Digitalna tehnika

Ljudje v vsakdanjem življenju sicer računamo v **desetiškem** (decimalnem) številskem sestavu, računalniki in druga krmilja pa zaradi uporabljene tehnologije razlikujejo in uporabljajo le dvoje stanj - je napetost, ni napetosti, odprt ventil, zaprt ventil, kar zaznamujemo z dvema znakoma 0 in 1. Ta dva znaka predstavljata **zalogo vrednosti dvojiškega** (binarnega) številskega sestava z osnovo 2.

Šestnajstiški (heksadecimalni) številski sestav je zelo blizu dvojiškemu številskemu sestavu in zato omogoča zelo enostavne pretvorbe med njima v obeh smereh (**štiri binarne števke** zapišemo z eno heksadecimalno števko).
Šestnajstiški številski sestav uporabljamo za bolj pregleden in enostaven prikaz vrednosti, ki so sicer zapisane v računalniku dvojiškem številskem sestavu.

Pretvorbe med številskimi sestavi in kodiranje znakov si oglej v dokumentu "**kodiranje informacij**".

Pravila preklopne algebre (stikalna, Boolova algebra)

V **Boolovi algebri** lahko spremenljivka zavzame le dve vrednosti: 0 in 1.

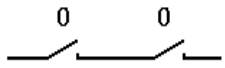
Spremenljivke med seboj povezujemo z operacijami **IN** (prikažemo z znakom ì ali Ê), **ALI** (prikažemo z znakom í ali +) in **NE** (prikažemo z znakom ë ali -).

Za nazornost razumevanja Boolove algebre predstavimo spremenljivke in povezovanje med njimi tudi **s stikali:**

X, X1, A, ... spremenljivke z zalogo vrednosti {0, 1}

0, 1 ... nespremenljivo logično stanje

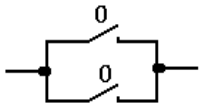
1.) $0 \cdot 0 = 0$



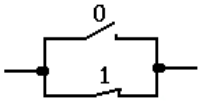
2.) $0 \cdot 1 = 0$

3.) $1 \cdot 1 = 1$

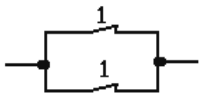
4.) $0 + 0 = 0$



5.) $1 + 0 = 1$

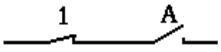


6.) $1 + 1 = 1$

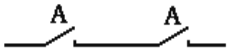


7.) $0 \cdot A = 0$

8.) $1 \cdot A = A$

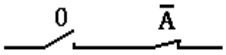


9.) $A \cdot A = A$

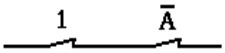


10.) $A \cdot \overline{A} = 0$

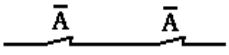
11.) $0 \cdot \overline{A} = 0$



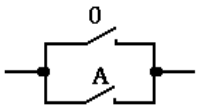
12.) $1 \cdot \overline{A} = \overline{A}$



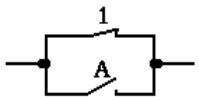
13.) $\overline{A} \cdot \overline{A} = \overline{A}$



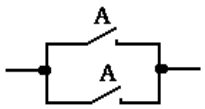
14.) $0 + A = A$



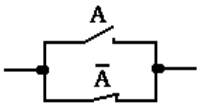
15.) $1 + A = 1$



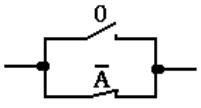
16.) $A + A = A$



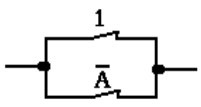
17.) $A + \overline{A} = 1$



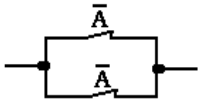
18.) $0 + \overline{A} = \overline{A}$



19.) $1 + \overline{A} = 1$



20.) $\overline{A} + \overline{A} = \overline{A}$

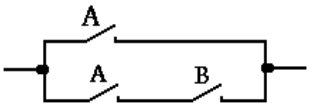


Poenostavitve logičnih izrazov:

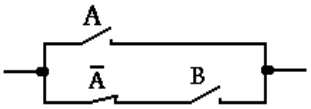
Pri reševanju logičnih enačb si lahko veliko pomagamo s poenostavitvami logičnih izrazov, kar v končni fazi pomeni poenostavitev realiziranega vezja, pri čemer funkcionalnost ostane nespremenjena.

Za boljše razumevanje poenostavitev je prikazana realizacija logične funkcije s stikali.

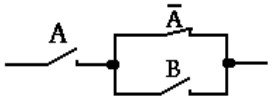
1.) $A + (A \cdot B) = A$



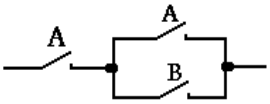
2.) $A + (\overline{A} \cdot B) = A + B$



3.) $A \cdot (\overline{A} + B) = A \cdot B$



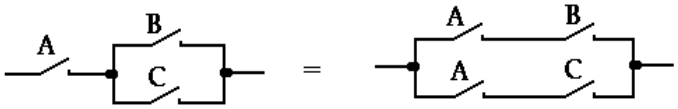
4.) $A \cdot (A + B) = A$



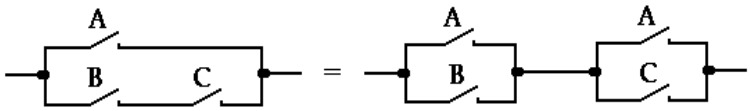
Zakon distribucije

Pri reševanju logičnih enačb in poenostavljanju je koristno uporabiti tudi zakon distribucije.

$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$



$A + (B \cdot C) = (A + B) \cdot (A + C)$



Logični simboli (stikalne enačbe)

Z **logičnimi simboli (stikalnimi enačbami)**, ki so **standardizirani**, prikazujemo **logične funkcije,** ki jih elementi izvedejo glede na vhodne signale.

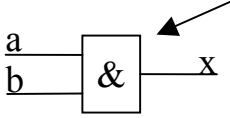
IN (AND)

$x = a \cdot b$

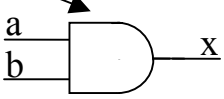
$x = a \wedge b$

$x = a \,\&\, b$

a	b	x
0	0	0
0	1	0
1	0	0
1	1	1



(IEC standard))



(USA standard)

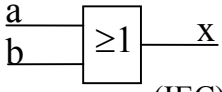
Pravilnostna tabela prikazuje **logične povezave** med vhodnimi (a, b) in izhodnimi spremenljivkami (x). Vhodne spremenljivke zavzamejo **vse možne** kombinacije vrednosti.

ALI (OR)

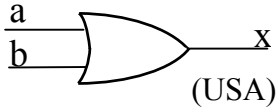
$x = a + b$

$x = a \vee b$

a	b	x
0	0	0
0	1	1
1	0	1
1	1	1



(IEC)

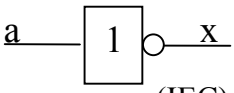


(USA)

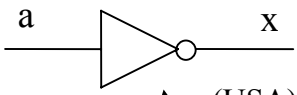
NE (NOT)

$x = \overline{a}$

a	x
0	1
1	0



(IEC)



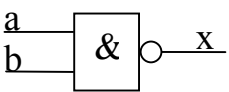
(USA)

inverter, negator

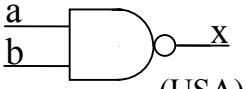
NE - IN (NAND)

$x = \overline{a \cdot b}$

a	b	x
0	0	1
0	1	1
1	0	1
1	1	0



(IEC)

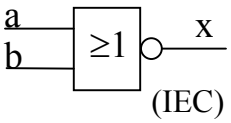


(USA)

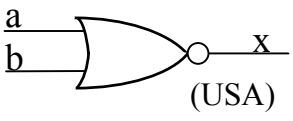
NE - ALI (NOR)

$x = \overline{a + b}$

a	b	x
0	0	1
0	1	0
1	0	0
1	1	0



(IEC)



(USA)

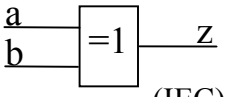
Antivalenca (ekskluzivni OR - XOR)

$z = (a \cdot \overline{b}) + (\overline{a} \cdot b) = a \not\Leftrightarrow b$

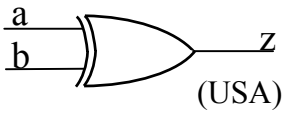
beremo: a XOR b

$z = a \oplus b$

a	b	z
0	0	0
0	1	1
1	0	1
1	1	0



(IEC)



(USA)

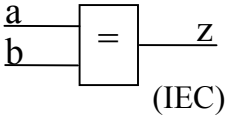
Ekvivalenca (ekskluzivni NOR - XNOR)

$z = (a \cdot b) + (\overline{a} \cdot \overline{b}) = a \Leftrightarrow b$

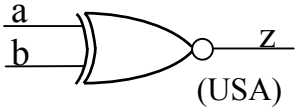
beremo: a XNOR b

$z = \overline{a \oplus b}$

a	b	z
0	0	1
0	1	0
1	0	0
1	1	1



(IEC)



(USA)

Ekvivalenca je negirana antivalenca!

De Morganovi pravili

$$1.)\quad \overline{A \wedge B} = \overline{A} \vee \overline{B}$$

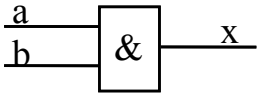
$$2.)\quad \overline{A \vee B} = \overline{A} \wedge \overline{B}$$

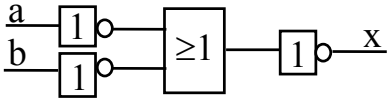
Z uporabo De Morganovih pravil **prevedemo** konjunktivno obliko funkcije (IN) v disjuktivno bliko (ALI) in obratno.

