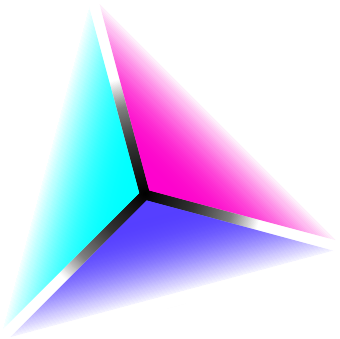
Struktura a architektura počítačů

Katedra číslicového návrhu
Fakulta informačních technologií
České vysoké učení technické

Architektura souboru instrukcí, činnost procesoru

BI-SAP, duben 2021



Obsah

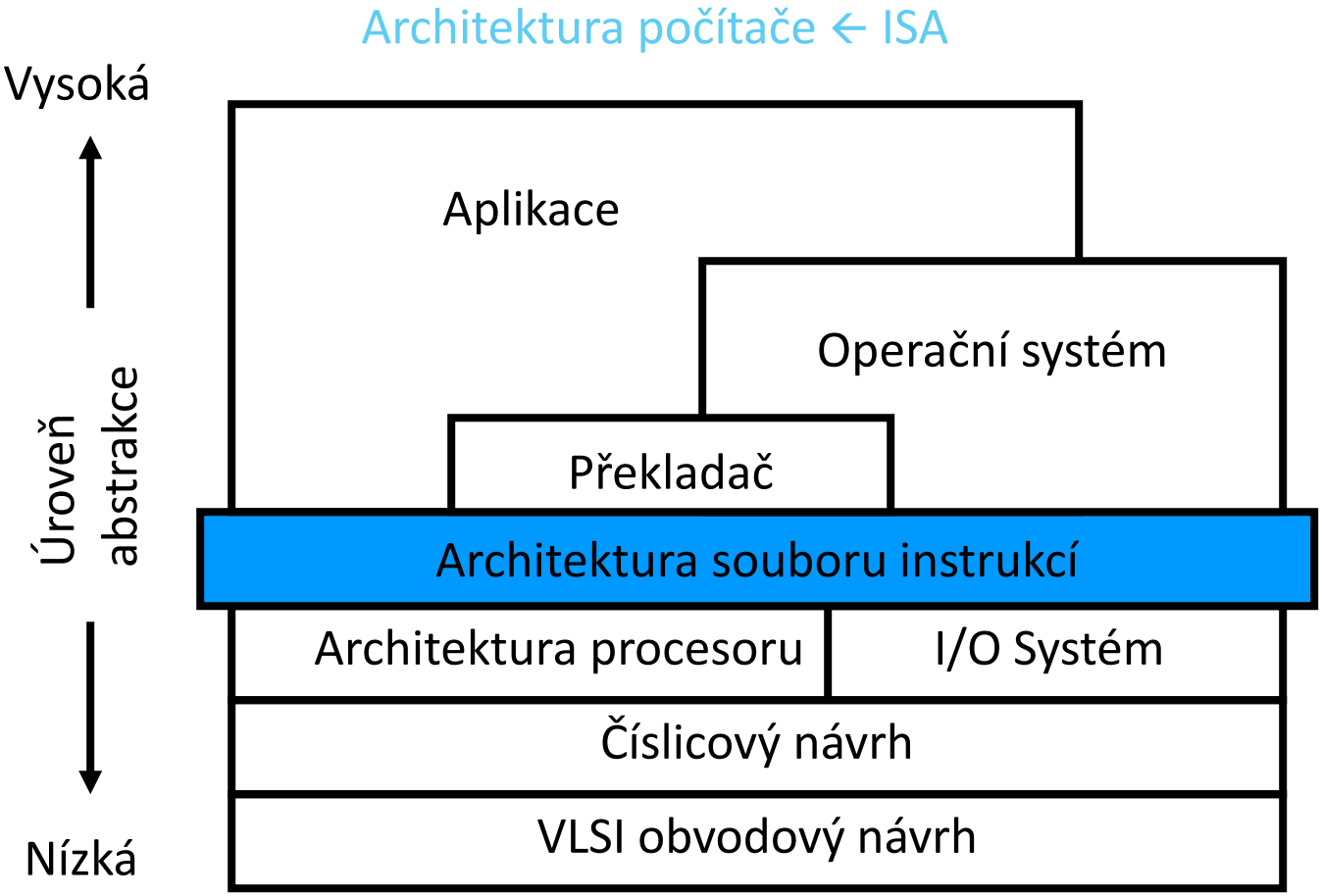
- Architektura souboru instrukcí (ISA) v kontextu architektury číslicového počítače
- Struktura instrukce, strojový kód
- Adresace operandů
- Typy architektur souboru instrukcí:
  - střadačově orientovaná
  - zásobníkově orientovaná
  - GPR
- Procesor a jeho činnost (zpracování instrukcí)
- Jazyk symbolických instrukcí (JSI, asembler)

Poznámka: AVR ISA je v přednáškách použita jen jako příklady, půjde o ISA obecně

některé podklady jsou převzaty z předmětů BI-APS a BI-JPO

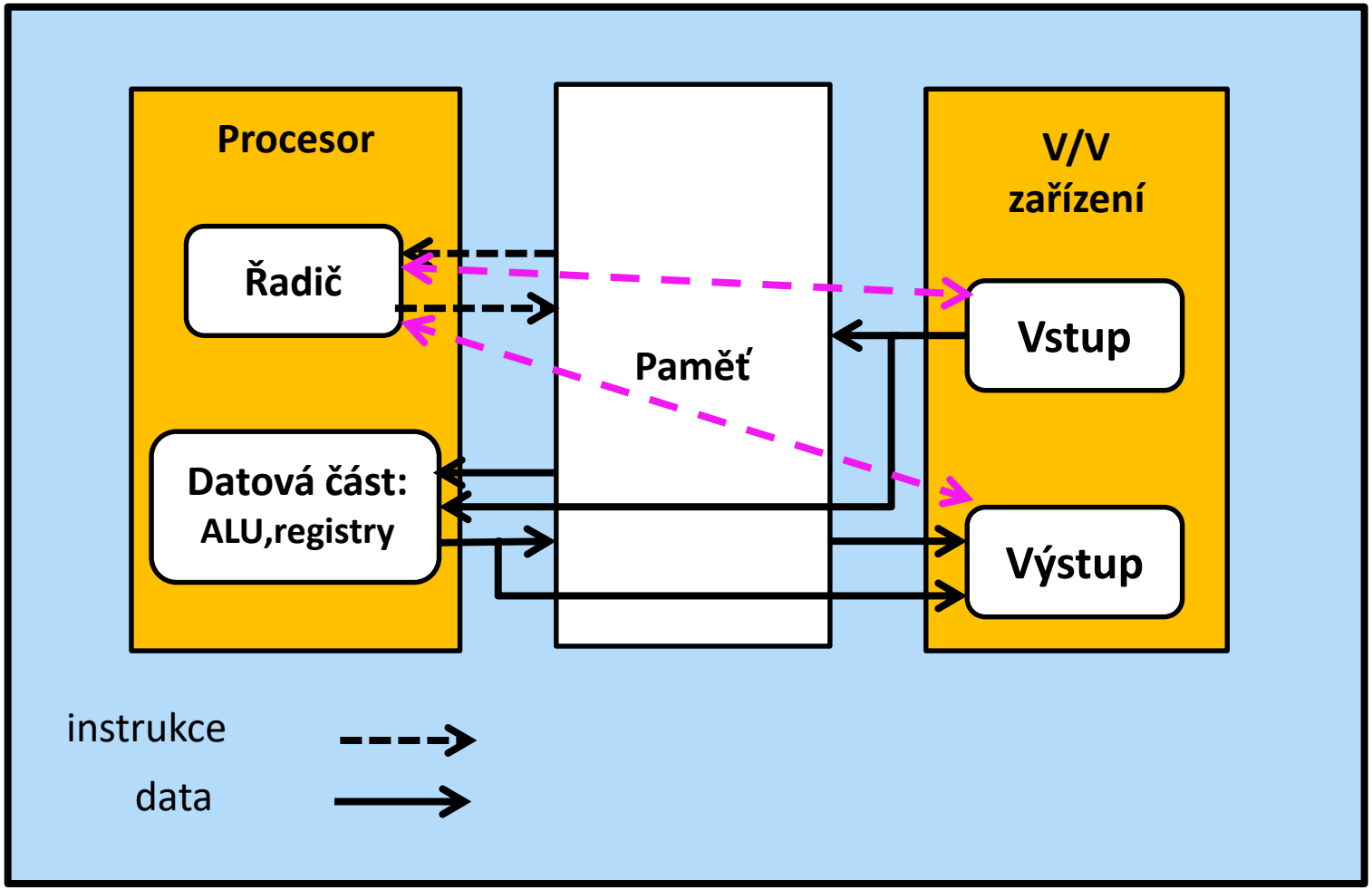
Architektura souboru instrukcí

- ISA = Instruction Set Architecture

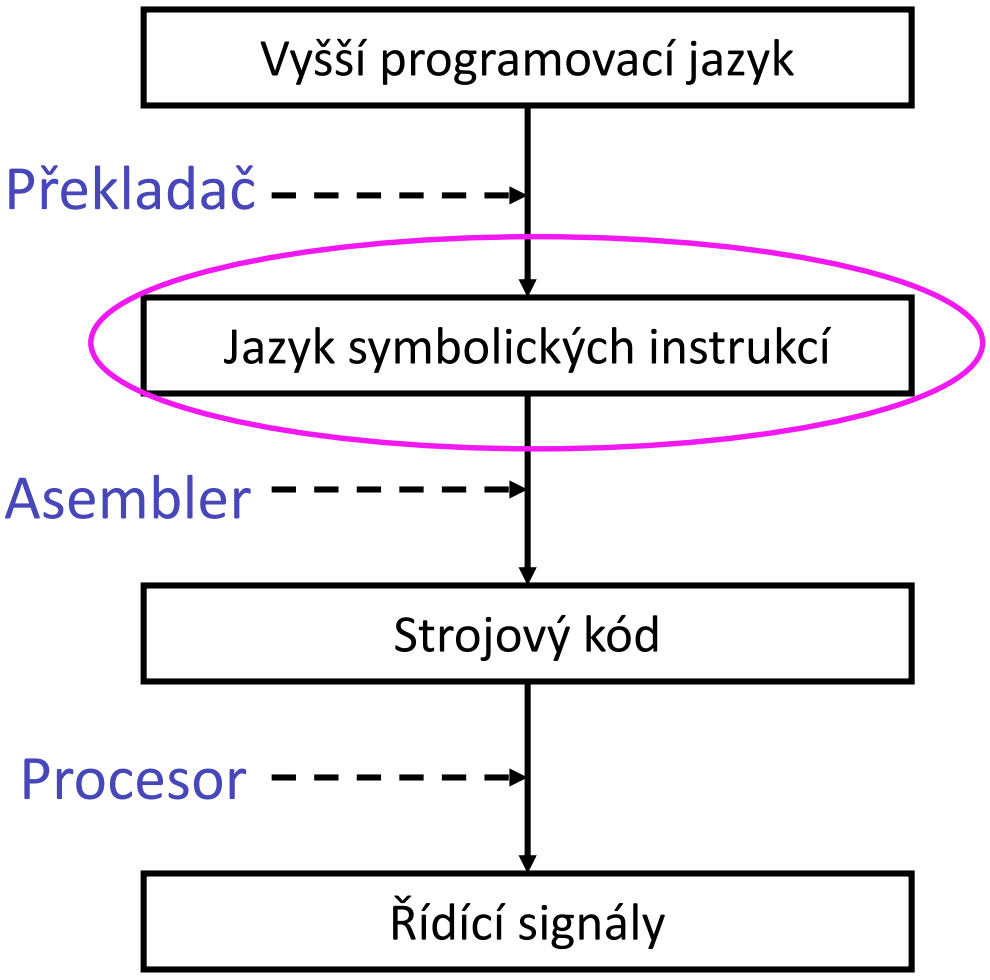


ISA vs Von Neumann

- Co dělá řadič? Řídí činnost počítače.
- Co dělá počítač? Zpracovává programy, tedy instrukce. Jak?



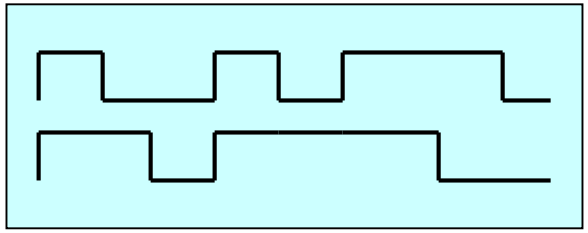
z úvodní přednášky:



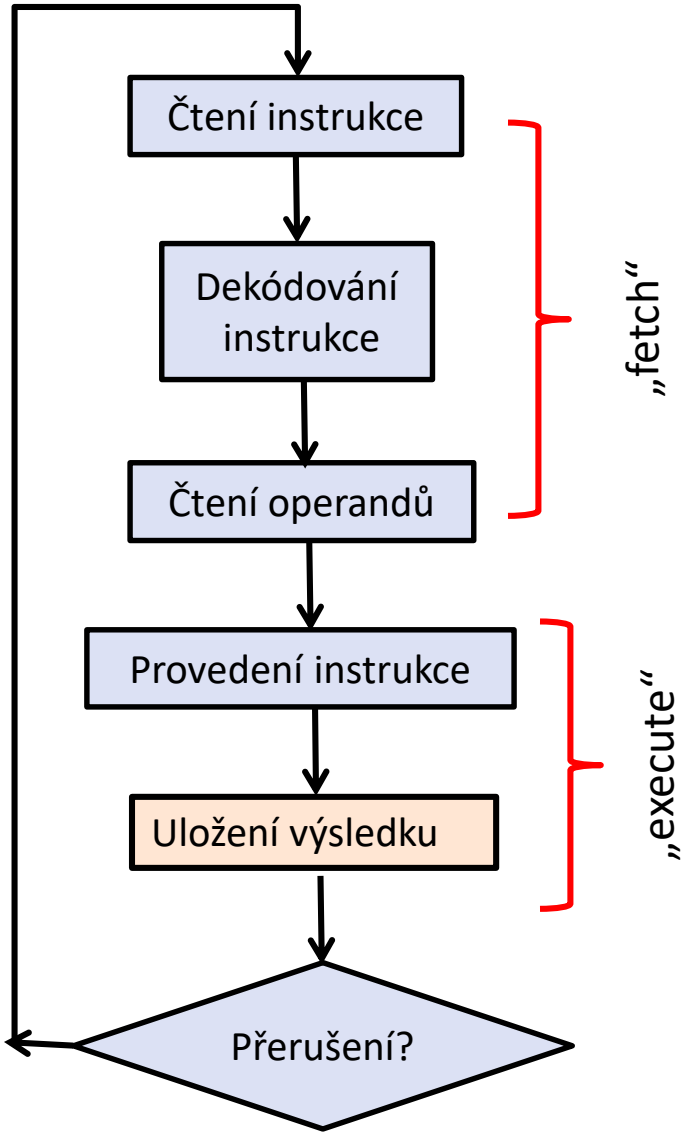

```
a = b+c
If (a > MAX) a = MAX
for (i==0; i<a; i++)
```

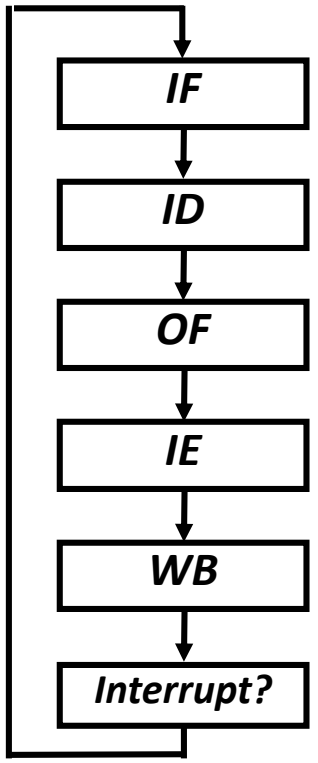
```
mov reg1, konst[0]
mov reg2, konst[2]
add reg1, reg2
jc lab
```

```
0000 1111 0101 0111
1011 0001 1110 0011
1100 1000 1001 0110
```



