

Architektura počítačových systémů

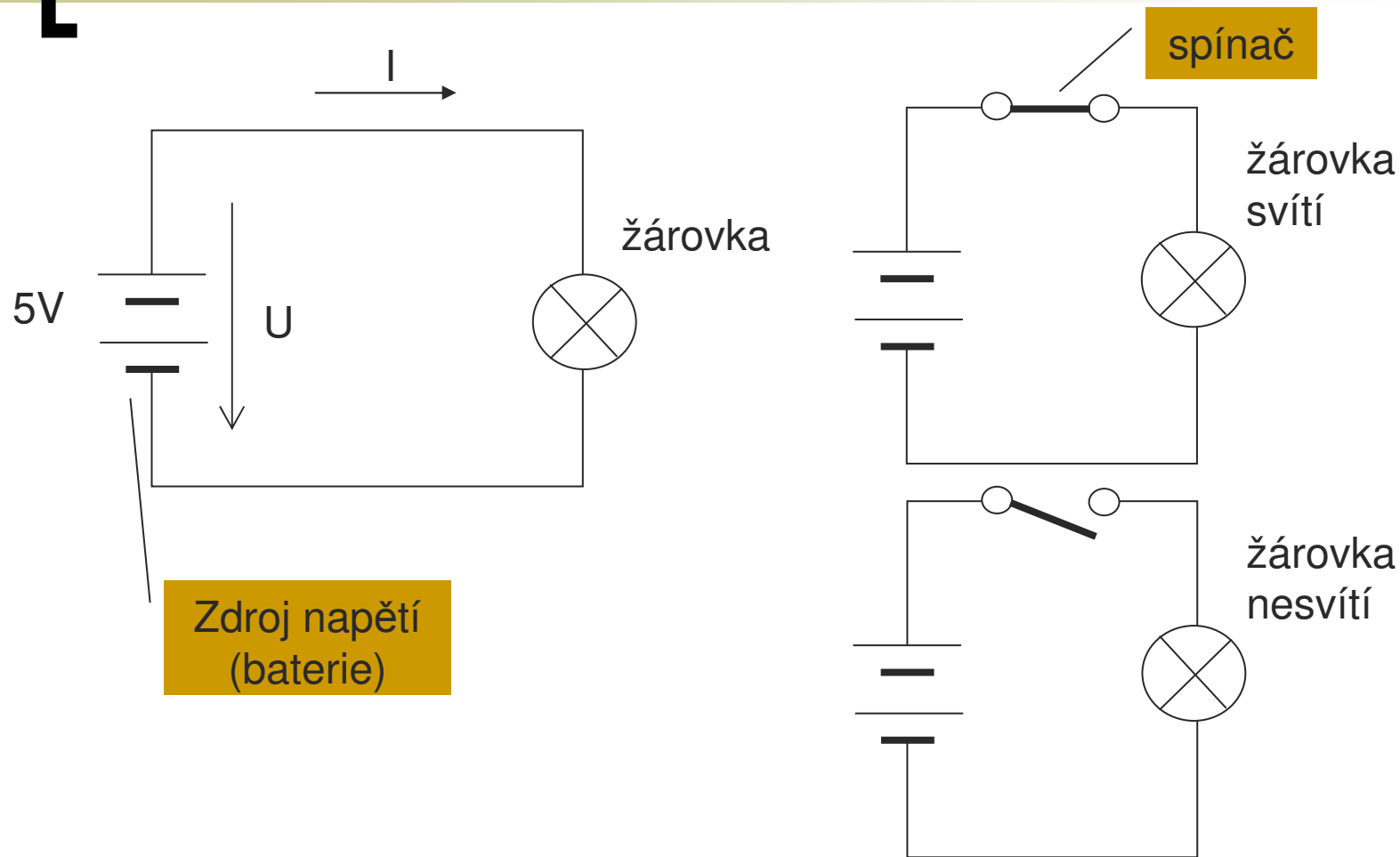
UAI/606 Přednáška blok 3

Miroslav Skrbek
mskrbek@prf.jcu.cz

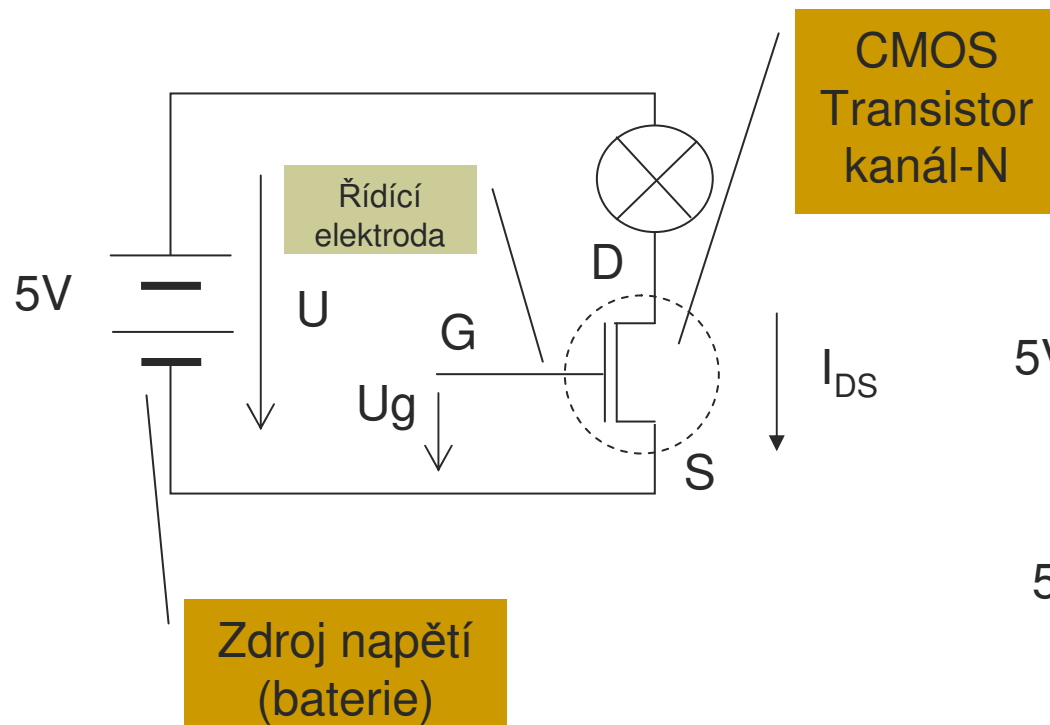
*Ústav aplikované informatiky
Přírodovědecká fakulta
Jihočeské univerzity v Českých Budějovicích*

Platné pro šk.r. 2013/2014

Elektrický obvod – opakování fyzika zš, sš

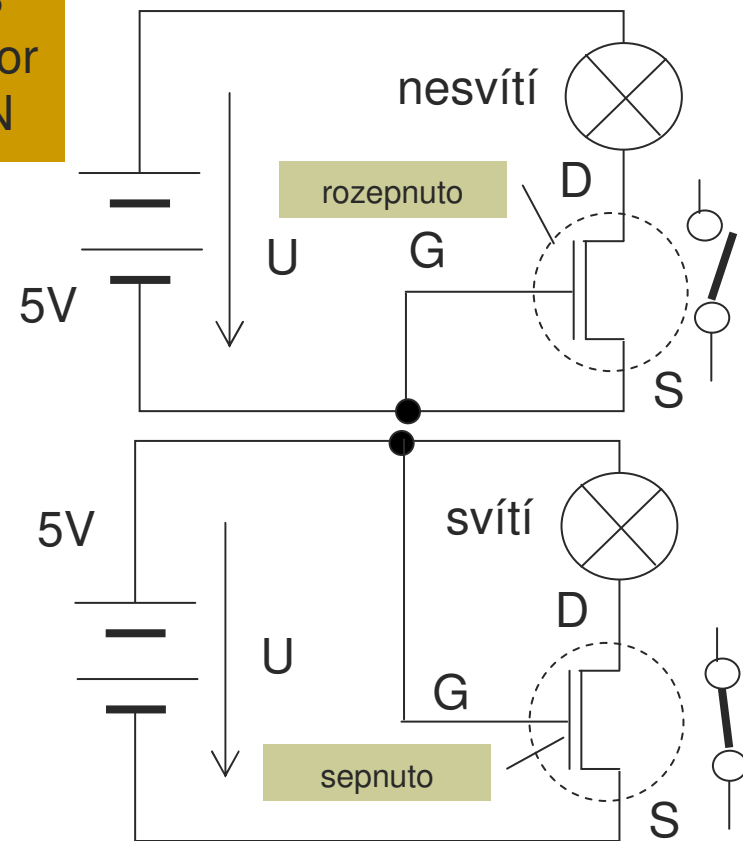


[Tranzistor CMOS kanál N]

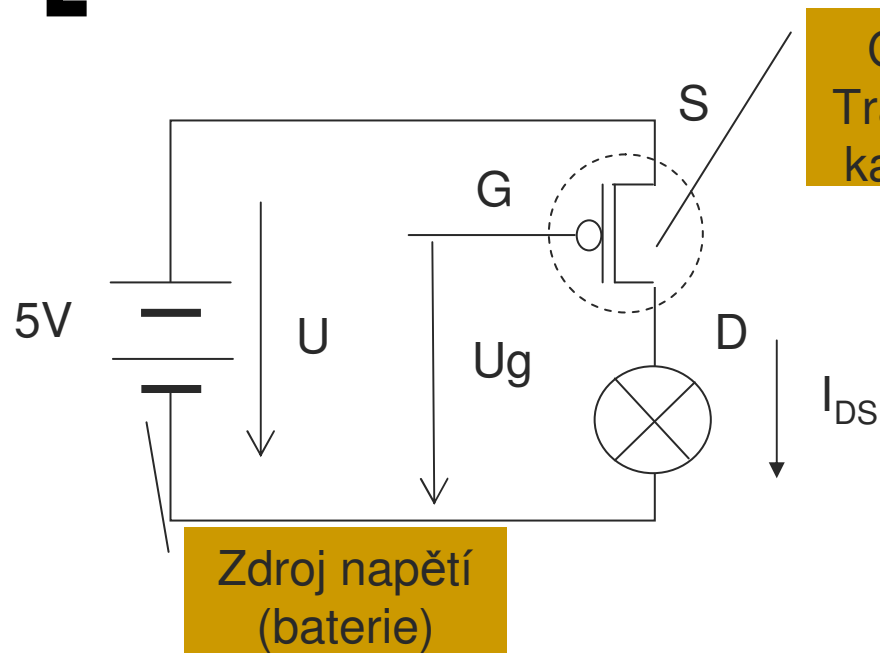


$U_g > 0,7V \dots$ sepnuto (prochází proud)

$U_g < 0,7V \dots$ rozepruto (neprochází proud)

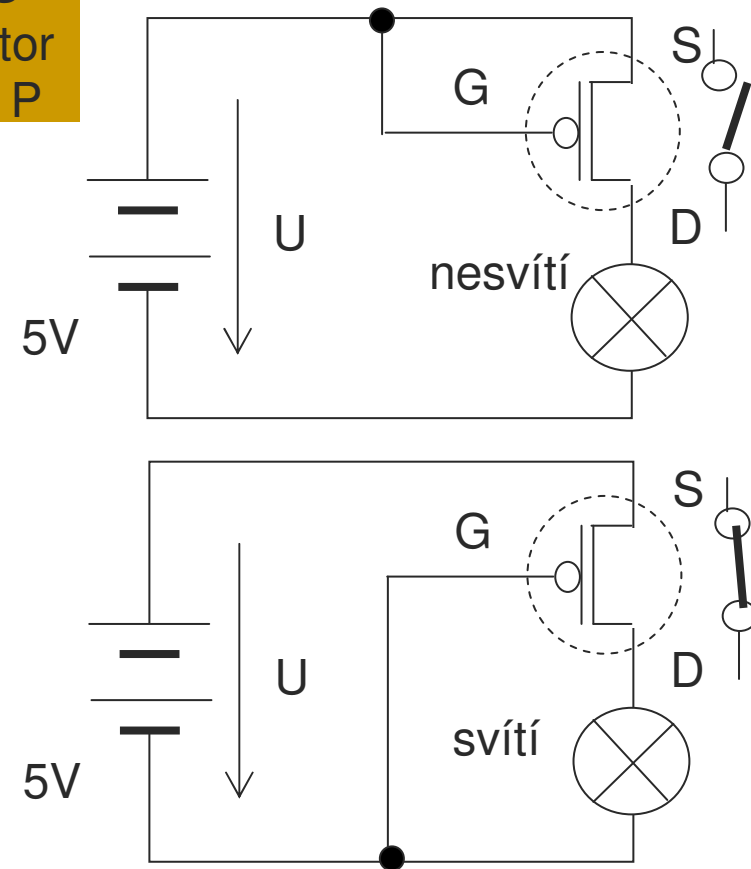


[Tranzistor CMOS kanál P]



$U_g < U - 0,7V$... sepnuto (prochází proud)

$U_g > U - 0,7V$... rozepnuto (neprochází proud)



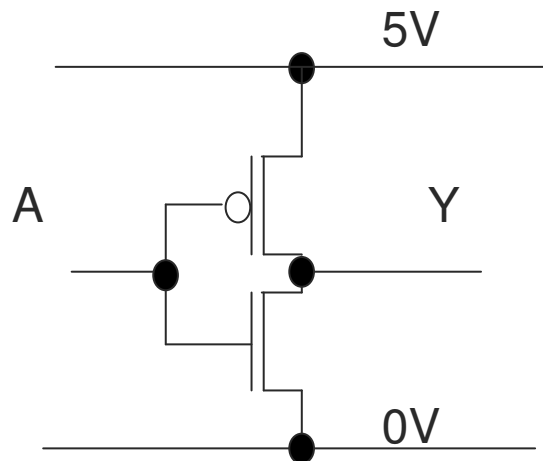
[Logický obvod - invertor]

Logické úrovně (zjednodušeno)

5V ... logická jednička **1**

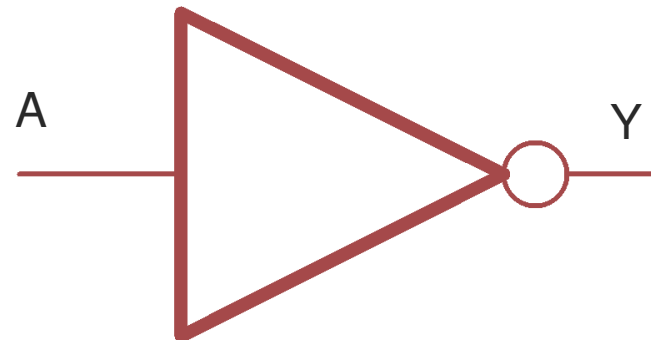
0V ... logická nula **0**

Invertor (negace)

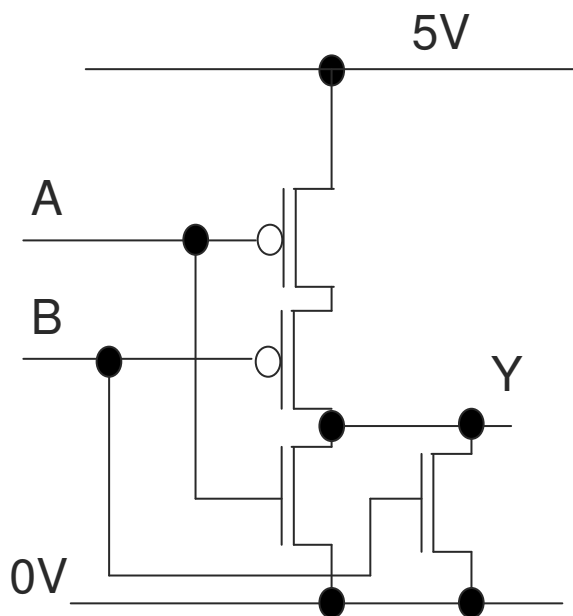


$$Y = \neg A$$

A	Y
0 (0V)	1 (5V)
1 (5V)	0 (0V)

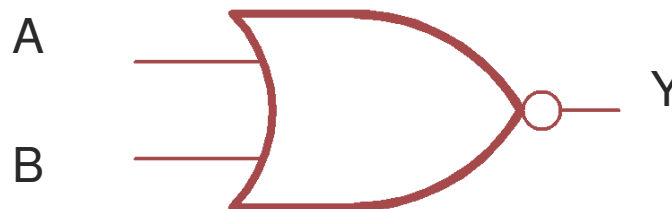


[Logický obvod - NOR]



$$Y = \neg(A \vee B)$$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

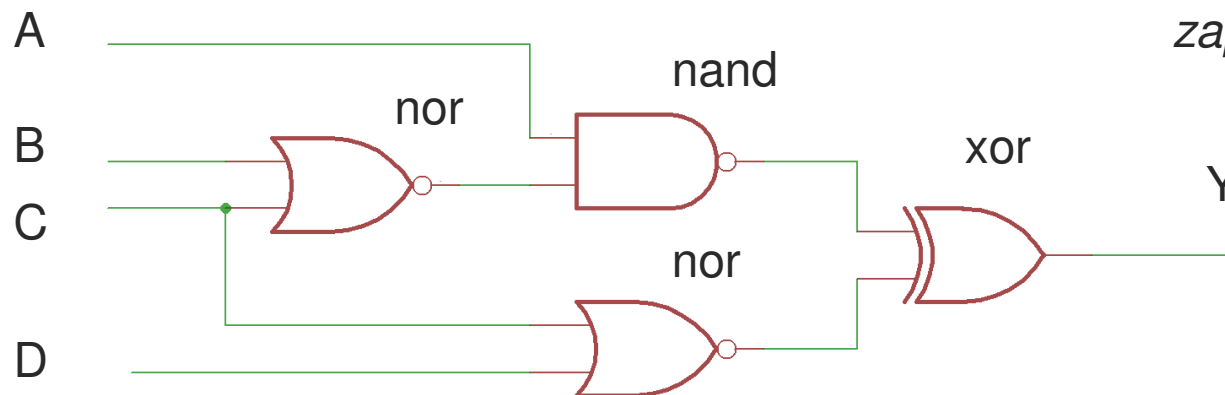


[Kombinační logické obvody]

Stavební prvky: logické obvody AND, OR, NOR (negovaný OR), AND, NAND (negovaný AND), XOR (exclusive or), invertor

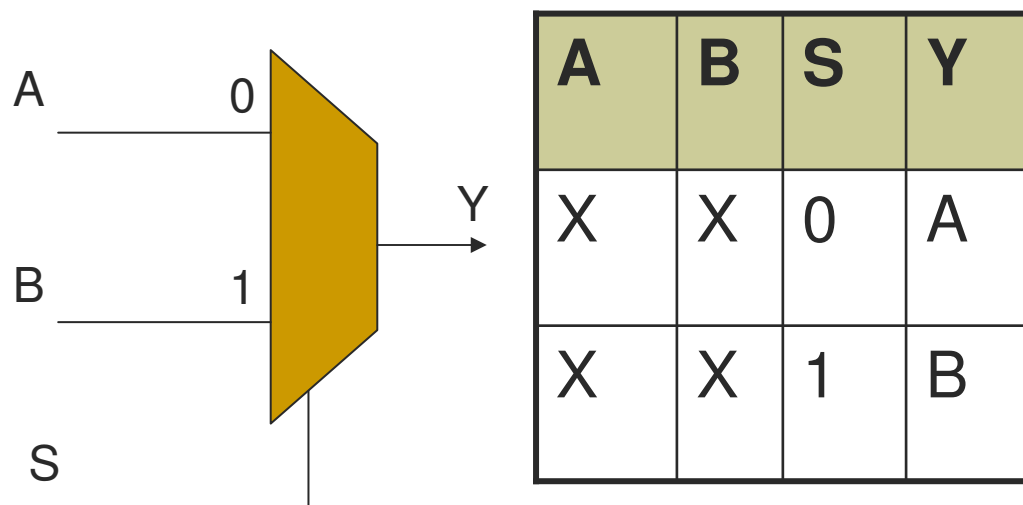
Pospojováním se tvoří složitější logické funkce.

*Sestrojte
pravdivostní
tabulku pro toto
zapojení*



[Multiplexer]

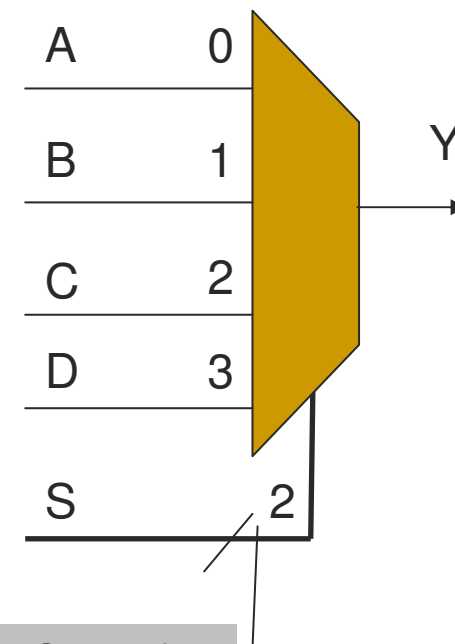
Dvouvstupový multiplexer



X – log. nula nebo jedna

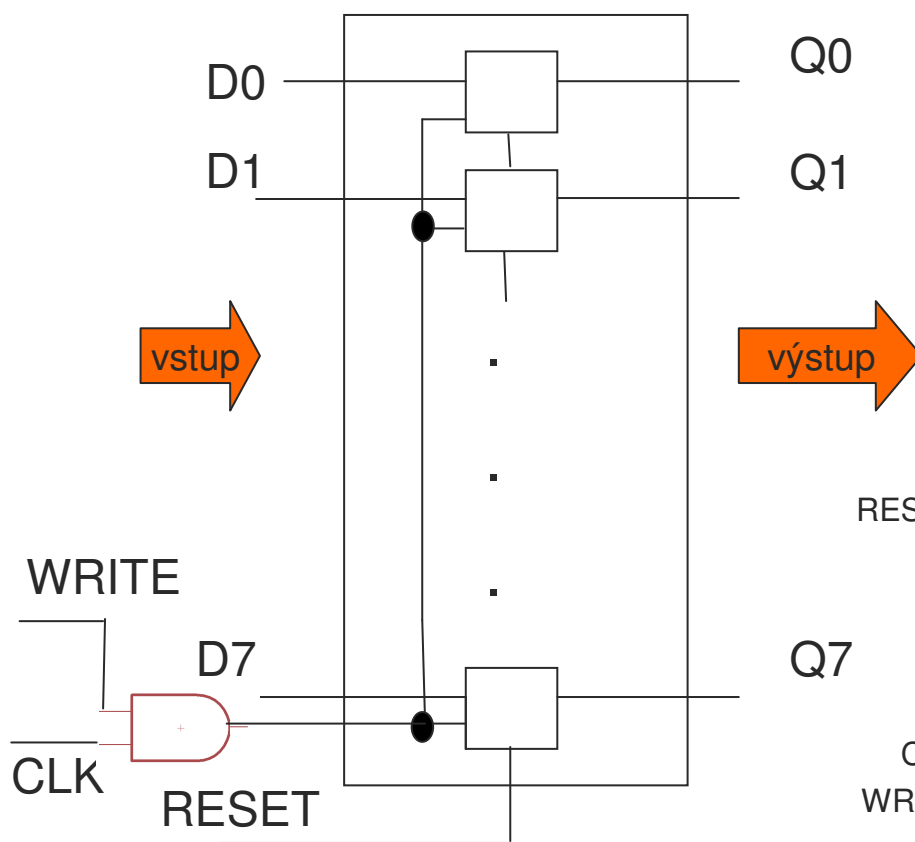
Multiplexer vybírá jeden ze dvou nebo více vstupů na jediný výstup Y. Můžete si tento obvod funkčně představit jako přepínač. Vybraný vstup je určen vstupem S.

Čtyřvstupový multiplexer



Označuje dva vodiče

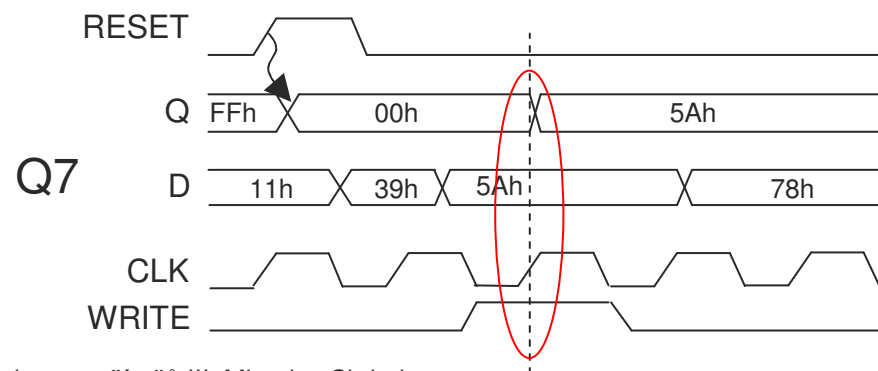
[Registr]



Sada klopných obvodů D se společným hodinovým vstupem tvoří registr.

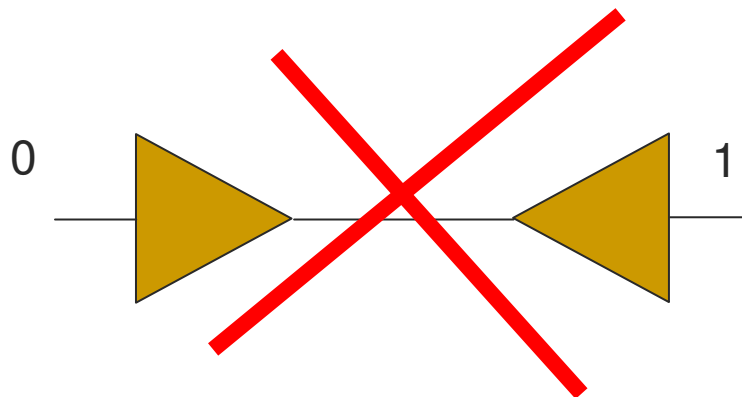
Takto si například představte registry v procesoru a CLK jako hodinový kmitočet procesoru např. 3GHz

Registr na obrázku je schopen si zapamatovat číslo v rozsahu 0-255, tedy má kapacitu 1 byte.

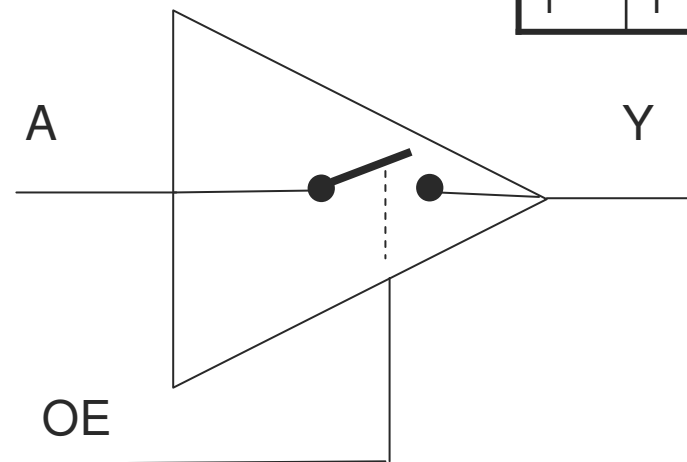


[Třístavové budiče]

Výstupy logických obvodů nelze spojit – hrozí zkrat při rozdílných logických úrovních.



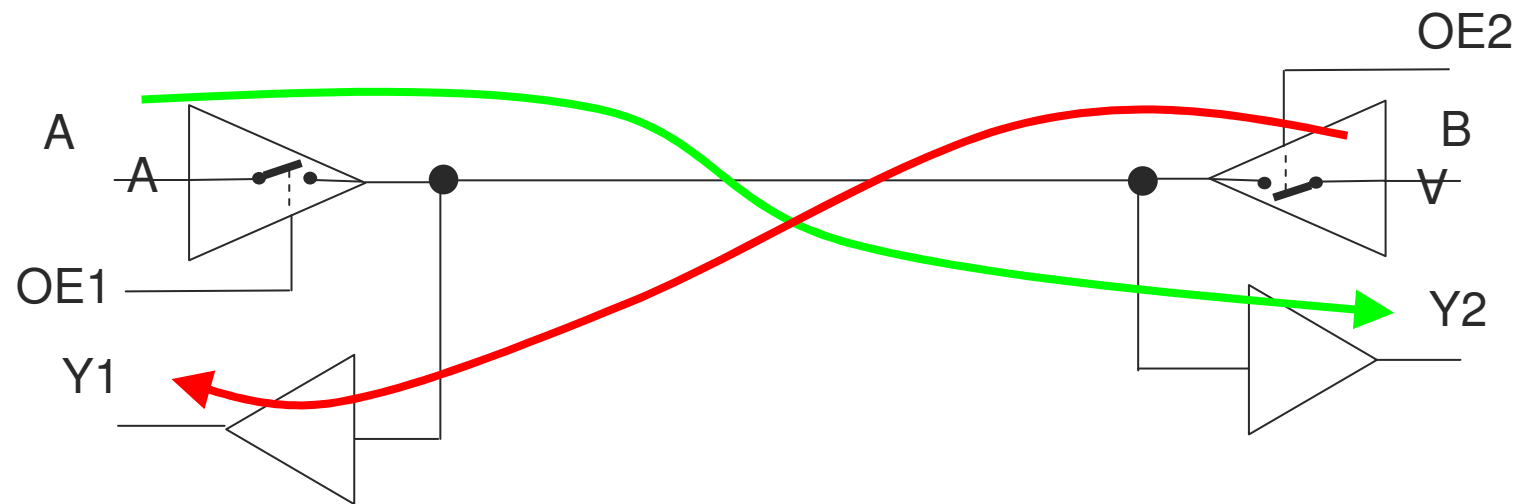
Třístavový budič



OE	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Pokud $OE=1$, pak Y kopíruje logickou hodnotu na A. Pro $OE=0$ se budič odpojí a na výstupu Y nevynucuje žádnou logickou úroveň. Mluvíme o tom, že výstup je ve třetím stavu a značíme Z.

[Obousměrná sběrnice]



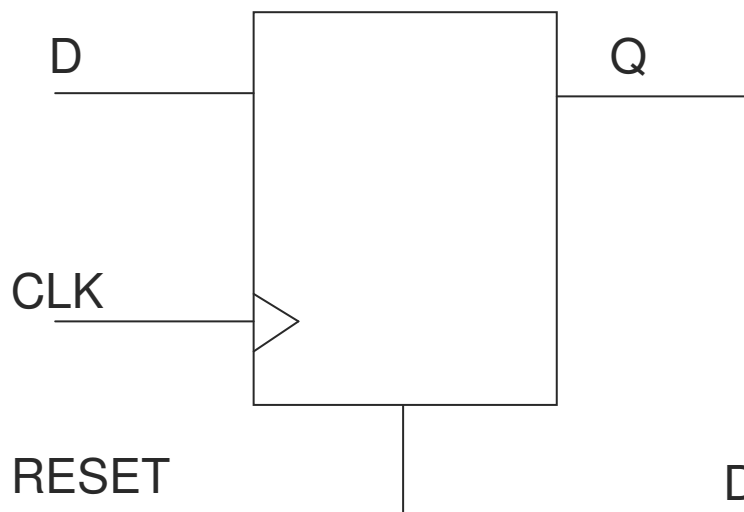
Přenos zleva doprava: OE1=1, OE2=0 (musí být! nula, jinak zkrat), Y2 má stejnou hodnotu jako A

Přenos z prava doleva: OE1=0 (musí být! nula, jinak zkrat), OE2=1, Y1 má stejnou hodnotu jako B

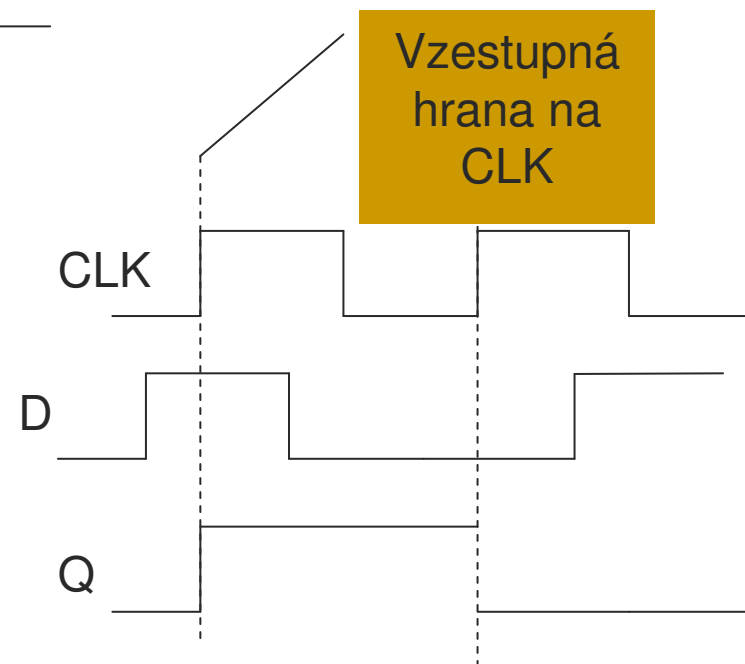
[Klopné obvody (typ D)]

Klopný obvod D je jednobitovou pamětí.

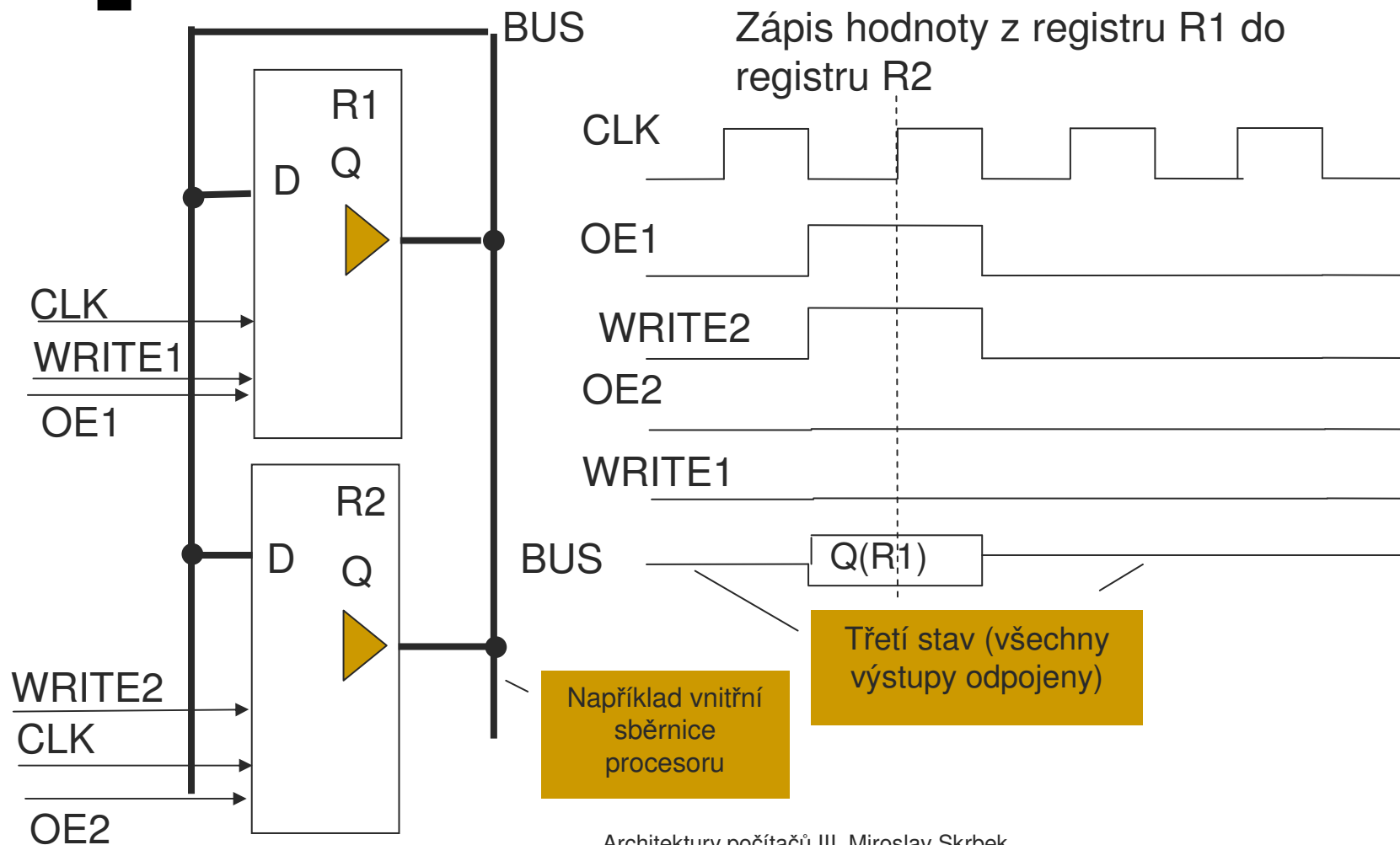
Jednička na signálu RESET nastaví Q do nuly bez ohledu na CLK a D. Používá se k inicializaci klopného obvodu.



Při vzestupné hraně signálu CLK se přepíše logická hodnota ze vstupu D na výstup Q. Mimo vzestupnou hranu na CLK se může D měnit libovolně a na výstup Q to nemá vliv. Obvod si tedy pamatuje poslední hodnotu na D zapsanou vzestupnou hranou CLK.

