CPE300: Digital System Architecture and Design

Fall 2011

MW 17:30-18:45 CBC C316

Input and Output

11072011

http://www.egr.unlv.edu/~b1morris/cpe300/

Outline

- Chapter 8 Overview
- I/O Subsystem
- Programmed I/O
- I/O Interrupts

Chapter 8 Overview

- The I/O subsystem
  - I/O buses and addresses
- Programmed I/O
  - I/O operations initiated by program instructions
- I/O interrupts
  - Requests to processor for service from an I/O device
- Direct Memory Access (DMA)
  - Moving data in and out without processor intervention
- I/O data format change and error control
  - Error detection and correction coding of I/O data

Three Requirements of I/O Design

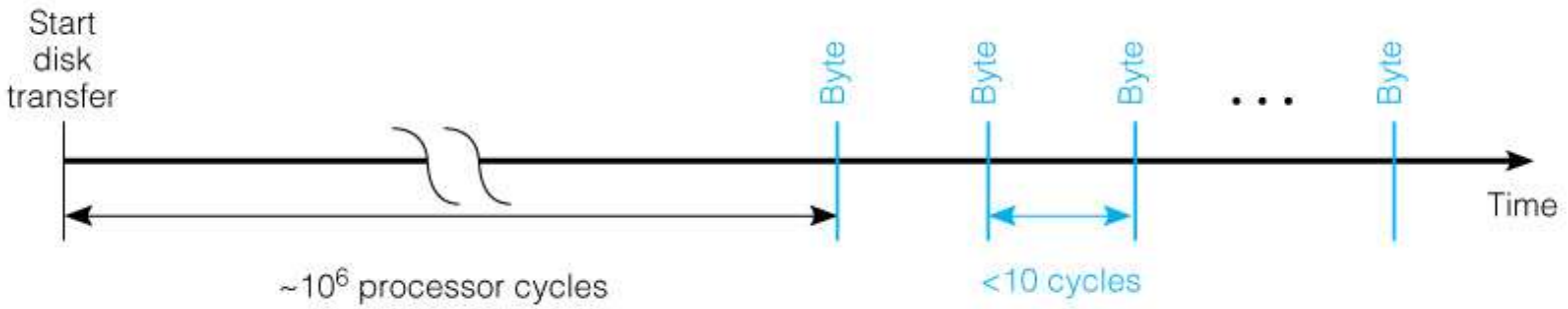
1. Data location
   - Device selection
   - Address of data within device
2. Data transfer
   - Amount of data varies with device and may need to be specified
   - Transmission rate varies greatly with device
   - Data may be input, output, or either
3. Data synchronization
   - Data should only be sent to an output device when it is ready to receive
   - Processor should only read data when it is available from an input device

Location of I/O Data

- Data location may be trivial once device is determined
  - Character from a keyboard
  - Character out to a serial printer
- Location may require searching
  - Record number on a tape drive
  - Track seek and rotation to sector on a disk
- Location may not be simple binary number (unsigned integer address)
  - Drive, platter, track, sector, word on a disk cluster

Data Transfer

- Amount of data transferred by different I/O devices can be quite different
  - Mouse button click vs. large disk copy



  - A keyboard delivers one character about 1/10 second at the fastest
  - Rate varies for disk seek where there is a rotation delay followed by block transfer

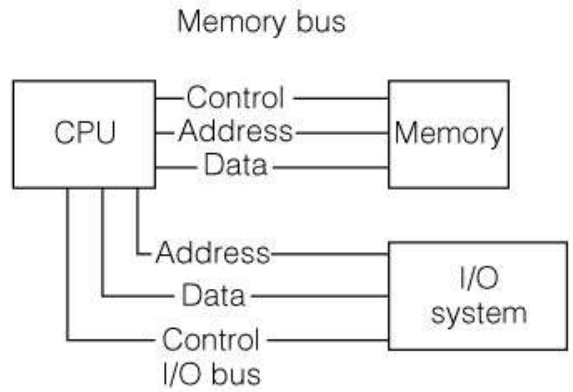
Data Synchronization

- I/O devices are not usually timed by the master clock
  - Rates can differ greatly from processor speed
  - Generally asynchronous
- Processor determines state of device and transfers information at clock ticks
  - I/O status and information must be stable at the clock tick when it is accessed
  - Processor must know when an output device can accept new data
  - Processor must know when an input device is ready to supply new data

I/O Interface Standardization

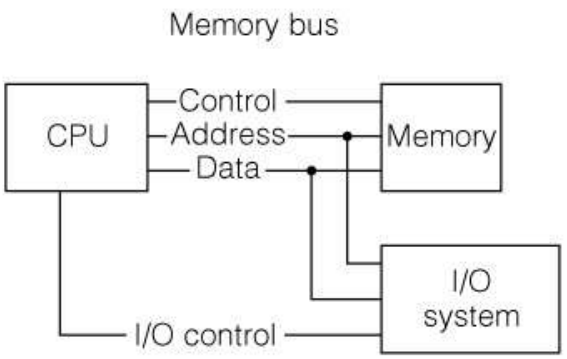
- Standardization provides contract the defines data structure an interface between CPU and I/O device
  - Helps reduce location and synchronization issues
- Data location structure is device dependent
  - Device is selected by processor
  - Location within device is information passed to device
  - Device handles low-level (internal) details and provides a uniform interface for CPU
- Device status bits can be used for synchronization
  - Data available signal from input device
  - Ready to accept data from output device
  - Other forms or synchronization may be required due to speed constraints (interrupts and DMA)

I/O Connections and Bus Structure



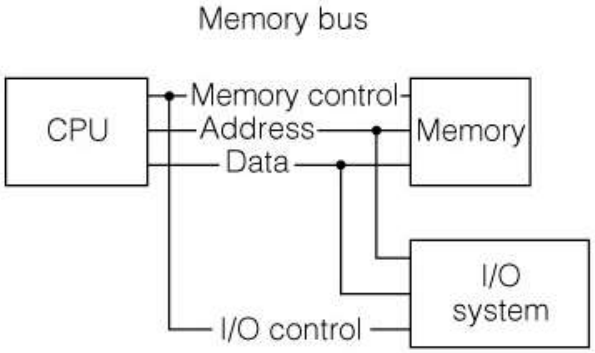
(a) Separate memory and I/O buses (isolated I/O)

- Isolated I/O
- Allows tailoring bus to its purpose
- Requires many connections to CPU (pins)



(b) Shared address and data lines

- Shared I/O
- Memory and I/O access can be distinguished
- Timing and synchronization can be different for each subsystem



