

Architektura počítačů I

UAI/698 Přednášky

ver 1.1.2

Miroslav Skrbek
mskrbek@prf.jcu.cz

Ústav aplikované informatiky

Přírodovědecká fakulta

Jihočeské univerzity v Českých Budějovicích

Platné pro šk.r. 2024/2025

Literatura

- [1] David A. Patterson and John L. Hennessy. *Computer Organization and Design. The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design.
- [2] David A. Patterson and John L. Hennessy. *Computer Architecture. A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design.

Reprezentace znaků a čísel v počítači

Zápis čísel - historie

Zářezy na holi - II (2), IIII (4), IIII (~~5~~), IIII III (~~8~~)

Římské číslice - I(1), V(5), X(10), L(50), C(100), D(500), M(1000)

Příklady: II (2), IV (4), IX (9), CCLXI (256), MMXII (2012),

Problematické počítání, pomůcka abakus (počítadlo)

Poziční soustavy - 123_{10} (sto dvacet tři), 1110_2 (14)



Arabské číslice

Problém nuly, dlouho neřešen, bez nuly náchylné na chybné čtení čísla

Poziční soustavy I

Číslo v poziční soustavě z-adické soustavě

$$A_z = (a_n a_{n-1} \dots a_1 a_0, a_{-1} + \dots a_{-m})_z; a_i, n, m \in \mathbb{N}$$

Řádová čárka (nebo tečka)

Příklady: $(11345)_{10}$, $(1100.011)_2$, $(ABCD.1234)_{16}$

$$\begin{aligned} A_z &= a_n z^n + a_{n-1} z^{n-1} + \dots + a_1 z^1 + a_0 + a_{-1} z^{-1} + \dots a_m z^m = \\ &= \sum_{i=0}^n a_i z^i + \sum_{i=1}^m a_{-i} z^{-i} = \sum_{i=-m}^n a_i z^i \end{aligned}$$

a_i je z-adická číslice

z je základ soustavy

n je nevyšší řád s nenulovou číslicí

m je nejnižší řád s nenulovou číslicí

Dvojková (binární) soustava

$$z = 2$$

Mocnina	Hodnota
2^{-2}	0,25
2^{-1}	0,5
2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256

Mocnina	Hodnota
2^9	512
2^{10}	1024 (1Ki)
2^{11}	2048
2^{12}	4096
2^{13}	8192
2^{14}	16384
2^{15}	32768
2^{16}	65536 (64Ki)
2^{20}	1024^2 (1 Mi)
2^{30}	1024^3 (1 Gi)
2^{40}	1024^4 (1 Ti)

$$A = 1001101_2$$

$$\begin{aligned} A &= 1 \cdot 2^6 + 0 \cdot 2^5 + \\ &0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + \\ &0 \cdot 2^1 + 1 \cdot 2^0 = \\ &64 + 8 + 4 + 1 = 77 \end{aligned}$$

Ki (kibi)
Mi (mebi)
Gi (gibi)

Příklady

Převeďte

$$101110111_2 \rightarrow ??_{10}$$

$$111101101_2 \rightarrow ??_{10}$$

$$101_2 \rightarrow ??_{10}$$

$$58_{10} \rightarrow ??_2$$

$$101_{10} \rightarrow ??_2$$

Algoritmus převodu desítková → dvojková soustava čísla celá kladná

z (základ cílové číselné soustavy)

156 : 2 = 78	zbytek → 0	nejnižší řád (LSB)
78 : 2 = 39	zbytek → 0	least significant bit
39 : 2 = 19	zbytek → 1	
19 : 2 = 9	zbytek → 1	
9 : 2 = 4	zbytek → 1	
4 : 2 = 2	zbytek → 0	
2 : 2 = 1	zbytek → 0	
1 : 2 = 0	zbytek → 1	nejvyšší řád (MSB)

most significant bit

$$156_{10} \rightarrow 10011100_2$$

Poznámka: Algoritmus je platný i pro převod do jiných číselných soustav, pouze dělíme jiným základem.

Algoritmus převodu desítková \rightarrow dvojková soustava
zlomková část na intervalu $<0,1$

$0,375 * 2 = 0,75$ celá část $\rightarrow 0$ nejvyšší řád (2^{-1})
 $0,75 * 2 = 1,5$ celá část $\rightarrow 1$
 $0,5 * 2 = 1$ celá část $\rightarrow 1$ nejnižší řád (2^{-3})

$$0,375_{10} \rightarrow 0,011_2$$

Poznámka: Algoritmus je platný i pro převod do jiných číselných soustav, pouze násobíme jiným základem.

Sčítání čísel ve dvojkové soustavě

Součet

a/b	0	1
0	0	1
1	1	10

Přenos do
vyššího řádu

$$\begin{array}{r} 10011101_2 \\ 10110001_2 \\ \hline 101001110_2 \end{array}$$

Arrows indicate carry propagation from the bottom row to the top row.

Přenos do
vyššího řádu

Násobení čísel ve dvojkové soustavě

Součin

a/b	0	1
0	0	0
1	0	1

Používáme stejný algoritmus, který nás naučili na základní škole pro desítkovou soustavu

$$\begin{array}{r} 1001_2 \\ 110_2 \\ \hline 0000 \\ 1001 \\ 1001 \\ \hline 110110_2 \end{array}$$

Dělení čísel ve dvojkové soustavě

$$10101_2 : 110_2 = 011_2$$

1010

-110

1001

-110

011


Používáme stejný algoritmus, který nás naučili na základní škole pro desítkovou soustavu.

Všimněte si, že je snazší uhodnout, kolikrát se dělitel vejde do dělence (buď se vejde, nebo ne). V desítkové soustavě musíme ještě uhodnout, kolikrát (např. osmkrát).

Algoritmus převodu dvojková \rightarrow desítková soustava *čísla celá kladná*

Princip je shodný s převodem z desítkové do dvojkové soustavy, ale operace je nutno provádět ve dvojkové soustavě.

10011011101	:	1010	=	1111100	zbytek	\rightarrow	101	(5)
1111100	:	1010	=	1100	zbytek	\rightarrow	100	(4)
1100	:	1010	=	1	zbytek	\rightarrow	10	(2)
1	:	1010	=	0	zbytek	\rightarrow	1	(1)



Algoritmus převodu dvojková \rightarrow desítková zlomková část na intervalu $<0,1$)

Princip je shodný s převodem z desítkové do dvojkové soustavy, ale operace je nutno provádět ve dvojkové soustavě.

0,0011010011	*	1010	=	10,0000111110	celá část \rightarrow 10	(2)
0,0000111110	*	1010	=	0,1001101100	celá část \rightarrow 0	(0)
0,1001101100	*	1010	=	110,0000111000	celá část \rightarrow 110	(6)
0,0000111000	*	1010	=	0,1000110000	celá část \rightarrow 0	(0)
0,1000110000	*	1010	=	101,0111100000	celá část \rightarrow 101	(5)
0,0111100000	*	1010	=	100,1011000000	celá část \rightarrow 100	(4)
0,1011000000	*	1010	=	110,1110000000	celá část \rightarrow 110	(6)
0,1110000000	*	1010	=	1000,1100000000	celá část \rightarrow 110	(8)
0,1100000000	*	1010	=	111,1000000000	celá část \rightarrow 111	(7)
0,1000000000	*	1010	=	101,0000000000	celá část \rightarrow 101	(5)

$0,0011010011_2 \rightarrow 0,2060546875_{10}$

Šestnáctková soustava

$$z = 16$$

Tuto tabulku budete znát z paměti

Dekadicky	Binárně	Hexadecimálně
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Dekadicky	Binárně	Hexadecimálně
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

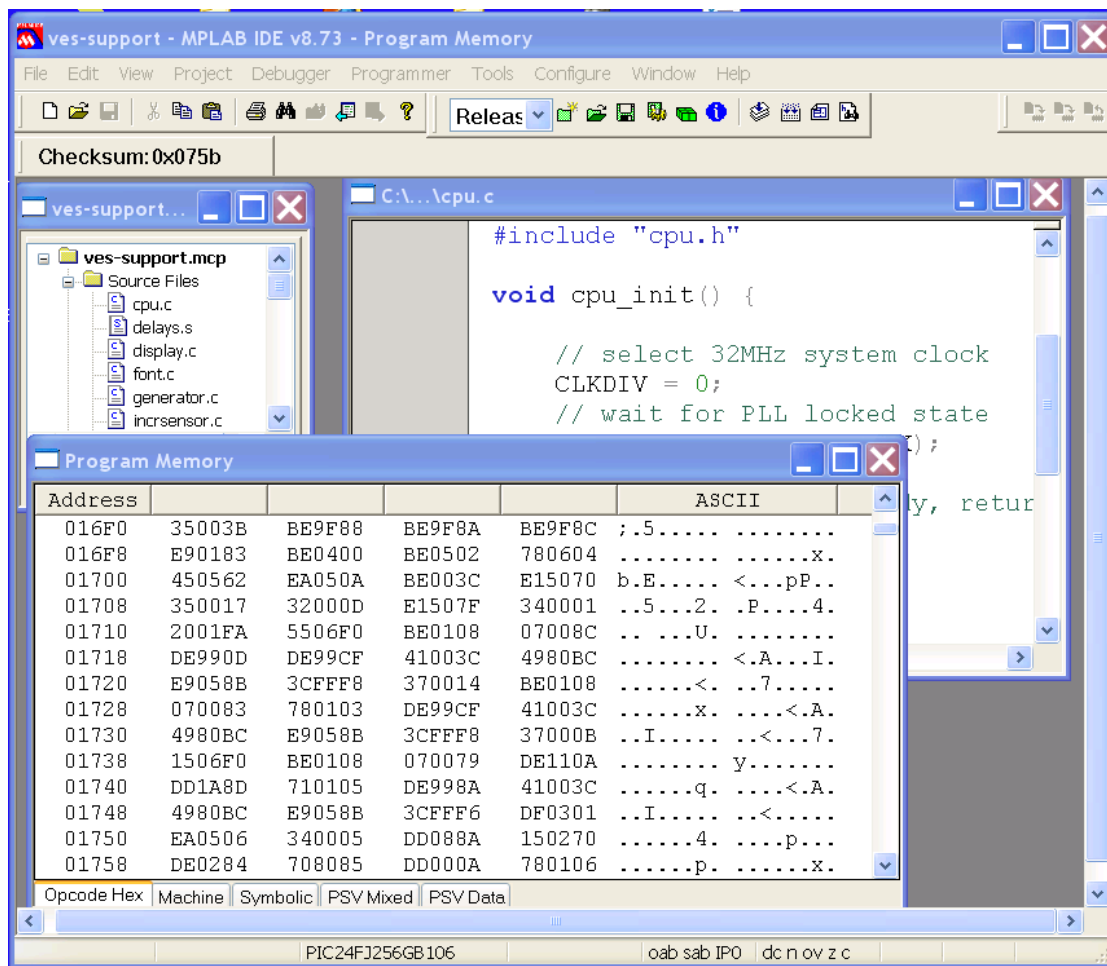
Příklady

$$11_{16} = 1 \cdot 16^1 + 1 \cdot 16^0 = 17_{10}$$

$$FF_{16} = 15 \cdot 16^1 + 15 \cdot 16^0 = 255_{10}$$

$$A5_{16} = 10 \cdot 16^1 + 5 \cdot 16^0 = 165_{10}$$

Výpis paměti v šestnáctkové soustavě – současný vývojový nástroj mikrokontroléry



The screenshot shows the MPLAB IDE v8.73 interface. The main window displays the source code for `C:\...\cpu.c`, which includes `"cpu.h"` and defines a `cpu_init()` function. The `Program Memory` window is open, showing a hexadecimal dump of memory. The dump is organized into columns for Address, Hex, and ASCII. The memory dump shows the following data:

Address	Hex	Hex	Hex	Hex	ASCII
016F0	35003B	BE9F88	BE9F8A	BE9F8C	;.5.....
016F8	E90183	BE0400	BE0502	780604x.
01700	450562	EA050A	BE003C	E15070	b.E..... <...pP..
01708	350017	32000D	E1507F	340001	..5...2. .P....4.
01710	2001FA	5506F0	BE0108	07008CU.....
01718	DE990D	DE99CF	41003C	4980BC<.A...I.
01720	E9058B	3CFFF8	370014	BE0108<. ..7.....
01728	070083	780103	DE99CF	41003Cx.<.A.
01730	4980BC	E9058B	3CFFF8	37000B	..I..... ..<...7.
01738	1506F0	BE0108	070079	DE110A y.....
01740	DD1A8D	710105	DE998A	41003Cq.<.A.
01748	4980BC	E9058B	3CFFF6	DF0301	..I..... ..<.....
01750	EA0506	340005	DD088A	1502704.p...
01758	DE0284	708085	DD000A	780106p.x.

The `Program Memory` window also shows the `Opcode Hex` tab selected, and the `Machine` tab is active. The status bar at the bottom indicates the target device is `PIC24FJ256GB106`.

Příbuzné soustavy

Příbuzné číselné soustavy jsou takové číselné soustavy, kde základ jedné soustavy je mocninou základu druhé soustavy.

Příklady

$z=10, z=100, z=1000, \dots$

$z=3, z=9, z=27, \dots$

$z=2, z=4, z=8, z=16, \dots$

Převod je snadný, ukážeme si to mezi dvojkovou a šestnáctkovou soustavou:

1011		0001		0010		1010		0010		₂
B		1		2		A		2		₁₆

Číslo ve dvojkové soustavě rozdělíme na skupiny po čtyřech číslicích a čtveřice převedeme na hexadecimální číslice. Pro opačný převod pouze zapíšeme hexadecimální číslice binárně. U čtyřkové soustavy vytváříme dvojice a u osmičkové trojice.

Příklady

Převeďte

$10100010010101001.01010_2 \rightarrow ??_{16}$

$101010100111110.1101010_2 \rightarrow ??_8$

$1010101001000.101101010_2 \rightarrow ??_4$

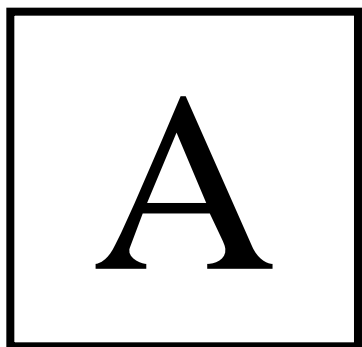
$A1B2C3D4.E5F6_{16} \rightarrow ??_2$

$347345.767337_8 \rightarrow ??_2$

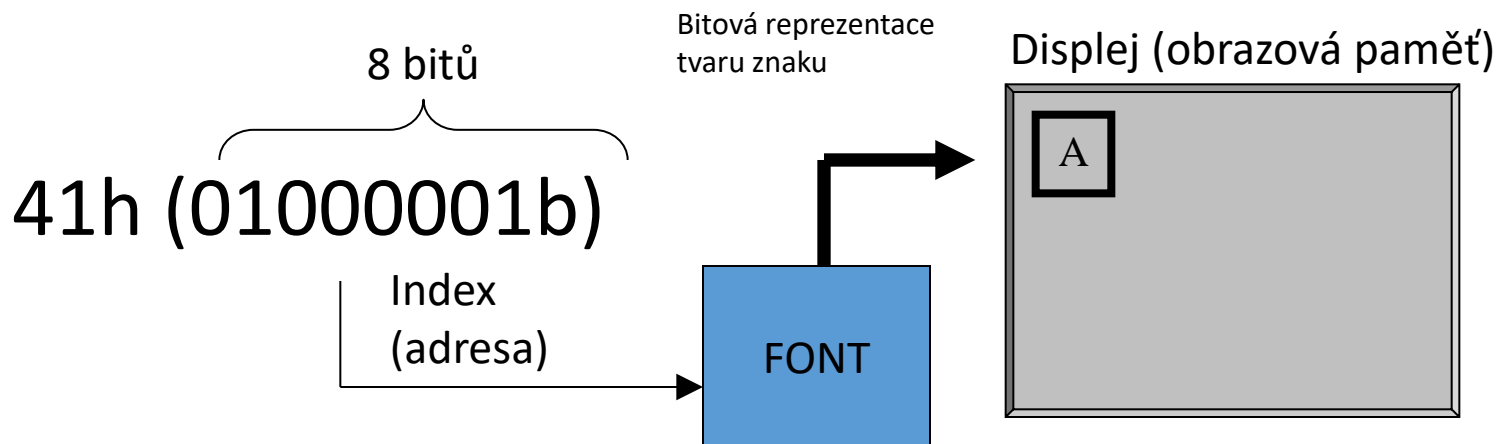
$1322230.2110_4 \rightarrow ??_2$

Kódování znaků

ASCII (American Standard
Code for Information
Interchange)



41h
(01000001b)



ASCII tabulka

<div><div>b₇ →</div><div>b₆ →</div><div>b₅ →</div><div><div>Column →</div><div>Row ↓</div></div></div>					0	0	0	0	1	0	1	1	1	1	1	1	
Bits					b ₄ ↓	b ₃ ↓	b ₂ ↓	b ₁ ↓		0	1	2	3	4	5	6	7
	0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p				
	0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q				
	0	0	1	0	2	STX	DC2	"	2	B	R	b	r				
	0	0	1	1	3	ETX	DC3	#	3	C	S	c	s				
	0	1	0	0	4	EOT	DC4	\$	4	D	T	d	t				
	0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u				
	0	1	1	0	6	ACK	SYN	&	6	F	V	f	v				
	0	1	1	1	7	BEL	ETB	'	7	G	W	g	w				
	1	0	0	0	8	BS	CAN	(8	H	X	h	x				
	1	0	0	1	9	HT	EM)	9	I	Y	i	y				
	1	0	1	0	10	LF	SUB	*	:	J	Z	j	z				
	1	0	1	1	11	VT	ESC	+	;	K	[k	{				
	1	1	0	0	12	FF	FC	,	<	L	\	l					
	1	1	0	1	13	CR	GS	-	=	M]	m	}				
	1	1	1	0	14	SO	RS	.	>	N	^	n	~				
	1	1	1	1	15	SI	US	/	?	O	_	o	DEL				


Zdroj: http://en.wikipedia.org/wiki/File:ASCII_Code_Chart-Quick_ref_card.png

Koduje znaky do 7 bitů (0-127). Nezohledňuje diakritiku.

Jiná kódování

- ISO8859-1 (8bitů, západoevropské jazyky)
- ISO8859-2 (8bitů, středoevropské jazyky)
- CP1250 (8bitů, windows)
- UNICODE
 - UTF7
 - UTF8 (úsporný)
 - UTF16 (16 bitů, nejpoužívanější)
 - UTF32 (32 bitů)

Kódování UNICODE (UTF8, UTF16, UTF32)

	UTF32	UTF16	UTF8
0	0x00000030	0x0030	0x30
A	0x00000041	0x0041	0x41
Á	0x000000C1	0x00C1	0xC3 0x81
û	0x0000016F	0x016F	0xC5 0xAF
	0x000103A0	0xD800 0xDFA0	0xF0 0x90 0x8E 0xA0

UTF8

0xxxxxxx

110xxxxx 10xxxxxx

11000101 10101111 C5 AF (16F)

11000011 10000001 C3 81 (C1)

1110xxxx 10xxxxxx 10xxxxxx

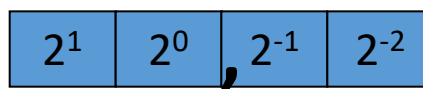
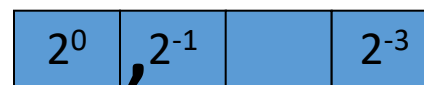
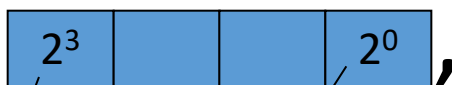
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

11110000 10010000 10001110 10100000 F0 90 8E A0 (103A0)

Reprezentace záporných čísel

Řádová mřížka (nový pojem)

Z důvodu omezení při zobrazení čísel (např. na rozsah 1 bytu) se v počítači zavádí pojem řádová mřížka, která popisuje formát zobrazených čísel. Typické šířky 8, 16, 32, 64 bitů.



Nejvyšší řád
řádové mřížky
($n=3$)

Nejnižší řád
řádové mřížky
($m=0$)

Modul řádové mřížky $Z=2^{n+1}$

Zobrazení (kódování) záporných čísel

V paměti počítače lze uložit pouze kladná celá čísla. Např. v paměti (nebo registru) o velikosti 1 byte (8 bitů) lze uložit jedno číslo z intervalu $\langle 0, 255 \rangle$.

Záporná čísla se musí tedy mapovat (kódovat, zobrazovat) na kladná celá čísla vhodným typem zobrazení.

Používaná zobrazení (kódy)

- Doplnkový kód (veškeré celočíselné typy)
- Přímý kód (mantisa čísel s pohyblivou řádovou čárkou)
- Aditivní kód (exponent čísel s pohyblivou řádovou čárkou)

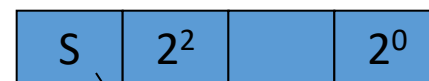
Doplňkový kód

0 - 127 - -128 - 255

Nejlevnější byt = 1 = záporné
0 = kladné

$$D(x) = \begin{cases} x, & 0 \leq x \leq \frac{Z}{2} - 1 \\ Z + x, & -\frac{Z}{2} \leq x < 0 \end{cases}$$

Kde Z je modul řádové mřížky 2^{n+1} .



Znaménko

Příklad 1 byte (8 bitů) $Z=2^8=256$

x	D (x)	x	D (x)
0	0	-1	255
1	1	-2	254
...		...	
127	127	-127	129
		-128	128

Operace v doplňkovém kódu

Převod kladného čísla na záporné a naopak: negujeme jednotlivě všechny bity v řádové mřížce a pak přičteme jedničku.

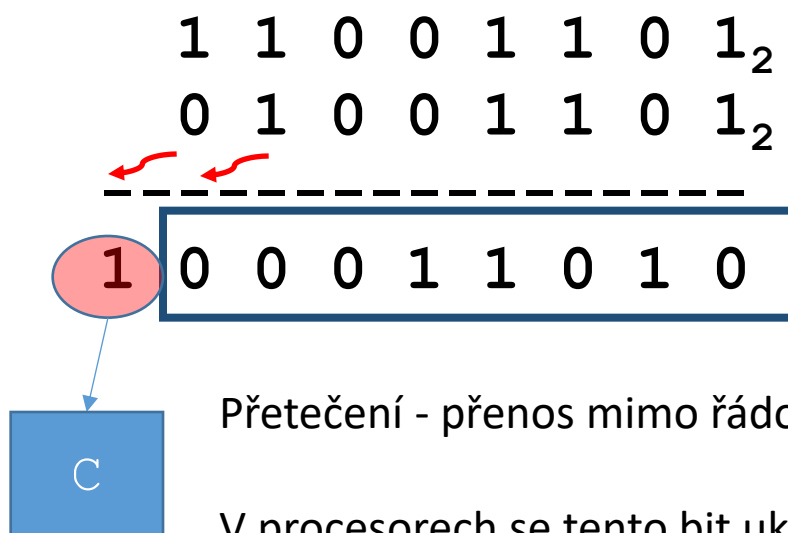
```
11101100    (-20)
00010011    negace (1->0 a 0->1, po bitech)
00010100    +1 => 20
```

Součet čísel v doplňkovém kódu: stejně jako u čísel bez znaménka

Násobení čísel v doplňkovém kódu: buď převést na kladná, vynásobit a pak podle znaménka výsledek upravit, nebo standardně vynásobit a aplikovat korekce.

Přetečení (celá kladná čísla)

Osmibitová řádková mřížka – odpovídá datovému typu byte (Java) nebo unsigned char (C, C++)



Carry bit se nastaví vždy, když součet přesáhne hodnotu Z-1, pro naši mřížku tedy hodnotu 255.

Přetečení - přenos mimo řádkovou mřížku.

V procesorech se tento bit ukládá do bitu C (carry) v příznakovém registru.

Přetečení (čísla v doplňkovém kódu) I

Příklad uvádíme pro osmibitovou řádovou mřížku – odpovídá datovému typu char (C, C++) nebo byte(Java), tj. čísla v rozsahu 0-255.

Nenastane přetečení (overflow)

$$\begin{array}{r} 1\ 1\ 0\ 0\ 1\ 1\ 0\ 1_2 \\ +\ 0\ 1\ 0\ 0\ 1\ 1\ 0\ 1_2 \\ \hline 1\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 0 \end{array}$$

Diagram showing the addition of two 8-bit numbers. The first number is 11001101₂ (-51) and the second is 01001101₂ (77). The result is 10001101₂ (26). Red arrows point to the carry bits from the 7th and 6th positions. The result is boxed, and the carry bit from the 8th position is crossed out with a blue 'X'.

Nastane přetečení (overflow)

$$\begin{array}{r} 1\ 1\ 0\ 0\ 1\ 1\ 0\ 1_2 \\ +\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1_2 \\ \hline 1\ 0\ 1\ 0\ 1\ 1\ 1\ 0\ 0 \end{array}$$

Diagram showing the addition of two 8-bit numbers. The first number is 11001101₂ (-51) and the second is 10001111₂ (-113). The result is 101011100₂ (-164). Red arrows point to the carry bits from the 7th and 6th positions. The result is boxed, and the carry bit from the 8th position is crossed out with a blue 'X'.

Přetečení pro čísla se znaménkem v doplňkovém kódu se indikuje bitem overflow v příznakovém registru procesoru.

Přetečení (čísla v doplňkovém kódu) II

Příklad uvádíme pro osmibitovou řádovou mřížku – odpovídá datovému typu char (C, C++) nebo byte(Java), tj. čísla v rozsahu 0-255.

Nenastane přetečení (overflow)

$$\begin{array}{r} 0\ 1\ 0\ 0\ 1\ 1\ 0\ 1_2 \\ +\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1_2 \\ \hline \end{array}$$

$$\begin{array}{r} \cancel{1}\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 0 \\ \hline \end{array}$$

Nastane přetečení (overflow)

$$\begin{array}{r} 0\ 1\ 0\ 0\ 1\ 1\ 0\ 1_2 \\ +\ 0\ 1\ 0\ 0\ 1\ 1\ 1\ 1_2 \\ \hline \end{array}$$

$$\begin{array}{r} 1\ 0\ 0\ 1\ 1\ 1\ 0\ 1 \\ \hline \end{array}$$

157

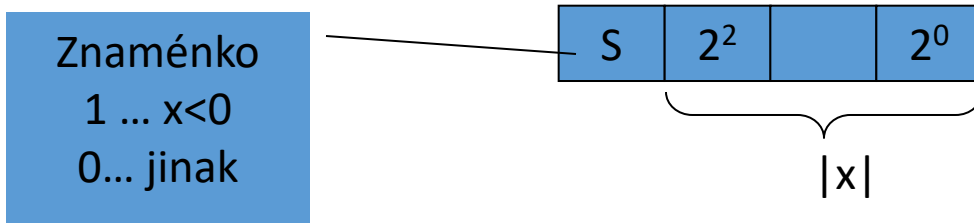
Detekce přetečení: pokud nastane přenos buď z nejvyššího řádu mřížky nebo do nejvyššího řádu mřížky. Pokud nenastane žádný přenos nebo oba, pak přetečení nenastalo.

Mimo
rozsah

Přímý kód

$$P(x) = \begin{cases} x & 0 \leq x \leq 2^n - 1 \\ 2^n - x & -2^n + 1 \leq x < 0 \end{cases}$$

n je nejvyšší řád řádové mřížky



V přímém kódu existují dvě nuly (kladná a záporná). Rozsah pro osmibitové číslo se znaménkem je tedy $\langle -127, 127 \rangle$

Aditivní kód

$$A(x) = x + K$$

Typicky $K = Z/2$ (polovina modulu řádové mřížky)

Pro $Z=16$ je typicky $K=8$

x	A(x)	x	A(x)
0	8	-1	7
1	9	-2	6
...		...	
7	15	-8	0

Čísla v plovoucí řádové čárce

Diagram illustrating the components of a floating-point number $A = m.z^e$:

- Exponent**: Points to the superscript e .
- Mantisa**: Points to the mantissa m .
- Základ číselné soustavy**: Points to the base z .

Známe z desítkové soustavy

Počítačově zapsáno $A = 3.25E3$

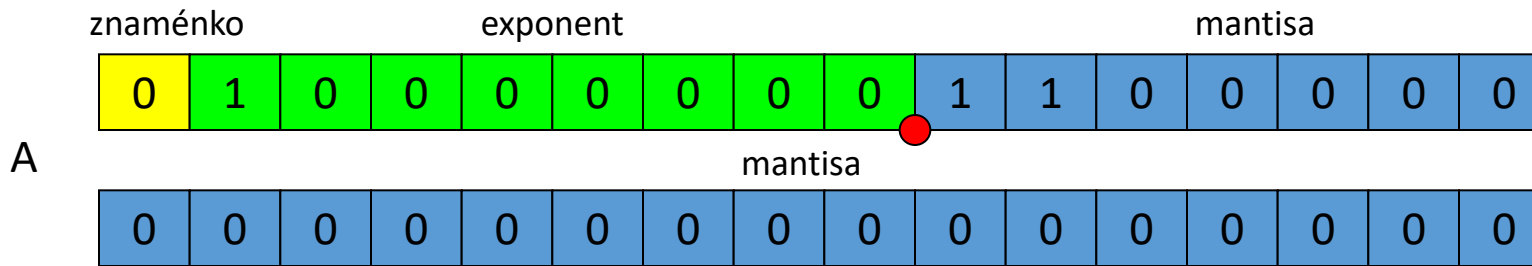
$$A = 3,25 \cdot 10^3$$

Ve dvojkové soustavě

$$B = 1,11_2 \cdot 2^3 = 14_{10}$$

Reprezentace čísel dle normy IEEE754

Binary32 (float) jednoduchá přesnost (single precision)



Exponent – aditivní kód $K=127$

Mantisa v rozsahu $1 \leq |m| < 2$, jednička (1, ...) v normalizovaném tvaru je vždy přítomna, nevyjadřuje se, proto se jí říká skrytá jednička.

$$A = 1,75 \cdot 2^1 = 3,5$$

Binary64 (double) dvojitá přesnost – 11 bitů exponent - aditivní kód $K=1023$, 52 bitů mantisa, 1 bit znaménko

Součet čísel v plovoucí řádové čárce

- Doplnit mantisy o skryté jedničky
- Posuvem mantisy srovnat exponenty na exponent s vyšší hodnotou. Posouváme mantisu s nižším exponentem vpravo (tj. dělíme ji postupně dvěma). Posun kompenzujeme zvětšením exponentu o jedničku.
- Sečteme mantisy (respektujeme znaménka)
- Mantisu normalizujeme. Pokud hodnota mantisy (co do absolutní hodnoty) je větší nebo rovna dvěma, pak posuneme vpravo a upravíme exponent. Pokud naopak je mantisa menší než jedna, posouváme mantisu vlevo. Výsledek navíc musíme zaokrouhlit, protože posunem vpravo může dojít ke ztrátě přesnosti (vysunutí bitů mimo řádovou mřížku).
- Nastavíme znaménko výsledku

Násobení čísel v plovoucí řádové čárce

- Doplnit mantisy o skryté jedničky
- Sečíst exponenty
- Vynásobit mantisy
- Výsledek normalizovat a zaokrouhlit
- Nastavit znaménko výsledku

Ukládání vícebytových čísel v paměti počítače

Paměť je adresovaná po bytech (= každý byte má svoji adresu)

Proměnná (typ) : obsah

a (byte) : 0x34

b (short) : 0x12AB

c (int) : 0x5676BCDE

Adresa	Obsah paměti	Little endian
0x00010007	56	
0x00010006	76	
0x00010005	BC	
0x00010004	DE	
0x00010003	12	
0x00010002	AB	
0x00010001		
0x00010000	34	

Adresa	Obsah paměti	Big endian
0x00010007	DE	
0x00010006	BC	
0x00010005	76	
0x00010004	56	
0x00010003	AB	
0x00010002	12	
0x00010001		
0x00010000	34	

Jiné kódování čísel - Grayův kód

Je kódem, kde mezi dvěma po sobě jdoucími kódovými slovy dochází pouze ke změně v jednom bitu.

Číslo	Binární	Grayův
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Grayův kód se používá například v optických snímačích polohy nebo pro implementaci asynchronních konečných automatů.

Kombinační logické obvody

Booleovská proměnná a funkce

Definice 1: Booleovská proměnná je proměnná, která nabývá pouze hodnot 0 nebo 1.

Přímé proměnné: a, b, c, \dots

Negované proměnné: $\bar{a}, \bar{b}, \bar{c}, \dots$

Definice 2: Booleovská funkce n proměnných je funkce $f : \{0,1\}^n \rightarrow \{0,1\}$

Booleovská funkce n proměnných $y = f(x_1, x_2, \dots, x_n)$,
kde y, x_1, x_2, \dots, x_n jsou booleovské proměnné.

Pro n booleovských proměnných existuje 2^{2^n} booleovských funkcí

Funkce dvou proměnných

a	b	0	NOR		\bar{a}		\bar{b}	XOR	NAND	AND	\Leftrightarrow	b	=>	a		OR	1
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Minterm a maxterm

a	b	c	Minterm	Maxterm
0	0	0	$m_0 = \bar{a}\bar{b}\bar{c}$	$M_0 = a + b + c$
0	0	1	$m_1 = \bar{a}\bar{b}c$	$M_1 = a + b + \bar{c}$
0	1	0	$m_2 = \bar{a}b\bar{c}$	$M_2 = a + \bar{b} + c$
0	1	1	$m_3 = \bar{a}bc$	$M_3 = a + \bar{b} + \bar{c}$
1	0	0	$m_4 = a\bar{b}\bar{c}$	$M_4 = \bar{a} + b + c$
1	0	1	$m_5 = a\bar{b}c$	$M_5 = \bar{a} + b + \bar{c}$
1	1	0	$m_6 = ab\bar{c}$	$M_6 = \bar{a} + \bar{b} + c$
1	1	1	$m_7 = abc$	$M_7 = \bar{a} + \bar{b} + \bar{c}$

Minterm je součinem všech proměnných (buď přímých nebo negovaných). Pokud pro daný řádek proměnná nabývá log. 1, použije se přímá proměnná. V opačném případě negovaná proměnná.

Maxterm je součtem všech proměnných (buď přímých nebo negovaných). Pokud pro daný řádek proměnná nabývá log. 1, použije se negovaná proměnná. V opačném případě přímá proměnná.

Pozn.: žádná proměnná se v mintermu a maxtermu neopakuje dvakrát, a to ať v přímé nebo negované podobě.

Úplné normální formy

Úplná normální disjunktivní forma (ÚNDF) je součtem (disjunkcí) mintermů.

Úplná normální konjunktivní forma (ÚNKF) je součinem (konjunkcí) maxtermů.

a	b	c	y		
0	0	0	0		M0
0	0	1	1	m1	
0	1	0	0		M2
0	1	1	0		M3
1	0	0	0		M4
1	0	1	1	m5	
1	1	0	1	m6	
1	1	1	0		M7

$$y = m1 + m5 + m6 = \bar{a} \cdot \bar{b} \cdot c + a \cdot \bar{b} \cdot c + a \cdot b \cdot \bar{c}$$

$$y = M0 \cdot M2 \cdot M3 \cdot M4 \cdot M7 = \\ (a + b + c) \cdot (a + \bar{b} + c) \cdot \\ (a + \bar{b} + \bar{c}) \cdot (\bar{a} + b + c) \cdot (\bar{a} + \bar{b} + \bar{c})$$

Pravdivostní tabulka -> booleovský výraz (formule)

ÚNDF (Úplná normální disjunktční forma)

$$y = \bar{a}.\bar{b}.c + a.\bar{b}.c + a.b.\bar{c}$$

Minterm
m

$$y = \overline{a.\bar{b}.\bar{c} + a.b.c}$$

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

a	b	c	y
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Když je více jedniček než nul, je výhodnější popsat výrazem negaci funkce. Pak se zaměřujeme na nuly.

Booleova algebra

Zákony	+(or)	.(and)
Neutrality nuly a jedničky	$x + 0 = x$	$x \cdot 1 = x$
Komutativní	$x + y = y + x$	$x \cdot y = y \cdot x$
Asociativní	$(x + y) + z = x + (y + z)$	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$
Distributivní	$(x + y) \cdot (x + z) = x + y \cdot z$	$x \cdot (y + z) = x \cdot y + x \cdot z$
Idempotence	$x + x = x$	$x \cdot x = x$
Agresivity nuly a jedničky	$x + 1 = 1$	$x \cdot 0 = 0$
Absorbce	$x + xy = x$	$x \cdot (x + y) = x$
Absorbce negace	$x + \bar{x} \cdot y = x + y$	$x \cdot (\bar{x} + y) = x \cdot y$
Negace negace	$\bar{\bar{x}} = x$	
Vyloučeného třetího	$x + \bar{x} = 1$	$x \cdot \bar{x} = 0$
De Morganovy	$\overline{x + y} = \bar{x} \cdot \bar{y}$	$\overline{x \cdot y} = \bar{x} + \bar{y}$

Algebraický důkaz zákonů absorbce negace

$$x(\bar{x} + y) = xy$$

Důkaz:

$$x(\bar{x} + y) = \underbrace{x\bar{x}}_0 + xy = xy$$

$$x + \bar{x}y = x + y$$

Důkaz:

$$\begin{aligned} x + \bar{x}y &= x \overbrace{(1 + y)}^1 + \bar{x}y = x + xy + \bar{x}y = \\ &= x + xy + \bar{x}y = x + y \underbrace{(x + \bar{x})}_1 = x + y \end{aligned}$$

Zákony booleovy algebry využijeme k minimalizaci booleovské funkce

$$\begin{aligned} y &= a.b.c + a.\bar{b}.c + a.\bar{b}.\bar{c} = a.c(b + \bar{b}) + a.\bar{b}.\bar{c} = \\ &= a.c.1 + a.\bar{b}.\bar{c} = a(c + \bar{b}.\bar{c}) = a.(c + \bar{b}) \end{aligned}$$

Minimalizace booleovské funkce je postup, kterým výraz zjednodušíme z pohledu počtu přímých a negovaných proměnných a počtu termů ve výrazu. Obecně řečeno snížíme počet operací, které je nutno provést.

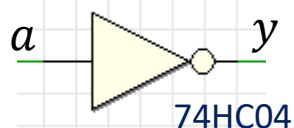
Metody:

- Algebraická minimalizace (úpravy výrazu využitím zákonů booleovy algebry)
- Karnaughovy mapy
- Algoritmus Quine-McCluskey

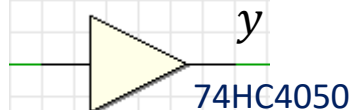
Logické obvody - logické (booleovské) funkce implementované v hardwaru

Přehled nejčastěji užívaných logických hradel

Invertor $y = \bar{a}$

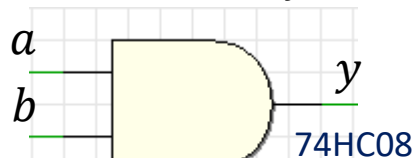


Buffer $y = a$

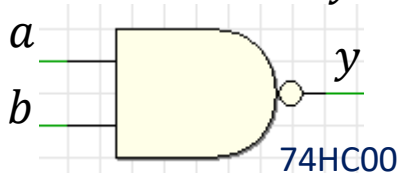


Vynucené zpoždění
nebo zesílení signálu

Hradlo AND $y = ab$



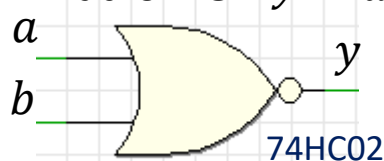
Hradlo NAND $y = \overline{ab}$



Hradlo OR $y = a + b$

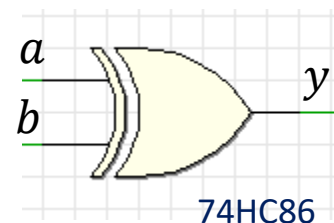


Hradlo NOR $y = \overline{a + b}$



Hradlo XOR

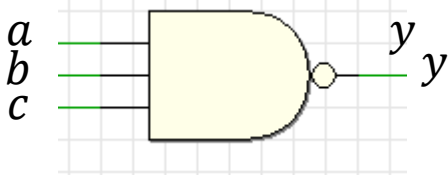
$$y = a \oplus b$$



Vícevstupá hradla, uvádíme pouze pro NAND, obdobně pro AND, OR, NOR, XOR

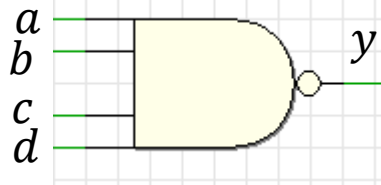
Třívstupový NAND

$$y = \overline{abc}$$



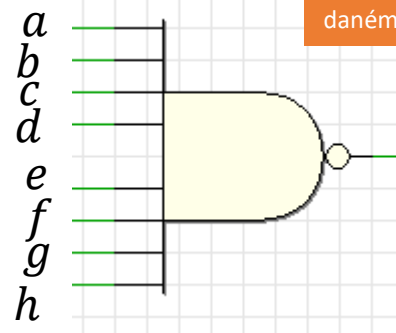
Čtyřvstupový NAND

$$y = \overline{abcd}$$



Osmivstupový NAND

$$y = \overline{abcdefgh}$$



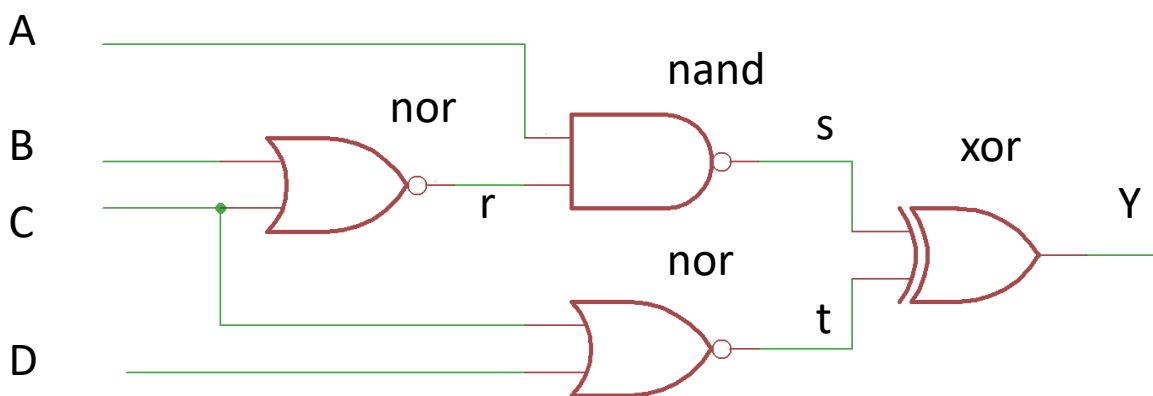
PRO ZVĚDAVCE:

U každého obvodu uvádíme číslo (např. 74HC00) pod kterým můžete výše uvedené hradlo zakoupit. Pro další informace použijte vyhledávač a dotaz např. „74HC00 datasheet pdf“ a získáte podrobné informace o daném obvodu.

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.



