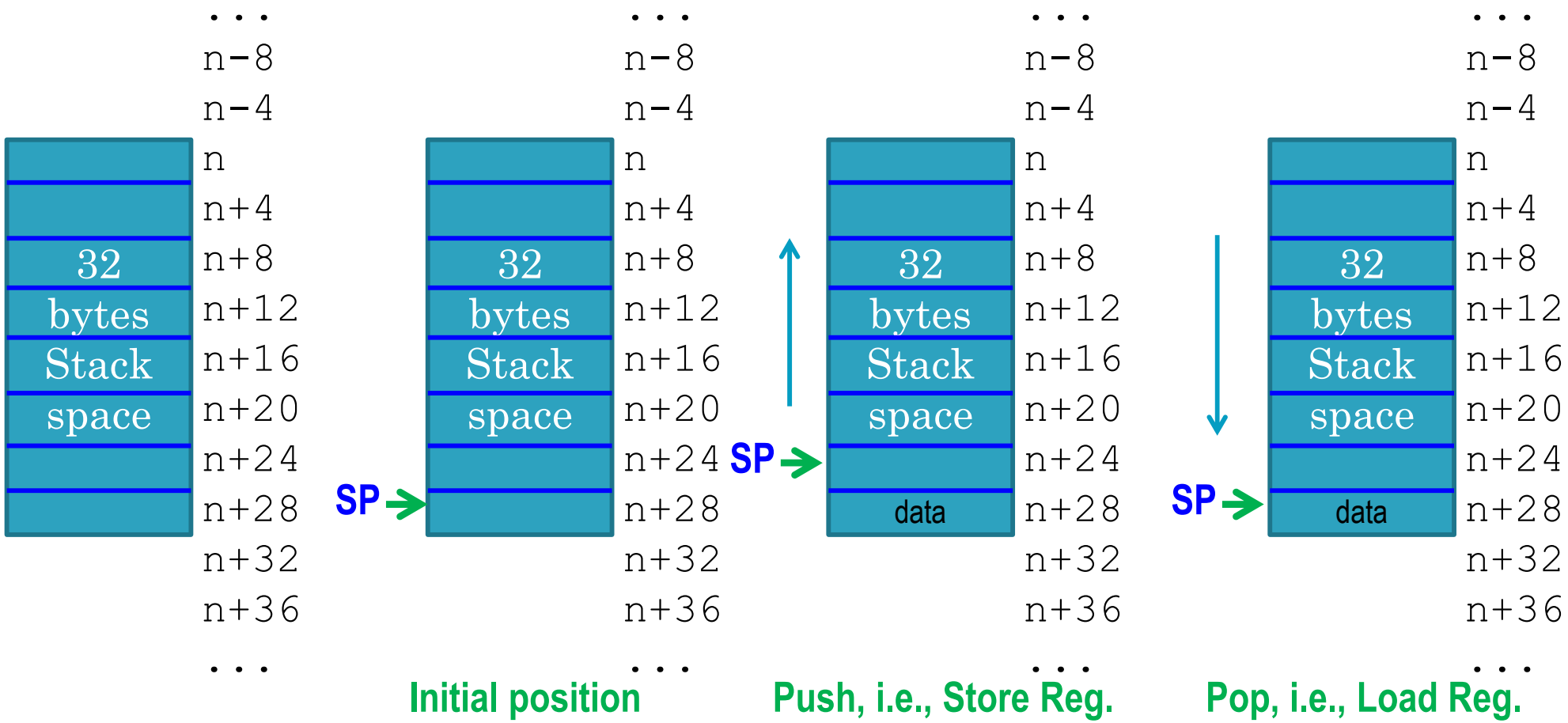



- Stack type: **FA** (i.e., Class=**F**ull and Stack growth=**A**scending)
  (**STMFA** and **LDMFA**)
  - Empty stack ➔ SP points to just before the stack space
  - Pushing on the stack ➔ SP to be **I**ncremented **B**efore (**STMIB**)
  - Popping off the stack ➔ SP to be **D**ecremented **A**fter (**LDMDA**)



