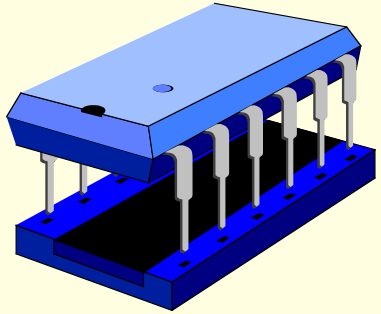
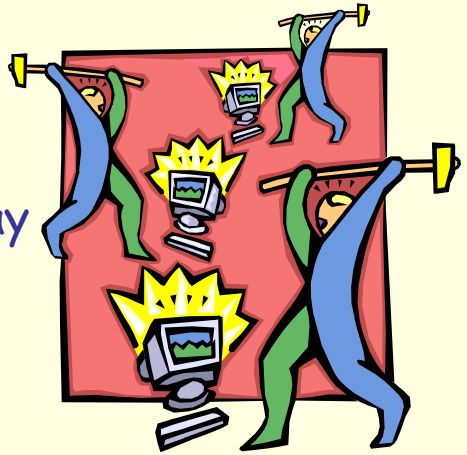
Pentium I/O:

I/O, I/O and off to work we go

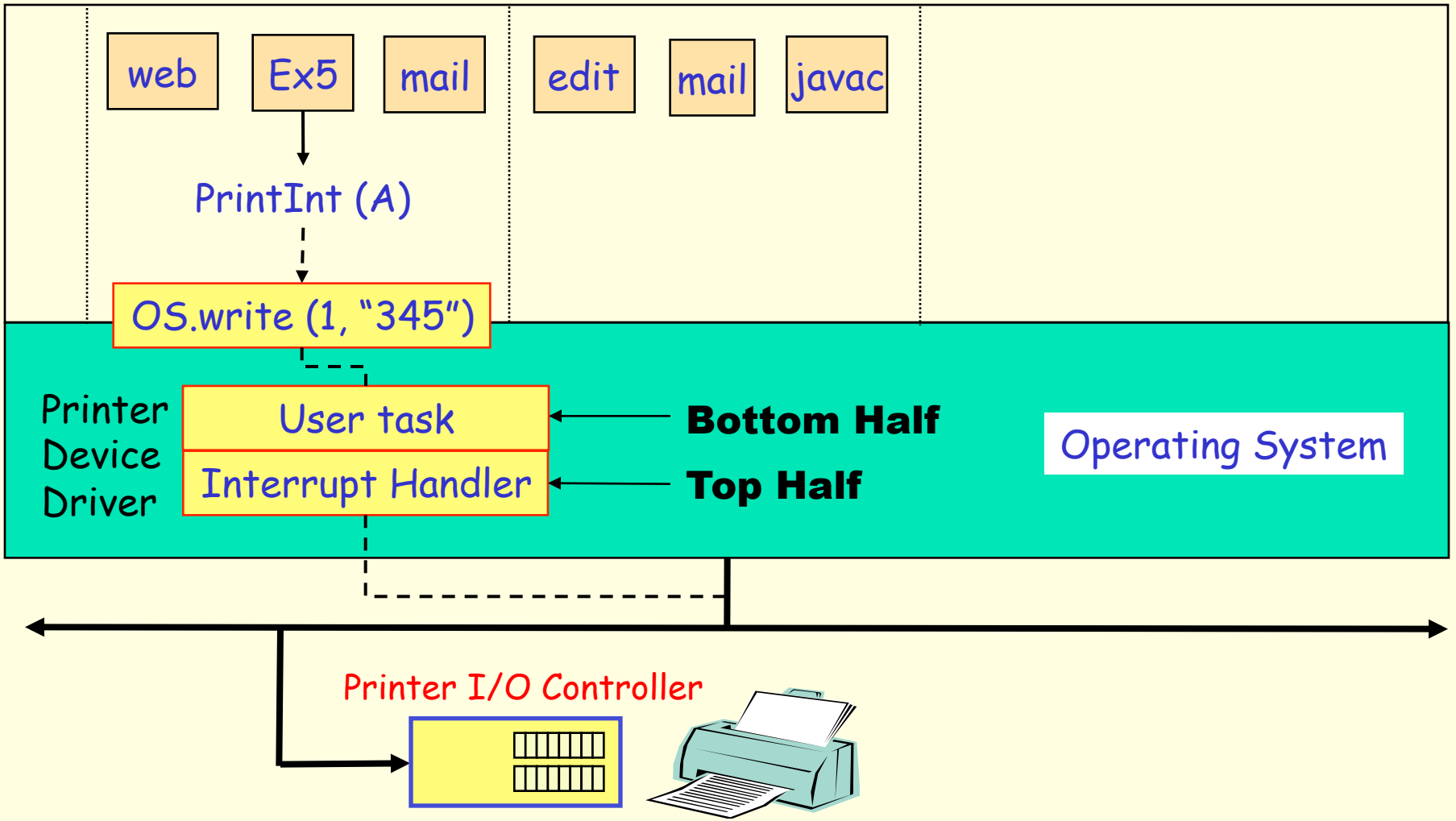
Kin K. Leung
kin.leung@imperial.ac.uk