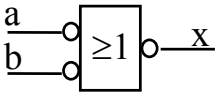
Pri pretvorbi tudi upoštevamo, da je: $\overline{\overline{A}} = A$

Primeri pretvorb logičnih funkcij

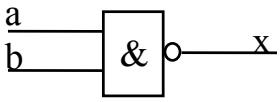
a) 'IN' lahko zamenjamo z 'ALI' in inverterji $\overline{\overline{a \cdot b}} = \overline{\overline{a} + \overline{b}}$

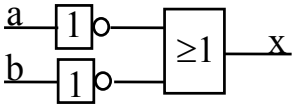




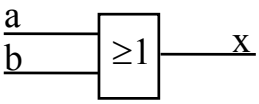


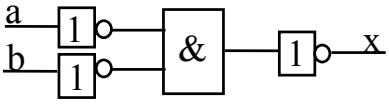
b) 'NE - IN (NAND)' zamenjamo z 'ALI' in inverterji $\overline{a \cdot b} = \overline{a} + \overline{b}$

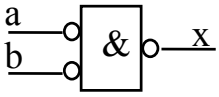




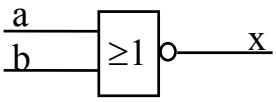
c) 'ALI' zamenjamo z 'IN' in inverterji $\overline{\overline{a + b}} = \overline{\overline{a} \cdot \overline{b}}$

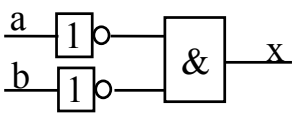






d) 'NE - ALI (NOR)' zamenjamo z 'IN' in inverterji $\overline{a + b} = \overline{a} \cdot \overline{b}$





Pravilnostne tabele, zapis logične funkcije, Karnaugh-Veitchev diagram in realizacija logične funkcije s kombinacijskim vezjem

OPOZORILO!!!

V tem poglavju so uporabljeni **ANSI simboli**, ker je tako slika v omejenem prostoru bolj pregledna.
V praksi so dovoljeni **IEC simboli**, ki jih uporabljamo tudi pri pouku predmeta RSM.

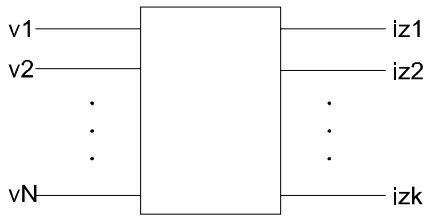
Kombinacijsko vezje je vezje, katerega izhodni signali so odvisni od kombinacij vhodnih signalov.

$$iz_1 = f_1\ (v_1,v_2,\ldots,v_N)$$
$$iz_2 = f_2\ (v_1,v_2,\ldots,v_N)$$
$$\vdots$$
$$iz_k = f_k\ (v_1,v_2,\ldots,v_N)$$

v … vhod
iz … izhod
N … število vhodnih spremenljivk (signalov)
k … število izhodnih spremenljivk (signalov)

Število izhodnih spremenljivk **ni odvisno** od števila vhodnih spremenljivk.

Bločna predstavitev krmilja, pri katerem so izhodi funkcija vhodnih spremenljivk



Postopek, ki pripelje do realizacije kombinacijskega vezja bomo najbolje spoznali iz primera naloge:

Dva delavca vzdržujeta štiri zahtevne stroje A, B, C in D. Delavec D_1 je usposobljen le za popravila strojev C in D, delavec D_2 pa zna popravljati vse. V vsakem trenutku se lahko pokvari katerikoli stroj.. Poišči funkcijo in realiziraj vezje, ki bo javilo alarm, ko delavca ne bosta mogla sprejeti vseh popravil.

Na osnovi zastavljenega problema bomo sestavili **pravilnostno tabelo** oz. **tabelo vhodnih stanj.**
Vhodne spremenljivke predstavljajo štirje stroji A, B, C in D, **izhodna spremenljivka** pa je alarm. Vseh vhodnih kombinacij je 2^4, torej 16.

vhodne spremenljivke				*izhodna spremenljivka*	*minterme*	*maksterme*
A	B	C	D	alarm		
0	0	0	0	0	$\overline{A}\cdot\overline{B}\cdot\overline{C}\cdot\overline{D}$	$A+B+C+D$
0	0	0	1	0	$\overline{A}\cdot\overline{B}\cdot\overline{C}\cdot D$	$A+B+C+\overline{D}$
0	0	1	0	0	$\overline{A}\cdot\overline{B}\cdot C\cdot\overline{D}$	$A+B+\overline{C}+D$
0	0	1	1	0	$\overline{A}\cdot\overline{B}\cdot C\cdot D$	$A+B+\overline{C}+\overline{D}$
0	1	0	0	0	$\overline{A}\cdot B\cdot\overline{C}\cdot\overline{D}$	$A+\overline{B}+C+D$
0	1	0	1	0	$\overline{A}\cdot B\cdot\overline{C}\cdot D$	$A+\overline{B}+C+\overline{D}$
0	1	1	0	0	$\overline{A}\cdot B\cdot C\cdot\overline{D}$	$A+\overline{B}+\overline{C}+D$
0	1	1	1	1	$\overline{A}\cdot B\cdot C\cdot D$	$A+\overline{B}+\overline{C}+\overline{D}$
1	0	0	0	0	$A\cdot\overline{B}\cdot\overline{C}\cdot\overline{D}$	$\overline{A}+B+C+D$
1	0	0	1	0	$A\cdot\overline{B}\cdot\overline{C}\cdot D$	$\overline{A}+B+C+\overline{D}$
1	0	1	0	0	$A\cdot\overline{B}\cdot C\cdot\overline{D}$	$\overline{A}+B+\overline{C}+D$
1	0	1	1	1	$A\cdot\overline{B}\cdot C\cdot D$	$\overline{A}+B+\overline{C}+\overline{D}$
1	1	0	0	1	$A\cdot B\cdot\overline{C}\cdot\overline{D}$	$\overline{A}+\overline{B}+C+D$
1	1	0	1	1	$A\cdot B\cdot\overline{C}\cdot D$	$\overline{A}+\overline{B}+C+\overline{D}$
1	1	1	0	1	$A\cdot B\cdot C\cdot\overline{D}$	$\overline{A}+\overline{B}+\overline{C}+D$
1	1	1	1	1	$A\cdot B\cdot C\cdot D$	$\overline{A}+\overline{B}+\overline{C}+\overline{D}$

V splošnem je v pravilnostni tabeli **2^N vhodnih kombinacij**.

V pravilnostno tabelo torej vnesemo stanja izhodnih spremenljivk v odvisnosti od vhodnih kombinacij.
V našem primeru delavca ne bosta mogla sprejeti vseh popravil, ko bodo v okvari trije stroji ali pa prva dva, ker je le eden od delavcev usposobljen za popravilo prvih dveh strojev.

Do zapisa logične funkcije, ki podaja odvisnost izhodne spremenljivke od vhodnih spremenljivk, lahko pridemo:

- neposredno iz **pravilnostne tabele**
- z uporabo **Veitchevega diagrama**, v katerega vnesemo stanja izhodnih spremenljivk iz pravilnostne tabele

Običajno je uporaba K-V diagrama **bolj enostavna** in **hitrejša** in se uporablja pri manjšem številu vhodnih spremenljivk.

Zapis logične funkcije neposredno iz pravilnostne tabele

Logično funkcijo lahko iz tabele dobimo na dva različna načina:

- kot **PDNO** (PDO) - (**P**opolna **D**isjunktivna **N**ormalna **O**blika) funkcije, ki jo dobimo z disjunktivno povezavo **minterm**
- kot **PKNO** (PKO) - (**P**opolna **K**onjunktivna **N**ormalna **O**blika) funkcije, ki jo dobimo s konjunktivno povezavo **maksterm**

PDNO zapis logične funkcije iz tabele

Do tega zapisa pridemo, če v pravilnostni tabeli na mestih, kjer izhodna spremenljivka zavzame vrednost "logična ena", izpišemo **minterme** in jih povežemo z **disjunkcijo- ALI operatorjem**.
Minterma je konjunktivna (IN) logična povezava vseh vhodnih spremenljivk. Na mestih, kjer vhodne spremenljivke zavzamejo vrednost "0", so le-te negirane.

Za naš primer bi torej zapisali minterme na mestih, kjer je izhod 1, kot:

$$m1 = \overline{A} \cdot B \cdot C \cdot D$$
$$m2 = A \cdot \overline{B} \cdot C \cdot D$$
$$m3 = A \cdot B \cdot \overline{C} \cdot \overline{D}$$
$$m4 = A \cdot B \cdot \overline{C} \cdot D$$
$$m5 = A \cdot B \cdot C \cdot \overline{D}$$
$$m6 = A \cdot B \cdot C \cdot D$$

Logična funkcija, ki določa alarm, bi imela naslednji PDNO zapis:

$$alarm = (\overline{A} \cdot B \cdot C \cdot D) + (A \cdot \overline{B} \cdot C \cdot D) + (A \cdot B \cdot \overline{C} \cdot \overline{D}) + (A \cdot B \cdot \overline{C} \cdot D) + (A \cdot B \cdot C \cdot \overline{D}) + (A \cdot B \cdot C \cdot D)$$

Tak zapis lahko skrajšamo z uporabo **pravil preklopne algebre** in dobimo **minimiziran zapis (MDO)**:

$$alarm = A \cdot B \cdot \overline{C} \cdot (D + \overline{D}) + A \cdot B \cdot C \cdot (D + \overline{D}) + \overline{A} \cdot B \cdot C \cdot D + A \cdot \overline{B} \cdot C \cdot D$$
$$\Downarrow$$
$$alarm = A \cdot B \cdot (C + \overline{C}) + \overline{A} \cdot B \cdot C \cdot D + A \cdot \overline{B} \cdot C \cdot D$$
$$\Downarrow$$
$$alarm = A \cdot (B + \overline{B} \cdot C \cdot D) + \overline{A} \cdot B \cdot C \cdot D$$
$$\Downarrow$$
$$alarm = A \cdot (B + C \cdot D) + \overline{A} \cdot B \cdot C \cdot D$$
$$\Downarrow$$
$$alarm = A \cdot B + A \cdot C \cdot D + \overline{A} \cdot B \cdot C \cdot D$$
$$\Downarrow$$
$$alarm = B \cdot (A + \overline{A} \cdot C \cdot D) + A \cdot C \cdot D$$
$$\Downarrow$$
$$alarm = B \cdot (A + C \cdot D) + A \cdot C \cdot D$$
$$\Downarrow$$
$$alarm = (A \cdot B) + (B \cdot C \cdot D) + (A \cdot C \cdot D)$$

V primeru, da bi izhodna spremenljivka zavzela vrednost "0" pri majhnem številu vhodnih kombinacij, bi bilo enostavneje zapisati funkcijo z mintermi na mestih, kjer je vrednost izhodne spremenljivke "0". V takem primeru bi disjunktivna povezava minterm podajala negirano funkcijo alarma, torej $\overline{alarm}$.

