McGill

Computer Organization

Input/Output

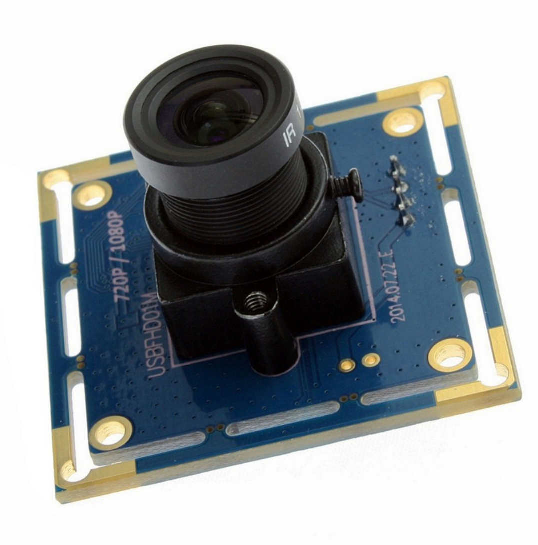
ECSE 324

Fall 2020

Prof. Christophe Dubach

Source material from Prof. Warren Gross. Some material from Hamacher, Vranesic, Zaky, and Manjikian, "Computer Organization and Embedded Systems, 6th ed", 2012, McGraw Hill and "Introduction to the ARM Processor using Altera Toolchain".

Input / Output



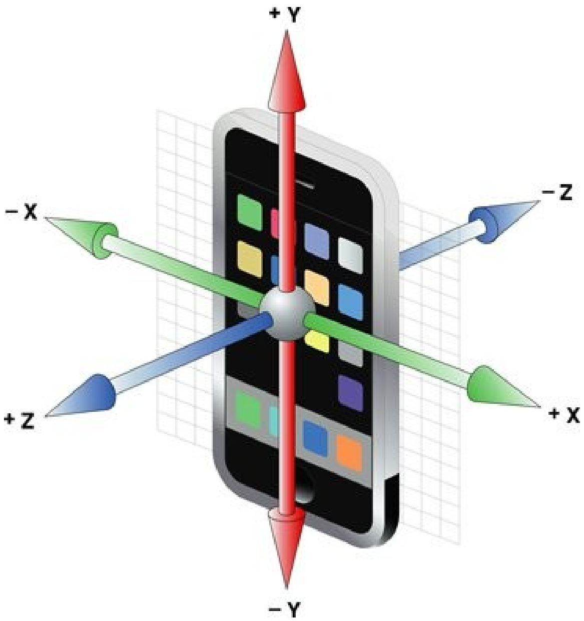
Input / Output

USB (universal serial bus)

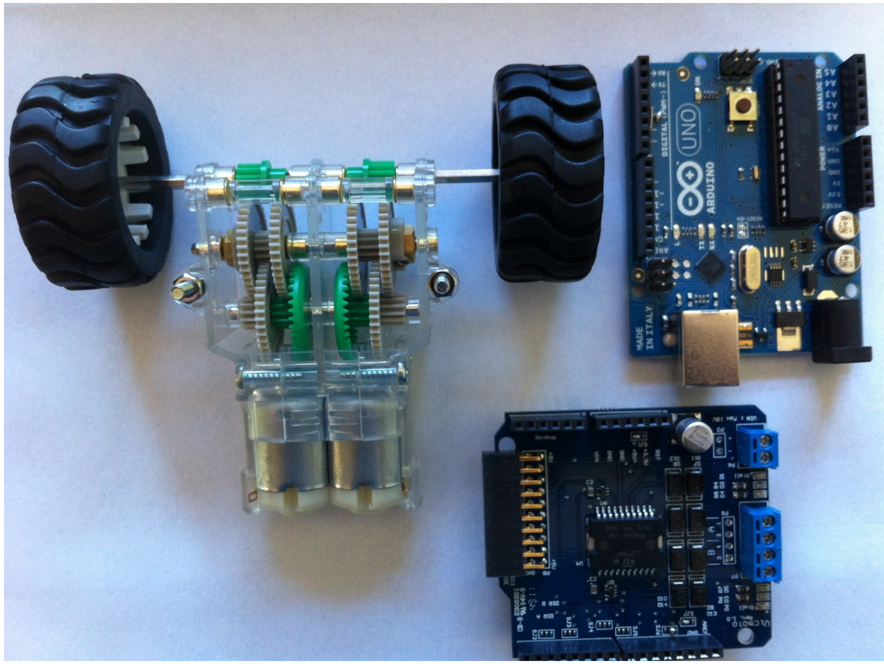
WiFi

Ethernet networking

Input / Output



sensors



actuators

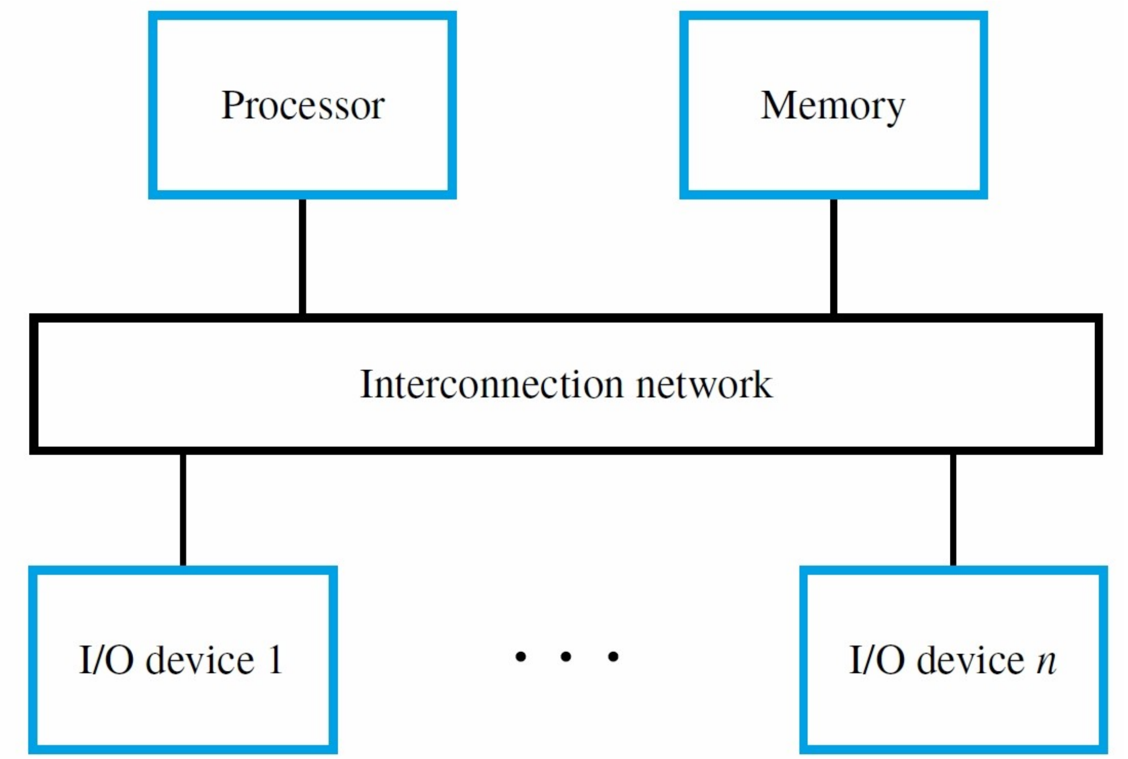
Software aspects of I/O: Memory Mapped Registers

Textbook 3.1, D.8.1

Accessing I/O Devices

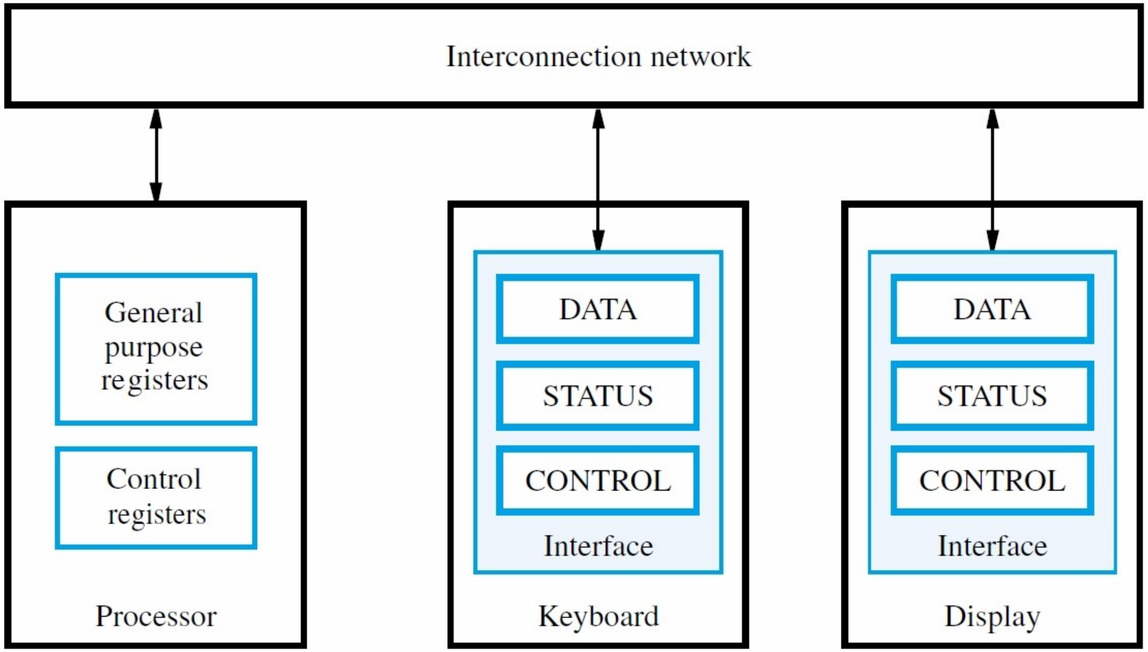
- Computer system components communicate through an interconnection network

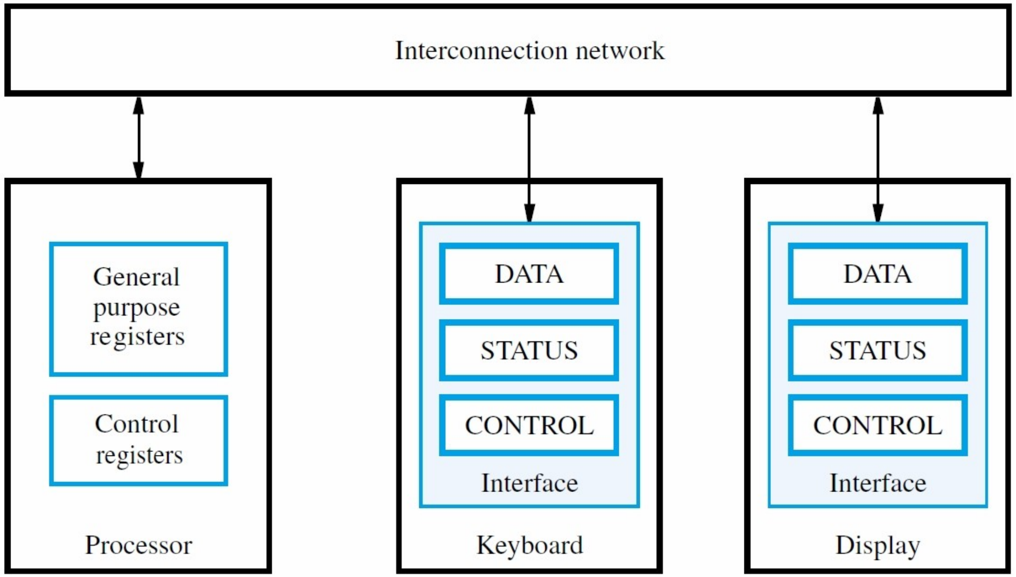
- From a programmers point of view, locations implemented as I/O registers within same address space



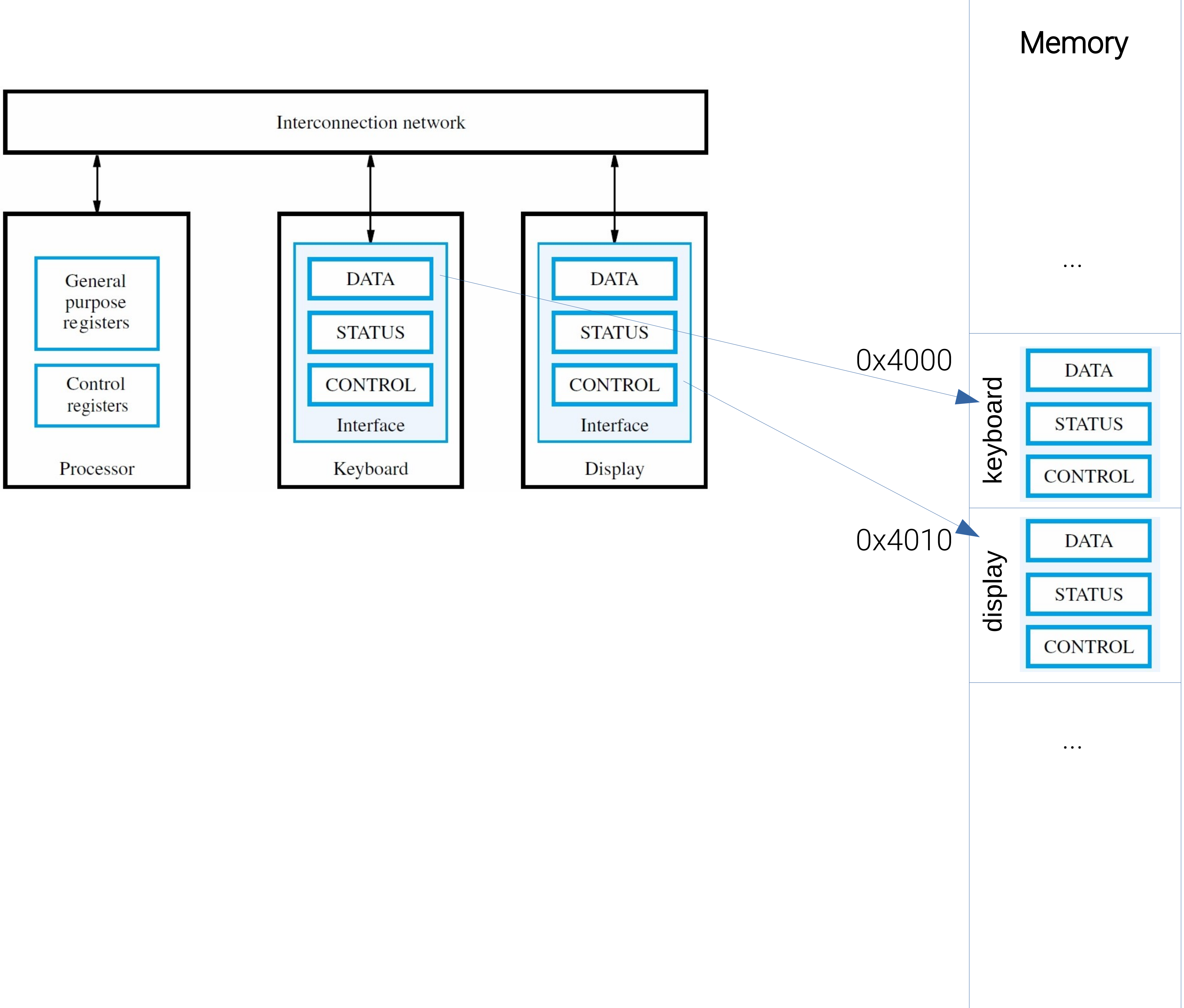
I/O Device Interface

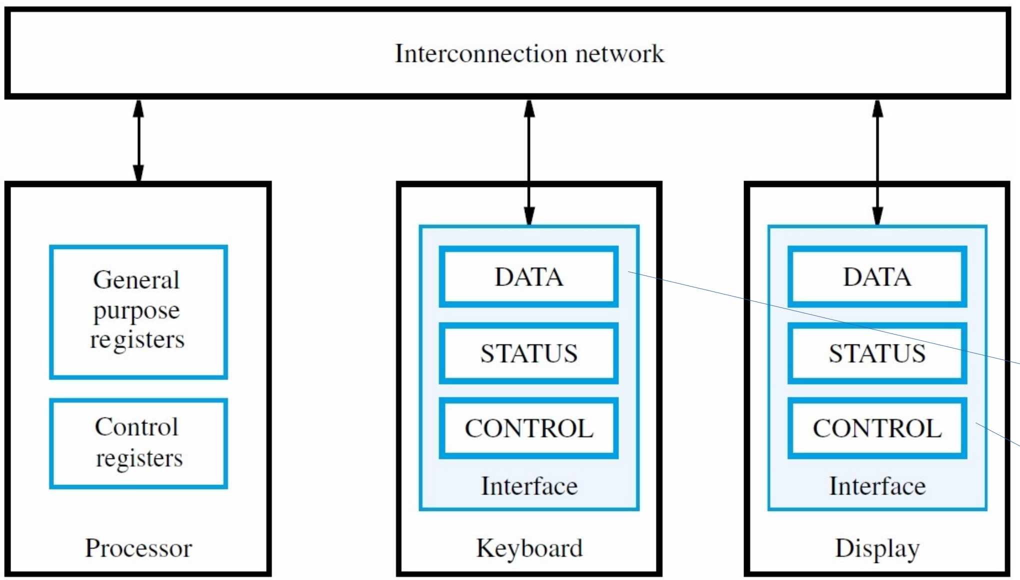
- An *I/O device interface* is a circuit between a device and the interconnection network
- Provides the means for data transfer and exchange of status and control information
- Includes *data*, *status*, and *control* registers accessible with load and store instructions
- Memory-mapped I/O enables software to view these registers as locations in memory

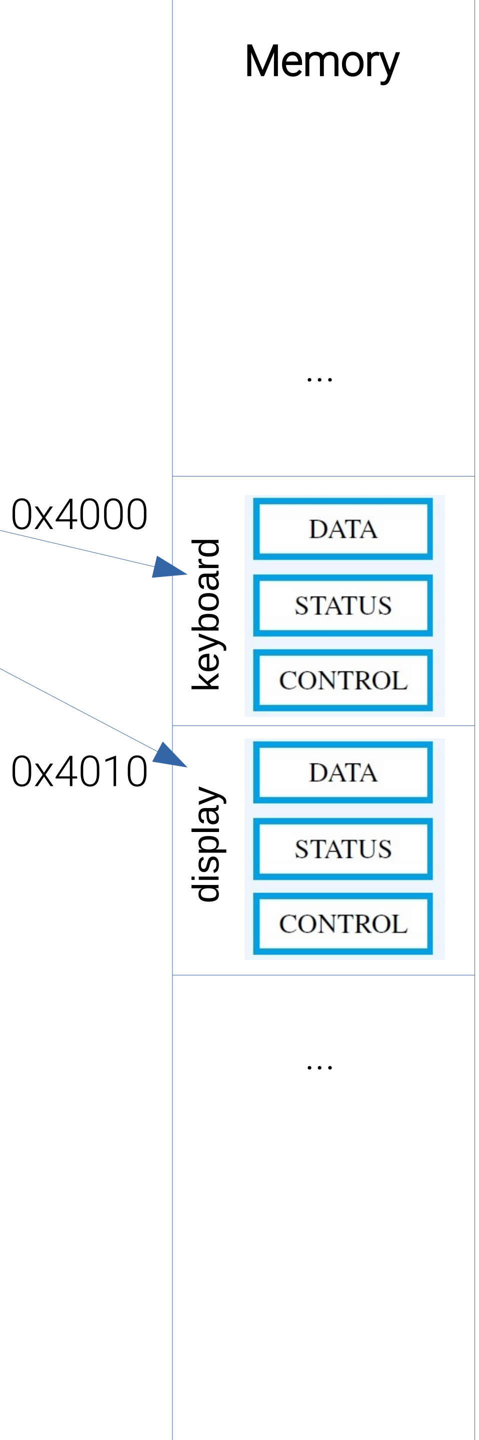



Interconnection network
General purpose registers
Control registers
Processor
DATA
STATUS
CONTROL
Interface
Keyboard
DATA
STATUS
CONTROL
Interface
Display

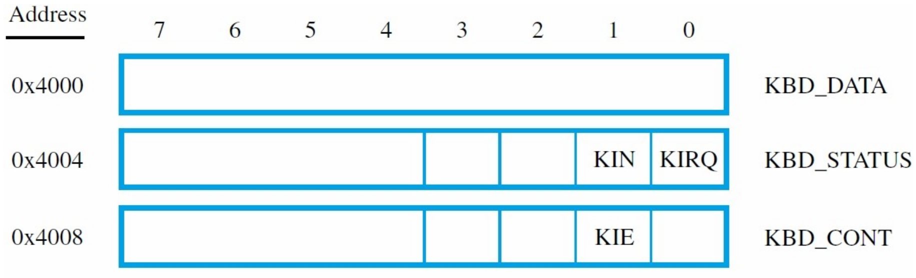
Memory


Memory
Interconnection network
General purpose registers
Control registers
Processor
DATA
STATUS
CONTROL
Interface
Keyboard
DATA
STATUS
CONTROL
Interface
Display
...
0x4000
keyboard
DATA
STATUS
CONTROL
0x4010
display
DATA
STATUS
CONTROL
...