Instrukční cyklus





1. Instruction Fetch ... IF: načtení instrukce
2. Instruction Decode ... ID: dekódování instrukce
3. Operand Fetch ... OF: načtení operandu(ů)
4. Instruction Execution ... IE: vykonání instrukce
5. Write Back ... WB: zapsání výsledku
   (také Result Store)

Struktura instrukce, strojový kód

Co musí instrukce obsahovat:

instrukce = příkaz, zakódovaný jako číslo

- co se má provést
- s čím se to má provést (operandy)
- kam se má uložit výsledek
- kde se má pokračovat

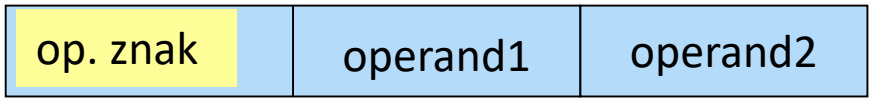
Tyto informace jsou obsaženy v instrukci ... tzn. **explicitně** např. počítač SAPO, tzv. 5 **adresový**

nebo jsou v instrukci jen částečně, a jsou dány architekturou počítače ... tzn. **implicitně** (von Neumannova architektura) → pokračuje se následující instrukcí programu, která je na další adrese .... kde je v procesoru uložena tato adresa? Speciální registr PC (Program Counter)

Příklady strojových instrukcí

instrukce: *operační znak, OZ, opcode* | operand(y)

3 adresová instrukce



2 adresová instrukce

výsledek se ukládá na místo prvního operandu, zavedení operace **přesun**

$$\{x\} - \{y\} \rightarrow z \quad = \quad \begin{cases} \{x\} \rightarrow z \\ \{z\} - \{y\} \rightarrow z \end{cases}$$

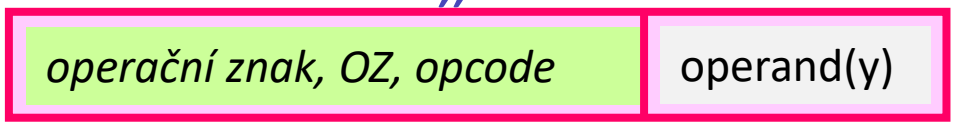
1 adresová instrukce

zavedení „pracovního“ registru – STŘADAČ, ACCUMULATOR

$$\{x\} - \{y\} \rightarrow z \quad = \quad \begin{cases} \{x\} \rightarrow S \\ \{S\} - \{y\} \rightarrow S \\ \{S\} \rightarrow z \end{cases}$$

Více „střadačů”

instrukce:



více registrů k „obecnému“ použití

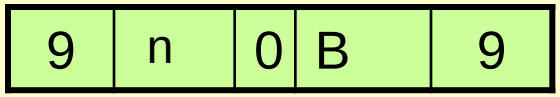
Příklad:

Motorola 68000

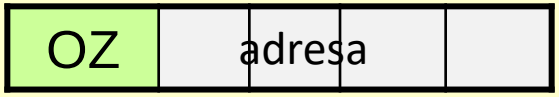
Datové registry **D_0, D_1, ..., D_7** á 32b

Odčítání 32b: **D_n – paměť → D_n**

OZ:



instrukce:



<D3> - <18FF20 ÷ 18FF23> → D3

96B90018FF20

- zápis ve strojovém kódu je nepřehledný a špatně se programuje →
- vyšší programovací jazyk VPJ: Pascal, Java, C
- jazyk symbolických instrukcí JSI .... operační znak i adresy (operandy) jsou zapsány symbolicky .....asembler

Příklad kódování instrukce v AVR

Symbolický zápis instrukce

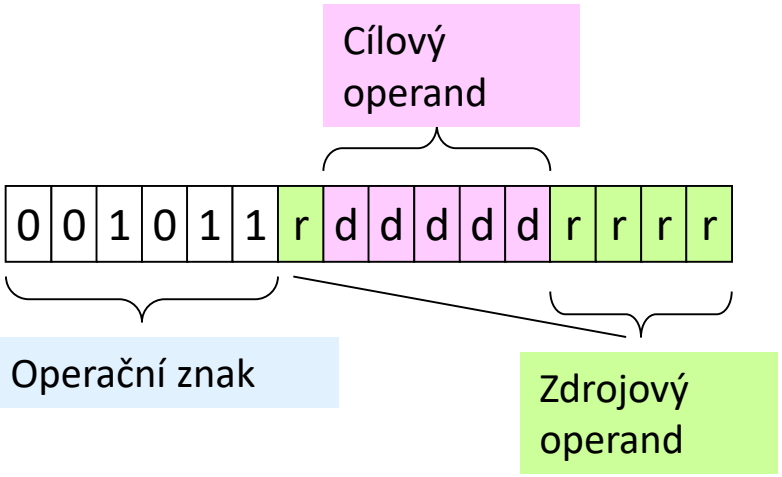
```
mov r1, r2
```

Zdrojový operand

Cílový operand

Jméno instrukce

Strojový kód instrukce `mov`, kterou načítá a dekóduje mikropočítač (16 b)



Příklad

```
mov r1, r2          : 0x2C12
```

Příklad programu

- v symbolickém vyjádření a ve strojovém kódu:

Tuto transformaci provádí program zvaný asembler

Adresa v paměti programu

Symbolické vyjádření programu

```
ldi     r17, 0x55
ldi     r18, 0x88
mov     r16, r17
mov     r17, r18
mov     r18, r16
jmp     0
```



```
0000: E515
0001: E828
0002: 2F01
0003: 2F12
0004: 2F20
0005: 940C
0006: 0000
```

Obsah paměti programu

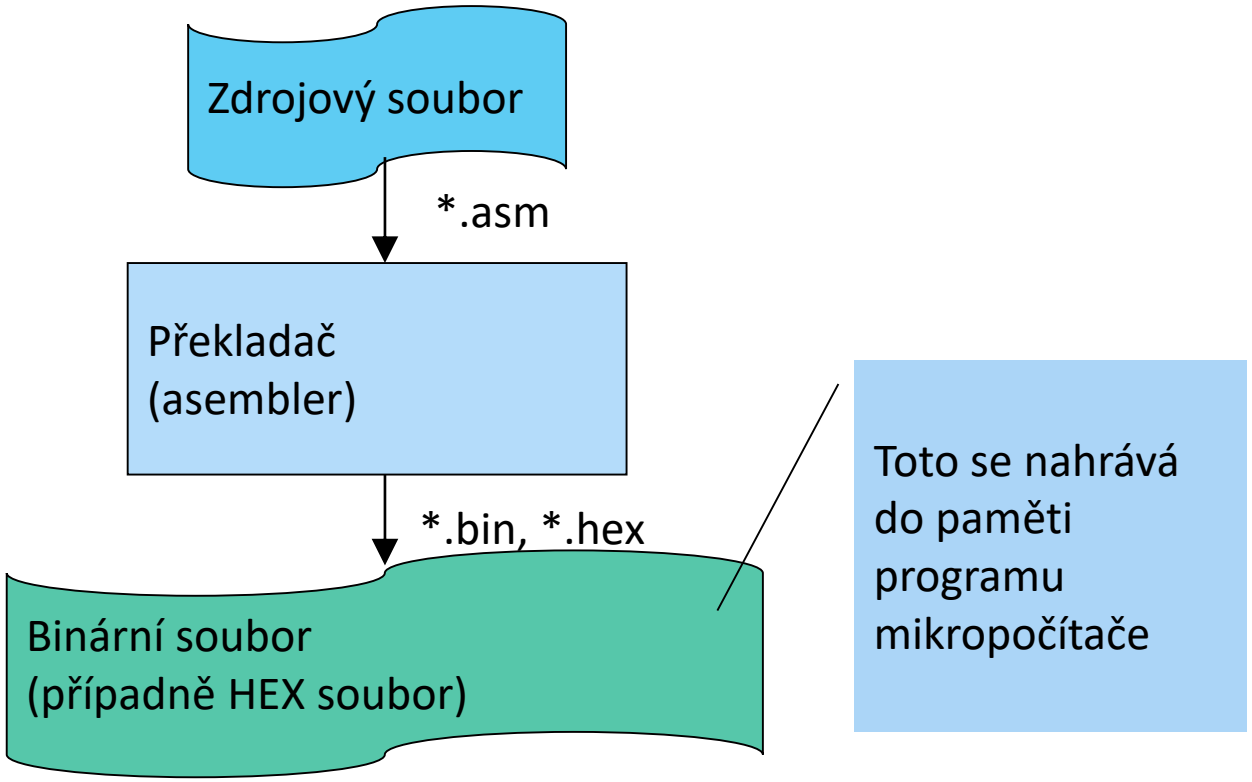
Toto je mikropočítačem načítáno a dekódováno jako instrukce

Převod mezi symbolickým vyjádřením a strojovým kódem zajišťuje překladač (asembler)

Toto píše programátor

Poznámka: mlčky předpokládáme, že strojový kód umísťujeme od adresy 0x0000.

Technologie tvorby programu v asembleru



Architektura souboru instrukcí, ISA

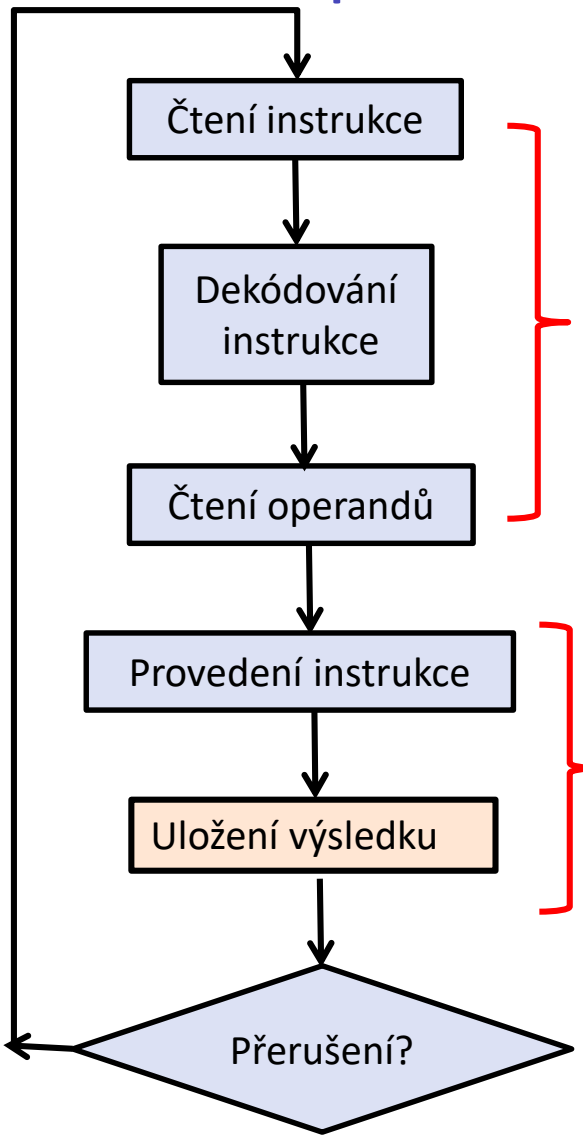
Co je třeba určit:

- Typy a formáty instrukcí, instrukční soubor (jaké instrukce?)
- Datové typy, kódovaní a reprezentace, způsob uložení dat v paměti
- Módy adresování paměti a přístup do paměti dat a instrukcí
- Mimořádné stavy

Výhody:

- Abstrakce (výhoda – různé implementace stejné architektury)
- Definice rozhraní mezi nízko-úrovňovým SW a HW
- Standardizuje instrukce, bitové vzory strojového jazyka

Zpracování instrukcí, instrukční cyklus



Formát a kódování instrukcí

– Jak se instrukce dekóduje?

Umístění operandů a výsledku

– Kolik explicitních operandů je v instrukci pro ALU?

– Jak jsou operandy umístěny v paměti nebo jinde?

– Který operand může být v paměti?

Typy dat a velikosti operandů

Operace v ISA

– Které jsou podporovány ?

Výběr další instrukce

– Skoky, větvení programu, volání podprogramů

– *„fetch-decode-execute“ je implicitní!*

Adresace operandů

tzn. *jak je určen operand v instrukci*

Přímá adresace

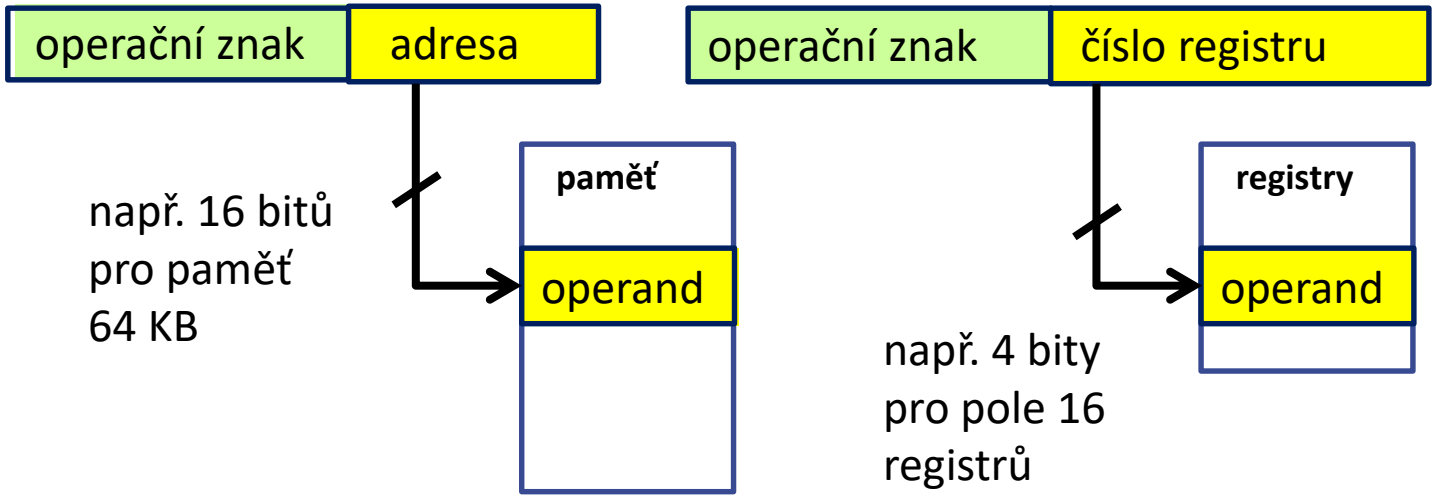
implicitní

...v instrukci je jen operační znak

bezprostřední
- immediate

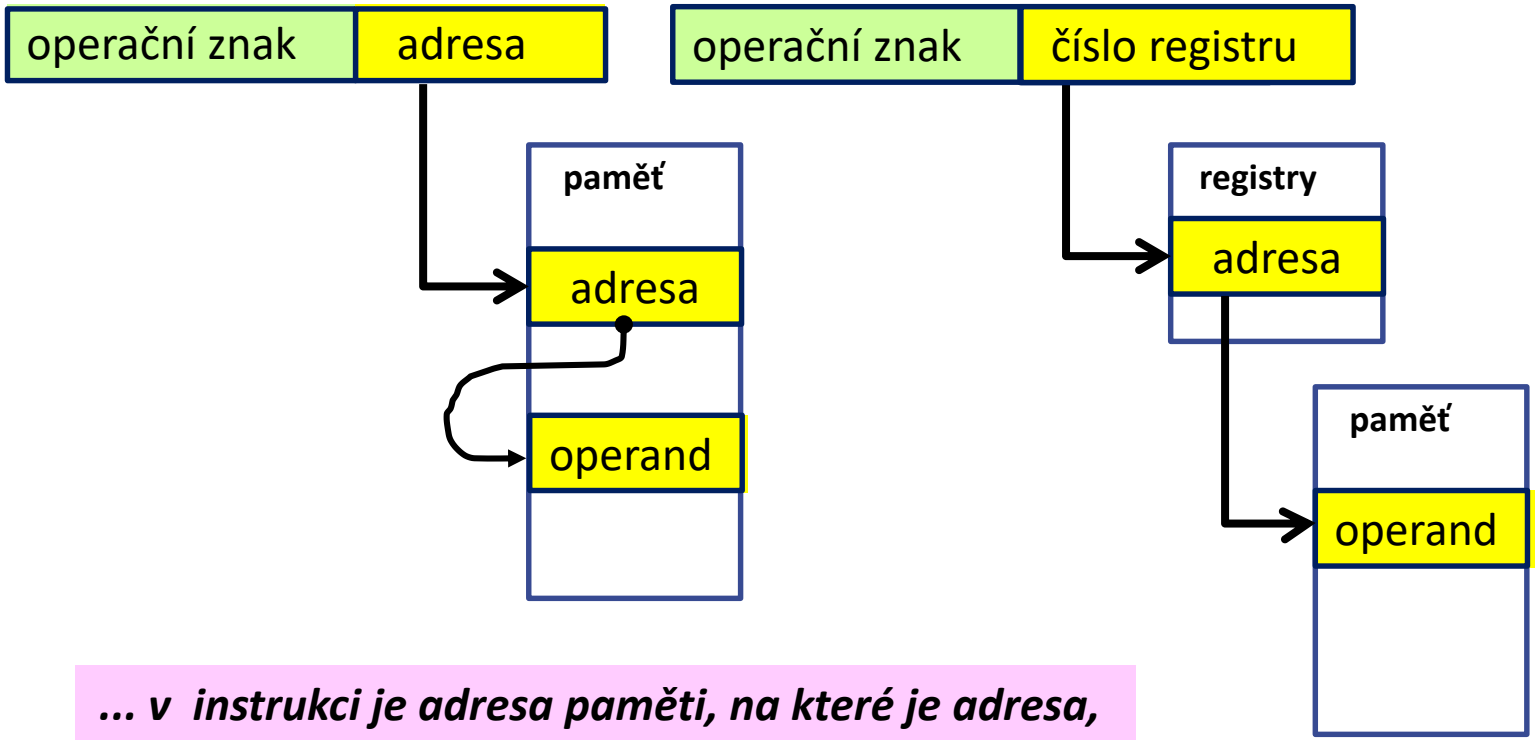
...v instrukci je konstanta = přímý operand

