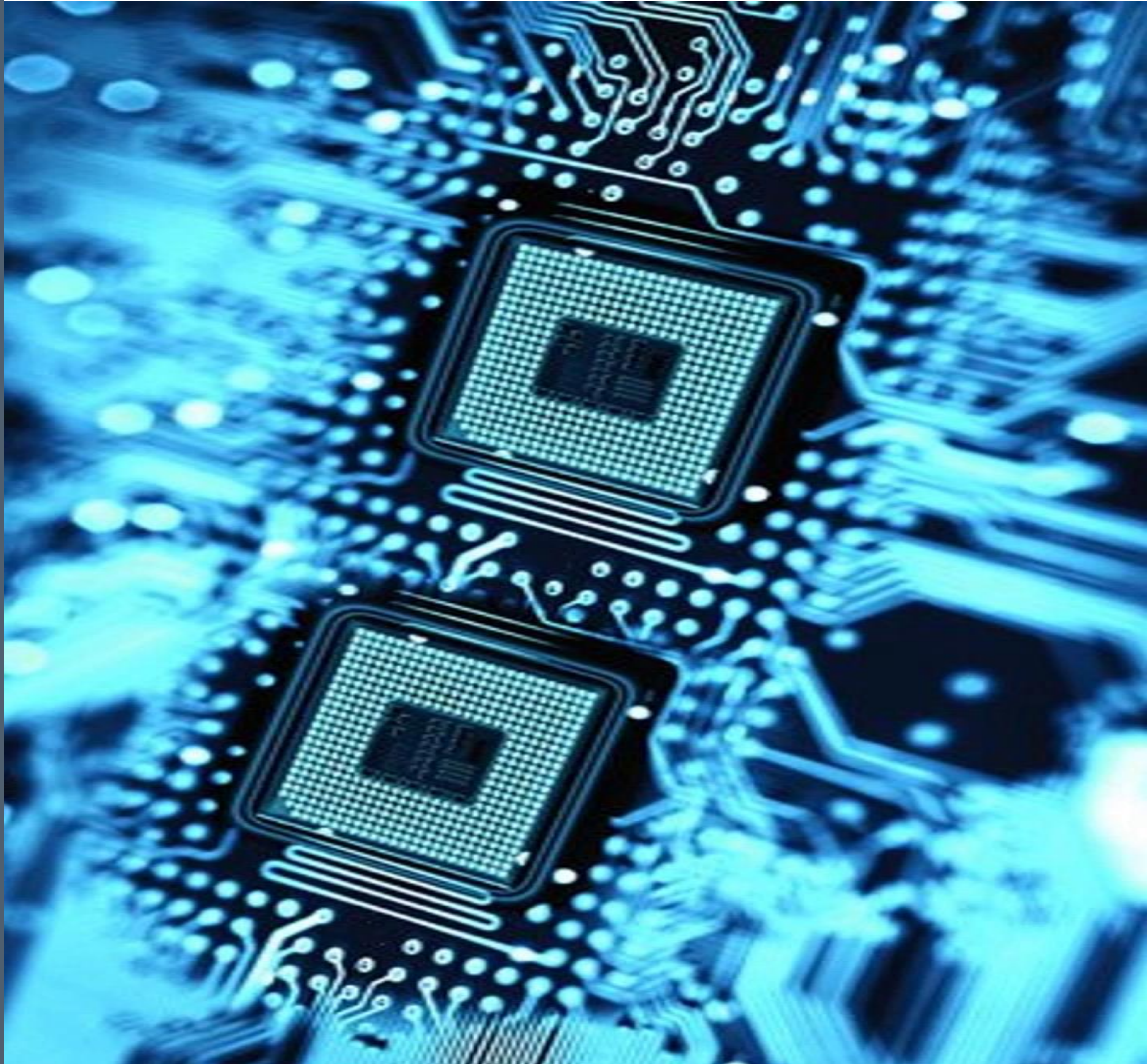
Computer organization & architecture

Course by: Dr. Ahmed Sadek

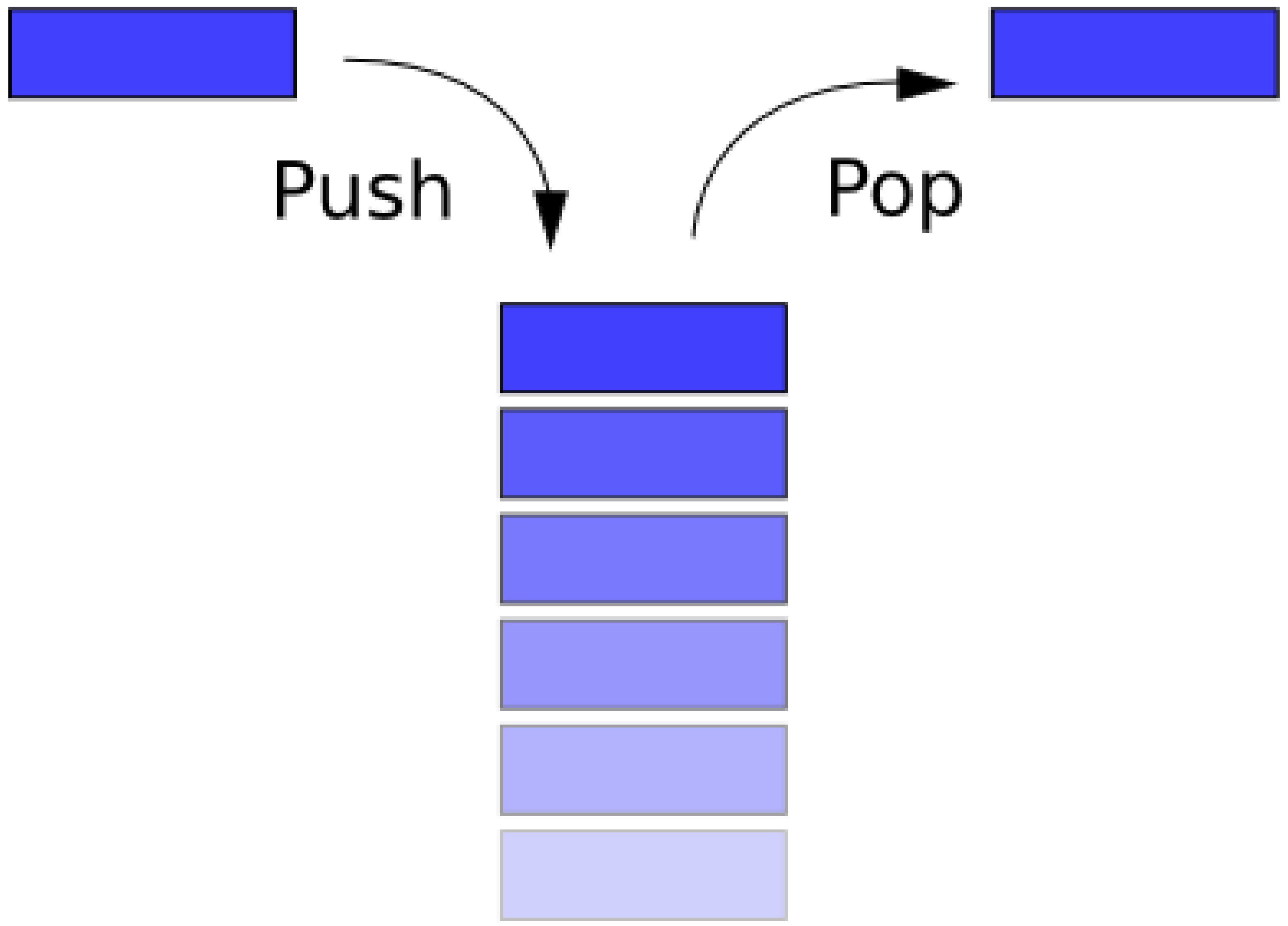
Lab By: Mahmoud Badry

Interrupts - Procedures

Chapter 5

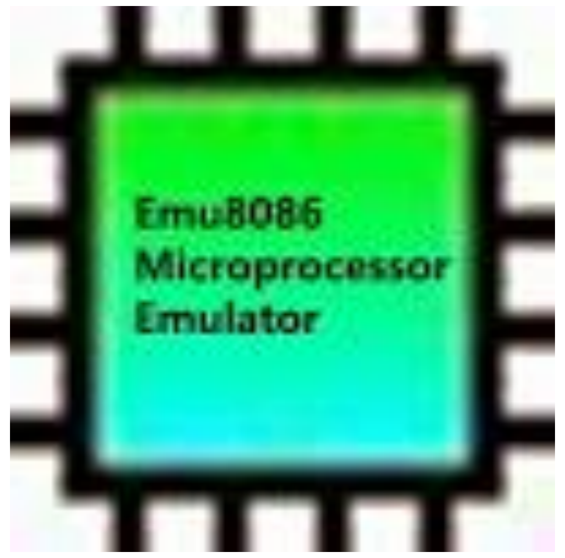


What we need to learn more?

