UNIVERZA V LJUBLJANI
FAKULTETA ZA ELEKTROTEHNIKO

Zbirka rešenih nalog pri predmetu
RAZVOJ DIGITALNIH SISTEMOV

MATEJ MOŽEK

Ljubljana, 28. avgust 2019

Ta stran je namenjena katalogizaciji CIP.

Predgovor

Zbirka rešenih nalog je v prvi vrsti namenjena študentom elektrotehnike smeri Elektronika na univerzitetnem in visokošolskem strokovnem študiju. Naloge v zbirki ilustrirajo snov, ki jih pokrivata predmet Razvoj digitalnih sistemov, ki se predava v drugem letniku na Fakulteti za elektrotehniko, smer Aplikativna elektronika.
Predmet podaja znanja s področja sodobnega digitalnega načrtovanja na osnovi HDL jezikov (poudarek na VHDL) in funkcionalnih simulatorjev (Logisim). Cilj predmeta je samostojno načrtovanje digitalnih vezij s klasičnimi TTL vezji s sodobnimi načrtovalskimi orodji.

Izbor in organizacija tvarine v knjigi sta skrbno premišljena saj izhajata iz dolgoletnih izkušenj, ki sem jih pridobil med vodenjem vaj s področja načrtovanja digitalnih struktur. Na začetku vsakega poglavja je dodan kratek uvod, ki razlaga in povzema postopke za reševanje nalog. Rešitve nalog so opremljene z razlago, ki omogoča enostavno razumevanje poteka reševanja.

Kljub vsej pazljivosti pri pisanju se v zbirki zagotovo nahajajo neodkrite napake, za katere se že vnaprej opravičujem, obenem bom vesel vsake povratne informacije za izboljšanje prihodnjih izdaj.

avtor

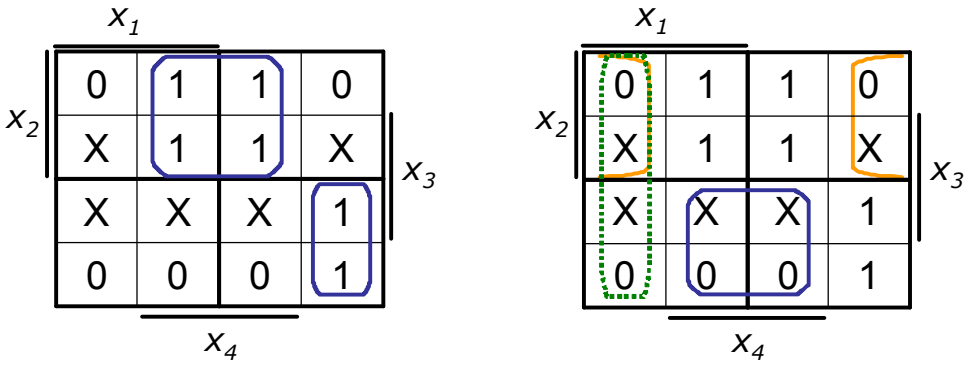
Matej Možek

1. Določite minimalno normalno obliko (MNO) za logično funkcijo f z redundantnimi vhodnimi kombinacijami.

$$f^4 = \sum(0,1,6,7,14,15) \text{ in } \sum_X(3,5,9,11,13)$$

Funkcijo v PDNO z redundancami moramo izpisati v Veitch−ev diagram, od koder bomo lahko izvajali postopek minimizacije.

$$f^4 = \sum(0,1,6,7,14,15) \text{ in } \sum_X(3,5,9,11,13)$$



Prikazana sta dva Veitch−eva diagrama. Levega uporabljamo za tvorbo MDNO, saj prikazuje funkcijo f, desnega za tvorbo MKNO, ker združujemo minterme negirane funkcije.

MDNO:

$$f_{MDNO} = x_2 \cdot x_4 + \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_4}$$

Za MKNO pa združujemo ničle (izražamo negirano funkcijo) in izrazimo minimalno obliko negirane funkcije. Nato uporabimo De Morgan−ov teorem, da pridemo do konjunktivne izražave.

$$\overline{f} = x_1 \cdot \overline{x_4} + \overline{x_2} \cdot x_4 + x_2 \cdot \overline{x_4}$$
$$f = \overline{x_1 \cdot \overline{x_4} + \overline{x_2} \cdot x_4 + x_2 \cdot \overline{x_4}}$$
$$f = \overline{x_1 + \overline{x_4}} \cdot \overline{\overline{x_2} + x_4} \cdot \overline{x_2 + \overline{x_4}}$$
$$f_{MKNO} = \left(\overline{x_1} + x_4\right) \cdot \left(x_2 + \overline{x_4}\right) \cdot \left(\overline{x_2} + x_4\right)$$

Za minimalno normalno obliko moramo določiti, katera izvedba funkcije je cenejša (manjši COST vezja).

OBLIKA	VRAT	VHODOV	COST
MDNO	3	7	10
MKNO	4	9	13

Cenejša izvedba je MDNO, saj je strošek vezja manjši (COST), zato je MNO=MDNO.

2. Z uporabo pravil Boole−ove logike zapišite logično funkcijo v KNO z NAND (Sheffer−jevimi) operatorji.

$$f(x_1, x_2, x_3) = (x_1 + x_2) \cdot (\overline{x_2} + \overline{x_3}) \cdot x_3$$

Direktne poti za pretvorbo KNO v zapis s NAND (Sheffer−jevimi) operatorji ni, zato KNO najprej pretvorimo v DNO obliko, tako da izvedemo AND operacije nad dvočleniki:

$$f(x_1, x_2, x_3) = (x_1 + x_2) \cdot (\overline{x_2} + \overline{x_3}) \cdot x_3$$

V drugem členu bomo dobili člen $x \cdot x' = 0$ po lastnostih Boole−ove algebre:

$$f(x_1, x_2, x_3) = (x_1 \cdot \overline{x_2} + x_1 \cdot \overline{x_3} + x_2 \cdot \overline{x_3}) \cdot x_3$$

S preostalimi členi izpišemo AND operacije:

$$f(x_1, x_2, x_3) = x_1 \cdot \overline{x_2} \cdot x_3 + x_1 \cdot \overline{x_3} \cdot x_3 + x_2 \cdot \overline{x_3} \cdot x_3$$

Podobno pri drugem in tretjem členu funkcije dobimo $x \cdot x' = 0$:

$$f(x_1, x_2, x_3) = x_1 \cdot \overline{x_2} \cdot x_3$$

Nastala funkcija ima samo en člen, ki ga moramo izraziti z NAND operatorji. Funkcijo negiramo in dobimo:

$$\overline{f(x_1, x_2, x_3)} = \overline{x_1 \cdot \overline{x_2} \cdot x_3}$$

Končno izrazimo še negacijo funkcije *f*:

$$\overline{x} = 1 \uparrow x = x \uparrow x$$

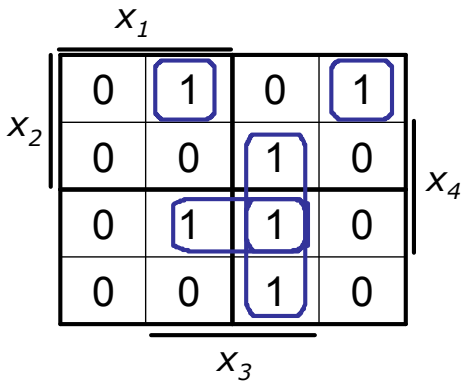
Nazadnje izrazimo original funkcije *f*, kot zahteva naloga:

$$f(x_1, x_2, x_3) = 1 \uparrow \left(x_1 \uparrow \overline{x_2} \uparrow x_3 \right)$$

3. Zgradite vezje, katerega izhod postane '1', ko se na vhodu pojavi eno izmed praštevil (2, 3, 5, 7, 11, 13). Števila na vhodu so kodirana s 4−bitno Gray−evo kodo. Za realizacijo uporabite samo NAND vrata. Kolikšna je COST funkcija realiziranega vezja?

S 4−bitno Grayevo kodo lahko kodiramo 16 števil (0 - 15). Najprej zapišemo pravilnostno tabelo v kateri desetiška števila kodiramo v Grayevi kodi (npr. $7_{10}=0100_{Gray}$). Izhod mora biti '1' vsakič, ko se na vhodu pojavi praštevilo. Izpisano funkcijo nato izpišemo v Veitch−ev diagram, od koder bomo lahko izvajali postopek minimizacije v PDNO, iz te oblike pa bomo prešli na PSNO (NAND) operatorji.

št.	x_1	x_2	x_3	x_4	f
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	1	1
3	0	0	1	0	1
4	0	1	1	0	0
5	0	1	1	1	1
6	0	1	0	1	0
7	0	1	0	0	1
8	1	1	0	0	0
9	1	1	0	1	0
10	1	1	1	1	0
11	1	1	1	0	1
12	1	0	1	0	0
13	1	0	1	1	1
14	1	0	0	1	0
15	1	0	0	0	0



$$f_{MDNO} = \overline{x_1} \cdot \overline{x_2} \cdot x_3 + \overline{x_2} \cdot x_3 \cdot x_4 + \overline{x_1} \cdot x_3 \cdot x_4 + x_1 \cdot x_2 \cdot x_3 \cdot \overline{x_4} + \overline{x_1} \cdot x_2 \cdot \overline{x_3} \cdot \overline{x_4}$$

Da bi iz PDNO prešli na realizacijo s samimi NAND vrati, funkcijo v PDNO dvakrat negiramo in uporabimo de Morgan−ov teorem:

$$\overline{\overline{f_{MDNO}}} = \overline{\overline{\overline{x_1} \cdot \overline{x_2} \cdot x_3 + \overline{x_2} \cdot x_3 \cdot x_4 + \overline{x_1} \cdot x_3 \cdot x_4 + x_1 \cdot x_2 \cdot x_3 \cdot \overline{x_4} + \overline{x_1} \cdot x_2 \cdot \overline{x_3} \cdot \overline{x_4}}}$$

$$f_{PSNO} = \overline{\left(\overline{x_1} \cdot \overline{x_2} \cdot x_3\right) \cdot \left(\overline{x_2} \cdot x_3 \cdot x_4\right) \cdot \left(\overline{x_1} \cdot x_3 \cdot x_4\right) \cdot \left(x_1 \cdot x_2 \cdot x_3 \cdot \overline{x_4}\right) \cdot \left(\overline{x_1} \cdot x_2 \cdot \overline{x_3} \cdot \overline{x_4}\right)}$$

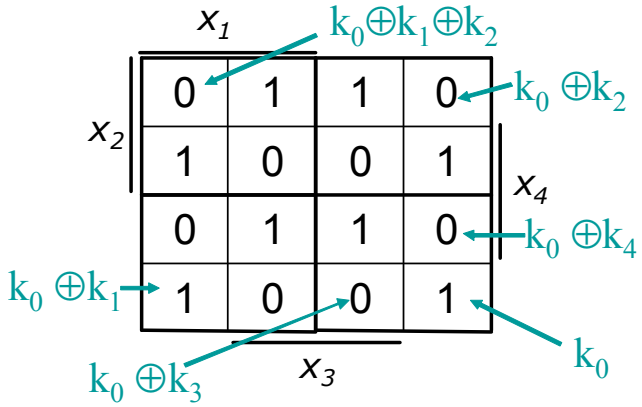
Za nastalo PSNO moramo določiti COST vezja (brez inverterjev):

OBLIKA	**VRAT**	**VHODOV**	**COST**
PSNO	6	22	28

4. Z uporabo pravil Boole−ove logike ali Veitch−evih diagramov pokažite, da je logična funkcija linearna. Izračunajte koeficiente linearnosti in jo izrazite v obliki linearnega polinoma.

$$f(x_1,x_2,x_3,x_4)=\overline{x_2}\cdot x_3\cdot x_4+x_2\cdot\overline{x_3}\cdot x_4+x_2\cdot x_3\cdot\overline{x_4}+\overline{x_2}\cdot\overline{x_3}\cdot\overline{x_4}$$

Disjunktivno normalno obliko logične funkcije vpišemo v Veitchev diagram, tako da za vsak konjunktivni izraz poiščemo ustrezne pravokotnike in vanje vpišemo funkcijske vrednosti 1. Prepognemo kvadrat z mintermom m_0 proti x_4 in nato oba skupaj proti spremenljivki x_3. Rezultat prepogiba je negacija vrednosti, kar izpolnjuje pogoj linearnosti, zato prepognemo štiri kvadrate proti spremenljivki x_2. Rezultat prepogiba je negacija v vseh štirih kvadratih, zato nadaljujemo z zadnjim prepogibom osmih kvadratov proti x_1, kjer imamo enakost. Pregledali smo cel Veitchev diagram in ker smo povsod dobili izpolnjen pogoj enakosti ali popolne negacije pomeni, da je logična funkcija linearna. Prepogibanje je prikazano v knjigi, stran 79, vaja 6.5.5.



Podana funkcija je funkcija 4 spremenljivk, zato lahko njeno splošno izražavo kot linearno funkcijo pišemo kot:

$$f(x_1,x_2,x_3,x_4)=k_0\oplus k_1x_1\oplus k_2x_2\oplus k_3x_3\oplus k_4x_4$$

S pomočjo Veitch−evega diagrama izračunamo koeficiente.
Iz enačb sledi: $k_0=1$ in $k_0\oplus k_4=0$, kar pomeni $1\oplus k_4=0\rightarrow k_4=1$.
Iz enačbe $k_0\oplus k_3=0$, kar pomeni $1\oplus k_3=0\rightarrow k_3=1$.
Če napišemo še enačbo za $k_0\oplus k_2=0$, kar pomeni $0\oplus k_2=0$ sledi da je $k_2=1$.
Iz enačbe $k_0\oplus k_1=1$, kar pomeni $1\oplus k_1=0\rightarrow k_1=0$.
In končna rešitev:

$$f(x_1,x_2,x_3,x_4)=1\oplus x_2\oplus x_3\oplus x_4$$

Do istega pridemo lahko z uporabo pravil Boole−ove logike: Izhodišče je podana funkcija.

$$f(x_1,x_2,x_3,x_4)=\overline{x_2}\cdot x_3\cdot x_4+x_2\cdot\overline{x_3}\cdot x_4+x_2\cdot x_3\cdot\overline{x_4}+\overline{x_2}\cdot\overline{x_3}\cdot\overline{x_4}$$

Izvedemo razčlenitev po x_4:

$$f(x_1,x_2,x_3,x_4)=\left(\overline{x_2}\cdot x_3+x_2\cdot\overline{x_3}\right)\cdot x_4+\left(x_2\cdot x_3+\overline{x_2}\cdot\overline{x_3}\right)\cdot\overline{x_4}$$

Člena v oklepajih predstavljata XOR operacijo in pa EQU operacijo. Za dve spremenljivki velja da je XOR negacija ekvivalence.

$$x \oplus y = \bar{x} \cdot y + x \cdot \bar{y}$$
$$x \equiv y = \overline{x \oplus y} = \bar{x} \cdot \bar{y} + x \cdot y$$

Zgornji enakosti vpišemo v enačbo za funkcijo f, s tem da ekvivalenco raje pišemo kot negacijo XOR funkcije.

$$f(x_1, x_2, x_3, x_4) = (x_2 \oplus x_3) \cdot x_4 + \overline{(x_2 \oplus x_3)} \cdot \overline{x_4}$$

Uvedemo novo spremenljivko g:

$$g = x_2 \oplus x_3$$

Funkcijo zapišemo kot:

$$f(x_1, x_2, x_3, x_4) = g \cdot x_4 + \bar{g} \cdot \overline{x_4}$$

Od koder sledi

$$f(x_1, x_2, x_3, x_4) = \overline{g \oplus x_4}$$

Iz zgornje enačbe sledi, da imamo negacijo XOR funkcije, kar lahko zopet izrazimo z XOR funkcijo kot:

$$\bar{x} = 1 \oplus x$$

Če z obratnim vstavljanjem izrazimo funkcijo f s spremenljivkami $x_1..x_4$ dobimo končen rezultat:

$$f(x_1, x_2, x_3, x_4) = 1 \oplus x_2 \oplus x_3 \oplus x_4$$

Od tod bi podobno lahko prebrali koeficiente:

$$f(x_1, x_2, x_3, x_4) = k_0 \oplus k_1 x_1 \oplus k_2 x_2 \oplus k_3 x_3 \oplus k_4 x_4$$
$$f(x_1, x_2, x_3, x_4) = 1 \oplus 0 \cdot x_1 \oplus 1 \cdot x_2 \oplus 1 \cdot x_3 \oplus 1 \cdot x_4$$

5. Realizirajte funkcijo podano funkcijo *f* s čim manj izbiralniki 4/1.

$$f(a,b,c,d) = \left(a \cdot \bar{b} + b \cdot c \cdot \bar{d} + b \cdot c\right) \cdot \left((a \cdot c \cdot d) \cdot \left(\bar{c} + d\right)\right)$$

Funkcija *f* je podana v večnivojski (nenormalni) obliki:

$$f(a,b,c,d) = \left(a \cdot \bar{b} + b \cdot c \cdot \bar{d} + b \cdot c\right) \cdot \left((a \cdot c \cdot d) \cdot \left(\bar{c} + d\right)\right)$$

zato jo najprej poenostavimo z uporabo pravil Boole−ove logike. Izpišemo desni člen funkcije in uporabimo lastnost Boole−ove logike $x \cdot \bar{x} = 0$, lastnost $x \cdot x = x$ in lastnost $0 + x = x$.

$$f(a,b,c,d) = \left(a \cdot \bar{b} + b \cdot c \cdot \bar{d} + b \cdot c\right) \cdot \left(a \cdot c \cdot d \cdot \bar{c} + a \cdot c \cdot d \cdot d\right)$$

Nad rezultatom ponovno uporabimo lastnost Boole−ove logike $x \cdot \bar{x} = 0$ in lastnost $x \cdot x = x$.

$$f(a,b,c,d) = \left(a \cdot \bar{b} + b \cdot c \cdot \bar{d} + b \cdot c\right) \cdot (a \cdot c \cdot d)$$

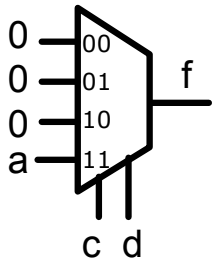
Rezultat vnesemo v levi del funkcije in znova uporabimo omenjene lastnosti Boole−ove logike:

$$f(a,b,c,d) = \left(a \cdot \bar{b} \cdot a \cdot c \cdot d + b \cdot c \cdot \bar{d} \cdot a \cdot c \cdot d + b \cdot c \cdot a \cdot c \cdot d\right)$$

Dobimo dva člena in ju zapišemo v obliki PDNO, ki jo nato minimiziramo s pomočjo Veitch−evega diagrama ali z uporabo lastnosti združevanja Boole−ove algebre $x + \bar{x} = 1$:

$$f(a,b,c,d) = a \cdot \bar{b} \cdot c \cdot d + b \cdot a \cdot c \cdot d$$
$$f_{PDNO}(a,b,c,d) = V(11,15)$$
$$f_{MDNO}(a,b,c,d) = a \cdot c \cdot d \cdot \left(\bar{b} + b\right) = a \cdot c \cdot d$$

in jo realiziramo z enim izbiralnikom 4/1, tako da naredimo Shannon−ov razvoj funkcije. Glede na kombinacijo naslovnih vhodov izbiralnika dobimo 6 možnih rešitev (ac, ca, ad, da, cd, dc).



6. Realizirajte funkcijo podano funkcijo *f* s čim manj izbiralniki 4/1.

$$f(a,b,c,d)=\left(a\cdot\bar{b}+(b\equiv c)\right)\oplus\left(\left(a\uparrow c\uparrow d\right)\downarrow(c\cdot d)\right)$$

Funkcijo najprej razdelimo na dva dela, pri vmesnem operatorju XOR:

$$g(a,b,c,d)=\left(\left(a\uparrow c\uparrow d\right)\downarrow(c\cdot d)\right)$$
$$h(a,b,c,d)=\left(a\cdot\bar{b}+(b\equiv c)\right)$$
$$f(a,b,c,d)=g\oplus h$$

Najprej analiziramo funkcijo *g* tako da izpišemo Piercev in Sheffer–jev operator.

$$g(a,b,c,d)=\left(\left(a\uparrow c\uparrow d\right)\downarrow(c\cdot d)\right)$$
$$g(a,b,c,d)=\overline{\overline{a\cdot c\cdot d}+c\cdot d}$$
$$g(a,b,c,d)=a\cdot(c\cdot d)\cdot\left(\overline{c\cdot d}\right)$$

V zadnjem členu funkcije *g* vidimo, da gre za konjunkcijo negirane in originalne spremenljivke. Iz lastnosti Boole–ove algebre sledi $x\cdot\bar{x}=0$, tako da je celoten izraz *g(a,b,c,d)='0'*. V celotni funkciji *f* smo torej dobili:

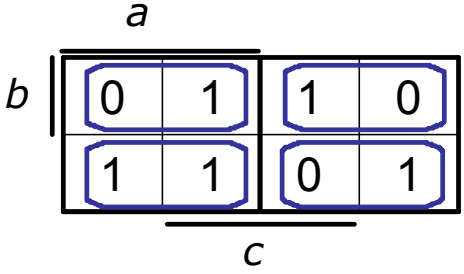
$$f(a,b,c,d)=g\oplus h=0\oplus h=h$$

Analizirajmo še funkcijo *h*, tako da izpišemo operacijo ekvivalence:

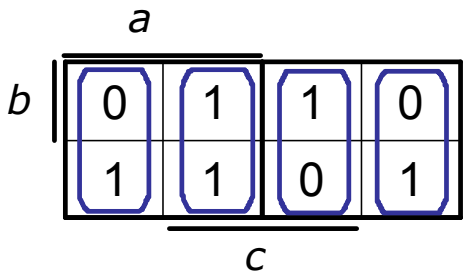
$$h(a,b,c,d)=\left(a\cdot\bar{b}+(b\equiv c)\right)$$
$$h(a,b,c,d)=a\cdot\bar{b}+b\cdot c+\bar{b}\cdot\bar{c}$$

Dobljeno funkcijo vrišemo v Veitch–ev diagram, saj bomo od tam bolj očitno videli razvoj po spremenljivkah izbiralnika 4/1. Možni razvoji so trije: *ab*, *ac* in *bc*.

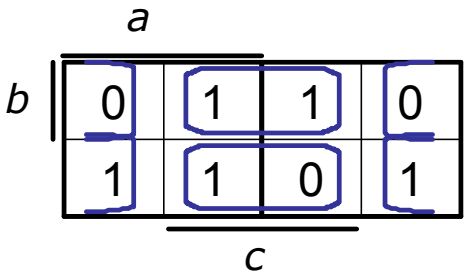
Razvoj po spremenljivkah *ab*:



Razvoj po spremenljivkah *ac*:



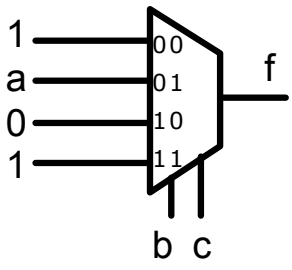
Razvoj po spremenljivkah *bc*:



V spodnji tabeli izpišemo ustrezne vrednosti podatkovnih vhodov izbiralnika glede na kombinacijo naslovov izbiralnika a_1a_2:

a_1	a_2	*ab*	*ac*	*bc*
0	0	c'	b'	1
0	1	c	b	a
1	0	1	b'	0
1	1	c	1	1

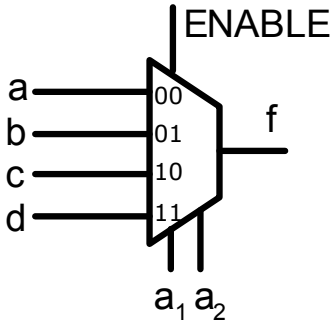
Izberemo razvoj po spremenljivkah *bc*, da bo realizacija z izbiralnikom najbolj ugodna, saj takrat ne potrebujemo nobene negacije spremenljivke, ampak samo konstanti '0' in '1'.



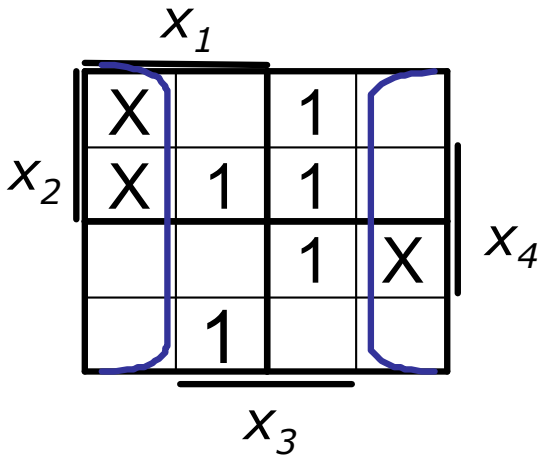
7. Realizirajte funkcijo $f^4 = V(3,6,7,10,15)$ z redundančnimi mintermi pri $V_x(1,12,13)$ s čim manj izbiralniki 4/1, ki imajo vhod "ENABLE".

Delovanje izbiralnika 4/1 z vhodom "ENABLE" povzema spodnja tabela:

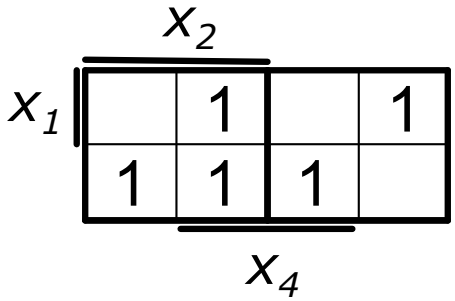
ENABLE	a_1	a_2	*f*
0	X	X	0
1	0	0	a
1	0	1	b
1	1	0	c
1	1	1	d



Funkcijo narišemo v Veitch−ev diagram, da si jo lažje predstavljamo:

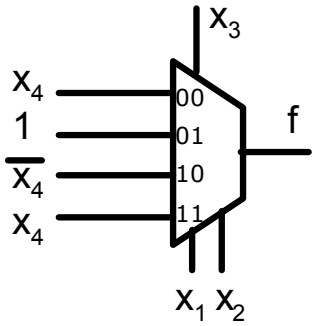


Čim imamo na voljo izbiralnik z ENABLE vhodom preverimo ali obstaja spremenljivka v osnovni ali negirani obliki, pri kateri so vsa polja enaka '0' vključno z redundancami X. V zgornjem Veitch−evem diagramu je to spremenljivka x_3: Namreč, če je x_3='0', potem lahko vse redundance izberemo tako, da bo *f='0'* za vse vrednosti x_3='0'. Ko to spremenljivko določimo, narišemo samo tisti del Veitch−evega diagrama štirih spremenljivk, pri katerem bo x_3='1'.



Nastali Veitch−ev diagram razvijemo po vseh možnih kombinacijah dveh spremenljivk in rezultat zapišemo v tabeli. Vse realizacije so enako kompliciran, tako da je vseeno katero realiziramo. Odločimo se za realizacijo po x_1x_2.

a_1	a_2	x_1x_2	x_1x_4	x_2x_4
0	0	x_4	x_2	x_1
0	1	1	1	x_1'
1	0	x_4'	x_2'	x_1'
1	1	x_4	x_2	1



8. Realizirajte funkcijo $f^4 = \&(2,5-7,9,14)$ z redundančnimi makstermi pri $\&_x(0,4,13)$ s čim manj izbiralniki 4/1.

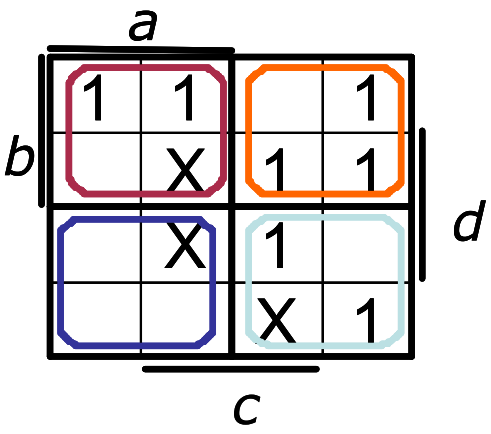
Funkcija *f* je podana v obliki PKNO z redundancami. Za potrebe realizacije jo najprej pretvorimo v obliko PDNO. To storimo tako, da maksterme preslikamo v minterme. V pravilnostno tabelo funkcije najprej zapišemo številke mintermov (m) in pripadajoče številke makstermov (M). Vpišemo *f='0'* za vse maksterme in *f='X'* za vse redundantne maksterme. Na preostala mesta vpišemo *f='1'* in preberemo pri katerih mintermih je *f='1'* oz. *f='X'* ter funkcijo izrazimo v obliki PDNO.

m	*M*	*a*	*b*	*c*	*d*	*f*
0	15	0	0	0	0	1
1	14	0	0	0	1	0
2	13	0	0	1	0	X
3	12	0	0	1	1	1
4	11	0	1	0	0	1
5	10	0	1	0	1	1
6	9	0	1	1	0	0
7	8	0	1	1	1	1
8	7	1	0	0	0	0
9	6	1	0	0	1	0
10	5	1	0	1	0	0
11	4	1	0	1	1	X
12	3	1	1	0	0	1
13	2	1	1	0	1	0
14	1	1	1	1	0	1
15	0	1	1	1	1	X

Dobimo:

$$f = V(0,3-5,7,12,14) \text{ in } V_x(2,11,15)$$

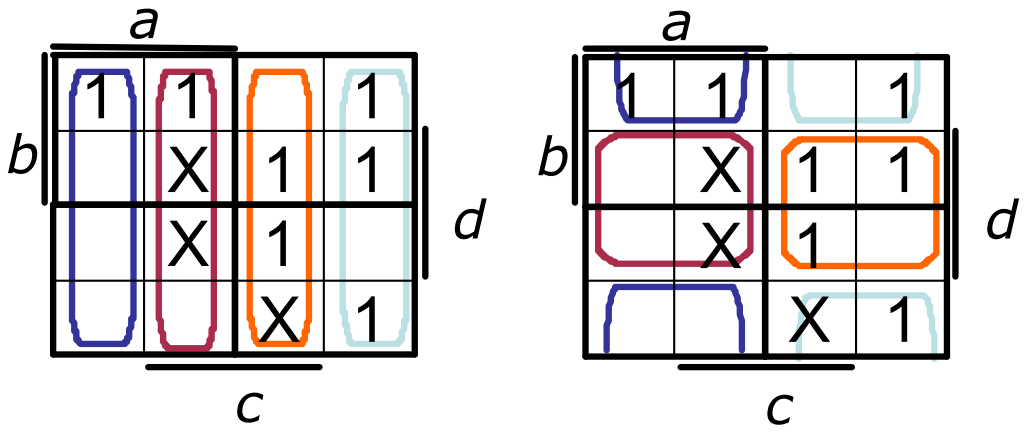
Dobljeno funkcijo vrišemo v Veitch−ev diagram. Ker iščemo najcenejšo realizacijo z izbiralnikom 4/1, bomo naredili razvoj po vseh kombinacijah naslovnih spremenljivk v Veitchev−em diagramu. Če izberemo kot naslovni spremenljivki a b, potem dobimo:



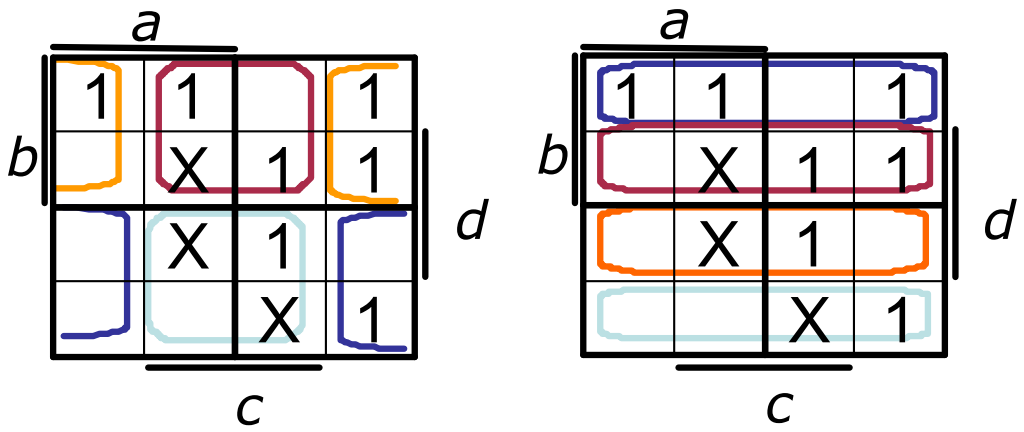
V zgornjem Veitch−evem diagramu so označena vsa štiri polja štirih mintermov, če izberemo vhodni spremenljivki a in b. Zgornji levi kvadrat (rdeč) pomeni, da bo to polje izbrano ko bosta ab="11", oranžni kvadrat ko bo ab="01", temno modri ko bo ab="10" in svetlo modri ko bo ab="00". Vsakega od teh kvadratov poskušamo opisati s čimbolj enostavno funkcijo: Vrednost zgornjega levega kvadrata opišemo s spremenljivko d', če postavimo redundanco na '0'. Vrednost spodnjega desnega kvadrata je bolj kompliciran, saj moramo vsako '1' opisati posebej: Za zgornjo '1' v tem kvadratu velja c·d, za spodnjo '1' pa c'·d'. Funkcija bo torej c·d+ c'·d', kar je enačba funkcije ekvivalence. Še bolj zoprna za realizacijo je funkcija zgornjega desnega kvadrata: d+ c'·d'. Najbolj enostavna realizacija je spodnji levi kvadrat, ki je kar '0', če postavimo redundanco na '0'. Zato, da bi pregledali še ostale možnosti, moramo narisati še preostalih pet kombinacij dveh naslovnih vhodov.

