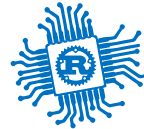




Introduction

Lecture 1



Welcome

to the *Microprocessor Architecture* engineering class

You will learn

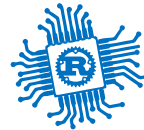
- how hardware works
- how to actually build your own hardware device
- the Rust programming Language

We expect

- to come to class
- ask a lot of questions



Team



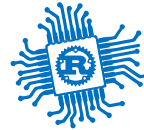
Our team

Lectures

- Alexandru Radovici

Labs

- Andrei Zamfir
- Dănuț Aldea
- Irina Bradu
- Genan Omer
- Cristiana Precup
- Eva Cosma
- Victor Lișman
- Roi Bachynskyi



Outline

Lectures

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

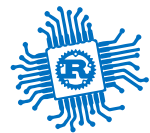
Labs + Project Support

- 12 labs

Project

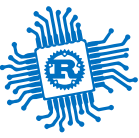
- Build a hardware device running software written in Rust
- Presented at PM Fair during the last week of the semester



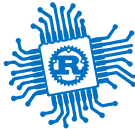


Grading

Part	Description	Points
<u>Lecture</u> <u>tests</u>	You will have a test at every class with subjects from the previous class.	0.2p
<u>Final</u> <u>Lecture test</u>	You will have a test during one of the lectures in May.	1.8p
<u>Lab</u>	Your presence at every lab will be graded.	1p
<u>Lab Test</u>	A final test at the lab - will scale your lab grade	1p
<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
Final Test	You will have to take an exam during the last week of the semester.	4p
Total	<i>You will need at least 4.5 points to pass the subject.</i>	11p

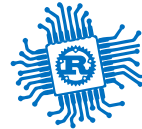


Subjects



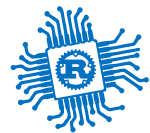
Theory

- 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

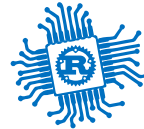


Practical

- How to use the STM32 Nucleo-U545RE-Q
 - 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*



Why Rust



Let's see some code

```
1  #include <stdio.h>
2  #include <stdint.h>
3
4  void printBinary(uint32_t num) {
5      for (int i = 31; i >= 0; i--) {
6          printf("%d", (num >> i) & 1);
7          if (i % 8 == 0) printf(" ");
8      }
9      printf("\n");
10 }
11
12 int main()
13 {
14     uint8_t a;
15     uint32_t b;
16
17     a = 0x01;
18     b = a << 24;
19
20     printBinary(a);
21     printBinary(b);
22
23     return 0;
24 }
```

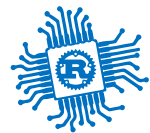


Variables in C

```
1  #include <stdio.h>
2
3  int8_t, uint8_t
4  int16_t, uint16_t
5  int32_t, uint32_t
```

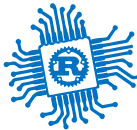
Variables in Rust

```
1  u8, u16, u32, u64, u128
2  i8, i16, i32, i64, i128
3  usize // word size (eg - 32b for 32b processor)
4  isize // word size (eg - 32b for 32b processor)
5
6  //NOTES:
7  char // 4 bytes != u8 //UTF-8 not ASCII like in C
8  b"str" // ASCII string (slide)
9  "str" // UTF-8 string (slice)
10
11 's' // char
12 b's' // u8 (ASCII char)
```



Lets see how C++ is doing

Link	Memory-safety issue	Why Rust prevents it
<u>ZDI-24-854</u>	Unchecked AES-key length copied into fixed stack buffer => overflow enables network-adjacent remote code execution.	Safe Rust enforces bounds checks; unchecked stack copies aren't possible.
<u>Toyota: single bit flip</u>	Stack overflow/memory corruption can kill RTOS task, bypass fail-safes => loss of throttle control (unintended acceleration).	Rust prevents overflows/races; hardware bit-flips and logic bugs remain possible.
<u>CrowdStrike RCA (Channel File 291)</u>	Template field-count mismatch caused out-of-bounds input-array read in sensor, triggering Windows system crash/BSOD.	Rust bounds-checked indexing prevents OOB reads; you get an error instead.



How do we keep C++ code functional in safety-critical applications?

Lots of tooling and processes

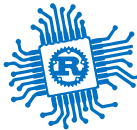
Coding standards / restricted subsets: MISRA C/C++, AUTOSAR C++14, SEI CERT C/C++, JSF AV C++ (plus documented deviations/waivers).

Static analysis & compliance checking: rule checkers + dataflow analyzers (e.g., Polyspace, Coverity/CodeSonar/Parasoft, clang-tidy) integrated in CI.

Compiler hardening & build gates: warnings-as-errors, stack protection, fortified libc checks / hardening bundles (e.g., `_FORTIFY_SOURCE`, `-fstack-protector-strong`, `-fhardened`).

Dynamic bug finding (test builds): sanitizers for memory/UB/races (ASan/UBSan/TSan), plus coverage-guided fuzzing (libFuzzer).

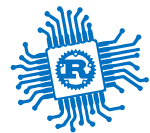
Safety evidence overhead: mandated reviews + traceability + V&V activities (ISO 26262 / DO-178C / IEC 61508-style workflows).



Why Rust - some tech insights

The tagline of Rust is No Undefined Behavior.

- no null reference; the Rust compiler explicitly asks developers to check this;
- no implicit cast, even adding a `u32` to a `u8` must be casted;
- safe access to shared data across threads verified at compile time;
- uses type states to move runtime checks to compile time and force developers to check;
- clearly defined data types, unlike `int` or `long` ;
- safe unions, that provide a safeguard to prevent wrong interpretation of data;
- clear code organization into crates and modules;
- backward compatibility at crate level.



Does Rust remove the need for tooling?

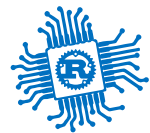
No, but it sure makes code safer and dev faster

Rust's advantage is biggest in safe Rust.

As unsafe/FFI grows, assurance shifts back to: unsafe policy & reviews, FFI boundary correctness, sanitizers/fuzzing on risky edges, dependency/toolchain governance; plus the same ISO/DO-178 traceability & V&V.