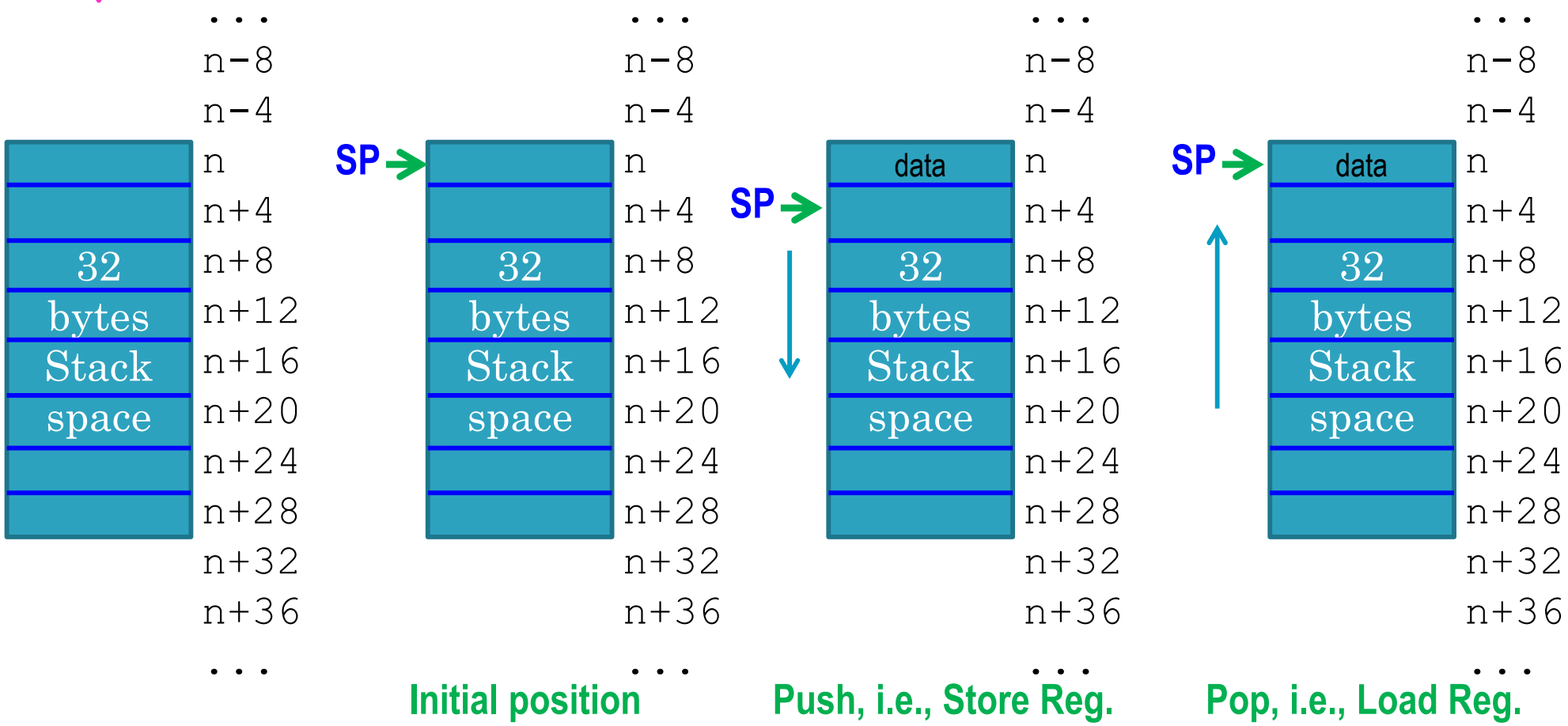


# Block Moves and Stack Operations




- Stack type: **ED** (i.e., Class=**E**mpty and Stack growth=**D**escending)
  (**STMED** and **LDMED**)
  - Empty stack ➔ SP points to the last memory word in the stack
  - Pushing on the stack ➔ SP to be **D**ecremented **A**fter (**STMDA**)
  - Popping off the stack ➔ SP to be **I**ncremented **B**efore (**LDMIB**)

# Block Moves and Stack Operations




- Stack type: **EA** (i.e., Class=**E**mpty and Stack growth=**A**scending)
  (**STMEA** and **LDMEA**)
  - Empty stack ➔ SP points to the first memory word in the stack
  - Pushing on the stack ➔ SP to be **I**ncremented **A**fter (**STMIA**)
  - Popping off the stack ➔ SP to be **D**ecremented **B**efore (**LDMDB**)

