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/

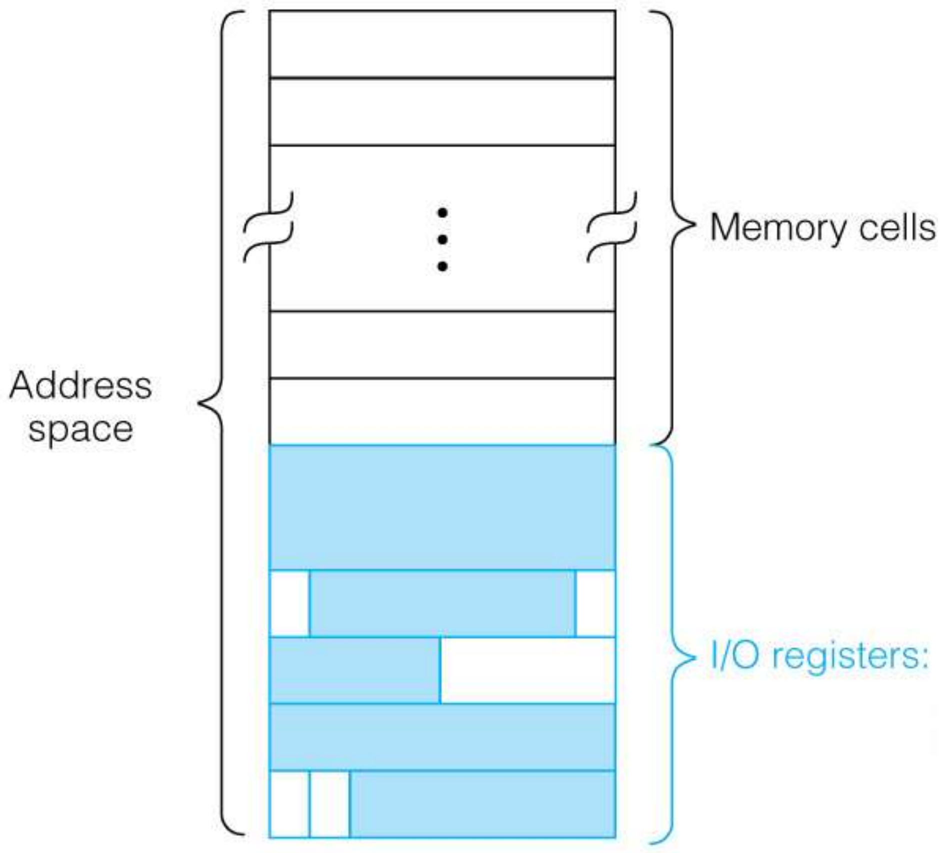
Outline

- Review Programmed I/O
- I/O Interrupts
- DMA
- I/O Format and Error Control

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

Memory Mapped I/O



- Memory divided to reserve space for I/O registers
  - I/O register appears as memory address to CPU
- 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

