Introduction to ARM

about ARM microcontrollers

Copyright © Wyliodrin SRL 2024, licensed under CC BY-SA 4.0.

Introduction to ARM

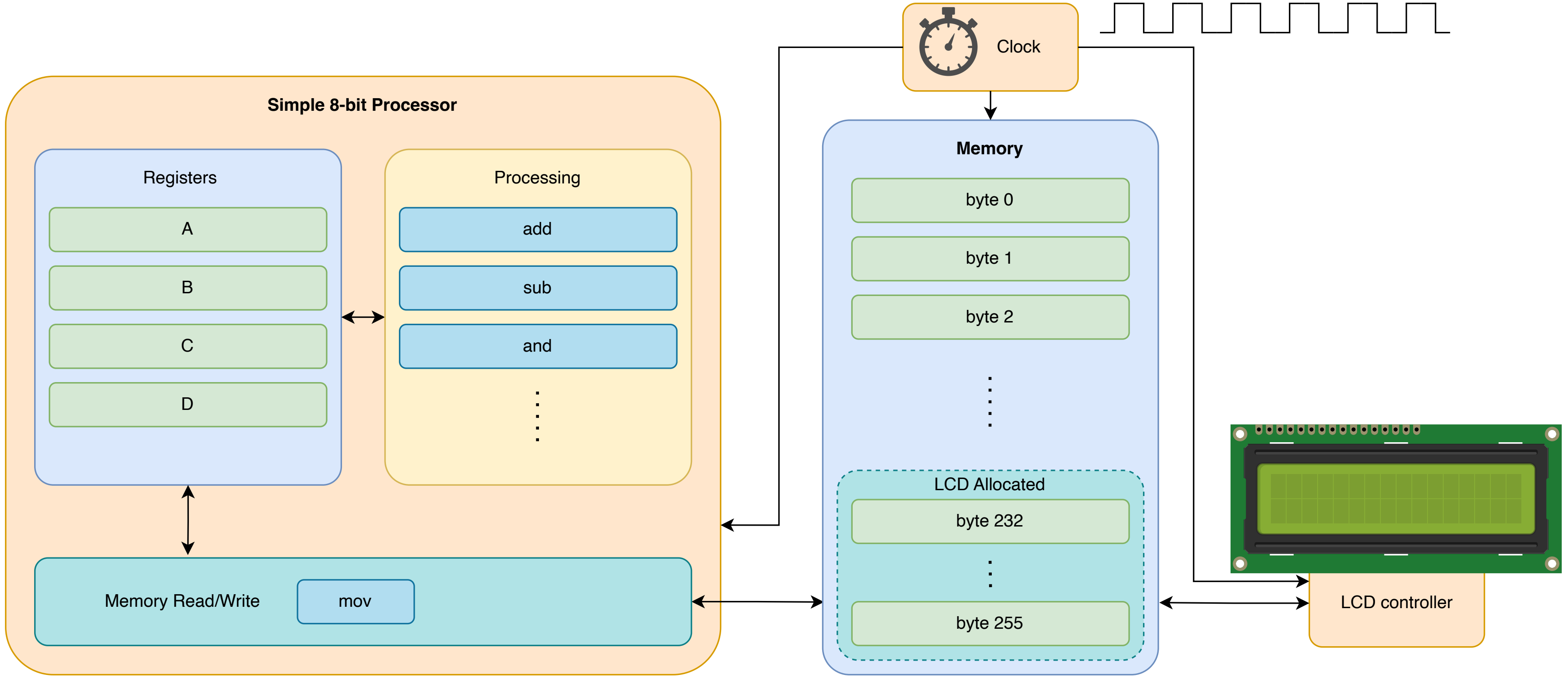
- Memory Mapped I/O
- Exceptions
- Memory Protection

MMIO

Memory Mapped Input Output

8 bit processor

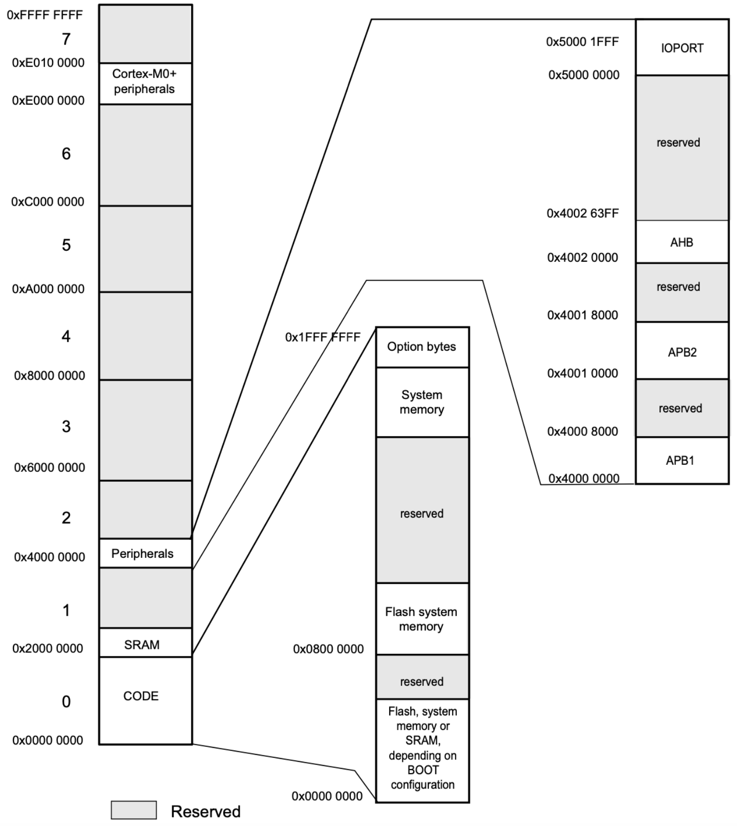
a simple 8 bit processor with a text display



STM32L0x2

A real MCU

Cortex-M0+ Peripherals	MCU's *settings* and internal peripherals, available at the same address on all M0+
Peripherals	GPIO, USART, SPI, I2C, USB, etc
Flash	The storage space
SRAM	RAM memory
@0x0000_0000	Alias for SRAM or Flash



System Control Registers

@0xe000_0000

Compute the actual address

- 0xe000_0000 + Offset

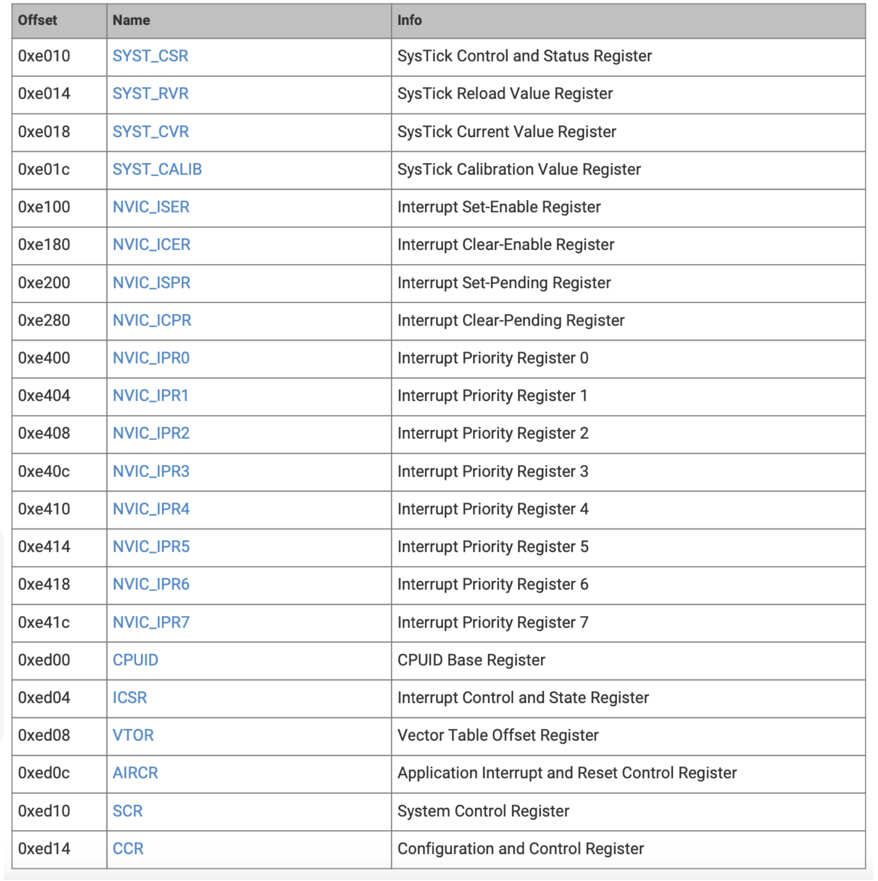
Examples:

- SYST_CSR: **0xe000_e010** (*0xe000_0000* + *0xe010*)
- CPUID: **0xe000_ed00** (*0xe000_0000* + *0xed00*)

```rust
const SYS_CTRL: usize = 0xe000_0000;
const CPUID: usize = 0xed00;

let cpuid_reg = (SYS_CTRL + CPUID) as *const u32;
let cpuid_value = unsafe { *cpuid_reg };
```

⚠ Compilers optimize code and processors use cache!