Rust in Android: move fast and fix things

- A 1000x reduction in memory safety vulnerability density compared to Android's C and C++ code
- With Rust changes having a 4x lower rollback rate and spending 25% less time in code review, the safer path is now also the faster one



Who supports Rust-lang

Some links to read

[the NSA: U.S. and International Partners Issue Recommendations to Secure Software Products Through Memory Safety](#)

[The White House: Back to the building blocks: A path toward secure and measurable software](#)

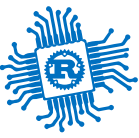
[How Rust went from a side project to the world's most-loved programming language](#)

[On Rust-lang adoption based on git-hub adoption](#)

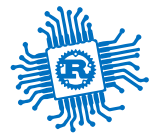
[Rust developers at Google are twice as productive as C++ teams](#)

Some companies that are building up Rust teams in embedded:

- Airbus, Ampere, Bae Systems, Boeing, Ford, General Dynamics, Hyundai, Northrop Grumman, NXP, Thales, Toyota, Volvo



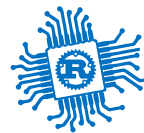
Apollo Guidance Computer



We choose to go to the moon

John F. Kennedy, Rice University, 1961

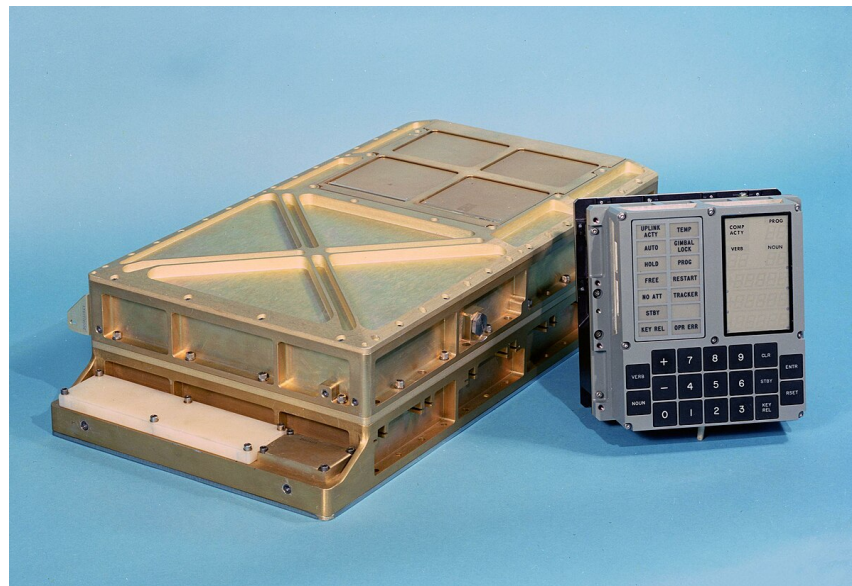
*in this decade and do the other things, **not because they are easy, but because they are hard**, because **that goal will serve to organize and measure the best of our energies and skills**, because that challenge is one that we are willing to accept, one we are unwilling to postpone, and one which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



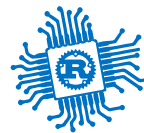
AGC

August 1966

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

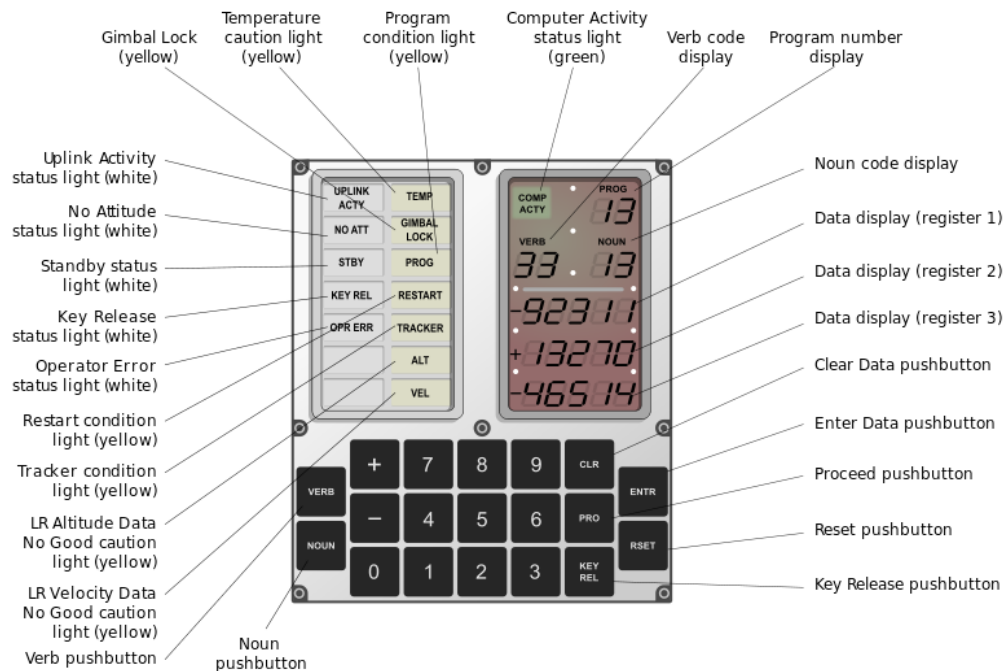


This landed the *moon eagle*.

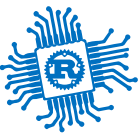


DSKY

Display and keyboards



Simulator



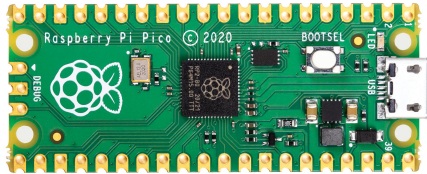
What is a microprocessor?



