CZ1006
CE1006

Modular Programming

A/P Goh Wooi Boon

Modular Programming

Subroutines

Learning Objectives (5a)

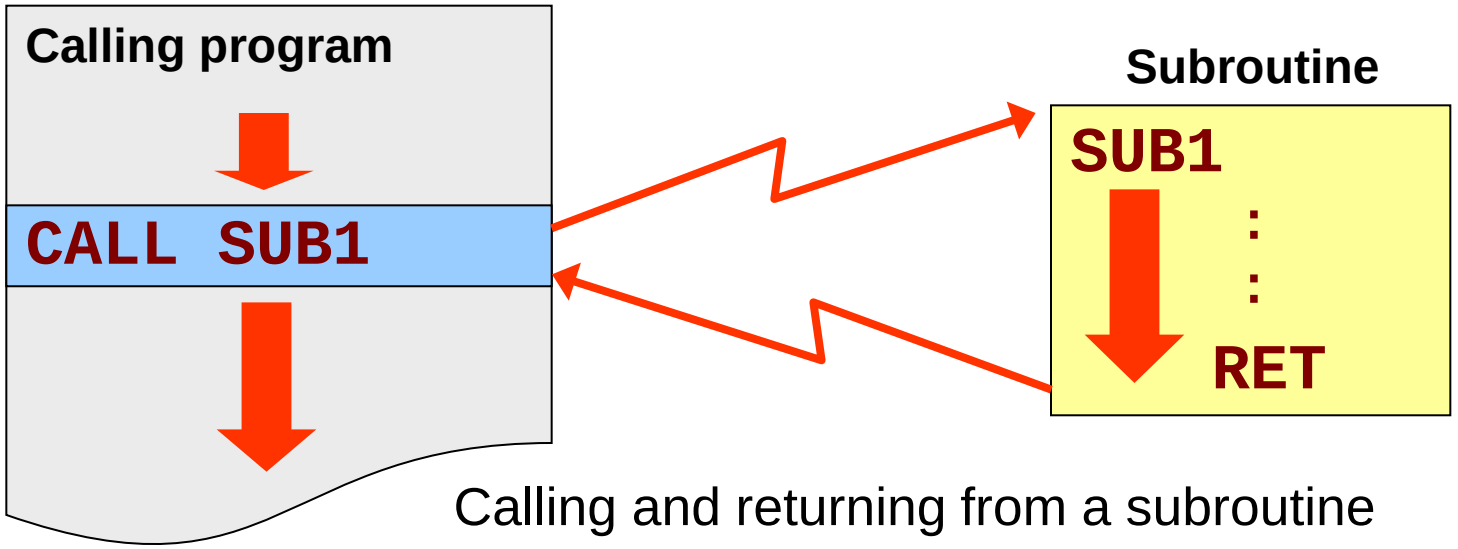
1. Describe the VIP instructions to implement subroutines.
2. Describe how the stack is used to support a subroutine call.

Modular Program Design

- Large, complex software should be decomposed into several less complex **modules**.
  - Modules can be designed and tested **independently**.
  - Modules can reduce overall **program size** as the same code segments may be required in several places
  - Modules that are general can be **re-used** in other projects.
- Characteristics of a good software module.
  - Loose coupling – **data** within module is entirely **independent** of other modules (local variables).
  - Strong modularity – should perform a **single** logically coherent **task**.

Subroutines

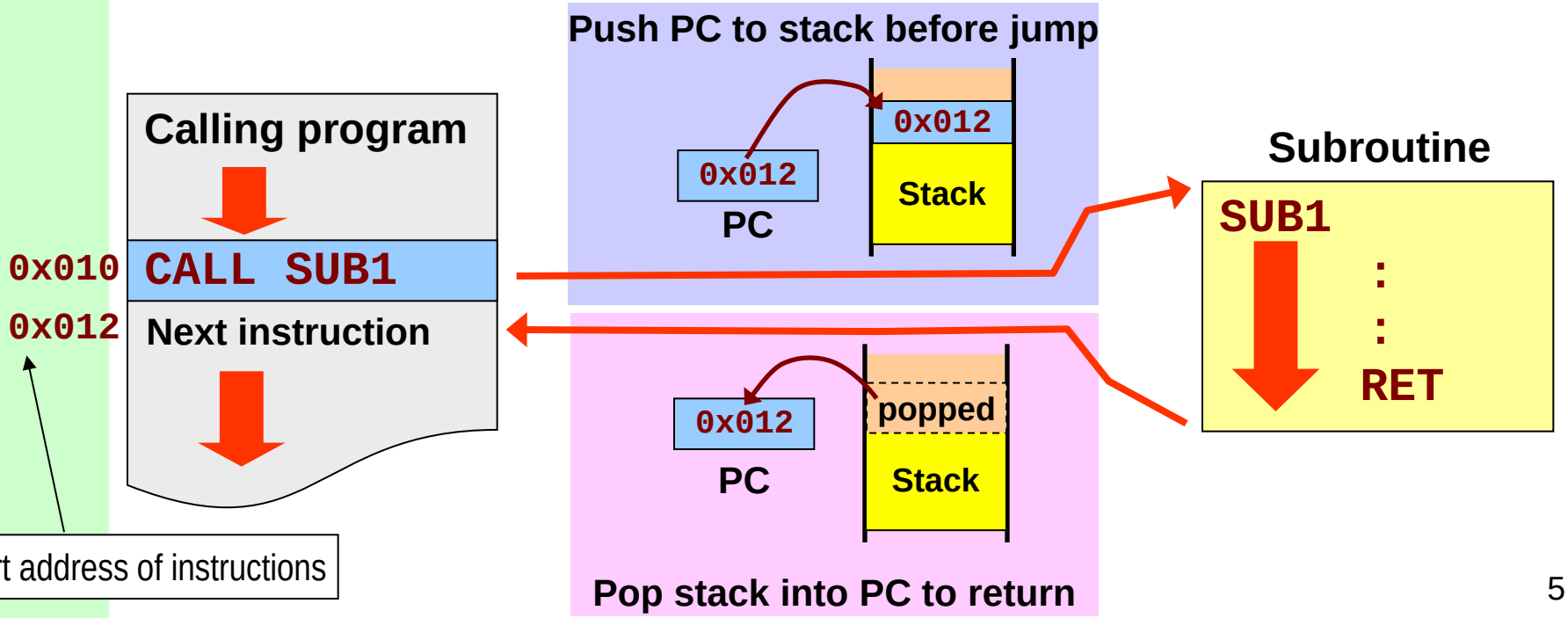
- Modules are implemented as **subroutines**.
- The same subroutine (or **function** in C) can be called from various parts of the program.
- On completion, it returns control to the place immediately after where the subroutine was called.
- In VIP, a combination of **`CALL`** and **`RET`** allow subroutine calls to be implemented.



