# Chapter 8: Input and Output

**Topics**

**8.1 The I/O Subsystem**
- I/O buses and addresses

**8.2 Programmed I/O**
- I/O operations initiated by program instructions

**8.3 I/O Interrupts**
- Requests to processor for service from an I/O device

**8.4 Direct Memory Access (DMA)**
- Moving data in and out without processor intervention

**8.5 I/O Data Format Change and Error Control**
- Error detection and correction coding of I/O data

1

# Three Requirements of I/O Data Transmission

**(1) Data location**
- Correct device must be selected
- Data must be addressed within that device

**(2) Data transfer**
- Amount of data varies with device and may need be specified
- Transmission rate varies greatly with device
- Data may be output, input, or either with a given device

**(3) Synchronization**
- For an output device, data must be sent only when the device is ready to receive it
- For an input device, the processor can read data only when it is available from the device

2

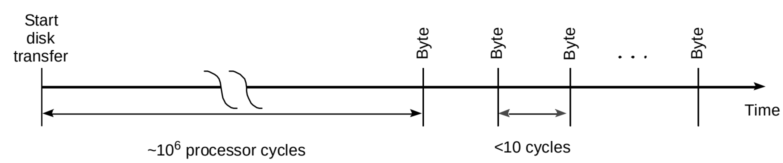
# Location of I/O Data

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

3

# Fig 8.1 Disk Data Transfer Timing

- Keyboard delivers one character about every 1/10 second at the fastest
- Rate may also vary, as in disk rotation delay followed by block transfer




4

## Synchronization—I/O Devices Are Not Timed by Master Clock

- **Not only can I/O rates differ greatly from processor speed, but I/O is asynchronous**
- **Processor will interrogate state of device and transfer information at clock ticks**
- **I/O status and information must be stable at the clock tick when it is accessed**
- **Processor must know when output device can accept new data**
- **Processor must know when input device is ready to supply new data**

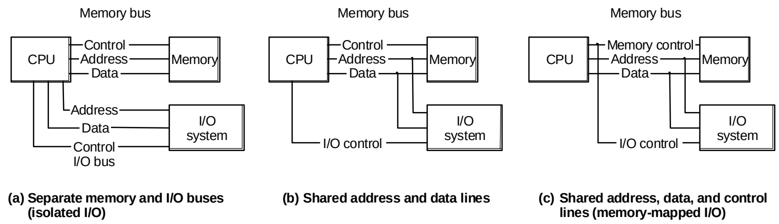
5

## Reducing Location and Synchronization to Data Transfer

- **Since the structure of device data location is device dependent, device should interpret it**
  - **The device must be selected by the processor, but**
  - **Location within the device is just information passed to the device**
- **Synchronization can be done by the processor reading device status bits**
  - **Data available signal from input device**
  - **Ready to accept output data from output device**
- **Speed requirements will require us to use other forms of synchronization: discussed later**
  - **Interrupts and DMA are examples**

6

# Fig 8.2 Independent and Shared Memory and I/O Buses




- **Allows tailoring bus to its purpose, but**
- **Requires many connections to CPU (pins)**

