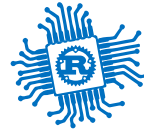


Introduction

Lecture 1



Welcome

to the *Microprocessor Architecture* engineering class

You will learn

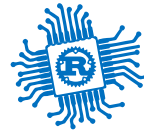
- how hardware works
- how to actually build your own hardware device
- the Rust programming Language

We expect

- to come to class
- ask a lot of questions



Team



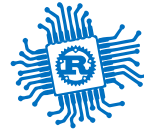
Our team

Lectures

- Alexandru Radovici

Labs

- Teodor Dicu
- Alexandru Ungureanu
- Andrei Zamfir
- Dănuț Aldea
- Ioana Culic
- Cristiana Precup
- Layla El-Ghoul
- Gabriel Păvăloiu
- Andrei Batasev



Outline

Lectures

- 12 lectures
- 1 Q&A lecture for the project

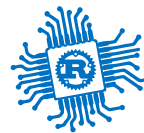
Labs

- 12 labs

Project

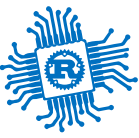
- Build a hardware device running software written in Rust
- The cost for the hardware is around 150 RON
- Presented at PM Fair during the last week of the semester



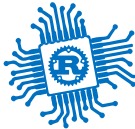


Grading

Part	Description	Points
<u>Lecture tests</u>	You will have a test at every class with subjects from the previous class.	2p
<u>Lab</u>	Your work at every lab will be graded.	2p
<u>Project</u>	You will have to design and implement a hardware device. Grading will be done for the documentation, hardware design and software development.	3p
Exam	You will have to take an exam during the session.	4p
Total	<i>You will need at least 4.5 points to pass the subject.</i>	11p

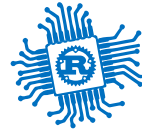


Subjects



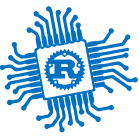
Theory

- How a microprocessor works
- How the ARM Cortex-M processor works
- Using digital signals to control devices
- Using analog signals to read data from sensors
- How interrupts work
- How asynchronous programming works (async/await)
- How embedded operating systems work

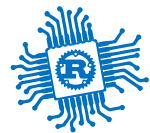


Practical

- How to use the Raspberry Pi Pico
 - Affordable
 - Powerful processor
 - Good documentation
- How to program in Rust
 - Memory Safe
 - *Java-like features, without Java's penalties*
 - Defines an embedded standard interface *embedded-hal*



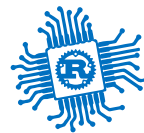
Apollo Guidance Computer



We choose to go to the moon

John F. Kennedy, Rice University, 1961

*in this decade and do the other things, **not because they are easy, but because they are hard**, because **that goal will serve to organize and measure the best of our energies and skills**, because that challenge is one that we are willing to accept, one we are unwilling to postpone, and one which we intend to win, and the others, too.*

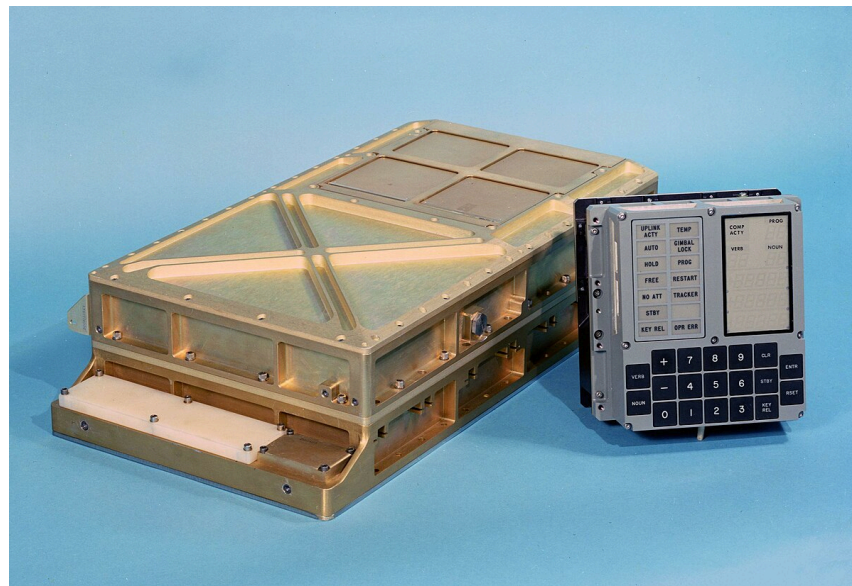


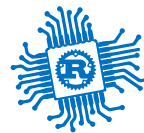
AGC

August 1966

Frequency	2.048 MHz
World Length	15 + 1 bit
RAM	4096 B
Storage	72 KB
Software API	AGC Assembly Language

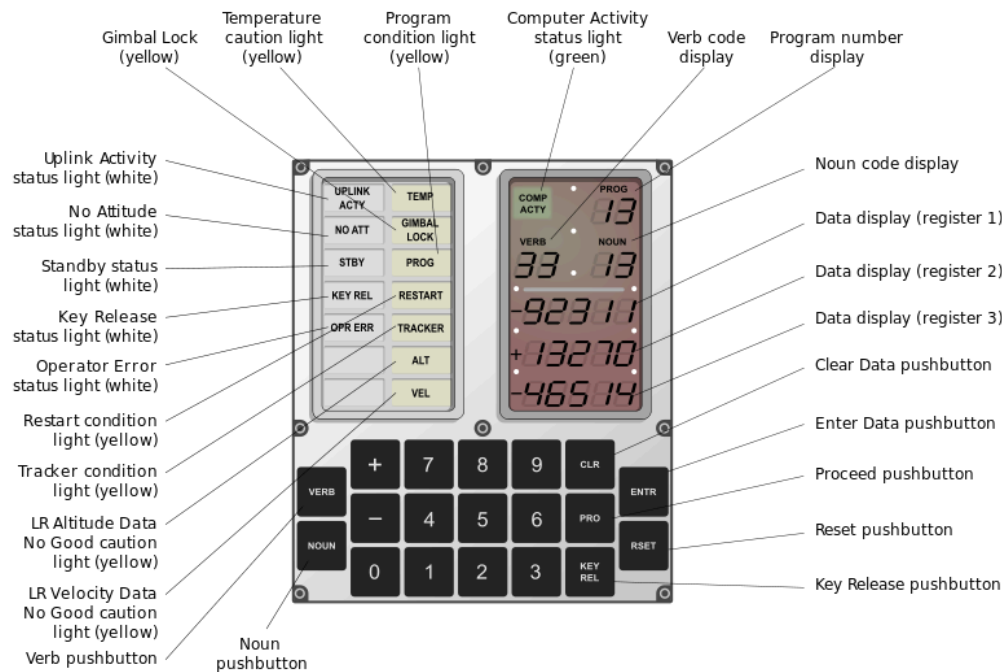
This landed the *moon eagle*.