- These I/O device registers are memory-mapped



Address	7	6	5	4	3	2	1	0	
0x4000									KBD_DATA
0x4004							KIN	KIRQ	KBD_STATUS
0x4008							KIE		KBD_CONT

(a) Keyboard interface

Memory-mapped I/O

Locations associated with I/O devices are accessed with Load and Store instructions

```
LDR R2, =0x4000    // R2 points to the data reg
LDR R1, [R2]       // read from the data reg
STR R1, [R2]       // write to the data reg
```

I/O synchronization

- E.g. Read keyboard characters, store in memory, and display on screen
  - A keyboard's data input rate (keyboard to processor) is likely to be only a few characters per second – limited by user's typing speed.
  - The rate of character output (processor to display) is likely to be much faster - say thousands of characters per second.
  - The processor can execute billions of instructions per second – much faster than the display can accept data!
- Need a way to synchronize the timing of an I/O device with the processor.
  - How do you know at what time an input device has data ready for the processor to read?
  - How do you know at what time the output device is ready to receive data written by the processor?

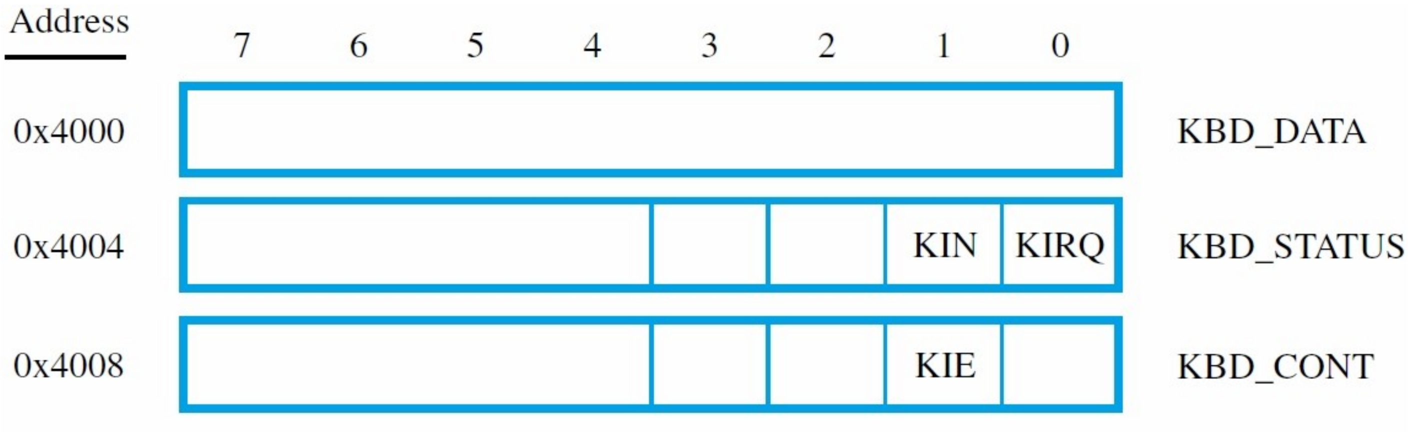
Software aspects of I/O: Polling

Textbook 3.1, D.8.1

Programmed-controlled I/O

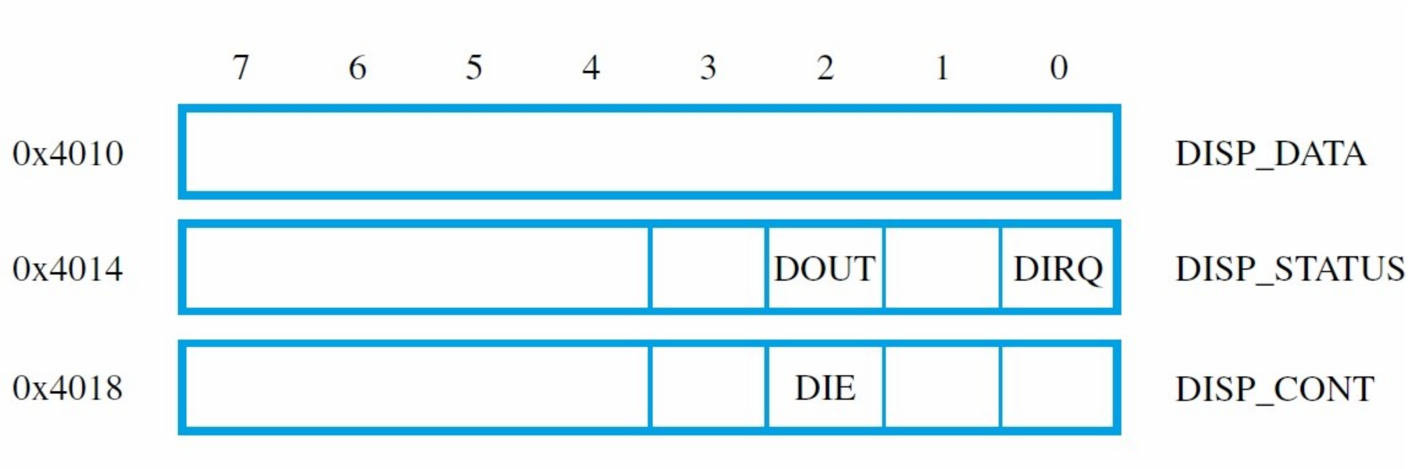
- Assume that the I/O devices have a way to send a `ready` signal to the processor
  - For keyboard, indicates character can be read so processor uses a load to access data register
  - For display, indicates character can be sent so processor uses a store to access data register
- The `ready` signal in each case is a status flag in status register that is *polled* by the processor.

Polled I/O: reading



- Assume a device with 8-bit I/O registers.
- For example, keyboard has `KIN` status flag in bit $\mathtt{b}_1$ of KBD_STATUS register at address `0x4004`
- Processor polls `KBD_STATUS` register, checking whether `KIN` flag is 0 or 1
- If `KIN` is 1, processor reads `KBD_DATA` register

Polled I/O: writing



	7	6	5	4	3	2	1	0	
0x4010									DISP_DATA
0x4014						DOUT		DIRQ	DISP_STATUS
0x4018						DIE			DISP_CONT

- For example, display has `DOUT` status flag in bit b_2 of DISP_STATUS register at address `0x4014`

- Processor polls `DISP_STATUS` register, checking whether `DOUT` flag is 0 or 1

- If `DOUT` is 1, processor writes `DISP_DATA` register

- You have to poll a device's status register for each time you read or write it's data register

Wait Loop for Polling I/O Status

- Program-controlled I/O implemented with a wait loop for polling keyboard status register:
- Keyboard circuit places character in `KBD_DATA` and sets `KIN` flag in `KBD_STATUS`
- Circuit clears `KIN` flag when `KBD_DATA` is read
- Assume that the address for `KBD_DATA (0x4000)` has been loaded into `R1`.

```
READWAIT: LDRB   R3, [R1, #4]  // read byte from KBD_STATUS
          TST    R3, #2        // check the value of KIN
          BEQ    READWAIT      // branch when KIN=0
          LDRB   R3, [R1]      // read from KBD_DATA
```

Wait Loop for Polling I/O Status

- Display circuit sets `DOUT` flag in `DISP_STATUS` after previous character has been displayed
- Circuit automatically clears `DOUT` flag when `DISP_DATA` register is written
- Assume that the address for `DISP_DATA` has been loaded into `R2`.

```
WRITEWAIT:    LDRB   R4, [R2, #4]  // read byte from DISP_STATUS
              TST    R3, #4        // check the value of DOUT
              BEQ    WRITEWAIT     // branch when DOUT=0
              STRB   R3, [R2]      // write to DISP_DATA
```

Example

- Consider complete program that use polling to read, store, and display a line of characters (“echo” to the display).
- Program finishes when carriage return (`CR`) character is entered on keyboard
- Assume `R0` points to the first byte of the memory area where the line is to be stored.

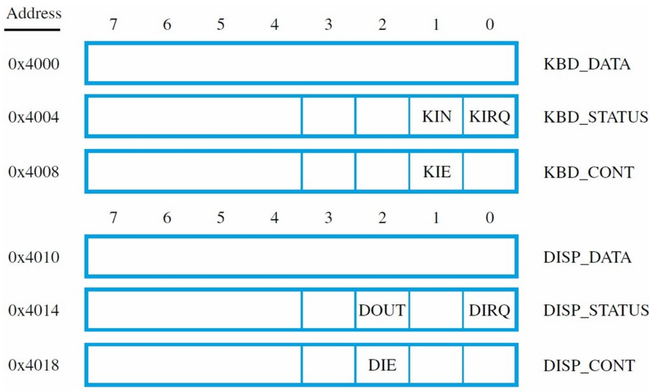
```
        LDR   R1, =0x4000    // KBD
        LDR   R2, =0x4004    // DISP

READ:   LDRB  R3, [R1, #4]   // load KBD_STATUS byte and
        TST   R3, #2         // wait for character
        BEQ   READ

        LDRB  R3, [R1]       // read the character and
        STRB  R3, [R0], #1   // store it in memory

ECHO:   LDRB  R4, [R2, #4]   // load DISP_STATUS byte and
        TST   R4, #4         // wait for display
        BEQ   ECHO           // to be ready

        STRB  R3, [R2]       // send character to display
        TEQ   R3, #CR        // if not carriage return
        BNE   READ           // read more characters
```



Software aspects of I/O: Interrupts

Textbook 3.2, D.7, D.8.2

Example

Waiting for a pizza when you are studying for exams

polling

interrupt

Interrupts

- Polling with a wait loop has a big drawback: processor is kept busy.
- With long delay before I/O device is ready, cannot perform other useful computation
- Instead of using a wait loop, an alternative is to let the I/O device alert the processor when it is ready
- Hardware sends an *interrupt-request signal* to the processor at the appropriate time
- Meanwhile, processor performs useful tasks

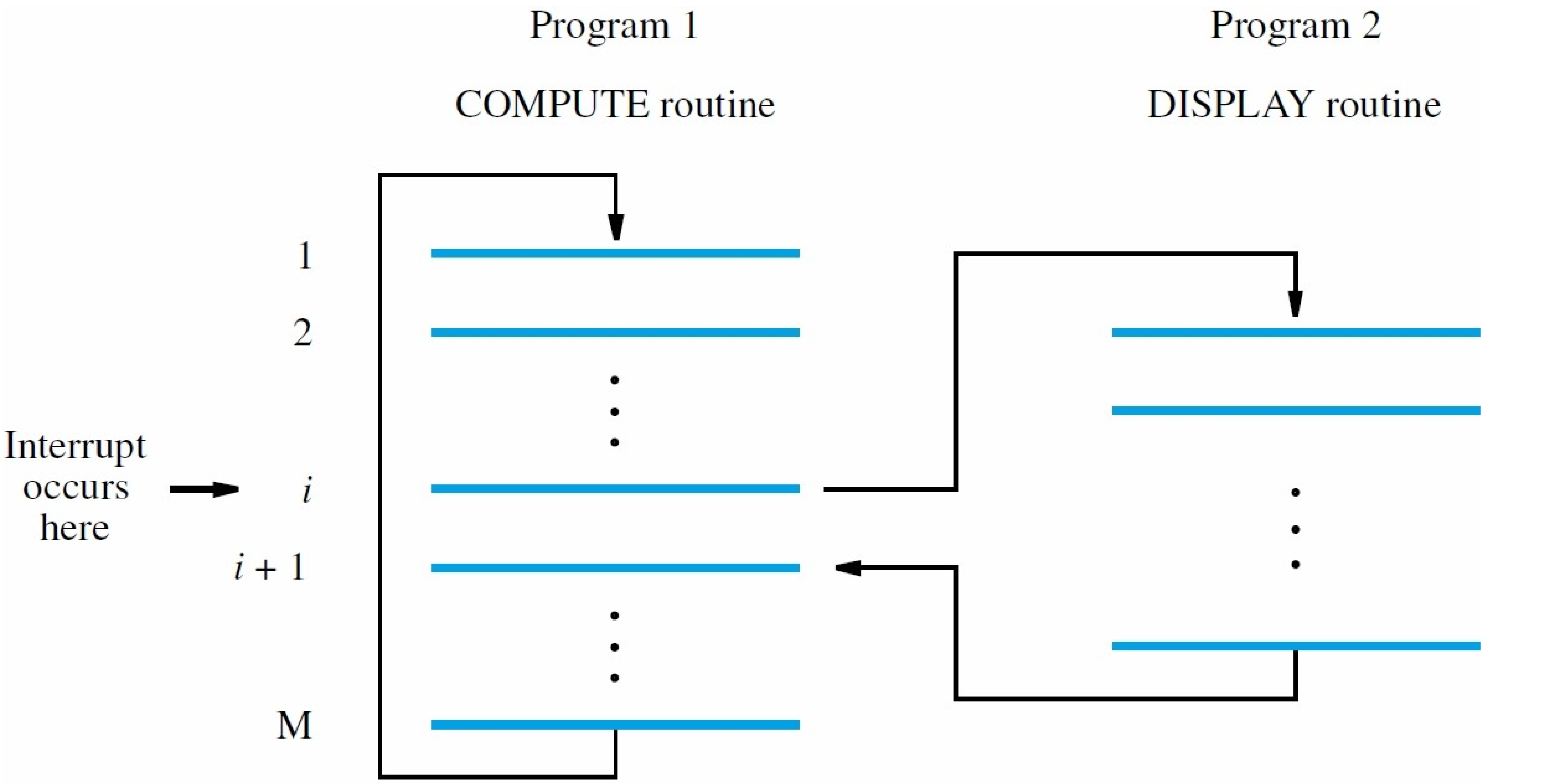
Example of Using Interrupts

- Consider a task with extensive computation and periodic display of current results (every 10s)

- A timer circuit can be used for the desired 10s interval - it can send the processor a signal every 10s

- Polling would spend each 10s interval waiting for the signal, with no time doing computation.

- The timer can raise an interrupt-request signal to processor every 10s



- Processor completes instruction `i`, and then suspends `COMPUTE` execution to execute `DISPLAY`, then returns to execute instruction `i+ 1`

- `DISPLAY` is short; time is mostly spent in `COMPUTE`

Interrupt-Service Routine

- `DISPLAY` is an *interrupt-service routine (ISR)*.

- Unlike a subroutine, it can can be *executed at any time*, not in response to a call.

- For example, assume interrupt signal asserted when processor is executing instruction `i`

- Instruction completes, then `PC` saved to temporary location before executing `DISPLAY` (`PC` set to the first instruction in the ISR)

- *Return-from-interrupt* instruction in `DISPLAY` restores `PC` with address of instruction `i + 1`

Issues for handling of interrupts

- Must save return address, processor registers and status registers since they could be changed by the ISR

- After return-from-interrupt, the saved information must be restored so that the original program can continue execution without being affected by the ISR.

- Saving/restoring of general-purpose registers can be automatic or program-controlled – usually the minimum is saved (PC and status register) to reduce *interrupt latency*, the time before the interrupt is serviced after it is raised.

- ISR is responsible for saving any other registers.

Acknowledging the interrupt

- *Interrupt-acknowledge signal* from processor tells device that interrupt has been recognized
- In response, device removes interrupt request
- Acknowledgment can be done by accessing status or data register in device interface

Enabling and Disabling Interrupts

- We may only want to respond to the timer interrupt during the compute routine, but not at other times.

- Between the time the interrupt is requested and is acknowledged, we only want the ISR to be called once.

→ we need a way to disable interrupts.

    - A bit in the processor status register can globally disable interrupts (i.e. ignore any interrupt-request signals from I/O devices).
    - A bit in an I/O device's control register can disable interrupts from that device.

Event Sequence for an Interrupt

1. Processor status register has IE bit
2. Program sets IE to 1 to enable interrupts
3. When an interrupt is recognized, processor saves program counter and status register
4. IE bit cleared to 0 so that same or other signal does not cause further interrupts
5. After acknowledging and servicing interrupt, restore saved state, which sets IE to 1 again

Handling Multiple Devices

What if more than one device initiates an interrupt?

Q1: How does the processor know which device is requesting an interrupt?

A1: poll the IRQ bit in each device's status register

`IRQ`: interrupt request

Handling Multiple Devices

What if more than one device initiates an interrupt?

Q2: How is the starting address for the correct ISR obtained?

A2: Call device-specific routine for first set IRQ bit that is encountered. Service the interrupt and then the next interrupt can be serviced.

Disadvantage: time is spent polling the IRQ bits of devices that are not requesting any service.

To reduce interrupt latency, use *vectored interrupts*.

Vectored Interrupts

- A requesting device identifies itself directly with a device-specific signal or a binary ID code sent to the processor
- The *interrupt-vector table stores the address of (or a branch instruction to) the corresponding ISR.*
- The interrupt vector table is located at fixed address, typically in the lowest memory addresses (e.g. first 128 bytes of memory may be reserved for 32 interrupt vectors)
- ISRs can be located anywhere in memory

Nested interrupts

Q3: Should a device be allowed to interrupt the processor while another interrupt is being serviced?