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

- **Least expensive option**
- **Speed penalty**

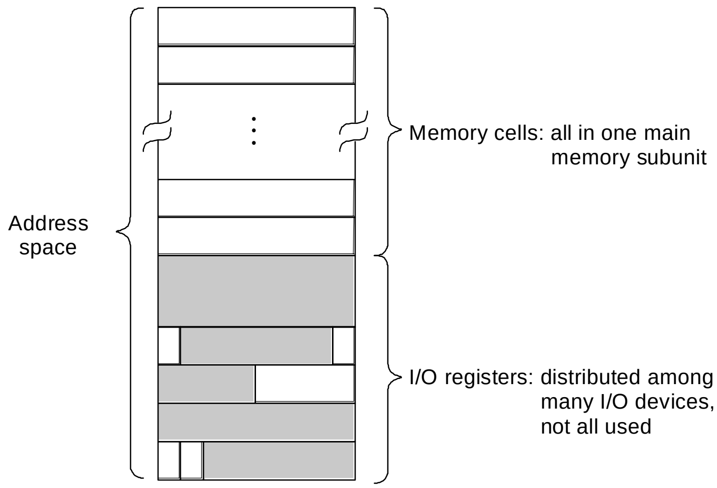
7

# Memory-Mapped I/O

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

8

## Fig 8.3 Address Space of a Computer Using Memory-Mapped I/O



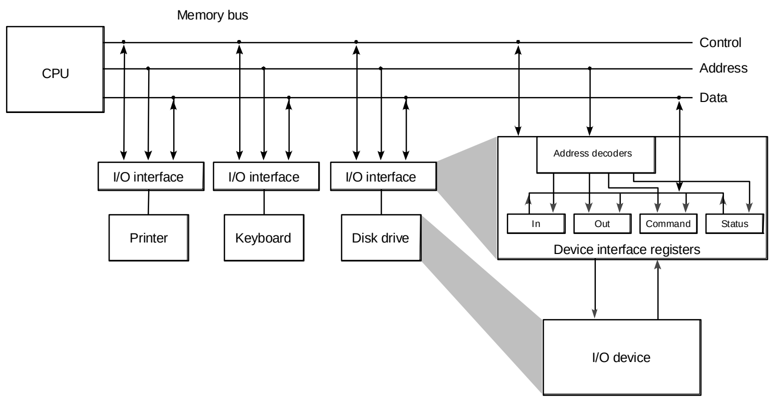

9

## Programmed I/O

- **Requirements for a device using programmed I/O**
  - **Device operations take many instruction times**
  - **One word data transfers—no burst data transmission**
- **Program instructions have time to test device status bits, write control bits, and read or write data at the required device speed**
- **Example status bits:**
  - **Input data ready**
  - **Output device busy or off-line**
- **Example control bits:**
  - **Reset device**
  - **Start read or start write**

10

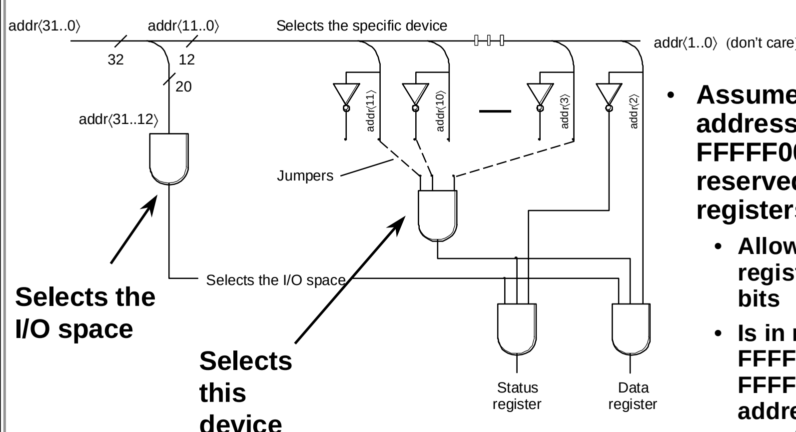
# Fig 8.4 Programmed I/O Device Interface Structure




- **Focus on the interface between the unified I/O and memory bus and an arbitrary device. Several device registers (memory addresses) share address decode and control logic.**