přímá .. direct



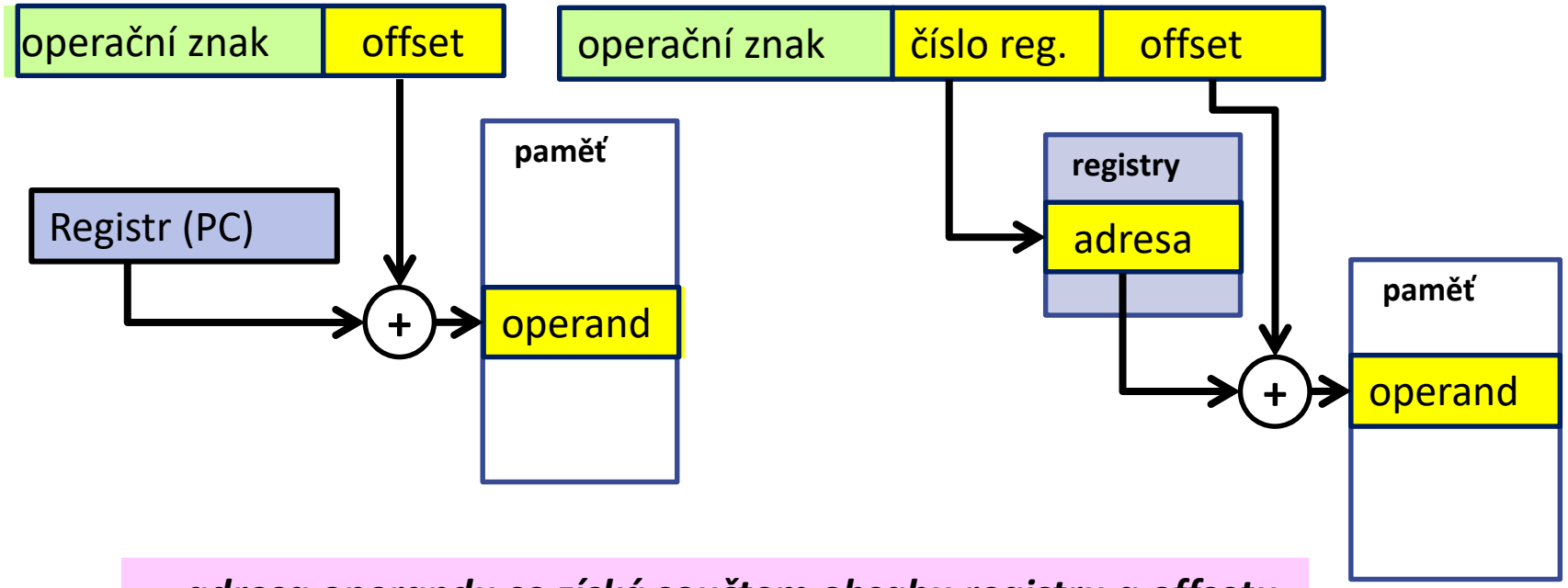
...v instrukci je adresa paměti na které je operand nebo číslo registru, ve kterém je operand

Nepřímá adresace



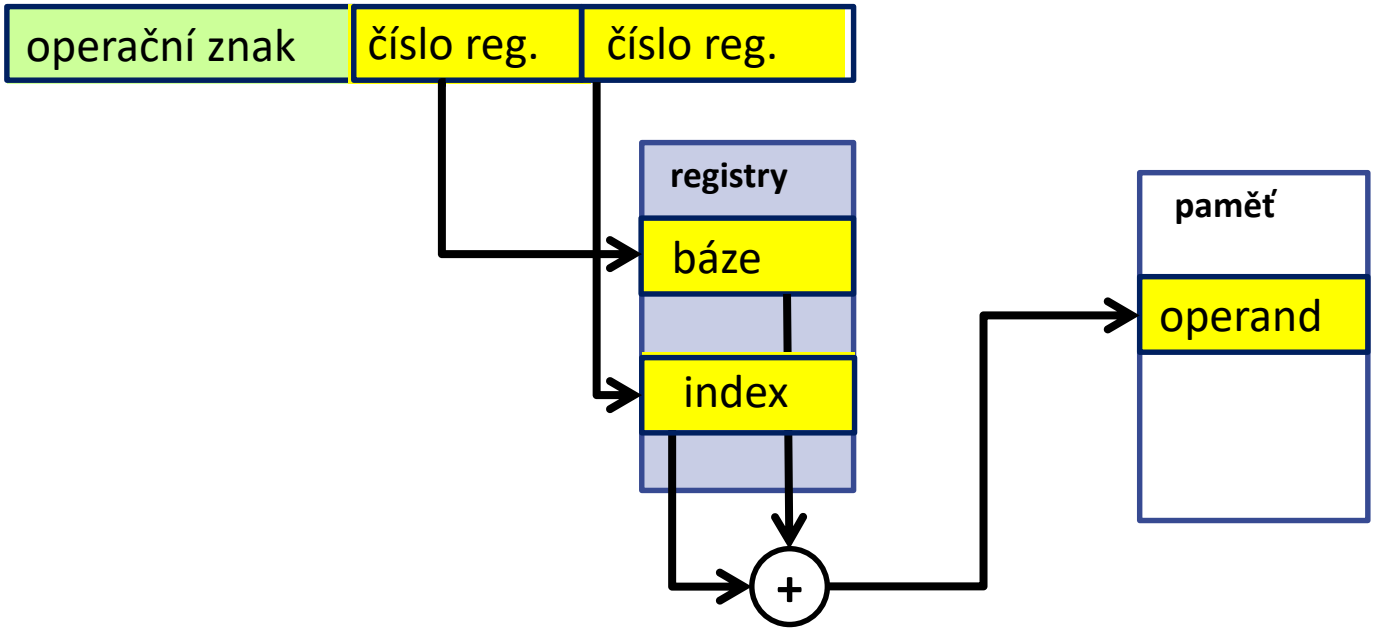
... v instrukci je adresa paměti, na které je adresa, na které je operand
nebo číslo registru, ve kterém je adresa paměti, na které je operand

Relativní adresace operandů



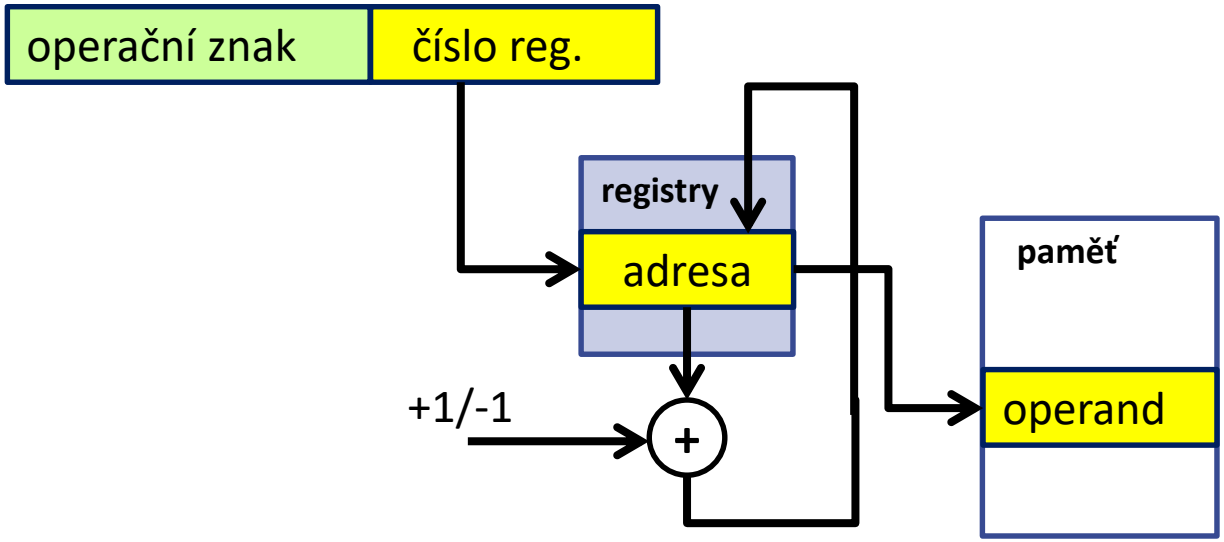
... adresa operandu se získá součtem obsahu registru a offsetu.
Registr může být buď implicitní (PC) nebo je určen v instrukci

Indexová adresace operandů



... adresa operandu se získá součtem obsahu dvou registrů (báze a indexu). Např. je to výhodné při práci s maticemi ...

Autoinkrementace a autodekrementace



... operand se získá na adrese, která obsahem registru a je navíc inkrementovaná nebo dekrementovaná

Např. SP (ukazatel zásobníku) ukazuje na vrchol tj. na první volnou pozici v zásobníku, kam je možné ukládat data a „hodí se", aby se adresa v tomto registru se zápisem/čtením zároveň dekrementovala/inkrementovala.

ISA: Základní třídy

Rozdělení podle toho, kde jsou operandy, se kterými instrukce pracuje:

Střadačově (akumulátorově) orientovaná ISA (jeden registr = střadač):

1 operand	ADD A	acc ← acc + mem[A]
	ADD (A + IX)	acc ← acc + mem[A + IX] IX je **indexovací registr**

Zásobníkově orientovaná ISA (registry uspořádané do zásobníku)

0 operandů	ADD	stack(top-1) ← stack(top) + stack(top-1)
		top--

ISA orientovaná na registry pro všeobecné použití (více registrů)
(GPR = General Purpose Registers)**:**

2 operandy	ADD A B	EA(A) ← EA(A) + EA(B)
3 operandy	ADD A B C	EA(A) ← EA(B) + EA(C)

EA … Efektivní adresa (určuje registr, nebo operand v paměti)

Střadačově orientovaná ISA

- s absolutní adresací

nejstarší ISA (1949-60) – vyvinula se z kalkulaček

LOAD	A	acc ← **mem[A]**
STORE	A	**mem[A]** ← acc
ADD	A	acc ← acc + **mem[A]**
SUB	A	acc ← acc - **mem[A]**
…		
SHIFT LEFT		acc ← 2 x acc
SHIFT RIGHT		acc ← acc / 2
JUMP A		PC ← A
JGE	A	if (0≤acc) then PC ← A
LOAD ADDR	X	načtení adresy operandu X do acc
STORE ADDR	X	uložení adresy operandu X z acc

Typicky méně než 24 instrukcí! Hardware byl velmi drahý.

Střadačově orientovaná ISA dnes

- Z indexovacích registrů se vyvinuly **speciální registry pro nepřímou adresaci**, zvláštním typem je **stack pointer SP** (ukazatel na vrchol zásobníku).
- Procesory také zahrnují **pracovní registry** (tzv. ***zápisníková paměť***). Toto pole registrů snižuje četnost přístupů do paměti.
- Implicitním operandem ALU je **vždy střadač** (druhý operand může být v registrech nebo v paměti).

Použita v prvních mikroprocesorech: 4004, 8008, 8080, 6502,…

Dnes použita v některých mikrokontrolérech: 8051, 68HC11, 68HC05, …

ISA X86 .... „ISA s více střadači…“
=> s nástupem i386 upravena na GPR.

Střadačově orientovaná ISA - shrnutí

Výhody :

- jednoduchý HW
- minimální vnitřní stav procesoru ⇒ rychlé přepínání kontextu
- krátké instrukce (záleží na typu druhého operandu)
- jednoduché dekódování instrukcí

Nevýhody :

- častá komunikace s pamětí (dnes problém)
- omezený paralelismus mezi instrukcemi

Tento typ ISA byl populární v 50. a 70. letech -
hardware byl drahý, paměť byla rychlejší než CPU.

Zásobníkově orientovaná ISA

Využití **„hardwarového zásobníku“** při vykonávání programu:

- Vyhodnocení výrazů
- Vnořená volání podprogramů
  - předávání návratové adresy a parametrů
  - lokální proměnné

Příklad: Burroughs B5000, 1961

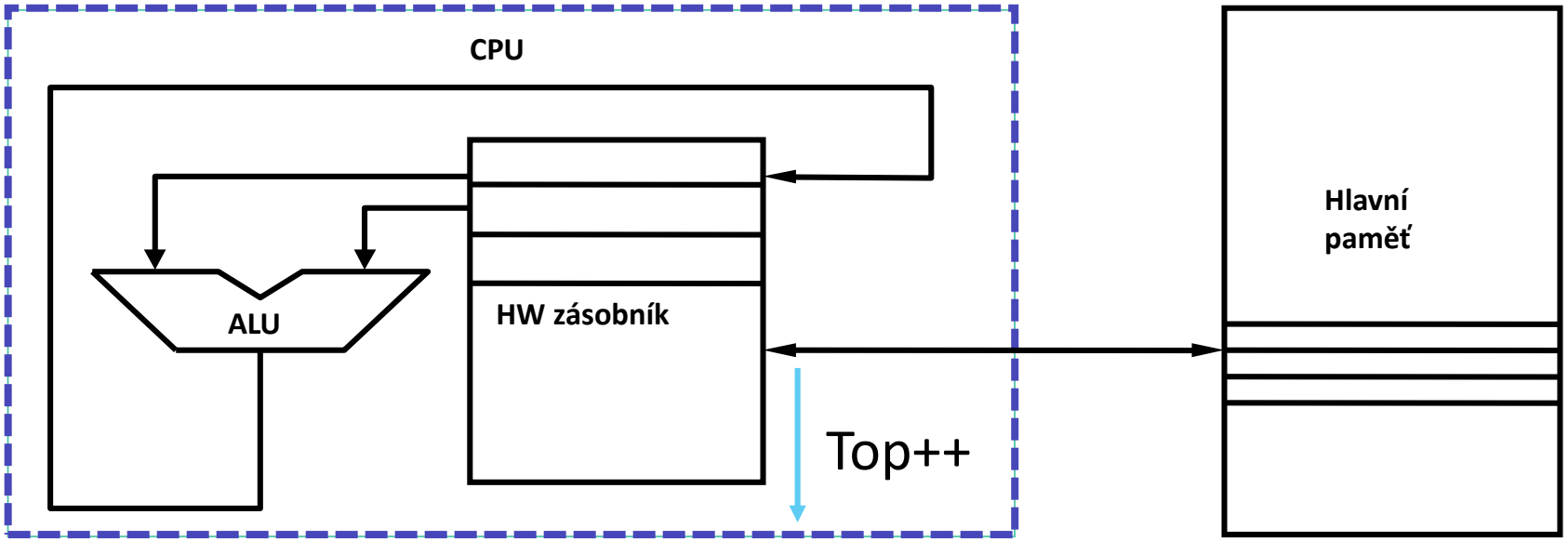
- počítač navržený k podpoře jazyka **ALGOL 60 a COBOL.**
- vyhodnocení výrazů podporoval **hardwarový zásobník**
- **https://www.cs.uaf.edu/2010/fall/cs441/proj1/b5000/**

Zásobníkově orientovaná ISA

Hardwarový zásobník = sada registrů s ukazatelem na vrchol (uvnitř CPU)

Instrukce :

PUSH A	Stack[++Top] ← mem[A]
POP A	mem[A] ⟵ Stack[Top--]
ADD,SUB, …	Stack[Top-1] ⟵ Stack[Top] **OP** Stack[Top-1]
	Top--



HW zásobník = registry uspořádané do struktury zásobníku, které jsou v procesoru, ne v hlavní paměti!

