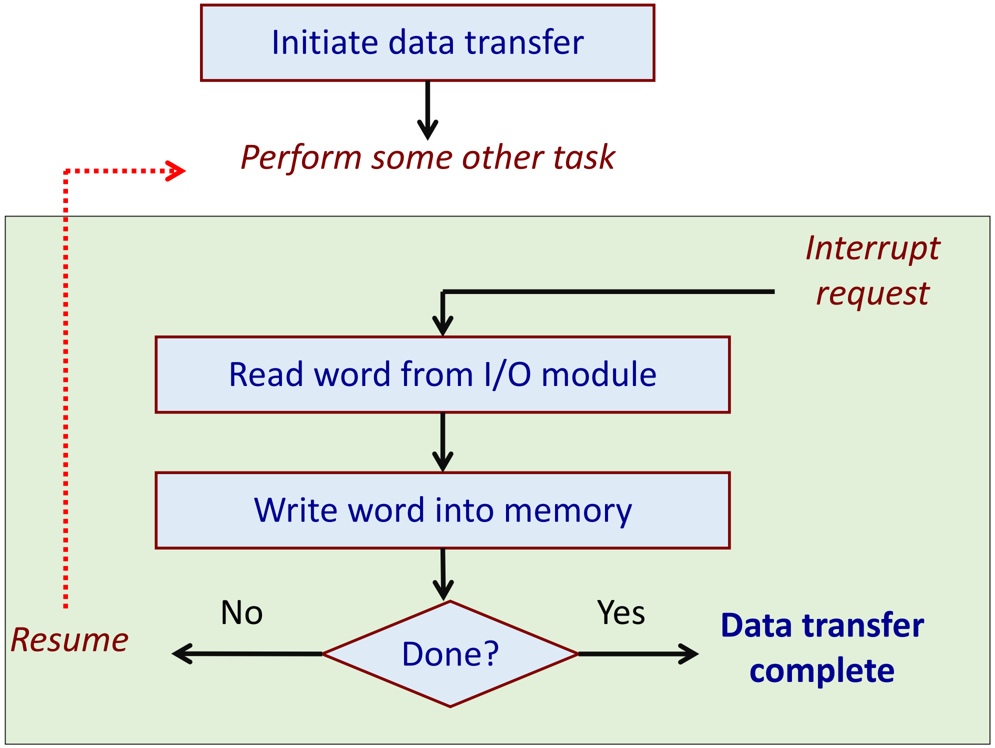
(c) Interrupt-Driven Data Transfer

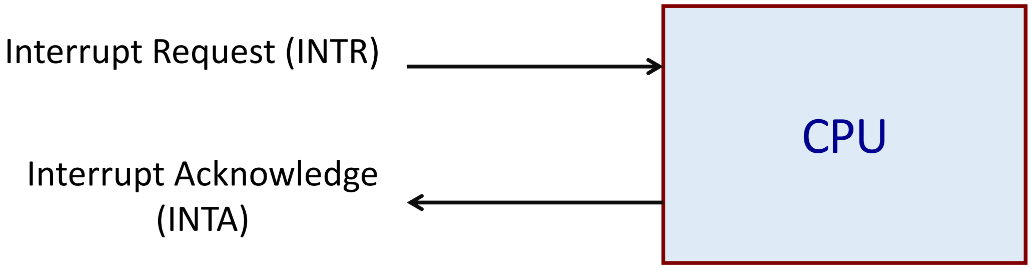
- The CPU initiates the data transfer and proceeds to perform some other task.
- When the I/O module is ready for data transfer, it informs the CPU by activating a signal (called *interrupt request*).
- The CPU suspends the task it was doing, services the request (that is, carries out the data transfer), and returns back to the task it was doing.
- Characteristics:
  - CPU time is not wasted while checking the status of the I/O module.
  - CPU time is required only during data transfer, plus some overheads for transferring and returning control.



This part of the program that gets activated when interrupt request comes is called *interrupt handler* or *interrupt service subroutine* (ISS).

Some Features of Interrupt-Driven Data Transfer

- How is ISS different from a normal subroutine or function?
  - A function is called from well-defined places in the calling program.
    - Only the relevant registers need to be saved on entry to the function, and restored before return.
  - The ISS can get invoked from anywhere in the program that was executing.
    - Depends on when the interrupt request signal arrived.
    - So potentially all the registers that are used in the ISS needs to be saved and restored.



- We shall learn later why the *INTA* signal is required.