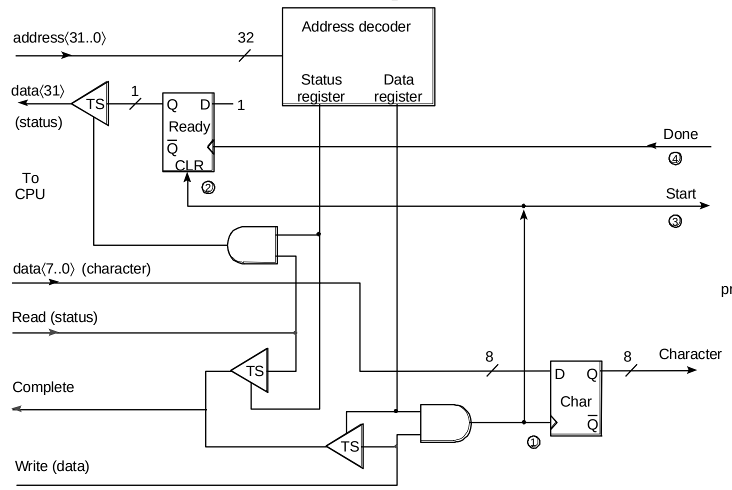


# Fig 8.5 SRC I/O Register Address Decoder

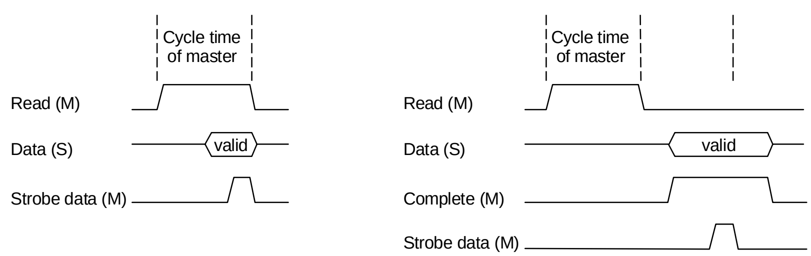



- **Assumes SRC addresses above $FFFFF000_{16}$ are reserved for I/O registers**
  - **Allows for 1024 registers of 32 bits**
  - **Is in range $FFFF0000_{16}$ to $FFFFFFFF_{16}$ addressable by negative displacement**

# Fig 8.6 Interface Design for SRC Character Output




13

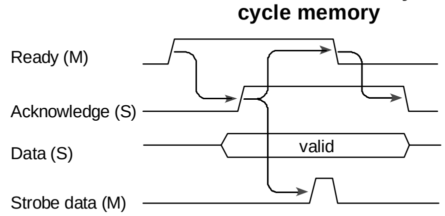



**(a) Synchronous input**

**(b) Semisynchronous input**

- **Used for register to register inside CPU**
- **Used for memory to CPU read with few cycle memory**

# Fig 8.7 Synchronous, Semi-synchronous, and Asynchronous Data Input

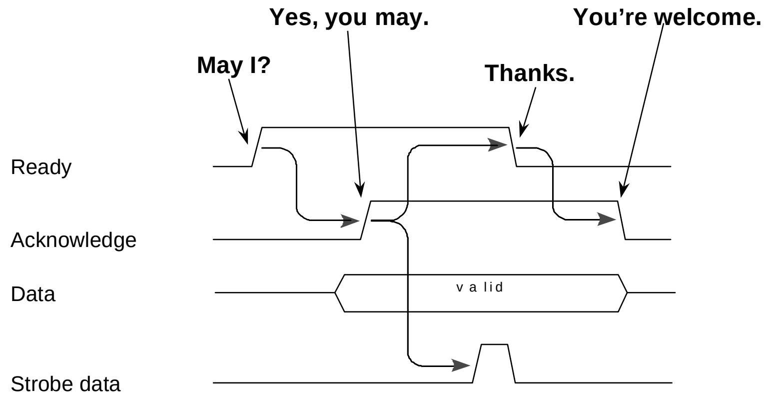



**(c) Asynchronous input**

- **Used for I/O over longer distances (feet)**

14

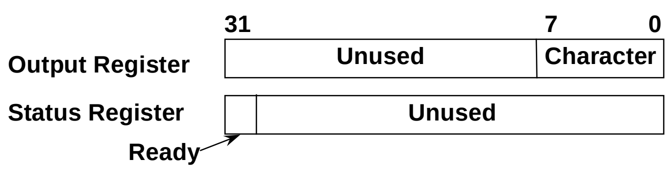
# Fig 8.7c  Asynchronous Data Input




15

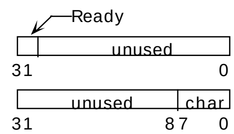
# Example:  Programmed I/O Device Driver for Character Output

- **Device requirements:**
  - **8 data lines set to bits of an ASCII character**
  - **Start signal to begin operation**
  - **Data bits held until device returns Done  signal**
- **Design decisions 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**




16

# Fig 8.8 Program Fragment for Character Output




```
       Status register  COSTAT = FFFFF110H

       Output register    COUT = FFFFF114H

       lar    r3, Wait      ;Set branch target for wait.
       ldr    r2, Char      ;Get character for output.
Wait:  ld     r1, COSTAT    ;Read device status register,
       brpl   r3, r1        ;Branch to Wait if not ready.
       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**
- **A 10 MIPS SRC would execute 10,000 instructions waiting for a 1,000 character/sec printer**

17

# Fig 8.9 Program Fragment to Print 80-Character Line


