

# Návrh systémů IoT

## Mikrokontroléry I.

Stanislav Vitek

Katedra radioelektroniky

České vysoké učení technické v Praze

# Vestavné systémy v automobilě

Adaptive cruise control systems detect if cars in front of you are too close and, if necessary, adjust the vehicle's throttle, may apply brakes, and/or sound an alarm.



Advanced airbag systems have crash-severity sensors that determine the appropriate level to inflate the airbag, reducing the chance of airbag injury in low-speed accidents.



Tire pressure monitoring systems send warning signals if tire pressure is insufficient.

Drive-by-wire systems sense pressure on the gas pedal and communicate electronically to the engine how much and how fast to accelerate.



Cars equipped with wireless communications capabilities, called *telematics*, include such features as navigation systems, remote diagnosis and alerts, and Internet access.

# Vestavné systémy v automobilě

Dnešní automobily vyšší střední třídy mívají více než 100 různých procesorů:

- Bezpečnostní pásy; přístrojová deska; ovládání motoru; ABS; automatická kontrola stability; navigační systém; informační a zábavní systém; systém zabraňující kolizím; kontrola tlaku v pneumatikách; hlídání jízdního pruhu; adaptivní tempomat; klimatizace; ovládání airbagů; elektrické ovládání oken a centrální zamykání; parkovací asistent; automatické ovládání stěračů; alarm a imobilizér; elektricky ovládaná sedadla; elektrický posilovač řízení; elektronická převodovka; aktivní odpružení.

# Obecné procesory

- Programovatelné zařízení
  - mikroprocesor
  - mikrokontrolér
- Hlavní části
  - Programová a datová paměť
  - Obecná datová cesta
  - Výbava registry
  - Obecná ALU
- Aplikačně-specifické procesory (ASIC)
  - Optimalizovaná datová cesta
  - Speciální funkční bloky

# Dedikované procesory

- Jednoúčelový digitální obvod
- Hlavní části
  - Komponenty nutné k provádění jednoho programu
  - Nemá programovou paměť
- Výhody
  - Malý
  - Rychlý
  - Nízká spotřeba

# Vestavné systémy

- Dedikovaná funkcionalita
- Provoz v reálném čase
- Malé rozměry a nízká hmotnost
- Nízká spotřeba energie
- Drsné prostředí
- Provoz kritický z hlediska bezpečnosti
- Cenově výhodné

# Architektura Počítače

- Počítačová architektura odkazuje na ty vlastnosti systému, které jsou viditelné programátorovi nebo, jinými slovy, ty vlastnosti, které mají přímý vliv na logické provádění programu.
  - **Architektonické atributy**
    - Sada instrukcí
    - Počet bitů použitých pro reprezentaci různých datových typů (například čísel, znaků)
    - Mechanismy pro vstup/výstup
    - Techniky pro adresaci paměti.
- *Je architektonickým rozhodnutím, zda bude počítač obsahovat instrukci násobení.*

# Organizace počítače

- Počítačová organizace se odkazuje na provozní jednotky a jejich vzájemné propojení, které realizují architektonické specifikace.
- **Organizační atributy transparentní pro programátory**
  - Řídící signály
  - Rozhraní mezi počítačem a periferiemi
  - Použitá paměťová technologie.
- *Je organizačním rozhodnutím, zda bude tato instrukce implementována speciální jednotkou násobení nebo mechanismem, který opakovaně využívá jednotky sčítání systému.*



# Hlavní Strukturální Komponenty

- **Centrální Procesorová Jednotka (CPU)**
  - Kontroluje provoz počítače a provádí zpracování dat.
  - Řídící jednotka, aritmetická a logická jednotka (ALU), registry
  - Propojení CPU (nějaký mechanismus, který umožňuje komunikaci mezi řídicí jednotkou, ALU a registry)
- **Hlavní Paměť** - ukládá data.
- **Vstup/Výstup (I/O)** - přenos data mezi počítačem a jeho vnějším prostředím.
- **Systemové Propojení - sběrnice** - mechanismus, který umožňuje komunikaci mezi CPU, hlavní pamětí a I/O.

# Další Strukturální Komponenty

- **System Přerušeni**
  - Zjednodušuje periferie a zlepšuje využití strojového času, protože CPU nemusí pravidelně kontrolovat jejich stav.
- **Dekodér Adresy**
  - Obvykle je potřeba k generování řídicích signálů pro paměti a periferie s mapovanou pamětí z adres.
- **Generátor Hodin**
  - Musí být připojen k CPU.

# Harvardská architektura

- Navržena Howardem Hathawayem Aikenem v roce 1937.
- První realizace: Harvard Mark I.
- Architektura zahrnuje pět základních bloků.
  - Rozdělení paměti na dvě samostatné části.
  - **Programová paměť** - instrukce; také konstanty a vstupní brány.
    - U počítače Harvard Mark I byla "programová paměť" perforovaná papírová páska, ze které byl program čten a přímo vykonáván.
  - **Datová paměť** - proměnné; volitelně i prostor pro I/O.
    - Datová slova mohou nebo nemusí mít odlišnou šířku než instrukce.

# Výhody a nevýhody Harvardské architektury

- Díky přidělení paměti mohou mít programová a datová paměť samostatné sběrnice, takže je možné číst jak instrukce, tak data současně, což vede k vyšší rychlosti zpracování programu.
- Vyšší bezpečnost, protože data nemohou přepsat část kódu, jako například při útoku přetečení bufferu.

# Von Neumannova architektura (Princetonská)

- Odvozena od návrhu počítačové architektury z roku 1945 s názvem "First Draft of a Report on the EDVAC" od maďarsko-amerického matematika Johna von Neumanna.
- Má pouze jednu paměť (**operační paměť**), sdílenou programem i daty.
- Instrukce i data sdílejí stejný paměťový prostor
  - musí mít stejnou šířku slova
  - řídicí jednotka může instrukce a data odlišit pouze podle jejich pořadí nebo adresy.
  - Data sdílejí stejnou adresu, datovou sběrnici a řídicí sběrnici.
- Operace jsou sekvenční, což ztěžuje práci s paralelismem v návrhu.

# Výhody a nevýhody Von Neumannovy architektury

- Jelikož instrukce a data sdílí společnou sběrnici, instrukce načítání a provádění operace s daty nemohou probíhat současně. Operace mikropočítače s Von Neumannovou architekturou je obvykle pomalejší než systém s architekturou Harvard pracující na stejné frekvenci hodin, známé jako *Von Neumannova uzávěrka*.
- Program se může upravit sám, což je spojeno s vážným rizikem bezpečnosti, jako například *útokem přetečení bufferu*.
- Návrh architektury Von Neumann je jednodušší než návrh Harvardovy a může být levnější.

# Architektura Sady Instrukcí (ISA)

- Je to součást počítačové architektury, která popisuje základní programový model.
- Odděluje programátora (nebo překladač) od skutečné fyzické implementace počítače.
- Stejná ISA může být definována na počítačích s úplně odlišnou mikroarchitekturou. Například pro Intel a AMD můžeme psát identický kód, i když vnitřní struktura obou čipů je zcela odlišná.
- ISA definuje:
  - Které operace jsou podporovány procesorem.
  - Datové typy, registry, režimy adresace, organizaci paměti.
  - Přerušování a výjimky, operace vstupu a výstupu.

# RISC vs. CISC

- Sada instrukcí RISC obsahuje relativně malý počet jednoduchých instrukcí (33-150).
- Velikost instrukcí RISC je obvykle jedno slovo.
- Provedení instrukcí RISC je ve stejném počtu strojových cyklů, obvykle přímo úměrné počtu slov instrukce.
  - *Čas provádění různých instrukcí na procesorech CISC se liší významně, i když mají stejnou délku, v závislosti na složitosti mikrokódu.*
- Programy na procesorech RISC jsou obvykle rychlejší, ale kvůli více instrukcím potřebují více úložného prostoru.
- Čas provádění programu na procesorech RISC může být přesně určen.
  - *Každá instrukce na procesoru CISC má odlišný čas provádění.*