Offset	Name	Info
0xe010	SYST_CSR	SysTick Control and Status Register
0xe014	SYST_RVR	SysTick Reload Value Register
0xe018	SYST_CVR	SysTick Current Value Register
0xe01c	SYST_CALIB	SysTick Calibration Value Register
0xe100	NVIC_ISER	Interrupt Set-Enable Register
0xe180	NVIC_ICER	Interrupt Clear-Enable Register
0xe200	NVIC_ISPR	Interrupt Set-Pending Register
0xe280	NVIC_ICPR	Interrupt Clear-Pending Register
0xe400	NVIC_IPR0	Interrupt Priority Register 0
0xe404	NVIC_IPR1	Interrupt Priority Register 1
0xe408	NVIC_IPR2	Interrupt Priority Register 2
0xe40c	NVIC_IPR3	Interrupt Priority Register 3
0xe410	NVIC_IPR4	Interrupt Priority Register 4
0xe414	NVIC_IPR5	Interrupt Priority Register 5
0xe418	NVIC_IPR6	Interrupt Priority Register 6
0xe41c	NVIC_IPR7	Interrupt Priority Register 7
0xed00	CPUID	CPUID Base Register
0xed04	ICSR	Interrupt Control and State Register
0xed08	VTOR	Vector Table Offset Register
0xed0c	AIRCR	Application Interrupt and Reset Control Register
0xed10	SCR	System Control Register
0xed14	CCR	Configuration and Control Register

Compiler Optimization

compilers optimize code

Write bytes to the `UART` (serial port) data register

```rust
const UART_TX: *const u8 = 0x400_3400;
for b in b"Hello, World".iter() {
    unsafe { UART_TX.write(*b); }
}
```

1. The compiler does not know that `UART_TX` is a register and uses it as a memory address.
2. Writing several values to the same memory address will result in having the last value stored at that address.
3. The compiler optimizes the code write the value

```rust
const UART_TX: *const u8 = 0x400_3400;
unsafe { UART_TX.write(b'd'); }
```

.

No Compiler Optimization

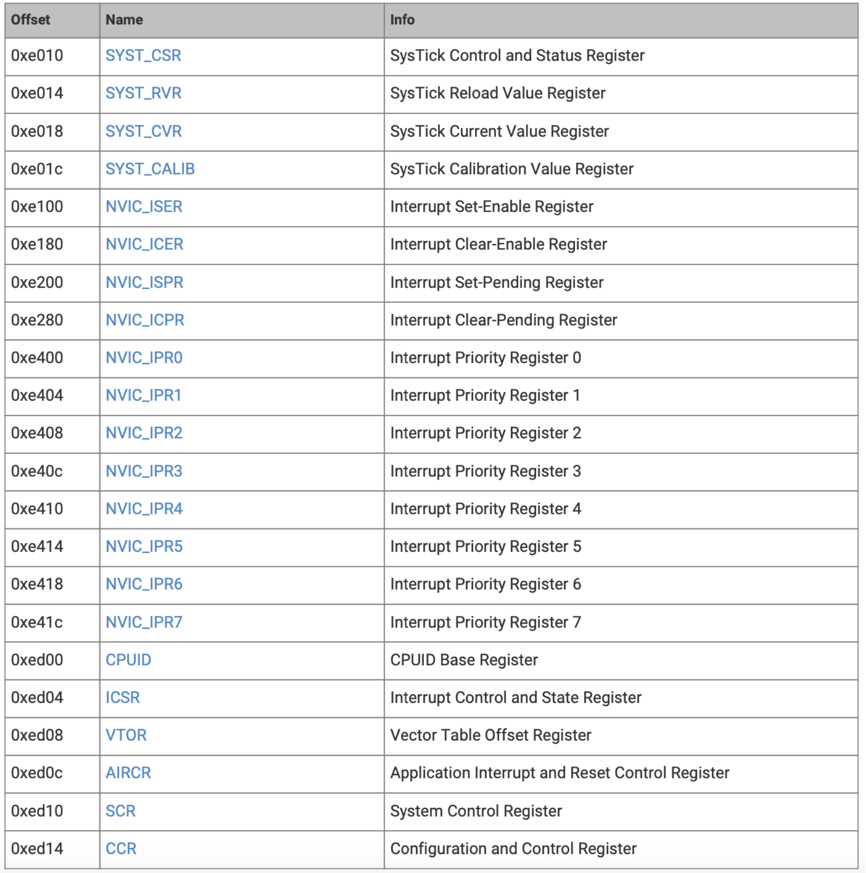
CPUID: **0xe000_ed00** *(0xe000_0000 + 0xed00)*

```rust
use core::ptr::read_volatile;

const SYS_CTRL: usize = 0xe000_0000;
const CPUID: usize = 0xed00;

let cpuid_reg = (SYS_CTRL + CPUID) as *const u32;
unsafe {
  // avoid compiler optimization
    read_volatile(cpuid_reg)
}
```

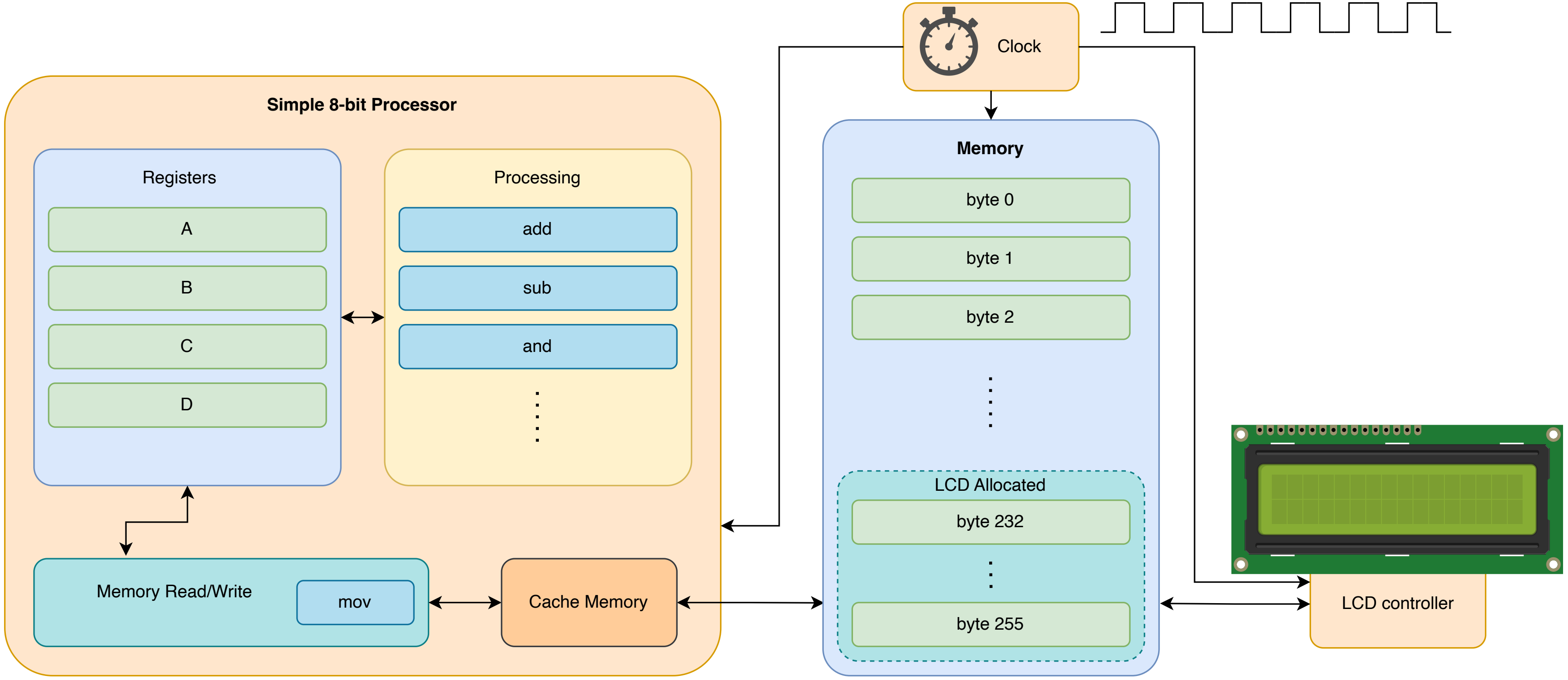
`read_volatile`, `write_volatile`	**no** compiler **optimization**
`read`, `write`, `*p`	**use** compiler **optimization**



