XMUT202 Digital Electronics

Stacks Programming

Week 10 Lecture 1

Agatha Rachmat
School of Engineering and Computer Science
Victoria University of Wellington

Today's topic

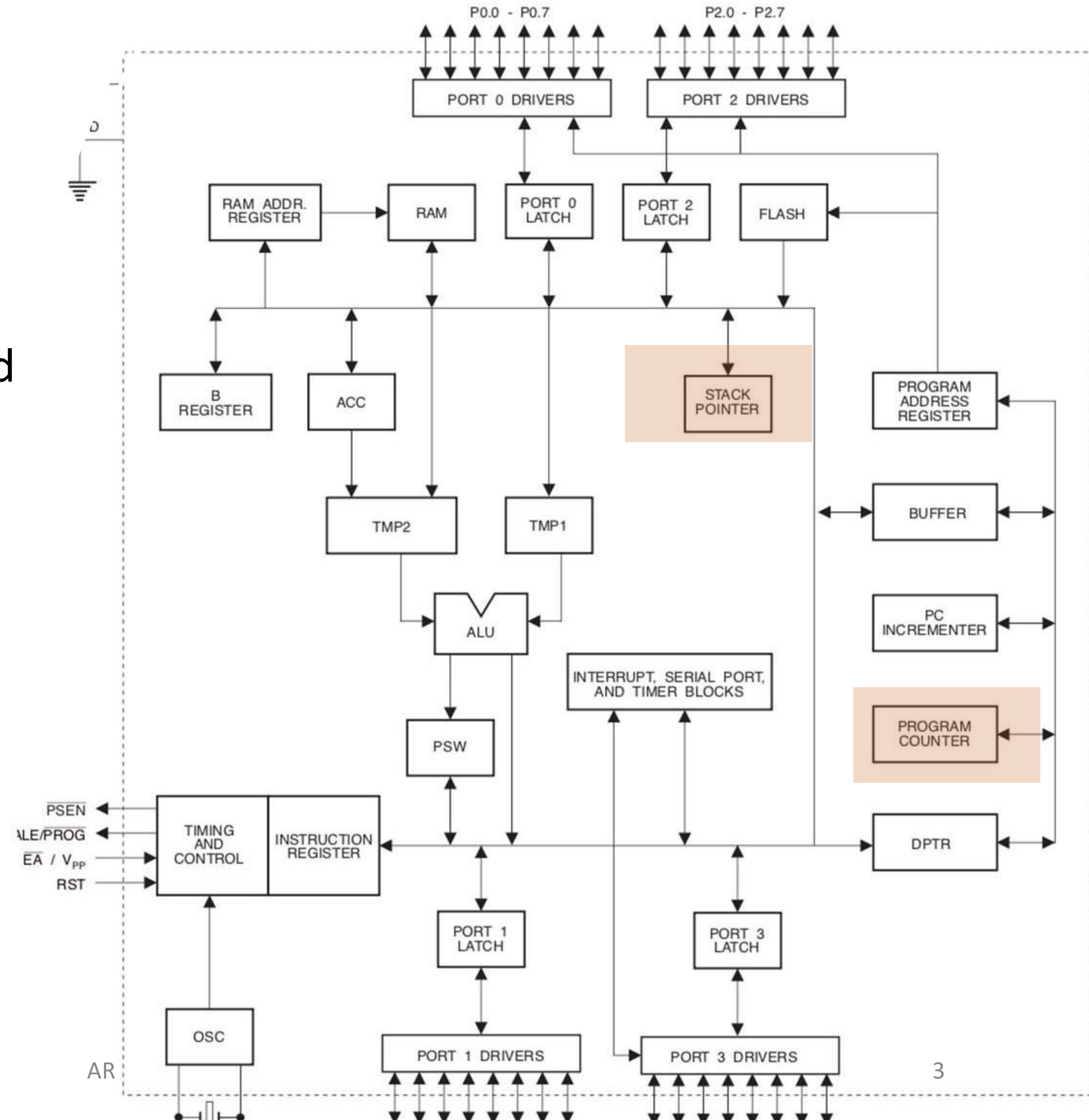
- Intro to stack.
- On chip data memory organisation
- Stack in 8051.
- Stacks mechanism.
- Custom stack.
- Instruction with stack

Intro to Stack

- Common in computer architecture and programming
- Key data structure in programming
- Related to memory usage

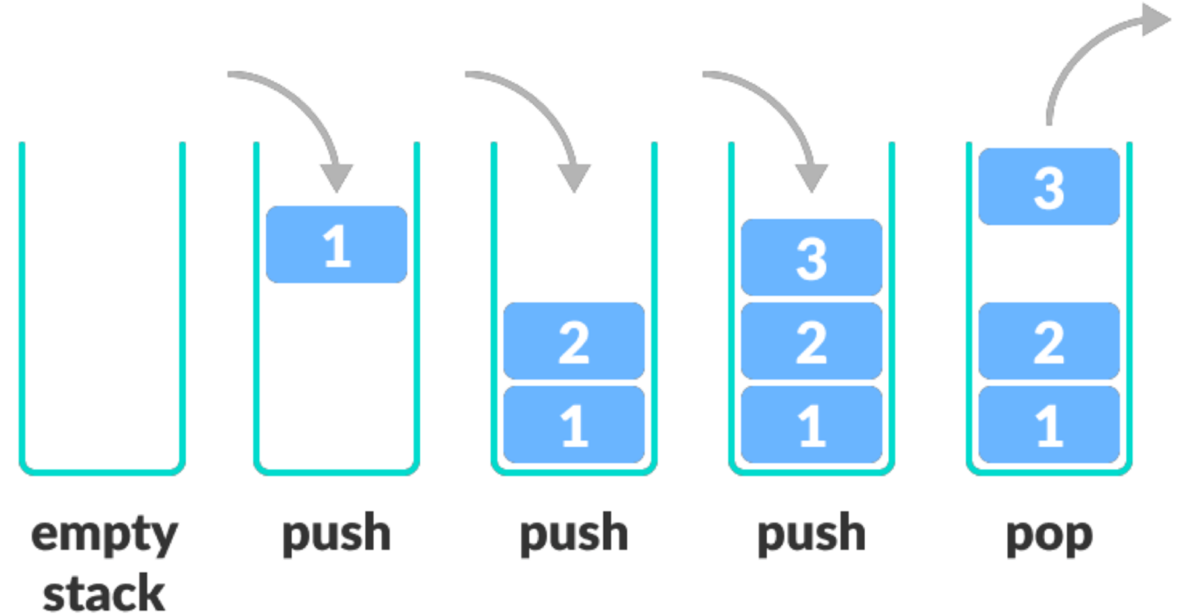
Important registers in the 8051 microcontroller are:

- Stack pointer
- Program Counter



Intro to Stack

Stack: a memory region to which data may be 'pushed' and from which data may be 'popped'.