```
   Status Register  LSTAT = FFFFF130H
   Output Register   LOUT = FFFFF134H
   Command Register  LCMD = FFFFF138H
       lar    r1, Buff      ;Set pointer to character buffer.
       la     r2, 80        ;Initialize character counter and
       lar    r3, Wait      ; branch target.
Wait:  ld     r0, LSTAT     ;Read Ready bit,
       brpl   r3, r0        ; test, and repeat if not ready.
       ld     r0, 0(r1)     ;Get next character from buffer,
       st     r0, LOUT      ; and send to printer.
       addi   r1, r1, 4     ;Advance character pointer, and
       addi   r2, r2, -1    ; count character.
       brnz   r3, r2        ;If not last, go wait on ready.
       la     r0, 1         ;Get a print line command,
       st     r0, LCMD      ; and send it to the printer.
```

18

# Multiple Input Device Driver Software

- 32 low-speed input devices
  - Say, keyboards at -10 characters/sec
  - Max rate of one every 3 ms
- Each device has a control/status register
  - Only Ready status bit, bit 31, is used
  - Driver works by <u>polling</u> (repeatedly testing) Ready bits
- Each device has an 8 bit input data register
  - Bits 7..0 of 32-bit input word hold the character
- Software controlled by pointer and Done flag
  - Pointer to next available location in input buffer
  - Device's done is set when CR received from device
  - Device is idle until other program (not shown) clears Done

19

# Driver Program Using Polling for 32-Character Input Devices

| | |
|---|---|
| FFFFF300 | Dev 0 CTL |
| FFFFF304 | Dev 0 IN |
| FFFFF308 | Dev 1 CTL |
| FFFFF30C | Dev 1 IN |
| FFFFF310 | Dev 2 CTL |
| | ⋮ |

- 32 pairs of control/status and input data registers

r0 - working reg        r1 - input char.
r2 - device index        r3 - none active

```
CICTL   .equ    FFFFF300H      ;First input control register.
CIN     .equ    FFFFF304H      ;First input data register.
CR      .equ    13             ;ASCII carriage return.
Bufp:   .dcw    1              ;Loc. for first buffer pointer.
Done:   .dcw    63             ;Done flags and rest of pointers.
Driver: lar     r4, Next       ;Branch targets to advance to next
        lar     r5, Check      ; character, check device active,
        lar     r6, Start      ; and start a new polling pass.
```

20

## Driver Program Using Polling for 32-Character Input Devices (cont'd)