A3: Some devices need to be serviced quickly, even if it means interrupting a currently executing ISR!

- Assign a priority to each I/O device
- Only accept an interrupt from a device with higher priority when servicing a device with lower priority
- Interrupts from lower-priority devices are ignored
- An ISR should save the PC and SR on the stack and acknowledge the current interrupt before enabling nesting by enabling interrupts.

Handling Multiple Devices

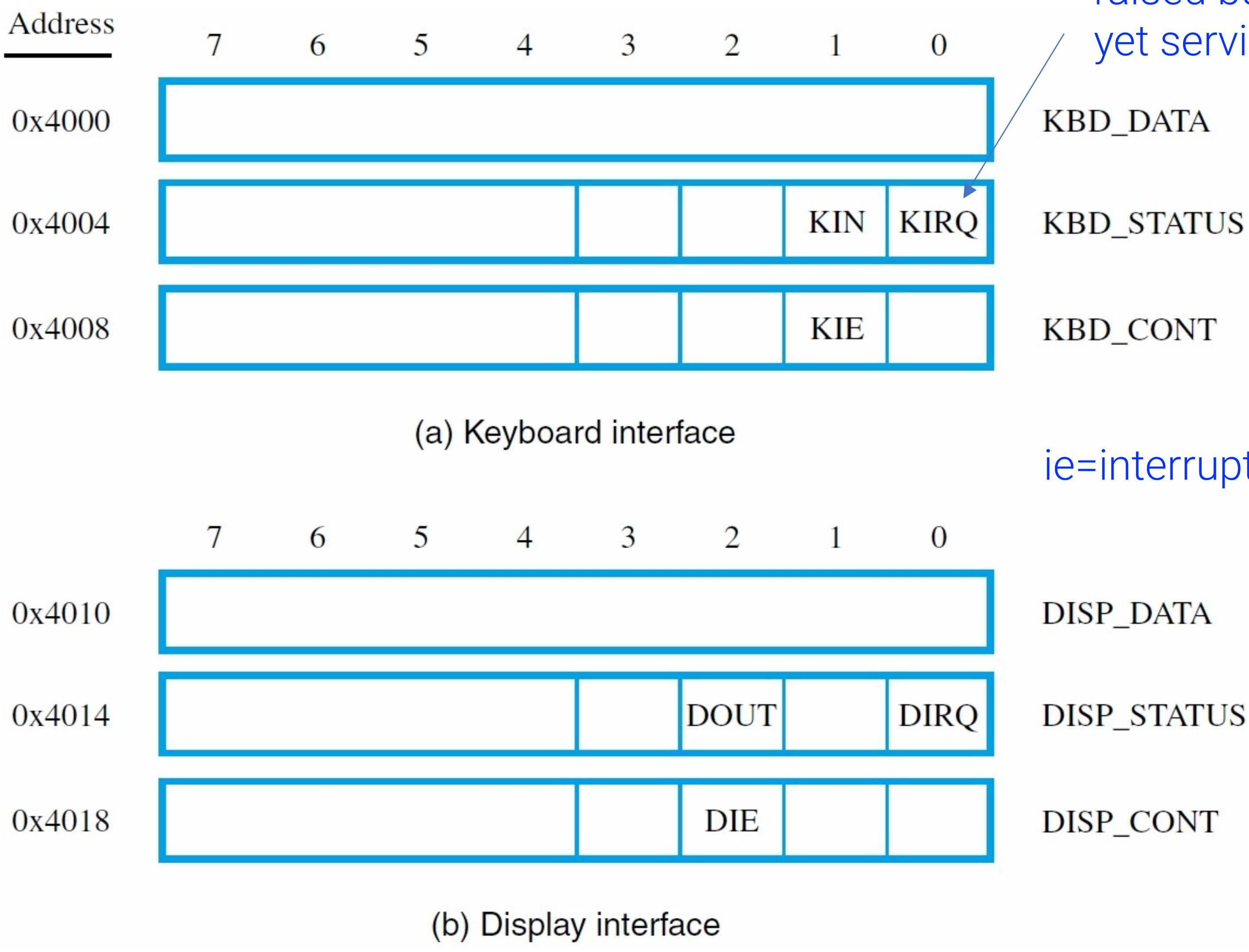
Q4: How are two simultaneous requests handled?

A4: A way to resolve the conflict (arbitration) is required

- When polling I/O status registers, the service order is determined by polling order.
- Vectored interrupts require hardware arbitration based on priority and fairness
- Hardware must select only one device to provide index to the vector table.

Device registers

‘1’ if interrupt raised but not yet serviced



Address	7	6	5	4	3	2	1	0	
0x4000									KBD_DATA
0x4004							KIN	KIRQ	KBD_STATUS
0x4008							KIE		KBD_CONT

(a) Keyboard interface

ie=interrupt enable

	7	6	5	4	3	2	1	0	
0x4010									DISP_DATA
0x4014						DOUT		DIRQ	DISP_STATUS
0x4018						DIE			DISP_CONT

(b) Display interface

Exceptions

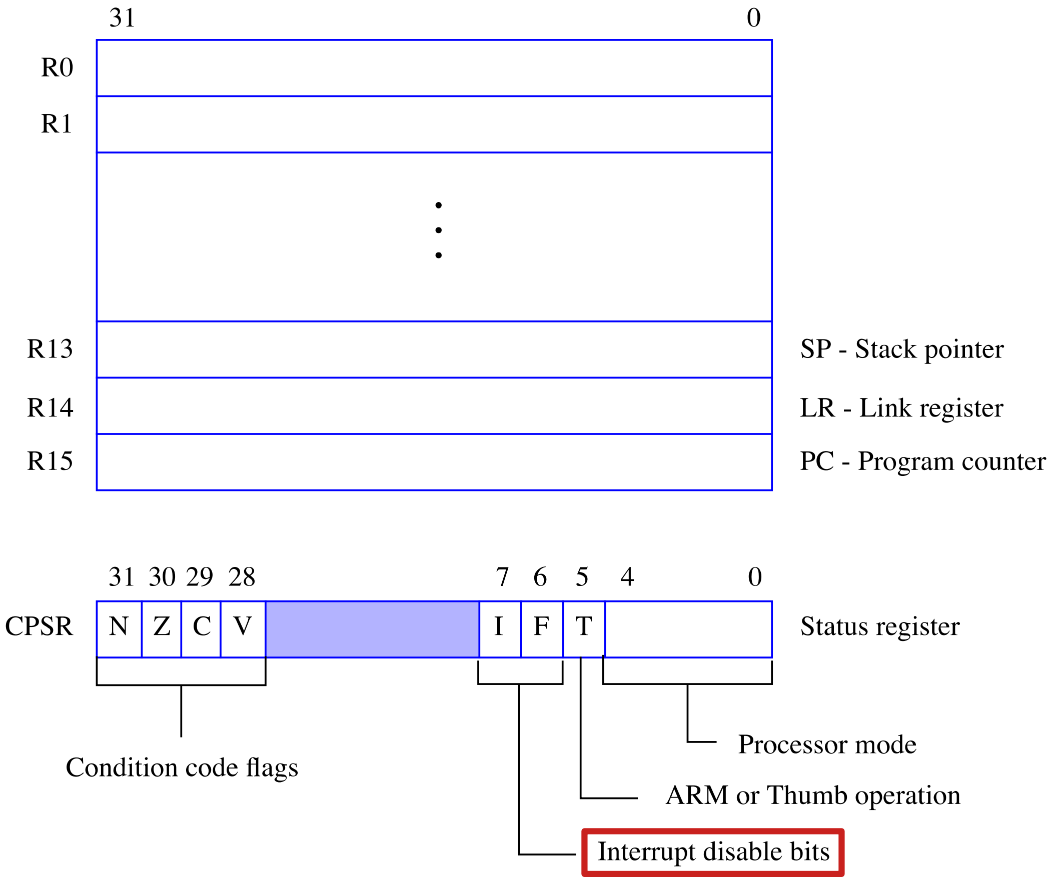
- An exception is any interruption of execution, not just for I/O
  - Recovery from errors: detect division by zero, or instruction with an invalid OP code
  - Debugging: use of trace mode & breakpoints
  - Operating system uses software interrupts

Recovery from Errors

- After saving state, service routine is executed
- Routine can attempt to recover (if possible) or inform user, perhaps ending execution
- With I/O interrupt, instruction being executed at the time of request is allowed to complete
- If the instruction is the *cause* of the exception, service routine must be executed immediately
- Thus, return address may need adjustment

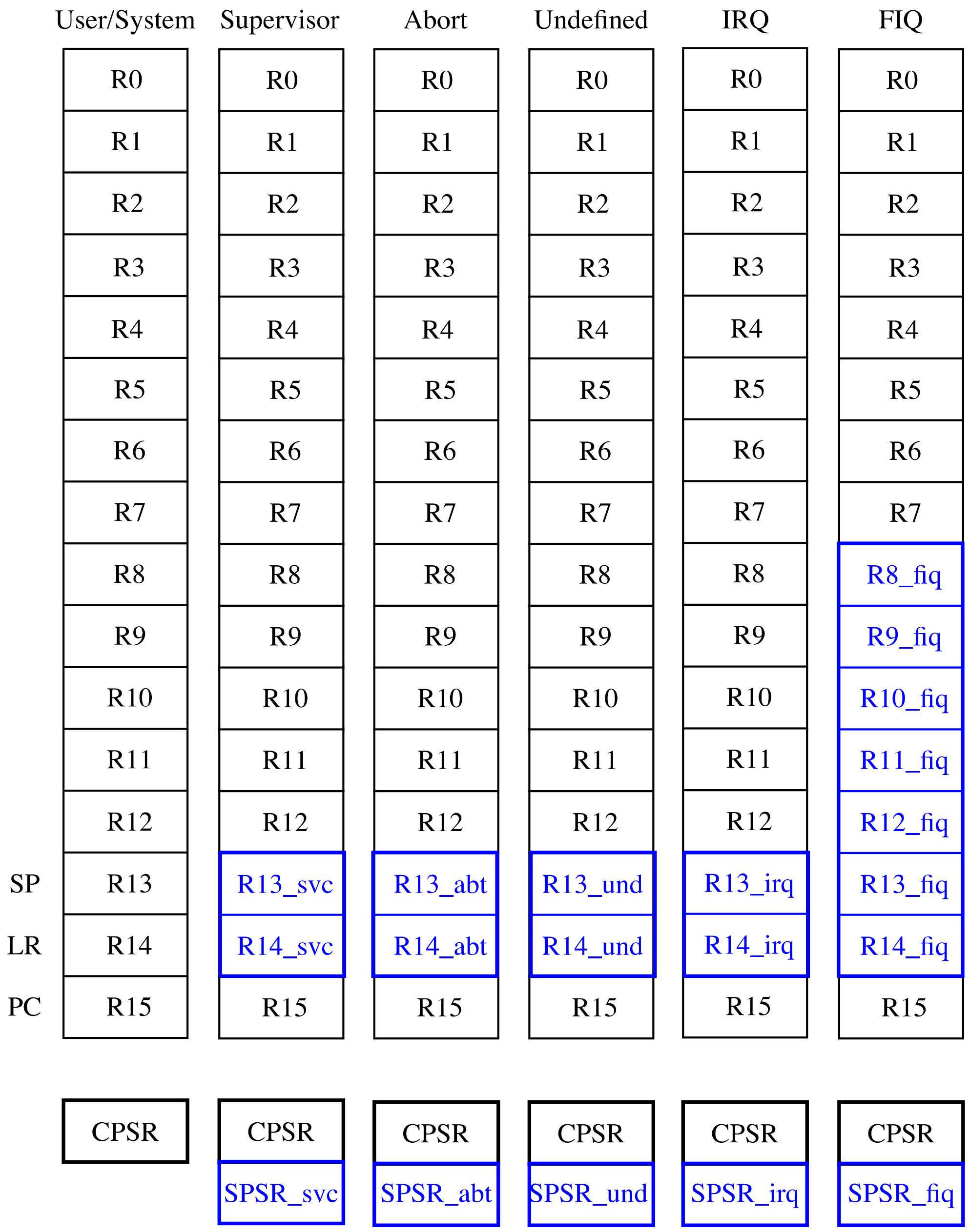
ARM Processor Modes

- The ARM processor has 7 *operating modes* that determine what system resources a program has access to.
- When an interrupt is received, the processor switches into one of two modes:
  - IRQ mode – entered when a normal interrupt is received
  - FIQ mode – entered in response to a *fast interrupt* request



Banked registers

- Some modes have an extra set of shadow registers that are used instead of the usual register when in that mode.
- E.g. in `IRQ` mode, access to "`R13`" are to "`R13_irq`" instead of the real `R13`
- This avoids having to save registers – fast
- `FIQ` maintains a bank of shadow registers for `R8-R12` as well – no need to save on stack.



	User/System	Supervisor	Abort	Undefined	IRQ	FIQ
	R0	R0	R0	R0	R0	R0
	R1	R1	R1	R1	R1	R1
	R2	R2	R2	R2	R2	R2
	R3	R3	R3	R3	R3	R3
	R4	R4	R4	R4	R4	R4
	R5	R5	R5	R5	R5	R5
	R6	R6	R6	R6	R6	R6
	R7	R7	R7	R7	R7	R7
	R8	R8	R8	R8	R8	R8_fiq
	R9	R9	R9	R9	R9	R9_fiq
	R10	R10	R10	R10	R10	R10_fiq
	R11	R11	R11	R11	R11	R11_fiq
	R12	R12	R12	R12	R12	R12_fiq
SP	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
LR	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	R15	R15	R15	R15	R15	R15
	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

SPSR = Saved Program Status Register

ARM return-from-interrupt

Recall that in PC-relative addressing we had to consider that the PC is incremented by 8 because the processor prefetches the next instruction.

```
i   (currently executing instruction)
i+1
I+2     ←  PC
```

To return from the interrupt, fetch instruction `i+1` by decrementing the address stored in the `LR` by 4

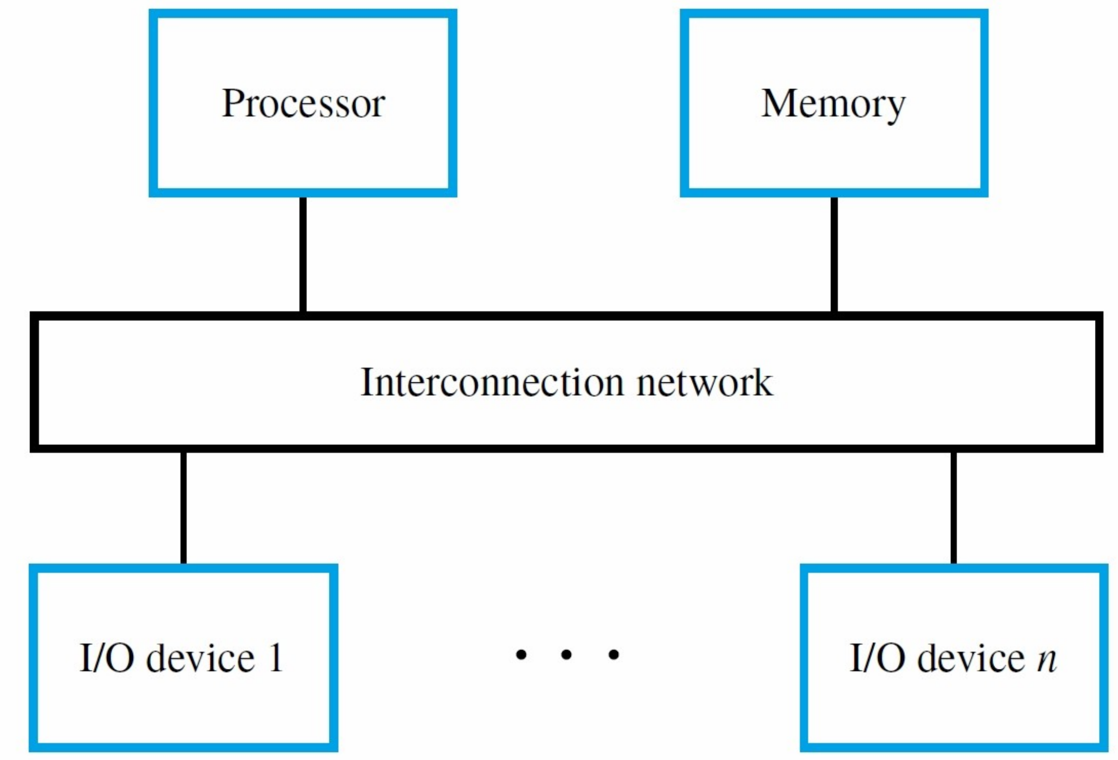
```
SUBS   PC, LR, #4
```

Hardware aspects of I/O: Bus protocols

Textbook 7.1-7.3

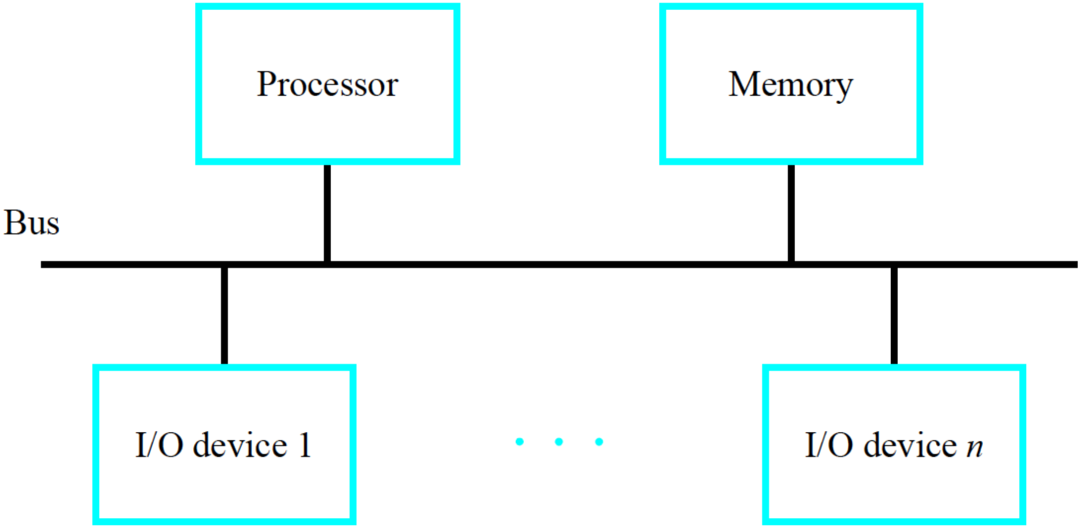
Interconnection networks

- An *interconnection network* is used to transfer data among the processor, memory, and I/O devices
- A commonly-used interconnection network is called a *bus*



