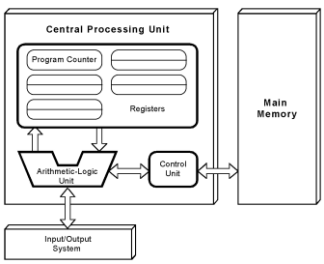
I/O Architecture, Interrupts, Exceptions

CMSC 313
Sections 01, 02

Fetch Execute Cycle

1.8 The von Neumann Model

- This is a general depiction of a von Neumann system:
- These computers employ a fetch-decode-execute cycle to run programs as follows . . .



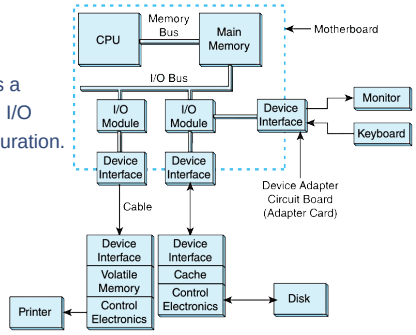
I/O Architectures

7.4 I/O Architectures

- We define input/output as a subsystem of components that moves coded data between external devices and a host system.
- I/O subsystems include:
  - Blocks of main memory that are devoted to I/O functions.
  - Buses that move data into and out of the system.
  - Control modules in the host and in peripheral devices
  - Interfaces to external components such as keyboards and disks.
  - Cabling or communications links between the host system and its peripherals.

7.4 I/O Architectures

This is a model I/O configuration.



7.4 I/O Architectures

- I/O can be controlled in five general ways.
  - *Programmed I/O* reserves a register for each I/O device. Each register is continually polled to detect data arrival.
  - *Interrupt-Driven I/O* allows the CPU to do other things until I/O is requested.
  - *Memory-Mapped I/O* shares memory address space between I/O devices and program memory.
  - *Direct Memory Access* (*DMA*) offloads I/O processing to a special-purpose chip that takes care of the details.
  - *Channel I/O* uses dedicated I/O processors.

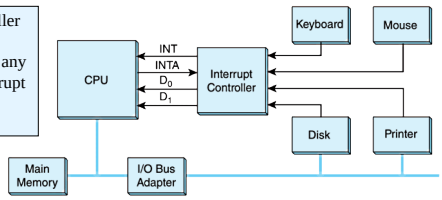
7

7.4 I/O Architectures

This is an idealized I/O subsystem that uses interrupts.

Each device connects its interrupt line to the interrupt controller.

The controller signals the CPU when any of the interrupt lines are asserted.



8

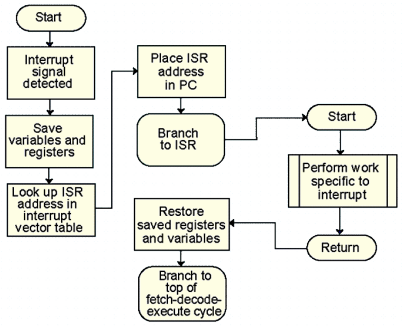
7.4 I/O Architectures

- Recall from Chapter 4 that in a system that uses interrupts, the status of the interrupt signal is checked at the top of the fetch-decode-execute cycle.
- The particular code that is executed whenever an interrupt occurs is determined by a set of addresses called *interrupt vectors* that are stored in low memory.
- The system state is saved before the interrupt service routine is executed and is restored afterward.

We provide a flowchart on the next slide.

9

7.4 I/O Architectures



10

7.4 I/O Architectures

- In memory-mapped I/O devices and main memory share the same address space.
  - Each I/O device has its own reserved block of memory.
  - Memory-mapped I/O therefore looks just like a memory access from the point of view of the CPU.
  - Thus the same instructions to move data to and from both I/O and memory, greatly simplifying system design.
- In small systems the low-level details of the data transfers are offloaded to the I/O controllers built into the I/O devices.

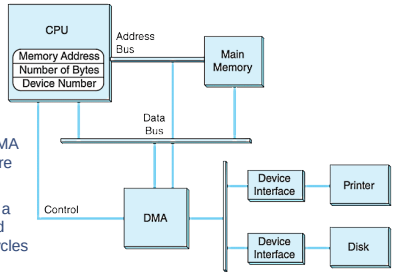
11

7.4 I/O Architectures

This is a DMA configuration.

Notice that the DMA and the CPU share the bus.

