

Architektúry počítačových systémov

2. rok ZS

Osnova

- Základné pojmy
- Počítačové systémy
- Číslicové systémy
- Číselné sústavy
- Logické hradlá
- Vlastnosti číslicových súčiastok
- Tranzistory CMOS

Základné pojmy

- **Informatika, informatizácia a informačná spoločnosť**
 - Informatika je veda o získavaní, zbere, prenose, triedení, ukladaní, uchovávaní (pamätaní), aktualizovaní, spracovaní, vyhodnocovaní a využívaní informácií na úrovni signálov, údajov, symbolov, správ, poznatkov a znalostí.

Základné pojmy

- **Výpočtová technika**

- skúma zákonitosti a princípy tvorby informačných procesov a spôsoby ich realizácie v konkrétnych, reálnych informačných systémoch;
- zaoberá sa vlastnosťami a zákonitosťami týchto procesov a ich algoritmickou realizáciou pomocou prostriedkov výpočtovej techniky;
- zahrňuje tvorbu, vývoj a využitie programových a technických prostriedkov výpočtovej techniky, ako nástroja na automatizované a automatické spracovanie informácií, reprezentujúcich údaje, signály, správy a poznatky v jednotlivých aplikáčnych oblastiach informatiky, vrátane komunikačných procesov v sietovom prostredí.

Základné pojmy

Číslicový počítač (ČP) je zložitý univerzálny číslicový systém (automat) určený na **samočinné vykonávanie** postupnosti operácií (výpočtov) nad údajmi zobrazenými **číslicovým kódom**, na základe vopred pripraveného a v **pamäti uloženého programu** (algoritmu).

Základné pojmy

- **Systém a jeho vlastnosti**
 - **Systém je súhrn prvkov** komponovaných do jedného celku za účelom dosiahnutia definovaného cieľa (napr.: automatizácia výpočtu na základe algoritmu).
 - **Kompozícia systému** nepredstavuje ich jednoduchý súčet, ale vytvára nové vlastnosti, ktoré v jednotlivých prvkoch nie sú prítomné. Takáto vlastnosť sa nazýva **zložitosť systému** (napr.: elektronický prvak - logický člen - funkčná jednotka - počítačový systém).
 - Opísanie systému sa uskutočňuje na základe určenia jeho **funkcie** a **štruktúry**.

Základné pojmy

- **Funkcia systému** vyjadrená prostredníctvom špecifikácie a opisu procesov, ktoré sú v ňom definované, predstavuje pravidlo na dosiahnutie požadovaného cieľa.
- **Štruktúra systému** vyjadrená prostredníctvom abstraktných alebo inžinierskych foriem zobrazenia (grafy, jazyky, schémy a pod.) vyjadruje kompozíciu jeho prvkov a ich vzájomných väzieb.

Základné pojmy

- **Organizácia systému** (funkčná a štruktúrna) je spôsob vytvorenia systému na báze kompozície prvkov z ktorých pozostáva za účelom dosiahnutia zadaných funkčných vlastností.
- Systém vytvorený na báze jedného alebo viacerých ČP resp. ich komponentov sa nazýva **počítačový systém (PS)**, ktorý je charakterizovaný svojou
 - funkčnou organizáciou,
 - štruktúrnou organizáciou.

Základné pojmy

- **Funkčná organizácia PS** je definovaná
 - formou zobrazovania informácií,
 - štruktúrou inštrukčných súborov,
 - charakterom vnútorného a vonkajšieho riadenia operácií,
 - spôsobom riadenia procesu spracovania informácií,
 - prístupovým režimom používateľov k PS a jeho aplikačnými možnosťami.

Základné pojmy

- **Štruktúrna organizácia PS** je definovaná
 - logickým a systémovým usporiadaním jeho komponentov,
 - spôsobom ich vzájomnej komunikácie v priebehu spracovania informácií,
 - konštrukčnou a technologickou realizáciou jeho komponentov.

Základné pojmy

- Architektúra PS **zahrňuje**
 - požiadavky programových prostriedkov vrátane techniky programovania úloh - **inštrukčno orientovaná architektúra** (**Instruction Set Architecture, ISA**),
 - vnútornú organizáciu jeho technických prostriedkov - **implementačno orientovaná architektúra / mikroarchitektúra** (**Implementation Architecture, IA**).

Základné pojmy

- Z hľadiska požiadaviek programovania na úrovni assembléra **inštrukčno orientovaná architektúra PS predstavuje**
 - jeho **abstrakciu prostredníctvom súboru inštrukcií**
 - kódy operácií (operačné kódy);
 - adresovacie módy;
 - špecifikáciu registrov;
 - virtuálnu pamäť a pod.

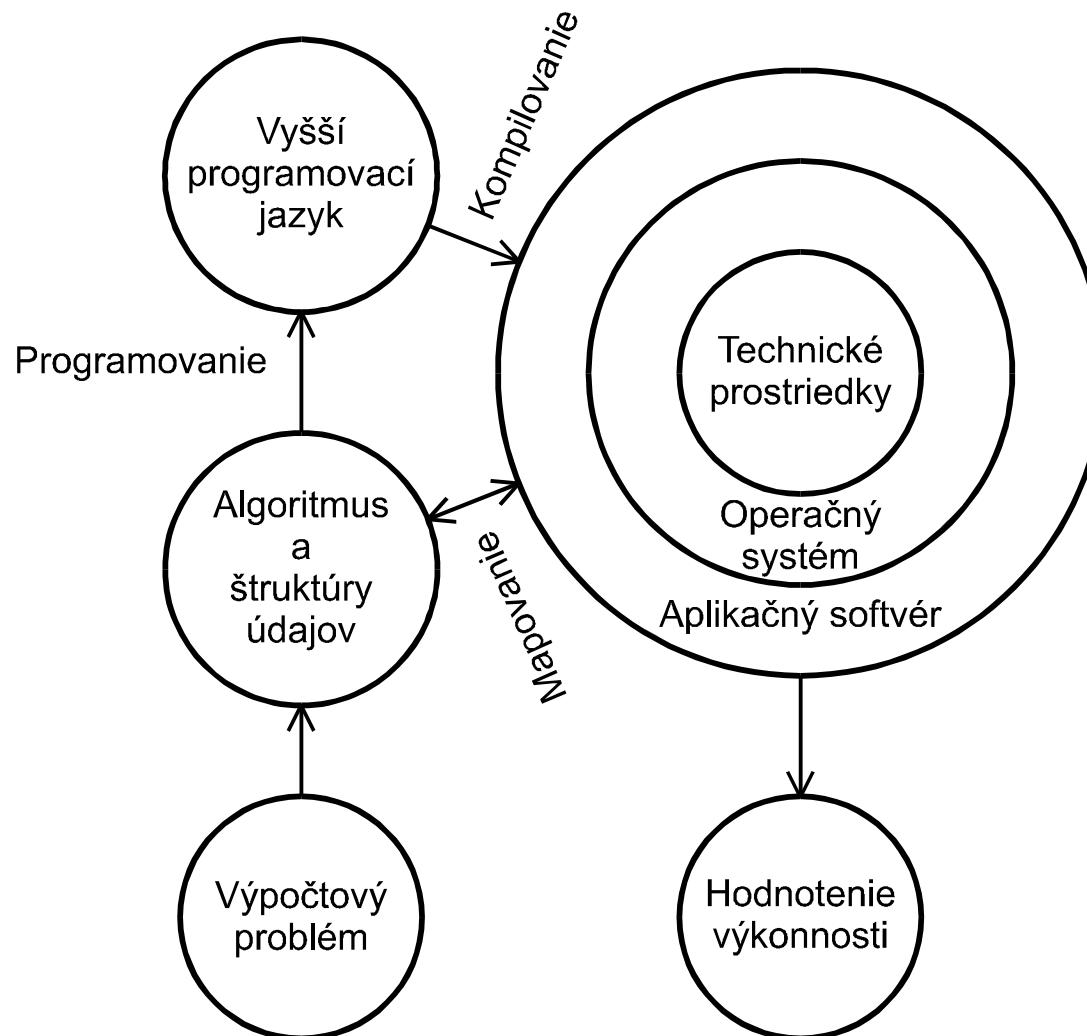
Základné pojmy

- Z hľadiska implementácie tech. prostriedkov **implementačno orientovaná architektúra PS predstavuje**
 - jeho **abstrakciu na úrovni štruktúrnej organizácie**
 - centrálnych procesorových jednotiek (CPU),
 - komponentov pamäťového podsystému (cache, fyzická pamäť, ...),
 - zbernicových systémov,
 - mikrokódu,
 - prúdových funkčných jednotiek,
 - jednotiek na paralelné spracovanie a pod.

Základné pojmy

- **Výpočtový problém**
 - spracovania numerických údajov;
 - spracovania signálov;
 - spracovania symbolov a znalostí;
 - spracovania databáz;
 - spracovanie komunikačných procesov.
- Ich charakteristické vlastnosti ovplyvňujú štruktúru komponentov PS.

Model PS

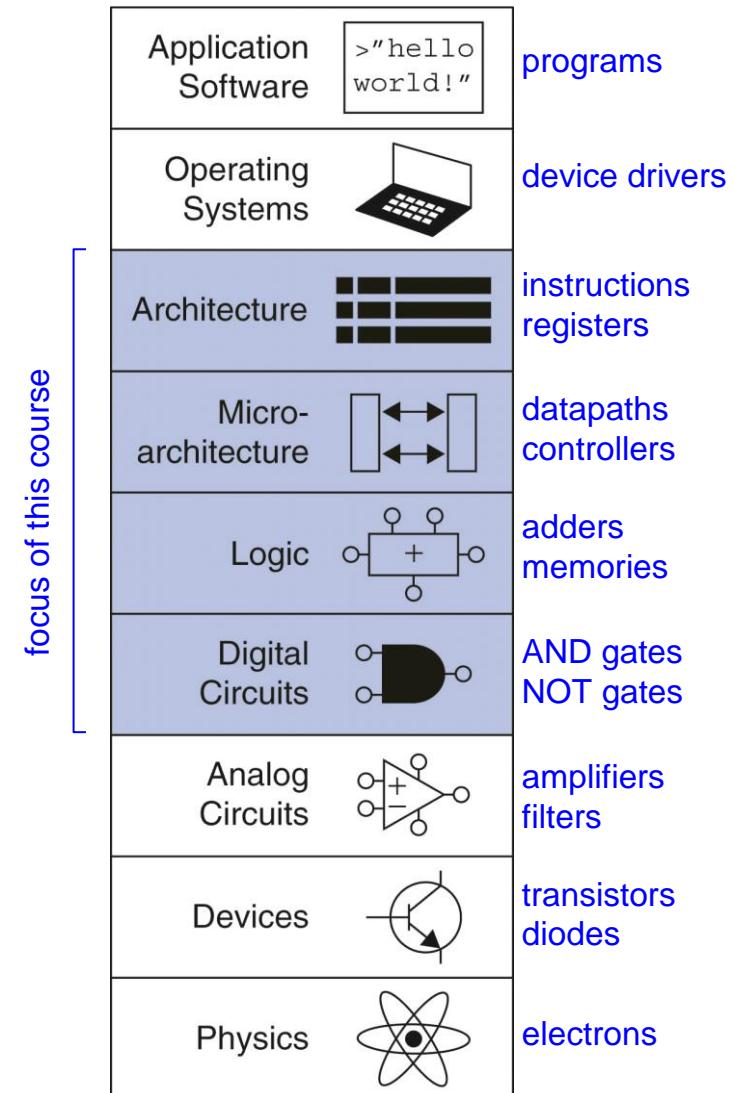


Piliere číslicových systémov

- Abstrakcia
- Disciplína
- Trojica HMJ
 - Hierarchia
 - Modularita
 - Jednotnosť

Abstrakcia

- Špecifikuje kompozíciu systému.
- Skrýva detaile ak nie sú podstatné z hľadiska návrhu hardvéru na príslušnej úrovni.
- Systém môže byť zobrazený mnohými rôznymi úrovňami abstrakcie.
- Na najnižšej úrovni abstrakcie sa nachádza opis komponentov systému z hľadiska fyzikálnych zákonov opisujúcich tok elektrónov
- Na najvyššej úrovni abstrakcie sa nachádza aplikačný softvér, špecifikujúci účel systému.
- V rámci tohto predmetu sa budeme zaoberať s úrovňami
 - logických hradieb
 - kombinačných a sekvenčných logických obvodov definujúce komponenty PS
 - mikroarchitektúry
 - architektúry



Disciplína

- Úmyselné obmedzenie návrhových rozhodnutí, aby sme mohli pracovať produktívnejšie a na vyššej úrovni abstrakcie

Disciplína

(pokračovanie)

- Činnosť digitálnych obvodov sa opisuje s väzbou na diskrétné rozdelenie úrovne napäťia, zatiaľ čo analógové obvody používajú napätie v celom rozsahu. Preto sú digitálne obvody podmnožinou analógových obvodov. Digitálne obvody sú však oveľa jednoduchšie. Obmedzením sa na digitálne obvody môžeme menšou námahou a „efektívnejšie“ vytvárať sofistikované systémy.

• **Hierarchia**

- Rozdeľovanie systému do modulov a každý z týchto modulov do podmodulov, pokial' nie sú jednotlivé časti jednoduché na pochopenie.

• **Modularita**

- Moduly majú dobre definované funkcie a rozhrania, takže sa ľahko spájajú bez nepredvídaných vedľajších účinkov

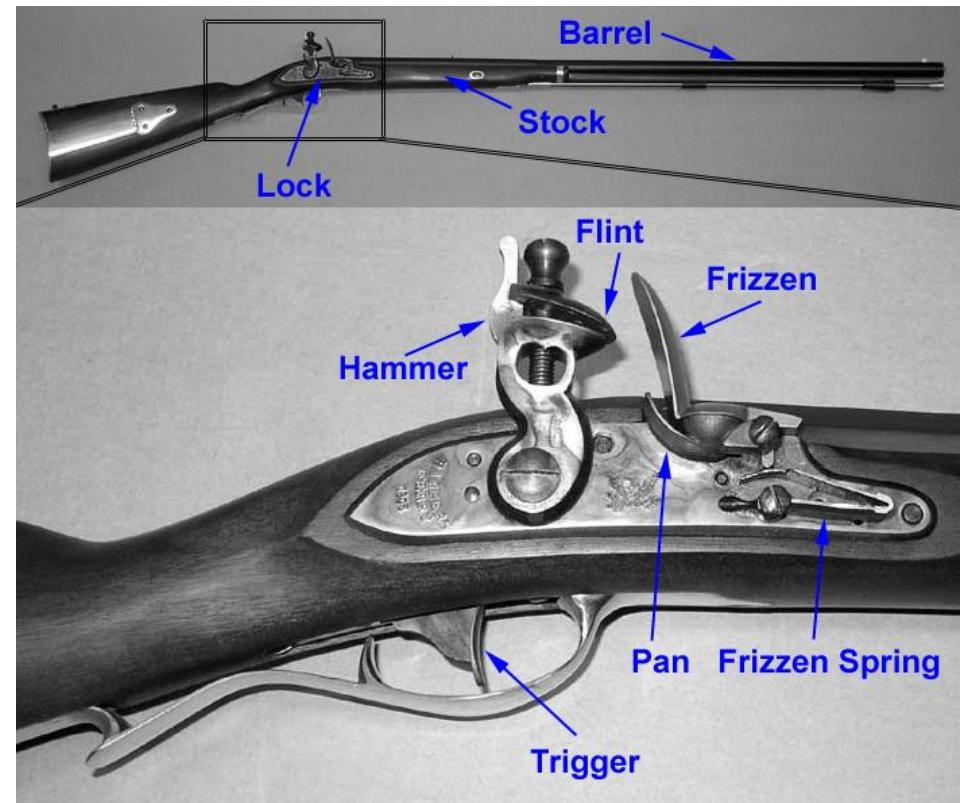
• **Jednotnosť**

- Hľadá jednotnosť medzi modulmi. Spoločné moduly sa opakovane používajú, čím sa znížuje počet odlišných modulov, ktoré je potrebné navrhnúť.

Príklad: Kresadlová puška

• Hierarchia

- **Tri hlavné moduly:**
zámok (lock), nábojová komora (stock), a hlaveň (barrel)
- **Podmoduly zámku*:**
kladivo/kohútik (hammer),
kohútik/kresadlo (flint),
ociel'ka (frizzen),
panvička (pan), pružina ocel'ky (frizzen spring)



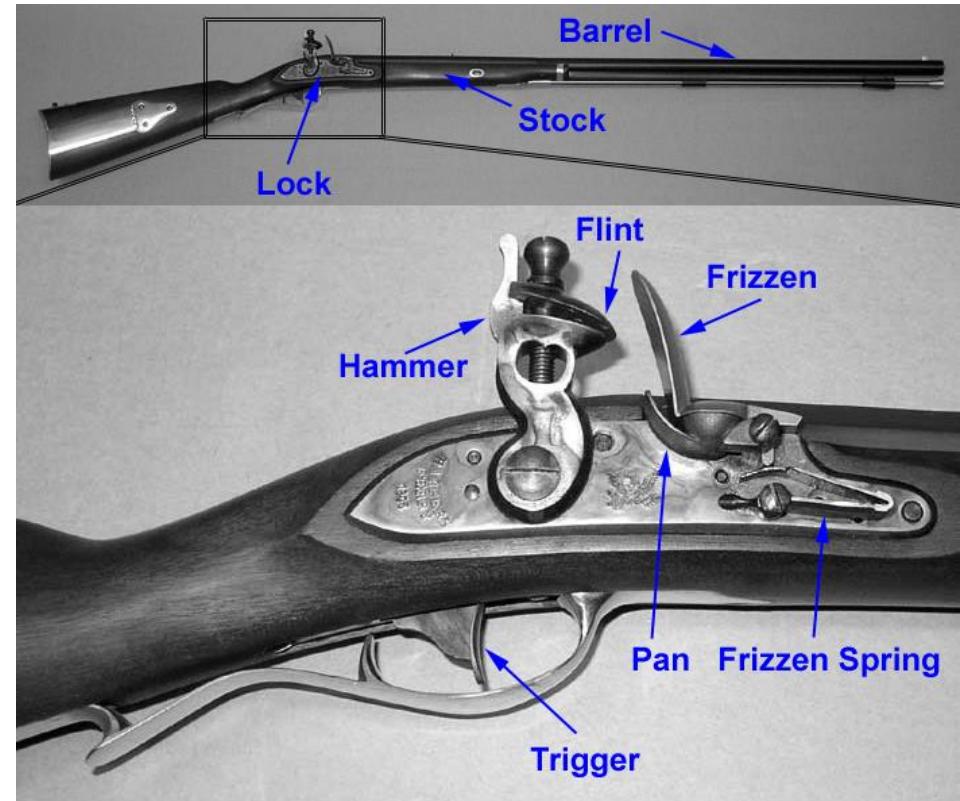
Príklad: Kresadlová puška

- **Modularita**

- Funkcia zásobníka: spája hlaveň a zámok
- Rozhranie zásobníka: dĺžka a pripojenie montážnych kolíkov

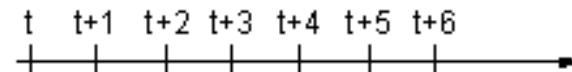
- **Jednotnosť**

- Dostupné náhradné diely



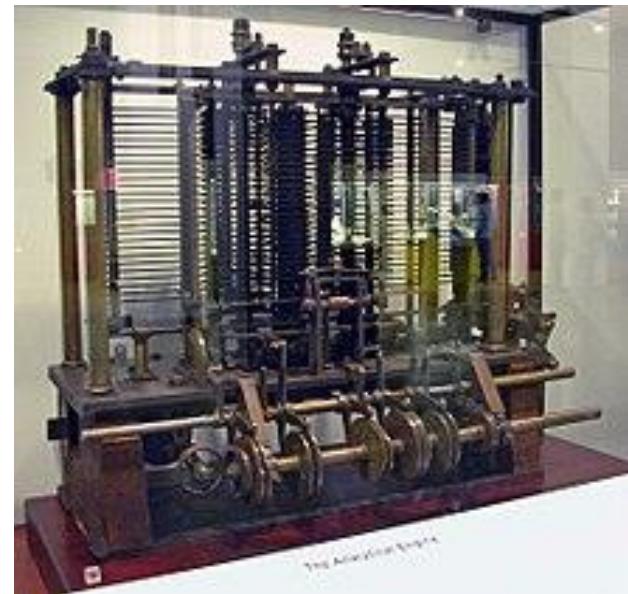
Diskrétné veličiny

- Väčšina fyzikálnych veličín je spojitá veličina
 - Napätie, prúd, frekvencia, ...
- Číslicové systémy pracujú s diskrétnymi veličinami
 - Premenné sú číslicové, čo znamená, že nadobúdajú hodnoty iba z konečných množín hodnôt.
 - Správanie sa systému je definované v diskrétnom čase, ktorý je tvorený usporiadanou postupnosťou diskrétnych bodov na spojitej časovej osi.



Analytický stroj

- Charles Babbage pracoval na ňom od 1833 – do smrti (1871)
- Považuje sa za prvý číslicový počítač.
- Pozostáva z pamäte a „mlynčeka“.
 - Pamäť sa skladal z mechanických registrov, schopných reprezentovať diskrétné hodnoty (0-9).
 - Mlynček slúžil ako aritmeticko-logická jednotka (ALU).



George Boole, 1815-1864

- Zakladateľ matematickej logiky, vybudoval základy, na ktorých stojí dnešná informatika.
- Systém na ohodnotenie pravdivostných hodnôt výrazov zložených z logických spojok AND, OR, NOT a logických premenných nadobúdajúcich iba dve hodnoty – 1 a 0.



Binárne hodnoty

- **Dve diskrétné hodnoty:**
 - 1 a 0
 - 1, TRUE, HIGH
 - 0, FALSE, LOW
- **1 a 0:** úroveň napäťia, smer otáčania kolies, hladina tekutiny, a pod.
- Číslicové obvody používajú príslušnú hladinu **napäťia** na reprezentáciu log. hodnôt 1 a 0
- **Bit:** *Binary digit – základná jednotka informácie*

Číslicové systémy, číselné sústavy

- **Základné pojmy**

- Ľubovoľnú diskrétnu informáciu v číslicovom počítači zobrazujeme prostredníctvom sústavy symbolov, ktorým môžeme priradiť rôzne číselné hodnoty.
- Reprezentácia číselných hodnôt vopred definovanou sústavou číslic sa nazýva číselná sústava (ČS).

- **Číselné sústavy (ČS)**

- Pozičné
- Nepozičné
- Symetrické
- Nesymetrické

Číslicové systémy, číselné sústavy

- Pozičné ČS s prirodzeným sledovaním váh
 - Rád číslice i
 - Váha číslice v_i
 - Základ číselnej sústavy z
 $v_i = z^i$
 - Abeceda symbolov $A = \{\alpha_1, \alpha_2, \dots, \alpha_m\}$
 $(\alpha_j) < z; j = 1, 2, \dots, m; m = z \geq 2$
 - Reálne číslo
- $$N_z = \pm (\alpha_j)_n (\alpha_j)_{n-1} \dots (\alpha_j)_0, (\alpha_j)_{-1} (\alpha_j)_{-2} \dots (\alpha_j)_{-k}$$

Číslicové systémy, číselné sústavy

- Pozičné ČS s prirodzeným sledovaním váh

– Hodnota číslica

$$[N_z] = \sum_{i=-k}^n \left((\alpha_j)_i \times z^i \right) [N_z]$$

$$[13,14]_{10} = 1 \times 10^1 + 3 \times 10^0 + 1 \times 10^{-1} + 4 \times 10^{-2}$$

Číslicové systémy, číselné sústavy

- **Pozičné ČS s prirodzeným sledovaním váh**

- Maximálne číslo
- Minimálne číslo
- Krok diskrétnosti
- Kapacita číselnej sústavy
- Počet zobrazujúcich rádov

$$N_{max} = z^n - z^k$$

$$N_{min} = z^{-k}$$

$$h = z^{-k}$$

$$K = z^m$$

$$m = \lceil \log_z(N + 1) \rceil$$

Číslicové systémy, číselné sústavy

- **Číselné sústavy**

- Desiatková $A = \{0,1,\dots,9\}$
- Binárna $A = \{0,1\}$
- Osmičková $A = \{0,1,\dots,7\}$
- Hexadecimálna $A = \{0,1,\dots,9, A, B, C, D, E, F\}$

Dvojková (binárna) sústava

- $2^0 =$
- $2^1 =$
- $2^2 =$
- $2^3 =$
- $2^4 =$
- $2^5 =$
- $2^6 =$
- $2^7 =$
- $2^8 =$
- $2^9 =$
- $2^{10} =$
- $2^{11} =$
- $2^{12} =$
- $2^{13} =$
- $2^{14} =$
- $2^{15} =$

Dvojková (binárna) sústava

- $2^0 = 1$
- $2^1 = 2$
- $2^2 = 4$
- $2^3 = 8$
- $2^4 = 16$
- $2^5 = 32$
- $2^6 = 64$
- $2^7 = 128$
- $2^8 = 256$
- $2^9 = 512$
- $2^{10} = 1\text{`}024$
- $2^{11} = 2\text{`}048$
- $2^{12} = 4\text{`}096$
- $2^{13} = 8\text{`}192$
- $2^{14} = 16\text{`}384$
- $2^{15} = 32\text{`}768$
- $2^{32} = 4\text{`}294\text{`}967\text{`}296$

Prevody medzi sústavami

- Z binárnej do desiatkovej:
 - Urč hodnotu čísla 10011_2
- Z desiatkovej do binárnej:
 - Reprezentuj hodnotu 47_{10} v binárnej číselnej sústave

Prevody medzi sústavami

- Z binárnej do desiatkovej:
 - Urč hodnotu čísla 10011_2
 - $16 \times 1 + 8 \times 0 + 4 \times 0 + 2 \times 1 + 1 \times 1 = 19_{10}$
- Z desiatkovej do binárnej:
 - Reprezentuj hodnotu 47_{10} v binárnej číselnej sústave
 - $32 \times 1 + 16 \times 0 + 8 \times 1 + 4 \times 1 + 2 \times 1 + 1 \times 1 = 101111_2$

Kapacita číselnej sústavy

- N-ciferné desiatkové číslo
 - Aká je kapacita?
 - Aký je rozsah?
 - Príklad: 3-ciferné desiatkové číslo:
- N-bitové binárne číslo
 - Aká je kapacita?
 - Aký je rozsah? :
 - Príklad: 3-bitové číslo:

Kapacita číselnej sústavy

- N-ciferné desiatkové číslo
 - Aká je kapacita? 10^N
 - Aký je rozsah? $[0, 10^N - 1]$
 - Príklad: 3-ciferné desiatkové číslo:
 - $10^3 = 1000$ possible values
 - Range: $[0, 999]$
- N-bitové binárne číslo
 - Aká je kapacita? 2^N
 - Aký je rozsah? $[0, 2^N - 1]$
 - Príklad: 3-bitové číslo:
 - $2^3 = 8$ possible values
 - Range: $[0, 7] = [000_2 \text{ to } 111_2]$

Hexadecimálne čísla

- Základ sústavy: 16
- Abeceda je tvorená symbolmi:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
A, B, C, D, E , F

Hexadecimálne čísla

Hex symbol	Hodnota	Binárna reprezentácia
0	0	
1	1	
2	2	
3	3	
4	4	
5	5	
6	6	
7	7	
8	8	
9	9	
A	10	
B	11	
C	12	
D	13	
E	14	
F	15	

Hexadecimálne čísla

Hex symbol	Hodnota	Binárna reprezentácia
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Prevody medzi sústavami

- Prevod z hexa. do binárnej sústavy:
 - Preved' $4AF_{16}$ (also written $0x4AF$) do binárnej sústavy
- Prevod z hexa. do desiatkovej sústavy:
 - Preved' $0x4AF$ do desiatkovej sústavy

Prevody medzi sústavami

- Prevod z hexa. do binárnej sústavy:
 - Preved' $4AF_{16}$ (also written $0x4AF$) do binárnej sústavy
 - $0100\ 1010\ 1111_2$
- Prevod z hexa. do desiatkovej sústavy:
 - Preved' $0x4AF$ do desiatkovej sústavy
 - $16^2 \times 4 + 16^1 \times 10 + 16^0 \times 15 = 1199_{10}$

Prevody v pozičných ČS

- Uskutočnite prevod čísla $N = (2E5,A)_{16}$ vyjadreného v šesťnáškovej číselnej sústave do desiatkovej sústavy.

$$(N)_{16} \equiv (2E5, A)_H = 2 \times 16^2 + 14 \times 16^1 + 5 \times 10^0 + 10 \times 16^{-1} = (741,625)_{10}$$

- Uskutočnite prevod osmičkového čísla $N = (1074,46)_8$ do desiatkovej číselnej sústavy.

1074	1	0	7	4
	8	64	568	
$\times 8$	8	71	572	

46	6	4	0
	0,75	0,59375	
: 8	4,75	0,59375	

výsledok hľadaného prevodu je

$$(1074,46)_8 = (572,59375)_{10}$$

Bity, Bajty a „Nibbles“...

- Bits

10010110

most significant bit least significant bit

- Bytes & Nibbles

byte
10010110
nibble

- Bytes

CEBF9AD7

most significant byte least significant byte

Informačné jednotky

- $2^{10} = 1 \text{ kilo}$ $\approx 1000 \text{ (1024)}$
- $2^{20} = 1 \text{ mega}$ $\approx 1 \text{ million (1,048,576)}$
- $2^{30} = 1 \text{ giga}$ $\approx 1 \text{ billion (1,073,741,824)}$

Mocniny čísla dva

- Určte presnú a približnú hodnotu čísla 2^{24}
- Koľko rôznych hodnôt dokážeme reprezentovať pomocou 32-bitového čísla?

Mocniny čísla dva

- Určte presnú a približnú hodnotu čísla 2^{24}
 $2^4 \times 2^{20} \approx 16 \text{ million}$
- Koľko rôznych hodnôt dokážeme reprezentovať pomocou 32-bitového čísla?
 $2^2 \times 2^{30} \approx 4 \text{ billion}$

Aritmetika v binárnej ČS

Sčítanie: $\begin{array}{r} 1 \ 1 \ 1 \\ + 1 \ 0 \ 1 \ 1 \ 0 \ 1 , \ 0 \ 1 \\ \hline 1 \ 1 \ 1 \ 1 \ 0 , \ 1 \ 0 \\ 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 | , \ 1 \ 1 \end{array}$ - prenosy pri operácii sčítania

Odčítanie: $\begin{array}{r} 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \\ - 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \\ \hline 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \end{array}$ - výpožičky jedničiek z predchádzajúcich rádov

Aritmetika v binárnej ČS

- Sčítaj nasledujúce dve 4-bitové čísla

$$\begin{array}{r} 1001 \\ + 0101 \\ \hline \end{array}$$

- Sčítaj nasledujúce dve 4-bitové čísla

$$\begin{array}{r} 1011 \\ + 0110 \\ \hline \end{array}$$

Aritmetika v binárnej ČS

- Sčítaj nasledujúce dve 4-bitové čísla

$$\begin{array}{r} & & 1 \\ & 1001 \\ + & 0101 \\ \hline 1110 \end{array}$$

- Sčítaj nasledujúce dve 4-bitové čísla

$$\begin{array}{r} 111 \\ 1011 \\ + 0110 \\ \hline 10001 \end{array}$$

Pretečenie!

Pretečenie

- Číslicové systémy na reprezentáciu číselných hodnôt (vstupných operandov) používajú N-bitov
- **Pretečenie:** ak hodnotu výsledku operácie nie je možné reprezentovať na N-bitoch;
- Pozri predchádzajúci príklad $11 + 6$

Reprezentácia celých čísel

- Priamy kód (PK)
- Inverzný kód (IK)
- Doplňkový kód (DK)
- Kód s posunutou nulou (KPN)

Priamy kód

$$(N^P)_B = N_{sign}^P \underbrace{N_{n-1}^P \dots N_0^P}_{|(N^P)_B|} = N_{sign}^P (N_{mod}^P)_B$$

kde

$$N_{sign}^P = \begin{cases} 0 \Leftarrow [N] \geq [0] \\ 1 \Leftarrow [N] < [0] \end{cases}$$

Priamy kód

- 1 znamienkový bit, $N-1$ bit pre magnitúdu
- Najľavejší bit je znamienkový bit
 - Kladné čísla: z.bit = 0
 - Záporné čísla: z.bit = 1
- Reprezentuje ± 6 v priamom kóde:
 - +6 =
 - 6 =
- N -bitový priamy kód pokrýva interval:

$$A : \{a_{N-1}, a_{N-2}, \dots, a_2, a_1, a_0\}$$

$$A = (-1)^{a_{n-1}} \sum_{i=0}^{n-2} a_i 2^i$$

Priamy kód

- 1 znamienkový bit, $N-1$ bit pre magnitúdu
- Najľavejší bit je znamienkový bit
 - Kladné čísla: z.bit = 0 $A : \{a_{N-1}, a_{N-2}, \dots, a_2, a_1, a_0\}$
 - Záporné čísla: z.bit = 1 $A = (-1)^{a_{n-1}} \sum_{i=0}^{n-2} a_i 2^i$
- Reprezentuje ± 6 v priamom kóde:
 - +6 = **0110**
 - 6 = **1110**
- N -bitový priamy kód pokrýva interval:
 $[-(2^{N-1}-1), 2^{N-1}-1]$

Priamy kód

- Nevýhoda:
 - Sčítanie nemusí „fungovať“, napr. $-6 + 6$:

$$\begin{array}{r} 1110 \\ + 0110 \\ \hline 10100 \end{array}$$

10100 (nesprávny výsledok!)

- Dve „verzie“ nuly (± 0):

1000 (záporná)
0000 (kladná)

Doplňkový kód

$$(N^D)_B = N_{sign}^D N_{n-1}^D N_{n-2}^D \dots N_0^D$$

kde

$$N_{sign}^D = \begin{cases} 0 \Leftrightarrow [N] \geq [0] \\ 1 \Leftrightarrow [N] < [0] \end{cases}$$

$$[(N^D)_B] = \begin{cases} |N| & \Leftrightarrow [N] \geq [0] \\ 2^n - |N| & \Leftrightarrow [N] < [0] \end{cases}$$

Doplnkový kód

- Netrpí nedostatkami priameho kódu
 - Sčítanie funguje
 - Existuje len jedna reprezentácia nuly

Doplňkový kód

$$A = a_{n-1} \left(-2^{n-1} \right) + \sum_{i=0}^{n-2} a_i 2^i$$

- Najväčšie 4-bitové číslo:
- Najmenšie 4-bitové číslo:
- Najvýznamnejší bit reprezentuje z.bit
(0 = kladné, 1 = záporné)
- N -bitový doplnkový kód pokrýva interval:

Doplňkový kód

$$A = a_{n-1} \left(-2^{n-1} \right) + \sum_{i=0}^{n-2} a_i 2^i$$

- Najväčšie 4-bitové číslo: **0111**
- Najmenšie 4-bitové číslo: **1000**
- Najvýznamnejší bit reprezentuje z.bit
(0 = kladné, 1 = záporné)
- N -bitový doplnkový kód pokrýva interval:

$$[-(2^{N-1}), 2^{N-1}-1]$$

Doplnkový kód

- Reprezentujte $6_{10} = 0110_2$ v DK
- Určte hodnotu čísla reprezentovaného zápisom $1001_{2\text{DK}}$?

Doplňkový kód

- Reprezentujte $6_{10} = 0110_2$ v DK
 - 1. 1001
 - 2. + 1 $1010_2 = -6_{10}$
- Určte hodnotu čísla reprezentovaného zápisom $1001_{2\text{DK}}$?
 - 1. 110
 - 2. + 1 $111_2 = 7_{10}$, so $1001_2 = -7_{10}$

Doplnkový kód: sčítanie

- Sčítaj čísla $6 + (-6)$ reprezentované v DK

$$\begin{array}{r} 0110 \\ + 1010 \\ \hline \end{array}$$

- Sčítaj čísla $-2 + 3$ reprezentované v DK

$$\begin{array}{r} 1110 \\ + 0011 \\ \hline \end{array}$$

Doplňkový kód: sčítanie

- Sčítaj čísla $6 + (-6)$ reprezentované v DK

$$\begin{array}{r} \textcolor{blue}{111} \\ 0110 \\ + 1010 \\ \hline 10000 \end{array}$$

- Sčítaj čísla $-2 + 3$ reprezentované v DK

$$\begin{array}{r} \textcolor{blue}{111} \\ 1110 \\ + 0011 \\ \hline 10001 \end{array}$$

Inverzný kód

$$(N^I)_B = N_{sign}^I N_{n-1}^I N_{n-2}^I \dots N_0^I$$

kde

$$N_{sign}^I = \begin{cases} 0 \Leftrightarrow [N] \geq [0] \\ 1 \Leftrightarrow [N] < [0] \end{cases}$$

$$[(N^I)_B] = \begin{cases} |N| & \Leftrightarrow [N] \geq [0] \\ 2^n - 1 - |N| & \Leftrightarrow [N] < [0] \end{cases}$$

Kód s posunutou nulou

$$(N^M)_B = N_{sign}^M N_{n-1}^D N_{n-2}^D \dots N_0^D$$

kde

$$N_{sign}^M = \begin{cases} 1 \Leftrightarrow [N] \geq [0] \\ 0 \Leftrightarrow [N] < [0] \end{cases}$$

$$[(N^M)_B] = \begin{cases} 2^n + |N| \quad \Leftrightarrow \quad [N] \geq [0] \\ 2^n - |N| \quad \Leftrightarrow \quad [N] < [0] \end{cases}$$

Zmena dĺžky reprezentácie

- Reprezentuj N -bitové číslo pomocou M -bitov ($M > N$) :
 - Kopírovanie hodnoty z.bitu
 - Pridávanie núl sprava

Kopírovanie z.bitu

- **Príklad 1:**
 - 4-bitová reprezentácia hodnoty 3 = **0011**
 - 8-bitová reprezentácia hodnoty 3: **00000011**
- **Príklad 2:**
 - 4-bitová reprezentácia hodnoty -5 = **1011**
 - 8-bitová reprezentácia hodnoty: **11111011**

Pridávanie núl

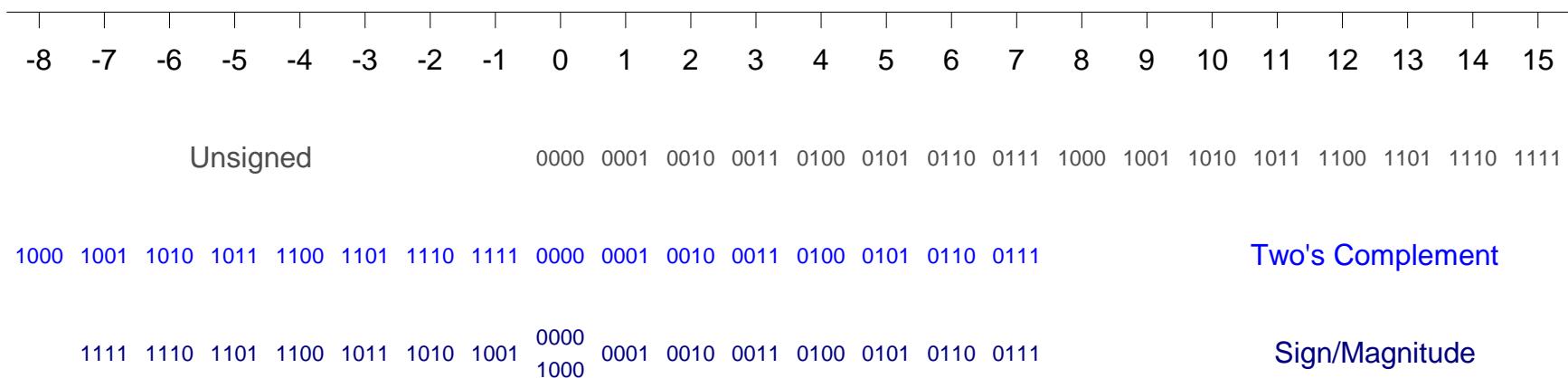
- **Príklad 1:**
 - 4-bitová hodnota = $0011_2 = 3_{10}$
 - 8-bitová hodnota: $00000011 = 3_{10}$
- **Príklad 2:**
 - 4-bitová hodnota = $1011 = -5_{10}$
 - 8-bitová hodnota: $00001011 = 11_{10}$

Pozor na záporné čísla

Zhrnutie

Číselná sústava	Interval zobr.
Pozičná	$[0, 2^N-1]$
Priamy kód	$[-(2^{N-1}-1), 2^{N-1}-1]$
Doplnkový kód	$[-2^{N-1}, 2^{N-1}-1]$

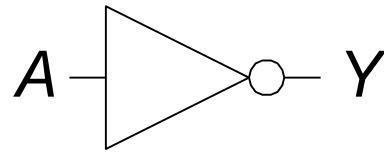
4-bitová reprezentácia:



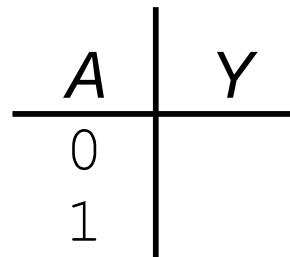
- **Realizujú logické funkcie:**
 - invertovanie (NOT), AND, OR, NAND, NOR, etc.
- **Jedno-vstupové:**
 - NOT gate, buffer
- **Dvoj-vstupové:**
 - AND, OR, XOR, NAND, NOR, XNOR
- **Viac-vstupové**

1-vstupové hradlá

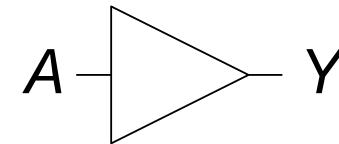
NOT



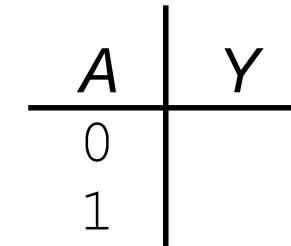
$$Y = \overline{A}$$



BUF

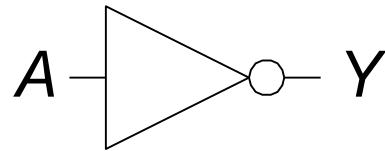


$$Y = A$$



1-vstupové hradlá

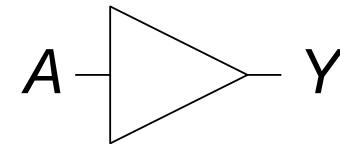
NOT



$$Y = \overline{A}$$

A	Y
0	1
1	0

BUF

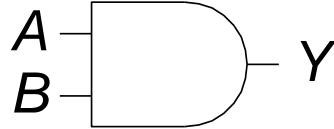


$$Y = A$$

A	Y
0	0
1	1

2-vstupové hradlá

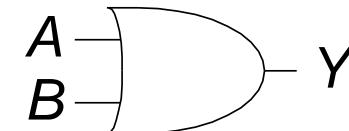
AND



$$Y = AB$$

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

OR

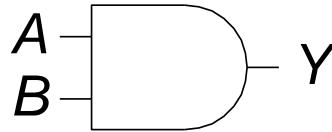


$$Y = A + B$$

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

2-vstupové hradlá

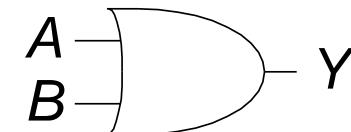
AND



$$Y = AB$$

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

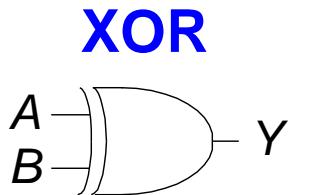
OR



$$Y = A + B$$

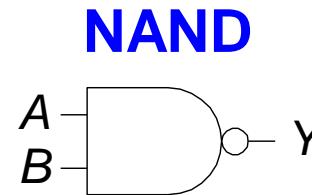
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

2-vstupové hradlá (odvodené)



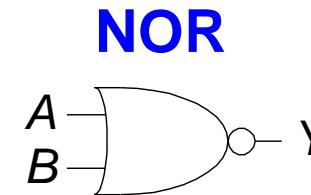
$$Y = A \oplus B$$

A	B	Y
0	0	
0	1	
1	0	
1	1	



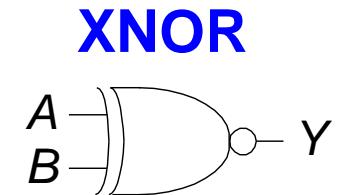
$$Y = \overline{AB}$$

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



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

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

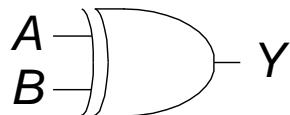


$$Y = \overline{A \oplus B}$$

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

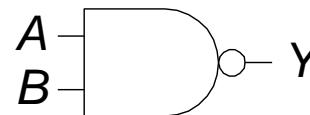
2-vstupové hradlá (odvodené)

XOR



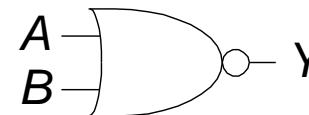
$$Y = A \oplus B$$

NAND



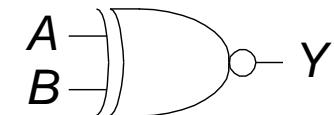
$$Y = \overline{AB}$$

NOR



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

XNOR



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

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

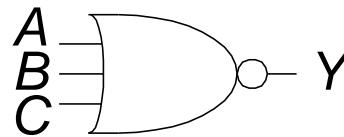
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

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

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

Viacvstupové hradlá

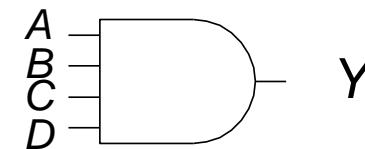
NOR3



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

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

AND4

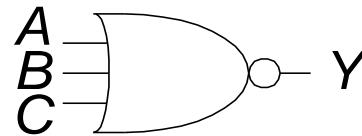


$$Y = ABCD$$

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

Viacvstupové hradlá

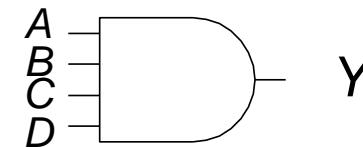
NOR3



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

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

AND4



$$Y = ABCD$$

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

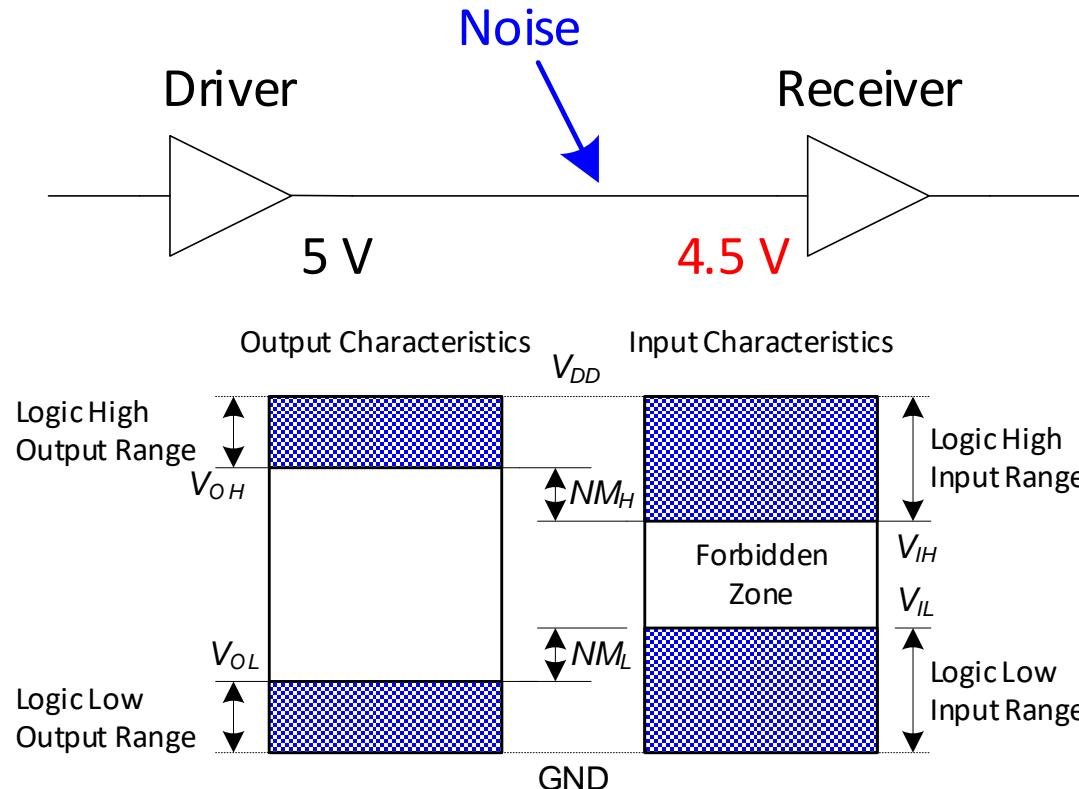
- Multi-input XOR: Odd parity

Logické úrovne

- Číslicové signály majú konečný počet diskrétnych hodnôt, väčšinou dve; 1 a 0
- Príklad požitia týchto hodnôt:
 - 0 = zem (GND) alebo 0 volt
 - 1 = napájanie (V_{DD}) alebo 5 volt
- Ako reprezentovať 4,99V? Je to 0 alebo 1?
- A čo v prípade 3,2V?

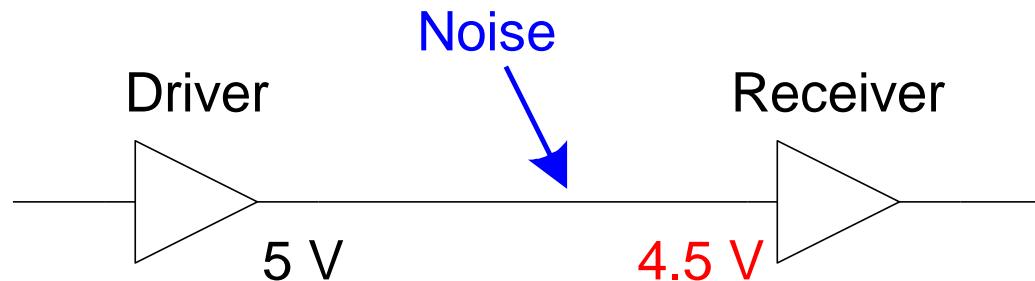
Logické úrovne

- Napäťové pásmo pre 1 and 0
- Rozdielne pásmo pre vstup a výstup

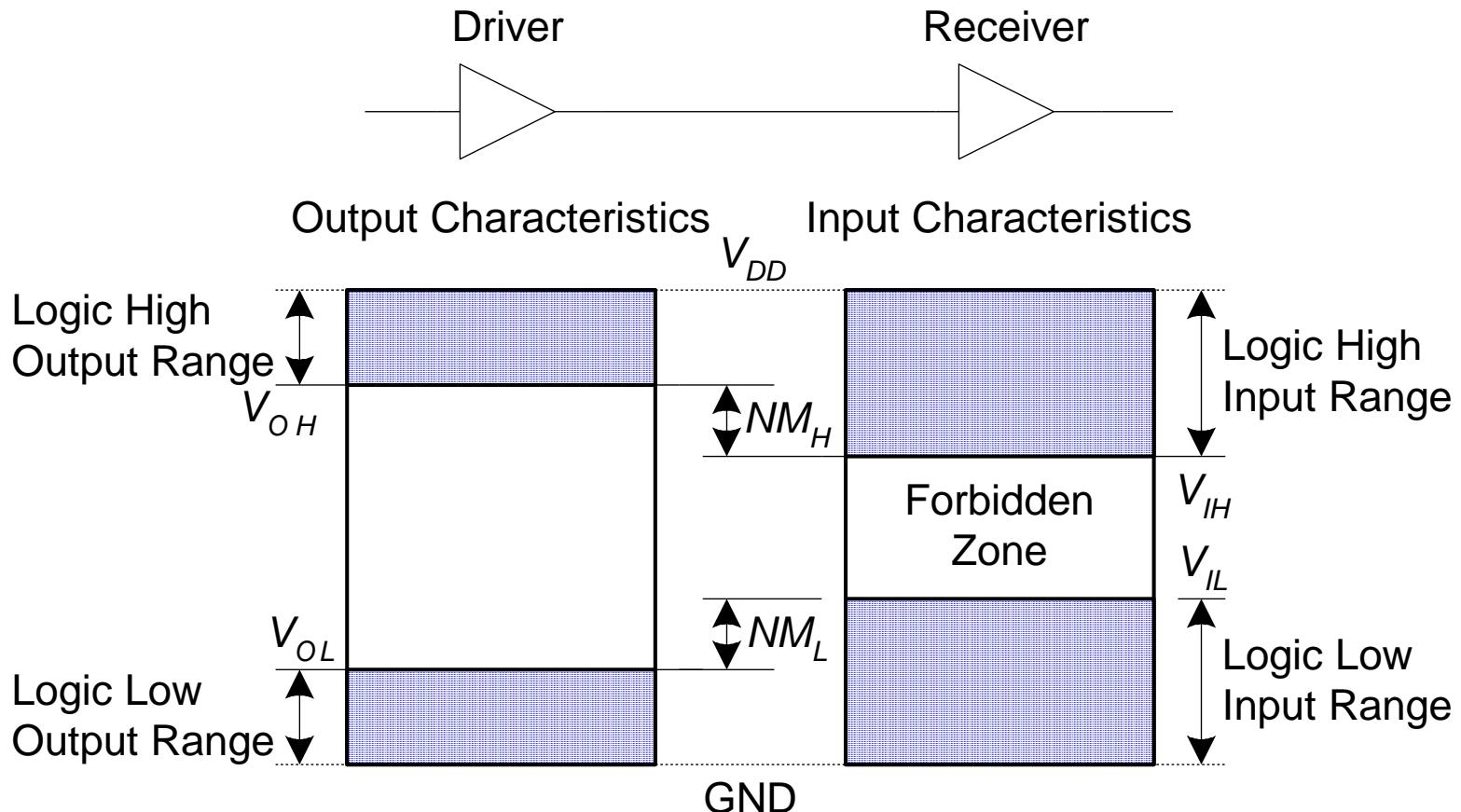


Čo je šum?

- **Všetko čo degraduje kvalitu signálu**
 - Napr., odpor vodiča, šum od zdroja, coupling to neighboring wires, etc.
- **Napr.: vysielač na svojom výstupe produkuje 5V, ale kvôli dlhému vedeniu a odporu vedenia prijímač „vidí“ len 4.5 V**



Napäťová rezerva



$$NM_H = V_{OH} - V_{IH}$$

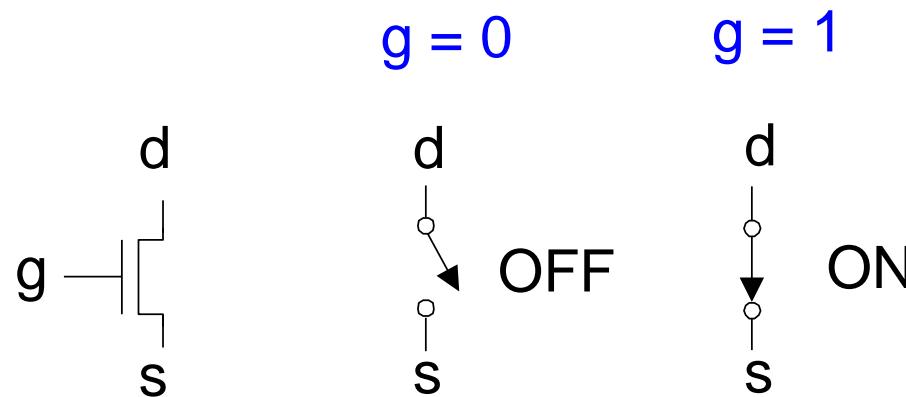
$$NM_L = V_{IL} - V_{OL}$$

Napäťové úrovne

Logic Family	V_{DD}	V_{IL}	V_{IH}	V_{OL}	V_{OH}
TTL	5 (4.75 - 5.25)	0.8	2.0	0.4	2.4
CMOS	5 (4.5 - 6)	1.35	3.15	0.33	3.84
LV TTL	3.3 (3 - 3.6)	0.8	2.0	0.4	2.4
LV CMOS	3.3 (3 - 3.6)	0.9	1.8	0.36	2.7

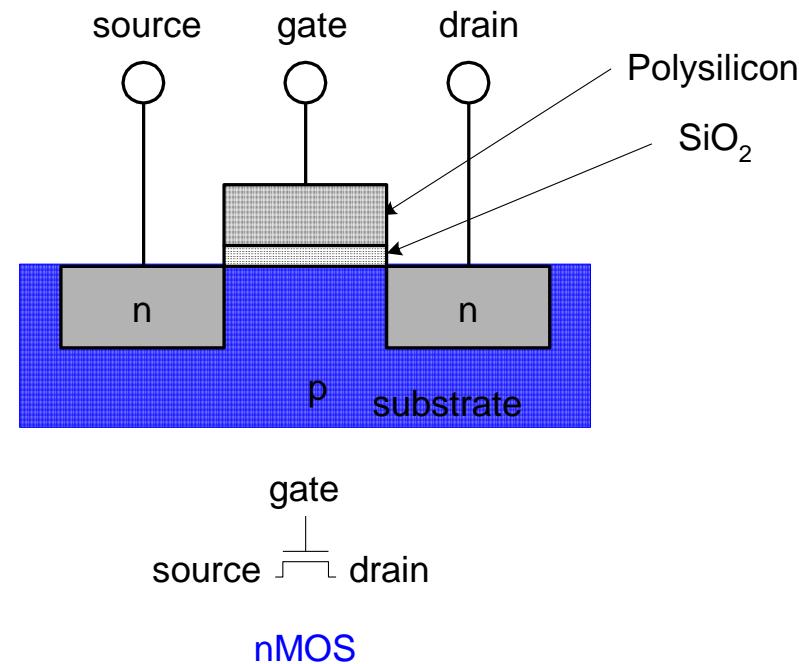
Tranzistory

- Základným stavebným prvkom hradieb
- Má 3-elektródy
 - bipolárne: C, B, E; unipolárne: D, G, S
 - d a s sú spojené (ON) ak g je 1



MOS Tranzistor

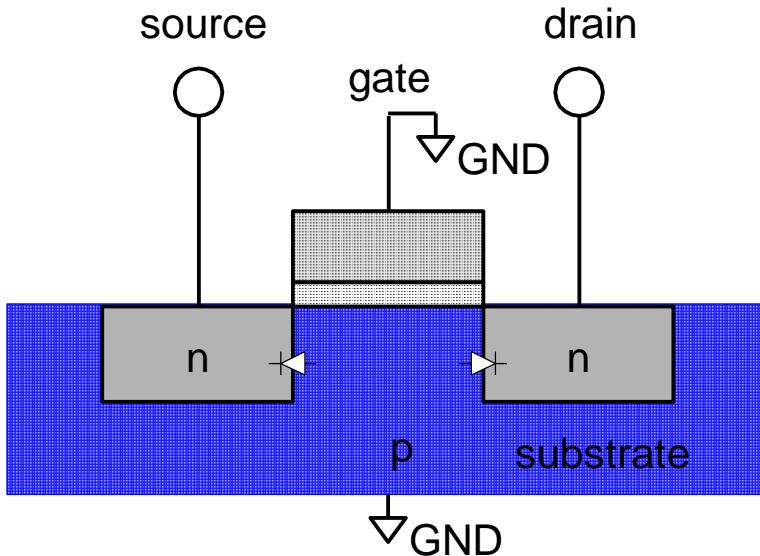
- **Metal oxide silicon (MOS) tranzistor:**
 - Najrozšírenejší druh tranzistorov riadených elektrickým poľom
 - Vodivosť kanálu medzi *s* a *d* je riadená nepäťom privedeným na *g*



nMOS

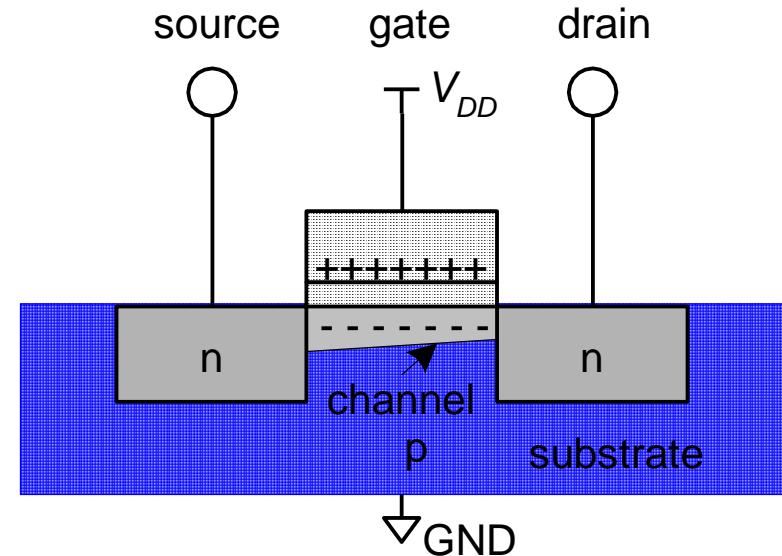
Gate = 0

OFF (nebol vytvorený vodivý kanál medzi s a d)



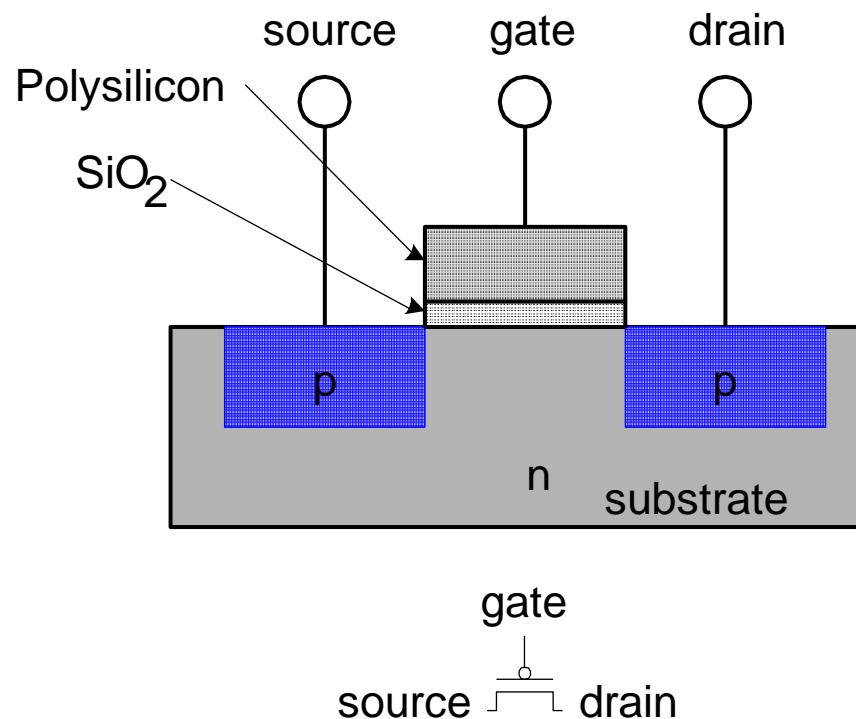
Gate = 1

ON (existuje vodivý kanál medzi s a d)



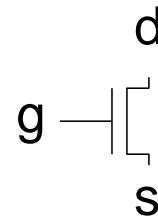
pMOS

- pMOS tranzistor je komplementom k nMOS
 - je ON ak Gate = 0
 - je OFF ak Gate = 1

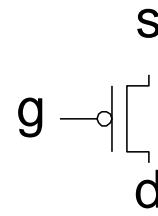


Funkcia tranzistorov MOS

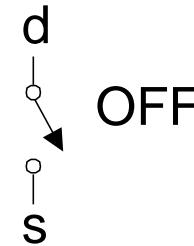
nMOS



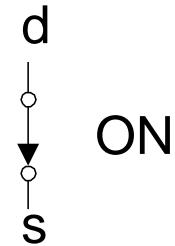
pMOS



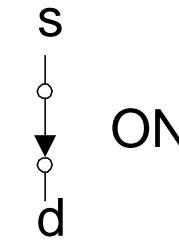
$g = 0$



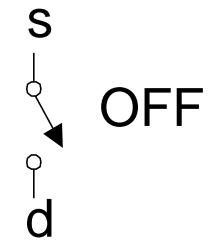
$g = 1$



ON

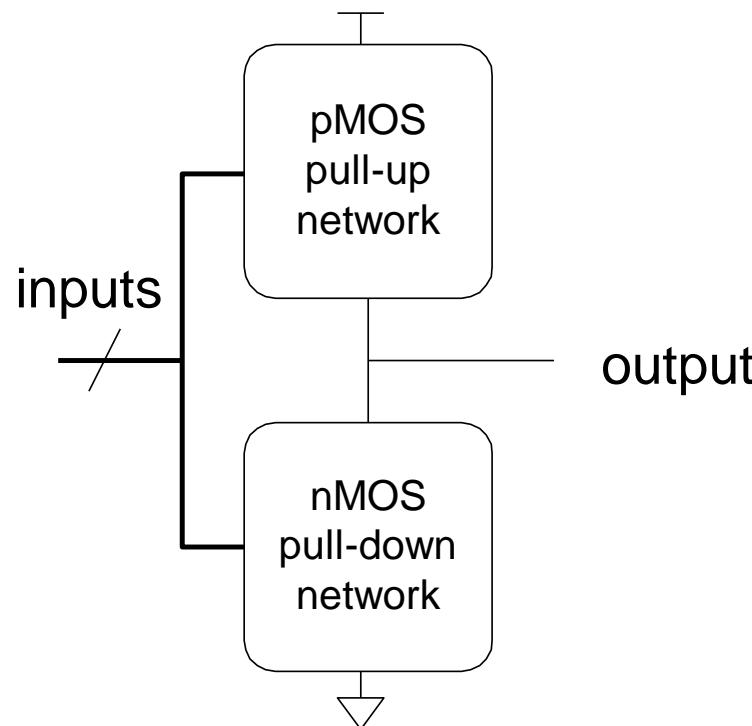


OFF

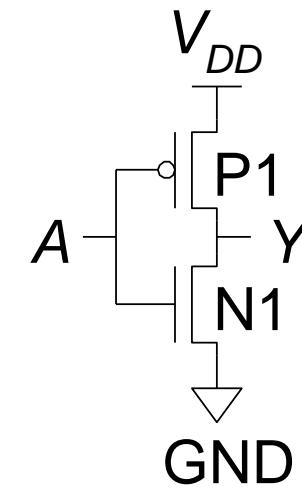
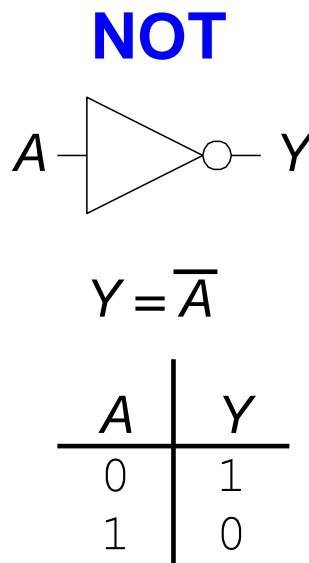


Funkcia tranzistorov MOS

- **nMOS:** dobre prenáša „nuly“
- **pMOS:** dobre prenáša „jednotky“

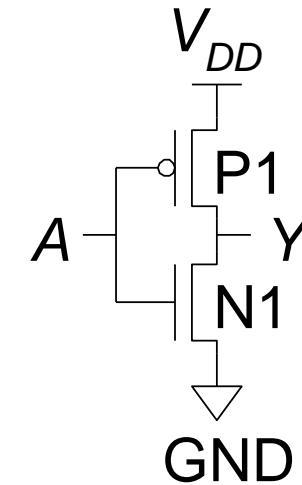
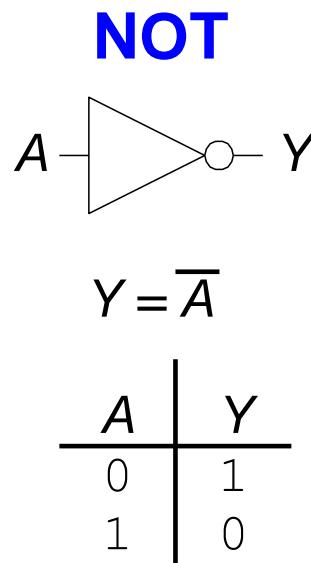


CMOS hradlá: Hradlo NOT



A	P1	N1	Y
0			
1			

CMOS hradlá: Hradlo NOT



A	P1	N1	Y
0	ON	OFF	1
1	OFF	ON	0

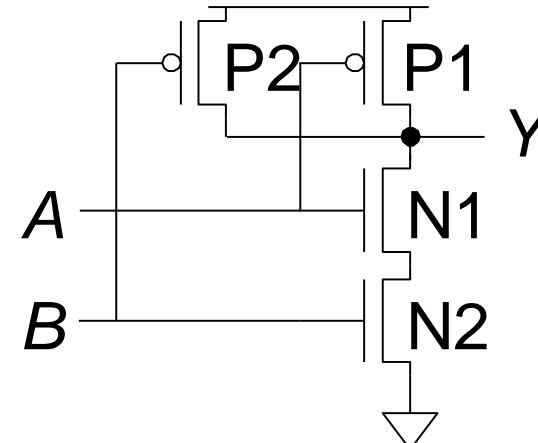
CMOS hradlá: Hradlo NAND

NAND



$$Y = \overline{AB}$$

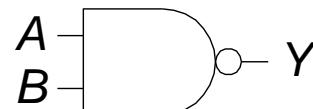
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0



A	B	P1	P2	N1	N2	Y
0	0					
0	1					
1	0					
1	1					

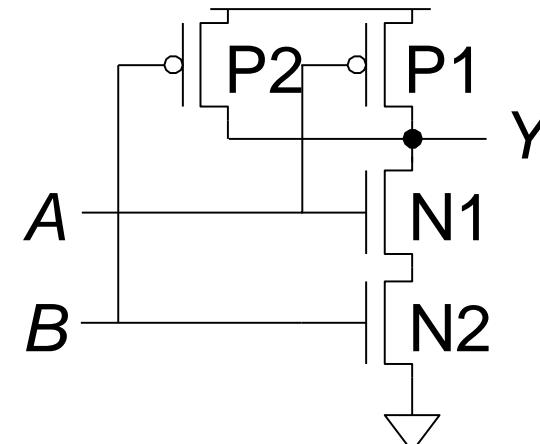
CMOS hradlá: Hradlo NAND

NAND



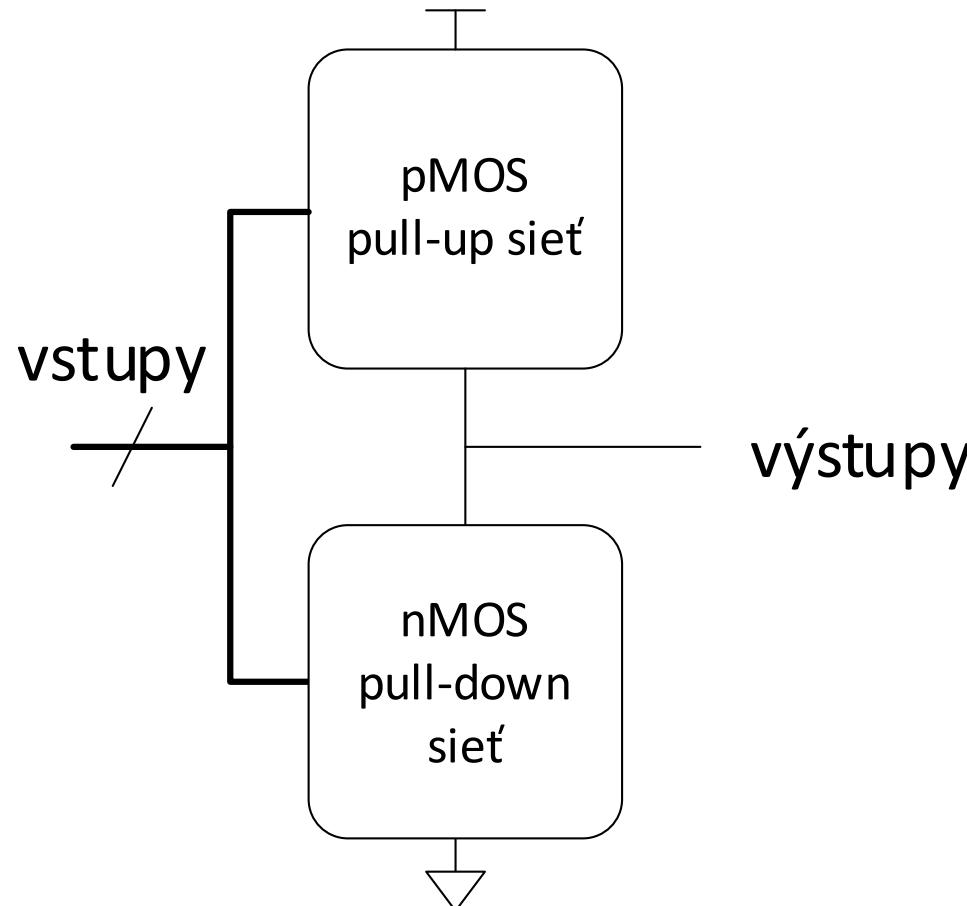
$$Y = \overline{AB}$$

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



A	B	P1	P2	N1	N2	Y
0	0	ON	ON	OFF	OFF	1
0	1	ON	OFF	OFF	ON	1
1	0	OFF	ON	ON	OFF	1
1	1	OFF	OFF	ON	ON	0

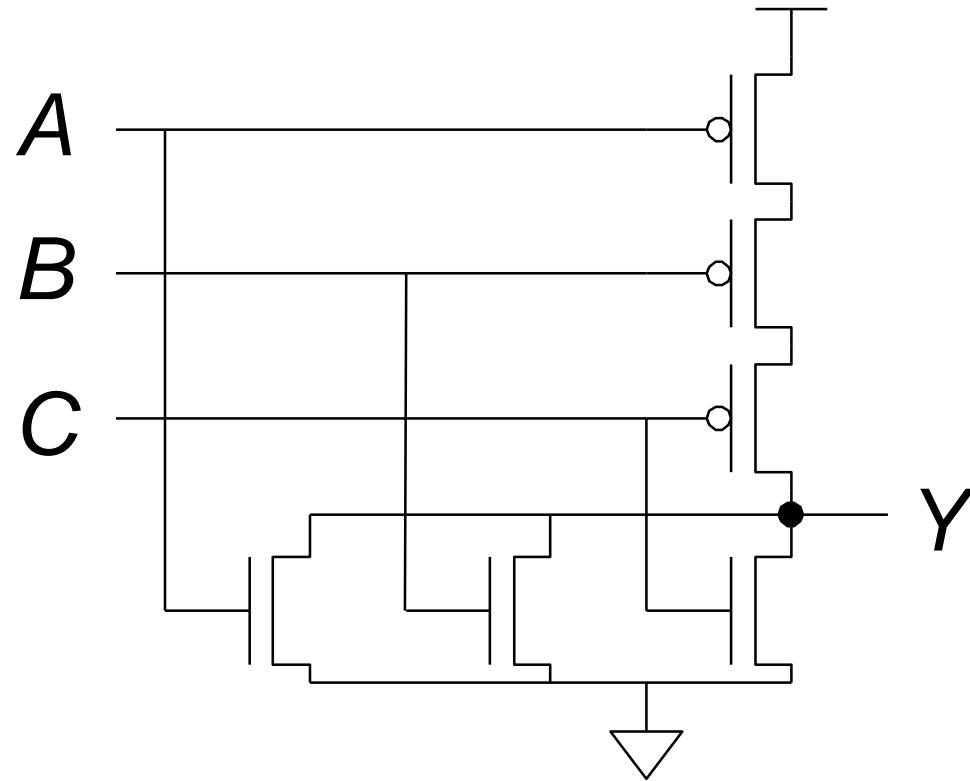
Stavba CMOS hradiel



Hradlo NOR

Vytvorte 3-vstupové hradlo NOR na báze MOS tranzistorov.

Hradlo NOR3

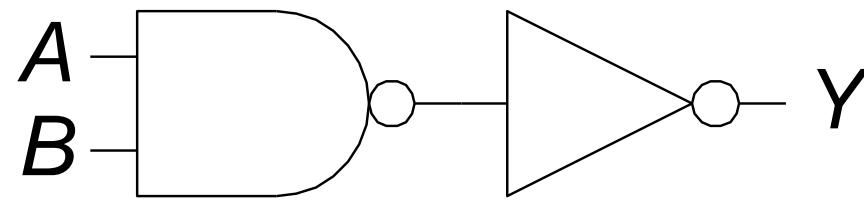


Hradlo AND2

Vytvorte 2-vstupové AND hradlo na báze MOS tranzistorov.

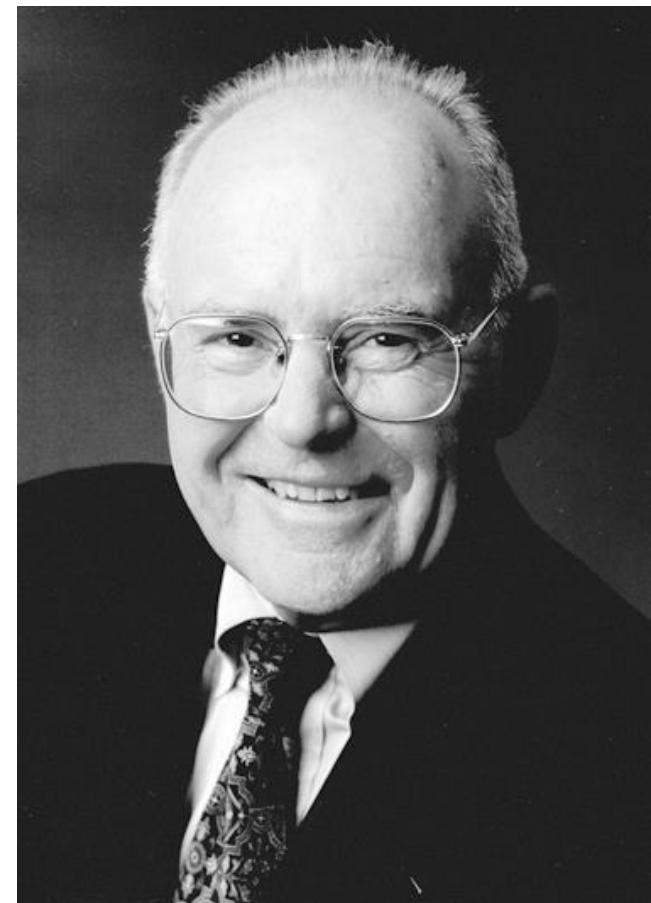
Hradlo AND2

Vytvorte 2-vstupové AND hradlo na báze MOS tranzistorov.

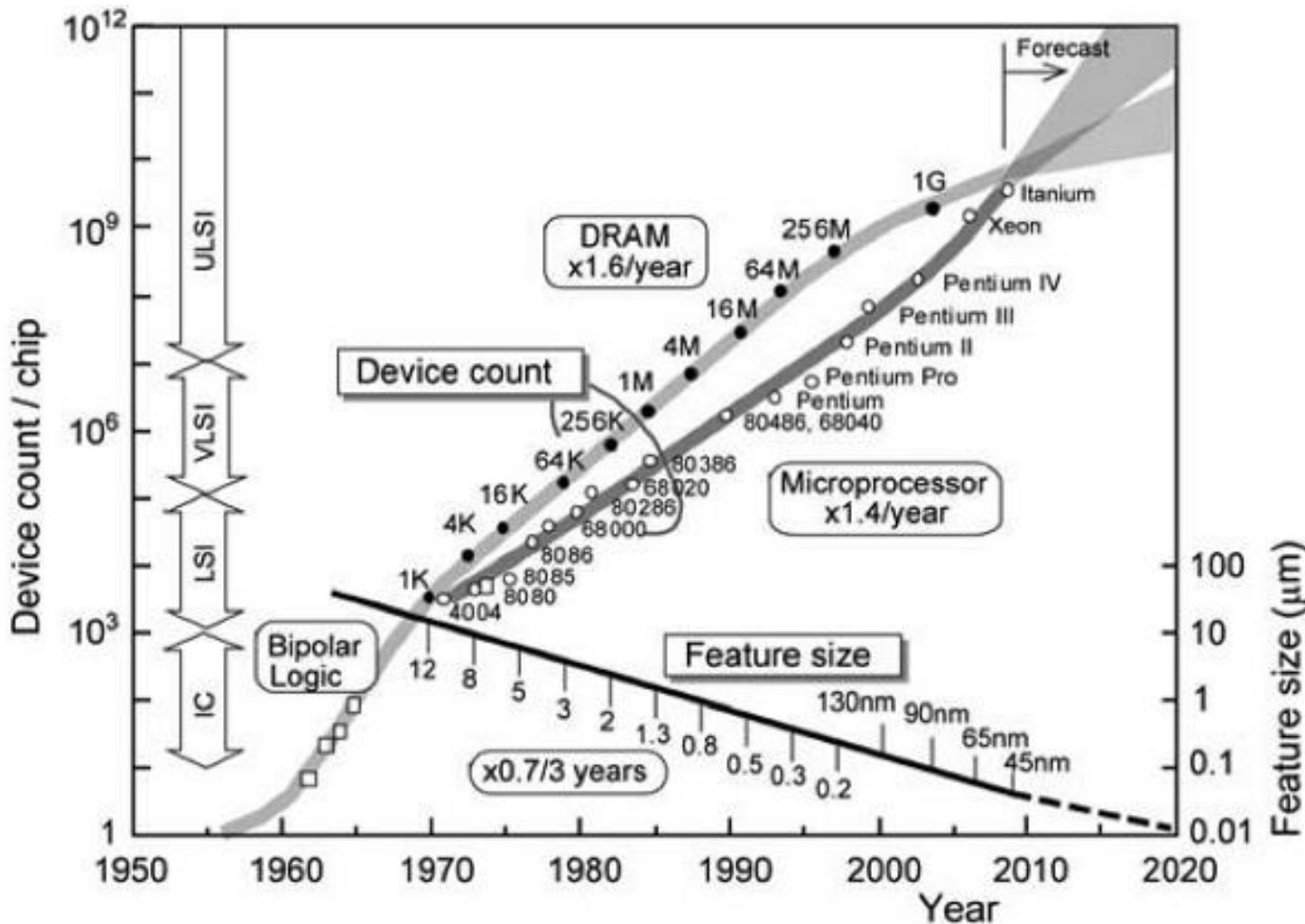


Gordon Moore, 1929-

- Spoluzakladateľ firmy Intel.
- **Moore-ov zákon:**
Počet tranzistorov sa zdvojnásobí každý „rok“
(empirické pravidlo, 1965)
Počet tranzistorov sa zdvojnásobi cca. každé 2 roky.



Mooreov zákon



Referencia

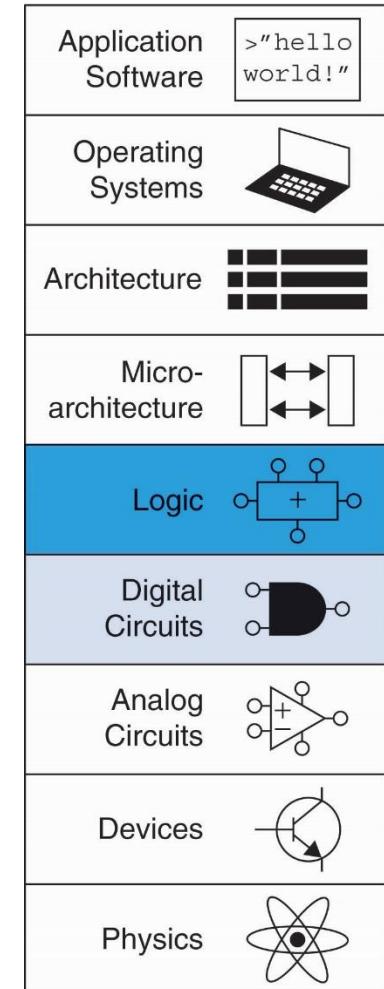
- David Money Harris and Sarah L. Harris,
Digital Design and Computer Architecture,
Chapter 1: From Zero to One, Second Edition
© 2012 by Elsevier Inc.

Architektúry počítačových systémov

2. rok ZS

Osnova

- **Úvod**
- **Booleova algebra**
- **Booleovské rovnice**
- **Kombinačná logika**
- **Karnaughove mapy**
- **Kombinačné logické obvody**
- **Časové charakteristiky**

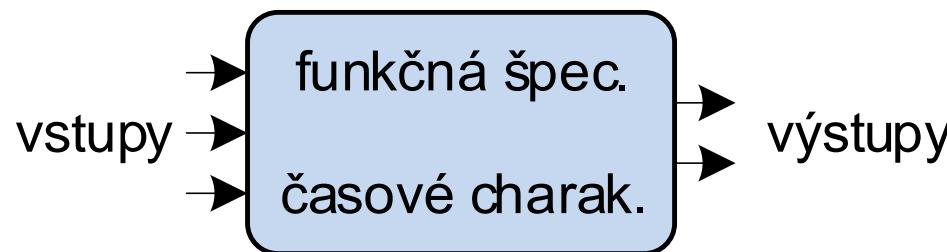


Logický systém je charakterizovaný

- abstraktným opisom funkcie systému,
- **štruktúra systému** → *logický obvod*

Logický obvod (LO) je definovaný pomocou:

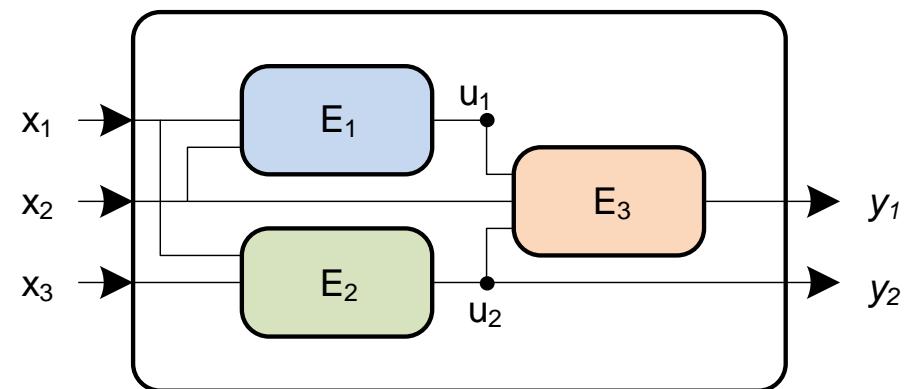
- svojich vstupov (X) a výstupov (Y),
- funkčnej špecifikácie a
- časových charakteristík.



Logické obvody

- Štruktúru logického obvodu tvoria logické členy a vzájomné väzby medzi nimi. Väzby medzi členmi v štruktúre kombinačného obvodu sa uskutočňujú v uzloch.
- Uzly (spojenie dvoch alebo viacerých kanálov)
 - Vstupy: x_1, x_2, x_3
 - Výstupy: y_1, y_2
 - Vnútorné uzly: u_1, u_2
- Logické členy
 - E_1, E_2, E_3

$$LO = (U, E)$$

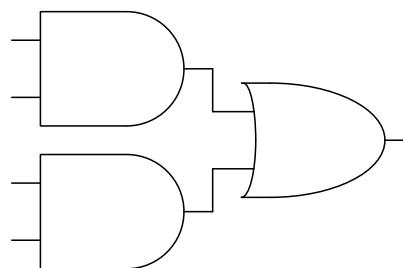


Typy logických obvodov

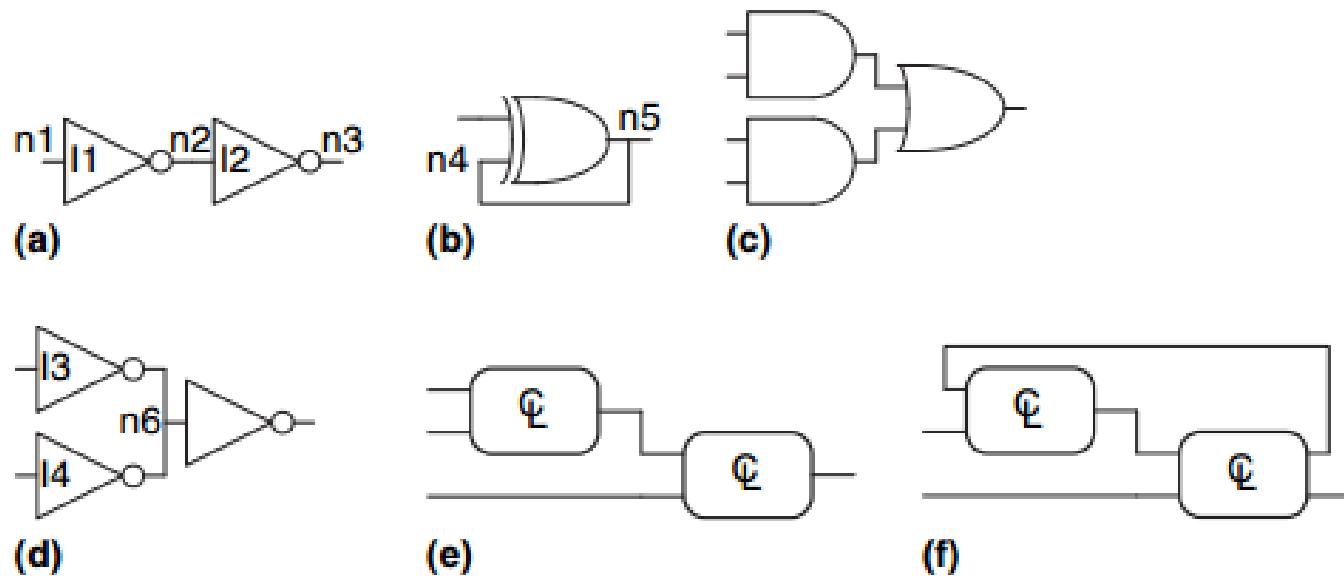
- **Kombinačný logický obvod (KLO)**
 - Výstup kombinačného obvodu závisí len od skutočnej okamžitej kombinácie vstupných hodnôt.
- **Sekvenčný logický obvod (SLO)**
 - Sú logické obvody, ktorých výstup závisí od kombinácie vstupov a od vnútorných stavov obvodu z predchádzajúceho taktu.

Kombinačný log. obvod

- Každý element obvodu je kombinačný obvod.
- Každý vstup je spojený s výstupom len z jedného elementu obvodu.
- Obvod neobsahuje slučky.
- **Príklad:**



Kombinačný log. obvod

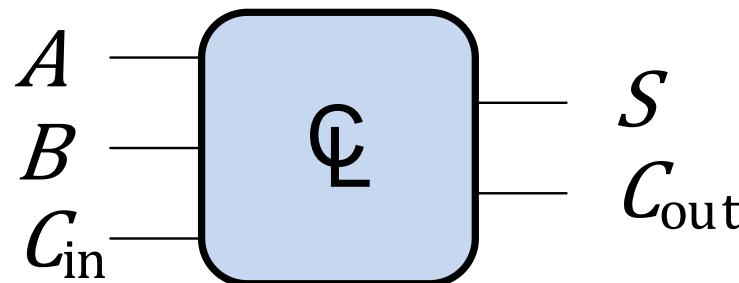


- a) je KLO
- b) nie je KLO
- c) je KLO

- d) nie je KLO
- e) je KLO
- f) nie je KLO

Booleovské rovnice

- Definujú vzťah medzi vstupnými a výstupnými b. premennými.
- Príklad:
$$S = F(A, B, Cin)$$
$$C_{out} = F(A, B, Cin)$$



$$S = A \otimes B \otimes C$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Definície

- Komplement: negovaná b. premenná

$\bar{A}, \bar{B}, \bar{C}$

- Literál: b. premenná v priamej alebo v negovanej forme

$A, \bar{A}, B, \bar{B}, C, \bar{C}$

Definície

- Implikant: súčin všetkých vstupných b. premenných v priamej alebo negovanej forme

$ABC, A\bar{B}C, A\bar{B}\bar{C}$

- Implicit: súčet všetkých vstupných b. premenných v priamej alebo negovanej forme

$(A + B + C), (A + \bar{B} + C), (A + \bar{B} + \bar{C})$

Disjunktívna normálna forma

- Každá b. funkcia môže byť vyjadrená v DNF
- DNF je súčet implikantov
- Implikant je súčin literálov
- Každý implikant definuje pravdivostnú hodnotu TRUE
 - Pokrýva príslušný riadok pravdivostnej tabuľky

A	B	Y	implikant	označenie implikantu
0	0	0	$\bar{A}\bar{B}$	m_0
0	1	1	$\bar{A}B$	m_1
1	0	0	$A\bar{B}$	m_2
1	1	1	AB	m_3

$$Y = F(A, B) =$$

Disjunktívna normálna forma

- Každá b. funkcia môže byť vyjadrená v DNF
- DNF je súčet implikantov
- Implikant je súčin literálov
- Každý implikant definuje pravdivostnú hodnotu TRUE
 - Pokrýva príslušný riadok pravdivostnej tabuľky

A	B	Y	implikant	označenie implikantu
0	0	0	$\bar{A}\bar{B}$	m_0
0	1	1	$\bar{A}B$	m_1
1	0	0	$A\bar{B}$	m_2
1	1	1	AB	m_3

$$Y = F(A, B) =$$

Disjunktívna normálna forma

- Každá b. funkcia môže byť vyjadrená v DNF
- DNF je súčet implikantov
- Implikant je súčin literálov
- Každý implikant definuje pravdivostnú hodnotu TRUE
 - Pokrýva príslušný riadok pravdivostnej tabuľky

A	B	Y	implikant	označenie implikantu
0	0	0	$\bar{A}\bar{B}$	m_0
0	1	1	$\bar{A}B$	m_1
1	0	0	$A\bar{B}$	m_2
1	1	1	AB	m_3

$$Y = F(A, B) = \bar{A}B + AB = \Sigma(1, 3)$$

Konjunktívna normálna forma

- Každá b. funkcia môže byť vyjadrená v KNF
- KNF je súčin implicantov
- Implicant je súčet literálov
- Každý implicant definuje pravdivostnú hodnotu FALSE
 - Pokrýva príslušný riadok pravdivostnej tabuľky

A	B	Y	implicant	označenie implicantu
0	0	0	$A + B$	M_0
0	1	1	$A + \bar{B}$	M_1
1	0	0	$\bar{A} + B$	M_2
1	1	1	$\bar{A} + \bar{B}$	M_3

$$Y = F(A, B) = (A + B) \cdot (\bar{A} + B) = \Pi(0, 2)$$

Booleova algebra

- Aplikovaním axióm a teorém na b. rovnicu je možné danú rovnicu zjednodušiť
- Premenné sú dvojhodnotové
- **Princíp duality :**
 - AND a OR, 0 a 1 sú „zameniteľné“

Axiómy B-algebry

	Axióma		Duálna axióma	Názov
A1	$B = 0 \text{ if } B \neq 1$	A1'	$B = 1 \text{ if } B \neq 0$	Binárny element
A2	$\bar{0} = 1$	A2'	$\bar{1} = 0$	Negácia
A3	$0 \cdot 0 = 0$	A3'	$1 + 1 = 1$	súčin/súčet
A4	$1 \cdot 1 = 1$	A4'	$0 + 0 = 0$	súčin/súčet
A5	$0 \cdot 1 = 1 \cdot 0 = 0$	A5'	$1 + 0 = 0 + 1 = 1$	súčin/súčet

Teorémy B-algebry

	Teoréma		Duálna teoréma	Názov
T1	$B \cdot 1 = B$	T1'	$B + 0 = B$	Identita / neutrálnosť konštanty
T2	$B \cdot 0 = 0$	T2'	$B + 1 = 0$	Nulitnosť / dominancia
T3	$B \cdot B = B$	T3'	$B + B = B$	Idempotentnosť
T4		$\bar{\bar{B}} = B$		Involutívnosť / dvojitá negácia
T5	$B \cdot \bar{B} = 0$	T5'	$B + \bar{B} = 1$	Komplement / vylúčenie tretieho / zákon sporu

Teóremy B-algebry viac. premenných

	Teórema		Duálna teórema	Názov
T6	$B \cdot C = C \cdot B$	T6'	$B + C = C + B$	Komutatívnosť
T7	$(B \cdot C) \cdot D = B \cdot (C \cdot D)$	T7'	$(B + C) + D = B + (C + D)$	Asociatívnosť
T8	$(B \cdot C) + (B \cdot D) = B \cdot (C + D)$	T8'	$(B + C) \cdot (B + D) = B + (C \cdot D)$	Distributívnosť
T9	$B \cdot (B + C) = B$	T9'	$B + (B \cdot C) = B$	Absorbcia
T10	$(B \cdot C) + (B \cdot \bar{C}) = B$	T10'	$(B + C) \cdot (B + \bar{C}) = B$	Spojovanie
T11	$(B \cdot C) + (\bar{B} \cdot D) + (C \cdot D) = B \cdot C + \bar{B} \cdot D$	T11'	$(B + C) \cdot (\bar{B} + D) \cdot (C + D) = (B + C) \cdot (\bar{B} + D)$	Konsenzus
T12	$\overline{B_0 \cdot B_1 \cdot \dots} = \overline{B_0} + \overline{B_1} + \dots$	T12'	$\overline{B_0 + B_1 + \dots} = \overline{B_0} \cdot \overline{B_1} \cdot \dots$	De Morganové pravidlá

Zjednodušenie b. rovníc

Príklad 1:

- $Y = AB + \bar{A}\bar{B}$

Zjednodušenie b. rovníc

Príklad 1:

- $$\begin{aligned} Y &= AB + \overline{AB} \\ &= B(A + \overline{A}) \quad T8 \\ &= B(1) \quad T5' \\ &= B \quad T1 \end{aligned}$$

Zjednodušenie b. rovníc

Príklad 2:

- $Y = A(AB + ABC)$

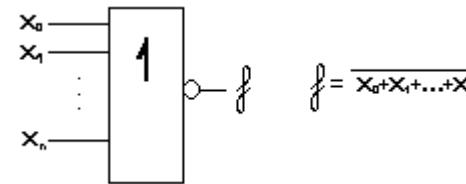
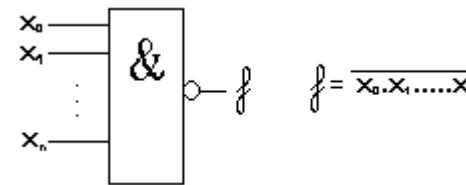
Zjednodušenie b. rovníc

Príklad 2:

- $$\begin{aligned} Y &= A(AB + ABC) \\ &= A(AB(1 + C)) \quad T8 \\ &= A(AB(1)) \quad T2' \\ &= A(AB) \quad T1 \\ &= (AA)B \quad T7 \\ &= AB \quad T3 \end{aligned}$$

Algebraické vyjadrenia a LO

- Booleova algebra:
 - Na báze: NOT, AND, OR
- Shefferova algebra
 - Na báze: NAND
- Peirceova algebra
 - Na báze: NOR



Algebraické vyjadrenia a LO

- **Prevodov NAND na OR s neg. vstupmi:**

- Zmena hradla z NAND na OR s negovanými vstupmi



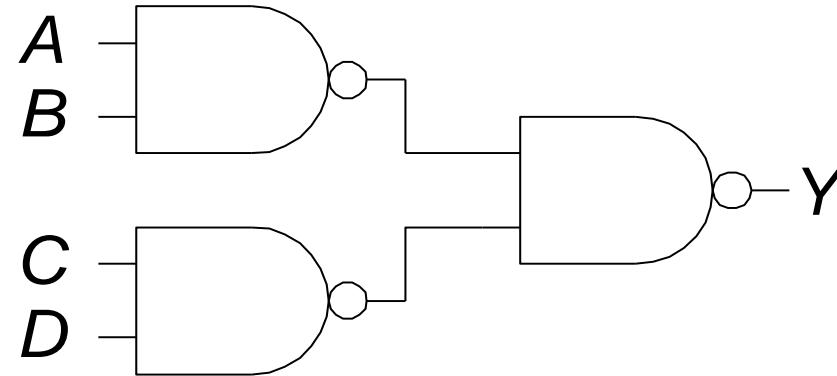
- **Prevod OR s neg. vstupmi na NAND:**

- Zmena hradla z OR na NAND



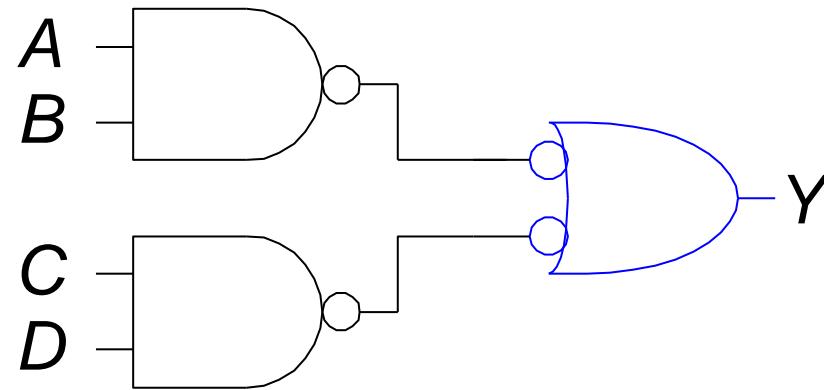
Algoritmus spätného šírenia negácie

- Určte zápis booleovskej rovnice pre nasledujúcu schému LO.



Algoritmus spätného šírenia negácie

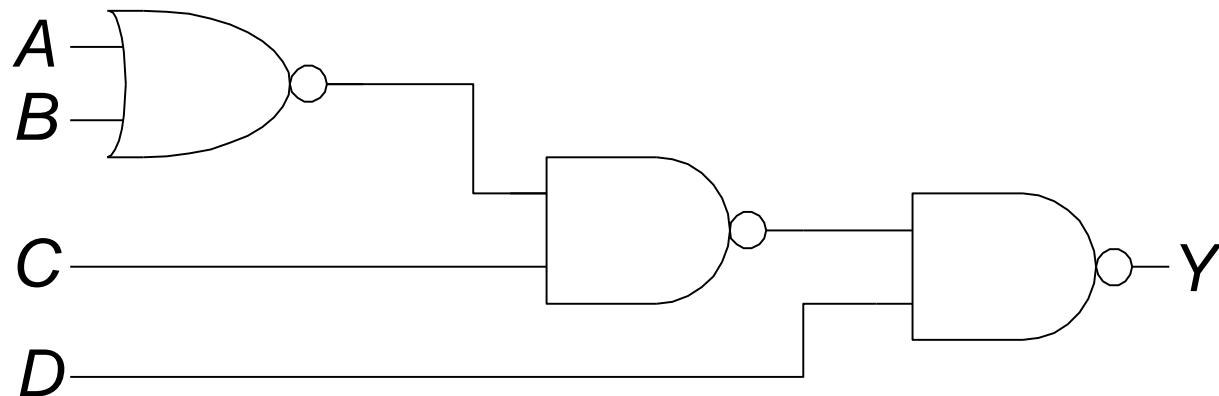
- Určte zápis booleovskej rovnice pre nasledujúcu schému LO.



