Hardware Interrupts

The Particular Challenges of I/O Programming

- With “memory” programming :
  1. Data transfers between memory occur as result of instruction **fetch-execute** cycle
     - Time to complete: (order of) microseconds

  – Program runs **synchronously**: instructions fetched then executed.
     - **CPU** controls the data transfers between memory.

The Particular Challenges of I/O Programming

1. Input/output often involve **physical movement** of I/O devices (keypads, sensors, switches)
   - Response times determined by **physical nature of device** (e.g. switches bouncing, A/D conversion, movement of disk head)
   - Response times an **order of magnitude slower** than instruction execution

2. I/O devices operate **asynchronously** from the processor (and the program being run)
   - Availability of data for input OR device for output **not under control of CPU**
   - Data transfer: processor and I/O device must **synchronize** or "handshake"

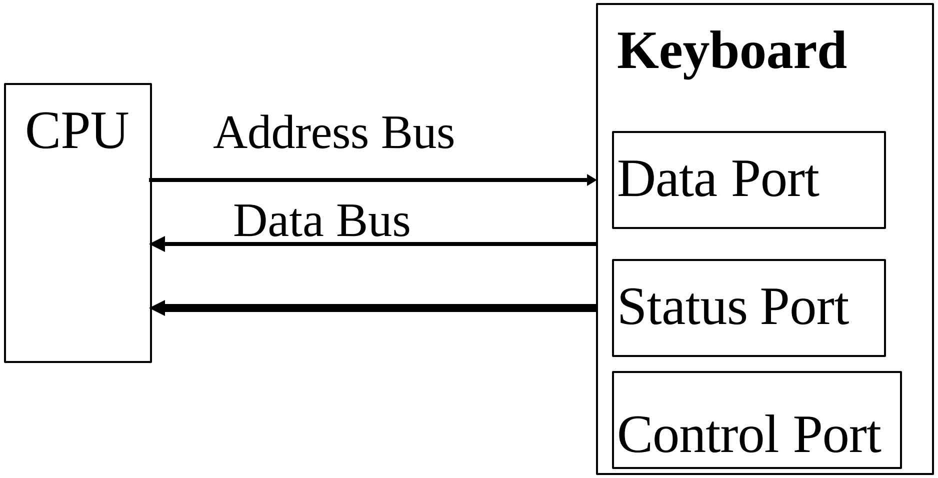
Example : Polled Keyboard

- **Simple Keyboard model**:
  - Key pressed on keyboard
  - ASCII encoded character (associated with key) available in keyboard data port.
  - Bit in status port indicates "Key Ready".
  - Bit in status port cleared when key read from data port

- Pseudo-Code for Polling Keystrokes :

```
LOOP
     UNTIL status port  == KEYSTROKE_ENTERED
Read Keystroke from Data port
```

Example : Polled Keyboard



Example : Polled Timing Loop

- In a program, what is time?

- Example : software-only solution for a timing loop

  - Write a loop that simply counts to waste time (busy waiting)

```
for ( int i = 0; i < 10000; i++ ) {
 for (int j = 0; j < 10000; j++) {
    }  // empty body
}
```

- **Advantage**: simple software; no explicit h/w involved
- **Disadvantage:**
  - Timing based on execution speed of the processor.
  - Not portable: execution speed varies on different machines
    (download old DOS games?)

Polling : CPU-centric Handshake

Is Switch E up?

- Program tests status of device before transferring data

```
 while (!(getSwitches() & 1000 0000B)) ;
              processSwitch()
```