33

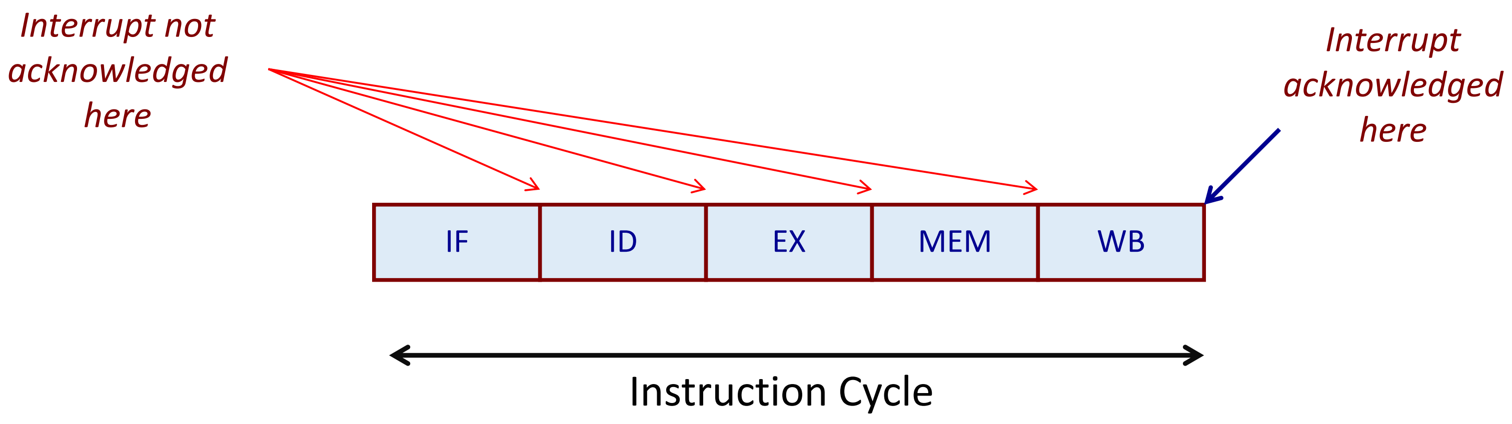
Some Challenges in Interrupts

- For multiple sources of interrupts, how to know the address of the ISS?
- How to handle multiple interrupts?
  - While an interrupt request is being processed, another interrupt request might come.
  - Enabling, disabling and masking of interrupts.
- How to handle simultaneously arriving interrupts?
- Sources of interrupts other than I/O devices.
  - Exceptions, TRAP / System Call, etc.

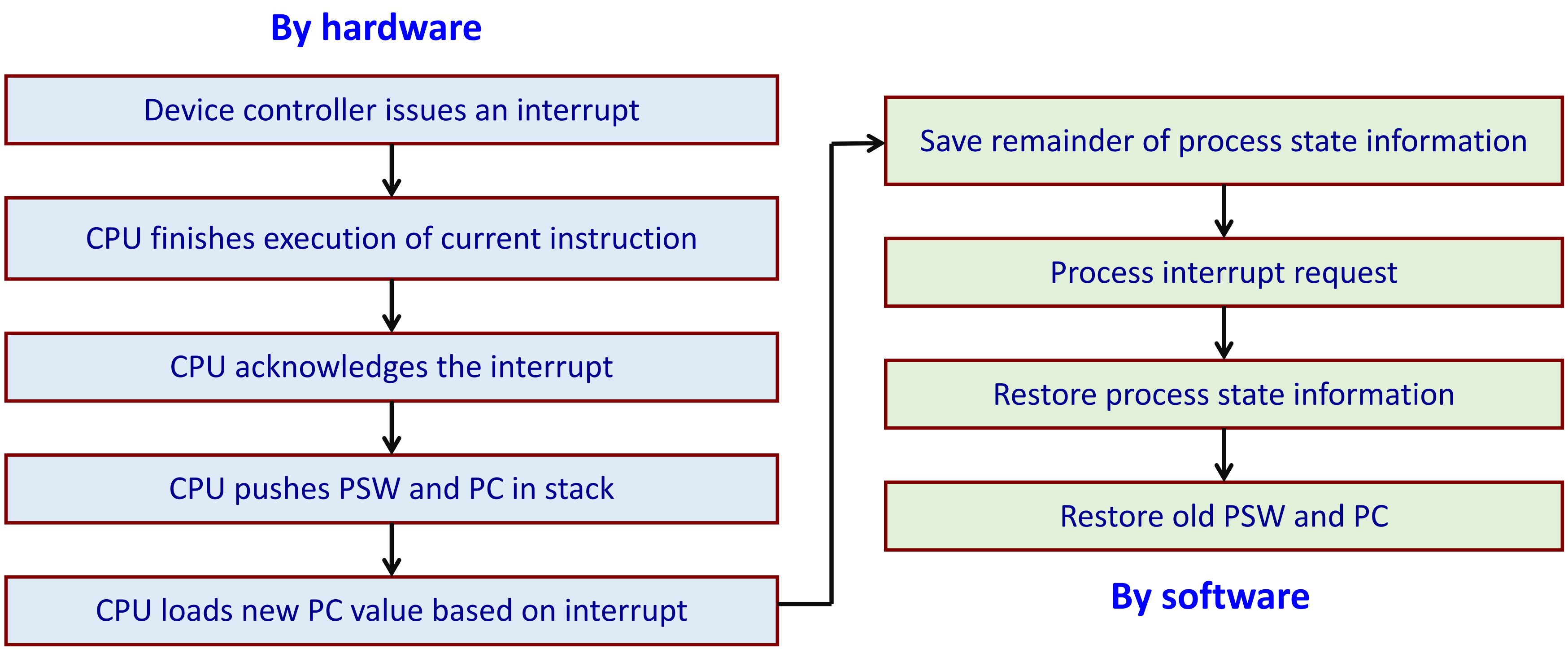
What happens when an interrupt request arrives?