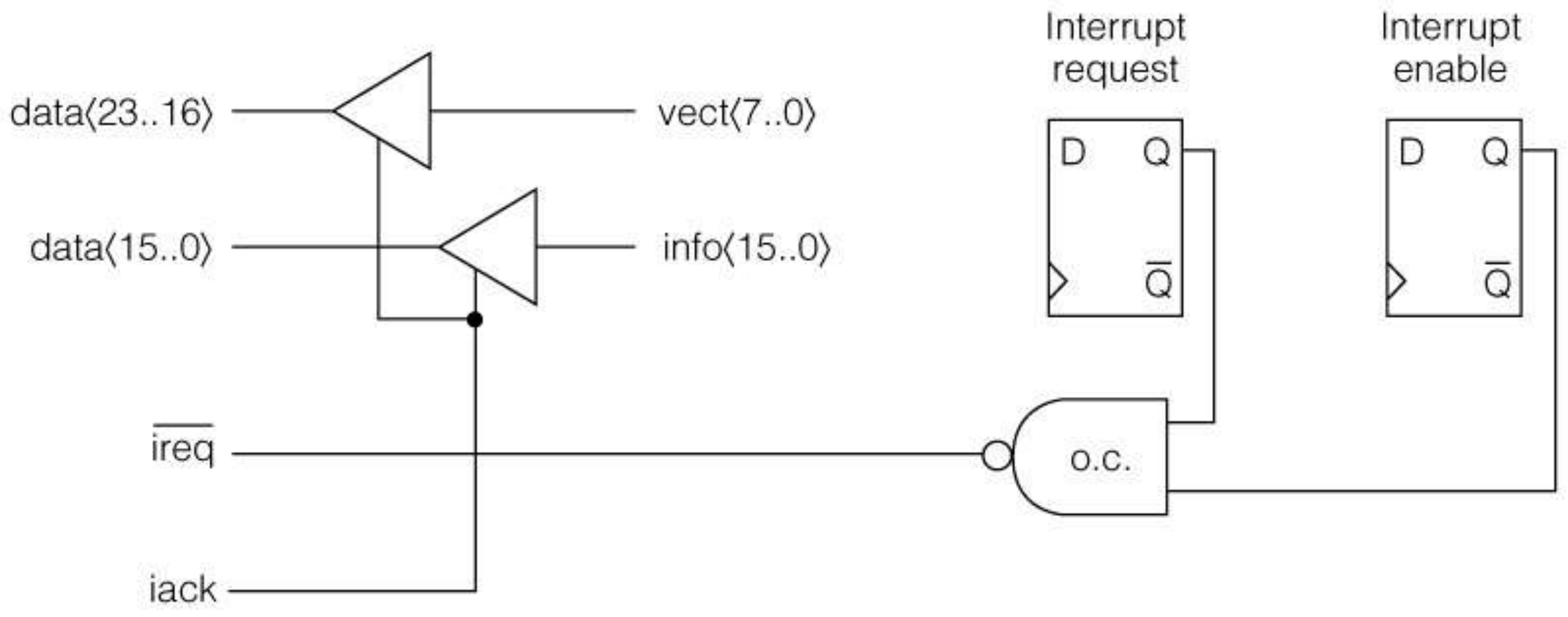
Key Concepts: Programmed I/O

- 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

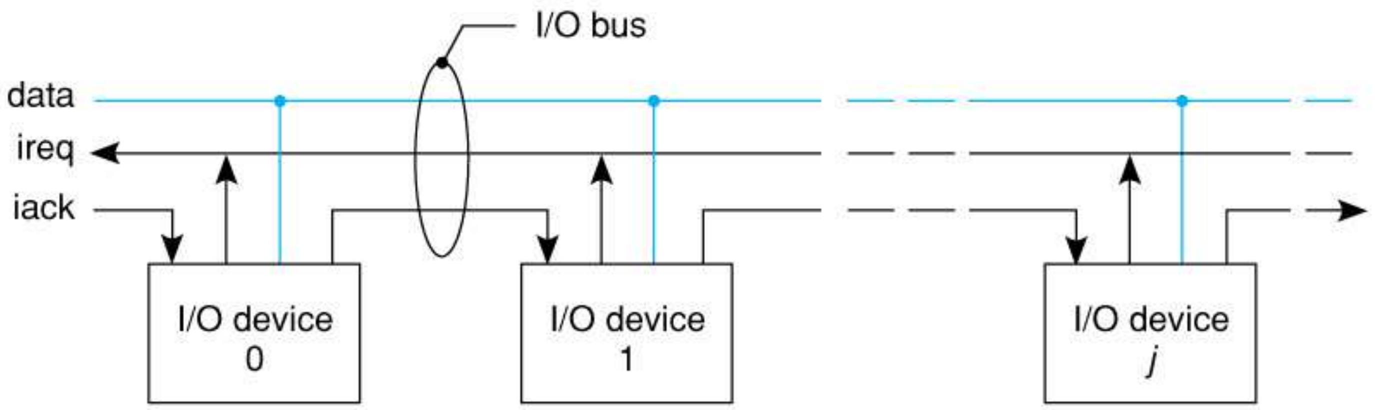
Simplified Interrupt Interface Logic



- 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

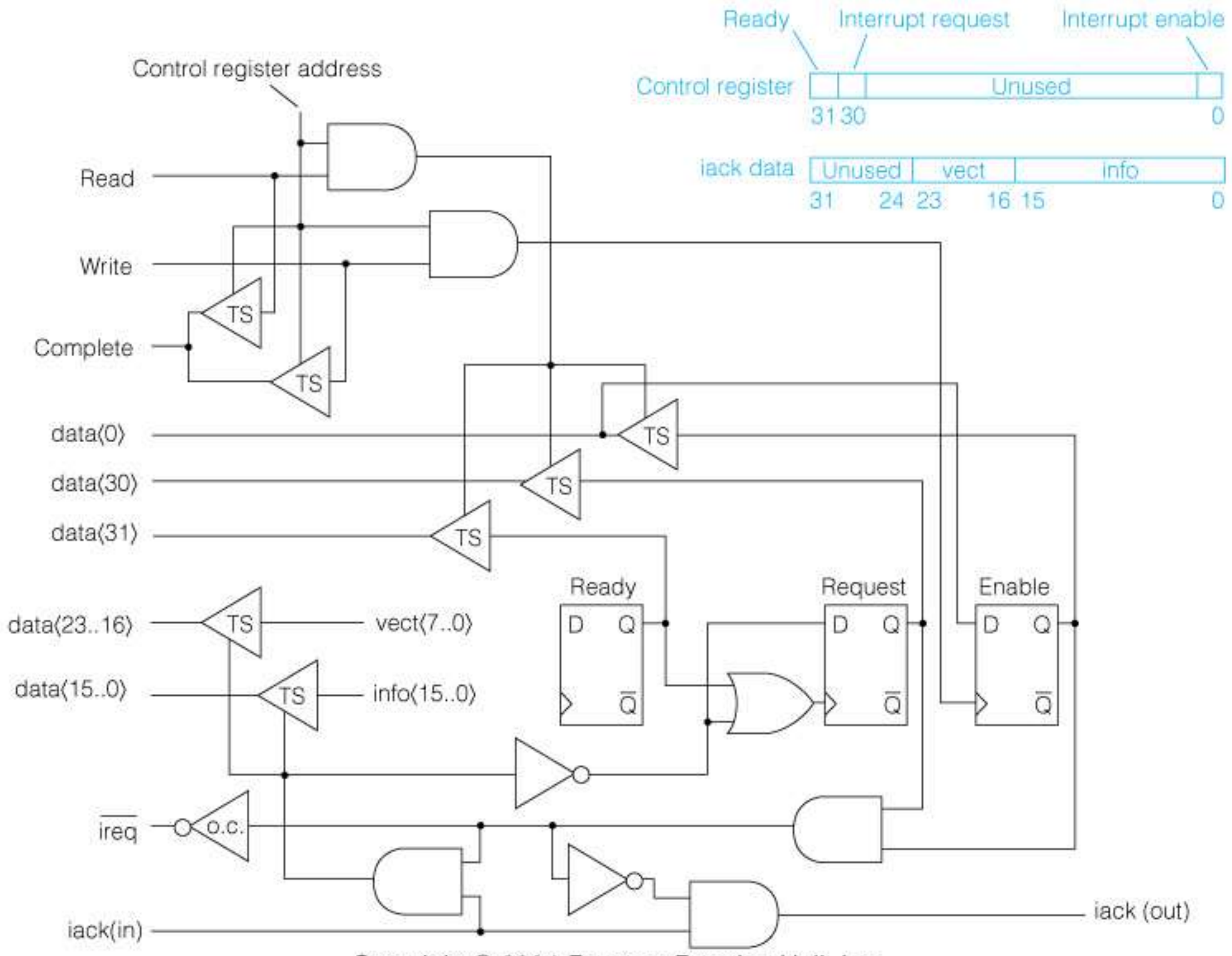
Interrupt Priority Chain

- 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

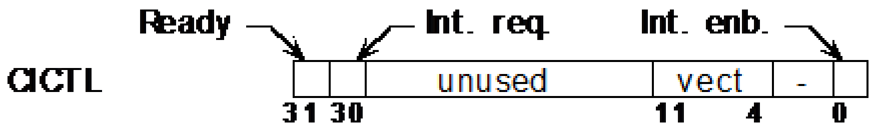
Interrupt Logic for SRC I/O Interface



- 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   .equ    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
