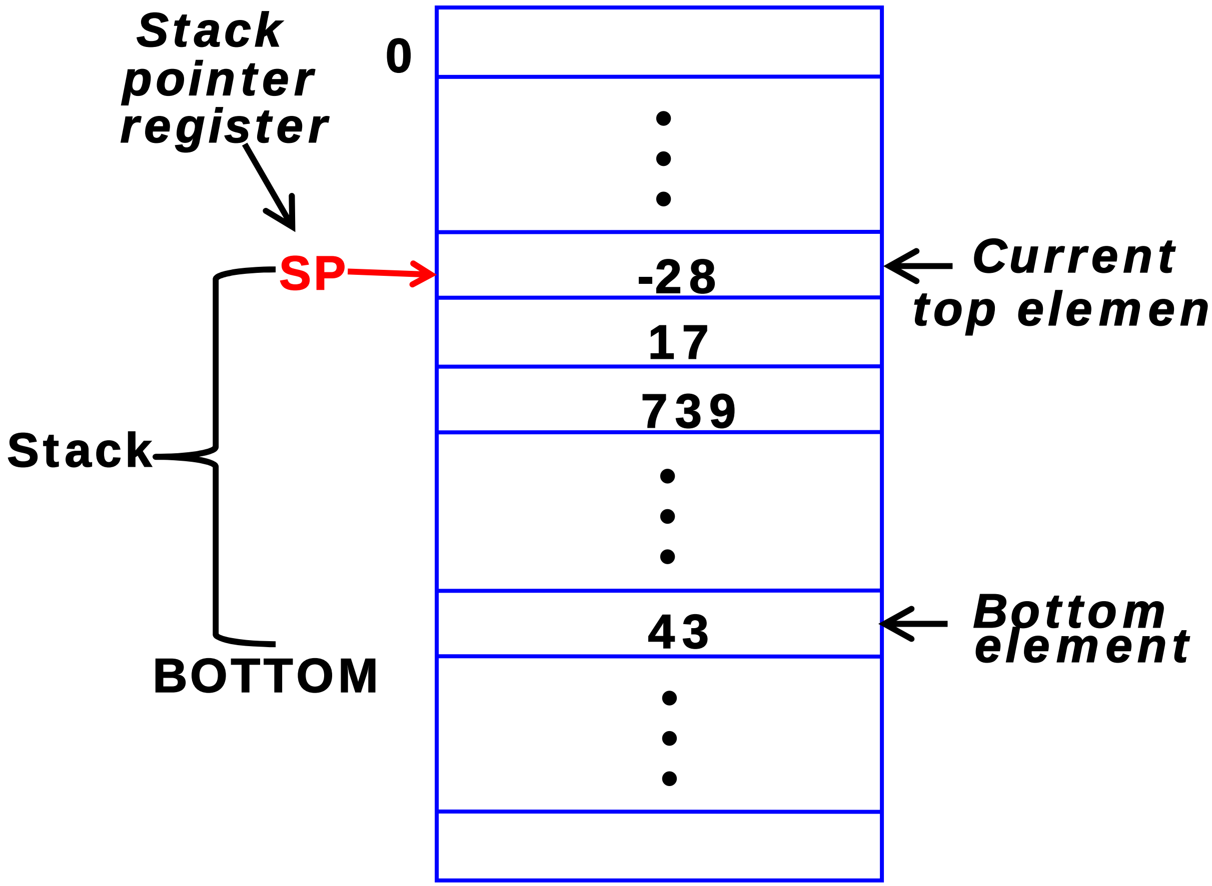
14a Stacks
CSC 230

Department of Computer Science
University of Victoria

Stallings chapters 12,13 (skip Intel portions)
M&H: chapter 4 translated from ARC
ARM Manual

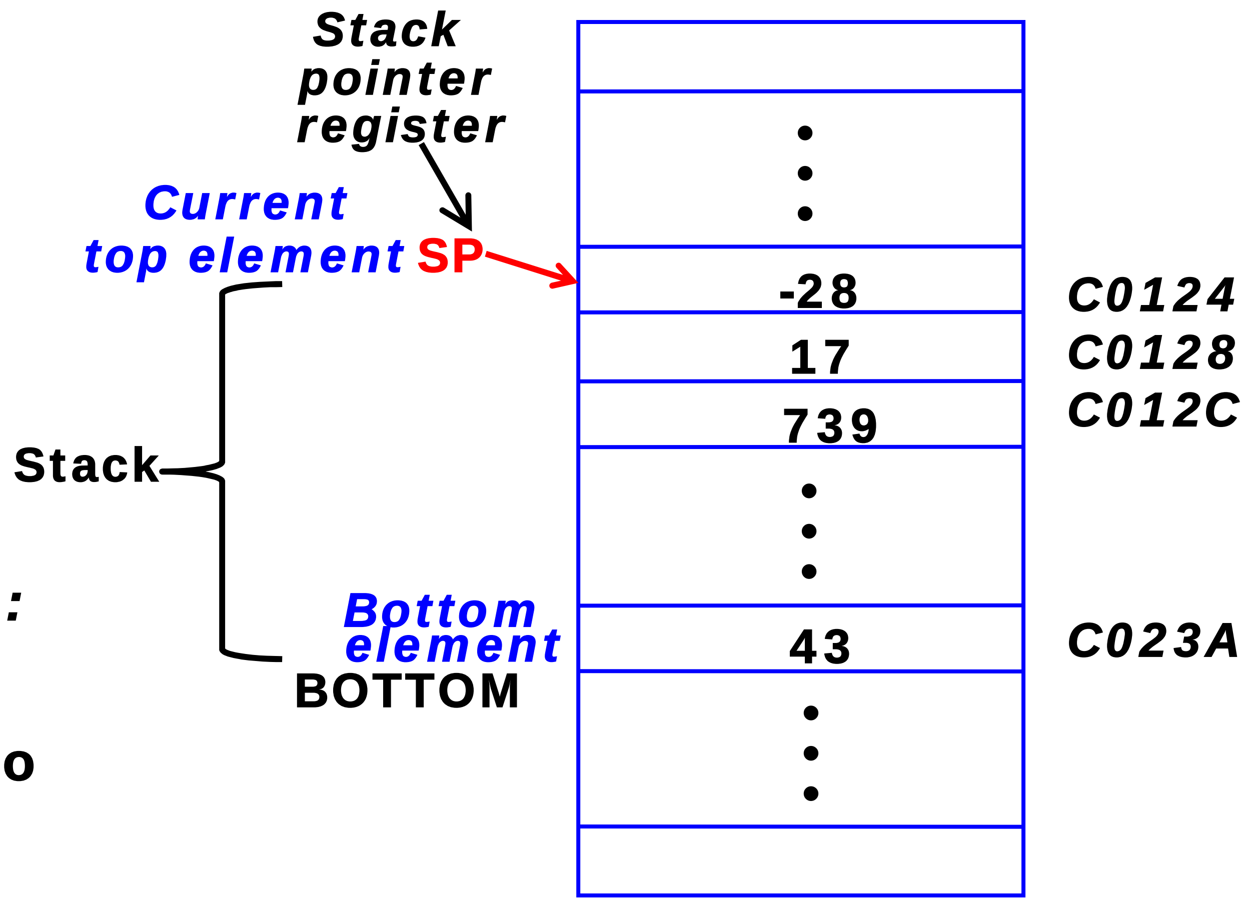
Remember Stacks?