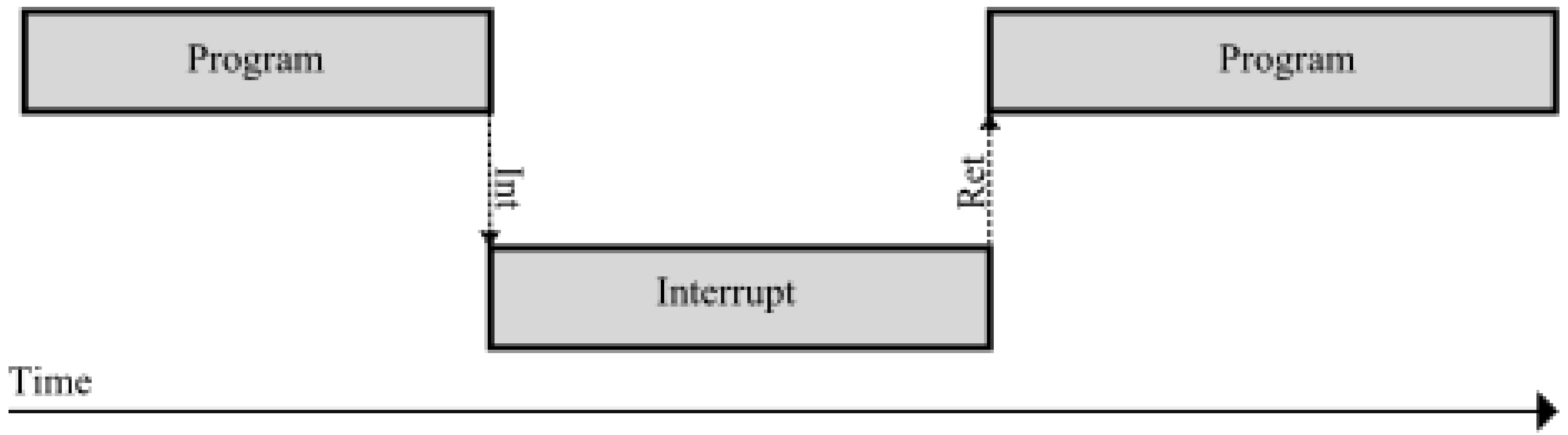


- **Learn** how to do **input-output** in **assembly** language.
- **learn** about the **runtime stack***,* how it is the fundamental mechanism that makes it possible to **call** and **return** from **functions** (we call them **procedures***).*
- Begin **logically dividing** your **programs** into **procedures**.

Interrupts



Interrupts

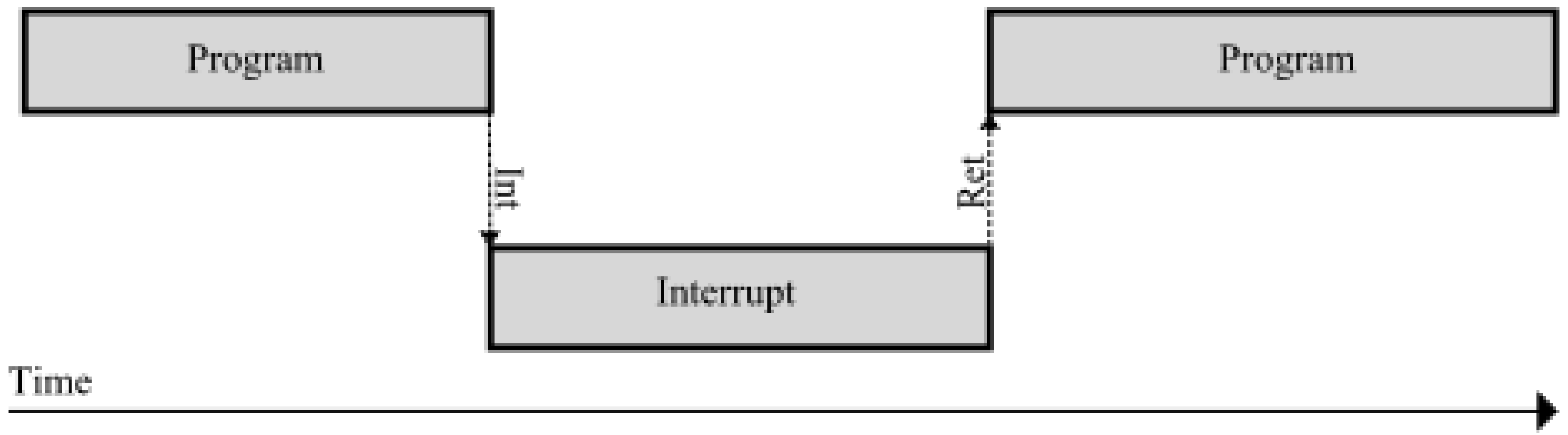


- **Interrupts** can be seen as a **number** of **functions**. These **functions** make the **programming** much **easier**, instead of writing a code to print a character you can simply **call** the **interrupt** and it will **do everything** for you.
- **Interrupts** are also triggered by **different hardware**, these are called **hardware interrupts**. Currently we are interested in **software interrupts** only. Most of them are used for I/O operations.
- To make a **software interrupt** there is an **INT** instruction, it has very simple **syntax**:

```
INT Value
```

Where **value** can be a **number** between **0** to **0FFh**.

