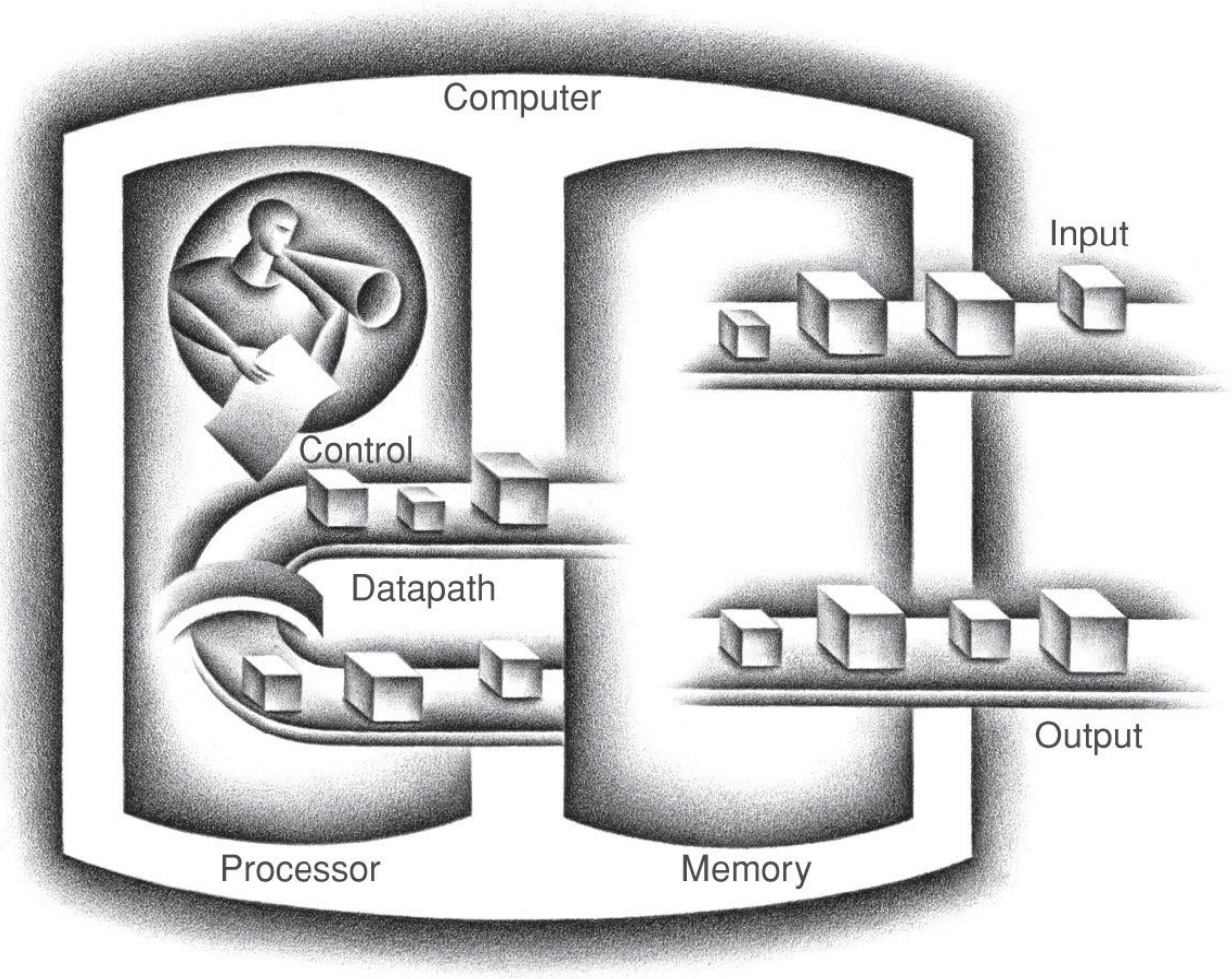
Chapter 7: I/O System

Ngo Lam Trung, Pham Ngoc Hung, Hoang Van Hiep

[with materials from *Computer Organization and Design*, MK and M.J. Irwin's presentation, PSU 2008]

Computer Organization

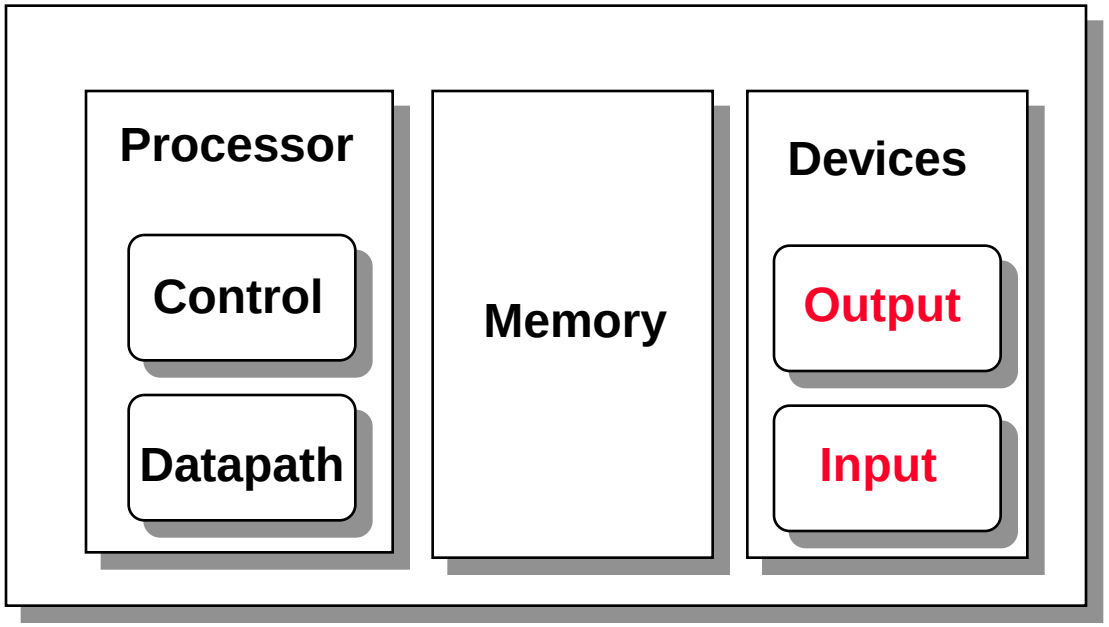
- Computer needs the interface to communicate with outside world