Offset	Name	Info
0xe010	SYST_CSR	SysTick Control and Status Register
0xe014	SYST_RVR	SysTick Reload Value Register
0xe018	SYST_CVR	SysTick Current Value Register
0xe01c	SYST_CALIB	SysTick Calibration Value Register
0xe100	NVIC_ISER	Interrupt Set-Enable Register
0xe180	NVIC_ICER	Interrupt Clear-Enable Register
0xe200	NVIC_ISPR	Interrupt Set-Pending Register
0xe280	NVIC_ICPR	Interrupt Clear-Pending Register
0xe400	NVIC_IPR0	Interrupt Priority Register 0
0xe404	NVIC_IPR1	Interrupt Priority Register 1
0xe408	NVIC_IPR2	Interrupt Priority Register 2
0xe40c	NVIC_IPR3	Interrupt Priority Register 3
0xe410	NVIC_IPR4	Interrupt Priority Register 4
0xe414	NVIC_IPR5	Interrupt Priority Register 5
0xe418	NVIC_IPR6	Interrupt Priority Register 6
0xe41c	NVIC_IPR7	Interrupt Priority Register 7
0xed00	CPUID	CPUID Base Register
0xed04	ICSR	Interrupt Control and State Register
0xed08	VTOR	Vector Table Offset Register
0xed0c	AIRCR	Application Interrupt and Reset Control Register
0xed10	SCR	System Control Register
0xed14	CCR	Configuration and Control Register

8 bit processor

with cache



No Cache or Flush Cache

- Cache types:
  - *write-through* - data is written to the cache and to the main memory (bus)
  - *write-back* - data is written to the cache and later to the main memory (bus)
- few Cortex-M MCUs have cache
- the Memory Mapped I/O region is set as *nocache*
- for chips that use cache
  - *nocache* regions have to be set manually (if MCU knows)
  - or, the cache has to be flushed before a `volatile_read` and after a `volatile_write`
  - beware DMA controllers that can’t see the cache contents

AIRCR

Application Interrupt and Reset Control Register

```rust
use core::ptr::read_volatile;
use core::ptr::write_volatile;

const SYS_CTRL: usize = 0xe000_0000;
const AIRCR: usize = 0xed0c;

const VECTKEY: u32 = 16;
const SYSRESETREQ: u32 = 2;

let aircr_register = (SYS_CTRL + AIRCR) as *mut u32;
let mut aircr_value = unsafe {
    read_volatile(aircr_register)
};

aircr_value = aircr_value & ~(0x1111 << VECTKEY);
aircr_value = aircr_value | (0x05fa << VECTKEY);
aircr_value = aircr_value | (1 << SYSRESETREQ);

unsafe {
    write_volatile(aircr_register, aircr_value);
}
```

AIRCR Register

Offset: 0xed0c

Bits	Name	Description	Type	Reset
31:16	VECTKEY	Register key: Reads as Unknown On writes, write 0x05FA to VECTKEY, otherwise the write is ignored.	RW	0x0000
15	ENDIANESS	Data endianness implemented: 0 = Little-endian.	RO	0x0
14:3	Reserved.	-	-	-

Bits	Name	Description	Type	Reset
2	SYSRESETREQ	Writing 1 to this bit causes the SYSRESETREQ signal to the outer system to be asserted to request a reset. The intention is to force a large system reset of all major components except for debug. The C_HALT bit in the DHCSR is cleared as a result of the system reset requested. The debugger does not lose contact with the device.	RW	0x0
1	VECTCLRACTIVE	Clears all active state information for fixed and configurable exceptions. This bit: is self-clearing, can only be set by the DAP when the core is halted. When set: clears all active exception status of the processor, forces a return to Thread mode, forces an IPSR of 0. A debugger must re-initialize the stack.	RW	0x0
0	Reserved.	-	-	-

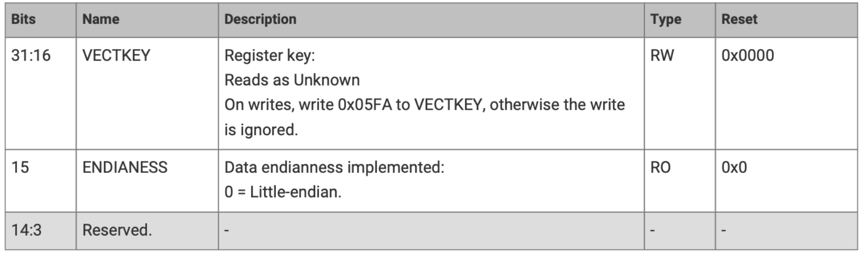
Read and Write

they do stuff

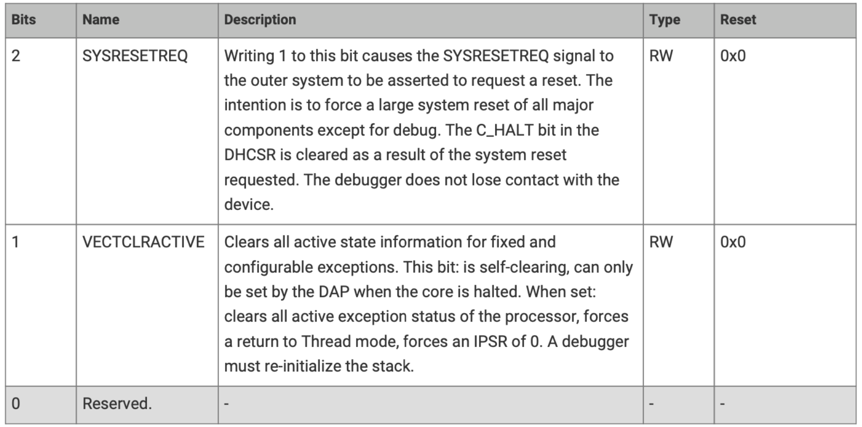
- Read
  - reads the value of a register
  - might ask the peripheral to do something
- Write
  - writes the value to a register
  - might ask the peripheral to do something
    - SYSRESETREQ

AIRCR Register

Offset: 0xed0c



Bits	Name	Description	Type	Reset
31:16	VECTKEY	Register key: Reads as Unknown On writes, write 0x05FA to VECTKEY, otherwise the write is ignored.	RW	0x0000
15	ENDIANESS	Data endianness implemented: 0 = Little-endian.	RO	0x0
14:3	Reserved.	-	-	-



Bits	Name	Description	Type	Reset
2	SYSRESETREQ	Writing 1 to this bit causes the SYSRESETREQ signal to the outer system to be asserted to request a reset. The intention is to force a large system reset of all major components except for debug. The C_HALT bit in the DHCSR is cleared as a result of the system reset requested. The debugger does not lose contact with the device.	RW	0x0
1	VECTCLRACTIVE	Clears all active state information for fixed and configurable exceptions. This bit: is self-clearing, can only be set by the DAP when the core is halted. When set: clears all active exception status of the processor, forces a return to Thread mode, forces an IPSR of 0. A debugger must re-initialize the stack.	RW	0x0
0	Reserved.	-	-	-

SVD XML File

System View Description

```xml
<device schemaVersion="1.1"
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance" xs:noNamespaceSchemaLocation="CMSIS-SVD.xsd">
  <name>RP2040</name>
  <peripherals>
    <name>PPB</name>
    <baseAddress>0xe0000000</baseAddress>
    <register>
      <name>CPUID</name>
      <addressOffset>0xed00</addressOffset>
      <resetValue>0x410cc601</resetValue>
      <fields>
        <field>
          <name>IMPLEMENTER</name>
          <description>Implementor code: 0x41 = ARM</description>
          <bitRange>[31:24]</bitRange>
          <access>read-only</access>
        </field>
        <!-- rest of the fields of the register -->
      </fields>
    </register>
  </peripherals>
</device>
```