Interrupts



- You may **think** that **there** are **only 256 functions**, but that is **not correct**. Each **interrupt** may have **sub-functions**. To specify a **sub-function AH** register should be **set** before **calling interrupt**.
- An example interrupt:

  INT 10h / AH = 0Eh

  Input: **AL** = character to write
- **Simple example** on **INT 10h/ AH=0Eh**

```
mov al, 'a'
mov ah, 0eh
int 10h    ;Print a on screen
```

Interrupts

- INT 20h **Exits the program.**

- INT 21h / AH=2Ah - get system date;
  - return: **CX = year** (1980-2099). **DH = month**. **DL = day**. **AL = day of week** (00h=Sunday)

- All **set** of **interrupts functions** can be found on this **PDF**: **“8068 interrupts.pdf”**

emu8086.inc

Library of common functions

Library of common functions - emu8086.inc

- To make **programming easier** there are some **common functions** that can be **included** in your program. To make your **program use functions defined** in **other file** you should **use** the **INCLUDE** directive followed by a **file name**.

```
INCLUDE emu8086.inc
```

- **Assembler** automatically **searches** for the **file** in the **same folder** where the **source file** is located, and **if** it **cannot find** the file there - it searches in **Inc folder** .
- The emu8086.inc **library** is created by the **emulator developers**. It contains **Macros** and **procedures**.

emu8086.inc Macros

- We **present** some of emu8086.inc **macros**:
  - **PUTC char** - **macro** with **1 parameter**, **prints** out an **ASCII char** at **current cursor position**.
  - **PRINT string** - **macro** with **1 parameter**, **prints** out a **string**.
  - **PRINTN string** - **macro** with **1 parameter**, **prints** out a **string**. The same as PRINT but automatically adds "**carriage return**" at the **end** of the **string**.

emu8086.inc Procedures

- We **present** some of emu8086.inc **Procedures:**
  - PRINT_STRING - **procedure** to **print** a null terminated **string** receives address of string in **DS:SI** register. To use it **declare**: DEFINE_PRINT_STRING before **END** directive.

  - GET_STRING - **procedure** to **get** a null terminated **string** from a user, the received string is written to buffer at **DS:DI**, buffer size should be in **DX**. **Procedure stops** the input **when 'Enter'** is **pressed**. To use it **declare**: DEFINE_GET_STRING before **END** directive.

  - CLEAR_SCREEN - **procedure** to **clear** the **screen**, and set **cursor position** to top of it. To use it **declare**: DEFINE_CLEAR_SCREEN before **END** directive.

emu8086.inc Procedures

- We **present** some of emu8086.inc **Procedures:**
  - **SCAN_NUM** - **procedure** that **gets** the **multi-digit SIGNED number** from the **keyboard**, and **stores** the **result** in **CX** register. To use it **declare**: **DEFINE_SCAN_NUM** before **END** directive.

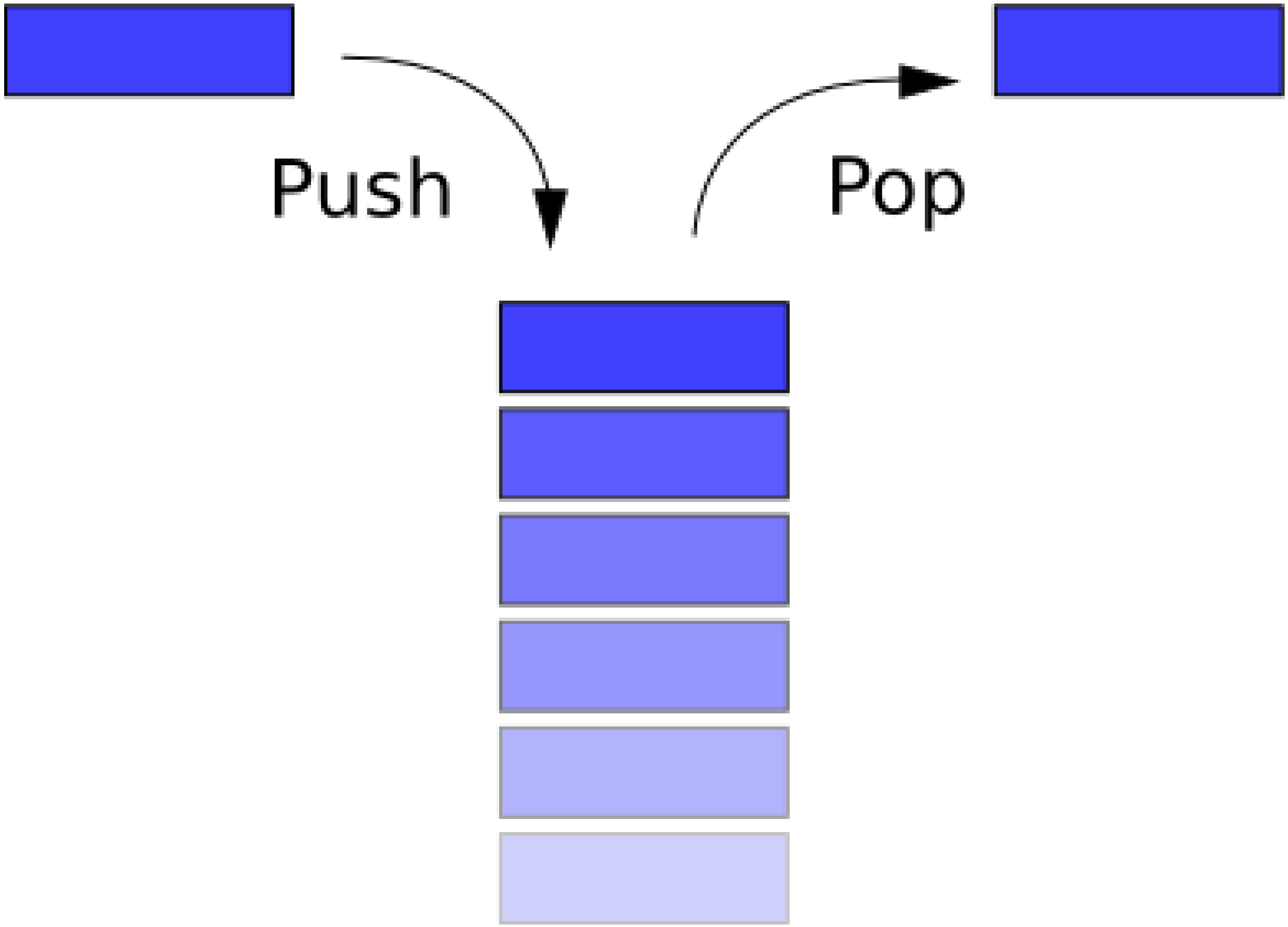
  - **PRINT_NUM** - **procedure** that **prints** a **signed number** in **AX** register. To use it **declare**: **DEFINE_PRINT_NUM** and **DEFINE_PRINT_NUM_UNS** before **END** directive.