Push: adds data to the stack.

Pop (AKA 'Pull'): the most recently added data is removed from the stack.

A stack is a "LIFO" structure: Last In, First Out.

Intro to Stack

- The computer must keep track of the top of the stack: as the stack grows, the memory address will increase.
- This address is stored by the stack pointer.
- As the stack grows, the stack pointer holds the address of the most recently added item.
- Stacks are used to temporarily store memory addresses while the computer does others. e.g., memory addresses before jumps to subroutines may be stored on the stack.
- Many CISC computers have a hardware stack; most RISC machines have a software stack.

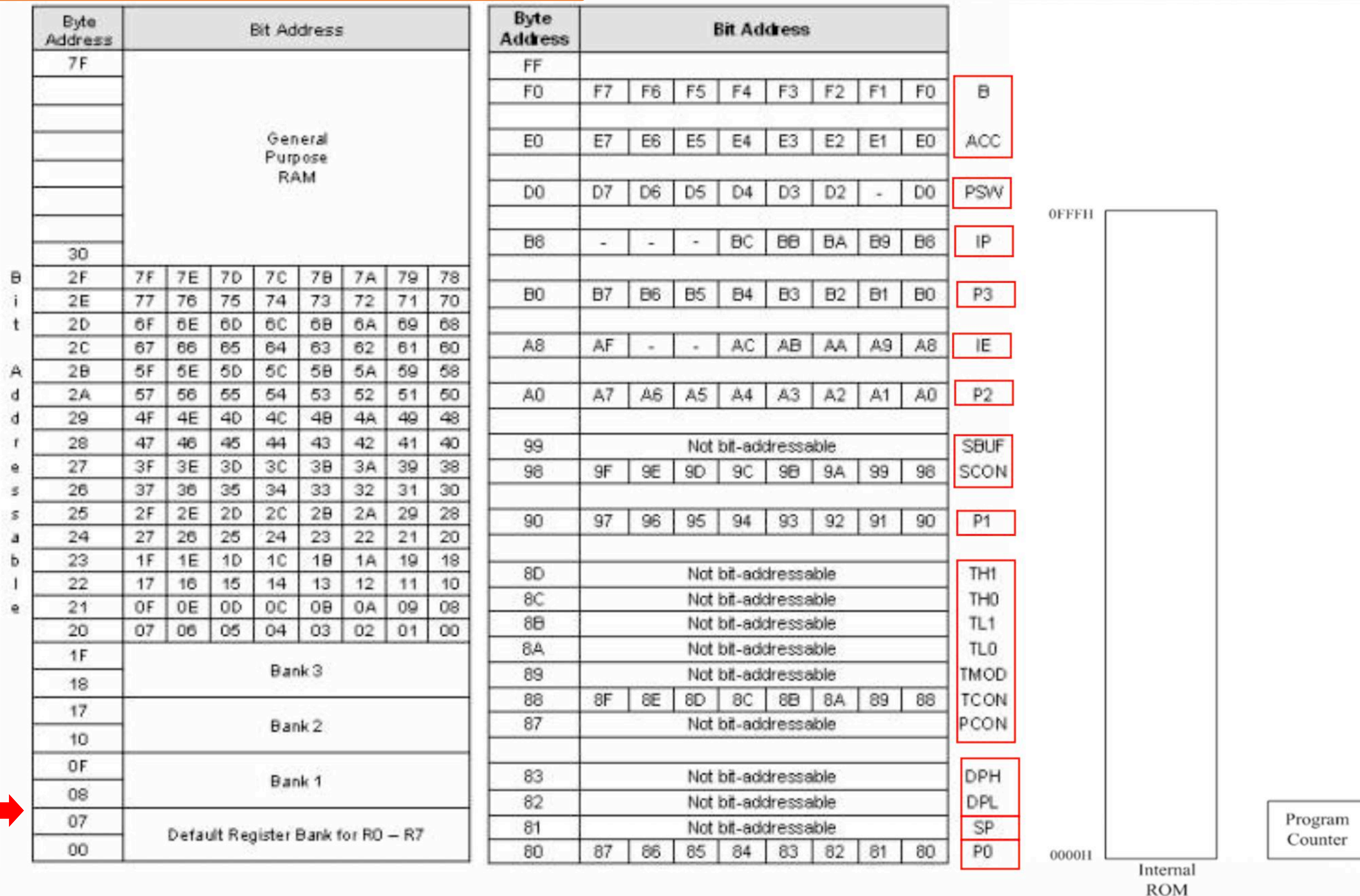
Intro to Stack

The **Stack Pointer (SP)** in the **8051 microcontroller** is a special register that **points** to the **top of the stack** in RAM.
A **stack** is a temporary storage area used to store:

1. return addresses during function calls
2. temporary data
3. register values during interrupts

The Stack Pointer is an **8-bit register** that stores the address of the top of the stack.
In the 8051:
•SP is initialised to **07H** after reset.
•This means the first pushed data is stored at **08H**.

On-Chip Data Memory Organisation



Byte Address	Bit Address							
7F	General Purpose RAM							
30								
2F	7F	7E	7D	7C	7B	7A	79	78
2E	77	76	75	74	73	72	71	70
2D	6F	6E	6D	6C	6B	6A	69	68
2C	67	66	65	64	63	62	61	60
2B	5F	5E	5D	5C	5B	5A	59	58
2A	57	56	55	54	53	52	51	50
29	4F	4E	4D	4C	4B	4A	49	48
28	47	46	45	44	43	42	41	40
27	3F	3E	3D	3C	3B	3A	39	38
26	37	36	35	34	33	32	31	30
25	2F	2E	2D	2C	2B	2A	29	28
24	27	26	25	24	23	22	21	20
23	1F	1E	1D	1C	1B	1A	19	18
22	17	16	15	14	13	12	11	10
21	0F	0E	0D	0C	0B	0A	09	08
20	07	06	05	04	03	02	01	00
1F	Bank 3							
18								
17	Bank 2							
10								
0F	Bank 1							
08								
07	Default Register Bank for R0 – R7							
00								

Bit Addressable

Byte Address	Bit Address								
FF									
F0	F7	F6	F5	F4	F3	F2	F1	F0	B
E0	E7	E6	E5	E4	E3	E2	E1	E0	ACC
D0	D7	D6	D5	D4	D3	D2	-	D0	PSW
B8	-	-	-	BC	BB	BA	B9	B8	IP
B0	B7	B6	B5	B4	B3	B2	B1	B0	P3
A8	AF	-	-	AC	AB	AA	A9	A8	IE
A0	A7	A6	A5	A4	A3	A2	A1	A0	P2
99	Not bit-addressable								SBUF
98	9F	9E	9D	9C	9B	9A	99	98	SCON
90	97	96	95	94	93	92	91	90	P1
8D	Not bit-addressable								TH1
8C	Not bit-addressable								TH0
8B	Not bit-addressable								TL1
8A	Not bit-addressable								TL0
89	Not bit-addressable								TMOD
88	8F	8E	8D	8C	8B	8A	89	88	TCON
87	Not bit-addressable								PCON
83	Not bit-addressable								DPH
82	Not bit-addressable								DPL
81	Not bit-addressable								SP
80	87	86	85	84	83	82	81	80	P0

Intro to Stack

The CPU uses both areas together while executing instructions. CPU as the “manager”:

- it reads instructions
- uses the **SFRs (right side)** to control hardware
- uses the **RAM (left side)** to store data temporarily

1. CPU + LEFT SIDE (RAM)

The CPU uses the left memory area to:

- store temporary data
- keep variables
- store stack data
- access register banks

Example:

```
MOV 30H,#55H
```

CPU action:

```
Store 55H into RAM address 30H
```

This uses the LEFT side.

