Embedded Systems CSEN701

Dr. Catherine Elias

Eng. Abdalla Mohamed

Office: C1.211
Mail : abdalla.abdalla@guc.edu.eg

Eng. Mohamed Elshafie

Office: C1.211
Mail : Mohamed.el-shafei@guc.edu.eg

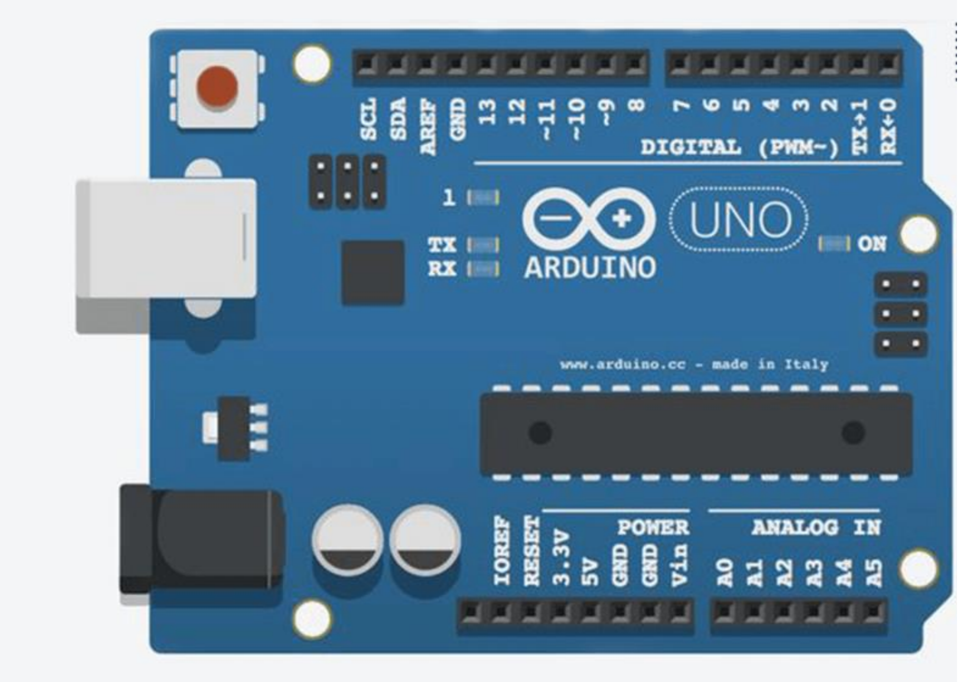
Eng. Maysarah El Tamalawy

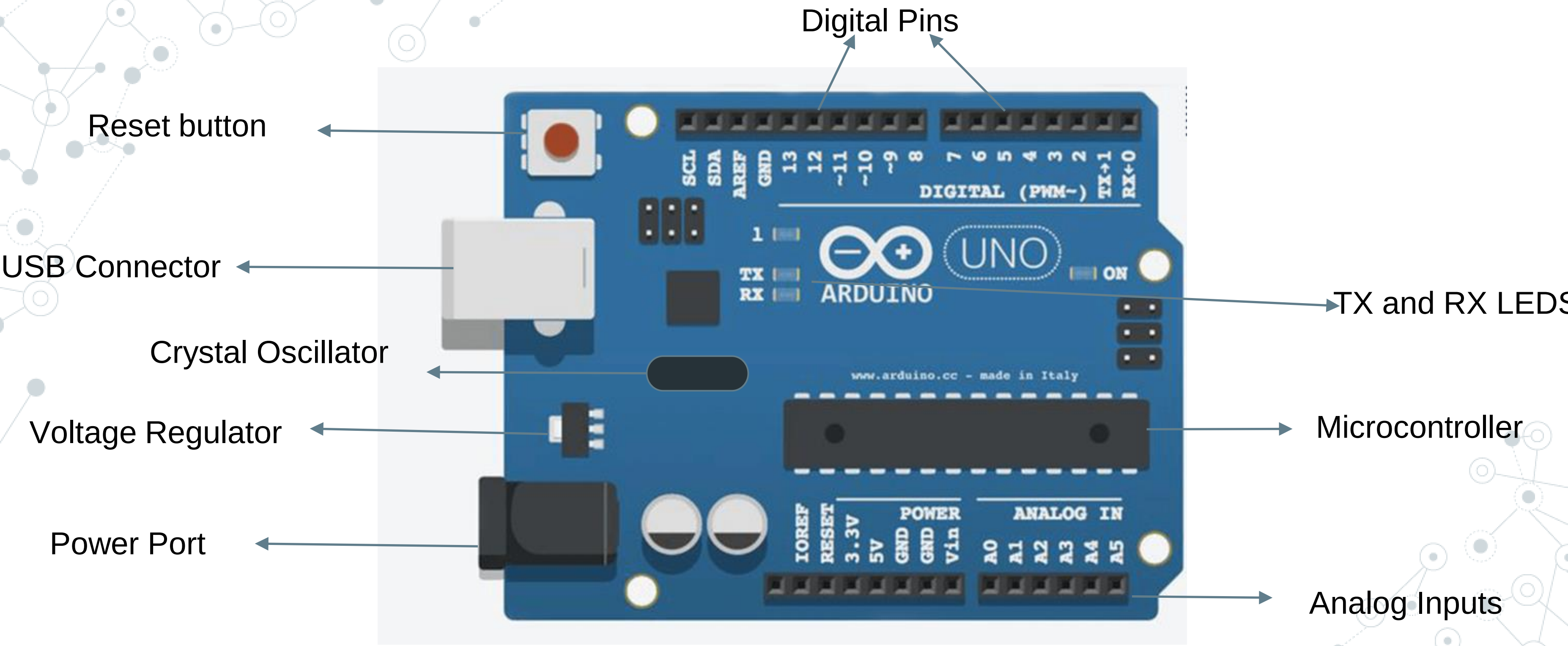
Office: C7.201
Mail : Maysarah.mohamed@guc.edu.eg

Outline :

- Recap.
- **Arduino Components.**
- Harvard Architecture vs Von Neuman Architecture.
- AVR architecture & preipherals .
- GPIO in AVR
- Bitwise operations
- GPIO in ARM

Can you name the components?

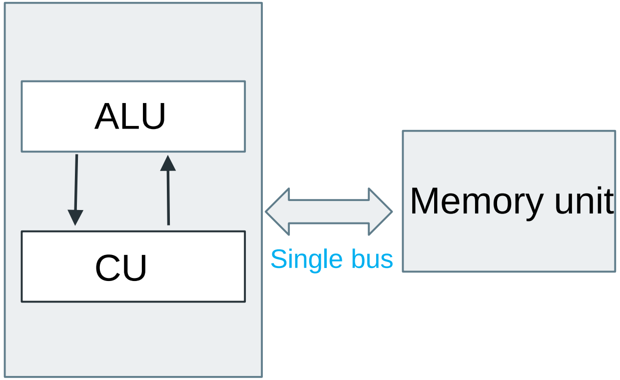



Digital Pins
Reset button
USB Connector
Crystal Oscillator
Voltage Regulator
Power Port
TX and RX LEDS
Microcontroller
Analog Inputs
SCL SDA AREF GND 13 12 ~11 ~10 ~9 8 7 ~6 ~5 4 ~3 2 TX→1 RX←0
DIGITAL (PWM~)
UNO
ARDUINO
ON
TX
RX
www.arduino.cc - made in Italy
IOREF RESET 3.3V 5V GND GND Vin
POWER
ANALOG IN
A0 A1 A2 A3 A4 A5