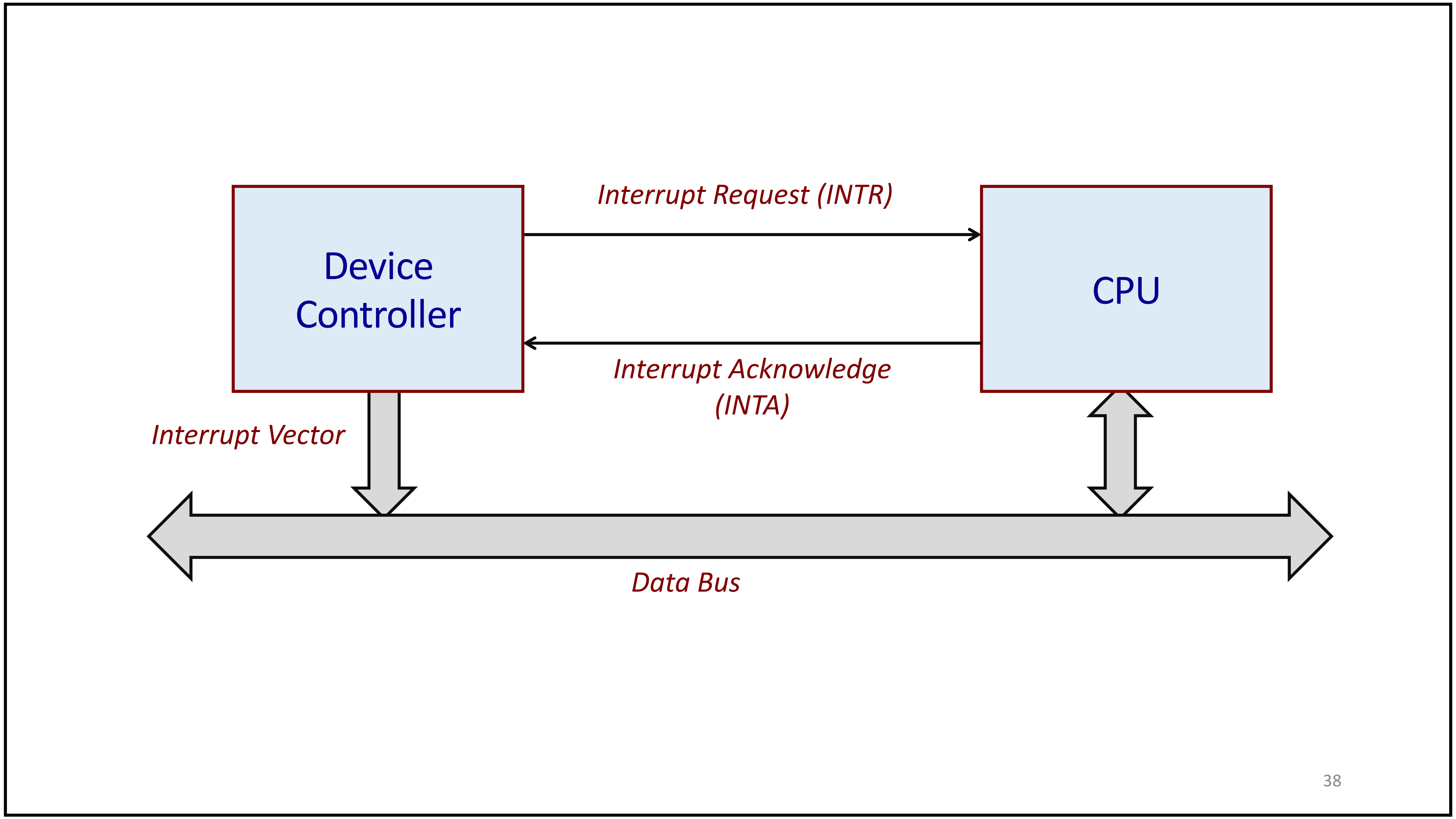
- At the end of the current instruction execution, the *PC* and *program status word* (PSW) are saved in stack automatically.
  - PSW contains status flags and other processor status information.
- The interrupt is acknowledged, the interrupt vector obtained, based on which control transfers to the appropriate ISS.
  - Different interrupting devices may have different ISS's.
- After handling the interrupt, the ISR executes a special *Return From Interrupt* (RTI) instruction.
  - Restores the PSW and returns control to the saved PC address.
  - Unlike normal RETURN where PSW is not restored.

- An instruction cycle typically consists of several machine cycles.
  - For MIPS32, there are 5 machine cycles.