    - $\frac{(10+17\times 8)}{20} = 7.3\ \mu\text{sec}$

Nested Interrupts

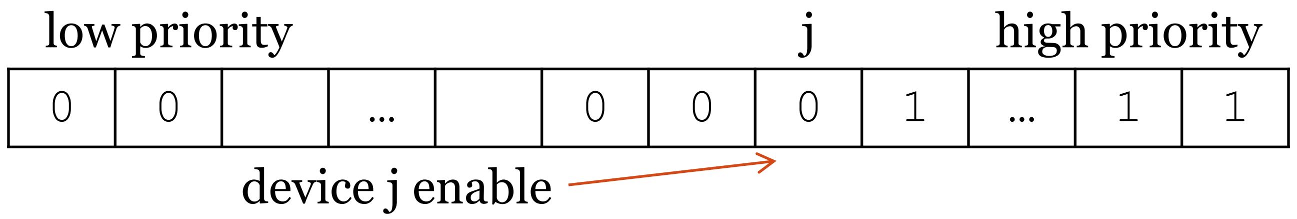
- Some high speed devices have a deadline for interrupt response
  - Possible when limited buffering
  - Longer response times may miss data
  - Real-time control system might fail to meet specs
- Require method to interrupt an interrupt handler
  - Higher priority (fast device) will be processed completely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

