ECE391- Computer System Engineering

Lecture 9

Programable Interrupt Controller

University of Illinois at Urbana- Champaign
Fall 2021

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Office of the Provost

Dear Colleagues,

We have shared several updates about Fall 2021 in the past several weeks as we monitor the progress with managing the pandemic and respond to new science-based COVID-19 guidance from the CDC, IDPH, CUPHD and our own SHIELD team. Below is a condensed version of what we need to know as we prepare for in-person classes in a few weeks.

Face Coverings

- Everyone (faculty, staff, students, visitors) is required to wear a face covering in university facilities. Read more about face coverings and face shields here.
- If a student is not wearing a face covering in your class, ask them to put one on. If they refuse, dismiss the class and report the student to the Office for Student Conflict Resolution for further discipline by filling out this form. Call UIPD, 217-333-1216, only if an individual becomes belligerent, disruptive and threatening.

Announcements

- PS2 Posted - Committed to the master (main) branch on GitLab by 5:59PM on 9/21
- MP2 Posted - All checkpoints should be committed to the master(main) branch on GitLab by:
  - Checkpoint 1: 5:59PM on 10/5
  - Final Checkpoint: 5:59PM on 10/12

ECE391 EXAM 1

- EXAM I – Wednesday, September 29th, 7:00pm-9:00 PM
  - Location: ECEB 1002
- Online details to follow, but expect it to be proctored and at same time.
- NO Lecture on Tuesday, September 28
- Review Session
  - ?

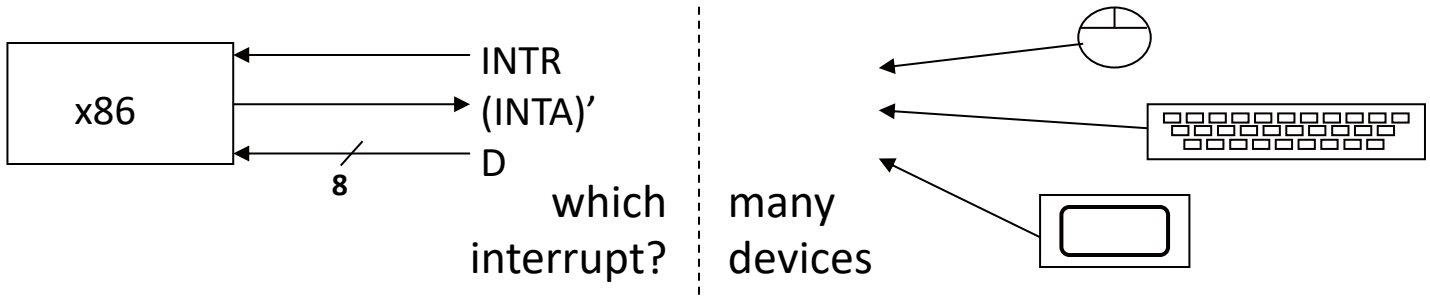
Topics covered by EXAM 1

- Material covered in lectures (Lecture1 – Lecture10)
  - x86 Assembly
  - C (e.g., Calling Convention)
  - Synchronization
  - Interrupt control (using PIC)
- Material covered in discussions
- MP1

Lecture Topics

- Programmable interrupt controller (PIC)
  - motivation & design
  - hardware for x86
  - Linux abstraction of PIC

