The Programmable Interrupt Controller
Interrupt Controller

- Before: The 8086 processor has two hardware interrupt signals
- We’ve seen at least 2 interrupt sources.

“Decides from which vector table location to load ISR address”
Interrupt Vectors: Deciding which ISR to run

• Auto-vectored interrupts: vector predefined as part of CPU design

  – For each HW signal, CPU goes to a particular interrupt-type in vector table.

  – **8086 Example**: NMI (auto-vectored -> Interrupt-type 2)
    • NMI asserted, 8086 executes INT type 2 interrupt behaviour:
      – save processor state
      – obtain ISR address from vector 2 (memory address 0:8)
      – execute type 2 ISR

• More than one device shares the NMI signal? (e.g. RAM and power supply)
  – NMI ISR must check (poll) each device (which one caused the interrupt?)
Interrupt Vectors: Deciding which ISR to run

• Vectored interrupts: vector determined during systems design.

  – CPU performs “interrupt-acknowledge” cycle
    -> reads interrupt-type from data bus
    • Interrupting device can provide interrupt-type
    • 8086: interrupt controller (IC)

  – Vectored interrupts: robust method for multiple devices connected
to single interrupt line (no polling!)
    • Interrupt-type: mapping to unique ISR for each device
    • Interrupt controller acts as a multiplexer

  – **8086 Example:** INTR is a vectored interrupt
Interrupt Controller

• Interrupt controller acts as a funnel for multiple device interrupts
  – Allows many devices to share the 8086’s single INTR signal
Interrupt Controller – INTA cycle

- Interrupt Acknowledge Cycle (INTR line asserted by IC)
  - CPU and interrupt controller “handshake” **in hardware**
    - **No** software involved
  - Interrupt controller “knows” which device caused INTR signal
  - Interrupt controller “tells” CPU a unique *interrupt-type* associated with interrupting device
    - It writes the interrupt-type on the data bus
  - CPU uses interrupt-type to execute appropriate interrupt behaviour (i.e. device’s ISR)
Interrupt Controller

Interrupt controllers can have complex behaviour

- *programmable* (select mode of operation)
  - must be initialised before use
  - PIC: I/O devices that are read and written to.

- Example: specific interrupt-type associated with each interrupting device is often programmable
The Intel Programmable Interrupt Controller (PIC)

In 80x86 based PCs, the interrupt controller used is the **Intel 8259A**

It supports 8 device inputs: $IR_0 \rightarrow IR_7$
Daisy-Chaining or Cascading the PIC

- Maximum configuration: 1 master PIC and up to 8 slave PICs, allowing up to 64 devices to generate interrupts
  - Modern PC’s have a master + (at least) one slave

```
+-------------------+    +-------------------+    +-------------------+
|     CPU           |    |     Master PIC    |    |     Slave PIC     |
|                   |    |                   |    |                   |
|       INTR        |    |                   |    |                   |
|                   |    +-------------------+    +-------------------+
|                               |    | Interrupting Device|
|                               |    | Interrupting Device|
|                               |    | Interrupting Device|
|                               |    +-------------------+    +-------------------+
|                               |    | slave PIC’s INTR signal |
|                               |    +-------------------+    +-------------------+
|                               |    | interrupt signals from devices |
|                               |    | value exchanged during interrupt acknowledge hand-shake |
```
The PC configuration of 80x86 and 8259 PIC

(NB The PC is one particular configuration of 8086 and PIC)

During power-up, BIOS programs (initialises) the master PIC:
IR0 → IR7 mapped to interrupt types 08h → 0Fh
PC Example : Keyboard

- Assume interrupts enabled (IF = 1)

- Keyboard hardware asserts IR1 at PIC, PIC generates INTR signal to CPU
  - Interrupt acknowledge: PIC identifies interrupt source as type 9
  - CPU executes the INT 9h behaviour
    - Saves the flags
    - Clears IF and TF (Disabling interrupts at processor)
    - Saves CS and IP
    - Reads interrupt-type = 9h from the Data bus and vectors to ISR pointed to by double word at 0:9h*4

- Execution of ISR 9 caused by hardware interrupt mechanism
  - No software involved in the invocation of ISR 9!
Some (as yet) Unanswered Questions:

1. Two devices generate interrupts at the same time: which ISR executed first?

   order?