Če izberemo kot naslovni spremenljivki a in c, dobimo levi Veitchev diagram, če a in d, pa desnega. Podobno kot v prejšnjem primeru poiščemo realizacije

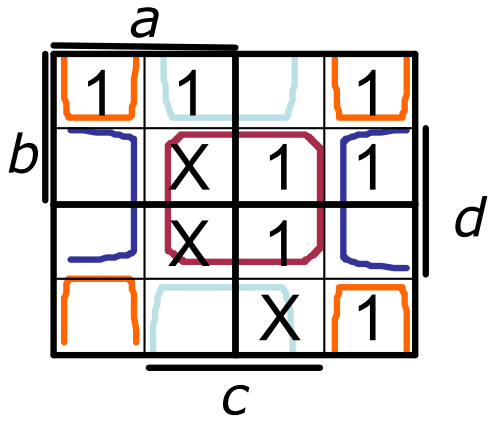
ustreznih kvadratov in iščemo najcenejšo realizacijo: Izogibamo se veliko različnim funkcijam in iščemo inačice kvadratov, ki vsebujejo same '1' ali same '0'. Pri razvoju po a in c imamo pri ac="00" najneugodnejšo funkcijo, saj vsebuje tri '1'; b+b'·d', medtem ko je razvoju po a in d imamo pri ad="01" najneugodnejšo funkcijo, saj vsebuje tri '1'; a+a'·d'.

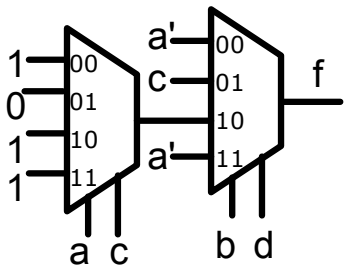


Nato izberemo naslovni spremenljivki b in c, (levi Veitchev diagram) in b in d (desni diagram). Pri razvoju po b in c imamo pri bc="10" najneugodnejšo funkcijo (oranžen), saj vsebuje tri '1'; a'+a·d', medtem ko je razvoju po b in d imamo pri bd="10" (temno moder) najneugodnejšo funkcijo, saj vsebuje tri '1'; a+a'·c'.



Zadnja kombinacija naslovnih vhodov je cd. Pri tem razvoju imamo pri cd="00" (oranžen) najneugodnejšo funkcijo, saj vsebuje tri '1'; b+a'·b'. Nobena od kombinacij ni ugodna, zato izberemo realizacijo po razvoju b in d.



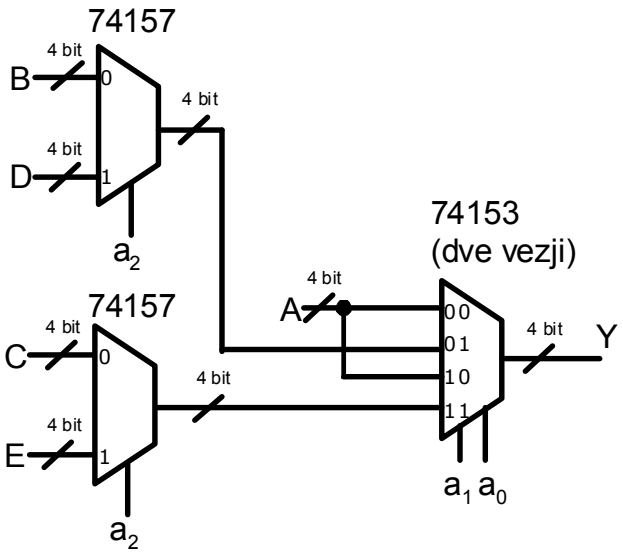


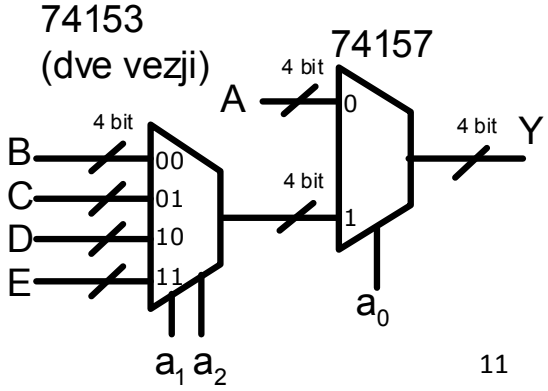
9. Realizirajte izbiralnik vodil (ang. bus multiplexer) s petimi 4-bitnimi vhodnimi vodili A, B, C, D, E. Glede na naslovni vhod $a_2a_1a_0$ je izhod y določen s tabelo:

a_2	a_1	a_0	y
0	0	0	A
0	0	1	B
0	1	0	A
0	1	1	C
1	0	0	A
1	0	1	D
1	1	0	A
1	1	1	E

Uporabite lahko največ 3 od sledečih integriranih vezij:
74153 (2*MUX 4/1, skupni naslov)
74157 (4* MUX 2/1, skupni naslov)
7400 (4*NAND)
7404 (6*NEG)
7408 (4*AND)
7432 (4*OR)

Naloga zahteva, da realiziramo izbiralnik MUX 8/1, ki je 4−biten, saj izbiramo med petimi 4-bitnimi vhodnimi vodili A, B, C, D, E. Iz tabele delovanja se vidi, da je vodilo A izbrano za vse kombinacije naslovov, pri katerih je $a_1\ a_0 = 10$ ali $a_1\ a_0 = 00$. Te kombinacije lahko izvedemo z dvema vezjema 74153; vsako od teh vezij vsebuje dva izbiralnika 4/1 in imata skupno naslovno vodilo. Naslovno vodilo $a_1\ a_0$ vežemo soležno: a_1 prvega vezja in drugega 74153 skupaj in a_0 prvega 74153 in drugega 74153 skupaj. Tako dobimo 4−bitni izbiralnik 4/1. Na drugem nivoju realiziramo 4−bitna izbiralnika 2/1, pri katerem vežemo na naslovni vhod a_2. Vsakega od teh izbiralnikov realiziramo z enim vezjem 74157, ki vsebuje štiri vezja 1−bitnih izbiralnikov 2/1. Vsi izbiralniki znotraj 74157 imajo že skupni naslov povezan, torej vežemo samo vzporedno obe vezji 74157, kot je narisano na sliki. Na eno vezje 74157 vežemo 4−bitni vhod B oz. D, na vezje 74157 vežemo 4−bitni vhod C oz. E. Vendar ta rešitev vsebuje 4 vezja, zato jo poskusimo dodatno minimizirati. Ponuja se nam tudi druga možnost izvedbe, pri kateri na prvem nivoju uporabimo en 4−bitni 2/1 izbiralnik, na drugem nivoju pa dva 4−bitna 4/1 izbiralnika. Realizacija posameznih izbiralnikov je enaka kot pri prvi rešitvi.





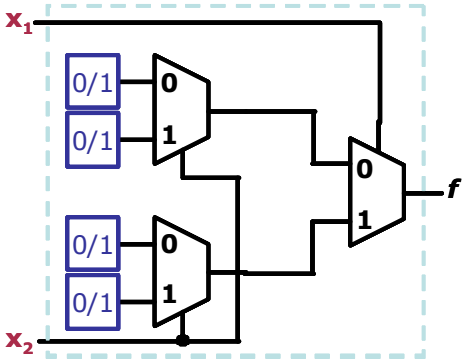
10. Realizirajte funkcijo f z dvovhodnimi vpoglednimi tabelami (ang. look−up table).

$$f(x_1,x_2,x_3,x_4)=(x_1\equiv x_3)\oplus(x_2\equiv x_4)$$

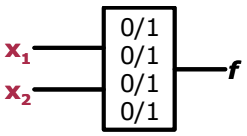
Funkcija je že primerna za realizacijo z dvovhodnimi vpoglednimi tabelami (ang. look−up table) v vezju FPGA in je ni treba posebej predelovati.

$$f(x_1,x_2,x_3,x_4)=(x_1\equiv x_3)\oplus(x_2\equiv x_4)$$

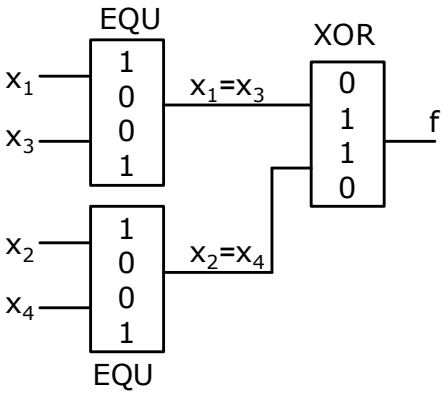
Dvovhodne vpogledne tabele (LUT2) so sestavljene iz pomnilnika (4 RAM celice) in treh 2/1 izbiralnikov. V vsako RAM celico so vpisane vrednosti, ki realizirajo eno od 16 osnovnih dvovhodnih funkcij.



x_1	x_2	AND	OR	XOR	EQU
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	1



Na zgornji sliki sta prikazani struktura dvovhodne vpogledne tabele (levo) in posplošeni simbol, ki ga uporabljamo pri risanju realizacij funkcij (desno) ter primer vsebine RAM celic za nekaj osnovnih funkcij (XOR, EQU). Tako lahko LUT uporabljamo za realizacijo funkcij v več nivojih.



11. Z uporabo Boole−ovih postulatov preoblikujte podano funkcijo v minimalni normalni obliki tako, da bo primerna za realizacijo s čim manj dvovhodnimi vpoglednimi tabelami (look−up tabelami) v vezju FPGA.

$$f(x_1,x_2,x_3,x_4,x_5,x_6,x_7) = x_1x_4x_6 + x_1x_5x_6 + x_2x_4x_6 + x_2x_5x_6 + x_3x_4x_6 + x_3x_5x_6 + x_7$$

V vseh konjunktivnih izrazih se nahaja spremenljivka x_6, tako da jo izpostavimo

$$f = x_6(x_1x_4 + x_1x_5 + x_2x_4 + x_2x_5 + x_3x_4 + x_3x_5) + x_7$$

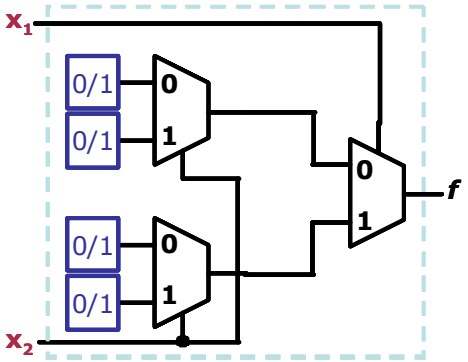
Nadalje poiščemo podobne spremenljivke, ki bi jih lahko izpostavili. Izpostavimo lahko x_4 pri treh členih in x_5 pri drugih treh členih.

$$f = x_6(x_1x_4 + x_1x_5 + x_2x_4 + x_2x_5 + x_3x_4 + x_3x_5) + x_7$$
$$f = x_6(x_4(x_1 + x_2 + x_3) + x_5(x_1 + x_2 + x_3)) + x_7$$
$$f = x_6((x_1 + x_2 + x_3)(x_4 + x_5)) + x_7$$

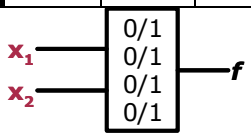
Nastaneta dva člena, ki jima je skupna disjunkcija x_1, x_2 in x_3, ki jo ponovno lahko izpostavimo. Ker želimo funkcijo realizirati z dvovhodnimi vpoglednimi tabelami moramo še trovhodno disjunkcijo razstaviti v dva nivoja.

$$f = x_6(((x_1 + x_2) + x_3)(x_4 + x_5)) + x_7$$

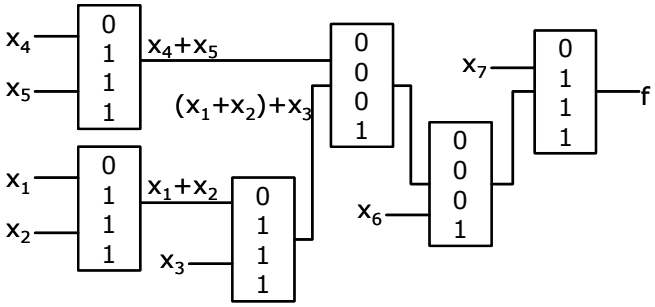
Dvovhodne vpogledne tabele (LUT2) so sestavljene iz pomnilnika (4 RAM celice) in treh 2/1 izbiralnikov. V vsako RAM celico so vpisane vrednosti, ki realizirajo eno od 16 osnovnih dvovhodnih funkcij.



x_1	x_2	AND	OR	XOR	EQU
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	1



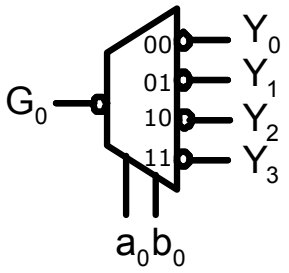
Na zgornji sliki sta prikazani struktura dvovhodne vpogledne tabele (levo) in posplošeni simbol, ki ga uporabljamo pri risanju realizacij funkcij (desno) ter primer vsebine RAM celic za nekaj osnovnih funkcij (AND, OR, XOR, EQU). Tako lahko LUT uporabljamo za realizacijo funkcij v več nivojih.



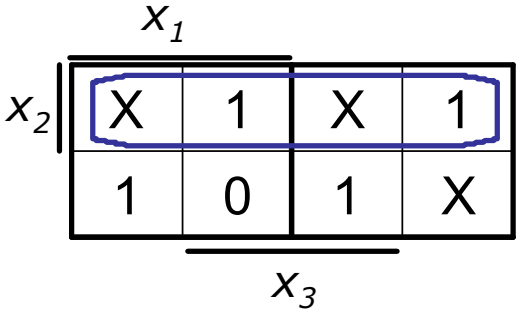
12. Realizirajte funkcijo $f^A = V(1, 2, 4, 7)$ z redundančnimi mintermi pri $V_x(0, 3, 6)$ z enim TTL dekoderjem 74139. TTL dekoder 74139 je dvojni 2/4 dekoder z vhodom za omogočenje elementa (G) in izhodi Y_0, Y_1, Y_2, Y_3 v negativni logiki.

Delovanje dekoderja 74139[1] povzema spodnja tabela:

G_0	a_0	b_0	Y_0	Y_1	Y_2	Y_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

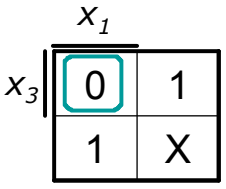


Funkcijo f narišemo v Veitch−ev diagram, da si jo lažje predstavljamo:

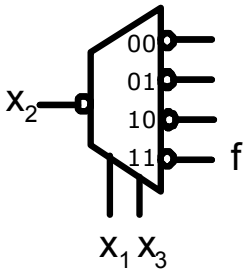


Čim imamo na voljo dekodirnik z ENABLE vhodom v negativni logiki preverimo ali obstaja spremenljivka v osnovni ali negirani obliki, pri kateri so vsa polja enaka '1' vključno z redundancami X.

V zgornjem Veitch−evem diagramu je to spremenljivka x_2: Namreč, če je x_2='1', potem lahko vse redundance izberemo tako, da bo f='1' za vse vrednosti x_2='1'. Ko določimo spremenljivko za omogočenje elementa (G), opazujemo samo preostali del Veitch−evega diagrama. Spodnje 4 vrednosti diagrama narišemo v novem Veitch−evem diagramu 2 spremenljivk.



Dekoder ima aktivno nizke izhode, zato iz nastalega diagrama realiziramo *negacijo* funkcije f zato v Veitch−evem diagramu združujemo ničle, kar nastopa samo v primeru ko sta x_1='1' in x_3='1'.

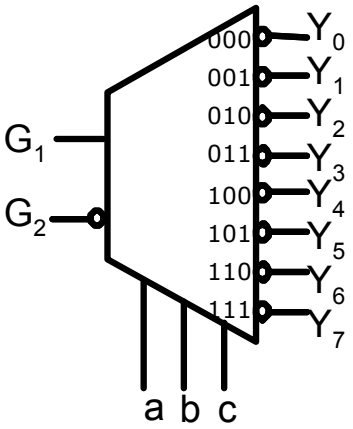


[1]http://www.alldatasheet.com/view.jsp?Searchword=74HC139

13. Realizirajte funkcijo $f^3 = V(0, 2, 3, 5, 6)$ z dekoderjem 74138 in čim manj dvovhodnimi vrati. Dekoder 74138 je 3/8 dekoder z naslovnimi vhodi *a*, *b*, *c* in aktivno nizkimi izhodi Y_0, Y_1, Y_2, Y_3,Y_4, Y_5, Y_6, Y_7 v negativni logiki. Element ima vhod za omogočenje (*G1*) v pozitivni logiki in dva vhoda (G2A, G2B) za omogočenje v negativni logiki.

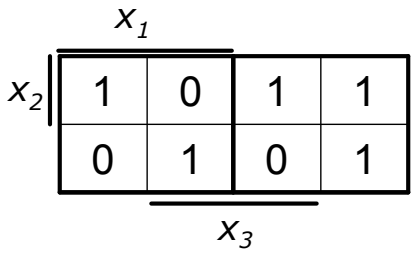
Delovanje dekoderja 74138[2] povzema spodnja tabela:

G_1	G_2'*	*a*	*b*	*c*	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
X	1	X	X	X	1	1	1	1	1	1	1	1
0	X	X	X	X	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1	1	1
1	0	1	0	0	1	1	1	1	0	1	1	1
1	0	1	0	1	1	1	1	1	1	0	1	1
1	0	1	1	0	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	1	1	1	1	1	0

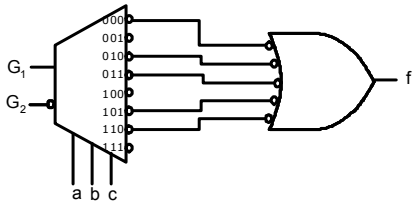


* G2'=G2A·G2B

Funkcijo *f* narišemo v Veitch−ev diagram, da si jo lažje predstavljamo:



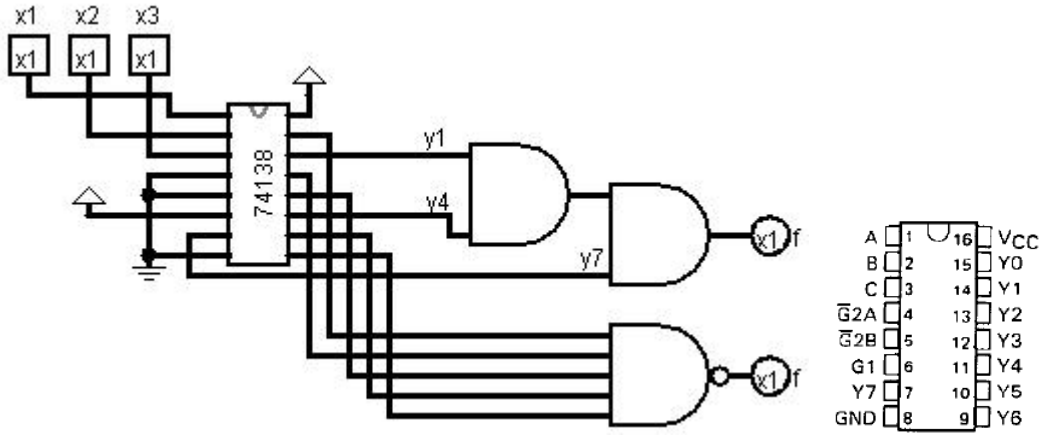
Čim imamo na voljo dekodirnik z ENABLE vhodom v negativni logiki preverimo ali obstaja spremenljivka v osnovni ali negirani obliki, pri kateri so vsa polja enaka '1' vključno z redundancami X. Te možnosti pri realizaciji funkcije na diagramu ni. Če želimo funkcijo realizirati, moramo zbrati '1' v diagramu. Ker so izhodi dekodirnika v negativni logiki, bomo pred vhodi OR vrat narisali negacije.



Upoštevamo De Morganovo enakost (x'+y' = (x·y)') in OR vrata pretvorimo v NAND vrata (ang. *pushing the bubble*).

[2]http://www.alldatasheet.com/view.jsp?Searchword=74HC138

Druga možnost realizacije je, da funkcijo *f* pretvorimo v PKNO: Poiščemo manjkajoče minterme v $f^3 = V(0, 2, 3, 5, 6)$ in jih pretvorimo v ustrezne maksterme $f^3 = \&(6, 3, 0)$. Na drugem nivoju PKNO postavimo trovhodna AND vrata na vhod katerih vodimo *manjkajoče minterme (1*, 4, 7), saj dekoder na izhodih ($Y_0 \ldots Y_7$) realizira številke mintermov ($m_0 \ldots m_7$) in ne makstermov. Trovhodna AND vrata realiziramo z manj dvovhodnimi vrati kot ena 5 vhodna NAND vrata, zato je PKNO realizacija cenejša.

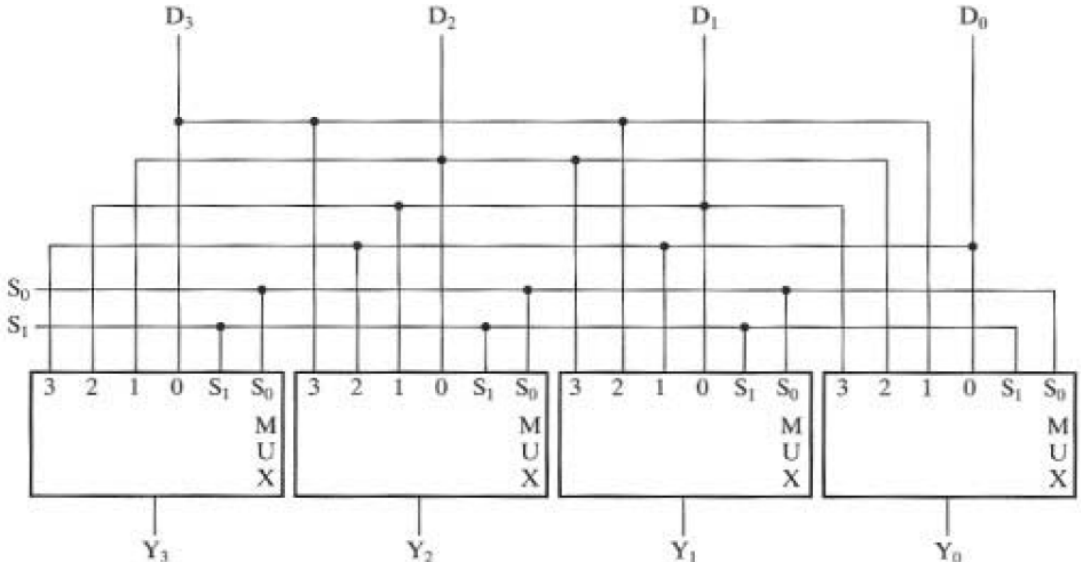


Na sliki realizacije sta narisani PKNO (zgornji izhod *f*) in PDNO (spodnji izhod *f*) izvedbi funkcije. Vezje realizacije funkcije je v predlogah vaj na domači strani predmeta v imeniku Logisim\decoder\dmux_3_8_74138_and_2_input_gates.circ

14. Izdelajte 4−bitni vzporedni pomikalnik podatkov (ang. barrel shifter), ki omogoča operacije glede na stanje krmilnih vhodov S_0 in S_1. Vezje ima 4−bitni vhod $D_3D_2D_1D_0$ in 4−bitni izhod $Y_3Y_2Y_1Y_0$ in izbirni vhod S_1S_0, ki določa koliko mest se bo vsebina pomaknila. Če je na začetku na vhodih stanje $Y_3Y_2Y_1Y_0=D_3D_2D_1D_0$ se ob izbranem številu mest pomika (vhod S_1S_0) na izhodih pojavi pomaknjena vsebina kot kaže spodnja tabela. Za realizacijo uporabite izbiralnike 4/1.

S_1	S_0	Y_3	Y_2	Y_1	Y_0	*funkcija*
0	0	D_3	D_2	D_1	D_0	ni pomika
0	1	D_2	D_1	D_0	D_3	pomik 1 mesto levo
1	0	D_1	D_0	D_3	D_2	pomik 2 mesti levo
1	1	D_0	D_3	D_2	D_1	pomik 3 mesta levo

V modernih procesorjih večkrat želimo pomikati vsebino podatka ali operanda na nivoju mikroukazov za n mest *naenkrat*. V ta namen lahko uporabimo posebno kombinacijsko vezje vzporedni pomikalnik podatkov (ang. barrel shifter), ki to naredi v enem ciklu signala ure. Če bi uporabljali sekvenčno realizacijo pomikalne enote, kot je denimo pomikalni register, bi za pomikanje za *n* mest potrebovali *n* ciklov, saj v enem ciklu pri pomikalnem registru naredimo naenkrat samo en pomik.
Realizacijo vezja lahko narišemo neposredno iz tabele, saj se iz nje vidi, na kateri naslov moramo priključiti kateri vhod (D_i), da dobimo realiziran izhod izbiralnika (Y_i). Za vsak bit izhoda porabimo en izbiralnik 4/1.



Za večbitne strukture vzporednega pomikalnika podatkov postane kombinacijska izvedba prezahtevna, saj zahteva za svoje delovanje velik *fan-in*, saj za *n* bitno izvedbo potrebujemo *n*/1 izbiralnike, zato se takšne strukture vežejo v nivojih. Pri MSI izvedbah brez realizacije MUX v nivojih lahko izdelamo največ 8 bitna pomikanja, saj je 8/1 MUX največji izbiralnik v TTL družini.

Takšno kombinacijsko strukturo pomikalnika podatkov v VHDL realizira (`with … select`) stavek. Vhod S določa za koliko mest se bo vsebina pomikalnika rotirala.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity barrel_shifter_4bit is
    Port (  S  : in STD_LOGIC_VECTOR (2 downto 0);
            D  : in  STD_LOGIC_VECTOR (3 downto 0);
            Y  : out  STD_LOGIC_VECTOR (3 downto 0)
          );
end barrel_shifter_4bit;