Zapis minimizirane (poenostavljene) logične funkcije iz Karnaugh-Veitchevega (K-V) diagrama

Veitchev diagram uporabljamo za grafično predstavitev stanj izhodnih spremenljivk v odvisnosti od kombinacij vhodnih spremenljivk in **za poenostavitve logičnih funkcij**.

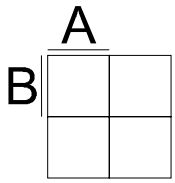
K-V diagram sestavlja 2^N polj za N vhodnih spremenljivk. Vhodne spremenljivke zapišemo v diagram tako, da:

- vsaka vhodna spremenljivka pokriva **polovico vseh polj** v diagramu
- vsaka vhodna spremenljivka pokriva **polovico polj vsake druge** vhodne spremenljivke

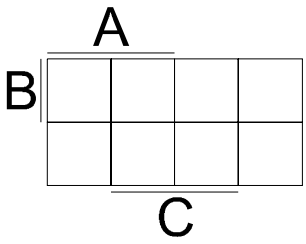
Običajno uporabljamo K-V diagrame za največ 4 ali 5 vhodnih spremenljivk, ker sicer postanejo nepregledni.

Značilne oblike diagramov so prikazane na spodnjih slikah:

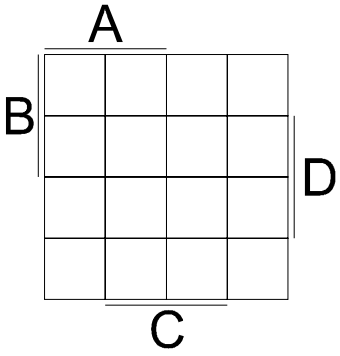
Za 2 vhodni spremenljivki:



Za 3 vhodne spremenljivke:

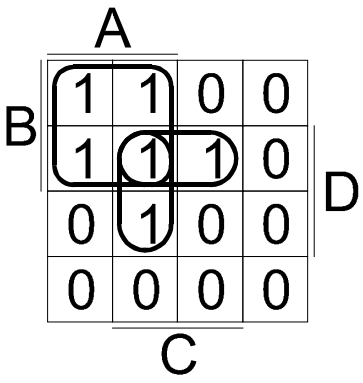


Za 4 vhodne spremenljivke:



Za 5 vhodnih spremenljivk bi pripravili K-V diagram tako, da bi dva diagrama za 4 spremenljivke z zrcalnim razporedom polj postavili drugega poleg drugega.

Za naš primer bi imel K-V diagram naslednjo obliko:



Iz tako dobljenega K-V diagrama lahko zelo enostavno dobimo zapis logične funkcije v poenostavljeni - **minimizirani obliki MDO** (Minimalni Disjunktivni Obliki), to je v najenostavnejši obliki.

Pravila, ki se jih pri tem držimo:

- poiščemo skupine - polja čimveč sosednjih "1" v vodoravnih in navpičnih smereh, jih označimo in zapišemo
- vsako polje lahko vsebuje le 2, 4, 8, torej 2^N enic
- polja se lahko medsebojno prekrivajo
- čimvečje je polje, tembolj enostaven je njen logični zapis.
- K-V diagram si predstavljamo zvit v navpični in vodoravni smeri, zato so enice, ki ležijo na robovih, sosednje

Iz našega diagrama tako dobimo naslednji MDO zapis logične funkcije:

$$alarm = (A \cdot B) + (B \cdot C \cdot D) + (A \cdot C \cdot D)$$

Realizacija logičnega vezja

Tak zapis logične funkcije predstavlja osnovo za realizacijo vezja, ki bo dajalo na izhodu (-ih) pričakovan odziv na osnovi vhodnih signalov.

Vsako vezje lahko realiziramo le z **NAND vrati**, **NOR vrati** ali **kombinacijo** obeh vrst vezij.

Če želimo npr. vezje realizirati le **z uporabo NAND vrat**, bomo logično enačbo preoblikovali tako, da bodo v njej nastopali le operatorji predstavljeni z "**·**", torej **AND povezave**. Logične operatorje "+" moramo torej **zamenjati** z operatorji "·". Pri preoblikovanju si pomagamo z De Morganovimi pravili.

Tako v našem primeru poteka postopek pretvorbe na naslednji način:

$$alarm = (A \cdot B) + (B \cdot C \cdot D) + (A \cdot C \cdot D)$$

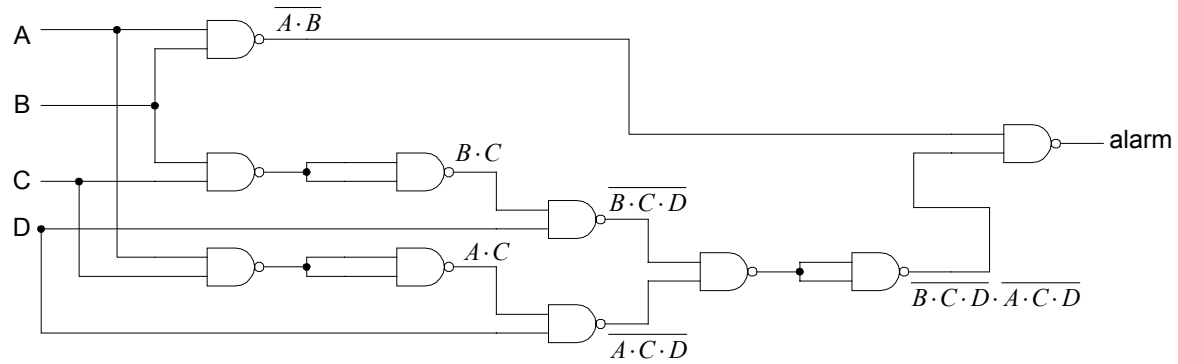
$$\Downarrow$$

$$alarm = \overline{\overline{(A \cdot B) + (B \cdot C \cdot D) + (A \cdot C \cdot D)}}$$

$$\Downarrow$$

$$alarm = \overline{\overline{(A \cdot B)} \cdot \overline{(B \cdot C \cdot D)} \cdot \overline{(A \cdot C \cdot D)}}$$

Na osnovi te enačbe lahko sestavimo logično vezje:



Ostane nam še **preverjanje** pravilnosti delovanja vezja. To izvedemo tako, da na vhodih generiramo različne vhodne kombinacije iz pravilnostne tabele in na izhodu opazujemo odziv. Odziv (izhodni signal) se mora natančno ujemati s stanji v pravilnostni tabeli.

Na enak način pretvorimo logično enačbo tako, da vsebuje le operatorje "+", čemur nato sledi realizacija z **NOR vrati.**

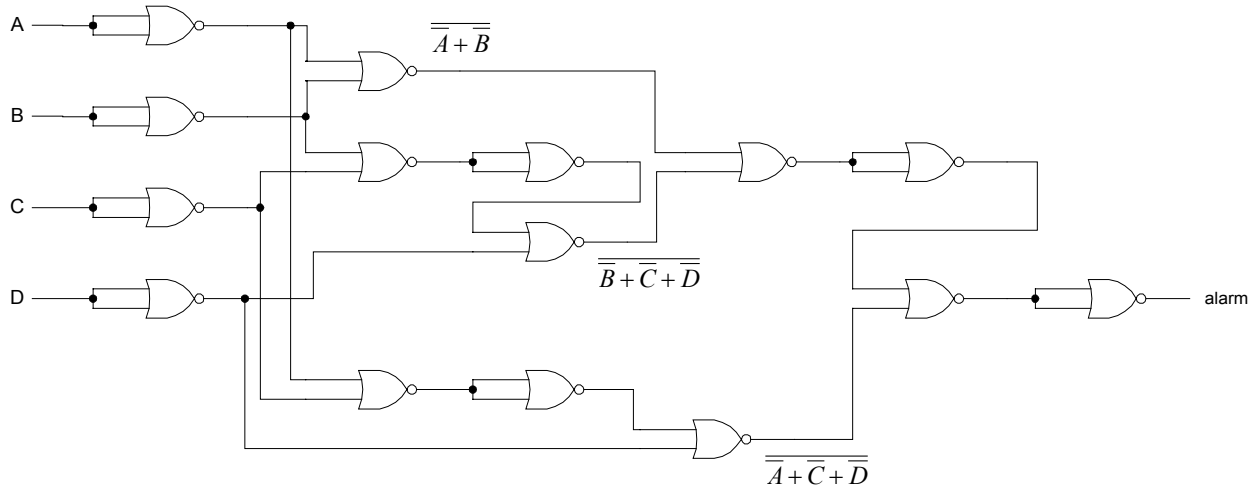
$$alarm = (A \cdot B) + (B \cdot C \cdot D) + (A \cdot C \cdot D)$$

$$\Downarrow$$

$$alarm = \overline{\overline{(A \cdot B)}} + \overline{\overline{(B \cdot C \cdot D)}} + \overline{\overline{(A \cdot C \cdot D)}}$$

$$\Downarrow$$

$$alarm = \overline{(\overline{A} + \overline{B})} + \overline{(\overline{B} + \overline{C} + \overline{D})} + \overline{(\overline{A} + \overline{C} + \overline{D})}$$



Obe vezji sta **enakovredni**, saj opravljata isto funkcijo.

Naloge:

- Realiziraj logično vezje z 2-vhodnimi NAND vrati tako, da bo javilo, če bo 3-bitno število praštevilo.

- Izdelaj logično vezje, ki bo izvajalo množenje dveh dvobitnih števil in pri tem dajalo štiribitni rezultat. Uporabi n-vhodna NOR vrata.

  Namig: uporabi ločen K-V diagram za vsak izhodni bit.

- Z dvovhodnimi NAND vrati realiziraj vezje, ki bo predstavljalo 2/4 dekoder.

- Izdelaj logično vezje, ki bo povedalo, če je število, predstavljeno s štirimi biti, BCD število.

- Realiziraj logično vezje, ki bo omogočilo vžig motorja le v primeru, če bo ključ v ključavnici, voznik na svojem sedežu in pripet z varnostnim pasom. Če je v avtu sopotnik, mora biti tudi on pripet z varnostnim pasom.