Review: Major Components of a Computer

- Input + Output = I/O system
  - Hard disk
  - Network
  - USB drive

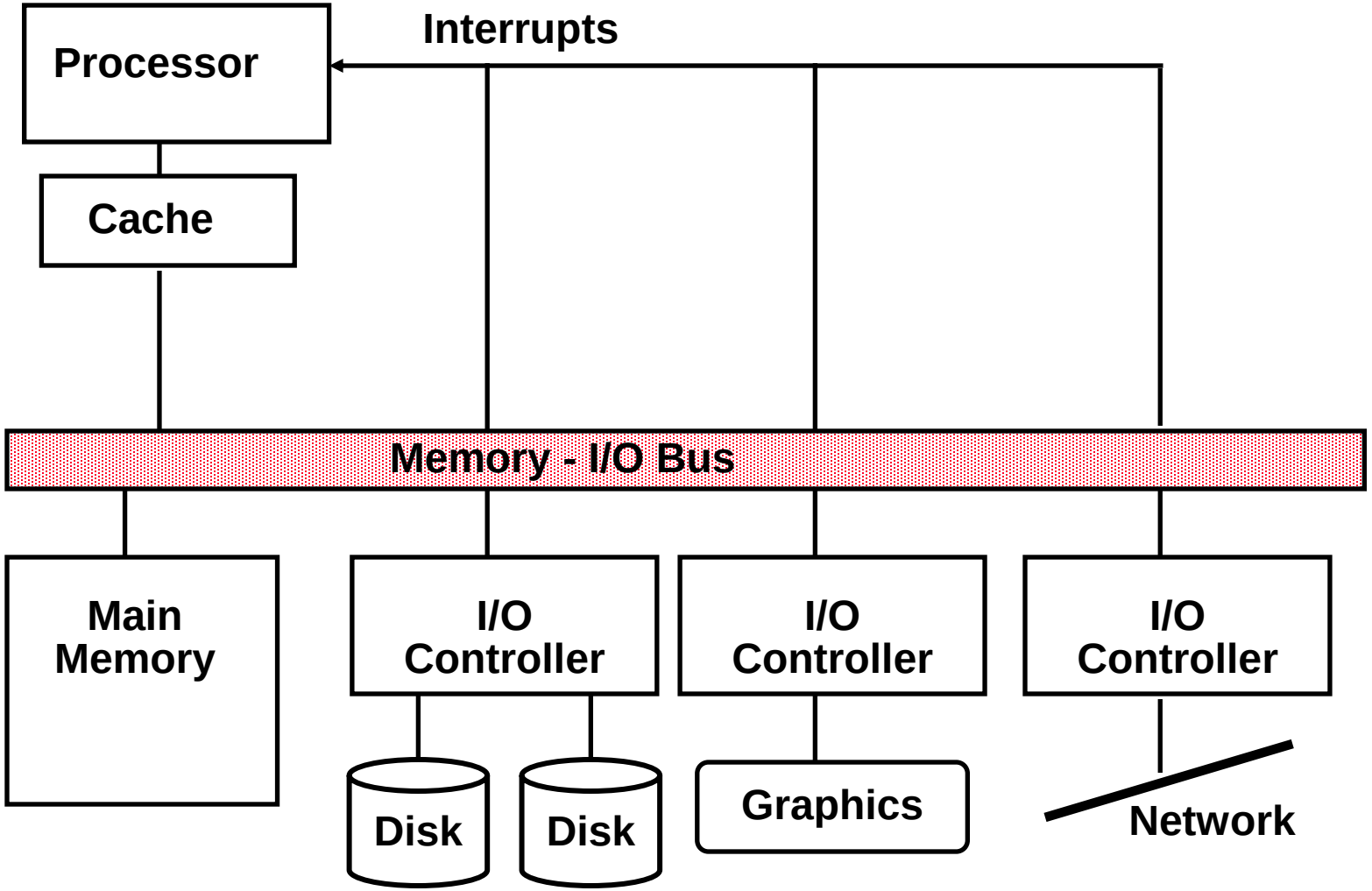
and thousands of other devices…



Important metrics

- For processor and memory: performance and cost
- For I/O system: what are the most important?
  - Performance
  - Expandability
  - Dependability
  - Cost, size, weight
  - Security
  - …

A Typical I/O System



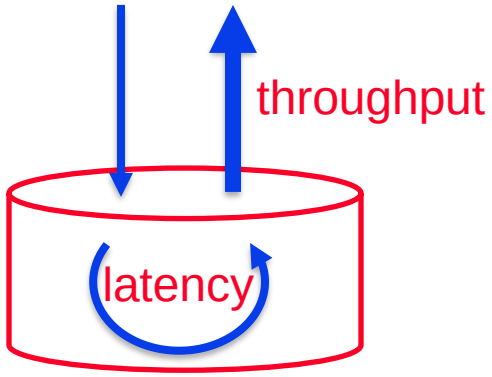
Input and Output Devices

- I/O devices are incredibly diverse with respect to
  - **Behavior** – input, output or storage
  - **Partner** – human or machine
  - **Data rate** – the peak rate at which data can be transferred between the I/O device and the main memory or processor

Device	Behavior	Partner	Data rate (Mb/s)
Keyboard	input	human	0.0001
Mouse	input	human	0.0038
Laser printer	output	human	3.2000
Magnetic disk	storage	machine	800.0000-3000.0000
Graphics display	output	human	800.0000-8000.0000
Network/LAN	input or output	machine	100.0000-10000.0000

I/O Performance Measures

- I/O bandwidth (throughput) – amount of information that can be input/output and communicated across an interconnect between the processor/memory and I/O device per unit time
  1. How much data can we move through the system in a certain time?
  2. How many I/O operations can we do per unit time?