Calling and returning from a subroutine

Subroutine Call and the System Stack

- In order to return to the calling program, a **return address** is saved away to the system stack before jumping to the subroutine.
- The return address is the **current PC** contents, which points to the start address of the instruction immediately after the **`CALL`** instruction.



CALL

- **CALL** is used to make a subroutine call.
  - **Return address** (i.e. current PC contents) is first pushed to the system stack.
  - Contents in **SP** is therefore decrement by 1.
  - An absolute jump is made to the start address where the subroutine is found.
  - VIPAS allows subroutine **address labels** to be used.

Subroutine Call **(VIP examples)**

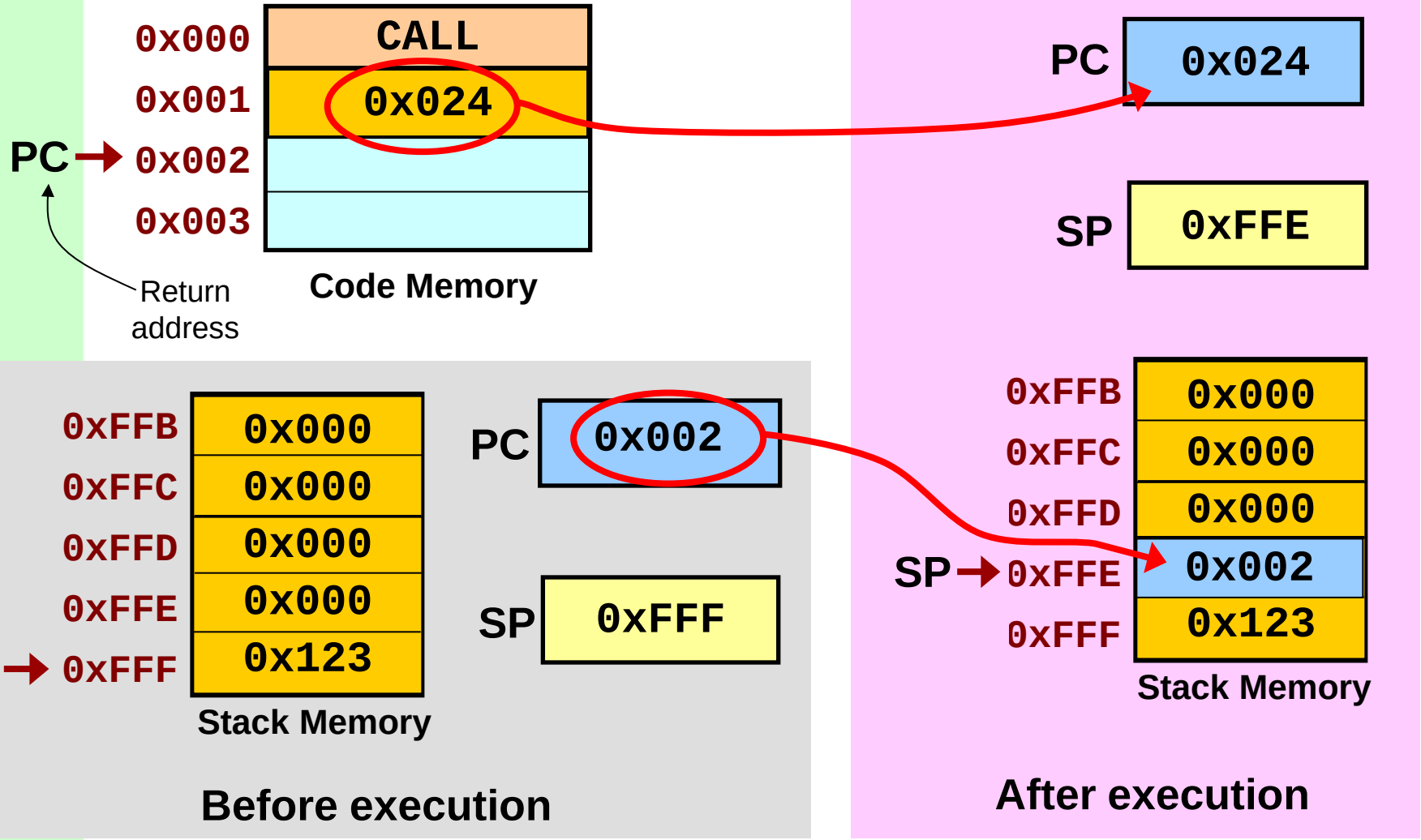
```
CALL #0x020      ; using actual address
CALL SortSub     ; using labels
```

  - Care must be taken when implementing many **nested subroutine** calls as risk of stack overflow may occur.