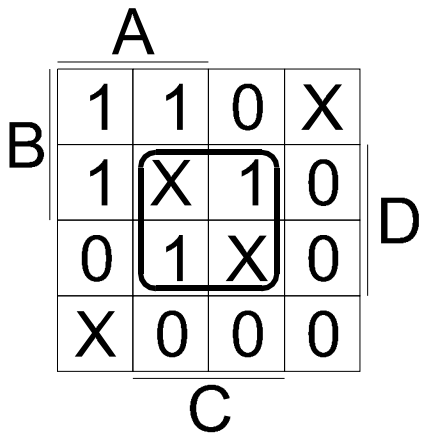
- Imamo elektronsko ključavnico, ki deluje s presvetljevanjem. Zapiši logično funkcijo, ki bo aktivirala alarm, če bo kdo poskušal odpreti ključavnico z drugo kombinacijo kot je pravilna, ki je 11001 (ABCDE). V mirovnem stanju so vsa mesta presvetljena, zato je signal 11111.

  Rešitev: $\overline{A \cdot B \cdot \overline{C} \cdot \overline{D} \cdot E + A \cdot B \cdot C \cdot D \cdot E}$

Redundantne kombinacije

Včasih se zgodi, da se katere od vhodnih kombinacij sploh ne morejo pojaviti. Kot primer lahko navedemo mejni stikali pri osebnem dvigalu. Nikoli nista aktivirani hkrati stikali za začetni in končni položaj dvigala. Taki kombinaciji vhodnih spremenljivk pravimo ***redundantna (odvečna)***. Tako se v BCD kodi tudi nikoli ne pojavijo kombinacije 1111, 1110, 1101, ...

Redundantne kombinacije je možno koristno uporabiti pri poenostavitvah funkcij. Ker se nikoli ne pojavijo, lahko privzamemo, da bodo izhodi pri teh kombinacijah zavzeli vrednost "0" ali "1", kar nam pač ustreza. V pravilnostni tabeli označimo tak izhod z "**X**" **(don't care position).**



Če v zgornjem K-V diagramu vzamemo dve redundantni kombinaciji kot logični enici, bomo pri zapisu funkcije imeli skupino štirih enic, ki jo je mogoče zelo enostavno zapisati kot:

$$f = C \cdot D$$

Tako smo prišli do minimalnega zapisa funkcije.

Logična vezja s spominom

Kombinacijska vezja lahko spremenimo tako, da stanje izhodnih spremenljivk v času t_{n+1} ni odvisno le od kombinacij vhodnih spremenljivk ampak ***tudi od notranjega stanja v vezju*** pred spremembo vhodnih spremenljivk v času t_n . Kombinacijsko vezje smo torej **dopolnili s spominom** in dobili **sekvenčno vezje**.

Iz kombinacijskega vezja izdelamo pomnilniško vezje tako, da povežemo **izhode nazaj na vhode**.

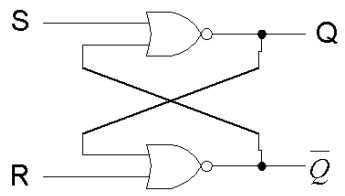
Osnovna pomnilniška vezja so **flip-flopi** in **zapahi (latch)**, ki so **bistabilna** vezja, saj preklapljajo med dvemi stabilnimi stanji (flip----flop).

Vsem FF in zapahom je skupno:

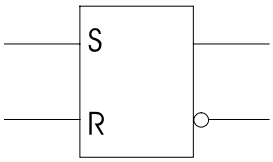
- v vsakem trenutku se nahaja v enem od dveh možnih stanj. V tem stanju **se izhod zadrži,** dokler ga sprememba na vhodu ne preklopi v drugo stanje. V bistvu je to 1-bitni pomnilnik
- vsak ima **dva izhoda** Q in $\overline{Q}$, ki sta **komplementarna** drug drugemu

Najbolj enostaven spominski element je **SR flip-flop** , ki ga pogosto imenujemo tudi **asinhronski SR flip-flop.**

Sestavimo ga lahko iz dvojih dvovhodnih NOR vrat.



S	R	Q	$\overline{Q}$
0	0	Q_0	$\overline{Q}_0$
0	1	0	1
1	0	1	0
1	1	prepovedano	

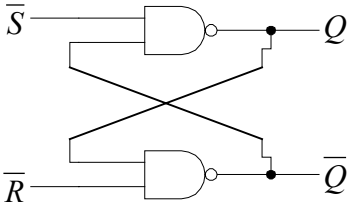


Že oznaka flip-flopa pove bistvo delovanja. Eden od vhodov je **Set** vhod, drugi pa **Reset** vhod. Logična "1" na Set vhodu postavi izhod Q v logično stanje "1", logična "1" na Reset vhodu pa briše Q izhod v logično stanje "0". Če sta na obeh vhodih nizka logična nivoja, ne pride na izhodu do spremembe, torej vezje pomni prejšnje stanje.
Uporaba aktivnih signalov na obeh vhodih hkrati **ni dovoljena**, ker je po vrnitvi vhodov v stanje "0", stanje na izhodih nepredvidljivo, pa tudi oba izhoda sta v stanju "1", in nista več komplementarna.

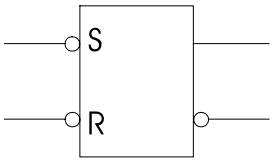
Analizo delovanja SR FF pričnemo z logično "1" na enem od vhodov, ker le tako izločimo vpliv drugega vhoda v NOR vrata oz. vpliv notranjega stanja, ki pa ni znano.

Delovanje FF lahko opišemo s pravilnostno tabelo (tabelo stanj), ki je predstavljena zgoraj ob sliki RS flip-flopa.

Tudi RS flip-flop lahko sestavimo **iz NAND vrat**. Od prejšnjega FF z NOR vrati se razlikuje po vhodnih signalih, ki so negirani in je zato aktivni signal na "set" in "reset" vhodu logična "0". Vendar to razliko znamo zelo hitro odpraviti z dodatnim inverterjem (negatorjem) na vsakem vhodu.



$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$
0	0	prepovedano	
0	1	1	0
1	0	0	1
1	1	Q_0	$\overline{Q}_0$



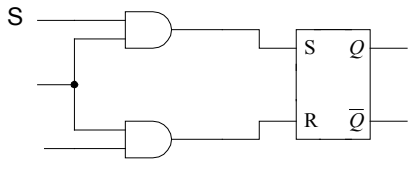
V tem primeru pa analizo vezja pričnemo z logično "0" na vhodu "set" ali "reset", ker v takem primeru drugi signal na vhodu NAND vrat nima vpliva.

Flip-flopi, pri katerih je preklop odvisen **le od stanj** na S in R vhodih, so ***asinhroni SR oz. RS flip-flopi***.

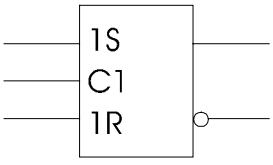
Delovanje RS FF bi lahko primerjali z delovanjem izmeničnega stikala. Če ga potisnemo v eno smer, se luč prižge, če ga potisnemo v drugo smer, luč ugasne. Če roko umaknemo, ostane prejšnje stanje nespremenjeno in se obdrži (pomnjenje!). Ni pa dovoljeno hkrati ugašati in prižigati luči.

Včasih pa je potrebno, da v določenem času onemogočimo vpliv vhodov S in R. RS flip-flop preoblikujemo tako, da preko vrat na vsakega od obeh vhodov dovedemo še nek signal, ki mu pravimo ***urin signal, taktni signal, clock signal ali sinhronizacijski signal T***.
Flip-flop torej lahko preklaplja **le**, ko bo taktni signal v **aktivnem stanju** (v tem primeru v stanju "1").

Tako smo dobili ***statični*** *RS flip-flop*. Ta RS flip-flop je krmiljen ***s stanjem*** oz. ***nivojem taktnega signala T.***

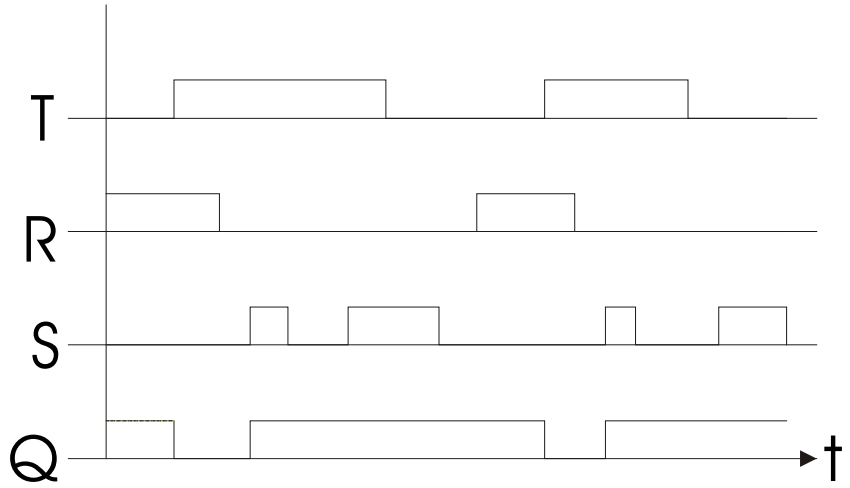


T	S	R	Q
0	0	0	Q_0
0	0	1	Q_0
0	1	0	Q_0
0	1	1	Q_0
1	0	0	Q_0
1	0	1	0
1	1	0	1
1	1	1	prepovedano



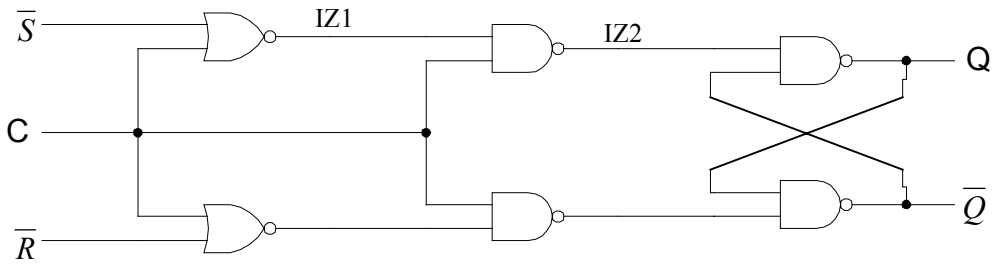
Delovanje digitalnih vezij lahko prikažemo tudi s **časovnim diagramom**, ki prikazuje funkcijske povezave med vhodi in izhodi v odvisnosti od časa. Uporabo časovnega diagrama si bomo ogledali na naslednjem primeru.