A Single-Bus System

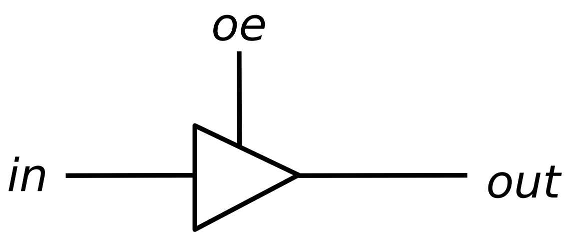
- A bus is a set of shared wires
- Only one pair source/destination units can use the bus to transfer data at any one time
- Hardware manages access to the bus to enforce this constraint



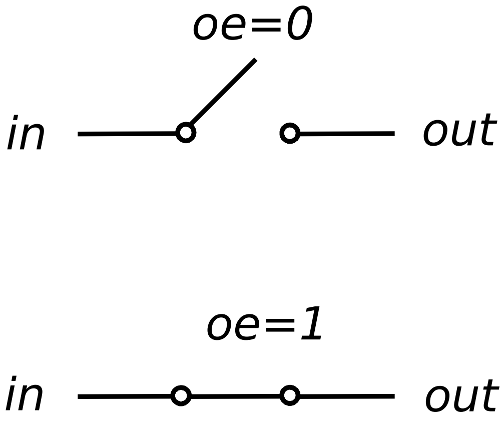
Tri-State Buffers

- When the control signal "output enable" (oe) is low the buffer is completely disconnected from the output
  - When oe is high, the buffers drives in onto out
  - The disconnected state "Z" is "high impedance"

oe	*in*	*out*
0	0	Z
0	1	Z
1	0	0
1	1	1

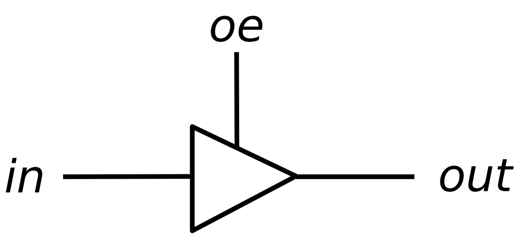


(a) A tri-state buffer

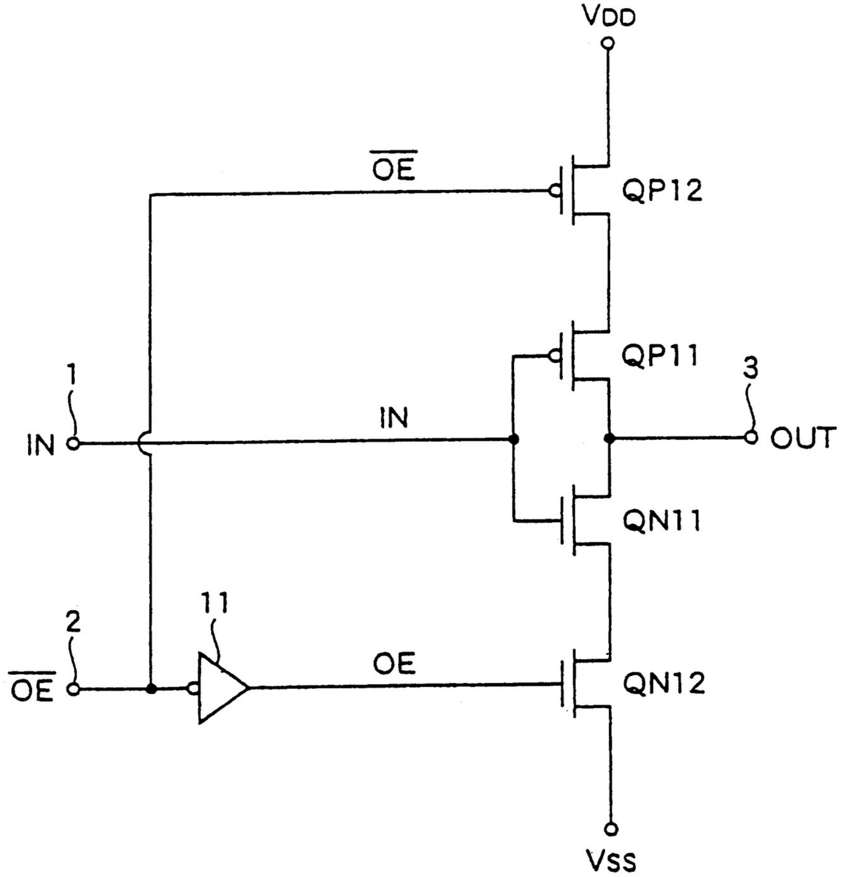


(b) Equivalent circuit

Tri-State Buffer C-MOS Implementation

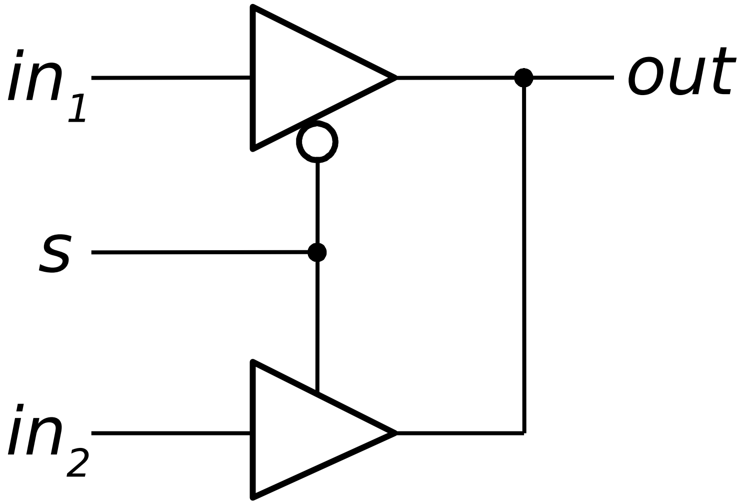


V_{DD}= supply Voltage at Drain (5V)

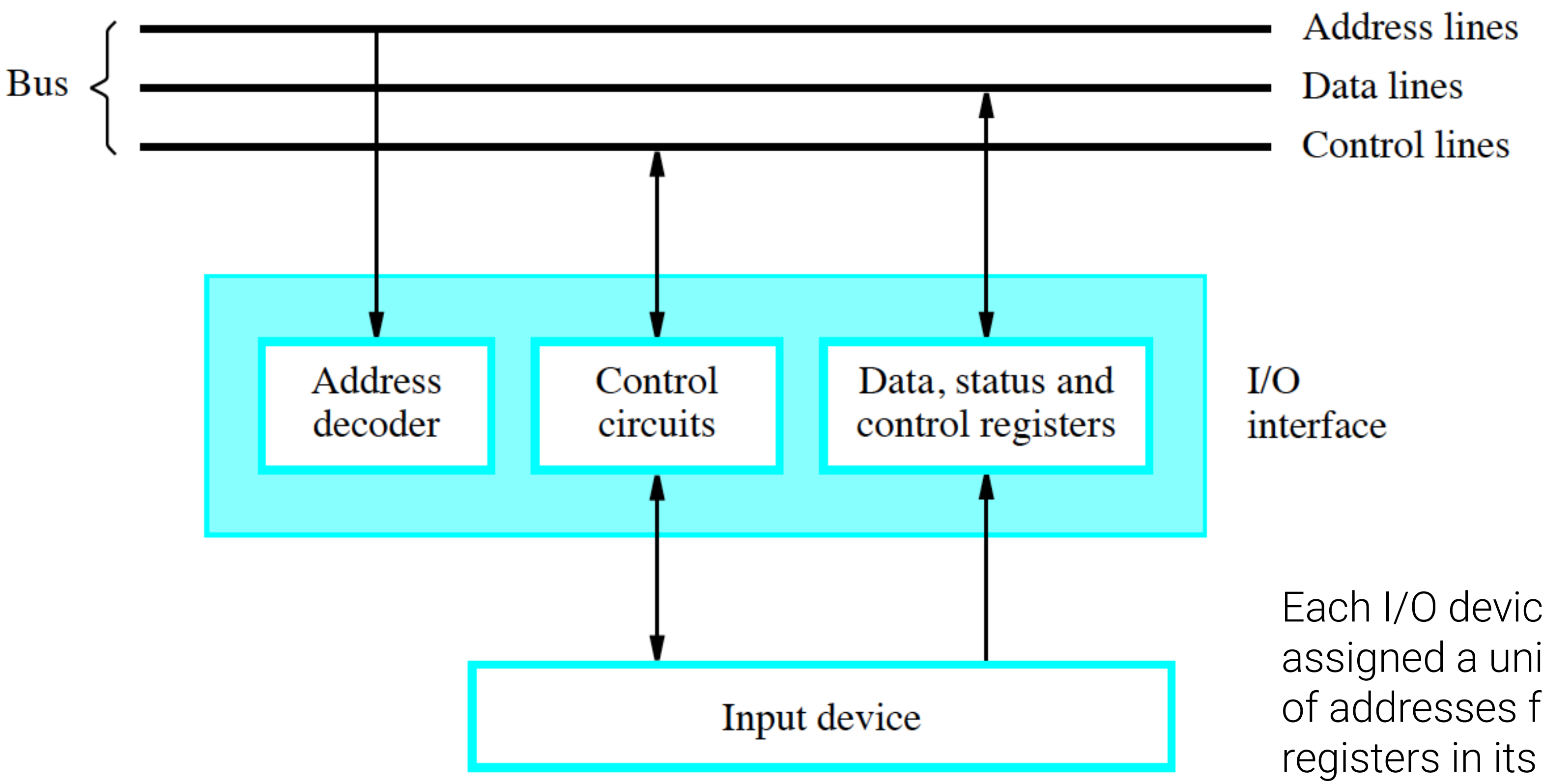


V_{SS}= supply Voltage at Source (0V)

Tri-State Buffers: select between two inputs



I/O interface for an input device



Each I/O device is assigned a unique set of addresses for the registers in its interface

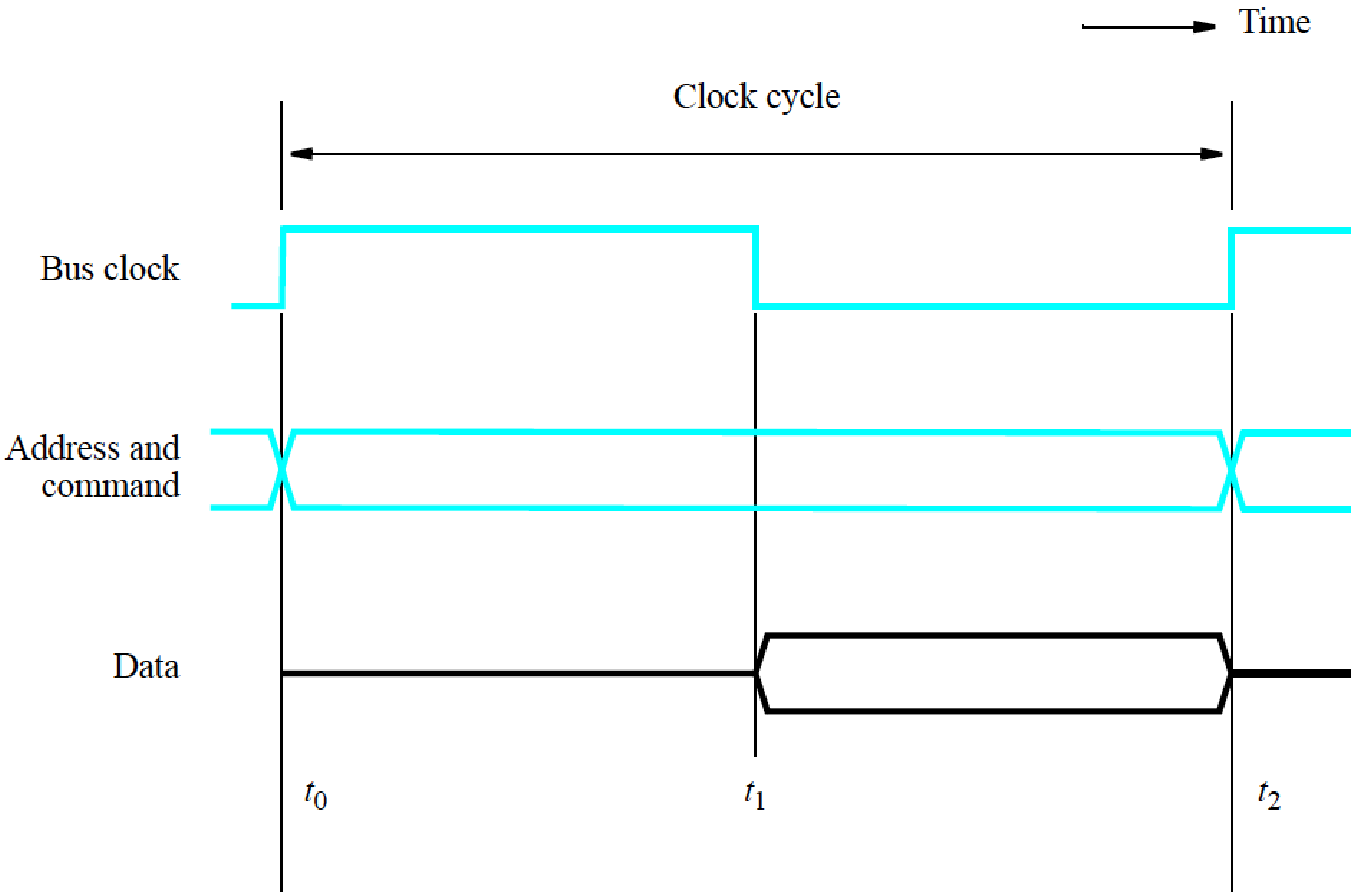
Bus protocols

- A bus protocol is a set of rules that govern when a device may place information on the bus, when it may load data on the bus into one of its registers, etc…
- Control signals indicate what and when actions are to be taken.
- $R/\overline{W}$ control line specifies whether a read or write is to be performed (read when 1, write when 0).
  - Data size parameter (byte, halfword, word) can be indicated by other control lines.
- One of the two devices controls the transfer initiating the read or write commands (*master*) and the other is *slave*. Usually, but not always, the processor is the master.
- Other control lines convey timing information. Two approaches to timing of bus transfers:
  - *Synchronous :* all devices derive timing information from a *bus clock*
  - *Asynchronous*

Input (read) transfer timing on a synchronous bus

$t_1 - t_0$ > max. propagation delay of bus + time for slave to decode address and control signals

$t_2 - t_1$ > max propagation delay of bus + setup time of master's register



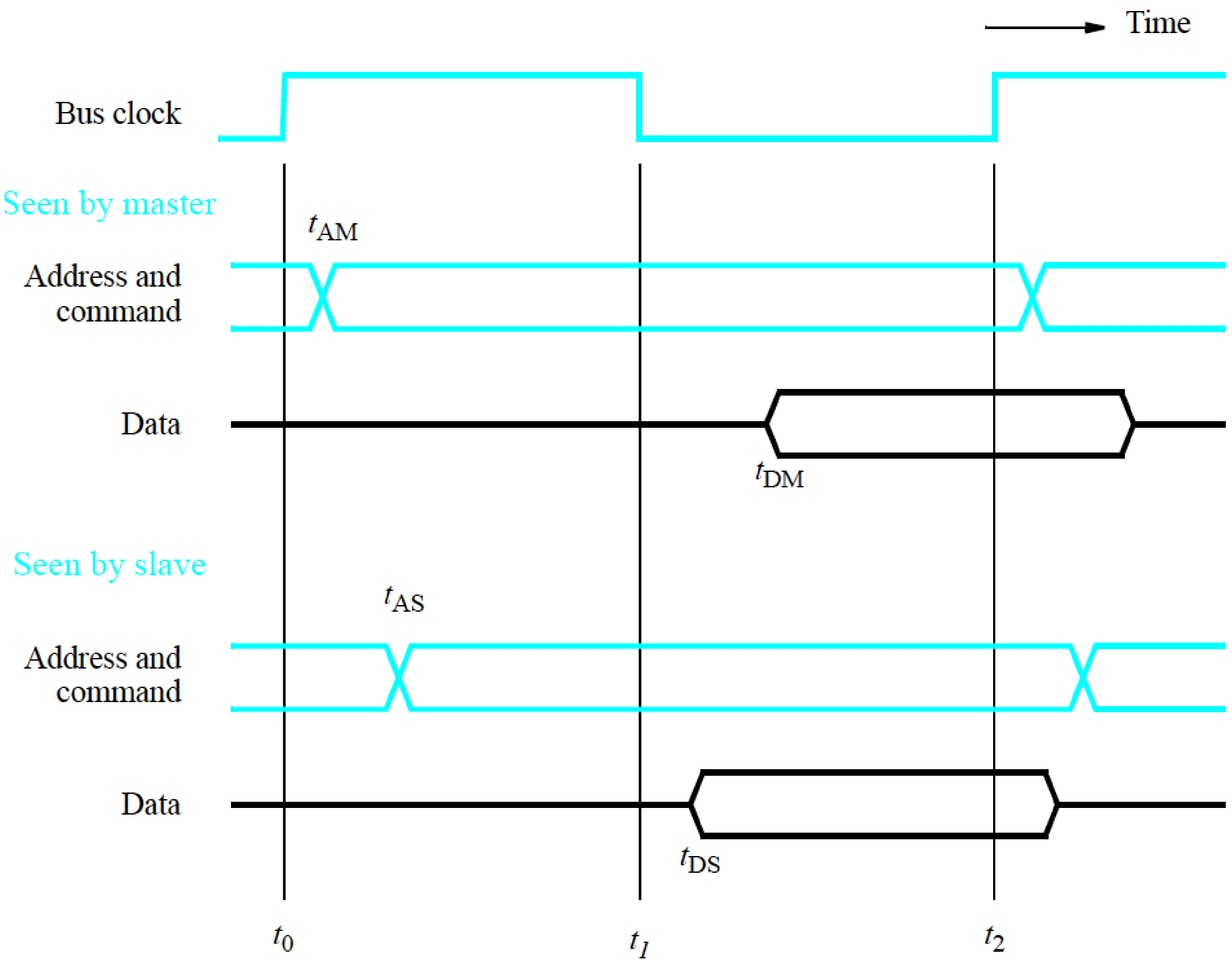
Input transfer timing on a synchronous bus

- Write is similar: master puts data on data lines at t_0. At t_2 the addressed device loads the data into its data register.

- Signals propagate to different devices at different times depending on their location on the bus.

- Assume that the bus clock is seen at all devices at the same time.
    - System designers spend a lot of time making sure this is true.

A detailed timing diagram for the input transfer



A detailed timing diagram for the input transfer

- Not all devices operate at the same speed.

- $t_2 - t_0$ must be chosen to accommodate the longest delays on the bus and the slowest device interface.