- ❑ List of contiguous data elements with access only from one end
- ❑ LIFO data structure: Last In First Out
- ➔ **PUSH:** place a new element on the top of the stack
- ➔ **POP:** remove element from the top of the stack



A stack of words in memory

Stacks and Memory

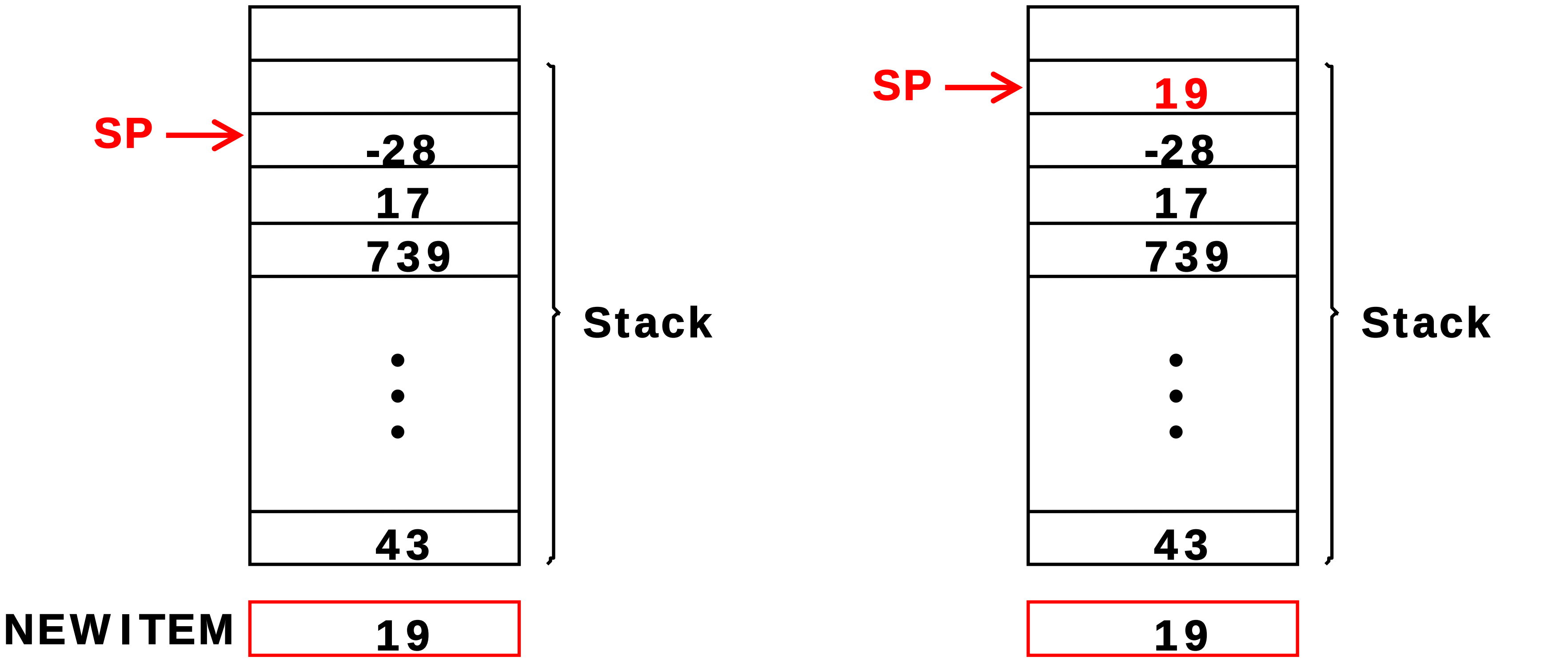


- ***Easiest and most common:***

  Stack grows from bottom to top thus top element has "lower" address than bottom

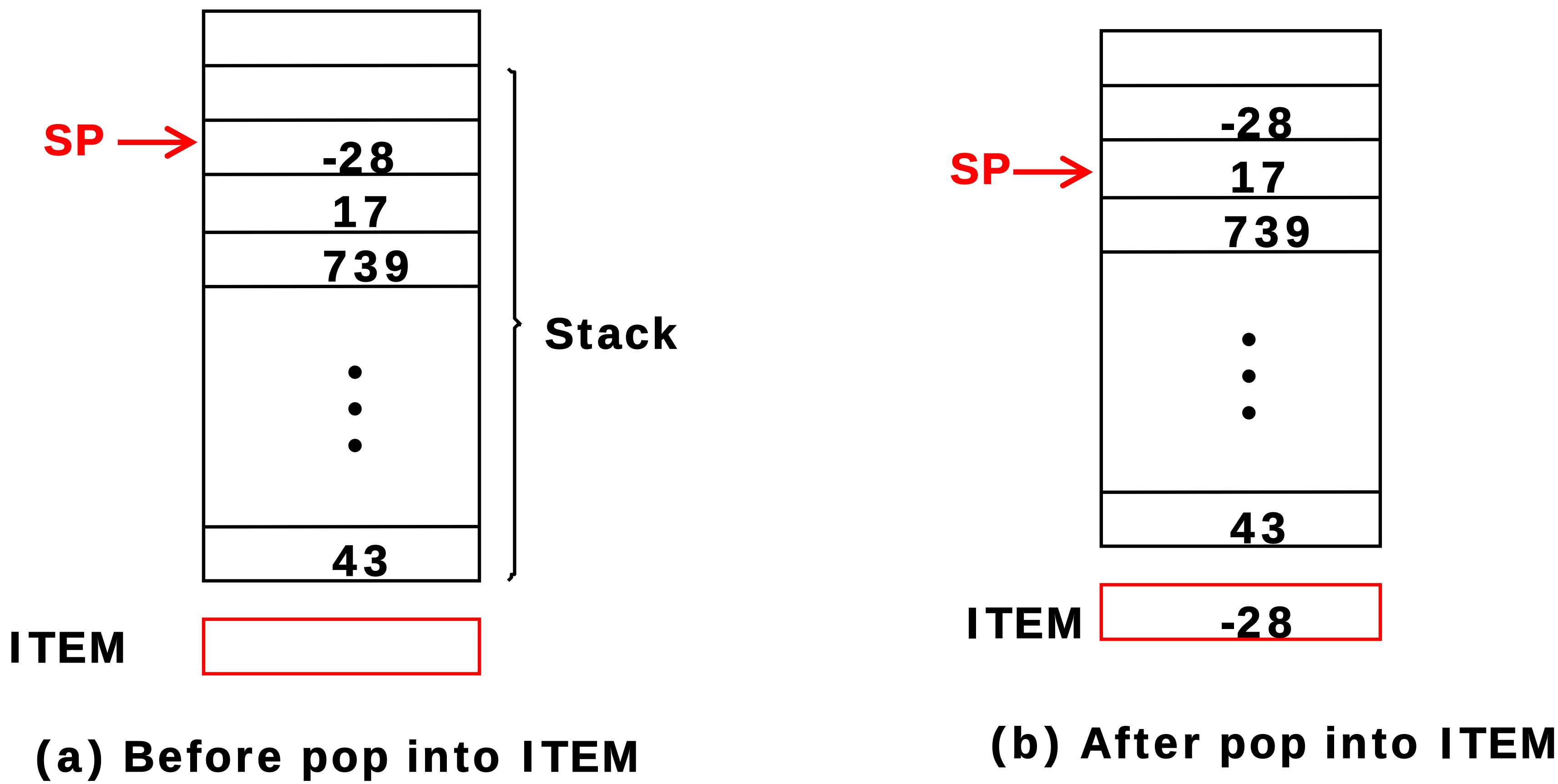
A stack of words in memory

Effect of stack operations: PUSH



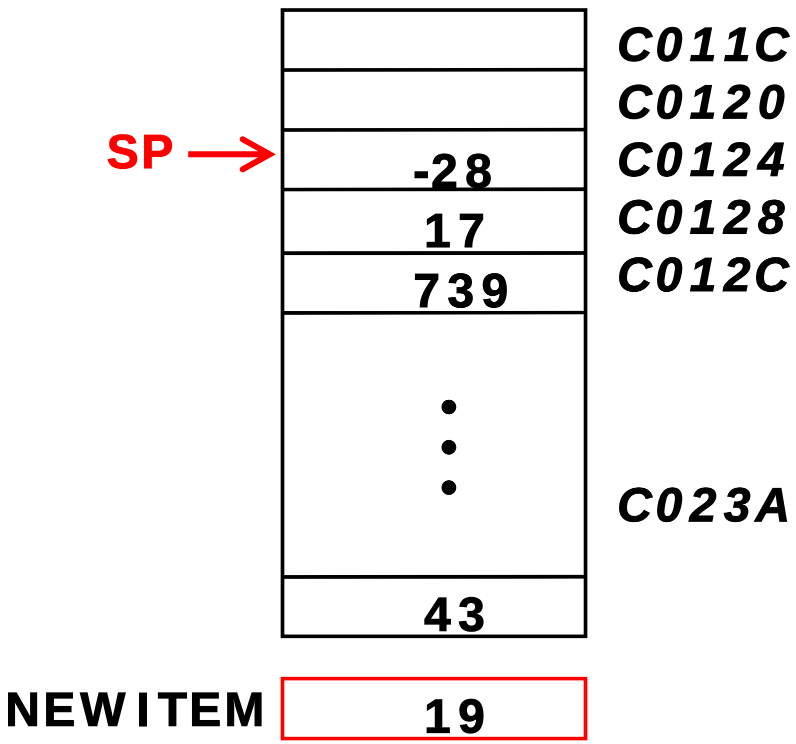
(a) Before push of NEWITEM (b) After push from NEWITEM

