IMP

Mikroprocesorové a vestavěné systémy

Organizace

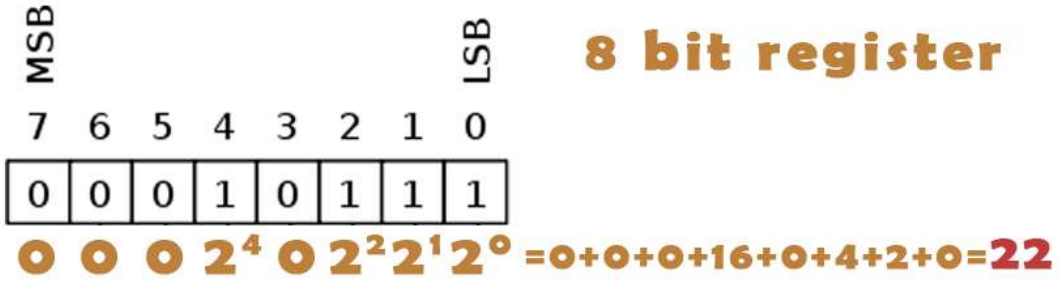
- Přednášky vždy v 10:00 v D105.
- „Demo“ cvičení – jen některé týdny po přednášce (celkem 6).
- Za body:
  - Laboratorní cvičení – od 5. týdne v L306, celkem 4 úlohy, 1x za 14 dní, až 16b.
  - Projekt – samostatné řešení, výběr zadání od 3. týdne, až 14b (pro **zápočet** min. 5).
  - Půlsemestrálka v termínu přednášky dne 4. 11. 2022, až 19b.

  - Písemná zkouška za 51b, minimum 20.

Smysl předmětu

- ukázat, jak to vypadá tam, kde se „dotýká“ hardware a software
  - jak vlastně dosáhnu toho, že psaním kódu se něco stane ve fyzickém světě, že z nehmotných příkazů je najednou něco, co „pohne hmotou“?

To se nejlépe ukazuje na „embedded“, ale platí to obecně.



Smysl předmětu

- Ukázat, co je to „embedded“, čím je specifické a čím je stejné jako jiné počítačové systémy.

- Ukázat, že vlastně i čistý programátor má moc ovlivňovat fyzický svět a když si to neuvědomuje, že může napáchat velké škody, přispět k ekologickým katastrofám.

- Ukázat, že i každý si dneska může snadno naprototypovat nějaký jednoduchý gadget (pokud chápe základní principy).

Vestavné systémy

Mikroprocesorové a vestavěné systémy (IMP)

Richard Růžička

Fakulta informačních technologií VUT v Brně

Proč vestavěné systémy?

Zpočátku byly systémy realizovány **mechanicky**,

Později pokrok umožnil realizovat systémy **elektricky** (elektronicky),

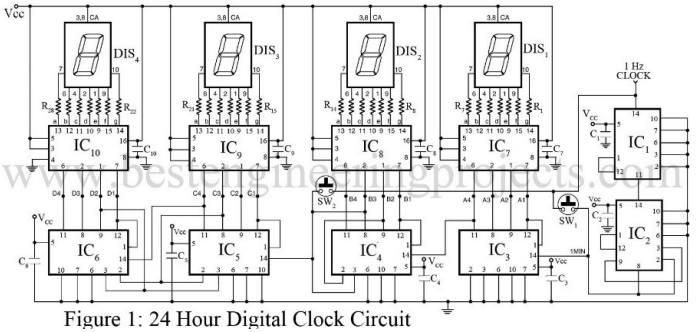


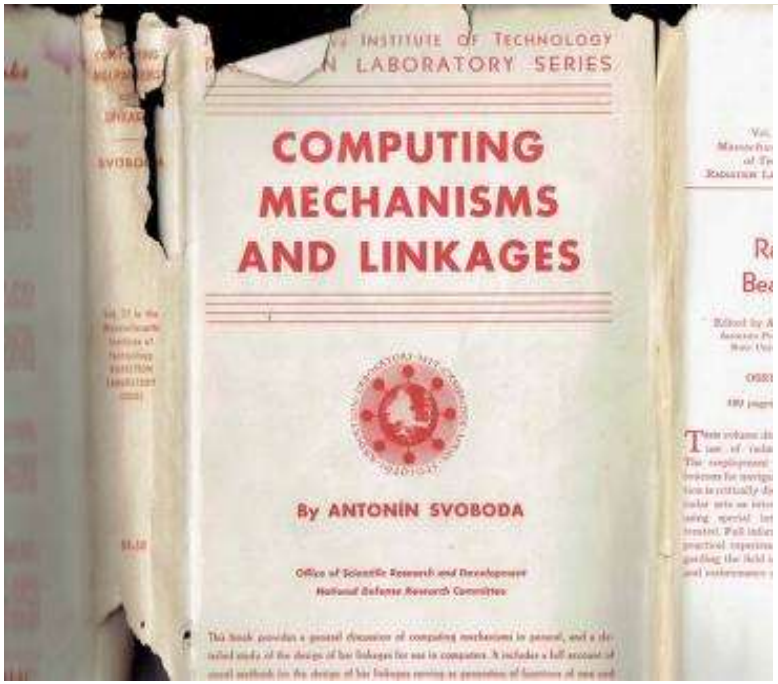
Figure 1: 24 Hour Digital Clock Circuit

Dnes bychom nejraději realizovali systémy **jako software**.

```
if (miliseconds >= 1000) {
  seconds++;
  miliseconds=0;
}

if (seconds >= 60) {
  minutes++;
  seconds = 0;
}

if (minutes >= 60) {
  hours++;
  minutes = 0;
}

if (hours >= 24) {
  hours = 0;
}
```

Proč vestavěné systémy?

Zpočátku byly systémy realizovány **mechanicky**.



← Kniha A. Svobody z roku 1948

Antonín Svoboda, matematik z ČVUT, v ní popisuje, jak realizovat mechanicky výpočetní systémy.
V letech 1936 – 1938 navrhl mechanický počítač pro řízení protiletadlové palby pro armádu ČSR.

Během války pracoval na tématu na MIT v USA a navrhl radarem řízený protiletadlový systém pro US NAVY (MARK 56).

1950 navrhoval první čs. elektronický počítač SAPO
1956 elektronkový počítač EPOS
1964 odchází opět do USA a stává se profesorem na UCLA

Mechanické systémy

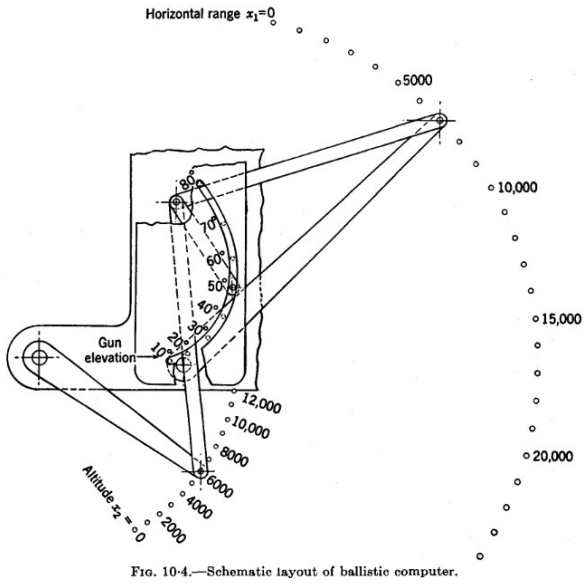
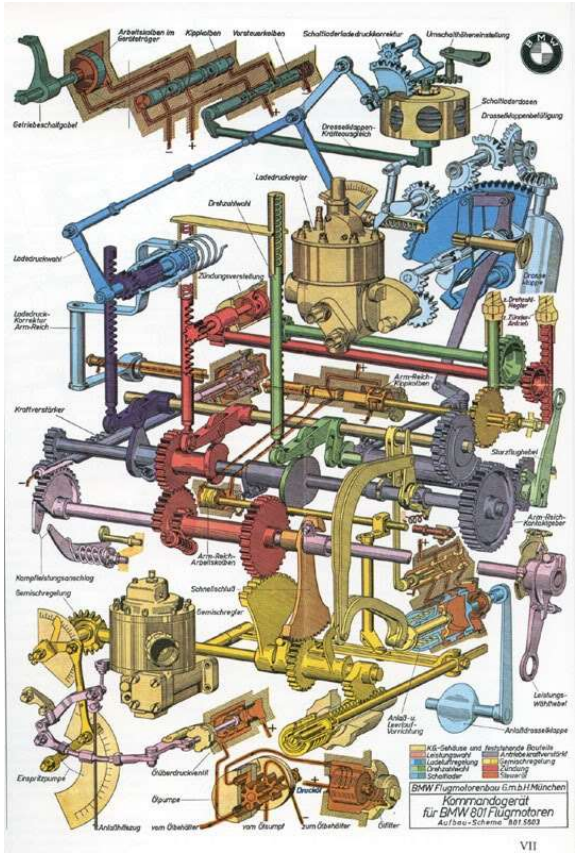


FIG. 10·4.—Schematic layout of ballistic computer.

Jednoduché schéma balistického počítače ze Svobodovy knihy.



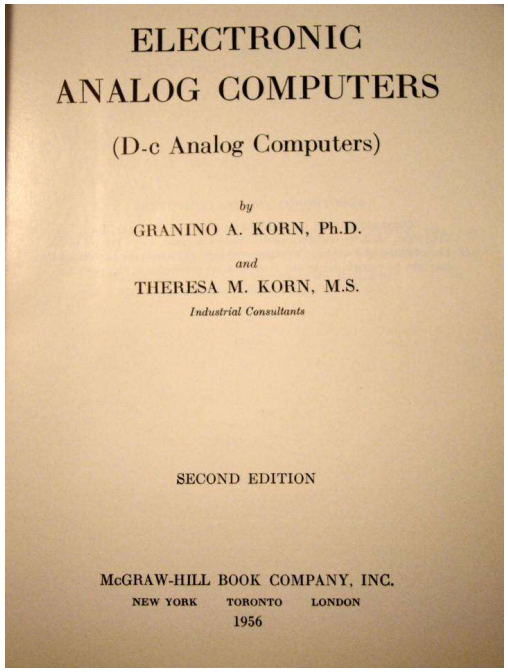
Kommandogerät für BMW 801 Flugmotoren

Mechanické systémy

- Nevýhody:
  - přesnost a spolehlivost závisí na opotřebení – mění se s časem (dobou používání),
  - citlivost na prach, teplotu, otřesy,
  - vyžaduje častou údržbu,
  - náročný návrh – součásti se konstruují přímo pro konkrétní účel, obtížné dodatečné změny.

Elektronické systémy

- Pokrok v technologiích (elektronika pevné fáze látek) umožňuje navrhovat a vyrábět systémy jako elektronický obvod.


ELECTRONIC
ANALOG COMPUTERS

(D-c Analog Computers)

by

GRANINO A. KORN, Ph.D.

and

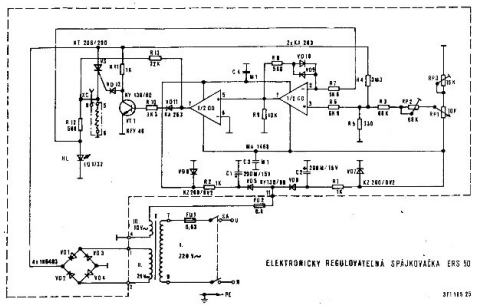
THERESA M. KORN, M.S.

Industrial Consultants

SECOND EDITION

McGRAW-HILL BOOK COMPANY, INC.
NEW YORK TORONTO LONDON
1956

… zpočátku spojité (analogové), což mohou být v podstatě elektronické obdoby mechanických předchůdců



a později diskrétní (číslicové).

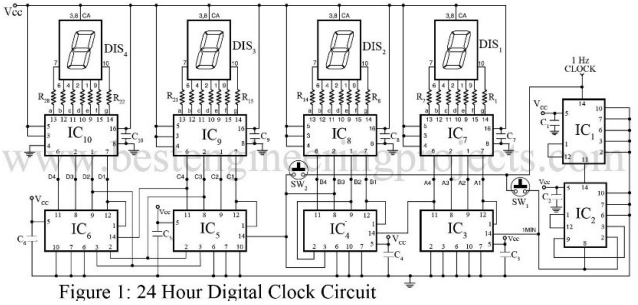


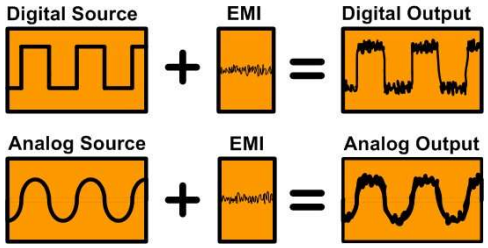
Figure 1: 24 Hour Digital Clock Circuit

Elektronické systémy

V čem je pokrok oproti mechanickým systémům?

- Odstraněno mechanické opotřebení!
- Menší a lehčí,
- flexibilnější, pokud jde o zdroj a přívod energie, snímání vstupních veličin (elektrická vazba je lepší než mechanická) a realizaci výstupů (elektrické signály se pak převádí na silové/energetické působení téměř libovolného rozsahu, mechanický výstup obvykle „téměř nic neutáhne“ a ještě má při přenosu ztráty).
- Jednodušší návrh – součásti se vyrábí průmyslově (kvalita), návrhář je prostě koupí a navrhne pouze finální sestavu.