PIC Motivation and Design

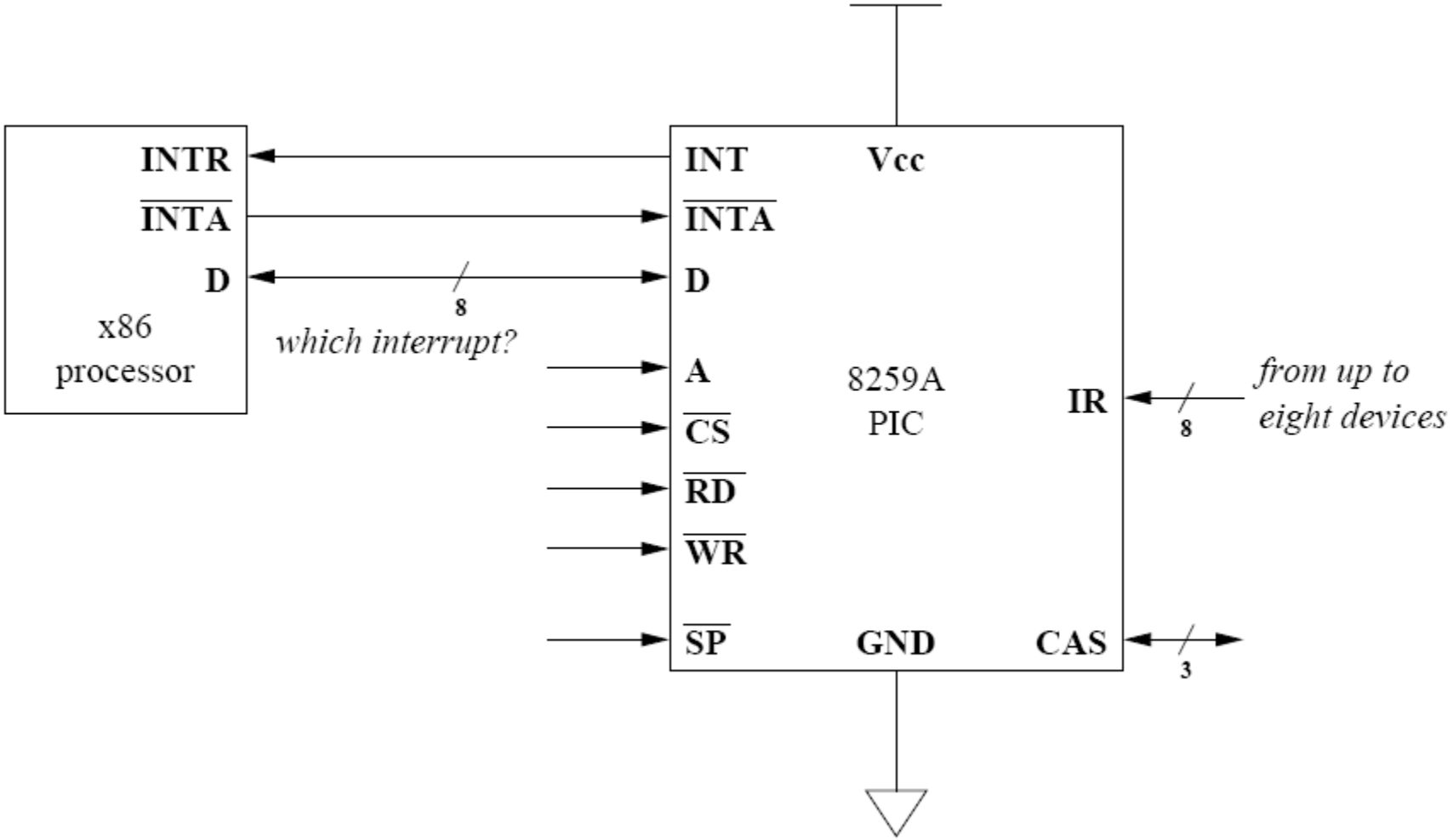


- How do we connect devices to the processor's interrupt input?
- An OR gate? why not?

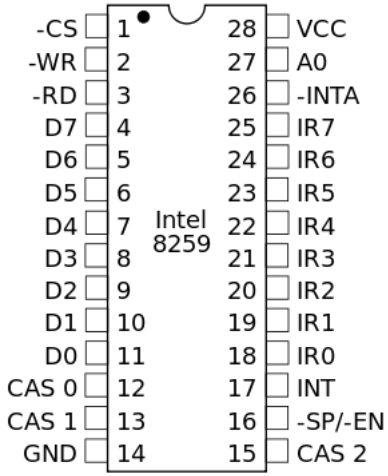
PIC Motivation and Design (cont.)