Effect of stack operations: POP



In general for a PUSH on STACK

1. SP contains the address of the top of the stack
2. Decrement SP to point to the slot towards lower memory
3. Copy to the stack in the location pointed to by the updated SP




```
Subtract #4 from SP

Store NEWITEM, [SP]
```

Assume R2=NEWITEM

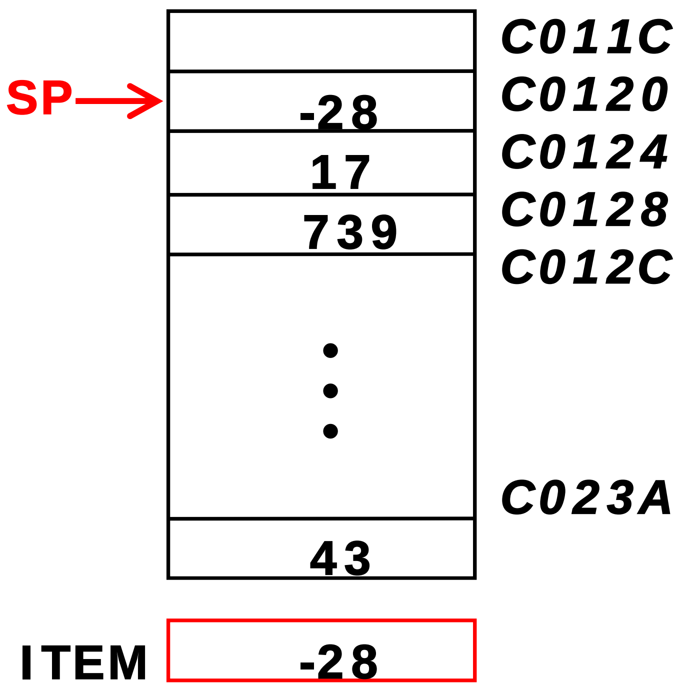
```
SUB    SP,SP,#4
STR    R2, [SP]
```

Better:

```
STR    R2, [SP,#-4]!
```

In general for a POP from STACK

1. SP contains the address of the top of the stack
2. Copy from the location pointed to by SP into a register
3. Increment SP to point to the slot towards higher memory