  - **PRINT_NUM_UNS** - **procedure** that **prints** out an **unsigned number** in **AX** register. To use it **declare**: **DEFINE_PRINT_NUM_UNS** before **END** directive.

- Now lets review “StringCopy.asm”

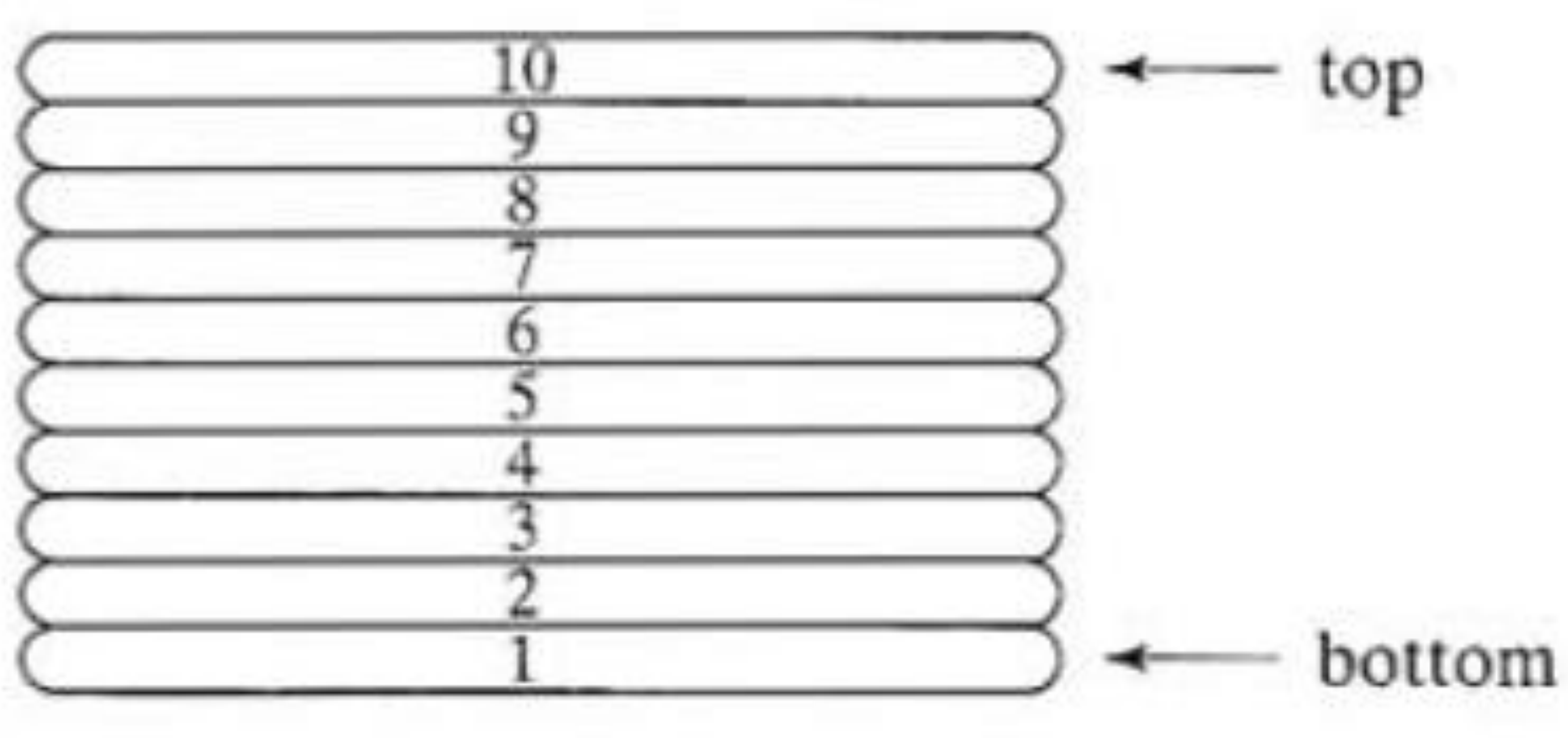