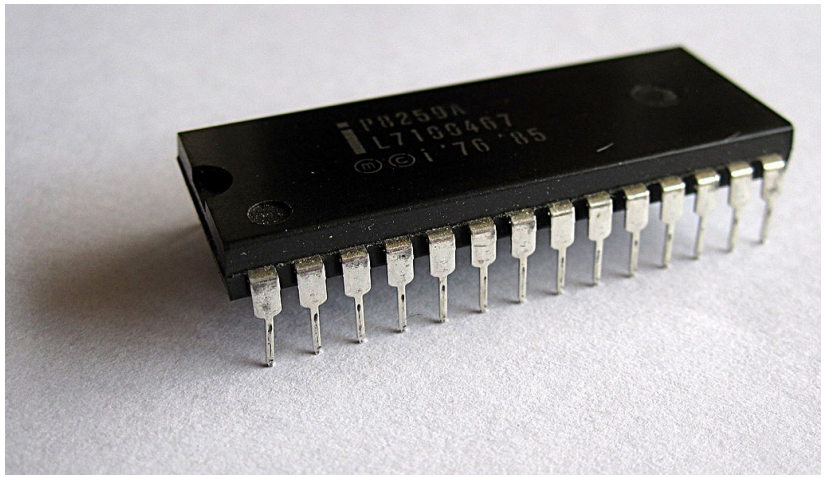
- who writes the vector # ?
  - possible to build arbiter, but…
  - what if more than one raised interrupt?
- extra work for processor to query all devices
  - must execute interrupt code for device that raised interrupt
  - could have been more than one device
  - no way to tell with OR gate
- not all devices support query
  - many devices too simplistic to support query
  - and operations (e.g., reading from port) may not be idempotent
- might be nice to have concept of priority and preemption, i.e., interrupting an interrupt handler

8259A Programmable Interrupt Controller (PIC)



Intel 8259





[images: Wikipedia users German, Nixdorf]

Logical Model of PIC Behavior

- Watch for interrupt signals
  - up to eight devices
  - one interrupt line each

Logical Model of PIC Behavior

- Track which devices/input lines are currently in service by processor
  - i.e., processor is executing interrupt handler for that interrupt
  - using internal state

Logical Model of PIC Behavior (cont.)

- When a device raises an interrupt
  - If priority is higher than those of in-service interrupts
    - report the highest-priority raised to the processor
    - mark that device as being in service
  - else
    - do nothing

Logical Model of PIC Behavior (cont.)

- When processor reports EOI (end of interrupt) for some interrupt
  - remove the interrupt from the in-service mask
  - check for raised interrupt lines that should be reported to processor

Logical Model of PIC Behavior (cont.)

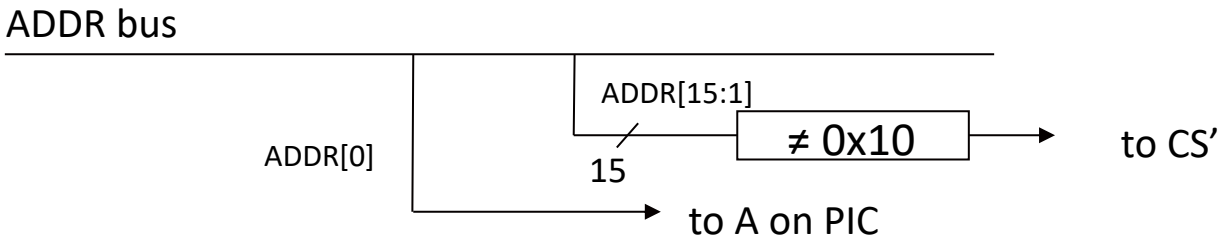
- Protocol for reporting interrupts
  - PIC raises INTR
  - processor strobes INTA’ (active low) repeatedly
    - creates cycles for PIC to write vector to data bus
    - (must follow spec timing! PIC is not infinitely fast!)
  - processor sends EOI with specific combinations of A & D inputs (A is from address bus, D is from data bus)

Logical Model of PIC Behavior (cont.)