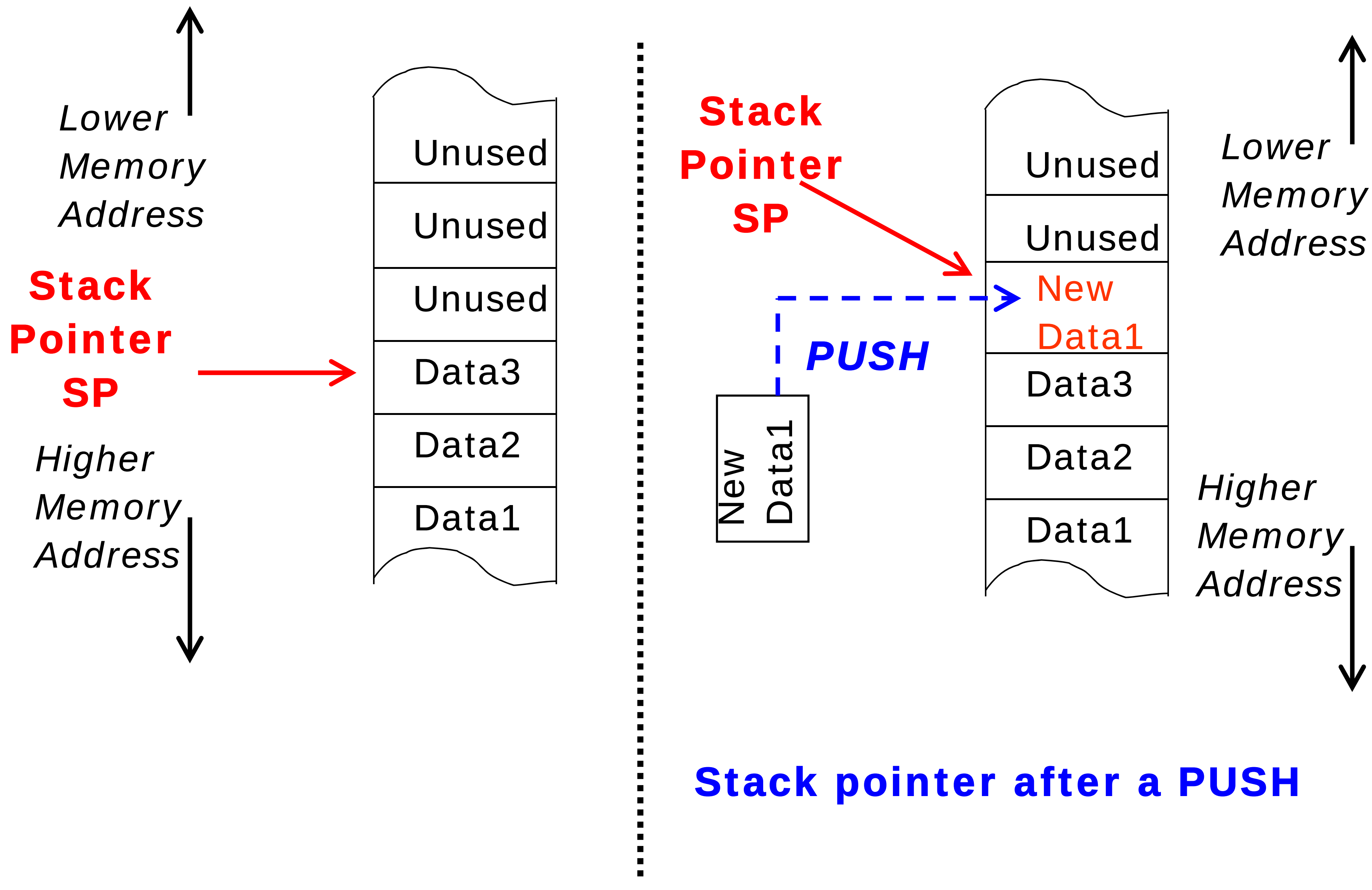

```
Load NEWITEM, [SP]

Add #4 to SP
```

```
Assume NEWITEM→ R2
LDR    R2, [SP]
ADD    SP,SP,#4
```