2. CPU + RIGHT SIDE (SFRs)

The CPU uses SFRs to control hardware functions.

Example:

```
MOV P1,#0FFH
```

CPU action:

```
Send 11111111 to Port 1 pins
```

Here:

- `P1` is an SFR on the RIGHT side
- CPU communicates with external pins through it

Intro to Stack

3. CPU Using BOTH Together

Example:

```
MOV SP,#30H
PUSH ACC
```

Step 1: CPU accesses SP (RIGHT side)

`SP` is an SFR register.

CPU stores:

```
SP = 30H
```

Step 2: CPU executes PUSH

CPU internally does:

```
SP = SP + 1
```

Now:

```
SP = 31H
```

Step 3: CPU stores ACC into RAM (LEFT side)

CPU takes the value in ACC and stores it in:

```
RAM[31H]
```

So:

Part	Used For
RIGHT side	SP register controls stack
LEFT side	actual stack data stored

Intro to Stack

To the CPU:

- Both are memory addresses
- It accesses them by address

Example:

The CPU reads/writes addresses internally.

Address	**Meaning**
30H	RAM location
90H	P1 register
81H	SP register
E0H	ACC

Intro to Stack

Suppose:

```
MOV SP,#30H
PUSH ACC
```

Then internally:

SP register

```
SP = 31H
```

RAM location

```
RAM[31H] = ACC value
```

So:

- SP is the “pointer” stored in the RIGHT side
- 31H is the actual memory location on the LEFT side

Bit Addressable

Byte address								
30								
2F	7F	7E	7D	7C	7B	7A	79	78
2E	77	76	75	74	73	72	71	70
2D	6F	6E	6D	6C	6B	6A	69	68
2C	67	66	65	64	63	62	61	60
2B	5F	5E	5D	5C	5B	5A	59	58
2A	57	56	55	54	53	52	51	50
29	4F	4E	4D	4C	4B	4A	49	48
28	47	46	45	44	43	42	41	40
27	3F	3E	3D	3C	3B	3A	[illegible]	38
26	37	36	35	34	33	32	31	30
25	2F	2E	2D	2C	2B	2A	[illegible]	28
24	27	26	25	24	23	22	21	20
23	1F	1E	1D	1C	1B	1A	19	18
22	17	16	15	14	13	12	11	10
21	0F	0E	0D	0C	0B	0A	09	08
20	07	06	05	04	03	02	01	00
1F / 18	Bank 3							
17 / 10	Bank 2							
0F / 08	Bank 1							
07 / 00	Default Register Bank for R0 – R7							

Byte address							
B8	-	-	-	BC	BB	BA	B9
B0	B7	B6	B5	B4	B3	B2	B1
A8	AF	-	-	AC	AB	AA	A9
A0	A7	A6	A5	A4	A3	A2	A1
99	Not bit-addressable						
98	9F	9E	9D	9C	9B	9A	99
90	97	96	95	94	93	92	91
8D	Not bit-addressable						
8C	Not bit-addressable						
8B	Not bit-addressable						
8A	Not bit-addressable						
89	Not bit-addressable						
88	8F	8E	8D	8C	8B	8A	89
87	Not bit-addressable						
83	Not bit-addressable						
82	Not bit-addressable						
81	Not bit-addressable						
80	87	86	85	84	83	82	81

On-Chip Data Memory Organisation

Most 8051 internal registers are mapped to on-chip RAM and, therefore, have an address (21 of these SFR):

- Stack Pointer
- Data Pointer
- PSW, TMOD, etc.
- ACC, B, R0-R7 etc.

Exceptions:

- Program Counter.
- Instruction Register.

Reason: Little point in addressing or manipulating these registers directly.

On-Chip Data Memory Organisation

Byte Address	Bit Address							
7F	General Purpose RAM							
30								
2F	7F	7E	7D	7C	7B	7A	79	78
2E	77	76	75	74	73	72	71	70
2D	6F	6E	6D	6C	6B	6A	69	68
2C	67	66	65	64	63	62	61	60
2B	5F	5E	5D	5C	5B	5A	59	58
2A	57	56	55	54	53	52	51	50
29	4F	4E	4D	4C	4B	4A	49	48
28	47	46	45	44	43	42	41	40
27	3F	3E	3D	3C	3B	3A	39	38
26	37	36	35	34	33	32	31	30
25	2F	2E	2D	2C	2B	2A	29	28
24	27	26	25	24	23	22	21	20
23	1F	1E	1D	1C	1B	1A	19	18
22	17	16	15	14	13	12	11	10
21	0F	0E	0D	0C	0B	0A	09	08
20	07	06	05	04	03	02	01	00
1F	Bank 3							
18								
17	Bank 2							
10								
0F	Bank 1							
08								
07	Default Register Bank for R0 – R7							
00								

Bit Addressable (rows 20–2F)

Byte Address	Bit Address								
FF									
F0	F7	F6	F5	F4	F3	F2	F1	F0	B
E0	E7	E6	E5	E4	E3	E2	E1	E0	ACC
D0	D7	D6	D5	D4	D3	D2	-	D0	PSW
B8	-	-	-	BC	BB	BA	B9	B8	IP
B0	B7	B6	B5	B4	B3	B2	B1	B0	P3
A8	AF	-	-	AC	AB	AA	A9	A8	IE
A0	A7	A6	A5	A4	A3	A2	A1	A0	P2
99	Not bit-addressable								SBUF
98	9F	9E	9D	9C	9B	9A	99	98	SCON
90	97	96	95	94	93	92	91	90	P1
8D	Not bit-addressable								TH1
8C	Not bit-addressable								TH0
8B	Not bit-addressable								TL1
8A	Not bit-addressable								TL0
89	Not bit-addressable								TMOD
88	8F	8E	8D	8C	8B	8A	89	88	TCON
87	Not bit-addressable								PCON
83	Not bit-addressable								DPH
82	Not bit-addressable								DPL
81	Not bit-addressable								SP
80	87	86	85	84	83	82	81	80	P0