Zásobníkově orientované ISA

většinou vyhynuly před rokem 1980

Výhody :

- jednoduchá a efektivní adresace operandů
- krátké instrukce
- vysoká hustota kódu (krátké programy)
- jednoduché dekódování instrukcí
- neoptimalizující překladač lze napsat snadno

Nevýhody:

- nelze náhodně přistupovat k lokálním datům
- zásobník je sekvenční (omezuje paralelismus)
- přístupy do paměti je těžké minimalizovat

Zásobníkově orientované ISA po roce 1980

Inmos Transputers (1985 – 1996) http://www.transputer.net/tn/06/tn06.html#x1-160006

- navrženy k podpoře efektivního paralelního programování pomocí paralelního programovacího jazyka **Occam**
- **IMS T800** byl v druhé polovině 80. let nejrychlejší 32-bitový CPU
- zásobníkově orientovaná ISA zjednodušila implementaci
- podpora pro rychlé přepínání kontextu

Forth machines

- Forth je zásobníkově orientovaný jazyk, minimalistická implementace
- používá se v řídících a kybernetických aplikacích (i IoT)
- několik výrobců (Rockwell, Patriot Scientific)

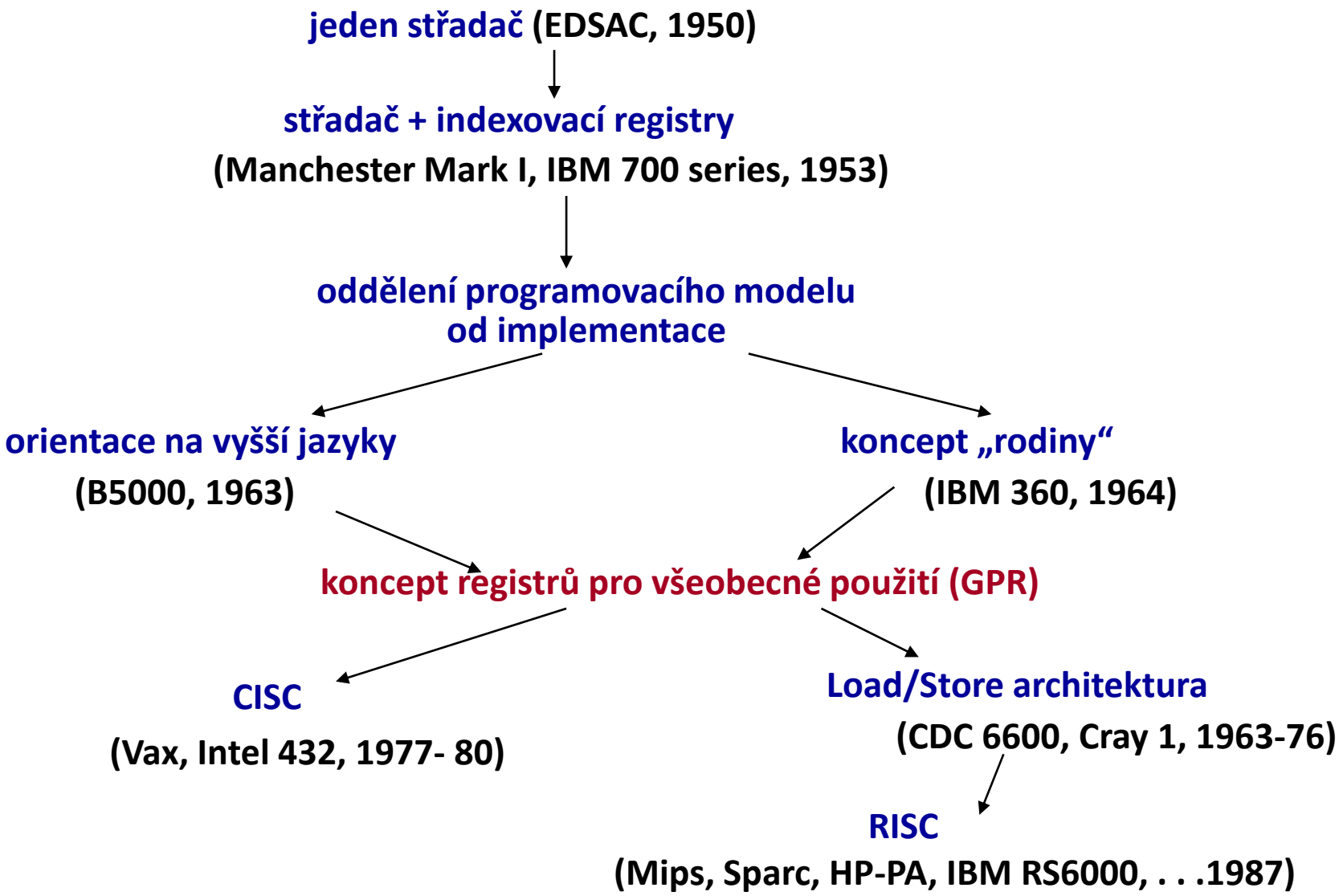
Intel x87 FPU …

- nepříliš dobře navržený zásobník pro vyhodnocování FP výrazů
- překonán architekturou SSE2 FP v Pentiu 4

Java Virtual Machine, .NET

- navržen pro **SW emulaci** (podobně jako PostScript)
- Sun PicoJava a další HW implementace

Vývoj ISA


jeden střadač (EDSAC, 1950)
střadač + indexovací registry
(Manchester Mark I, IBM 700 series, 1953)
oddělení programovacího modelu
od implementace
orientace na vyšší jazyky
(B5000, 1963)
koncept „rodiny“
(IBM 360, 1964)
koncept registrů pro všeobecné použití (GPR)
CISC
(Vax, Intel 432, 1977- 80)
Load/Store architektura
(CDC 6600, Cray 1, 1963-76)
RISC
(Mips, Sparc, HP-PA, IBM RS6000, . . .1987)

GPR ISA, výhody

... dnes převládá

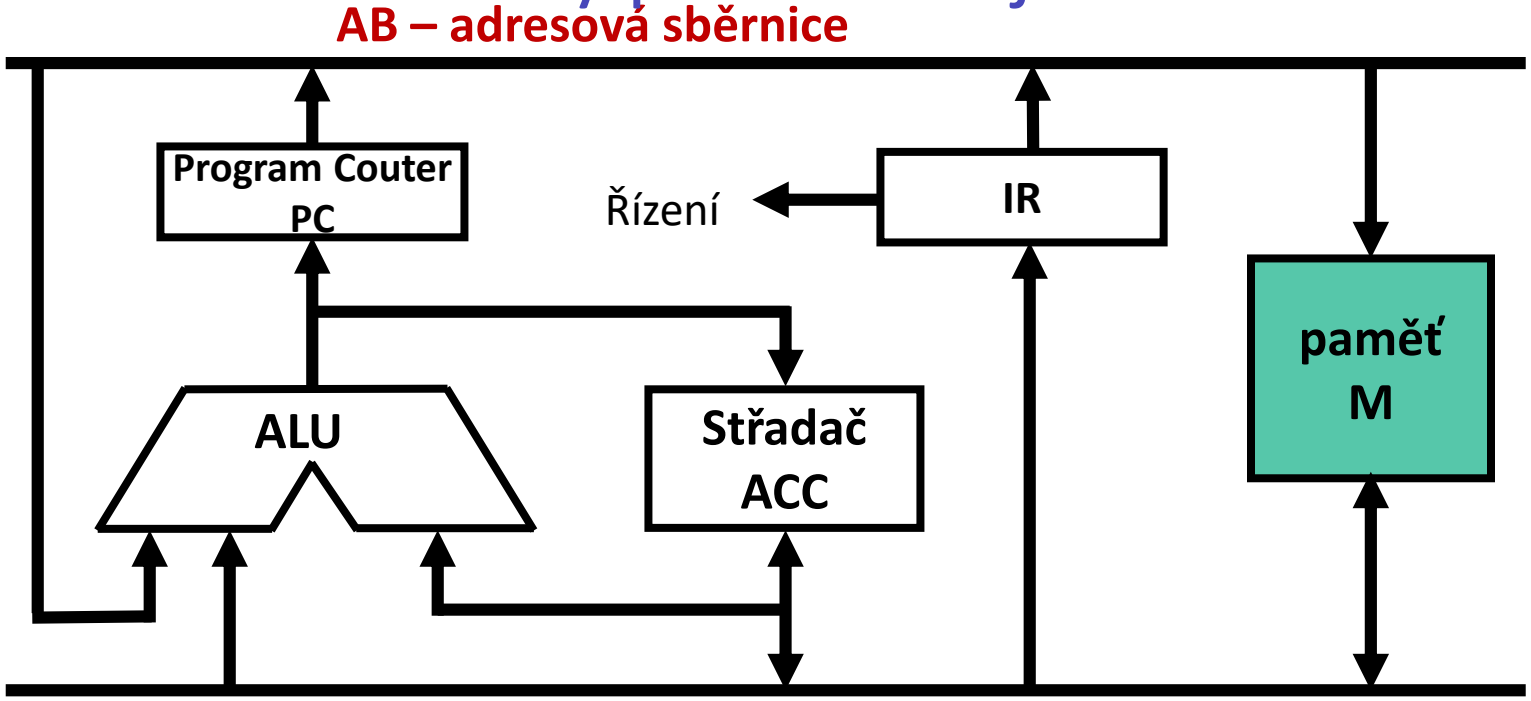
Po roce 1975 používají všechny nové procesory nějakou podobu GPR (registry pro všeobecné použití .. ***general purpose registers***)

- Registry jsou rychlejší než paměť (včetně cache !!)
- K registrům lze přistupovat náhodně (naopak ... zásobník je přísně sekvenční)
- Registry mohou obsahovat mezivýsledky a lokální proměnné
- Méně častý přístup do paměti ⟶ potenciální urychlování

Nevýhody GPR ISA

- omezený počet registrů
- složitější překladač (optimalizace použití registrů)
- přepnutí kontextu trvá déle
- registry nemohou obsahovat složitější datové struktury (records …)
- k objektům v registrech nelze přistupovat přes ukazatele (omezuje alokaci registrů)

Jednoduchý procesor a jeho činnost



M – paměť

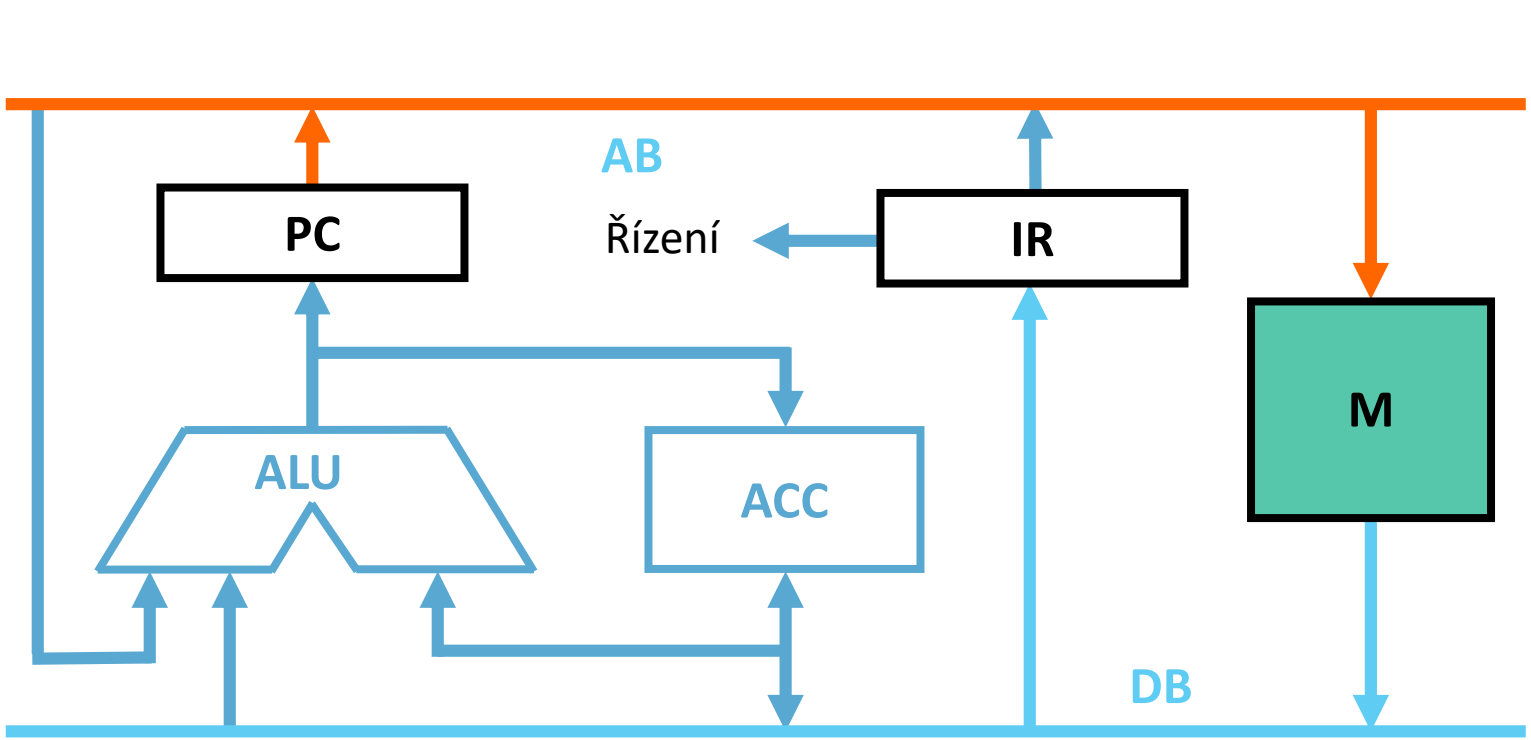
PC – **Program Counter**, programový čítač, obsahuje adresu instrukce, která bude vykonána v dalším cyklu

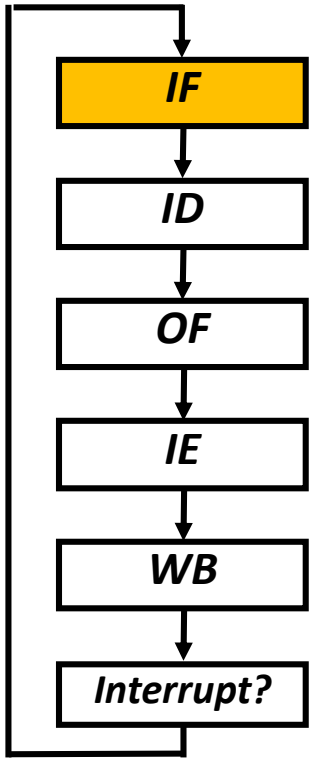
ACC – **Střadač** (accumulator), obsahuje zpracovávaná data