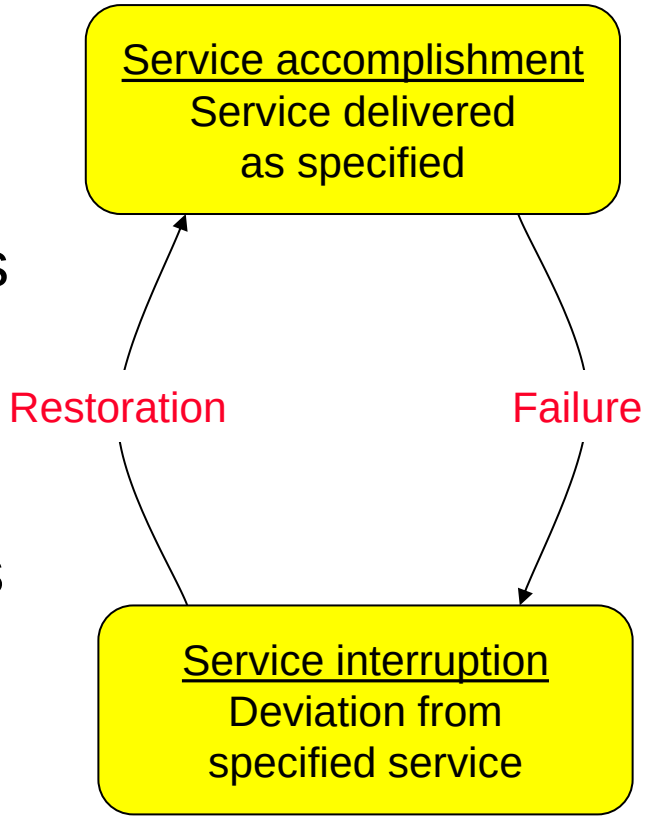
- I/O response time (latency) – the total elapsed time to accomplish an input or output operation

- Many applications require *both* high throughput and short response times

Failure

- Hardware operates in two states:
  1. *Service accomplishment*, the service is delivered as specified.
  2. *Service interruption*, the delivered service is different from the specified service.

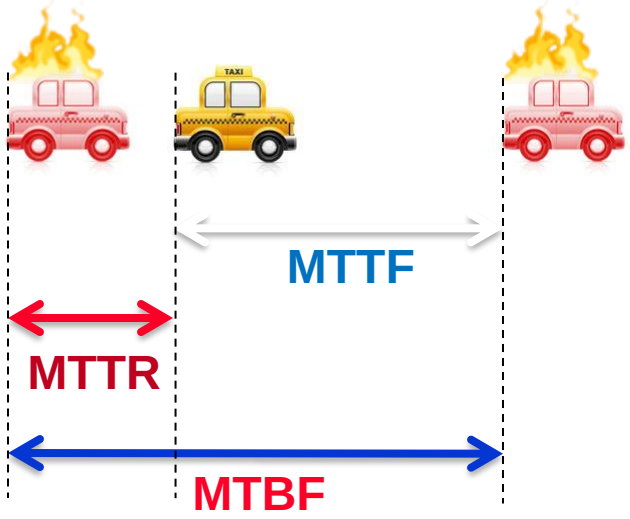
- Changes from (1) to (2): failures
- Changes from (2) to (1): restorations
- Permanent failure: service is stopped permanently
- Intermittent failure: system oscillates between the two states → difficult to diagnose



Dependability: Reliability and Availability

- Mean Time To Failure (MTTF): average time of normal operation between two consecutive failure
- Mean Time To Repair (MTTR): average time of service interruption when failure occurs
- Reliability: measured by MTTF
- Availability:

$$\text{Availability} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}$$



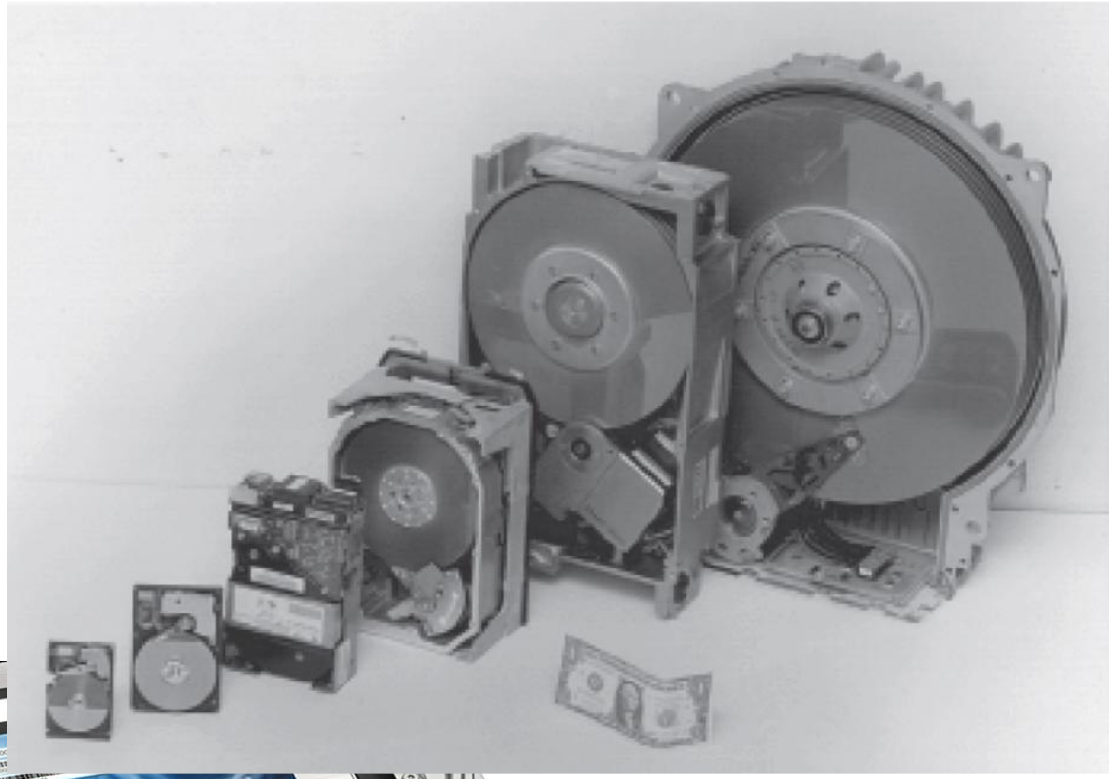
- Example MTTF:
  - Seagate ST33000655SS: 1,400,000 hours @25˚C
  - Samsung 860 EVO SSD: 1,500,000 hours

Improving MTTF and availability

- Fault avoidance: making better quality hardware
- Fault tolerance: using redundancy for back up and maintain service even in case of fault
- Fault forecasting: predicting when fault may happen to replace component before it fails → shorten MTTR
  - SMART: hardware failure prediction

Disk storage

- HDD (magnetic)
- SSD (solid-state)



System Interconnection

- Processor
- Memory
- I/O devices

- How to connect them physically?