# Block Moves and Stack Operations

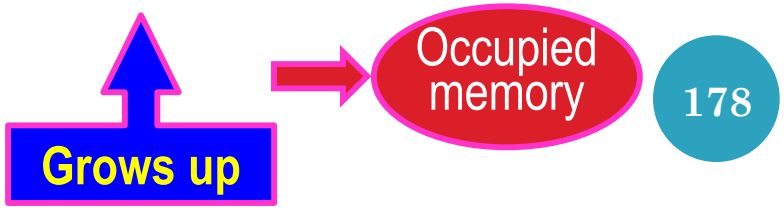
- A stack operation can be described either by
  - ***what*** it does
    - *FD* *Full Descending*
    - *FA* *Full Ascending*
    - *ED* *Empty Descending*
    - *EA* *Empty Ascending*
  - ***how*** it does
    - ***DB*** ***Decremented Before***
    - ***DA*** ***Decremented After***
    - ***I B*** ***Incremented Before***
    - ***I A*** ***Incremented After***

**FD**, **FA**, **ED**, and **EA** are ***pseudo*** notation.

The assembler will translate these ***pseudo*** notation to the **IA**, **DB**, **DA**, and **IB** notation.

For example,

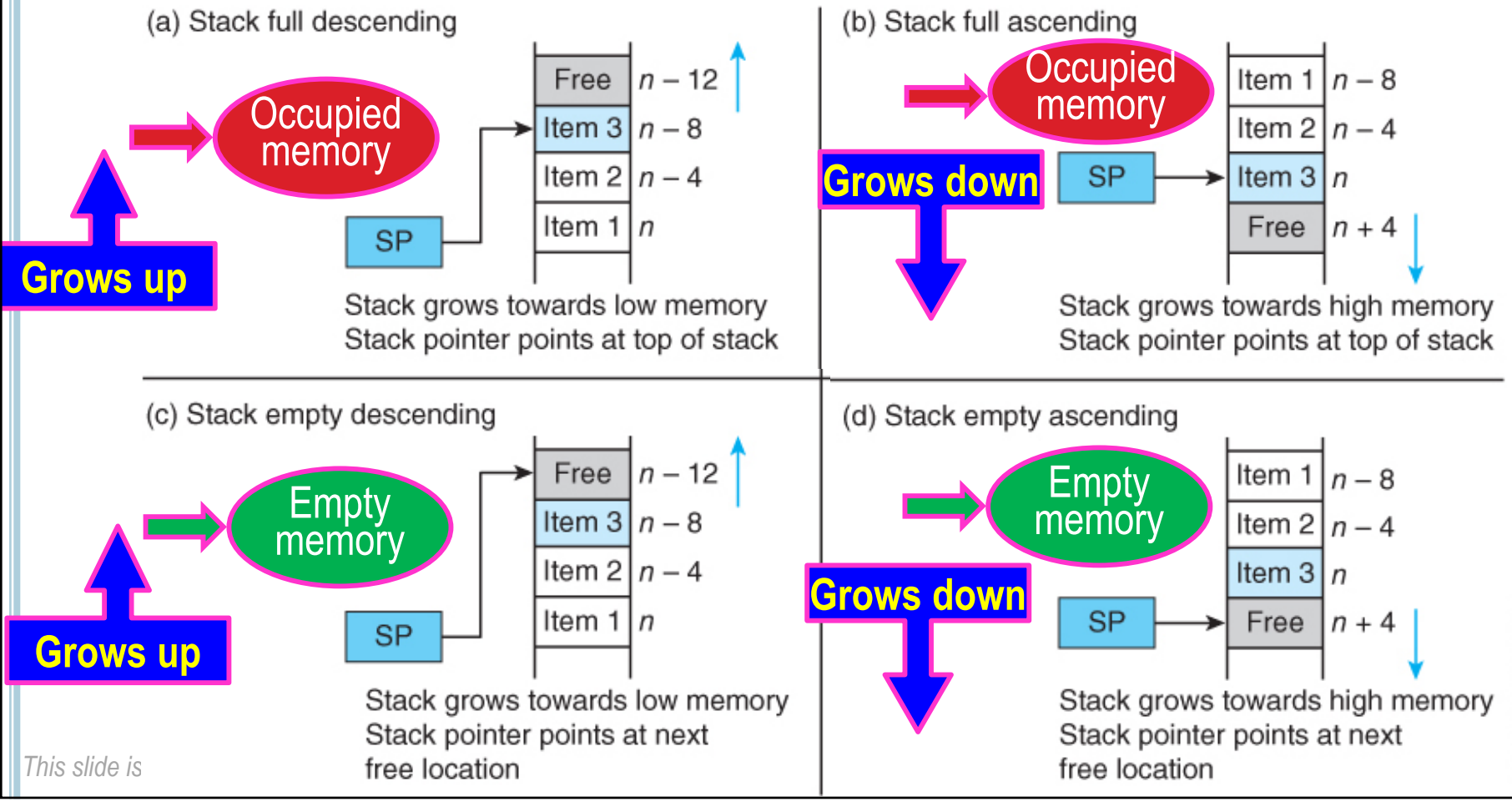
- We can write `STMFD sp!,{r0,r1}` when pushing r0 and r1 onto the stack,
  - Also, can be written as `STMDB sp!,{r0,r1}`
- We can write `LDMFD sp!,{r0,r1}` when popping r0 and r1 off the stack.
  - Also, can be written as `LDMIA sp!,{r0,r1}`