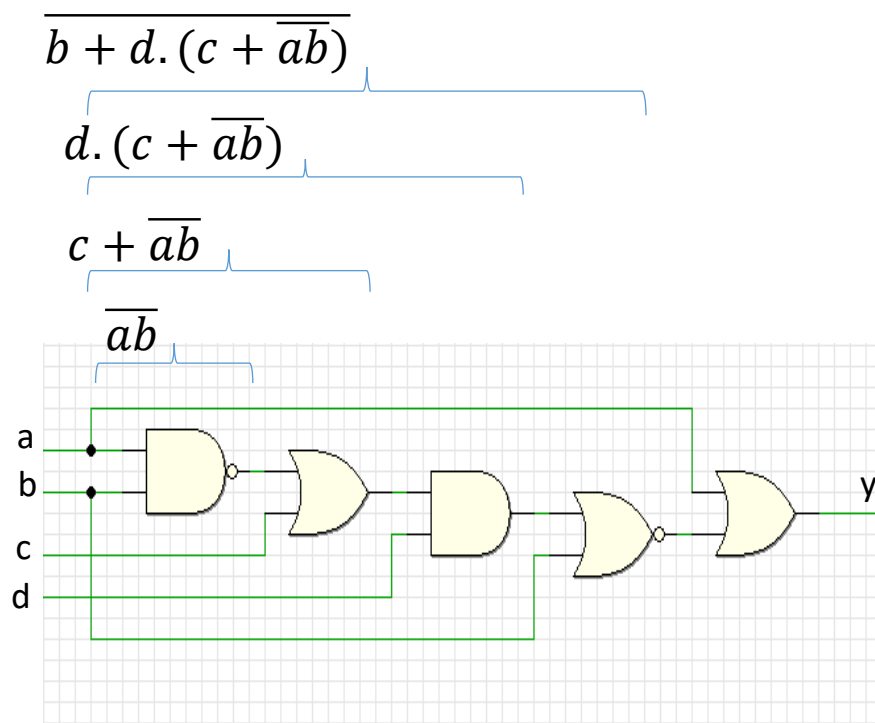
*Sestrojte
pravdivostní
tabulku pro toto
zapojení.*

*Návod: postupujte od
vstupu k výstupu a
postupně si vytvořte
pravdivostní tabulky pro
vnitřní signály a výstup,
zde v pořadí r, t, s a
nakonec Y.*

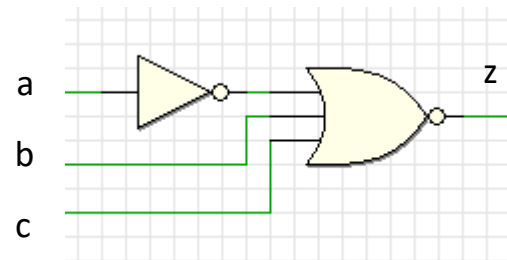
Pro daný logický výraz nakreslete odpovídající schéma

$$y = a + b + d \cdot (c + \overline{ab})$$

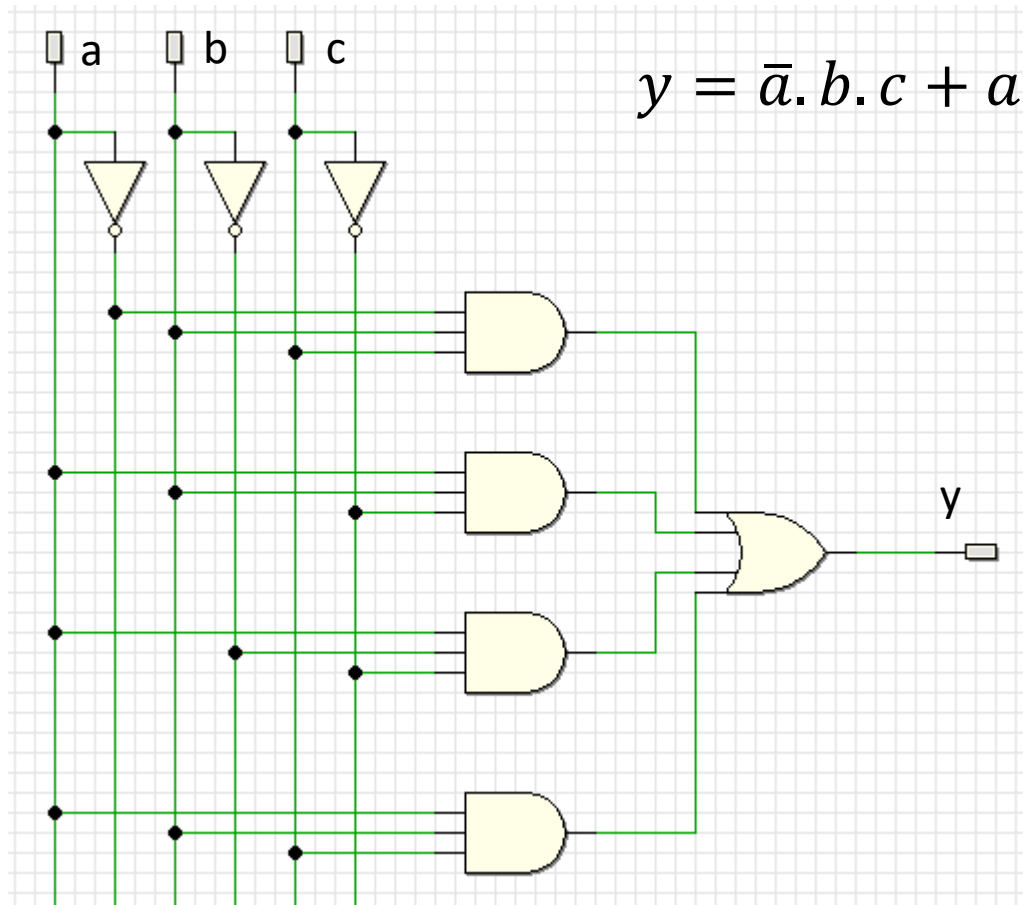
Řešení



$$z = \overline{\overline{a} + b + c}$$



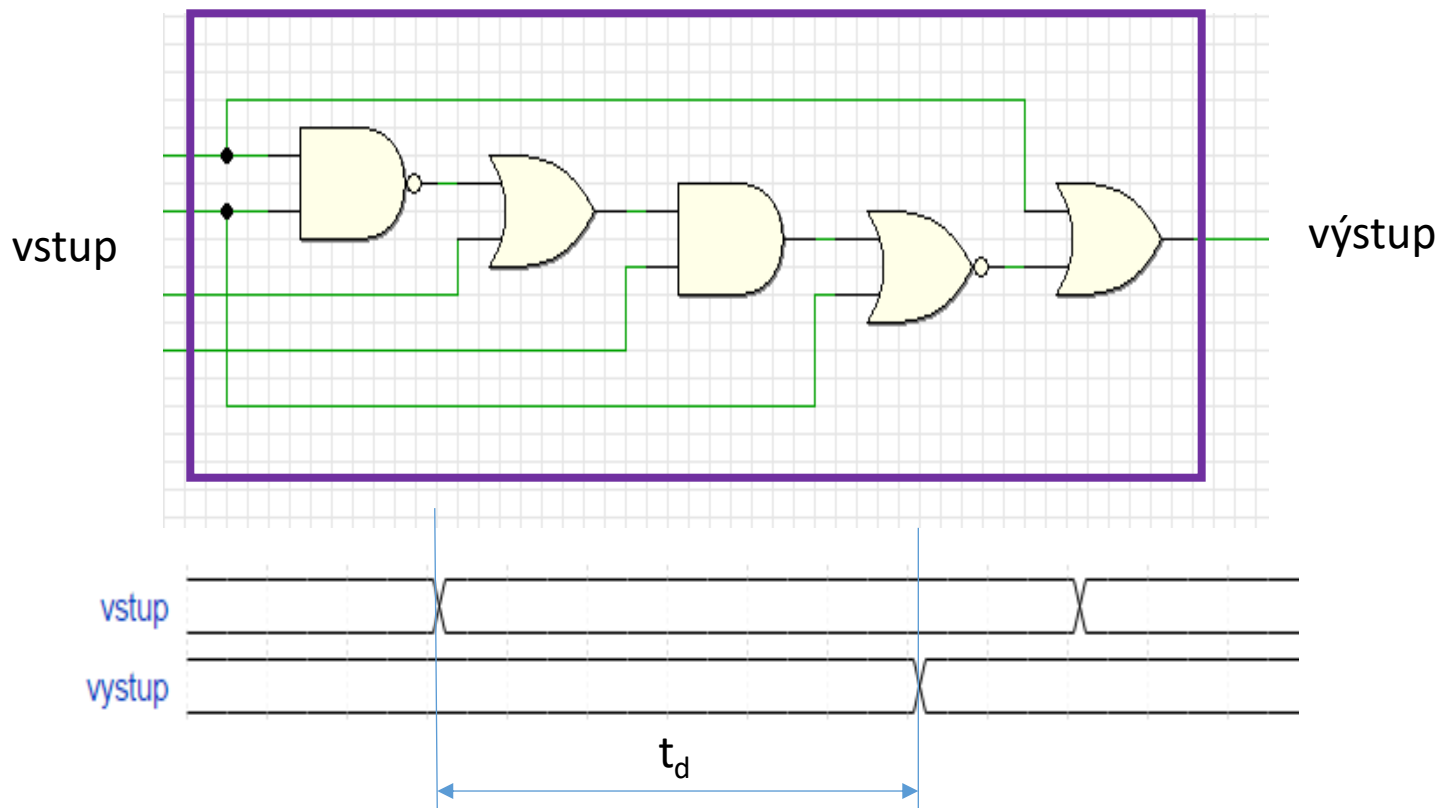
ÚNDF → schéma zapojení



$$y = \bar{a}.b.c + a.b.\bar{c} + a.\bar{b}.\bar{c} + a.b.c$$

a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Chování kombinačního obvodu v čase



T_d je zpoždění kombinačního obvodu. Je to doba mezi změnou vstupu a tomu odpovídající změně výstupu. Každé hradlo má určité zpoždění a zpoždění narůstá, jak se změna vstupu šíří obvodem. Celkové zpoždění t_d je tedy závislé na tom, které bity se mění na vstupu. V návrhu obvodů se musí vždy počítat s největším možným zpožděním.

Popis kombinačního obvodu ve VHDL

VHDL (VHSIC* Hardware Description Language) je jazyk pro popis hardware. Jiným obdobným a často užívaným jazykem je jazyk Verilog. Tyto jazyky se používají pro popis složitých obvodů, jako jsou například procesory. Dnes tyto jazyky nabyly na významu díky snadno dostupným a populárním programovatelným obvodům FPGA a nejsou jen doménou návrhářů integrovaných obvodů

Logická funkce ve VHDL

```
-- deklarace signálů
signal a: std_logic;  -- jednobitový signál (0,1,U,Z...)
signal b: std_logic;  -- jednobitový signál (0,1,U,Z...)
signal c: std_logic;  -- jednobitový signál (0,1,U,Z...)
signal y: std_logic;  -- jednobitový signál (0,1,U,Z...)
```

```
y <= a or b and (c or a) after 25 ns;
```

<= (signal assignment) můžeme chápat jako přiřazení, ale je z hlediska simulace je to přenesení hodnoty booleovského výrazu na výstup logického obvodu y se zpožděním (after 25ns, měřeno v simulačním čase). Simulátor tento řádek nevyhodnocuje jen tehdy, když výpočet programu dospěje na daný řádek (typické pro běžný programovací jazyk), ale vždy, když dojde ke změně hodnoty signálů a, b, c.

Logické operátory ve VHDL: not, or, and, xor, xnor.

* Very High Speed Integrated Circuit – americkou vládou podporovaný program v 80 letech minulého století

Základní kombinační obvody

Kombinační obvody pro sčítání I - pulsčítačka

$$\begin{aligned}0_2 + 0_2 &= 0_2 \\0_2 + 1_2 &= 1_2 \\1_2 + 0_2 &= 1_2 \\1_2 + 1_2 &= \textcolor{red}{1}0_2\end{aligned}$$

Aritmetický
součet

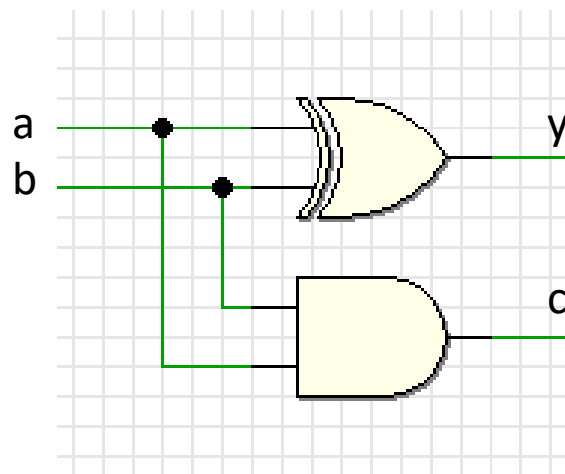
Přenos do vyššího řádu
(CARRY)

Pravdivostní tabulka

a	b	y	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

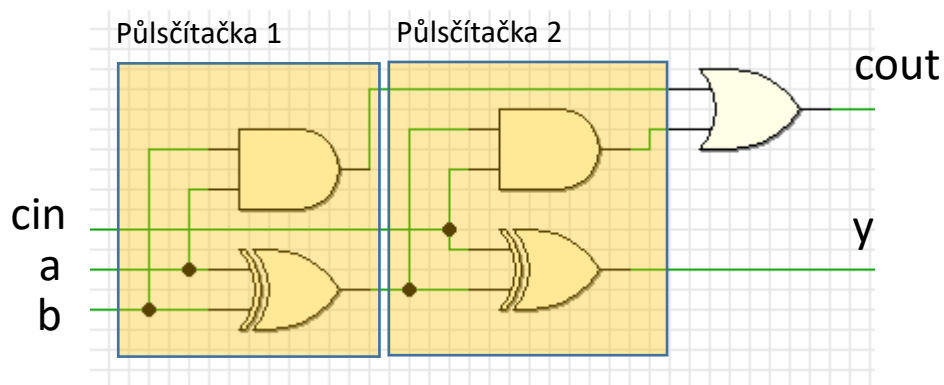
$$\begin{aligned}y &= a \oplus b \\c &= a \cdot b\end{aligned}$$

Schéma zapojení



Kombinační obvody pro sčítání I – úplná sčítačka

a	b	C _{in}	y	c _{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

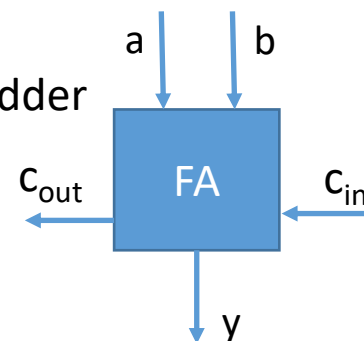


$$s = a \oplus b$$

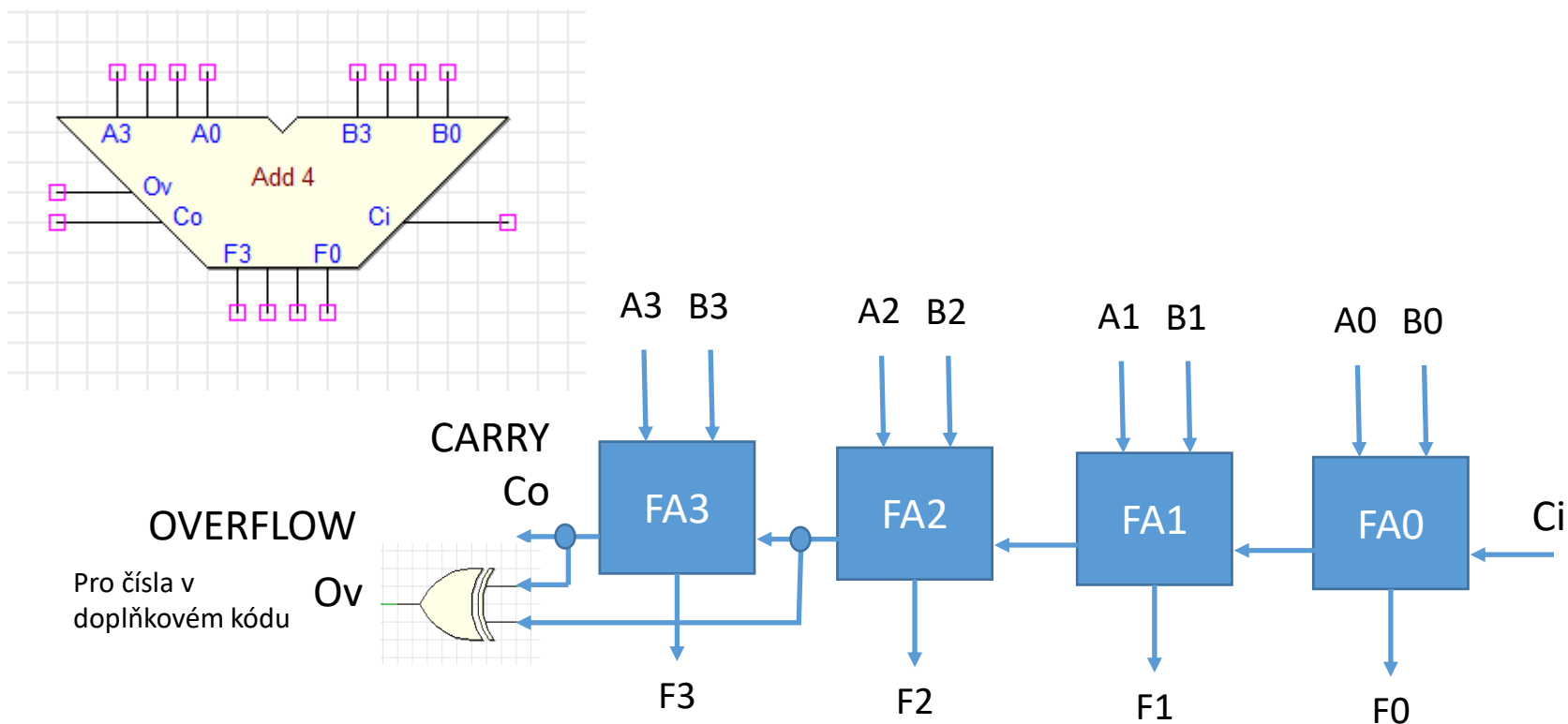
$$y = s \oplus c_{in}$$

$$c_{out} = a \cdot b + s \cdot c_{in}$$

FA – full adder



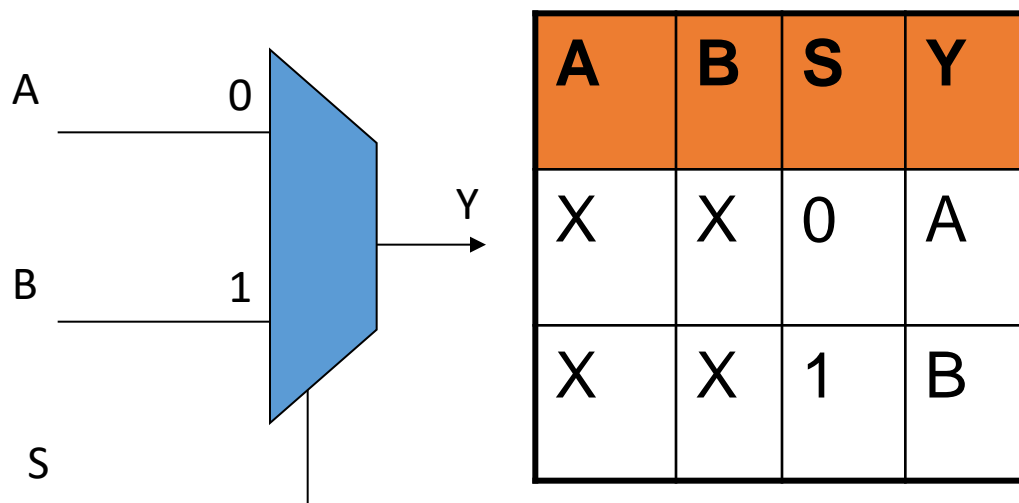
Vícebitová sčítačka (4 bitová)



FAx – (Full Adder) úplná sčítačka

Multiplexor (anglicky Multiplexer)

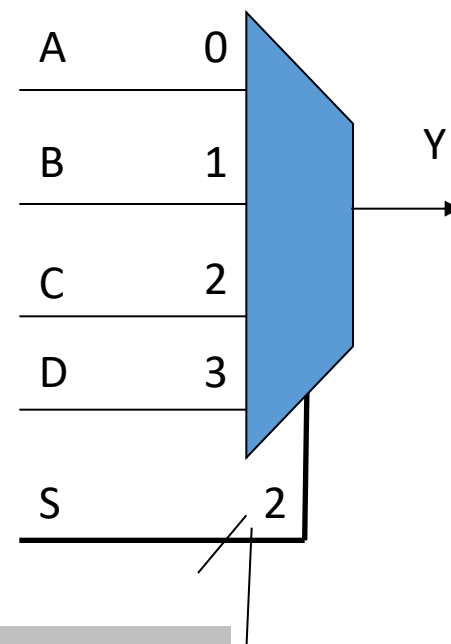
Dvouvstupový multiplexor



X – log. nula nebo jedna

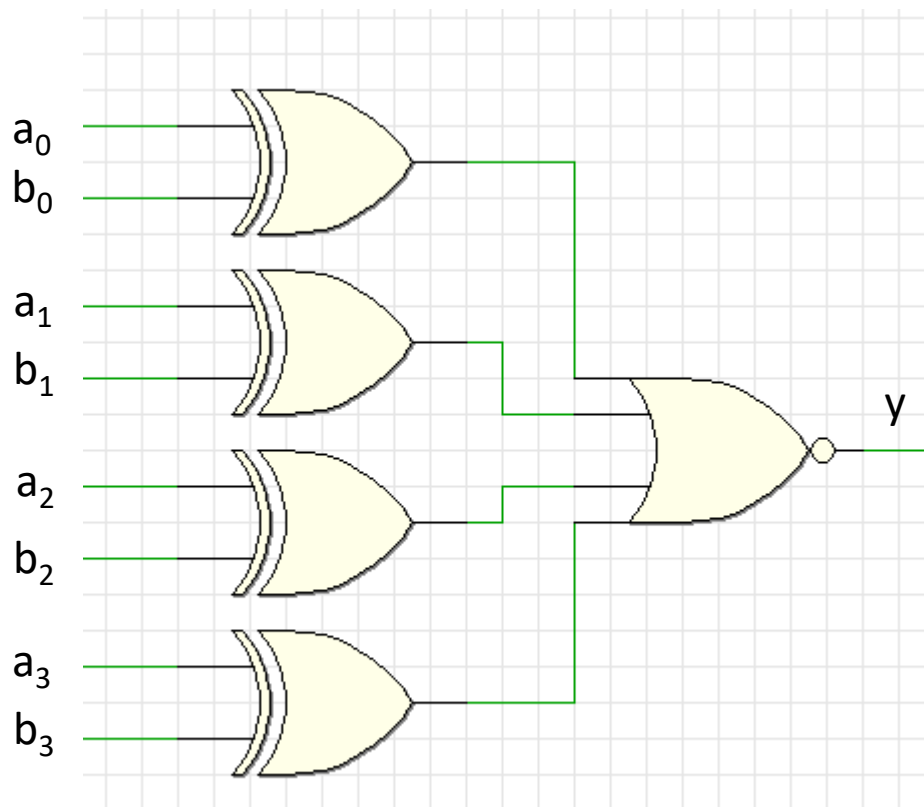
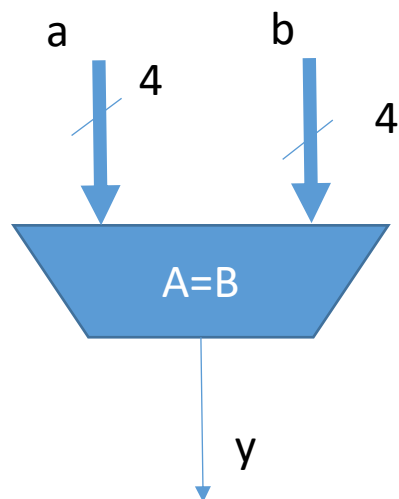
Multiplexor vybírá jeden ze dvou nebo více vstupů na jediný výstup Y. Můžete si tento obvod funkčně představit jako přepínač. Vybraný vstup je určen vstupem S.

Čtyřvstupový multiplexor



Označuje dva vodiče

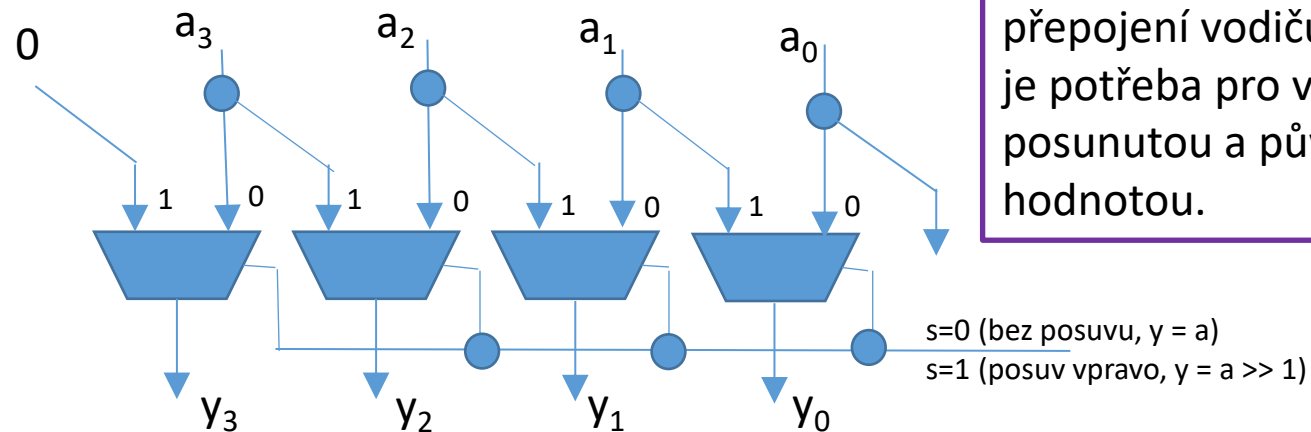
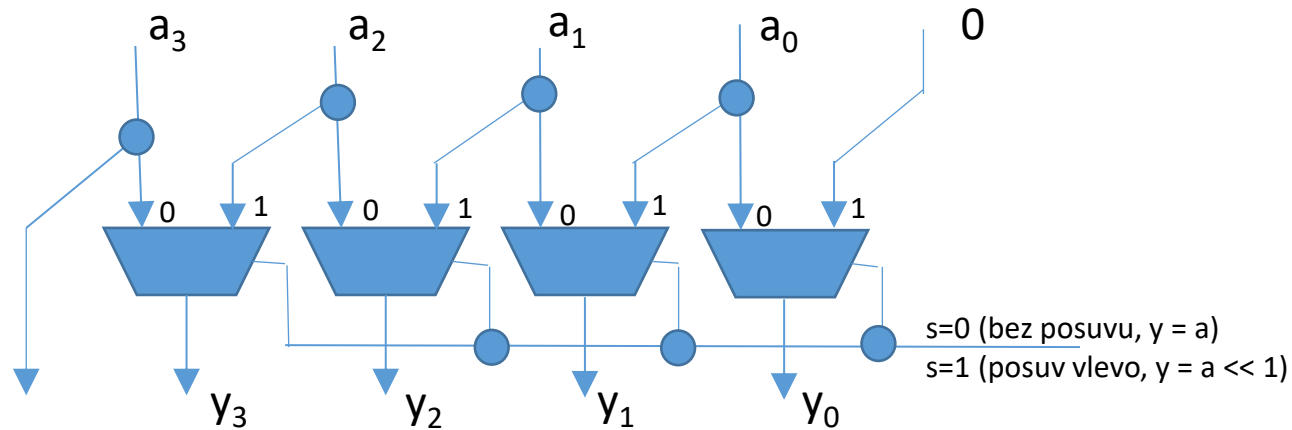
Komparátor



Výstup komparátoru je log. 1 pokud $a = b$.

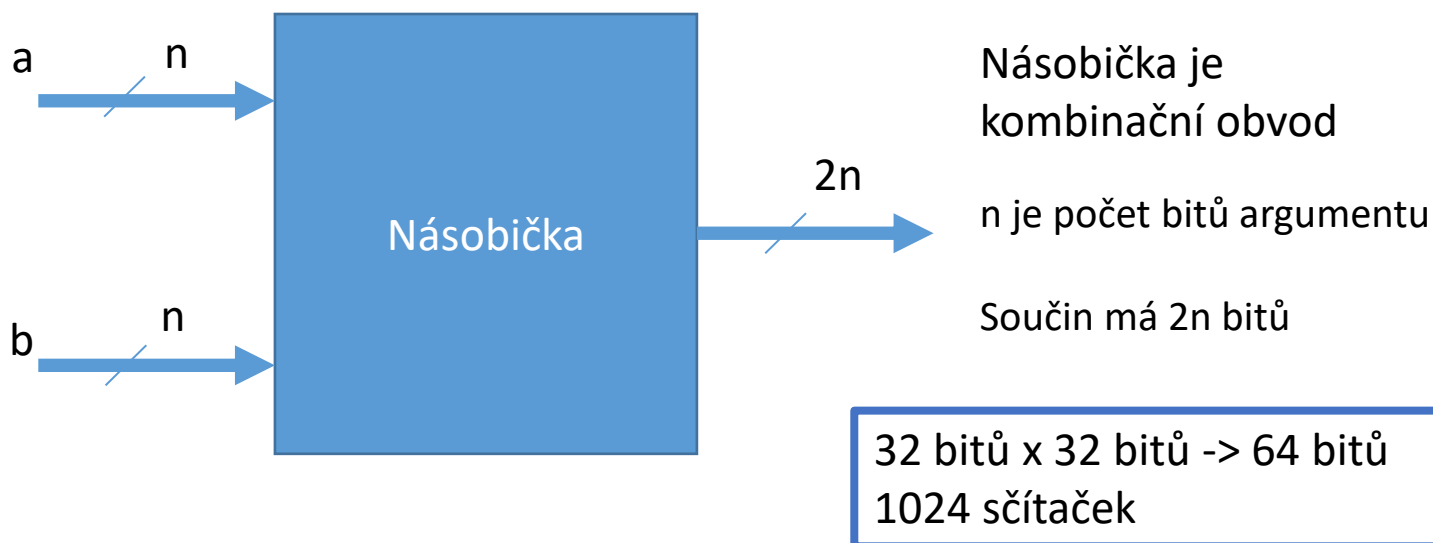
Existují komparátory, které indikují kromě rovnosti také $a < b$ a $a > b$.

Posuvy



Posun je vlastně jen přepojení vodičů, multiplexor je potřeba pro výběr mezi posunutou a původní hodnotou.

Násobička



Násobička je kombinační obvod, který obsahuje n^2 sčítaček. Sčítačky tvoří n sériově řazených stupňů, které určuje poměrně značné celkové zpoždění násobičky.

Ne každý procesor má násobičku (tj. instrukci pro násobení). Pokud procesor nemá instrukci násobení, pak se násobení implementuje programem v assembleru, který obsahuje logické instrukce, instrukce pro posuvy a součet.

Sekvenční logické obvody

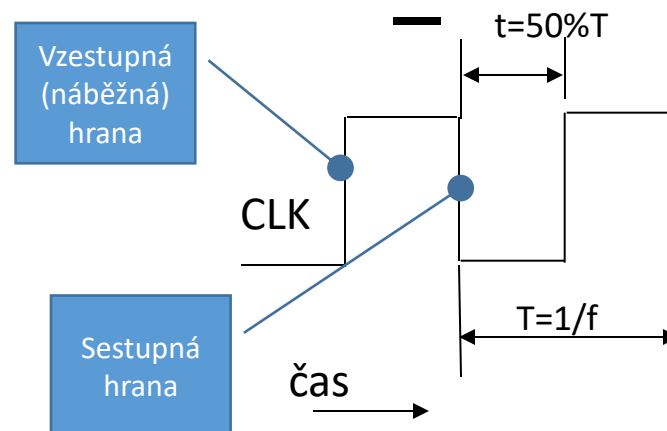
Hodinový signál

Hodinový signál je číslicový signál (0/1), který se mění z 1->0 a 0->1 s určitou frekvencí a proporcí mezi úrovní 1 a 0 v poměru 1:1. Tedy 50% periody 1 a 50% periody 0.

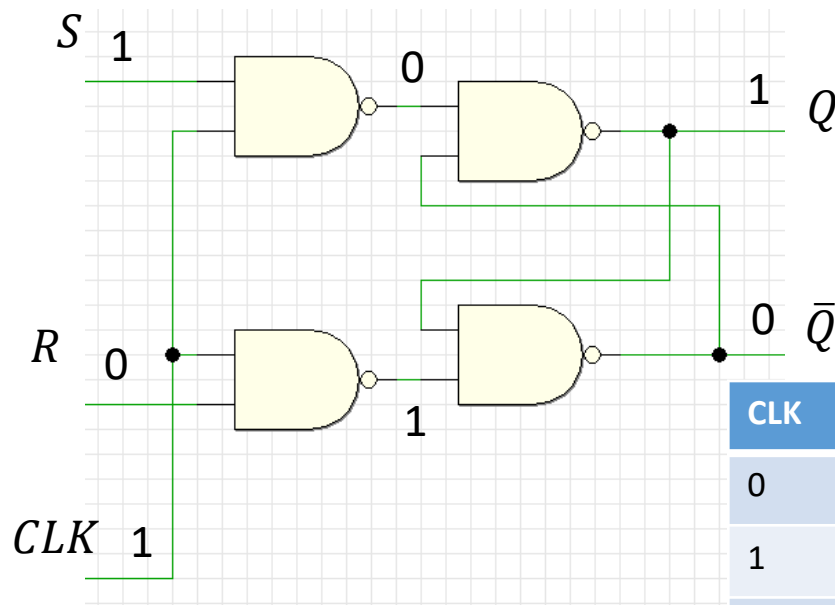
Hodinový signál vstupuje do všech sekvenčních obvodů, určuje okamžik provedení a také rychlost provádění návazných operací.

Základním parametrem hodinového signálu je frekvence (f_{clk}). Často se uvádí hodinová frekvence procesoru, která je rovněž měřítkem jeho výkonu.

32768Hz - nízkopříkonové mikrokontroléry,
500kHz-24MHz –standardní mikrokontroléry
30-200MHz – středně a vysoce výkonné mikrokontroléry
200MHz-1GHz – embedded procesory (ARM, x86)
>1GHz – desktopové a serverové procesory



Synchronní klopný obvod R-S (hladinový)



Paměťová funkce

CLK	R	S	Q	\bar{Q}
0	X	X	Q	\bar{Q}
1	0	0	Q	\bar{Q}
1	0	1	1	0
1	1	0	0	1
1	1	1	Q = 1, \bar{Q} = 1 pokud CLK=1, nepredikovatelný výsledek po přechodu CLK na hodnotu 0	

S (Set) nastavení výstupu Q do jedničky

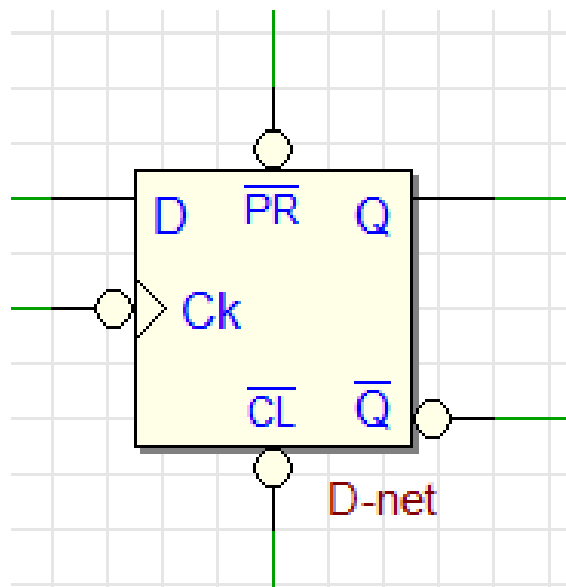
R (Reset) nastavení výstupu Q do nuly

Zakázaný stav
(nesmí nastat)

O synchronním klopném obvodu R-S

- Je to základní synchronní klopný obvod
- Z něj se odvozují další klopné obvody: J-K, D, T
- Synchronní znamená, že je synchronizován hodinovým signálem
- Hladinový obvod znamená, že výstup Q se může měnit po celou dobu, kdy je hodinový signál v log. 1. Proto se musí zajistit stabilní vstupy po celou tuto dobu, aby došlo k nejvýše jedné změně výstupu Q
- Výše uvedená nevýhoda se dnes řeší tzv. hranovými klopnými obvody, které reagují (mění výstup Q) pouze s hranou (vzestupnou/sestupnou) hodinového signálu. Tyto obvody jsou konstrukčně složitější a obsahují více R-S klopných obvodů

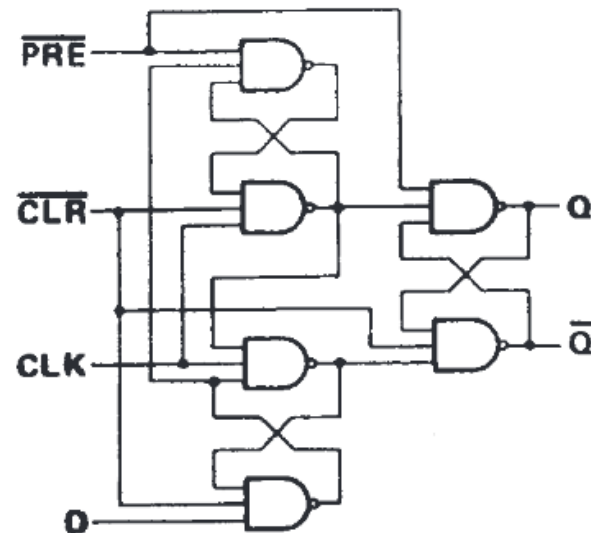
Hranový klopný obvod typu D



PR(PRE) je asynchronní nastavení
Q na log. 1 (aktivní v log. 0)

CL(CLR) je asynchronní nastavení
Q na log. 0 (aktivní v log. 0)

Pozn.: asynchronní znamená, že není
synchronizován s hodinovým signálem a
přechod do aktivní úrovně (v našem případě log.
0) způsobí odpovídající okamžitou změnu na Q.



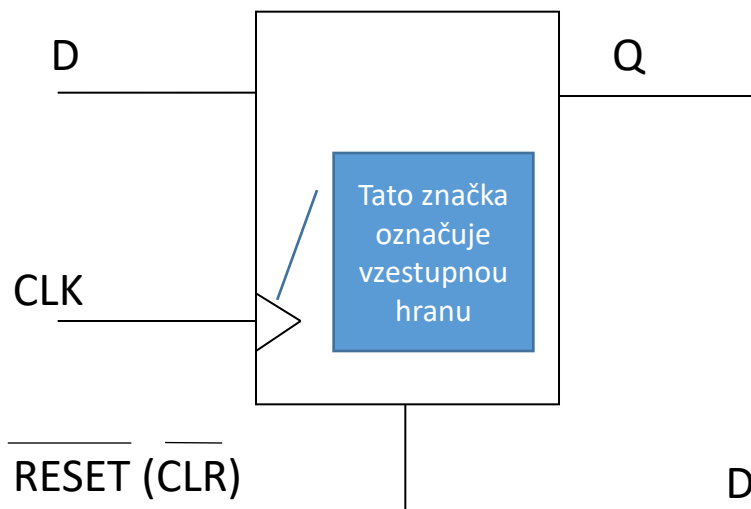
*Převzato z DUAL D-TYPE EDGE TRIGGERED FLIP-FLOPS
WIDTH PRESET AND CLEAR, datasheet, Texas Instruments
Incorporated, 1988*

Tento obvod lze zakoupit v obchodě se součástkami
pod označením např. 74LS74

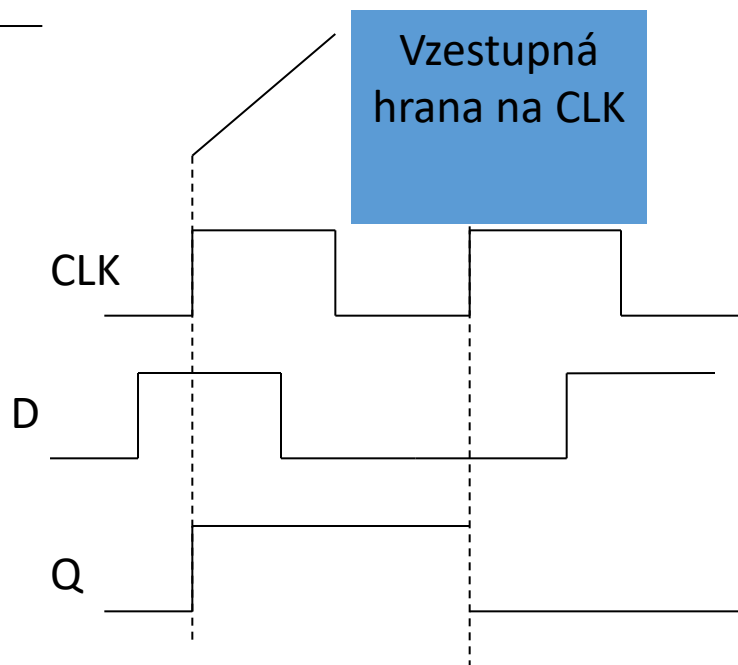
Funkce hranového klopného obvodu typu D

Klopný obvod D je jednobitovou pamětí.

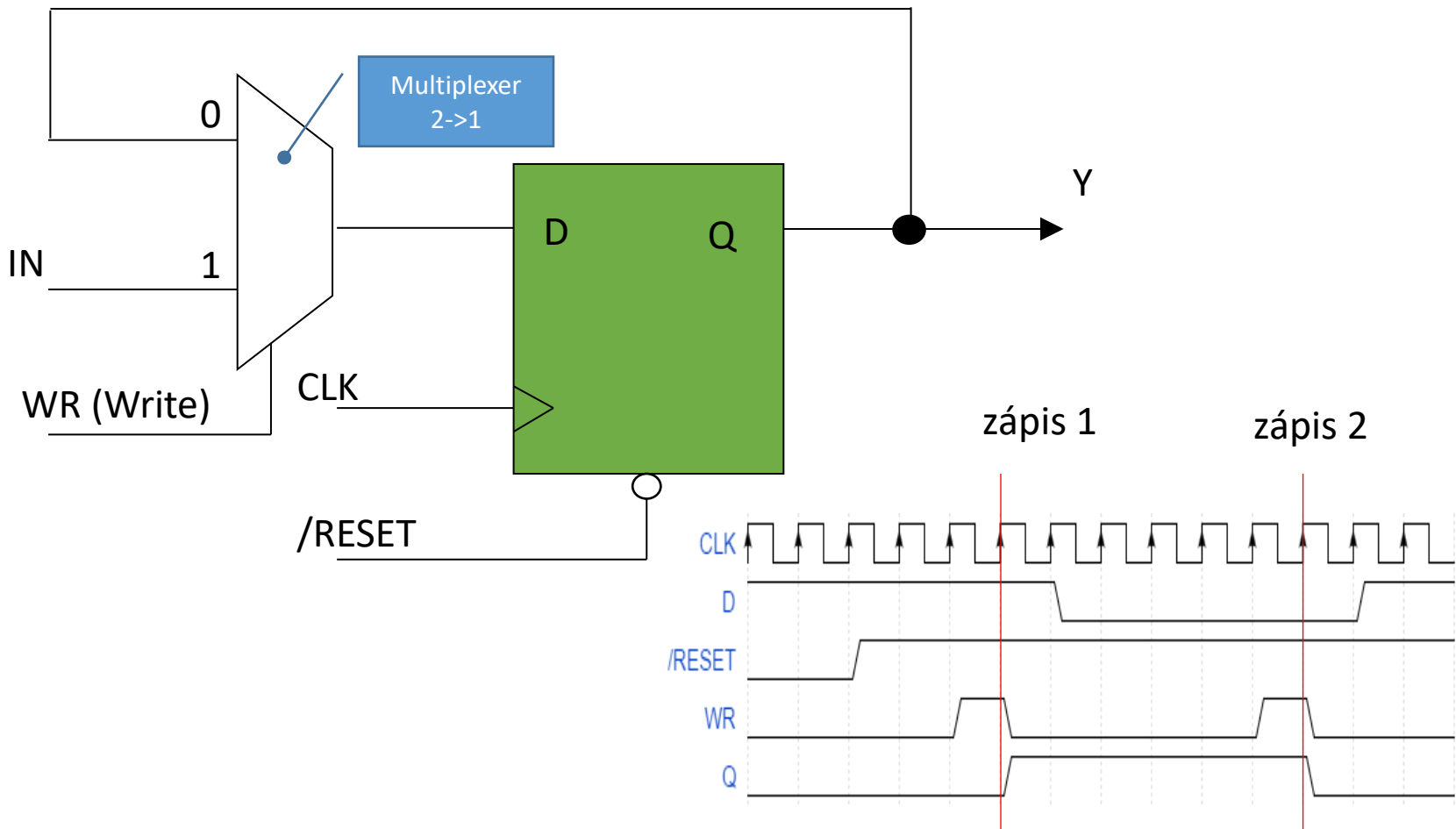
Nula na signálu RESET nastaví Q do nuly bez ohledu na CLK a D. Používá se k inicializaci klopného obvodu.



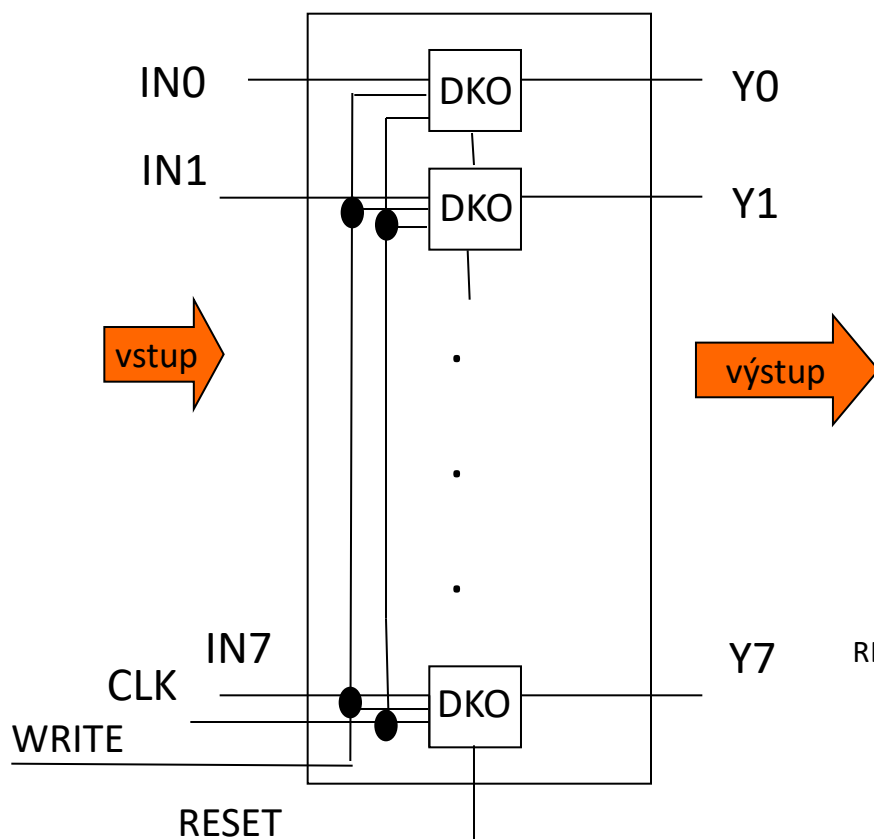
Při vzestupné hraně signálu CLK se přepíše logická hodnota ze vstupu D na výstup Q. Mimo vzestupnou hranu na CLK se může D měnit libovolně a na výstup Q to nemá vliv. Obvod si tedy pamatuje poslední hodnotu na D zapsanou vzestupnou hranou CLK.



Jednobitový registr se synchronním zápisem a asynchronním nulováním



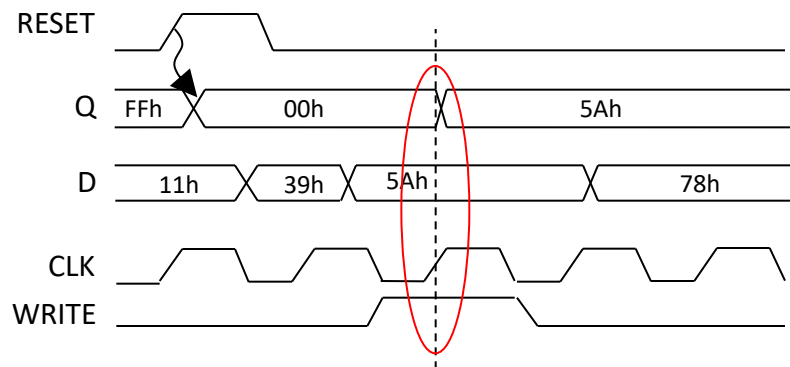
Vícebitový (paralelní) registr



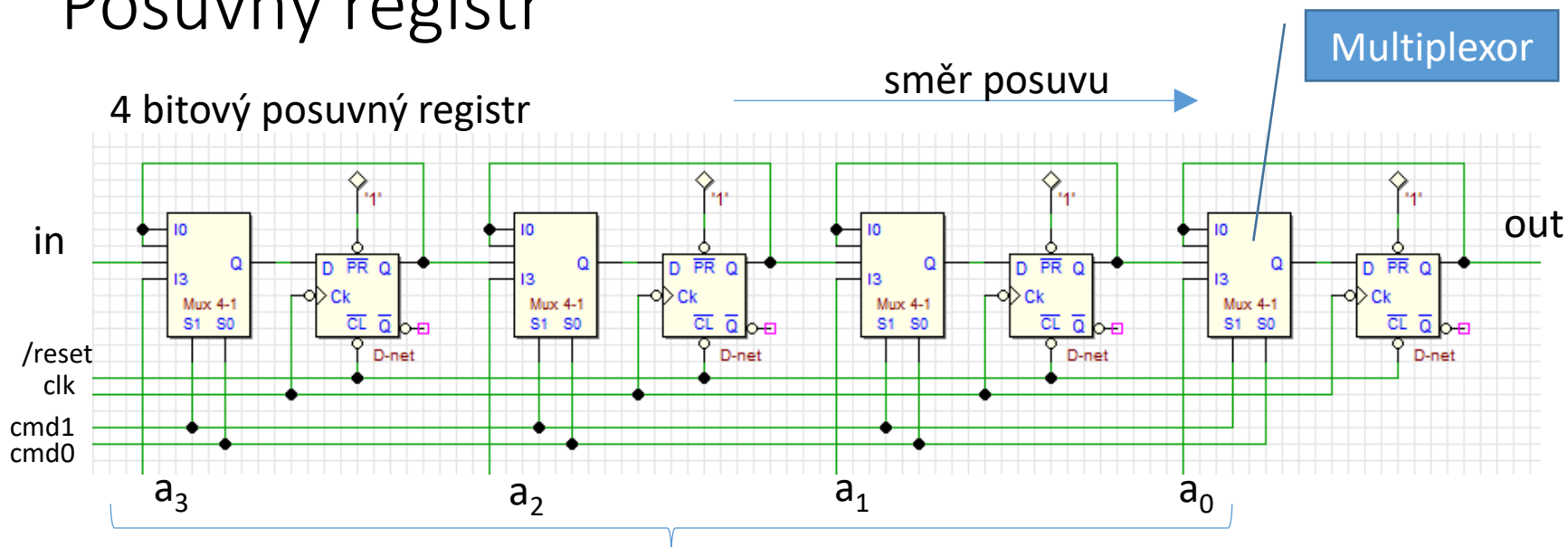
Sada klopných obvodů D se společným hodinovým vstupem tvoří (paralelní) registr.