(Execution example)

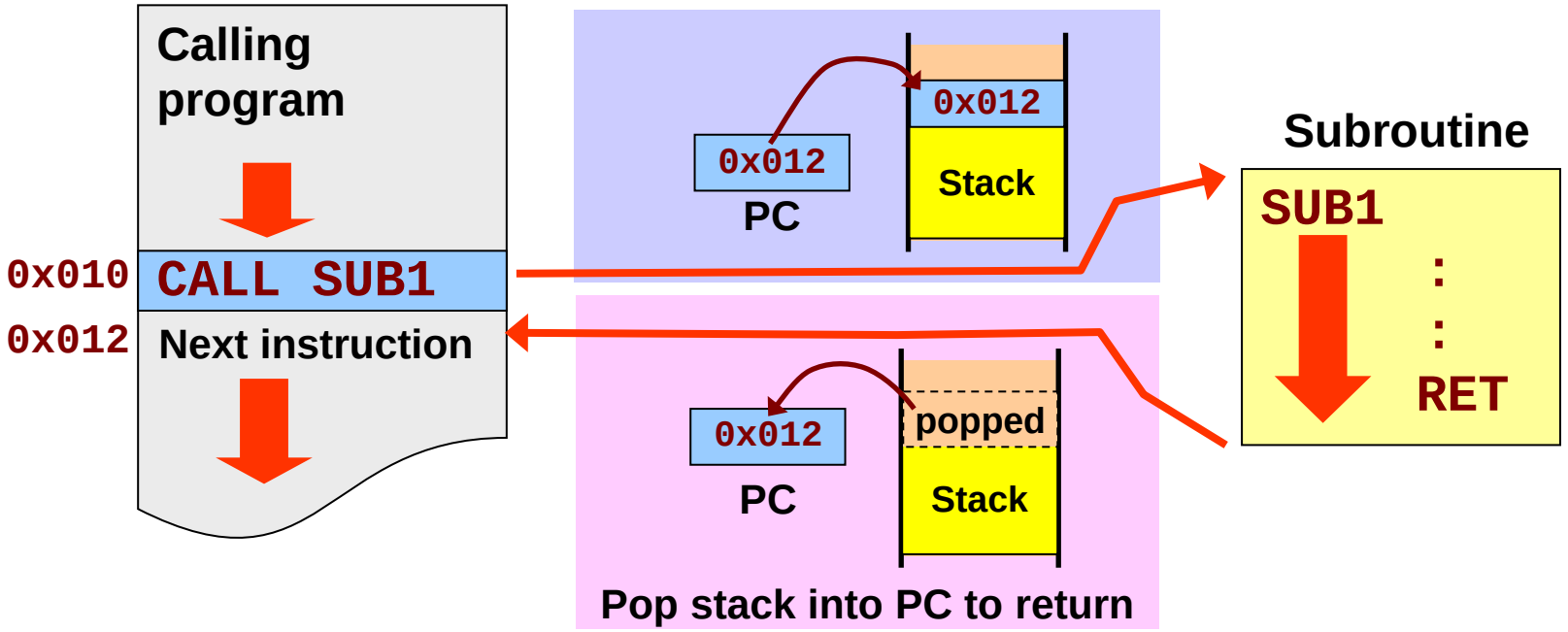
CALL

- e.g. **CALL #0x024**



RET

- **RET** returns from a subroutine.
  - Return address currently on the top of the stack is popped and placed into **PC** and contents in **SP** increments by 1.



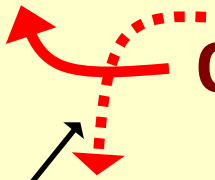
  - **Note:** Any pushes to the stack done within the subroutine must be accompanied by the **same numbers** of pops before **RET** is executed. This ensures the return address is currently at the top of the stack before executing **RET**.

Recursive Subroutine

- A recursive routine calls itself within its own body.
  - Recursion is an elegant way to solve algorithms or mathematical expressions that have **systematic repetitions**. (e.g. factorial **n! = n×(n-1)×(n-2)….2×1**)

Carelessly written recursive program may cause stack overflow during execution

Recursive Code Example



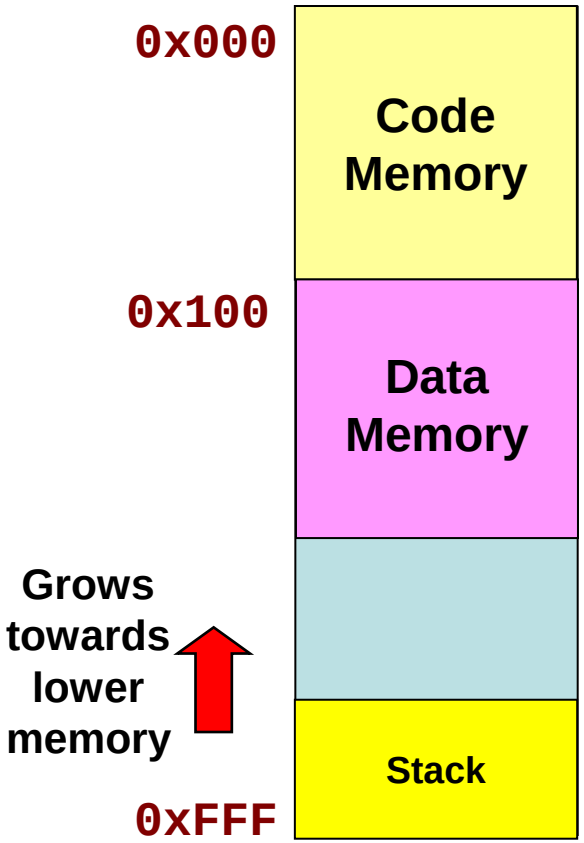
```
Recur   :                ; Subroutine Recur
        CALL  Recur      ; call Recur with Recur routine
          :
        RET              ; return to calling program
```

Note: Some condition must be reached that allows **`CALL Recur`** to be skipped in order to avoid infinite recursion and stack overflow.

(VIP example)

System Stack

- The system stack is a very important requirement in all microprocessor system.
- Beside subroutine calls, system stack is also used in **exception** handling and as a temporary storage area for **local variables** and subroutine **parameters**.
- Since stack grows towards lower memory, it can be maintained at the high **RAM** area, e.g. starting at **0xFFF**.