# Block Moves and Stack Operations

- The ARM's literature uses four terms to describe stacks:
  a) FD *full descending* Figure 3.52a
  b) FA *full ascending* Figure 3.52b
  c) ED *empty descending* Figure 3.52c
  d) EA *empty ascending* Figure 3.52d

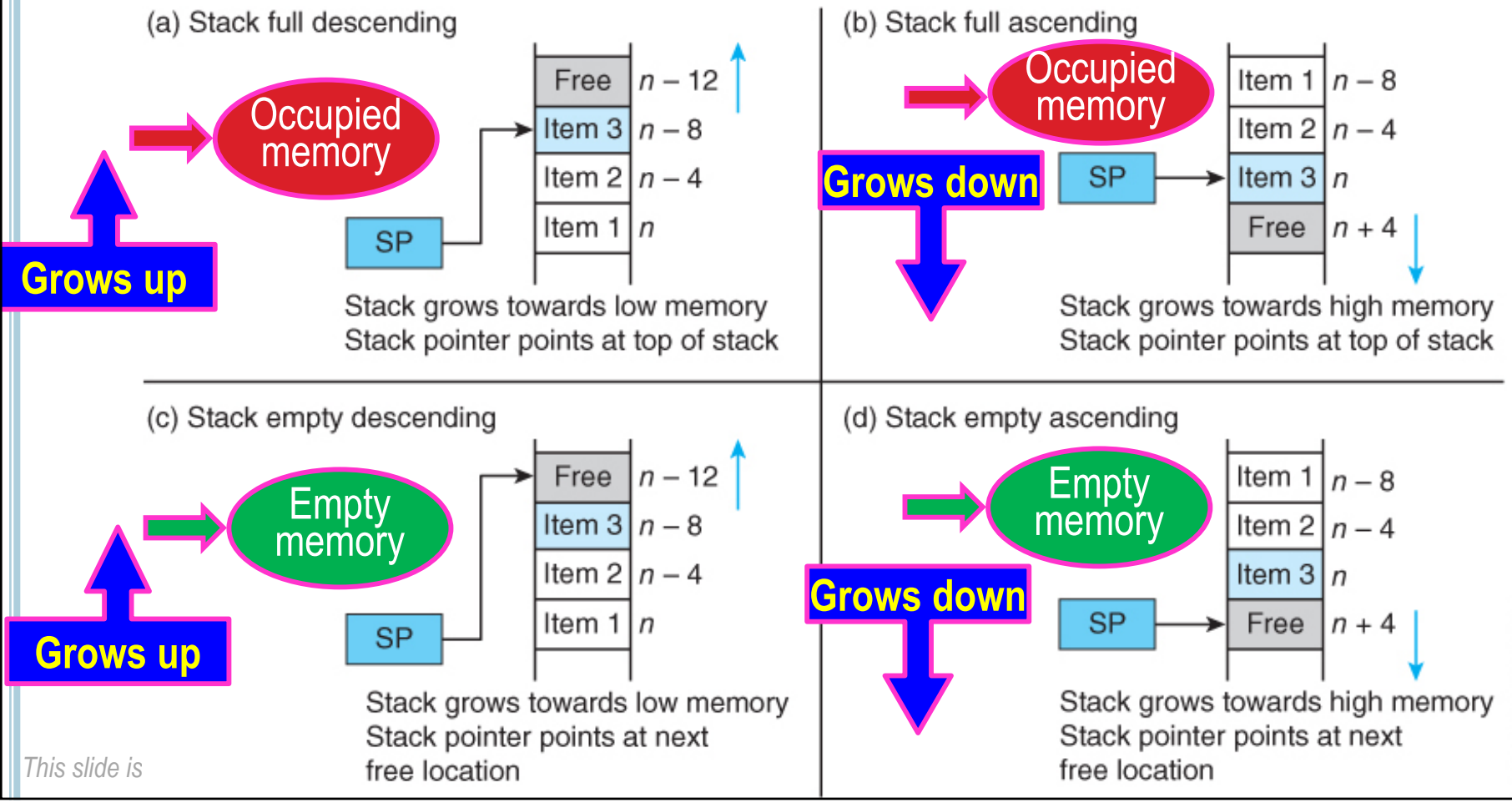
FIGURE 3.59 ARM's four stack modes



# Block Moves and Stack Operations

- A stack is described as *full* if the stack pointer *points to the top element* of the stack.
- If the stack pointer *points to the next free* element in the stack, then the stack is called *empty*.

FIGURE 3.59 ARM's four stack modes



# Block Moves and Stack Operations

- ARM's block move instruction is useful because it supports ***four*** possible ***stack types***.
- The differences among these four types are
  - the *direction* in which the stack grows
    - up (i.e., *descending toward lower addresses*), *or*
    - down (i.e., *ascending toward higher addresses*)

    ARM uses the terms ***ascending*** and ***descending*** to describe the growth of the stack toward ***higher*** or ***lower*** addresses, respectively.

  - whether the *stack pointer points at*
    - the item currently at the *top of the stack* or
    - the *next free item* on the stack.

Bit#23 (Up/down)

Bit#24 (Pre-post)

# Block Move Example

- The block move provides a convenient means of copying data between memory regions. ***This is NOT a stack application.***
- In the next example we copy 256 words (1024 bytes) from Table 1 to Table 2.

```
     ADR     r0,Table1     ;r0 points to source