Exceptions

for the ARM Cortex-M0+ processor

Bibliography

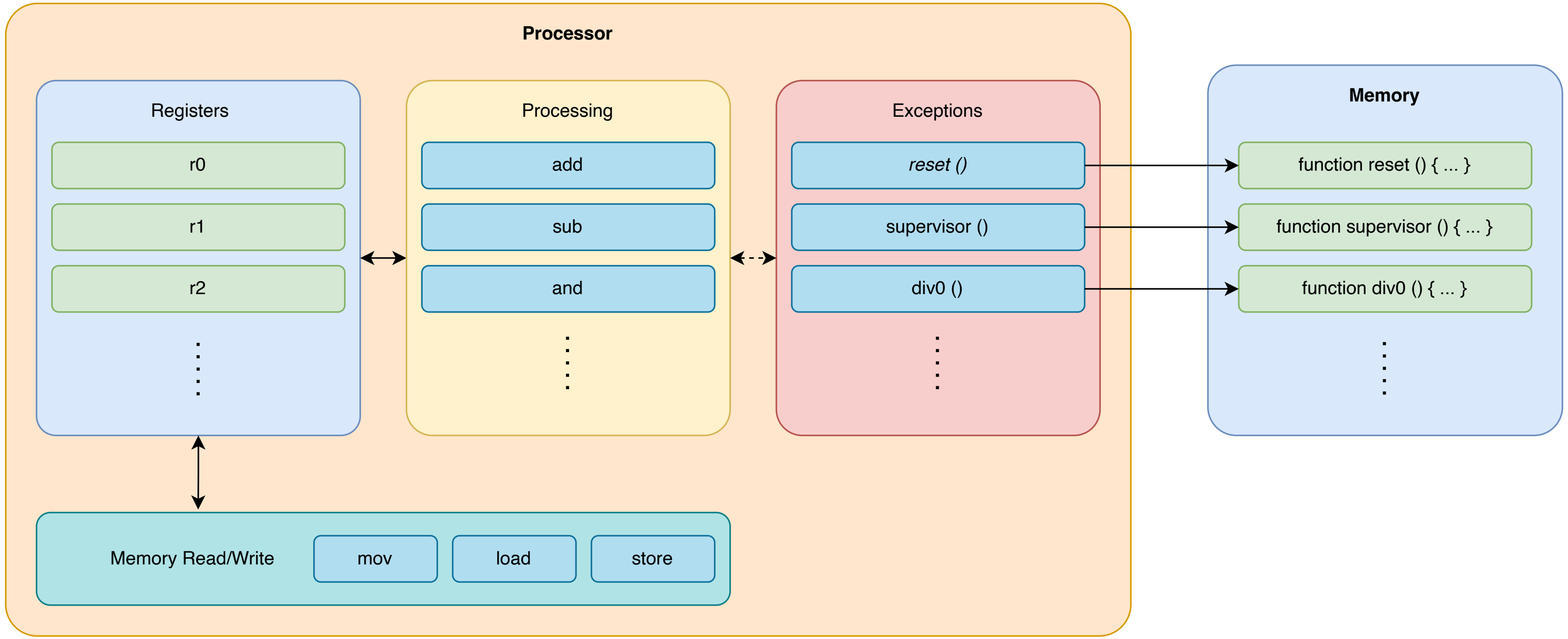
for this section

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

- Chapter 4 - *Architecture*
  - Section 4.4 - *Stack Memory Operations*
  - Section 4.5 - *Exceptions and Interrupts*
- Chapter 8 - *Exceptions and Interrupts*
  - Section 8.1 - *What are Exceptions and Interrupts*
  - Section 8.2 - *Exception types on Cortex-M0 and Cortex-M0+*

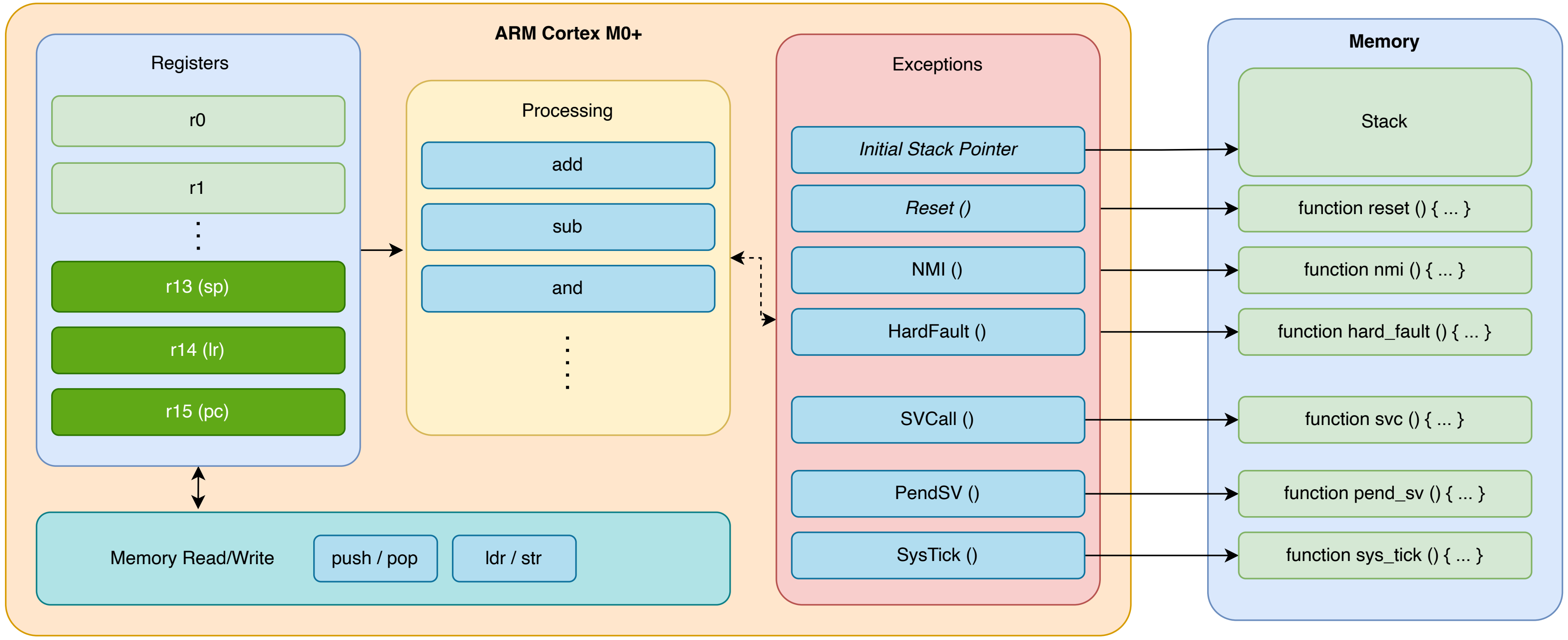
Processor Exceptions

what happens if something does not work as required



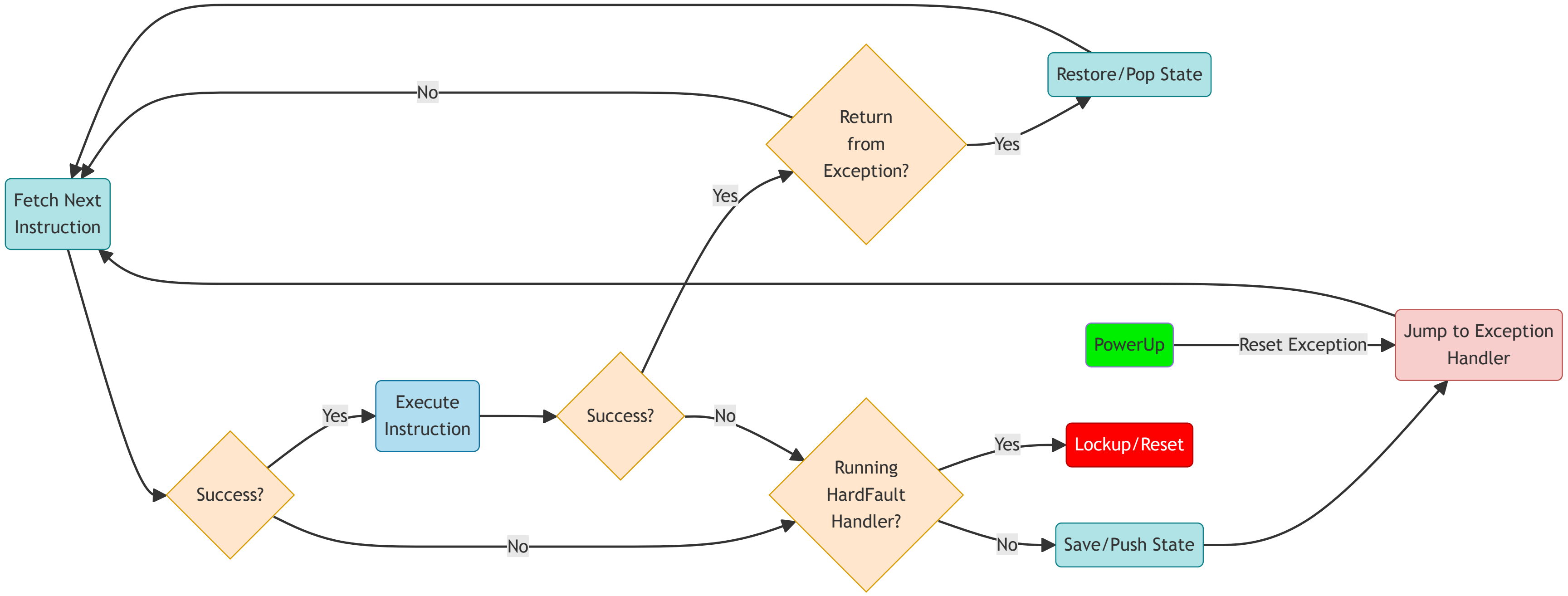
ARM Cortex-M0+ Exceptions

what happens if something does not work as required



Exception (HardFault) Handling

ARM Cortex-M0+ has one **actual exception**, *HardFault*



- the exception table of RP2040 at address 0x1000_0100 (start of the boot area + 4 bytes)
- the processor generates a *Reset* exception when it starts