```
Start:  la      r2, 0           ;Point to first device, and
        la      r3, 1           ; set all inactive flag.
Check:  ld      r0,Done(r2)     ;See if device still active, and
        brmi    r4, r0          ; if not, go advance to next device.
        ld      r3, 0           ;Clear the all inactive flag.
        ld      r0,CICTL(r2)    ;Get device ready flag, and
        brpl    r4, r0          ; go advance to next if not ready.
        ld      r0,CIN(r2)      ;Get character and
        ld      r1,Bufp(r2)     ; correct buffer pointer, and
        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       ;Check for carriage return, and
        brnz    r4, r0          ; if not, 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 a device is active, make a new pass.
```

21

## Characteristics of the Polling Device Driver

- If all devices active and always have character ready,
- Then 32 bytes input in 547 instructions
- This is data rate of 585 KB/s in a 10 MIPS CPU
- But, if CPU just misses setting of Ready, 538 instructions are executed before testing it again
- This 53.8 μsec delay means that a single device must run at less than 18.6 Kchars/s to avoid risk of losing data
- Keyboards are thus slow enough

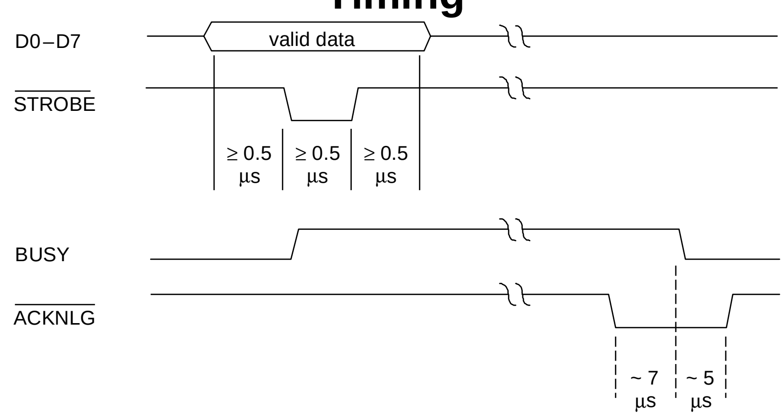
22

# Tbl 8.1 Signal Names and Functions for the Centronics Printer Interface

| Interface Signal | Direction | Description |
|---|---|---|
| $\overline{\text{STROBE}}$ | Out | Data out strobe |
| D0 | Out | Least significant data bit |
| D1 | Out | Data bit |
| ... | ... | ... |
| D7 | Out | Most significant data bit |
| $\overline{\text{ACKNLG}}$ | In | Pulse on done with last character |
| BUSY | In | Not ready |
| PE | In | No paper when high |
| SLCT | In | Pulled high |
| $\overline{\text{AUTOFEEDXT}}$ | Out | Auto line feed |
| $\overline{\text{INIT}}$ | Out | Initialize printer |
| $\overline{\text{ERROR}}$ | In | Can’t print when low |
| $\overline{\text{SLCTIN}}$ | Out | Deselect protocol |

23

# Fig 8.11 Centronics Printer Data Transfer Timing




- **Minimum times specified for output signals**
- **Nominal times specified for input signals**

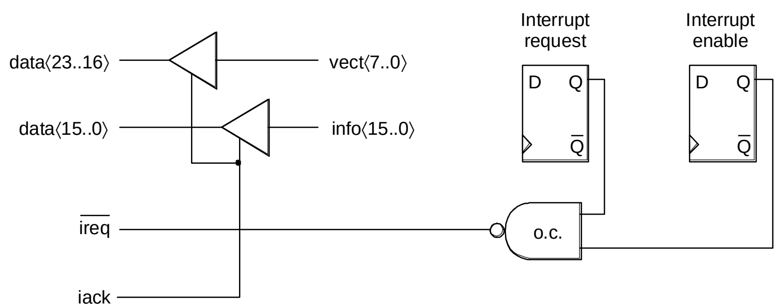
24

## I/O Interrupts

- Key idea: instead of processor executing wait loop, device requests interrupt when ready
- In SRC the interrupting device must return the vector address and interrupt information bits
- Processor must tell device when to send this information—done by acknowledge signal
- Request and acknowledge form a communication handshake pair
- It should be possible to disable interrupts from individual devices

25

## Fig 8.12 Simplified Interrupt Circuit for an I/O Interface




- Request and enable flags per device
- Returns vector and interrupt information on bus when acknowledged

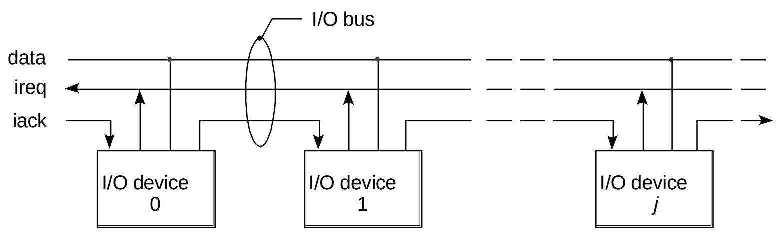
26



1.

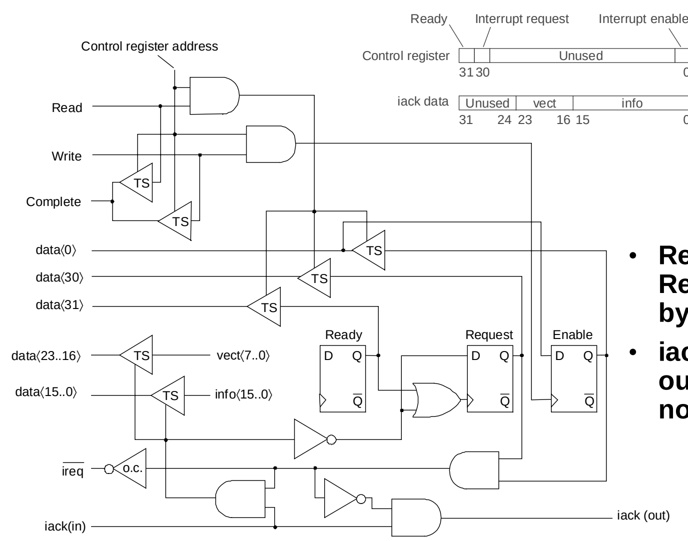
# Fig 8.13 Daisy-Chained Interrupt Acknowledge Signal

- **How does acknowledge signal select one and only one device to return interrupt information?**
- **One way is to use a priority chain with acknowledge passed from device to device**




27

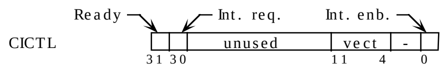
# Fig 8.14 Interrupt Logic in an I/O Interface




- **Request set by Ready, cleared by acknowledge**
- **iack only sent out if this device not requesting**

28

# Getline Subroutine for Interrupt-Driven Character I/O