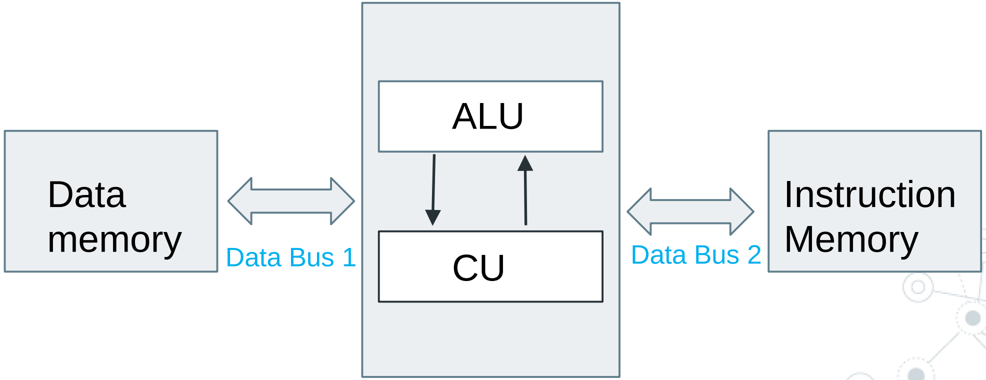
Outline :

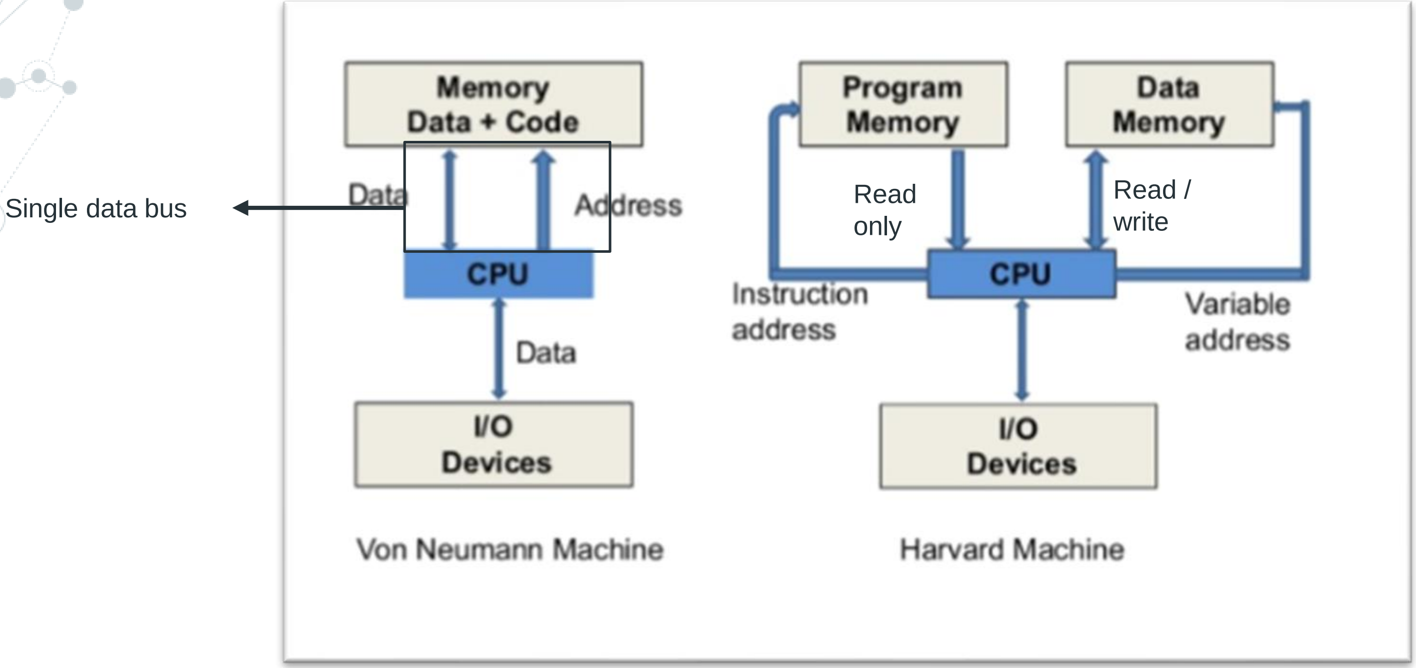
- Recap.
- Arduino Components.
- **Harvard Architecture vs Von Neuman Architecture.**
- AVR architecture & preipherals .
- GPIO in AVR
- Bitwise operations
- GPIO in ARM

Von Neuman



Harvard




Single data bus
Memory
Data + Code
Data
Address
CPU
Data
I/O
Devices
Von Neumann Machine
Program
Memory
Data
Memory
Read
only
Read /
write
Instruction
address
CPU
Variable
address
I/O
Devices
Harvard Machine

Von Neuman

- Instruction fetch and data operation cannot happen at the time.
- The instructions and data are in the same place so they use a common bus.

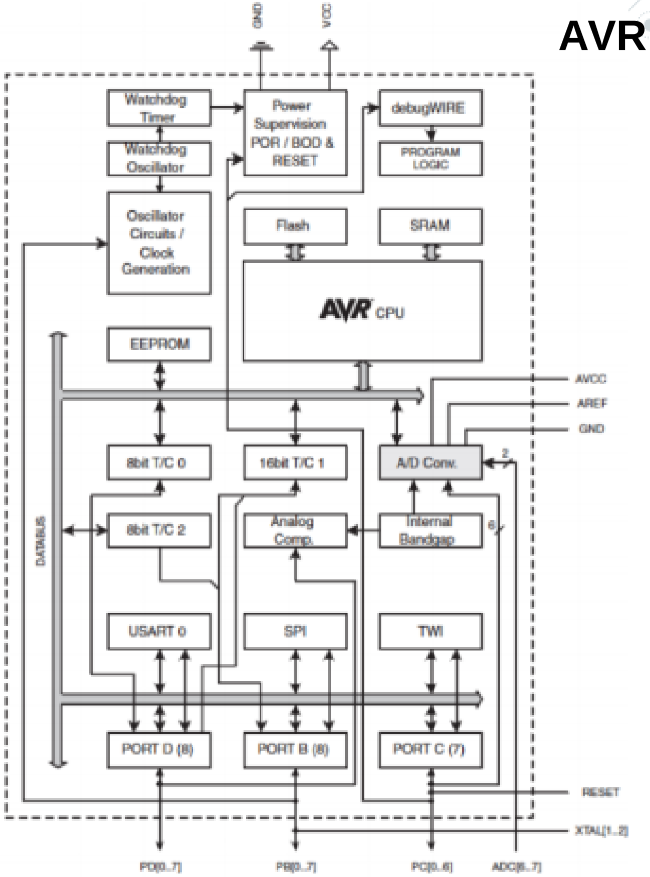
Harvard

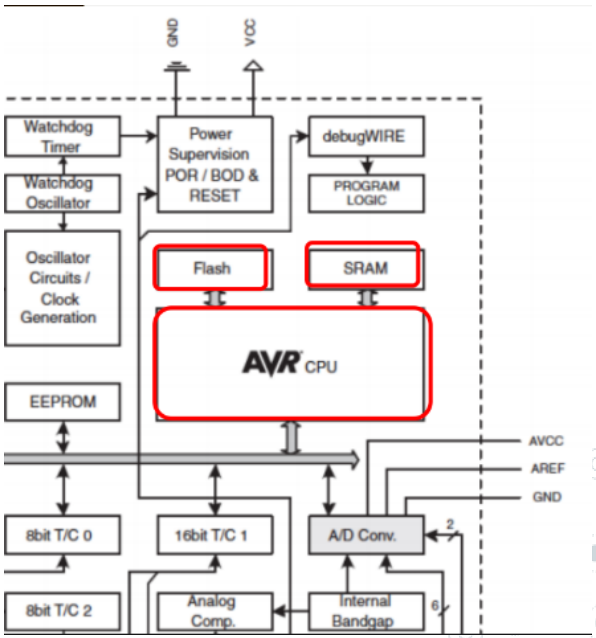
- The instructions are in separate memory.
- There are two buses one between control unit and instruction memory and one between data memory and control unit.