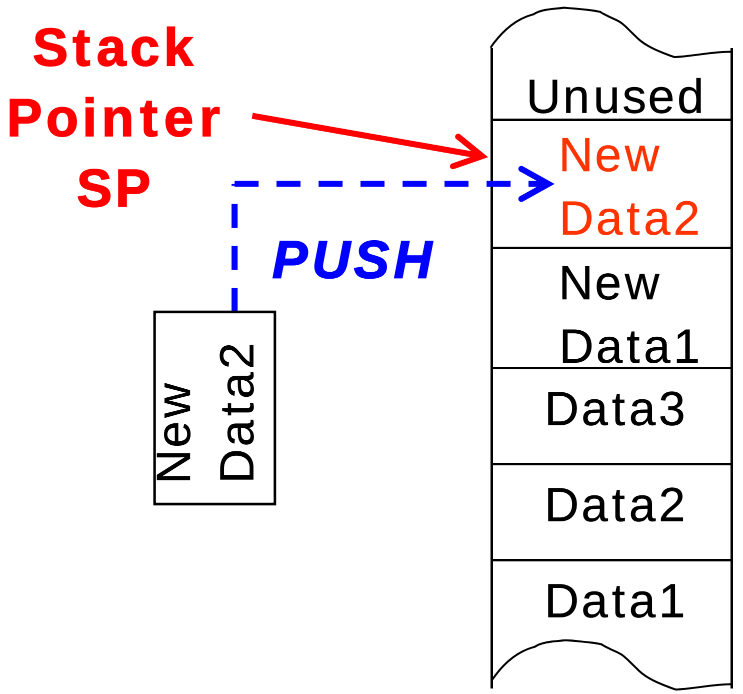
```
Better:
LDR    R2, [SP],#4
```

General (visual) summary 1

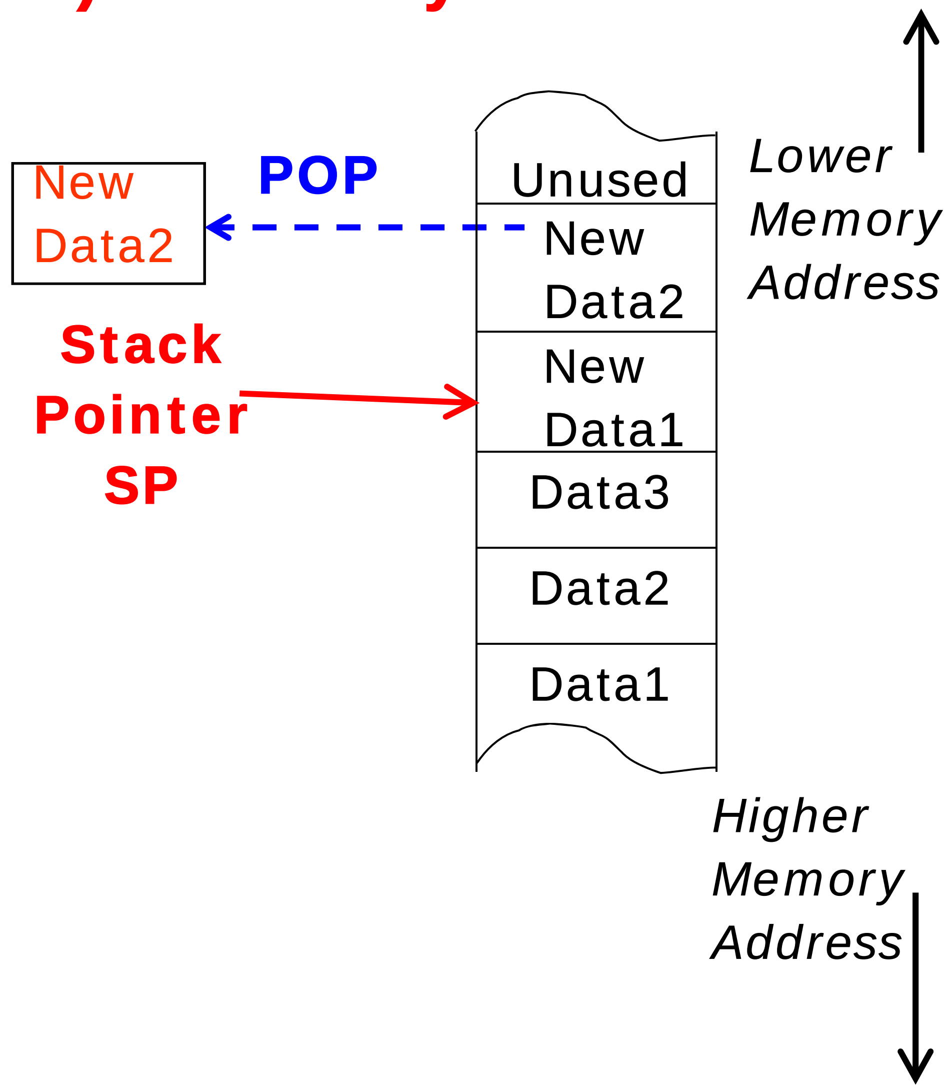


Stack pointer after a PUSH

General (visual) summary 2



Stack pointer after a second PUSH



Stack pointer after a POP

Load / Store (Multiple) Operands with stack

(See ARM Manual as well)

Stack can be programmed to be:

1. **Ascending:** stack grows upwards, starting from a low address towards a higher address

2. **Descending:** stack grows downwards, starting from a high address towards a lower address

---

A. **Empty:** stack pointer points to the next free space

B. **Full:** stack pointer points to the last accessed item

NOTE: system stack is full descending

Name	Stack	General
Pre-increment Load	LDMED	LDMIB
Post-increment Load	LDMFD	LDMIA
Pre-decrement Load	LDMEA	LDMDB
Post-decrement Load	LDMFA	LDMDA
Pre-increment Store	STMFA	STMIB
Post-increment Store	STMEA	STMIA
Pre-decrement Store	STMFD	STMDB
Post-decrement Store	STMED	STMDA

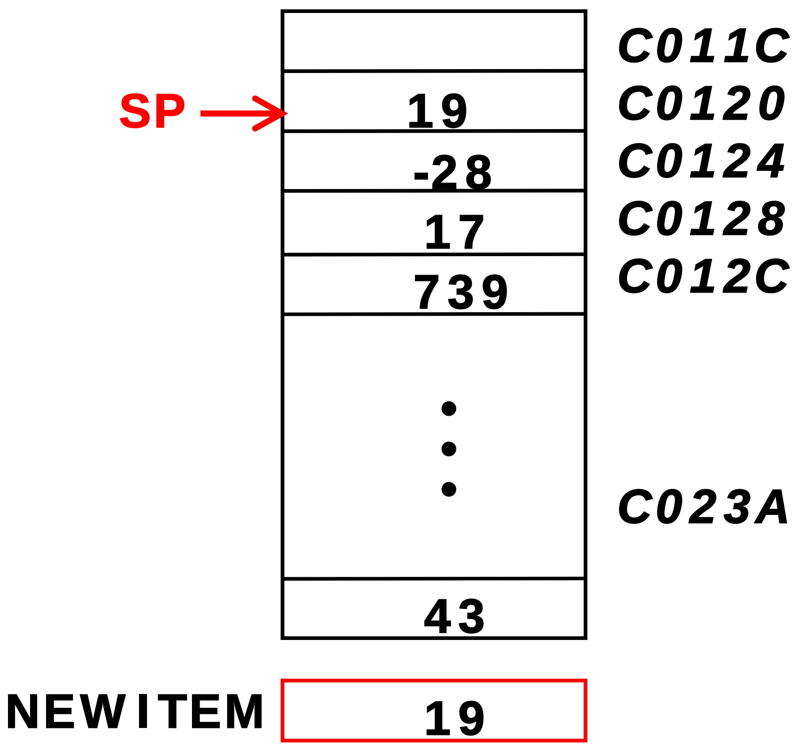
always use these ones!

More examples

```
STMFA   r13!, {r1-r3}        @Push to a full ascending stack
LDMFA   r13!, {r1-r3}        @Pop from a full ascending stack
STMFD   r13!, {r0-r5}        @Push to a full descending stack
LDMFD   r13!, {r0-r5}        @Pop from a full descending stack
STMEA   r13!, {r0,r4,r5}     @Push to empty ascending stack
LDMEA   r13!, {r0,r4,r5}     @Pop from empty ascending stack
STMED   r13!, {r0-r2,r6}     @Push to empty descending stack
LDMED   r13!, {r0-r2,r6}     @Pop from empty descending s.
```

In general for a PUSH on STACK with STM

1. SP contains the address of the top of the stack
2. Decrement SP to point to the slot towards lower memory
3. Copy to the stack in the location pointed to by the updated SP




```
Subtract #4 from SP

Store NEWITEM, [SP]
```

```
Assume R2=NEWITEM
SUB    SP,SP,#4
STR    R2, [SP]
```

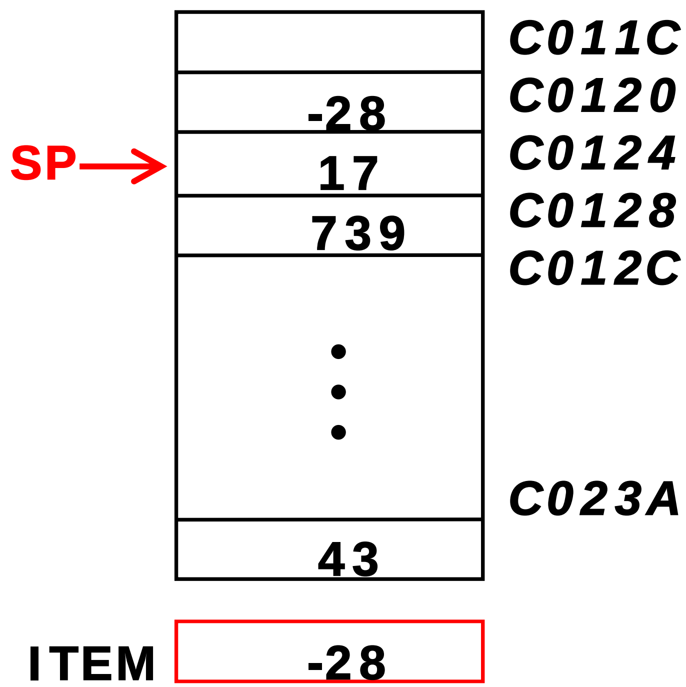
```
Better:
STR    R2, [SP,#-4]!
```

Equivalent, easier and more efficient for two or more registers

```
Much Better:
STMFD SP!, {R2}
```