```
;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.
```

29

# Interrupt Handler for SRC Character Input

```
        .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.
```

30

# General Functions of an 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

31

# Interrupt Response Time

- Response to another interrupt is delayed until interrupts reenabled by rfi
- Character input handler disables interrupts for a maximum of 17 instructions
- If the CPU clock is 20 MHz, it takes 10 cycles to acknowledge an interrupt, and average execution rate is 8 CPI

Then 2nd interrupt could be delayed by

$$(10 + 17 \times 8) / 20 = 7.3\ \mu\text{sec}$$

32

## Nested Interrupts—Interrupting an Interrupt Handler

- **Some high-speed devices have a deadline for interrupt response**
  - **Longer response times may miss data on a moving medium**
  - **A real-time control system might fail to meet specifications**
- **To meet a short deadline, it may be necessary to interrupt the handler for a slow device**
- **The higher priority interrupt will be completely processed before returning to the interrupted handler**
- **Hence the designation nested interrupts**
- **Interrupting devices are priority ordered by shortness of their deadlines**

33

## Steps in the Response of a Nested Interrupt Handler

**(1) Save the state changed by interrupt (IPC and II);**
**(2) Disable lower priority interrupts;**
**(3) Reenable exception processing;**
**(4) Service interrupting device;**
**(5) Disable exception processing;**
**(6) Reenable lower priority interrupts;**
**(7) Restore saved interrupt state (IPC and II);**
**(8) Return to interrupted program and reenable exceptions.**

34

# Fig 8.16 Interrupt Masks for Executing Device *j* Handler

- **Conceptually, a priority interrupt scheme could be managed using device-enable bits**
- **Order the bits from left to right in order of increasing priority to form an <u>interrupt mask</u>**
- **Value of the mask when executing device j interrupt handler is**

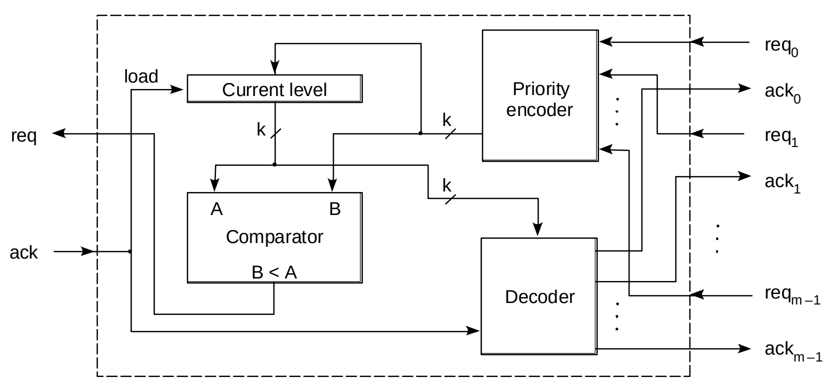
low priority high priority

| 0 0 • • • 0 0 0 1 1 • • • 1 1 1 |

device *j* enable



# Fig 8.17 Priority Interrupt System with $m = 2^k$ Levels



## Direct Memory Access (DMA)

- **Allows external devices to access memory without processor intervention**
- **Requires a DMA interface device**
- **Must be "set up" or programmed and transfer initiated**

37

## Steps a DMA Device Interface Must Take to Transfer a Block of Data

**1. Become bus master**

**2. Send memory address and R/W signal**

**3. Synchronized sending and receiving of data using Complete signal**

**4. Release bus as needed (perhaps after each transfer)**

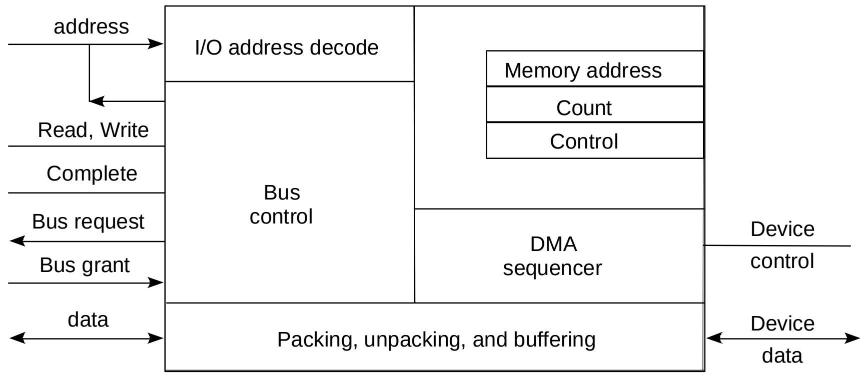