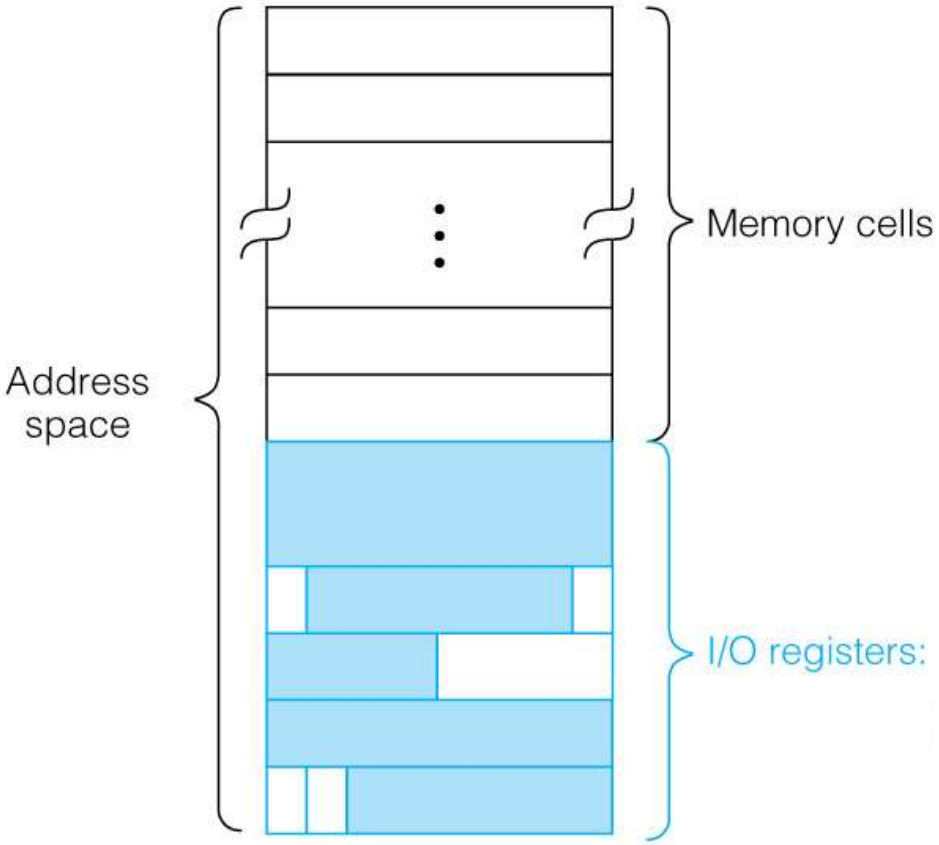
(c) Shared address, data, and control lines (memory-mapped I/O)

- Memory-mapped I/O
- Standardized data transfer
- Least expensive option

Memory Mapped I/O

- Combined memory control and I/O control lines for a single unified bus
- I/O device registers appear to processor as memory addresses
- Reduces the number of connections to the processor chip
  - Increased generality may require more control signals
- Standardizes data transfer to and from processor
  - Asynchronous operation is optional with memory but demanded by I/O devices

Memory Mapped I/O Address Space

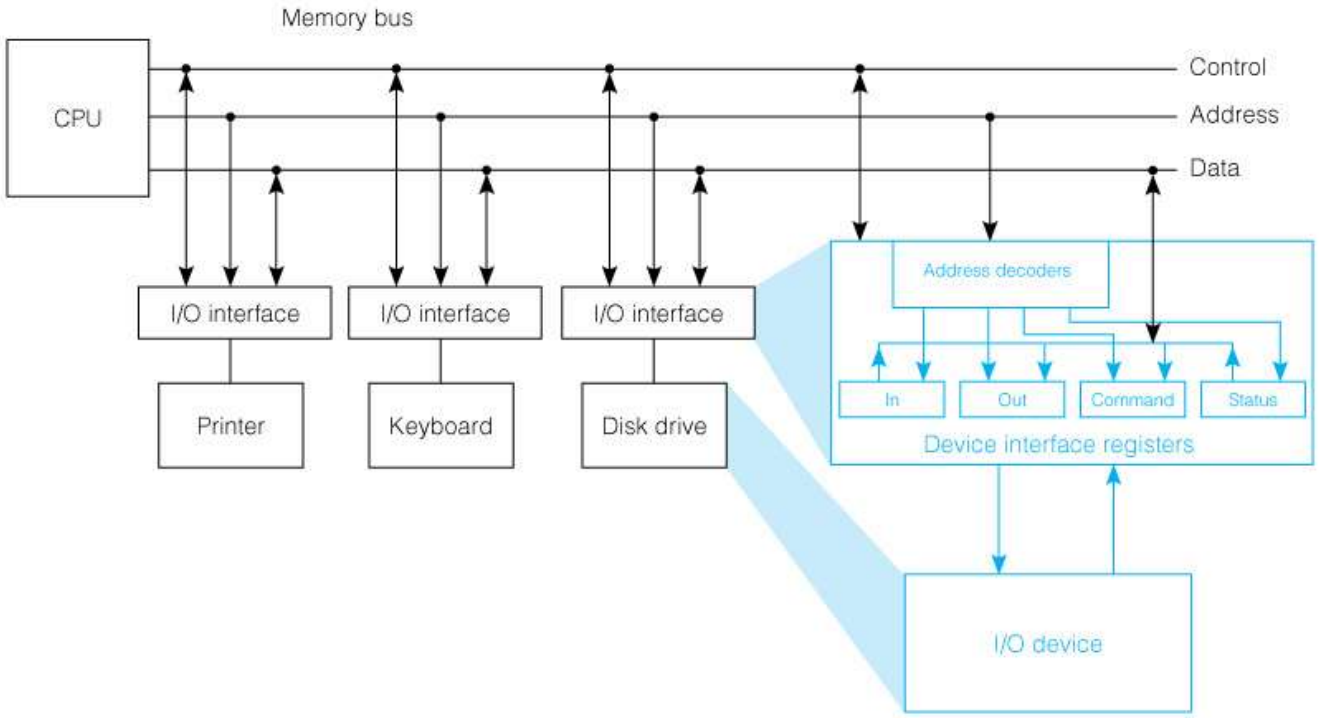


- Memory divided to reserve space for I/O registers
- I/O registers physically distributed between different device interfaces
  - Only a small fraction in use in a system
- Not all bits of memory word needed for a particular device register
  - Ignore unused bits

Programmed I/O

- Device requirements
  - Operations take many instruction execution times
  - One word data transfers – no burst data transmission
- Process has time to test device status bits, write control bits, and read/write data at the required device speed
  - Example status bits
    - Input data ready, output device busy or offline
  - Example control bits
    - Reset device, start read/write
- Define I/O interface to match the speed and protocol of processor bus with that required by the I/O device