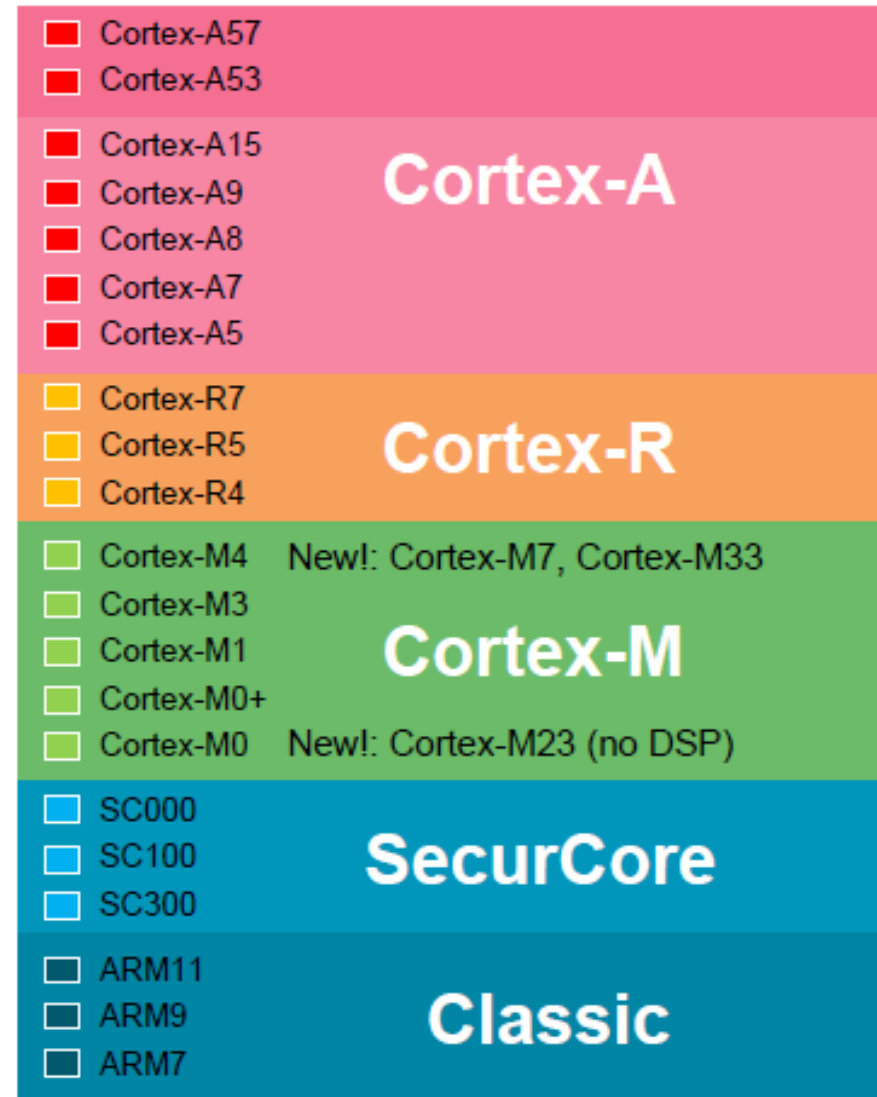


# Architektura ARM

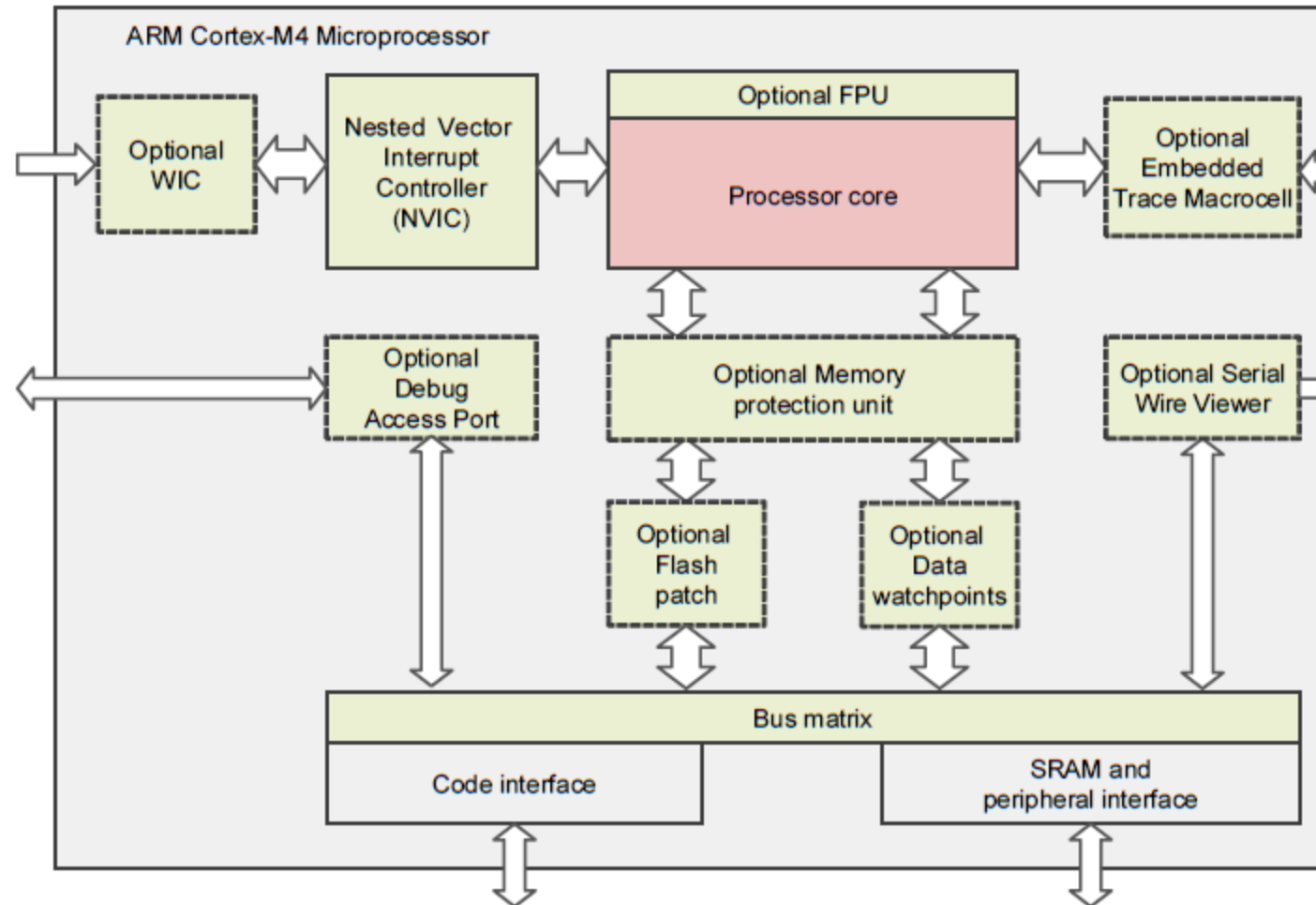
- Architektura ARM představuje
  - soubor specifikací,
  - instrukční sady,
  - modelu provedení,
  - organizace paměti,
  - uspořádání a časování instrukcí, atd.
- Procesor ARM je vyvíjen s využitím jedné z architektur ARM.

# Cortex jádra

- Rodina ARM Cortex-A:
  - Procesory pro aplikace, podpora OS a výkonných aplikací
  - Smartphony a smart TV.
- Rodina ARM Cortex-R:
  - RT aplikace s vysokým výkonem a spolehlivostí
- Rodina ARM Cortex-M:
  - Mikrokontroléry, citlivé na náklady, podpora SoC.



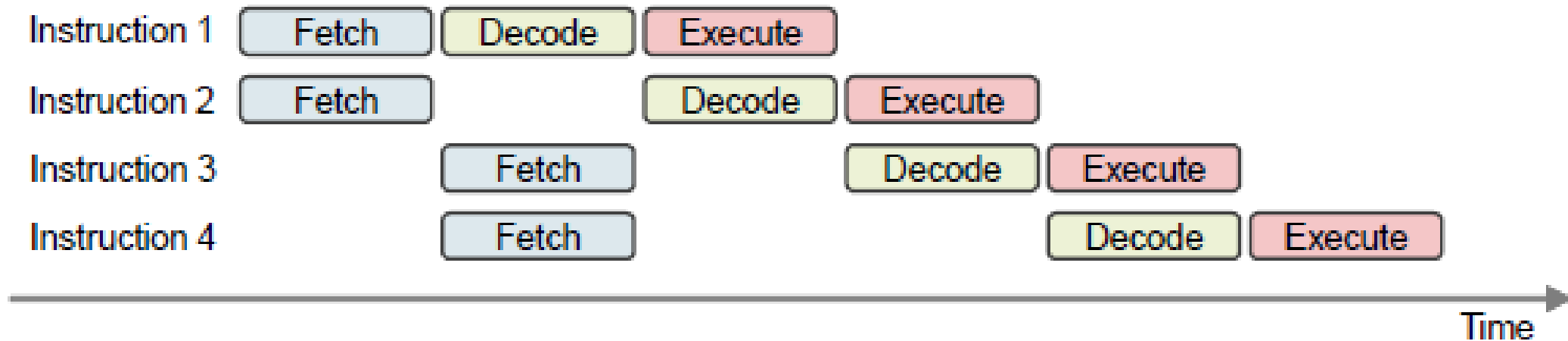
# Blokový diagram Cortex-M4



# Blokový diagram Cortex-M4

- **Jádro procesoru:**
  - Interní registry, ALU, datová cesta a řídicí logika.
  - Šestnáct 32bitových registrů pro obecné a speciální využití.
- **Třístupňová pipeline procesoru:**
  - Načítání, dekódování a provedení.
  - Některé instrukce mohou vyžadovat více cyklů na provedení, v takovém případě je pipeline zablokována.
  - Pipeline je vyprázdněna při vykonání instrukce větvení.
  - Lze načíst až dvě instrukce během jednoho přenosu (16bitové instrukce).

# Pipeline



# Blokový diagram Cortex-M4

- **Vektorový řadič přerušení (NVIC):**
  - Až 240 signálů žádosti o přerušení a nemaskovatelné přerušení (NMI).
  - Automaticky zpracovává vnořená přerušení a porovnává priority.
- **Řadič probuzení přerušení (WIC):**
  - Pro aplikace s nízkou spotřebou energie může mikrokontrolér přejít do režimu spánku.
  - Když je detekována žádost o přerušení, WIC může informovat jednotku správy napájení, aby zapnula systém.
- **Jednotka pro ochranu paměti (volitelná):**
  - Používá se k ochraně obsahu paměti, např. nastavuje některé paměťové oblasti jako pouze pro čtení nebo brání uživatelským aplikacím v přístupu k privilegovaným datům aplikace.

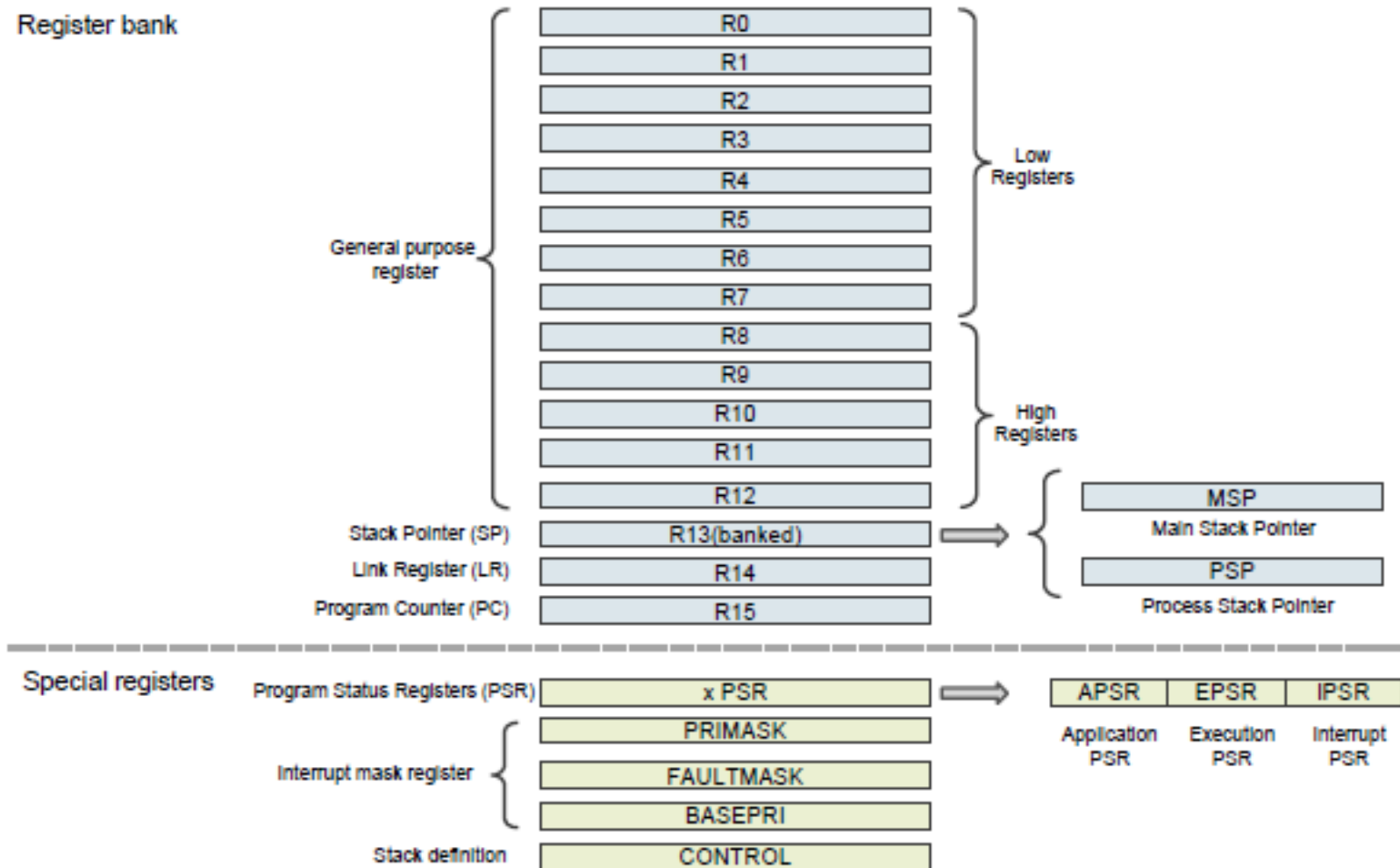
# Blokový diagram Cortex-M4

- **Propojovací sběrnice**
  - Umožňuje přenos dat na různých sběrnících současně.
  - Poskytuje správu přenosu dat, např. vyrovnávací paměť zápisu, operace orientované na bity (bit-banding).
  - Může obsahovat mosty sběrnic (např. most sběrnice AHB-to-APB) pro propojení různých sběrnic do sítě s použitím jediné globální paměťové oblasti.
  - Zahrnuje interní sběrnici, datovou cestu v jádře procesoru a jednotku rozhraní AHB LITE.
- **Podsystem pro ladění:**
  - Řídí ladící kontrolu, programové body přerušení a sledování dat.
  - Během ladění může zastavit jádro procesoru a umožnit vývojáři analyzovat stav procesoru v daném okamžiku.

