zmeny od predchádzajúcej verzie (nie všetky len na tieto si pamätám) :

- pravidlá kreslenia schém pri hradlách
- digitálna abstrakcia sa zmenila na digitálnu disciplínu
- pridaná číselná sústava - dvojkový komplement
- pridaná definícia bitu, nibble a byte
-

Topic 1. Od nuly k jednotke

1.1. Princípy návrhu číslicových systémov (abstrakcia, disciplína, kompozícia, hierarchia, modularita, jednotnosť). Charakteristika číslicových systémov. Pozičné číselné sústavy, diskrétna reprezentácia čísel.

1.1.1.a) Abstrakcia

- skrývanie detailov keď nie sú dôležité
- na systém sa dá pozerať z viacerých úrovní (levelov)
- význam abstrakcie je v tom, že nemusíme vedieť pri práci s jednou úrovňou o úrovniach nad a podňou (aj keď je dobré o nich niečo vedieť).

Príklad 1.

Levely abstrakcie v počítačových systémoch:
*1. Na najnižšom levely abstrakcie je **fyzika** - pohyby elektrónov*
*2. Systém je skonštruovaný pomocou **elektronických súčiastok** ako napríklad tranzistorov...*
*3. **Analógové obvody** spájajú elektronické súčiastky do komponentov.*
(Ich vstupom a výstupom sú
*4. **Digitálne obvody** ako napríklad logické členy(logic gates) pomáhajú rozdeliť napätia na 2 skupiny 0 a 1..*
*5. V **logickom dizajne** vytvárame komplexnejšie štruktúry z digitálnych obvodov - ako napríklad sčítačku (kombinačný logický obvod, ktorý realizuje sčítanie čísel v binárnej sústave).*
*6.**Mikroarchitektúra** spája architektonický a logický level abstrakcie*
*7. **Architektúra** popisuje počítač z pohľadu programátora).*
*8. **Operačný systém** má na starosti nízkoúrovňové detaily ako napríklad spravovanie pamäte.*
*9. **Aplikačný softvér** (aplikácia) je program ktorý umožňuje používateľovi pri uskutočňovaní činností určitého typu napríklad pri manipulácii s textami, grafikou*

1.1.1.b) Disciplína

- zámerné obmedzenie návrhových možností tak aby sme mohli produktívnejšie pracovať na vyšších úrovniach abstrakcie

Príklad 2
Digitálna disciplína.

- digitálne obvody používajú diskrétne napätia 0,1 narozdiel od analógových, ktoré používajú nepretrité (s oveľa väčším rozsahom), a preto sú vlastne digitálne obvody podmožinou analógových, a preto by

mali byť menej schopné ako analógové

- na druhú stranu digitálne obvody sa omnoho ľahšie konštruujú, a preto keď sa limitujeme na používanie digitálnych obvodov môžeme ľahko vytvárať zložité systémy, ktoré ľahko prekonajú svojich analógových predchodcov

- príklady: mobily, televízie, CD,

1.1.1.c) 3- Y (po Slovensky to je skôr ia,ta a sť).

- tieto princípy sa dajú aplikovať aj na softvérový aj hardvérový vývoj

Hierarchia

- princípom hierarchie je roztriedenie systému na stále menšie a menšie časti až kým nie sú jednoduché na pochopenie

Modularita

- jej zmyslom je to, že pre systém by mali byť navrhnuté vhodné rozhrania, funkcie tak aby sa dali jednoducho pospájať tak aby bolo predídené možným problémom

Pozn.

- *je to stupeň na ktorom by mali byť komponenty systému separované a znovupoužiteľné.*

Jednotnosť

- hľadanie jednotnosti v moduloch - hľadanie znovupoužiteľných častí systému
- znižuje počet modulov, ktoré musia byť navrhnuté

1.1.1.d) Kompozícia.

- správne spojenie komponentov do jedného celku
- jednotlivé komponenty musia byť na správnom mieste

1.1.2.a) Charakteristika číslicových systémov

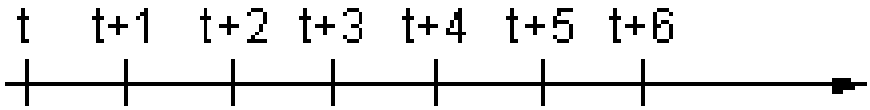
http://aplo.fiit.stuba.sk/aps/frames/generuj.php?id=8

Číslicový systém chápeme ako formálny model, ktorý možno definovať nad reálnym (napr. elektronickým) číslicovým zariadením. Formálny model opisuje toto zariadenie pri danom stupni abstrakcie.

Číslicový systém je dynamický systém, ktorý možno charakterizovať týmto spôsobom:

-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.



Číselné sústavy:
1.1.2 b) Pozičné číselné sústavy

- číselná sústava je spôsob, akým sú zapisované čísla pomocou znakov (nazývaných číslice(cifry))
- z hľadiska určenia hodnoty čísla z daného zápisu rozlišujeme 2 základné typy číslicových sústav

a)nepozičné - nezáleží na tom v akom poradí je dané číslo napísané
Pr:
A = 10, B = 20, C = 4
CBCAA = 4 + 20 + 4 + 10 + 10 = 48 výhodou bolo jednoduché spočítavanie a odčitavanie.

b) pozičné - záleží na tom v akom poradí je dané číslo napísané hodnota každej číslice je daná jej pozíciou v sekvencii symbolov
- kľučovou charakteristikou pozičných sústav je ich základ (báza), zvyčajne je to prirodzené číslo väčšie ako 1.
- vznikli spolu s vynájdením 0.

Diskrétna reprezentácia čísel
- zobrazuje hodnoty, ktoré môže pozícia nadobúdať

napríklad v desiatkovej od 0 - 9 v binárne 0,1...

Všeobecný spôsob na prevod medzi sústavami (z desiatkovej do x - kovej):
- prevádzané číslo predelím základom pozn. musí to byť celočíselné delenie (zvyšok sa zapisuje odzadu).
- výsledok delenia použijeme v ďalšom kroku
- kroky opakujeme až kým nie je výsledok delenia rovný nule

Desiatková sústava

- desiatková sústava používa 10 číslic : 0,1,...,9
- hodnoty čísel sú sprava doľava:
- desiatková sústava má základ 10
- dá sa zapísať ako súčet všetkých: (číslic * váha jednotlivého stĺpca)

Všeobecne:
$číslo_{10} = \sum_{n=0} x_{n+1} * 10^n$ = toto je vtip hehe nepísať do testu

Príklad:
$1574_{10} = 4 * 10^0 + 7 * 10^1 + 5 * 10^2 + 1 * 10^3$

Dvojková sústava

- dvojková sústava používa 2 číslice : 0,1
- hodnoty čísel sú sprava doľava:
- dvojková sústava má základ 2
- dá sa zapísať ako súčet všetkých: (číslic * váha jednotlivého stĺpca)

Všeobecne:

$číslo_2 = \sum_{n=0} x_{n+1} * \ 2_{10}^{n}$ = opakovaný vtip je vtipom$_{10}$

Príklad:

$0101_2 = 1 * 2^0 + 0 * 2^1 + 1 * 2^2 + 0 * ^3 = 5_{10}$

Šestnástková sústava

- šestnástková sústava používa 10 číslic : 0,1,...,9 a 6 písmen: A,B,...,F
- hodnoty čísel sú sprava doľava:
- šestnástková sústava má základ 16
- dá sa zapísať ako súčet všetkých: (číslic * váha jednotlivého stĺpca)

pričom ak 10...15 substituujeme na A...F
tak:

$číslo_{16} = \sum_{n=0} x_{n+1} * \ 16_{10}^{n}$ = Dobré vtipy neomrzia$_{10}$.

Príklad:

$AE_{16} = 14 * 16^0 + 10 * 16^1 = 174_{10}$

Spätný prevod:

$127_{10} = 7F_{16}$

127 : 16 = 7 zv 15

7 : 16 = 0 zv 7

$15_{10} = F_{16}$

- v digitálnych systémoch sa najviac používa binárna a hexadecimálna číslicová sústava

Dvojkový komplement

- kladné číslo je také ako v normálnom binárnom tvare
- záporné číslo získame tak, že znegujeme všetky bity a pripočítame 1.

Bit, Byte Nibble

- bit je základná a nejmenšia jednotka dát 0,1
- nibble = 4bity
- byte = 8 bitov

1.2. Stavebné prvky číslicových systémov, hradlá. Spojité veličiny: základ

číslicových systémov. Tranzistory typu CMOS.

1.2.1.a) Stavebné prvky číslicových systémov

- digitálne, alebo číslicové systémy sú systémy, ktoré vykonávajú operácie na binárnych premenných

1.2.1.b) Hradlá

- logické hradlá sú jednoduché digitálne obvody, ktoré vezmú jeden alebo viac binárnych vstupov a vyprodukujú binárny výstup

- zakresľujú sa tak, že vidíme vstup(vstupy), kreslíme ich vľavo(alebo hore) a výstupy vpravo(dole)

- pri vstupe sa píše nejaké písmeno zo začiatku abecedy pri výstupe Y.

-drôty sa spájajú pomocou bodky, drôty bez bodky nie sú prepojené

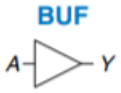
- vzťah medzi vstupmi a výstupom sa dá popísať tabuľkou pravdivostných hodnôt alebo Booleovskou funkciou

NOT hradlo

$Y = \overline{A}$

A	Y
0	1
1	0

BUFFER hradlo



$Y = A$

A	Y
0	0
1	1

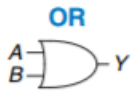
AND hradlo

$Y = AB$

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

OR hradlo

Y = A + B booleovská funkcia



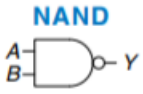
$Y = A + B$

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

Hradlá s dvoma vstupmi

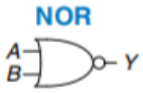
$Y = A \oplus B$

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



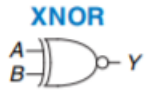
$Y = \overline{AB}$

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



$Y = \overline{A + B}$

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

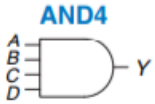


$Y = \overline{A \oplus B}$

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

Hradlá s viac než 2 vstupmi

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



$Y = ABCD$

1.2.1.c) Spojité veličiny, základ číslicových systémov

- číslicové systémy používajú diskrétne premenné, avšak premenné sú reprezentované spojitými fyzikálnymi hodnotami ako napríklad napätie na drôte

- preto sa musí nájsť cesta ako reprezentovať binárny signál A

- napríklad ak 0V potom A = 0

a 5V potom A = 1

- 0 a 1 sú takzvané logické úrovne

šum

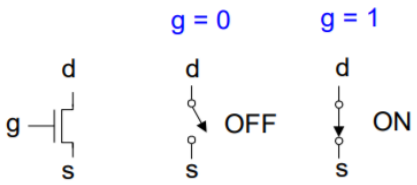
- je to všetko čo degraduje signál

- každý reálny systém musí prijať nejaký šum tak napríklad 4.97 V by malo byť prezentované ako 1 ale čo 4.3V alebo 2.5?

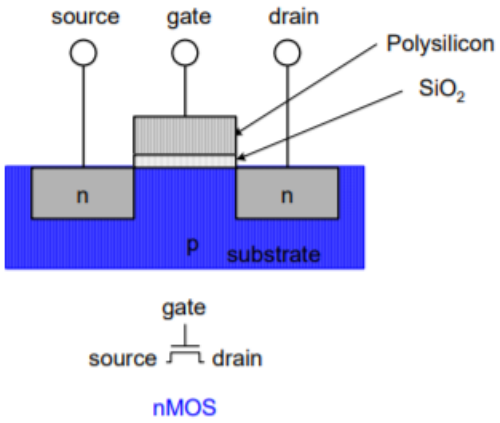
1.2.1.d) cMos tranzistory

- hradlá sú vytvorené z tranzistorov

- tranzistor je elektricky ovládateľný spínač, ktorý mení stav 0 alebo 1 ak je na ovládací terminál privedený prúd



- MOS tranzistory

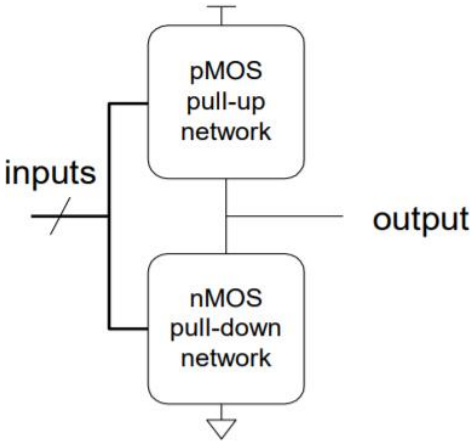


- nMOS dobre prepúšťa 0

- pMOS dobre prepúšťa 1

cMOS

- schéma cMOS



- MOSFET = Metal + Oxide + Semiconductor + Field effect transistor

- je to spôsob vytvárania logických členov, ktorou sa vyrába väčšina logických integrovaných obvodov

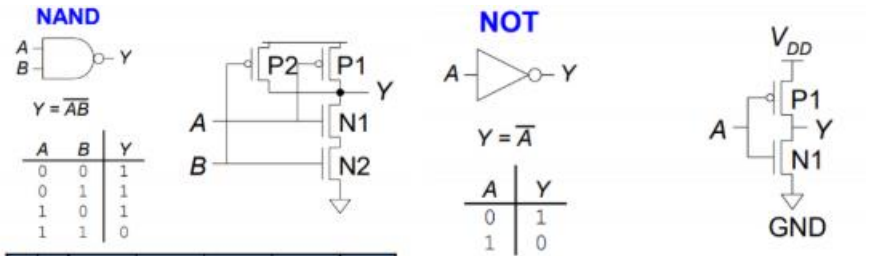
- nízka spotreba a vysoká odolnosť voči šumu

- z každého páru tranzistorov MOSFET z ktorých je zložené hradlo je vždy jeden v nevodivom stave

ZHRNUTIE

- reálny svet je analógový, obmedzili sme sa ale na digitálny, aby sa mohli vyvíjať zložitejšie systémy
- binárne premenné majú 2 stavy 0 a 1, HIGH a LOW, TRUE a FALSE

- hradlá sa väčšinou vytvárajú z cMOS tranzistorov, ktoré sa správajú ako elektrinou kontrolované prepínače
- nMOS tranzistory, ktoré dobre prepúšťajú 0 sa zapnú ak je hradlo 1
- pMOS tranzistory, ktoré dobre prepúšťajú 1 sa zapnú ak je hradlo 0
cMOS = nMOS + pMOS



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

Y = $\overline{A}$

A	Y
0	1
1	0

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

Topic 2 Počítačová aritmetika

2.1 Opis algoritmu celočíselného násobenia + ilustračný príklad. Opis algoritmu sčítania reálnych čísel vyjadrených v pohyblivej rádovej čiarke + ilustračný príklad.

- nech A[n-1:0], B[n-1:0],Q[n-1:0],C[1] a PC sú registre. Nech PC slúži ako počítadlo cyklov. Zaveďme C#Q#A na označenie zreťazenia registrov.

Potom:

Algoritmus celočíselného násobenia

```
BEGIN
  C#Q := 0 || PC := n
  WHILE (PC > 0)
    IF A(0) = 1 THEN
      C#Q := Q + B
    ENDIF
    C#Q#A := 0 # SR1(C#Q#A) ||
      PC := PC – 1
  END
END
```

Kroky celočíselného násobenia $(+13)_{10} \times (-5)_{10}$ v BinČS

- $A = |(+13)_{10}| = |(1101)_2|, A_z = 0; B = |(-5)_{10}| = |(0101)_2|, B_z = 1$
- Znamienko výsledku = $A_z \oplus B_z = 0 \oplus 1 = 1$

C	#	Q	#	A		PC	Poznámka
0		0000		1101		4	C#Q := 0 || PC := $\lceil \log_2(\max(\lvert A\rvert;\lvert B\rvert)+1) \rceil$
	+	0101					A(0) = 1 ⇒ C#Q := Q + B
0		0101		1101			
0		0010		1110	1	3	C#Q#A := 0 # SR1(C#Q#A) || PC := PC – 1
0		0001		0111	0	2	C#Q#A := 0 # SR1(C#Q#A) || PC := PC – 1
	+	0101					A(0) = 1 ⇒ C#Q := Q + B
0		0110		0111			
0		0011		0011	1	1	C#Q#A := 0 # SR1(C#Q#A) || PC := PC – 1
	+	0101					A(0) = 1 ⇒ C#Q := Q + B
0		1000		0011			
0		0100		0001		0	C#Q#A := 0 # SR1(C#Q#A) || PC := PC – 1

- popis algoritmu

A = A(n-1)A(n-2)...A(0)
Ak C = 0

Q = 0000
a B = 1011

tak Q + B = C#Q = 01011

1. Reťazec C = 0 a Q = 0000 PC = n
2. Kým je (PC > 0)
AK A(0) = 1 POTOM
reťazec C#Q = Q + B
POSUN retazca C#Q#A dolava
pocitadlo = pocitadlo - 1
KONIEC

Sčítanie reálnych čísel vyjadrených v pohyblivej rádovej čiarke

https://moodle.fei.tuke.sk/pluginfile.php/44868/mod_resource/content/6/Floating-point%20addition.pdf

2.2 Opis algoritmu celočíselného delenia + ilustračný príklad. Opis algoritmu násobenia reálnych čísel vyjadrených v pohyblivej

rádovej čiarke + ilustračný príklad.

