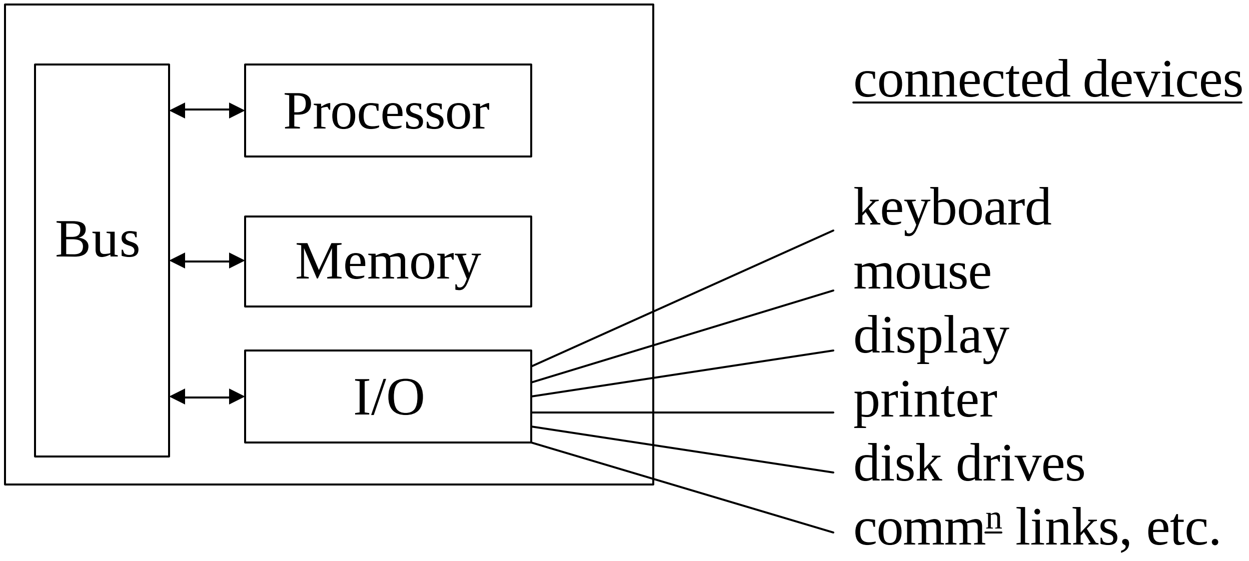
Parallel Input/Output

Basic Concepts of I/O

- **Input/Output** is the information exchange between CPU and (external) connected devices
- Block Diagram of a Simple Computer System



Bus

Processor

Memory

I/O

connected devices

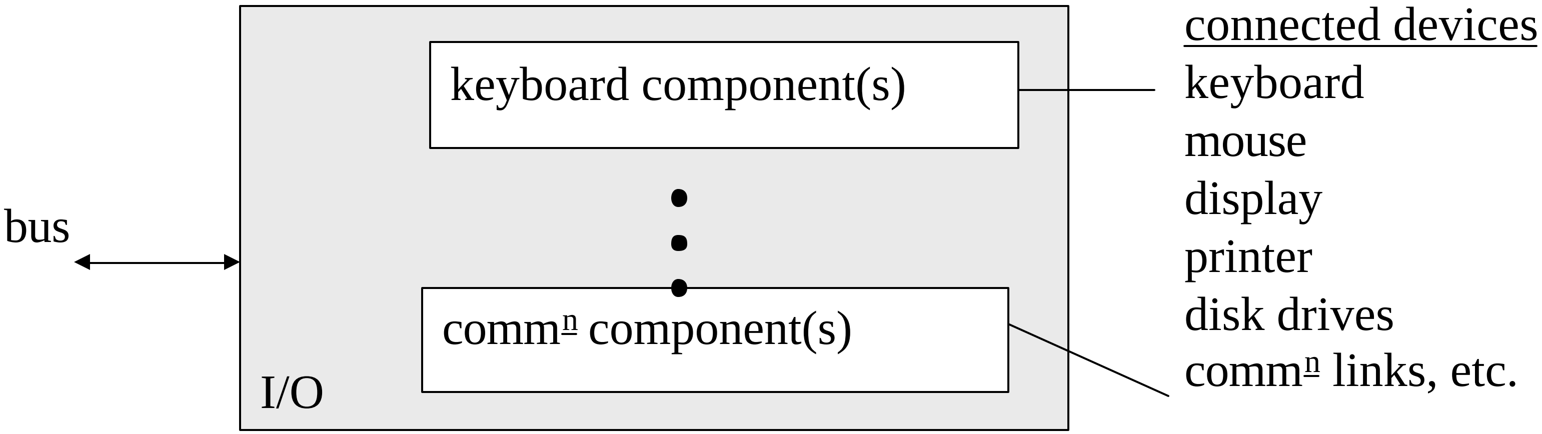
keyboard
mouse
display
printer
disk drives
comm^{n} links, etc.