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.



Posuvný registr



Cmd(1:0)	Funkce
0x	Bez změny
10	Posun vpravo
11	Nahrát hodnotu paralelně

Posuvný registr se užívá všude, kde serializují a deserializují data. Např. USB, Ethernet, UART (COM port), SATA, PCIe.

Data paralelně vstupují vstupem a, sériově vystupují na výstupu out. Nebo paralelně vstupují vstupem in a čtou se na výstupech klopných obvodů (není zakresleno). Multiplexor přepíná požadovanou funkci.

Popis sekvenčních obvodů – konečné automaty (KA, anglicky FSM)

Konečný automat je definován $KA = (Q, \Sigma, \Gamma, \delta, \omega, q_0)$

Q ... množina stavů

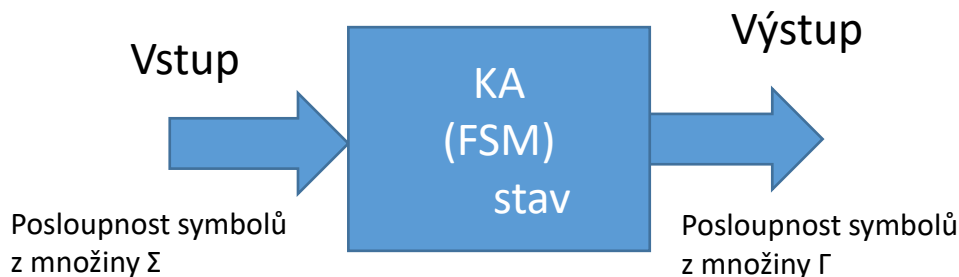
Σ ... množina vstupních symbolů

Γ ... množina výstupních symbolů

δ ... přechodová funkce

ω ... výstupní funkce

q_0 ... počáteční stav



$$\delta: Q \times \Sigma \longrightarrow Q$$

Typ Moore

$$\omega: Q \longrightarrow \Gamma$$

Typ Meally

$$\omega: Q \times \Sigma \longrightarrow \Gamma$$

Přechodová a výstupní funkce KA (Moore)

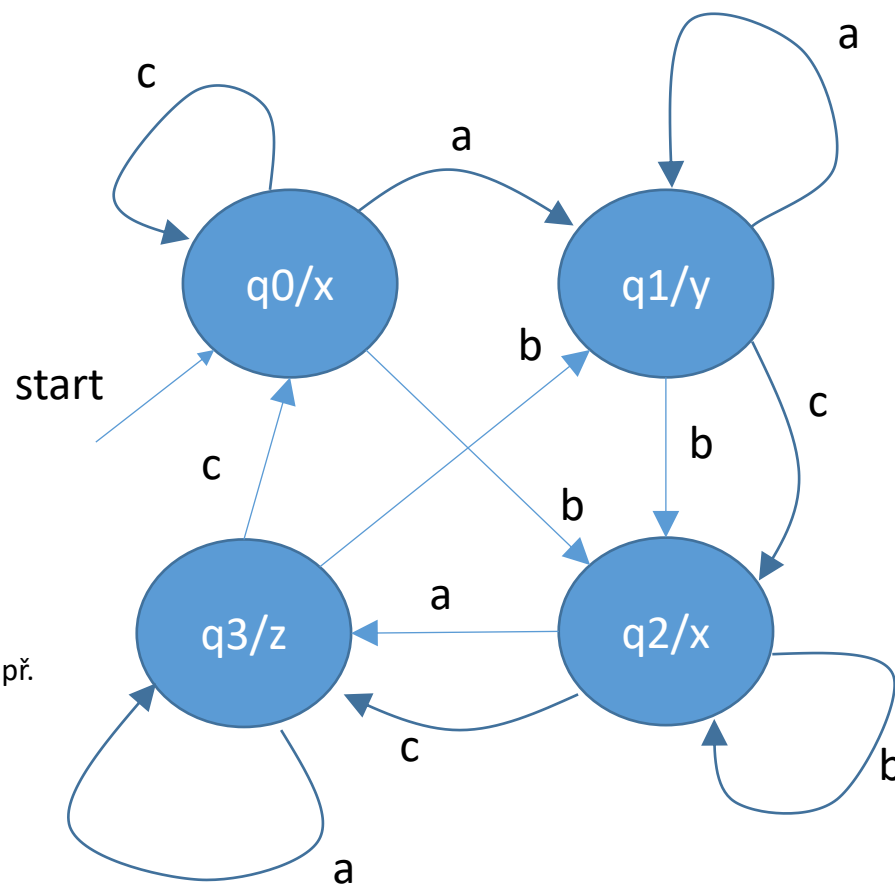
Tabulka přechodů

Q(t)	Q(t+1)		
	a	b	c
q0	q1	q2	q0
q1	q1	q2	q2
q2	q3	q2	q3
q3	q3	q1	q0

Tabulka výstupů

Q(t)	výstup
q0	x
q1	y
q2	x
q3	z

Symbols a,b,c,x,y a z musí být posléze kódovány binárně. Např. a:00, b:01, c:10; x:00, y:01, z:10.

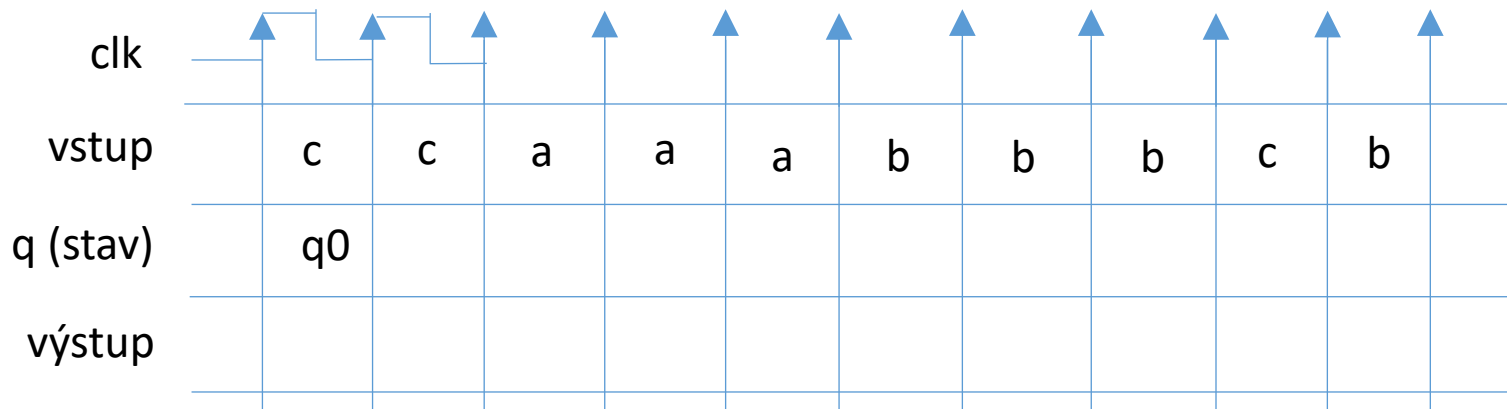
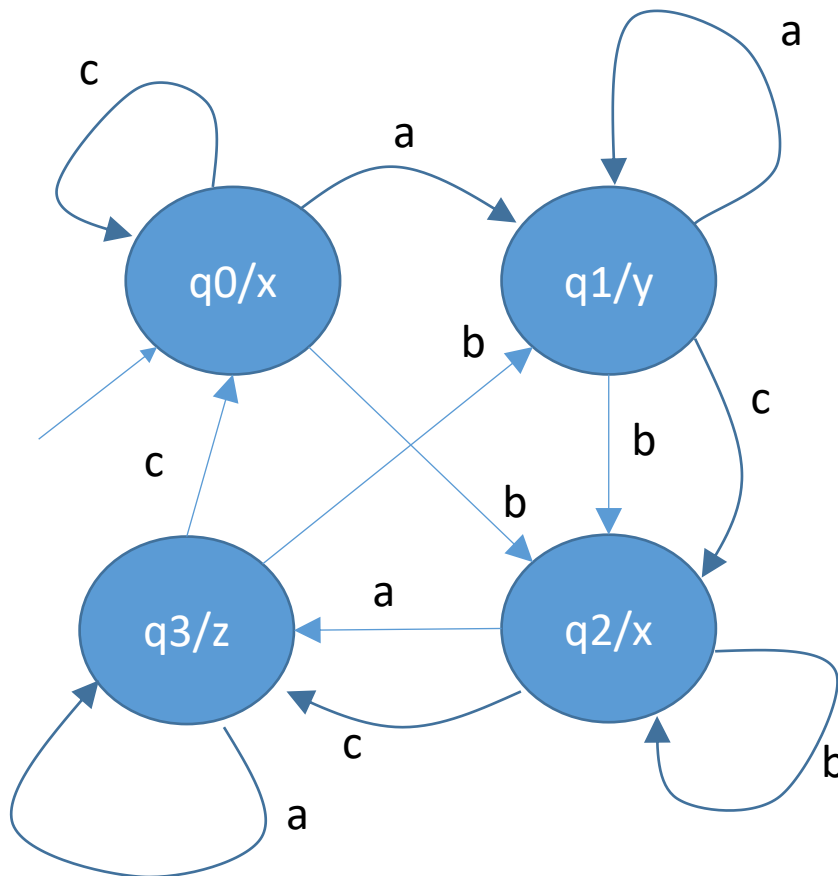


Určete výstupní posloupnost pro vstup ccaaabbbcbbaabcc

Příklad

Určete výstupní posloupnost
pro vstup ccaaabbbcbbaabcc

Počáteční stav



Přechodová a výstupní funkce KA (Moore)

Tabulka přechodů

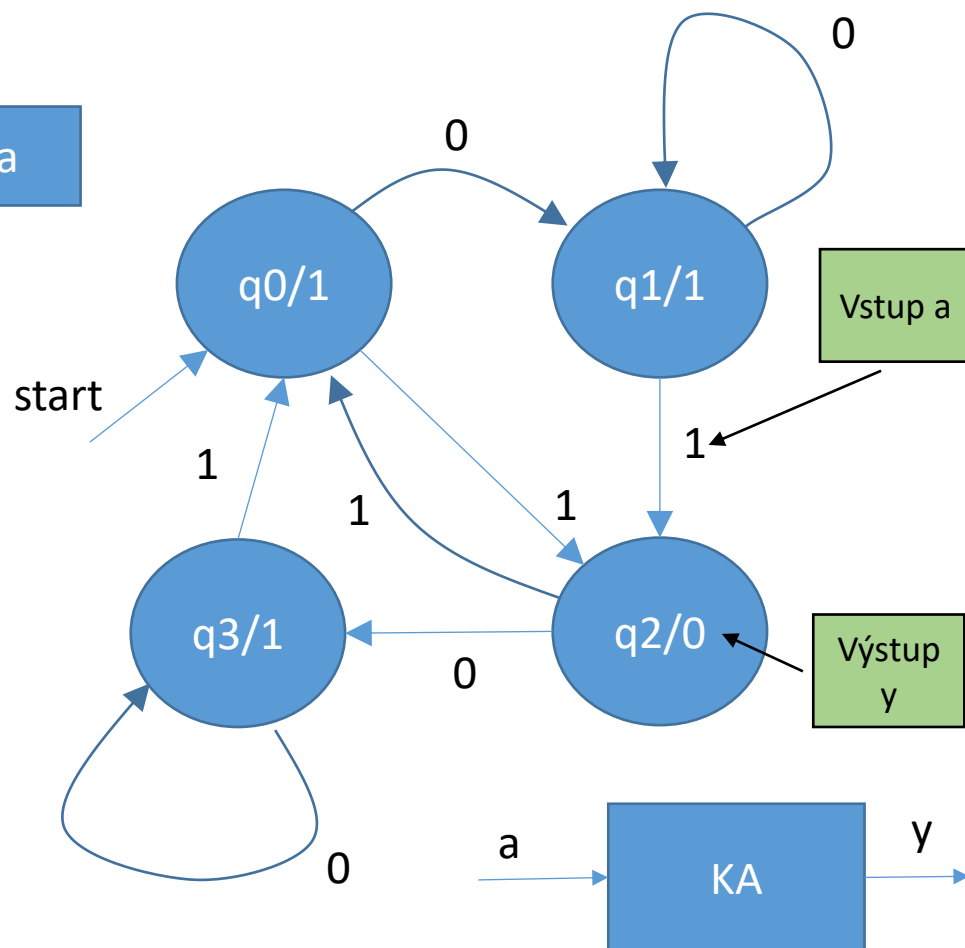
Q(t)	Q(t+1)	
	0	1
q0	q1	q2
q1	q1	q2
q2	q3	q0
q3	q3	q0

vstup a

Tabulka výstupů

Q(t)	Výstup y
q0	1
q1	1
q2	0
q3	1

Určete výstupní posloupnost pro vstup:
00010001001111111000000000011111



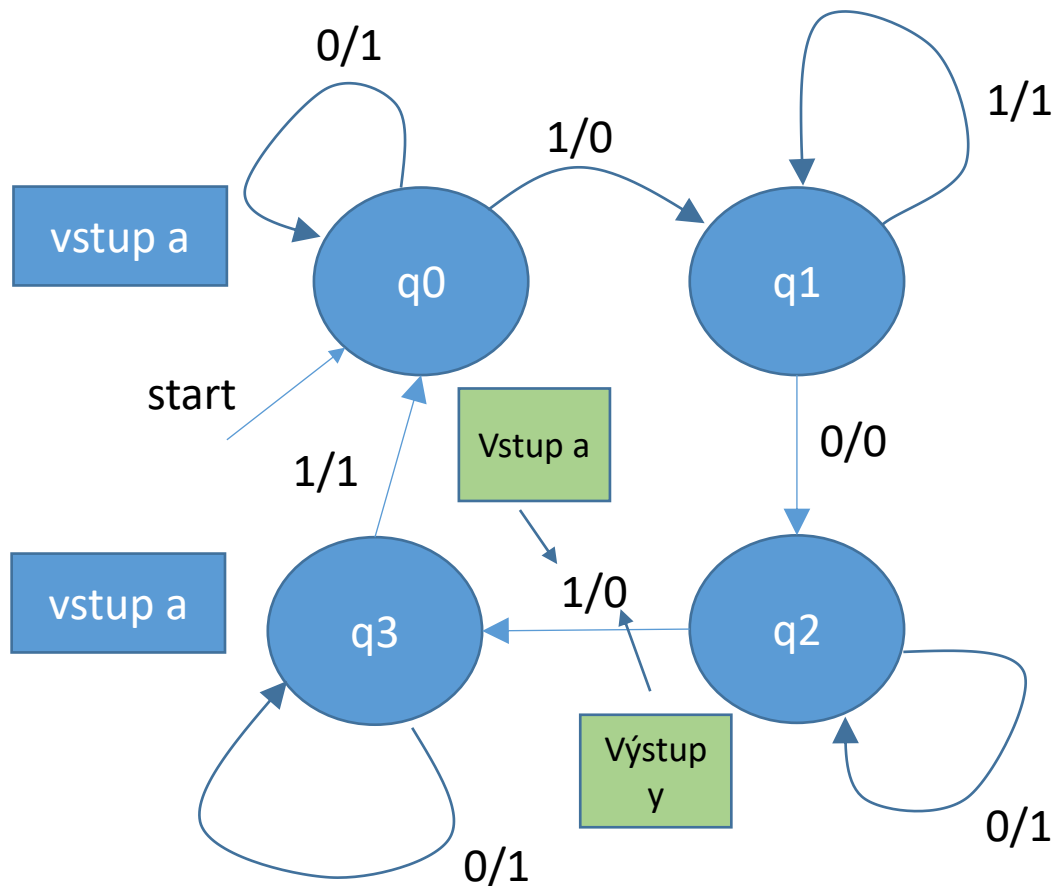
Přechodová a výstupní funkce KA (Meally)

Tabulka přechodů

Q(t)	Q(t+1)	
	0	1
q0	q0	q1
q1	q2	q1
q2	q2	q3
q3	q3	q0

Tabulka výstupů

Q(t)	Výstup y	
	0	1
q0	1	0
q1	0	1
q2	1	0
q3	1	1

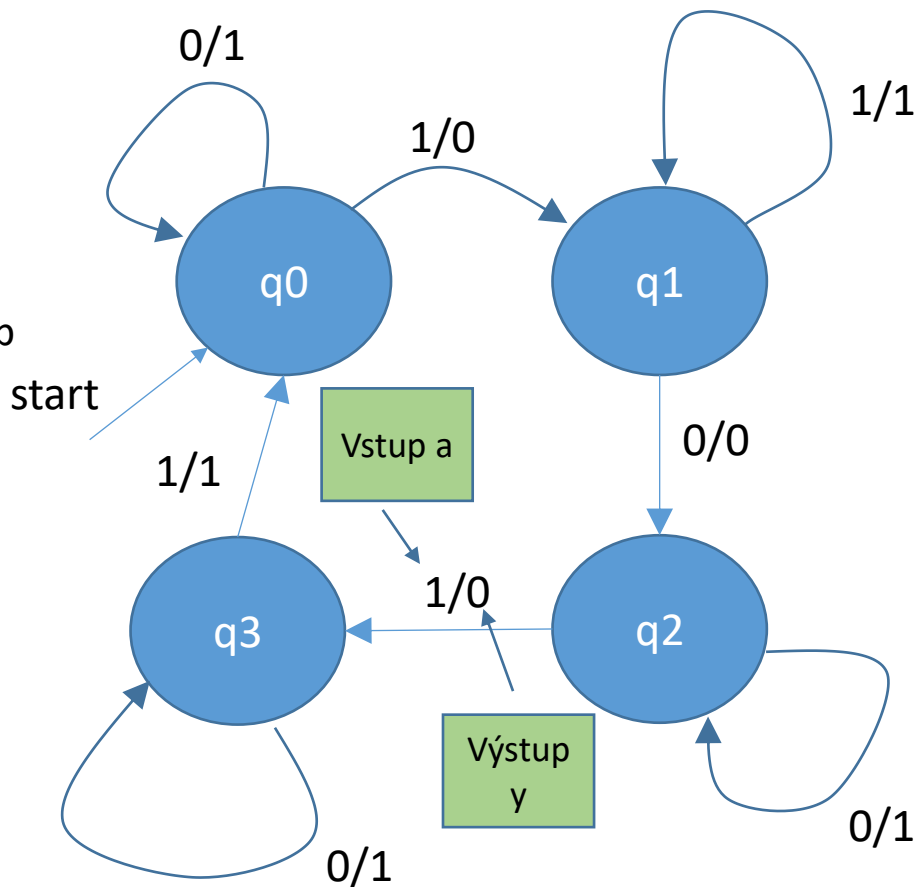
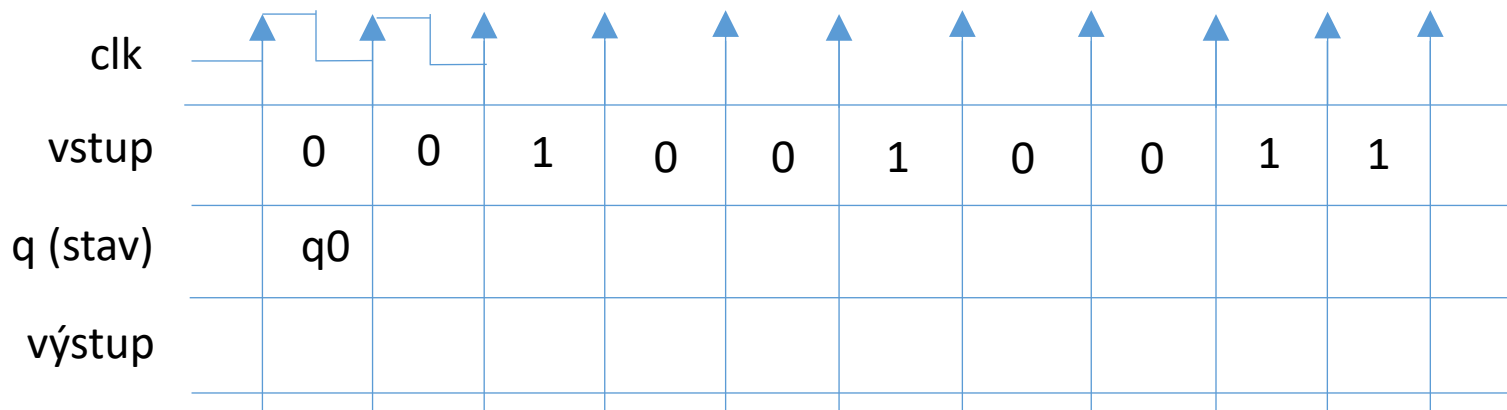


Určete výstupní posloupnost pro vstup
0010010011111100011000

Příklad

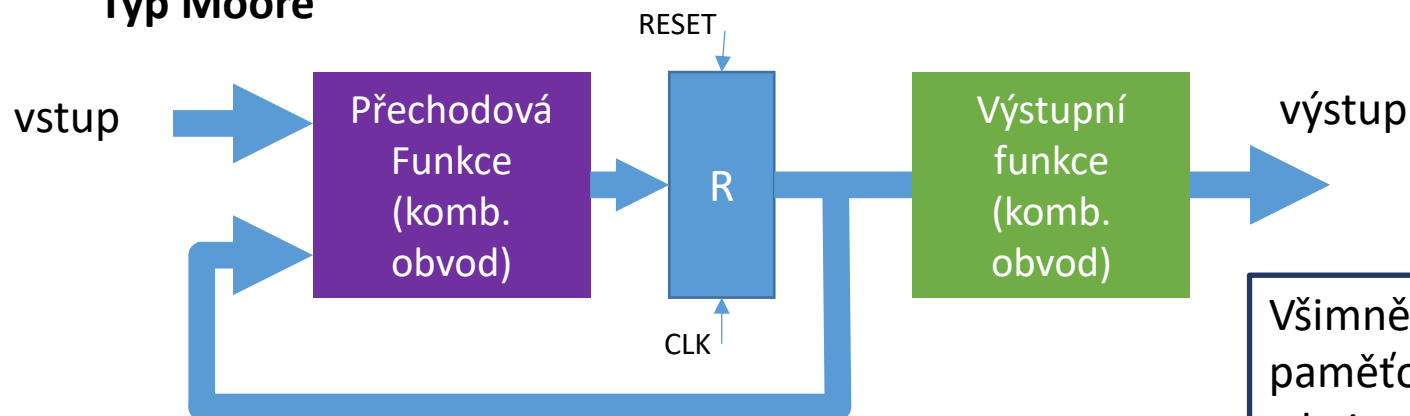
Určete výstupní posloupnost pro vstup
001001001111100011000

Počáteční stav

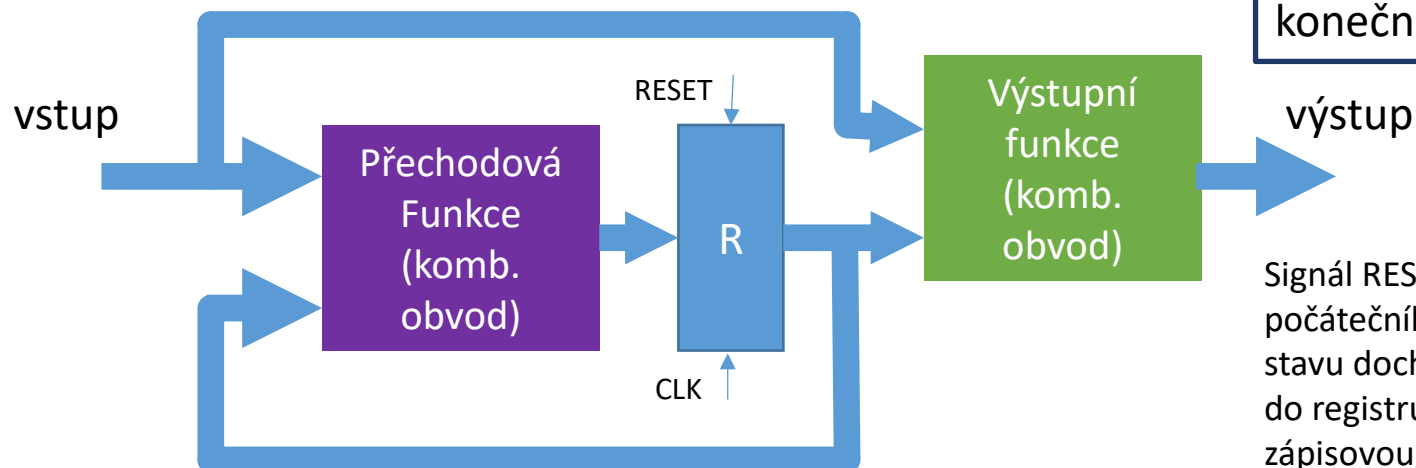


Hardwarová realizace

Typ Moore



Typ Meally



Všimněte si, že jediným paměťovým prvkem zde je registr R, který uchovává stav konečného automatu.

Signál RESET nastavuje KA do počátečního stavu. Ke změně stavu dochází s každým zápisem do registru R, tedy každou zápisovou hranou signálu CLK.

Konečný automat ve VHDL

```
signal stav: integer := 0;

process (clk, reset)
begin
    if reset = '1' then
        state <= 0;
        y <= '0';
    elsif (clk'event and clk = '1')
        case state is
            when 0 =>
                state <= 0;
                y <= '1';
            when 1 =>
                state <= 2
                y <= '0';
            ...
            when others =>
                state <= 0;
                y <= '0';
        end case;
    end if;
end process;
```

Z tohoto popisu lze v návrhových systémech pro integrované obvody nebo pro programovatelné obvody FPGA vytvořit funkční hardwarově implementovaný konečný automat.

`clk'event and clk='1'` reprezentuje vzestupnou hranu hodinového signálu. Lze číst jako změna na clk a zároveň úroveň na clk po změně je rovna log. 1.

`y` je výstup konečného automatu.
`reset` nastavuje konečný automat do počátečního stavu. Reset je asynchronní (může nastat kdykoliv)

Poznámka: symbol ... označuje zkrácení kódu na tomto slidu.

Softwarová implementace KA I

Moore

```
int stav = 0;

int prechod(int vstup) {
    int vystup = tabulka_vystupu[stav];
    stav = tabulka_prechodu[stav][vstup];
    return vystup;
}
```

Tabulka přechodů je
dvourozměrné pole

Meally

```
int stav = 0;

int prechod(int vstup) {
    int vystup = tabulka_vystupu[stav][vstup];
    stav = tabulka_prechodu[stav][vstup];
    return vystup;
}
```

Tabulka výstupů je
jednorozměrné pole (Moore)
nebo dvojrozměrné pole (Meally)

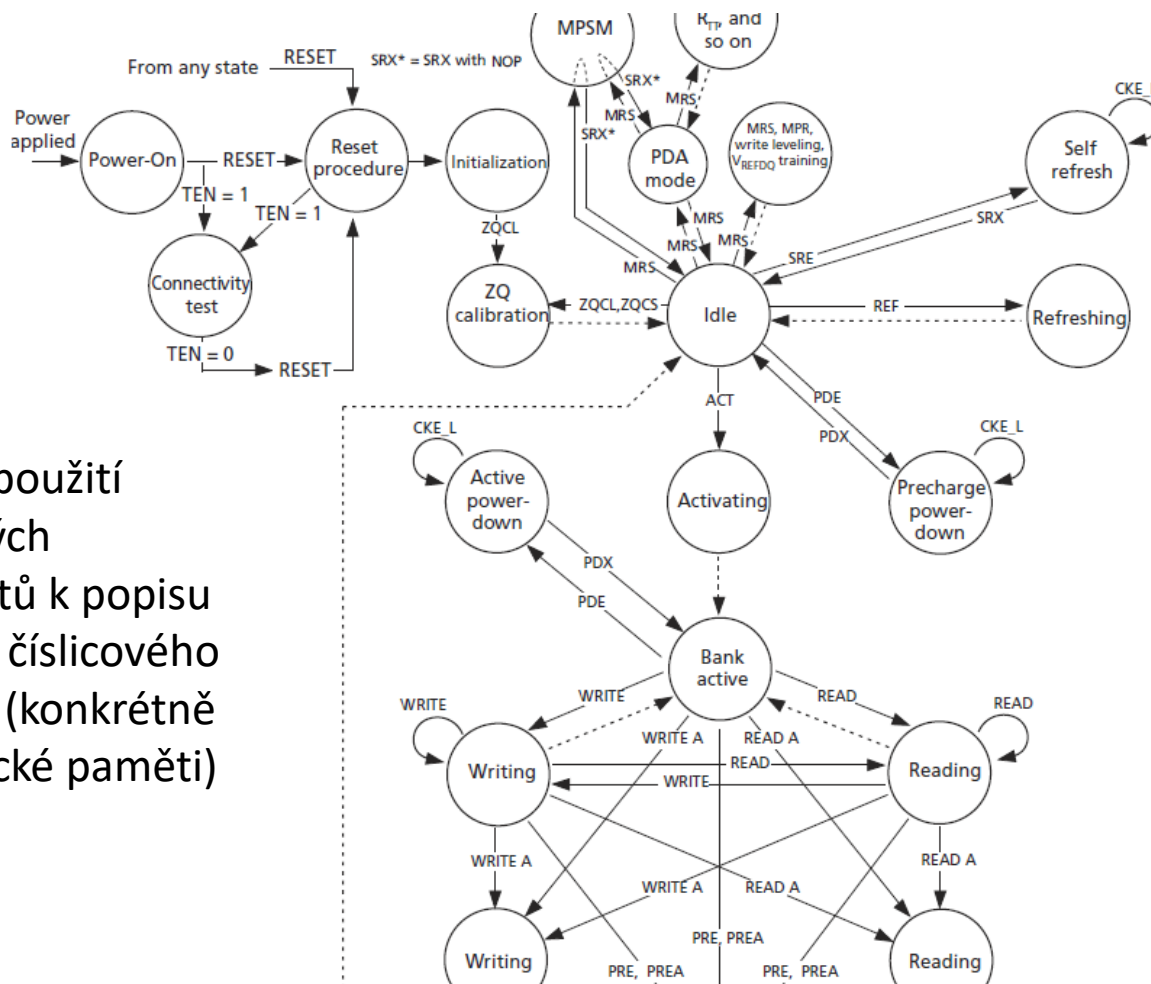
Softwarová implementace KA II

Meally

```
int stav = 0;

int prechod(int vstup) {
    int vystup = 0;
    switch(stav) {
        case 0:
            switch (vstup) {
                case 0: stav = 1; vystup = 0; break;
                case 1: stav = 0; vystup = 1; break;
            }
            break;
        case 1:
            ...
            break;
    }
    return vystup;
}
```

Výřez grafu přechodů DDR4 pamětí



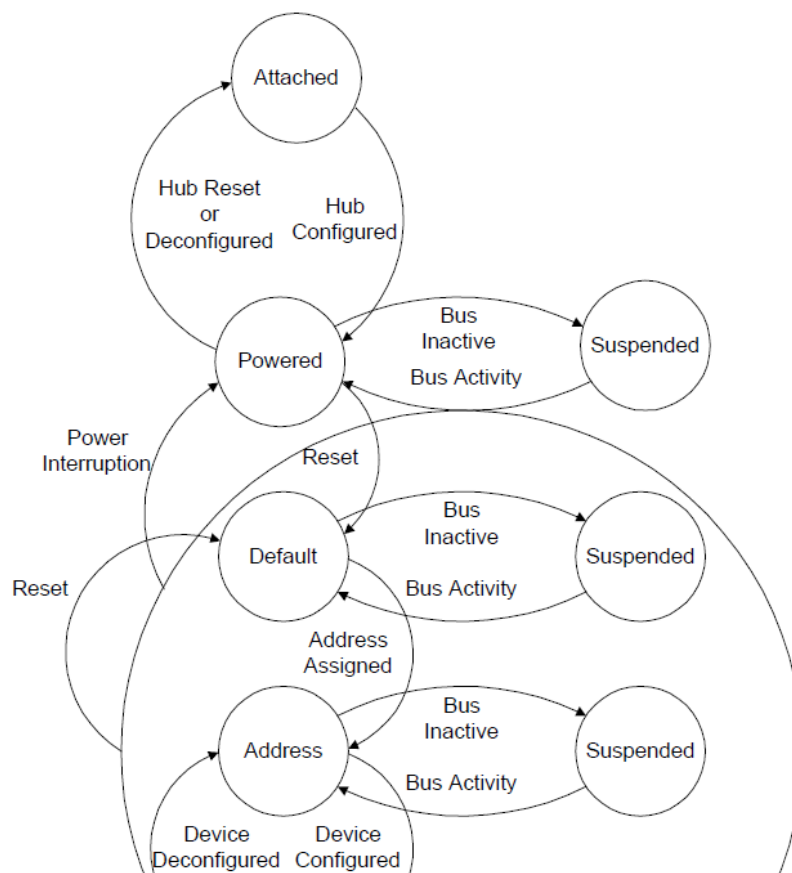
Příklad použití
konečných
automatů k popisu
chování číslicového
obvodu (konkrétně
dynamické paměti)

Převzato z DDR4 SDRAM MT40A1G4, MT40A512M8, MT40A256M16 Datasheet, Technická literatura Micron, 2014

Příklad popisu chování USB zařízení grafem přechodů konečného automatu

Stavy USB zařízení a přechody mezi nimi.

Zde je uveden pouze neúplný výřez.

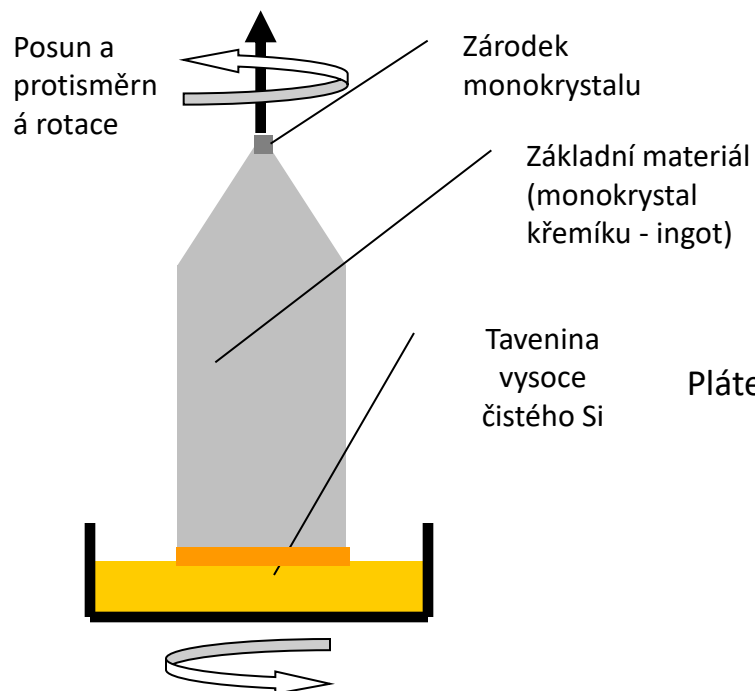


Převzato z USB 2.0 Specification, Rev. 2, April 2000, <http://www.usb.org>

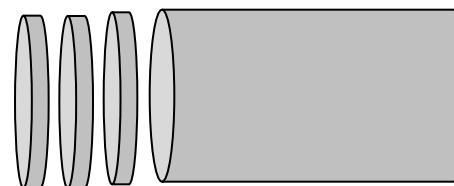
Integrované obvody

Výroba integrovaného obvodu I

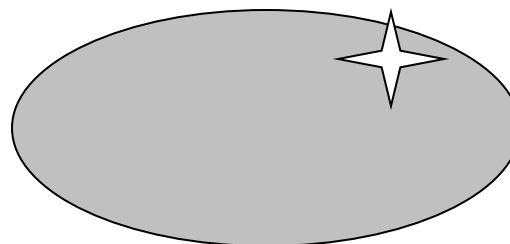
Monokrystal křemíku



Ingot se řeže na tenké plátky (wafers). Wafer připomíná tvarově DVD disk

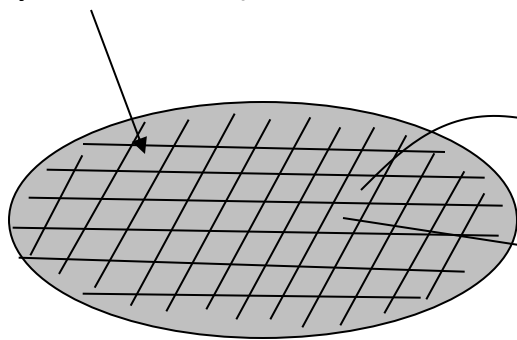


Plátek se důkladně vyleští a je připraven pro další zpracování

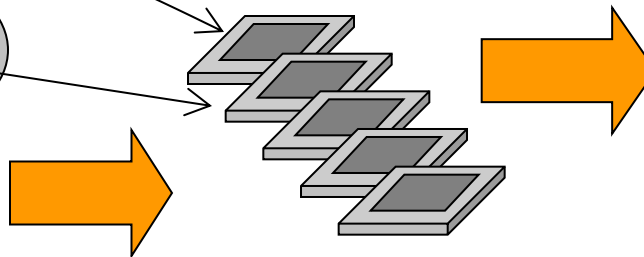


Výroba integrovaného obvodu II

Na plátku se litografickými technikami vytvoří elektrické obvody (mnoho stejných obvodů)



Plátek se rozřeže na jednotlivé čipy (die)



Čipy zapouzdří a připojí k vývodům



Hlavním parametrem litografických metod je rozlišení (jemnost struktur)

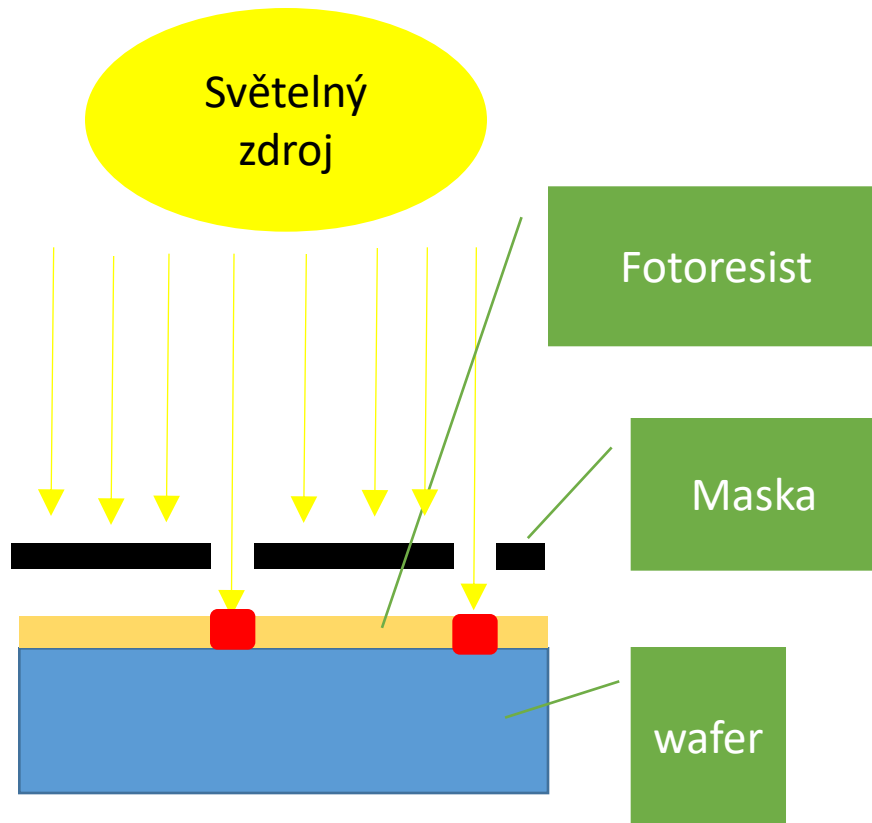
Technologie je charakterizována poloviční vzdáleností mezi identickými strukturami (paměťové buňky), velikost nejmenšího elementu (u tranzistoru)

Jednotlivé kroky procesu jsou bedlivě kontrolovány, funkční obvody jsou testovány a špatné kusy průběžně vyřazovány. Závěrečné testování proběhne po zapouzdření.

Výtěžnost = poměr mezi počtem dobrých zapouzdřených čipů a počtem započatých čipů v procentech

22nm(2012), 14nm (2016), 10 nm (2017), 7nm (2018), 5nm (~2020).

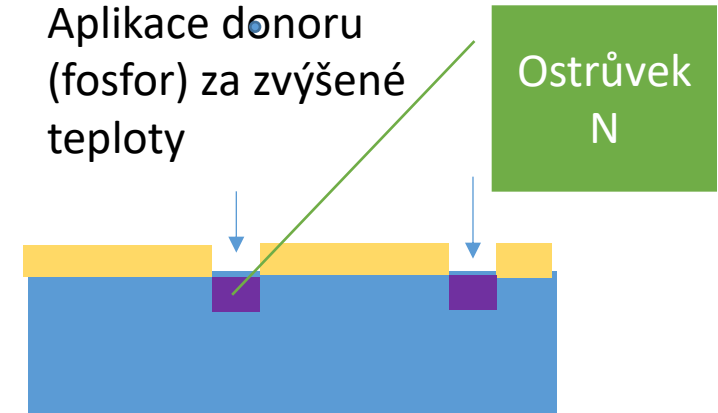
Litografie



Po odleptání
ozářených míst



Aplikace donoru
(fosfor) za zvýšené
teploty



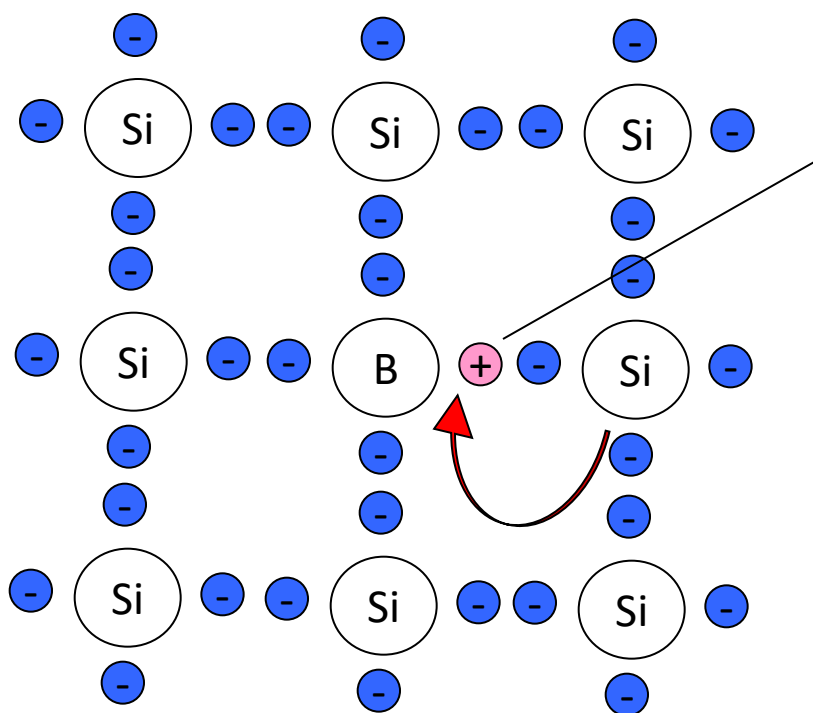
Informace, že při výrobě se používá technologie 22nm nám odráží, jak jemné mohou být struktury vytvářené na čipu.

Polovodiče - vodivost P

Křemík má ve valenční vrstvě 4 elektrony.

Bór pouze 3, jeden chybí

Děrová vodivost



Díra (chybí elektron)

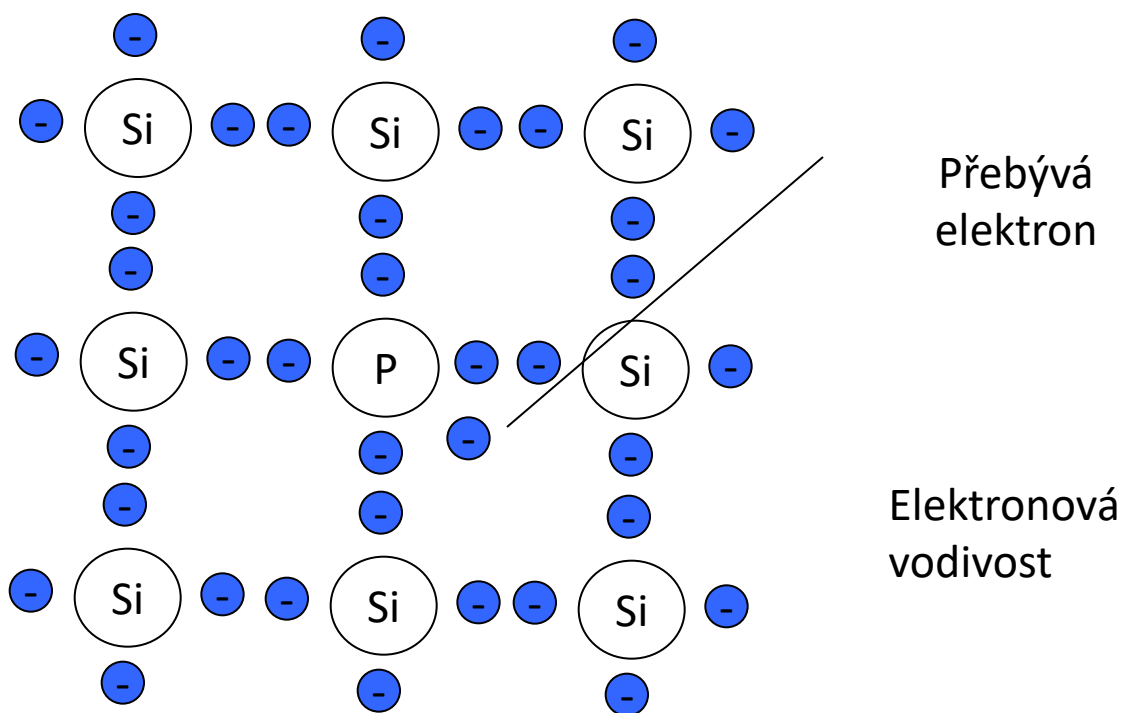
Elektrony z valenčního pásu mohou snadno přecházet do děr, a po nich zbudou opět díry. Díry jsou nositeli kladného náboje a mohou se pohybovat v elektrickém poli.