Celočíselné delenie

https://moodle.fei.tuke.sk/pluginfile.php/44867/mod_resource/content/8/Unsigned%20Division.pdf

Násobenie reálnych čísel vyjadrených v pohyblivej rádovej čiarke

https://moodle.fei.tuke.sk/pluginfile.php/44865/mod_resource/content/5/Floating-point%20multiplication.pdf

2.3 Opis Boothovho algoritmu + ilustračný príklad. Opis algoritmu násobenia reálnych čísel vyjadrených v pohyblivej rádovej

čiarke + ilustračný príklad.

https://moodle.fei.tuke.sk/pluginfile.php/44866/mod_resource/content/7/The%20Booth%20algorithm.pdf

- OPIS:

- boothov algoritmus sa správa ku kladným a k záporným číslam rovnako

- funguje na tom ,že reťazce 0 a 1 v multiplikátore nepotrebujú sčítavanie len posúvanie.

Multiplikand * Multiplikátor = výsledok

TOPIC 3 Kombinatorický logický dizajn

3.1 Boolove funkcie, rovnice (disjunktívna, konjunktívna forma). Boolova algebra, axiómy, teorémy, pravidlá. Karnaughové

mapy a pravidlá minimalizácie.

- logické obvody sa skladajú z
- vstupov, výstupov, časových a funkcionálnych špecifikácií
funkcionálne špecifikácie - vzťah medzi vstupmi a výstupmi
časové špecifikácie - čas kým sa vstupy zmenia na výstupy odpovedia
- existujú kombinatorické a sekvenčné logické obvody

3.1.a) Boolove funkcie

- zaoberajú sa s premennými, ktoré sú buď Pravdivé alebo Nepravdivé takže sú vhodné na popis digitálnej logiky

Termíny používané pri boolovských funkciách

komplement A = A'
literal - premenná alebo jej komplement
implikant - súčin všetkých literálov
minterm - logický súčin všetkých vstupov funkcie napriklad A * B (AND)
maxterm - logický súčet všetkých vstupov funkcie napríklad A + B (OR)
poradie operácií - NOT, AND, OR
príklad Y = A + BC sa číta ako A OR (B AND C)
alebo Y = A´B + BCD´
(NOT(A) AND B) OR (B AND C AND (NOT D))
- disjunktívna a konjunktívna forma

Disjunktívna forma
- alebo aj suma súčinov
- každý riadok pravdivostnej tabuľky obsahuje tzv. minterm alebo konjunkciu všetkých premenných
- vieme napísať Boolovskú funkciu pre každú tabuľku pravdivostných hodnôt keď sčítame všetky mintermi pre každý výstup kedy je Y pravdivý
- napríklad v pravdivostnej tabuľke 2.8 je 1 len v druhom riadku a preto
Y = A´B.

A	B	Y	minterm	minterm name
0	0	0	$\overline{A}\ \overline{B}$	m_0
0	1	1	$\overline{A}\ B$	m_1
1	0	0	$A\ \overline{B}$	m_2
1	1	0	$A\ B$	m_3

Figure 2.8 Truth table and minterms

- pre tabuľku 2.9
Y = A´B + AB

A	B	Y	minterm	minterm name
0	0	0	$\overline{A}\,\overline{B}$	m_0
0	1	1	$\overline{A}\,B$	m_1
1	0	0	$A\,\overline{B}$	m_2
1	1	1	$A\,B$	m_3

Figure 2.9 Truth table with multiple TRUE minterms

F(A,B)= Σ(m1, m3) - iný spôsob zápisu

Konjunktívna forma
- alebo aj sučín súm
- každý riadok pravdivostnej tabuľky obsahuje tzv. maxterm alebo disjunkciu všetkých premenných
- vieme napísať Boolovskú funkciu pre každú tabuľku pravdivostných hodnôt keď vynásobime všetky maxtermi pre každý výstup kedy je Y nepravdivý

A	B	Y	maxterm	maxterm name
0	0	0	$A + B$	M_0
0	1	1	$A + \overline{B}$	M_1
1	0	0	$\overline{A} + B$	M_2
1	1	1	$\overline{A} + \overline{B}$	M_3

Figure 2.13 Truth table with multiple FALSE maxterms

F(A,B)= Π(M0, M2)
alebo Y = (A + B)(A´+B)

3.1.b) Boolova algebra

- keď zapíšeme Boolovské rovnice pomocou tabuľky pravdivostnej hodnoty nie je zaručené, že tieto rovnice budú viesť k najjednoduchšej podobe množiny hradiel
- pomocou Boolovskej algebry vieme tieto rovnice zjednodušiť
- pravidlá Boolovskej algebry sú v mnohých prípadoch jednoduchšie, pretože premenné majú možné len 2 hodnoty
- boolovská algebra je založená na axiómoch, pri ktorých predpokladáme, že sú správne
- z týchto axiómov odvádzame všetky Vety Boolovskej algebry

AXIÓMY

Table 2.1 Axioms of Boolean algebra

	Axiom		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

- ak nahradíme • so znakom AND alebo + so znakom OR tvrdenie bude stále pravdivé
- A1 - hovorí že, ak premenná B je 0 ak nie je 1 a A1´ hovorí, že ak premenná nie je 1 tak je 0 spoločne to znamená, že pracujeme v Boolovskom alebo v binárnom poli nuliek a jednotiek.
- A2 a A2´ - definuje operáciu NOT - negácia
-A3 - A5 - definujú AND
A3´ - A5´ - definujú OR

TEORÉMY jednej premennnej

Table 2.2 Boolean theorems of one variable

	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$		Involution
T5	$B \bullet \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Complements

T1 - definuje, že pre všetky Boolovské funkcie B AND 1 = B
T1´ - definuje, že pre všetky Boolovské funkcie B OR 0 = B

- (T1) znamená to, že ak máme 2 vstupy B a 1 a máme hradlo AND tak môžme odstrániť hradlo AND a nahradiť ho drôtom, ktorý je pripojený k vstupu B

-(T1´) ak máme vstupy B a 0 a máme hradlo OR tak môžme odstániť hradlo a nahradiť ho drôtom, ktorý je pripojený k vstupu B

T2 - definuje, že pre všetky Boolovské funkcie B AND 0 = 0
T2´ - definuje, že pre všetky Boolovské funkcie B + 1 = 1

T2 - v hardvéri to znamená, že ak jeden zo vstupov hradla AND je 0 vieme nahradiť hradlo AND s drôtom, ktorý je naviazaný DOLE(0).

T2´ - v hardvéri to znamená, že ak jeden zo vstupov hradla OR je 1 vieme nahradiť hradlo OR s drôtom, ktorý je naviazaný HORE(1).

T3 - **Idempotencia** definuje, že premenná AND premenná je rovná samej sebe a rovnako aj OR(T3´)

v hardvéri:

T4 - ***Involúcia*** definuje, že komplement komplementu je premenná sama osebe

T5 - komplementárny teorém hovorí, že premenná a(AND) jej komplement = 0 , pretože jedna z nich musí byť 0
naopak T5´ definuje, že premenná alebo(OR) jej komplement je 1, pretože jedna z nich musí byť 1.

Hardvér

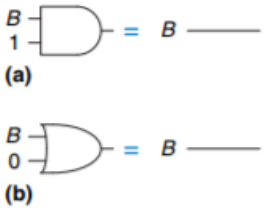


Figure 2.14 Identity theorem in hardware: (a) T1, (b) T1′

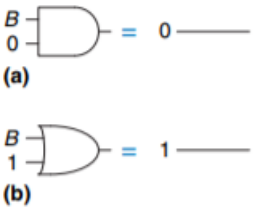


Figure 2.15 Null element theorem in hardware: (a) T2, (b) T2′

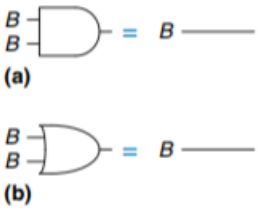


Figure 2.16 Idempotency theorem in hardware: (a) T3, (b) T3′

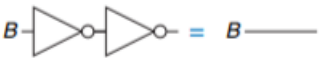


Figure 2.17 Involution theorem in hardware: T4

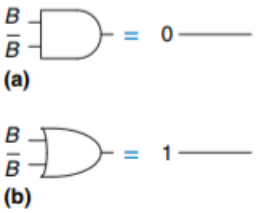


Figure 2.18 Complement theorem in hardware: (a) T5, (b) T5′

Teorémy viac premenných

- teorémy T6 - T12 popisujú ako z+,jednodušíme rovnice viacerých premenných

Table 2.3 Boolean theorems of several variables

	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D) = B \bullet C + \overline{B} \bullet D$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D) = (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots} = (\overline{B}_0 + \overline{B}_1 + \overline{B}_2 \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots} = (\overline{B}_0 \bullet \overline{B}_1 \bullet \overline{B}_2 \ldots)$	De Morgan's Theorem

T6 - komutatívnosť nezáleží na poradí vstupov
T7 - asociatívnosť to ako rozložíme vstupy pre funkcie AND a OR neovplyvňuje výsledok
T8 - distributívny teorém - "roznásobenie zátvorky" v normálnej algebre to platí len pre súčin ale v Boolovskej to platí aj pre súčet
T9 - T11 - ukazuje ako sa môžme zbaviť premenných navyše
- T12 De Morganove vzorce - komplement súčinu všetkých premenných je rovných súčtu je rovný súčtu komplementov každej jednej premennej a platí to aj naopak.

Elementárne logické funkcie
Logický súčin - AND
Logický súčet - OR
Negácia - NOT -

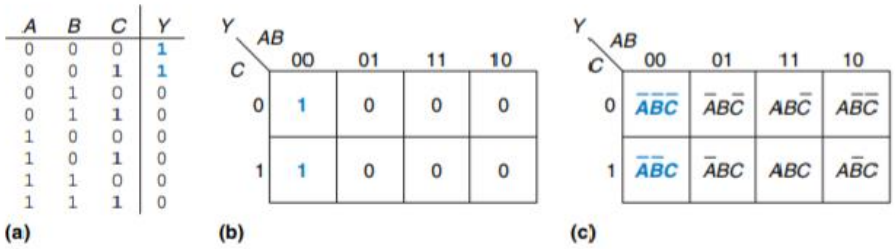
Odvodené (vektorové) logické funkcie
Negácia logického súčinu - NAND
Negácia logického súčtu – NOR
Nerovnosť - XOR; Rovnosť NXOR

Karnaughové mapy

- grafická metóda zjednodušovania Boolovských funkcií
- logická minimalizácia vyžaduje kombináciu členov napríklad

PA + PA´ = P

- Karnaughove mapy robia kombinovateľné členy ľahko viditeľné tým, že ich uložia vedľa seba v tabuľke



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

(a)

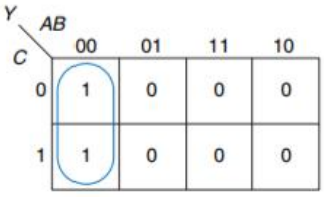
Y / C \ AB	00	01	11	10
0	1	0	0	0
1	1	0	0	0

(b)

Y / C \ AB	00	01	11	10
0	$\bar{A}\bar{B}\bar{C}$	$\bar{A}B\bar{C}$	$AB\bar{C}$	$A\bar{B}\bar{C}$
1	$\bar{A}\bar{B}C$	$\bar{A}BC$	ABC	$A\bar{B}C$

(c)

Figure 2.43 Three-input function: (a) truth table, (b) K-map, (c) K-map showing minterms



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

Figure 2.44 K-map minimization

- minimalizácia
Y = A´B´C´ + A´B´C = A´B´(C + C´) = A´B´

Pravidlá:

1. Použite čo najmenší kruhov na pokrytie všetkých jednotiek
2. Všetky štvorce v kruhu musia obsahovať jednotku
3. Počet štvorcov v kruhu musí byť mocninou dvojky
4. Kruhy musia mať maximálnu veľkosť
5. Kruhy môžu ísť aj okolo rohov Karnaughovej mapy
6. Jednotka v Karnaughovej mape môžu byť obsiahnuté viackrát v rôznych kruhoch, ak to zabezpečuje použitie menšieho počtu kruhov

3.2.a) Logické členy a schémy viacúrovňových obvodov

-LOGICKÉ ČLENY : viď 1.2.1 b)

SCHÉMY VIACÚROVŇOVÝCH OBVODOV

- logika v disjunktívnej forme sa nazýva dvojúrovňová logika, pretože obsahuje literály napojené na úroveň AND hradiel, ktoré sú napojené na úroveň OR hradiel
- často sa vytvárajú viac ako 2 levely logických hradiel
- niektoré logické funkcie potrebujú obrovské množstvo hardvéru pri vytváraní dvojstupňovej logiky
- príkladom je funkcia XOR viacerých premenných
I. Y = A´B´C + A´BC´ + AB´C´ + ABC
- táto rovnica sa nedá zjednodušiť avšak
II. (A ⊕ B) ⊕ C je omnoho jednoduchšia

Graficky:

I.

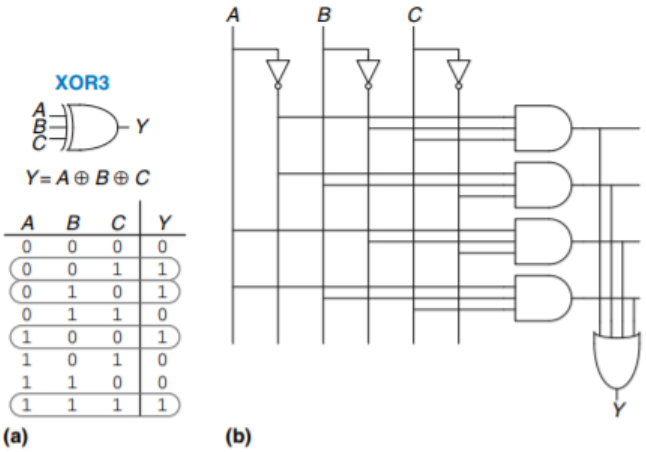


Figure 2.30 Three-input XOR: (a) functional specification and (b) two-level logic implementation

II.