0FFFH

0000H

Internal ROM

Program Counter

On-Chip Data Memory Organisation

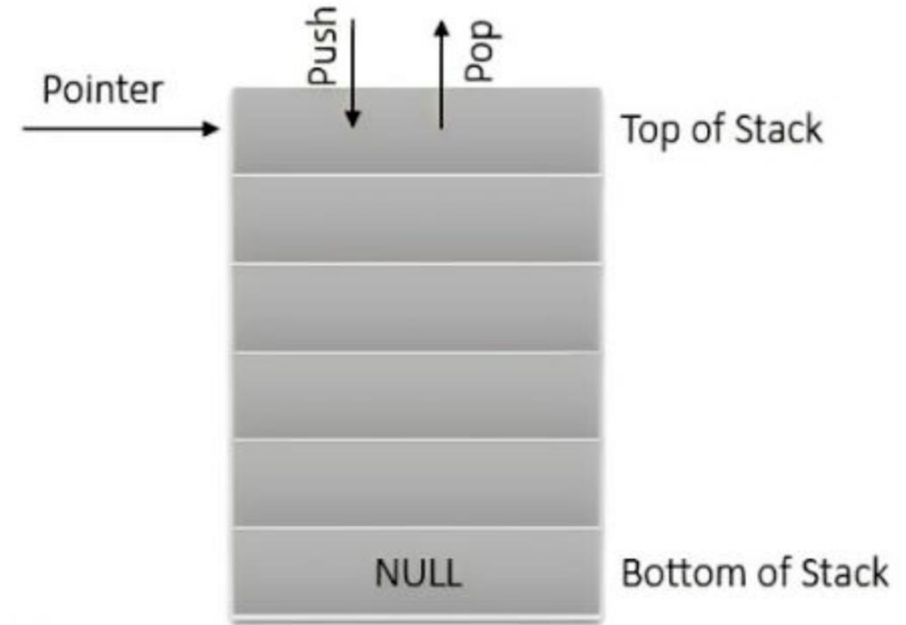
The **right side registers control or access the memory areas shown on the left side**.

Left Diagram	**Right Diagram**
Internal RAM	Control Registers (SFRs)
Stores data	Controls operations
Stack data stored here	SP register located here
Register banks exist here	PSW selects banks
Addresses 00H–7FH	Addresses 80H–FFH

Stack in 8051

Reasons for stack implementation in 8051:

- When calling a subroutine or serving an interrupt, it is necessary to preserve the return address.
- Also, often, we need to preserve the contents of any relevant registers



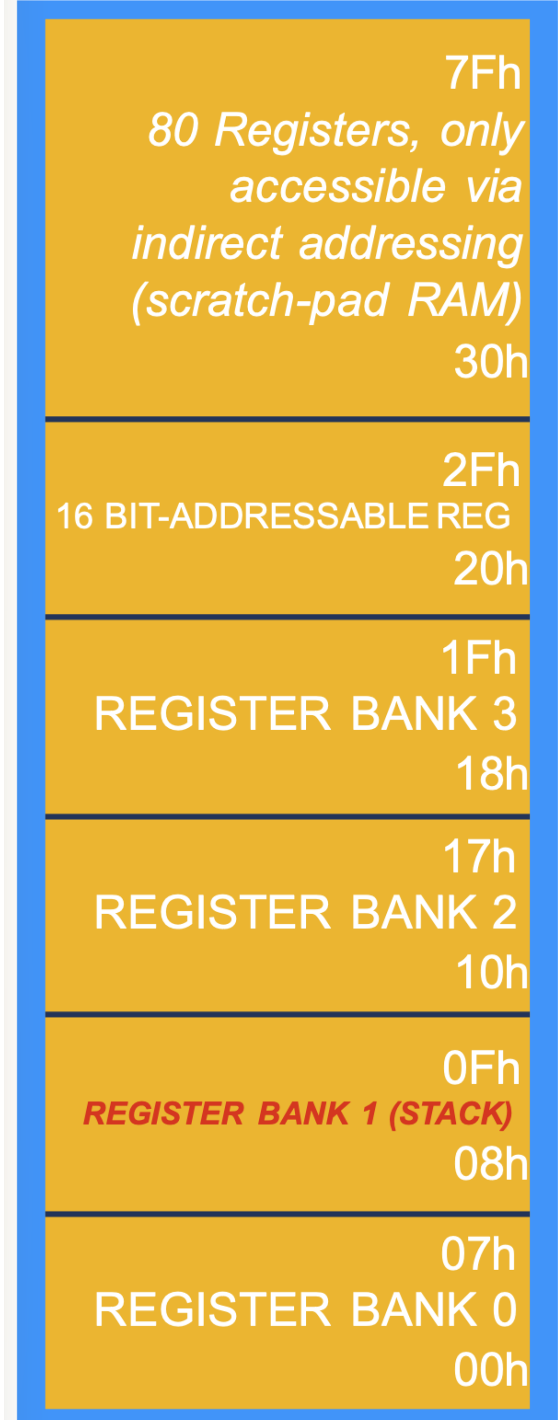
As in general microprocessor programming, the 8051 stack is a special area of data memory for temporary storage.

- It is also a LIFO (Last In, First Out) structure.
- A special Stack Pointer (SP) register is used to store the address of the top of the stack in the 8051.

Stack in 8051

The Stack is implemented using a register bank in the 8051, as shown here.

- Some programming processes in 8051 are also implemented with a stack mechanism, e.g. subroutine calling, interrupt systems, etc.
- By default stack is stored in the register, and it can be customised in the user program to be stored somewhere else.
- Useful in 8051 programming for working with a small set of data to be stored.



Stack in 8051

Stack Pointer (SP) is a register memory-mapped to location 81H.

- Pushing to the stack increments the SP *before* writing data.
- Popping from the stack reads data and then *decrements* the SP.
- The 8051 stack is kept in internal RAM, and is limited to addresses accessible by indirect addressing (i.e. the first 128/256 bytes).
- It is possible to relocate the stack by changing the value of the Stack Pointer.
- The system uses the stack to manage program flow, both user (CALL) and interrupts.

83	**Not bit-addressable**								**DPH**
82	**Not bit-addressable**								**DPL**
81	**Not bit-addressable**								**SP**
80	**87**	**86**	**85**	**84**	**83**	**82**	**81**	**80**	**P0**

AR

Stack in 8051

The reset value of the SP is 07H, which is just after the first register bank Bank 1.