- Číslicový obvod má navíc jasně definovanou přesnost, na kterou nemá vliv stárnutí součástek, okolní šum, kolísání napájení atd.



Dnes: systém jako software

V čem je pokrok oproti elektronickým systémům?

- Snadný a komfortní návrh – popíšeme chování systému (v nějakém vhodném jazyce) – a to je celé! **Vytvářením popisu vlastně rovnou vzniká sám systém**.

  Komfort - umožňuje navrhovat složitější systémy,
  - umožňuje navrhovat levněji,
  - otevírá možnosti více lidem realizovat své nápady,

- Vysoká flexibilita – systém lze snadno změnit, doplnit, rozšířit, opravit.

Příklad: chci postavit digitální hodinky

= elektronický přístroj, který má měřit a zobrazovat čas.

☞ Porovnejme způsob realizace čistě hardwarové a softwarové.

Jak na to?

… navrhneme elektronický obvod.

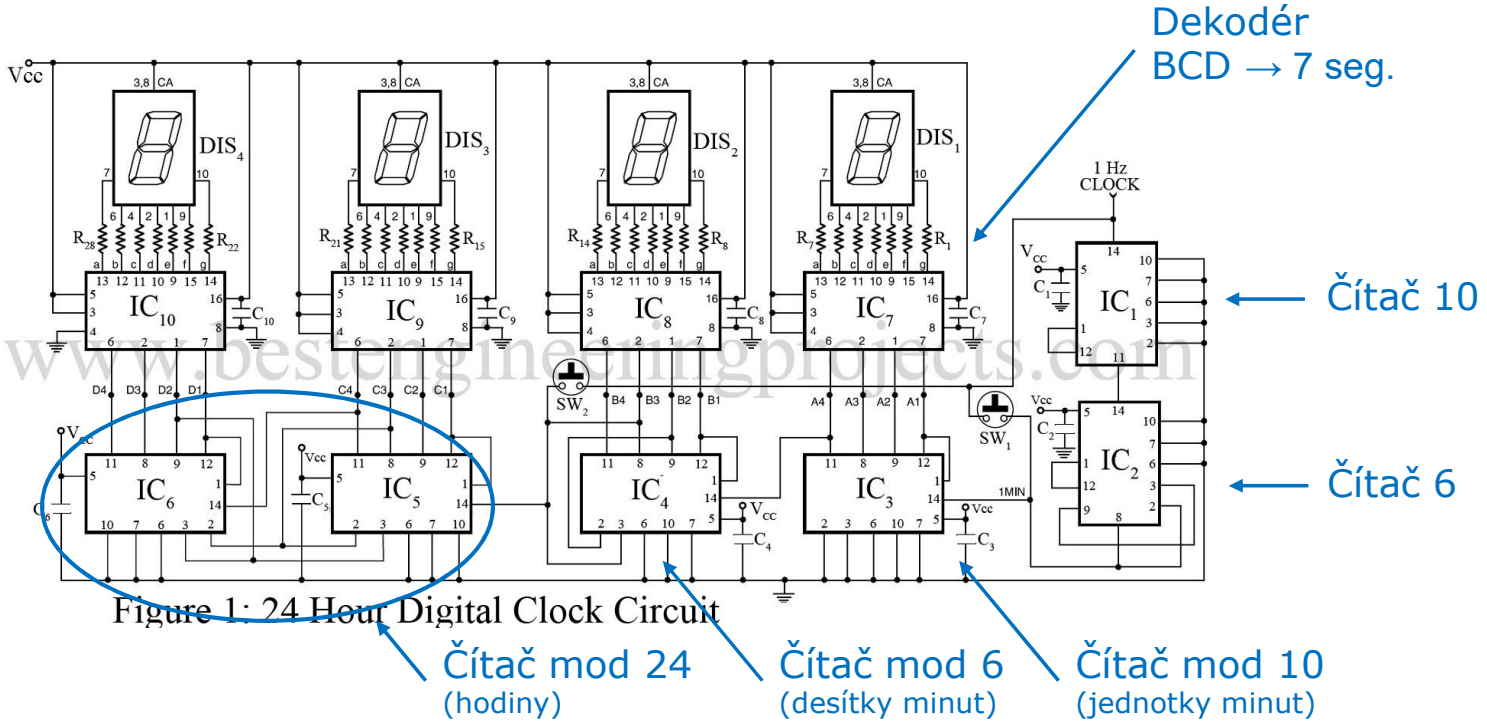
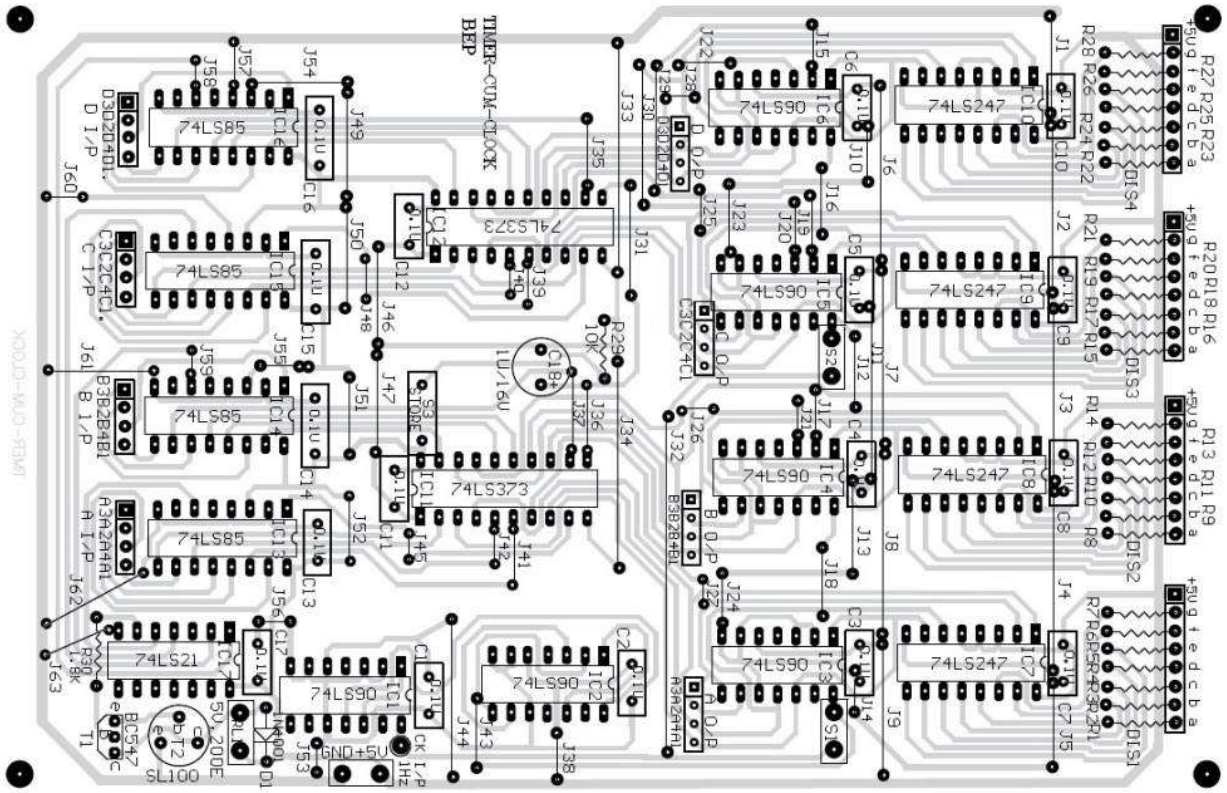


Figure 1: 24 Hour Digital Clock Circuit

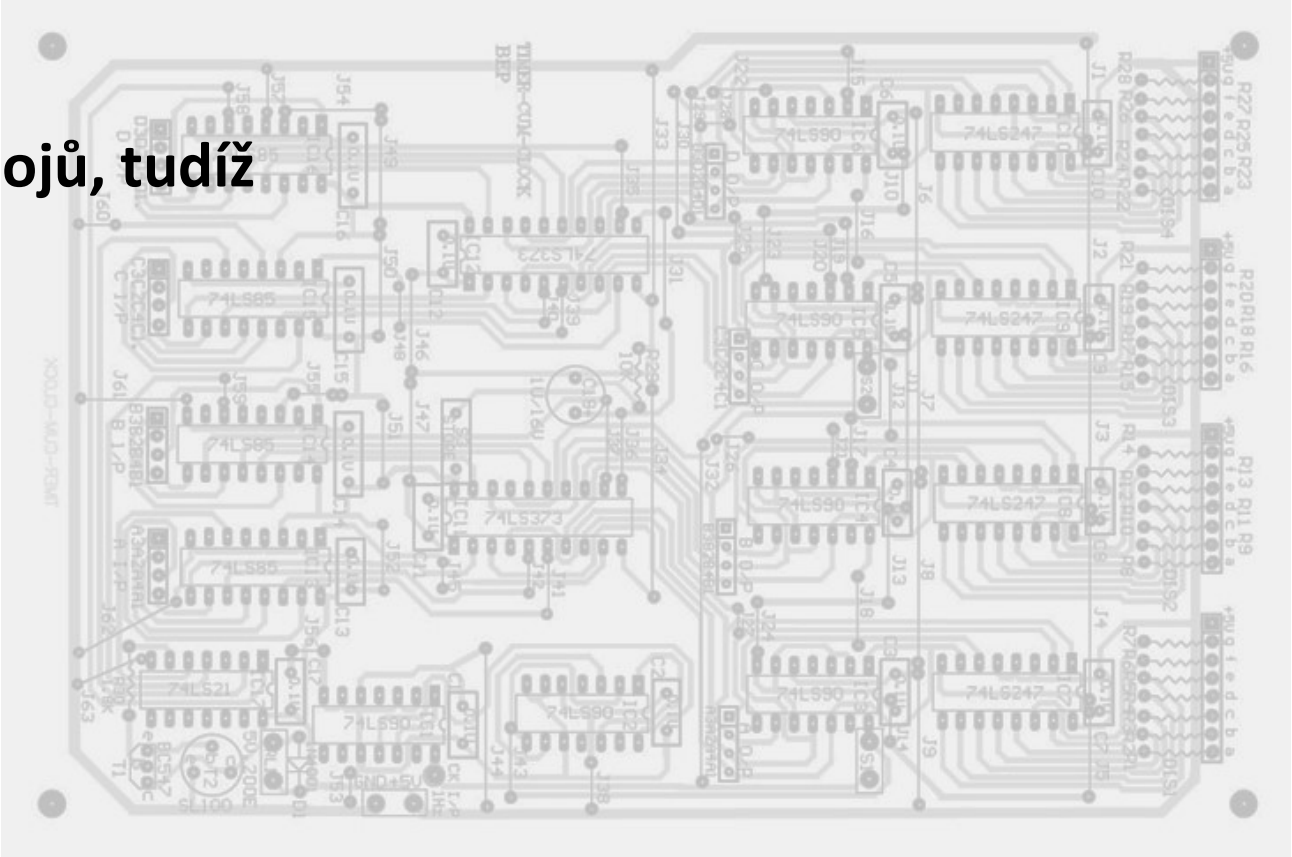
Digitální hodiny jako elektronický obvod

Fyzická realizace na desce plošných spojů:



Digitální hodiny jako elektronický obvod

- **Pěkné, ale:**
  - **mnoho součástek a spojů, tudíž**
    - drahé,
    - rozměrné a těžké,
    - nespolehlivé.



… leda by se to dalo všechno nějak miniaturizovat a dát na jeden čip.