Programmed I/O Device Interface Structure



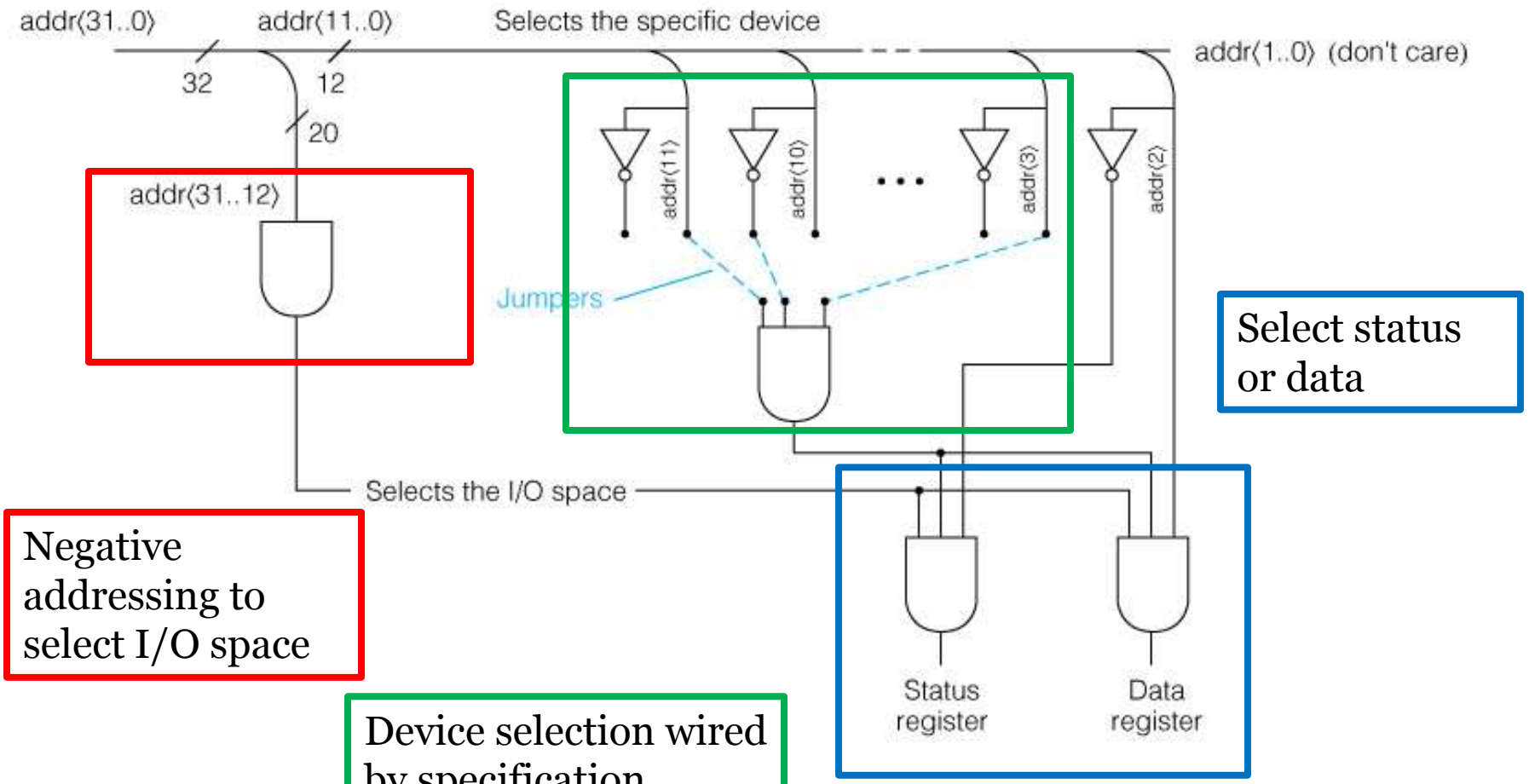
- I/O interface specifies connections between device and control, address, and data bus
  - Decodes addresses just for registers of specific device
  - Control bits from processor to device stored in command register
  - Control signals from device to CPU stored in status registers

SRC I/O Ports

- Addresses above `0xFFFF000` are reserved for I/O registers
  - 1024 32-bit I/O registers
  - Negative displacement addressing
    - oxFFFFFFFF – oxFFFF0000
- Ports arbitrarily chosen within I/O space (need specifications)

Port Name	Hex Address	Name
CICTL	0xFFFFF300	Keyboard control port
CIN	0xFFFFF304	Keyboard data port
COSTAT	0xFFFFF110	Printer status port
COUT	0xFFFFF114	Printer data port
LSTAT	0xFFFFF130	Line printer status
LOUT	0xFFFFF134	Line printer data port
LCMD	0xFFFFF138	Line printer command port