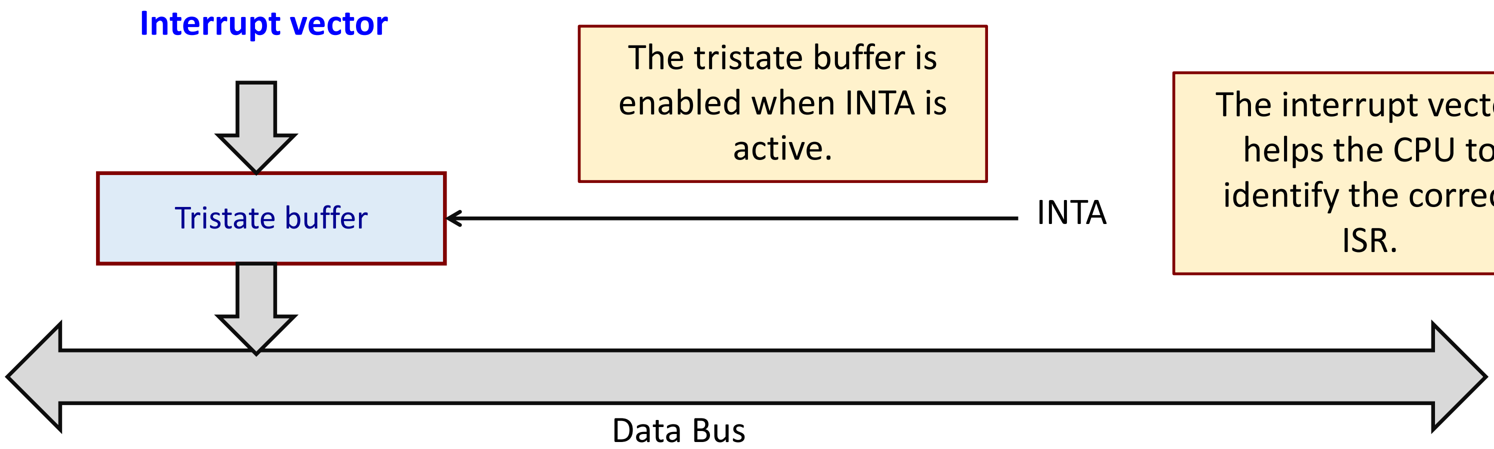


General Interrupt Processing



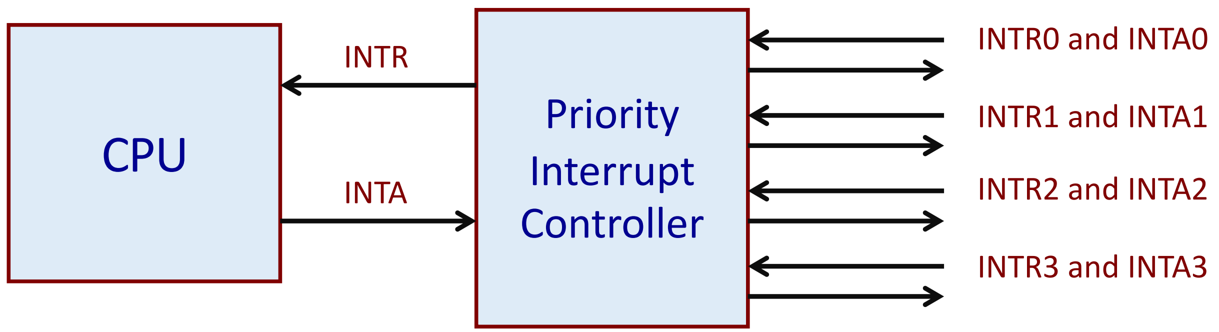

Device Controller
CPU
Interrupt Request (INTR)
Interrupt Acknowledge (INTA)
Interrupt Vector
Data Bus
38

- How is the interrupt vector sent on the data bus in response to INTA?
  a) Device controller sends *INTR* to the CPU.
  b) CPU finishes the current instruction and sends back *INTA*.
  c) Device controller sends *interrupt vector* (or number) over data bus.
  d) CPU reads the interrupt vector, and identifies the device.



Multiple Devices Interrupting the CPU

- A common solution is to use a *priority interrupt controller*.
  - The interrupt controller interacts with CPU on one side and multiple devices on the other side.
  - For simultaneous interrupt requests, interrupt priority is defined.
  - The interrupt controller is responsible for sending the interrupt vector to CPU.



- How it works?
  - The *INTR* line is made active when one or more of the device(s) activate their interrupt request line.

    $$INTR = INTR0 + INTR1 + INTR2 + INTR3$$

  - When the CPU sends back *INTA*, the interrupt controller sends back the corresponding acknowledge to the interrupting device, and puts the interrupt vector on the data bus.
  - The interrupt controller is programmable, where the interrupt vectors for the various interrupts can be programmed (specified).
  - For more than one interrupt request simultaneously active, a priority mechanism is used (e.g. *INTR0* is highest priority, followed by *INTR1*, etc.).

How is interrupt nesting handled?

- Consider the scenario:
  a) A device *D*0 has interrupted and the CPU is executing the ISS for *D*0.
  b) In the mean time, another device *D*1 has interrupted.
- Two possible scenarios here:
  1) *D*1 will interrupt the ISS for *D*0, get processed first, and then the ISS for *D*0 will be resumed. → *CREATES PROBLEM FOR MULTI NESTING*
  2) Disable the interrupt system automatically whenever an interrupt is acknowledged so that handling of nested interrupts is not required.

- Typical instruction set architectures have the following instructions:
  - EI : Enable interrupt
  - DI : Disable interrupt
- For the second scenario as discussed, the ISS will give an *EI* instruction just before *RTI*.
  - Some ISA combine *EI* and *RTI* in a single instruction.
- The *DI* instruction is sometimes used by the operating system to execute atomic code (e.g. semaphore wait and signal operations).
  - Nobody should interrupt the code while it is being executed.

Cases that make interrupt handling difficult

- For some interrupts, it is not possible to finish the execution of the current instruction.
  - A special *RETURN* instruction is required that would return and restart the interrupted instructions.
- Some examples:
  - **a)** **Page fault interrupt**: A memory location is being accessed that is not presently available in main memory.
  - **b)** **Arithmetic exception**: Some error has occurred during some arithmetic operation (e.g. division by zero).

Handling Multiple Devices

- Suppose that a number of devices capable of generating interrupts are connected to the CPU.
- The following questions need to be answered.
  a) How can the CPU identify the interrupting device?
  b) How can the CPU obtain the starting address of the appropriate ISS?
  c) Should interrupt nesting be allowed?
  d) How should two or more simultaneous interrupt requests be handled?

45

(a) Device Identification

- Suppose that an external device requests an interrupt by activating an *INTR* line that is common to all the devices. That is,

$$INTR = INTR_1 + INTR_2 + \ldots + INTR_n$$

- Each device can have a status bit indicating whether it has interrupted.
  - CPU can *poll* the status bits to find out who has interrupted.
- A better alternative is to use the *interrupt vector* concept discussed earlier.
  - The interrupting device sends a special identifying code on the data bus upon receiving the interrupt acknowledge.

(b) Find Starting Address of ISS

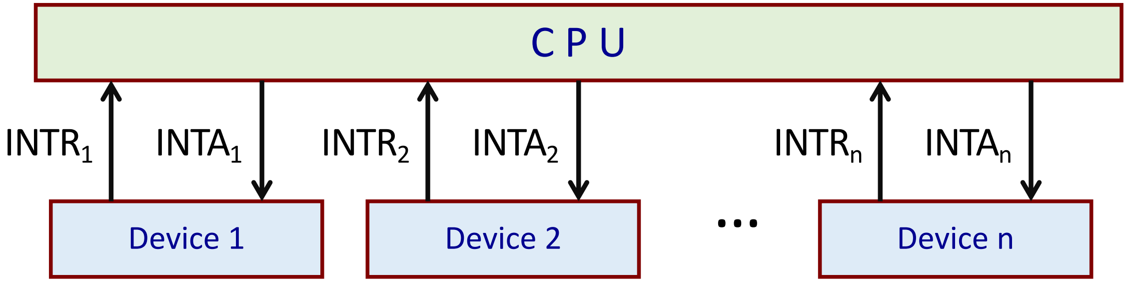
- For a processor with multiple interrupt request inputs, the address of the ISS can be fixed for each individual input.
  - Lacks flexibility.
- If we use the interrupt vector scheme discussed earlier, the device is able to identify itself to the CPU.
  - CPU can then lookup a table where the ISS addresses for all the devices are stored.
  - The interrupt latency is somewhat increased, since we are not immediately jumping to the ISS.