In general for a POP from STACK with LDM

1. SP contains the address of the top of the stack
2. Copy from the location pointed to by SP into a register
3. Increment SP to point to the slot towards higher memory




```
Load NEWITEM, [SP]

Add #4 to SP
```

Assume NEWITEM → R2

```
LDR    R2, [SP]
ADD    SP,SP,#4
```

Better:

```
LDR    R2, [SP],#4
```

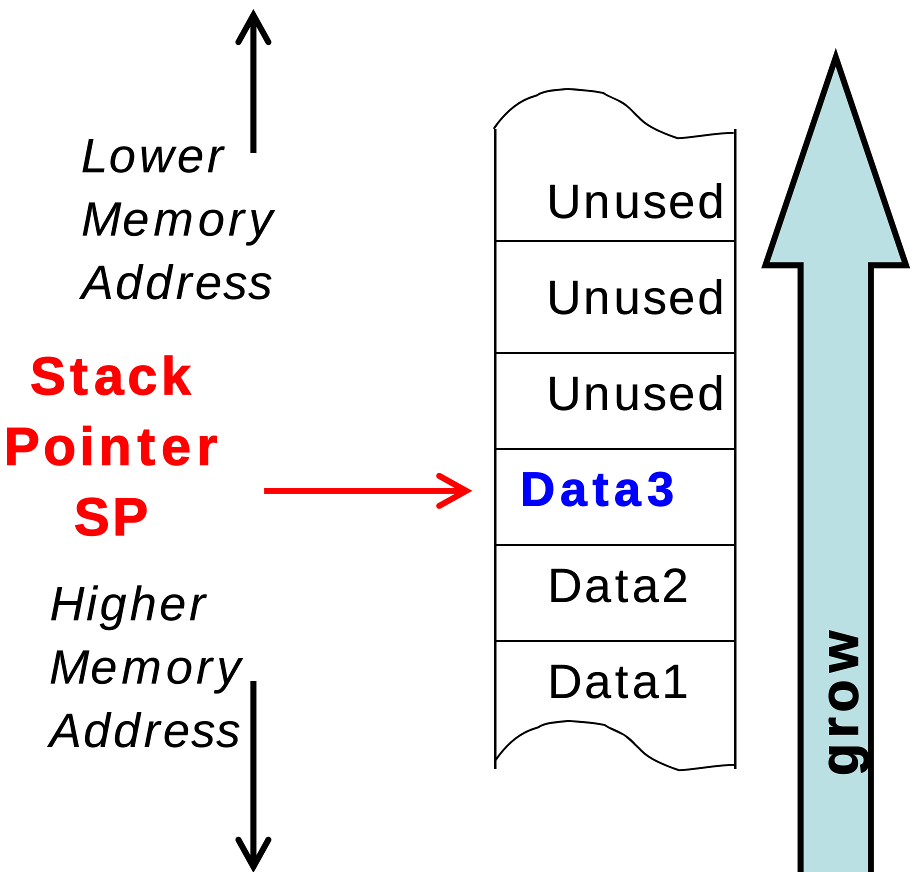
Much Better:

```
LDMFD SP!, {R2}
```

Equivalent, easier and more efficient for two or more registers

STMFD r13!, {r0-r5} @Push to a full descending stack

LDMFD sp!, {r0-r5} @Pop from a full descending stack



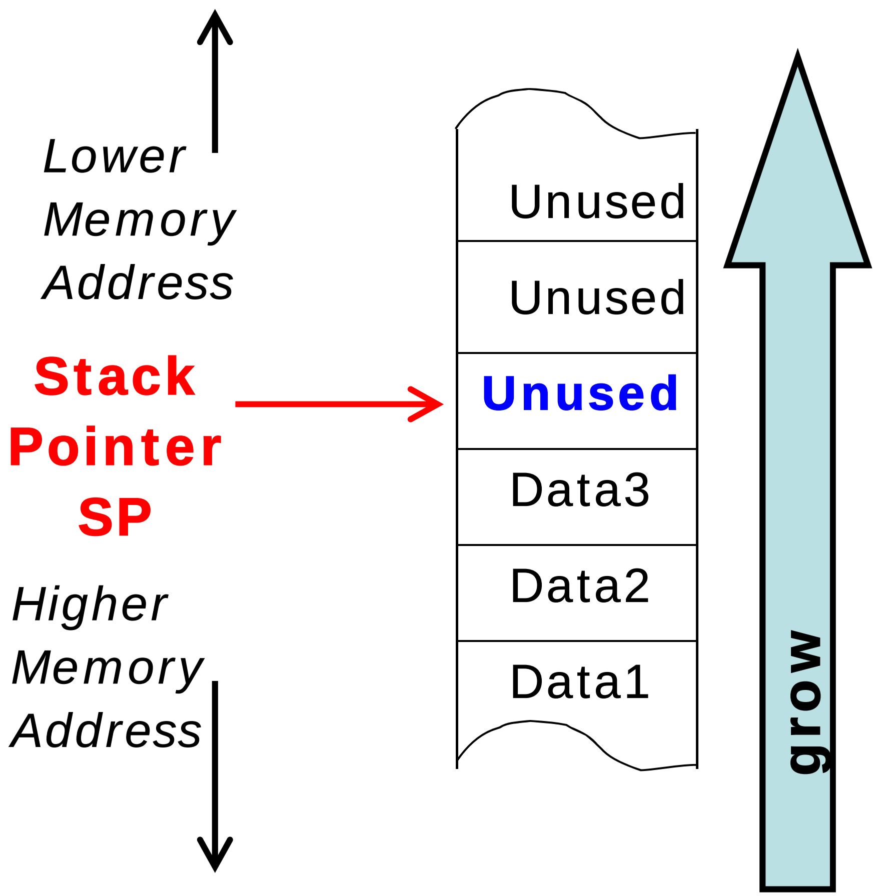
Descending ➔ stack grows towards a lower memory address

Full ➔ stack pointer points to (has address of) last element entered

```
STMED  r13!, {r0-r5}@Push to empty descending stack

LDMED  r13!, {r0-r5}  @Pop from empty descending stack
```