# Registry Cortex-M4

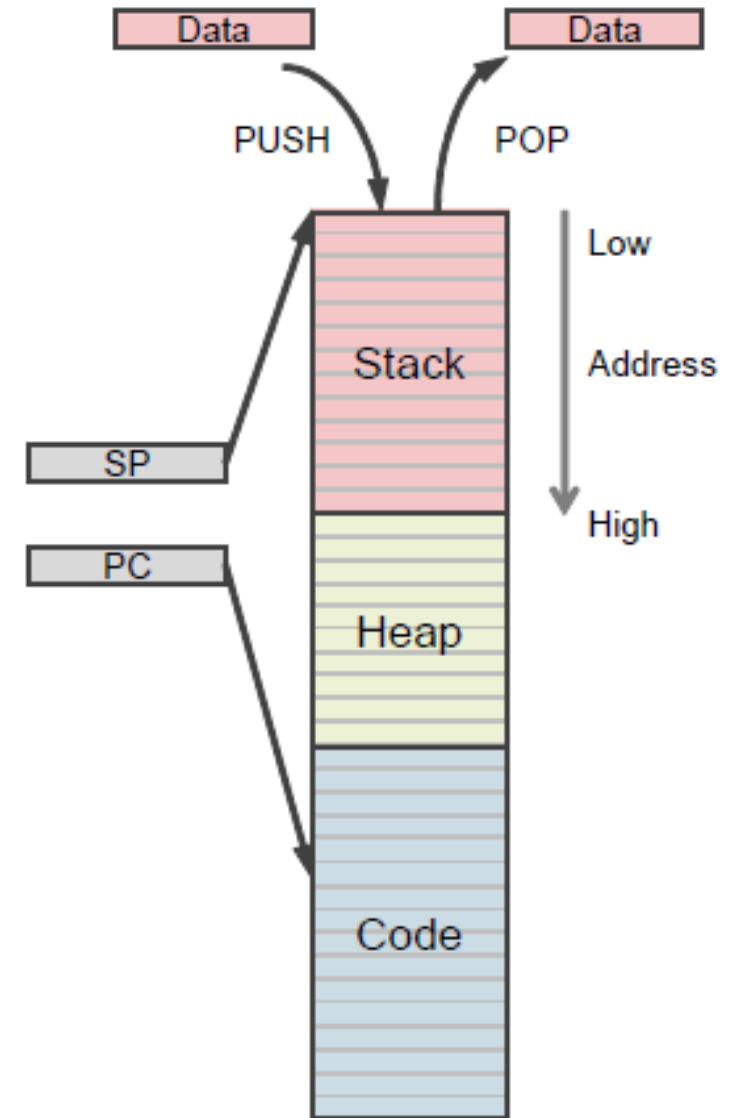
- **Registry procesoru:**
  - Ukládání a zpracování dočasných dat uvnitř jádra procesoru.
  - Architektura load-store.
- **Registry Cortex-M4:**
  - Šestnáct 32b registrů (třináct pro obecné účely).
  - Speciální registry.





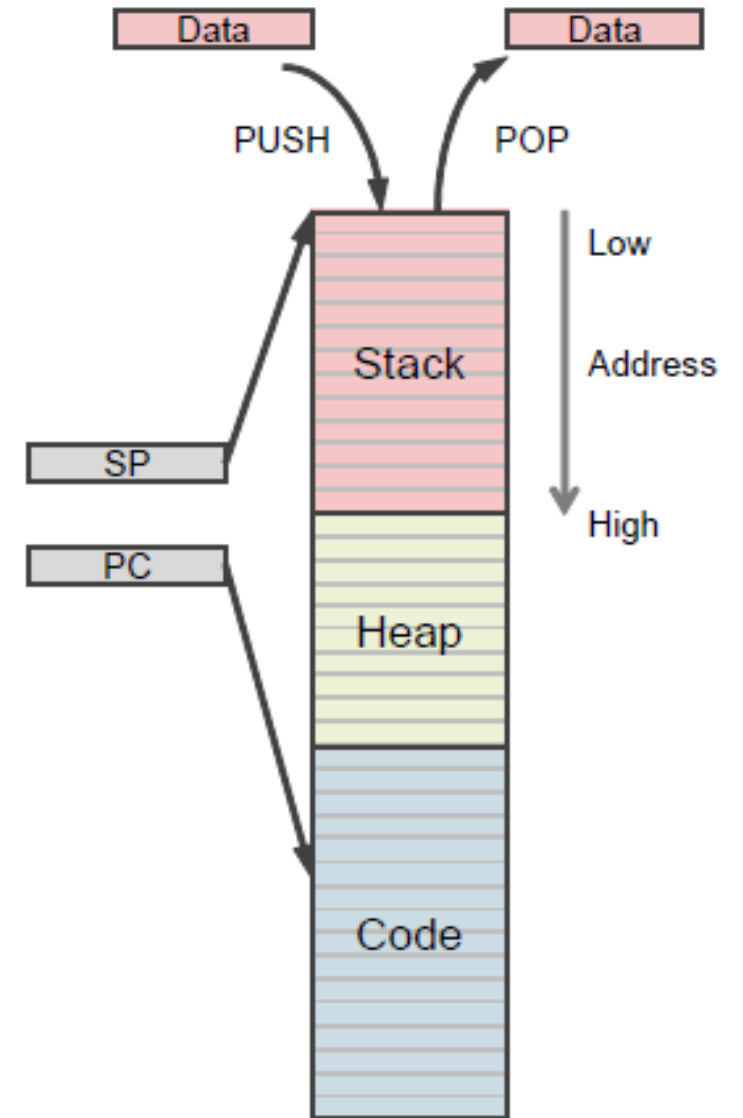
# Registry Cortex-M4

- R0 – R12:
  - Registry pro obecné účely.
  - R0 – R7 mohou být přístupovány k jakékoli instrukci.
  - R8 – R12 nemusí být přístupné některými instrukcemi Thumb (16bitovými).



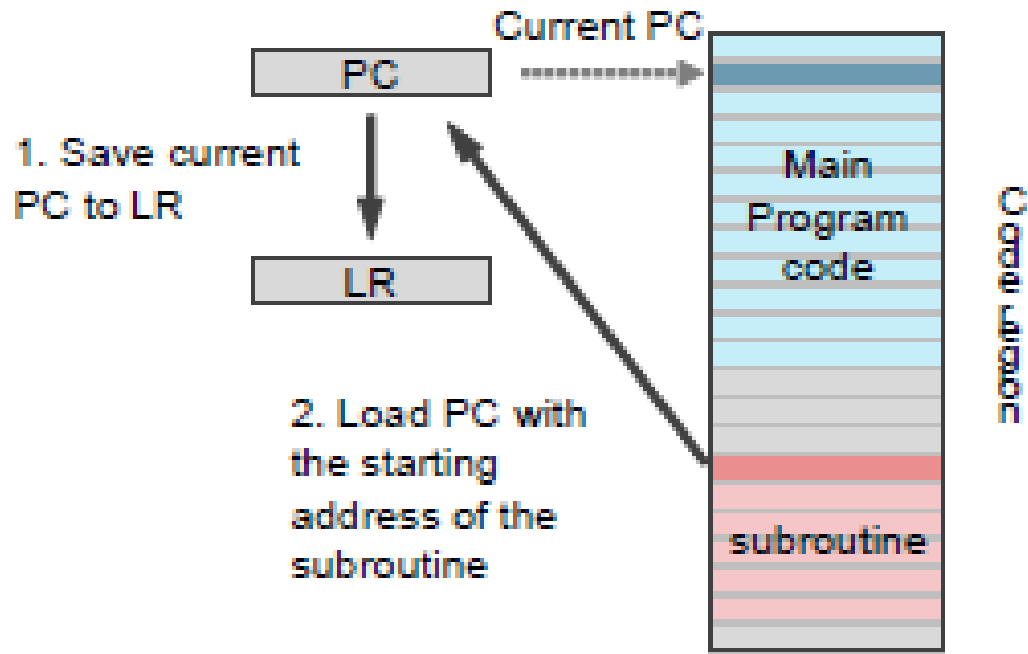
# Registry Cortex-M4

- R13: Ukazatel zásobníku (SP):
  - Zaznamenává aktuální adresu zásobníku.
  - Používá se pro ukládání kontextu programu při přepínání mezi úkoly.
  - Cortex-M4 má dvě SP: hlavní SP, používaný v aplikacích vyžadujících privilegovaný přístup, např. jádro OS, a procesní SP, používaný v kódu aplikace na základní úrovni (když není spuštěn manipulátor výjimek).

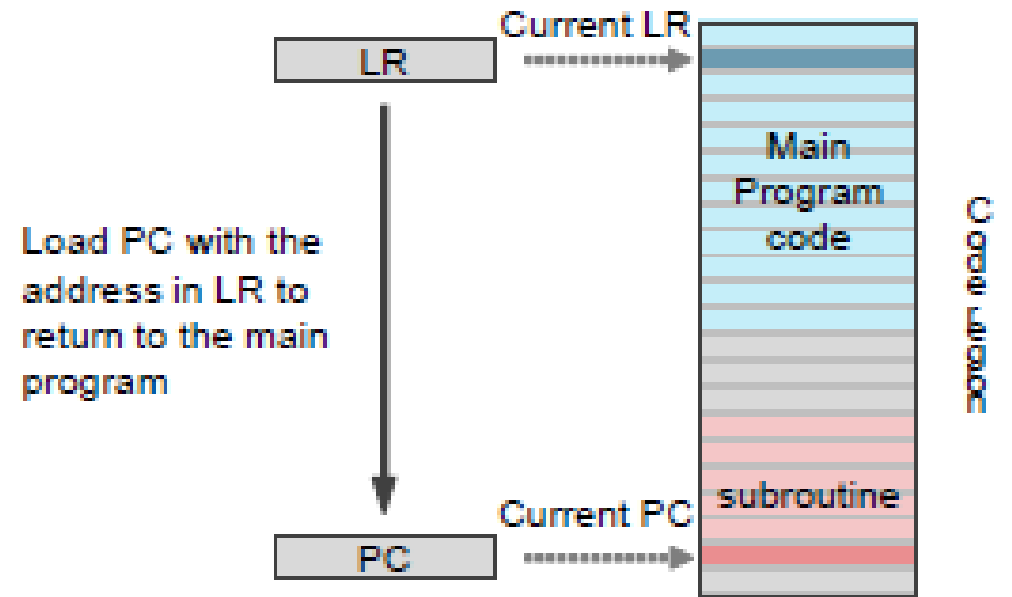


# Registry Cortex-M4

- **Počítač programu (PC)**
  - Zaznamenává adresu aktuálního instrukčního kódu
  - Automaticky se zvyšuje o 4 při každé operaci (pro 32-bitový instrukční kód), s výjimkou větvení
  - Při větvení, například při volání funkcí, se PC změní na konkrétní adresu, zatímco aktuální PC uloží do Link Register (LR)
- **R14: Link Register (LR)**
  - LR slouží k uložení návratové adresy podrutiny nebo volání funkce
  - Po dokončení funkce načte programový čítač (PC) hodnotu z LR