Polovodiče - vodivost N

Křemík má ve valenční vrstvě 4 elektrony.

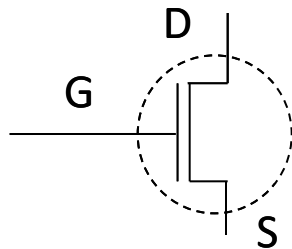
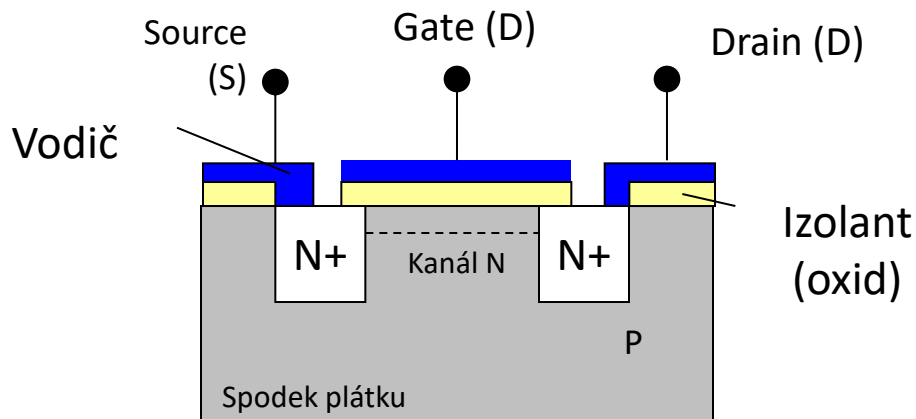
Fosfor má 5 valenčních elektronů, pouze 4 jsou potřeba na doplnění valenční vrstvy.

Přebytečný elektron snadno může přejít do vodivostní energetické hladiny a stát se volným elektronem.

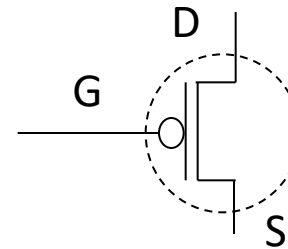
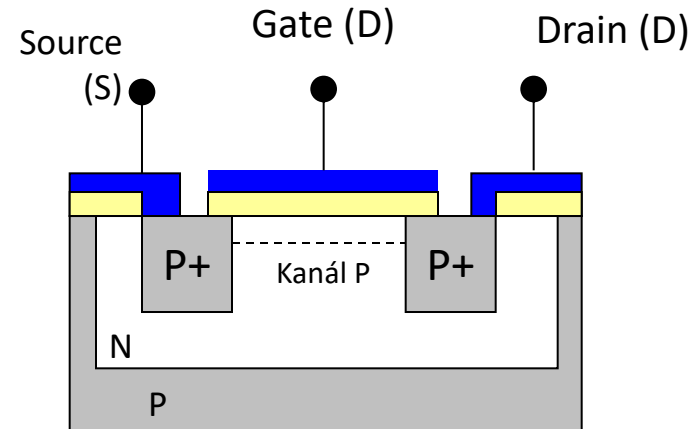


Technologie CMOS – dva komplementární tranzistory MOSFET

Tranzistor typu N (NMOS)



Tranzistor typu P (PMOS)



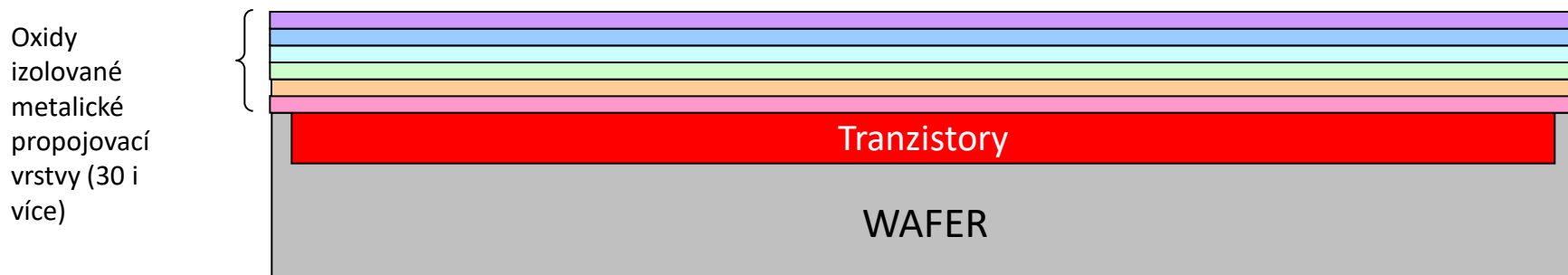
Zjednodušené
schématické
značky

Poznámky:

- tranzistory jsou v příčném řezu
- CMOS – Complementary MOS (Metal-Oxide-Semiconductor)

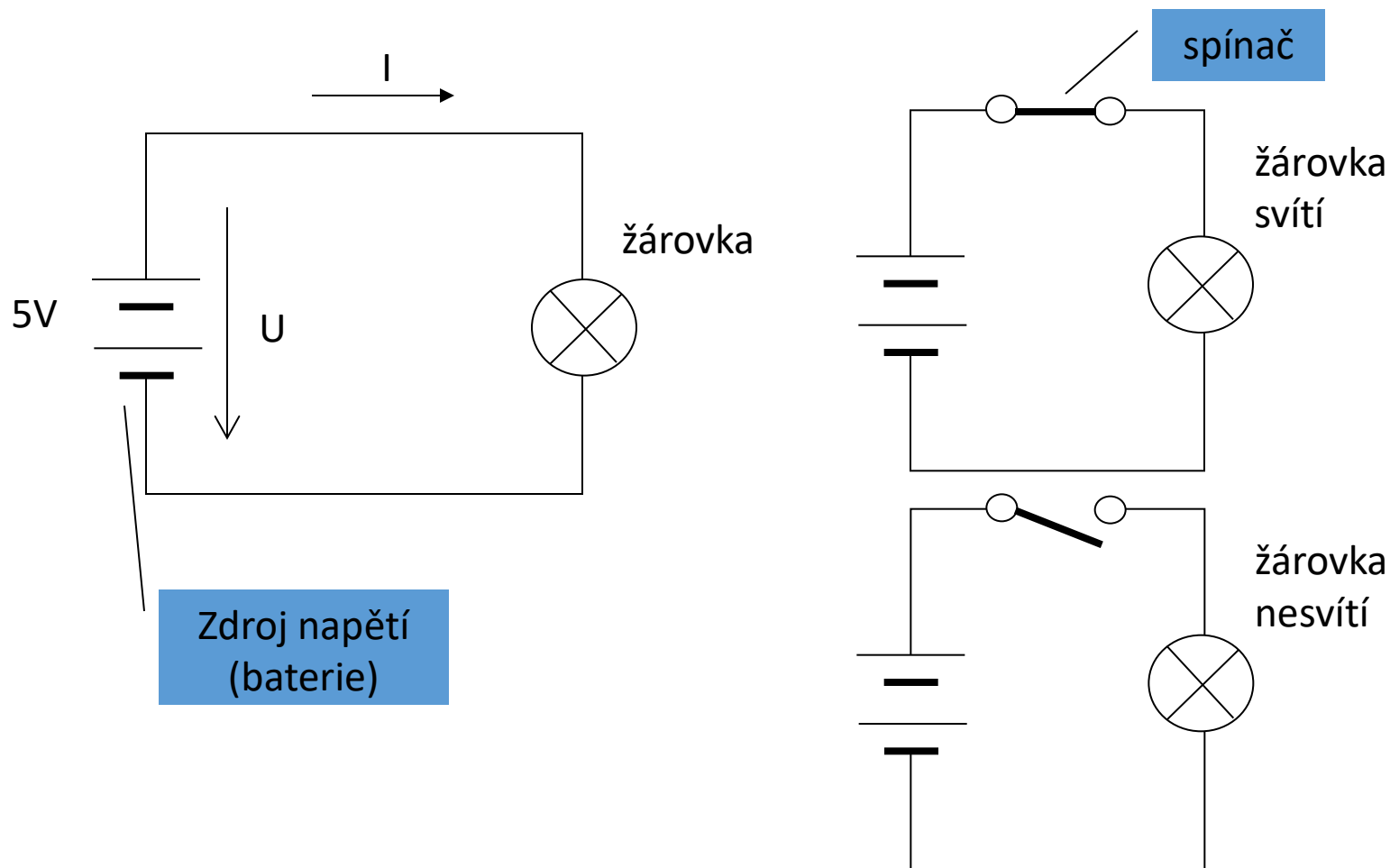
Obvody na čipu

Výkonnými prvky jsou tranzistory, které pracují jako spínače, ostatní struktury jako logické členy, registry, statické paměti jsou jen záležitostí propojení tranzistorů v metalických vrstvách.



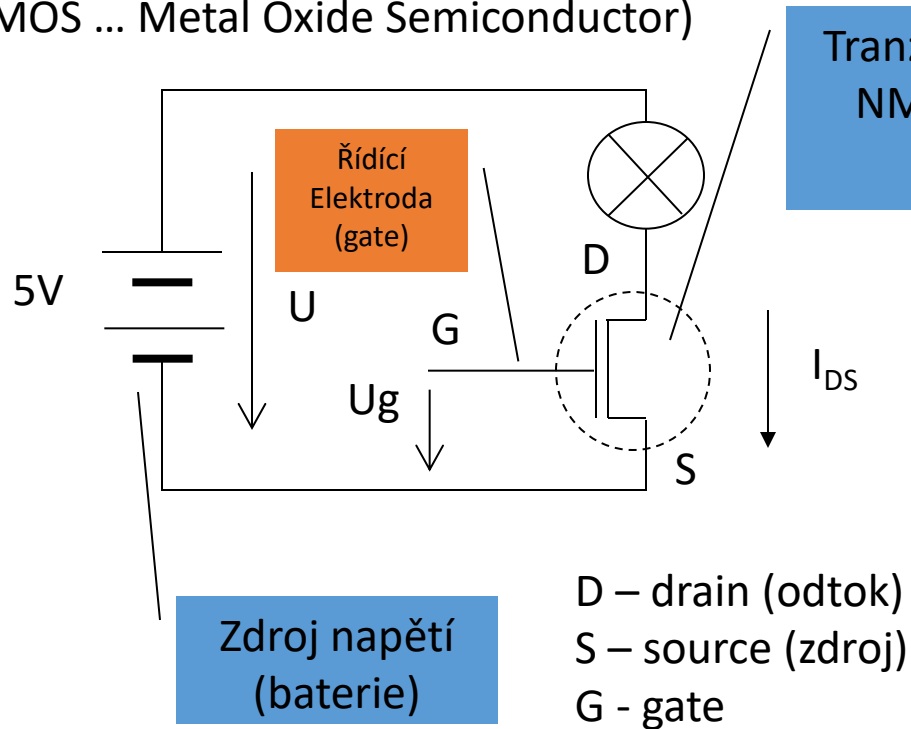
Intel Core (3. gen):	1400 mil. tranzistorů,	22nm,	2012
Intel Core 2 Duo	: 410 mil. tranzistorů,	45nm,	2008
Intel Pentium M	: 55 mil. tranzistorů,	90nm,	2003

Elektrický obvod – opakování fyzika zš, sš



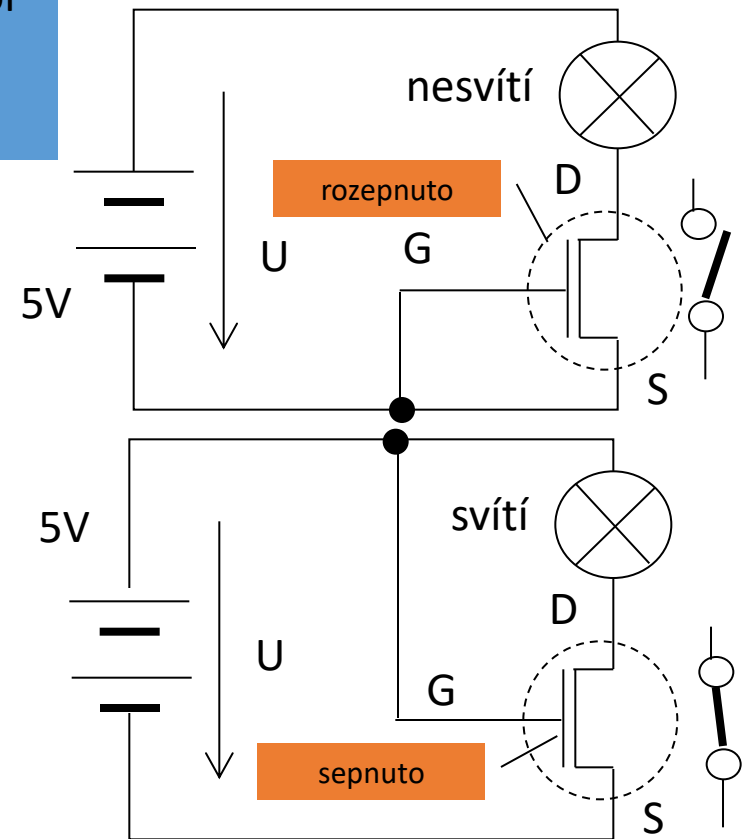
Tranzistor NMOS

(MOS ... Metal Oxide Semiconductor)

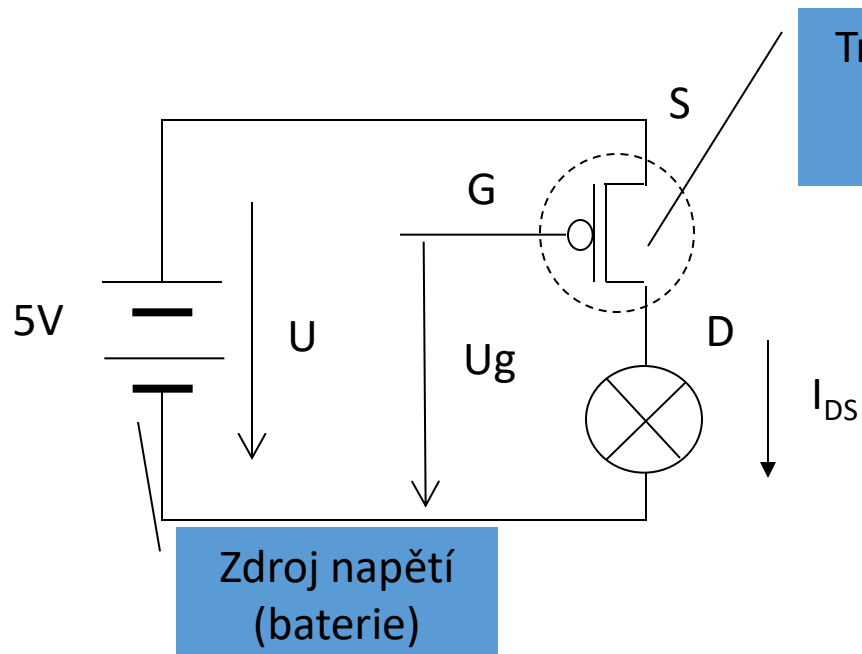


$U_g > 0,7V$... sepnuto (prochází proud)

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

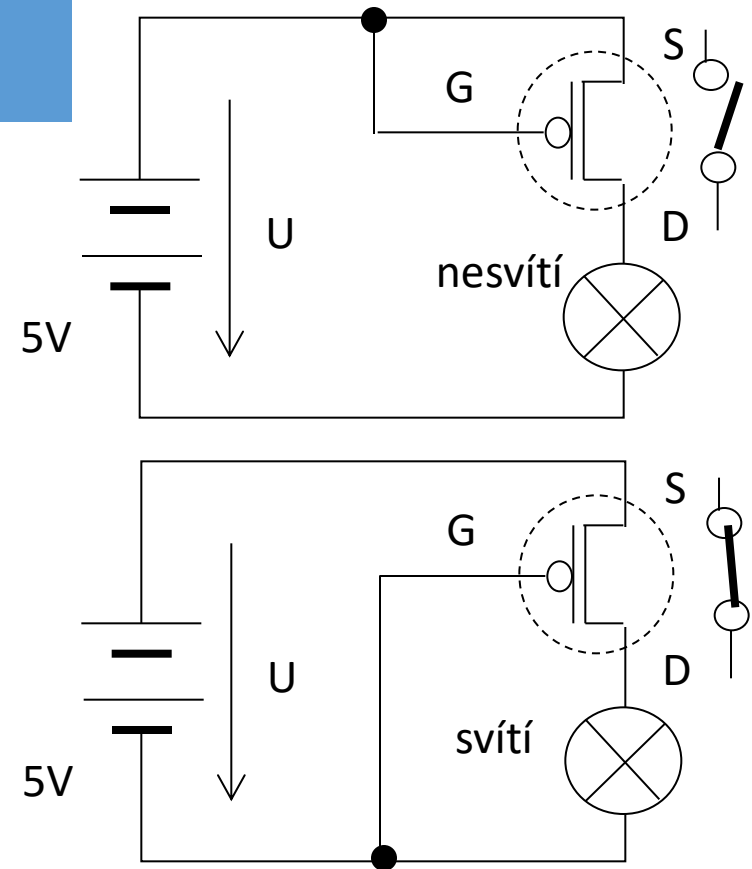


Tranzistor PMOS



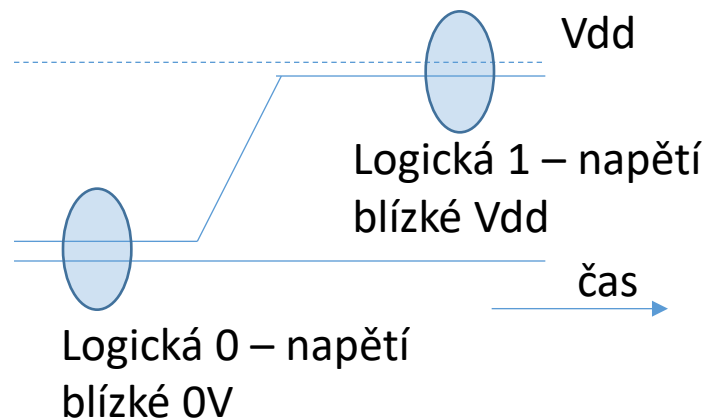
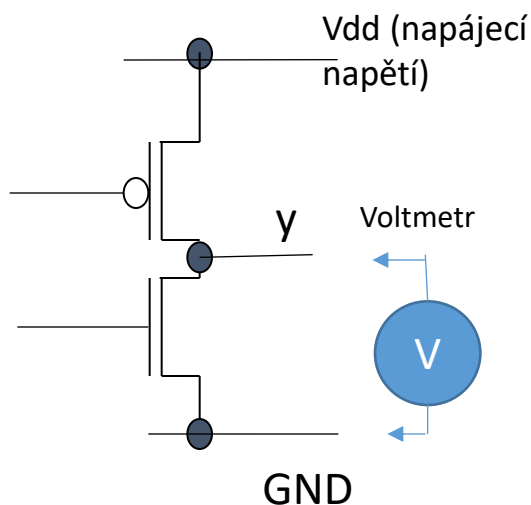
$U_g < U - 0,7V$... sepnuto (prochází proud)

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



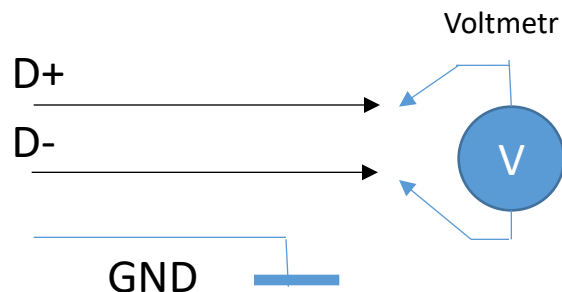
Logické úrovně

Nesymetricky (proti zemi)



Symetricky (kroucená dvoulinka)
Dovoluje rychlejší přenosy dat na větší vzdálenosti.

Sériové přenosy dat: Ethernet, USB, PCIe, Firewire

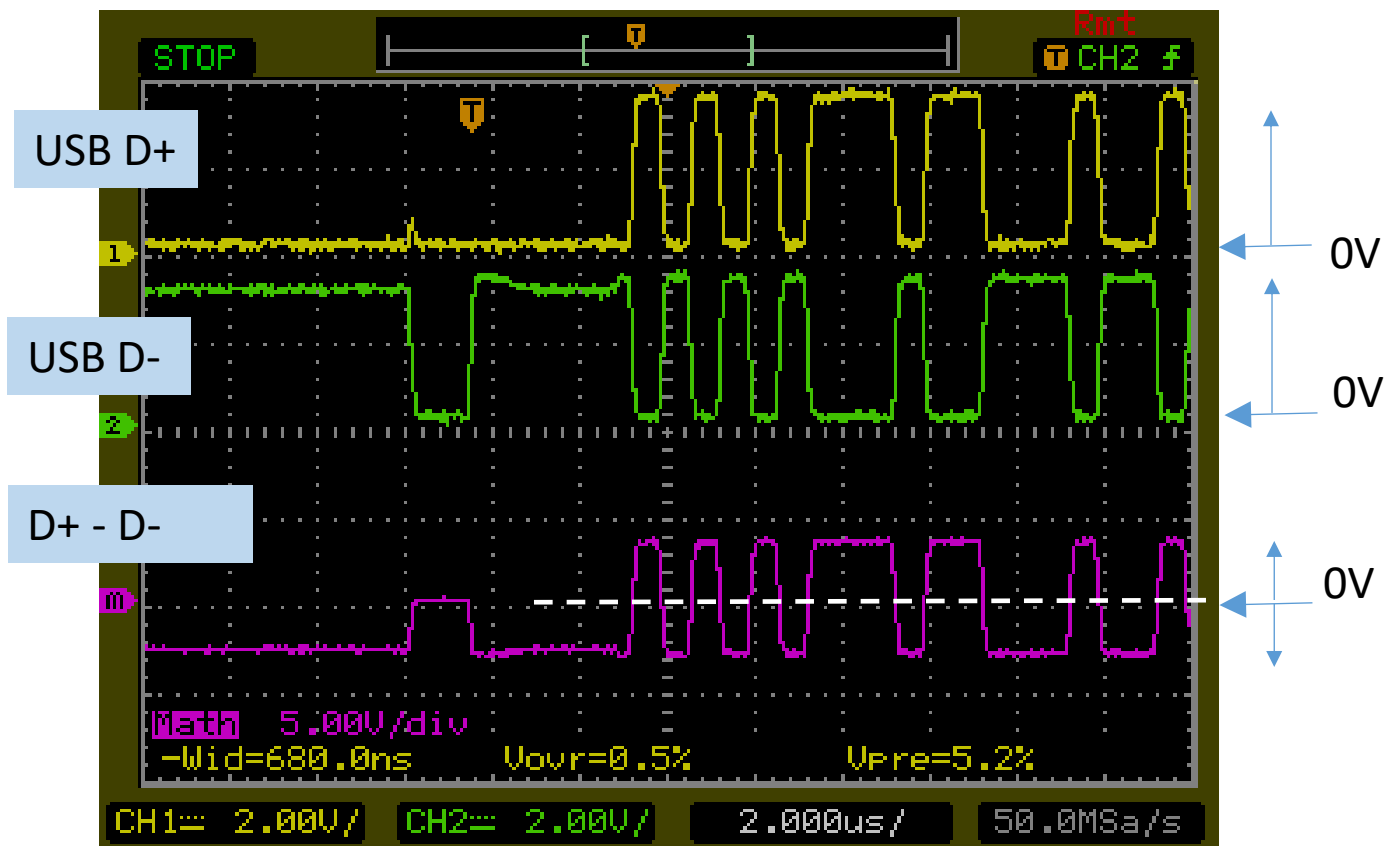


Rozdíl napětí na vodičích.

Např. USB

$D+ - D- > 200 \text{ mV}$ log. 1,
 $D- - D+ > 200 \text{ mV}$ log. 0

Logické signály na obrazovce osciloskopu



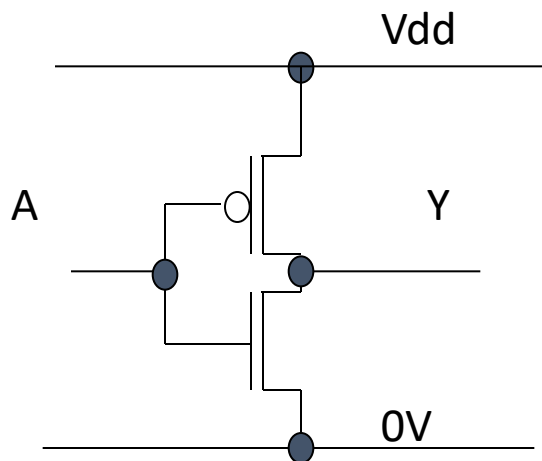
Logický obvod – invertor

Logické úrovně (zjednodušeno)

Vdd (např. 5V) ... logická jednička **1**

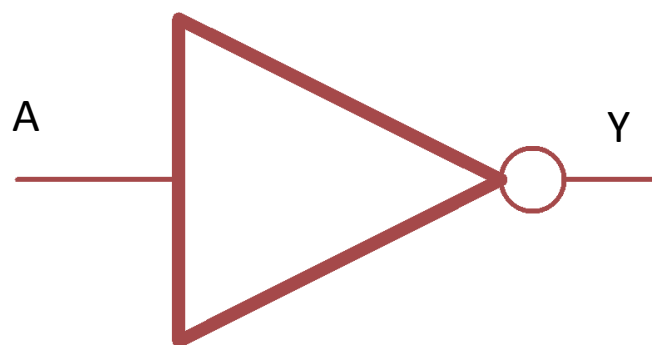
0V ... logická nula **0**

Invertor (negace)



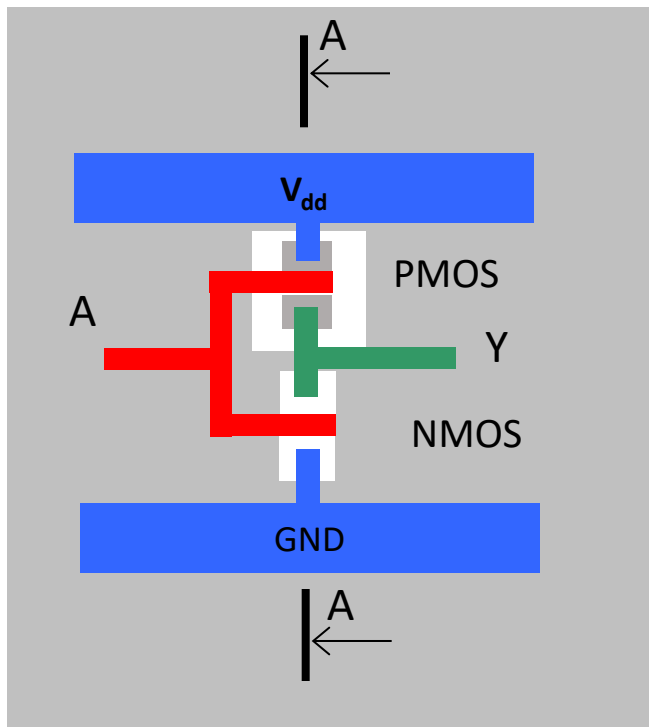
$$Y = \overline{A}$$

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

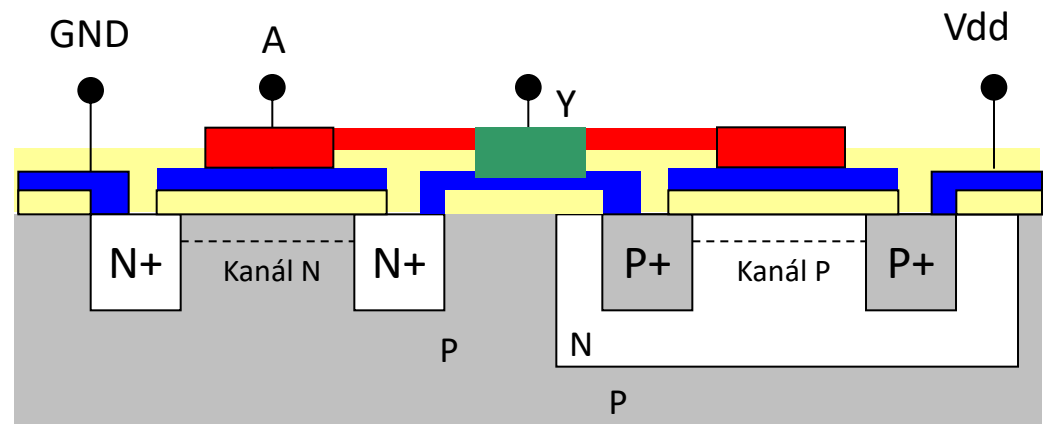


Na čipu

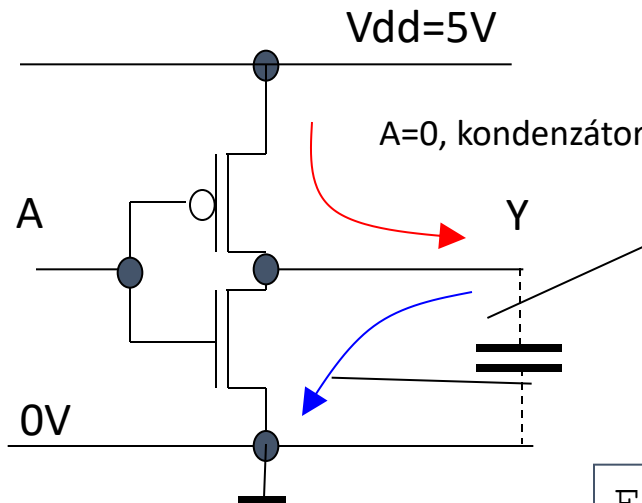
Půdorys (pohled shora)



Řez A-A



Proč procesory topí aneb spotřeba logických obvodů

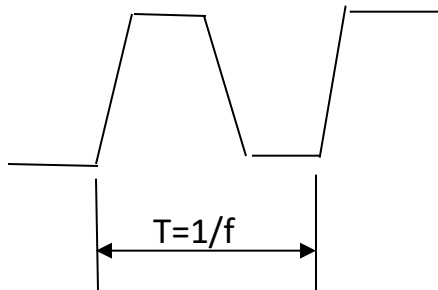


Parazitní kapacita C [pF]
(kondenzátor)

$A=1$, kondenzátor se vybíjí

$$E_c = 1/2 C U^2 \quad \text{pro jednu změnu}$$
$$P = dE/dt = 2E_c/T = 2E_c f = CU^2 f$$

A:



Pro procesor: $P = a CU^2 f$

koeficient a reflektuje aktivitu procesoru

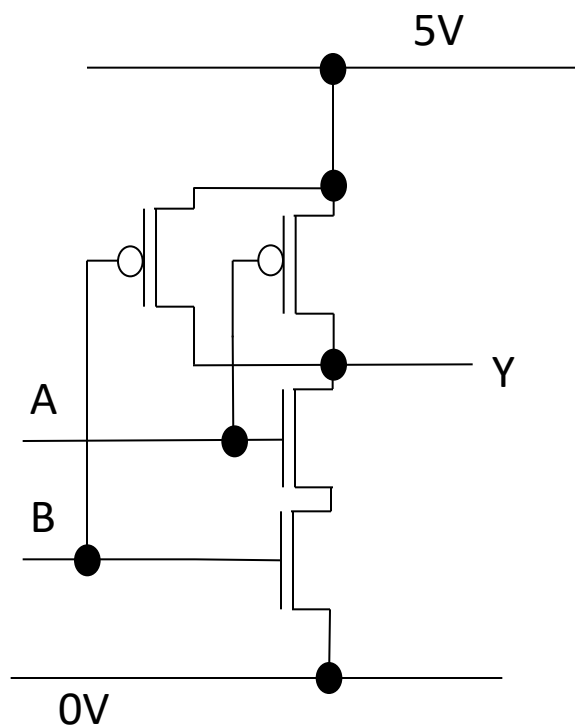
Výkon, který se na procesoru přemění v teplo roste přímo úměrně s hodinovou frekvencí a s kvadrátem napájecího napětí.

Jaké máme limity

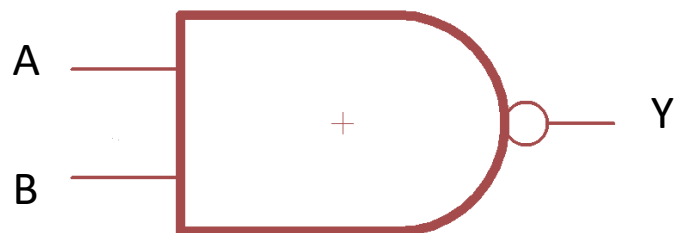
- Hodinovou frekvenci není možné příliš zvyšovat nad 3GHz, jinak procesor neuchladíme
- Snižujeme napájecí napětí
 - 5V standard dříve
 - 3,3V dnes vně procesoru (sběrnice)
 - 1,8V 1,5V dnes typicky jádro procesoru, paměti
 - Dynamická změna napětí za provozu (se snížením napájecího napětí musíme snížit i frekvenci procesoru – při nižším napětí obvody pracují pomaleji)
- Napájecí napětí nelze dále zásadně snižovat, narážíme na prahová napětí tranzistorů
- Tuto past řešíme paralelizací (vícejádrové procesory). Počet tranzistorů na čipu lze zatím stále zvyšovat zvětšováním rozlišení litografie.

Logické obvody – hradlo NAND (CMOS)

$$Y = \overline{AB}$$

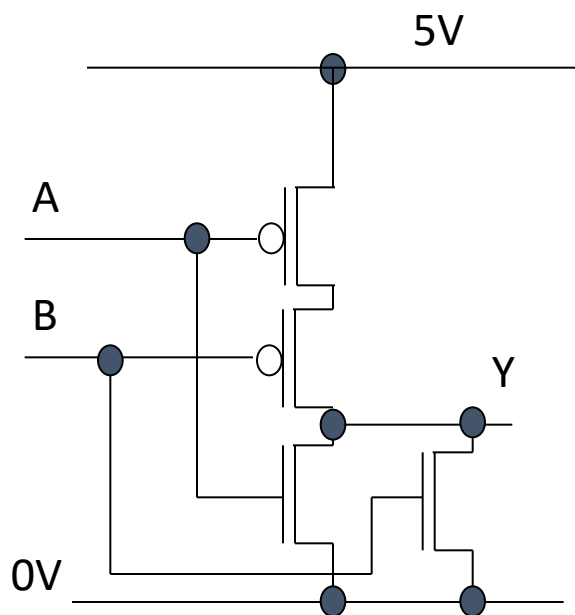


A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

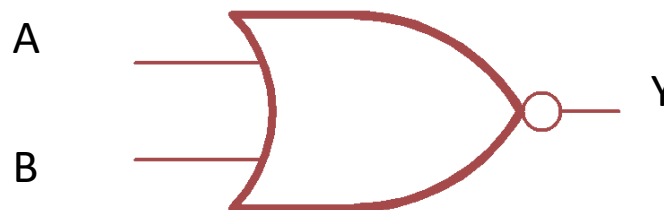


Logické obvody - hradlo NOR (CMOS)

$$Y = \overline{A + B}$$



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



Sběrnice

Sběrnice

Sběrnice je skupina vodičů, které propojují dvě a více zařízení. Tato skupina vodičů tvoří určitý logický celek nebo slouží stejnému účelu nebo pro přenos určitého typu signálu.

U sběrnic pro přenos dat musí být definováno časování sběrnice (průběhy signálů v čase), případně protokol v rámci, kterého jsou data přenášena

Charakteristickou vlastností sběrnice je jedna přenosová transakce v čase v daném směru a současný přenos dat do více zařízení najednou (broadcast).

Sběrnice rozlišujeme dle

- Účelu
- Synchronizace
- Směru přenosu dat
- Způsobu přenosu dat

Taxonomie sběrnic dle

- Účelu
- Synchronizace
- Směru přenosu dat
- Způsobu přenosu dat

Taxonomie dle účelu

- **Adresová sběrnice**

- Slouží pro přenos adresy mezi procesorem, pamětí a ostatními částmi systému

- **Datová sběrnice**

- Slouží pro přenos dat mezi procesorem, pamětí a ostatními částmi systému
- Za datovou sběrnici obecně můžeme pokládat jakoukoliv sběrnici, po které se přenášejí data

- **Řídící sběrnice**

- Slouží pro přenos řídicích signálů jako jsou například signály read (RD), write (WR), byte enable (BE)

- **Systémová sběrnice**

- Sběrnice pro přenos dat mezi procesorem, pamětí, periferiemi
- Typicky zahrnuje adresovou, datovou a řídicí sběrnici, ale může se jednat i jednu sběrnici (např. PCI) jejíž protokol implementuje přenos adresy, dat a realizaci čtecích a zápisových cyklů do paměti a periferií
- Transakce na systémové sběrnici jsou přímo vyvolány instrukcemi pro zápis/čtení paměti a ve vstupně/výstupním adresním prostoru.
- Příklady: PCI, PCIe, HyperTransport, DMI (Direct Media Interface)

- **Periferní sběrnice**

- Sběrnice mezi řadičem periferních sběrnic a periferiemi na dané sběrnici
 - USB, SATA, SAS, SCSI, SMBus
-

- **Procesorová sběrnice (Front-bus)**

- Sběrnice mezi procesorem a severním můstkem (zahrnuje přenos adresy a dat)
- HyperTransport, DMI

- **Paměťová sběrnice**

- Sběrnice mezi severním můstkem a pamětí, dnes spíše mezi procesorem a pamětí (zahrnuje přenos adresy a dat)

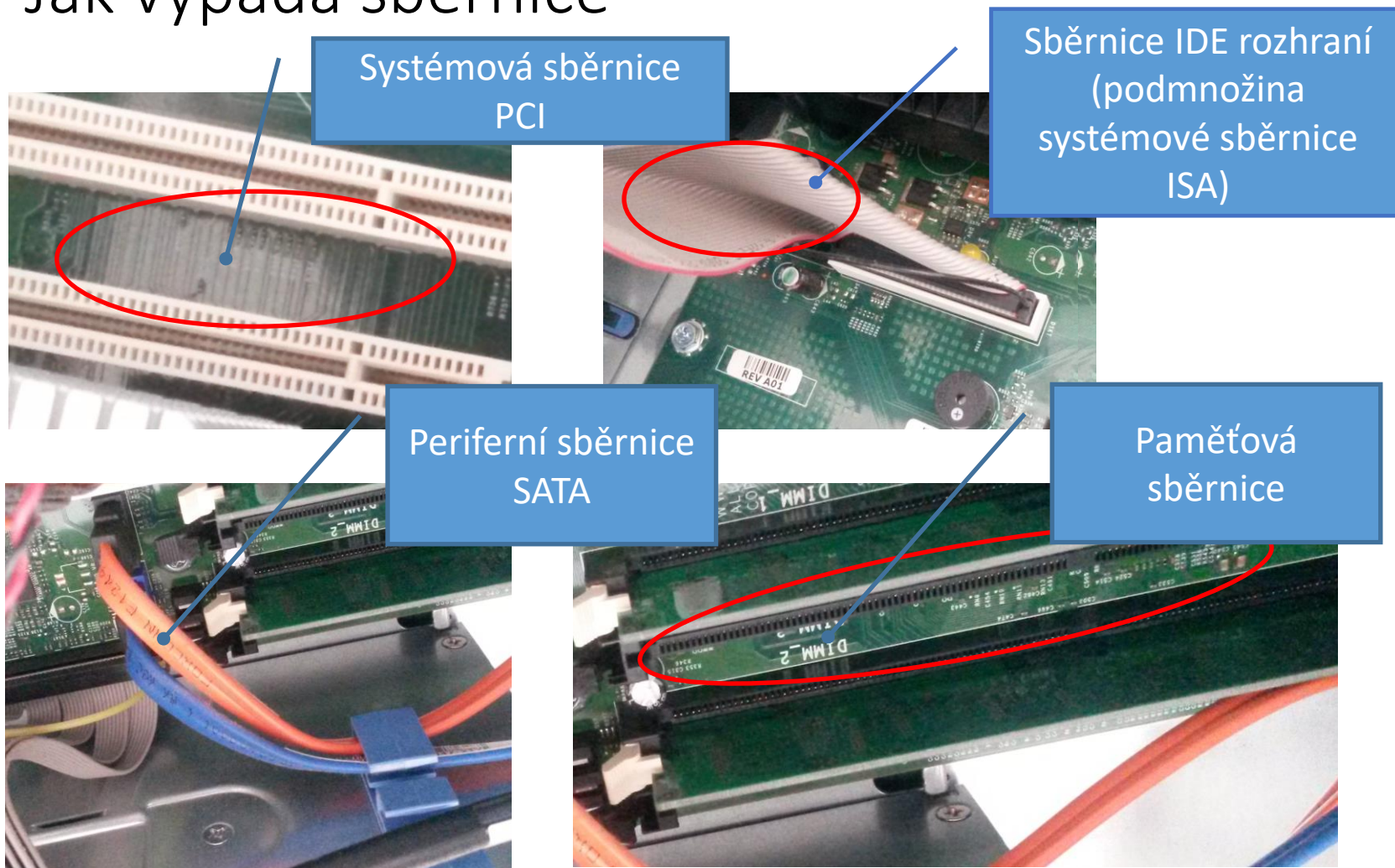
Taxonomie dle směru přenosu dat

- Jednosměrná
 - Typicky adresová sběrnice
 - Dvě jednosměrné sběrnice ale v opačném směru realizuje full duplexní přenos dat mezi dvěma body
 - Umožňuje přenos dat z jednoho místa na více míst současně (broadcast)
- Obousměrná
 - Přenos jedním a druhým směrem se multiplexuje v čase
 - Přenos jedním směrem nemůže probíhat současně s přenosem v druhém směru
 - Mluvíme často o tzv. half duplexu
 - Data můžeme současně přenášet na více míst
 - Pokud je třeba přenášet data z více míst propojených sběrnicí, musí se vyloučit kolize. Ze kterého místa se budou v dané okamžiku přenášet data rozhoduje proces zvaný arbitrace sběrnice.