SRC I/O Register Decoder

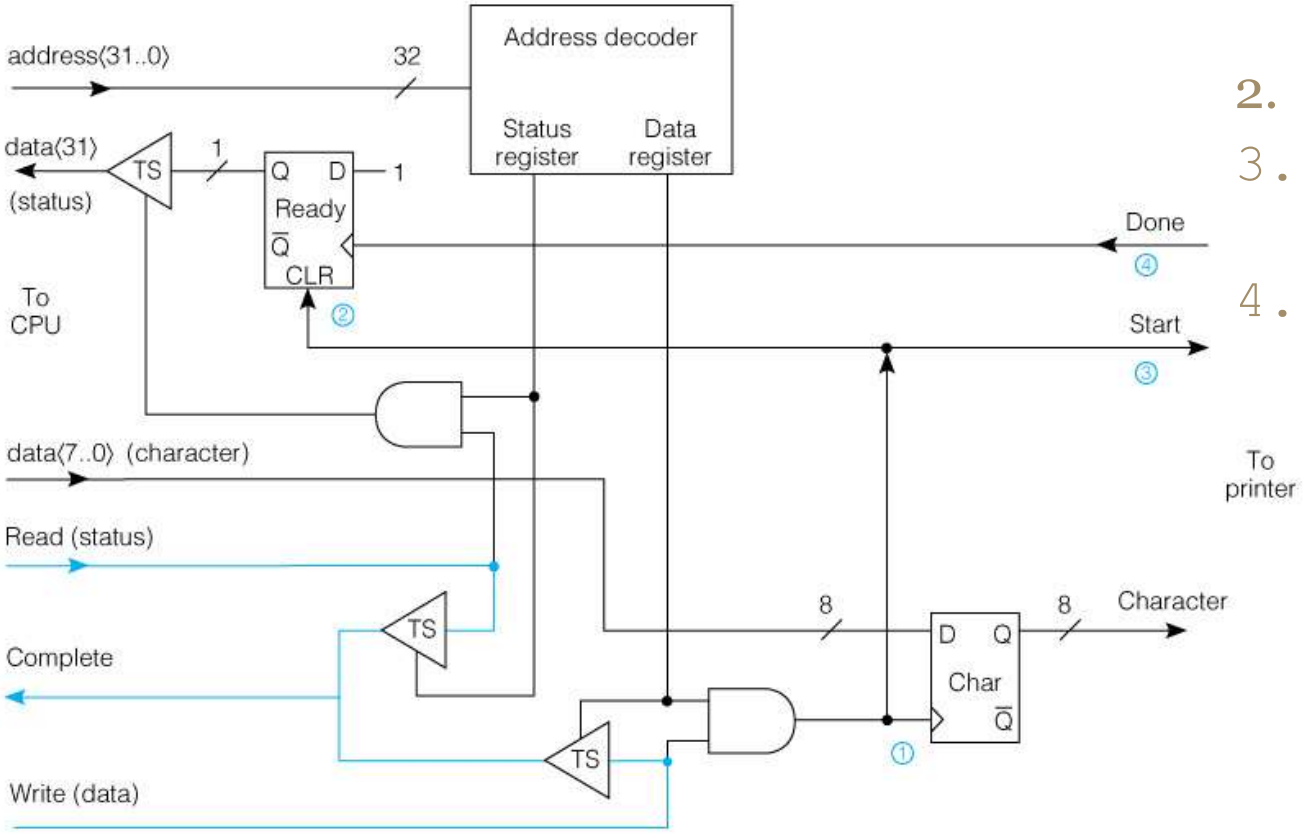


Negative addressing to select I/O space

Device selection wired by specification

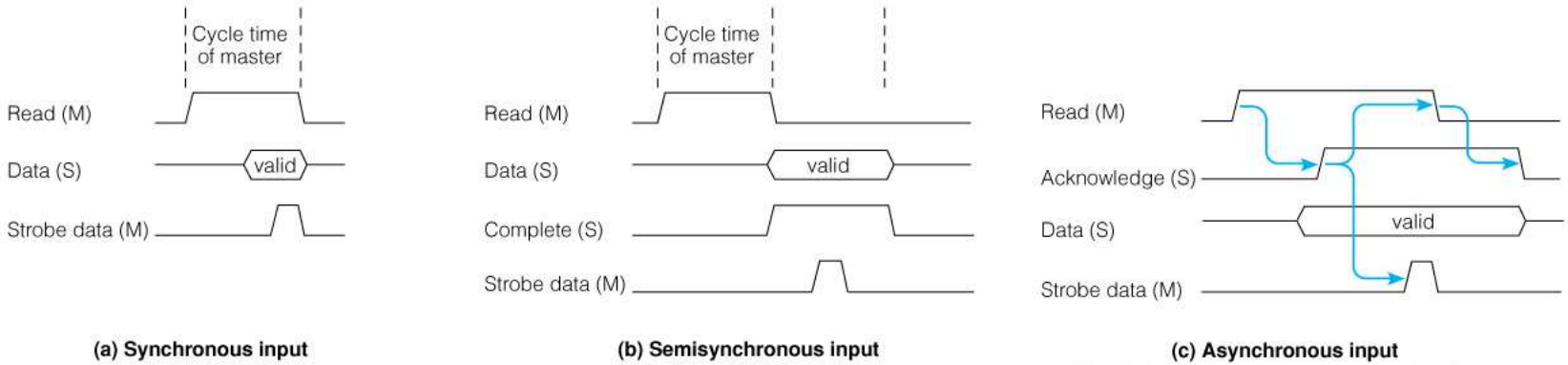
Select status or data

SRC Character Output Interface



1. Output initiated with write to `Char` register
2. Clears `Ready` register
3. `Start` signal sent to printer
4. `Done` signal from printer sets `Ready`

Synchronization of Data Input

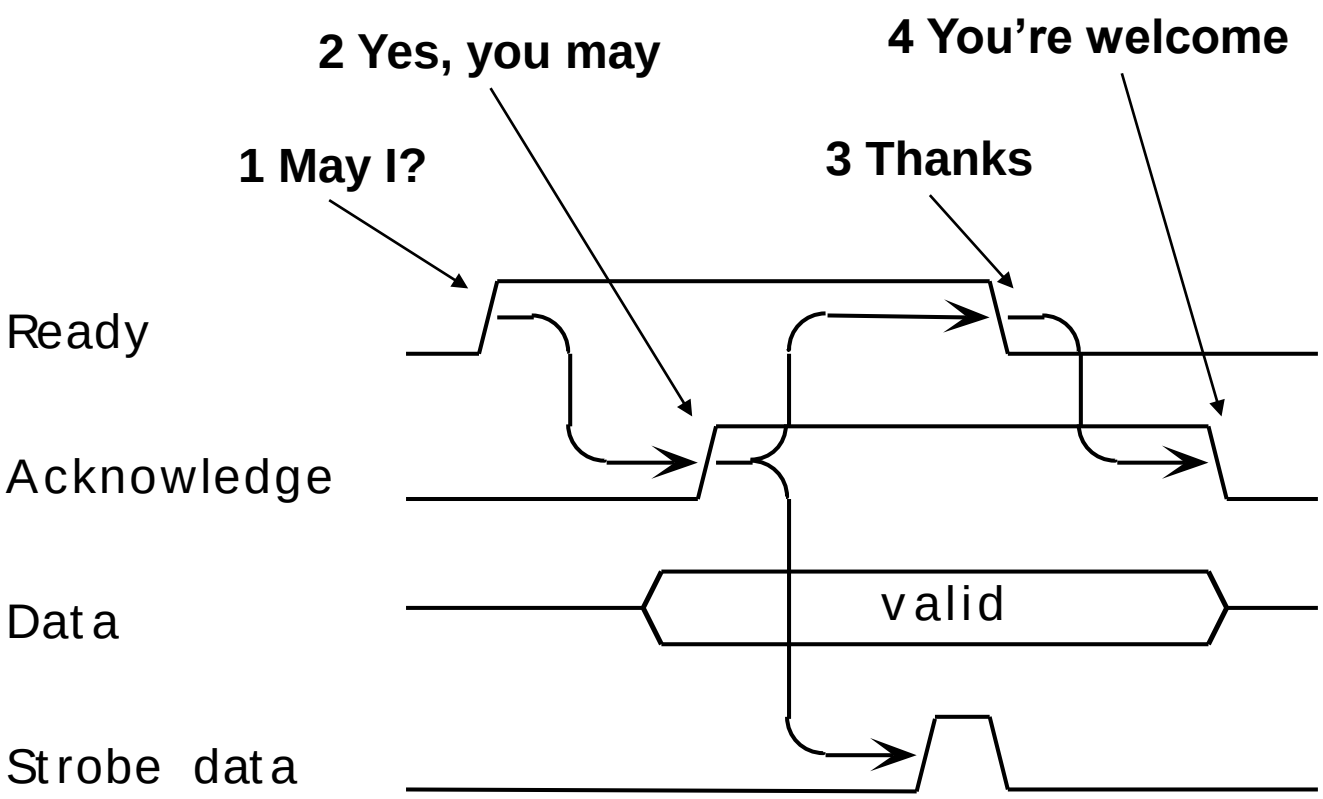


- Synchronous input
- Register-to-register transfer
- Data strobed at end of cycle