Microcontroller (MCU)

Integrated in embedded systems for certain tasks

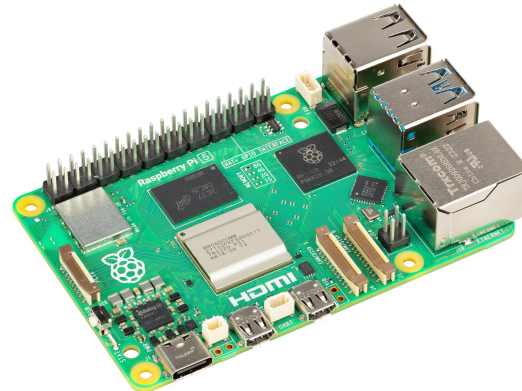
- low operating frequency (MHz)
- a lot of I/O ports
- controls hardware
- does not require an Operating System
- costs \$0.1 - \$25
- annual demand is billions



Microprocessor (CPU)

General purpose, for PC & workstations

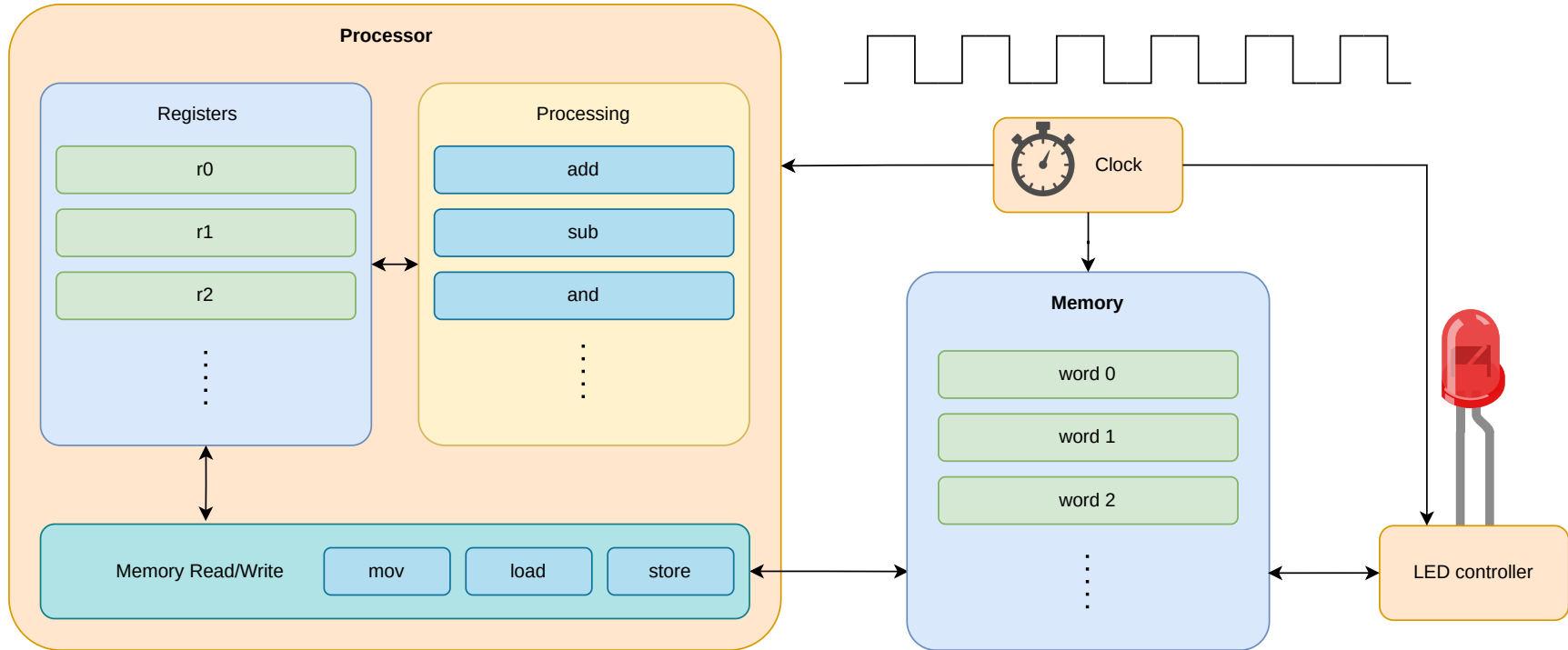
- high operating frequency (GHz)
- limited number of I/O ports
- usually requires an Operating System
- costs \$75 - \$500
- annual demand is tens of millions