Primer:
Na osnovi signalov na vhodih RS flip-flopa, ki so prikazani v obliki časovnega diagrama, določi odziv FF na njegovem izhodu Q. Pred začetkom opazovanju se je izhod Q nahajal v stanju "1".

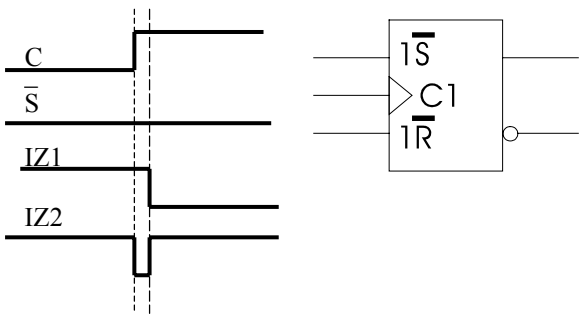


Še ena vrsta RS flip-flopa je pogosto v uporabi in se razlikuje od doslej predstavljenih v načinu krmiljenja: to je ***dinamični RS flip-flop***.

To je FF, ki na vhodu tudi sprejema urin signal, toda s to razliko, da vezje v tem FF reagira **le na *spremembo stanja urinega signala*,** ko logični nivo "0" preide v logično stanje "1". To spremembo imenujemo *tudi* ***prednja (pozitivna) strmina*** *ali* ***prednja fronta*** *ali* ***prednji bok***. Poudariti je treba, da morata biti signala na vhodih R in S *že pripravljena*, torej v pravih stanjih. Stanji na teh dveh vhodih bosta povzročili prehod v želeno stanje, ko bo prišlo do spremembe urinega signala. Take vrste FF predstavlja primer ***sinhronega*** pomnilniškega vezja.

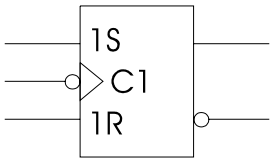


C	$\overline{S}$	$\overline{R}$	Q
↑	0	0	prepovedano
↑	0	1	1
↑	1	0	0
↑	1	1	Q_0



Trikotnik v simbolu na urinem vhodu C opozarja na proženje ob pozitivni strmini.

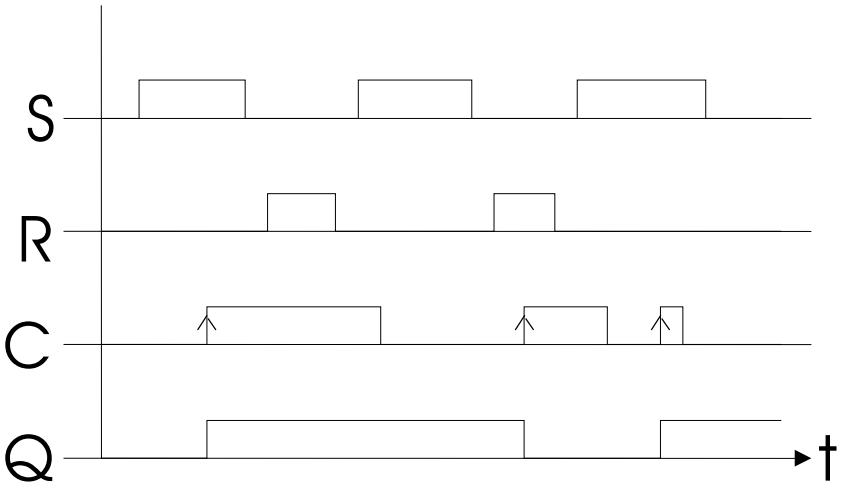
V SR flip-flopu na zgornji shemi, ki deluje na prednjo fronto, izkoristimo za generiranje impulza, ki povzroči proženje FF, čas, ki ga signal potrebuje, da se razširi z vhoda NOR vrat na njihov izhod (propagation delay). Ta čas je zelo kratek, toda trajanje ni pomembno (glej prikaz signalov IZ1 in IZ2 na zgornji shemi!).
Seveda pa je FF lahko narejen tudi tako, da reagira na *zadnjo (negativno) strmino (fronto)* urinega impulza.
V simbolu to pokažemo s krožcem (negacijo) na urinem vhodu.



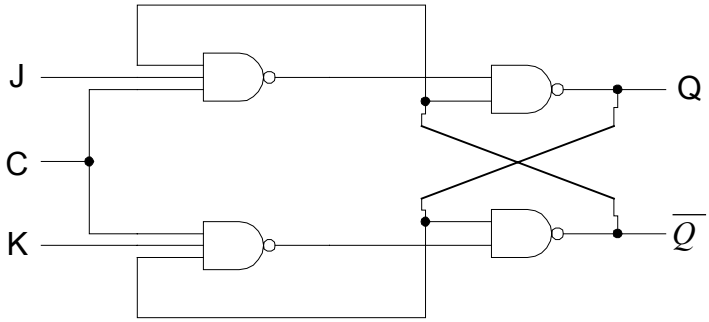
Taka vrsta flip-flopa je zelo primerna za medsebojno povezovanje več flip-flopov. Z urinim signalom zagotovimo, da vsi FF preklopijo istočasno, torej je njihovo delovanje ***sinhronizirano***.

Primer:

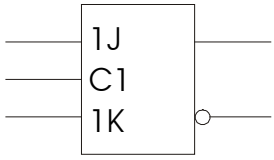
Dopolni časovni diagram sinhronega RS flip-flopa, ki deluje na prednjo strmino urinega signala, s potekom izhodnega signala Q. Upoštevaj, da do preklopov FF lahko pride le ob prednjih strminah taktnega signala. Pred opazovanjem je FF v logičnem stanju "0".



Poleg RS flip-flopov, ki imajo to neugodno lastnost, da ne smeta biti oba vhoda R in S hkrati v visokem logičnem stanju, poznamo nekoliko izpopolnjeno verzijo, to je **JK flip-flop**. V bistvu je to modificiran RS flip-flop, pri katerem je vlogo S vhoda prevzel J, vlogo R vhoda pa K vhod, s tem, da je dovoljeno tudi stanje visokega nivoja na obeh vhodih hkrati**. V takem primeru se stanje na izhodu invertira glede na predhodno stanje.** Vpliv na preklop ima tisti vhod, ki more spremeniti izhodno stanje. Če se izhod nahaja v stanju "0", lahko vpliva J vhod in ga preklopi v "1".



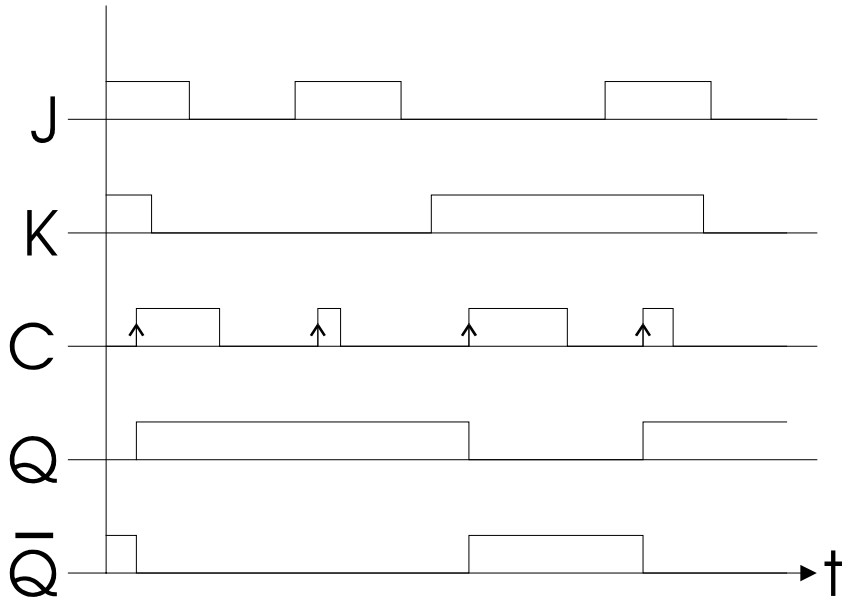
J	K	Q
0	0	Q_0
0	1	0
1	0	1
1	1	$\overline{Q}_0$



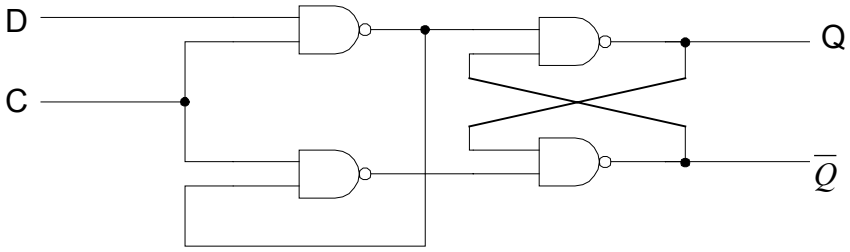
Analizo vezja pričnemo s predpostavko, da sta vhoda J in K v stanju J = K = 1, izhod Q pa tudi v visokem logičnem stanju.

Primer:

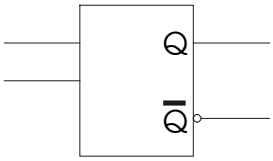
Ugotovi, kakšen bo potek signala na izhodih JK FF v odvisnosti od dogajanja na vhodih in vriši signal v zgornji časovni diagram. Pred spremembo je bil izhod Q v logičnem stanju "0". JK FF deluje na prvo strmino.



Iz RS flip-flopa lahko z drobno spremembo pridemo do **D flip-flopa oz. D pomnilne celice**. Za razliko od RS FF ima D celica en sam vhod, ki mu pravimo *D vhod*. Tako tudi ne moreta biti oba vhoda hkrati v visokem stanju kot pri RS. Drugi vhod v vezju D celice nadomestimo z negirano vrednostjo D vhoda. *Iz vzbujalne ali karakteristične tabele* in z analizo vezja lahko povzamemo, da se v D celici ob vsakem taktnem impulzu stanje z vhoda prepiše na izhod. Ko taktnega signala ni več, to stanje na izhodu ostane nespremenjeno- se zapahne, zato takemu tipu vezja pravimo tudi *zapah (latch).* D flip-flop je tipična pomnilna celica, ki si zapomni stanje na svojem vhodu.