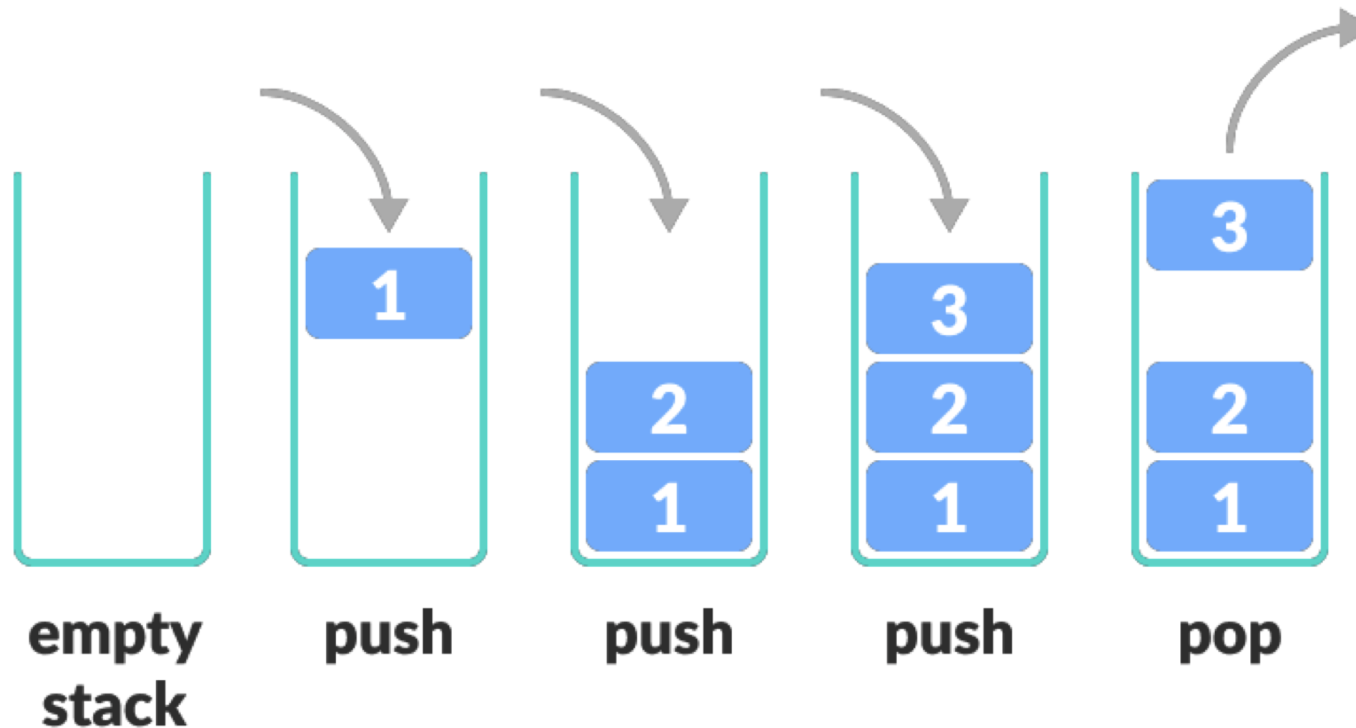
- The maximum available memory for the 8051 stack is 128 bytes.
- This is not a lot, so we need to be careful that we do not run out of memory and cause an overflow.
- Therefore, we must avoid recursive-type programs.

Byte Address	Bit Address							
7F	General Purpose RAM							
30								
2F	7F	7E	7D	7C	7B	7A	79	78
2E	77	76	75	74	73	72	71	70
2D	6F	6E	6D	6C	6B	6A	69	68
2C	67	66	65	64	63	62	61	60
2B	5F	5E	5D	5C	5B	5A	59	58
2A	57	56	55	54	53	52	51	50
29	4F	4E	4D	4C	4B	4A	49	48
28	47	46	45	44	43	42	41	40
27	3F	3E	3D	3C	3B	3A	39	38
26	37	36	35	34	33	32	31	30
25	2F	2E	2D	2C	2B	2A	29	28
24	27	26	25	24	23	22	21	20
23	1F	1E	1D	1C	1B	1A	19	18
22	17	16	15	14	13	12	11	10
21	0F	0E	0D	0C	0B	0A	09	08
20	07	06	05	04	03	02	01	00
1F	Bank 3							
18								
17	Bank 2							
10								
0F	Bank 1							
08								
07	Default Register Bank for R0 – R7							
00								

Stack Mechanism

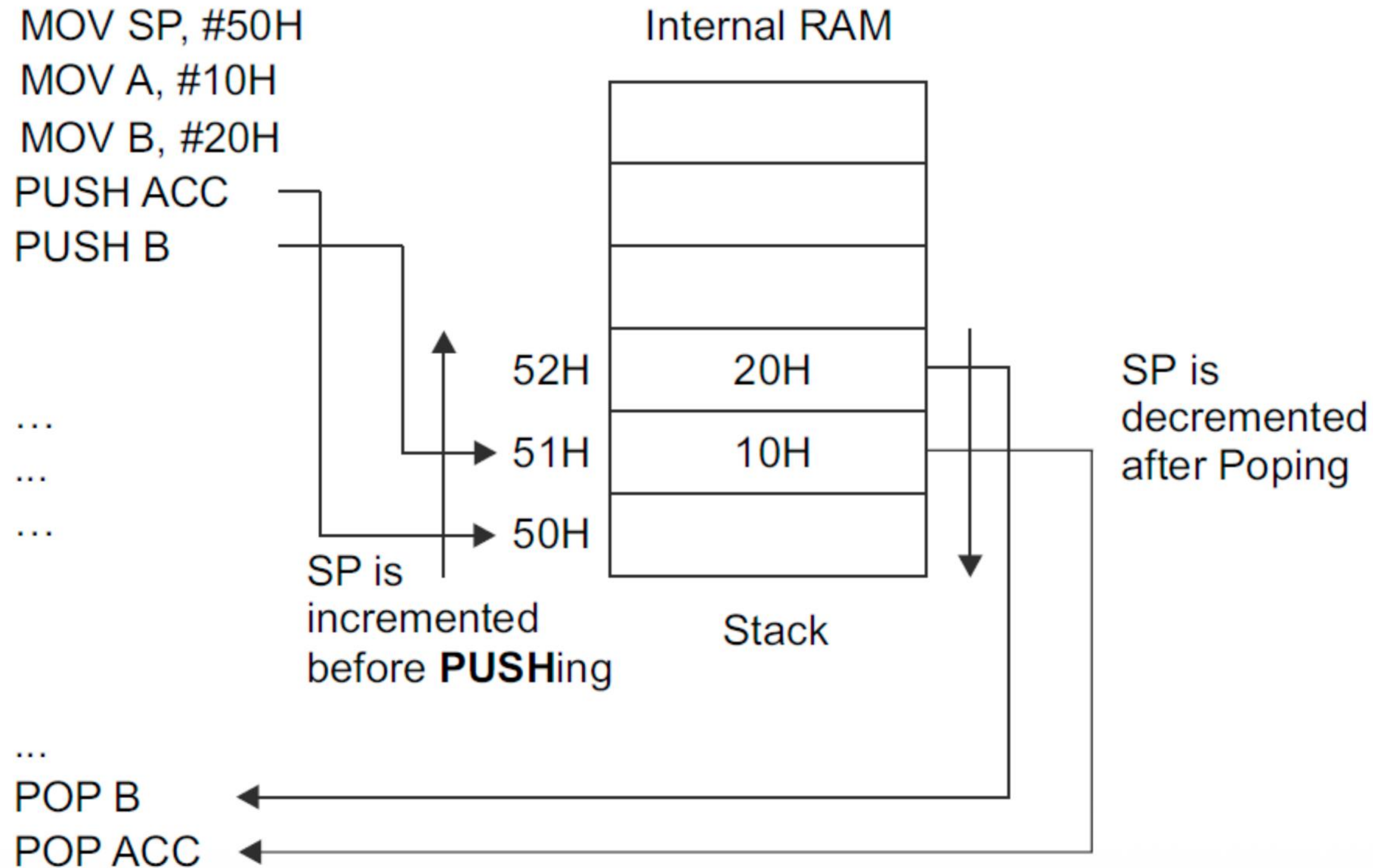
PUSH and POP are special instructions associated with the stack.