Interrupts

for ARM Cortex-M0+

Bibliography

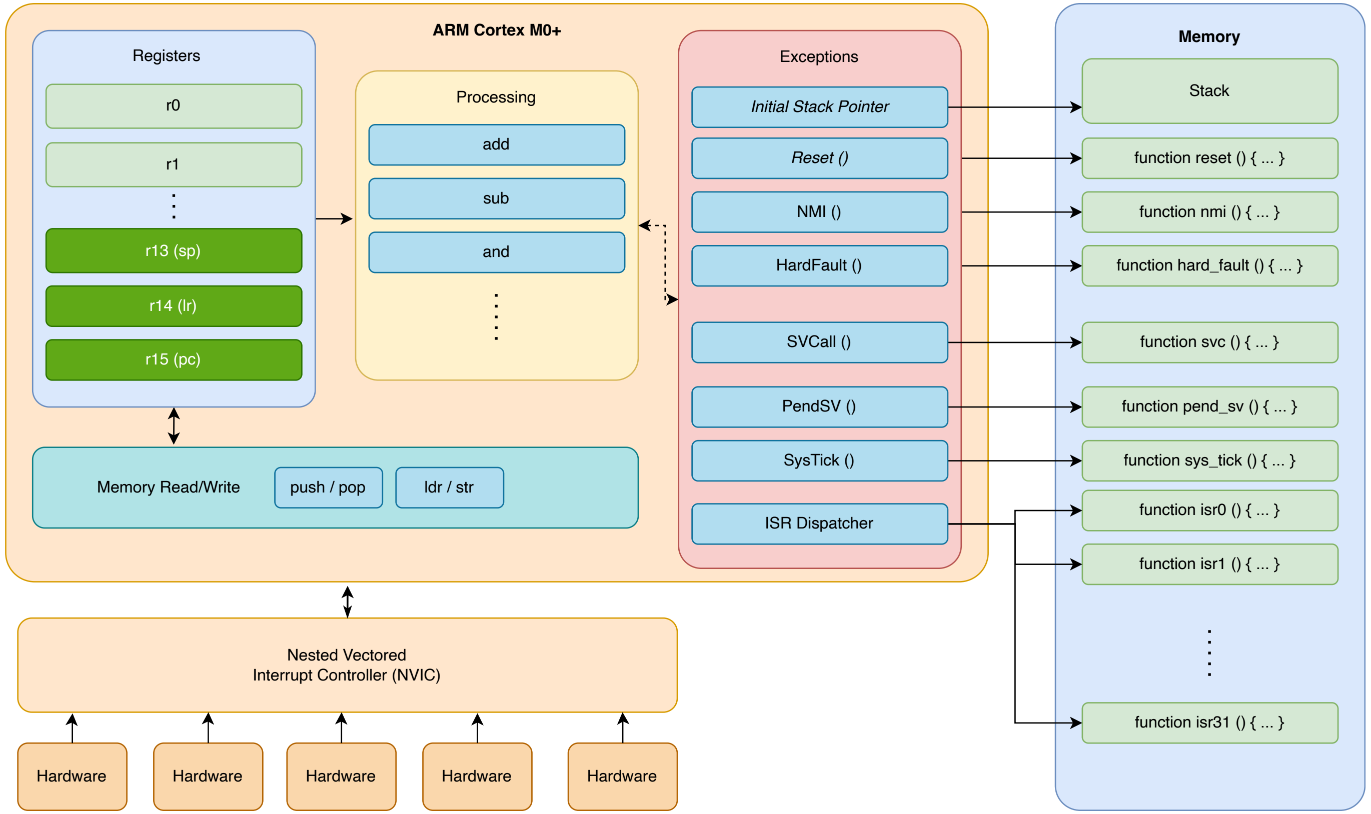
for this section

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

- Chapter 8 - *Exceptions and Interrupts*
  - Section 8.1 - *What are Exceptions and Interrupts*
  - Section 8.3 - *Brief Overview of the NVIC*
  - Section 8.4 - *Definition of Exception Priority Levels*
  - Section 8.5 - *Vector Table*
  - Section 8.6 - *Exception Sequence Overview*
- Chapter 11 - *Fault Handling*
  - Section 11.1 - *Fault Exception Overview*
  - Section 11.2 - *What Can Cause a Fault*
  - Section 11.7 - *Lockup*

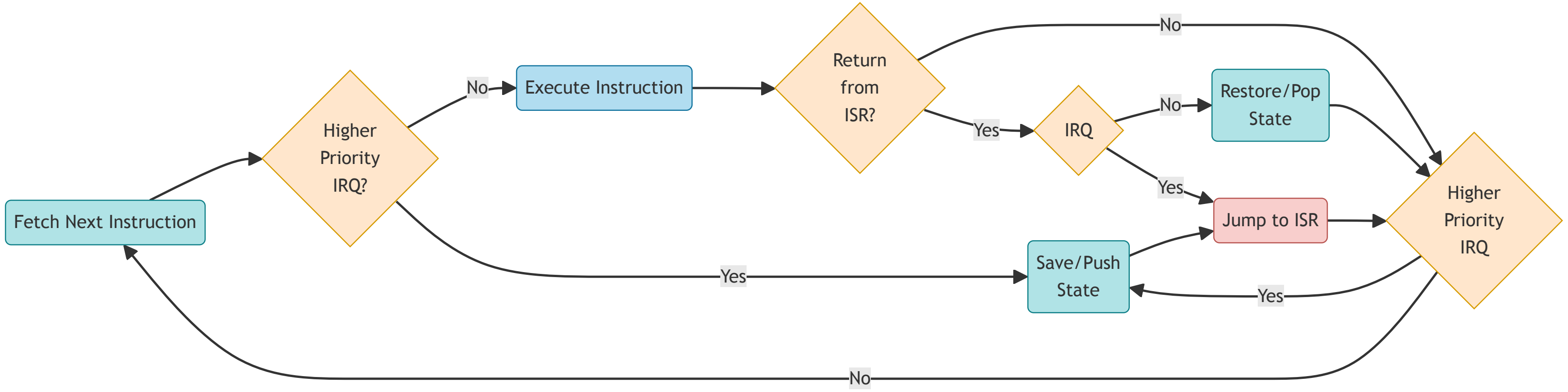
ARM Cortex-M0+ Interrupts

some hardware device notifies the MCU



Interrupt Handling

ARM Cortex-M0+

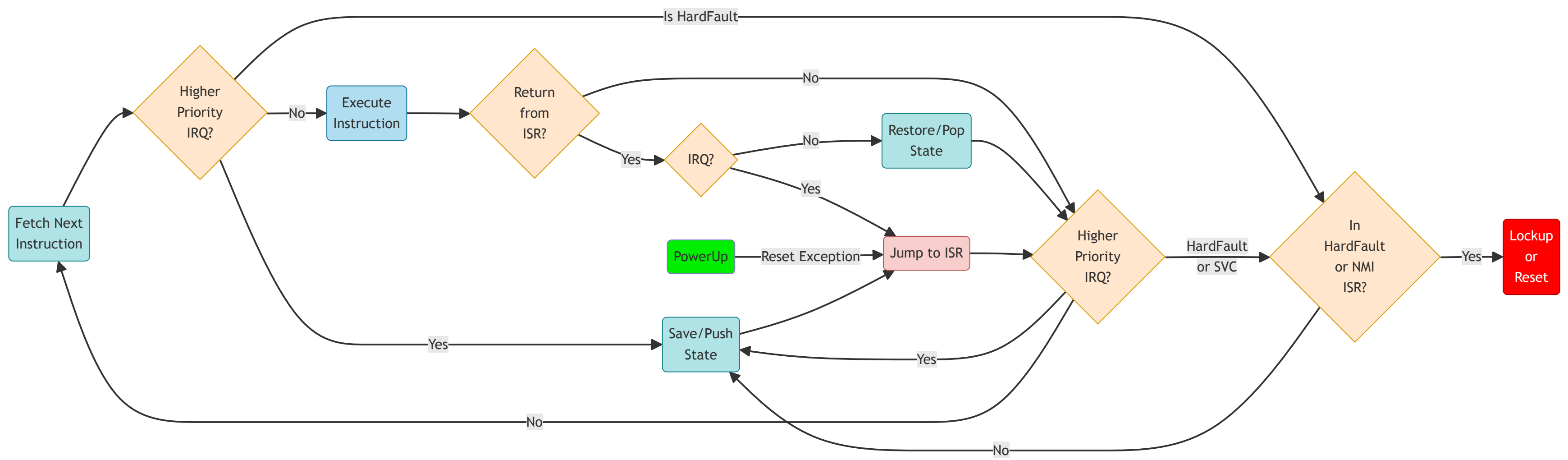


IRQ	Interrupt Request
ISR	Interrupt Service Routine

- the interrupt vector (table) of RP2040 starts at address 0x1000_0040 (after the exceptions table with 15 interrupts)
- ARM Cortex-M0+ has a maximum of 32 interrupt requests (IRQs)