Taxonomie sběrnic dle způsobu přenášení dat

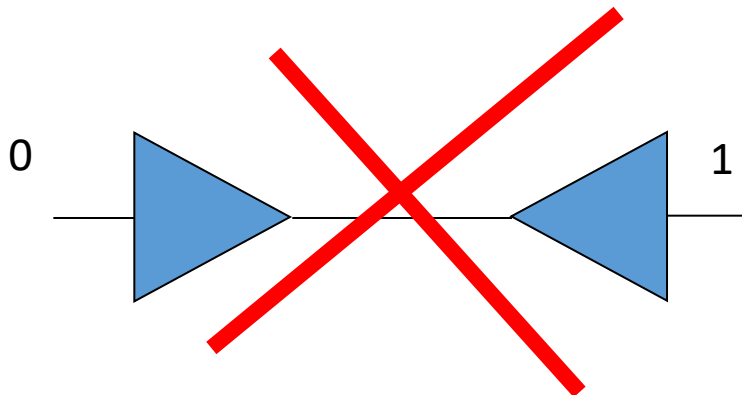
- Paralelní sběrnice
 - Přenos dat probíhá paralelně po více vodičích např. 32 bitů
 - Data musí dorazit do cíle současně. Při dnešních rychlostech přenosu hraje roli délka jednotlivých vodičů (kompenzace nestejně délky meandry)

Jak vypadá sběrnice

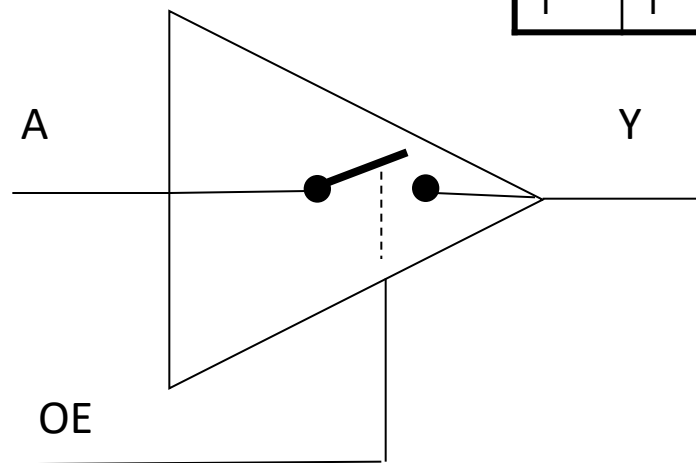


Třístavové budiče

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



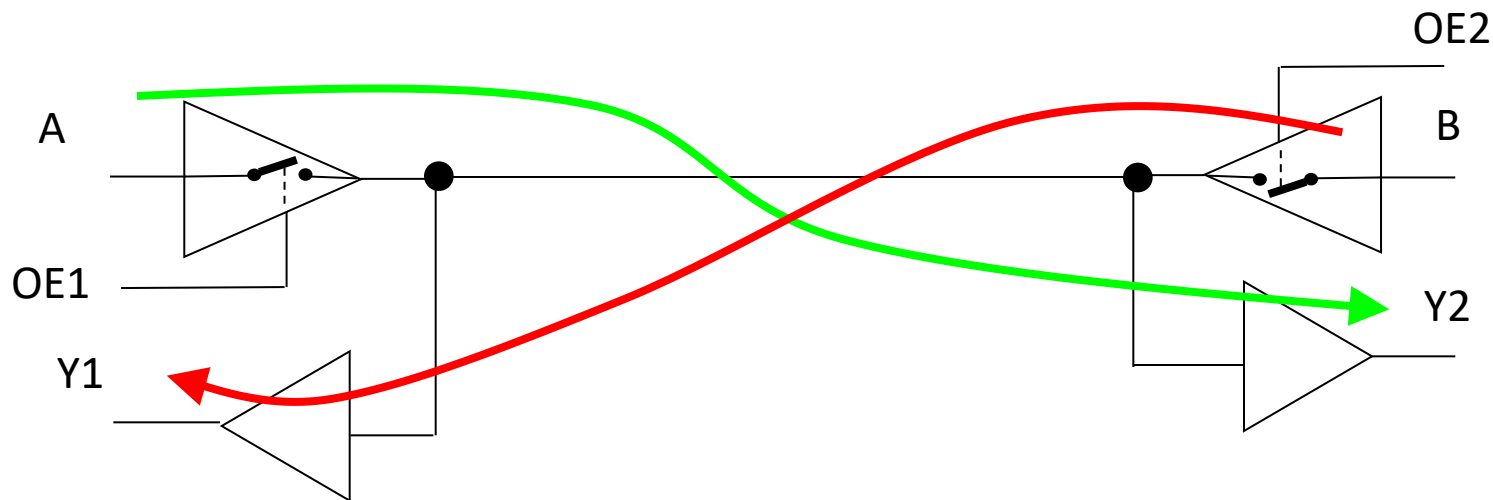
Třístavový budič



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

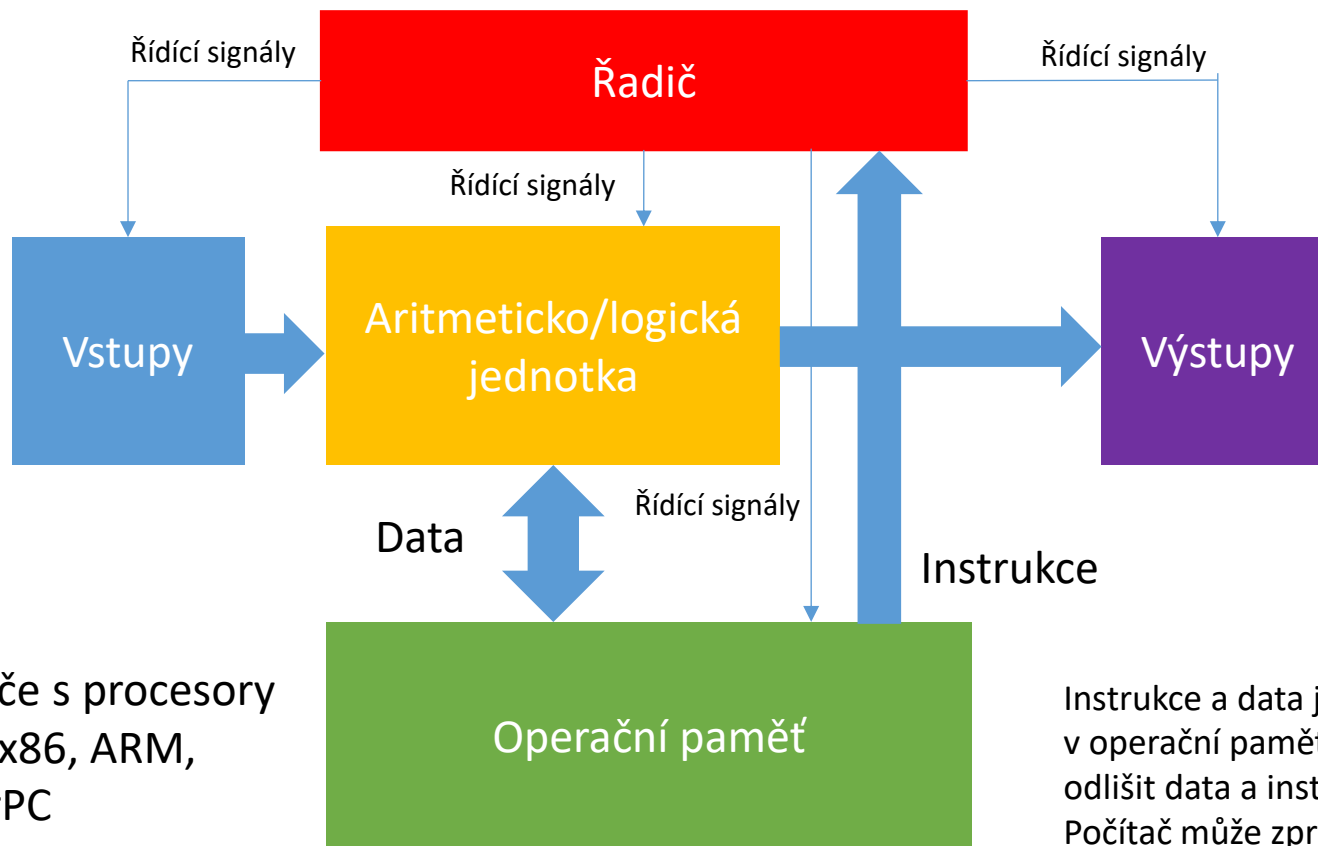


Přenos zleva doprava: $OE1=1$, $OE2=0$ (musí být! nula, jinak zkrat), $Y2$ má stejnou hodnotu jako A

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

Základní architektury počítačů

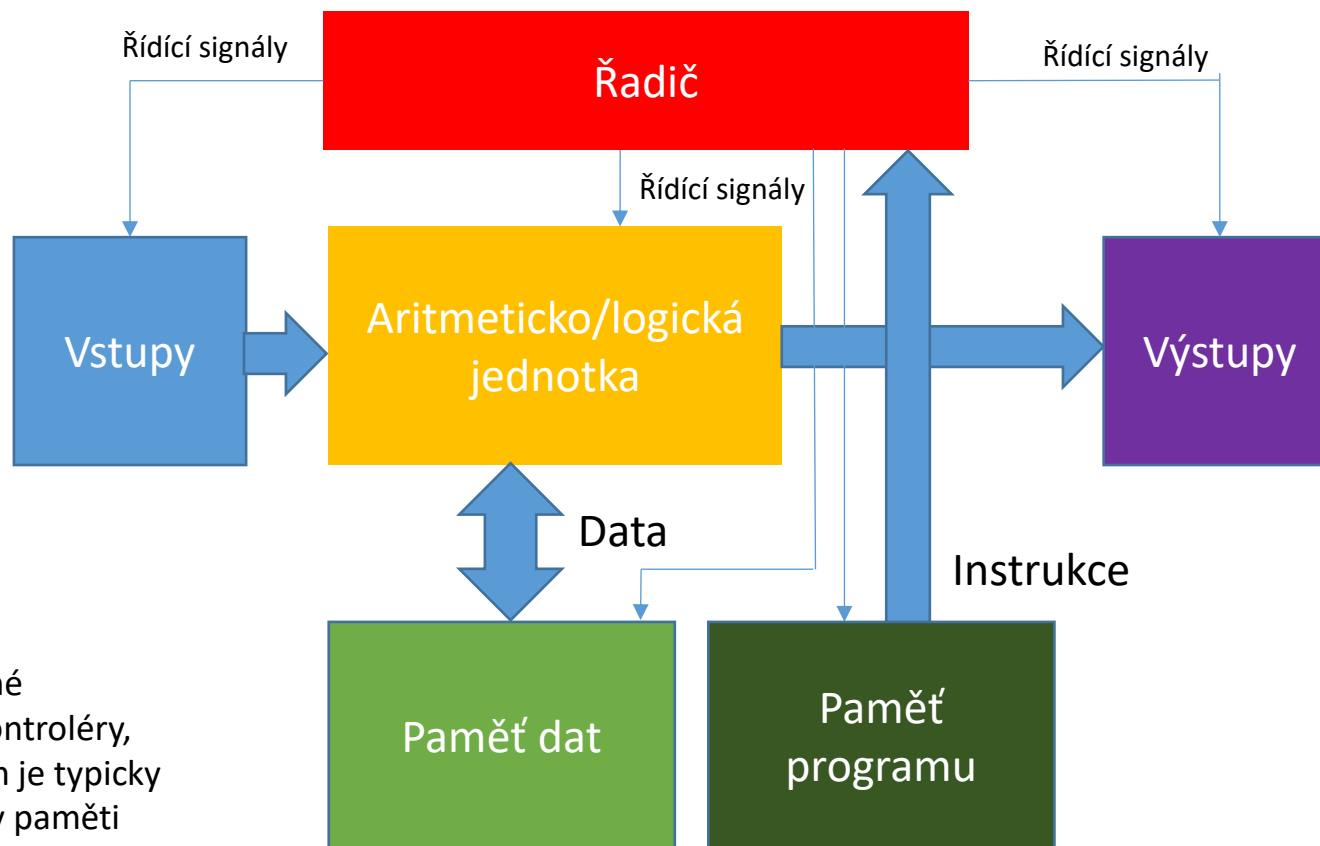
Von Neumannova architektura



Počítače s procesory
např. x86, ARM,
PowerPC

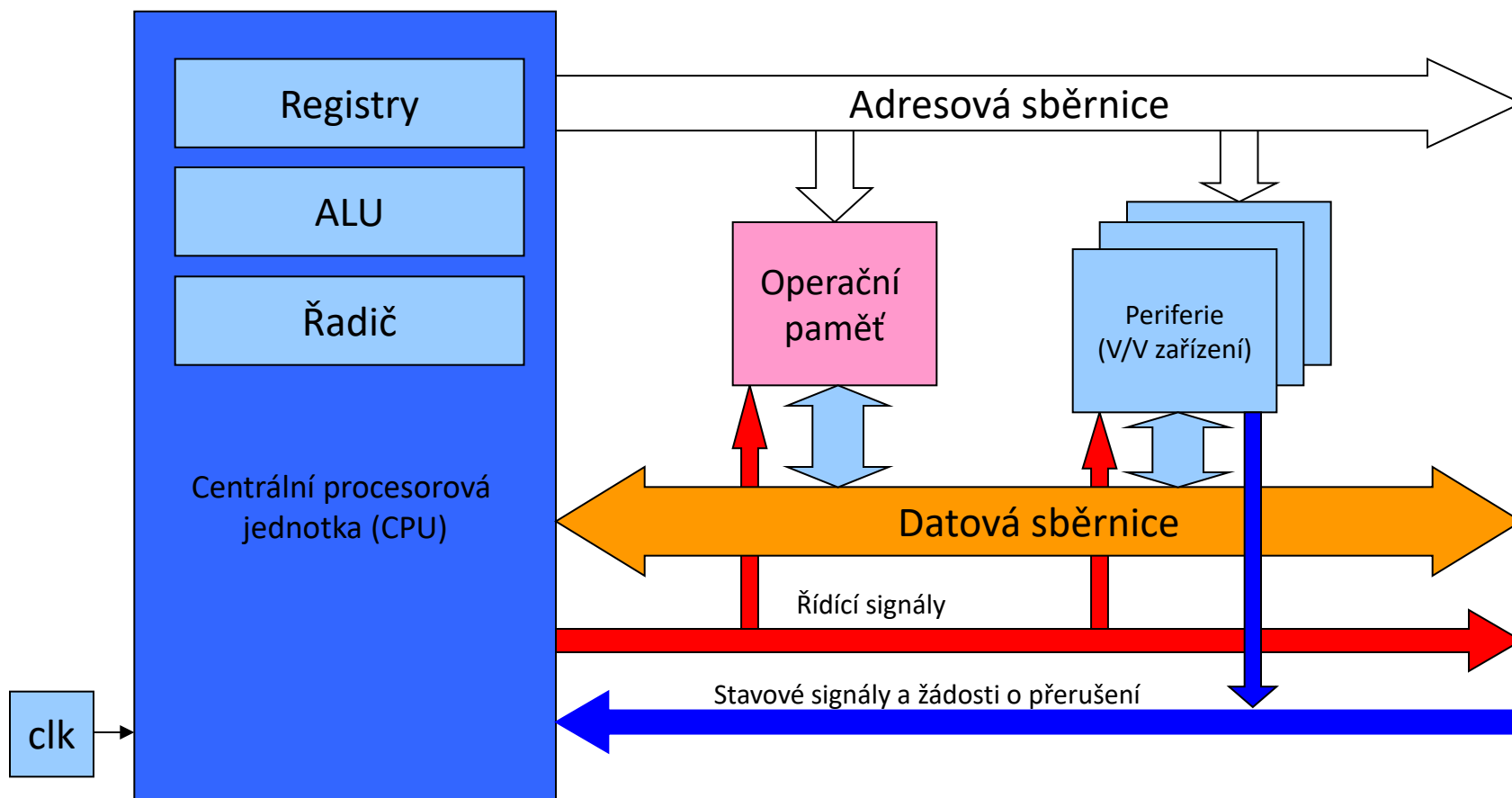
Instrukce a data jsou uložena
v operační paměti. Nelze
odlišit data a instrukce.
Počítač může zpracovat data
jako instrukci a naopak
instrukci jako data.

Harvardská architektura

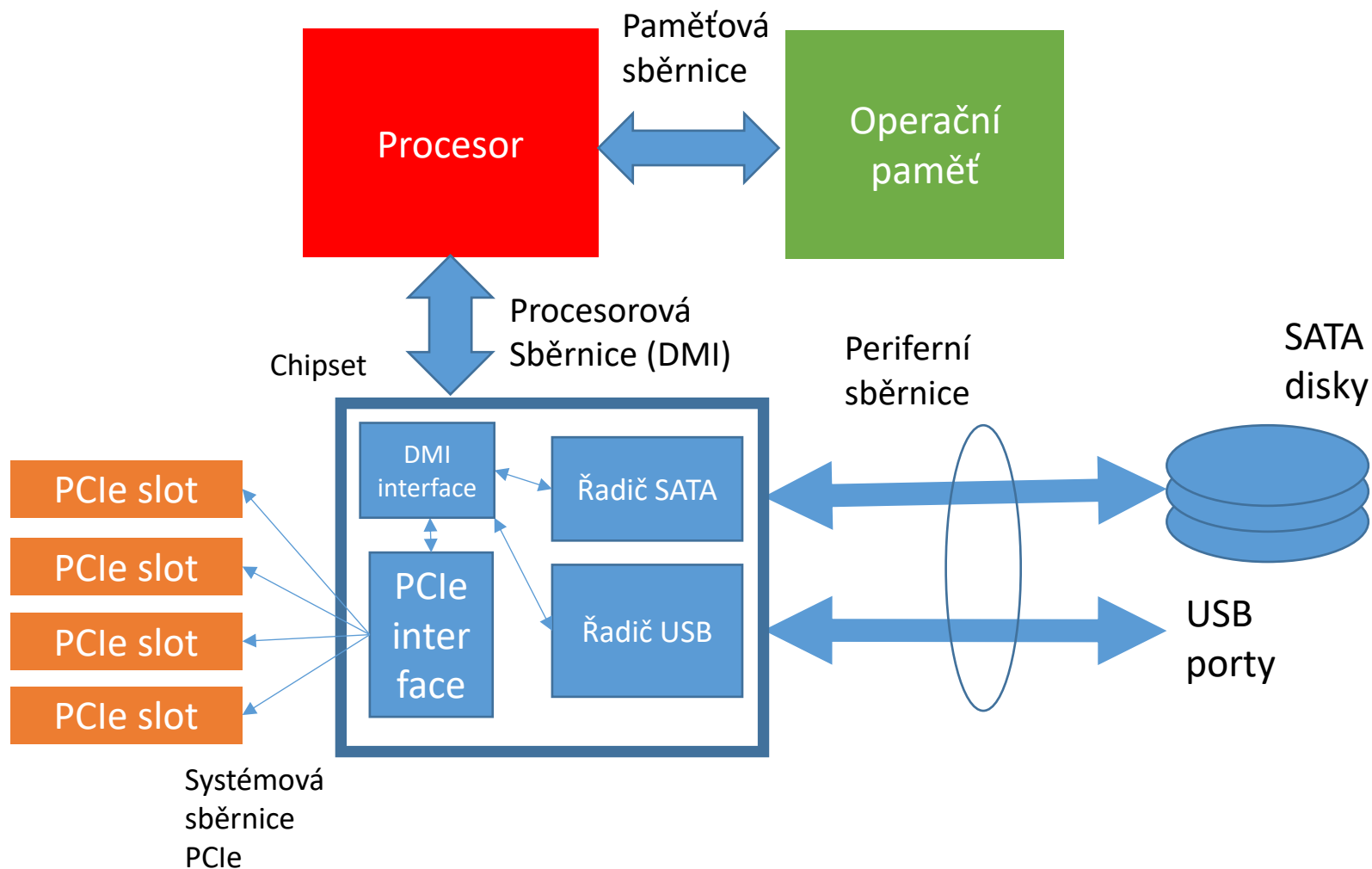


Současné mikrokontroléry, program je typicky uložen v paměti programu typu FLASH

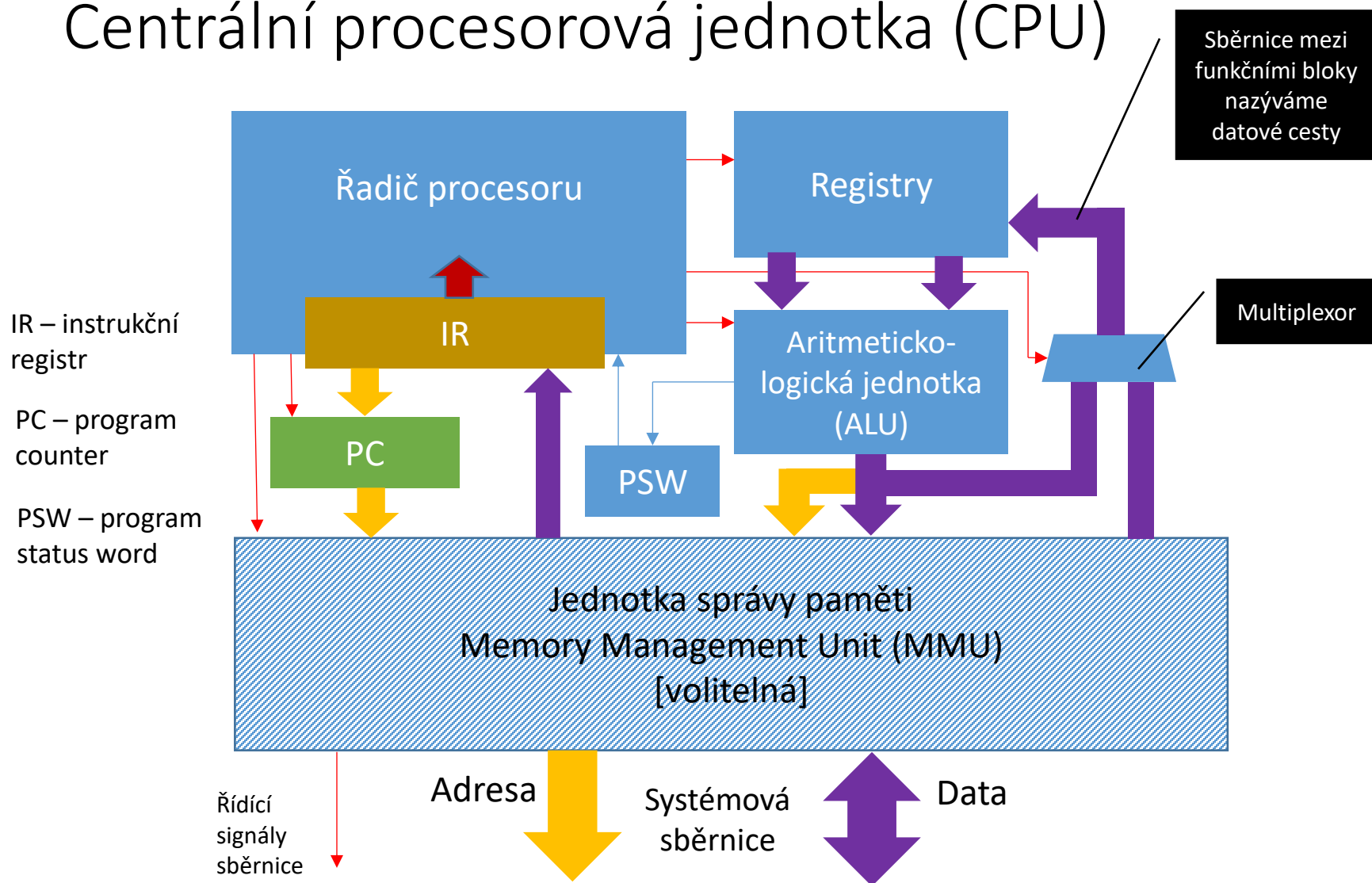
Blokové schéma prvních PC s procesorem 8086



Architektura současného PC



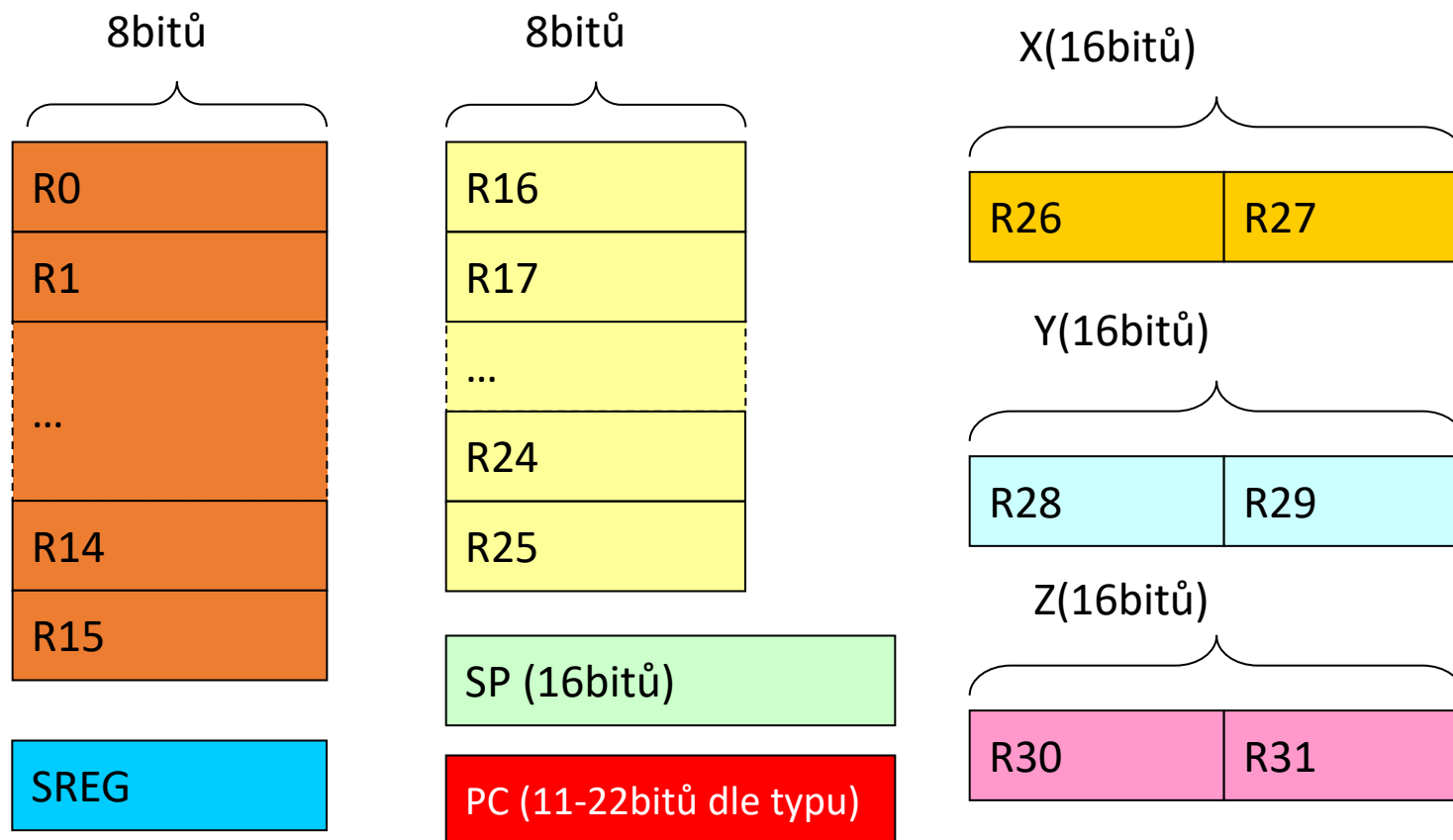
Centrální procesorová jednotka (CPU)



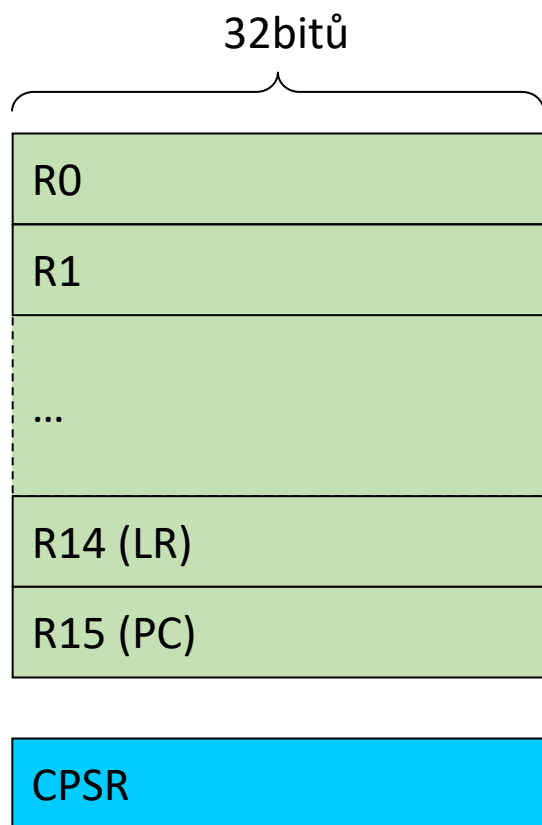
Registry procesoru

- Udržují stav procesoru
- Slouží o odkládání mezivýsledků
- Tvoří operandy aritmetických a logických operací
- Typické registry
 - Program Counter (PC)
 - Adresa instrukce, která je načtena do instrukčního registru
 - Typicky zvětšuje hodnotu o délku právě prováděné instrukce
 - Je měněn instrukcemi pro řízení instrukčního toku (skoky, volání podprogramu)
 - Instrukční registr (IR)
 - Do IR se načítá instrukce, která bude provedena v rámci instrukčního cyklu
 - Programátorsky nepřístupný
 - Stavový registr (PSW – Program Status Word)
 - Ukazatel zásobníku (SP – Stack Pointer)
 - Registry pro všeobecné použití (GPR – General Purpose Registers)
 - Plně programově přístupné
 - Explicitně určené v instrukcích (operandy instrukcí)

Registry AVR (Mikrokontroléry ATMega)



Registry architektury ARM32 (zjednodušeno)

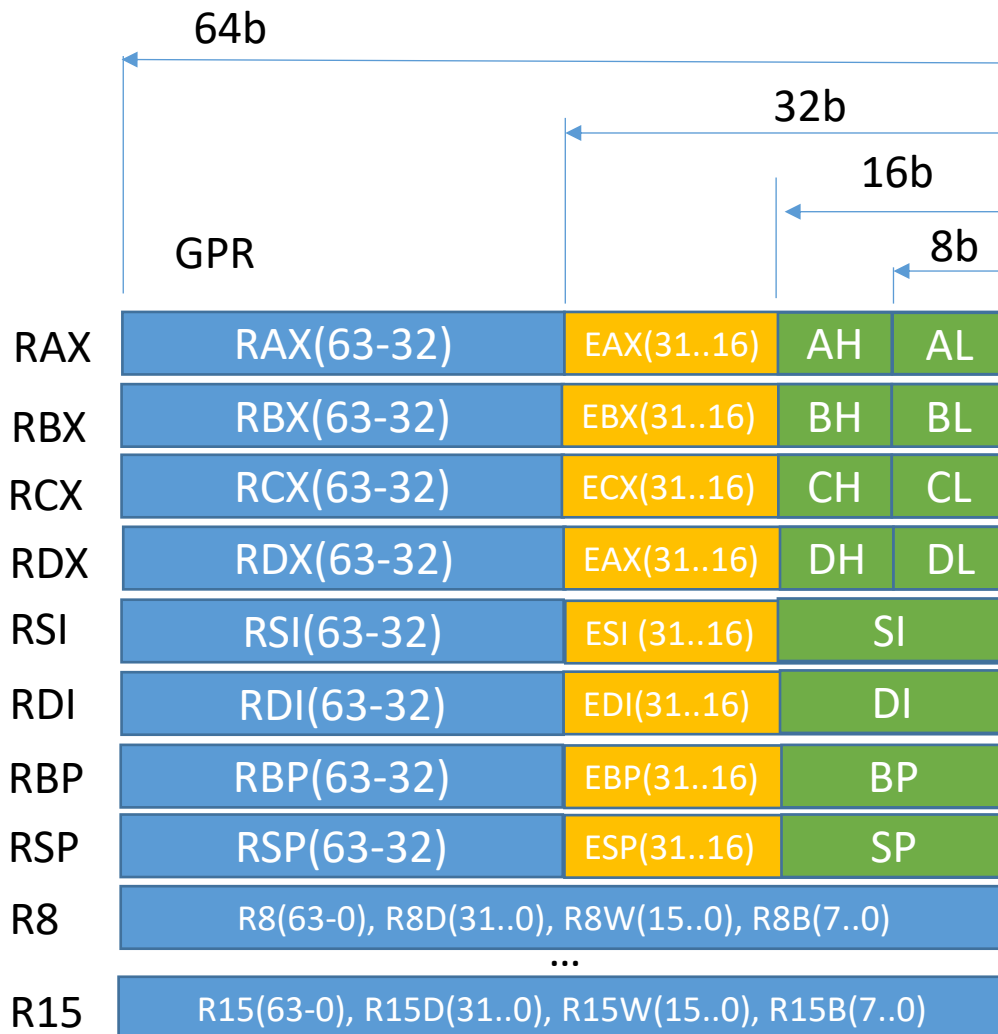


LR – link registr (návratová adresa při skoku do podprogramu)

PC – čítač programu (Program counter)

CPSR – stavový registr (Current Program Status Register)

Registry architektury AMD64 (pouze část)



Program counter

RIP(63..0)

Stavový registr

RFLAGS(63..0)

SSE registry

XMM0 (4x32b)

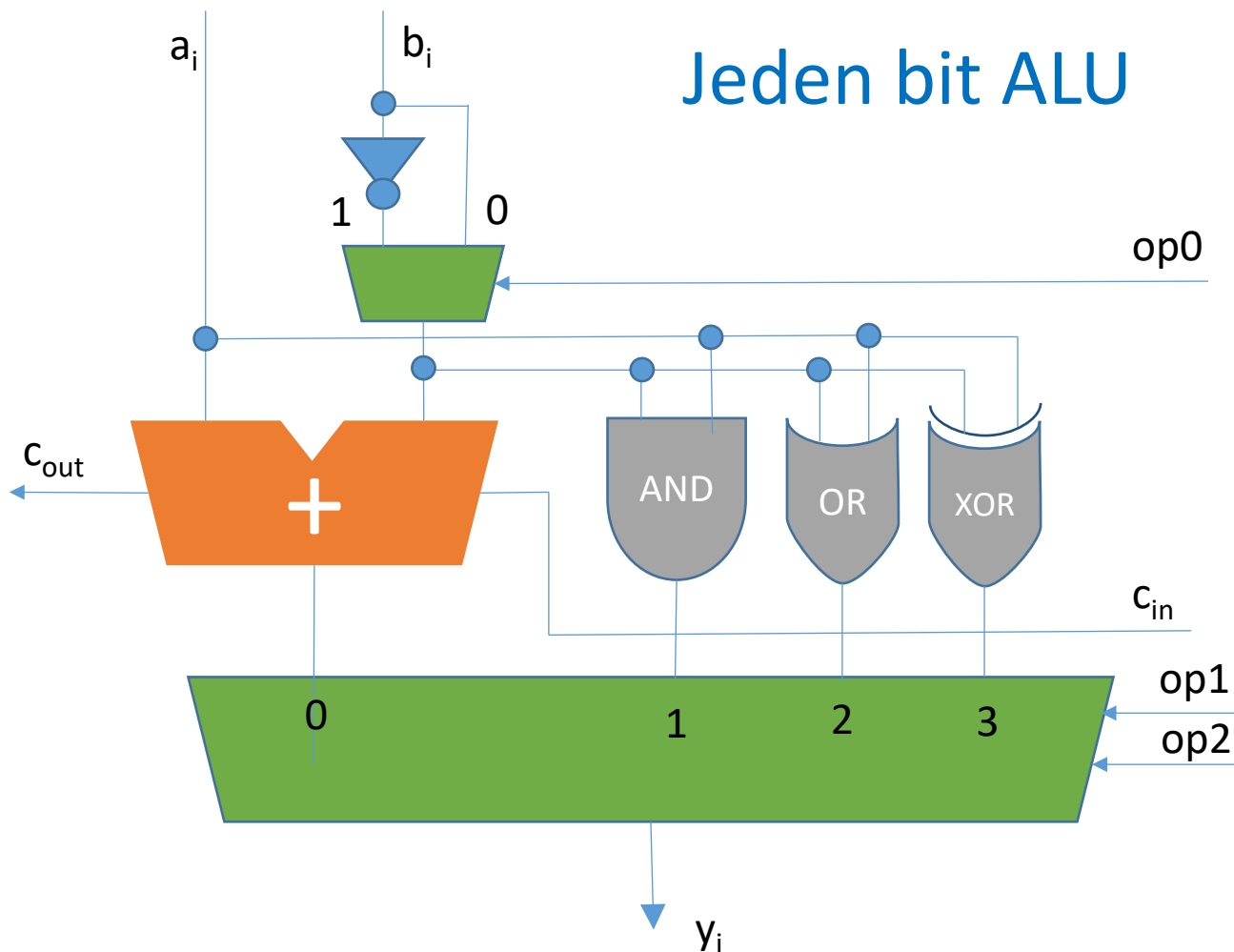
XMM1 (4x32b)

...

XMM7 (4x32b)

Aritmeticko-logická jednotka (celočíselná)

Jeden bit ALU



op2	op1	op0	op
0	0	0	ADD
0	0	1	SUB
0	1	0	AND
1	0	0	OR
1	1	0	XOR

Program status word (PSW)

PSW je registr pro uložení příznaků a stavové informace procesoru.

Typické příznaky

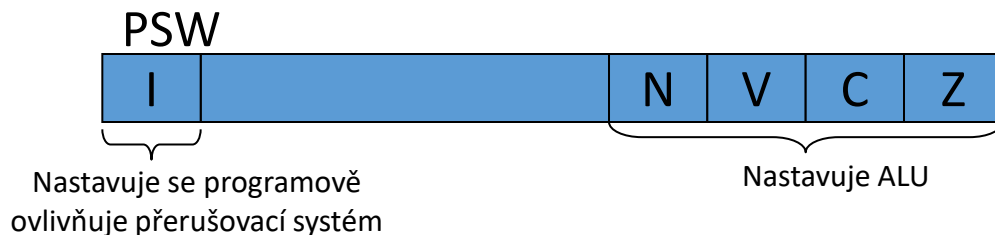
Z (zero) – výsledek operace je nulový

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 $-127-2$).

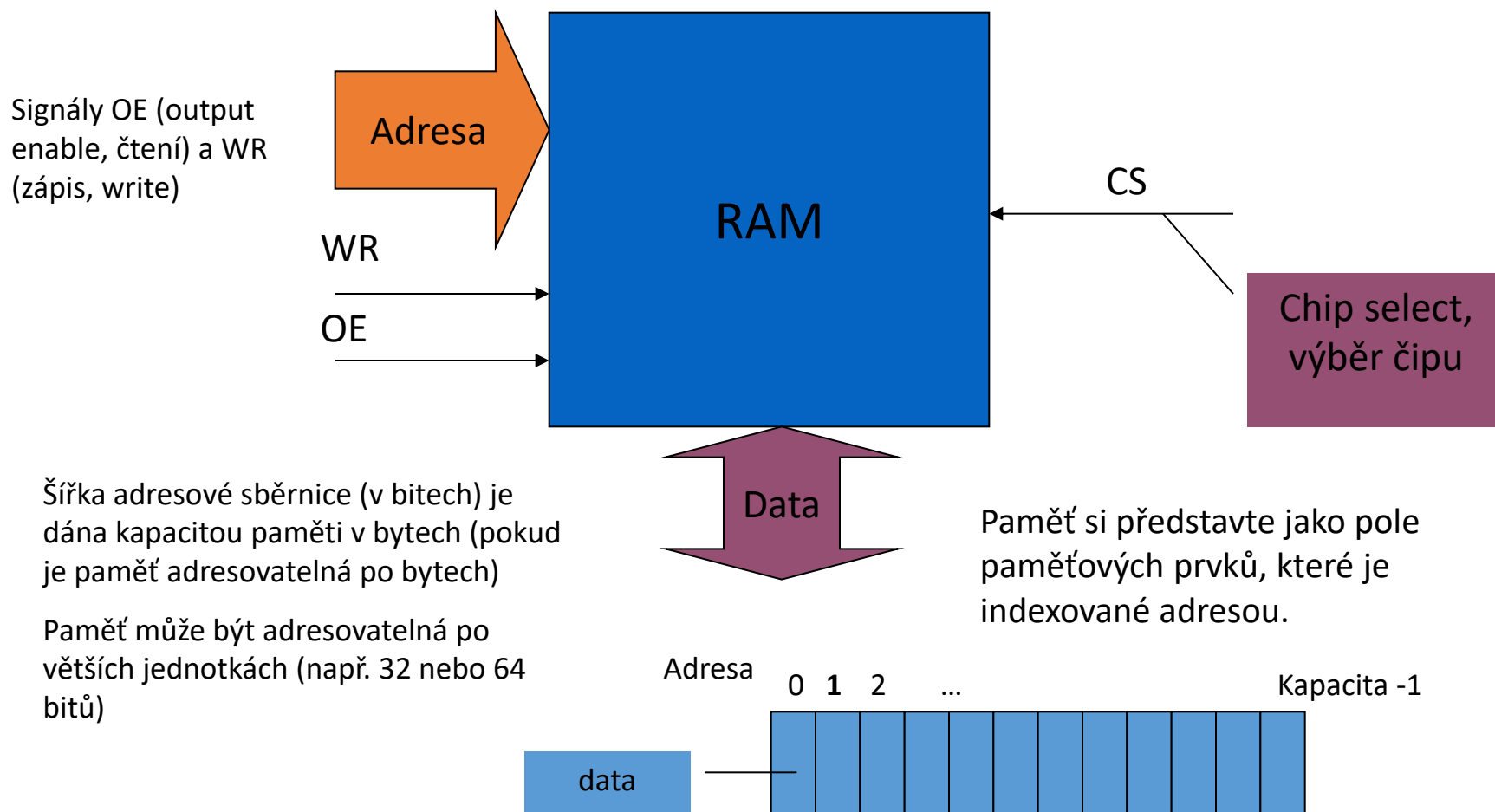
N (negative) – výsledek je záporný

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

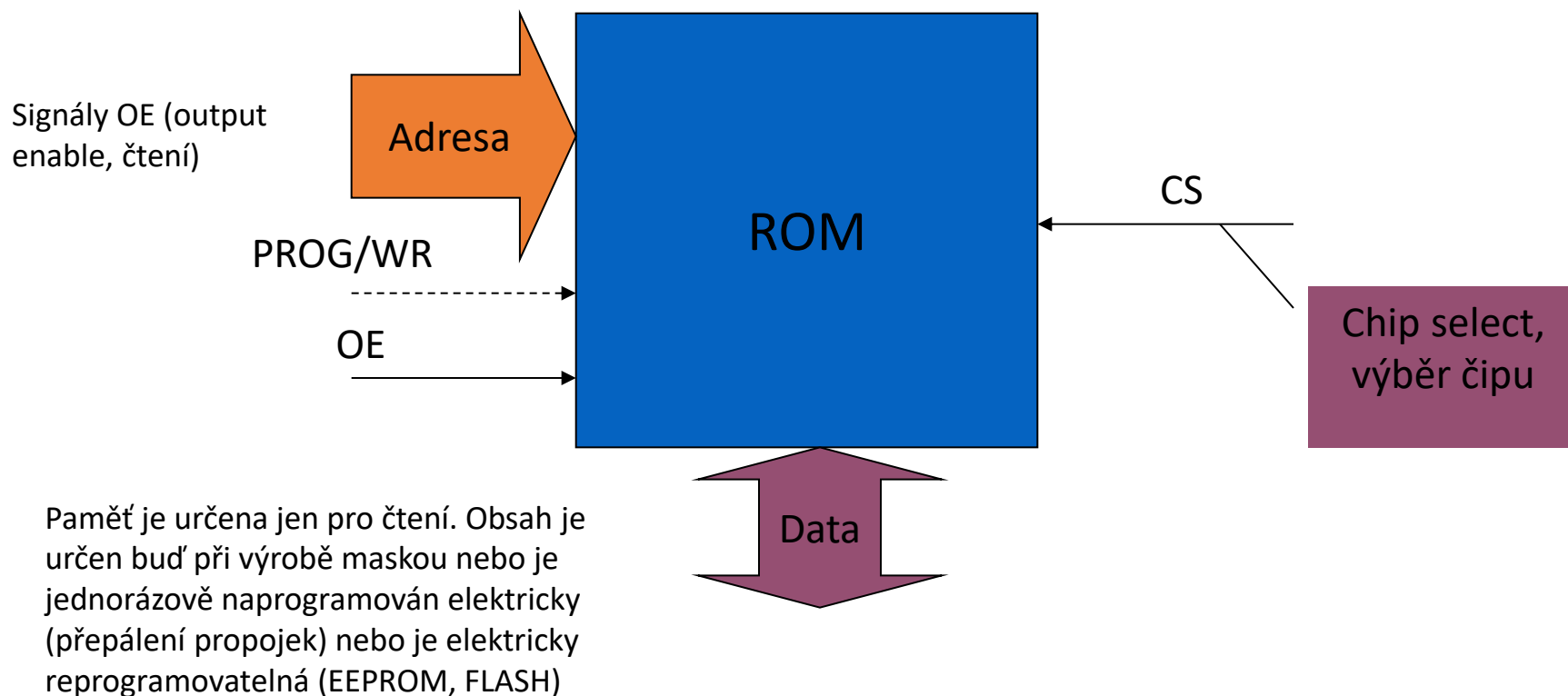


Paměti

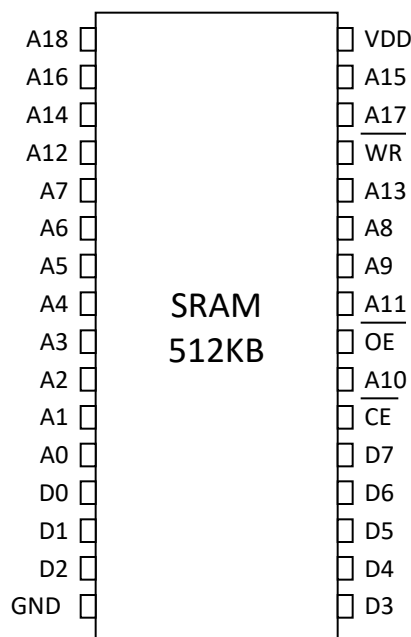
Paměť RAM (Random Access Memory)



Paměť ROM (Read Only Memory)



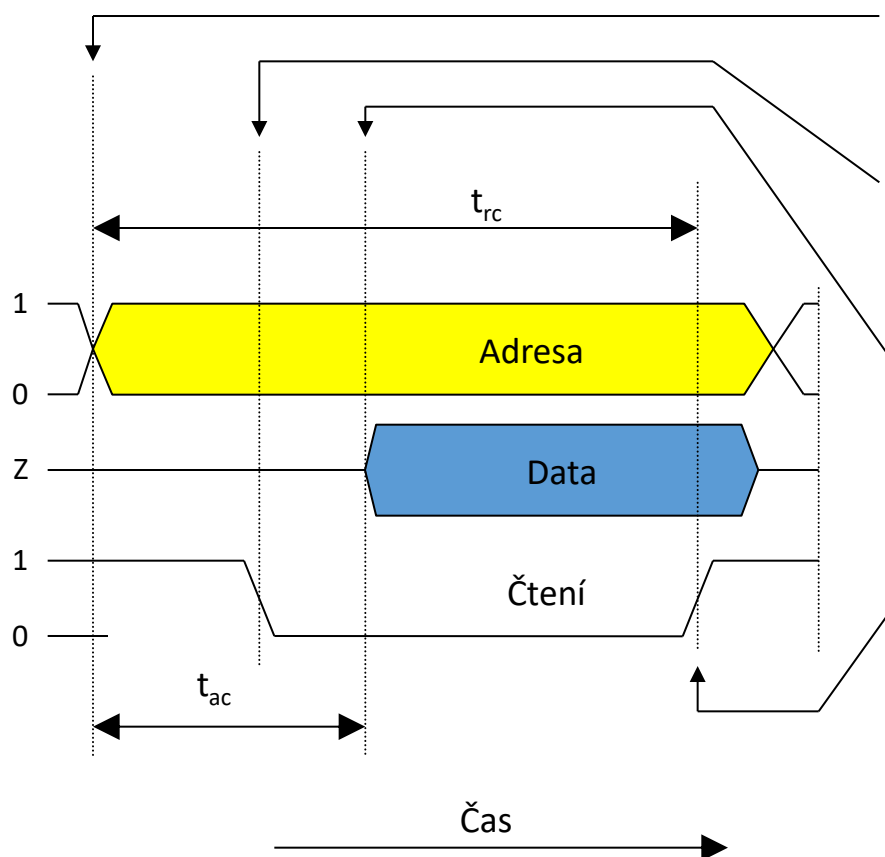
Statická paměť RAM



- 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.