Figure 2.31 Three-input XOR using two-input XORs

POZNÁMKA bubble pushing (Toto by som si pozeral nakoniec nie je to v žiadnej otázke)

používa sa na prekreslenie obvodov NOT bubliny sa medzi sebou zrušia

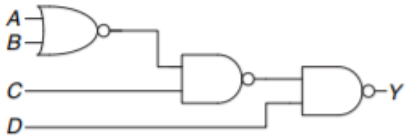


Figure 2.33 Multilevel circuit using NANDs and NORs

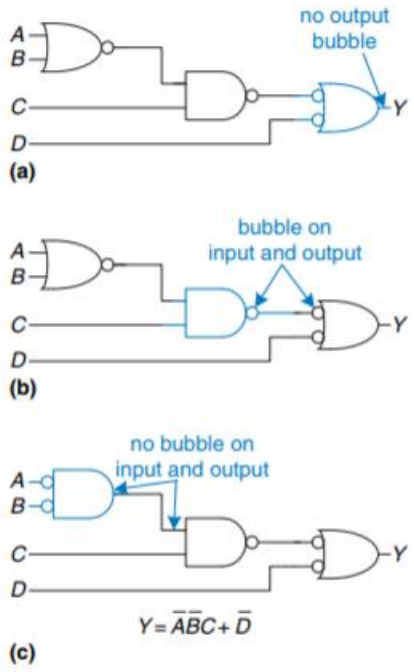
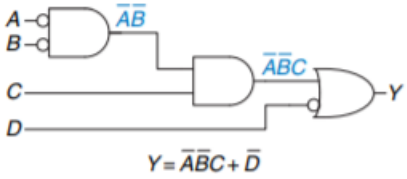


Figure 2.35 Logically equivalent bubble-pushed circuit



Pravidlá:

1. Začnite vo výstupe a potom smerom k vstupom
2. Posuňte bubliny z výstupu naspäť
3. Nakreslite hradlá tak aby ste bubliny zrušili

3.2.b) Viachodnotová logika a význam X a Z

- logický systém, ktorého výrazy nadobúdajú v interpretácii viac ako dve pravdivostné hodnoty

Význam X a Z

Zakázaná hodnota : X

- symbol X indikuje, že uzol obvodu má neznámu alebo zakázanu hodnotu.
- často sa to stáva, ak je uzol Y napojený aj na HIGH aj na LOW, táto situácia je považovaná za chybu, a preto sa jej musíme vyhýbať, túto situáciu nazývame spor (contention)
- spor často vedie, k poškodeniu a prehriatiu obvodu
- hodnoty X sú často používané aj pre neinicializovanú premennú
- niekedy sa symbol X používa na indikovanie premenných na ktorých nezáleží "don´t care values", v tabuľkách pravdivostných hodnôt

- ak sa tento symbol nachádza v tabuľke tak nie je dôležitý môže byť 0 alebo 1
- keď sa X nachádza v obvode, tak to znamená, že má neznámu alebo zakázanú hodnotu

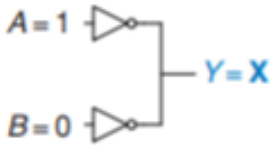


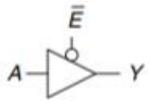
Figure 2.39 Circuit with contention

Význam: Z
- symbol Z indikuje, že uzol nie je ani v hodnote HIGH ani v LOW
- často sa vyskytuje mylné tvrdenie, že uzol Z je to isté ako 0 v skutočnosti môže byť jej hodnota 0,1 alebo medzi 0 a 1
- nieje priamo považovaný za chybu
- jednou z možností využitia je trojstavový buffer má tri možnosti výstupu 0, 1, Z ak je E = 1 tak obvod povoľuje výstup Z toto sa využíva v zberniciach aby mohol so zdieľanou pamäťou kedykoľvek ktokoľvek komunikovať
- viac rôznych vstupov, presne jeden je aktívny

Tristate Buffer

E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Figure 2.40 Tristate buffer



$\bar{E}$	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

Figure 2.41 Tristate buffer with active low enable

3.3.a) Kombinačné logické obvody

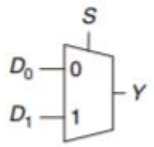
- kombinačné logické obvody sú často používané vo väčších stavebných blokoch v komplexnejších systémoch
- každý prvok obvodu je kombinačný
- neobsahuje cyklické cesty
- príkladmi sú sedemsegmentovka, plná sčítačka

MULTIPLEXORY
- patria medzi najviac používané obvody
- vyberajú si výstup z viacerých vstupov podľa hodnoty vybraného signálu

Príklady multiplexorov:

2:1 Multiplexor

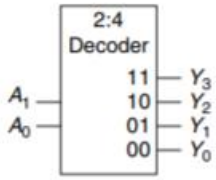
- S je takzvaný kontrolný signál, pretože kontroluje to čo multiplexor robí
- multiplexor si vyberá medzi dvojdátovými vstupmi podľa výberu:
  $S = 0$, $Y = D_0$
a ak $S = 1$, $Y = D_1$



S	D_1	D_0	Y
0	0	0	0
0	0	1	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

Figure 2.54 2:1 multiplexer symbol and truth table

DEKÓDERY
- n vstupov a 2^N výstupov
- výstupy sa volajú "one-hot", pretože práve jeden je (HIGH) v danom čase



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

Figure 2.63 2:4 decoder

3.3.b) Časové charakteristiky kombinačných logických obvodov

- jednou z najdôležitejších vecí pri vytváraní obvodov je to aby obvody bežali čo najrýchlejšie
- výstupu trvá nejaký čas, kým odpovie na zmenu vstupu

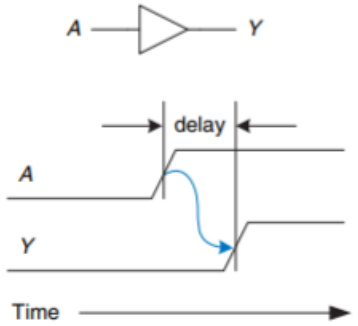
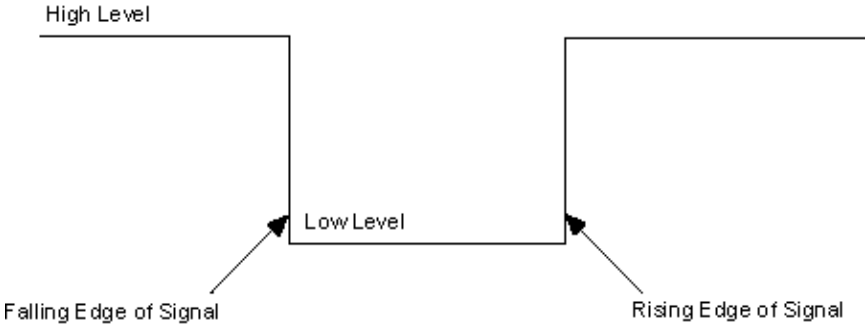


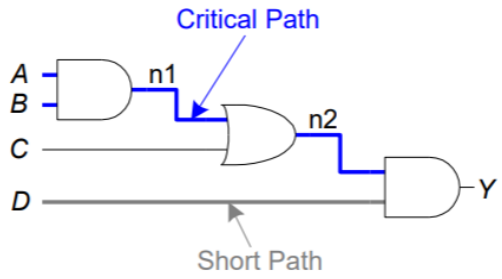
Figure 2.66 Circuit delay

- obrázok 2.66 zobrazuje oneskorenie medzi zmenou vstupu a ďalšou zmenou výstupu
- tento obrázok sa nazýva časový diagram
- miesto, kde sa mení LOW na HIGH sa nazýva "rising edge" (stúpajúca hrana)
  naopak zmena z HIGH na LOW sa nazýva "falling edge" (klesajúca hrana)



- propagation delay = t_{pd} = maximálne oneskorenie medzi vstupom a výstupom
  contamination delay = t_{cd} = minimálne oneskorenie medzi vstupom a výstupom
- oneskorenie je spôsobené odporom v obvode alebo rýchlosťou svetla

- dôvody prečo sa t_{pd} rôzni od t_{cd}
- viacero vstupov a výstupov niektoré sú rýchlejšie ako iné
- obvody sú rýchlejšie keď sa ochladzujú a naopak spomaľujú
- kritická cesta a najkratšia cesta



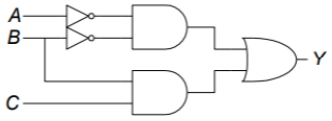
Critical (Long) Path: $t_{pd} = 2t_{pd\_AND} + t_{pd\_OR}$

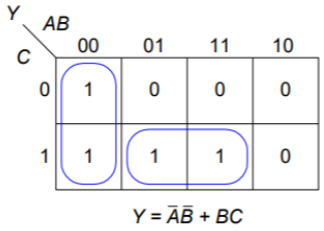
Short Path: $t_{cd} = t_{cd\_AND}$

3.3.c) Hazardy

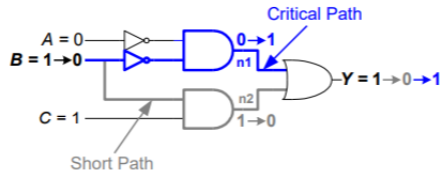
- zmena jedného vstupu spôsobí zmenu viacerých výstupov
- väčšinou nespôsobujú problémy je dôležité vedieť, že existujú

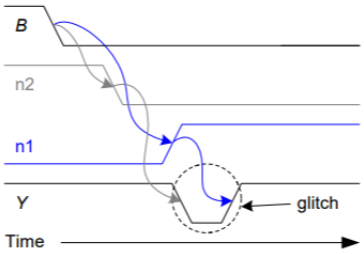
What happens when A = 0, C = 1, B falls?





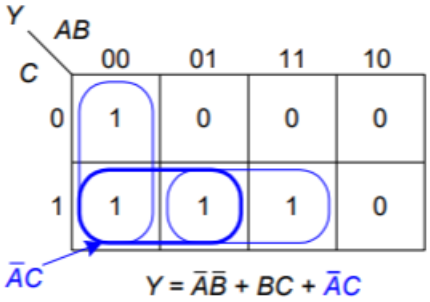
$Y = \bar{A}\bar{B} + BC$

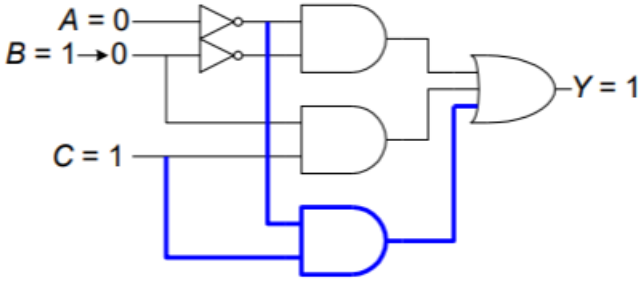




Oprava hazardov

Fixing the Glitch





- najjednoduchším spôsobom ako sa zbaviť hazardu je pridať redundantný(prvok navyše) prvok

ZHRNUTIE:

- digitálny obvod je modul so vstupmi a výstupmi, ktoré majú diskrétne hodnoty a časové a funkcionálne špecifikácie
- kombinačné obvody - ich výstup závisí od aktuálnych vstupov a výstupov
- funkcia kombinačných obvodov je daná tabuľkou pravdivostných hodnôt alebo Boolovskou rovnicou
- boolovské rovnice môžeme dostať z tabuľky pravdivostných hodnôt v disjunktívnom alebo konjuktívnom tvare
- boolovské rovnice vieme zjednodušovať pomocou pravidiel Boolovskej algebry
- boolovská algebra je postavená na axiómoch

- Karnaughova mapa je grafické zobrazenie na minimalizáciu funkcií do 4 premenných
- hradlá sú pospájané, tak aby vytvorili kombinačné obvody, ktoré vykonávajú definovanú funkciu
- každá funkcia v disjunktívnej forme vie byť skonštruovaná 2 úrovňovou logikou: (NOT, AND , OR) hradlá
- od funkcie závisí, či nie je vhodnejšia viacúrovňová logika, s viacerými typmi hradiel
- hradlá vedia byť skonštruované na vytváranie väčších obvodov napríklad multiplexorov alebo dekóderov
- multiplexor - umožňuje zlučovať niekoľko vstupov do jedného výstupu
- dekóder - nastaví jeden z výstupov na 1 (HIGH) a ostatné na 0 (LOW) v závislosti od vstupov
- časové špecifikácie:
  - propagationa a contamination delay (oneskorenie)
- z tejto špecifikácie vyplýva to ako rýchlo sa zmení výstup

propagation - najdlhší čas
contamination - najkratší čas

Topic 4

4.1 Stavebné prvky pamäťových blokov

Pamäte
- pamäťová bunka je základným stavebným blokom počítačovej pamäte
- pamäťová bunka je obvody, ktorý obsahuje bit binárnej informácie a musí byť nastavený na logickú 1 alebo logickú 0
- na zapamätanie polí binárnych reťazcov
- funkcia pamäte v operačnej časti ČS
- statická pamäť typu RAM (Random Access Memory)
- dynamická pamäť typu DRAM (Dynamic RAM)

Funkcia pamäte v riadiacej časti ČS
- statická pamäť typu ROM (Read Only Memory)
- jednorázovo programovateľná pamäť typu PROM (Programmable ROM),
- preprogramovateľná pamäť typu EPROM (Erasable PROM),
- RWM (Read Write Memory)

4.1. a) Preklápacie obvody a typy preklápacích obvodov

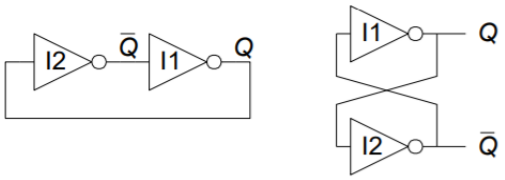
-Sekvenčná logika
- sekvenčné obvody sú všetky obvody, ktoré nie sú kombinatorické
- výstup sekvenčných obvodov závisí od aktuálnej ale aj od predchádzajúcich hodnôt vstupov - má pamäť
- stav - informácia o obvode potrebná na definovanie ďalšieho správania
- flip-flop a latch(preklápacie) obvody sú jednoduché sekvenčné obvody, ktoré uskladňujú jeden bit stave
- základným stavebným blokom pamäte je astabilný prvok
- astabilný prvok - prvok s 2 stavmi

Typy preklápacích obvodov: Podľa stavov
1. Astabilné - nemajú žiadny stabilný stav neustále kmitajú z jedného do druhého stavu
2. Monostabilné - majú jeden stabilný stav
3. Bistabilné - obidva stavy sú stabilné
4. Schmittov -zvláštny typ Preklápacích obvodov, slúži na úpravu impulzov

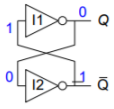
Podľa synchronizácie
- asynchrónne - preklápacie obvody, ktoré sa preklopia po príchode vstupných signálov a na základe vnútorného stavu, nepotrebujú žiadny ďalší signál
- sú všeobecnejšie ako synchrónne ale v praxi sa ukázali ako zložitejšie na návrh
- **synchrónne -** preklopia sa až po príchode synchronizačného signálu (ten povolí reakciu obvodu na vstupné signály).

Podľa typu synchronizácie:
- Synchronizácia úrovňou hodinového signálu (úrovňová alebo hladinová synchronizácia) (Level Triggered Latch)
- Synchronizácia nábežnou hranou hodinového signálu (Positive edge triggered flip-flop)
- Synchronizácia zostupnou hranou hodinového signálu (Negative edge triggered flip-flop)

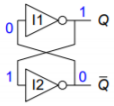
Bistabilný obvod:
- základný stavebný blok iných stavových prvkov
- má 2 výstupy Q,Q´ a žiadne vstupy
príklad:



- Consider the two possible cases:
  - $Q = 0$:
    then $\bar{Q} = 1$, $Q = 0$ (consistent)



  - $Q = 1$:
    then $\bar{Q} = 0$, $Q = 1$ (consistent)



- Stores 1 bit of state in the state variable, Q (or $\bar{Q}$)
- But there are **no inputs to control the state**

SR Latch:

- jeden z najjednoduchších sekvenčných obvodov
- skladá sa z NOR hradiel
- má vstupy a výstupy
- jeho stavy sa dajú kontrolovať pomocou vstupov S a R
- S = set : Sprav výstup 1
  R = reset : Sprav výstup 0

Problém: Nastavenie jedného zo vstupov určuje nie len aký má byť stav ale aj kedy sa má zmeniť, pre návrh obvodov je lepšie ak sú separované, tento problém rieši takzvaný D - latch

Case	S	R	Q	$\bar{Q}$
IV	0	0	Q_{prev}	$\bar{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

Figure 3.5 SR latch truth table

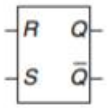


Figure 3.6 SR latch symbol

D Latch:

- obsahuje 2 signály dátový D(kontroluje na čo sa má zmeniť výstup) a hodinový CLK signál, ktorý rozhoduje o tom kedy sa má stav zmeniť

- ak je CLK = 1
  D prechádza cez Q (priepustný). D Latch aktualizuje stav priebežne občas je ale potrebné aby sa stav aktualizoval len v špecifickom čase - to rieši D Flip-Flop
- ak je CLK = 0
  D má tú istú hodnotu (nepriepustný), blokuje priechodnosť dát cez Q

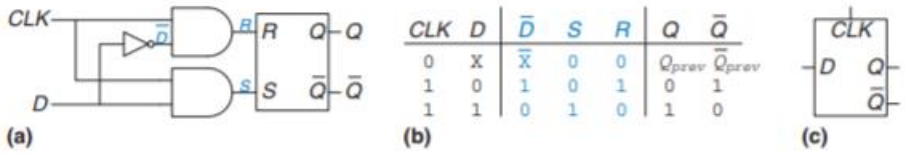


Figure 3.7 D latch: (a) schematic, (b) truth table, (c) symbol

D preklápací obvod

- má vstupy CLK a D
- nazýva sa aj (Positive edge triggered flip-flop)
- je vytvorený z 2 D Latchov a kontrolovaný doplnkovými CLK
- **keď je CLK = 0**
  L1 je priepustný
  L2 je nepriepustný
  D prechádza do N1
- **keď je CLK = 1**
  L2 je priepustný
  L1 je nepriepustný
  N1 prechádza cez Q
- **na hrane hodín (keď sa CLK mení z 0 na 1)**