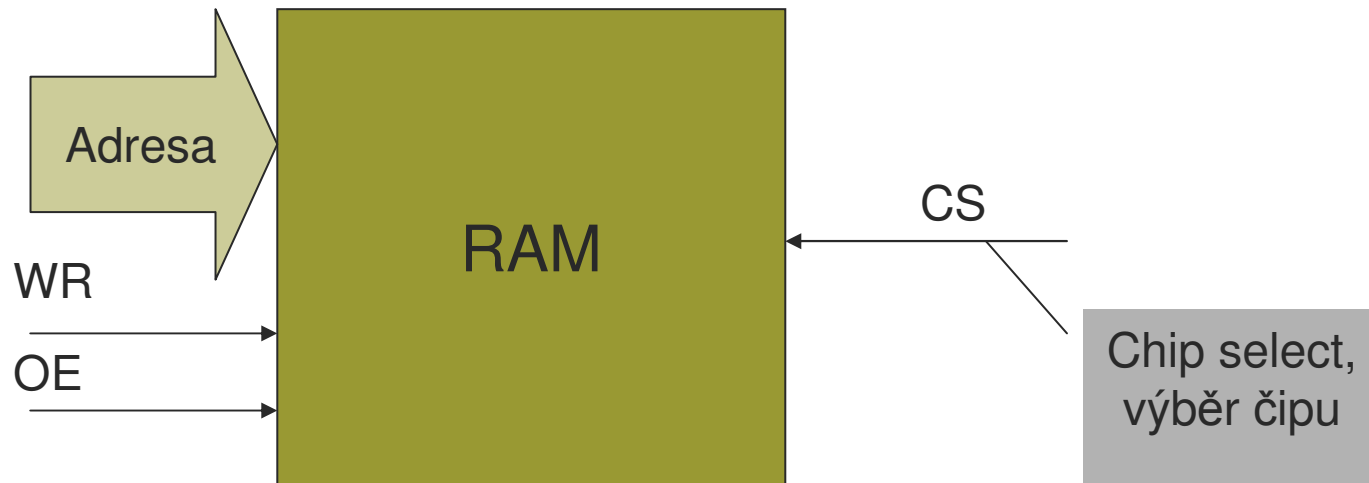


[Registr s třístavovým budičem]



[Paměť RAM]

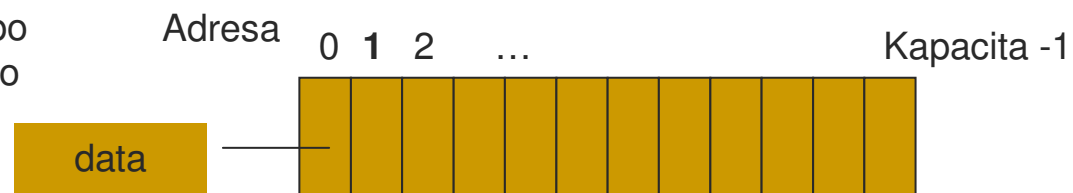
Signály OE (output enable, čtení) a WR (zápis, write)



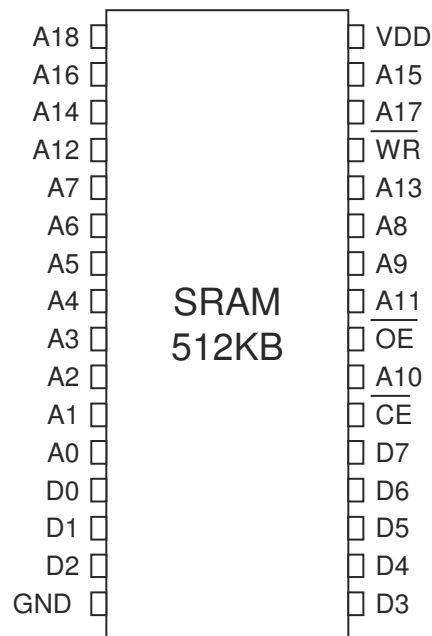
Šířka adresové sběrnice (v bitech) je dána kapacitou paměti v bytech (pokud je paměť adresovatelná po bytech)

Paměť může být adresovatelná po větších jednotkách (např. 32 nebo 64 bitů)

Paměť si představte jako pole paměťových prvků, které je indexované adresou.



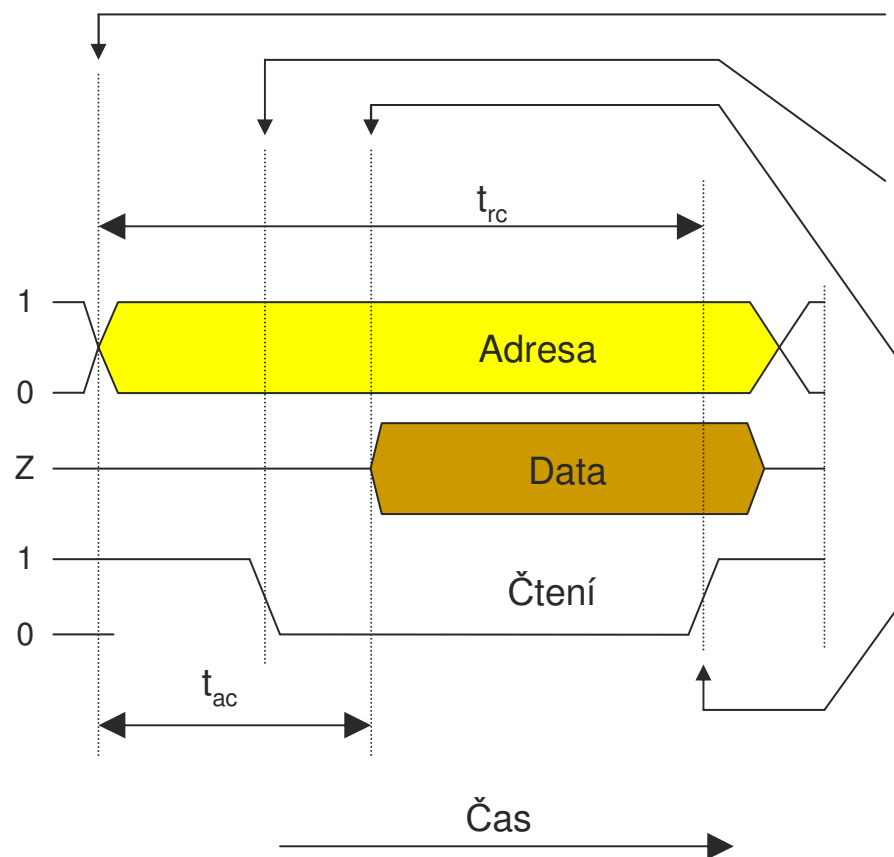
Paměťové obvody pro hlavní paměť- statická paměť RAM



- Statická RAM
- Použití v menších počítačových systémech (např. pro řídicí aplikace)
- Paměť neztratí zapsaná data dokud je připojeno napájení
- Kapacita např. 512KB
- Adresa (A0-A18)
- Data (D0-D7)
- Čtecí signál (OE)
- Zápisový signál (WR)
- Výběr čipu (CE)

Poznámka: uvedenou kapacitu paměti považujte za příklad. Existují paměti i s jinými kapacitami např. 128KB, 64KB, 32KB. U jiných kapacit se odpovídajícím způsobem mění počty adresových vodičů.

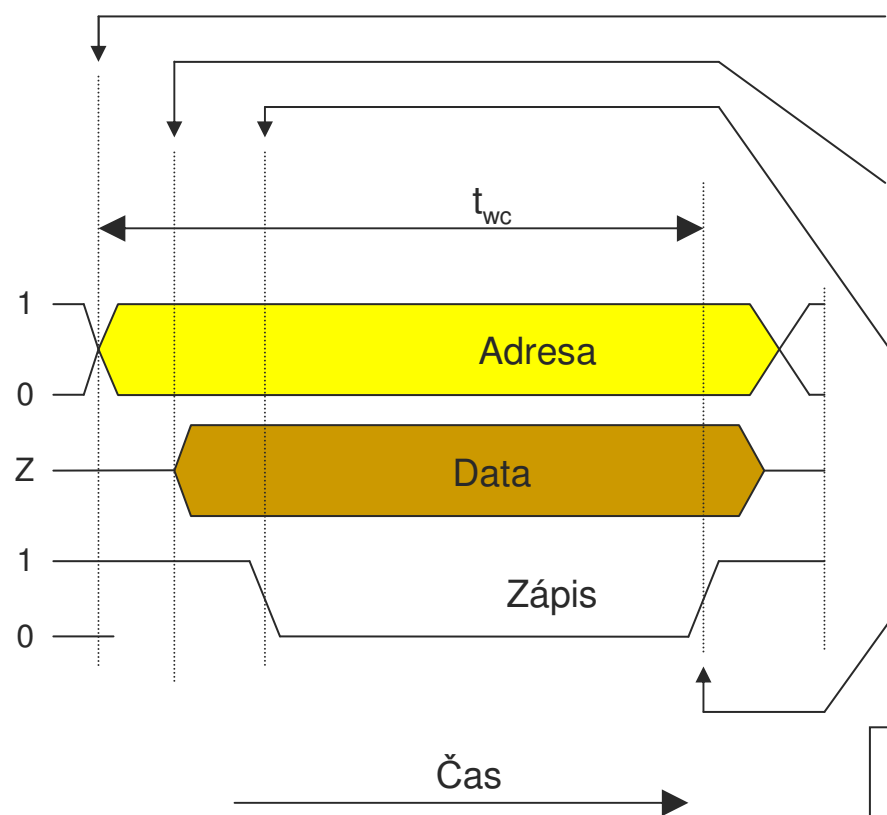
Čtení z paměti



- Vystavení adresy na adresovou sběrnici
- Aktivace čtecího impulsu
- Na datové sběrnici se objeví data
- Ukončení čtecího impulsu

t_{rc} – read cycle time
(celková přístupová doba do paměti)
 t_{ac} – přístupová doba od změny adresy

[Zápis do paměti]

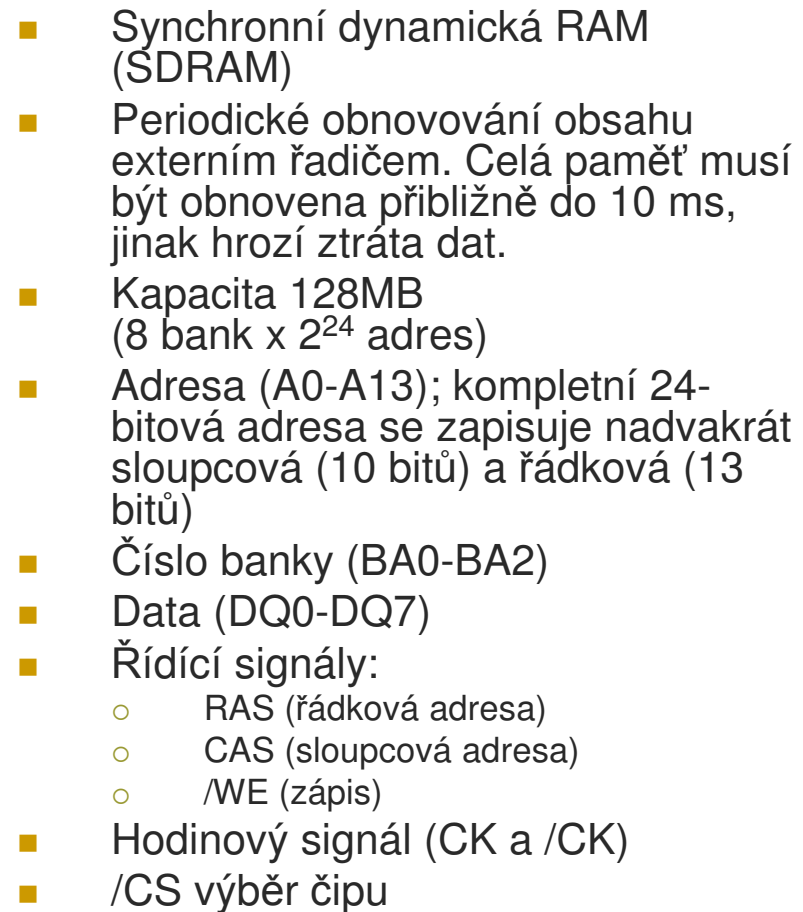


- Vystavení adresy na adresovou sběrnici
- Vystavení dat na datovou sběrnici
- Aktivace zápisového impulsu
- Ukončení zápisového impulsu

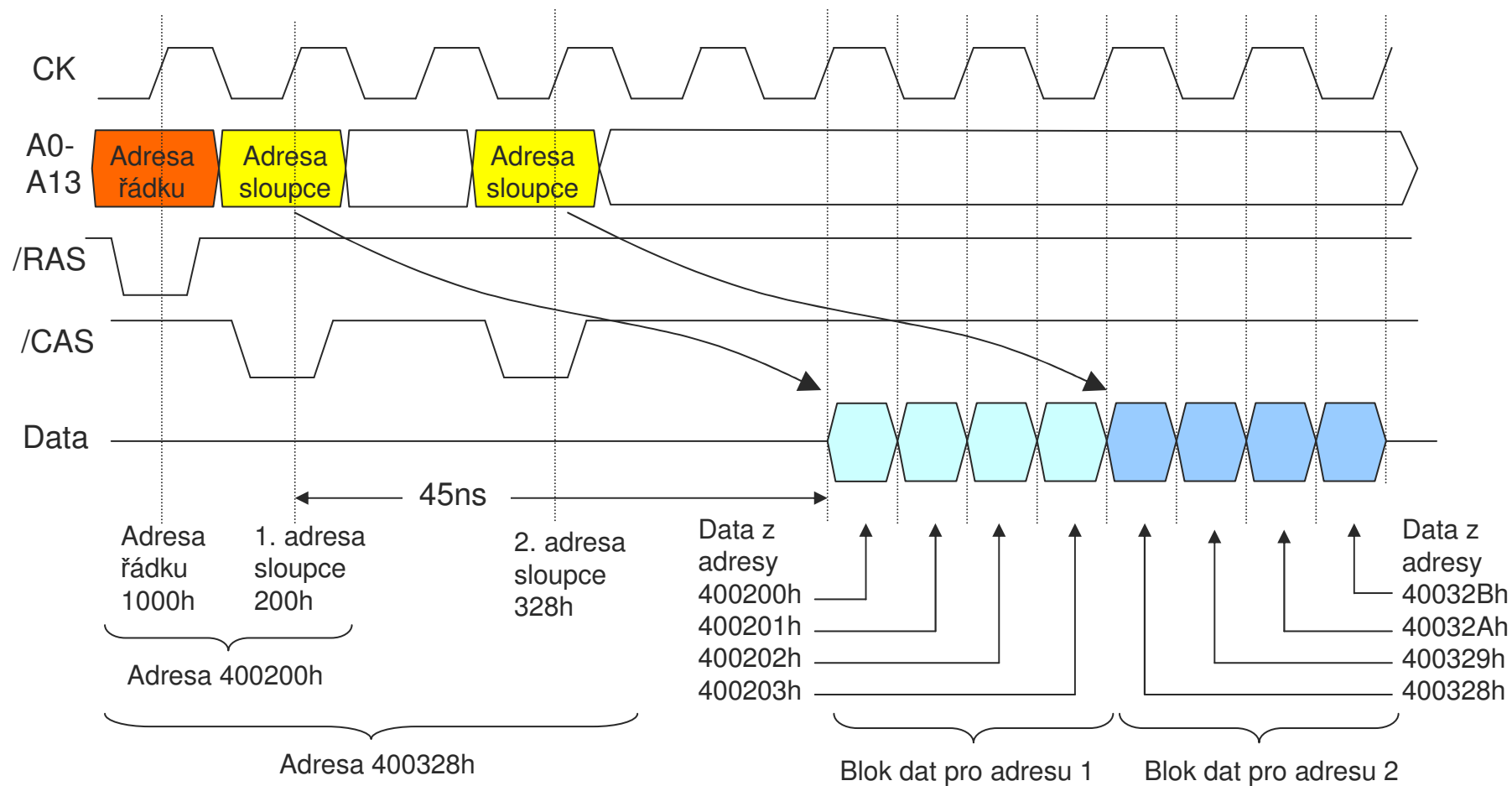
t_{wc} – write cycle time
(celková přístupová doba do paměti)

I

/ značí negovaný
signál

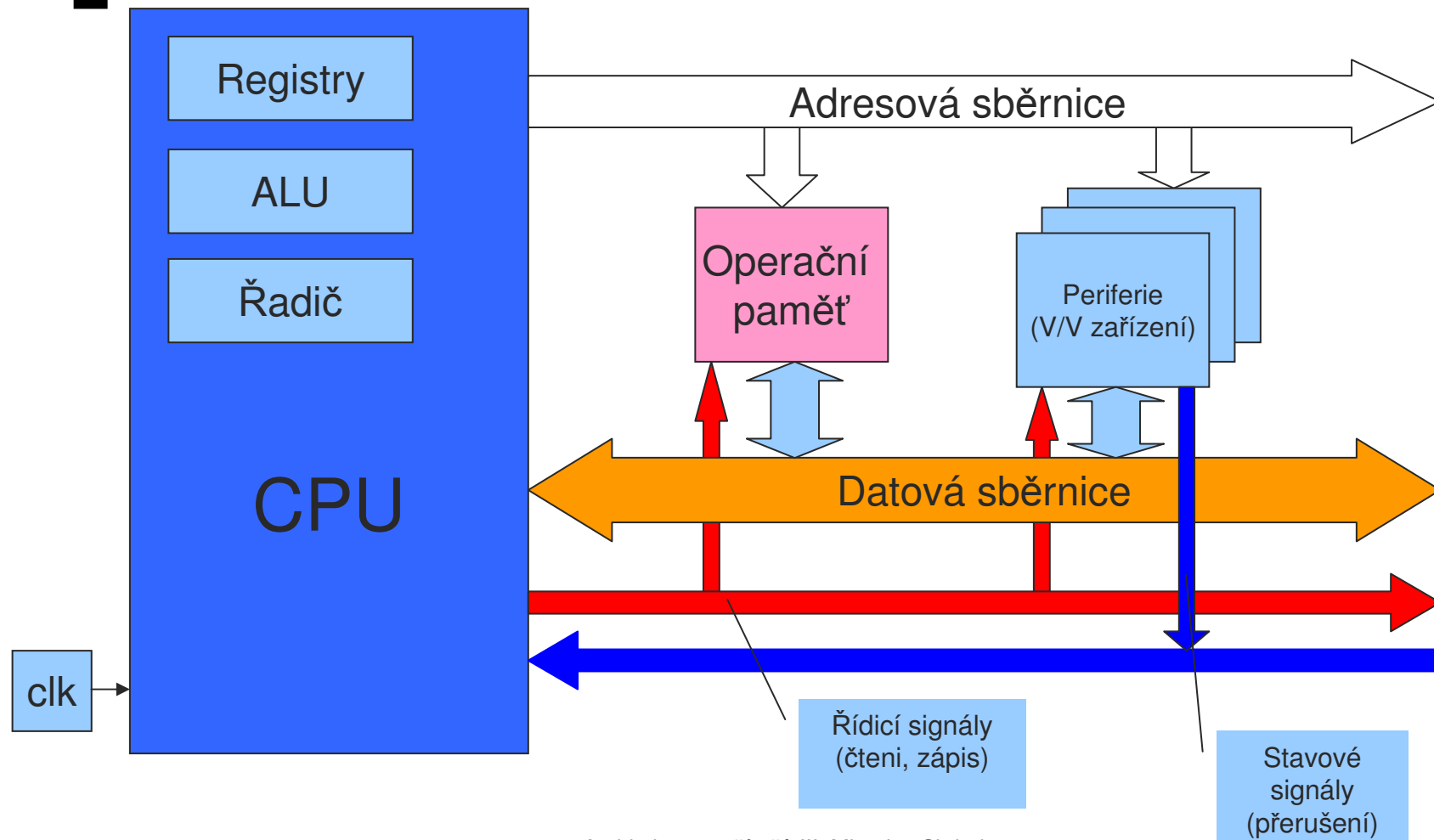


Čtení z paměti



[Navrhování procesorů]

Blokové schéma počítače



[Architektura CISC]

- Complex Instruction Set Computer (CISC) - počítač s rozsáhlým souborem instrukcí
- Instrukční sada obsahuje
 - Složité instrukce (např. kopírování bloku dat paměti) i jednoduché instrukce
 - Instrukcí je hodně
 - Typicky různá délka instrukcí (co do zakódování, tak i trvání)
- Pozitiva
 - Snížená četnost načítání instrukcí
 - Možnost vícenásobného využití funkčních jednotek v různých fázích vykonání instrukce
 - Přítomnost mikroprogramového řadiče dává možnost změnit instrukční repertoár
- Negativa
 - Složité instrukce jsou specializované, nutnost různých variant, aby skládačka byla úplná
 - Velký počet instrukcí => složitý dekodér instrukcí => dekodování jednoduchých a obvykle nejčtenějších instrukcí (např. sčítání) trvá dlouho
 - Nutnost mikroprogramovatelných řadičů
 - Instrukce typicky trvají různě dlouho, těžko se zavádí proudové zpracování

[Architektura RISC]

- Reduced Instruction Set Computer (RISC) – redukována instrukční sada
- Instrukce
 - Jen jednoduché
 - Typicky kódovány stejným počtem bitů
 - Typicky vykonány v jednom, nebo několika málo taktech
- Pozitiva
 - Jednoduchost – malé množství instrukcí
 - Jednoduchý dekodér instrukcí => rychlé dekódování instrukcí
 - Umožňuje proudové zpracování instrukcí
 - Rychlý obvodový řadič

[Zásadní otázka]

Kolik instrukcí musí minimálně mít počítač ?

Překvapivá odpověď na dalším
slajdu !

[Jednoinstrukční počítač]

```
subjmpifneg a, b, c ; mem(a) = mem(b) - mem(a), if (mem(a) < 0) goto c
```

Co dělá tato sada instrukcí ?

```
subjmpifneg tmp, tmp, pc+1 ; mem[tmp] = 0  
subjmpifneg a, tmp, pc+1 ; mem[a] = -mem[a]  
subjmpifneg a, b, pc+1 ; mem[a] = mem[b] - (- mem[a])
```

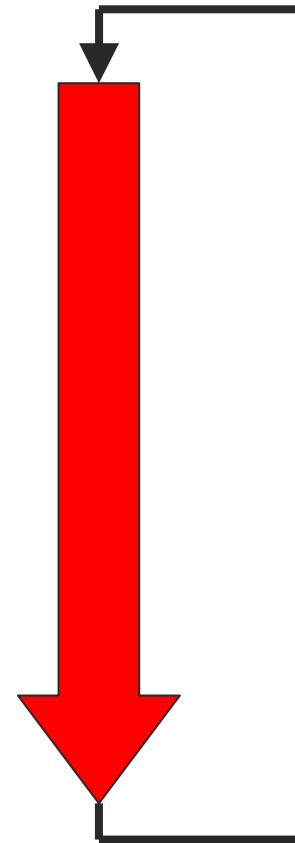
Jedná se o bezregistrový počítač s jednou aritmetickou operací typu paměť-paměť

Zpracováno dle:

Patterson A., Hennessy L.: Computer organization & design: the hardware/software interface. 2nd edition, Morgan Kaufmann Publishers, Inc., 1997. ISBN 1-55860-428-6, ISBN 1-55860-491-X

[Instrukční cyklus počítače]

- IF (Instruction Fetch)
– načtení instrukce
- ID (Instruction Decode)
– dekódování instrukce
- OF (Operand Fetch)
– načtení operandů
- EX (Execute)
– vykonání instrukce
- WB (Write Back)
– zapsání výsledku
- Interrupt detection
(test žádosti o přerušení)



[Řadič procesoru]

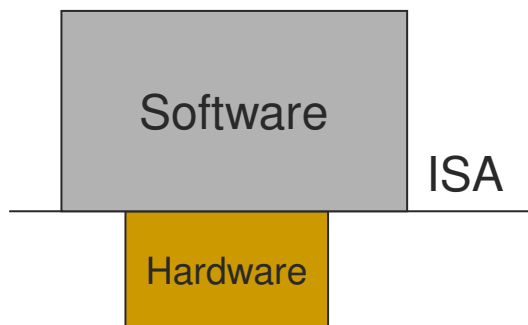
- Realizuje instrukční cyklus
- Řídí vykonávání dílčích operací v rámci instrukčního cyklu
- Generuje řídící signály
- Reaguje je stavové signály (příznaky aritmetických operací, vstup přerušení, apod.)
- Relizace
 - Obvodový řadič (konečný automat, D-klopné obvody + kombinační logika)
 - Mikroprogramovaný řadič
 - Realizuje složitější instrukce
 - Má paměť pro uložení mikroinstrukcí
 - Instrukce procesoru je realizována vykonáním sady mikroinstrukcí

[Instruction Set Architecture (ISA)]

Instruction Set (soubor instrukcí) tvoří rozhraní mezi hardware a software. Architektura souboru instrukcí má zásadní vliv na architekturu procesoru.

ISA zahrnuje

- registry procesoru
- instrukční sadu procesoru
 - datové typy
 - operace
 - specifikace operandů
 - adresní režimy
- kódování instrukcí do binární podoby
- adresní prostory
- výjimky



Příklad instrukcí

Instrukce x86

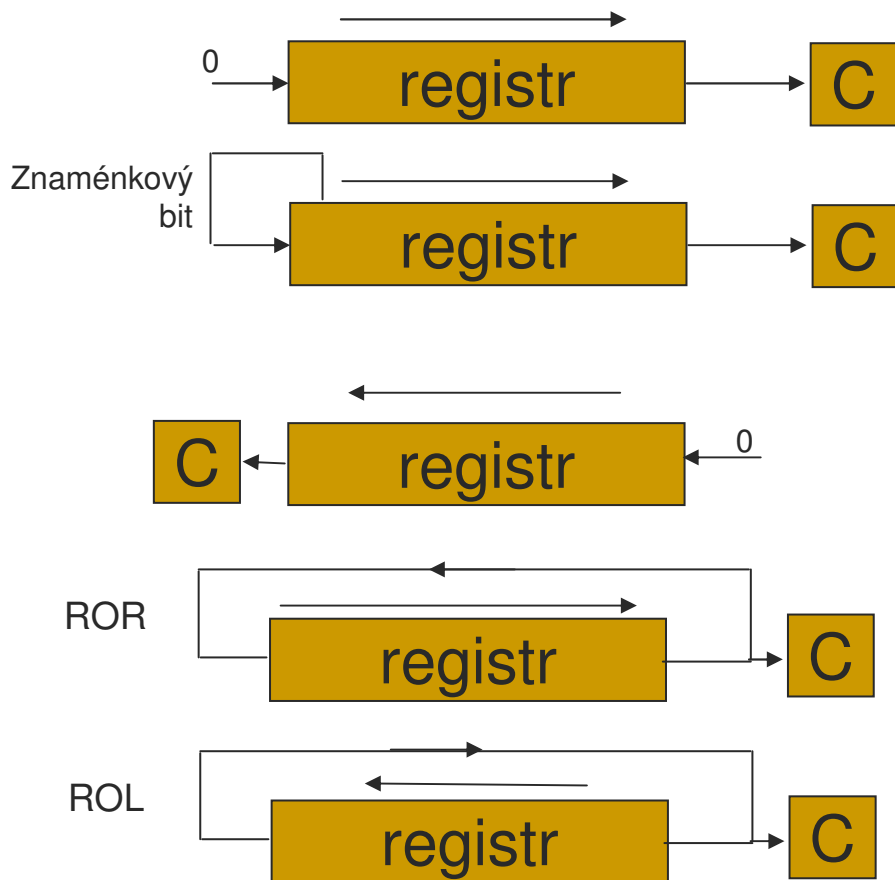
Adresa	Kód instrukce (hex)	Symbolický zápis instrukce
402c38:	89 04 24	mov %eax, (%esp)
402c3b:	e8 b0 06 01 00	call 0x4132f0
402c40:	3b 05 a0 11 42 00	cmp 0x4211a0, %eax
402c46:	0f 8e 24 fa ff ff	jle 0x402670
402c4c:	a3 a0 11 42 00	mov %eax, 0x4211a0
402c51:	e9 1a fa ff ff	jmp 0x402670
402c56:	8b 8d d4 fd ff ff	mov -0x22c(%ebp), %ecx
402c5c:	8b 41 70	mov 0x70(%ecx), %eax
402c5f:	89 04 24	mov %eax, (%esp)
402c62:	e8 79 75 01 00	call 0x41a1e0
402c67:	3b 05 40 11 42 00	cmp 0x421140, %eax
402c6d:	0f 8e ef f9 ff ff	jle 0x402662
402c73:	a3 40 11 42 00	mov %eax, 0x421140
402c78:	e9 e5 f9 ff ff	jmp 0x402662
402c7d:	8d 76 00	lea 0x0(%esi), %esi
402c80:	3d ff ff ff 00	cmp \$0xffffffff, %eax
402c85:	8b 15 70 10 42 00	mov 0x421070, %edx

získáno příkazem: `objdump -d /bin/ls`

[Typy instrukcí]

- Aritmetické: ADD (součet), SUB(rozdíl), MUL(násobení), DIV(dělení), CMP porovnání
- Logické: AND (log. součin), OR (log. součet), COM (negace, complement), XOR(excl. OR).
- Posuvy: SHL/SHR/ASR (posun vlevo, vpravo, aritmetický vpravo), ROL, ROR (rotace vlevo, vpravo), RLC, RRC (rotace vlevo, vpravo přes carry)
- Skokové instrukce: JMP (nepodmíněný skok), JZ, JC podmíněné skoky, CALL skok do podprogramu, RET, RETI návrat z podprogramu/přerušení.
- Přesuny MOV (přesun), XCH (exchange – výměna)

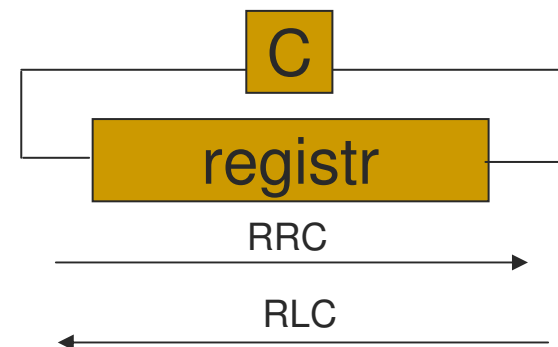
[Posuny a rotace]



SHR (shift right) posun všech bitů vpravo, zleva se nasouvá nula, bit vpravo, který opouští registr se ukládá do carry.

ASR (arithmetic shift right) posun všech bitů vpravo, zleva se nasouvá znaménkový bit (nejvyšší řád), bit vpravo, který opouští registr se ukládá do carry. Používá se pro posun čísel v doplňkovém kódu (dělení dvěma)

SHL (shift left) posun všech bitů vlevo, zprava se nasouvá nula, bit vlevo, který opouští registr se ukládá do carry



[Program status word (PSW)]

PSW je registr pro uložení příznaků a stavové informace procesoru.

Typické příznaky

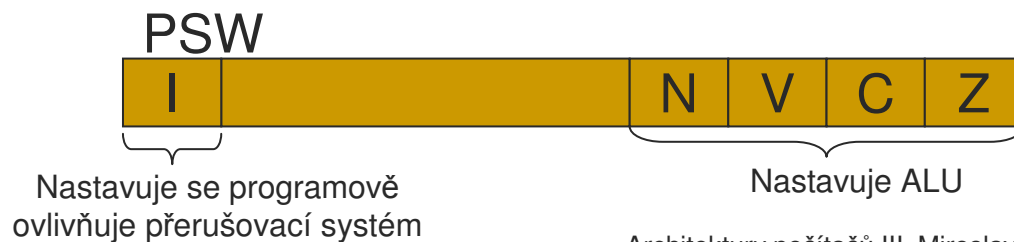
Z (zero) – výsledek operace je nulový

C (carry) – výsledek aritmetické operace způsobil přetečení z nejvyššího řádu, často se užívá také v posunech

V (overflow) – výsledek aritmetické operace v doplňkovém kódu je mimo rozsah (např. v osmibitovém registru $127+1$ nebo $-127-2$).

N (negative) – výsledek je záporný

I – příznak povolení přerušení (1 povoleno, 0 – zakázáno)



[Skokové instrukce I]

- Nepodmíněné (skok na adresu proveden vždy)
 - Přímé (adresa skoku je součástí instrukce)
 - Nepřímé (adresa skoku je v registru, na který se instrukce skoku odkazuje)
- Podmíněné (skok se provede pouze pokud je splněna podmínka, tj. testovaný bit v příznakovém registru má požadovanou hodnotu)
 - Podmínky (EQ – equal ($Z=1$), NEQ – nonequal ($Z=0$), GE – greater or equal ($C=0$ pro čísla bez znaménka, $N=0$ pro čísla se znaménkem), LT – less than ($C=1$ pro čísla bez znaménka, $N=1$ pro čísla se znaménkem), atd. Například BEQ addr, BNEQ addr, BGE addr, apod.
 - Přímé testování příznaků JZ addr, JNZ addr, JC addr, JNC addr, JN addr (negative), JP addr (positive) apod.

[Skokové instrukce II]

■ Skoky absolutní

- Instrukce nebo registr udává přímo adresu kam se skočí
- Realizováno jednoduše přiřazením
 $PC = \langle \text{cílová adresa skoku} \rangle$, např. $PC = 0x0010$

■ Skoky relativní

- Instrukce udává hodnotu, která se přičte k aktuální pozici PC a tak se vypočte cílová adresa skoku
- Relativní skok je nezávislý na přesunu programu v paměti počítače, tj. vypočítané cílové adresy dopadají do správných míst i po přesunu (\Rightarrow snadná relokace)
- Realizováno přiřazením $PC = PC + \langle \text{posunutí} \rangle$
- Posunutí automaticky počítají překladače, programátor se o to nemusí starat

[Příklad programu]

```
0000:    MOV R0, #10    ; naplní R0 hodnotou 10
0001:    MOV R1, #3     ; naplní R1 hodnotou 3
0002:    MOV R2, #1     ; naplní R2 hodnotou 1
0003:    MOV R3, #0     ; naplní R3 hodnotou 0
0004:    ADD R3, R1     ; R3 = R3 + R1
0005:    SUB R0, R2     ; R0 = R0 - R2
0006:    JNZ 0004       ; skok na adresu 4, pokud
                        ; je výsledek předchozí
                        ; operace nenulový
0007:    STOP          ; zastavení programu,
                        ; výsledek je v R3
```

R0,R1,R2 a R3 jsou registry procesoru. V tomto případě je levý operand cílový (a zdrojový současně).