How a microprocessor works

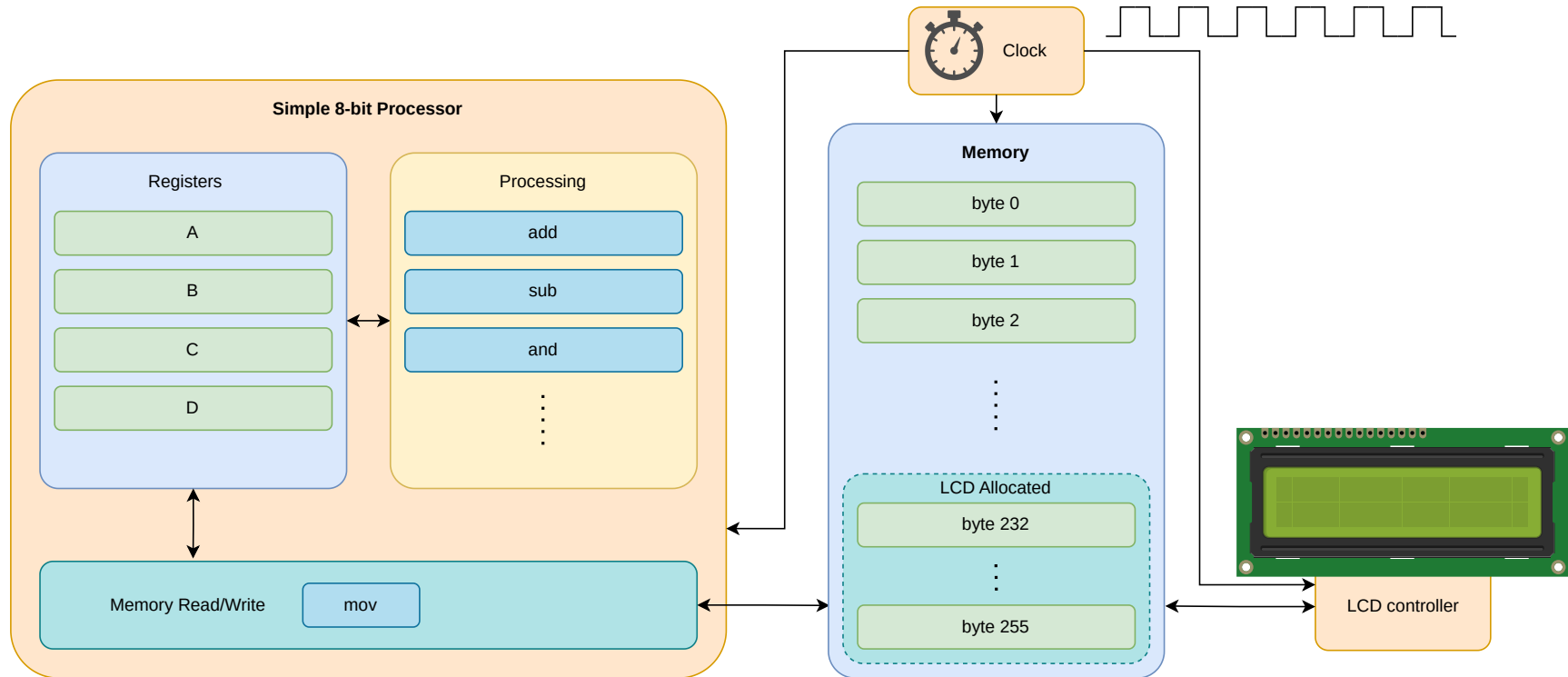
This is a simple processor

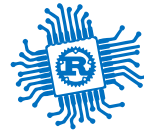




8 bit processor

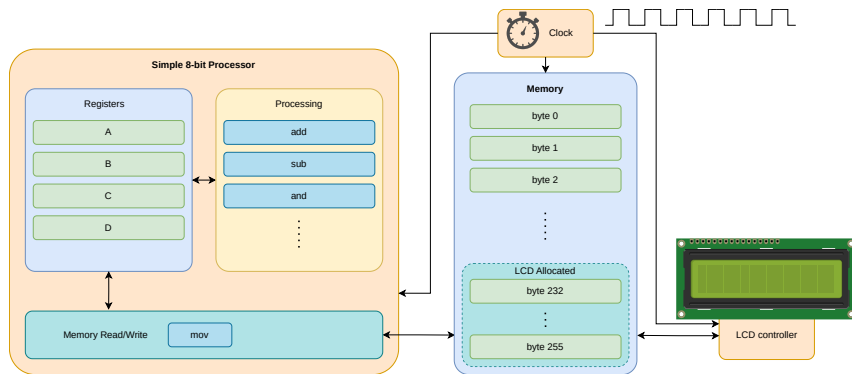
a simple 8 bit processor with a text display