CPU does nothing but wait

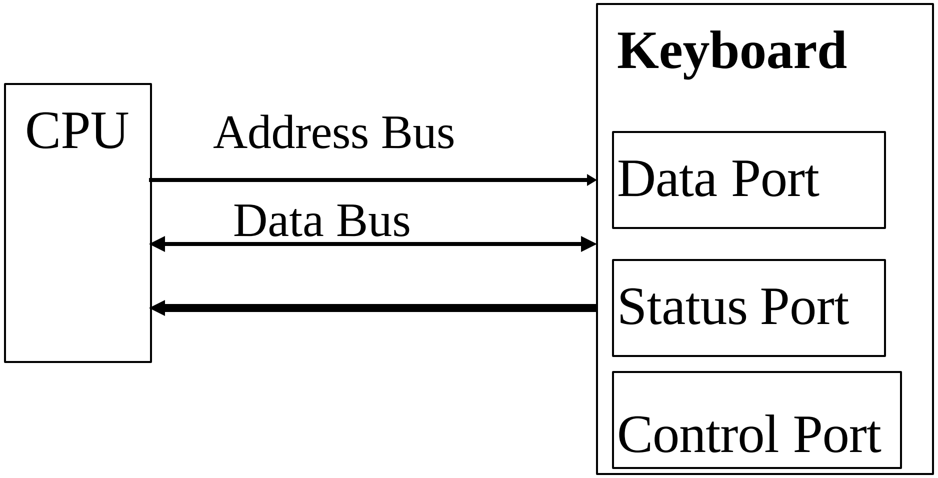
- Or

```
loop
      if (getSwitches() & 1000 0000B)
              processSwitch()
            else
              doOtherWork()
      endif
endLoop
```

CPU does other work "in-between" waiting but may miss instant when Switch E changes

Example : Interrupt-Driven Keyboard

- **Simple Keyboard model**:
  - Key pressed on the keyboard device: ASCII character available in keyboard data port; bit in status port indicate “Key Ready”.
  - Interrupt sent to the CPU.
  - Bit in status port cleared when key read from the data port



•Hardware Interrupts require hardware signal from device to the processor.

Hardware Interrupts

- Hardware Interrupts: require **hardware interrupt mechanism**
  - **hardware signal** (i.e. wired connection to the CPU)
  - I/O component uses signal to inform CPU an event has happened.
    - No busy waiting, no polling.
    - Programming requires an "event-driven" mindset

- **Processor responds** to hardware interrupt; stops current processing and
  - Saves current processor state (CS, IP, FLAGS)
  - Clears IF and TF
  - Decides from which vector table location to load ISR address
  - Executes ISR
  - When ISR executes IRET: processor state restored; execution returns to "interrupted" processing

More later

Learning the Event-Driven Mindset

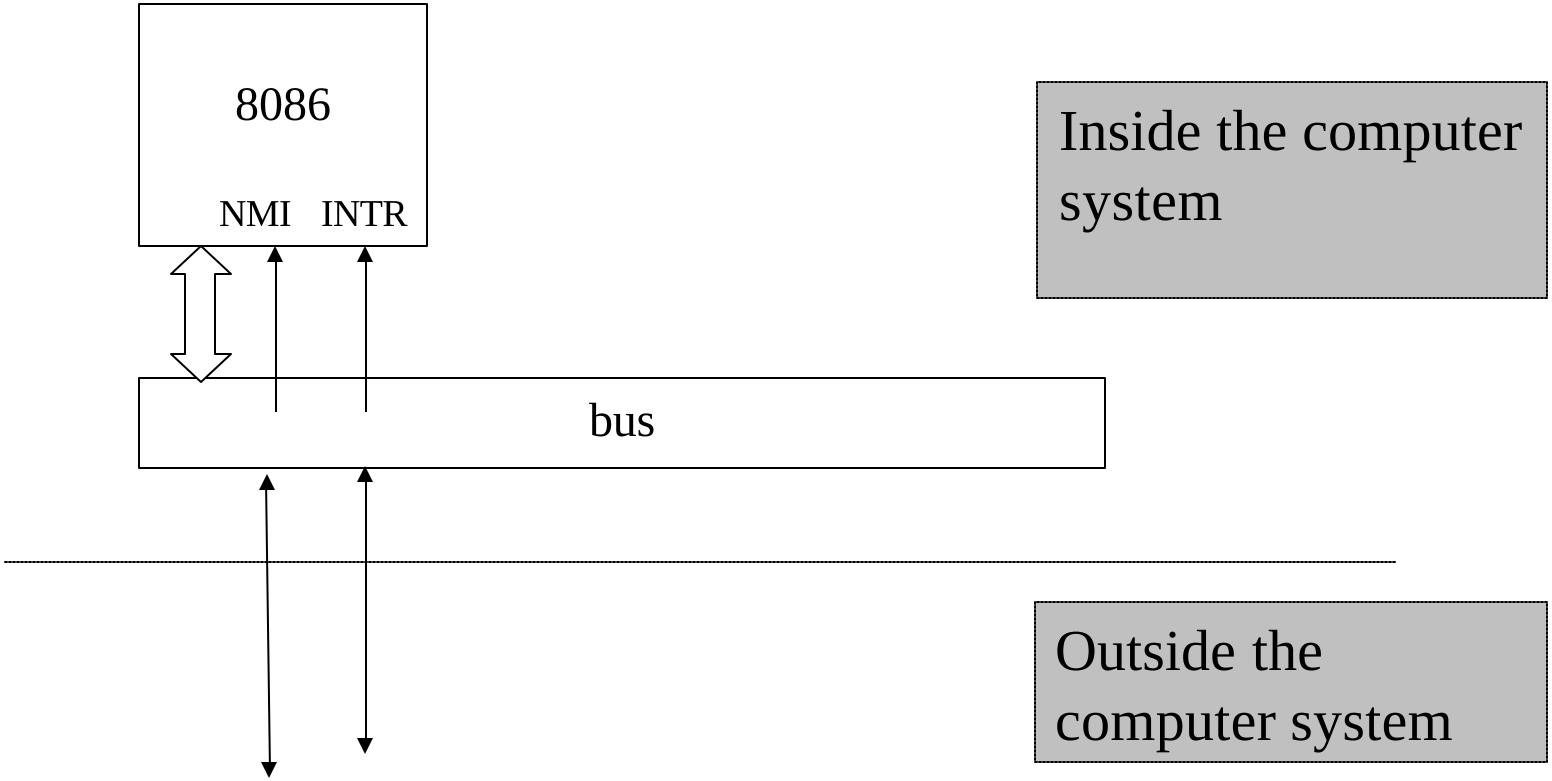
- Interrupted processing: doesn’t “know” it was interrupted
  - Processor:
    1. **temporarily suspended** current thread of control
    2. **ran ISR**
    3. **resumed** suspended thread of control