- Semi-synchronous input
- Memory-to-CPU transfer with few memory cycles
- `Complete = Done` signal

- Asynchronous input
- Useful for I/O because of differences in speed
- Uses a hardware handshaking protocol

Asynchronous Data Input



- Hardware handshake

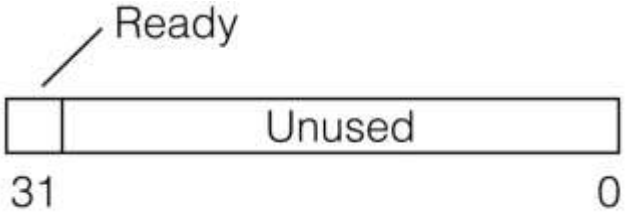
1. Ask for data
2. Data is made valid and request acknowledged
3. Data received and `Ready` lowered
4. `Acknowledge` lowered to complete transaction

Programmed I/O Device Driver Example

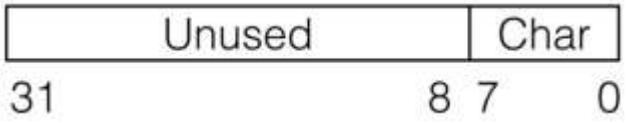
- Software to manage character output
- Device requirements
  - 8 data lines for bits of an ASCII character
  - Start signal to begin operation
  - Data bits must be held until device returns `Done`
- Design decisions for matching bus to device
  - Use low order 8 bits of word for character
  - Make loading of character register signal Start
  - Clear `Ready` status bit on `Start` and set it on `Done`
  - Return `Ready` as sign of status register for easy testing

Character Output Program Fragment

Status register COSTAT = `0xFFFFF110`



Output register COUT = `0xFFFFF114`




```
        lar   r3, Wait   ;set branch target for wait
        ld    r2, Char   ;get character for output
Wait:   ld    r1, COSTAT ;read device status register
        brpl  r3, r1     ;test for ready otherwise repeat
        st    r2, COUT   ;output character and start device
```

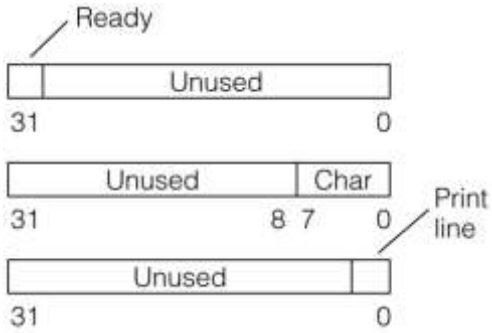
- For readability: I/O registers are all caps., program locations have initial cap., and instruction mnemonics are lower case
- Polling loop to check if device is ready
  - A 10 MIPS SRC would execute 10,000 instructions waiting for a 1,000 character/sec printer

80 Char Line Printer Code Fragment

Status register LSTAT = `0xFFFFF130`

Output register LOUT = `0xFFFFF134`

Output register LCMD = `0xFFFFF138`




```
        lar     r1, Buff    ;set pointer to char buffer
        la      r2, 80      ;init char counter
        lar     r3, Wait    ;init branch target
Wait:   ld      r0, LSTAT   ;read ready bit
        brpl    r3, r0      ;check until ready
        ld      r0, 0(r1)   ;get next char from buffer
        st      r0, LOUT    ;send char to printer
        addi    r1, r1, 4   ;increment char pointer (32-bit word)
        addi    r2, r2, -1  ;decrement char counter
        brnz    r3, r2      ;if more char loop
        la      r0,1        ;set print line command
        st      r0, LCMD    ;send command to printer
```

Poll char

Complete 80 char line

Print line

Multiple Input Device Driver

- 32 low speed input devices
  - Keyboards at ~10 characters/sec
  - Max rate of one every 3 msec
- Each device has control/status register
  - Only `Ready` status bit (bit 31) is used
  - Driver works by polling (repeated testing) of `Ready` bits
- Each device has 8-bit input data register
  - Bits 7..0 of 32-bit input word hold character
- Software controlled by pointer and `Done` flag
  - Pointer to next available location in input buffer
  - Device `Done` is set when char received from device
  - Device is idle until other program (main) clears done

32 Input Device Polling Driver

0xFFFF300	Dev0CICTL
0xFFFF304	Dev0CIN
0xFFFF308	Dev1CICTL
0xFFFF30C	Dev1CIN
0xFFFF310	Dev2CICTL
0xFFFF314	Dev2CIN
	…

- 32 Pairs of control/status and input data registers
- `r0` – working register
- `r1` – input char
- `r2` – device index
- `r3` – no device active

Initialization

```
CICTL   .equ    FFFFF300H   ;first input control reg
CIN     .equ    FFFFF304H   ;first input data reg
CR      .equ    13          ;ASCII carriage return
Bufp:   .dcw    1           ;location for first buffer pointer
Done:   .dcw    63          ;done flags and rest of pointers
Driver: lar     r4, Next    ;branch target for next character
        lar     r5, Check   ;check device is active
        lar     r6, Start   ;start polling pass through 32 devices
```

32 Input Device Polling Code

```
Start:  la    r2, 0          ;Point to first device, &
        la    r3, 1          ;  set all inactive flag.
Check:  ld    r0, Done(r2)   ;If device not still active,
        brmi  r4, r0         ;  go advance to next.
        la    r3, 0          ;Clear the all inactive flag.
        ld    r0, CICTL(r2)  ;Get device ready flag, & go
        brpl  r4, r0         ;  move to next if not ready.
        ld    r0, CIN(r2)    ;Get character and
        ld    r1, Bufp(r2)   ;  correct bufer pointer, &
        st    r0, 0(r1)      ;  store character in buffer.
        addi  r1, r1, 4      ;Advance character pointer,
        st    r1, Bufp(r2)   ;  and return it to memory.
        addi  r0, r0,-CR     ;If not carriage return,
        brnz  r4, r0         ;  go advance to next device.
        la    r0, -1         ;Set done flag to -1 on
        st    r0, Done(r2)   ;  detecting carriage return.
Next:   addi  r2, r2,8       ;Advance device pointer, and
        addi  r0, r2,-256    ;  if not last device,
        brnz  r5, r0         ;  go check next one.
        brzr  r6, r3         ;If any active, make new pass.
```