I/O System Interconnect Issues

- A bus is a shared communication link (a single set of wires used to connect multiple subsystems) that needs to support a range of devices with widely varying latencies and data transfer rates
  - Advantages
    - Versatile – new devices can be added easily and can be moved between computer systems that use the same bus standard
    - Low cost – a single set of wires is shared in multiple ways
  - Disadvantages
    - Creates a communication bottleneck – bus bandwidth limits the maximum I/O throughput

- The maximum bus speed is largely limited by
  - The length of the bus
  - The number of devices on the bus

Types of Buses

- Processor-memory bus (“Front Side Bus”, proprietary)
  - Short and high speed
  - Matched to the memory system to maximize the memory-processor bandwidth
  - Optimized for cache block transfers
- I/O bus (industry standard, e.g., SCSI, USB, Firewire)
  - Usually is lengthy and slower
  - Needs to accommodate a wide range of I/O devices
  - Use either the processor-memory bus or a backplane bus to connect to memory
- Backplane bus (industry standard, e.g., ATA, PCIexpress)
  - Allow processor, memory and I/O devices to coexist on a single bus
  - Used as an intermediary bus connecting I/O busses to the processor-memory bus

I/O Transactions

- An I/O transaction is a sequence of operations over the interconnect that includes a request and may include a response either of which may carry data.
- An I/O transaction typically includes two parts
  1. Sending the address
  2. Receiving or sending the data

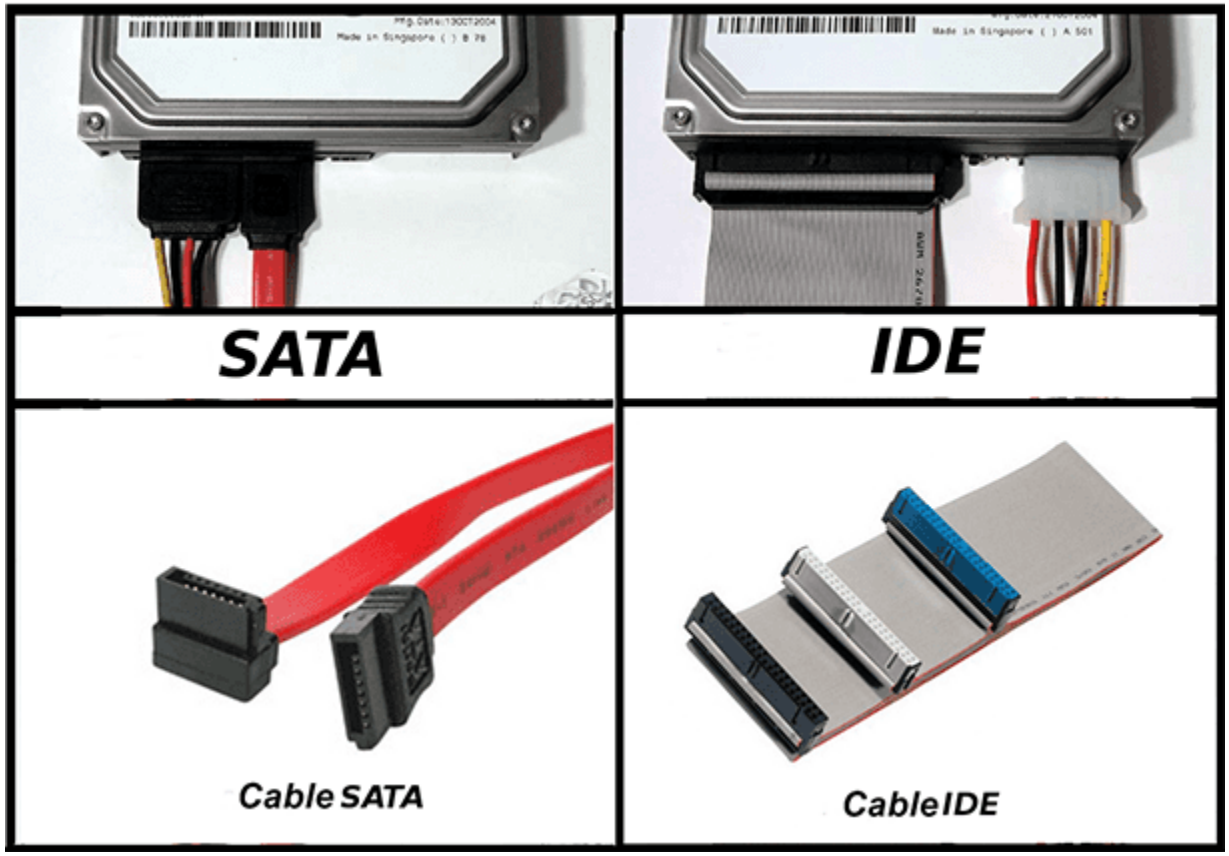
- Bus transactions are defined by what they do to memory
  - A read transaction reads data from memory (to either the processor or an I/O device) — output
  - A write transaction writes data to the memory (from either the processor or an I/O device) — input