[Zásobník]

- Procesory implementují zásobník
- Zásobník je uložen v paměti
- Jeden registr (SP-stack pointer) je vyčleněn jako ukazatel na vrchol zásobníku
- Zásobník roste shora dolů (x86) nebo zdola nahoru (některé mikrokontroléry)
- Na zásobník se primárně ukládají návratové adresy pro skoky do podprogramu (funkce v C je podprogramem)
- Na zásobník také ukládají parametry funkcí a odkládají obsahy registrů

[Podprogramy]

Instrukce **CALL** adr

```
ldi r16, 10  
ldi r17, 1  
ldi r18, 5  
ldi r19, 0  
call fu_add16  
...
```

Výsledek r17:r16
0x010F

```
ldi r16, 255  
ldi r17, 2  
ldi r18, 1  
ldi r19, 0  
call fu_add16  
...
```

Výsledek r17:r16
0x0300

fu_add16:

; podprogram

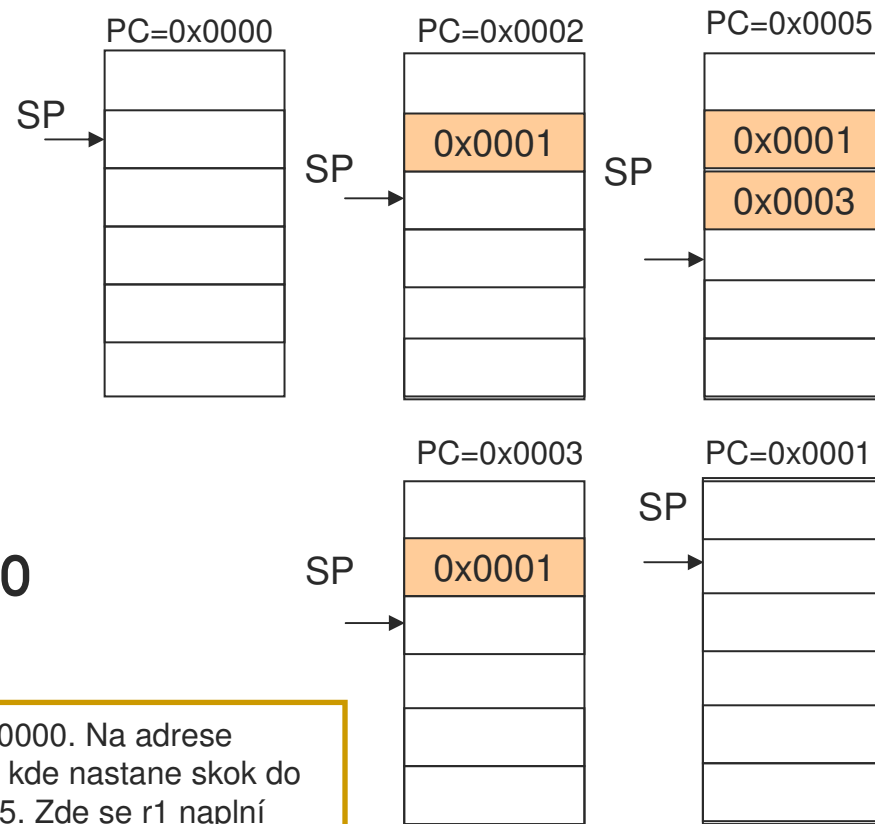
```
add r16, r18  
adc r17, r19  
ret
```

Návratová adresa se ukládá na
zásobník

Zásobník při volání podprogramu

Příklad programu

```
0000:  call 0002
0001:  jmp  0001
0002:  call 0005
0003:  add  r1, #5
0004:  ret
0005:  mov  r1, #10
0006:  ret
```

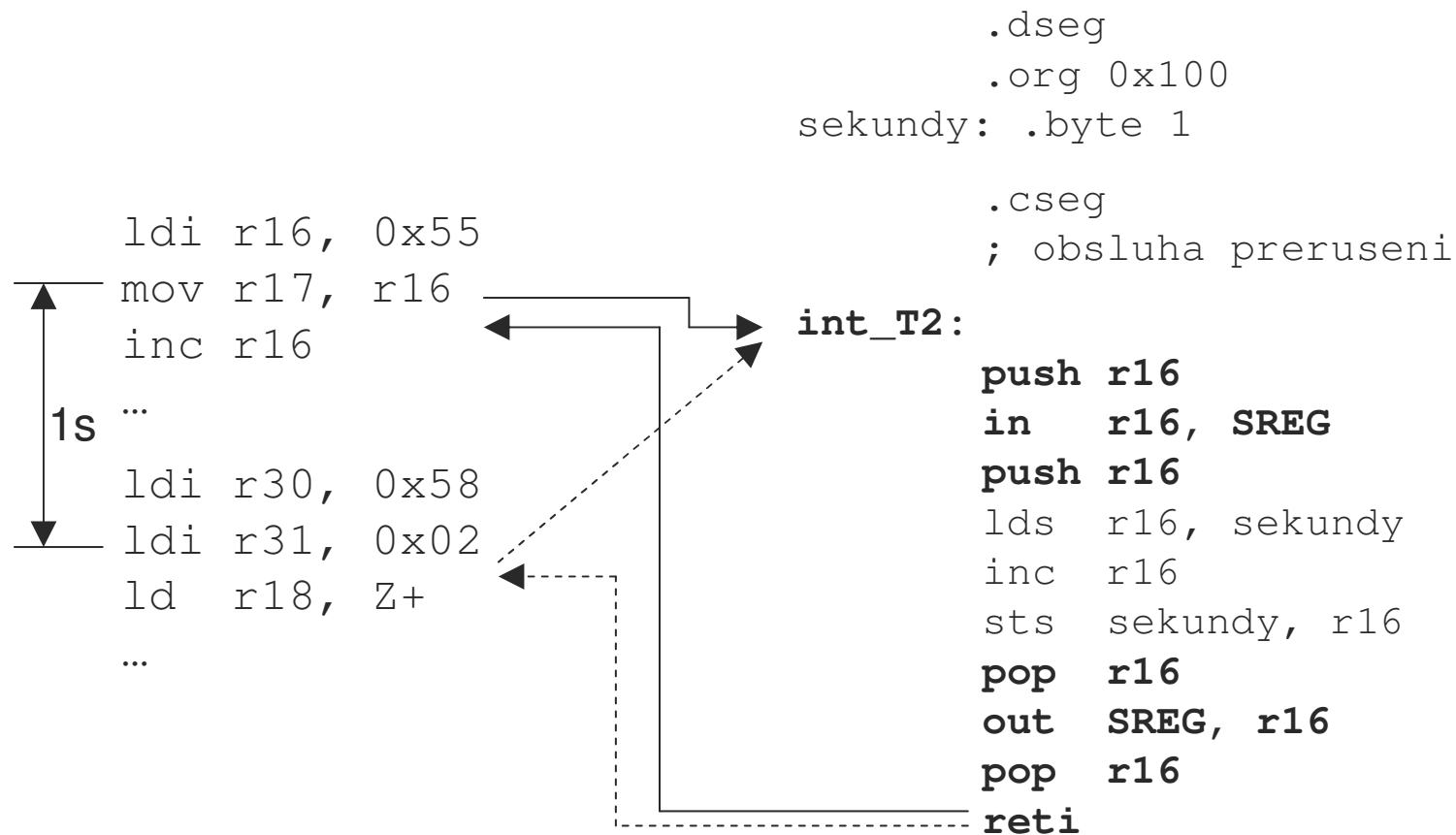


Po resetu se program začne provádět od adresy 0x0000. Na adrese 0x0000 je skok do podprogramu na adresu 0x0002, kde nastane skok do dalšího (vnořeného) podprogramu na adrese 0x0005. Zde se r1 naplní hodnotou 10. Po návratu z vnořeného podprogramu (RET na 0x0006), pokračuje program na adrese 00003h, kde se k r1 přičte 5. Po instrukci RET na adrese 0x0004, program pokračuje na adrese 0x0001, kde vstoupí do nekonečné smyčky.

[Přerušení]

- Vyvolání podprogramu vnější nebo vnitřní událostí
- Vnější přerušení jsou asynchronní (mohou přijít kdykoliv a v kterékoliv části prováděného programu)
- Každý procesor má typicky jeden vstup pro přerušení (pokud je více zdrojů přerušení, použije se řadič přerušení, který vybírá na základě priority)

[Přerušení (příklad)]



[Obsluhy přerušení]

Řadič přerušení, který vybral na základě priority mezi více aktivními žádostmi o přerušení informuje procesor o tom, který zdroj přerušení vybral. Na základě této informace určuje procesor adresu obslužného podprogramu pro přerušení.

Jsou dvě varianty:

1. Obslužné podprogramy pro konkrétní zdroje přerušení mají pevně stanovené adresy v paměti. Častý případ mikrokontrolérů (8051, AVR, ARM, ...).
2. V paměti je tabulka adres počátků obsluh přerušení a každému zdroji přerušení je určena jedna položka v této tabulce. Užito například u x86 (real mode) a dsPIC. Obdobou je IDT (interrupt descriptor table) - x86 v protected módu, Deskriptor obsahuje více informací, cílový segment a offset, privilege level apod.

Typické vlastnosti obsluhy přerušení

- Musí být transparentní (po návratu nesmí změnit hodnotu žádného registru procesoru, ani příznakového)
- Začátek obsahuje řadu instrukcí push, které ukládají registry použité v obsluze přerušení (v první řadě příznakového registru)
- Konec obsahuje řadu instrukcí pop, které obnovují stav registrů ze zásobníku
- Pro často se opakující přerušení musí obsahovat minimum kódu, jinak se dramaticky sníží výkon celého systému, případně dojde ke ztrátě některých žádostí o přerušení
- Končí instrukcí RETI, která je podobná instrukci RET, ale může integrovat další funkce například automatické obnovení příznaků (x86) nebo informovat řadič přerušení o dokončení obsluhy přerušení a pokyn k výběru dalšího zdroje přerušení (pokud je nějaká žádost aktivní) AVR, 8051.

Určení počátků obslužných rutin pro přerušení

- Pevné adresy – dány při návrhu procesoru, typicky na začátku paměti, např. Atmel AVR, Intel 8051, ARM
- Vektory přerušení – tabulka(y) v hlavní paměti, kde pro každý zdroj přerušení se nachází adresa počátku obsluhy přerušení. Položka je vyplněna softwarově před povolením příslušného přerušení. X86 v reálném módu, Microchip PIC24F.

[Vstupně/výstupní operace]

- Periferie se ovládají přes registry (např. IDE disk má 8+2 registry)
- Registry se mapují buď do vyhrazeného vstupně/výstupního (IO) prostoru (PC) nebo se mapují do paměťového prostoru (AVR)
- **Registry periférií se nesmí podléhat kešování.**
- U vstupně/výstupních registrů může mít význam
 - Zapsání hodnoty
 - Zapis bez ohledu na hodnotu
 - Čtení hodnoty
 - Čtení bez ohledu na přečtenou hodnotu
- Neplatí, co zapíšeš to také přečtu
- Hodnota zapsaná do registru ovlivňuje dále periférii

[Typy ISA]

- Střadačová (mikrokontroléry)
- Zásobníková (Java VM)
- Registrová
 - load-store (RISC procesory)
 - register-memory (x86)
 - memory-memory (výjimečná)

[Střadačová architektura]

- Má jeden registr (střadač), který je implicitně užíván aritmetickými a logickými operacemi
- Aritmetické a logické operace mají jako jeden operand střadač a druhý je čten z paměti. Výsledek je ukládán do střadače.
- Jednoduchý hardware
- Příliš mnoho operací přesunu (MOV) při vkládání operandu do střadače a uklízení výsledků ze střadače do paměti

[Střadačová ISA - příklad]

Střadač

A

Program counter

PC

Registr příznaků (Program Status Word)

C Z

Instrukce

MOV A, #const

MOV A, adr

MOV adr, A

ADD A, adr

SUB A, adr

SUBB A, adr

ADC A, adr

INC A

DEC A

INC adr

DEC adr

JZ adr

JNZ adr

JC adr

JNC adr

JMP adr

SHL

SHR

SAR

AND A, adr

OR A, adr

XOR A, adr

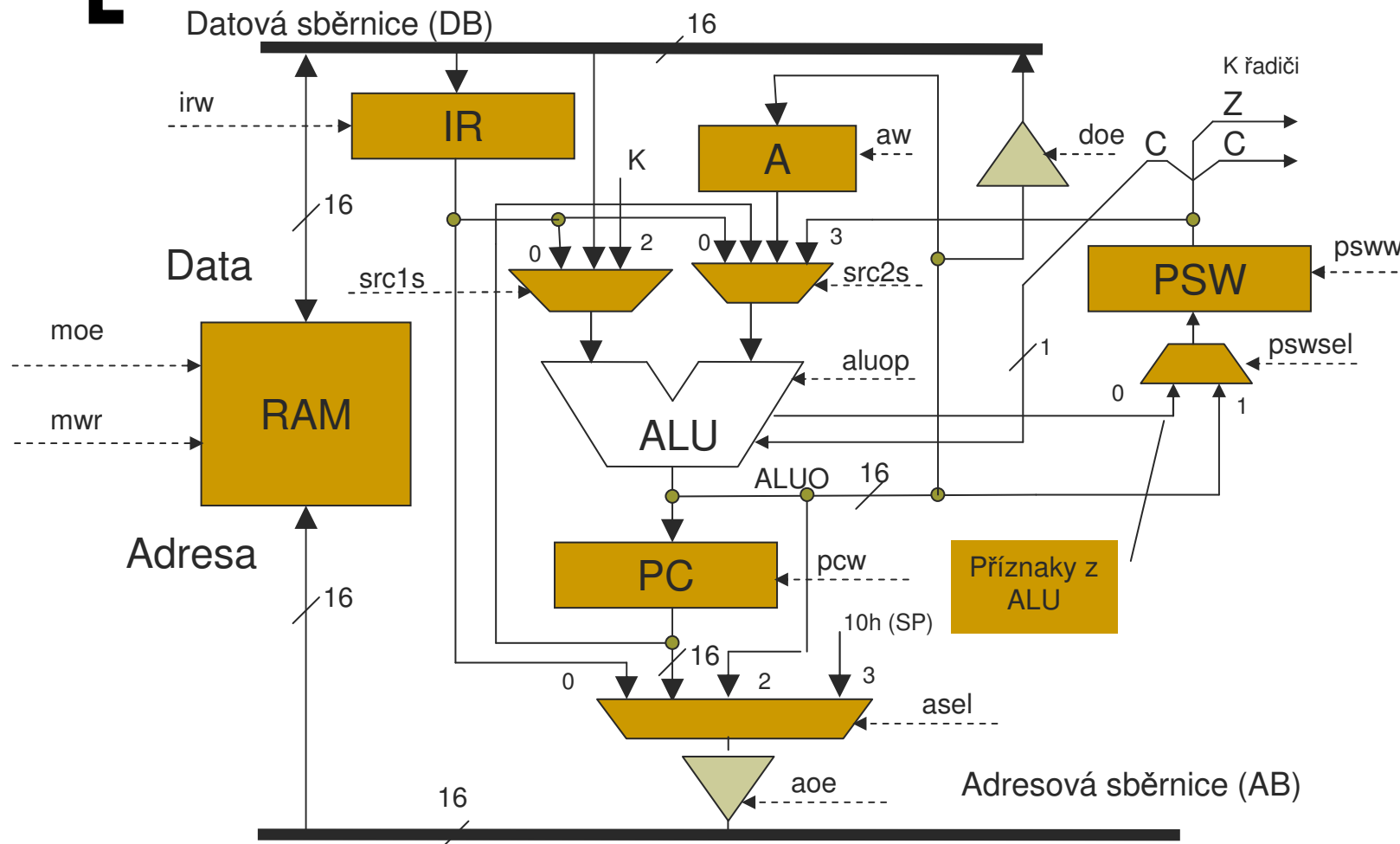
COM

CALL adr

RET

RETI

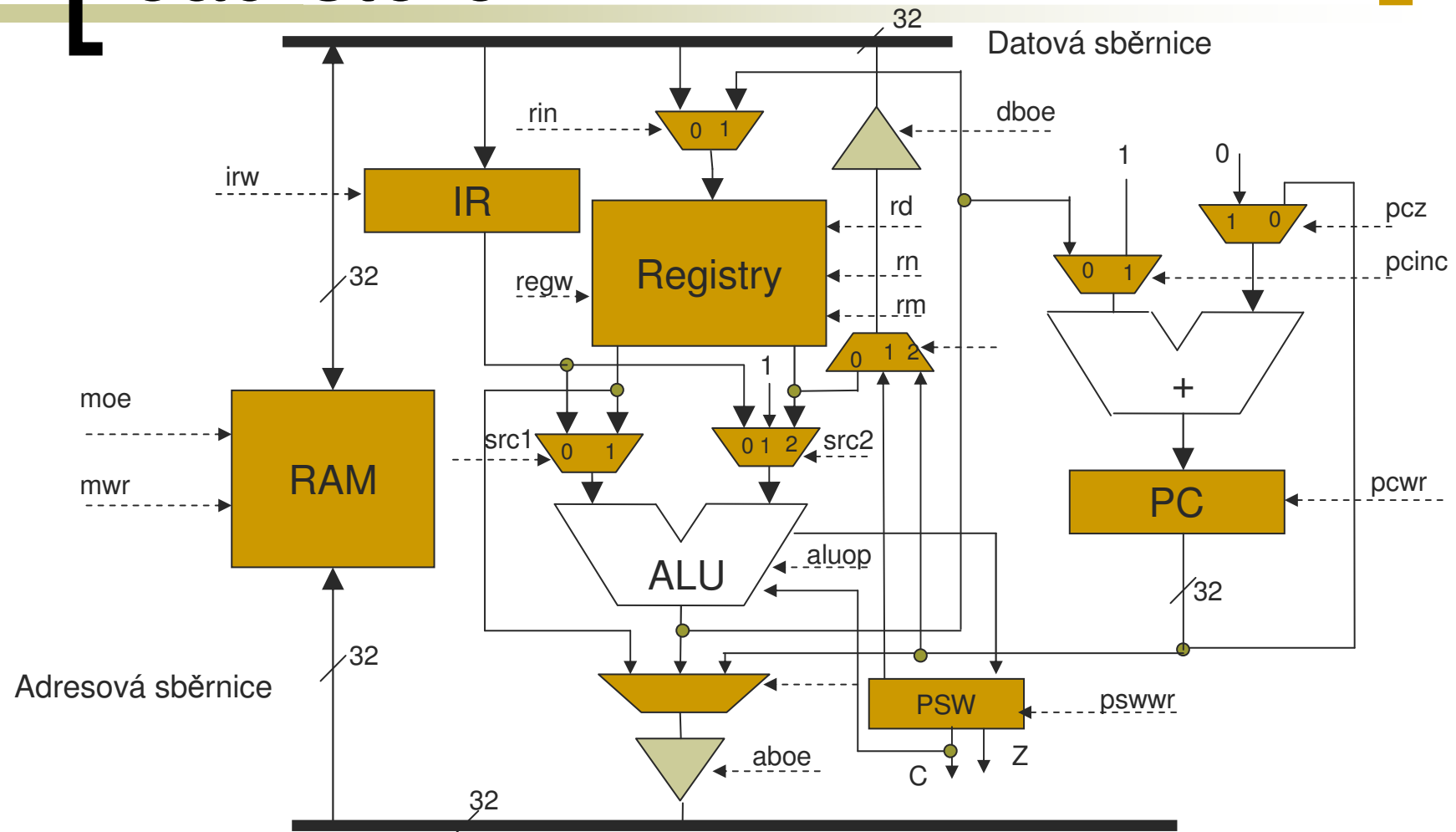
Střadačová architektura – datové cesty a řídicí signály



[Microkcode I



Registrová architektura typu load-store



[Registrová ISA (Load-Store) - příklad]

Registry

R0
R1
R2
R3

Program counter

PC

Registr příznaků (Program Status Word)

PSW	C	Z
-----	---	---

Instrukce

MOV Rd, #const

LD Rn, adr

ST adr, Rn

LD Rd, [Rn]

ST [Rn], Rm

ADD Rd, Rn, Rm

SUB Rd, Rn, Rm

SUBB Rd, Rn, Rm

ADC Rd, Rn, Rm

INC Rn

DEC Rn

JZ adr

JNZ adr

JC adr

JNC adr

JMP adr

SHL Rn

SHR Rn

SAR Rn

AND Rd, Rn, Rm

OR Rd, Rn, Rm

XOR Rd, Rn, Rm

COM Rn

CALL adr

RET

RETI

V Load-store architektuře jsou aritmetické operace prováděny pouze mezi registry.
Pro přístup do paměti se používají instrukce load (LD) nebo store (ST).

Registrová – příklady formátů instrukcí

tříoperandová instruce

ADD rd, rn, rm ; $rd \leftarrow rn + rm$

OZ (operační znak)

01100010	rn	rm	rd
----------	----	----	----

dvouoperandová instruce

ADD rd, rn ; $rd \leftarrow rd + rn$

OZ

01100010	rn	rd
----------	----	----

Rd značí cílový registr

Operační znak
odlišuje jednoznačně
instrukce od sebe !



Reálné příklady kódování instrukcí

Mikrokontrolér řady AVR (Atmel)

[Kódování instrukcí AVR I]

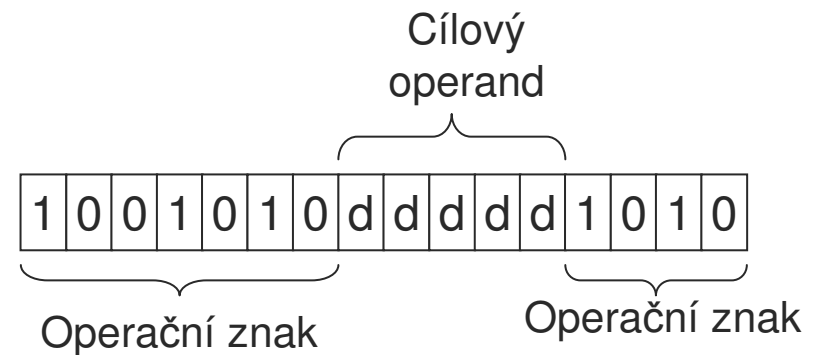
Symbolický zápis instrukce

dec r1

Cílový operand

Jméno
instrukce

Strojový kód instrukce *dec*



Příklad

dec r1

: 0x941A

[Kódování instrukcí AVR II]

Symbolický zápis instrukce

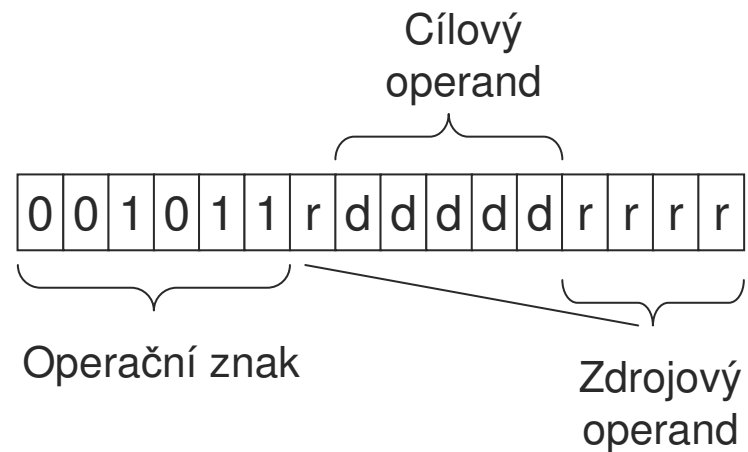
mov r1, r2

Zdrojový operand

Cílový operand

Jméno
instrukce

Strojový kód instrukce **mov**:



Příklad

mov r1, r2 : 0x2C12

Kódování instrukcí AVR III

Symbolický zápis instrukce

ldi r17, 0x54

Přímá konstanta

Cílový operand

Jméno instrukce

Strojový kód instrukce **ldi**:

Cílový operand
R(dddd + 16)

1	1	1	0	K	K	K	K	d	d	d	d	K	K	K	K
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Operační znak

Konstanta

Příklad

ldi R17, 0x54

: 0xE514

[Kódování instrukcí AVR IV]

Symbolický zápis instrukce

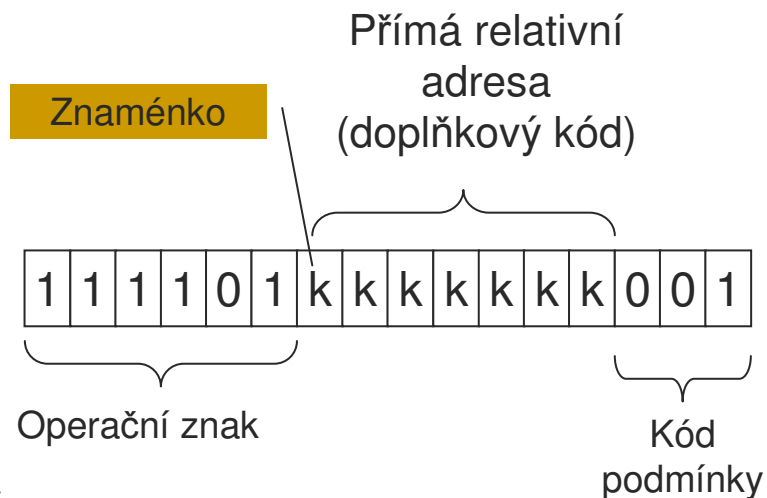
brne **PC-4**

**Cílová adresa
relativního skoku**

*V symbolickém vyjádření se adresa
relativního skoku udává absolutně
(`brne 0x100`, `brne PC+6`).
Překladač si pro vygenerování
instrukce relativní adresu vypočte.*

**Jméno
instrukce**

Strojový kód instrukce `brne`:



Příklady

<code>brne PC-4</code>	: 0xF7D9
<code>brne PC+0xA</code>	: 0xF449

Pozor ! Absolutní adresa skoku se počítá jako $PC+k+1$

[Kódování instrukcí AVR V]

Symbolický zápis instrukce

`jmp 0x125`

Cílová adresa skoku
(absolutní)

Jméno
instrukce

Strojový kód instrukce `jmp`:

Operační znak

Operační znak

1	0	0	1	0	1	0	k	k	k	k	k	1	1	0	k
k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k

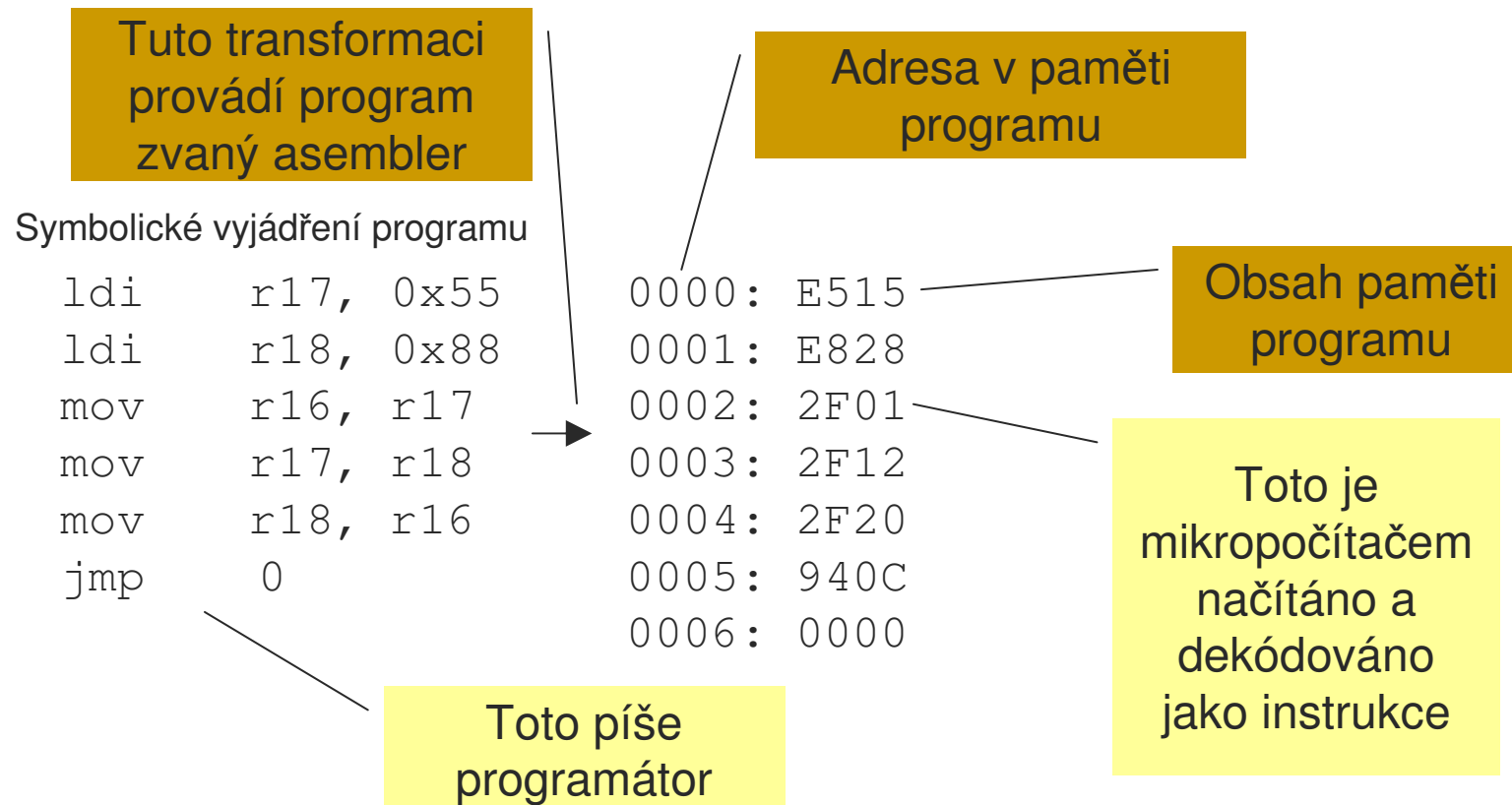
Přímá absolutní
adresa

Příklad

`jmp 0x125`

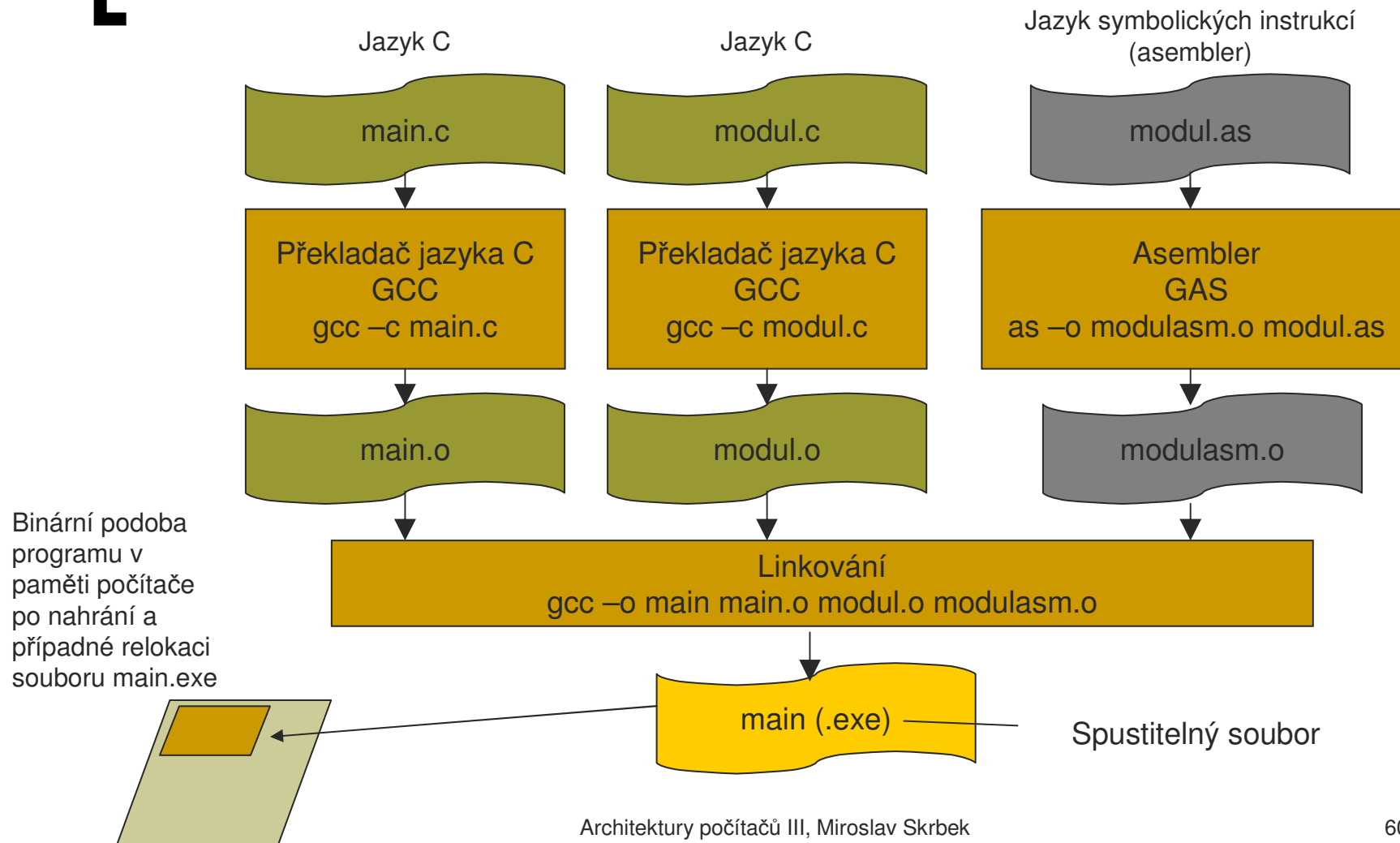
: **0x940C, 0x0125**

Příklad programu v symbolickém vyjádření a ve strojovém kódu



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

Překlad z vyšších programovacích jazyků



[Adresní módy]

- Registrový MOV R1, R2
- Přímá konstanta MOV R1, #100
- Přímá adresa LD R1, 100 nebo LD R1,[100]
- Nepřímá adresa MOV R1, [R2]
 - s postinkrementací LD R1,[R2+]
 - s preinkrementací LD R1,[+R2]
 - s postdekrementací LD R1,[R2-]
 - s predekrementací LD R1,[-R2]
- Nepřímá s posunutím LD R1, [R2+100]
- Nepřímá s indexací LD R1, [R2+R3+100]
- Nepřímá s indexací a měřítkem LD R1, [100+R2+R3*2]

[Registrová ISA (obecná) - příklad]

Aritmetické instrukce mohou mít jeden operand z paměti

Registry

R0
R1
R2
R3

Program counter

PC

Registr příznaků (Program Status Word)

PSW	C	Z
-----	---	---

Instrukce

MOV Rd, #const

MOV Rd, adr

MOV adr, Rd

MOV Rd, [Rn]

MOV [Rn], Rm

ADD Rd, Rn

ADD Rd, [Rn]

SUB Rd, Rn

SUB Rd, [Rn]

SUBB Rd, Rn

ADC Rd, Rn

INC Rd

DEC Rd

JZ adr

JNZ adr

JC adr

JNC adr

JMP adr

SHL Rd

SHR Rd

SAR Rd

AND Rd, Rn

OR Rd, Rn

XOR Rd, Rn

COM Rd

CALL adr

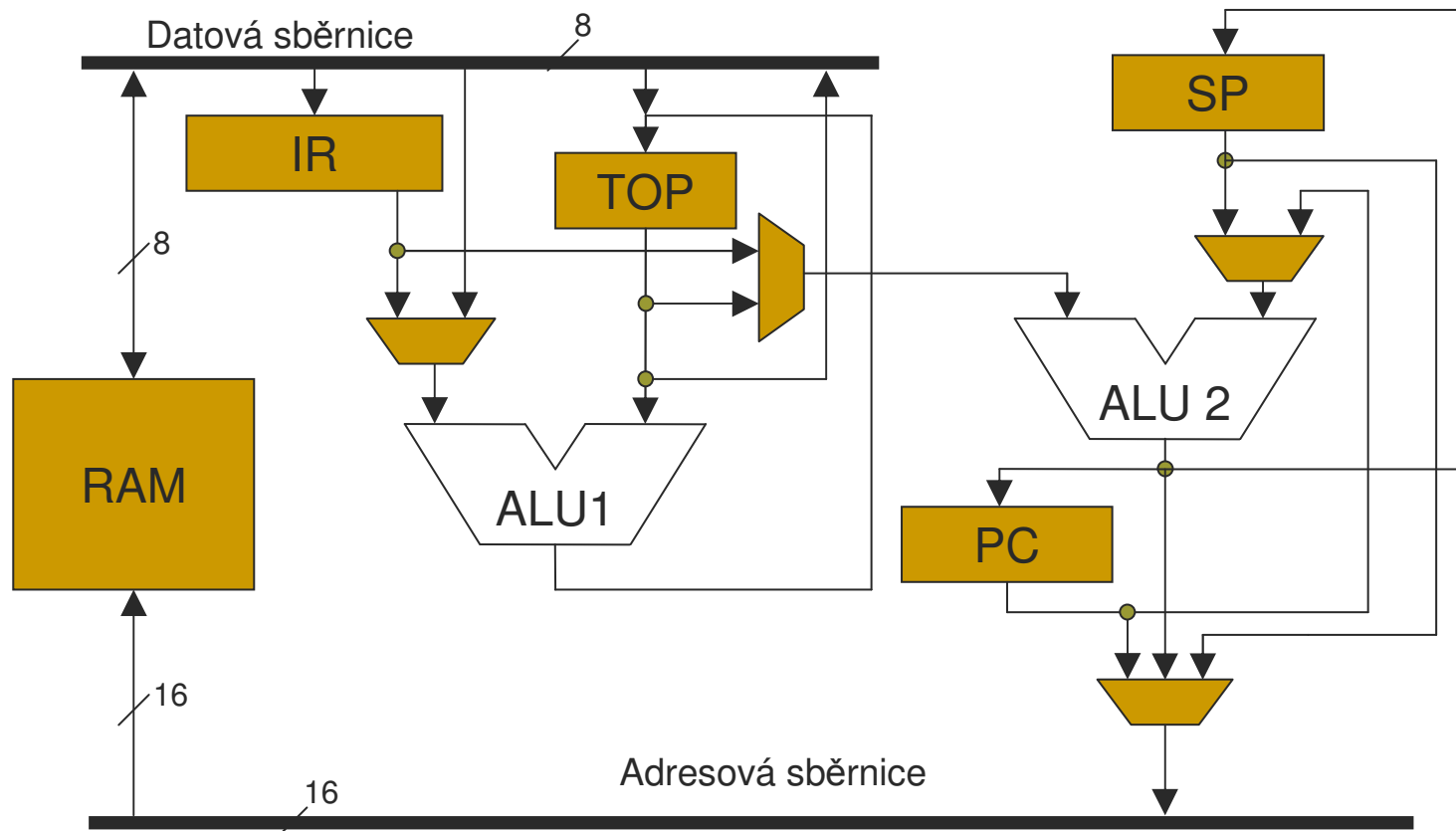
RET

RETI

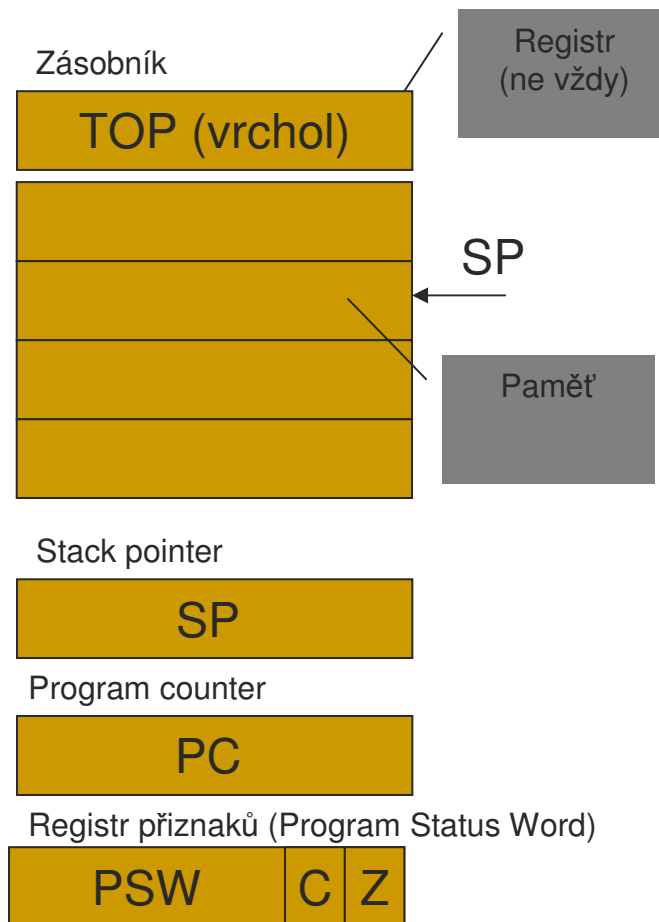
[Zásobníková architektura]

- Operandy aritmetických operací jsou na zásobníku (vrchol a položka pod vrcholem), operandy se odstraní a výsledek se uloží na zásobník
- Instrukce mají krátké kódování, u aritmetických operací stačí jen operační znak
- Skokové instrukce jsou stejné jako u jiných ISA
- Nevýhodou je příliš mnoho přesunů mezi pamětí a zásobníkem
- Zásobník lze hardwarově realizovat s omezenou hloubkou, proto je nutný mechanismus odkládání (spiling) do a znovu naplňování (filling) zásobníku z paměti.