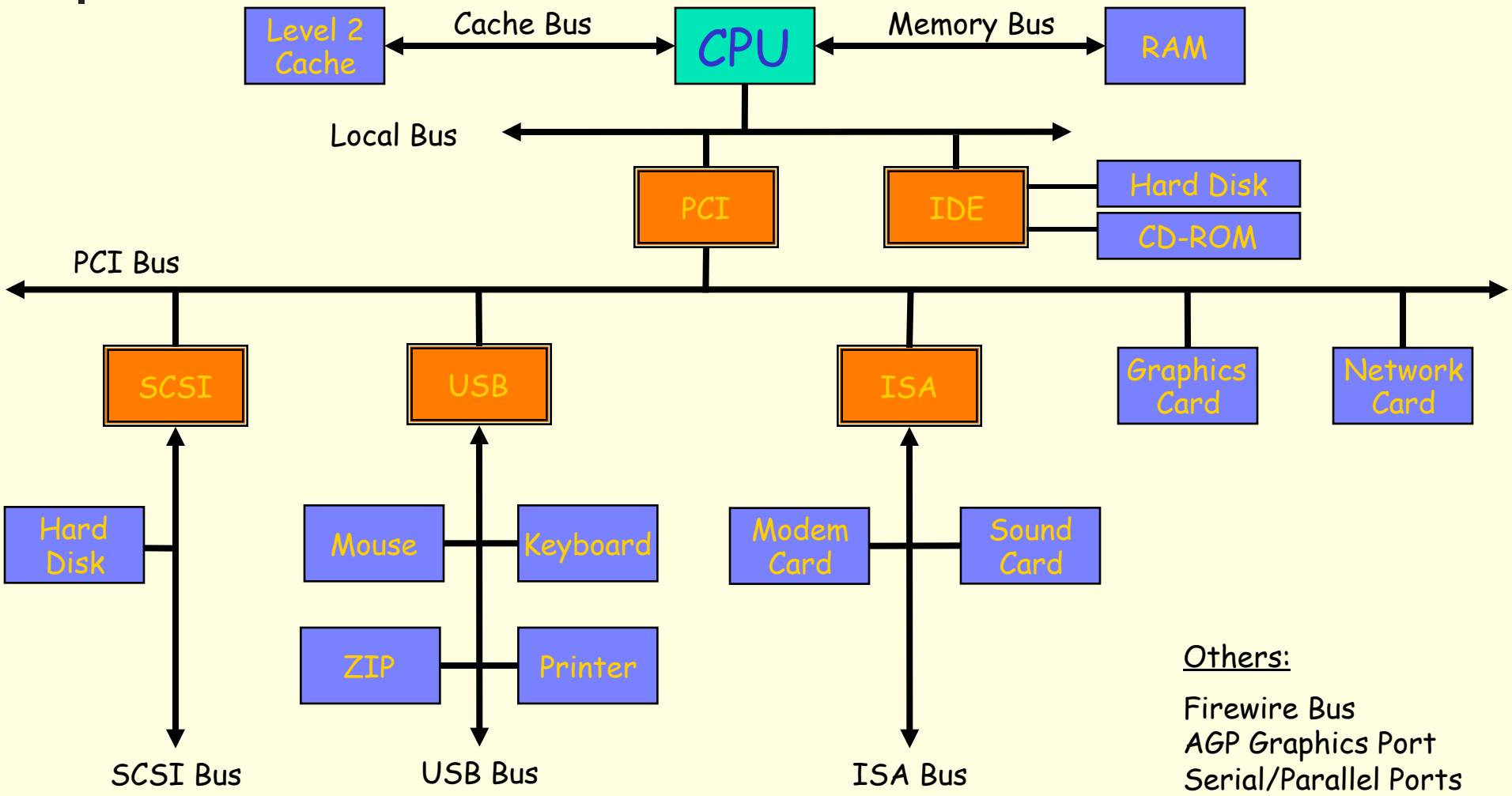
Heavily based on materials from Naranker Dulay



Example: Introduction



Pentium I/O Structure

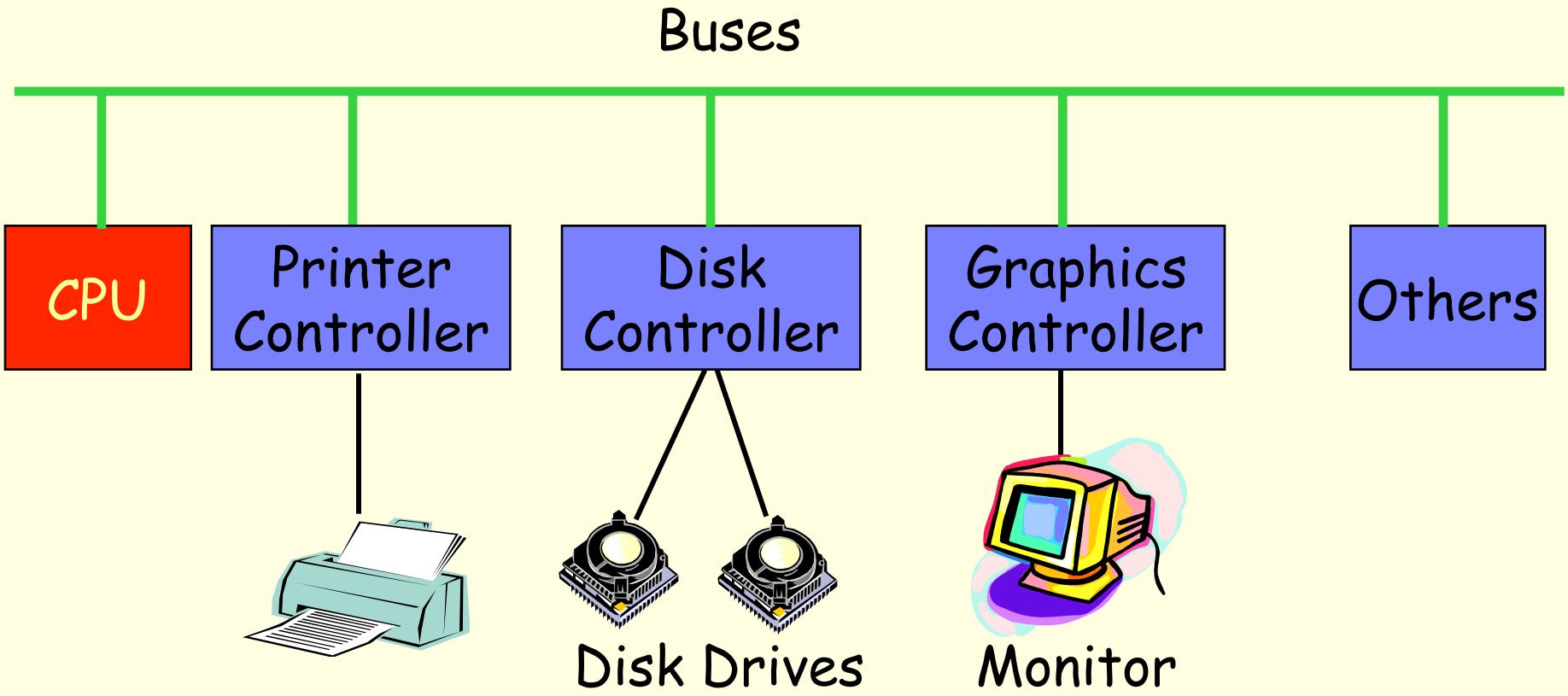


Others:

Firewire Bus
AGP Graphics Port
Serial/Parallel Ports

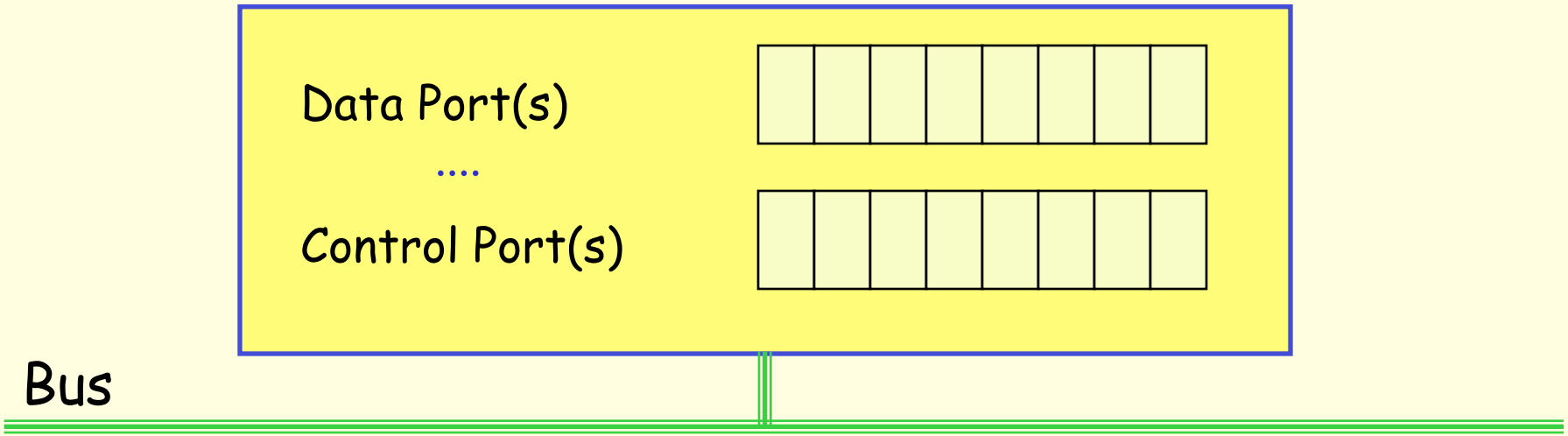
I/O Controllers

- Sometimes known as I/O Adapters or I/O Modules. Provide the CPU with a programming interface.



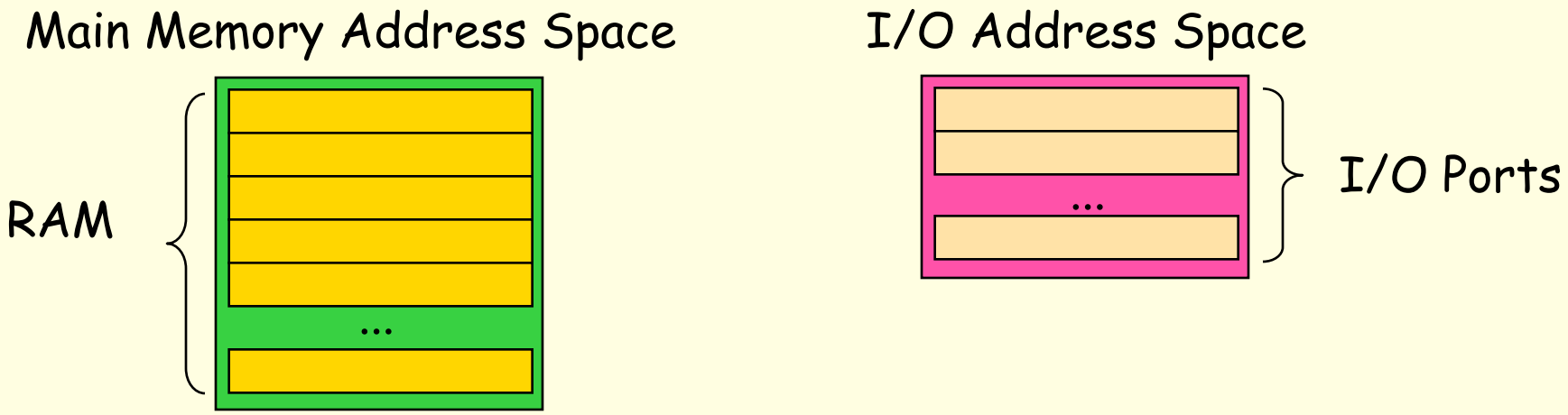
I/O Ports

A Simple I/O Controller



- **Data Ports** are used for passing data to/from the CPU and an I/O device.
- **Control Ports** are used to issue **I/O COMMANDS** (e.g. Write Char) and to check the **STATUS** of a device (e.g. is the device **BUSY** or has an **ERROR** occurred). We write to *specific bits* of a Control Port in order to issue commands, and we read other bits in order to check the status of the device