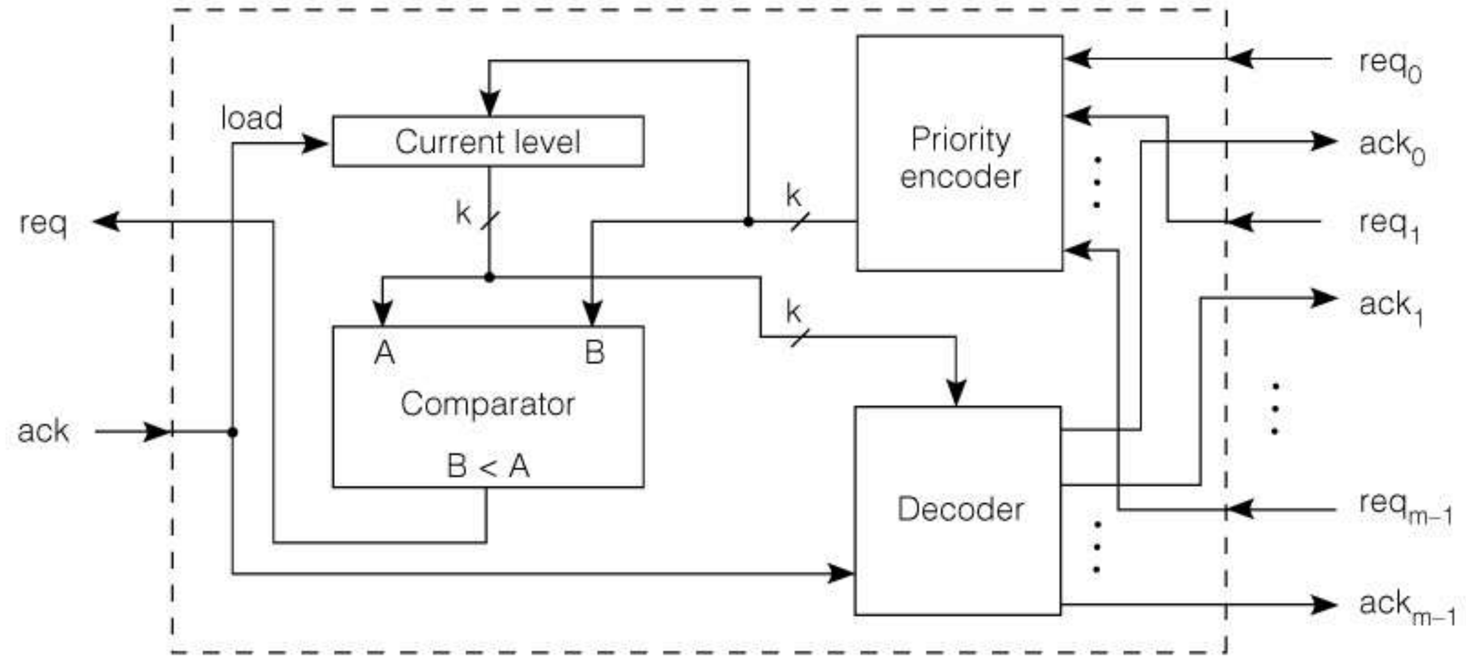


low priority							j			high priority	
0	0		...		0	0	0	1	...	1	1

  - 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

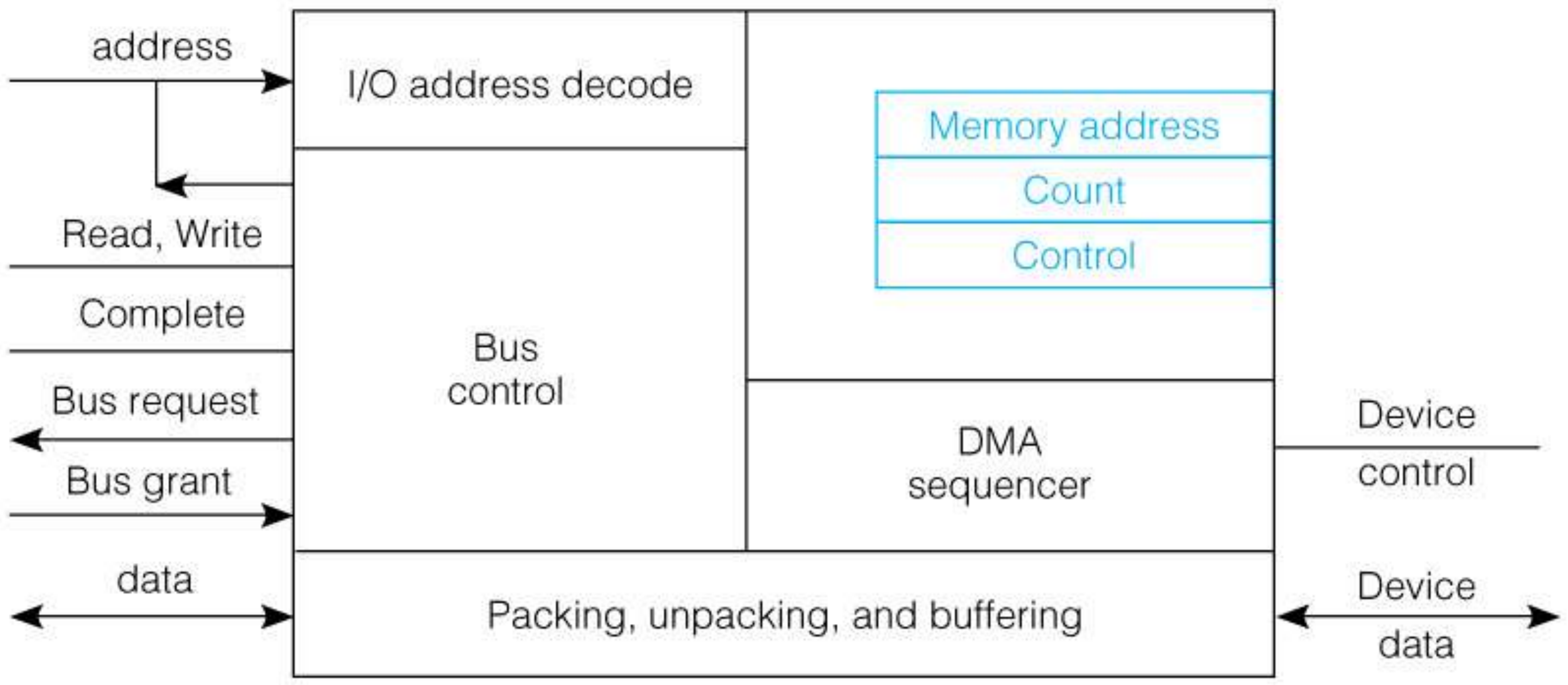
Direct Memory Access (DMA)

- Allows external devices to access memory without processor intervention
  - I/O device doesn't need handshake communication with processor
  - Useful for high speed devices
  - Prevents use of processor cycles for device-to-memory operations
- Requires a DMA interface device
  - Device must be programmed (set up) and have transfer initiated