Call a subroutine



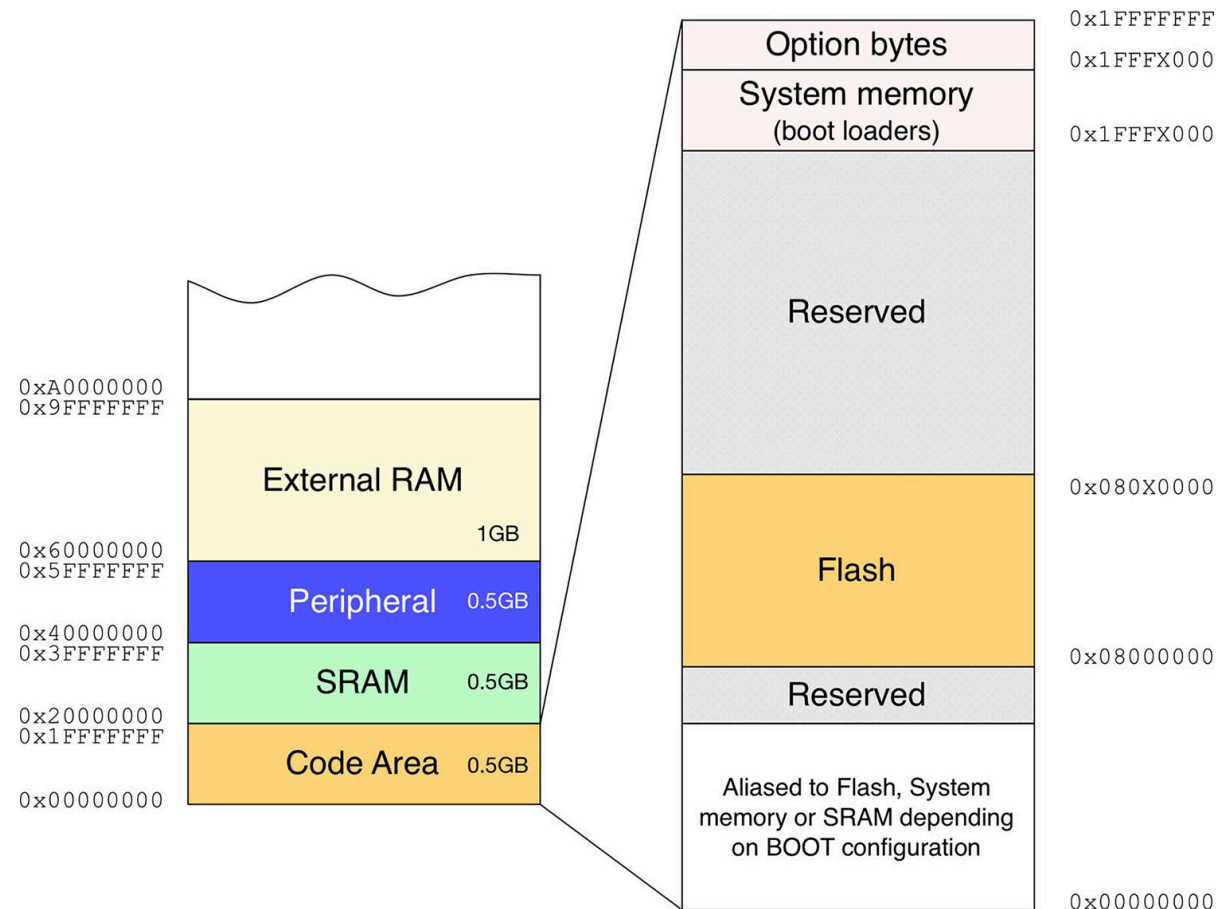
Return from a subroutine to the main program

# Registry Cortex-M4

- **xPSR, kombinovaný registr stavu programu**
  - Poskytuje informace o běhu programu a vlajkách ALU
  - Aplikační PSR (APSR)
  - Přerušovací PSR (IPSR)
    - Číslo ISR - aktuálně vykonávané číslo rutiny obsluhy přerušení
  - Exekuční PSR (EPSR)
- **Aplikační PSR (APSR)**
  - N: negative flag - nastavena na jedničku, pokud je výsledek z ALU záporný
  - Z: zero flag - nastavena na jedničku, pokud je výsledek z ALU nula
  - C: carry flag - nastavena na jedničku, pokud dojde k přetečení
  - V: overflow flag - nastavena na jedničku, pokud dojde k přetečení

# Organizace paměti

- Programová paměť, datová paměť, registry a I/O porty jsou organizovány v rámci stejné lineární adresové prostoru o velikosti 4 GB
- Bajty jsou zakódovány v malém koncovém formátu v paměti
- Adresovatelný paměťový prostor je rozdělen do 8 hlavních bloků, každý o velikosti 512 MB
- Adresy jsou ve formátu 4B

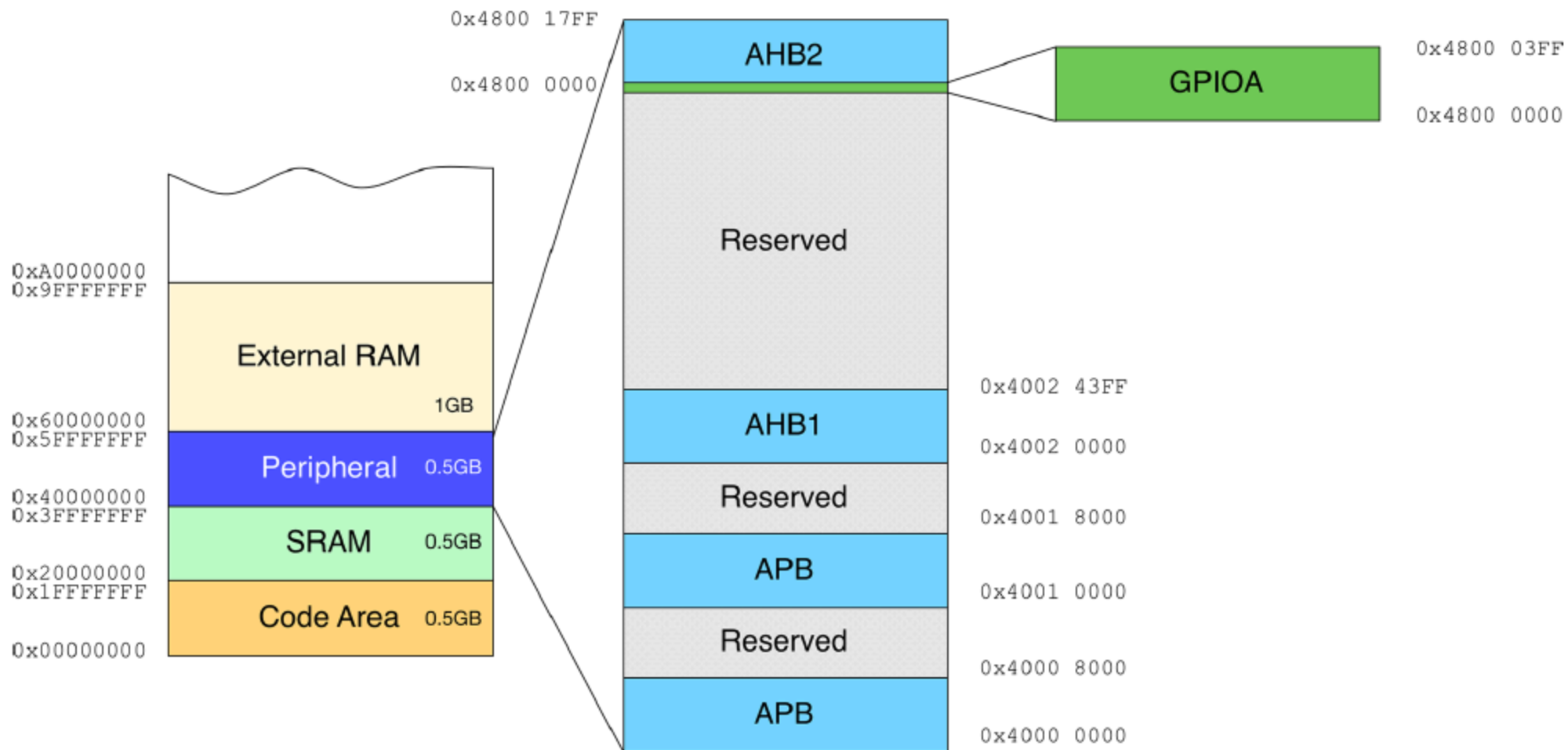


# Organizace paměti

- Všechny procesory Cortex-M mapují oblast kódu od adresy `0x0000 0000`
- Oblast kódu zahrnuje tabulku resetovacích vektorů
  - Ukazatel na začátek zásobníku (obvykle v SRAM)
  - Reset Handler - adresa hlavní funkce programu
- SRAM začíná na adrese `0x2000 0000`
  - Pokus o přístup na místo mimo tuto oblast způsobí výjimku Bus Fault
- Dalších 0,5 GB paměti je věnováno mapování periférií.
  - Každé periferní zařízení poskytované MCU (časovače, rozhraní I2C a SPI, USART atd.) má v této oblasti svůj alias
  - Je na konkrétním MCU, aby tuto paměťovou oblast organizoval.