I/O Port Addressing 1: Separate I/O Address Space

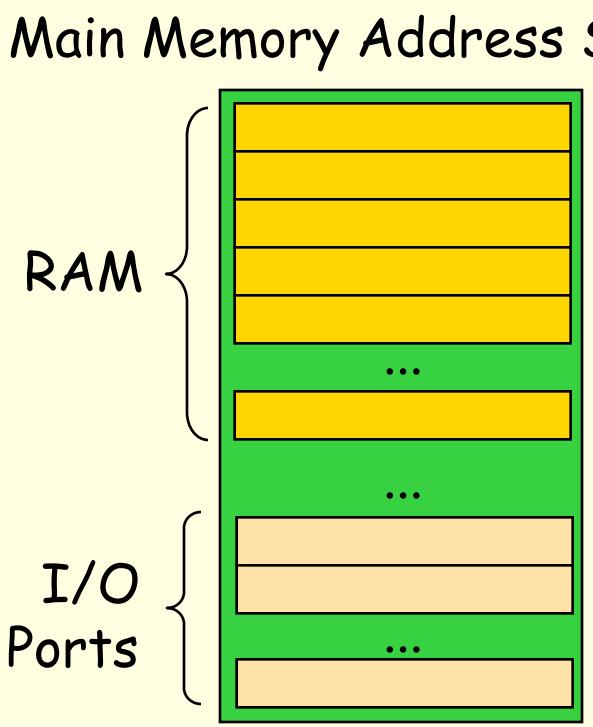


- In this approach, I/O Ports have their own (very small) I/O address space and the architecture provides some method for accessing them, e.g. special I/O instructions:

```
in  ax, 20        ; copy 16-bits from I/O port 20 into register AX
out 35, al        ; copy 8-bits from register AL to I/O port 35
```

- The Control bus is used to signal if a transfer is for the I/O Address Space or the Main Memory Address Space.
- Pentium provides 64K 8-bit I/O ports numbered 0 to 65535

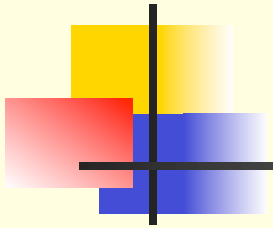
I/O Port Addressing 2: Memory-Mapped I/O



- In this approach, I/O ports "appear" to the programmer as normal memory locations.
- Any instruction that accepts a memory operand can be used to manipulate an I/O port, e.g. MOV, AND, OR, TEST.

```
or [80004444H], al

; Set bits in I/O Port byte at
  80004444H
```

- The Pentium supports both Memory-Mapped I/O and I/O Port Addressing via a seperate I/O address space.



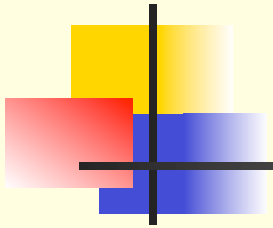
Four I/O Schemes

- **Programmed I/O**
  Continually "poll" a device's control port until the device is ready, then initiate transfer.

- **Interrupt-Driven I/O**
  Initiate transfer and then do something else. Device will "interrupt" the CPU when transfer is complete.

- **DMA I/O**
  Initiate large data (block) transfer. Device will transfer block to/from main memory and then "interrupt" the CPU after block is transferred.

- **I/O Processor**
  Delegate complex I/O processing tasks to a dedicated processor.

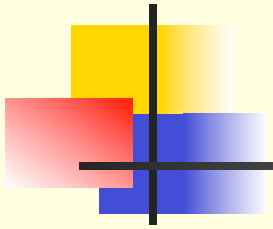


Programmed I/O

- Check Control Port before reading/or writing to Data Port
- **Example**: Writing a block of data to an I/O Device

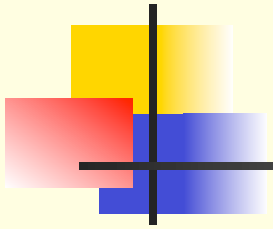
```
foreach byte in buffer do
        loop    Read Status Bit(s) of CONTROL Port
                if Status Bit(s) indicate ERROR then "Handle" Error
                exit when Status Bit(s) indicate DEVICE is "READY"
        endloop
        Copy byte from Buffer to DATA Port
        Issue Write Request by writing to Command bits in CONTROL Port
endfor
```

- Note: for some Output (Input) devices the mere act of writing (reading) to (from) the Data Port initiates a new output (input) operation.



Summary of Programmed I/O

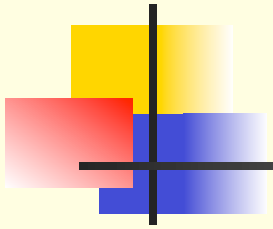
- Simple to Program
- Can guarantee Response times
- Poor CPU utilisation (we waste CPU time **Busy Waiting**)
- Multiple Devices are awkward to handle



Interrupt-driven I/O

- Initiate transfer and then do something else. Device will "interrupt" CPU when transfer is complete. On detecting an interrupt, CPU transfers control to device's interrupt handling procedure (interrupt handler).
- The CPU's Fetch-Execute cycle now becomes:

```
loop
        Fetch Instruction
        Decode Instruction
        Execute Instruction
        if INTERRUPT pending then
                "CALL" Interrupt Handler for this particular Interrupt
        endif
endloop
```



Interrupt Processing

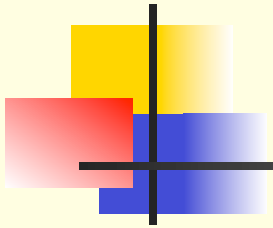
CPU

Complete the current instruction
Identify the interrupt
Save "Processor State" e.g. Push
    Registers such as EIP & EFLAGS
Call Interrupt Handler for
    Identified interrupt

Interrupt Handler

Process Interrupt (PROGRAMMER)
Return from Interrupt restoring
    processor state, i.e. Pop
    Registers