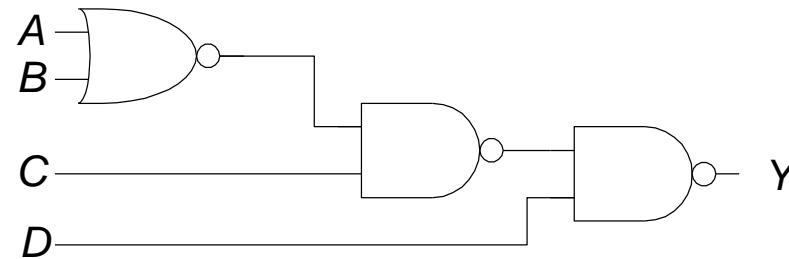
$$Y = AB + CD$$

Algoritmus spätného šírenia negácie

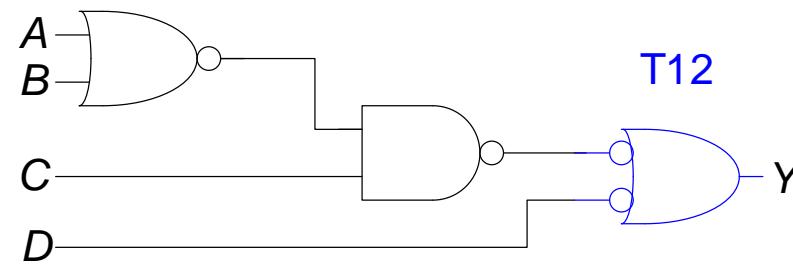
- Algoritmus spätného šírenia negácie
- Aplikuj T_{12} a T_4



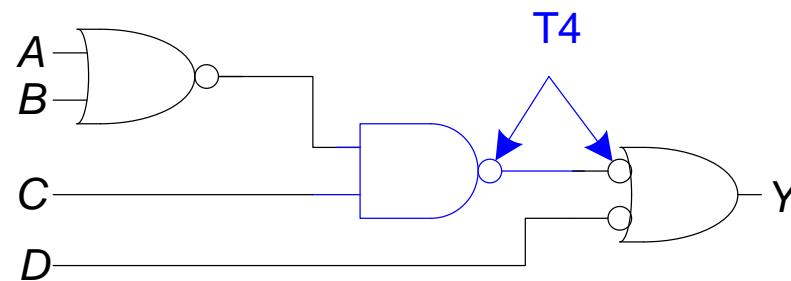
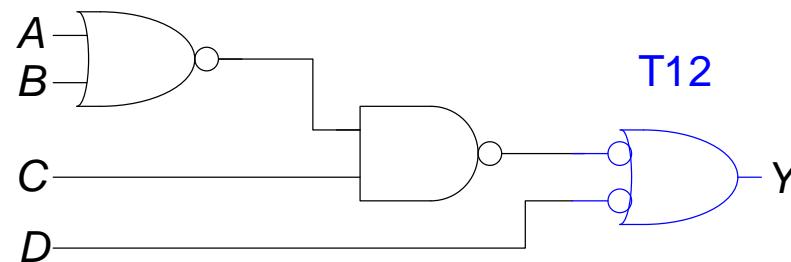
Algoritmus spätného šírenia negácie



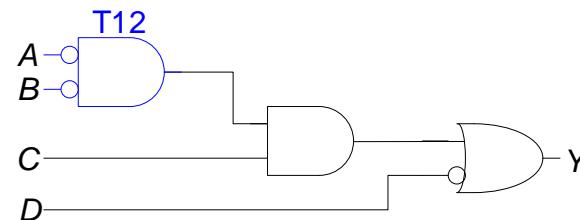
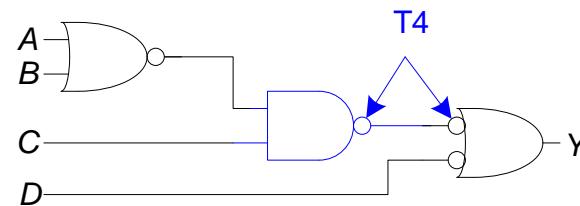
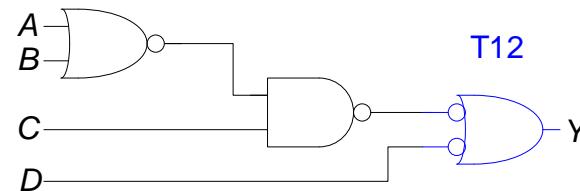
Algoritmus spätného šírenia negácie



Algoritmus spätného šírenia negácie



Algoritmus spätného šírenia negácie



$$Y = \bar{A}\bar{B}C + \bar{D}$$

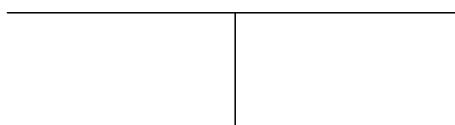
Pravidlá kreslenia schém LO

- Vstupy sú naľavo (alebo v hornej časti schémy obvodu)
- Výstupy sú napravo (alebo v dolnej časti schémy obvodu)
- Hradlá sú orientované zľava doprava, resp. zhora nadol
- Každý kanál sa znázorňuje samostatnou čiarou

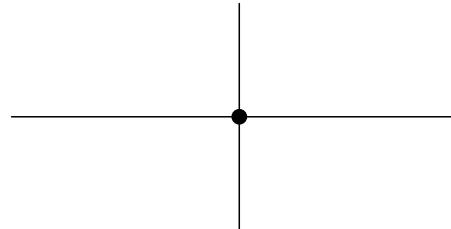
Pravidlá kreslenia schém LO

- Vodivé spojenie medzi kanálmi
 - Spoj v tvare T
 - Kríženie vodičov s bodkou
- Kríženie vodičov bez bodky: nie je vodivé spojenie medzi kanálmi

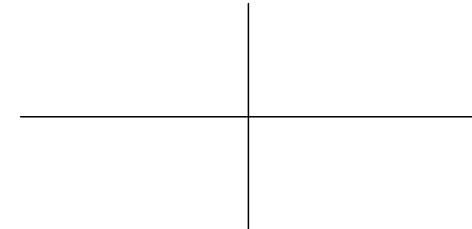
je spojenie



Je spojenie



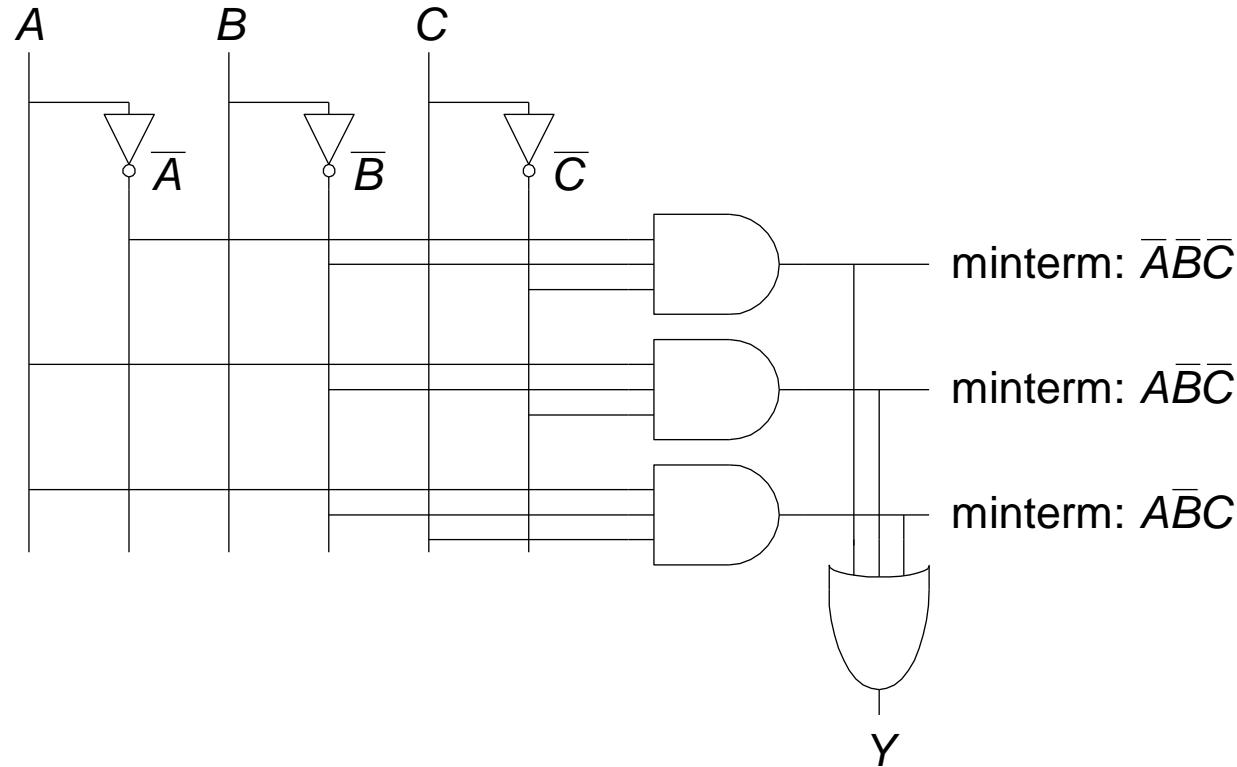
nie je spojenie



Pravidlá kreslenia schém LO

- Príklad:

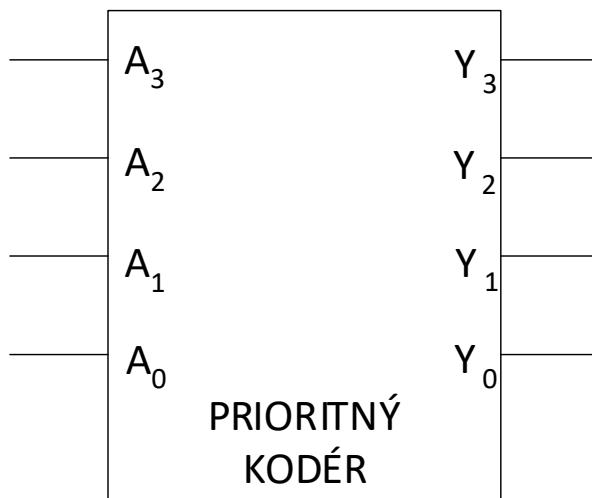
$$Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$$



Obvody s viacerými výstupmi

- **Príklad: Prioritný kodér**

Aktivácia výstupu na základe pozície najvýznamnejšej „1“ v vstupnom slove.

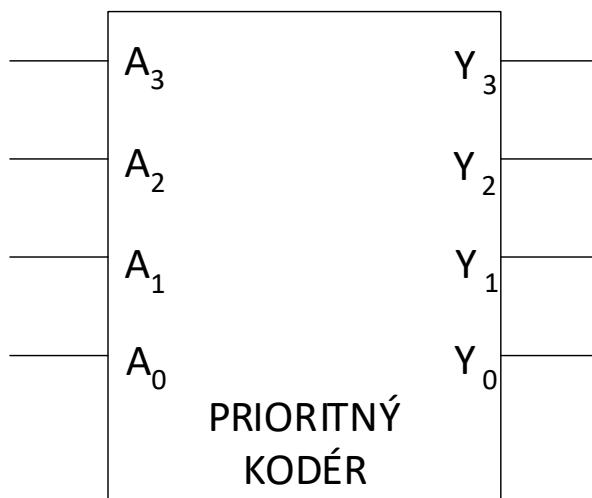


A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	1	0	0	0
0	0	1	1	1	1	1	1
0	1	0	0	0	0	0	0
0	1	0	1	0	1	0	1
0	1	1	0	1	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	1	0	0
1	1	1	0	1	1	0	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1

Prioritný kodér typu One-hot

- **Príklad: Prioritný kodér**

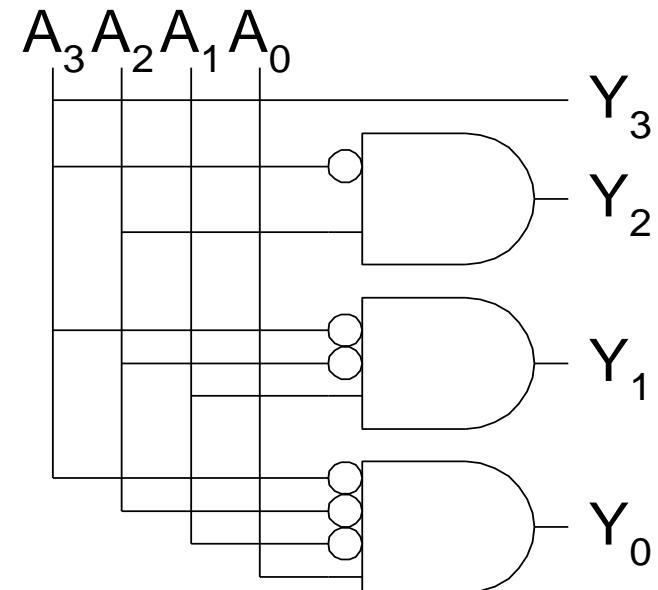
Aktivácia výstupu na základe pozície najvýznamnejšej „1“ v vstupnom slove.



A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	1	0	0	0
0	1	1	1	1	0	0	0
1	0	0	0	0	1	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	1	0	0
1	1	0	0	0	1	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	1	1	0	0

Prioritný kodér typu One-hot

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	0	0	0
0	0	1	1	0	0	0	0
1	0	0	0	1	1	0	0
1	0	0	1	0	0	0	0
1	0	1	0	0	0	0	0
1	1	0	0	1	1	0	0
1	1	1	0	0	0	0	0
1	1	1	1	1	0	0	0



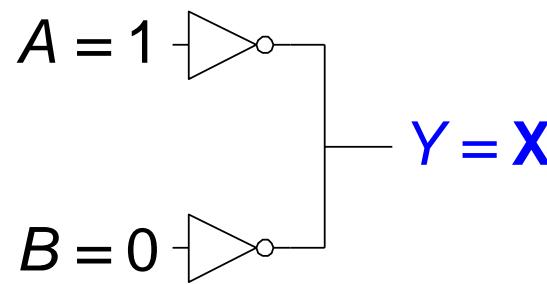
Signál s hodnotou „X“

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	0	1	0	0	0	0	0
0	1	1	0	0	1	0	0
0	0	1	1	0	0	0	0
1	1	0	0	1	1	0	0
1	0	0	1	0	0	0	0
1	0	1	0	0	0	0	0
1	0	1	1	1	1	0	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	0	0
1	1	1	0	1	1	0	0
1	1	1	1	1	1	0	0

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

„X“ a konflikt

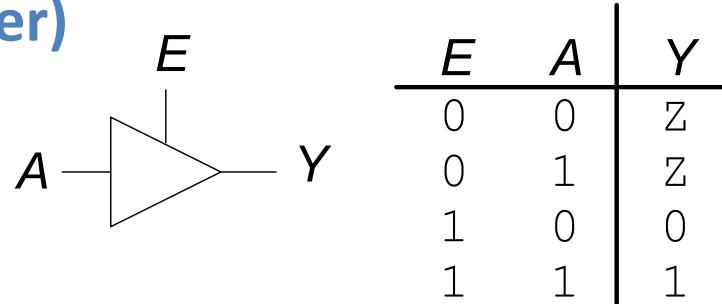
- X reprezentuje hodnotu na ktorej nezáleží (PT, K-map)
- X reprezentuje konflikt
 - signál je budený z dvoch rôznych zdrojov signálu, pričom každý zdroj dáva inú logickú hodnotu ('0' a '1').
 - Hodnota napäťia je niekde medzi GND a VDD
 - Môže byť vyhodnotené raz ako GND, inokedy ako VDD, alebo je v zakázanej zóne
 - Mení sa v závislosti od „sily“ zdroja, teploty, času, prítomného šumu
 - Môže ovplyvňovať nepriaznivo spotrebú



- **Pozor:**
 - Prítomnosť X naznačuje chybu v návrhu.
 - **Význam X** – podľa kontextu

Vysokoimpedančná hodnota „Z“

- Hodnota 'Z'
 - Reprezentuje vysokoimpedančný (plávajúci) stav zdroja signálu
 - „Plávajúci“ výstupný signál môže mať hodnotu 0 ako aj 1
 - Stav vysokej impedancie daného uzla v obvode nie je možné overiť voltmetrom (meraním napäťia).
 - **Budič (trojstavový buffer)**



Vysokoimpedančná hodnota „Z“

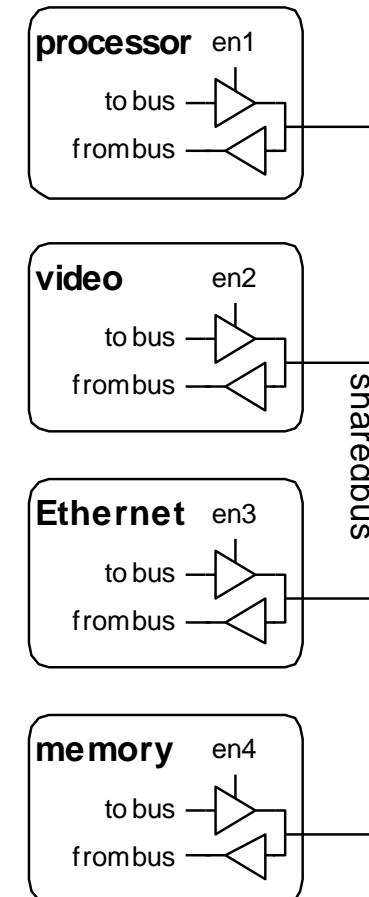
- Súčiastky s trojstavovými výstupmi
 - Je možné paralelne spájať
 - Z celej skupiny paralelne spojených súčiastok, výstup len jednej z nich môže byť v stave „0“ alebo „1“. Ostatné musia byť v stave „Z“.

Vysokoimpedančná hodnota „Z“

- Obvod s výstupom v stave „Z“
 - Neovplyvňuje napätie v uzle
 - Ak by bol obvod samostatný, nebolo by jeho výstupné napätie vôbec definované.
 - Avšak, napätie v uzle - spravidla - je definované niektorou z ostatných súčiastok v uzle, napr. rezistorom spojeným so zemou, napájacím napätím, alebo iným napätím.

Trojstavové zbernice

- Obvody (periférne zariadenia) sú v stave vysokej impedancie, tzn. odpojené od zbernice
 - Je možné pripojiť niekoľko periférií
 - Na rovnakej linke nemôže byť súčasne viac než jeden vysielač aktívny

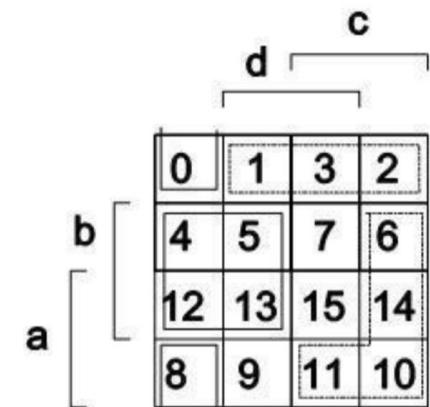


Karnaughové mapy (K-mapy)

- Princíp minimalizácie vyjadrenia B-funkcie pomocou Karnaughových máp spočíva v aplikovaní pravidla spojovania na všetky dvojice susedných elementárnych súčinov.
- Susednými elementárnymi súčinmi sa nazývajú elementárne súčiny tých istých premenných, ktoré sa líšia v hodnote práve jednej premennej.

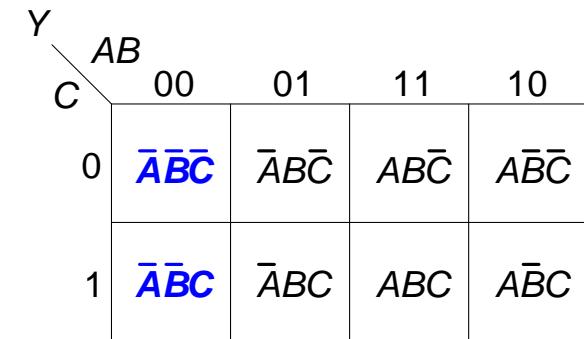
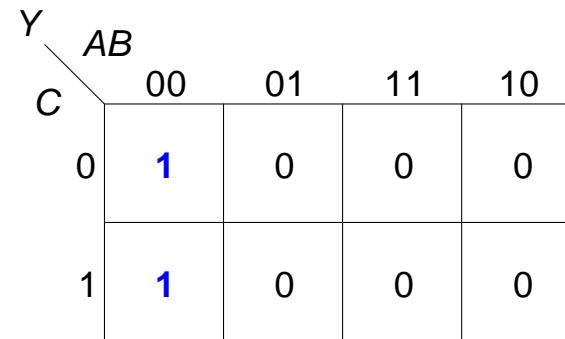
K-mapy

- Každý elementárny súčin rádu r predstavuje v mape tzv. pravidelnú konfiguráciu 2^{n-r} štvorčekov.
- Pravidelnou konfiguráciou stupňa k sa nazýva konfigurácia 2^k štvorčekov, z ktorých každý má v nej práve k susedov.
- Príklady pravidelných (plná čiara) a nepravidelných konfigurácií (prerušovaná čiara) sú na obrázku.



K-mapa 3 premenných

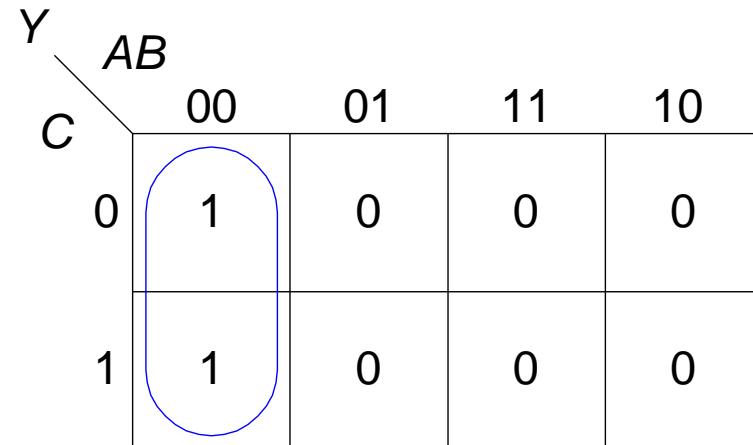
A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



K-mapa 3 premenných

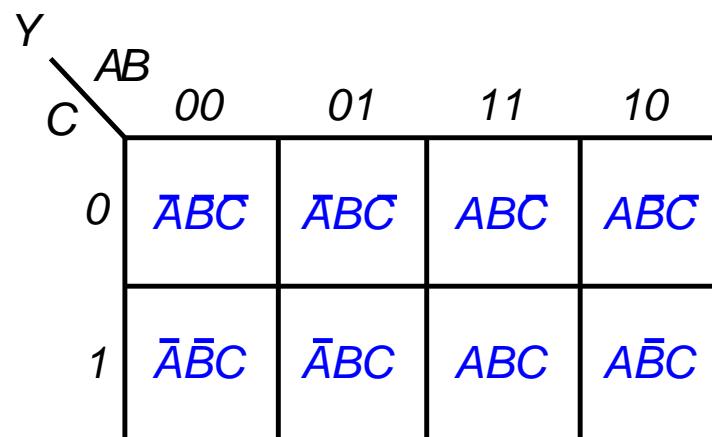
- Vytvárame pravidelné konfigurácie
- Krúžkujeme „1“ patriace do pravidelnej konfigurácie

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



$$Y = \bar{A}\bar{B}$$

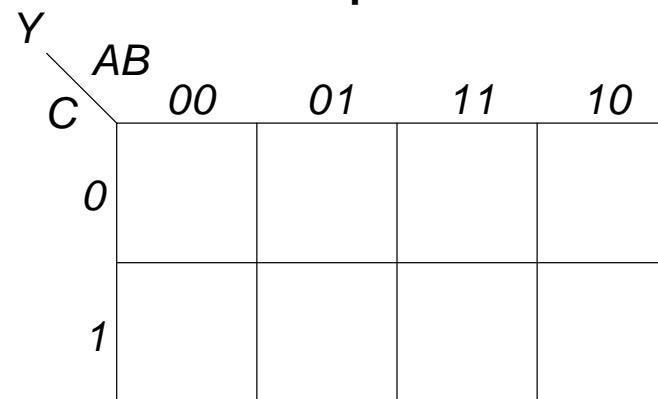
K-mapa 3 premenných



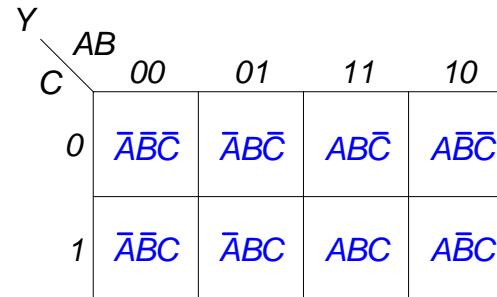
Pravdivostná tabuľka

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

K-mapa



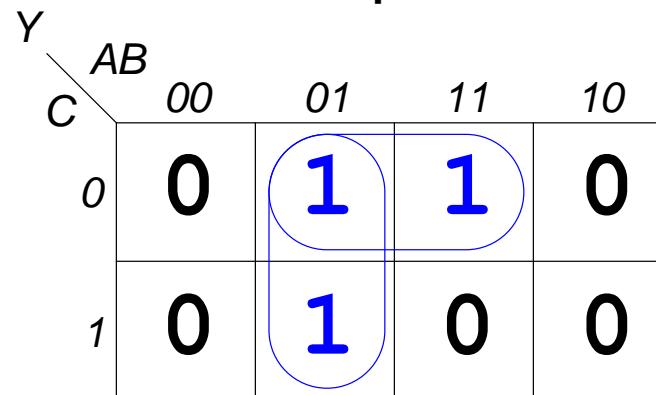
K-mapa 3 premenných



Pravdivostná tabuľka

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

K-mapa



$$Y = \bar{A}B + B\bar{C}$$

K-mapa: definície

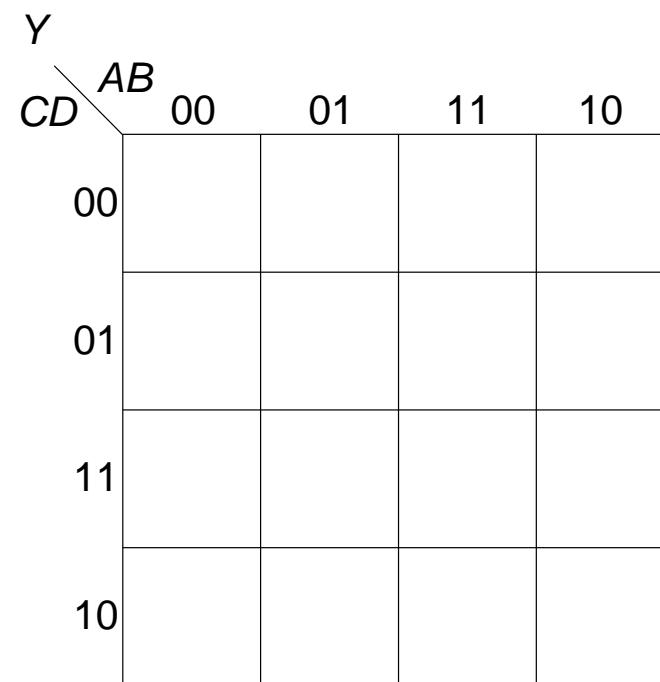
- **Komplement:** b. premenná v negovanej forme
 A, B, C
- **Literál:** b. premenná v priamej forme alebo jej komplement
 $A, \bar{A}, B, \bar{B}, C, \bar{C}$
- **Implikant:** súčin literálov
 $ABC, A\bar{C}, \bar{B}C$
- **Prostý implikant:** implikant definovaný s pravidelnou konfiguráciou s maximálnym možným stupňom k

K-mapa: pravidlá

- Každá „1“ musí byť zaradená aspoň do jednej pravidelnej konfigurácie
- Preferované sú pravidelné konfigurácie s maximálnym stupňom k
- Ak K-mapa obsahuje neurčené body (X), tieto body sa stanú súčasťou pravidelnej konfigurácie vtedy a len vtedy, ak by to viedlo „lepšej“ minimalizácií b. funkcie

K-mapa 4 premenných

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0



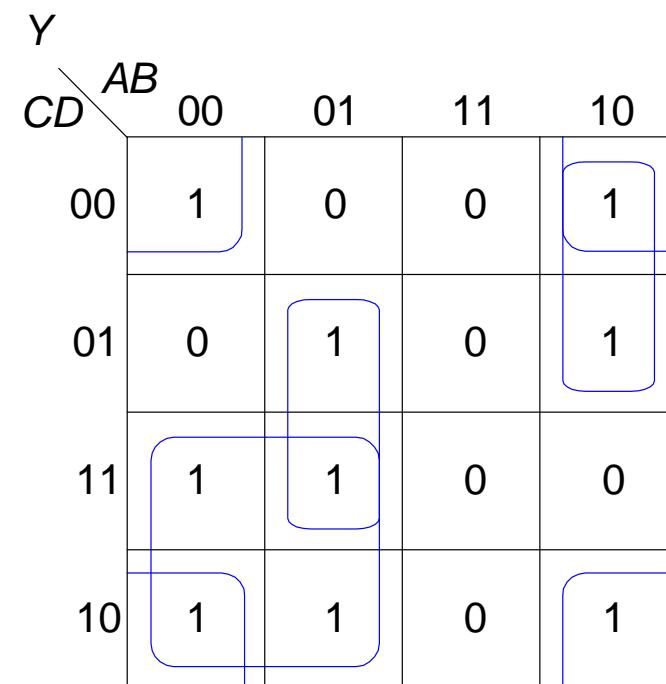
K-mapa 4 premenných

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

		AB	00	01	11	10
Y	CD	00	1	0	0	1
	01	0	1	0	1	
	11	1	1	0	0	
	10	1	1	0	1	

K-mapa 4 premenných

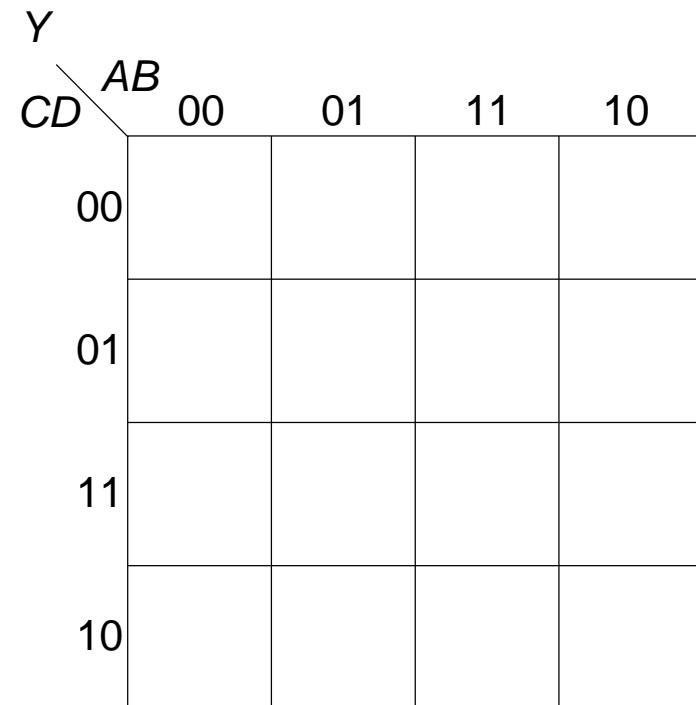
A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0



$$Y = \bar{A}C + \bar{A}BD + A\bar{B}\bar{C} + \bar{B}\bar{D}$$

K-mapa s neurčenými bodmi

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	X
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X



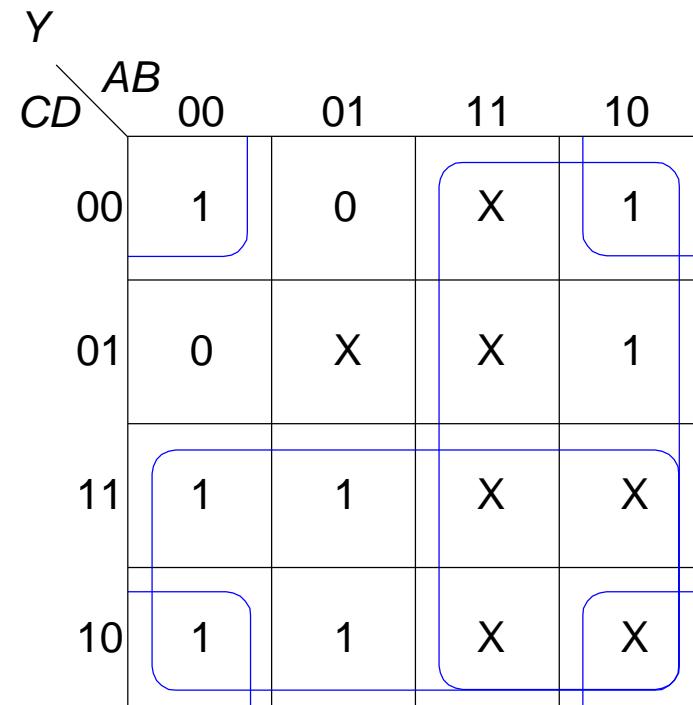
K-mapa s neurčenými body

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	X
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

	AB	00	01	11	10
CD	00	1	0	X	1
	01	0	X	X	1
	11	1	1	X	X
	10	1	1	X	X

K-mapa s neurčenými body

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	X
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X



$$Y = A + \bar{B}\bar{D} + C$$

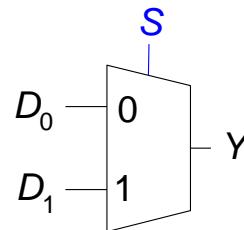
Kombinačné logické obvody

- Multiplexor
- Dekodér
- Demultiplexor (samoštúdium)
- Komparátor (samoštúdium)
- Prevodník kódu (samoštúdium)
- Polovičná a úplná sčítačka (samoštúdium)

Multiplexor (MUX)

- Má M adresných (riadiacich), N dátových vstupov a jeden výstup
- $M = \log_2 N$
- Príklad:

2:1 Mux

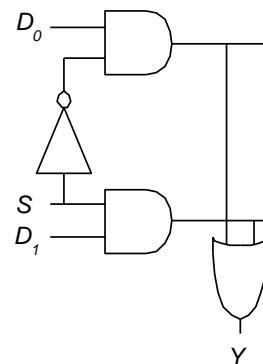
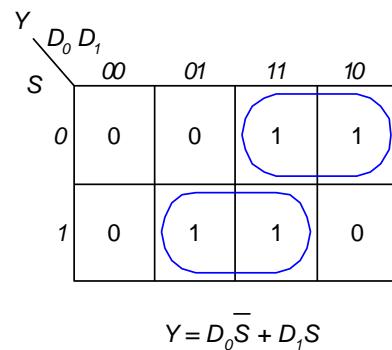


S	D_1	D_0	Y	S	Y
0	0	0	0	0	D_0
0	0	1	1	1	D_1
0	1	0	0		
0	1	1	1		
1	0	0	0		
1	0	1	0		
1	1	0	1		
1	1	1	1		

Implementácia MUX

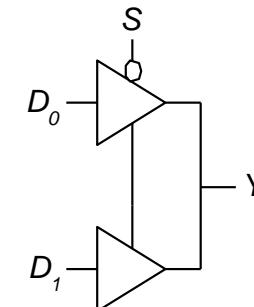
- Pomocou hradiel

- DNF



- Trojstavové buffre

- Pre N -vstupový MUX použi N 3-buffre
- Len jeden z N buffrov má byť aktívny

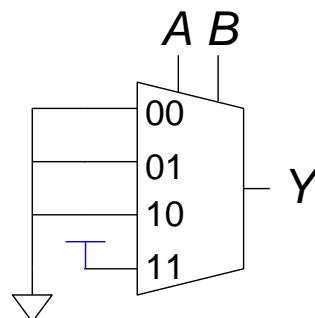


Realizácia LO pomocou MUX

- MUX ako prehľadávacia tabuľka (angl. look-up table, LUT)

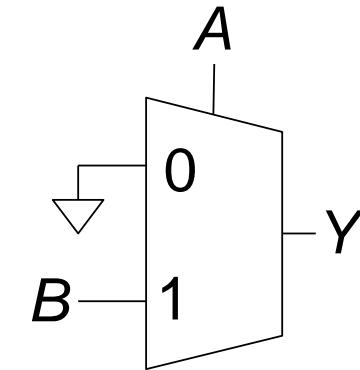
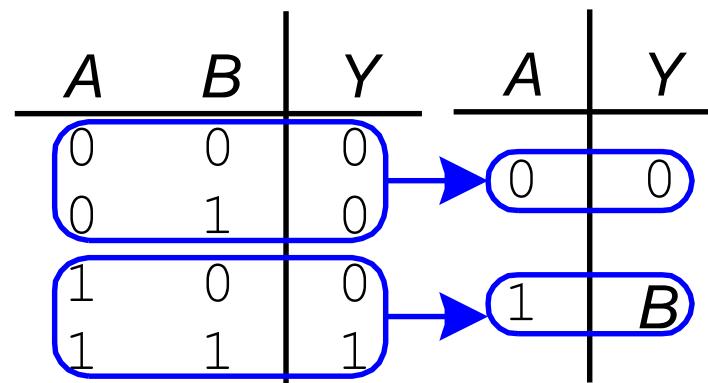
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$$Y = AB$$



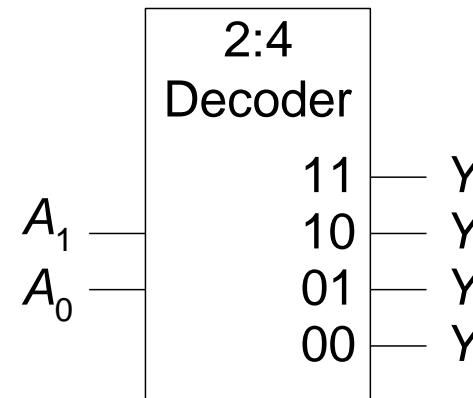
Redukcia MUX

$$Y = AB$$



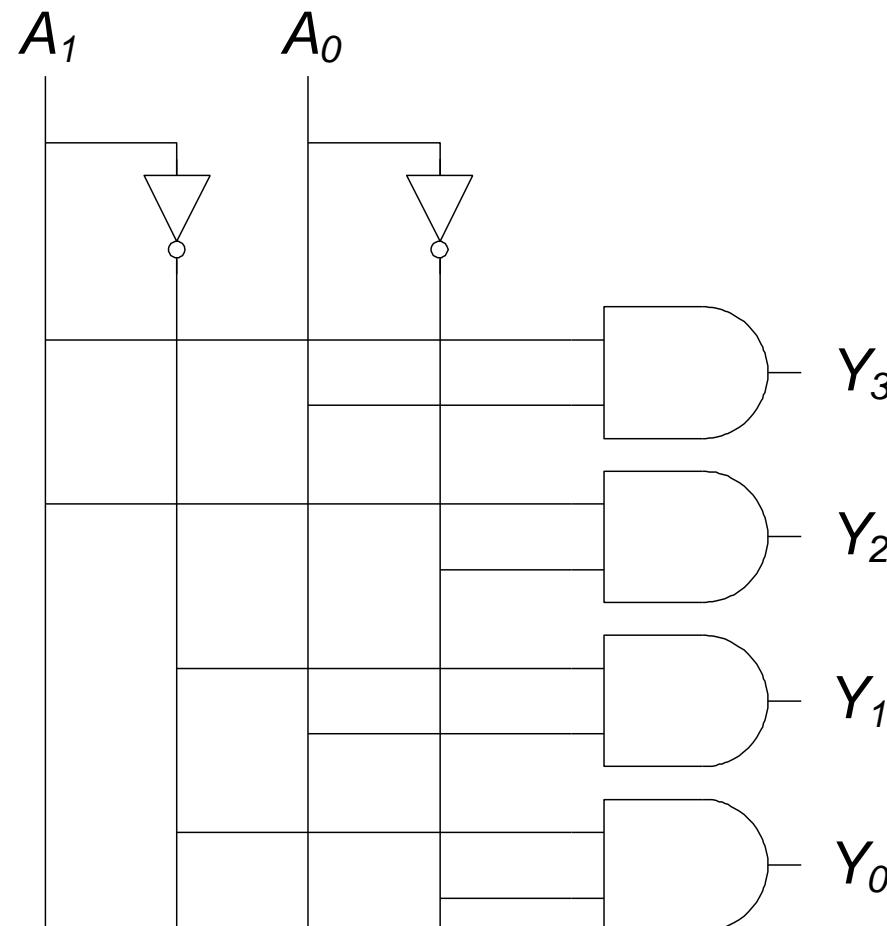
Dekodéry

- Má N vstupov, 2^N výstupov
- One-hot kódovanie: v danom čase je len jeden výstup HIGH



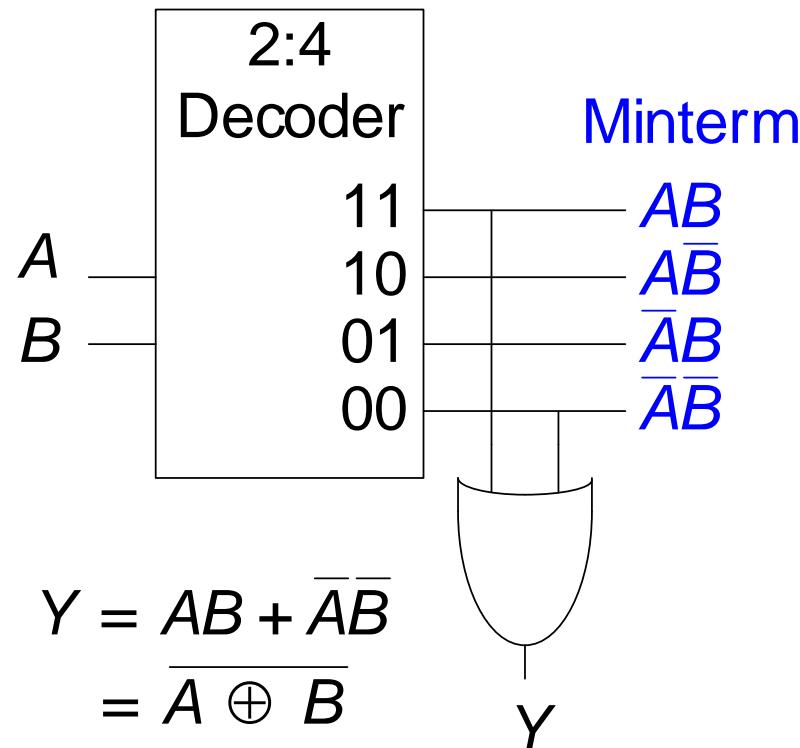
A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Implementácia dekodéra



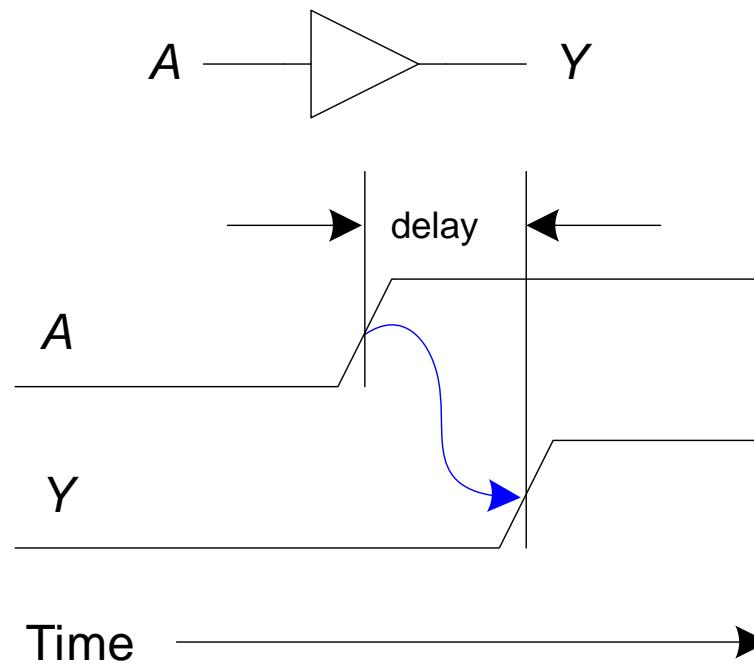
Realizácia LO pomocou dekodéra

- OR pre implikanty



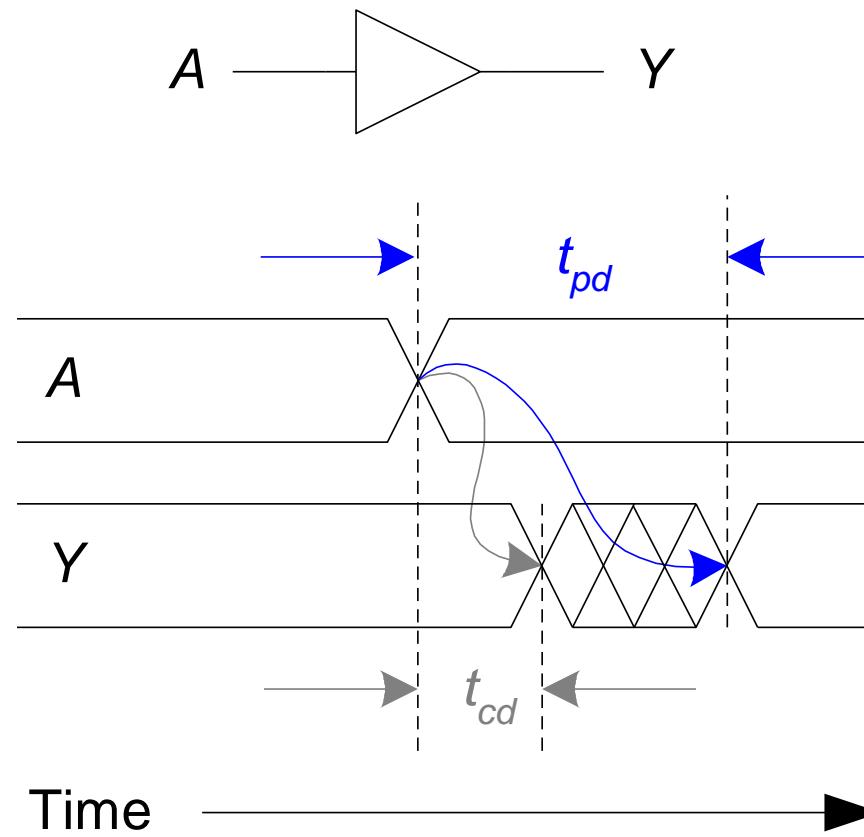
Časové charakteristiky

- Dynamické charakteristiky číslicových signálov
- Doba oneskorenia priechodu signálu



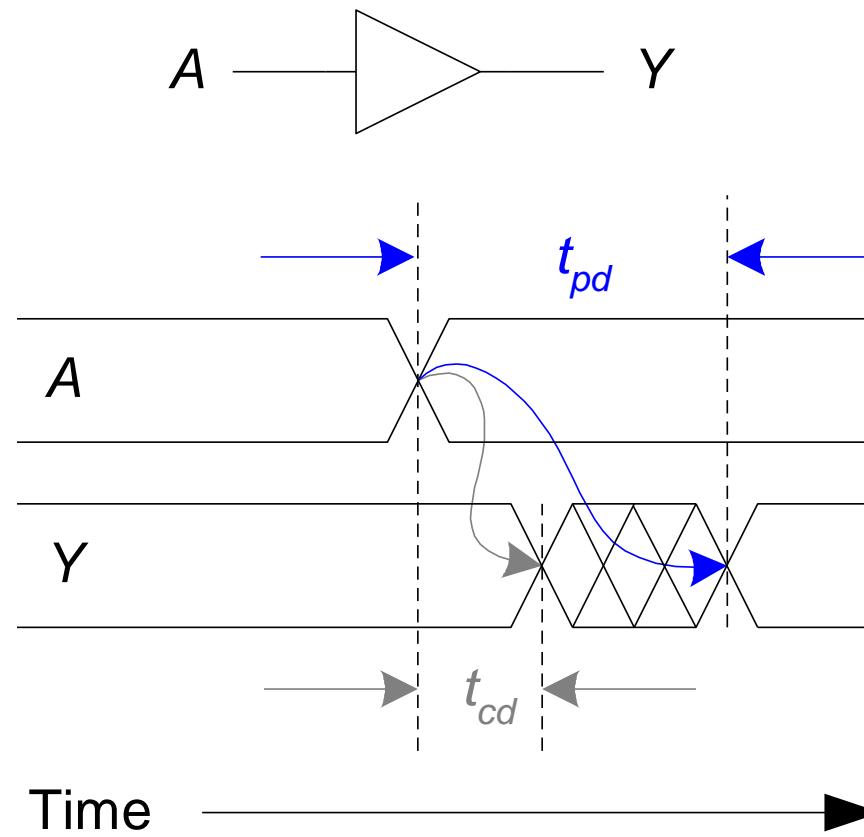
Časové charakteristiky

- Propagačné oneskorenie (angl. propagation delay): $t_{pd} = \max$ oneskorenie signálu



Časové charakteristiky

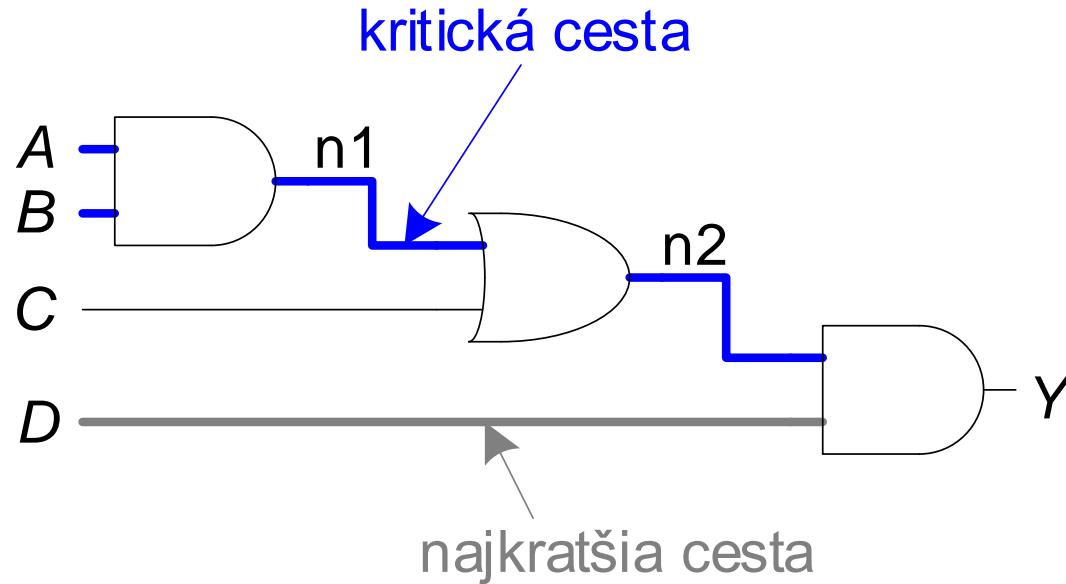
- Kontaminačné oneskorenie (angl. contamination delay): $t_{cd} = \min$ oneskorenie signálu



Propagačné a kontaminačné oneskorenie

- Oneskorenie je spôsobené
 - Vlastnosťami vodičov (z čoho je vodič vyrobený)
 - Rýchlosť pohybu elektrónov
- Prečo $t_{pd} \neq t_{cd}$:
 - Rozdielna doba nábehu (angl. rise time) a dobehu signálu (angl. fall time)
 - Signály nemusia doraziť súčasne na vstupy viacvstupového obvodu
 - Rýchlosť prenosu signálu klesá zvyšujúcou sa teplotou vodiča

Kritická a najkratšia cesta



Kritická (dlhá) cesta: $t_{pd} = 2t_{pd_AND} + t_{pd_OR}$

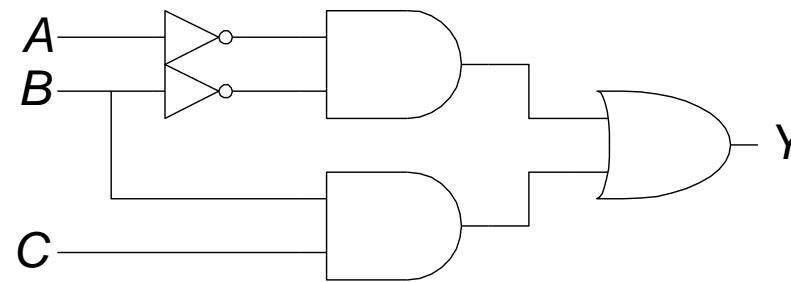
Krátká cesta: $t_{cd} = t_{cd_AND}$

Falošné impulzy, hazardy

- Ak jedna zmena vstupného signálu vyvolá viacnásobnú zmenu výstupného signálu
- O tom, či vznikne falošný impulz, kedy vznikne a či vôbec vznikne rozhodujú časové charakteristiky vstupných signálov

Príklad: falošný impulz

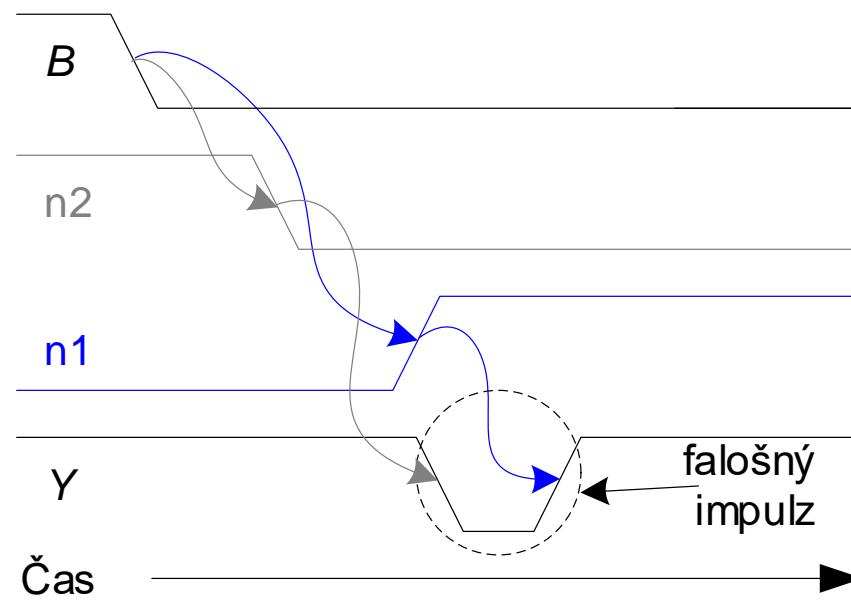
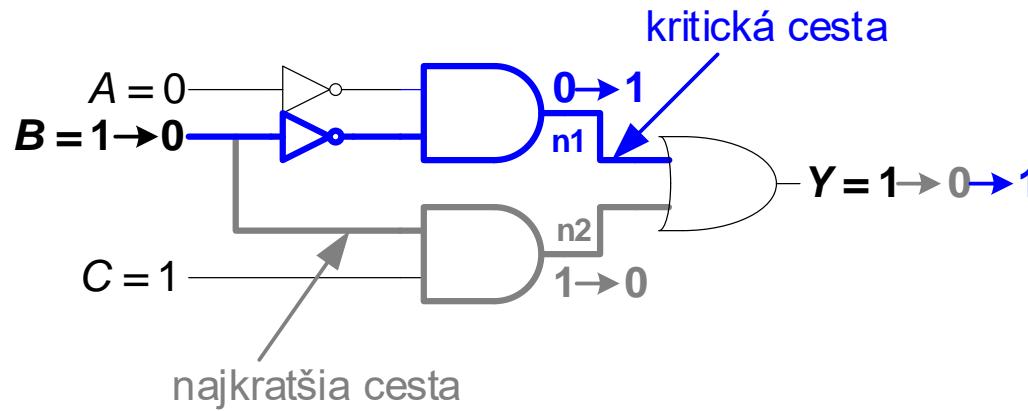
- Čo sa stane ak $A = 0, C = 1$, a $B: 0 \rightarrow 1$?



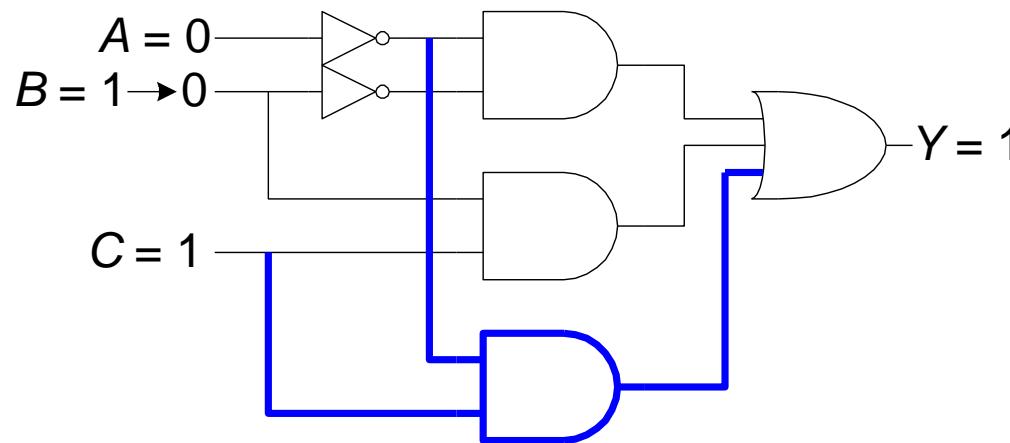
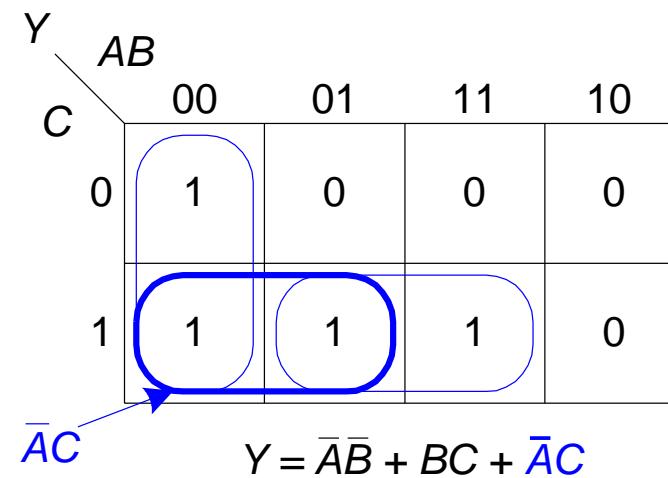
		AB	00	01	11	10
		C	0	0	0	0
Y	AB	0	1	0	0	0
		1	1	1	1	0

$$Y = \bar{A}\bar{B} + BC$$

Príklad: falošný impulz



Blokovanie falošného impulzu



Referencia

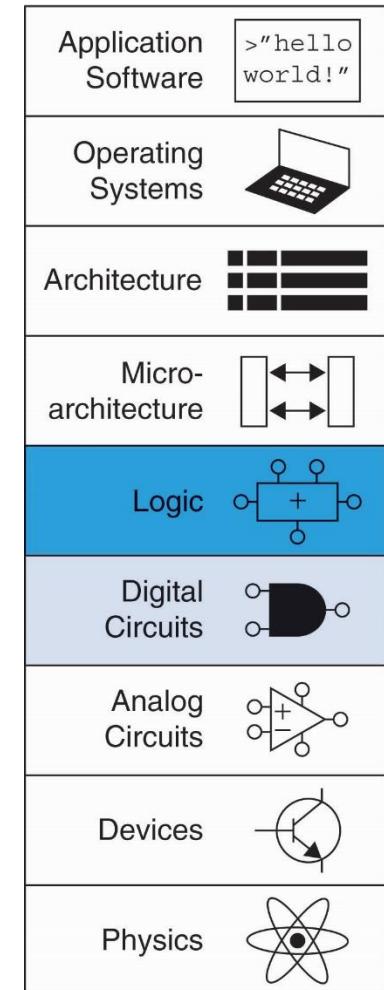
- David Money Harris and Sarah L. Harris,
Digital Design and Computer Architecture,
Chapter 2: Combinational Logic Design,
Second Edition © 2012 by Elsevier Inc.

Architektúry počítačových systémov

2. rok ZS

Osnova

- **Úvod**
- **Preklápacie obvody**
- **Synchrónne systémy**
- **Stavové automaty**
- **Časové charakteristiky**
- **Paralelizmus**



- Sekvenčné logické obvody (SLO)
 - výstup závisí od kombinácie vstupov a od vnútorných stavov obvodu z predchádzajúceho taktu
 - *má pamäť*.
- Definície:
 - **Stav:** konfigurácia nezávislých (vstupných) a závislých (vnútorných a výstupných) veličín
 - **Preklápacie obvody:** uchováva jednobitovú informáciu
 - **Sekvenčný obvod:** pozostáva z KLO a z preklápacích obvodov

Sekvenčný logický obvod

- V logických systémoch vo všeobecnosti výstupný vektor v danom okamihu nie je jednoznačne učený okamžitým vektorom na vstupe obvodu, ale je závislý od sekvencie (postupnosti) vektorov na vstupe v predchádzajúcich taktoch, preto sa tieto systémy nazývajú sekvenčné logické systémy.

Sekvenčný logický obvod

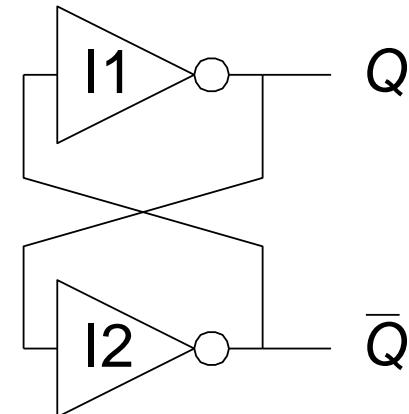
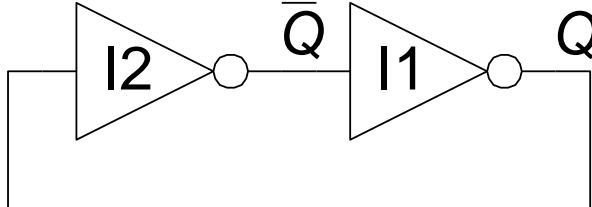
- To znamená, že sekvenčný logický systém obsahuje podsystémy, ktoré sú schopné pamätať si informáciu z predchádzajúcich taktov. Táto informácia je vyjadrená vnútorným stavom systému, ktorý je reprezentovaný určitou kombináciou hodnôt vnútorných premenných. Hodnoty vnútorných premenných sú uchované v pamäťovej časti obvodu.

Obvody určené na pamäťanie

- Na uchovávanie vnútorného stavu
 - Bistabilné obvody
 - Preklápacie obvody
 - Hladinou riadené (angl. latch) a hranou riadené (angl. flip-flop)
 - Synchrónne a asynchrónne
 - D, T (samoštúdium)
 - SR, JK (samoštúdium),

Bistabilný obvod

- Základ obvodov určených na pamätanie info.
- Dokáže si pamätať informácie
- Má dva stabilné stavy: Q, \bar{Q}



Bistabilný obvod

- Analýza správania sa obvodu:

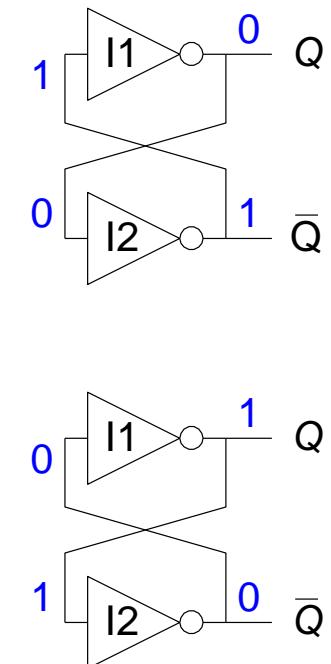
- $Q = 0$:

- potom $\bar{Q} = 1, Q = 0$ (konzistentný)

- $Q = 1$:

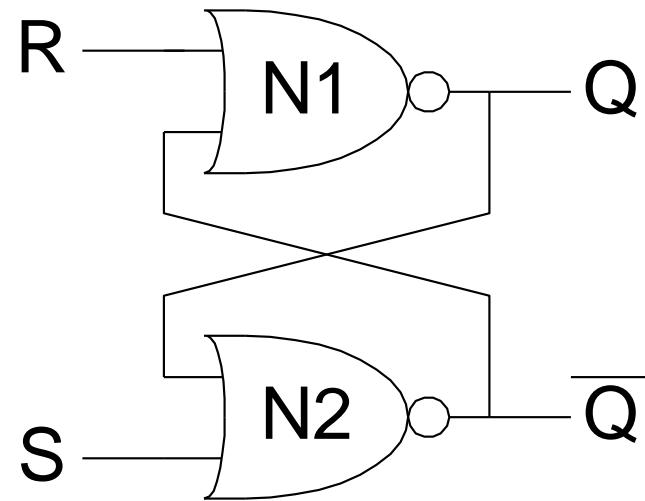
- potom $\bar{Q} = 0, Q = 1$ (konzistentný)

- Uchováva 1 bit informácie pomocou stavovej premennej Q (resp. \bar{Q})
- Nemá žiadny vstup**



SR (Set/Reset) preklápací obvod

- SR PO

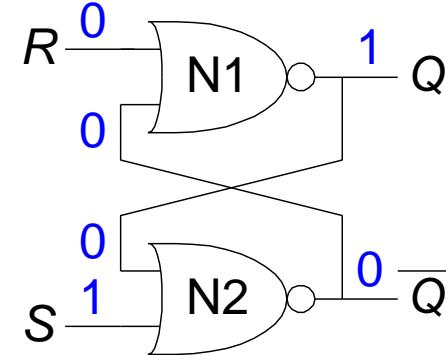


- Možné kombinácie vstupov:
 - $S = 1, R = 0$
 - $S = 0, R = 1$
 - $S = 0, R = 0$
 - $S = 1, R = 1$

Analýza SR PO

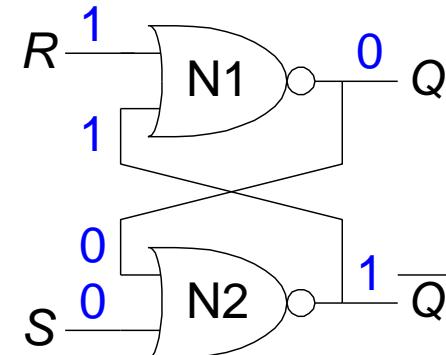
- $S = 1, R = 0$:

potom $Q = 1$ a $\bar{Q} = 0$



- $S = 0, R = 1$:

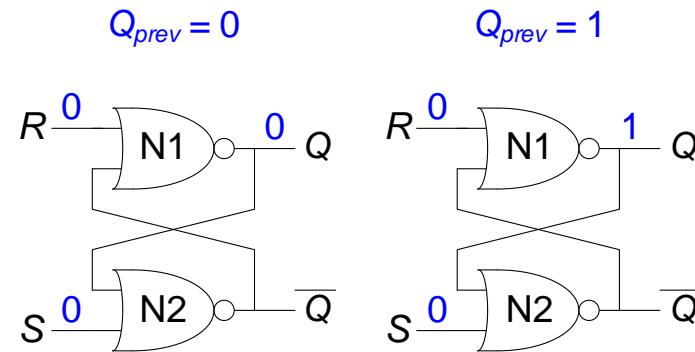
potom $Q = 0$ a $\bar{Q} = 1$



Analýza SR PO

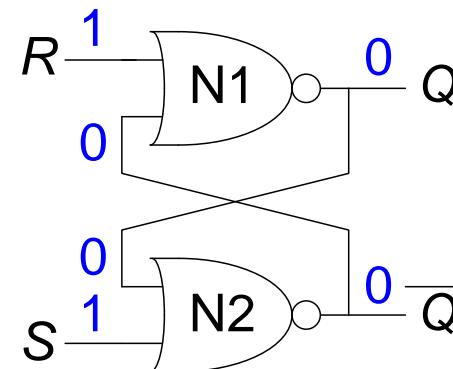
- $S = 0, R = 0$:

potom $Q = Q_{prev}$



- $S = 1, R = 1$:

potom $Q = 0$ a $\bar{Q} = 0$



Analýza SR PO

- $S = 0, R = 0$:

potom $Q = Q_{prev}$

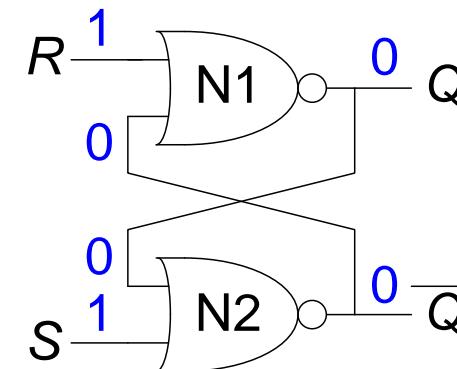
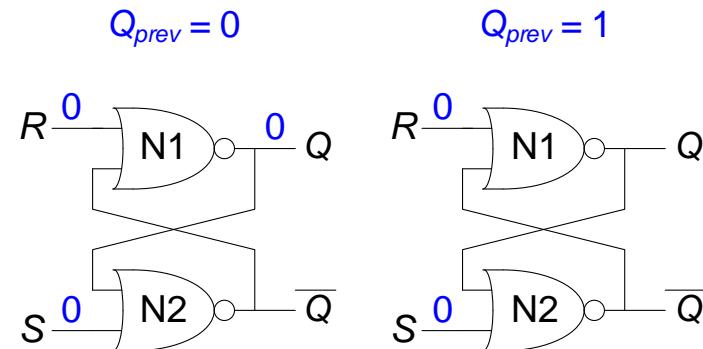
- **Pamäť!**

- $S = 1, R = 1$:

potom $Q = 0$ a $\bar{Q} = 0$

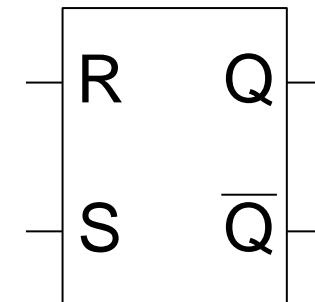
- **Neprípustný stav**

$\bar{Q} \neq NOT Q$



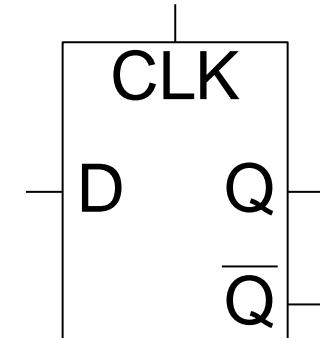
- SR znamená Set/Reset
 - Uchováva 1 bit informácie (Q) o stave systému
- Na kontrolu sa používajú vstupy S a R
 - **Set:** Nastaví výstup na 1
 $(S = 1, R = 0, Q = 1)$
 - **Reset:** Nastaví výstup na 0
 $(S = 0, R = 1, Q = 0)$
- **Je potrebné zabrániť kombinácií $S = R = 1 !!!$**

SR Latch
Symbol

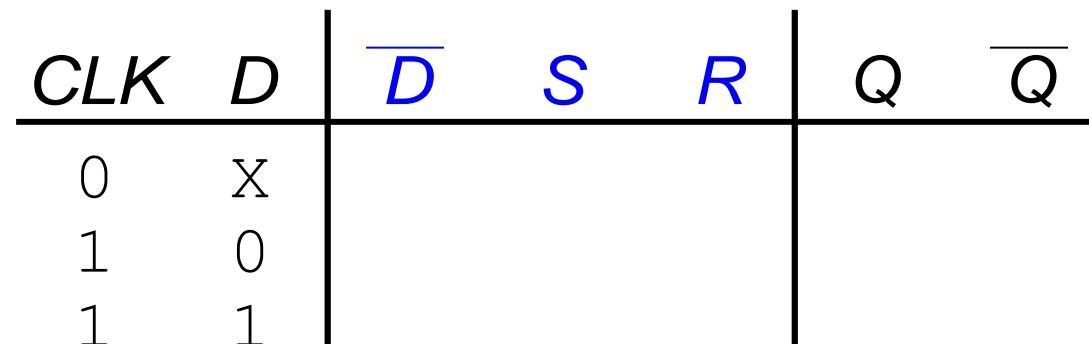
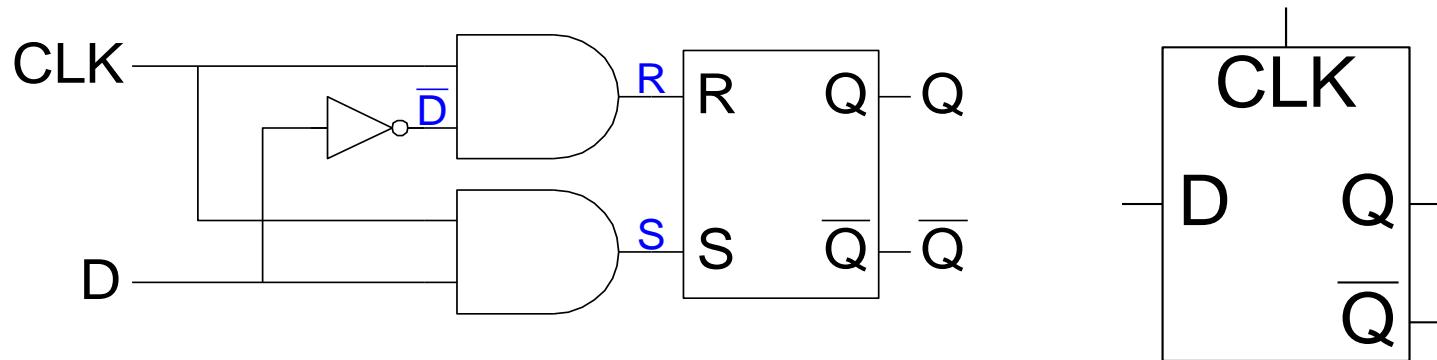


- Dva vstupy: CLK , D
 - CLK (*clock*): definuje moment vzorkovania vstupu
 - D (*data*): na akú hodnotu má byť nastavený výstup
- Funkcia
 - Ak $CLK = 1$,
 D sa dostane na Q (*transparentný*)
 - Ak $CLK = 0$,
 Q si pamätá predchádzajúcu hodnotu

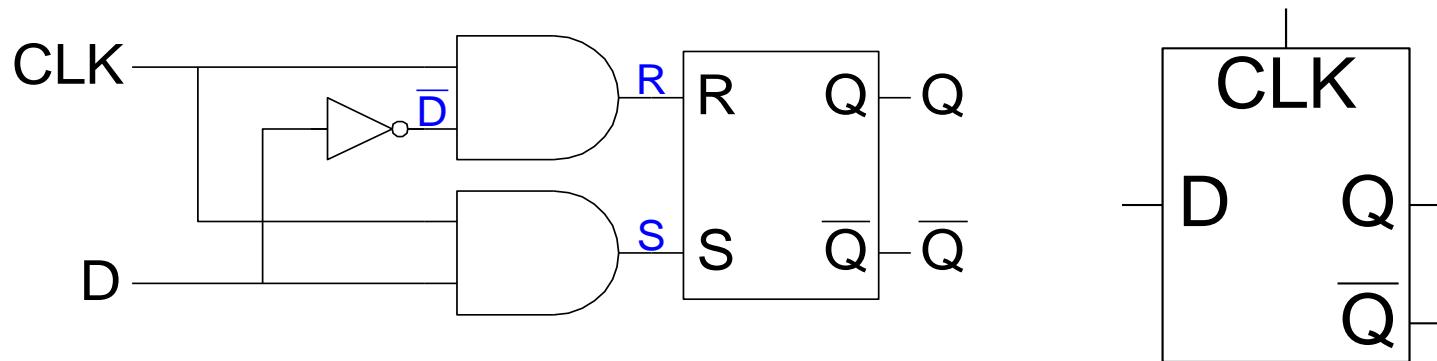
D Latch
Symbol



Implementácia D PO



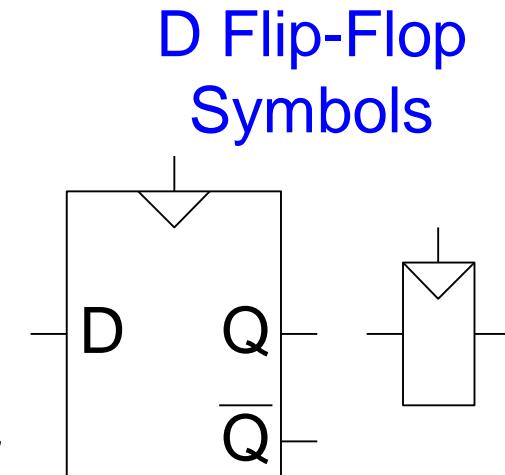
Implementácia D PO



CLK	D	\bar{D}	S	R	Q	\bar{Q}
0	X	X	0	0	Q_{prev}	\bar{Q}_{prev}
1	0	1	0	1	0	1
1	1	0	1	0	1	0

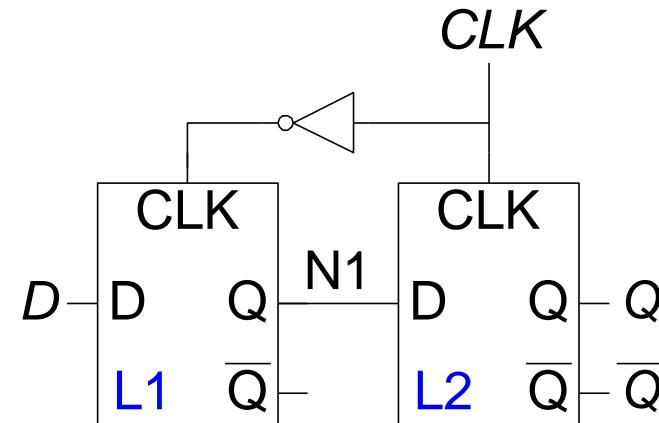
D Flip-Flop (D FF)

- **Vstupy:** CLK , D
- **Funkcia**
 - Vzorkuje D pri prechode CLK z jednej úrovne na druhú. Napr.
 - Pri zmene CLK z 0 na 1, $Q = D$, inak, $Q = Q_{prev}$
 - Q sa mení len pri zmene CLK
- Hranou riadený PO
 - Vzostupná (nábehová)
 - Zostupná (dobebehová)

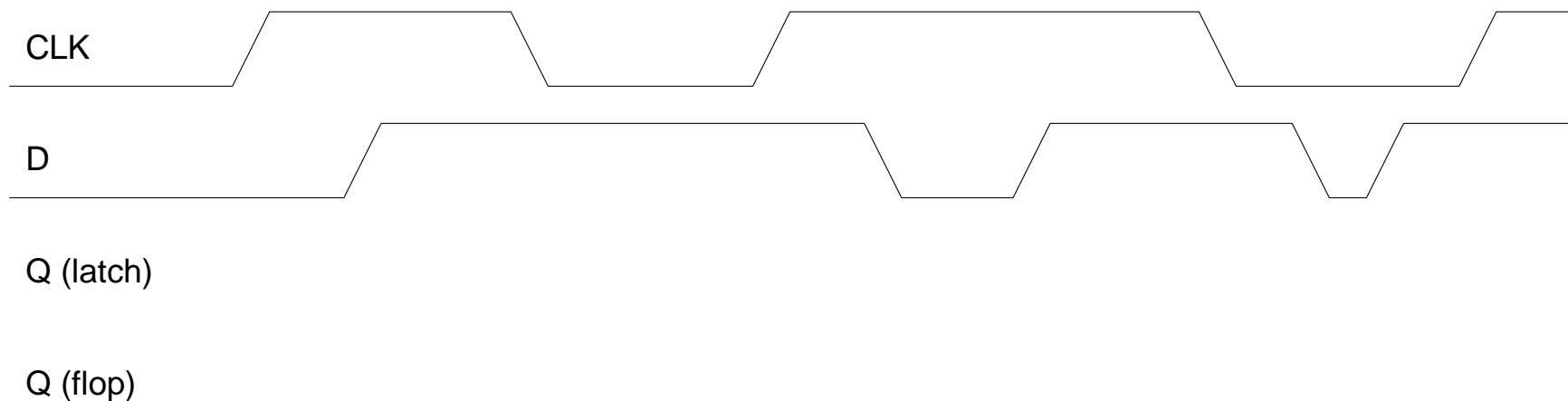
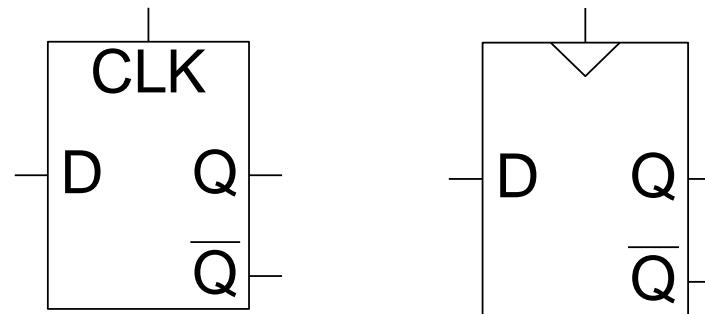


Implementácia D FF

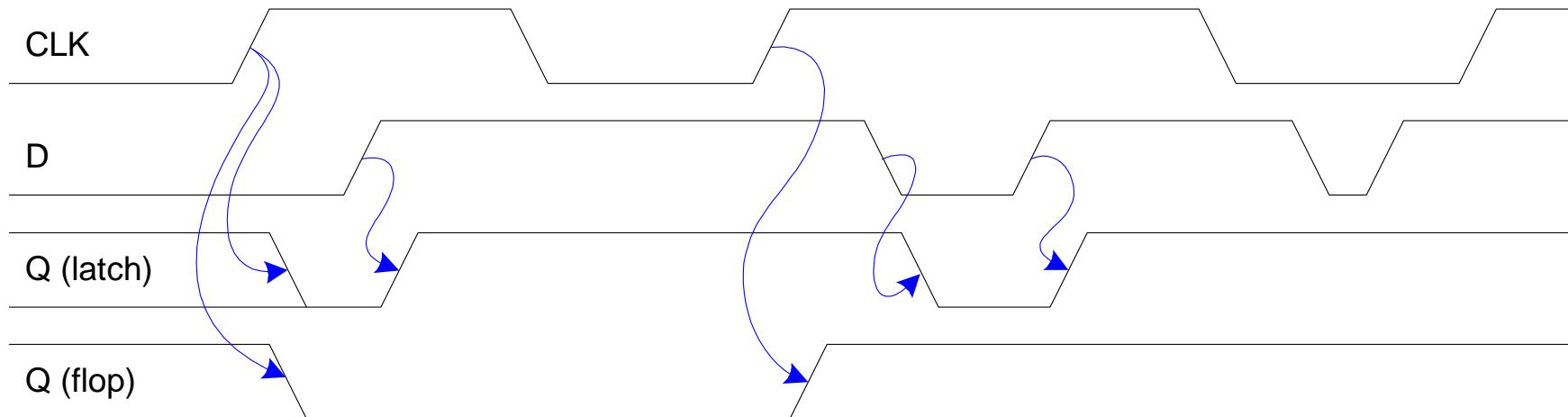
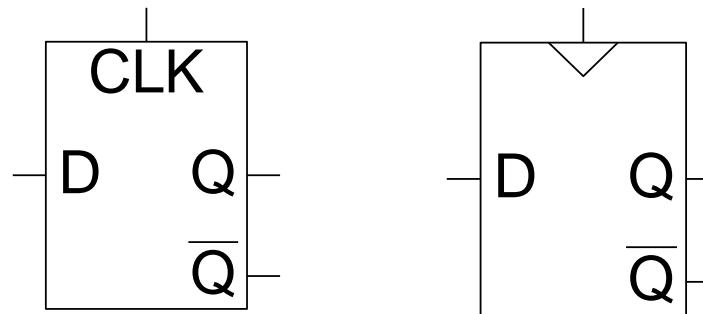
- Dva hladinou riadené D PO (L1 and L2) kontrolované komplementárnymi hodinovými impulzmi
- Ak $\text{CLK} = 0$
 - L1 je transparentný
 - L2 je blokovaný
 - D sa dostane do N1
- Ak $\text{CLK} = 1$
 - L2 je transparentný
 - L1 je blokovaný
 - N1 sa dostane na Q
- Tzn. že pri zmene CLK (Ak CLK sa mení z $0 \rightarrow 1$)
 - D sa dostane na Q



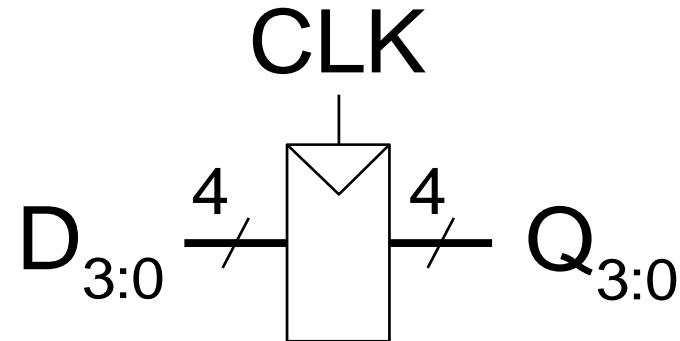
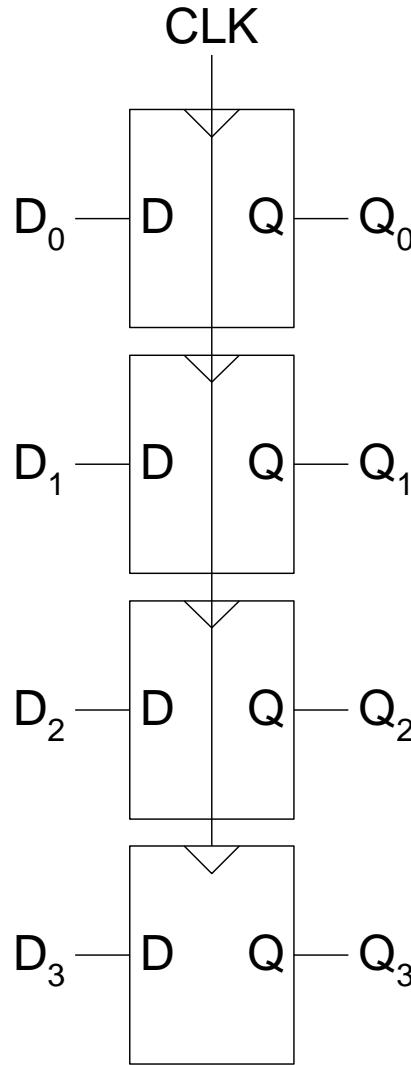
D Latch vs. D Flip-Flop



D Latch vs. D Flip-Flop

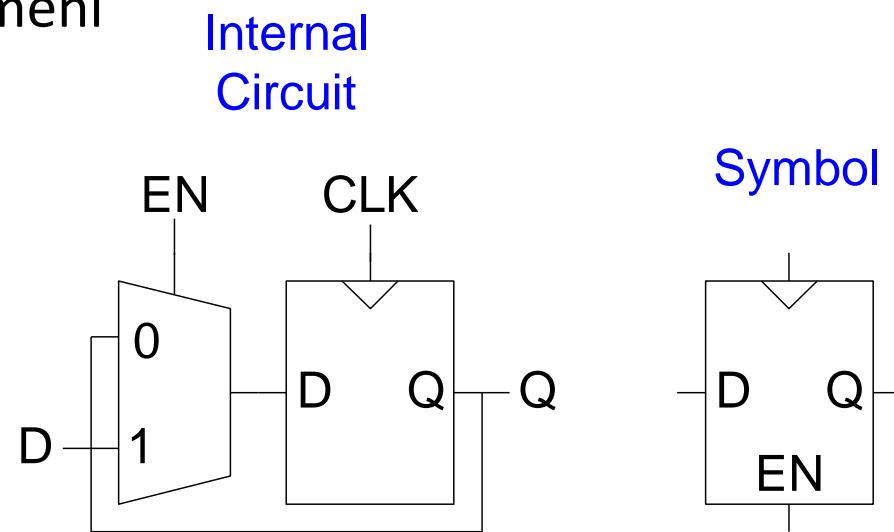


Registre



D FF so vstupom EN

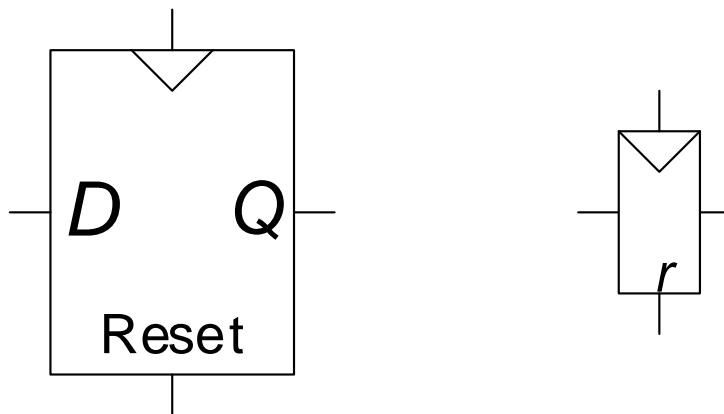
- **Vstupy:** CLK , D , EN
 - Vstup EN able riadi moment, kedy sa dostane dátový vstup D na výstup
- **Funkcia**
 - $EN = 1$: $Q = D$ pri nábehu (dobehu) CLK
 - $EN = 0$: Q sa nemení



D FF s možnosťou a. resetu

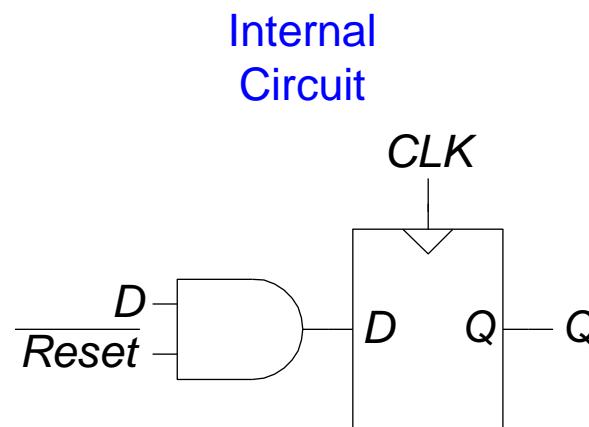
- **Vstupy:** CLK , D , *Reset*
- **Funkcia:**
 - *Reset* = 1: $Q = 0$
 - *Reset* = 0: pracuje v „normálnom“ režime

Symboly



D FF s možnosťou resetu

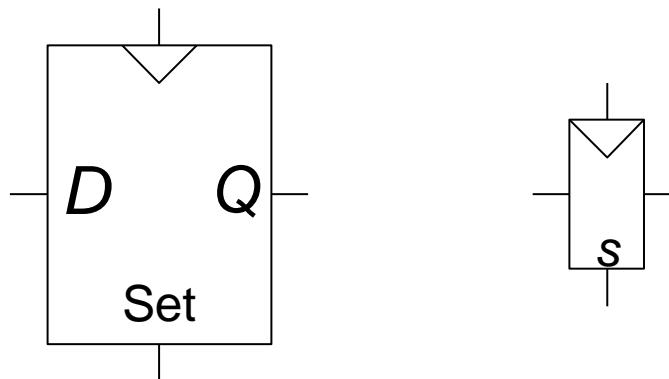
- Dva typy:
 - **Synchrónne**: len pri zmene CLK
 - **Asynchronné**: hned' čo sa nastaví $Reset = 1$



D FF s možnosťou nastavovania

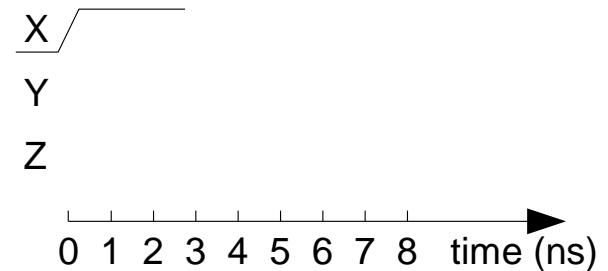
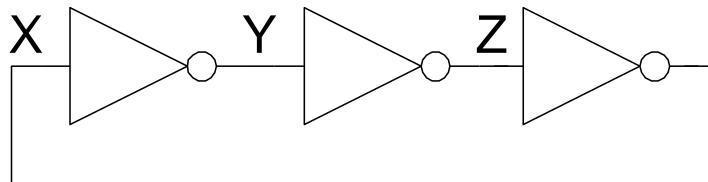
- **Vstupy:** CLK , D , Set
- **Funkcia:**
 - $Set = 1$: $Q = 1$
 - $Set = 0$: pracuje v „normálnom“ režime

Symbols



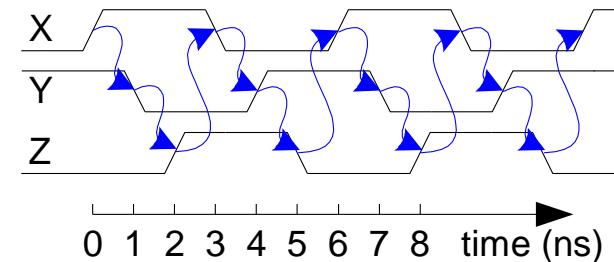
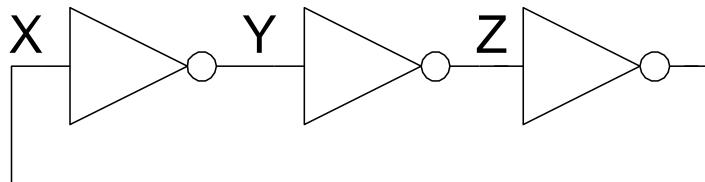
Sekvenčný LO

- Čo je SLO: to čo nie je KLO 😊
- A čo je toto?



Sekvenčný LO

- Čo je SLO: to čo nie je KLO 😊
- A čo je toto?