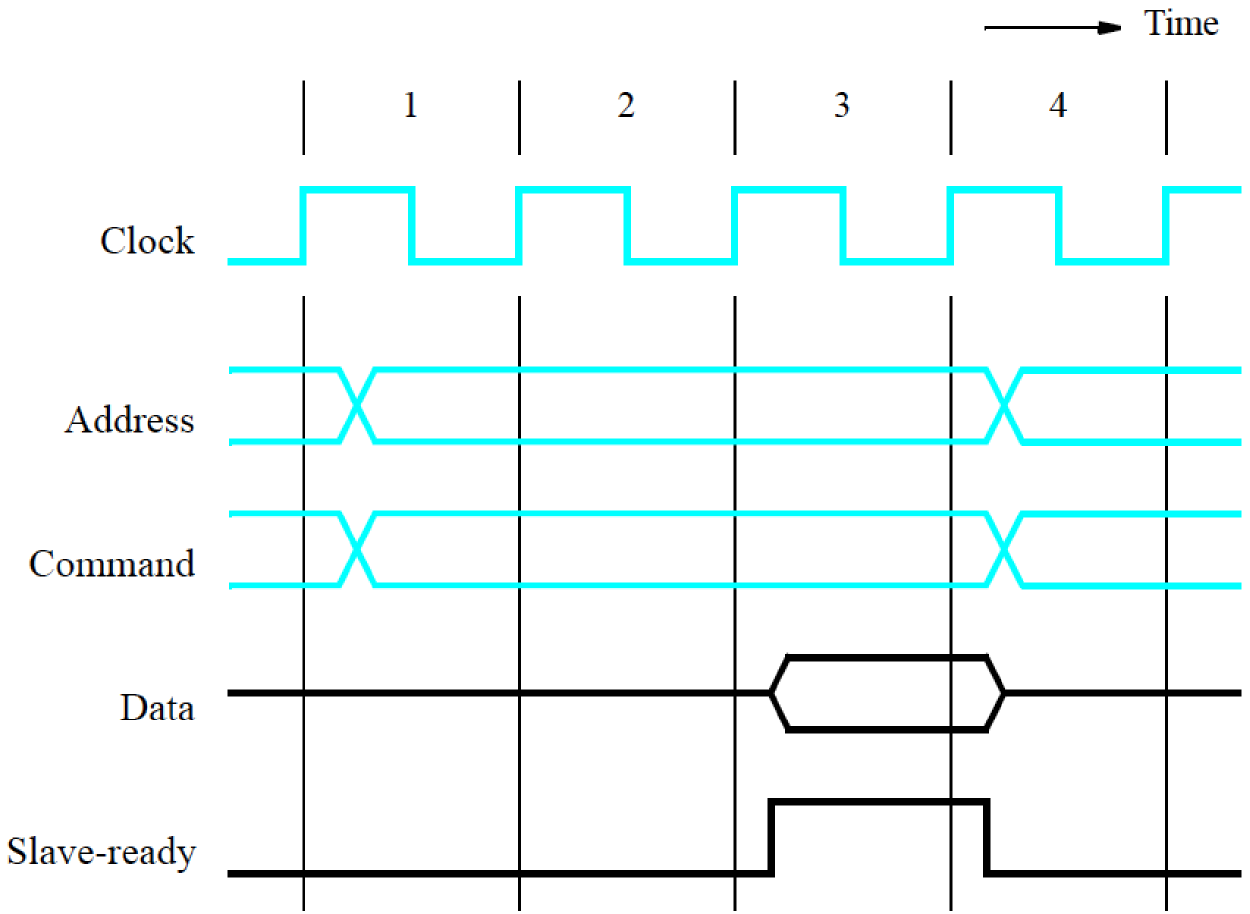
- All devices must operate at the speed of the slowest device!

- Master assumes that the data is made available, or has been received at t_2, but what happens if there is a malfunction?

Multiple-cycle data transfers

- To address both of these issues, most bus protocols include a device *response* signal.
  - A device response signal indicates that the address was successfully decoded and that it is ready to participate in a data transfer operation.
- Can also be used to adjust the delay of a transfer operation.
  - Usually this is accomplished by allowing a data transfer to span multiple cycles.

An input transfer using multiple clock cycles

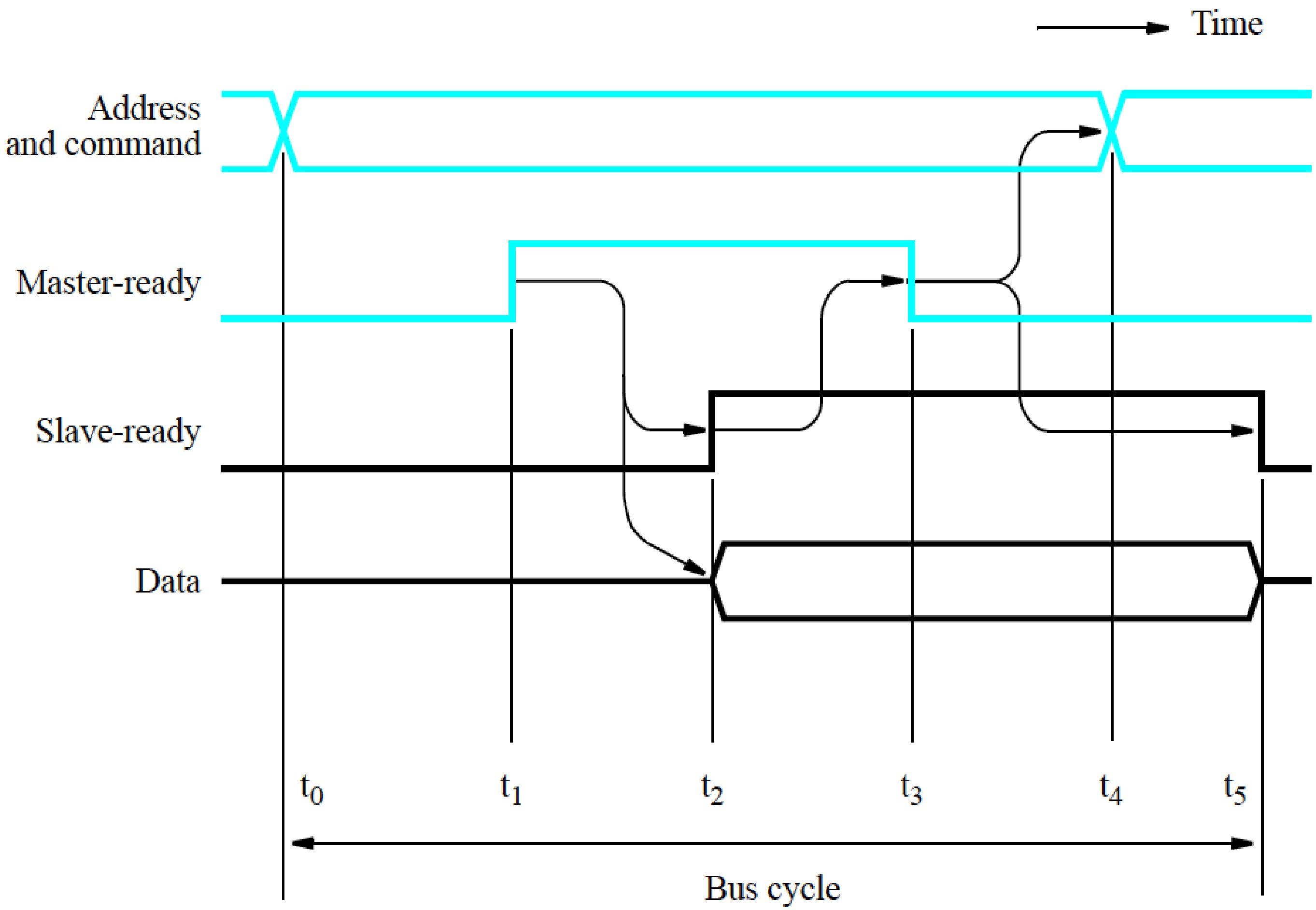


CC1: Master initiates read, slave decodes
CC2: Slave accesses data
CC3: Data is ready, slave places data on bus and asserts Slave_ready.
Master loads data into register at end of cycle.
CC4: Slave de-asserts Slave_ready and Master may initiate a new transfer

Asynchronous protocol

- Timing automatically adjusts to delays – no bus clock.
- Handshake protocol (exchange of command and response signals between master and slave) – each signal change results in a response: *full handshake* or *fully-interlocked*.
- Data transfer is controlled by two interlocked signals: `Master-ready` and `Slave-ready`.
- Whenever the processor takes an action, it waits for the device interface to respond before taking the next action, and vice-versa.

Handshake control: input operation



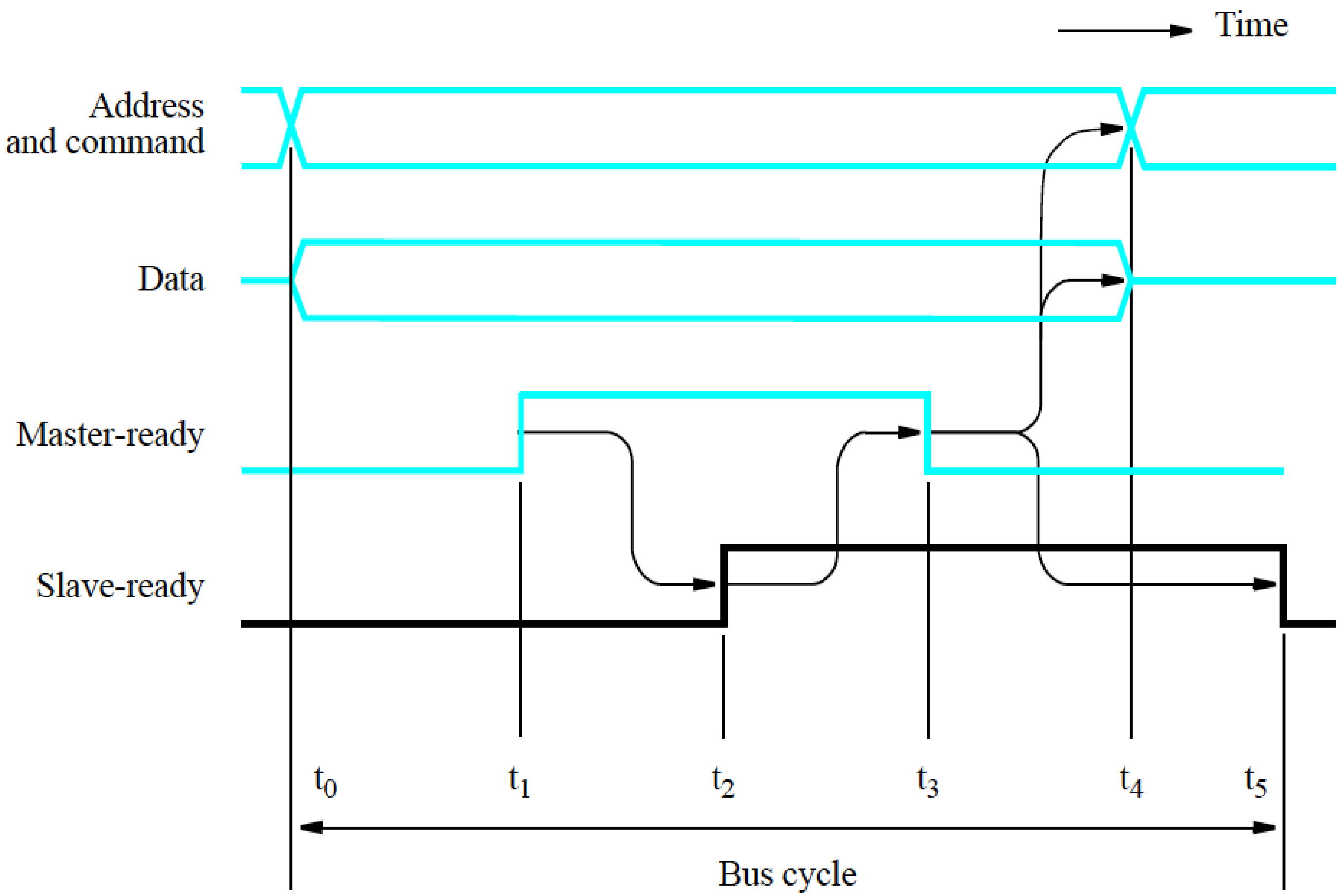
$t_1 - t_0$ > max. bus skew to prevent Master_ready from arriving at a device before the address and command and to permit enough time for device address decoding

$t_2 - t_1$ depends on distance between master and slave and slave circuit delay

$t_3 - t_2$ > max. bus skew + setup time

$t_4 - t_3$ > max. bus skew

Handshake control: output operation



Synchronous vs. Asynchronous

- Asynchronous adjusts to the timing of each device automatically.
- Synchronous requires careful timing design.
- Asynchronous transfer requires four end-to-end delays (2 round trips)
- Synchronous transfer only requires one round trip
- Synchronous is used in modern high-speed busses.

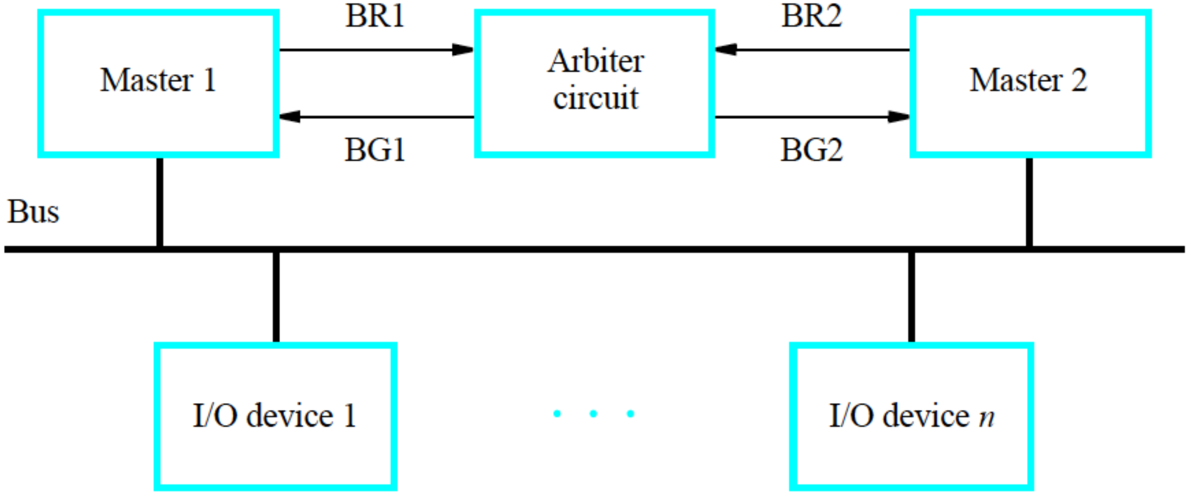
Arbitration: granting access to a shared resource

Say several devices wish to be bus master, e.g:

- There are multiple processors (cores) on the same bus
- The processor wishes to write to the bus, and an I/O device wishes to write directly to memory

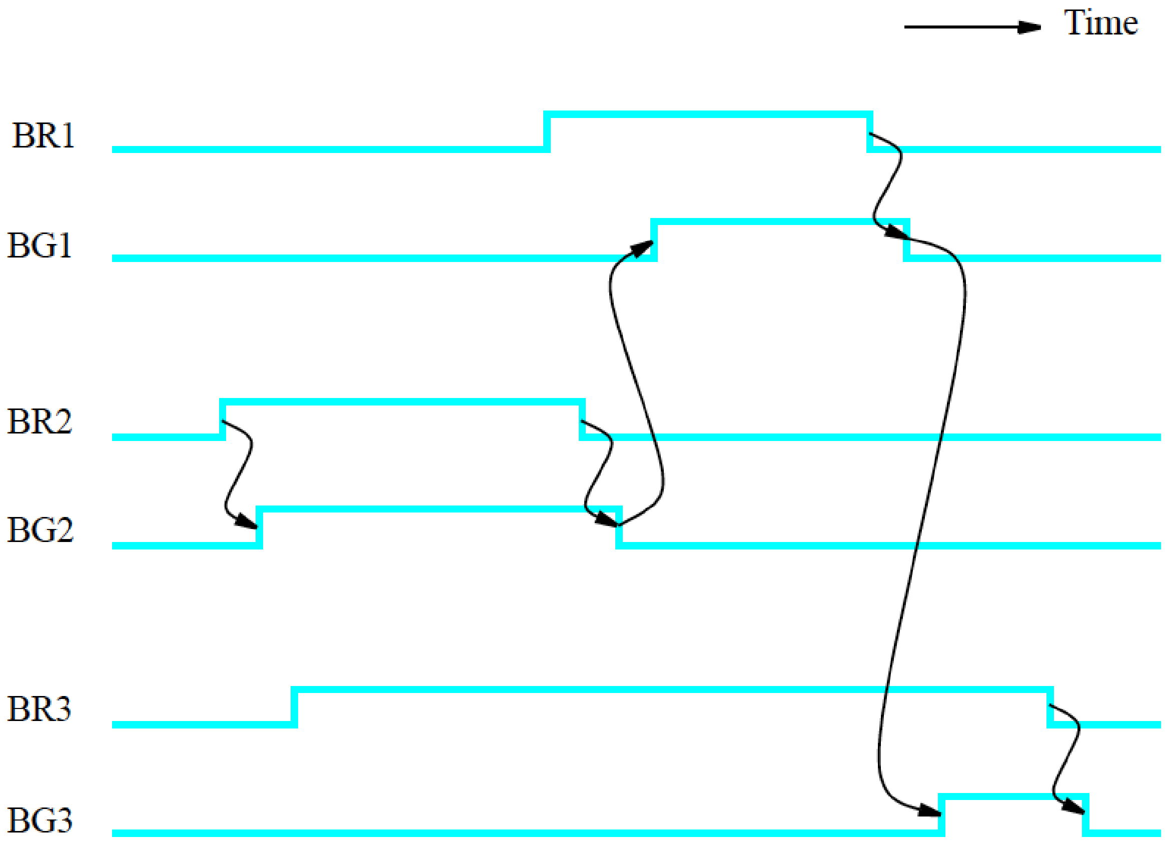
    Direct memory access (DMA)

Bus arbitration



- Devices request bus mastership.
- An arbiter grants the bus to the highest priority device.
- Control lines on the bus are used to request and grant the bus.