Outline :

- Recap.
- Arduino Components.
- Harvard Architecture vs Von Neuman Architecture.
- **AVR architecture & preipherals .**
- GPIO in AVR
- Bitwise operations
- GPIO in ARM

AVR architecture from Data Sheet





AVR architecture

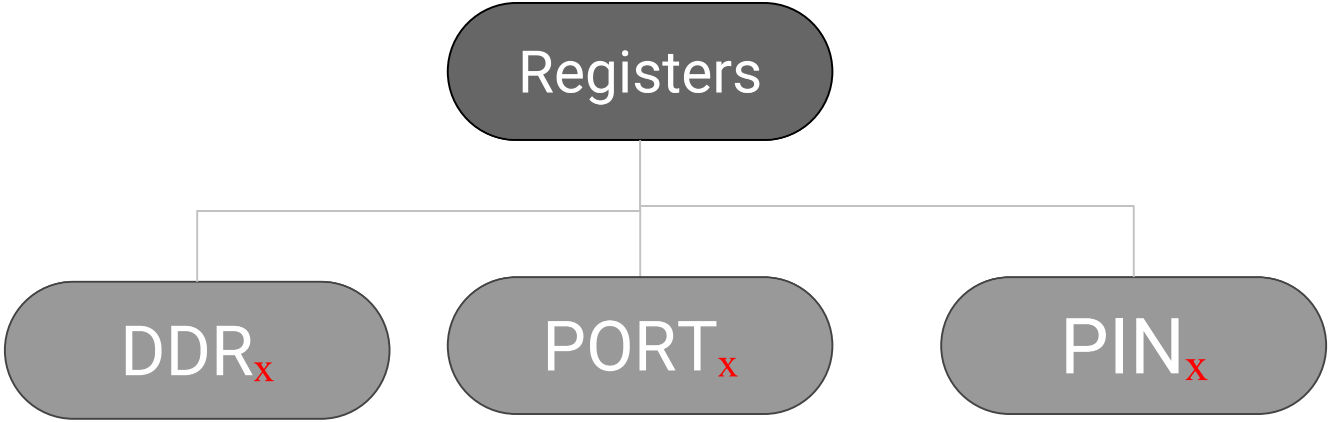
- **Simple Instruction Set:** AVR uses a small, well-defined set of instructions, each of which performs a single operation. This simplicity leads to efficient and fast execution so it’s based on RISC architecture.
- **Separate Program and Data Memory:** AVR architecture features distinct memory spaces for program instructions (Flash memory) and data (SRAM). This separation allows for simultaneous access to program and data, improving performance.
- **Separate Buses:** It employs separate buses for program memory and data memory, enabling parallel fetching of instructions and data.

Outline :

- Recap.
- Arduino Components.
- Harvard Architecture vs Von Neuman Architecture.
- AVR architecture & preipherals .
- **GPIO in AVR**
- Bitwise operations
- GPIO in ARM

Ports in Arduino AVR

- A port on the Arduino is a group of pins, each consists of 3 types of registers that control the functionality of this port. These registers determine the setup of the pins.



* x is the name of the port (A ,B, C or D) *

DDR register

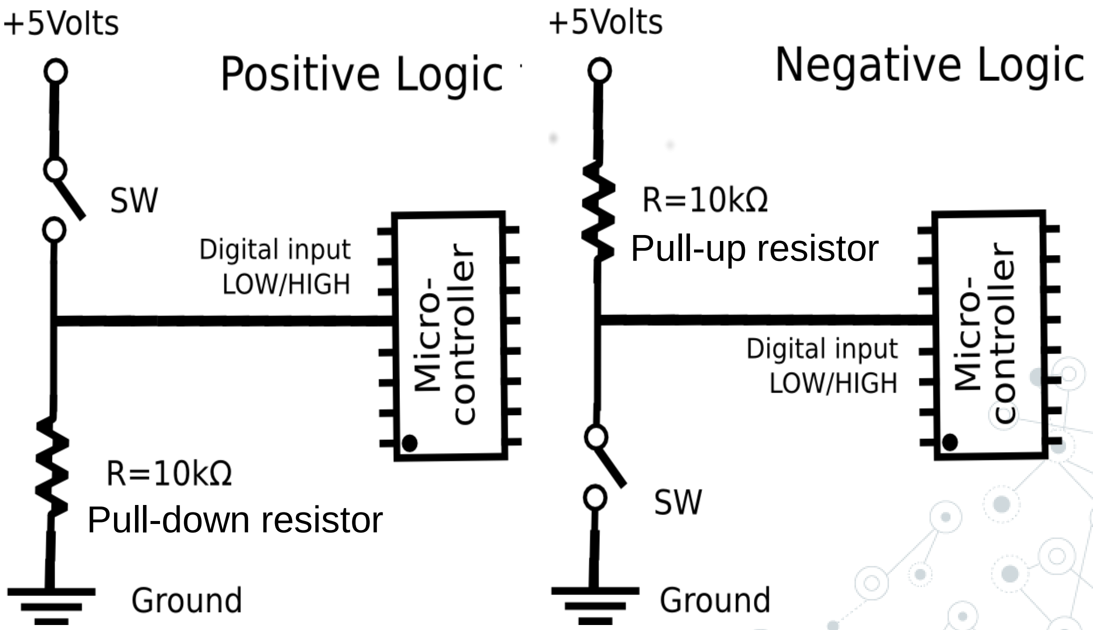
- DDR register is the register the determine the data direction for a group of pins .
- You can select whether a certain pin is input or output by changing the value of the corresponding bit in the DDR register .
- Ex:- if we want to set pin 5 in port B as input , then this bit is set to 0 , if we want to set it to output then it is set to 1
- So if all pins are set to input and only pin 5 is set to output in port B, the value of the DDR_B is 00100000 = 0x20

Bit	7	6	5	4	3	2	1	0
	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Bits 7:0 – DDRBn: Port B Data Direction [n = 7:0]