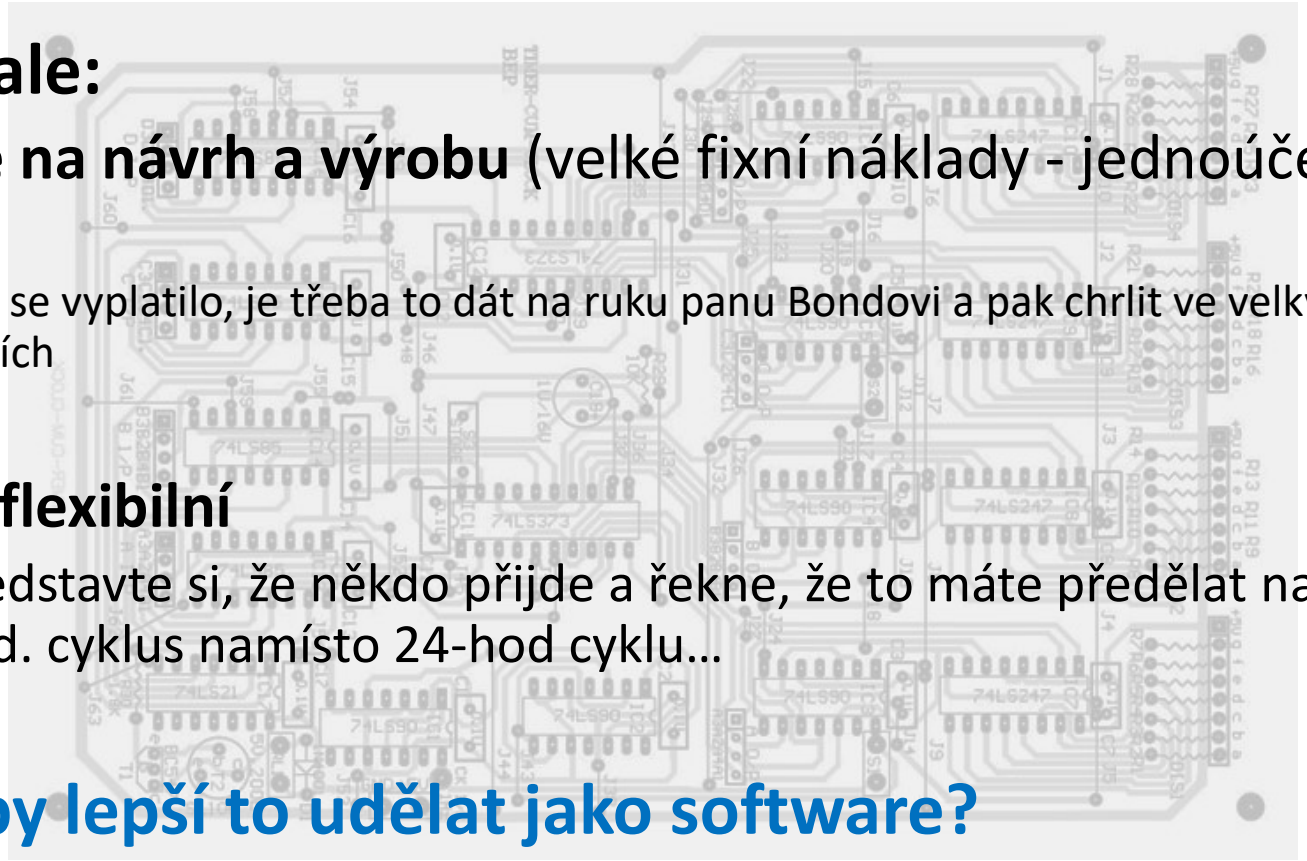
System On Chip (SOC)

Náramkové hodinky
Hamilton Pulsar z roku 1970.

Cena $2100 (přepočteno na dnešní hodnotu dolaru $12000)

V roce 1973 je nosil James Bond ve filmu Žít a nechat zemřít.

Digitální hodiny jako čip (SOC)



- **Pěkné, ale:**
  - **drahé na návrh a výrobu** (velké fixní náklady - jednoúčelový čip)
    - aby se vyplatilo, je třeba to dát na ruku panu Bondovi a pak chrlit ve velkých sériích
  - **málo flexibilní**
    - představte si, že někdo přijde a řekne, že to máte předělat na 12-hod. cyklus namísto 24-hod cyklu…

Nebylo by lepší to udělat jako software?

Digitální hodinky jako software

- Hodinky jako software?
- Proč ne, popsat jejich chování v nějakém programovacím jazyce bychom určitě dokázali:

```
if (miliseconds >= 1000) {
  seconds++;
  miliseconds=0;
}

if (seconds >= 60) {
  minutes++;
  seconds = 0;
}

if (minutes >= 60) {
  hours++;
  minutes = 0;
}

if (hours >= 24) {
  hours = 0;
}
```

```
#define N0 0b00111111
#define N1 0b00000110
#define N2 0b01011011
#define N3 0b01001111
#define N4 0b01100110
#define N5 0b01101101
#define N6 0b01111101
#define N7 0b00000111
#define N8 0b01111111
#define N9 0b01101111

void sn(int number, byte display) {

  byte n;

  switch (number) {
    case 0:
      n = N0; break;
    case 1:
      n = N1; break;
    case 2:
      n = N2; break;
    case 3:
```

Digitální hodinky jako software

OK, program napíšeme, ale na čem poběží, aby to nebylo nakonec horší než „hardwarové“ řešení?

Proč vestavěné systémy?

- Software je nehmotný, nemůže fungovat sám o sobě, musí běžet na počítači.
- Řada systémů **může být provedena jako software v počítači** (aplikace)**,**

ale hodně systémů potřebujeme mít fyzicky. Pouze chceme definovat jejich chování pomocí software.

To vede na **elektronické obvody, do nichž je**, pro účely definování chování (řízení) **vestavěn počítač**.

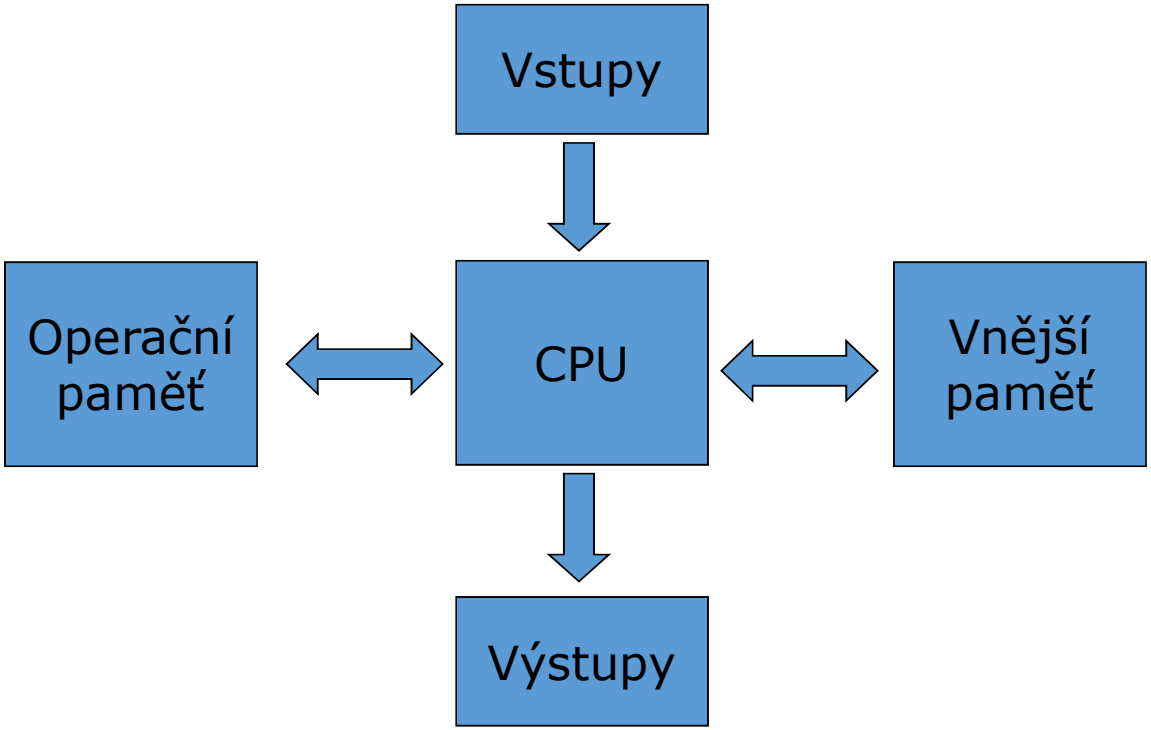
Sám počítač je vlastně také (elektronický) systém.

Proto hovoříme o vestavných systémech.

Co to je počítač?

- Nejobecnější schéma číslicového počítače

  (von Neumannovo schéma) – co potřebujeme, aby běžel program?



Jaký počítač?

- Potřebujeme **určitě procesor a** (operační) **paměť**, aby náš program „někdo“ vykonal.
- **Procesor** asi **nemusí být moc rychlý** (nechceme obvykle přehrávat HD video, ale třeba jen počítat čas) a **paměť nemusí být moc velká**.
- Jako **vstupy a výstupy nepotřebujeme GUI** a k tomu monitor, klávesnici, myš. Stačí jen **umět se připojit na okolní elektrické obvody** – vestavět počítač do aplikace.

Kde vzít takový počítač?

- Ukazuje se, že **výše uvedené požadavky** na počítač **se v průmyslu objevují „poměrně často“** (desítky miliard kusů ročně).

➢ Proto existuje velmi prosperující odvětví, které řešení dodává (a nejlevnější doslova za pár korun).

Mikrokontrolér (MCU)

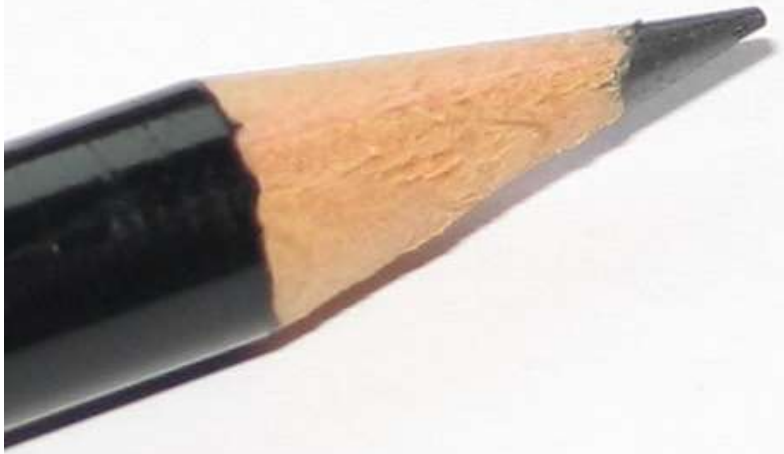
Malý levný počítač, který umí vykonávat program (např. napsaný v C) a podle něj **řídit zařízení** (elektrické obvody), **do kterého je vestavěn**, se nazývá

MIKROKONTROLÉR

Jak vypadají?

Například:

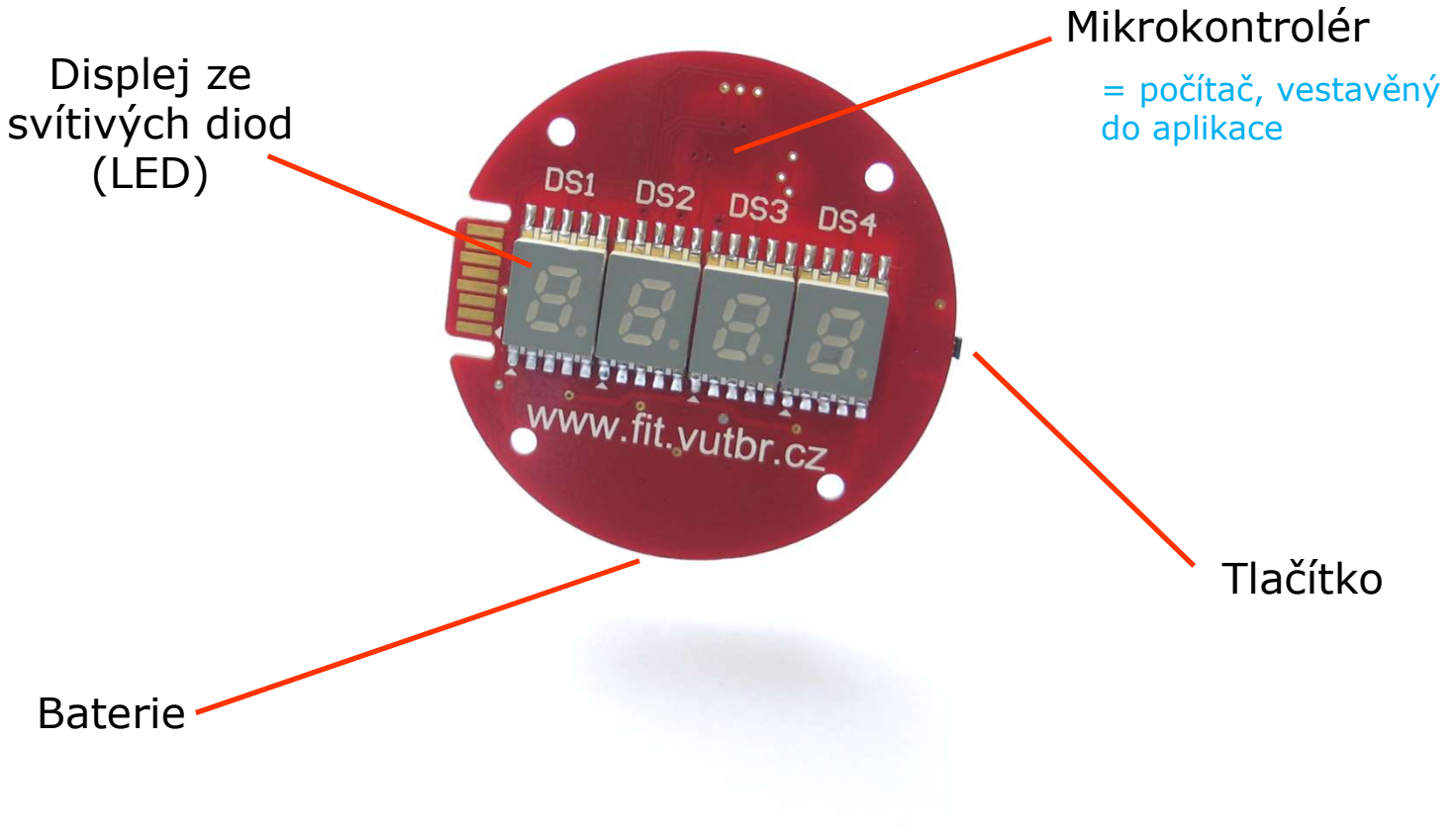
NXP (Freescale)
S08QE8



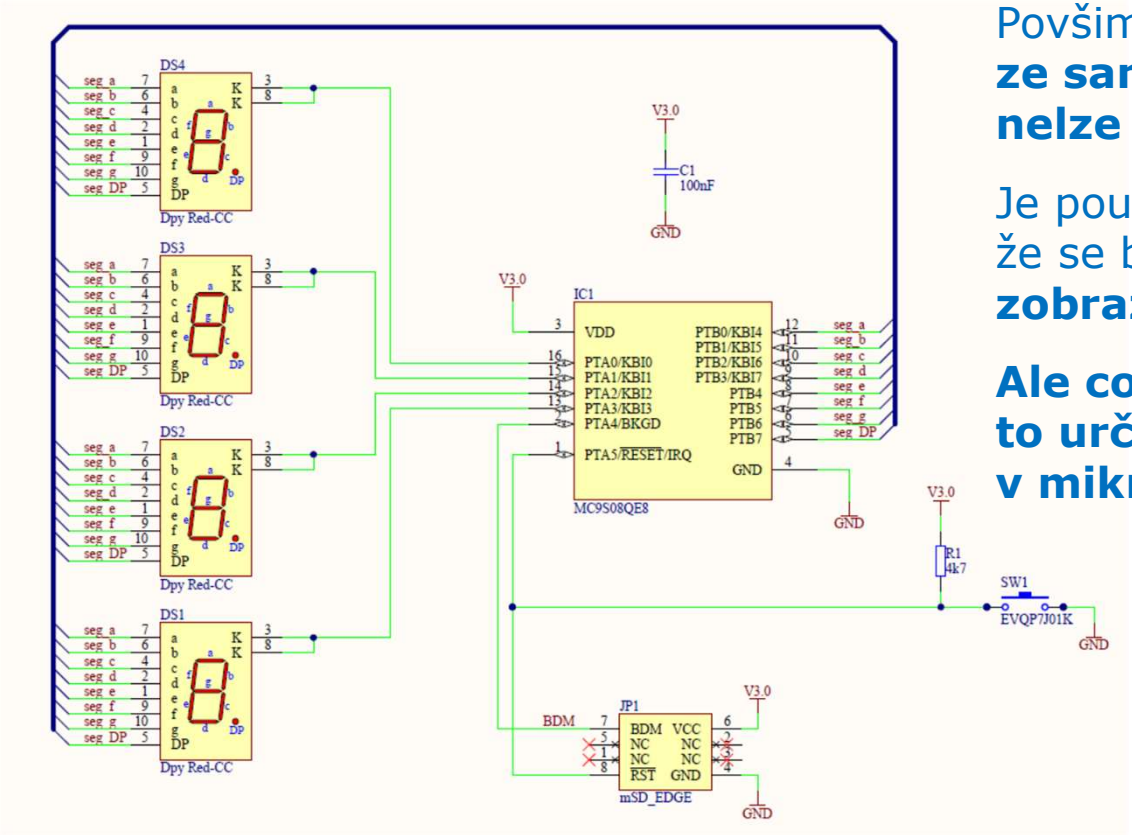
Všechny obvody počítače jsou ukryté v plastovém pouzdře

Kovové vývody slouží k připojení vstupů a výstupů

Hodinky s mikrokontrolérem



Hodinky s mikrokontrolérem



Povšimněte si, že
ze samotného schématu nelze zjistit funkci.

Je pouze zřejmé,
že se bude **něco zobrazovat na displeji.**

**Ale co a jak,
to určuje <u>software</u>
v mikrokontroléru.**

… jestli to budou hodinky nebo třeba počitadlo předmětů, určí jen software, který do mikrokontroléru nahrajeme a na něm spustíme.

Vestavný počítačový systém OBECNĚ

- Máme problém, který bychom **chtěli/potřebovali řešit** (alespoň částečně) **softwarově**.
- **Aby software běžel, potřebujeme počítač(ový systém).**
- Ale nechceme tam mít celý běžný počítač (PC).
- **Navrhneme vestavný počítačový systém** (součástí toho systému bude i software, řešící náš problém).

Pozor!

- Příklad *neslouží* k tomu, abychom demonstrovali, že „softwarové“ řešení je lepší než „hardwarové.“

  To nemusí být úplně vždy pravda, ale

  je celá řada situací, kdy se to hodí – a to

  jsou příležitosti pro nasazení vestavných

  systémů

➢ Najdete nějaké přednosti hardwarového řešení podobných problémů?

Vestavné systémy

(Embedded Systems)

nějakou představu už máme, teď trochu obecněji

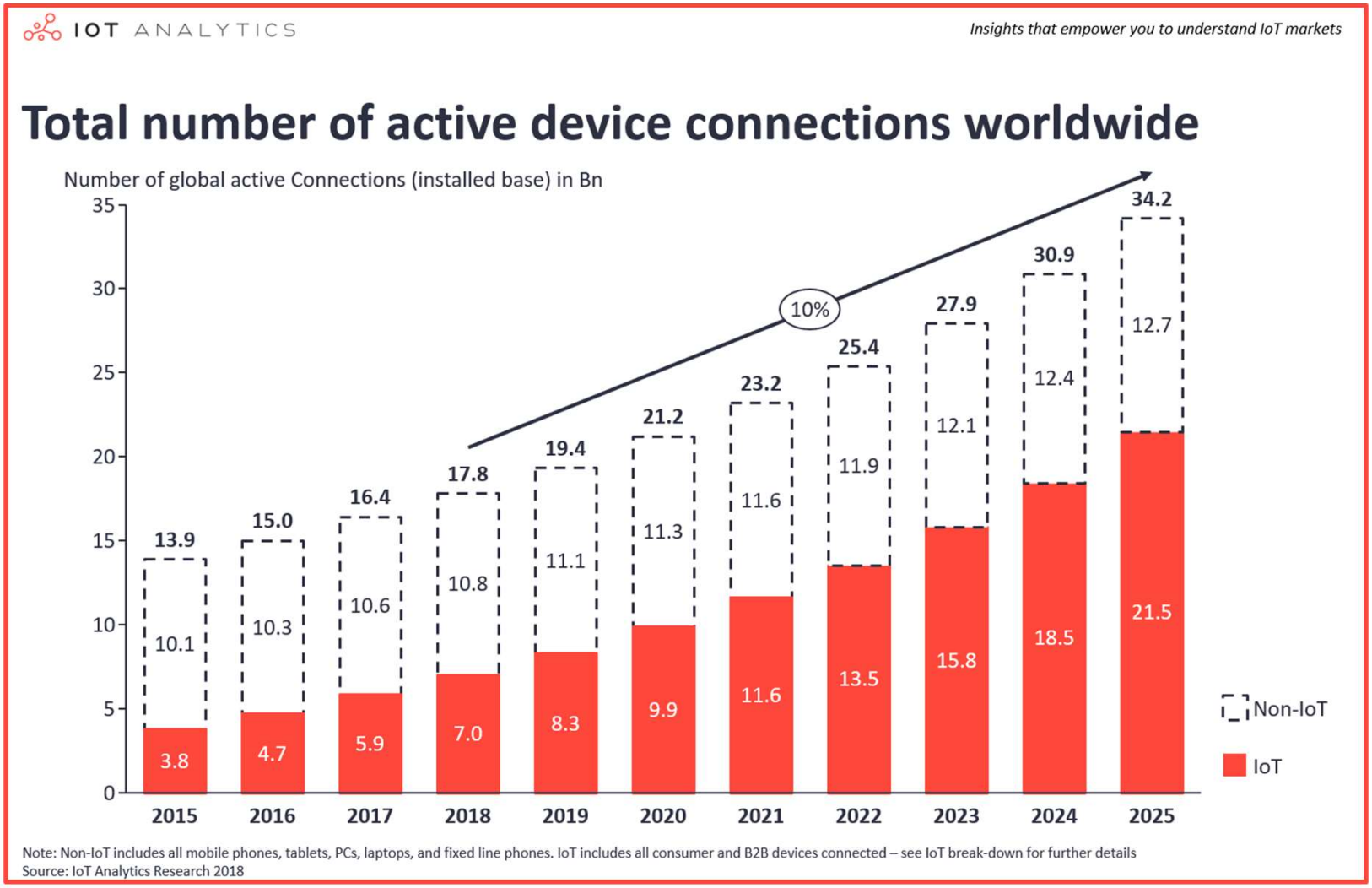
Co je vestavný systém (Embedded System)?

- Dokument řídicího výboru ESTP (Embedded System Technology Platform) Evropské strategické iniciativy definuje embedded systems jako kombinaci hardwaru a softwaru, jejímž smyslem je řídit externí proces, zařízení nebo systém.
- Počítač, který je zabudován do systému, ale pro uživatele není jako počítač viditelný.
- Slouží pro řízení celého systému a poskytuje výpočetní podporu pro jednotlivé komponenty systému

Výběr aplikací vestavných systémů

- Automobily a další stroje
- Komunikační a měřicí přístroje
- Veškeré domácí spotřebiče
- Počítačové komponenty
  a periferie
- Průmyslové aplikace
  (Průmysl 4.0)
- Robotika a autonomní systémy
- Senzorové sítě

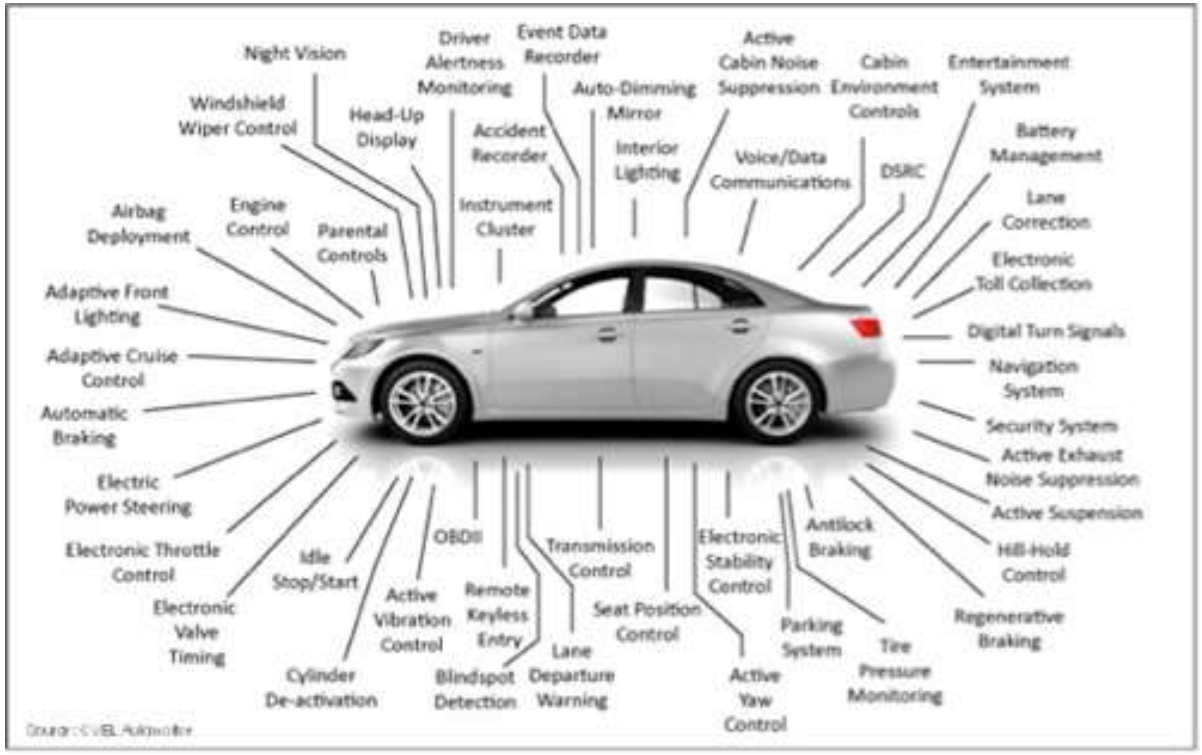
Internet of Things



Vestavěné systémy v automobilech

Početné procesory propojené sítí (~100), s varietou typů:

- 8-bit. – uzamykání dveří, světla, stěrače, atd;
- 16-bit. – většina funkcí; 32-bit – řízení motoru, airbagů,..
- Více jak 30% ceny auta tvoří elektronika



Vestavěný systém x univerzální počítač

- Jeden program po celý život, specifický pro aplikaci.
- Uživatel by neměl ani tušit, že pracuje s počítačem.
- Hlavní interakce nemusí být s člověkem – čidla pro snímání prostředí, ovládání akčních členů.
- Startuje sám bez lidského zásahu.

- Uživatel spouští nejrůznější programy, jaké zrovna potřebuje.
- Periferie hlavně pro interakci s uživatelem, který zadává vstupy a získává řešení.
- Nutnost operačního systému, soubory, ukládání dat,…

Je telefon vestavný systém?