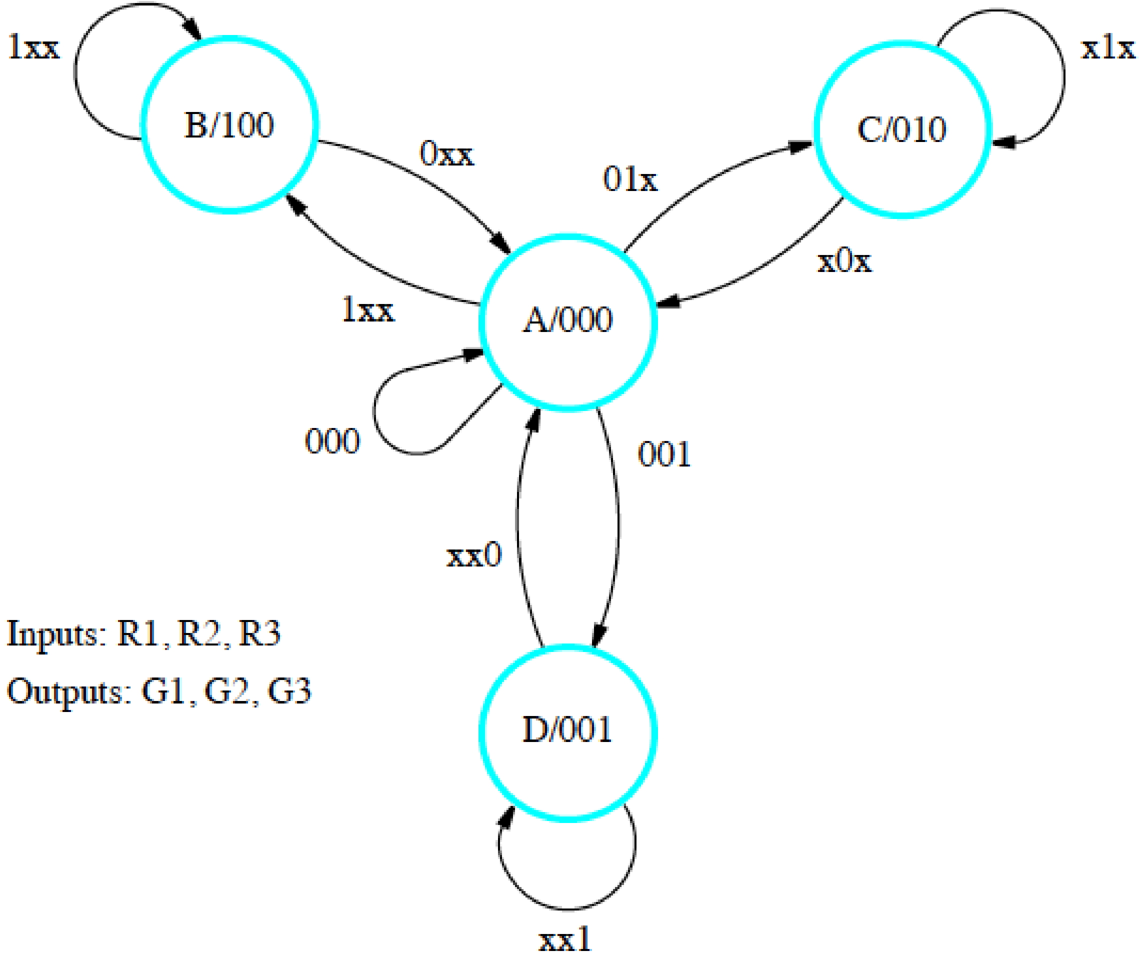
Granting the bus
priorities: BR1 > BR2 > BR3



Textbook Example 7.2

An arbiter receives three request signals, `R1, R2, R3`, and generates three grant signals `G1, G2, G3`. `R1` has the highest priority and `R3` the lowest.

Draw a state diagram that describes the behavior of this arbiter.



Inputs: R1, R2, R3

Outputs: G1, G2, G3

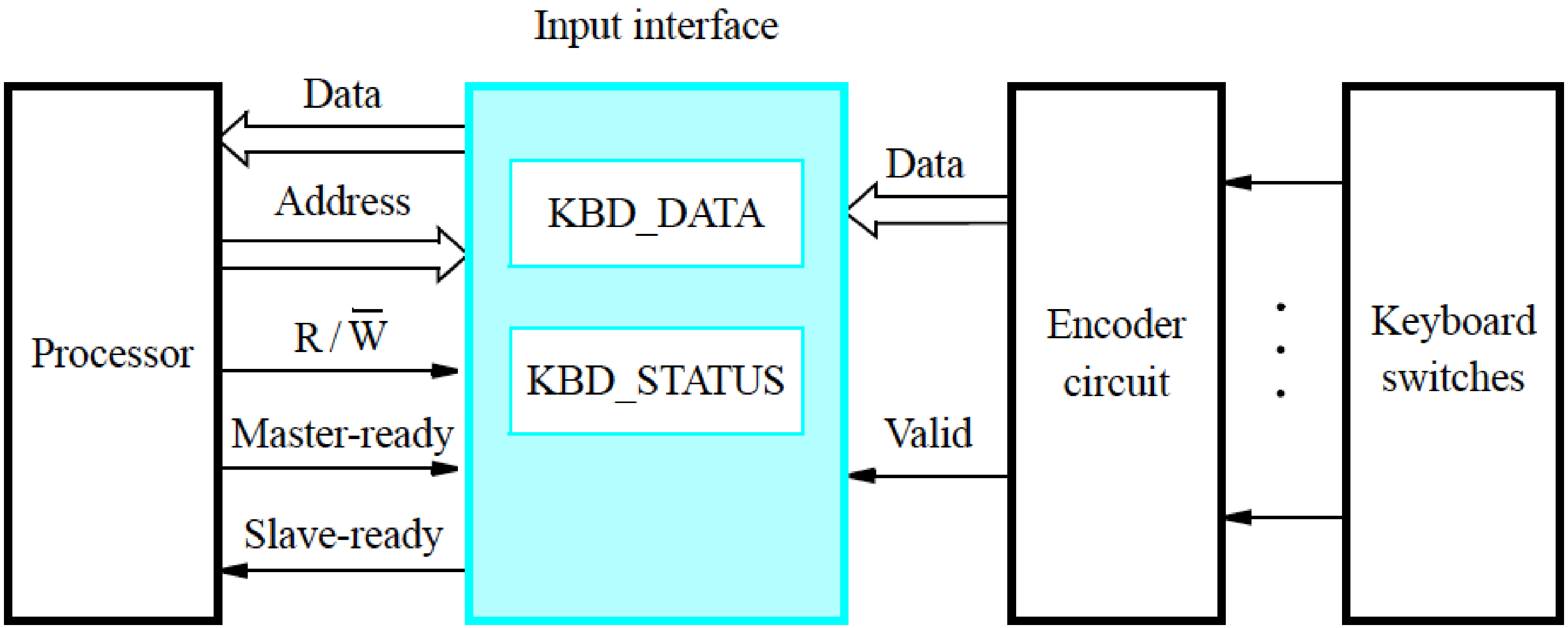
Hardware aspects of I/O: Parallel and Serial Interfaces

Textbook 7.4, 7.5

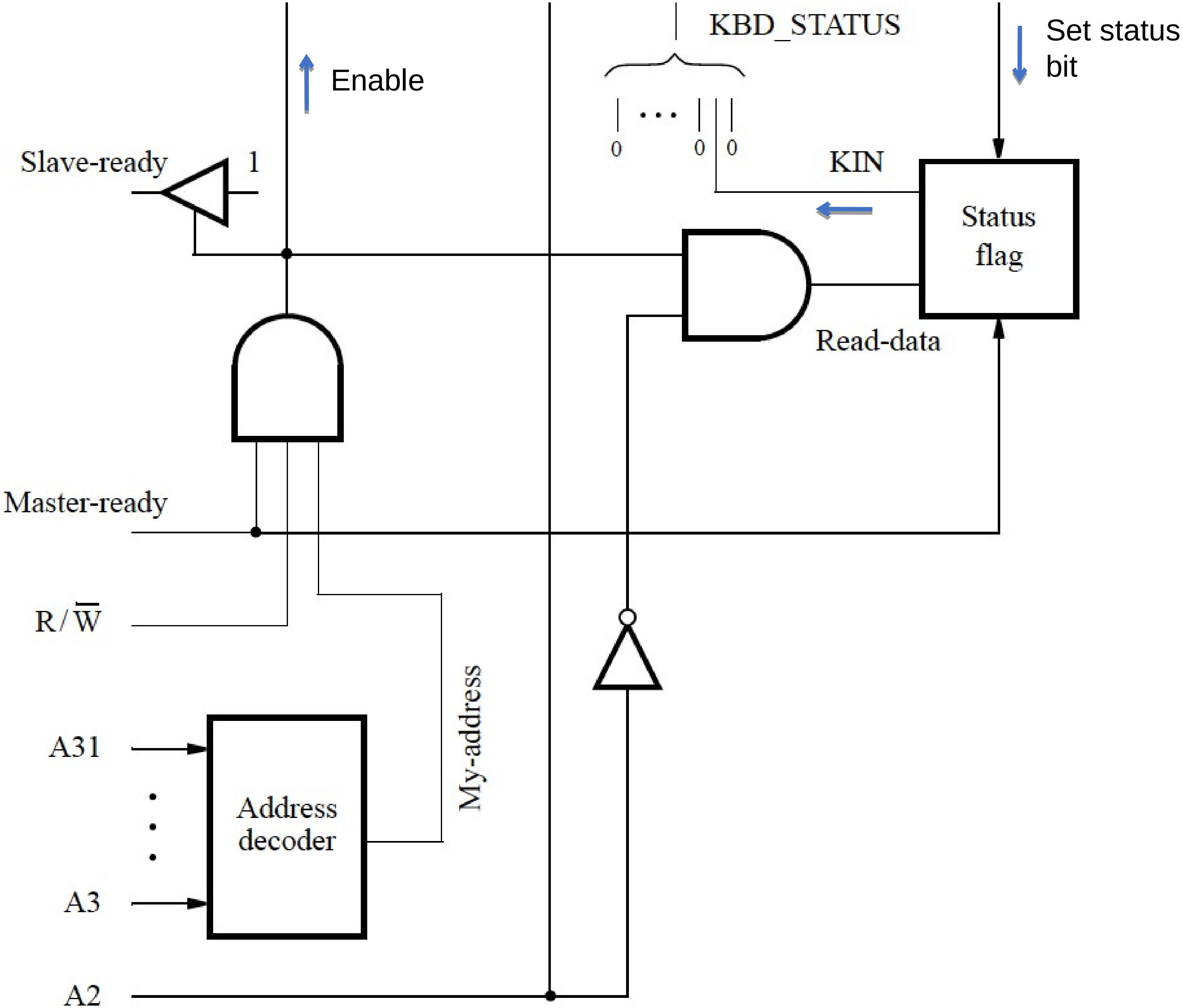
I/O ports

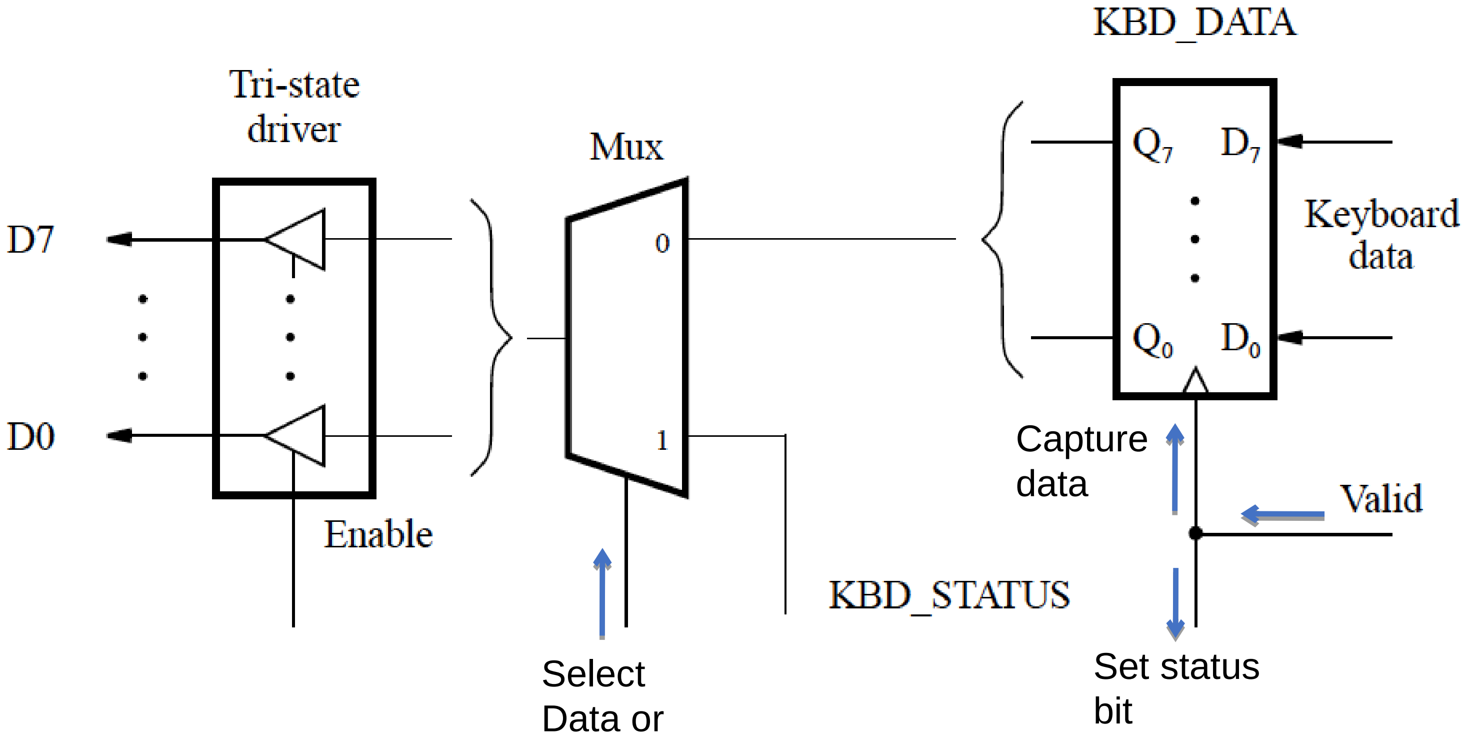
- An I/O port connects a device to the bus.
- Parallel ports transfer several bits of data simultaneously
- Serial ports transfer one bit at a time.
  - Communication with the processor is still parallel – conversion from parallel to serial happens inside the interface circuit

Input port: keyboard to processor



1. Valid changes from 0 to 1 --> KBD_STATUS.KIN = 1 and KBD_DATA loaded
2. Processor read of KBD_DATA --> KBD_STATUS.KIN = 0


KBD_STATUS
Set status
bit
Enable
Slave-ready
1
0
0
0
KIN
Status
flag
Read-data
Master-ready
R / $\overline{\text{W}}$
My-address
A31
Address
decoder
A3
A2



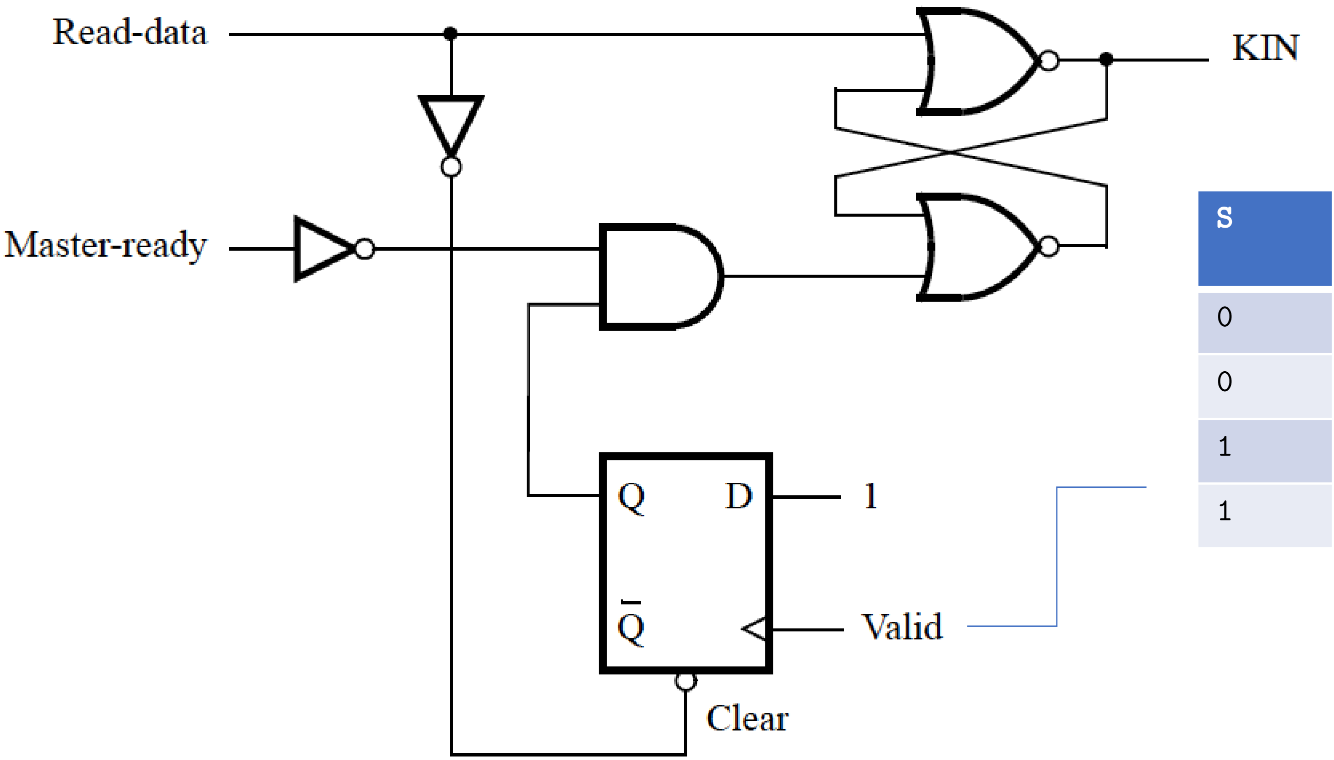
Capture data

Select Data or status

Set status bit

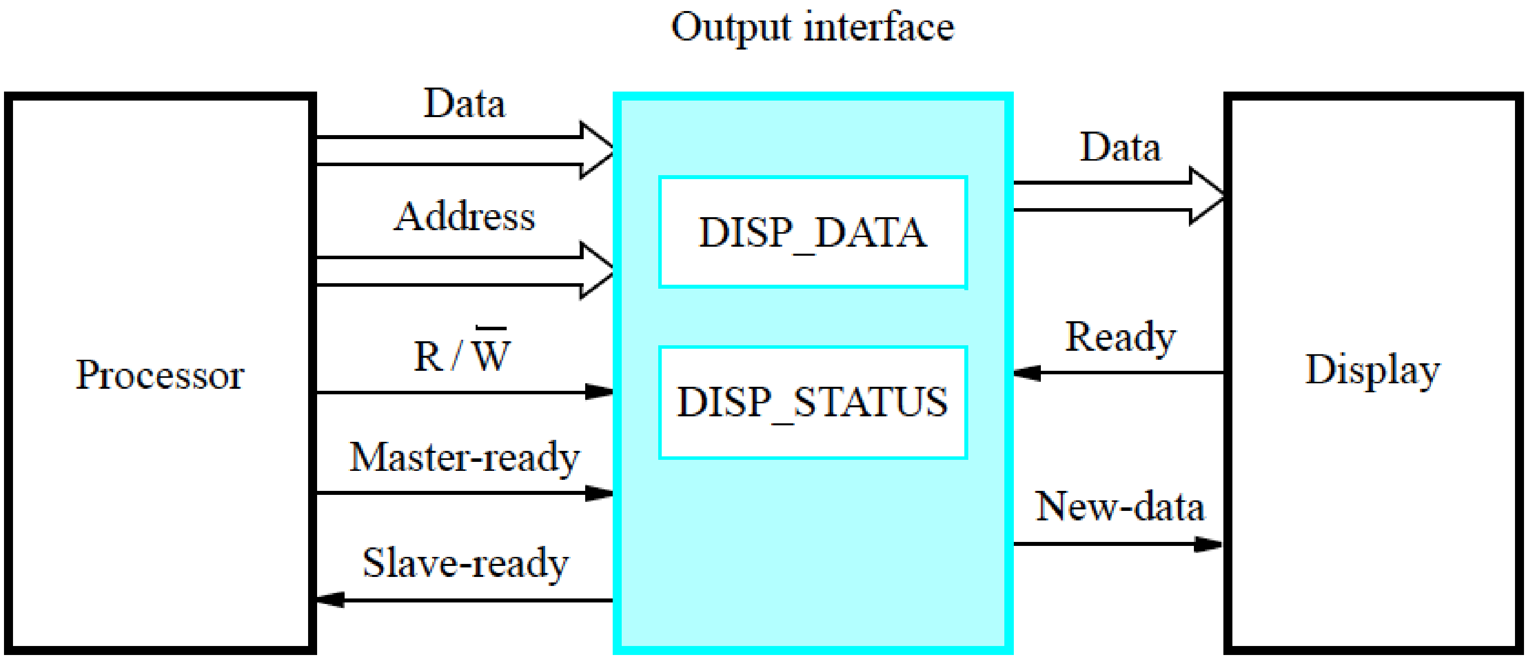
Detail of status flag control

- KIN is set by Valid and cleared by a read operation, but only when Master-ready is not asserted.