Zásobníková architektura (zjednodušený příklad)



Zásobníková - příklad

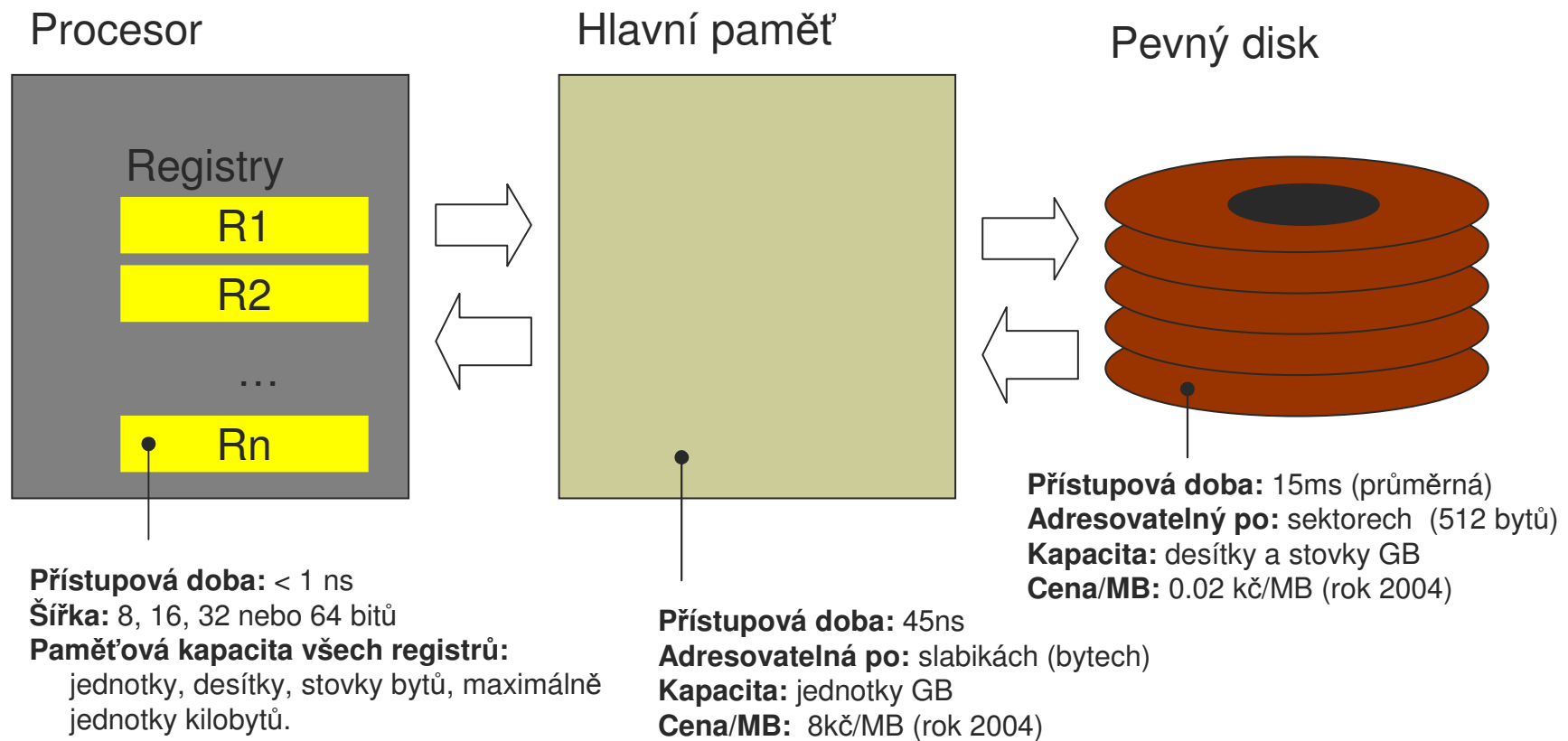


Instrukce

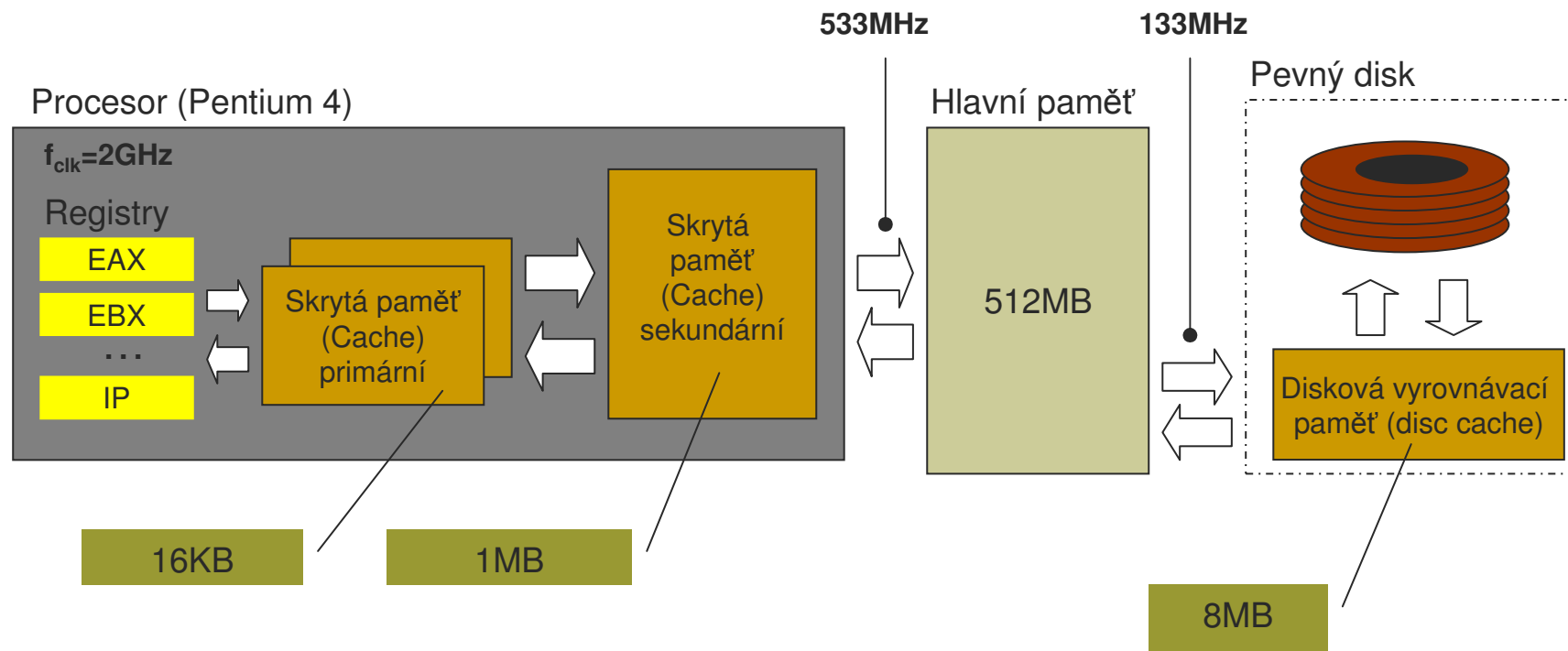
PUSH #imm
 POP
 LOAD
 LOAD offset
 ADD
 SUB
 MUL
 SHL
 SHR
 AND
 OR
 XOR
 COM

JZ adr
 JNZ adr
 JC adr
 JNC adr
 JMP adr
 CALL adr
 RET
 RETI

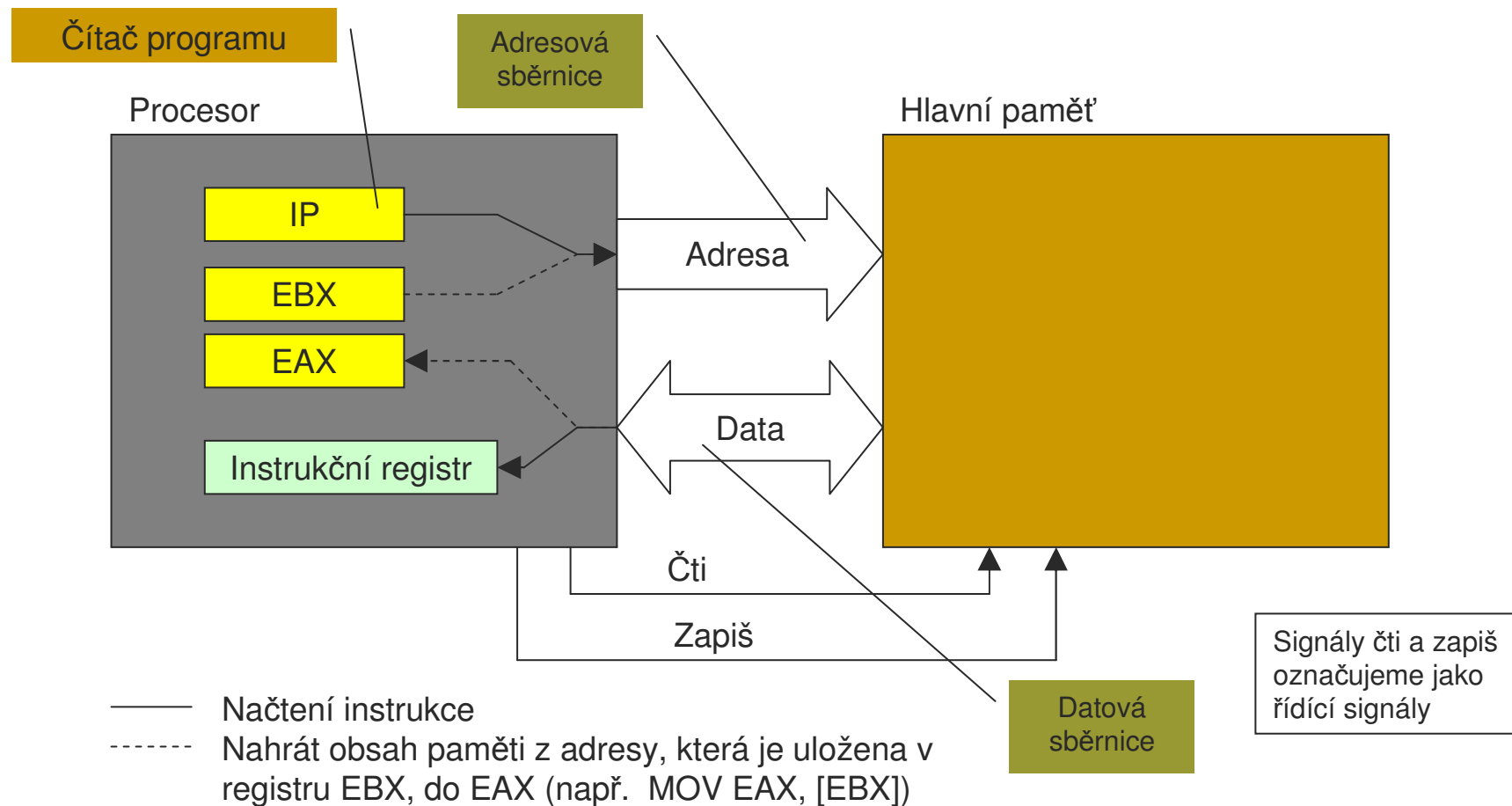
Hierarchie paměťového podsystemu



Příklad: paměťový podsystem počítače PC



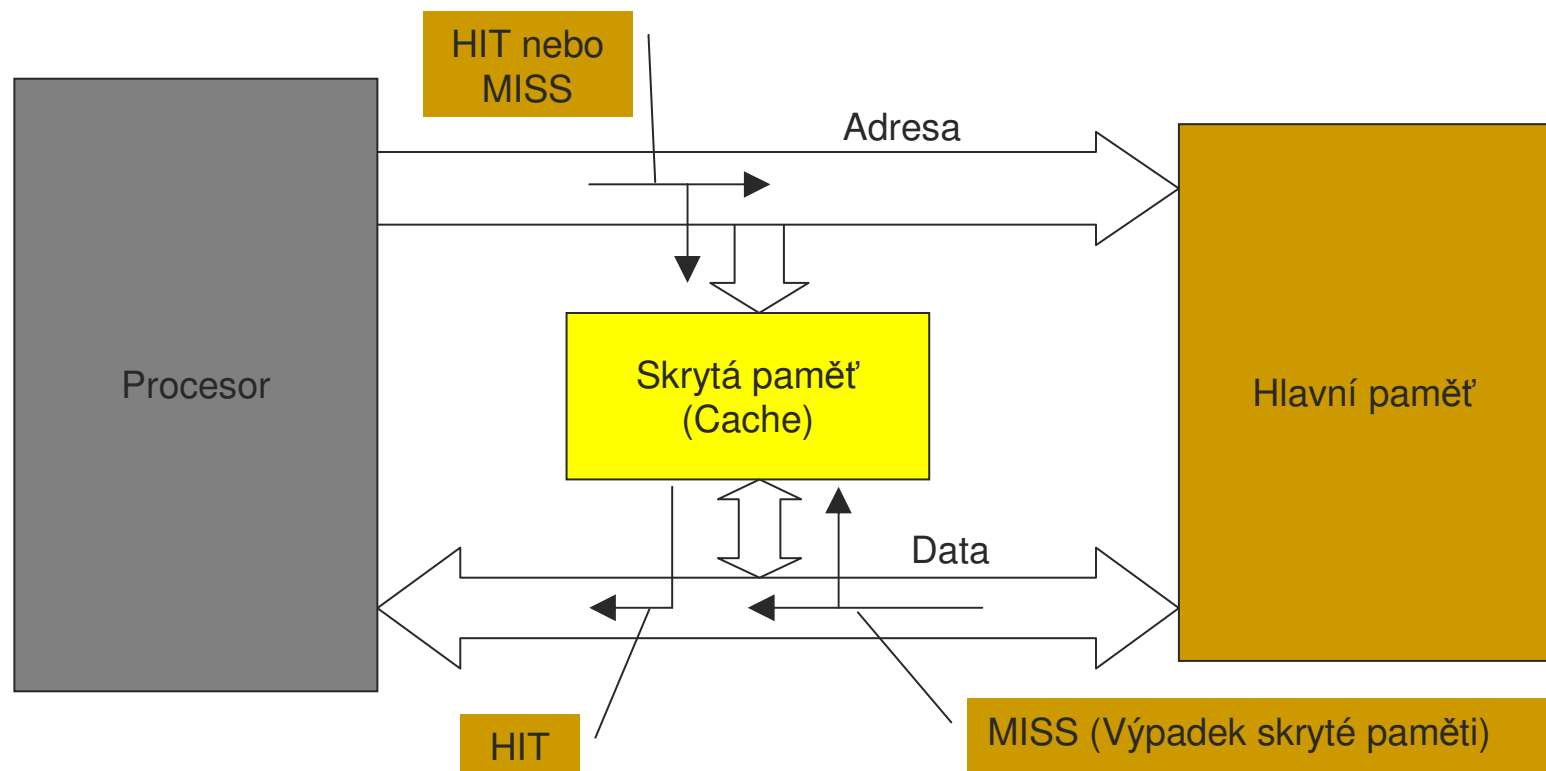
Výměna dat mezi procesorem a hlavní pamětí



[Skrytá paměť (Cache)]

- Slouží ke zkrácení přístupové doby hlavní paměti
- Přístupová doba skryté paměti je podstatně kratší než pro hlavní paměť
- Kapacita skryté paměti je podstatně menší než kapacita hlavní paměti
- První přístup k datům je dán přístupovou dobou hlavní paměti, ale každý další přístup k též datům (pokud nebyla vřazena ze skryté paměti) je dán přístupovou dobou do skryté paměti
- Skrytá paměť je transparentní (z funkčního hlediska procesor nepozná, jestli je skrytá paměť přítomna nebo ne)

[Funkce skryté paměti]



HIT – data na adrese generované procesorem jsou ve skryté paměti (kratší přístupová doba)

MISS - data na adrese generované procesorem nejsou ve skryté paměti a musí se načíst z hlavní paměti (delší přístupová doba)

Příklad

Bez skryté paměti

```
MOV  AX, [1030h] { 45ns}
ADD  AX, [1101h] { 45ns}
ADD  AX, [1101h] { 45ns}
ADD  AX, [1101h] { 45ns}
ADD  AX, [1101h] { 45ns}
-----
{ 225ns}
```

Se skrytou pamětí

```
MOV  AX, [1030h] { 45ns}
ADD  AX, [1101h] { 45ns}
ADD  AX, [1101h] { 10ns}
ADD  AX, [1101h] { 10ns}
ADD  AX, [1101h] { 10ns}
-----
{ 120ns}
```

Celkový čas potřebný
pro komunikaci s
hlavní pamětí

Se skrytou pamětí v tomto případě ušetříme 105ns

V našem příkladu je přístupová doba do hlavní paměti 45ns a přístupová doba do skryté paměti 10ns. Dále předpokládáme, že skrytá paměť je před provedením fragmentu programu prázdná.

Instrukce ovlivňující skrytou paměť

Protože se skrytá paměť se chová transparentně, není důvod zavádět speciální instrukce pro práci s ní.

Existují následující výjimky:

<code>PREFETCH adr</code>	– nahraje obsah hlavní paměti z adresy <code>adr</code> do skryté paměti
<code>CLFLUSH adr</code>	– zneplatní data z adresy <code>adr</code> ve skryté paměti (následuje-li čtení z <code>adr</code> , pak se vždy data načtou z hlavní paměti)
<code>INVD</code>	– zneplatni celý obsah skryté paměti
<code>WBINVD</code>	– zapiš obsah skryté paměti do hlavní paměti a zneplatni celý obsah skryté paměti

`PREFETCH` a `CLFLUSH` se používají při zpracování větších objemů dat. Dává se jimi paměťovému systému předem na vědomí, že určitá data budou brzy potřeba a naopak (tj. nebudou již potřeba).

[Adresní prostor]

- Souvislý rozsah adres generovaný procesorem pro přístup k paměti
- Velikost dána počtem adresních bitů
- Velikost je vždy mocnina dvou
- Nejmenší adresovatelná datová jednotka v adresním prostoru může být:
 - 1 bit (ve speciálních případech)
 - 1 slabika (1 byte), nejčastější (PC)
 - 1 slovo 16-bitů, 32-bitů, 64-bitů atp.
- Do fyzického adresního prostoru se mapuje fyzická paměť (tj. paměť v paměťových čípech)
- Adresní prostor nemusí být vždy celý vyplněn fyzickou pamětí
- Pokud procesor podporuje více adresních prostorů, neznamená to, že každý prostor bude mít svou vlastní adresovou sběrnici. Adresní prostory obvykle sdílí jednu adresovou sběrnici a jednotlivé adresní prostory jsou odlišeny oddělenými čtecími a zápisovými signály



Příklady adresních prostorů

- **Adresní prostor programu**

Z tohoto adresního prostoru čte procesor instrukce. Často je v něm povolena jen operace čtení. Počet bitů čítače programu musí korespondovat s velikostí tohoto prostoru.

- **Adresní prostor dat**

Do tohoto prostoru se mapují paměti RAM pro dočasné ukládání dat. Vždy jsou povoleny obě operace, tj. čtení i zápis. Maximum z počtu bitů v registrech, které jsou určeny nepřímou adresací v tomto adresním prostoru a počtu bitů přiděleným přímým adresám v instrukcích čtení a zápisu dat, musí korespondovat s velikostí tohoto adresního prostoru.

- **Vstupně/výstupní adresní prostor**

Do tohoto prostoru se mapují registry periférií. Tento prostor nebývá příliš velký např. 64K adres u procesorů x86. Vždy jsou povoleny obě operace čtení i zápis. V tomto prostoru často neplatí základní podmínka pro paměť tj. $\text{write}(\text{adr}, v1), v2=\text{read}(\text{adr}); v1 == v2$. Procesory, které nemají oddělen vstupně/výstupní adresní prostor, mapují registry periférií do adresního prostoru dat.

- **Fyzický adresní prostor**

Do tohoto adresního prostoru se přímo mapují paměťové čipy. Velikost tohoto prostoru je určena šířkou adresové sběrnice procesoru (tj. počtem adresových vodičů mezi procesorem a hlavní pamětí). U procesorů, které nemají virtuální adresní prostory, jsou adresní prostory programu a dat současně fyzickými adresními prostory. V takovém případě mluvíme jen adresních prostorech programu a dat a jejich fyzičnost již nezdůrazňujeme, protože se to automaticky předpokládá.

- **Virtuální adresní prostor**

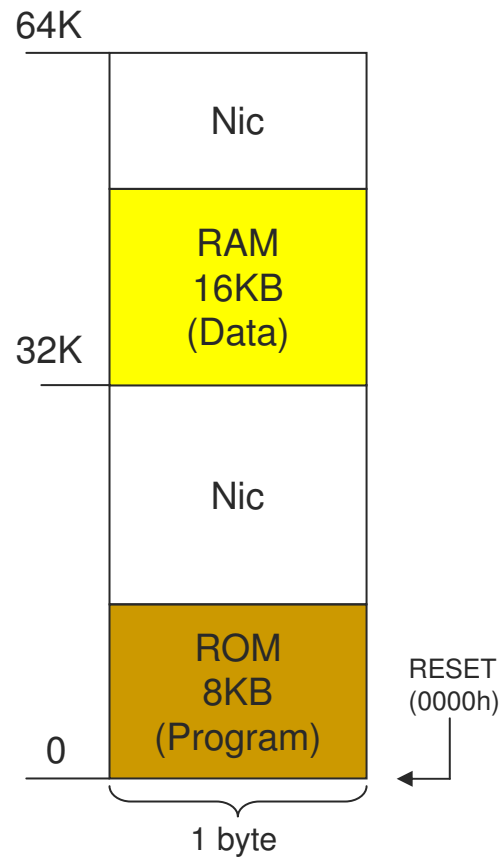
Je to logický (opak fyzického) adresní prostor přiřazený např. jednomu procesu (programu). Do tohoto prostoru se pak přes soustavu tabulek mapuje fyzická paměť z fyzického adresního prostoru. Součet velikostí virtuálních adresních prostorů všech současně běžících procesů mnohonásobně převyšuje velikost fyzického adresního prostoru a tudíž i instalované fyzické paměti. Součet fyzické paměti namapované v daném okamžiku ve všech virtuálních adresních prostorech musí být menší nebo roven celkové velikosti instalované fyzické paměti a velikosti odkládacího prostoru na disku (swap file nebo swap partition).

Mapa adresního prostoru (mapa paměti)

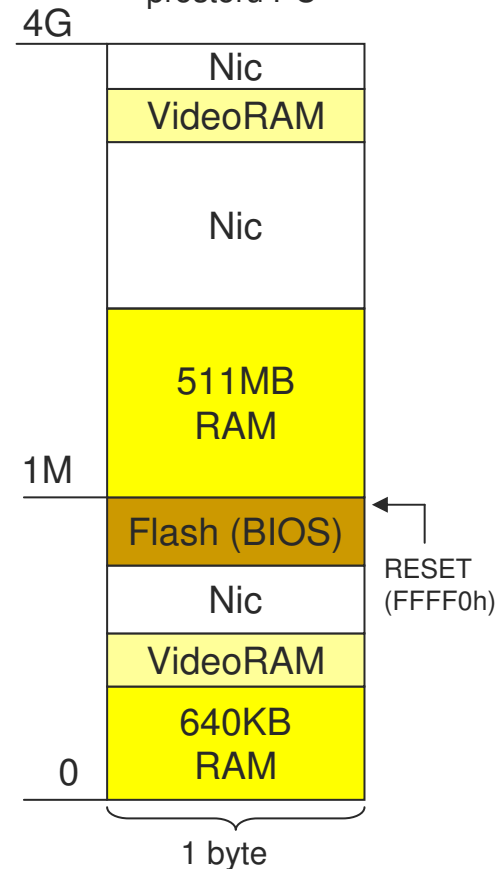
- Popisuje obsazení adresního prostoru fyzickou pamětí
- Mapa je často nesouvislá (neobsazené oblasti mezi obsazenými)
- Střídají se paměti různých typů (např. RAM, ROM)
- Mapa je jednou ze základních informací, kterou musí programátor v assembleru nastudovat, aby věděl, na které adresy umístit program, proměnné a zásobník. Znalost map adresních prostorů je nezbytná pro programování na úrovni operačního systému a driverů
- O adresních prostorech se dočteme v dokumentaci procesoru, ale mapu adresních prostorů musíme hledat v dokumentaci počítače, protože mapa adresního prostoru závisí na konkrétním zapojení paměťových čipů v paměťovém subsystému.

Příklady map adresních prostorů

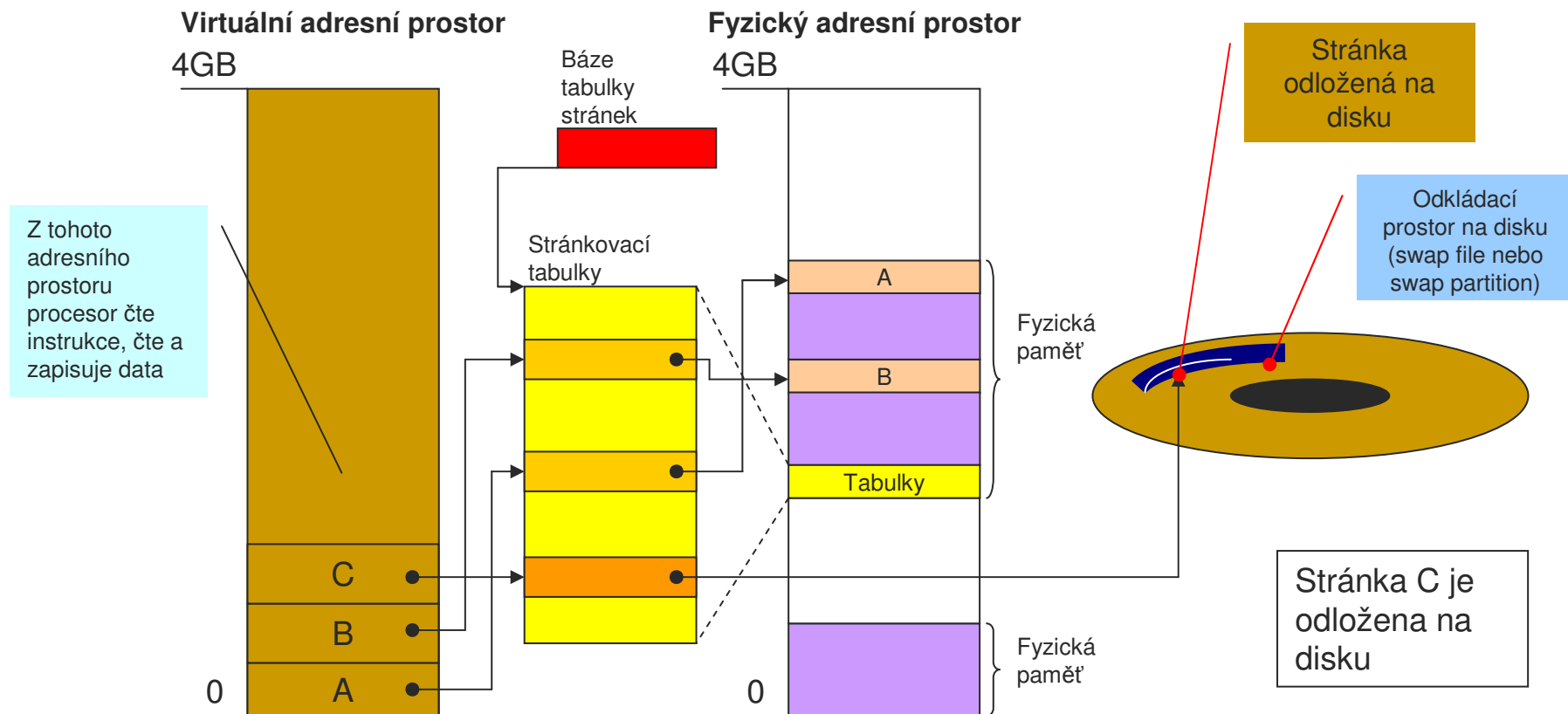
Mapa adresního prostoru
jednoduchého mikropočítače



Mapa fyzického adresního
prostoru PC

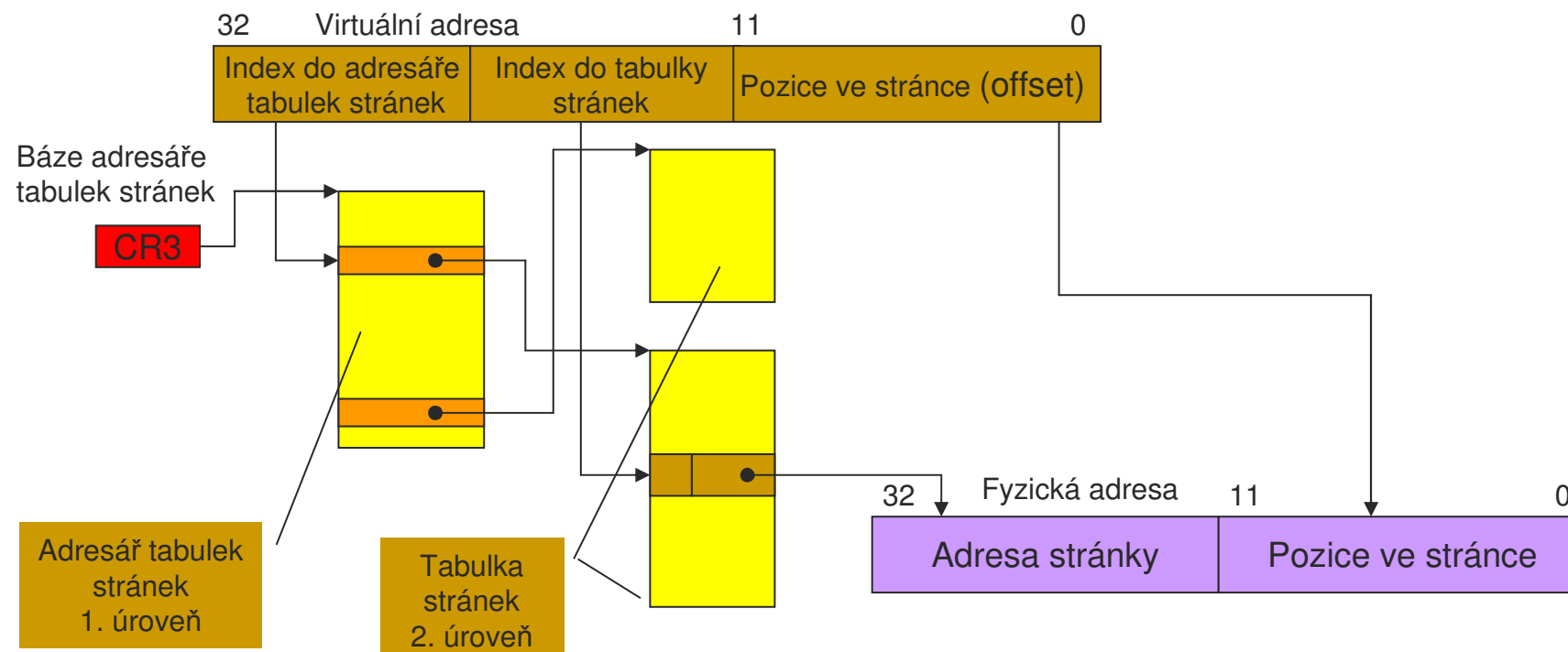


Virtuální adresní prostor - stránkování



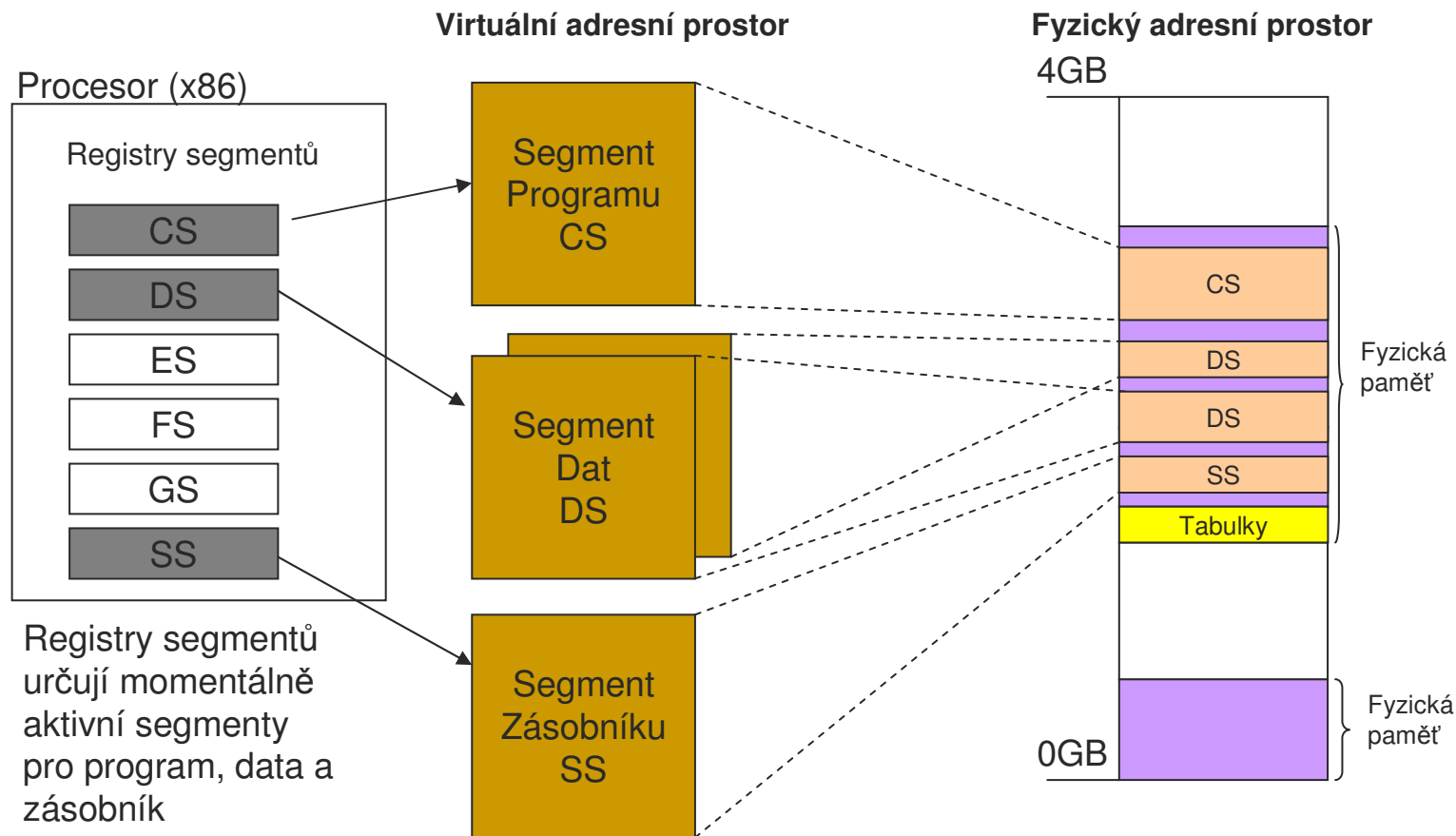
Virtuální adresní prostor je rozdělen do **stránek** (bloků paměti o velikosti 4KB). Operační systém modifikuje záznamy v tabulkách a tím mapuje stránky z fyzické paměti do stránek ve virtuálním adresovém prostoru. Každý proces má své vlastní tabulky stránek, které mapují jinou část fyzické paměti. Tím se zajistí, že jeden proces nemůže přepsat data jinému procesu.