The DMA runs at a higher priority and steals memory cycles from the CPU.



12

7.4 I/O Architectures

- Very large systems employ channel I/O.
- Channel I/O consists of one or more I/O processors (IOPs) that control various channel paths.
- Slower devices such as terminals and printers are combined (*multiplexed*) into a single faster channel.
- On IBM mainframes, multiplexed channels are called *multiplexor channels*, the faster ones are called *selector channels*.

13

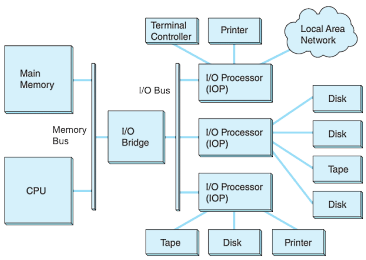
7.4 I/O Architectures

- Channel I/O is distinguished from DMA by the intelligence of the IOPs.
- The IOP negotiates protocols, issues device commands, translates storage coding to memory coding, and can transfer entire files or groups of files independent of the host CPU.
- The host has only to create the program instructions for the I/O operation and tell the IOP where to find them.

14

7.4 I/O Architectures

- This is a channel I/O configuration.



15

Interrupts

16

Motivating Example

```
; An Assembly language program for printing data
        MOV EDX, 378H          ;Printer Data Port
        MOV ECX, 0             ;Use ECX as the loop counter
XYZ:    MOV AL, [ABC + ECX]    ;ABC is the beginning of the memory area
                               ; that characters are being printed from
        OUT [DX], AL           ;Send a character to the printer
        INC ECX
        CMP ECX, 100000        ; print this many characters
        JL XYZ
```

Issues:

- What about difference in speed between the processor and printer?
- What about the buffer size of the printer?
  - Small buffer can lead to some lost data that will not get printed

Communication with input/output devices needs handshaking protocols

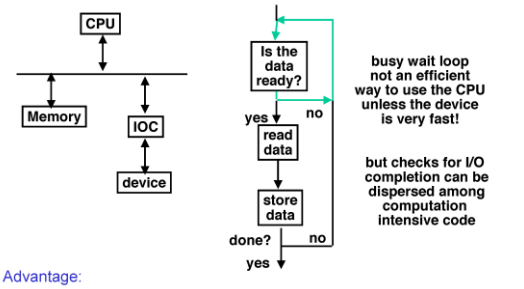
Communicating with I/O Devices

- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- This can be accomplished in two different ways:
  - Polling:
    - The I/O device put information in a status register
    - The OS periodically check the status register
  - I/O Interrupt:
    - An I/O interrupt is an externally stimulated event, asynchronous to instruction execution but does NOT prevent instruction completion
    - Whenever an I/O device needs attention from the processor, it interrupts the processor from what it is currently doing
    - Some processors deals with interrupts as special exceptions

These schemes requires heavy processor's involvement and suitable only for low bandwidth devices such as the keyboard

* Slide is partially a courtesy of Dave Patterson

Polling: Programmed I/O



- Advantage:
  - Simple: the processor is totally in control and does all the work
- Disadvantage:
  - Polling overhead can consume a lot of CPU time

* Slide is courtesy of Dave Patterson

Polling in 80386

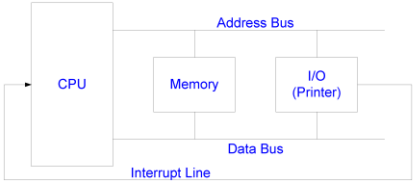
```
        MOV EDX, 379H          ;Printer status port
        MOV ECX, 0
XYZ:    IN AL, [DX]            ;Ask the printer if it is ready
        CMP AL, 1              ;1 means it's ready
        JNE XYZ                ;If not try again
        MOV AL, [ABC + ECX]
        DEC EDX                ;Data port is 378H
        OUT [DX], AL           ;Send one byte
        INC ECX
        INC EDX                ;Put back the status port
        CMP ECX, 100000
        JL XYZ
```

Issues:

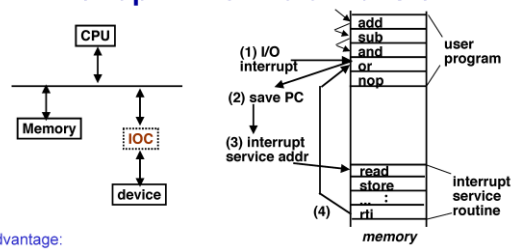
- Status registers (ports) allows handshaking between CPU and I/O devices
- Device status ports are accessible through the use of typical I/O instructions
- CPU is running at the speed of the printer (what a waste!!)