- PUSH – Insert the content into the stack.
- POP – Retrieve the content from the stack.



Stack Mechanism

This is how the contents of the Accumulator and B registers can be stored and retrieved from the stack



Stack Mechanism

With a call to a subroutine, for example, “ACALL”.

```
ACALL MyProgram
..
MyProgram:
..
RET
```

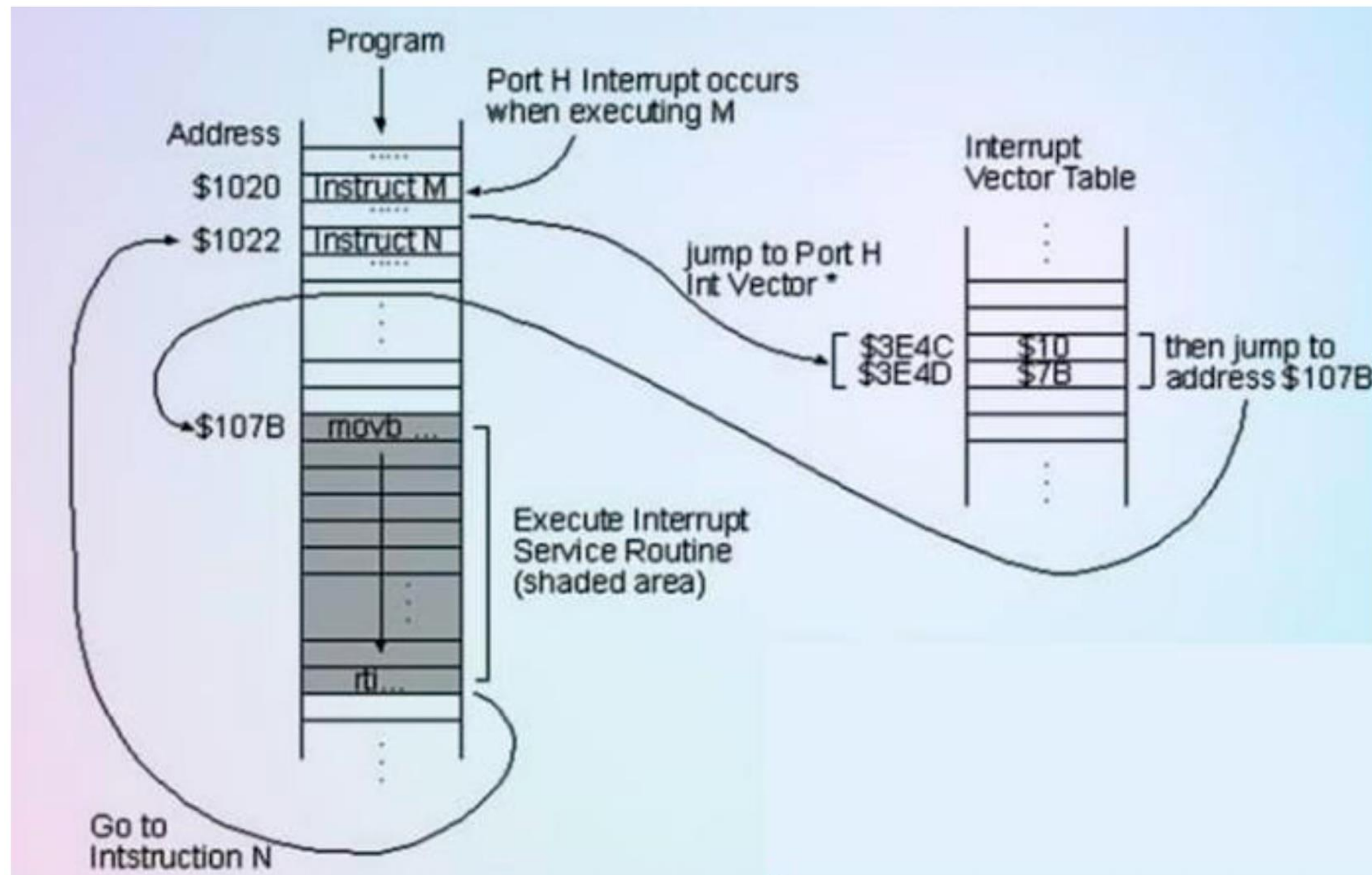
The operation will increase the PC by 2.

- Then, it pushes the 16-bit PC value onto the stack (low-order bytes first) and increments the stack pointer twice.
- At the end of the subroutine, the **RET** instruction pops the high byte and low byte address of the PC from the stack and decrements the SP by 2.
- The execution of the instruction will result in the program resuming from the location just after the “CALL” instructions.

Stack Mechanism

A similar procedure occurs with an interrupt.

- The user can also use the stack for temporary storage, which is often a way to pass variables to and from subroutines.



Example 1: Push and Pop to/from Stack

Write a code that moves values 25H and 45H to registers 1 and 4, and stores and retrieves them to the stack, respectively.

```
MOV R1, #25H     ;assign R1 with value 25H
MOV R4, #45H     ;assign R4 with value 45H
PUSH R1                  ;Store R1 to stack
PUSH R4                  ;Store R4 to stack
POP R3           ;Retrieve value 45H from stack
```

Example 1: Push and Pop to/from Stack

```
MOV R1, #25H     ;assign R1 with value 25H
MOV R4, #45H     ;assign R4 with value 45H
PUSH R1          ;Store R1 to stack
PUSH R4          ;Store R4 to stack
POP R3           ;Retrieve value 45H from stack
```

Assume default:

SP = 07H

So the first PUSH stores data at **08H**.