[Stránkování – překlad adresy]



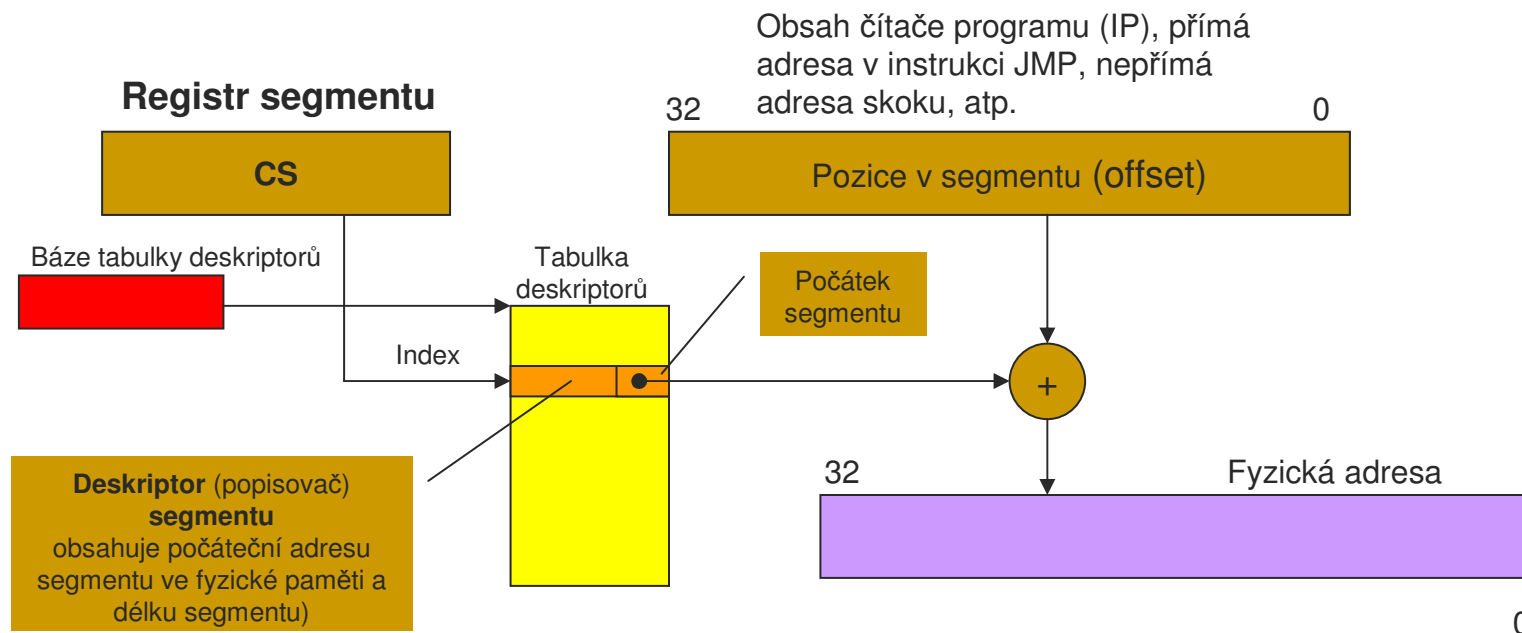
Pokud položka v tabulce stránek indikuje, že stránka je odložena na disk (tj. není v paměti) je vygenerována výjimka a operační systém zpracuje tuto výjimku tak, že odloženou stránku načte z disku zpět do paměti. Pokud není v hlavní paměti místo, pak se nejprve odloží na disk stránka, která se dlouho nepoužívala a pak se načte stránka, která způsobila výjimku.

Virtuální adresní prostor - segmentovaný



[Segmentace – překlad adres]

Fyzický prostor je rozdělen do segmentů (bloků paměti s různou velikostí).



Pozn.: toto schéma platí pro všechny segmentové registry a adresy, které se s nimi implicitně nebo explicitně párují. Uvedený CS registr považujte za příklad.

Význam virtuálních adresních prostorů

- **Zvyšuje stabilitu operačního systému**

Adresové prostory jednotlivých procesů jsou odděleny, proto chyba v programu nemůže způsobit, že jeden proces přepíše data jiného procesu. Podobně lze omezit přístup zápisu nebo čtení do některých částí paměti nastaveními v deskriptorech segmentů nebo v tabulkách stránek. Jakékoliv porušení těchto omezení je indikováno výjimkou.

- **Odstraňuje nutnost relokace programu**

Mají-li se zavést dva programy do fyzické paměti, musí každý program začínat na jiné adrese. Problém je, že tato adresa není předem známa, ale překladač tuto adresu nutně potřebuje, aby mohl vypočítat cílové adresy absolutních skoků. Pokud se má již přeložený program přemístit na jinou adresu, musí se přepočítat cílové adresy skoků (relokovat program), případně změnit ukazatele na všechny statické datové struktury pokud poloha datové paměti se také mění.

Naopak dva různé programy zavedené do dvou různých virtuálních prostorů od stejné adresy mohou ležet na různých adresách ve fyzické paměti. Proto není relokace potřeba.

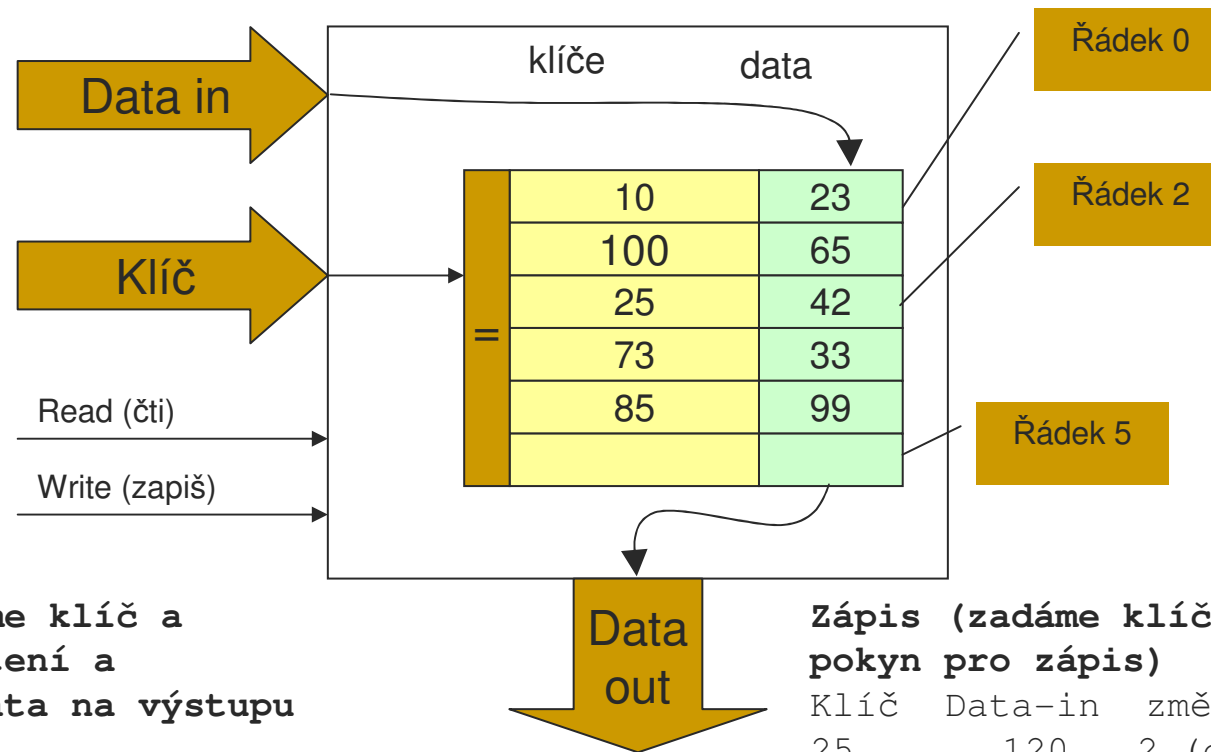
- **Zavádí transparentní mechanismus pro mapování diskové paměti do adresních prostorů**

Mechanismy virtuálních adresních prostorů dovolují přistupovat k disku stejným způsobem, jako když pracujeme s hlavní pamětí. Běžně užívané sekvenční operace čtení a zápisu do souborů jsou nahrazeny operacemi čtení a zápisu do paměti. Paměťovými operacemi můžeme zapisovat a číst z libovolného místa v souboru a v jakémkoliv pořadí. Tento mechanismus se označuje jako paměťově mapované soubory.

Konstrukce skryté paměti (cache)

- Plně asociativní
- Přímá adresovaná
(stupeň asociativity = 1)
- S omezenou asociativitou
(stupeň asociativity > 1)

[Asociativní paměť - princip]



Čtení (zadáme klíč a pokyn pro čtení a očekáváme data na výstupu Data-out)

Klíč	Data-out
25	42
85	99

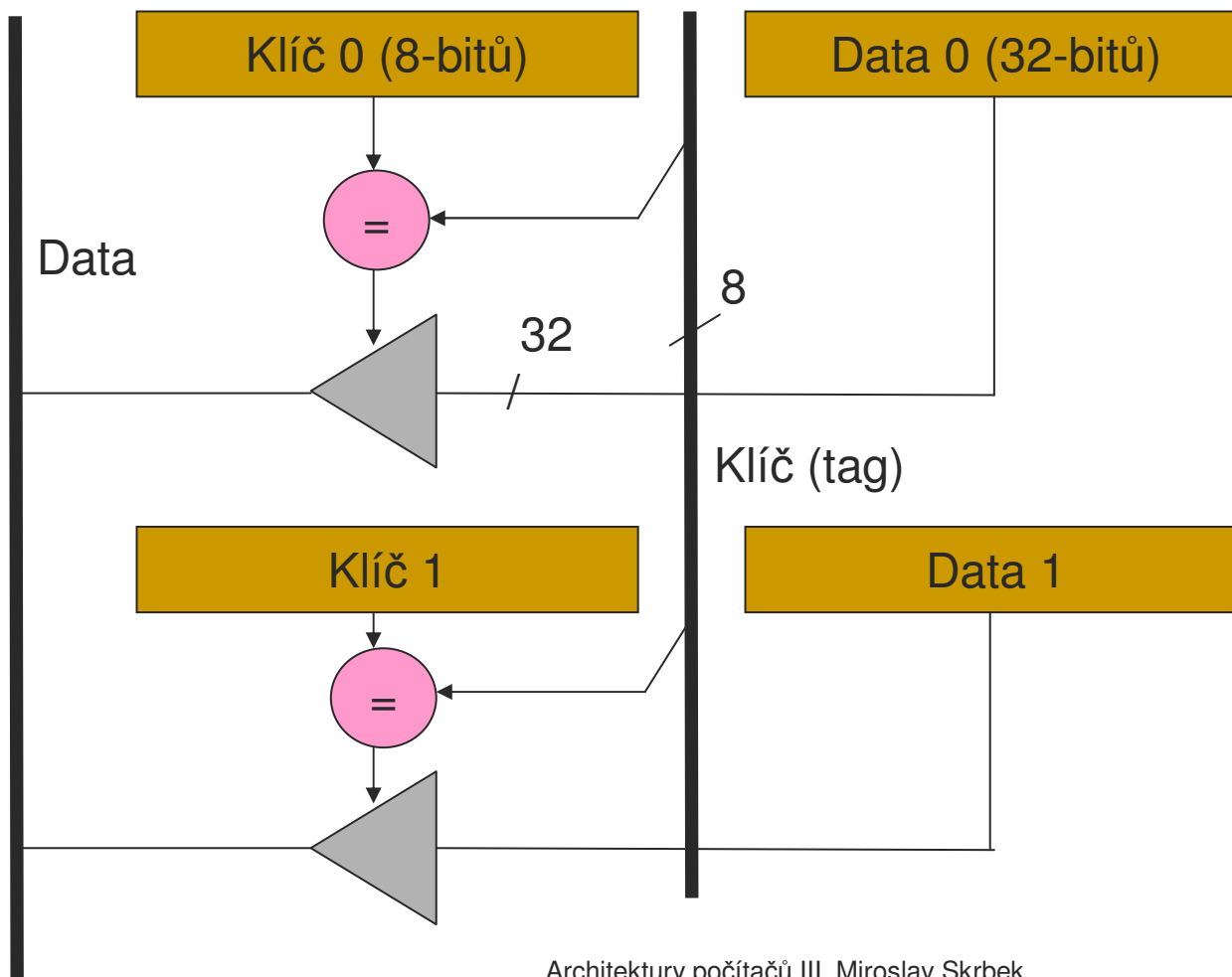
Zápis (zadáme klíč, data-in a pokyn pro zápis)

Klíč	Data-in	změna_řádku
25	120	2 (data na 120)
10	130	0 (data na 130)
15	11	5 (změna klíče na 15 a dat na 11)

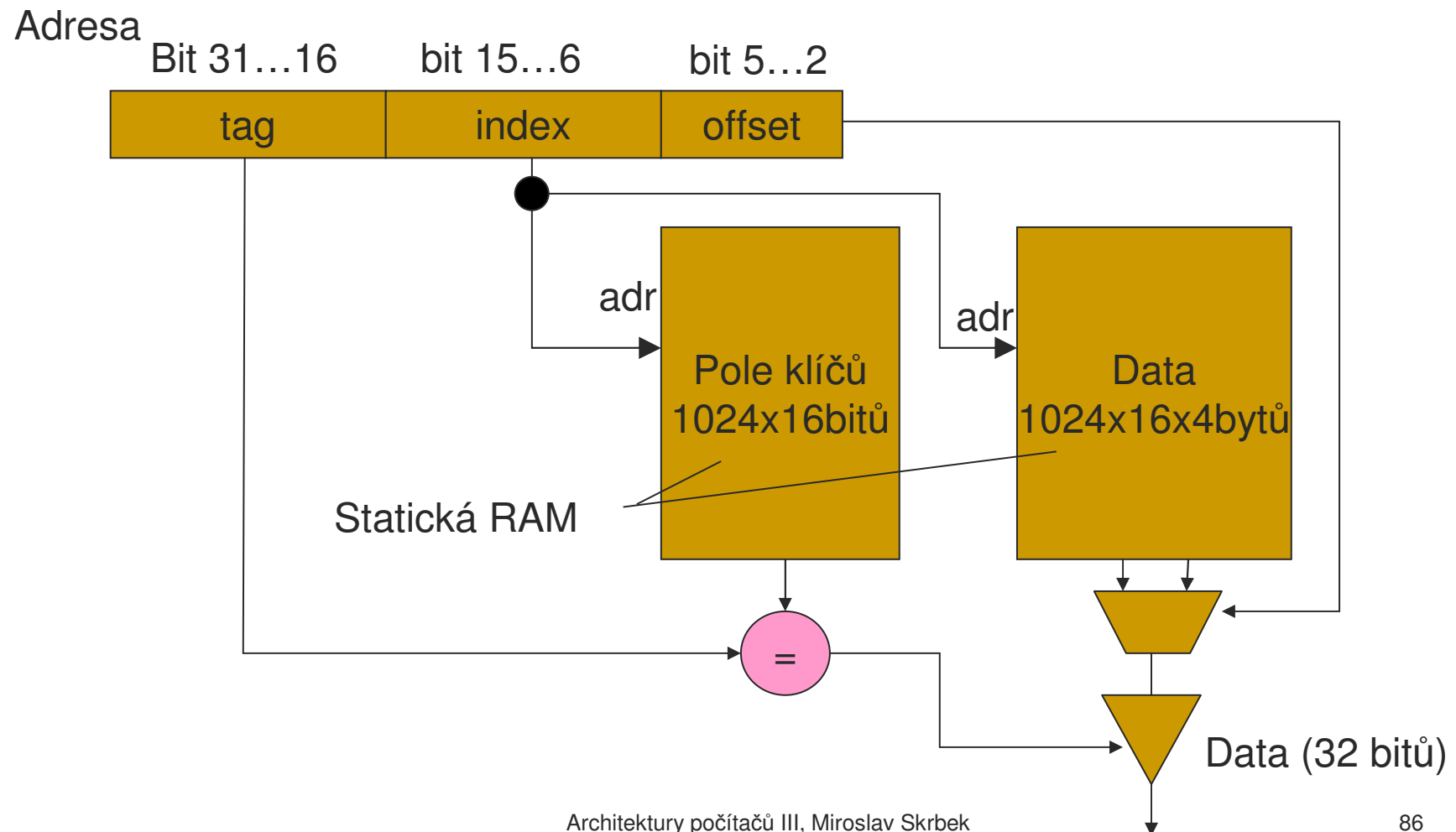
Ostraňování položek z asociativní paměti (ze skryté paměti)

- Náhodně
 - Náhodně vybereme položku (řádek), která bude odstraněna (vymazána) a nahrazena novým obsahem (klíč i data)
- LRU (Least Recently Used)
 - Vyřazuje se nejméně často užívaná položka (řádek)
 - Realizováno čítačem pro každou položku (při každém přístupu na danou položku se zvýší čítače všech ostatních položek o jedničku)

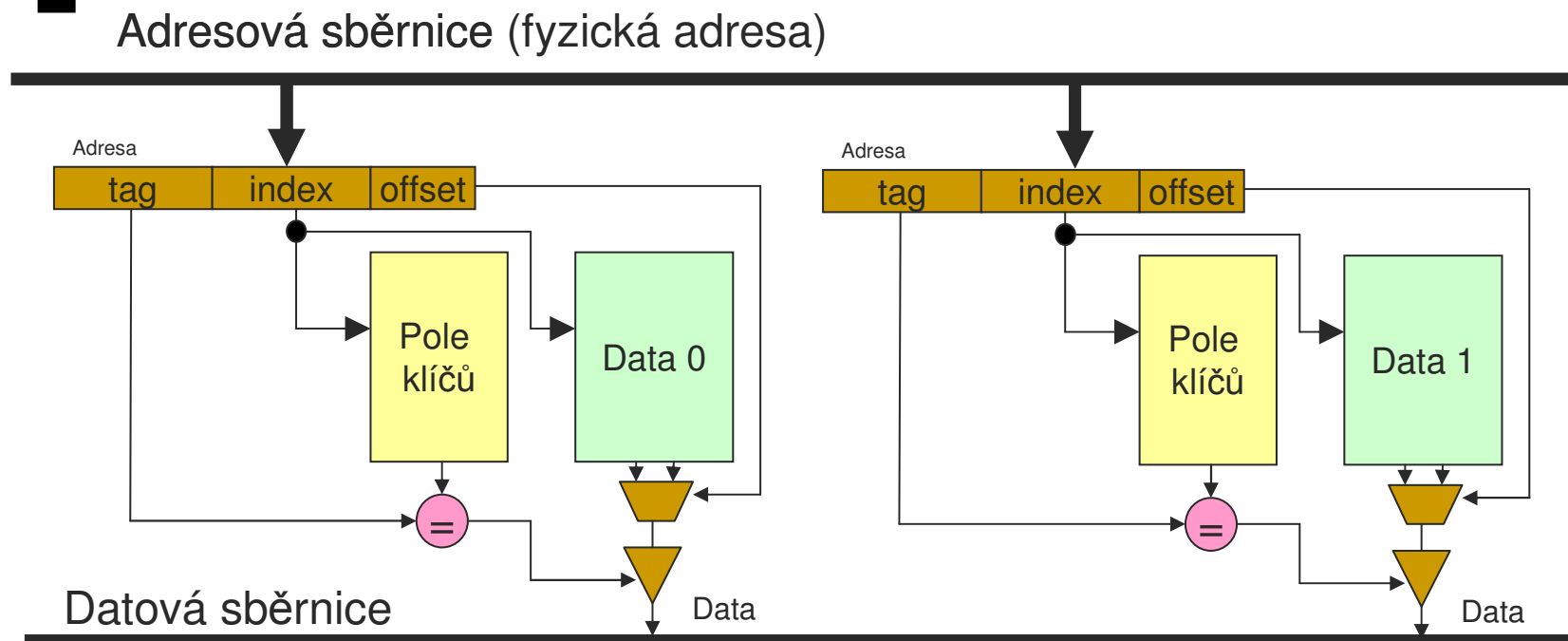
[Plně asociativní paměť]



Přímo adresovaná skrytá paměť (cache)



Stupeň asociativity 2

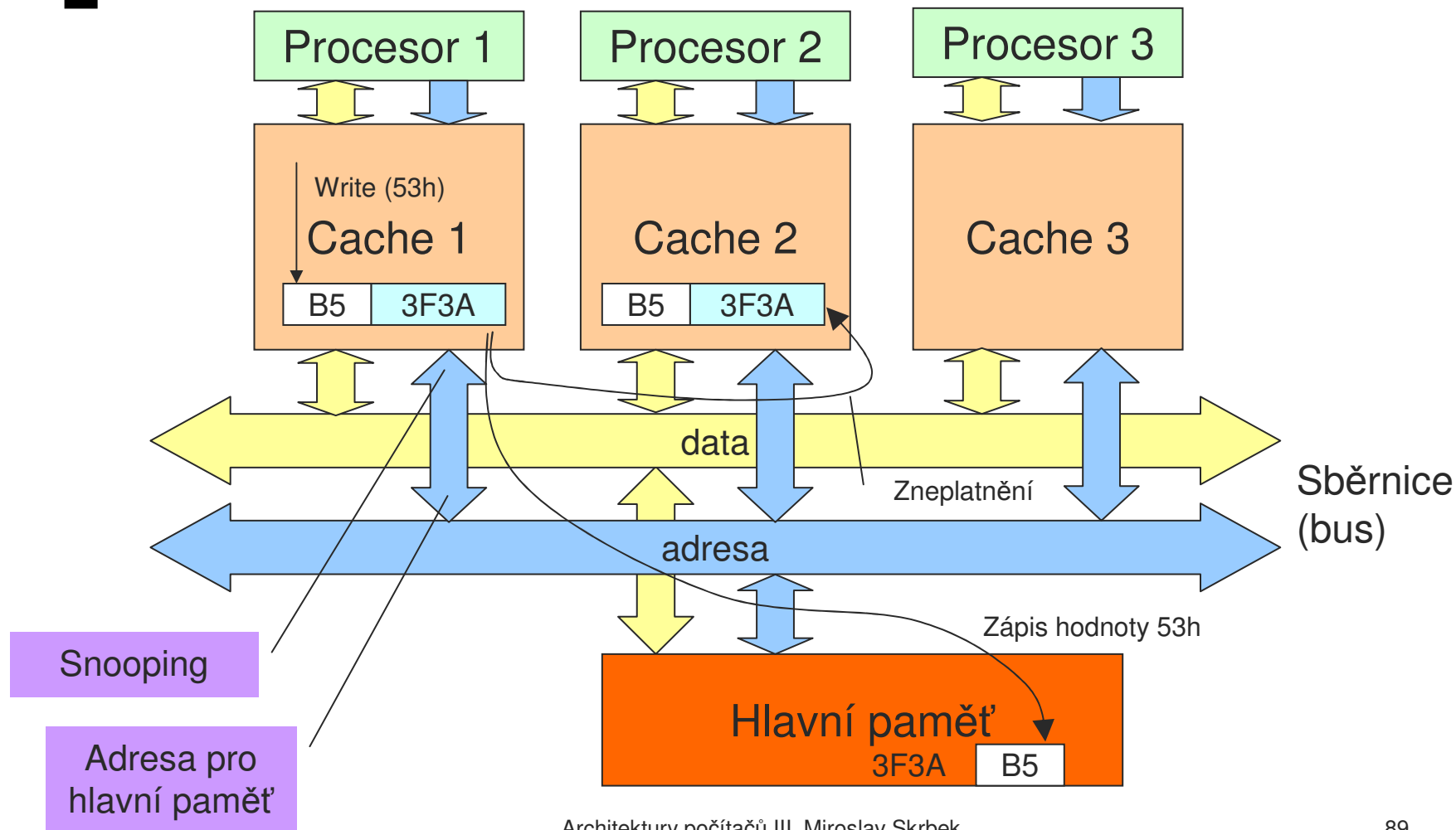


Kromě stupně asociativity 2 se užívá stupeň asociativity 4.

Koherence paměťového podsystemu v více-processorových počítačích se sdílenou pamětí

- V systému je více procesorů (jader) a každý z nich má skrytou paměť (cache)
- Obsah na dané adrese (např. 3F3Ah) může být současně uložen v hlavní paměti a jedné nebo více skrytých pamětech
- Pokud některý z procesorů hodnotu na adrese 3F3Ah přepíše jinou hodnotou, pak je nutné aktualizovat hodnotu nejen v hlavní paměti, ale i skrytých pamětech ostatních procesorů
- K tomuto účelu slouží paměťové koherenční protokoly

Koherence v paměťovém podsystemu

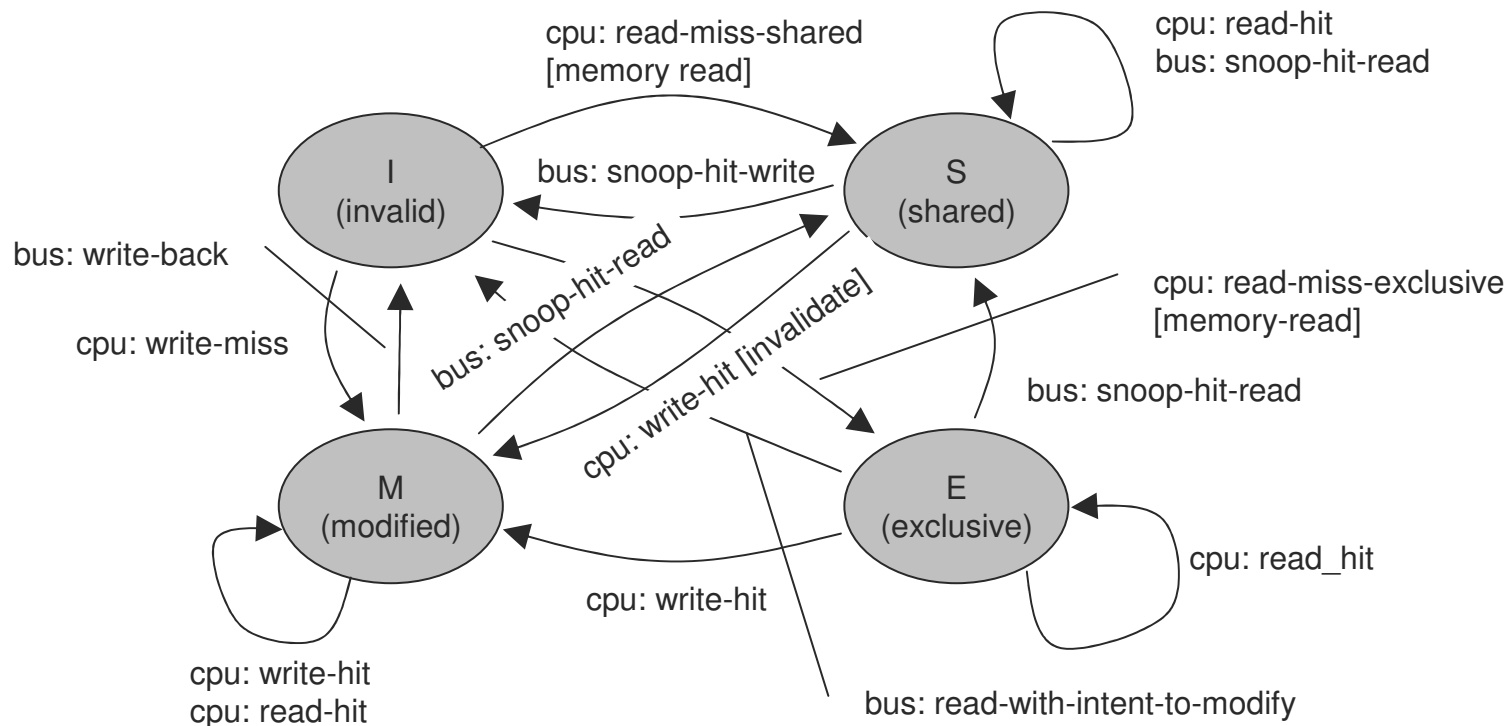


Nejčastěji používané metody zajištění koherence

- Write-through + snooping
 - Při zápisu do paměti se aktualizuje skrytou paměť a zároveň proběhne zápisový cyklus do paměti
 - Ostatní skryté paměti sledují adresovou a řídící sběrnici, rozpoznávají zápisové cykly do paměti a adresa zápisu se shoduje s některou položkou ve skryté paměti, tak položku zneplatní
- MESI protokol
 - Dovoluje zpožděný zápis dat ze skryté paměti do hlavní paměti, jen pokud třeba uvolnit některou položku ze skryté paměti (write-back)
 - Zvyšuje výkon systému (redukce zápisových cyklů do paměti)
 - Složitější na implementaci

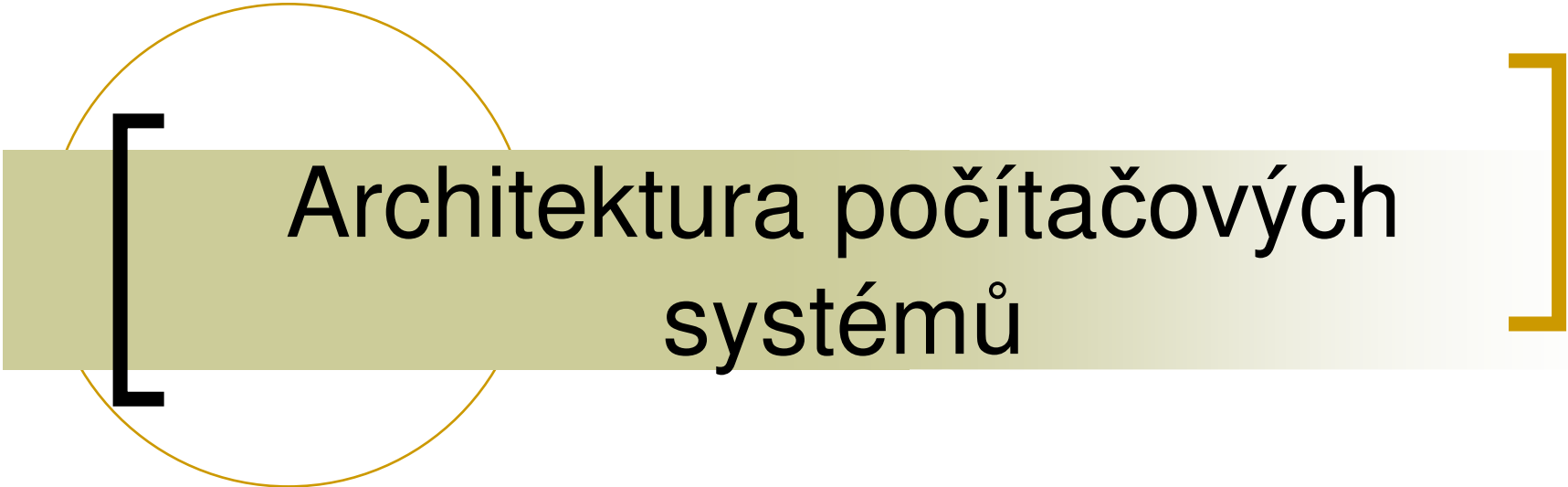
[MESI protokol]

Každá položka ve skryté paměti má příznaky, které kódují čtyři stavy. Přechody mezi stavy jsou popsány následujícím konečným automatem.



[MESI protokol - vysvětlivky]

- **cpu:read-hit**
 - Procesor čte z paměti, data jsou přítomna ve skryté paměti
- **cpu:read-miss**
 - Procesor čte z paměti, data nejsou přítomna ve skryté paměti
- **cpu:write-hit**
 - Procesor zapisuje do paměti, data jsou přítomna ve skryté paměti
- **cpu: write-miss**
 - Procesor zapisuje do paměti, data nejsou přítomna ve skryté paměti
- **bus:snoop-hit-read**
 - Na sběrnici probíhá čtecí cyklus (jiného procesoru) a data odpovídající čtecímu cyklu jsou přítomna ve skryté paměti
- **bus:snoop-hit-write**
 - Na sběrnici probíhá zápisový cyklus (jiného procesoru) a data odpovídající zápisovému cyklu jsou přítomna ve skryté paměti
- **bus: read-with-intent-to-modify**
 - Jiný procesor čte data z paměti za účelem následné změny těchto dat (následného zápisu). Typicky na základě speciální prefetch instrukce
- **bus: invalidate**
 - Jiný procesor zneplatňuje danou položku ve skryté paměti. Typicky na základě zápisu (změny položky existující položky) ve skryté paměti.



Architektura počítačových systémů

UAI/606 Přednáška blok 4

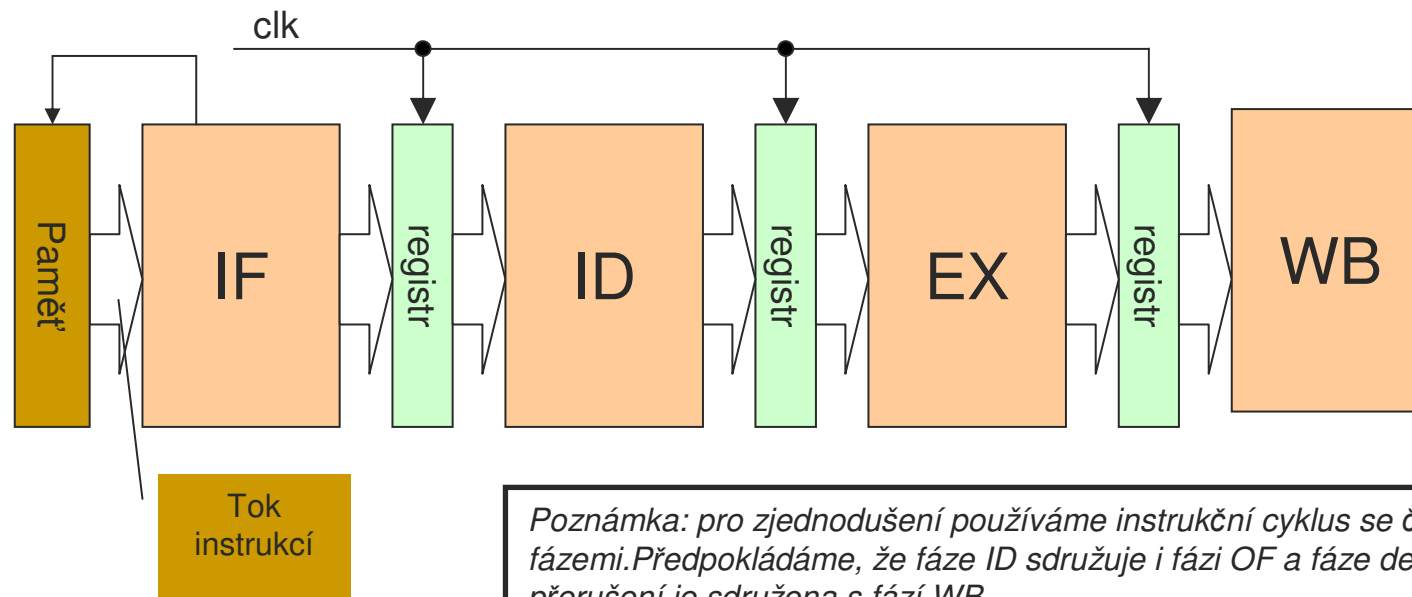
Miroslav Skrbek
mskrbek@prf.jcu.cz

*Ústav aplikované informatiky
Přírodovědecká fakulta
Jihočeské univerzity v Českých Budějovicích*

Platné pro šk.r. 2012/2013

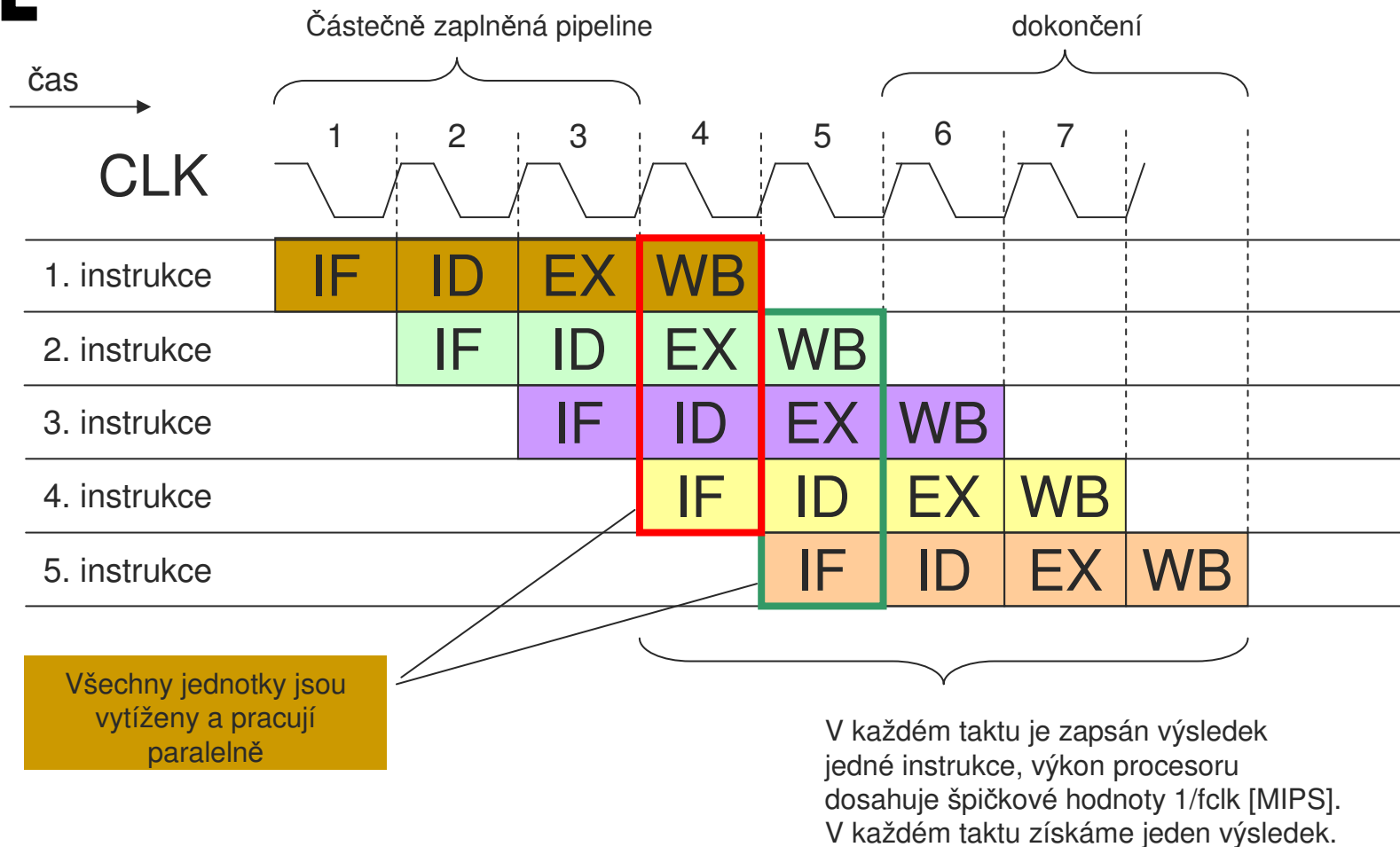
Proudové zpracování instrukcí (pipeline)

Máme 4 jednotky (IF, ID, EX, WB), které pracují paralelně a každá jednotka v daném taktu zpracovává sobě odpovídající fázi instrukčního cyklu, ale jiné instrukce než ostatní jednotky. Činnost připomíná výrobní linku, kde každý pracovník provádí stále jednu operaci a předává polotovar svému kolegovi. Na konci vypadne hotový výrobek.