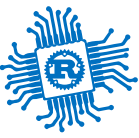


DSKY

Display and keyboard



Simulator



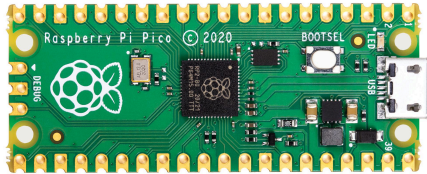
What is a microprocessor?



Microcontroller (MCU)

Integrated in embedded systems for certain tasks

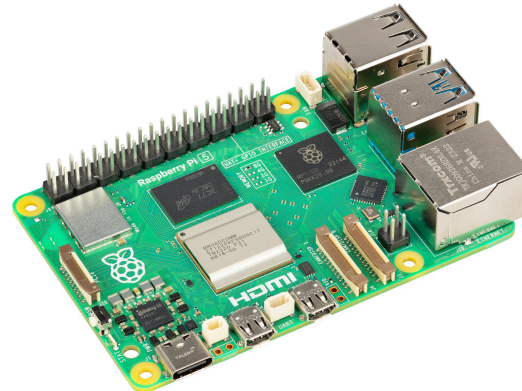
- low operating frequency (MHz)
- a lot of I/O ports
- controls hardware
- does not require an Operating System
- costs \$0.1 - \$25
- annual demand is billions



Microprocessor (CPU)

General purpose, for PC & workstations

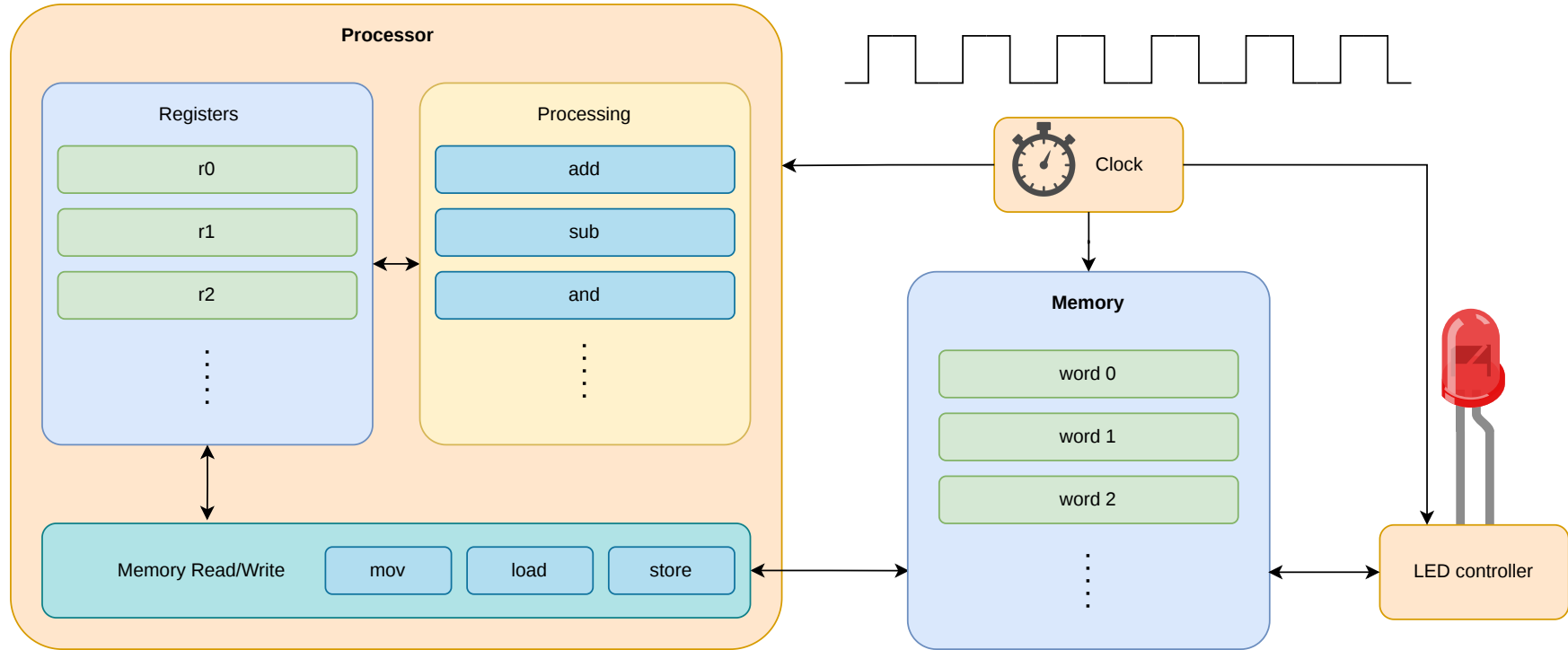
- high operating frequency (GHz)
- limited number of I/O ports
- usually requires an Operating System
- costs \$75 - \$500
- annual demand is tens of millions