(c) Interrupt Nesting

- A simple approach:
  - Disable all interrupts during the execution of an ISS.
  - This ensures that the interrupt request from one device will not cause more that one interruptions.
  - ISS's are typically short, and the delay they may cause in handling a second interrupt request is often acceptable.
- *Interrupt priority:*
  - Some interrupting devices may be assigned higher priorities than others.
  - Example: timer interrupt to maintain a real-time clock.
  - Higher priority interrupt may interrupt the ISS of lower priority ones.

(d) Simultaneous Requests

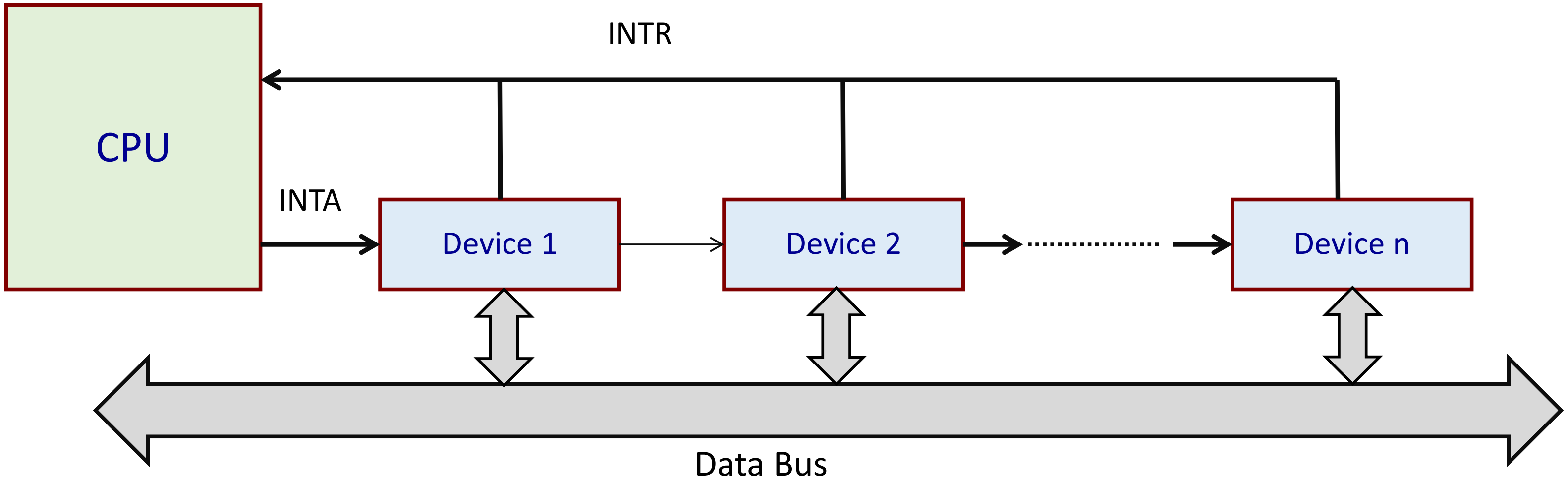
- Here we consider the problem of simultaneous arrivals of interrupt requests from two or more devices.
  - CPU should have some mechanism by which only one request is serviced while the others are delayed or ignored.
  - If the CPU has multiple interrupt request lines, it can have a *priority scheme* where it accepts the request with the highest priority.



$INTA_i$ has higher priority than $INTR_j$, where $i < j$.

- Another way to assign priority is to use polling using *daisy chaining*.
  - In polling, priority is automatically assigned based on the order in which the devices are polled.
  - In daisy chain connection, the *INTR* line is common to all the devices, but the *INTA* line is connected in a daisy chain fashion allowing it to propagate serially through the devices.
  - A device when it receives *INTA*, passes the signal to the next device only if it had not interrupted. Else, it stops the propagation of *INTA*, and puts the identifying code on the data bus.
  - Thus, the device that is electrically closest to the CPU will have the highest priority.