architecture arch of barrel_shifter_4bit is
begin
   --vhod S doloca, koliko mest se rotira vsebina
     with S select
     Y <= D(3) & D(2) & D(1) & D(0)  when "00", -- nic mest levo
          D(2) & D(1) & D(0) & D(3)  when "01", -- eno mesto levo
          D(1) & D(0) & D(3) & D(2)  when "10", -- dve mesti levo
          D(0) & D(3) & D(2) & D(1)  when others; -- tri mesta levo
end arch;
```

15. Izdelajte 9 bitni kombinacijski generator lihe parnosti (paritete). Vezje postavi izhod *f='1'*, če je število enic na 9 bitnem vhodu liho, sicer je *f='0'*.

Generator parnosti (paritite) je vezje, ki se uporablja za enostavno preverjanje pravilnosti prenosa informacije, ki je izvedeno s štetjem oddanih enic v informaciji. Oddajnik odda to število enic (parnost) skupaj z informacijo, sprejemnik pa preveri skladnost parnosti in sprejete informacije. Vezje obstaja v sekvenčni in kombinacijski izvedbi. Sekvenčno izvedbo si bomo ogledali pozneje pri obdelavi sekvenčnih vezij.
Kombinacijska izvedenka temelji na XOR operaciji, oz. seštevanju po modulu 2. Če seštevamo po modulu 2, števili seštejemo in v vsoti upoštevamo samo najnižje mesto (LSB). Seštevanje po modulu 2 ima dva možna rezultata: LSB='0' in LSB='1'.
Če želimo v danem dvojiškem številu prešteti koliko enic vsebuje (sodo, liho) dejansko tvorimo vsoto po modulu 2 med posameznimi biti števila.
Naj bo na vhodu število 10111_2. Število enic preštejemo tako, da seštevamo posamezne med sabo (kar je enako kot če preštejemo število enic) :

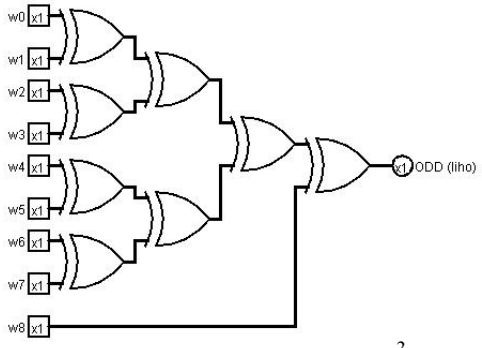
$$\begin{array}{r} 1 \\ +0 \\ +1 \\ +1 \\ +1 \\ \hline 100_2 \end{array}$$

V tem primeru so štiri enice, kar ugotovimo tako, da preberemo vsoto bitov (100_2). Število enic je sodo, kar ugotovimo iz LSB='0'.

Izhod generatorja parnosti je lahko *f='1'*, ko je število enic števila na vhodu liho. Takrat govorimo o lihi parnosti (ang. *odd parity*). Druga možnost *soda parnost* (ang. *even parity*), kjer postane f='1', ko je število enic števila na vhodu *sodo*. Za 9 bitno število moramo tvoriti XOR med posameznimi biti:

$$ODD = w_0 \oplus w_1 \oplus w_2 \oplus w_3 \oplus w_4 \oplus w_5 \oplus w_6 \oplus w_7 \oplus w_8$$
$$ODD = (w_0 \oplus w_1) \oplus (w_2 \oplus w_3) \oplus (w_4 \oplus w_5) \oplus (w_6 \oplus w_7) \oplus w_8$$

Za n bitno funkcijo XOR lahko uporabimo lastnost združevanja (asociativnost) in vezje n−vhodnih XOR vrat realiziramo kot kaskado dvovhodnih vrat. Opis delovanja in vezje 9 bitnega generatorja lihe parnosti je v predlogah vaj na domači strani predmeta v imeniku Logisim\combinatorial\odd_parity_checker_9_bit.circ:



MSI kombinacijski generator parnosti je vezje 74280[3], ki realizira funkcijo devetih spremenljivk XOR in XNOR, vendar ne z XOR in XNOR vrati, pač pa z uporabo Boole−ove realizacije XOR $x \oplus y = \bar{x} \cdot y + x \cdot \bar{y}$ in XNOR $x \equiv y = \bar{x} \cdot \bar{y} + x \cdot y$.

Liha dvojiška števila imajo LSB='1', soda števila pa LSB='0'. Včasih želimo šteti samo po sodih ali samo po lihih vrednostih, kar izvedemo tako, da štejemo normalno in opazujemo kdaj je LSB mesto enako '1' oz. '0'.

[3] http://www.alldatasheet.com/view.jsp?Searchword=74280

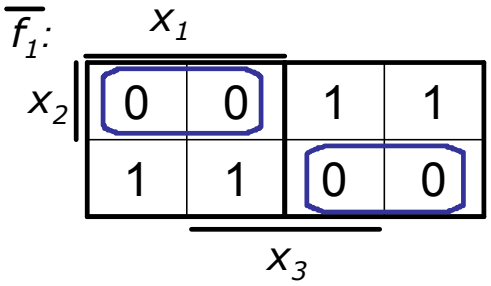
16. Uporabite PAL3L3 (namišljen čip) za realizacijo naslednjih funkcij:
- $f_1 = x_1 \oplus x_2$
- f_2 = konjunkcijo treh spremenljivk
- f_3 = funkcijo treh spremenljivk, ki vrne '1' pri vsaj dveh enicah na vhodih.

Vezje ima 3 vhode in 3 izhode. Vsaka disjunkcija (OR) ima 4 produkte (AND). L pa pomeni, da je izhod negiran. Povezave oz. 'varovalke' označite s piko (●).

Za funkcije zapišemo najprej pravilnostno tabelo, nato narišemo Veitch−eve diagrame.

x_1	x_2	x_3	f_1	f_2	f_3
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	1	0	0
0	1	1	1	0	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	0	0	1
1	1	1	0	1	1

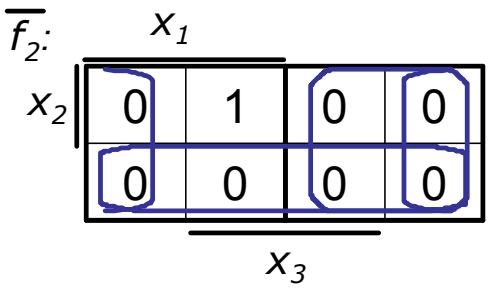
Vezje PAL ima negirane izhode, zato bomo pri realizaciji funkcij z Veitch−evimi diagrami realizirali f' in ne f.



Prvo funkcijo zapišemo enostavno, saj je negacija XOR funkcije dveh spremenljivk kar funkcija ekvivalence:

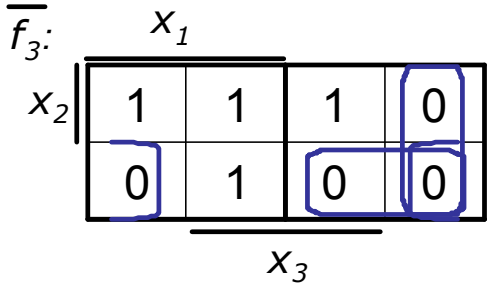
$$\overline{f_1} = x_1 \cdot x_2 + \overline{x_1} \cdot \overline{x_2}$$

Podobno lahko naredimo za drugo funkcijo, kjer za negacijo konjunkcije treh spremenljivk uporabimo De Morgan−ovo enakost.



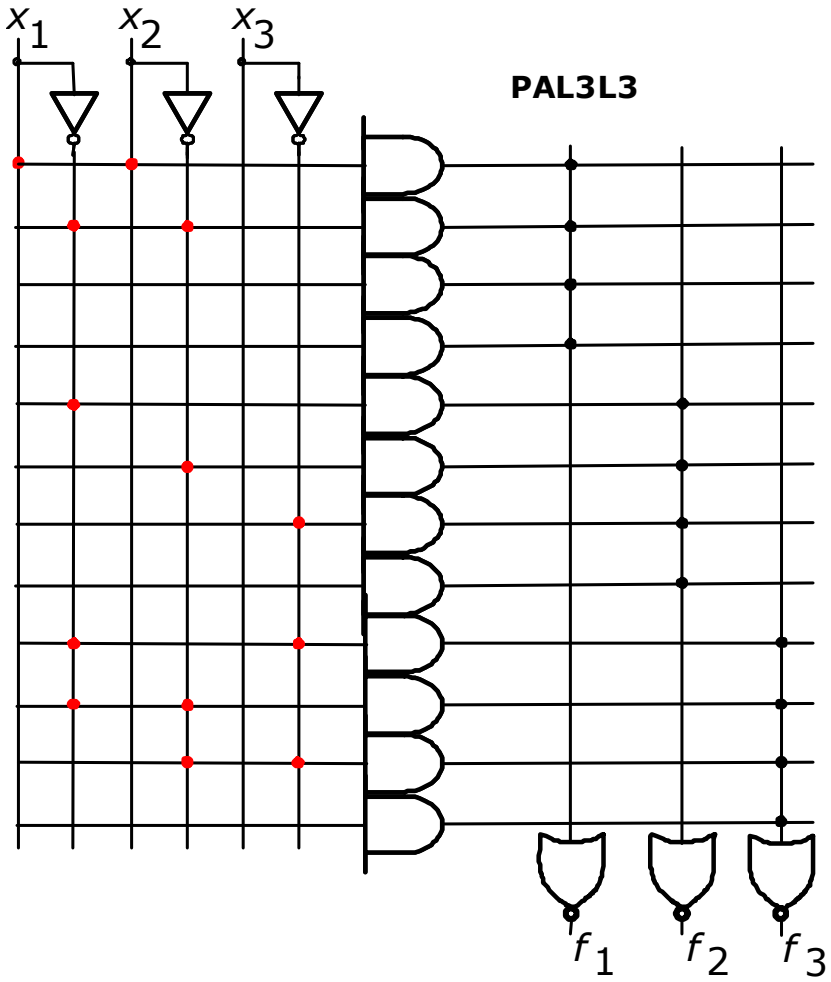
$$f_2 = \overline{\overline{x_1 \cdot x_2 \cdot x_3}}$$
$$\overline{f_2} = \overline{x_1} + \overline{x_2} + \overline{x_3}$$

Zadnjo funkcijo minimiziramo z uporabo Veitch−evega diagrama, tako da zbiramo ničle.



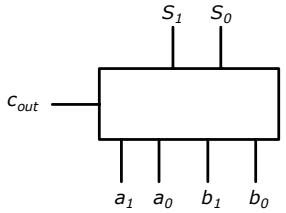
$$\overline{f_3} = \overline{x_1} \cdot \overline{x_3} + \overline{x_1} \cdot \overline{x_2} + \overline{x_2} \cdot \overline{x_3}$$

Pri realizaciji PAL vezja upoštevamo poenostavljeno strukturo, pri kateri ne vežemo vsake povezave na konjunkcije, saj so AND vrata na narisani strukturi 6−vhodna. Vezje PAL3L3 je AND−NOR arhitekture in vsebuje 4 konjunkcije na en NOR člen. Pri PAL vezju je programabilen samo AND del vezja.



Predstavljeno PAL3L3 vezje je sicer izmišljeno, vendar demonstrira strukturo in uporabo večjih (realnih) PAL vezij kot so npr. PAL14L4, GAL16V8 in GAL22V10. GAL vezja so nadgradnja osnovne PAL strukture. Slednji so izključno kombinacijski, GAL vezja pa imajo v OLMC (ang. Output Logic MacroCell) strukturi še D−FF, s katerim lahko realiziramo tudi sekvenčna vezja.

17. Uporabite PAL vezje, za realizacijo bita vsote s_1 pri seštevanju dveh dvobitnih števil a_1a_0 in b_1b_0.
Izmišljeno vezje PAL ima 4 vhode in 2 izhoda. Vsaka disjunkcija (OR) ima 4 produkte (AND). Oba izhoda OR vrat sta povezana nazaj na AND matriko v negirani in nenegirani obliki. Izhodi OR vrat so aktivno visoki (nenegirani). Povezave oz. 'varovalke' označite s piko (●).



Za funkcije zapišemo najprej pravilnostno tabelo, nato narišemo Veitch−eve diagrame.

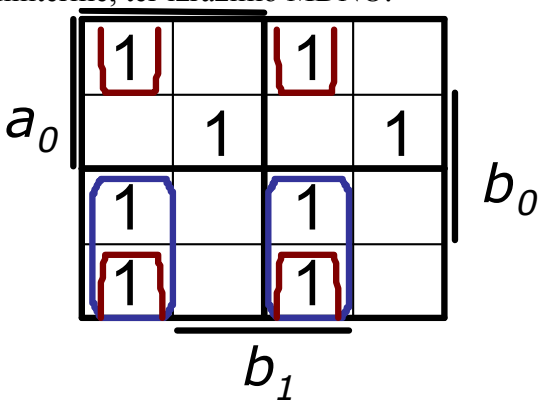
Pri seštevanju dveh dvobitnih števil imamo na vsakem vhodu a_1a_0 kvečjemu 3_{10}, torej so možne vsote od 0_{10} do 6_{10}. Zanimajo nas tiste vsote, pri katerih je bit s_1 postavljen na '1'. To so števila 2_{10} (010_2), 3_{10} (011_2), 6_{10} (110_2).

a_1	a_0	b_1	b_0	c_{out}	s_1	s_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

Iz zgornje tabele preberemo bit vsote s_2 v obliki PDNO, tako da posamezno številko minterma tvorimo sestavljeno iz bitov (a_1a_0 in b_1b_0):

$$s_1 = V(2,3,5,6,8,9,12,15)$$

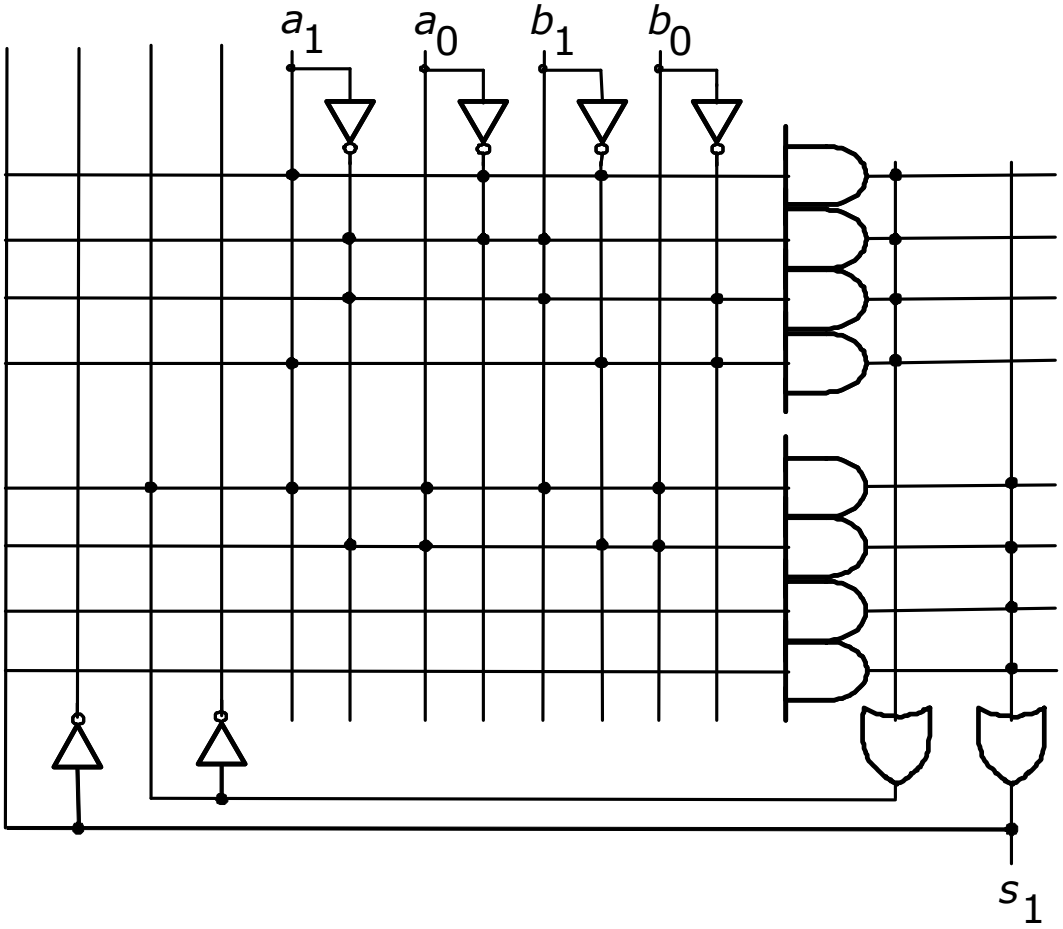
Funkcijo narišemo v Veitchev diagram in združimo morebitne sosedne minterme, ter izrazimo MDNO.



MDNO oblika pretirano poenostavljena, saj so zgornje štiri enice v diagramu XOR funkcija treh spremenljivk.

$$s_1 = a_1 \cdot \overline{a_0} \cdot \overline{b_1} + \overline{a_1} \cdot \overline{a_0} \cdot b_1 + \overline{a_1} \cdot b_1 \cdot \overline{b_0} +$$
$$+ a_1 \cdot \overline{b_1} \cdot \overline{b_0} + \overline{a_1} \cdot a_0 \cdot \overline{b_1} \cdot b_0 + a_1 \cdot a_0 \cdot b_1 \cdot b_0$$

Funkcije XOR neposredno s PAL vezjem ne moremo realizirati, saj lahko realiziramo samo DNO oblike. Narišemo celotno vezje PAL strukture in vstavimo pike (●) v AND matriko tam, kjer želimo programirati določeno spremenljivko v členu MDNO.



Če bi želeli realizirati celoten seštevalnik dveh dvobitnih števil a_1a_0 in b_1b_0 bi podobno realizirali še LSB bit vsote (s_0) in izhodni prenos (c_{out}). Za LSB bit vsote bi po minimizaciji v Veitch−evem diagramu zapisali $s_0=a_0 \oplus b_0$, kar se vidi tudi iz pravilnostne tabele seštevalnika. Izhodni prenos (c_{out}) bi izrazili v obliki PDNO:

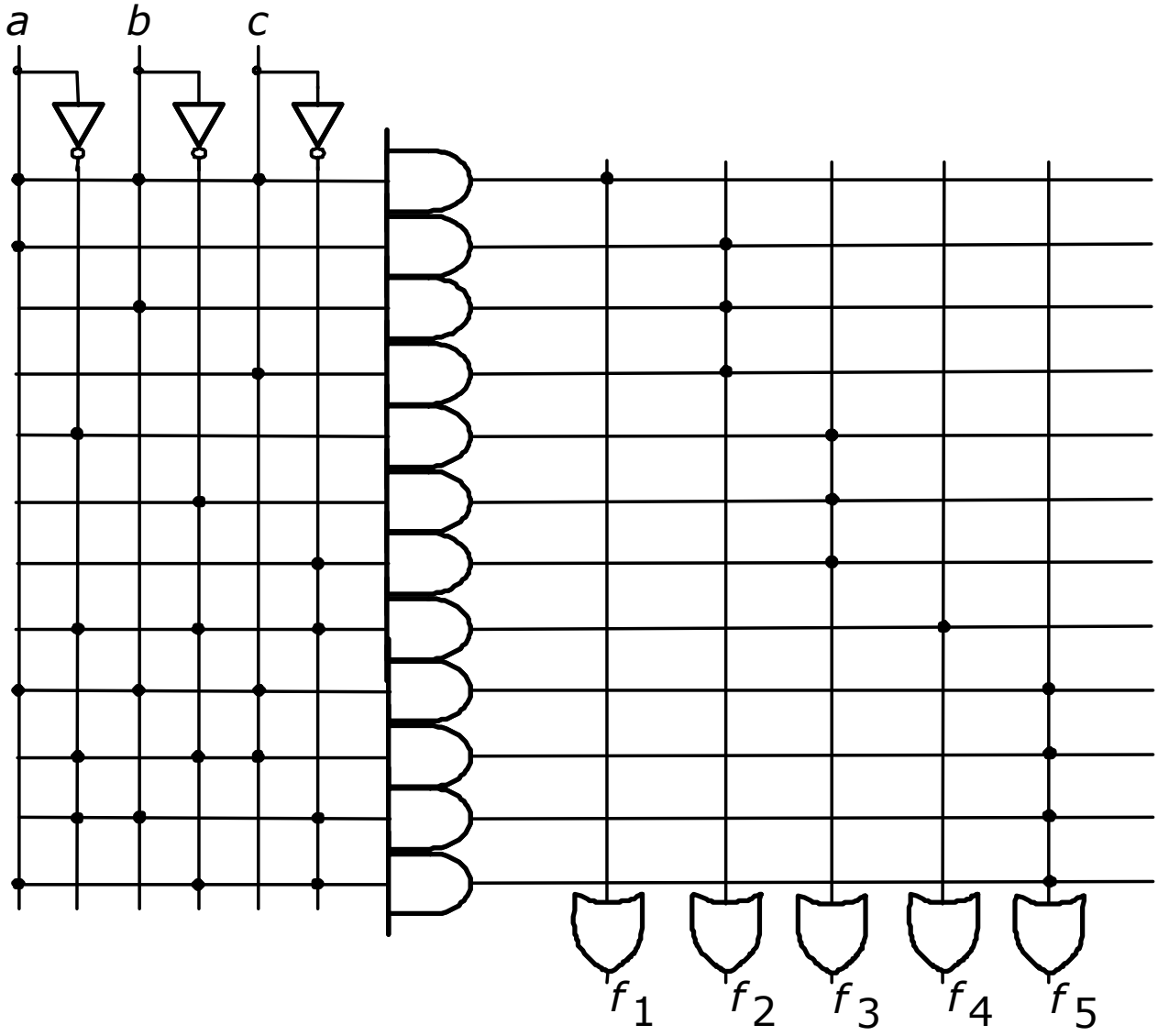
$$c_{out} = V(7,10,11,13,14,15)$$

18. Uporabite PLA vezje za realizacijo naslednjih funkcij:
$f_1 = a \cdot b \cdot c \quad f_2 = a + b + c \quad f_3 = a \uparrow b \uparrow c \quad f_4 = a \downarrow b \downarrow c \quad f_5 = a \oplus b \oplus c$
PLA vezje ima 3 vhode in 5 izhodov. Povezave oz. 'varovalke' označite s piko (●).

Funkcije najprej poenostavimo z uporabo pravil Boole−ove algebre, tako da jih prevedemo na disjunktivno normalno obliko (DNO):

$$f_1 = a \cdot b \cdot c$$
$$f_2 = a + b + c$$
$$f_3 = a \uparrow b \uparrow c = \overline{a \cdot b \cdot c} = \bar{a} + \bar{b} + \bar{c}$$
$$f_4 = a \downarrow b \downarrow c = \overline{a + b + c} = \bar{a} \cdot \bar{b} \cdot \bar{c}$$
$$f_5 = a \oplus b \oplus c$$
$$f_5 = V(1,2,4,7) = a \cdot b \cdot c + \bar{a} \cdot \bar{b} \cdot c + \bar{a} \cdot b \cdot \bar{c} + a \cdot \bar{b} \cdot \bar{c}$$

Narišemo celotno vezje PLA strukture in vstavimo pike (●) v AND in OR matriki tam, kjer želimo programirati določeno spremenljivko v členu DNO.



19. Uporabite ROM vezje za realizacijo naslednjih funkcij:

$g_1 = x_1 + \overline{x_2} \cdot \overline{x_3}$ $g_2 = \overline{x_1} \cdot x_3 + x_1 \cdot x_2$ $g_3 = \overline{x_2} \cdot \overline{x_3} + x_1 \cdot x_2$ $g_4 = \overline{x_2} \cdot x_3 + x_1$

ROM vezje ima 3 vhodne spremenljivke in 4 bitno vsebino. Povezave oz. 'varovalke' označite s piko (●).

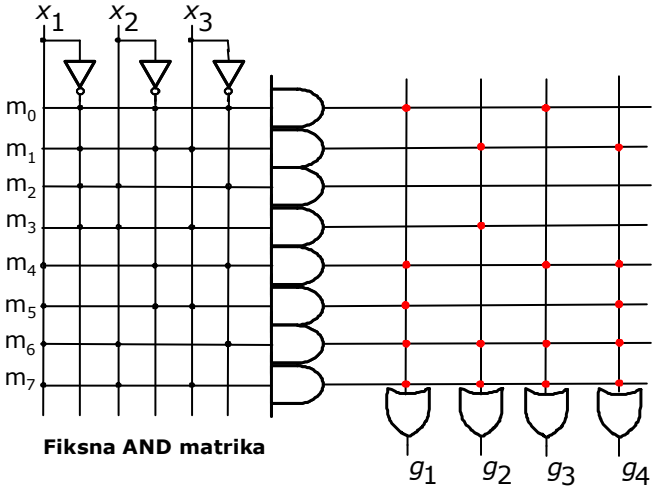
Če se funkcije ne nahajajo v popolni disjunktivni normalni obliki (PDNO), jih prevedemo v to obliko z uporabo pravil Boole−ove algebre. Funkcijo lahko tudi izpišemo v Veitch−ev diagram in izpišemo številke mintermov, kjer je funkcija enaka '1'.

$$g_1(x_1,x_2,x_3) = x_1 + \overline{x_2} \cdot \overline{x_3} = x_1 \cdot \left(\overline{x_2} \cdot \overline{x_3} + x_2 \cdot \overline{x_3} + \overline{x_2} \cdot x_3 + x_2 \cdot x_3\right) + \left(\overline{x_1} + x_1\right) \cdot \overline{x_2} \cdot \overline{x_3}$$
$$g_1(x_1,x_2,x_3) = x_1 \cdot \overline{x_2} \cdot \overline{x_3} + x_1 \cdot x_2 \cdot \overline{x_3} + x_1 \cdot \overline{x_2} \cdot x_3 + x_1 \cdot x_2 \cdot x_3 + x_1 \cdot \overline{x_2} \cdot \overline{x_3} + \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}$$
$$g_1(x_1,x_2,x_3) = x_1 \cdot \overline{x_2} \cdot \overline{x_3} + x_1 \cdot x_2 \cdot \overline{x_3} + x_1 \cdot \overline{x_2} \cdot x_3 + x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3}$$
$$g_1(x_1,x_2,x_3) = V(4,6,5,7,0)$$

Podobno storimo še za preostale funkcije:

$$g_2 = \overline{x_1} \cdot x_3 + x_1 \cdot x_2 = \overline{x_1} \cdot \left(\overline{x_2} + x_2\right) \cdot x_3 + x_1 \cdot x_2 \cdot \left(\overline{x_3} + x_3\right)$$
$$g_2(x_1,x_2,x_3) = \overline{x_1} \cdot \overline{x_2} \cdot x_3 + \overline{x_1} \cdot x_2 \cdot x_3 + x_1 \cdot x_2 \cdot \overline{x_3} + x_1 \cdot x_2 \cdot x_3$$
$$g_2(x_1,x_2,x_3) = V(1,3,6,7)$$
$$g_3 = \overline{x_2} \cdot \overline{x_3} + x_1 \cdot x_2 =$$
$$g_3(x_1,x_2,x_3) = \left(\overline{x_1} + x_1\right) \cdot \overline{x_2} \cdot \overline{x_3} + x_1 \cdot x_2 \cdot \left(\overline{x_3} + x_3\right) =$$
$$g_3(x_1,x_2,x_3) = \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} + x_1 \cdot \overline{x_2} \cdot \overline{x_3} + x_1 \cdot x_2 \cdot \overline{x_3} + x_1 \cdot x_2 \cdot x_3$$
$$g_3(x_1,x_2,x_3) = V(0,4,6,7)$$
$$g_4(x_1,x_2,x_3) = \overline{x_2} \cdot x_3 + x_1$$
$$g_4(x_1,x_2,x_3) = \left(\overline{x_1} + x_1\right) \cdot \overline{x_2} \cdot x_3 + x_1 \cdot \left(\overline{x_2} \cdot \overline{x_3} + x_2 \cdot \overline{x_3} + \overline{x_2} \cdot x_3 + x_2 \cdot x_3\right)$$
$$g_4(x_1,x_2,x_3) = \left(\overline{x_1} + x_1\right) \cdot \overline{x_2} \cdot x_3 + x_1 \cdot \left(\overline{x_2} \cdot \overline{x_3} + x_2 \cdot \overline{x_3} + \overline{x_2} \cdot x_3 + x_2 \cdot x_3\right)$$
$$g_4(x_1,x_2,x_3) = \overline{x_1} \cdot \overline{x_2} \cdot x_3 + x_1 \cdot \overline{x_2} \cdot x_3 + x_1 \cdot \overline{x_2} \cdot \overline{x_3} + x_1 \cdot x_2 \cdot \overline{x_3} + x_1 \cdot \overline{x_2} \cdot x_3 + x_1 \cdot x_2 \cdot x_3$$
$$g_4(x_1,x_2,x_3) = V(1,5,4,6,7)$$

PDNO je najprimernejša oblika za realizacijo z ROM, ker je matrika AND fiksna. Programirane vrednosti AND matrike predstavljajo vse minterme funkcije treh spremenljivk ($x_1x_2x_3$) od m_0 do m_7. Številka minterma določa naslov lokacije ROM pomnilnika.



Narišemo celotno vezje ROM strukture in vstavimo pike (●) v OR matriki tam, kjer želimo programirati določeno spremenljivko v členu PDNO.

Vsebino ROM pomnilnika običajno podajamo s tabelo v kateri programirane povezave (●) v OR matriki pišemo kot '1' na danem mestu vsebine.

Minterm $g_i(x_1x_2x_3)$	*Naslov lokacije* $x_1x_2x_3$	*Vsebina lokacije* $g_1g_2g_3g_4$
m_0	000_2	1010_2
m_1	001_2	0101_2
m_2	010_2	0000_2
m_3	011_2	0100_2
m_4	100_2	1011_2
m_5	101_2	1001_2
m_6	110_2	1111_2
m_7	111_2	1111_2

Realni ROM elementi imajo 8 bitne podatke, zloge ali včasih oktete (ang. *byte, octet*). Vsebina ROM elementov se podaja v datoteki, ki jo nato programiramo z posebnim inštrumentom (ROM programatorjem).
Najenostavnejši način podajanja zapisa vsebine ROM elementa je v surovi dvojiški obliki (ang. *raw binary file*) v kateri si 8 bitni podatki sledijo zapisani v dvojiški obliki. Kompleksnejša zapisa podajanja vsebine ROM s tabelo sta INTEL šestnajstiška oblika (ang. *Intel hex record*) in Motorola SREC oblika (ang. *Motorola S record*).

20. Realizirajte pretvornik kode, ki na vhodu sprejme zadnjih 5 bitov ASCII kode v dvojiškem zapisu, na izhodu pretvori to kodo v 4−bitno dvojiško predstavitev izbranega ASCII znaka. Pri pretvorbi kode upoštevajte, da se na vhodu lahko pojavijo ASCII znaki števk '0' …'9' in velike črke 'A'….'F'. Ostale kombinacije so redundantne. Za realizacijo uporabite PAL14L4, ki uporablja AND−NOR logiko.
    Primer: Če je na vhodu ASCII znak števila nič '0', kar iz ASCII tabele preberemo kot 30_{16}, potem na izhodu dobimo kombinacijo 0000_2, kar je šestnajstiško število 0_{16}. Če je na vhodu ASCII znak 'E', kar iz ASCII tabele preberemo kot 45_{16}, potem na izhodu dobimo kombinacijo 1110_2, kar je šestnajstiško število E_{16}.

Iz ASCII tabele preberemo kode znakov, ki se lahko pojavijo na vhodu − torej ASCII znaki števk '0' …'9' in pa velike črke 'A'….'F'

znak	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
ASCII $koda_{16}$	30	31	32	33	34	35	36	37	38	39	41	42	43	44	45	46

V prvi stolpec zapišemo ASCII kodo znaka in jo izpišemo v dvojiškem sistemu v drugi koloni. Opazimo, da se vrednosti res razlikujejo šele v spodnjih 5 bitih ASCII kode, zato jih izpišemo v stolpce vhodnih spremenljivk A…E.

$znak_{16}$	**$znak_2$**	**minterm**	**A**	**B**	**C**	**D**	**E**	**y_3**	**y_2**	**y_1**	**y_0**
30	0011 0000	16	1	0	0	0	0	0	0	0	0
31	0011 0001	17	1	0	0	0	1	0	0	0	1
32	0011 0010	18	1	0	0	1	0	0	0	1	0
33	0011 0011	19	1	0	0	1	1	0	0	1	1
34	0011 0100	20	1	0	1	0	0	0	1	0	0
35	0011 0101	21	1	0	1	0	1	0	1	0	1
36	0011 0110	22	1	0	1	1	0	0	1	1	0
37	0011 0111	23	1	0	1	1	1	0	1	1	1
38	0011 1000	24	1	1	0	0	0	1	0	0	0
39	0011 1001	25	1	1	0	0	1	1	0	0	1
41	0100 0001	1	0	0	0	0	1	1	0	1	0
42	0100 0010	2	0	0	0	1	0	1	0	1	1
43	0100 0011	3	0	0	0	1	1	1	1	0	0
44	0100 0100	4	0	0	1	0	0	1	1	0	1
45	0100 0101	5	0	0	1	0	1	1	1	1	0
46	0100 0110	6	0	0	1	1	0	1	1	1	1

Funkcije zapišemo v PDNO obliki − ob tem si pomagamo s številkami mintermov v zgornji tabeli. Ker je PAL realiziran v AND−NOR logiki, bomo zbirali logične '0' na izhodu, tako da dobimo izhodno negacijo, ki jo zahteva NOR. S 5 biti lahko zapišemo minterme 0..31, tako da so vsi mintermi, ki niso navedeni v zgornji tabeli redundantni, torej je njihova vrednost poljubna (X).

$$
\begin{aligned}
\overline{y_3} &= \sum(16-23) + \sum_X(0,7-15,26-31) \\
\overline{y_2} &= \sum(16-25,1,2) + \sum_X(0,7-15,26-31) \\
\overline{y_1} &= \sum(16,17,20,21,24,25,3,4) + \sum_X(0,7-15,26-31) \\
\overline{y_0} &= \sum(16,18,20,22,24,1,3,5) + \sum_X(0,7-15,26-31)
\end{aligned}
\tag{14.1}
$$

Izrišemo Veitch−eve diagrame za 5 spremenljivk. To storimo tako, da postavimo dva Veitch−eva diagrama 4 spremenljivk enega ob drugega ju ločimo z najbolj pomembno spremenljivko. Lego spremenljivk lahko menjamo, vendar se moramo zavedati, da se ob tem spreminjajo tudi številke mintermov. Spodnja realizacija je ločena po najmanj pomembni spremenljivki E.

Primer Veitch−evega diagrama 5 spremenljivk z označenimi številkami mintermov:

	E							
	A				**A**			
B	25	19	13	9	24	28	12	8
	27	31	15	11	26	30	14	10
	19	23	7	3	18	22	6	2
	17	21	5	1	16	20	4	0
	C				**C**			**D**

Vezje PAL14L4[4] vsebuje 14 vhodov, 4 izhode in ima 4 konjunkcije (AND) na eno disjunkcijo (OR), kar upoštevamo pri realizaciji MDNO.

Izhodni diagram za y_3:

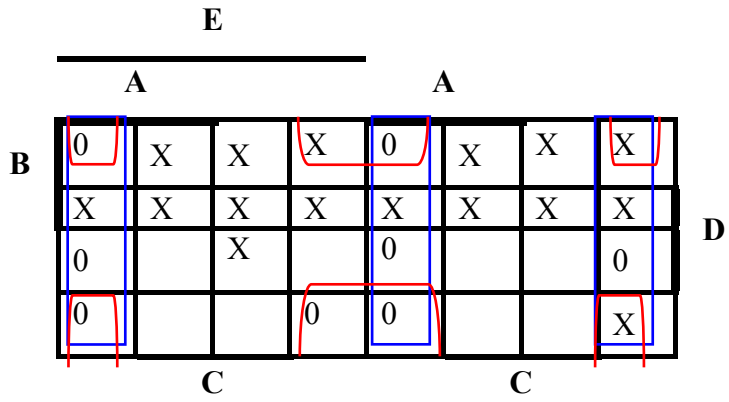
	E							
	A				**A**			
B		X	X	X		X	X	X
	X	X	X	X	X	X	X	X
	0	0	X		0	0		
	0	0			0	0		X
	C				**C**			**D**

Združimo lahko 8 ničel skupaj in dobimo funkcijo dveh spremenljivk:

$$\overline{y_3} = A \cdot \overline{B} \tag{14.2}$$

[4] http://www.ic-elect.si/pub/files/product_files/120142040100.pdf

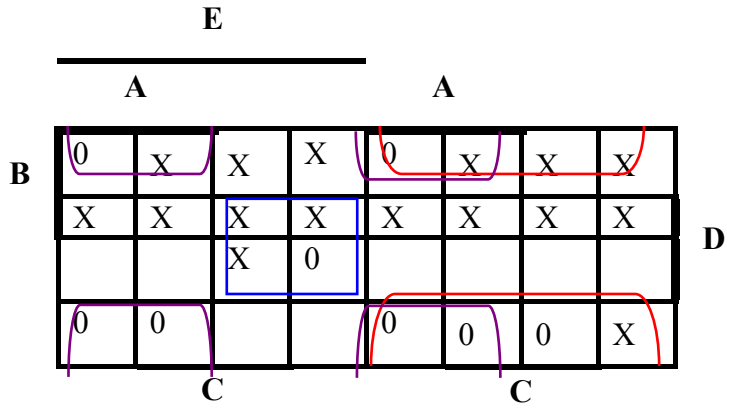
Izhodni diagram za y_2:



Združimo skrajni levi stolpec ničel in skrajno desnega, da dobimo prvi člen izraza $A \cdot \overline{C}$. Nato združimo peti stolpec in skrajno desni stolpec v drugi člen izraza $\overline{E} \cdot \overline{C}$, nato združimo še ničle v vseh kotih diagrama in ničle na sredini, da dobimo še člen $\overline{C} \cdot \overline{D}$.

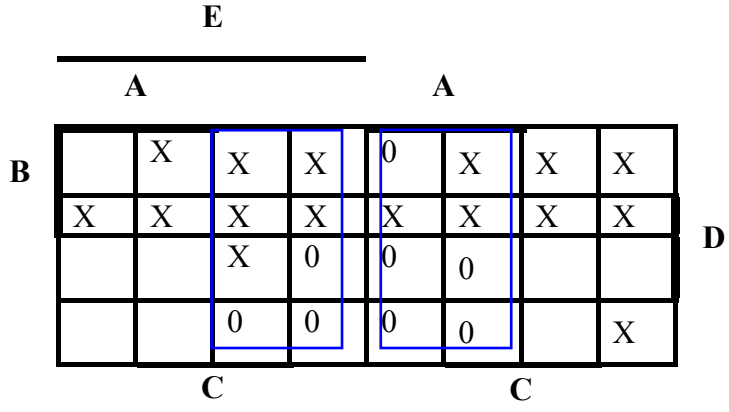
$$\overline{y_2} = A \cdot \overline{C} + \overline{E} \cdot \overline{C} + \overline{C} \cdot \overline{D} \tag{14.3}$$

Izhodni diagram za y_1:



$$\overline{y_1} = A \cdot \overline{D} + \overline{D} \cdot \overline{E} + \overline{A} \cdot D \cdot E \tag{14.4}$$

Izhodni diagram za y_0:



$$\overline{y_0} = A \oplus E \tag{14.5}$$

21. Realizirajte primerjalnik velikosti (ang. magnitude comparator) dveh 8 bitnih nepredznačenih števil A in B. Primerjalnik na izhod postavi '1', ko je A>B, sicer je izhod '0'.

Realizacijo primerjalnika dveh nepredznačenih 8 bitnih števil bi lahko sicer poskusili izdelati s pomočjo ROM elementov, vendar bi v tem primeru morali realizirati vseh 256 *256 kombinacij, kar pomeni, da bi za realizacijo porabili en 64k ROM element. Realizacija bi bila precej neposredna, saj bi zgornjih 8 bitov porabili za prvo primerjano besedo A, spodnjih 8 bitov naslova pa za drugo primerjano besedo B. Na dano lokacijo bi vpisali '1', če bi veljalo A>B in '0' sicer.
Če bi isti princip želeli uporabiti na primeru 32 bitnih števil, si predstavljajte koliko spomina bi porabili.
Mikroprocesorji zato raje operacijo primerjave števil izvajajo s pomočjo odštevanja dveh števil in opazovanja najbolj pomembnega bita (bit predznaka).
Obstaja še bolj enostavno vezje, ki omogoča primerjavo nepredznačenih števil, a pri tem ne uporablja odštevanja, ampak primerjavo bitov. Tovrstna realizacija je primerna za izdelavo večjih (recimo 32−bitnih) primerjalnikov.

Na enostavnem zgledu si oglejmo kako deluje 4 bitni primerjalnik velikosti.
Če primerjamo števili A=9_{10} in B=7_{10} je jasno A>B. Zakaj?
A=1001= $A_3A_2A_1A_0$
B=0111= $B_3B_2B_1B_0$
Število 9_{10} je večje od 7_{10}, saj velja $A_3>B_3$, oziroma $A_3 \cdot \overline{B_3} = 1$.

Če primerjamo števili A=13_{10} in B=11_{10} je spet A>B. Zakaj?
A=1101
B=1011
Zato ker velja $A_2>B_2$, torej pišemo $A_2 \cdot \overline{B_2} = 1$ ob pogoju, da sta A_3 in B_3 enaka

In končno − če primerjamo A=11_{10} in B=10_{10} bo spet veljalo A>B. Zakaj?
A=1011
B=1010
Zato ker velja $A_0>B_0$, torej pišemo $A_0 \cdot \overline{B_0} = 1$
Zato ker velja $A_0>B_0$, torej pišemo $A_0 \cdot \overline{B_0} = 1$ ob pogoju, da sta A_3 in B_3 enaka **in** A_2 in B_2 enaka **in** A_1 in B_1 enaka.

Za izdelavo primerjalnika dveh 8 bitnih števil bomo torej potrebovali najprej primerjalnik enakosti dveh števil, ki ga izdelamo tako, da po soležnih bitih nad dvema številoma izvajamo operacijo ekvivalence (negacija XOR). Za 8 bitno število zapišemo rezultate primerjave med soležnimi biti s funkcijami C_0 …C_7.

$$C_0 = \overline{A_0 \oplus B_0} \quad .... \quad C_7 = \overline{A_7 \oplus B_7} \tag{15.1}$$

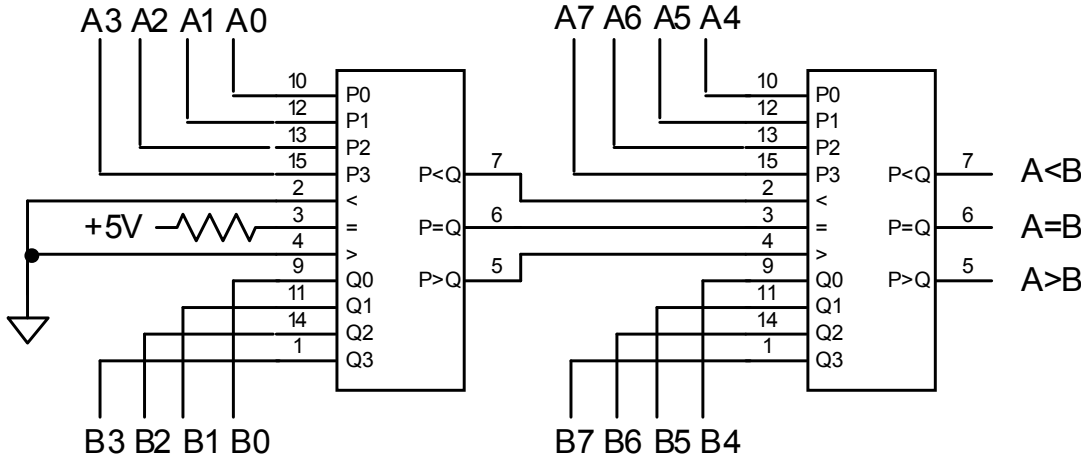
Ko sta soležna bita na i−tem mestu enaka bo vrednost funkcije C_i='1', sicer '0'. Če zapišemo spoznanje iz zgleda primerjave števil A=11_{10} in B=10_{10} (ki se razlikujeta na LSB mestu), potem bi rezultat primerjave števil zapisali s funkcijo:

$$f_0 = C_7 \cdot C_6 \cdot C_5 \cdot C_4 \cdot C_3 \cdot C_2 \cdot C_1 \cdot A_0 \cdot \overline{B_0} \tag{15.2}$$

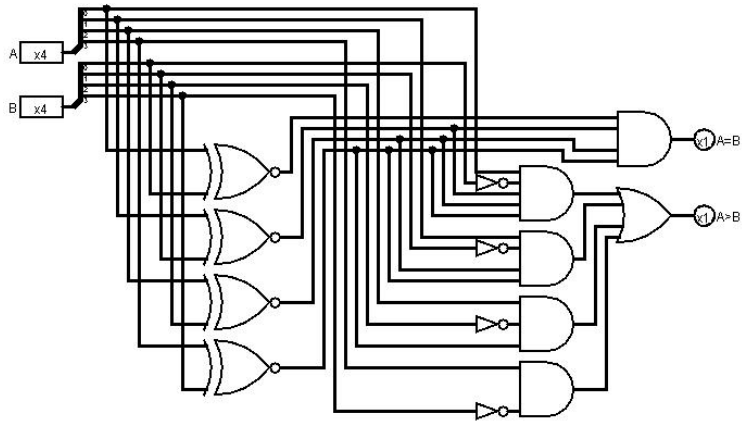
Za razlikovanja na ostalih mestih bi podobno lahko zapisali f_1, če se razlikujeta na mestu LSB+1 itd. do MSB mesta, pri katerem bi zapisali f_7:

$$
\begin{aligned}
f_0 &= C_7 \cdot C_6 \cdot C_5 \cdot C_4 \cdot C_3 \cdot C_2 \cdot C_1 \cdot A_0 \cdot \overline{B_0} \\
f_1 &= C_7 \cdot C_6 \cdot C_5 \cdot C_4 \cdot C_3 \cdot C_2 \cdot A_1 \cdot \overline{B_1} \\
f_2 &= C_7 \cdot C_6 \cdot C_5 \cdot C_4 \cdot C_3 \cdot A_2 \cdot \overline{B_2} \\
f_3 &= C_7 \cdot C_6 \cdot C_5 \cdot C_4 \cdot A_3 \cdot \overline{B_3} \\
f_4 &= C_7 \cdot C_6 \cdot C_5 \cdot A_4 \cdot \overline{B_4} \\
f_5 &= C_7 \cdot C_6 \cdot A_5 \cdot \overline{B_5} \\
f_6 &= C_7 \cdot A_6 \cdot \overline{B_6} \\
f_7 &= A_7 \cdot \overline{B_7}
\end{aligned}
\qquad (15.3)
$$

Za izvedbo celotnega vezja primerjalnika naredimo še OR operacijo med funkcijami $f_0 \ldots f_7$. Cenovno je ta rešitev ugodna za realizacijo v VHDL. Za realizacijo z elementi obstajajo posebni gradniki iz 74 družine vezij kot je denimo 7485[5], ki primerjajo 4 bitna števila. Na spodnji sliki je prikazana kaskadna vezava dveh 7485, s katerima realiziramo primerjavo dveh 8−bitnih števil A in B.



Funkcijo dveh 7485 opravlja eno vezje 74682[6], ki primerja velikost 8−bitnih števil. Z vezavo 74682 elementov enostavno izvedemo primerjavo 32−bitnih števil.
Opis delovanja in vezje 4 bitnega primerjalnika velikosti je v predlogah vaj na domači strani predmeta v imeniku Logisim\combinatorial\magnitude_comparator_4_bit.circ:



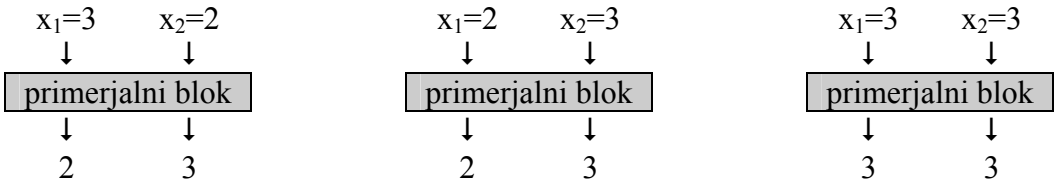
[5] http://www.alldatasheet.com/view.jsp?Searchword=74f85
[6] http://www.alldatasheet.com/view.jsp?Searchword=74HC682

Izvedbo primerjave dveh 8−bitnih števil zapišimo v VHDL:

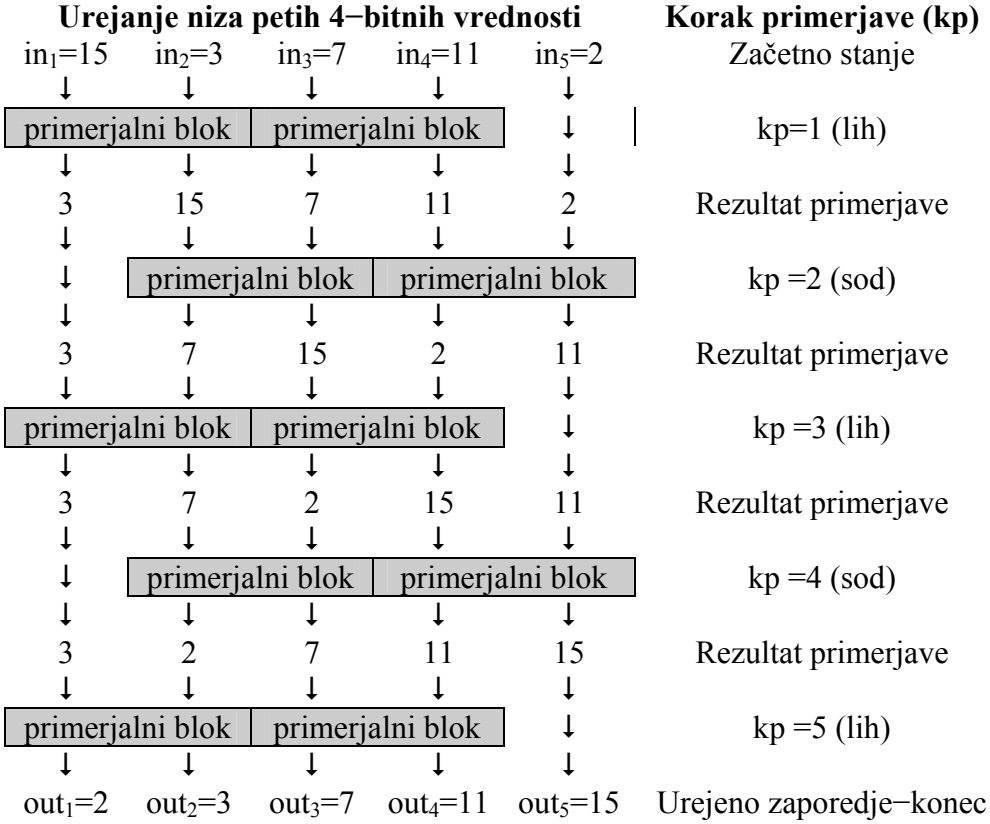
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity compare8bit is
    Port (  a, b : in STD_LOGIC_VECTOR (7 downto 0);
            a_vecji_od_b, a_enak_b, a_manjsi_od_b : out STD_LOGIC);
end compare8bit;

architecture arch of compare8bit is
signal c, f: STD_LOGIC_VECTOR (7 downto 0);
signal a_vecji_od_b_sig, a_enak_b_sig: STD_LOGIC;
begin
enakost_mest: FOR i IN 0 TO 7 GENERATE
                c(i) <= a(i) xnor b(i); --tvorba vseh primerjav
enakosti z xnor
              END GENERATE;
-- ce so vsa mesta stevil a, b enaka, sta stevili enaki
a_enak_b_sig <= c(0) and c(1) and c(2) and c(3) and c(4) and c(5)
and c(6) and c(7);

-- uvedemo vmesni signal (zico) a_enak_b _sig, da se izognemo tipu
inout pri izhodu a_enak_b, saj bomo v nadaljevanju a_enak_b povezali
tudi na vhod.
a_enak_b <= a_enak_b_sig;

-- ali je a > b na mestu 0
f(0) <= c(7) and c(6) and c(5) and c(4) and c(3) and c(2) and c(1)
and a(0) and not b(0);
-- ali je a > b na mestu 1
f(1) <= c(7) and c(6) and c(5) and c(4) and c(3) and c(2) and a(1)
and not b(1);
-- ali je a > b na mestu 2
f(2) <= c(7) and c(6) and c(5) and c(4) and c(3) and a(2) and not
b(2);
-- ali je a > b na mestu 3
f(3) <= c(7) and c(6) and c(5) and c(4) and a(3) and not b(3);
-- ali je a > b na mestu 4
f(4) <= c(7) and c(6) and c(5) and a(4) and not b(4);
-- ali je a > b na mestu 5
f(5) <= c(7) and c(6) and a(5) and not b(5);
-- ali je a > b na mestu 6
f(6) <= c(7) and a(6) and not b(6);
-- ali je a > b na mestu 7
f(7) <= a(7) and not b(7);

--ali na kateremkoli mestu velja pogoj vecje?
a_vecji_od_b_sig <= f(0) or f(1) or f(2) or f(3) or f(4) or f(5) or
f(6) or f(7);

-- uvedemo vmesni signal (zico) a_vecji_od_b_sig, da se izognemo
tipu inout pri izhodu a_ vecji_b, saj bomo v pri izvedbi izhoda
a_manjsi_od_b signal a_vecji_od_b povezali tudi na vhod.
a_vecji_od_b <= a_vecji_od_b_sig;

--a je manjsi od b ce ni enak in ce ni vecji: Uporabimo signala
a_vecji_od_b_sig in a_enak_od_b_sig na vhodu izbiralnika 2/1 zato
smo ju uvedli prej
a_manjsi_od_b <= '1' when (a_vecji_od_b_sig = '0') and (a_enak_b_sig
= '0') else '0';
end arch;
```