DMA Device Interface Steps for Transfer

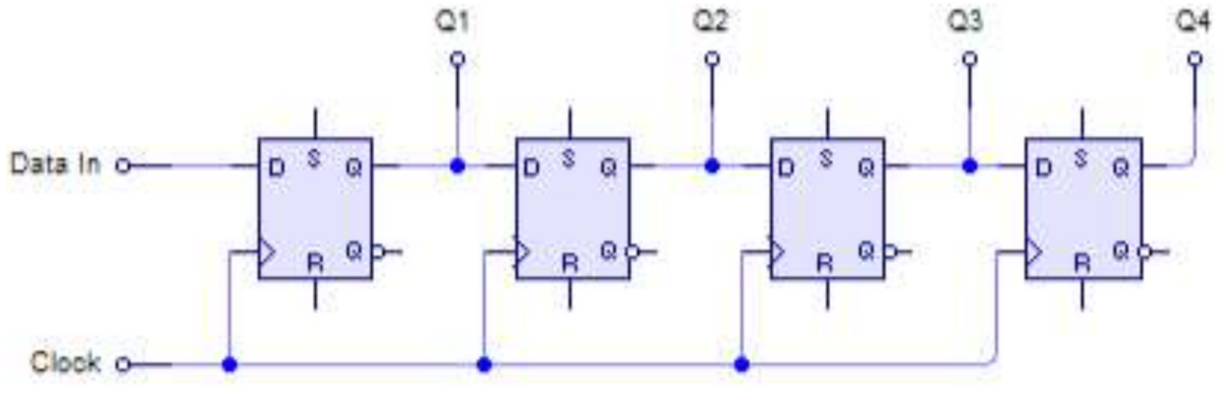
1. Become bus master
   - Only one master can control a bus
   - Handshake protocol
2. Send memory address and `R/W` signal
3. Synchronize sending and receiving of data using `Complete` signal
4. Release bus as needed (could be after each transfer)
   - Allow other DMA device or CPU access
5. Advance memory address to point to next data item
6. Count number of items transferred and check for end of data block
7. Repeat if more data needs to be transferred

