Module 2.1
The Input/Output System

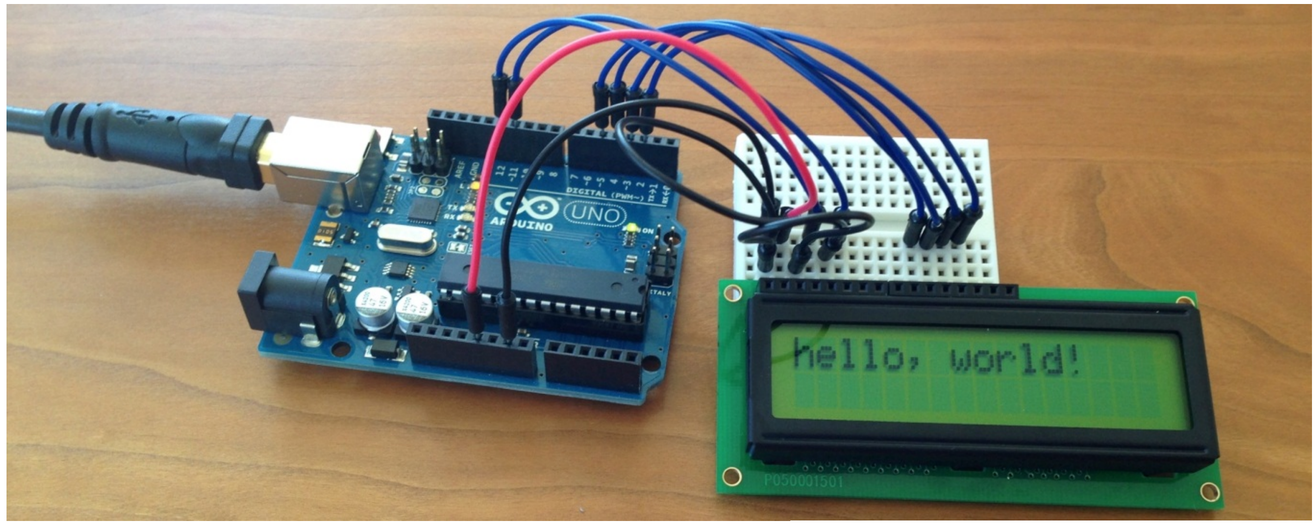
ec

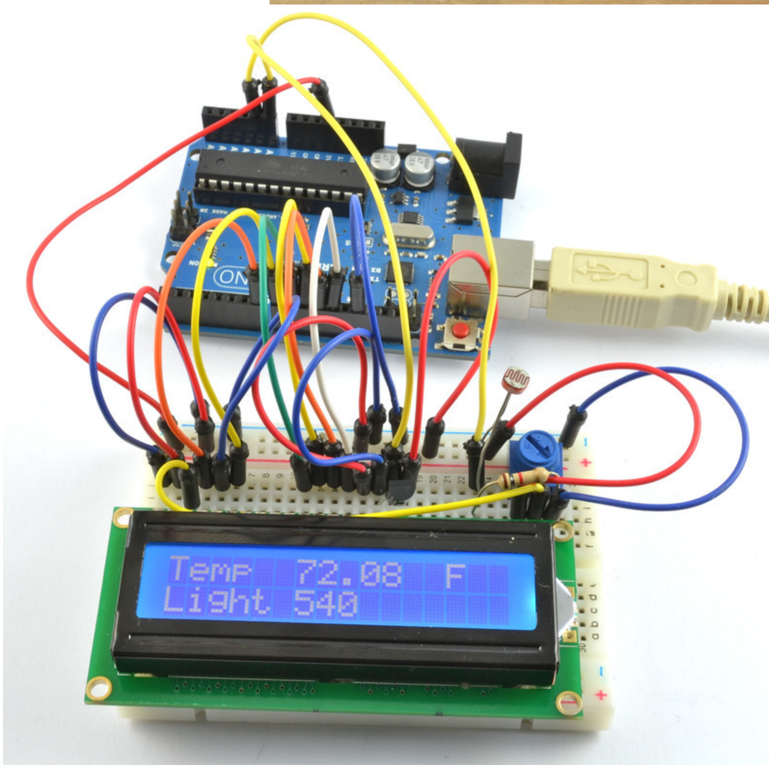
Outline

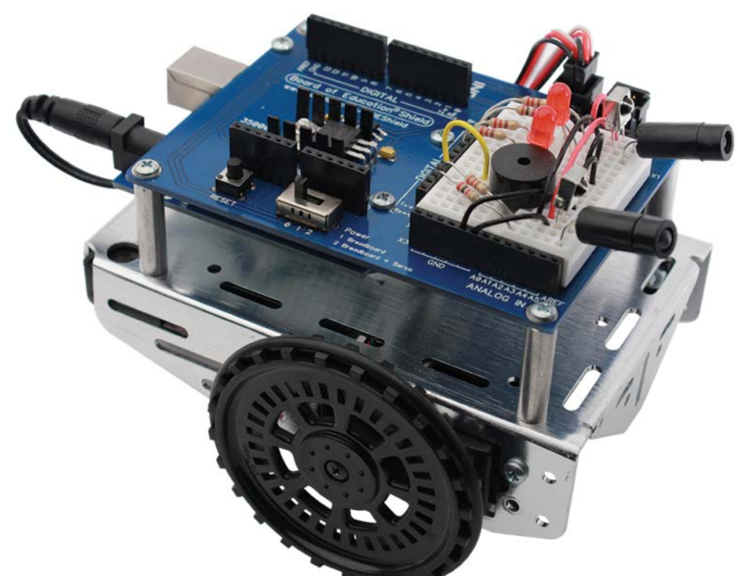
- Introduction
- Structure and functions of the I/O System
- Programmed I/O **LAB2**
- Exceptions
- Interrupt-based I/O **LAB3**

- References:
  - D. A. Patterson & J. L. Hennessy; Computer Organization and Design. The hardware/software interface. Chapter 6.
  - W. Stallings; Computer Organization and Architecture.