- What about A, CS', RD', and WR' ?
  - A = address match for ports
  - CS' = chip select (does processor want PIC to read/write?)
  - RD' and WR' defined from processor's point of view
    - RD' = processor will read data (vector #) from PIC
    - WR' = processor will write data (command, EOI) to PIC

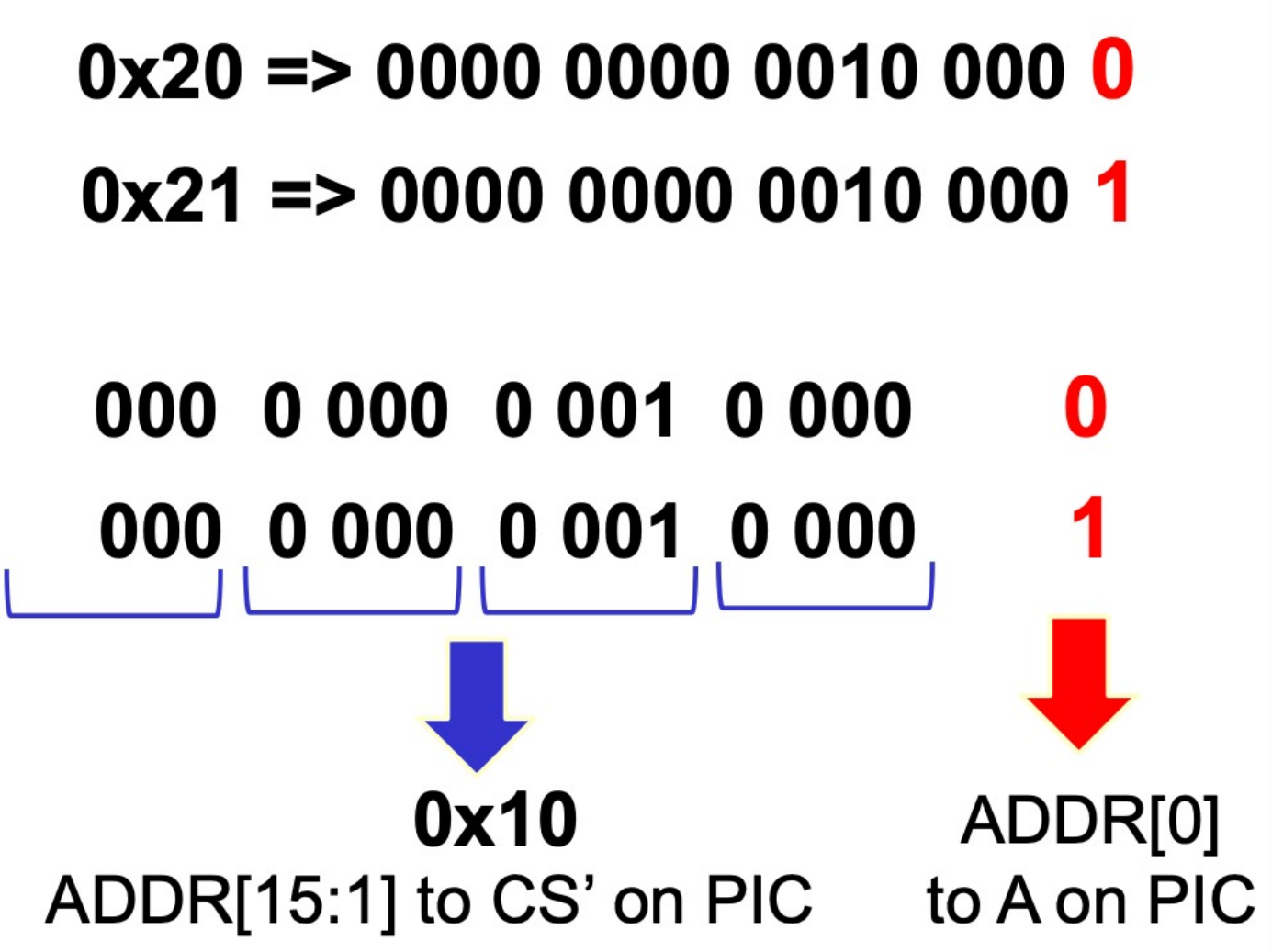
Logical Model of PIC Behavior (cont.)