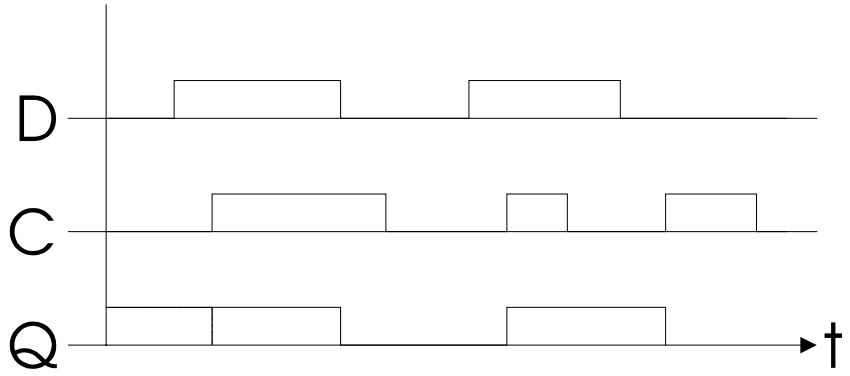


C	D	Q
1	0	0
1	1	1



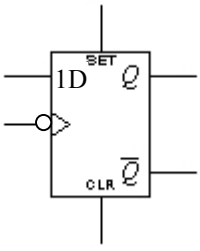
Do pomnjenja pride ko pade C na 0, torej ob zadnji strmini C signala.

Primer:
Pokaži časovno spreminjanje izhodnega stanja pri naslednjem poteku vhodnih signalov.



Vsaka sinhronska pomnilna celica ima lahko poleg sinhronih vhodov tudi *asinhrone*. **Asinhroni so zato, ker delujejo *neodvisno od takta*.** Ti vhodi imajo največkrat imeni "***clear***" in "***preset***". Te vrste vhodi služijo predvsem za postavitev vezja v želeno začetno stanje. S "clear" vhodom postavimo izhod vezja Q v stanje "0", s "preset" vhodom pa v stanje "1".

Primer simbola:

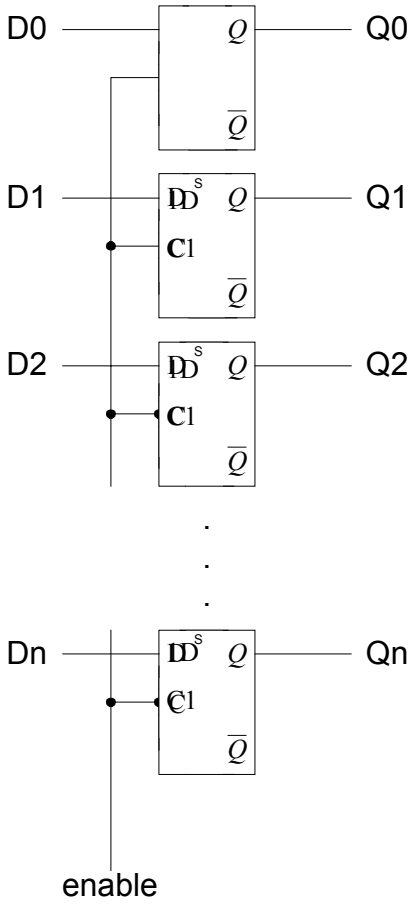


Registri kot sekvenčna vezja

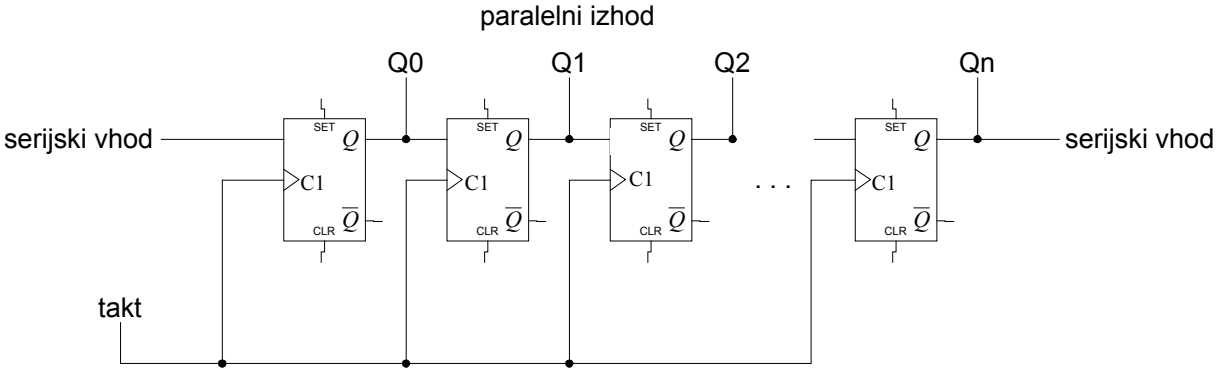
Če več flip-flopov povežemo vzporedno ali zaporedno, dobimo register. Glede na način delovanja in uporabo ločimo:

- registre *za shranjevanje podatkov - pomnjenje*, za kar uporabljamo **zapahe (latch**)
- **pomikalne registre (shift)** za *pomikanje vsebin*.

Najpogosteje se srečujemo pri mikroprocesorjih prav z registri-zapahi. To je npr. skupina D pomnilnih celic, ki so vezane vzporedno in od katerih vsaka dobi na svojem vhodu D podatkovni bit, ki naj ga shrani, vse celice pa so krmiljene s skupnim urinim signalom, ki v bistvu predstavlja signal za vpis. Tako se v n celic istočasno vpiše n-bitni podatek in ostane nespremenjen do naslednjega vpisa. Shema predstavlja **register kot pomnilniški blok**.



S **pomikalnimi (shift) registri** lahko pomikamo vsebine levo ali desno. Bistvena razlika glede na pomnilniške registre je v tem, da imamo pri pomikalnih registrih *kaskadno (zaporedno)* vezavo pomnilnih celic. Taka vezava omogoča, da se vsebina z izhoda vsake celice prenese na vhod naslednje. Ker imajo vse pomnilne celice krmiljenje izvedeno z istim urinim signalom, se pomiki v vseh celicah zgodijo istočasno. Končni rezultat vseh teh pomikov je pomik celotne vsebine (n bitov) v levo ali desno za en bit.

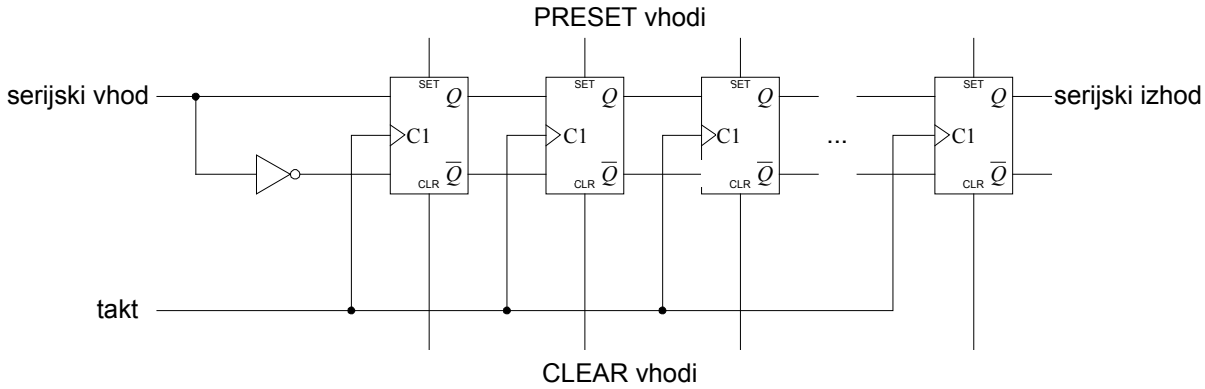


Shema prikazuje vhod v pomikalni register preko enega bita (serijski vhod) in prav tako izhod iz registra (serijski izhod).

Ob vsakem bitu, ki vstopi v register ob urinem signalu, (ki ga predstavlja v našem primeru prednja fronta), en bit tudi zapusti register. Register se tako napolni šele po n taktih urinega signala. Običajno je vsebina pomikalnega registra dostopna v celoti tudi na paralelnih izhodih (Qo, ... Qn).

Tudi vhodi so lahko paralelni, tako da pomikalni register v celoti vpišemo naenkrat. To dosežemo z dodanimi asinhronimi vhodi "clear" in "preset" na vsaki pomnilni celici.

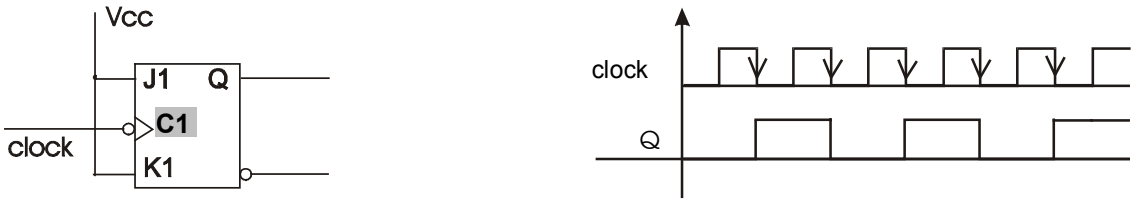
Spodnja shema prikazuje primer izvedbe pomikalnega registra z JK flip-flopi.



Asinhroni in sinhroni digitalni števci

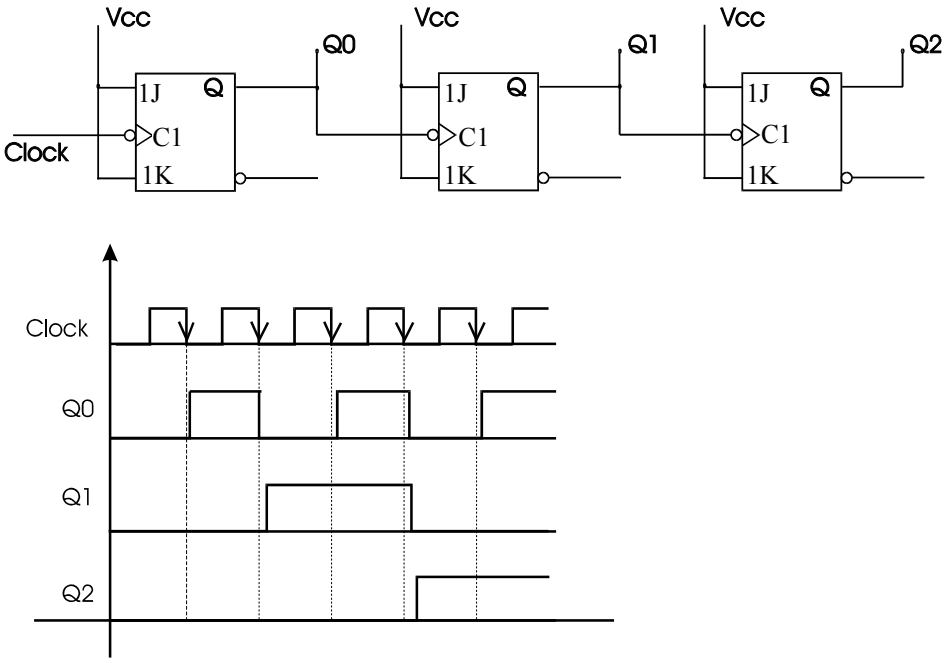
Flip-flope lahko povežemo tako, da tvorijo digitalne števce. Najbolj enostaven števec je prav gotovo binarni števec, ki predstavlja tudi osnovo za razumevanje bolj zahtevnih števcev.