Continue with next instruction



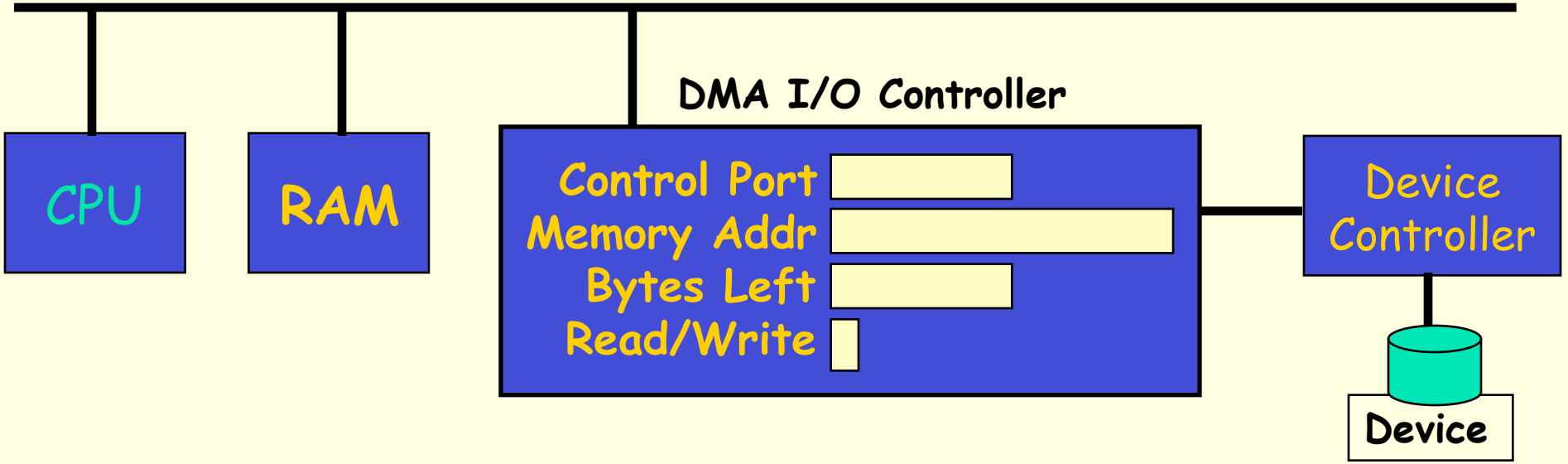
Summary of Interrupt Driven I/O

- Big step forward compared with programmed I/O
- But Interrupt-Processing Time is relatively Expensive
- Bad for high-speed, high-data volume devices that might lose data if they are not serviced quickly-enough.
- Bad if many Devices continually require attention.

We need to be able to reduce the number of interrupts generated

DMA I/O for Block Transfers

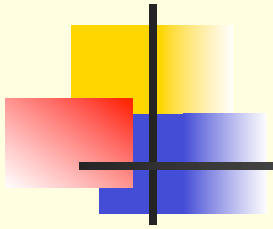
- CPU writes **start address** of block, **number of bytes** of block and **direction of transfer** to DMA's I/O ports and issues Start Command.
- DMA controller transfers block of data between the Device and Main Memory without the direct intervention of the CPU.
- On completion the DMA Controller interrupts CPU.



Cycle-Stealing

- If the CPU and a DMA controller attempt to access memory at the same time, then the DMA controller will always be given priority (why?). This is known as **Cycle-Stealing**.

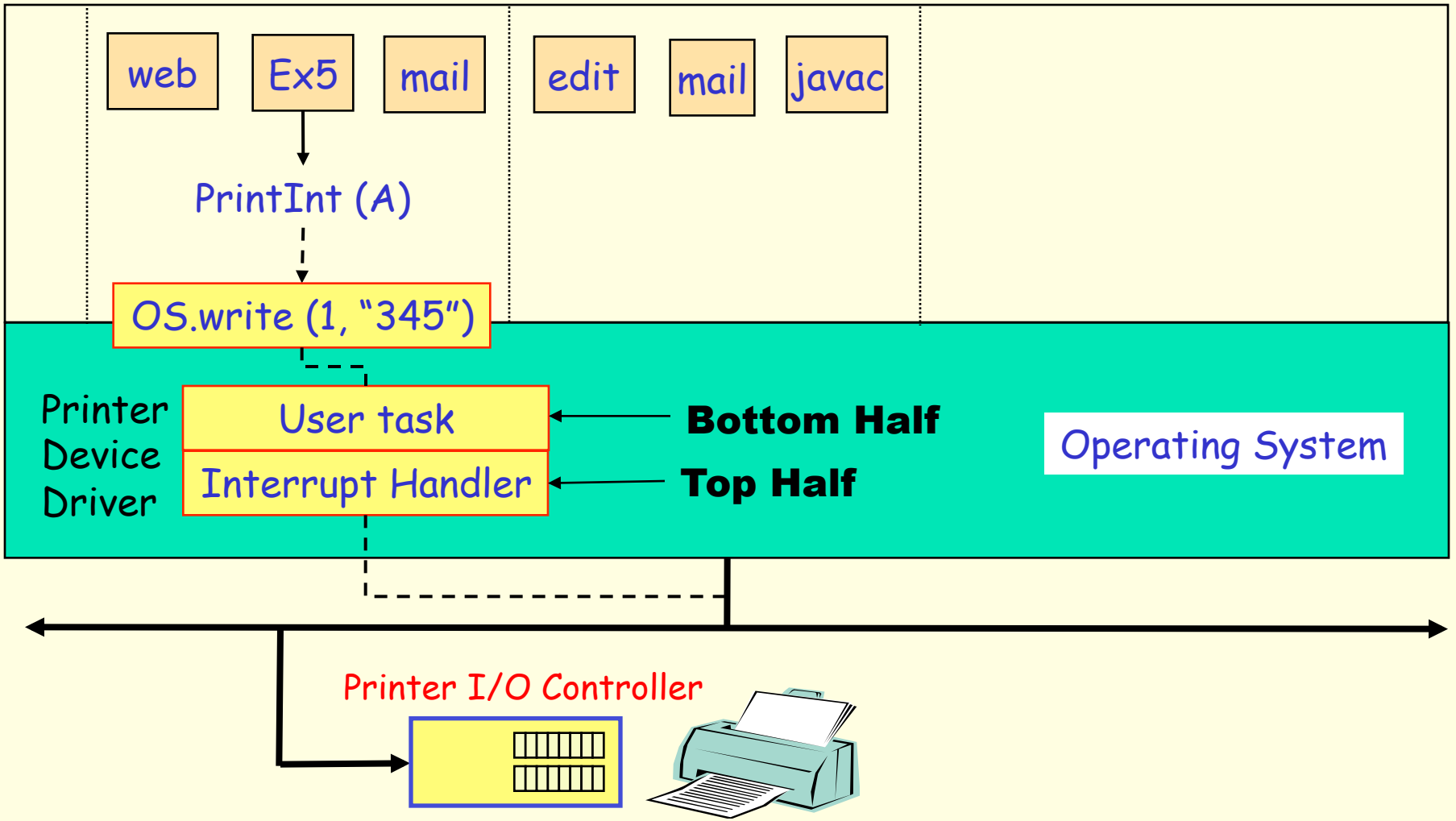
	Time to perform an N-byte transfer
Interrupt I/O:	Time for 1 Interrupt * N
DMA I/O:	Time for 1 Interrupt + (N * Memory Access time)