22. Realizirajte kombinacijsko vezje, ki sortira pet 4−bitnih nepredznačenih števil po algoritmu sodo−lihih zamenjav (ang. odd−even transposition). Vezje na svojem vhodu sprejme 5 števil (`in1, in2, in3, in4, in5`) in jih na izhodu predstavi v naraščajočem zaporedju (`out1, out2, out3, out4, out5`).

Algoritem sodo−lihih zamenjav temelji na paralelizaciji bubble sort algoritma. Pri algoritmu "odd−even transposition" gre za primerjavo dveh sosednih vrednosti. Za sortiranje n števil po velikosti moramo izvesti n korakov. Katere sosednje vrednosti primerjamo, določa parnost koraka primerjave (lih, sod).
Osnovni blok primerjave predstavlja križno stikalo (ang. crossbar switch), ki ga krmili primerjalnik (ang. magnitude comparator). Blok ima dva vhoda (x_1, x_2) in dva izhoda (y_1 in y_2). Primerjalnik primerja dva vhoda med sabo ($x_1 > x_2$) in glede na to, če je rezultat primerjave res (true), potem zamenja vhoda ($y_1 = x_2$ in $y_2=x_1$), sicer je izhod enak vhodu($y_1=x_1$ in $y_2=x_2$).
Primer: Če je na prvem vhodu bloka število $x_1=3$, na drugem pa $x_2=2$, se bo na prvem izhodu bloka pojavila nespremenjena kombinacija (prvi izhod $y_1=3$, drugi izhod $y_2=2$). Če pa je na prvem vhodu število $x_1=2$, na drugem pa $x_2=3$, se bo na prvem izhodu pojavilo število $y_1=3$, na drugem pa $y_2=2$.



Primer urejanja: V poteku algoritma označimo zgoraj opisani blok s pravokotnikom, ki ima dva vhoda in dva izhoda. Oglejmo si potek primerjave petih 4−bitnih elementov.



Opisani algoritem lahko razširimo na poljubno število vhodnih vrednosti, le shemo moramo dodelati tako, da rabimo za *n* števil v vhodnem nizu *n* korakov primerjave. V smeri vrstic v danem koraku primerjamo vse signale razen enega (če jih je liho število sortiranih števil) in vse signale (če je sodo število sortiranih števil).
Bistvena prednost takega sortiranja števil je, da nima vmesnih zakasnitev (hitrost), saj ni zaporedna (oz. realizacija v času) ampak je kombinacijska (realizacija v prostoru) Celotna zakasnitev sortiranja je enaka zakasnitvi vrat, ki tvorijo vse primerjalne bloke na poti od vhoda do izhoda.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sort is
   Port ( in1, in2, in3, in4, in5 : in STD_LOGIC_VECTOR(3 downto 0);
      out1, out2, out3, out4, out5 : out STD_LOGIC_VECTOR(3 downto 0)
   );
end sort;

architecture arch of sort is
signal dat1s0, dat2s0, dat3s0, dat4s0, dat5s0, dat1s1, dat2s1,
dat3s1, dat4s1, dat5s1, dat1s2, dat2s2, dat3s2, dat4s2, dat5s2,
dat1s3, dat2s3, dat3s3, dat4s3, dat5s3, dat1s4, dat2s4, dat3s4,
dat4s4, dat5s4, dat1s5, dat2s5, dat3s5, dat4s5, dat5s5 :
STD_LOGIC_VECTOR(3 downto 0);

begin
dat1s0 <= in1;
dat2s0 <= in2;
dat3s0 <= in3;
dat4s0 <= in4;
dat5s0 <= in5;
dat1s1 <= dat2s0 when (dat1s0 < dat2s0) else dat1s0;
dat2s1 <= dat1s0 when (dat1s0 < dat2s0) else dat2s0;
dat3s1 <= dat4s0 when (dat3s0 < dat4s0) else dat3s0;
dat4s1 <= dat3s0 when (dat3s0 < dat4s0) else dat4s0;
dat5s1 <= dat5s0;
dat1s2 <= dat1s1;
dat2s2 <= dat3s1 when (dat2s1 < dat3s1) else dat2s1;
dat3s2 <= dat2s1 when (dat2s1 < dat3s1) else dat3s1;
dat4s2 <= dat5s1 when (dat4s1 < dat5s1) else dat4s1;
dat5s2 <= dat4s1 when (dat4s1 < dat5s1) else dat5s1;
dat1s3 <= dat2s2 when (dat1s2 < dat2s2) else dat1s2;
dat2s3 <= dat1s2 when (dat1s2 < dat2s2) else dat2s2;
dat3s3 <= dat4s2 when (dat3s2 < dat4s2) else dat3s2;
dat4s3 <= dat3s2 when (dat3s2 < dat4s2) else dat4s2;
dat5s3 <= dat5s2;
dat1s4 <= dat1s3;
dat2s4 <= dat3s3 when (dat2s3 < dat3s3) else dat2s3;
dat3s4 <= dat2s3 when (dat2s3 < dat3s3) else dat3s3;
dat4s4 <= dat5s3 when (dat4s3 < dat5s3) else dat4s3;
dat5s4 <= dat4s3 when (dat4s3 < dat5s3) else dat5s3;
dat1s5 <= dat2s4 when (dat1s4 < dat2s4) else dat1s4;
dat2s5 <= dat1s4 when (dat1s4 < dat2s4) else dat2s4;
dat3s5 <= dat4s4 when (dat3s4 < dat4s4) else dat3s4;
dat4s5 <= dat3s4 when (dat3s4 < dat4s4) else dat4s4;
dat5s5 <= dat5s4;

out1 <= dat1s5;
out2 <= dat2s5;
out3 <= dat3s5;
out4 <= dat4s5;
out5 <= dat5s5;

end arch;
```

23. Realizirajte 4−bitni kodirnik prioritete (ang. priority encoder). Kodirnik prioritete ima 4 vhode $w_0 ... w_3$. Vhod w_0 naj ima najnižjo prioriteto in vhod w_3 najvišjo prioriteto. Izhod kodirnika prioritete je dvojiška zaporedna številka tistega vhoda $y_1 y_0$, ki ima najvišjo prioriteto izmed vseh postavljenih. Izhod postane enak $z='0'$, ko noben vhod ni postavljen na '1'.
    Delovanje ilustrira spodnja tabela:

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

Analize zgornje tabele bi se lahko lotili tako, da bi narisali Veitchev diagram 4 spremenljivk za vse tri izhodne spremenljivke, vendar lahko s pravilnim razmišljanjem nalogo močno skrajšamo.
Kodirnik prioritete je namreč posebna izvedba kodirnika, ki temelji na prioriteti vhodnih signalov. V kodirniku prioritete ima vsak vhod določen svoj nivo prioritete. Izhod kodirnika prioritete je dvojiška zaporedna številka vhoda, ki ima najvišjo prioriteto. Ko je postavljen vhod z višjo prioriteto, so ostali z nižjo ignorirani. Če analiziramo tabelo delovanja, lahko zapišemo štiri funkcije, ki določajo kdaj je posamezen vhod postavljen na '1'.

$$\begin{aligned} i_0 &= \overline{w_3} \cdot \overline{w_2} \cdot \overline{w_1} \cdot w_0 \\ i_1 &= \overline{w_3} \cdot \overline{w_2} \cdot w_1 \\ i_2 &= \overline{w_3} \cdot w_2 \\ i_3 &= w_3 \end{aligned} \qquad (17.1)$$

Tabelo delovanja razširimo, da zapišemo še vmesne funkcije:

w_3	w_2	w_1	w_0	i_3	i_2	i_1	i_0	y_1	y_0	z
0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1	0	0	1
0	0	1	X	0	0	1	0	0	1	1
0	1	X	X	0	1	0	0	1	0	1
1	X	X	X	1	0	0	0	1	1	1

Iz zgornje tabele lahko povzamemo

$$\begin{aligned} y_0 &= i_1 + i_3 \\ y_1 &= i_2 + i_3 \\ z &= i_0 + i_1 + i_2 + i_3 \end{aligned} \qquad (17.2)$$

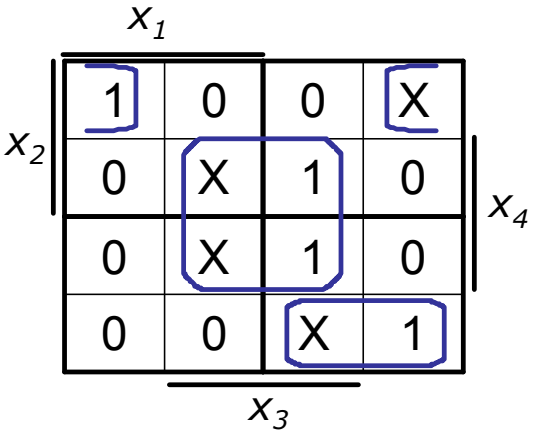
Podobno vezje 8−bitnega kodirnika prioritete je 74148[7], le logika vhodov in izhodov je negativna.
Opis delovanja in vezje 4 bitnega primerjalnika velikosti je v predlogah vaj na domači strani predmeta v imeniku Logisim\combinatorial\ priority_encoder_4_bit.circ

[7] http://www.alldatasheet.com/view.jsp?Searchword=74148

24. Realizirajte funkcijo $f^4 = \&(1,2,5,6,7,9,10,14)$ in redundantnimi makstermi $\&_x(0,4,11,13)$ s čim manj 4−bitnimi aritmetičnimi−logičnimi enotami (ALU). Negacije vhodnih spremenljivk izvedite z ALU.

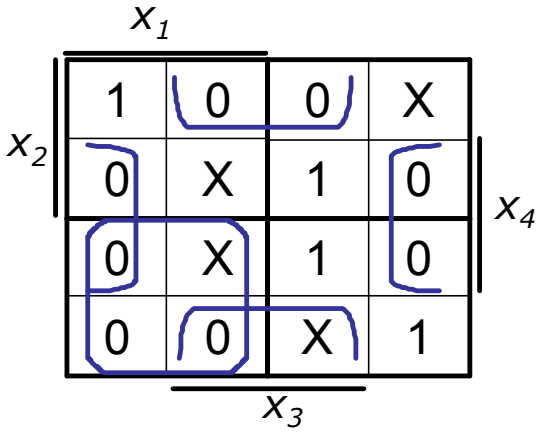
Funkcija f je podana v obliki PKNO z redundancami. Za potrebe realizacije jo najprej pretvorimo v obliko PDNO. To storimo tako, da maksterme preslikamo v minterme. V pravilnostno tabelo funkcije najprej zapišemo številke mintermov (m) in pripadajoče številke makstermov (M). Vpišemo f='0' za vse maksterme in f='X' za vse redundantne maksterme. Na preostala mesta vpišemo f='1' in preberemo pri katerih mintermih je f='1' oz. f='X' ter funkcijo izrazimo v obliki PDNO.

m	M	x_1	x_2	x_3	x_4	f
0	15	0	0	0	0	1
1	14	0	0	0	1	0
2	13	0	0	1	0	X
3	12	0	0	1	1	1
4	11	0	1	0	0	X
5	10	0	1	0	1	0
6	9	0	1	1	0	0
7	8	0	1	1	1	1
8	7	1	0	0	0	0
9	6	1	0	0	1	0
10	5	1	0	1	0	0
11	4	1	0	1	1	X
12	3	1	1	0	0	1
13	2	1	1	0	1	0
14	1	1	1	1	0	0
15	0	1	1	1	1	X



$$f_{MDNO} = x_3 \cdot x_4 + x_1 \cdot \overline{x_2} \cdot \overline{x_4} + x_2 \cdot \overline{x_3} \cdot \overline{x_4}$$

Podobno storimo še za MKNO.



Dobimo:

$$f = V(0,3,7,12) \text{ in } V_x(2,4,11,15)$$

Funkcijo minimiziramo, zapišemo v MDNO in MKNO ter poiščemo MNO.

$$\overline{f_{MKNO}} = x_3 \cdot \overline{x_4} + \overline{x_3} \cdot x_4 + x_1 \cdot \overline{x_2}$$

$$f_{MKNO} = \overline{x_3 \cdot \overline{x_4} + \overline{x_3} \cdot x_4 + x_1 \cdot \overline{x_2}}$$

$$f_{MKNO} = \left(\overline{x_3} + x_4\right) \cdot \left(x_3 + \overline{x_4}\right) \cdot \left(\overline{x_1} + x_2\right)$$

V obeh realizacijah preštejemo operatorje in predpostavimo, da so vsi operatorji samo dvovhodni. Od to sledi: MNO=MKNO. Aritmetično−logično enota lahko poleg aritmetičnih naenkrat realizira štiri dvovhodne logične operacije *istega tipa* (OR, AND, NOT, NOR, NAND, XOR, XNOR), zato nas zanima realizacija zgornje

funkcije z dvovhodnimi operatorji enega tipa. Pri realizaciji so zato primerne čimbolj nenormalne oblike (večnivojske oblike), samo da vsebujejo operatorje ene vrste. Podana funkcija je v MNO, zato za neposredno realizacijo s 4−bitno ALU ni primerna, saj vsebuje operaciji AND in OR − torej bi za realizacijo rabili najmanj dve aritmetični−logični enoti in tretjo za izvedbo inverterjev. Funkcijo MNO prevedemo na operator enega tipa − operator NOR, kar pomeni obliko PNO (Pierce−va normalna oblika funkcije):

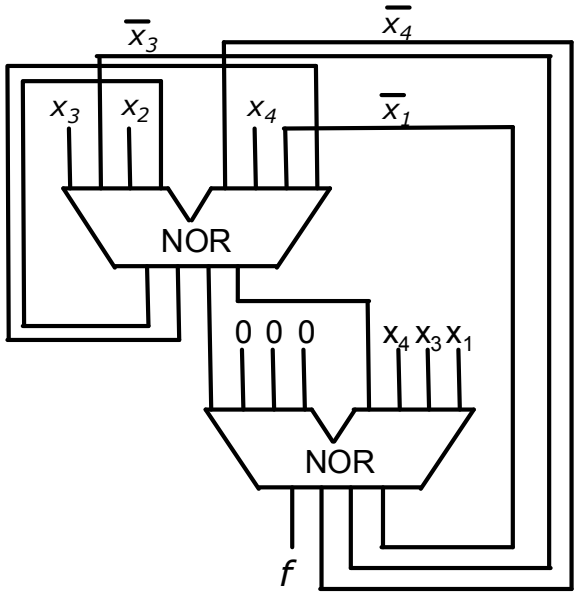
$$f_{PNO} = \overline{\overline{\left(\overline{x_3} + x_4\right) \cdot \left(x_3 + \overline{x_4}\right) \cdot \left(x_2 + \overline{x_1}\right)}}$$

$$f_{PNO} = \overline{\overline{\left(\overline{x_3} + x_4\right)} + \overline{\left(x_3 + \overline{x_4}\right)} + \overline{\left(x_2 + \overline{x_1}\right)}}$$

$$f_{PNO} = \left(\left(\overline{x_3} \downarrow x_4\right) \downarrow \left(x_3 \downarrow \overline{x_4}\right)\right) \downarrow \left(x_2 \downarrow \overline{x_1}\right)$$

V zadnji vrstici smo posebej izpostavili prva dva člena, zato da je bolj razvidno celotno število nivojev, ki jih mora ALU realizirati. Čeprav smo za izhodišče vzeli MNO, ki ima najmanjše število operatorjev, se tudi pri tej realizaciji ne moremo izogniti dvema ALU enotama, saj je celotno število nivojev 5. Preostale operatorje v drugi ALU lahko izkoristimo za tvorbo negacij. Upoštevamo lastnost NOR funkcije:

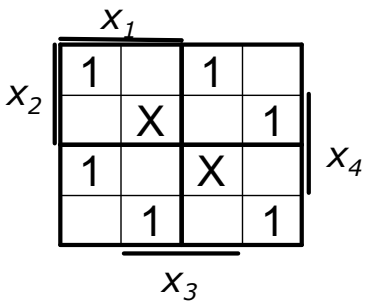
$$\overline{x} = \overline{0 + x} = 0 \downarrow x$$



25. Realizirajte podano funkcijo f z redundancami s čim manj 4−bitnimi aritmetičnimi−logičnimi enotami (ALU). Negacije vhodnih spremenljivk izvedite z ALU.

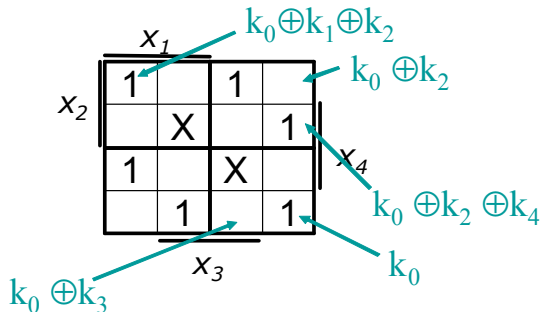
$$f(x_1,x_2,x_3,x_4) = V(0,5,6,9,10,12) \quad in \quad V_x(3,15)$$

Funkcijo najprej izrišemo v Veitch−ev diagram:



Funkcija vsebuje same diagonalne člene, zato realizacija v obliki KNO oz. DNO ne nudi minimalne oblike. Če se izkaže, da je funkcija linearna, jo lahko realiziramo s pomočjo XOR funkcij. Linearnost funkcije ugotavljamo tako, da prepogibamo kvadrate diagrama: Začnemo v desnem spodnjem kotu (kjer je minterm 0) in prepognemo kvadrat navzgor, da se spremeni samo ena spremenljivka naenkrat (**x4** postane 0 v prvi iteraciji).

Opazujemo, ali se prepogne na novi kvadrat čisto enako ali pa popolnoma negirano. Če postavimo obe redundanci na '1', lahko s prepogibanjem ugotovimo, da je funkcija linearna.



Podana funkcija je funkcija 4 spremenljivk, zato lahko njeno splošno izražavo kot linearno funkcijo pišemo kot:

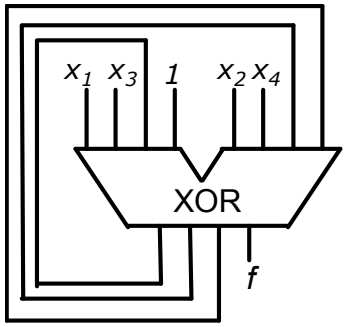
$$f(x_1,x_2,x_3,x_4) = k_0 \oplus k_1x_1 \oplus k_2x_2 \oplus k_3x_3 \oplus k_4x_4$$

S pomočjo Veitch−evega diagrama izračunamo koeficiente.
Iz enačb sledi: $k_0=1$ in $k_0 \oplus k_3=0$, kar pomeni $1 \oplus k_3 = 0 \rightarrow k_3=1$.
In če napišemo še enačbo za $k_0 \oplus k_2=0$, kar pomeni $1 \oplus k_2 = 0$ sledi da je $k_2=1$.
Iz enačbe $k_0 \oplus k_2 \oplus k_4 = 1$, kar pomeni $1 \oplus 1 \oplus k_4 = 1 \rightarrow k_4=1$.
Analiziramo naprej in dobimo $k_0 \oplus k_1 \oplus k_2=1$, kar pomeni $1 \oplus k_1 \oplus 1=0 \rightarrow k_1=1$.
Vstavimo dobljene koeficiente v enačbo za splošno izražavo in dobimo:

$$f(x_1,x_2,x_3,x_4) = 1 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4$$

Aritmetično−logično enota lahko poleg aritmetičnih naenkrat realizira štiri dvovhodne logične operacije *istega tipa* (OR, AND, NOT, NOR, NAND, XOR, XNOR), zato nas zanima realizacija zgornje funkcije z dvovhodnimi operatorji enega tipa. Pri realizaciji uporabimo lastnost združevanja, ki velja za XOR funkcijo.

$$f(x_1,x_2,x_3,x_4) = 1 \oplus \big((x_1 \oplus x_2) \oplus (x_3 \oplus x_4)\big)$$



26. Realizirajte hipotetično aritmetično logično enoto z izbiralniki 2/1, ki opravlja različne operacije nad enobitnima vhodoma A in B. ALU ima vhod za način delovanja M (ang. mode), s katerim izbiramo med logičnim in aritmetičnim načinom in dva vhoda S_1, S_0 za izbiro ustrezne funkcije. Izhod ALU je izbrana funkcija F.

M=0			M=1		
S_1	S_0	F	S_1	S_0	F
0	0	0	0	0	A’
0	1	A OR B	0	1	B’
1	0	A≡B	1	0	A minus B
1	1	A·B	1	1	A plus B

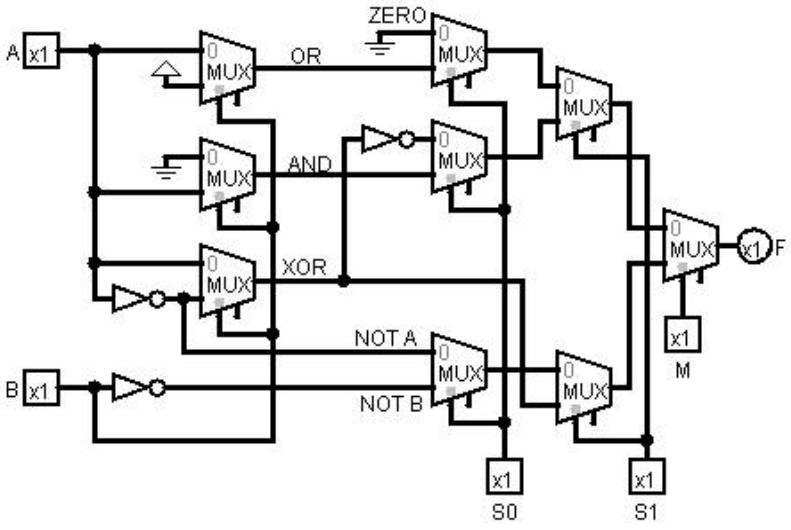
Najbolj neposredna možnost realizacije je risanje Veitch−evega diagrama za 5 spremenljivk (M, S_1, S_0, A, B), kar je zamudno. Namesto tega raje poskušajmo s kaskadno vezavo izbiralnikov realizirati posamezne operacije. Na najbolj pomembnem mestu kaskadne vezave izbiralnikov izberemo način delovanja ALU (vhod M). S tem razdelimo tabelo operacij na dva dela. Preglejmo funkcije, ki jih želimo realizirati in izpišimo tabelo operacij.

A	B	A plus B	A minus B	A OR B	A≡B	A AND B
0	0	0	0	0	1	0
0	1	1	1	1	0	0
1	0	1	1	1	0	0
1	1	0	0	1	1	1

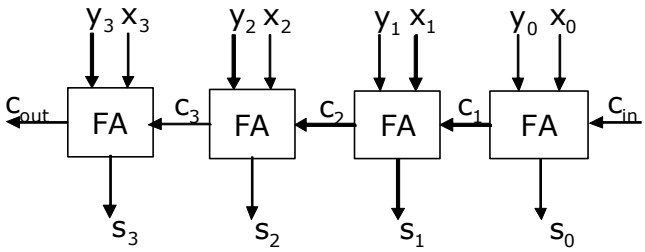
Iz zgornje tabele sledi, da so XOR, seštevanje in odštevanje enobitnih operandov ista operacija. Operacije XOR, AND ter OR realizirajmo z izbiralnikom 2/1. Če izberemo za naslovno spremenljivko B so funkcijski ostanki razvoja:

B	A⊕B	A OR B	A≡B	A AND B
0	A	A	A'	0
1	A'	1	A	A

Opis delovanja in vezje hipotetične ALU je v predlogah vaj na domači strani predmeta v imeniku Logisim\alu\hyp_alu_1_bit_1.circ:



27. V jeziku VHDL programirajte ripple−carry (RC) seštevalnik, ki sešteva dve 4−bitni števili *x* in *y*. Seštevalnik ima vhodni prenos c_{in}, 4−bitno vsoto *s* ter izhodni prenos c_{out}.



Vezje RC seštevalnika je veriga polnih seštevalnikov (FA), pri kateri je izhod FA prejšnjega mesta vezan na vhodni prenos naslednjega mesta, kot kaže zgornja slika. Za izvedbo seštevalnika moramo najprej realizirati komponentno enobitnega polnega seštevalnika (`fulladd`).