- Polling: CPU **asks devices whether there is anything to do**.
  - sequential programming: next instruction determined by control transfer instructions.

- Interrupts: **device tells CPU** it is time to do something … NOW.
  - event-driven programming.
  - external hardware spontaneously cause control transfer (interruption in default program sequence).

Interrupt Mechanism on the Intel 8086

- 8086 has two hardware interrupt signals
  - NMI non-maskable interrupt
  - INTR maskable interrupt



Interrupt Mechanism on the Intel 8086

- Interrupt signals can occur anytime.
- When does processor consider interrupt signals?

The complete instruction execution cycle :

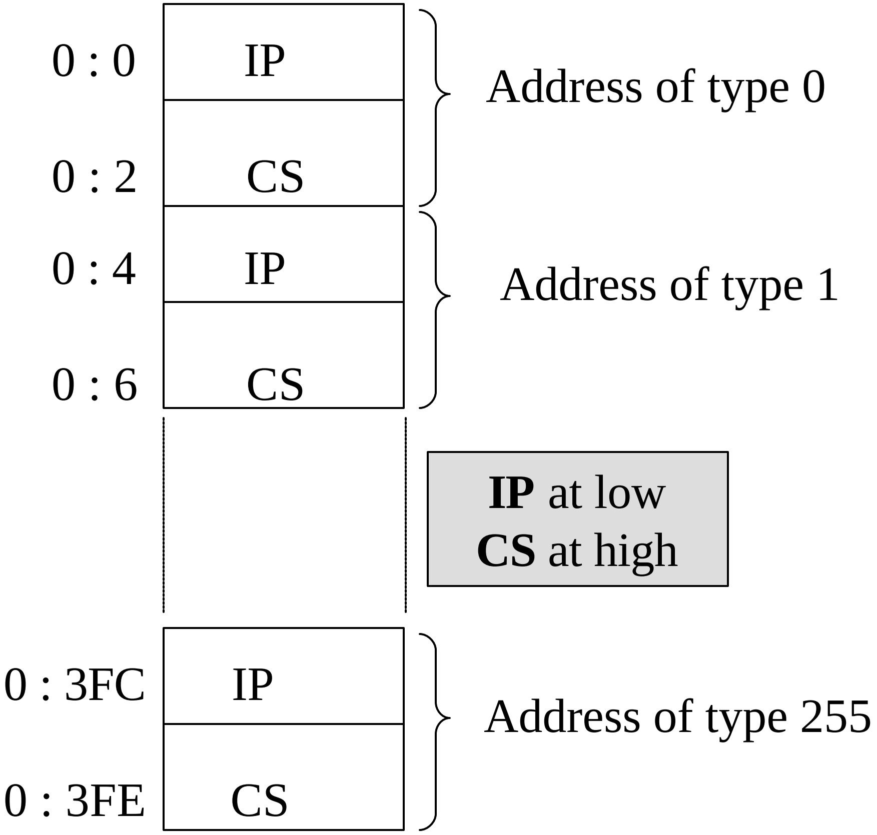
1. Fetch instruction & adjust IP
2. Decode instruction
3. Execute instruction
4. **Check NMI**: **if NMI asserted**, perform related behaviour
5. If **IF = 1** check INTR: **if INTR asserted**,
   perform related behaviour