Characteristics of Polling Device Driver

- All devices active and always have a char ready
  - 32 bytes of input in 547 instructions
  - Data rate of 585 KB/s in 10 MIPS CPU
- CPU just misses setting of `Ready`
  - 538 instructions executed before testing device again
  - 53.8 μsec delay requires a device run below 18.6k chars/sec to avoid risk of losing data
    - Keyboards are slow enough

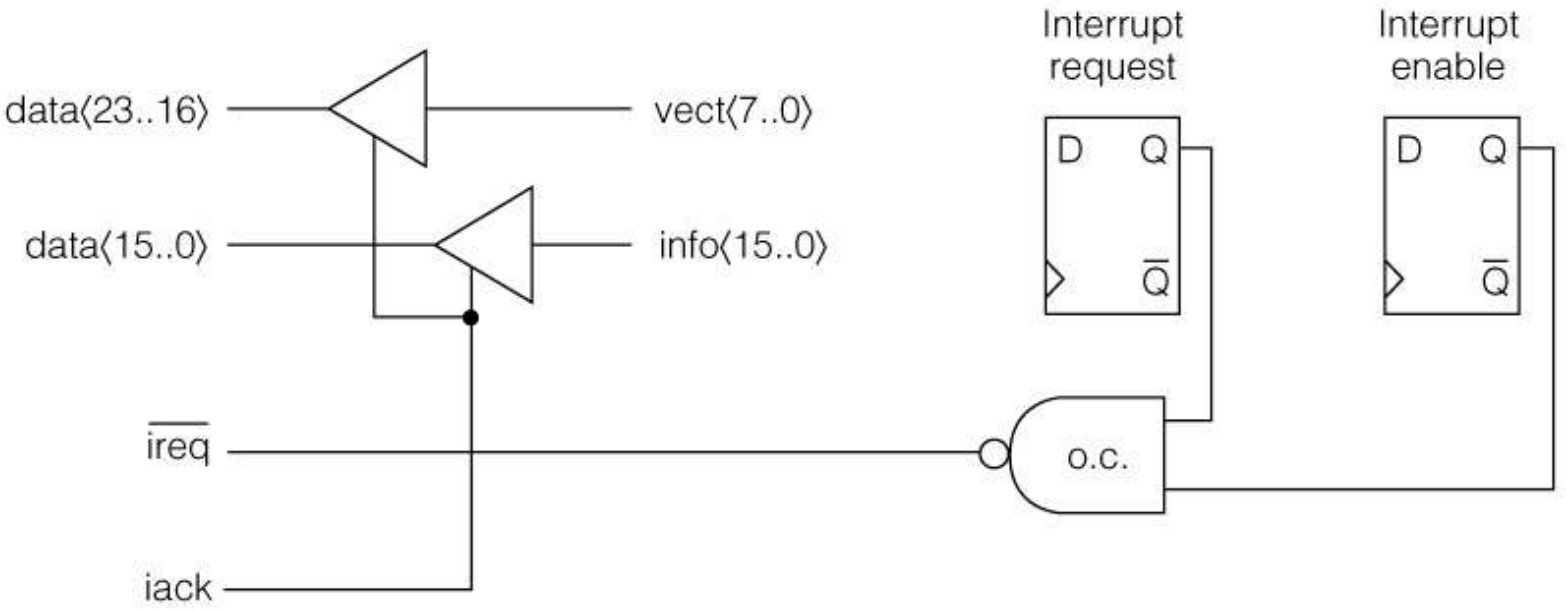
Key Concepts: Programmed I/O

- Appropriate when speed of device does not overwhelm CPU processing ability
- Processor will likely spend significant time in busy-wait loops (polling)
- Hardware interface usually consists of a buffer register for data and status bits
- Interface often employs asynchronous handshaking protocol (mismatched speeds)
- Device driver required to ensure proper communication of data, control and status between CPU and device

I/O Interrupts

- Programmed I/O spend time in busy-wait loops
- Device requests service when ready with interrupt
- SRC interrupting device must return the vector address and interrupt information bits
  - Processor must tell device when this information
  - Accomplished with acknowledge signal
- Request and acknowledge form a communication handshake pair
- It should be possible to disable interrupts from individual devices

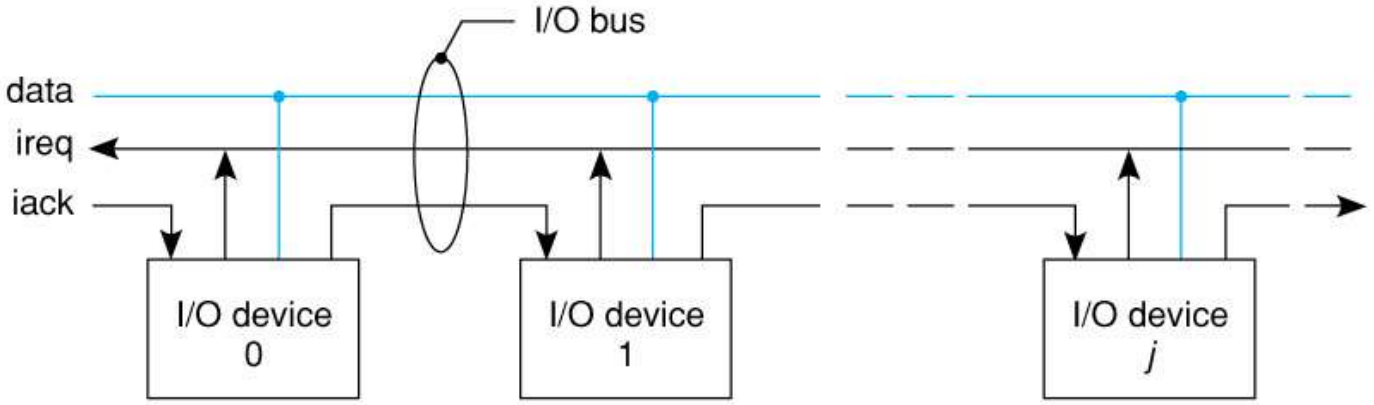
Simplified Interrupt Interface Logic



- Request and enable flags for each device
- Returns vector and interrupt information on bus when acknowledged
  - Vector – location of ISR
  - Info – device specific information
- Open collector NAND (wired-OR) so all devices can connect to the single `ireq` line