Basic Concepts of I/O

- Interfacing and programming I/O devices: different from our previous programs
  - Electrical characteristics different from CPUs
    - Analog devices, power, current drive
  - I/O devices operate **asynchronously** from the CPU (and the program being run)
    - Transfer data: processor and I/O device **synchronize** or “handshake” to exchange information

Basic Concepts of I/O

- Independent I/O **components** associated with each connected device.



- I/O components: interfaces that “electrically” connect external device to computer’s internal bus.
  - Bus connection allows CPU to read and/or write device

I/O Ports

- Port: allows exchange of information between bus (connected to CPU and memory) and I/O components (connected to devices)
- 3 kinds of Ports:
  - **Control**: write values to these – control behaviour of component/device
  - **Status**: read values from these – find out about current state of component/device
  - **Data**: read and/or write values of these – exchange application information
- Some ports: read-only, write-only or read&write.
- Ports: often bit-mapped.

I/O Addresses

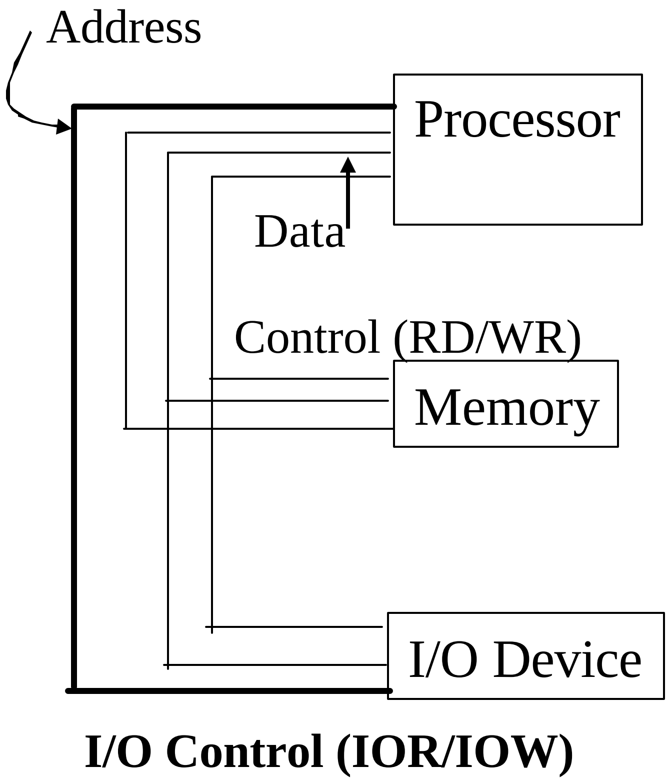
- When connected to a computer system, each port is assigned an I/O address
  - Device (port) identified by its **I/O addresses**
  - CPU read/write from/to I/O address to receive/send data from/to device