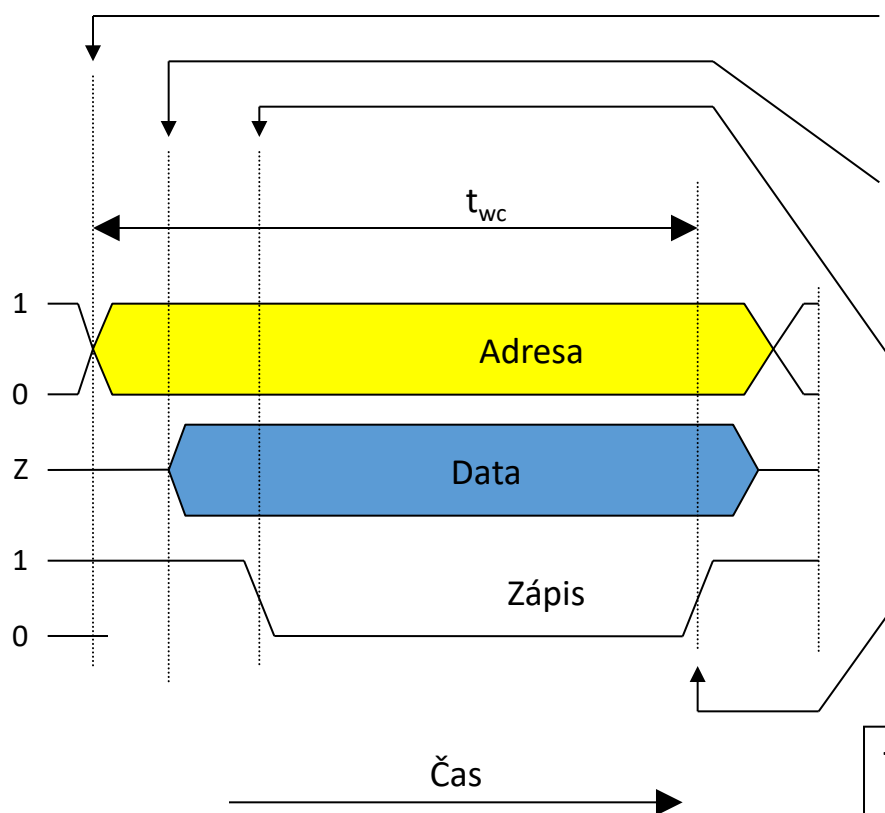
Čtení z paměti



- 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

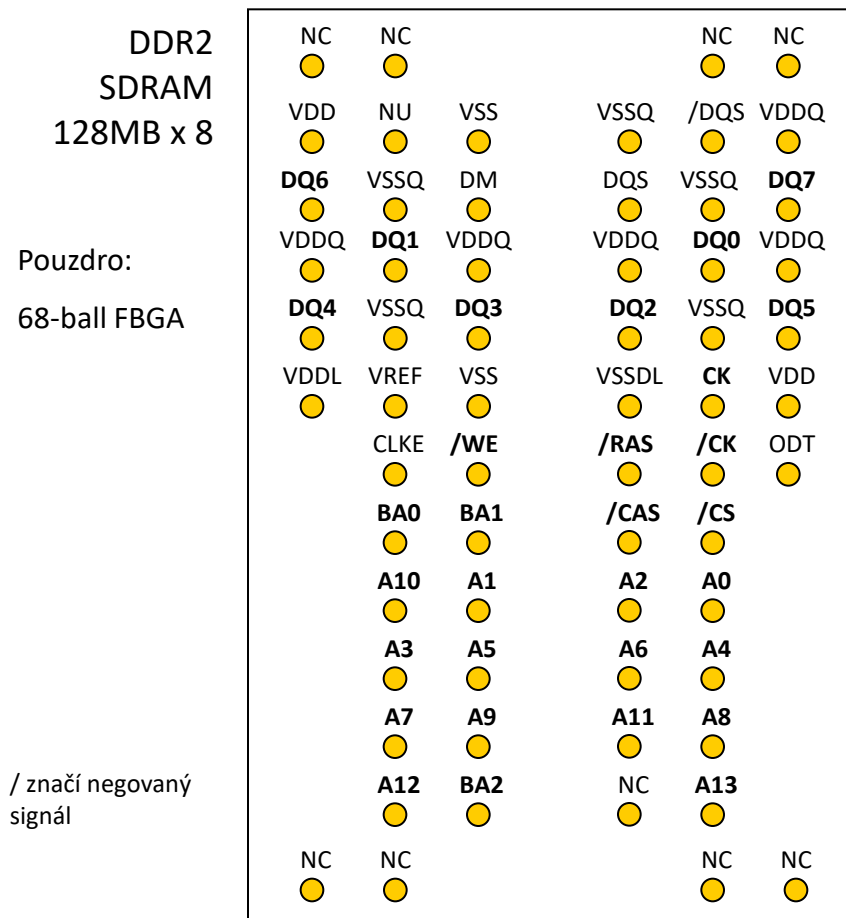


- Vystavení adresy na adresovou sběrnici
- Vystavení dat na datovou sběrnici
- Aktivace zápisového impulsu
- Ukončení zápisového impulsu

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

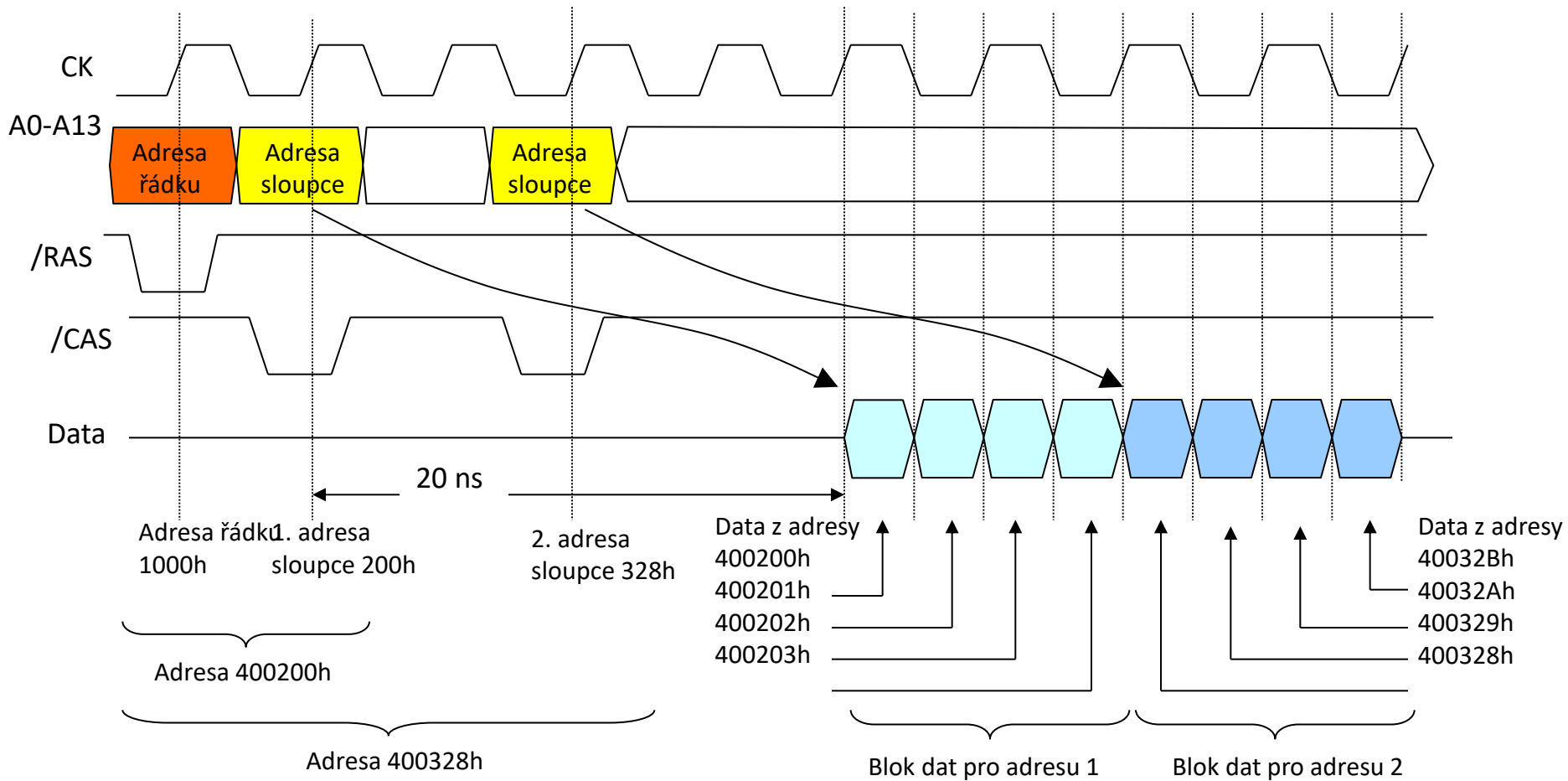
Paměťové obvody pro operační paměť

- dynamická paměť RAM (typ DDR2)



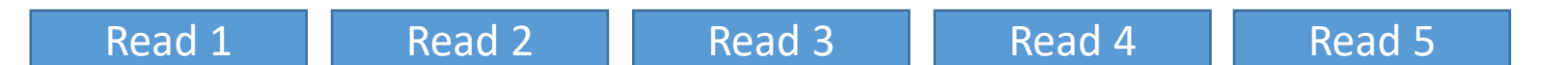
- Synchronní dynamická RAM (SDRAM)
- Periodické obnovování obsahu externím řadičem. Celá paměť musí být obnovena přibližně do 10 ms, jinak hrozí ztráta dat.
- Kapacita 128MB (8 bank x 2^{24} adres)
- Adresa (A0-A13); kompletní 24-bitová adresa se zapisuje nadvakrát sloupcová (10 bitů) a řádková (13 bitů)
- Číslo banky (BA0-BA2)
- Data (DQ0-DQ7)
- Řídící signály:
 - RAS (řádková adresa)
 - CAS (sloupcová adresa)
 - /WE (zápis)
- Hodinový signál (CK a /CK)
- /CS výběr čipu

Čtení z paměti

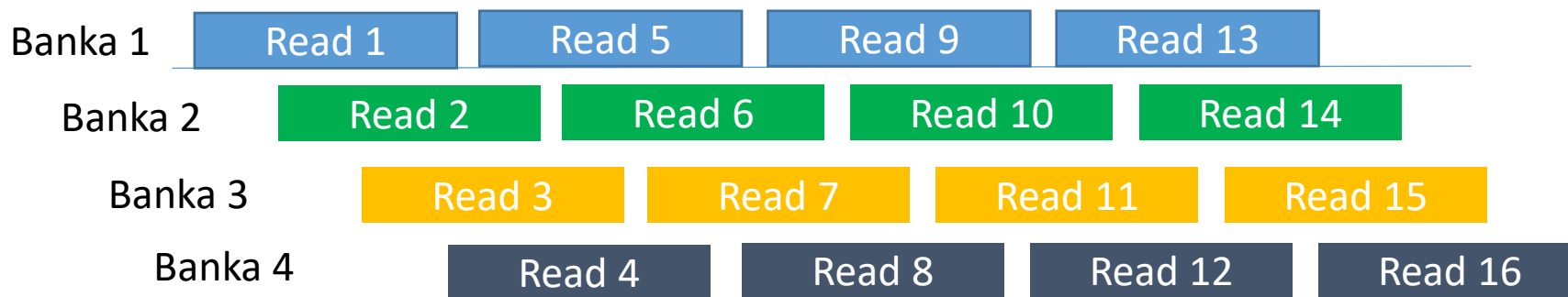


Prokládání paměťových cyklů

Neprokládané paměťové cykly (SRAM)



Prokládané paměťové cykly (DDR3, DDR4)



Prokládaná paměť má svoji kapacitu rozdělenou do stejně velkých částí (bank), které pracují nezávisle (paralelně). Všechny banky sdílejí jedno rozhraní (adresa, data, řízení), takže jednotlivé cykly se musí startovat postupně. Protože přístupová doba do paměti je podstatně větší než odpovídá přenosu dat, lze díky nezávislosti startovat jednotlivé přenosy s překryvem a tak zvýšit množství dat přenesených za jednotku času a vytížit tak datovou sběrnici, která dostatečnou přenosovou rychlost (dnes 3.2G datových přenosů/s)

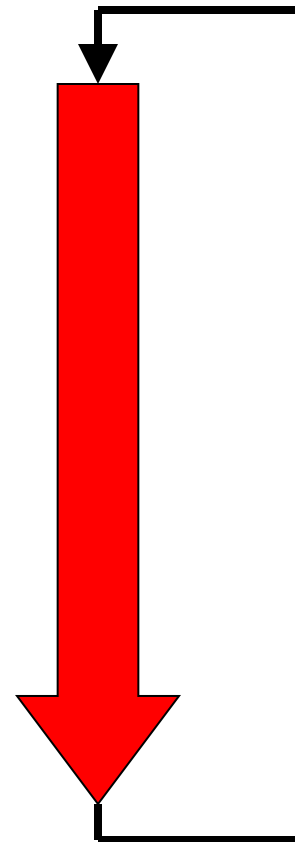
Řadič procesoru a základní instrukční cyklus počítače

Řadič procesoru

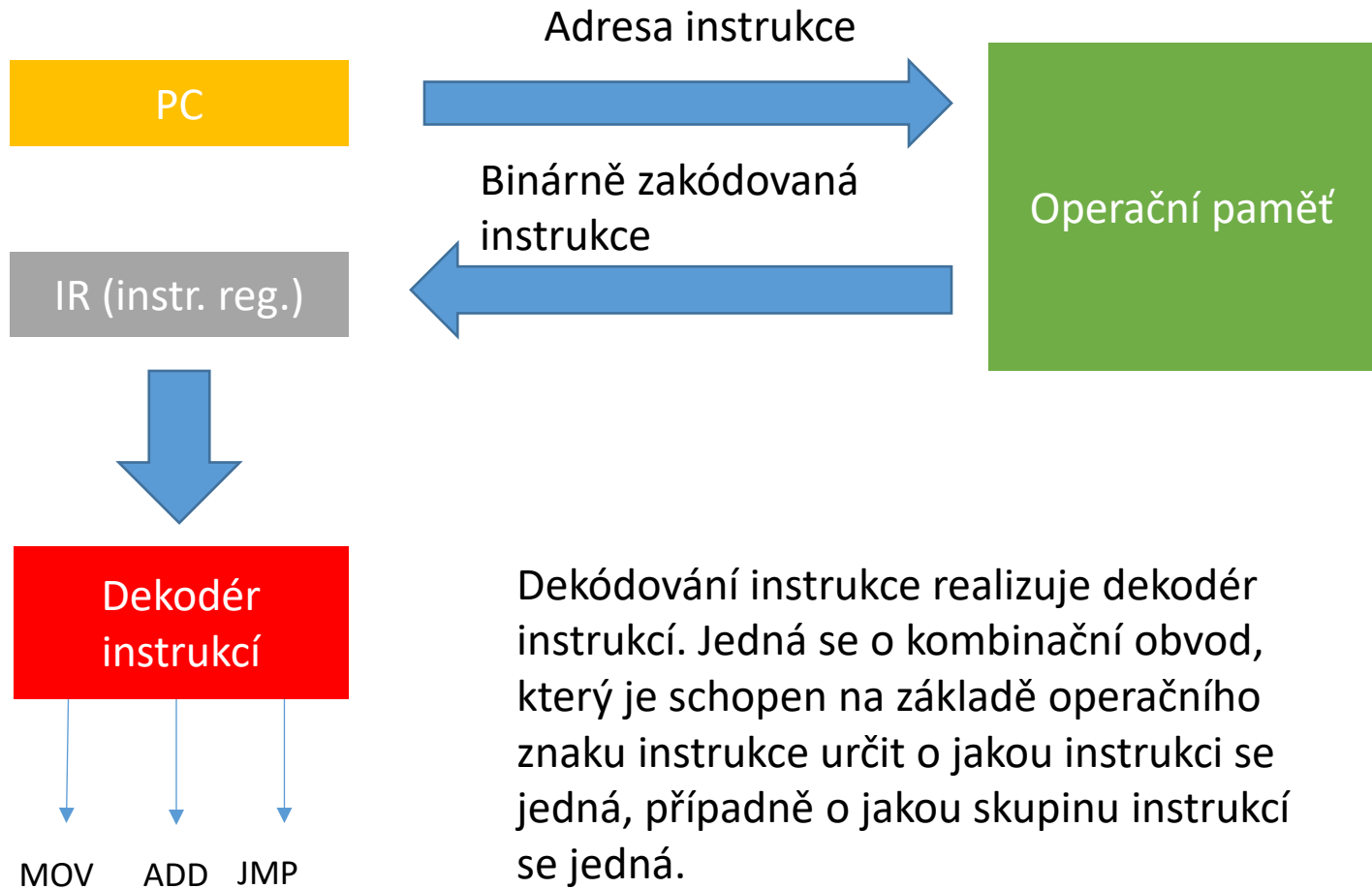
- Realizuje instrukční cyklus
- Řídí vykonávání dílčích operací v rámci instrukčního cyklu
- Generuje řídicí signály
- Reaguje je stavové signály (příznaky aritmetických operací, vstup přerušení, apod.)
- Relizace
 - Obvodový řadič (konečný automat, D-klopné obvody + kombinační logika)
 - Mikroprogramovaný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

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)



Načtení a dekódování instrukce

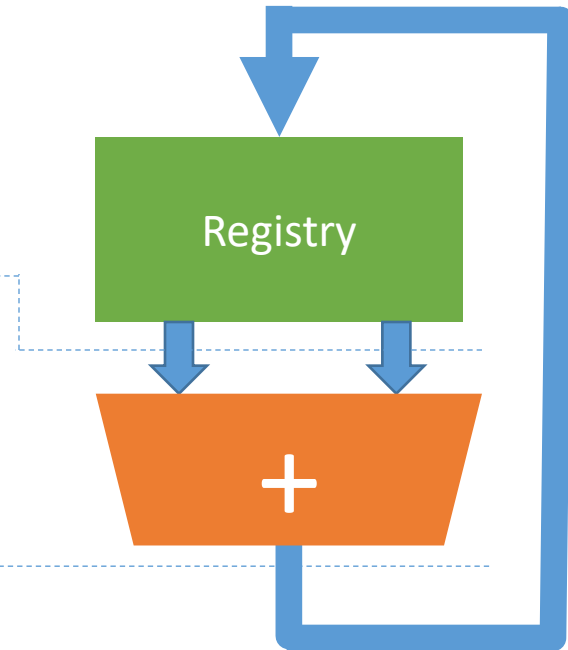


Načtení operandu, vykonání instrukce a zápis výsledku

Načtením operandů rozumíme přesun operandů (např. instrukce součtu ADD) z GPR registrů nebo paměti na vstupy ALU.

Vykonáním instrukce rozumíme provedením všech operací, které určí výsledek instrukce. U instrukce ADD je to výpočet součtu operandů.

Zapsáním výsledku rozumíme zapsání výsledků získaných ve fázi vykonání instrukce do registrů nebo paměti. U instrukce ADD se jedná o zápis výstupu ALU do registrů nebo paměti.



Detekce přerušení

Přerušení je režim, ve kterém procesor vykonává specifikou, předem danou část programu, tzv. obsluhu přerušení (interrupt handler).

Vykonání obsluhy přerušení se provádí na základě vnějšího signálu (typicky se označuje INT), který nazýváme žádostí o přerušení.

K testování signálu žádosti o přerušení dochází na konci instrukčního cyklu. Pokud může být žádosti vyhověno (např. přerušení není maskováno), provede se speciální instrukční cyklus, který odpovídá instrukci volání podprogramu. Pak následují běžné instrukční cykly.

Typy instrukcí

- **Aritmetické:** ADD (součet), SUB(rozdíl), MUL(násobení), DIV(dělení), CMP (CP) 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)

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

Adresní módy

- Registrový MOV R1, R2; MOV rax, rbx
- Přímá konstanta MOV R1, #100; MOV rax, 10
- Přímá adresa LD R1, 100; MOV R1,[100]
- Nepřímá adresace MOV R1, [R2]
 - s postinkrementací LD R1,[R2+]
 - s preinkrementací LD R1,[+R2]
 - s postdekrementací LD R1,[R2-]
 - s predekrementací LD R1,[-R2]
- Nepřímá adresace s bazovým registrem a posunutím LD R1, [R2+100]
- Nepřímá adresace s bazovým registrem, index registrem a posunutím LD R1, [R2+R3+100]
- Nepřímá s bazovým registrem, index registrem, měřítkem a posunutím LD R1, [100+R2+R3*2]; MOV rax,[rbx+rdx*4+offs]
- PC relativní

Přímá konstanta a přímá adresa

Přímá konstanta se používá k naplnění registru (případně paměti) konkrétní hodnotou. Přímá konstanta je součástí instrukce.

```
MOV rax, 10 ; rax ← 10
ADD rax, 10 ; rax ← rax + 10
MOV QWORD PTR a[rip], 10 ; a ← 10
```

Přímá adresa je adresa do paměti, která je součástí instrukce. Například ve spojitosti s instrukcí MOV se používá pro nahrání registru z konkrétního místa paměti.

```
MOV rax, [10] ; rax ← MEM[10]
MOV QWORD PTR [100], 10 ; MEM[100] ← 10
```

Nepřímá adresace

Při nepřímé adresaci se adresa do paměti získává typicky z registru, u některých procesorů existuje i varianta získání adresy z paměti.

```
MOV rax, [rbx]           ; rax ← MEM[rbx]
MOV [rcx], rax           ; MEM[rcx] ← rax
MOV QWORD PTR [rsi], 10  ; MEM[rsi] ← 10
ADD rcx, [rbx]           ; rcx ← rcx + MEM[rbx]
```

Nepřímá adresace s báзовým registrem a posunutím

```
MOV rax, [rbx+10]        ; rax ← MEM[rbx+10]
MOV 23[rcx], rax         ; MEM[rcx+23] ← rax
MOV QWORD PTR 6[rsi], 10 ; MEM[rsi+6] ← 10
```

Nepřímá adresace s báзовým registrem a index registrem

```
MOV rax, [rbx+rsi]       ; rax ← MEM[rbx+rsi]
MOV [rcx][rdi], rax      ; MEM[rcx] ← rax
MOV QWORD PTR [rsi], 10  ; MEM[rsi] ← 10
```


Adresní mód s báзовým registrem, index registrem a měřítkem

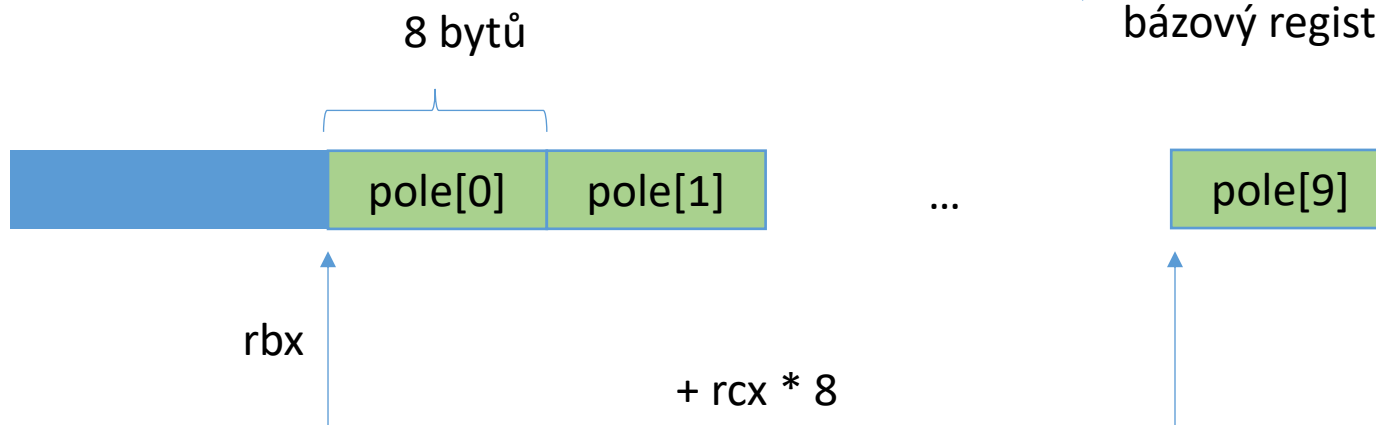
```
long pole[10];
```

```
lea rax, pole[rip]  
mov rcx, 9  
mov rax, [rbx+rcx*8]
```

$rax \leftarrow \text{pole}[9]$

Index registr

bázový registr



Adresní mód s báзовým registrem, index registrem, měřítkem a posunutím

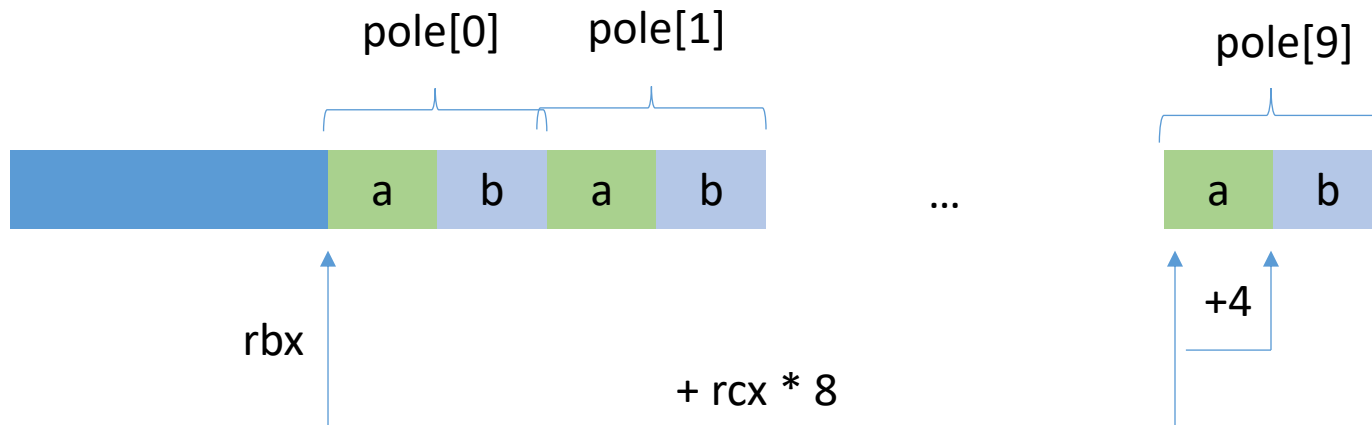
```
typedef struct {  
    int a,  
    int b  
} element_t;  
  
element_t pole[10];
```

```
lea rbx, pole[rip]  
mov rcx, 9  
mov eax, [rbx+rcx*8+4]
```

$\text{eax} \leftarrow \text{pole}[9].\text{b}$

index

bázový registr



Instrukce MOV

MOV zkratka od MOVE přesun. Typicky z registru do registru, registru do paměti, paměti do registru nebo paměti do paměti. Ne všechny možnosti jsou u daného procesoru k dispozici nebo vázány na instrukci MOV.

```
MOV r1, r2      ; r1 <- r2
MOV rbx, rax    ; rbx <- rax
MOV w0, w1      ; w1 <- w0 (assembler některých procesorů
                  ; má cílový operand vpravo)
```

Anglický termín MOVE přesně nevystihuje instrukci MOV. Funkčně přesně odpovídá přiřazení proměnných ve vyšších programovacích jazycích. Například MOV r1, r2 přesně odpovídá přiřazení $r1 = r2$. Zdrojový operand se nemění, cílový se přepíše novou hodnotou.

Pozor ! Neplést s operací MOVE u souborů, kde ve zdrojovém adresáři soubor zmizí a objeví se adresáři cílovém. Tam jde o skutečný přesun typicky adresářové položky.

Aritmetické instrukce

```
ADD r1, r2          ; r1 <- r1 + r2
ADD rbx, rax         ; rbx <- rbx + rax
ADD w0, w1, w2       ; w2 <- w1 + w0   (cílový operand vpravo)
ADDC r1, r2          ; r1 <- r1 + r2 + C
SUB r1, r2           ; r1 <- r1 - r2
SUBB r1, r2          ; r1 <- r1 - r2 - C
CP r1, r2            ; r1 - r2, zapisuje pouze příznaky
CPC r1, r2           ; r1 - r2 - C, zapisuje pouze příznaky
INC r1               ; r1 <- r1 + 1
DEC rbx              ; rbx <- rbx - 1
INC w0, w1           ;
IMUL rax, rbx        ; rax <- rax * rbx
IDIV rax, rbx        ; rax <- rax / rbx
```

Aritmetické instrukce nastavují příznaky ve stavovém registru. Typicky nastavují

- C – carry (přenos nejvyššího řádu řádové mřížky)
- Z – zero (nulový výsledek)
- OV – overflow (přetečení v doplňkovém kódu)
- N – negative (záporný výsledek)

Logické instrukce

Logické operace

```
AND rax, rbx      ; rax <- rax and rbx
OR  rax, rbx      ; rax <- rax or  rbx
XOR rax, rbx      ; rax <- rax xor rbx
NOT rax           ; rax = ~ rax (bitová negace)
```

Posuvy

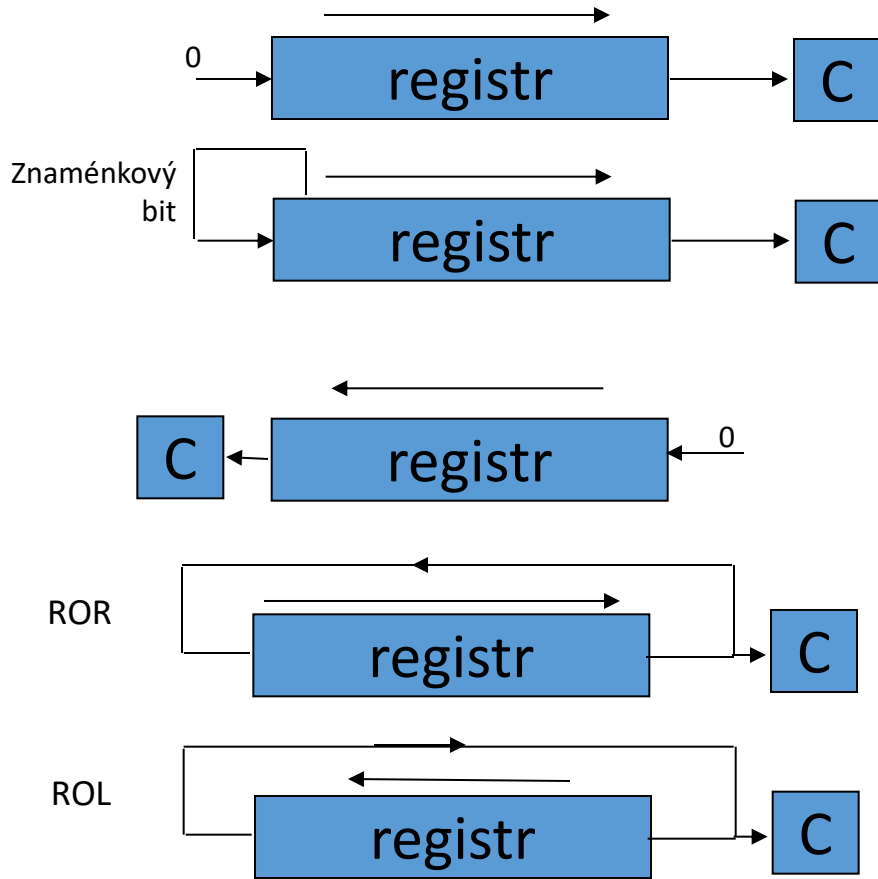
```
SHL rax, 1        ; logický posuv vlevo
SHR rax, 1        ; logický posuv vpravo
SAR rax, 1        ; aritmetický posuv vpravo
```

Rotace

```
ROL rax, 1        ; rotace vlevo
ROR rax, 1        ; rotace vpravo
RCL rax, 1        ; rotace s carry vlevo
RCR rax, 1        ; rotace s carry vpravo
```

Místo jedničky může být CL registr. Pak se jedná o vícebitový posuv nebo rotaci

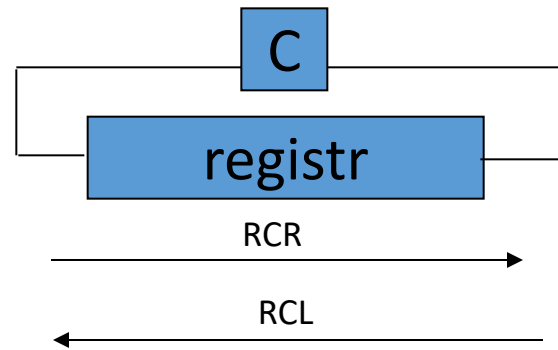
Posuvy a rotace



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



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 add, 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
 $PC = \text{<cílová adresa skoku>}$, 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 $PC = PC + \text{<posunutí>}$
- 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 se také ukládají parametry a lokální proměnné funkcí, odkládají se zde 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

fu_add16:

; podprogram

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

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

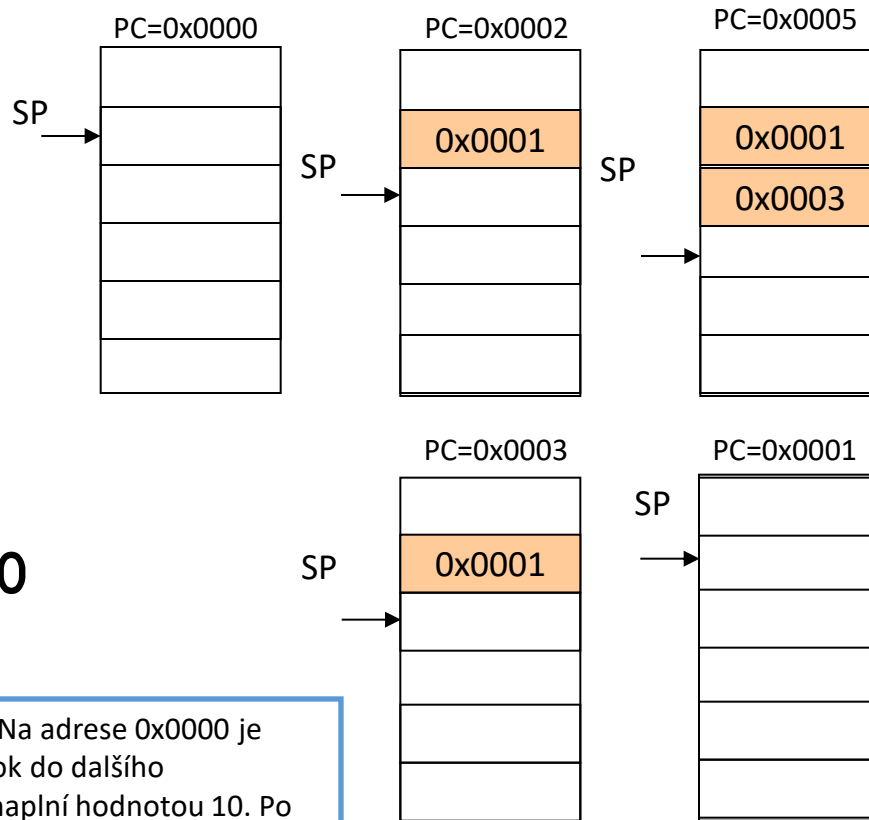
Výsledek r17:r16
0x0300

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
```

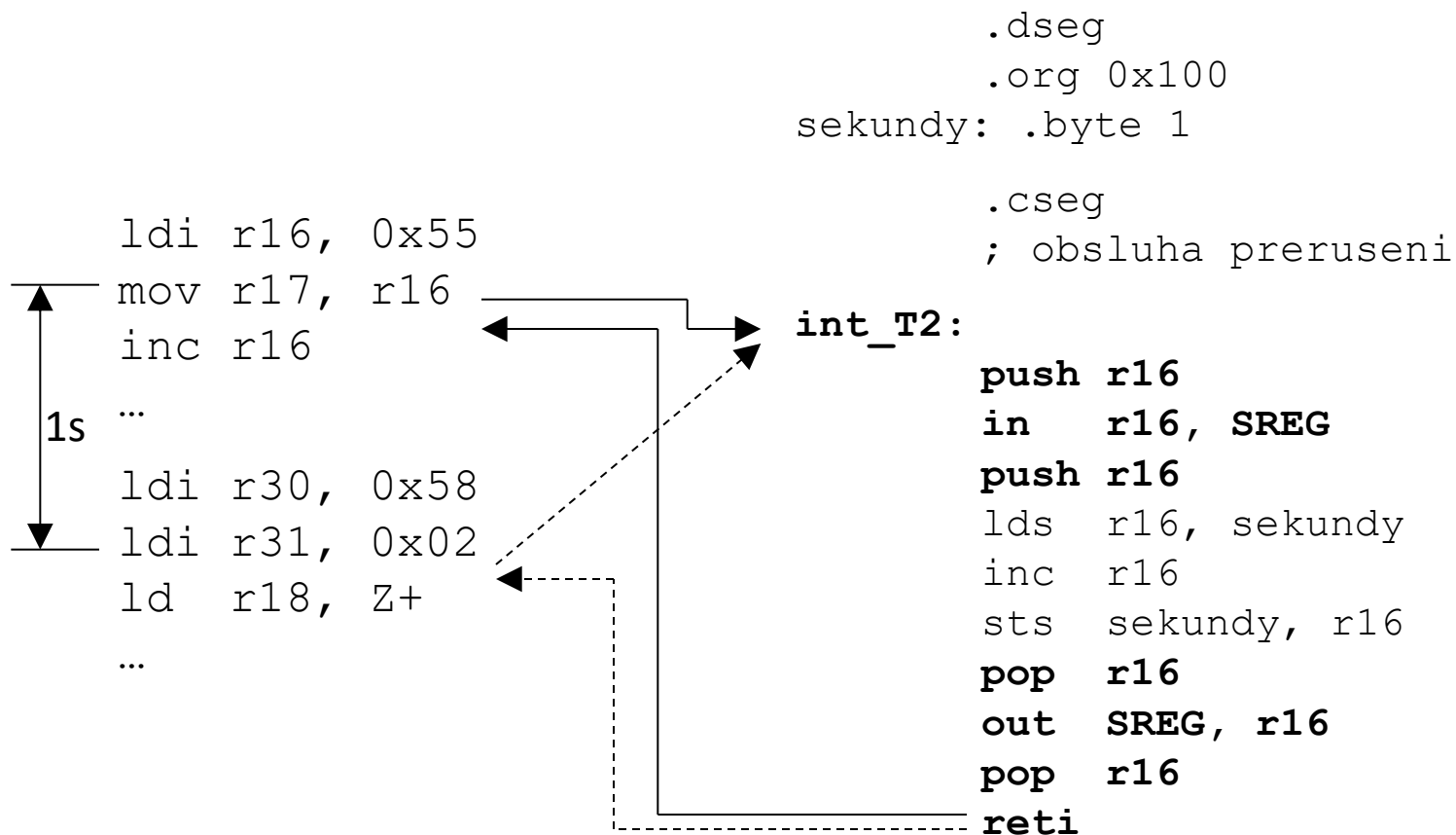


Po resetu se program začne provádět od adresy 0x0000. Na adrese 0x0000 je skok do podprogramu na adresu 0x0002, kde nastane skok do dalšího (vnořeného) podprogramu na adrese 0x0005. Zde se r1 naplní hodnotou 10. Po návratu z vnořeného podprogramu (RET na 0x0006), pokračuje program na adrese 00003h, kde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

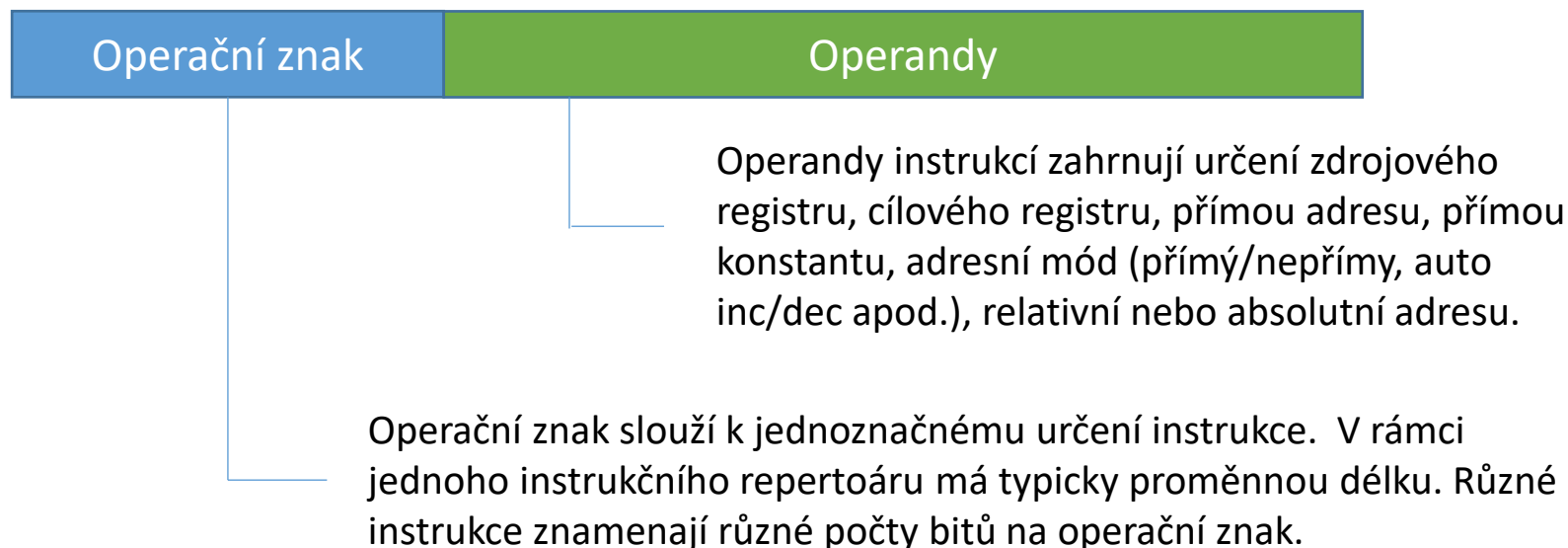
Vstupně/výstupní operace

- 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, ARM)
- **Registry periférií se nesmí podléhat kešování.**
- U vstupně/výstupních registrů může mít význam
 - Zapsání hodnoty
 - Zápis 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
- Vstupně výstupní instrukce IN a OUT. Například: IN DX, EAX; OUT EAX, DX

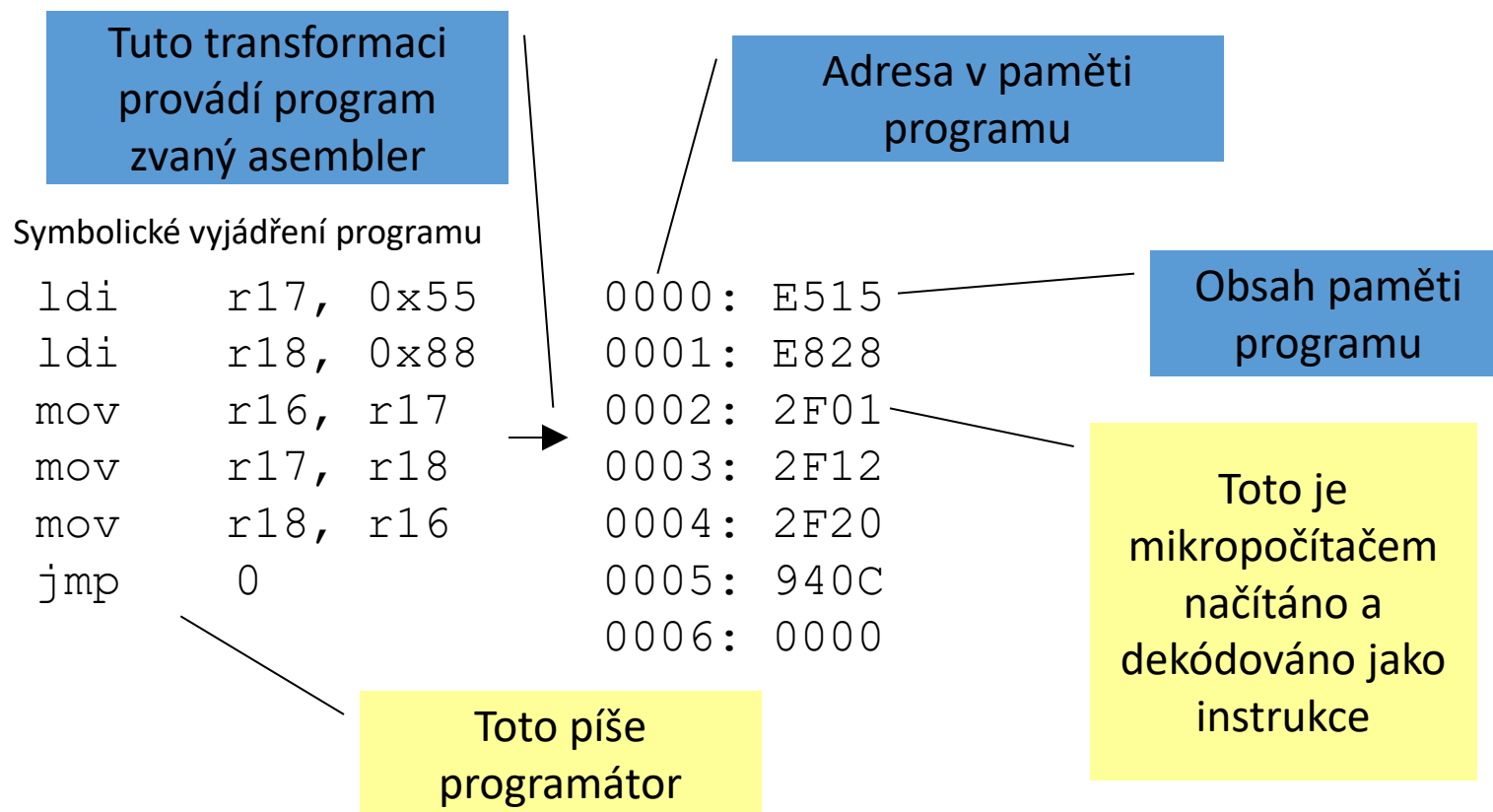
Instrukce ve strojovém kódu

Instrukce s pevnou délkou (např. 32 bitů), instrukční repertoár může mít delší instrukce (např. 64 bitů), ale ty jsou minoritně zastoupeny.

Instrukce s proměnnou délkou. Délka se mění například od jednoho do několika bytů.



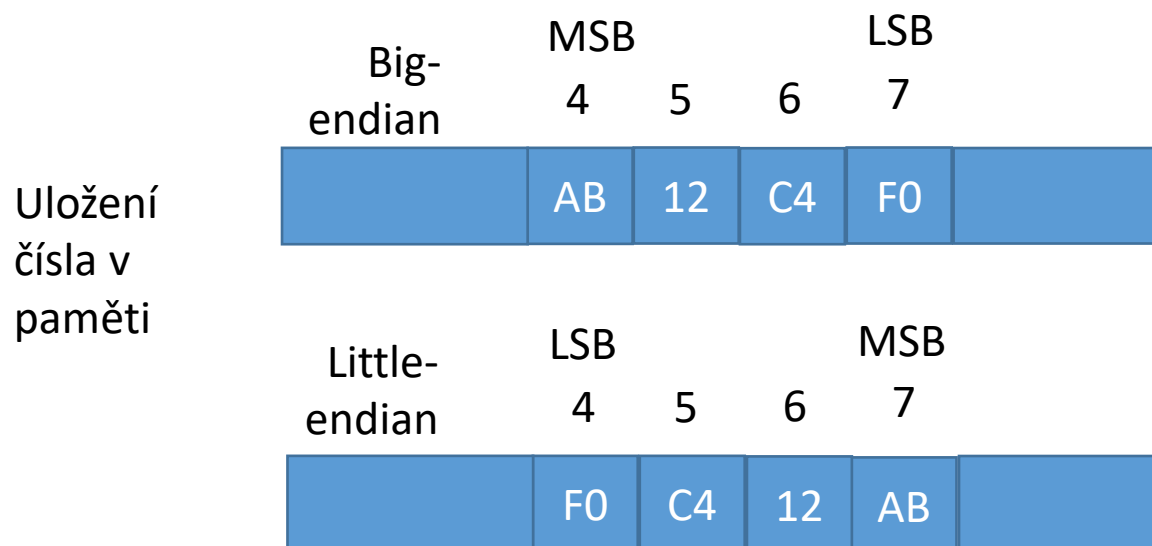
Příklad programu v symbolickém vyjádření a ve strojovém kódu



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

Endianness – Little-endian a Big-endian

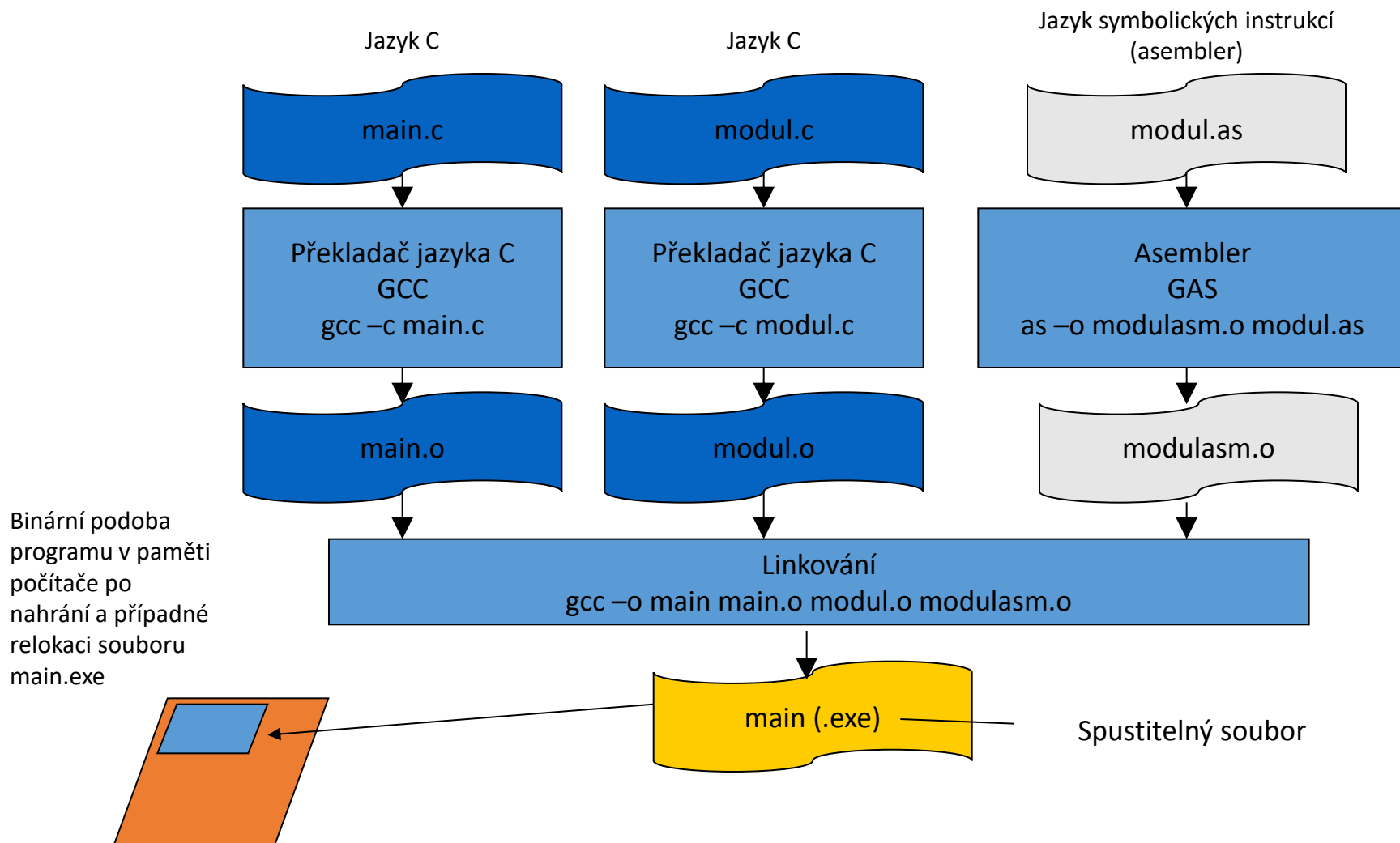
Číslo: 0xAB12C4F0



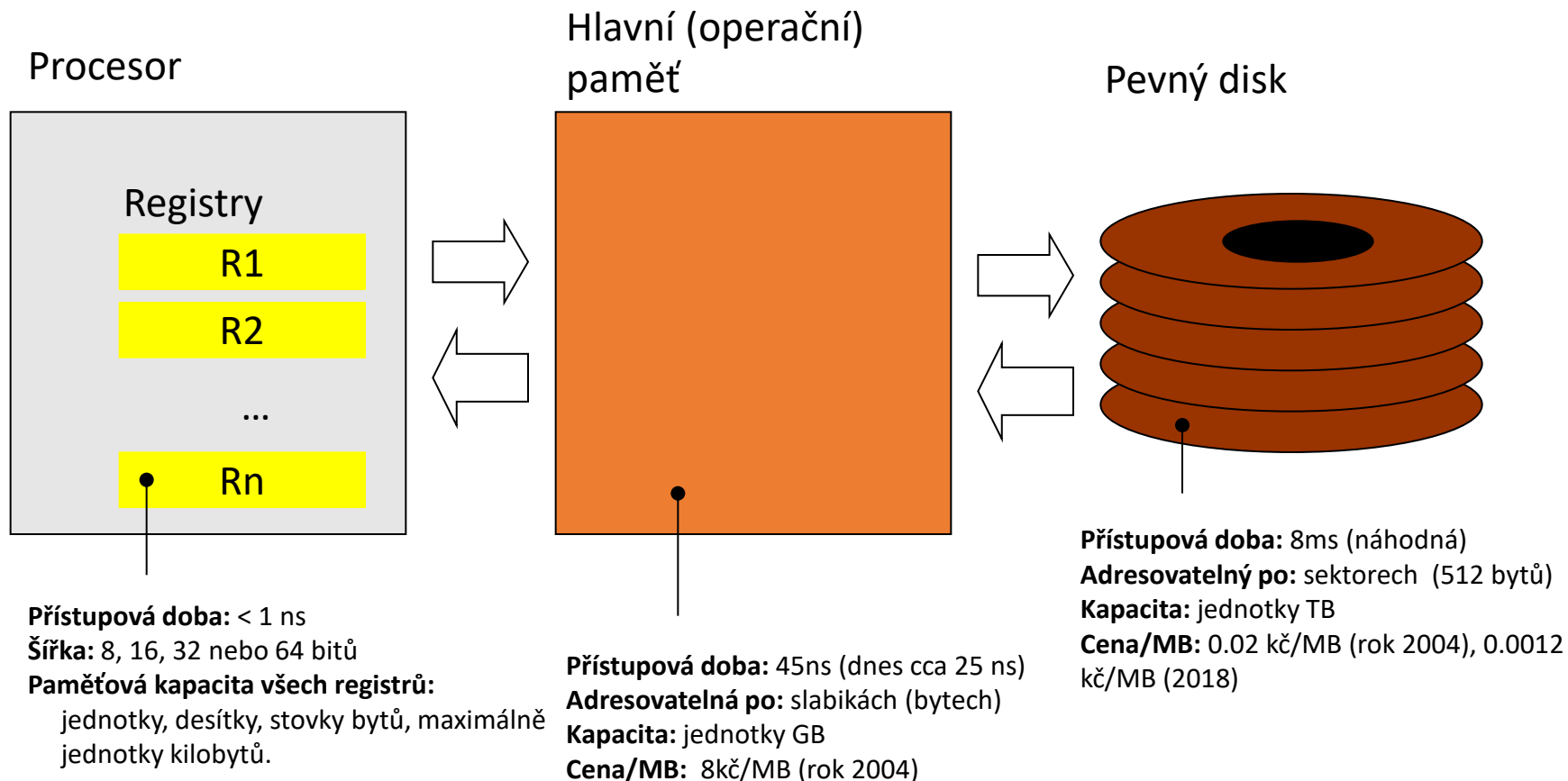
MSB – most significant bit(s) nejvýznamnější bit(y)

LSB - least significant bit(s) nejméně významné bit(y)

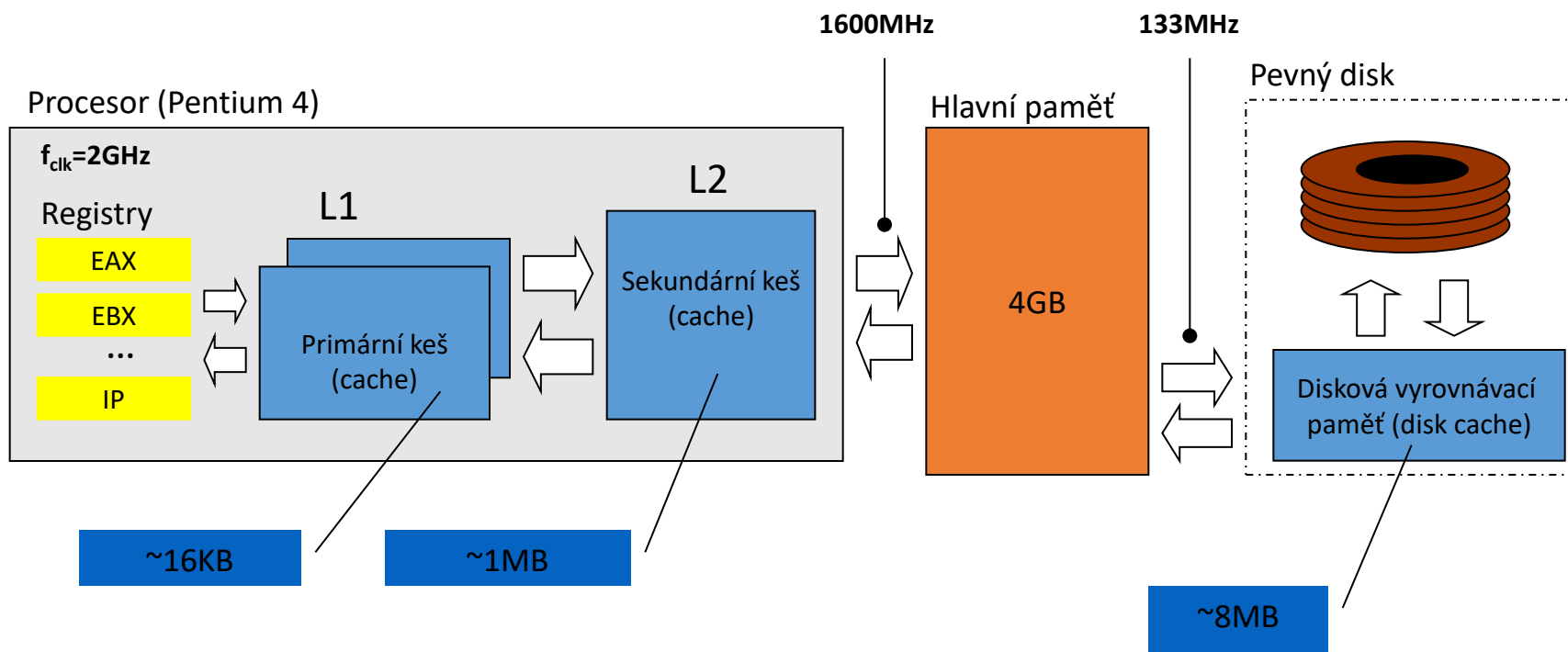
Překlad z vyšších programovacích jazyků



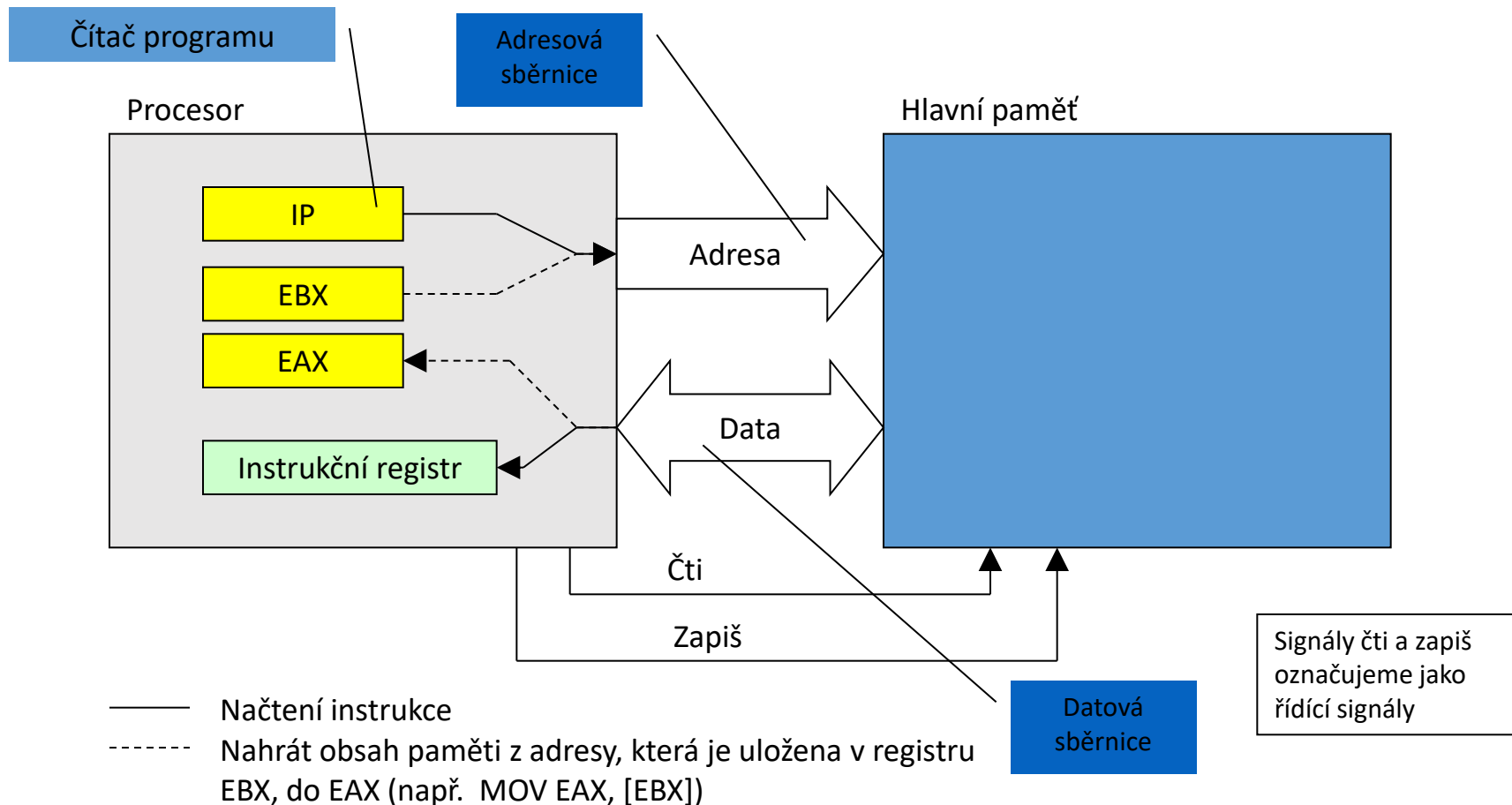
Hierarchie paměťového pod systému



Příklad: paměťový podsystém počítače PC



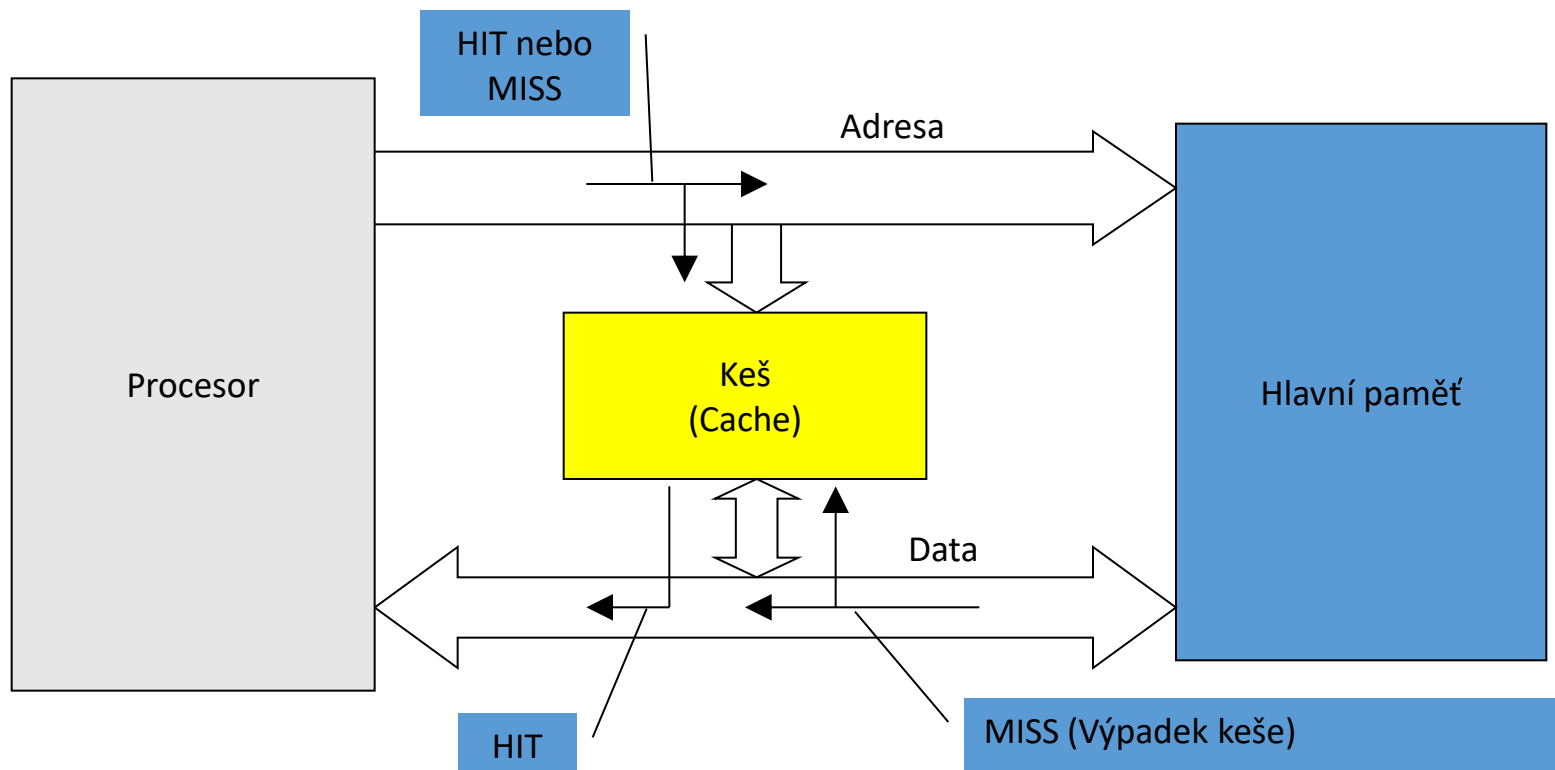
Výměna dat mezi procesorem a hlavní pamětí



Keš (Cache, skrytá paměť)

- 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 témže datům (pokud nebyla vřazena ze skryté paměti) je dán přístupovou dobou do skryté paměti
- Keš je transparentní (z funkčního hlediska procesor nepozná, jestli je přítomna nebo ne)

Funkce keše



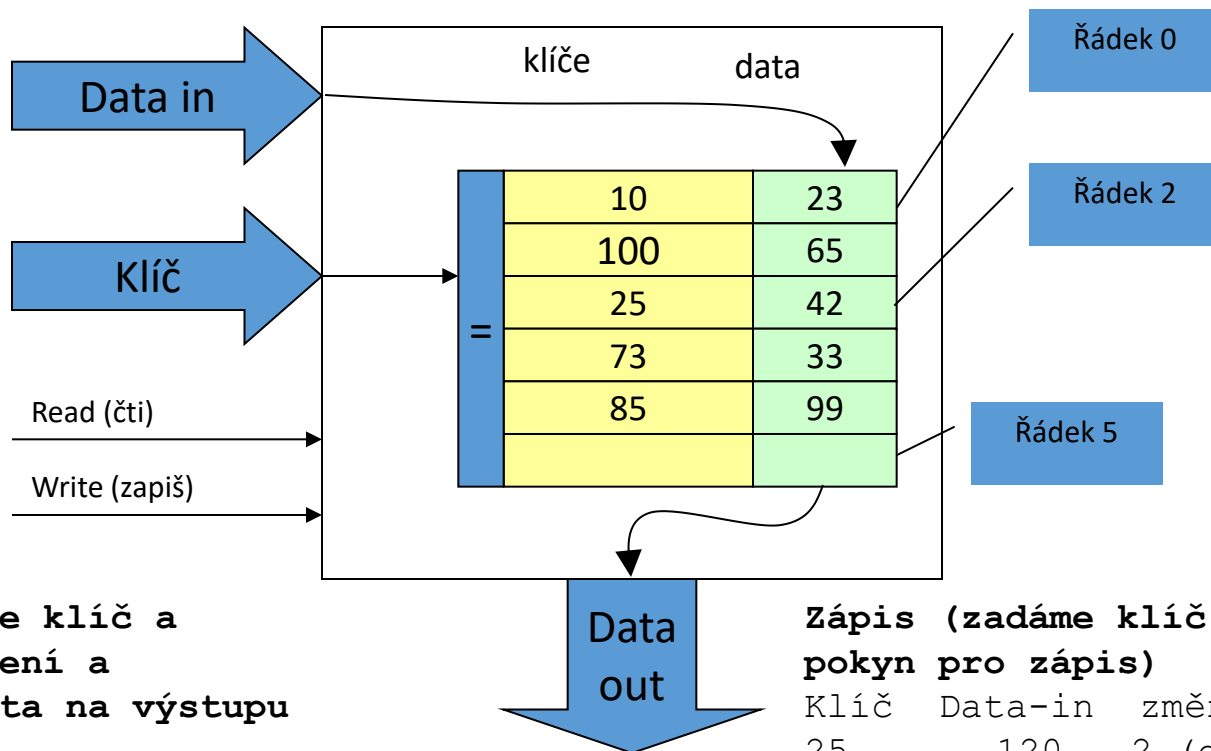
HIT – data na adrese generované procesorem jsou v keši (kratší přístupová doba)

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

Konstrukce keší (caches)

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

Asociativní paměť - princip



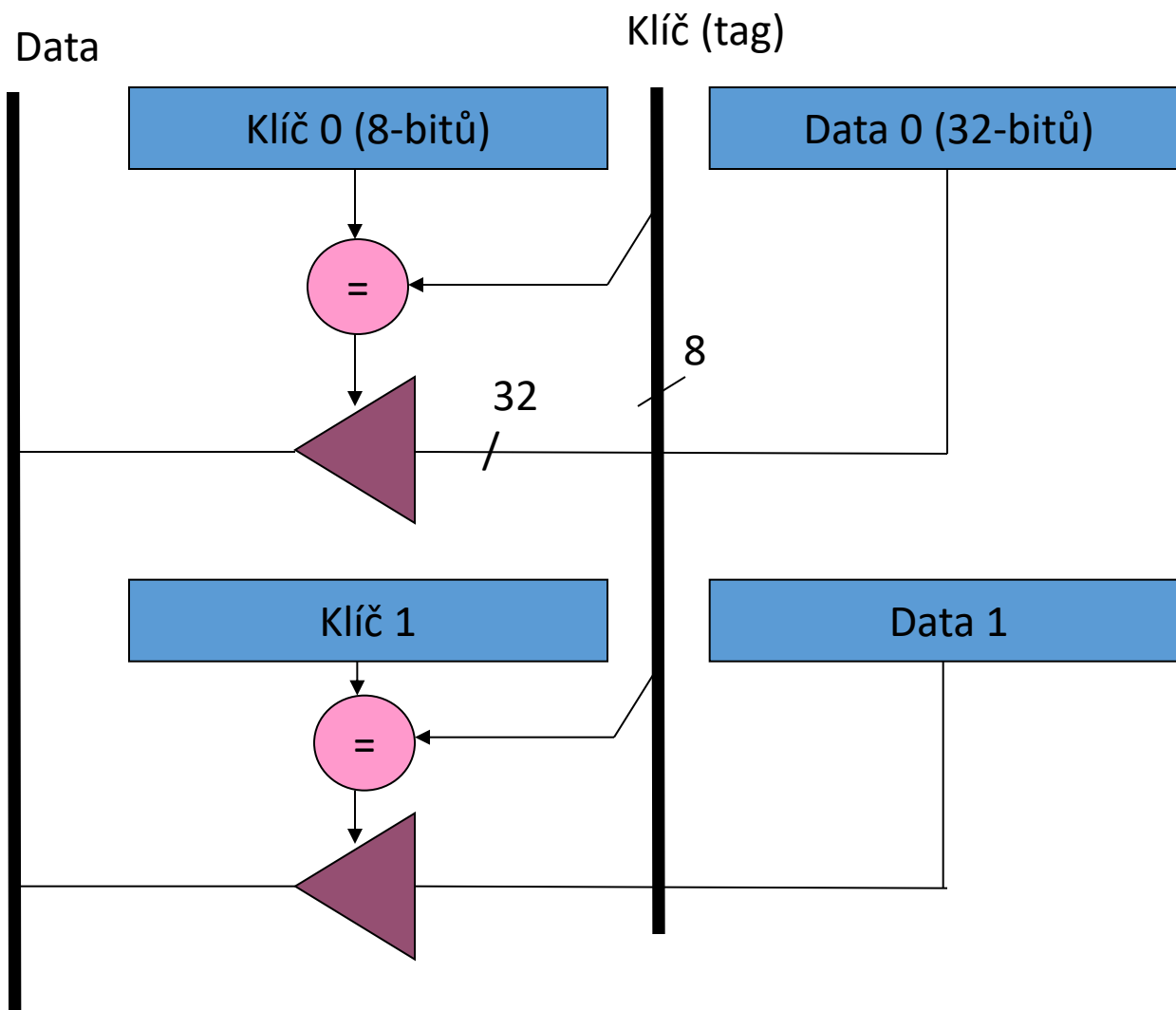
Čtení (zadáme klíč a pokyn pro čtení a očekáváme data na výstupu Data-out)

Klíč	Data-out
25	42
85	99

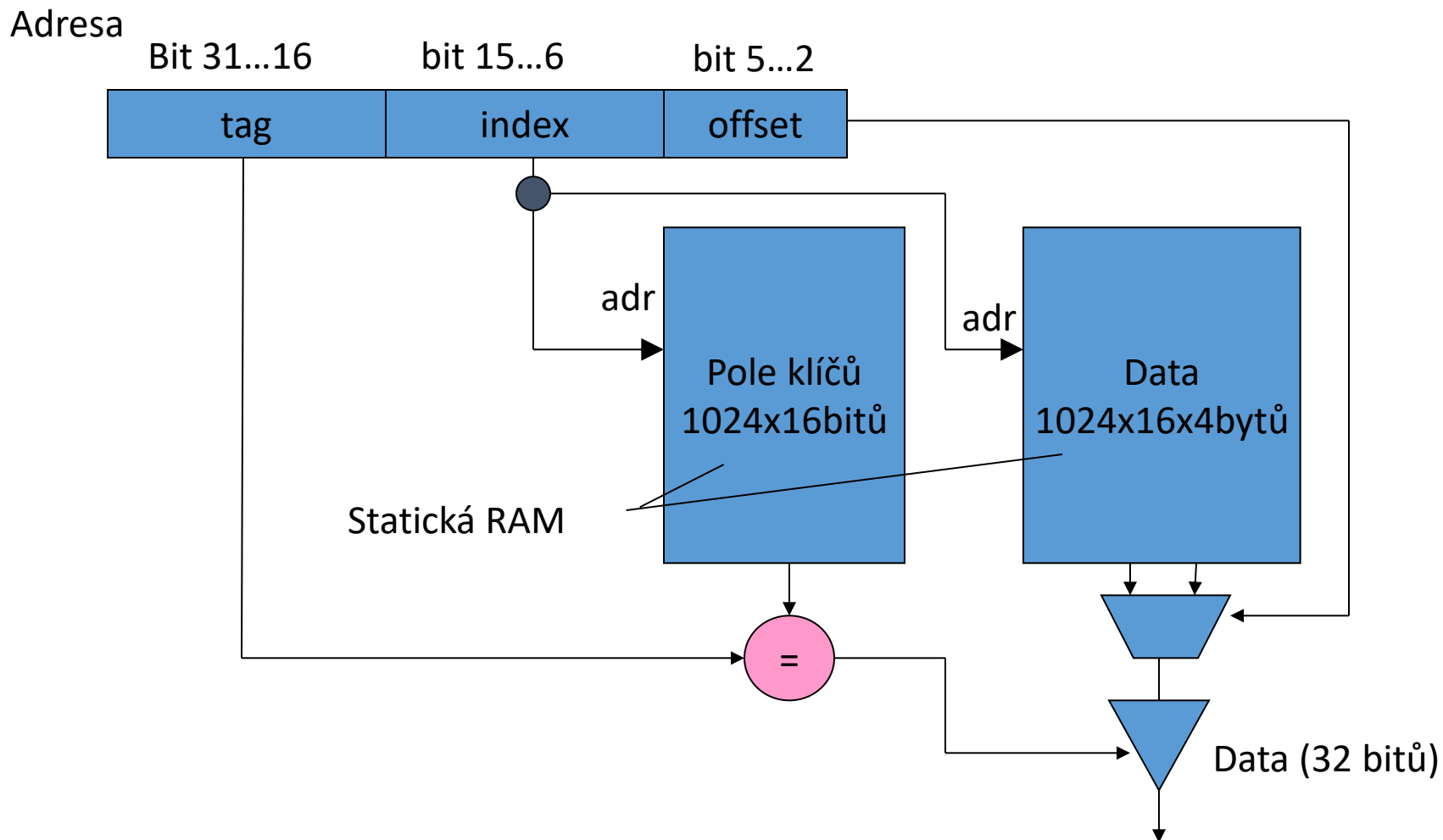
Zápis (zadáme klíč, data-in a pokyn pro zápis)

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)

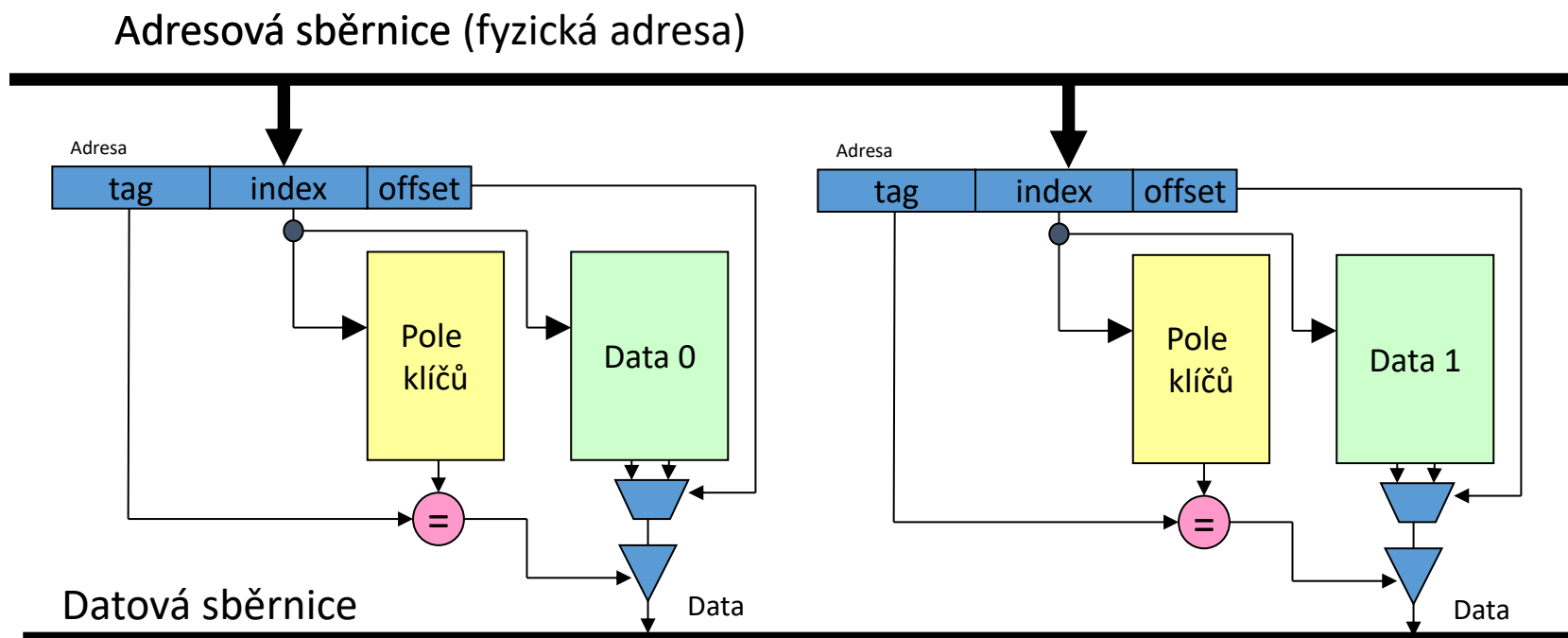
Plně asociativní paměť



Přímo adresovaná keš – keš se stupněm asociativity 1



Keš se stupněm asociativity 2



Kromě stupně asociativity 2 se často užívá stupeň asociativity 4

Odstraňování položek z keše při stupni asociativity větším jak jedna

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

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

S keší v tomto případě ušetříme 105ns

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

Časová a prostorová lokalita (souvisí s kešemi, jsou to vlastnosti programu)

Keše využívají časové a prostorové lokality. Bez těchto vlastností programu by nebyly účinné.

Časová lokalita znamená, že se v programu používají opakovaně data ze stejné adresy. Tedy dvě po sobě jdoucí instrukce v krátkém časovém okamžiku použijí data ze stejné adresy (obsah stejné proměnné).

Prostorová lokalita znamená, že se v programu používají opakovaně data z blízkých adres. Načteme-li najednou celý řádek keše, pak čtení několika následujících adres nevede k výpadku.

Implementace algoritmů, při kterých chceme dosáhnout maximálního výkonu procesoru musíme optimalizovat s ohledem na časovou a zejména prostorovou lokalitu, aby byly keše co nejúčinnější.

Instrukce ovlivňující keše

Protože se keše se chovají transparentně, není důvod zavádět speciální instrukce pro práci s nimi.

Existují následující výjimky:

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

`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 povolena jen operace čtení. Počet bitů čítače programu musí korespondovat s velikostí tohoto prostoru.

- **Adresní prostor dat**

Do toho 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 toho 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. `write(adr, v1), v2=read(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.

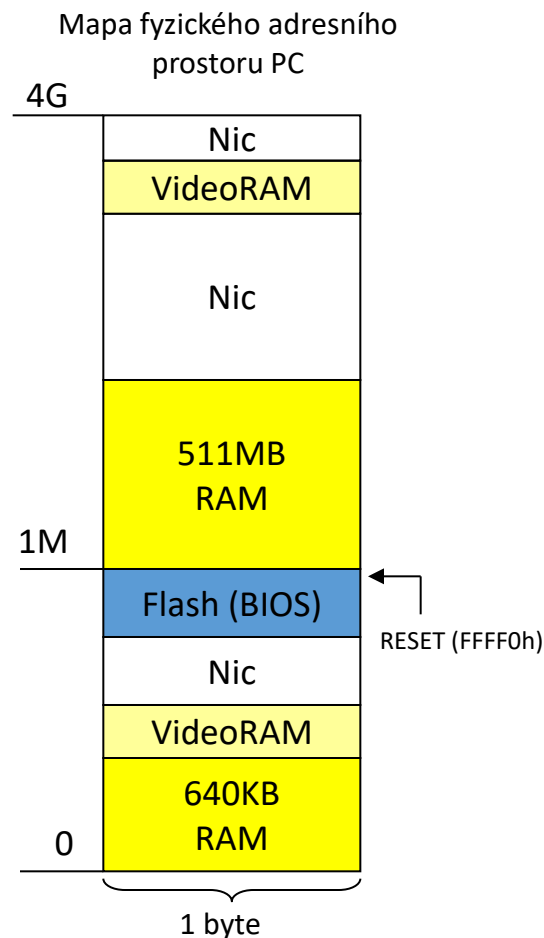
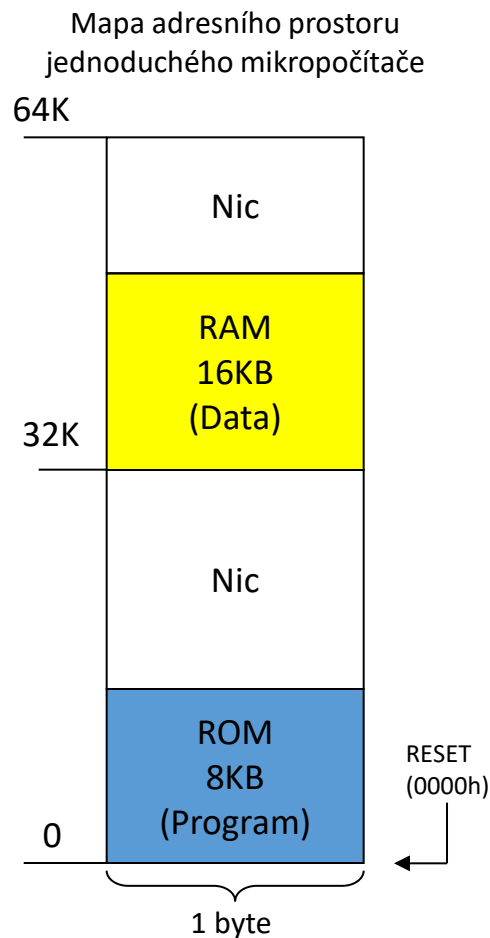
- **Virtuální adresní prostor**

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).

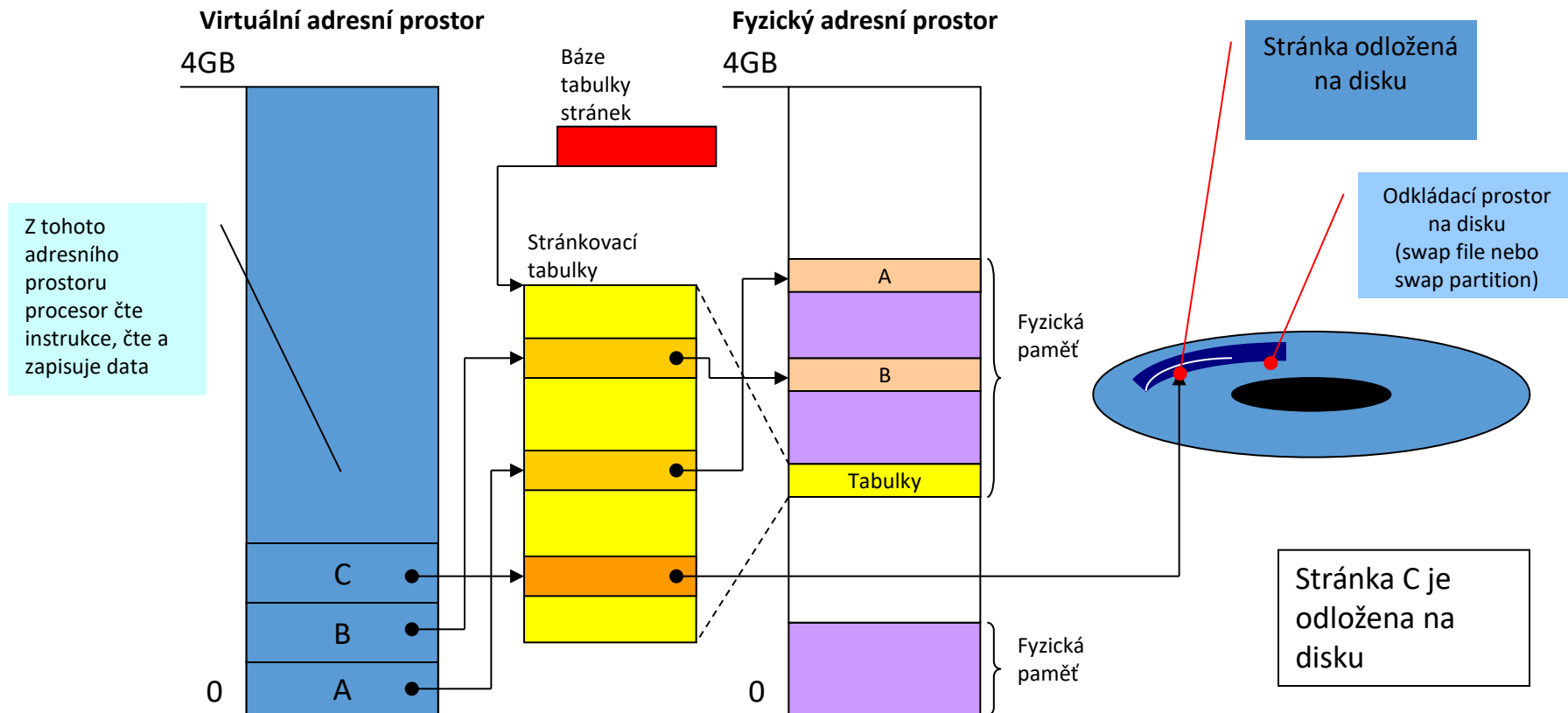
Mapa adresního prostoru (mapa paměti)

- 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

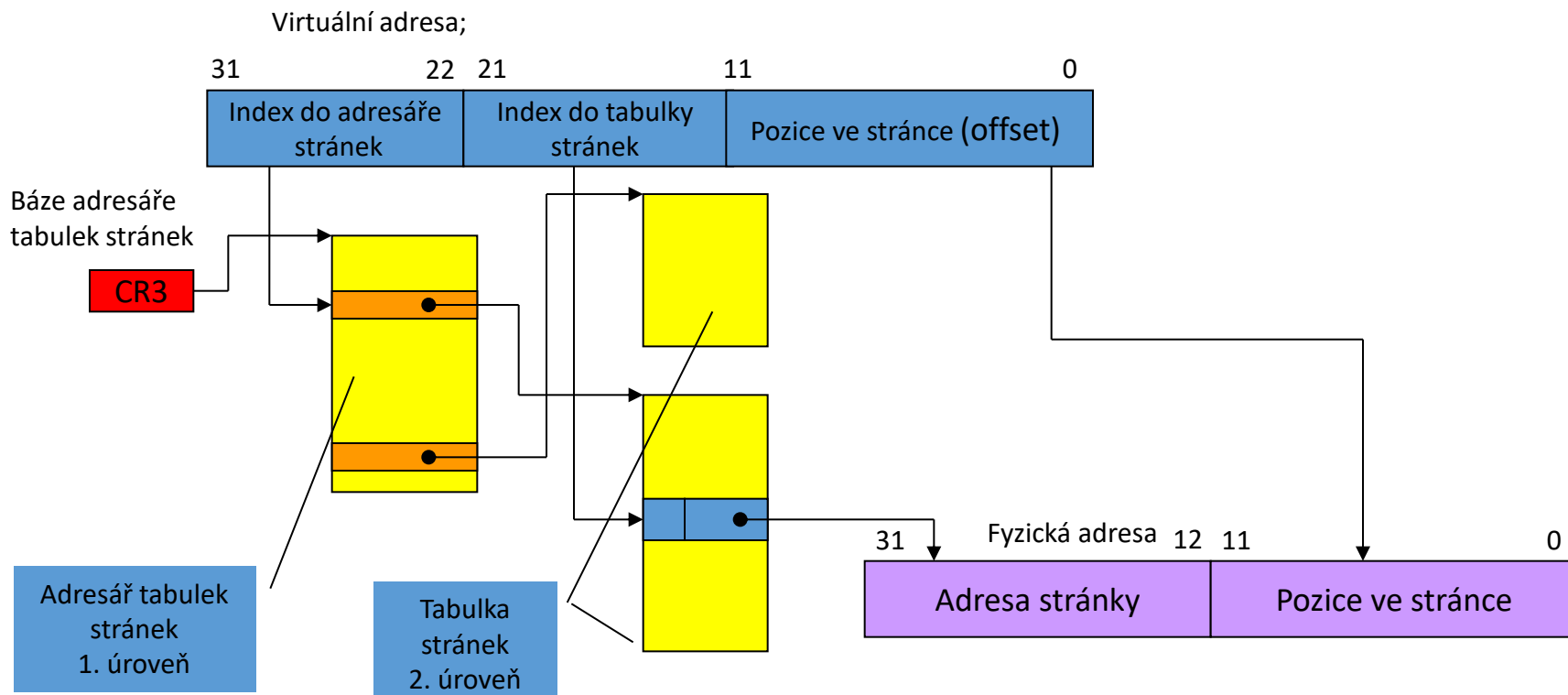


Virtuální adresní prostor - stránkováný



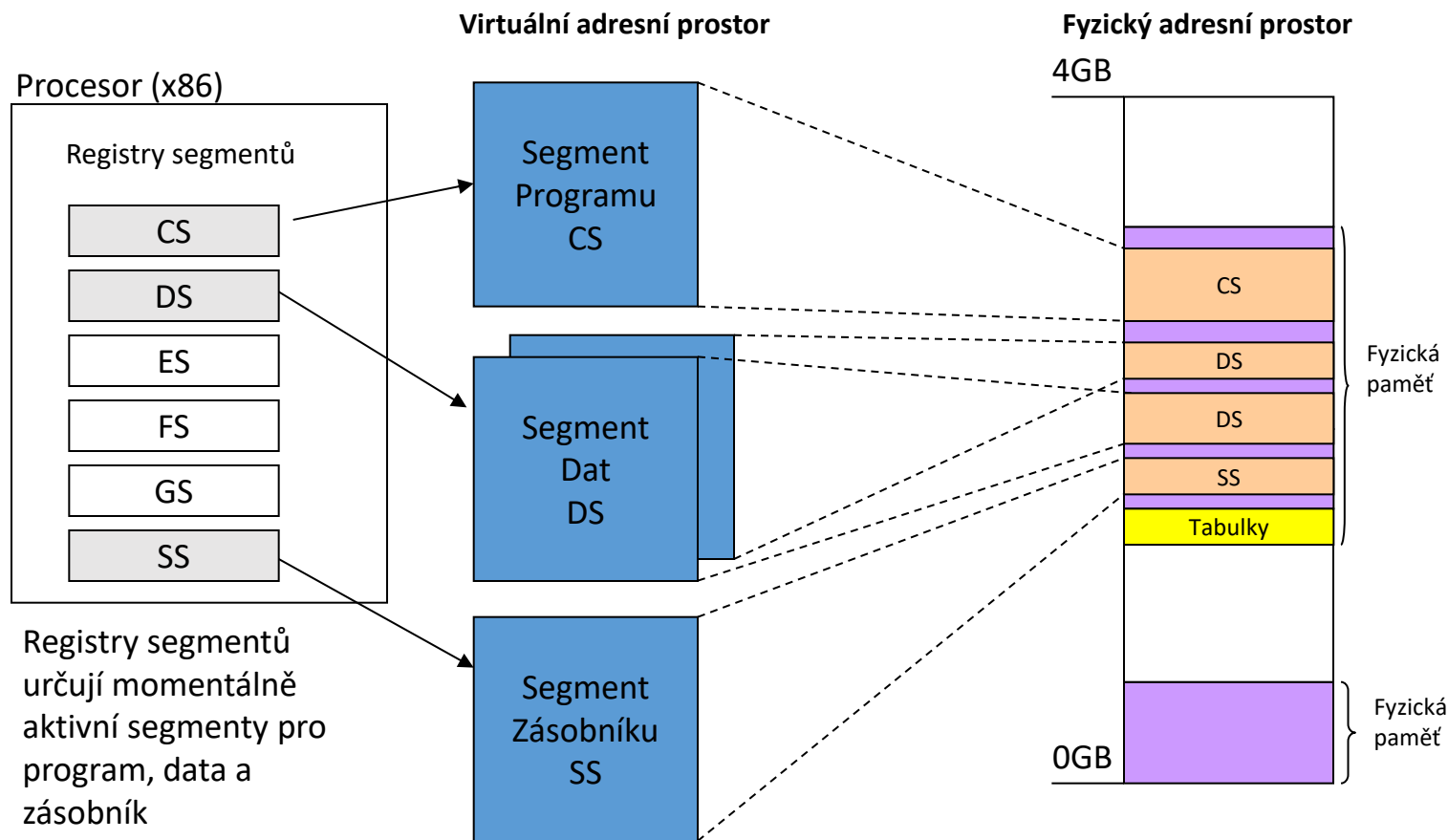
Virtuální adresní prostor je rozdělen do **stránek** (bloků paměti o velikosti 4KB). Operační systém modifikuje záznamy v tabulkách a tím mapuje stránky z fyzické paměti do stránek ve virtuálním adresovém prostoru. Každý proces má své vlastní tabulky stránek, které mapují jinou část fyzické paměti. Tím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

Stránkování – překlad adresy



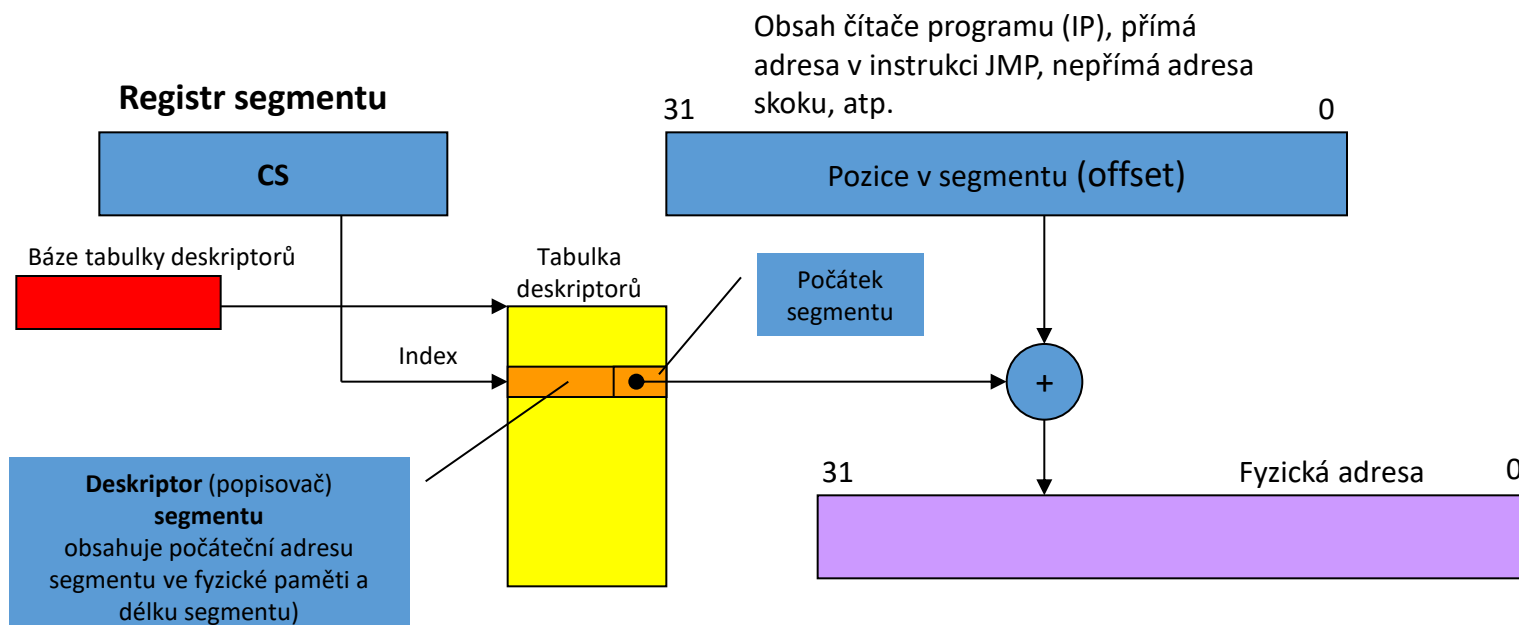
Pokud položka v tabulce stránek indikuje, že stránka je odložena na disku (tj. není v paměti) je vygenerována výjimka a operační systém zpracuje tuto výjimku tak, že odloženou stránku načte z disku zpět do paměti. Pokud není v hlavní paměti místo, pak se nejprve odloží na disk stránka, která se dlouho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

TLB (Translation Lookaside Buffer)

TLB je asociativní paměť (s částečnou nebo plnou asociativitou), která urychluje překlad adres (virtuální->fyzická).

Klíčem je virtuální adresa, hodnotu je fyzická adresa. Např. při stránkování potřebujeme 2-3 čtecí operace (podle hloubky hierarchie stránek) než jsme schopni určit fyzickou adresu.

TLB proces urychluje tím, že v sobě uchovává páry virtuální adresa – fyzická adresa a v těchto párech se vyhledává asociativně s klíčem virtuální adresa.

Přímý přístup do paměti (DMA – Direct Memory Access)

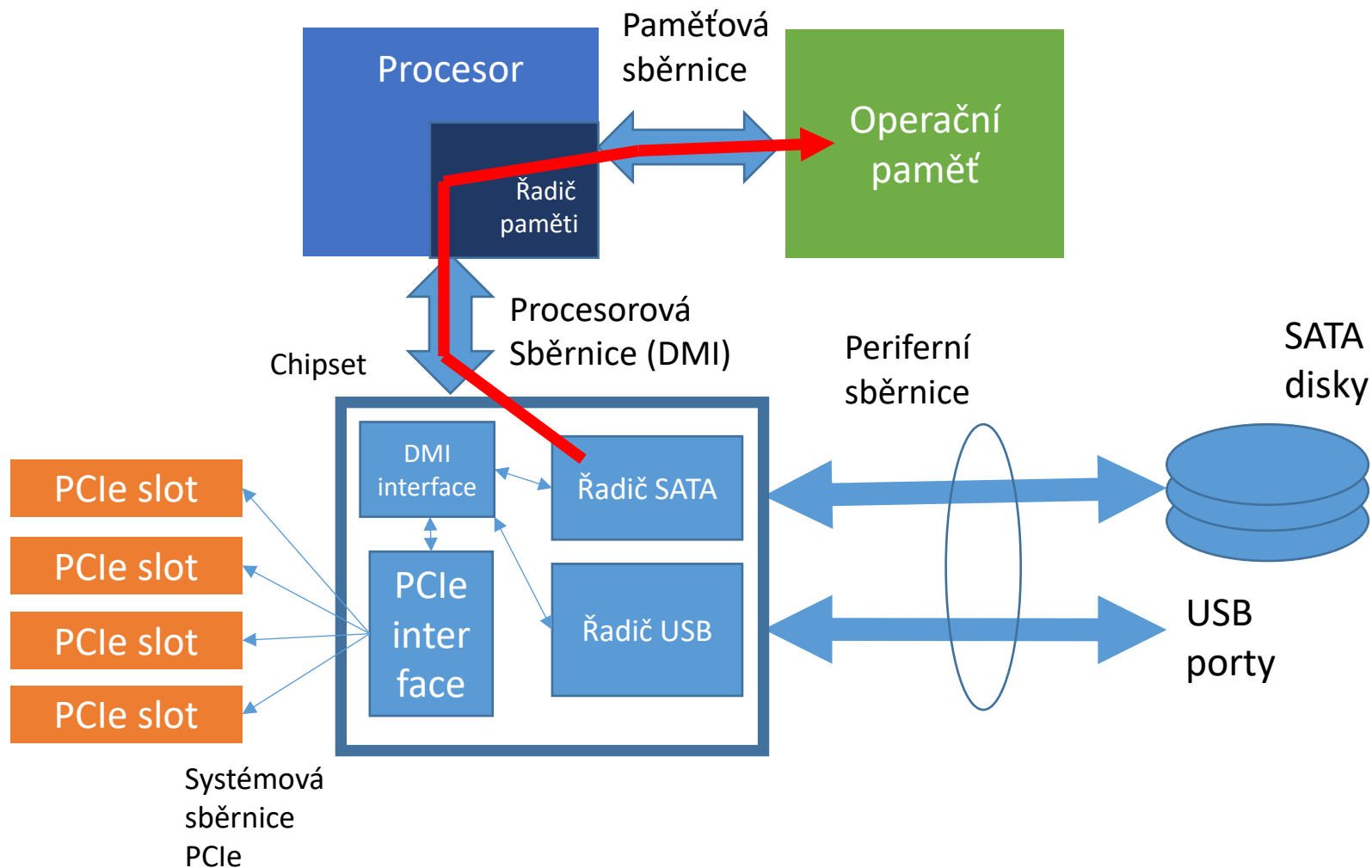
Jedná se o přenos dat mezi periferií a operační pamětí bez účasti procesoru. Bez účasti procesoru znamená, že v procesoru neběží program, který by přenos zajišťoval. Přenos je zajištěn na hardwarové úrovni řadičem DMA.

Využíváno řadiči disku a řadiči USB.

Řadič DMA musí umět převzít systémovou sběrnici (tj. generovat všechny adresové a řídicí signály pro paměť a periférii), čímž dočasně blokuje přístup procesoru k operační paměti. Procesor v řadiči (zápisem do jeho registrů) nastaví typicky počáteční adresu v operační paměti, kam se data přenesou, délku přenosu v bytech a odstartuje přenos. Řadič DMA informuje přerušením procesor, že byl přenos ukončen.

U prvních PC (do zavedení sběrnice PCI) byly DMA přenosy zajišťovány řadiči DMA na motherboardu. Dnes jsou součástí periférie a periférie přebírá sběrnici (Busmastering)

Přímý přístup do paměti (disk->memory)

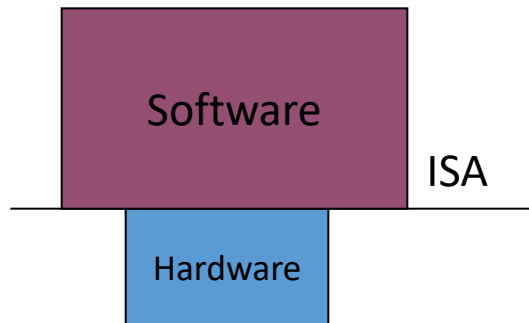


Instruction Set Architecture (ISA)

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



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 nejčastějších 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) – redukovaná 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

CISC vs. RISC

CISC (jednoduché instrukce)



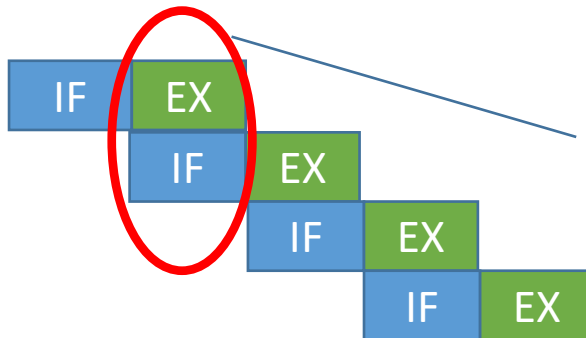
Výkon procesoru v instrukcích za sekundu je roven převrácené hodnotě součtu trvání fáze IF a EX.

CISC (složité instrukce)



Výkon procesoru je vyšší (méně fází IF) a silně závisí na podílu složitějších instrukcí. Jednodušší instrukce se typicky používají častěji. Nikdy není možno dosáhnout zdvojnásobení výkonu.

RISC (má pouze jednoduché instrukce)



RISC architektura využívá proudového zpracování instrukcí. Načtení instrukce a vykonání předchozí instrukce probíhá paralelně. IF a EX fáze musí trvat stejně dlouho a počet instrukcí vykonaných za sekundu je převrácená hodnota trvání fáze EX. V ideálním případě se výkon procesoru zdvojnásobí proti variantě CISC jednoduché instrukce. Proudové zpracování instrukcí připomíná pásovou výrobní linku.

Typy ISA

- Střadačová (mikrokontroléry 8051, M68HC05)
- Zásobníková (Java VM)
- Registrová
 - load-store (ATMega328, ARM)
 - register-memory (x86)
 - memory-memory (výjimečná)

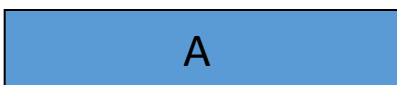
Střadačová ISA

Střadačová ISA

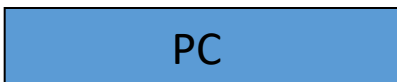
- Má jeden registr (střadač), který je implicitně užíván aritmetickými a logickými operacemi
- Aritmetické a logické operace mají jako jeden operand střadač a druhý je čten z paměti. Výsledek je ukládán do střadače.
- Jednoduchý hardware
- Příliš mnoho operací přesunu (MOV) při vkládání operandu do střadače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

Střadačová ISA - příklad

Střadač



Program counter



Registr příznaků (Program Status Word)



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

Registrová ISA

Registrová ISA (Register-Memory)

- 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

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

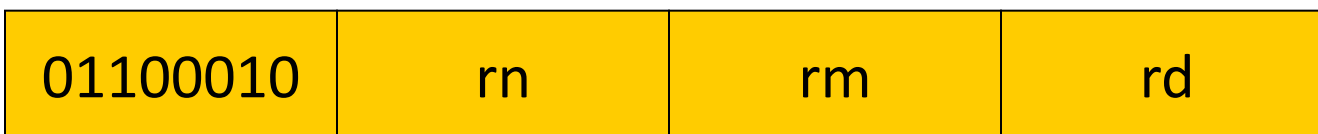
V Load-store architektuře jsou aritmetické operace prováděny pouze mezi registry. Pro přístup do paměti se používají instrukce load (LD) nebo store (ST).

Registrová – příklady formátů instrukcí

tříoperandová instruce

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

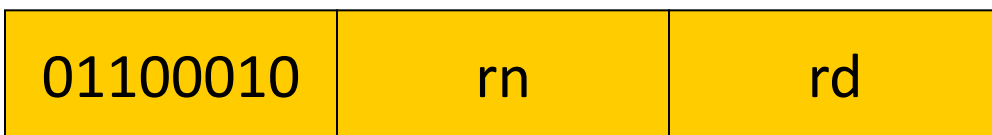
OZ (operační znak)



dvouoperandová instruce

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

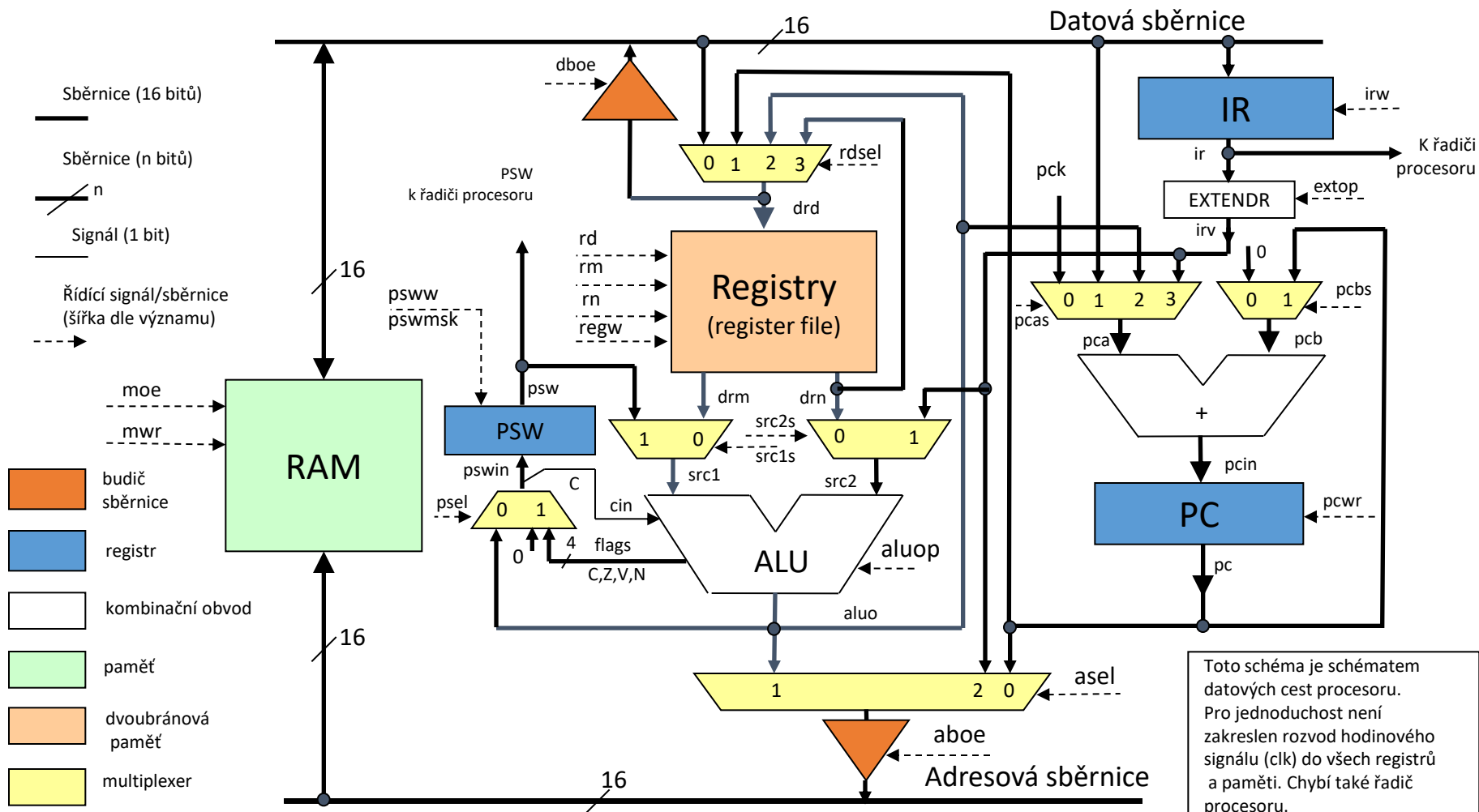
OZ



Rd značí cílový registr

Operační znak odlišuje
jednoznačně instrukce
od sebe !

Registrová šestnáctibitová architektura typu load-store

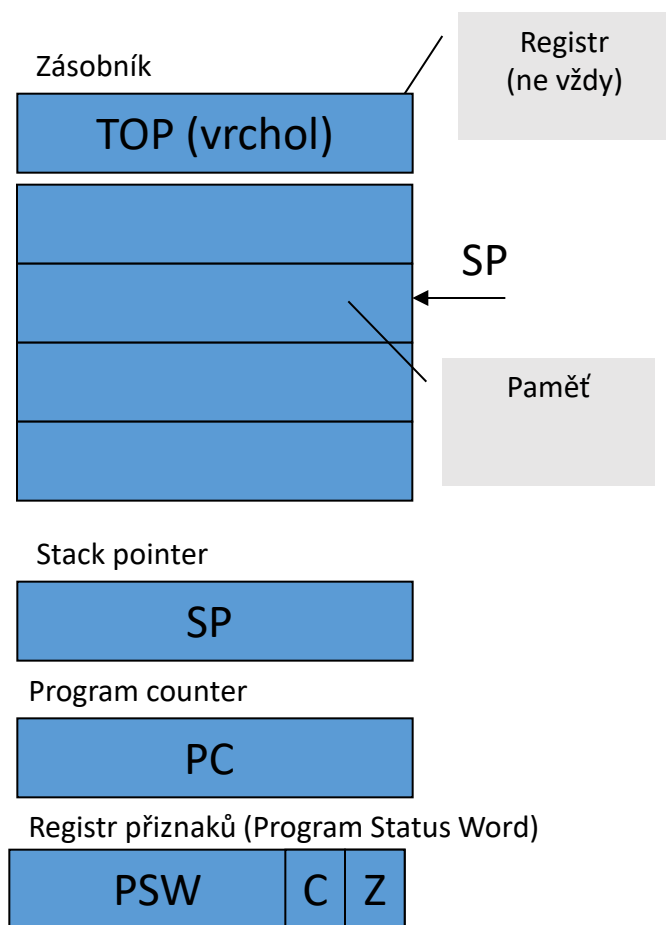


Zásobníková ISA

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 - 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

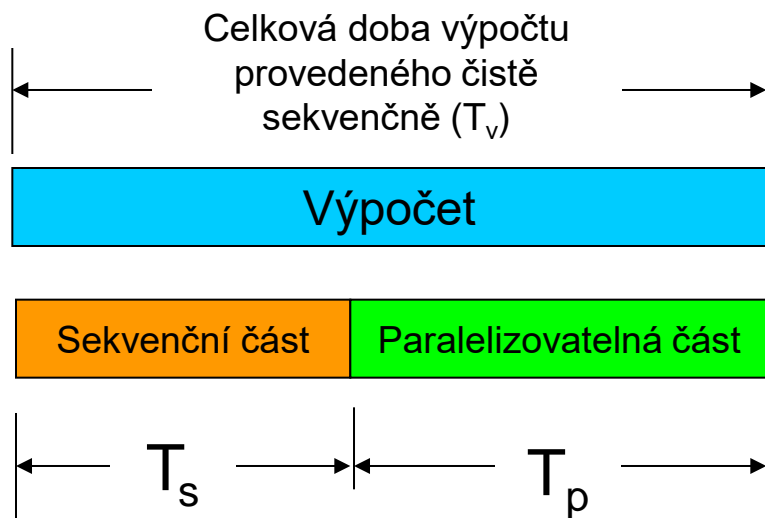
Java Bytecode