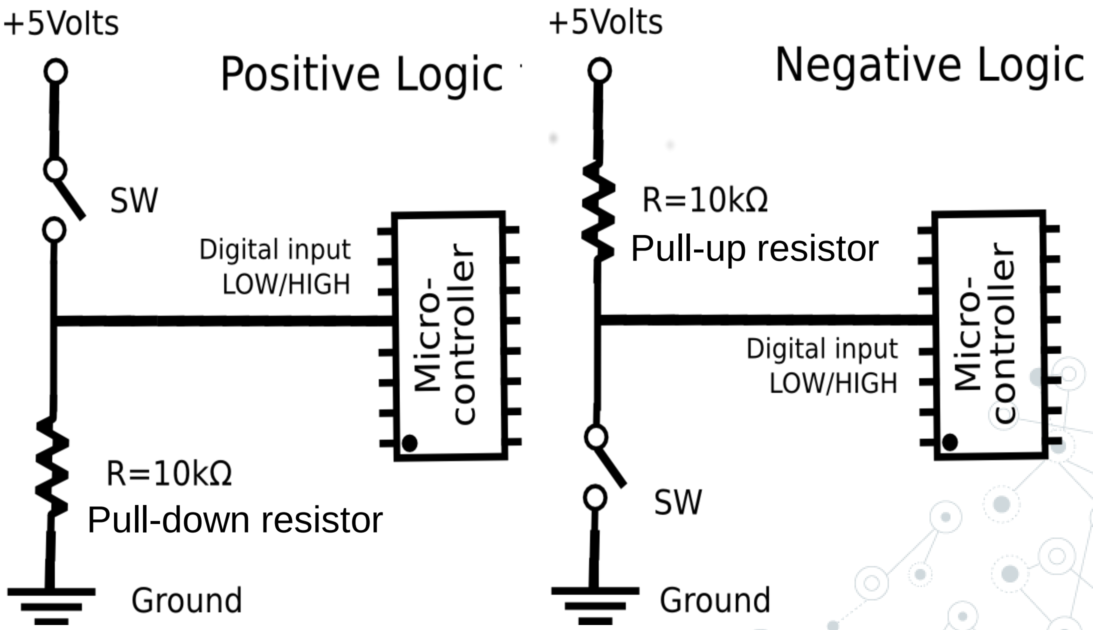
But first we have to understand different digital logics …

- Positive logic is the **default logic** , the input is initially low until the button is ON /activated so it deliver High voltage (5v) to the Microcontroller/LED when pressed.
- Sensors/Pins working with positive logic are called Active-HIGH.
- Pull-down resistors are associated with positive logic to **pull** the initial pin value **down** to GND thus preventing the floating of the input value.
- Button Is OFF/open -- Input = GND (0V)
- Button is ON/Closed -- Input = VCC (5V)



But first we have to understand different digital logics …

- Negative Logic connection operates in an opposite manner, the input is initially high until the button is On/activated it delivers GND (0V) to the Microcontroller/LED.
- Sensors/Pins working with negative logic are called Active-Low.
- Pull-up resistors are associated with negative logic to **pull** the initial pin value **up** to High voltage (5V) thus preventing the floating of the input value .
- Button Is OFF/open -- Input = VCC (5V)
- Button is ON/Closed -- Input = GND (0V)



PORT register

- PORT registers have 2 functionalities :
  - If the bit is set to output in DDR register :
    - If a bit in the register is set to 1 , then the corresponding pin is driven HIGH
    - If a bit in the register is set to 0 , then the corresponding pin is driven LOW

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PORT register

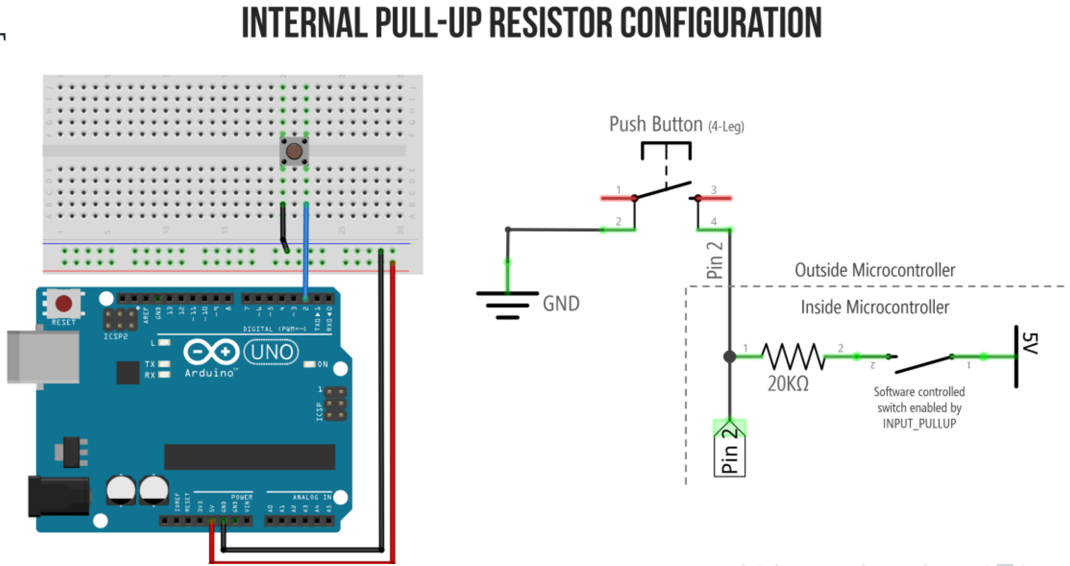
- PORT registers have 2 functionalities :
  - If the bit is set to input in DDR register :
    - If a bit in the PORT register is set to 1 , then the internal pull up resistor is activated.
    - If a bit in the PORT register is set to 0 , then the pin is tri-stated (default input pin).

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PORT register

- If the bit is set to input 0 in DDR register and the same bit in the PORT register is set to 1 , then the internal pull up resistor is activated at the corresponding pin .

- The Initial Value of this pin will be High in case of no input signal .



Dr.Catherine Elias
Eng. Abdalla Mohamed Eng. Yousef Abdelshafy Eng. Mina Wasfy

PIN register

- PIN registers are used to read the input data from a port pin
- When the pin is set as input in the DDR, and the pull-up resistor is enabled (in the PORT register) then the bit will indicate the state of the signal at the pin.

Bit	7	6	5	4	3	2	1	0	
0x03 (0x23)	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Outline :

- Recap.
- Arduino Components.
- Harvard Architecture vs Von Neuman Architecture.
- AVR architecture & preipherals .
- GPIO in AVR
- **Bitwise operations**
- GPIO in ARM

Bitwise operations in C

- Bitwise operations are used to directly manipulate registers in embedded C .
- Bitwise operations are used to :
  - Set bits (LOGIC HIGH)
  - Clear bits (LOGIC LOW)
  - Toggle bits (XORING)
  - Shift bits

Operator	Description
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right
~	one's complement

SET BITS

DDRD = 0b00000000 ; DDRD = 0x00 ; (hexadecimal)

Set bits 0 and 2 as outputs :

DDRD = 0b00000101 ; DDRD = 5 ; (PORT Assignment)

DDRD |= 5 ; / (*DDRD = DDRD | 0b00000101*)

DDRD		0 0 0 0 0 0 0 0
OR	|	
5		0 0 0 0 0 1 0 1

Hint 1 : any bit ORED | with 0 is unchanged

Hint 2 : any bit ORED | with 1 is SET

DDRD 0 0 0 0 0 1 0 1 (bit assignment)

Bit	7	6	5	4	3	2	1	0
	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Bits 7:0 – DDRBn: Port B Data Direction [n = 7:0]