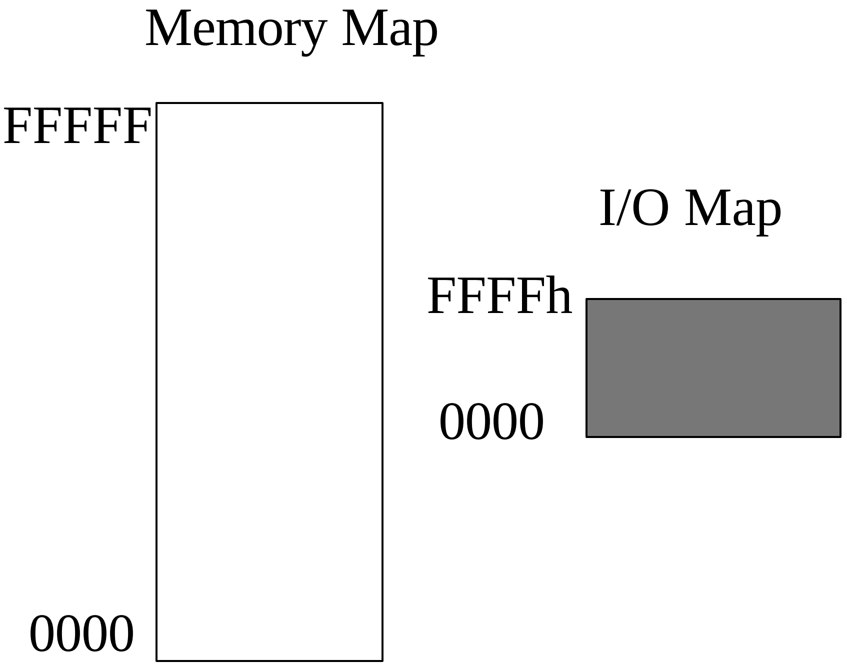
- Microprocessor architectures: two kinds of I/O addresses
  1. Isolated I/O
  2. Memory-Mapped I/O

Concept !

I/O Addressing Schemes : Isolated I/O

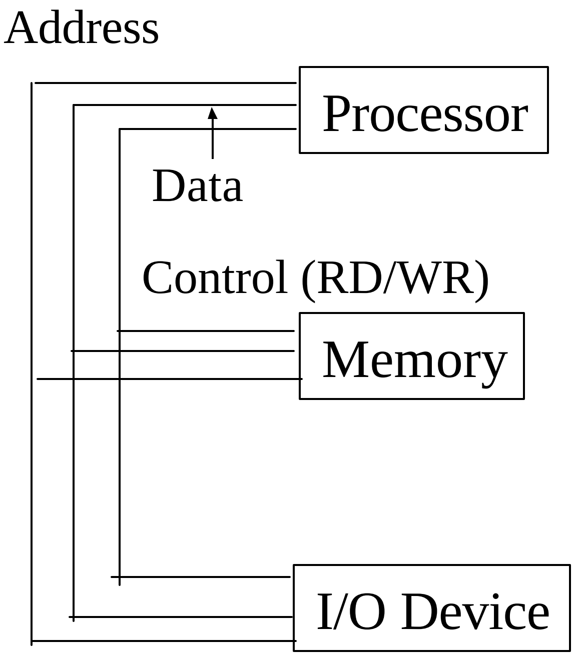
- Microprocessor: **dedicated instructions** for I/O operations.
- Separate address space for I/O devices.

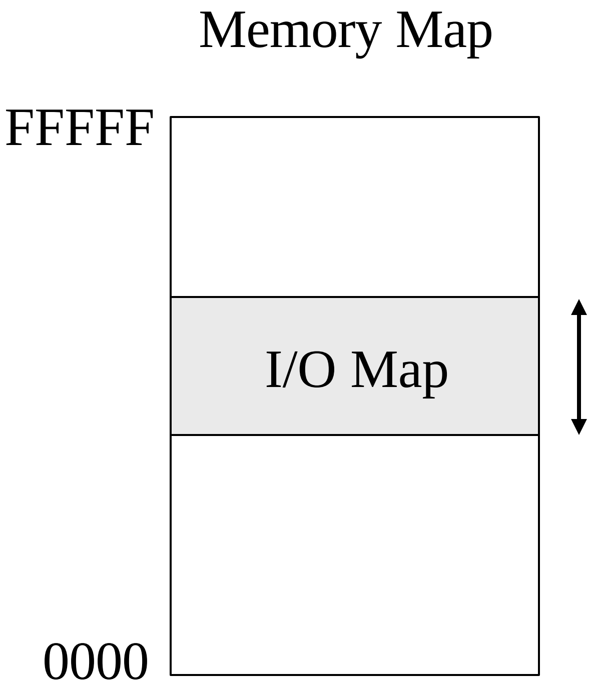




I/O Addressing Scheme : Memory Mapped I/O

- Microprocessor: same instruction set to perform **memory and I/O** operations.
- I/O devices and memory components resident in same memory space.