Že en sam flip-flop lahko uporabimo kot **števec, ki deli z 2**. Naj bo to JK - FF. **J in K vhoda povežemo skupaj in ju priključimo na Vcc napajanje**, kar pomeni, da imamo v tem primeru opraviti s **T celico**.
Če je FF občutljiv na negativno strmino, to pomeni, da se bo stanje na izhodu FF spremenilo ob vsaki zadnji strmini "clock" impulza. **Signal "clock" predstavlja dogodke, ki jih štejemo.**



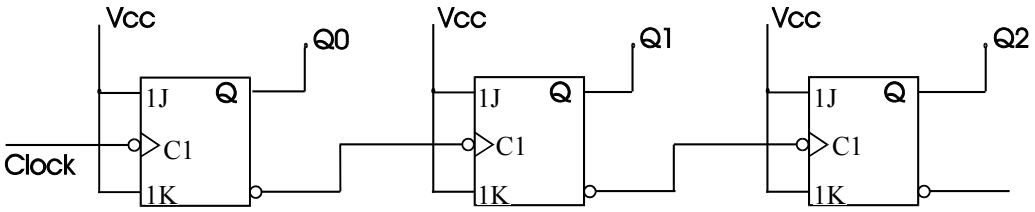
Iz slike je lahko razbrati, da je frekvenca izhodnega signala enaka polovični frekvenci vhodnega signala.

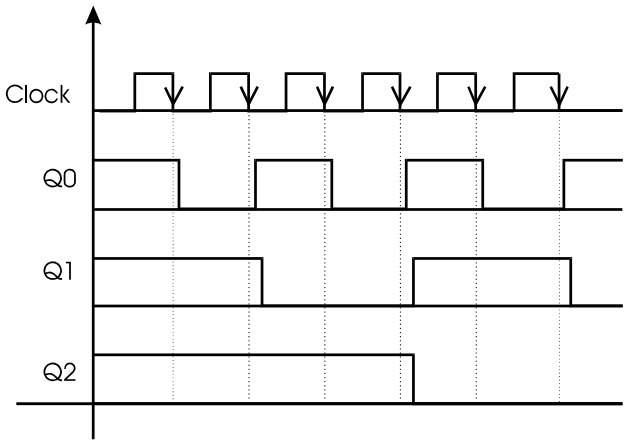
Če povežemo več takih FF zaporedno, tako da izhod predhodne stopnje vodimo v vhod naslednje, vsaka stopnja deli z 2 vhodni signal, ki ga dobi. Tri zaporedno vezane stopnje torej pomenijo deljenje signala z 8. Opraviti imamo z **asinhronim števcem**, ki šteje po **modulu 8**. S tujko mu pravimo tudi *"ripple counter".*



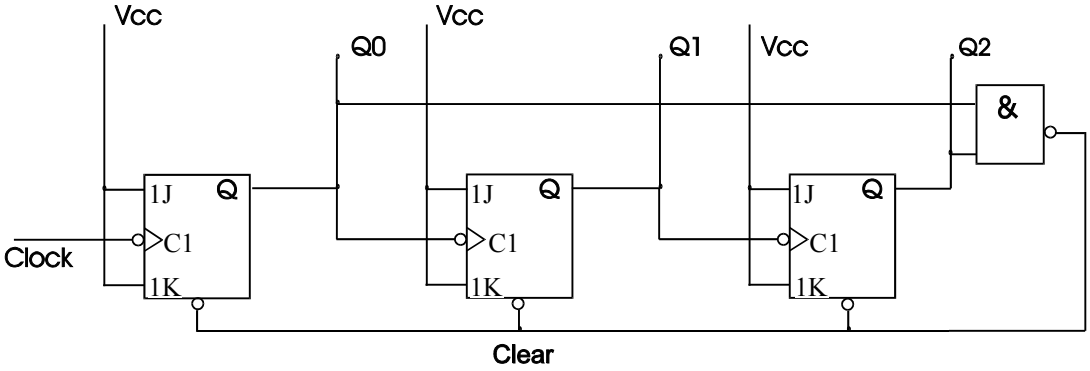
Slabost asinhronega števca je, da vsaka celica preklopi v drugem času, saj šele sprememba na izhodu predhodne celice povzroči preklop naslednje. Posledica tega je, da je **preklop vsakega naslednjega FF bolj zakasnjen**, kar vidimo tudi na sliki. Do težav pride, če stanje števca beremo v času spreminjanja, ko so se na prvih celicah že zgodile spremembe, na zadnjih pa je še staro stanje.

Števec lahko tudi šteje ***nazaj,*** če signal, ki ga vodimo iz predhodne stopnje v naslednjo, **invertiramo.**

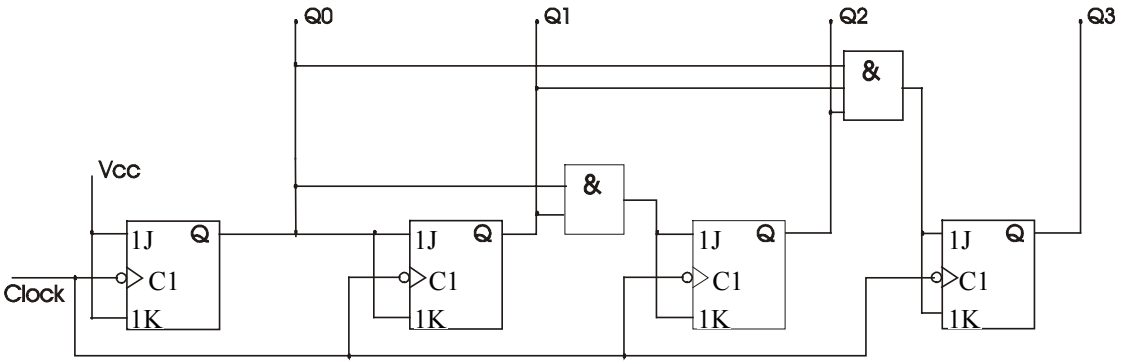


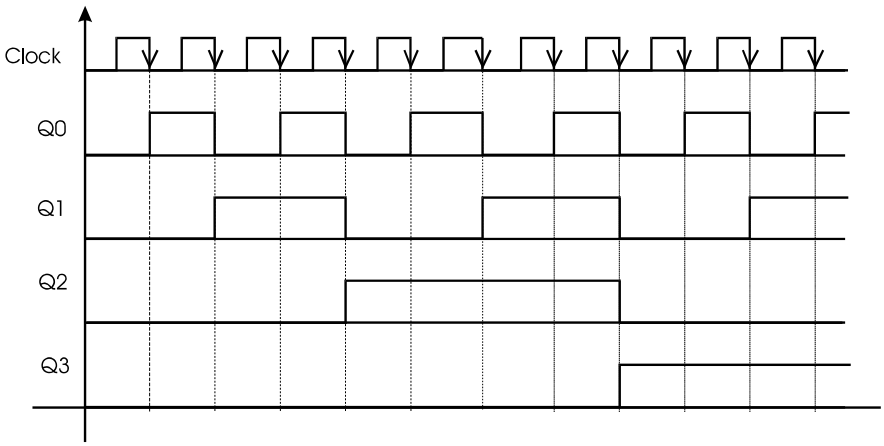


Če dodamo še kakšna NE - IN vrata, lahko izdelamo števec, ki šteje **po modulu N**. Problem rešimo tako, da poskrbimo za generiranje signala, ki bo pri vrednosti N postavil vse pomnilne celice v stanje 0, zato se bo štetje začelo znova. Tako bi namesto števca po modulu 8, kot smo ga imeli na prvi sliki, izdelali števec, ki šteje po modulu 5, takole:



Težave, ki jih povzroča asinhronski števec, lahko odpravimo z uvedbo **sinhronega štetja**. Števec, ki ga dobimo, je **sinhronski števec.**
Bistvena lastnost tega števca je, da vse celice **spremenijo stanje hkrati**, ob aktivni strmini taktnega impulza. To zahteva tudi malce drugačno medsebojno povezavo celic. Z IN vrati, ki jih dodamo v vezje, poskrbimo, da se na vsaki naslednji stopnji pojavi preklop izhoda **šele takrat**, kot je določeno z zaporedjem štetja oz. z binarno kodo. Tretja celica mora preklopiti, ko sta preklopljeni že prva in druga celica, torej šele ob četrtem impulzu. Četrta celica pa preklopi po sedmem oziroma ob osmem impulzu. Predstavljeni števec bo štel do 16, ker ima 4 pomnilne celice vezane zaporedno.





Tudi te vrste števcev lahko povežemo tako, da štejejo po poljubnem modulu.

Digitalni števci se zelo veliko uporabljajo, npr. v digitalnih urah, merilnikih frekvence, računalnikih, digitalnih voltmetrih in številnih drugih aplikacijah.

Družine logičnih vezij

Digitalna logična vezja (ali tudi **preklopna vezja**), ki opravljajo enake logične funkcije, so lahko izdelana v različnih tehnologijah, zato govorimo o več družinah logičnih vezij. Eni družini pripadajo vezja, ki so izdelana v isti tehnologiji in imajo zato enake skupne lastnosti.

Danes se največ uporabljajo različne družine **TTL** (Tranzistor-Tranzistor Logic) in **CMOS** (Complementary MOS), za katere veliko število proizvajalcev ponuja ogromno število logičnih vezij (več 1000) z različnimi funkcijami. Za cel razred hitrejša je **ECL** (Emitter-coupled Logic), ki temelji na uporabi GaAs vezij, vendar se razmeroma malo uporablja. TTL in ECL sta izdelani v bipolarni tehnologiji, CMOS pa v komplementarni MOS tehnologiji.