Programming

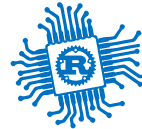
in Rust



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

Assembly

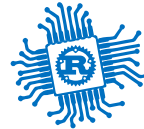
```
1     JMP start
2     hello: DB "Hello World!" ; Variable
3           DB 0 ; String terminator
4     start:
5         MOV C, hello ; Point to var
6         MOV D, 232 ; Point to output
7         CALL print
8         HLT ; Stop execution
9     print: ; print(C:*from, D:*to)
10        PUSH A
11        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

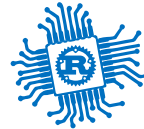
a working example for the previous code

Start



Real World Microcontrollers

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

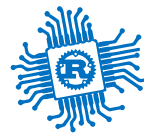


Bibliography

for this section

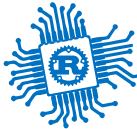
Joseph Yiu, *The Definitive Guide to ARM® Cortex®-M0 and Cortex-M0+ Processors, 2nd Edition*

- 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>

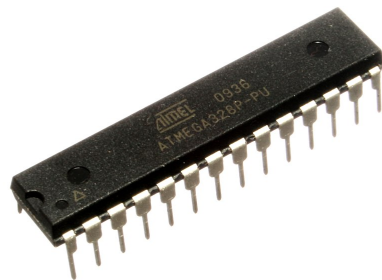




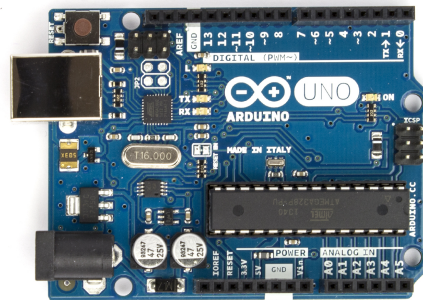
AVR

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

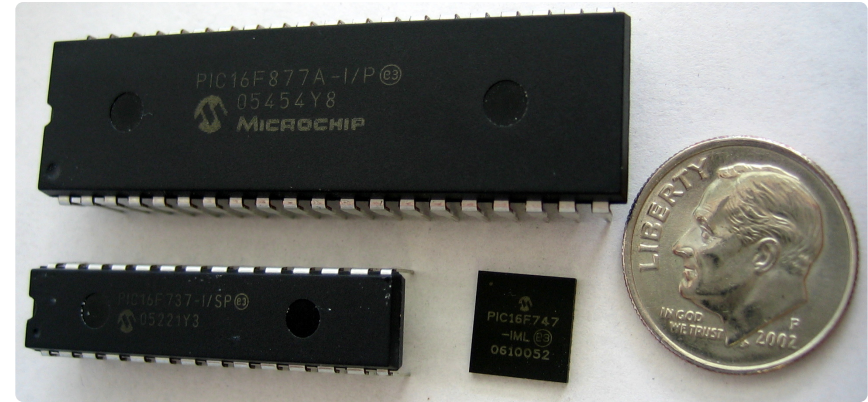




PIC

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>

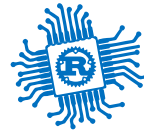




TriCore



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

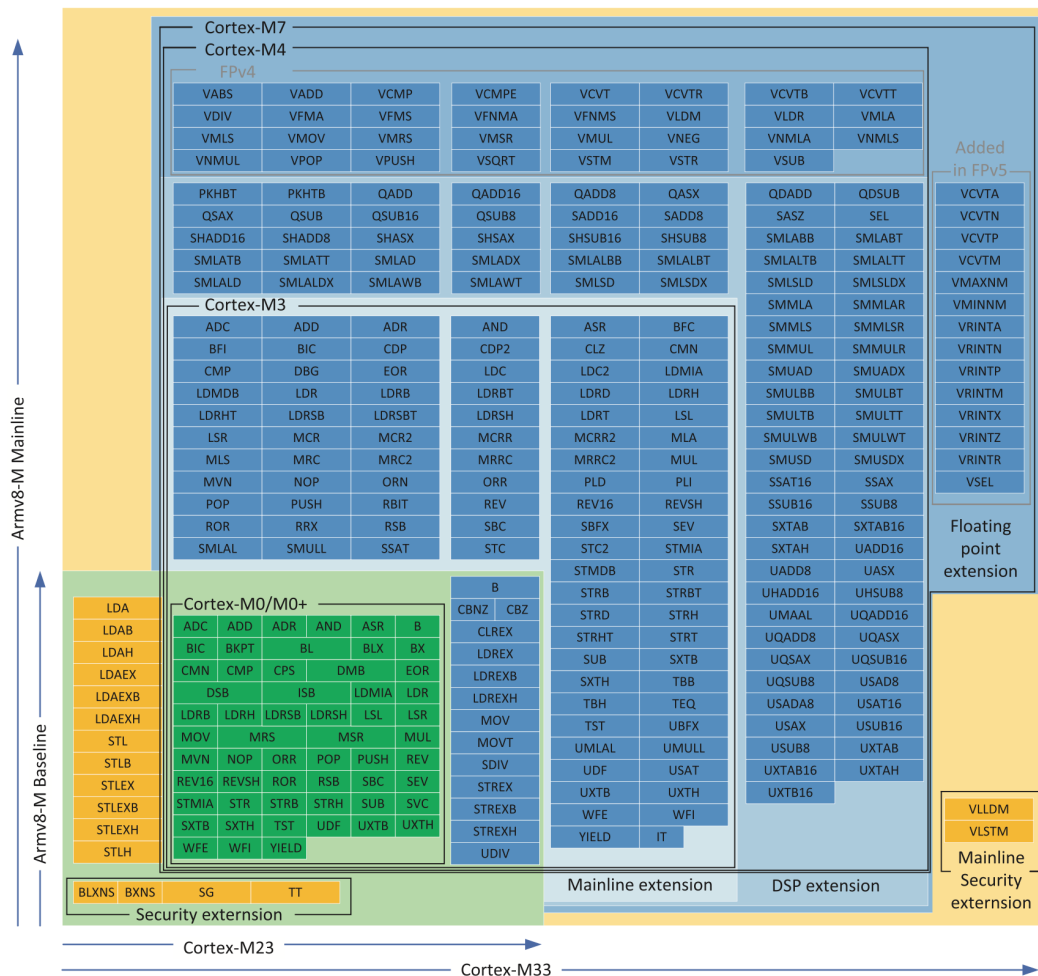
arm

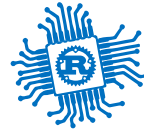
Vendor	Qualcomm, NXP, Nordic Semiconductor, Broadcom, Raspberry Pi
ISA	ARMv6-M (Thumb and some Thumb-2) ARMv7-M (Thumb and Thumb-2) ARMv8-M (Thumb and Thumb-2)
Word	32
Frequency	1 - 900 MHz
Storage	up to a few MB
Variants	<i>M0, M0+, M3, M4, M7, M23, M33</i>



Fun Facts

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



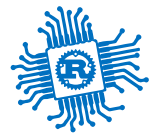


RISC-V rv32i(a)mc

Fifth generation of RISC ISA