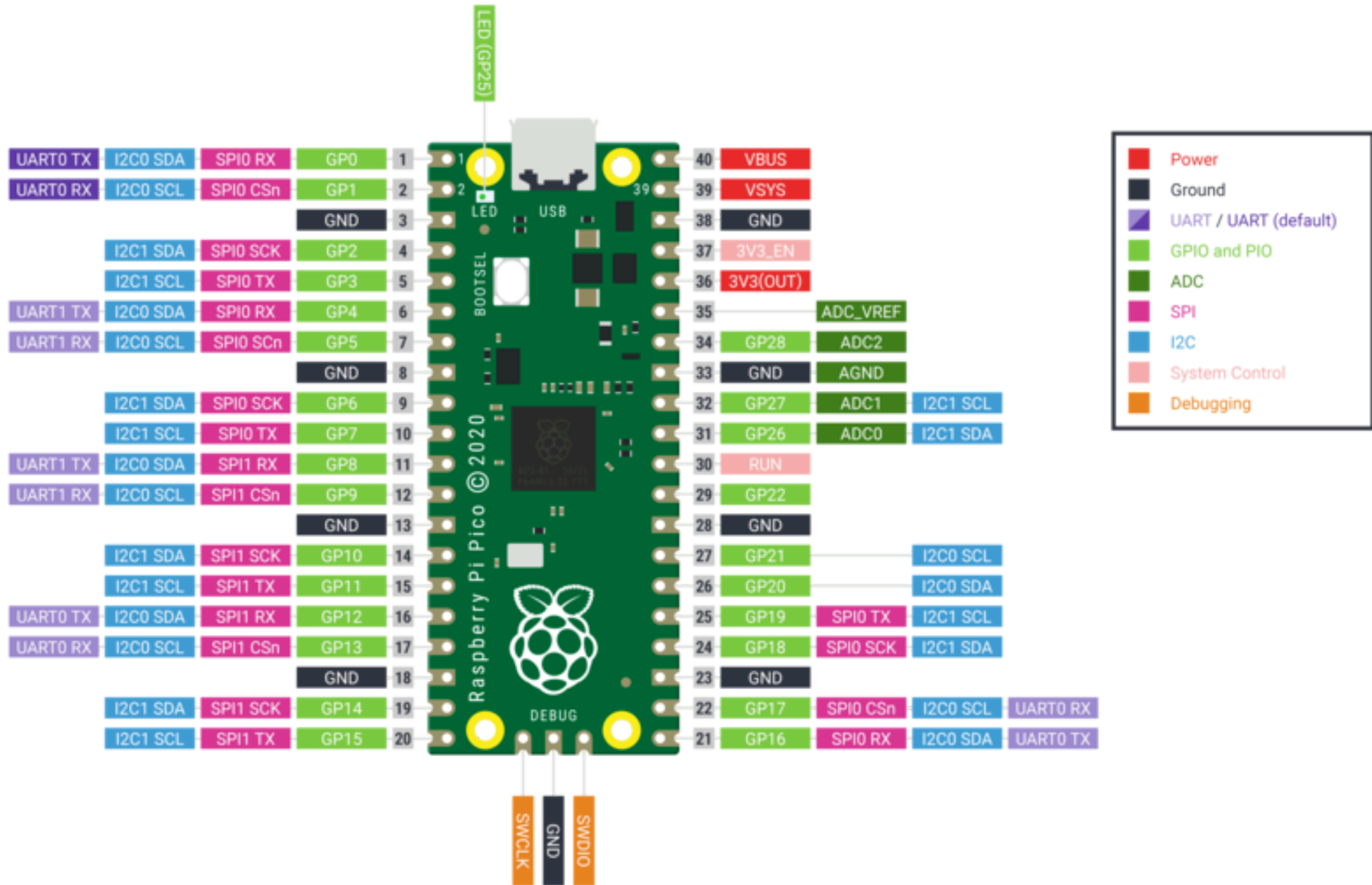
# Mapa paměťových oblastí periférií



Ukázka kódu pro STM32 (live)

# Raspberry Pi Pico

- Všechny předchozí desky Raspberry Pi, jako například Raspberry Pi 3 Model B+, Raspberry Pi 4 Model B nebo menší Raspberry Pi Zero, byly vybaveny procesory Broadcom (BCM2835, BCM2836, BCM2711 atd.).
- Raspberry Pi Pico je vybaven RP2040, což je mikrokontrolér navržený Raspberry Pi, první vlastní procesor od Raspberry Pi Foundation.
- RP2040 je založen na dvou jádrech ARM Cortex-M0+ s taktovací frekvencí až 133 MHz a je vyráběn 40 nm technologií.
- MCU RP2040 má také port MicroPython a zavaděč UF2 v paměti ROM pro snadné nahrání programu.

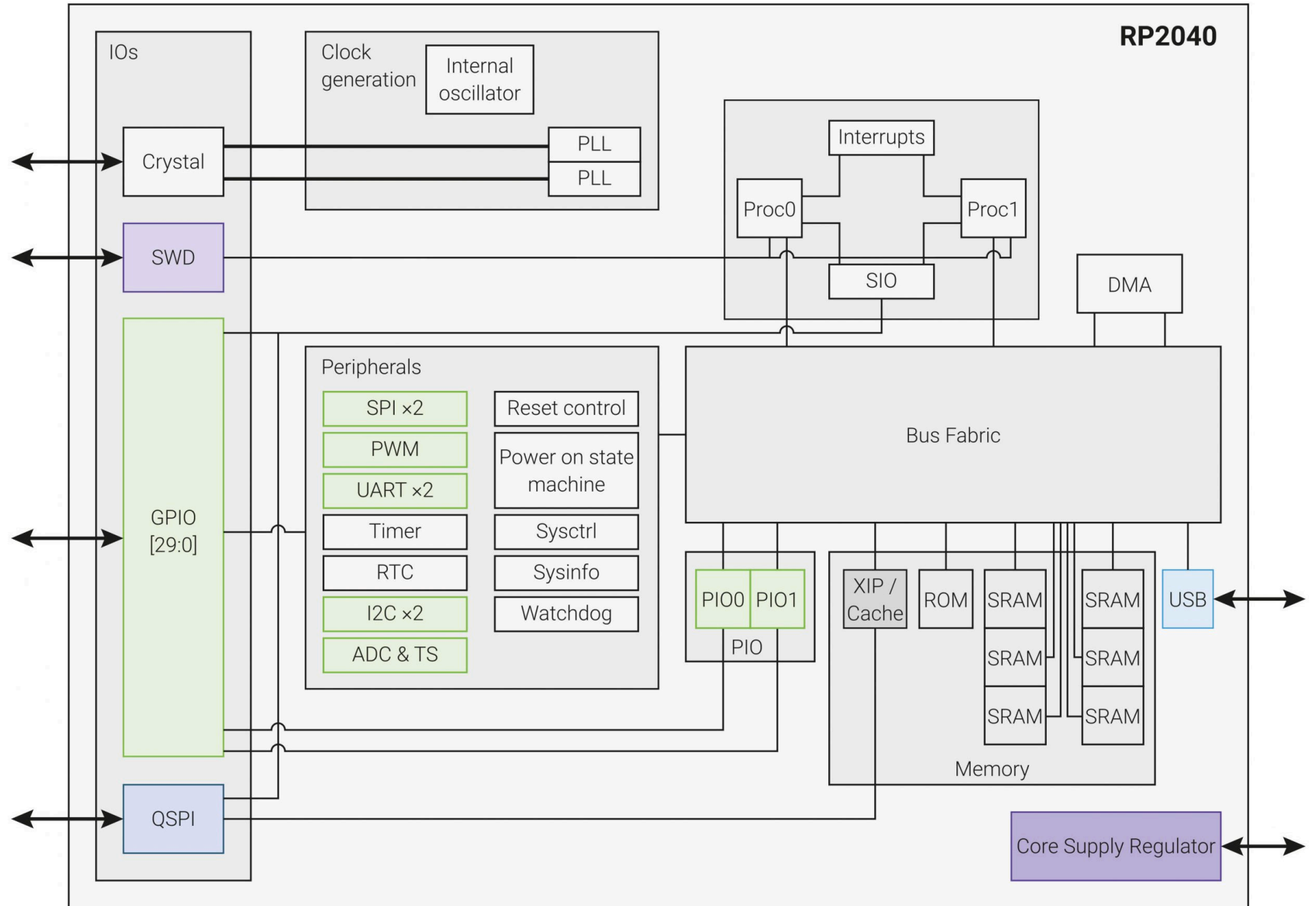


# Raspberry Pi Pico

- Mikrokontrolér RP2040
- 2 MB paměti SPI Flash
- Port Micro-USB typu B pro napájení a programování
- 40 vstupních a výstupních pinů typu DIP s okrajovým zalisováním
- 3pinové rozhraní ARM pro sériové ladění (SWD)
- 12 MHz krystalový oscilátor
- Tlačítko pro výběr bootování
- Jedna uživatelská LED dioda (připojená ke GPIO 25, na modelu **W** k wi-fi řadiči)
- 3,3V Buck-Boost SMPS převodník

# Mikrokontrolér RP2040

- Dvě jádra ARM Cortex-M0+
- Taktovací frekvence až 133 MHz
- 264 KB vestavěné paměti SRAM
- 30 pinů GPIO
- Až 16 MB paměti Flash mimo čip
- Čtyřkanálový ADC s 12bitovým rozlišením
- Programovatelné IO (PIO)
- Další periferie - 2x UART, 2x SPI řadič, 2x I2C řadič, 16 PWM kanálů, USB 1.1 řadič

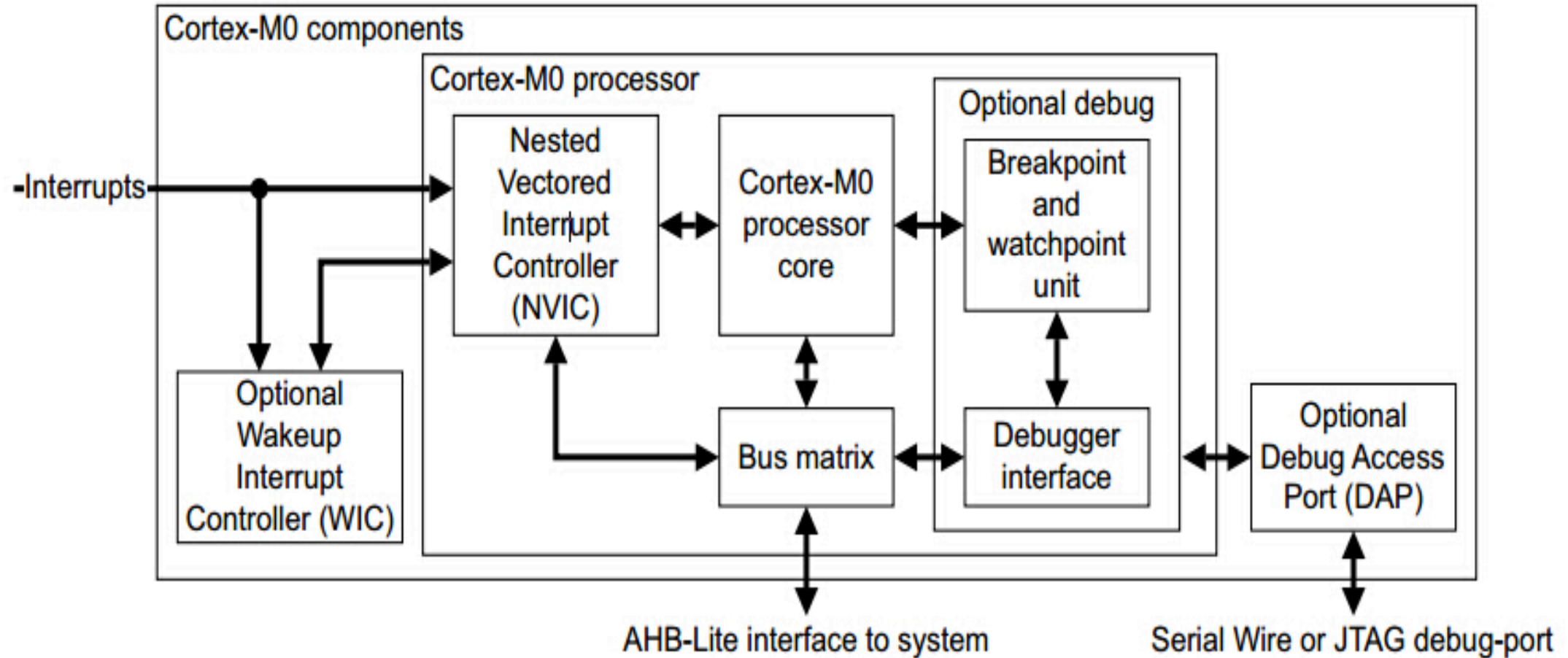


# ARM Cortex-M0+

- Komunikační rozhraní jádra
  - Externní AHB-Lite interface -> busfabric
  - Debug Access Port (DAP)
  - Single-cycle I/O Port -> [SIO periferie](#)
- Konfigurace jádra
  - 32bitový, Little Endian, 8 MPU (Memory Protection Unit)
  - Podpora ladění (2 drátové rozhraní SWD)
  - 26 ext. přerušení, 34 WIC (Wake-up Interrupt Controller)
  - Všechny registry resetovány po restartu