CPU ←→ External World

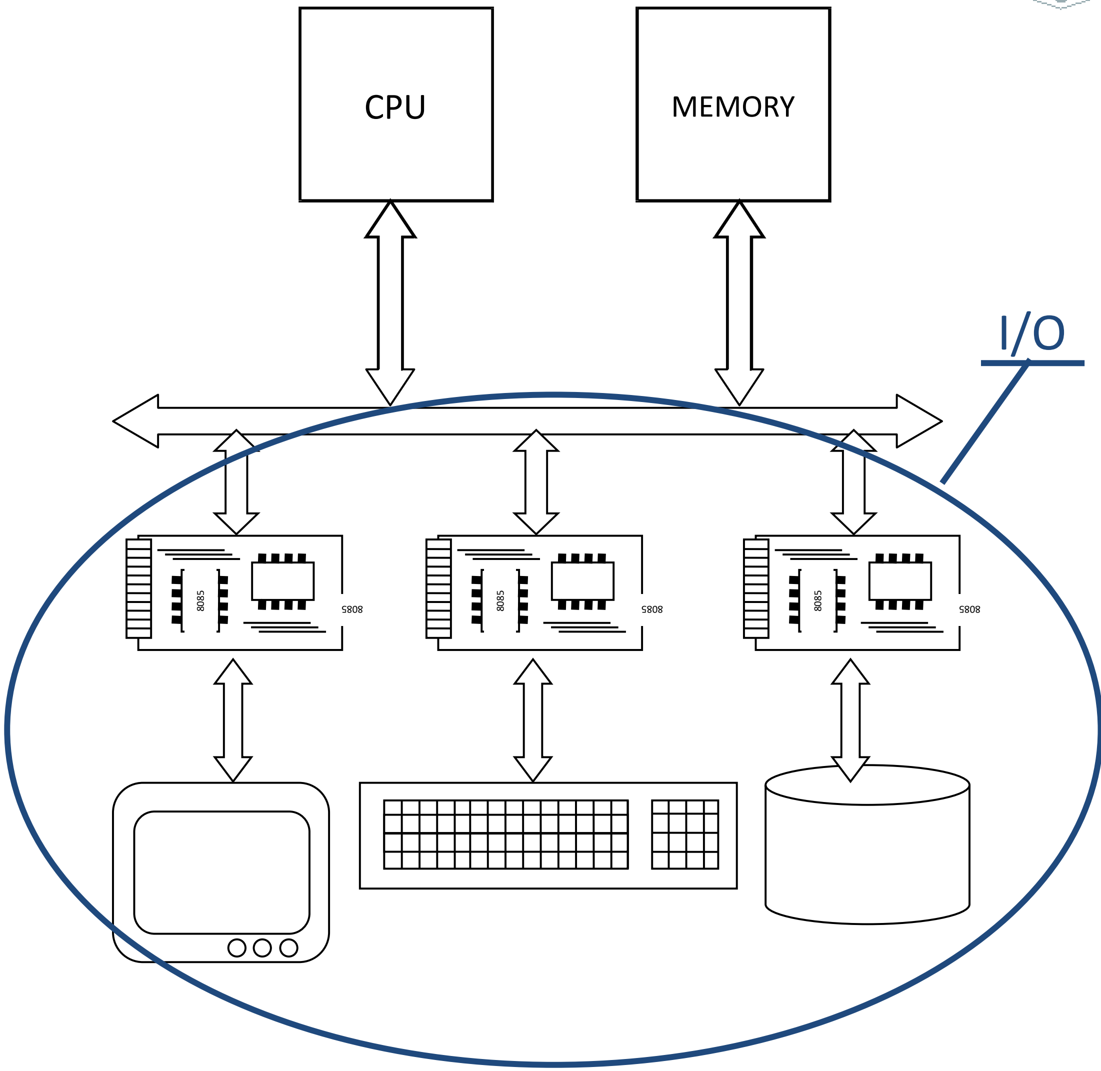






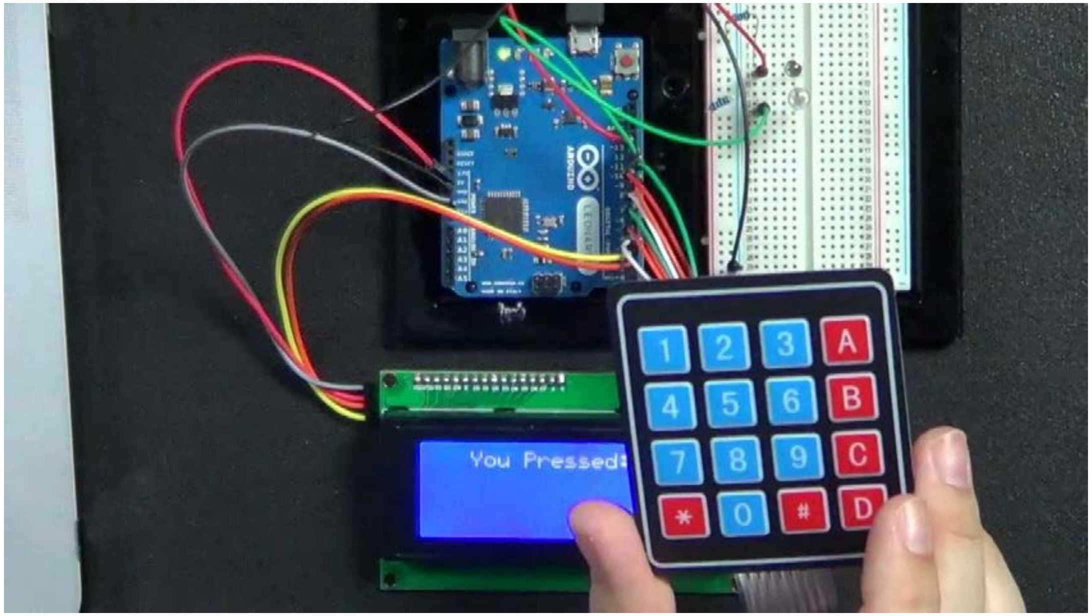
Need of the I/O system

- CPU ←→ External world
- Data input:
  - Device to CPU
- Data output:
  - CPU to device
- In many cases, the I/O system is the element that limits system performance



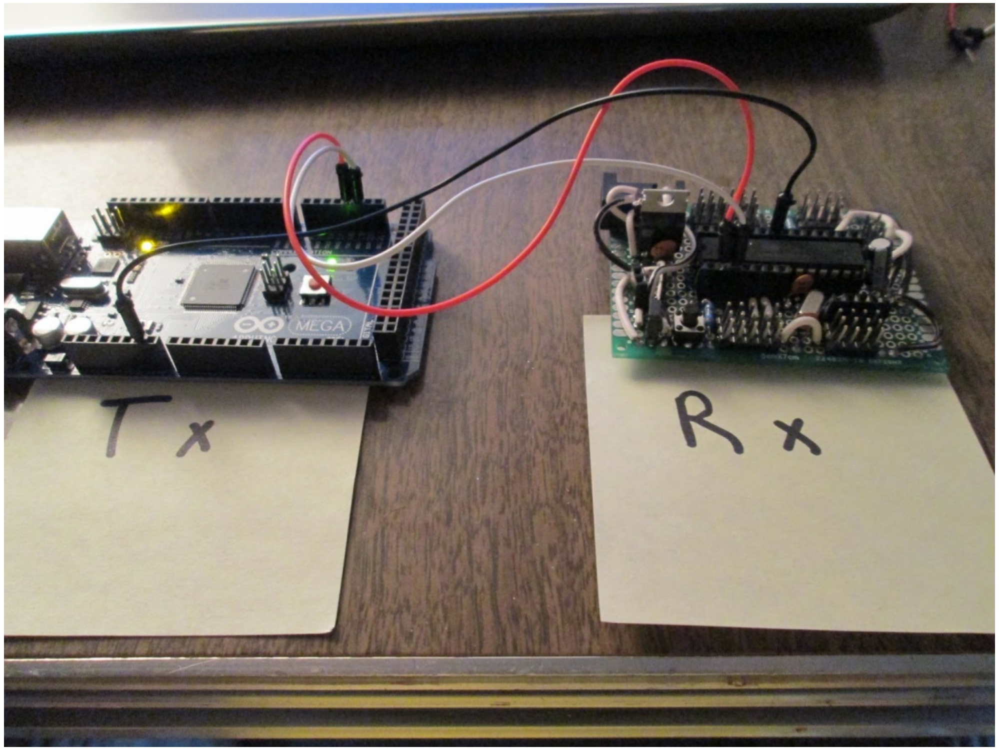
Types of peripherals/devices

- Data display devices.
  - These devices interact with users, transfering data between them and the machine
  - Mouse, keyboard, screen, printer, etc.



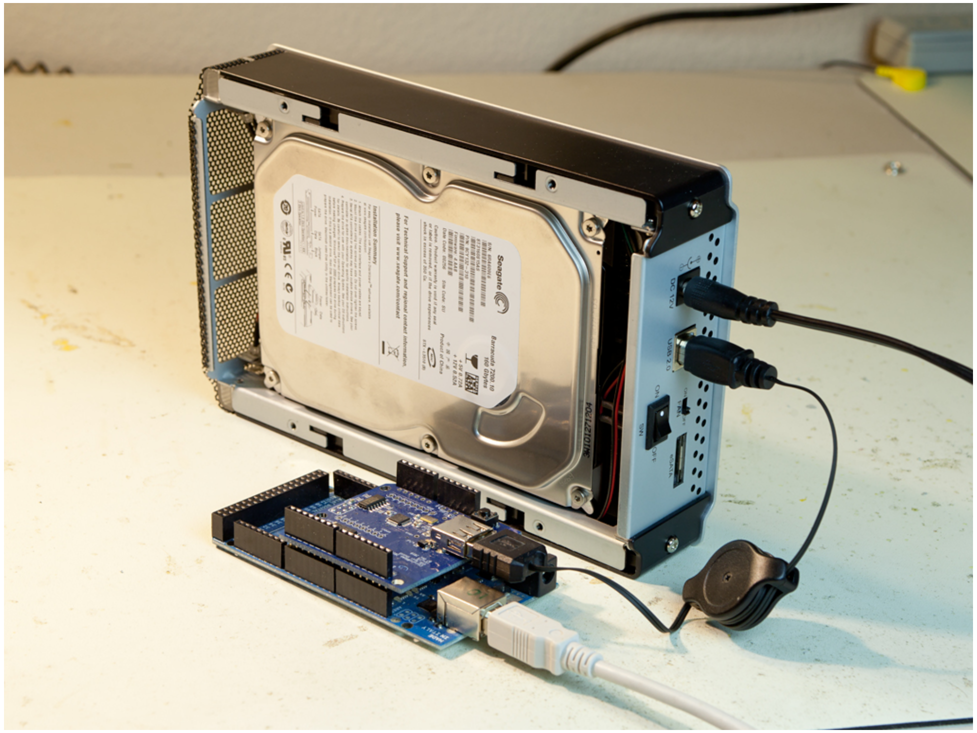
Types of peripherals/devices

- Devices for communication.
  - These devices allow data transfers between machines
  - Network



Types of peripherals/devices

- Data storage devices
  - The lower components of the memory hierarchy (discs,...) where permanent storage is provided.



Different features

- Type of data transmitted
  - Bytes, blocks, etc.
- Transfer rate
  - Data Bandwidth: amount of data that can be transferred per unit of time (measured in Mbit/s, MB/s ...)
    - For example, storage (SATA disks) and network (Gigabit Ethernet) devices provide high bandwidth
  - Latency: time required for the first data to reach the destination (measured in seconds)
    - Latency of most I/O devices is higher than the processor speed

Diversity of I/O devices

Device	Behavior	Partner	Data Rate (KB/sec)
Keyboard	Input	Human	0.01
Mouse	Input	Human	0.02
Line Printer	Output	Human	1.00
Laser Printer	Output	Human	100.00
Graphics	Output	Human	100,000.00
Network-LAN	Communication	Machine	10,000.00
Floppy disk	Storage	Machine	50.00
Optical Disk	Storage	Machine	10, 000.00
Magnetic Disk	Storage	Machine	30,000.00

Each device needs a different interface (the I/O Controller)

We need different ways of interacting with them (I/O techniques)

Outline

- Introduction
- Structure and functions of the I/O System
- Programmed I/O
- Exceptions
- Interrupt-based I/O