Synchronous and Asynchronous Buses

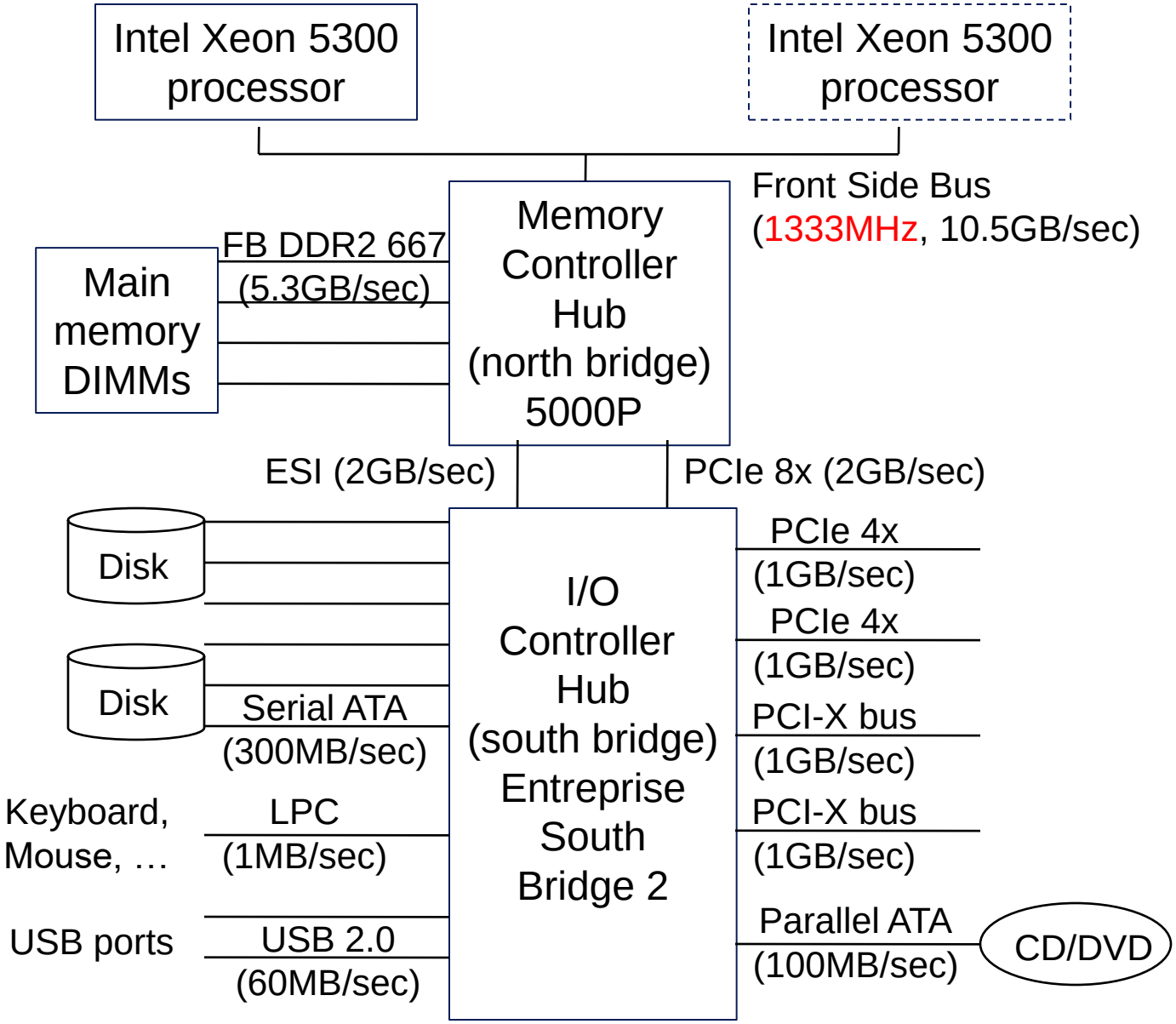
- Synchronous bus (e.g., processor-memory buses)
  - Includes a clock in the control lines and has a fixed protocol for communication that is relative to the clock
  - Advantage: involves very little logic and can run very fast
  - Disadvantages:
    - Every device communicating on the bus must use same clock rate
    - Short distance
- Asynchronous bus (e.g., I/O buses)
  - It is not clocked, so requires a handshaking protocol and additional control lines (ReadReq, Ack, DataRdy)
  - Advantages:
    - Can accommodate a wide range of devices and device speeds
    - Can be lengthened without worrying about clock skew or synchronization problems
  - Disadvantage: slow(er)

Example: Synchronous to asynchronous

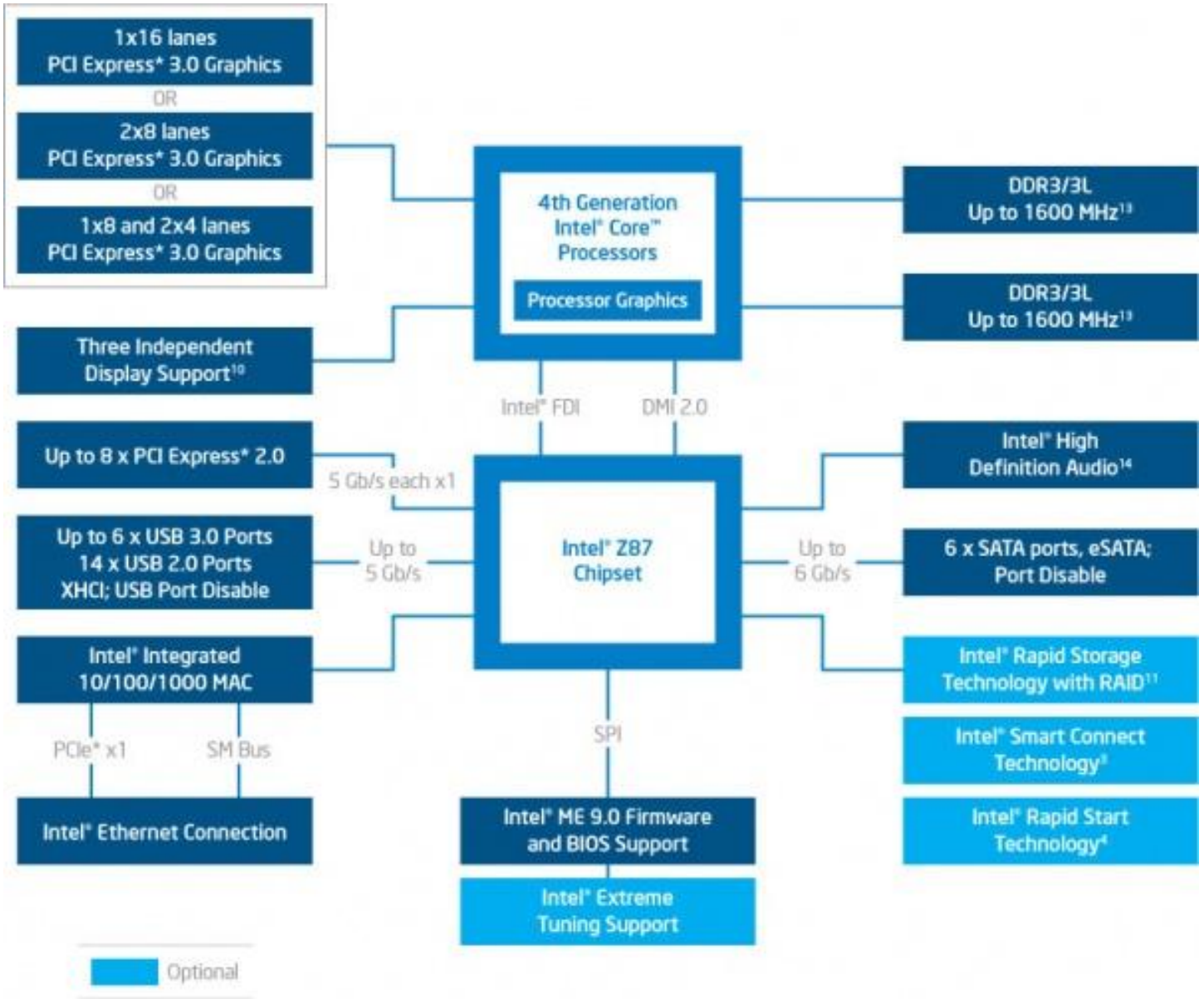
- It is difficult to use parallel wires running at a high clock rate ➔ a few one-way wires running at a very high "clock" rate (~2GHz)