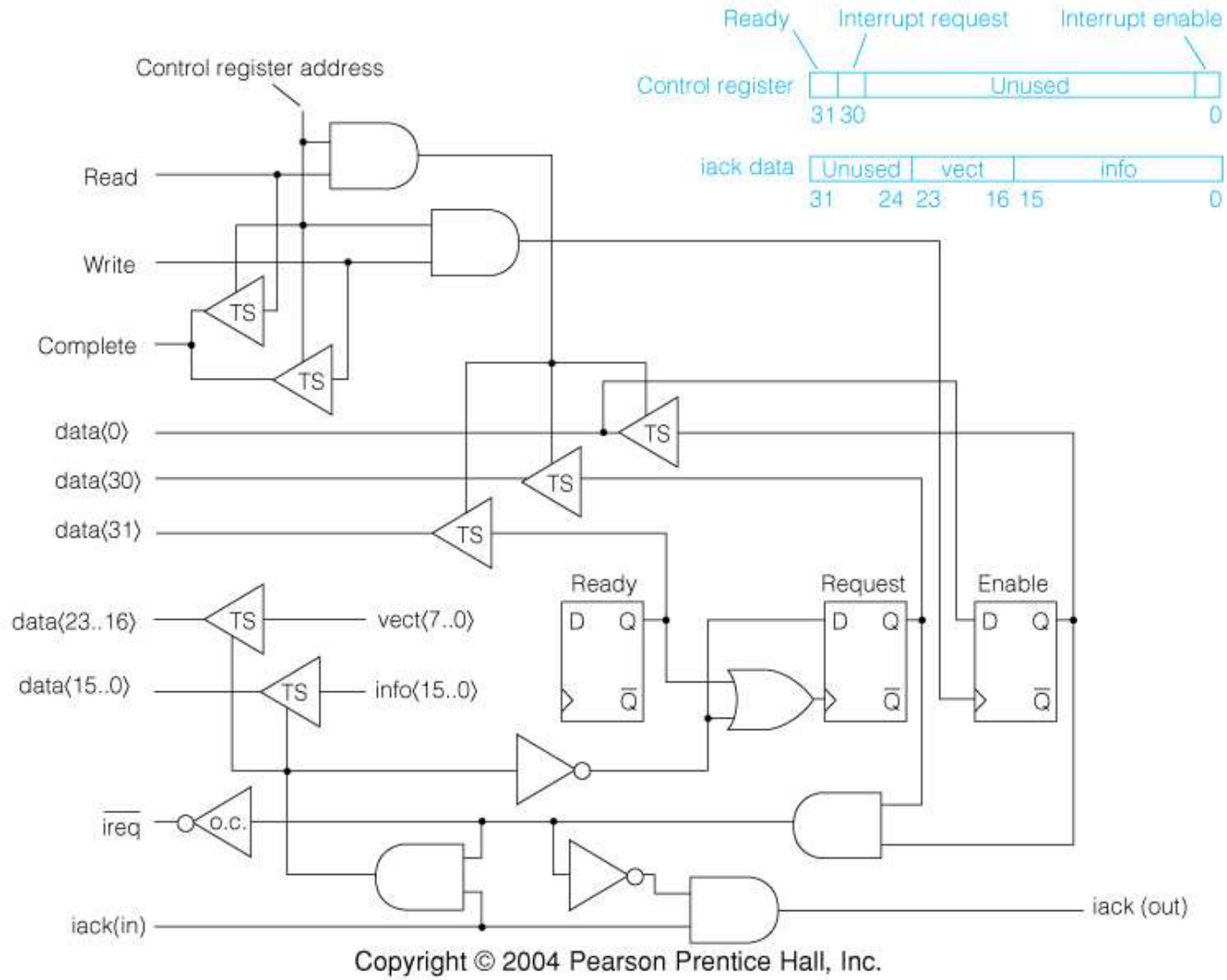
Interrupt Priority Chain

- Wired-OR allows multiple devices to request interrupts simultaneously
  - Must select appropriate device for acknowledgment
- Priority chain passes `iack` from device to device
  - $iack_j = iack_{j-1} \wedge \overline{req_{j-1} \wedge enb_{j-1}}$
  - Requires each enable for each device



Arrange devices with higher priority with lower j

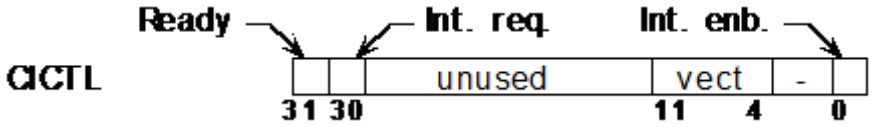
Interrupt Logic for SRC I/O Interface



- Request set by `Ready` and cleared by acknowledge
- `iack` only sent out if this device is not requesting

Subroutine for Interrupt Driven I/O

- Initialization routine




```
;Getline is called with return address in R31 and a pointer to a
;character buffer in R1. It will input characters up to a carriage
;return under interrupt control, setting Done to -1 when complete.
CR      .equ    13              ;ASCII code for carriage return.
CIvec   .equu   01F0H           ;Character input interrupt vector address.
Bufp:   .dw     1               ;Pointer to next character location.
Save:   .dw     2               ;Save area for registers on interrupt.
Done:   .dw     1               ;Flag location is -1 if input complete.
Getln:  st      r1, Bufp        ;Record pointer to next character.
        edi                     ;Disable interrupts while changing mask.
        la      r2, 1F1H        ;Get vector address and device enable bit
        st      r2, CICTL       ; and put into control register of device.
        la      r3, 0           ;Clear the
        st      r3, Done        ; line input done flag.
        een                     ;Enable Interrupts
        br      r31             ;   and return to caller.
```

Interrupt Handler for SRC Char Input

- Handler sit in the interrupt vector location and is initiated on request

```
        .org    CIvec           ;Start handler at vector address.
        str     r0, Save        ;Save the registers that
        str     r1, Save+4      ; will be used by the interrupt handler.
        ldr     r1, Bufp        ;Get pointer to next character position.
        ld      r0, CIN         ;Get the character and enable next input.
        st      r0, 0(r1)       ;Store character in line buffer.
        addi    r1, r1, 4       ;Advance pointer and
        str     r1, Bufp        ; store for next interrupt.
        lar     r1, Exit        ;Set branch target.
        addi    r0,r0, -CR      ;Carriage return? addi with minus CR.
        brnz    r1, r0          ;Exit if not CR, else complete line.
        la      r0, 0           ;Turn off input device by
        st      r0, CICTL       ; disabling its interrupts.
        la      r0, -1          ;Get a -1 indicator, and
        str     r0, Done        ; report line input complete.
Exit:   ldr     r0, Save        ;Restore registers
        ldr     r1, Save+4      ; of interrupted program.
        rfi                     ;Return to interrupted program.
```