VIP memory map

Summary

- **CALL** and **RET** are the two basic instructions for implementing subroutines.
- The stack is used to save and retrieve the **return PC address** during a subroutine call.
- Every **CALL** execution must be eventually accompanied by its respective **RET** to ensure the stack does not overflow.

Modular Programming

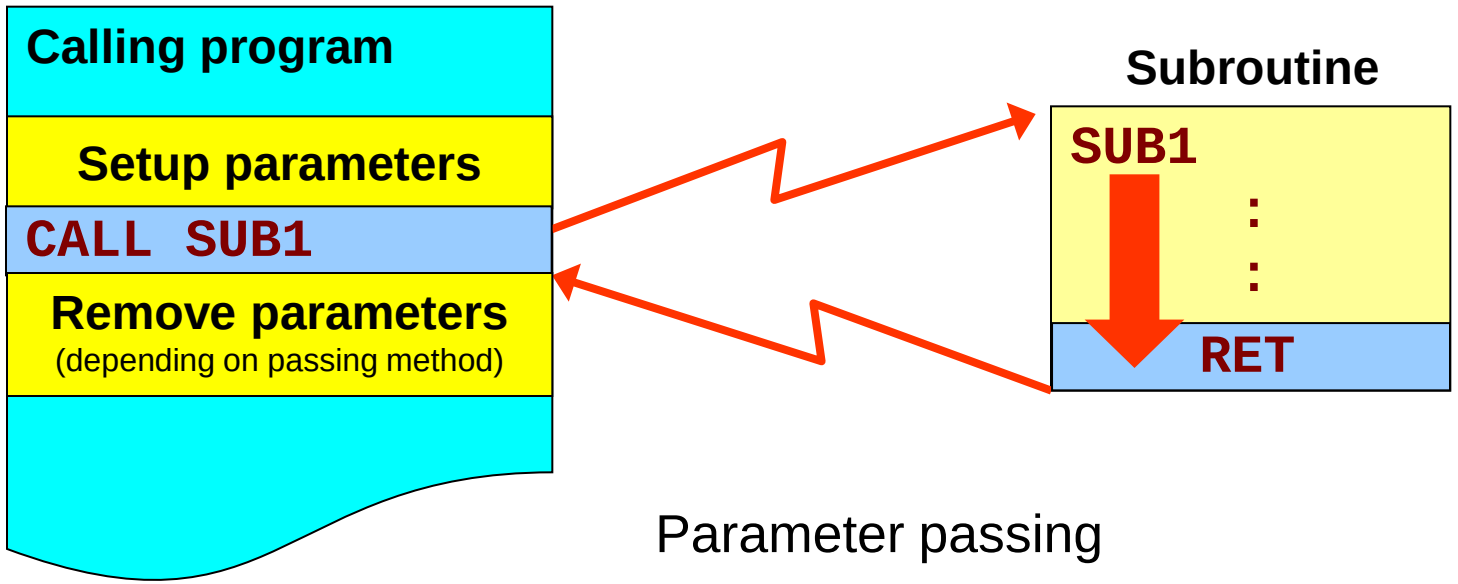
Parameter Passing by Register and by Memory Block

Learning Objectives (5b)

1. Describe how registers can be used to pass parameters.
2. Describe the pros and cons of using registers to pass parameters.
3. Describe how a memory block can be used to pass parameters.

Parameter Passing

- Calling programs need to pass parameters to influence a subroutine's execution.
  - Parameters must be setup properly before the subroutine is called and appropriately removed after returning.
  - There are three basic methods to pass parameters, via **registers**, **memory block** or the **system stack**.



Parameter passing

Parameter Passing using Registers

- Parameters are placed into the register before calling the subroutine.
  - **Number** of parameters passed are **limited** to the **available registers**.
  - Useful when number of parameters are **small**.
  - **Pro – efficient** as parameters are already in register within the subroutine and can be used immediately.
  - **Con** - **reduces available registers** that can be used within subroutine.
  - **Con – lacks generality** due to the limited number of registers.