Poznámka: pro zjednodušení používáme instrukční cyklus se čtyřmi fázemi. Předpokládáme, že fáze ID sdružuje i fázi OF a fáze detekce přerušení je sdružena s fází WB...

Průběh proudového zpracování instrukcí



[Hazardy]

- Strukturní
 - Dané omezeními ve struktuře procesoru
 - Omezeními vnitřního propojení
 - Přístupem ke zdrojům (např. dvě instrukce chtějí naráz výsledek zapsat do registrové banky a technicky lze zapsat pouze jeden výsledek)
- Datové
 - Vyplývají z datových závislostí mezi instrukcemi
 - Například: druhá instrukce vyžaduje zdrojový operand, který je výsledkem první instrukce.
- Řídící
 - Jsou způsobeny změnou instrukčního toku. Typicky skokovými instrukcemi, ale i přerušením nebo výjimkami.

[Datové hazardy]

- RAW (read after write)
 - Druhá instrukce se snaží číst data dříve, než je první instrukce zapíše do registrové banky
- WAW (write after write)
 - Druhá instrukce se snaží zapsat (do téhož registru nebo téhož paměťového místa) než to učiní první instrukce
- WAR (write after read)
 - Druhá instrukce zapíše data dříve, než je první instrukce přečte. První instrukce přečte chybně novou hodnotu.

[Problémy v pipeline (hazardy)]

- Špičkového výkonu je možno dosáhnout je při naplněné pipeline.
- V některých situacích je ale nutné pipeline vypláchnout (zrušením rozpracovaných instrukcí nebo počkat na dokončení rozpracovaných instrukcí) nebo pozastavením některých instrukcí v pipeline vložením čekacích taktů (stall)
- Vkládání čekacích taktů v době naplňování, doběhu nebo pozastavení pipeline snižuje výkon procesoru
- Situace, které si vynucují vkládání čekacích taktů se nazývají hazardy

[Datové hazardy - příklady]

Hazard RAW – SUB přečte hodnotu r1 dříve, než ji ADD stačí zapsat do registru

```
ADD r1, r3, r7
SUB r4, r1, r2
```

Datová
závislost

Hazard WAR – ADD zapíše hodnotu do r1 dříve, než ji stačí instrukce ST (store) načíst z registru pro následné uložení do paměti. Načtení může r1 může být v pipeline odloženo do další fáze kvůli preinkrementaci, kdy je nutné nejprve zvýšit r3 o jedničku, aby se získala adresa pro uložení r1.

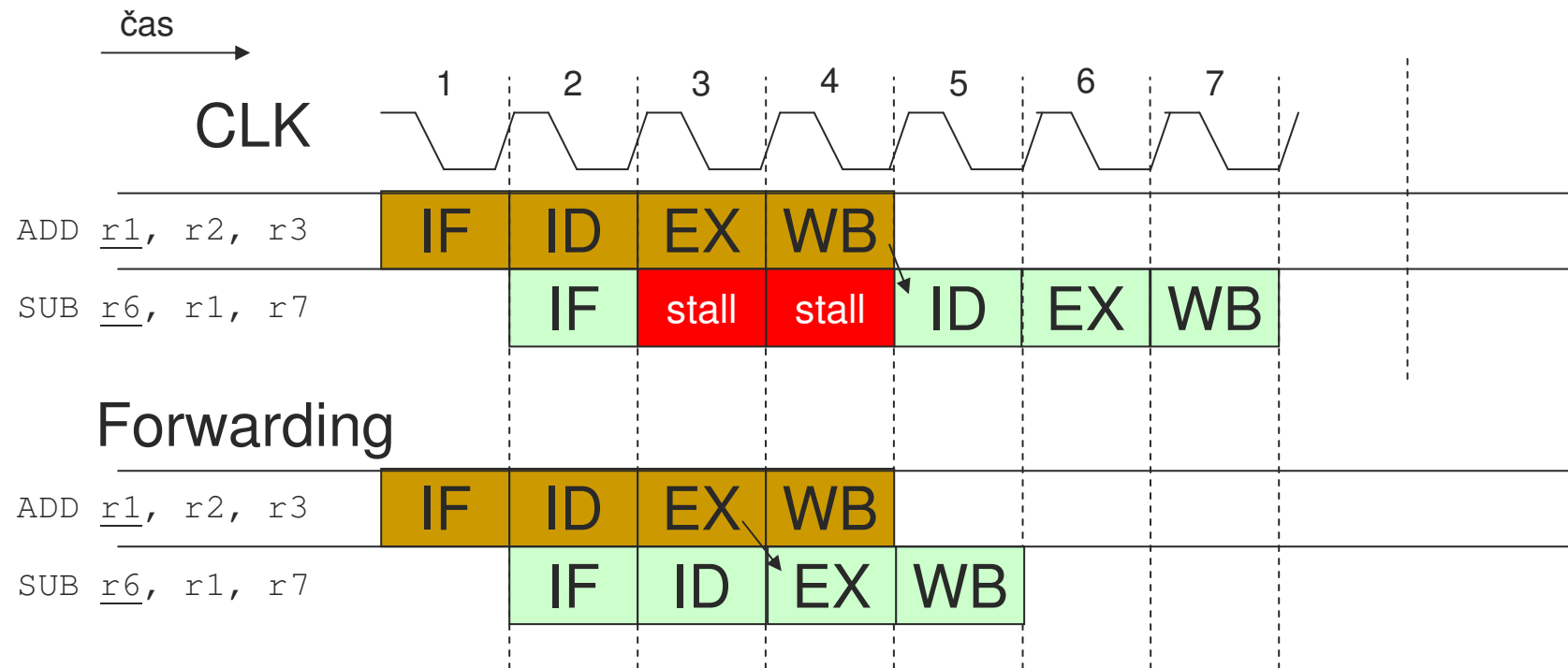
```
ST (+r3), r1
ADD r1, r3, r2
```

Hazard WAW – Součet čísel floating point (FADD) má velkou latency (trvá více taktů), proto se může stát, že nedojde-li ke skoku, pak nedojde k přepsání obsahu r1 instrukcí ADD, ale program pokračuje s výsledkem instrukce FADD a to neodpovídá zápisu programu.

```
FADD r1, r2, r3
JZ loop
ADD r1, r4, r2
```

Poznámka: cílový registr je označen podtržením, závislé registry zvýrazněny.

Řešení RAW hazardů - Forwarding

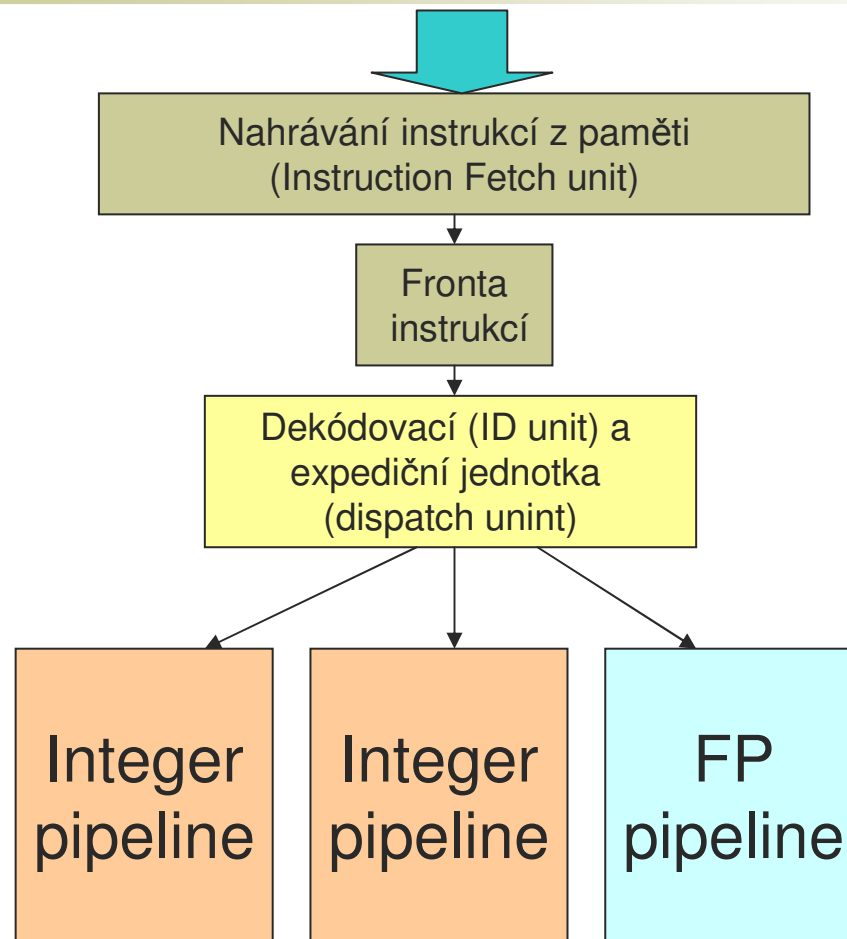


Dodatečné propojení výstupu ALU se vstupem ALU (samozřejmě přes pomocný registr), je možné předat výsledek předchozí instrukce před fází WB jako operand přímo do fáze EX následující instrukce. Díky tomu není nutné čekat až se výsledek zapíše do registru r1 a pak jej v dalším taktu z registru r1 vyzvednout.

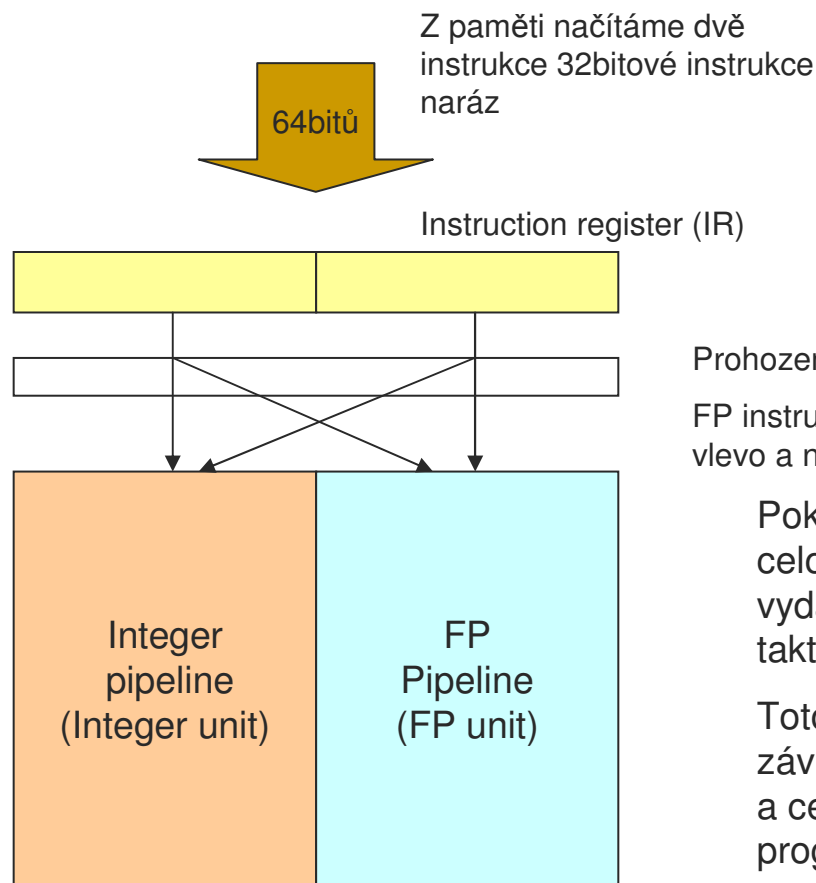
[Superskalární architektury]

- ILP (Instruction level paralelism) paralelismus na úrovni instrukcí
- CPI (clock per instruction) < 1
- Vydání více než jedné instrukce v jednom taktu, typicky 2-8
- Více funkčních pipeline, minimálně celočíselná a floating point, může být i více celočíselných v kombinaci s floating point
- Program má podobu sekvenčního programu, paralelizace se provádí automaticky v hardware
- Optimalizující překladače mohou ovlivnit výkon procesoru vhodnou změnou pořadí instrukcí

[Superskalární architektura]



Jednoduchá superskalární architektura (statické plánování)



Pozn.: pokud instrukce mají různou délku, je možné načíst do IR registru více než dvě instrukce (proto se před IR předřazuje fronta instrukcí, ze které se pak vybírá po dvojicích)

Prohození instrukce (swap)

FP instrukce se může objevit vlevo a naopak

Pokud se v programu po dvojicích prolínají celočíselné i FP operace, je možné dosáhnout vydání dvou instrukcí ke zpracování v jednom taktu.

Toto je ideální stav, který narušují datové závislosti a nevyvážený poměr mezi počtem FP a celočíselných instrukcí v daném úseku programu.

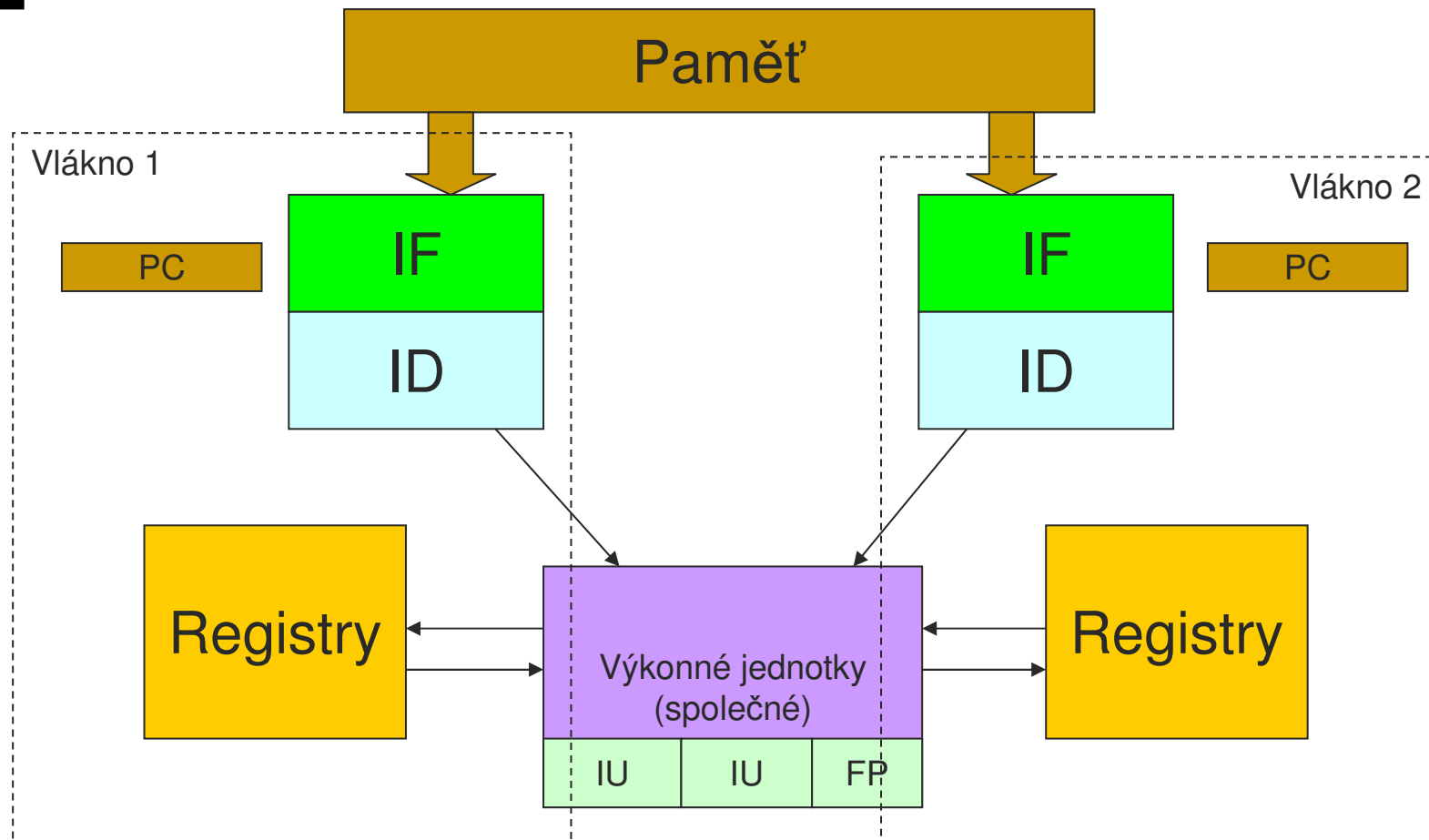
[CDC6600]

- Rok 1964
- 4 FP jednotky
- 5 adresačních jednotek
- 7 celočíselných jednotek
- Dynamické plánování
- Zpracování instrukcí mimo pořadí
- Používá scoreboarding pro vydávání instrukcí ke zpracování

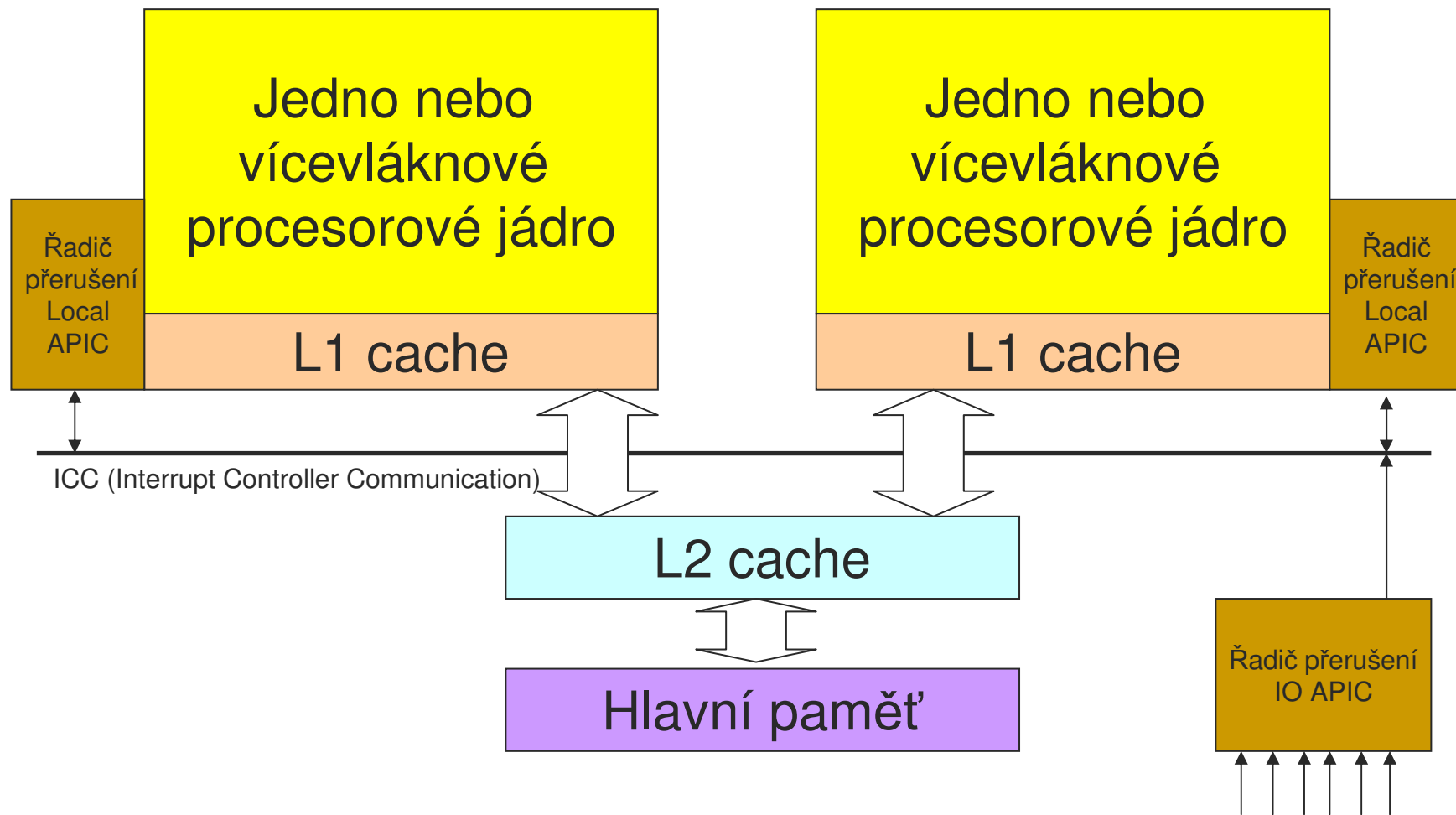
Zpracování instrukcí mimo pořadí

- Po dekodování se instrukce řadí do fronty. Odkud jsou vydávány ke zpracování podle toho, zda mohou být zpracovány (bez hazardu), tj. v jiném pořadí než byly zapsány do fronty.
- Instrukce může být vydána pokud je volná jí odpovídající jednotka, jsou k dispozici všechny operandy, případně není zpracovávána instrukce, jejíž cílový operand je shodný se zdrojovým operandem instrukce, která se má vydat ke zpracování.
- Po dokončení instrukce je aktualizován stav procesoru tak, aby byly vyloučeny hazardy typu WAW a WAR.
- Několik instrukcí se vždy zpracovává ve funkčních jednotkách a několik je připraveno na zpracování. Tím je ve stádiu rozpracovanosti více instrukcí (větší úsek programu) a instrukce mohou díky tomu přeskupit tak, aby se minimalizoval počet stall taktů.
- Pro zjišťování závislostí mezi instrukcemi se používá obdobných technik jako v dataflow architekturách

Vícevláknové procesory



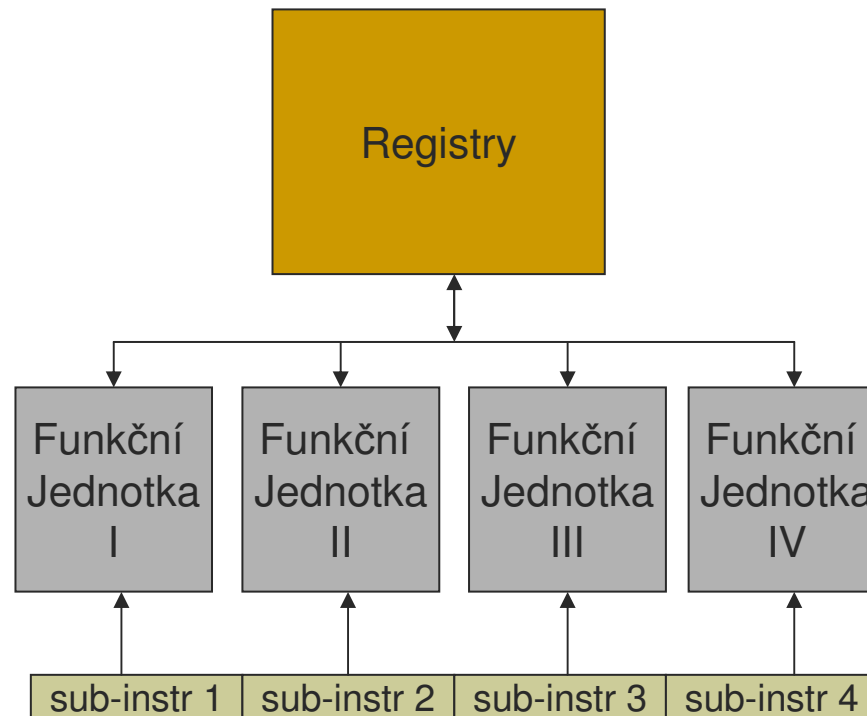
Vícejádrové procesory



[Architektury VLIW]

- Více paralelně pracujících jednotek
- Explicitně vyjádřený paralelismus
- Široké instrukce rozdělené do několika sub-instrukcí
- Pouze jedna skoková sub-instrukce na instrukci

[Příklad VLIW architektury]



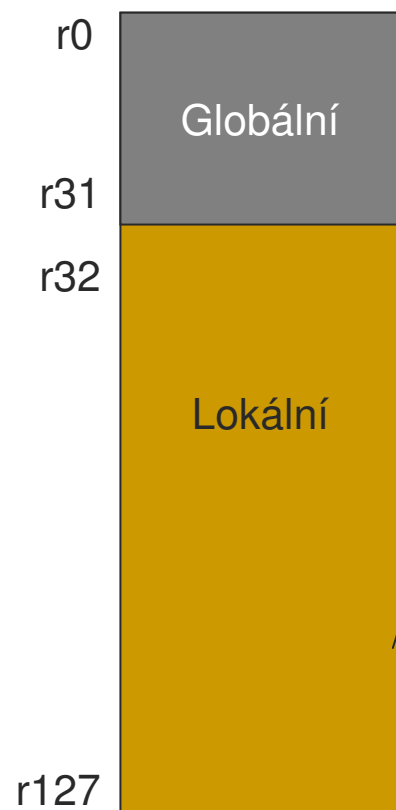
[Problémy a jejich řešení]

- Potřeba mnohabránové registrové banky
 - řešení separace registrových bank funkčních jednotek a omezení výměny dat mezi bankami
 - Například celočíselná funkční jednotka a jednotka FP mohou mít oddělené registry
- Subinstrukce v rámci jedné instrukce musí být nezávislé
 - řeší překladač sledováním závislostí v generovaném kódu a změnou pořadí vkládání subinstrukcí do jednotlivých instrukcí
- Nesmí být více jak jedna skoková subinstrukce v instrukci
 - Skokové subinstrukce mají vymezen jeden daný slot
- Pro dosažení maximální výkonnosti musí být všechny sloty pro subinstrukce vyplněné
 - Řeší překladač, pokud to lze dosáhnout

[Intel Itanium]

- VLIW architektura
- Instrukce 128 bitů
*3 sub-instrukce (3 * 41 bitů + 5 bitů template)*
- Podmíněné instrukce
- 128 registrů (celočíslné, 64-bitů)
registrový zásobník, rotace registrů
- 128 FP registrů
- Predikátové registry
- Registry pro adresy skoků
- Aplikační registry a registry pro měření výkonnosti
- Režimy IA-64 a IA-32

[Celočíselné (GP) registry]



Tyto registry tvoří zásobník, který se posouvá při skoku do podprogramu

R0 – vždy je nula
R12 – typicky ukazatel zásobníku v paměti
R14-R31 – volně k použití

[Aritmetické instrukce]

```
add r1 = r2, r3  
add r1 = r2, r3, 1  
add r1 = imm, r2
```

$r1 = r2 + r3 + 1$

```
sub r1 = r2, r3  
sub r1 = r2, r3, 1  
sub r1 = imm, r2
```

$r1 = (r2 \ll \text{count}) + r3$

```
shladd r1 = r2, count, r3
```

Poznámka: chybí zde instrukce násobení celých čísel, která je realizována jako instrukce floating-point jednotky pracující s FP registry.

Logické instrukce, posuvy a přesuny

Logické instrukce

```
or r1 = r2, r3
or r1 = imm8, r2

and r1 = r2, r3
and r1 = imm8, r2

andcm r1 = r2, r3
andcm r1 = imm8, r2

xor r1 = r2, r3
xor r1 = imm8, r2
```

Posuvy

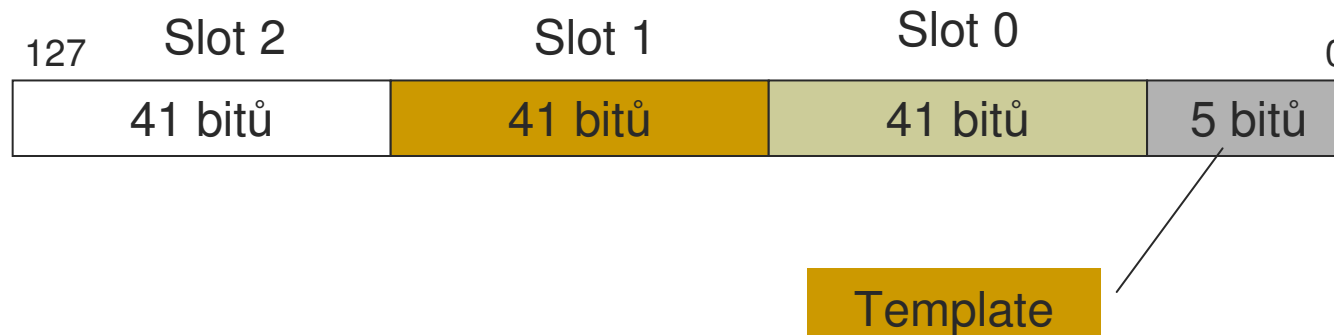
```
shr      r1 = r2, r3
shr.u    r1 = r2, r3
shr      r1 = r2, count
shr.u    r1 = r2, count

shl      r1 = r2, r3
shl      r1 = r2, count
```

Přesuny

```
mov      r7 = r6
mov      r34 = imm22
movl     r32 = imm64
```

[Paralelní provádění instrukcí]



- Každý slot pojme jednu instrukci
- Skupina – jedna nebo více instrukcí provedených paralelně
- V rámci skupiny musí být jedno v jakém pořadí se instrukce dokončí
- V rámci skupiny jsou zakázány datové závislosti typu WAW, a RAW

[Vytváření skupin]

```
add r1 = r2, r3; sub r6 = r7, r8 ; add r5 = r4, r10;;  
add r1 = r1, r6; sub r5 = r5, r11; nop                ;;
```

Skupiny jsou odděleny dvěma středníky



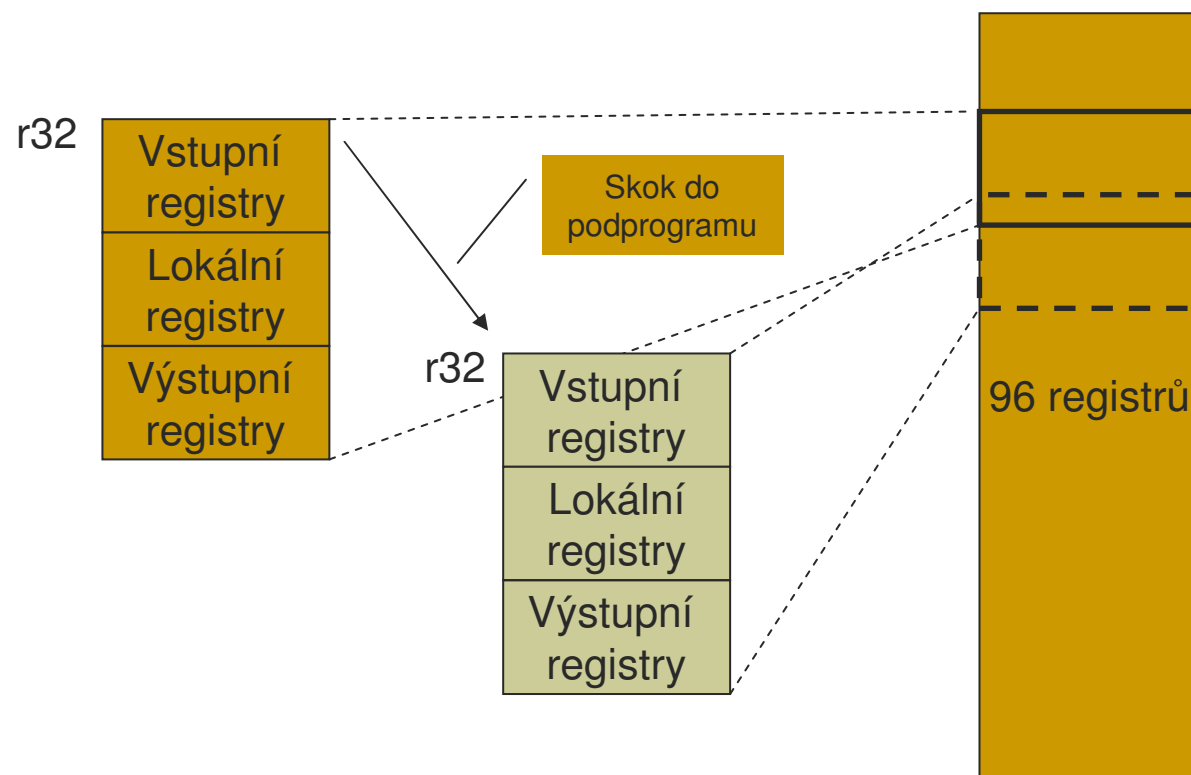
Konec skupiny

```
add r1 = r2, r3; sub r6 = r7, r8 ;;  
add r5 = r4, r10; ...
```



Konec skupiny

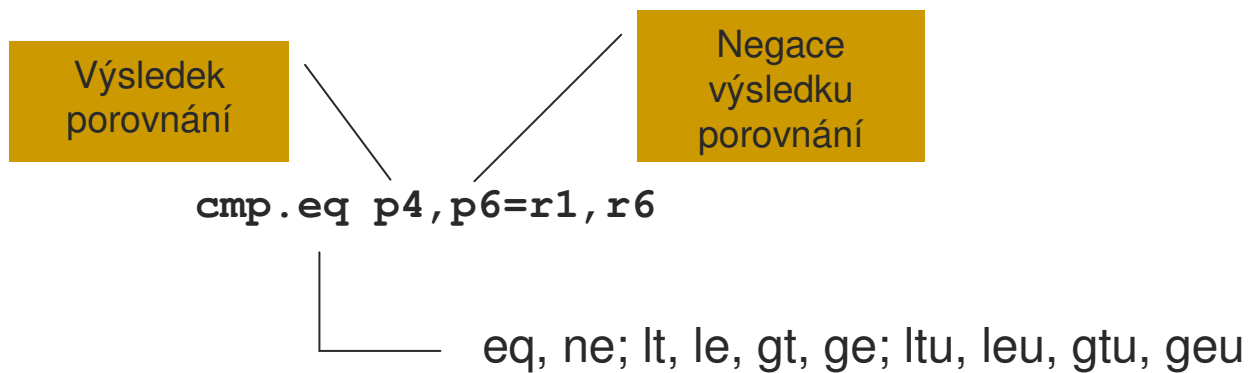
[Registrový zásobník]



[Predikátové registry]

- 64 jednobitových registrů
- Označené $p_0 - p_{63}$
- p_0 je vždy 1
- Nastavovány porovnávacími instrukcemi
- Většinu instrukcí je možno podmínit těmito registry

[Porovnávací instrukce]



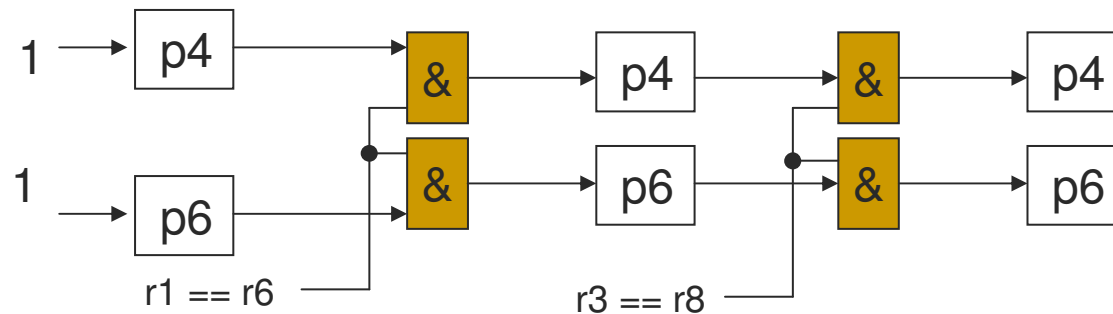
Poznámka: u značí operaci bez znaménka

Porovnávací instrukce

Nelze provést paralelně - datová závislost typu WAW

~~cmp.eq p4, p6=r1, r6; cmp.eq p4, p6=r3, r8~~

cmp.eq.and p4, p6=r1, r6; cmp.eq.and p4, p6=r3, r8 ;;



Paralelně realizuje podmínku: `if (r1==r6 && r3 == r8) ...`

[Podmíněné instrukce]

```
if (r7==r8) r3=r4 + r5; else r8 = r2 + r6;
```

```
cmp.eq p3, p4 = r7, r8;;
```

```
(p3) add r3 = r4, r5; (p4) add r8 = r2, r6;;
```

Pokud není registr p3 nebo p4 nastaven na jedničku odpovídající instrukce se nahradí instrukcí NOP

Samotné porovnání lze také podmínit

```
(p5) cmp.eq p3, p4 = r7, r8;;
```

[Řídící spekulace]

```
ld.s r14 = [r15]  
ld.s r17 = [r16]
```

.s značí že instrukce nevyvolá výjimku, ale nastaví NAT bit

```
...  
cmp.eq p6, p7=r16, r18  
(p6) chk.s r14, recovery1  
(p7) chk.s r17, recovery2  
...
```

```
continue:
```

```
...  
br.ret.sptk.many b0
```

Pokud je podmínka splněna tak se dále použije se r14 jinak se použije r17

```
recovery1:
```

```
...
```

```
br.sptk.many continue
```

```
recovery2:
```

```
...
```

```
br.sptk.many continue
```

Recovery code se vyvolá v případě, že je nastaven NAT bit u testovaného registru

Poznámka: řídící spekulace nám dovoluje předřadit instrukce load z obou větví podmíněného příkazu před vlastní vyhodnocení podmínky, aniž by výjimka ve větvi, která se nakonec nepoužije, narušila běh programu ve větvi, která naopak použita bude.