```
// Bytecode stream: 03 3b 84 00 01 1a 05 68 3b a7 ff f9
// Disassembly:
iconst_0 // 03
istore_0 // 3b
iinc 0, 1 // 84 00 01
iload_0 // 1a
iconst_2 // 05
imul // 68
istore_0 // 3b
goto -7 // a7 ff f9
```

Příklad převzat z: <http://www.javaworld.com/article/2077233/core-java/bytecode-basics.html>

Paralelizace a Amdahlův zákon

Amdahlův zákon

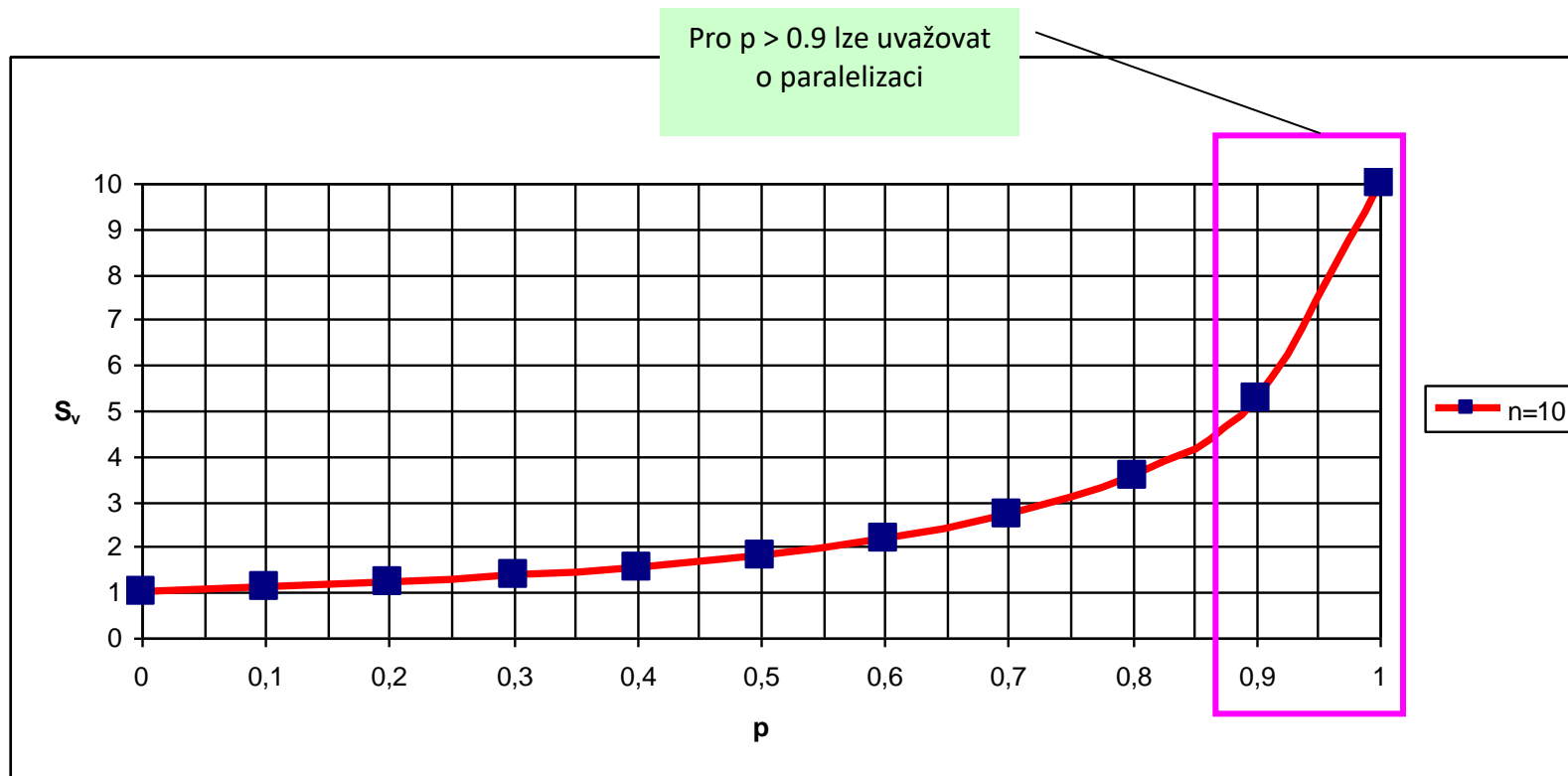


Poměr paralelizovatelné
části výpočtu k celkové
době výpočtu

$$p = \frac{T_p}{T_v}, p_{\%} = 100 \frac{T_p}{T_v} [\%]$$

$$S_v = \frac{T_v}{T_s + \frac{T_p}{n}} = \frac{n}{n \frac{T_s}{T_v} + \frac{T_p}{T_v}} = \frac{n}{n \left(1 - \frac{T_p}{T_v} \right) + \frac{T_p}{T_v}} = \frac{n}{n(1-p) + p}$$

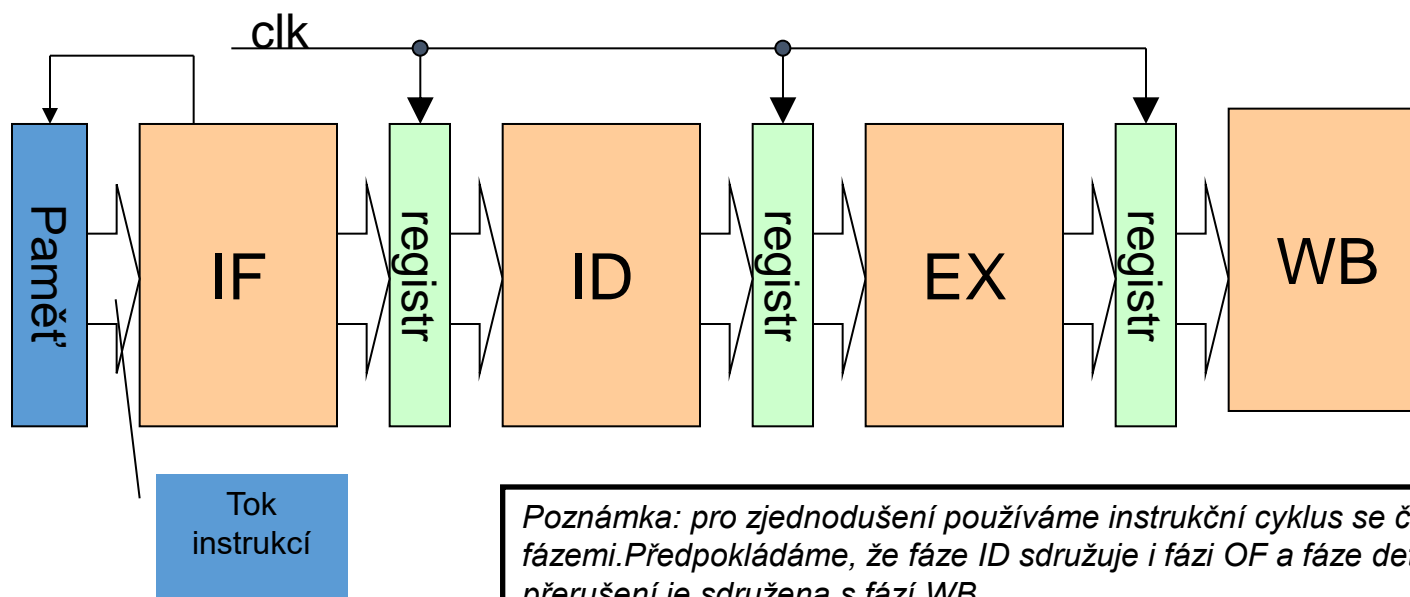
Závislost zrychlení S_v na p



Proudové zpracování instrukcí a dat

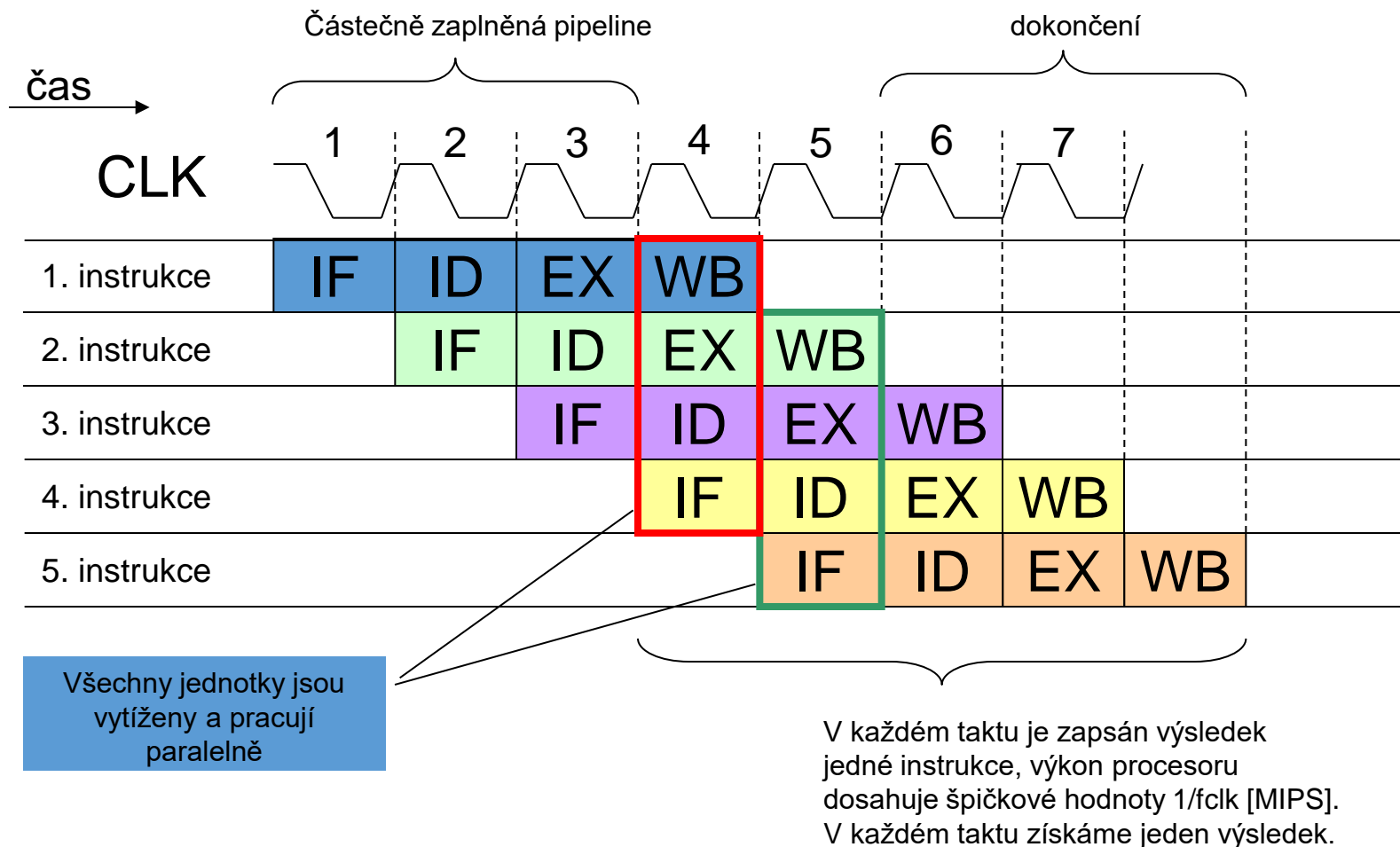
Proudové zpracování instrukcí (pipeline)

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.



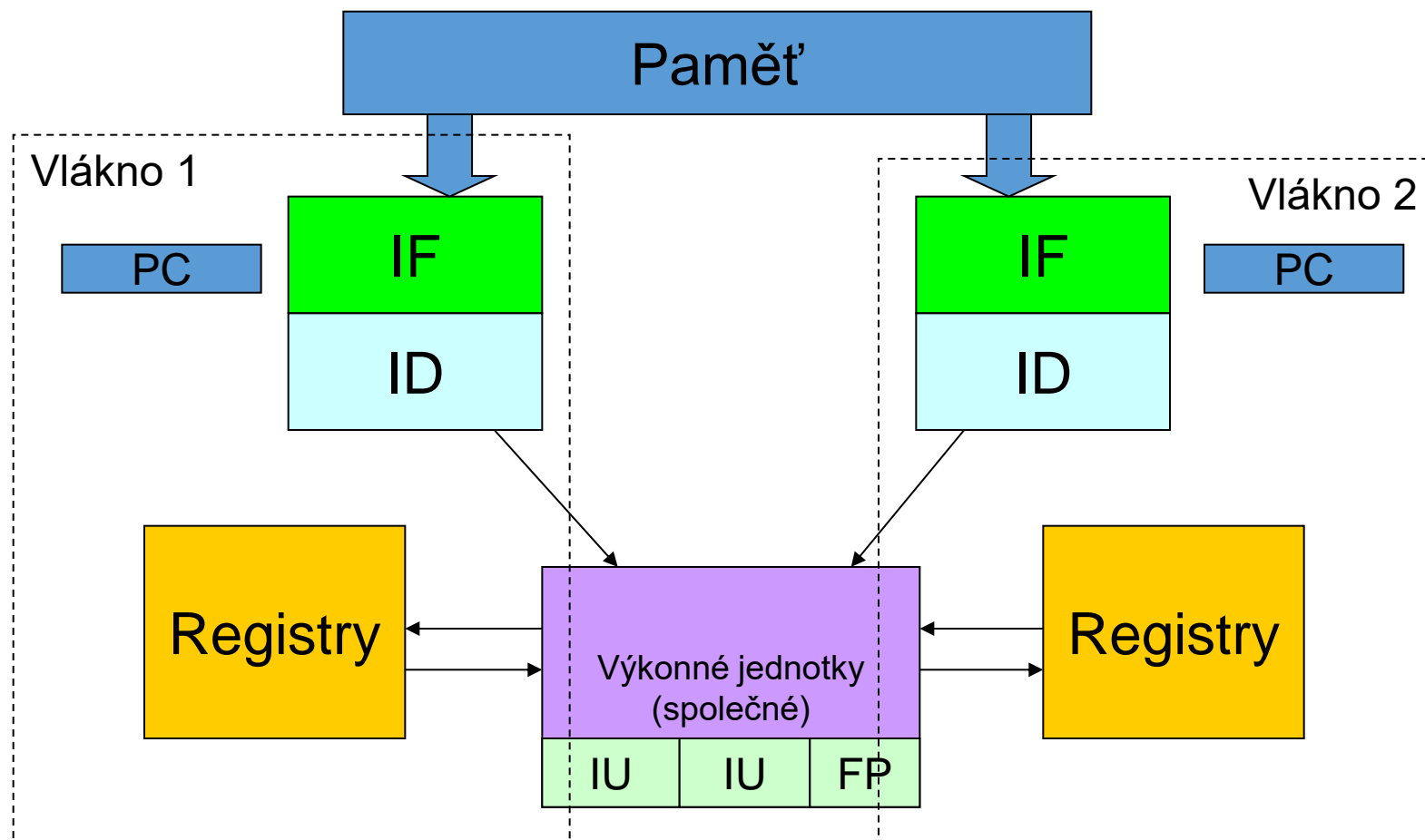
Poznámka: pro zjednodušení používáme instrukční cyklus se čtyřmi fázemi. Předpokládáme, že fáze ID sdružuje i fázi OF a fáze detekce přerušení je sdružena s fází WB...

Průběh proudového zpracování instrukcí

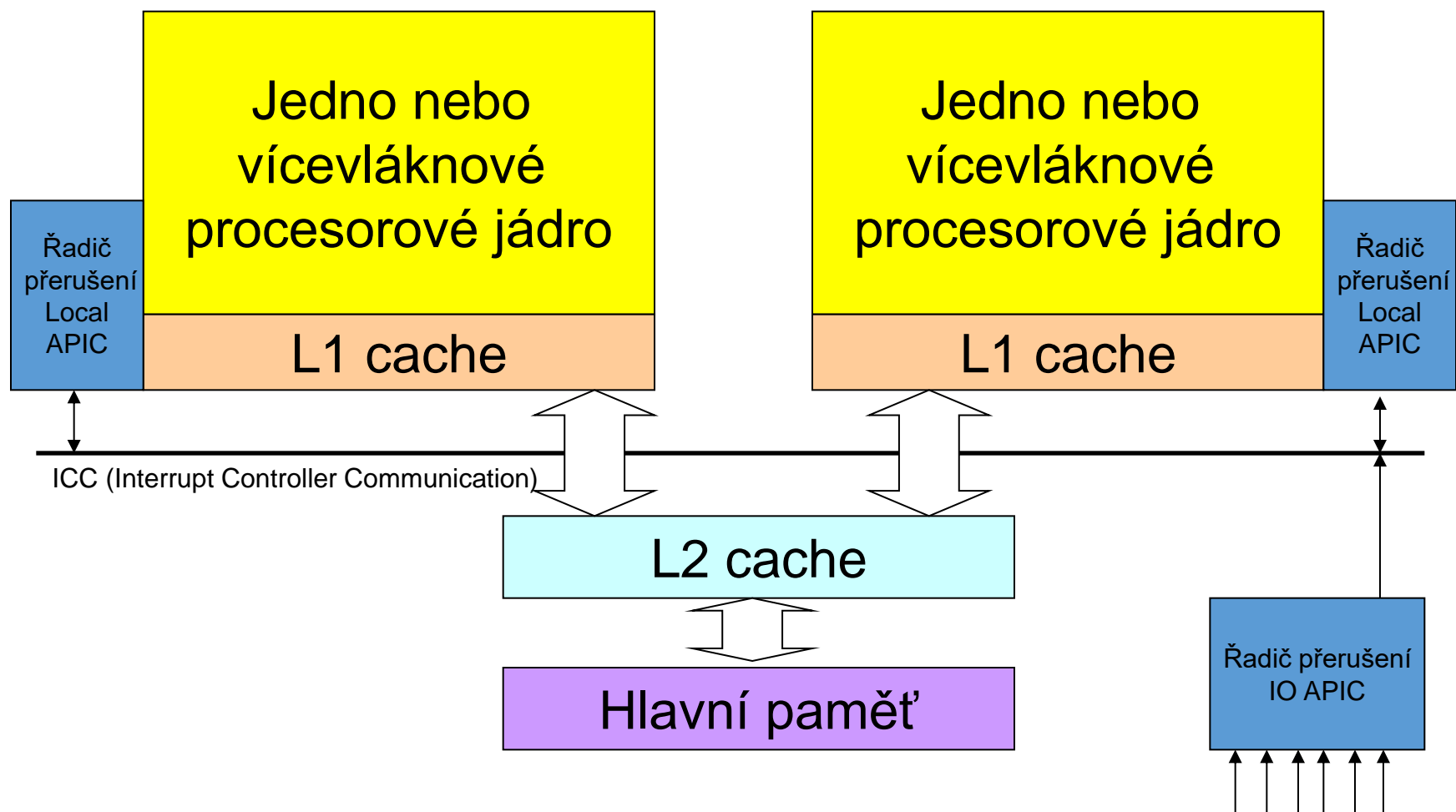


Složitější architektury využívající paralelismu

Vícevláknové procesory



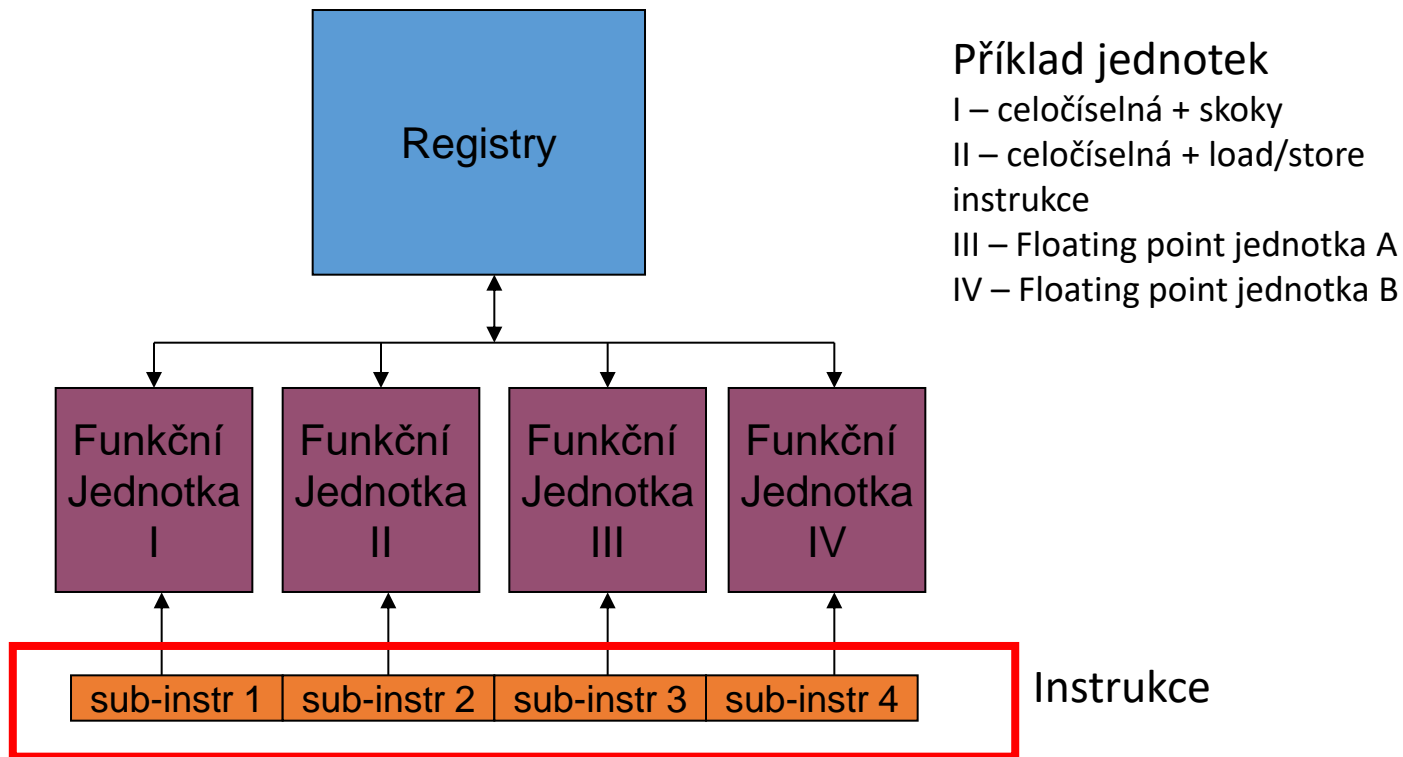
Vícejádrové procesory



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



SIMD Instrukční sady (datový paralelismus)

Intel SSE, SSE2, SSE3, SSE4, SSE4.1, SSE4.2

Registry XMM0-XMM7(15) (128 bitů)

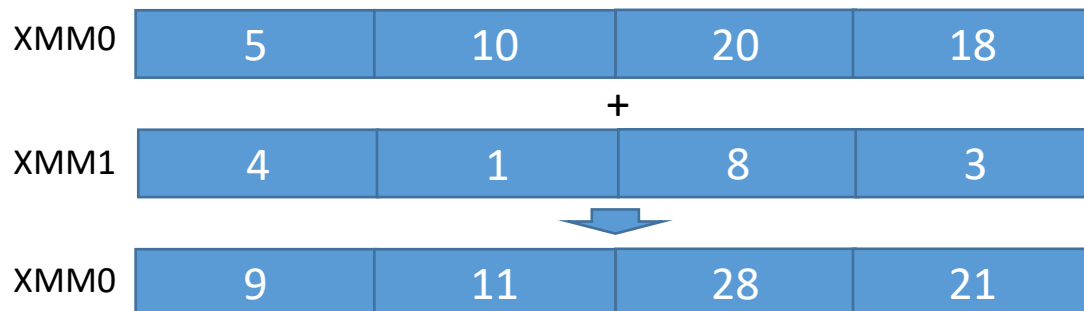
Intel AVX

Registry XMM0-XMM15 (128 bitů),
YMM0-YMM15 (256 bitů)

Intel AVX-512

Registry XMM0-XMM31 (128 bitů),
YMM0-YMM31 (256 bitů)
ZMM0-ZMM31 (512 bitů)

PADD (SSE2), obdobně ADDPS pro 4xFP32



SIMD (Single Instruction Multiple Data) - architektura procesoru, kde pro jednu instrukci provedeme více stejných operací. SIMD jednou z typů paralelních architektur dle Flynnovy klasifikace.