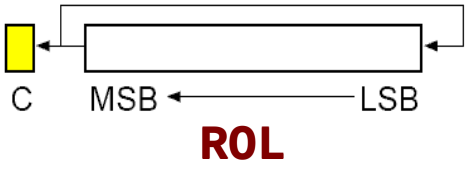
Subroutine Example

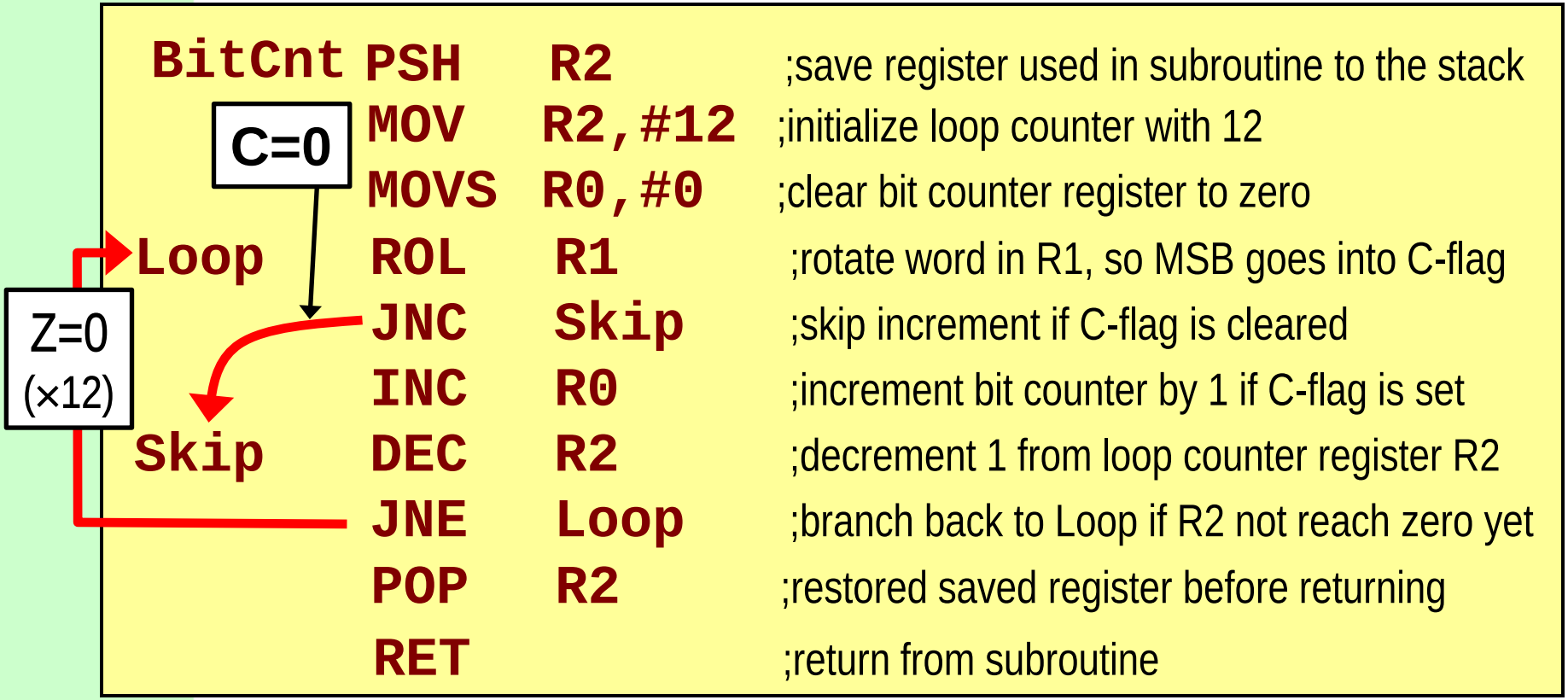
Bit Counting Subroutine

- Write a subroutine to:
  - Count the number of "1" bits in a word.
  - Return result in register **R0**.
- **Design considerations:**
- How do we transfer the word into the subroutine?
  - Put the word into a **register**, which can then be accessed within the subroutine (e.g. register **R1**).
- How do we check if each individual bit is a "1" or a "0"?
  - Rotate **R1** left 12 times. After each rotate, test carry bit to check if (**C**=1). If yes, increment bit counter register **R0**.

Bit Counting Subroutine

Possible Solution #1





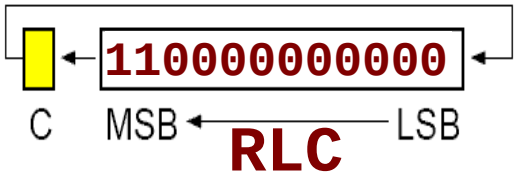

```
BitCnt  PSH   R2        ;save register used in subroutine to the stack
        MOV   R2,#12    ;initialize loop counter with 12
        MOVS  R0,#0     ;clear bit counter register to zero
Loop    ROL   R1        ;rotate word in R1, so MSB goes into C-flag
        JNC   Skip      ;skip increment if C-flag is cleared
        INC   R0        ;increment bit counter by 1 if C-flag is set
Skip    DEC   R2        ;decrement 1 from loop counter register R2
        JNE   Loop      ;branch back to Loop if R2 not reach zero yet
        POP   R2        ;restored saved register before returning
        RET             ;return from subroutine
```

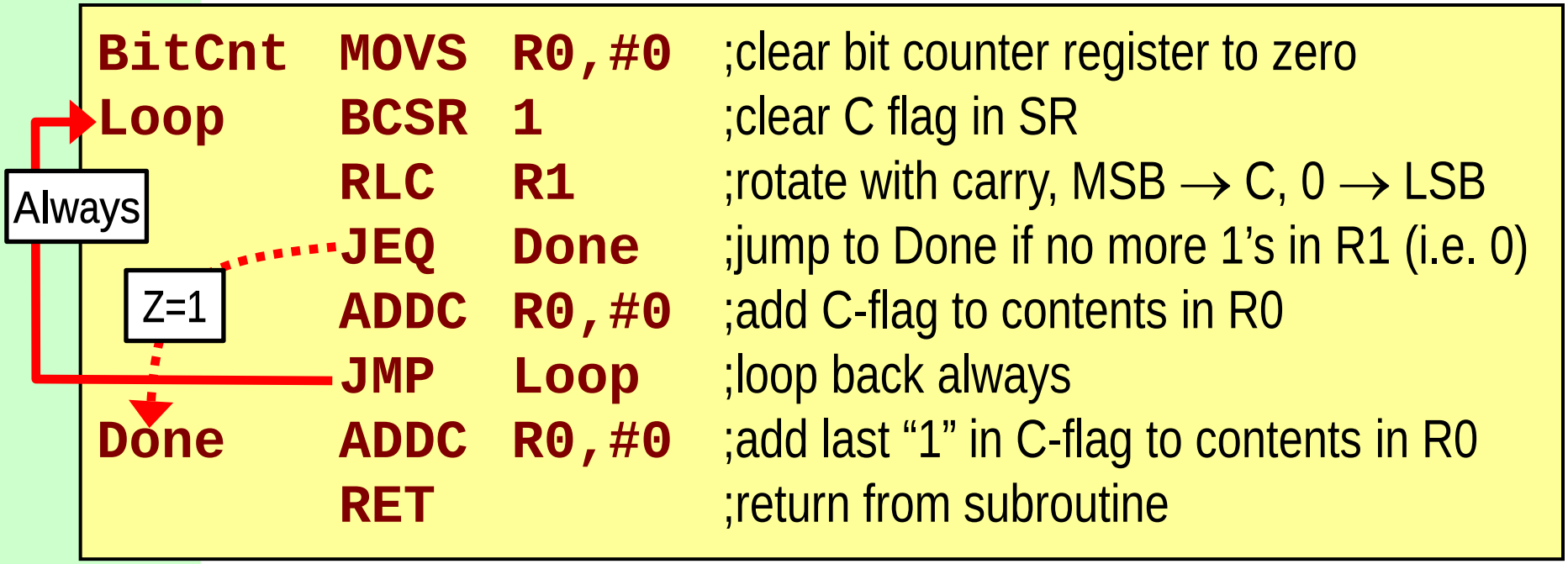
R0 = register to pass out result **R1** = register to pass parameter in **R2** = Loop counter

Note: The subroutine uses **R2** but **restores** its content to its original values **before returning** as **R2** may be used by the calling program.