Daisy Chain Arrangement

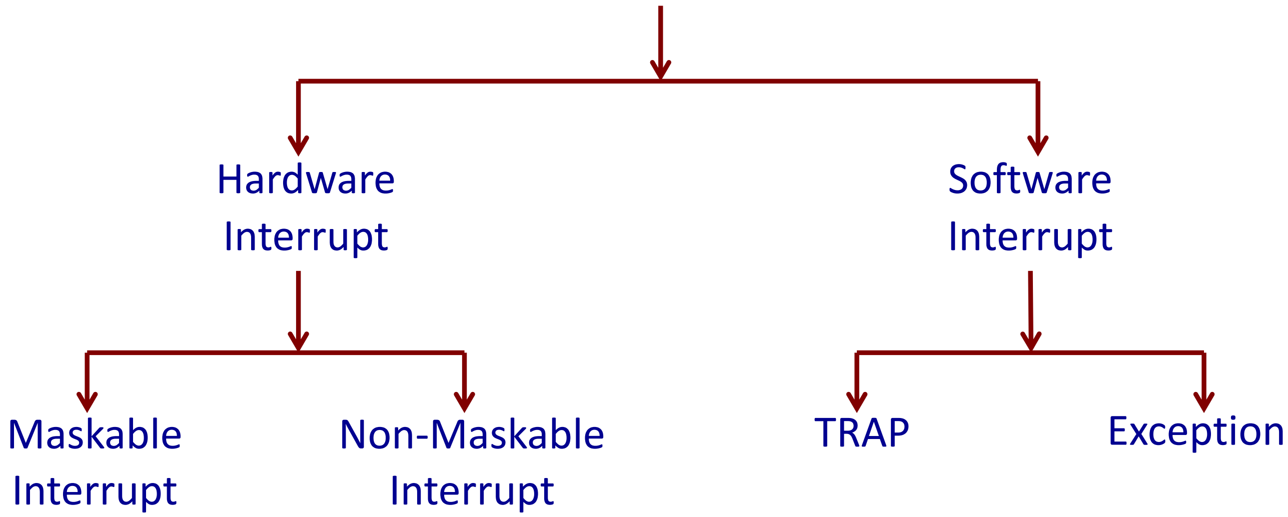


51

51

51

Types of Interrupts



- *Hardware Interrupt:*
  - The interrupt signal is coming from a device external to the CPU.
  - Example: keyboard interrupt, timer interrupt, etc.
- *Maskable Interrupt:*
  - Hardware interrupts that can be masked or delayed when a higher priority interrupt request arrives.
  - There are processor instructions that can selectively mask and unmask the interrupt request lines of the CPU.

- *Non-Maskable Interrupt:*
  - Interrupts that cannot be delayed and should be handled by the CPU immediately.
  - Examples: power fail interrupts, real-time system interrupts, etc.
- *Software Interrupt:*
  - They are caused due to execution of some instructions.
  - Not caused due to external inputs.
- *TRAP:*
  - They are special instructions used to request services from the operating system.
  - Also called *system calls*.
- *Exception:*
  - These are unplanned interrupts generated while executing a program.
  - They are generated from within the system.
  - Examples: invalid opcode, divide by zero, page fault, invalid memory access, etc.

Direct Memory Access (DMA)

55

Introduction

- In the data transfer methods discussed under programmed I/O, it is assumed that machine instructions are used to transfer the data between I/O device and memory.
  - Not very suitable when large blocks of data are required to be transferred at high speed (e.g. transfer of a disk block).
- An alternate approach is *Direct Memory Access* (DMA).
  - Allows transfer of a block of data directly between an I/O device and memory, without continuous CPU intervention.

- Why programmed I/O is not suitable for high-speed data transfer?

a) Several program instructions have to be executed for each data word transferred between the I/O device and memory.

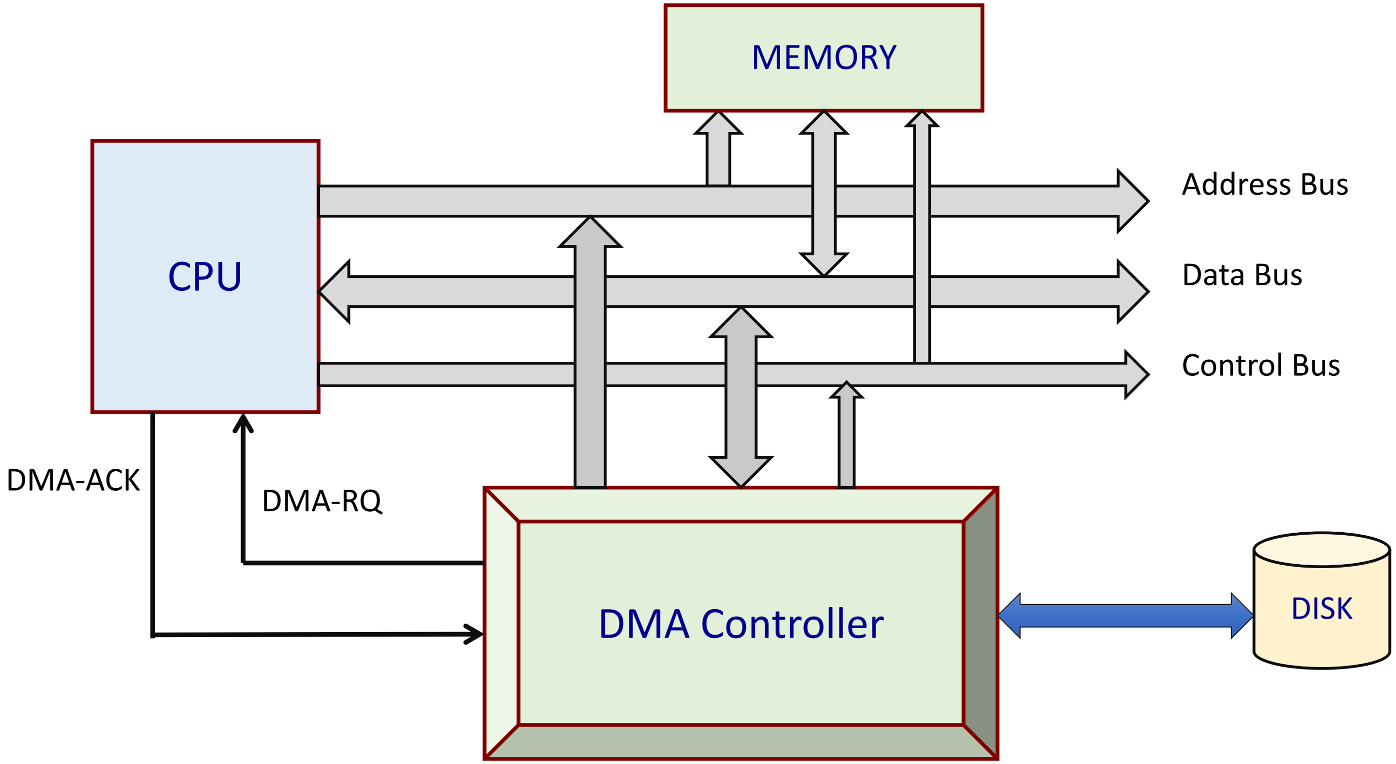
  - Suppose 20 instructions are required for each word transfer.
  - The CPI of the machine running at 1 GHz clock is 1.
  - So, 20 nsec is required for each word transfer → maximum 50 M words/sec
  - Data transfer rates of fast disks are higher than this figure.

b) Many high speed peripheral devices like disk have a synchronous mode of operation, where data are transferred at a fixed rate.