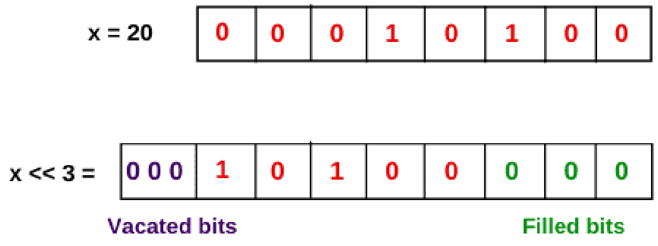


Fig: Shifting bits towards left 3 times

bit 1	bit 2	&	||	^	~ bit 1	~ bit 2
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

Dr.Catherine Elias
Eng. Abdalla Mohamed Eng. Yousef Abdelshafy Eng. Mina Wasfy

SET BITS

DDRD = 0b00000000 ; DDRD = 0x00 ; (hexadecimal)

Set bits 0 and 2 as outputs :

DDRD |= (1<<0) |(1<<2) ; (1<<bit number)

DDRD	OR	0 0 0 0 0 0 0 0
(1<<0)	|	0 0 0 0 0 0 0 1
(1<<2)	|	0 0 0 0 0 1 0 0
DDRD =		0 0 0 0 0 1 0 1

Hint : any bit ORED | with 0 is unchanged

Hint 2 : any bit ORED | with 1 is SET

only targeted bits are assigned and set

Bit	7	6	5	4	3	2	1	0
	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Bits 7:0 – DDRBn: Port B Data Direction [n = 7:0]

DDRD = 0b00000101 ;

DDRD |=5 ;

DDRD |= 0x05 ;

DDRD |= (1<<0)|(1<<2);

CLEAR BITS

DDRD = 0b00000101 ; DDRD = 0x05 ; (hexadecimal)

Clear bit 2 to change it to input pin :

DDRD = 0b00000001 ; DDRD = 1 ; (PORT Assignment)
DDRD & = 0b11111011 ;

0 0 0 0 0 1 0 1
AND &
1 1 1 1 1 0 1 1

DDRD 0 0 0 0 0 0 0 1 (bit assignment)

Hint 1 : any bit ANDED | with 1 is unchanged

Hint 2 : any bit ANDED | with 0 is cleared

Bit	7	6	5	4	3	2	1	0
	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Bits 7:0 – DDRBn: Port B Data Direction [n = 7:0]

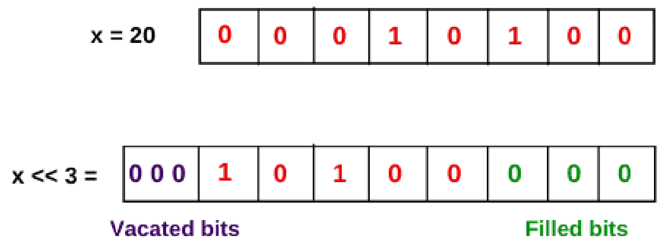


Fig: Shifting bits towards left 3 times

bit 1	bit 2	&	||	^	~ bit 1	~ bit 2
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

CLEAR BITS

DDRD = 0b00000101 ; DDRD = 0x05 ; (hexadecimal)

Clear bit 2 to change it to input pin :

DDRD & = 0b11111011 ; 0b11111011 == ~(00000100)

DDRD &= ~(0b00000100)

DDRD &= ~(1<<2) ; (1<<bit number)

Hint 1 : any bit ANDED | with 1 is unchanged

Hint 2 : any bit ANDED | with 0 is cleared

DDRD		0 0 0 0 0 1 0 1
AND	&	
~(1<<2)		1 1 1 1 1 0 1 1
DDRD		0 0 0 0 0 0 0 1

only targeted bits are assigned and cleared

Bit	7	6	5	4	3	2	1	0
	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Bits 7:0 – DDRBn: Port B Data Direction [n = 7:0]

DDRD = 0b00000001 ;

DDRD &=0b11111011 ;

DDRD &= ~(0b00000100) ;

DDRD &= ~(1<<2);

Bitwise operations in a Nutshell

- SET BIT : REG | = (1<<bit_number) | (1<<bit_number) …..
- CLEAR BIT : REG &= ~(1<<bit_number) & ~(1<<bit_number)….
- TOGGLE BIT : REG ^=(1<<bit_number)^(1<<bit_number)…..

HINTS :

- ✓ **any bit ORED | with 0 is unchanged**
- ✓ **any bit ORED | with 1 is SET**
- ✓ **any bit ANDED | with 1 is unchanged**
- ✓ **any bit ANDED | with 0 is cleared**
- ✓ **any bit XORED | with 1 is Toggled**

Outline :

- Recap.
- Arduino Components.
- Harvard Architecture vs Von Neuman Architecture.
- AVR architecture & preipherals .
- GPIO in AVR
- Bitwise operations
- **GPIO in ARM**

EX1

Implement an embedded C code to :

1. Connect push button A to pin 5 in PORT C (Positive Logic)
2. Connect push button B to pin 3 in PORT B (Negative Logic)
3. Apply the Internal pullup resistor to pin3 PORT B
4. Configure PIN 2 in PORTD as output
5. Connect Pin 2 to RED LED Pin5-- RED (Hardware step)
6. Turn on Red LED when A is pressed
7. Turn off the RED LED when B is pressed .

EX1

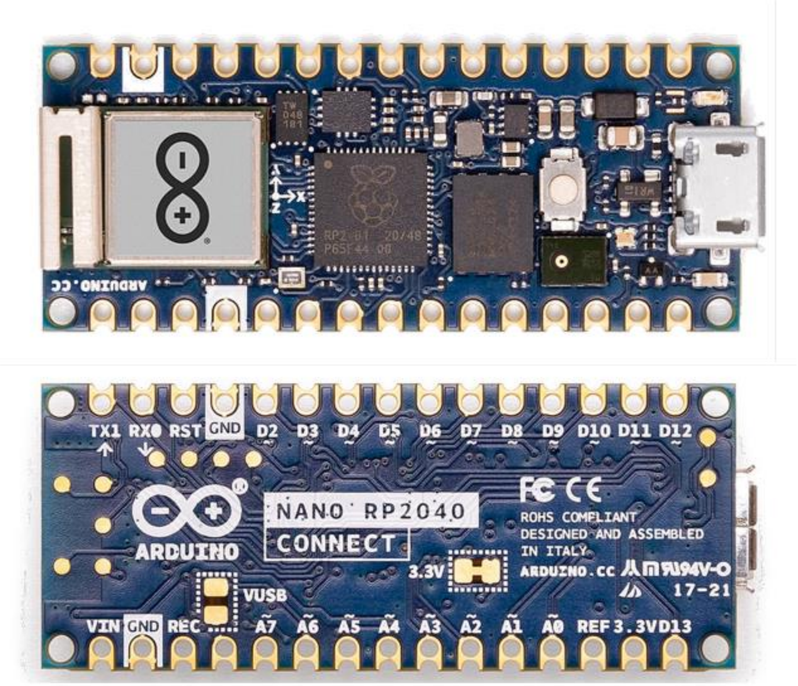
```
#include <avr/io.h>
int main (void){
 DDRB = 0x00 ; DDRD = 0x00 ; DDRC = 0x00 ; PORTC = 0x00 ; PORTB=0x00; PORTD=0x00 ;   // initialize the registers
 DDRC &=~(1<<5)  ;  // configure pin5 as input  in PORTA ( pushbutton A is connected to PIN 5 in positive Logic )
 DDRB &=~(1<<3)  ;  // configure pin3 as input in PORTB ( pushbutton B is connected to PIN 3 in negative Logic )
 PORTB |= (1<<3) ; // set bit 3 to HIGH to activate the internal pull-up resistor at pin 3
 DDRD   |= (1<<2) ; // configure pin 2 as an output pin  at PORTD
 while (1 ) {
 if ( PINC & ( 1<<5) ) {   //  HINT : (PINC & 0b00100000) is only true when bit 5 at PINC is 1 (pushbutton A is pressed +ve L)
 PORTD |= (1<<2) ;  // set the output to HIGH  to TURN ON the LED
 }
 if ( !(PINB & (1<<3) ) {   // (PINB & (1<<3)) is true when Bit 3 is ON ( not pressed ) so it will be false (!) if  pressed ( -ve L)
       PORTD &= ~(1<<2)  ; // set the output to LOW  by clearing bit 2        // pushbutton B is connected in negative logic
   } }                                    }
```