Intel *Uses* Isolated I/O

Implementation of a Concept

- 80x86 family, I/O addresses range 0-FFFFh
- PC: devices assigned standard I/O addresses (used by all manufacturers of PCs)

– Keyboard	60h
– Speaker	61h
– Parallel Printer (LPT1)	3BCh-3BFh

I/O ports ^{1} memory cells

- Memory transfers :

```
MOV AL, [61h]
```

- I/O transfers :

```
IN AL, 61h
OUT 61h, AL
```

Intel 8086 IN Instruction

Mnemonic : `IN`

Semantics : Read from I/O port

Syntax :

```
IN      AL, imm8
IN      AX, imm8
```

`IN AL, imm8` ← 8-bit read

`IN AX, imm8` ← 16-bit read

Legacy of 8085 which had an 8-bit I/O space

– `imm8`: 8-bit I/O address in the range 00h-FFh

```
IN      AL, DX
IN      AX, DX
```

Addressing Modes are different!

– DX: 16-bit I/O address in the range 0000h-FFFFh

Intel 8086 OUT Instruction

Mnemonic `OUT`

Semantics : Write to I/O port

Syntax :

Destination looks like immediate!

```
    OUT         imm8, AL
```
← 8-bit write

```
    OUT         imm8, AX
```
← 16-bit write

```
    OUT         DX, AL
    OUT         DX, AX
```

I/O Example

- We have a display device for ASCII characters
- Programmer's model: one write-only data port at I/O Address = 04E9H
  - Display "cursor driven": ASCII character written to data port displayed at current cursor position
  - Cursor position maintained by the display device
  - When a character is written, cursor position is "advanced"
    - Advancement handles new lines and scrolling too.

- Write a code fragment showing the display of the character 'A'

I/O Example

Solution : Write a code fragment showing the display of the character ‘A’

```
MOV  DX, 04E9H
MOV  AL, 41h
OUT DX, AL
….
```

This port address is 16 bits. Must load it into DX first (Immediate only for 8-bit port addresses)

A character is a byte

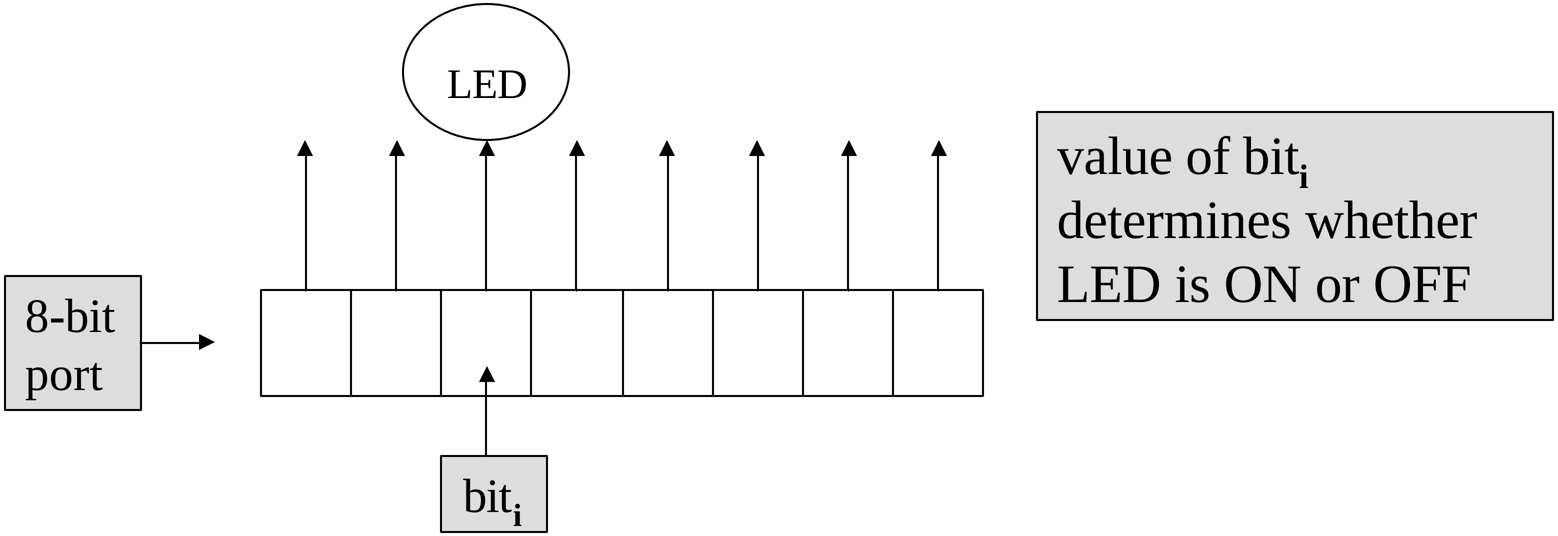
```
Question :
    …
    IN   AL, DX
```