- Map to ports 0x20 & 0x21
  - given ADDR bus
  - logic to form A and CS' inputs to PIC



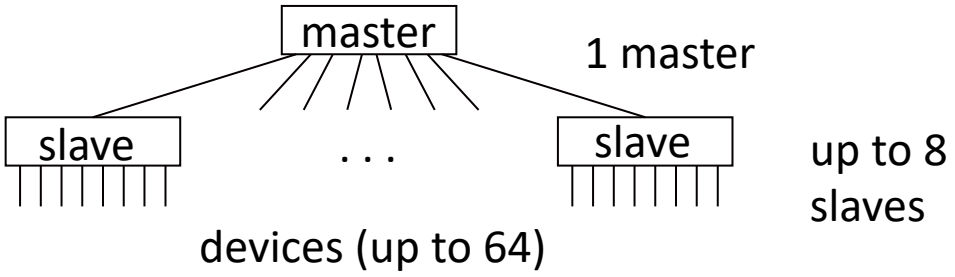
- Remember
  - the PIC is asynchronous!
  - all interactions have timing constraints
  - see specifications for details

Logical Model of PIC Behavior (cont.)

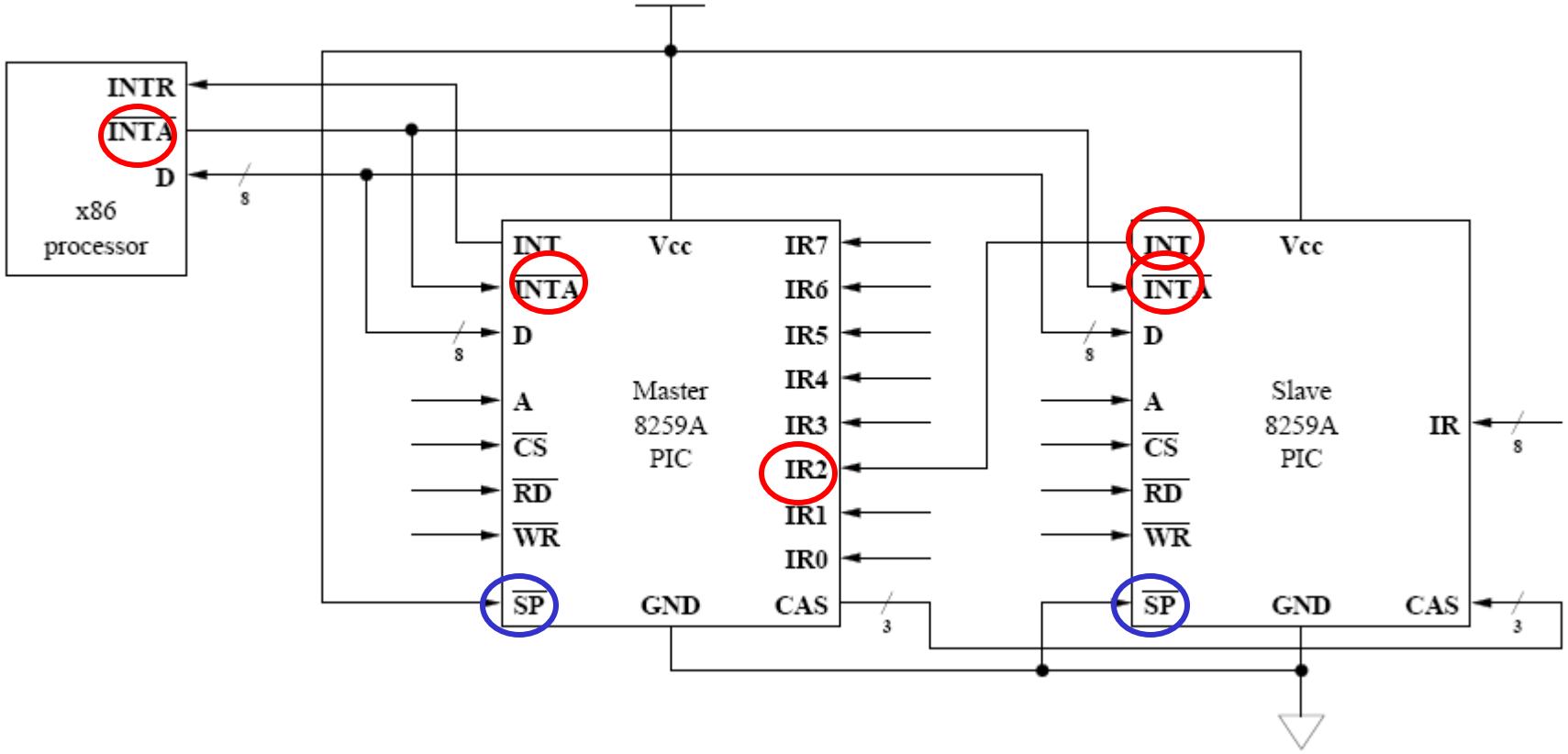


Logical Model of PIC Behavior (cont.)

- Are eight devices enough? No? What then?
  - hook 2, 3, or 4 PICs to processor?
  - [same problem as before!]
  - design a big PIC?
  - [waste of transistors; not cheap in 8259A era]
  - design several PICs?
  - [waste of humans! (and production costs)]
- Better answer: cascade



Cascade Configuration of PICs



Cascading (cont.)

- Previous figure showed x86 configuration of two 8259A's
  - master 8259A mapped to ports 0x20 & 0x21
  - slave 8259A mapped to ports 0xA0 & 0xA1
  - slave connects to IR2 on master
- Question
  - prioritization on 8259A: 0 is high, 7 is low
  - what is prioritization across all 15 pins in x86 layout?
  - (highest) M0…M1…S0…S7…M3…M7 (lowest)

Cascading (cont.)

- Which PIC should write interrupt vector when when processor strobes INTA'?
  - Slave doesn't know which interrupt master reported to CPU
  - Master doesn't know which device connected to slave raised the slave INT line.
- Master tells which slave they should write
  - CAS bus transmits number (3-bits)
  - SP for differentiating between master and slave

Cascading (cont.)

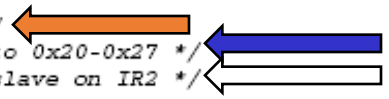
- In Linux (initialization code to be seen shortly)
  - master IR's mapped to vector #'s 0x20 – 0x27
  - slave IR's mapped to vector #'s 0x28 – 0x2F
  - remember the IDT?