A Typical I/O System



Intel Core i7 with Z87 chipset



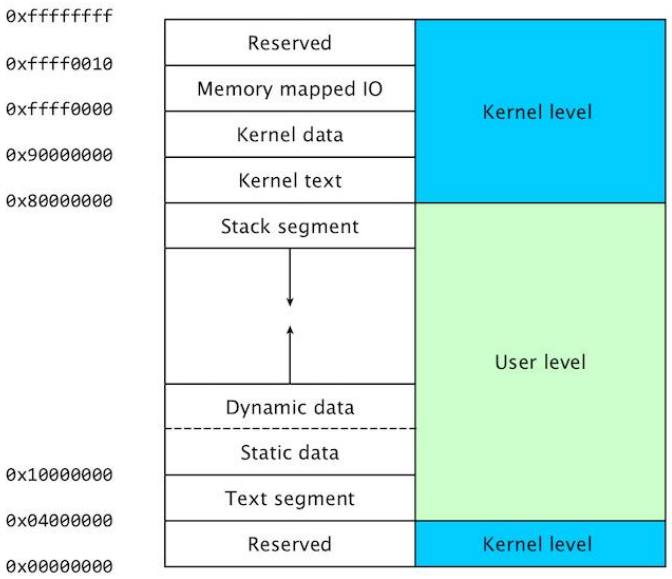
Interfacing I/O Devices

- Physical connection is done, now how about data transfer?

- How is a user I/O request transformed into a device command and communicated to the device?
- How is data actually transferred to or from a memory location?
- What is the role of the operating system?

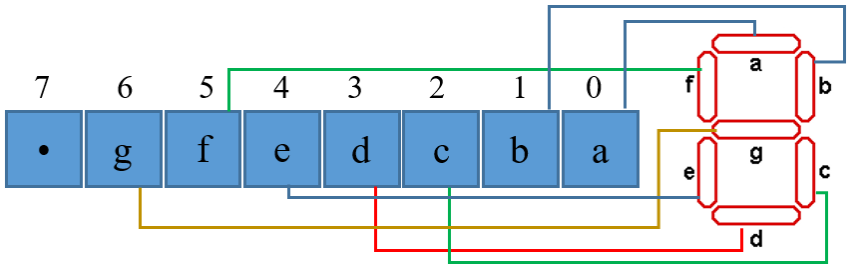
Communication of I/O Devices and Processor

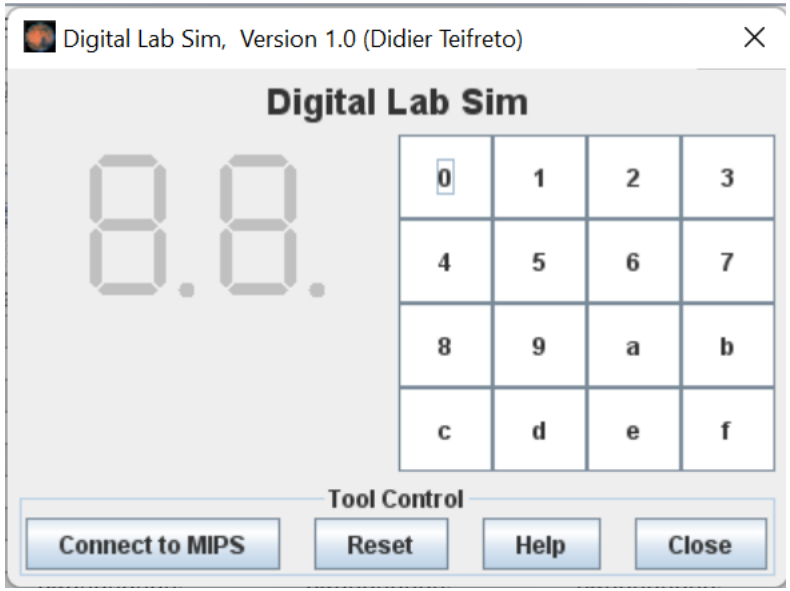
- How the processor directs (find) the I/O devices
  - Special I/O instructions
    - Must specify both the device and the command
  - Memory-mapped I/O
    - I/O devices are mapped to memory addresses
    - Read and writes to those memory addresses are interpreted as commands to the I/O devices
    - Load/stores to the I/O address space can *only* be done by the OS
- Memory map



Example: controlling 7-seg LED in MARS

- Tools→Digital Lab Sim: 2x 7-seg LEDs display
  - Byte value at address 0xFFFF0010 : command right seven segment display
  - Byte value at address 0xFFFF0011 : command left seven segment display

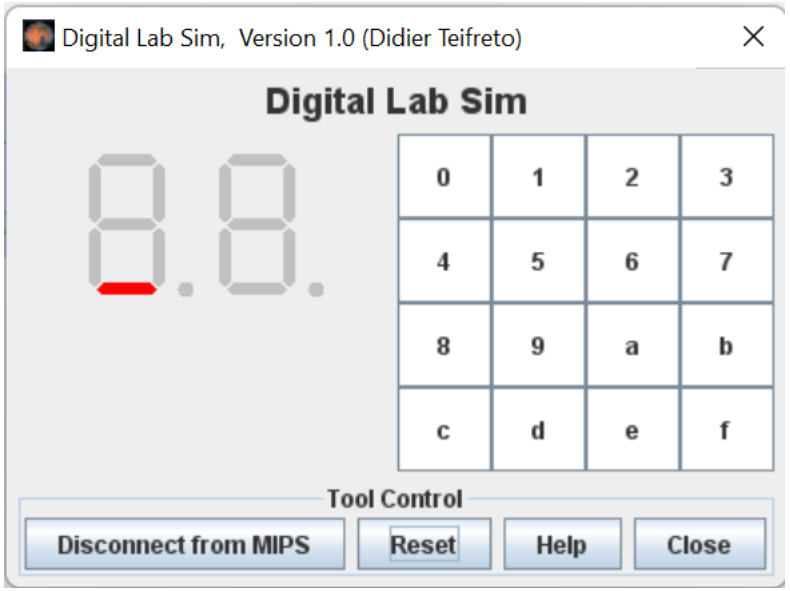




Example: controlling 7-seg LED in MARS

- Tools→Digital Lab Sim: 2x 7-seg LEDs display
  - Byte value at address 0xFFFF0010 : command right seven segment display
  - Byte value at address 0xFFFF0011 : command left seven segment display

```
li    $a0,  0x8          #value
li    $t0,  0xFFFF0011 #address
sb    $a0,  0($t0)    #turn-on
```