S	Read-data	KIN
0	0	hold
0	1	0
1	0	1
1	1	0

An output interface

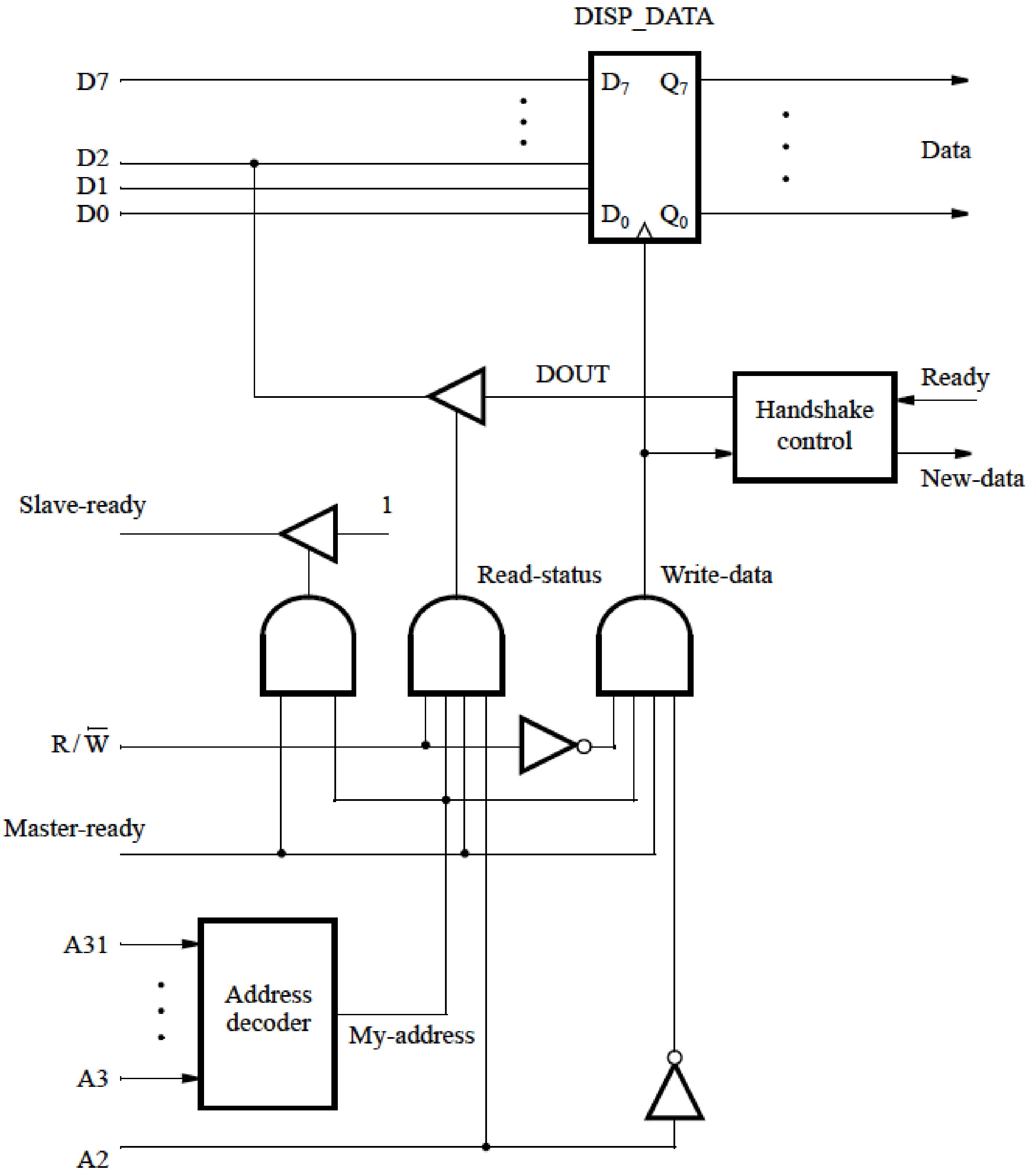


1. Display asserts Ready --> DISP_STATUS.DOUT = 1
2. Processor checks and finds DISP_STATUS.DOUT = 1 --> send char to DISP_DATA
3. This sets DISP_STATUS.DOUT to 0 and New-data to 1
4. Display sets Ready to 0 and accepts and displays the character in DISP_DATA

Write with A2 = 0 loads a byte into DISP_DATA

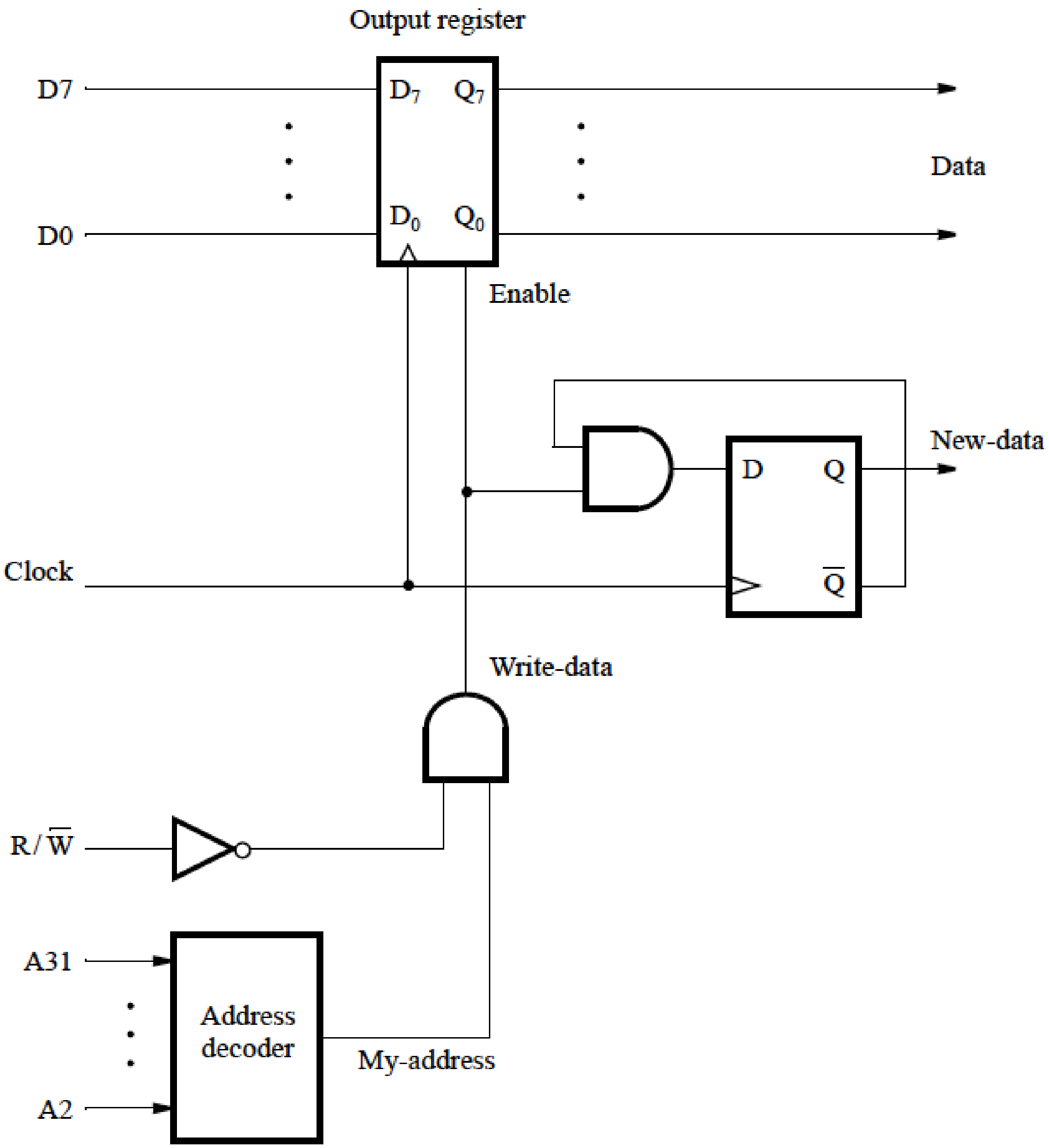
Read with A2 = 1 reads DISP_STATUS

DOUT is b2 of DISP_STATUS.



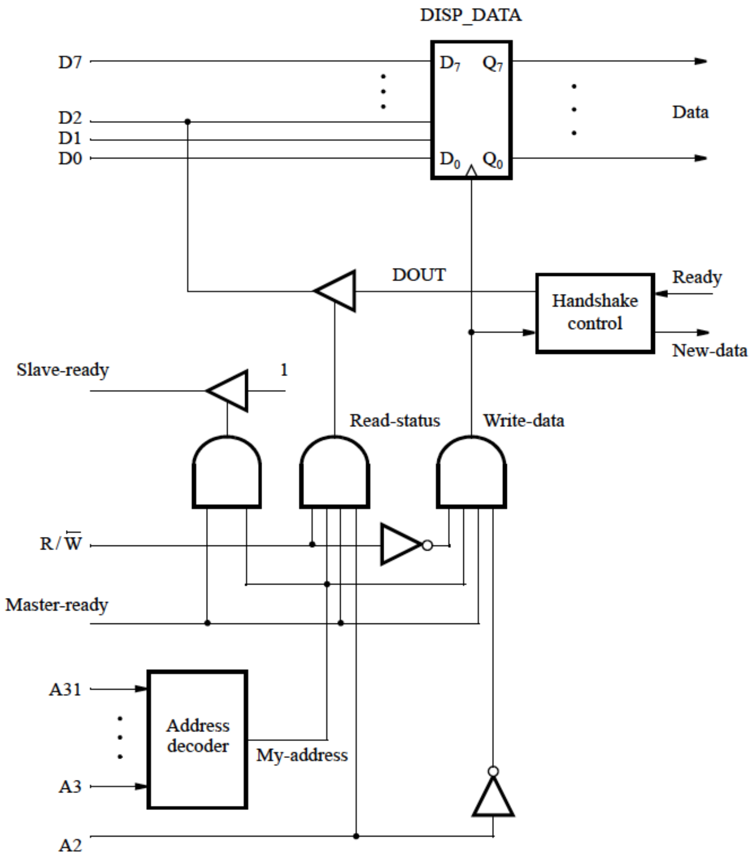
Textbook Example 7.3

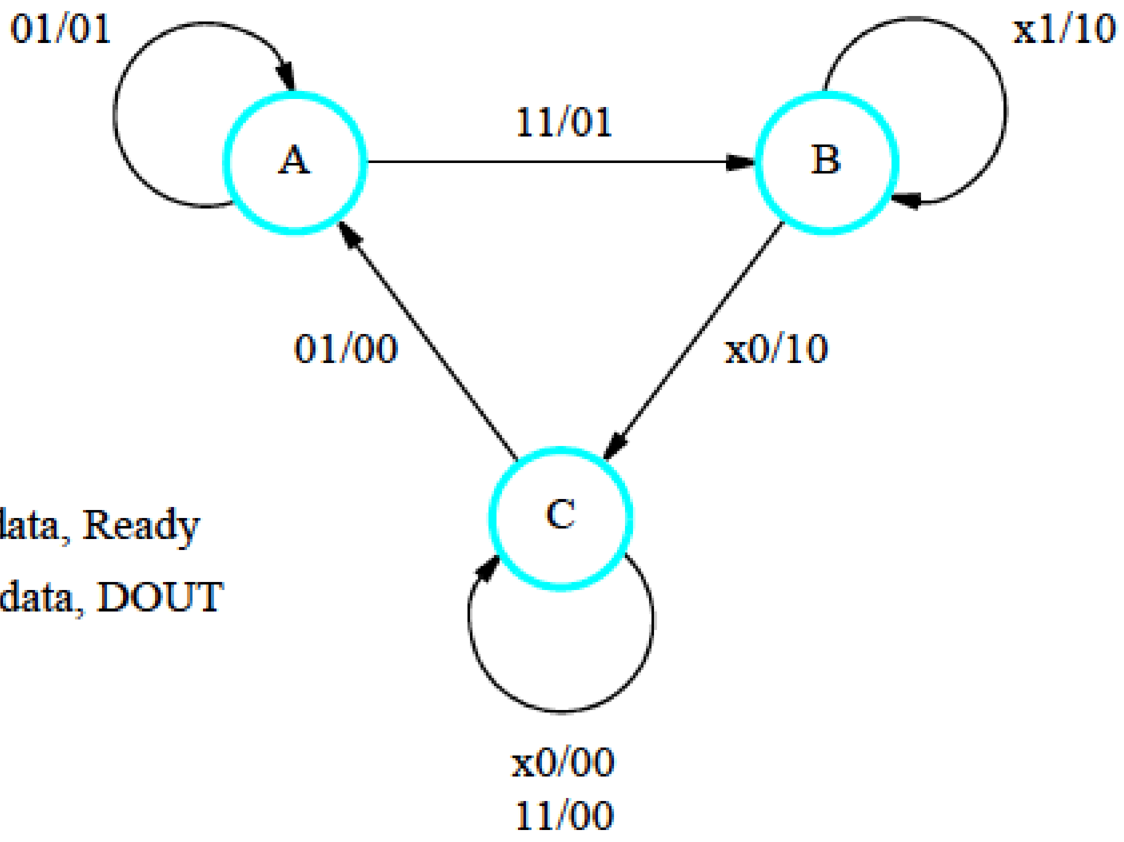
Design an output interface circuit for a synchronous bus. When data are written into the data register of this interface the interface sends a pulse with width of one clock cycle on a line called `New-data`. The pulse lets the output device connected to the interface know that new data are available.


Output register
D7
D0
D_7
Q_7
D_0
Q_0
Data
Enable
New-data
D
Q
$\overline{Q}$
Clock
Write-data
$R/\overline{W}$
A31
A2
Address decoder
My-address

Textbook Example 7.4

Draw a state diagram for a finite-state-machine (FSM) that represents the behavior of the handshake control circuit





Inputs: Write-data, Ready

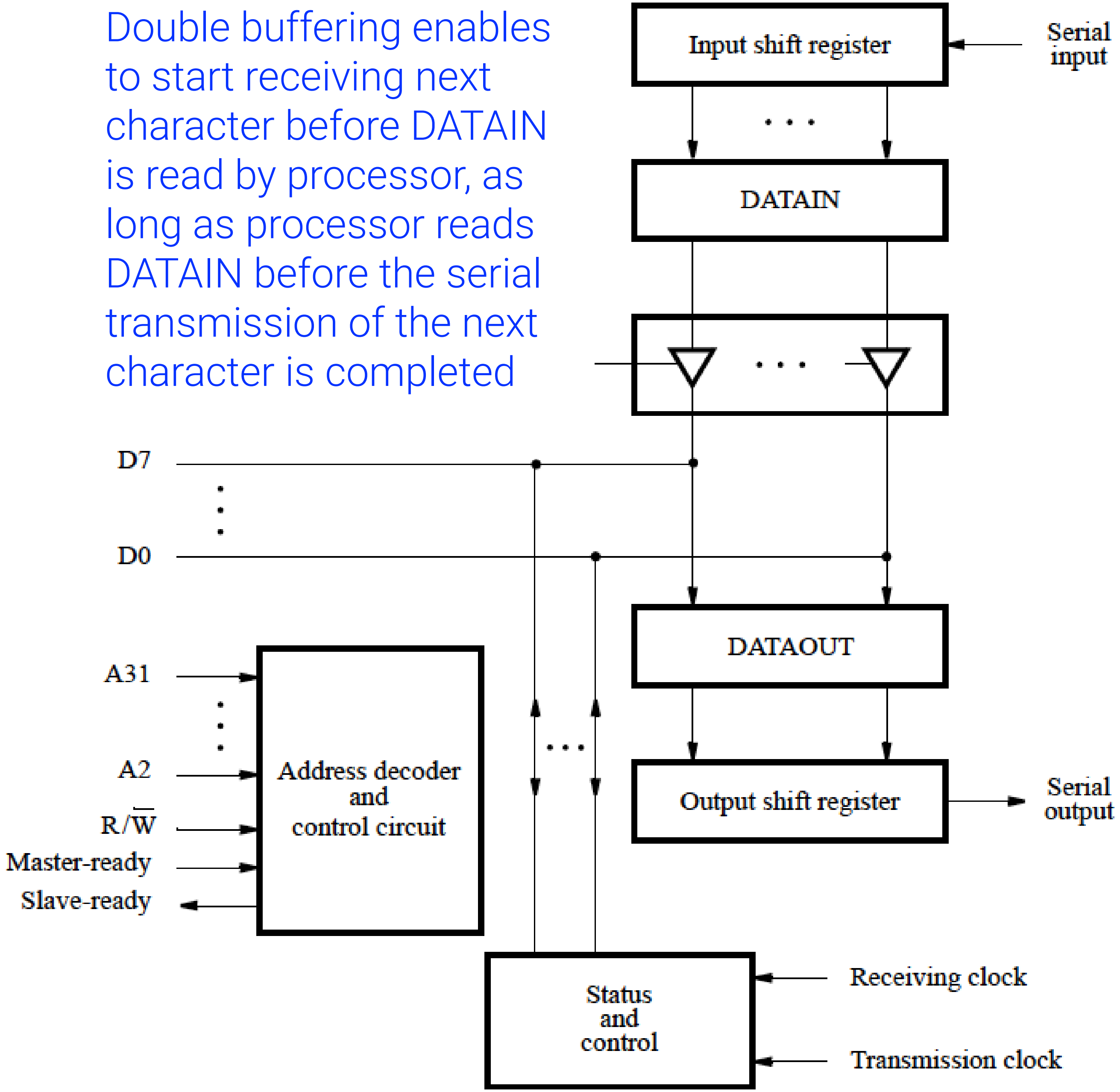
Outputs: New-data, DOUT

Serial links

- Many I/O interconnections use serial data transmission.
  - More suitable for longer distances
  - Less expensive
- Data are transmitted one bit at a time.
- Requires a means for the receiver to recover timing information.
- A simple scheme for low-speed transmission is known as “start-stop,” using a Universal Asynchronous Receiver Transmitter (UART).