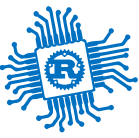




How a microprocessor works

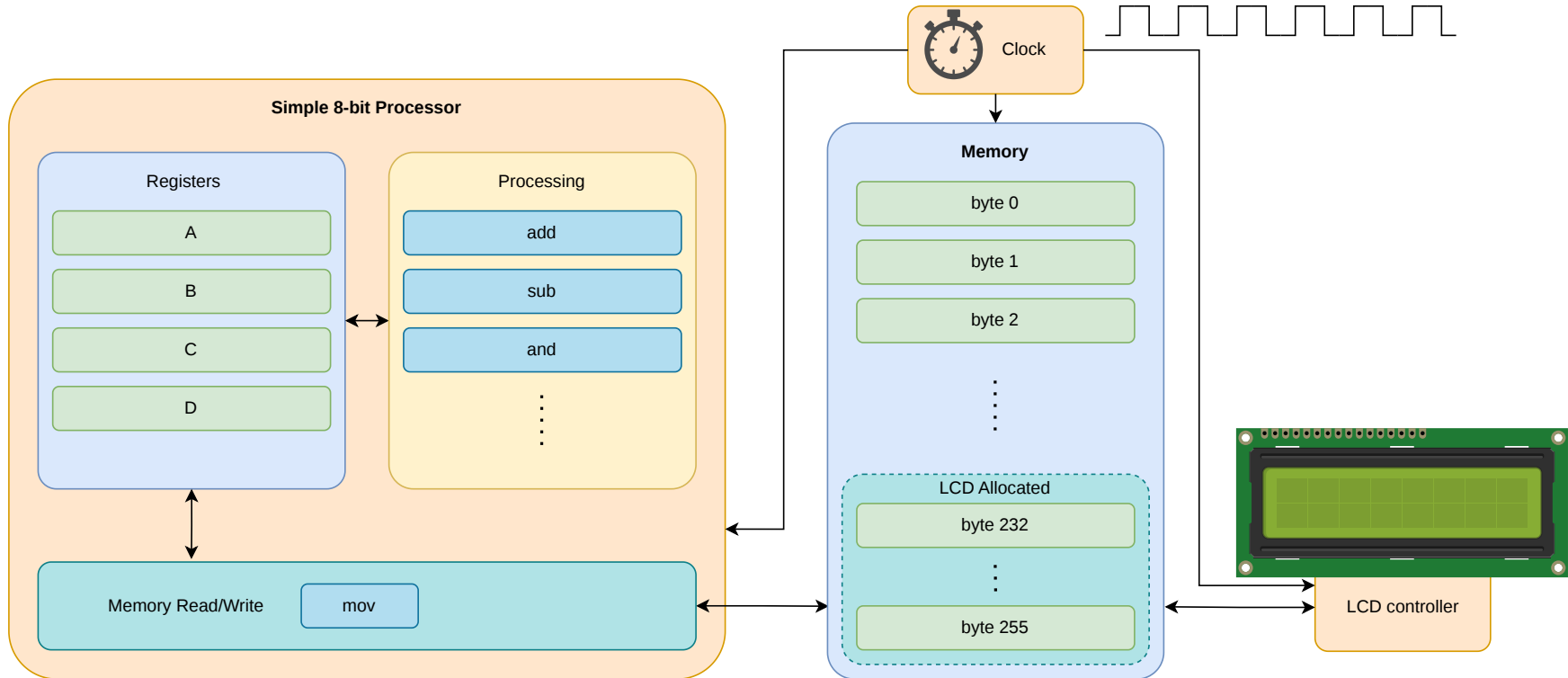
This is a simple processor

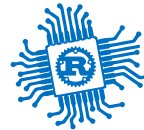




8 bit processor

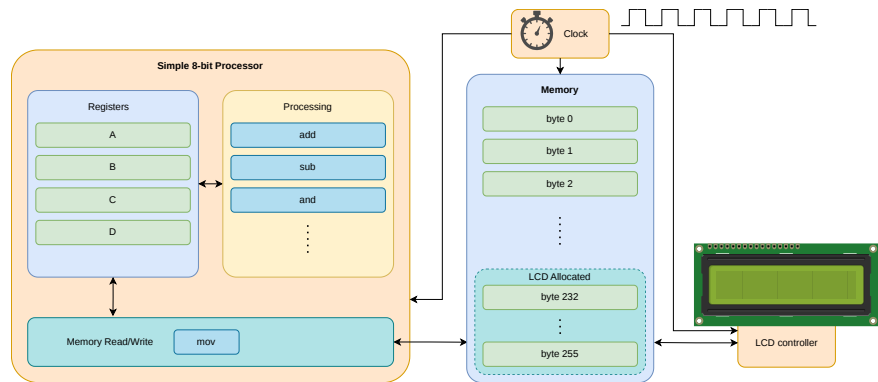
a simple 8 bit processor with a text display





Programming

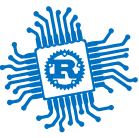
in Rust



```
1 use eight_bit_processor::print;
2
3 static hello: &str = "Hello World!";
4
5 #[start]
6 fn start() {
7     print(hello);
8 }
```

Assembly

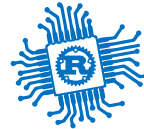
```
1     JMP start
2     hello: DB "Hello World!" ; Variable
3           DB 0 ; String terminator
4     start:
5         MOV C, hello ; Point to var
6         MOV D, 232 ; Point to output
7         CALL print
8         HLT ; Stop execution
9     print: ; print(C:*from, D:*to)
10        PUSH A
11        PUSH B
12        MOV B, 0
13    .loop:
14        MOV A, [C] ; Get char from var
15        MOV [D], A ; Write to output
16        INC C
17        INC D
18        CMP B, [C] ; Check if end
19        JNZ .loop ; jump if not
20
21        POP B
22        POP A
23        RET
```



Demo

a working example for the previous code

Start



Real World Microcontrollers

Intel / AVR / PIC / TriCore / ARM Cortex-M / RISC-V rv32i(a)mc

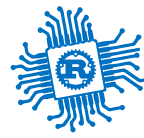


Bibliography

for this section

Joseph Yiu, *The Definitive Guide to ARM® Cortex®-M0 and Cortex-M0+ Processors, 2nd Edition*

- Chapter 1 - *Introduction*
- Chapter 2 - *Technical Overview*



Vendor	Intel
ISA	8051, 8051
Word	8 bit
Frequency	a few MHz
Storage	?
Variants	<i>8048, 8051</i>

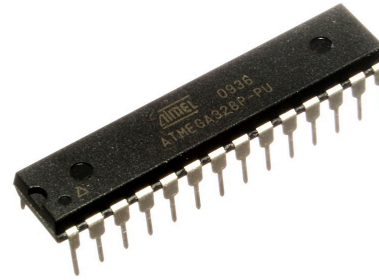




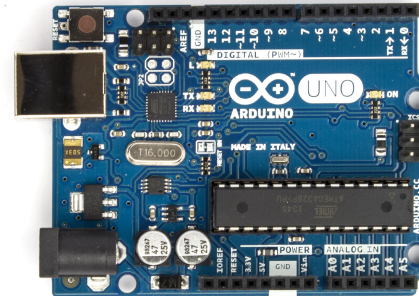
AVR

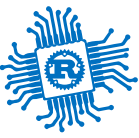
probably *Alf and Vegard's RISC processor*

Authors	Alf-Egil Bogen and Vegard Wollan
Vendor	Microchip (<i>Atmel</i>)
ISA	AVR
Word	8 bit
Frequency	1 - 20 MHz
Storage	4 - 256 KB
Variants	<i>ATmega</i> , <i>ATtiny</i>



Board

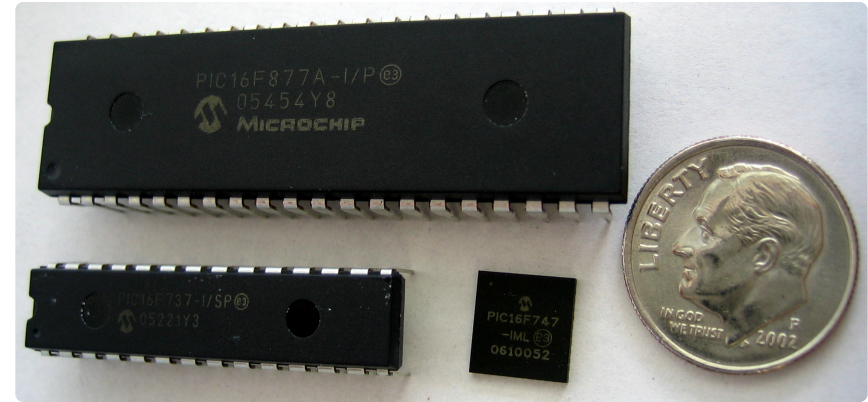




PIC

Peripheral Interface Controller / Programmable Intelligent Computer

Vendor	Microchip
ISA	PIC
Word	8 - 32
Frequency	1 - 20 MHz
Storage	256 B - 64 KB
Variants	<i>PIC10, PIC12, PIC16, PIC18, PIC24, PIC32</i>