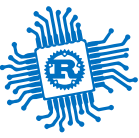


Authors	University of California, Berkeley
Vendor	Espressif System
ISA	rv32i(a)mc
Word	32 bit
Frequency	1 - 200 MHz
Storage	4 - 256 KB
Variants	<i>rv32imc, rv32iamc</i>



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*



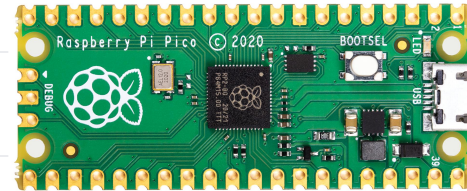
RP2040

the MCU

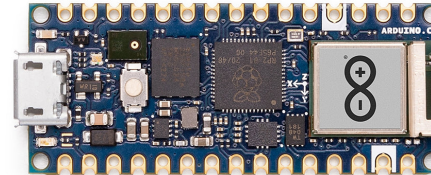
Boards

that use RP2040

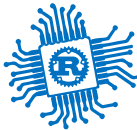
Raspberry Pi Pico (W)



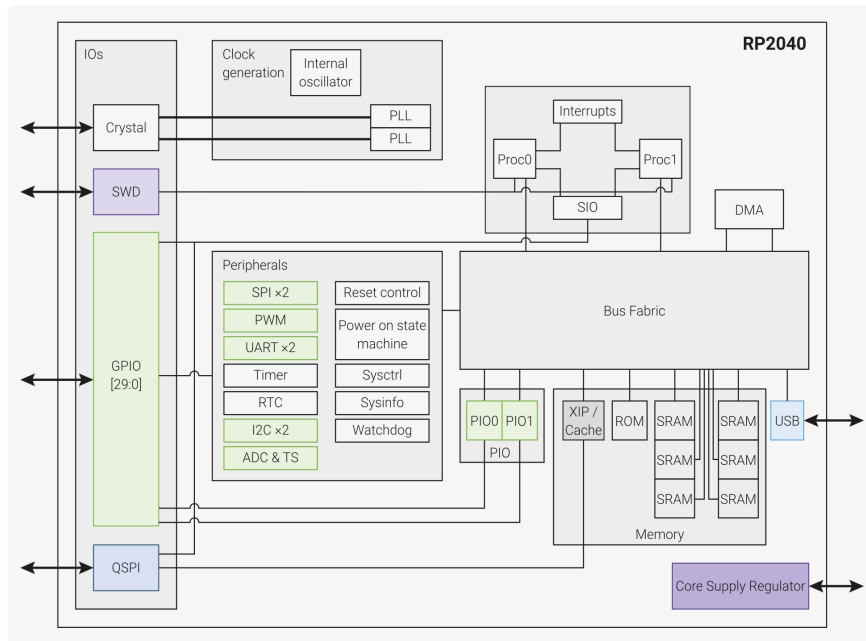
Arduino Nano RP2040 Connect



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



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

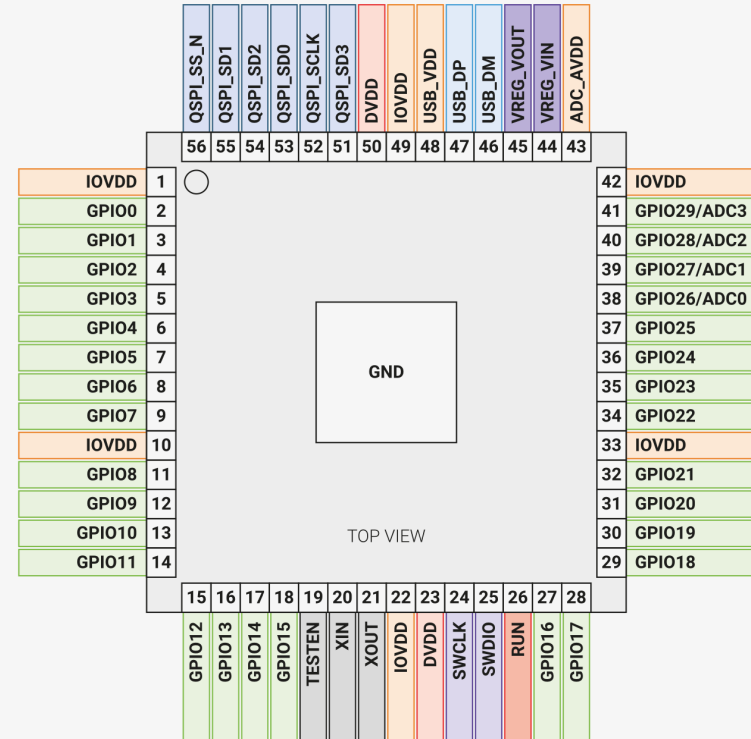
RTC Real Time Clock

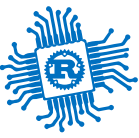
I2C Inter-Integrated Circuit

PIO Programmable Input/Output

Pins

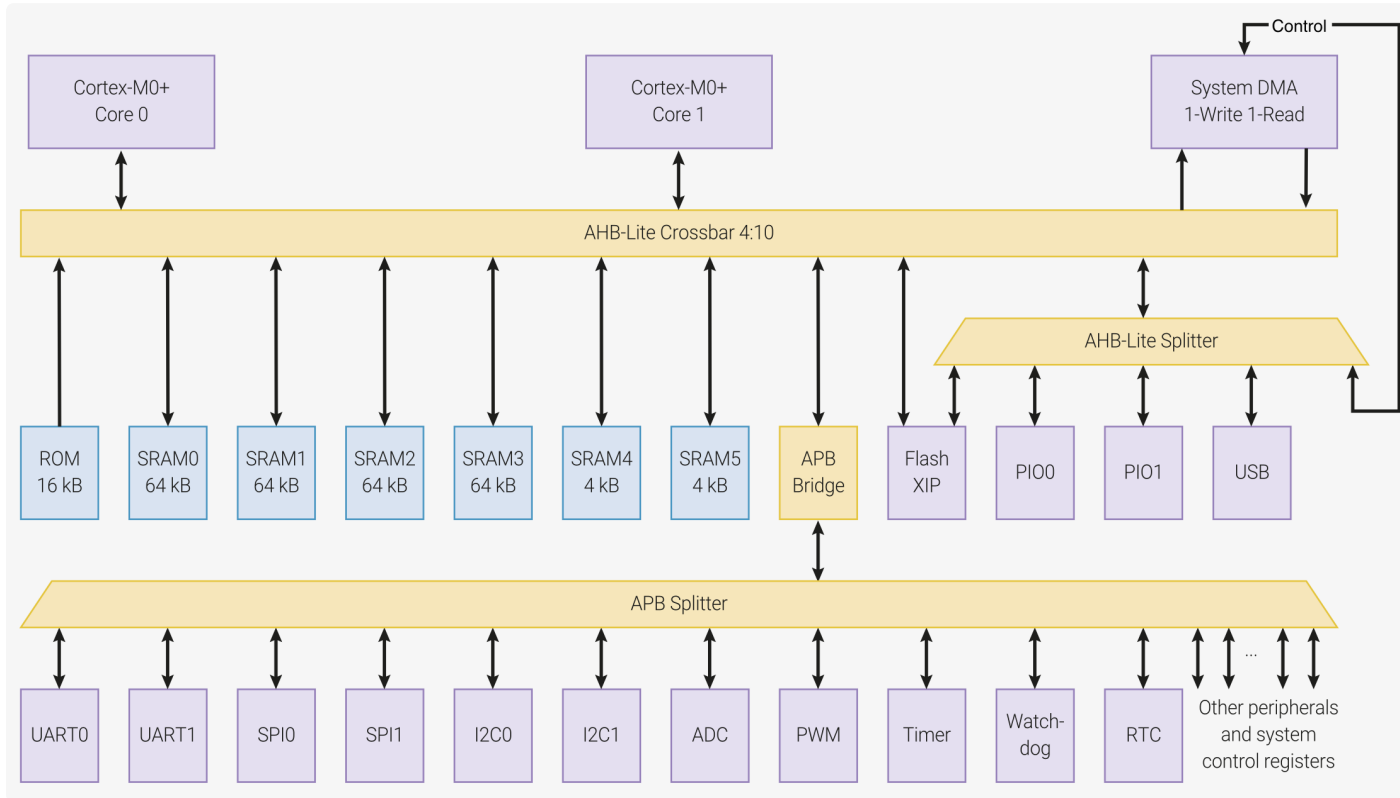
have multiple functions

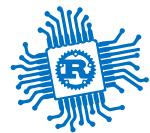




The Bus

that interconnects the cores with the peripherals





STM32U545RE

ARM Cortex-M33, built by STMicroelectronics



Bibliography

for this section

STMicroelectronics, STM32U5 Reference Manual

- Chapter 2 - *Memory and bus architecture*
 - Section 2.1 - *System architecture*

STMicroelectronics, STM32U5 Datasheet

- Chapter 2 - *"Description"*
- Chapter 4 - *"Pinout, pin description, and alternate function"*



STM32U545RE

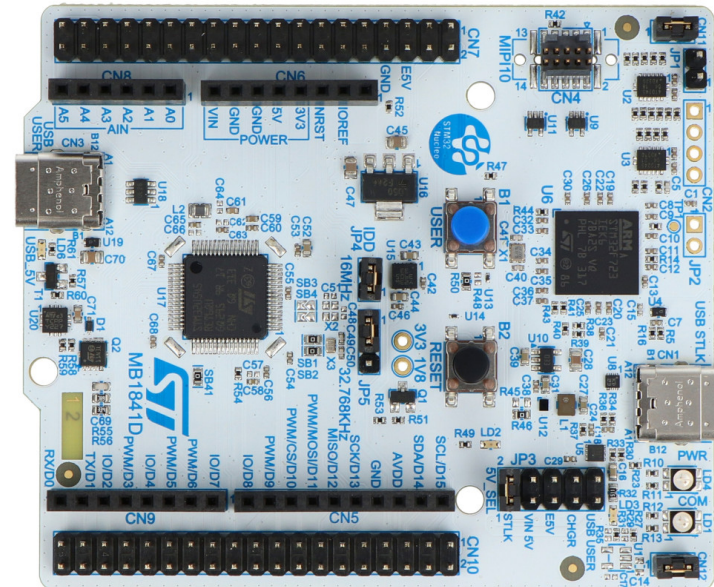
the MCU

Board

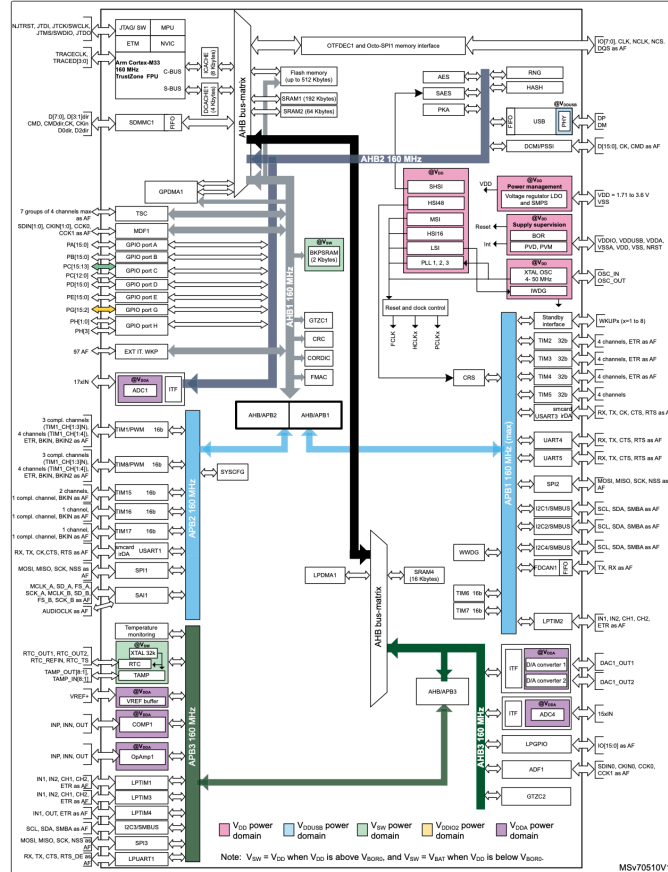
that use STM32U545RE

Nucleo U545RE-Q

Vendor	STMicroelectronics
Variant	ARM Cortex-M33
ISA	ARMv8-M
Cores	1
Word	32 bit
Frequency	up to 160 MHz
RAM	272 KB



The Chip



Peripherals

Parallel interface

**FSMC 8-/16-bit
(TFT-LCD, SRAM,
NOR, NAND)**

Timers

19 timers including:
2 x 16-bit advanced
motor control timers
4 x ULP timers
5 x 16-bit timers
4 x 32-bit timers