- Bez vstupov 3 „vnútorné“ signály
- Astabilný obvod → osciluje
- Periódna je závislá na charakteristike invertora
- Má to *slučku*: výstup pripojený na vstup

Synchrónny sekvenčný LO

- Do spätnoväzobnej slučky vkladáme registre
- Registre uchovávajú info. o vnútornom stave **systému**
- Stav sa mení pri zmene CLK: systém je **synchronizovaný** s hodinovým signálom
- Návrh synchrónnych sekvenčných LO
 - Každý prvak obvodu je buď KLO alebo register
 - Obsahuje aspoň jeden register
 - Rovnaký CLK pre všetky registre
 - V slučke je aspoň jeden register
- Typické synchrónne sekvenčné LO
 - Konečno stavového automaty
 - Prúdové funkčné jednotky (angl. pipelines)

Konečno-stavové automaty (KSA)

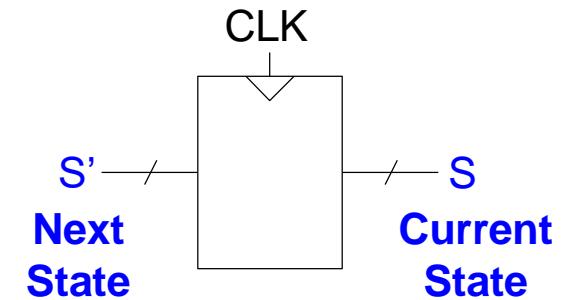
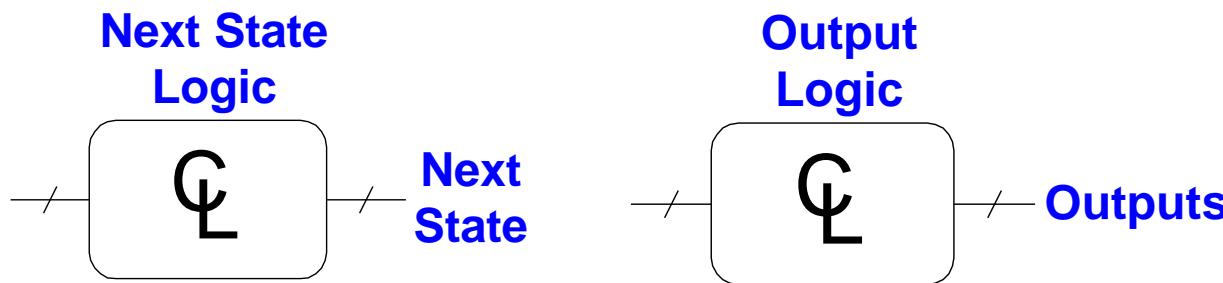
- Obsahuje:

- **Stavový register**

- Na uchovávanie stavu
 - Preklopenie stavov pri zmene CLK

- **Kombinačná logika**

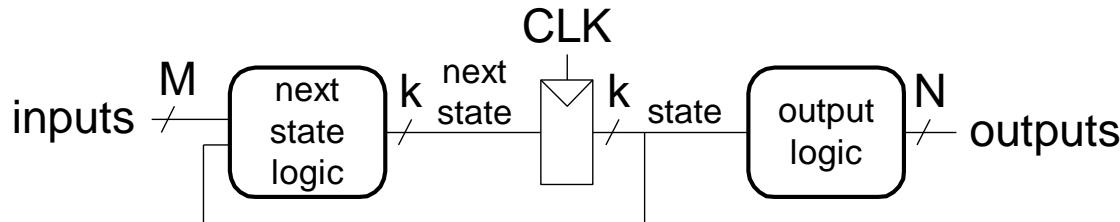
- Zodpovedná za budenie signálov, ktoré ovplyvňujú vnútorný stav systému
 - Zodpovedná za zmenu na výstupe



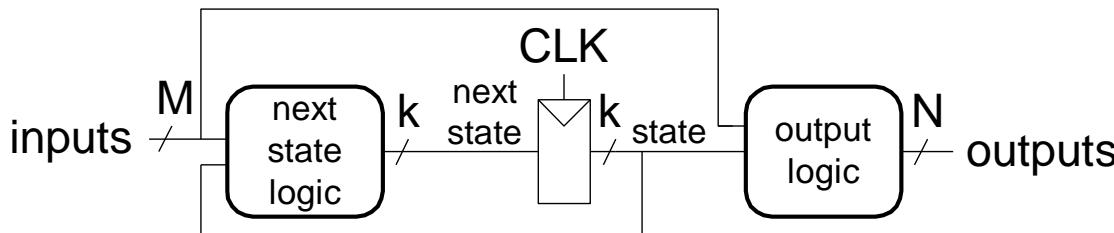
Konečno-stavové automaty

- Ďalší stav je definovaný aktuálnym vnútorným stavom systému a momentálnym stavom na vstupe
- Dva typy:
 - **Automat Moore**: výstup je závislý len od vnútorného stavu
 - **Automat Mealy**: výstup je závislý tak od vnútorného stavu ako aj od vstupu

Moore FSM

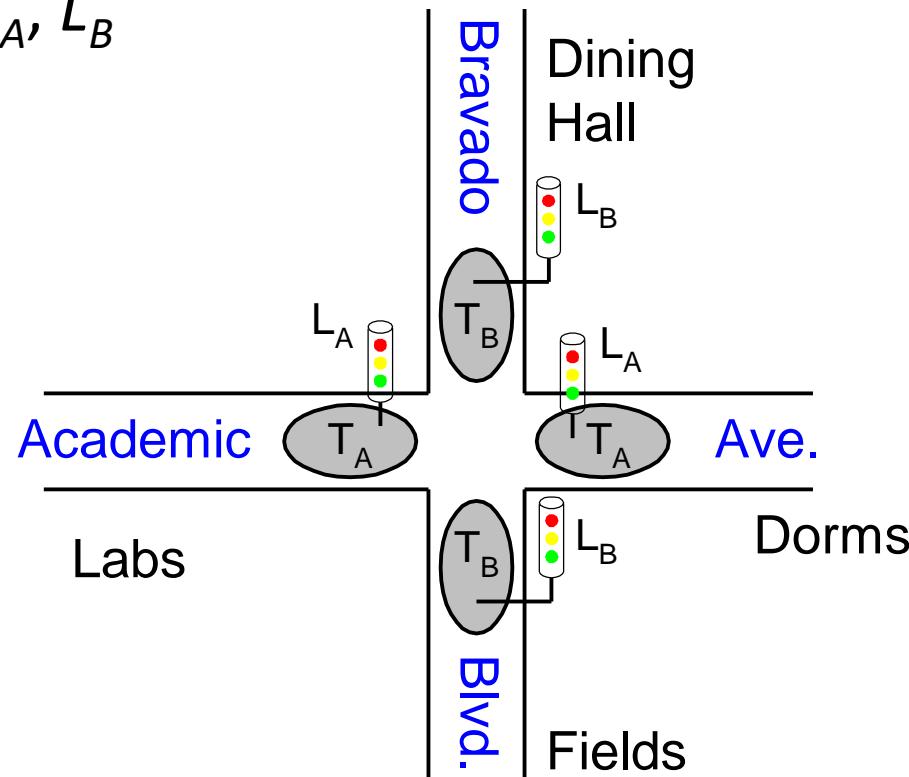


Mealy FSM



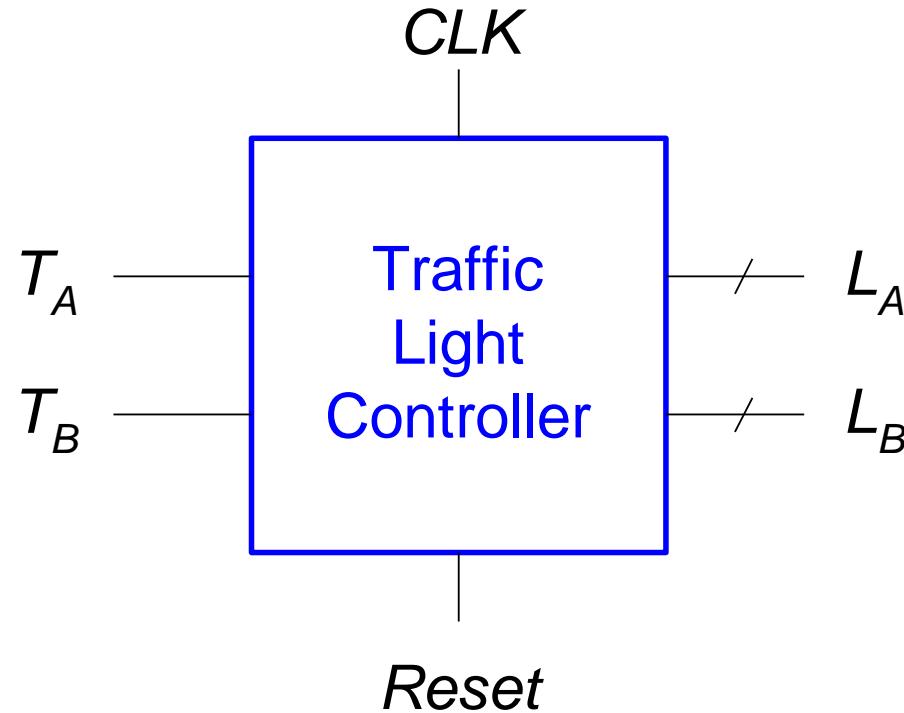
Príklad na KSA

- Riadenie križovatky (semafor+snímač)
 - Snímače: T_A, T_B (TRUE ak je zachytená premávka)
 - Semafore: L_A, L_B



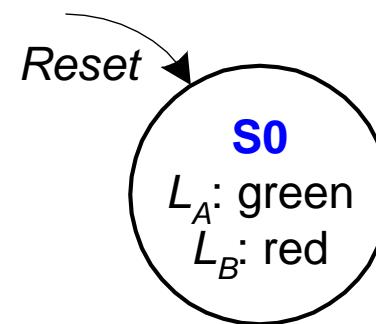
Čierna skrinka

- Vstupy: CLK , $Reset$, T_A , T_B
- Výstupy: L_A , L_B



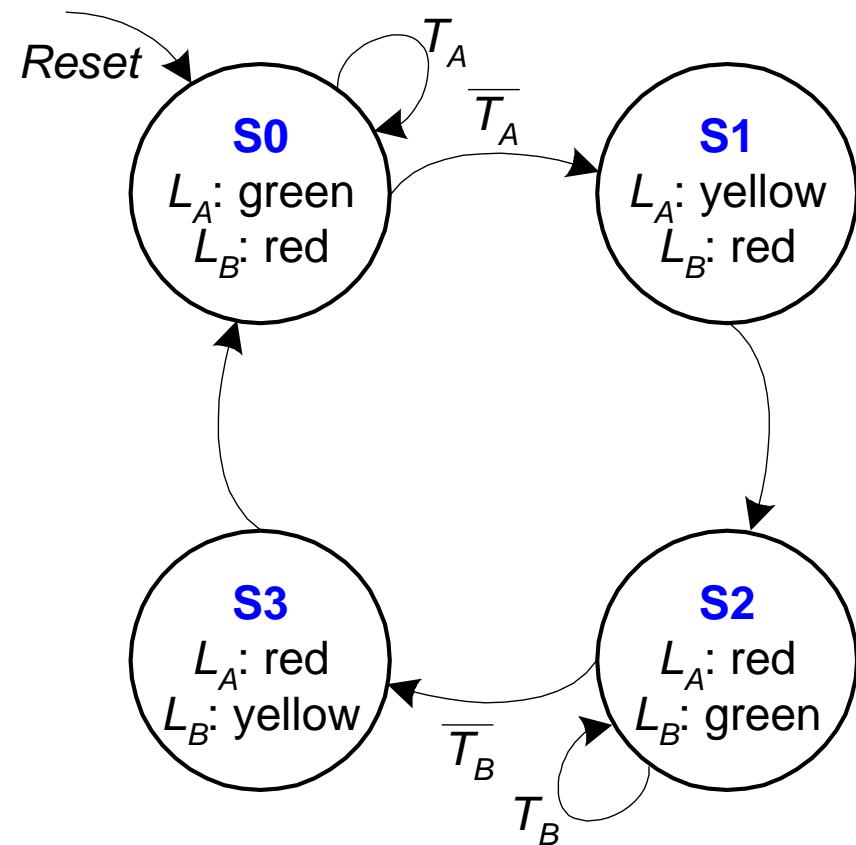
Graf prechodov

- **KSA Moore:** výstup je značený v uzle
- **Stavy:** Uzly
- **Prechody:** Hrany



Graf prechodov

- **KSA Moore:** výstup je značený v uzle
- **Stavy:** Uzly
- **Prechody:** Hrany



Tabuľka prechodov

Aktuálny stav S	Vstupy		Nasledujúci stav S'
	T_A	T_B	
S0	0	X	
S0	1	X	
S1	X	X	
S2	X	0	
S2	X	1	
S3	X	X	

Tabuľka prechodov

Aktuálny stav s	Vstupy		Nasledujúci stav s'
	T_A	T_B	
s_0	0	X	s_1
s_0	1	X	s_0
s_1	X	X	s_2
s_2	X	0	s_3
s_2	X	1	s_2
s_3	X	X	s_0

Kódovanie stavov

Aktuálny stav		Vstupy		Nasl. stav	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X		
0	0	1	X		
0	1	X	X		
1	0	X	0		
1	0	X	1		
1	1	X	X		

Stav	Kód
S0	00
S1	01
S2	10
S3	11

Kódovanie stavov

Aktuálny stav		Vstupy		Nasl. stav	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Stav	Kód
S0	00
S1	01
S2	10
S3	11

$$\begin{aligned}S'_1 &= S_1 \oplus S_0 \\S'_0 &= \overline{S_1} \overline{S_0} T_A + S_1 \overline{S_0} T_B\end{aligned}$$

Tabuľka výstupov

Aktuálny stav		Výstupy			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0				
0	1				
1	0				
1	1				

Výstup	Kód
green	00
yellow	01
red	10

Tabuľka výstupov

Aktuálny stav		Výstupy			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Výstup	Kód
green	00
yellow	01
red	10

$$L_{A1} = S_1$$

$$L_{A0} = \overline{S_1}S_0$$

$$L_{B1} = \overline{S_1}$$

$$L_{B0} = S_1S_0$$

Schéma KSA: Stavový register

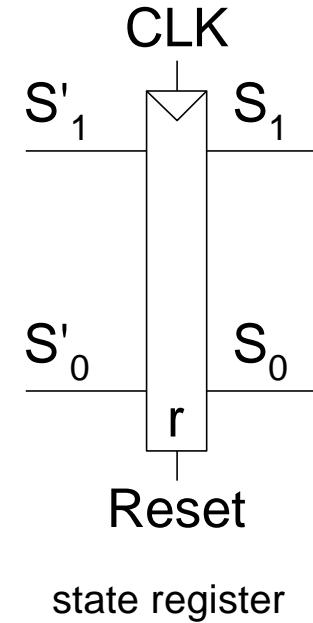


Schéma KSA: Logika prechodov

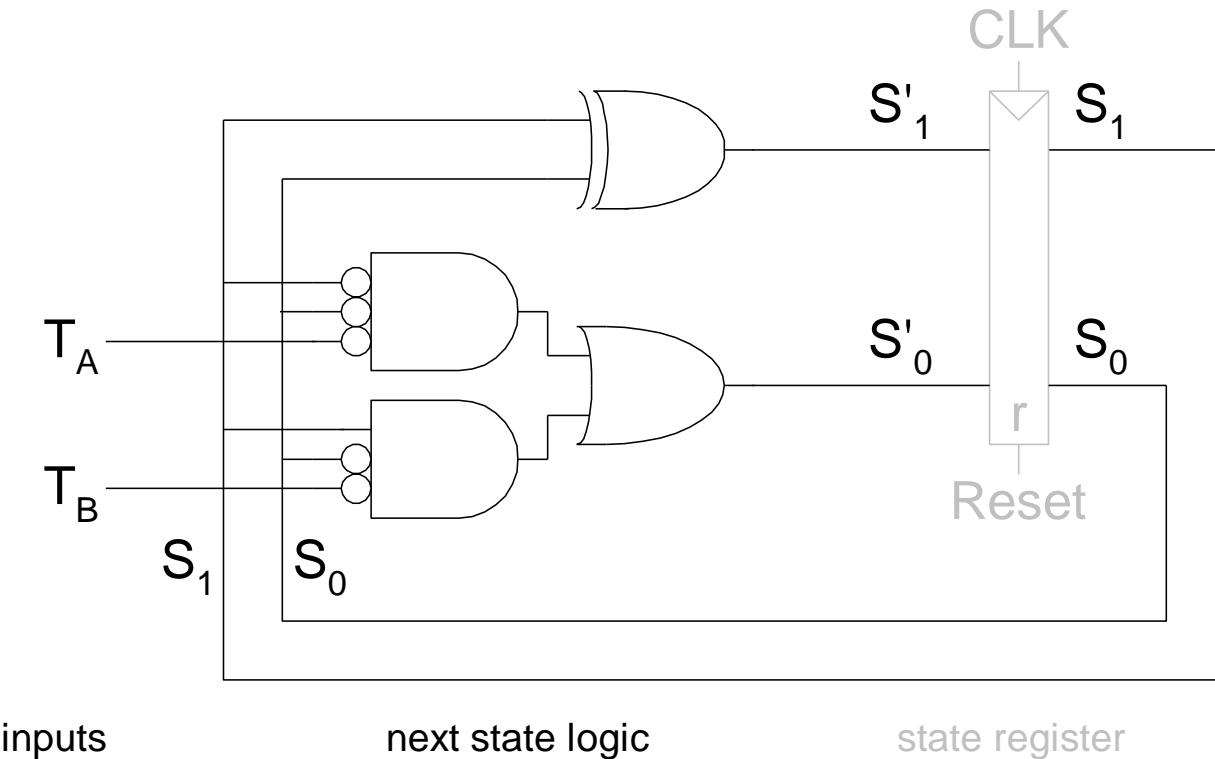
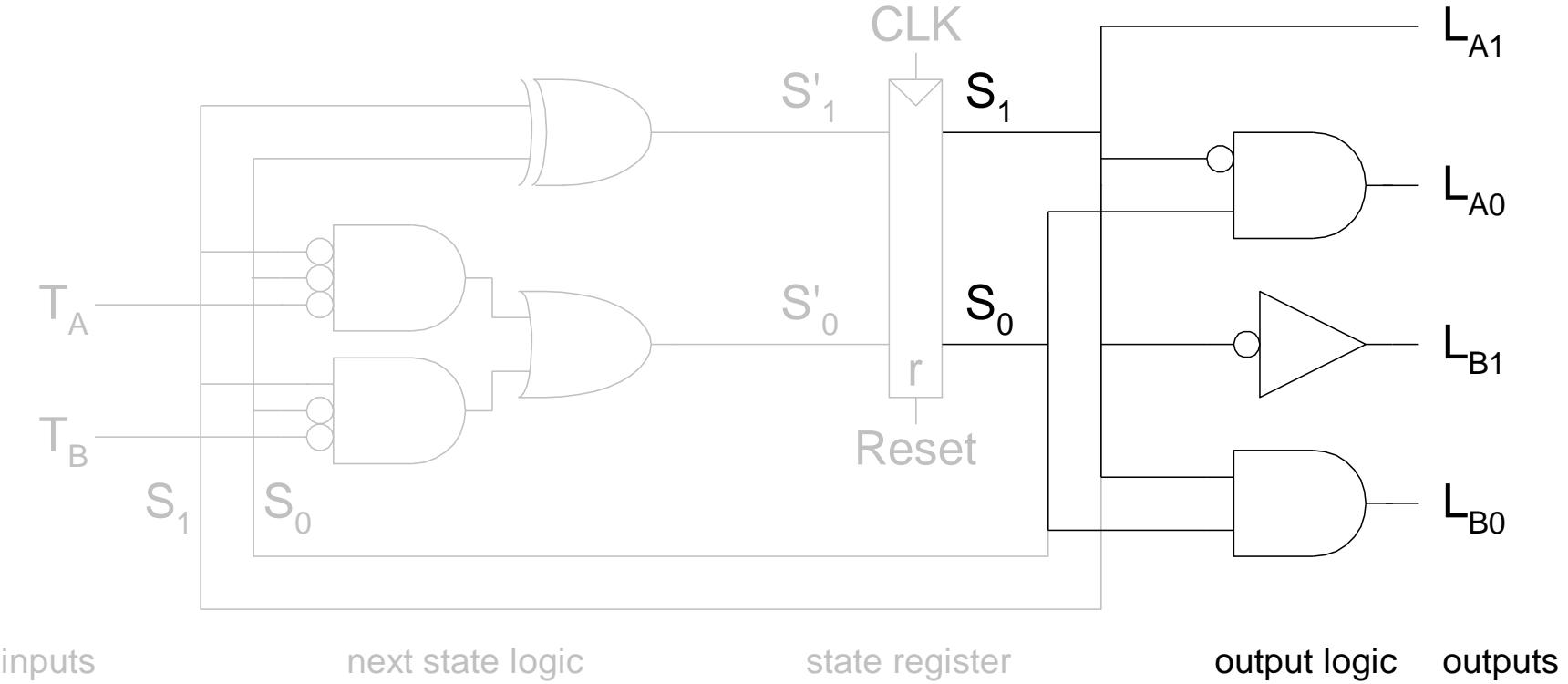
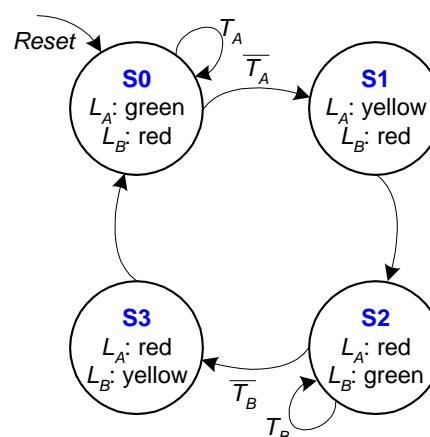
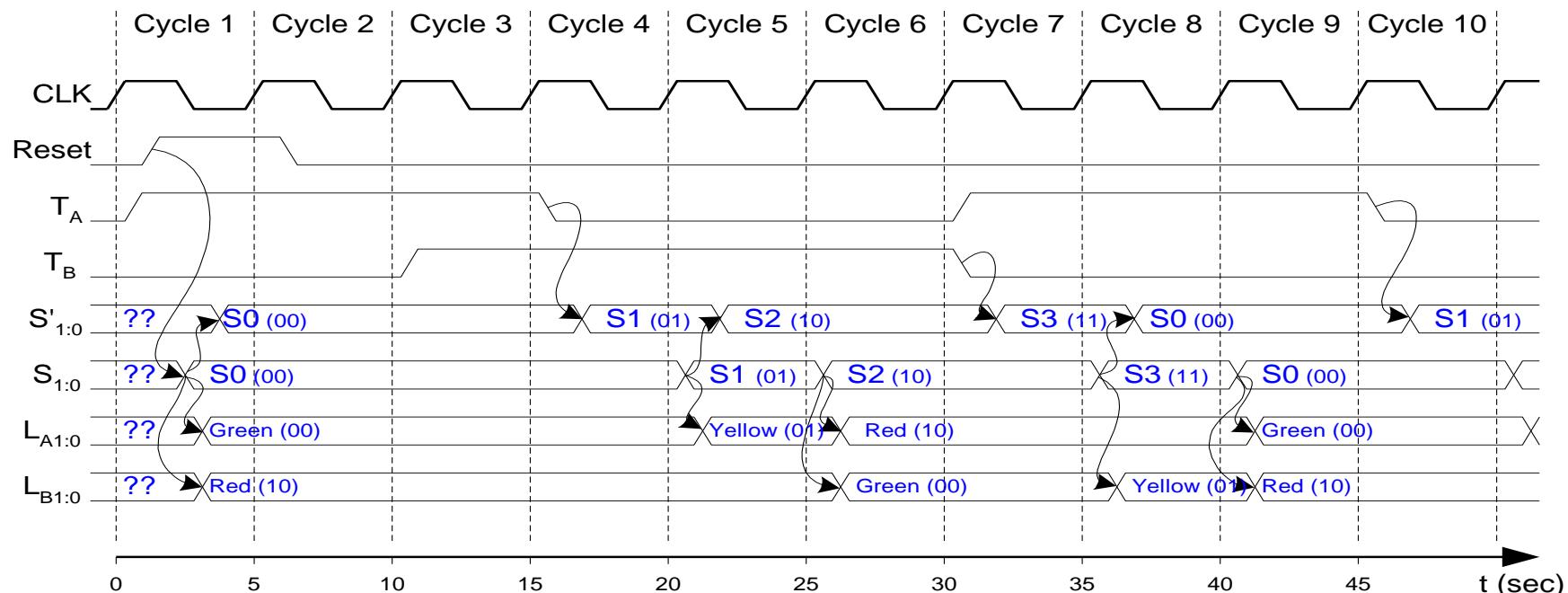


Schéma KSA: Logika výstupov



Priebeh signálov v KSA

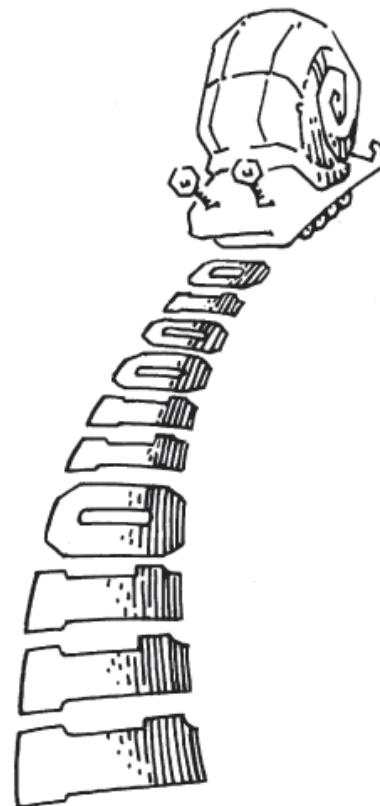


Kódovanie stavov v KSA

- **Binárne kódovanie:**
 - pre 4 stavy 2 bity: 00, 01, 10, 11
- **One-hot kódovanie**
 - pre 4 stavy 4 bity, 0001, 0010, 0100, 1000
 - Len jeden bit je nastavený na 1 ostatné sú 0
 - vyžaduje viac PO
 - Budiace a výstupné funkcie sú zväčša „jednoduchšie“

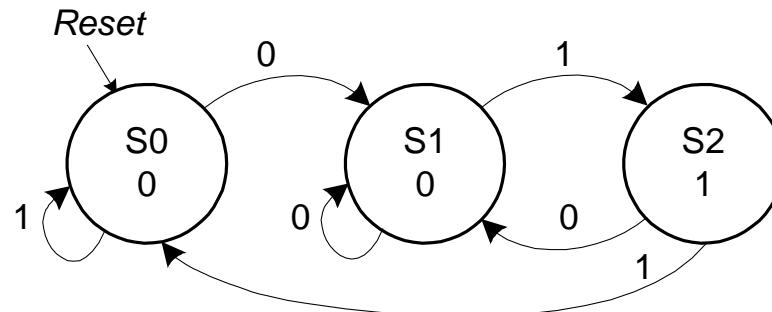
Moore vs. Mealy

- Slimák lezie po papieri na ktorom sú nakreslené nulky a jedničky. Zakaždým, čo slimák prechádza cez kombináciu 01, zasmeje sa. Modelujte mozog slimáka na báze automatov Moore a Mealy.

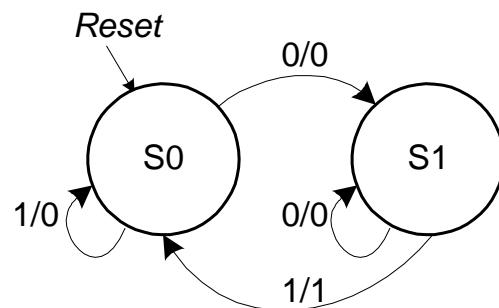


Graf prechodov

Moore FSM



Mealy FSM



KSA typu Mealy: hrany sú označované hodnotou vstupu/výstupu

Tabuľka prechodov: KSA Moore

Akt. stav		Vstup	Nasl. stav	
s_1	s_0	A	s'_1	s'_0
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		

Stav	Kód
S0	00
S1	01
S2	10

Tabuľka prechodov: KSA Moore

Akt. stav		Vstup	Nasl. stav	
s_1	s_0	A	s'_1	s'_0
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0

Stav	Kód
S0	00
S1	01
S2	10

$$\begin{aligned}s'_1 &= s_0 A \\ s'_0 &= \bar{A}\end{aligned}$$

Tabuľka výstupov: KSA Moore

Aktuálny stav		Výstup
S_1	S_0	Y
0	0	
0	1	
1	0	

Tabuľka výstupov: KSA Moore

Aktuálny stav		Výstup
S_1	S_0	Y
0	0	0
0	1	0
1	0	1

$$Y = S_1$$

Tabuľka prechodov a výstupov: KSA Mealy

Akt. stav	Vstup	Nasl. stav	Výstup
s_0	A	s'_0	γ
0	0		
0	1		
1	0		
1	1		

Stav	Kód
s_0	00
s_1	01

Tabuľka prechodov a výstupov: KSA Mealy

Akt. stav	Vstup	Nasl. stav	Výstup
s_0	A	s'_0	γ
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

Stav	Kód
s_0	00
s_1	01

Schéma KSA typu Moore

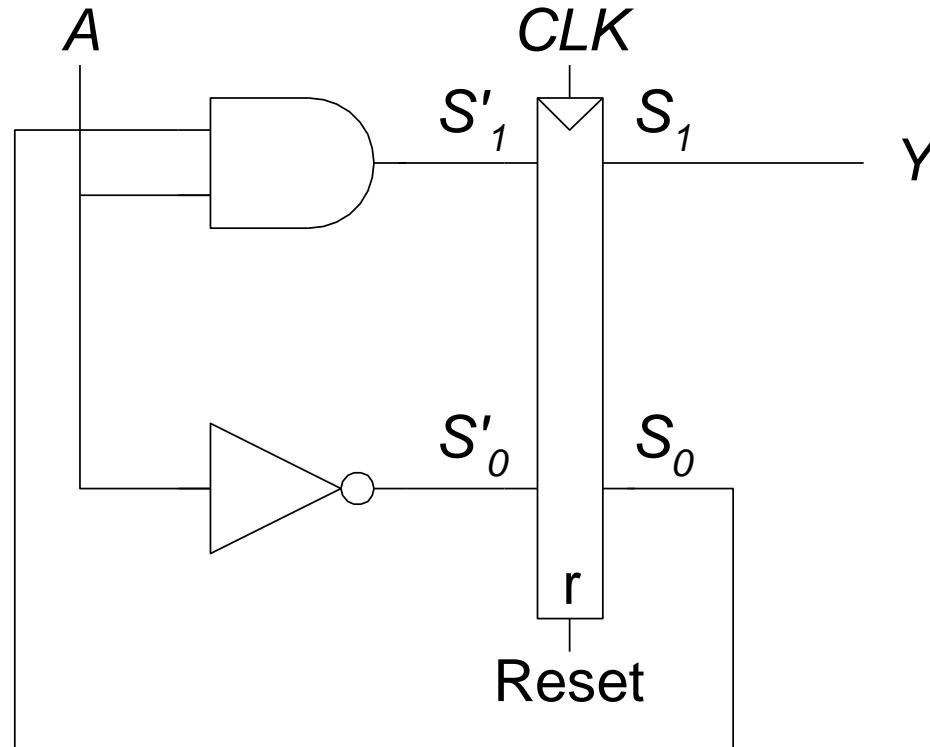
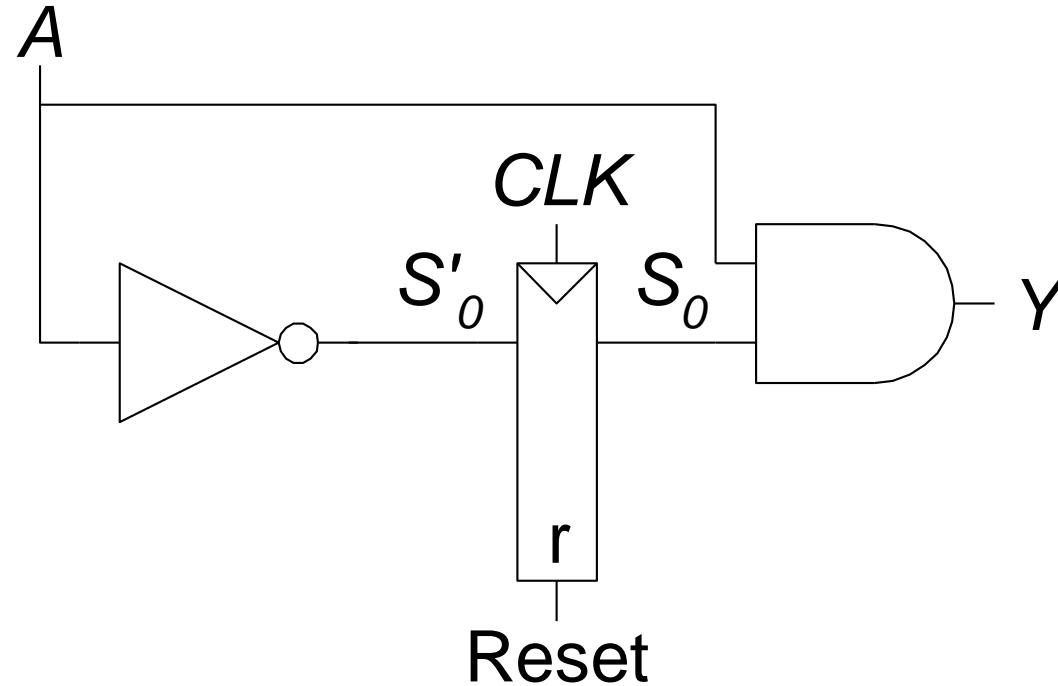
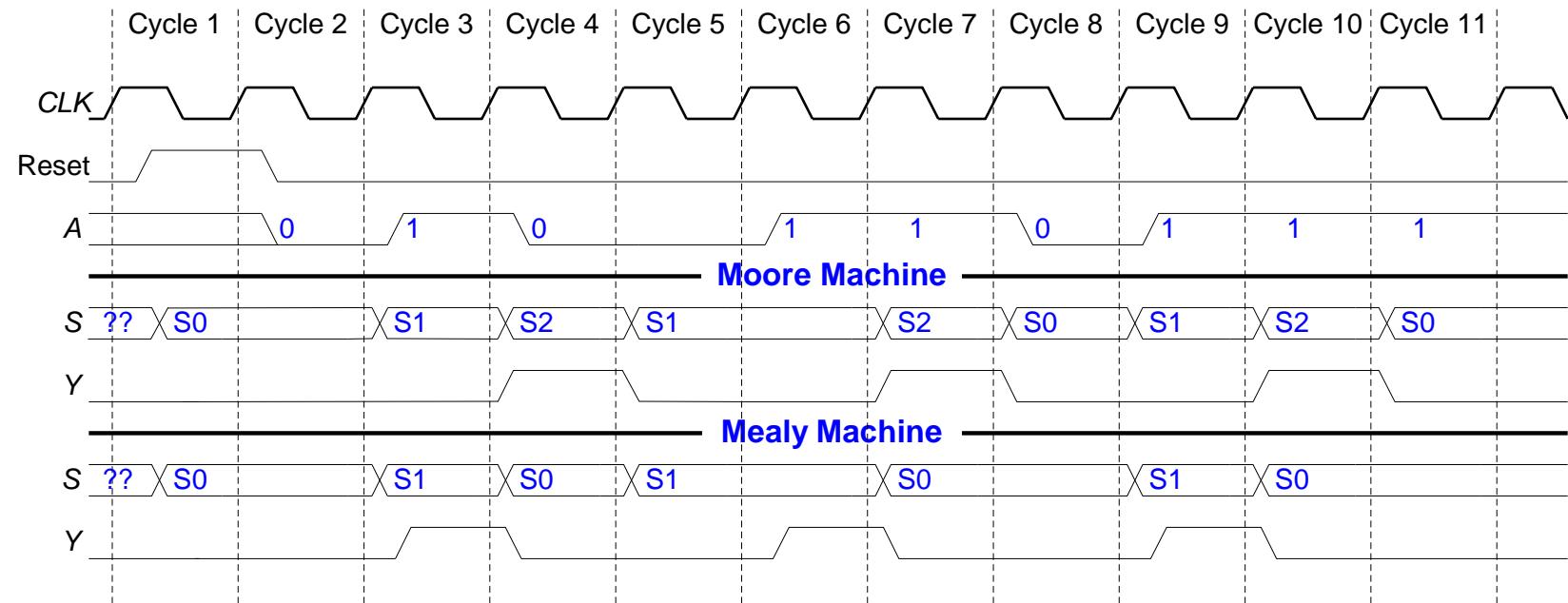


Schéma KSA typu Mealy



Priebeh signálov v KSA



Faktorizácia KSA

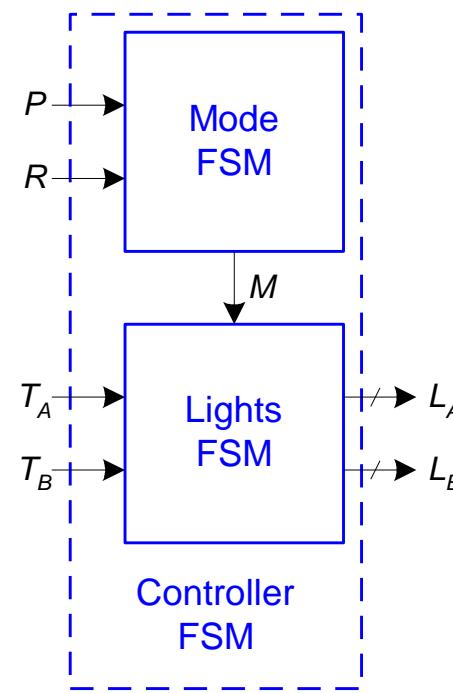
- Rozdelenie „veľkého“ KSA na menšie interagujúce KSA
- Príklad: Riadenie križovatky s funkciou festivalu.
 - Dva ďalšie vstupy: P , R
 - Ak $P = 1$, aktivuj režim festivalu a zablokuj Academic Ave.
 - Ak $R = 1$, deaktivuj režim festivalu

KSA s festivalovým režimom

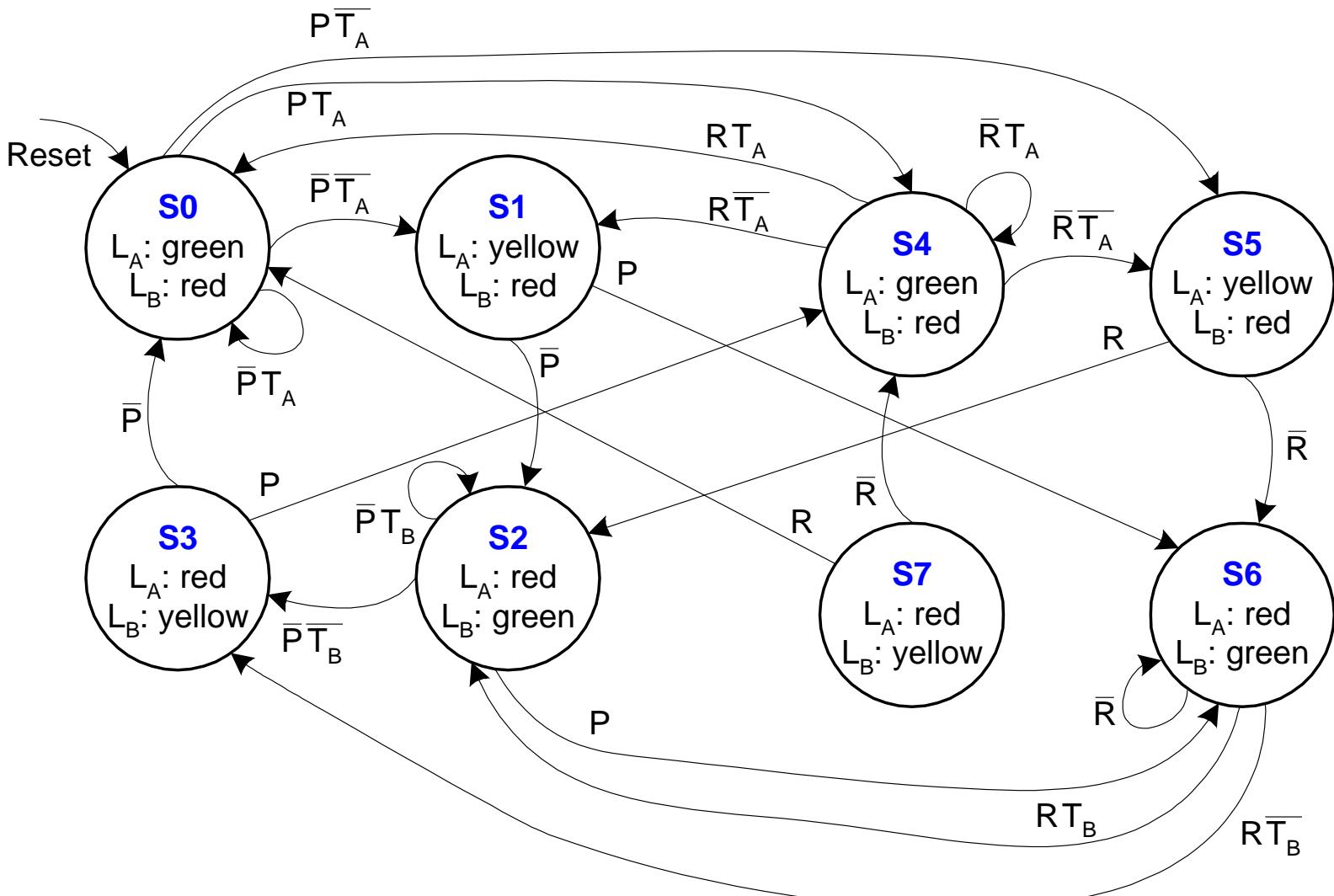
KSA



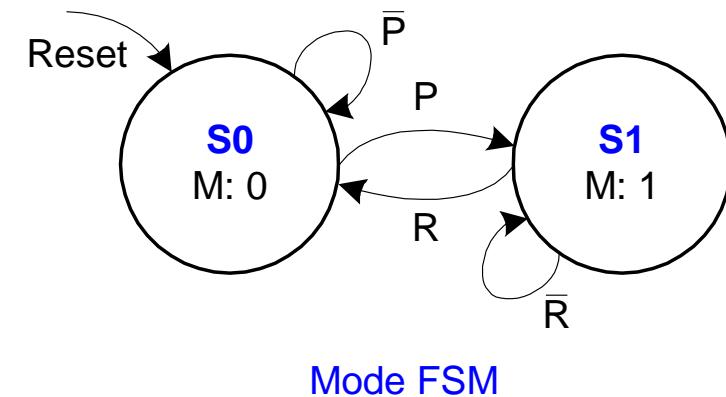
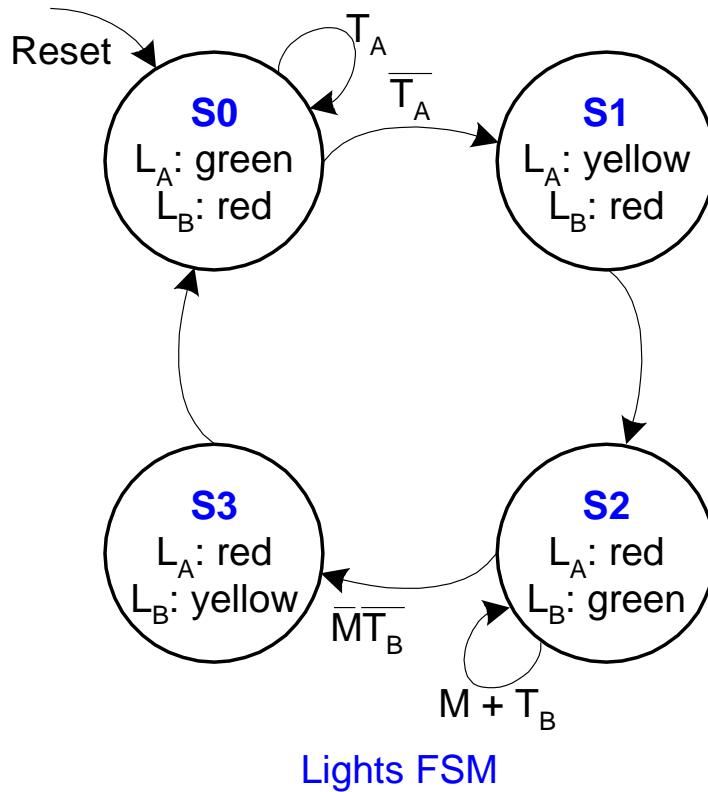
Faktorizovaný KSA



KSA



Faktorizovaný KSA



Syntéza KSA

- Urč vstupy a výstupy
- Načrtni graf prechodov
- Zostav tabuľku prechodov a výstupov
- Zvol kódovanie
- Pre KSA typu Moore:
 - Aplikuj zvolené kódovanie na tabuľku prechodov
 - Aplikuj zvolené kódovanie na tabuľku výstupov
- Pre KSA typu Mealy:
 - Aplikuj zvolené kódovanie na tabuľku prechodov a výstupov
- Odvod' budiace a výstupné funkcie
- Nakresli schému LO

Časové charakteristiky SLO

- Hranou riadený PO
 - Pre správnu činnosť PO je nutné dodržať niektoré dynamické parametre vstupných signálov:
 - doba predstihu,
 - doba presahu,
 - minimálna strmost' hrany hodinového impulzu,
 - minimálna doba jeho trvania

Časové charakteristiky SLO

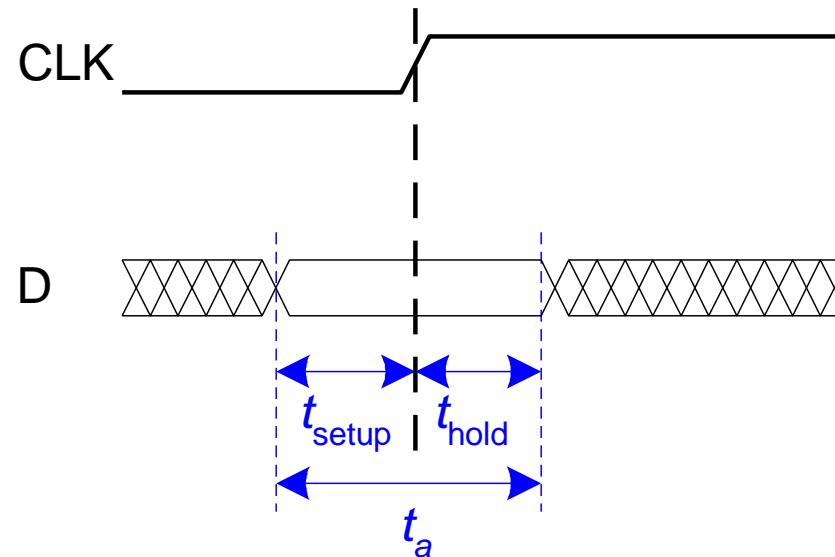
- Hranou riadený PO
 - Pre správnu činnosť PO je nutné dodržať niektoré dynamické parametre vstupných signálov:
 - doba nastavenia (predstihu),
 - doba presahu,
 - minimálna strmost' hrany hodinového impulzu,
 - minimálna doba jeho trvania
 - Nedodržanie niektorých z týchto podmienok môže vyvolať metastabilný stav, kedy výsledné preklopenie je náhodné.

Časové charakteristiky SLO

- Základné vlastnosti hranou riadeného PO
 - Rozhodujúci je stav na vstupoch v tesnom okolí nábežnej hrany hodinového impulzu.
 - V iných okamžikoch sú zmeny na vstupoch bezvýznamné.
 - Nevhodné časovanie hodinových impulzov a vstupných signálov môže vyvolať metastabilný stav.
 - Metastabilný stav má za následok nespoľahlivú funkciu preklápacieho obvodu. PO sa môže rozkmitať, alebo sa môže preklopiť do nesprávneho stavu, alebo sa môže preklopiť do správneho stavu s veľkým oneskorením. Dobu oneskorenia nie je možné predvídať.

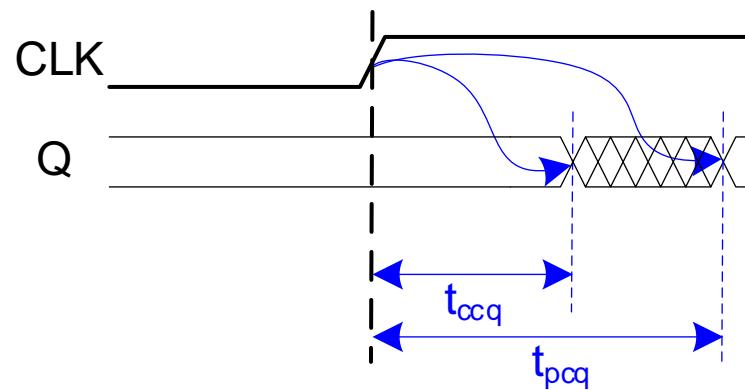
Časovanie signálov v SLO

- **Doba nastavenia:** t_{setup} = časový úsek pred zmenou CLK, počas ktorého vstup by mal byť stabilný (t.j. nemenný)
- **Doba presahu:** t_{hold} = časový úsek po zmene CLK, počas ktorého vstup by mal byť stabilný
- **Vzorkovací interval:** t_a = v tomto časovom intervale sa nesmie zmeniť hodnota vstupu ($t_a = t_{\text{setup}} + t_{\text{hold}}$)



Časovanie signálov v SLO

- Propagačné oneskorenie: t_{pcq} = časový úsek (začínajúci so zmenou CLK a) po uplynutí ktorého je garantované, že Q má stabilnú (ustálenú) hodnotu
- Kontaminačné oneskorenie: t_{ccq} = časový úsek (začínajúci so zmenou CLK a) po uplynutí ktorého sa začína prejavovať zmena na Q (tj. ešte nemusí mať ustálenú hodnotu)

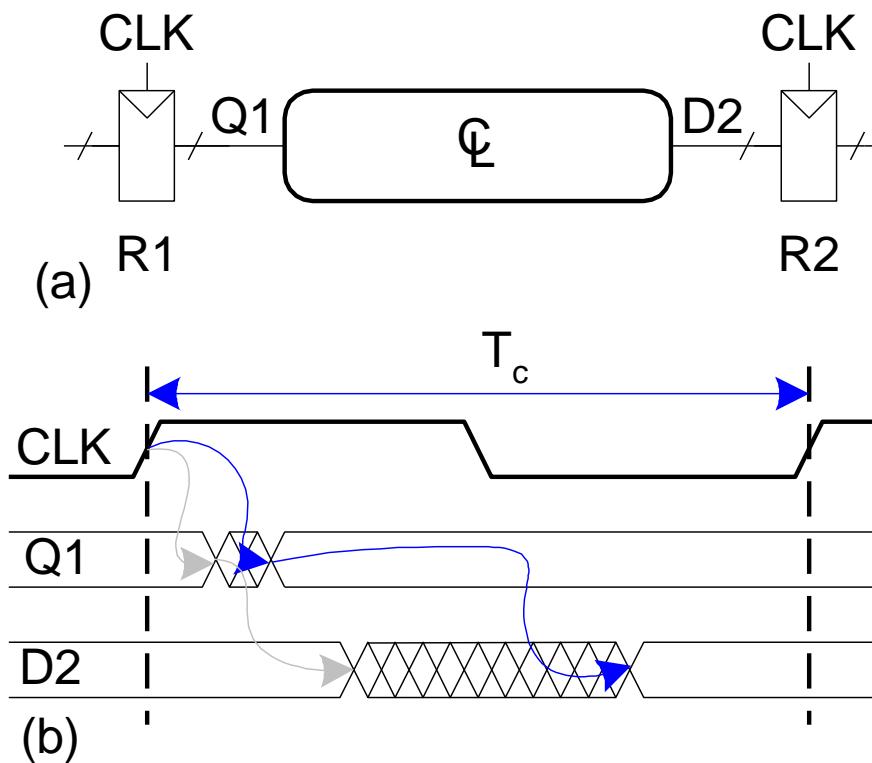


Časovanie signálov v SLO

- Vstupy do SLO musia byť stabilné počas vzorkovania
 - minimálne počas doby t_{setup} pred zmenou CLK
 - minimálne počas doby t_{hold} po zmene CLK

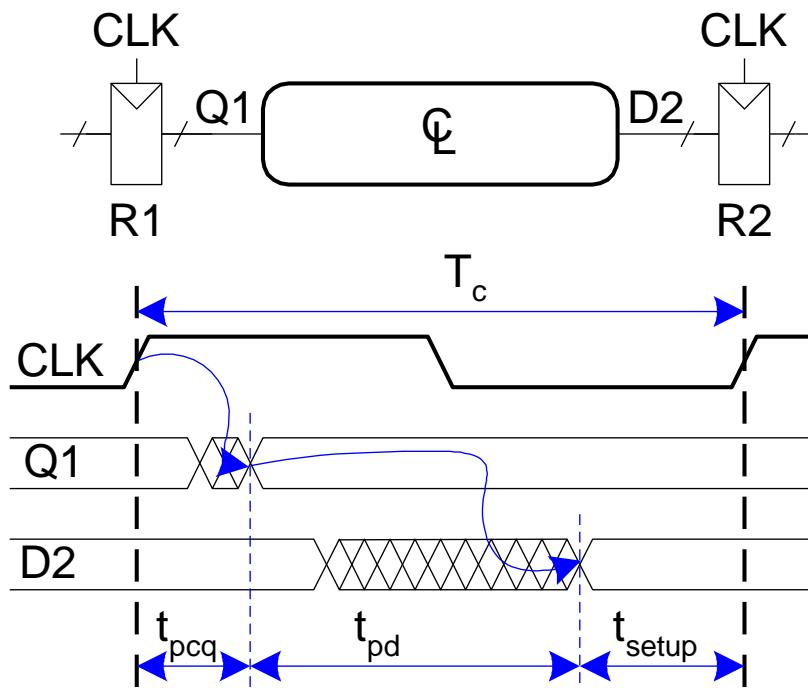
Časovanie signálov v SLO

- Signál, kt. prechádza cez KLO medzi registrami je charakterizovaný **kontaminačným** (minimálnym) a **propagačným** (maximálnym) oneskorením



Doba nastavenia

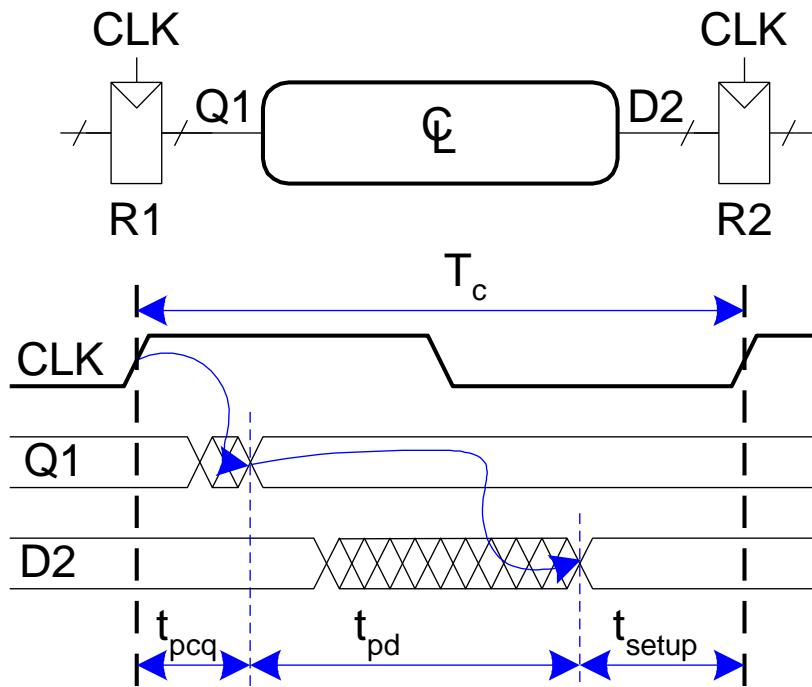
- Závisí od **propagačného oneskorenia** signálu, ktorý prechádza cez KLO spájajúci registre R1 a R2
- Vstup do R2 musí byť stabilný počas doby t_{setup} pred zmenou CLK



$$T_c \geq$$

Doba nastavenia

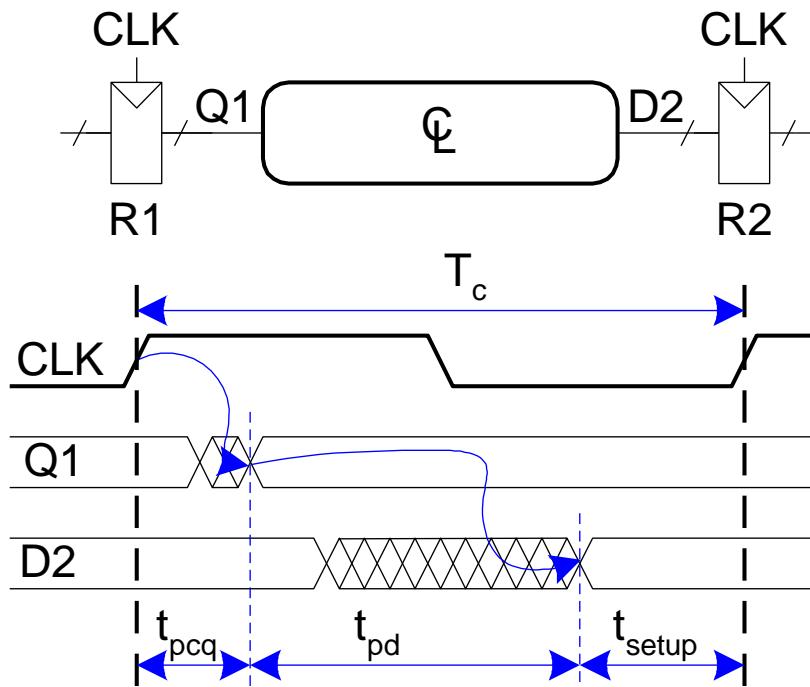
- Závisí od **propagačného oneskorenia** signálu, ktorý prechádza cez KLO spájajúci registre R1 a R2
- Vstup do R2 musí byť stabilný počas doby t_{setup} pred zmenou CLK



$$T_c \geq t_{\text{pcq}} + t_{\text{pd}} + t_{\text{setup}}$$
$$t_{\text{pd}} \leq$$

Doba nastavenia

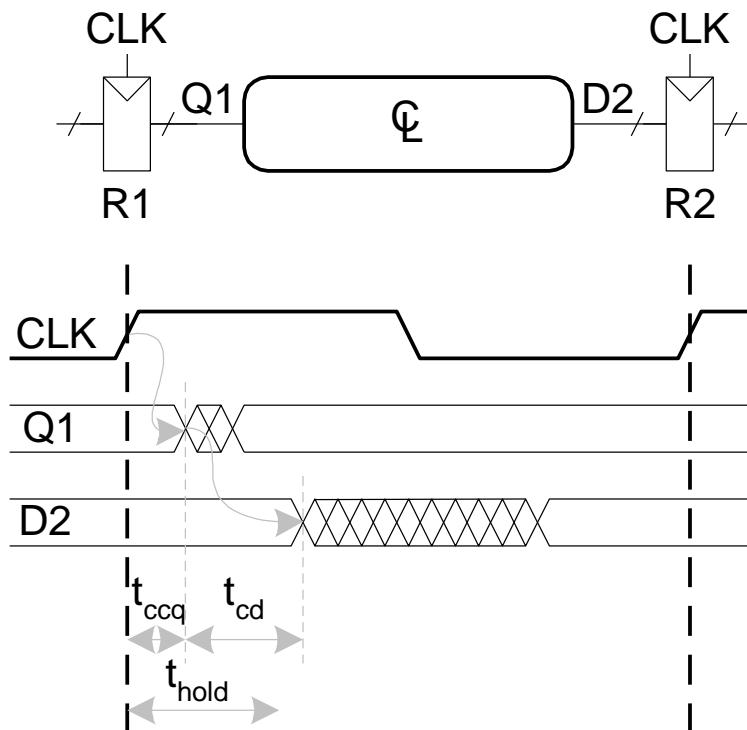
- Závisí od **propagačného oneskorenia** signálu, ktorý prechádza cez KLO spájajúci registre R1 a R2
- Vstup do R2 musí byť stabilný počas doby t_{setup} pred zmenou CLK



$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}}$$
$$t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}})$$

Doba presahu

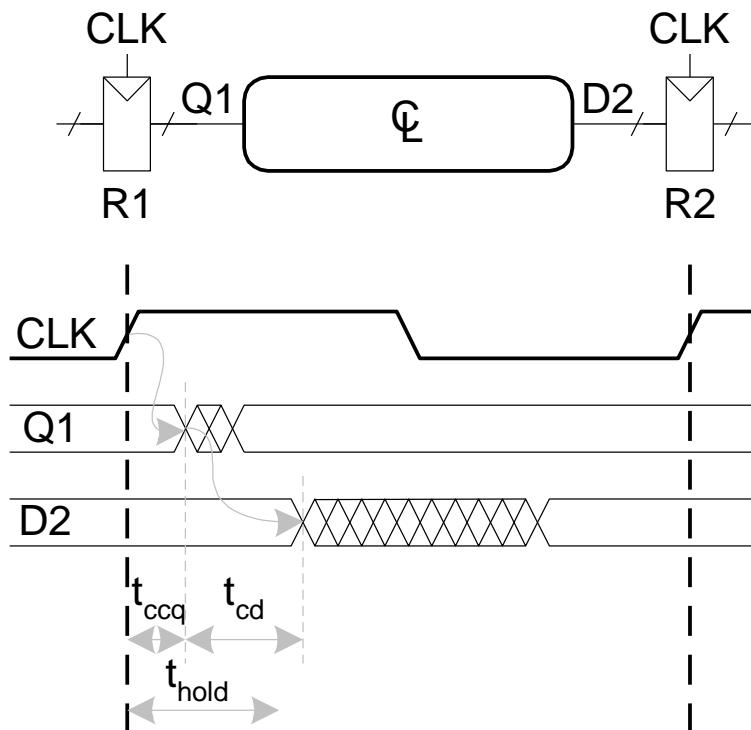
- Závisí od kontaminačného oneskorenia signálu, ktorý prechádza cez KLO spájajúci registre R1 a R2
- Vstup do R2 musí byť stabilný počas doby t_{hold} po zmene CLK



$t_{hold} <$

Doba presahu

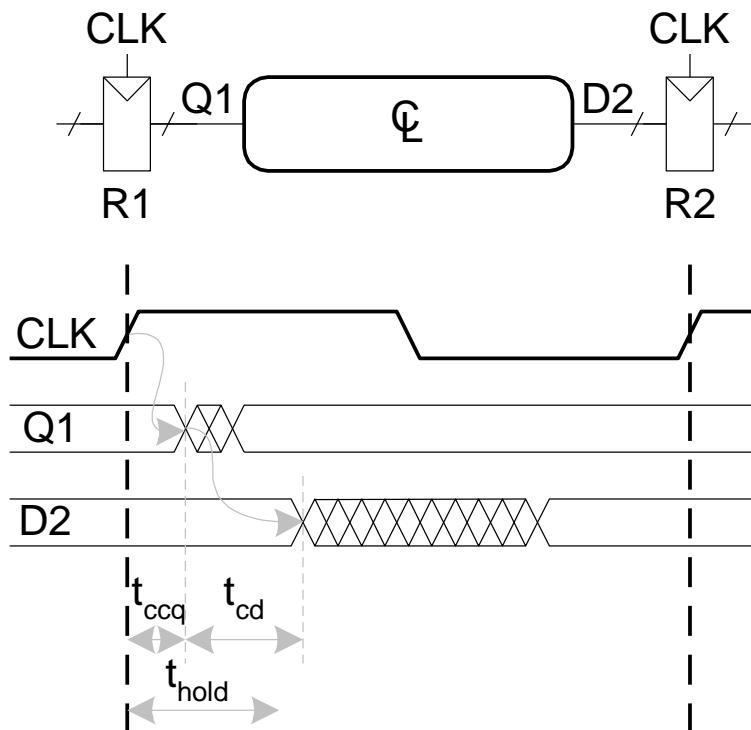
- Závisí od kontaminačného oneskorenia signálu, ktorý prechádza cez KLO spájajúci registre R1 a R2
- Vstup do R2 musí byť stabilný počas doby t_{hold} po zmene CLK



$$t_{hold} < t_{ccq} + t_{cd}$$
$$t_{cd} >$$

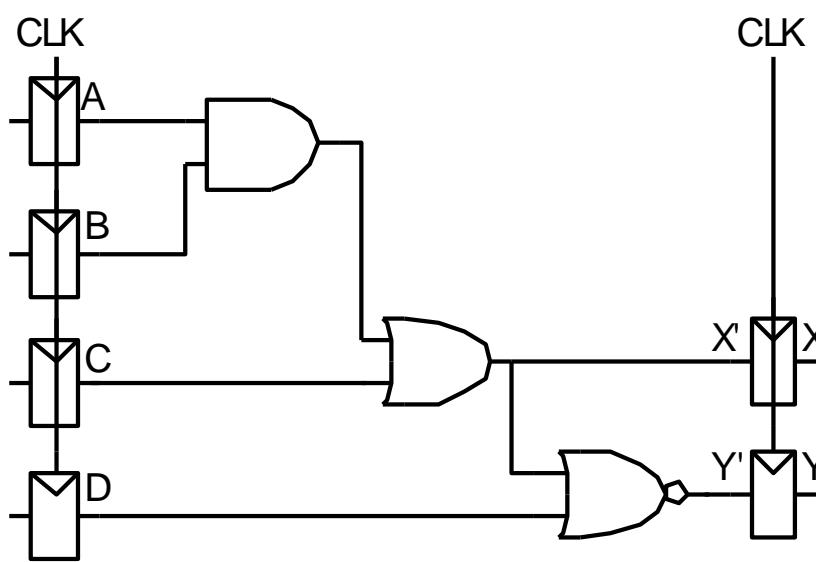
Doba presahu

- Závisí od kontaminačného oneskorenia signálu, ktorý prechádza cez KLO spájajúci registre R1 a R2
- Vstup do R2 musí byť stabilný počas doby t_{hold} po zmene CLK



$$t_{hold} < t_{ccq} + t_{cd}$$
$$t_{cd} > t_{hold} - t_{ccq}$$

Príklad



$$t_{pd} =$$

$$t_{cd} =$$

Doba nastavenia:

$$T_c \geq$$

$$f_c =$$

Časové charakteristiky

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{\text{setup}} = 60 \text{ ps}$$

$$t_{\text{hold}} = 70 \text{ ps}$$

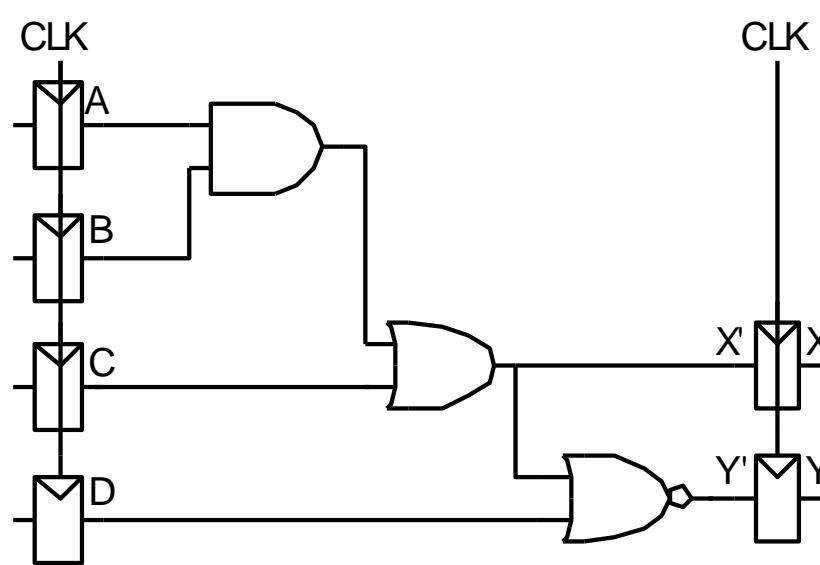
per hradlo

t_{pd}	= 35 ps
t_{cd}	= 25 ps

Doba presahu

$$t_{ccq} + t_{cd} > t_{\text{hold}} ?$$

Príklad



$$t_{pd} = 3 \times 35 \text{ ps} = 105 \text{ ps}$$

$$t_{cd} = 25 \text{ ps}$$

Doba nastavenia:

$$T_c \geq (50 + 105 + 60) \text{ ps} = 215 \text{ ps}$$

$$f_c = 1/T_c = 4.65 \text{ GHz}$$

Časové charakteristiky

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{\text{setup}} = 60 \text{ ps}$$

$$t_{\text{hold}} = 70 \text{ ps}$$

per hradlo

$$\begin{cases} t_{pd} &= 35 \text{ ps} \\ t_{cd} &= 25 \text{ ps} \end{cases}$$

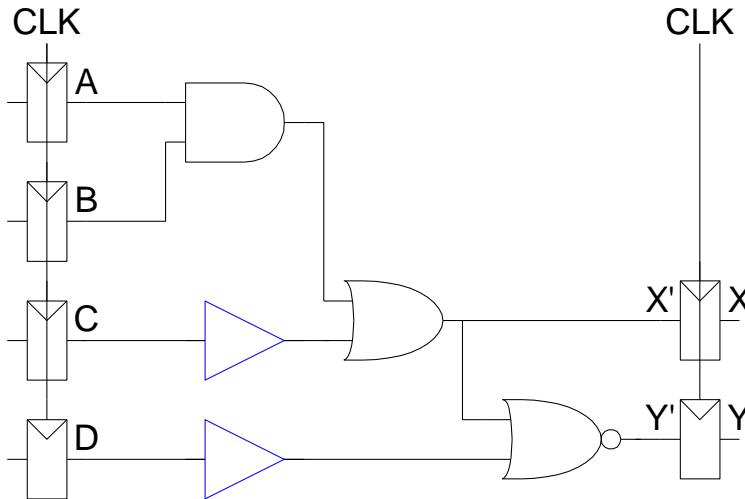
Doba presahu:

$$t_{ccq} + t_{cd} > t_{\text{hold}} ?$$

$$(30 + 25) \text{ ps} > 70 \text{ ps} ? \text{ No!}$$

Príklad

Pridaním budičov do najkratšej cesty



$$t_{pd} =$$

$$t_{cd} =$$

Doba nastavenia:

$$T_c \geq$$

$$f_c =$$

Časové charakteristiky

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{\text{setup}} = 60 \text{ ps}$$

$$t_{\text{hold}} = 70 \text{ ps}$$

per hradlo

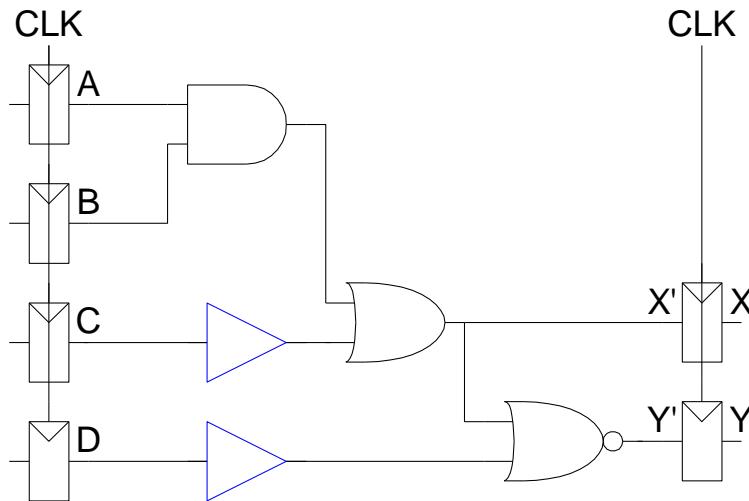
$$\begin{cases} t_{pd} &= 35 \text{ ps} \\ t_{cd} &= 25 \text{ ps} \end{cases}$$

Doba presahu:

$$t_{ccq} + t_{cd} > t_{\text{hold}} ?$$

Príklad

Pridaním budičov do najkratšej cesty



$$t_{pd} = 3 \times 35 \text{ ps} = 105 \text{ ps}$$

$$t_{cd} = 2 \times 25 \text{ ps} = 50 \text{ ps}$$

Doba nastavenia:

$$T_c \geq (50 + 105 + 60) \text{ ps} = 215 \text{ ps}$$

$$f_c = 1/T_c = 4.65 \text{ GHz}$$

Časové charakteristiky

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{\text{setup}} = 60 \text{ ps}$$

$$t_{\text{hold}} = 70 \text{ ps}$$

per hradlo

$$\begin{cases} t_{pd} &= 35 \text{ ps} \\ t_{cd} &= 25 \text{ ps} \end{cases}$$

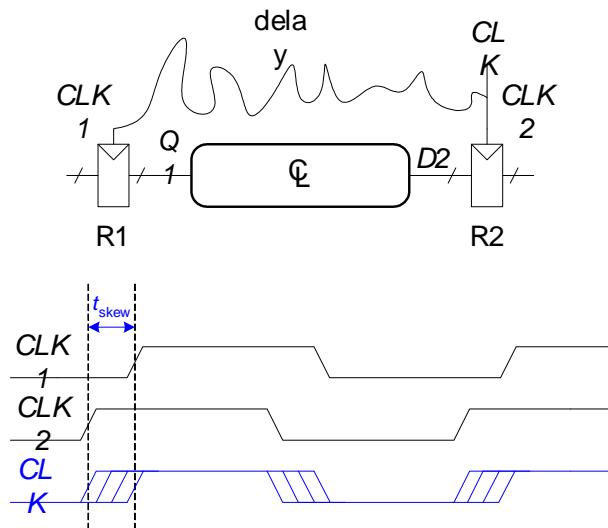
Doba presahu:

$$t_{ccq} + t_{cd} > t_{\text{hold}} ?$$

$$(30 + 50) \text{ ps} > 70 \text{ ps} ? \text{ Yes!}$$

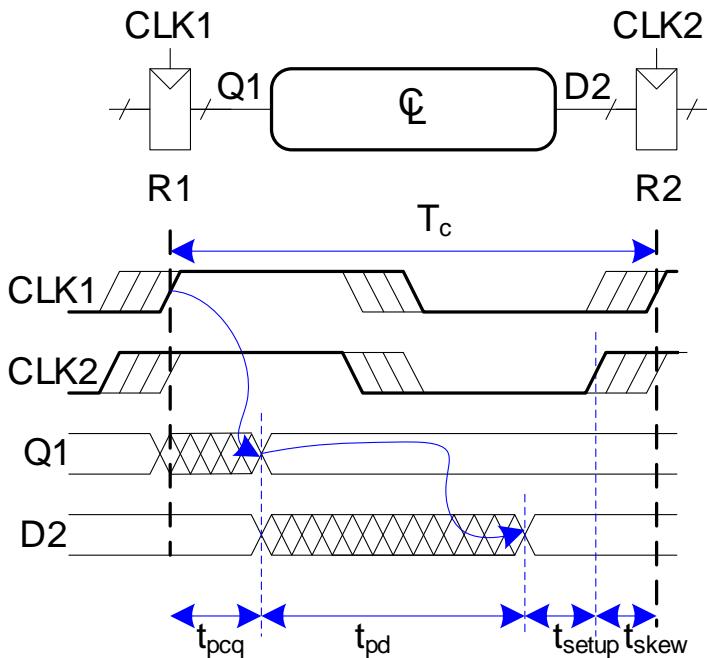
Odchýlka v časovaní

- Hodinový impulz prichádzajúci na vstupy rôznych registrov nie v rovnakom momente.
 - Vzniká odchýlka
- Analyzuje sa najhorší možný prípad



Časovanie & Čas nastavenia

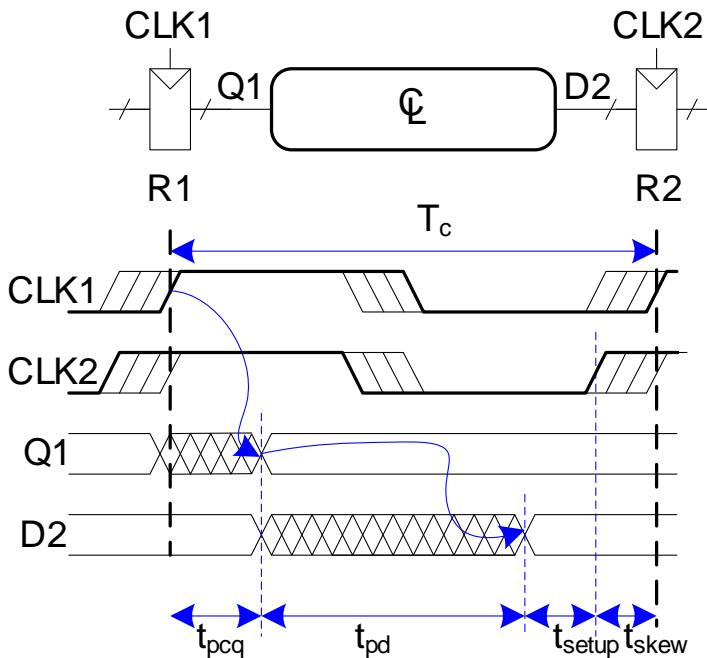
- Najhorší možný prípad: zmena na CLK2 nastane skôr než na CLK1



$$T_c \geq$$

Časovanie & Čas nastavenia

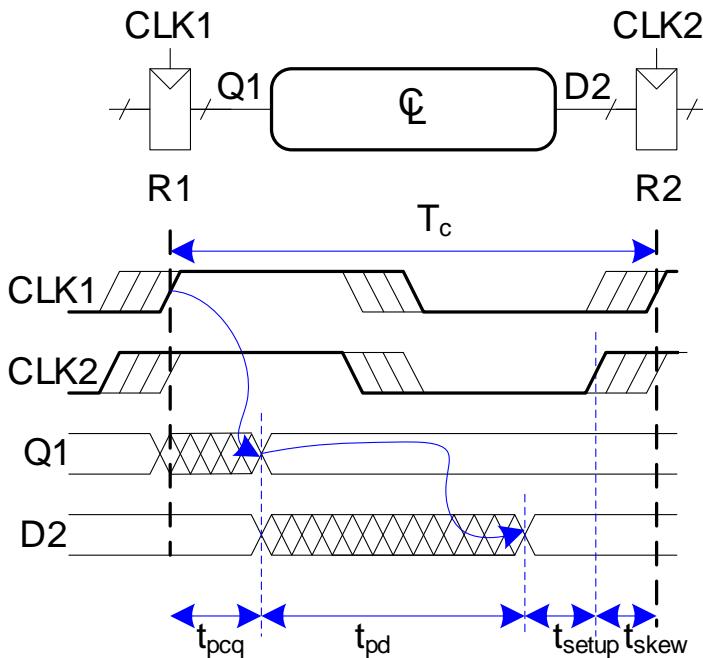
- Najhorší možný prípad: zmena na CLK2 nastane skôr než na CLK1



$$T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}$$
$$t_{pd} \leq$$

Časovanie & Čas nastavenia

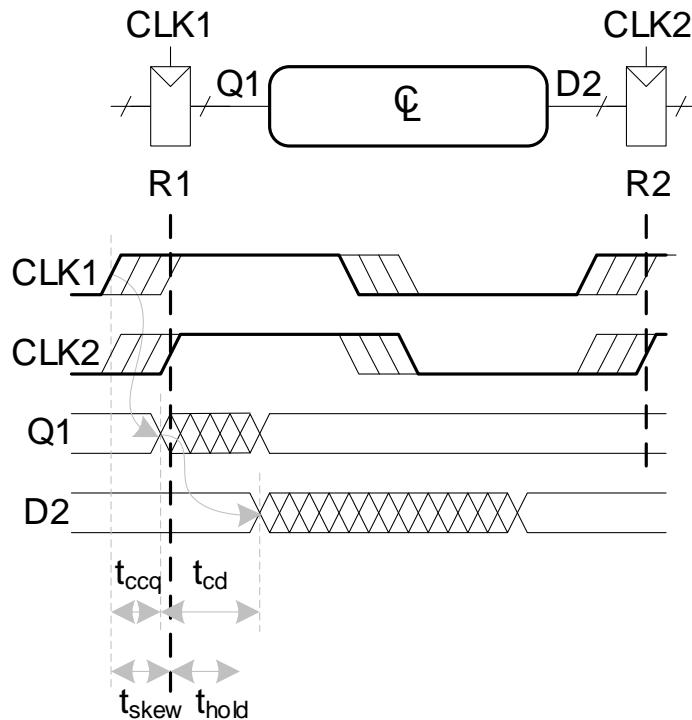
- Najhorší možný prípad: zmena na CLK2 nastane skôr než na CLK1



$$T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}$$
$$t_{pd} \leq T_c - (t_{pcq} + t_{setup} + t_{skew})$$

Časovanie & Čas presahu

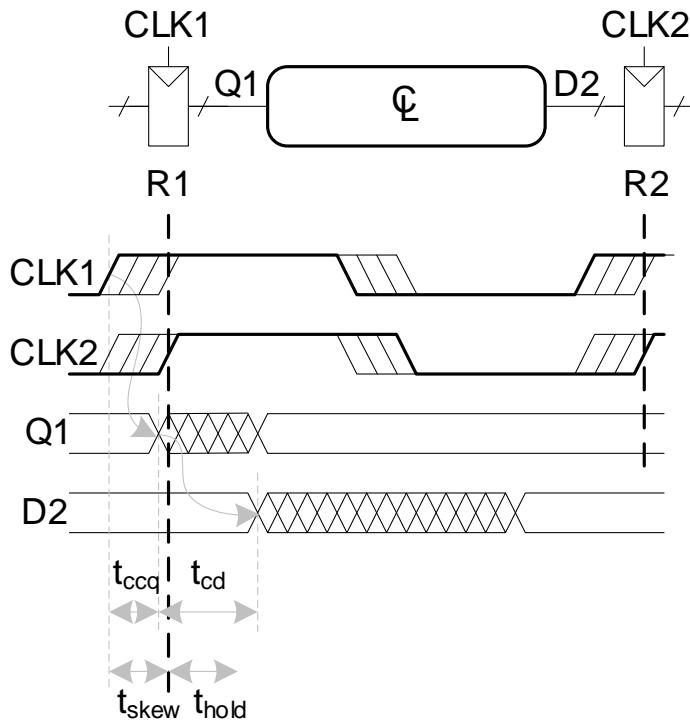
- Najhorší možný prípad: zmena na CLK2 nastane skôr než na CLK1



$$t_{ccq} + t_{cd} >$$

Časovanie & Čas presahu

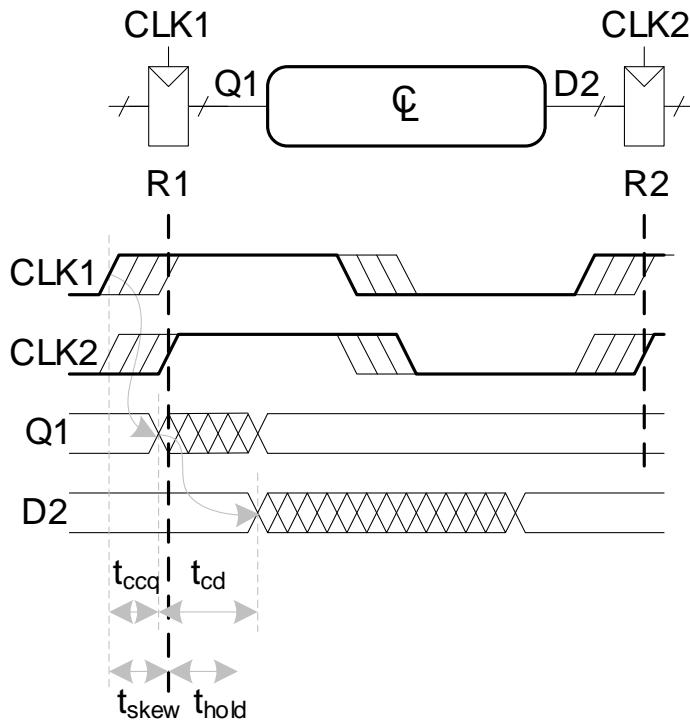
- Najhorší možný prípad: zmena na CLK2 nastane skôr než na CLK1



$$t_{ccq} + t_{cd} > t_{hold} + t_{skew}$$
$$t_{cd} >$$

Časovanie & Čas presahu

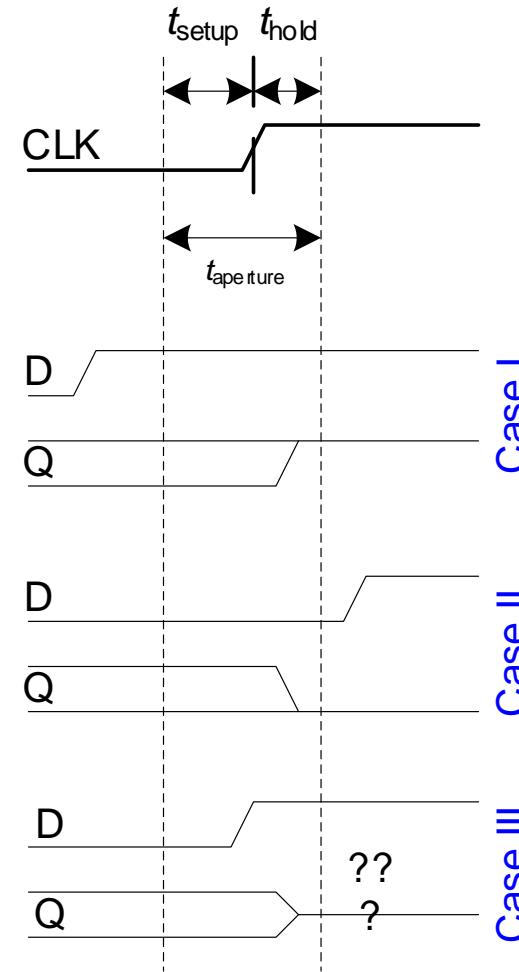
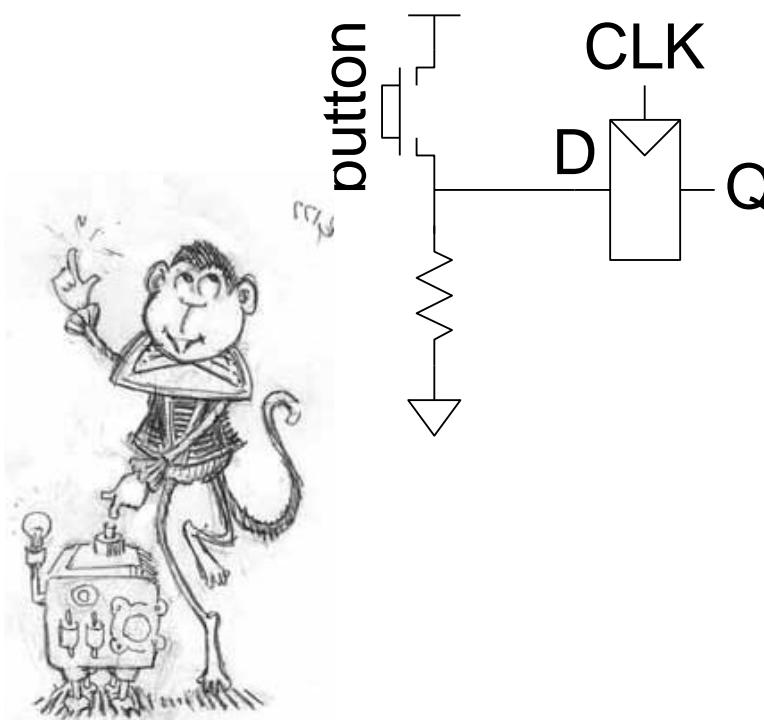
- Najhorší možný prípad: zmena na CLK2 nastane skôr než na CLK1



$$t_{ccq} + t_{cd} > t_{hold} + t_{skew}$$
$$t_{cd} > t_{hold} + t_{skew} - t_{ccq}$$

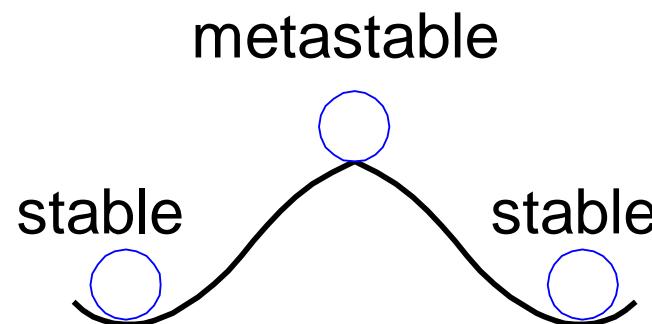
Porušenie dynamickej disciplíny

- Asynchrónne vstupy môžu mať za následok nepredvídateľné správanie sa obvodu



Metastabilný stav

- **Bistabilné zariadenia:** dva stabilné stavy a jeden metastabilný stav
- **Flip-flop:** dva stabilné stavy (1 a 0) a jeden metastabilný stav
- Ak flip-flop sa ocitne v metastabilnom stave, je ťažké určiť ako dlho v tomto stave ostane



Paralelizmus

- **Token:** dátový vstup
- **Doba odozvy:** čas prechodu tokenu log. systémom
- **Priepustnosť:** počet tokenov spracovaných za jednotku času

Paralelizmus zvyšuje priepustnosť

Paralelizmus

- Dva typy paralelizmu:
 - Priestorový paralelizmus
 - Zvýšenie počtu funkčných jednotiek
 - Temporálny (časový) paralelizmus
 - Princíp zreťazeného spracovania
 - Úloha sa rozdelí na dielčie kroky, ktoré sa vykonávajú súbežne
 - Napr. výrobná linka

Paralelizmus: príklad

- Pečenie koláča
 - Príprava cesta: 5 minút
 - Pečenie: 15 minút
 - Aká je odozva a priepustnosť „systému“?

Paralelizmus: príklad

- Ben peče koláč
 - Príprava cesta: 5 minút
 - Pečenie: 15 minút
 - Aká je odozva a priepustnosť „systému“?

Odozva = $5 + 15 = 20$ minút = **1/3 hodín**

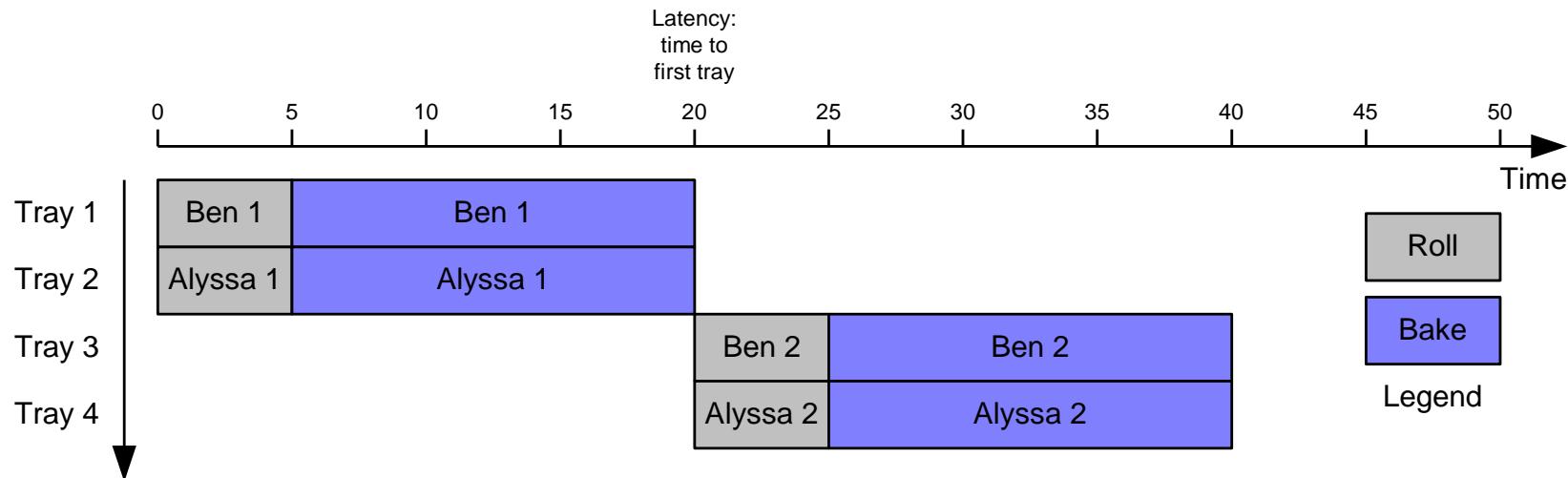
Priepustnosť = $(1 \text{ várka}) / (1/3 \text{ hodín}) = 3 \text{ várky/hodina}$

Paralelizmus: príklad

- Aká bude odozva a priepustnosť ak Ben paralelizuje svoj postup?
 - **Priestorový paralelizmus:** Ben poprosí Allysu, aby mohol použiť jej rúru
 - **Temporálny paralelizmus:**
 - Dve fázy: príprava cesta a pečenie
 - Tieto fázy sa prekrývajú
 - Kým sa peče prvá várka, pripravuje cesto pre druhú várku.

Priestorový paralelizmus

Spatial
Parallelism

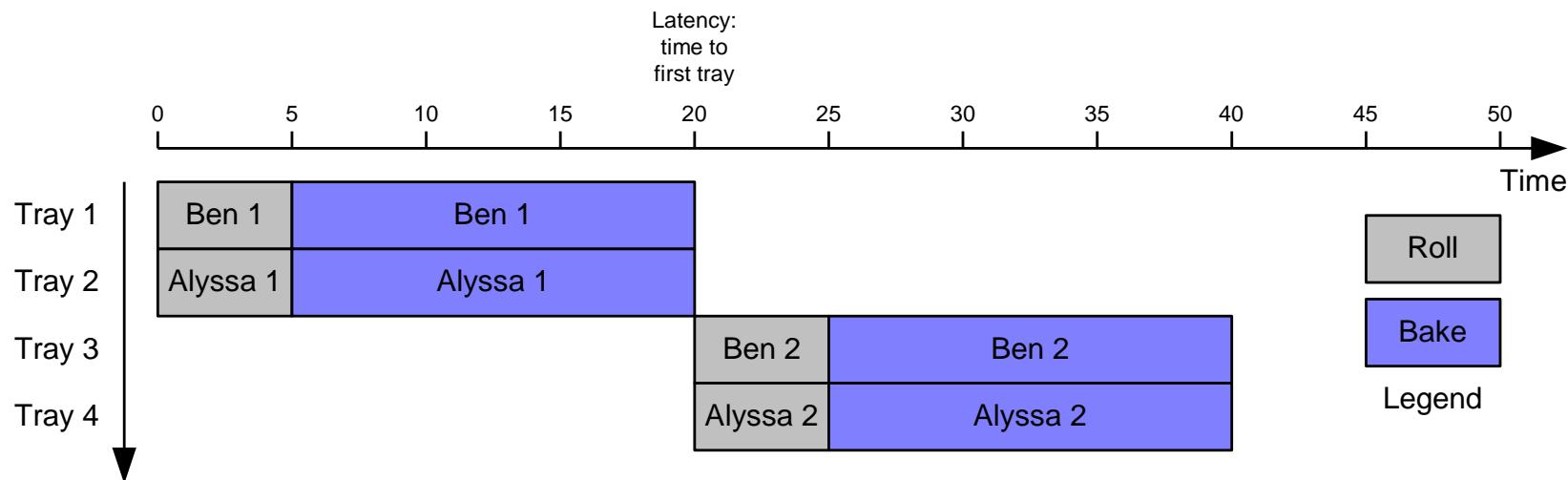


Odozva = ?

Priepustnosť = ?

Priestorový paralelizmus

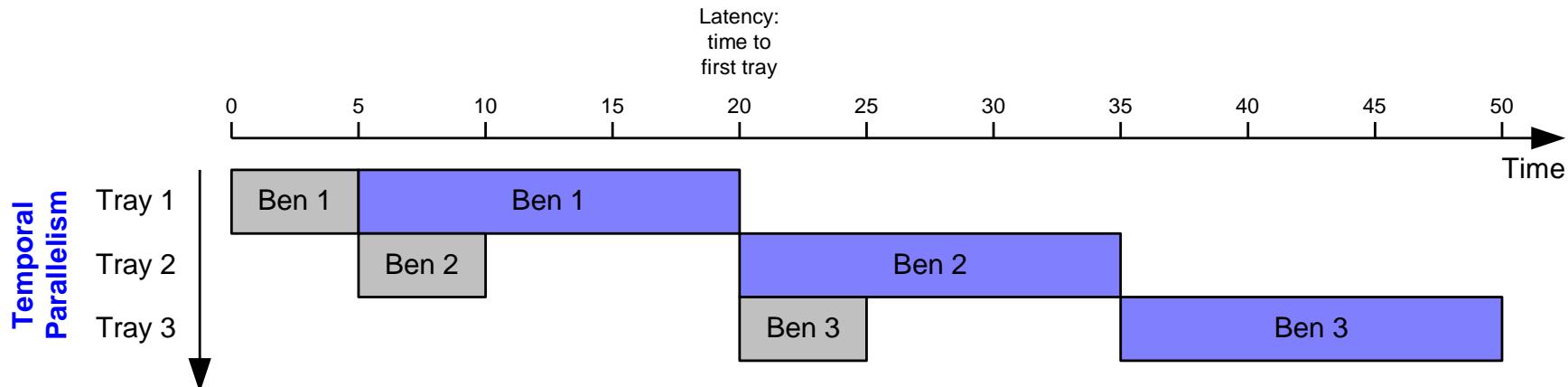
Spatial
Parallelism



$$\text{Odozva} = 5 + 15 = 20 \text{ minút} = \mathbf{1/3 \text{ hodín}}$$

$$\text{Priepustnosť} = (2 \text{ várky}) / (1/3 \text{ hodín}) = \mathbf{6 \text{ várok/hodina}}$$

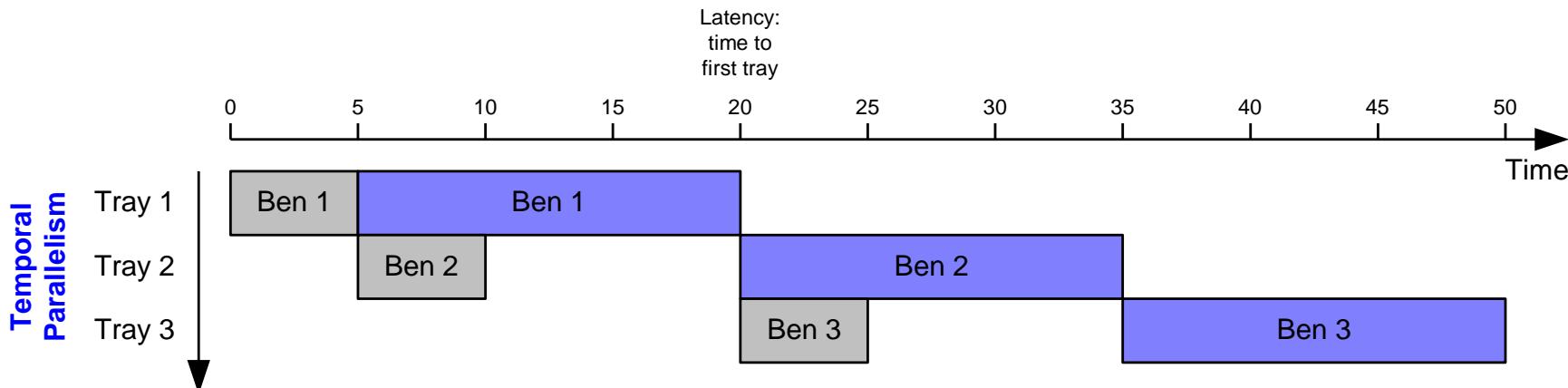
Temporálny parallelizmus



Odozva = ?

Priepustnosť = ?

Temporálny paralelizmus



Odozva = $5 + 15 = 20$ minút = **1/3 hodín**

Priepustnosť = $(1 \text{ várka}) / (1/4 \text{ hodín}) = 4 \text{ várky/hodina}$

Aplikovaním oboch typov paralelizmu **8 várkov/hodina**

Referencia

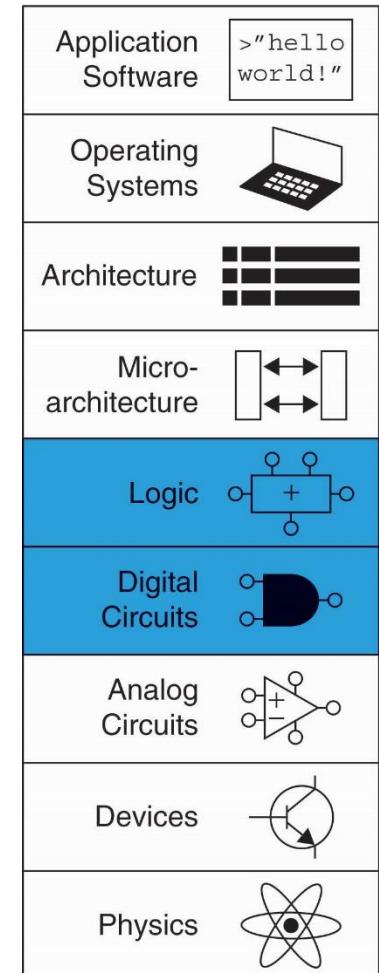
- David Money Harris and Sarah L. Harris,
Digital Design and Computer Architecture,
Chapter 3: Sequential Logic Design, Second
Edition © 2012 by Elsevier Inc.