ALU – **Arithmetic Logic Unit**, aritmeticko-logická jednotka, vykonává operace s daty

IR – **Instruction Register**, registr instrukce, obsahuje kód aktuálně prováděné instrukce – *pro programátora je skrytý*

Čtení instrukce



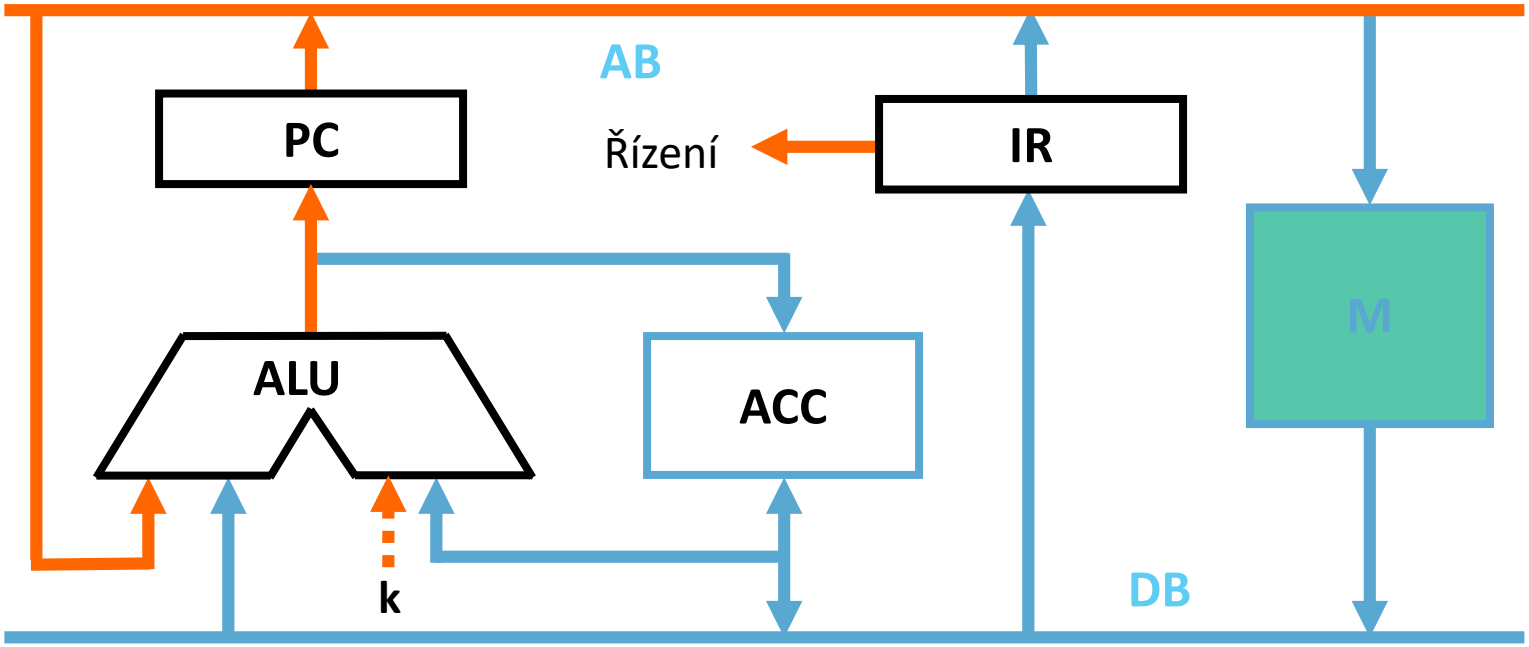


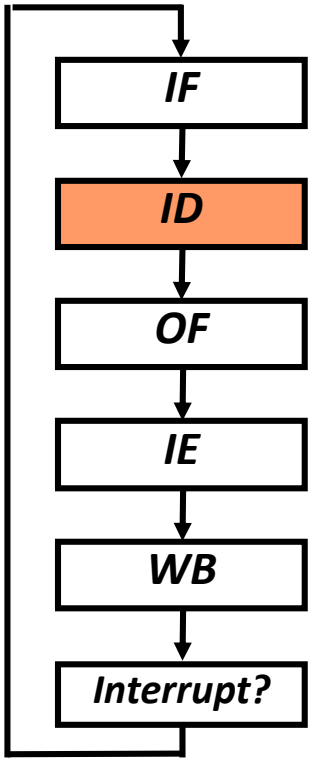
Procesor – pošle adresu instrukce z PC na AB

M – vystaví obsah paměťového místa instrukce z adresy v PC na DB

IR – kód instrukce z DB je zapsán do IR

Dekódování instrukce

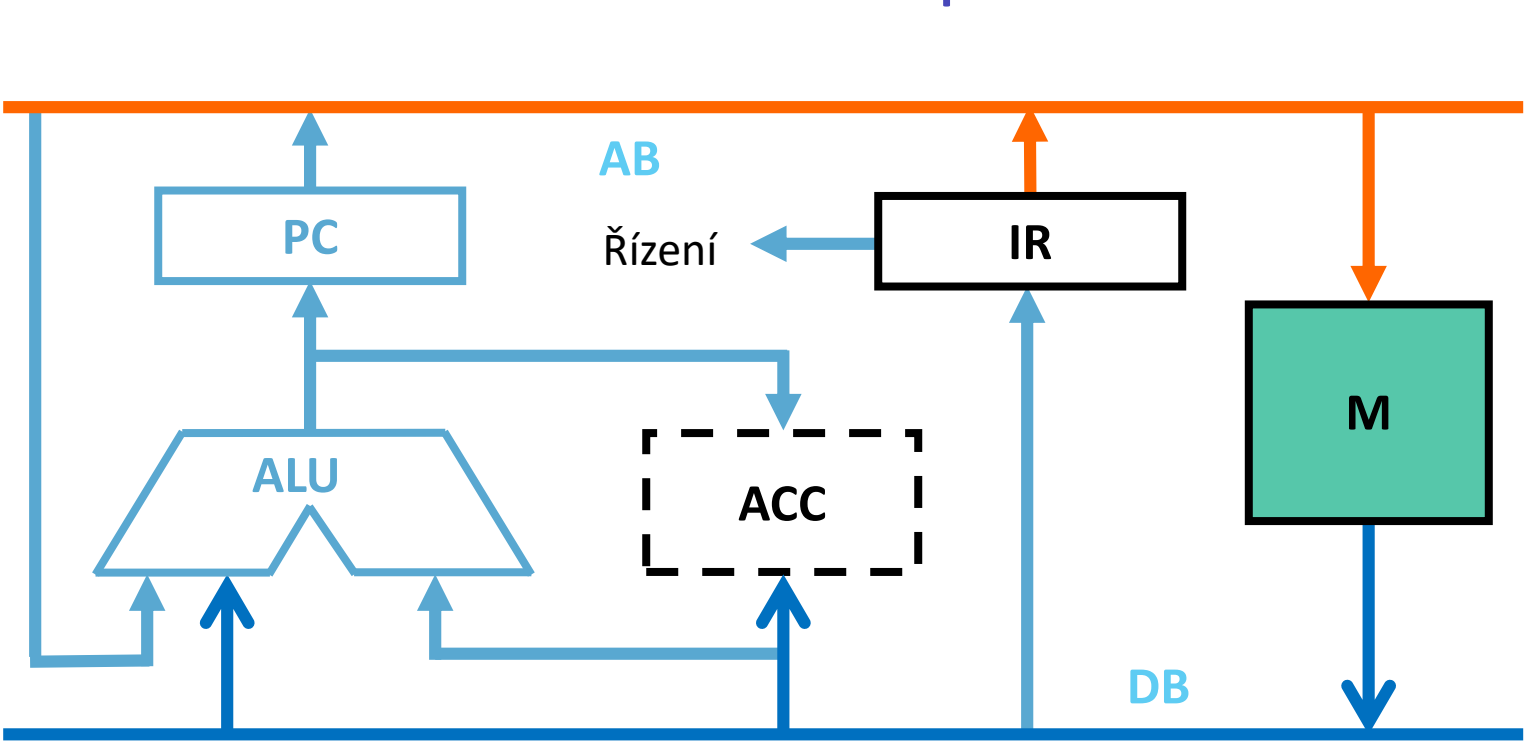


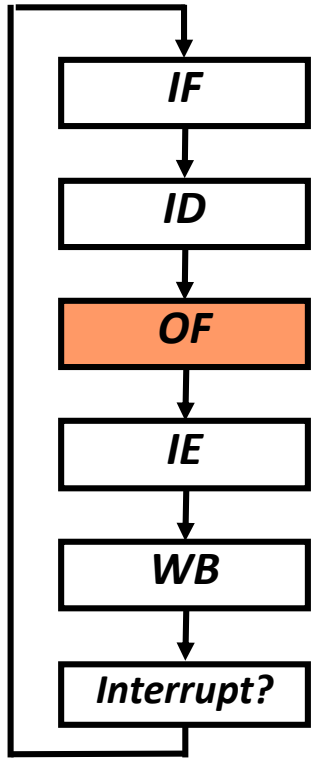


IR – instrukční kód z IR je dekódován interní logikou (řadičem) a současně jsou generovány řídící signály pro ALU a další interní obvody procesoru.

PC – programový čítač vystaví hodnotu na AB, ALU zvětší tuto hodnotu o ***k*** a zapíše nazpět do PC (když pc je původní obsah ***PC ⇒ pc ← pc+k***), hodnota k je určena dekódováním instrukce, sekvenční zpracování ⇒k=1, skok ⇒ k nějaké celé číslo s omezeného intervalu.

Čtení operandů

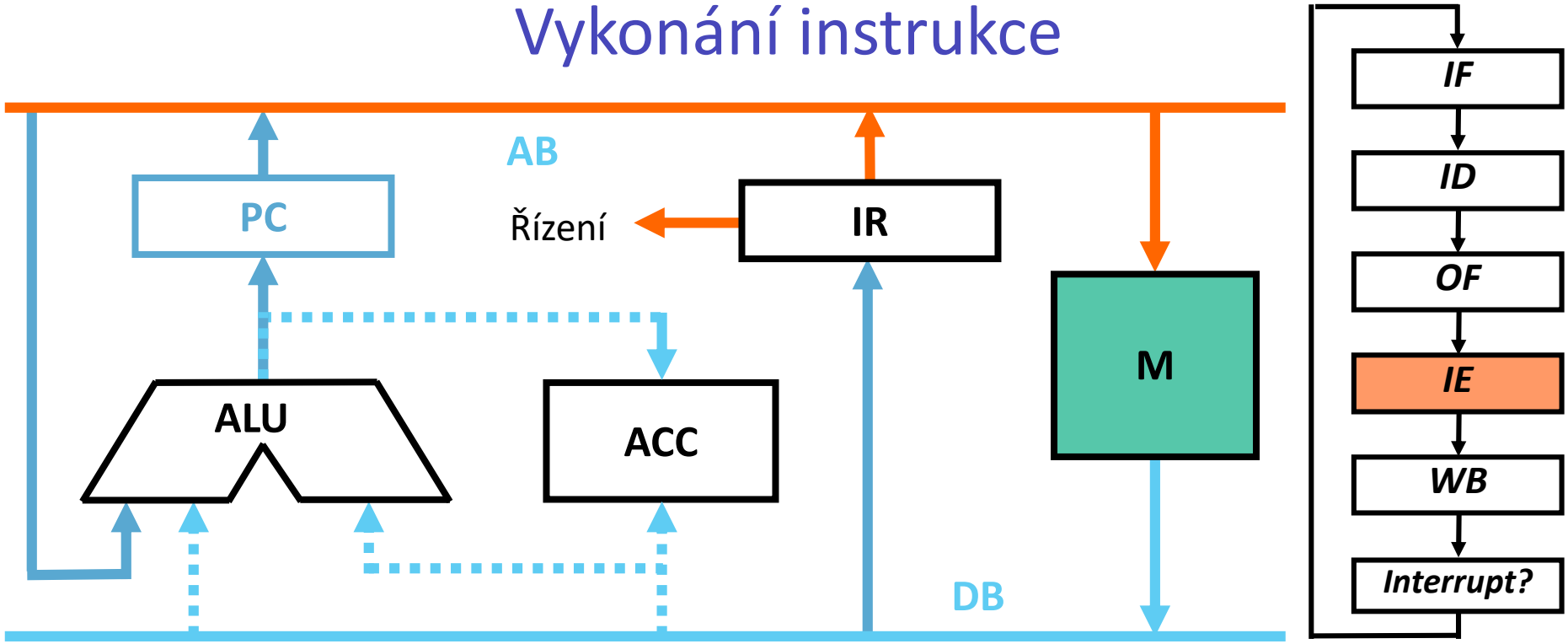




IR – zapsání adresy operandu, který má být při vykonávání instrukce použit, na AB

M – paměť vystaví hodnotu operandu na DB, takto je operand připraven pro zpracování buď v ALU nebo v ACC

Vykonání instrukce



IR – vystavuje na AB adresu hodnoty operandu, který má být při vykonání instrukce použit

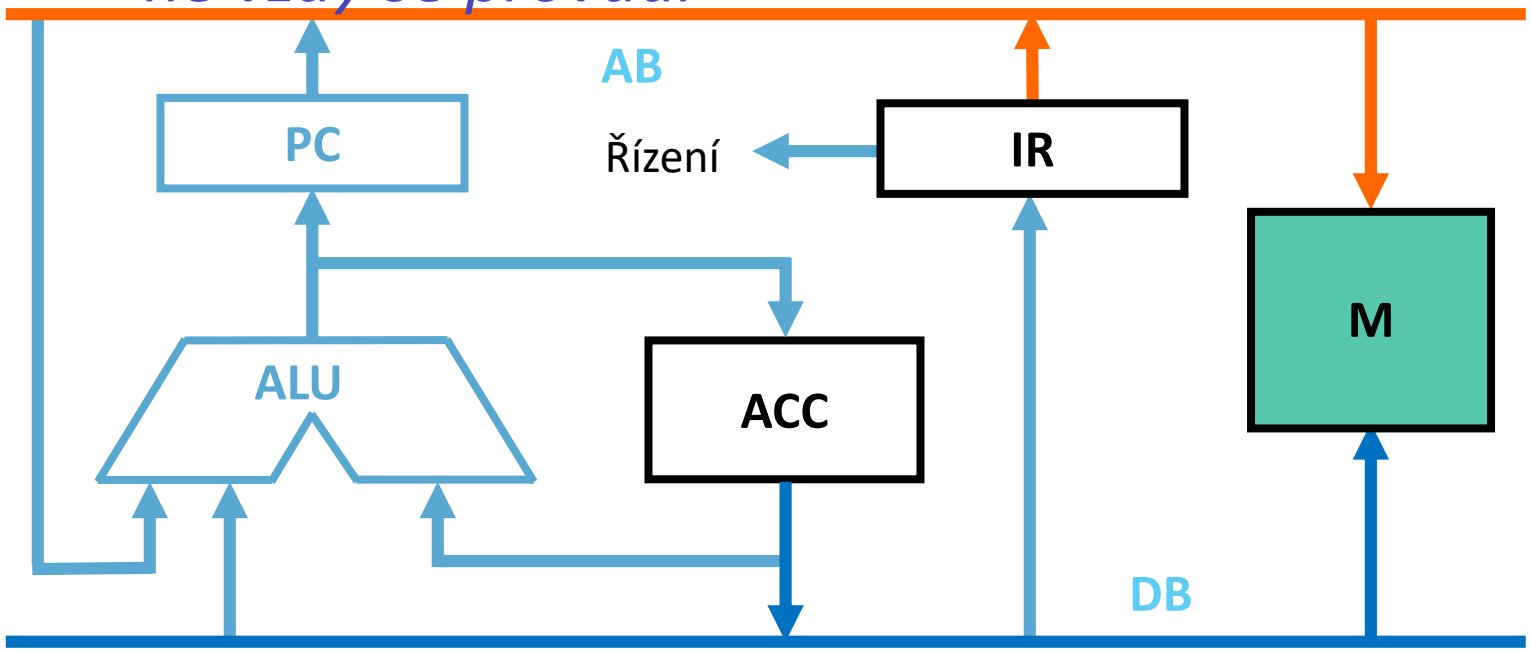
ALU – operace je vykonána v ALU podle instrukce, jejíž kód je v IR a generuje pomocí interní logiky řídící signály