Exceptions are Software Interrupt Requests

with a negative IRQ number and a higher priority



- Reset (-14)
- HardFault (-13)
- SVC (-5)
- PendSV (-2)
- SysTick (-1)

IRQ	Interrupt Source	IRQ	Interrupt Source	IRQ	Interrupt Source	IRQ	Interrupt Source	IRQ	Interrupt Source
0	TIMER_IRQ_0	6	XIP_IRQ	12	DMA_IRQ_1	18	SPI0_IRQ	24	I2C1_IRQ
1	TIMER_IRQ_1	7	PIO0_IRQ_0	13	IO_IRQ_BANK0	19	SPI1_IRQ	25	RTC_IRQ
2	TIMER_IRQ_2	8	PIO0_IRQ_1	14	IO_IRQ_QSPI	20	UART0_IRQ		
3	TIMER_IRQ_3	9	PIO1_IRQ_0	15	SIO_IRQ_PROC0	21	UART1_IRQ		
4	PWM_IRQ_WRAP	10	PIO1_IRQ_1	16	SIO_IRQ_PROC1	22	ADC_IRQ_FIFO		
5	USBCTRL_IRQ	11	DMA_IRQ_0	17	CLOCKS_IRQ	23	I2C0_IRQ		

Boot

of the RP2040

Bibliography

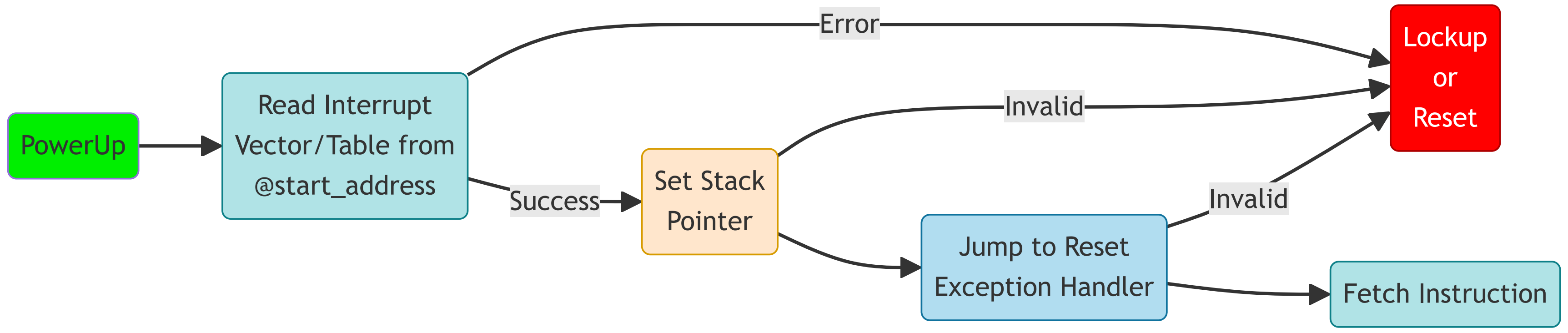
for this section

Raspberry Pi Ltd, *RP2040 Datasheet*

- Chapter 2 - *System Description*
  - Section 2.7 - *Boot sequence*
  - Section 2.8 - *Bootrom*
    - Subsection 2.8.1 - *Processor Controlled Boot Sequence*

Boot

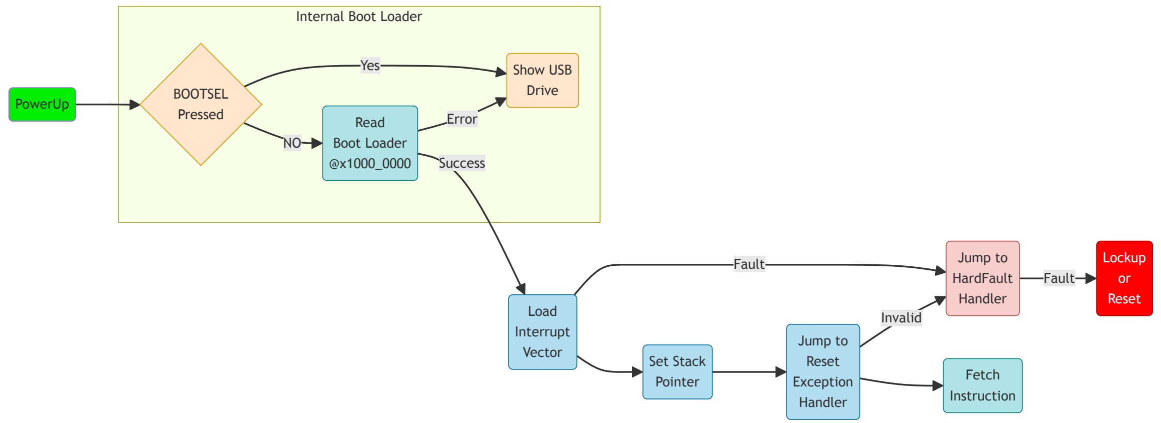
how the ARM Cortex-M0+ starts



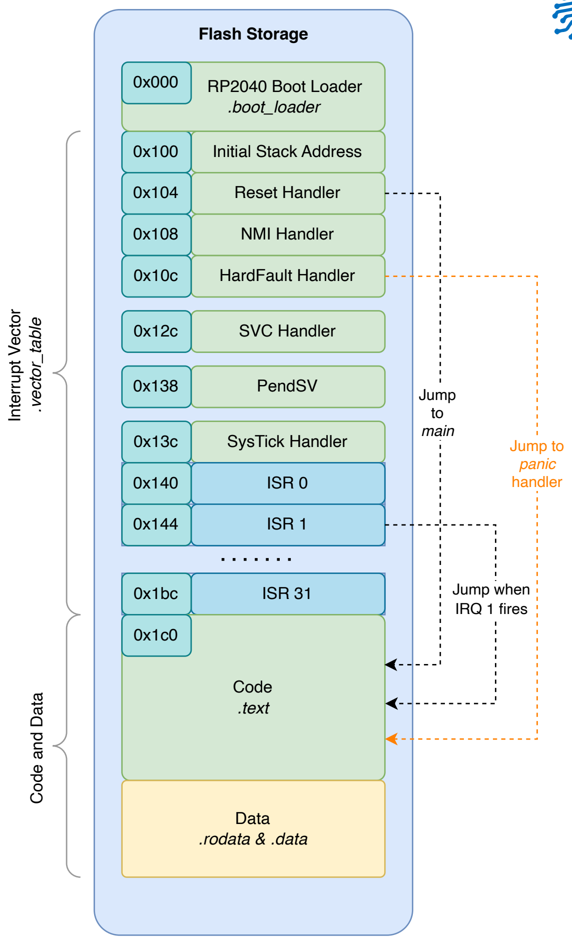
- the *start_address* for RP2040 is 0x1000_0100
- RP2040 has another boot loader that it loads from 0x1000_0000

Boot

The RP2040 boot process



The internal boot loader cannot be overwritten and assures that bricking the device is difficult.