Exercise

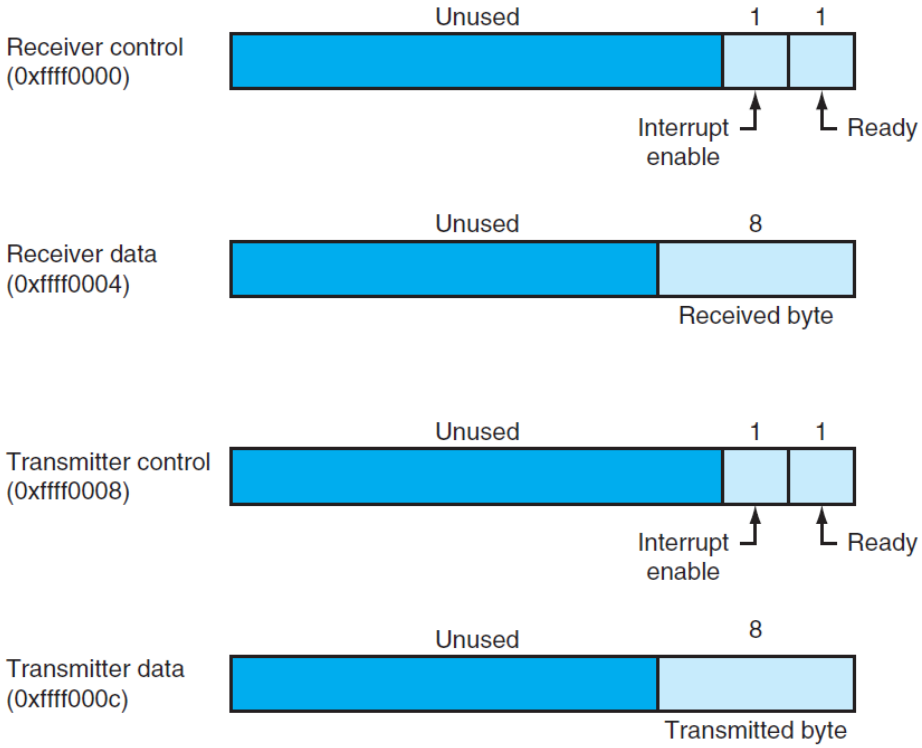
- Write program to display the value 24 to Digital Lab Sim

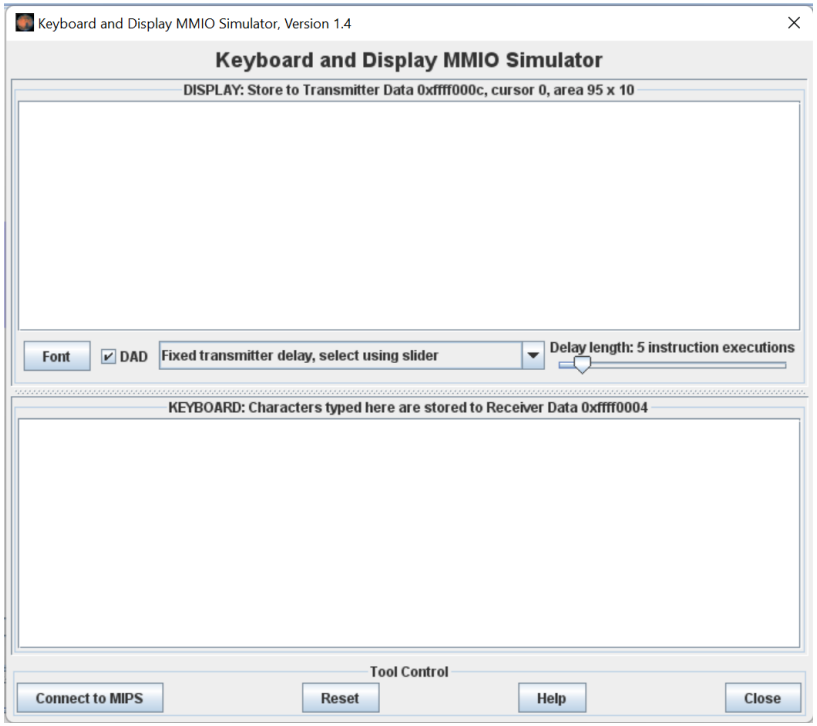
Communication of I/O Devices and Processor

- How I/O devices communicate with the processor
  - Polling
  - Interrupt driven I/O
  - Direct memory access
- Polling – the processor periodically checks the status of an I/O device to determine its need for service
  - Processor is totally in control – but does all the work
  - Can waste a lot of processor time due to speed differences

Example: polling the terminal

- Terminal:
  - Input: receiver control (0xffff0000) and data (0xffff0004)
  - Output: transmitter control (0xffff0008) and data (0xffff000c)