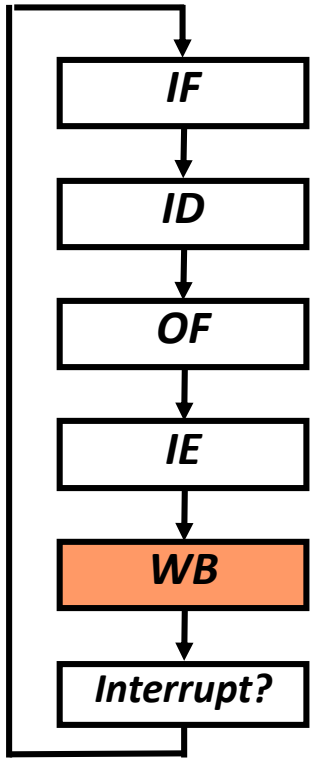
M – paměť může mít nadále vystavenou hodnotu operandu na DB

ACC – slouží jako cílový registr, může být ale také zdrojovým

Zápis výsledku

- *ne vždy se provádí*





IR – vystaví hodnotu adresy paměti, kam má být hodnota obsažená v ACC zapsána

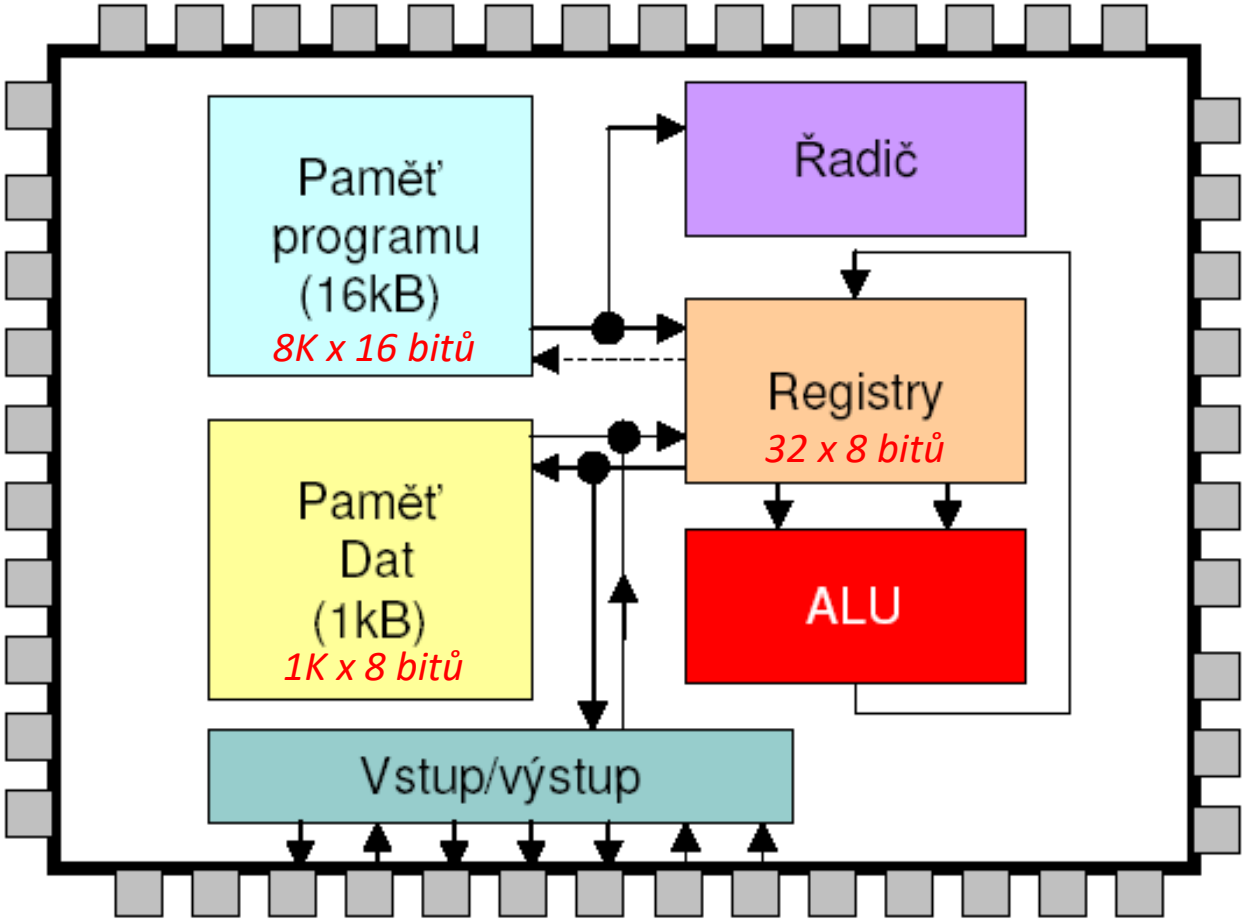
ACC – vystaví hodnotu obsaženou v něm na DB

M – hodnota z DB je zapsána do paměti

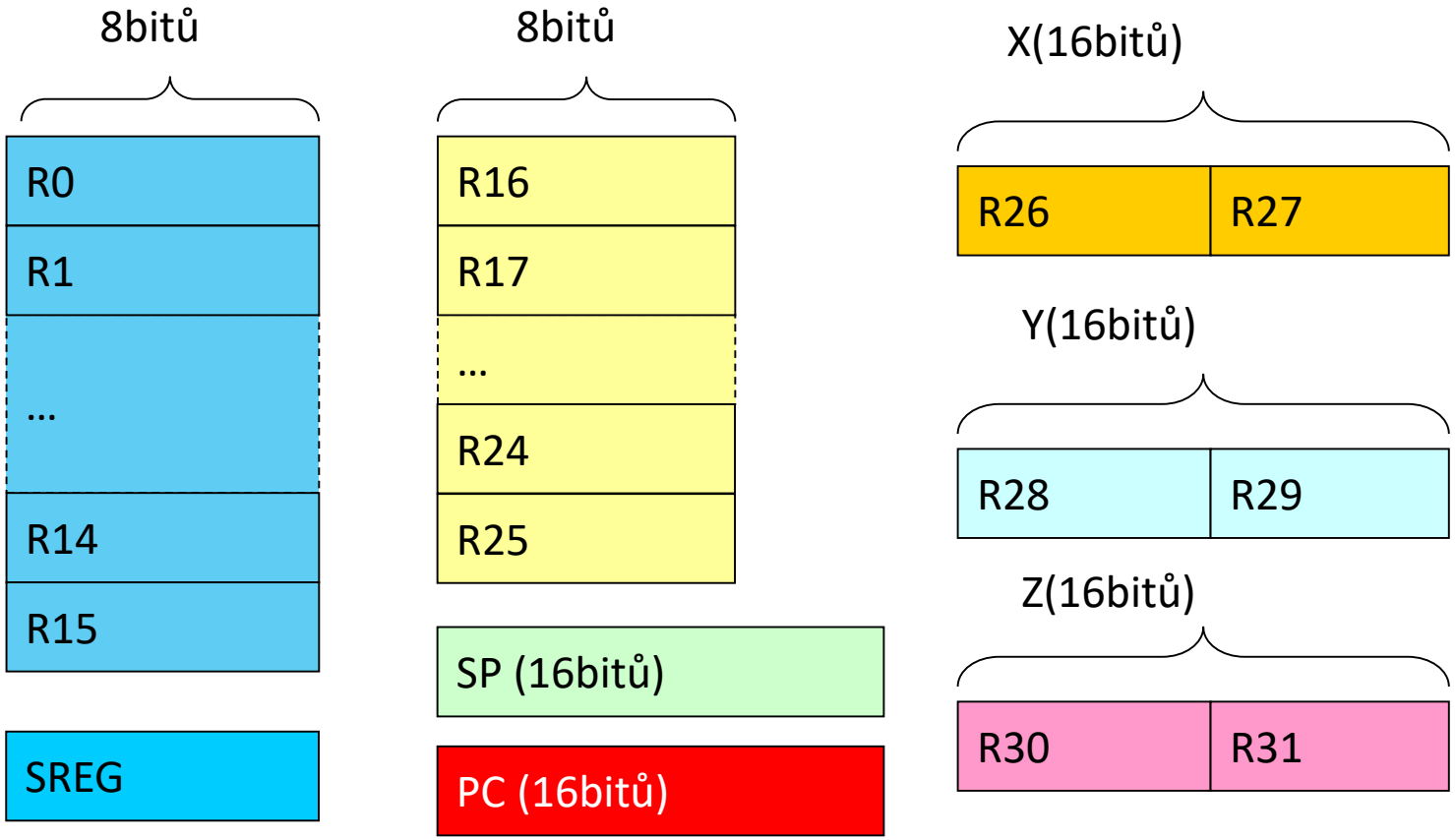
C je carry flag

Táto fáze základního cyklu může být, nebo nemusí být vykonána. V některých procesorech je táto fáze zákl. cyklu vykonávaná jako separátní instrukce.

Architektura **ATMEGA169**



Připomenutí: Registry AVR … 32



Stavový registr – SREG („flagy“)

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

C: Carry flag

Z: Zero flag

N: Negative flag

V: two‘s complement oVerflow indicator

S: N⊕V for signed tests

H: Half carry flag

T: Transfer bit used by BLD and BST instructions

I: Global Interrupt Enable/Disable flag

Instrukce

- Instrukce = existuje pro ni kód a umístění na určité adrese v paměti
- Pseudoinstrukce = např. vyhrazení místa pro proměnné, deklarace proměnných
- Direktiva = instrukce pro překladač
- Makroinstrukce = konkrétní instrukce se na několik míst v programu doplní až při překladu

Deklarace proměnných

= Pseudoinstrukce

- vyhrazení místa v paměti (pro výsledek)
- zadání vstupních dat

Příklad deklarace dat podle typu:

SHORT - jednobytový operand se znaménkem

BYTE - jednobytový operand bez znaménka

WORD - dvoubytový operand

obvyklá deklarace ... DS, DB, DW

- **AVR:** ... DB, DW (definuje konstantu), BYTE (rezervuje slabiku pro proměnnou),

  .... MACRO, ENDMACRO – začátek/konec makroinstrukce

Přesuny dat

- MOV *kam, co*

  kam .... registr

  co ... registr (obsah registru) … v AVR pouze registr

- LDI *registr, konstanta*

Instrukce pro přesuny mezi procesorem a pamětí:

- ST *paměť, registr*
- LD *registr, paměť*

- PUSH, POP … uložení/výběr ze zásobníku

Kde je zásobník, pokud ISA není zásobníková architektura?

Zásobník - stack

Operace:

- PUSH Rr... uložení registru na zásobník

  Rr → STACK

  post-dekrementace:

  Program counter:

  PC ← PC + 1

  Stack:

  SP ← SP – 1

- POP Rd ... nahrání registru ze zásobníku

  Rd ← STACK

  pre-inkrementace:

  Program counter:

  PC ← PC + 1

  Stack:

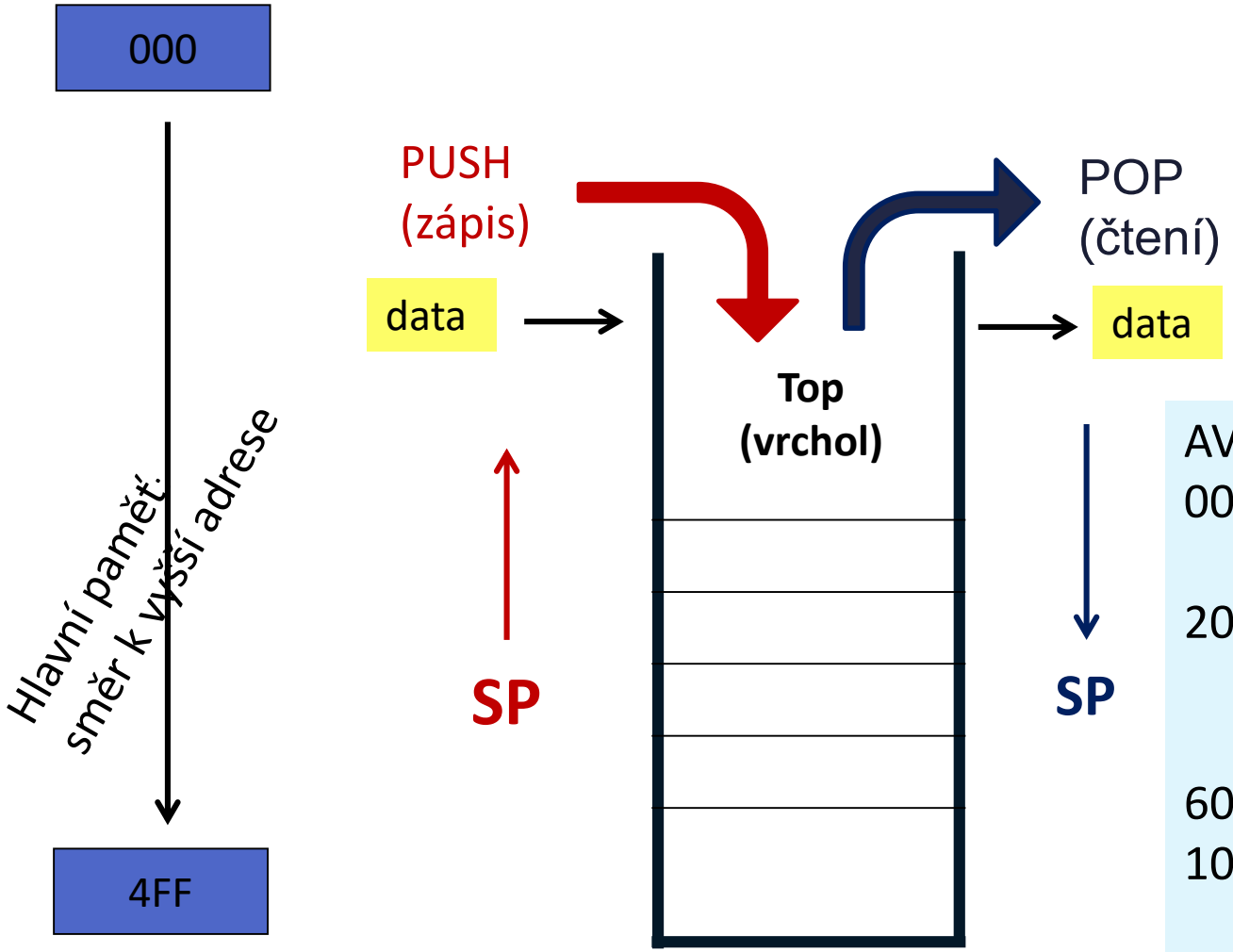
  SP ← SP + 1

Kde je v počítači registr SP, na co ukazuje a kde je celý zásobník?

Zásobník ...

- Zásobník roste směrem k nižším adresám (*podle typu procesoru – pak jsou jinak realizované operace PUSH a POP*)
- Obsah SP registru ukazuje na vrchol zásobníku ⟶ na první prázdnou položku, na kterou se bude zapisovat (*např. instrukcí PUSH, ale i při volání podprogramu CALL nebo při přerušení*)
- Je simulován v hlavní paměti (*v AVR v paměti dat, při psaní programu je třeba ho inicializovat*)

.... zásobník



AVR:
00h-1Fh registry
R0-R31,
20h-5Fh V/V registry
dosažitelné instr.
IN a OUT,
60h-FFh další V/V reg.,
100h-4FF paměť RAM
pro volné použití
a pro zásobník

v AVR je třeba inicializace zásobníku:

Zásobník inicializujeme na dostatečně vysokou adresu, protože roste směrem dolů. Maximální počáteční hodnota SP pro ATmega169 je 0x4FF

```
.include "m169def.inc"
    ldi r16, 0x00
    out SPL, r16
    ldi r16, 0x04
    out SPH, r16.
```

Vložení definičního souboru V/V registrů ATmega169 do zdrojového kódu.