8086 Vector Table

Just another example of Array Programming

- Array of 256 entries (reserved memory) location 0:0
  - Each entry: address of an interrupt service routine (ISR).
  - Address: FAR Pointer (CS:IP pair) (32-bits = 4bytes)
  - Array occupies addresses from 0:0 to 0:3FF (256*4)

- Each entry identified by unique "interrupt-type" (number; 0-255)
  - Interrupt-type = i, offset to entry in vector table = 0000H + 4*i



IP at low
CS at high

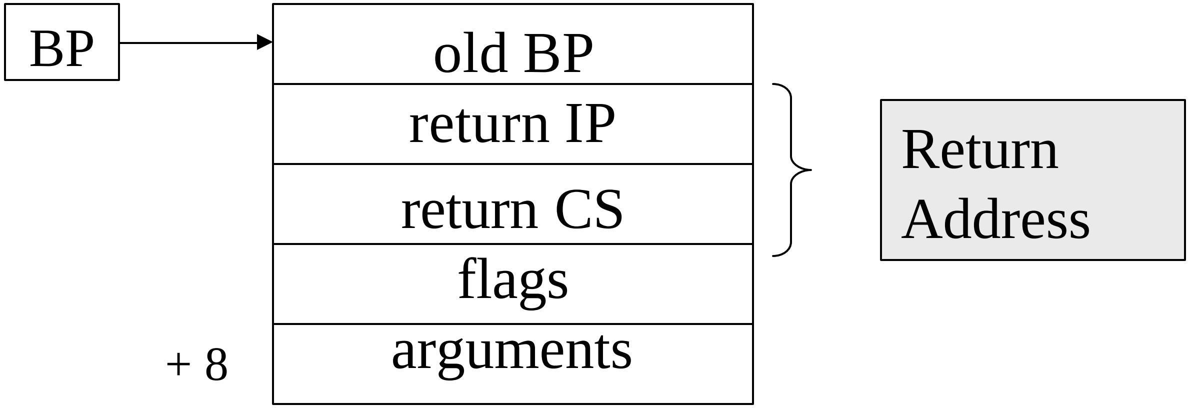
Intel 8086 Vector Table

- The 256 vector table has been mapped out
- A *brief* look:

Interrupt Types	**Descriptions**
0 … 1F	Reserved by Intel Includes : BIOS 10-16h
20h	Terminate a COM program
21-24h	DOS Functions
33h	Mouse Functions
60-6Bh	Available for Applications
80-F0h	Reserved
F1-FFh	Available for Applications

INT Stack Frame

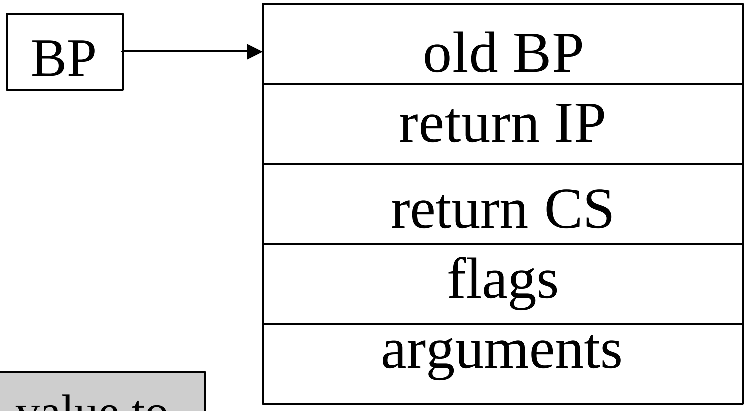
- ISR stack frame: different from subroutines!
  - Return address: a FAR address (CS:IP)
  - FLAGs are also pushed.