```
LIBRARY ieee;
USE ieee.std_logic_1164.all ;
ENTITY fulladd IS
   PORT ( Cin, x, y : IN STD_LOGIC;
          s, Cout: OUT  STD_LOGIC );
END fulladd;
ARCHITECTURE arch OF fulladd IS
BEGIN
   s <= x XOR y XOR Cin;
   Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y);
END arch;
```

Nastalo komponentno enobitnega polnega seštevalnika (`fulladd`) povežemo v verigo z uporabo povezovalnega stavka (`PORT MAP`). Neposredno realizacijo iz zgornje slike bi lahko zapisali takole:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY adder4 IS
   PORT ( x3, x2, x1, x0,
          y3, y2, y1, y0,
          Cin  : IN STD_LOGIC;  -- signali operandov x,y in vhodnega
prenosa
         s3, s2, s1, s0   : OUT STD_LOGIC; --signali vsote
         Cout      : OUT STD_LOGIC  --izhodni prenos
         );
END adder4;

ARCHITECTURE arch OF adder4 IS
   SIGNAL c1, c2, c3 : STD_LOGIC; --signali (zice) vmesnih prenosov
   COMPONENT fulladd
      PORT ( Cin, x, y : IN   STD_LOGIC ;
             s, Cout   : OUT  STD_LOGIC );
   END COMPONENT ;
BEGIN
   --primer polozajnega povezovanja v povezovalnem stavku
   stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 );
   stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2 );
   stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3 );
   --primer imenskega povezovanja v povezovalnem stavku
   stage3: fulladd PORT MAP ( Cin => c3, Cout => Cout, x => x3, y =>
y3, s => s3);
END arch;
```

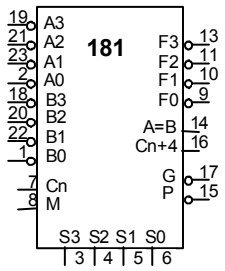
Prikazana sta dva primera uporabe povezovalnega stavka (`PORT MAP`): Prvi uporablja položajno povezovanje (*ang. positional association*), pri katerem v povezovalnem

stavku uporabljati enak vrstni red signalov, kot je naveden v deklaraciji komponente. Zadnji primer povezovalnega stavka uporablja imensko povezovanje (*ang. named association*), pri katerem vrstni red navajanja signalov ni pomemben. Imensko povezovanje je bolj prilagodljivo, zahteva pa veliko več pisanja.

Če bi želeli v VHDL realizirati N−bitni RC seštevalnik, bi povezovalni stavek komponente enobitnega polnega seštevalnika (`fulladd`) nadgradili z zanko (`FOR … GENERATE`), ki avtomatsko tvori zaporedje VHDL stavkov znotraj zanke glede na tekoči indeks (`i`). Seveda moramo za to spremeniti enobitne signale S, X, Y v N−bitne vektorske signale (`STD_LOGIC_VECTOR(N-1 downto 0)`) in v deklaraciji entitete definirati celoštevilski (`natural`) parameter N.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY rc_n_bit IS
   GENERIC( N : natural := 4 --stevilo mest sestevalnika
          );
   PORT ( Cin : IN STD_LOGIC;   --vhodni prenos
          X,Y : IN STD_LOGIC_VECTOR(N-1 downto 0); --operanda x,y
          S : OUT STD_LOGIC_VECTOR(N-1 downto 0); --vsota s
          Cout: OUT STD_LOGIC  --izhodni prenos
);
END rc_n_bit;

ARCHITECTURE arch OF rc_n_bit IS
   --signal za vmesne prenose RC sestevalnika
   SIGNAL c: STD_LOGIC_VECTOR(N downto 0);
   --deklaracija komponente FA
   COMPONENT fulladd
      PORT ( Cin, x, y : IN   STD_LOGIC;
             s, Cout   : OUT STD_LOGIC);
   END COMPONENT ;
BEGIN
   C(0) <= Cin;  --vhodni prenos

   ----------------------------------------------------------------
   --zaporedje spodnjih port map ukazov se da
   --zapisati krajse s for-generate zanko za n=4
   ----------------------------------------------------------------
   --stage0: fulladd PORT MAP ( C(0), x(0), y(0), s(0), c(1) );
   --stage1: fulladd PORT MAP ( C(1), x(1), y(1), s(1), c(2) );
   --stage2: fulladd PORT MAP ( C(2), x(2), y(2), s(2), c(3) );
   --stage3: fulladd PORT MAP ( C(3), x(3), y(3), s(3), c(4) );
   FOR i IN 0 TO N-1 GENERATE
     stages: fulladd PORT MAP (C(i), X(i), Y(i), S(i), C(i+1));
   END GENERATE;


   Cout <= C(N); --izhodni prenos je MSB vektorja c

END arch;
```

28. Realizirajte 32−bitni Carry Look Ahead (CLA) seštevalnik z uporabo 4−bitnih ALU 74181 in CLAG 74182.



ALU 74181 je 4−bitna aritmetično logična enota. Je kombinacijsko vezje, ki realizira eno izmed 48 aritmetičnih in logičnih operacij. Ima vhode za oba 4−bitna operanda: A[3:0], B[3:0], vhod za prenos C_n, izbiro tipa operacije S[3:0] in način delovanja: logični (M='1'), aritmetični (M='0'). Izhodi vezja so rezultat funkcije F[3:0], izhodni prenos C_{n+4}, funkciji širjenja in tvorjenja P, G ter izhod primerjalnika enakosti A==B. Oba vhoda operandov in rezultat operacije vezja so negirani, kar je zoprno za uporabo. Kako se izognemo uporabi inverterjev na vhodih in izhodih ALU?

- Pri logičnih funkcijah uporabimo dualno funkcijo:
  $f(x_1 \ldots x_n)=(f(x_1', x_2', x_3' \ldots, x_n'))'$
  - dualna funkcija AND↔OR
  - dualna funkcija XOR↔EQU.
- Pri aritmetičnih funkcijah
  negiramo C_n in C_{n+4}.
  - Pri seštevanju vzamemo
    S=1001 → C_n=1, ne C_n=0.
  - Izhodni prenos invertiramo

Operacije 74181 povzema spodnja tabela:

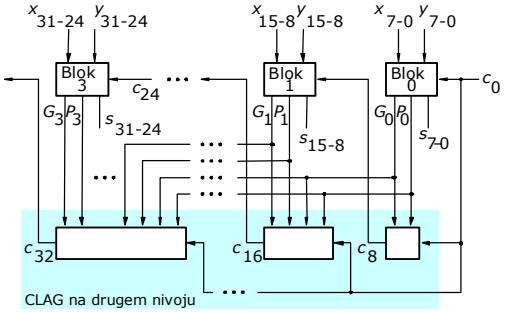
$S_3S_2S_1S_0$	M=1	M=0	
		C_n=0	C_n=1
0 0 0 0	F = A’	F = A minus 1	F = A
0 0 0 1	F = (AB)’	F = AB minus 1	F = AB
0 0 1 0	F = A’∨B	F = AB’ minus 1	F = AB’
0 0 1 1	F = 1	F = minus 1 (2’K)	F = 0
0 1 0 0	F=(A∨B)’	F=A plus (A∨B’)	F = A plus (A ∨ B’) plus 1
0 1 0 1	F = B’	F=AB plus (A∨B’)	F = AB plus (A ∨ B’) plus 1
0 1 1 0	F=A≡B	F=A minus B minus 1	F = A minus B
0 1 1 1	F=A∨B’	F = A ∨ B’	F = (A ∨ B’) plus 1
1 0 0 0	F = A’B	F=A plus (A∨B)	F = A plus (A ∨ B) plus 1
1 0 0 1	F=A⊕B	F = A plus B	F = A plus B plus 1
1 0 1 0	F = B	F=AB’ plus (A∨B)	F = AB plus (A ∨ B) plus 1
1 0 1 1	F = A∨B	F = A ∨ B	F = (A ∨ B) plus 1
1 1 0 0	F = 0	F = A plus A	F = A plus A plus 1
1 1 0 1	F = AB’	F = AB plus A	F = AB plus A plus 1
1 1 1 0	F = AB	F = AB’ plus A	F = AB’ plus A plus

1 1 1 1	F = A	F = A	F = A plus 1

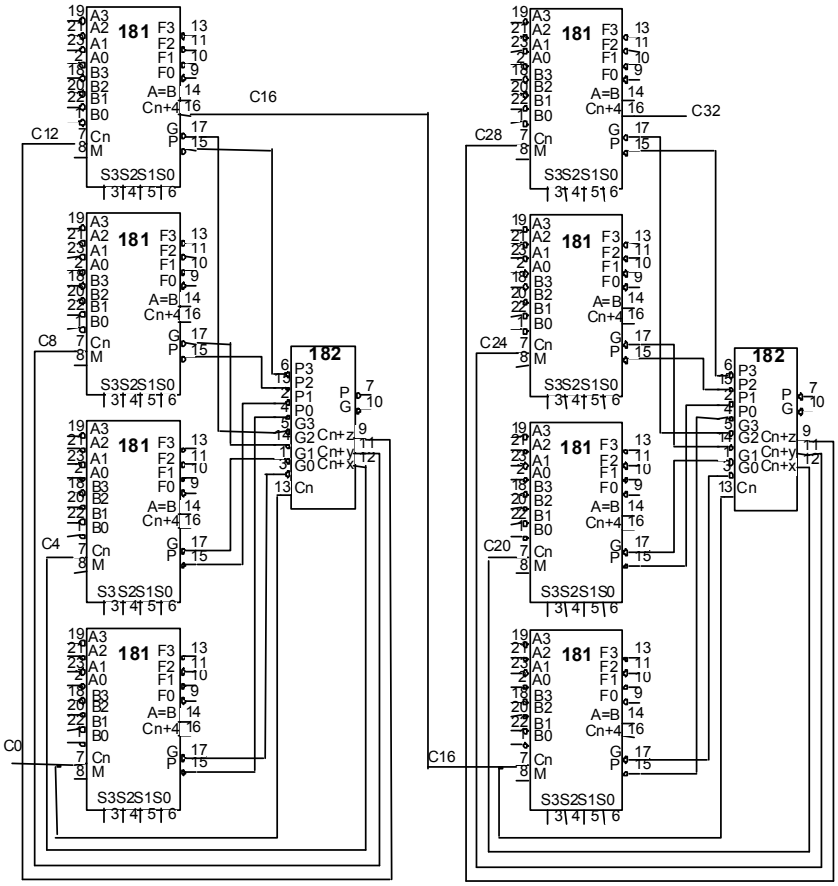
Zato, da realiziramo 32−bitni CLA seštevalnik, potrebujemo 8 vezij 74181. Vsako od teh vezij ima svoj izhodni prenos C_{n+4}, vendar ga ne bomo uporabljali, saj naloga zahteva realizacijo CLA seštevalnika. Namesto verižnega prenosa (RC) uporabimo raje P in G izhoda posamezne ALU in te funkcije vodimo na vhod CLAG 74182. Na izhodu CLAG se pojavijo izhodni prenosi za krmiljenje ostalih ALU na višjih mestih.

$$C_8 = G_0 + P_0 \cdot c_0$$
$$C_{16} = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot c_0$$
$$C_{24} = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot c_0$$
$$C_{32} = G_2 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0$$

CLAG izračuna 3 izhodne prenose (C_{n+x}, C_{n+y}, C_{n+z}), torej lahko s 4 ALU in enim CLAG tvorimo 16−bitni CLA seštevalnik. Za načrtovanje 32−bitnega CLA seštevalnika bomo torej povezali dva 16−bitna CLA seštevalnika. Izhodni prenos prvega 16−bitnega CLA seštevalnika (c_{16}) vodimo na vhodni prenos naslednjega 16−bitnega CLA seštevalnika in njegov CLAG. Blok shema nastale strukture je prikazana na spodnji sliki.



Končna realizacija 32−bitnega CLA seštevalnika je podana na spodnji sliki:



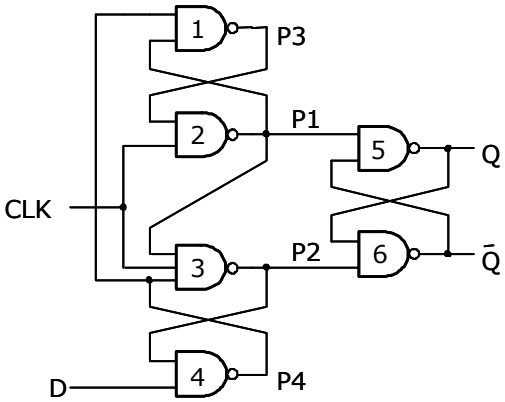
Isto nalogo lahko rešimo s pomočjo VHDL:
Osnovno celico CLA seštevalnika predstavlja polnega seštevalnika in blok funkcij tvorjenja in širjenja ()

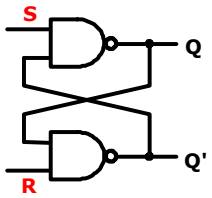
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY cla_gp IS PORT ( Cin, x, y : IN STD_LOGIC ;
                        s, Cout : OUT STD_LOGIC ) ;
END cla_gp;

ARCHITECTURE arch OF cla_gp IS
   SIGNAL g, p : STD_LOGIC ;
BEGIN
   g <= x and y;
   p <= x or y;
   s <= x xor y xor cin;
   Cout <= g or ( p and Cin );
END arch;

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY cla8 IS
PORT ( Cin : IN STD_LOGIC ;
     X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0) ;
     S : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) ;
     Cout, Overflow : OUT STD_LOGIC
     ) ;
END cla8 ;
```

Z uporabo povezovalnega stavka (`PORT MAP`) povežemo osnovno strukturo CLA seševalnika (`cla_gp`).

```
ARCHITECTURE arch OF cla8 IS
--vektor prenosov med stopnjami rc sestevalnika
SIGNAL C : STD_LOGIC_VECTOR(1 TO 7) ;
-- naslednja signala rabimo zato, ker bi sicer moral
-- cout signal biti tipa inout
-- in pa MSB bit vsote S(7) tudi tipa inout
-- v izrazu za racunanje overflow-a
SIGNAL Coutsignal, SMsb : STD_LOGIC;

BEGIN
   stage0: cla_gp PORT MAP ( Cin, X(0), Y(0), S(0), C(1) ) ;
   --namesto vmesnih stopenj zapisemo raje for ... generate zanko,
   --s katero realiziramo 14 port map stavkov.
   stageX: FOR i IN 1 TO 6 GENERATE
     stage1to6: cla_gp PORT MAP ( C(i), X(i), Y(i), S(i), C(i+1) ) ;
   END GENERATE;

   stage7: cla_gp PORT MAP ( C(7), X(7), Y(7), SMsb, Coutsignal);

   Cout <= Coutsignal;
   S(7) <= SMsb;

   Overflow <= Coutsignal XOR X(7) XOR Y(7) XOR SMsb;

END arch;
```

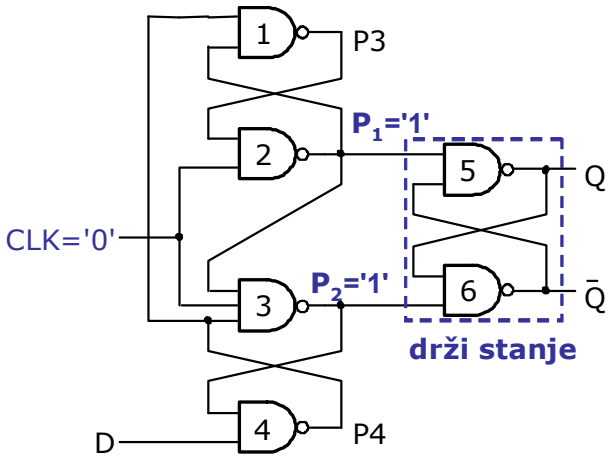
29. Razložite delovanje D flip-flopa, ki je prožen na pozitivni rob signala ure (*clk*). Za realizacijo vezja so uporabljena samo RS zapahi, sestavljeni iz NAND vrat.
Vezavo ilustrira spodnja slika. Pri razlagi uporabite številke NAND vrat in poimenovanja signalov na sliki.



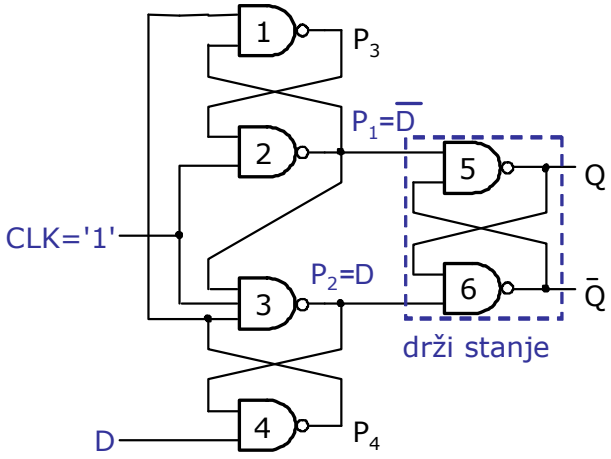


Vezje D flip-flopa, ki je prožen na pozitivni rob signala ure (*clk*) je sestavljeno iz treh RS zapahov (desni del slike). NAND 3 in 4 tvorita vhodni RS zapah, ki zapahne vrednost D, čim je signal ure '1'. Drugi RS zapah (iz vrat 1 in 2) drži to vrednost zapahnjeno dokler CLK vztraja na '1'. Kratko opišimo delovanje preko celotnega cikla signala ure:

Ko je CLK='0', sta izhoda NAND vrat 2 in 3 enaka '1'. Od tod sledi da je stanje P_1=P_2='1', kar drži izhodni zapah, ki je sestavljen iz vrat 5 in 6 v stanju ohranjanja vrednosti. Istočasno je P_3=D in P_4=D'.

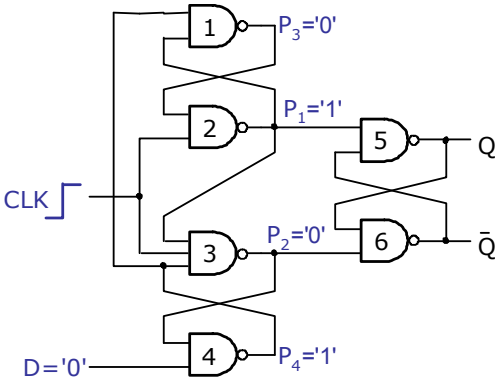


Ko postane CLK='1', se vrednosti P_3 in P_4 preneseta preko vrat 2 in 3, kar povzroči da je P_1=D' in P_2= D. To posledično postavi Q=D in Q'=D'.
Za zanesljivo delovanje morata biti P_3 in P_4 stabilna, ko se CLK spreminja iz '0' na '1'. Zato je čas vzpostavitve (*ang. setup time*) tega flip-flopa enak zakasnitvi iz vhoda D preko vrat 4 in 1 na P_3. Čas zadrževanja (*ang. hold time*) tega vezja je določen z zakasnitvijo preko vrat 3, saj ko je stabilen P_2, spremembe D niso več bistvene.
Pomembno je, da ko se CLK spremeni na '1', spremembe vhoda D ne vplivajo na stanje flip-flopa dokler je CLK='1'.

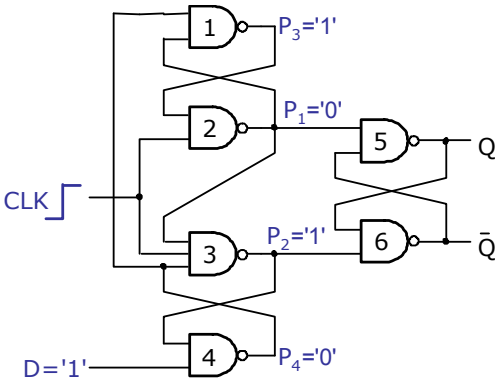


Opišimo še bolj podrobno vpis podatka (*D*) ob sprednjem robu signala ure:

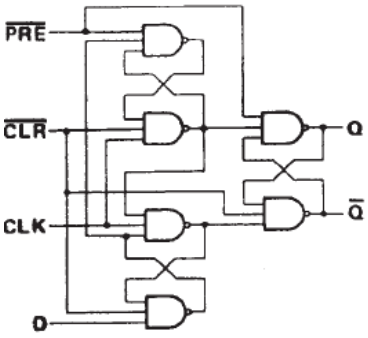
Če je *D='0'* ob pozitivnem robu signala ure je P_2='0', kar povzroči da je izhod NAND vrat 4 enak P_4='1' dokler je *CLK='1'*, ne glede na vrednost D. Izhod P_3='0', saj je na vhodu NAND vrat 1 logična '1'. Če je P_3='0', potem mu je izhod NAND vrat 2 komplementaren (negiran) − torej P_1='1'. Podobno sklepanje velja za izhodni FF (NAND vrata 5 in 6): Iz osnovnega delovanja RS zapaha z NAND vrati sledi, da je R='0', S='1', torej bo to SET izhodnega zapaha. Na vhodu je P_1='1', torej bo Q='0' in ker je P_2='0', bo Q'='0'.



Če je *D='1'* ob pozitivnem robu signala ure, je P_1='0', kar postavi izhod vrat 1 in 3 na P_2=P_3='1'. Iz osnovnega delovanja RS zapaha z NAND vrati sledi, da je R='1', S='0', torej bo to RESET izhodnega zapaha. Zato ta flip-flop ignorira spremembe D, ko je CLK enak '1'.



Če bi v predlaganem vezju zamenjali vsa NAND vrata z NOR vrati, bi dobili D−FF, ki je prožen na negativni rob signala ure (*clk*). Če strukturi iz NAND vrat dodamo še vhoda za asinhrono postavljanje (PRE') in brisanje (CLR') v negativni logiki, dobimo D−FF, kot je 7474[8].

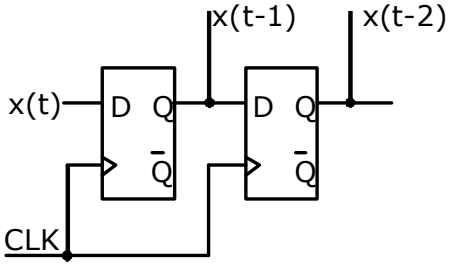


PRE'	*CLR'*	*CLK*	*D*	*Q*	*Q'*	*komentar*
0	1	X	X	1	0	asinhrono postavljanje Q='1'
1	0	X	X	0	1	asinhrono brisanje Q='0'
0	0	X	X	1 (_⌜)	1 (_⌜)	nedovoljena uporaba (Q in Q' nista komplementarna)
1	1	_⌜	H	1	0	SET Q='1'
1	1	_⌜	L	0	1	RESET Q='0'
1	1	0	X	Q_0	Q_0'	HOLD

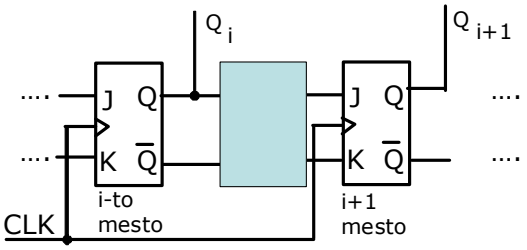
[8] http://www.alldatasheet.com/view.jsp?Searchword=SN7474DR

30. Z uporabo JK−FF in logičnih vrat realizirajte sekvenčno vezje z vhodom *x* in izhodom *y*, ki vrne y='1', kadar sta dva zaporedna vhoda enaka in različna od enega prej, torej x(t) = x(t-1) ≠ x(t-2). Sicer je y(t) = '0'.
V začetnem stanju pri t=0 predpostavite, da so vsi prejšnji vhodi x(-1), x(-2), ... enaki '0'.

Pomnjenje preteklih vrednosti vhoda *x* lahko izvedemo s pomočjo pomikalnega registra.

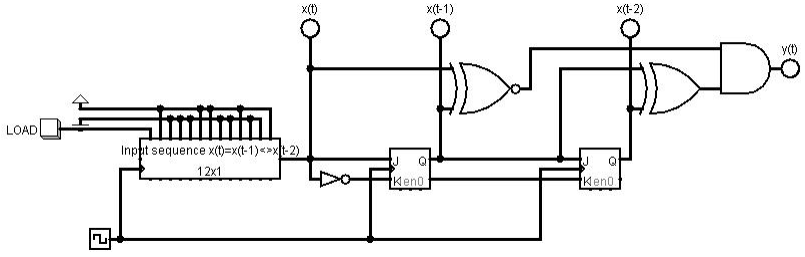


Zapomniti si moramo trenutno vrednost *x(t)* ter še dve prejšnji *x(t−1)* in *x(t−2)*. Za pomnjenje dveh preteklih vrednosti bomo načrtali 2−bitni pomikalni register. Zapišemo tabelo prehajanja stanj z *i*−tega mesta na (*i+1*) mesto. Enačba izhoda za (*i+1*) mesto pomikalnega registra je enačba D−FF ($Q_{i+1}(t+1)=D_{i+1}$), saj je pomikalni register običajno izveden kot veriga D−FF, pri katerih je izhod prejšnjega FF vezan na vhod naslednjega. Ena možnost realizacije je realizacija D−FF z uporabo JK−FF. Ta možnost je potratna, zato raje zapišemo vzbujalno tabelo za opazovano mesto in določimo vhode glede na spremembo stanja ($Q_{i+1}(t) \rightarrow Q_{i+1}(t+1)$) na (i+1) mestu.