General Functions of Interrupt Handler

1. Save the state of the interrupted program
2. Do programmed I/O operations to satisfy the interrupt request
3. Restart or turn off the interrupting device
4. Restore the state and return to the interrupted program

Interrupt Response Time

- Response to another interrupt is delayed until interrupts are re-enabled by rfi
- Character input handler disables interrupts for a maximum of 17 instructions
  - 20 MHz CPU, 10 cycles for acknowledge of interrupt, and average execution rate of 8 CPI
  - Second interrupt could be delayed by
    - $\frac{(10+17\times8)}{20} = 7.3\ \mu\text{sec}$

Nested Interrupts

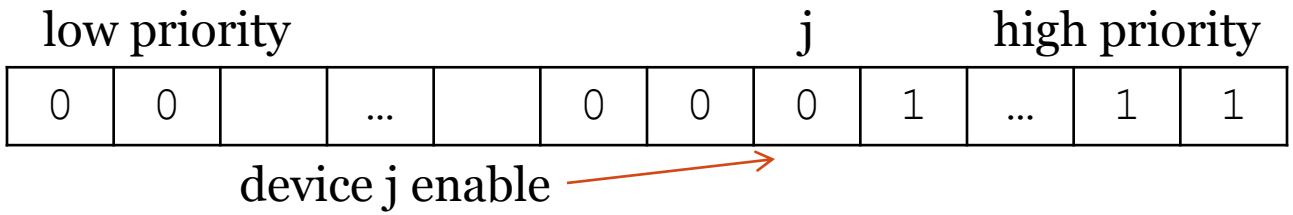
- Some high speed devices have a deadline for interrupt response
  - Possible when limited buffering
  - Longer response times may miss data
  - Real-time control system might fail to meet specs
- Require method to interrupt an interrupt handler
  - Higher priority (fast device) will be processed completely before returning to interrupt handler
- Interrupting devices are priority ordered by shortness of their deadlines

Steps in Response of Nested Interrupt Handler

1. Save the state changed by interrupt (`IPC` & `II`)
2. Disable lower priority interrupts
3. Re-enable exception processing
4. Service interrupting device
5. Disable exception processing
6. Re-enable lower priority interrupts
7. Restore saved interrupt state (`IPC` & `II`)
8. Return to interrupted program and re-enable exceptions

Interrupt Masks

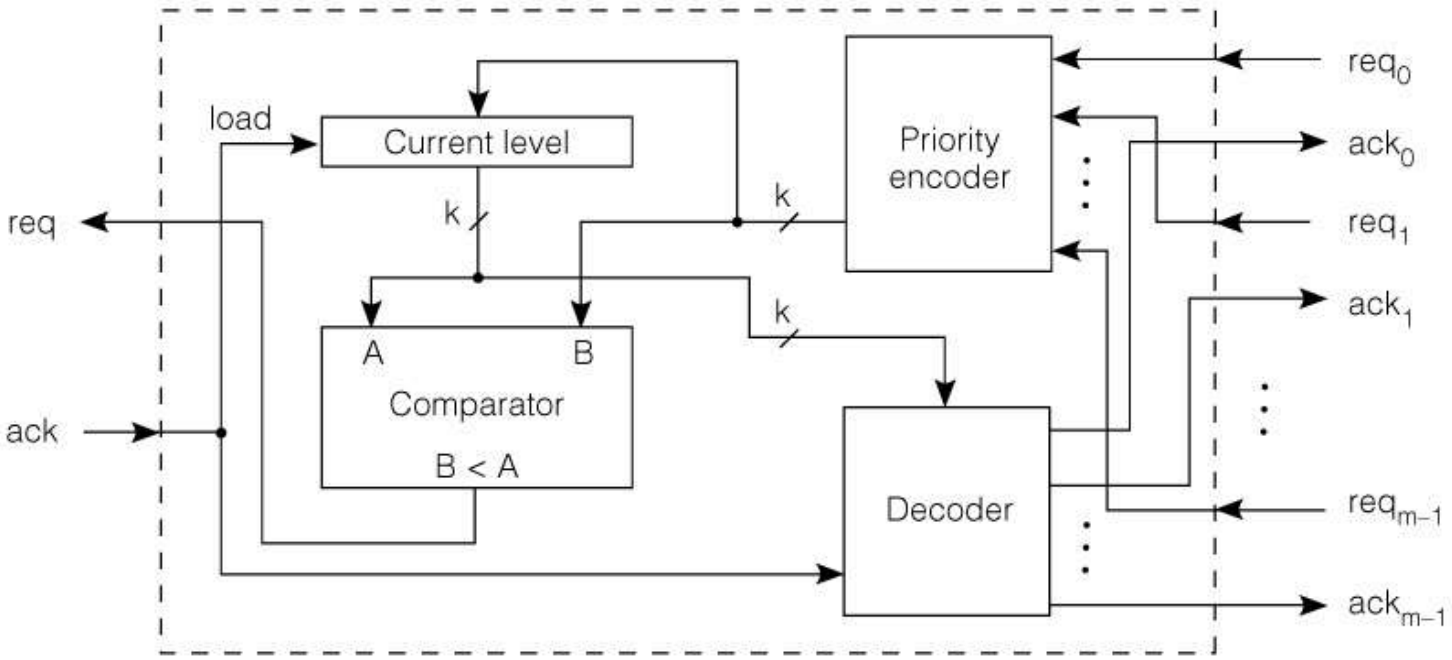
- Priority interrupt scheme could be managed using device enable bits
- Order bits from left to right in order of increasing priority to form interrupt mask
  - Mask value when executing device j interrupt handler



  - Notice only devices to left of j are enabled

Priority Interrupt System

- Priority groups – $m = 2^k$ groups



- Encoder gives binary index j of lowest numbered request (device with highest priority)
- Compare requesting priority with current - lower sends new request
- Acknowledge sets new priority level and sends `ack` to device

Key Concepts: Interrupt-Driven I/O

- CPU does not worry about device until I/O is ready for service
- Interrupt handler (ISR) is invoked by request for service from the device
- Individual device interrupts can be enabled/disabled by software
  - Provides mechanism for prioritizing interrupts and preventing interrupts in critical sections
- Interrupt nesting allows higher priority interrupts to interrupt lower priority ones