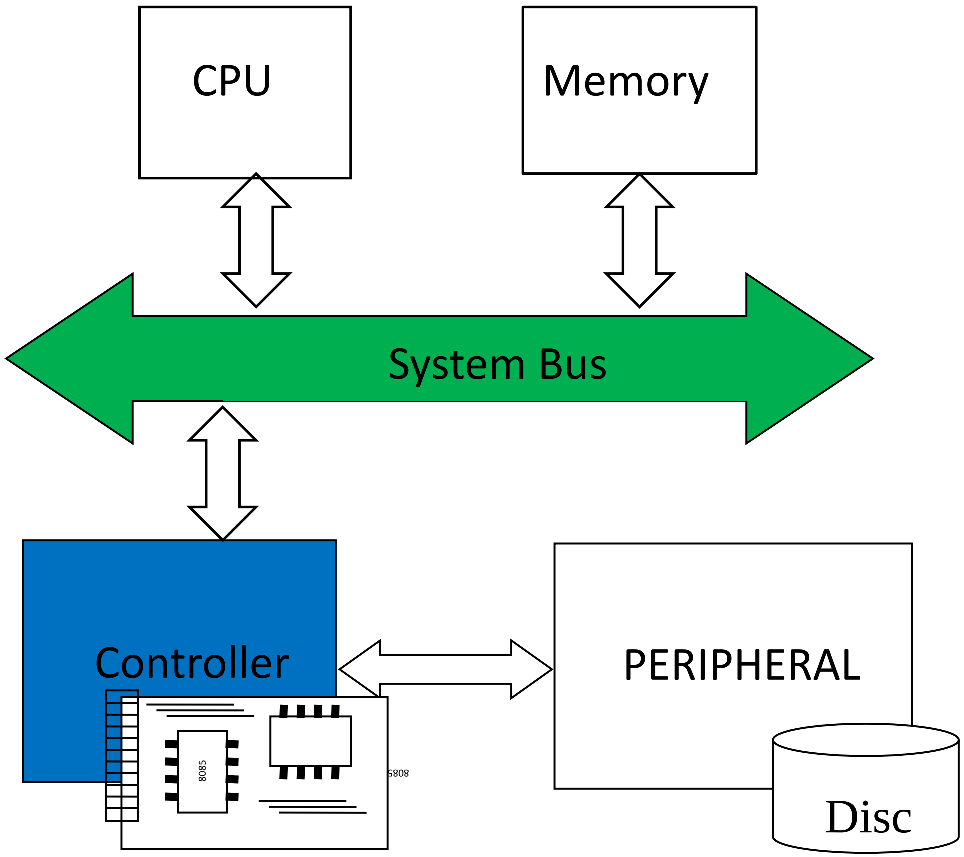
I/O system structure

- Components:
  - Peripheral/device
  - I/O module, controller (hardware)
    - Board
  - Software Driver
    - Driver

  OS

  - Comunication channel
    - Bus (shared) or point-to-point

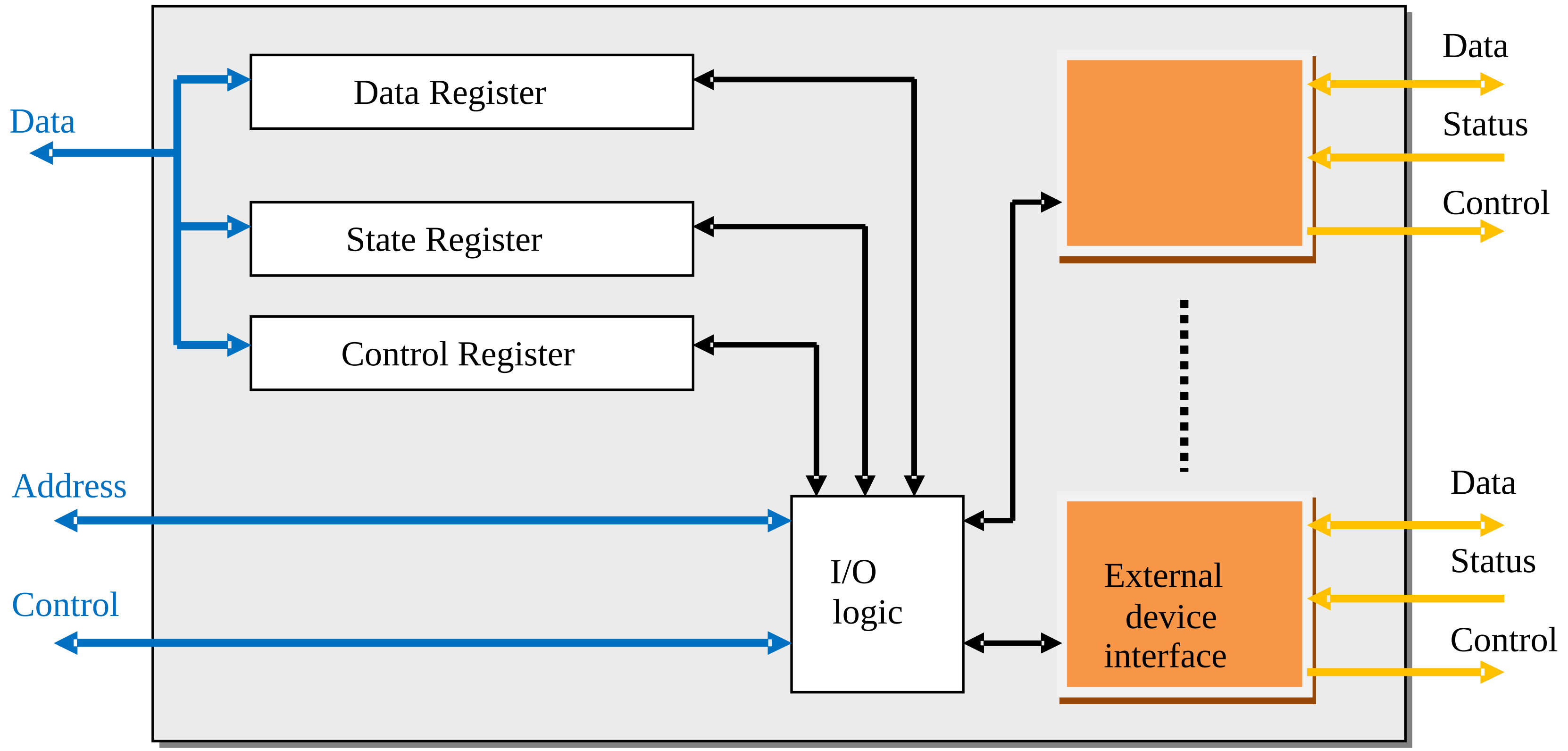
- I/O system organization



Device Controller

- Interface between a device and the processor.
- Funtions:
  – Control and timing
    - It adapts the different transfer rates
  – Communication with the processor or memory
  – Communication with the peripheral
  – Buffering (intermediate storage of data)
  – Detection of errors

Structure of a device controller



Registers in a device controller

- **Data Register**
  - Used to communicate data between CPU and a peripheral
- **State Register**
  - Contains the device status: ready, error, busy?
- **Control Register**
  - Encodes the request from the CPU to the device (print a character, read a data block, etc.)
- **WARNING: These are NOT registers in the CPU. They are usually part of a different chip.**

I/O transaction

- It includes two parts:
  - **Request: Sending the address (and command)**
    - The CPU writes in the control register of the chosen controller. ADDRESING.
  - **Response: Data transfer**
    - The CPU reads/writes data from/to the data register of the controller in due time. SYNCHRONIZATION

Addressing I/O controllers

- Isolated I/O
  - Different address space on the main memory and the I/O addresses
  - There are specific I/O instructions
    - IN dir_E/S, Ri (CPU $\leftarrow$ Peripheral)
    - OUT Ri, dir_E/S (Peripheral $\leftarrow$ CPU)
  - Example: Intel x86

- Memory Mapped I/O
  - Memory and the I/O devices share the address map
  - They can use the same load/store instructions (lw/sw)
    - LOAD Ri, dir_E/S (CPU $\leftarrow$ Peripheral)
    - STORE Ri, dir_E/S (Peripheral $\leftarrow$ CPU)
  - Example: ARM
    - Each register in an I/O module is located in a memory address within a reserved range

Example: GPIO Controller

- It manages multifunction pins in the chip
- For switching on a LED connected to one pin:
  - The CPU writes in a control register to set the pin as an output pin. (0 output, 1 input)
  - The CPU writes in a data register the value to show on the LED. (0 on, 1 off)
- The GPIO controller includes an electronic circuit for:
  - Providing the pin a voltage of Vcc or GND depending on the data register value (if configured as output)
  - Reading the value from a pin (if configured as input)

Example: GPIO Controller

- Addresses:

  Control register PCONB: 0x01D20008
  Data register PDATB: 0x01D2000C

- Request:
  - The CPU writes in PCONB to set the pin as an output pin.

```
mov r0,#0
ldr r1,=PCONB    @0x01D20008
str  r0,[r1]
```

- Data transfer:
  - The CPU writes in PDATB the value to show on the LED.

```
mov r0,#0
ldr r1,=PDATB    @0x01D2000C
str  r0,[r1]
```