- Consider a disk rotating at 7200 rpm, with average rotational delay of 4.15 msec.
- Suppose there are 64 Kbytes of data recorded in every track.
- Once the disk head reaches the desired track, there will be a sustained data transfer at rate 64 Kbytes / 4.15 msec = 15.4 MBps.
- This sustained data transfer rate is comparable to the memory bandwidth, and cannot be handled by programmed I/O.

DMA Controller

- A hardwired controller called the *DMA controller* can enable direct data transfer between I/O device (e.g. disk) and memory without CPU intervention.
  - No need to execute instructions to carry out data transfer.
  - Maximum data transfer speed will be determined by the rate with which memory read and write operations can be carried out.
  - Much faster than programmed I/O.


MEMORY
CPU
Address Bus
Data Bus
Control Bus
DMA-ACK
DMA-RQ
DMA Controller
DISK

Steps Involved

a) When the CPU wants to transfer data, it initializes the DMA controller.
   - How many bytes to transfer, address in memory for the transfer.

b) When the I/O device is ready for the transfer, the DMA controller sends *DMA-RQ* signal to the CPU.

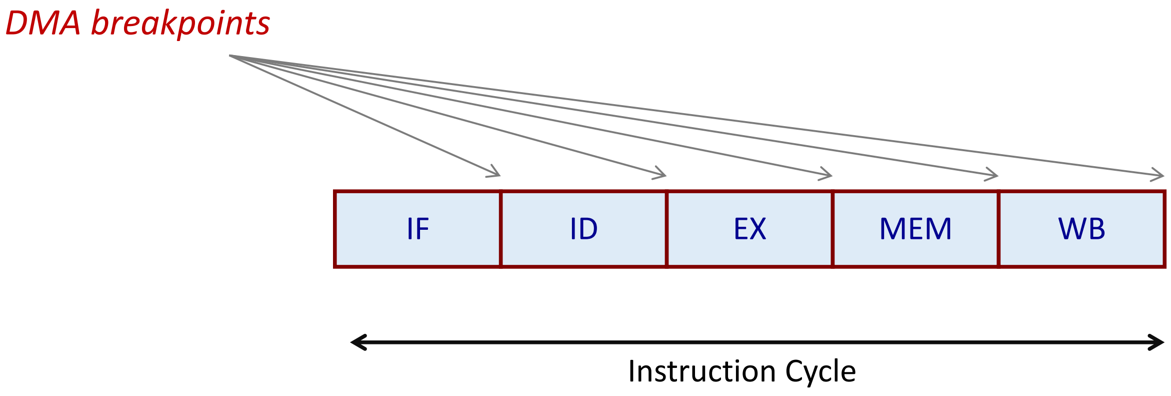
c) CPU waits till the next DMA breakpoint, relinquishes control of the bus (i.e. puts them in high impedance state), and sends *DMA-ACK* to DMA controller.

d) Now DMA controller enables its bus interface, and transfers data directly to/from memory.

e) When done, it deactivates the *DMA-RQ* signal.

f) The CPU again begins to use the bus to access memory.

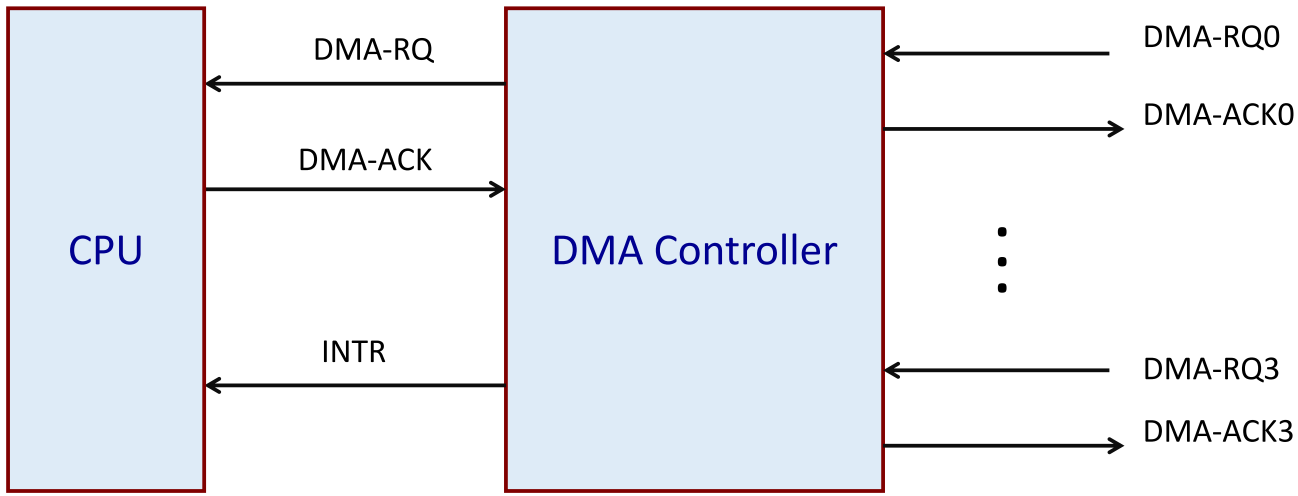
- The DMA breakpoints:
  - DMA request can be acknowledged at the end of any machine cycle.