Trenutna vsebina registra		*Vsebina registra ob pomiku*	*Vhoda FF na mestu i+1*		*Pomen stanja*
$Q_i(t)$	$Q_{i+1}(t)$	$Q_{i+1}(t+1)$	J_{i+1}	K_{i+1}	
0	0	0	0	X	RESET/HOLD
0	1	0	X	1	INVERT/RESET
1	0	1	1	X	INVERT/SET
1	1	1	X	0	SET/HOLD

Če v dobljenih vhodih za mesto (i+1) izberemo primerne redundance, dobimo vhoda $J_{i+1}= Q_i(t)$ in $K_{i+1}= Q_i(t)'$. To velja tudi za vhodno mesto, torej bomo na vhodni JK−FF vezali vhod $J_0= x(t)$ in $K_0= x(t)'$. Primerjavo neenakosti mest izvedemo z XOR vrati ($x(t) \oplus x(t-1)$), medtem ko primerjavo enakosti izvedemo z XNOR vrati ($x(t-1) \oplus x(t-2)$)'. Veljati morata oba pogoja, kar izvedemo z AND vrati.



Vezje generatorja je v predlogah avditornih vaj na domači strani predmeta:
Logisim\shift_reg\sequence_detector_x(t)_is_x(t_1)_is_not_x(t_2).circ

Seveda poleg take rešitve vedno obstaja klasični pristop reševanja s teorijo avtomatov, vendar smo pri tem precej omejeni. Čim bi morali primerjati dlje v zgodovino npr. x(t−4), x(t−5) … klasična teorija avtomatov hitro pripelje do minimizacije v Veitch−evih diagramih 5, 6 … spremenljivk, kar je dolgotrajen proces.

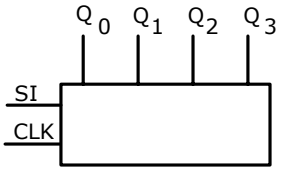
Vhod in trenutno stanje			*Naslednje stanje (ob pomiku)*		*Vhodi JK−FF*			
x	$Q_1(t)$	$Q_2(t)$	$Q_1(t+1)$	$Q_2(t+1)$	J_1	K_1	J_2	K_2
0	0	0	0	0	0	X	0	X
0	0	1	0	0	0	X	X	1
0	1	0	0	1	X	1	1	X
0	1	1	0	1	X	1	X	0
1	0	0	1	0	1	X	0	X
1	0	1	1	0	1	X	X	1
1	1	0	1	1	X	0	1	X
1	1	1	1	1	X	0	X	0

Če primerno izberemo redundance, niti ni potrebno risati Veitch−evih diagramov, ampak lahko dobimo enako rešitev kot pri realizaciji s pomočjo pomikalnega registra:

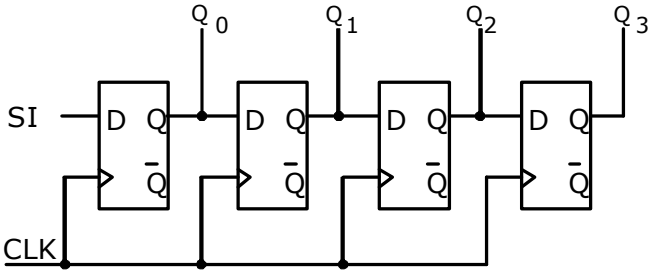
$$J_1 = x \qquad K_1 = \overline{x}$$
$$J_2 = Q_1(t) \quad K_2 = \overline{Q_1(t)}$$

Primerjavo na izhodu tako dobljenega registra (x, Q_1, Q_2) v splošnem izvedemo s primerjalnikom enakosti (ang. equality comparator), realiziranega z XOR oz. z XNOR vrati.

31. Sestavite 4−bitni pomikalni register s T-celicami in izbiralniki 2/1. Register ima zaporedni vhod *SI* (ang. serial input), in vzporedni izhod (Q_0, Q_1, Q_2, Q_3).



Zaporedno−vzporedni (SIPO) pomikalni register, realiziran s pomočjo D−FF, je veriga kaskadno vezanih D−FF, v kateri je izhod prejšnjega flip−flopa Q_{i-1} vezan na vhod naslednjega flip−flopa D_i.



Če želimo pomikalni register sestaviti iz T−FF in 2/1 izbiralnikov, moramo pravzaprav realizirati celico D−FF s pomočjo T−FF in 2/1 izbiralnikov. V ta namen zapišemo tabelo D−FF, pri kateri dodamo izhodni stolpec T vhoda.

D	*Q(t)*	*Q(t+1)*	*T*
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

Iz tabele sledi, da je *T* vhod XOR operacija Q(t) in vhoda D−FF, ki ga realiziramo.

$$Q(t+1) = Q(t) \oplus D$$

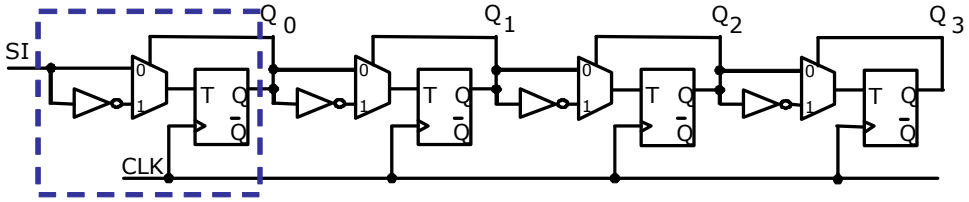
XOR vrata moramo realizirati s pomočjo 2/1 izbiralnikov, zato zapišemo enačbo XOR funkcije:

$$f = x \oplus y = \bar{x} \cdot y + x \cdot \bar{y}$$

Funkcijo *f* realiziramo z izbiralnikom tako, da naredimo razvoj po spremenljivki *x* in dobimo:

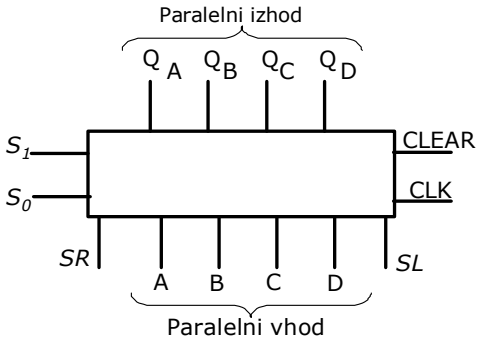
x	*f*
0	y
1	y'

Če nastali D−FF iz T−FF in 2/1 izbiralnika sestavimo skupaj v 4−bitni pomikalni register dobimo spodnjo realizacijo, v kateri je izvedba D−FF označena črtkano.



Opis delovanja in vezje pomikalnega registra je v predlogah avditornih vaj na domači strani predmeta: Logisim\shift_reg\shift_reg_4bit_using_tff_mux.circ

32. Z uporabo D flip−flopov, ki so proženi na pozitivni rob signala ure *CLK*, prikažite sintezo univerzalnega 4−bitnega pomikalnega registra, ki ima dva funkcijska vhoda S_0 in S_1 in opravlja funkcije po spodnji tabeli: Register ima tudi zaporedna vhoda za pomik v levo (SL − serial left), pomik v desno (SR − serial right) in asinhroni vhod za brisanje CLEAR (aktiven nizek).



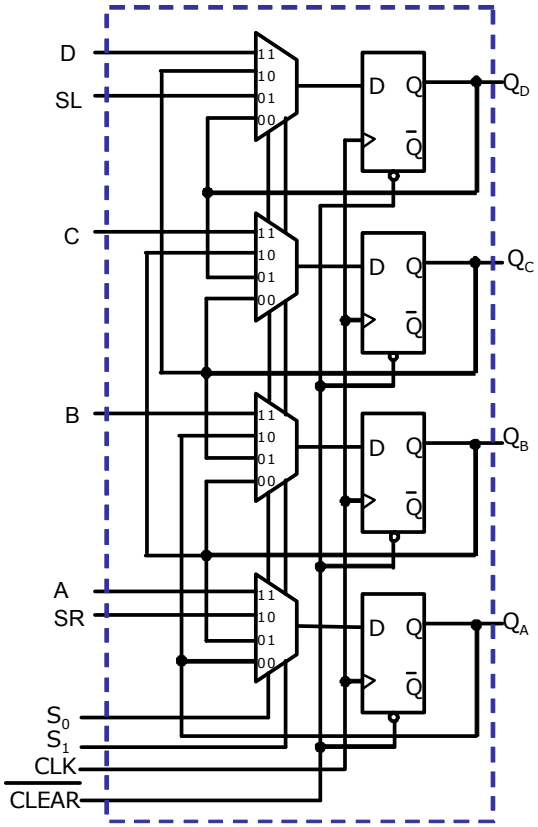
S_1	S_0	*funkcija*
0	0	drži stanje
0	1	pomik vsebine eno mesto desno
1	0	pomik vsebine eno mesto levo
1	1	nalaga vsebino z vhodov ABCD

Vsako od operacij izpišemo v pravilnostno tabelo v kateri združimo funkcijska bita S_1 in S_0 in trenutno stanje na i−tem mestu registra $Q_i(t)$. Mesta registra od leve proti desni so Q_i =(Q_A, Q_B, Q_C, Q_D) Realizacija z D flip−flopi nam analizo močno poenostavi, zaradi enačbe D flip−flopa: $D= Q(t+1)$.

Tabela 1: Prehajanje stanj univerzalnega registra.

S_1	S_0	$Q_i(t+1)$	*funkcija*
0	0	$Q_i(t)$	HOLD
0	1	$Q_{i+1}(t)$	SHR
1	0	$Q_{i-1}(t)$	SHL
1	1	x_i	LOAD

Iz poenostavljene tabele prehajanja stanj univerzalnega registra sestavimo realizacijo, ki bo vključevala izbiralnike MUX 4/1 in D−FF.
Na naslovna vhoda vseh MUX 4/1 vodimo funkcijska signala S_1 in S_0. Potem na vsakem podatkovnem vhodu realiziramo ustrezno funkcijo.



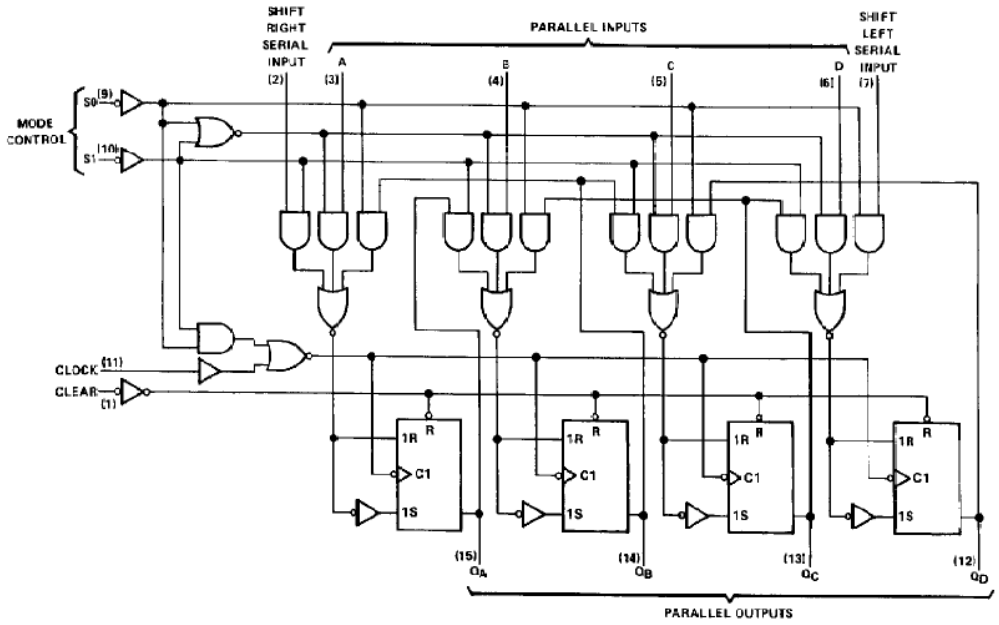
Stanje S_1S_0="00" pomeni držanje stanja (*HOLD*), torej bodo trenutne vrednosti D−FF ohranile vrednost $Q_i(t+1)= Q_i(t)$. Na sliki to realiziramo tako, da vodimo izhod D−FF nazaj na vhod pri podatkovnem vhodu 00. Stanje S_1S_0="01" pomeni pomik desno (*SHR* − ang. shift right), torej bodo D−FF pomaknili vsebino eno mesto desno. Pomik desno pomeni, da na mesto skrajno levega bita vpišemo vrednost zaporednega vhoda

SR, nato Q_A vodimo na vhod Q_B in tako do skrajno desnega bita. Stanje S_1S_0="10" pomeni pomik levo (*SHL*− ang. shift left), torej bodo D−FF pomaknili vsebino eno mesto levo. Pomik levo pomeni, da na mesto skrajno desnega bita vpišemo vrednost zaporednega vhoda SL, nato Q_D vodimo na vhod Q_C in tako do skrajno levega bita. S_1S_0="11" pomeni vzporedno nalaganje z vhodov (*LOAD*) $Q_D(t+1)=D$, $Q_C(t+1)=C$, $Q_B(t+1)=B$, $Q_A(t+1)=A$. Na tabeli smo i−ti vhod za vzporedno nalaganje označili kot $x_i=(A, B, C, D)$

Na spodnji sliki je prikazana shema pomikalnega registra 74194[9], izvedenega z D−FF, proženimi na negativni rob signala ure. Navzven je element prožen na pozitivni rob signala ure, saj je slednji voden preko NOR vrat. Dejanska izvedba je zelo podobna načelni realizaciji z MUX 4/1 in D−FF, vendar je izvedba precej enostavnejša (cenejša). D−FF so realizirani z RS−FF, pri katerih velja R=S'. Izbiralniki 4/1 so realizirani z AND in OR vrati. Stanje zadrževanja (*HOLD*) je realizirano tako, da je signal ure (*CLOCK*) voden preko ojačevalnika in AND vrat. Ko sta S_1S_0 ="00", bodo ta AND vrata signal ure držala na logični '0' in zato D−FF držijo vsebino. S tem se realizacija poenostavi. Za preostale funkcije registra si oglejmo vhod D_A prvega FF:

$$D_A = Q_B \cdot \overline{S_0} + A \cdot \overline{\overline{S_0} + \overline{S_1}} + SL \cdot \overline{S_1}$$
$$D_A = Q_B \cdot \overline{S_0} + A \cdot S_0 \cdot S_1 + SL \cdot \overline{S_1}$$

Prvi člen zgornje enačbe realizira pomikanje levo, drugi člen realizira vzporedno nalaganje in zadnji člen pomikanje desno. Negacija izhoda NOR vrat pri D_A odpade, ker z R=S' realiziramo D' vhod in ne D vhoda.



Slika 1: Univerzalni MSI 4−bitni pomikalni register 74194.

Tudi za ostale vhode FF lahko zapišemo:

$$D_B = Q_C \cdot \overline{S_0} + B \cdot S_0 \cdot S_1 + Q_A \cdot \overline{S_1}$$
$$D_C = Q_D \cdot \overline{S_0} + C \cdot S_0 \cdot S_1 + Q_B \cdot \overline{S_1}$$
$$D_D = SR \cdot \overline{S_0} + D \cdot S_0 \cdot S_1 + Q_C \cdot \overline{S_1}$$

Vezje ima še asinhroni signal za brisanje vsebine CLEAR, ki je vezan na signal za brisanje D−FF.

[9] http://www.alldatasheet.com/view.jsp?Searchword=74194

Isti register realizirajmo tudi v VHDL. Vsebino prikazujemo na izhodu (Q), vzporedno nalaganje vsebine realiziramo z vhodi (A, B, C, D). Spremembe prožimo na prednji rob signala ure (rising_edge(CLK)). Operacijo nastavljamo na dvobitnem vhodu (S). Ob brisanju (CLEAR) se vsebina registra postavi na (Q = "0000"), kar krajše zapišemo kot (Q = x"0").
Za operacije nad vsebino registra definiramo vmesni signal (Y), s katerim dosežemo enostavno branje in postavljanje ob pomikanju. Če vmesnega signala (Y) ne bi uporabljali, ampak bi operacije pomikanja tvorili neposredno z izhodom (Q), bi moral biti signal (Q) tipa (inout), čemur se izogibamo.

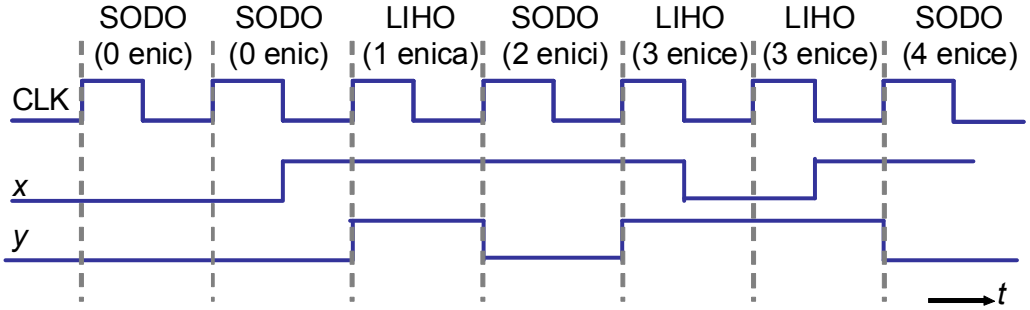
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity unireg is
   Port ( CLEAR, CLK, SR, SL, A, B, C, D : in STD_LOGIC;
          S : in STD_LOGIC_VECTOR (1 downto 0);
          Q : out STD_LOGIC_VECTOR (3 downto 0)
        );
end unireg;

architecture arch of unireg is
signal Y :  STD_LOGIC_VECTOR (3 downto 0) := x"0";
--Y je vmesni signal registra
begin
PROCESS (CLK, CLEAR, S)
   BEGIN
      IF CLEAR = '1' THEN
         Q <= x"0";  --brisanje registra
      ELSIF rising_edge( CLK ) THEN
         case S is
            when "00" => Y <= Y; -- drzi vsebino
            when "01" => Y <= SR & Q(3 downto 1); --pomik desno
            when "10" => Y <= Y(2 downto 0) & SL; --pomik levo
            when others => Y <=  D & C & B & A; --nalaganje vsebine
         end case;
      END IF;
   END PROCESS;
   Q <= Y;-- povezi vsebino registra na izhod
end arch;
```

33. V VHDL programirajte 8−bitni pomikalni register ki bo pomikal vsebino "00000001" v levo, dokler '1' ne pride do MSB položaja. Takrat se začne vsebina pomikati proti LSB položaju. Ko pride do LSB položaja se postopek pomikanja ponovi v obratni smeri. Trenutna smer pomikanja naj bo zapisana v spremenljivki (`dir`). Vsebino pomikanja naj si vezje zapomni v spremenljivki (`Y`). Vsebino registra prikazujte na izhodu (`Q`). Ob ponastavitvi (`rst`) naj se v register naloži začetna vrednost pomikanja (`"00000001"`).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity knight_rider is
   Port ( clk, rst : in STD_LOGIC;
          Q : out  STD_LOGIC_VECTOR(7 DOWNTO 0)
          );
end knight_rider;

architecture arch of knight_rider is
signal Y : STD_LOGIC_VECTOR(7 DOWNTO 0);
signal dir : STD_LOGIC;
begin

process( rst, dir, clk )
begin
   if rst = '1' then
      Y <= "00000001"; --nalozi zacetno vsebino registra
      dir <= '0'; --postavi zacetno smer pomikanja
   elsif rising_edge( clk ) then
      if ( dir = '0' ) then
         Y <= Y( 6 downto 0 ) & Y(7); -- pomakni levo
         if Y = "01000000" then  -- ce je skrajno levo
            dir <= '1'; -- obrni smer
         end if;
      else
         Y <= Y(0) & Y( 7 downto 1 ); -- pomakni desno
         if Y = "00000010" then -- ce je skrajno desno
            dir <= '0'; -- obrni smer
         end if;
      end if;
   end if;
end process;
Q <= Y;   -- povezi vmesno spremenljivko na izhod
end arch;
```

34. Realizirajte generator lihe parnosti (paritete) kot avtomat končnih stanj, ki šteje število enic v serijskem zaporedju bitov *x* na vhodu: Izhod vezja *y* naj bo '1', ko je na vhodu liho število enic in '0' ko je na vhodu sodo število enic. Ob resetu avtomata je število enic na vhodu sodo (nič enic). Za realizacijo uporabite D flip−flope, prožene na sprednji rob signala ure *CLK*.

Primer delovanja generatorja parnosti povzema spodnja slika:



Postopek sinteze zahteva, da realiziramo avtomat končnih stanj Moore−ove izvedbe:

Najprej bomo razvili diagram prehajanja stanj, ki opisuje delovanje vezja za liho preverjanje parnosti. Vezje je lahko v enem od dveh stanj: v sekvenci je bilo do tega trenutka liho ali sodo število enic. Kadar je na vhodu 1, je potrebno preklopiti v drugo stanje. Na primer, če je bilo do tega trenutka prisotnih liho ·število enic in je trenutni vhod 1, potem bomo imeli sedaj sodo ·število enic. Če pa bo na vhodu 0, ostane v istem stanju. Narisani diagram prehajanja stanj ima dve stanji, ki označujeta trenutno število enic na vhodu − torej LIHO in SODO. Izhod zapišemo pod stanjem (LIHO='1', SODO='0'). Vrednosti na vhodu x povzročajo spreminjanje stanj, ki so označene z usmerjenimi povezavami. Če je se na vhodu pojavi '0' (ne glede na to v katerem stanju smo) ostanemo v tem stanju: Jasno − saj štejemo samo '1'. Če smo v stanju LIHO in se na vhodu pojavi '1', preidemo v SODO. Če smo v stanju SODO in se na vhodu pojavi '1', preidemo v LIHO. Povedano povzema spodnji diagram prehajanja stanj

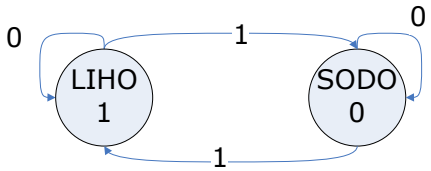


Diagram prehajanja stanj opišemo s tabelo prehajanja stanj:

vhod x	trenutno stanje Q(t)	naslednje stanje Q(t+1)
0	SODO	SODO
0	LIHO	LIHO
1	SODO	LIHO
1	LIHO	SODO

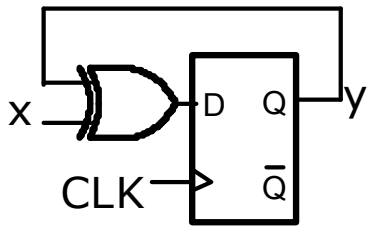
Če stanja kodiramo glede na njihov izhod LIHO '1' in SODO '0', potem dobimo novo aplikacijsko tabelo prehajanja stanj.

x	Q(t)	Q(t+1)	D	y
0	0	0	0	0
0	1	1	1	0
1	0	1	1	1
1	1	0	0	1

Iz tabele prehajanja stanj avtomata določimo enačbo za D−FF. Potrebno število FF je 1, saj sta stanji samo dve. Iz aplikacijske tabele sledi:

$$D = Q(t+1)$$
$$Q(t+1) = x \cdot \overline{Q(t)} + \bar{x} \cdot Q(t) = x \oplus Q(t)$$
$$y = Q(t)$$

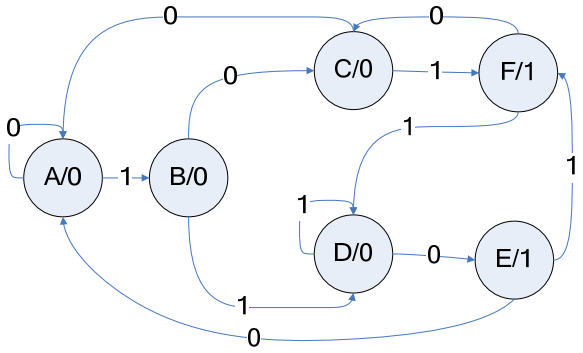
Izvedba vezja je:



Realizirali smo T−FF, prožen na sprednji rob signala ure.
Delovanje vezja si lahko ogledate v predlogah Logisim na domači strani predmeta:
Logisim\ff\T_ff_using_D_ff_and_xor.circ

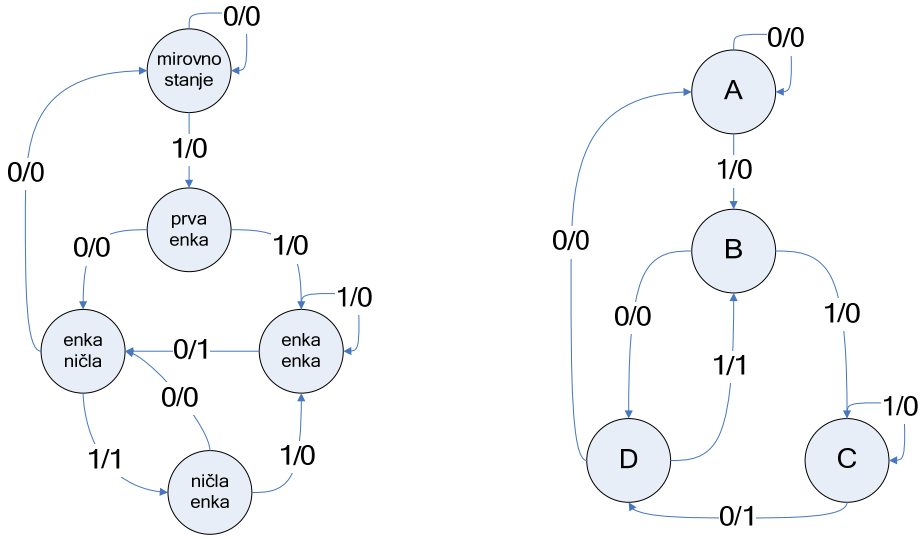
35. Narišite diagram stanj za avtomat končnih stanj, ki ima vhod w in izhod z. Avtomat končnih stanj postavi izhod z='1', ko se na vhodu pojavi zaporedje "110" ali "101", sicer je z='0'. Prekrivanje vzorcev je dovoljeno.
Delovanje avtomata končnih stanj povzema spodnje časovno zaporedje vhoda in izhoda.

w	0	1	1	0	1	1	0	1	1	1	0	0	0
z	–	0	0	0	**1**	**1**	0	**1**	**1**	0	0	**1**	0



Pomen stanj v zgornjem avtomatu označimo kot:
A → mirovno stanje oz. prej je bila ničla na vhodu
B → "1" na vhodu
C → kombinacija "10" na vhodu
D → kombinacija "11" na vhodu
E → kombinacija "101" na vhodu
F → kombinacija "110" na vhodu

Narišemo diagram še za Mealy−evo izvedbo:



Slika 2: Avtomat končnih stanj Moore−ove izvedbe (zgoraj) in Mealy−eve izvedbe (spodaj).

Stanju "ničla enka" se lahko izognemo, saj je to stanje ekvivalentno stanju "prva enka" in dobimo izvedbo avtomata, prikazano na desni strani.

36. Realizirajte avtomat končnih stanj, ki ima vhod w in izhod z. Avtomat končnih stanj postavi izhod z=1 ko se na vhodu pojavi zaporedje "1001" ali "1111", sicer je z='0'. Prekrivanje vzorcev je dovoljeno.
Delovanje avtomata končnih stanj povzema spodnje časovno zaporedje vhoda in izhoda.

w	0	1	0	1	1	1	1	0	0	1	1	0	0	1	1	1	1	1
z	–	0	0	0	0	0	0	**1**	0	0	**1**	0	0	0	**1**	0	0	**1**

Prikazali bomo izvedbo avtomata končnih stanj Moore−ove izvedbe.

Diagram prehajanja stanj:

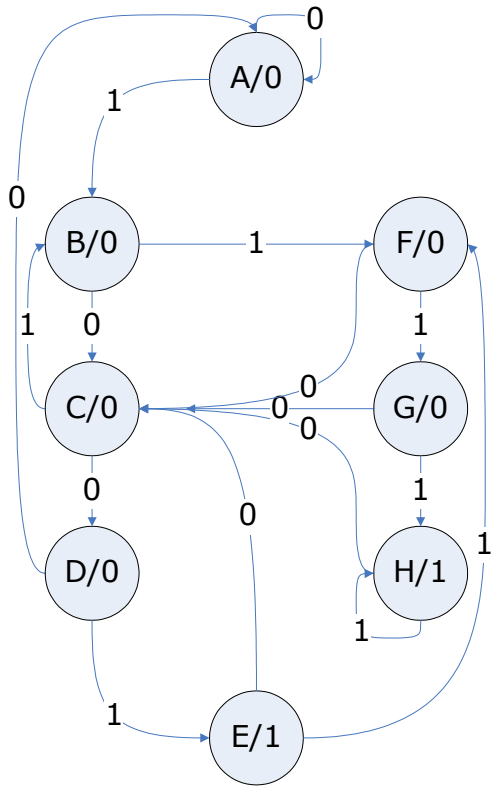


Diagram stanj začnemo risati tako, da najprej narišemo začetno stanje (A) v katerega se vračamo vedno, kadar sekvenca ne bo podobna tisti, ki jo zaznavamo (1111 oz. 1001). V tem stanju vztrajamo toliko časa, dokler je na vhodu '0', saj se nobena od sekvenc ne začenja z '0'. Ko pride na vhod prva '1', preidemo v drugo stanje (B), v katerem imamo dve možnosti, saj se sekvenci razlikujeta na drugem mestu: Če na vhod tega stanja pride '0', bomo zaznavali sekvence tipa '10XX', če pa pride '1', potem pa sekvence tipa '11XX'. Recimo, da v stanju B na vhod pride '0', torej napredujemo v stanje C. V tem stanju se ponovno lahko pojavi '0' − torej bi bila na vhodu sekvenca tipa '100X', kar bi nas vodilo do naslednjega stanja D. Če pa se pojavi na vhodu logična '1' je sicer sekvenca napačna, vendar moramo to predstaviti tako kot da je na vhod prišla že prva '1' od sekvence - naloga namreč pravi, da se sekvence lahko med seboj prekrivajo. S podobnim načinom razmišljanja narišemo naslednje stanje E, v katero napredujemo iz D samo, ko vanj pride '1', kar pomeni da smo v tem stanju zaznali sekvenco "1001", zato je v tem stanju izhod vezja enak '1'. Stanja F, G in H služijo pomnjenju koliko enic je prišlo na vhod vezja in sicer: stanje F pomenita "11" na vhodu vezja, stanje G tri enice in zadnje stanje H četrto enico sekvence. Slednje stanje vztraja, dokler je na vhodu '1', saj tako rešimo prekrivanje vzorca '1111'. Če pa v katerem koli od stanj F, G in H pride na vhod '0', se postavimo v stanje C, saj to stanje pomeni, da je na vhodu sekvenca '10XX'.

Tabela prehajanja stanj:

trenutno stanje		**naslednje stanje**		**izhod**
pomen stanja	koda stanja	w=0	w=1	z
začetno stanje	A	A	B	0
prva enica '1001' ali '1111' na vhodu	B	C	F	0
prva ničla '1001' na vhodu	C	D	B	0
druga ničla '1001' na vhodu	D	A	E	0
druga enica '1001' na vhodu	E	C	F	1
druga enica '1111' na vhodu	F	C	G	0
tretja enica '1111' na vhodu	G	C	H	0
četrta enica '1111' na vhodu	H	C	H	1

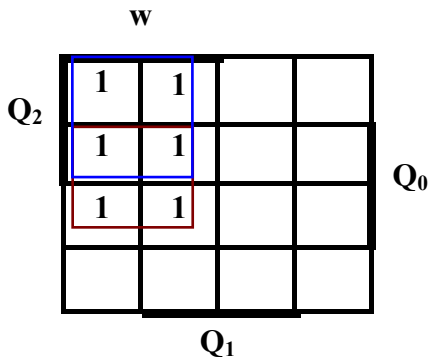
Za izvedbo bomo rabili najmanj 3 FF, saj je stanj 8. Izberemo zaporedno kodiranje stanj se pravi A=000, B=001, C=010, D=011, E=100, F=101, G=110, H=111.