I/O Processors

- Sometimes even DMA I/O is in-adequate.
- A more powerful approach is to use one or more dedicated I/O processors to relieve the CPU of I/O related tasks.
- I/O processors are more capable than DMA controllers and are often full-featured CPU's with their own local memory for buffering data and executing I/O related tasks.

Device Drivers



Example: Printer Device Driver (Outline)

- To write a string to the Printer. String terminated by a zero byte.

Bottom-Half (User task)

Copy 1st char from string to Printer's ***Data Port***

Issue Write Request by writing 1 to Printer's ***Control Port*** W bit

OS.SUSPEND (suspend bottom-half & run another process)

...

Return from Top-Half

Top-Half (Interrupt Handler)

IF not ***end-of-string*** {

Copy next char from String to ***Data Port***

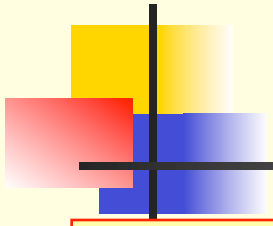
Issue Write Request by writing 1 to ***Control Port***'s W bit

} **ELSE** {

OS.RESUME bottom-half (when convenient)

}

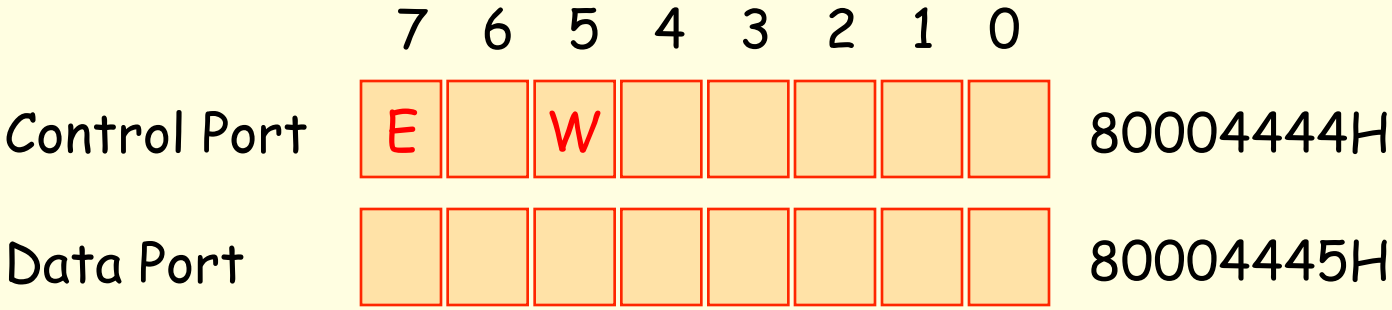
Return from Interrupt



Questions

- How does the CPU know **which** interrupt handler to call?
- How is an interrupt handler actually **called**?
- What happens when we **return** from an interrupt handler?
- How is data passed between the top-half and bottom-half of the device driver?
- What happens if another interrupt is signalled while the interrupt-handler is executing?

Our Printer's I/O Ports



STATUS (checked by reading the Control Port's W and E bits)

Bit W=0	Printer is idle
Bit W=1	Printer is busy
Bit E=0	No Error (All's well)
Bit E=1	An Error has occurred

COMMANDS (issued by writing to the Control Port's W bit)

Set W=1	Causes ASCII Character in Data Port to be printed. On completion I/O controller will reset bit W to 0 and generate an Interrupt (see later).