Bit Counting Subroutine

Alternative Solution #2






```
BitCnt  MOVS  R0,#0   ;clear bit counter register to zero
Loop    BCSR  1       ;clear C flag in SR
        RLC   R1      ;rotate with carry, MSB → C, 0 → LSB
        JEQ   Done    ;jump to Done if no more 1's in R1 (i.e. 0)
        ADDC  R0,#0   ;add C-flag to contents in R0
        JMP   Loop    ;loop back always
Done    ADDC  R0,#0   ;add last "1" in C-flag to contents in R0
        RET           ;return from subroutine
```

R0 = Register to pass out result. **R1** = Register for passing parameter in

Note:
1. Unlike the solution #1, the parameter passed in is **destroyed**.
2. This example shows one of the uses of the **ADDC** instruction.
3. **Faster** routine if most of the 1's are in the MSB side of the register.
4. No need for additional register **R2**, so no need to save & restore

Parameter Passing using Memory

- A region in memory is treated like a mailbox and is used by both the calling program and subroutine.
  - Parameters to be passed are gathered into a **block** at a predefined memory location
  - The start address of the memory block is passed to the subroutine via an **address register**.
  - Useful for passing **large number of parameters**.

Subroutine Example

Lower to Upper Case Subroutine

- Write a subroutine to:
  - To convert an ASCII string from lower to upper case.
  - The string is terminated by a NULL character (**0x000**).
  - The start address of the string is passed via **R1**.
  - A segment of the calling program shows how the parameter is setup and the subroutine called:

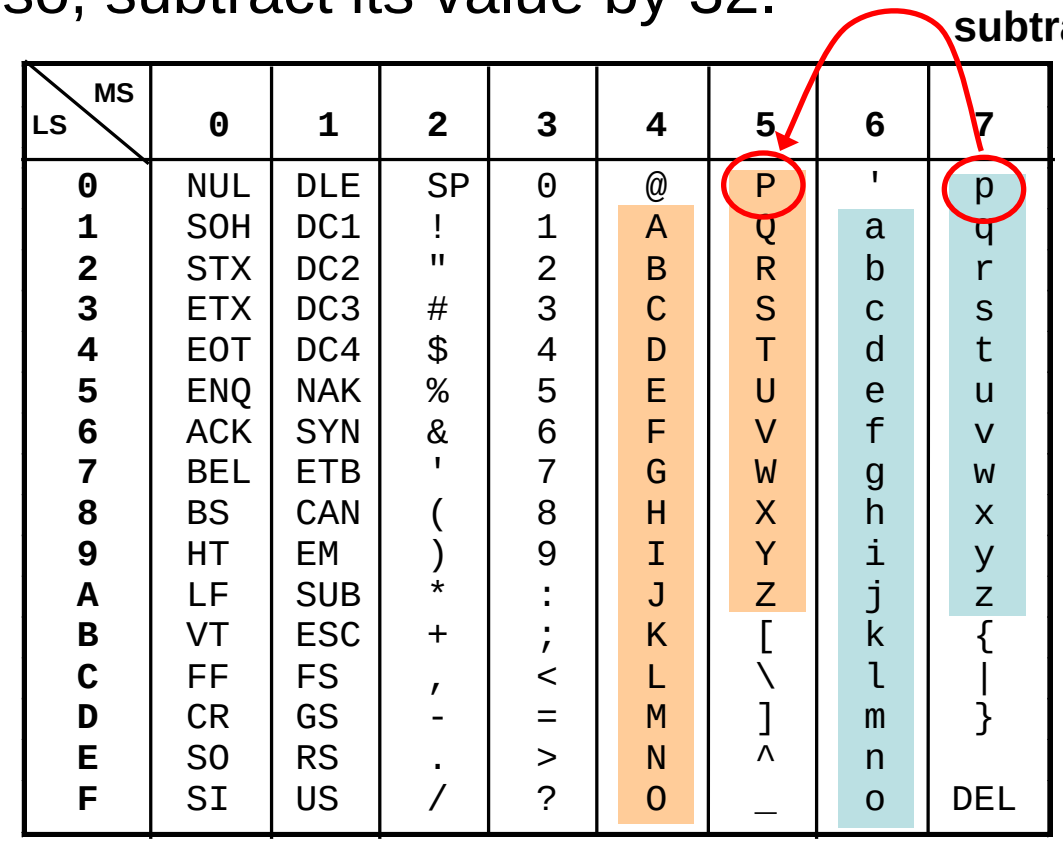
```
;Calling program
    :
MOV   R1,#0x100   ;move start addr. of string to R1
CALL  Lo2Up       ;branch to Lo2Up subroutine
    :
```

Address	Memory
0x100	"a"
0x101	"p"
0x102	"p"
0x103	"l"
0x104	"e"
0x105	0x000
0x106	:

Lower to Upper Case Subroutine

Algorithm Design

- How to convert an ASCII character from lower to upper case?
  - Check that the character's value is between 'a' and 'z'.
  - If so, subtract its value by 32.