READ ONLY



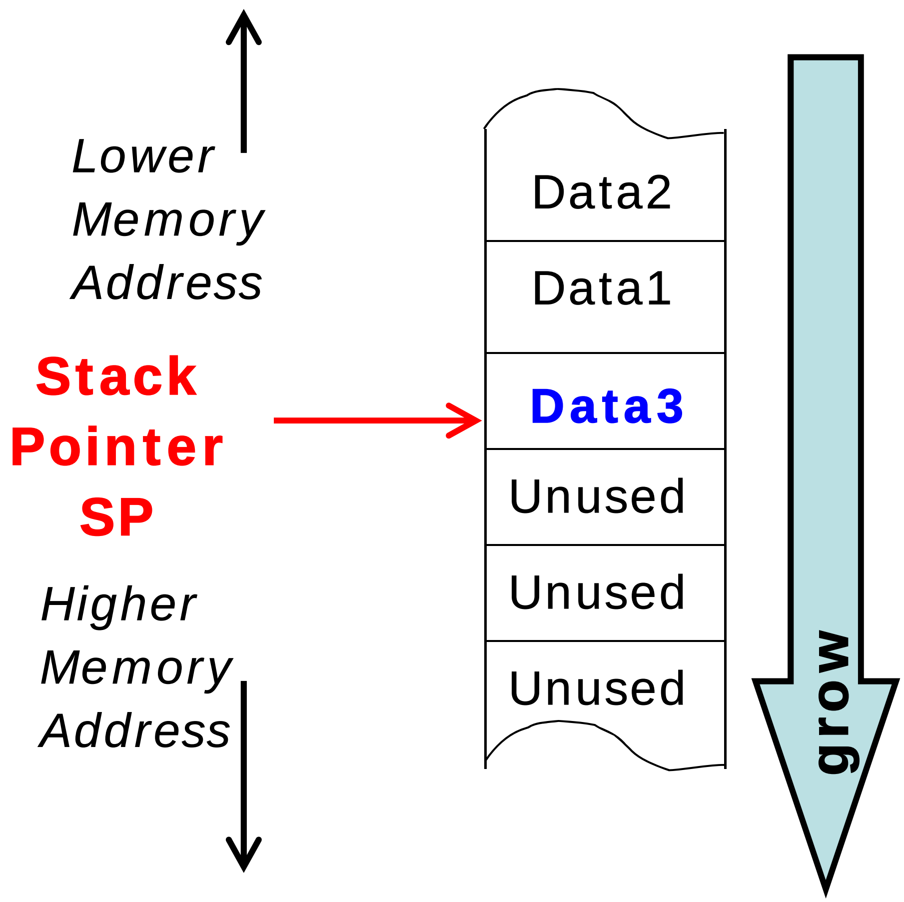
Descending ➔ **stack grows towards a lower memory address**

Empty ➔ **stack pointer points to (has address of) next empty element to be used**

STMFA r13!, {r0-r5} @Push onto a full ascending stack

LDMFA r13!, {r0-r5} @Pop from a full ascending stack

READ ONLY



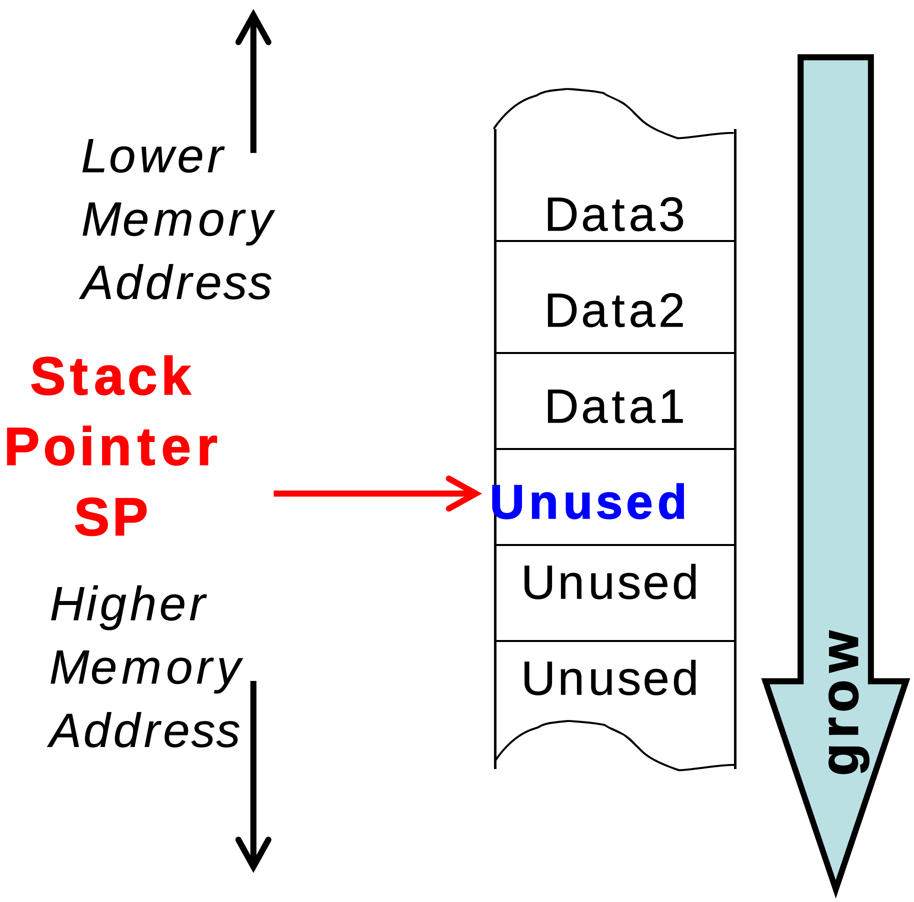
Ascending ➔ stack grows towards a higher memory address