  - What does this mean about parameter access *<u>if</u> we passed parameters to the ISR on the stack ?*
    - *<u>Can</u>* we pass parameters to an ISR *<u>for a software interrupt</u>* ?

Returning from an ISR

- RET instruction will not work
  - Why?



- When writing ISRs, use **IRET**.
  1. Pops 32-bit return address (CS:IP)
  2. Pops flags

Restores FLAGS value to what they were before IF and TF were cleared !

- **Example** :

```
isr PROC FAR
   IRET
isr ENDP
```

- Never use "CALL" to invoke ISR; IRET at the end will cause incorrect return.

Installing an ISR

- INT mechanism only works if ISR address loaded into correct vector!
  - This is called “installing” the ISR in the vector table

Installation of an ISR

```
myint_type      EQU   40     ; install as vector
myvector        EQU   myint_type * 4

.code
myisr     PROC  FAR
    IRET
myisr     ENDP

main PROC
    MOV   AX, 0
    MOV   ES, AX    ; ES → vector table segment
    MOV   ES:myvector , OFFSET myisr
    MOV   ES:myvector+2 , @code
        . . .
    INT    myint_type        ; invoke ISR
```

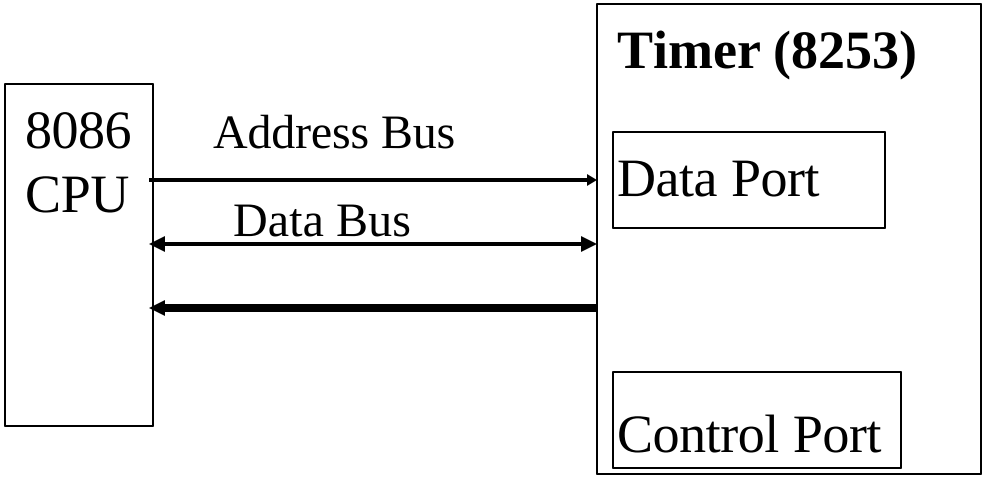
segment override for destination segment !

Installing the ISR

- When installing ISR, you over-write previous value (inserted by OS during startup)
  - Entry may have a useful DOS value .
  - Even "unused" entries have address of a default ISR (consisting of a simple return)

- Existing contents should be saved before installing new vectors.
  - Saved values: restored before the program exit
  - Save/restore not done: the OS (eg. DOS) might not run properly.
  - Does this sound familiar? (Hint : Consider SYSC-3006 Subroutine Policies)