Nekatera **LSI** (Large Scale Integration- vezja, kjer je na majhni površini zelo veliko polprevodniških elementov), kot npr. mikroprocesorji, so izdelana tudi v **NMOS** tehnologiji, pa tudi v kombinaciji bipolarne in CMOS, kar imenujejo **BiCMOS** vezja.

Vezja iste družine in z isto funkcijo, vendar različnih proizvajalcev, so med seboj večinoma zamenljiva (kompatibilna), tako po lastnostih in razporedu nožic v ohišju, pa tudi oznake vezij se ujemajo.

Pri prebiranju tega teksta si pomagaj s tabelo, ki prikazuje lastnosti posameznih vrst vezij!!!

Tako med TTL vezji, ki imajo oznako **74 ali 54** (military izvedba), srečamo naslednje družine:

- **74** - najstarejša, ki se v novih projektih ne uporablja več;
- **74S** (Schottky)- nekajkrat hitrejša, vendar tudi z večjo porabo;
- **74LS** (Low power Schottky)- nekaj hitrejša pri zelo zmanjšani porabi;
- **74AS** (Advanced Schottky)- nadomešča 74S, vendar je hitrejša in deluje pri manjši moči;
- **74ALS** (Advanced Low power Schottky)- nadomešča 74LS, je pa tudi hitrejša in ima manjšo porabo.

Primeri oznak vezij so: 7400 za dvovhodna NAND vrata, 74LS374 za 8-bitni latch, itd.

Za vse te družine je značilno, da uporabljajo ***iste napajalne napetosti (5V) in imajo enake napetostne nivoje***, zato jih med seboj lahko poljubno kombiniramo (glej tabelo o tipičnih podatkih!).

Tudi znotraj CMOS se nadaljuje razvoj v smeri povečanja hitrosti in zmanjšanja porabe. Povedali smo že, da je prednost CMOS tehnologije pred bipolarno v tem, *da* ***vezje v mirovnem stanju praktično nima skoraj nobene porabe moči***. Z naraščanjem števila preklopov se poraba veča, dokler ne doseže ali celo preseže porabe bipolarnih vezij.

- najstarejša serija z oznako **40xx**, ki ima napajalno napetost od 3V do 15V;
- **74C**- po funkciji in nožicah kompatibilna s TTL družino;
- **74HC** (High speed CMOS)- z napajalno napetostjo od 2V do 6V;
- **74AC** (Advanced CMOS)- z večjim izhodnim tokom;
- **74ACT in 74HCT**, ki sta po logičnih nivojih kompatibilni s TTL družino in zato lahko vgrajujemo vezja skupaj s TTL vezji brez vmesnih prilagoditev.

Pri logičnih vezjih moramo biti predvsem pozorni na:

- **zakasnitev vezja - tp** (propagation delay), to je čas, ki ga vezje potrebuje, da se na izhodu odzove na spremembo signala na vhodu vezja;
- **maksimalno frekvenco spreminjanja vhodnih signalov - fmax**, pri kateri vezje še lahko preklaplja;
- **maksimalno in minimalno napajalno napetost vezja**, pri kateri vezje še pravilno deluje;
- **izhodno obremenljivost vezja - fan out**, ki pove, koliko vezij iste družine lahko poganja izhod vezja;
- upoštevati moramo **izhodni tok** v nizkem izhodnem logičnem nivoju, ki se lahko bistveno razlikuje od izhodnega toka v visokem izhodnem logičnem nivoju;
- **izhodne napetostne nivoje Vol in Voh ter vhodne napetostne nivoje Vil in Vih**; tako mora biti v splošnem nizek izhodni napetostni nivo čim nižji in čim bliže absolutni napetosti 0 V, visok izhodni nivo pa čim višji in čim bliže Vcc. Pri vhodnih nivojih pa naj bo prag za logično ničlo čim višji, za logično 1 pa čim nižji. Razlika med vhodnim in izhodnim nivojem ***povečuje odpornost vezja na motilne signale - odpornost proti šumu***. Različni vhodni nivoji pri različnih družinah pa onemogočajo neposredno povezovanje vezij različnih družin- to je možno samo s prilagoditvijo vhodov.

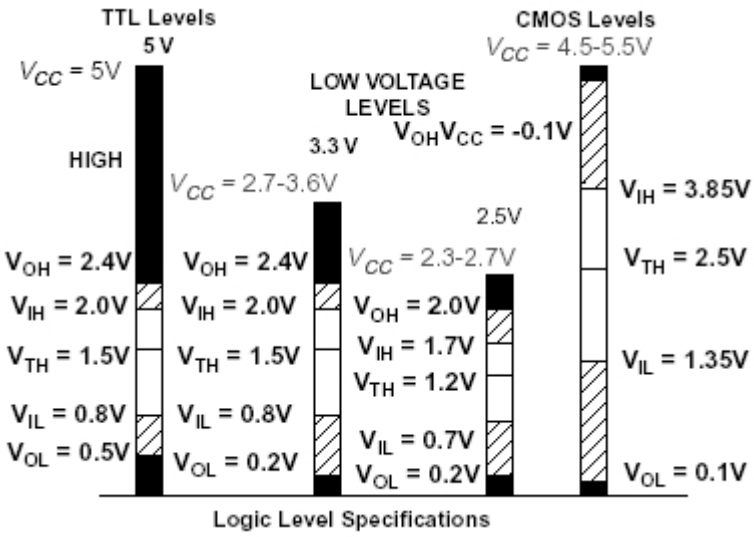
CMOS vezja se odlikujejo po:

- manjši dovzetnosti za motnje
- manjši porabi
- napajanje je po potrebi lahko nižje (npr. pri procesorjih 3,3 V ali celo 1 V zaradi manjše porabe in manjše nevarnosti za preboj) ali pa višje (npr. 15 V), če želimo večjo odpornost na motnje
- imajo boljše izhodne napetostne nivoje
- ima boljše vhodne napetostne nivoje

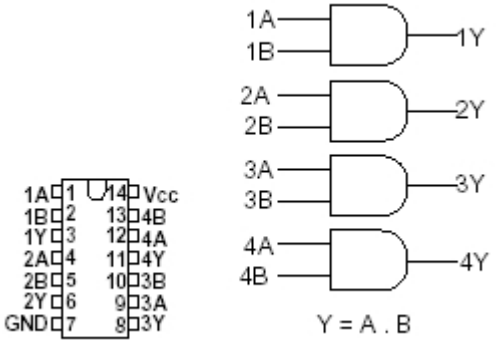
Pri povezovanju logičnih vezij **upoštevamo še**:

- neuporabljene vhode vežemo na visok ali nizek logični nivo (GND ali Vcc), da vhodi niso "plavajoči";
- neuporabljene **izhode** pustimo "v zraku", torej **neprikljućene**, da ne prihaja do "kratkih stikov";
- vezje mora najprej dobiti napajalno napetost, šele nato zunanje vhodne signale;
- ker predstavljajo povezave med vezji induktivnosti L (kar ima za posledico pojav inducirane napetosti $Ui=L^{\bullet}\Delta i/\Delta t$), zmanjšamo njihov vpliv z dodajanjem blokirnih kondenzatorjev 0.01μF na napajalne linije ob vezja (med GND in Vcc).

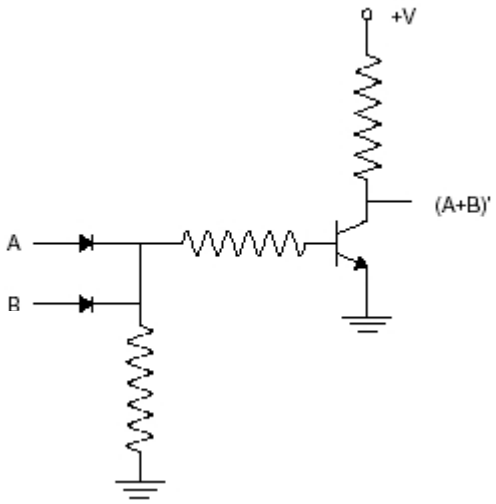
Primerjava vhodnih in izhodnih logičnih nivojev digitalnih vezij različnih družin. Slika nazorno kaže, da morajo biti izhodni napetostni nivoji za določeno logično stanje "boljši" od vhodnih. Tako se med izhodom vezja, ki daje signal in vhodom vezja, ki signal sprejema lahko doda še motnja (signal se še dodatno poslabša), pa bo še vedno interpretacija logičnih nivojev pri sprejemu pravilna.

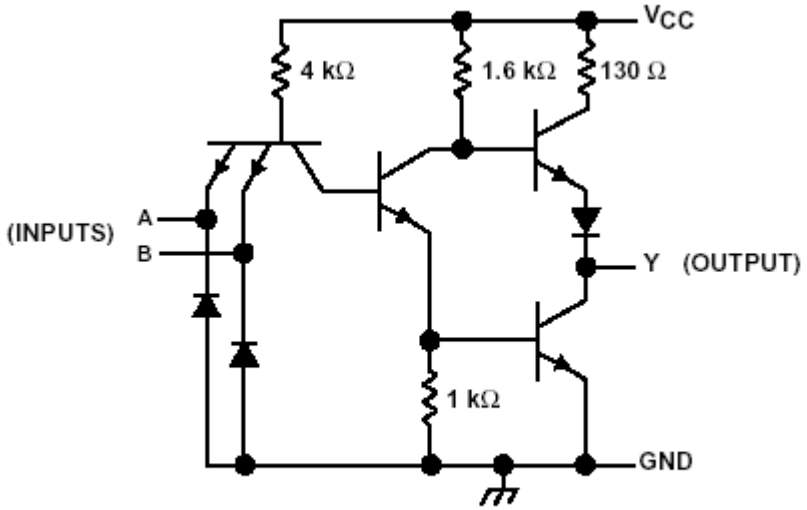


Primer diskretnega digitalnega integriranega vezja s štirimi dvovhodnimi logičnimi IN vrati v enem ohišju



Primer logičnih vrat izdelanih v diodno tranzistorski logiki DTL, ki pa se ne uporablja več





Transistor–transistor logic implementation of a NAND gate [4].

Primer logičnih NE-IN vrat izdelanih s TTL tehnologijo