# ARM Cortex-M0+ architektura



# Zdroje hodinového kmitočtu

## clk\_ref

- Interní Ring Oscilator (ROSC), může být přepnutý na externí krystalový (XOSC)
- 6-12MHz

## clk\_sys

- Během reset napájen z clk\_ref, pak většinou přepnutý na PLL
- 125MHz

## clk\_peri

- Typicky napájen z clk\_sys, ale umožňuje periferiím být nezávislé (clk\_sys je možné softwarově změnit)
- 12-125MHz

# Zdroje hodinového kmitočtu

## clk\_usb, clk\_adc

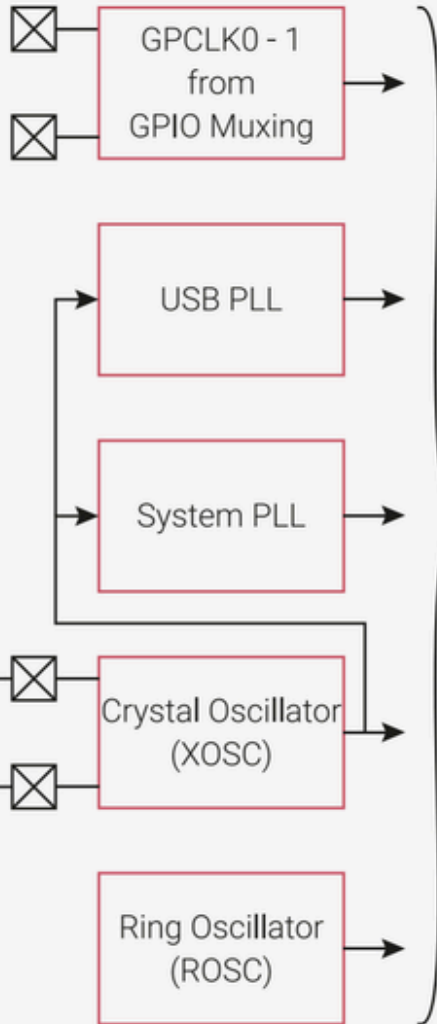
- Referenční hodiny pro USB a ADC
- 48MHz

## clk\_rtc

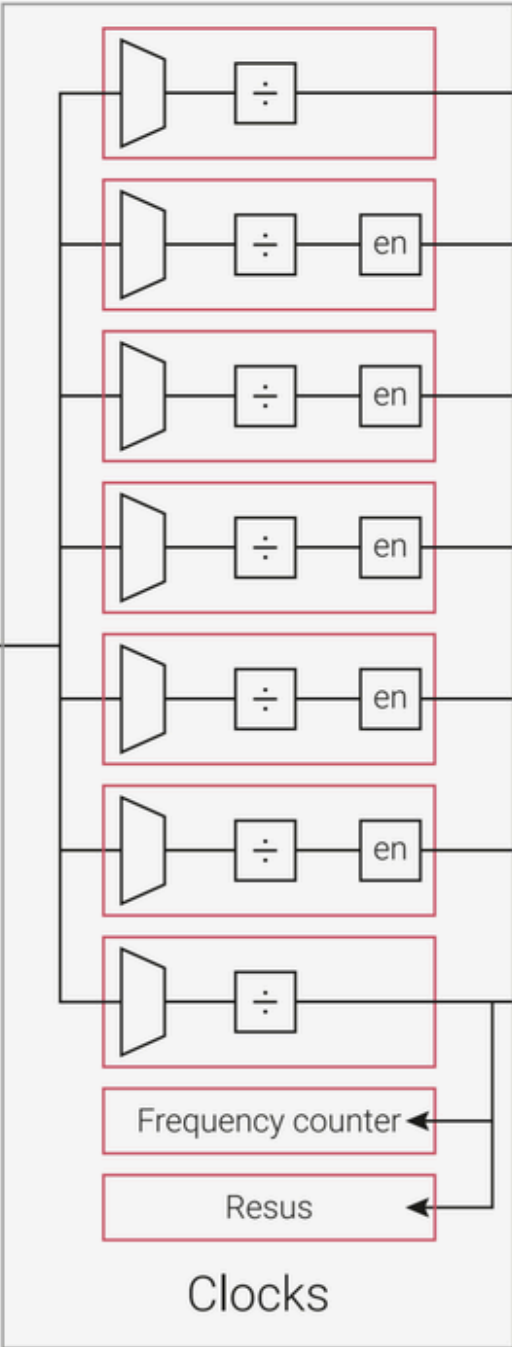
- RTC dělí kmitočty pro získání 1s reference
- 46875Hz

Další podrobnosti v [dokumentaci](#).