D prechádza cez Q

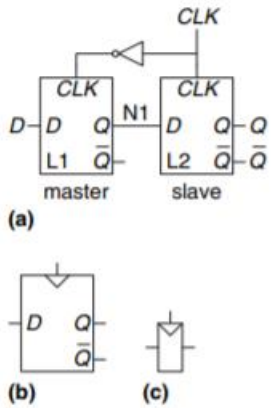


Figure 3.8 D flip-flop: (a) schematic, (b) symbol, (c) condensed symbol

4.1.b) Registre

- N-bitový register je usporiadaný súbor preklápacích obvodov majú spoločný vstup CLK, a preto sú všetky bity registra aktualizované v tú istú chvíľu
- registre sú kľučové stavebné bloky väčšiny sekvenčných obvodov

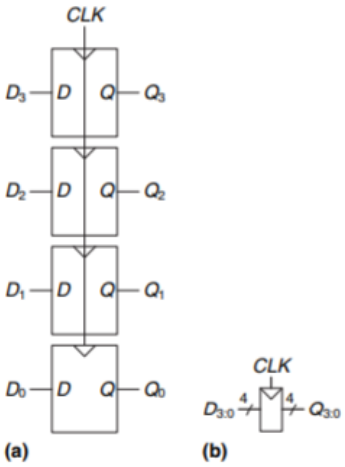


Figure 3.9 A 4-bit register: (a) schematic and (b) symbol

4.1.c) Konečnostavové automaty

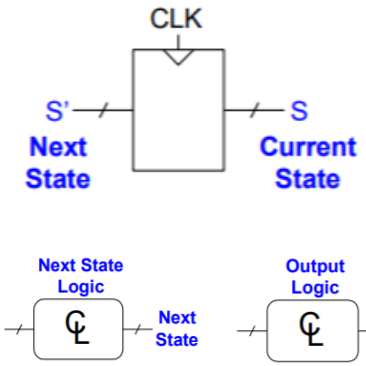
- sekvenčné stavové automaty vedia byť nakreslené vo forme konečnostavového automatu
- **skladajú sa z:**

Stavového registra

- ukladá aktuálny stav
- načítava ďalší stav

Kombinačnej logiky

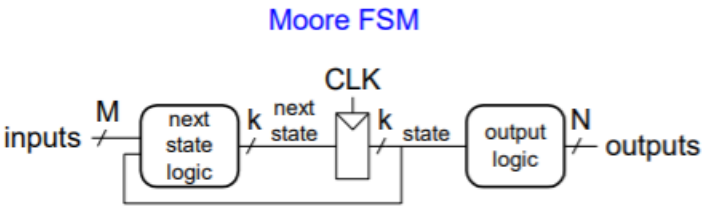
- vypočíta ďalší stav
- vypočíta výstup



- ďalší stav sa určuje aktuálnym stavom a výstupom

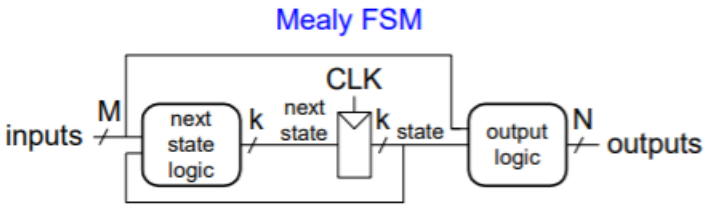
Moore

- výstup závisí len od aktuálneho stavu



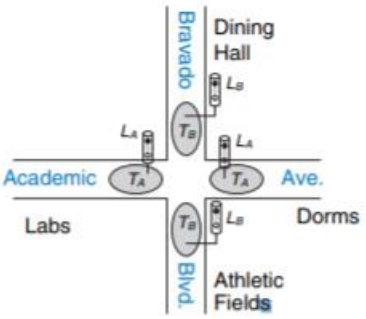
Mealy

- výstup závisí od aktuálneho stavu a vstupov

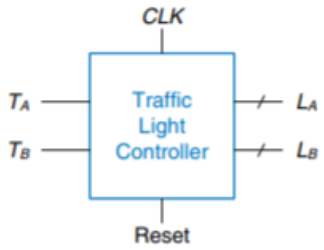


Príklad:

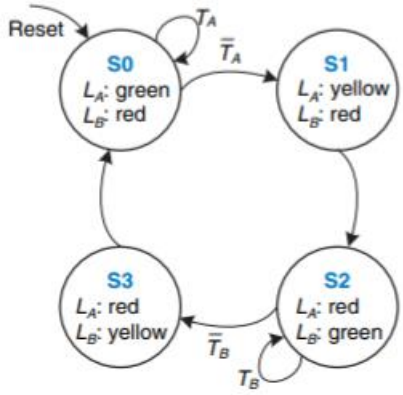
T_A,T_B - senzory, každý z nich indikuje či sa pri semafore nachádzajú ľudia
L_A,L_B - semafory - môžu mať farbu červenú, žltú alebo zelenú, konečnostavový automat má 2 vstupy (senzory) a 2 výstupy (semafory)
- hodiny majú 5 sekundovú periódu, vždy keď prejde 5 sekúnd svetlá sa zmenia na základe senzorov
- takisto sa tu nachádza aj tlačidlo reset tak aby sa dal nastaviť základný stav (inicializačný)



- pohľad na stavový automat cez black box



- diagram zmeny stavov



- kruhy reprezentujú stavy, čiary reprezentujú prechody medzi stavmi
- hodiny nie sú v diagrame avšak sú pritomné ale tie len kontrolujú, kedy má nastať zmena

Tabuľka stavov:

Table 3.1 State transition table

Current State S	Inputs T_A	Inputs T_B	Next State S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Table 3.2 State encoding

State	Encoding $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Table 3.3 Output encoding

Output	Encoding $L_{1:0}$
green	00
yellow	01
red	10

- z týchto tabuliek sa dá dostať boolovská rovnica pre ďalší stav a logika výstupu

Návrh konečnostavového automatu

1. Identifikácie vstupov a výstupov
2. Nákres diagramu prechodu medzi stavmi
3. Napísanie tabuľky prechodov medzi stavmi
- **Moore:**
  - prepísanie tabuľky pomocou Stavového kódovanie
  - napísanie tabuľky výstupov

  Mealy:
  - prepísanie kombinovanú tabuľku prechodov a tabuľku výstupov so stavovými kódovaniami
4. Napísanie Boolovskej rovnice pre ďalší stav a pre výstup
5. Nákres schémy obvodu

(k tomu Mealy Moore som chcel dať čo najviac ale neviem či to je málo alebo až som prestrelil :D)

4.2. a) Charakteristika synchrónnych sekvenčných logických obvodov

- prerušenie cyklických ciest vložením registrov
- Cyklická cesta - výstup je napojený naspäť na vstup
- registre obsahujú stavy systému
- zmeny systému na hranici hodín, systém je synchronizovaný s hodinami

- pravidlá kompozície synchrónnych sekvenčných obvodov
1. Každý prvok obvodu je buď register alebo kombinatorický obvod
2. Minimálne jeden prvok obvodu je register
3. Všetky regitre prijímajú rovnaký hodinový signál
4. Každá cyklická cesta obsahuje minimálne jeden register

Príkladmi synchrónnych sekvenčných obvodov sú:
1. Konečnostavové automaty
2. pipelines

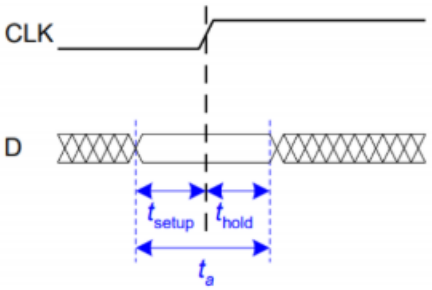
4.2. b) Hazardy

- hodiny musia byť pomalé aby sa predišlo predbiehanie

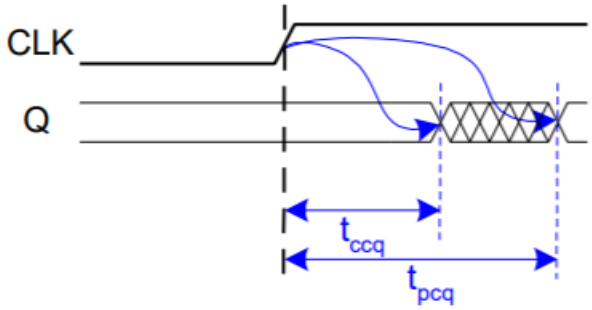
- zložitejšie na pochopenie ľahšie na implementáciu ako asynchrónne obvody
- sú dostatočné ale preukázalo sa, že sú obmedzujúce. Pri rýchlych mikroprocesoroch môžu niektoré registre prijať oneskorené hodinové cykly
tri druhy:
1. **statický** – pri zmene jednej vstupnej premennej sa okamžite mení výstup pred stabilizáciou na správnu hodnotu
2. **dynamický** – pri jedinej zmene vstupu sa výstup zmení viac ako jedenkrát
3. **funkčný** – vzniká zmenou na viacerých vstupoch

4.3. a) Časové charakteristiky sekvenčných obvodov

- preklápací obvod kopíruje vstup D na výstup Q na stúpajúcej hrane hodín, tento proces nazývame samplovanie D na hrane hodín
- ak je D stabilné buď na 0 alebo na 1 tak je toto správanie jasne definované, problém nastáva vtedy, keď sa D mení v tom istom čase ako keď hodiny rastú
Príklad :
- tento problém je podobný tomu ako pri fotografovaní
- ak odfotíme žabu presne v momente ako vyskočí uvidíme ju rozmazanú
- túto vlastnosť môžme popísať tak, že objekt musí nejakú dobu ostať na rovnakom mieste, po odfotení
- táto vlastnosť sa pri sekvenčných obvodoch popisuje pomocou "setup" času(čas kým dáta hodinovej hrany musia byť stabilné) a
"hold" času (čas po skočení hodinovej hrany kedy dáta musia byť stabilné)
- $t_{aperture}$ - čas v okolí hodinovej hrany kedy dáta musia byť stabilné

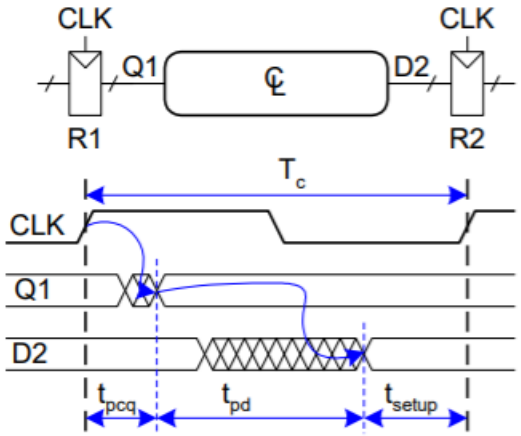


- propagation delay - čas po skočení hodinovej hrany, ktorý garantuje, že výstup Q sa nebud meniť
- contamination delay - čas po skočení hodinovej hrany, ktorý umožňuje Q byť nestabilné, meniť stav



- **Dynamická disciplína**
- vstupy synchrónnych sekvenčných obvodov musia byť stabilné počas času prierezu(aperture) v okolí hodinovej hrany
- vstupy musia byť stabilné aspoň t_{setup} pred hodinovou hranou
  aspoň t_{hold} po hodinovej hrane
  omeškanie medzi registrami má minimálne a maximálne omeškania, závisiace na omeškaniach obvodových prvkov

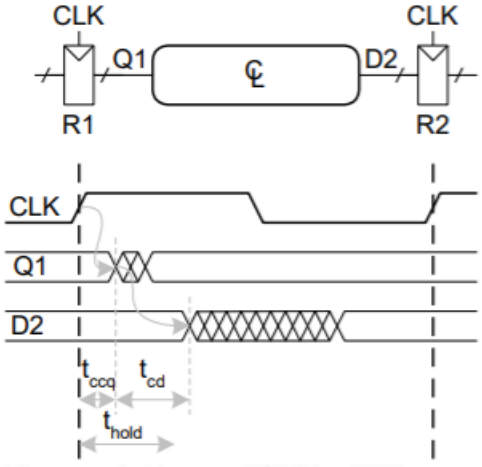
Časové obmedzenia pre Setup time
- závisí na maximálnom oneskorení z registra R1 cez kombinačnú logiku do R2
- vstup do registra 2 musí byť stabilný minimálne t_{setup} pred hodinovou hranou



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

T_c - hodinová perióda
t_{pcq} - Q- propagation delay
t_{pd} - propagation delay

Časové obmedzenia pre Hold time
- závisí na minimálnom oneskorení z registra R1 cez kombinačnú logiku do R2
- vstup do registra musí byť stabilný minimálne t_{hold} po hodinovej hrane



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

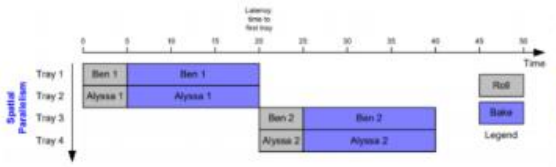
Hodinový sklz:
- rozdiel medzi dvoma hodinovými hranami
- hodiny neprichádzajú na všetky registre v ronaký čas
- hľadáme najhoršiu možnú analýzu pre garanciu že dynamická disciplína nieje porušená pre žiaden register – v systéme je mnoho registrov

Paralelné spracovanie:

- rýchlosť systémuje charakterizujeme cez latenciu a priepustnosť informácie, ktorá cez ňu prechádza
- definujme token ako skupinu vstupov, ktoré sú spracované na vyprodukovanie skupiny výstupov
- latencia - čakacia doba/čas za ktorý token prejde zo začiaku na koniec
- priepustnosť je počet tokenov, ktoré sú vyprodukované za jednotku času
- priepustnosť sa vie zlepšiť produkovaním viacerých tokenov naraz, túto vlastnosť nazývame paralelizmus a má viac foriem priestorovú a dočasnú
-Priestorový paralelizmus - je dodaných viac kópií hardvéru tak, že sa dá splniť viacero úloh naraz
- Dočasný paralelizmus - úloha sa rozdelí do viac úrovní, viacero úloh vie byť rozdelených medzi úrovniami a v každom čase sa môže vykonávať viac úloh

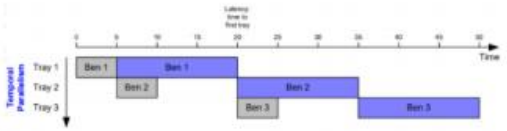
- Ben Bitdiddle bakes cookies to celebrate traffic light controller installation
- 5 minutes to roll cookies
- 15 minutes to bake
- What is the latency and throughput without parallelism?

Latency = 5 + 15 = 20 minutes = **1/3 hour**
Throughput = 1 tray/ 1/3 hour = **3 trays/hour**



Latency = 5 + 15 = 20 minutes = **1/3 hour**
Throughput = 2 trays/ 1/3 hour = **6 trays/hour**

- What is the latency and throughput if Ben uses parallelism?
  - **Spatial parallelism:** Ben asks Allysa P. Hacker to help, using her own oven
  - **Temporal parallelism:**
    - two stages: rolling and baking
    - He uses two trays
    - While first batch is baking, he rolls the second batch, etc.



Latency = 5 + 15 = 20 minutes = **1/3 hour**
Throughput = 1 trays/ 1/4 hour = **4 trays/hour**

Using both techniques, the throughput would be **8 trays/hour**

ZHRNUTIE
- na rozdiel od kombinačnej logiky, sekvenčná logika si pamätá predchádzajúce a aktuálne vstupy a závisí od nich
- táto pamäť sa nazýva stav logiky
- sekvenčné obvody sú zložité na analýzu a ľahko vedia byť zostrojené na pár opatrne navrhnutých stavebných blokov
- najdôležitejším prvkom je preklápací obvod, ktorý prijíma CLK - clock a vstup D - dáta a produkuje výstup Q
- preklápací obvod kopíruje D na Q na dvíhajucej hrane hodín, inými slovami pamätá si predchádzajúci stav Q
- skupina preklápacích obvodov, ktoré majú spoločné CLK sa nazýva register

- preklápací obvod môže prijímať aj kontrolné signály reset alebo enable
- existujú rôzne typy sekvenčnej logiky ale my sa obmedzíme na synchrónne sekvenčné obvody
- synchrónne sekvenčné obvody obsahujú bloky kombinačnej logiky prepojené registrom
- konečnostavové automaty sú technikou na zostrojovanie sekvenčných obvodov
- pravidlá zostrojovania:
1. Identifikácia vstupov a výstupov
2. Nákres diagramu vzťahu medzi stavmi
3. Výber kódovania pre stavy a prekreslenie diagramu na tabuľku a tabuľku výstupov
4. Z tejto tabuľky vieme navrhnúť kombinačnú logiku na výpočet ďalšieho stavu a výstupu
5. Nákres obvodu
- Časové špecifikácie
CLK - Q propagation delay
CLK - Q contamination delay
- oneskorenia - kedy sa po zmene CLK signálu zmení výstup
- t aperture(prierezový čas)
- je čas počas, ktorého musia byť dáta stabilné
t aperture = t setup + t hold
-setup - čas predtým ako hrana skočí a dáta musia byť stabilné
- hold - čas po skočení hodinovej hrany
- dĺžka minimálneho cyklu systému = tpd + tpc + tsetup
- celková rýchlosť systému sa počíta cez latenciu a priepustnosť
- latencia je čas potrebný na prechod tokenov
- priepustnosť je počet tokenov, ktoré vie systém vyprodukovať za jednotku času
- token - skupina vstupov potrebná na vyprodukovanie skupiny výstupov

Topic 5

5.1 Úvod do jazyka VHDL, moduly, kompozícia zdrojového súboru vhd, simulácia, syntéza. Návrh kombinačných logických obvodov v jazyku VHDL, behaviorálny opis modulov.