Zápis hodnoty 400h do SP

Poznámka: V definičním souboru m169def.inc jsou adresy dolní a horní poloviny SP registru definovány takto:
.equ SPL = 0x3d (dolní polovina SP)
.equ SPH = 0x3e (horní polovina SP)

Zásobník ... použití

- Ukládání návratových adres na zásobník při volání podprogramu
- Uložení návratové adresy při vyvolaném přerušení
- Předávání parametrů podprogramům
- Lokální proměnné podprogramů
- Dočasné uložení registrů pro zachování transparentnosti kódu. Typicky v obsluze přerušení.

Aritmetické

- binární
  - ADD
  - ADC
  - SUB
  - SBB
  - CMP
- unární
  - NOT
  - INC
  - DEC

CMP, CP – porovnání:
jako SUB, ale neuloží výsledek, jen příznaky
****

AVR používá místo SBB .... SBC
CP: porovnání registrů,
CPC: porovnáni registrů včetně carry
CPI: porovnání registru s konstantou

Úkol: který bit AVR při odčítání skutečně používá, B (Borrow) nebo C (Carry)?

Logické

- binární
  - AND
  - OR
  - XOR
- unární
  - NEG

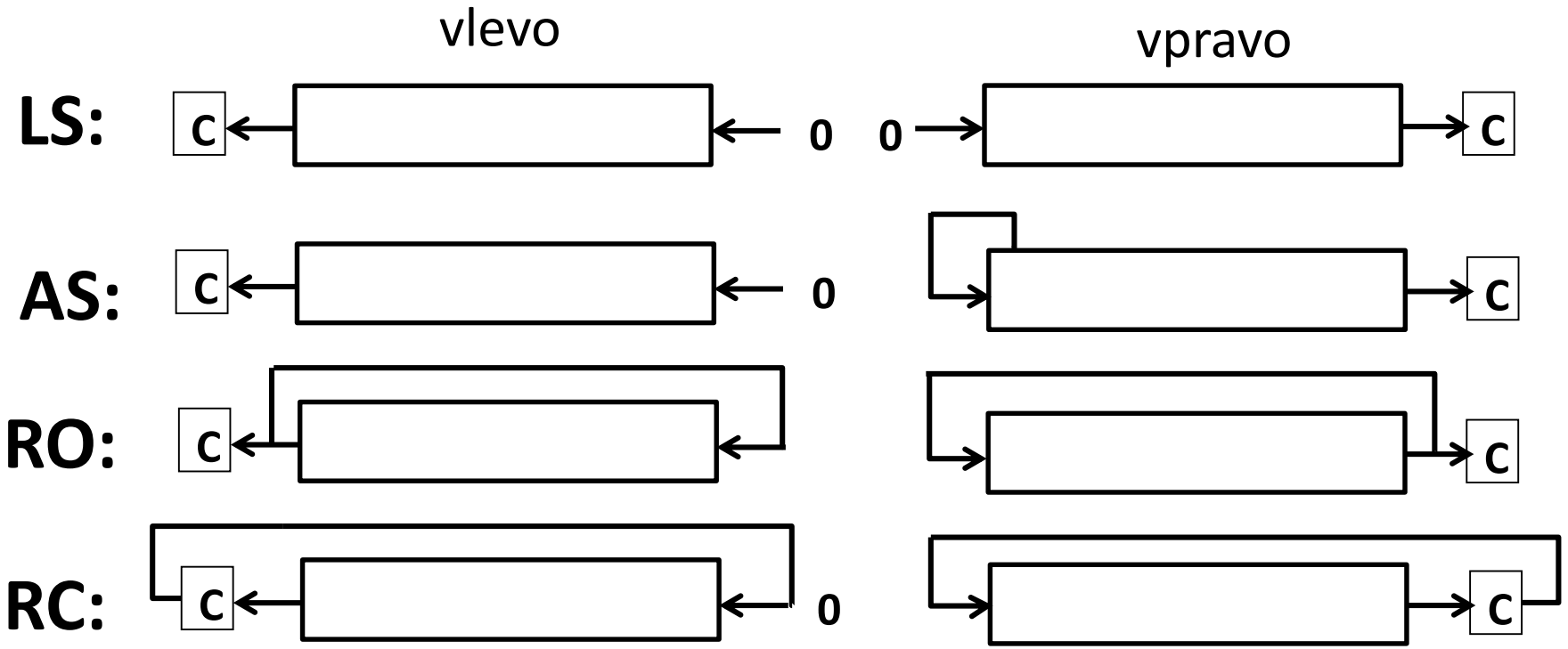
****

AVR:

- EOR (XOR), COM (negace)
- CLR vynulování ... (Rd←Rd ⊕ Rd)
- SER nastavení

Posuvy

- Logický posuv (Shift, LS)
- Aritmetický posun vpravo (a vlevo)
- Cyklické posuvy (rotace nebo rotace přes Carry flag RC)



Posuvy … AVR

- LSL, LSR – posuvy logické (SHIFT)
- ROL, ROR – rotace přes Carry
- ASR – aritmetický posuv vpravo

- Proč není v AVR aritmetický posuv vlevo?

Další instrukce, viz dokumentace AVR na courses SAP/laboratoře

Skoky

Změna pořadí provádění instrukcí

- Nepodmíněný skok JMP
- Podmíněné skoky BRxx *(BR namísto J):* změna běhu programu bez zapamatování místa odkud se „skákalo“

Na rozdíl od volání podprogramu, kdy je třeba si zapamatovat, kam se vrátit.

Příklad …. AVR

BRBC: skok jestliže bit S ve SREG je „0“

Syntax:	Operandy:
BRBC S,k	$0 \le s \le 7$, $-64 \le k \le +63$

Operace:

If SREG(s) = 0 then PC ← PC + k + 1 else PC ← PC + 1

Programový čítač:

PC ← PC + **k** + 1

PC ← PC + 1 když podmínka není splněna

Podobně BRBS

Další podmíněné skoky

- BRCS – BRCC ..... S – set, C – clear

  Syntax: BRCS k *C je carry flag*
- BREQ ... Z=1, BRNE ..... Z=0, *Z je zero flag*
- BRGE ... N xor V = 0 *(v AVR test příznaku S = N xor V)*
- a další ....
- Procesor umí nastavit či vynulovat příznaky v *SREG*

(např. pro carry: set SEC, clear CLC)

- Ulehčení se zaměřením na kód zobrazení čísel:

Relace	Doplňkový kód	Nezáporná čísla
$<$	BRLT	BRLO
$=$	BREQ	BREQ
$\neq$	BRNE	BRNE
$\geq$	BRGE	BRSH

Změna pořadí provádění instrukcí

- Instrukcemi skoku:
  - nepodmíněný ….. JMP, *i relativní (PC), nepřímý (Z)*
  - podmíněný … BRxx *(a další)*

  realizace: změna obsahu PC
- Voláním podprogramu …….. CALL

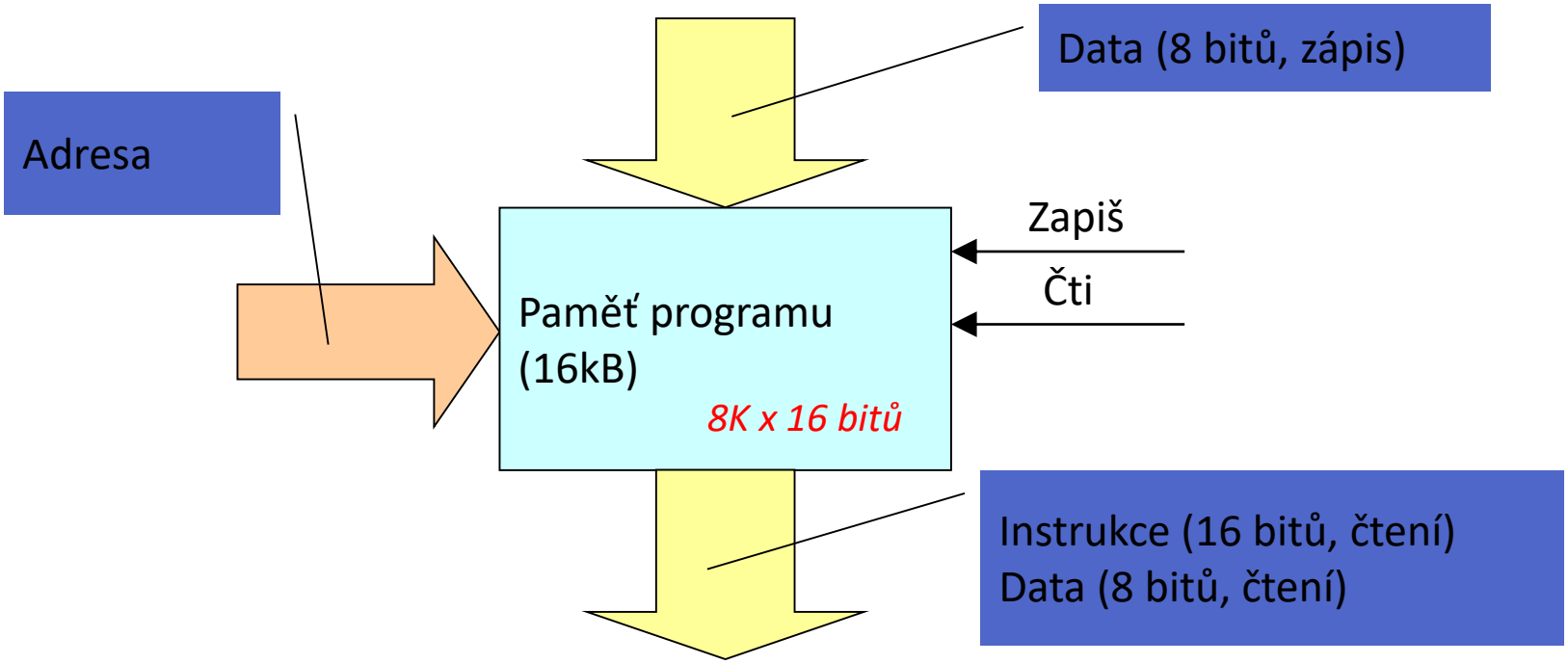
  realizace:
  - *změna obsahu PC a uložení návratové adresy na zásobník,*
  - *při návratu (konci) podprogramu se pomocí instrukce* RET *nastaví obsah PC na adresu instrukce následující po* CALL
- Přerušením ... *viz dále*

Podprogramy

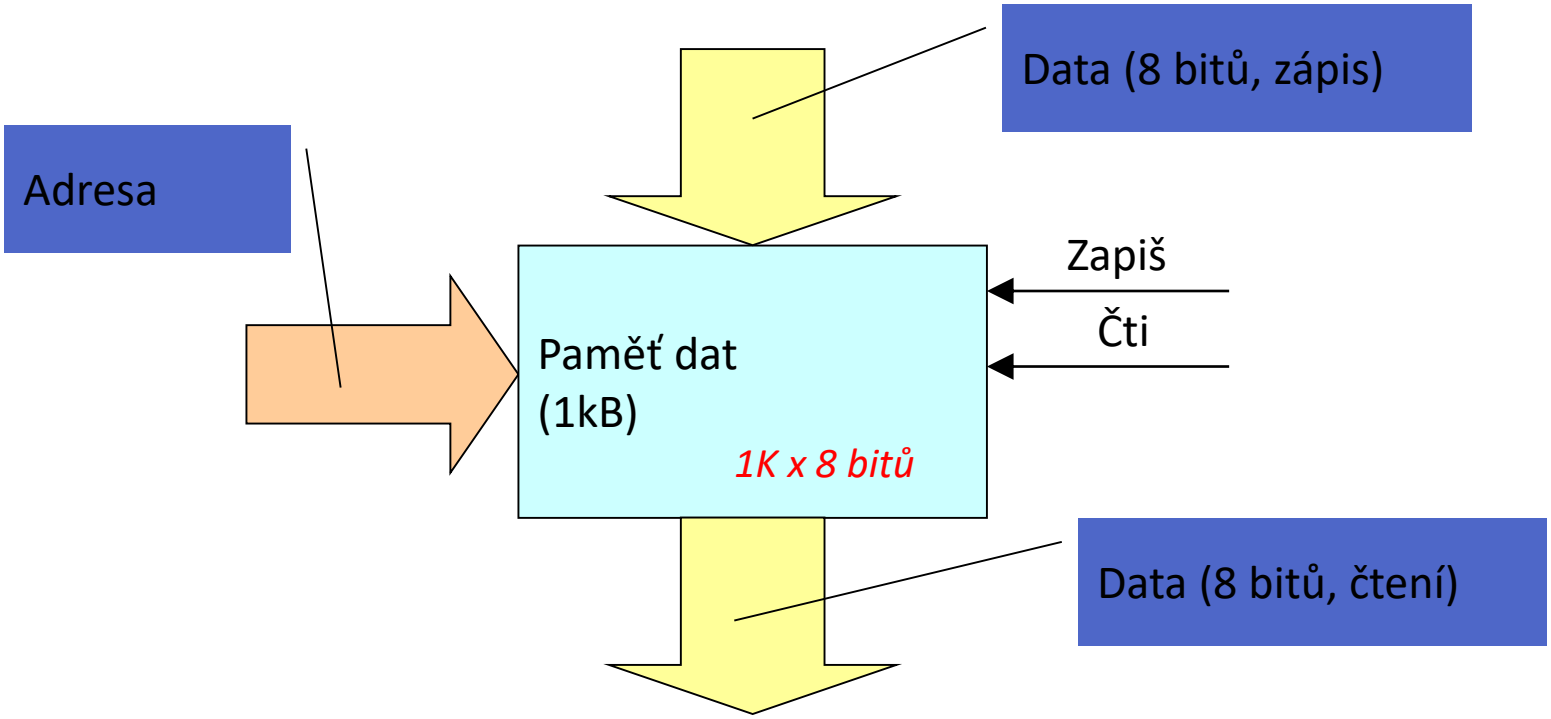
Volání ... CALL *k* *k je adresa, i relativní a nepřímé*

- Umožňuje strukturovat program a vytvářet kratší programy
- Při tzv. volání je nutné na rozdíl od skoků zapamatovat, kam se vrátit, tzn. *uložit návratovou adresu*
- Ukládá se na **zásobník**
- Při návratu (na konci podprogramu je třeba instrukce RET … návrat z podprogramu)
- Předávání parametrů: přes registry, paměť nebo zásobník

Paměť programu



Paměť dat


Data (8 bitů, zápis)
Adresa
Zapiš
Čti
Paměť dat
(1kB)
1K x 8 bitů
Data (8 bitů, čtení)

Podprogramy: princip

- Volání ... CALL *k* — *k je adresa*

Programový čítač:

PC ← *k*

Stack (zásobník, simulovaný v hlavní paměti):

STACK ← PC + *2* .. *podle délky instrukce*

Ukazatel zásobníku SP

post-dekrementace: SP ← SP – 2, (2 slabiky, 16 bitů)

- Návrat z podprogramu .... RET

Při volání se ukládá návratová adresa na zásobník,

Při návratu se obnovuje „původní“ obsah PC ze zásobníku.