[NaT (Not a Thing)]

- Indikují výjimku
- Je to buď 65. bit `gr` registru nebo hodnota `NaTVal` uložená v FP registru
- Instrukce jako `add`, `sub` apod. propagují NaT hodnotu NaT bitu do cílového registru
- Instrukce jako `fadd`, `fsub` apod. propagují hodnotu `NaTVal` do cílového registru
- Instrukce `chk` (check) kontroluje NaT bit a pokud je nastaven provede skok na recovery code.

[Datová spekulace]

```
ld8.a r14 = [r15]
```

```
...
```

```
st8 [r16] = r17  
chk.a r14, recovery
```

```
add r16=r14, r15
```

.a značí spekulativní nahrání dat
(advanced load)
Adresa a cílový registr se uloží jako další
záznam do ALAT

Pokud adresa v r15 je totožná s
adresou v r16, položka v ALAT
odpovídající adrese je
vymazána (zneplatněna)

Recovery code se vyvolá v případě, že
nebyla nalezena položka v ALAT
odpovídající cílovému registru r14

Poznámka: místo chk.a můžeme použít ld8.c r14=[r15], která v případě neexistence položky v ALAT, nahraje znovu data do registru. V registru r15 musí být stejná adresa jako u ld8.a.

ALAT = Advanced Load Address Table architektura počítačových systémů, Miroslav
Skrbek

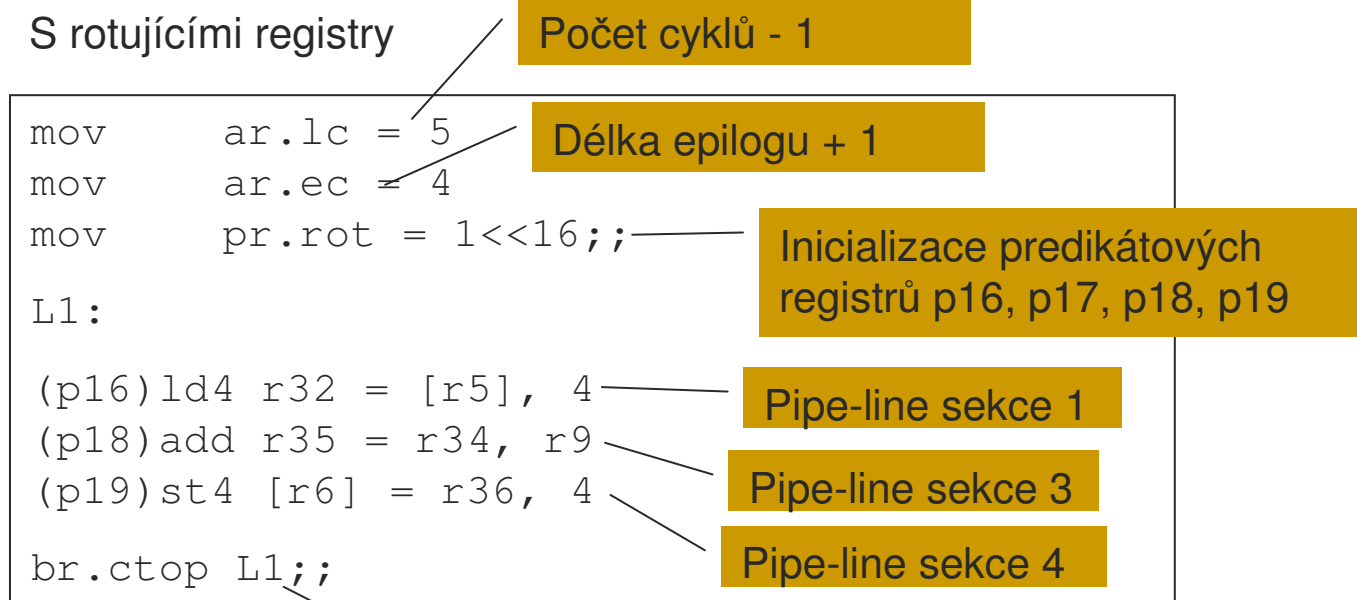
[Podpora cyklů I]

Počet cyklů - 1

```
mov     ar.lc = 5  
L1:  
... tělo cyklu ...  
br.cloop L1;;
```

Odečte jedničku od LC, odrotuje registrovou banku, a skočí pokud lc != 0

[Podpora cyklů II]



Odečte jedničku od LC (v epilogu od EC), odrotuje registrové banky, a skočí pokud EC != 0

[Podpora cyklů III]

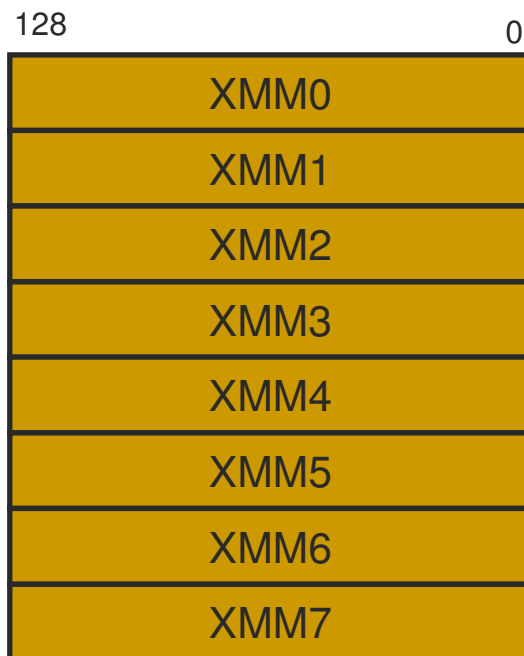
	Cyklus	M	I	M	B	p16	p17	p18	p19	LC	EC
Prolog	0	ld4	nop	nop	br	1	0	0	0	5	4
	1	ld4	nop	nop	br	1	1	0	0	4	4
	2	ld4	add	nop	br	1	1	1	0	3	4
Kernel	3	ld4	add	st4	br	1	1	1	1	2	4
	4	ld4	add	st4	br	1	1	1	1	1	4
	5	ld4	add	st4	br	1	1	1	1	0	4
Epilog	6	nop	add	st4	br	0	1	1	1	0	3
	7	nop	add	st4	br	0	0	1	1	0	2
	8	nop	nop	st4	br	0	0	0	1	0	1
	9					0	0	0	0	0	0

[SIMD architektury v procesorech]

[Technologie SSE, SSE2, SSE3]

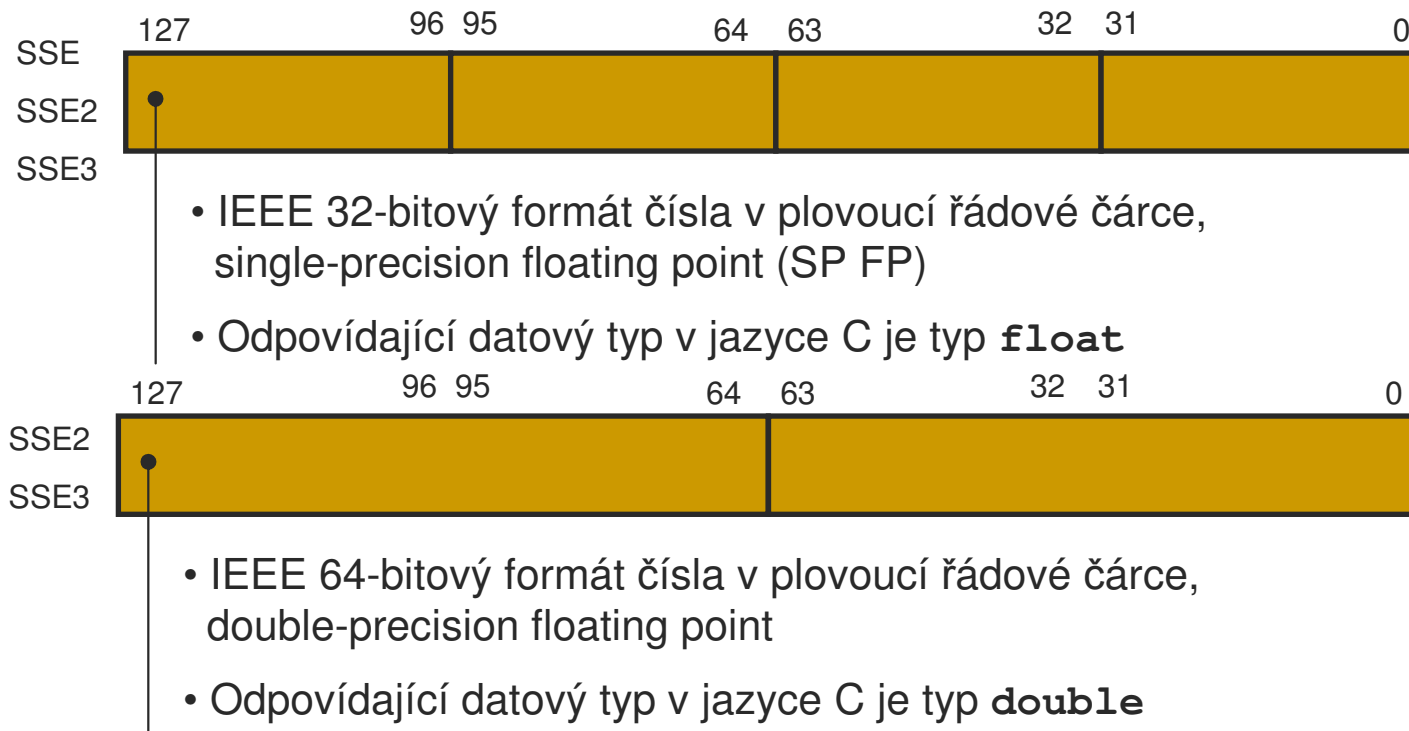
- **Streaming SIMD Extensions**
- Vyvinuta firmou Intel
- Poprvé implementováno v procesorech Pentium III
- Určeno pro podporu 2D a 3D grafiky

[SSE registry]

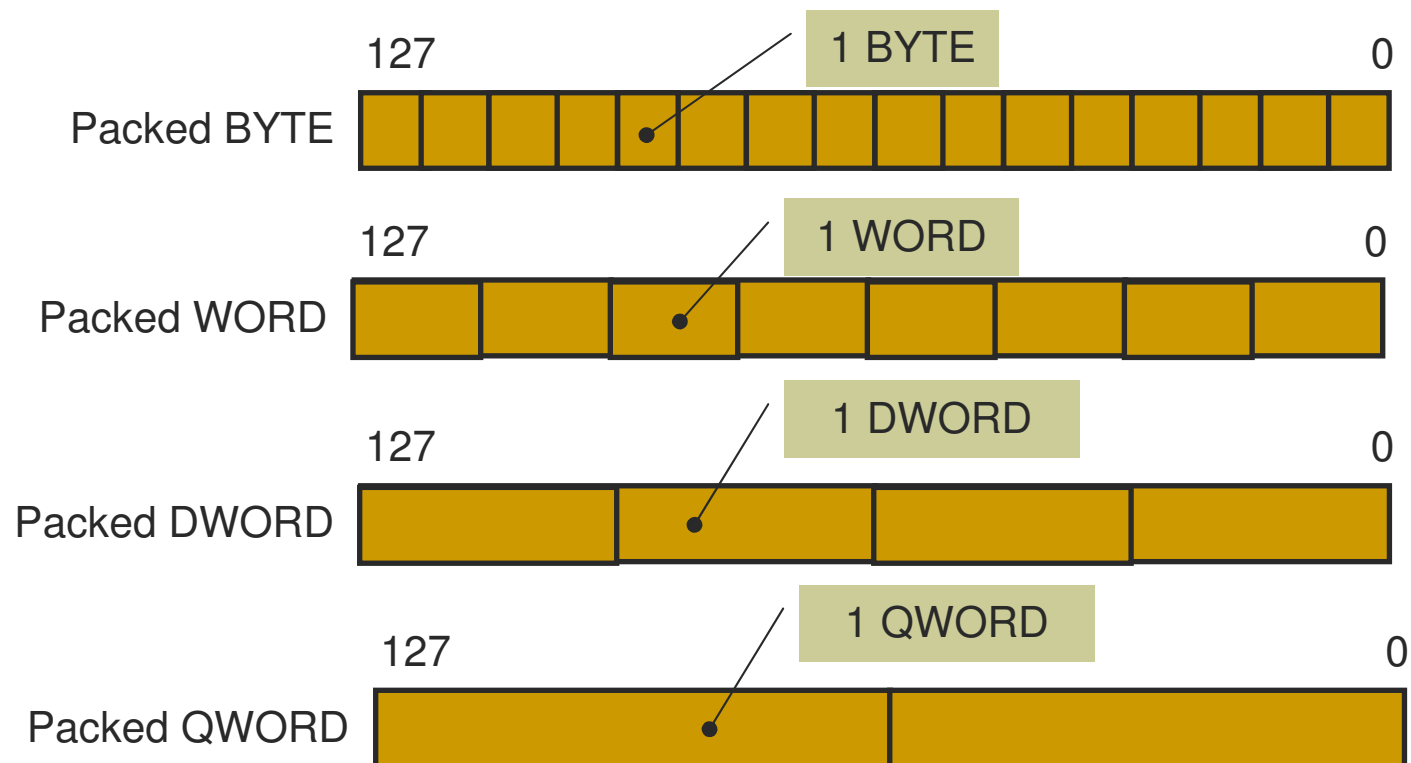


Sada XMM registrů je oddělenou sadou registrů, která se nesdílí s žádnými jinými registry. Proto je například možno bez problémů míchat SSE a FP instrukce.

[Datové typy I]



[Datové typy II (jen SSE2 a SSE3)]



[SIMD a skalární FP operace]

SSE zavádí kromě SIMD také skalární FP operace nad XMM registry

Skalární FP operace se vykoná pouze nad jedním FP číslem, které leží na nejnižších 32(64) bitech registru.

Důvodem pro zavedení těchto operací je zachování stejného registrového modelu pro SIMD a skalární operace.

Bez skalárních instrukcí je programátor nucen přejít na standardní 64-bitové FP instrukce, kde se ale používá zásobníkový model.

[Aritmetické Instrukce]

SSE, SSE2, SSE3	SSE2, SSE3
ADDPS, ADDSS;	ADDPD, ADDSD
SUBPS, SUBSS;	SUBPD, SUBSD
MULPS, MULSS;	MULPD, MULSD
DIVPS, DIVSS;	DIVPD, DIVSD
RCPPS, RCPSS;	RCPPD, RCPSD
SQRTPS, SQRTSS;	SQRTPD, SQRTSD
MAXPS, MAXSS;	MAXPD, MAXSD
MINPS, MINSS;	MINPD, MINSD

[ADD|MUL|...][P|S][S|D]

S - single, D - double
P - packed, S - scalar
operace

```
ADDPS    xmmreg1, xmmreg2/mem128
ADDPS    xmm0,    xmm1
```

[Logické instrukce]

ANDPS; ANDPD
ANDNPS; ANDNPD
ORPS; ORPD
XORPS; XORPD

[Instrukce porovnání]

CMPPS, CMPSS; CMPPD, CMPSD
COMISS ; COMISD
UCOMISS ; UCOMISD

CMP [P | S] [S | D]

└── S - single, D - double
└── P - packed, S - scalar
└── operace porovnání pro FP čísla

CMPPS xmmreg1, xmmreg2/mem128

{U}COMIS [S | D]

└── S - single, D - double
└── skalární operace porovnání pro FP čísla
s výsledkem uloženým do EFLAGS
└── U - tolerantnější při generování
výjimek pro čísla typu NaN)

CMPPS xmmreg1, xmmreg2/mem128

[Konverze I]

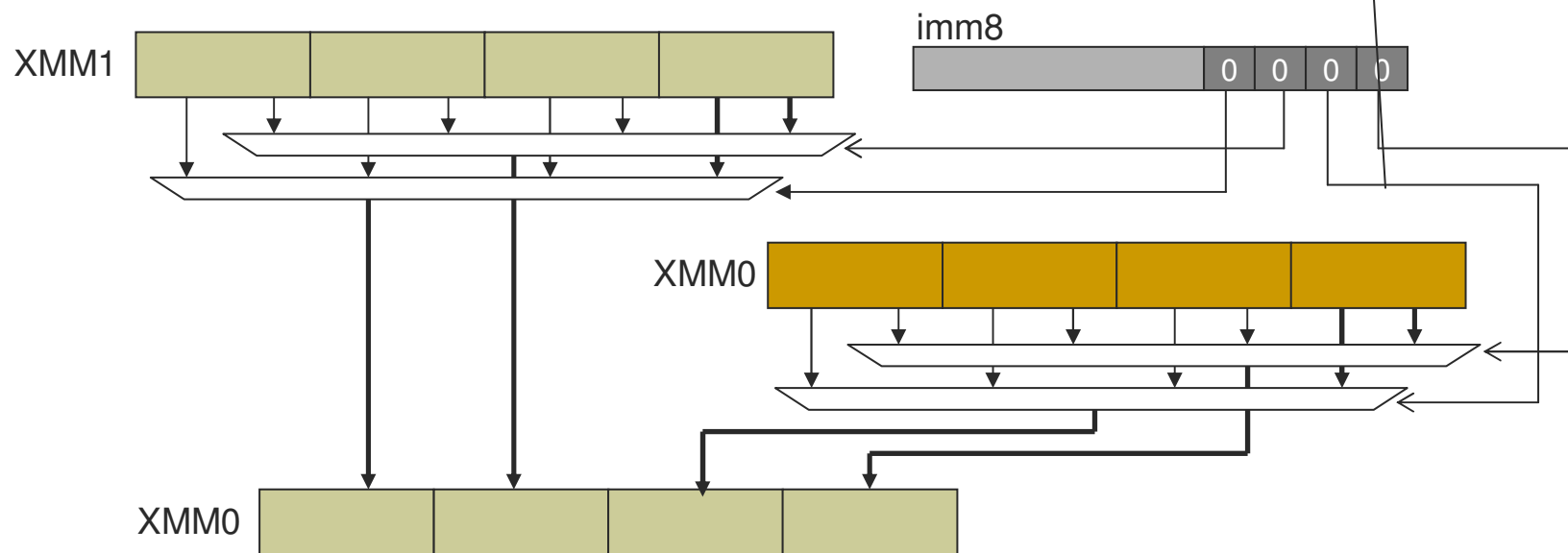
SHUFPS ; SHUFDPD

SHUFP[S|D]

S - single, D - double

SHUFPS xmmreg1, xmmreg2/mem128, imm8

Šířka 2 bity

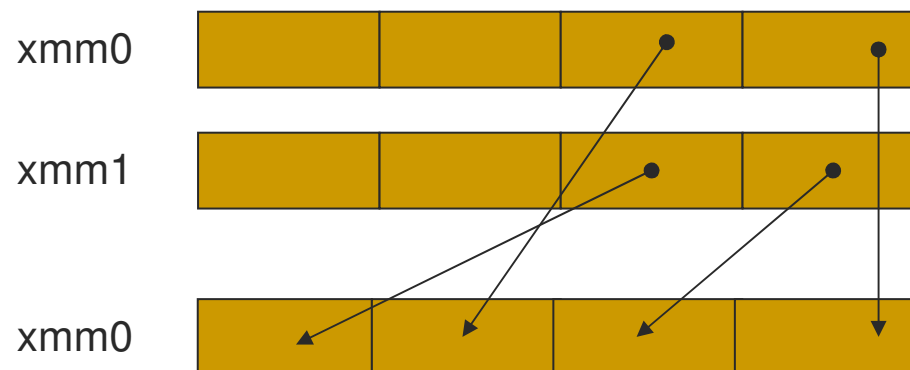


shufps xmm0, xmm1, 0x00

[Konverze II]

PUNPCKHPS, PUNPCKLPS; PUNPCKHPD, PUNPCKLPD

PUNPCKLPS xmm0, xmm1



Poznámka: konverze s H pracují s horní polovinou zdrojových operandů.

Konverze s D pracují s 64-bitovými hodnotami.

[Konverze III]

CVTPI2PS, CVTPS2PI; CVTPI2PD, CVTPD2PI
CVTSI2SS, CVTSS2SI; CVTSI2SD, CVTSD2SI

CVT[P|S]I2[P|S][S|D]

└── S - single, D - double
└── P - packed, S - scalar
└── konverze
celých čísel v mmx registru
na FP čísla do xmm registru

CVTPI2PS xmmreg1, mmreg2/mem64

CVT[P|S][S|D]2[P|S]I

└── P - packed, S - scalar
└── S - single, D - double
└── opačná konverze FP -> integer

CVTPS2PI xmmreg1, mmreg2/mem64

Poznámka: existují instrukce CVTT, které nahrazují zaokrouhlení useknutím (truncation)

[Jiné SIMD implementace]

- AltiVec
PowerPC, Motorola
- SunVis
Sparc, SUN
- PA-RISC Multimedia Instructions

Cell Broadband Engine

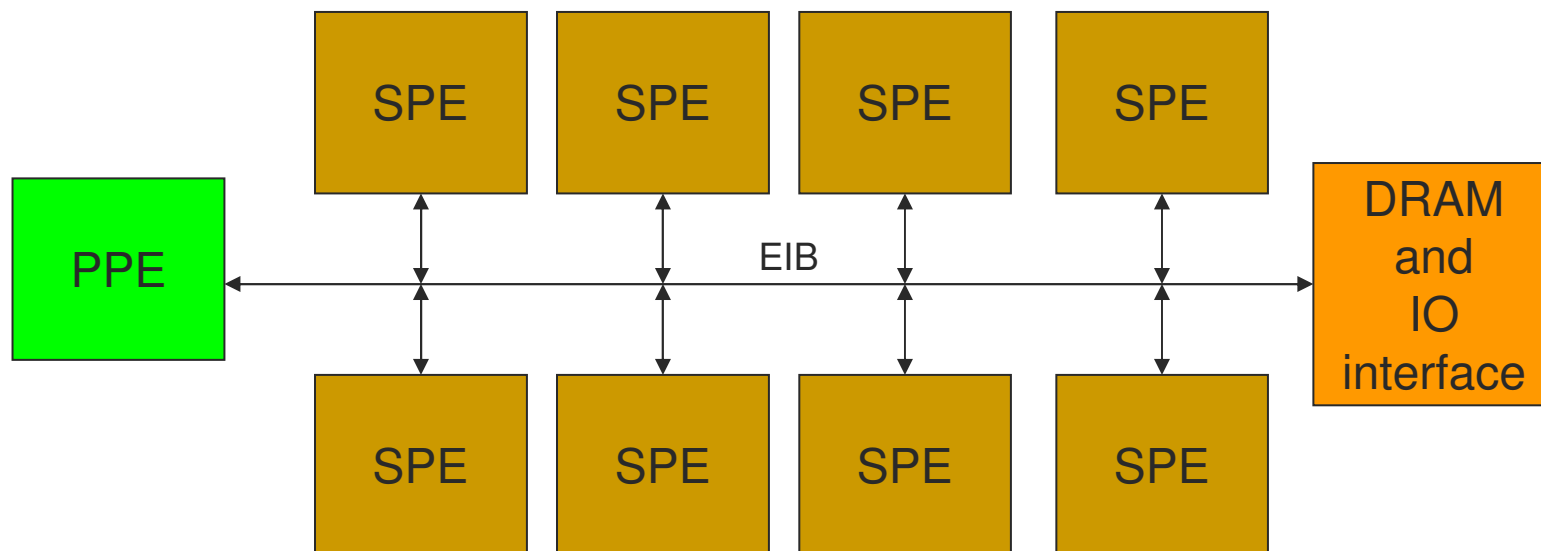
History

- 2000 IBM, SCEI/Sony Toshiba Alliance
- 2001 Design Center
- 2005 Disclosure of Technical Documentation

Highlights

- Supercomputer on a chip
- Multi-core processor (9 cores)
- 3.2 GHz clock frequency

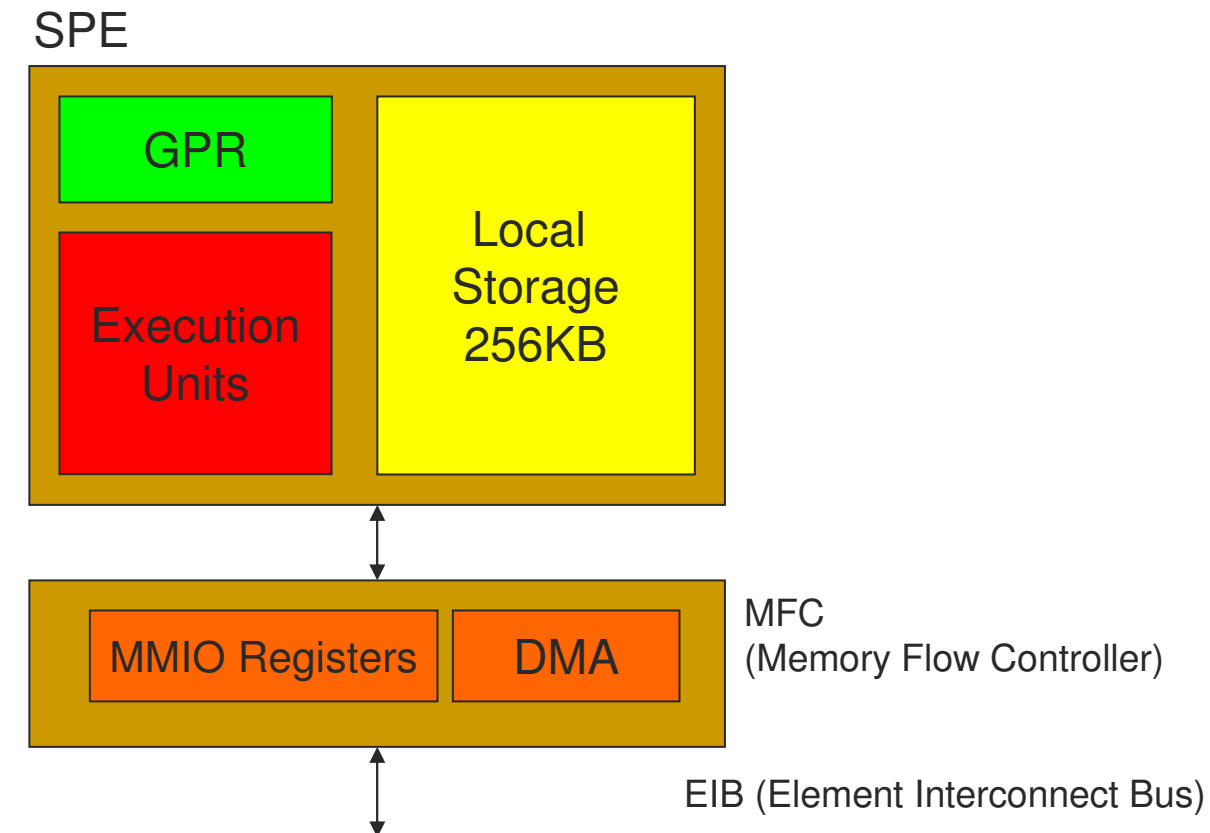
[Cell Broadband Engine]



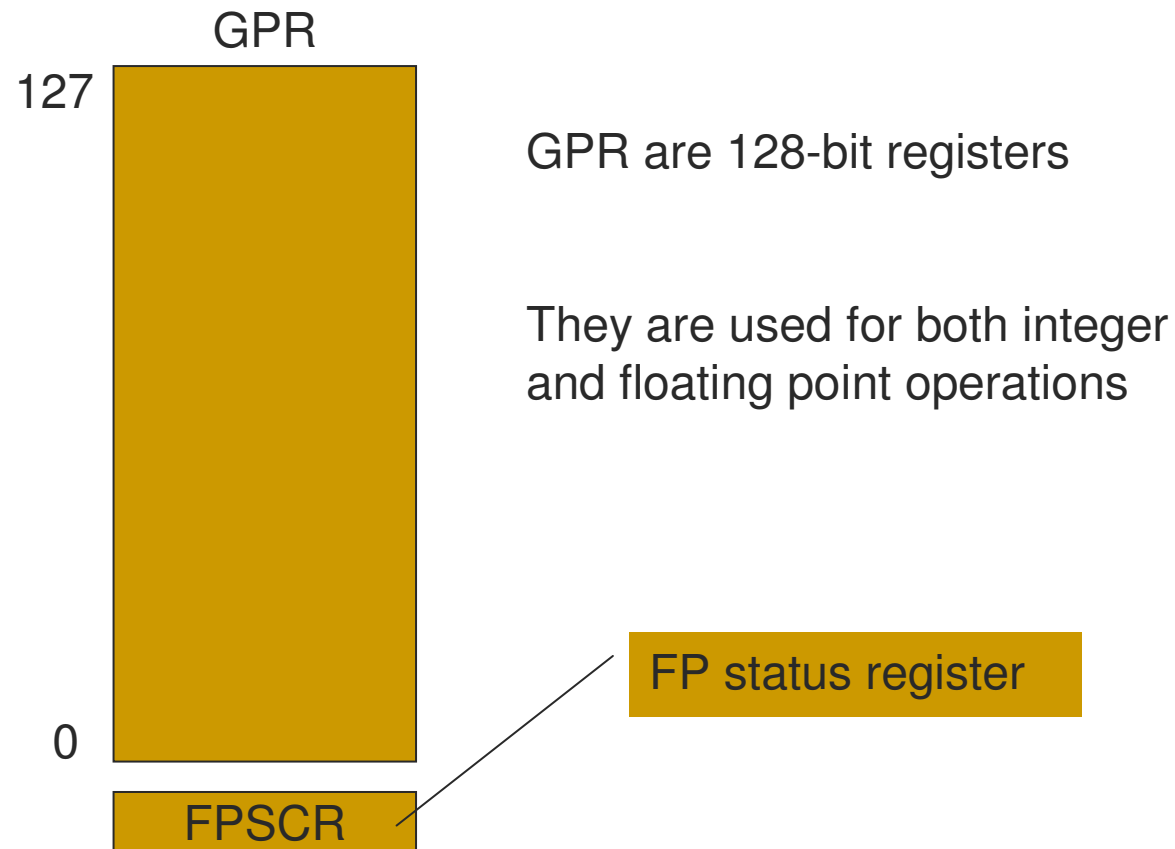
PowerPC Processor Element (PPE)

- General purpose processor
- 64-bit RISC, dual-thread
- Conforms to the PowerPC architecture version 2.02
- Alitvec (Vector/SIMD Multimedia Extensions)
- 32 integer GPR (64-bit registers),
- 32 floating point registers (64-bit registers)
- 32 vector registers (128-bit registers)

Synergic Processor Element (SPE)

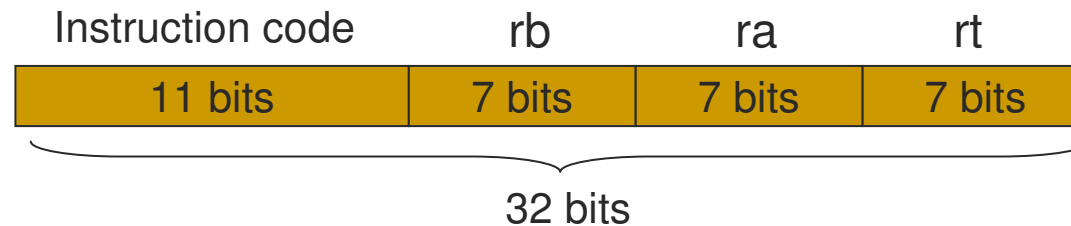


[SPE Registers]



[Instruction Format]

Three-operand arithmetic instructions



Example:

a \$1, \$2, \$3 ; \$1 ← \$2 + \$3

| 00011000000 | 0000011 | 0000010 | 0000001 |

[Integer Addition]

```
ah  rt,ra,rb      ; rt ← ra + rb (8 x halfword)
ahi rt,ra,value    ; rt ← ra + value (8 x halfword)

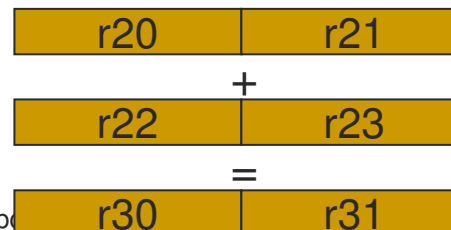
a  rt,ra,rb      ; rt ← ra + rb (4 x word)
ai rt,ra, value   ; rt ← ra + value (4 x word)

addx rt           ; rt ← ra + rb + rtlsb

cg rt,ra,rb      ; rt ← carry_out(ra + rb) (carry is lsb)
cg rt,ra,rb      ; rt ← carry_out(ra + rb + rtlsb)
```

64-bit addition

```
a    $31,$21,$23
cg   $30,$21,$23
addx $30,$20,$22
```



Arhitektura po

Skrbek

[Integer Subtaction]

```
sfh  rt,ra,rb      ; rt ← rb - ra (8 x halfword)
sfhi rt,ra,value    ; rt ← value - ra (8 x halfword)

sf  rt,ra,rb      ; rt ← rb - ra (4 x word)
sfi  rt,ra, value  ; rt ← value - ra (4 x word)

sfx rt,ra,rb      ; rt ← rb - ra - rtlsb

bg  rt,ra,rb      ; rt ← borrow_of(rb - ra)
bgx rt,ra,rb      ; rt ← borrow_of(rb - ra - rtlsb)
```

Task: explain usage of the bgx instruction.

[Integer Multiplication]

```
mpy rt,ra,rb ; rt ← ra * rb (word ← halfword * halfword)
mpyu rt,ra,rb ; rt ← ra * rb (unsigned)
```

```
mpyi rt,ra,value ; rt ← ra * value
mpyiu rt,ra,value ; rt ← ra * value (unsigned)
```

```
mpya rt,ra,rb,rc ; rt ← ra * rb + rc
                  ; word ← halfword * halfword + word
```

```
mpyh rt,ra,rb ; rt ← (ra >> 16) * rb (for 32-bit mpy)
mpys rt,ra,rb ; rt ← (ra * rb) >> 16
```

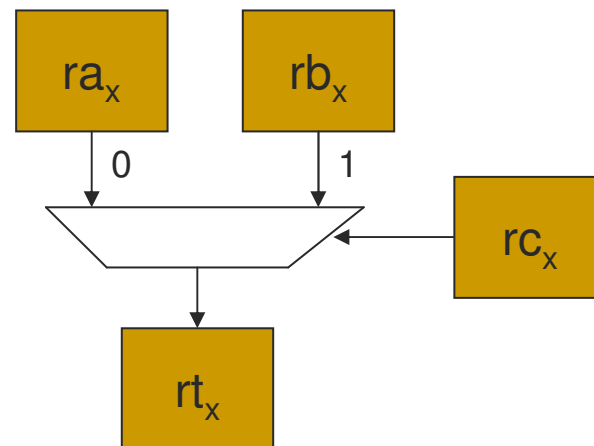
```
mpyhh rt,ra,rb ; rt ← (ra >> 16)*(rb >> 16)
mpyhhu rt,ra,rb ; rt ← (ra >> 16)*(rb >> 16) (unsigned)
mpyhha rt,ra,rb ; rt ← (ra >> 16)*(rb >> 16) + rt;
mpyhhou rt,ra,rb ; rt ← (ra >> 16)*(rb >> 16) + rt; (unsigned)
```

[Logical Instructions]

- AND
- AND with immediate
- AND with complement
- OR
- OR with immediate
- OR with complement
- OR cross (or for all word in a qword)
- XOR
- XOR with immediate
- NAND
- NOR
- Equivalent (XOR NOT)

[Special Logical Operations I]

`selb rt,ra,rb,rc ; bit selection`



The x index denotes the bit, on which the operation is applied

The `celb` instruction can be used for if-then-else or `?:` operator implementation

[Special Logical Operations II]

shufb rt, ra, rb, rc

Each byte of the rc register selects one byte from the concatenation of the ra and rb registers. The result is stored into a byte in the rt register. The resulting byte location corresponds to its selector in the rc register.

Three special values of the selector byte store one of the 00h, ffh, 80h values into the resulting byte.