Stack Operations

Section 4

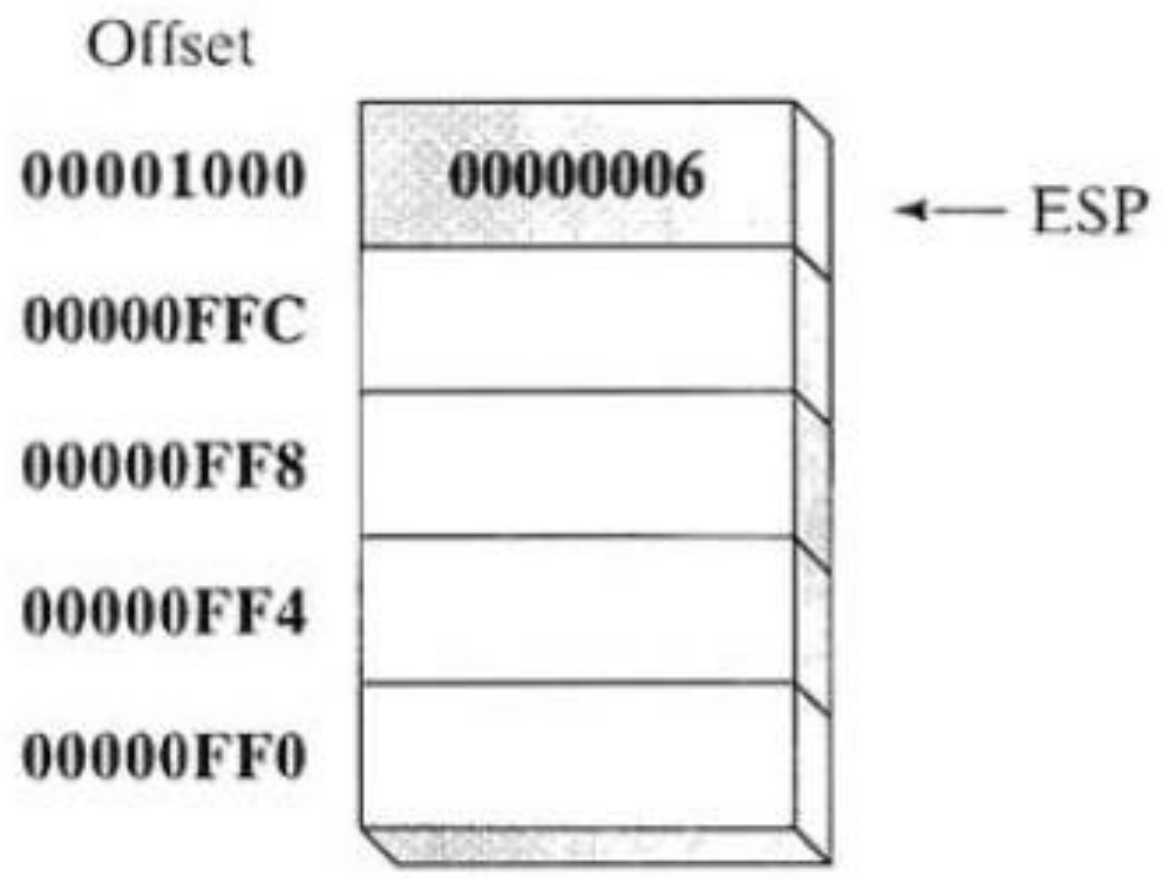


Stack

- A **stack** is called a **LIFO** *structure* (**last-in**, **first-out**). because the last value put into the stack is always the first value taken out.