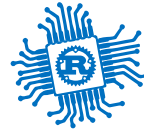




TriCore



Vendor	Infineon
ISA	AURIX32
Word	32 bit
Frequency	hundreds of MHz
Storage	a few MB
Variants	<i>TC2xx, TC3xx, TC4xx</i>

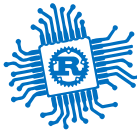


ARM Cortex-M

Advanced RISC Machine

arm

Vendor	Qualcomm, NXP, Nordic Semiconductor, Broadcom, Raspberry Pi
ISA	ARMv6-M (Thumb and some Thumb-2) ARMv7-M (Thumb and Thumb-2) ARMv8-M (Thumb and Thumb-2)
Word	32
Frequency	1 - 900 MHz
Storage	up to a few MB
Variants	<i>M0, M0+, M3, M4, M7, M23, M33</i>

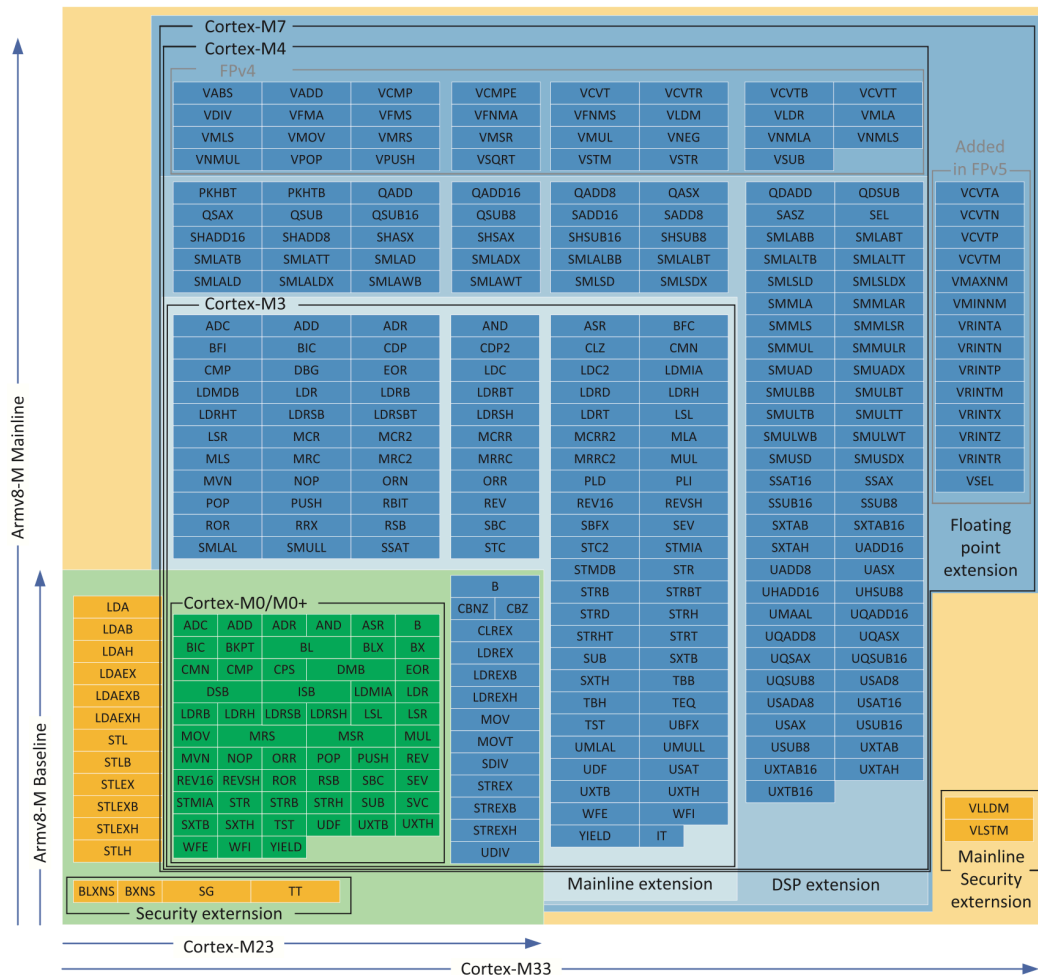


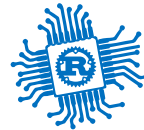
ARM Cortex-M Instruction Set

what the MCU can do

Fun Facts

- M0/M0+ has no `div`
- M0 - M3 have no floating point
- M23 and M33 have security extensions



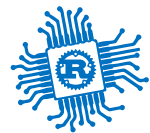


RISC-V rv32i(a)mc

Fifth generation of RISC ISA

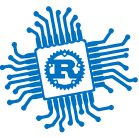


Authors	University of California, Berkeley
Vendor	Espressif System
ISA	rv32i(a)mc
Word	32 bit
Frequency	1 - 200 MHz
Storage	4 - 256 KB
Variants	<i>rv32imc, rv32iamc</i>



RP2040

ARM Cortex-M0+, built by Raspberry Pi

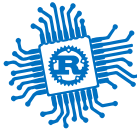


Bibliography

for this section

Raspberry Pi Ltd, *RP2040 Datasheet*

- Chapter 1 - *Introduction*
- Chapter 2 - *System Description*
 - Section 2.1 - *Bus Fabric*



RP2040

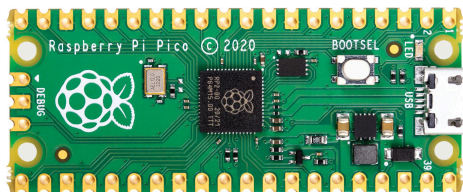
the MCU

Vendor	Raspberry Pi
Variant	ARM Cortex-M0+
ISA	ARMv6-M (Thumb and some Thumb-2)
Cores	2
Word	32 bit
Frequency	up to 133 MHz
RAM	264 KB

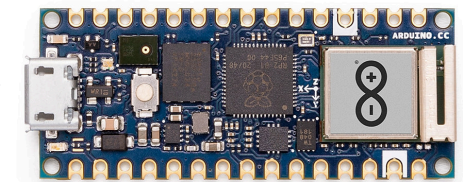
Boards

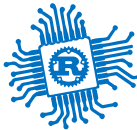
that use RP2040

Raspberry Pi Pico (W)