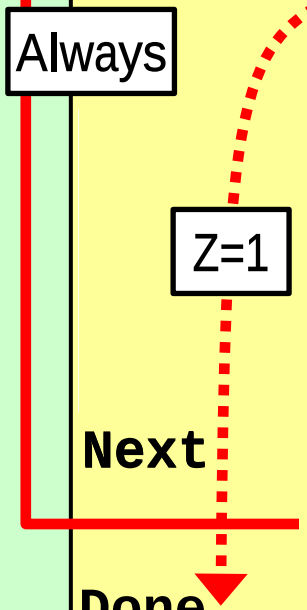


LS \ MS	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	'	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	
F	SI	US	/	?	O	_	o	DEL

ASCII Character Set (7-Bit Code)

Lower to Upper Case Subroutine

Possible Solution




```
Lo2Up   PSHM    0x003       ;save registers used within subroutine
Loop    MOV     R0,[R1]     ;get current char from string in memory
        JEQ     Done        ;if NULL char, branch to Done
        CMP     R0,#0x061   ;compare with lower limit "a"
        JLT     Next        ;if smaller than "a", do not convert
        CMP     R0,#0x07A   ;compare with upper limit "z"
        JGT     Next        ;if greater than "z" do not convert
        SUB     R0,#32      ;convert to upper case by subtracting 32
        MOV     [R1],R0     ;write modified char back to string in memory
Next    INC     R1          ;increment pointer to next char in string
        JMP     Loop        ;branch back to Loop if not zero
Done    POPM    0x003       ;restored saved registers before returning
        RET                 ;return from subroutine
```

Note: Subroutine modifies **R0** & **R1** but restores their original contents before returning. This produces a **transparent subroutine** that does not effect the proper operation of calling program

Summary

- Passing parameters **using registers** is the simplest and fastest.
  - Number of parameters that can be passed is **limited** by the **available registers**.
- Passing parameters using a **memory block** can support a **large number** of parameters or data types like arrays.

Modular Programming

Passing Parameters using the Stack

Learning Objectives (5c)

1. Describe how parameters can be passed and retrieve using the stack.
2. Identify the difference between passing by value and by reference.
3. Describe how a transparent subroutine can be implemented.

Parameter Passing using Stack

- Parameters are pushed onto the stack before calling the subroutine and retrieved from the stack within the subroutine.
  - Most **general** method of parameter passing – no registers needed, supports recursive programming.
  - **Large** numbers of parameters can passed as long as stack does not overflow.
  - Parameters pushed to the stack must be **removed** by the calling program immediately after returning from subroutine.
  - If not, repeated pushing of parameters to the stack will lead to a stack overflow.

Subroutine Example

Sum from 1 to N

- Write a subroutine to:
  - Sum the positive numbers from 1 to **N**, where **N** is a value passed to the subroutine.
  - The computed sum should be directly updated to a memory variable **Answer**, whose address is **`0x100`**.
  - All parameters are to be passed via the **stack**.

Solution:

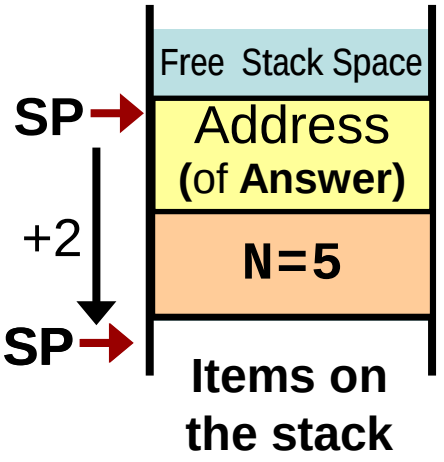
- Push two parameters on stack, the value of **N** and **address** of memory variable **Answer**.
- Use **register indirect** addressing to directly update the memory variable **Answer** within the subroutine.

Sum from 1 to N Subroutine

Calling the Subroutine

```
        :
        :                          ;find the sum of (1+2+3+4+5), where N=5
Parameters setup
        PSH   #5                   ;push value of  N=5 to the stack
        PSH   #0x100               ;push address of memory variable to stack
        CALL  Sum1N                ;call subroutine Sum1N
        ADD   SP,#2                ;add 2 to pop the two parameters from stack
        :
```

Remove parameters from the Stack

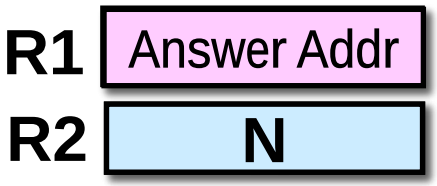


Note:

1. Parameters set up before calling the subroutine are **removed** immediately after returning from subroutine.
2. Removal is done by returning the contents of the stack pointer to its **original value** before the parameters were pushed to the stack.

Sum from 1 to N Subroutine

Possible Solution




```
Sum1N   PSHM   7            ;save registers R0,R1 & R2 to stack
        MOV    R2,[SP+5]    ;retrieve value of N from stack to R2
        MOV    R1,[SP+4]    ;retrieve Answer's address from stack
        MOVS   R0,#0        ;clear summation register R0 to 0
Loop    ADD    R0,R2        ;add current value in R2 to R0
        DEC    R2           ;decrement current value in R2 by 1
        JNE    Loop         ;jump back to Loop if R2 not yet zero
        MOV    [R1],R0      ;write sum to Answer using reg. indirect
        POPM   7            ;restored saved registers before return
        RET                 ;return from subroutine
```