I/O DMA Device Interface Architecture



Reformatting Data

- Data changes in format going between the processor/memory and an I/O device
- Parallel/serial conversion
  - Use shift registers



Serial to parallel conversion

Error Detection and Correction

- It is possible for data to be corrupted during transmission
- Bit error rate (BER) is the probability that a bit will be in error when read
  - Statistical property
  - Important in I/O where noise and signal integrity cannot be easily controlled (comes up all the time in communications)
  - 10^{-18} inside a processor
  - Between 10^{-8} – 10^{-12} or worse in outside world
- Many techniques to deal with errors
  - Parity check
  - SECDED encoding
  - CRC

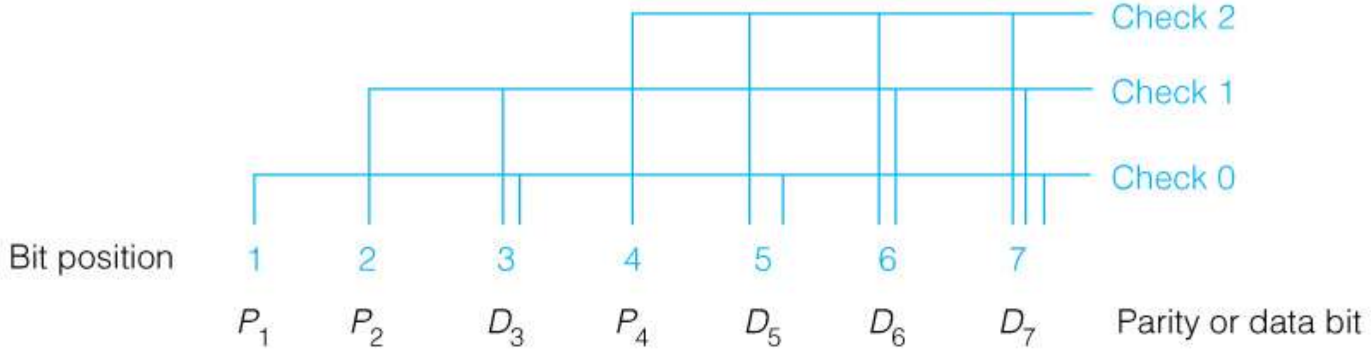
Parity Check

- An additional parity bit is added to transmitted data
- Even parity
  - Parity bit value selected to have even number of 1 bits in data
- Odd parity
  - Parity bit value selected to have odd number of 1 bits in data
- Example: `10011010`
  - 4 (even) number of 1 bits
  - Even parity: `10011010`**`0`**
  - Odd parity: `10011010`**`1`**