[Shift and Rotate Instructions]

- Shift left (halfword, word, quadword)
- Rotate left/right
- Rotate (arithmetic) left/right

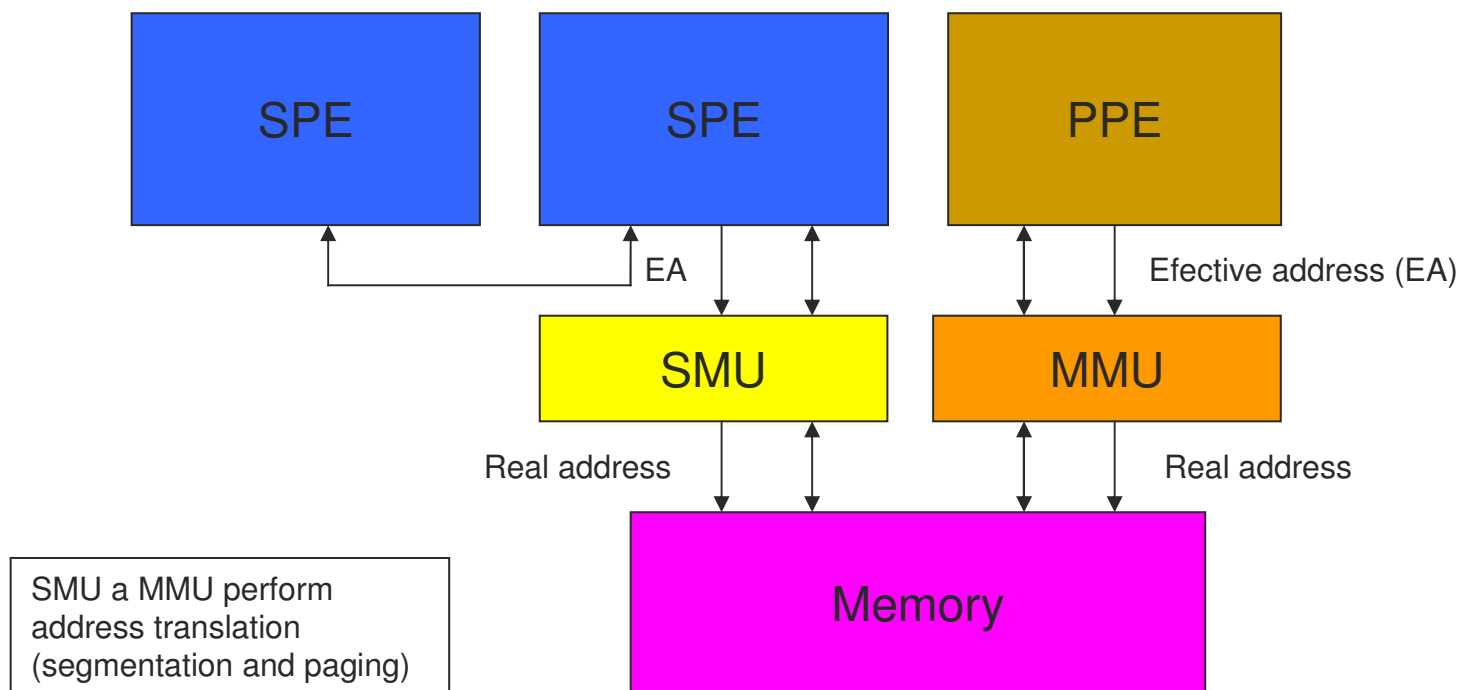
[Floating Point Operations]

- ADD, SUB, MUL (Single, Double)
- Multiply and add (Single, Double)
- Multiply and subtract (Single, Double)
- Division implemented by the reciprocal function (steps: estimate, interpolate, Newton-Raphson)
- Square root implemented by reciprocal square root (steps: estimate, interpolate, Newton-Raphson)

[Communication (SPE \leftrightarrow PPE, SPE \leftrightarrow SPE)]

- Mailboxes
- Signals

DMA transfers (SPE ↔ MEMORY, SPE ↔ SPE)

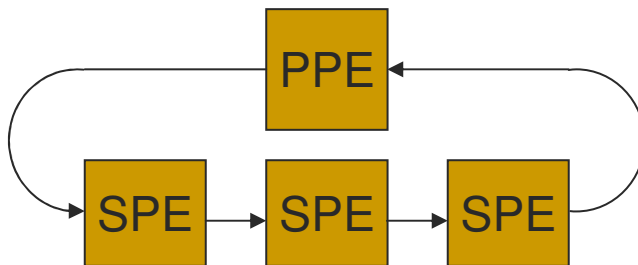


[Programming]

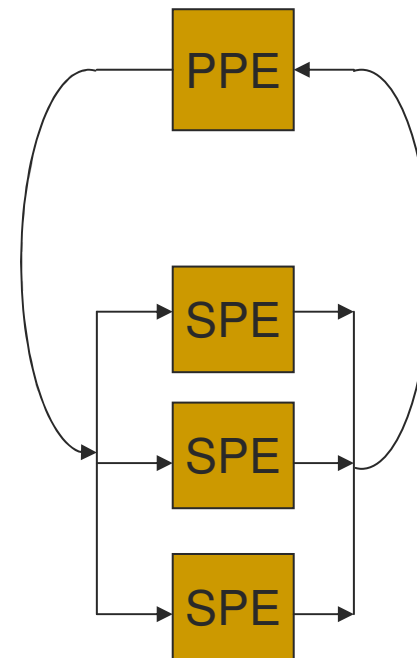
- GCC based tools for PPE and SPE
- Software Development Kit
- Cell Broadband Engine Simulator

[Application partitioning]

Pipelining



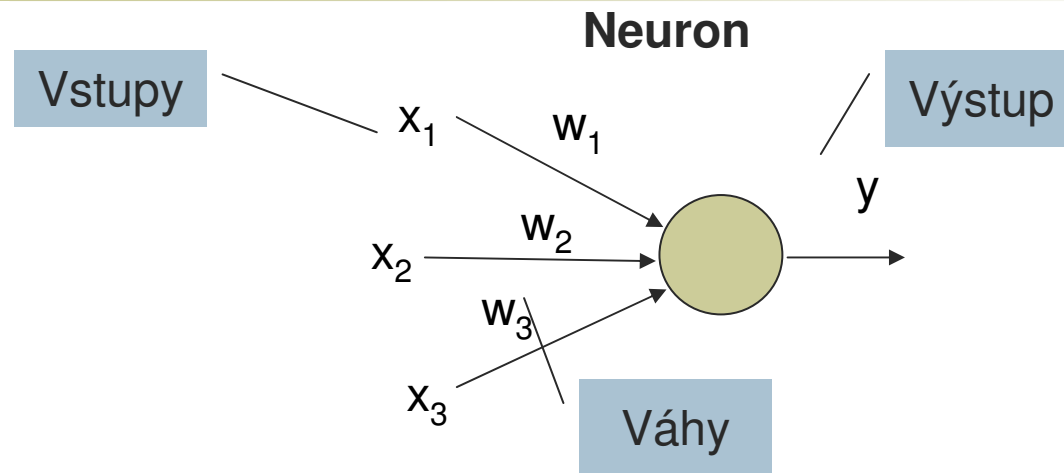
Parallel processing



[Implementace neuronových sítí]

Neuročipy

[Neuronové sítě]



Perceptron
(Frank Rosenblatt,
50 léta 20. století)

RBF Neuron

$$y = S\left(\sum_{i=1}^N w_i x_i + \Theta\right)$$

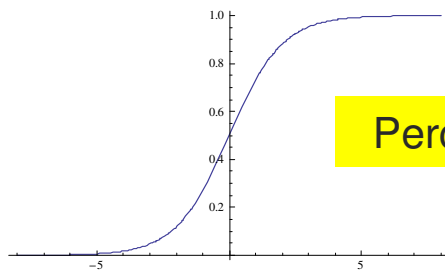
$$y = G\left(\sqrt{\sum_{i=1}^N (x_i - w_i)^2}\right)$$

Nelineární výstupní
funkce

Práh

Výstupní funkce neuronu

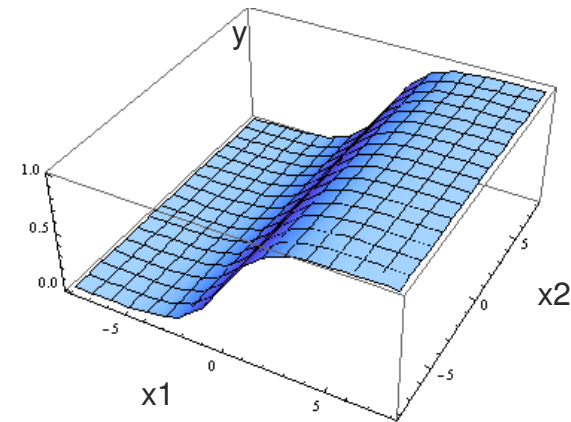
Výstupní funkce - Sigmoida



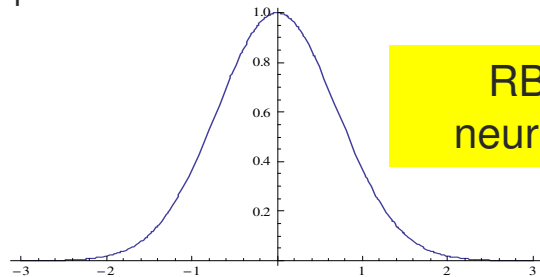
Perceptrony

$$y = S(x) = \frac{1}{1 + e^{-\gamma \cdot x}}$$

Odezva neuronu se dvěma vstupy

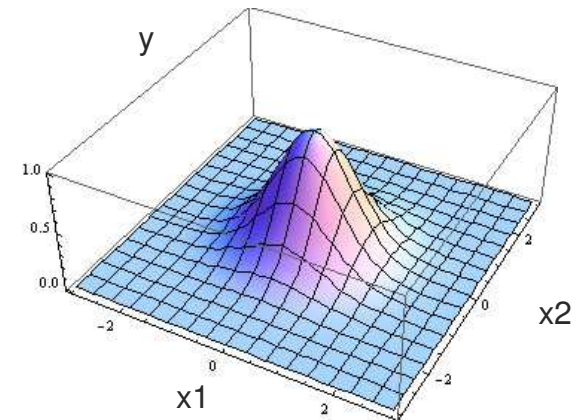


Výstupní funkce – Gaussova křivka



RBF
neurony

$$y = G(x) = e^{-\frac{x^2}{2\sigma^2}}$$



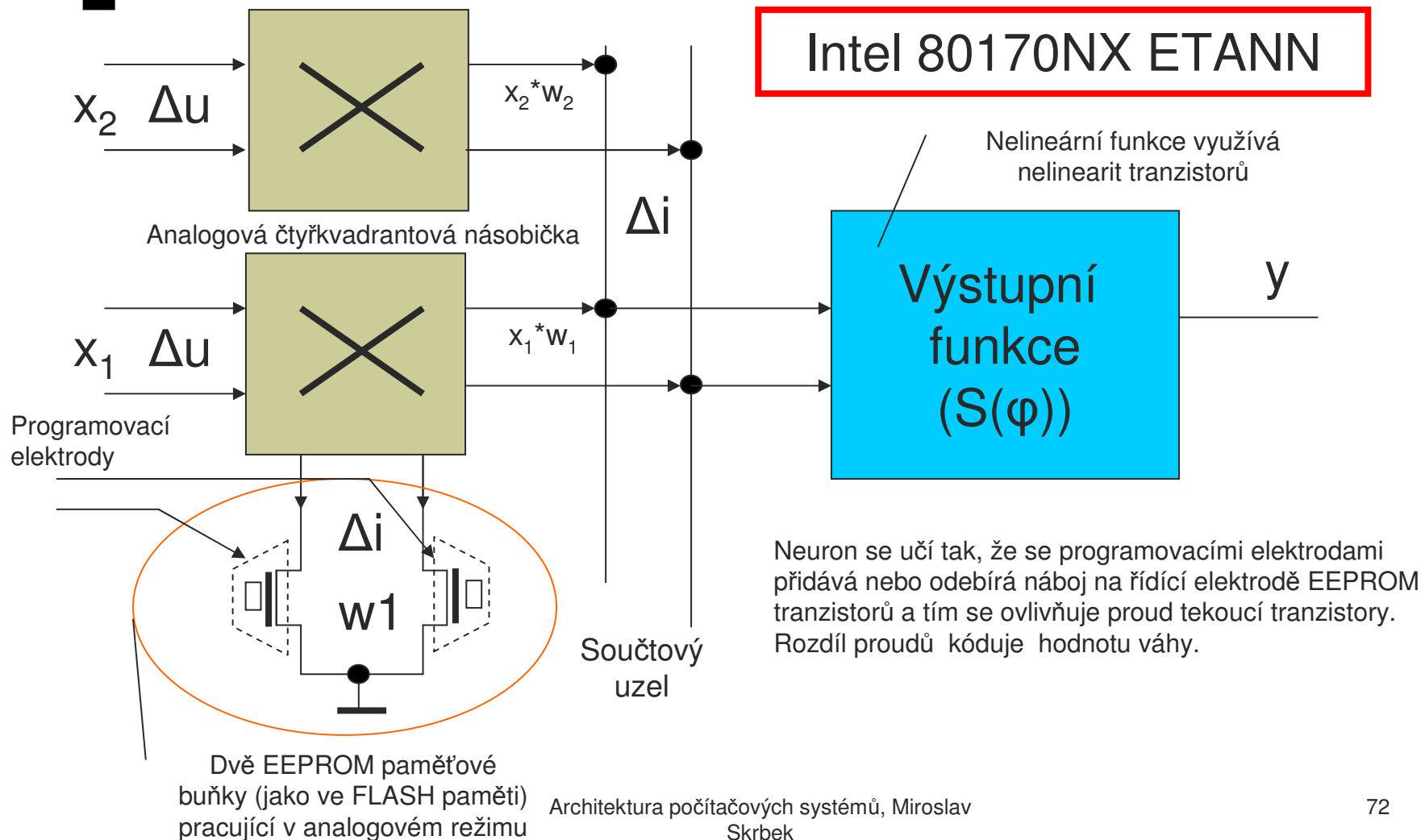
[Neuročipy]

Neuročipy jsou integrované obvody pro implementaci neuronových sítí. Integrují několik jednotek, desítek maximálně stovek neuronů. Implementují vztahy na předchozích slidech.

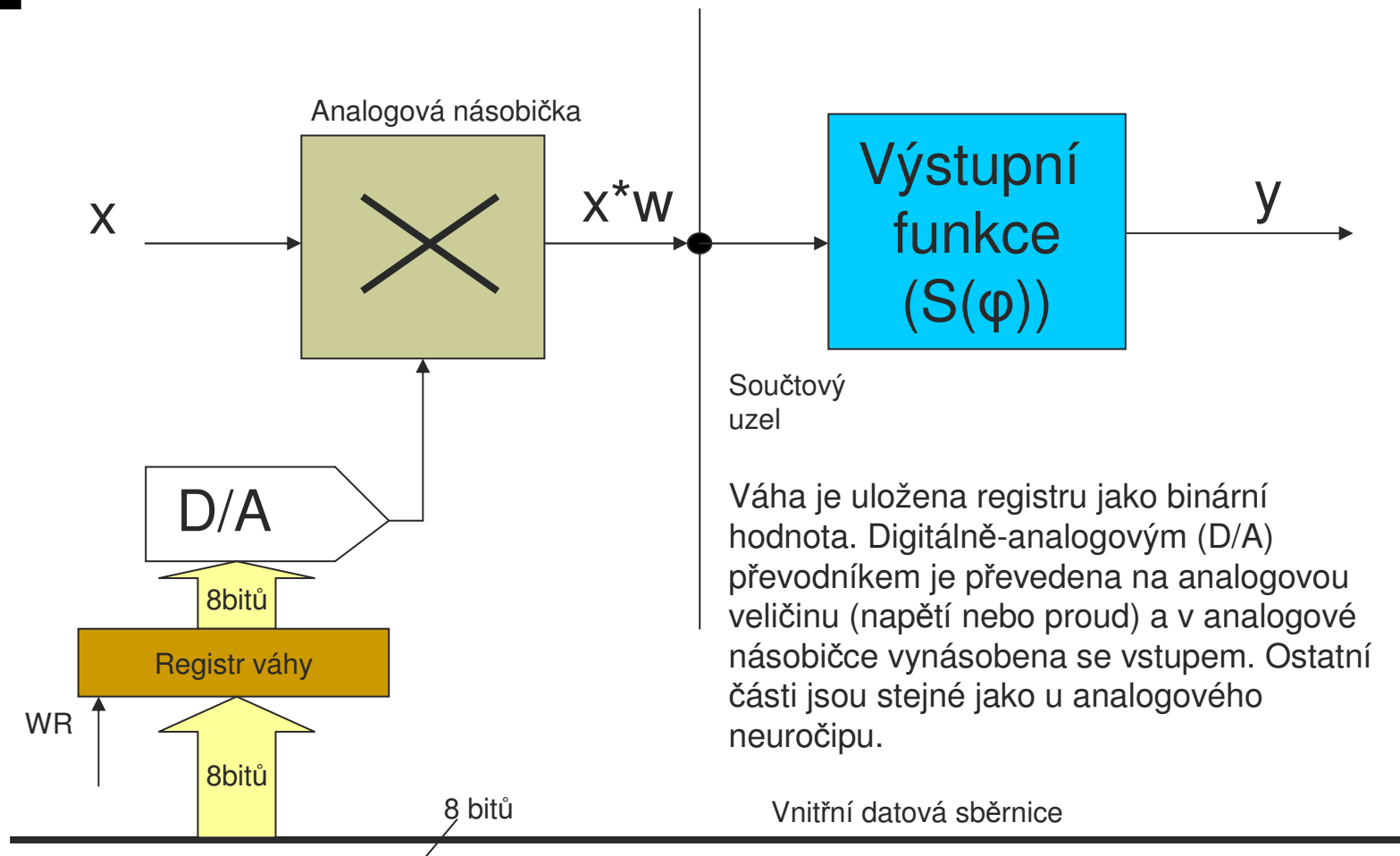
Podle typu implementace rozlišujeme

- *analogové* - hodnoty jsou reprezentovány spojitou veličinou – napětí nebo proud)
- *číslicové* – hodnoty binárně kódované jako v počítačích
- *hybridní* – výpočet probíhá analogově, uložení dat je číslicové
- *pulsní* – hodnoty jsou kódovány do frekvence a časového posuvu impulzů (blízké lidským neuronům)

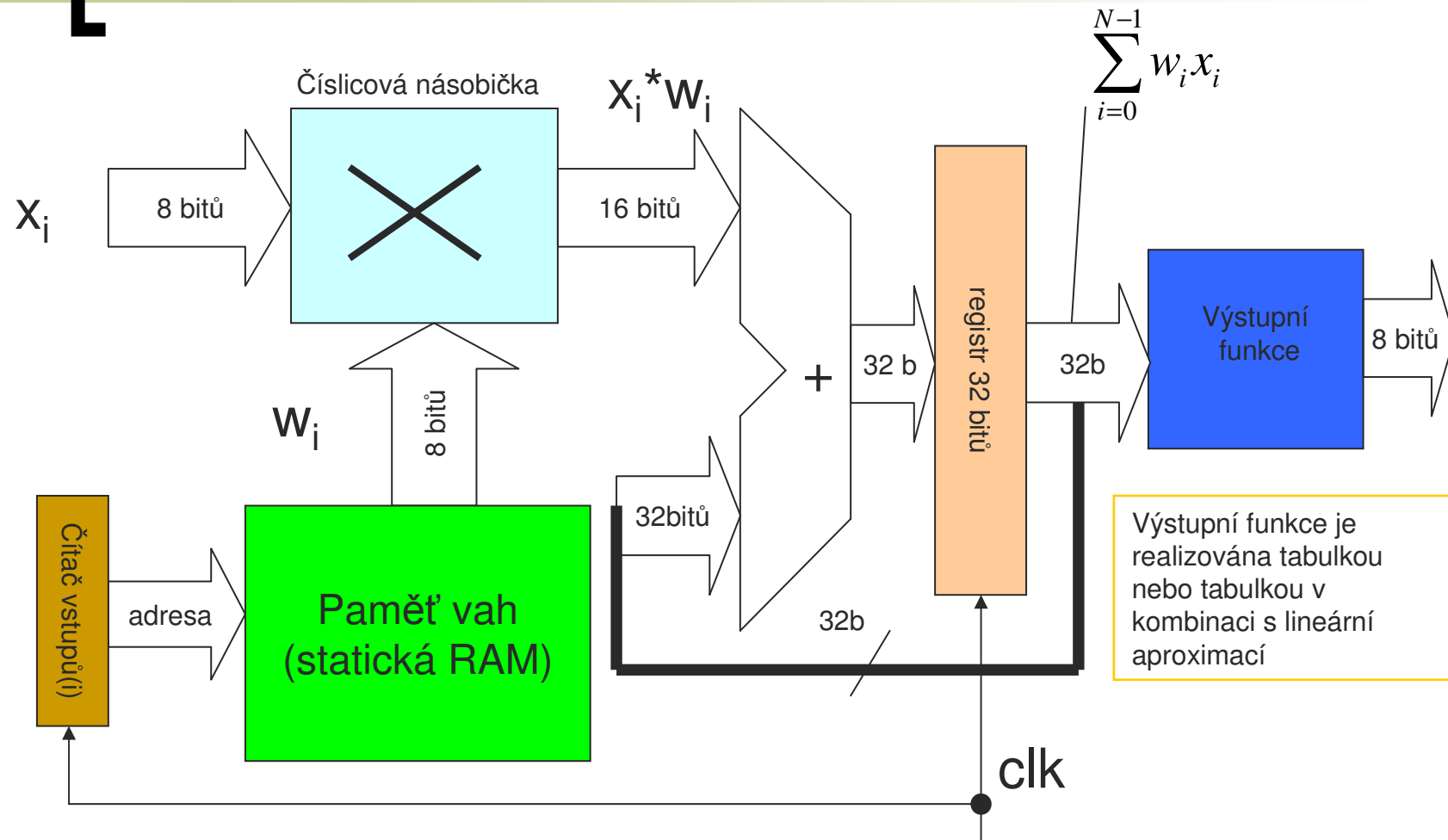
[Analogové neuročipy]



[Hybridní neuročip]



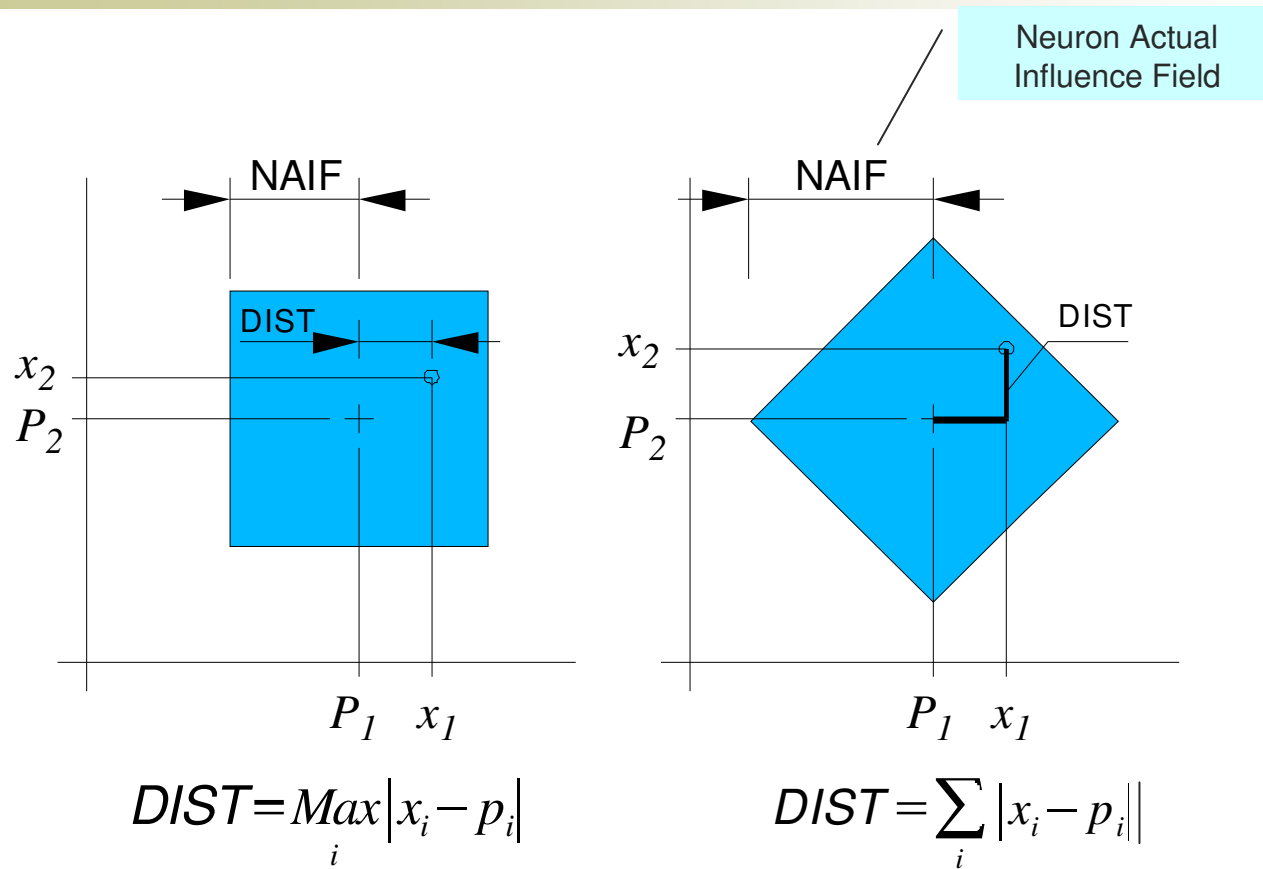
Číslicové neuročipy s perceptrony



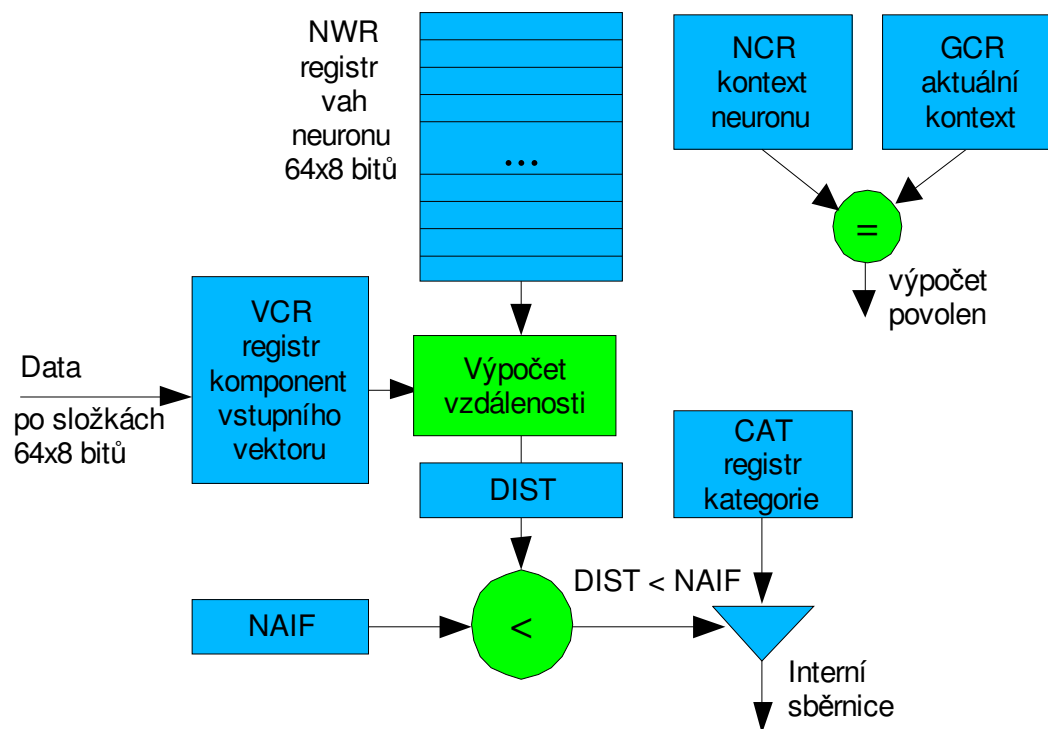
[ZISC[®]36 - neuročip]

- 36 neuronů typu RBF v jednom čipu
- 64 vstupů
- neomezeně rozšiřitelné v počtu neuronů
- vestavěný algoritmus učení
- rychlé učení i vybavování ($\sim \mu\text{s}$)
- produkt IBM (Francie)

Metriky neuronů



Struktura neuronu



[Klasifikace]

- Zapsat 64 složek klasifikovaného vektoru
- Neurony s $DIST < NAIF$ se stanou aktivní
- Přechází výsledek klasifikace

Identifikován
(identified)

Aktivní neurony
patří do stejné
kategorie

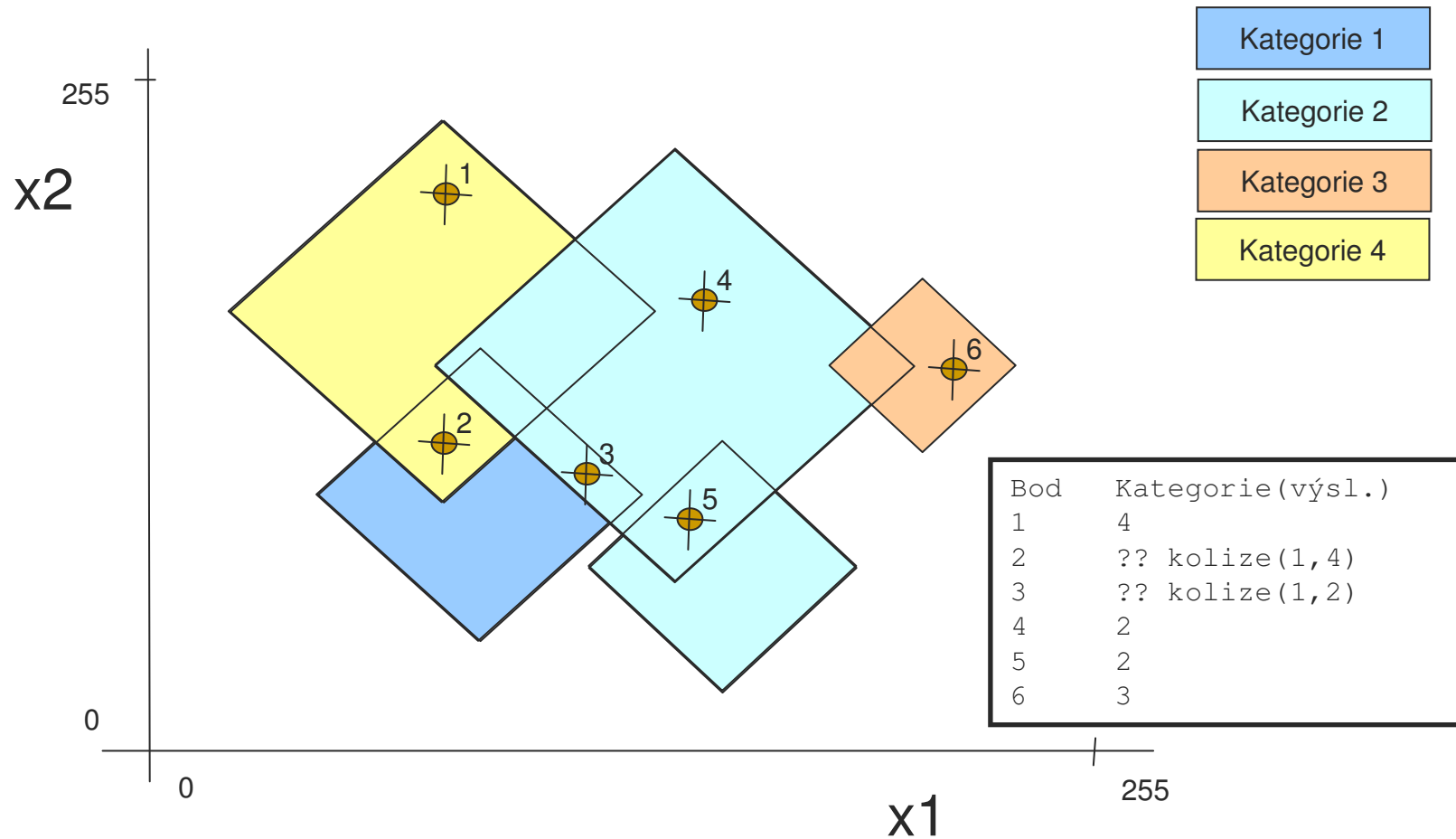
Neklasifikován
(unclassified)

Aktivní neurony
patří do různých
kategorií

Neklasifikován
(no activity)

Žádný z neuronů
není aktivní

Příklad klasifikace pro dva vstupy



[Učení]

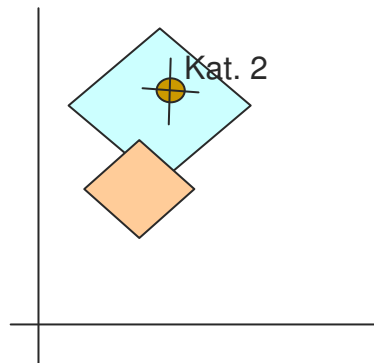
- Zapsat 64 (nebo méně) složek učeného vektoru
- Zapsat kategorii

Příklad trénovací množiny pro dva vstupy

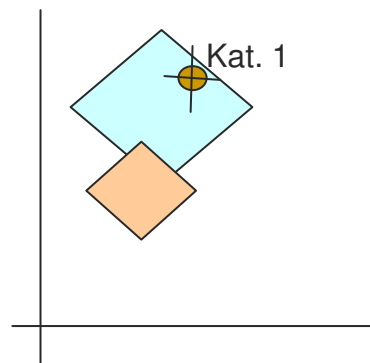
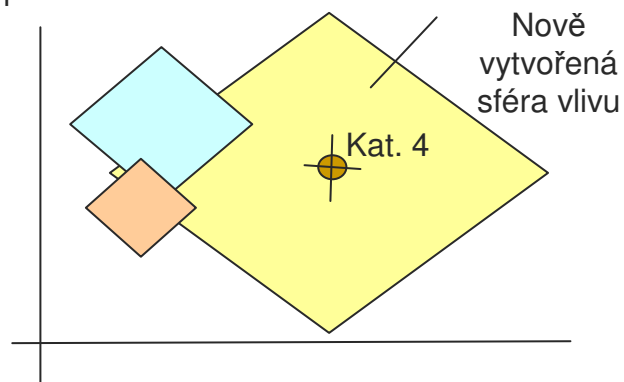
x1	x2	kategorie
5	10	1
200	30	2
50	40	1
100	25	3

...
└──┬──┘
Složky Kategorie

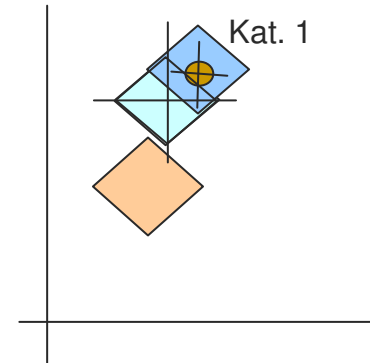
[Princip učení]



Učený vzor (viz. bod) polohou spadá do kategorie 2 a je to vzor kategorie 2, proto se nestane nic.



Učený vzor (viz. bod) polohou spadá do kategorie 2 ale je to vzor kategorie 1. NAIF (sféra vlivu) neuronu kategorie 2 se zmenší a v učeném bodě se vytvoří nový neuron se sférou vlivu omezenou na minimální vzdálenost od středů všech ostatních neuronů.



Učený vzor (viz. bod) polohou nespadá do sféry vlivu žádného neuronu, proto se vytvoří nový neuron se středem v učeném bodě a se sférou vlivu omezenou na minimální vzdálenost od středů všech ostatních neuronů.

[Implementační platformy neuronových sítí]

- Neuročipy (zákaznické integrované obvody)
- Signálové procesory (DSP)
- Obvody FPGA (Field-programmable Gate Array)
- Simulace neuronových sítí na počítačích

[Grafické procesory (GPU)]

- Slouží pro akceleraci grafických výpočtů
- Jsou integrovány přímo do grafických karet
- Využívají masivní paralelismus
- Počty tranzistorů šplhají ke 3 miliardám
- Poskytují programovatelné vertex shadery (manipulace s 3D modely) a pixel shadery (rasterizace)
- Programování přes OpenCL, CUDA, DirectX
- Používají se i pro negrafické výpočty

[NVIDIA Tesla]

- 1.31 teraflops(double)
3.95 teraflops(float)
(TESLA K20X)
- 2668 CUDA Cores
- 6GB Memory
- 250GB/s přenosová rychlost pro paměť

[Princip programování]

- Pro skupinu vláken píšeme jeden program v jazyce C
- Program, který je spuštěn v některém vlákně dostane identifikátor vlákna, podle kterého se příkazy if určí konkrétní činnost programu v daném vlákně.
- Maximálního výkonu se obvykle dosahuje pokud všechna vlákna zpracovávají stejný kód (SIMD), nicméně se architektura jeví jako MIMD.
- Vyššího výkonu se dosahuje pro výpočty v omezené přesnosti (SP, float, 32-bitů) než pro datový typ double (DP, double, 64-bitů)

[Příklad programu]

```
kernel void sectiVektory(global const float* a,  
global const float* b, global float* c, int size) {  
  
    int id = get_global_id(0);  
    if (id >= size) {  
        return;  
    }  
    // sečti vektory po složkách  
    c[id] = a[id] + b[id];  
}
```