External clocks  
or  
Relaxation  
oscillators



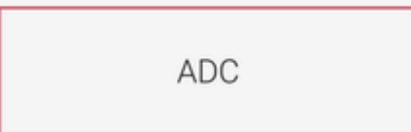
Clock  
sources



clk\_gpout0-3



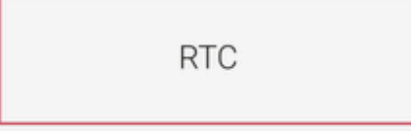
clk\_adc



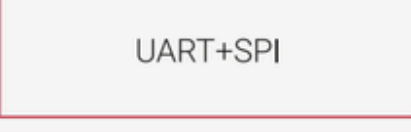
clk\_usb



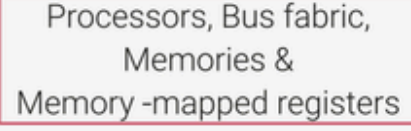
clk\_rtc



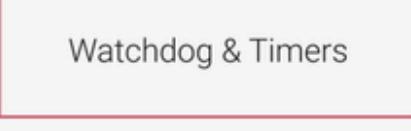
clk\_peri



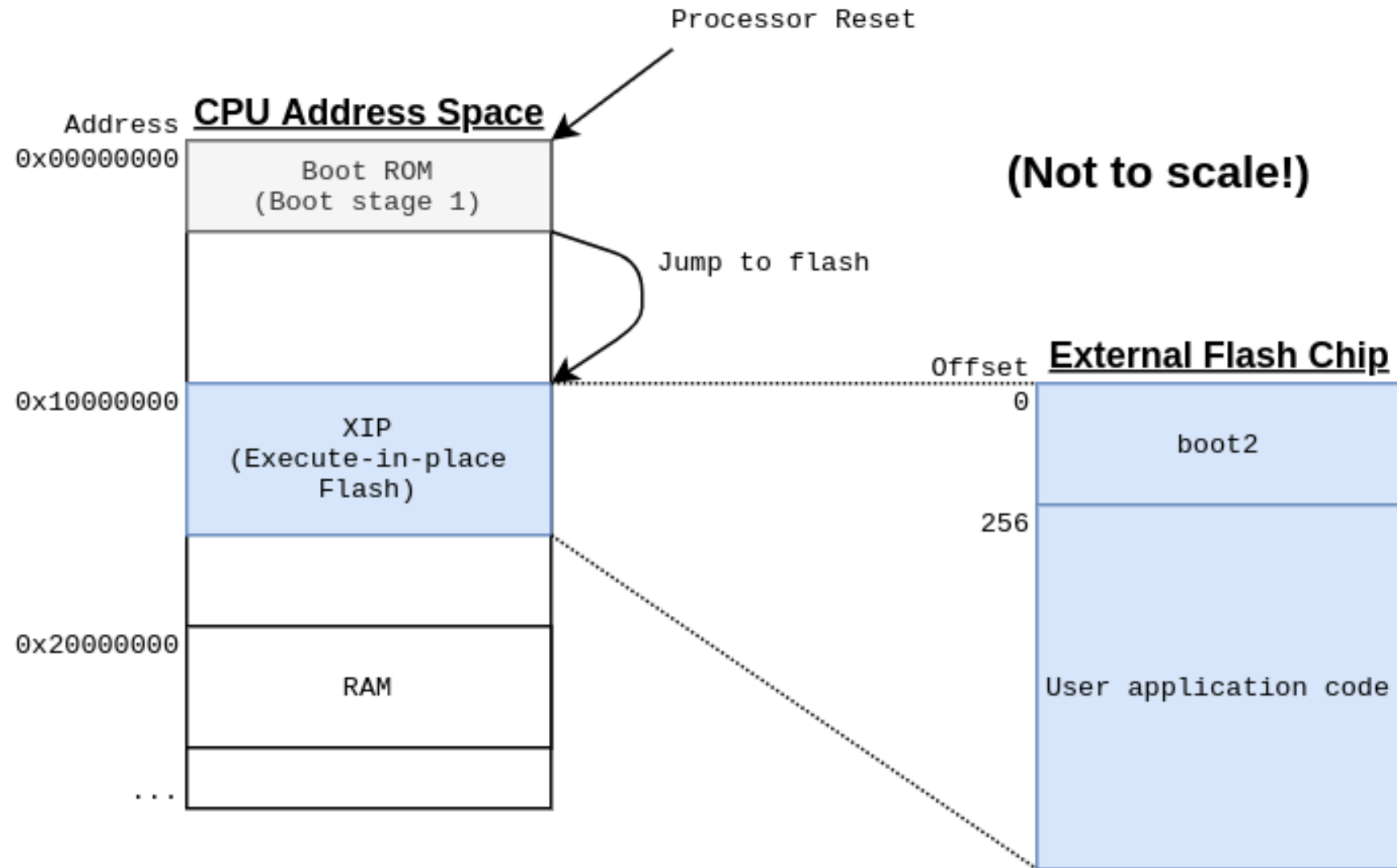
clk\_sys



clk\_ref

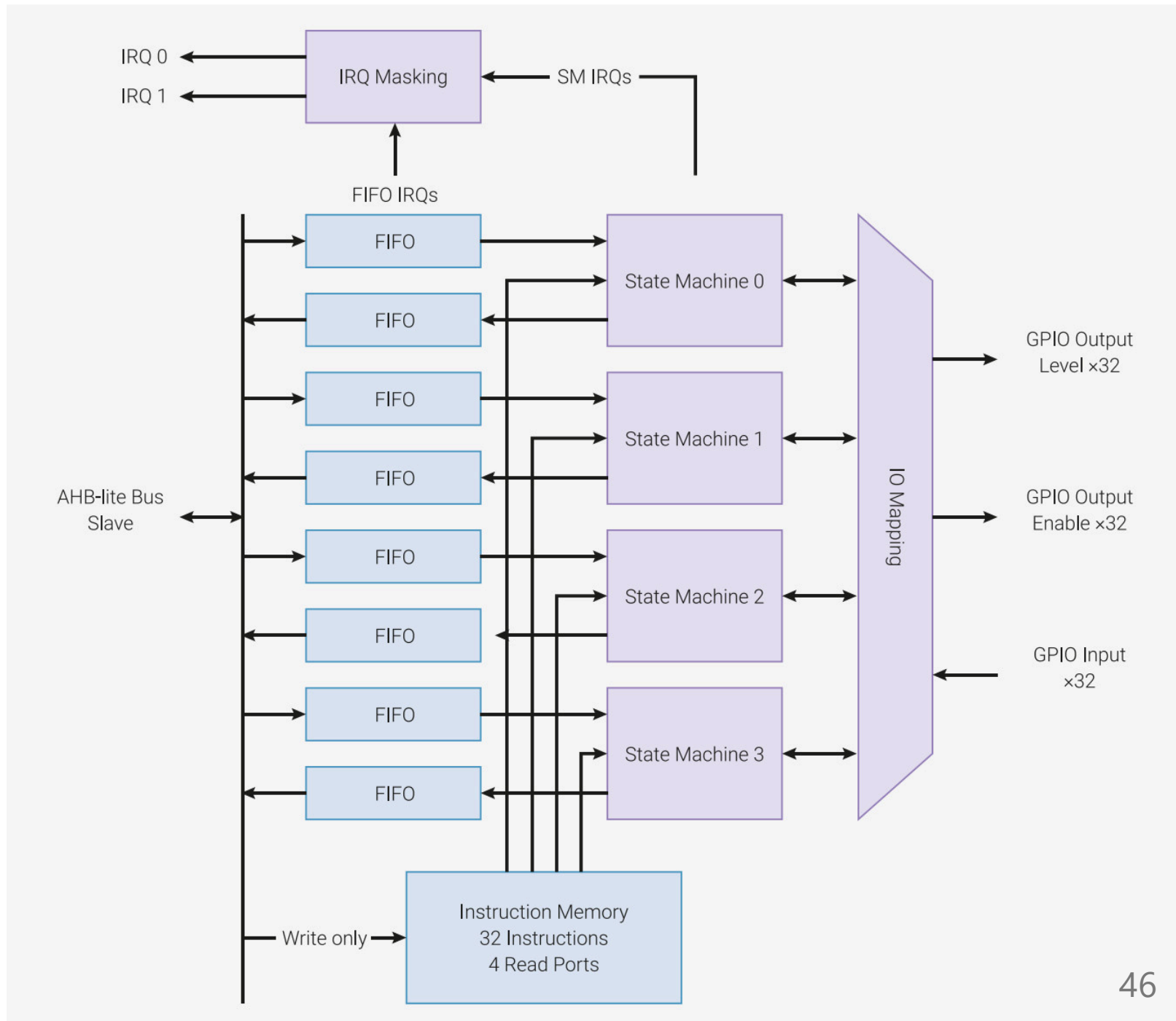


# Paměť



# PIO

- Implementuje řadu protokolů na HW úrovni
- Může pracovat nezávisle na CPU
- Každý RP2040 má dvě PIO instance

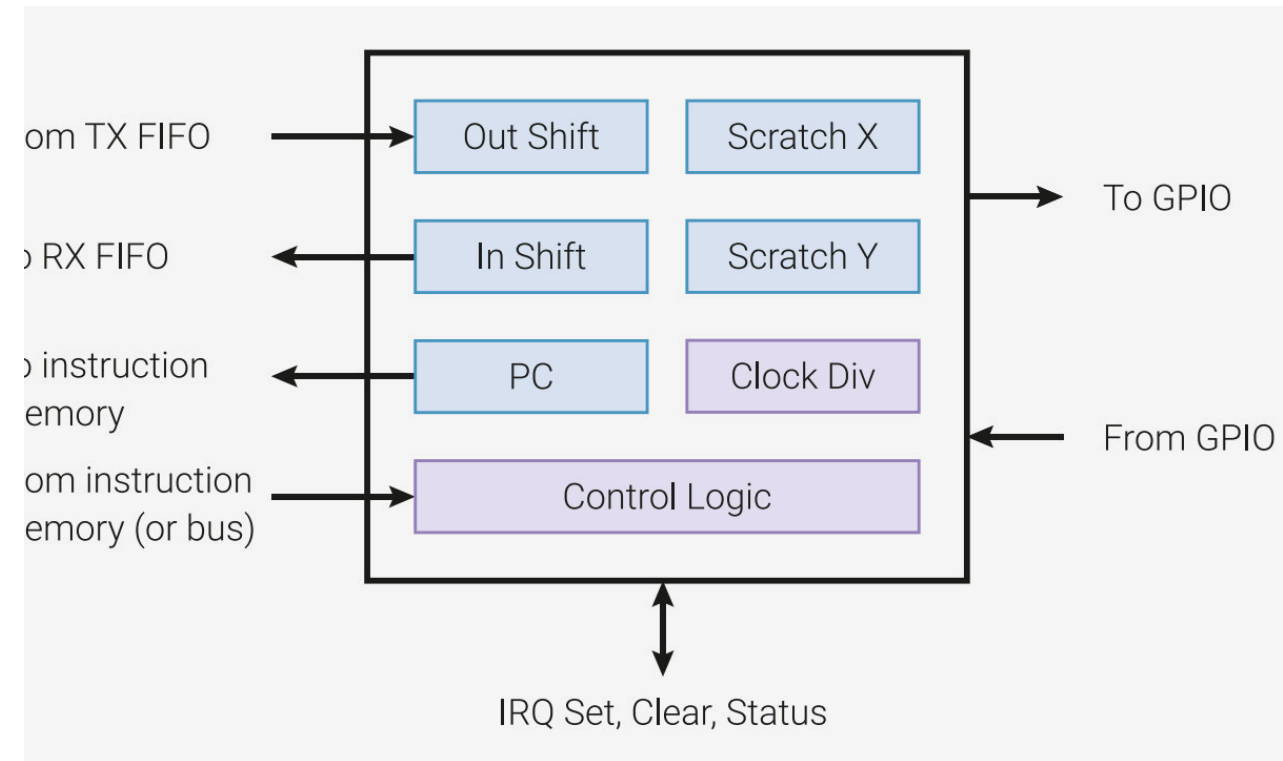


# PIO State machine

- Každý PIO má 4 **stavové stroje**. Tyto stavové stroje fungují jako malé, velmi omezené procesory schopné spouštět instrukce, které se nacházejí ve sdílené paměti instrukcí.
- Paměť instrukcí může pojmout celkem až **32 instrukcí**. Každý stavový stroj však může čerpat instrukce odkudkoli z této paměti.
- Můžete například vytvořit 4 samostatné programy, každý s 8 instrukcemi, které se spustí samostatně na stavových strojích. Případně můžete mít jeden program s 16 instrukcemi, který poběží na všech 4 stavových strojích (a zbylých 16 míst pro instrukce nebude dělat nic).

# PIO State machine

- Každý stavový stroj má přístup ke 2 FIFO, které můžete použít k odesílání a přijímání dat do/z hlavního procesoru
- Ve výchozím nastavení je jedna FIFO určena pro odchozí data a druhá pro data příchozí.
- Lze je však nastavit tak, aby se obě používaly pro odchozí nebo obě pro příchozí data, pokud by bylo třeba zdvojnásobit velikost vyrovnávací paměti.





# PIO State machine

- Kromě toho má každý stavový stroj přístup ke sdílené bance 8 příznaků přerušení. Ty mají různé využití, například pro synchronizaci stavových strojů nebo upozornění procesoru, že jsou některá data připravena ke spotřebování.
- A konečně, každý stavový stroj může ovládat kterýkoli z 32 pinů GPIO RP2040. Aby však stavový stroj mohl ovládat skupinu pinů, musí být tyto piny seskupeny do souvislé sady.
- Každý stavový stroj (nezapomeňte: v RP2040 je celkem 8 stavových strojů - 4 v každém PIO) se skládá ze sady registrů a určité řídicí logiky.

# Micropython

- <https://docs.micropython.org/en/latest/rp2/general.html>
- <https://docs.micropython.org/en/latest/rp2/tutorial/intro.html>

---

Micropython není jedinou implementací Pythonu pro mikrokontroléry

- [CircuitPython](#), derivát Micropythonu, udržuje Adafruit, [rozdíly](#)
- MicroPython pro [BBC micro:bit](#)

# Obecná kontrola RPi

## Modul `machine`

- Abstraktní vrstva pro komunikaci s HW (shodná pro více kontrolérů)

```
import machine

machine.freq()           # get the current frequency of the CPU
machine.freq(240000000) # set the CPU frequency to 240 MHz
```

## Modul `rp2`

- funkcionálnita specifická pro RP2040

```
import rp2
```

# Modul machine

- Modul obsahuje specifické funkce související s hardwarem na konkrétní desce.
- Většina funkcí modulu umožňuje přímý a neomezený přístup k hardwarovým blokům systému (jako je procesor, časovače, sběrnice atd.) a jejich ovládání.
- Při nesprávném použití může dojít k poruše, zablokování, v krajním případě i k poškození hardwaru.
- Na vhodném hardwaru nabízí MicroPython možnost psát obsluhy přerušení v jazyce Python. Obsluhy přerušení - známé také jako rutiny obsluhy přerušení (ISR) - jsou definovány jako **callback funkce**. Ty se provádějí v reakci na událost, jako je spuštění časovače nebo změna napětí na pinu.

# Přístup do paměti

- Modul definuje tři objekty pro přímý přístup do paměti

## machine.mem8

- Zápis/čtení 8 bitů paměti

## machine.mem16

- Zápis/čtení 16 bitů paměti

## machine.mem32

- Zápis/čtení 32 bitů paměti

# Příklad přístup do paměti

- Příklad specifický pro platformu STM32

```
import machine
from micropython import const

GPIOA = const(0x48000000)
GPIO_BSRR = const(0x18)
GPIO_IDR = const(0x10)

# set PA2 high
machine.mem32[GPIOA + GPIO_BSRR] = 1 << 2

# read PA3
value = (machine.mem32[GPIOA + GPIO_IDR] >> 3) & 1
```

# Reset zařízení 1/2

## `machine.reset()`

- Reset zařízení se stejným efektem jako externím Reset signálem

## `machine.soft_reset()`

- Soft reset interpretu, odstraní všechny objekty Pythonu a resetuje haldu Pythonu.
- Pokusí se zachovat způsob, kterým je uživatel připojen k MicroPython REPL (např. sériový, USB, Wifi).