Synchronization

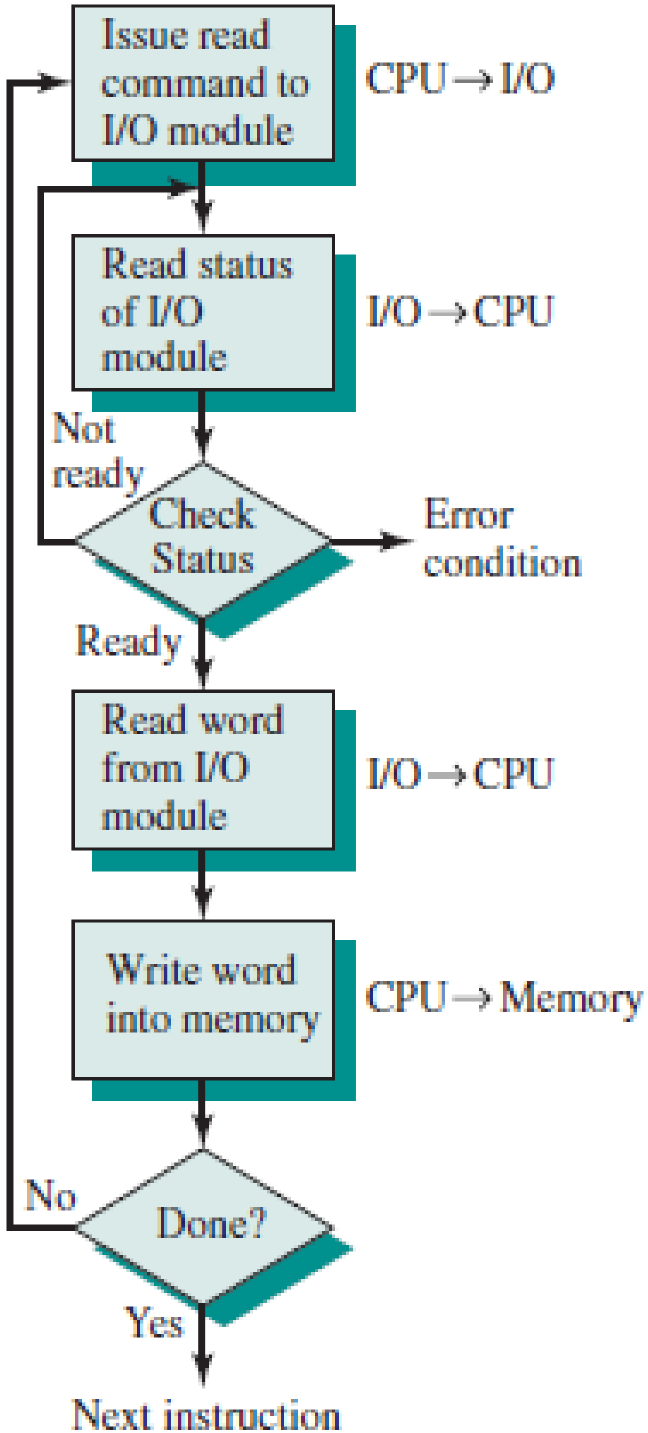
- The CPU and the devices have different speeds.
  - The CPU needs to verify that the device is ready to receive the data.
  - And also when the transfer is finished.
  - And that it was succesful…
- Two basic I/O mechanisms:
  - Programmed I/O (polling)
  - Interrupt-driven I/O

Outline

- Introduction
- Structure and functions of the I/O System
- Programmed I/O
- Exceptions
- Interrupt-based I/O

What is it?

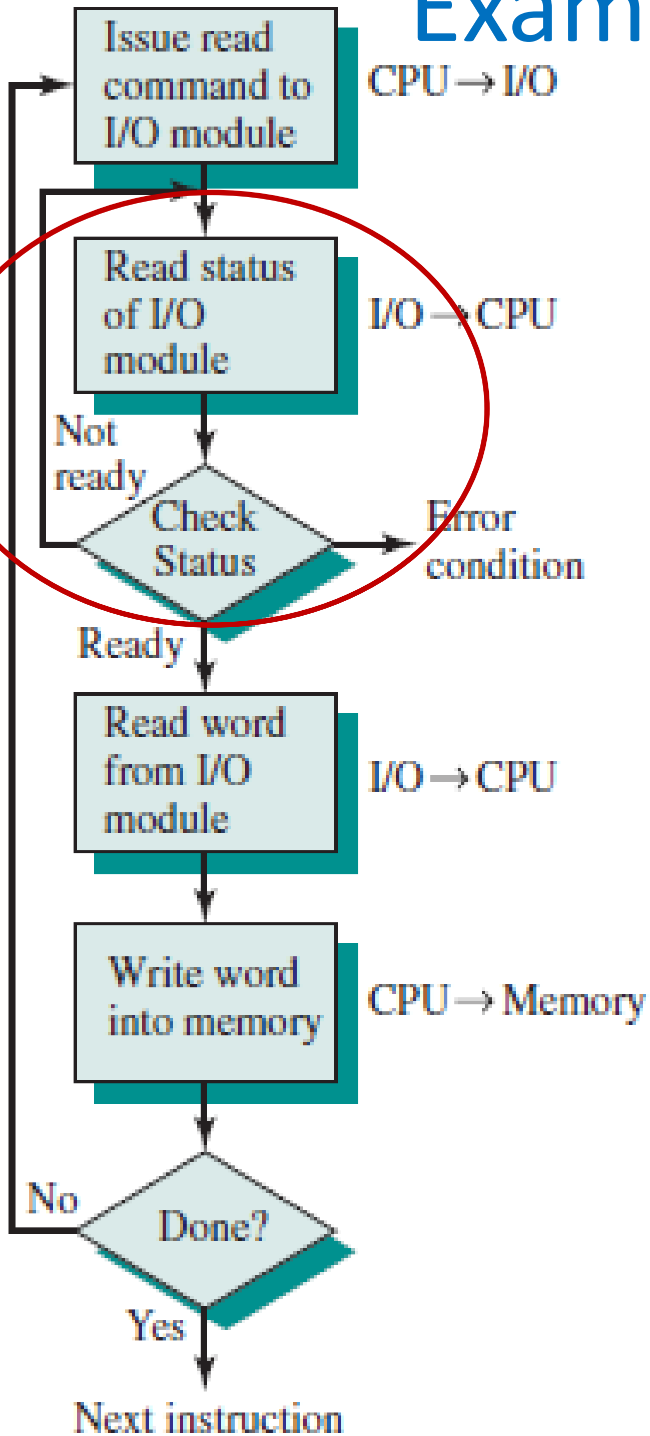
- When the CPU wants to make a transfer:
  - It entrers a loop which keeps reading the state register (“polling”) until the device is ready to transfer data
  - Transfer data



I/O Controlled by program (polling)

- It is easy but it has several problems
  - The CPU does no useful work during the wait loop
    - With slow devices the loop could be repeated thousands/millions of times.
  - The functionalities of the program stop during I/O operation
    - Example: In a video-game, it is not possible to stop the flow of the game until the user presses a key.
  - Difficulties to communicate simultaneously with multiple peripherals
    - It can not communicate with a peripheral while waiting that another one be ready for a transfer.

Example: GPIO Controller



Bit 6 in register PDATG is:
0 if button-1 is pressed
1 if it is not

```
int  reg                        pseudocode

reg = rPDATG
If ( reg-bit 6 ) == 0)
            button-pressed=Yes
else
            button-pressed=NO
```

```
            ldr r0,=PDATG
…
loopDet:    ldr r1,[r0]
            and r1,r1,#0x0040        @ bit 6 only
            cmp r1,#0
            bneq loopDet      @ still waiting
@ button1 pressed
```

Example: GPIO Controller

```
            ldr r0,=PDATG
…
loopDet: ldr r1,[r0]
            and r1,r1,#0x0040           @ bit 6 only
            cmp r1,#0
            bneq loopDet       @ still waiting
@ button pressed
```

Exercise: ¿how many times is the loop executed if I press the button once per second, the processor runs at 40MHz and every instruction takes 3 cicles to complete?

40 x 1000000 cycles/s / 12 cycles/iteration = 2,6 millions iterations in 1 second

* Explain Lab 2

Is there any room for improvement?

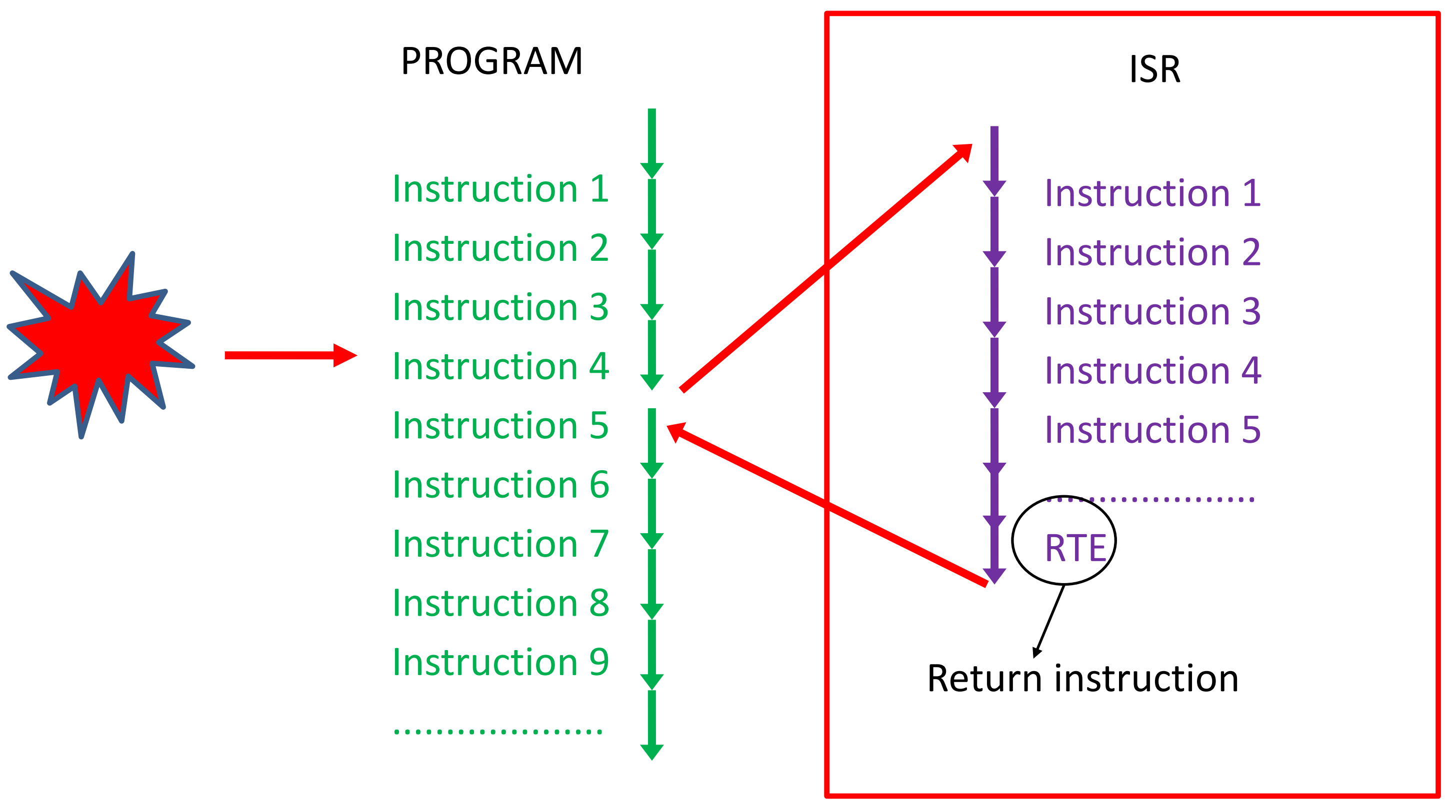
- The synchronization loop executes useless instructions
- Could the peripheral notify the CPU when it is done with its operation?
  - The CPU could do useful work while the peripheral makes the requested operation
  - When the peripheral finishes, the CPUtakes the control of the I/O operation
- This mechanism is denoted as *I/O Controlled by interrupts*