Example : Hardware Timing on a PC

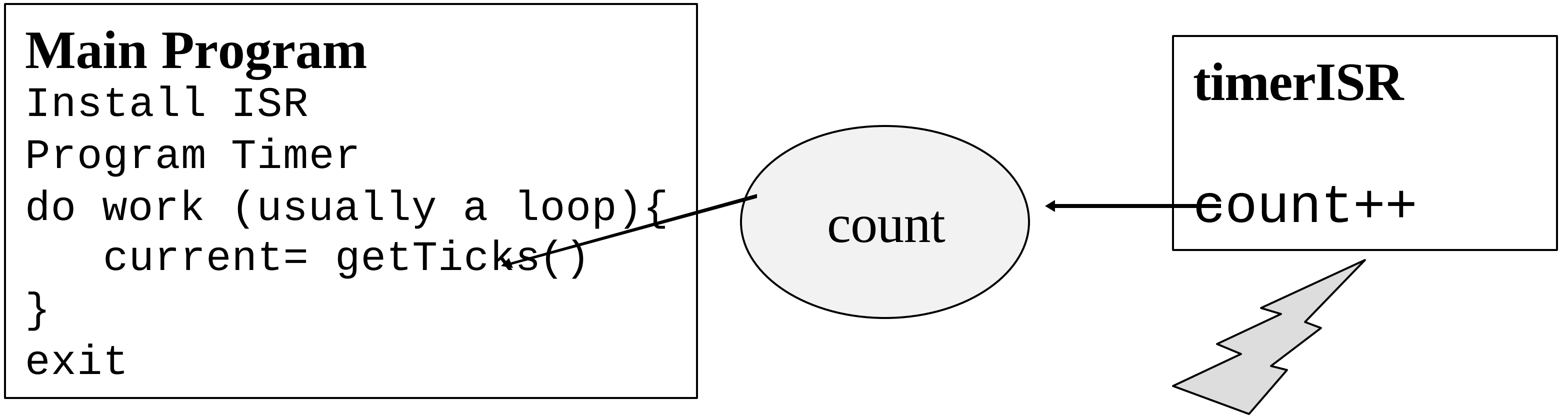


- Provides timing signals
- When connected to CPU, provides regular interrupts.
- Programmable : different interrupt rates
- On a PC, invokes INT 8 ISR.

- Applications use 8253 to provide timing information
  - eg. DOS time-of-day
- What does the ISR do?
- How does the application get the time-of-day?

Typical Timing Software

- Two software components - main program and timerISR
- Shared variable: long unsigned ticks
- timerISR increments count every “tick”
- Main program reads count whenever it needs to
  - Subroutine: `double getTicks()`



Main Program

```
Install ISR
Program Timer
do work (usually a loop){
   current= getTicks()
}
exit
```

count

timerISR

```
count++
```

Skeleton Timing Software

```
; PIC registers and constants
        PIC_COMMAND_REG       equ  20H
        PIC_IMR_REG           equ  21H
        EOI                   equ  20H

; Timer registers and constants
        TIMER_0_REG           equ  40h
        TIMER_CTRL_REG        equ  43H
        TIMER_0_MODE          equ  36H
```

PIC – Programmable Interrupt Controller
(Details later)

EOI – End of interrupt
(Details later)