Přístroj na telefonování
řízený počítačem

x

Počítač, který umožňuje
také telefonování

Realizace vestavných systémů

Současné možnosti implementace

přehledné shrnutí

	Implementation	Design Cost	Unit Cost	Upgrades & Bug Fixes	Size	Weight	Power	System Speed
Dedicated Hardware	Discrete Logic	low	mid	hard	large	high	?	very fast
	ASIC	high ($500K/ mask set)	very low	hard	tiny - 1 die	very low	low	extremely fast
	Programmable logic – FPGA, PLD	low to mid	mid	easy	small	low	medium to high	very fast
Software Running on Generic Hardware	Microprocessor + memory + peripherals	low to mid	mid	easy	small to med.	low to moderate	medium	moderate
	Microcontroller (int. memory & peripherals)	low	mid to low	easy	small	low	medium	slow to moderate
	Embedded PC	low	high	easy	medium	moderate to high	medium to high	fast

Počítače pro vestavěné systémy

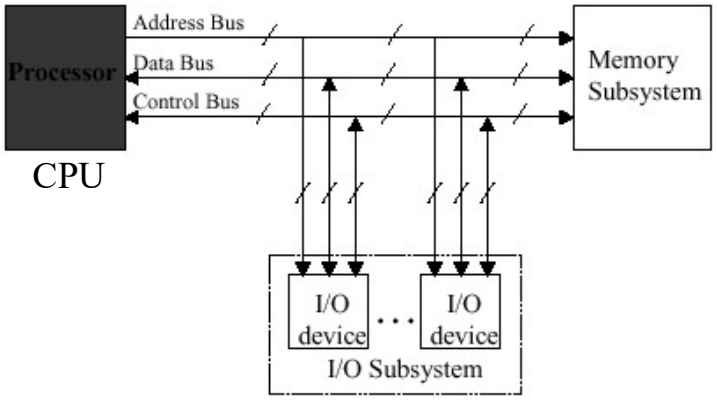
- Lze postavit klasický počítačový systém s CPU, pamětí,
- koupit modul,
- lze osadit mikrokontrolér (MCU),
- použít „soft-core“ např. do FPGA, pro ASIC, …

Klasický počítačový systém

- Pro vestavěný systém se dnes už moc nevidí:

➢ k dispozici je nepřeberné množství MCU různých výkonů,

➢ a naproti tomu klasických CPU je omezený výběr, optimalizace pro klasické „desktop“ a „mobile“ aplikace.

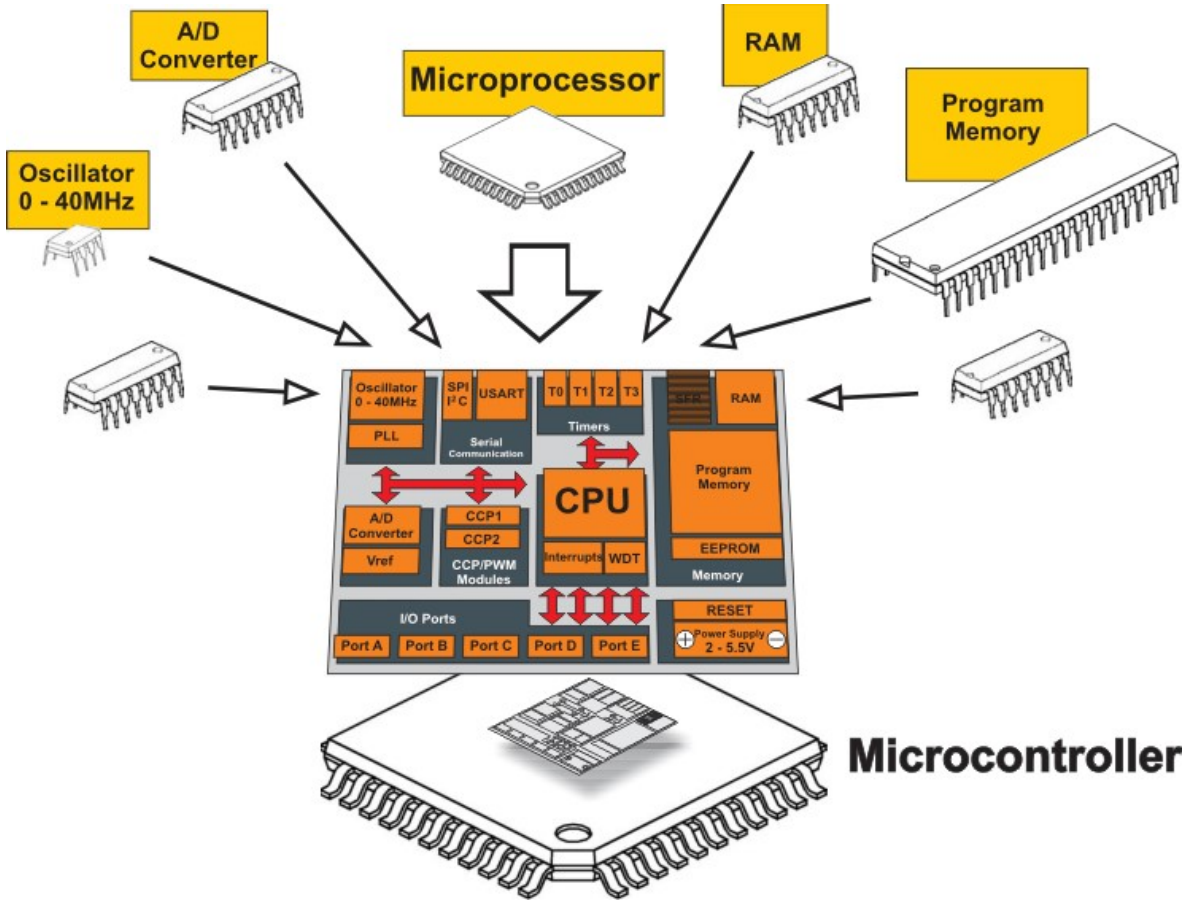


Moduly pro vestavné systémy

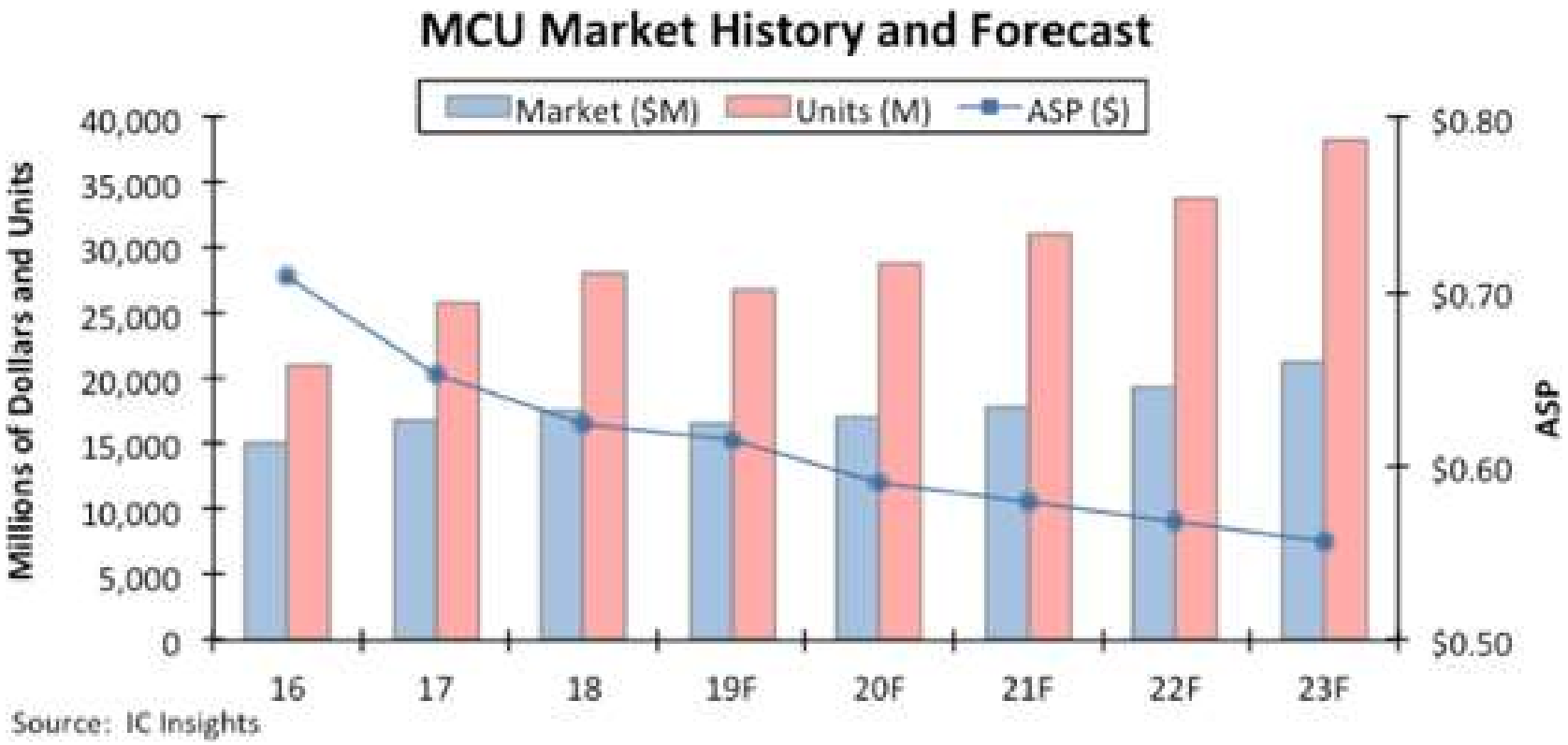
Typicky pro aplikace, kde se požaduje výkon a velká paměť (zpracování multimediálních dat, analýza signálů).

Mikrokontrolér (MCU)

= jednočipový počítač, optimalizovaný pro vestavění do aplikace



Prodej mikrokontrolérů na světovém trhu



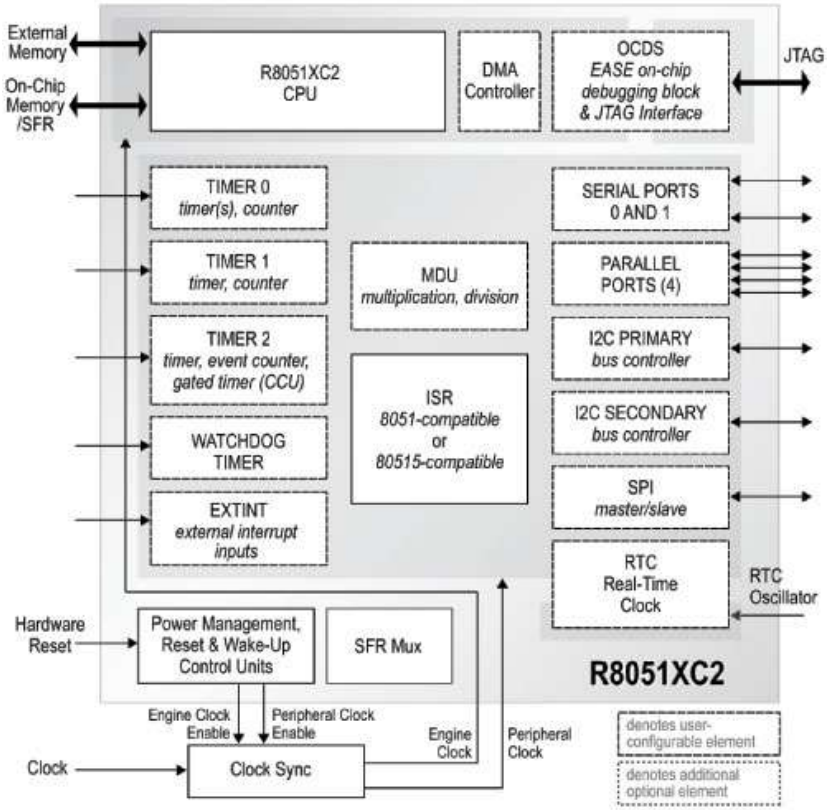
> Tento předmět se zabývá věcmi, kterých se ročně prodá desítky miliard kusů!

Soft Core MCU

- Příklad: CAST, inc.