Outline

- Introduction
- Structure and functions of the I/O System
- Programmed I/O
- Exceptions
- Interrupt-based I/O

Exceptions

- It is an unexpected event which causes a change in the normal flow of instructions in a program.



Exception Types

- Hardware exceptions
  - Internal: produced by the CPU (division by zero, overflow, illegal instruction, illegal address, negative square root, etc.)
  - External: produced by I/O devices (Interrupts)
- Software exceptions (trap, swi): produced by the execution of CPU instructions. Used by the operating systems to gain control of the CPU or provide services.

Flow of control when there is an Exception

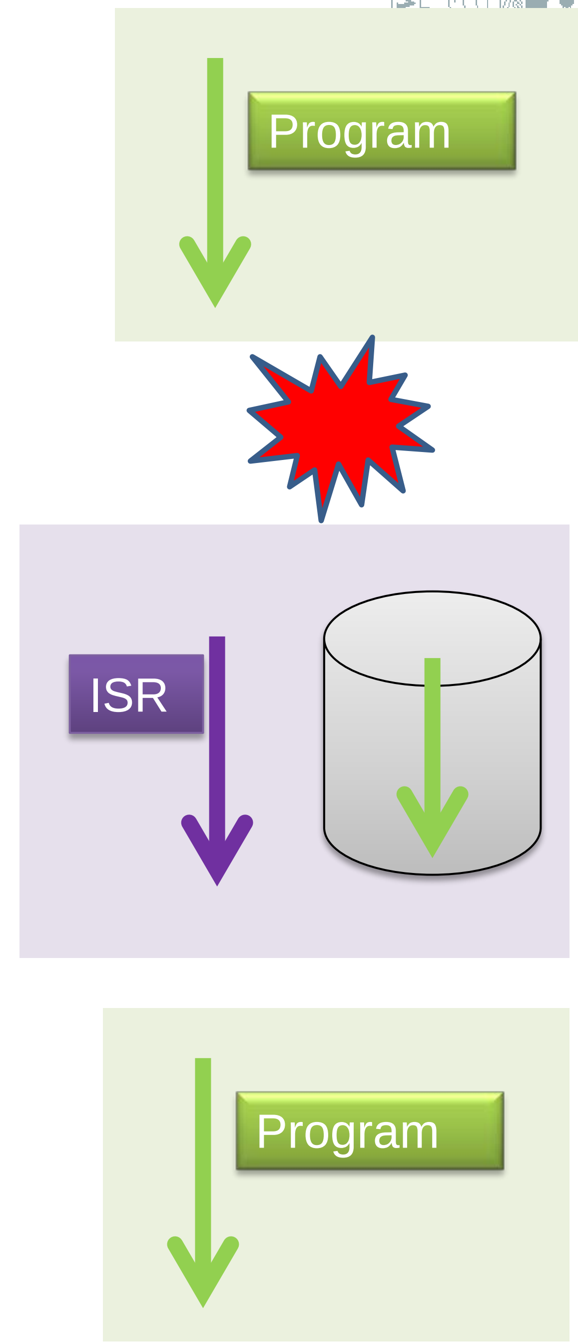
1) The CPU **stops executing** instructions from the user program.

2) **It stores information** so that execution can continue from the same point once the exception is managed

3) Changes ***to a new operating mode*** to manage the exception

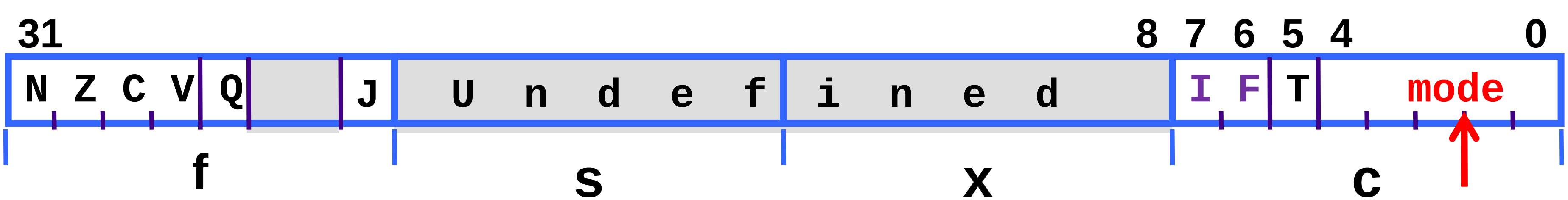
4) It executes an **Interrupt Service Routine** (RTI or *ISR*)

5) After returning from ISR, it **continues execution from the user program (almost always)**



Exceptions and operating modes

- All processors have at least two operating modes
  - **User**: Access to limited resources
  - **Priviledge**: Access to all the resources
- How is mode change performed?
  - From priviledge to user mode: change the mode bits in the status register
  - From user mode to priviledge mode: causing an exception.



ARM: operating modes

User (usr): normal execution

System : priviledge mode for the operating system. Same registers than user mode.

FIQ : Fast interrupts for data transfer

IRQ : General purpose interrupts

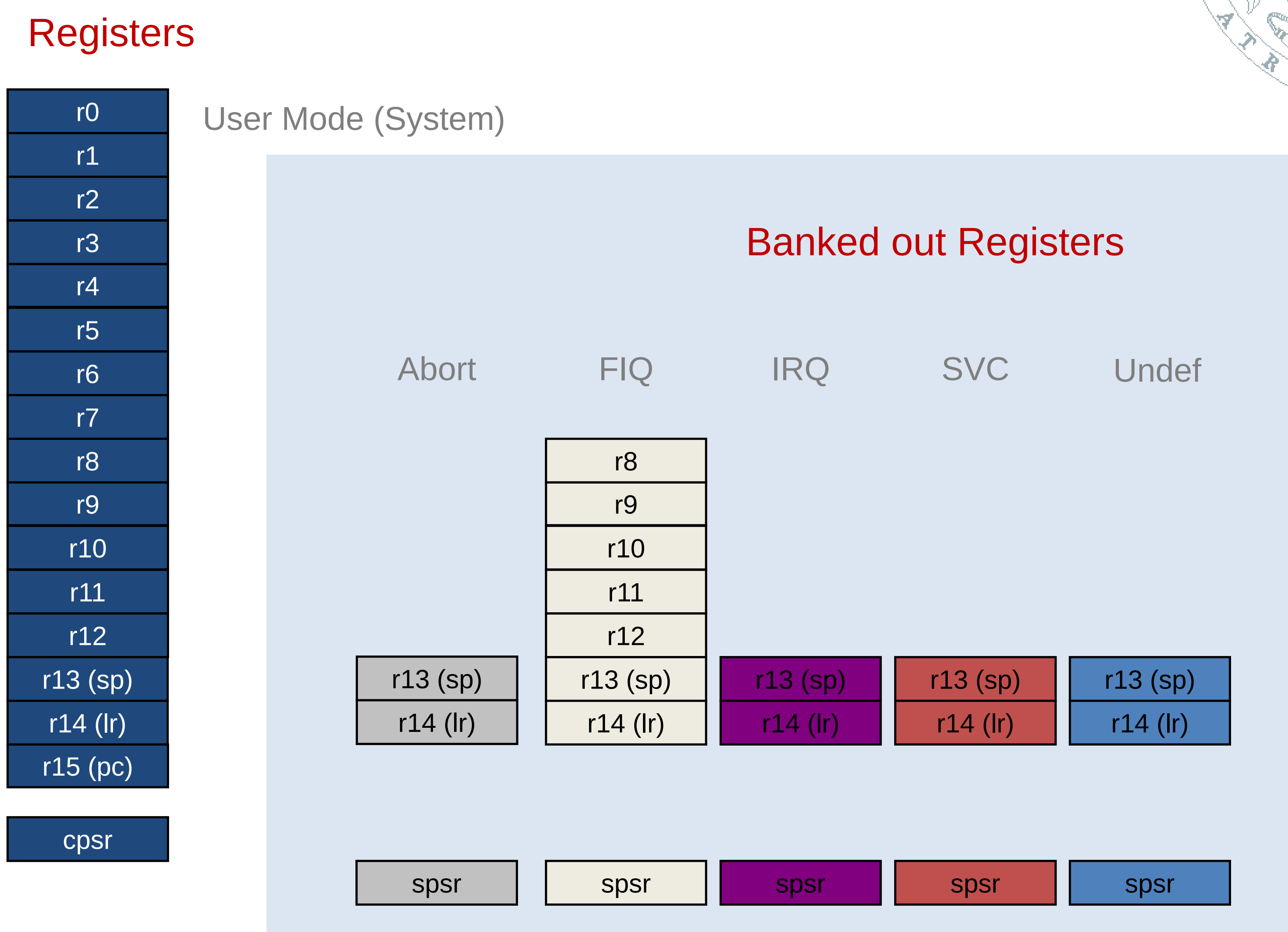
Supervisor (svc): protected mode for the operating system

Abort (abt): used to deal with memory faults (prefetching or data)

Undef (und): used to deal with undefined instructions

S3C44B0X User's Manual. Chapter 2 (Programming Model) pages 2-3 to 2-14. We are not using Thumb state only ARM state.