Architektúry počítačových systémov

2. rok ZS

Osnova

- Návrhové prostriedky LS
- Kombinačná logika
- Behaviorálny & štrukturálny opis
- Sekvenčná logika
- Konečno-stavové automaty
- Parametrizované moduly
- Verifikácia, „testbench“ súbory



- Jazyk pre opis hardvéru (Hardware description language; HDL):
 - Navrhujeme log. systémy
 - (CAD) nástroj „zosyntetizuje“ LS
- Dva hlavné HDL:
 - **VHDL (VHDL 2008)**
 - Vývoj iniciovaný Ministerstvom obrany USA
 - IEEE štandard (1076) od roku 1987
 - Aktualizovaný v 2008 (IEEE STD 1076-2008)
 - **Verilog (SystemVerilog)**
 - Predstavený spoločnosťou Gateway Design Automation
 - IEEE štandard (1364) od roku 1995
 - Rozšírený v 2005 (IEEE STD 1800-2009)

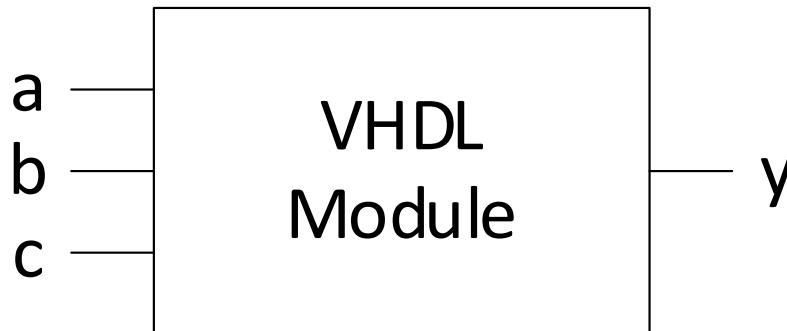
Od opisu k hradlám

- **Syntéza**
 - Prevod HDL kódu do *netlist* reprezentácie (tj, zoznam pomenovaných hradiel/komponentov a vodičov)
- **Simulácia**
 - Verifikácia činnosti obvodu
 - Na základe vstupu testujeme výstup

Dôležité:

HDL nie je programovací jazyk (v bežnom slova zmysle),
je to jazyk, ktorý slúži na opis HW

VHDL moduly



Spôsoby opisu LS:

- **Behaviorálny:** na čo slúži / z pohľadu chovania
- **Štrukturálny:** modulárny opis / z akých komponentov pozostáva
- **Toku dát** (dataflow): tok dát medzi komponentmi

Behaviorálny opis vo VHDL

VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity example is
    port( a, b, c: in STD_LOGIC;
          y: out STD_LOGIC);
end;

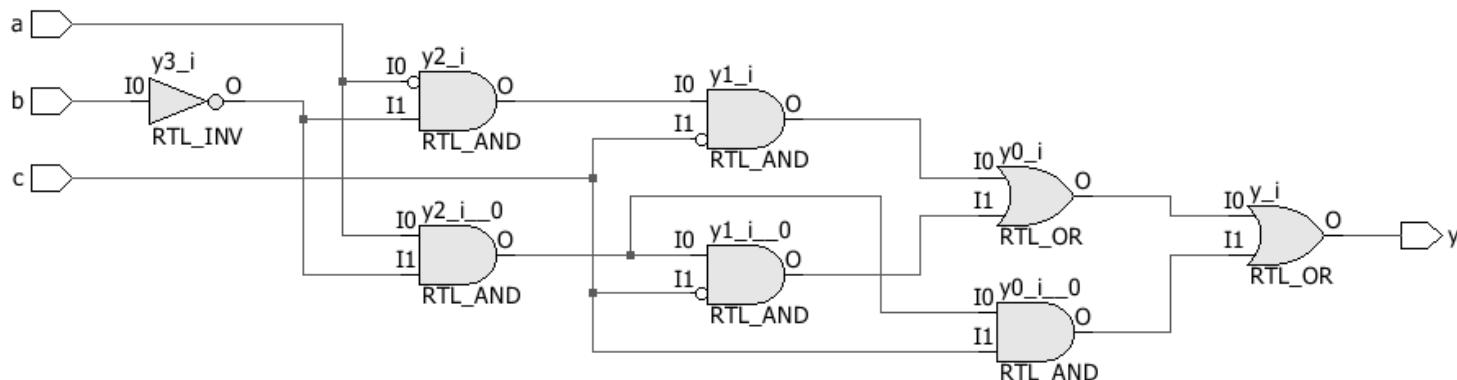
architecture behavioral of example is
begin
    y <=  (not a and not b and not c) or
          (a and not b and not c) or
          (a and not b and c);
end;
```

Syntéza obvodu

VHDL:

```
y <=  (not a and not b and not c) or  
       (a and not b and not c) or  
       (a and not b and c);
```

Vizualizácia syntézy:

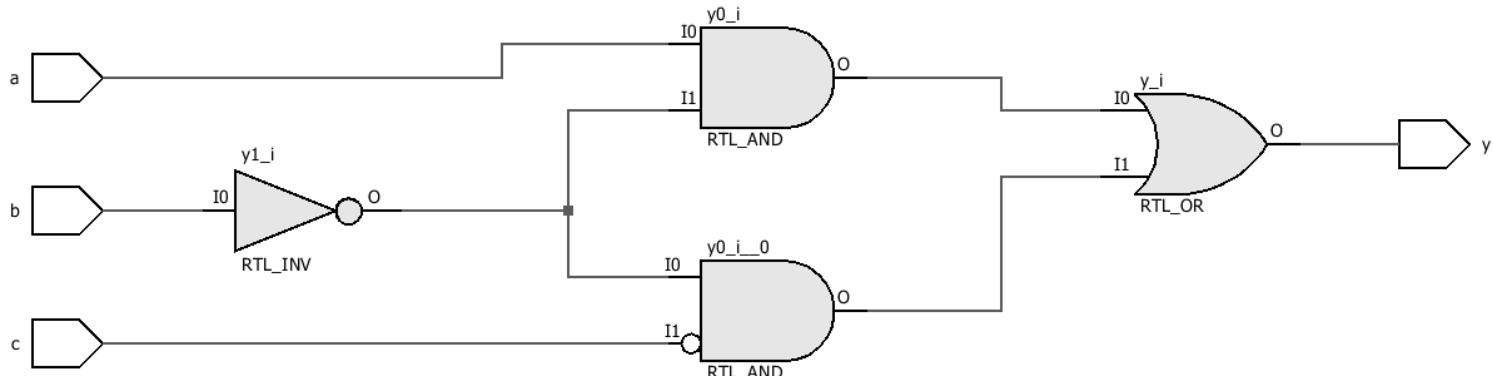


Syntéza obvodu

VHDL:

```
y <=  (not a and not b and not c) or  
       (a and not b and not c) or  
       (a and not b and c);
```

Vizualizácia syntézy:



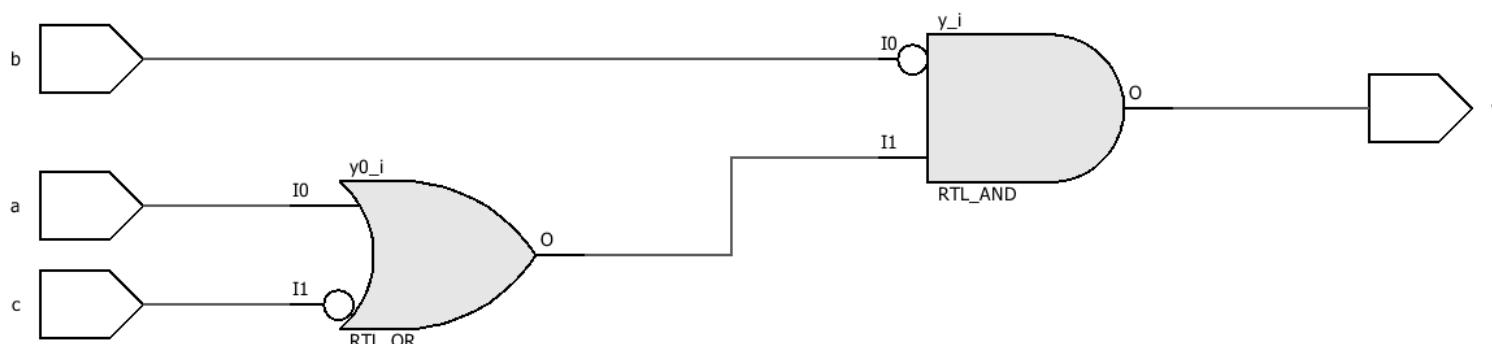
```
y <=  (a and not b) or (not b and not c);
```

Syntéza obvodu

VHDL:

```
y <=  (not a and not b and not c) or  
       (a and not b and not c) or  
       (a and not b and c);
```

Vizualizácia syntézy:

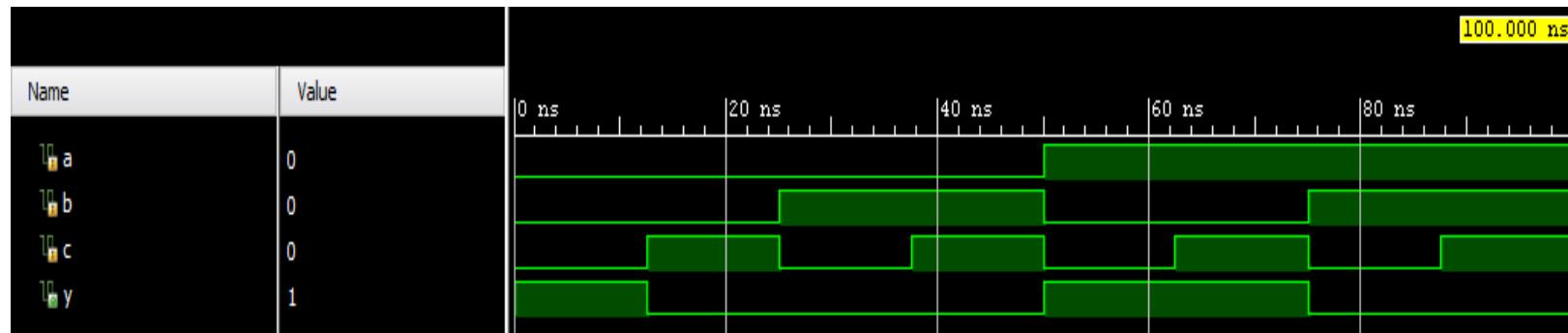


```
y <=  not b and (a or not c);
```

Simulácia činnosti obvodu

VHDL:

```
y <=  (not a and not b and not c) or  
       (a and not b and not c) or  
       (a and not b and c);
```



Syntax VHDL

- Je „case insensitive“
 - **Príklad:**
 - reset a Reset reprezentujú tie isté signály
 - Sú výnimky:
 - V dátovom type std_logic je len “X” a nie “x” (syntaktická chyba)
- Všeobecná syntax
 - pismeno{[_]pismeno_alebo_číslica}**
 - **Príklad:** 2mux je syntakticky nesprávny zápis
- Komentáre:
 - -- toto je obyčajna poznamka
 - /* viac-
riadkova poznamka */ ... Od VHDL 2008

Štrukturálny opis – Hierarchické modelovanie

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity nand3 is
    port(a, b, c : in STD_LOGIC;    y: out STD_LOGIC);
end;

architecture struct of nand3 is
component and3
    port(a, b, c : in STD_LOGIC; y: out STD_LOGIC);
end component;
component inv
    port(a, b, c : in STD_LOGIC; y: out STD_LOGIC);
end component;

signal n1: STD_LOGIC; -- internal signal

begin
    inst_and3: and3 port map(a, b, c, n1);
    inst_inv : inv port map(n1, y);
end;
```

Štrukturálny opis – Hierarchické modelovanie

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and3 is
    port(a, b, c : in STD_LOGIC;    y: out STD_LOGIC);
end;

architecture behav of and3 is
Begin
    y <= a and b and c;
end;
```

Štrukturálny opis – Hierarchické modelovanie

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

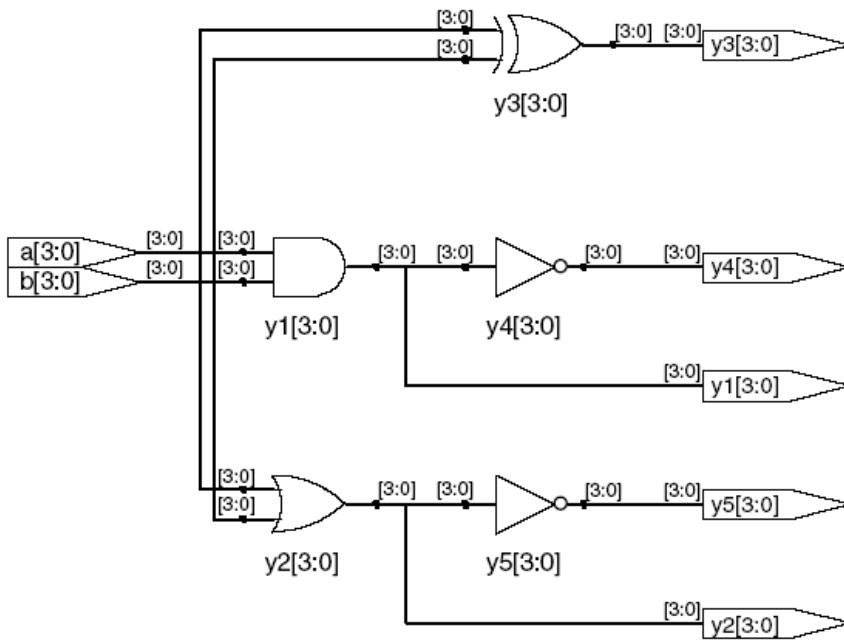
entity inv is
    port(a : in STD_LOGIC;      y: out STD_LOGIC);
end;

architecture behav of inv is
Begin
    y <= not a;
end;
```

Operátory

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

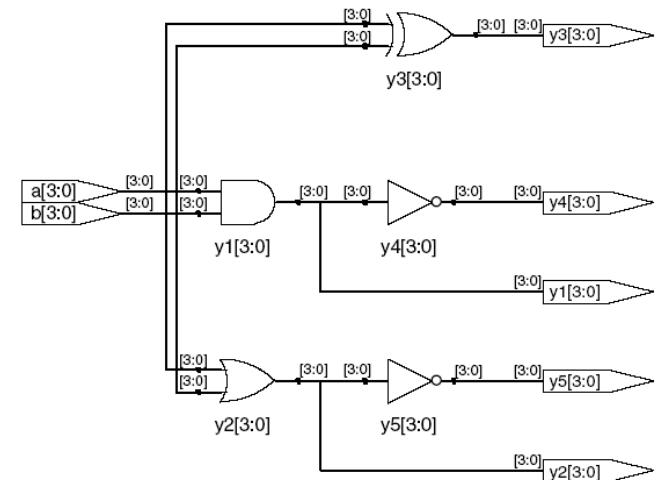
entity gates is
port(a, b: in STD_LOGIC_VECTOR(3 downto 0);
      y1, y2, y3, y4, y5: out STD_LOGIC_VECTOR(3 downto 0));
end;
```



Operátory

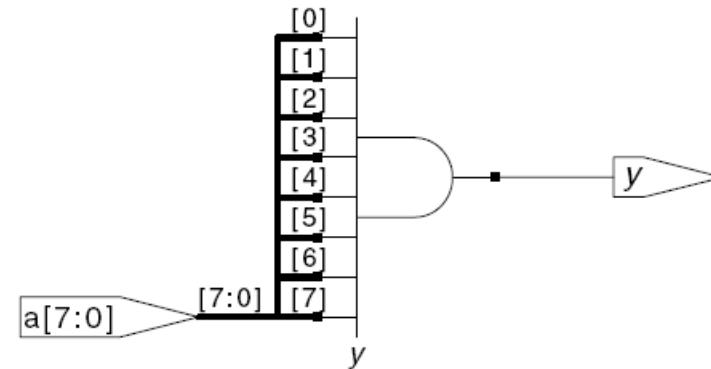
architecture behav of gates is
begin

```
/* Five different two-input logic
   gates acting on 4 bit busses */
y1 <= a and b;          -- AND
y2 <= a or b;           -- OR
y3 <= a xor b;          -- XOR
y4 <= not (a and b);   -- NAND
y5 <= not (a or b);    -- NOR
end;
```



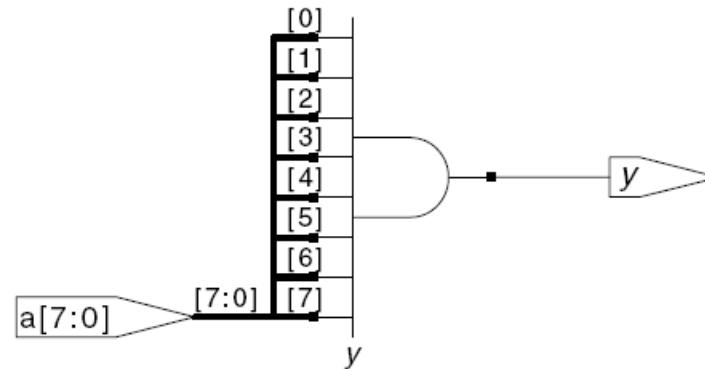
Operátor redukcie

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
  
entity and8 is  
    port(a: in STD_LOGIC_VECTOR(7 downto 0);  
          y: out STD_LOGIC);  
end;
```



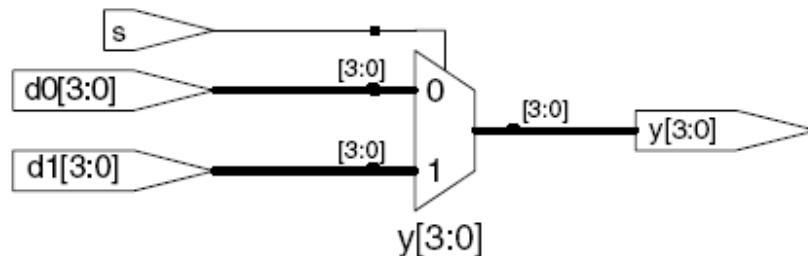
Operátor redukcie

```
architecture synth of and8 is
begin
    y <= and a; -- needs VHDL 2008
    -- and a is much easier to write than
    -- y <= a(7) and a(6) and a(5) and a(4) and
    -- a(3) and a(2) and a(1) and a(0);
end;
```



Podmienené priradenie

```
architecture synth of mux2 is
begin
    y <= d1 when s else d0;
end;
```

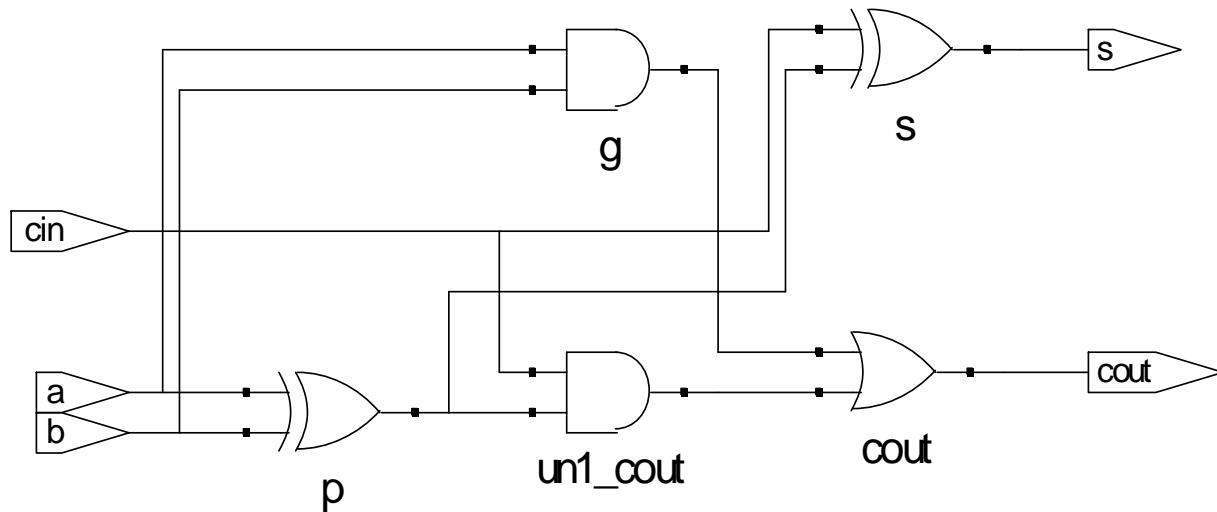


Note that prior to the 2008 revision of VHDL, one had to write `when s = '1'` rather than `when s`.

Vnútorné signály

```
architecture synth of fulladder is
    signal p, g: STD_LOGIC; -- internal signals
begin
    p <= a xor b;
    g <= a and b;
    s <= p xor cin;

    cout <= g or (p and cin);
end;
```



Priorita operátorov

Najvyššia

not	not
*, /, mod, rem	mult, div, mod
+, -	add, sub
rol, ror	rotate
srl, sll	shift logical
<, <=, >, >=	relative comparison
=, /=	equal, not equal
and, nand	AND, NAND
xor, xnor	XOR, XNOR
or, nor	OR, NOR

Najnižšia

Číselné typy

Formát: NB,,hodnota"

N = počet bitov, **B** = základ číselnej sústavy

NB nie je povinná položka (prednastavená je decimal)

Číslo	# Bity	Základ	Hodnota	Bin. reťazec
3B"101"	3	binary	5	101
B"11"	unsized	binary	3	11
8B"11"	8	binary	3	00000011
8B"1010_1011	8	binary	171	10101011
3D"6"	3	decimal	6	110
6O"42"	6	octal	34	100010
8X"AB"	8	hexadecimal	171	10101011
others=> '0'	n	binary	0	00...0
others=> '1'			$2^n - 1$	11...1

Operátor spájania (zret'azenia)

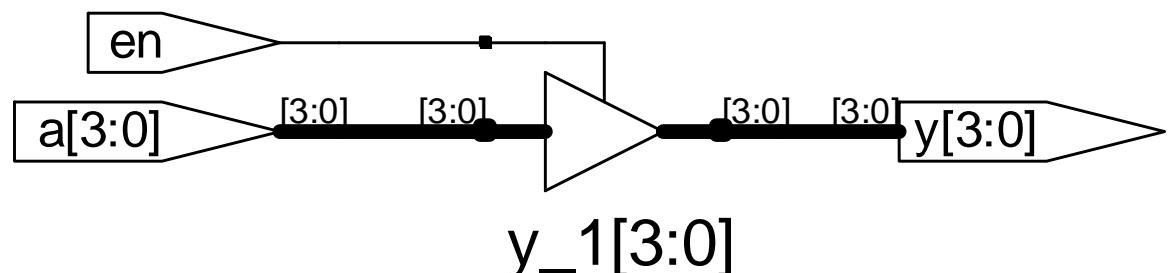
```
y <= (c(2 downto 1), d(0), d(0), d(0), c(0), 3B"101");  
  
/*  
   The () aggregate operator is used to concatenate  
   busses. y must be a 9-bit STD_LOGIC_VECTOR.  
*/  
  
z <= ("10", 4 => '1', 2 downto 1 =>'1', others =>'0')  
  
/*  
   If z is an 8-bit STD_LOGIC_VECTOR,  
   the above statement produces:  
       z = 1 0 0 1 0 1 1 0  
*/
```

Z: „Plávajúci“ výstup

VHDL:

```
entity tristate is
    port(a: in STD_LOGIC_VECTOR(3 downto 0);
         en: in STD_LOGIC;
         y: out STD_LOGIC_VECTOR(3 downto 0));
end;
```

```
architecture synth of tristate is
begin
    y <= a when en else "ZZZZ";
end;
```

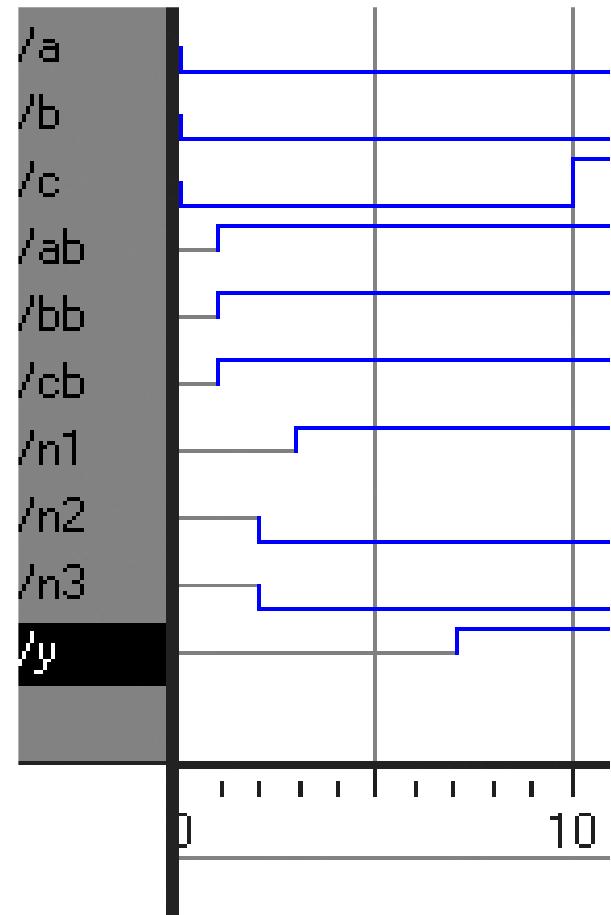


Oneskorenie

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

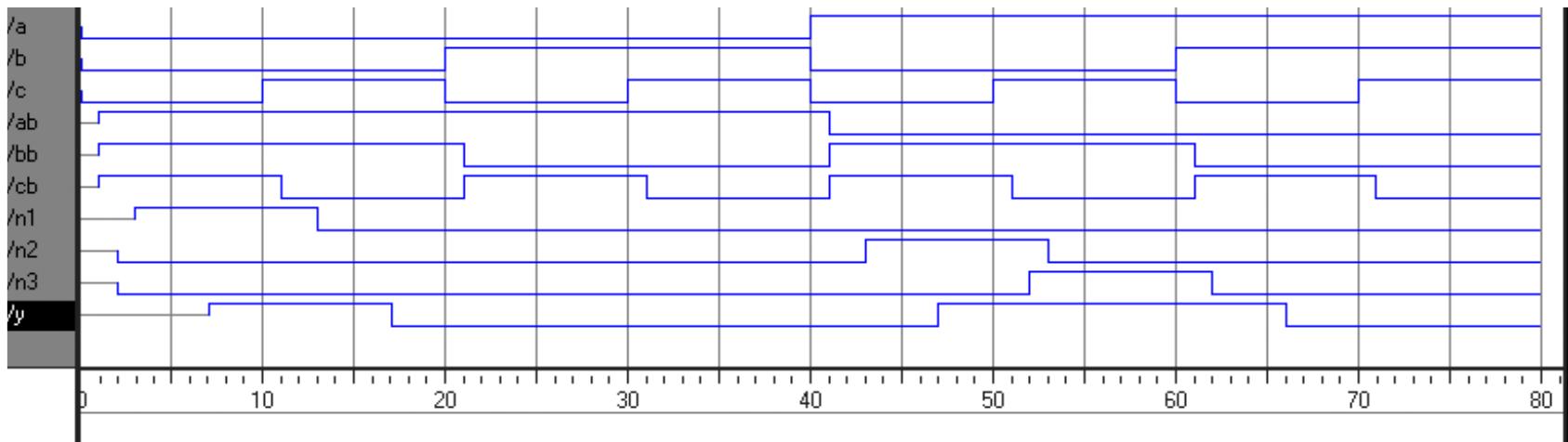
entity example is
    port(a, b, c: in STD_LOGIC;
          y: out STD_LOGIC);
end;

architecture synth of example is
signal
    ab, bb, cb, n1, n2, n3: STD_LOGIC;
begin
    ab <= not a after 1 ns;
    bb <= not b after 1 ns;
    cb <= not c after 1 ns;
    n1 <= ab and bb and cb after 2 ns;
    n2 <= a and bb and cb after 2 ns;
    n3 <= a and bb and c after 2 ns;
    y <= n1 or n2 or n3 after 4 ns;
end;
```



Oneskorenie

```
architecture synth of example is
signal ab, bb, cb, n1, n2, n3: STD_LOGIC;
begin
    ab <= not a after 1 ns;    bb <= not b after 1 ns;
    cb <= not c after 1 ns;    n1 <= ab and bb and cb after 2 ns;
    n2 <= a and bb and cb after 2 ns;
    n3 <= a and bb and c after 2 ns;
    y <= n1 or n2 or n3 after 4 ns;
end;
```



Sekvenčná logika

- VHDL používa návrhové vzory na modelovanie správania sa preklápacích obvodov a stavových automatov
 - Príkaz **process**
 - Sekvenčné príkazy
 - if,
 - case
 - loop

Príkaz process

Základná stavba príkazu:

```
process (sensitivity list)
begin
    statement;
end process;
```

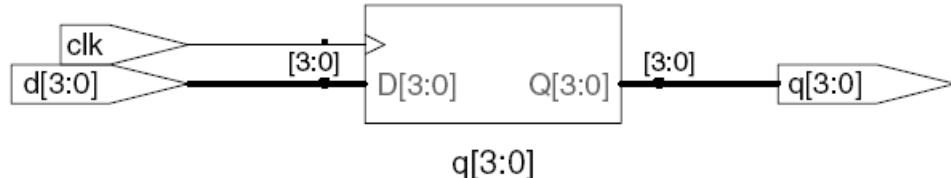
Príkazy vo vnútri bloku **process** sú vykonané len vtedy, ak je zaznamenaná zmena aspoň jedného signálu z *citlivostného zoznamu* príkazu process.

Modelovanie PO typu DFF

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity flop is
    port(clk: in STD_LOGIC;
          d: in STD_LOGIC_VECTOR(3 downto 0);
          q: out STD_LOGIC_VECTOR(3 downto 0));
end;

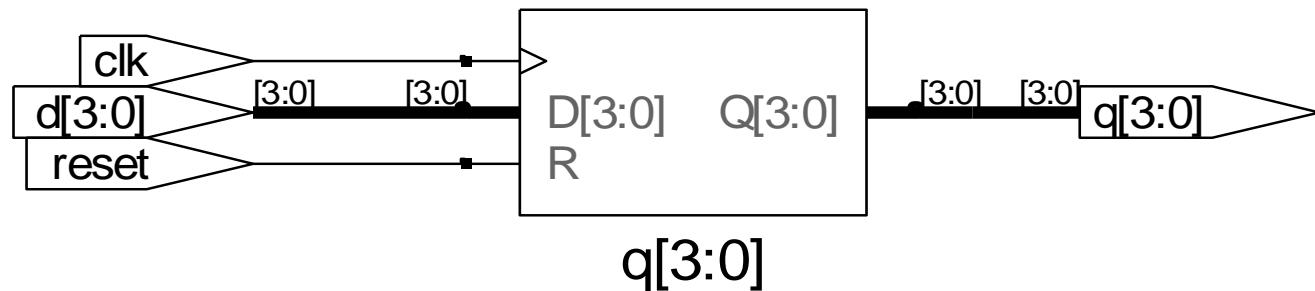
architecture synth of flop is
begin
    process(clk) begin
        if rising_edge(clk) then
            q <= d;
        end if;
    end process;
end;
```



Modelovanie DFF s možnosťou resetu

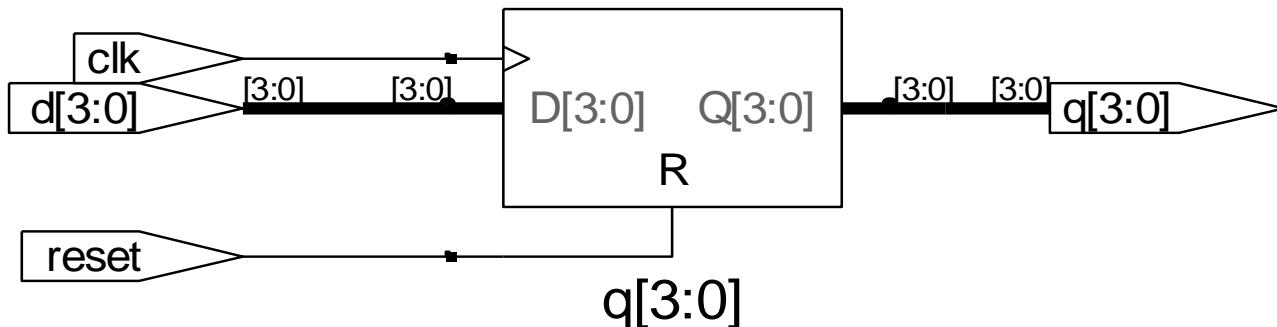
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity flopr is
    port(clk, reset: in STD_LOGIC;
          d: in STD_LOGIC_VECTOR(3 downto 0);
          q: out STD_LOGIC_VECTOR(3 downto 0));
end;
```



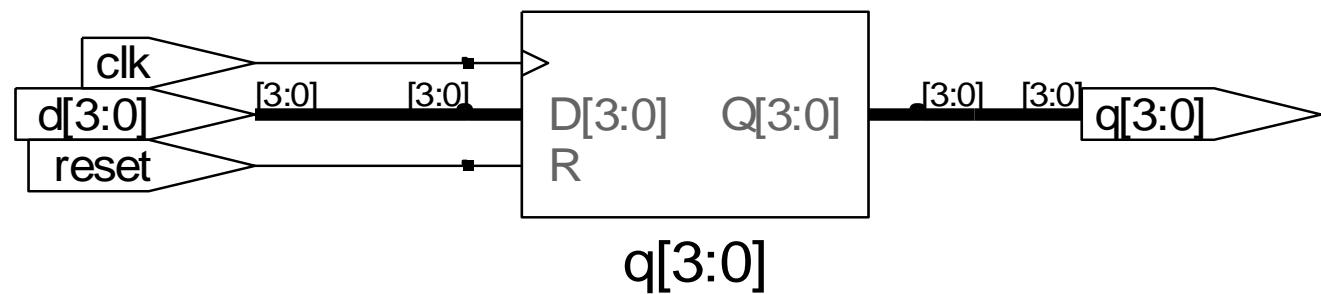
Modelovanie DFF s možnosťou resetu

```
architecture asynchronous of flopr is
begin
    process(clk, reset) begin
        if reset then
            q <= "0000";
        elsif rising_edge(clk) then
            q <= d;
        end if;
    end process;
end;
```



Modelovanie DFF s možnosťou resetu

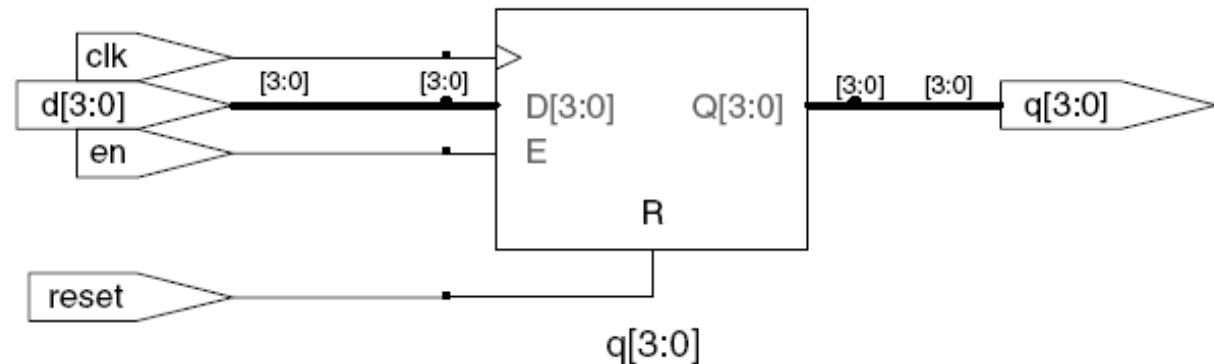
```
architecture synchronous of flopr is
begin
    process(clk) begin
        if rising_edge(clk) then
            if reset then q <= "0000";
            else q <= d;
        end if;
    end if;
    end process;
end;
```



Modelovanie DFF s možnosťou resetu a blokovania

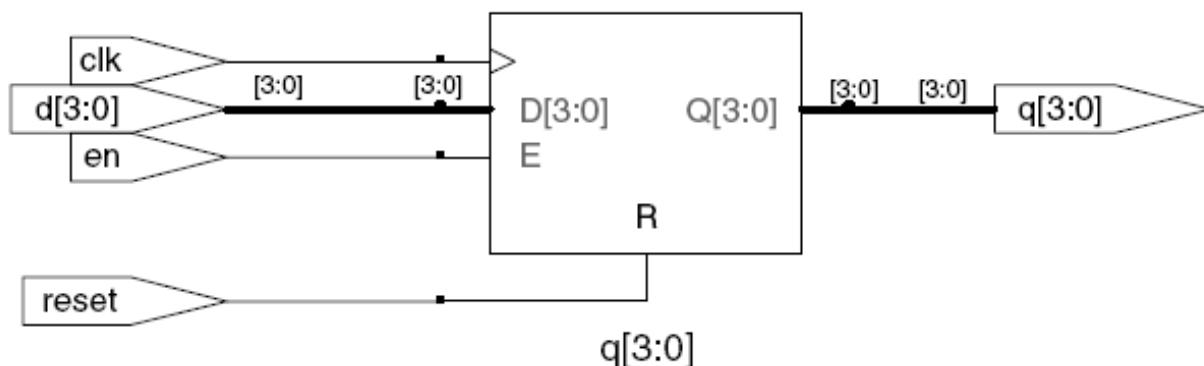
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity flopenr is
    port(clk, reset, en: in STD_LOGIC;
          d: in STD_LOGIC_VECTOR(3 downto 0);
          q: out STD_LOGIC_VECTOR(3 downto 0));
end;
```



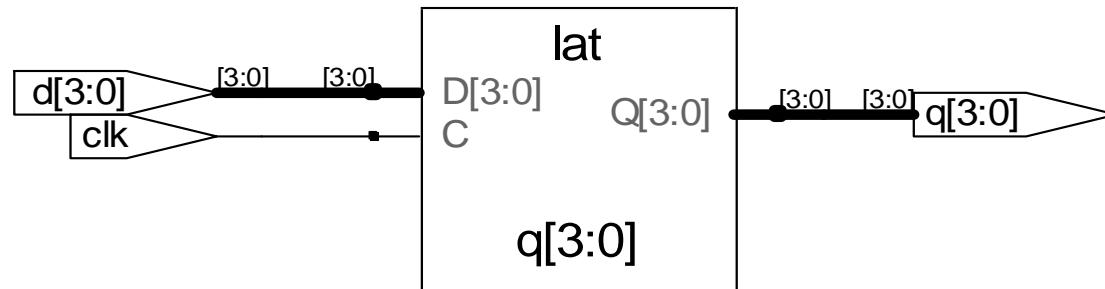
Modelovanie DFF s možnosťou resetu a blokovania

```
architecture asynchronous of flopenr is
-- asynchronous reset
begin
    process(clk, reset) begin
        if reset then
            q <= "0000";
        elsif rising_edge(clk) then
            if en then
                q <= d;
            end if;
            end if;
        end process;
    end;
```



PO typu „latch“

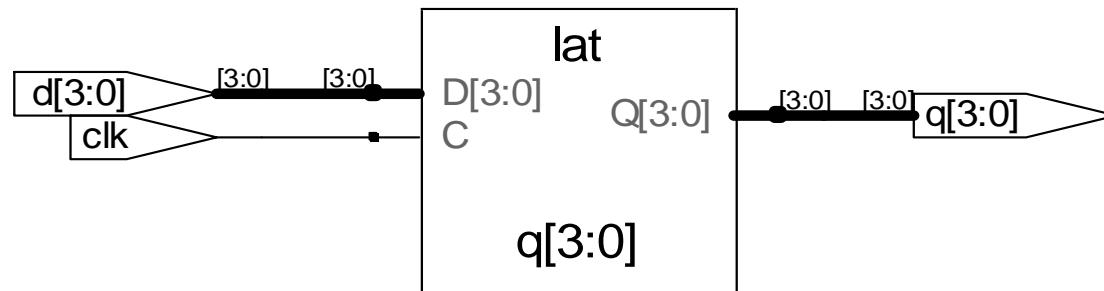
```
library IEEE; use IEEE.STD_LOGIC_1164.all;  
entity latch is  
    port(clk: in STD_LOGIC;  
          d: in STD_LOGIC_VECTOR(3 downto 0);  
          q: out STD_LOGIC_VECTOR(3 downto 0));  
end;
```



Upozornenie: Použitie úrovňou riadených PO v súvislosti s FPGA môže viest' k porušeniu požadovaných časových charakteristík LO.

PO typu „latch“

```
architecture synth of latch is
begin
    process(clk, d)
    begin
        if clk = '1' then
            q <= d;
        end if;
    end process;
end;
```



Súčasťou citlivostného zoznamu sú signály clk a d.

Príkazy v bloku *process* sú vyhodnotené zakaždým, čo je zaznamenaná zmena na clk alebo na d. Ak clk je HIGH, potom $q = d$.

Sekvenčné príkazy

- Sekvenčný príkaz je súčasťou príkazu/bloku process:
 - if / else
 - case, case?
 - loop

KLO pomocou process

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
  
entity gates is  
    port(a,b : in STD_LOGIC_VECTOR(3 downto 0);  
          y1,y2,y3,y4,y5: out STD_LOGIC_VECTOR(3 downto 0));  
end;
```

KLO pomocou process

```
-- combinational logic using a process statement
architecture behav of gates is
begin
  process(all)
  begin
    y1 <= a and b;          -- AND
    y2 <= a or b;           -- OR
    y3 <= a xor b;          -- XOR
    y4 <= not (a and b);   -- NAND
    y5 <= not (a or b);    -- NOR
  end process;
end;
```

Na modelovanie kombinačných logických obvodov (KLO) je vhodnejšie použiť konkurenčné (paralelné) príkazy.

KLO pomocou case

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity seven_seg_decoder is
    port(data: in STD_LOGIC_VECTOR(3 downto 0);
         segments: out STD_LOGIC_VECTOR(6 downto 0));
end;
```

KLO pomocou case

```
architecture synth of seven_seg_decoder is
begin
    process(all) begin
        case data is
            -- abcdefg
            when X"0" => segments <= "1111110";
            when X"1" => segments <= "0110000";
            when X"2" => segments <= "1101101";
            when X"3" => segments <= "1111001";
            when X"4" => segments <= "0110011";
            when X"5" => segments <= "1011011";
            when X"6" => segments <= "1011111";
            when X"7" => segments <= "1110000";
            when X"8" => segments <= "1111111";
            when X"9" => segments <= "1110011";
            when others => segments <= "0000000"; -- required
        end case;
    end process;
end;
```

KLO pomocou case

- Príkaz case viedie na syntézu KLO len vtedy ak sme pokryli všetky možné vstupné kombinácie
 - pozor na 9 hodnotový std_logic
 - pozor na citlivostný zoznam
- Použi **when others**

KLO pomocou case?

```
-- PRIORITY CIRCUIT USING DON'T CARES

library IEEE;
use IEEE.STD_LOGIC_1164.all;

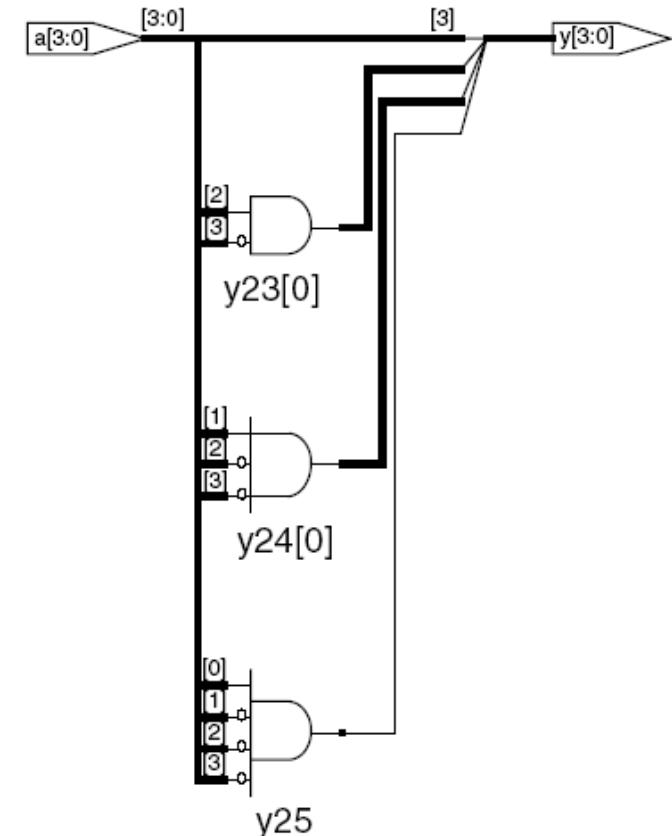
entity priority_casez is
    port(a: in STD_LOGIC_VECTOR(3 downto 0);
         y: out STD_LOGIC_VECTOR(3 downto 0));
end;
```

Od VHDL 2008.

Príkaz **case?** na rozdiel od „obyčajného“ **case** dokáže pracovať aj s „**don't care**“ hodnotami.

KLO pomocou case?

```
architecture dontcare of priority_casez is
begin
    process(all) begin
        case? a is
            when "1---" =>y<= "1000";
            when "01--" =>y<= "0100";
            when "001-" =>y<= "0010";
            when "0001"=>y<= "0001";
            when others=>y<= "0000";
        end case?;
    end process;
end;
```



Blokujúce vs neblokujúce priradenie

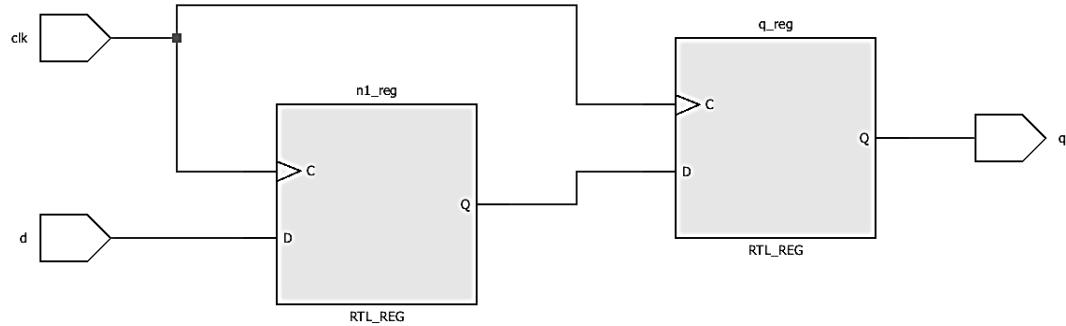
Použí process(clk) a (ne)blokujúce priradenie na modelovanie synchrónneho SLO.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity sync is
    port(clk: in STD_LOGIC;
          d: in STD_LOGIC;
          q: out STD_LOGIC);
end;
architecture synth of sync is
begin
...
end
```

Blokujúce vs neblokujúce priradenie

Použi process(clk) a (ne)blokujúce priradenie na modelovanie synchrónneho SLO.

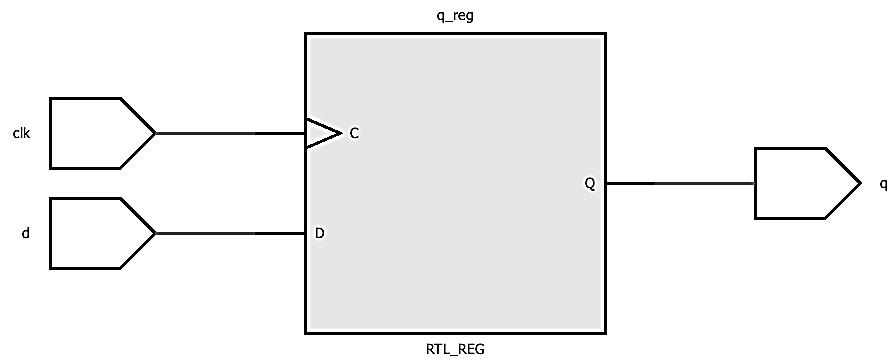
```
-- Good synchronizer using nonblocking assignments
process(clk) begin
    if rising_edge(clk) then
        n1 <= d; -- nonblocking
        q   <= n1; -- nonblocking
    end if;
end process;
```



Blokujúce vs neblokujúce priradenie

Použi process(clk) a (ne)blokujúce priradenie na modelovanie synchrónneho SLO.

```
-- Bad synchronizer using blocking assignments
process(clk)
    variable n1: STD_LOGIC;
begin
    if rising_edge(clk) then
        n1 := d; -- blocking
        q <= n1;
    end if;
end process;
```



Odporúčania

- **SLO:** Použi príkaz **process (clk)** a neblokujúce priradenie (**<=**) pre modelovanie SLO

```
process (clk) begin  
    if rising_edge(clk) then  
        q <= d; // nonblocking  
    end if;  
end process;
```

- **KLO:** Použi konkurenčné (paralelné) príkazy na tvorbu KLO

```
y <= d0 when s = '0' else d1;
```

Odporúčania

- Pre zložitejšie KLO môžeš použiť príkaz process (all)^{*}, resp. zarad' do citlivostného zoznamu každý signál, ktorý je na pravej strane priradenia, resp. je súčasťou predikátu. Použi blokujúce priradenie pre pomocné premenné.

```
process(all)
    variable p, g: STD_LOGIC;
begin
    p := a xor b; -- blocking
    g := a and b; -- blocking
    s <= p xor cin;
    cout <= g or (p and cin);
end process;
```

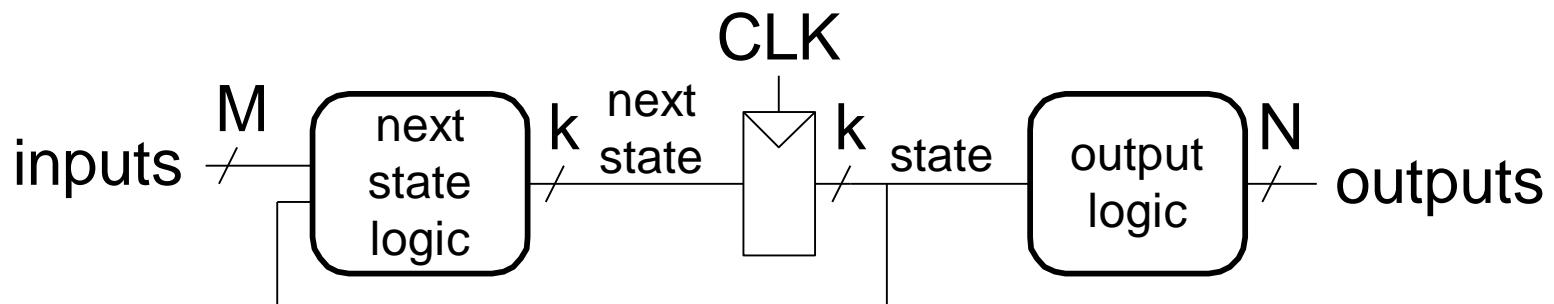
* Od VHDL 2008

Odporúčania

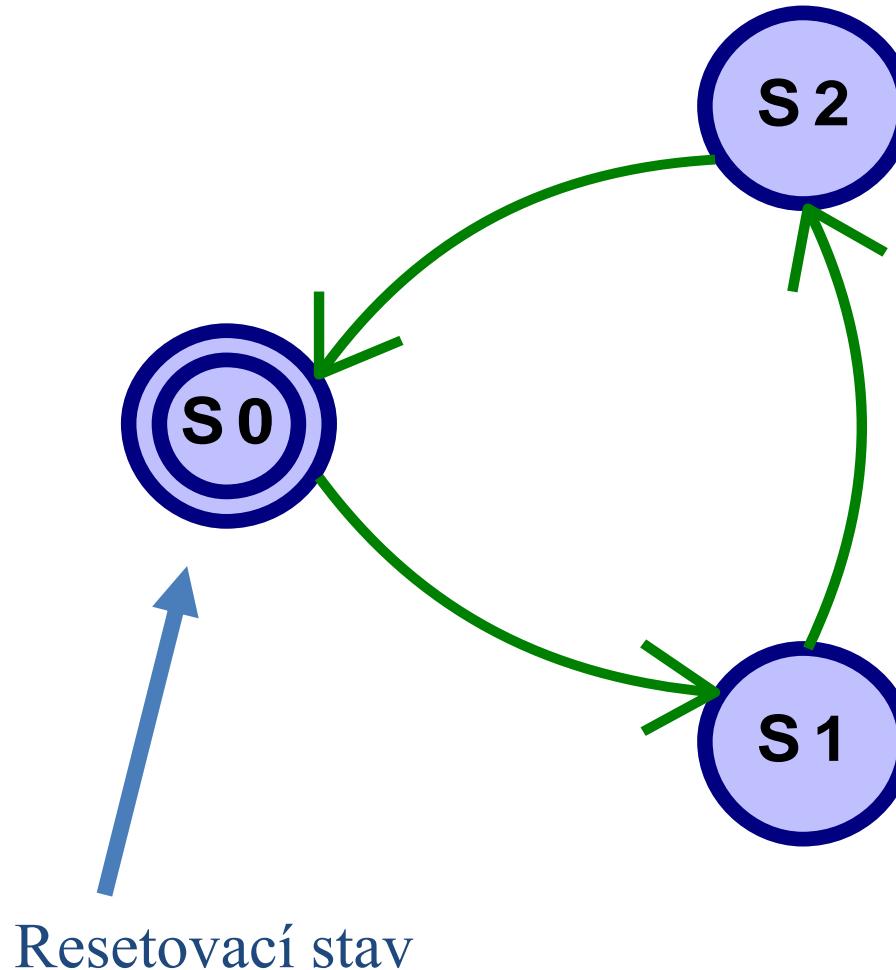
- Aplikuj pravidlo pri nastavení hodnoty signálu:
 - jeden signál jeden process, resp.
 - jeden signál jeden priradovací príkaz

Konečno-stavové automaty(FSMs)

- **Tri bloky:**
 - Vstupný logický obvod
 - Stavový register
 - Výstupný logický obvod



Príklad: Deliteľnosť tromi



VHDL kód

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity divideby3FSM is
    port(clk, reset: in STD_LOGIC;
          y: out STD_LOGIC);
end;
architecture synth of divideby3FSM is
    type statetype is (S0, S1, S2);
    signal state, nextstate: statetype;
begin
    -- state register
    process(clk, reset) begin
        if reset then
            state <= S0;
        elsif rising_edge(clk) then
            state <= nextstate;
        end if;
    end process;
    -- next state logic
    nextstate <= S1 when state = S0 else S2 when state = S1 else S0;
    -- output logic
    y <= '1' when state = S0 else '0';
end;
```

Parametrizované moduly

- HDLs umožňuje parametrizovať moduly
 - Klúčové slovo **generic**
 - generická konštantá vo vnútri bloku entity
 - nastavenie funkcie modulu
 - nastavenie bitovej šírky
 - Klúčové slovo **generate**
 - Genreujúci príkaz
 - úsporný opis rovníc
 - opakujúce sa log. štruktúry
- { opis pomocou rekurzie alebo cyklov

Parametrizované moduly

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is
    generic(width: integer := 8);
    port(d0,
          d1: in STD_LOGIC_VECTOR(width-1 downto 0);
          s: in STD_LOGIC;
          y: out STD_LOGIC_VECTOR(width-1 downto 0));
end;
architecture synth of mux2 is
begin
    y <= d1 when s else d0;
end;
```

Parametrizované moduly

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
  
entity mux4_8 is  
    port(d0, d1,  
          d2, d3: in STD_LOGIC_VECTOR(7 downto 0);  
          s: in STD_LOGIC_VECTOR(1 downto 0);  
          y: out STD_LOGIC_VECTOR(7 downto 0));  
end;
```

Parametrizované moduly

```
architecture struct of mux4_8 is
component mux2
    generic(width: integer := 8);
    port(d0,
          d1: in STD_LOGIC_VECTOR(width-1 downto 0);
          s: in STD_LOGIC;
          y: out STD_LOGIC_VECTOR(width-1 downto 0));
end component;
signal low, hi: STD_LOGIC_VECTOR(7 downto 0);
begin
    lowmux: mux2 port map(d0, d1, s(0), low);
    himux: mux2 port map(d2, d3, s(0), hi);
    outmux: mux2 port map(low, hi, s(1), y);
end;
```

Simulácia / testbench súbory

- Na simuláciu modulov použijeme testbench súbor
 - Zvyčajne blok *entita* je prázdna
 - Blok *architecture* obsahuje inštanciu *unit under test* (uut) testovaného modulu
- Testbench nie je určený na syntézu
- Typy:
 - Jednoduchá funkčná
 - S autokontrolou
 - S autokontrolou na báze testovacieho súboru s testovacími vektormi

Príklad

- Navrhnite modul sillyfunction v jazyku VHDL na realizáciu funkcie:

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

Testbench súbor

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
  
entity testbench1 is -- no inputs or outputs  
end;  
  
architecture sim of testbench1 is  
    component sillyfunction  
        port(a, b, c: in STD_LOGIC;  
              y: out STD_LOGIC);  
    end component;  
  
    signal a, b, c, y: STD_LOGIC;  
  
-- continued on the next slide
```

Jednoduchý Testbench

```
begin
  -- instantiate unit under test
  uut: sillyfunction port map(a, b, c, y);
  -- apply inputs one at a time
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1'; wait for 10 ns;
    b <= '1'; c <= '0'; wait for 10 ns;
    c <= '1'; wait for 10 ns;
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1'; wait for 10 ns;
    b <= '1'; c <= '0'; wait for 10 ns;
    c <= '1'; wait for 10 ns;
    wait; -- wait forever
  end process;
end;
```

Testbench s autokontrolou

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity testbench2 is -- no inputs or outputs
end;

architecture sim of testbench2 is
component sillyfunction
    port(a, b, c: in STD_LOGIC;
         y: out STD_LOGIC);
    end component;

    signal a, b, c, y: STD_LOGIC;

-- continued on the next slide
```

Testbench s autokontrolou

```
begin
  -- instantiate unit under test
  dut: sillyfunction port map(a, b, c, y);
  -- apply inputs one at a time
  -- checking results
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
    assert y = '1' report "000 failed.";
    c <= '1'; wait for 10 ns;
    assert y = '0' report "001 failed.";
    b <= '1'; c <= '0'; wait for 10 ns;
    assert y = '0' report "010 failed.";
    c <= '1'; wait for 10 ns;
    assert y = '0' report "011 failed.";

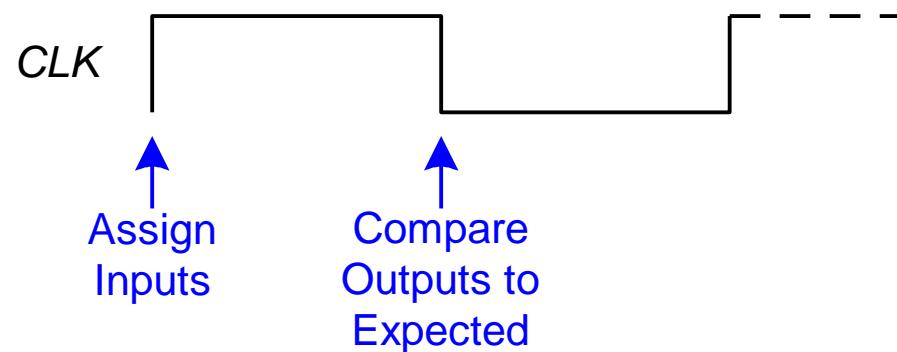
  -- continued on the next slide
```

Testbench s autokontrolou

```
a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;  
    assert y = '1' report "100 failed."  
c <= '1'; wait for 10 ns;  
    assert y = '1' report "101 failed."  
b <= '1'; c <= '0'; wait for 10 ns;  
    assert y = '0' report "110 failed."  
c <= '1'; wait for 10 ns;  
    assert y = '0' report "111 failed."  
wait; -- wait forever  
end process;  
end;
```

- Súbor s vektormi
 - vstupy a očakávané výstupy
- Aplikuj postup:
 1. Vytvor inštanciu UUT testovaného modulu
 2. Vytvor hodinový signál
 3. Načítaj testovacie vektory z testovacieho súboru
 4. Inicializuj vstupy s test. vektorom
 5. Prečítaj výstupy z UUT
 6. Porovnaj výstupy s očakávanými výstupmi
 - a) Signalizuj prípadnú chybu

- Hodinový signál:
 - Inicializuj vstupy (počas nábehovej hrany)
 - Porovnaj výstup s očakávanými výstupmi (počas dobehovej hrany)



- Hodinový signál je nevyhnutný v prípade synchrónnych sekvenčných obvodov

Súbor s testovacími vektormi

- example.tv
- Obsahuje vektor v tvare abc_yexpected

```
000_1  
001_0  
010_0  
011_0  
100_1  
101_1  
110_0  
111_0
```

Testbench s a-kontrolou na báze testovacieho súboru

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_TEXTIO.ALL;
use STD.TEXTIO.all;

entity testbench3 is -- no inputs or outputs
end;

architecture sim of testbench3 is
    component sillyfunction
        port(a, b, c: in STD_LOGIC;
             y: out STD_LOGIC);
    end component;
    signal a, b, c, y: STD_LOGIC;
    signal y_expected: STD_LOGIC;
    signal clk, reset: STD_LOGIC;

-- continued on the next slide
```

Testbench s a-kontrolou na báze testovacieho súboru

```
begin
  -- instantiate unit under test
  dut: sillyfunction port map(a, b, c, y);
  -- generate clock
  process begin
    clk <= '1'; wait for 5 ns;
    clk <= '0'; wait for 5 ns;
  end process;

  -- at start of test, pulse reset
  process begin
    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
  end process;

  -- continued on the next slide
```

Testbench s a-kontrolou na báze testovacieho súboru

```
-- run tests
process is
    file tv: text;
    variable L: line;
    variable vector_in: std_logic_vector(2 downto 0);
    variable dummy: character;
    variable vector_out: std_logic;
    variable vectornum: integer := 0;
    variable errors: integer := 0;
begin
    FILE_OPEN(tv, "example.tv", READ_MODE);

-- continued on the next slide
```

Testbench s a-kontrolou na báze testovacieho súboru

```
while not endfile(tv) loop
    -- change vectors on rising edge */
    wait until rising_edge(clk);
    -- read the next line of testvec and split into pieces
    readline(tv, L);
    read(L, vector_in);
    read(L, dummy); -- skip over underscore
    read(L, vector_out);
    (a, b, c) <= vector_in(2 downto 0) after 1 ns;
    y_expected <= vector_out after 1 ns;
    -- check results on falling edge
    wait until falling_edge(clk);
    if y /= y_expected then
        report "Error: y = " & std_logic'image(y);
        errors := errors + 1;
    end if;
    vectornum := vectornum + 1;
end loop;
```

Testbench s a-kontrolou na báze testovacieho súboru