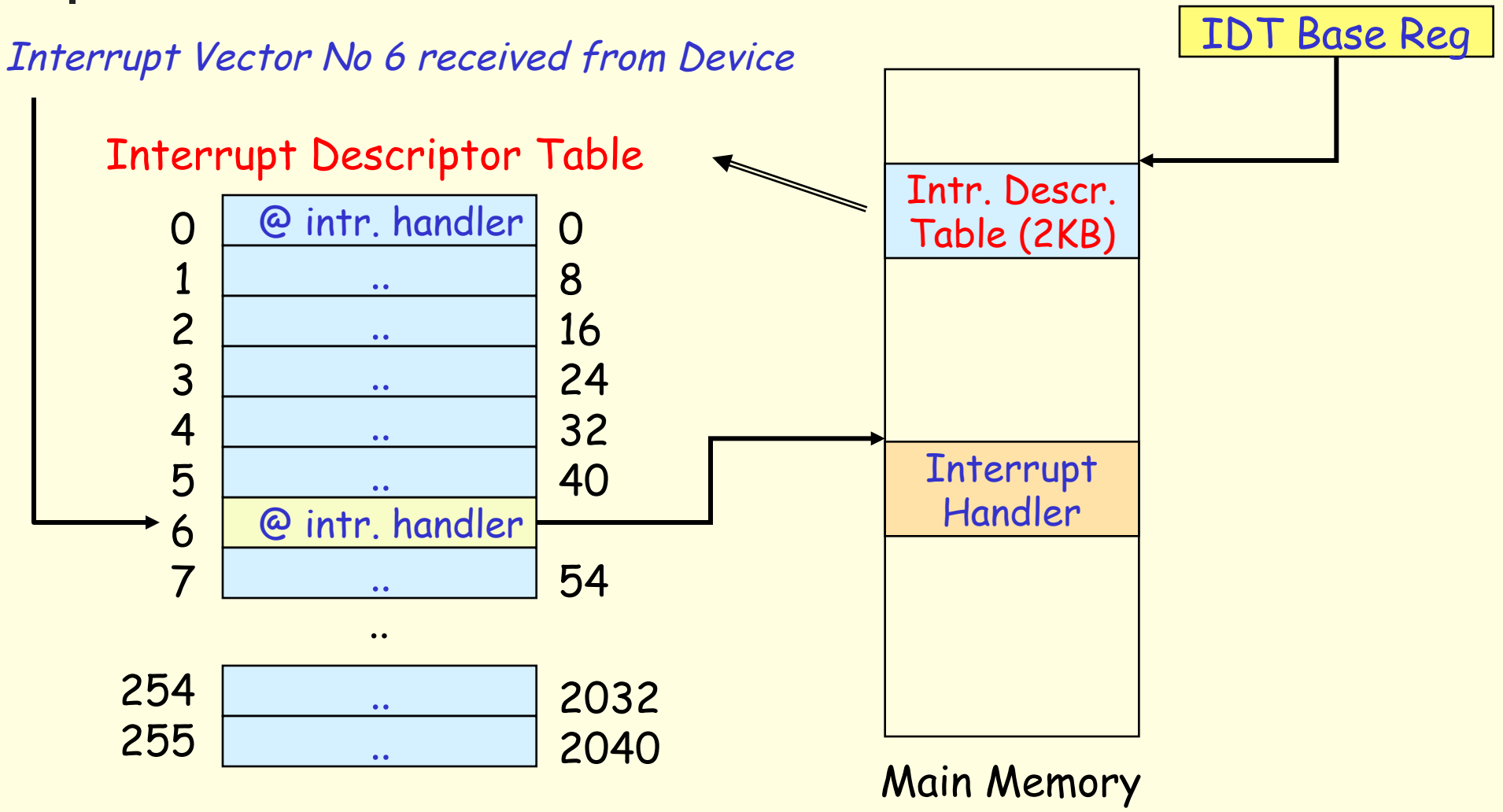
Bottom-Half of our Printer Driver

```
controlport     equ     80004444h       ; address of control port
dataport        equ     80004445h       ; address of data port
string          resb    8192            ; 8Kb string buffer
strptr          resd    1               ; ptr to chars in string buffer

---------------------------------------------------------------------

bottomhalf:     mov     eax, string     ; get address of string
                mov     [strptr], eax   ; save pointer to 1st char
                mov     al, [eax];      ; get 1st char
                jz      skip            ; skip if char is zero-byte
                mov     [dataport], al  ; else copy char to data port
                or      [controlport], 20H ;  and issue write request
                call    OS.SUSPEND      ; suspend bottom-half
skip:           ret                     ; return from bottom-Half
```

Interrupt Handler for Printer Driver

```
handler:    sti                                 ; re-enable interrupts

            push    eax                         ; save registers used in handler
            inc     [strptr]                    ; advance to next char
            mov     eax, [strptr]   ; copy pointer to register
            mov     al, [eax]                   ;     and get char
            jz      endofstr                    ; skip if end of string

            mov     [dataport], al      ; copy char to data port
            or      [controlport], 20h  ; issue write request
            jmp     exit
endofstr:   call        OS.RESUME               ; ask O/S to resume bottom half

exit:       pop     eax

            iret
```

Locating the Interrupt Handler



Locating the Interrupt Handler

- When a device wishes to interrupt the CPU, it sends a special "interrupt" signal to the the CPU, along with an number that identifies the interrupt. This number is called the **INTERRUPT VECTOR NUMBER** and on the Pentium is a number in the range 0 to 255 (unsigned byte).
- The Interrupt Vector Number is used to index an **INTERRUPT DESCRIPTOR TABLE** (IDT) with 256 entries. Each entry of this table is a 64-bit "Descriptor" which includes the Interrupt Handler's start address (32-bits).
- *Actually the start address is 48-bits (the other 16-bits define the memory segment of the interrupt handler). Segmentation will be covered in the 2nd year.*
- The start address of the Interrupt Descriptor Table is held in a special CPU register called the **IDT Base Register (IDTR).**

Types of Interrupt

- External I/O device generated Interrupts - Asynchronous
  I/O Device sends an **INTERRUPT VECTOR NUMBER** to the CPU via Buses.
  Vector Numbers 32-255 are reserved for External Interrupts.

- CPU-Generated Interrupts (EXCEPTIONS) - Synchronous
  E.g. Attempt to execute an illegal operation, divide-by-zero.
  Vector numbers 0 to 18 are reserved for CPU Exceptions.

- Software-Generated Interrupts - Synchronous
  Interrupts can also be generated explicitly with a instruction, e.g.

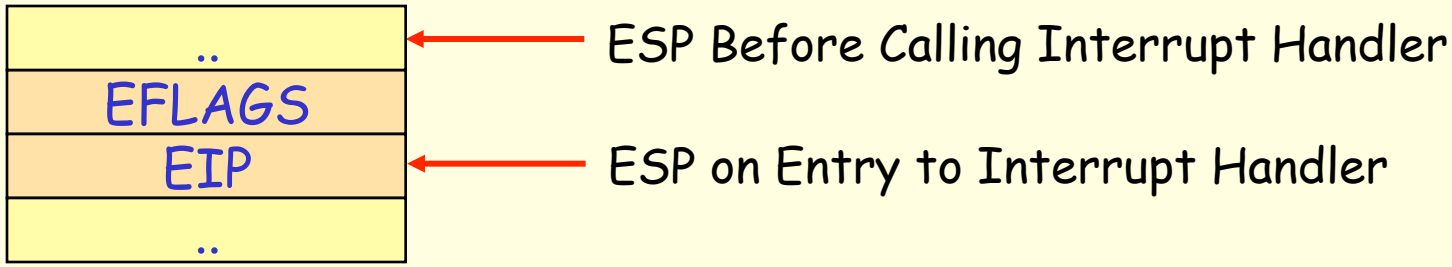
  ```
  int  80H        ; Calls interrupt handler for intr. vector number 80H
  ```

  Such interrupts are commonly used to call an operating system method (a so-called **SYSTEM CALL**) since interrupts cause the privilege level to be switched to kernel-mode from user-mode.