ARM: operating modes and registers

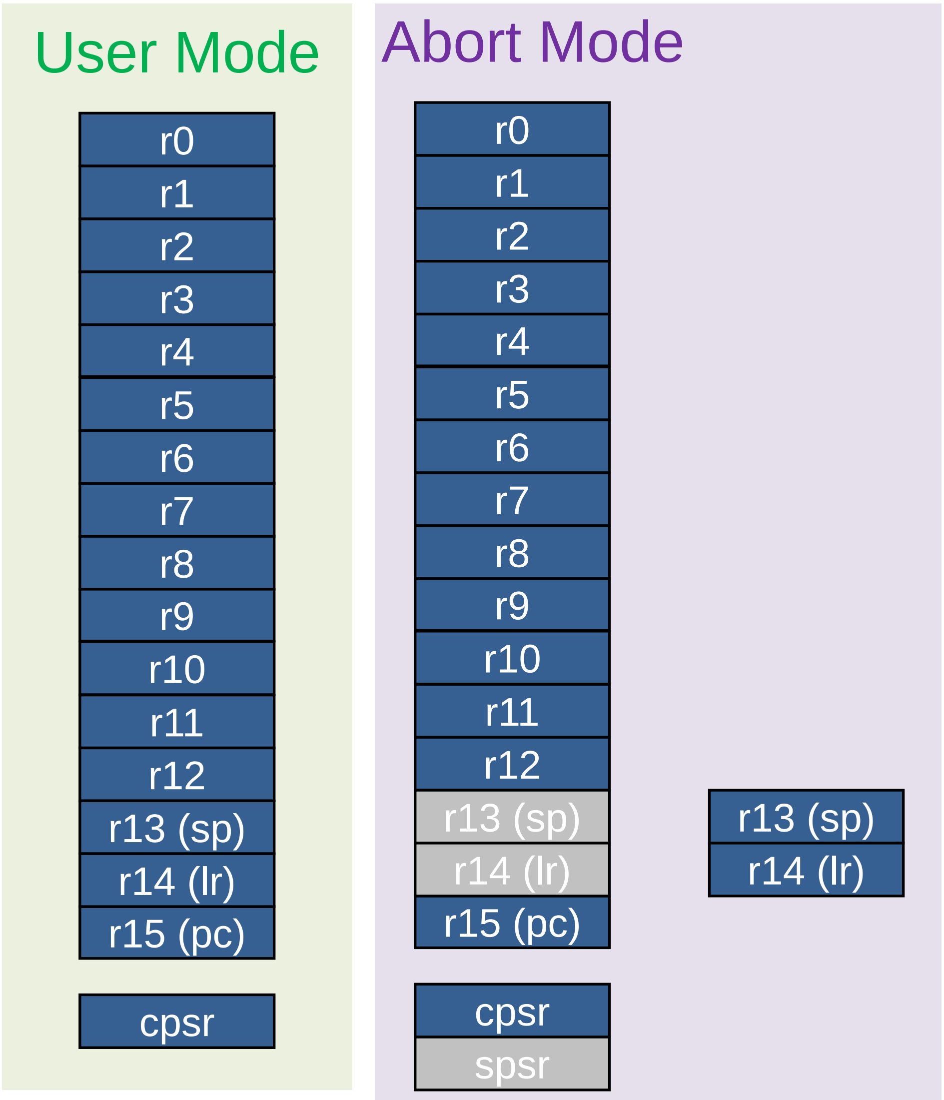


ARM: exception management

When there is an exception:

1) The CPU **stops executing instructions from the program.**
2) It changes to the **operating mode corresponding to the** exception (i.e. Abort)
3) **It stores information** so that execution can go on from the same point later:
   1) **CPSR in SPSR_abort**
   2) **Return address in LR_abort**
   3) **Remaining registers in the stack**

   All the stack pointers have to be initilized on system boot.



ARM: exception management

When there is an exception:

4) It branches to the **Interrupt Service Routine** (*ISR*)
   It is stored in a fixed memory address called Exception Vector.

5) Returning from the ISR **continues with the user program (almost ever)**

Exception	Vector
Reset	0x00
Data Abort	0x10
FIQ	0x1C
IRQ	0x18
Prefetch Abort	0x0C
Undef	0x04
SWI	0x08

Interrupts

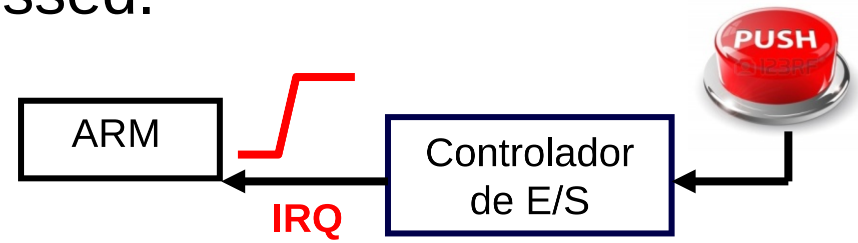
- An interrupt occurs due to an external signal to the processor.
  - It is an asynchronous event, which notifies the processor that some event has happened.
  - It can happen at any moment.
  - Example of an interruption due to an input/output operation:
    - For reading a button, the driver could be programmed to generate an IRQ each time the button is pressed.

Exception vs interrupt

An **exception** is caused by the ***execution of instructions from the program.***
Execution of the same instruction will always generate an exception.
As an example, if we try to write a word in a non aligned address a DATA ABORT will be produced.
The CPU will branch to the ISR_Dabort routine.

```
ldr r0,=0x0a333333
str r1,[r0]  @ Data Abort
```

An **interrupt** is caused by an ***external signal (from the outside).***
It is an asynchronous event that requires processor attention.
An interrupt caused by an I/O operation can be, for example:
The controller of the button can be programmed so that it causes an interrupt (IRQ line) every time the button is pressed.

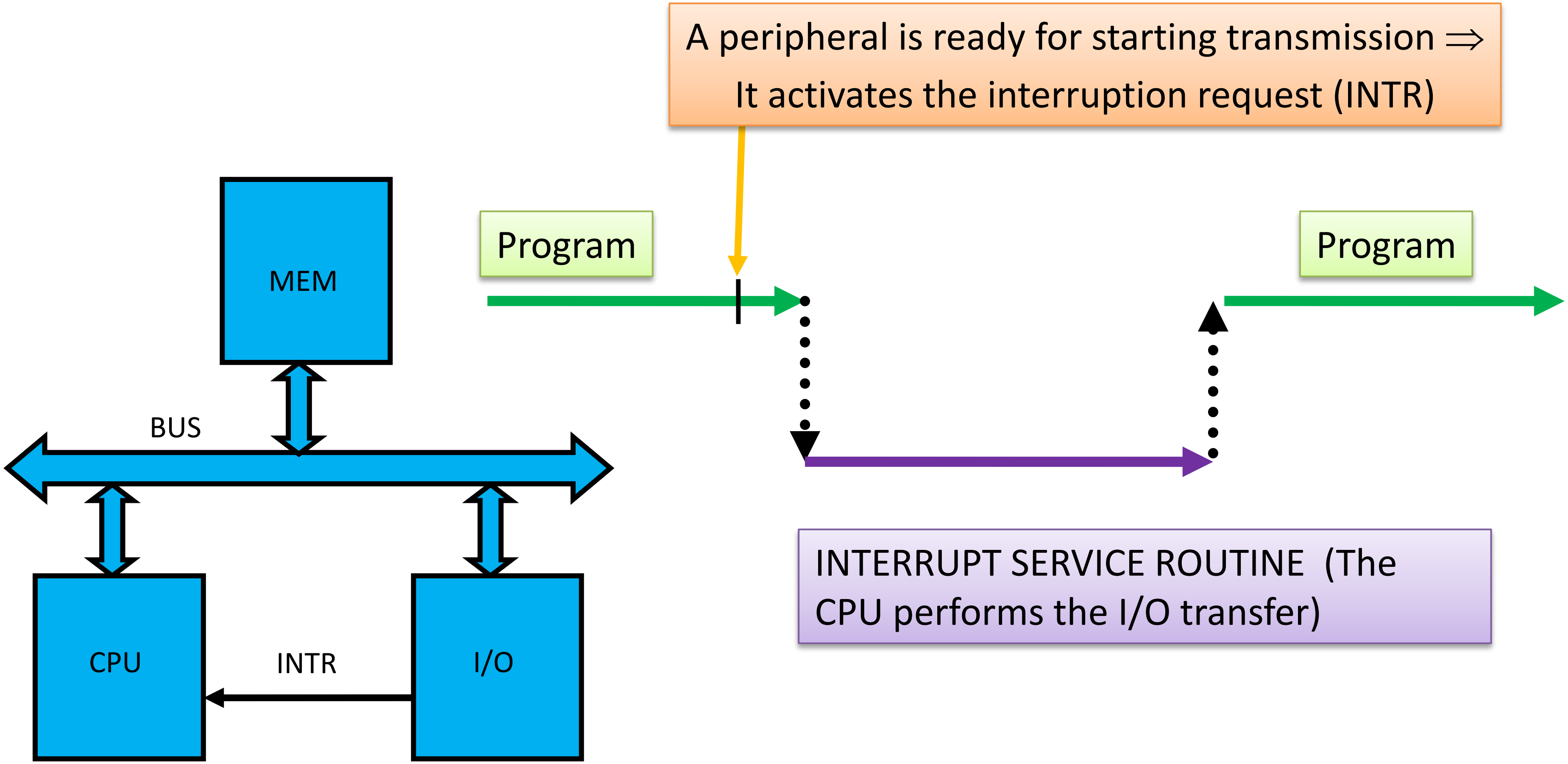


Outline

- Introduction
- Structure and functions of the I/O System
- Programmed I/O
- Exceptions
- Interrupt-based I/O