Vzbujalna tabela:

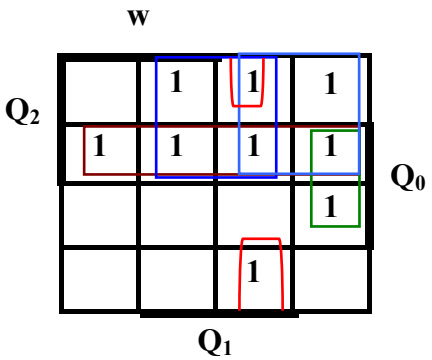
trenutno stanje				naslednje stanje						izhod
				w=0			w=1			
	$Q_2(t)$	$Q_1(t)$	$Q_0(t)$	$Q_2(t+1)$	$Q_1(t+1)$	$Q_0(t+1)$	$Q_2(t+1)$	$Q_1(t+1)$	$Q_0(t+1)$	z
A	0	0	0	0	0	0	0	0	1	0
B	0	0	1	0	1	0	1	0	1	0
C	0	1	0	0	1	1	0	0	1	0
D	0	1	1	0	0	0	1	0	0	0
E	1	0	0	0	1	0	1	0	1	1
F	1	0	1	0	1	0	1	1	0	0
G	1	1	0	0	1	0	1	1	1	0
H	1	1	1	0	1	0	1	1	1	1

Iz vzbujalne tabele sestavimo tri Veitcheve diagrame, pri katerih sestavljamo funkcije:

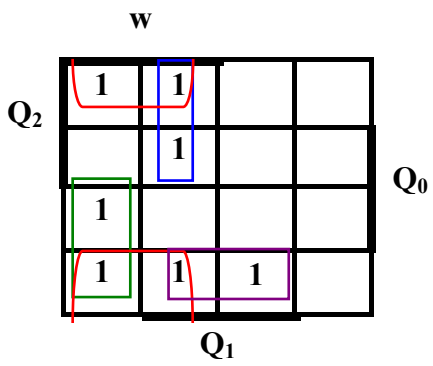
$Q_2(t+1)$:



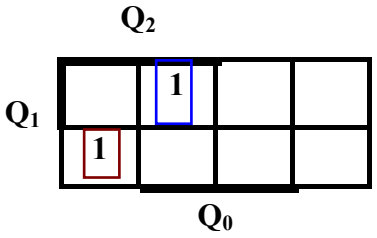
$Q_1(t+1)$:



$Q_0(t+1)$:



z:



Funkcije v Veitchevih diagramih minimiziramo in dobimo enačbe za realizacijo s pomočjo D flip−flopov. Realizacija s pomočjo D flip−flopov je najbolj enostavna, saj je enačba D flip−flopa:

$$D = Q(t+1) \tag{29.1}$$

kar pomeni, da lahko iz minimizacije Veitchevih diagramov naslednjih stanj zapišemo enačbe za vhode D flip−flopov:

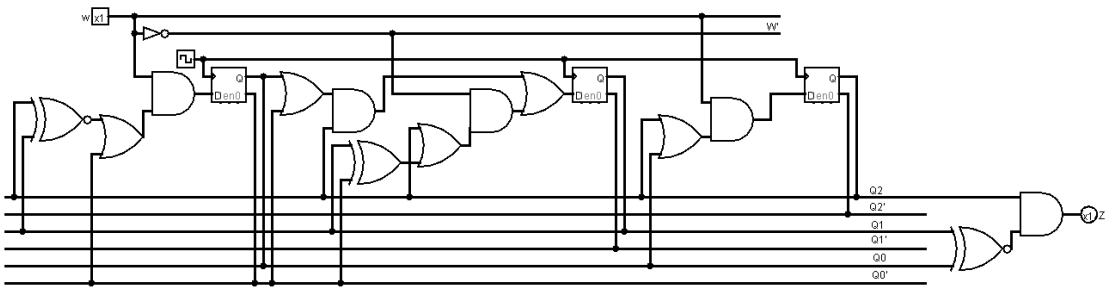
$$\begin{aligned} D_2 &= w \cdot Q_2(t) + w \cdot Q_0(t) = w \cdot (Q_2 + Q_0) \\ D_1 &= Q_2(t) \cdot (Q_1(t) + Q_0(t)) + \overline{w} \cdot \left(Q_2(t) + \overline{Q_1(t)} \cdot Q_0(t) + Q_1(t) \cdot \overline{Q_0(t)}\right) = \\ D_1 &= Q_2 \cdot (Q_1 + Q_0) + \overline{w} \cdot (Q_2 + Q_1 \oplus Q_0) \\ D_0 &= w \cdot \overline{Q_0(t)} + w \cdot \overline{Q_2(t)} \cdot \overline{Q_1(t)} + w \cdot Q_2(t) \cdot Q_1(t) + \overline{Q_1(t)} \cdot Q_1(t) \cdot \overline{Q_0(t)} = \\ D_0 &= w \cdot \left(\overline{Q_0} + \overline{Q_2 \oplus Q_1}\right) \end{aligned} \tag{29.2}$$

Izhod z lahko zapišemo kot:

$$z = Q_2(t) \cdot \left(Q_1(t) \cdot Q_0(t) + \overline{Q_1(t)} \cdot \overline{Q_0(t)}\right) = Q_2 \cdot \overline{Q_1 \oplus Q_0} \tag{29.3}$$

Vezje avtomata narišemo iz enačb (29.2) in (29.3).

Opis delovanja in vezje avtomata, ki primerja enakost znotraj 4 period signala ure je v predlogah vaj na domači strani predmeta v imeniku Logisim\fsm\ moore_1001_1111.circ



37. Z uporabo D spominskih celic, proženih na sprednji rob signala ure, načrtajte Mealy−ev avtomat, ki postavi izhod y_1='1', če se na vhodu *x* pojavi zaporedje "101". To velja, dokler se na vhodu *x* ni pojavilo zaporedje "011". Ko avtomat zazna zaporedje "011", do izklopa napajanja postavi izhod y_2 ='1'. V vseh ostalih primerih sta izhoda enaka '0'. Velja tudi $y_1 \cdot y_2$='0'. Prekrivanje zaporedij "101" je možno. Delovanje avtomata povzema spodnje časovno zaporedje.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
x	1	0	1	0	1	1	0	1	0	0	1	1	0
y_1	0	0	**1**	0	**1**	0	0	0	0	0	0	0	0
y_2	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Prikazali bomo izvedbo avtomata končnih stanj Mealy−eve izvedbe. Avtomat se ob vklopu napajanja nahaja v stanju "start". V tem stanju ne vztraja, ampak preide v stanje "prva ničla", če se na izhodu pojavi '0', oz. "prva enica", če se na vhodu pojavi '1'. Z uvedbo posebnega stanja smo zagotovili, da se je na vhodu zagotovo pojavil prvi bit zaznavanega zaporedja. Če se v stanju "prva ničla" pojavi '1', preide v stanje "enica" od koder sta dve možnosti: Če se na vhodu pojavi '0', preide v stanje "ničla", kjer se lahko vhodno zaporedje "101" zaključi tako, da ponovno preide v stanje "enica", ob tem da postavi izhod y_1='1'. Ker sta zaporedji izbrani tako, da se prekrivata na dveh mestih ("101" in "011") bo druga možnost stanje "stop". Prekrivanje zaporedja "101" je možno, saj avtomat prehaja med stanjema "enica" in "ničla". Če se v stanju "enica" pojavi še ena '1', to pomeni, da je bilo na vhodu zaporedje "011" in avtomat preide v stanje "stop" od koder ni vrnitve (x = X), saj po besedilu naloge takrat izhod y_2='1' vztraja v nedogled, medtem ko mora biti zaradi zahteve $y_1 \cdot y_2$='0' izhod y_1='0'. Če se v stanju "prva enica" pojavi zaporedje "011", avtomat podobno preide v stanje "stop" preko stanj "ničla" in nato "enica".

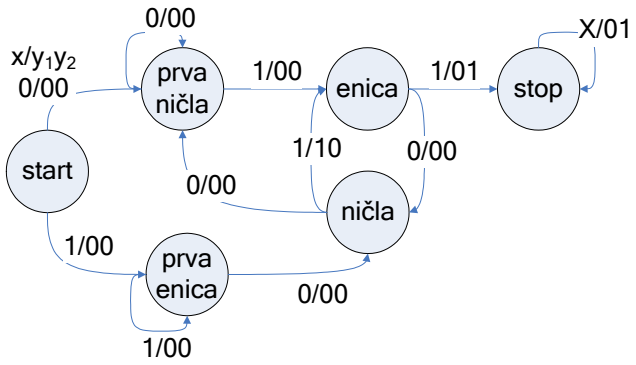


Diagram prehajanja stanj izpišemo v tabelo prehajanja stanj:

Trenutno stanje	Naslednje stanje	
	x=0	x=1
"start"	"prva ničla"/00	"prva enica"/00
"prva ničla"	" prva ničla"/00	"enica"/00
"prva enica"	"ničla"/00	"prva enica"/00
"enica"	"ničla "/00	"stop"/01
"ničla"	"prva ničla"/00	"enica"/10
"stop"	"stop"/01	"stop "/01

Stanj avtomata je 6, torej bomo za njihovo kodiranje potrebovali tri flip−flope. Za analizo avtomata izberemo kodiranje stanj:

Stanje	$Q_2(t)$	$Q_1(t)$	$Q_0(t)$
"start"	0	0	0
"prva ničla"	0	0	1
"prva enica"	0	1	0
"enica"	0	1	1
"ničla"	1	0	0
"stop"	1	0	1

Predlagano kodiranje uporabimo nad tabelo prehajanja stanj in izhod zapišemo ločeno glede na vhod x=0 oz. x=1.

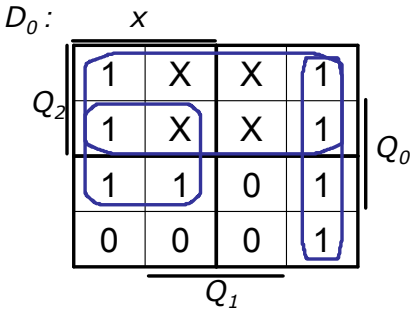
Trenutno stanje			Naslednje stanje						y_1		y_2	
			x=0			x=1						
$Q_2(t)$	**$Q_1(t)$**	**$Q_0(t)$**	**$Q_2(t+1)$**	**$Q_1(t+1)$**	**$Q_0(t+1)$**	**$Q_2(t+1)$**	**$Q_1(t+1)$**	**$Q_0(t+1)$**	**x=0**	**x=1**	**x=0**	**x=1**
0	0	0	0	0	1	0	1	0	0	0	0	0
0	0	1	0	0	1	0	1	1	0	0	0	0
0	1	0	1	0	0	0	1	0	0	0	0	0
0	1	1	1	0	0	1	0	1	0	0	0	1
1	0	0	0	0	1	0	1	1	0	1	0	0
1	0	1	1	0	1	1	0	1	0	0	1	1
1	1	0	X	X	X	X	X	X	X	X	X	X
1	1	1	X	X	X	X	X	X	X	X	X	X

Dobljeno tabelo razdelimo na dva dela, saj bomo izhoda y_1y_2 obravnavali posebej. Najprej zapišemo samo aplikacijsko tabelo za enačbe vhodov D−FF pri čemer za vsak FF upoštevamo enačbo $Q_i(t+1)=D_i$:

Trenutno stanje			Naslednje stanje					
			x=0			x=1		
$Q_2(t)$	**$Q_1(t)$**	**$Q_0(t)$**	**D_2**	**D_1**	**D_0**	**D_2**	**D_1**	**D_0**
0	0	0	0	0	1	0	1	0
0	0	1	0	0	1	0	1	1
0	1	0	1	0	0	0	1	0
0	1	1	1	0	0	1	0	1
1	0	0	0	0	1	0	1	1
1	0	1	1	0	1	1	0	1
1	1	0	X	X	X	X	X	X
1	1	1	X	X	X	X	X	X

Za vsak vhod FF lahko izrišemo Veitch−ev diagram 4 spremenljivk, saj je funkcija D_i odvisna od vhoda in trenutnega stanja $Q_2(t)Q_1(t)Q_0(t)$.

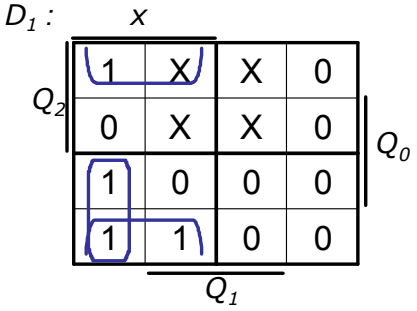
Za D_0 moramo narisati Veitch–ev diagram in funkcijo minimizirati:



Zapišemo enačbo za vhod D_0:

$$D_0 = Q_2(t) + Q_0(t) \cdot x + \overline{Q_1(t)} \cdot \overline{x}$$

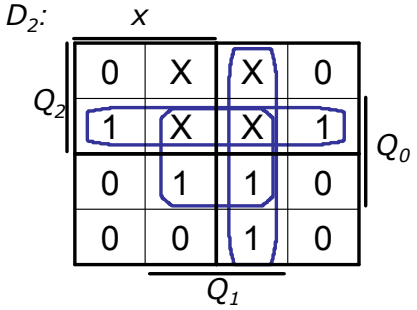
Podobno storimo za D_1:



Zapišemo enačbo za vhod D_1:

$$D_1 = x \cdot \overline{Q_0(t)} + x \cdot \overline{Q_2(t)} \cdot \overline{Q_1(t)}$$

In enako za D_2:



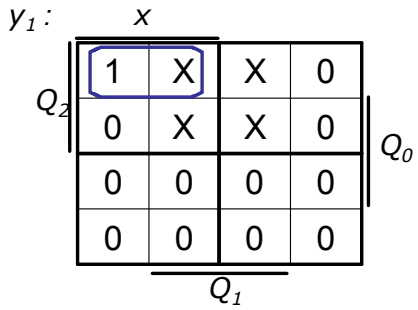
Zapišemo enačbo za vhod D_0:

$$D_2 = Q_2(t) \cdot Q_0(t) + \overline{x} \cdot Q_1(t) + Q_1(t) \cdot Q_0(t)$$

Za sintezo izhodov zapišemo samo del aplikacijske tabele, ki zajema izhoda y_1 in y_2:

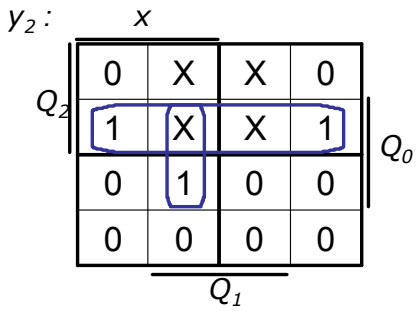
Trenutno stanje			Izhod			
			y_1		y_2	
$Q_2(t)$	**$Q_1(t)$**	**$Q_0(t)$**	**x=0**	**x=1**	**x=0**	**x=1**
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	1
1	0	0	0	1	0	0
1	0	1	0	0	1	1
1	1	0	X	X	X	X
1	1	1	X	X	X	X

Iz tabele narišemo Veitch–ev diagram za izhod y_1:



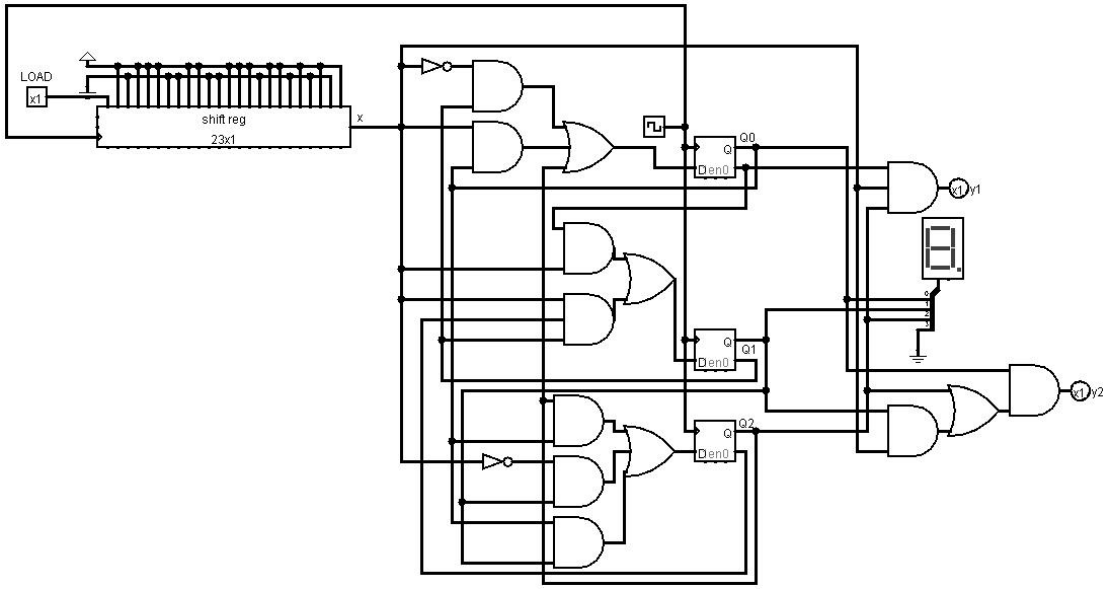
$$y_1 = Q_2(t) \cdot \overline{Q_0(t)} \cdot x$$

In podobno za izhod y_2:



$$y_2 = (Q_2(t) + x \cdot Q_1(t)) \cdot Q_0(t)$$

Vezje narišemo:



Vezje se nahaja v predlogah avditornih vaj na domači strani predmeta:
Logisim\fsm\ mealy_101_011_stop.circ

Preizkusite, ali se vezje res obnaša glede na prikazani časovni diagram v besedilu naloge. Zaradi zagotavljanja konsistentnega zaporedja vhoda *x*, kot je narisano v besedilu naloge, smo časovni potek *x* realizirali z 23−bitnim pomikalnim registrom. Vezje najprej postavite v začetno stanje (CTRL+R), nato pomikalni register naložite tako, da pritisnete na gumb LOAD in naredite en ročni impulz ure pomikanja (dvakrat pritisnete CTRL+T). Nalaganje zaključite tako, da še enkrat pritisnete gumb LOAD.

38. Realizirajte avtomat končnih stanj, ki ima dva vhoda w_1 in w_2 in izhod z. Namen avtomata končnih stanj je primerjava zaporedja vhodnih signalov w_1 in w_2: Avtomat postavi izhod $z=1$, če je štiri predhodne periode signala ure veljalo $w_1=w_2$, sicer je izhod z=0.
Delovanje avtomata končnih stanj povzema spodnje časovno zaporedje vhodov in izhoda.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
w_1	0	1	1	0	1	1	1	0	0	0	1	1	0
w_2	1	1	1	0	1	0	1	0	0	0	1	1	1
z	−	0	0	0	0	**1**	0	0	0	0	**1**	**1**	**1**

Prikazali bomo izvedbo avtomata končnih stanj Moore−ove izvedbe.

Diagram prehajanja stanj:

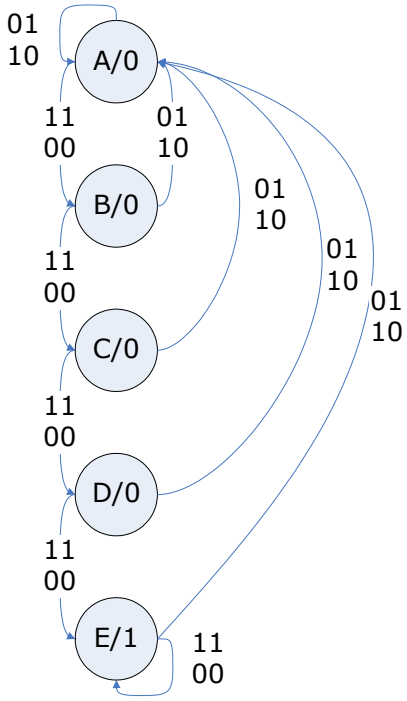


Diagram stanj začnemo risati tako, da najprej narišemo začetno stanje (A) v katerega se vračamo vedno, kadar vhodna signala w_1 in w_2 ne bosta enaka (01 ali 10). V tem stanju vztrajamo toliko časa, dokler sta vhoda različna. Šele ko postaneta enaka (11 ali 00), preidemo v drugo stanje (B), v ponovimo primerjavo vhodov in tako naprej preko stanja C, D in E. Če sta signala enaka še četrti cikel signala ure (stanje E), potem postavimo izhod na '1', v vseh ostalih stanjih je enak '0'. Čim pride do neenakosti vhodov se vrnemo nazaj v izhodiščno stanje A. Iz obnašanja vezja (tabela) je razvidno, da v stanju E ostajamo toliko časa, dokler sta signala enaka (tabela − trenutek $t_9 \rightarrow t_{10} \rightarrow t_{11}$).

Iz diagrama prehajanja stanj povzamemo naslednjo tabelo prehajanja stanj:

trenutno stanje		naslednje stanje				izhod
pomen stanja	koda stanja	w_1w_2=00	w_1w_2=01	w_1w_2=10	w_1w_2=11	z
začetno stanje	A	B	A	A	B	0
prva enakost vhodov	B	C	A	A	C	0
druga enakost vhodov	C	D	A	A	D	0
tretja enakost vhodov	D	E	A	A	E	0
četrta enakost vhodov	E	E	A	A	E	1

Če si dobljeno tabelo prehajanja stanj ogledamo, opazimo da bi se morda določena stanja dalo minimizirati z uporabo metode razdelkov. Poglejmo če se stanja dajo minimizirati z metodo razdelkov: V prvi iteraciji zberemo v skupine stanja, ki imajo enak izhod (A..D imajo izhod '0', E pa '1'):

$$P_1 = \{ABCD\}\{E\} \tag{30.1}$$

V naslednji iteraciji določimo naslednja stanja skupine stanj {ABCD} pri vseh vhodnih kombinacijah:

$$\begin{aligned} &\{ABCD\}: \\ &w = 00 \rightarrow \{BCDE\} \\ &w = 01 \rightarrow \{AAAA\} \\ &w = 10 \rightarrow \{AAAA\} \\ &w = 11 \rightarrow \{BCDE\} \end{aligned} \tag{30.2}$$

Iz enačbe (33.2) sledi, da stanje D ne sodi v to skupino, saj njegov izhod vodi v stanje E, ki je v drugi skupini stanj. Od tod sledi nova razdelitev skupin stanj: {ABC}, {D} in {E}.

$$\begin{aligned} &\{ABC\}: \\ &w = 00 \rightarrow \{BCD\} \\ &w = 01 \rightarrow \{AAA\} \\ &w = 10 \rightarrow \{AAA\} \\ &w = 11 \rightarrow \{BCD\} \end{aligned} \tag{30.3}$$

Iz enačbe (33.3) se vidi, da nobeno od stanj ni ekvivalentno, saj bi podobno kot je veljalo v iteraciji (33.3), veljalo tudi v vseh naslednjih iteracijah za stanja A, B in C. Na koncu bi vsako stanje ostalo samo v svoji skupini, kar pomeni, da nobeno od stanj ni ekvivalentno drugemu.
Tabela prehajanja stanj se zdi vredna poenostavljanja, saj njeno minimizacijo lahko naredimo s stališča vhodov, kjer na vhod avtomata namesto posameznih vhodov w_1 in w_2 (4 kombinacije) vodili kar njun rezultat enakosti vhodov. Imenujmo ga *enakost*.

w_1	w_2	*enakost*
0	0	1
0	1	0
1	0	0
1	1	1

kar je funkcija ekvivalence vhodov w_1 in w_2.

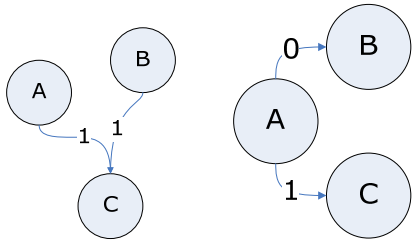
$$enakost = \overline{w_1 \oplus w_2} \tag{30.4}$$

Poenostavljena tabela prehajanja stanj bi se glasila:

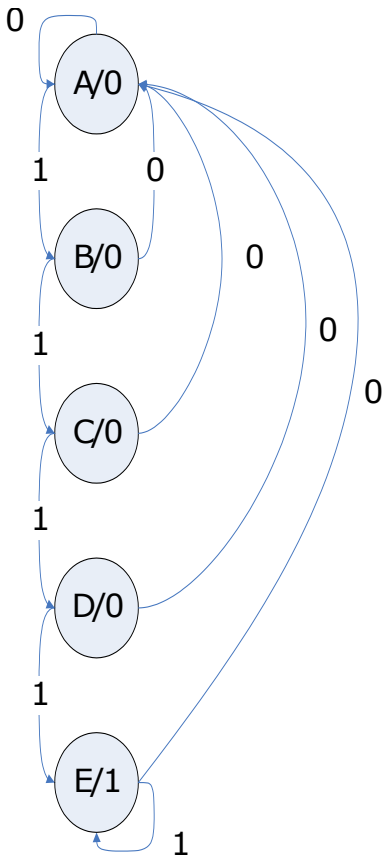
trenutno stanje		naslednje stanje		izhod
pomen stanja	koda stanja	*enakost=0*	*enakost=1*	z
začetno stanje	A	A	B	0
prva enakost vhodov	B	A	C	0
druga enakost vhodov	C	A	D	0
tretja enakost vhodov	D	A	E	0
četrta enakost vhodov	E	A	E	1

Do sedaj smo ugotovili,da bomo za izvedbo rabili najmanj 3 FF, saj je stanj 5.

Oglejmo si še izbiro kodiranja stanj avtomata. Pri izbiri kodiranja stanj avtomata moramo poiskati stanja, ki jih kodiramo kot sosedna. Pri tem iščemo takšne vzorce v diagramu stanj:



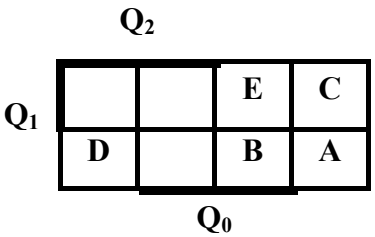
Slika 3: Vzorci pri iskanju sosednosti stanj



1. Stanja, ki imajo enako naslednje stanje za dani vhod kodiramo kot sosedna. Stanji A in B sta sosedni kot kaže levi del Slika 3.
2. Naslednja stanja, ki izvirajo iz enakega trenutnega stanja kodiramo kot sosedna. Stanji B in C sta sosedni kot kaže desni del Slika 3.
3. Stanja, ki imajo enak izhod za dani vhod kodiramo kot sosedna stanja.

Zgornja pravila si oglejmo na primeru poenostavljenega diagrama prehajanja stanj, ki vsebuje samo eno vhodno spremenljivko *enakost*:

1. Pravilo 1: Stanje A mora biti sosedno stanjem B, C in D in E.
2. Pravilo 2:
   2.1. {C, A} (sledi iz stanja B)
   2.2. {D,A} (sledi iz stanja C)
   2.3. {E,A}(sledi iz stanja D)
3. Stanja A,B,C in D so sosedna.



Q_2	Q_1	Q_0	Stanje
0	0	0	A
0	0	1	B
0	1	0	C
1	0	0	D
0	1	1	E

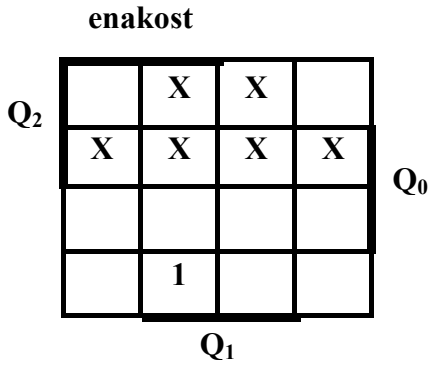
Sosedna stanja postavimo v Veitchev diagram trenutnih stanj flip−flopov Q_2, Q_1 in Q_0. Narisani primer predstavlja samo eno izmed mnogih izbir. Optimalno kodiranje stanj je ročno zelo težko določiti, saj imamo v Veitchevem diagramu 3 spremenljivk kvečjemu 3 sosednja polja − medtem ko moramo stanju A določiti 4 sosedna polja. Izberemo kodiranje stanj: A=000, B=001, C=010, D=100, E=011.

Vzbujalna tabela:

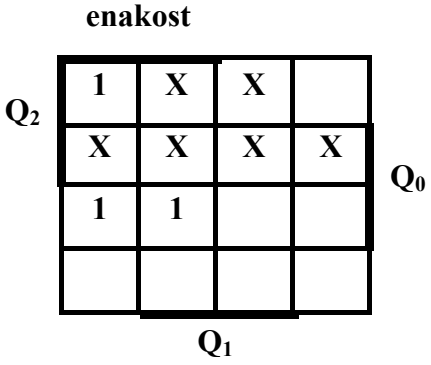
	trenutno stanje			naslednje stanje						izhod
				enakost=0			enakost=1			
	$Q_2(t)$	$Q_1(t)$	$Q_0(t)$	$Q_2(t+1)$	$Q_1(t+1)$	$Q_0(t+1)$	$Q_2(t+1)$	$Q_1(t+1)$	$Q_0(t+1)$	z
A	0	0	0	0	0	0	0	0	**1**	0
B	0	0	1	0	0	0	0	**1**	0	0
C	0	1	0	0	0	0	**1**	0	0	0
D	1	0	0	0	0	0	0	**1**	**1**	0
E	0	1	1	0	0	0	0	**1**	**1**	1
/	1	0	1	X	X	X	X	X	X	X
/	1	1	0	X	X	X	X	X	X	X
/	1	1	1	X	X	X	X	X	X	X

Preostala stanja v tabeli so kodirana z X, saj avtomat v ta stanja nikdar ne zaide. Iz vzbujalne tabele sestavimo tri Veitcheve diagrame, pri katerih sestavljamo funkcije:

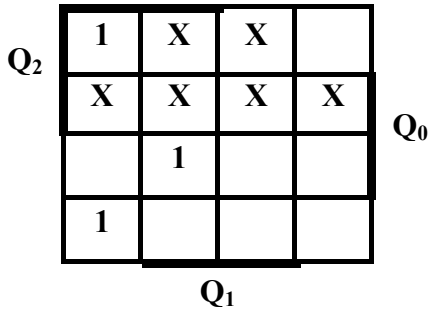
$Q_2(t+1)$:



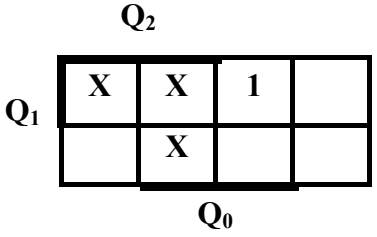
$Q_1(t+1)$:



$Q_0(t+1)$:



z:



Funkcije v Veitchevih diagramih minimiziramo in dobimo enačbe za realizacijo s pomočjo D flip−flopov. Realizacija s pomočjo D flip−flopov je najbolj enostavna, saj je enačba D flip−flopa:

$$D = Q(t+1) \tag{30.5}$$

kar pomeni, da lahko iz minimizacije Veitchevih diagramov naslednjih stanj zapišemo enačbe za vhode D flip−flopov:

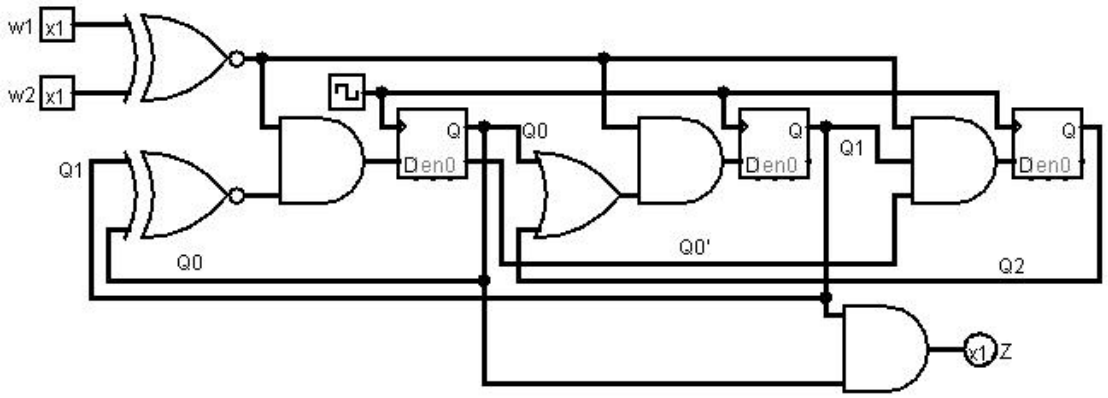
$$\begin{aligned} D_2 &= Q_2(t+1) = enakost \cdot Q_1(t) \cdot \overline{Q_0(t)} \\ D_1 &= Q_1(t+1) = enakost \cdot (Q_2(t) + Q_0(t)) \\ D_0 &= Q_0(t+1) = enakost \cdot \overline{Q_1(t)} \cdot \overline{Q_0(t)} + enakost \cdot Q_1(t) \cdot Q_0(t) = \\ D_0 &= enakost \cdot \overline{Q_1(t) \oplus Q_0(t)} \end{aligned} \tag{30.6}$$

Izhod z lahko zapišemo kot:

$$z = Q_1(t) \cdot Q_0(t) \tag{30.7}$$

Vezje avtomata narišemo iz enačb (30.6) in (30.7).