Retrieve stack parameters

Z=0

Z=1

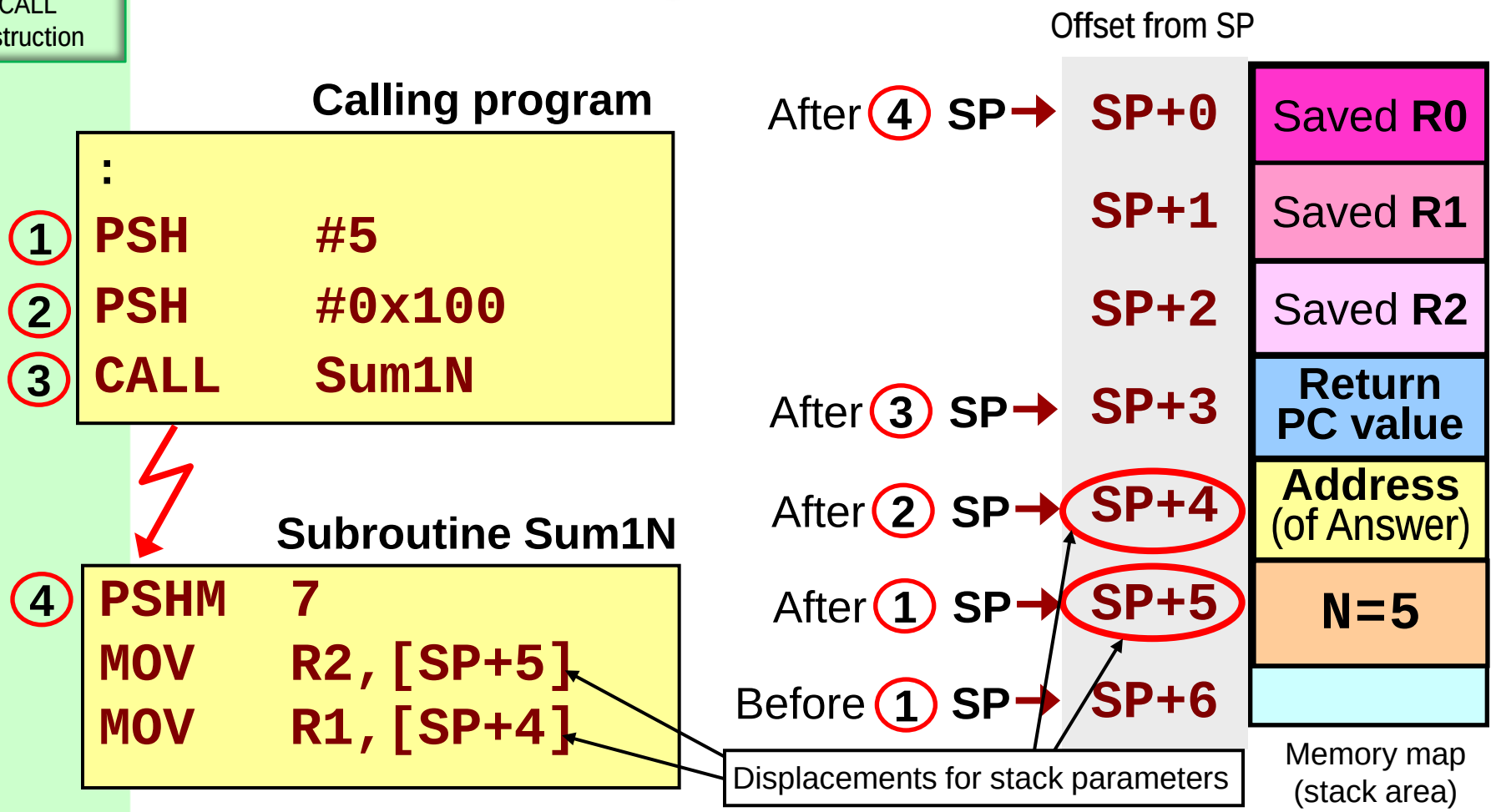
Note: The subroutine needs three registers. **R0** to compute the sum from 1 to N. **R1** to be an address pointer to the memory variable **Answer** where the results will written to. **R2** holds the value of N, which is decremented by 1 after each loop till it reaches 0.

Transparent Subroutine

CALL Instruction

Sum from 1 to N Subroutine

Accessing Stack Parameters



Note: **SP indirect with offset** can be used to access parameters on the stack but knowledge of all items on the stack is needed to compute the correct offset from the current **SP** position.

Passing by value and by reference

- Parameters are passed to subroutines in 2 ways:
  - **Pass by value** – the value of the data (or variable) is passed to the subroutine.
  - **Pass by reference** – the address of the variable is passed to the subroutine .

Calling program

```
PSH    #5
PSH    #0x100
CALL   Sum1N
```

- When is passing by reference used?
  - When the **parameter** passed is to be **modified** by the subroutine.
  - **Large quantity** of data (e.g. array) have to be passed between subroutine and calling program.

C Function Example

- **C** function to compute the mean value of **N** elements in an integer **Array**.

Passing by reference

Passing by value

```
int Mean (int *Array, int N)
{
        int avg, i;
        int sum = 0;

        for (i=0; i<N; i++)
          sum = sum + Array[i];
        avg = sum / N;

        return avg;
}
```

Local variables used only by the function

Function's single output is normally returned via the R0 register

Note: Observe that **local variables** are required by the function to compute the result.

Transparent Subroutines

- A transparent subroutine will **not affect any CPU resources** used by the program calling it.
- To achieve this, all working registers used by the subroutine must be **saved on the stack** on entry and **restored from stack** before returning.

Select s R3,R2,R1,R0

```
SUB1 PSHM 0x00C      ; save R2 & R3 to stack (0x00C=....1100₂)
     :
     :               ; registers R2 and R3 are
     :               ; used in subroutine
     POPM 0x00C      ; restore R2 & R3 by popping from stack
     RET             ; return to calling program
```

- VIP provide efficient multiple stack push (**PSHM**) and pop (**POPM**) instructions to perform this task.

Summary

- Using the **stack** is the most favored means of passing parameters.
  - A combination of methods can be used to implement **Functions**, e.g. parameters passed in via stack and a single result value passed out via a register (e.g. **R0**).
  - Stack-based parameter passing supports **recursion**.
- Parameters is passed by **value** or **reference**.
  - Passing by reference allows the subroutine to directly access memory variables within the calling program .
- A **transparent** subroutine requires registers **used within** the routine to be saved on the **stack** on entry and restore before returning.