                           ;(note pseudo-op ADR)
     ADR     r1,Table2     ;r1 points to the destination
     MOV     r2,#32        ;32 blocks of 8 = 256 words to move
Loop LDMFD   r0!,{r3-r10}  ;REPEAT Load 8 registers (r3 to r10)
     STMFD   r1!,{r3-r10}  ;store the registers at
                           ;their destination
     SUBS    r2,r2,#1      ;decrement loop counter
     BNE     Loop          ;UNTIL all 32 blocks of
                           ;8 registers moved
```

Is it right to use LDMFD and STMFD?

- The two block move instructions above allow us to move eight registers (i.e., 32 bytes) at once.

# Block Move Example

- The block move provides a convenient means of copying data between memory regions. ***This is NOT a stack application.***
- In the next example we copy 256 words (1024 bytes) from Table 1 to Table 2.

```
     ADR     r0,Table1     ;r0 points to source
                           ;(note pseudo-op ADR)
     ADR     r1,Table2     ;r1 points to the destination
     MOV     r2,#32        ;32 blocks of 8 = 256 words to move
Loop LDMIA   r0!,{r3-r10}  ;REPEAT Load 8 registers (r3 to r10)
     STMIA   r1!,{r3-r10}  ;store the registers at
                           ;their destination
     SUBS    r2,r2,#1      ;decrement loop counter
     BNE     Loop          ;UNTIL all 32 blocks of
                           ;8 registers moved
```

LDMIA and STMIA, not LDRFD and STRFD
Not correct in the book page 220

- The two block move instructions above allow us to move eight registers (i.e., 32 bytes) at once.