Outline :

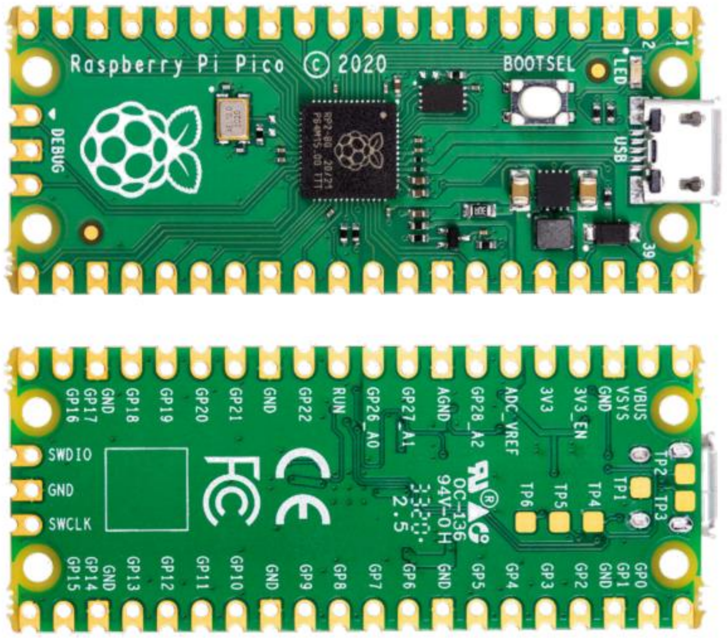
- Recap.
- Arduino Components.
- Harvard Architecture vs Von Neuman Architecture.
- AVR architecture & preipherals .
- GPIO in AVR
- Bitwise operations
- **GPIO in ARM**