```
if (errors = 0) then
    report "NO ERRORS -- " & integer'image(vectornum) &
           " tests completed successfully."
    severity failure;
else
    report integer'image(vectornum) &
           " tests completed, errors = " &
           integer'image(errors)
    severity failure;
end if;
end process;
end;
```

Referencia

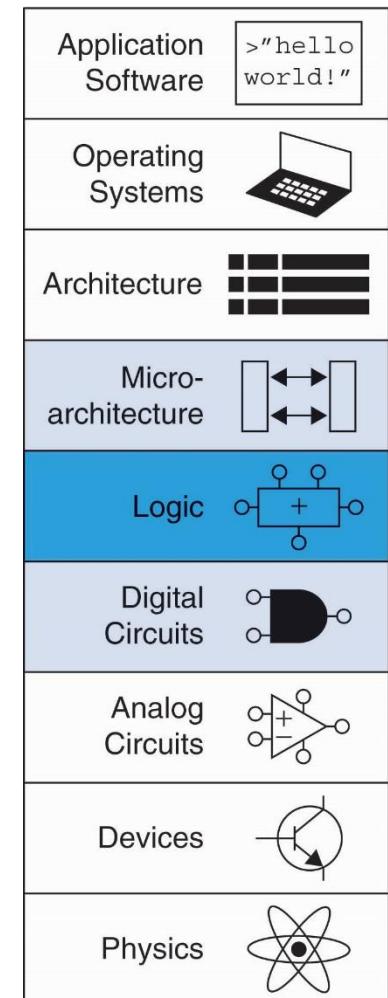
- David Money Harris and Sarah L. Harris,
Digital Design and Computer Architecture,
Chapter 4: Hardware Description Languages,
Second Edition © 2012 by Elsevier Inc.

Architektúry počítačových systémov

2. rok ZS

Osnova

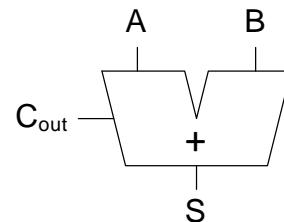
- **Obvody na vykonávanie operácií**
- **Reprezentácia reálnych čísel**
- **Obvody určené na uchovávanie informácií**
- **Logické polia**



- **Komponenty číslicových systémov:**
 - hradlá, multiplexory, dekodéry, obvody na realizáciu aritmetických a logických operácií, počítadlá, registre, pamäte, ...
- **Pri návrhu a kompozícii sa uplatňujú princípy HMJ:**
 - Rozdelenie systémov do modulov
 - Moduly majú dobre zadefinované funkcie a rozhrania
 - Možnosť znovupoužitia

1-bitové sčítacky

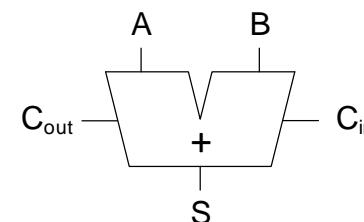
polovičná
sčítacka



A	B	C _{out}	S
0	0		
0	1		
1	0		
1	1		

$$\begin{matrix} S \\ C_{\text{out}} \end{matrix} =$$

úplná
sčítacka

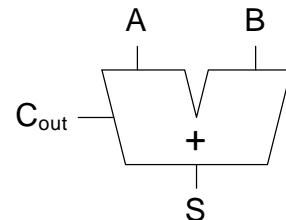


C _{in}	A	B	C _{out}	S
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	
1	0	0	0	
1	0	1	1	
1	1	0	0	
1	1	1	1	

$$\begin{matrix} S \\ C_{\text{out}} \end{matrix} =$$

1-bitové sčítacky

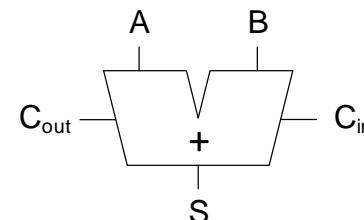
polovičná
sčítacka



A	B	C _{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$\begin{aligned} S &= \\ C_{\text{out}} &= \end{aligned}$$

úplná
sčítacka

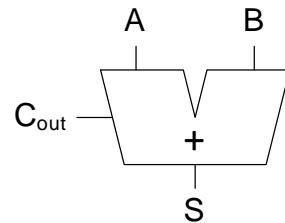


C _{in}	A	B	C _{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$\begin{aligned} S &= \\ C_{\text{out}} &= \end{aligned}$$

1-bitové sčítacky

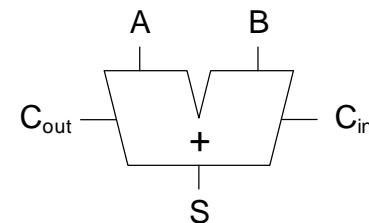
polovičná
sčítadla



A	B	C _{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$\begin{aligned}S &= A \oplus B \\C_{\text{out}} &= AB\end{aligned}$$

úplná
sčítadla



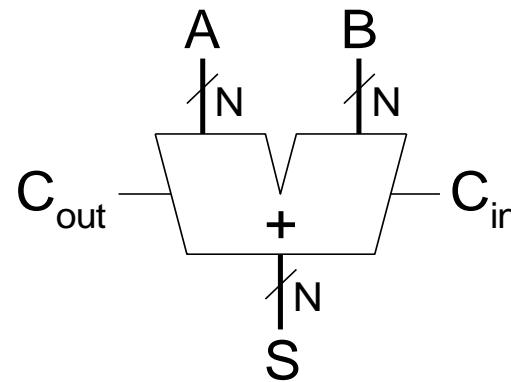
C _{in}	A	B	C _{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$\begin{aligned}S &= A \oplus B \oplus C_{\text{in}} \\C_{\text{out}} &= AB + AC_{\text{in}} + BC_{\text{in}}\end{aligned}$$

Sčítačky

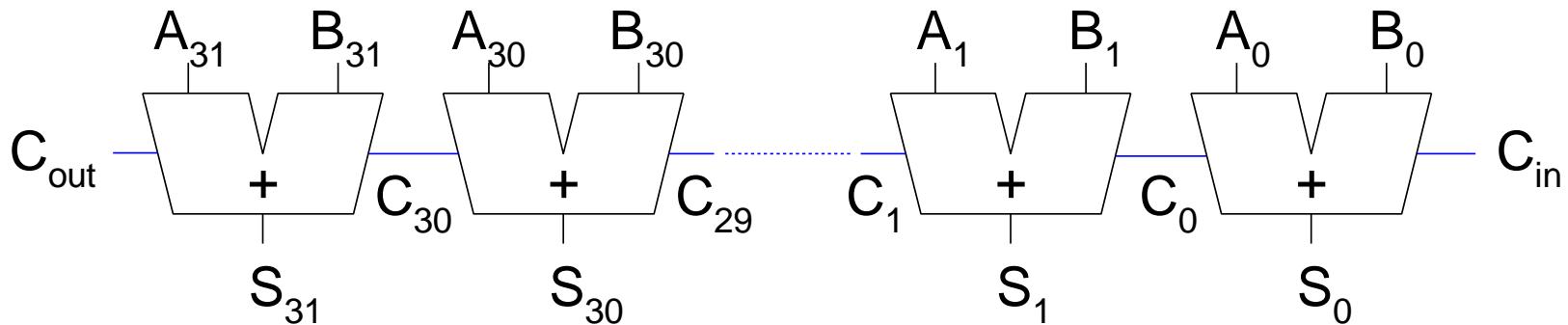
- Sčítanie dvoch N-bitových čísel
 - Sčítačka so sériovým prenosom (pomalá)
 - Sčítačka so zrýchleným prenosom (rýchlejšia)
 - Paralelné prefixové sčítačky (najrýchlejšie)
- Sčítačky so ZP a prefixové sčítačky sú rýchle ale potrebujú viac hradiel na svoju realizáciu

Symbol



Sériová sčítačka

- Angl. Ripple-Carry Adder
- Kaskáda 1b sčítačiek
- Prenos sa šíri od najnižšieho až po najvyšší rád
- Výhoda: jednoduchý návrh
- Nevýhoda: **pomalá**



Sériová sčítačka - oneskorenie

$$t_{ripple} = N \times t_{FA}$$

kde t_{FA} je oneskorenia na úplnej sčítačke

Sčítačka so zrýchleným prenosom

- Angl. Carry-Lookahead Adder (CLA)
- Výpočet prenosu (C_{out}) z k -bit veľkého bloku pomocou *vlastného* a *tranzitívneho* prenosu
- **Pojmy:**
 - Prenos z *i-teho rádu* nastane vtedy ak je to dané *vlastným prenosom* (G_i) alebo v súčinnosti s indikovaným príznakom *tranzitívneho prenosu* (P_i)
 - Prenos v *i-tom ráde* nastane ak $A_i = B_i = 1$.

$$G_i = A_i B_i$$

- Prenos v *i-tom ráde* môže nastáť ak aspoň jeden z A_i a B_i je 1.

$$P_i = A_i + B_i$$

- Prenos (C_i) je definovaný

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$

Sčítačka so zrýchleným prenosom

- **Step 1:** Počítaj G_i a P_i pre všetky i v bloku k
- **Step 2:** Počítaj $G_{i:j}$ a $P_{i:j}$ pre blok k
- **Step 3:** Počítaj C_i na báze $G_{i:j}$, $P_{i:j}$ a C_{i-1}

Sčítačka so zrýchleným prenosom

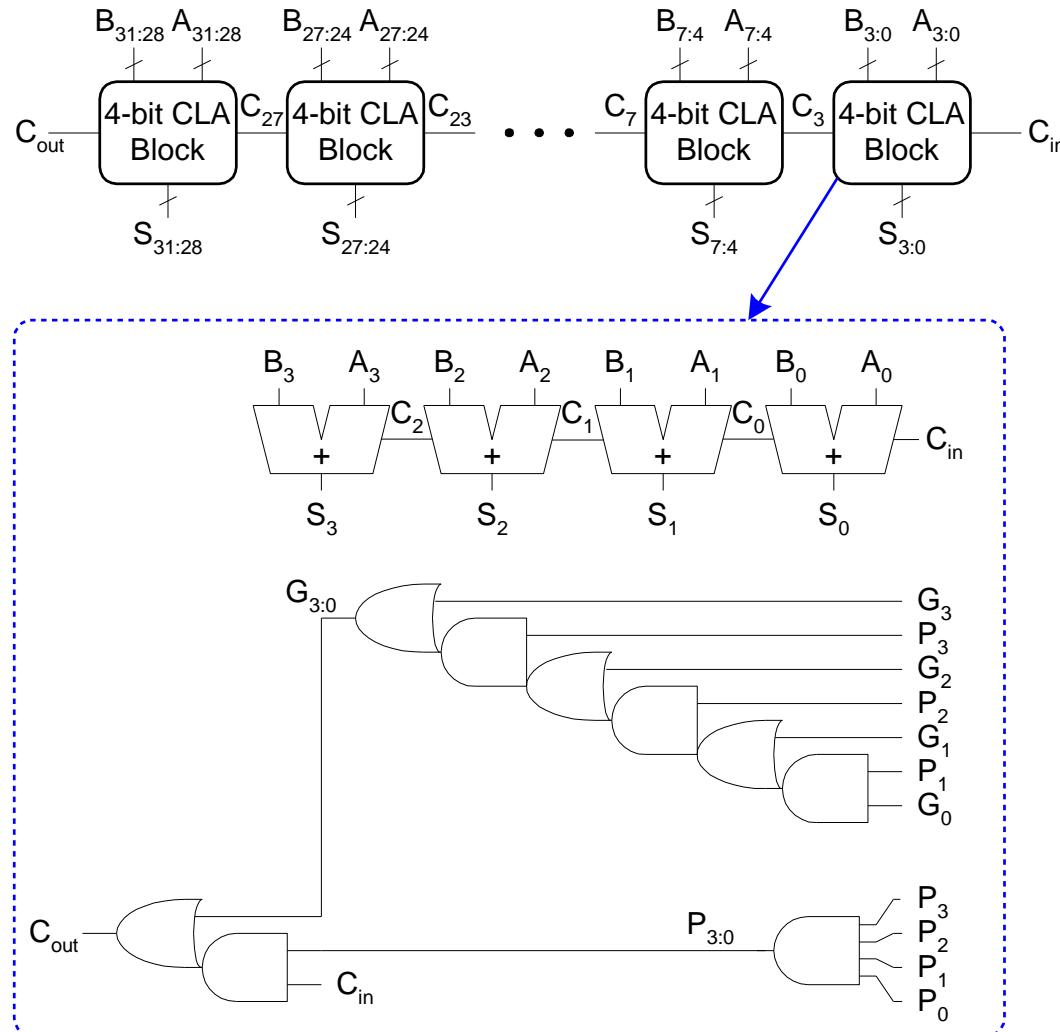
- Príklad: 4b blok ($G_{3:0}$ a $P_{3:0}$) :

$$\begin{aligned}G_{3:0} &= G_3 + P_3(G_2 + P_2(G_1 + P_1 G_0)) \\P_{3:0} &= P_3 P_2 P_1 P_0\end{aligned}$$

- Všeobecne,

$$\begin{aligned}G_{i:j} &= G_i + P_i(G_{i-1} + P_{i-1}(G_{i-2} + P_{i-2} G_j)) \\P_{i:j} &= P_i P_{i-1} P_{i-2} P_j \\C_i &= G_{i:j} + P_{i:j} C_{i-1}\end{aligned}$$

32b:4b CLA



CLA - oneskorenie

Pre N -bitovú CLA s k -bitovými blokmi:

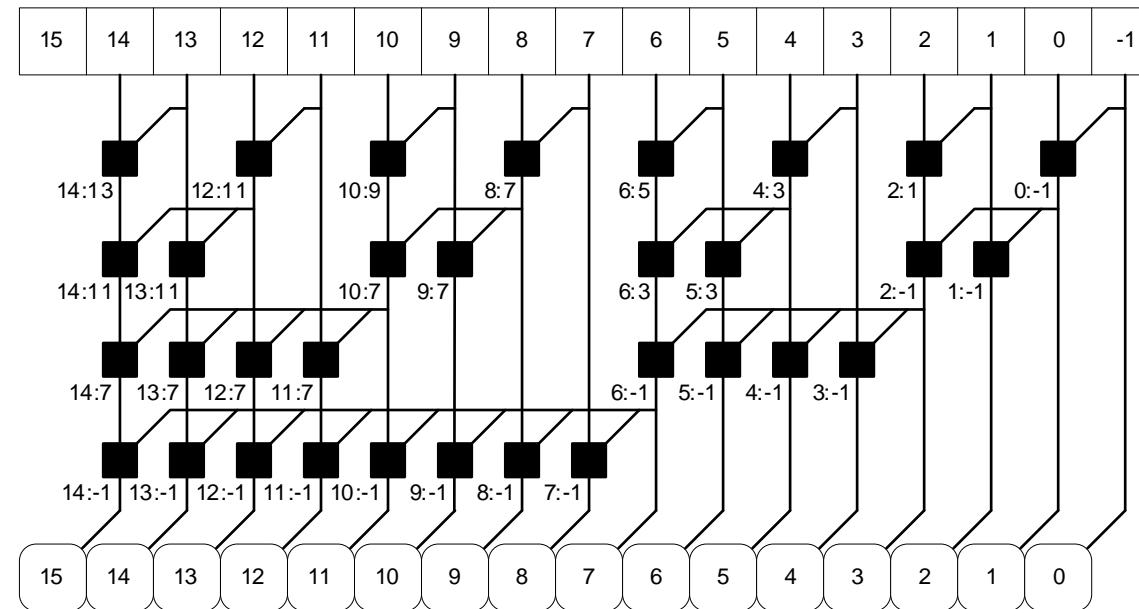
$$t_{CLA} = t_{pg} + t_{pg_block} + (N/k - 1)t_{AND_OR} + kt_{FA}$$

- t_{pg} : oneskorenie spôsobené P_i, G_i
- t_{pg_block} : oneskorenie spôsobené $P_{i:j}, G_{i:j}$
- t_{AND_OR} : oneskorenie dané prenosom z C_{in} na C_{out} ; AND/OR hradlá na konci k -bitového CLA bloku

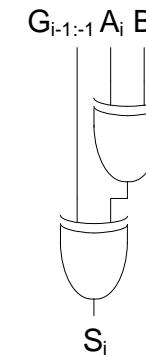
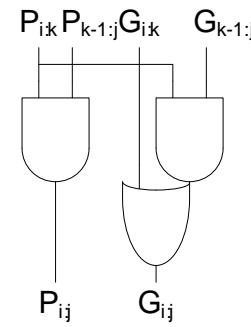
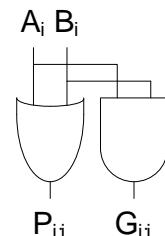
Prefixová sčítáčka

- Angl. Prefix Adder
- Niekoľko typov:
 - 1973: Kogge-Stone
 - 1980: Ladner-Fisher
 - 1981: H. Ling
 - 1982: Brent-Kung
 - 1987: Han-Carlson
 - 2001: Beaumont-Smith

Schéma prefixovej sčítáčky



Legenda



Prefixová sčítačka - oneskorenie

$$t_{PA} = t_{pg} + \log_2 N(t_{pg_prefix}) + t_{XOR}$$

- t_{pg} : oneskorenie dané výpočtom v „bielych štvorčekoch“ (AND alebo OR hradlo)
- t_{pg_prefix} : oneskorenie dané výpočtom v „čiernych štvorčekoch“ (AND-OR hradlo)

Porovnávanie sčítáčiek

Porovnaj oneskorenia 32-bit RCA, CLA a prefixovej sčítaceky

- CLA pozostáva z 4-bitových blokov
- Oneskorenie 2-vstupových hradieb = 100 ps
- Oneskorenie úplnej sčítaceky = 300 ps

Porovnávanie sčítáčiek

Porovnaj oneskorenia 32-bit RCA, CLA a prefixovej sčítáčky

- CLA pozostáva z 4-bitových blokov
- Oneskorenie 2-vstupových hradieb = 100 ps; Oneskorenie úplnej sčítáčky = 300 ps

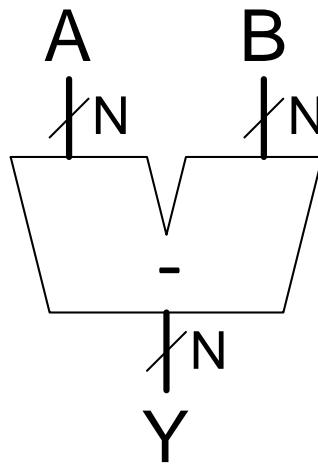
$$\begin{aligned} t_{\text{ripple}} &= N \times t_{\text{FA}} = 32 \times (300 \text{ ps}) \\ &= \mathbf{9,6 \text{ ns}} \end{aligned}$$

$$\begin{aligned} t_{\text{CLA}} &= t_{\text{pg}} + t_{\text{pg_block}} + (N/k - 1) \times t_{\text{AND_OR}} + k \times t_{\text{FA}} \\ &= [100 + 600 + (7) \times 200 + 4 \times (300)] \text{ ps} \\ &= \mathbf{3,3 \text{ ns}} \end{aligned}$$

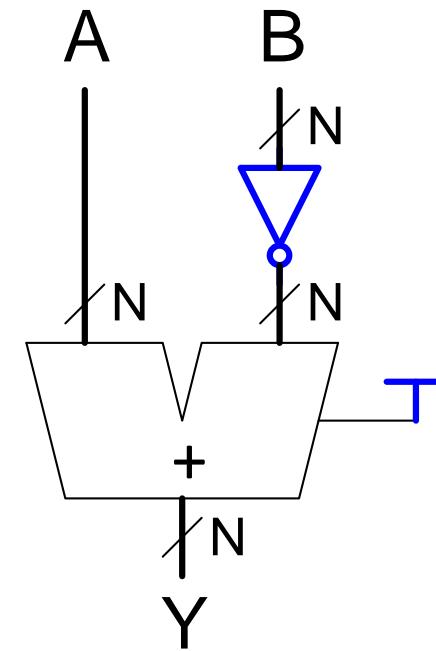
$$\begin{aligned} t_{\text{PA}} &= t_{\text{pg}} + \log_2 N \times (t_{\text{pg_prefix}}) + t_{\text{XOR}} \\ &= [100 + \log_2 32 \times (200) + 100] \text{ ps} \\ &= \mathbf{1,2 \text{ ns}} \end{aligned}$$

Odčítačka

Symbol

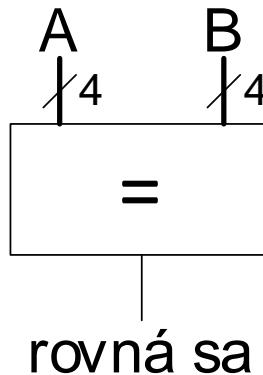


Implementácia

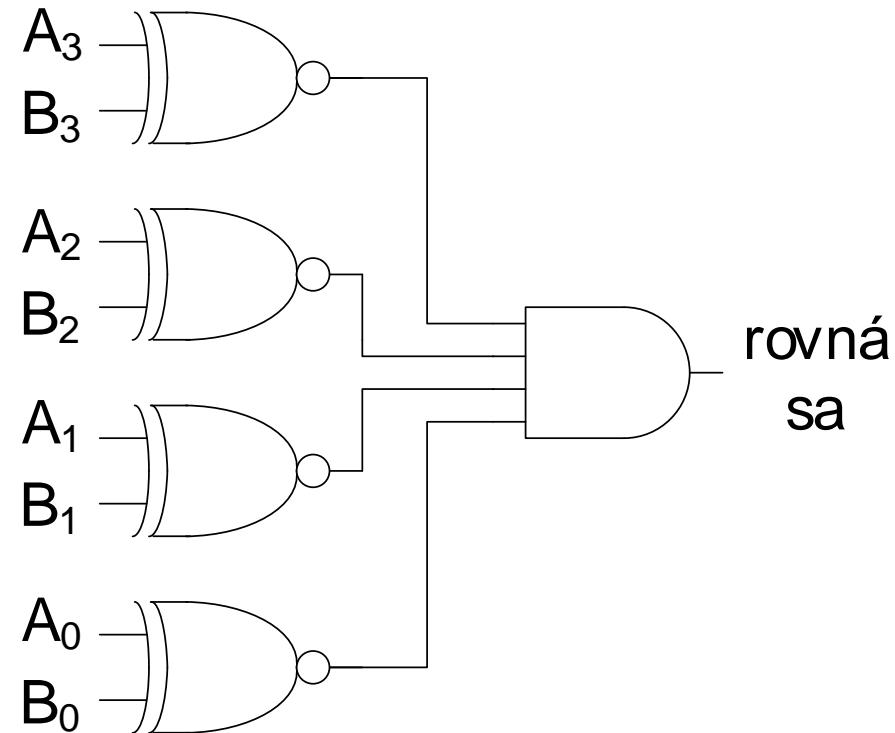


Komparátor : rovnosť

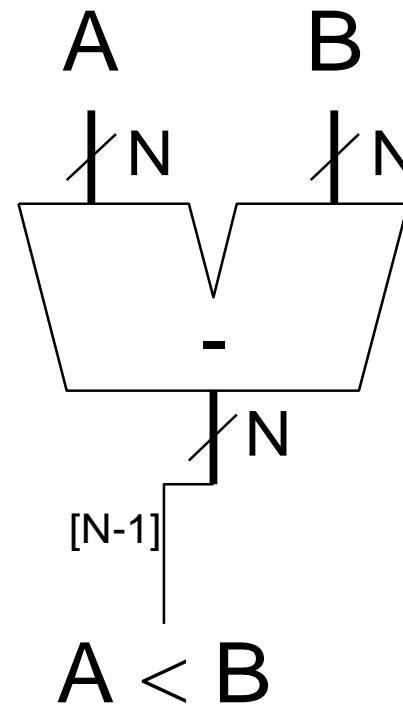
Symbol



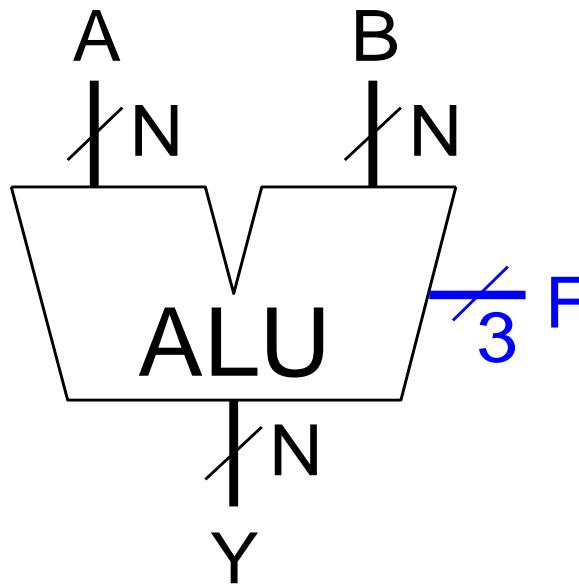
Implementácia



Komparátor : menší

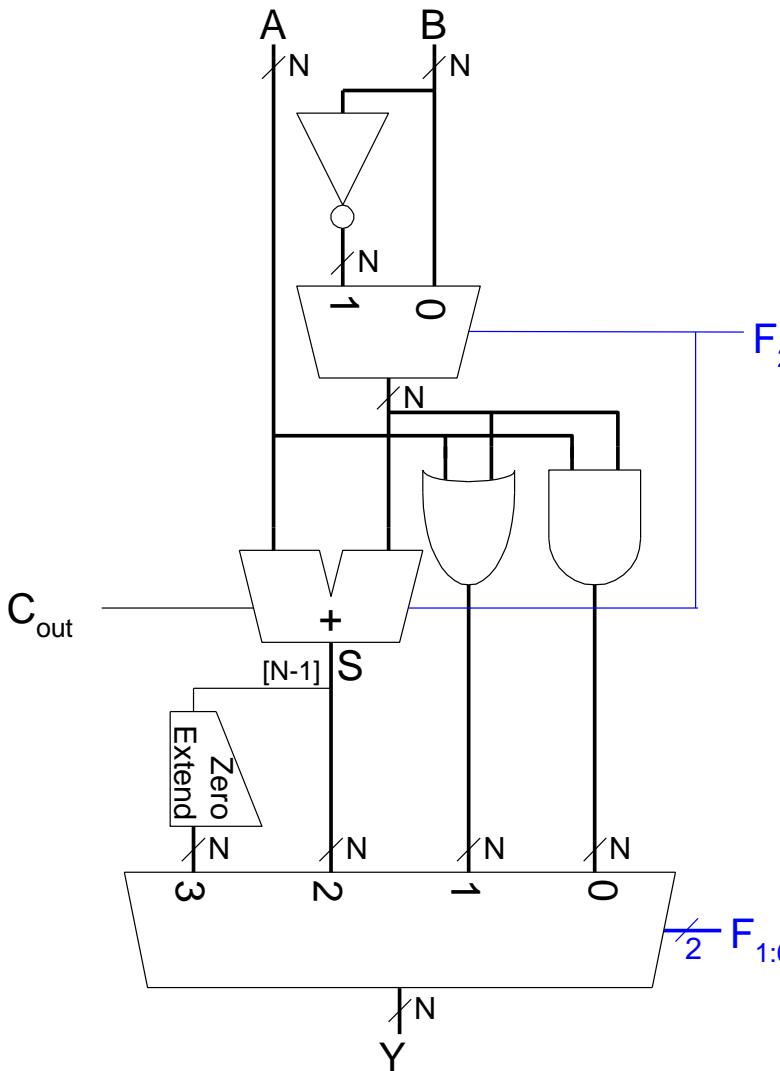


Aritmeticko-logická jednotka (ALJ)



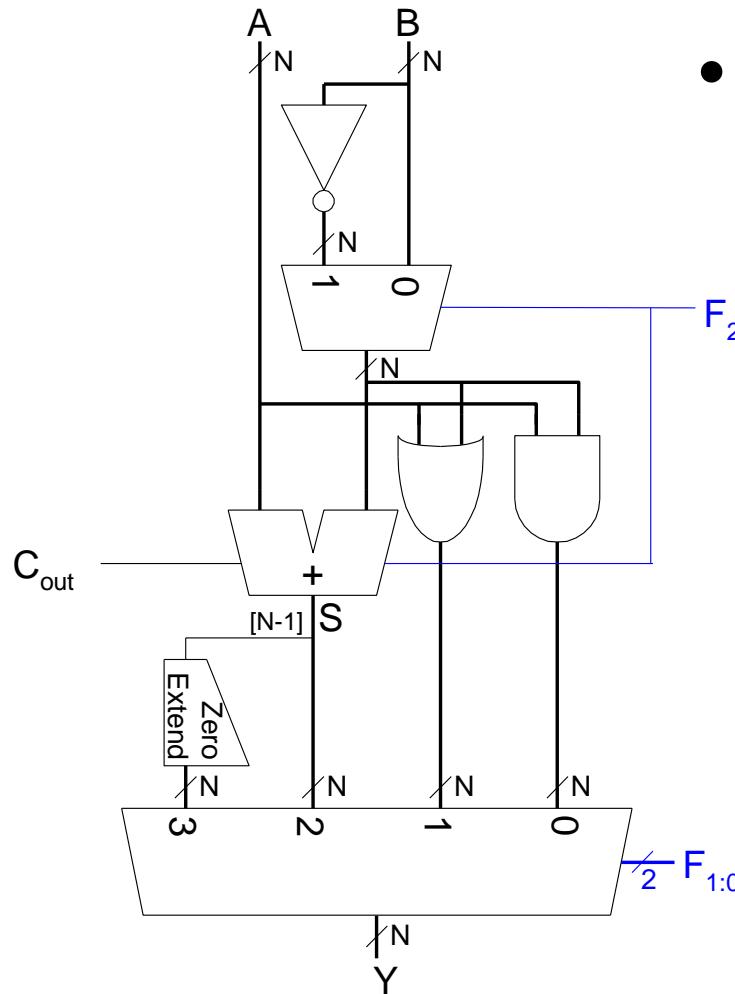
$F_{2:0}$	Funkcia
000	$A \& B$
001	$A B$
010	$A + B$
011	not used
100	$A \& \sim B$
101	$A \sim B$
110	$A - B$
111	SLT

Jednoduchá ALJ



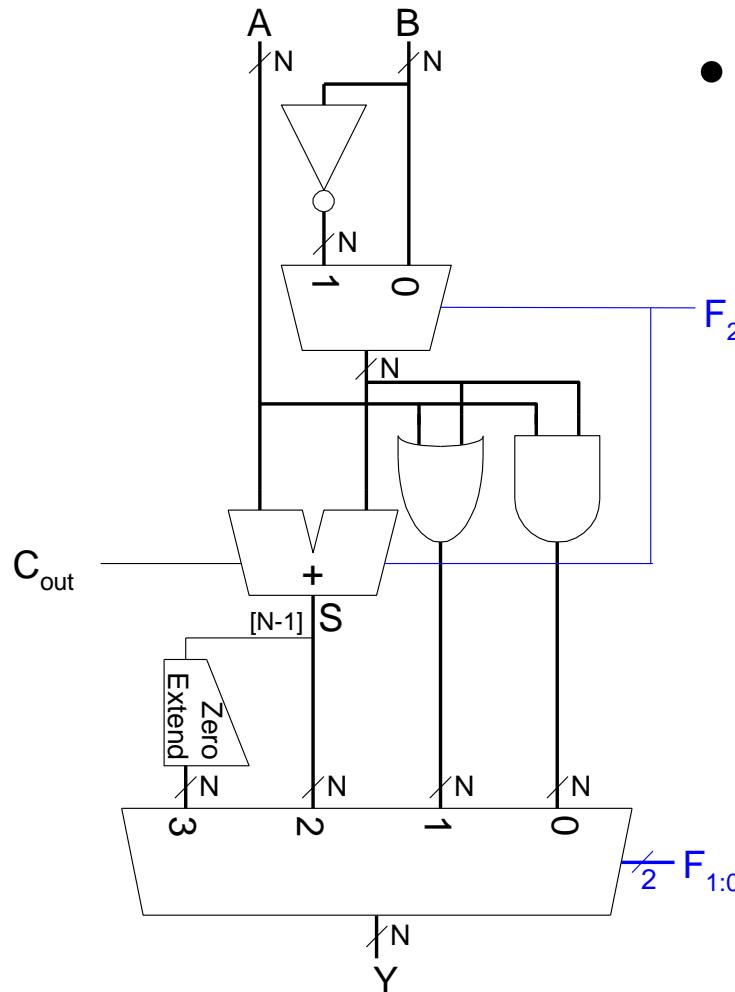
F _{2:0}	Funkcia
000	A & B
001	A B
010	A + B
011	not used
100	A & ~B
101	A ~B
110	A - B
111	SLT

Jednoduchá ALJ - príklad



- Vykonaj operáciu SLT (angl. set on less than) nad $A = 25$ a $B = 32$

Jednoduchá ALJ - príklad



- Vykonaj operáciu SLT nad $A = 25$ and $B = 32$
 - Nakoľko $A < B$, so Y má byť 1 (0x00000001)
 - $F_{2:0} = 111$
 - $F_2 = 1$ (sčítačka funguje ako odčítačka), $25 - 32 = -7$
 - Pre -7 znamienkový (msb) bit = 1 ($S_{31} = 1$)
 - $F_{1:0} = 11$ multiplexor zabezpečí $Y = S_{31}$ (zero extended) = 0x00000001.

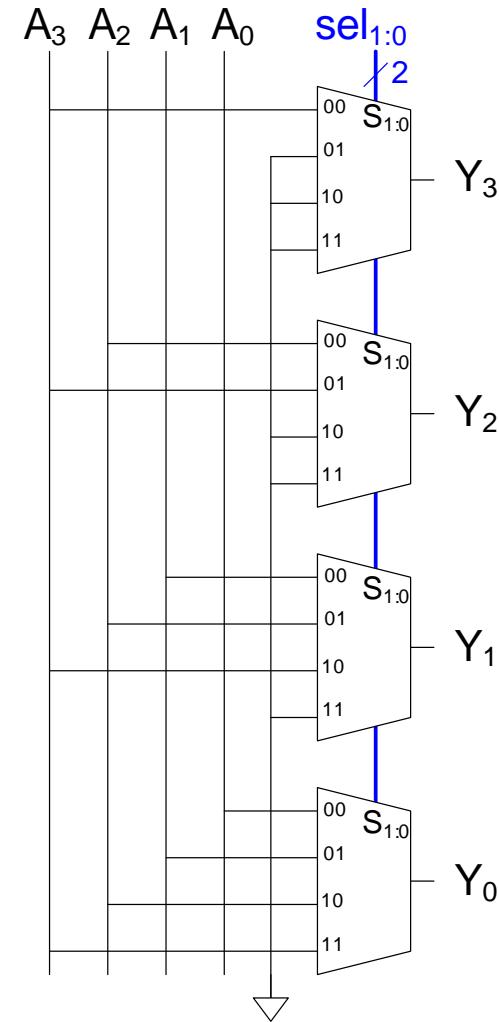
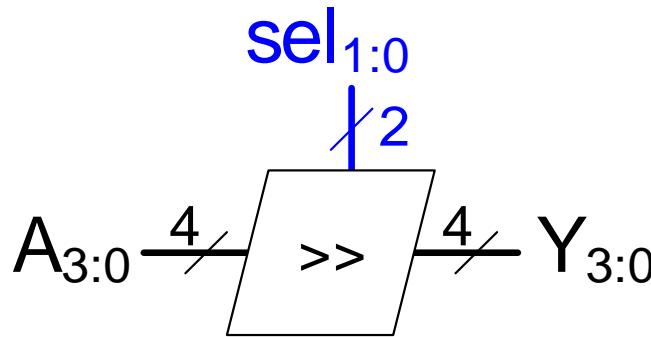
Posuvné (funkčné) jednotky (PJF)

- **Logický posun:** posun doprava alebo doľava s pridaním 0
 - $11001 \gg 2 =$
 - $11001 \ll 2 =$
- **Aritmetický posun:** posun doľava je ako logický posun doľava; pri posune doprava sa kopíruje hodnota msb.
 - $11001 \ggg 2 =$
 - $11001 \lll 2 =$
- **Rotácia:** bit ktorý sa pri rotácií dostane mimo rámec zobrazenia sa objaví v pozícii msb (doprava) alebo lsb (doľava)
 - $11001 \text{ ROR } 2 =$
 - $11001 \text{ ROL } 2 =$

Posuvné jednotky

- **Logický posun:**
 - $11\textcolor{red}{001} \gg 2 = \textcolor{blue}{00}\textcolor{red}{110}$
 - $11\textcolor{red}{001} \ll 2 = \textcolor{red}{001}00$
- **Aritmetický posun:**
 - $11\textcolor{red}{001} \ggg 2 = \textcolor{blue}{11}\textcolor{red}{110}$
 - $11\textcolor{red}{001} \lll 2 = \textcolor{red}{001}00$
- **Rotácia:**
 - $11\textcolor{red}{001} \text{ ROR } 2 = 01\textcolor{blue}{110}$
 - $11\textcolor{red}{001} \text{ ROL } 2 = \textcolor{red}{001}11$

Návrh posuvnej jednotky (KLO)



PFJ v úlohe násobičky / deličky

- $A \ll N = A \times 2^N$
 - $00001 \ll 2 = 00100 \quad (1 \times 2^2 = 4)$
 - $11101 \ll 2 = 10100 \quad (-3 \times 2^2 = -12)$
- $A \ggg N = A \div 2^N$
 - $01000 \ggg 2 = 00010 \quad (8 \div 2^2 = 2)$
 - $10000 \ggg 2 = 11100 \quad (-16 \div 2^2 = -4)$

Násobička

Decimal

$$\begin{array}{r} 230 \\ \times 42 \\ \hline 460 \\ + 920 \\ \hline 9660 \end{array}$$

multiplicand

multiplier

partial
products

result

Binary

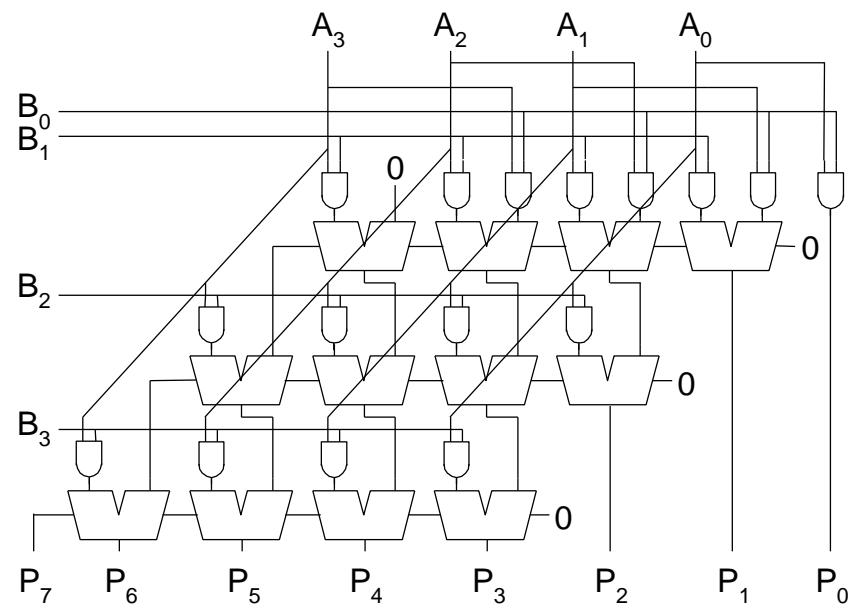
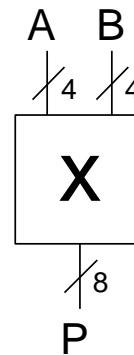
$$\begin{array}{r} 0101 \\ \times 0111 \\ \hline 0101 \\ 0101 \\ 0101 \\ + 0000 \\ \hline 0100011 \end{array}$$

$$230 \times 42 = 9660$$

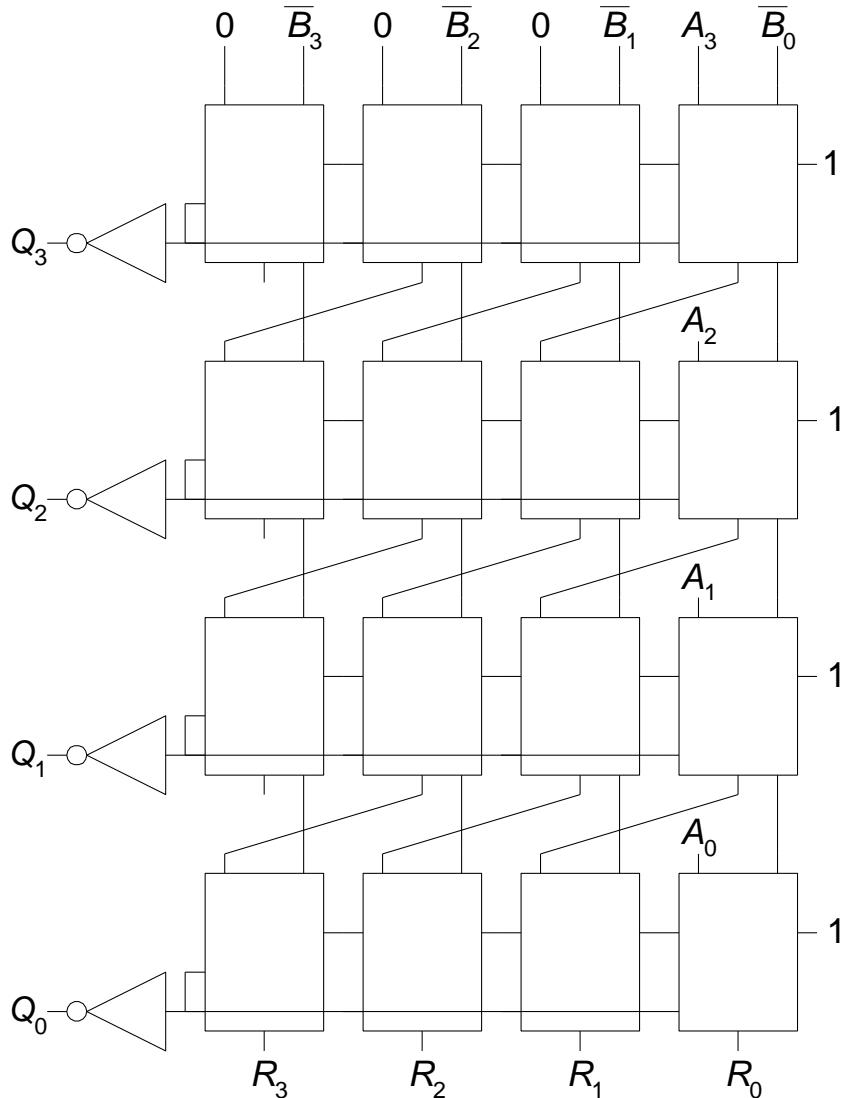
$$5 \times 7 = 35$$

Násobička 4×4

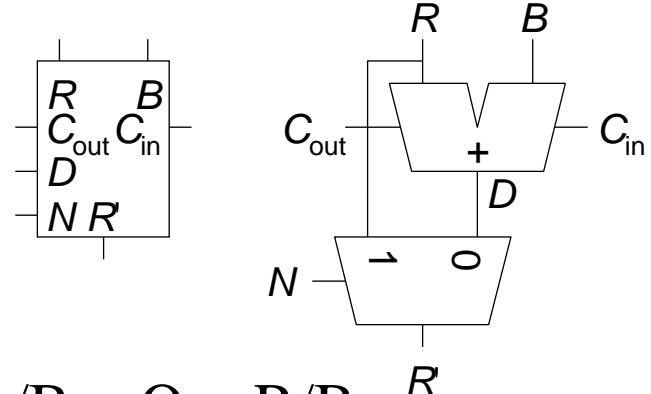
$$\begin{array}{r} & A_3 & A_2 & A_1 & A_0 \\ x & B_3 & B_2 & B_1 & B_0 \\ \hline A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\ A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 \\ A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 \\ + & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 \\ \hline P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0 \end{array}$$



Delička 4×4



Legenda



$$A/B = Q + R/B$$

Algoritmus:

$$R' = 0$$

for $i = N-1$ to 0

$$R = \{R' << 1. A_i\}$$

$$D = R - B$$

if $D < 0$, $Q_i=0$, $R' = R$

else $Q_i=1$, $R' = D$

$$R' = R$$

Reprezentácia čísel

- ČP používa binárnu č. sústavu
 - **Prirodzené čísla**
 - Bezznamienkové binárne čísla
 - **Celé čísla**
 - Priamy kód
 - Doplnkový kód
 - Kód s posunutím
- A čo v prípade **reálnych čísel**?

Reálne čísla

- Reprezentácia v podobe:

- Pevná rádová čiarka (PRČ)
 - Q notácia
 - Pohyblivá rádová čiarka (PHRČ)

significand \times *base*^{*exponent*}

$$1, \alpha_{-1} \alpha_{-2} \dots \alpha_{-k} \times b^E$$
$$15 = 1,5 \times 10^1$$

Pevná rádová čiarka

- Vyjadri 6,75 v Q4.4 (= 4 bity pre celú časť vrátane znamienka a 4 pre desatinnú časť):

01101100

0110,_{blue}1100

$$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

- Pozícia desatinnej čiarky je striktne určená
- Je potrebné sa dohodnúť na počte bitov vyhradených pre celú a pre desatinnú časť čísla

Pevná rádová čiarka

- Vyjadri $7,5_{10}$ v Q4.4.
- Vyjadri $-7,5_{10}$ v Q4.4.

Pevná rádová čiarka

- Vyjadri $7,5_{10}$ v Q4.4.

01111000

- Vyjadri $-7,5_{10}$ v Q4.4.

10001000

<https://www.allaboutcircuits.com/technical-articles/fixed-point-representation-the-q-format-and-addition-examples/>

Pevná rádová čiarka

- Reprezentácia čísla
 - Priamy kód
 - Doplňkový kód
- Príklad: Vyjadri $-7,5_{10}$ v Q4.4
 - Priamy kód:
 - Doplňkový kód:

Pevná rádová čiarka

- Reprezentácia čísla
 - Priamy kód
 - Doplňkový kód
- Príklad: Vyjadri $-7,5_{10}$ v Q4.4
 - Priamy kód:

11111000

- Doplňkový kód:

1. $+7,5:$ 01111000

2. Invertuj bity: 10000111

3. Pripočítaj +1: + 1

10001000

Pohyblivá rádová čiarka

- Čísla v pohyblivej rádovej čiarke sa vyjadrujú v tvare

$$\pm M \times b^E$$

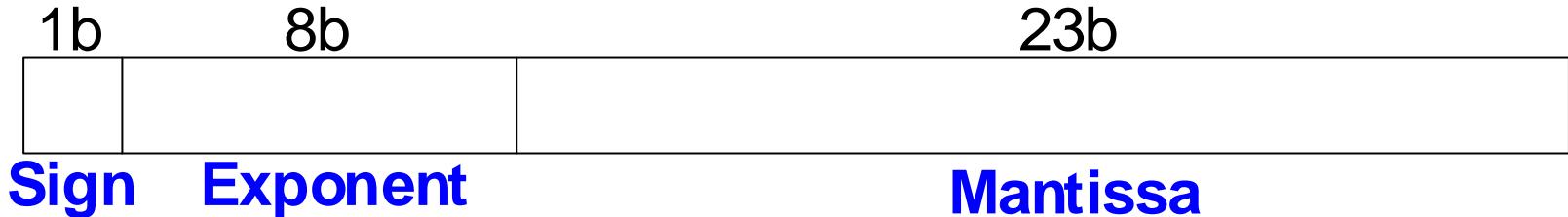
- **M** = mantisa
- **B** = základ
- **E** = exponent

- Napríklad

$$273 = 2,73 \times 10^2$$

- $M = 2,73$; $B = 10$; $E = 2$

Pohyblivá rádová čiarka



- **Príklad:** Vyjadrite reálne číslo $228,0_{10}$ v 32-bitovom formáte

Budú prezentované tri verzie zápisu – posledná je podľa **IEEE 754 standardu**

Pohyblivá rádová čiarka – 1 verzia

1. Binárna reprezentácia:

$$228_{10} = 11100100_2$$

2. Vedecký binárny zápis:

$$11100100_2 = 1,11001_2 \times 2^7$$

3. 32-bitovú reprezentácia čísla v pohyblivej rádovej čiarke

- Znamienkový bit = 0 (kladné)
- 8-bitový exponent = 7
- Mantisa na 23 bitoch (doplnené 0)

1b	8b	23b
Zb	Exponent	Mantisa
0	00000111	11 1001 0000 0000 0000 0000

Pohyblivá rádová čiarka – 2.verzia

- Všimneme si, že mantisa začína s 1:
 - $228_{10} = 11100100_2 = \textcolor{red}{1},\textcolor{blue}{11001} \times 2^7$
- Preto nie je potrebné prvú jednotku uchovať
 - Je *vždy* prítomná
 - Ušetrili sme jeden bit → Zväčšil sa rozsah

1b	8b	23b
Zb	Exponent	Mantisa
0	00000111	110 0100 0000 0000 0000 0000

Pohyblivá rádová čiarka – 3.verzia

- Charakteristika čísla: bias = 127 (0111111_2)
 - Exponent s posunutou nulou = exponent + bias
 - Exponent = 7 a preto charakteristika čísla sa rovná:
 $127 + 7 = 134 = 0x10000110_2$
- Formát jednoduchej presnosti podľa IEEE 754

1b	8b	23b
0	10000110	110 0100 0000 0000 0000 0000
Zb	Charak- teristika	Mantisa

v hexadecimálnom tvare: **0x43640000**

Pohyblivá rádová čiarka

Zobraz $-58,25_{10}$ podľa IEEE 754 vo formáte JP

1. Binárna reprezentácia:

$$58,25_{10} = \textcolor{blue}{111010,01}_2$$

2. Vedecký zápis:

$$\textcolor{blue}{1,1101001} \times 2^5$$

3. Výpočet charakteristiky a zápis na 32b:

Znamienkový bit: **1** (záporné)

Charakteristika: $(127 + 5) = 132 = \textcolor{blue}{10000100}_2$

Mantisa: **110 1001 0000 0000 0000 0000**

1b	8b	23b
Zb	Charakt.	Mantisa
1	100 0010 0	110 1001 0000 0000 0000 0000

v hexadecimálnom tvare: **0xC2690000**

Pohyblivá rádová čiarka

Číslo	Zb	Charak.	Mantisa
0	X	00000000	00000000000000000000000000000000
∞	0	11111111	00000000000000000000000000000000
$-\infty$	1	11111111	00000000000000000000000000000000
NaN	X	11111111	non-zero

IEEE-754 formáty

- **Formát jednoduchej presnosti:**
 - 32 bitov
 - 1b znamienko, 8b charakteristika, 23b mantisa
 - bias = 127
- **Formát dvojitej presnosti:**
 - 64 bitov
 - 1b znamienko, 11b charakteristika, 52b mantisa
 - bias = 1023

PHRČ: zaokrúhlenie

- **Pretečenie:** číslo je príliš veľké pre zobrazenie v danom formáte
- **Podtečenie:** číslo je príliš malé pre zobrazenie v danom formáte
- **Techniky zaokrúhl'ovania:**
 - Nadol (down)
 - Nahor (up)
 - Smerom k nule (toward zero)
 - Na najbližšiu hodnotu (to nearest)
- **Príklad:** zaokrúhli $1,100101$ ($1,578125$) na 3b za d. čiarkou
 - Nadol: 1,100
 - Nahor: 1,101
 - Smerom k nule: 1,100
 - Na najbližšiu hodnotu: 1,101 ($1,625$ je bližšie k $1,578125$ než $1,5$)

Sčítanie čísel vyjadrených podľa IEEE-754

1. Urč hodnotu zb, charakteristiku a mantisu čísel
2. Pridaj „vedúcu“ 1 k mantisám sčítancov
3. Porovnaj exponenty, vyrovnaj exponenty ak je to nutné
4. Vykonaj posuv mantisy ak je to potrebné
5. Sčítaj/odčítaj mantisy sčítancov
6. Normalizuj mantisu a uprav hodnotu exponentu výsledku ak to je potrebné
7. Zaokrúhli výsledok
8. Zobraz výsledok podľa IEEE-754

Sčítanie čísel vyjadrených podľa IEEE-754 vo formáte JP - príklad

Sčítaj nasledujúce dve čísla zobrazené podľa IEEE-754 a uložené vo formáte jednoduchej presnosti:

0x3FC00000

0x40500000

Sčítanie čísel vyjadrených podľa IEEE-754 vo formáte JP - príklad

1. Urč hodnoty zb, charakteristik a mantís čísel

1 bit	8 bits	23 bits
Sign	Exponent	Fraction
0	01111111	100 0000 0000 0000 0000 0000
1 bit	8 bits	23 bits
0	10000000	101 0000 0000 0000 0000 0000
Sign	Exponent	Fraction

Prvý operand (N1): $S = 0, E = 127, F = .1$

druhý operand (N2): $S = 0, E = 128, F = .101$

2. Pridaj „vedúcu“ 1 k mantisám sčítancov

N1: 1,1

N2: 1,101

Sčítanie čísel vyjadrených podľa IEEE-754 vo formáte JP - príklad

3. Porovnaj exponenty

$$127 - 128 = -1 \rightarrow \text{posuň N1 1 bit doprava}$$

4. Vykonaj posuv mantisy ak je to potrebné

$$\text{posuň N1 1 bit doprava : } 1,1 >> 1 = 0,11 \ (\times 2^1)$$

5. Sčítaj mantisy

$$\begin{array}{r} 0,11 \times 2^1 \\ + 1,101 \times 2^1 \\ \hline 10,011 \times 2^1 \end{array}$$

Sčítanie čísel vyjadrených podľa IEEE-754 vo formáte JP - príklad

6. Normalizuj mantisu a uprav hodnotu exponentu

$$10,011 \times 2^1 = 1,0011 \times 2^2$$

7. Zaokrúhlí výsledok

Nie je potrebné (výsledok je presne zobraziteľný pomocou 23b)

8. Zobraz výsledok podľa IEEE-754

$$S = 0, E = 2 + 127 = 129 = 10000001_2, F = 001100..$$

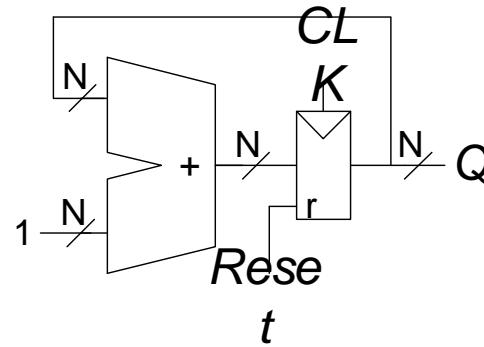
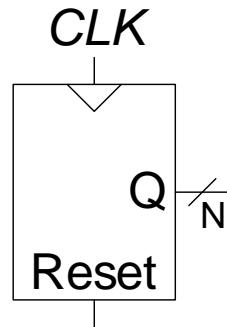
1 bit	8 bits	23 bits
Sign	Exponent	Fraction
0	10000001	001 1000 0000 0000 0000 0000

v hexadecimálnom tvare: **0x40980000**

Počítadlo

- Hodnota sa inkrementuje/dekrementuje pri každom takte
 - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001...
- Prípady použitia:
 - Riadenia synchronizačných impulzov v digitálnych zobrazovačoch (VGA monitory)
 - Programové počítadlo

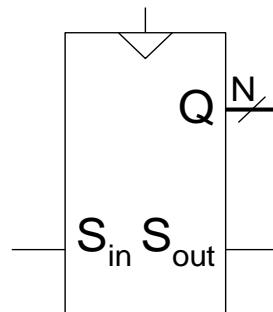
Symbol Implementácia



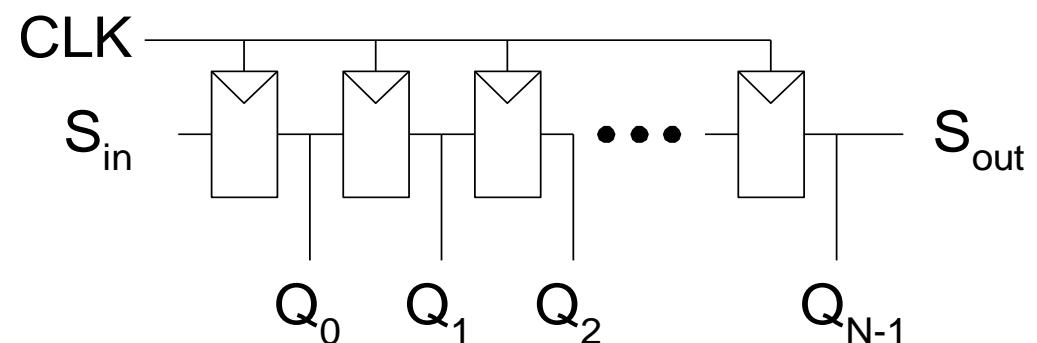
Posuvný register

- Pri každom takte vstupuje do a vystupuje z regisra 1b
- *Prevod sériového kódu na kód paralelný:*
- Vstupné dáta sú privezené na vstup (S_{in}) bit po bite a po patričnom počte impulzov sú tieto údaje k dispozícii na výstupoch ($Q_{0:N-1}$)

Symbol:

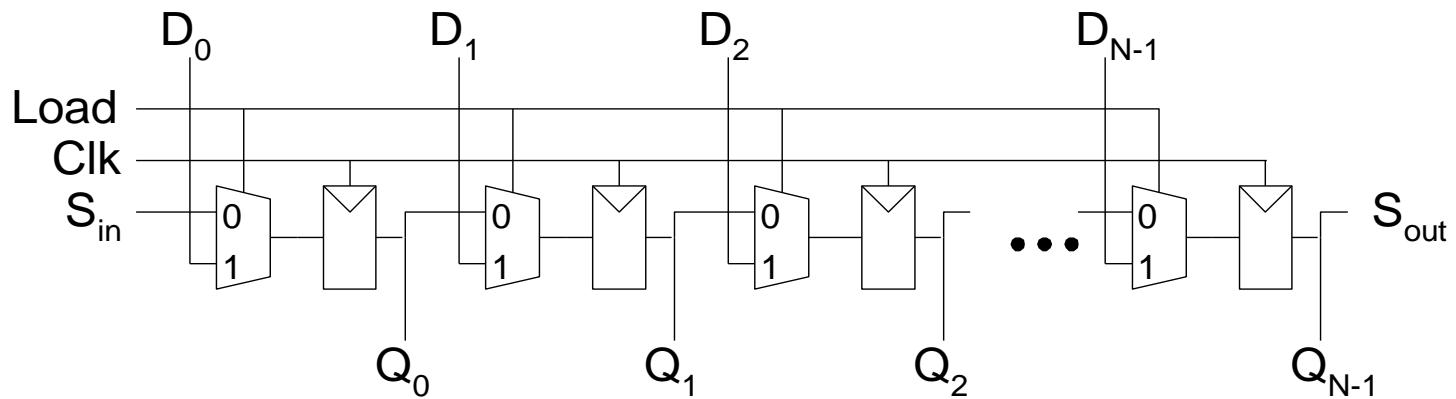


Implementácia



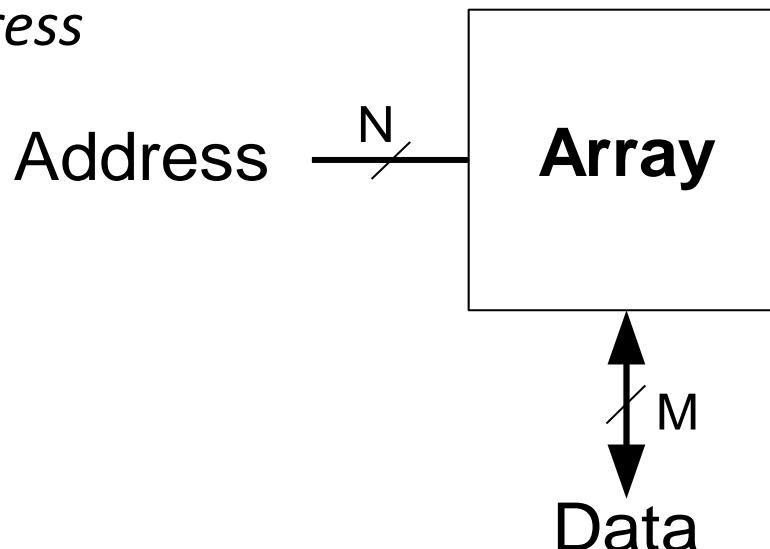
Posuvný register s paralelným nastavením

- Ak $Load = 1$, pracuje ako (bežný) N -bitový register
- Ak $Load = 0$, pracuje ako posuvný register
- Môže pracovať ako prevodník sériového kódu na kód paralelný (S_{in} to $Q_{0:N-1}$) alebo ako prevodník paralelného kódu na kód sériový ($D_{0:N-1}$ to S_{out})



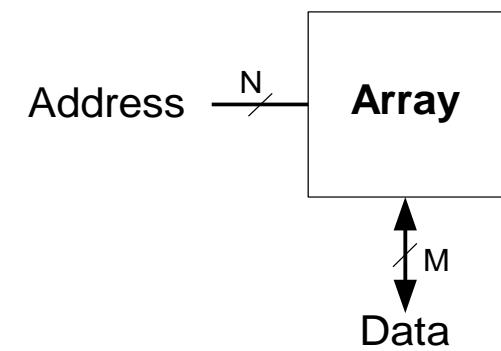
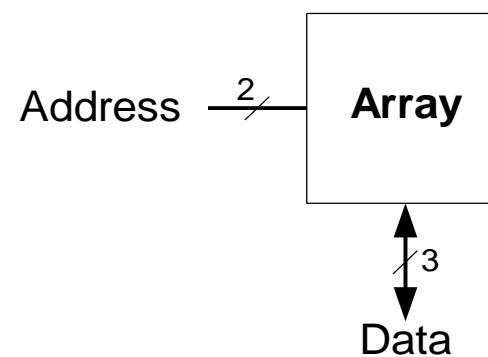
Pamäte

- Slúži na pamätanie údajov
- 3 základné typy:
 - Dynamic random access memory (DRAM)
 - Static random access memory (SRAM)
 - Read only memory (ROM)
- M -bit široký dátový výstup/vstup *Data* pre čítanie/zápis do pamäte na adresu definovanú N -bit širokým vstupom *Address*



Pamäť

- 2D pole pamäťových buniek (*bit cells*)
 - Pamäťová matica
- Každý element dokáž uchovať 1b údaj
- N adresových bitov a M dátových bitov:
 - 2^N riadkov a M stĺpcov
 - **Hĺbka**: počet riadkov (počet slov)
 - **Šírka**: počet stĺpcov (veľkosť slova)
 - **Veľkosť**: hĺbka \times šírka = $2^N \times M$

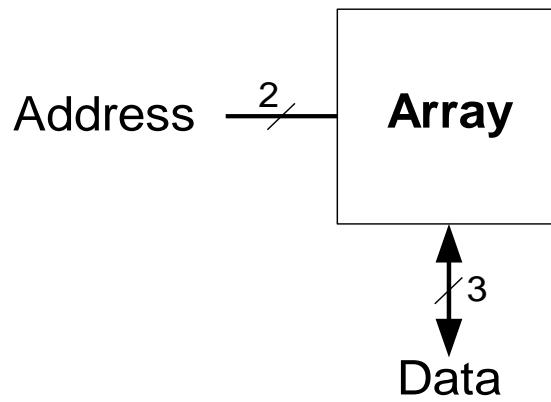


	Address	Data
11	0 1 0	
10	1 0 0	
01	1 1 0	
00	0 1 1	

The table has four rows, each representing a memory location. The first three columns are labeled "Address" and the last column is labeled "Data". The "depth" of the array is indicated by a vertical double-headed arrow on the right side, and the "width" is indicated by a horizontal double-headed arrow at the bottom.

Pamäť

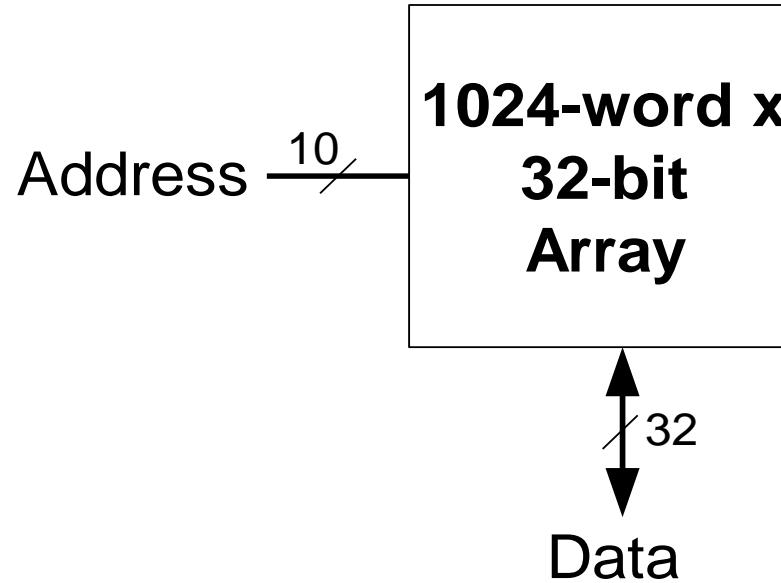
- $2^2 \times 3$ -bitová matica
- Počet slov: 4
- Veľkosť slova: 3b
- Napríklad, 3b slovo uložené na adrese 10 je 100



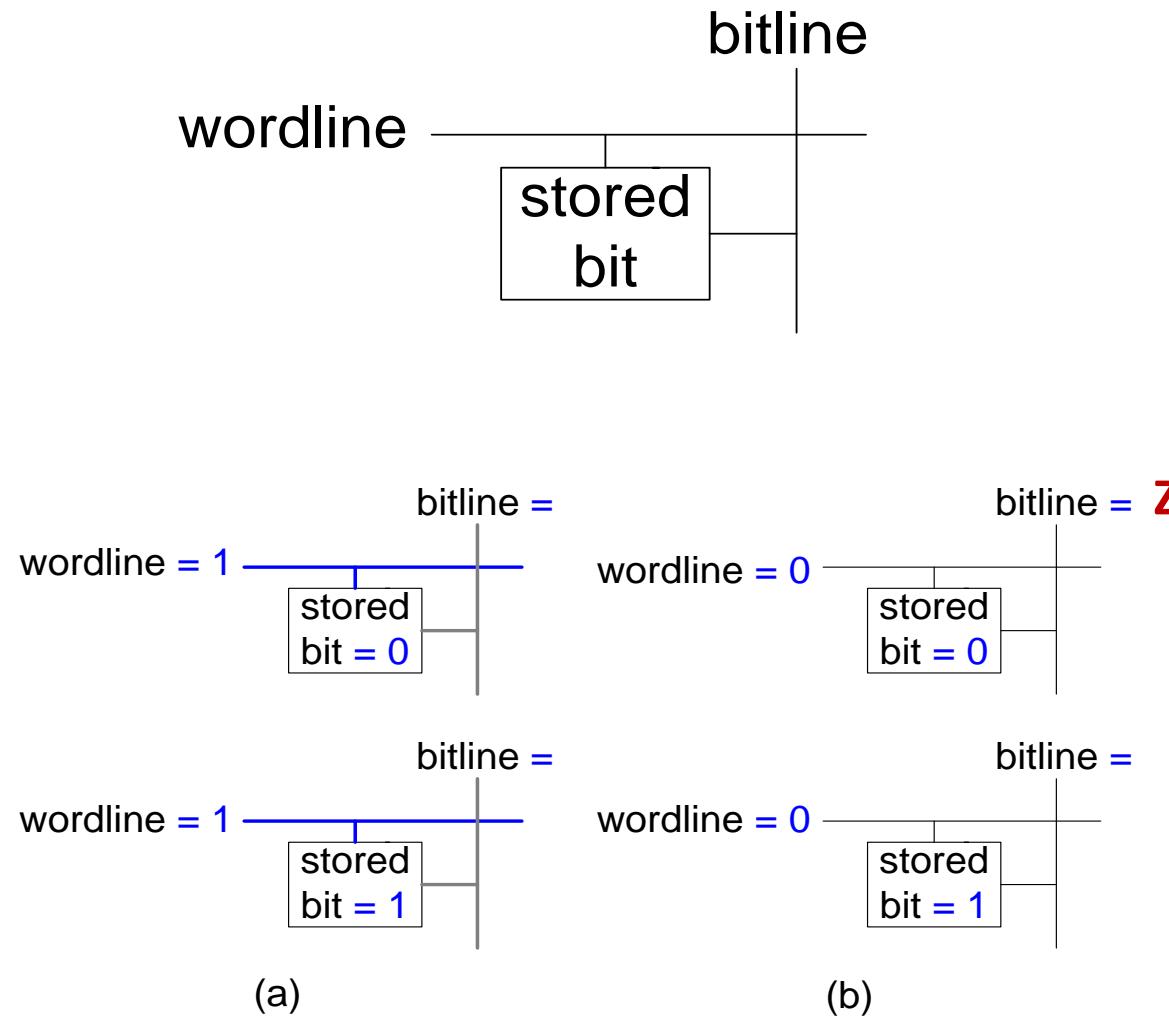
	Address	Data
11		0 1 0
10		1 0 0
01		1 1 0
00		0 1 1

A double-headed vertical arrow on the right side of the table is labeled "depth". A double-headed horizontal arrow at the bottom is labeled "width".

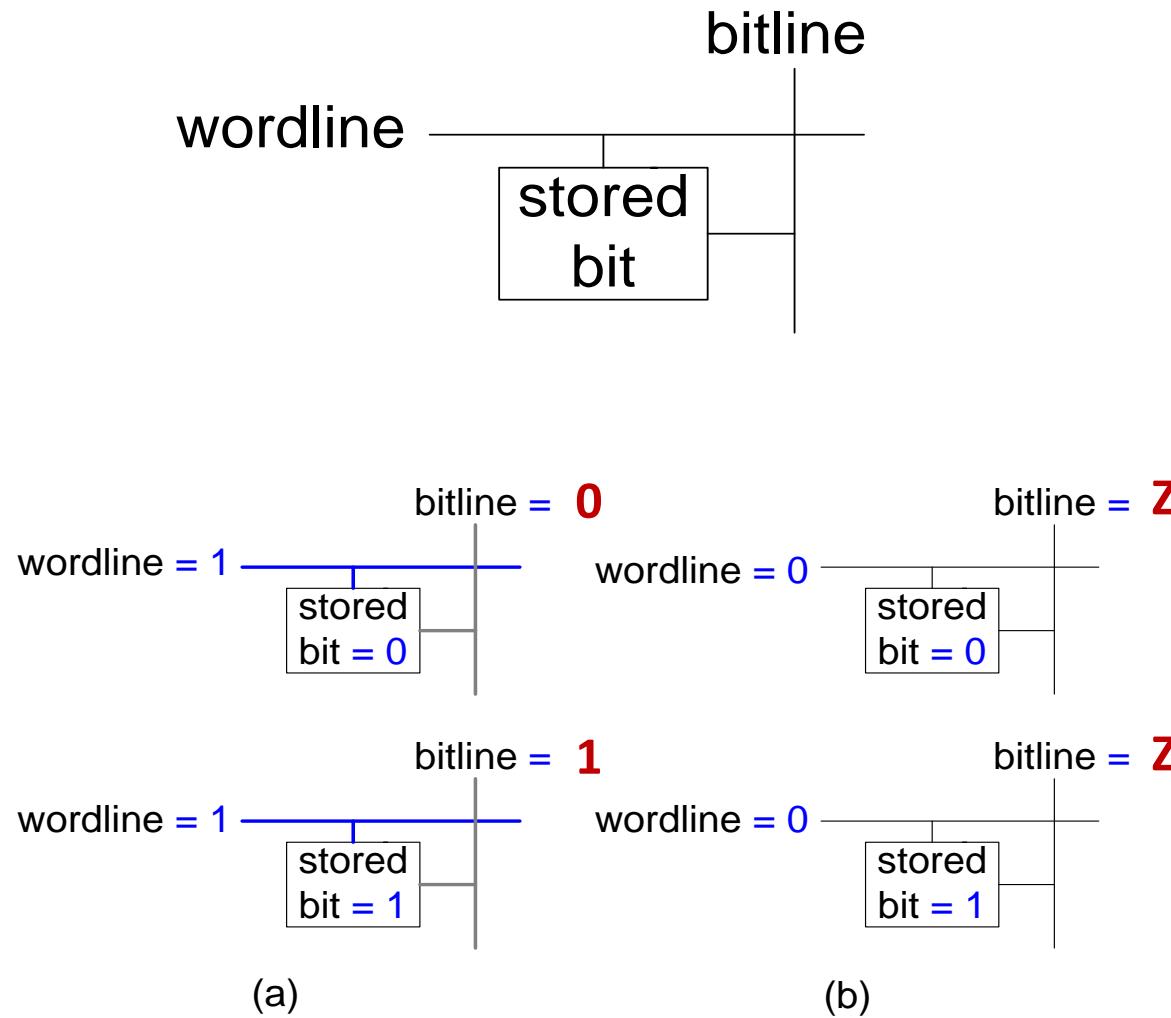
Pamäť



Realizácia pamäti

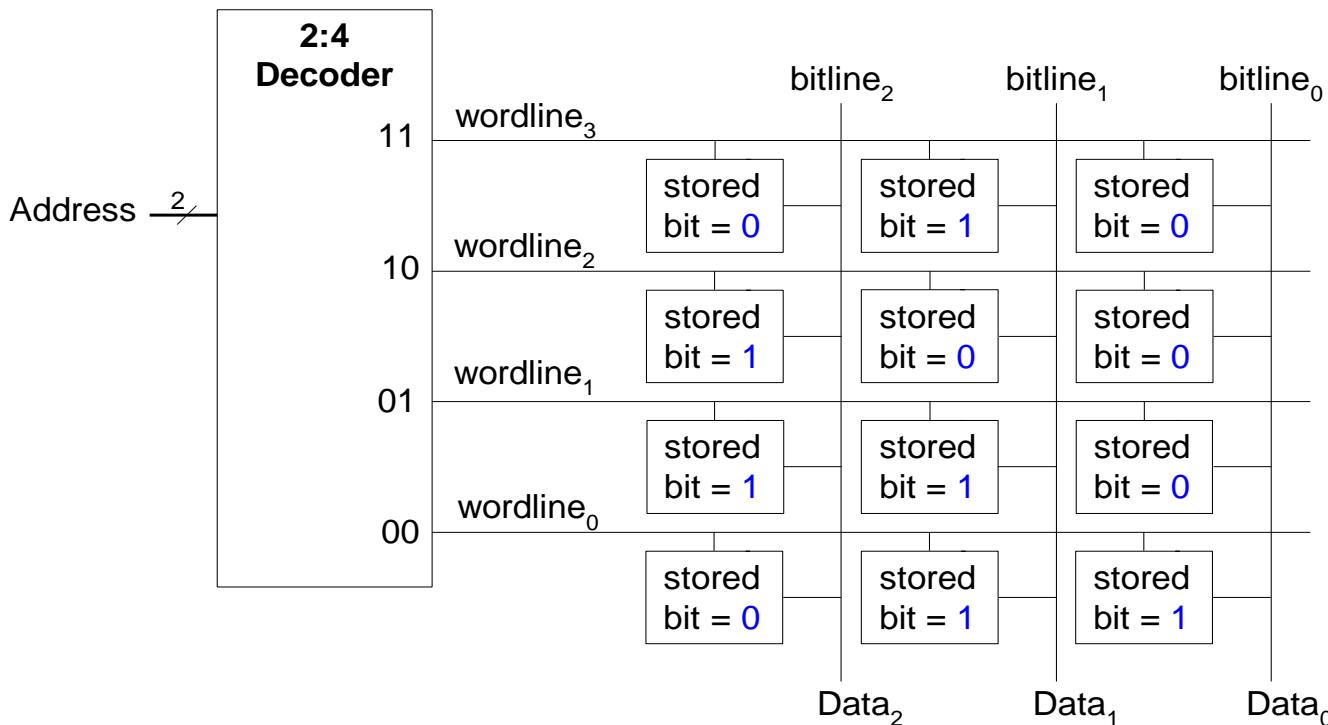


Realizácia pamäti



Pamäť

- Adresná linka (wordline):
 - unikátna adresa slova
 - jeden riadok v pamäti pre čítanie/zápis
 - len jedna linka je v stave HIGH v danom momente



Typy pamäti

- Random access memory (RAM)
 - **volatilná pamäť**
 - energeticky závislá
- Read only memory (ROM)
 - **nevolutilná pamäť**
 - energeticky nezávislá

RAM: s náhodným prístupom

- Stráca obsah po odpojení elektrickej energie
- Čítanie a zápis sú relatívne rýchle operácie
- Hlavná pamäť počítača je typu RAM (DRAM)

Čas zápisu do pamäte je rovnaký bez ohľadu na umiestnenie údaja v pamäti. Jej opakom je pamäť so sekvenčným prístupom.

ROM: určené len na čítanie

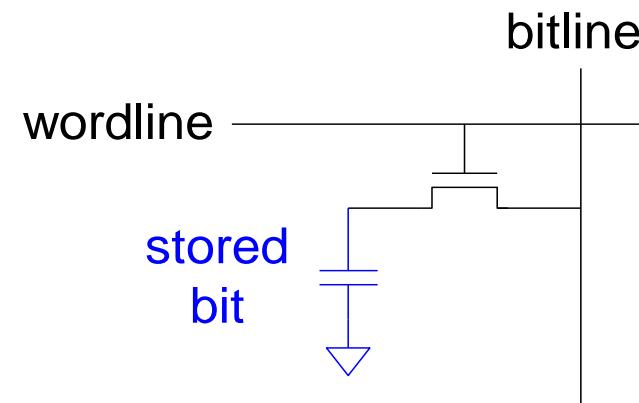
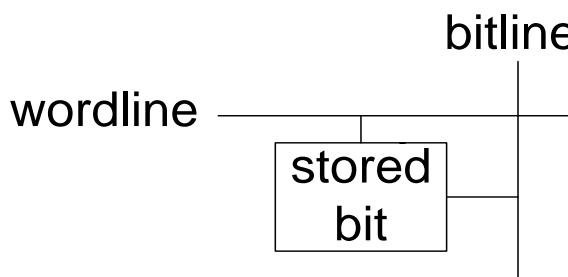
- **NVRAM** = Nestráca obsah po odpojení elektrickej energie
- Rýchle čítanie, zápis je pomalý (alebo nemožný)
- BIOS
- PROM, EPROM, EAROM, EEPROM, Flash pamäť

Pamäte RAM

- **DRAM** (Dynamic random access memory)
- **SRAM** (Static random access memory)
- Rozdiel je v spôsobe ukladania dát:
 - DRAM používa kondenzátory
 - SRAM používa dvojicu invertorov v slučke (cross-coupled inverters)

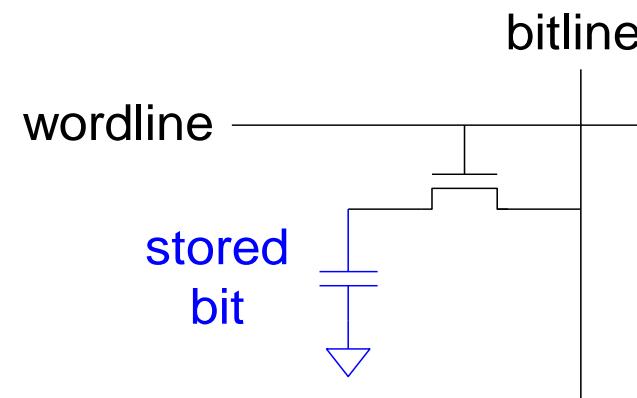
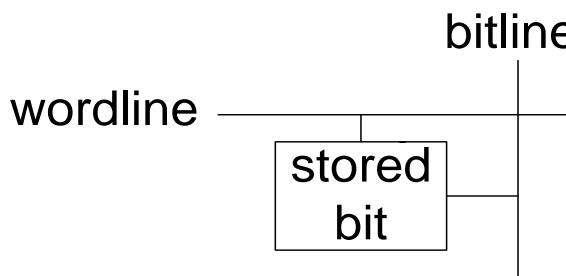
DRAM

- Konštrukčným základom bunky dynamickej pamäte je kondenzátor.
- Kondenzátor je schopný určitý čas udržať elektrický náboj a môže sa teda nachádzať v dvoch základných stavoch:
 - nabitem
 - nenabitem.



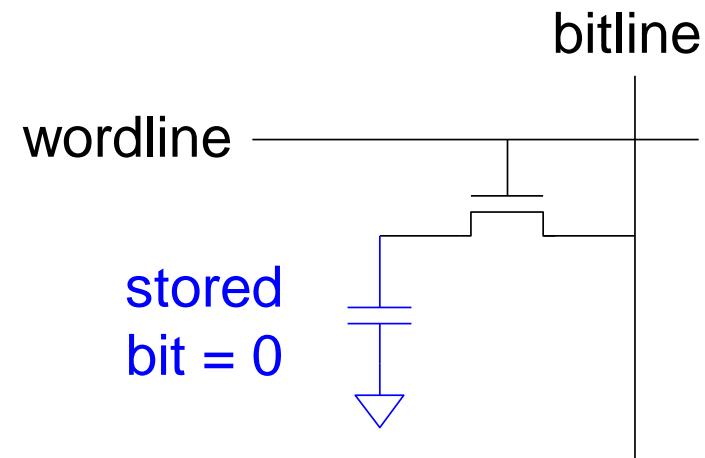
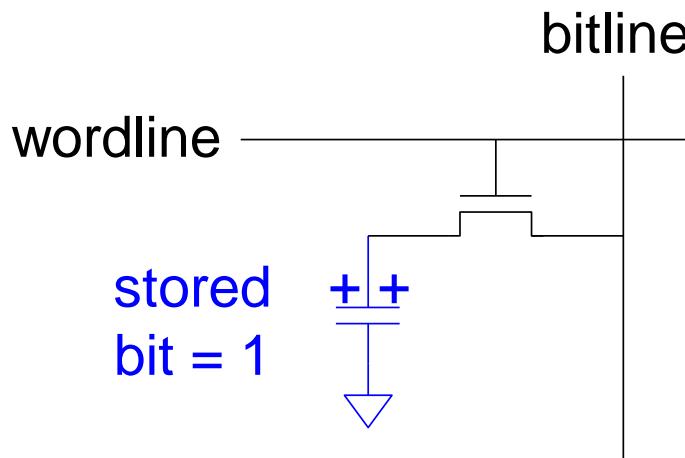
DRAM

- Tieto stavy môžeme využiť na reprezentáciu jedného bitu informácie (napríklad nabitý kondenzátor reprezentuje logickú jednotku a vybitý logickú nulu).
- Kondenzátor po nabití svoj náboj postupne stráca, preto je nutné pravidelne jeho náboj obnovovať – z čoho pochádza aj názov "dynamická pamäť".

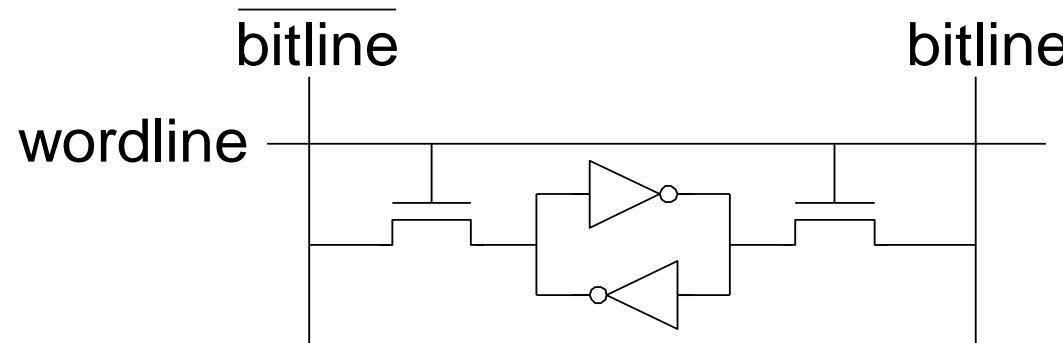
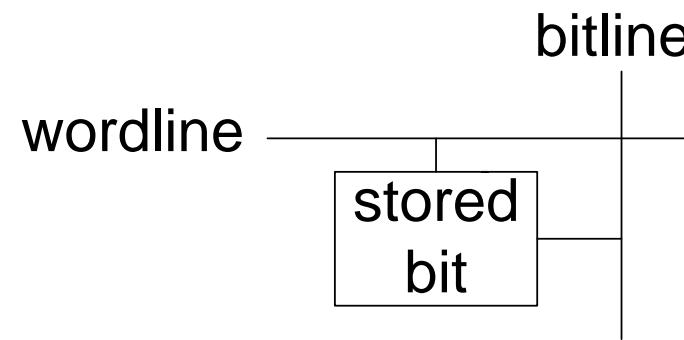


DRAM

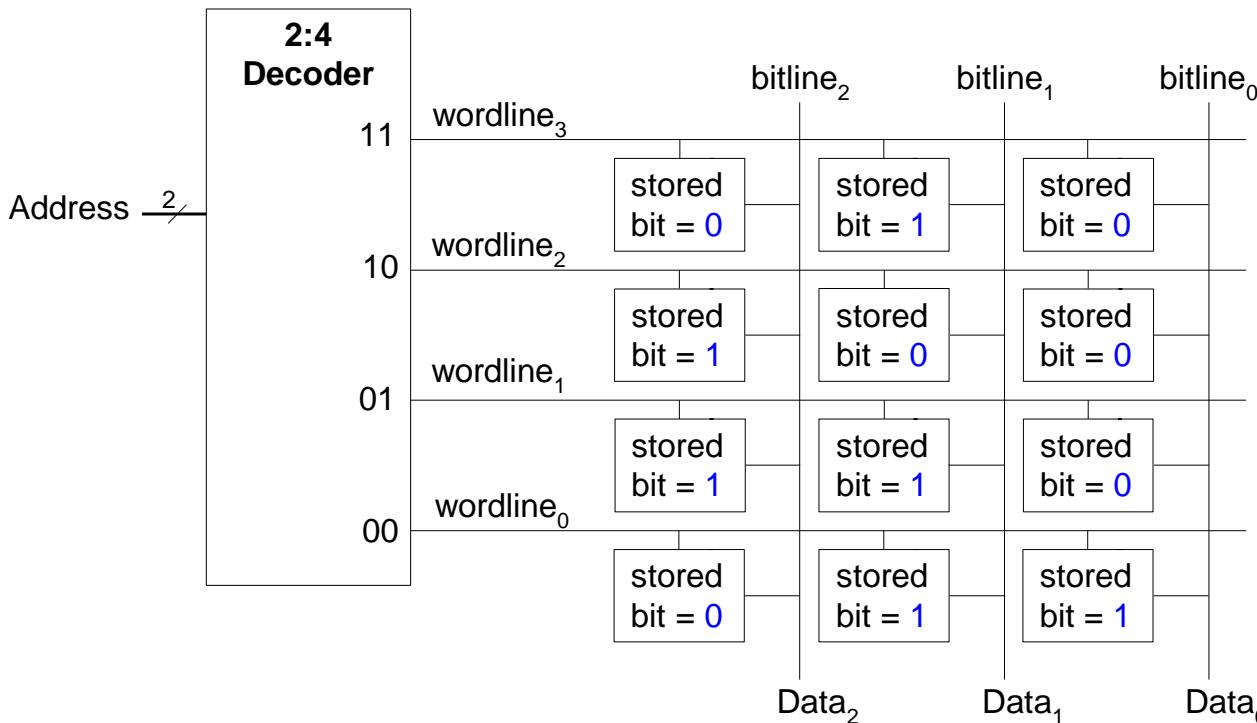
- Po zadaní a rozdelení adresy sa pomocou dekodéra riadku otvoria všetky prístupové tranzistory v adresovanom riadku slova. To spôsobí pripojenie kondenzátorov z pamäťových buniek v tomto riadku na jednotlivé bitové stĺpce.
- V každom bitovom stĺpci je pripojený práve jeden kondenzátor z adresovaného riadku.



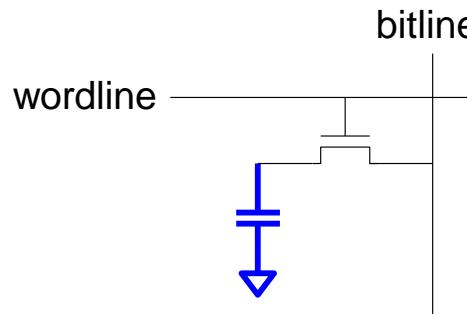
SRAM



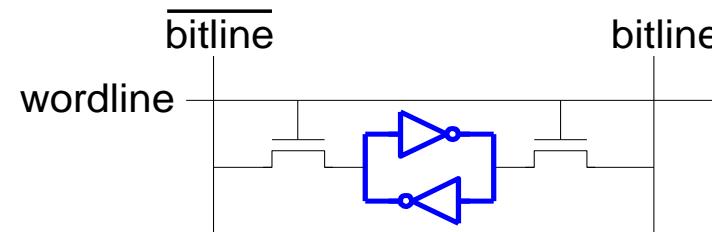
Rekapitulácia



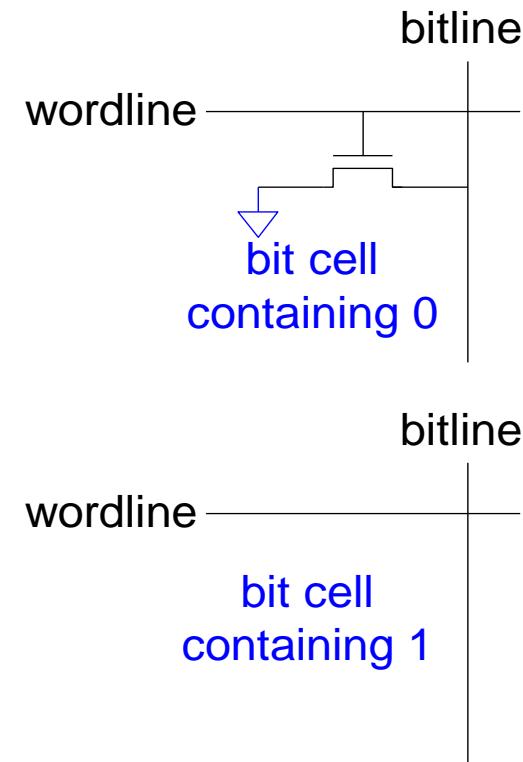
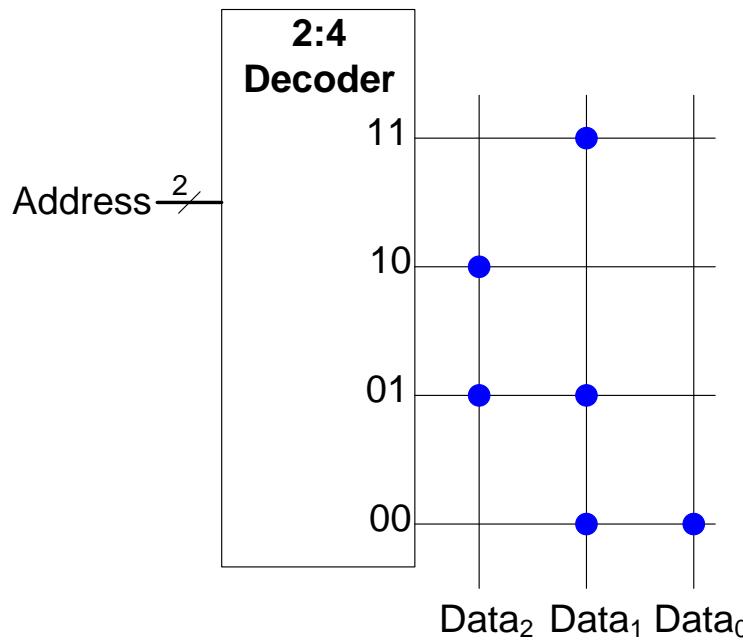
DRAM bunka:



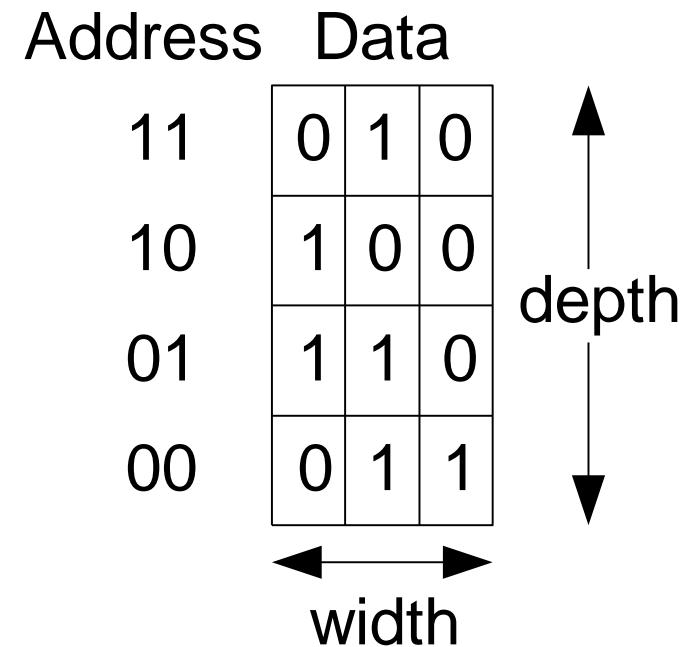
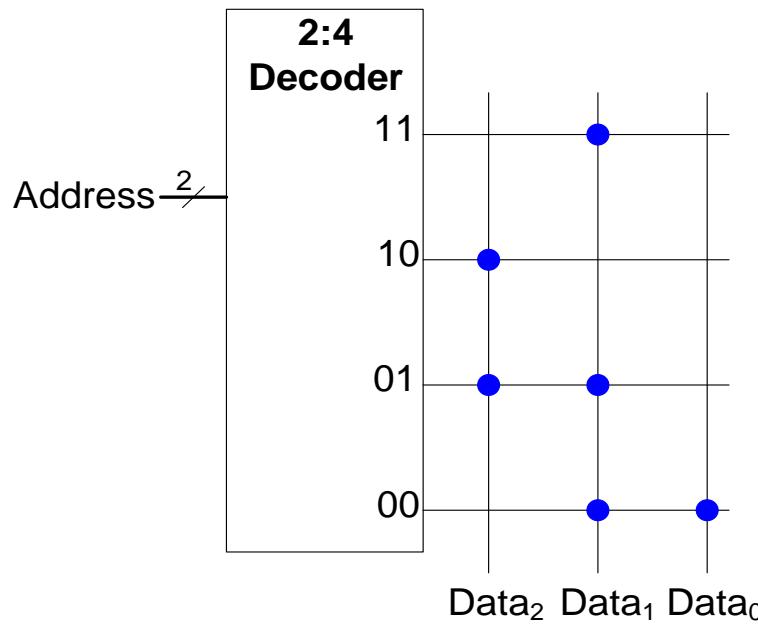
SRAM bunka:



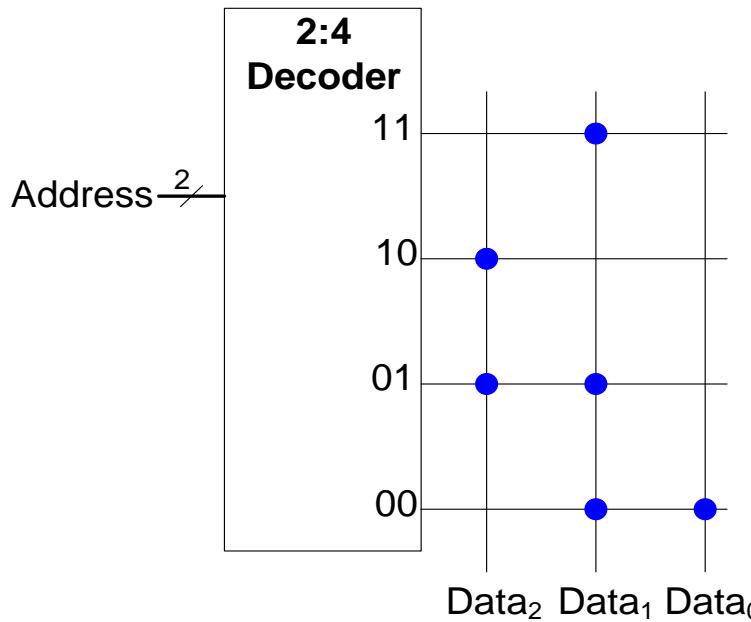
ROM: „dot“ notácia



ROM



Logika pomocou ROM



$$Data_2 = A_1 \oplus A_0$$

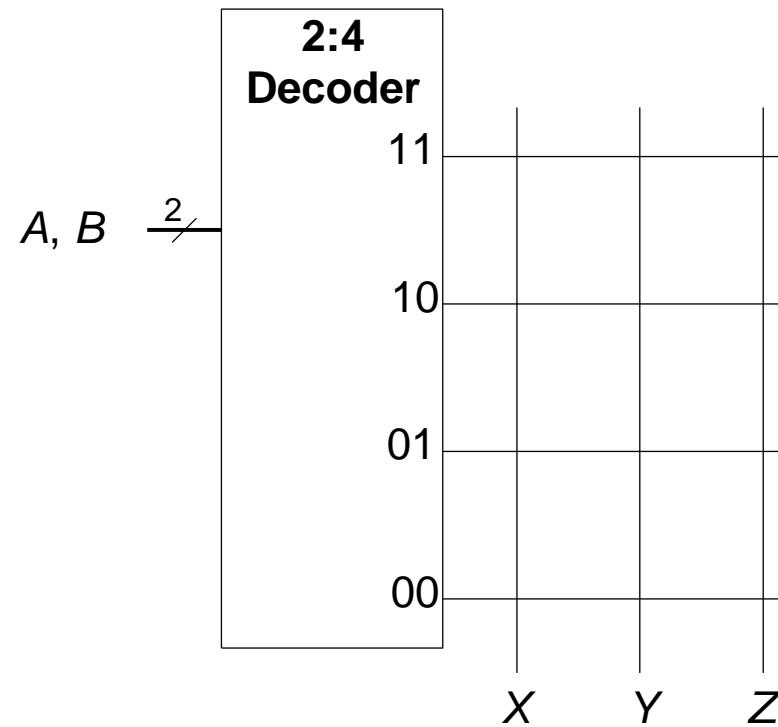
$$Data_1 = \overline{A_1} + A_0$$

$$Data_0 = \overline{A_1} \overline{A_0}$$

Logika pomocou ROM

Implementácia log. funkcií pomocou $2^2 \times 3$ bitovej ROM:

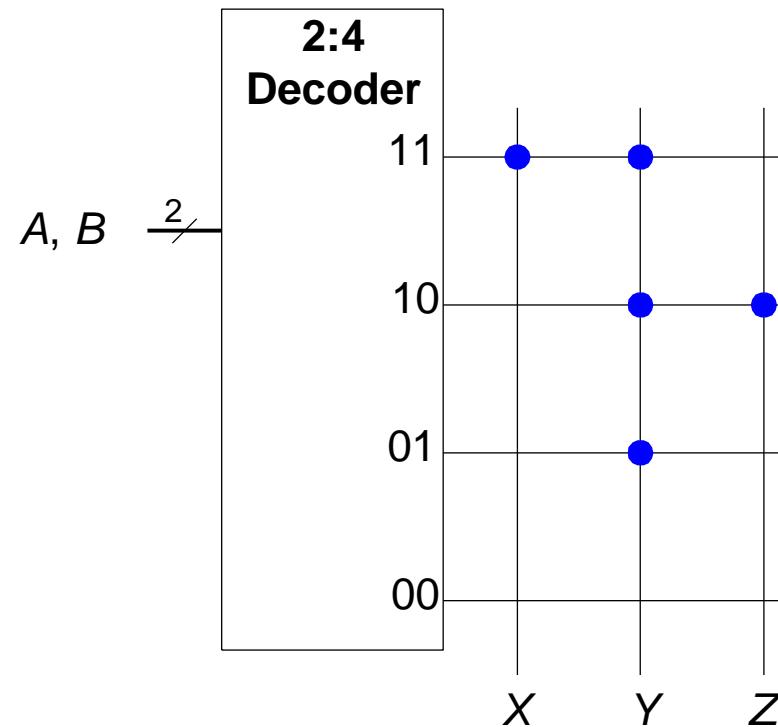
- $X = A B$
- $Y = A + B$
- $Z = A \bar{B}$



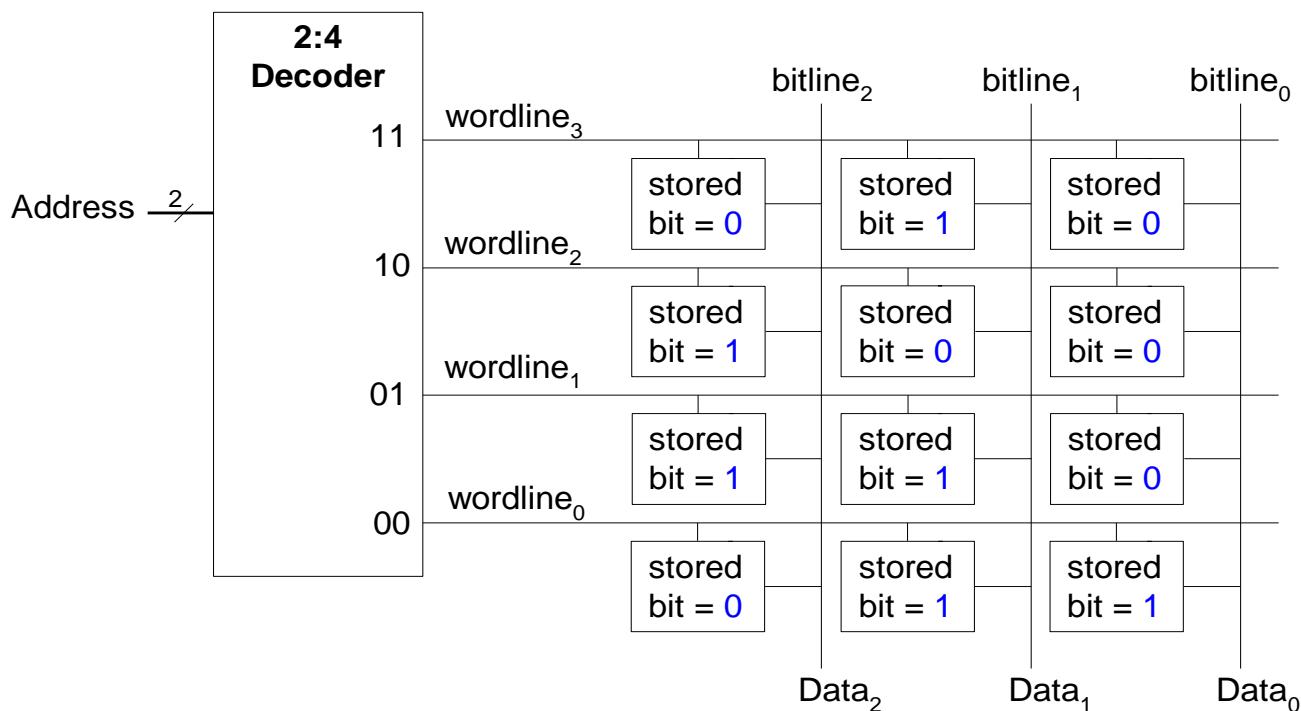
Logika pomocou ROM

Implementácia log. funkcií pomocou $2^2 \times 3$ bitovej ROM:

- $X = A B$
- $Y = A + B$
- $Z = A \bar{B}$



Logika pomocou pamäte



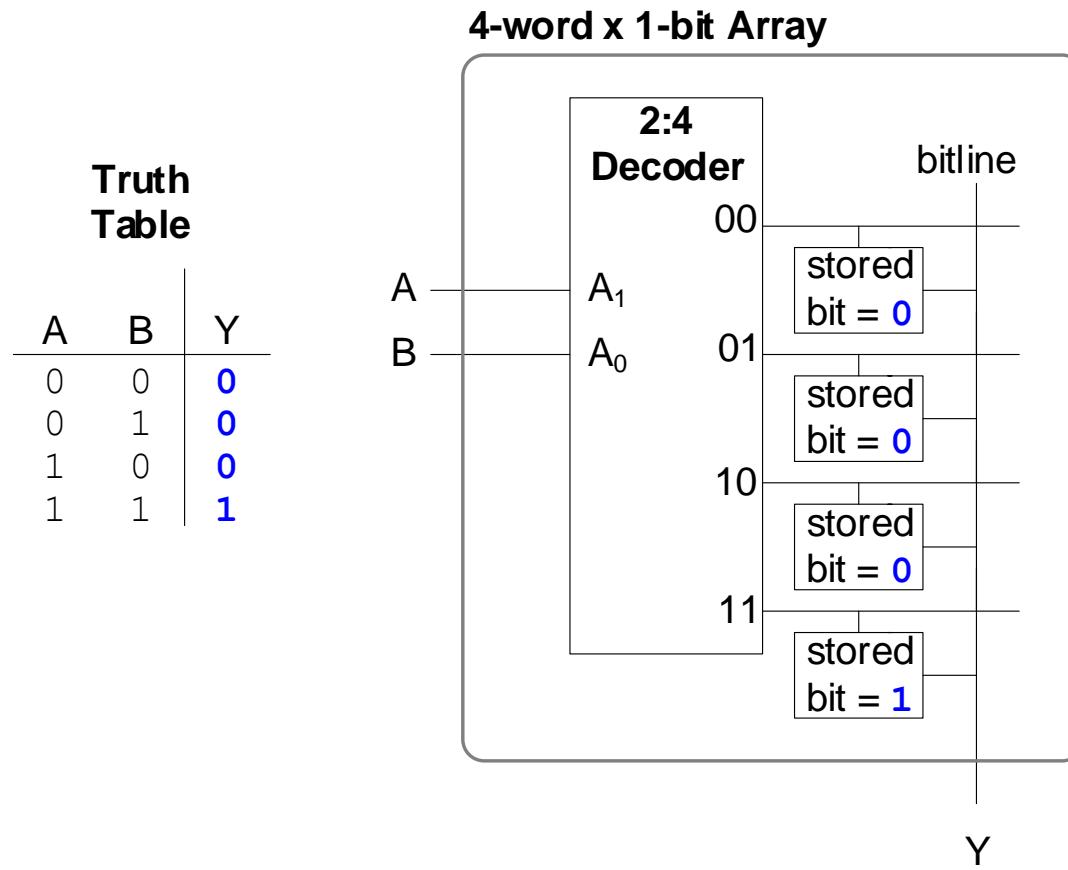
$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = \overline{A}_1 + A_0$$

$$Data_0 = \overline{A}_1 \overline{A}_0$$

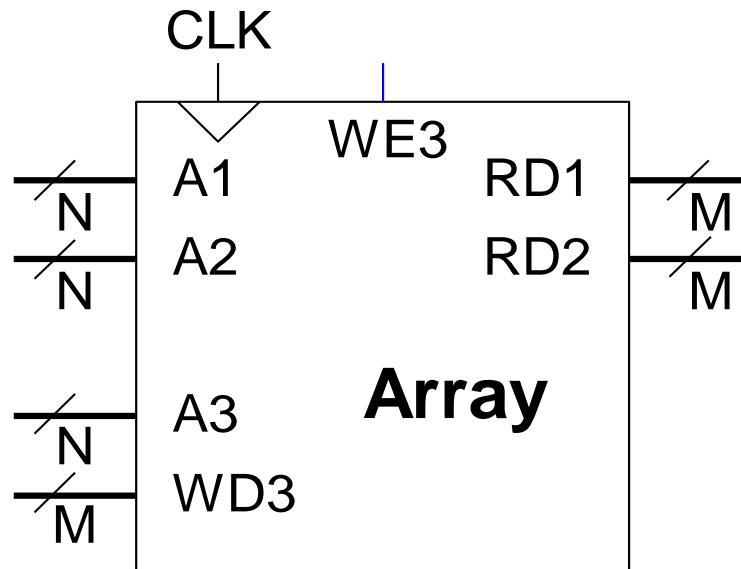
Logika pomocou pamäte

Vyhľadávacie tabuľky (look-up tables; LUTs) implementujú funkciu pravdivostnej tabuľky



Viacportové pamäťe

- **Port:** zabezpečuje vstup/výstup;
- Napr. pamäť s 3 portmi
 - 2 porty pre čítanie (A1/RD1, A2/RD2)
 - 1 port pre zápis (A3/WD3, WE3 povolí zápis)
- **Súbor registrov:** malá viacportová pamäť



SystemVerilog Memory Arrays

```
// 256 x 3 memory module with one read/write port
module dmem( input logic      clk, we,
              input logic[7:0]  a
              input logic [2:0] wd,
              output logic [2:0] rd);

    logic [2:0] RAM[255:0];

    assign rd = RAM[a];

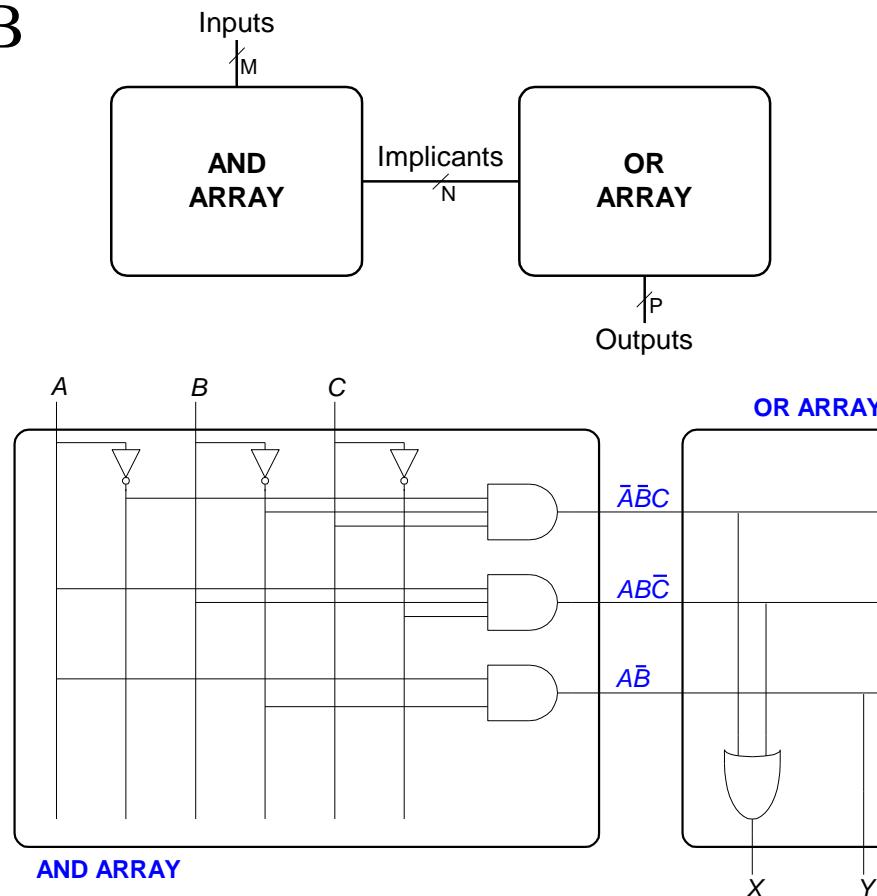
    always @ (posedge clk)
        if (we)
            RAM[a] <= wd;
endmodule
```

Logické polia

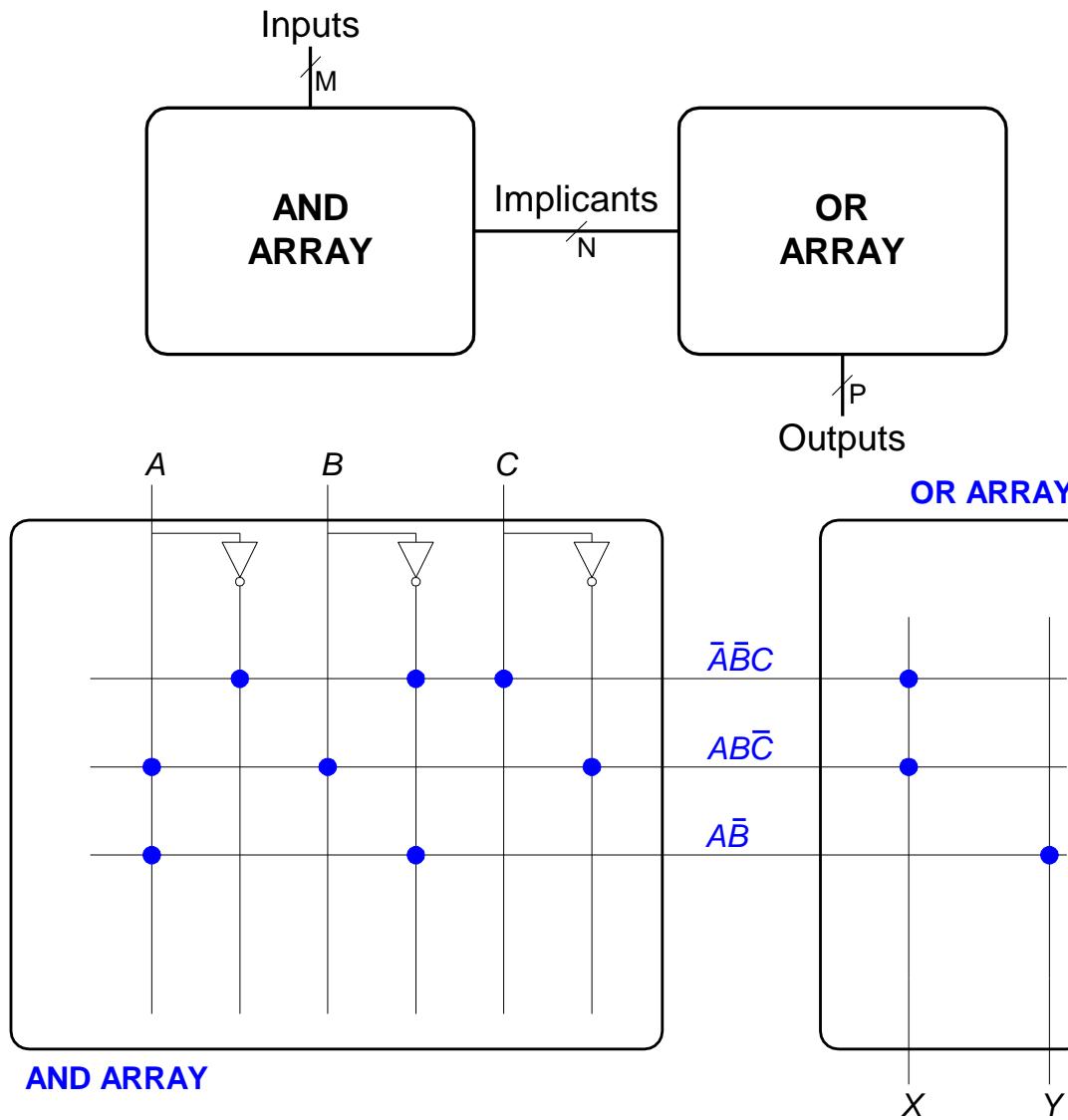
- **PLAs** (Programmable logic arrays)
 - Pole AND členov napojené na pole OR členov
 - Kombinačná logika
- **FPGAs** (Field programmable gate arrays)
 - Pole logických elementov (LEs)
 - Kombinačná aj sekvenčná logika
 - Reprogramovateľné / silná podpora prototypovania

PLA

- $X = \bar{A}\bar{B}C + A\bar{B}\bar{C}$
- $Y = A\bar{B}$



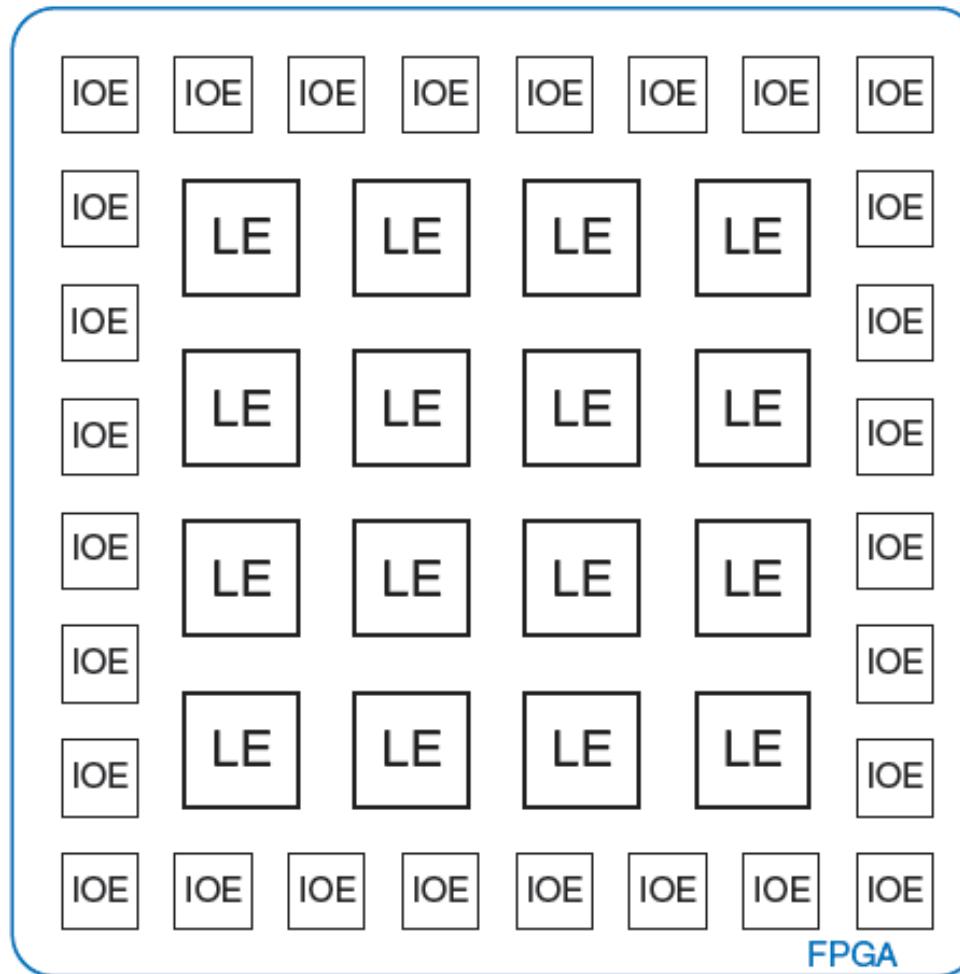
PLA: „dot“ notácia



FPGA: Field Programmable Gate Array

- Pozostáva:
 - **LE** (Logic elements): realizácia logiky
 - **IOE** (Input/output elements): rozhranie pre komunikáciu s vonkajším svetom
 - **Prepojovacia siet'**: spája LE a IOE
 - Iné komponenty
 - Násobičky
 - DSP jednotky
 - pamäte (BRAM, DRAM, ...)

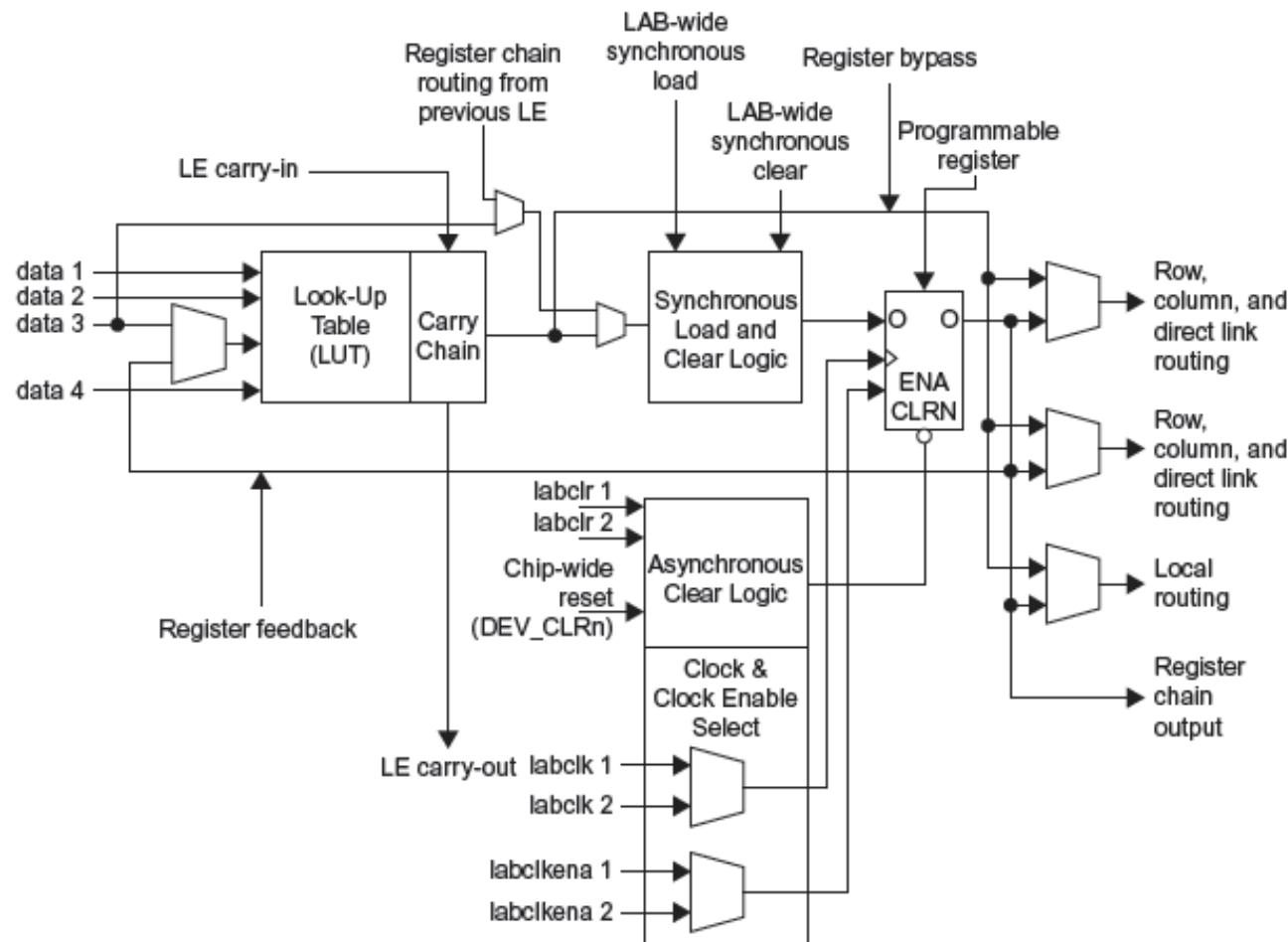
Koncepčná schéma



LE: Logický element

- Pozostáva:
 - **LUT** (lookup tables): realizujú pravdivostnú tabuľku
 - **Preklápacie obvody** (flip-flops): realizujú pamäť pre sekvenčnú logiku
 - **Multiplexory**: spájajú LUT a preklápacie obvody

Altera Cyclone IV LE



Príklad

Prezentujte konfiguráciu Cyclone IV LE na realizáciu funkcií:

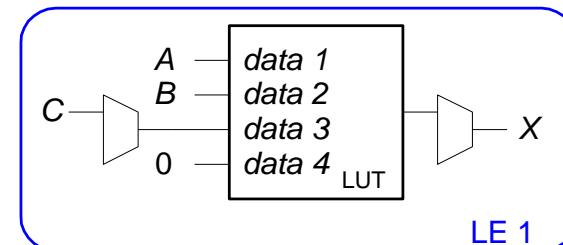
- $X = \bar{A}\bar{B}C + AB\bar{C}$
- $Y = A\bar{B}$

Príklad

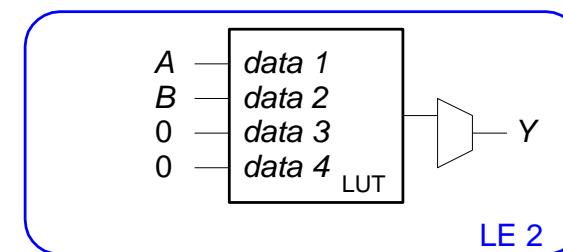
Prezentujte konfiguráciu Cyclone IV LE na realizáciu funkcií:

- $X = \bar{A}\bar{B}C + AB\bar{C}$
- $Y = A\bar{B}$

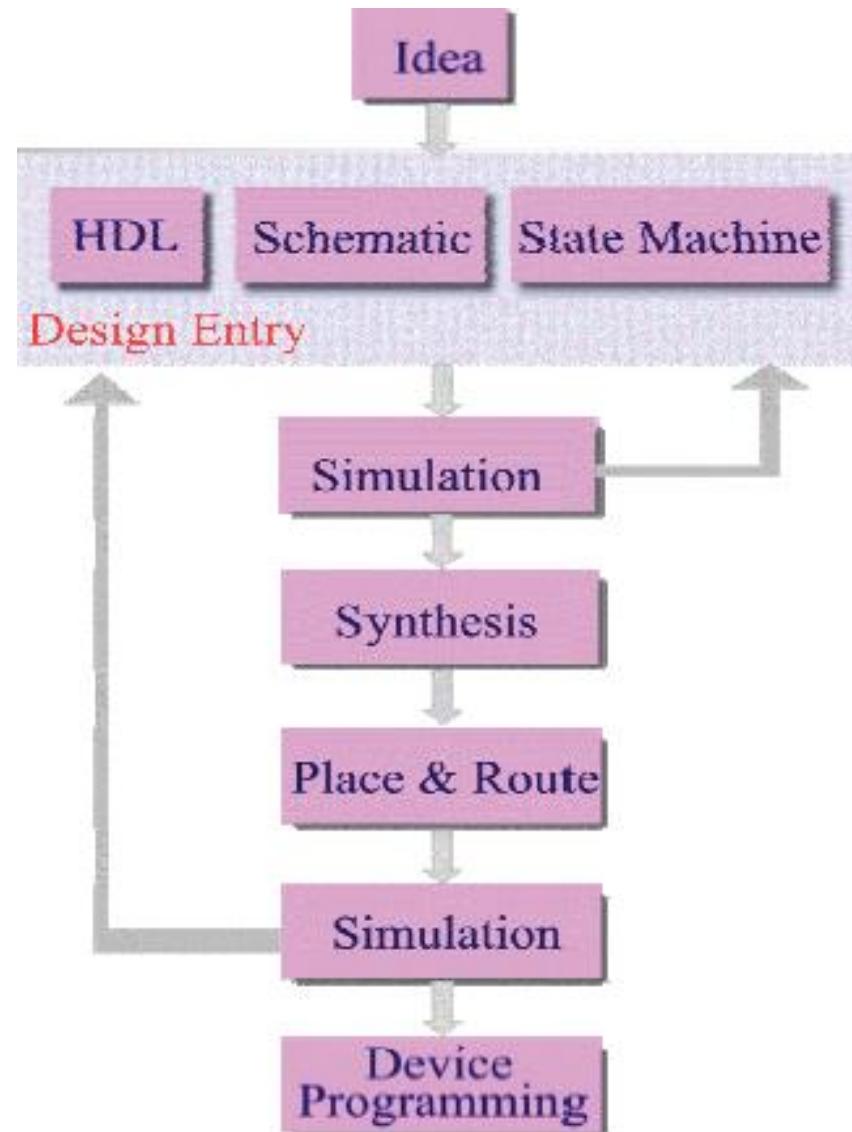
(A) data 1	(B) data 2	(C) data 3	(X) data 4	LUT output
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	0
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	0



(A) data 1	(B) data 2	(C) data 3	(X) data 4	LUT output
0	0	X	X	0
0	1	X	X	0
1	0	X	X	1
1	1	X	X	0



Od návrhu k implementácii



Referencia

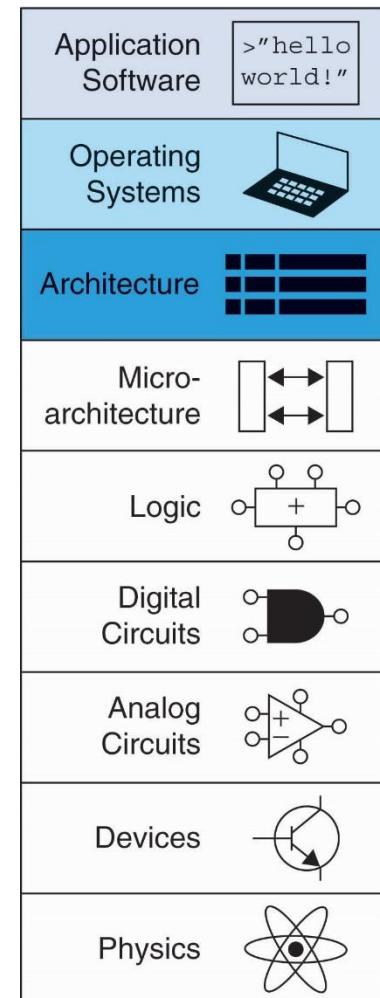
- David Money Harris and Sarah L. Harris,
Digital Design and Computer Architecture,
Chapter 5: Digital Building Blocks, Second
Edition © 2012 by Elsevier Inc.

Architektúry počítačových systémov

2. rok ZS

Osnova

- Asembler
- Strojový kód
- Programovanie
- Adresovacie módy
- Preklad, spájanie, zavedenie



Úvod

- **ISA:** inštrukčno-orientovaná architektúra
 - abstrakcia prostredníctvom súboru inštrukcií
 - súbor inštrukcií & dát & miesto uloženia & adresovacie módy
- **IA:** implementačno-orientovaná architektúra
 - abstrakcia na úrovni štruktúrnej organizácie

Application Software	programs
Operating Systems	device drivers
Architecture	instructions registers
Micro-architecture	datapaths controllers
Logic	adders memories
Digital Circuits	AND gates NOT gates
Analog Circuits	amplifiers filters
Devices	transistors diodes
Physics	electrons

Asembler

- **Inštrukcie:** „príkazy napísané v jazyku počítača“
 - **Asembler:** jazyk symbolických inštrukcií / je pre ľudí na čítanie vhodná forma strojového kódu
 - **Strojový jazyk/kód:** inštrukcie sú v podobe binárneho reťazca / je vhodná pre spracovanie v digitálnych zariadeniach
- **MIPS architektúra:**
 - Vývoj 80-tych rokoch na Stanfordskej univerzite pod vedením Johna Hennessyho.
 - Použitá v zariadeniach od Silicon Graphics, Nintendo, a Cisco

Princípy návrhu počítačových architektúr

Hennessy a Patterson navrhujú postupovať pri návrhu architektúry tak, aby boli dodržané nasledujúce odporúčania:

- 1. Jednoduchosť podporuje jednotnosť**
- 2. Jadro musí byť rýchle**
- 3. Zníženie počtu komponentov zvyšuje rýchlosť („Čo je malé to je rýchle“)**
- 4. Dobrý návrh vychádza z kompromisov**

Inštrukcia: sčítavanie

C kód

$a = b + c;$

MIPS kód

add a, b, c

- **add:** mnemonika (binárny kód nahradený skráteným názvom) definujúca operáciu
- **b, c:** zdrojové operandy (nad ktorými sa operácia zrealizuje)
- **a:** cieľový operand (do ktorého sa zapíše operand)

Inštrukcia: odčítavanie

- Obdoba sčítavania – len iná mnemonika

C kód

$a = b - c;$

MIPS kód

sub a, b, c

- **sub:** mnemonika
- **b, c:** zdrojové operandy
- **a:** cieľový operand

Odporúčanie č. 1

Jednoduchosť podporuje jednotnosť

- Konzistentný inštrukčný formát
- Rovnaký počet operandov (dva zdrojové a jeden cielový)
- Jednoduché dekódovanie

Zložitejšie inštrukcie

- Zložitejšie príkazy sa preložia do postupnosti MIPS inštrukcií.

C kód

$a = b + c - d;$

MIPS kód

add t, b, c # $t = b + c$
sub a, t, d # $a = t - d$

Odporúčanie č. 2

Jadro musí byť rýchle

- MIPS definuje len jednoduché, často používané inštrukcie a preto hardvér určený na dekódovanie a vykonanie inštrukcií je jednoduchý, malý a rýchly
- Menej časté a komplexnejšie príkazy sú realizované ako postupnosť jednoduchých inštrukcií
- MIPS je predstaviteľom ***reduced instruction set (RISC) architektúr***
- Intel x86 je predstaviteľom ***complex instruction set (CISC) architektúr***

Operandy

- Ich hodnoty sú ukladané v
 - registroch,
 - pamäti(-ach) alebo
 - sú definované ako konštanty (zvané aj ako priame operandy) a vtedy sú súčasťou inštrukčného formátu

Operandy v registroch

- MIPS má 32 32-bitových registrov
- Registre sú rýchlejšie ako pamäťové moduly
- MIPS je “32-bitová architektúra” lebo realizuje operácie nad 32-bitovými slovami

Odporúčanie č. 3

Čo je malé to je rýchle

- MIPS preferuje operácie nad operandmi uloženými v registroch

Operandy v registroch

- Hodnoty operandov v registroch:
 - Názov registra začína symbolom \$
 - Príklad: \$0, “register zero”, “dollar zero”
 - \$0 reprezentuje 0.
 - \$s0-\$s7 sa používajú na uchovávanie hodnôt premenných, ktoré vystupujú vo výpočtoch
 - \$t0 - \$t9 sa používajú na uchovávanie medzivýsledkov, ktoré vznikajú pri zložitejších výpočtoch
 - Iné

MIPS súbor registrov

Označenie	Číselný kód	Význam
\$0	0	Konštanta 0
\$at	1	Rezervované pre pseudoinstrukcie
\$v0-\$v1	2-3	Návratová hodnota z funkcií
\$a0-\$a3	4-7	Argumenty funkcií
\$t0-\$t7	8-15	Dočasné premenné
\$s0-\$s7	16-23	Premenné
\$t8-\$t9	24-25	Dočasné
\$k0-\$k1	26-27	OS špecifické
\$gp	28	Bázová adresa globálneho dátového segmentu
\$sp	29	Smerník zásobníka
\$fp	30	Smerník rámca
\$ra	31	Návratová adresa z funkcie

Register-Register inštrukcie

C kód

$a = b + c$

MIPS kód

\$s0 = a, \$s1 = b, \$s2 = c
add \$s0, \$s1, \$s2

Operandy v pamäti

- 32 registrov nestačí pre spracovanie veľkého množstva dát
- Pamäť má väčšiu kapacitu ale v porovnaní s regisrami je pomalá
- Často „odkazované“ operandy sú uložené v registroch

Adresovanie

- Každé 32b slovo má svoju unikátnu adresu

Word Address	Data	
...
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Čítanie z pamäte

- Inštrukcia *load*
- Mnemonika: *load word (lw)*
- Formát:
lw \$s0, 5(\$t1)
- Výpočet adresy:
 - K bázovej adrese ($\$t1$) sa pripočíta posuv zvaný aj *offset* (5)
 - Adresa = $(\$t1 + 5)$
- Výsledok:
 - $\$s0$ obsahuje hodnotu uloženú v bunke $(\$t1 + 5)$

Čítanie z pamäte

- **Príklad.** Ulož slovo z p. bunky s adresou 1 do \$s3
 - Adresa = $(\$0 + 1) = 1$
 - \$s3 = 0xF2F1AC07 po čítaní

Asembler

`lw $s3, 1($0) # ulož hodnotu slova z adresy 1 do $s3`

Word Address	Data	
...
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Zápis do pamäte

- Inštrukcia *store*
- Mnemonika: *store word (sw)*

Zápis do pamäte

- **Príklad:** Ulož hodnotu z \$t4 do p. bunky s adresou 7
 - Adresa: $(\$0 + 0x7) = 7$

Posuv môže byť zapísaný aj v hexadecimálnom tvare

Asembler

```
sw $t4, 0x7($0) # ulož hodnotu z $t4  
# do p.bunky s adresou 7
```

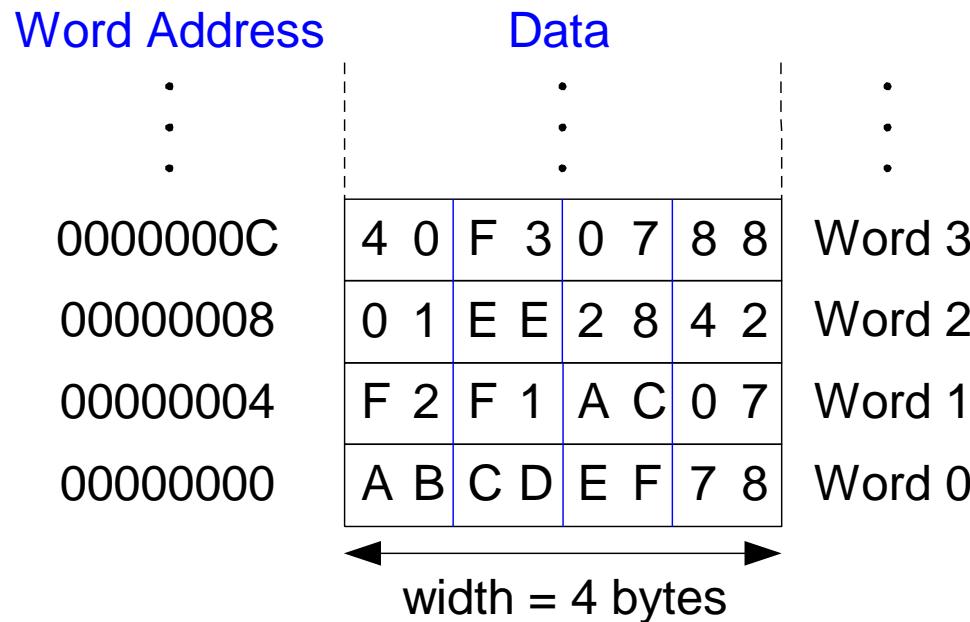
Word Address	Data	
:	:	:
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

- Hoci sa spracovanie informácií v počítači uskutočňuje prostredníctvom slovnej organizácie pamäte (v súčasnosti sú najčastejšie používané 32-bitové a 64-bitové slová), fyzicky sú tieto rozdelené do bajtov.
- Za účelom sprístupňovania viacbajtových informačných slov s premenlivou dĺžkou sa stáva prirodzenou požiadavka, aby bajty boli individuálne adresované.

- Zarovnávanie viacbajtových objektov
 - Usporiadanie bajtov objektov v informačných slovách
- Ukladací endian
 - Malý endian
 - Veľký endian

Adresovanie slov po bajtoch

- Každý bajt v slove má svoju unikátnu adresu
- Čítanie a zápis po bajtoch: *load byte (lb)* a *store byte (sb)*
- 32b slovo = 4 bajty / adresa nasledujúceho slova +4



Adresovanie slov po bajtoch

- Adresa slova v pamäti je násobkom 4.
Napríklad,
 - adresa 2. slova v pamäti je $2 \times 4 = 8$
 - adresa 10. slova je $10 \times 4 = 40$ (0x28)
- **MIPS používa bajtovú organizáciu pamäte pri ukladaní 32b slov**

Čítanie slova z pamäti

- **Príklad:** Čítaj 1. slovo a ulož do \$s3.
- \$s3 = 0xF2F1AC07 po čítaní

MIPS kód

lw \$s3, 4(\$0)

Word Address	Data								
...
0000000C	4	0	F	3	0	7	8	8	Word 3
00000008	0	1	E	E	2	8	4	2	Word 2
00000004	F	2	F	1	A	C	0	7	Word 1
00000000	A	B	C	D	E	F	7	8	Word 0

width = 4 bytes

Zápis slova do pamäti

- **Príklad:** Ulož obsah \$t7 do pamäti na adresu 0x2C (44)

MIPS kód

sw \$t7, 44(\$0)

Word Address	Data								
...
0000000C	4	0	F	3	0	7	8	8	Word 3
00000008	0	1	E	E	2	8	4	2	Word 2
00000004	F	2	F	1	A	C	0	7	Word 1
00000000	A	B	C	D	E	F	7	8	Word 0

width = 4 bytes

Ukladací endian

- Ako číslovať bajty objektov/slov?
- **Malý endian:** v smere od najnevýznamnejších (LSB) po najvýznamnejšie bajty (MSB)
- **Veľký endian:** v smere od jeho najvýznamnejších bajtov po najnevýznamnejšie bajty
- Nemá to vplyv na adresovanie slova

Big-Endian

Byte Address	Word Address
:	C
C D E F	8
8 9 A B	4
4 5 6 7	0
0 1 2 3	

MSB LSB

Little-Endian

Byte Address	Word Address
:	F
F E D C	B
B A 9 8	7
7 6 5 4	3
3 2 1 0	

MSB LSB

Ukladací endian

Malý endián				Adresa	Adresa	Veľký endián			
3	2	1	0	0		0	0	1	2
7	6	5	4	4		4	4	5	6
11	10	11	10	8		8	8	9	10
15	14	13	12	12		12	12	13	14

↔ bajt (8b)

↔ slovo (32b)

↔ bajt (8b)

↔ slovo (32b)

Ukladací endian

Veľký endián

0	J	I	M	
4	S	M	I	T
8	H	0	0	0
12	0	0	0	21
16	0	0	1	4

a)

Malý endián

	M	I	J
T	I	M	S
0	0	0	H
0	0	0	21
0	0	1	4

b)

Prenos

0		M	I	J
4	T	I	M	S
8	0	0	0	H
12	21	0	0	0
16	4	1	0	0

c)

Prenos a výmena

J	I	M	
S	M	I	T
H	0	0	0
0	0	0	21
0	0	1	4

d)

Odporúčanie č. 4

Dobrý návrh vychádza z kompromisov

- Rôzne inštrukčné formáty majú svoju výhodu
 - **add, sub:** 3 registre / operandy
 - **lw, sw:** 2 registre/ operandy + konštanta
- Počtom inštrukčných formátov netreba „prehnáť“
 - Aby to bolo v súlade s odporúčaniami 1 až 3.

Operandy: konštanta / priamy

- **lw** a **sw** pracujú s *priamym* operandom (konštantou)
- Operand je súčasťou IF *priamo*
- Používa sa 16-bitový DK
- Príklad:
 - **addi**: add immediate
 - **subi**: existuje?

C kód

```
a = a + 4;  
b = a - 12;
```

MIPS kód

```
# $s0 = a, $s1 = b  
addi $s0, $s0, 4  
addi $s1, $s0, -12
```

Strojový jazyk / kód

- Binárna reprezentácia inštrukcie
- 32b inštrukcie
 - Jednoduchosť podporuje jednotnosť:
32-bit dáta & inštrukcie
- 3 inštrukčné formáty (IF):
 - **Typ R:** operandy sú v registroch
(napr. operačná inštrukcia)
 - **Typ I:** používa priamy operand
(napr. pamäťová inštrukcia)
 - **Typ J:** vetvenia

Inštrukcia typu R

- Typu *R*(egister)
- 3 operandy:
 - **rs, rt:** zdrojové registre
 - **rd:** cieľový register
- Ďalšie položky:
 - **op:** kód operácie (0 pre inštrukciu typu R)
 - **funct:** pole na nastavenie rôznych typov inštrukcií
 - **shamt:** pre posuvné operácie definuje počet posuvov, inak 0

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Príklad

Assembly Code

```
add $s0, $s1, $s2  
sub $t0, $t3, $t5
```

Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

Poznámka. Poradie registrov v MIPS IF je

add rd, rs, rt

Inštrukcia typu I

- Typu I(mmediate)
- 3 operandy:
 - rs, rt: operandy v registroch
 - imm: 16-bit konštanta v DK
- Ďalšie položky:
 - op: kód operácie (každá inštrukcia má vlastný kód)
- Operácia je definovaná kódom operácie

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Príklad

Assembly Code

```
addi $s0, $s1, 5  
addi $t0, $s3, -12  
lw   $t2, 32($0)  
sw   $s1, 4($t1)
```

Field Values

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4

6 bits 5 bits 5 bits 16 bits

Machine Code

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)

6 bits 5 bits 5 bits 16 bits

Poznámka. Pozor na poradie operandov v asembléri:

addi rt, rs, imm

lw rt, imm(rs)

sw rt, imm(rs)

Inštrukcia typu J

- Typu J(ump)
- 26b pre adresu skoku (addr)

J-Type



Sumarizácia

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

J-Type

op	addr
6 bits	26 bits

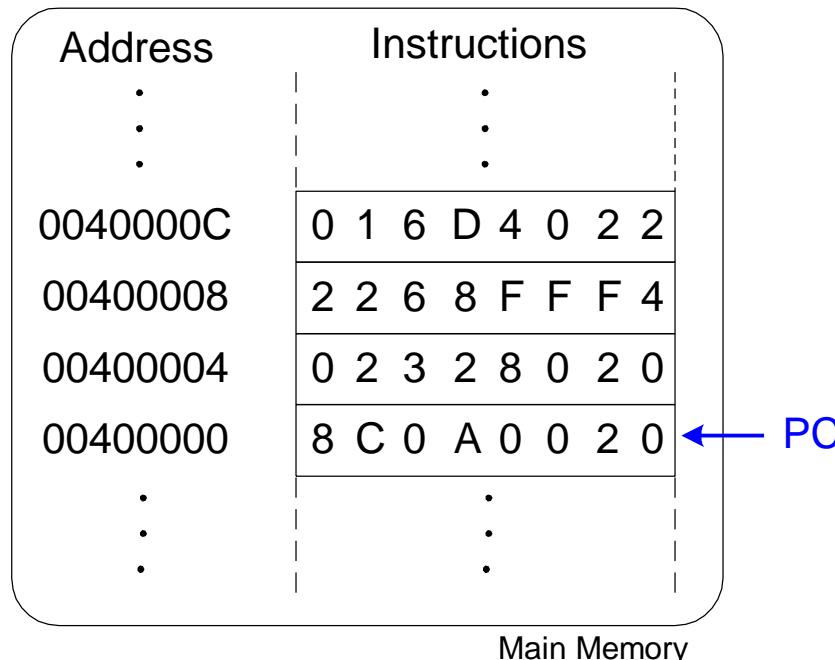
Program v pamäti

- Inštrukcia a aj dátá sú v pamäti
- Rozdielna sekvencia inštrukcií = rozdielny program (aj keď výsledok môže byť rovnaký)
- Nový výpočtový problém:
 - Nie je potrebný iný HW stačí len zmeniť program
- Spracovanie programu:
 - Procesor číta inštrukcie z pamäte počítača
 - Procesor vykonáva inštrukcie

Program v pamäti

	Assembly Code	Machine Code
1w	\$t2, 32(\$0)	0x8C0A0020
add	\$s0, \$s1, \$s2	0x02328020
addi	\$t0, \$s3, -12	0x2268FFF4
sub	\$t0, \$t3, \$t5	0x016D4022

Stored Program



Programové počítadlo (PC):
definuje poradie spracovania inštrukcií

Spracovanie IF

- Začína sa s ***op***
 - ovplyvňuje dekódovanie
- Napríklad ak ***op*** = 0
 - Je to inštrukcia typu R
 - Bity z poľa ***funct*** určujú operáciu

	Machine Code				Field Values				Assembly Code				
(0x2237FFF1)	op 001000	rs 10001	rt 10111	imm 1111111111110001	op 8	rs 17	rt 23	imm -15	addi \$s7, \$s1, -15				
	2 2	2 3	3 7	F F	F F	F F	1						
(0x02F34022)	op 000000	rs 10111	rt 10011	rd 01000	shamt 00000	funct 100010	op 0	rs 23	rt 19	rd 8	shamt 0	funct 34	sub \$t0, \$s7, \$s3
	0 2	2 F	F 3	4 4	0 2	2 2							

Programovanie

- Vyššie programovacie jazyky :
 - Napr. C, Java, Python
 - Vyššia forma abstrakcie
- Jazykové konštrukcie vyšších programovacích jazykov:
 - Príkazy if/else
 - Aritmetické cykly for
 - Logické cykly while
 - Kompozitné dátové štruktúry (napr. polia)
 - Funkcie/procedúry

Logické inštrukcie

- **and, or, xor, nor**
 - and: používa sa pri **maskovaní** bitov
 - Príklad: maskovanie okrem posledného bajtu
$$0xF234012F \text{ AND } 0x000000FF = 0x0000002F$$
 - or: používa sa pri **rekombinácií** bitov
 - Rekombinuj slovo 0xF2340000 s 0x000012BC:
$$0xF2340000 \text{ OR } 0x000012BC = 0xF23412BC$$
 - nor: používa sa pri **invertovaní** bitov:
 - A NOR \$0 = NOT A
- **andi, ori, xori**
 - 16b priamy operand je doplnený zľava nulami („zero-extended“)
 - nori neexistuje

Logické inštrukcie: príklad č. 1

Source Registers									
\$s1	1111	1111	1111	1111	0000	0000	0000	0000	0000
\$s2	0100	0110	1010	0001	1111	0000	1011	0111	

Assembly Code

and \$s3, \$s1, \$s2
or \$s4, \$s1, \$s2
xor \$s5, \$s1, \$s2
nor \$s6, \$s1, \$s2

Result

\$s3									
\$s4									
\$s5									
\$s6									

Logické inštrukcie: príklad č. 1

Assembly Code

```
and $s3, $s1, $s2
or $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Source Registers									
\$s1	1111	1111	1111	1111	0000	0000	0000	0000	0000
\$s2	0100	0110	1010	0001	1111	0000	1011	0111	

Result									
\$s3	0100	0110	1010	0001	0000	0000	0000	0000	0000
\$s4	1111	1111	1111	1111	1111	0000	1011	0111	
\$s5	1011	1001	0101	1110	1111	0000	1011	0111	
\$s6	0000	0000	0000	0000	0000	1111	0100	1000	

Logické inštrukcie: príklad č. 2

Assembly Code

```
andi $s2, $s1, 0xFA34    $s2  
ori  $s3, $s1, 0xFA34    $s3  
xori $s4, $s1, 0xFA34    $s4
```

Source Values							
\$s1	0000	0000	0000	0000	0000	0000	1111 1111
imm	0000	0000	0000	0000	1111	1010	0011 0100
zero-extended							
Result							

Logické inštrukcie: príklad č. 2

Assembly Code

```
andi $s2, $s1, 0xFA34  
ori $s3, $s1, 0xFA34  
xori $s4, $s1, 0xFA34
```

		Source Values									
\$s1		0000	0000	0000	0000	0000	0000	1111	1111		
imm		0000	0000	0000	0000	1111	1010	0011	0100		
		zero-extended									
		Result									
	\$s2	0000	0000	0000	0000	0000	0000	0011	0100		
	\$s3	0000	0000	0000	0000	1111	1010	1111	1111		
	\$s4	0000	0000	0000	0000	1111	1010	1100	1011		