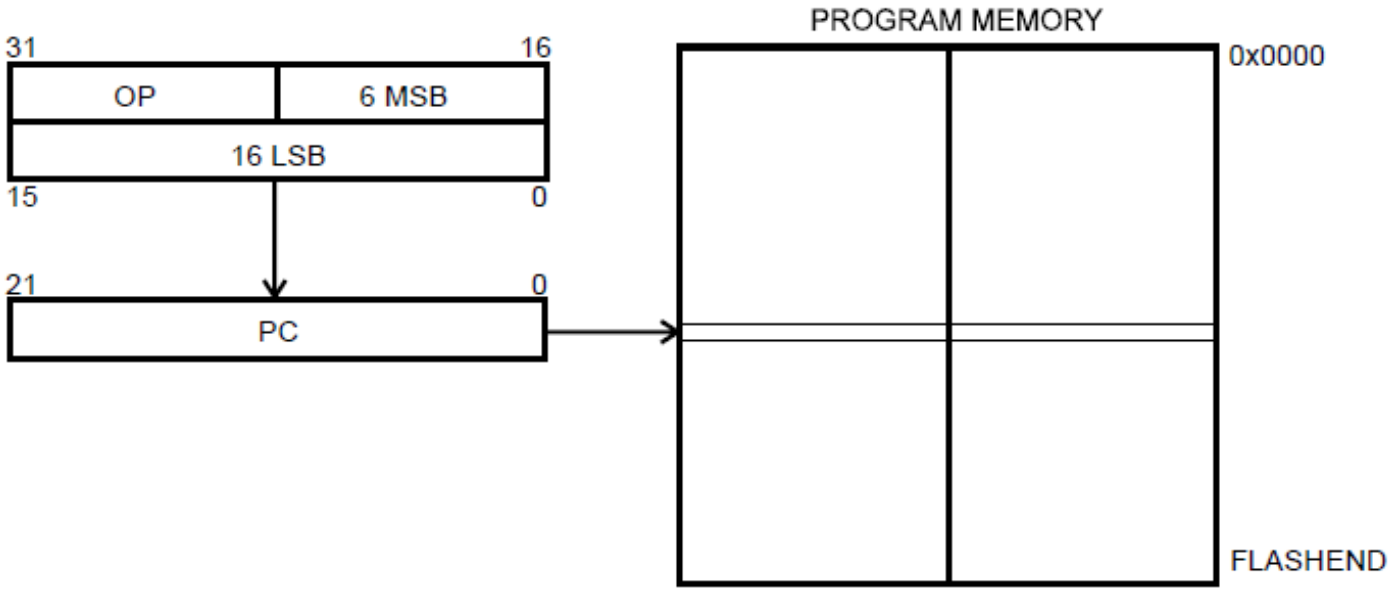
Ukazatel zásobníku:

pre-inkrementace: SP ← SP + 2, (2 slabiky, 16 bitů)

Kontrolní otázka: Proč se při instrukcích PUSH/POP odčítá/přičítá k SP jednička a při CALL dvojka?

Skok a volání podprogramu

- PC ← k .... instrukce má 32 bitů

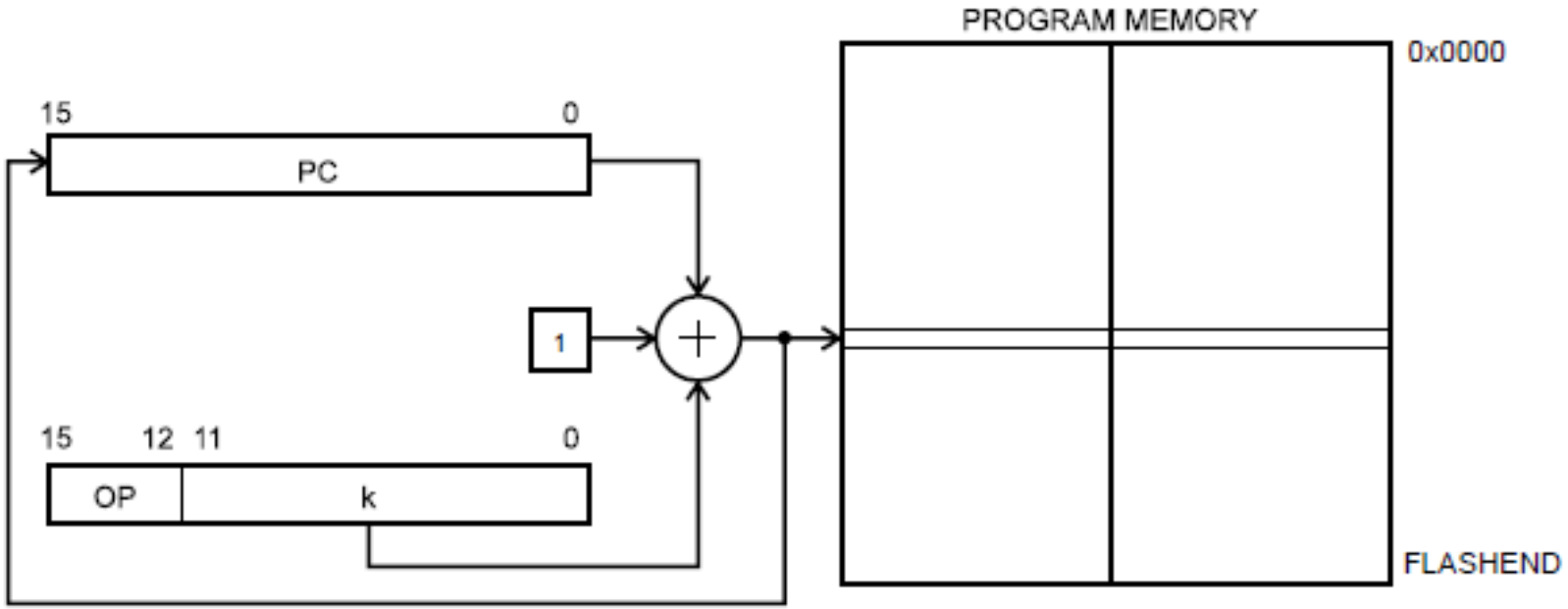


i pro vyšší verze AVR

U AVR existují další typy volání: relativní: RCALL, nepřímý přes Z: ICALL ... jen 3 takty

Relativní skok a volání podprogramu

- PC ← PC + *k* + 1 adresa *k* je z intervalu -2048 to 2047 ... 12bitů

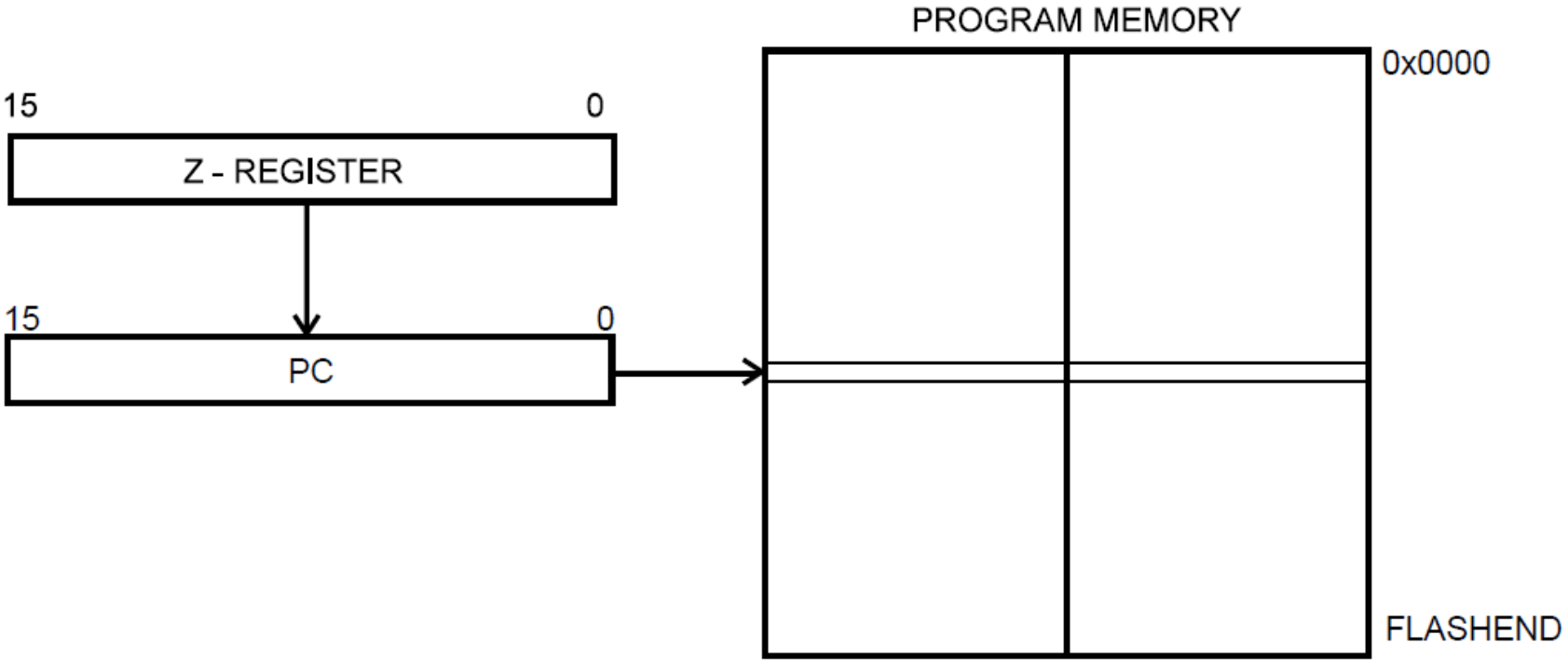


Nepřímý skok a volání podprogramu

ICALL, IJMP ... žádný operand, PC ← Z,

pro ICALL: STACK ← PC + 1 (instrukce má jen OZ/OP (16 bitů) a žádný operand)

SP ← SP – 2, (2 slabiky, 16 bitů)



Příklad ... AVR volání podprogramu

Př. Sečtěte:

8000h + 0100h + 0100h

```
ldi   r16, 0x00
ldi   r17, 0x80
ldi   r18, 0x00
ldi   r19, 0x01
call  add16

…
call  add16
….
```

Podprogram

```
; Součet dvou 16bitových čísel
; vstup: R17:R16 sčítanec 1
; R19:R18 sčítanec 2
;
; výstup: R17:R16 součet
;
; používá: R16, R17,
; R18, R19, SREG

add16:  add r16, r18
        adc r17, r19
        ret
```

Parametry jsou předávány v registrech

Poznámka: nezapomeňte přidat na začátek programu inicializaci zásobníku

Přerušení - interrupt

- původně obsluhu V/V zařízení řídil procesor
- z důvodů rychlosti snaha o nezávislost V/V operací na procesoru (*historicky první paralelismus*)
- nutnost synchronizace, upozornění V/V zařízení na své požadavky → přerušení
- příčina se obvykle zjistí až při „obsluze přerušení“ (= kus programu, podprogram pro zpracování přerušení)

... přerušení ...

- Vnější (periférie, uživatel)
  - Nemaskovatelné (**NMI**)
  - Maskovatelné (z řadiče přerušení)
- Vnitřní
  - Chyby operandů, výsledků, krokování,…
  - Instrukce **INT n** (n konstanta) … ***neumí všechny procesory***
- Postup při obsluze přerušení
  1. Uloží se na zásobník informace o právě probíhajícím programu: **SREG, PC**
  2. Zakáže se další přerušení
  3. Nastaví se **PC** na adresu začátku podprogramu, který provádí „obsluhu“ daného typu přerušení
  4. Provedení přerušení (= obsluha, realizace podprogramu pro toto přerušení)
  5. Návrat z programu obsluhy přerušení a obnova informaci patřící původnímu programu, ve kterém bylo přerušení vyvoláno **SREG, PC**

... přerušení ...

Maskování přerušení pomocí příznaku **I** – ve stavovém registru SREG

- Povolení např. pomocí nové instrukce **STI**, která nastaví **I**, tj. $\mathbf{I} \leftarrow 1$
- Zákaz např. pomocí nové instrukce **CLI**, která nuluje **I**, tj. $\mathbf{I} \leftarrow 0$

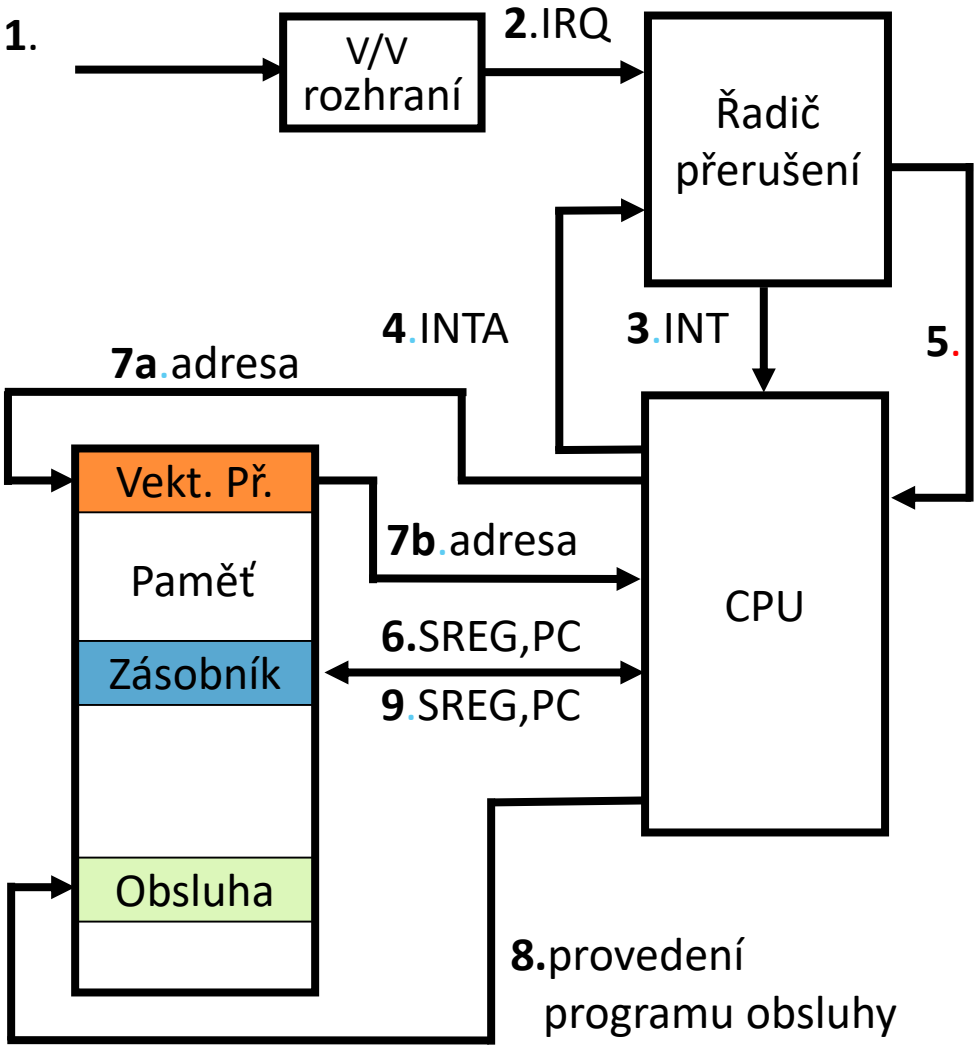
Vektor přerušení

- Obsahuje adresu podprogramu obsluhy přerušení, která je zapsána do **PC**
- V paměti *někdy bývá* na nejnižších hodnotách vyhrazen prostor, kde je umístěno **n** (INT n) vektorů přerušení, tj. adres podprogramů pro zpracování daného typu přerušení … **v AVR si přerušovací rutiny musí napsat programátor**

Řadič přerušení

- Hardwarové zařízení přijímající signály z vstupně-výstupných zařízení
- Identifikuje požadavky na přerušení podle jejích priorit **IRQ** (Interrupt request)
- Generuje přerušovací signál ***INT***

Přerušení x86



HW

1.- 3. Vznik žádostí o přerušení

4. Rozhodnutí o obsluze (**IF**=1 a **INTA**)

5. Identifikace příčiny přerušení (číslo typu)

6. Uložení stavové informace **SREG** a **PC** na zásobník

7. Nalezení začátku podprogramu pro obsluhu daného typu přerušení pomocí vektoru přerušení (nové **PC**)

8. Provedení podprogramu obsluhy přerušení

9. Návrat do přerušeného programu a obnovení **PC** a **SREG**

SW

HW počítače - shrnutí

- **Všechny počítače se skládají z 5 základních částí:**
  - Datová část (ALU) - v procesoru
  - Řídící část (řadič) – v procesoru
  - Hlavní paměť – často mimo procesor
  - Vstupní zařízení
  - Výstupní zařízení
- **Paměťový systém**
  - Caches: rychlé, dražší, kapacitně menší, umísťované blíž k procesoru
  - Hlavní paměť: pomalejší, levnější, větší
  - Vnější paměť: ještě pomalejší, ale velká kapacita
  - Záložní paměť: CD, DVD, flash, magnetická páska
- **Vstupní a výstupní zařízení mají nejméně pravidelné struktury:**
  - Široký rozsah rychlosti: klávesnice vs. grafika
  - Široký rozsah požadavku: rychlost, cena, norma atd.