Skeleton Timing Software

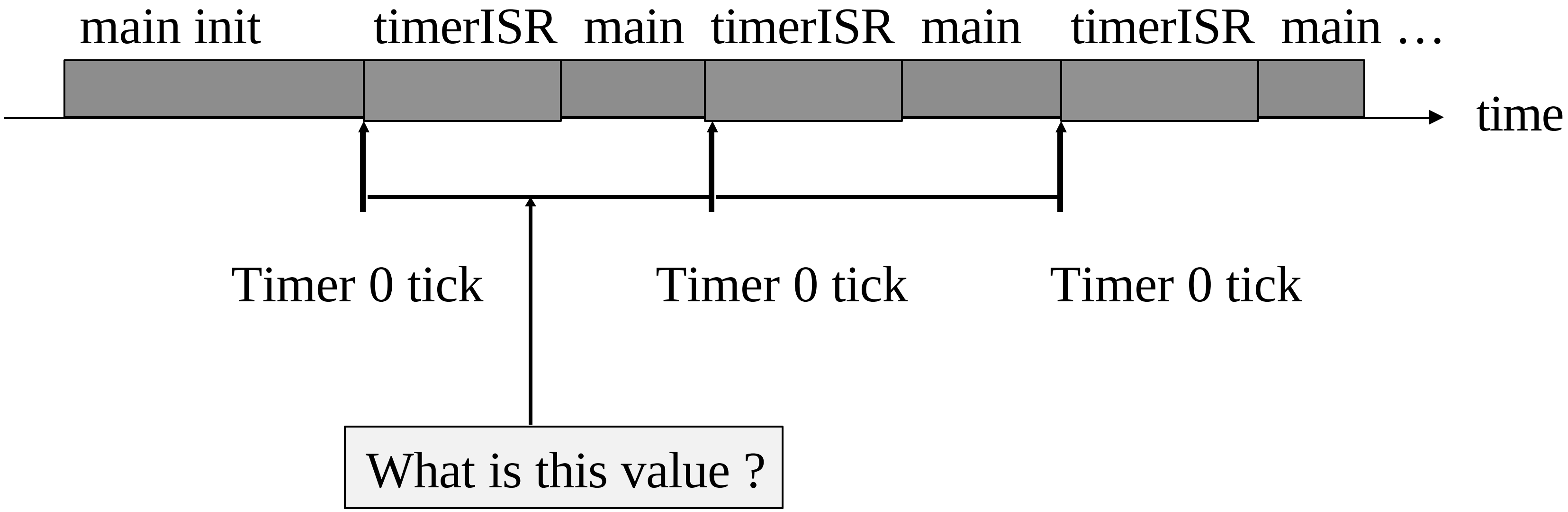
```
.code
SUB    AX , AX        ; trick!  AX = 0 !
MOV   count_low , AX  ; tick count  =  0
MOV   count_high , AX

; Save original INT 8 vector
CLI                        ; Disable interrupts

; Install ISR as new INT 8 vector
MOV AX, 0    ; Not necessary in this case
MOV ES, AX
MOV BX, 8*4
MOV ES:[BX], offset timerisr
MOV ES:[BX+2], CS
```

```
; Program timer 0 to interrupt at 20 Hz
   MOV AL , TIMER_0_MODE          ; Control Register
   MOV DX , TIMER_CTRL_REG
   OUT DX , AL
   MOV AL , 0BH                   ; scaling factor =  E90BH
   MOV DX , TIMER_0_REG
   OUT DX , AL                    ; write low byte
   MOV AL , 0E9H
   OUT DX , AL                    ; write high byte

;  Enable Interrupts at PIC and at processor
   MOV DX , PIC_IMR_REG
   IN  AL , DX
   MOV old_pic_mask, AL           ; Save for later restore
   AND AL , 0FEH                  ; clear bit 0 of imr
   OUT DX , AL
   STI                            ; Enable Interrupts
   . . .
   CALL get_ticks
```

```
timerisr   PROC FAR
   STI
   PUSH   DS       ; Save EVERY register used
   PUSH   DX
   PUSH   AX
   MOV    AX, SEG DATA
   MOV    DS, AX
   INC    count_low
   JNC    done
   INC    count_high

done:
   CLI                ; Lock out all ints until IRET
   MOV    AL, EOI     ; Send EOI to PIC (Details later)
   MOV    DX, PIC_COMMAND_REG
   OUT    DX, AL
   POP    AX          ; Restore registers
   POP    DX
   POP    DS
   IRET
timerisr   ENDP
```