I/Os

Touch-sensing controller
Camera Interface

Connectivity

USB Host/Device
1 x SD/SDIO/MMC, 3 x SPI,
4 x I²C, CAN FD,
1 x Octo-SPI,
4 x USART + 1 x LPUART

Digital

AES (256-bit),
SHA-1, SHA-256,
TRNG, PKA, 1 x SAI,
1 x MDF, 2 x ADF

Analog

1x 14-bit ADC 2 MSPS,
1x 12-bit ADC 2 MSPS,
2 x DAC, 2 x comparators,
1 x op amps
1 x temperature sensor

Arm® Cortex®-M33 CPU

160 MHz

TrustZone®

FPU

MPU

ETM

LPDMA

ART Accelerator™

CORDIC

FMAC

Up to 512-Kbyte

Flash memory

Dual Bank

274-Kbyte RAM

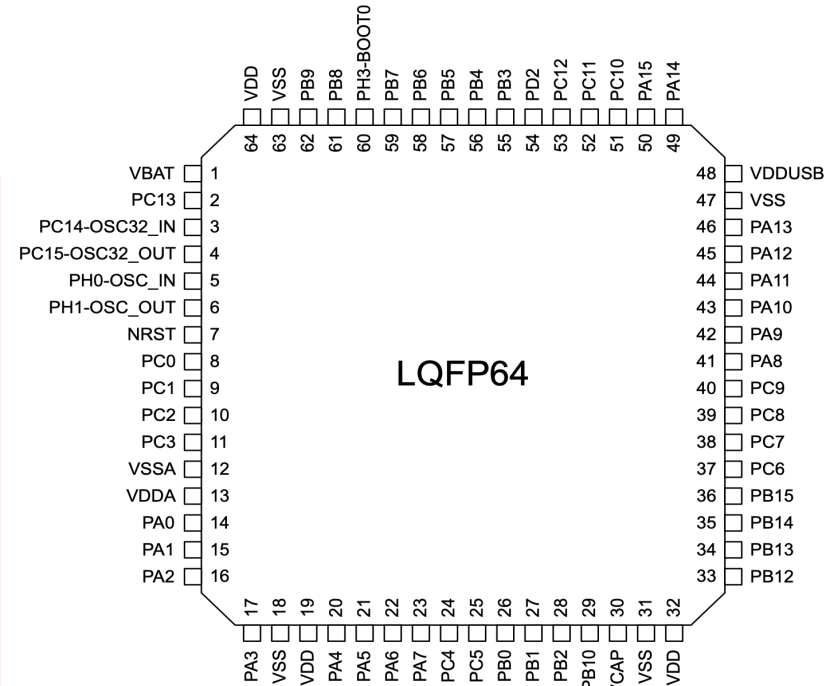
Datasheet STM32U545RE



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 diagram illustrates the internal architecture of the STM32F405, showing the hierarchy of the bus system and the connection of various peripherals.

Top Level: The **Arm Cortex-M33** is connected to the **AHB bus-matrix** via its **C-BUS** and **S-BUS**. The **GPDMA1** is also connected to the AHB bus-matrix.

AHB bus-matrix: This central bus connects to several peripheral blocks:

- AHB/APB2** (x3): These blocks are connected to the AHB bus-matrix and the APB2 bus.
- USB**: Connected to the AHB bus-matrix.
- ADC1**: Connected to the AHB bus-matrix.
- GPIO Ports**: Connected to the AHB bus-matrix.
- Other AHB**: A dashed box representing other AHB peripherals.
- Flash memory 512 Kbytes**: Connected to the AHB bus-matrix.
- SRAM 1 192 Kbytes** (x2): Connected to the AHB bus-matrix.

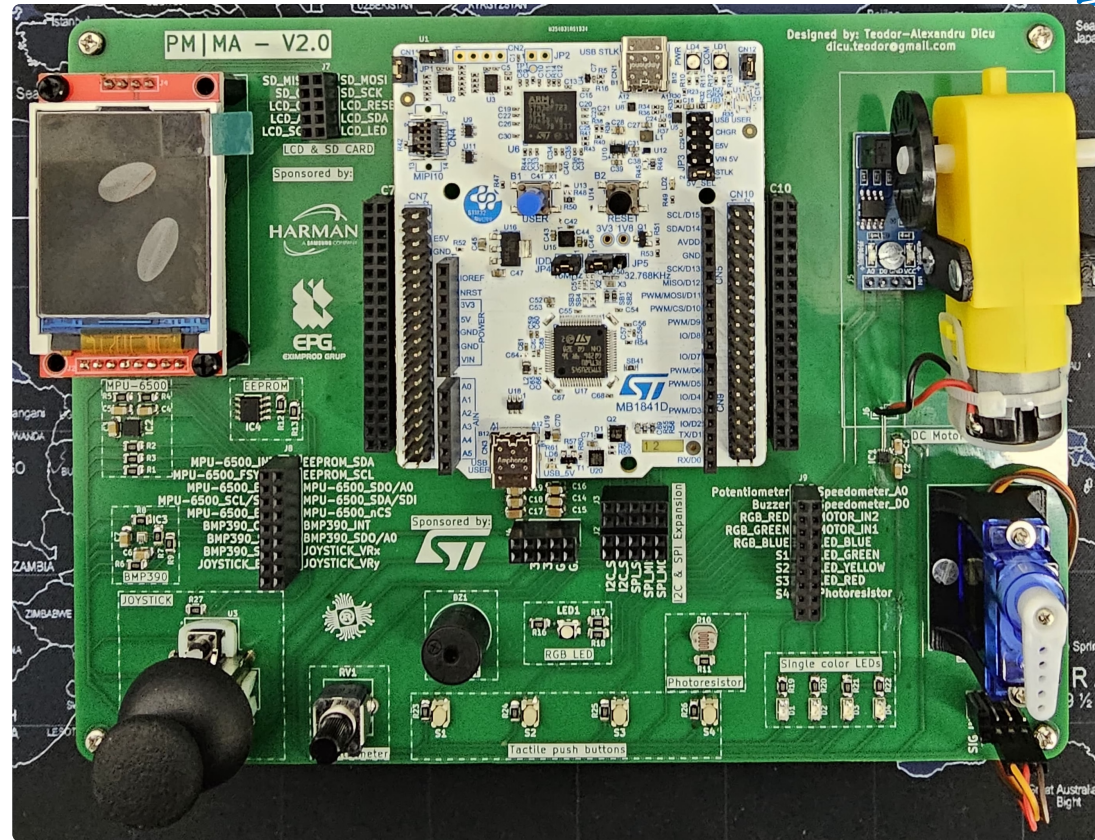
APB Buses: The AHB bus-matrix connects to three APB buses:

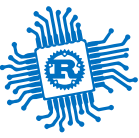
- APB3:** Connected to the AHB/APB2 blocks and the **Other APB3** (dashed box).
- APB2:** Connected to the AHB/APB2 blocks, the **Other APB2** (dashed box), and the **TIM2/PWM**, **USART1**, and **SPI1** peripherals.
- APB1:** Connected to the **SPI2**, **USART3**, **I2C1**, and **Other APB1** (dashed box) peripherals.