Example 1: Push and Pop to/from Stack

```
MOV R1, #25H        ;assign R1 with value 25H
MOV R4, #45H        ;assign R4 with value 45H
PUSH R1             ;Store R1 to stack
PUSH R4             ;Store R4 to stack
POP R3              ;Retrieve value 45H from stack
```

Step 1

`MOV R1, #25H`

Store value `25H` into register `R1`.

Suppose Bank 0 is active:

`R1 = RAM[01H]`

Now:

`R1 = 25H`

Step 2

`MOV R4, #45H`

Store `45H` into `R4`.

Bank 0:

`R4 = RAM[04H]`

Now:

`R4 = 45H`

Step 3 — PUSH R1

`PUSH R1`

What CPU does internally:

Before PUSH

`SP = 07H`

`R1 = 25H`

CPU increments SP first

`SP = 08H`

CPU stores R1 value into RAM[08H]

RAM[08H] = 25H

Memory now:

Address	Value
08H	25H

Example 1: Push and Pop to/from Stack

```
MOV R1, #25H        ;assign R1 with value 25H
MOV R4, #45H        ;assign R4 with value 45H
PUSH R1             ;Store R1 to stack
PUSH R4             ;Store R4 to stack
POP R3              ;Retrieve value 45H from stack
```

Step 4 — PUSH R4

`PUSH R4`

Before PUSH

`SP = 08H`
`R4 = 45H`

CPU increments SP

`SP = 09H`

Store R4 into RAM[09H]

`RAM[09H] = 45H`

Memory now:

Address	Value
09H	45H
08H	25H

Step 5 — POP R3

`POP R3`

POP takes data from the TOP of stack.

Before POP

`SP = 09H`
`RAM[09H] = 45H`

CPU copies RAM[09H] into R3

`R3 = 45H`

CPU decreases SP

`SP = 08H`

Final Values

Register	Value
R1	25H
R4	45H
R3	45H
SP	08H

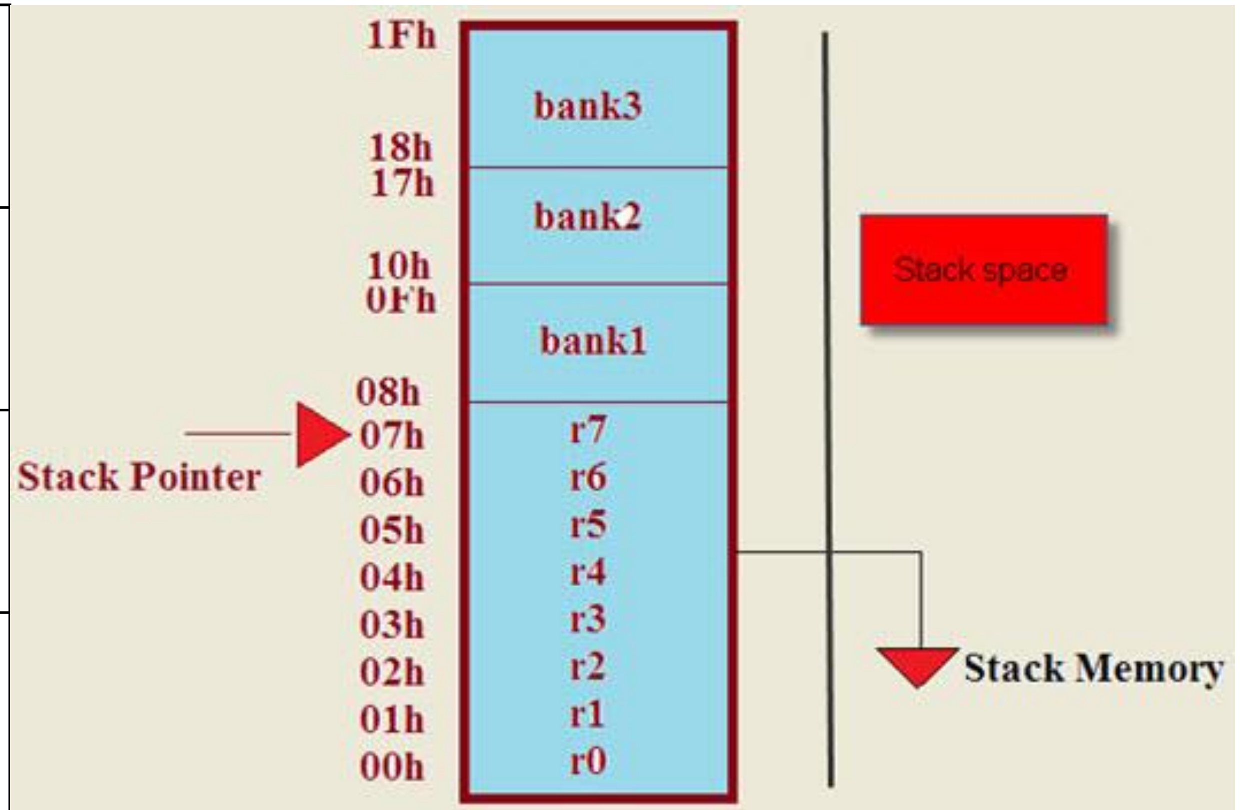
Custom 8051 Stack

The 8051's CPU features an 8-bit stack pointer.
By default, the stack pointer points to address 0x07.
The stack pointer increments by 1 (i.e. counting up).
This is the address immediately below the stack.
Register Bank 1, R0 (address 0x08) is therefore the default start of the stack.

Address	Region
1F	Register Bank 3
18	
17	Register Bank 2
10	
0F	Register Bank 1
08	
07	Default Register Bank for R0-R7
00	

Custom 8051 Stack

1Fh	Register Bank 3
18h	
17h	Register Bank 2
10h	
0Fh	Register Bank 1
08h	
07h	Default Register Bank 0
00h	



Custom 8051 Stack

Note that Register Bank 1 and the stack share the same space.

- If we need to use Bank 1, we can relocate the stack.

We can specify that the stack pointer is stored in a register bank other than the default register bank.

Other memory locations are also permissible, as shown in the example below; SP is initially stored at memory location 0x30H.

```
;Load 0x2FH to stack pointer
;Stack now starts at 0x30H
MOV SP,#2FH
```

Instruction with Stack

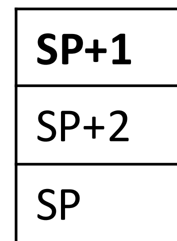
Some instructions that use the stack:

- PUSH:

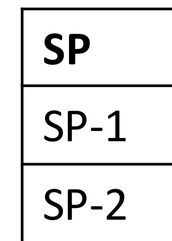
    PUSH addr ; 2 cycles, increments stack pointer (SP) by 1 and then moves addr to the address within SP.