Example: reading 1 byte from terminal

- Polling for data, then read when data is available

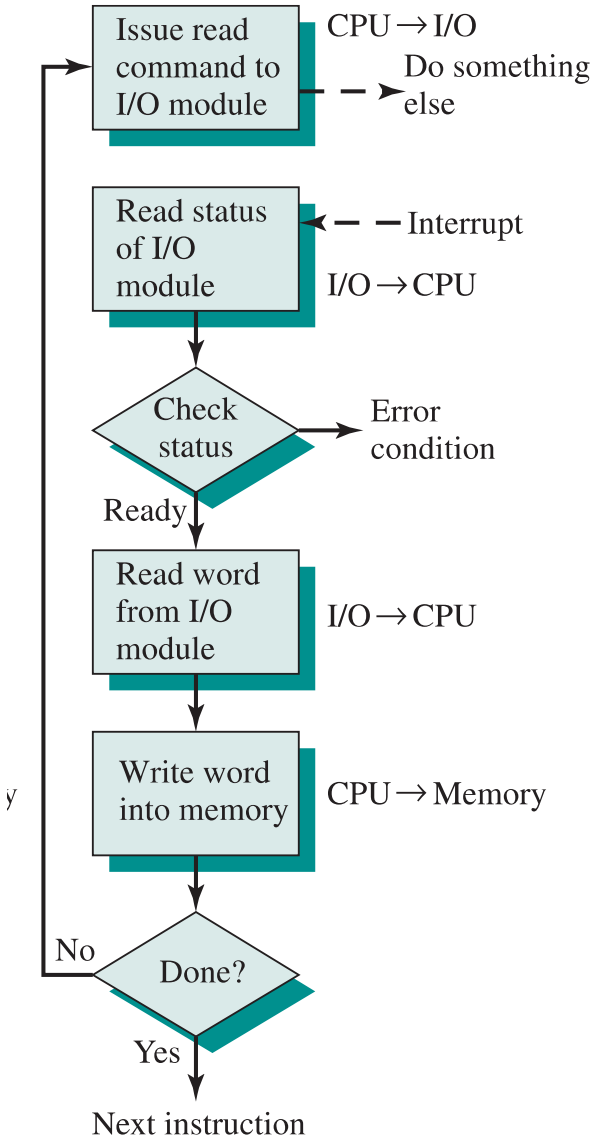
```
.eqv KEY_READY  0xFFFF0000
.eqv KEY_CODE   0xFFFF0004
.text
    li   $s0,  KEY_CODE
    li   $s1,  KEY_READY
WaitForKey:
    lw   $t1, 0($s1) # check data available
    beq  $t1, $zero, WaitForKey
    # if $t1 == 0 then Polling
ReadKey:
    lw   $t0, 0($s0)

    li $v0, 11
    move $a0, $t0
    syscall
```

Exercise

- Write a program to continuously read data from the terminal, encode the data by shifting it 3 position in the ASCII table, then write the encoded data to the terminal.
- Remember to check for terminal input and output ready before read/write.

Interrupt driven I/O

- The I/O device issues an interrupt to indicate that it needs attention.
- The processor detects and “serves” the interrupt by executing a handler (aka. Interrupt service routine).



Interrupt Driven I/O

- Advantages of using interrupts
  - Relieves the processor from having to continuously poll for an I/O event;
  - User program progress is only suspended during the actual transfer of I/O data to/from user memory space
- Disadvantage – special hardware is needed to
  - Indicate the I/O device causing the interrupt and to save the necessary information prior to servicing the interrupt and to resume normal processing after servicing the interrupt

RISC-V Interrupt

- Control and Status Registers (CSRs): indicate the state of the CPU and allow software to control the behavior of the CPU.

- ustatus (status register)
- ucause (interrupt cause)
- utvec (trap vector)
- uie (interrupt enable)
- uip (interrupt pending)
- uepc (exception program counter)

Registers | Floating Point | Control and Status

Name	Number	
ustatus	0	
fflags	1	
frm	2	
fcsr	3	
uie	4	
utvec	5	
uscratch	64	
uepc	65	
ucause	66	
utval	67	
uip	68	
cycle	3072	
time	3073	
instret	3074	
cycleh	3200	
timeh	3201	
instreth	3202	

Interrupt handling flow

- 1. Declare the interrupt handling routine (Interrupt Service Routine - ISR)
- 2. Load the **interrupt handling routine** address into the **utvec** register.
- 3. Depending on the program, set the interrupt source in the **uie** register.
- 4. Enable global interrupts, set the uie bit of the **ustatus** register.
- 5. (For RARS simulator) Set up the simulation tool to enable interrupts (Keypad, Timer Tool, ...)

Interrupt handling

- When an interrupt occurred: RISC-V jumps to interrupt service routine
- Inside ISR
  - Classify the source of interrupt, depending on the interrupt source, perform the corresponding processing.
  - Exit from ISR with instruction **`uret`** (exception return). This basically restores PC with value in EPC.

Example: detect a keypad button pressed (1/3)

```
.eqv IN_ADDRESS_HEXA_KEYBOARD       0xFFFF0012
.data
    message: .asciz "Someone's presed a button.\n"
# MAIN Procedure
.text
main:
    # Load the interrupt service routine address to the UTVEC register
    la      t0, handler
    csrrs   zero, utvec, t0

    # Set the UEIE (User External Interrupt Enable) bit in UIE register
    li      t1, 0x100
    csrrs   zero, uie, t1       # uie - ueie bit (bit 8)

    # Set the UIE (User Interrupt Enable) bit in USTATUS register
    csrrsi  zero, ustatus, 0x1  # ustatus - enable uie (bit 0)
    # Enable the interrupt of keypad of Digital Lab Sim
    li      t1, IN_ADDRESS_HEXA_KEYBOARD
    li      t3, 0x80  # bit 7 = 1 to enable interrupt
    sb      t3, 0(t1)
```

Example: detect a keypad button pressed (2/3)

```
    # -------------------------------------------------------
    # No-end loop, main program, to demo the effective of interrupt
    # -------------------------------------------------------
loop:
    nop
    # Delay 10ms
    li      a7, 32
    li      a0, 10
    ecall
    nop
    j       loop
end_main:
```

Example: detect a keypad button pressed (3/3)

```
# Interrupt service routine
handler:
    # ebreak # Can pause the execution to observe registers
    # Saves the context
    addi    sp, sp, -8
    sw      a0, 0(sp)
    sw      a7, 4(sp)

    # Handles the interrupt: Shows message in Run I/O
    li      a7, 4
    la      a0, message
    ecall

    # Restores the context
    lw      a7, 4(sp)
    lw      a0, 0(sp)
    addi    sp, sp, 8

    # Back to the main procedure
    uret
```

Direct Memory Access (DMA)

- For high-bandwidth devices (like disks) interrupt-driven I/O would consume a *lot* of processor cycles
- With DMA, the DMA controller has the ability to transfer large blocks of data directly to/from the memory without involving the processor
  1. The processor initiates the DMA transfer by supplying the I/O device address, the operation to be performed, the memory address destination/source, the number of bytes to transfer
  2. The DMA controller manages the entire transfer (possibly thousand of bytes in length), arbitrating for the bus
  3. When the DMA transfer is complete, the DMA controller interrupts the processor to let it know that the transfer is complete
- There may be multiple DMA devices in one system
  - Processor and DMA controllers contend for bus cycles and for memory

Summary

- Characteristics of I/O system and devices
- I/O performance measures
- I/O system organization
- Methods for I/O operation and control
  - Polling
  - Interrupt
  - DMA