5.1.a) Úvod do jazyka VHDL

- HDL - Hardware description language je jazyk popisujúci štruktúru a správanie elektrických obvodov
- 2 napoužívanejšie HDL
1 VHDL 2008
2 SystemVerilog

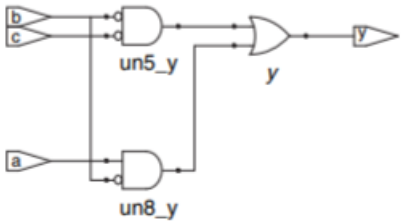
- Dva hlavné účely HDL sú logické simulácie a syntézy.

Simulácia

- vstupy aplikované na obvod
- výstupy skontrolované (či sú správne)
- debugovanie v simulácii narozdiel od hardvéru (ušetrené peniaze)

Syntéza

- Transformuje HDL kód do netlistu popisujúceho hardvér (t. j. zoznam hradiel a káblov ktoré ich spájajú)
- Logický syntetizér optimalizuje množstvo potrebného hardware
- netlist môže byť textový súbor alebo schéma



Syntetizovaný obvod
- dôležité je myslieť na hardvérm ktorý by mal HDL produkovať

Moduly
- blok hardvéru so vstupmi a výstupmi sa nazýva module
pr: AND hradlo, multiplexor
- 2 najdôležitejšie štýly na opis funkcionality hardvér sú behaviorálny a štrukturálny
Behaviorálny:
- popisuje to čo modul robí
Štrukturálny
- popisuje ako sú moduly vytvorené z menších častí (aplikácia hierarchie)

5.1.b) Kompozícia VHDL kódu

Príklad VHDL kódu:
- výpočet funkcie y = a´b´c´+ ab´c´ + ab´c
3 časti:
library, entity a architecture
STD_LOGIC - tieto signály môžu mať hodnotu 0 alebo 1
vstupy a,b,c
výstupy y
  v časti architecture je výpočet boolovskej funkcie
zátvorky sú nutné, pretože VHDL nemá dobre nastavené poradie AND a OR
Syntax:
- reset = Reset (not case-sensitive)
- názvy nesmú začínať s číslom
- komentáre
  -- single line
-/* multiline
        comment */

VHDL

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

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

architecture synth of sillyfunction 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;
```

5.1.c) Návrh kombinačných logických obvodov

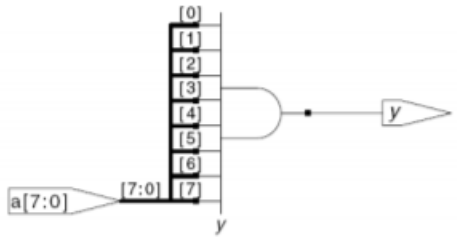
- **Bitové operátory**

y <= a and b; AND
y <= a or b; OR
y <= a xor b; XOR
y<= not(a and b) NAND
y<= not (a or b) NOR

Redukčné operátory

- viacvstupová brána fungujúca na jednej zbernici
Pr:
- opis 8 vstupovej AND brány

```
port(a : in STD_LOGIC_VECTOR( 7 downto 0);
          y: out STD_LOGIC);
```



viac vstupová
je jednoduchšie napísať
y <= a;
ako y <= a(7) and a(6) ... and a(0)
zbernica - skupina viacerých signálov

Podmienkové prirovnania
- vyberajú si výstup medzi alternatívami založenými na vstupe nazvanom podmienka
Pr: Multiplexor 2 : 1

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

entity mux2 is
    port(d0,d1 :       in STD_LOGIC_VECTOR(3 downto 0)
                    s   :              in STD_LOGIC;
          y:                 int STD_LOGIC_VECTOR(3 downto 0));
```

```
end;
```

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

- vnútorné premenné
- niekedy sa hodí rozložiť zložitejšiu funkciu na viac krokov, napríklad plná sčítačka

is a circuit with three inputs and two outputs defined by the following equations:

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in} \quad (4.1)$$

If we define intermediate signals, P and G,

$$P = A \oplus B$$
$$G = AB \quad (4.2)$$

we can rewrite the full adder as follows:

$$S = P \oplus C_{in}$$
$$C_{out} = G + PC_{in} \quad (4.3)$$

p a g sú takzvané vnútorné premenné
VHDL kód:

VHDL

In VHDL, *signals* are used to represent internal variables whose values are defined by *concurrent signal assignment statements* such as `p <= a xor b;`

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

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
  signal p, g: STD_LOGIC;
begin
  p <= a xor b;
  g <= a and b;

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

Čísla, Z - ka a X - ka, Oneskorenia
- čísla vedia byť špecifikované v binárnej, osmičkovej, desiatkovej alebo v hexadecimálnej sústave
- Z - desatinná hodnota, ktoré je vhodná na popis trojstavového bufferu, ktorého výstup "pláva", ak je enable = 0

VHDL

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

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;
```

- X - indikuje zakázanú logickú úroveň ak sa v simulácii nachádzajú hodnoty X skoro vždy to znamené nejaký bug alebo zlý kód
- môže to viesť k nepredvídateľnému správaniu obvodu

Oneskorenia:

- HDL príkazy vedia byť spojené s oneskoreniami v určitých jednotkách
- sú nápomocné počas simulácie na predvídanie toho ako rýchlo bude obvod pracovať, a takisto na proces debuggovania
- tieto oneskorenia sú ignorované počas syntézy

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 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;
```

In VHDL, the `after` clause is used to indicate delay. The units, in this case, are specified as nanoseconds.

5.2.a) Štrukturálny opis modulov

- opisuje modul tak ako je vyskladaný z menších modulov

Pr: 4 : 1 Multiplexor z troch 2 : 1 Multiplexorov.

- každá kópia 2 : 1 Multiplexora je volaná ako instancia
- viac instancií rovnakého modulu sú odlíšené pomocou rozdielnych mien (lowmux, highmux, finalmux) - princíp jednotnosti(regularity) jedna súčiastka je použitá až 3x

VHDL

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

entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux4 is
  component mux2
    port(d0,
         d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal low, high: STD_LOGIC_VECTOR(3 downto 0);
begin
  lowmux:   mux2 port map(d0, d1, s(0), low);
  highmux:  mux2 port map(d2, d3, s(0), high);
  finalmux: mux2 port map(low, high, s[1], y);
end;
```

5.2.b) Sekvenčná logika

- idiómy sú špecifické spôsoby popisu rôznych tried a logiky
- HDL syntetizéry rozpoznávajú viacero idiómov a menia ich na špecifické sekvenčné obvody
- rôzne programovacie štýly môžu byť správne nasimulované ale ich syntéza do obvodov môže viesť k errorom
- táto sekcia ukazuje správne idiómy na opis registrov a latchov

Registre:

- (Positive edge triggered D flip-flops)
- príklad dole ukazuje Idióm pre tieto preklápacie obvody
- vo VHDL príkaz *process* funguje tak, že signál si udržuje starú hodnotu, kým sa nestane nejaká udalosť na zozname , ktorá ju zmení, teda takýto kód vie byť použitý na opis sekvenčného obvodu s pamäťou

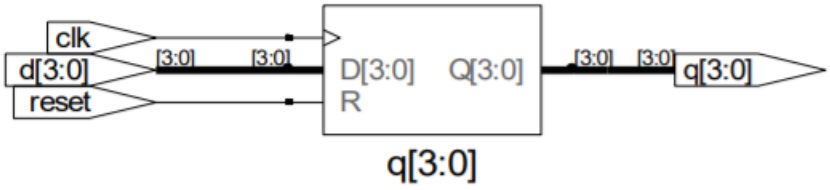
VHDL

```
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;
```

Figure 4.14 `flop` **synthesized circuit**

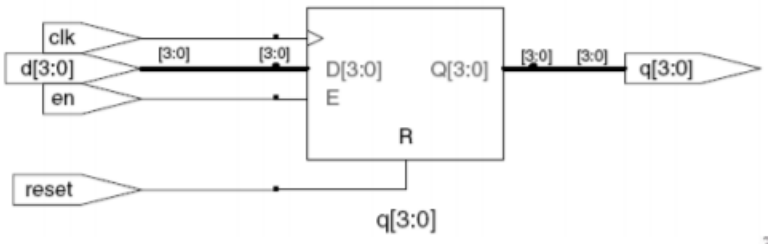
Resettable

```
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;
```



Enable

```
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;
```



Latch

- väčšinou nie je vhodné používať latch ale radšej edge-triggered preklápací obvod

5.2.c) Sekvenčné príkazy jazyka VHDL

- kombinačná logika s použitím príkazu 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;
```

- príkazy, ktoré môžu byť len vo vnútri príkazu process
1. If,else
2. case,case?

- kombinačná logika s použitím príkazu case

Pr: Sedemsegmentovka

VHDL

```
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;

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";
    end case;
  end process;
end;
```

- when others - musí byť použité pre defaultný prípad
- case príkaz skontroluje hodnotu dát, keď sú dáta = 0 tak príkaz vykoná akciu nastavenie segmentu na 1111110
- case skontroluje všetky ostatné hodnoty, others statement je cesta ako definovať výstup pre všetky prípady nie len tie explicitne zadané
- takisto sú pomocou príkazu case vytvárané dekódery

IF STATEMENT

- príkaz process môže tiež obsahovať aj príkaz if, ktorý môže byť nasledovaný príkazom else
- keď sú všetky možné kombinácie vstupov pokryté, príkaz implikuje kombinačnú logiku inak produkuje sekvenčnú logiku
- príkladom je obvod priorít

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity priorityckt is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth of priorityckt is
begin
  process(all) begin
    if    a(3) then y <= "1000";
    elsif a(2) then y <= "0100";
    elsif a(1) then y <= "0010";
    elsif a(0) then y <= "0001";
    else            y <= "0000";
    end if;
  end process;
end;
```

Unlike SystemVerilog, VHDL supports conditional signal assignment statements (see HDL Example 4.6), which are much like `if` statements but can appear outside processes. Thus, there is less reason to use processes to describe combinational logic.

5.2.d) Blokujúce a neblokujúce priradenia

- ak sa neriadime pomocou návodu je možné napísať kód, ktorý funguje počas simulácie a nefunguje počas syntézy

Návod ako používať blokujúce a neblokujúce priradenia

VHDL

1. Use `process(clk)` and nonblocking assignments to model synchronous sequential logic.

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

2. Use concurrent assignments outside `process` statements to model simple combinational logic.

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

3. Use `process(all)` to model more complicated combinational logic where the `process` is helpful. Use blocking assignments for internal variables.

```
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;
```

4. Do not make assignments to the same variable in more than one `process` or concurrent assignment statement.

5.3.a) Konečnostavové automaty v jazyku VHDL

- konečnostavový automat obsahuje stavový register a 2 bloky kombinačnej logiky na výpočet ďalšieho stavu a výstupy sú dané aktuálnym stavom a vstupom
- HDL popis stavového automatu je rozdelený na 3 časti

1. Model stavového registra
2. Logika ďalšieho stavu
3. Logika výstupu

VHDL

```
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;
```

This example defines a new *enumeration* data type, `statetype`, with three possibilities: `S0`, `S1`, and `S2`. `state` and `nextstate` are `statetype` signals. By using an enumeration instead of choosing the state encoding, VHDL frees the synthesizer to explore various state encodings to choose the best one.

The output, `y`, is 1 when the `state` is `S0`. The inequality comparison uses `/=`. To produce an output of 1 when the state is anything but `S0`, change the comparison to `state /= S0`.

5.3.b) Parametrizované moduly

- všetky naše moduly mali fixnú šírku a dĺžku výstupov
- napríklad sme museli definovať samostatné moduly pre 4 a 8-bitové širokopásmové multiplexory 2:1
- HDL umožňujú variabilnú šírku bitov pomocou parametrizovaných modulov
- generické a generizované príkazy

VHDL

```
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;
```

The `generic` statement includes a default value (8) of `width`. The value is an integer.

```
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;

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;
```

The 8-bit 4:1 multiplexer, `mux4_8`, instantiates three 2:1 multiplexers using their default widths.

In contrast, a 12-bit 4:1 multiplexer, `mux4_12`, would need to override the default width using `generic map`, as shown below.

```
lowmux: mux2 generic map(12)
             port map(d0, d1, s(0), low);
himux:  mux2 generic map(12)
             port map(d2, d3, s(0), hi);
outmux: mux2 generic map(12)
             port map(low, hi, s(1), y);
```

5.3.c) Testbenche

- testbench je HDL modul, ktorý sa používa na testovanie iného modulu nazvaného prístroj pod testom (DUT device under test)
- testbench obsahuje príkazy na aplikovanie vstupov do DUT a ideálne na kontrolu všetkých správnych výstupov, ktoré sú produkované
- vstupy a možné výstupy sa nazývajú testovacie vektory
- test funkcie modulu sillyfunction

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

- sillyfunction je jednoduchý modul, takže môžeme spraviť testovanie všetkých možných vektorov

VHDL

```
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;
begin
  -- instantiate device under test
  dut: 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;
```

The `process` statement first applies the input pattern 000 and waits for 10 ns. It then applies 001 and waits 10 more ns, and so forth until all eight possible inputs have been applied.

At the end, the process waits indefinitely; otherwise, the process would begin again, repeatedly applying the pattern of test vectors.

- je to jednoduchý testbench, najprv vytvorí inštanciu DUT a potom otestuje výstupy
- blokujúce príkazy a oneskorenia sú používané na aplikáciu vstupov v správnom poradí
- používateľ musí vidieť výsledky simulácie a inšpekciou skontrolovať či sú produkované správne výstupy

1. Testbenche sú simulované rovnako ako iné HDL moduly, avšak nedajú sa syntetizovať

1. Kontrola správnych výstupov je zložitá a je možné, že sa vyskytnú errory
2. Ďalším problémom je to, že určovanie správnych výstupov je ľahšie ak máme návrh čerstvejšie v hlave , a preto ak spravíme menšie zmeny po nejakom čase určenie správneho výstupu je zložitejšie

2. Omnoho lepšie je preto písať samokontrolné testbenche, navyše pre väčšinu modulov je ťažké písať kód pre všetky vektory, a preto je dobré uložiť testovacie vektory do separovaných súborov

3. Testbench jednoducho prečíta testovacie vektory z iných súborov a aplikuje vstupné testovacie vektory do DUT, počká, skontroluje výstupné hodnoty z DUT a výstupné vektory, a potom opakuje kým a nedostane na záver súboru testovacích súborov

VHDL