Inštrukcie posuvu

- **sll:** logický posuv doľava
 - *Príklad:* sll \$t0, \$t1, 5 # \$t0 <= \$t1 << 5
- **srl:** logický posuv doprava
 - *Príklad:* srl \$t0, \$t1, 5 # \$t0 <= \$t1 >> 5
- **sra:** aritmetický posuv doprava
 - *Príklad :* sra \$t0, \$t1, 5 # \$t0 <= \$t1 >>> 5

Inštrukcie posuvu

- **sllv:** logický posuv doľava
 - *Príklad :* sllv \$t0, \$t1, \$t2 # \$t0 <= \$t1 << \$t2
- **srlv:** logický posuv doprava
 - *Príklad :* srlv \$t0, \$t1, \$t2 # \$t0 <= \$t1 >> \$t2
- **sraw:** aritmetický posuv doprava
 - *Príklad :* sraw \$t0, \$t1, \$t2 # \$t0 <= \$t1 >>> \$t2

Inštrukcie posuvu

Assembly Code

	op	rs	rt	rd	shamt	funct
sll \$t0, \$s1, 2	0	0	17	8	2	0
srl \$s2, \$s1, 2	0	0	17	18	2	2
sra \$s3, \$s1, 2	0	0	17	19	2	3

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

Field Values

Machine Code

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

Práca s konštantami

- 16b konštanta a inštrukcia addi:

C kód

```
// int is a 32-bit signed word
```

```
int a = 0x4f3c;
```

MIPS kód

```
# $s0 = a
```

```
addi $s0, $0, 0x4f3c
```

- 32b konštanta a čítanie najvýznamnejších 16 bitov pomocou **lui** („load upper immediate“) a vykonanie inštrukcie **ori**:

C kód

```
int a = 0xFEDC8765;
```

MIPS kód

```
# $s0 = a
```

```
lui $s0, 0xFEDC
```

```
ori $s0, $s0, 0x8765
```

Násobenie, Delenie

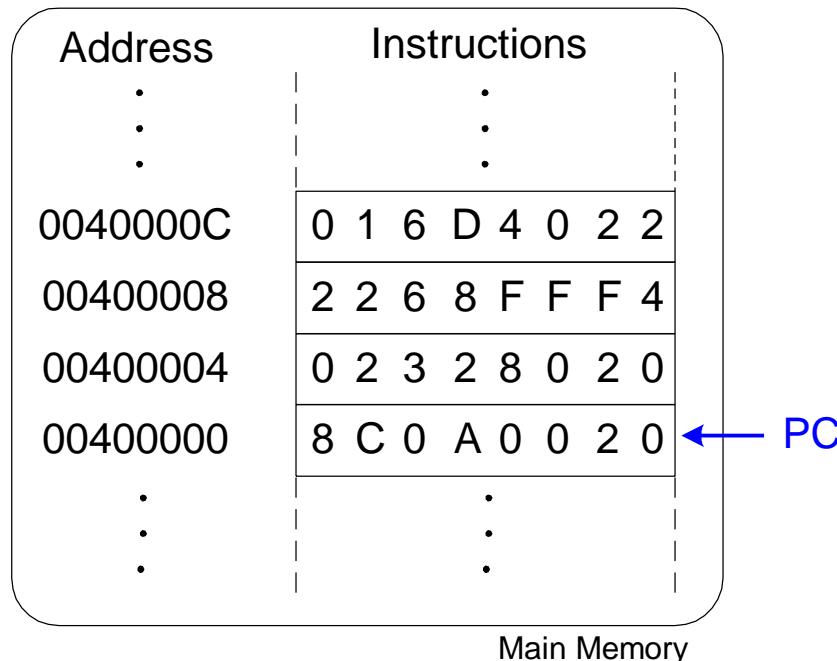
- Špeciálne registre: lo, hi
- $32 \times 32\text{b}$ násobenie, 64b výsledok
 - **mult** \$s0, \$s1
 - Výsledok v {hi, lo}
- 32b delenie, 32b podiel, zvyšok
 - **div** \$s0, \$s1
 - Podiel v lo
 - Zvyšok v hi
- Presun zo špeciálnych registrov lo/hi
 - **mflo** \$s2
 - **mfhi** \$s3

- Ovplyvňuje poradie vykonania inštrukcií
- Typy vetvení:
 - Podmienené
 - **beq** (branch if equal)
 - **bne** (branch if not equal)
 - Nepodmienené
 - **j** (jump)
 - **jr** (jump register)
 - **jal** (jump and link)

Sumarizácia

	Assembly Code	Machine Code
lw	\$t2, 32(\$0)	0x8C0A0020
add	\$s0, \$s1, \$s2	0x02328020
addi	\$t0, \$s3, -12	0x2268FFF4
sub	\$t0, \$t3, \$t5	0x016D4022

Stored Program



MIPS kód

addi \$s0, \$0, 4 # $\$s0 = 0 + 4 = 4$

addi \$s1, \$0, 1 # $\$s1 = 0 + 1 = 1$

sll \$s1, \$s1, 2 # $\$s1 = 1 \ll 2 = 4$

beq \$s0, \$s1, target # vykoná sa

addi \$s1, \$s1, 1 # nevykoná sa

sub \$s1, \$s1, \$s0 # nevykoná sa

target: # návest'

add \$s1, \$s1, \$s0 # $\$s1 = 4 + 4 = 8$

Návest'. Označujú miesta v programovom kóde. Na ich označenie nie je možné použiť kľúčové slová MIPS-u a sú ukončené dvojbodkou (:)

MIPS kód

addi	$\$s0, \$0, 4$	# $\$s0 = 0 + 4 = 4$
addi	$\$s1, \$0, 1$	# $\$s1 = 0 + 1 = 1$
sll	$\$s1, \$s1, 2$	# $\$s1 = 1 << 2 = 4$
bne	$\$s0, \$s1, \text{target}$	# sa nevykoná
addi	$\$s1, \$s1, 1$	# $\$s1 = 4 + 1 = 5$
sub	$\$s1, \$s1, \$s0$	# $\$s1 = 5 - 4 = 1$

target:

add	$\$s1, \$s1, \$s0$	# $\$s1 = 1 + 4 = 5$
------------	--------------------	----------------------

j

MIPS kód

addi \$s0, \$0, 4	# \$s0 = 4
addi \$s1, \$0, 1	# \$s1 = 1
j target	# skok na návest' <i>target</i>
sra \$s1, \$s1, 2	# nevykoná sa
addi \$s1, \$s1, 1	# nevykoná sa
sub \$s1, \$s1, \$s0	# nevykoná sa

target:

add \$s1, \$s1, \$s0	# \$s1 = 1 + 4 = 5
-----------------------------	--------------------

MIPS kód

0x00002000	addi \$s0, \$0, 0x2010
0x00002004	jr \$s0
0x00002008	addi \$s1, \$0, 1
0x0000200C	sra \$s1, \$s1, 2
0x00002010	lw \$s3, 44(\$s1)

jr je inštrukcia typu R

Príkazy

- Príkaz „if“
- Príkaz „if/else“
- Cyklus „while“
- Cyklus „for“

Príkaz „if“

C kód

```
if (i == j)  
    f = g + h;  
  
f = f - i;
```

MIPS kód

```
# $s0 = f, $s1 = g, $s2 = h  
# $s3 = i, $s4 = j
```

Príkaz „if“

C kód

```
if (i == j)  
    f = g + h;
```

```
f = f - i;
```

MIPS kód

```
# $s0 = f, $s1 = g, $s2 = h  
# $s3 = i, $s4 = j  
bne $s3, $s4, L1  
add $s0, $s1, $s2
```

```
L1: sub $s0, $s0, $s3
```

V asembleri sa testuje podmienka ($i \neq j$) na rozdiel od C kódu ($i == j$)

Príkaz „if/else“

C kód

```
if (i == j)  
    f = g + h;  
else  
    f = f - i;
```

MIPS kód

Príkaz „if/else“

C kód

```
if (i == j)
    f = g + h;
else
    f = f - i;
```

MIPS kód

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
bne $s3, $s4, L1
add $s0, $s1, $s2
j done
L1: sub $s0, $s0, $s3
done:
```

Príkaz „while“

C kód

```
// determines the power  
// of x such that  $2^x = 128$   
int pow = 1;  
int x = 0;  
  
while (pow != 128) {  
    pow = pow * 2;  
    x = x + 1;  
}
```

MIPS kód

V asembleri sa testuje podmienka ($pow == 128$) na rozdiel od C kódu ($pow != 128$).

Príkaz „while“

C kód

```
// determines the power  
// of x such that  $2^x = 128$   
  
int pow = 1;  
int x = 0;  
  
while (pow != 128) {  
    pow = pow * 2;  
    x = x + 1;  
}
```

MIPS akód

```
# $s0 = pow, $s1 = x  
  
addi $s0, $0, 1  
add $s1, $0, $0  
addi $t0, $0, 128  
while: beq $s0, $t0, done  
sll $s0, $s0, 1  
addi $s1, $s1, 1  
j while  
done:
```

V asembleri sa testuje podmienka ($pow == 128$) na rozdiel od C kódu ($pow != 128$).

Príkaz „for“

**for (inicializácia; podmienka; operácie riadenia cyklu)
príkaz(y)**

- **inicializácia:** vykoná sa pred cyklom
- **podmienka:** testuje sa pred začatím realizácie cyklu
- **operácie riadenia cyklu:** vykonajú sa po každom cykle
- **príkaz(y):** vykonajú sa, ak podmienka cyklu bola splnená

Príkaz „for“

C kód

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}
```

MIPS kód

```
# $s0 = i, $s1 = sum
```

Príkaz „for“

C kód

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}
```

MIPS kód

Príkaz „for“

C kód

```
// add the numbers from 0 to 9  
int sum = 0;  
int i;  
  
for (i=0; i!=10; i = i+1) {  
    sum = sum + i;  
}
```

MIPS kód

```
# $s0 = i, $s1 = sum  
addi $s1, $0, 0  
add $s0, $0, $0  
addi $t0, $0, 10  
for: beq $s0, $t0, done  
add $s1, $s1, $s0  
addi $s0, $s0, 1  
j for  
done:
```

Porovnávanie / relačné operátory

C kód

```
// add the powers of 2 from 1  
// to 100  
int sum = 0;  
int i;  
  
for (i=1; i < 101; i = i*2) {  
    sum = sum + i;  
}
```

MIPS kód

Porovnávanie / relačné operátory

C kód

```
// add the powers of 2 from 1  
// to 100  
int sum = 0;  
int i;  
  
for (i=1; i < 101; i = i*2) {  
    sum = sum + i;  
}
```

MIPS kód

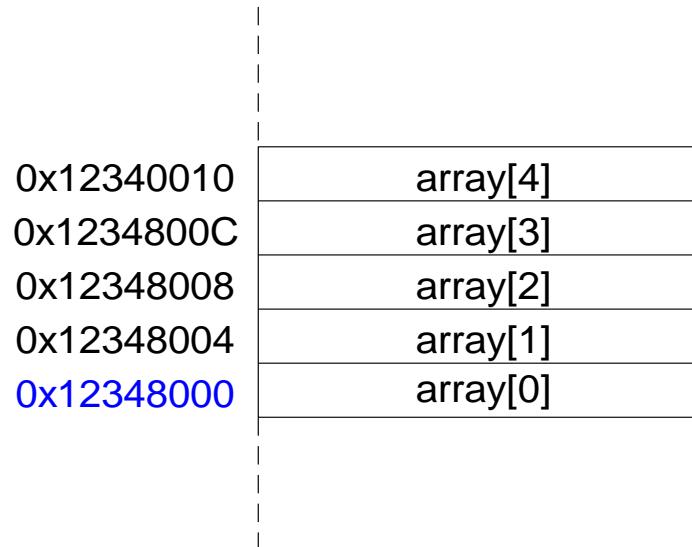
```
# $s0 = i, $s1 = sum  
addi $s1, $0, 0  
addi $s0, $0, 1  
addi $t0, $0, 101  
loop: slt $t1, $s0, $t0  
beq $t1, $0, done  
add $s1, $s1, $s0  
sll $s0, $s0, 1  
j loop  
done:
```

\$t1 = 1 if i < 101

- Kompozitná dátová štruktúra
 - Veľa rovnakých prvkov
- **Index**: určuje pozíciu prvku v poli
- **Veľkosť**: počet prvkov v poli

Polia

- Pole veľkosti 5 prvkov
- **Bázová adresa** = 0x12348000 (pozícia prvého prvku v poli, array[0])
- Prvý krok pri práci s poľom: čítaj bázovú adresu poľa



Sprístupnenie prvkov pola

// C kód

```
int array[5];
array[0] = array[0] * 2;
array[1] = array[1] * 2;
```

Sprístupnenie prvkov pola

// C kód

```
int array[5];
array[0] = array[0] * 2;
array[1] = array[1] * 2;
```

MIPS kód

array base address = \$s0

lui \$s0, 0x1234	# 0x1234 in upper half of \$s0
ori \$s0, \$s0, 0x8000	# 0x8000 in lower half of \$s0

lw \$t1, 0(\$s0)	# \$t1 = array[0]
sll \$t1, \$t1, 1	# \$t1 = \$t1 * 2
sw \$t1, 0(\$s0)	# array[0] = \$t1

lw \$t1, 4(\$s0)	# \$t1 = array[1]
sll \$t1, \$t1, 1	# \$t1 = \$t1 * 2
sw \$t1, 4(\$s0)	# array[1] = \$t1

Polia a „for“ cyklus

// C kód

```
int array[1000];
int i;

for (i=0; i < 1000; i = i + 1)
    array[i] = array[i] * 8;
```

MIPS kód

```
# $s0 = array base address, $s1 = i
```

Polia a „for“ cyklus

MIPS kód

\$s0 = array base address, \$s1 = i

initialization code

```
lui $s0, 0x23B8      # $s0 = 0x23B80000
ori $s0, $s0, 0xF000 # $s0 = 0x23B8F000
addi $s1, $0, 0       # i = 0
addi $t2, $0, 1000    # $t2 = 1000
```

loop:

```
slt $t0, $s1, $t2    # i < 1000?
beq $t0, $0, done    # if not then done
sll $t0, $s1, 2       # $t0 = i * 4 (byte offset)
add $t0, $t0, $s0     # address of array[i]
lw $t1, 0($t0)        # $t1 = array[i]
sll $t1, $t1, 3       # $t1 = array[i] * 8
sw $t1, 0($t0)        # array[i] = array[i] * 8
addi $s1, $s1, 1       # i = i + 1
j loop                # repeat
```

done:

ASCII kód

- *American Standard Code for Information Interchange*
- Každý znak má svoj unikátny kód
 - Napríklad, S = 0x53, o = 0x6F, S = 0x53
 - Vzdialenosť medzi kódmi, ktoré reprezentujú malé písmeno a veľké písmeno je 0x20 (32)

ASCII tabuľka

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	'	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	\$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(38	8	48	H	58	X	68	h	78	x
29)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	:	4B	K	5B	[6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	
2D	-	3D	=	4D	M	5D]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

Volanie funkcie

- **Volajúci:** v našom prípade je to *main()*
- **Volaný:** v našom prípade je to *sum()*

C kód

```
void main()
{
    int y;
    y = sum(42, 7);
    ...
}
```

```
int sum(int a, int b)
{
    return (a + b);
}
```

Práca s funkciami

- **Volajúci:**
 - odovzdáva **argumenty** volanej funkcie
 - „Vykoná skok“ na volanú funkciu
- **Volaný:**
 - **Vykoná funkciu**
 - **Vráti výsledok** výpočtu volajúcej funkcie
 - **Zabezpečí návrat** na miesto volania
 - **Nesmie prepísať obsah registrov a pamäťových miest**, ktoré používa volajúci

Práca s funkciami v MIPS

- **Volanie funkcie:** inštrukcia jal (jump and link)
- **Návrat z funkcie:** inštrukcia jr (jump register)
- **Argumenty:** \$a0 - \$a3
- **Návratová hodnota:** \$v0

Volanie funkcie

C kód

```
int main() {  
    simple();  
    a = b + c;  
}
```

```
void simple() {  
    return;  
}
```

MIPS kód

```
0x00400200 main: jal simple  
0x00400204      add $s0, $s1, $s2  
...  
0x00401020 simple: jr $ra
```

typ „void“ znamená, že funkcia nevracia hodnotu

Volanie funkcie

C kód

```
int main() {  
    simple();  
    a = b + c;  
}
```

```
void simple() {  
    return;  
}
```

MIPS kód

```
0x00400200 main: jal simple  
0x00400204      add $s0, $s1, $s2  
...
```

```
0x00401020 simple: jr $ra
```

jal: skok na návest' simple

$$\$ra = PC + 4 = 0x00400204$$

jr \$ra: skok na adresu uloženú v \$ra (0x00400204)

Argumenty & návratová hodnota

MIPS konvencia:

- Hodnoty argumentov: \$a0 - \$a3
- Návratová hodnota: \$v0

Argumenty & návratová hodnota

C kód

```
int main()
{
    int y;
    ...
    y = diffofsums(2, 3, 4, 5); // 4 arguments
    ...
}

int diffofsums(int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    return result;           // return value
}
```

Argumenty & návratová hodnota

MIPS kód

\$s0 = y

main:

...

addi \$a0, \$0, 2 # argument 0 = 2
addi \$a1, \$0, 3 # argument 1 = 3
addi \$a2, \$0, 4 # argument 2 = 4
addi \$a3, \$0, 5 # argument 3 = 5
jal diffofsums # call Function
add \$s0, \$v0, \$0 # y = returned value

...

\$s0 = result

diffofsums:

add \$t0, \$a0, \$a1 # \$t0 = f + g
add \$t1, \$a2, \$a3 # \$t1 = h + i
sub \$s0, \$t0, \$t1 # result = (f + g) - (h + i)
add \$v0, \$s0, \$0 # put return value in \$v0

Argumenty & návratová hodnota

MIPS kód

\$s0 = result

diffofsums:

```
add $t0, $a0, $a1    # $t0 = f + g  
add $t1, $a2, $a3    # $t1 = h + i  
sub $s0, $t0, $t1    # result = (f + g) - (h + i)  
add $v0, $s0, $0      # put return value in $v0  
jr $ra                # return to caller
```

- Funkcia diffofsums prepíše obsah 3 registerov:
 \$t0, \$t1, \$s0
- Funkcia diffofsums môže používať
 zásobník
na dočasné uchovávanie „nejakých“ hodnôt

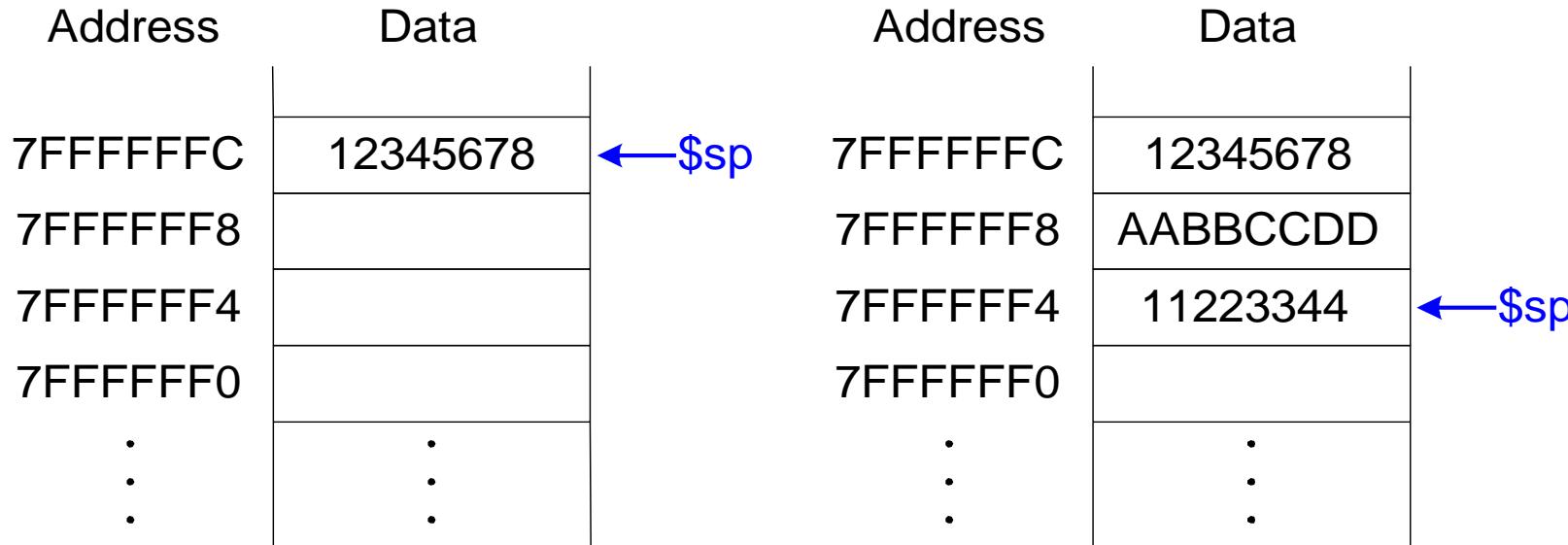
Zásobník

- Pamäť pre dočasné premenné
- Je typu LIFO (last-in-first-out)
- **Zväčšuje sa dynamicky:**
 - Ak potrebujem ukladať viac dát
 - ale len do určitej veľkosti
- **Zmenšuje sa dynamicky:**
 - Ak už nepotrebujem tie dátá



Zásobník

- Rastie smerom nadol (od adresy s väčšou hodnotou k adrese s menšou)
- Smerník zásobníka: \$sp ukazuje na vrchol zásobníka



Práca so zásobníkom

- Volaná funkcia môže ovplyvniť volajúcu funkciu jedine cez návratovú hodnotu
- Avšak funkcia diffofsums prepisuje obsah 3 registerov: \$t0, \$t1, \$s0

MIPS kód

\$s0 = result

diffofsums:

add \$t0, \$a0, \$a1

\$t0 = f + g

add \$t1, \$a2, \$a3

\$t1 = h + i

sub \$s0, \$t0, \$t1

result = (f + g) - (h + i)

add \$v0, \$s0, \$0

put return value in \$v0

jr \$ra

return to caller

Práca so zásobníkom

\$s0 = result

diffofsums:

addi \$sp, \$sp, -12 # make space on stack

to store 3 registers

sw \$s0, 8(\$sp) # save \$s0 on stack

sw \$t0, 4(\$sp) # save \$t0 on stack

sw \$t1, 0(\$sp) # save \$t1 on stack

add \$t0, \$a0, \$a1 # \$t0 = f + g

add \$t1, \$a2, \$a3 # \$t1 = h + i

sub \$s0, \$t0, \$t1 # result = (f + g) - (h + i)

add \$v0, \$s0, \$0 # put return value in \$v0

lw \$t1, 0(\$sp) # restore \$t1 from stack

lw \$t0, 4(\$sp) # restore \$t0 from stack

lw \$s0, 8(\$sp) # restore \$s0 from stack

addi \$sp, \$sp, 12 # deallocate stack space

jr \$ra # return to caller

Zásobník počas diffosums

Address Data

FC	?
F8	
F4	
F0	
.	.
.	.
.	.

(a)

Address Data

FC	?
F8	\$s0
F4	\$t0
F0	\$t1
.	.
.	.
.	.

(b)

Address Data

FC	?
F8	
F4	
F0	
.	.
.	.
.	.

(c)

← \$sp

stack frame

← \$sp

← \$sp

Registre

Uchovávané <i>Bezpečné z pohľadu volanej funkcie</i>	Neuchovávané <i>Bezpečné z pohľadu volajúcej funkcie</i>
\$s0-\$s7	\$t0-\$t9
\$ra	\$a0-\$a3
\$sp	\$v0-\$v1
pamäť nad \$sp	pamäť pod \$sp

Volanie viacero funkcií

proc1:

addi \$sp, \$sp, -4

make space on stack

sw \$ra, 0(\$sp)

save \$ra on stack

jal proc2

...

lw \$ra, 0(\$sp)

restore \$s0 from stack

addi \$sp, \$sp, 4

deallocate stack space

jr \$ra

return to caller

Ukladanie \$s registrov do zásobníka

\$s0 = result

diffofsums:

addi \$sp, \$sp, -4

make space on stack to

store one register

sw \$s0, 0(\$sp)

save \$s0 on stack

no need to save \$t0 or \$t1

add \$t0, \$a0, \$a1

\$t0 = f + g

add \$t1, \$a2, \$a3

\$t1 = h + i

result = (f + g) - (h + i)

sub \$s0, \$t0, \$t1

put return value in \$v0

add \$v0, \$s0, \$0

restore \$s0 from stack

lw \$s0, 0(\$sp)

deallocate stack space

addi \$sp, \$sp, 4

return to caller

jr \$ra

Rekurzívne volanie funkcií

C kód

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n * factorial(n-1));  
}
```

Rekurzívne volanie funkcií

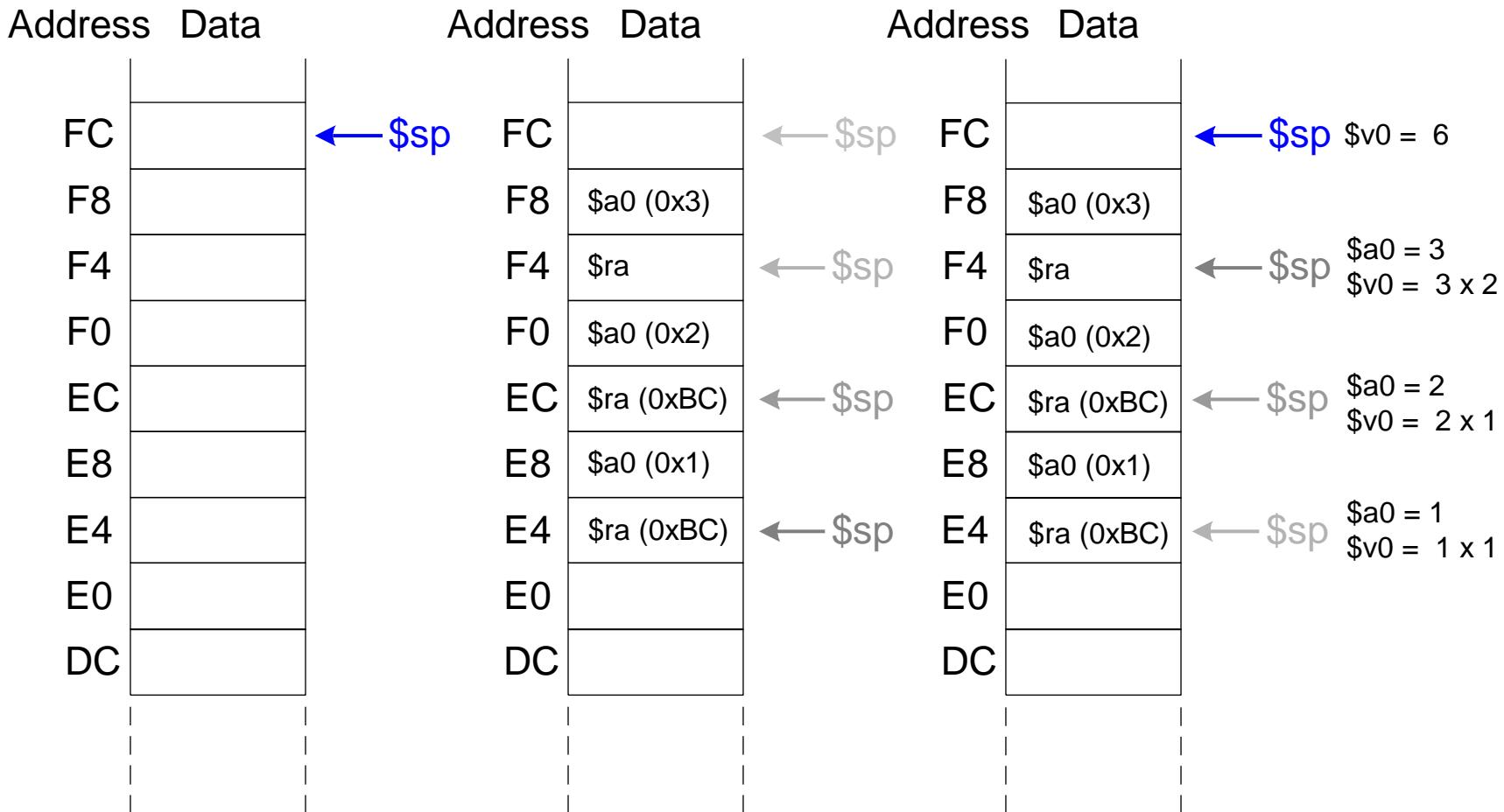
MIPS kód

Rekurzívne volanie funkcií

MIPS kód

```
0x90 factorial: addi $sp, $sp, -8      # make room
0x94      sw  $a0, 4($sp)        # store $a0
0x98      sw  $ra, 0($sp)        # store $ra
0x9C      addi $t0, $0, 2
0xA0      slt $t0, $a0, $t0      # a <= 1 ?
0xA4      beq $t0, $0, else      # no: go to else
0xA8      addi $v0, $0, 1        # yes: return 1
0xAC      addi $sp, $sp, 8        # restore $sp
0xB0      jr  $ra             # return
0xB4      else: addi $a0, $a0, -1    # n = n - 1
0xB8      jal factorial       # recursive call
0xBC      lw   $ra, 0($sp)        # restore $ra
0xC0      lw   $a0, 4($sp)        # restore $a0
0xC4      addi $sp, $sp, 8        # restore $sp
0xC8      mul $v0, $a0, $v0      # n * factorial(n-1)
0xCC      jr  $ra             # return
```

Zásobník počas rekurzie



Sumarizácia

- **Volajúci**
 - Argumenty ukladá do \$a0-\$a3
 - Uchová hodnoty potrebných registrov (\$ra, niekedy aj \$t0-t9)
 - Vykoná jal na volanú funkciu
 - Obnoví obsah uchovaných registrov
 - Načíta výsledok z \$v0 (ak je)
- **Volaný**
 - Uchová hodnoty registrov, ktorých obsah môže byť zmenený (\$s0-\$s7)
 - Vykoná inštrukcie v nej definované
 - Uloží výsledok výpočtu do \$v0 (ak sa to vyžaduje)
 - Obnoví obsah registrov
 - Vykoná inštrukciu jr \$ra

Adresovacie módy

Ako sa adresujú operandy?

- Registrový mód
- Bezprostredný / priamy mód
- Posúvací mód
- PC-Relatívny mód
- Nepriamy indexový

Registrvý mód

- Operand je uložený v registri
 - **add** \$s0, \$t2, \$t3
 - **sub** \$t8, \$s1, \$0

Bezprostredný mód

- 16b konštantou je súčasťou inštrukcie
 - **addi** \$s4, \$t5, -73
 - **ori** \$t3, \$t7, 0xFF

Posúvací mód

- Adresa operanda sa počíta ako:
 - Bázová adresa + posuv (definovaný konštantou v DK)
 - **lw** \$s4, 72(\$0)
 - adresa = \$0 + 72
 - **sw** \$t2, -25(\$t1)
 - adresa = \$t1 - 25

Adresovacie módy

PC-Relatívny mód

0x10	beq	\$t0, \$0, else
0x14	addi	\$v0, \$0, 1
0x18	addi	\$sp, \$sp, i
0x1C	jr	\$ra
0x20	else:	addi \$a0, \$a0, -1
0x24		jal factorial

Assembly Code

beq \$t0, \$0, else
(beq \$t0, \$0, 3)

Field Values

op	rs	rt	imm	
4	8	0	3	

6 bits 5 bits 5 bits 5 bits 6 bits

Adresovacie módy

Nepriamy indexovací mód

0x0040005C jal sum

...

0x004000A0 sum: add \$v0, \$a0, \$a1

JTA 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)

26-bit addr 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)
 0 1 0 0 0 0 2 8

Field Values

op	imm
3	0x0100028

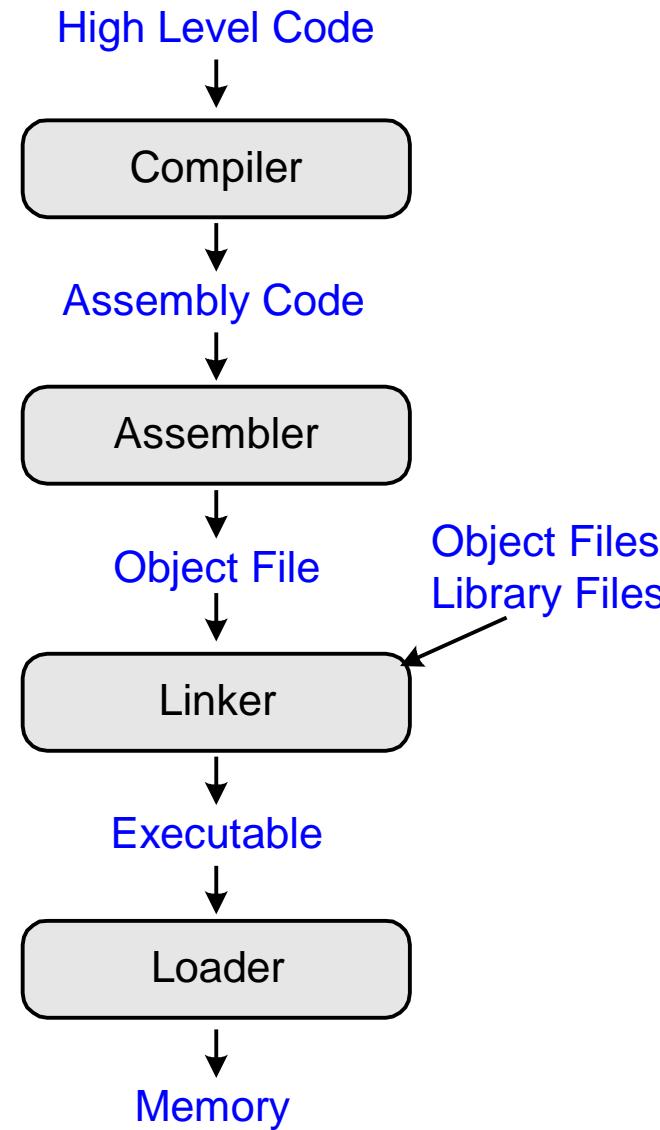
6 bits 26 bits

Machine Code

op	addr
000011	00 0001 0000 0000 0000 0010 1000 (0x0C100028)

6 bits 26 bits

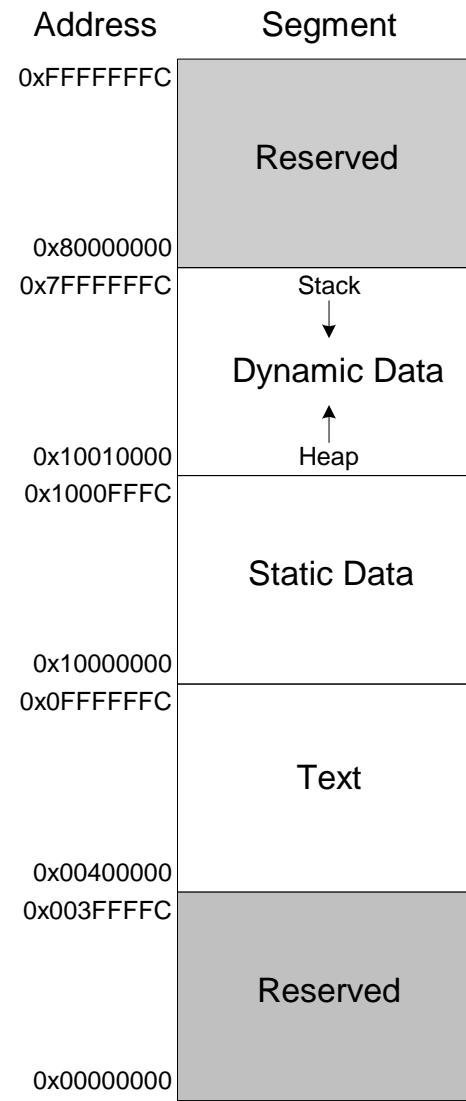
Preklad & spúšťanie programu



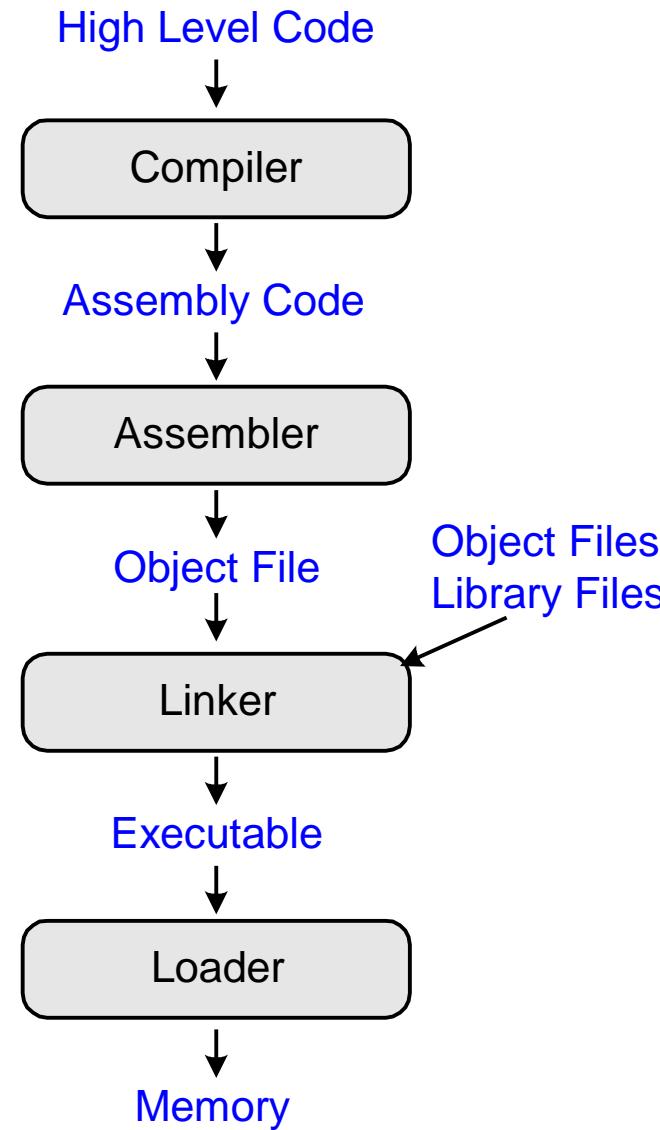
Čo je uložené v pamäti?

- Inštrukcie
- Dáta
 - Globálne/statické: alokované pred spustením programu
 - Dynamické: alokované počas behu programu
- Aká je veľkosť pamäte?
 - $2^{32} = 4$ gigabajty (4 GB)
 - Od adresy 0x00000000 až po 0xFFFFFFFF

MIPS pamäť



Preklad & spúšťanie programu



Príklad: Kód v C

```
int f, g, y; // global variables
```

```
int main(void)
```

```
{
```

```
    f = 2;
```

```
    g = 3;
```

```
    y = sum(f, g);
```

```
    return y;
```

```
}
```

```
int sum(int a, int b) {
```

```
    return (a + b);
```

```
}
```

Príklad: MIPS kód

```
int f, g, y; // global
```

```
int main(void)
{
```

```
    f = 2;
    g = 3;
```

```
    y = sum(f, g);
    return y;
}
```

```
int sum(int a, int b) {
    return (a + b);
}
```

```
.data
f:
g:
y:
.text
main:
    addi $sp, $sp, -4      # stack frame
    sw  $ra, 0($sp)        # store $ra
    addi $a0, $0, 2         # $a0 = 2
    sw  $a0, f              # f = 2
    addi $a1, $0, 3         # $a1 = 3
    sw  $a1, g              # g = 3
    jal sum                # call sum
    sw  $v0, y              # y = sum()
    lw   $ra, 0($sp)        # restore $ra
    addi $sp, $sp, 4         # restore $sp
    jr  $ra                 # return to OS
sum:
    add $v0, $a0, $a1        # $v0 = a + b
    jr  $ra                 # return
```

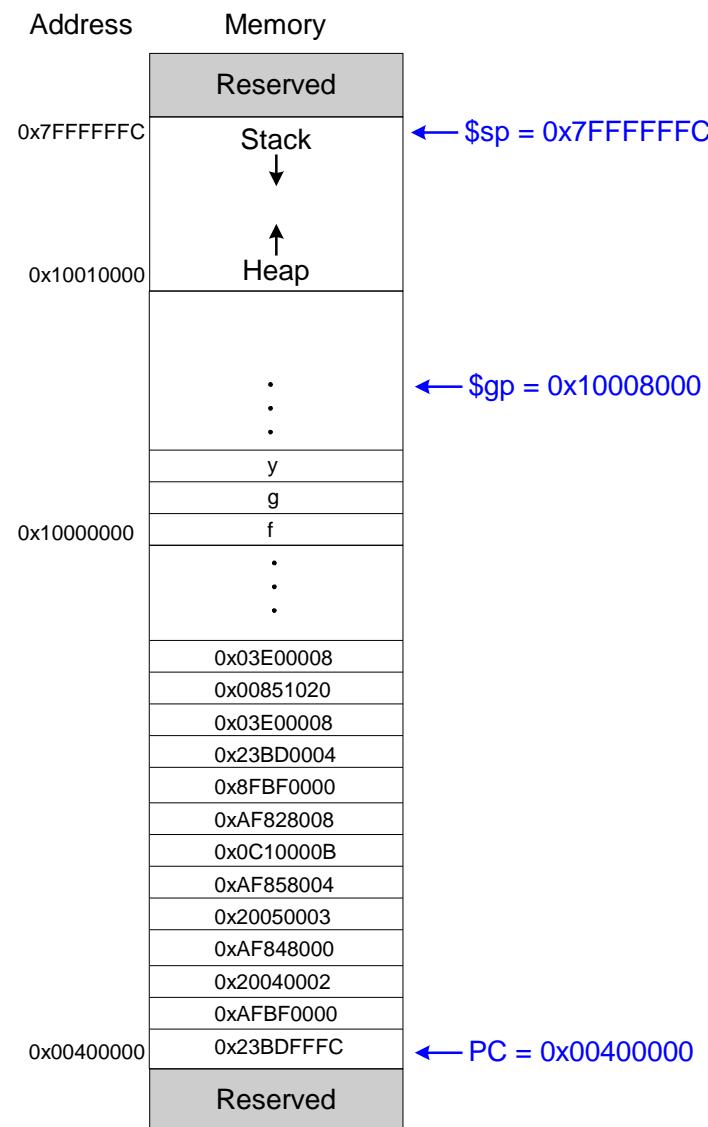
Príklad: Tabuľka symbolických adries

Symbol	Adresa
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

Príklad: Spustiteľný kód

Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)
Text segment	Address	Instruction
	0x00400000	0x23BDFFFFC
	0x00400004	sw \$ra, 0 (\$sp)
	0x00400008	addi \$a0, \$0, 2
	0x0040000C	sw \$a0, 0x8000 (\$gp)
	0x00400010	addi \$a1, \$0, 3
	0x00400014	sw \$a1, 0x8004 (\$gp)
	0x00400018	jal 0x0040002C
	0x0040001C	sw \$v0, 0x8008 (\$gp)
	0x00400020	lw \$ra, 0 (\$sp)
	0x00400024	addi \$sp, \$sp, -4
	0x00400028	jr \$ra
	0x0040002C	add \$v0, \$a0, \$a1
	0x00400030	jr \$ra
Data segment	Address	Data
	0x10000000	f
	0x10000004	g
	0x10000008	y

Príklad: Spustiteľný kód v pamäti



← \$sp = 0x7FFFFFFC

← \$gp = 0x10008000

← PC = 0x00400000

Referencia

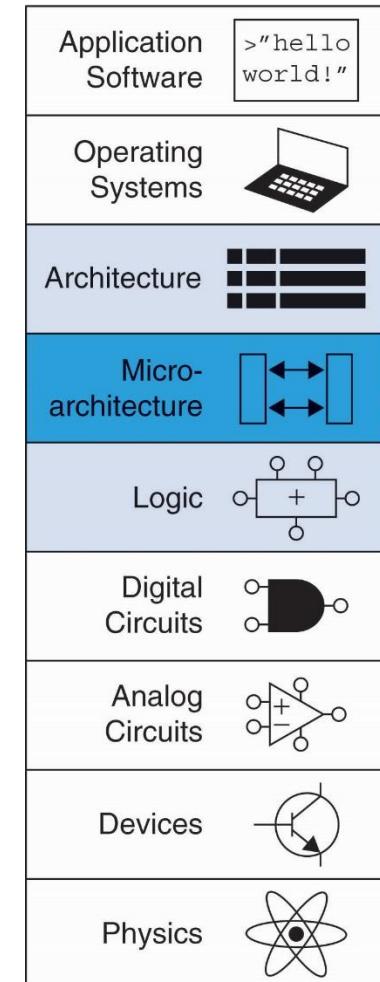
- David Money Harris and Sarah L. Harris,
Digital Design and Computer Architecture,
Chapter 6: Architecture, Second Edition ©
2012 by Elsevier Inc.

Architektúry počítačových systémov

2. rok ZS

Osnova

- **Úvod**
- **Výkonnostné parametre**
- **Jednocyklový procesor**
- **Viaccyklový procesor**
- **Prúdové spracovanie**
- **Spracovanie výnimiek**
- **Moderné mikroarchitektúry**



Úvod

- **Mikroarchitektúra:** abstrakcia na úrovni štruktúrnej organizácie
- Procesor:
 - **Tok dát/operandov:** spájanie funkčných jednotiek
 - **Tok riadenia:** distribúcia riadiacich signálov

Application Software	programs
Operating Systems	device drivers
Architecture	instructions registers
Micro-architecture	datapaths controllers
Logic	adders memories
Digital Circuits	AND gates NOT gates
Analog Circuits	amplifiers filters
Devices	transistors diodes
Physics	electrons

Mikroarchitektúra

- Rôzne IA pre tú istú ISA:
 - **Jednocyklová:** Každá inštrukcia sa vykoná za jeden strojový cyklus (SC)
 - **Viaccyklová:** Inštrukcie (v závislosti od typu) sa vykonajú za jeden alebo viac cyklov
 - **Prúdová:** Vykonanie inštrukcie sa rozloží na dielčie kroky & spracováva sa viac ako jedna inštrukcia v rámci daného inštrukčného cyklu

Výkonnostné parametre

- Čas vykonania programu

Čas odozvy = (#inštrukcií)(cyklus/inštrukcia)(sekundy/cyklus)

- Definícia:
 - CPI: počet SC procesora v priebehu ktorých sa vykoná jedna inštrukcia
 - Hodinová perióda: sekundy/cyklus
- Výzvou je nájsť správnu rovnováhu medzi parametrami:
 - Cena
 - Spotreba
 - Výkon

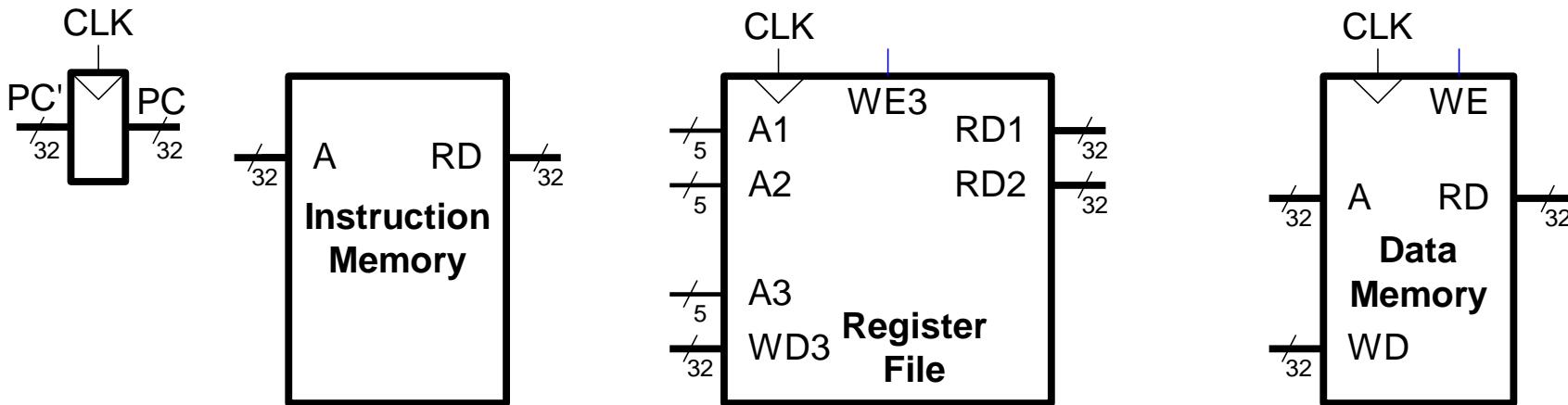
MIPS Procesor

- Nech je daná podmnožina MIPS inštrukčnej sady:
 - Inštrukcie typu R: and, or, add, sub, slt
 - Inštrukcie pre prácu s pamäťou: lw, sw
 - Inštrukcia vetvenia: beq

Vnútorný stav architektúry

- V ktorom (v akom) stave sa architektúra nachádza závisí od:
 - Hodnoty programového počítadla (PC)
 - Hodnôt uchovaných v registroch architektúry (#32)
 - Obsahu pamäte

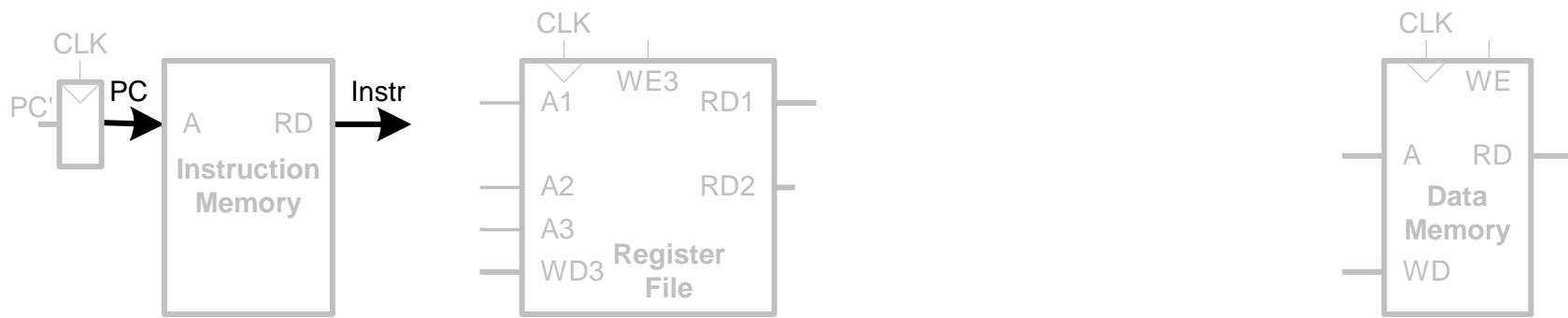
Stavové elementy MIPS



Jednocyklový MIPS procesor

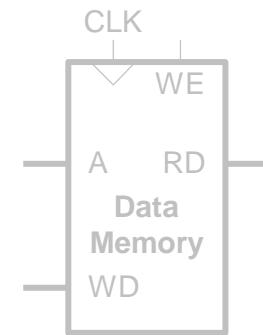
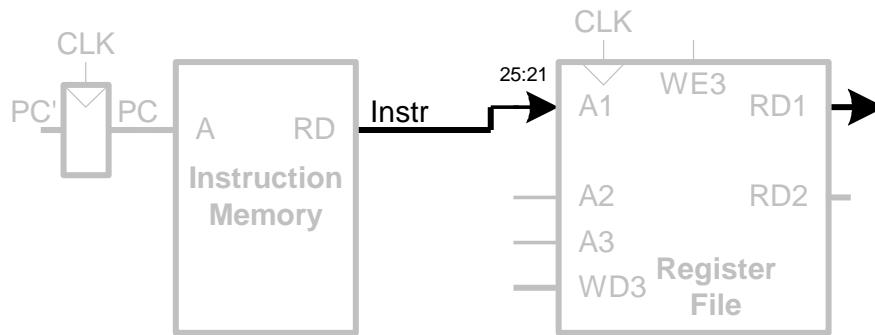
- Návrh procesora vychádza z návrhu jeho komponentov a prepojovacej sústavy, pričom sa zohľadňuje
 1. tok dát/operandov medzi funkčnými jednotkami procesora
 2. tok riadenia

Krok 1: Čítanie inštrukcie

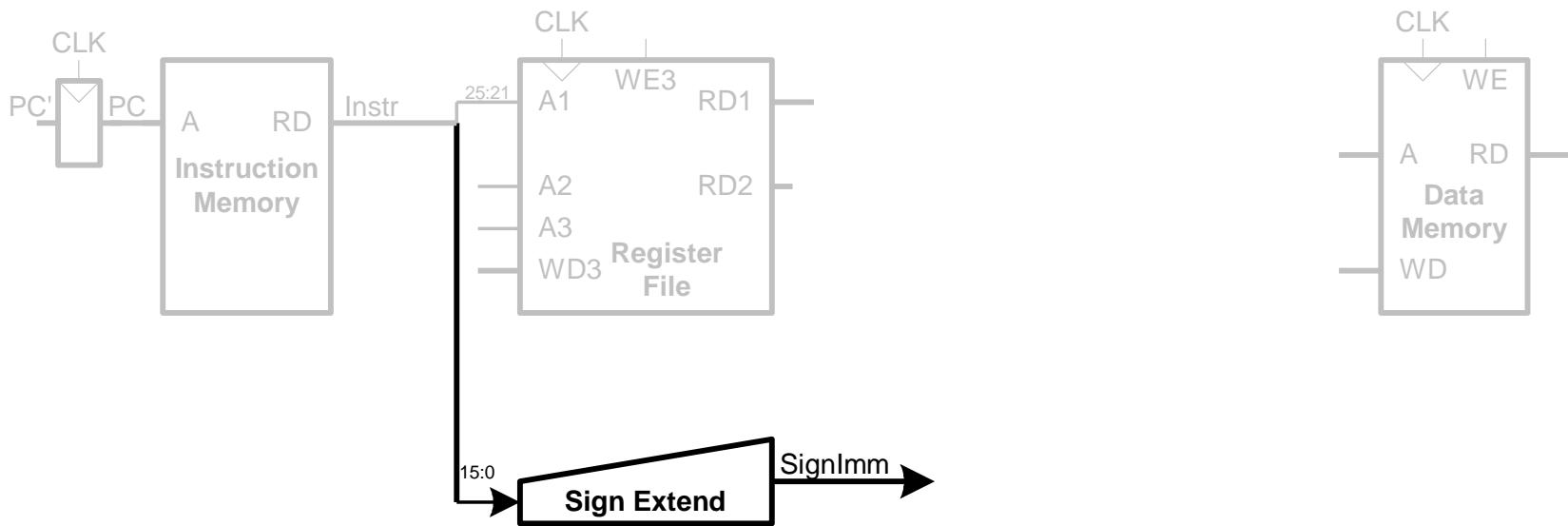


Jednocyklový MIPS procesor: čítanie z RF

Krok 2: Čítanie operandov zo súboru (pamäte) registrov (RF = register file)

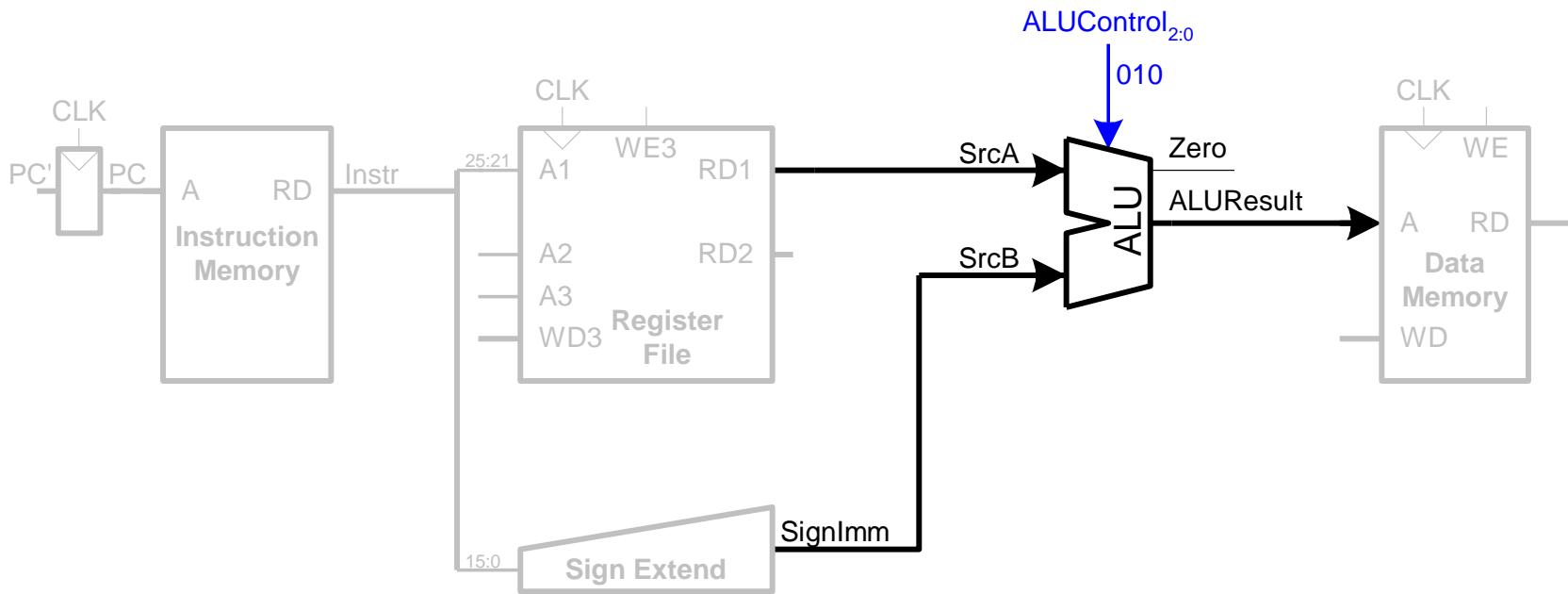


Krok 3: Podpora pre prácu s konštantami



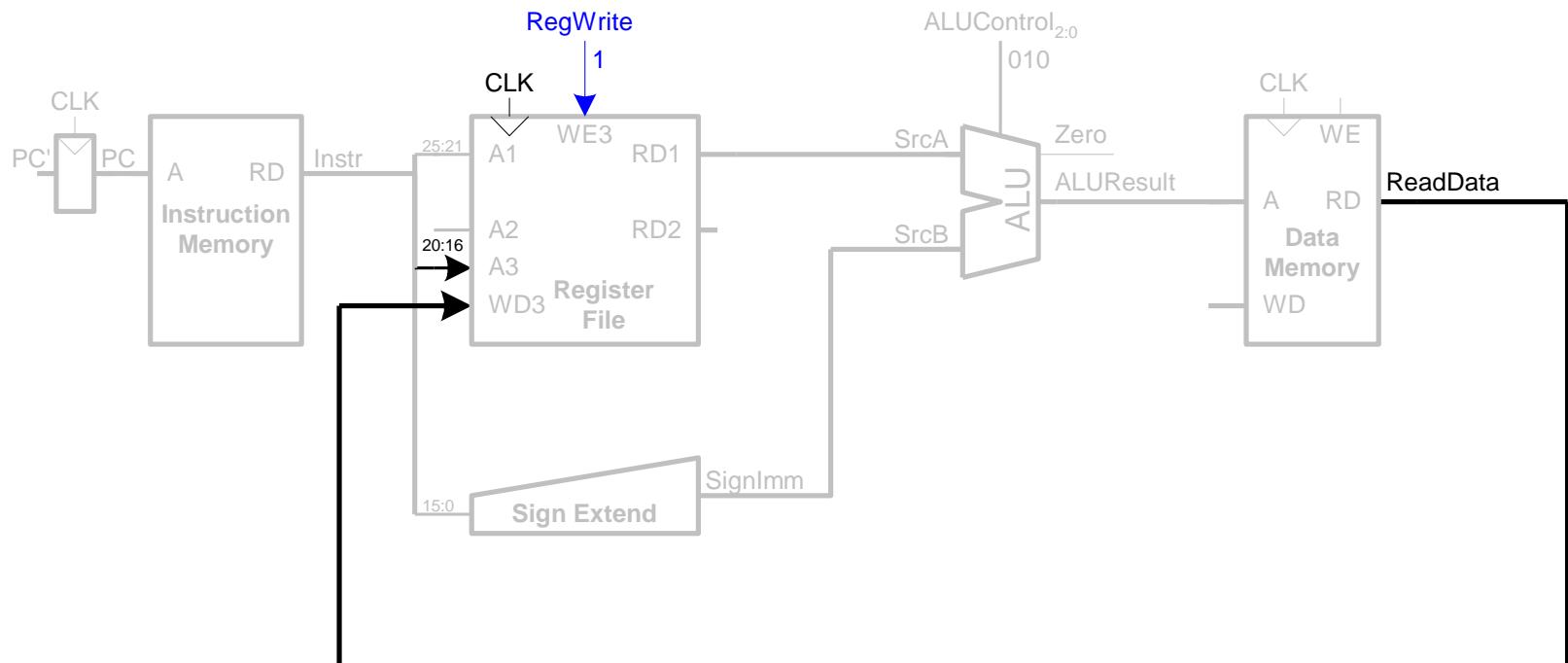
Jednocyklový MIPS procesor: výpočet adresy

Krok 4: Výpočet adresy pamäťovej bunky

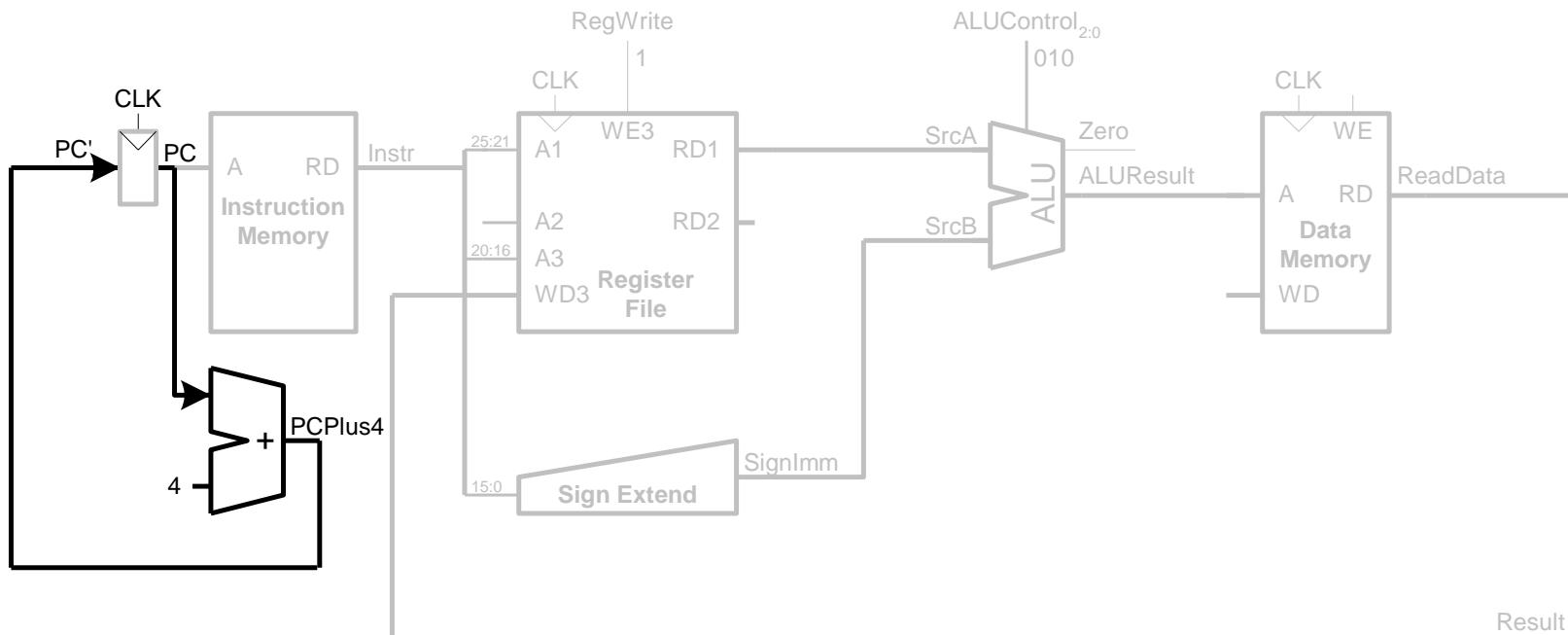


Jednocyklový MIPS procesor: čítanie z pamäte

- **Krok 5:** Čítanie dát z pamäte a zápis do súboru registrov

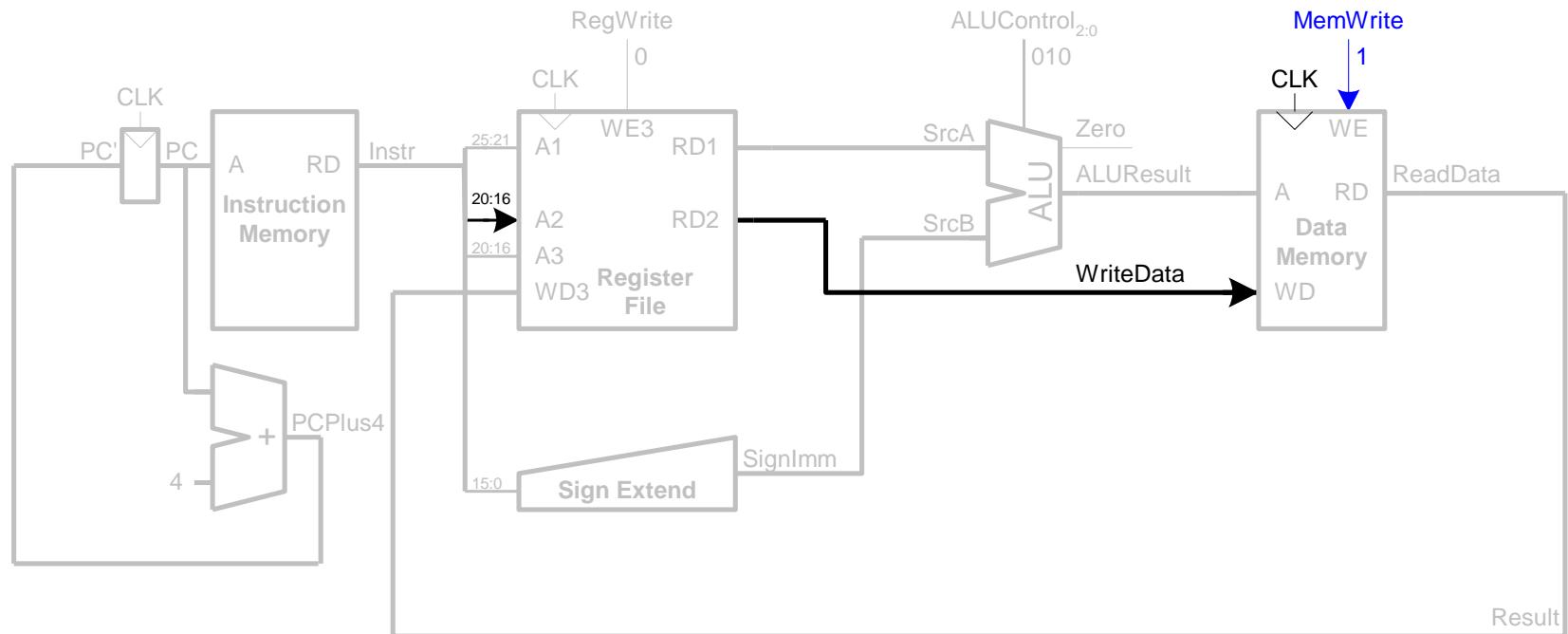


Krok 6: Určenie adresy nasledujúcej inštrukcie



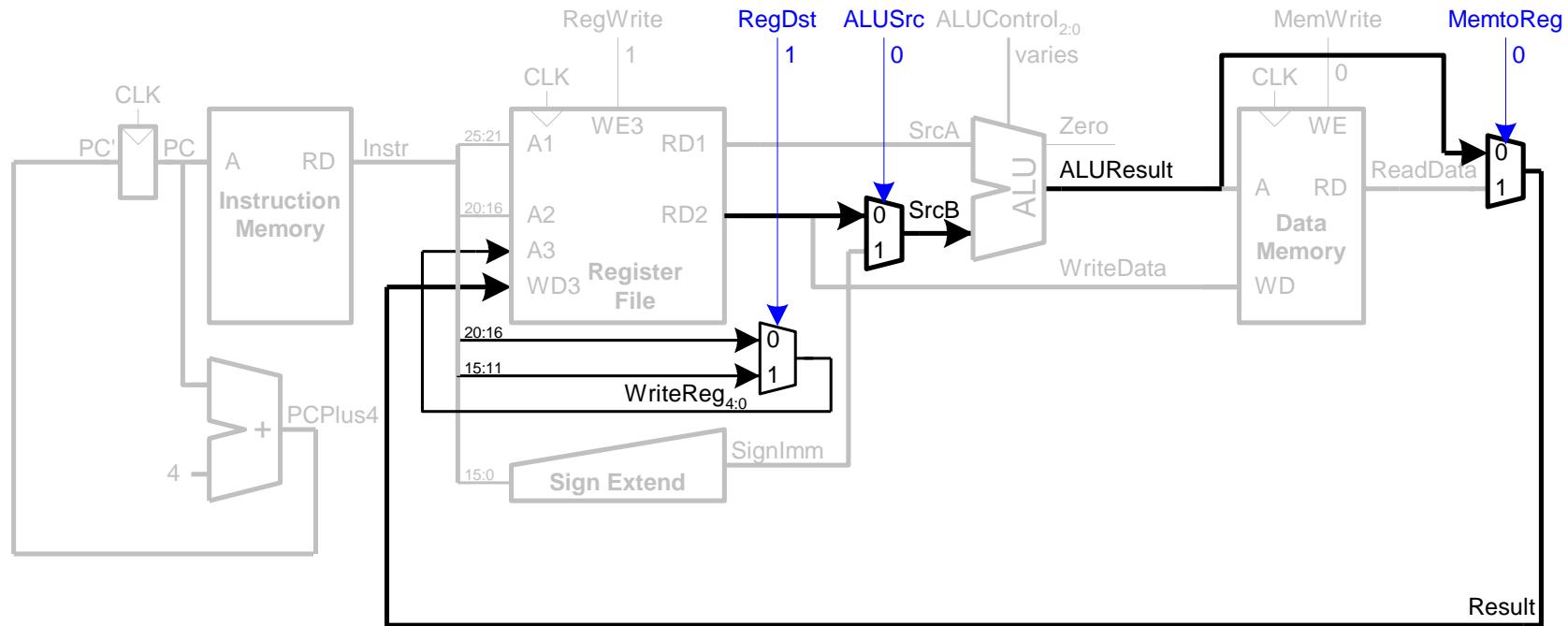
Jednocyklový MIPS procesor: zápis

Zápis dát uložených v registri rt do dátovej pamäte



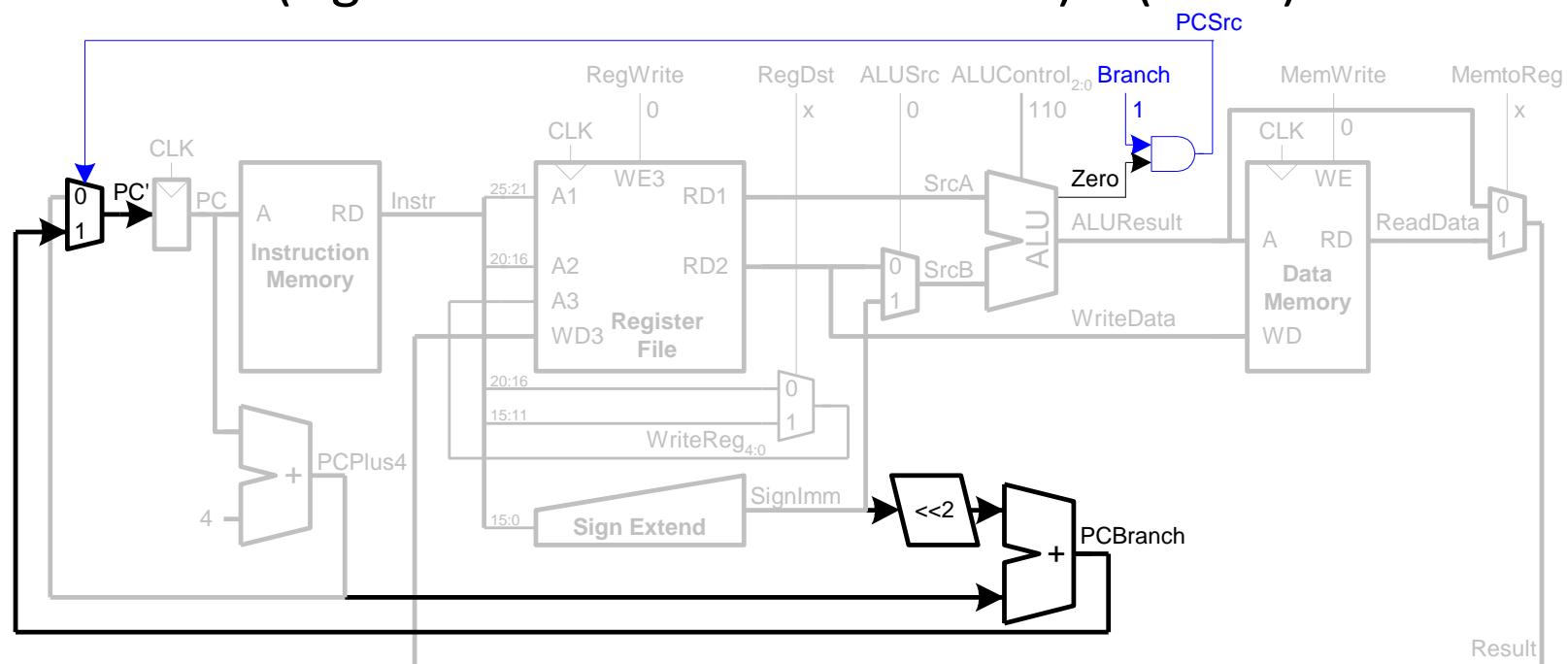
Jednocyklový MIPS procesor: inštrukcie typu R

- Čítanie z registrov rs a rt
- Zápis *ALUResult* do súboru registrov
- Zápis do registra rd (namiesto registra rt)

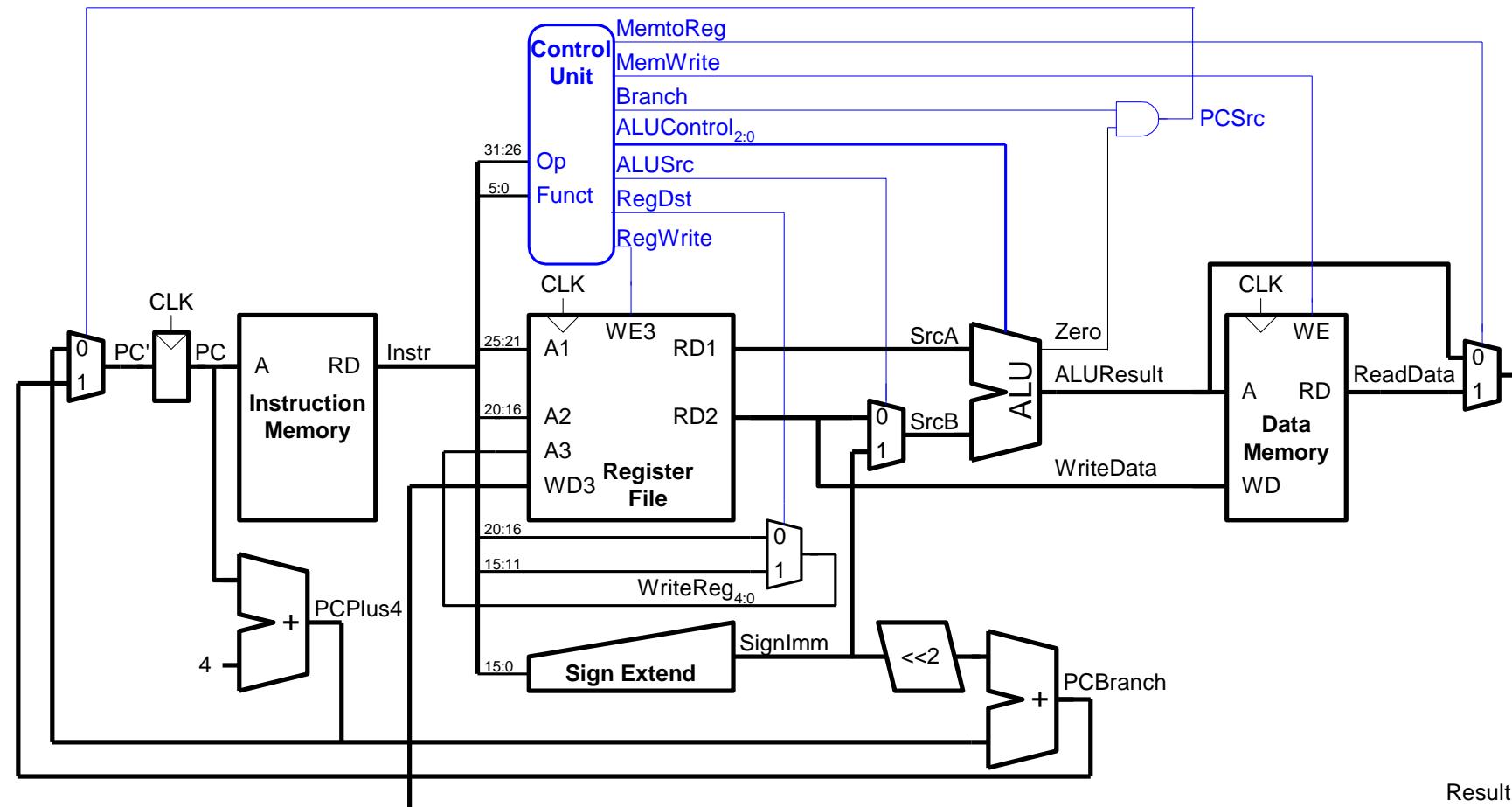


Jednocyklový MIPS procesor: beq

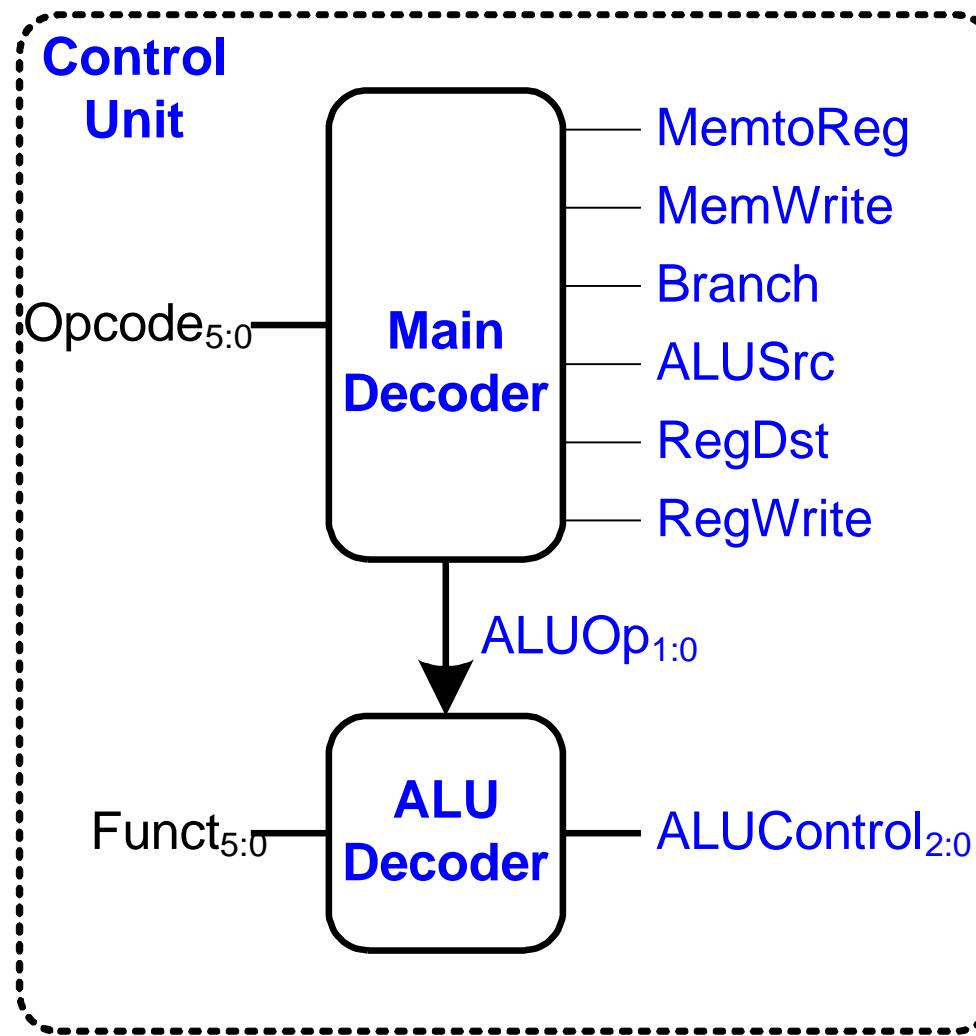
- Vyhodnotenie relácie „rovná sa“ nad obsahmi registrov rs a rt
- Výpočet adresy skoku:
$$\text{BTA} = (\text{sign-extended immediate} \ll 2) + (\text{PC} + 4)$$



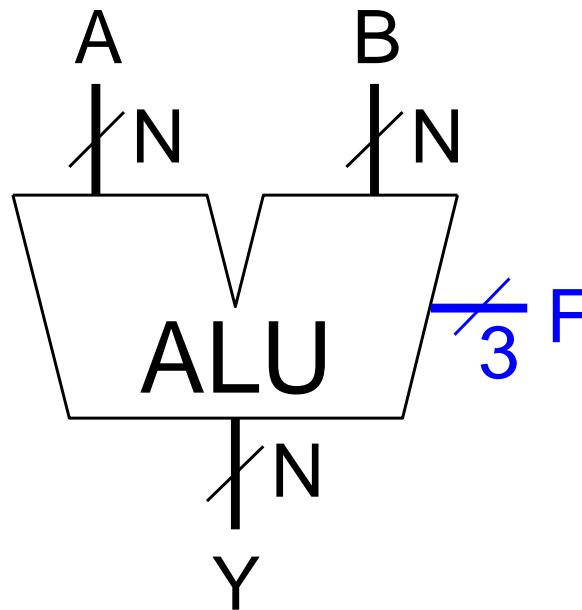
Jednocyklový MIPS procesor



Riadenie

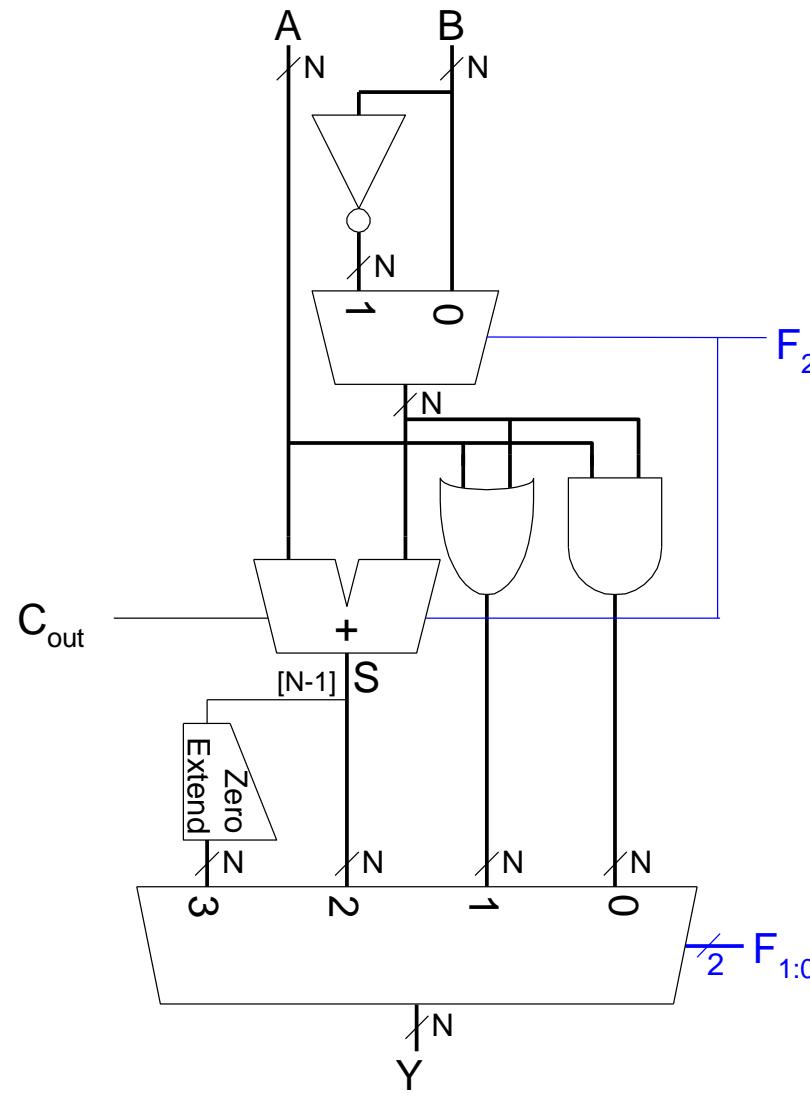


Aritmeticko-logická jednotka



$F_{2:0}$	Funkcia
000	$A \& B$
001	$A B$
010	$A + B$
011	not used
100	$A \& \sim B$
101	$A \sim B$
110	$A - B$
111	SLT

Aritmeticko-logická jednotka



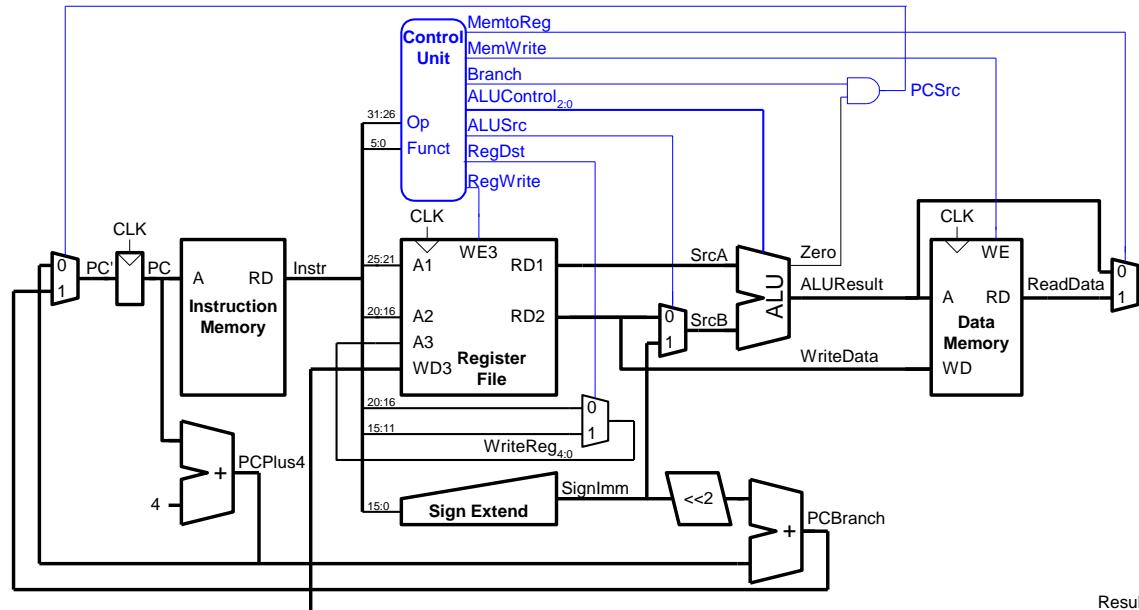
Dekodér arit.-log. jednotky

ALUOp _{1:0}	Význam
00	sčítanie
01	odčítanie
10	vid'. pole func
11	nepoužíva sa

ALUOp _{1:0}	func	ALUControl _{2:0}
00	X	010 (sčítanie)
X1	X	110 (odčítanie)
1X	100000 (add)	010 (sčítanie)
1X	100010 (sub)	110 (odčítanie)
1X	100100 (and)	000 (And)
1X	100101 (or)	001 (Or)
1X	101010 (slt)	111 (SLT)

Hlavný dekodér riadiacej časti procesora

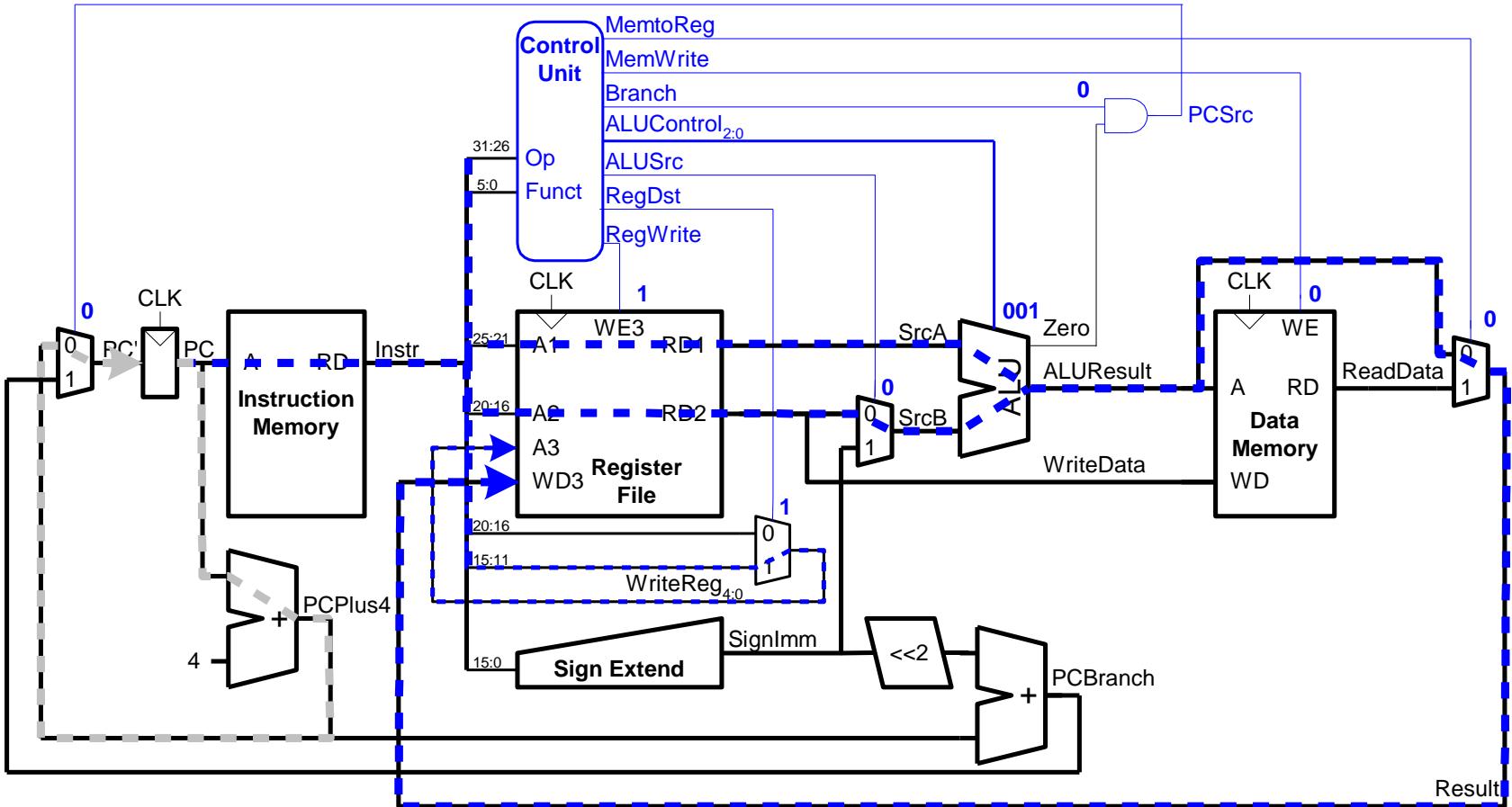
Inštrukcia	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
R-type	000000							
lw	100011							
sw	101011							
beq	000100							



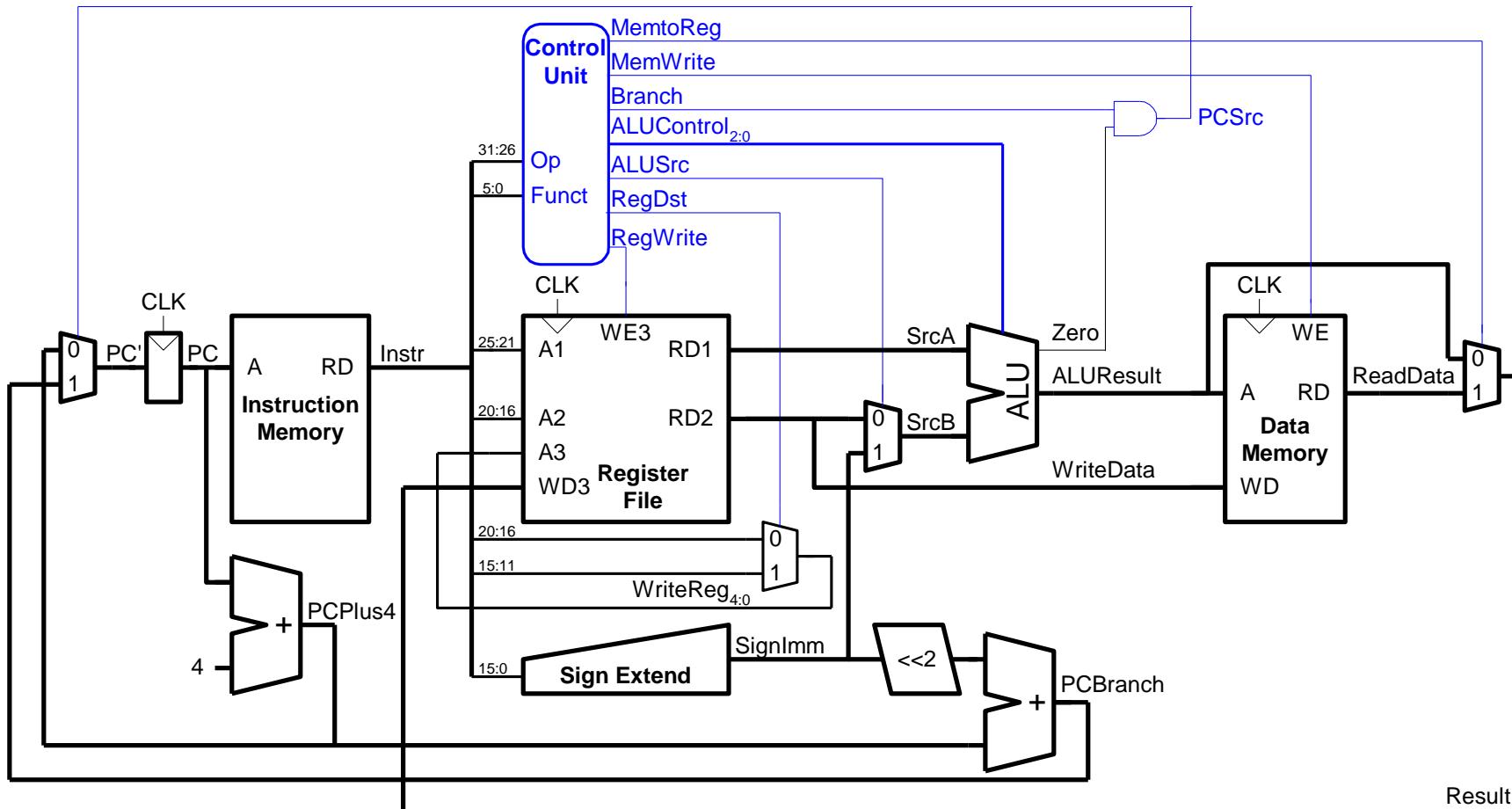
Hlavný dekodér riadiacej časti procesora

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	0	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

Spracovanie inštrukcie or



Podpora pre inštrukciu addi



Nevyžaduje úpravu smerovania toku dát/operandov

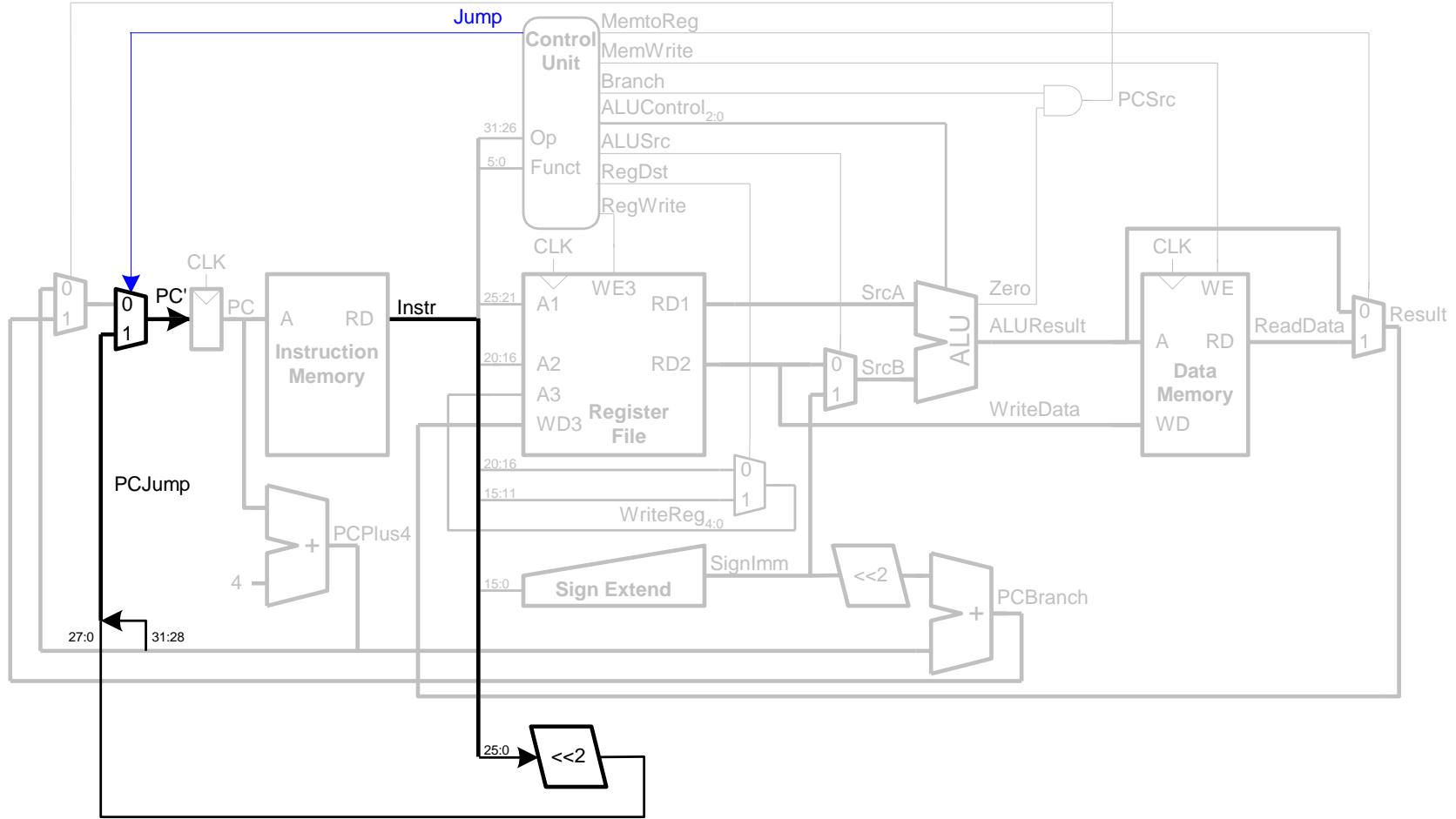
Riadiaca jednotka: addi

Inštrukcia	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000							

Riadiaca jednotka: addi

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000	1	0	1	0	0	0	00

Rozšírenie funkcionality o inštrukciu „j“



Hlavný dekodér riadiacej časti procesora

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
j	000100								

Hlavný dekodér riadiacej časti procesora

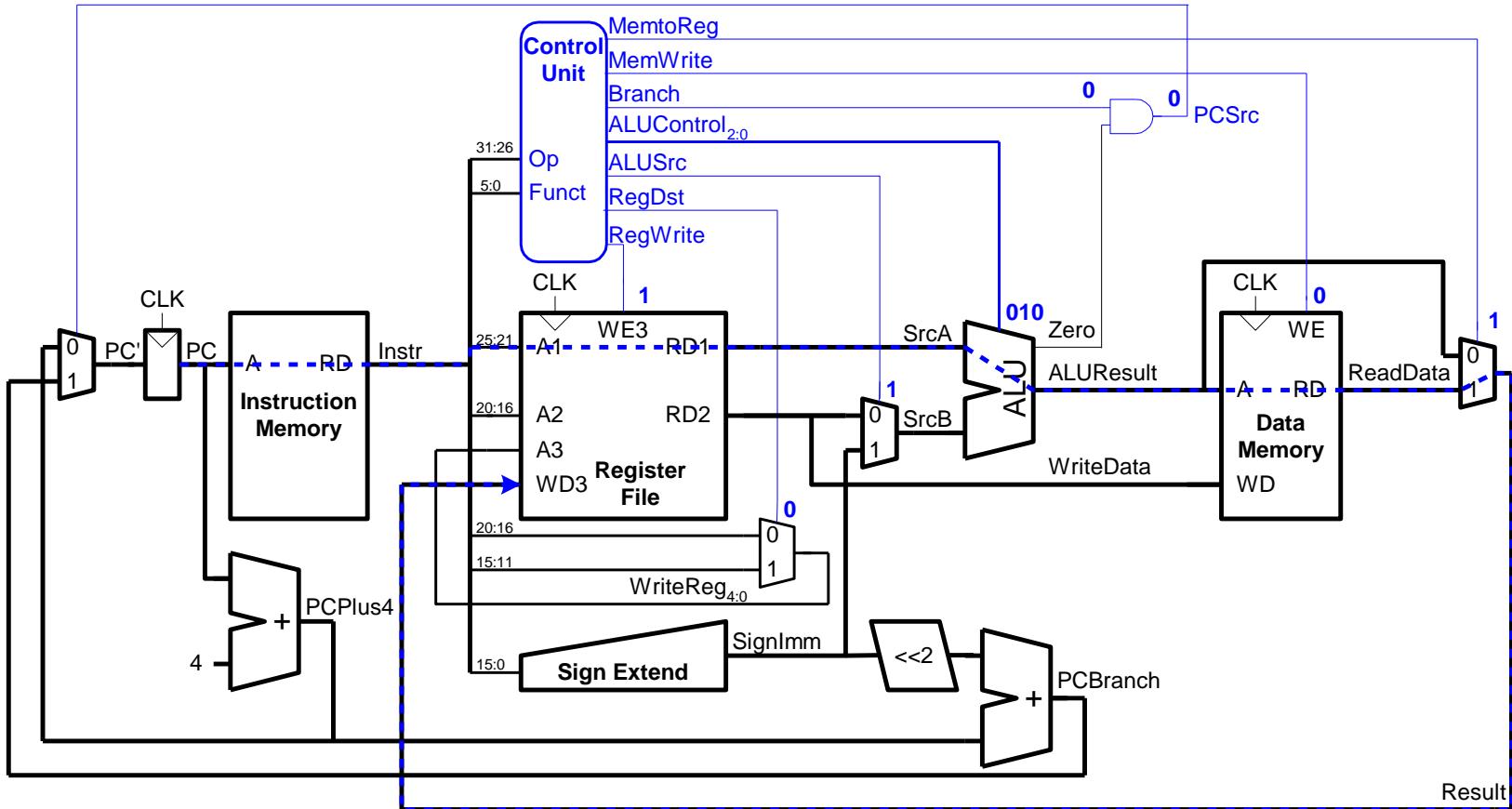
Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
j	000100	0	X	X	X	0	X	XX	1