I/O Controlled by interruptions

- Execution flow when an interrupt occurs:

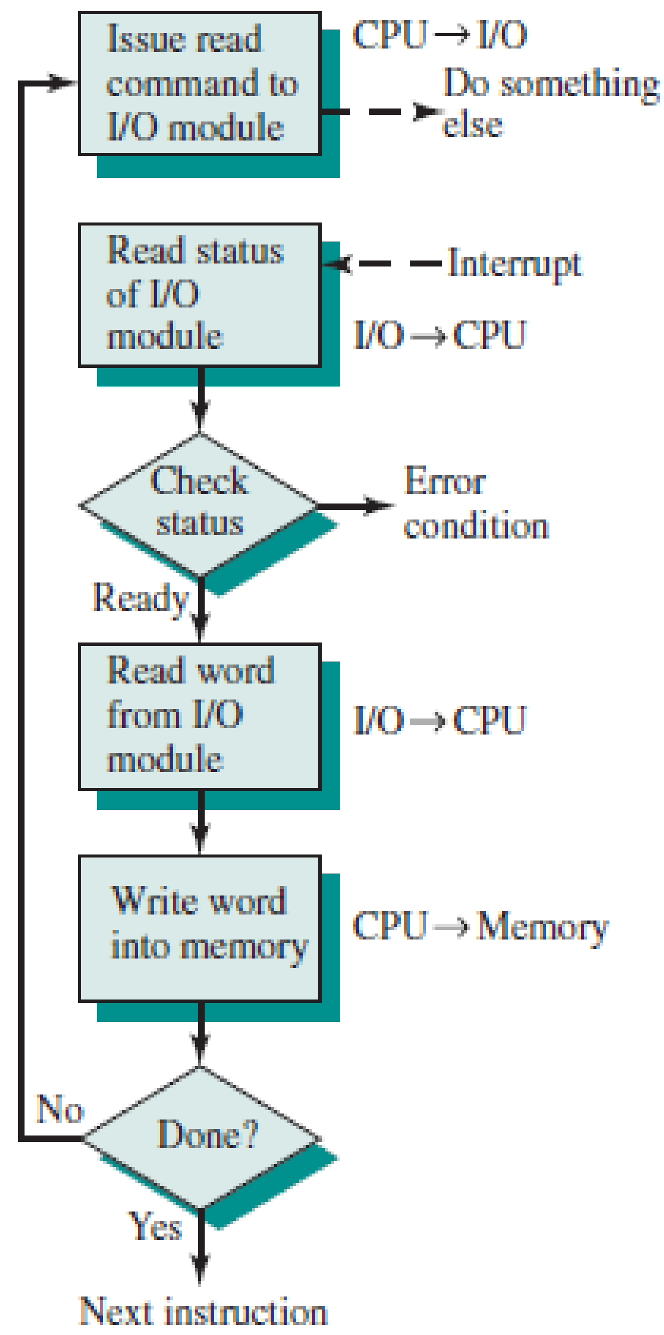


I/O Controlled by interruptions

The CPU starts the I/O operation and proceeds to execute other programs.
➡There is no wait loop

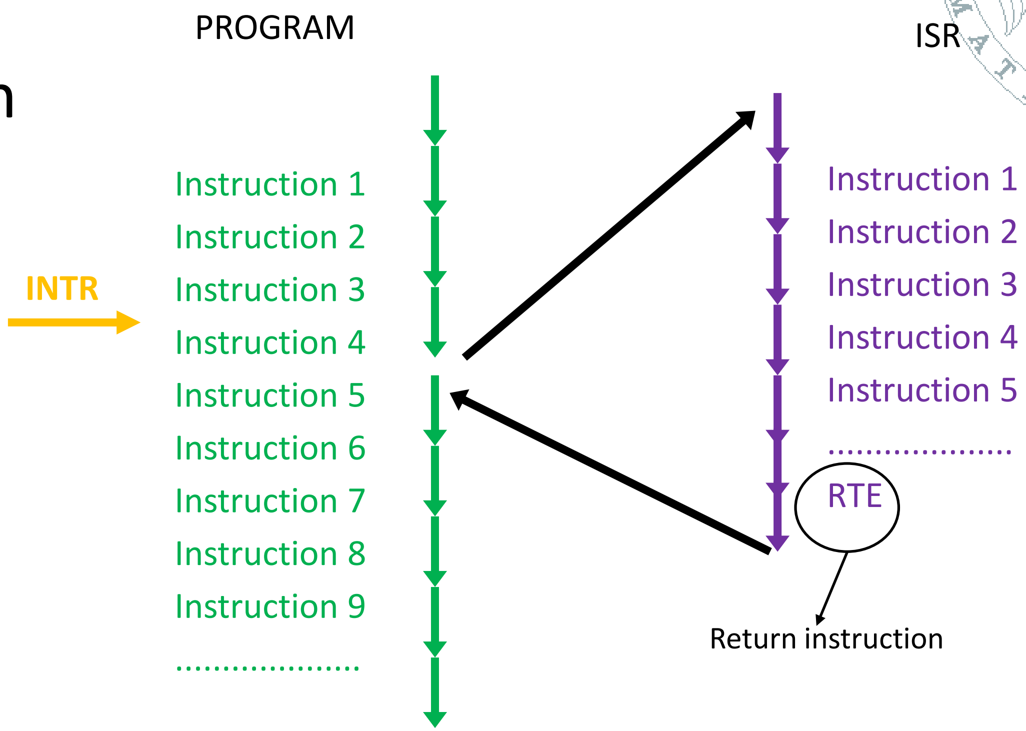
When a peripheral is ready to transmit data, it activates an **INTERRUPT REQUEST LINE**

When the CPU receives an interrupt request signal, it jumps to an INTERRUPT SERVICE ROUTINE (ISR), which is responsible for addressing the peripheral which interrupted the CPU and performing the I/O operation



Interrupt handling routine

- Analogies among a common routine and a IsR
  - Both break the normal execution of a program
  - The system returns to the breaking point after finishing the ISR
    - It must save the PC and registers that it modifies.



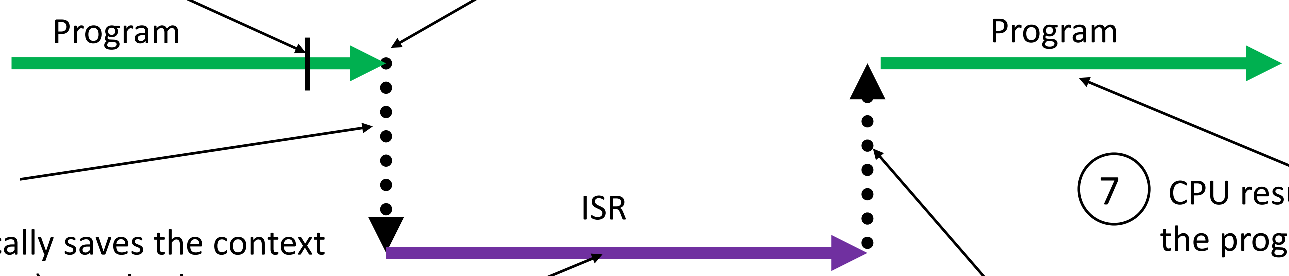
- Differences between a subroutine and ISR
  - In a subroutine, the programmer knows where the sequence is broken
  - An ISR can start at any time, the programmer can NOT control when it happens

Interrupt: Sequence of events

1 The program **enables** the interrupt system

2 The device requests an interrupt (INTR)

3 CPU finishes the execution the of the Instruction in flight and then it **checks for pending interrupts**



4 CPU automatically saves the context (PC and state reg), masks the interrupts and jumps to the ISR

5 CPU executes the ISR, which performs the I/O transfer/operation

6 CPU re-enables the interrupt system, restores the context (PC and state register) and returns to the program

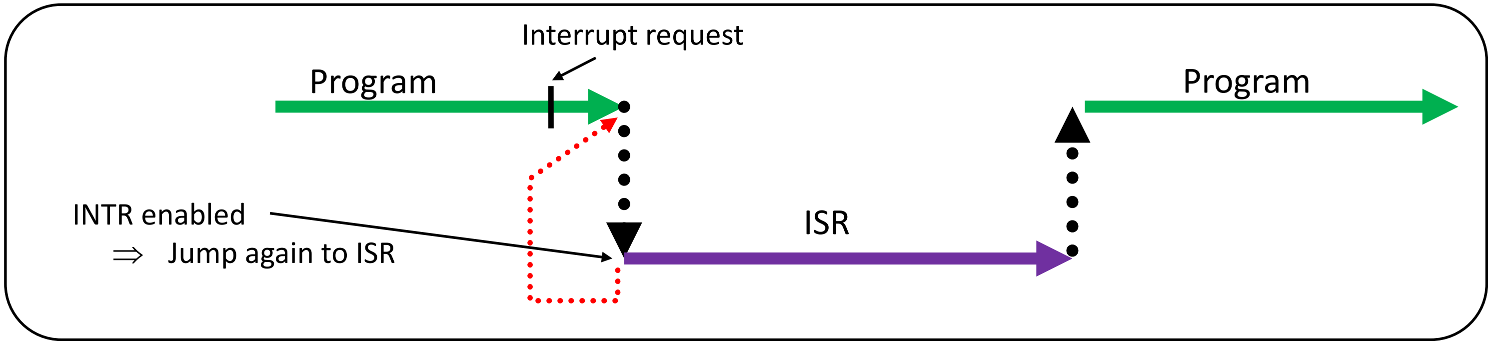
7 CPU resumes the execution of the program

Check of pending interrupt requests

- The CPU checks for pending interrupts (set on the INTR line) at the end of the execution of each Instruction
  - Because:
    - At this moment, only the PC, the status reg and program accessible registers (data and/or address registers) must be saved.
    - Interrupting in the middle of the execution of an instruction would require to save the value of all internal registers in the CPU
      - Instruction Register, registers containing address of data, registers containing the data read from memory, etc.