```
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;
begin
  -- instantiate device 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.";
    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;
```

The `assert` statement checks a condition and prints the message given in the `report` clause if the condition is not satisfied. `assert` is meaningful only in simulation, not in synthesis.

Samokontrolný Testbench

Zhrnutie:

- HDL je veľmi dôležitý nástroj pre moderného digital dizajnéra
- urýchľuje to prácu a zlepšuje a zrýchľuje možnosti debuggingu
- problém s debuggingom je ale v tom, že môže byť dlhší ak nemáme správnu predstavu o hardvéri, ktorý má náš kód implikovať
- HDL sa používa pre simuláciu aj pre syntézu
- simulácia je silný nástroj na testovanie systému ešte na počítači predtým než sa ten premení na hardvér
- simulátor ti dovolí skontrolovať hodnoty signálov vo vnútri systému, ktoré môže byť nemožné odmerať na časti hardvéru
- logická syntéza skonvertuje kód HDL na kód digitálneho logického obvodu
- najdôležitejšou vecou pri písaní HDL kódu je zapamätať si, že opisujeme naozajstný hardvér a nepíšeme vlastný počítačový program
- najväčšou chybou pri písaní HDL je písať HDL kód bez rozmýšľania o hardvére, ktorý chceme vyprodukovať
- keď nevieme aký hardvér opisujeme takmer vždy dostaneme niečo čo sme nechceli
- namiesto toho treba začať s nákresom blokového diagramu systému, identifikujeme, ktoré časti sú kombinačnou logikou a ktoré sekvenčnou alebo konečnostavovým automatom...
- potom napíšeme kód HDL pre každú časť používaním správnych idiómov na implikovanie hardvéru,

ktorý potrebujeme

TOPIC 6

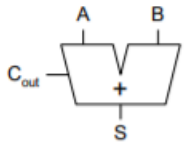
6.1. Digitálne logické obvody určené na realizáciu aritmetických operácií: sčítačky, odčítačky, komparátor, posuvná jednotka, jednoduché ALU.

Aritmetické obvody sú hlavnými stavebnými blokmi počítačov.
Počítače a digitálna logika vykonávajú mnohé aritmetické funkcie: pridanie, odčítanie, porovnávanie, posuny, násobenie a rozdelenie.

Sčítanie je jednou z najbežnejších operácií v digitálnych systémoch. Začneme tým, že budeme vytvárať 1-bitovú polovičnú sčítačku.
S je súčet A a B, jednobitových vstupov.
Ak A a B sú obe 1, S je 2, ktoré nemožno reprezentovať jednou binárnou číslicou.
Namiesto toho je v nasledujúcom stĺpci označený vykonaním Cout.

Polovična sčítačka môže byť postavená z brány XOR a brány AND.
Úplná sčítačka akceptuje prenos v cin, ako je znázornené na obrázku vpravo.
Úplnú sčítačku je možné zložiť z dvoch polovičných sčítačiek a hradla OR

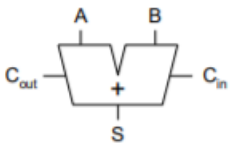
Half Adder



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$
$C_{out} = AB$

Full Adder



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

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

- Úplné sčítačky je možné zreťaziť vedľa seba a realizovať tak sčítanie viacbitových čísel

Viacbitové sčítačky

N-bitová sčítačka spája dva N-bitové vstupy A a B a prenos v Cin, aby vytvoril N-bitový výsledok S a

vykonal Cout.To sa bežne nazýva prenosový propagátor (CPA), pretože vykonávanie jedného bitu sa šíri do ďalšieho bitu.

3 typy: Ripple carry adder
Carry lookahead adder
Prefix adder

Ripple Carry

Cout z jednej fázy funguje ako Cin ďalšej fázy, ako je znázornené na obrázku pre 32-bitové pridanie.
Inak povedané príznak prenosu sa odovzdáva (propaguje) z jedného bitu do druhého.
Toto sa nazýva zväčšovacie zariadenie.
Je to dobrá aplikácia modularity a pravidelnosti: celý modul sčítača sa opakovane opakovane používa na vytvorenie väčšieho systému.
Ripplecarry sčítač má nevýhodu, že je pomalý, keď je N veľký.

Carry Lookahead(CLA)

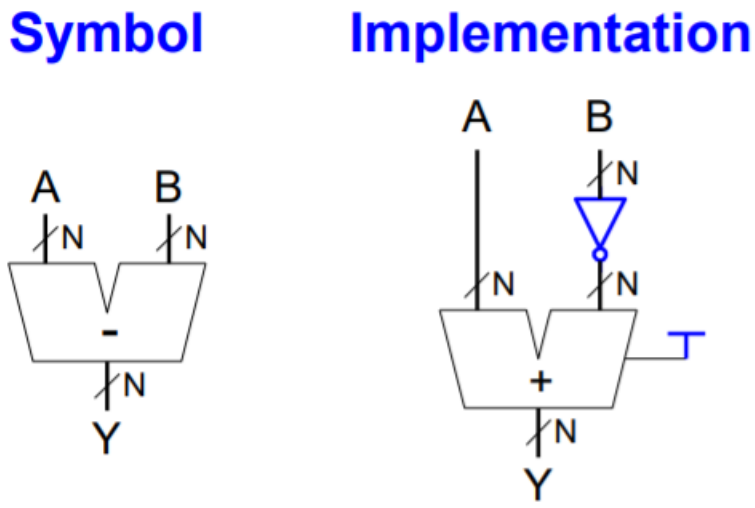
Hlavným dôvodom, prečo je Ripple Carry pomalý, je, že prenášané signály sa musia šíriť cez každý bit v sčítači.
CLA je ďalším typom sčítača, ktorý rieši tento problém rozdelením sčítača do blokov a poskytnutím obvodov na rýchle určenie vykonania bloku, akonáhle je známy príznak prenosu.
Preto sa hovorí, že sa pozerá skôr cez bloky než čaká na zvlnenie cez všetky plné zväzky vnútri bloku.
Omeškanie
Takže N-bitový sčítač rozdelený do k-bitových blokov má oneskorenie> Pre N> 16, teda Carry Lookahead je oveľa rýchlejší ako Ripple Carry
Zdá sa však, že oneskorenie sčítača sa lineárne zvyšuje s N.

Prefix Adder

Rozširuje generovanie a šírenie logiky zberača prenosu údajov prebiehajúceho prenosu, aby ešte viac vykonávali sčítavanie (nie rýchlejšie ?). Najprv vypočítajú G a P pre dvojice stĺpcov, potom pre bloky 4, potom pre bloky 8, potom 16 a tak ďalej, až kým nie je známy generujúci signál pre každý stĺpec.

Sumy sú vypočítané z týchto generovaných signálov.
Omeškanie
Stručne povedané, predčíslová adder dosahuje oneskorenie, ktoré rastie logaritmicky namiesto lineárne s počtom stĺpcov v sčítači.
Toto zrýchlenie je významné, najmä pri prídavných zariadeniach s 32 alebo viacerými bitmi, ale prichádza na úkor väčšieho hardvéru ako Carry Lookahead.

Odčítačky



- sčítačky vedia sčítať čísla kladné aj záporné
- odčítanie je jednoduché – otočíme znamienko druhého čísla a potom sčítame
- prevrátenie znamienka sa vykoná obrátením bitov a pridaním 1

Komparátory

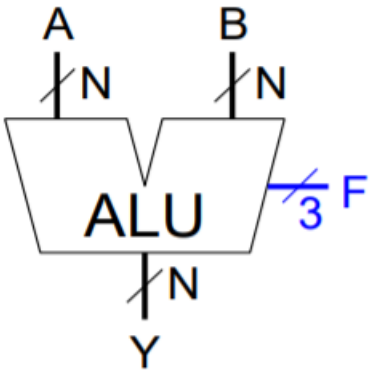
Komparátor určuje, či sú dve binárne čísla rovnaké alebo či je jedno väčšie alebo menšie než druhé. Komparátor dostane dve N-bitové binárne čísla A a B.

Existujú dva bežné typy komparátorov:

Komparátor rovnosti vytvára jediný výstup, ktorý udáva, či sa A rovná B (A == B).

Komparátor veľkosti vytvára jeden alebo viac výstupov, ktoré udávajú relatívne hodnoty A a B. Vyráta sa A – B a zisťuje sa znamienko .Ak je výsledok negatívny, A je menšie ako B a opačne

ALU



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Aritmetická / logická jednotka (ALU) kombinuje rôzne matematické a logické operácie do jednej jednotky.

Napríklad typická jednotka ALU môže vykonávať operácie sčítania, odčítania, porovnania veľkosti, AND a OR.
ALU tvorí srdce väčšiny počítačových systémov
Riadiaci signál F určuje, ktorá funkcia má vykonať.
Niektoré ALU produkujú extra výstupy, nazývané príznaky, ktoré označujú informácie o ALU výstupe
- napr. príznak pretečenia označuje, že výsledok sčítania pretiekol
- napr. nulový príznak označuje, že výsledok bol 0

Posuvná jednotka
Ako naznačuje názov, posúvač posúva binárne číslo doľava alebo doprava o určitý počet pozícií.

Existuje niekoľko typov bežne používaných posúvačov:
- **Logický shifter:** posúva do lava alebo prava a voľné miesta zaplní 0
  - Ex: 11001 >> 2 =
  - Ex: 11001 << 2 =

Aritmetický shifter: pri posúvaní doprava zaplnuje prázdne miesto kópiou starého znamienkového bitu
  - Ex: 11001 >>> 2 =
  - Ex: 11001 <<< 2 =
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: 11001 ROR 2 =
  - Ex: 11001 ROL 2 =

Príklady:
- **Logical shifter:**
  - Ex: 11001 >> 2 = 00110
  - Ex: 11001 << 2 = 00100
- **Arithmetic shifter:**
  - Ex: 11001 >>> 2 = 11110

– Ex: 11001 <<< 2 = 00100
- **Rotator:**
– Ex: 11001 ROR 2 = 01110
– Ex: 11001 ROL 2 = 00111

6.2 Reprezentácia reálnych čísel v číslicových počítačoch:

Táto časť zavádza systémy čísel s pevným a pohyblivým bodom, ktoré môžu reprezentovať racionálne čísla.

Čísla s pevnými bodmi sú analógové s desatinnými číslami; niektoré z bitov predstavujú celé číslo a zvyšok predstavujú zlomok.

Čísla s pohyblivou čiarou sú analogické s vedeckým zápisom, s mantisou a exponentom.

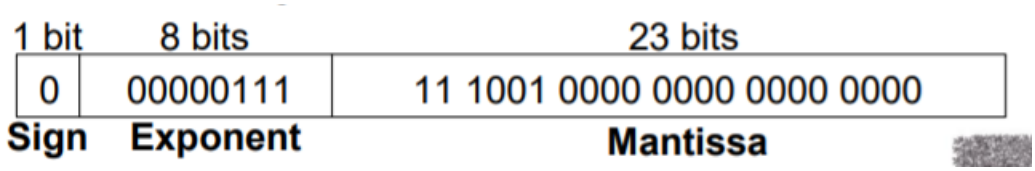
Fixed point numbers:
Označenie s pevnými bodmi má implicitný binárny bod medzi celočíselnými a zlomkovými bitmi, analogicky s desatinnou čiarkou medzi celé a čiastkové číslice bežného desatinného čísla.

Počet celých a zlomkových bitov musí byť vopred dohodnutý
- napr. 6.75 = 0110.1100

Floating point numbers:
Čísla s pohyblivou čiarou sú analogické s vedeckým zápisom. Omezujú obmedzenie mať konštantný počet celočíselných a zlomkových bitov, čo umožňuje zobrazenie veľmi veľkých a veľmi malých čísel.

Rovnako ako vedecká notácia, čísla s pohyblivou rádovou čiarkou majú znak, mantisa (M), základ (B) a exponent (E).

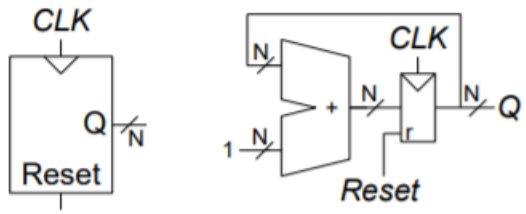
- vo všeobecnosti zapisujeme číslo vedeckou notáciou takto : ± M × BnaE
- M – mantisa, B – základ, E – exponent
- napr. M = 2.73, B = 10, E = 2



Alebo príklad 2: 22810 = 111001002

Počítadlá

Symbol **Implementation**



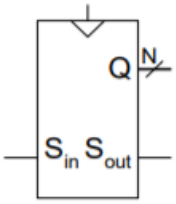
N-bitové binárne počítadlo, znázornené na obrázku, je sekvenčný aritmetický obvod s hodinovými a resetovacími vstupmi a N-bitový výstup Q.

Reset inicializuje výstup na hodnotu 0.

Počítadlo potom postupuje cez všetky 2 ^ N možné výstupy v binárnom poradí, zvyšujúc sa na stúpajúcom okraji hodín

Príklady: Zobrazenie digitálnych hodín
Počítadlo programov: sleduje aktuálne vykonávanie pokynov

Shift register



Zoskupovací register má hodiny, sériový vstup Sin, sériový výstup Sout a N paralelné výstupy Q.

Na každom nábežnom okraji hodín sa nový bit posunie zo Sin a všetko nasledujúci obsah sa posunie dopredu.

Posuvný register môže byť zostrojený z N flip-flops zapojených do série.
Vstup je poskytovaný sériovo (jeden bit v danom čase) v Sin.

Po cykloch N sú posledné N vstupy paralelne dostupné v Q.

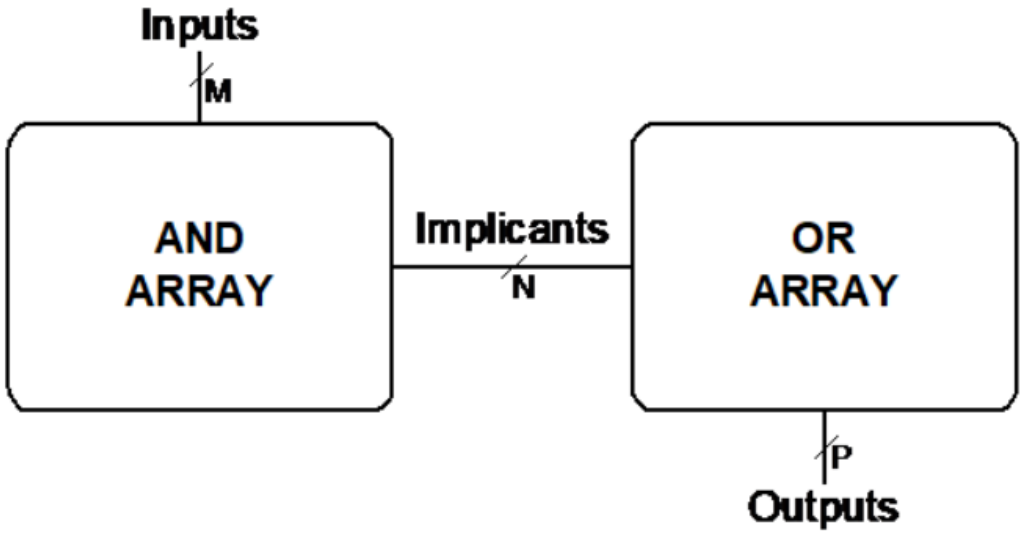
Shift Register s paralelným zaťažením

Keď Load = 1, pôsobí ako normálny N-bitový register
Keď Load = 0, pôsobí ako posuvný register

PLA(programmable logic array)

Programovateľné logické polia (PLA) implementujú kombinovanú logiku dvoch úrovní vo forme sumárnych produktov (SOP).

PLA sú vytvorené z poľa AND a potom nasleduje pole OR. Napríklad ROM môžu byť považované za osobitný prípad PLA.



FPGA

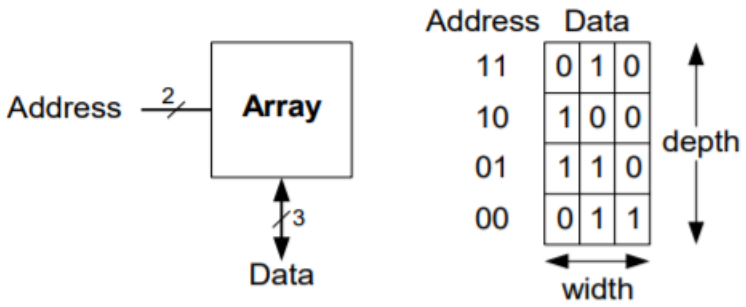
Pole programovateľné brány (FPGA) je pole rekonfigurovateľných brán.Teda FPGA sú postavené ako pole konfigurovateľných logických prvkov (LEs), ktoré sa tiež označujú ako konfigurovateľné logické bloky (CLBs).
Každý LE môže byť nakonfigurovaný na vykonávanie kombinovaných alebo sekvenčných funkcií.
LE je tvorený:

– **LUTs** (lookup tables): vykonáva kombinačnú logiku
– **Flip-flops:** vykonáva sekvenčnú logiku
– **Multiplexers:** spája LUTs a FlipFlops

Pomocou softvérových programovacích nástrojov môže používateľ realizovať návrhy na FPGA pomocou HDL alebo schémy.
FPGA sú silnejšie a flexibilnejšie ako PLA z niekoľkých dôvodov.
Môžu implementovať kombinovanú aj sekvenčnú logiku.

6.3 Pamaťové obvody

Pamäťové polia dokážu ukladať veľké množstvo dát.
2-rozmerné pole bitových buniek
Každá bitová bunka ukladá jeden bit
N bitov adresy a M dátových bitov:
2N riadky a stĺpce M
Hĺbka: počet riadkov (počet slov)
Šírka: počet stĺpcov (veľkosť slova)
Veľkosť poľa: hĺbka × šírka = 2N × M



Počas čítania pamäte sa uplatňuje wordline a príslušný rad bitline riadi bitové čiary HIGH alebo LOW. Počas zápisu do pamäte sa bitline najskôr riadia HIGH alebo LOW a potom sa uplatňuje wordline, ktorý umožňuje uloženie bitových hodnôt v tomto riadku bitline.

Klasifikácia pamätí:
Pamäte sú klasifikované na základe toho, ako ukladajú bity do bitovej bunky.

Najširšia klasifikácia je pamäť s náhodným prístupom (RAM) v porovnaní s pamäťou na čítanie (ROM).
RAM je energeticky závislá, čo znamená, že pri vypnutí napájania stráca svoje údaje.
ROM je energetický nezávislá, čo znamená, že uchováva svoje údaje na neurčito, a to aj bez zdroja.

RAM

Pamäť RAM sa nazýva pamäť s náhodným prístupom, pretože akékoľvek dátové slovo sa pristupuje s rovnakým oneskorením ako akékoľvek iné.

Typy RAM:

DRAM- Hodnota bitov je uložená na kondenzátore. Po čítaní sa hodnoty údajov prenášajú z kondenzátora na bitline.
Po zapísaní sa hodnoty údajov prenášajú z bitline na kondenzátor.
Čítanie ničí bitovú hodnotu uloženú na kondenzátore, takže dátové slovo musí byť po každom prečítaní obnovené (prepísané).

SRAM- Bitová hodnota je uložená na striedavých meničoch.
Statická pamäť RAM (SRAM, vyslovene "es-ram") je statická, pretože uložené bity nemusia byť obnovované.

Po potvrdení wordline sa zapnú obidva tranzistory nMOS a hodnoty údajov sa prenášajú do alebo z bitlines.

ROM

- uchováva bit ako prítomnosť alebo absenciu tranzistora
- je pamäť len na čítanie
- všetky ROM sú energeticky nezávislé, čiže zachovávajú údaje aj pri odpojení napájania
- ROM sa užíva na trvalé uchovanie údajov
- údaje sa jednoducho uložia a nedajú sa zmeniť
- pri čítaní, bitline je HIGH, wordline sa zapne
- ak je prítomný tranzistor, zmení bitline na LOW, ak je neprítomný, bitline zostáva HIGH

- koncepčne, ROM môžu byť zložené pomocou dvojúrovňovej logiky so skupinou AND brán, nasledovanou OR bránami
- PROM – programovateľná – obsah môže byť zmenený používateľom, nedá sa mazať
- EPROM – programovateľná, mazateľná pomocou ultrafialového svetla
- EEPROM – elektricky mazateľná pici tu

ZHRNUTIE

- stavebné bloky digitálnych zariadení sa používajú v mnohých digitálnych zariadeniach tieto bloky sú aritmetické obvody ako napríklad sčítačky, odčítačky, porovnávačky, posúvačky, násobičky a deličkym, sekvenčné obvody ako napríklad počítadla a posúvacie registre a polia pre pamäte a logiku
- táto kapitola skúmala aj fixed - point a floating - point reprezentácie čísel
- sčítačky sú základom väčšiny aritmetických obvodov
polovičná sčítačka sčíta dva 1-bitové vstupy A a B vyprodukuje dva výstupy
- plná sčítačka má 3 vstupy a vyprodukuje 2 výstupy
rozdiel medzi plnou a polovičnou je v tom, že plná sčítačka prijíma aj vstup C_{in}, ktorý prijíma C_{out} predchádzajúceho stĺpca
- N-bitová sčítačka **(často sa nazýva carry propagate adder (CPA))** spočíta 2 N-bitové vstupy, A a B a , C_{in} a vyprodukuje N-bitový výsledok S a C_{out}
rozdiel v zápise je v tom, že N-bitová sčítačka je nakreslená ako zbernica
- 3 bežné implementácie **CPA**

Ripple Carry:
- najjednoduchší spôsob N-bitovej CPA je spojiť dohromady N plných sčítačiek C_{out} jednej úrovne je C_{in} ďalšej
- je to dobrá aplikácia modulárnosti a jednotnosti, modul (plná sčítačka) je použitá viackrát na vytvorenie väčšieho systému
- jej problémom je to, že je pomalá keď je N veľké

Carry-Lookahead Adder
- najprimitívnejší dôvod toho, že sú ripple-carry sčítačky pomalé je v tom, že musia prejsť cez všetky bit
- CLA rozdelí sčítačky na bloky a rýcho vypočíta Cout bloku hneď keď zistí Cin

Prefix Adder
- najrýchlejšia sčítačka
- pri výbere sčítačky sa musíme rozhodovať podľa toho aké máme priority a čo náš systém potrebuje rýchlejšie sčítačky obsahujú viac hardvéru
- **odčítačka** zneguje druhý vstup a sčíta ho s prvým
- magnitúdová odčítačka odčíta 2 čísla a určí relatívnu hodnotu na základe znamienka výsledku
- **násobička** vytvorí čiastočné výsledky použitím hradiel AND, potom sčítajú bity použitím plných sčítačiek
- zlomky sú reprezentované buď použitím (fixed-point) alebo (floating-point)
- fixed-point = desatinné čísla
  floating-point = vedecká notácia napr: $1{,}1001 * 2^7$
- fixed-point používa bežné aritmetické obvody, avšak floating-point potrebujú zložitejší hardvér na zistenie znamienka, exponentu a mantissy

-väčšie pamäte sú organizovanév poliach slov
- pamäte majú jeden alebo viac portov na čítanie a/alebo písanie slov
- pamäte SRAM, DRAM strácajú svoj stav, keď nie sú napojené na energiu
- SRAM je rýchlejšia ako DRAM ale potrebuje viac tranzistorov

- register file - malé viacportové pole SRAM

- ROMs si uchovávajú svoj stav nadlho

-polia sú tiež bežné cesty na stavbu logiky
- pamäťové polia môžu byť používané ako vyhľadávacie tabuľky na vykonávanie kombinačných funkcií
- PLA , FPGA
- moderné FPGA sú jednoduché na preprogramovanie a sú dosť veľké a lacné na stavbu veľmi sofistikovaných digitálnych systémov
- sú často používané

Topic 7

7.1 Inštrukčno-orientovaná architektúra počítača, MIPS assembler. Typy MIPS inštrukcií: R, I, J. Programovanie architektúry

MIPS: aritmetické a logické inštrukcie.

- architektúra je programátorov pohľad na počítač
- je definovaná množinou príkazov, registrami a pamäťou
- existuje veľa architektúr napr. x86, MIPS...
- prvým krokom na pochopenie architektúry počítača je naučenie sa jej jazyka
- slová v jazyku počítača sa nazývajú inštrukcie, slovná zásoba sa nazýva množina inštrukcií (instruction set)
- všetky programy bežiace na počítači používajú rovnakú množinu príkazov
- všetky zložité programy vedia byť skompilované do série jednoduchých inštrukcií ako sčítavanie, odčítavanie a skok
- hardvér rozumie len 0 a jednotkám takže inštrukcie sú zakódované ako binárne čísla vo formáte nazvanom "machine language" (jazyk počítača)
- mikroprocesory sú digitálne systémy, ktoré čítajú a vykonávajú inštrukcie jazyka počítača
- ľudia ale berú čítanie human language za zložité, a preto sa preferuje reprezentovanie inštrukcií v tzv. jazyku assembly (assembly language).
- rôzne množiny inštrukcií sú ako rozdielne dialekty (majú toho veľa spoločného)
- často sa stáva, že existujú rôzne hardvérové implementácie jednej architektúry
- napr. Intel a AMD predávajú rôzne mikroprocesory pre tú istú architektúru, môžu spustiť rovnaké programy ale používajú iný hardvér (rozdiely v cene, vo výkone...)

4 princípy návrhu MIPS architektúry
1. Simplicity favors regularity (inštrukcie s konzistentným počtom operandov)
2. Make the common case fast (bežný príkaz je rýchlejší ak sú pridané len jednoduché a bežne používané inštrukcie)
3. Smaller is faster (čítanie dát z malej množiny registrov je jednoduchšie ako čítať ich z 1000 registrov)
4. Good design demands good compromises (dobrý dizajn potrebuje dobré kompromisy)

MIPS Assembler

- najčastejšou operáciou, ktorú počítač vykonáva je sčítavanie
Kódy:
a) Prvý kód je kód na sčítanie a druhý na odčítanie

MIPS Assembly Code