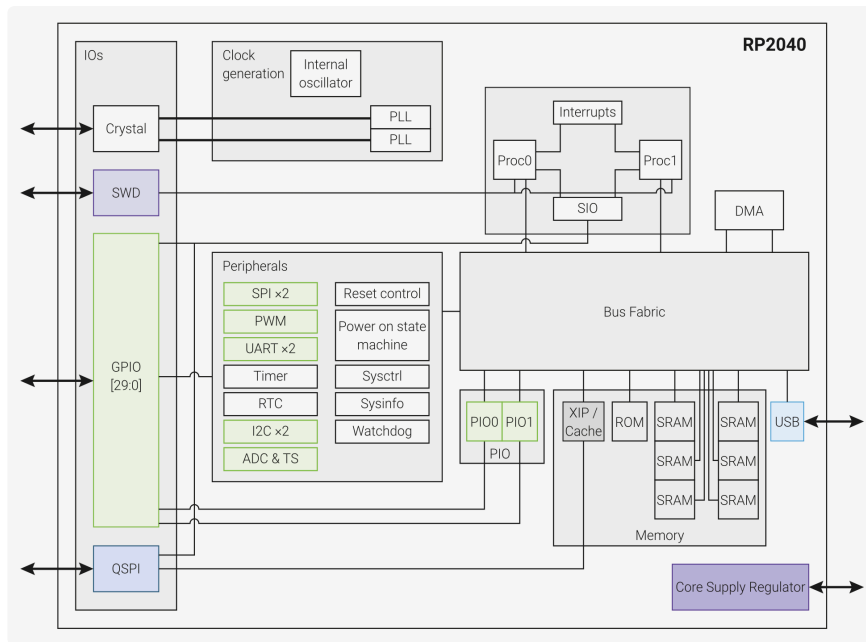


Arduino Nano RP2040 Connect





The Chip



GPIO: General Purpose Input/Output

SWD: Debug Protocol

DMA: Direct Memory Access

Peripherals

SIO Single Cycle I/O (implements GPIO)

PWM Pulse Width Modulation

ADC Analog to Digital Converter

(Q)SPI (Quad) Serial Peripheral Interface

UART Universal Async. Receiver/Transmitter

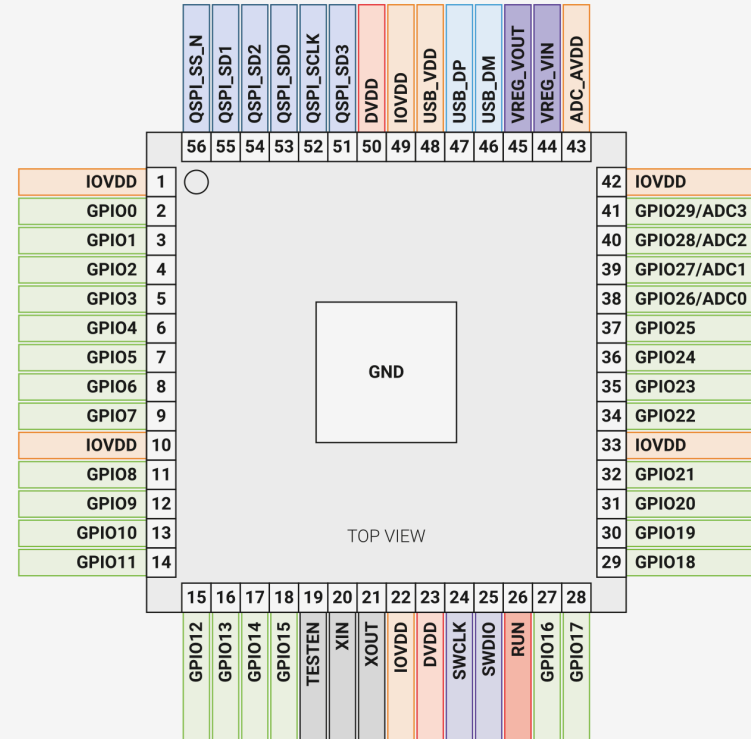
RTC Real Time Clock

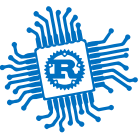
I2C Inter-Integrated Circuit

PIO Programmable Input/Output

Pins

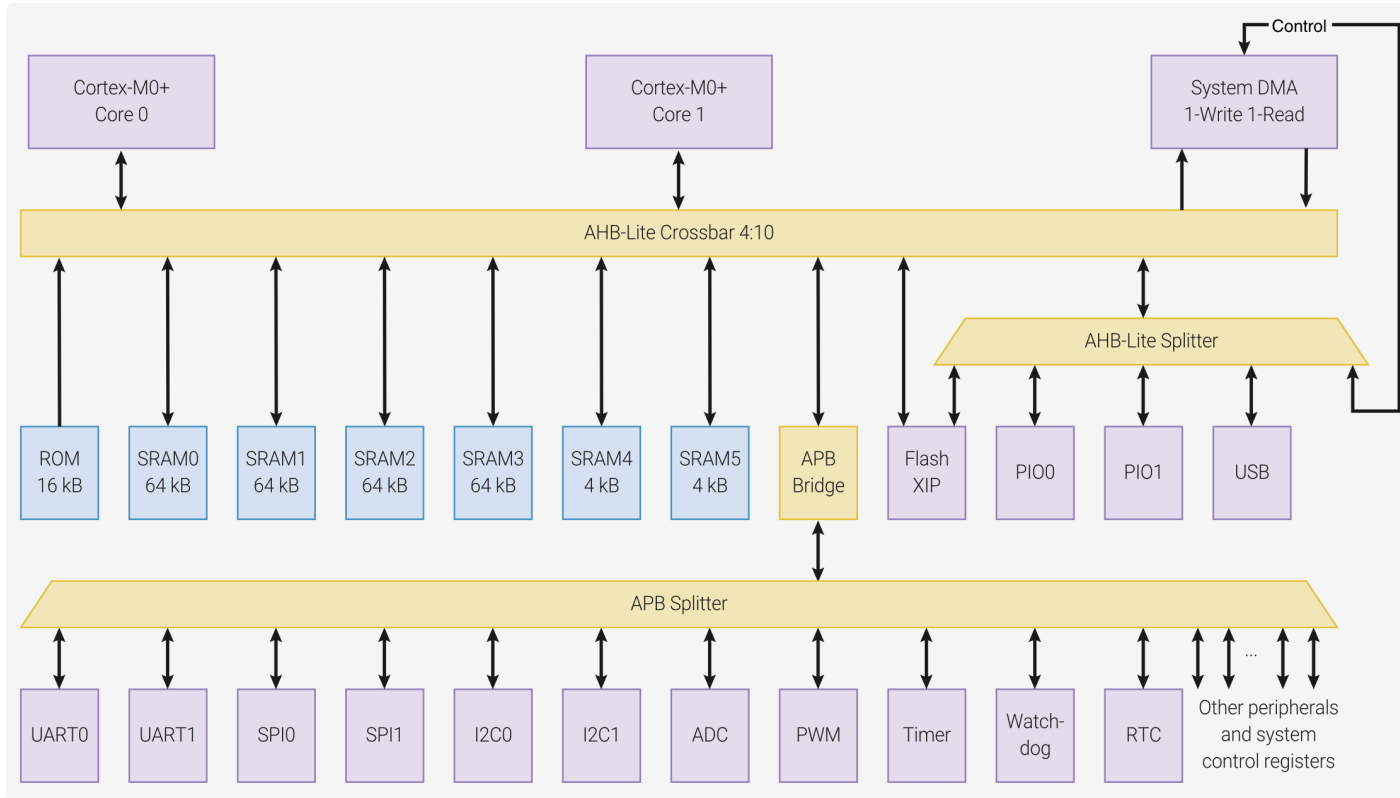
have multiple functions

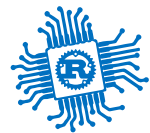




The Bus

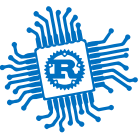
that interconnects the cores with the peripherals