* drawing is not at scale, code and data are significantly greater than the interrupt vector

Memory Protection

ARM: MPU, RISC-V: PMP

Bibliography

for this section

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

- Chapter 12 - *Memory Protection Unit*

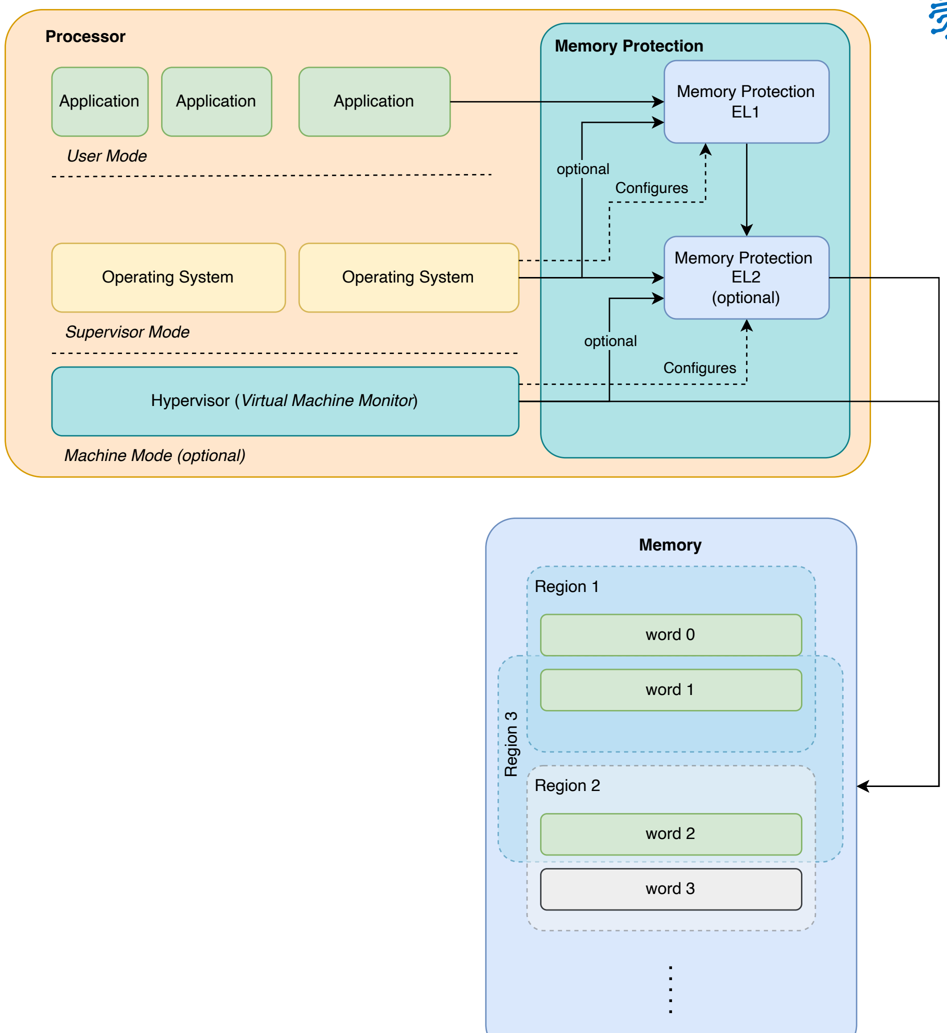
Memory Protection

memory access defined region by region

- **restricts** access to **physical memory**
- uses **physical addresses**

The processor works in three modes:

- **machine** mode (*optional*) - used at boot, allows access to everything
- **supervisor** mode - restricts access to some registers and accesses memory through Memory Protection EL2 (*if machine mode exists*)
- **user** mode - allows only ALU and memory access through Memory Protection



MPU for RP2040

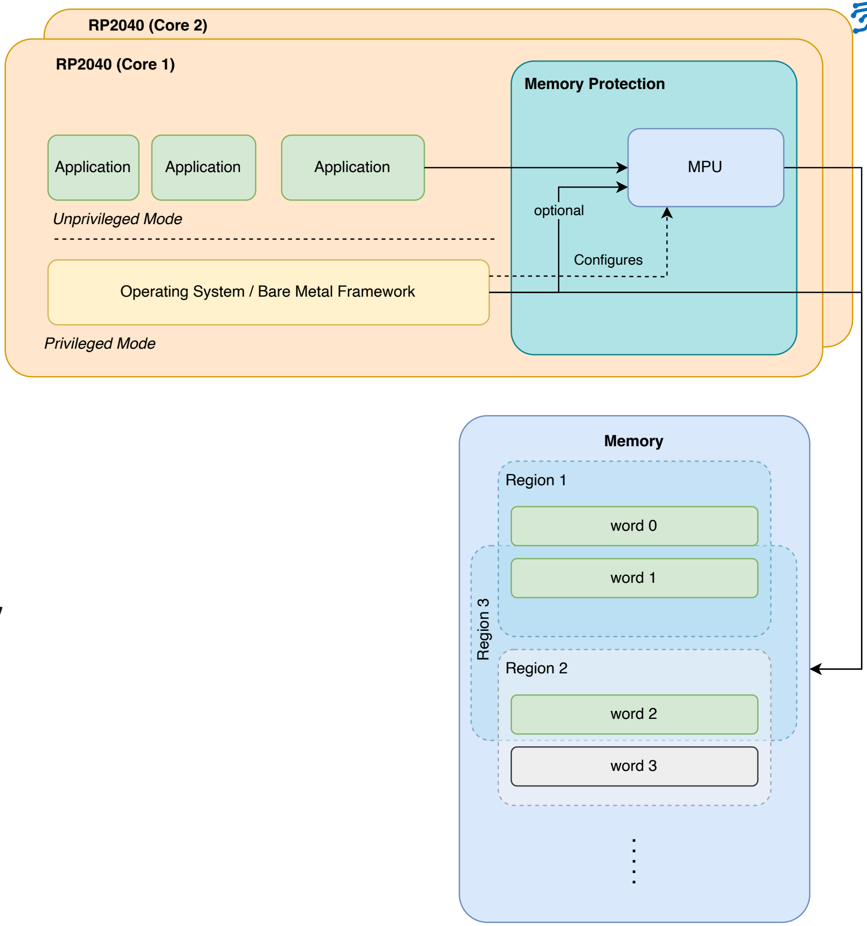
Cortex-M0+

The processor works in three modes:

- **handler** mode - *no restrictions* - used while executing ISRs and Exception Handlers
- **thread** mode
  - **privileged** *no restrictions* - usually used for the operating system
  - **unprivileged** mode - *allows only ALU and memory access through Memory Protection* - used for applications

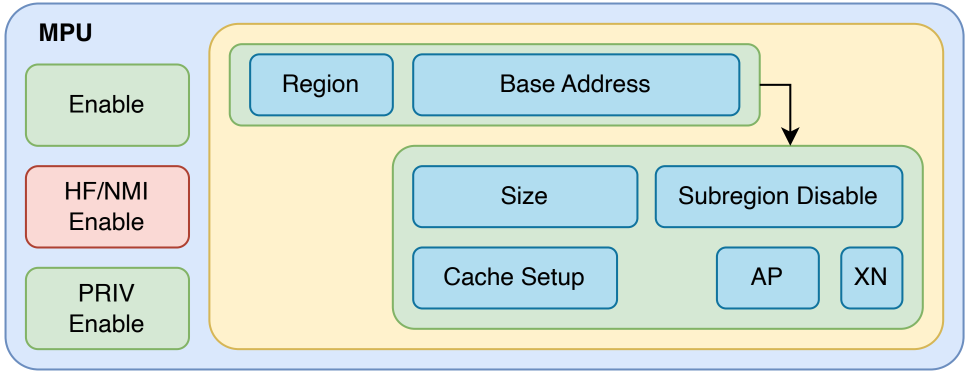
MPU allows 8 regions

- each region has up to 8 subregions
- permissions R W X



Memory Protection Unit

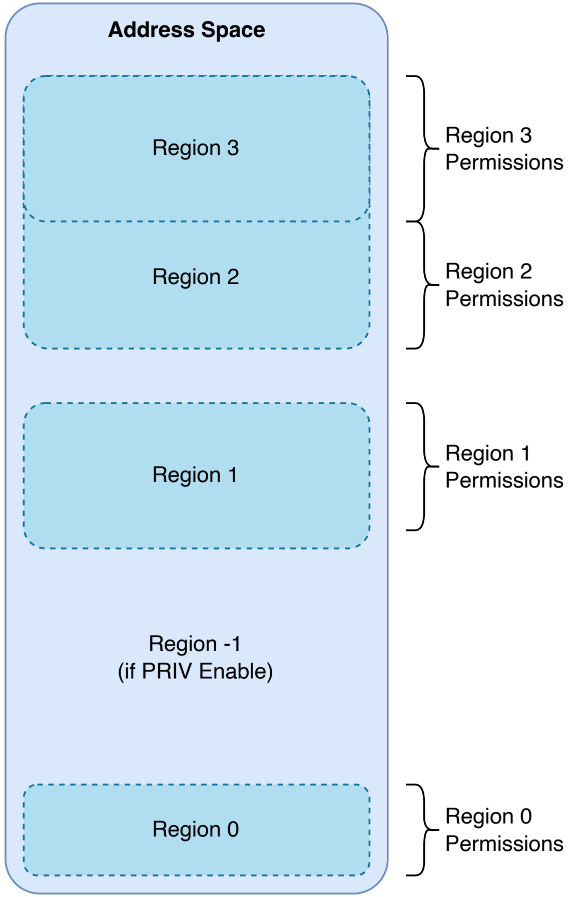
Cortex-M MPU



- allows the definition of *memory regions*
- regions can overlap, *highest region number* takes *priority*
- regions have access permissions (similar to rwx)