**5. Advance memory address to point to next data item**

**6. Count number of items transferred and check for end of data block**

**7. Repeat if more data to be transferred**

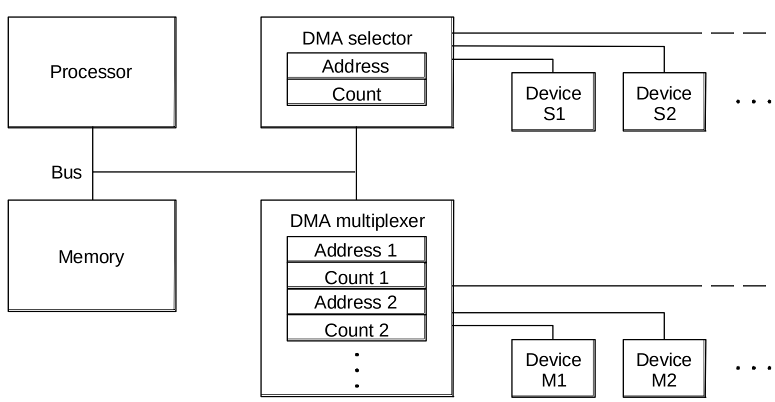
38

# Fig 8.18 I/O Interface Architecture for a DMA Device




39

# Fig 8.19 Multiplexer and Selector DMA Channels




40

# Error Detection and Correction

- Bit-error rate, BER, is the probability that, when read, a given bit will be in error
- BER is a statistical property
- Especially important in I/O, where noise and signal integrity cannot be so easily controlled
- $10^{-18}$ inside processor
- $10^{-8}$ - $10^{-12}$ or worse in outside world
- Many techniques
  - Parity check
  - SECDED encoding
  - CRC

41

# Parity Checking

- Add a parity bit to the word
- Even parity: add a bit if needed to make number of bits even
- Odd parity: add a bit if needed to make number of bits odd
- Example: for word 10011010, to add odd parity bit: 100110101

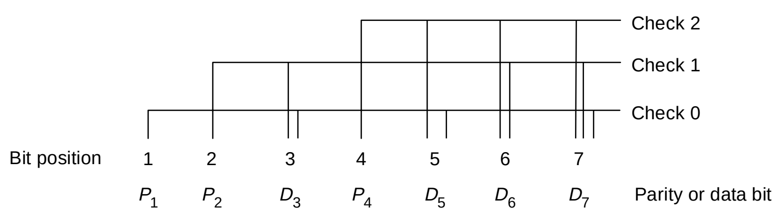
42

# Hamming Codes

- Hamming codes are a class of codes that use combinations of parity checks to both detect and correct errors.
- They add a group of parity check bits to the data bits.
- For ease of visualization, intersperse the parity bits within the data bits; reserve bit locations whose bit numbers are powers of 2 for the parity bits. Number the bits from l to r, starting at 1.
- A given parity bit is computed from data bits whose bit numbers contain a 1 at the parity bit number.



**Fig 8.20 Multiple Parity Checks Making Up a Hamming Code**




- **Add parity bits, $P_i$, to data bits, $D_i$.**
- **Reserve bit numbers that are a power of 2 for parity bits.**
- **Example: $P_1$ = 001, $P_2$ = 010, $P_4$ = 100, etc.**
- **Each parity bit, $P_i$, is computed over those data bits that have a "1" at the bit number of the parity bit.**
- **Example: $P_2$ (0*1*0) is computed from $D_3$ (0*1*1), $D_6$ (1*1*0), $D_7$ (1*1*1), ...**
- **Thus each bit takes part in a different combination of parity checks.**
- **When the word is checked, if only one bit is in error, all the parity bits that use it in their computation will be incorrect.**

## Example 8.1 Encode 1011 Using the Hamming Code and Odd Parity

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

- The final encoded number is 1 0 1 1 0 1 1.

- Note that the Hamming encoding scheme assumes that at most one bit is in error.

45

## SECDED (Single Error Correct, Double Error Detect)

- Add another parity bit, at position 0, which is computed to make the parity over all bits, data and parity, even or odd.