Inhibition or masking interrupts

- Before jumping to the ISR, it is necessary to mask or disable the interrupts
  - Reason: If the CPU does not mask/disable them, the CPU may enter into an infinite loop
    - The device still has not disabled the request when the CPU enters the ISR
    - If interrupts are enabled, the CPU will detect a pending interruption and jump again and again to the ISR
    - Before finishing the ISR it is **IMPORTANT to confirm** that the device has re-enabled the INTR request line



- Alternatives
  - Disabling globally
    - All interrupts are disabled ⇒ no device may interrupt during the execution of an ISR
  - Selective disabling or masking
    - When there are multiple interrupt levels, interrupts can be disabled only on the same level that requested the interrupt but not on the other levels

1) Identification of the source

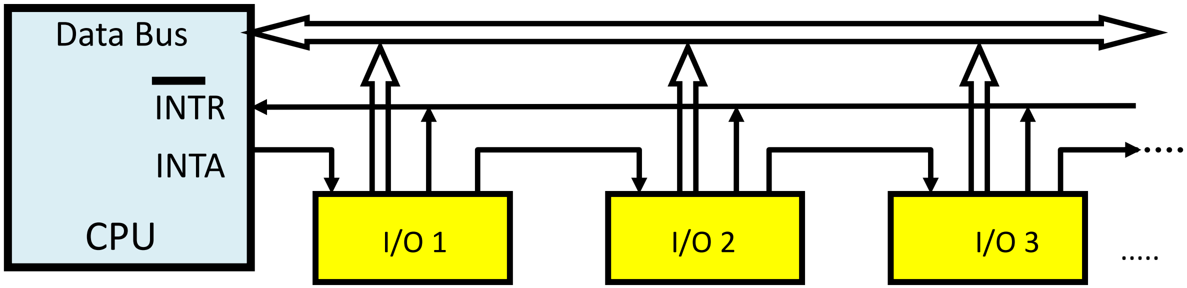
- CPU has a set of interrupt lines
  - Priorities among the different lines
  - Eg ARM: Reset is the highest priority line. FIQ has a higher priority than IRQ.

Non-vectorized Interruptions

- The ISR must identify which device requested the interruption
  - It checks in order the control registers of each HW controller (polling)
  - The order of this polling process determines the priority of the peripherals
  - When the ISR finds the requester peripheral, it clears the flag for that device

Vectorized Interruptions

- The device which has interrupted sends a code or vector number to the CPU from which it can be calculated the starting address of the device ISR
  - When the device receives an acknowledgment signal (INTA), it sends the ID vector number through the data bus
  - From the ID number it is calculated the memory address (array vector) where the starting address of the ISR is stored



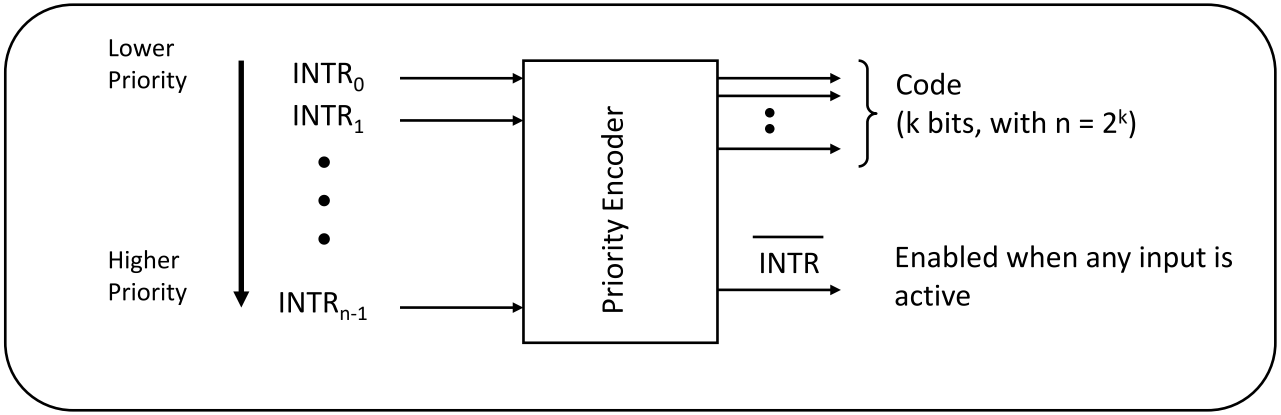
Comparison

- Advantages of Vectorized Interrupts
  - The transmission of INTA is completely done by hardware, thus it is much faster than the survey method
- Disadvantages of Vectorized Interrupts
  - The number of devices that can be identified with this method depends on the number of bits that can be used on the vector number
    - Example: With 4 bits, 16 devices can be identified at most
  - Combined Solution: It is posible to identify by codes of device sets
    - Several devices are grouped within a single ID
    - The ISR identifies (polling) the specific device that initiated the interruption

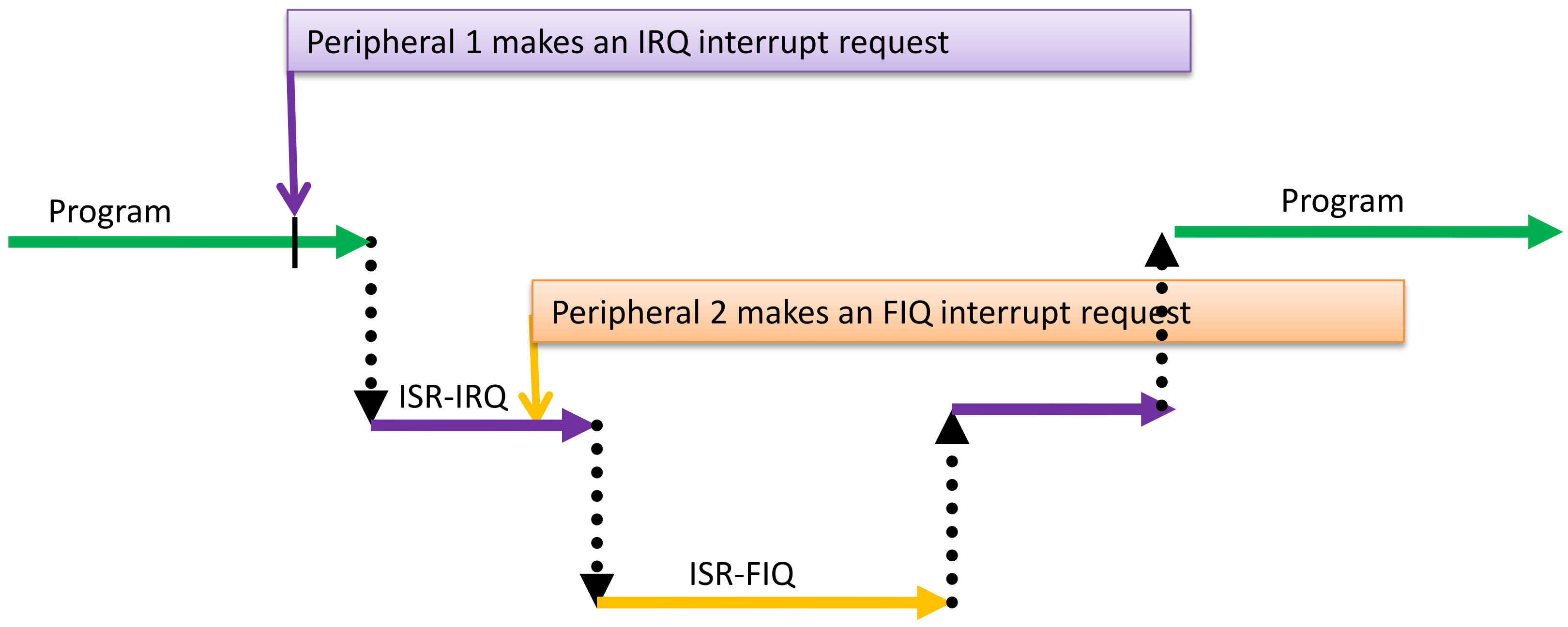
2) Second interrupt while in ISR

Multilevel Interrupts

- There are several interruption lines/levels
- Each one has its own different priority
- Each line can connect one or several devices
- Conflict Resolution (Simultaneous requests on separate lines)
  - Priority encoding ⇒ The line with a higher priority is chosen



Example of nested interrupts



* Explain Lab 3

ec