RP2350

ARM Cortex-M33, built by Raspberry Pi

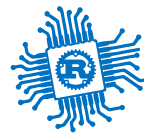


Bibliography

for this section

Raspberry Pi Ltd, *RP2350 Datasheet*

- Chapter 1 - *Introduction*
- Chapter 2 - *System Description*
 - Section 2.1 - *Bus Fabric*



RP2350

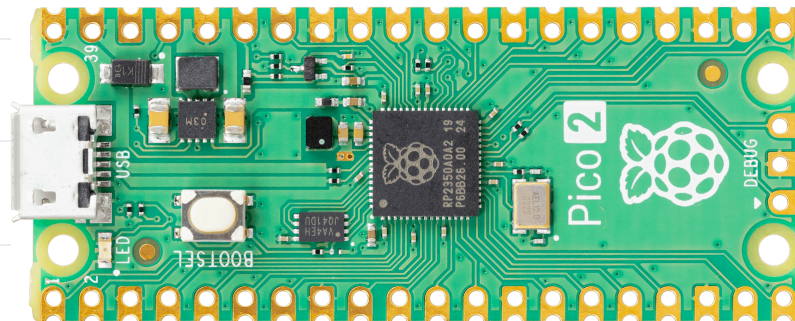
the MCU

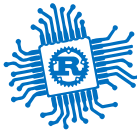
Boards

that use RP2350

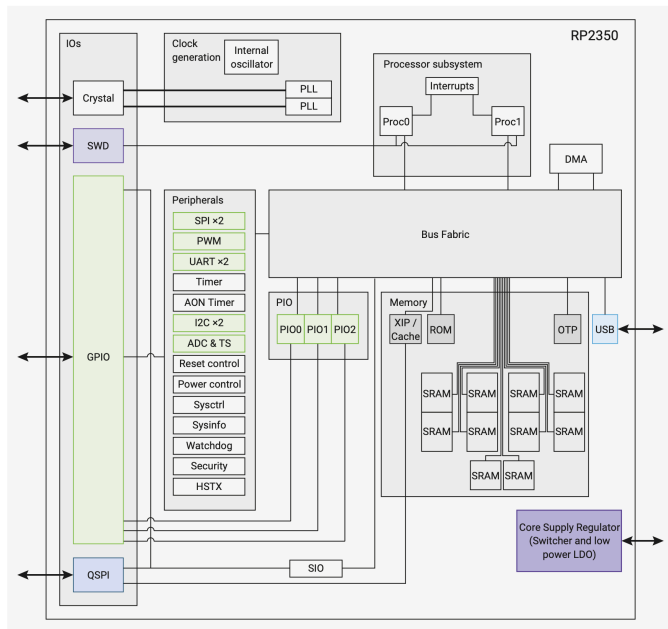
Raspberry Pi Pico 2 (W)

Vendor	Raspberry Pi
Variant	ARM Cortex-M33 / Hazard3 RISC-V
ISA	ARMv8-M / rv32iamc
Cores	2
Word	32 bit
Frequency	up to 150 MHz
RAM	520 KB





The Chip



GPIO: General Purpose Input/Output

SWD: Debug Protocol

DMA: Direct Memory Access

[Datasheet RP2350](#)

Peripherals

SIO Single Cycle I/O (implements GPIO)

PWM Pulse Width Modulation

ADC Analog to Digital Converter

(Q)SPI (Quad) Serial Peripheral Interface

UART Universal Async. Receiver/Transmitter

RTC Real Time Clock

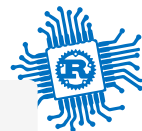
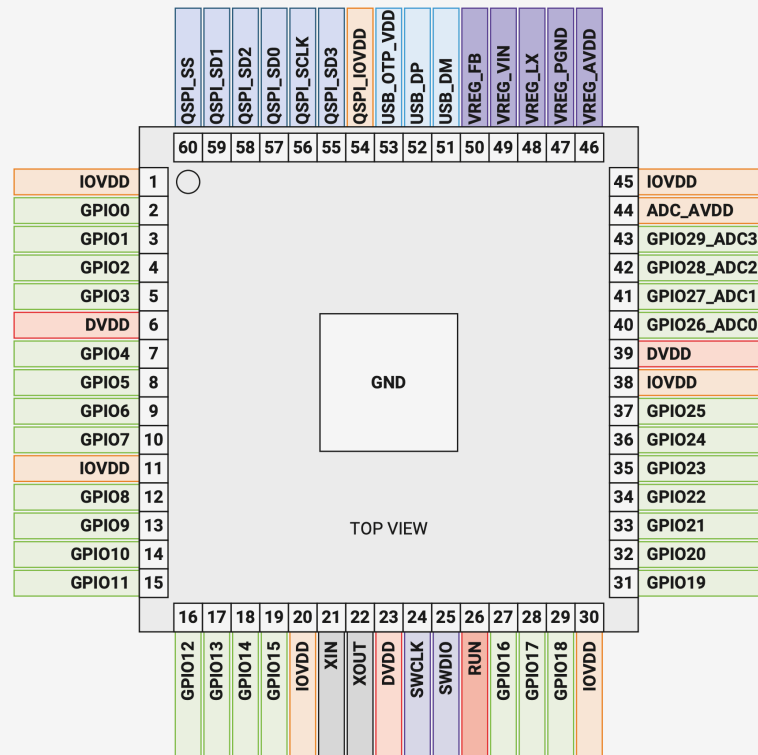
I2C Inter-Integrated Circuit