Hamming Codes

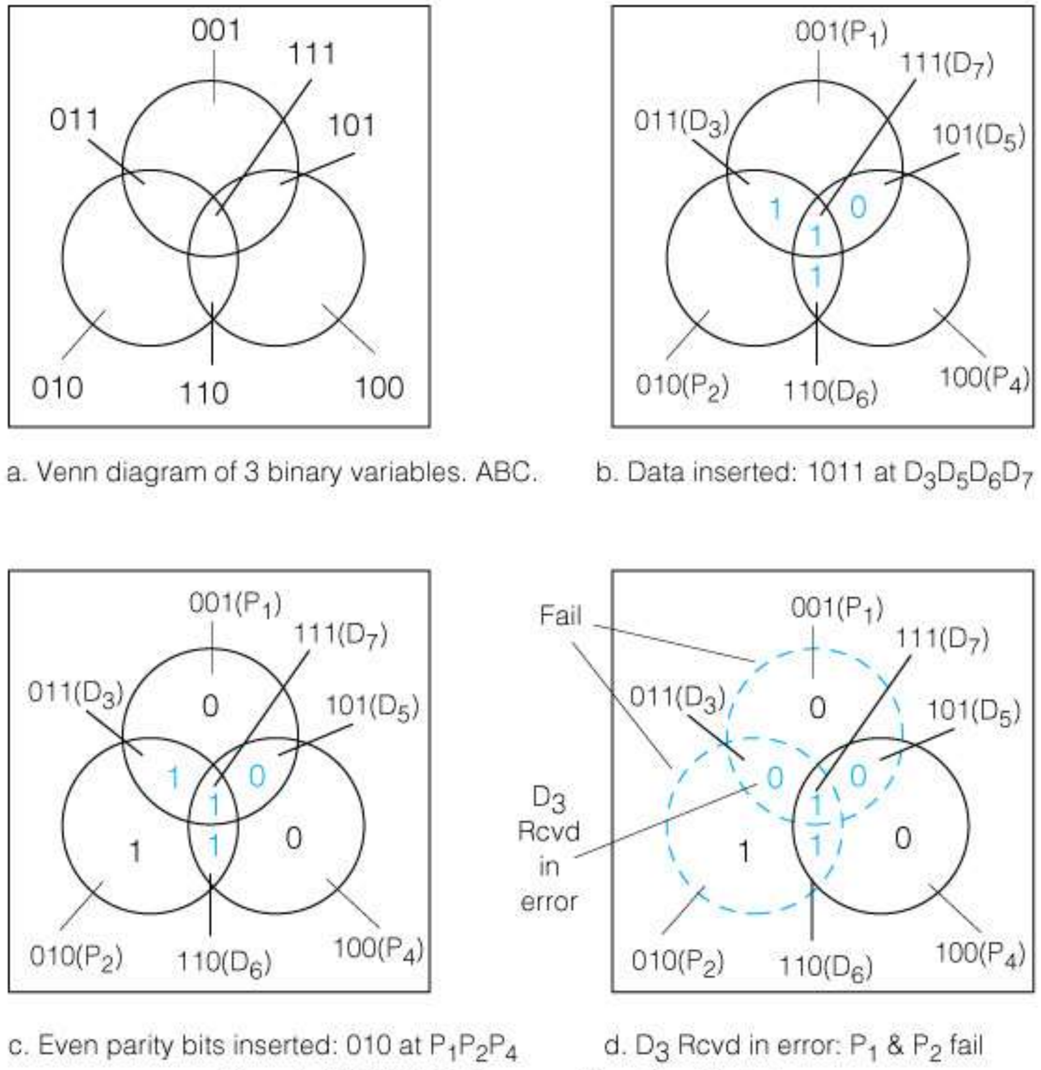
- Coding theory – how to encode a signal for transmission
  - Source coding – data compression
  - Channel coding – error correction
- Hamming codes are a class of codes that use combinations of parity checks to both detect and correct errors
  - Add group parity bits into the data bits
- Parity bits interspersed within data bits for ease of visualization
  - Parity bits placed in bit locations that are power of 2
  - Parity bit value computed with data bits in locations with a 1 in the binary representation of parity bit location

Multiple Parity Checks for Hamming Code



- Note: bit positions numbered beginning with 1 not 0
- Bit positions that are power of two reserved for parity bits
  - $P_1 = 001, P_2 = 010, P_4 = 100$
- Parity bit Pi computed over data bits with a 1 at parity bit location
  - $P_2 = 0\mathbf{1}0$ → uses $D_3 = 0\mathbf{1}1,\ D_6 = 1\mathbf{1}0,\ D_7 = 1\mathbf{1}1$
- Each data bit is involved in a number of parity checks
- For single bit error
  - All parity bits using that data bit will be in error
  - Binary encoding of checks indicates location of error

Venn Diagram for Hamming Codes



a. Venn diagram of 3 binary variables. ABC.

b. Data inserted: 1011 at $D_3D_5D_6D_7$

c. Even parity bits inserted: 010 at $P_1P_2P_4$

d. D_3 Rcvd in error: P_1 & P_2 fail

- Insert data into Venn diagram
- Sender computes and inserts parity bits (even)
- Receiver recomputes parity based on received signal
  - Detects errors based on parity bits
  - Corrects error given parity bits in error

Hamming Code Example

- Encode `1011` with odd Hamming code

- Insert the data bits: P_1 P_2 1 P_4 0 1 1
- P_1 is computed from $P_1 \oplus D_3 \oplus D_5 \oplus D_7 = 1$
  - $P_1 = 1$.
- P_2 is computed from $P_2 \oplus D_3 \oplus D_6 \oplus D_7 = 1$
  - $P_2 = 0$.
- P_4 is computed from $P_1 \oplus D_5 \oplus D_6 \oplus D_7 = 1$
  - $P_4 = 1$.