```
add a, b, c
```

MIPS Assembly Code

```
sub a, b, c
```

- využite prvého dizajnového princípu inštrukcie by mali mať rovnaký počet operandov v tomto prípade 2 zdrojové a 1 cieľový

- komplexnejší príklad kódu, nachádzajú sa tu aj komentáre v assemblery neexistujú viacriadkové komentáre

MIPS Assembly Code

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

- program potrebuje dočasnú premennú na odloženie čiastkového výsledku používanie viacerých inštrukcií na vykonanie viacerých operácií je príkladom druhého návrhového princípu počítačovej architektúry

- mips obsahuje len jednoduché často používané inštrukcie - *reduced instruction set computer (RISC)*
- architektúry, ktoré obsahujú veľa komplexných funkcií sú - *complex instruction set computers (CISC).*

Operandy: Registre, pamäť a konštanty
- inštrukcie operujú s operandmi
- premenné a,b,c z príkladov hore sú operandy, ale počítač pracuje s nulkami a s jednotkami a nie s premennými
- inštrukcie potrebujú fyzickú lokalitu, z ktorej získavajú binárne dáta
- operandy môžu byť uložené v registroch alebo v pamäti alebo môžu byť konštantami (tie sú uložené v samotnej inštrukcii)
- počítače potrebujú ukladať operandy na viacerých miestach aby sa dala optimalizovať rýchlosť a kapacita dát
- k operandom uloženým v registroch alebo ku konštantám sa dá dostať rýchlo ale ukladajú len malý objem dát
- k ďalším dátam sa musí pristupovať z pamäte , ktoré je veľká ale pomalá
- MIPS architektúra sa nazýva 32 bitová, pretože operuje na 32 bitoch
- používa 32 registrov, ktoré sa nazývajú množinou registrov alebo súborom registrom, čím ich je menej tým sa k ním rýchlejšie pristupuje (využitie 3 návrhového princípu)
- malé súbory registrov sa vytvárajú z malého SRAM poľa
- toto pole používa malý dekóder a bitové polia sú pripojené na relatívne málo pamäťových buniek, teda majú kratšiu kritickú cestu ako veľké pamäte

Registre:
- Príklad:

- inštrukcia and s operandmi registra
- v MIPSe register označujeme pomocou $ znaku
- registre, ktoré sa začínajú s $t nazývame dočasné registre

Code Example 6.4 REGISTER OPERANDS

MIPS Assembly Code

```
# $s0 = a, $s1 = b, $s2 = c
  add $s0, $s1, $s2      # a = b + c
```

Code Example 6.5 TEMPORARY REGISTERS

MIPS Assembly Code

```
# $s0 = a, $s1 = b, $s2 = c, $s3 = d

  sub $t0, $s2, $s3      # t = c - d
  add $s0, $s1, $t0      # a = b + t
```

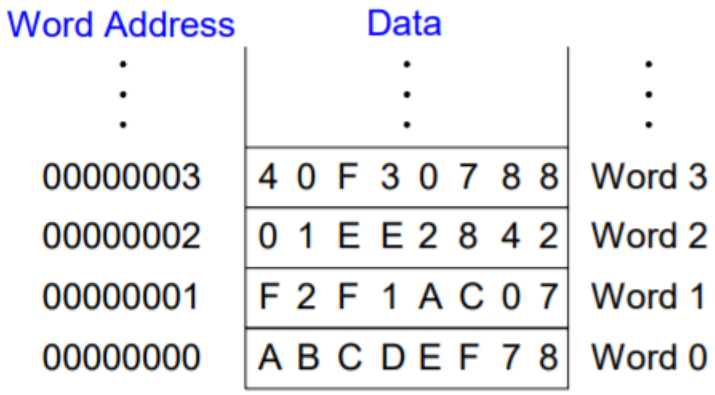
- množina registrov v MIPSe

Table 6.1 MIPS register set

Name	Number	Use
$0	0	the constant value 0
$at	1	assembler temporary
$v0–$v1	2–3	function return value
$a0–$a3	4–7	function arguments
$t0–$t7	8–15	temporary variables
$s0–$s7	16–23	saved variables
$t8–$t9	24–25	temporary variables
$k0–$k1	26–27	operating system (OS) temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	function return address

Pamäť

- príliš veľa dát na uloženie do 32 registrov
- viac dát je uložených v pamäti
- pamäť je veľká ale pomalá
- bežne používané premenné sa nachádzajú v registroch
- všetky 32 bitové slová majú vlastnú adresu byte-addressable



- MIPS používa inštrukciu load word (lw) na načítanie dát z pamäte do registra
- podobne store word inštrukcia (sw) sa používa na zápis dát z registra do pamäte

Code Example 6.6 READING WORD-ADDRESSABLE MEMORY

Assembly Code

```
# This assembly code (unlike MIPS) assumes word-addressable memory
  lw $s3, 1($0)     # read memory word 1 into $s3
```

Code Example 6.7 WRITING WORD-ADDRESSABLE MEMORY

Assembly Code

```
# This assembly code (unlike MIPS) assumes word-addressable memory
  sw  $s7, 5($0)    # write $s7 to memory word 5
```

- tieto kódy nie sú na MIPS...
- až kód dole ukazuje ako načítavať a zapisovať slova v MIPSe

Code Example 6.8 ACCESSING BYTE-ADDRESSABLE MEMORY

MIPS Assembly Code