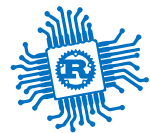
PIO Programmable Input/Output

Pins

have multiple functions

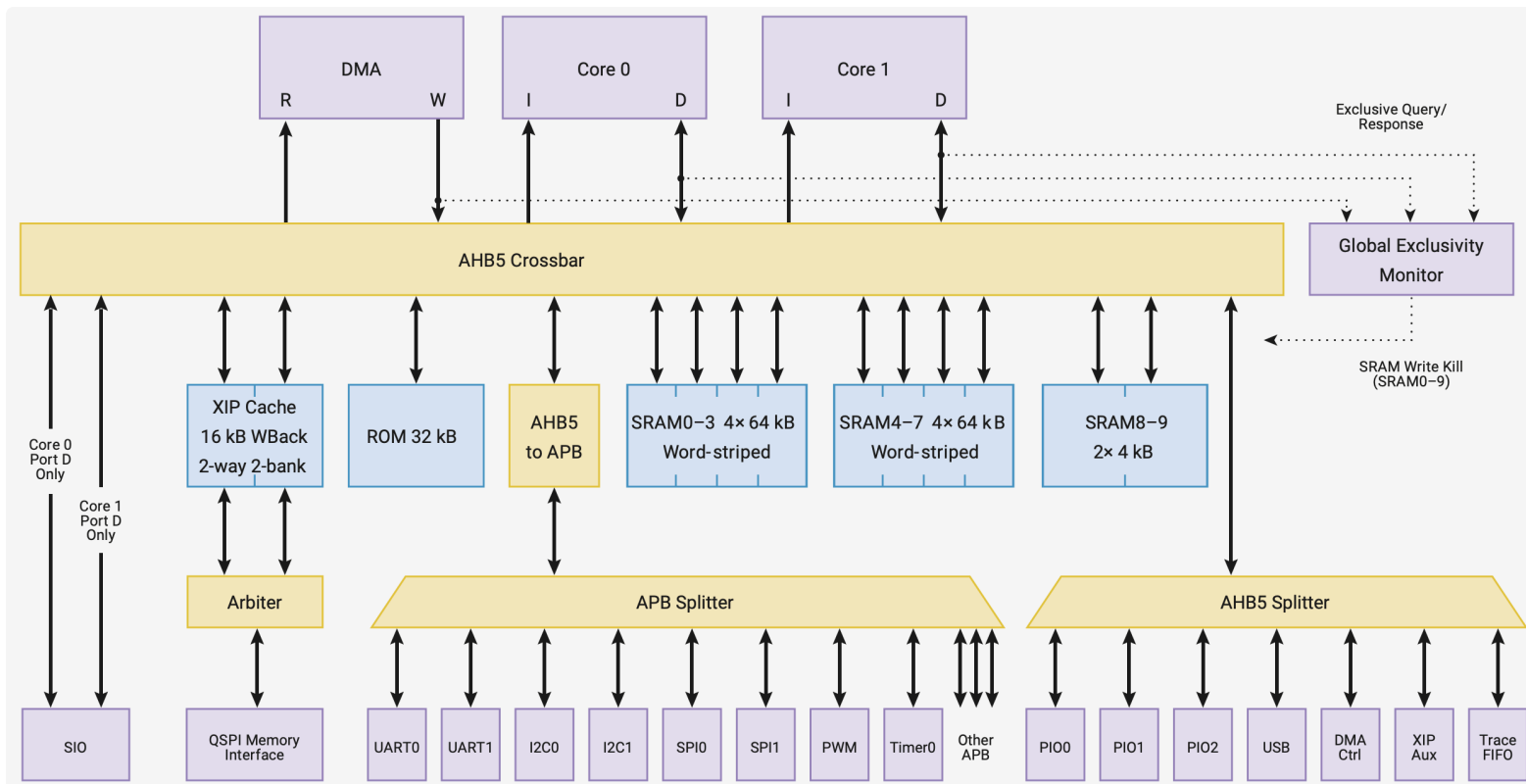
GPIO	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
0		SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PI00	PI01	PI02	QMI CS1n	USB OVCUR DET	
1		SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PI00	PI01	PI02	TRACECLK	USB VBUS DET	
2		SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PI00	PI01	PI02	TRACEDATA0	USB VBUS EN	UART0 TX
3		SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PI00	PI01	PI02	TRACEDATA1	USB OVCUR DET	UART0 RX
4		SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PI00	PI01	PI02	TRACEDATA2	USB VBUS DET	
5		SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PI00	PI01	PI02	TRACEDATA3	USB VBUS EN	
6		SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PI00	PI01	PI02		USB OVCUR DET	UART1 TX
7		SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO	PI00	PI01	PI02		USB VBUS DET	UART1 RX
8		SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PI00	PI01	PI02	QMI CS1n	USB VBUS EN	
9		SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO	PI00	PI01	PI02		USB OVCUR DET	
10		SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO	PI00	PI01	PI02		USB VBUS DET	UART1 TX
11		SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO	PI00	PI01	PI02		USB VBUS EN	UART1 RX
12	HSTX	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PI00	PI01	PI02	CLOCK GPIN0	USB OVCUR DET	
13	HSTX	SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO	PI00	PI01	PI02	CLOCK GPOUT0	USB VBUS DET	
14	HSTX	SPI1 SCK	UART0 CTS	I2C1 SDA	PWM7 A	SIO	PI00	PI01	PI02	CLOCK GPIN1	USB VBUS EN	UART0 TX
15	HSTX	SPI1 TX	UART0 RTS	I2C1 SCL	PWM7 B	SIO	PI00	PI01	PI02	CLOCK GPOUT1	USB OVCUR DET	UART0 RX
16	HSTX	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PI00	PI01	PI02		USB VBUS DET	
17	HSTX	SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PI00	PI01	PI02		USB VBUS EN	
18	HSTX	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PI00	PI01	PI02		USB OVCUR DET	UART0 TX
19	HSTX	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PI00	PI01	PI02	QMI CS1n	USB VBUS DET	UART0 RX
20		SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PI00	PI01	PI02	CLOCK GPIN0	USB VBUS EN	
21		SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PI00	PI01	PI02	CLOCK GPOUT0	USB OVCUR DET	
22		SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PI00	PI01	PI02	CLOCK GPIN1	USB VBUS DET	UART1 TX