Revízia: Odozva procesora

Čas odozvy

$$\begin{aligned} \text{čas odozvy} &= (\#\text{instrukcií})(\text{cykly/inštrukcia})(\text{sekundy/cyklus}) \\ &= \# \text{ inštrukcie} \times \text{CPI} \times T_C \end{aligned}$$

Výkon jednocyklového procesora



T_C je limitovaný najdlhšou trasou (lw)

Výkon jednocyklového procesora

- Kritická cesta:

$$T_c = t_{pcq_PC} + t_{\text{mem}} + \max(t_{RFread}, t_{sext} + t_{\text{mux}}) + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RFsetup}$$

- Vo väčšine prípadov:

- pamäť, ALU, súbor registrov

- $T_c = t_{pcq_PC} + 2t_{\text{mem}} + t_{RFread} + t_{\text{mux}} + t_{\text{ALU}} + t_{RFsetup}$

Príklad

Prvok	Parameter	Oneskorenie (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\text{read}}$	150
Register file setup	$t_{RF\text{setup}}$	20

$$T_c = ?$$

Príklad

Prvok	Parameter	Oneskorenia (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\text{read}}$	150
Register file setup	$t_{RF\text{setup}}$	20

$$\begin{aligned}T_c &= t_{pcq_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}} \\&= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps} \\&= 925 \text{ ps}\end{aligned}$$

Príklad

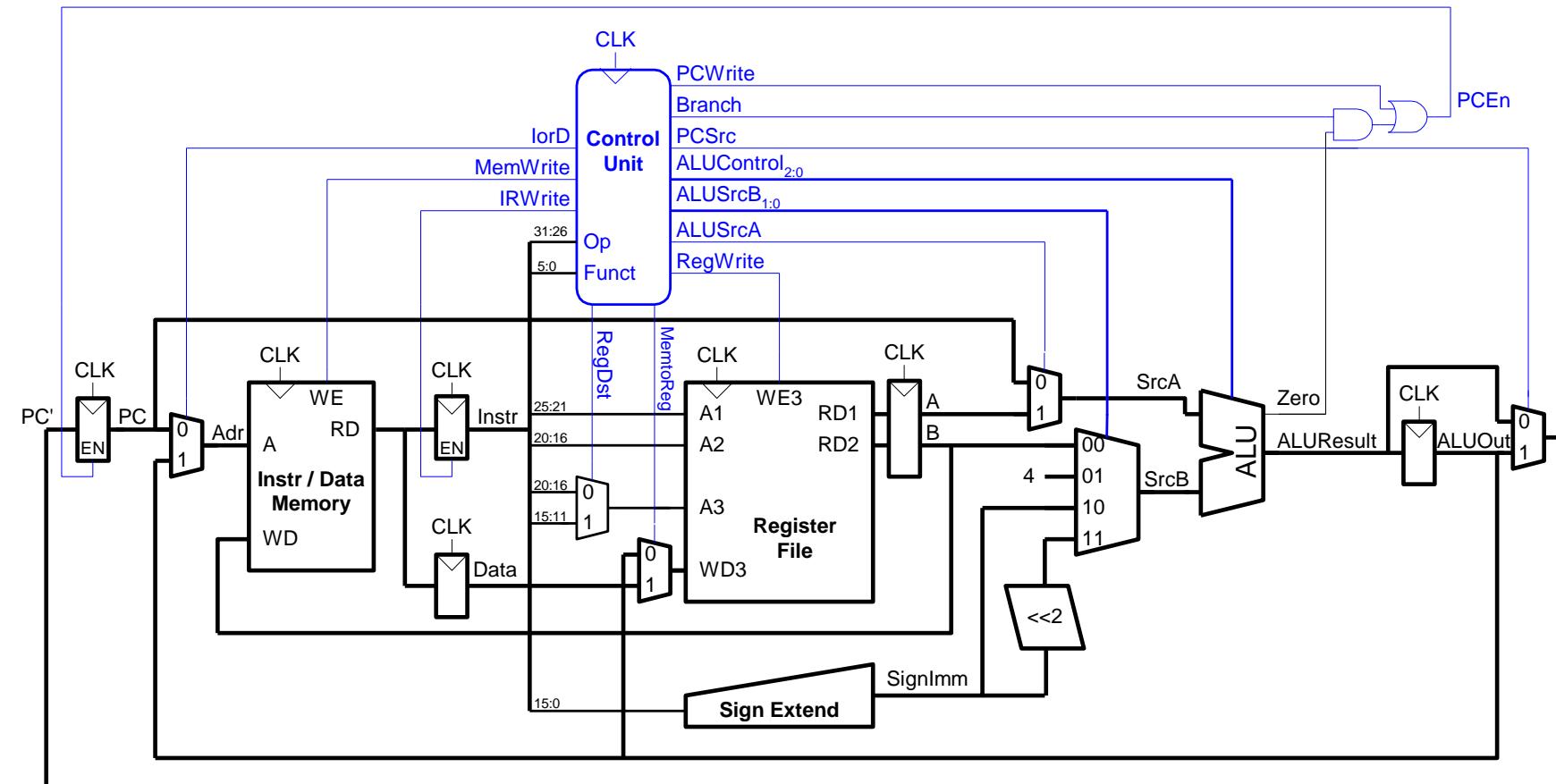
Program so 100 miliardami inštrukcií:

$$\begin{aligned}\text{Čas odozvy} &= \# \text{ inštrukcií} \times \text{CPI} \times T_C \\ &= (100 \times 10^9)(1)(925 \times 10^{-12} \text{ s}) \\ &= \mathbf{92.5 \text{ sekúnd}}\end{aligned}$$

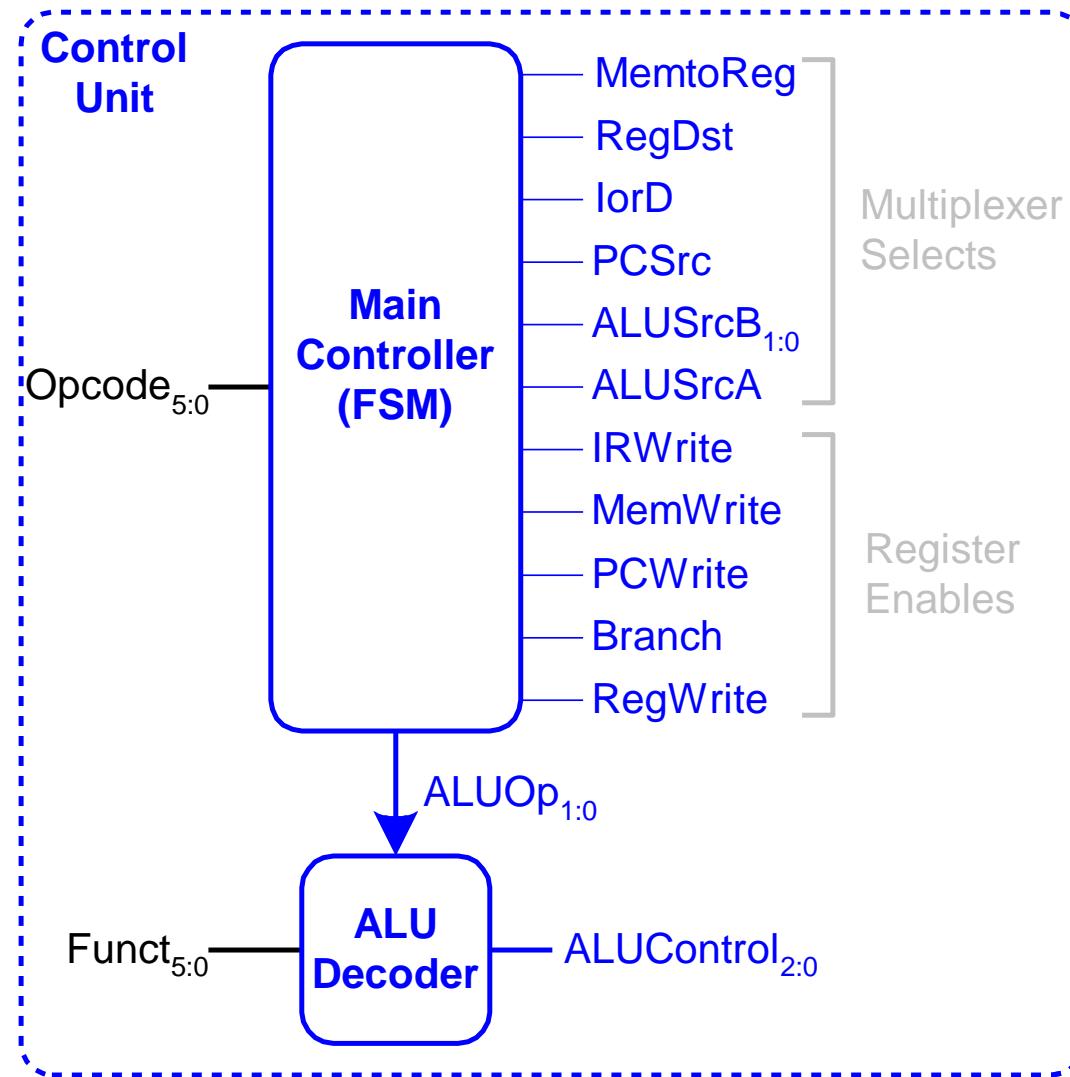
Viaccyklový MIPS procesor

- **Jednocyklový:**
 - + jednoduchý
 - Čas vykonania programu je daná časom vykonania pamäťovej inštrukcie (lw)
 - 2 sčítačky/ALU & 2 pamäte
- **Viaccyklový:**
 - + vyššia taktovacia frekvencia
 - + jednoduchšie inštrukcie vyžadujú kratší čas
 - + znovupoužiteľnosť niektorých funkčných jednotiek počas realizácie inštrukcie
 - zložitejšie riadenie
- **Podobný postup: návrh smerovania toku dát/operandov & smerovania riadiacich signálov**

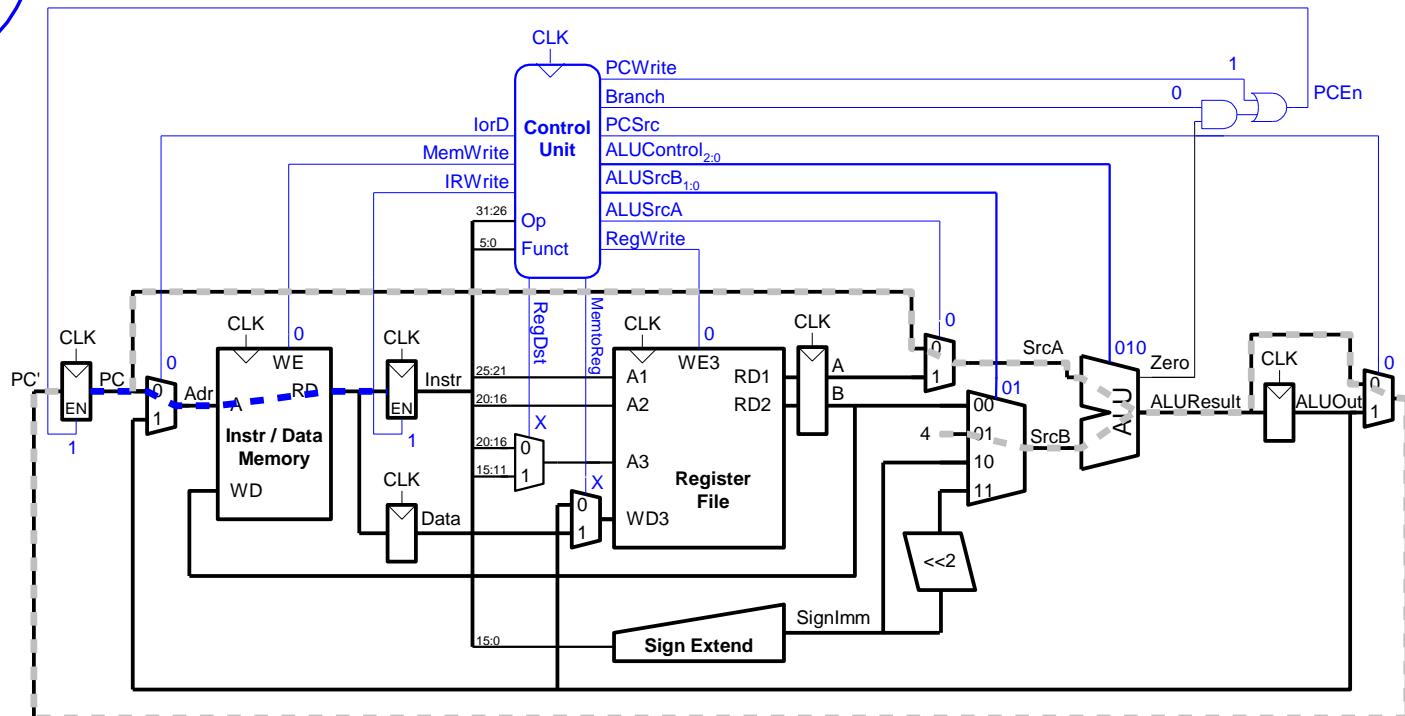
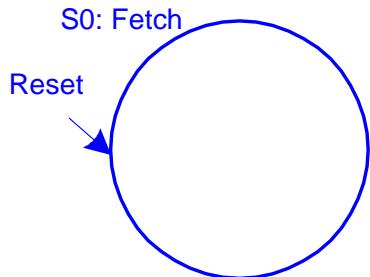
Viaccyklový MIPS procesor



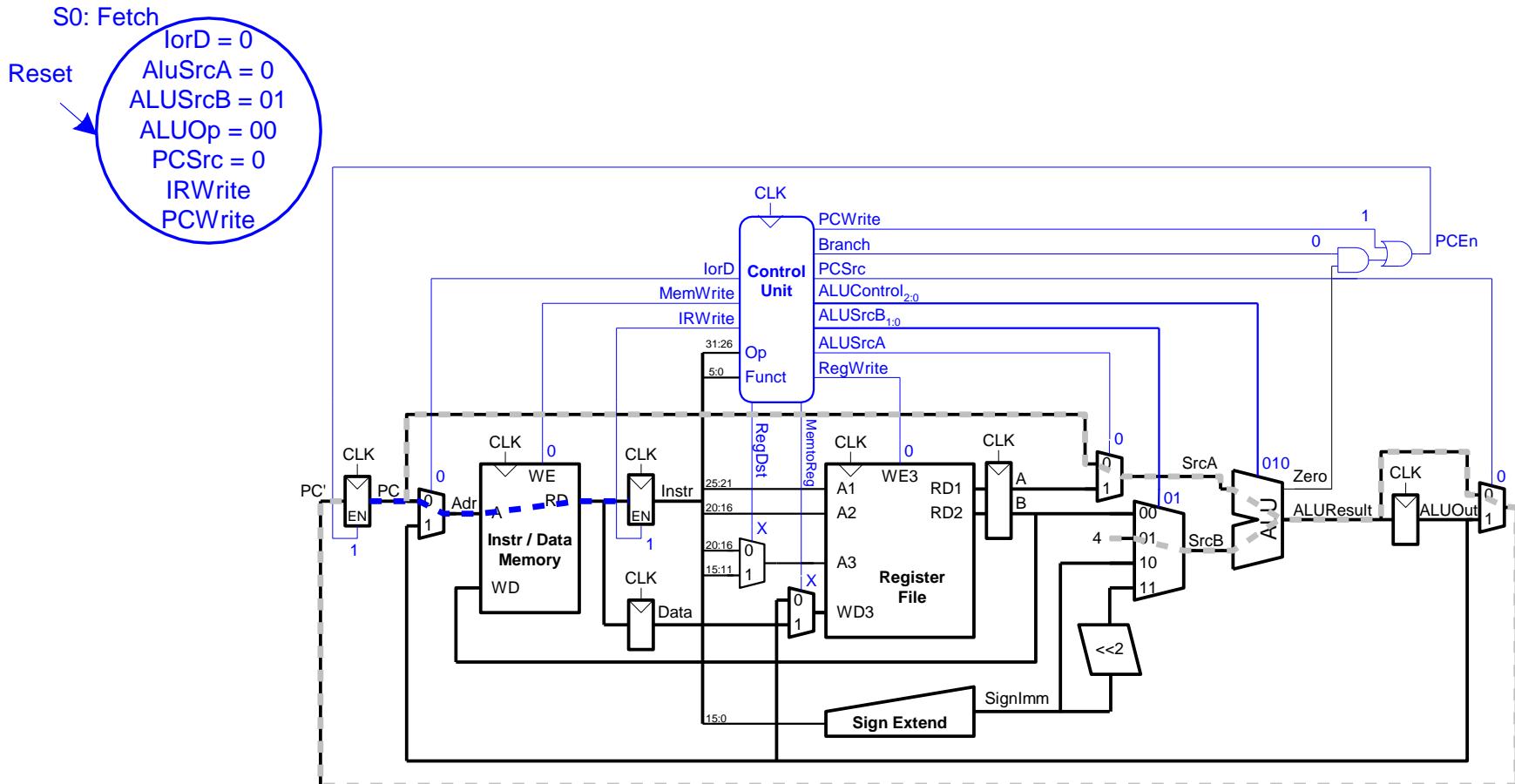
Riadiaca jednotka viaccyklového MIPS procesora



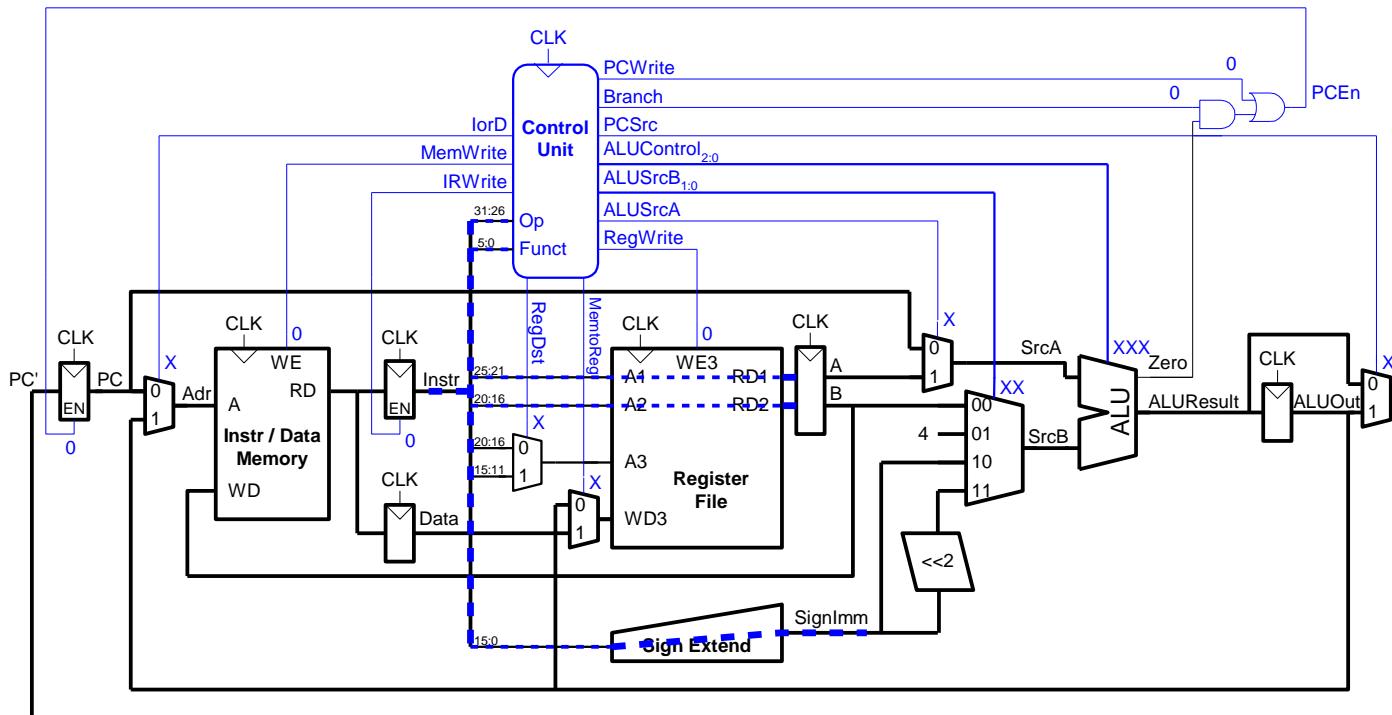
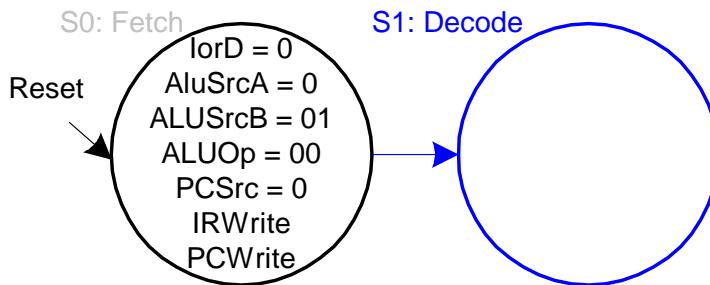
Konečnostavový automat: čítanie inštrukcie



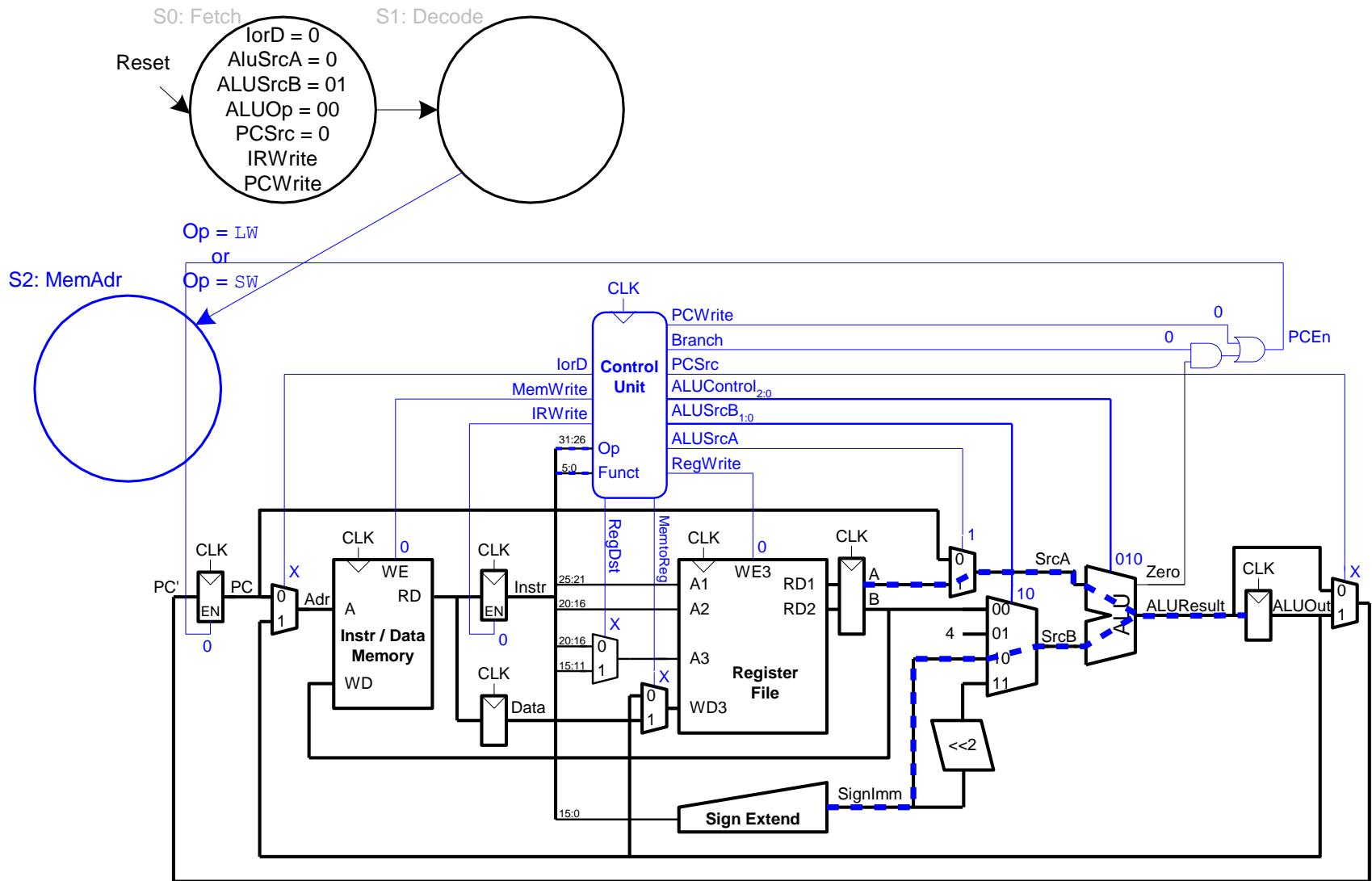
Konečnostavový automat: čítanie inštrukcie



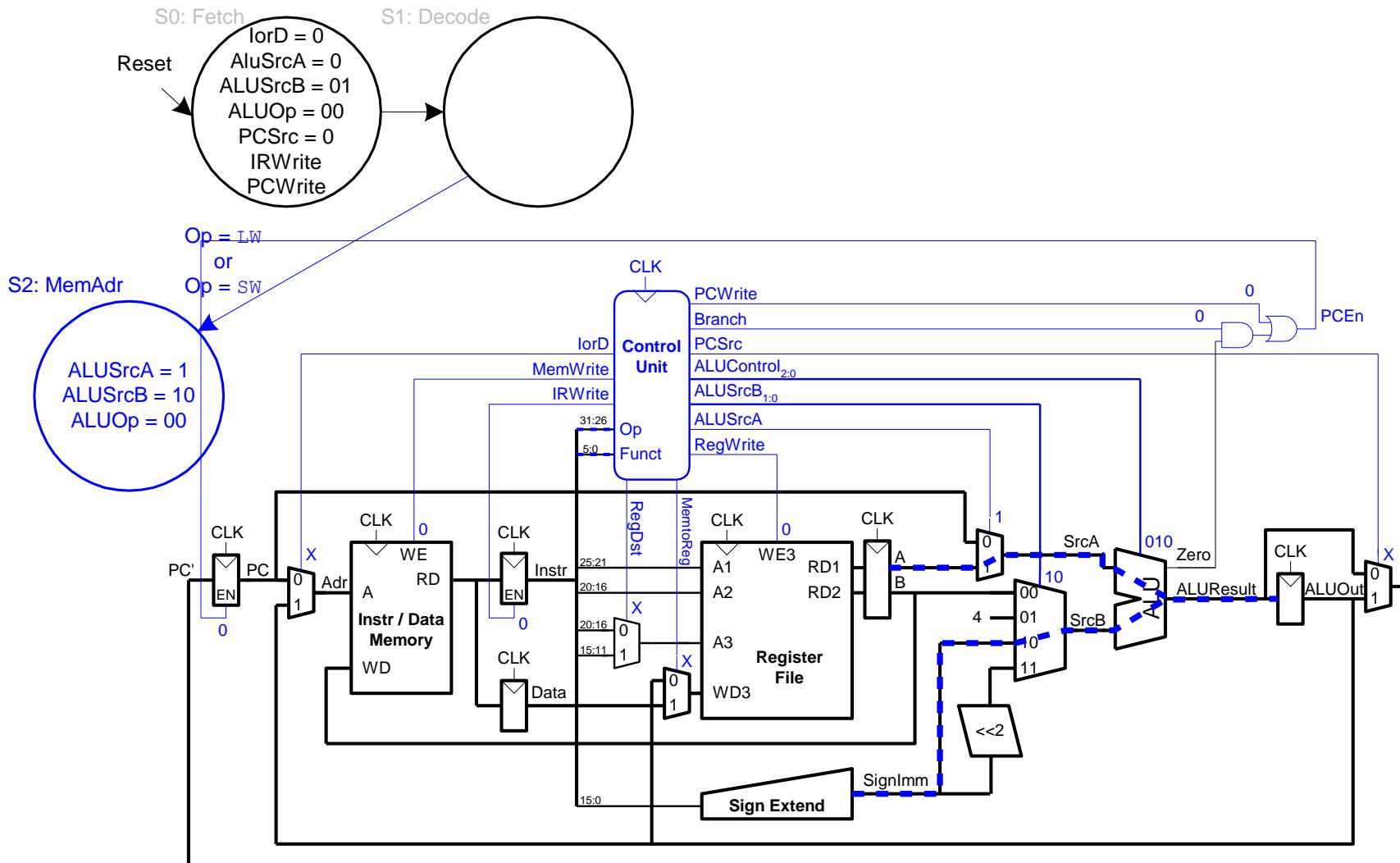
Konečnostavový automat: dekódovanie



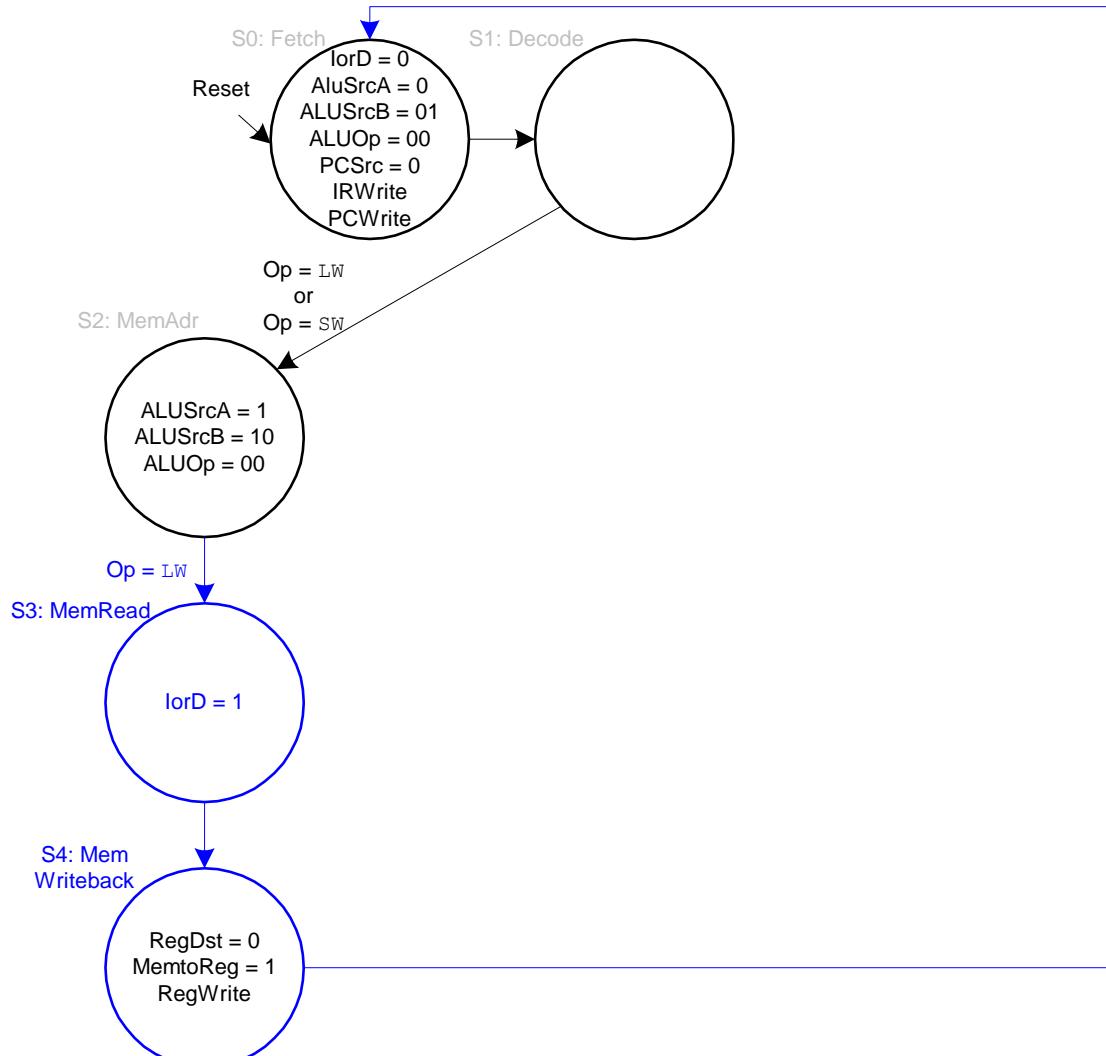
Konečnostavový automat: výpočet adresy



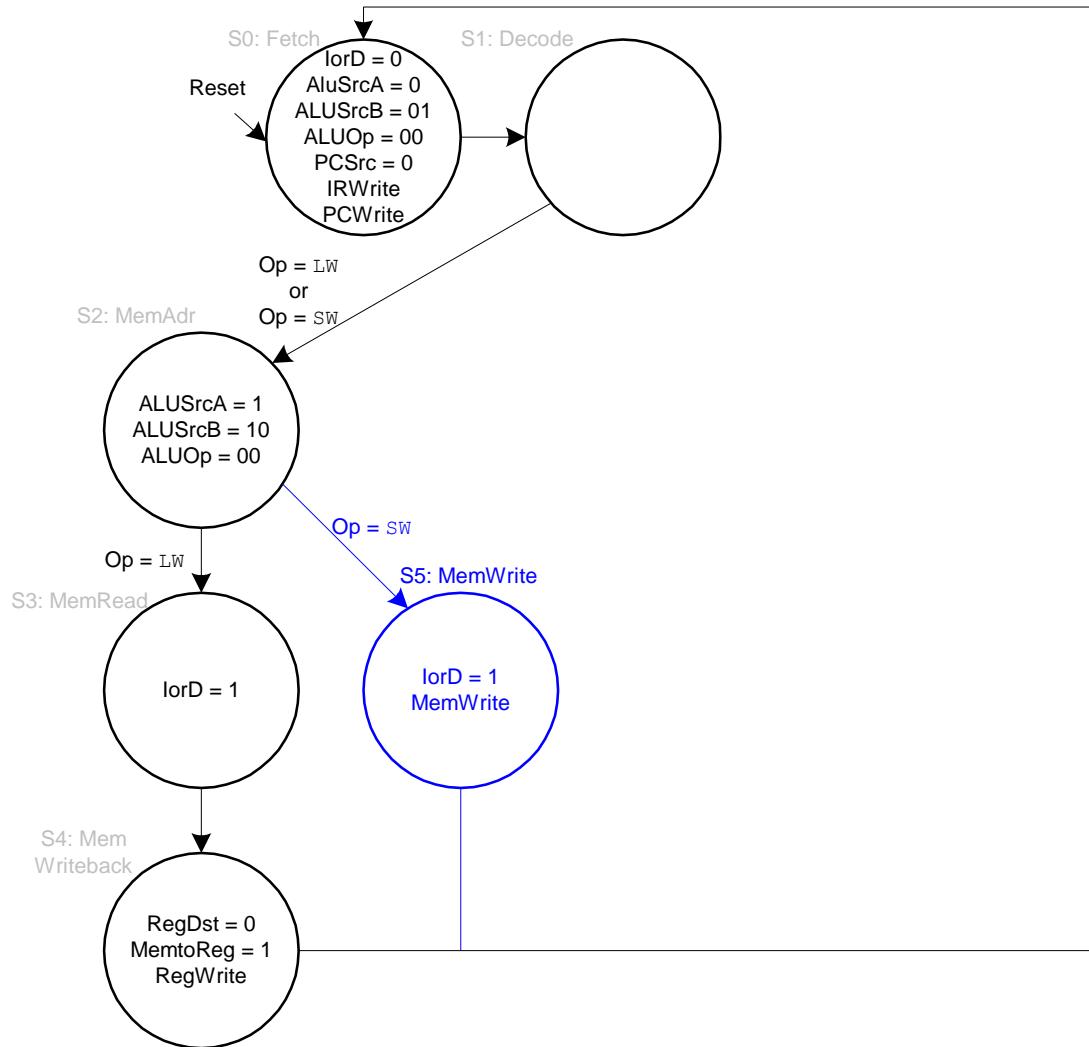
Konečnostavový automat: výpočet adresy



Konečnostavový automat: spracovanie lw

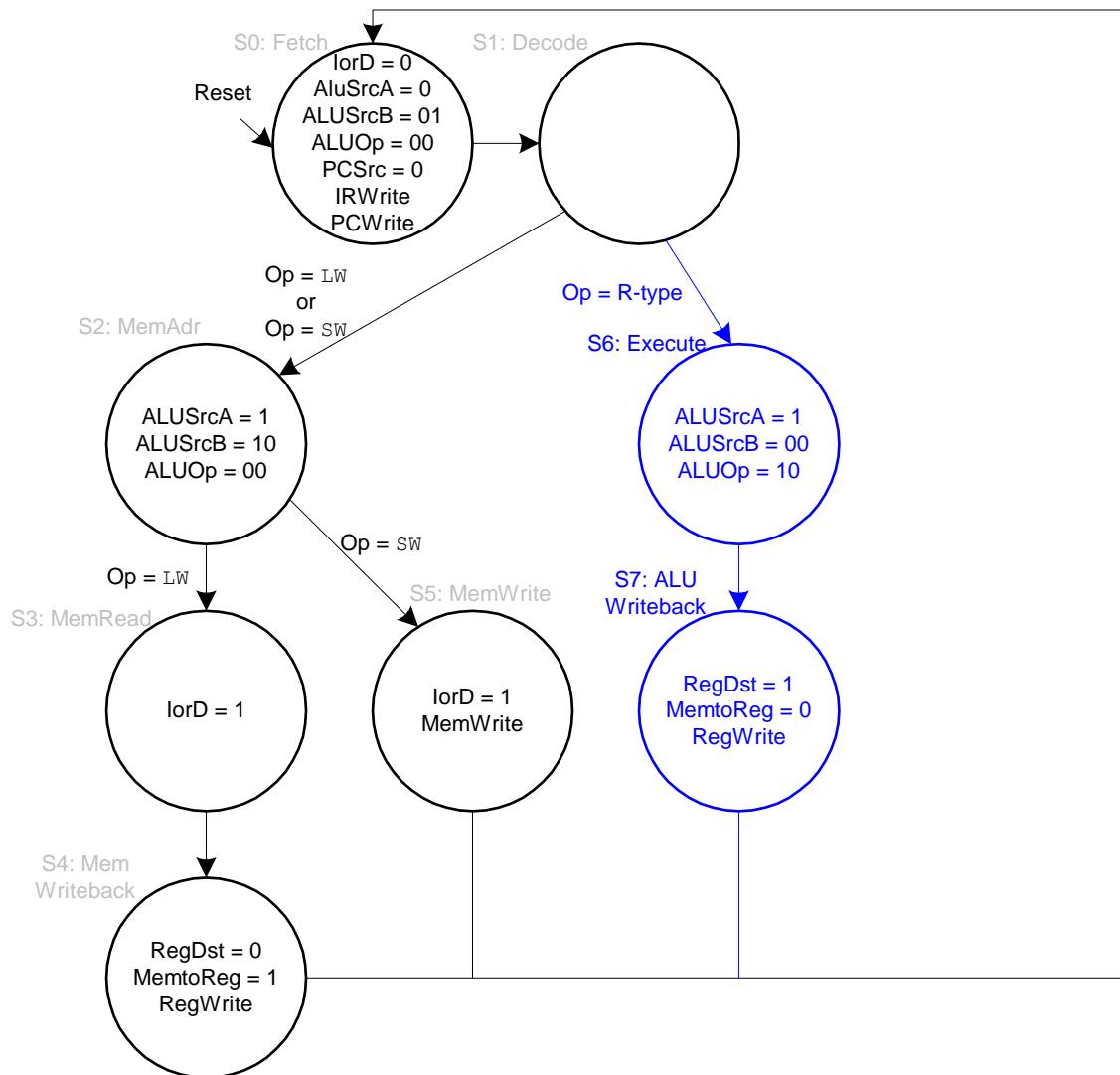


Konečnostavový automat: spracovanie sw



Konečnostavový automat:

spracovanie inštrukcií typu R



Výkon viaccyklového procesora

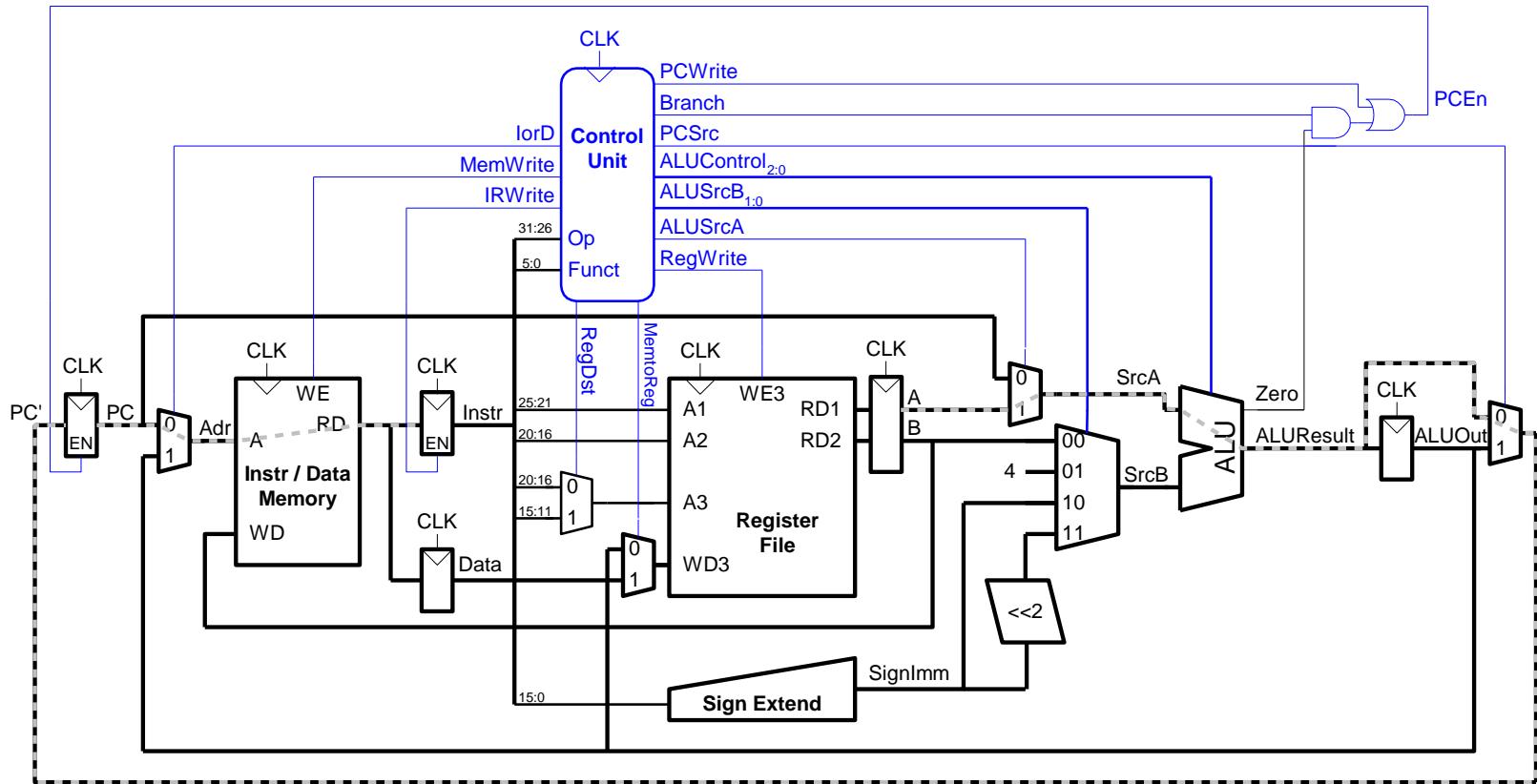
- Počet taktov na realizáciu inštrukcie závisí od typu inštrukcie:
 - 3 takty: beq, j
 - 4 takty: inštrukcie typu R, sw, addi
 - 5 taktov: lw
- CPI sa počíta ako vážený priemer
- SPECINT2000 benchmark:
 - 25% čítanie z pamäte
 - 10% zápis do pamäte
 - 11% vetvenia
 - 2% nepodmienené skoky
 - 52% inštrukcie typu R

$$\text{CPI} = (0.11 + 0.2)(3) + (0.52 + 0.10)(4) + (0.25)(5) = \mathbf{4.12}$$

Výkon viaccyklového procesora

Kritická cesta:

$$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$



Príklad

Element	Parameter	Oneskorenie (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\text{read}}$	150
Register file setup	$t_{RF\text{setup}}$	20

$$T_c = ?$$

Príklad

Element	Parameter	Oneskorenie (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\text{read}}$	150
Register file setup	$t_{RF\text{setup}}$	20

$$\begin{aligned}T_c &= t_{pcq_PC} + t_{\text{mux}} + \max(t_{\text{ALU}} + t_{\text{mux}}, t_{\text{mem}}) + t_{\text{setup}} \\&= t_{pcq_PC} + t_{\text{mux}} + t_{\text{mem}} + t_{\text{setup}} \\&= [30 + 25 + 250 + 20] \text{ ps} \\&= \mathbf{325 \text{ ps}}\end{aligned}$$

Jednocyklový vs viaccyklový MIPS

- Nech je daný program s počtom inštrukcií 100 miliárd. Program sa má vykonať na viaccyklovom MIPS procesore s parametrami
 - CPI = 4.12
 - $T_c = 325 \text{ ps}$

Čas odozvy =

Jednocyklový vs viaccyklový MIPS

- Nech je daný program s počtom inštrukcií 10^{11} . Program sa má vykonáť na viaccyklovom MIPS procesore s parametrami
 - CPI = 4.12
 - $T_c = 325 \text{ ps}$

$$\begin{aligned}\text{Čas odozvy} &= (\# \text{ inštrukcií}) \times \text{CPI} \times T_c \\ &= (100 \times 10^9)(4.12)(325 \times 10^{-12}) \\ &= 133,9 \text{ sekúnd}\end{aligned}$$

- Vykonávanie trvá **dlhšie** než v prípade jednocyklového procesora (92.5 sekúnd). Prečo?

Jednocyklový vs viaccyklový MIPS

100 miliárd inštrukcií

$$\begin{aligned}\text{Čas odozvy} &= (\# \text{ inštrukcií}) \times \text{CPI} \times T_c \\ &= (100 \times 10^9)(4.12)(325 \times 10^{-12}) \\ &= \mathbf{133.9 \text{ sekúnd}}\end{aligned}$$

Vykonávanie trvá **dlhšie** než v prípade jednocyklového procesora (92.5 sekúnd).
Prečo?

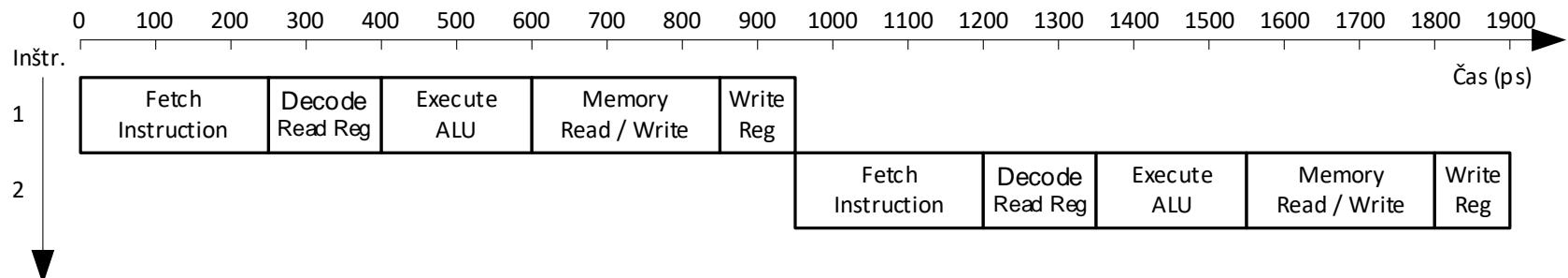
- Rôzne inštrukcie rôzna doba vykonania
- Extra doba na spracovanie ($t_{pcq} + t_{\text{setup}} = 50 \text{ ps}$)

Prúdový MIPS procesor

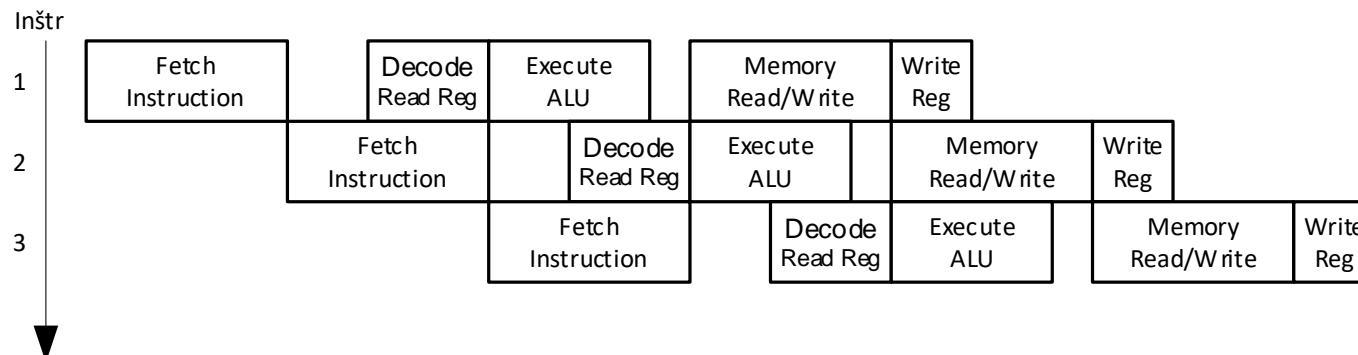
- Temporálny paralelizmus
- Rozdel' vykonanie inštrukcie v jednocyklovom procesore na 5 fáz:
 - Čítanie inštrukcie / Fetch
 - Dekódovanie inštrukcie / Decode
 - Vykonanie inštrukcie / Execute
 - Pamätanie výsledku / Memory
 - Zápis do pamäte / Writeback
- Pridaj záchytné registre medzi jednotlivé stupne zretázenia

Jednocyklový vs. prúdový MIPS

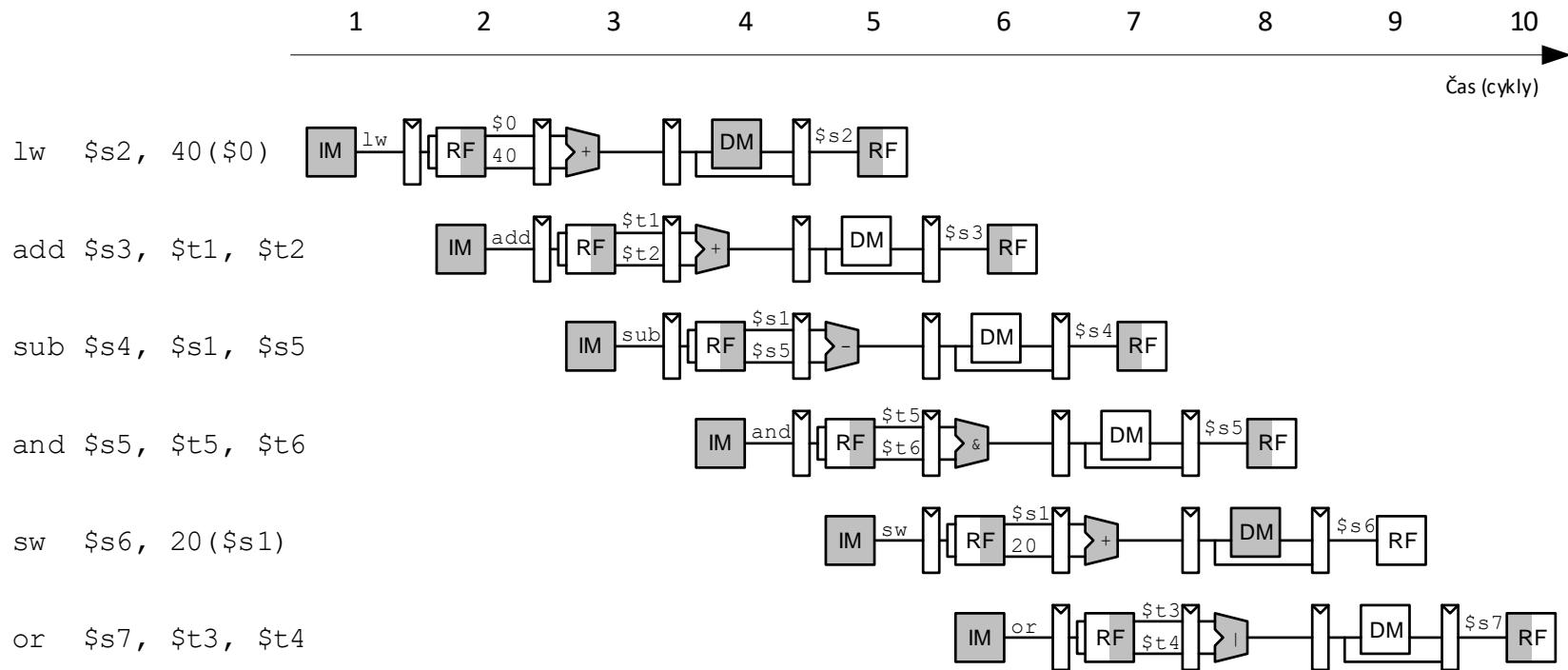
Jednocyklový MIPS



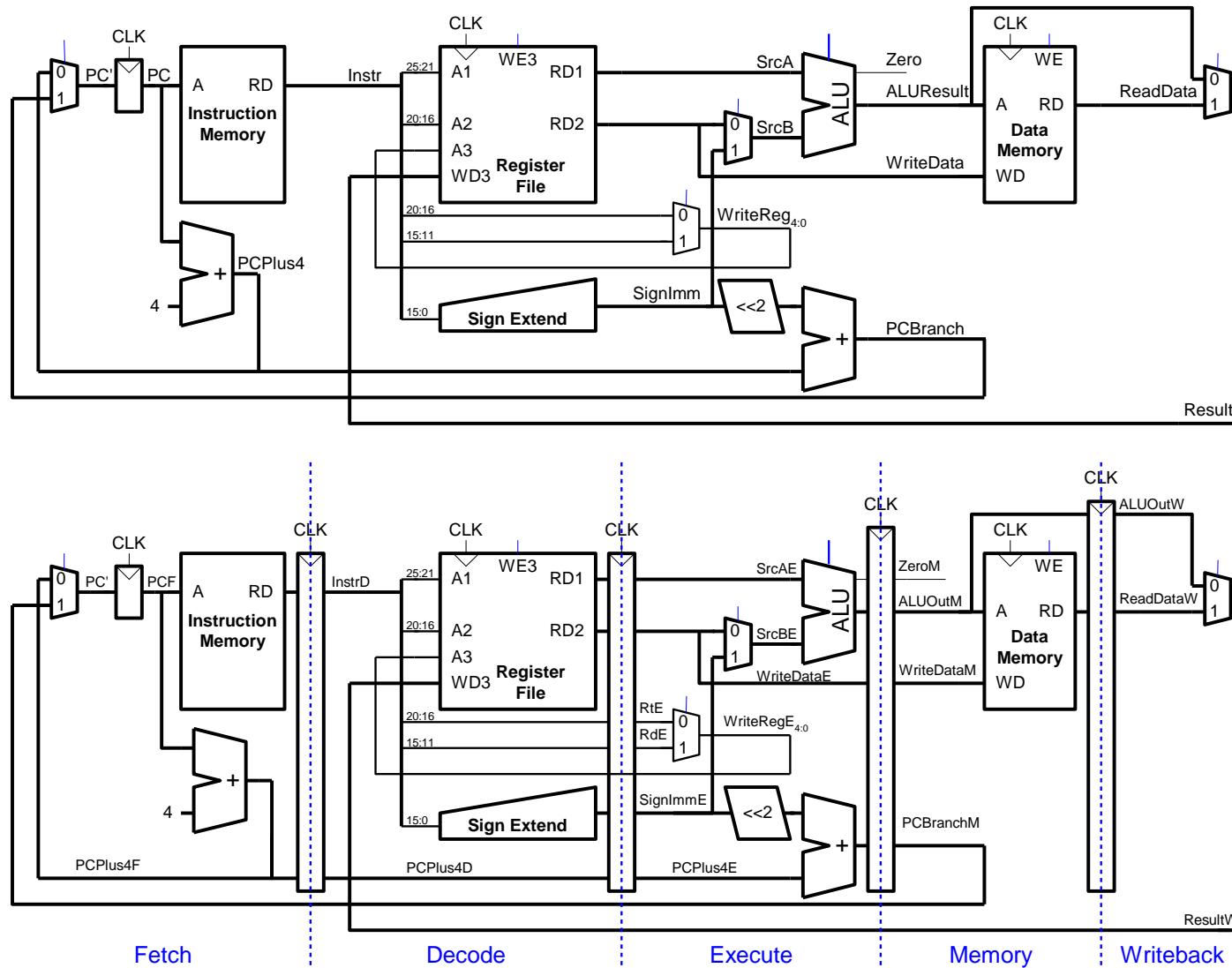
Zretežený MIPS



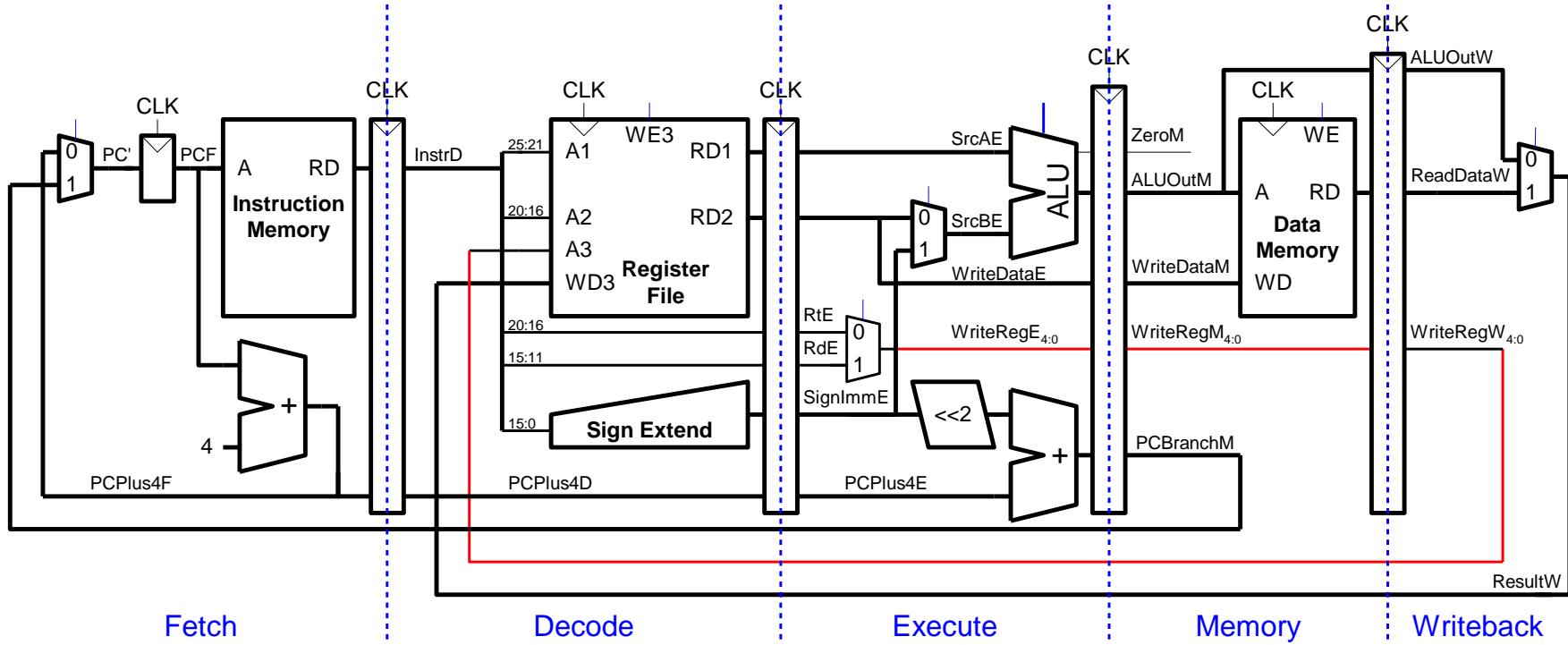
Zreteženie



Jednocyklový & prúdový MIPS

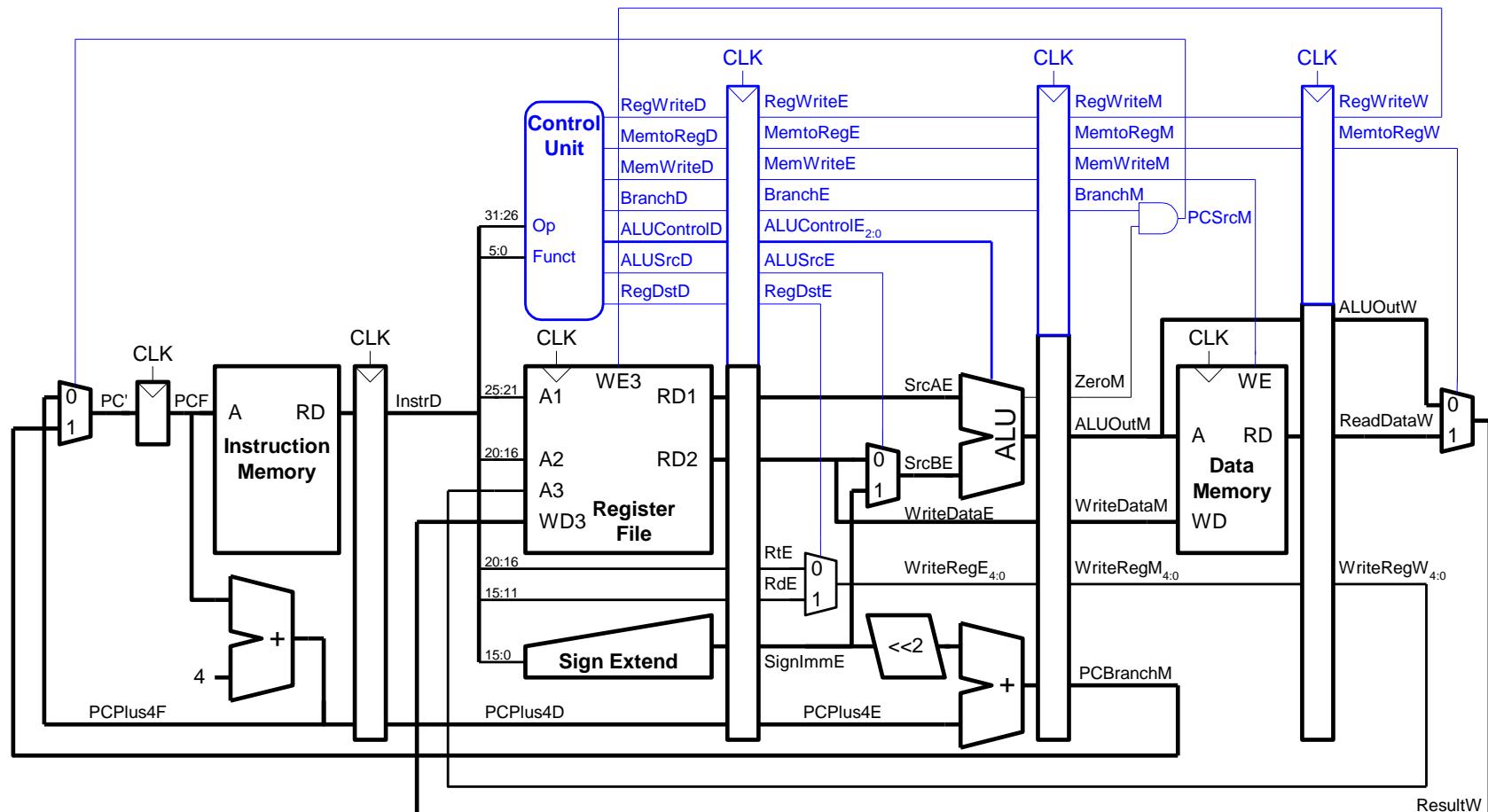


Úprava prúdového MIPS



Spracovanie WriteReg a Result vyžaduje úpravu v toku dát

Riadenie prúdového MIPS

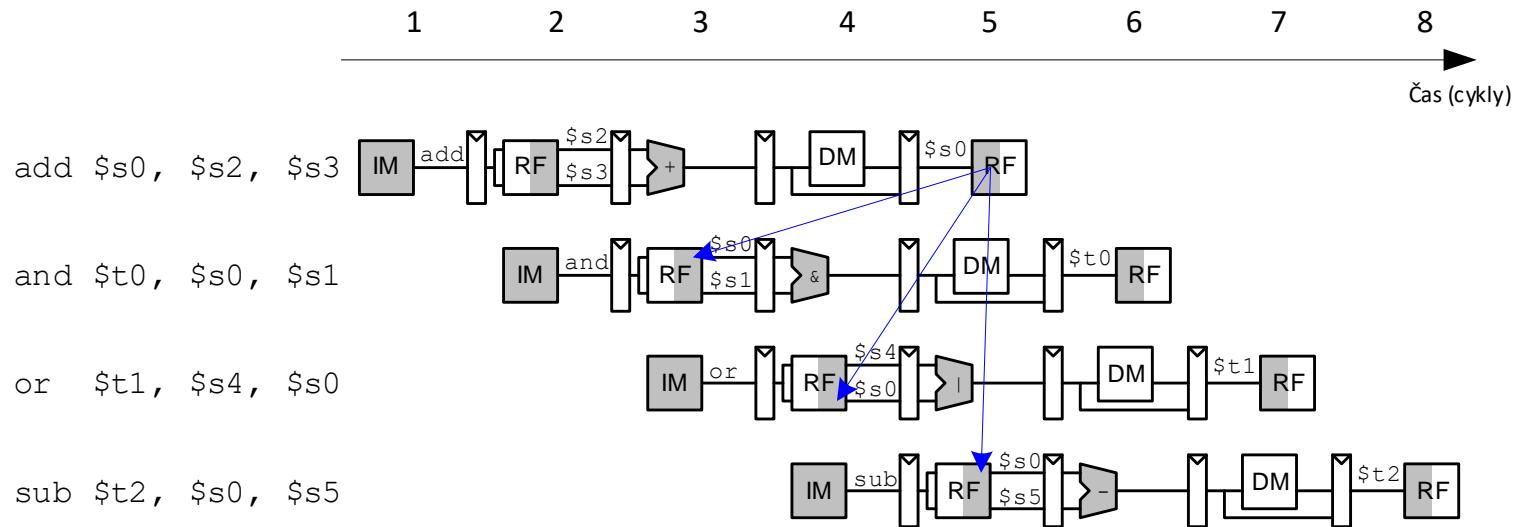


- **RJ prúdového MIPS = RJ jednocyklového MIPS**
- **Riadenie prispôsobené zreteženému spracovaniu**

Hazardy prúdového spracovania

- Ak vykonanie inštrukcie je závislé od inštrukcie, ktorá ešte nebola dokončená
- Typy:
 - **Údajový hazard:** v dôsledku nedostupnosti operandov pri vykonaní inštrukcií
 - **Hazard riadenia:** v dôsledku vetvení (nie je známa, ktorá je nasledujúca inštrukcia)
 - **Zdrojový (štrukturálny) hazard:** nie sú dostupné zdroje na vykonanie inštrukcie

Údajový hazard

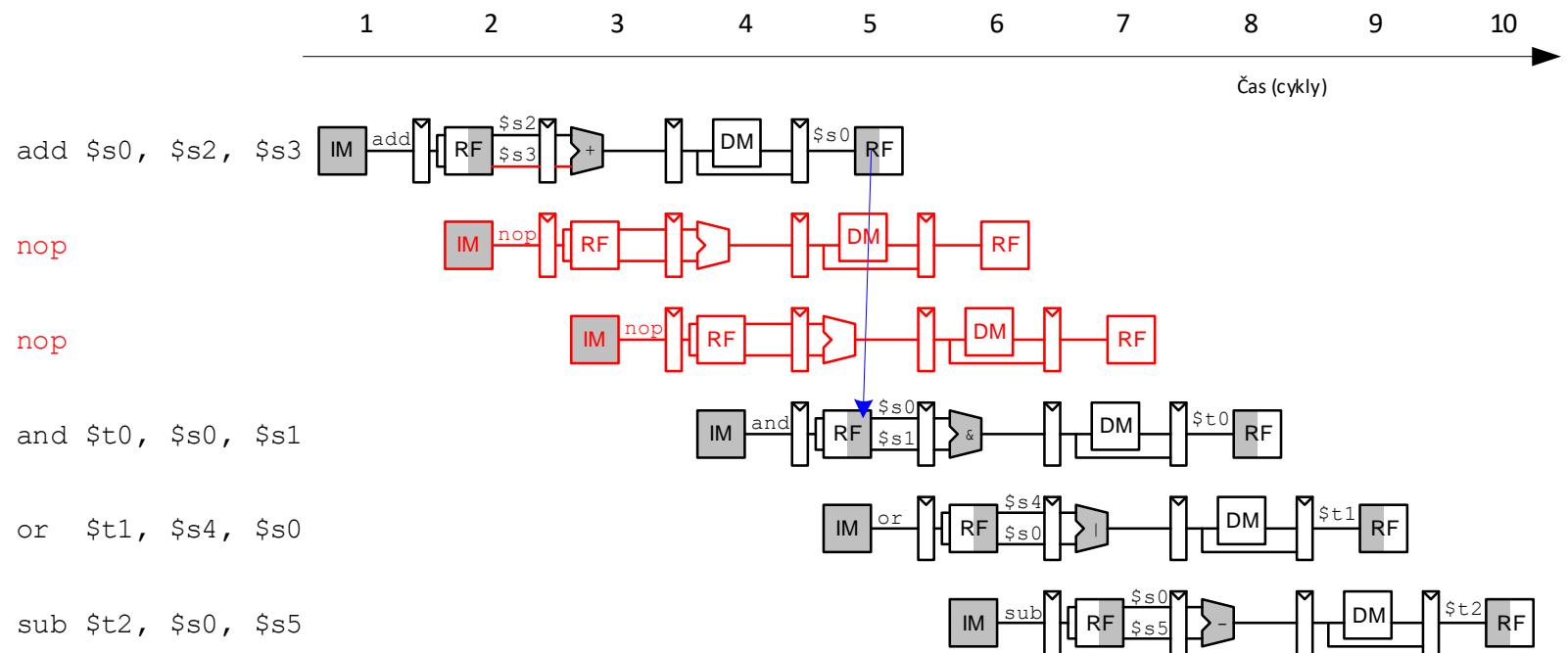


Eliminácia údajového hazardu

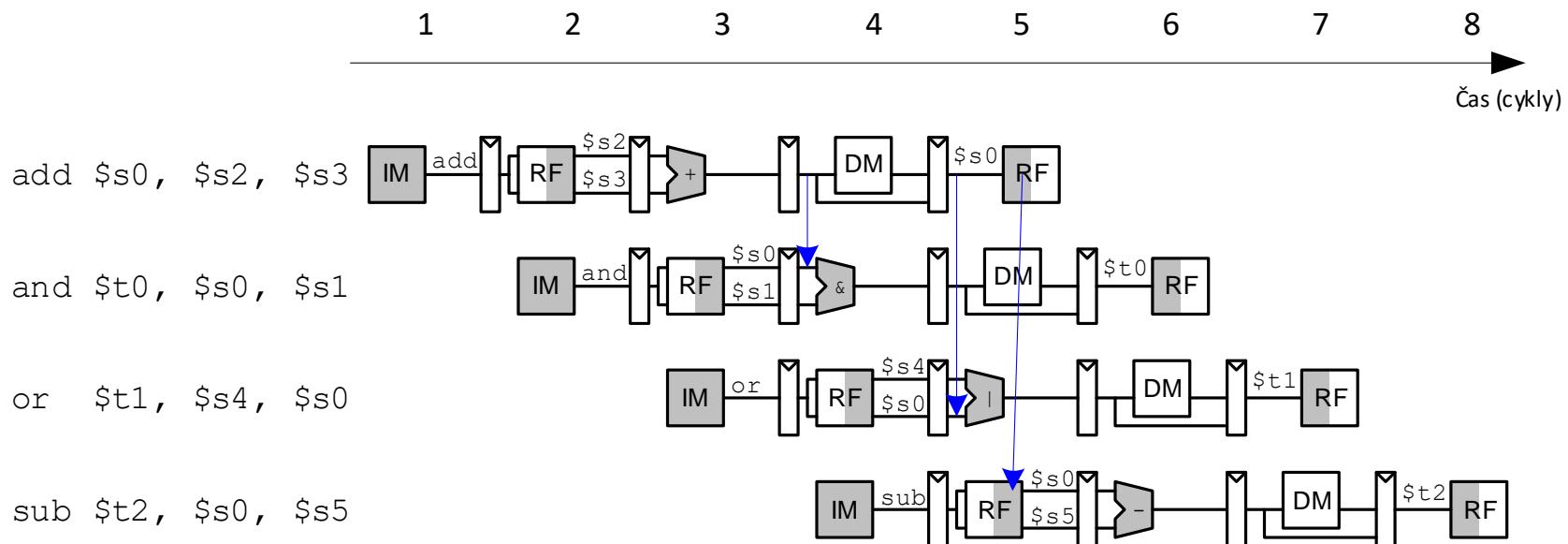
- Softvérové prostriedky
 - Vloženie NOP po inštrukcií, ktorá môže spôsobiť hazard
 - Preusporiadanie inštrukcií
- Hardvérové prostriedky
 - Blokovanie prúdového spracovania generovaním prázdnych cyklov
 - Dopredné generovanie výsledku predchádzajúcej inštrukcie pre jeho použitie ako vstupu pre nasledujúcu inštrukciu

Eliminácia hazardu sw prostriedkami

- Vkladanie NOP
- Preusporiadanie inštrukcií



Spätné šírenie / preposielanie



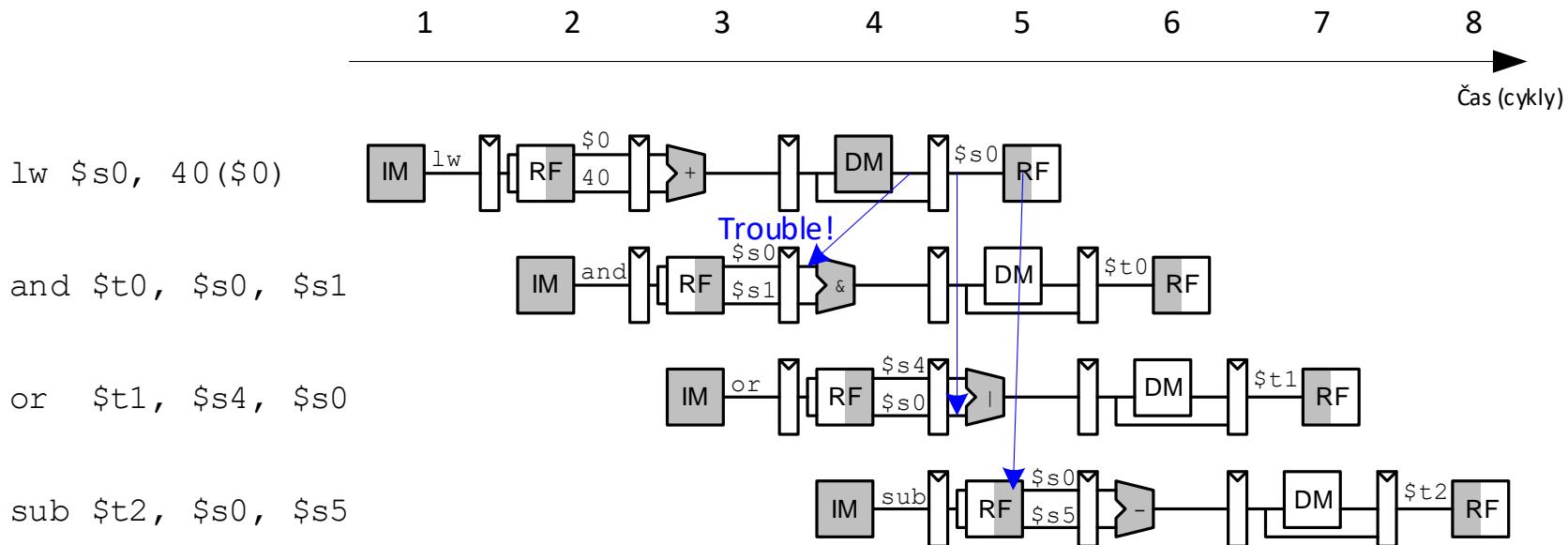
Preposielanie

- Ak výsledok vznikne skôr ako nasledujúca inštrukcia ho naozaj potrebuje je možné tento hazard riešiť preposielaním (forwarding)
- Ak použitý zdrojový register inštrukcie v stupni E zhoduje s cieľovým regisstrom v stupni M alebo WB

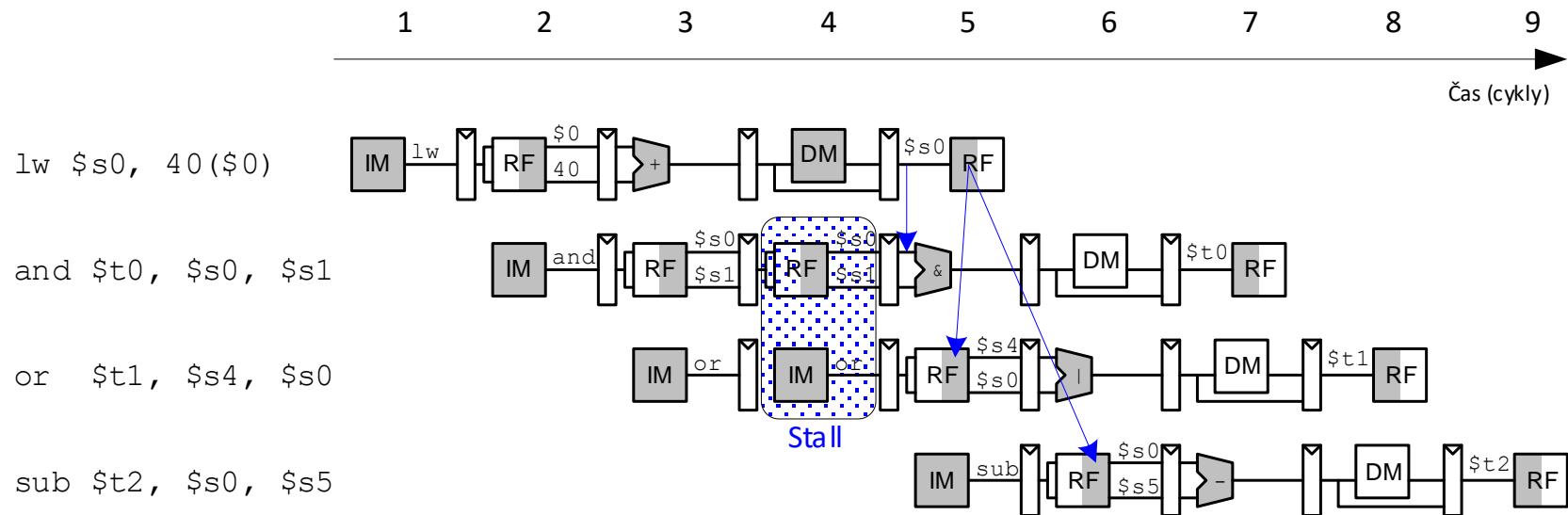
Preposielanie

- Preposielanie údajov do stupňa E(xecute) zo stupňov:
 - M(emory)
 - WB (Writeback)
- Čísla týchto registrov z týchto stupňov sú posielané do hazardnej jednotky (Hazard Unit)
 - Overuje sa či cielový register bude skutočne použitý k zápisu (tj., či sa jedná o cielový register – lw vs. sw) – RegWrite z M a WB

Pozastavenie (stalling)



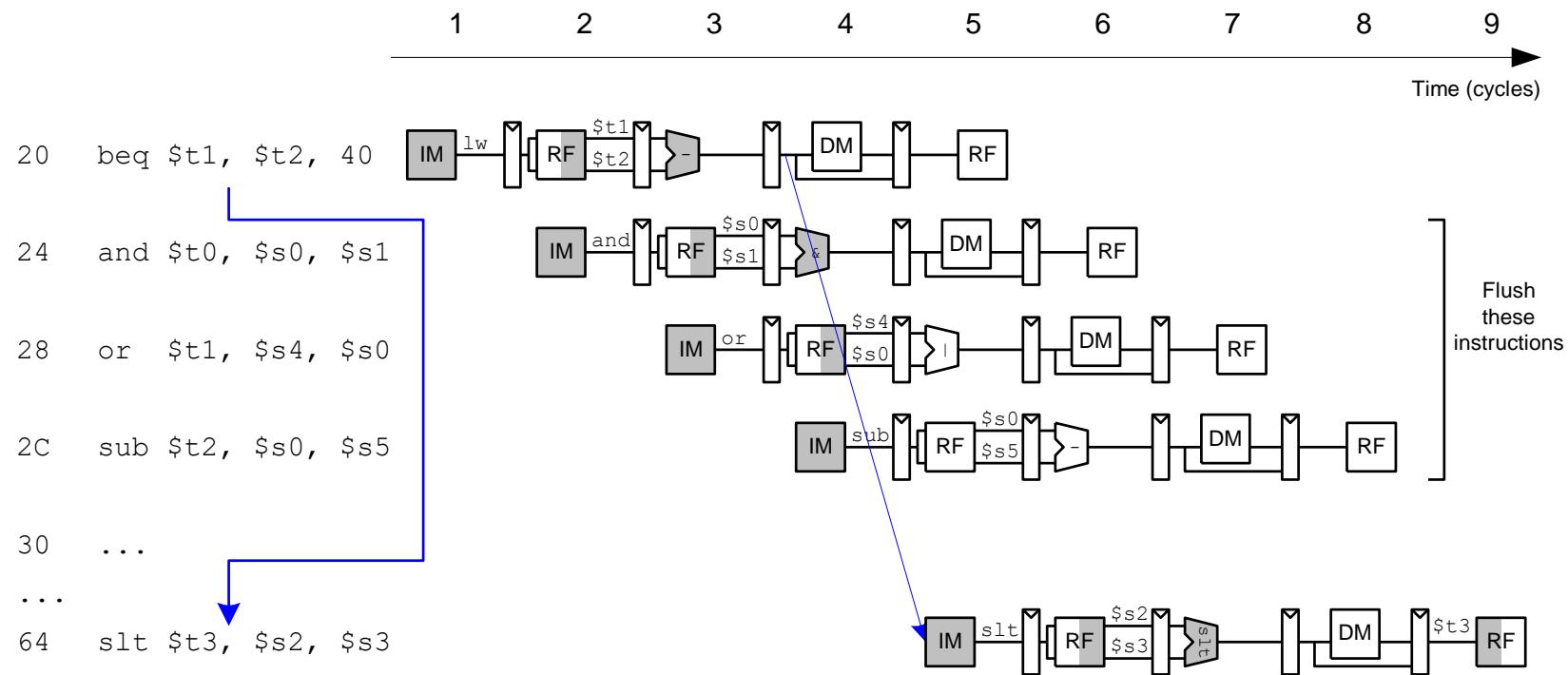
Pozastavenie



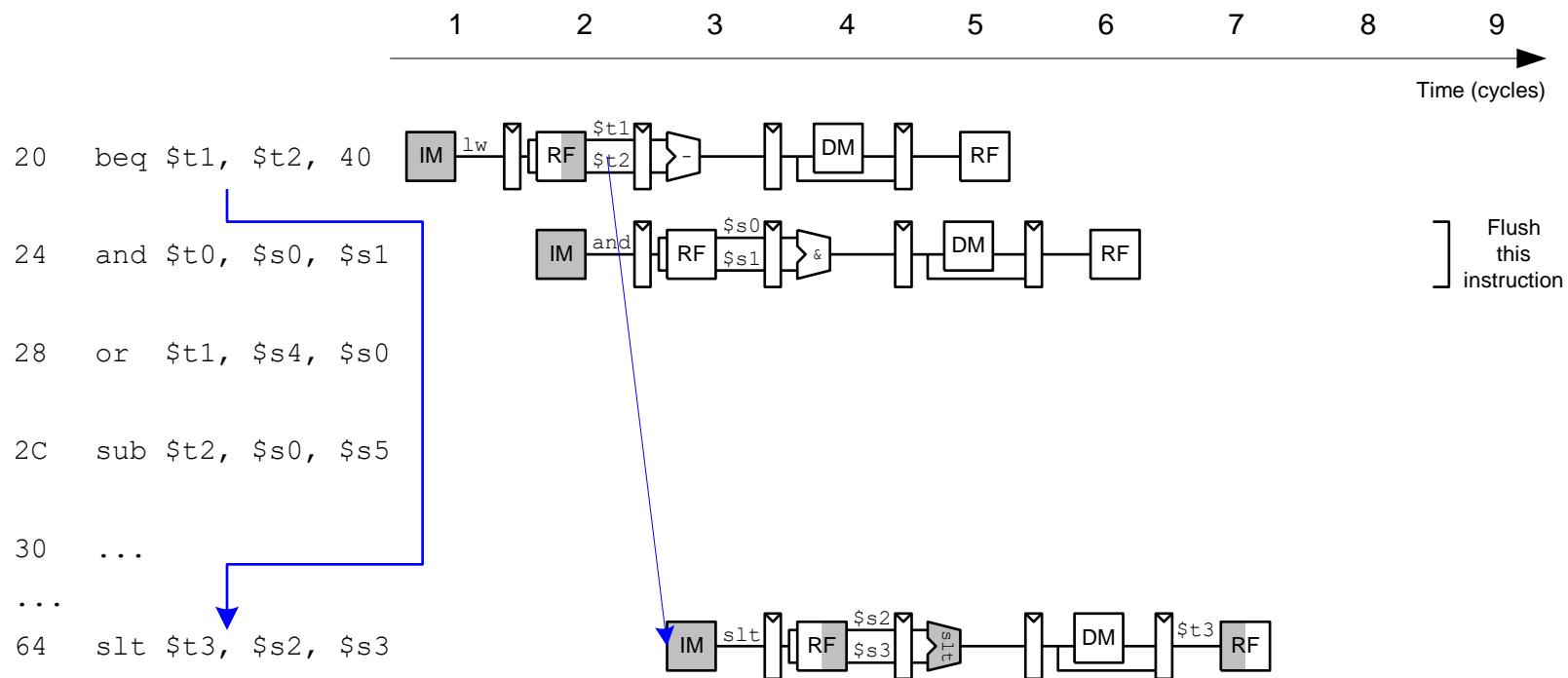
Hazardy riadenie

- **beq:**
 - Výsledok známy až v 4. stupni zreteženia
 - Nasledujúca inštrukcia sa číta ešte pred dokončením vetvenia
- **Pokutovanie nesprávneho odhadu skoku
(branch misprediction penalty)**
 - Vyradenie inštrukcií z procesu spracovania ak sa skok uskutoční
 - Predikovanie skoku

Hazardy riadenie



Predikovanie výsledku skoku



Predikcia vetvenia

- Zistuje sa či sa uskutoční alebo nie skok do druhej vetvy
 - Skoky previazané s cyklami (for, while) vo väčšine prípadov sa uskutočnia
 - Analyzuje sa pri tom história skokov
- Dobrá predikcia minimalizuje nutnosť vyradenia načítaných inštrukcií z prúdovej funkčnej jednotky (vyprázdenie prúdovej funkčnej jednotky)

Príklad

- SPECINT2000 benchmark:
 - 25% čítanie
 - 10% zápis
 - 11% vetvenia
 - 2% skoky
 - 52% inštrukcie typu R
- Nech:
 - 40% výskytu RAW pri čítaní
 - 25% inštrukcií vetvenia sú nesprávne predikované
 - Všetky skokové inštrukcie sa vykonajú
- Aká je priemerná CPI (ACPI)?

Príklad

- SPECINT2000 benchmark:
 - 25% čítanie
 - 10% zápis
 - 11% vetvenia
 - 2% skoky
 - 52% inštrukcie typu R
- Nech:
 - 40% výskytu RAW pri čítaní
 - 25% inštrukcií vetvenia sú nesprávne predikované
 - Všetky skokové inštrukcie sa vykonajú
- Aká je priemerná CPI (ACPI)?
 - Load/Branch CPI = 1 bez pozastavenia, 2 s pozastavením vykonávania
 - $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$
 - $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$

$$\begin{aligned} \text{ACPI} &= (0.25)(1.4) + (0.1)(1) + (0.11)(1.25) + (0.02)(2) + (0.52)(1) \\ &= 1.15 \end{aligned}$$

Výkon prúdového MIPS

- Kritická cesta:

$$T_c = \max \{$$

$$t_{pcq} + t_{\text{mem}} + t_{\text{setup}}$$

$$2(t_{RFread} + t_{\text{mux}} + t_{\text{eq}} + t_{\text{AND}} + t_{\text{mux}} + t_{\text{setup}})$$

$$t_{pcq} + t_{\text{mux}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{setup}}$$

$$t_{pcq} + t_{\text{memwrite}} + t_{\text{setup}}$$

$$2(t_{pcq} + t_{\text{mux}} + t_{RFwrite}) \}$$

Príklad

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20
Equality comparator	t_{eq}	40
AND gate	t_{AND}	15
Memory write	T_{memwrite}	220
Register file write	$t_{RFwrite}$	100 ps

$$\begin{aligned}T_c &= 2(t_{RFread} + t_{\text{mux}} + t_{eq} + t_{\text{AND}} + t_{\text{mux}} + t_{\text{setup}}) \\&= 2[150 + 25 + 40 + 15 + 25 + 20] \text{ ps} = 550 \text{ ps}\end{aligned}$$

Výkon prúdového MIPS

100 miliárd inštrukcií v programe

$$\begin{aligned}\text{Čas odozvy} &= (\# \text{ inštrukcií}) \times \text{CPI} \times T_c \\ &= (100 \times 10^9)(1.15)(550 \times 10^{-12}) \\ &= \mathbf{63 \text{ sekúnd}}\end{aligned}$$

Porovnanie

Procesor	Čas odozvy (sekúnd)	Zrýchlenie (oproti jednocyklovému)
Jedno-cyklový	92.5	1
Viac-cyklový	133	0.70
Prúdový	63	1.47

Spracovanie výnimiek

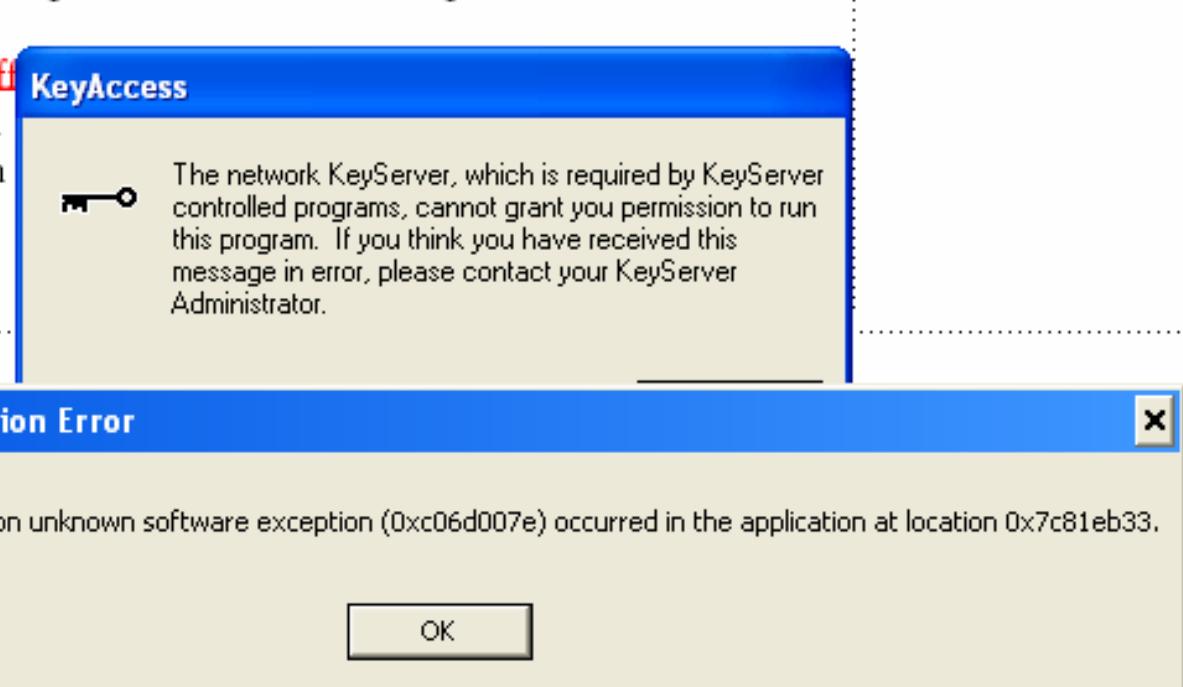
- Obslužná rutina spracovania výnimiek (*exception handler*)
- Dôvody:
 - Hardvérové *prerušenia*, napr. od klávesnice
 - Softvérové *prerušenia*, napr. delenie nulou
- Ak nastane prerušenie:
 - Zaznamená sa (register prerušenia / „cause register“)
 - Skok na obslužnú rutinu (0x80000180)
 - Návrat do programu (EPC register)

Example Exception

sequential circuits.¶

Can we design a spif?