```
lw $s0, 0($0)       # read data word 0 (0xABCDEF78) into $s0
lw $s1, 8($0)       # read data word 2 (0x01EE2842) into $s1
lw $s2, 0xC($0)     # read data word 3 (0x40F30788) into $s2
sw $s3, 4($0)       # write $s3 to data word 1
sw $s4, 0x20($0)    # write $s4 to data word 8
sw $s5, 400($0)     # write $s5 to data word 100
```

- Byte-addresable pamäte sú organizované buď ako big-endian alebo little - endian

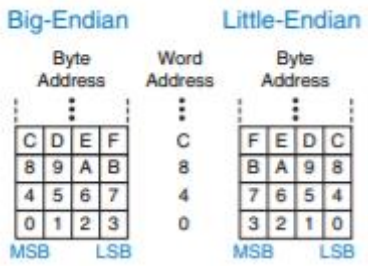


Figure 6.3 Big- and little-endian memory addressing

- lw a sw ukazujú ako používať konštanty
- tieto konštanty sa nazývajú okamžité, pretože ich hodnoty sú okamžite dostupné z inštrukcie a nepotrebujú prístup do registra ani do pamäte
- Add immediate (addi) príkaz je ďalším bežným príkazom v MIPS architektúre

MIPS Assembly Code

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

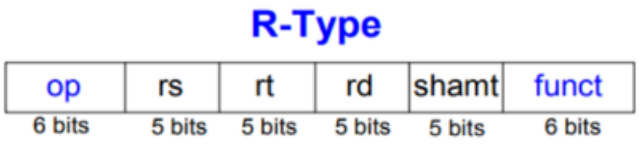
- problémom tohto kódu je v tom, že add a sub používajú 3 operandy z registra, zatiaľčo lw,sw a addi používajú 2 operandy z registra a konštantu, tým pádom porušujú 1 návrhový princíp
- tento problém vedie k poslednému návrhovému princípu Dobrý návrh potrebuje dobré kompromisy

TYPY MIPS INŠTRUKCIÍ

- program napísaný v assembleri je preložený do reprezentácie, ktoré používa iba 1 a 0 (machine language)
- MIPS používa 32 bitové inštrukcie, opäť podľa prvého návrhového princípu je najlepšou možnosťou uložiť všetky inštrukcie ako slová, ktoré sa uložia do pamäte, avšak nie všetky inštrukcie potrebujú 32 bitov - (môže to viesť k zbytočne príliš veľkej zloťitosti)
- MIPS poskytuje 3 typy inštrukcií typ R,I a J
- R - operuje na + 3 registroch
- I - na 2 registroch a na 16bitov okamžitých
- J(jump) - operuje na 26bitoch

Typ R

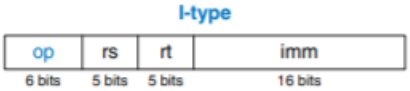
- skratka pre typ register
- R inštrukcie potrebujú 3 registre ako operandy (2 zdroje 1 cieľ)



rs,rt - zdroje
rd - cieľ
op - kód operácie (0 pre typy R inštrukcií)
funct - funkcia s (opcode), hovorí počítaču akú operáciu má vykonať
shamt - počet posunutí pre inštrukciu posunu, inak je 0

Typ I

- skratka pre okamžitý (immediate) typ

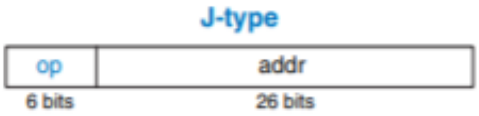


- imm - drží 16 bitovú okamžitú hodnotu, vždy je to zdroj
- rst je niekedy používaný ako cieľ napr pri (addi,lw) ale inak sa používa ako zdroj napríklad pre sw.
- operácie sú vždy determinované pomocou op

Typ J

- je používaný len s inštrukciou skoku

- addr - 26 bitový formát, inak J začína so 6 bitovým op



PROGRAMOVANIE ARCHITEKTÚRY

- programovacie jazyky ako napríklad Java alebo C sa nazývajú vysoko úrovňové jazyky, pretože bežia na vyššej úrovni abstrakcie

Aritmetické/logické inštrukcie

- Logické:

and,or,xor a nor
- sú to inštrukcie typu R

- **and, or, xor, nor**
  - `and`: useful for **masking** bits
    - Masking all but the least significant byte of a value:
      0xF234012F AND 0x000000FF = 0x0000002F
  - `or`: useful for **combining** bit fields
    - Combine 0xF2340000 with 0x000012BC:
      0xF2340000 OR 0x000012BC = 0xF23412BC
  - `nor`: useful for **inverting** bits:
    - A NOR $0 = NOT A
- **andi, ori, xori**
  - 16-bit immediate is zero-extended (*not* sign-extended)
  - `nori` not needed

Logical Instructions Example 1

Source Registers

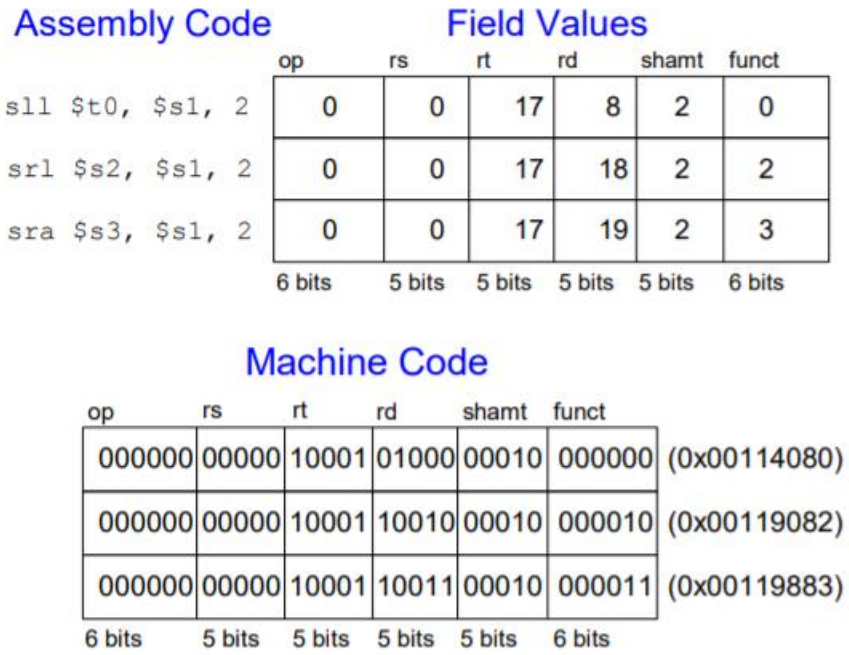
$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code	Result							
`and $s3, $s1, $s2`	$s3	0100	0110	1010	0001	0000	0000	0000	0000
`or $s4, $s1, $s2`	$s4	1111	1111	1111	1111	1111	0000	1011	0111
`xor $s5, $s1, $s2`	$s5	1011	1001	0101	1110	1111	0000	1011	0111
`nor $s6, $s1, $s2`	$s6	0000	0000	0000	0000	0000	1111	0100	1000

- andi, ori, xori - operujú na konštantách (immediates)

Posúvacie(shift) inštrukcie

- tieto inštrukcie posunú hodnotu v registri doľava alebo doprava o maximálne 31 bitov
- sll(shift left logical)
  srl(shift right logical)
  sra(shift right aritmetic)
  srllv(shift left logical variable)
  srlv (shift right logical variable)
  srav(shift right arithmetic variable)



Generovanie konštánt

- 16-bit constants using `addi`:

C Code

```
// int is a 32-bit signed word
int a = 0x4f3c;
```

MIPS assembly code

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

Inštrukcie násobenia a delenia

- násobenie 2 32 bitových čísel produkuje 64 bitový výsledok
- delenie 2 32 bitových čísel vyprodukuje 32 bitový výsledok a 32 bitový zvyšok

- MIPS architektúra má 2 špeciálne registre hi a lo, ktoré sa používajú na uloženie výsledok násobenia a delenia

príkazy: mult a div

7.2 Programovanie architektúry MIPS: podmienený a nepodmienený skok, preklad príkazu if-else, príkazu while, príkazu for.
Životný cyklus programu: od prekladu zdrojového textového kódu až po načítanie vykonateľného kódu do pamäte počítača.
Adresovacie módy.

- výhoda počítača nad kalkulačkou je schopnosť robiť rozhodnutia
- počítač vykonáva rôzne úlohy na základe vstup
- pr: (if/else, switch/case, while...)
- inštrukcie vetvenia modifikujú program tak aby preskočil cez sekciu kódu alebo zopakoval

predchádzajúci kód
- podmienené vetvenie vykoná test a vetví len ak je test TRUE
    vetva ak je rovné (beq)
    vetva ak nie je rovné (bne)
- nepodmienené vetvenie (skok) vetví vždy
    skok(j)
    skokový register (jr)
    skok a prepojenie (jal)

Podmienené vetvenie

- príkaz if hodnotí testovací výraz v zátvorkách

Ak je skúšobný výraz vyhodnotený ako true (nonzero), vyhlásenie (y) vnútri tela if je vykonané.
Ak je testovací výraz vyhodnotený ako false (0), vyhlásenie (y) vnútri tela

príklad:

Code Example 6.12 CONDITIONAL BRANCHING USING `beq`

MIPS Assembly Code

```
  addi  $s0, $0, 4          # $s0 = 0 + 4 = 4
  addi  $s1, $0, 1          # $s1 = 0 + 1 = 1
  sll   $s1, $s1, 2         # $s1 = 1 << 2 = 4
  beq   $s0, $s1, target    # $s0 == $s1, so branch is taken
  addi  $s1, $s1, 1         # not executed
  sub   $s1, $s1, $s0       # not executed

target:
  add   $s1, $s1, $s0       # $s1 = 4 + 4 = 8
```

Code Example 6.13 CONDITIONAL BRANCHING USING `bne`

MIPS Assembly Code

```
  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, target    # $s0 == $s1, so branch is not taken
  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
```

jump

Code Example 6.14 UNCONDITIONAL BRANCHING USING `j`

MIPS Assembly Code

```
  addi  $s0, $0, 4     # $s0 = 4
  addi  $s1, $0, 1     # $s1 = 1
  j     target         # jump to target
  addi  $s1, $s1, 1    # not executed
  sub   $s1, $s1, $s0  # not executed
target:
  add   $s1, $s1, $s0  # $s1 = 1 + 4 = 5
```

Code Example 6.15 UNCONDITIONAL BRANCHING USING `jr`

MIPS Assembly Code

```
0x00002000  addi  $s0, $0, 0x2010  # $s0 = 0x2010
0x00002004  jr    $s0              # jump to 0x00002010
0x00002008  addi  $s1, $0, 1       # not executed
0x0000200c  sra   $s1, $s1, 2      # not executed
0x00002010  lw    $s3, 44($s1)     # executed after jr instruction
```

- príkaz if/else

- if/else vykoná jeden z dvoch blokov kódu v závislosti od podmienky
- ak je podmienka TRUE vykoná sa if blok ak nie tak else
- podobne ako pri teste if assembler testuje opační podmienku ak je test i == j assembler testuje i != j

MIPS Assembly Code

```
# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
  bne $s3, $s4, else   # if i != j, branch to else
  add $s0, $s1, $s2    # if block: f = g + h
  j   L2               # skip past the else block
else:
  sub $s0, $s0, $s3    # else block: f = f - i
L2:
```

Cykly

- for a while
- while - opakuje blok kódu, kým je podmienka FALSE
  - for = while jediný rozdiel je v tom, že for pridáva premennú cyklu

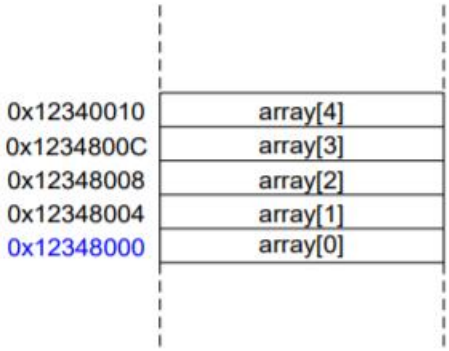
for (initialization; condition; loop operation)
statement

Polia

- polia sú vhodné na prístup k väčšiemu množstvu rovnakých dát
- index (prístup ku konkrétnym prvkom)
- veľkosť (počet prvkov)

príklad

- 5-element array
- **Base address** = 0x12348000 (address of first element, `array[0]`)
- First step in accessing an array: load base address into a register



prístup ku poľu

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

# MIPS assembly code
# 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
```

Funkcie

- vysokoúrovňové programovacie jazyky používajú funkcie na znovupoužitie často používaného kódu na spravenie programu modulárnejším a čitateľnejším
- funkcie majú vstupy(argumenty) a výstup (návratovú hodnotu)
- keď jedna funkcia volá druhú tá volajúca sa nazýva (Caller) a volaná funkcia sa nazýva (Callee)

Konvencie v MIPSE

- volanie funkcie (skoč a napoj, jump and link) jal
- návrat z funkcie skokový register (jump register) jr
- argumenty $a0 - $a3
- návratová hodnota $v0

MIPS assembly code

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

0x00401020 simple: jr $ra
```

príklad funkcie

príklad funkcie s návratovou hodnotou

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       #zavola funkciu
add $s0, $v0. $0 #y vratena hodnota
```

diffofsums:

```
add $t0, $a0, $a1 # t = a0 + a1
add $t1, $a2, $a3 # t1    = a2 + a3
sub $s0, $t1, $t2    # s = t1 - t2
add $t0, $s0, $0 # priradenie vysledku do t0
jr $ra #navrat do callera
```

pozn: treba si vsimnut ze sme pouzili presne 4 argumenty z registra teda maximalny pocet ak by sme prekrocili pocet argumentov, tak argumenty navyse by sa ulozili do takzvaneho Stacku

Stack

- je to pamäť do ktorej si ukladáme lokálne premenné z funkcií
- stack sa zväčšuje, ak procesor potrebuje viac miesta a zmenšuje sa, keď procesor nepotrebuje premenné v ňom uložené
- Stack je LIFO rada
- posledný prídeš prvý ideš preč
- stack pointer $sp ukazuje na vrch stacku
- volanie funkcie nesmie mať bočné efekty ale funkcia diffofsums prepisuje 3 registre vylepšená implementácia pomocou stacku by fungovala tak, že funkcia diffofsums by vytvorila miesto v stacku tým, že by dekrementovala stack pointer
- uložila pomocné premenné do novoalokovaného miesta
- vykoná zbytok funcie
- vymaže všetky premenné zo stacku
- inkrementuje stack pointer

Predrezervované Registre

- MIPS delí registre na rezervované a nerezervované
- $s0 - $s7 - rezervované registre
- $t0 - $t9 - nerezervované registre (dočasné)
- príklad s diffofsums sa dá ešte viac vylepšiť a to tým, že do stacku uložíme len s0, pretože t0 a t1 môžeme použiť ako dočasné registre, takže nemusia byť uložené
- predzervované registre sa nazývajú aj callee-save (preto lebo callee funkcia ich nemôže zmeniť)
- sumár všetkých registrov

Table 6.3 Preserved and nonpreserved registers

Preserved	Nonpreserved
Saved registers: $s0–$s7	Temporary registers: $t0–$t9
Return address: $ra	Argument registers: $a0–$a3
Stack pointer: $sp	Return value registers: $v0–$v1
Stack above the stack pointer	Stack below the stack pointer

Rekurzívne funkcie

- funkcia, ktorá nevolá inú sa nazýva listová funkcia príkladom je funkcia diffofsums
- jej opakom sú nelistové funkcie
- nelistové (nonleaf) funkcie sú zložitejšie, pretože musia uložiť nejakú dočasnú hodnotu, predtým než zavolajú ďalšiu funkciu a potom musia tie registre obnoviť
- rekurzívna funkcia je nelistová funkcia, ktorá volá sama seba
- príkladom je faktoriál

MIPS Assembly Code

```
0x90   factorial:  addi $sp, $sp, -8    # make room on stack
0x94               sw   $a0, 4($sp)     # store $a0
0x98               sw   $ra, 0($sp)     # store $ra
0x9C               addi $t0, $0, 2      # $t0=2
0xA0               slt  $t0, $a0, $t0   # n <= 1 ?
0xA4               beq  $t0, $0, else   # no: goto 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
```

Adresovacie módy

- MIPS používa 5 adresovacích módov

1. Register- Only

- používa registre pre všetky zdroje a destinácie
- R - typy používajú register-only addressing

2. Immediate Addressing

- používa 16 bitov okamžite spolu s registrami ako operandy
- napríklad *addi, lui*

3. Base Addressing

- príkazy prístupu do pamäte napríklad lw (load word) alebo sw (store word)

- efektívna adresa pamäte operandu je nájdená pridaním základnej adresy v registri RS do „sign-extended 16 bit offset“ nájdený v priamom poli.

4. PC - Relative Addressing

- podmienené vetvy používajú PC addressing na špecifikáciu nových hodnôt PC ak je vybratá vetva

5. Pseudo-Direct Addressing

- v priamom adresovaní, je adresa špecifikovaná ako inštrukcia skokové inštrukcie j a jal by najradšej používali priame adresovanie ale J - typ inštrukcií nemá dostatok bitov na špecifikovanie 32 - bitového JTA (jump target addressing), pretože 6 bitov sa používa na opc
- upravené priamé adresovanie sa nazýva pseudo directin

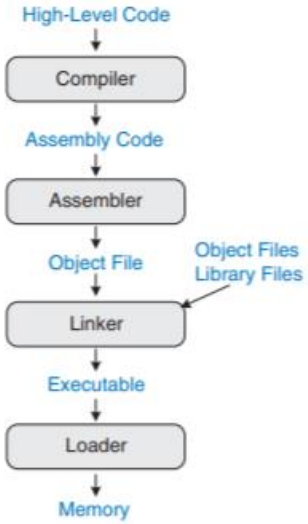
Fázy životného cyklu programu

- **pamäťová mapa** - miesto, ktoré definuje kde je sú uložené kódy, dáta a pamät v stacku
- MIPS má pamäťové rozpätie 2^{32} bytov = 4GB
- slovné adresy sú deliteľné 4 a rozsah majú od 0 do 0xFFFFFFFC
4294967292 - bytov
- MIPS rozdeľuje miesto adresy na 4 segmenty - textový, globálne dáta, dynamické dáta a rezervovaný segment

Preklad a štart programu

- Kompilácia:

kompilátor prekladá vysokoúrovňový kód na assembler kód
.data , .text --- sú klučové slová, ktoré zobrazujú kde začína textový a kde dátový segment



Assembling

- assembler mení assembly kód na objektový súbor, ktorý obsahuje "machine language" kód
- assembler prejde 2x cez kód
    1. Prejdenie - priradenie inštrukcií a nájdenie všetkých symbolov - mená globálnych premenných...
    2. Prechod - assembler produkuje kód machine language

Linkovanie

- väčšina väčších programov obsahuje viac ako jeden súbor
- ak programátor zmení len jeden súbor je strata času opäť kompilovať všetky ostatné súbory
- úlohou linkera je skombinovať všetky súbory do jedného nazvaného spustiteľný

Načitanie
- operačný systém načíta program prečítaním textových segmentov zo spustiteľného súboru z úložiska (HDD) do textového segmentu pamäte

Zhrnutie:
- k tomu aby sme mohli dávať príkazy počítaču musíme hovoriť jeho jazyku
- počítačová architektúra definuje to ako máme rozkazovať procesoru
- MIPS je 32 bitová architektúra, pretože operuje na 32 bitoch, má 32 registrov
- v princípe každý MIPS register sa dá použiť na hocičo ale podľa konvencie sú niektoré registre rezervované
- napríklad register $0 vždy obsahuje konštantu 0 $ra obsahuje návratovú hodnotu po inštrukcii jal a registre $a0 - $a3 a $v0 - $v1 obsahujú argumenty a návratové hodnoty funkcie
- MIPS má bytovo - adresovateľný pamäťový systém s 32-bitovými adresami
- inštrukcie majú dĺžku 32 bitov
- sila definovanie počítačovej architektúry je v tom, že program napísaný pre danú architektúru vie bežať na rôznych implementáciách danej architektúry