Microcontroller Top View

The Arduino RP2040: *Views*

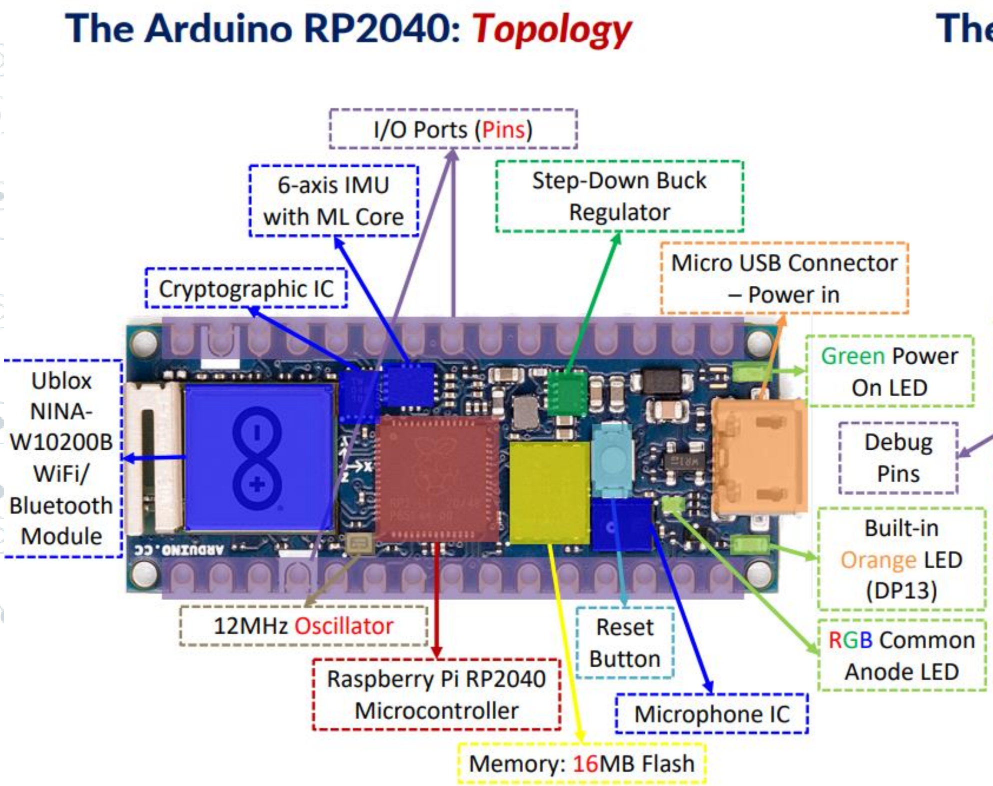


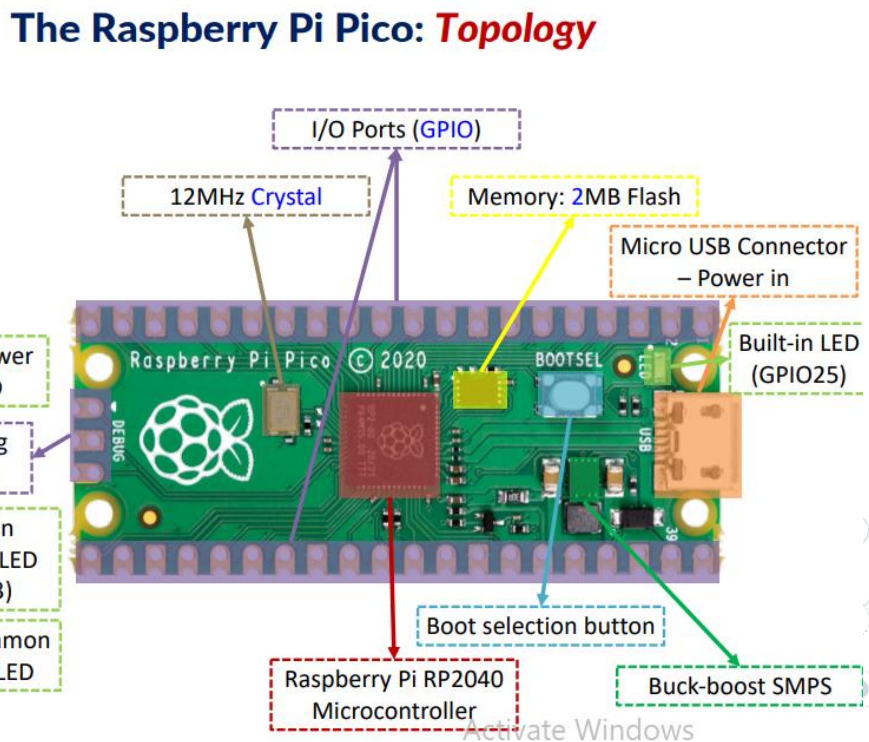
The Raspberry Pi Pico: *Views*



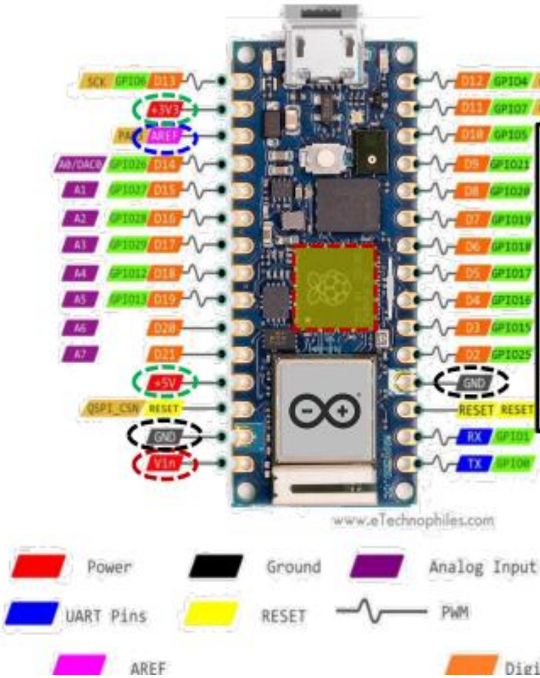
Dr.Catherine Elias
Eng. Abdalla Mohamed Eng. Yousef Abdelshafy Eng. Mina Wasfy

Microcontroller Top View

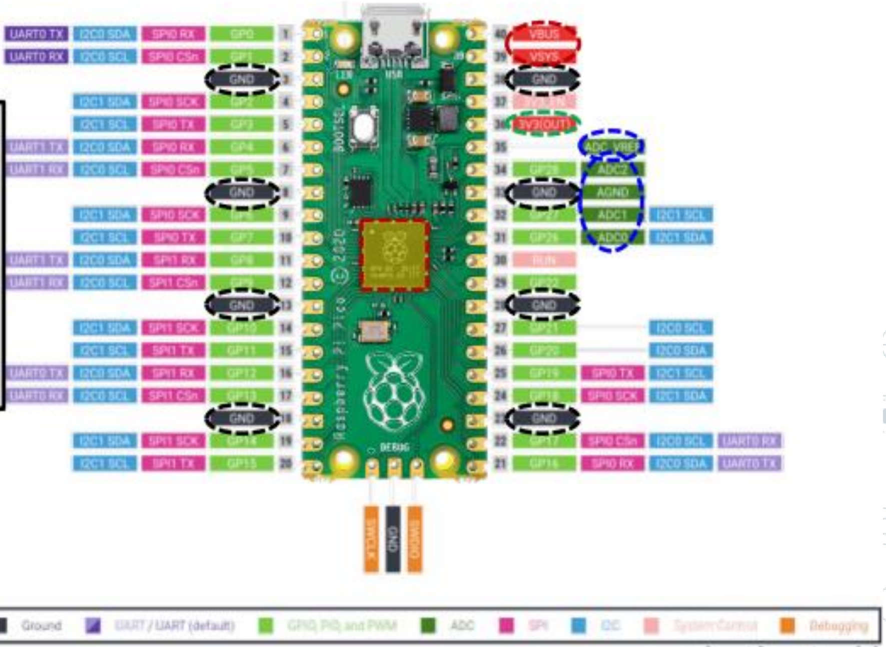




The Arduino RP2040: *Pinout*



The Raspberry Pi Pico: *Pinout*



- **Power In:**
  - Arduino Vin → 5V
  - Pi Pico Vin → 3.3V
- **GND**
- **Power Out:**
  - Arduino → 3.3V & 5V
  - Pi Pico → 3.3V
- **ADC & AREF**
- **Communication**

Note : Extra information Don't Memorize

The RP2040 GPIO Hardware Registers :

- There are 28 programmable GPIO pins on the Pico. There are 40 pins, but the others are ground, power and a couple of specialized pins .
- The registers shown in the diagram are used to control the input/output specifications in the DIO peripheral.
- DIO peripheral contains 32-bit hardware register which is mapped to 32-bits of memory in the RP2040's address space
- Check the datasheet for each register description.

Register	Address
gpio_in	0xd0000004
gpio_hi_in	0xd0000008
gpio_out	0xd0000010
gpio_set	0xd0000014
gpio_clr	0xd0000018
gpio_togl	0xd000001c
gpio_oe	0xd0000020
gpio_oe_set	0xd0000024
gpio_oe_clr	0xd0000028
gpio_togl	0xd000002c
gpio_hi_out	0xd0000030
gpio_hi_set	0xd0000034
gpio_hi_clr	0xd0000038
gpio_hi_togl	0xd000003c
gpio_hi_oe	0xd0000040
gpio_hi_oe_set	0xd0000044
gpio_hi_oe_clr	0xd0000048
gpio_hi_oe_togl	0xd000004c

Note : Extra information Don't Memorize

- **GPIO_OE**: Output enable register (1 for output, 0 for input). 32-bit register , each bit maps to a corresponding GPIO PIN resembling the **DDRX** .

- **GPIO_CTRL**: There is a 32-bit GPIO control register for each pin , separated by 8 bytes in the memory space , it is used to configure the function of the pins.

- **GPIO_OUT**: Output value register. Resembles **PORTX** in output pins, used to enter the output needed in the pin in its corresponding bit .

- **GPIO_IN**: Input value register. Used to read the Value of the input pin using its corresponding bit resembling PINX.

gpio.c

```c
#include <stdint.h>

#define GPIO_BASE       0x40014000  // Base address for GPIO registers
#define GPIO_OE         (*(volatile uint32_t *)(GPIO_BASE + 0x20))  // Output Enable Register
#define GPIO_OUT        (*(volatile uint32_t *)(GPIO_BASE + 0x10))  // Output Register
#define GPIO_IN         (*(volatile uint32_t *)(GPIO_BASE + 0x14))  // Input Register

// Macro to calculate control register address for a specific pin
#define GPIO_CTRL(pin)  (*(volatile uint32_t *)(GPIO_BASE + 0x04 + (pin * 8)))
int main() {
    // Set pin 0 as output
    GPIO_OE |= (1 << 0);        // Set bit 0 of the OE register to 1 (output enable)
    GPIO_CTRL(0) = 5;           // Set function select to SIO ( single input / output -- normal gpio operation ) for pin 0

    // Set pin 1 as input
    GPIO_OE &= ~(1 << 1);       // Set bit 1 of the OE register to 0 (input enable)
    GPIO_CTRL(1) = 5;           // Set function select to SIO (( single input / output -- normal gpio operation )) for pin 1

    // Set pin 0 high (output)
    GPIO_OUT |= (1 << 0);       // Set pin 0 to high

    // Read pin 1 (input)
    uint32_t pin1_value = (GPIO_IN & (1 << 1)) != 0;  // Read the state of pin 1

    // Set pin 0 low (output)
    GPIO_OUT &= ~(1 << 0);      // Set pin 0 to low

    // Continue running (this is just an example, so infinite loop)
    while (1);

    return 0;
}
```

Note : Extra information Don't Memorize

- **GPIO_OE**: Output enable register (1 for output, 0 for input).
  32-bit register , each bit maps to a corresponding GPIO PIN resembling the **DDRX** .

- **GPIO_OUT**: Output value register. Resembles **PORTX** in output pins, used to enter the output needed in the pin in its corresponding bit .

- **GPIO_IN**: Input value register. Used to read the Value of the input pin using its corresponding bit resembling PINX.