The Bus

that interconnects the cores with the peripherals

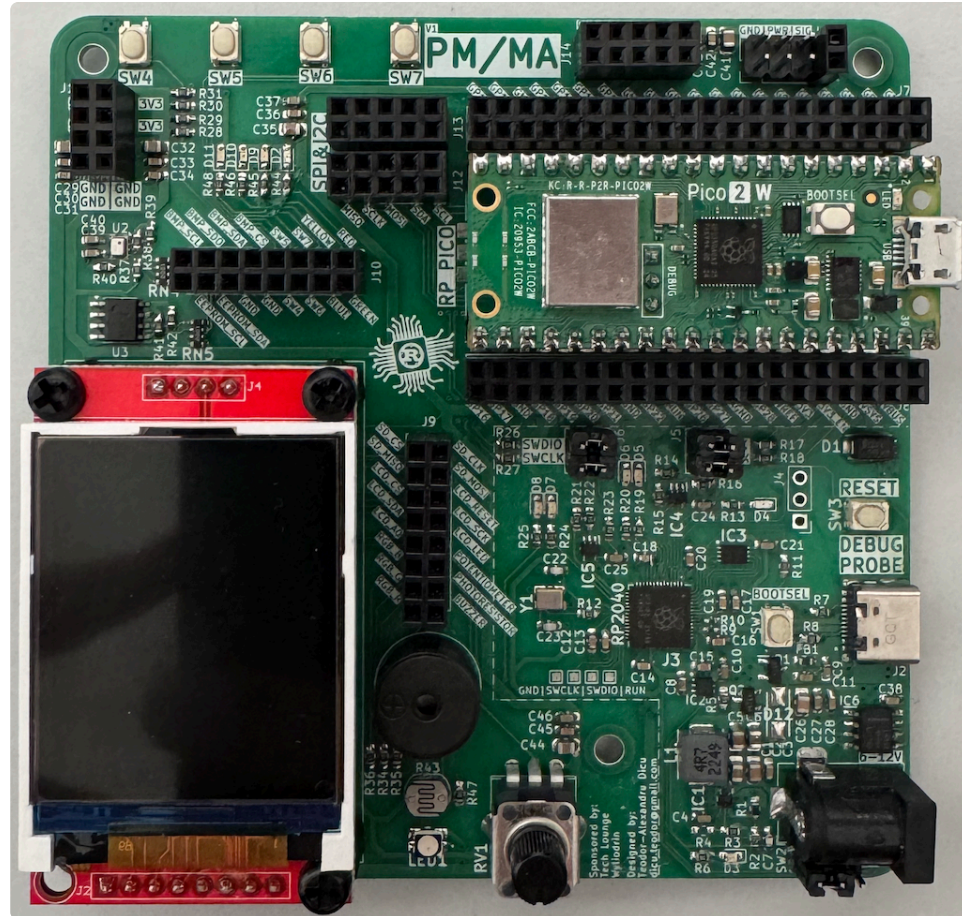


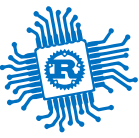


Lab Board

Schematics and PCB

- Raspberry Pi Pico/2 Slot
- RP2040 Debugger
- 4 buttons
- 5 LEDs
- potentiometer
- buzzer
- photoresistor
- I2C EEPROM
- BMP280 Pressure & Temp. sensor
- SPI LCD Display
- SD Card Reader
- USB-C connector
- servo connectors





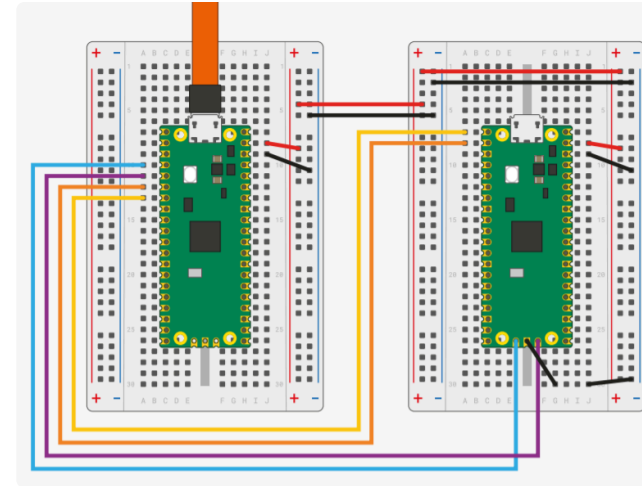
Project

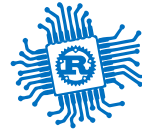
suggested hardware

- the hardware should not cost more than 150 RON
- we suggest having a Raspberry Pi Pico with a debugger

Raspberry Pi Pico 2W + Debug Probe

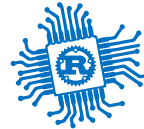
Raspberry Pi Pico 2W + Raspberry Pi Pico 1





Bitwise Ops

How to set and clear bits



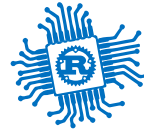
Set bit

set the 1 on position bit of register

```
1 fn set_bit(register: usize, bit: u8) -> usize {
2     // assume register is 0b1000, bit is 2
3     // 1 << 2 is 0b0100
4     // 0b1000 | 0b0100 is 0b1100
5     register | 1 << bit
6 }
```

Set multiple bits

```
1 fn set_bits(register: usize, bits: usize) -> usize {
2     // assume register is 0b1000, bits is 0b0111
3     // 0b1000 | 0b0111 is 0b1111
4     register | bits
5 }
```



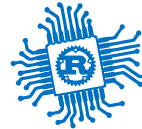
Clear bit

Set the 0 on position bit of register

```
1 fn clear_bit(register: usize, bit: u8) -> usize {
2     // assume register is 0b1100, bit is 2
3     // 1 << 2 is 0b0100
4     // !(1 << 2) is 0b1011
5     // 0b1100 & 0b1011 is 0b1000
6     register & !(1 << bit)
7 }
```

Clear multiple bits

```
1 fn clear_bits(register: usize, bits: usize) -> usize {
2     // assume register is 0b1111, bits is 0b0111
3     // !bits = 0b1000
4     // 0b1111 & 0b1000 is 0b1000
5     register & !bits
6 }
```



Flip bit

Flip the bit on position `bit` of `register`

```
1 fn flip_bit(register: usize, bit: u8) -> usize {
2     // assume register is 0b1000, bit is 2
3     // 1 << 2 is 0b0100
4     // 0b1100 ^ 0b0100 is 0b1000
5     register ^ 1 << bit
6 }
```

Flip multiple bits

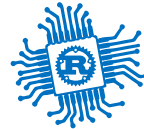
```
1 fn flip_bits(register: usize, bits: usize) -> usize {
2     // assume register is 0b1000, bits is 0b0111
3     // 0b1000 ^ 0b0111 is 0b1111
4     register ^ bits
5 }
```



Let's see a combined operation for value extraction

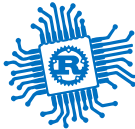
- We presume an 32 bits ID = `0b1100_1010_1111_1100_0000_1111_0110_1101`
- And want to extract a portion `0b1100_1010_1111_1100_0000_1111_0110_1101`

```
1  const MASK: u32 = 0b0000_0000_0000_0000_0000_1111_1111_1111;
2
3  fn print_binary(label: &str, num: u32) {
4      println!("{}", num);
5  }
6
7  fn main() {
8      let large_id: u32 = 0b1100_1010_1111_1100_0000_1111_0110_1101;
9      let extracted_bits = (large_id >> 20) & MASK;
10
11     // Print values in binary
12     print_binary("Original_", large_id);
13     print_binary("Mask_____", MASK);
14     print_binary("Extracted", extracted_bits);
15 }
16 /* RESULT
17 Original_: 11001010111111000000111101101101
18 Mask_____: 00000000000000000000111111111111
19 Extracted: 00000000000000000000110010101111 */
```



With nice formatting

```
1  const MASK: u32 = 0b0000_0000_0000_0000_1111_1111_1111;
2  fn format_binary(num: u32) -> String {
3      (0..32).rev()
4          .map(|i| {
5              if i != 0 && i % 4 == 0 {
6                  format!("{}", _, (num >> i) & 1)
7              } else {
8                  format!("{}", (num >> i) & 1)
9              }
10         })
11         .collect::<Vec<_>>()
12         .join("")
13     }
14     fn print_binary(label: &str, num: u32) { println!("{}", label, format_binary(num)); }
15     fn main() {
16         let large_id: u32 = 0b1100_1010_1111_1100_0000_1111_0110_1101;
17         let extracted_bits = (large_id >> 20) & MASK;
18         print_binary("Original_", large_id);
19         print_binary("Extracted", extracted_bits);
20     }
21     /* RESULTS:
22     Original_: 1100_1010_1111_1100_0000_1111_0110_1101
23     Extracted: 0000_0000_0000_0000_0000_1100_1010_1111 */
```



Conclusion

we talked about

- How a processor functions
- Microcontrollers (MCU) / Microprocessors (CPU)
- Microcontroller architectures
- ARM Cortex-M
- RP2040 and RP2350