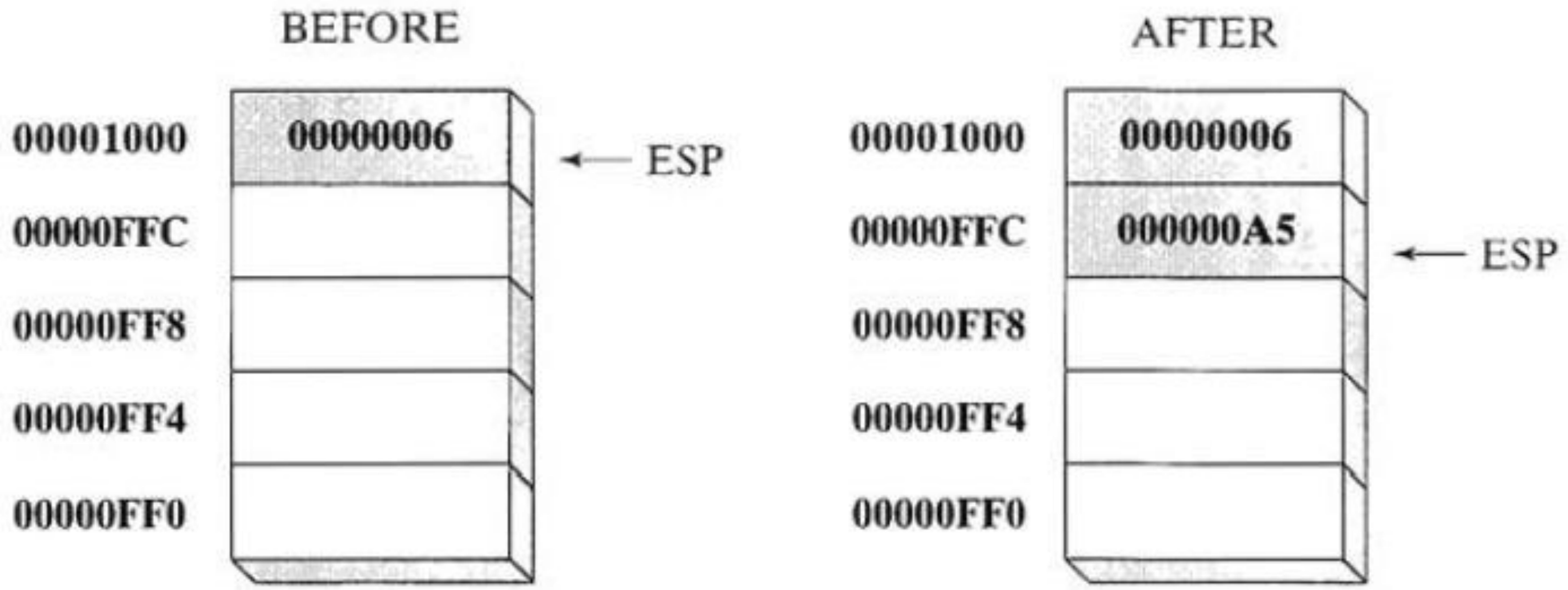


Runtime Stack

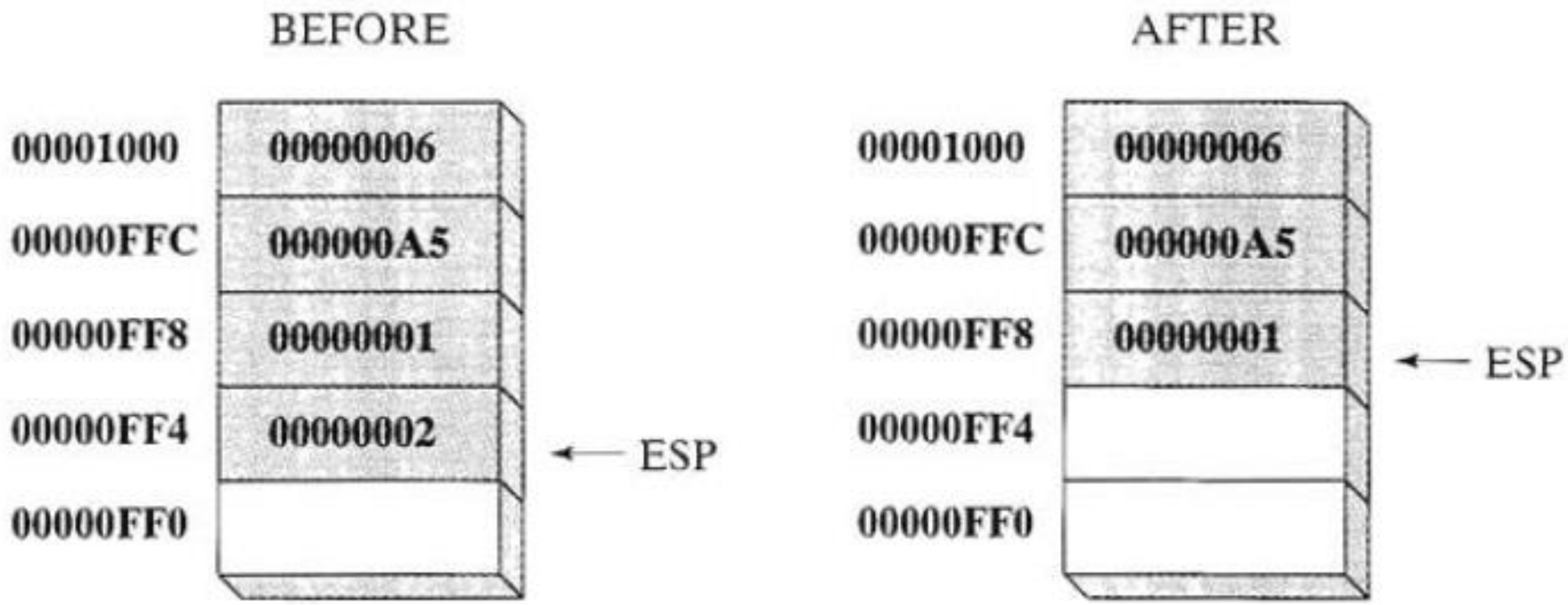


- The **runtime stack** is a **memory array** that is **managed** directly by the **CPU**, using two registers: **SS** and **SP**.
- We **rarely manipulate ESP** directly; instead , it is **indirectly** modified by **instructions** such as **CALL**, **RET** ,**PUSH**, and **POP**.
- The **stack pointer** register (**SP**) points to the **last integer** to be **added** to, or **pushed** on The **stack**.

Push Operation



Pop Operation



BEFORE

00001000	00000006	
00000FFC	000000A5	
00000FF8	00000001	
00000FF4	00000002	← ESP
00000FF0		

AFTER

00001000	00000006	
00000FFC	000000A5	
00000FF8	00000001	← ESP
00000FF4		
00000FF0		

Stack Applications

- There are **several** important **uses** of **stacks** in programs:
  - A **stack** makes a convenient **temporary save area** for **registers** when they are **used** for **more** than **one purpose**. After they are **modified**, they can be **restored** to their **original values**.
  - When the **CALL** instruction **executes**, the **CPU** saves the **current procedure's** return **address** on the **stack**.
  - When **calling** a **procedure**, we often **pass** input values called **arguments** . These can be **pushed** on the **stack**.
  - **Local variables** inside a **procedure** are created on the **stack** and are **discarded** when the **procedure ends**.

Stack instructions

- The **PUSH** instruction first **decrements SP** and then **copies** either a source **operand** into the **stack**.

```
PUSH r/m16
PUSH imm8/16
```

- The **POP** instruction first **copies** the **contents** of the **stack element** pointed to by **SP** into a **destination operand** and then **increments SP**.

```
POP r/m16
```

- The **PUSHF** instruction pushes the **16-bit EFLAGS** register on the stack, and **POPF** pops the stack into **EFLAGS**.

```
PUSHF
POPF
```

- Example:

```
PUSHF    ;save the flags
; any sequence of statements here
POPF    ;restore the flags
```

- The **PUSHA** instruction **pushes** all of the **16-bit general-purpose registers** on the **stack** in the following **order**: AX, CX, DX, BX, SP, BP, SI, and DI. The **POPA** instruction **pops** the same **registers** off the **stack** in **reverse** order.