Calling the Interrupt Handler

When an Interrupt occurs the Pentium CPU will:

- Complete the currently executing instruction
- *Push the EFLAGS Register onto the Stack*
- *Clear the Interrupt Flag bit in the EFLAGS register - this disables further interrupts*
- Push the Return Address (i.e. contents of the EIP register) onto the Stack
- Call the Interrupt Handler using the entry in the Interrupt Descriptor Table



- To return, the Interrupt Handler executes the **iret** instruction which restores the state (e.g. EFLAGS and EIP) and jumps to the return address on the stack.
  >> It is the programmer's responsibility to restore any other registers <<

Enabling & Disabling Interrupts

- The Pentium automatically disables interrupts when calling an Interrupt Handler and restores them when returning from the handler - this prevents other devices from interrupting the handler.
- Interrupts can be (re-) enabled more quickly by setting the **Interrupt Enable Flag IF** (bit 9) in the EFLAGS register with the **sti** instruction:

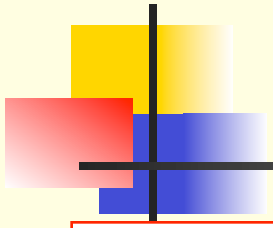
```
sti          ; enables interrupts i.e. sets interrupt enable flag bit in EFLAGS reg.
```

  This is commonly done early within an Interrupt Handler to allow other I/O devices to have their interrupts serviced quickly.

- To explicitly disable interrupts we can use:

```
cli          ; disable interrupts   i.e. clear IF bit in EFLAGS register
```

  This is useful where the bottom-half of a device-driver can run **concurrently** with the top-half, and needs to update data shared with the top-half. By enclosing the update instructions with CLI and STI we can ensure that the top-half cannot interrupt the top-half.



Device Driver Summary

- Device-drivers control I/O devices by reading and writing to the **I/O ports** of the device. With memory-mapped I/O this is akin to reading/writing to memory locations.
- I/O devices signal completion of I/O requests and errors by sending an **Interrupt Vector Number** to the CPU. This causes the CPU to (asynchronously) call (using the Interrupt Descriptor Table) the Device-driver's **INTERRUPT HANDLER**.
- The INTERRUPT HANDLER services the interrupt i.e. checks for errors and copies data to/from the memory area it shares with the Device-driver's BOTTOM-HALF.
- The **BOTTOM-HALF** of the Device driver runs as a thread within the OS and interacts with the device (via I/O ports), the bottom half (via shared memory) and with user-level processes.

- The handling of I/O devices within a computer system plays a critical role in the operation of a computer. Without robust I/O device drivers a computer system will be unstable, subject to random crashes and data corruption. The need for quality "engineering" of I/O device drivers is extremely high. Writing device drivers can be fun but must be taken very seriously.

Interrupt Vector Numbers

- The Pentium supports upto to 256 Interrupts numbered 0 to 255. This number is called the **Interrupt Vector Number.**

Examples

Interrupt Vector No	Cause
0	Divide by Zero attempted
6	Attempt to execute an Instruction with an Illegal opcode field
12	Stack Fault Exception
17	General Protection Violation, e.g. writing to read-only "segment"
32-255	User-defined, used for interrupts from I/O devices

Software Interrupts (System Calls)

- Programs can generate an interrupt with the INT instruction e.g.

```
int  80H          ; Calls interrupt handler for vector number 80H
```

- Software interrupts are commonly used for calling operating system functions (**System Calls**) where the address of the function need not be known.

- Software interrupts are also known as TRAPS.

LINUX System Calls

- Interrupt 80H is used for > 150 LINUX system calls.
- **System Call number** is passed in register EAX.
  Examples: 1=Exit Program, 3=Read from standard input, 4=Write to standard output.
- Parameters 1, 2, 3, 4, 5 are passed in registers ebx, ecx, edx, esi, edi. Result is passed back in register eax or via reference parameters.

- EXAMPLE: Write string to standard output

```
mov  eax, 4                    ; Linux system call 4: Write( )
mov  ebx, 1                    ; Parameter 1: 1=File descriptor for Std output
mov   ecx, AddrOfString        ; Parameter 2
mov   edx, LengthOfString      ; Parameter 3
int     80H                    ; Call Operating System
```

- We also need to save registers before the call, and pop them afterwards.

FreeBSD System Calls

- Interrupt 80H is used for FreeBSD system calls. Mostly the same as Linux.
- **System Call number** is passed in register eax.
- Parameters to FreeBSD system calls are passed on the stack (in reverse order). The result from the call is returned in register eax. An extra doubleword should also be pushed before the INT.

- EXAMPLE: Write string to Standard Output

```
mov   eax, 4                    ; System call 4: Write( )
push  dword LengthOfString      ; Parameter 3
push  dword AddrOfString        ; Parameter 2
push  dword 1           ; Parameter 1: 1=File descriptor for Std output
push  anyDWORD                  ; Push any DoubleWord
int    80H                      ; Call Operating System
add   esp, 16           ; Remove Parameters from Stack
```

MS-DOS System Calls

- Interrupt 21H (Vector number 33) is used for ~ 90 MS-DOS system calls
- **System call number** is passed in register AH. **Examples**: 1=Read character from standard input, 2=Write character to output, 4C=Exit program
- Parameters are passed in other registers (Documentation defines which registers, for which system call)

- EXAMPLE: MS-DOS Function 2 writes a char in register DL to the Standard Output

```
mov   ah, 2          ; MS-DOS system call 2: Write character to Std Output
mov   dl, 'a'        ; Parameter 1: Character to be output
int   21H            ; Call Operating System
```

- We also need to push registers before the call, and pop them afterwards.

That's all Folks !