Full ➔ stack pointer points to (has address of) last element entered

STMEA r13!, {r0-r5} @Push to empty ascending stack

LDMEA r13!, {r0-r5} @Pop from empty ascending stack

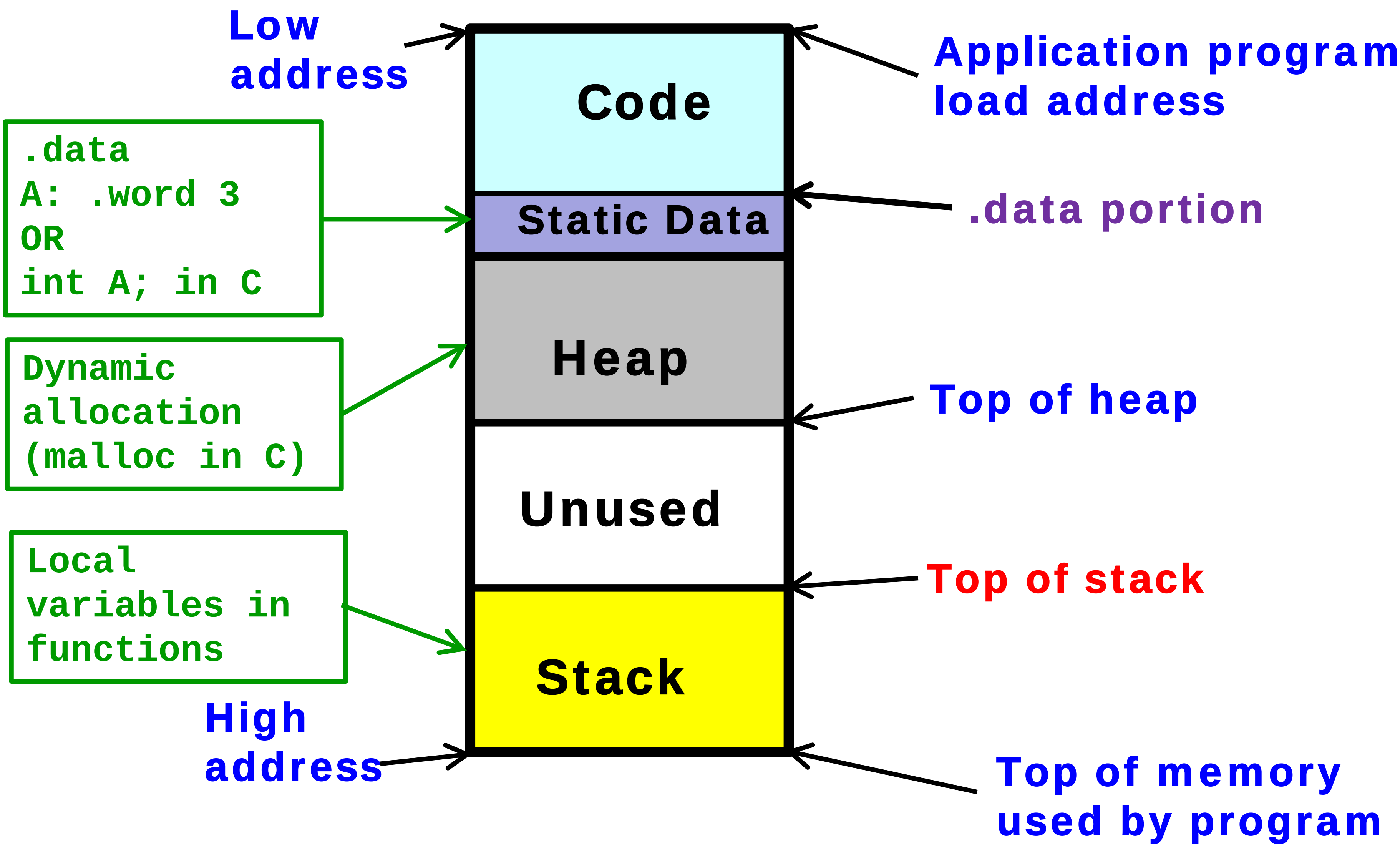
READ ONLY



Ascending ➔ **stack grows towards a higher memory address**

Empty ➔ **stack pointer points to (has address of) next empty element to be used**

Where is storage allocated within the program space?



What can one put on the STACK?

- Parameters passed to a subroutine
- Return addresses from subroutines (link register)
- Local variables
- Registers saved locally

Let's bring Together: Stacks, Subroutines, LDM and STM instructions

A. At the entry point in a function we need to copy the values in registers which will be used locally to some place in memory

B. At the exit point from a function we need to copy back from memory the values of the registers used locally, so that the calling routine will see no changes

A. use Stack as location where to copy registers
   → use STMFD

B. copy back from the Stack the copied registers
   → use LDMFD

Example Part 1: 2 inputs using registers, return in R0

```
@ ***  Simple Function Calls  ***
@ This program contains simple function calls
@ It illustrates the “BL" (branch and link)
@ instruction, as well as the standard methods of
@ returning from such calls.
@ A function/procedure and its caller need to agree
@ on a "calling convention", a statement about what
@ parameters are expected on entry, in what registers
@ and what is returned in what registers or memory
@ locations.

      .text
      .global     _start
_start:

@ Part 1: two inputs using registers, return in r0
```

```
@ Part 1: two inputs using registers, return in r0
        ldr   r1,=N1          @Set up parameters
        ldr   r1,[r1]         @r1 = value of N1
        ldr   r2,=N2
        ldr   r2,[r2]         @r2 = value of N2
        bl    max1   @r0 = max1(N1:r1,N2:r2)
exit:   swi   0x11            @ Terminate the program

@ max1: given 2 integers, it returns [illegible] the two
@ int:r0 max1 (int a:r1,b:r2)
max1:   stmfd sp!,{lr}
        mov   r0,r1
        cmp   r1,r2
        bge   endmax1
        mov   r0,r2           @ set return value
endmax1:
        ldmfd sp!,{pc}        @ and return to the caller
        .data
N1:     .word 15
N2:     .word 36
        .end
```

r1 and r2 are not changed here: no need to save?

DOCUMENTATION.....

Example Part 2: input parameters contains addresses of 2 variables, return in R0

```
@ Part 2: return MAX in r0, while R1 and R2 have addresses
@ of variables
       ldr    r1,=N1          @r1 = address of N1
       ldr    r2,=N2          @r2 = address of N2
       bl     max2            @r0 = max2(&N1:r1,&N2:r2)
exit:  swi    0x11            @ Terminate the program
max2:         @ int max1 (*a:r1,*b:r2)
@ max1: given 2 integers, it returns the max of the two
       stmfd  sp!,{r3,lr}    @save local register on stack
       ldr    r0,[r1]         @ R0 = N1
       ldr    r3,[r2]         @ R3 = N2
       cmp    r0,r3           @ compare N1 and n2
       bge    endmax2
       mov    r0,r3
endmax2:
       ldmfd  sp!,{r3,pc}     @ restore local register
                              @ and return to the caller
       .data
N1:    .word  15
N2:    .word  36
       .end
```

Example again: find max element in array

review

```
@ Find max element in an array with parameters:
@       R2 = array address
@       R1 = value of size of array
@       Return in R0 = max element
@       R0 = maxarray3 (size:R1, addr.array: R2)
        ldr   r1,=size
        ldr   r1,[r1]
        ldr   r2,=array
        bl    Maxarray3    @ int=maxarray3(& array:r2,
                           @      size:r1)
        swi   0x11
@****************************
maxarray3: code here (see next)
@****************************
        .data
array:        .word 10,-2,12, 13,-5,6,9,11,13,-2
size: .word 10
        .end
```

review

```
@ MaxArray3 function
@r0 = maxarray3 (size:r1, addr.array: r2)
Maxarray3:
       STMFD sp!,{r1,r2,r3,r4,lr}    @save registers
       LDR   r0,[r2]       @r3 = tempmax = array[0]
       MOV   r3,#1         @r0 = index into array
loop:  CMP   r3,r1         @is index at end of array?
       BEQ   endmaxarray3        @checked whole array
       LDR   r4,[r2,#4]!  @r4=array[i],increment r2
                           @ pointer
       CMP   r0,r4         @compare tempmax = array[index]
       BGE   incr          @tempmax ok, try next element
       MOV   r0,r4         @update tempmax
incr:  ADD   r3,r3,#1      @increment index
       BAL   loop
endmaxarray3:
       LDMFD r13!,{r1,r2,r3,r4,pc}   @restore registers
       @MOV  PC,LR         @ and return to the caller
       .data
array:       .word 10,-2,12, 13,-5,6,9,11,13,-2
```

better