- Best **usage** are in **procedures**:

```
myProc PROC
pusha
;Make Procedure operations
popa
ENDP
```

Defining and Using Procedures

Section 5

Defining a Procedure

- We can **define** a **procedure** as a **named block** of statements that **ends** in a **return** statement. A **procedure** is **declared** using the **PROC** and **ENDP** directives. It must be assigned a **name** (a **valid identifier**).

```
main PROC
.
.
main ENDP
```

- `ret` statement makes the **procedure returns** to the place it was **called** from.

Documenting Procedures

- A good habit to cultivate is that of adding **clear** and **readable documentation** to your programs. The following are a few **suggestions** for information that you can put at the **beginning** of each **procedure**:
  - A **description** of all tasks accomplished by the **procedure**.
  - A list of **input parameters** and their usage.
  - A description of any values **returned** by the procedure.
  - A list of any **special requirements**, called **preconditions**, that must be **satisfied before** the procedure is **called**.

Call and Return Example

```
• main PROC
  00000020 call MySub
  00000025 mov ax , bx
  …
  00000040 MySub PROC
           mov eax, edx
           ret
           MySub ENDP
```

- When **processor executes** `call MySub`, it **pushes** `00000025` into **stack**, and make **IP** value `00000040.`
- **After ret** statement is **executed** in **MySub**, the processor **pops** the **stack** so **IP** value will `00000025` to go to this address.

Local Labels and Global Labels

- By **default**, a **code label** (followed by a **single colon**) has **local scope**, making it **visible** only to **statements inside** its enclosing **procedure**. This **prevents** you from **jumping** or **looping** to a label **outside** the current **procedure**.
- If you want to **transfer control** to label **outside procedure**, the **label** must be **global**. To make a **global label** use **two colons**.
- Example:

```
GlobalLabel::
```

- Now lets **review** “array sum.asm” and “reverse string.asm”

THANKS