SIO: GPIO_OE Register

Offset: 0x020

Description

GPIO output enable

Table 24. GPIO_OE Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-
29:0	Set output enable (1/0 → output/input) for GPIO0...29. Reading back gives the last value written. If core 0 and core 1 both write to GPIO_OE simultaneously (or to a SET/CLR/XOR alias), the result is as though the write from core 0 took place first, and the write from core 1 was then applied to that intermediate result.	RW	0x00000000

2.3. Processor subsystem 46

RP2040 Datasheet

Note : Extra information
Don't Memorize

GPIO_CTRL Register Overview :

There is a 32-bit GPIO control register for each pin , separated by 8 bytes in the memory address space.

1.Purpose: The GPIO_CTRL register allows you to configure various attributes of GPIO pins, such as their modes, functions. **Function Selection**: Determines the function of the GPIO pin (e.g., GPIO, UART, SPI).

IO_BANK0: GPIO0_CTRL, GPIO1_CTRL, ..., GPIO28_CTRL, GPIO29_CTRL Registers

Offsets: 0x004, 0x00c, ..., 0x0e4, 0x0ec

Description

GPIO control including function select and overrides.

Table 285. GPIO0_CTRL, GPIO1_CTRL, ..., GPIO28_CTRL, GPIO29_CTRL Registers

Bits	Name	Description	Type	Reset
31:30	Reserved.	-	-	-
29:28	IRQOVER	0x0 → don't invert the interrupt 0x1 → invert the interrupt 0x2 → drive interrupt low 0x3 → drive interrupt high	RW	0x0
27:18	Reserved.	-	-	-
17:16	INOVER	0x0 → don't invert the peri input 0x1 → invert the peri input 0x2 → drive peri input low 0x3 → drive peri input high	RW	0x0
15:14	Reserved.	-	-	-
13:12	OEOVER	0x0 → drive output enable from peripheral signal selected by funcsel 0x1 → drive output enable from inverse of peripheral signal selected by funcsel 0x2 → disable output 0x3 → enable output	RW	0x0
11:10	Reserved.	-	-	-
9:8	OUTOVER	0x0 → drive output from peripheral signal selected by funcsel 0x1 → drive output from inverse of peripheral signal selected by funcsel 0x2 → drive output low 0x3 → drive output high	RW	0x0
7:5	Reserved.	-	-	-
4:0	FUNCSEL	Function select. 31 == NULL. See GPIO function table for available functions.	RW	0x1f

2.19. GPIO 247

RP2040 Datasheet

Note : Extra information Don't Memorize

However writing Low level code is extremely tiring and complicated !

gpio.c

gpio.c > ...

```c
#include <stdint.h>

#define GPIO_BASE       0x40014000  // Base address for GPIO registers
#define GPIO_OE         (*(volatile uint32_t *)(GPIO_BASE + 0x20))  // Output Enable Register
#define GPIO_OUT        (*(volatile uint32_t *)(GPIO_BASE + 0x10))  // Output Register
#define GPIO_IN         (*(volatile uint32_t *)(GPIO_BASE + 0x14))  // Input Register

// Macro to calculate control register address for a specific pin
#define GPIO_CTRL(pin)  (*(volatile uint32_t *)(GPIO_BASE + 0x04 + (pin * 8)))
int main() {
    // Set pin 0 as output
    GPIO_OE |= (1 << 0);        // Set bit 0 of the OE register to 1 (output enable)
    GPIO_CTRL(0) = 5;           // Set function select to SIO ( single input / output -- normal gpio operation ) for pin 0

    // Set pin 1 as input
    GPIO_OE &= ~(1 << 1);       // Set bit 1 of the OE register to 0 (input enable)
    GPIO_CTRL(1) = 5;           // Set function select to SIO (( single input / output -- normal gpio operation )) for pin 1

    // Set pin 0 high (output)
    GPIO_OUT |= (1 << 0);       // Set pin 0 to high

    // Read pin 1 (input)
    uint32_t pin1_value = (GPIO_IN & (1 << 1)) != 0;  // Read the state of pin 1

    // Set pin 0 low (output)
    GPIO_OUT &= ~(1 << 0);      // Set pin 0 to low

    // Continue running (this is just an example, so infinite loop)
    while (1);

    return 0;
}
```

Pico SDK C library provides a high-level API for the hardware peripherals

Link : https://www.raspberrypi.com/documentation/pico-sdk/

GPIO Functions in Pico SDK

1. **Initialization and Direction**
    - **gpio_init(uint gpio):** Initializes the specified GPIO pin.
    - **gpio_set_dir(uint gpio, bool out):** Sets the direction of the GPIO pin (input or output).
        - out = true for output, out = false for input.
2. **Setting and Reading GPIO States**
    - **gpio_put(uint gpio, bool value):** Sets the state of an output GPIO pin (HIGH or LOW).
    - **gpio_get(uint gpio):** Reads the current state of an input GPIO pin.
3. **Pull-up/Pull-down Control**
    - **gpio_pull_up(uint gpio):** Enables the internal pull-up resistor on the specified pin.
    - **gpio_pull_down(uint gpio):** Enables the internal pull-down resistor on the specified pin.
    - **gpio_disable_pulls(uint gpio):** Disables both pull-up and pull-down resistors on the pin.

EX2

Implement an embedded C on a Pico RP-2024 code to :

1. Connect push button A to pin 5 in PORT C (Positive Logic)
2. Connect push button B to pin 3 in PORT B (Negative Logic)
3. Apply the Internal pullup resistor to pin3 PORT B
4. Configure PIN 2 in PORTD as output
5. Connect Pin 2 to RED LED Pin5-- RED (Hardware step)
6. Turn on Red LED when A is pressed
7. Turn off the RED LED when B is pressed .

Utilized the GPIO SDK API to initialize pins , set their directions , set output and receive input .

gpio.c

gpio.c > main()

```c
int main() {
    // Define pins
    const uint LED_PIN = 2;                 // Output pin (for LED)
    const uint INPUT_PIN_A = 5;             // Input pin for button A (positive logic)
    const uint INPUT_PIN_B = 3;             // Input pin for button B (negative logic with pull-up)
    // Initialize output pin (LED)
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);  // Set as output
    // Initialize input pins
    gpio_init(INPUT_PIN_A);
    gpio_set_dir(INPUT_PIN_A, GPIO_IN);    // Set as input

    gpio_init(INPUT_PIN_B);
    gpio_set_dir(INPUT_PIN_B, GPIO_IN);    // Set as input
    gpio_pull_up(INPUT_PIN_B);              // Enable pull-up resistor for negative logic

    // Infinite loop to check button states and control the LED
    while (1) {
        // Read the state of button A (positive logic)
        bool button_a_state = gpio_get(INPUT_PIN_A);  // HIGH when not pressed, LOW when pressed
        // Read the state of button B (negative logic, pull-up enabled)
        bool button_b_state = gpio_get(INPUT_PIN_B);  // LOW when not pressed, HIGH when pressed
        if (button_a_state) {
            // If button A is pressed (positive logic), turn on the LED
            gpio_put(LED_PIN, 1);  // Set the LED pin to HIGH (turn on)
        }

        if (!button_b_state) {
            // If button B is pressed (negative logic), turn off the LED
            gpio_put(LED_PIN, 0);  // Set the LED pin to LOW (turn off)
        }

        sleep_ms(100);  // Small delay to debounce buttons
    }
    return 0;
}
```