Will AL contain 41H ?

Lab PC’s LED/Switch Box

- Labs: I/O Box attached to PCs
  - 5 LEDs (Light Emitting Diodes) – each either ON or OFF
  - 5 switches – each either ON or OFF

- **LEDs** connected to bits of an 8-bit **output** parallel port
  - Each LED driven by a particular bit in the port



value of bit$_i$ determines whether LED is ON or OFF

Programmer’s Model for the Lab LEDs

- LED data port address: `378 H`
- Bit configuration: LEDS are labelled 1 .. 5
  - [bit 7 = most significant ; bit 0 = least significant]
-

bit	7	6	5	4	3	2	1	0
LED	x	x	x	5	4	3	2	1

  - 1 through 5 indicate bits for LEDs 1 through 5
  - x indicates unused (don’t care what value is written)

- To turn LED ON: set bit associated with the LED
  - i.e.: write 8-bit value to port; bit associated with LED = 1
- To turn LED OFF: clear bit associated with LED

Programming the LEDs

- LED’s interface: 8-bit port.
  - If we want to set/clear a particular bit, we must write an entire byte to the LED port.
  - Writing any value to the LED port affects all LEDs !
  - Modify the state of one LED: must know state of all LEDs, but …
- Reading port is meaningless (write-only port)
  - We cannot read LED port to get the current state of all LEDs.

Programming the LEDs

- To manipulate LEDs individually, program must keep state of LEDs as a variable
  - updated each time a value is written to the LED port

```
LED_State      DB  ?    ; current state of LEDs
; To turn on LED x
   ; set appropriate bit in LED_State
   ; write LED_State to LED port
; To turn off LED x
   ; Clear the appropriate bit in LED_State
   ; Write LED_State to the LED port
```

Lab Switches

- 5 switches on the I/O box connected to 5 bits of 8-bit input parallel port
  - One bit (in port) per switch
  - Read-only port used to get current setting of all switches
    - A write to the port has no effect
  - Switch is ON, its bit is set (i.e. "1")
  - Switch is OFF, its bit is clear (i.e. "0")
  - Switches labelled "A" through "E"

- Switch data port address: `379 H`

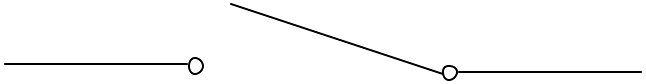
Bit config: [bit 7 = most signif ; bit 0 = least signif]

bit	7	6	5	4	3	2	1	0
Switch	E	D	C	B	A	x	x	x

- x indicates unused (undefined)

Switch de-bouncing

- Switch: mechanical device
  - Moving switch position: opens/closes circuit
  - Switches: metal contacts completing circuit when joined



- Switches in the lab: spring-loaded to hold open/closed position
  - When position change, contacts can bounce
- Program reading switch port, “value” of switch output will oscillate (open/closed) until bouncing stops
  - Program must filter out oscillations so that program only “sees” one switch state change per position change. This is called de-bouncing.

Simple De-Bouncing

- Write a loop that polls the switch until first change is seen
- Waste “enough” time (do-nothing-loop) until sure switch stopped bouncing
- Questions :
  - How much is “enough” time?
  - What if the program waits longer than necessary?
  - What if the program does not wait long enough ?

Adaptive De-Bouncing

```
In a loop, poll the switch until first change is seen
Set a loop counter to an init_value
Repeat {
    Poll switch
    If (switch has changed state again) {
        loop counter = init_value
    } else {
        decrement loop counter
    }
} until loop counter == 0
```

Explain why this approach is “adaptive” ?

Remaining Issue : Decide on the init_value