- POP:

    POP addr ; 2 cycles. First, addr is loaded with the value pointed to by SP. Then, SP is decremented by 1.



PUSH



POP

Instruction with Stack: Push

How PUSH Works

Syntax:

PUSH source

Steps:

1. Increment SP

SP = SP + 1

2. Store data into stack location

Example:

```
MOV SP,#2FH
MOV R1,#25H
```

PUSH always:

```
Increment SP first
Then store data
```

PUSH 1

Before PUSH:

SP = 2FH

R1 = 25H

During PUSH:

Step 1 - increment

SP = 30H

Step 2

Store 25H into RAM location 30H

Address	Value
30H	25H

Instruction with Stack: Pop

How POP Works

Syntax:

POP destination

Steps:

1. Read data from the current SP location

2. Decrement SP

Example:

```
POP 3 //remove R3
```

Suppose:

SP = 30H

[30H] = 25H

Step 1:

R3 = 25H

Step 2 - decrement

SP = 2FH

POP always:

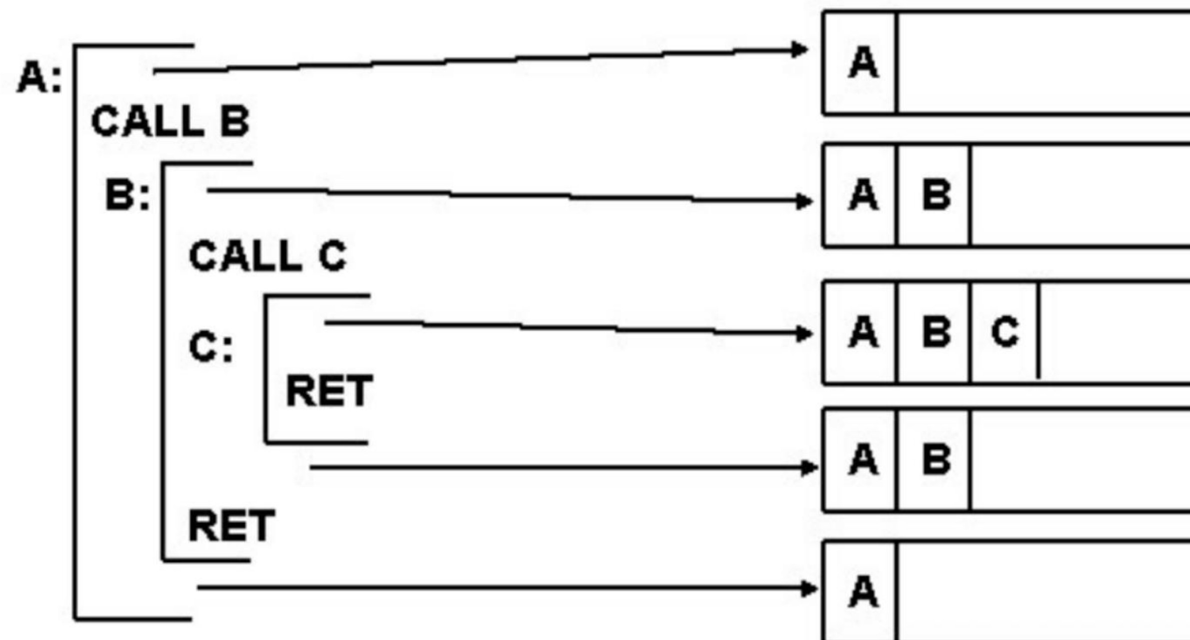
```
Read data first
Then decrement SP
```

Instruction with Stack

Some instructions that use the stack (cont.):

• ACALL (discussed in prev. slides): address of line following the ACALL is stored to the stack (2 bytes).

• RET: return from subroutine. The two-byte address stored in ACALL is popped from the top of the stack (MSB, then LSB) into PC.



Example 2: Stack with Customised Register

Create an assembly language program that initiates the stack at 0x30H. Then, it stores the value 25H and 45H to register R1 and R4, respectively.

Store the content of R1 and R4 to stack consecutively and then retrieve the content of the stack to R3 and R6, respectively. Comment in your program on the value of the stack pointer and the content of the registers throughout the flow of your program.

```
MOV SP, #2FH        ; Load0x2FH to stack pointer
                    ; Stack now starts at 0x30H
MOV R1, #25h        ; copy value 25H to register R1
MOV R4, 25H         ; copy value 45H to register R4
PUSH 1              ; increments SP to 08H and store R1 to stack 30H
PUSH 4              ; increments SP to 09H and store R4 to stack 31H
POP 3               ; Top of stack (at 31H) to R3, decrement SP to 30H
POP 6               ; Top of stack (at 30H) to R6, decrement SP to 2FH
```

Example 2: Stack with Customised Register

1. Set the Stack Pointer

`MOV SP, #2FH`

This loads `2FH` into the Stack Pointer.

So:

`SP = 2FH`

In 8051, when you use `PUSH`, the SP is:

1.incremented first

2.Then the data is stored

So the first stack location becomes:

`2FH + 1 = 30H`

2. Put values into registers

`MOV R1, #25H`

Now:

`R1 = 25H`

`MOV R4, #45H`

Now:

`R4 = 45H`

Example 2: Stack with Customised Register

3. PUSH R1 onto stack

`PUSH 1`

`1` means register `R1`.

What happens internally?

Current SP:

`SP = 2FH`

8051 first increments SP:

`SP = 30H`

Then stores R1 value (`25H`) into RAM location `30H`.

So:

Address	Value
30H	25H

Now:

`SP = 30H`

Example 2: Stack with Customised Register

3. PUSH R1 onto stack

`PUSH 1`

`1` means register `R1`.

What happens internally?

Current SP:

`SP = 2FH`

8051 first increments SP:

`SP = 30H`

Then stores R1 value (`25H`) into RAM location `30H`.

So:

Address	Value
30H	25H

Now:

`SP = 30H`

Example 2: Stack with Customised Register

4. PUSH R4 onto stack

```
PUSH 4
```

Pushes R4 onto stack.

Current SP:

```
SP = 30H
```

Increment first:

```
SP = 31H
```

Store R4 value (`45H`) into RAM location `31H`.

Now memory:

Address	Value
30H	25H
31H	45H

Now:
SP = 31H

Example 2: Stack with Customised Register

5. POP into R3

`POP 3`

`3` means register `R3`.

POP works opposite of PUSH:

1.read from current SP location

2.decrement SP

Current SP:

`SP = 31H`

Value at `31H` is:

`45H`

So:

`R3 = 45H`

Then SP decreases:

`SP = 30H`

6. POP into R6

`POP 6`

Current SP:

`SP = 30H`

Value at `30H` is:

`25H`

So:

`R6 = 25H`

Then:

`SP = 2FH`

Register	Value
R1	25H
R4	45H
R3	45H
R6	25H