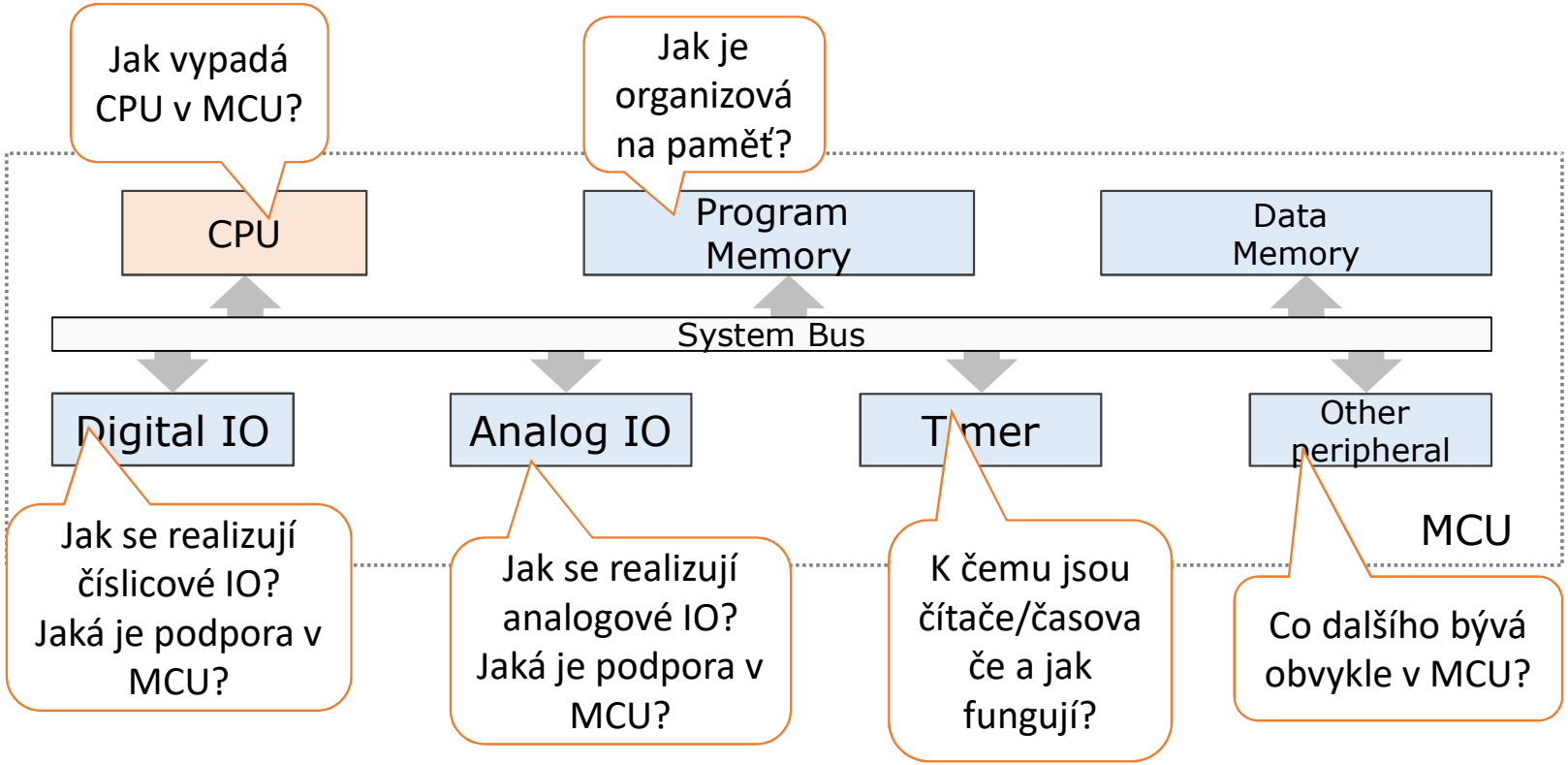
IP jádro 8051IP

- od 2 200 hradel
- od 21μW/MHz
- až 0,5 GHz
- 64k – 16M paměti



Co obvykle najdeme v mikrokontroléru?

… a o čem bude tento předmět.



Co je to mikrokontrolér

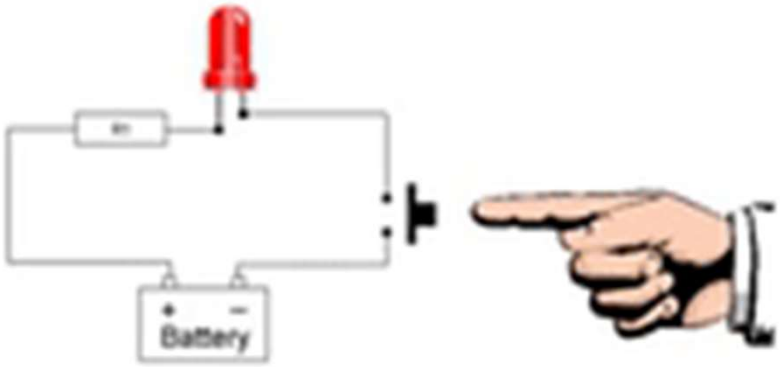
- Mikrokontrolér (MCU) je elektronická součástka, kterou lze vložit do (prakticky libovolného) zařízení jako **„řadič“/“mozek“** – elektronický obvod, přijímající a vydávající signály, kterými se řídí chování tohoto zařízení.

- Chování mikrokontroléru lze běžným způsobem naprogramovat – je to přece počítač.

Naprogramovat mikrokontrolér = popsat chování zařízení pomocí software!

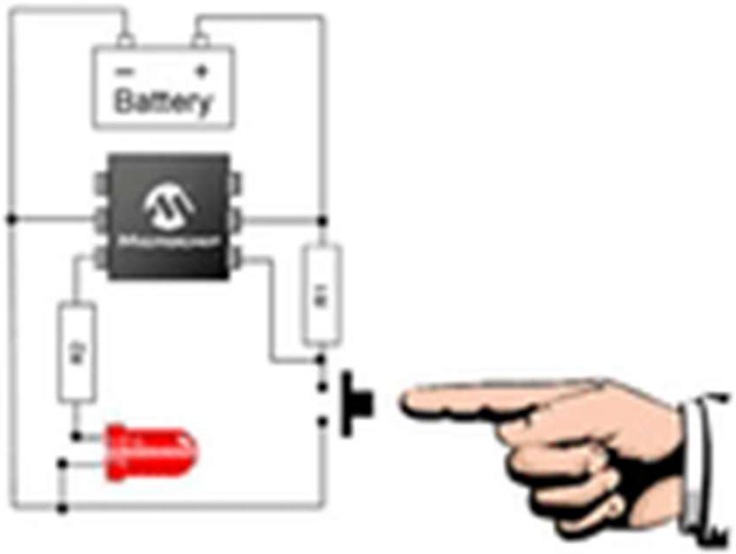
Where would I use a Microcontroller?

- Anywhere you would like to add intelligence…



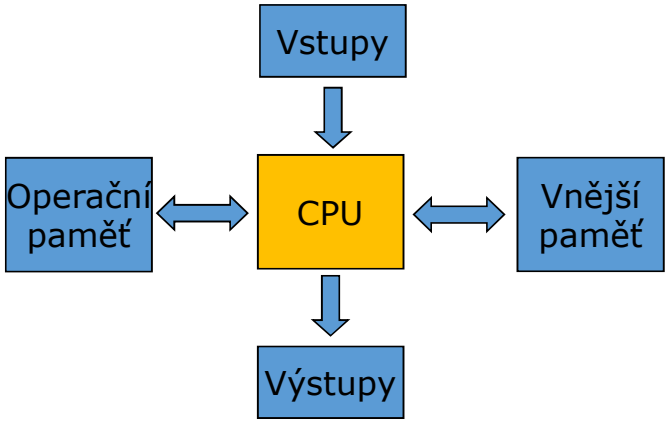
Where would I use a Microcontroller?

- Anywhere you would like to add intelligence…



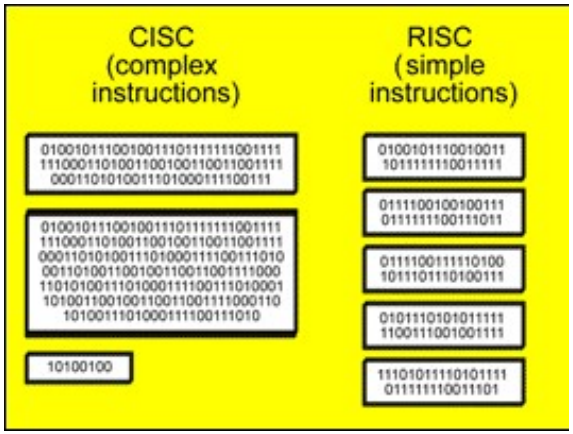
Počítač v mikrokontroléru,

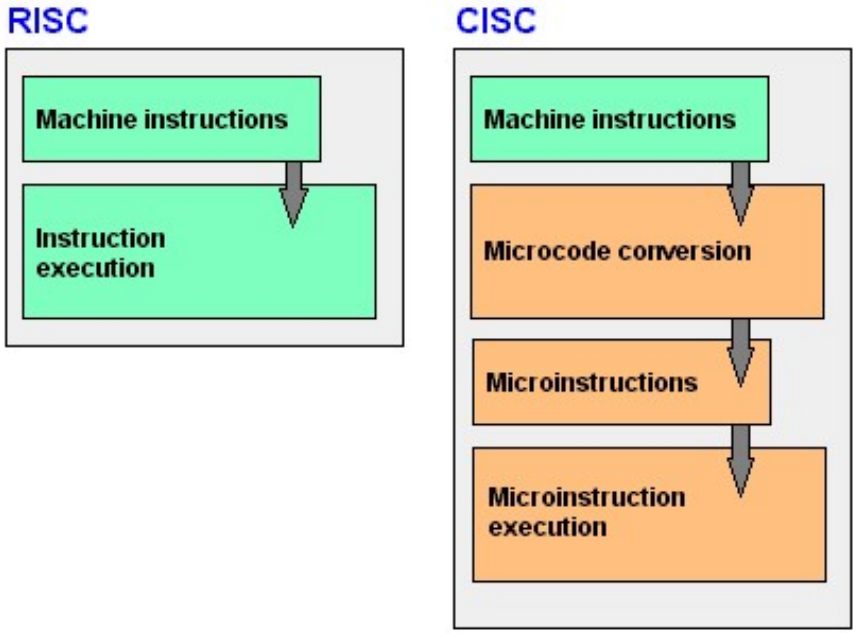
jádro = procesor počítače v mikrokontroléru



CISC x RISC

CISC – Complex Instruction Set Computer
RISC – Reduced Instruction Set Computer





CISC x RISC

CISC

- Mnoho typů instrukcí s mnoha variantami a mnoha adresovacími režimy.
- ➢ Typicky memory-to-memory instrukce (výběr operandů a uložení výsledků je součástí instrukce).
- ➢ Proto nebývá mnoho datových registrů, zpravidla jeden hlavní „střadač“ (Accumulator).
- ➢ Instrukce trvají více taktů.
- ➢ Na čipu dominuje logika pro implementaci instrukcí.

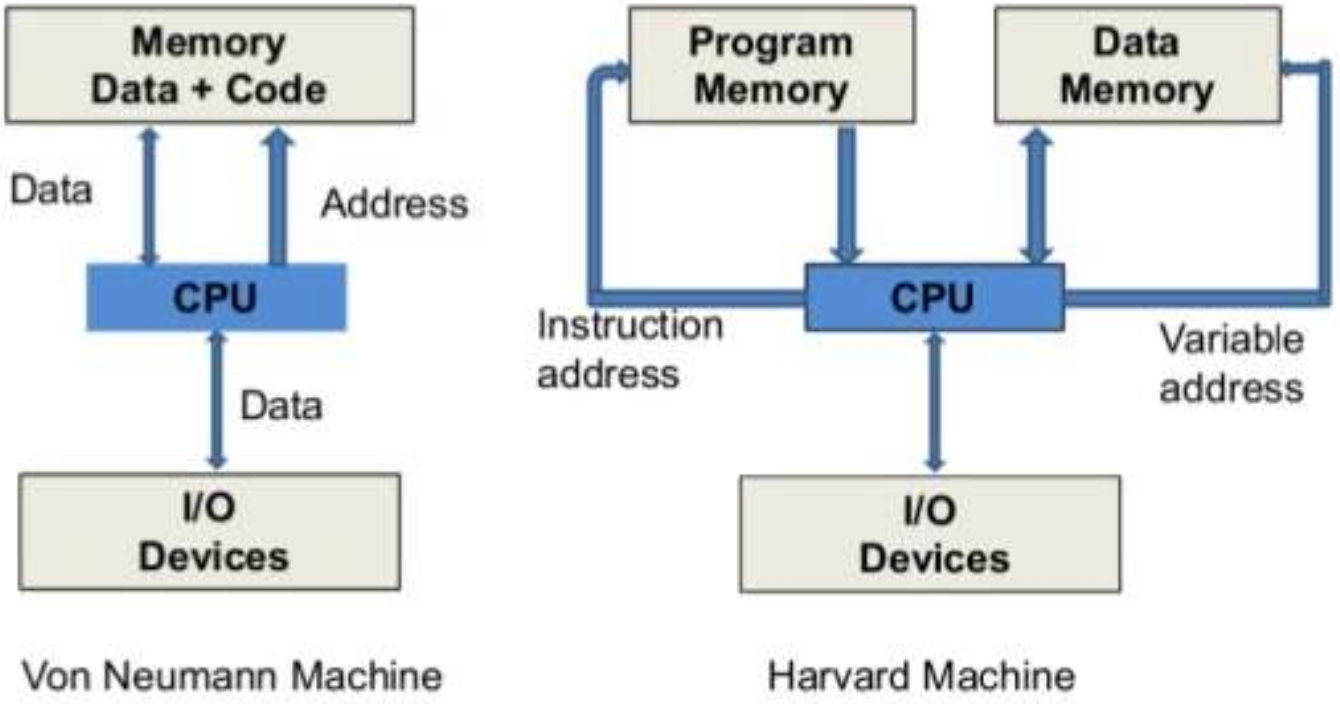
RISC

- Několik základních instrukcí, jednoduché režimy adresování.
- ➢ Typické jsou instrukce register-to-register, proto jsou separátní instrukce „LOAD“ a „STORE.“
- ➢ Typicky mnoho datových registrů.
- ➢ Instrukce trvají pokud možno všechny stejně, ideálně jeden takt.
- ➢ Na čipu dominuje paměť – registry.

CISC x RISC

- Dnešní rychlá paměť (nevadí hodně přístupů) a dobré kompilátory (i z vyššího jazyka lze získat efektivní kód) činí řadu výhod CISC architektur spornými.
- Původní hranice se smazávají, procesory „umí“ hodně instrukcí a přitom je umí provádět rychle a efektivně.