2. CPU executing ISR; second device interrupts: when should the second ISR be executed?

   interrupting an ISR?

   not possible unless ISR re-enables interrupts! i.e. IF = 1
Interrupt Priority

- Interrupting Devices are assigned priorities
  - Higher priority devices take precedence over lower priority ones
  - Priority applied whenever interrupts coincide
    - Multiple interrupts occur at the same time
    - New interrupts occur while processing ISR of previous interrupts.

- Typically, interrupt controllers manage priority issues.
  - In PC’s
    - Devices have pre-configured connections to PIC
      - Timer always IR0 and Keyboard is always IR1
    - DOS programs 8259A to assign priority based on device connection
      - IR0 == highest priority and IR7 == lowest priority

The lower the number, the higher the priority
Interrupt Priority Scenarios

1. Two devices generate interrupts at the same time: which ISR should be executed first?
Interrupt Priority Scenarios

2. CPU executing ISR; second device interrupts, when should the second ISR be executed?

- Two inputs to PIC: IRm and IRn where m < n
  - m higher priority than n!

- Device n asserts IRn; PIC generates interrupt. Device m asserts IRm; PIC generates another interrupt!

- PIC will try to allow higher priority interrupt to interrupt a lower priority ISR!
  - Second interrupt will not be recognized by processor until interrupts are re-enabled (IF = 1)

When is this?
# Interrupt Priority Scenarios

- Device $n$ asserts IR$_n$ (low priority) while IR$_m$ ISR (high priority) running.

<table>
<thead>
<tr>
<th>No STI in ISR</th>
<th>With STI in ISR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main INT 8 ISR INT 9 ISR Main</td>
<td>INT 8 ISR INT 9 ISR</td>
</tr>
<tr>
<td>IR0 IR1</td>
<td>IR0 IR1</td>
</tr>
</tbody>
</table>

- Low priority assertion “remembered” (latched) by PIC until high priority ISR finished (regardless of interrupts being enabled/disabled)

- Then, PIC generates another interrupt on behalf of device $n$

- Two More Questions:
  1. How many interrupts can the PIC remember?
  2. How does the PIC know when higher priority ISR is finished?
Pending Interrupts

- “Pending” Interrupt: interrupt signal latched somewhere in the system, but not yet been acknowledged by the processor
  - Interrupts can be pending at device and/or at the PIC

- Example: The Intel 8259 has an internal 8-bit register
  - one bit per IR input
  - When IR is asserted, associated bit is set
  - When interrupt on IR acknowledged, associated bit is cleared
  - In summary, the PIC has 1-bit memory for each IR
    - It can remember up to 1 pending interrupt for each IR
End-of-Interrupt (EOI)

- After sending interrupt to processor, PIC needs to know when it is safe to generate a lower priority interrupt
  - PIC requires feedback from the CPU

- End Of Interrupt (EOI): a command sent to PIC from the CPU
  - Not part of the INTA cycle; not done in hardware
  - Software command: i.e. something your program must do.
PIC Programmer’s Model

- PIC: I/O Device (I/O port addresses)
  - Two 8-bit ports:

  **Interrupt Mask Register** (Port 21H) read/write
  - Enable/disable individual interrupts at the PIC
  - bit i = 1 IRi is masked (not recognized by the PIC)
  - bit i = 0 IRi is unmasked (recognized by the PIC)

    Beware: mask at PIC \( \rightarrow \) bit = 1 mask at processor \( \rightarrow \) IF = 0

**Command Register** (Port 20H) - write-only
- Write 20H to inform PIC of end of interrupt (EOI)

Simple version here – on a need-to-know basis.
Complete details in ELEC 3601