- If one bit is in error, a unique set of Hamming checks will fail, and the overall parity will also be wrong.
- Let $c_i$ be true if check i fails, otherwise true.
- In the case of a 1-bit error, the string $c_{k-1}, \ldots, c_1, c_0$ will be the binary index of the erroneous bit.
- For example, if the $c_i$ string is 0110, then bit at position 6 is in error.
- If two bits are in error, one or more Hamming checks will fail, but the overall parity will be correct.
- Thus the failure of one or more Hamming checks, coupled with correct overall parity, means that 2 bits are in error.
- This assumes that the probability of 3 or more bits being in error is negligible.

46

## Example 8.2 Compute the Odd Parity SECDED Encoding of the 8-bit value 01101011

The 8 data bits 01101011 would have 5 parity bits added to them to make the 13-bit value

$P_0$ $P_1$ $P_2$ 0 $P_4$ 1 1 0 $P_8$ 1 0 1 1.

Now $P_1 = 0$, $P_2 = 1$, $P_4 = 0$, and $P_8 = 0$, and we can compute that $P_0$, overall parity, = 1, giving the encoded value:

1 0 1 0 0 1 1 0 0 1 0 1 1

47

## Example 8.3 Extract the Correct Data Value from the String 0110101101101, Assuming Odd Parity

- The string shows even parity, so there must be a single bit in error.
- Checks $c_2$ and $c_4$ fail, giving the binary index of the erroneous bits as 0110 = 6, so $D_6$ is in error.
- It should be 0 instead of 1.

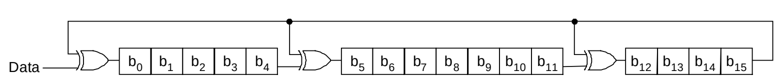
48

# Cyclic Redundancy Check, CRC

- When data is transmitted serially over communications lines, the pattern of errors usually results in several or many bits in error, due to the nature of line noise.
- The "crackling" of telephone lines is this kind of noise.
- 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.

49

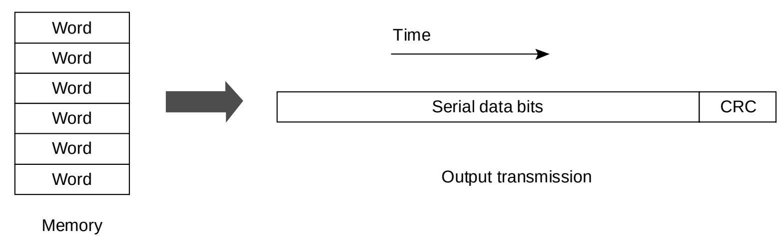
# Fig 8.21 CRC Generator Based on the Polynomial $x^{16} + x^{12} + x^5 + 1$




- The number and position of XOR gates is determined by the polynomial.
- CRC does not support error correction but the CRC bits generated can be used to detect multibit 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.

50

# Fig 8.22 Serial Data Transmission with Appended CRC Code




51

# Chapter 8 Summary

- **I/O subsystem has characteristics that make it different from main memory**
  - **Speed variations**
  - **Latency**
  - **Band width**
- **This leads to 3 different kinds of I/O:**
  - **Programmed I/O handled completely by software, from initiation until completion**
  - **Interrupt-driven I/O combines hardware for initiation and software for completion**
  - **DMA allows an all-hardware approach to I/O activities**
- **External connections to devices may require data format changes, and error detection and possibly correction**

52