Lab Board

- Nucleo U545RE-Q Slot / Board
- 4 buttons
- 5 LEDs
- potentiometer
- buzzer
- photoresistor
- I2C EEPROM
- MPU-6500 accelerometer & Gyro
- BMP 390 Pressure sensor
- SPI LCD Display
- SD Card Reader
- servo connectors
- stepper motor





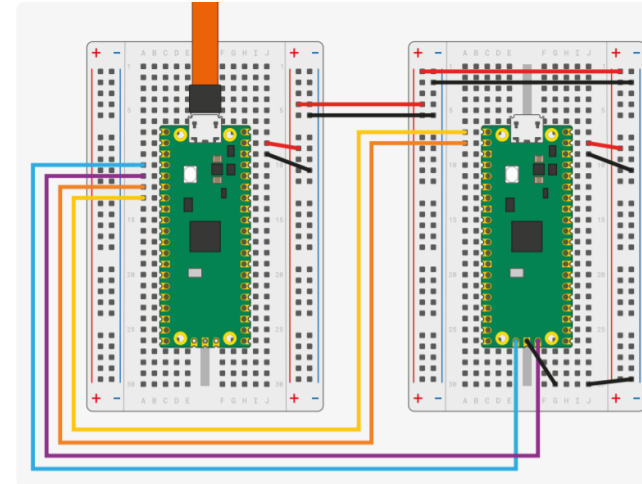
Project

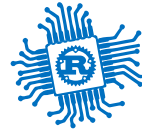
suggested hardware

- the hardware should not cost more than 150 RON
- STM32 Nucleo F446RE or Nucleo U545RE-Q board (include debuggers)
- Raspberry Pi Pico with a debugger

Raspberry Pi Pico 2W + Debug Probe

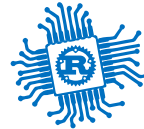
Raspberry Pi Pico 2W + Raspberry Pi Pico 1





Bitwise Ops

How to set and clear bits



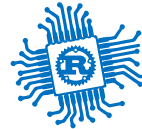
Set bit

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 }
```



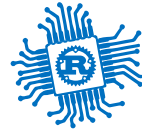
Clear bit

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 }
```



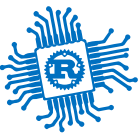
Flip bit

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

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

Flip multiple bits

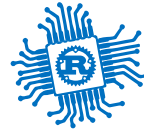
```
1 fn flip_bits(register: usize, bits: usize) -> usize {
2     // assume register is 0b1000, bits is 0b0111
3     // 0b1000 ^ 0b0111 is 0b1111
4     register ^ bits
5 }
```



Let's see a combined operation for value extraction

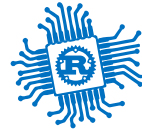
- We presume an 32 bits ID = `0b1100_1010_1111_1100_0000_1111_0110_1101`
- And want to extract a portion `0b1100_1010_1111_1100_0000_1111_0110_1101`

```
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::
```



Conclusion

we talked about

- How a processor functions
- Microcontrollers (MCU) / Microprocessors (CPU)
- Microcontroller architectures
- ARM Cortex-M
- RP2040 and STM32U545RE