UART

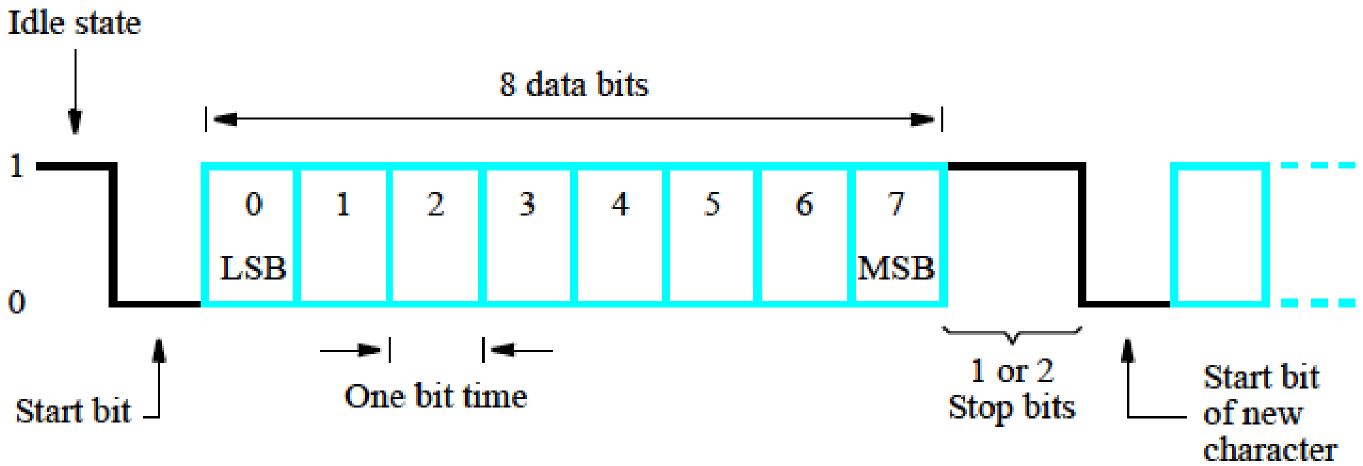
Double buffering enables to start receiving next character before DATAIN is read by processor, as long as processor reads DATAIN before the serial transmission of the next character is completed



Start-stop transmission

- Receiver and transmitted maintain their own unsynchronized clocks ($f_R \sim 16\ f_T$)

- Sample at middle of bit: modulo-16 counter reset at leading edge of start bit. At count of 8, check if the signal is still 0, and then reset counter. Sample next 8 bits at count of 16.



signals beginning of char

ensure proper transition for start bit of next char

Synchronous serial transmission

- Asynchronous works by detecting the 1-0 transition at the beginning of the start bit

- Very high-speed transmission – the waveform is not square and async is hard to get working

- A synchronous transmitter inserts codes (bit sequences) at the beginning of the transmission

- The receiver uses the knowledge of the code to generate a receiver clock that is synchronized to the transmitter clock

I/O interconnection standards

- Standards facilitate system integration using components from a variety of sources and encourage the development of many plug-compatible devices.

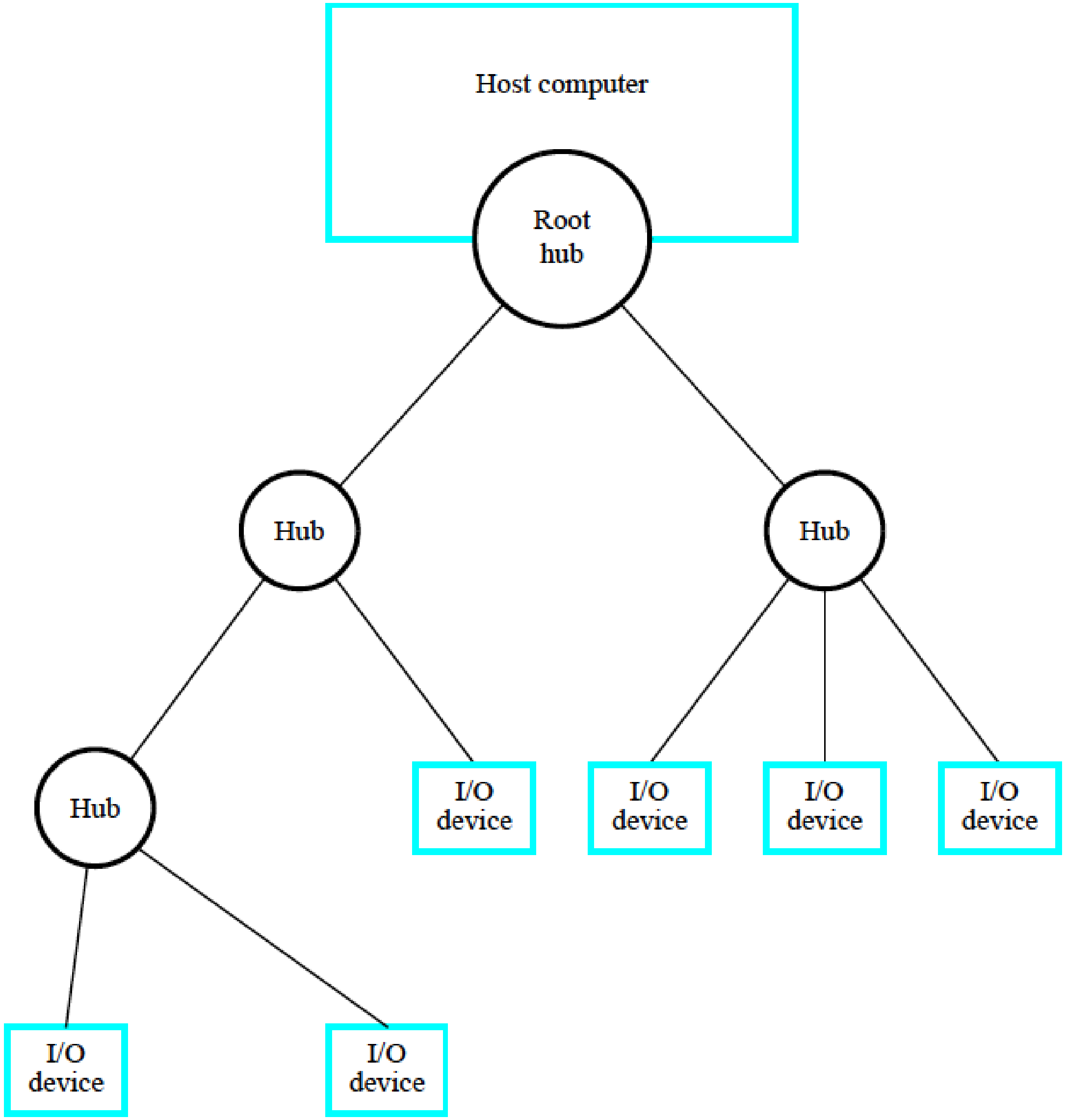
- Perhaps the most commonly encountered I/O standard today is the Universal Serial Bus (USB).
  - Memory keys, printers, external disk drives, cameras, etc.

USB features

- USB 1: 12 Mb/s, USB 2: 480 Mbps, USB 3: 5 Gbps
- Point-to-point connections using serial transmission and two twisted pairs (+5V, Ground, two data wires)
- Low-speed transmission is single-ended: one data wire for 0, the other for 1
- High-speed transmission uses *Differential signaling*
  - Data is encoded as the voltage difference between the two data wires
  - Noise is cancelled as it common to both wires

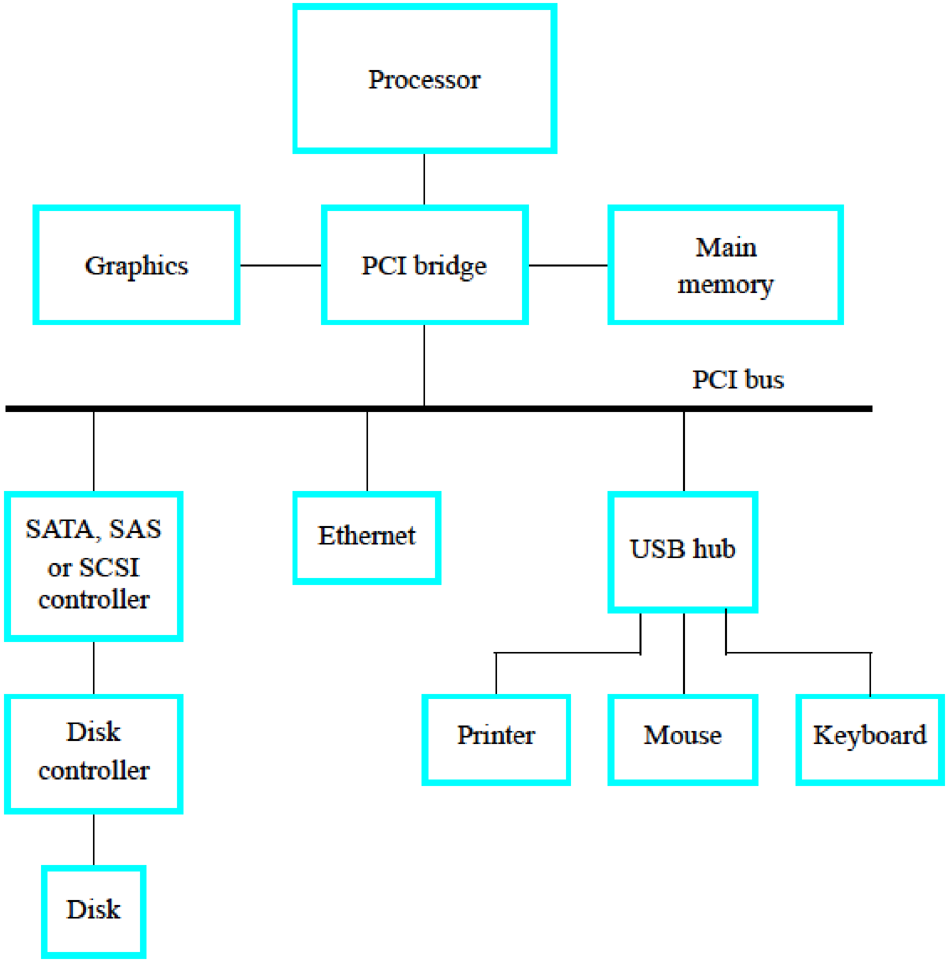
USB (Universal Serial Bus)

- Can connect many devices using simple point-to-point links and hubs
- Plug-and-play: system detects new device automatically
- USB works by polling devices to resolve simultaneous messages

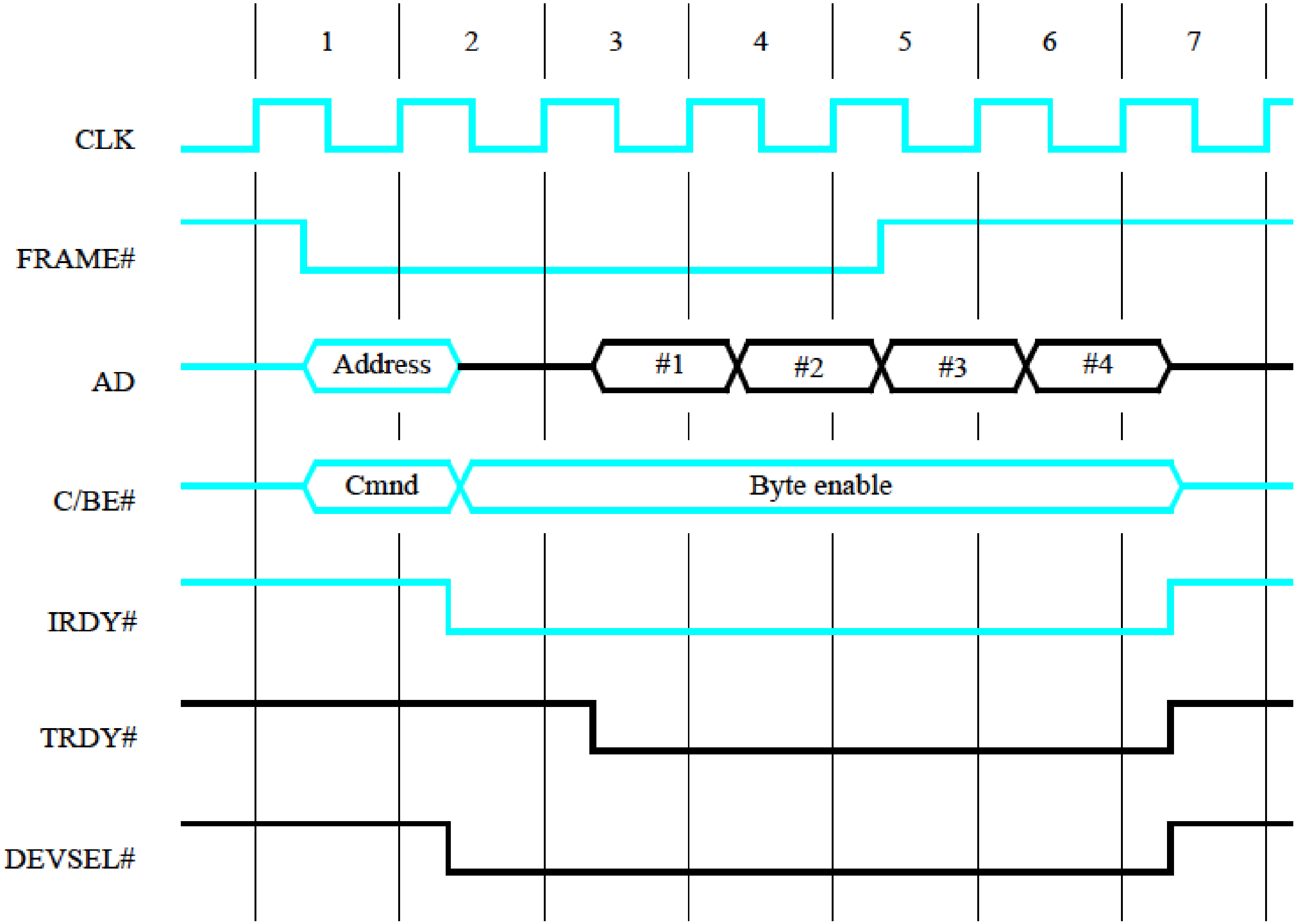


PCI (Peripheral Component Interconnect) Bus

- Processor-independent motherboard bus
- Devices on the PCI bus appear in the address space of the processor



Reading 4 bytes from device on PCI bus



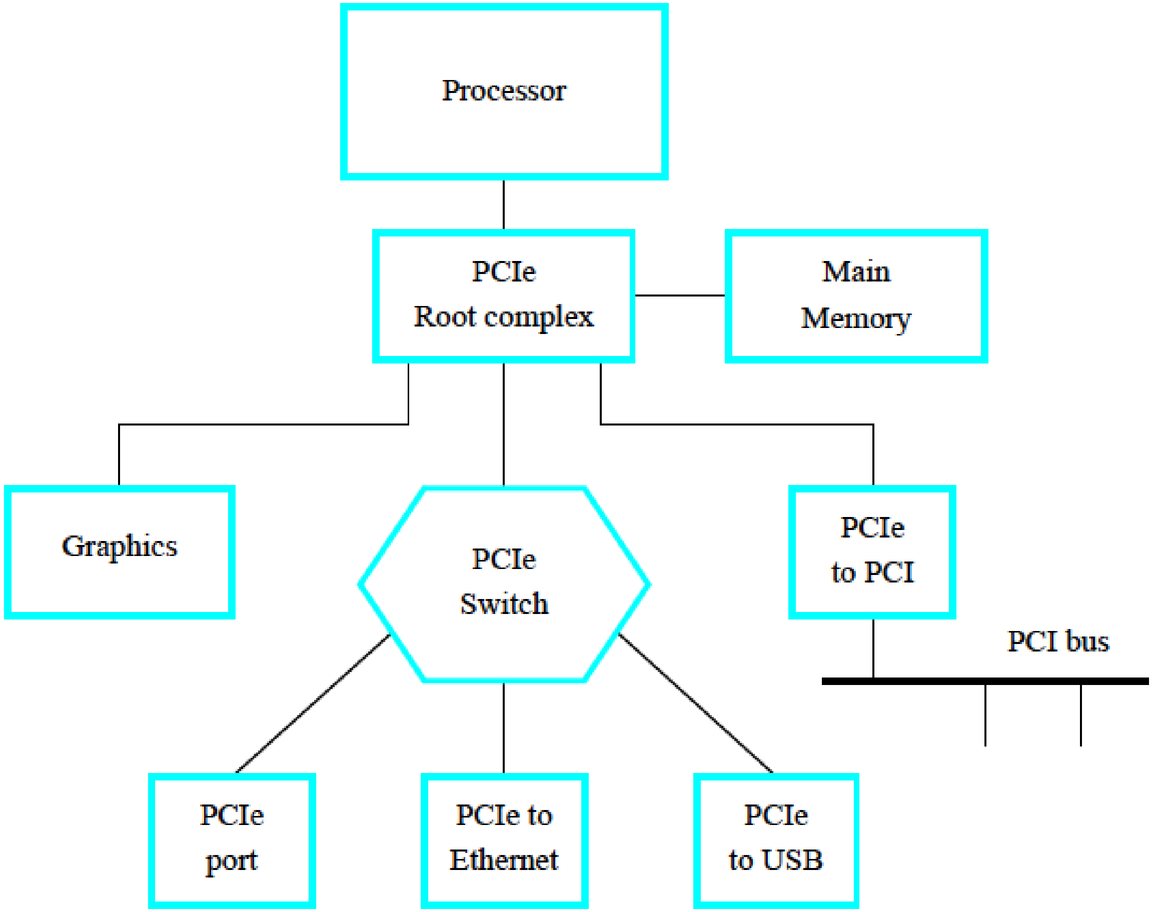
Plug-and-play

- PCI pioneered the plug-and-play feature, which was made possible by the bus’s initial connection protocol.
- There are up to 21 device connectors on the PCI bus.
- Each PCI-compatible device has a small ROM with information on the device characteristics.
- Processor scans all connectors to determine whether a device is plugged in.
- It assigns an address to each device and reads the contents of its ROM.
- With this information, it selects the appropriate device driver software, performs any initialization that may be needed, etc.

PCIexpress

- Point-to-point connections with one or more switches forming a tree.

- Root complex provides high-speed ports for memory and other devices



PCIe links

- The basic connection is called a lane.
- A lane consists of two twisted-pairs or optical lines
  for each direction of transmission.
- The data rate is 2.5 Gb/s in each direction.
- A connection to a device (link) may use up to 16 lanes.
- The PCIe protocols are fully compatible with PCI,
  e.g., the same initial connection protocol is used.

What’s next

- In the next chapter we will look at memory technology and how to build an efficient memory organization.