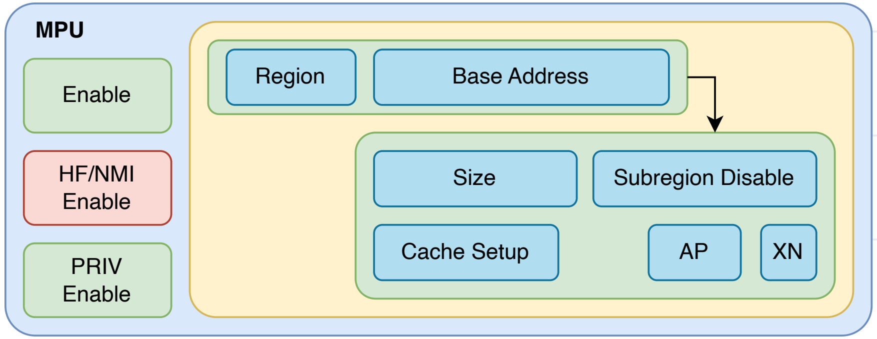
$$region\_size = min(256, 2^{size})$$

$$base\_address = region\_size \times N$$



Memory Protection Unit

Access Protection



AP Access Protection

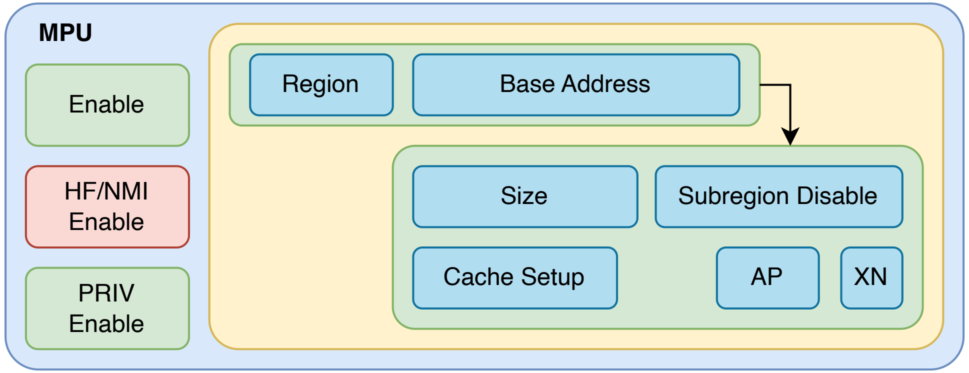
XN eXecute Never

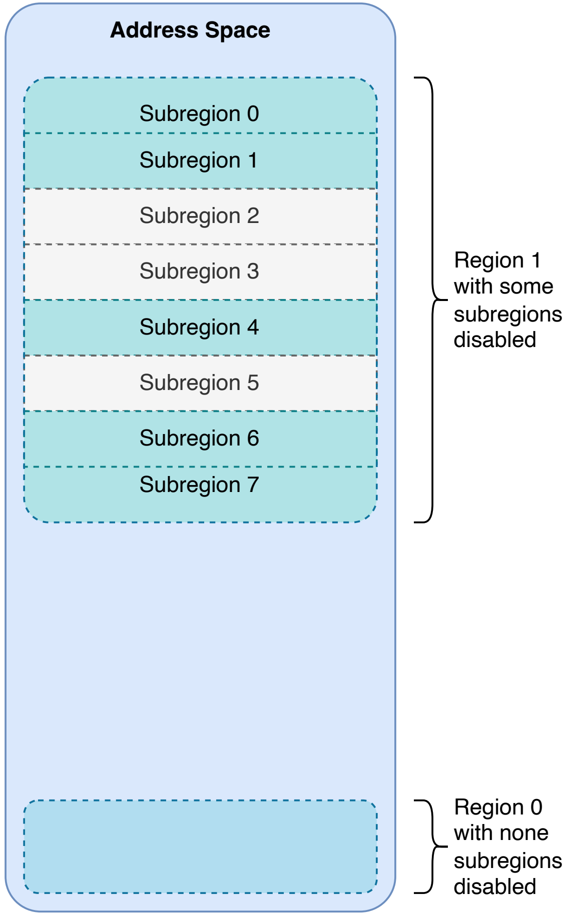
- faults if MCU has to read the next instruction from an *XN* region

AP	Privileged Mode	Unprivileged Mode
`000`	No Access	No Access
`001`	Read/Write	No Access
`010`	Read/Write	Read only
`011`	Read/Write	Read/Write
`100`	Do not use	Do not use
`101`	Read only	No Access
`110`	Read only	Read only
`111`	Read/Write	Read only

Subregions

- each region is divided in 8 subregion
- each bit in `Subregion Disable` disables a subregion
- a disabled subregion triggers a fault if accessed





Subregions' Usage

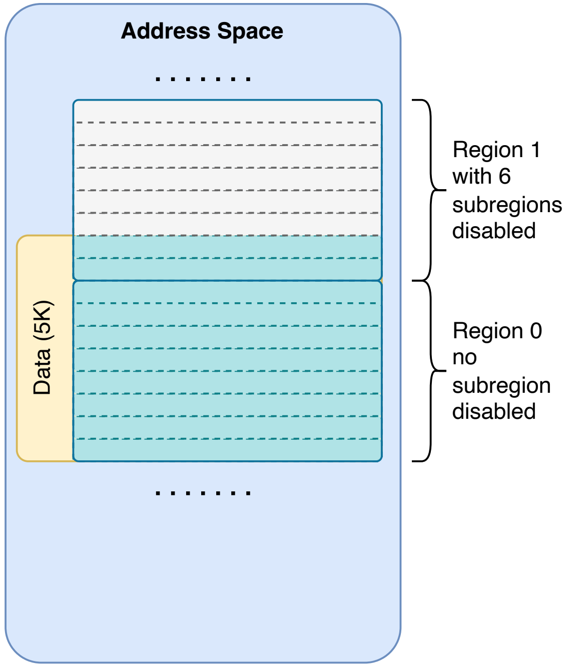
improve granularity

$$region\_size = min(256, 2^{size})$$

$$base\_address = region\_size \times N$$

$$subregion\_size = \frac{region\_size}{8}$$

- a 5K region is not allows (5K is not a power of 2)
- use two 4K regions back to back
- disable 6 of the subregions (subregion is 512B)


Address Space
....
Data (5K)
Region 1 with 6 subregions disabled
Region 0 no subregion disabled
....

Memory Layout

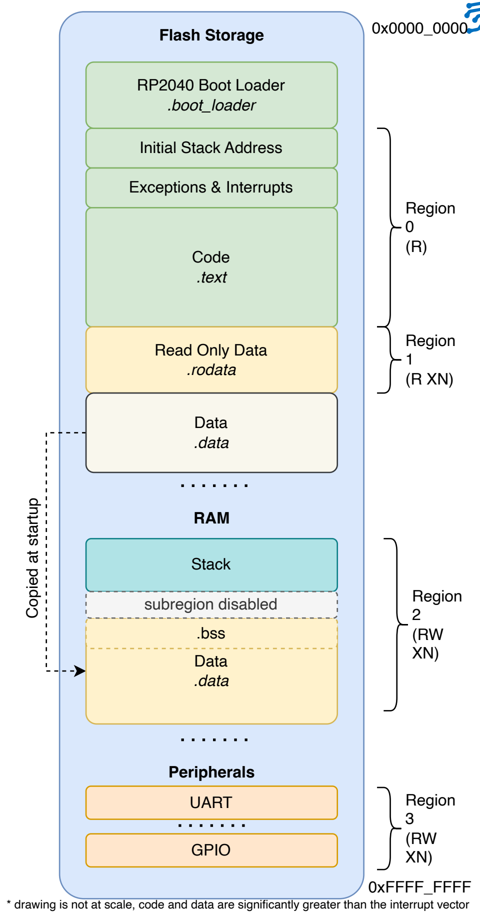
protection

Flash

- **Code** - *read* and *execute*
- **.rodata** - constants - *read only*
- **.data** - *in flash* - initialized global variables
  - is copied to RAM at startup by the `init` function
  - *should not be accessed after startup*

RAM

- **stack** - *read* and *write*
  - *usually protected by unaccessible memory before and after*
- **.data** - *in RAM* - global variables - *read* and write
- **.bss** - global variables (not initialized or initialized to `0`) - *read* and *write*



Conclusion

we discussed about

- Memory Mapped I/O
- Exceptions
- Memory Protection