Why cannot we have interrupt breakpoints at the end of any machine cycle?

- For every DMA channel, the DMA controller will have three registers:
  a) Memory address
  b) Word count
  c) Address of data on disk
- CPU initializes these registers before each DMA transfer operation.
- Before the data transfer, DMA controller requests the memory bus from the CPU.
- When the data transfer is complete, the DMA controller sends an interrupt signal to the CPU.

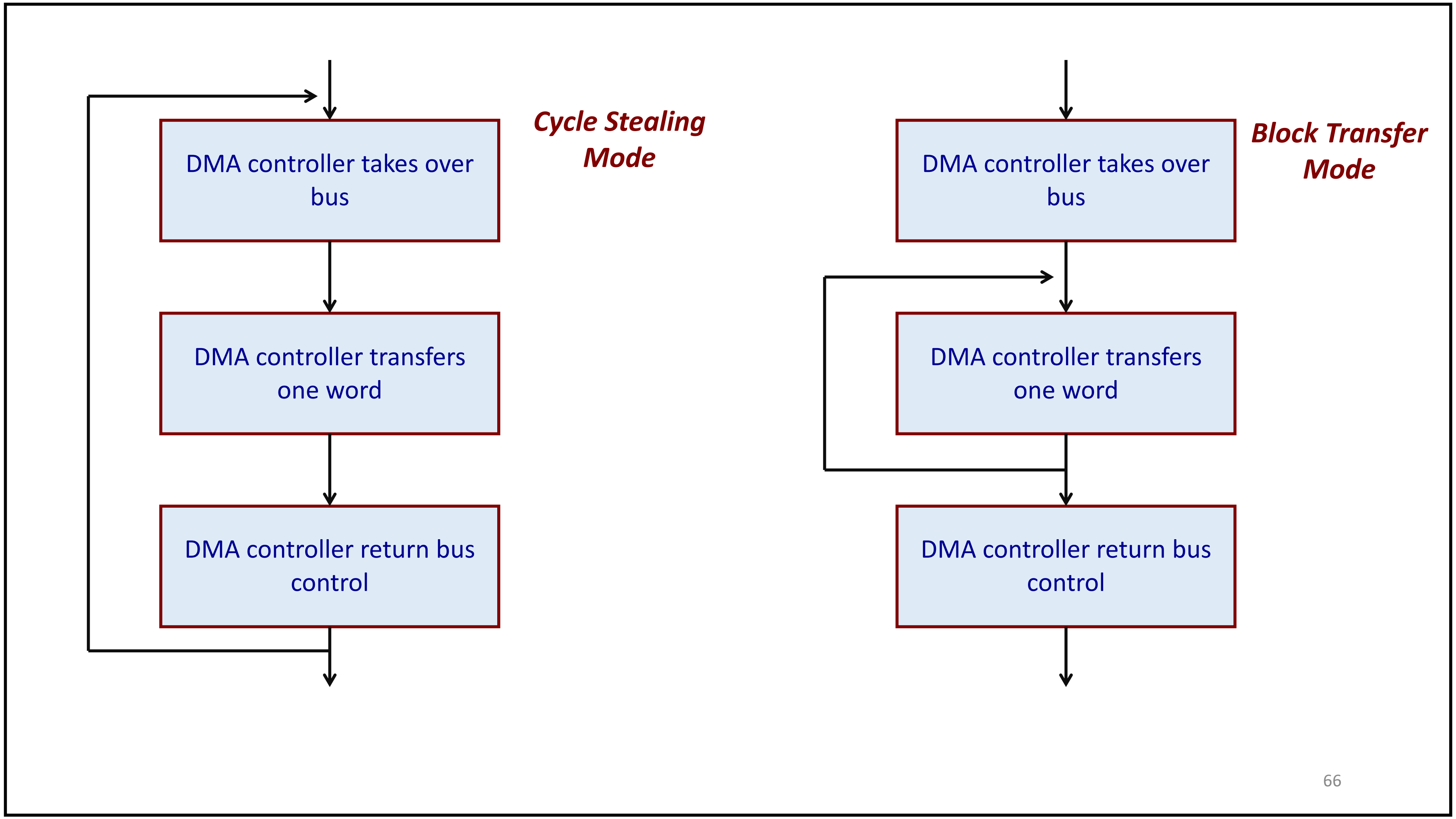
63


CPU
DMA Controller
DMA-RQ
DMA-ACK
INTR
DMA-RQ0
DMA-ACK0
DMA-RQ3
DMA-ACK3

DMA Transfer Modes

- DMA transfer can take place in two modes:
  - **a) DMA cycle stealing**
    - The DMA controller requests for the for a few cycles 1 or 2.
    - Preferably when the CPU is not using memory.
    - DMA controller is said to steal cycles from the CPU without the CPU knowing it.
  - **b) DMA block transfer**
    - The DMA controller transfers the whole block of data without interruption.
    - Results in maximum possible data transfer rate.
    - CPU will lie idle during this period as it cannot fetch any instructions from memory.

65


Cycle Stealing Mode
DMA controller takes over bus
DMA controller transfers one word
DMA controller return bus control
Block Transfer Mode
DMA controller takes over bus
DMA controller transfers one word
DMA controller return bus control
66

Others Applications of DMA

- Other than data transfer to/from high-speed peripheral devices, DMA can be used in some other areas as well:
  - High-speed memory-to-memory block move.
  - Refreshing dynamic memory systems, by periodically generating dummy read requests to the columns.