Von Neumannovská x Harvardská architektura



Von Neumannovská x Harvardská architektura

- Jediná operační paměť, jediný adresový prostor.
- ➢ Flexibilnější pro (měnící se) aplikace.
- ➢ Možnost samomodifikujícího se kódu.
- ➢ Sběrnice je úzké hrdlo (instrukce a operandy k nim musíme z paměti vybírat postupně).

- Instrukce a data mají každá „svoji“ paměť – dvojí adresový prostor!
- ➢ Umožňuje použít fyzicky rozdílné technologie paměti – např. pro program FLASH a pro data RAM.
- ➢ Umožňuje použít rozdílnou velikost buňky paměti – např. 8 bitů pro data a 18 bitů pro instrukci.
- ➢ Umožňuje v jednom taktu získat instrukci a zároveň operandy pro ni.

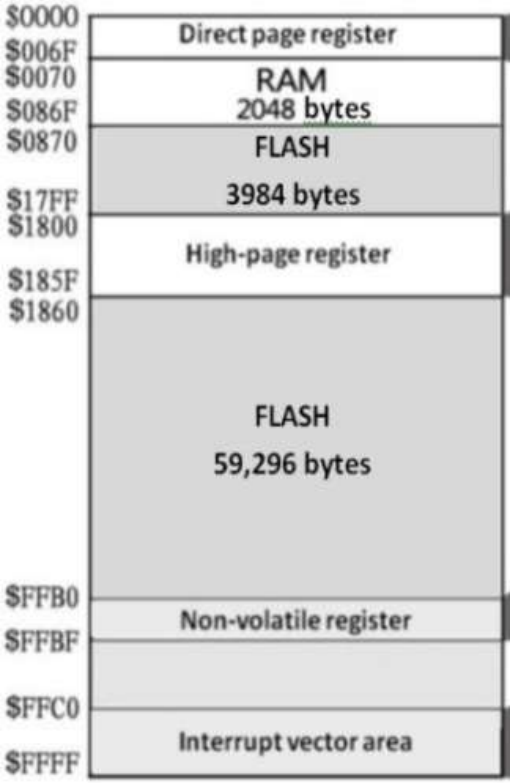
Von Neumannovská x Harvardská architektura

- U univerzálních počítačů zatím vítězí „von Neumann“ díky flexibilitě paměťového prostoru.
- U vestavných systémů (mikrokontrolérů) se používají oba přístupy.

➢ **Program** zde bývá **typicky trvalý** (reprezentuje popis chování systému, ten je daný), ukládá se do paměti typu FLASH, **data** jsou **typicky proměnná** (reflektují momentální stav systému, který se mění, jak se mění podmínky, sledované okolí, režim činnosti atd.)

Příklad von Neumannovského MCU

- Rodina MCU Freescale/NXP S08/S12



Jediný adresový prostor obsahuje paměť programu typu FLASH i paměť dat typu RAM.

Výhody:

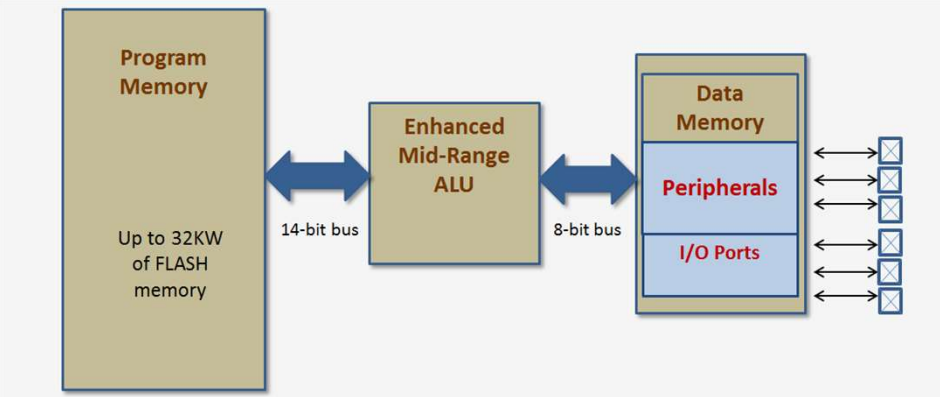
- Konstanty (tabulku konstant) lze mít klidně ve FLASH, přistupuje se k nim běžně jako k jakýmkoli jiným datům (to je u Harvardské architektury trochu problém).
- Program může běžet i v RAM jednak pro možnost samomodifikace, ale hlavně se používá pro low-power aplikace, protože RAM „žere" méně než FLASH (kritický úsek programu zkopírujeme do RAM, tam spustíme a FLASH můžeme vypnout).

Příklad Harvardského MCU

- Rodina Microchip PIC

Paměť programu (FLASH):
- Jedna adresa = vždy jedna celá instrukce (osvobození od bajtového zarovnání) = snadné a rychlé provádění instrukcí, typický RISC se vším všudy.

Paměť dat (RAM):
- V podstatě pole univerzálních a účelových registrů = krátká adresa může být zakódována v instrukcích jako operand = netřeba přidávat LOAD/STORE instrukce.



ARM Cortex M0+

- Jde o tzv. „jádro“ = procesor počítače v mikrokontroléru.
- 32 bitový RISC CPU, optimalizovaný pro MCU a nízkou spotřebu, Von Neumannovská architektura.
- Navrženo jako soupeř malých 8 bitových jader pro MCU (pokud jde o cenu, plochu čipu, spotřebu), ač jde o „ARM“, neumí původní instrukční sadu A32, ale používá „Thumb“.
- Thumb = instrukce kódovány převážně na 16 bitech (u A32 byl čistě „RISCovský“ koncept uniformních 32 bitových instrukcí), což je dobré pro MCU (program zabere méně paměti, spotřebuje méně energie).

Dnes standard jádra pro MCU, používá Freescale, NXP, STMicroelectronics, Texas Instruments,Toshiba, Microchip, Analog Devices, Silicon Labs, …

CPU

CORTEX-M0+

32-bit, Low-Power Processor at an 8-bit Cos

Add Intelligence

The Cortex-M0+ processor has the smallest footprint and lowest power requirements of all the Cortex-M processors. This is well-suited for low-cost devices, including smart sensors and mixed-signal systems on chip (SoCs), adding intelligence to devices that were not capable before.

Reduce Cost and Power

The exceptional code density of Cortex-M0+ significantly reduces memory requirements, which maximizes the use of on-chip Flash memory to save memory cost, reduce memory power, and increase maximum performance. Take advantage of 32-bit processing intelligence at an 8/16-bit processor cost point.

Longer Battery Life

The Cortex-M0+ processor allows developers to optimize power usage for specific applications with built-in, low-power features. With its three highly optimized low-power modes, the processor conserves energy to match processing demands.

ARM Cortex M0+

Table 4-1. System memory map

System 32-bit Address Range	Destination Slave	Access
0x0000_0000–0x07FF_FFFF[1]	Program flash and read-only data (Includes exception vectors in first 196 bytes)	All masters
0x0800_0000–0x1FFF_FBFF	Reserved	—
0x1FFF_FC00-0x1FFF_FFFF[2]	SRAM_L: Lower SRAM	All masters
0x2000_0000-0x2000_0BFF[2]	SRAM_U: Upper SRAM	All masters
0x2000_0C00–0x3FFF_FFFF	Reserved	–
0x4000_0000–0x4007_FFFF	AIPS Peripherals	Cortex-M0+ core & DMA
0x4008_0000–0x400F_EFFF	Reserved	–
0x400F_F000–0x400F_FFFF	General purpose input/output (GPIO)	Cortex-M0+ core & DMA
0x4010_0000–0x43FF_FFFF	Reserved	–
0x4400_0000–0x5FFF_FFFF	Bit Manipulation Engine (BME) access to AIPS Peripherals for slots 0-127[3]	Cortex-M0+ core
0x6000_0000–0xDFFF_FFFF	Reserved	–
0xE000_0000–0xE00F_FFFF	Private Peripherals	Cortex-M0+ core
0xE010_0000–0xEFFF_FFFF	Reserved	–
0xF000_0000–0xF000_0FFF	Micro Trace Buffer (MTB) registers	Cortex-M0+ core
0xF000_1000–0xF000_1FFF	MTB Data Watchpoint and Trace (MTBDWT) registers	Cortex-M0+ core
0xF000_2000–0xF000_2FFF	ROM table	Cortex-M0+ core
0xF000_3000–0xF000_3FFF	Miscellaneous Control Module (MCM)	Cortex-M0+ core
0xF000_4000–0xF7FF_FFFF	Reserved	–
0xF800_0000–0xFFFF_FFFF	IOPORT: GPIO (single cycle)	Cortex-M0+ core

Instrukční sada ARM Thumb 1

Operation		§	Assembler	Updates	Action	Notes
Move	Immediate		`MOVS Rd, #<imm>`	N Z	Rd := imm	imm range 0-255.
	Lo to Lo		`MOVS Rd, Rm`	N Z	Rd := Rm	Synonym of `LSLS Rd, Rm, #0`
	Hi to Lo, Lo to Hi, Hi to Hi		`MOV Rd, Rm`		Rd := Rm	Not Lo to Lo.
	Any to Any	6	`MOV Rd, Rm`		Rd := Rm	Any register to any register.
Add	Immediate 3		`ADDS Rd, Rn, #<imm>`	N Z C V	Rd := Rn + imm	imm range 0-7.
	All registers Lo		`ADDS Rd, Rn, Rm`	N Z C V	Rd := Rn + Rm	
	Hi to Lo, Lo to Hi, Hi to Hi		`ADD Rd, Rd, Rm`		Rd := Rd + Rm	Not Lo to Lo.
	Any to Any	T2	`ADD Rd, Rd, Rm`		Rd := Rd + Rm	Any register to any register.
	Immediate 8		`ADDS Rd, Rd, #<imm>`	N Z C V	Rd := Rd + imm	imm range 0-255.
	With carry		`ADCS Rd, Rd, Rm`	N Z C V	Rd := Rd + Rm + C-bit	
	Value to SP		`ADD SP, SP, #<imm>`		SP := SP + imm	imm range 0-508 (word-aligned).
	Form address from SP		`ADD Rd, SP, #<imm>`		Rd := SP + imm	imm range 0-1020 (word-aligned).
	Form address from PC		`ADR Rd, <label>`		Rd := label	label range PC to PC+1020 (word-aligned).
Subtract	Lo and Lo		`SUBS Rd, Rn, Rm`	N Z C V	Rd := Rn – Rm	
	Immediate 3		`SUBS Rd, Rn, #<imm>`	N Z C V	Rd := Rn – imm	imm range 0-7.
	Immediate 8		`SUBS Rd, Rd, #<imm>`	N Z C V	Rd := Rd – imm	imm range 0-255.
	With carry		`SBCS Rd, Rd, Rm`	N Z C V	Rd := Rd – Rm – NOT C-bit	
	Value from SP		`SUB SP, SP, #<imm>`		SP := SP – imm	imm range 0-508 (word-aligned).
	Negate		`RSBS Rd, Rn, #0`	N Z C V	Rd := – Rn	Synonym: `NEGS Rd, Rn`
Multiply	Multiply		`MULS Rd, Rm, Rd`	N Z * *	Rd := Rm * Rd	* C and V flags unpredictable in §4T, unchanged in §5T and above
Compare			`CMP Rn, Rm`	N Z C V	update APSR flags on Rn – Rm	Can be Lo to Lo, Lo to Hi, Hi to Lo, or Hi to Hi.
	Negative		`CMN Rn, Rm`	N Z C V	update APSR flags on Rn + Rm	
	Immediate		`CMP Rn, #<imm>`	N Z C V	update APSR flags on Rn – imm	imm range 0-255.
Logical	AND		`ANDS Rd, Rd, Rm`	N Z	Rd := Rd AND Rm	
	Exclusive OR		`EORS Rd, Rd, Rm`	N Z	Rd := Rd EOR Rm	
	OR		`ORRS Rd, Rd, Rm`	N Z	Rd := Rd OR Rm	
	Bit clear		`BICS Rd, Rd, Rm`	N Z	Rd := Rd AND NOT Rm	
	Move NOT		`MVNS Rd, Rd, Rm`	N Z	Rd := NOT Rm	
	Test bits		`TST Rn, Rm`	N Z	update APSR flags on Rn AND Rm	
Shift/rotate	Logical shift left		`LSLS Rd, Rm, #<shift>`	N Z C*	Rd := Rm << shift	Allowed shifts 0-31. * C flag unaffected if shift is 0.
			`LSLS Rd, Rd, Rs`	N Z C*	Rd := Rd << Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Logical shift right		`LSRS Rd, Rm, #<shift>`	N Z C	Rd := Rm >> shift	Allowed shifts 1-32.
			`LSRS Rd, Rd, Rs`	N Z C*	Rd := Rd >> Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Arithmetic shift right		`ASRS Rd, Rm, #<shift>`	N Z C	Rd := Rm ASR shift	Allowed shifts 1-32.
			`ASRS Rd, Rd, Rs`	N Z C*	Rd := Rd ASR Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Rotate right		`RORS Rd, Rd, Rs`	N Z C*	Rd := Rd ROR Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.