External Interrupt

- The fetch-execute cycle is a program-driven model of computation
- Computers are not totally program driven as they are also hardware driven
- An I/O interrupt is an externally stimulated event, asynchronous to instruction execution but does NOT prevent instruction completion
- Whenever an I/O device needs attention from the processor, it interrupts the processor from what it is currently doing
- Processors typically have one or multiple interrupt pins for device interface



Interrupt Driven Data Transfer

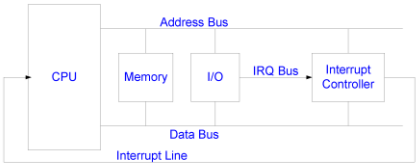


- Advantage:
  - User program progress is only halted during actual transfer
- Disadvantage: special hardware is needed to:
  - Cause an interrupt (I/O device)
  - Detect an interrupt (processor)
  - Save the proper states to resume after the interrupt (processor)

* Slide is courtesy of Dave Patterson

80386 Interrupt Handling

- The 80386 has only one interrupt pin and relies on an interrupt controller to interface and prioritize the different I/O devices
- Interrupt handling follows the following steps:
  1. Complete current instruction
  2. Save current program counter and flags into the stack
  3. Get interrupt number responsible for the signal from interrupt controller
  4. Find the address of the appropriate interrupt service routine
  5. Transfer control to interrupt service routine
- A special interrupt acknowledge bus cycle is used to read interrupt number
- Interrupt controller has ports that are accessible through IN and OUT



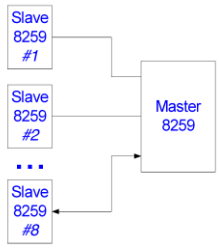
Interrupt Descriptor Table

Address	
b	Gate #0
b + 8	Gate #1
b + 16	Gate #2
b + 24	Gate #3
b + 32	Gate #4
b + 40	Gate #5
	. . .
b + 2040	Gate #255

- The address of an ISR is fetched from an interrupt descriptor table
- IDT register is loaded by operating system and points to the interrupt descriptor table
- Each entry is 8 bytes indicating address of ISR and type of interrupt (trap, fault etc.)
- RESET and non-maskable (NMI) interrupts use distinct processor pins
- NMI is used to for parity error or power supply problems and thus cannot be disables

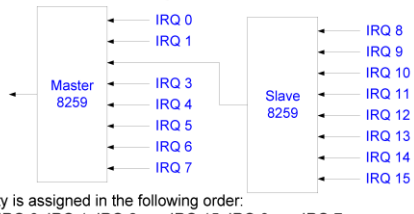
The 8259 Interrupt Controller

- Since the 80386 has one interrupt pin, an interrupt controller is needed to handle multiple input and output devices
- The Intel 8259 is a programmable interrupt controller that can be used either singly or in a two-tier configuration
- When used as a master, the 8259 can interface with up to 8 slaves
- Since the 8259 controller can be a master or a slave, the interrupt request lines must be programmable
- Programming the 8259 chips takes place at boot time using the OUT commands
- The order of the interrupt lines reflects the priority assigned to them



The ISA Architecture

- The ISA architecture is set by IBM competitors and standardizes:
  - The interrupt controller circuitry
  - Many IRQ assignments
  - Many I/O port assignments
  - The signals and connections made available to expansion cards
- A one-master-one-slave configuration is the norm for ISA architecture



- Priority is assigned in the following order:
  IRQ 0, IRQ 1, IRQ 8, …, IRQ 15, IRQ 3, …, IRQ 7

ISA Interrupt Routings

IRQ	ALLOCATION	INTRRUPT NUMBER
IRQ0	System Timer	08H
IRQ1	Keyboard	09H
IRQ3	Serial Port #2	OBH
IRQ4	Serial Port # 1	OCH
IRQ5	Parallel Port #2	ODH
IRQ6	Floppy Controller	OEH
IRQ7	Parallel Port # 1	OFH
IRQ8	Real time clock	70H
IRQ9	available	71 H
IRQ10	available	72H
IRQ11	available	73H
IRQ12	Mouse	74H
IRQ13	87 ERROR line	75H
IRQ14	Hard drive controller	76H
IRQ15	available	77H

linux1$ cat /proc/interrupts

Exceptions

28

Built-in Hardware Exceptions

Allocation	Int #
Division Overflow	00H
Single Step	01H
NMI	02H
Breakpoint	03H
Interrupt on Overflow	04H
BOUND out of range	05H
Invalid Machine Code	06H
87 not available	07H
Double Fault	08H
87 Segment Overrun	09H
Invalid Task State Segment	0AH
Segment Not Present	0BH
Stack Overflow	0CH
General Protection Error	0DH
Page Fault	0EH
(reserved)	0FH
87 Error	10H