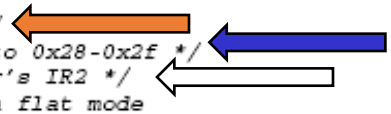
Interrupt Descriptor Table

0x00–0x1F defined by Intel	0x00	division error	
	⋮		
	0x02	NMI (non-maskable interrupt)	
	0x03	breakpoint (used by KGDB)	
	0x04	overflow	
	⋮		
	0x0B	segment not present	
	0x0C	stack segment fault	
	0x0D	general protection fault	
	0x0E	page fault	
	⋮		
0x20–0x27 master 8259 PIC	0x20	IRQ0 — timer chip	example of possible settings
	0x21	IRQ1 — keyboard	
	0x22	IRQ2 — (cascade to slave)	
	0x23	IRQ3	
	0x24	IRQ4 — serial port (KGDB)	
	0x25	IRQ5	
	0x26	IRQ6	
	0x27	IRQ7	
0x28–0x2F slave 8259 PIC	0x28	IRQ8 — real time clock	
	0x29	IRQ9	
	0x2A	IRQ10	
	0x2B	IRQ11 — eth0 (network)	
	0x2C	IRQ12 — PS/2 mouse	
	0x2D	IRQ13	
	0x2E	IRQ14 — ide0 (hard drive)	
	0x2F	IRQ15	
0x30–0x7F	⋮	APIC vectors available to device drivers	
0x80	0x80	system call vector (INT 0x80)	
0x81–0xEE	⋮	more APIC vectors available to device drivers	
0xEF	0xEF	local APIC timer	
0xF0–0xFF	⋮	symmetric multiprocessor (SMP) communication vectors	

Linux 8259A Initialization

```
void init_8259A(int auto_eoi)
{
        unsigned long flags;

        i8259A_auto_eoi = auto_eoi;

        spin_lock_irqsave(&i8259A_lock, flags);

        outb(0xff, 0x21);       /* mask all of 8259A-1 */
        outb(0xff, 0xA1);       /* mask all of 8259A-2 */

        /*
         * outb_p - this has to work on a wide range of PC hardware.
         */
        outb_p(0x11, 0x20);     /* ICW1: select 8259A-1 init */
        outb_p(0x20 + 0, 0x21); /* ICW2: 8259A-1 IR0-7 mapped to 0x20-0x27 */
        outb_p(0x04, 0x21);     /* 8259A-1 (the master) has a slave on IR2 */
        if (auto_eoi)
                outb_p(0x03, 0x21);     /* master does Auto EOI */
        else
                outb_p(0x01, 0x21);     /* master expects normal EOI */

        outb_p(0x11, 0xA0);     /* ICW1: select 8259A-2 init */
        outb_p(0x20 + 8, 0xA1); /* ICW2: 8259A-2 IR0-7 mapped to 0x28-0x2f */
        outb_p(0x02, 0xA1);     /* 8259A-2 is a slave on master's IR2 */
        outb_p(0x01, 0xA1);     /* (slave's support for AEOI in flat mode
                                    is to be investigated) */

        if (auto_eoi)
                /*
                 * in AEOI mode we just have to mask the interrupt
                 * when acking.
                 */
                i8259A_irq_type.ack = disable_8259A_irq;
        else
                i8259A_irq_type.ack = mask_and_ack_8259A;

        udelay(100);            /* wait for 8259A to initialize */

        outb(cached_21, 0x21);  /* restore master IRQ mask */
        outb(cached_A1, 0xA1);  /* restore slave IRQ mask */

        spin_unlock_irqrestore(&i8259A_lock, flags);
}
```





Comments on Linux' 8259A Initialization Code

- What is the auto_eoi parameter?
  - always = 0, not used in Linux
- Four initialization control words to set up the master 8259A
- Four initialization control words to set up the slave 8259A
- ICW1 0 start init, edge-triggered inputs, cascade mode, 4 ICWs
- ICW2 1 high bits of vector #
- ICW3 1 master: bit vector of slaves; slave: input pin on master
- ICW4 1 ISA=x86, normal/auto EOI

Comments on Linux' 8259A Initialization Code (cont.)

- What does the "_p" mean on the "outb" macros?
  - add PAUSE instruction after OUTB; "REP NOP" prior to P4
  - delay needed for old devices that cannot handle processor's output rate
- Critical section spans the whole function; why?
  - avoid other 8259A interactions during initialization sequence
  - (device protocol requires that four words be sent in order)
- Why use _irqsave for critical section?
  - this code called from other interrupt initialization routines
  - which may or may not have cleared IF on processor