Figure 2.11 shows a inputs, A and B, and on box indicates that it is this case, the function is



words, we say the output Y is a function of the two inputs A and B where the function performed is A OR B.¶

The *implementation* of the combinational circuit is independent of its functionality. Figure 2.1 and Figure 2.2 show two possible implementa-

Registre výnimiek

- Nie sú súčasťou súboru registrov (RF)
 - Register „cause“
 - Zaznamená typ výnimky
 - Súčasťou koprocessora 0, č. registra je 13
 - Register EPC (Exception PC)
 - Adresa inštrukcie ktorá výnimku vyvolala (PC)
 - Súčasťou koprocessora 0, č. registra je 14
- Inštrukcie pre prácu s koprocessorom 0
 - mfc0 \$t0, Cause
 - Obsah registra Cause sa presunie do \$t0

mfc0

010000	00000	\$t0 (8)	Cause (13)	000000000000
31:26	25:21	20:16	15:11	10:0

Dôvody vzniku výnimiek

Výnimka	Cause register
Hardvérové prerušenia	0x00000000
Systémové volanie	0x00000020
Bod prerušenia (breakpoint) / delenie 0	0x00000024
Nedefinovaná inštrukcia	0x00000028
Pretečenie (vyvolané s aritmetickými inštrukciami)	0x00000030

Moderné mikroarchitektúry

- „Hlboké“ prúdové funkčné jednotky
- Špekulatívne vetvenie
- Superskalárne procesory
- Vykonávanie mimo poradia (Out of Order Processors)
- Premenovávanie registrov
- SIMD
- Multivláknové procesory
- Multiprocesory

„Hlboké“ prúdové funkčné jednotky

- 10-20 stupňov
- Počet stupňov sa limituje z dôvodu:
 - Zvyšujúca pravdepodobnosť výskytu hazardu pri zvýšení počtu prúdových stupňov
 - Energetická náročnosť
 - Cena

Špekulatívne vetvenie

- Ideálny prúdový procesor: CPI = 1
- Zlá predikcia zvyšuje CPI
- **Statická predikcia vetvenia:**
 - Kontrola smeru skoku (dopredu alebo späť)
 - Ak späť, predikuj skok
 - Inak, nie
- **Dynamická predikcia vetvenia:**
 - Na základe histórie (niekoľko stoviek) skokov
 - *Zásobník cieľových adres skokov (Branch target buffer) :*
 - Cieľová adresa skoku
 - Bola alebo nebola skok

Príklad

```
add $s1, $0, $0    # sum = 0
```

```
add $s0, $0, $0    # i = 0
```

```
addi $t0, $0, 10   # $t0 = 10
```

for:

```
beq $s0, $t0, done  # if i == 10, branch
```

```
add $s1, $s1, $s0  # sum = sum + i
```

```
addi $s0, $s0, 1    # increment i
```

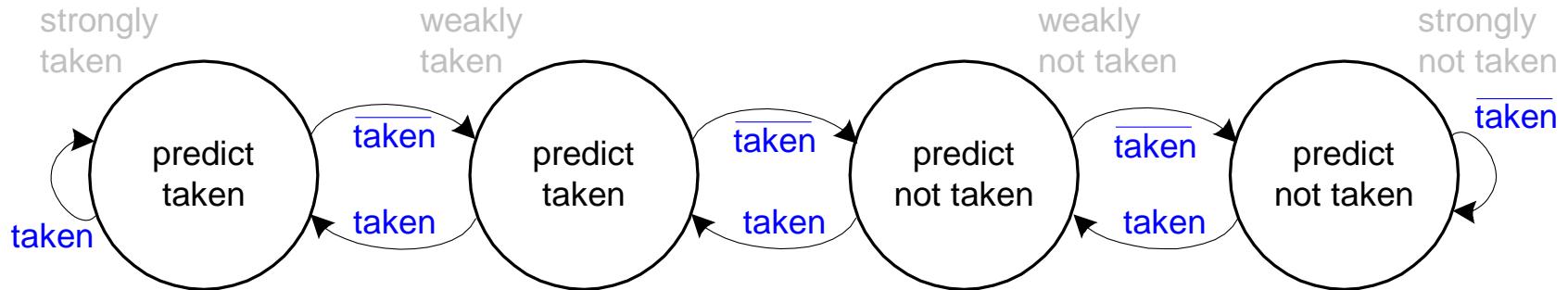
```
j for
```

done:

1-Bit prediktor vetvenia

- Zapamätaj si, či bol alebo nebol skok vykonaný
 - Ak bol, vykonaj skok opäť
- Prvý a posledný skok (pre cykly) nie je predikovaný správne

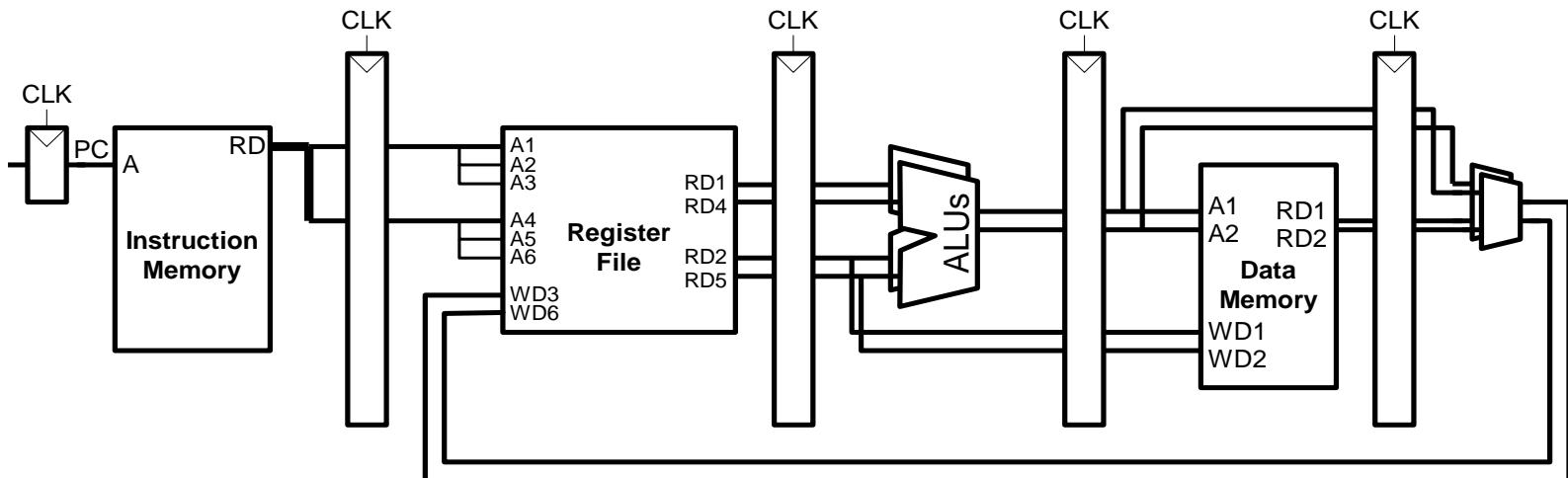
2-Bit prediktor vetvenia



Zlá predikcia len pre posledný cyklus

Superskalárna architektúra

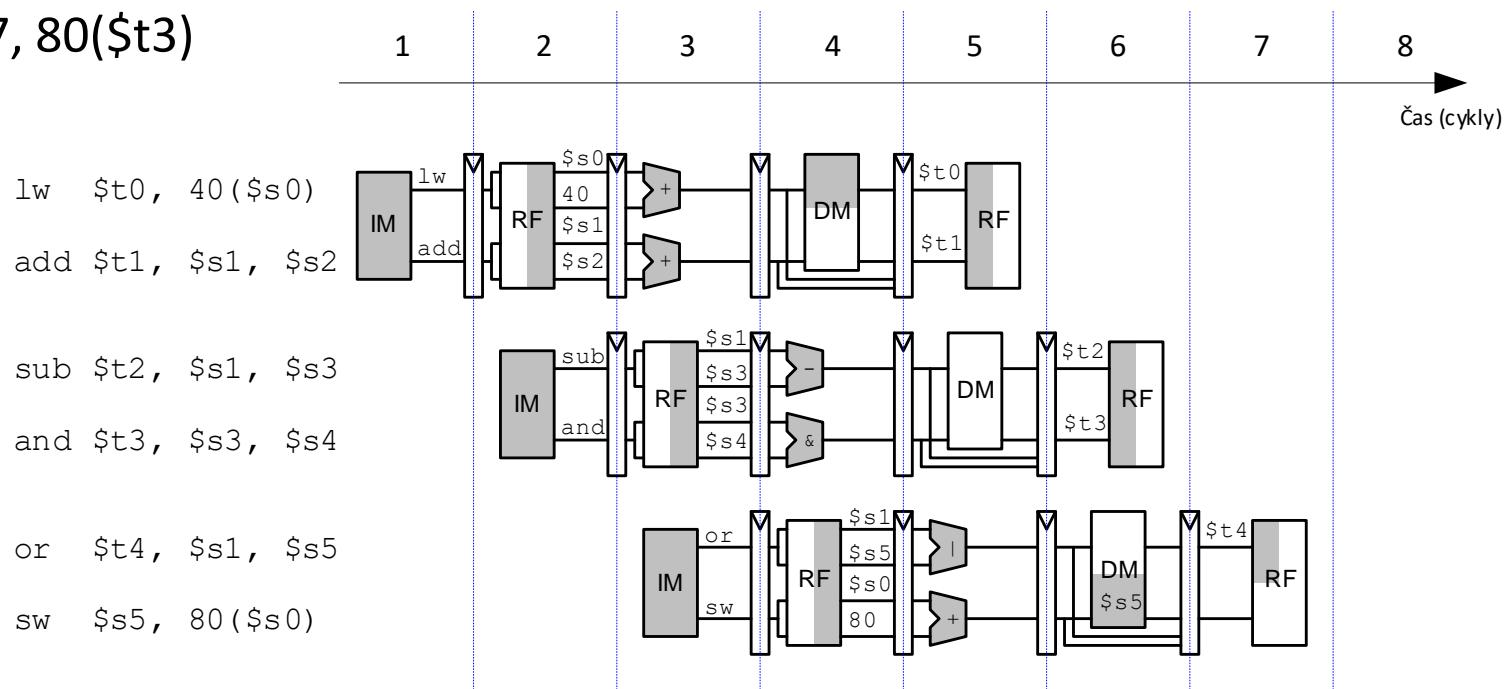
- Viac inštrukcií je spracovaných naraz
- Hazardy môžu byť problém



Príklad

lw \$t0, 40(\$s0)
add \$t1, \$t0, \$s1
sub \$t0, \$s2, \$s3
and \$t2, \$s4, \$t0
or \$t3, \$s5, \$s6
sw \$s7, 80(\$t3)

Ideálny IPC: 2
Aktuálny IPC: 2



Superskalárny proc. a hazardy

lw \$t0, 40(\$s0)

add \$t1, \$t0, \$s1

sub \$t0, \$s2, \$s3

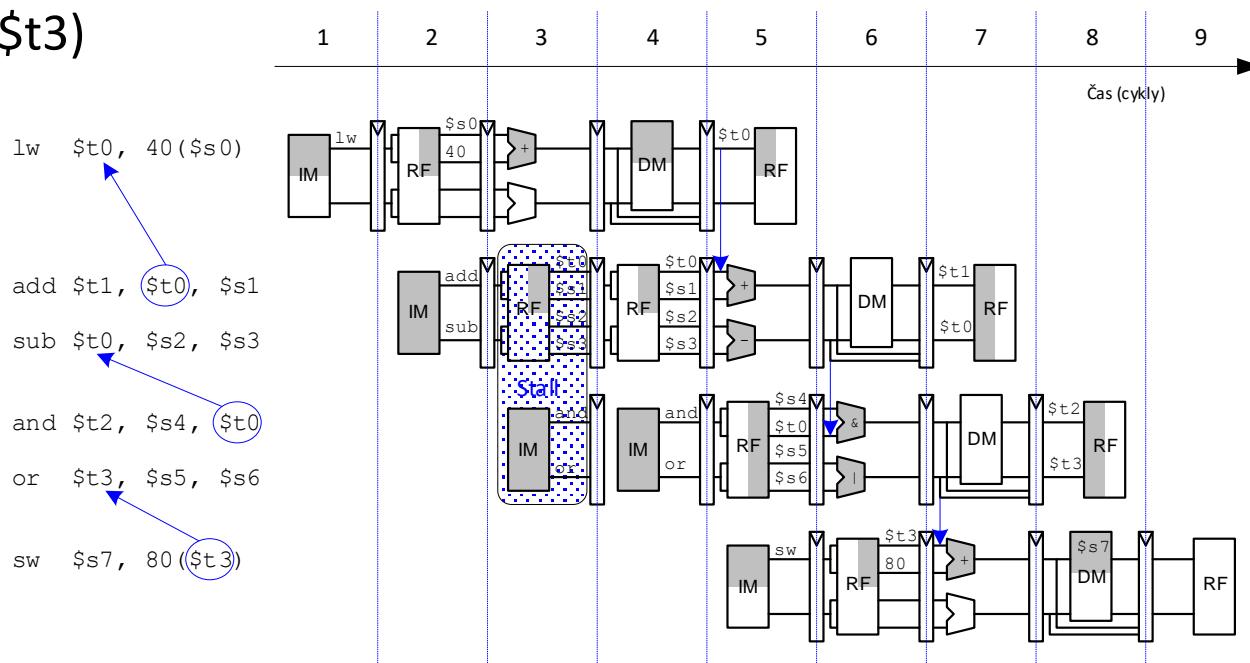
and \$t2, \$s4, \$t0

or \$t3, \$s5, \$s6

sw \$s7, 80(\$t3)

Ideálny IPC: 2

Aktuálny IPC: 6/5 = 1.17



Spracovanie inštrukcií mimo poradia

- Analýza niekoľkých inštrukcií
- Priprav čo najviac inštrukcií na spracovanie
- Spracuj inštrukcie aj mimo poradia (ak je to možné)
- **Závislosti:**
 - **RAW** (read after write): údajová závislosť
 - **WAR** (write after read): údajová antizávislosť
 - **WAW** (write after write): výstupná závislosť

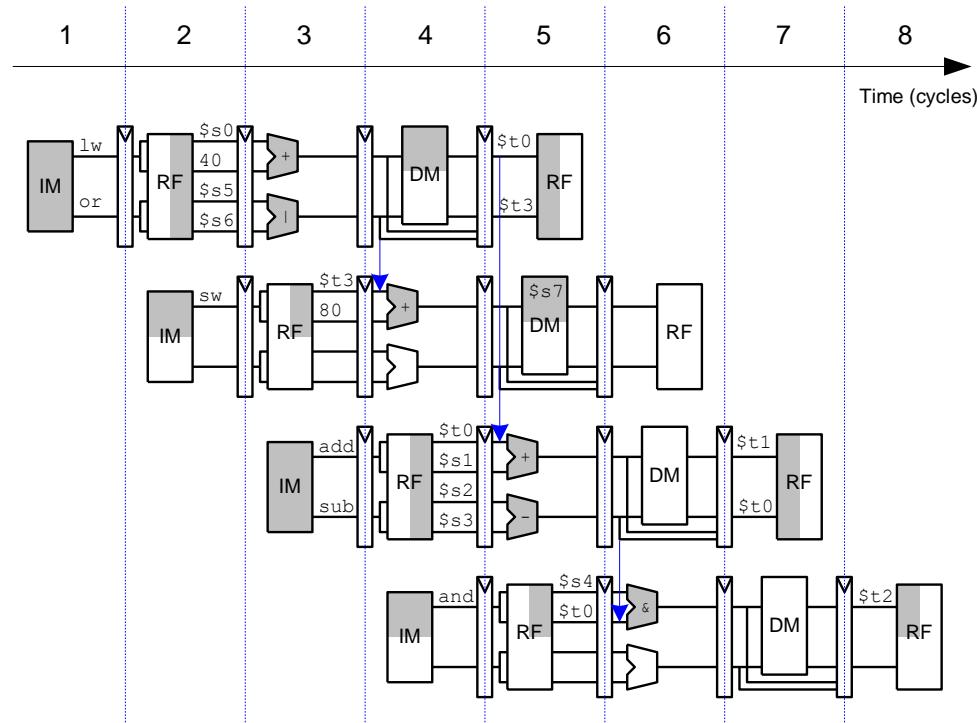
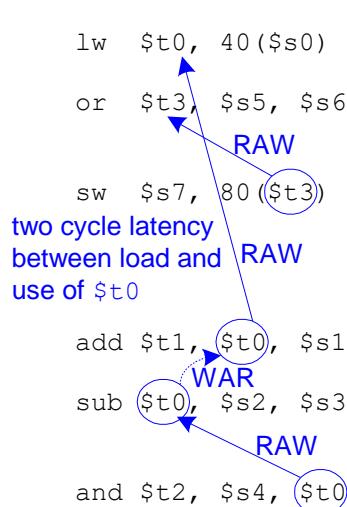
Plánovanie spracovania inštrukcií

- **Instruction level parallelism (ILP):** počet inštrukcií pripravených na spracovanie (priemer < 3)
- **Prehľadávací obvod (Scoreboard):**
 - Sleduje
 - Inštrukcie pripravené na spracovanie
 - Dostupnosť funkčných jednotiek
 - Hazardy

Príklad

lw \$t0, 40(\$s0)
 add \$t1, \$t0, \$s1
 sub \$t0, \$s2, \$s3
 and \$t2, \$s4, \$t0
 or \$t3, \$s5, \$s6
 sw \$s7, 80(\$t3)

Ideálny IPC: 2
Aktuálny IPC: $6/4 = 1.5$



Premenovávanie registrov

lw \$t0, 40(\$s0)

add \$t1, \$t0, \$s1

sub \$t0, \$s2, \$s3

and \$t2, \$s4, \$t0

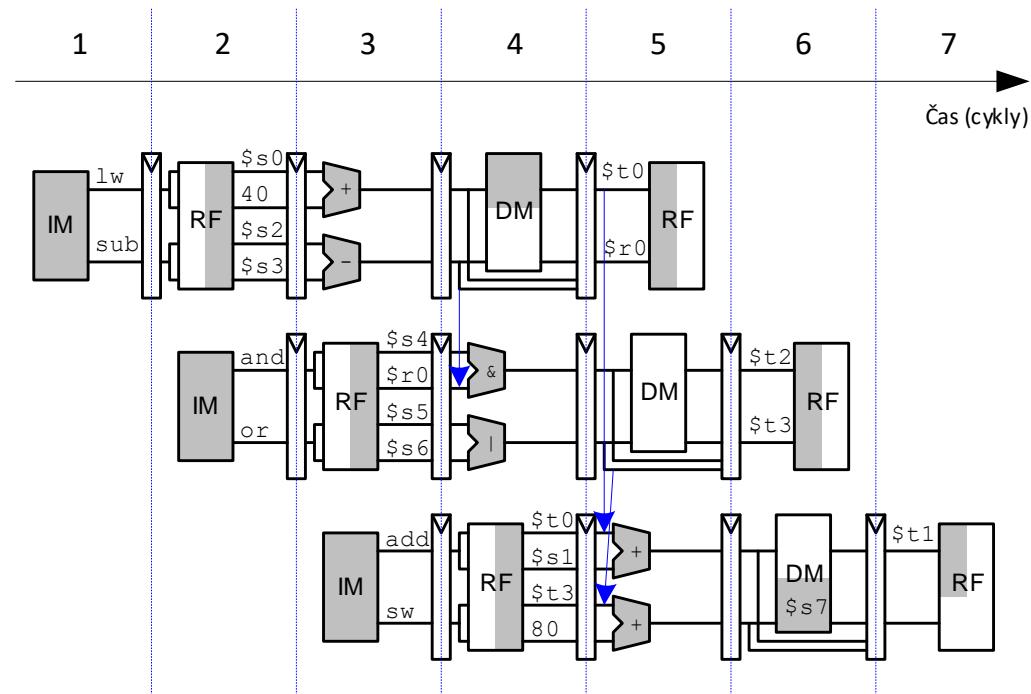
or \$t3, \$s5, \$s6

sw \$s7, 80(\$t3)

Ideálny IPC: 2

Aktuálny IPC: 6/3 = 2

lw \$t0, 40 (\$s0)
sub \$r0, \$s2, \$s3
2-cycle RAW
and \$t2, \$s4, \$r0
or \$t3, \$s5, \$s6
RAW
add \$t1, \$t0, \$s1
sw \$s7, 80 (\$t3)



- Single Instruction Multiple Data (SIMD)
 - Jeden prúd inštrukcií spracováva viacnásobný tok dát
 - Jedna riadiaca jednotka a viac vykonávacích jednotiek
 - Aplikačná doména: grafika
 - Napr. sčítaj štyri 8b skalárov

padd8 \$s2, \$s0, \$s1								
32	24	23	16	15	8	7	0	Bit position
a_3		a_2		a_1		a_0		\$s0
+	b_3		b_2		b_1		b_0	\$s1
<hr/>								
$a_3 + b_3$		$a_2 + b_2$		$a_1 + b_1$		$a_0 + b_0$		\$s2

Moderné prístupy

- **Multivláknové architektúry**
 - Mikro, nano, ...
- **Multiprocesory**
 - Jeden čip ale viac procesorov (jadier)

Proces vs vlákno

- **Proces:** program vykonávaný na počítači
 - Je možné spúšťať niekoľko procesov súčasne
 - Vyhľadávanie na internete, spúšťanie hudby, písanie
- **Vlákno:** časť programu
 - Proces môže pozostávať z niekoľkých vlákien
 - Napríklad textový procesor môže definovať samostatné vlákna pre spracovanie vstupov, kontroly gramatiky, tlače, a pod.

Vlákna v konvenčných procesoroch

- V danom čase „beží“ len jedno vlákno
- Ak sa pozastaví činnosť vlákna (napr. čakanie na vstup):
 - Uloží sa stav vlákna
 - Načíta a spustí sa druhé (čakajúce) vlákno
 - **Volá sa to zmena kontextu (context switching)**
 - Zmena kontextu medzi vláknenami jedného procesu je spravidla rýchlejšia ako zmena kontextu medzi procesmi.
- Zdá sa že všetky vlákna „bežia“ simultánne

Multithreading

- Aktivuje sa niekoľko vlákien súčasne:
 - Ak jedno je pozastavené/ukončené, načíta sa okamžite druhé
 - Ak jedno vlákno „neobsadí“ všetky hardvérové zdroje, druhé vlákno ich môže využiť
- Nezvyšuje mieru paralelizmu na úrovni inštrukcií (ILP) v rámci vlákna, ale zvyšuje priepustnosť

Intel to označuje pojmom “hyperthreading”

Multiprocesory

- Multiprocesorové systémy s definovanou komunikačnou topológiou
 - Tesne viazané multiprocesory
 - viacjadrové
 - Vol'ne viazané multiprocesory
 - multipočítače

Referencia

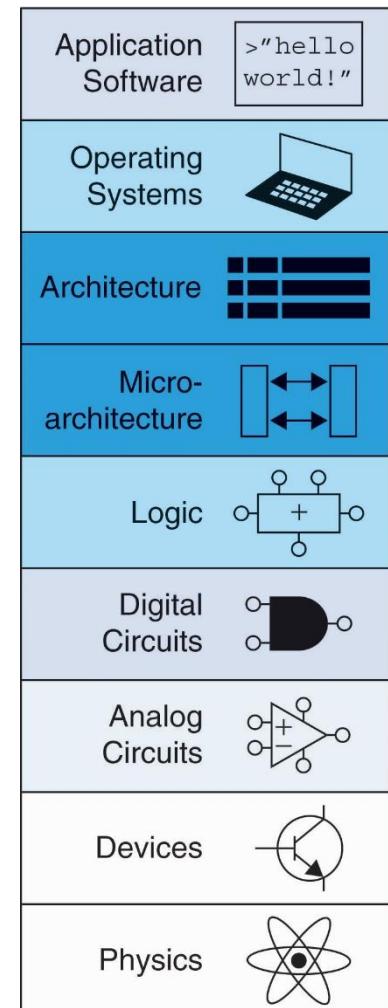
- David Money Harris and Sarah L. Harris,
Digital Design and Computer Architecture,
Chapter 7: Microarchitecture, Second Edition
© 2012 by Elsevier Inc.

Architektúry počítačových systémov

2. rok ZS

Osnova

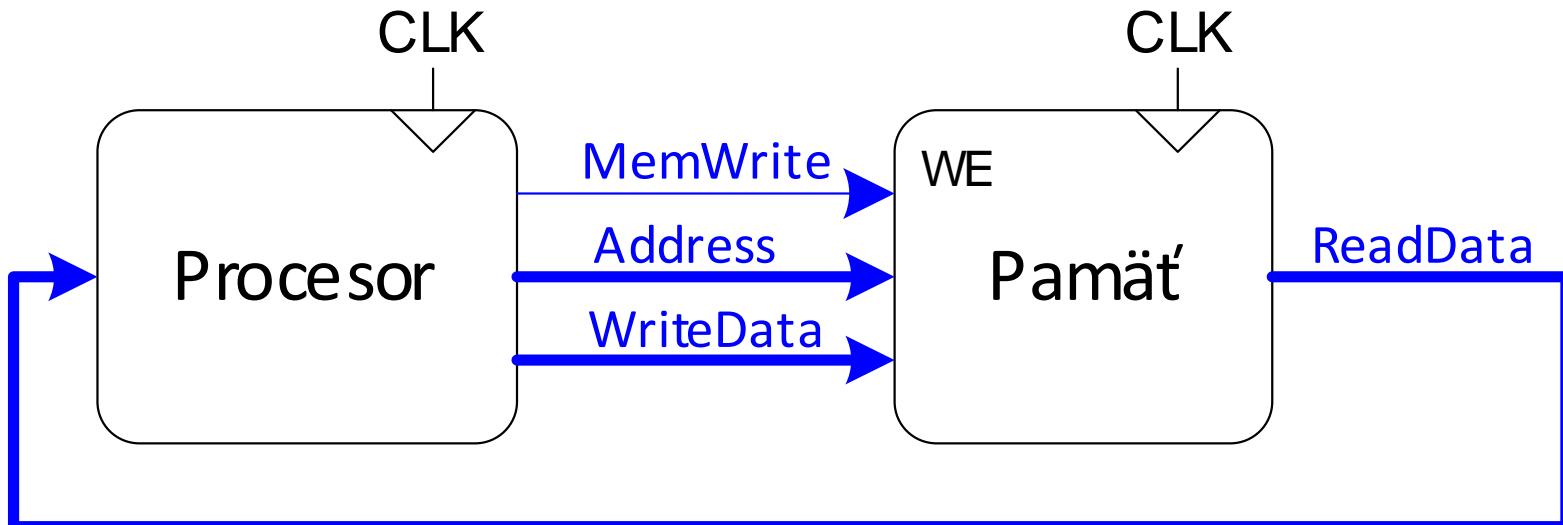
- Výkonnostné parametre
- Rýchla asociatívna pamäť
- Virtuálna Pamäť
- V/V pod systém



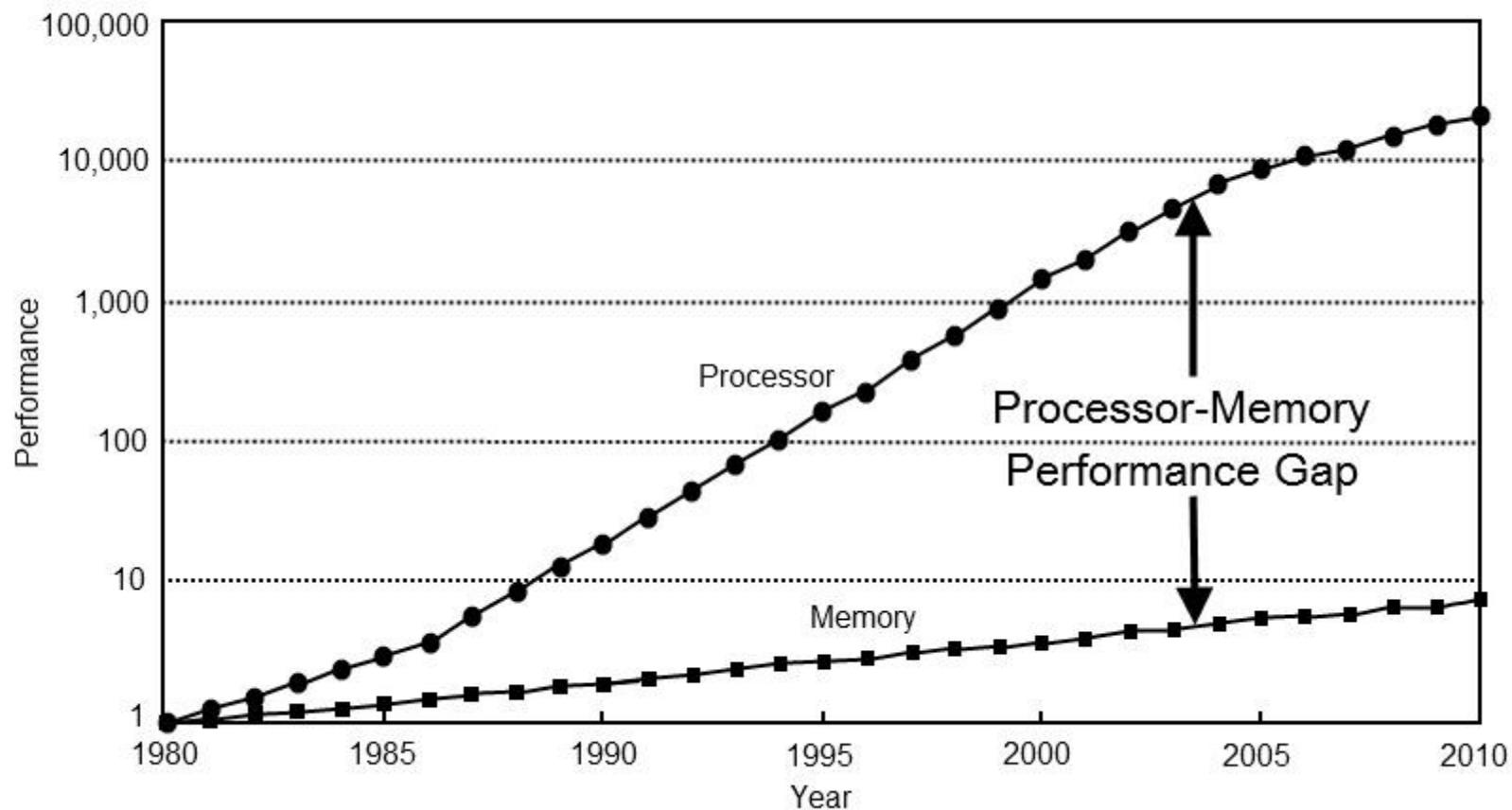
Výkonnostné parametre

- Výkon PS závisí od:
 - Výkonu procesora
 - Pamäťového podsystému (PaP)

Rozhranie pamäťového podsystému



Výkon: procesor vs pamäť

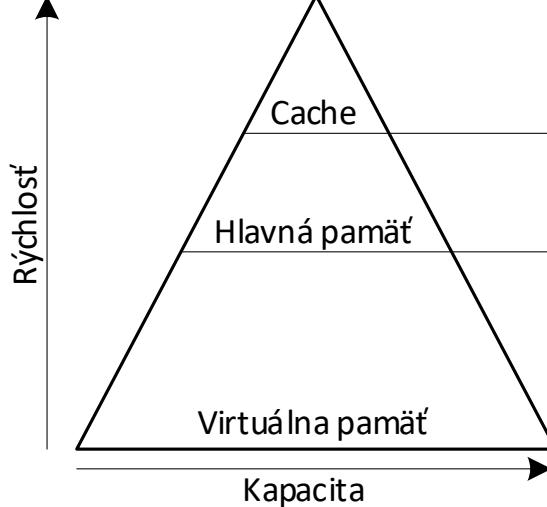


Návrh pamäťových modulov

- Cieľom je navrhnuť pamäťový podštém (PaP) tak rýchly ako je sám procesor
 - Aplikuje sa hierarchická stavba PaP
 - Vlastnosti ideálnej pamäte:
 - Rýchla
 - Lacná
 - Veľká kapacita
- Dve sú splnitelné!*



Hierarchia PaP



Technológia	Cena / GB	Priestupová doba (ns)	Šírka prenosového pásma (GB/s)
SRAM	\$10,000	1	25+
DRAM	\$10	10 - 50	10
SSD	\$1	100,000	0.5
HDD	\$0.1	10,000,000	0.1

- **Temporálna lokalita:**
 - Časová charakteristika prístupu
 - Ak sa údaj používa často, vysokou pravdepodobnosťou sa použije znova
 - **Ako to zužitkovat:** posunúť častou sprístupňované údaje do rýchlejšej pamäte PaP
- **Priestorová lokalita:**
 - Priestorová charakteristika prístupu
 - Ak sa údaj používa často, vysokou pravdepodobnosťou sa použijú aj údaje umiestnené v jeho blízkosti
 - **Ako to zužitkovat:** posunúť bloky dát z blízkosti frekventovaného prístupu do rýchlejšej pamäte PaP

Výkonnostné parametre

- **Úspešné sprístupnenie (hit):** údaj sa našiel na danej úrovni PaP
- **Neúspešné sprístupnenie (miss):** údaj sa nenašiel (treba sprístupniť z nižšej úrovne)

Koeficient úspešnosti (Hit Rate; HR)

$$\begin{aligned} \text{HR} &= \# \text{ úspešné sprístupnenie} / \# \text{ prístupov do pamäte} \\ &= 1 - \text{MR} \end{aligned}$$

Koeficient neúspešnosti (Miss Rate; MR)

$$\begin{aligned} \text{MR} &= \# \text{ neúspešné sprístupnenie} / \# \text{ prístupov do pamäte} \\ &= 1 - \text{HR} \end{aligned}$$

- **Priemerná doba prístupu do pamäte (AMAT):**

$$\text{AMAT} = \text{HR}_{\text{cache}} t_{\text{cache}} + \text{MR}_{\text{cache}} [\text{HR}_{\text{MM}} t_{\text{MM}} + \text{MR}_{\text{MM}} (\text{HR}_{\text{VM}} t_{\text{VM}})]$$

Príklad 1

- Program vstúpi do PaP 2000x (vykoná inštrukcie pre prácu s pamäťou - *lw, sw*)
- 1250x dát sú sprístupnené z pamäte cache
- V ostatných prípadoch dát sa nachádzajú mimo pamäte cache
- Aký je koeficient úspešnosti (**HR**) a koeficient neúspešnosti (**MR**) sprístupňovania údajov z pamäte cache?

Príklad 1

- Program vstúpi do PaP 2000x (vykoná inštrukcie pre prácu s pamäťou - *lw, sw*)
- 1250x dát sú sprístupnené z pamäte cache
- V ostatných prípadoch dát sa nachádzajú mimo pamäte cache
- Aký je koeficient úspešnosti (HR) a koeficient neúspešnosti (MR) sprístupňovania údajov z pamäte cache?

$$HR = 1250/2000 = 0,625$$

$$MR = 750/2000 = 0,375 = 1 - HR$$

Príklad 2

- Nech PaP pozostáva z 2 úrovní: cache a hlavná pamäť
- $t_{\text{cache}} = 1 \text{ cyklus}$, $t_{MM} = 100 \text{ cyklov}$
- Aká je AMAT programu z Príkladu 1?

Príklad 2

- Nech PaP pozostáva z 2 úrovní: cache a hlavná pamäť
- $t_{\text{cache}} = 1 \text{ cyklus}$, $t_{MM} = 100 \text{ cyklov}$
- Aká je AMAT programu z Príkladu 1?

$$\begin{aligned}\text{AMAT}^* &= t_{\text{cache}} + MR_{\text{cache}}(t_{MM}) \\ &= [1 + 0.375(100)] \text{ cycles} \\ &= \mathbf{38.5 \text{ cyklov}}\end{aligned}$$

*<https://www.geeksforgeeks.org/multilevel-cache-organisation/>

Gene Amdahl, 1922-

- **Amdahlov „zákon“:**
úsilie vynaložené na zvýšenie výkonu subsystému minie účinkom, pokial’ subsystém neovplyvní veľké percento celkového výkonu
- Vyjadruje maximálne predpokladané zrýchlenie systému potom, čo je vylepšená len niektorá z jeho častí.



Vyrovnávacia pamäť typu „cache“

- Pamäť určená na vyrovnávanie rýchlosťi prenosu údajov medzi procesorom a hlavnou pamäťou (HP)
- Je to rýchla asociatívna pamäť
- Doba sprístupnenia dát je typicky v jednotkách cyklov
- V ideálnom prípade procesor číta dáta z cache
- Je (dočasným) úložiskom najčastejšie sprístupňovaných údajov

Parametre cache

- **Kapacita (C):**
 - Veľkosť pamäte v bajtoch
- **Veľkosť bloku / skupiny slov v pamäti (b):**
 - Veľkosť prenášaných dát do cache
- **Počet blokov v pamäti ($B = C/b$):**
 - Prenos z hlavnej pamäte do cache sa uskutočňuje po skupinách slov
- **Stupeň asociativity (N):**
 - Ovplyvňuje miesto uloženia blokov do cache
- **Rám bloku cache ($S = B/N$):**
 - Skupina slov v cache v jednom ráme

Návrh pamäte cache

- Ktoré údaje presunúť do cache?
- Akú organizáciu cache zvoliť?
- Akú stratégiu zvoliť pri vyradení dát z cache?

Ktoré údaje presunúť do cache?

- Ideálne, často používané dáta sú uložené v cache
- Dá sa predvídať budúcnosť?
- Temporálna a priestorová lokalita:
 - **Temporálna lokalita**
 - Kopíruj požadované údaje z HP do cache
 - **Priestorová lokalita**
 - Kopíruj aj dáta uložené v susediacich bunkách

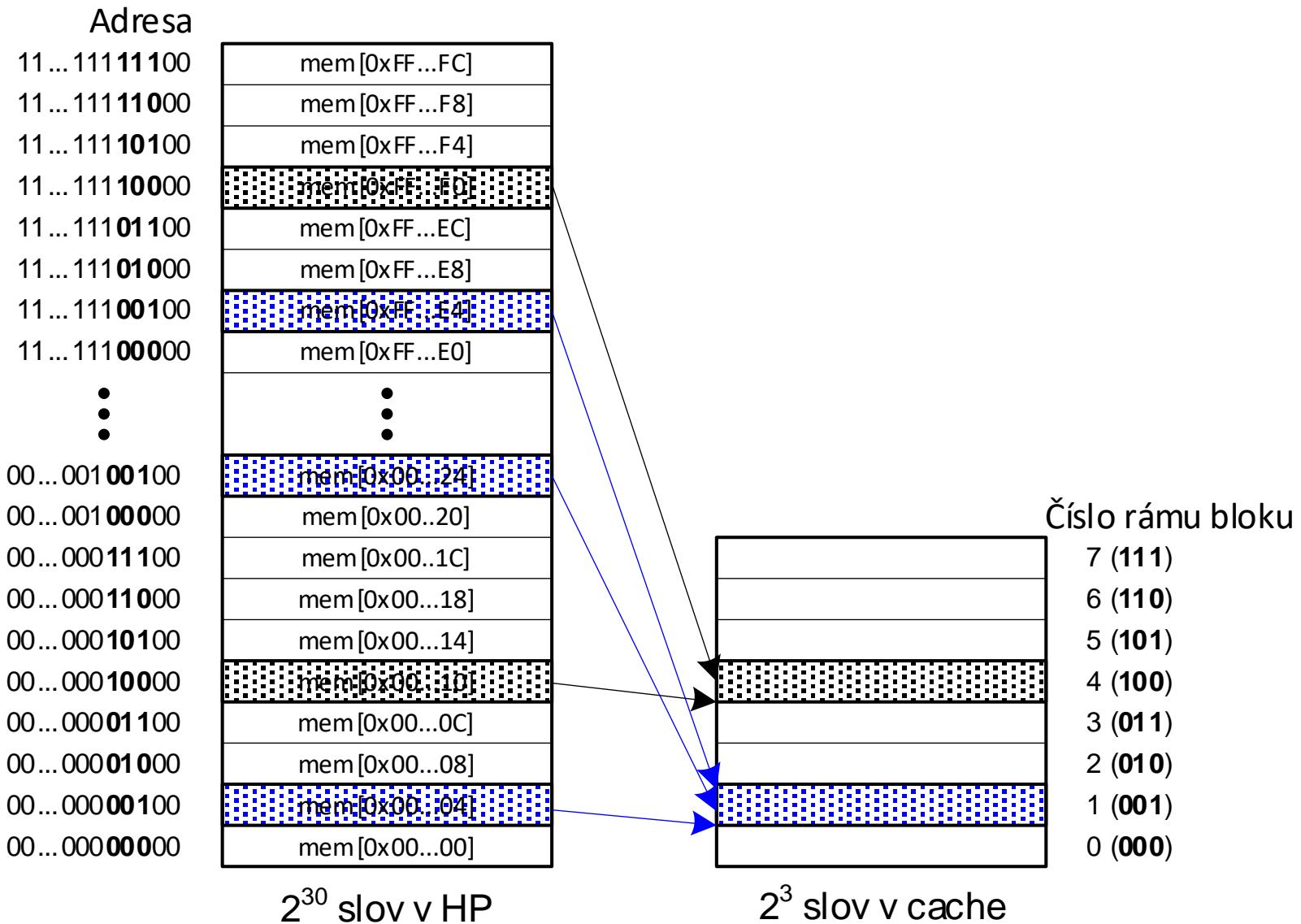
Akú organizáciu cache si zvoliť?

- Cache je delená na rámy blokov S
- Každý blok B z HP sa mapuje na jeden rám bloku S v cache
- Cache sa delí podľa stupňa asociativity:
 - **Cache s priamym mapovaním**
 - 1 blok per rám bloku ($S = B$)
 - **N -cestná asociatívna pamäť**
 - N blokov per rám bloku
 - **Plne asociatívna cache**
 - Všetky bloky do jedného rámu bloku

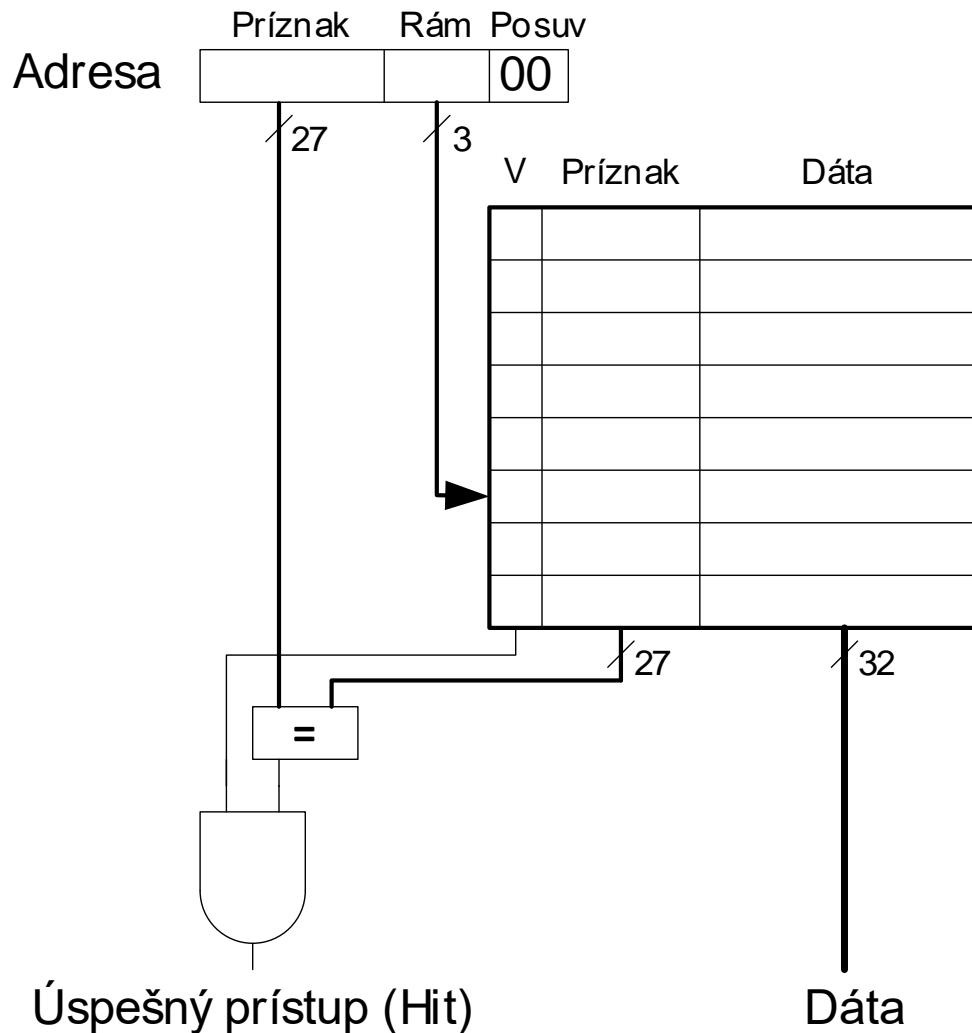
Príklad

- Sú dané nasledujúce parametre pamäte cache:
 - Kapacita $C = 8$ slov
 - Veľkosť bloku $b = 1$ slovo
 - Počet blokov $B = 8$

Cache s priamym mapovaním



Cache s priamym mapovaním

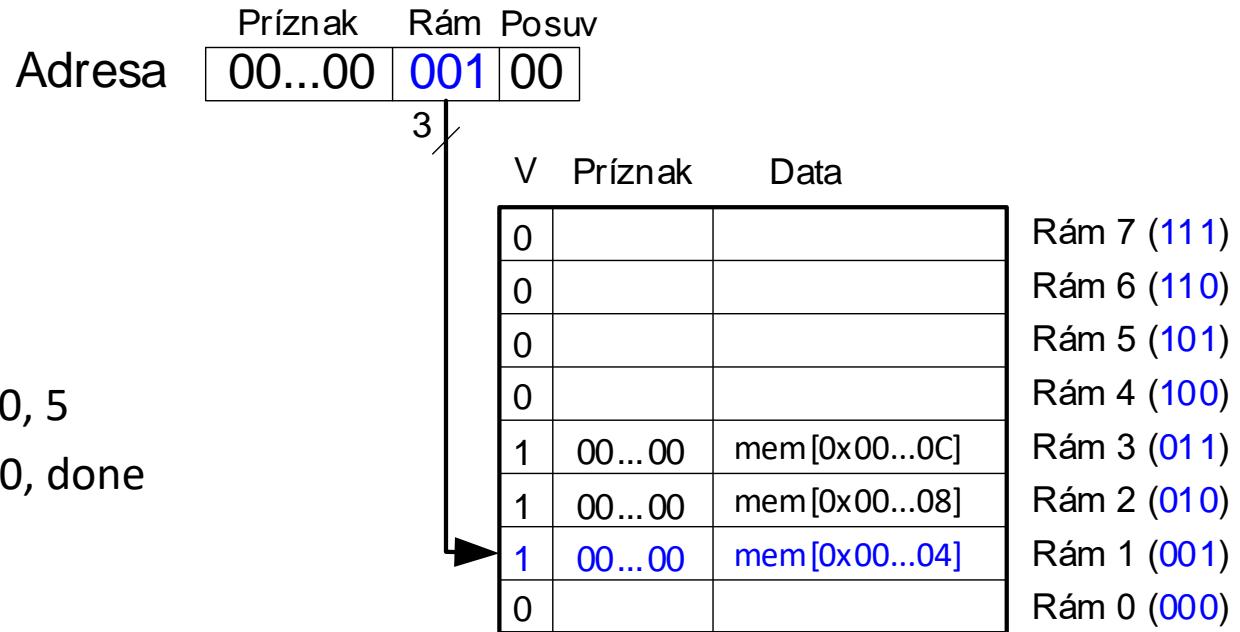


8-položiek
(rámov) x
(1+27+32)-
bitov
SRAM

Cache s priamym mapovaním

MIPS kód

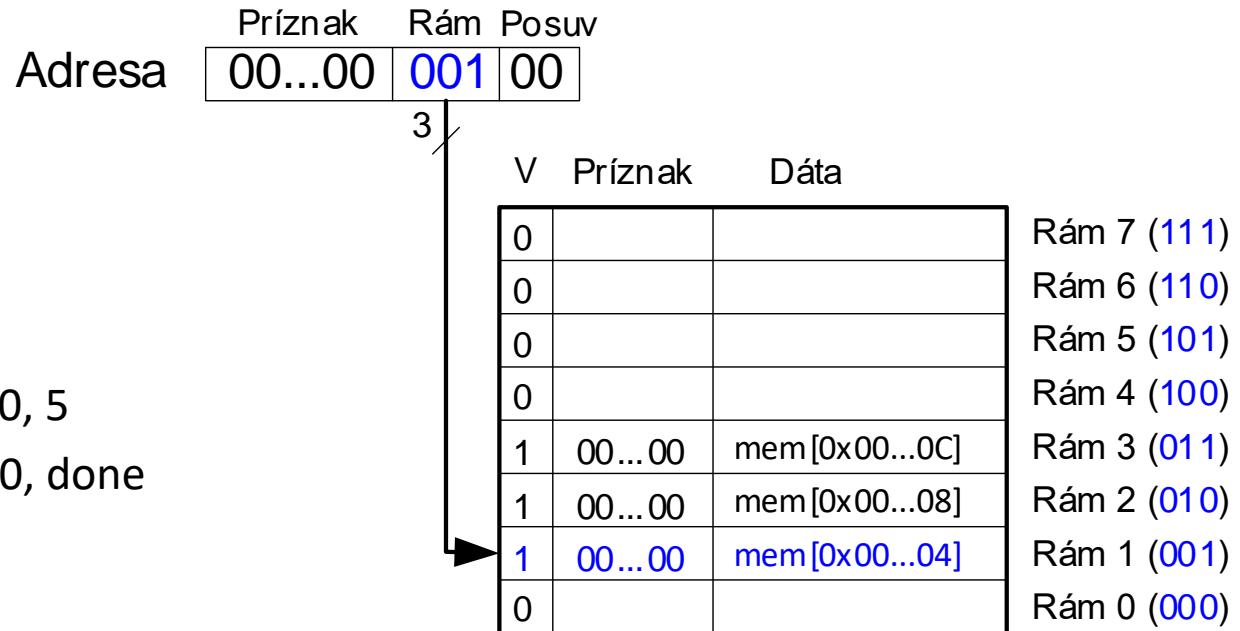
```
          addi $t0, $0, 5  
loop:   beq  $t0, $0, done  
        lw   $t1, 0x4($0)  
        lw   $t2, 0xC($0)  
        lw   $t3, 0x8($0)  
        addi $t0, $t0, -1  
        j   loop  
  
done:
```



Cache s priamym mapovaním

MIPS kód

```
          addi $t0, $0, 5
loop:    beq  $t0, $0, done
          lw   $t1, 0x4($0)
          lw   $t2, 0xC($0)
          lw   $t3, 0x8($0)
          addi $t0, $t0, -1
          j   loop
done:
```



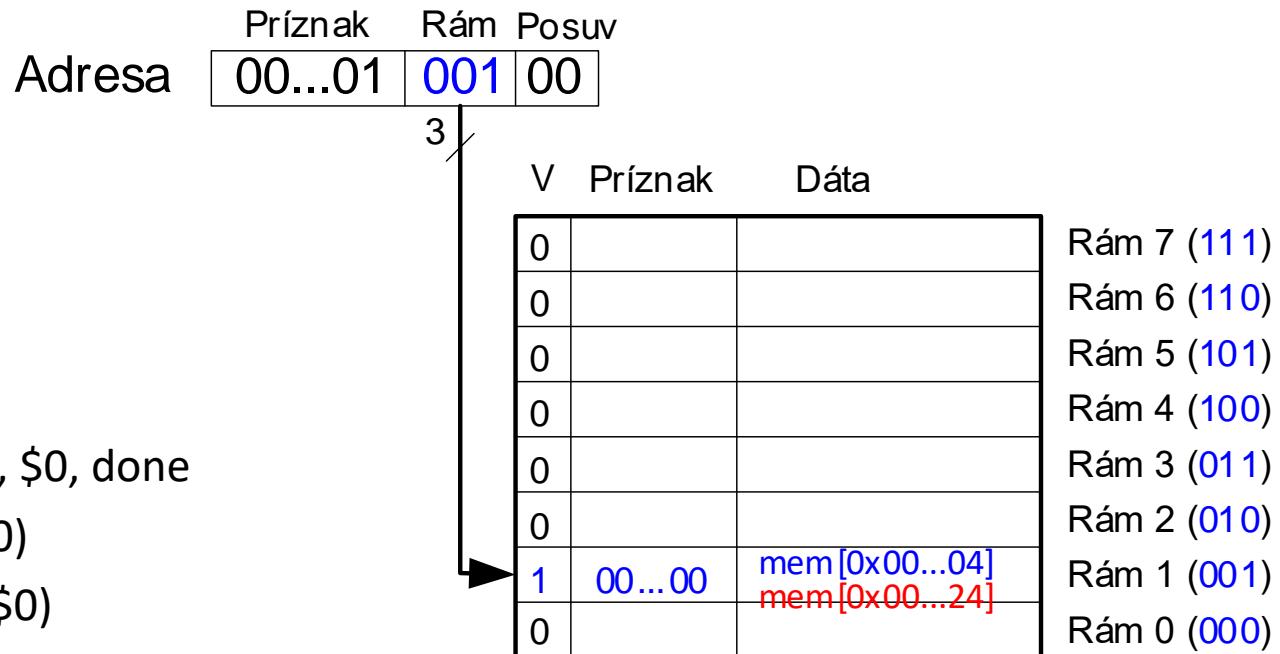
$$MR = 3/15$$

$$= 20\%$$

Cache s priamym mapovaním: konflikt

MIPS kód

```
addi $t0, $0, 5  
loop:    beq $t0, $0, done  
        lw $t1, 0x4($0)  
        lw $t2, 0x24($0)  
        addi $t0, $t0, -1  
        j loop  
done:
```

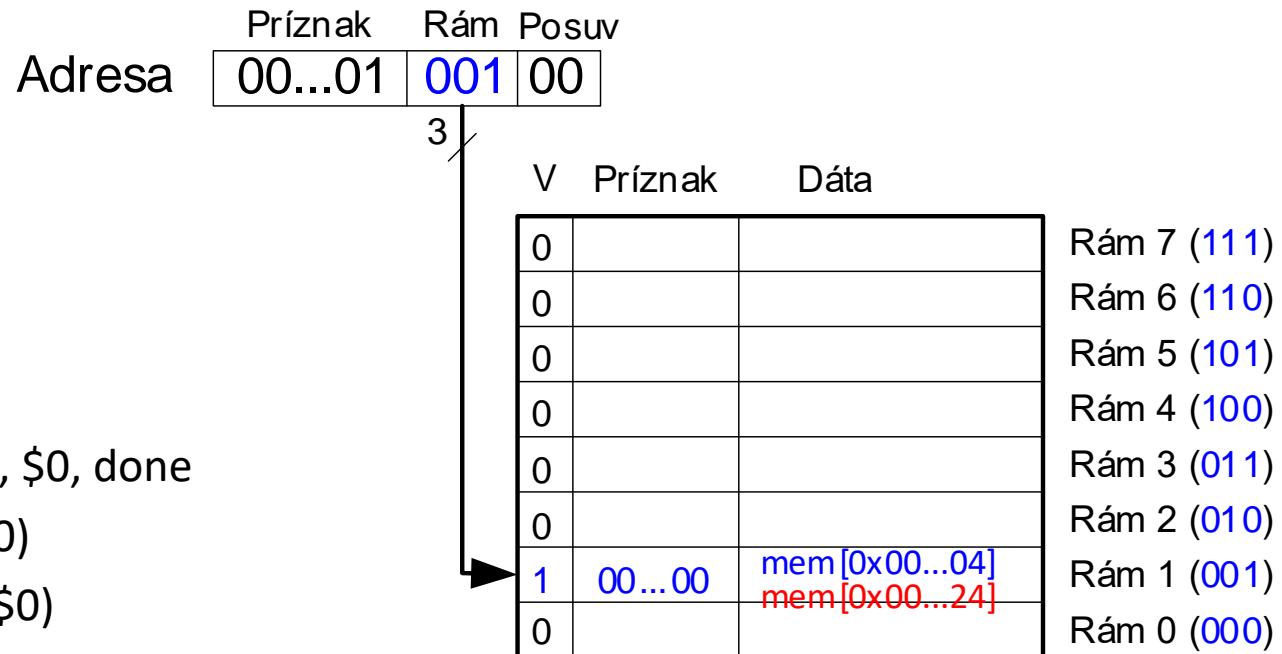


Miss Rate = ?

Cache s priamym mapovaním: konflikt

MIPS kód

```
addi $t0, $0, 5  
loop:    beq $t0, $0, done  
        lw $t1, 0x4($0)  
        lw $t2, 0x24($0)  
        addi $t0, $t0, -1  
        j loop  
done:
```

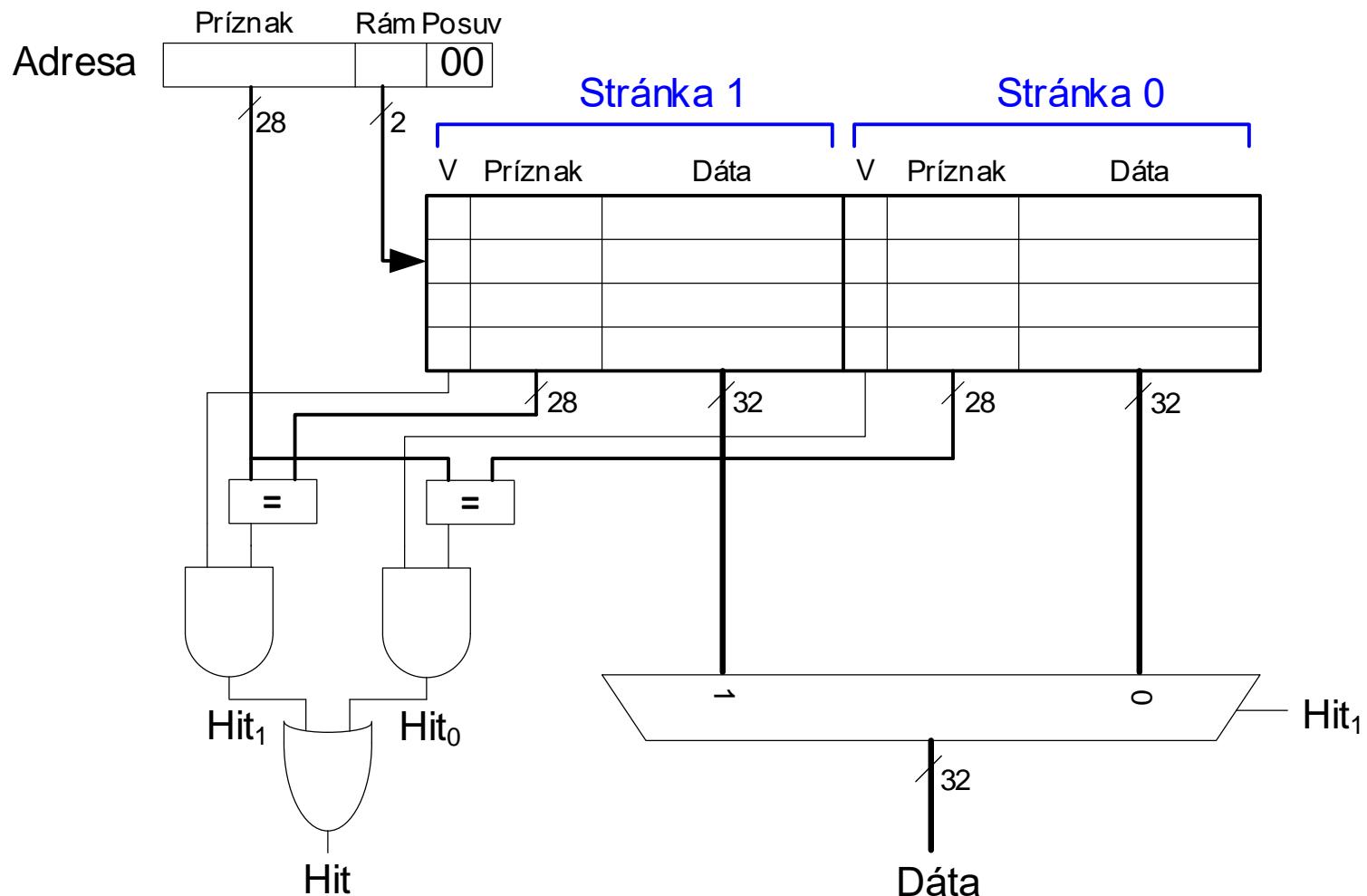


$$MR = 10/10$$

$$= 100\%$$

Konflikt!

N-cestná asociatívna pamäť



N-cestná asociatívna pamäť

MIPS kód

```
addi $t0, $0, 5  
loop:    beq $t0, $0, done  
          lw $t1, 0x4($0)  
          lw $t2, 0x24($0)  
          addi $t0, $t0, -1  
          j loop  
done:
```

MR = ?

Stránka 1			Stránka 0		
V	Príznak	Data	V	Príznak	Data
0			0		
0			0		
0			0		
0			0		

Rám 3
Rám 2
Rám 1
Rám 0

N-cestná asociatívna pamäť

MIPS kód

```
addi $t0, $0, 5  
loop:    beq $t0, $0, done  
          lw $t1, 0x4($0)  
          lw $t2, 0x24($0)  
          addi $t0, $t0, -1  
          j loop  
done:
```

$$\begin{aligned} \text{MR} &= 2/10 \\ &= 20\% \end{aligned}$$

Asociativita zníži počet konfliktov

Stránka 1			Stránka 0		
V	Príznak	Dáta	V	Príznak	Dáta
0			0		
0			0		
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]
0			0		

Rám 3
Rám 2
Rám 1
Rám 0

Plne asociatívna cache

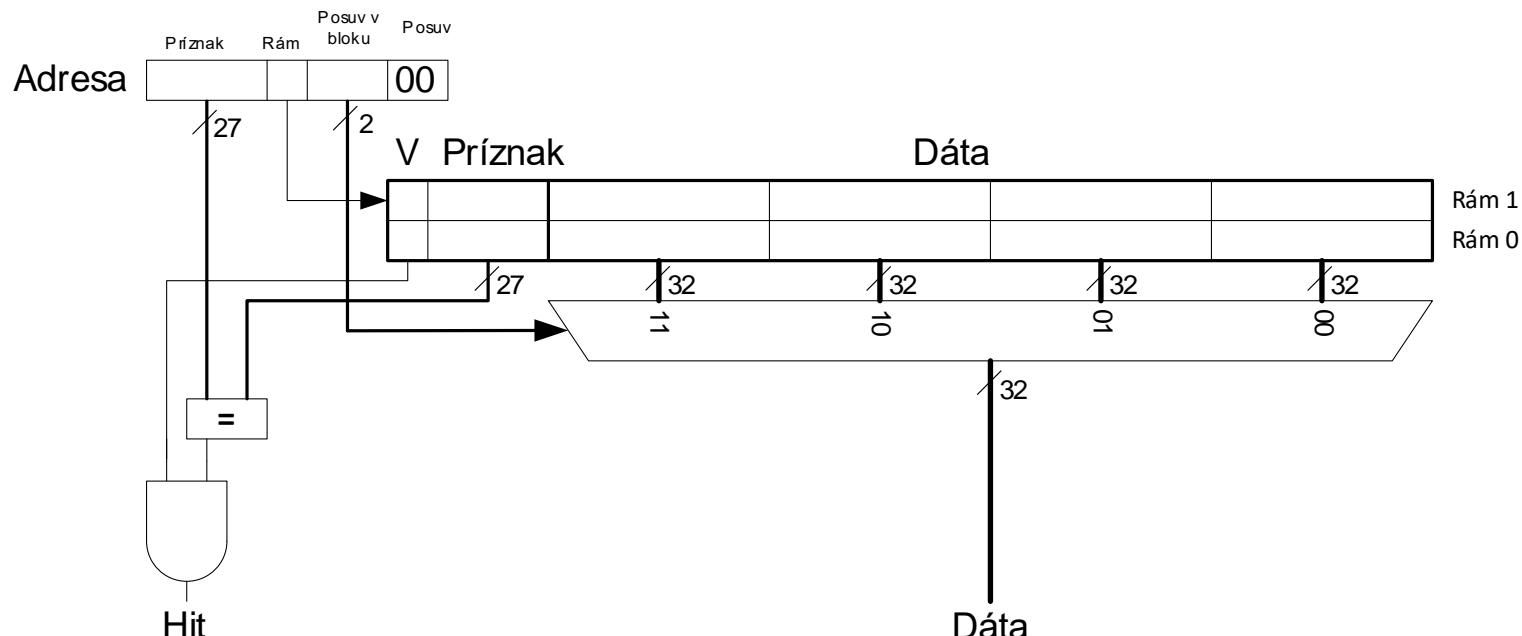
V	Príznak	Dáta	V	Príznak	Data	V	Príznak	Dáta																

Zníži počet konfliktov

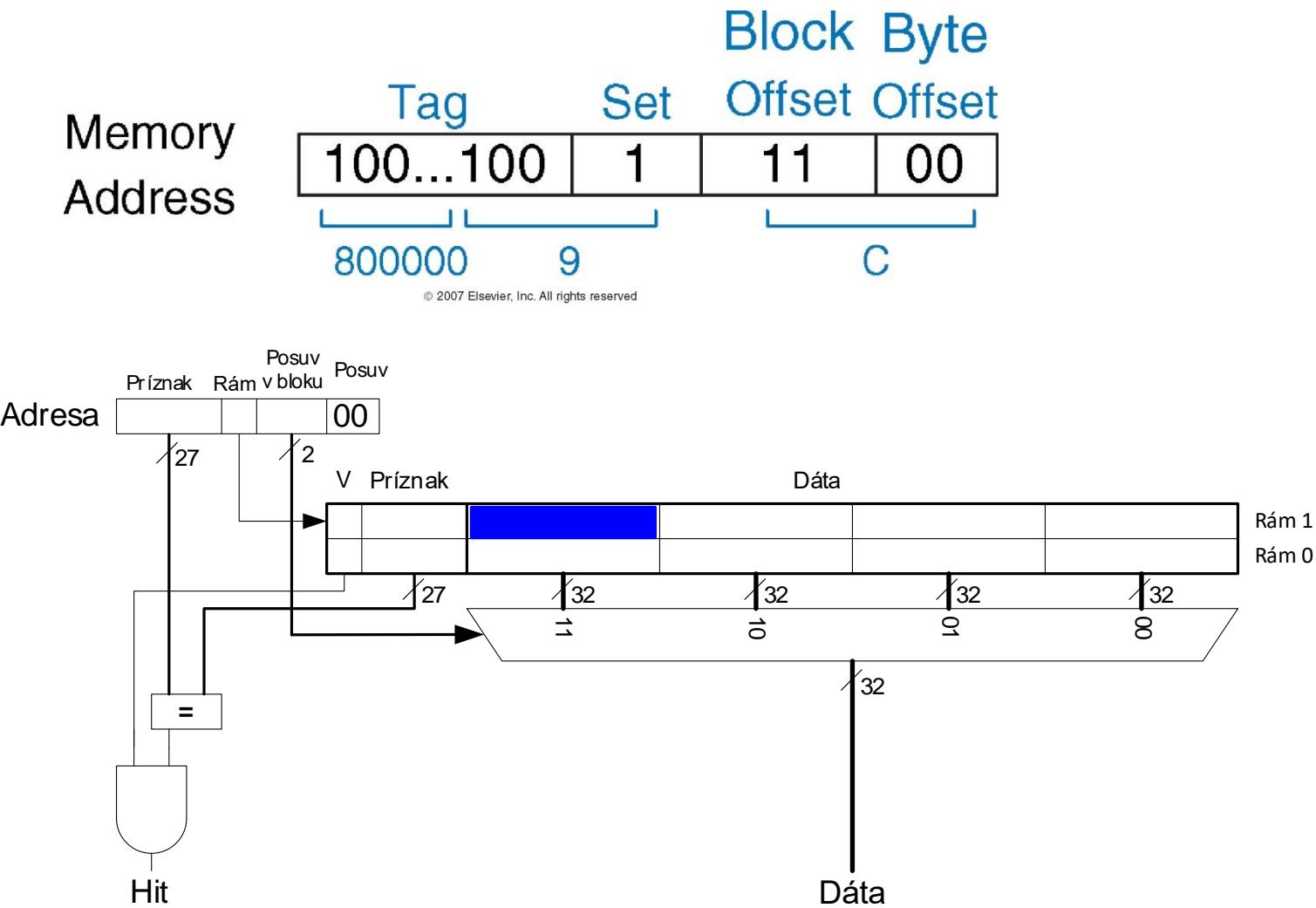
Je to drahé riešenie

Priestorová lokalita?

- Zväčšíme bloky:
 - *Velkosť bloku, $b = 4$ slová*
 - $C = 8$ slov
 - Priame mapovanie (1 blok per rám)
 - Počet blokov, $B = 2$ ($C/b = 8/4 = 2$)



Cache s väčším blokom



Cache s priamym mapovaním

addi \$t0, \$0, 5
loop: beq \$t0, \$0, done
 lw \$t1, 0x4(\$0)
 lw \$t2, 0xC(\$0)
 lw \$t3, 0x8(\$0)
 addi \$t0, \$t0, -1
 j loop
done:

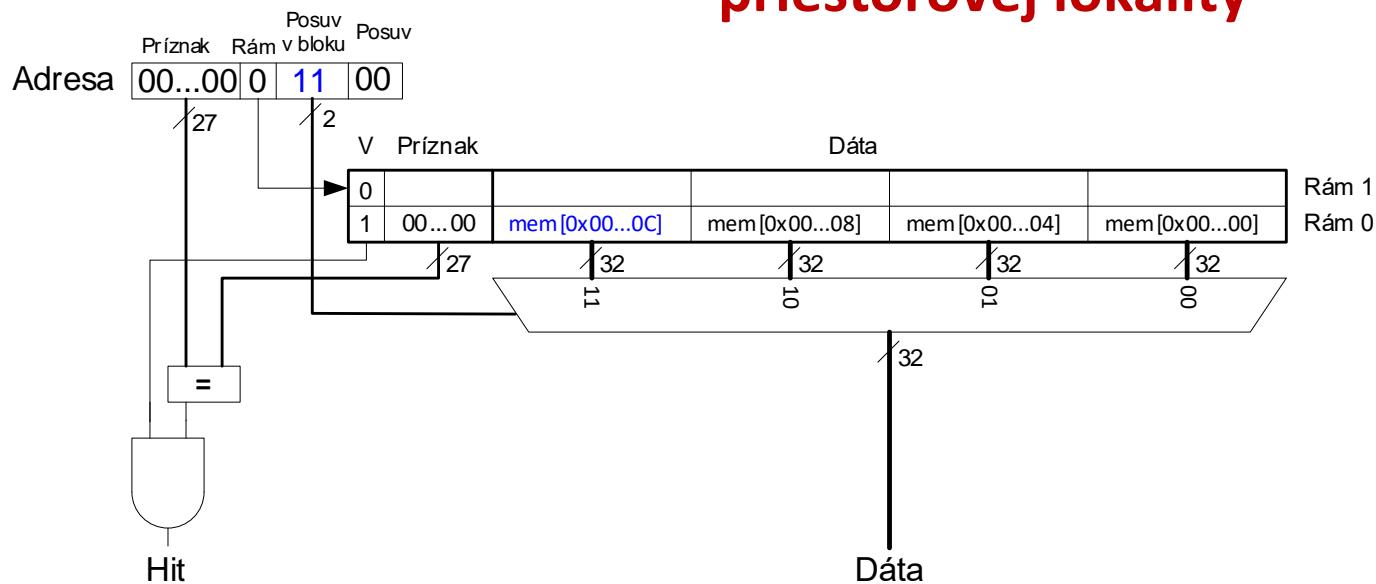
MR = ?

Cache s priamym mapovaním

```
addi $t0, $0, 5  
loop:    beq $t0, $0, done  
          lw  $t1, 0x4($0)  
          lw  $t2, 0xC($0)  
          lw  $t3, 0x8($0)  
          addi $t0, $t0, -1  
          j   loop  
  
done:
```

$$\begin{aligned} \text{MR} &= 1/15 \\ &= 6.67\% \end{aligned}$$

Väčšie bloky redukujú konflikty za pomocí priestorovej lokality



Sumár: organizácia cache

- Kapacita: C
- Veľkosť bloku: b
- Počet blokov v cache: $B = C/b$
- Počet blokov v ráme: N
- Počet rámov: $S = B/N$

Organizácia	Počet ciest/skupín/ blokov v ráme (N)	Počet rámov ($S = B/N$)
Priame mapovanie	1	B
N-cestná asociativita	$1 < N < B$	B / N
Plná asociativita	B	1

Kapacita pamäte cache

- Pamäť cache má obmedzenú kapacitu
- Ak je plná: program požaduje údaj X → musí sa uvoľniť priestor v cache → preto sa vyradí údaj Y
- **Vznikne konflikt** ak program potrebuje údaj Y
- Ako si zvoliť „správny“ Y na vyradenie z cache?
- **Metódy vyradovania**
 - Náhodný výber
 - Výber podľa príznaku aktivity
 - **Stratégia LRU (Least recently used):**
 - Vyradí sa najdlhšie nepoužívaný údaj (blok) z cache

Stratégia LRU

MIPS kód

```
lw $t0, 0x04($0)  
lw $t1, 0x24($0)  
lw $t2, 0x54($0)
```

Stránka 1				Stránka 0			
V	U	Príznak	Dáta	V	U	Príznak	Dáta
0	0			0			
0	0			0			
0	0			0			
0	0			0			

Rám 3 (11)
Rám 2 (10)
Rám 1 (01)
Rám 0 (00)

Stratégia LRU

MIPS kód

```
lw $t0, 0x04($0)  
lw $t1, 0x24($0)  
lw $t2, 0x54($0)
```

Stránka 1				Stránka 0			
V	U	Príznak	Dáta	V	Príznak	Dáta	
0	0			0			Rám 3 (11)
0	0			0			Rám 2 (10)
1	0	00...010	mem [0x00...24]	1	00...000	mem [0x00...04]	Rám 1 (01)
0	0			0			Rám 0 (00)

(a)

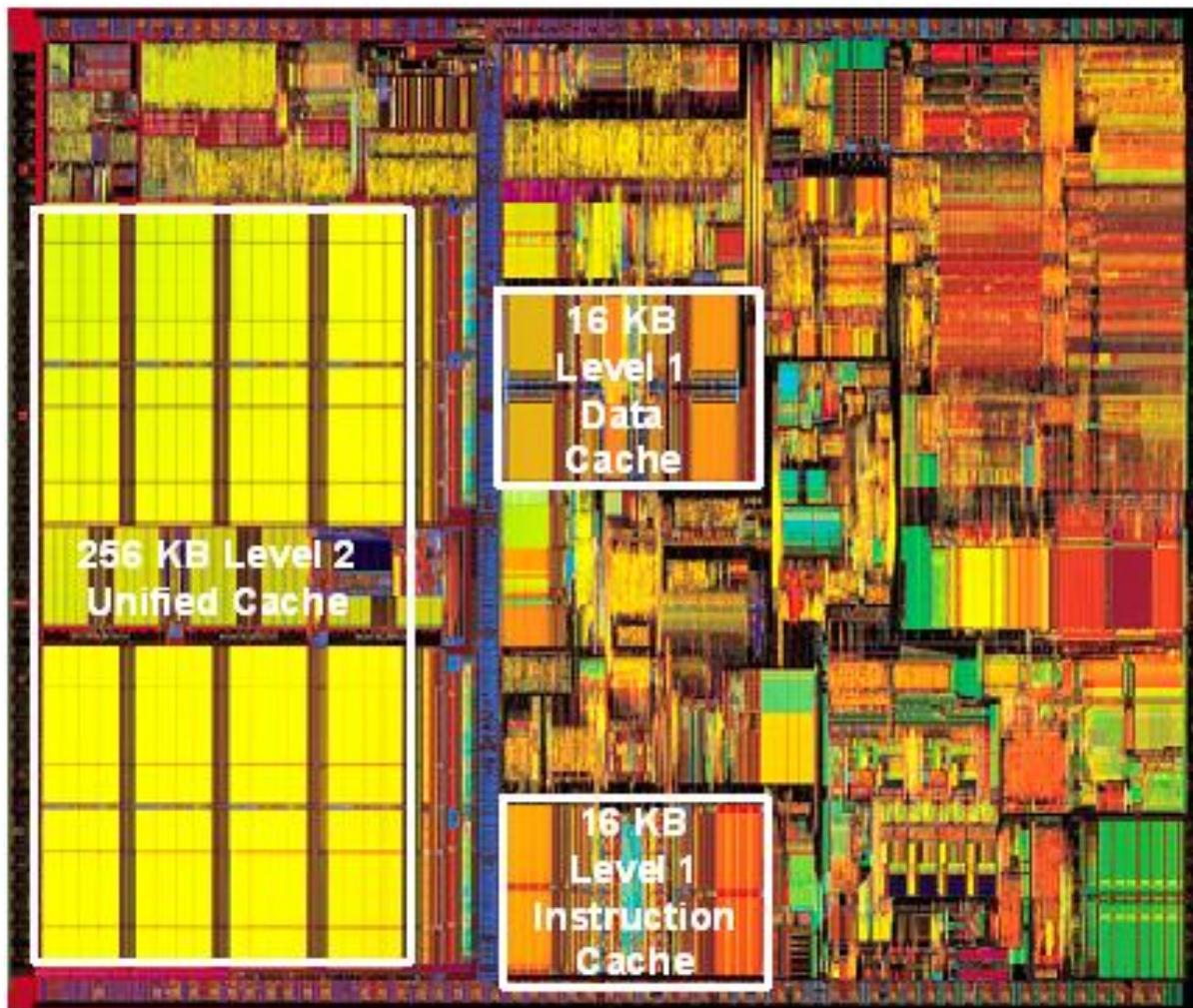
Stránka 1				Stránka 0			
V	U	Príznak	Dáta	V	Príznak	Dáta	
0	0			0			Rám 3 (11)
0	0			0			Rám 2 (10)
1	1	00...010	mem [0x00...24]	1	00...101	mem [0x00...54]	Rám 1 (01)
0	0			0			Rám 0 (00)

(b)

Viacúrovňová organizácia cache

- Veľká cache má nižší koeficient neúspešnosti a prístupová doba je väčšia v porovnaní s menšou cache
- Viacúrovňová organizácia cache predstavuje kompromis
- Úroveň 1: malá a rýchla (napr. 16 KB, 1 cyklus)
- Úroveň 2: veľká ale pomalá (napr. 256 KB, 2-6 cyklov)
- Moderné architektúry majú L1, L2, prípadne aj L3 cache

Intel Pentium III



Virtuálna pamäť

- Virtuálnou pamäťou sa získa väčší pamäťový priestor než čo nám ponúka hlavná pamäť (HP) v podobe napr. DRAM modulov.
- V súčasnosti je najčastejšie využívaná koncepcia virtuálnej pamäte, ktorá riadi a organizuje činnosť dvoch úrovní pamäti, ktorými sú hlavná (HP) a disková pamäť (DP).
- Hlavná pamäť (DRAM) vystupuje v úlohe cache pamäte pre pevný disk

Virtuálna pamäť

- **Virtuálna adresa**

- V programe sa používa virtuálna adresa na určenie miesta kde sa údaj nachádza
- Časť dát je v HP, časť na DP (napr. v DRAM)
- CPU zabezpečí preklad virtuálnej adresy na **fyzickú adresu** (adresu v DRAM)
- Údaje, ktoré nie sú v DRAM sa načítajú z DP

- **Ochrana pamäte**

- Každý program má vlastnú schému prekladu virtuálnych adres na fyzické
- Dva programy môžu použiť tú istú virtuálnu adresu pre rôzne dátu
- Programy „nevvidia“ adresný priestor iných programov
 - Výnimkou sú vírusy

Cache vs virtuálna pamäť

Cache	Virtuálna pamäť'
Blok	Stránka
Neúspešný prístup	Výpadok stránky
Príznak (klúč) asociatívneho výberu = kópia bázovej adresy bloku v cache	Virtuálna adresa stránky

Virtuálna pamäť

- Virtuálne adresy (VA) priestoru VP sú generované v priebehu komplikácie.
- Ich transformácia na fyzické adresy (FA) priestoru fyzickej pamäte (FP) sa uskutočňuje v priebehu spracovania programu.
- Uvedená transformácia adries sa nazýva **mapovanie pamäte**.

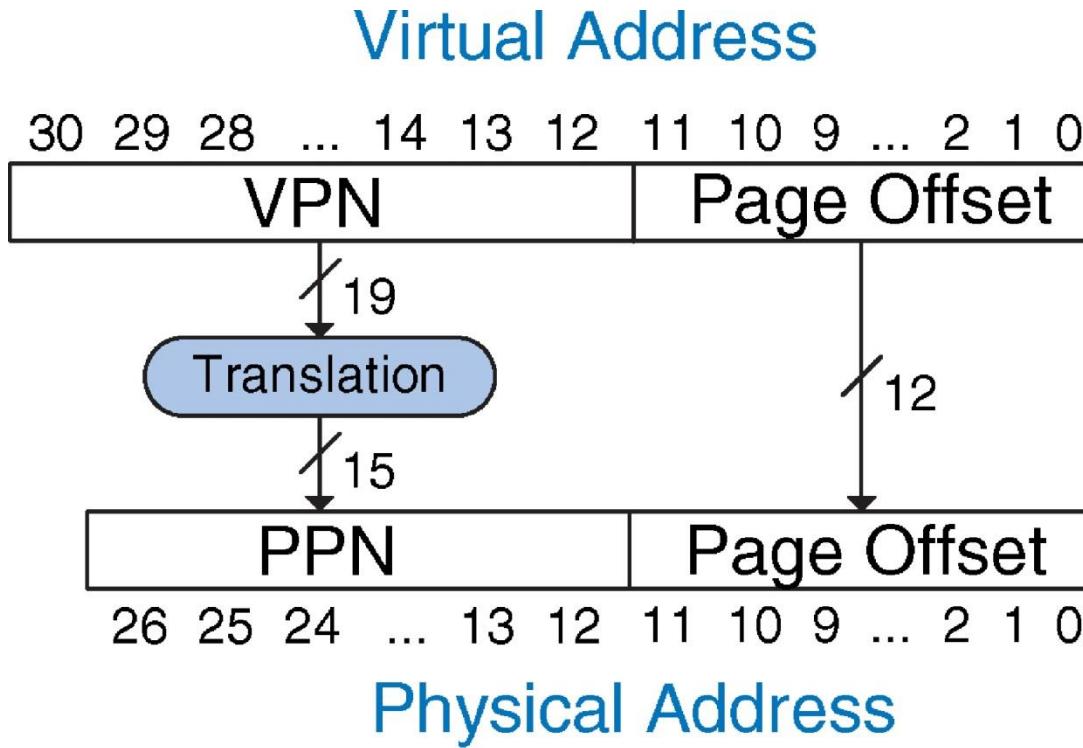
Virtuálna pamäť

- **Virtuálna adresa** je adresa, ktorou sa identifikuje slovo (blok) programu v logickom adresovom priestore.
- **Fyzická adresa** predstavuje adresu kópie tohto slova (bloku) lokalizovaného v HP, resp. v DP.
- Existujú dve skupiny VP:
 - s pevnou dĺžkou blokov, ktoré sa nazývajú **stránky** a
 - s premenlivou dĺžkou blokov, ktoré sa nazývajú **segmenty**.

Stránková organizácia pamäte

- Skupiny susedných 2^k pamäťových buniek sa zlučujú do pamäťového priestoru, ktorý sa nazýva
 - stránka v prípade VP
 - rám stránky v prípade FP

Mapovanie pamäte



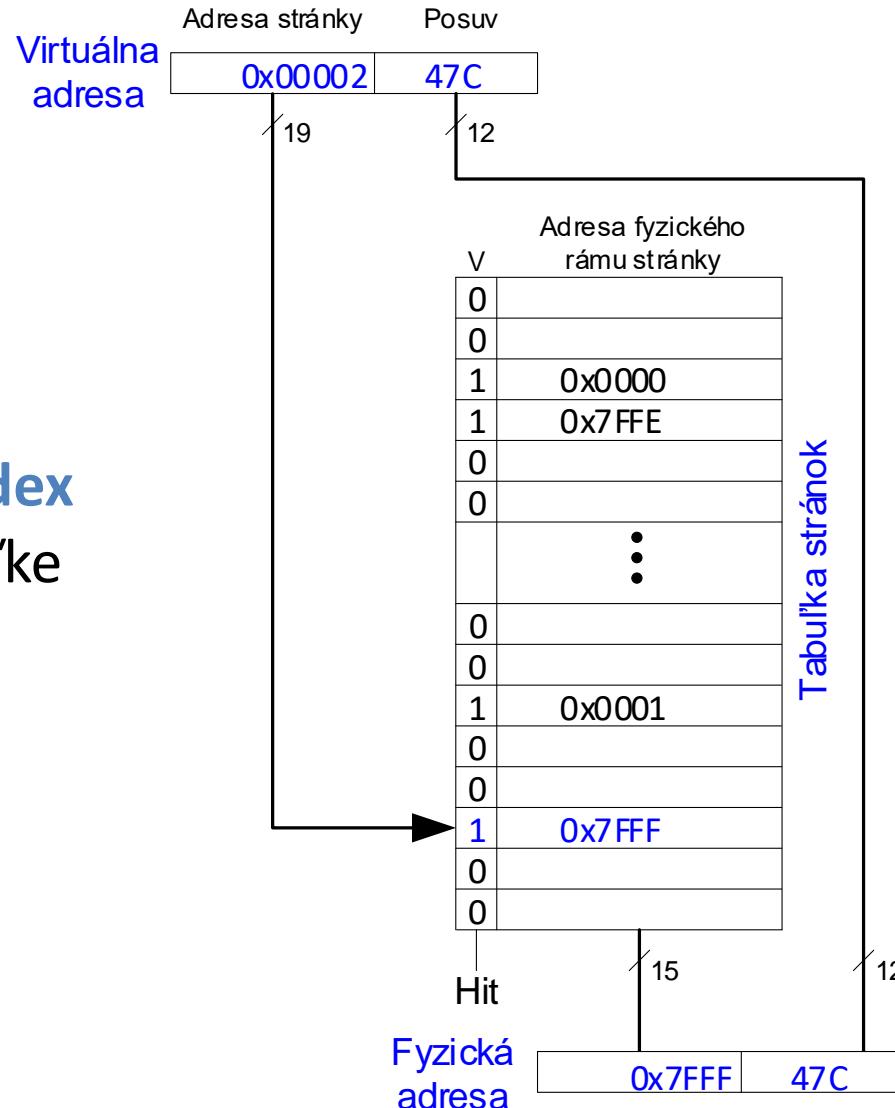
© 2007 Elsevier, Inc. All rights reserved.

Tabuľka stránok

- Priebežný stav pamäte ČP v procese mapovania je registrovaný pomocou **tabuľky stránok**.
- Jej základná funkcia vychádza z potreby priebežne registrovať prítomnosť kópií stránok premiestňovaných z priestoru virtuálnej pamäte do priestoru fyzickej pamäte ako stránkových rámov a opačne.

Tabuľka stránok

Adresa stránky funguje ako **index** položky v tabuľke stránok



Príklad 1

Určte fyzickú adresu zodpovedajúcu virtuálnej adrese **0x5F20**.

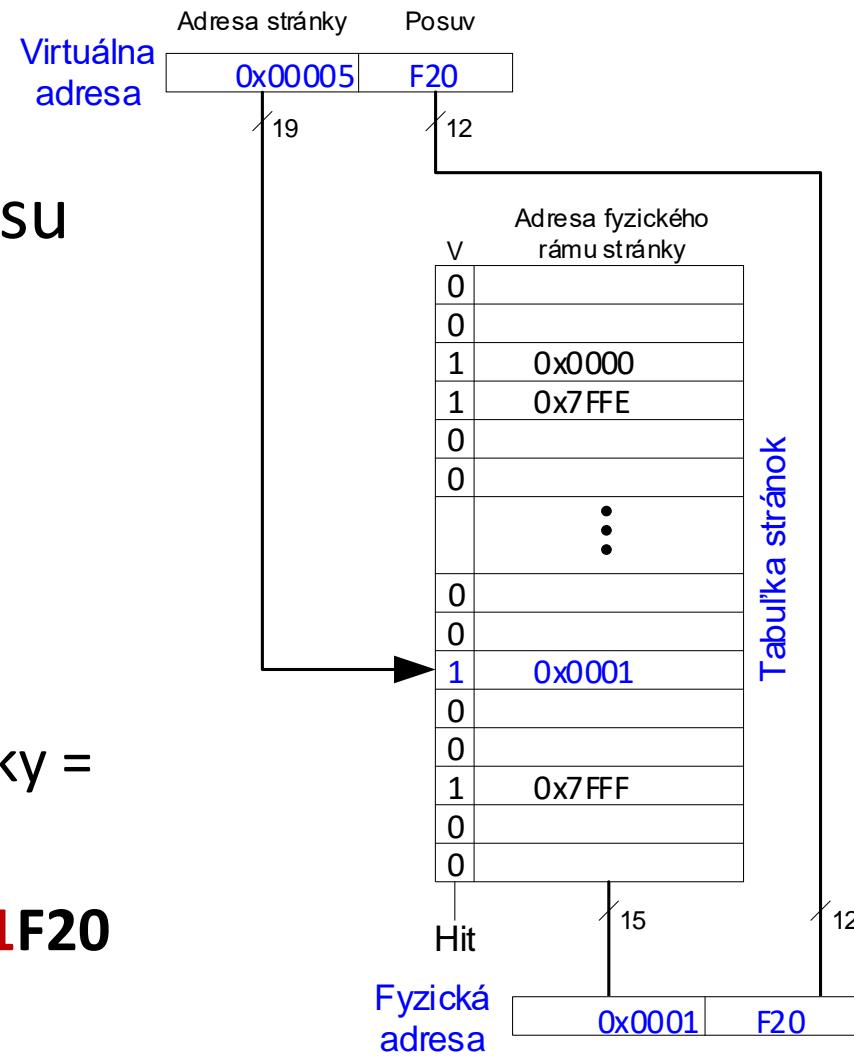
V	Adresa fyzického rámu stránky
0	
0	
1	0x0000
1	0x7FFE
0	
0	
	⋮
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

Tabuľka stránok

Príklad 1

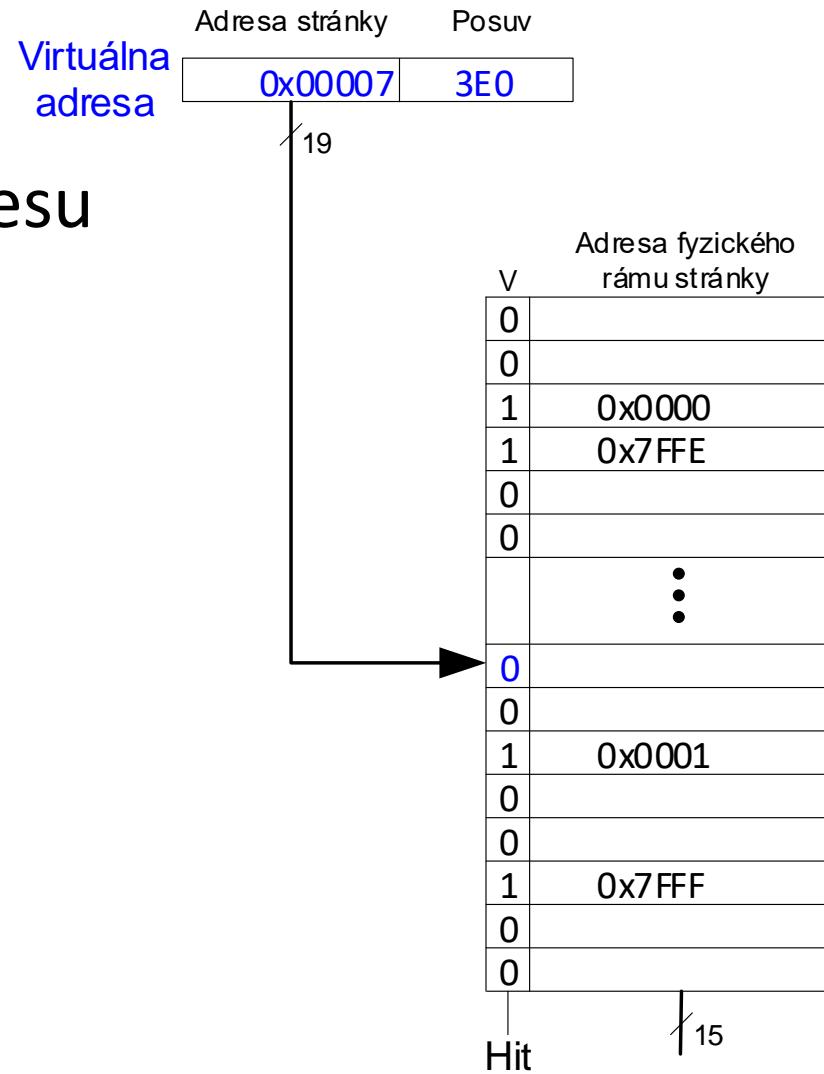
Určte fyzickú adresu zodpovedajúcu virtuálnej adrese 0x5F20.

- Adresa stránky = 5
- Pod indexom 5 => adresa rámu stránky = **1**
- Fyzická adresa: **0x1F20**



Príklad 2

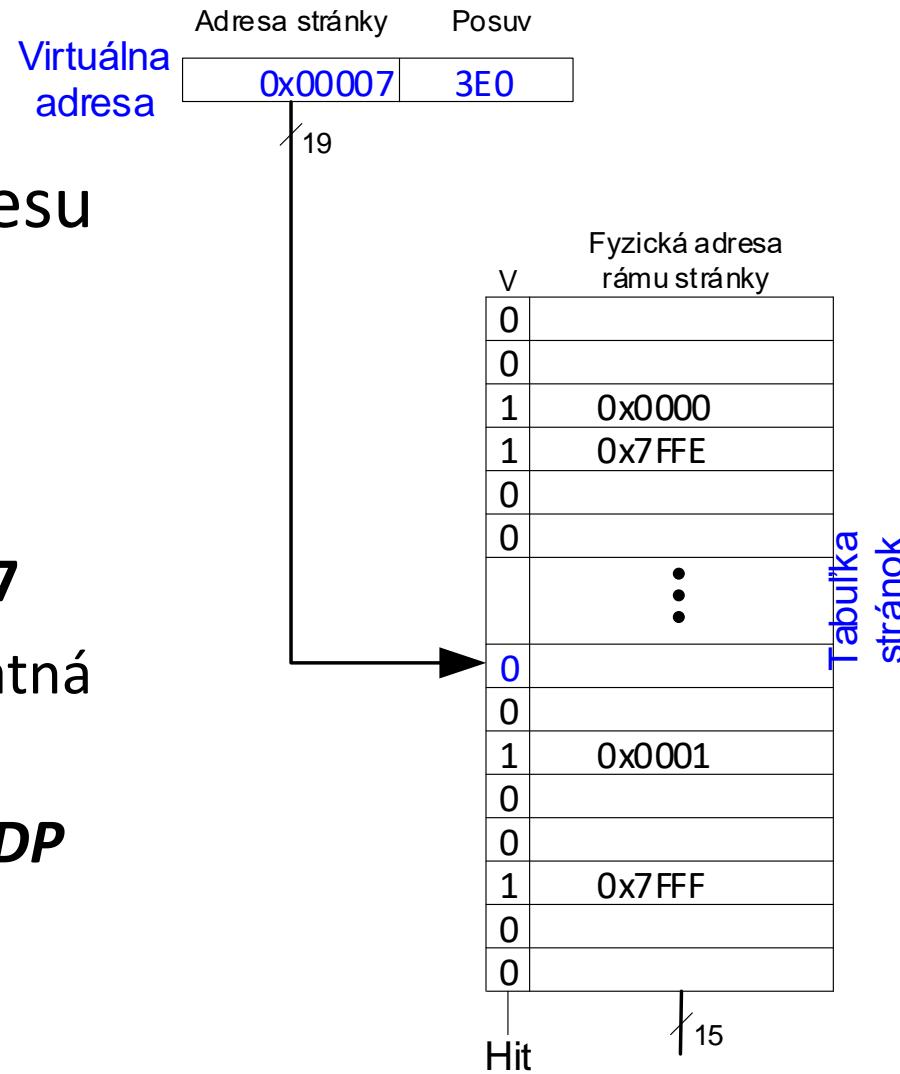
Určte fyzickú adresu
zodpovedajúcu
virtuálnej adrese
0x73E0?



Príklad 2

Určte fyzickú adresu zodpovedajúcu virtuálnej adrese **0x73E0?**

- Adresa stránky = 7
- Položka 7 je neplatná
- Stránka musí byť načítaná **do HP z DP**

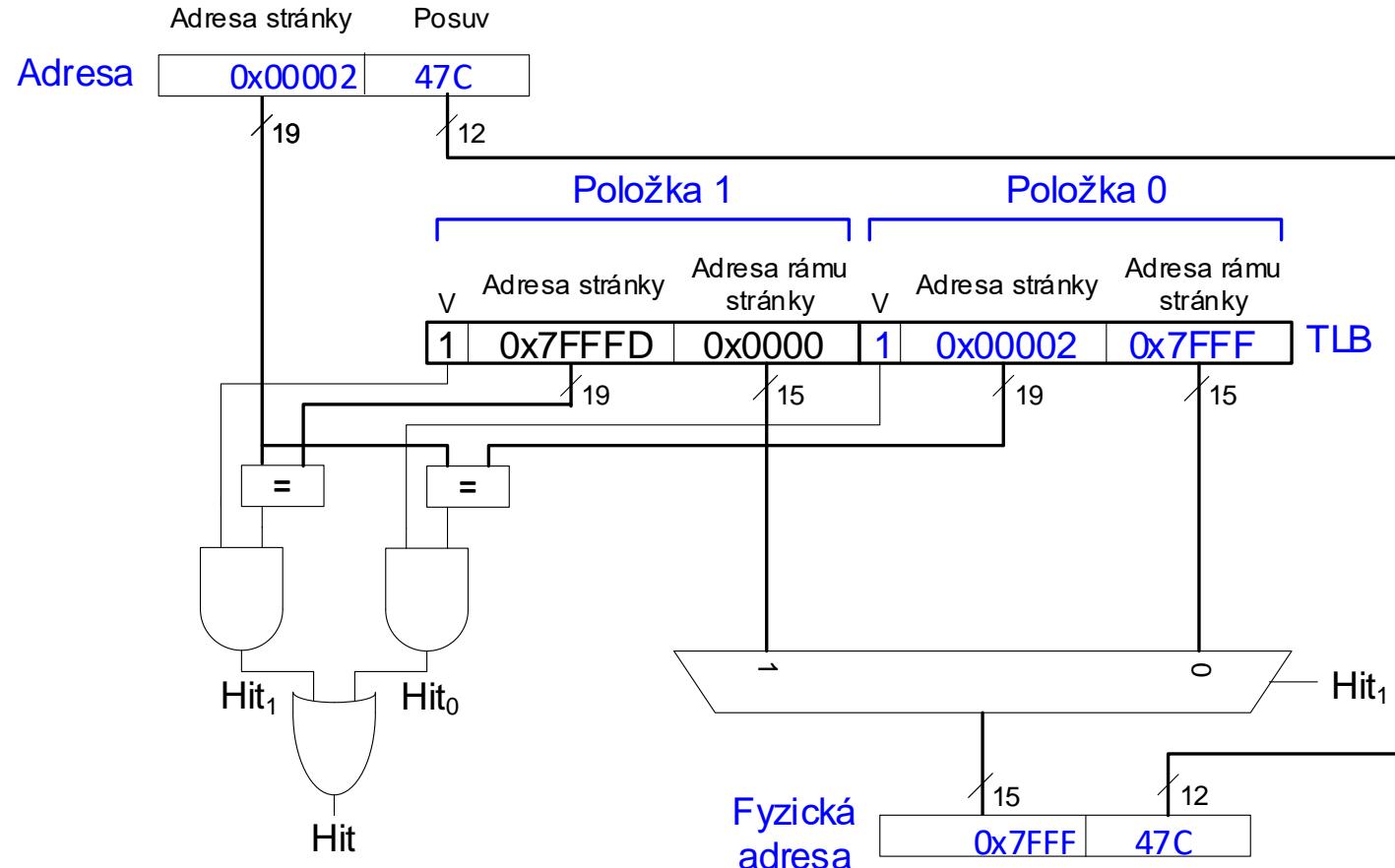


Pamäť preložených adres (TLB)

- **Translation Lookaside Buffer (TLB)**
 - Nedostatkom mapovania pamäte virtuálneho priestoru (ViP) pomocou TS je pomalé vykonávanie mapovania v dôsledku dvojnásobného prístupu do HP pri každom výpočte FA.
 - Východisko poskytuje mapovanie prostredníctvom pamäte preložených adres (TLB), v ktorej sú uložené adresy najčastejšie používaných stránok v práve prebiehajúcim intervale spracovania programu.
 - Je to asociatívna pamäť

- Z hľadiska prístupu
 - hlavne temporálna lokalita
- TLB
 - Nízka prístupová doba ~ 1 cyklus
 - Typicky 16 až 512 položiek
 - Redukuje # prístupov do HP v prípade väčšiny inštrukcií lw/sw

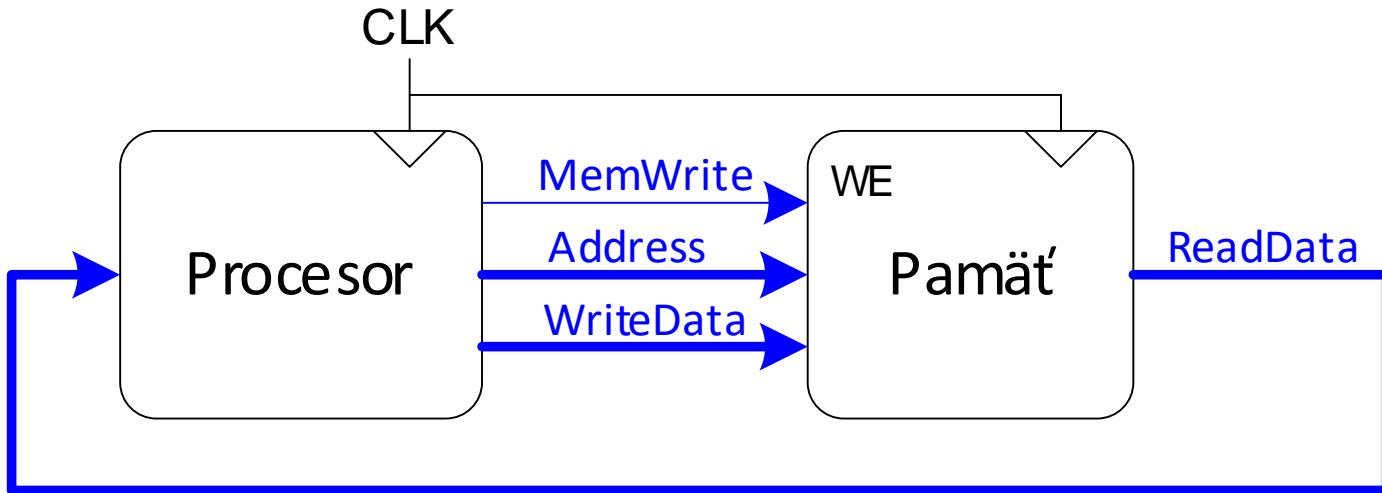
Ukážka TLB



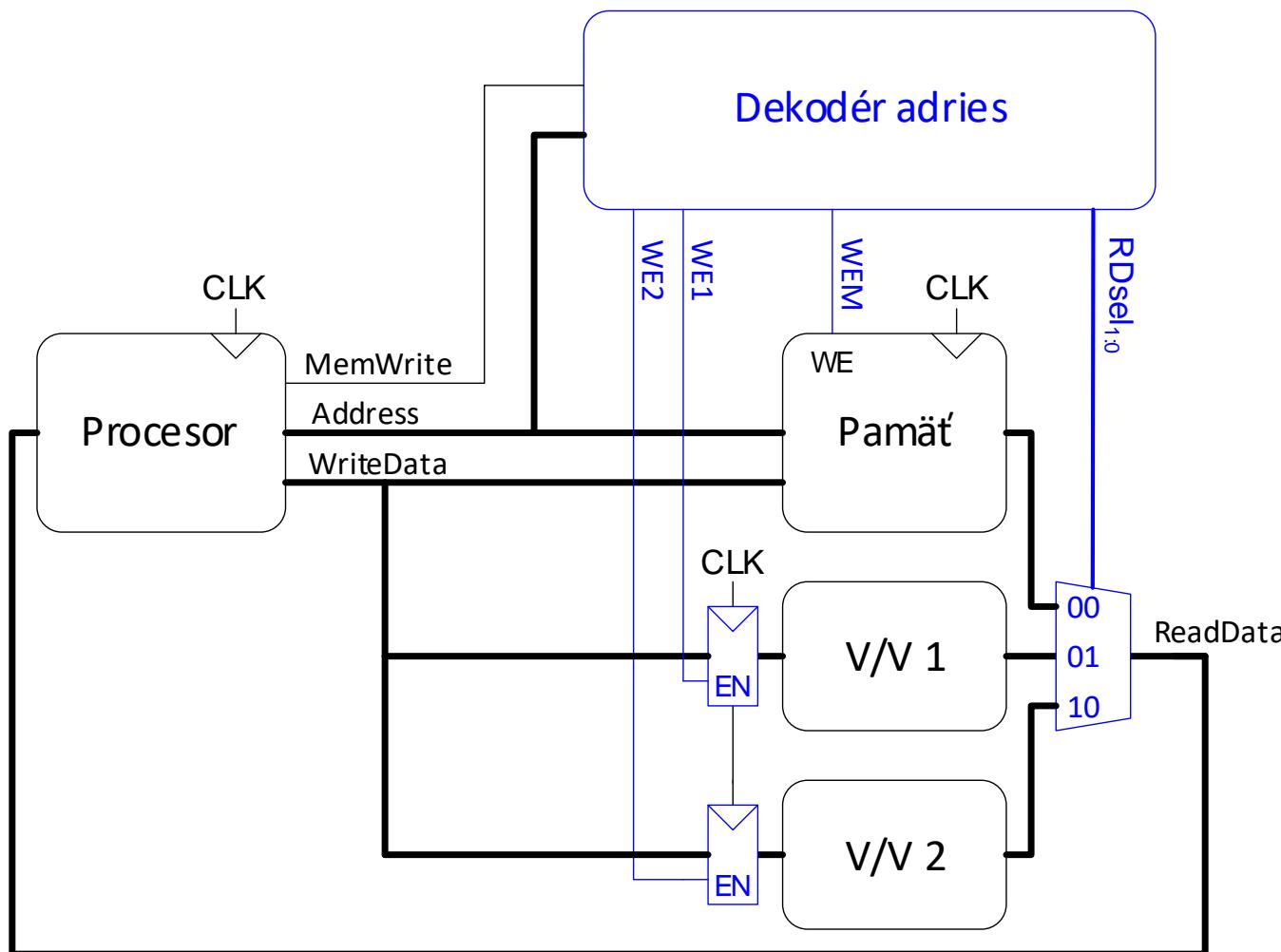
Vstupno-výstupný podsystém

- Procesor pri práci s V/V podsystémom uplatňuje rovnaký princíp sprístupňovania ako v prípade pamäťového podsystému
- Každé V/V zariadenie má svoju adresu
- Čítanie/zápis z/do V/V zariadenia vyžaduje adresu V/V zariadenia
- Časť adresného priestoru je vyhradená pre V/V zariadenia

Rozhranie pamäťového podsstému



Rozhranie V/V podsstémumu



Komponenty V/V podsystému

- **Dekodér adres**
 - Zistuje adresu zariadenie (pamäť, V/V) ktoré má komunikovať s procesorom
- **V/V registre:**
 - Obsahujú dátá určené pre V/V zariadenia
- **Multiplexor ReadData:**
 - Zabezpečuje smerovanie medzi V/V zariadením, prípadne pamäťou a procesorom

V/V podsystém

- V/V vložených systémov
 - MCU
 - GPIO porty, sériový port, A/D prevodníky, ...
- V/V „klasických“ počítačov
 - USB
 - PCI/PCIe
 - DDR
 - TCP/IP
 - SATA
 - etc.

Pozri kapitolu 8.6 a 8.7

Referencia

- David Money Harris and Sarah L. Harris,
Digital Design and Computer Architecture,
Chapter 8: Memory and I/O Systems, Second
Edition © 2012 by Elsevier Inc.