## `machine.reset_cause()`

- Vrátí příčinu resetu
- Příčinu popisují [konstanty](#)

# Reset zařízení 2/2

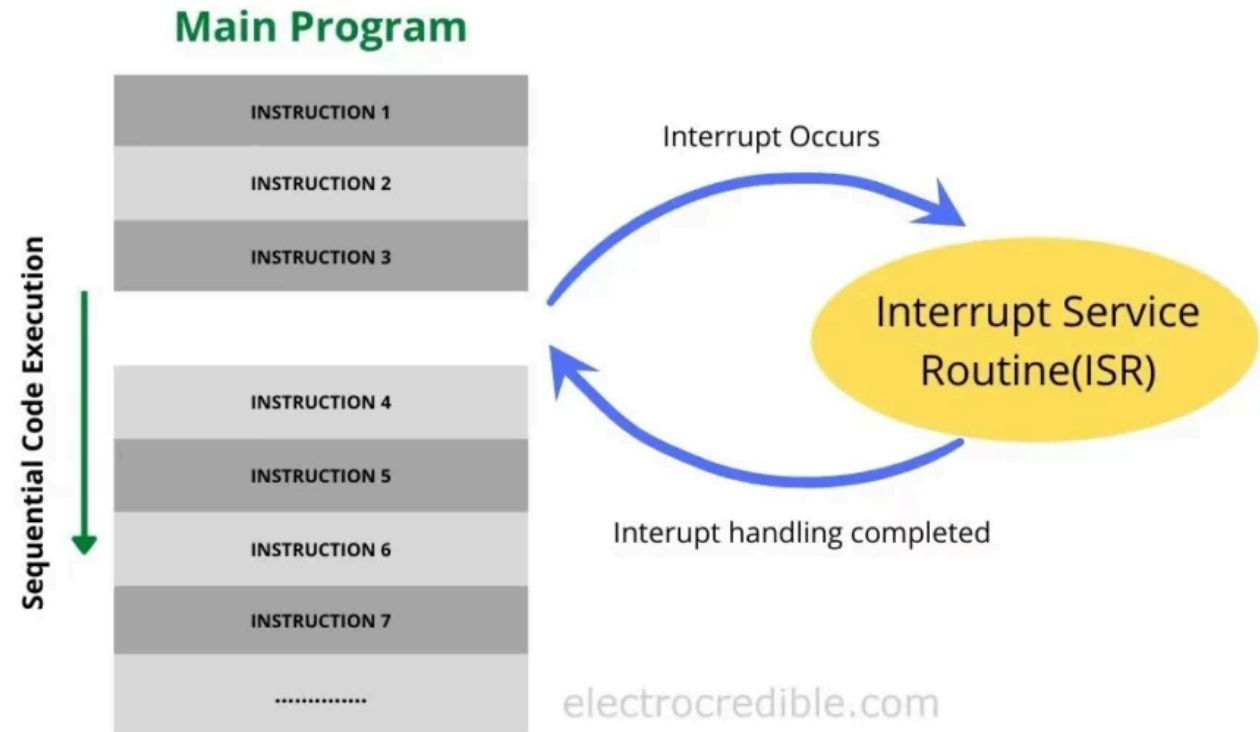
## `machine.bootloader([value])`

- Resetuje zařízení a vstoupí do jeho zavaděče.
- Obvykle se používá k uvedení zařízení do stavu, kdy je možné naprogramovat nový firmware.
- Některé porty podporují předání volitelného argumentu `value`, který může řídit, do kterého zavaděče se má vstoupit nebo co se mu má předat.



# Přerušení

- Přerušeni zpracovávají části softwaru nazývané Interrupt Service Routine(ISR).
- Když dojde k přerušeni, procesor začne provádět kód v rámci této rutiny.
- Po dokončení úkolu v rutině pokračuje procesor ve vykonávání kódu od místa, kde skončil.



# Přerušeni

- přerušeni lze zakázat (vypnout) a opětovně povolit.
- Některé subsystemy vyžadují přerušeni ke správnému fungování, takže jejich dlouhodobé vypnutí může ohrozit funkčnost jádra (např. watchdog).
- Přerušeni by měla být zakázána pouze na minimální dobu a poté opět povolena do předchozího stavu.

```
import machine

# Disable interrupts
state = machine.disable_irq()

# Do a small amount of time-critical work here

# Enable interrupts
machine.enable_irq(state)
```

# Přerušeni

## `machine.disable_irq()`

- Zakáže požadavky na přerušeni.
- Vrací předchozí stav IRQ, který by měl být považován za neprůhlednou hodnotu.
- Tato návratová hodnota by měla být předána funkci `enable_irq()` pro obnovení přerušeni do původního stavu před voláním `disable_irq()`.

## `machine.enable_irq(state)`

- Znovu povolí požadavky na přerušeni.
- Parametrem `state` by měla být hodnota, která byla vrácena z posledního volání funkce `disable_irq()`.

# Napájení

## `machine.freq([Hz])`

- Vrací frekvenci procesoru v Hz. Na některých portech lze tuto funkci použít také k nastavení frekvence procesoru zadáním Hz.

## `machine.idle()`

- Zastaví takt procesoru, což je užitečné pro snížení spotřeby energie kdykoli během krátkých nebo dlouhých období.
- Periferie pokračují v práci a provádění se obnoví, jakmile je vyvoláno jakékoli přerušení (na mnoha portech to zahrnuje přerušení systémového časovače, ke kterému dochází v pravidelných intervalech v řádu milisekund).

# Napájení

`machine.lightsleep([time_ms])`

`machine.deepsleep([time_ms])`

- Zastaví provádění programu a pokusí se o přechod do stavu nízké spotřeby.
- Pokud je zadáno `time_ms`, bude to maximální doba v milisekundách, po kterou bude uspání trvat. Jinak může uspání trvat neomezeně dlouho.
- S časovým limitem nebo bez něj se může provádění programu kdykoli obnovit, pokud dojde k události, které vyžadují zpracování. Takové události nebo zdroje probuzení by měly být nakonfigurovány před uspáním, jako je změna pinu nebo časový limit RTC.

# Napájení 3/4

Přesné chování a možnosti úspory energie u režimů **lightsleep** a **deepsleep** jsou velmi závislé na základním hardwaru, ale obecné vlastnosti jsou následující:

- **lightsleep** zachovává RAM a stav.
  - Po probuzení se pokračuje v provádění od bodu, kde byl požadován spánek, všechny subsystemy jsou funkční.
- **deepsleep** nesmí zachovat paměť RAM ani žádný jiný stav systému (např. periférie nebo síťová rozhraní).
  - Po probuzení je provádění obnoveno z hlavního skriptu, podobně jako při tvrdém nebo zapnutém resetu.
  - Funkce `reset_cause()` vrátí hodnotu `machine.DEEPSLEEP`, kterou lze použít k rozlišení probuzení v hlubokém spánku od jiných resetů.

# Napájení 4/4

`machine.wake_reason()`

- Vrací důvod probuzení ze spánku
- Zdroj popisují [konstanty](#)

## Další funkce

Další užitečné funkce jsou shrnuty na této [stránce](#).

# Časovač

- Periferie systémového časovače RP2040 poskytuje globální mikrosekundovou časovou základnu a generuje pro ni přerušení.
- Současně je k dispozici softwarový časovač, kterých je k dispozici neomezený počet (pokud to paměť dovolí).
- Časovač je popsán třídou `machine.Timer`

```
from machine import Timer

tim = Timer(period=5000, mode=Timer.ONE_SHOT, callback=lambda t:print(1))
tim.init(period=2000, mode=Timer.PERIODIC, callback=lambda t:print(2))
```



# GPIO

- GPIO popisuje třída `machine.Pin`

```
from machine import Pin

p0 = Pin(0, Pin.OUT)      # create output pin on GPIO0
p0.on()                   # set pin to "on" (high) level
p0.off()                  # set pin to "off" (low) level
p0.value(1)               # set pin to on/high

p2 = Pin(2, Pin.IN)      # create input pin on GPIO2
print(p2.value())        # get value, 0 or 1

p4 = Pin(4, Pin.IN, Pin.PULL_UP) # enable internal pull-up resistor
p5 = Pin(5, Pin.OUT, value=1) # set pin high on creation
```

# GPIO s přerušením

```
import time
from machine import Pin

pin_button = Pin(14, mode=Pin.IN, pull=Pin.PULL_UP)
pin_led     = Pin(16, mode=Pin.OUT)

def button_isr(pin):
    pin_led.value(not pin_led.value())

pin_button.irq(trigger=Pin.IRQ_FALLING, handler=button_isr)

while True:
    ...
```

# ADC

- ADC je popsán třídou `machine.ADC`

```
from machine import ADC
import utime

sensor_temp = ADC(4)
conversion_factor = 3.3 / (65535)

while True:
    reading = sensor_temp.read_u16() * conversion_factor
    temperature = 27 - (reading - 0.706)/0.001721
    print(temperature)
    utime.sleep(2)
```

# UART

- UART je popsán třídou [machine.UART](#)
- RP2040 má dvě UART periférie (UART0 a UART1)
  - Programovatelná délka dat (5-8 bitů) a počtu stop bitů (1 nebo 2)
  - FIFO v obou směrech až 32 bytů
- Přerušením lze sledovat příchod nebo odchod dat, status zařízení, chybu komunikace nebo timeout příjmu dat
- Obě zařízení lze konfigurovat na různých dvojicích TX a RX pinů
  - UART0: GP0-GP1, GP12-GP13, GP16-GP17
  - UART1: GP4-GP5, GP8-GP9

# UART

```
from machine import Pin, UART
import time

uart = UART(1, baudrate=9600, tx=Pin(4), rx=Pin(5))
uart.init(bits=8, parity=None, stop=2)

led = Pin("LED", Pin.OUT)

while True:
    uart.write('t')
    if uart.any():
        data = uart.read()
        if data == b'm':
            led.toggle()
    time.sleep(1)
```

# Programování PIO v Micropythonu

- PIO MCU RP2040 je možné i v rámci Micropythonu programovat na nižší úrovni
- PIO assembler: JMP, WAIT, IN, OUT, PUSH, PULL, MOV, IRQ, a SET
  - instrukce jsou zaměřené na manipulaci s bity
  - trvají právě jeden strojový cyklus
  - neimplementují žádné aritmetické operace
  - jediná logická operace je přesun s bitovým doplňkem
- Modul [rp2](#) implementuje wrapper pro instrukce v assembleru
  - Např. funkce `set()` vytváří wrapper pro instrukcí **SET**, která přepíná stav GPIO pinu nezávisle na hlavním procesoru.
- Příklady na [github.com](#)

# PIO

- Instrukce jsou vysoce optimalizovány na následující operace
  - pro serializaci a deserializaci proudů bitů do/ze slov,
  - posun slov do FIFO a vybírání slov z FIFO.
- Kromě lze vložit instrukce z externích zdrojů k provedení v libovolném hodinovém cyklu.
- Každá instrukce má navíc vyhrazeno několik bitů pro postranní nastavení (side-set) a/nebo zpoždění instrukce.
  - Side-set umožňuje další přímou manipulaci s bity na pinech GPIO a provádí se paralelně a nezávisle na vlastní instrukci.

# Příklad - generování obdélníkového signálu

- Cílem je vygenerovat signál na pinu 28
- Stavový stroj pracuje na frekvenci `clk_sys` - 125 MHz
  - Každá instrukce trvá právě jeden hodinový takt, lze prodloužit na 32 (1+31)
  - 32 cyklů trvá 256 ns -> frekvence signálu bude 1.953 MHz

```
import time
import rp2
from machine import Pin
@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def blink():
    wrap_target()
    set(pins, 1)    [31]
    set(pins, 0)    [31]
    wrap()
sm = rp2.StateMachine(0, blink, set_base=Pin(28))
sm.active(1)
```



# Příklad - generování obdélníkového signálu

- Program pro PIO je implementován ve funkci `blink()`
- Pro použití s PIO je třeba použít dekorátor `@rp2.asm_pio`
  - Parametry dekorátoru jsou např. nastavení pinu a FIFO
- Funkce `wrap()` a `wrap_target()` označující začátek kódu pro state machine, umožňují realizovat automaticky cyklus bez použití instrukce `JMP`
- Frekvenci signálu lze měnit
  - změnou délky trvání instrukce
  - změnou frekvence stavového stroje

```
sm = rp2.StateMachine(0, blink, freq=2500, set_base=Pin(28))
```