- The final encoded number is **`10`**`1`**`1`**`011`.

- Note: Hamming encoding scheme assumes that at most one bit is in error for error correction

SECDED (Single Error Correct, Double Error Detect)

- Hamming code with added parity bit
  - Parity bit at position 0 computed over entire Hamming code
- For single bit error
  - Unique set of Hamming checks will fail
  - Overall parity will also be wrong
  - Hamming check will indicate which bit position is in error
- For 2 bit error
  - One or more Hamming checks will fail
  - Overall parity will be correct.
- Assumes that the probability of 3 or more bits being in error is negligible.

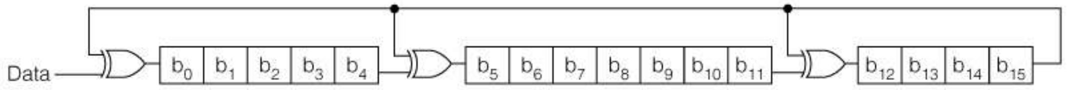
SECDED Example

- Odd parity SECDED of `1011`
- Hamming code = `1011011`
- Compute P_0 parity over entire code
  - 5 ones
  - $P_0 = 0$
- SECDED code = **`0`**`1011011`

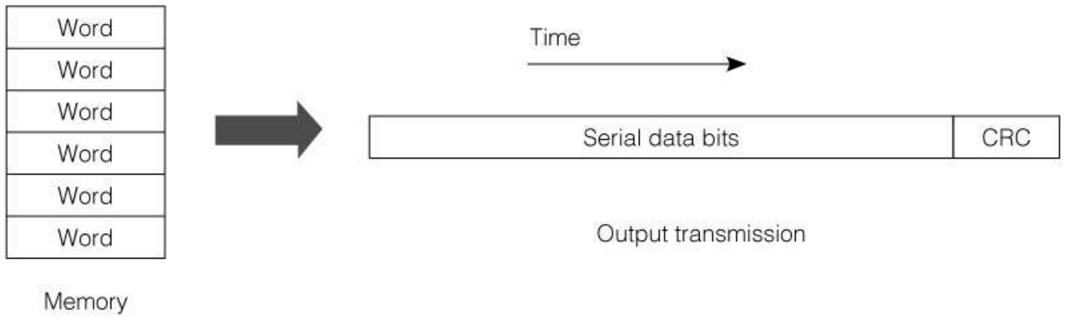
Cyclic Redundancy Check (CRC)

- Data transmitted serially (communication lines) has pattern of errors that results in several bits in error
  - Previous noise affected individual bits
- Parity checks are not as useful in these cases.
- Instead CRC checks are used.
- The CRC can be generated serially
- It usually consists of XOR gates.

Polynomial CRC Generator


Data
b_0 b_1 b_2 b_3 b_4
b_5 b_6 b_7 b_8 b_9 b_{10} b_{11}
b_{12} b_{13} b_{14} b_{15}

- The number and position of XOR gates is determined by the generating polynomial
- CRC does not support error correction but the CRC bits generated can be used to detect multi-bit errors.
- The CRC results in extra CRC bits, which are appended to the data word and sent along.
- The receiving entity can check for errors by recomputing the CRC and comparing it with the one that was transmitted.



Key Concepts: Data Formatting and Error Control

- Data representation is different internally and externally to CPU
  - I/O device may require different format conversion by interface
- Error detection and correction are often necessary for I/O subsystems because of higher error rates
- Simple parity checks are sufficient for low enough error rates (single bit)
- Hamming codes and SECDED allow for error detection and correction
- CRC checks are easy to implement and detect multiple bit errors
  - No correction – requires retransmission of data

Chapter 8 Summary

- I/O subsystems appear to programmer as a part of memory (memory-mapped I/O)
  - Special characteristics of data location, transfer, and synchronization make it different
- Combination of hardware and software protocols guarantee correct data transfer
  - Programmed I/O – uses an instruction to begin transfer and polls
  - Interrupt-driven I/O – uses exception handling to service I/O
  - DMA – interface allows device to control memory bus like the processor
- Data within processor and outside in devices have different characteristics that may require data format change (serial/parallel)
- I/O is more error prone than CPU hence requires error detection and/or correction