UMBC, CMSC313, Richard Chang <rchang@umbc.edu>

I/O Interrupt vs. Exception

- An I/O interrupt is just like the exceptions except:
  - An I/O interrupt is asynchronous
  - Further information needs to be conveyed
  - Typically exceptions are more urgent than interrupts
- An I/O interrupt is asynchronous with respect to instruction execution:
  - I/O interrupt is not associated with any instruction
  - I/O interrupt does not prevent any instruction from completion
    - You can pick your own convenient point to take an interrupt
- I/O interrupt is more complicated than exception:
  - Needs to convey the identity of the device generating the interrupt
  - Interrupt requests can have different urgencies:
    - Interrupt request needs to be prioritized
    - Priority indicates urgency of dealing with the interrupt
    - High speed devices usually receive highest priority

* Slide is courtesy of Dave Patterson

Internal and Software Interrupt

- Exceptions:
  - Exceptions do not use the interrupt acknowledge bus cycle but are still handled by a numbered ISR
  - Examples: divide by zero, unknown instruction code, access violation, …
- Software Interrupts:
  - The INT instruction makes interrupt service routines accessible to programmers
  - Syntax: "INT imm" with *imm* indicating interrupt number
  - Returning from an ISR is like RET, except it enables interrupts

	Ordinary subroutine	Interrupt service routine
Invoke	CALL	INT
Terminate	RET	IRET

- Fault and Traps:
  - When an instruction causes an exception and is retried after handling it, the exception is called faults (e.g. page fault)
  - When control is passed to the next instruction after handling an exception or interrupt, such exception is called a trap (e.g. division overflow)

Privileged Mode

Privilege Levels

- The difference between kernel mode and user mode is in the privilege level
- The 80386 has 4 privilege levels, two of them are used in Linux
  - Level 0: system level (Linux kernel)
  - Level 3: user level (user processes)
- The CPL register stores the current privilege level and is reset during the execution of system calls
- Privileged instructions, such as LIDT that set interrupt tables can execute only when CPL = 0

Stack Issues

- System calls have to use different stack since the user processes will have write access to them (imagine a process passing the stack pointer as a parameter forcing the system call to overwrite its own stack
- There is a different stack pointer for every privilege level stored in the task state segment

Summary: Types of Interrupts

- **Hardware vs Software**
  - **Hardware: I/O, clock tick, power failure, exceptions**
  - **Software: INT instruction**
- **External vs Internal Hardware Interrupts**
  - **External interrupts are generated by CPU's interrupt pin**
  - **Internal interrupts (exceptions): div by zero, single step, page fault, bad opcode, stack overflow, protection, …**
- **Synchronous vs Asynchronous Hardware Int.**
  - **Synchronous interrupts occur at exactly the same place every time the program is executed. E.g., bad opcode, div by zero, illegal memory address.**
  - **Asynchronous interrupts occur at unpredictable times relative to the program. E.g., I/O, clock ticks.**

Summary: Interrupt Sequence

- Device sends signal to interrupt controller.
- Controller uses IRQ# for interrupt # and priority.
- Controller sends signal to CPU if the CPU is not already processing an interrupt with higher priority.
- CPU finishes executing the current instruction
- CPU saves EFLAGS & return address on the stack.
- CPU gets interrupt # from controller using I/O ops.
- CPU finds "gate" in Interrupt Description Table.
- CPU switches to Interrupt Service Routine (ISR). This may include a change in privilege level. IF cleared.

Interrupt Sequence (cont.)

- ISR saves registers if necessary.
- ISR, after initial processing, sets IF to allow interrupts.
- ISR processes the interrupt.
- ISR restores registers if necessary.
- ISR sends End of Interrupt (EOI) to controller.
- ISR returns from interrupt using IRET. EFLAGS (inlcuding IF) & return address restored.
- CPU executes the next instruction.
- Interrupt controller waits for next interrupt and manages pending interrupts.