Instrukční sada ARM Thumb 2

Operation		§	Assembler	Action	Notes
Load	with immediate offset, word		`LDR Rd, [Rn, #<imm>]`	Rd := [Rn + imm]	imm range 0-124, multiple of 4.
	halfword		`LDRH Rd, [Rn, #<imm>]`	Rd := ZeroExtend([Rn + imm][15:0])	Clears bits 31:16. imm range 0-62, even.
	byte		`LDRB Rd, [Rn, #<imm>]`	Rd := ZeroExtend([Rn + imm][7:0])	Clears bits 31:8. imm range 0-31.
	with register offset, word		`LDR Rd, [Rn, Rm]`	Rd := [Rn + Rm]	
	halfword		`LDRH Rd, [Rn, Rm]`	Rd := ZeroExtend([Rn + Rm][15:0])	Clears bits 31:16
	signed halfword		`LDRSH Rd, [Rn, Rm]`	Rd := SignExtend([Rn + Rm][15:0])	Sets bits 31:16 to bit 15
	byte		`LDRB Rd, [Rn, Rm]`	Rd := ZeroExtend([Rn + Rm][7:0])	Clears bits 31:8
	signed byte		`LDRSB Rd, [Rn, Rm]`	Rd := SignExtend([Rn + Rm][7:0])	Sets bits 31:8 to bit 7
	PC-relative		`LDR Rd, <label>`	Rd := [label]	label range PC to PC+1020 (word-aligned).
	SP-relative		`LDR Rd, [SP, #<imm>]`	Rd := [SP + imm]	imm range 0-1020, multiple of 4.
	Multiple, not including base		`LDM Rn!, <loreglist>`	Loads list of registers (not including Rn)	Always updates base register, Increment After.
	Multiple, including base		`LDM Rn, <loreglist>`	Loads list of registers (including Rn)	Never updates base register, Increment After.
Store	with immediate offset, word		`STR Rd, [Rn, #<imm>]`	[Rn + imm] := Rd	imm range 0-124, multiple of 4.
	halfword		`STRH Rd, [Rn, #<imm>]`	[Rn + imm][15:0] := Rd[15:0]	Ignores Rd[31:16]. imm range 0-62, even.
	byte		`STRB Rd, [Rn, #<imm>]`	[Rn + imm][7:0] := Rd[7:0]	Ignores Rd[31:8]. imm range 0-31.
	with register offset, word		`STR Rd, [Rn, Rm]`	[Rn + Rm] := Rd	
	halfword		`STRH Rd, [Rn, Rm]`	[Rn + Rm][15:0] := Rd[15:0]	Ignores Rd[31:16]
	byte		`STRB Rd, [Rn, Rm]`	[Rn + Rm][7:0] := Rd[7:0]	Ignores Rd[31:8]
	SP-relative, word		`STR Rd, [SP, #<imm>]`	[SP + imm] := Rd	imm range 0-1020, multiple of 4.
	Multiple		`STM Rn!, <loreglist>`	Stores list of registers	Always updates base register, Increment After.
Push	Push		`PUSH <loreglist>`	Push registers onto full descending stack	
	Push with link		`PUSH <loreglist+LR>`	Push LR and registers onto full descending stack	
Pop	Pop		`POP <loreglist>`	Pop registers from full descending stack	
	Pop and return	4T	`POP <loreglist+PC>`	Pop registers, branch to address loaded to PC	
	Pop and return with exchange	5T	`POP <loreglist+PC>`	Pop, branch, and change to ARM state if address[0] = 0	
If-Then	If-Then	T2	`IT{pattern} {cond}`	Makes up to four following instructions conditional, according to pattern. pattern is a string of up to three letters. Each letter can be T (Then) or E (Else).	The first instruction after IT has condition cond. The following instructions have condition cond if the corresponding letter is T, or the inverse of cond if the corresponding letter is E. See Table **Condition Field**.
Branch	Conditional branch		`B{cond} <label>`	If {cond} then PC := label	label must be within – 252 to + 258 bytes of current instruction. See Table **Condition Field**.
	Compare, branch if (non) zero	T2	`CB{N}Z Rn,<label>`	If Rn {== | !=} 0 then PC := label	label must be within +4 to +130 bytes of current instruction.
	Unconditional branch		`B <label>`	PC := label	label must be within ±2KB of current instruction.
	Long branch with link		`BL <label>`	LR := address of next instruction, PC := label	This is a 32-bit instruction. label must be within ±4MB of current instruction (T2: ±16MB).
	Branch and exchange		`BX Rm`	PC := Rm AND 0xFFFFFFFE	Change to ARM state if Rm[0] = 0.
	Branch with link and exchange	5T	`BLX <label>`	LR := address of next instruction, PC := label Change to ARM	This is a 32-bit instruction. label must be within ±4MB of current instruction (T2: ±16MB).
	Branch with link and exchange	5T	`BLX Rm`	LR := address of next instruction, PC := Rm AND 0xFFFFFFFE	Change to ARM state if Rm[0] = 0.

Mikrokontrolér není „jen počítač“!

je toho v něm mnohem víc…

Proč MCU není *jen* počítač?

- **Počítač v MCU určitě je**, je ústřední a nejdůležitější komponentou.

… ale není to vše, ještě jsou tam další věci (a v tomto předmětu si budeme povídat hodně i o těch „dalších věcech“ v MCU, protože počítače už dobře znáte)

- Jsou tam **obvody pro vstupy a výstupy** (protože počítač – má-li něco řídit – potřebuje vědět, co se okolo děje a nějak to ovlivňovat).
- A ještě to nestačí – je tam spousta modulů, které **pomáhají počítači, aby se nemusel zabývat prkotinami** a stíhal to podstatné.

Počítač je sekvenční, ale svět okolo, do něhož je zrovna tento určen, je „paralelní“ – děje se hodně věcí současně! Moduly okolo jádra mu pomáhají to zvládnout.

Moduly v MCU

- Budeme se zabývat řadou dalších modulů, které se typicky vyskytují v mikrokontrolérech, ukážeme si na konkrétním příkladu MCU, jak fungují, jak se programují/nastavují a k čemu jsou dobré. Některé si zkusíte využít v laboratořích.
- Moduly v MCU typicky dělají činnosti, které je potřeba v aplikacích
  - dělat často,
  - jsou rutinní,
  - bylo by fajn je dělat „na pozadí“ (aby se procesor nemusel věnovat jen jim, ale mohl být k dispozici pro „důležitější“ věci).

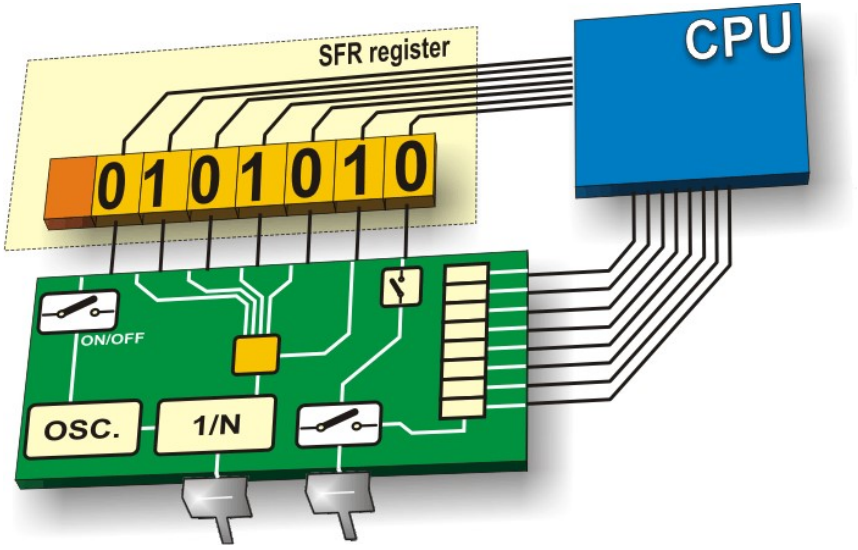
  Takové činnosti se pak „zadají“ příslušnému modulu a dál se na ně může „zapomenout“ nebo se jen zkontroluje výsledek.

Moduly v MCU

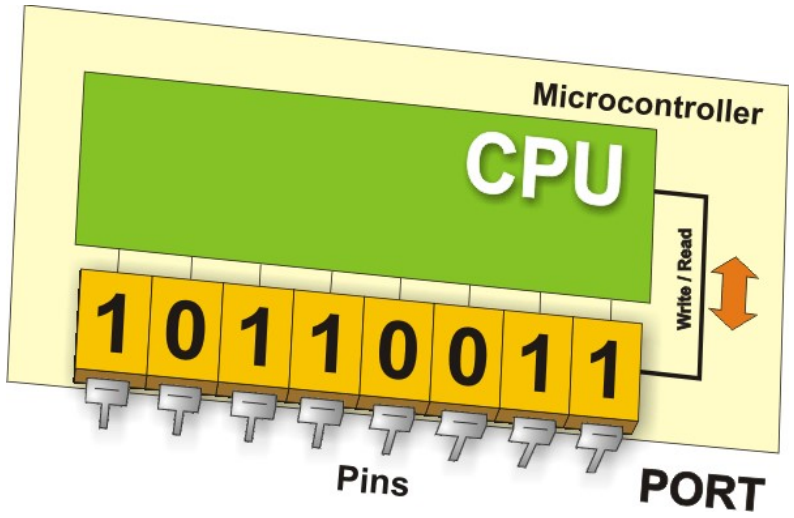
- Modul je typicky hardwarovou implementací nějaké funkce (kterou pak nemusíme programovat, ale jen si ji spustíme, když ji potřebujeme – výrobci MCU za ta desetiletí vývoje již dávno přišli na to, co „se hodí“).
  - Taková **implementace je efektivnější**, než kdyby to bylo naprogramováno jako sekvenční kód pro počítač – je to rychlejší, zabere to méně křemíku (plochy čipu), sežere to méně energie.
  - A hlavně: **běží to paralelně s počítačem**, což je blíže reálnému světu – postihnout děje reálného světa sekvenčním jednovláknovým počítačem naráží, jak sami víte, často na problémy.

Jak se moduly jeví programátorovi

- Na **definovaných adresách** jsou dostupné registry
  - do nich lze programem zapsat „co od modulu chci“
  - z nich lze přečíst momentální stav modulu či výsledek.



Příklad: jednoduchý vstup/výstup



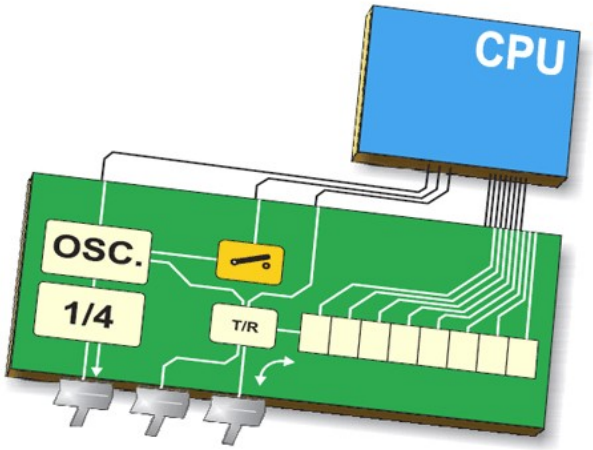
Výstup:

Zapíšu do registru 1 nebo 0 na příslušné místo ➔ na vývodu se objeví 1 nebo 0.

Vstup:

Na vývod je přivedena 1 nebo 0 ➔ hodnotu přečtu v registru na příslušném místě.

Příklad: sériová komunikace



Přijdou-li data zvenku, modul je přijme a jsou k dispozici v registru. Tam je program může přečíst. Jenže jak poznat, že data už přišla?

► musí existovat nějaký bit v nějakém dalším registru, který říká, zda máme nová data (příznak přijatých dat).

Tento příznak je třeba nejdříve otestovat a pak teprve číst data.

Nebo jinak – příchod dat způsobí přerušení a v jeho obsluze se čtou data.

Dokumentace!

- Dnešní MCU jsou složité – principy jsou vždy stejné (podobné) – ty si vysvětlíme na přednáškách a vyzkoušíme na cvičeních.
- Konkrétní věci ale musíme vždy hledat v dokumentaci.
- Typicky
  - Data Sheet … MCU jako elektronická součástka
  - Reference Manual … MCU z pohledu programátora – dokumentace k počítači na čipu
  - Application Notes … příklady použití
  - Errata … kde jsou známé chyby

Dokumentace k „našemu“ MCU

- Data Sheet: https://www.nxp.com/docs/en/data-sheet/KL05P48M48SF1.pdf
- Reference Manual: https://www.nxp.com/docs/en/reference-manual/KL05P48M48SF1RM.pdf
- App. Note příklady: https://www.nxp.com/docs/en/application-note/AN5075.pdf
- https://www.nxp.com/docs/en/application-note/AN4734.pdf

Shrnutí

- MCU **můžeme brát jako čip se softwarem definovatelným chováním**,
- je to další krok od jednoúčelově navržených elektronických obvodů.
- **Přesto to není pouze počítač** (i když počítač tam dominuje), ale obsahuje další (jednoúčelově navržené) obvody, které se typicky stejně hodí.
- **Spojuje tak výhody softwarového** řešení (flexibilita, pohodlí a snadnost návrhu a ladění) **a hardwarového řešení** (efektivní implementace, rychlost, paralelismus).
- Ale pozor, jde o kompromis, který nemusí vyhovět vždy.
- Ukazuje se však, že **pro drtivou většinu aplikací je výborný**. Zbytek se musí řešit jinak.