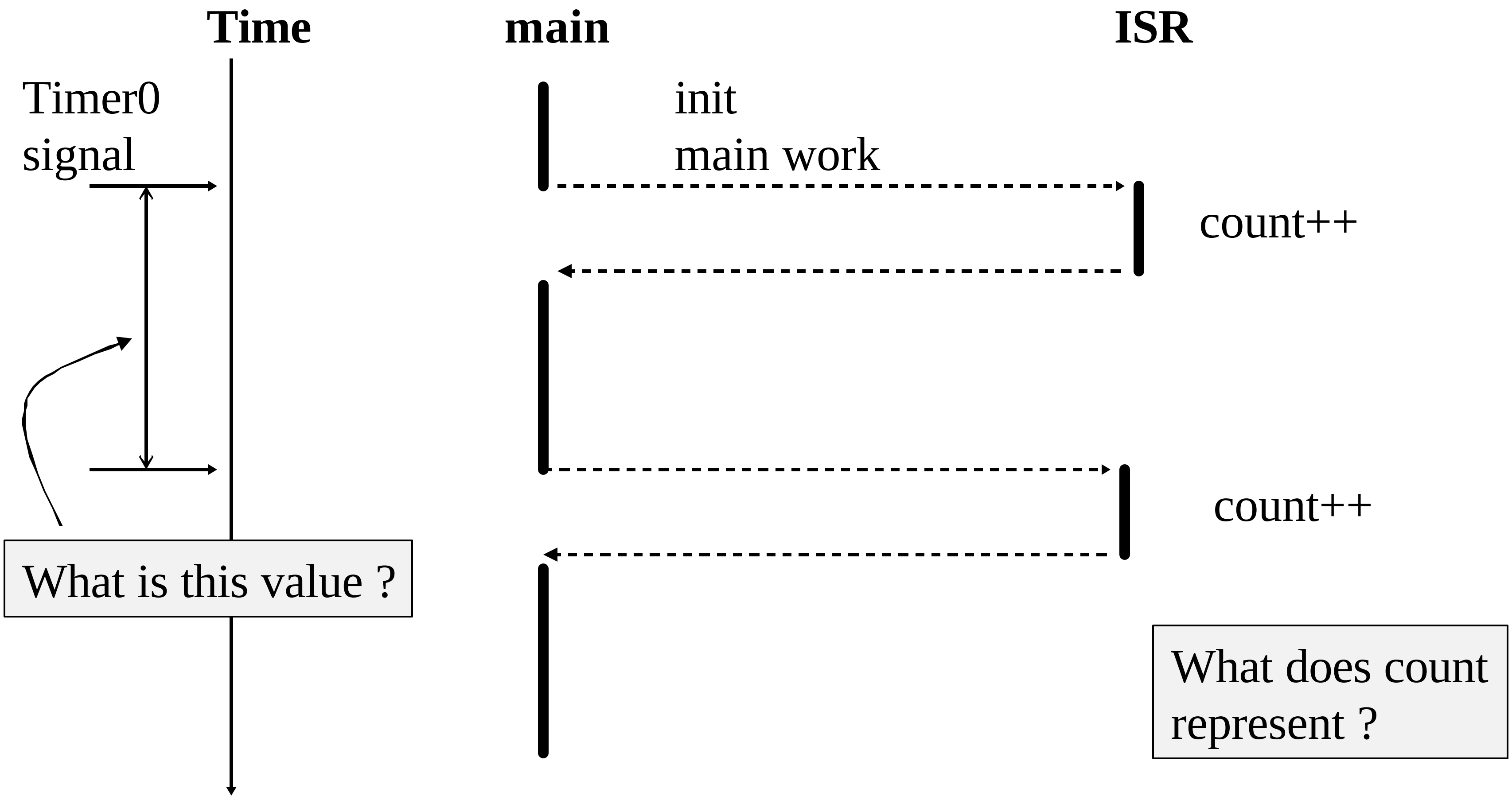
20Hz Real-Time Clock : A Timing Analysis

- CPU Utilization Diagram



20Hz Real-Time Clock : A Timing Analysis

- Thread Diagram



What is this value ?

What does count represent ?

Interrupt Maskability

- Maskable interrupts can be ignored by the CPU
  - Enabled before interrupting the CPU (set an associated flag).
  - 8086 Example : INTR is maskable
    - INTR is masked (disabled) when IF=0

- NonMaskable interrupts cannot be ignored by the CPU
  - 8086 Example: NMI is non-maskable
    - Used for catastrophic errors (e.g. RAM failure, etc).

Interrupt request can be *pending*: **signal is active but it has not yet serviced**

- maskable interrupts: may/not be serviced until/if it is enabled

8086 Instructions for Interrupt Masking

CLI - clears IF bit in flags register (IF = 0)

- disables (masks) interrupts at the processor
- processor does not monitor INTR line while IF = 0

STI - sets IF bit in flags register (IF = 1)

- enables (unmasks) interrupts at the processor
- processor monitors INTR line while IF = 1
- state of IF: does <u>not</u> affect NMI, software interrupts or dedicated interrupts (0..4)
- CLI/STI instructions disable/enable interrupts *at the processor*