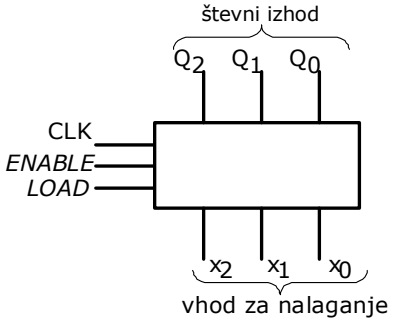
Opis delovanja in vezje avtomata, ki primerja enakost znotraj 4 period signala ure je v predlogah vaj na domači strani predmeta v imeniku Logisim\fsm\ fsm_four_periodes_of_equality.circ



39. Prikažite sintezo 3−bitnega sinhronega števca navzgor z omogočanjem štetja (*ENABLE*) in vzporednim nalaganjem (*LOAD*) z D flip−flopi, izbiralniki 2/1 in logičnimi vrati. Logika vseh krmilnih signalov je pozitivna.

S tovrstnim števcem realizirajte števec, ki šteje 2, 3, 4, 5, 2, 3, 4, 5 …

Uporabite poimenovanje signalov, kot je narisano na spodnji sliki.



Postopek sinteze zahteva, da zapišemo tabelo prehajanja stanj števca:

trenutno stanje			**naslednje stanje**			**D−FF**		
Q_2	**Q_1**	**Q_0**	**Q_2**	**Q_1**	**Q_0**	**D_2**	**D_1**	**D_0**
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	0
0	1	0	0	1	1	0	1	1
0	1	1	1	0	0	1	0	0
1	0	0	1	0	1	1	0	1
1	0	1	1	1	0	1	1	0
1	1	0	1	1	1	1	1	1
1	1	1	0	0	0	0	0	0

Iz tabele prehajanja stanj števca določimo enačbe D−FF:

Za D_0 se iz tabele vidi $D_0 = Q_0'$

Za D_1 narišemo Veitchev diagram:

D_1:

Q_1 \ Q_2				
	1	0	0	1
	0	1	1	0

Q_0

$D_1 = Q_0 \oplus Q_1$

Podobno za D_2 narišemo Veitchev diagram

D_2:

Q_1 \ Q_2				
	1	0	1	0
	1	1	0	0

Q_0

Za D_2 sledi:

$D_2=Q_2Q_1'+Q_2Q'_0+Q_2'Q_1Q_0$

iz česar lahko izpostavimo:

$D_2=Q_2(Q_1'+Q'_0) + Q_2'Q_1Q_0$

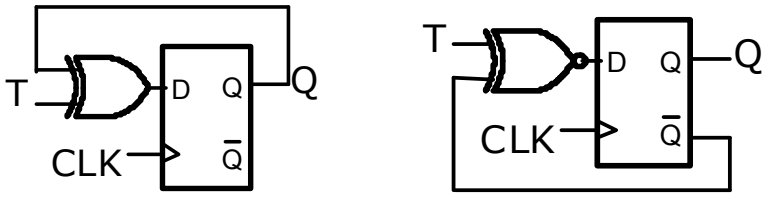
Uporabimo De Morganovo enakost:

$D_2=Q_2(Q_1Q_0)' + Q_2'(Q_1Q_0)$

Upoštevamo definicijo XOR operacije (a⊕b=a'b+ab') in dobimo:

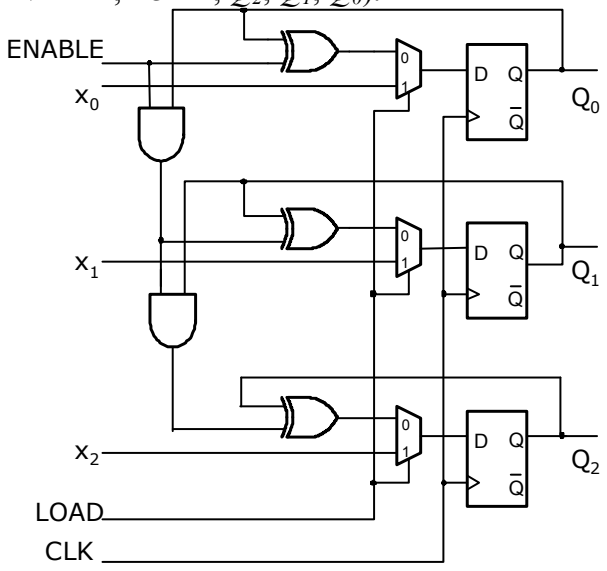
$D_2=Q_2 \oplus (Q_1 \cdot Q_0)$

Če enačbo $D_0=Q_0'$ zapišemo kot $D_0 = 1 \oplus Q_0$ in jo narišemo v vezju, smo pravzaprav realizirali T−FF s pomočjo D−FF, kot kaže levi del spodnje slike:



Slika 4: Realizacija T-FF s pomočjo D-FF in XOR vrat (levo) in XNOR vrat (desno)

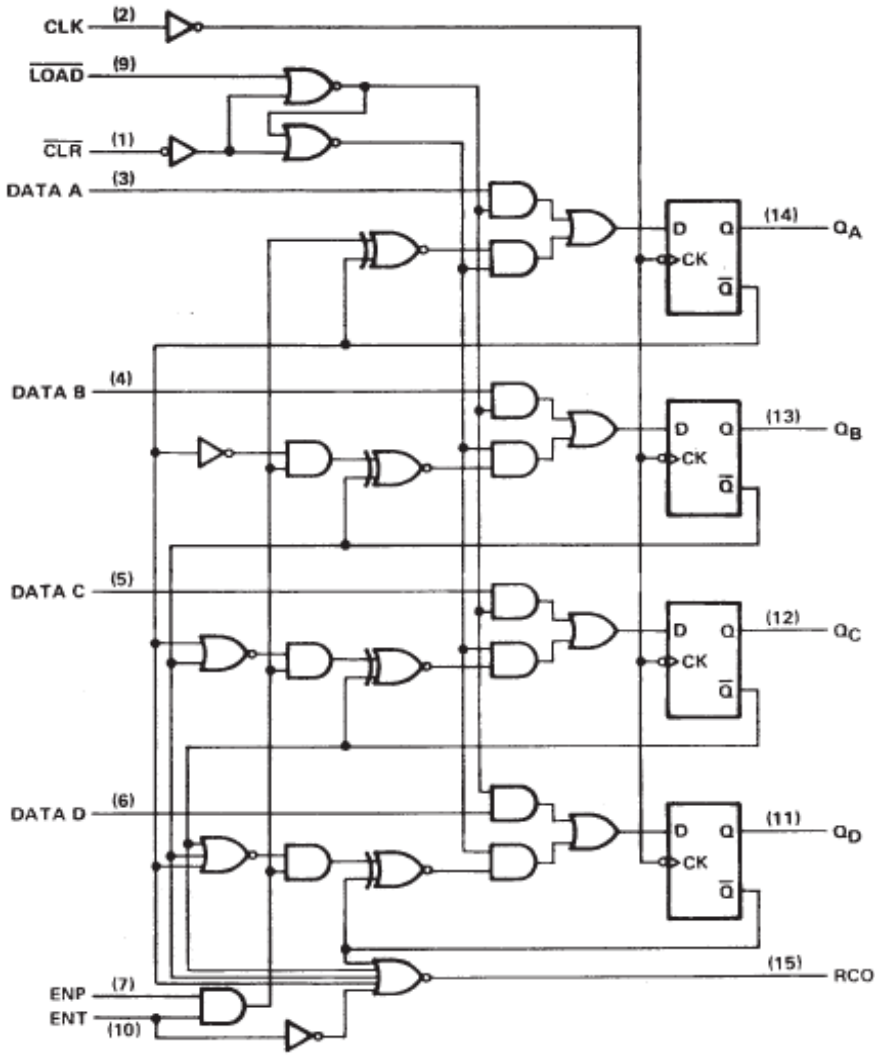
Naloga pravi, da moramo izdelati števec, ki ima vhod za omogočanje štetja (*ENABLE*): Če prvemu "T−FF" (D−FF z XOR vrati) postavimo vhod T_0 ='0' namesto T_0 ='1', vsi FF ne bodo šteli, ampak bodo ohranjali stanje. Torej, če na vhod T_0 postavimo zunanji signal *ENABLE*, števec ne bo štel, ampak ohranjal stanje, če bo *ENABLE='0'*. V verigi sinhronega števca so namreč vsi FF vezani tako, da so odvisni od prvega FF. Če stanje ohranja prvi, ga bodo tudi vsi ostali.
Za realizacijo signala za vzporedno nalaganje pa izkoristimo osnovno lastnost D−FF (pomnjenje). To storimo tako, da na vhod vsakega D−FF postavimo 2/1 izbiralnik, s katerim določimo, ali se bo dana informacija vpisala s številnega vhoda ali preko zunanjih priključkov. Do iste realizacije bi prišli, če bi v osnovni analizi upoštevali ta dva krmilna signala − analiza je veliko bolj zapletena, saj vsebuje Veitcheve diagrame za 5 spremenljivk (*ENABLE*, *LOAD*, Q_2, Q_1, Q_0).



Slika 5: Sinhroni števec z vzporednim nalaganjem (LOAD) in omogočanjem štetja (ENABLE) (3−bitna izvedba).

Če želimo z nastalim števcem šteti naraščajoče 2, 3, 4, 5 …, moramo ko števec pride do stanja 5 ($Q_2Q_1Q_0=101_2$) števec postaviti nazaj na stanje v stanje 2 ($Q_2Q_1Q_0= x_2x_1x_0= 010_2$), torej signal *LOAD* dekodiramo s pomočjo 2−vhodnih AND vrat. Pomembno pri tem je, da se pri dekodiranju zavedamo, da se Q_2 *='1'* in Q_0*='1'* v številni sekvenci pojavlja samo enkrat − če bi se večkrat, bi morali dekodirati tudi Q_1. Pri tovrstnih števcih ponavadi uporabljamo še en signal *RESET*, s katerim postavimo števec v začetno stanje, kar dosežemo tako, da na vhod D−FF za brisanje asinhrono priključimo signal *RESET*.

Nastali strukturi števca bi lahko na isti način dodali še četrti bit. Tako bi dobili podobno strukturo kot je 4−bitni sinhroni števec z vzporednim nalaganjem 74163[10]. Pri spodnji realizaciji tega vezja so uporabljeni D−FF, proženi na negativni rob signala ure (*CLK*). Štetje omogočimo s signaloma *ENP*, *ENT* (ang. enable parallel, enable transfer). Štetje je izvedeno tako, da se D−FF spremenijo v T−FF, kar dosežemo s pomočjo XNOR vrat, ki imajo en vhod vezan na števni signal (*ENT=ENP='1'*), drug vhod pa na izhod Q' FF. AND vrata pred XNOR zagotavljajo prenehanje štetja, če velja *ENT·ENP*≠'1'. Na vhodu D−FF je vezan enostaven izbiralnik iz dveh AND vrat, vezanih na OR vrata. Ta izbiralnik določa, ali bo števec štel, ali bo vzporedno naložil vrednost. Zgornja *AND* vrata izbiralnika so krmiljena s signalom *LOAD·CLR'*, spodnja pa z *LOAD'·CLR'*. Čim velja, da je *CLR'='1'* in *LOAD'='0'*, se v D−FF asinhrono naloži vsebina na vhodih Q_D Q_C Q_B Q_A=$DATA_D$ $DATA_C$ $DATA_B$ $DATA_A$, medtem ko dokler velja *LOAD'='1'* in *CLR'='1'*, bo števec štel navzgor. Če je *CLR'='0'*, je na obeh vhodih OR vrat izbiralnika '0' in stanje vseh D−FF se asinhrono postavi na $Q_DQ_CQ_BQ_A$=*"0000"*. Števni izhodi so Q_D Q_C Q_B Q_A. Izhod *RCO (ripple carry out)* se postavi na '1', ko je števec v stanju "1111" ob tem, da je *ENT='1'*. Signal RCO je izveden z NOR vrati na vhode katerih so priključeni negirani izhodi D−FF, kar je po De Morgan−ovem teoremu ekvivalent AND vratom.

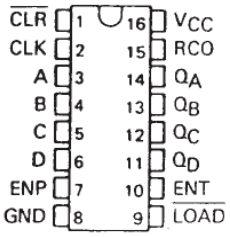


Slika 6: Struktura 4−bitnega MSI sinhronega števca z vzporednim nalaganjem (74163).

[10] http://www.alldatasheet.com/view.jsp?Searchword=74163

40. Realizirajte števec, ki šteje izmenično po modulih 7 in 11. Uporabite MSI 4−bitne števce z vzporednim nalaganjem 74163.

CLR	LOAD	ENP	ENT	CLK	A	B	C	D	QA	QB	QC	QD	RCO
0	X	X	X	POS	X	X	X	X	0	0	0	0	0
1	0	0	0	POS	X	X	X	X	A	B	C	D	*1
1	1	1	1	POS	X	X	X	X	števec šteje				*1
1	1	1	X	X	X	X	X	X	QA0	QB0	QC0	QD0	*1
1	1	X	1	X	X	X	X	X	QA0	QB0	QC0	QD0	*1



*RCO gre na '1', ko gre štetje iz 15 na 0. Q_A je najmanj pomemben bit štetja.

Izmenično štetje od $0..6_{10}$ in od $0..10_{10}$:

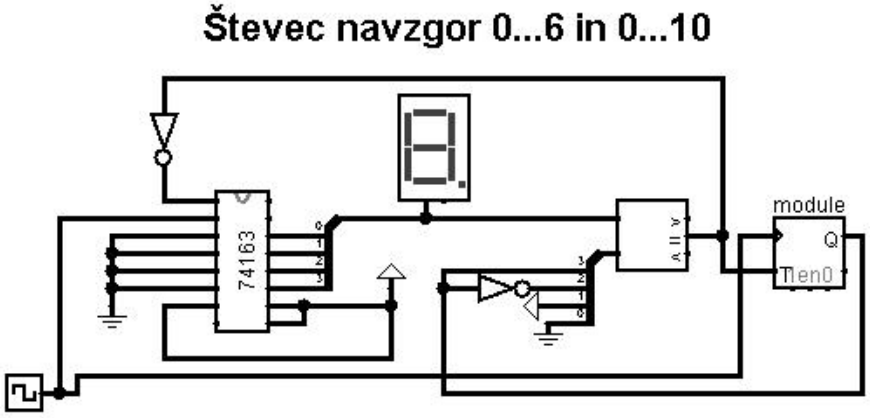
#	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0

Naloga zahteva, da realiziramo štetje 0..6 in 0..10. To lahko izvedemo z enim 4−bitnim števcem, ker štejemo največ do $10_{10}=1010_2$ za kar potrebujemo 4 mesta. Realizacija bo torej vključevala en MSI števec, s katerim bi lahko šteli od 0 do največ 15. Vse številne izhode števca vežemo na vhod 4−bitnega primerjalnika velikosti (magnitude comparator), katerega izhod se postavi na '1', ko števec prešteje do 6 oz. do 10. Ta signal uporabimo za krmiljenje *CLR'* signala − torej bomo takrat postavili števec na *"0000"*. Obenem vodimo ta signal na vhod T−FF, ki pomni modul štetja. Izhod T−FF bo '0', če števec šteje do 6_{10}, oz. T= '1', če šteje do 10_{10}. Število $6_{10}=0110_2$, medtem ko je število $10_{10}=1010_2$. Če števili 6_{10} in 10_{10} primerjamo, sta negirani na mestu primerjanja Q_3 in Q_2, medtem ko sta pri Q_1 in Q_0 enaki. To izkoristimo pri detekciji največje vrednosti štetja:

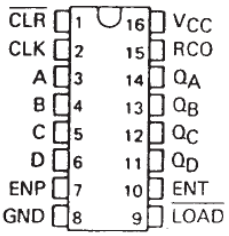
Če bo izhod T−FF *Q='1'*, bo števec štel do 1010_2 torej bomo na mesto primerjanja Q_3 vodili izhod T−FF (*Q*), na mesto primerjanja Q_2 pa invertiran izhod T−FF *Q'*.

Če bo izhod T−FF enak *Q='0'* bo števec štel do 0110_2, torej bosta takrat mesti primerjanja Q_3*='0'*, Q_2*='1'*, kar na vhodu primerjalnika velikosti pomeni štetje do največ 6_{10}.

Številni izhod celotnega števca je vezan na dvomestni šestnajstiški prikazovalnik z dekoderjem, kot je prikazano na sliki realizacije.



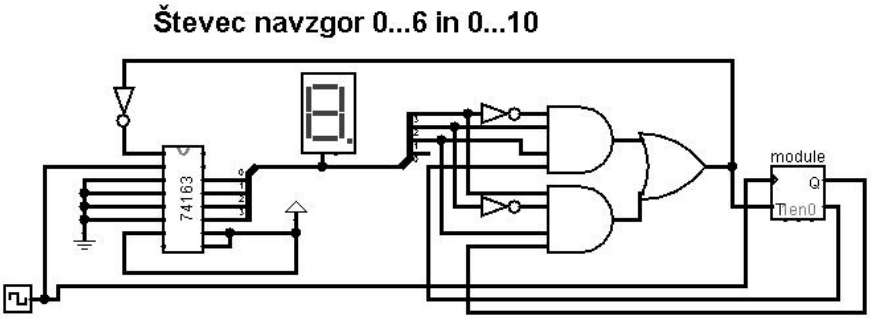
Razpored priključkov števca 74163



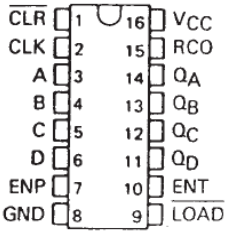
Realizacija s primerjalnikom velikosti je sicer najbolj logična izbira, a še zdaleč ni optimalna. Isto primerjavo lahko izvedemo z AND vrati, ki jim nastavimo kombinacijo primerjave z opazovanjem številnega zaporedja:

- Zgornja AND vrata primerjajo, kdaj so biti primerjanja $Q_3Q_2Q_1$ *="011"*, kar se v štetju $0..6_{10}$ zgodi samo takrat ko števec prešteje do 6_{10}.
- Spodnja AND vrata primerjajo, kdaj so biti primerjanja $Q_3Q_2Q_1$*="101"*, kar se v štetju $0..10_{10}$ zgodi samo takrat ko števec prešteje do 10_{10}.

Ko je rezultat primerjave zgornjih ali spodnjih vrat enak '1', postane vhod *T='1'* kar spremeni stanje izhoda T−FF (ang. toggle), ki zamenja vrednosti primerjave. Obenem izhod '1' OR vrat preko inverterja zbriše števec (*CLEAR'='0'*).

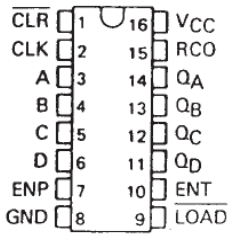


Razpored priključkov števca 74163



41. Realizirajte semafor, ki deluje na sledeč način: zelena luč gori 13 sekund, nato rumena luč 3 sekunde, nato rdeča luč 13 sekund in nato se vse skupaj ponavlja. Uporabite MSI 4−bitne števce z vzporednim nalaganjem 74163, logična vrata in demultiplekser. Frekvenca vhodnega signala ure je 1 Hz.

CLR	LOAD	ENP	ENT	CLK	A	B	C	D	QA	QB	QC	QD	RCO
0	X	X	X	POS	X	X	X	X	0	0	0	0	0
1	0	0	0	POS	X	X	X	X	A	B	C	D	*1
1	1	1	1	POS	X	X	X	X	števec šteje				*1
1	1	1	X	X	X	X	X	X	QA0	QB0	QC0	QD0	*1
1	1	X	1	X	X	X	X	X	QA0	QB0	QC0	QD0	*1



*RCO gre na '1', ko gre štetje iz 15 na 0. Q_A je najmanj pomemben bit štetja.

Delovanje semaforja realiziramo tako, da realiziramo izmenično štetje od $0..2_{10}$ in od $0..12_{10}$:

#	Q_D	Q_C	Q_B	Q_A
0	0	0	0	0
1	0	0	0	1
2	**0**	**0**	**1**	0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	**1**	**1**	**0**	0

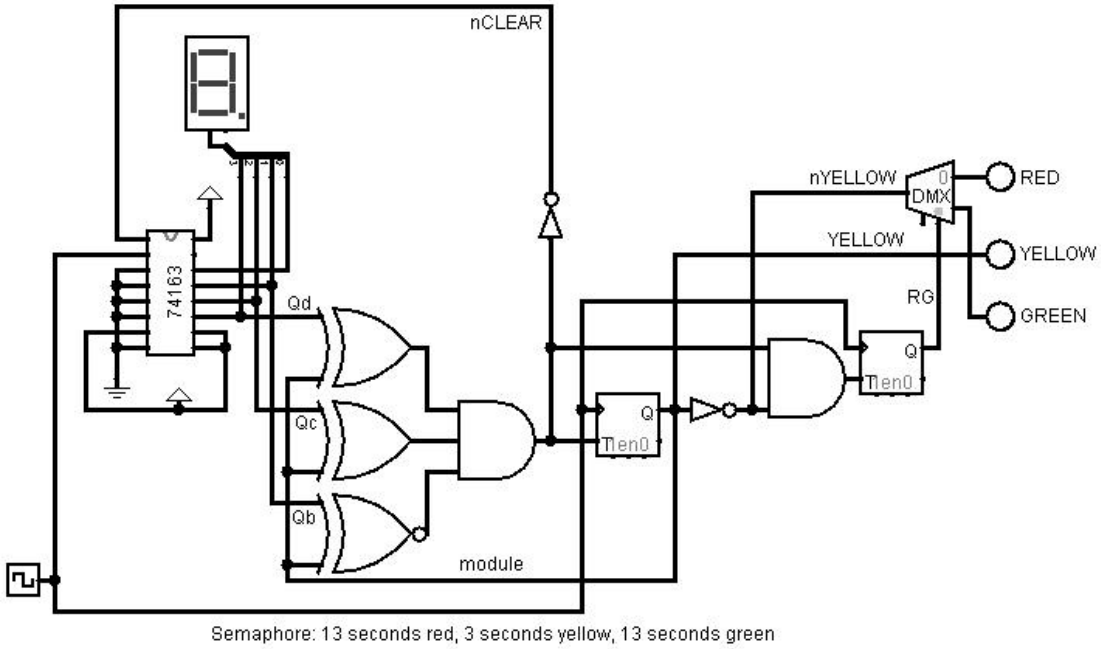
To lahko izvedemo z enim 4−bitnim števcem, ker štejemo največ do $12_{10}=1100_2$ za kar potrebujemo 4 mesta. Realizacija bo torej vključevala en MSI števec, s katerim bi lahko šteli od 0 do največ 15. Končni stanji štetja sta $02_{10}=0010_2$ in $12_{10}=1100_2$. Ti stanji dekodiramo tako, da je LSB štetja vedno (Q_A='0'), števne izhode (Q_D, Q_C, Q_B) vežemo na vhod enostavnega primerjalnika enakosti, katerega izhod se postavi na '1', ko števec prešteje do 2_{10} oz. do 12_{10}. Ta signal uporabimo za krmiljenje *CLR'* signala (na sliki rešitve *nCLEAR*) − ki postavi števec na *"0000"*. Obenem vodimo ta signal na vhod T−FF, ki pomni modul štetja. Izhod T−FF bo '0', če števec šteje do 12_{10}, oz. T= '1', če šteje do 2_{10}. Če števili 2_{10} in 12_{10} primerjamo, sta negirani na mestih primerjanja $Q_DQ_CQ_B$ medtem ko sta pri Q_A enaki, zato mesta Q_A ni potrebno primerjati. To izkoristimo pri detekciji največje vrednosti štetja. Primerjalnik izvedemo z dvemi XOR in enimi XNOR vrati.

Če bo izhod T−FF *Q='0'*, bo števec štel do "1100"$_2$. V tem primeru bomo izhod T−FF (signal *module* na sliki rešitve) vodili na primerjalnik, ki primerja, kdaj imajo mesta $Q_DQ_CQ_B$ = "110", kar se v štetju ($0..12_{10}$) pojavi samo pri stanju 12_{10}.

Če bo izhod T−FF *Q='1'*, bo števec štel do 0010_2. V tem primeru bomo izhod T−FF *(*signal *module*) vodili na primerjalnik, ki primerja, kdaj imajo mesta $Q_DQ_CQ_B$ = "001", kar se v štetju ($0..2_{10}$) pojavi samo pri 2_{10}.

Tako smo realizirali števec, ki šteje izmenično od $0..2_{10}$ in od $0..12_{10}$: Rumena luč (signal *YELLOW*) gori vedno, ko števec šteje od $0..2_{10}$, zato signal za rumeno luč

vodimo kar iz T−FF, ki določa modul štetja (signal *module*). Signala za rdečo in zeleno luč moramo dekodirati s pomočjo še enega T−FF, ki hrani informacijo, katera luč je bila nazadnje prižgana: rdeča ali zelena (signal *RG*). Signal *RG* vodimo na vhod demultiplekserja 1/2, ki prižge rdečo luč, ko je *RG='0'* in zeleno, ko je signal *RG='1'*. Podatkovni vhod demultiplekserja postavimo na '1' samo takrat, ko rumena luč ne gori (signal *nYELLOW*), saj bi v nasprotnem primeru na semaforju naenkrat goreli dve luči (rumena in rdeča oz. rumena in zelena).

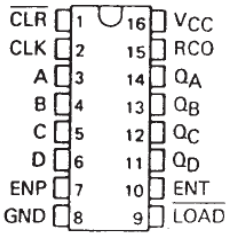


Semaphore: 13 seconds red, 3 seconds yellow, 13 seconds green

Opis delovanja in vezje semaforja je v predlogah avditornih vaj na domači strani predmeta: Logisim\counter\semafor_3_13.circ

42. Realizirajte števec, ki šteje do 30 in nazaj (0, 1,…, 29, 30, 29,… 1, 0, 1,..). Uporabite MSI 4−bitne števce z vzporednim nalaganjem 74163.

CLR	LOAD	ENP	ENT	CLK	A	B	C	D	QA	QB	QC	QD	RCO
0	X	X	X	POS	X	X	X	X	0	0	0	0	0
1	0	0	0	POS	X	X	X	X	A	B	C	D	*1
1	1	1	1	POS	X	X	X	X	števec šteje				*1
1	1	1	X	X	X	X	X	X	QA0	QB0	QC0	QD0	*1
1	1	X	1	X	X	X	X	X	QA0	QB0	QC0	QD0	*1



*RCO gre na '1', ko gre štetje iz 15 na 0. Q_A je najmanj pomemben bit štetja.

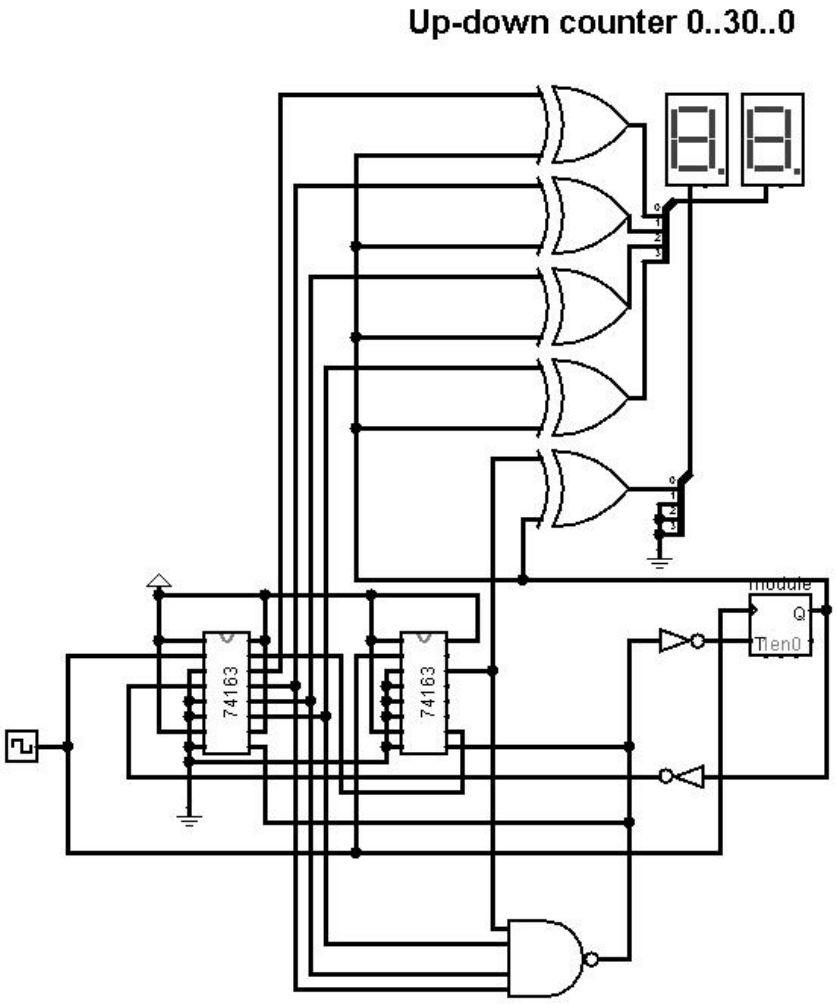
Štetje $0..30_{10}$:

#	Q_4	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	1	0	0
5	0	0	1	0	1
6	0	0	1	1	0
7	0	0	1	1	1
8	0	1	0	0	0
9	0	1	0	0	1
10	0	1	0	1	0
11	0	1	0	1	1
12	0	1	1	0	0
13	0	1	1	0	1
14	0	1	1	1	0
15	0	1	1	1	1
16	1	0	0	0	0
17	1	0	0	0	1
18	1	0	0	1	0
19	1	0	0	1	1
20	1	0	1	0	0
21	1	0	1	0	1
22	1	0	1	1	0
23	1	0	1	1	1
24	1	1	0	0	0
25	1	1	0	0	1
26	1	1	0	1	0
27	1	1	0	1	1
28	1	1	1	0	0
29	1	1	1	0	1
30	1	1	1	1	0

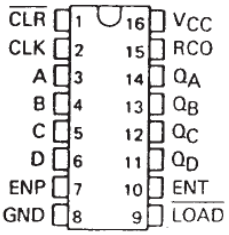
Naloga zahteva, da realiziramo štetje 0..30. Tega z enim 4−bitnim števcem ne moremo doseči, ker za hranjenje stanja $30_{10}=11110_2$ potrebujemo 5 mest. Realizacija bo torej vključevala dva MSI števca, ki sta vezana zaporedno − torej je RCO prvega števca vezan na ENT drugega števca. S takšno vezavo bi lahko šteli od 0 do največ 31. Vse številne izhode prvega števca in LSB višjega števca vežemo na vhod NAND vrat, katerih izhod se postavi na '0', ko števca preštejeta do 30. Ta signal porabimo za krmiljenje LOAD' signala. Obenem vodimo negacijo tega signala na T−FF, ki pomni smer štetja. Izhod T−FF bo '0', če je smer štetja navzgor. Sam MSI števec nima vhoda za smer štetja, zato moramo to posebej realizirati: Če opazujemo tabelo štetja na levi, lahko za štetje navzdol izhodne vrednosti števca komplementiramo, saj je 1_{10} eniški komplement 30_{10}, 2_{10} eniški komplement 29_{10}…

Komplementiranje realiziramo s pomočjo petih XOR vrat. Štetje navzdol realiziramo tako, da ob LOAD' signalu naložimo v števec 2_{10} in omogočimo komplementiranje, medtem ko se ob štetju navzgor v števec naloži 0_{10}. Zakaj 2_{10} in ne 1_{10}? Zato. da števec ne šteje … 29, **30**, **30**, 29, 28, 27 ...

Na sliki realizacije je nalaganje pravilnega začetnega stanja izvedeno z inverterjem, ki je vezan na T−FF in na B priključek vzporednega nalaganja prvega števca (glej razpored priključkov števca). Števni izhod celotnega števca je vezan na dvomestni šestnajstiški prikazovalnik z dekoderjem.

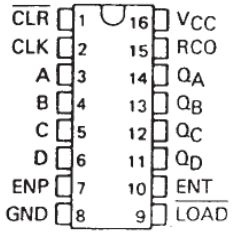


Razpored priključkov števca 74163



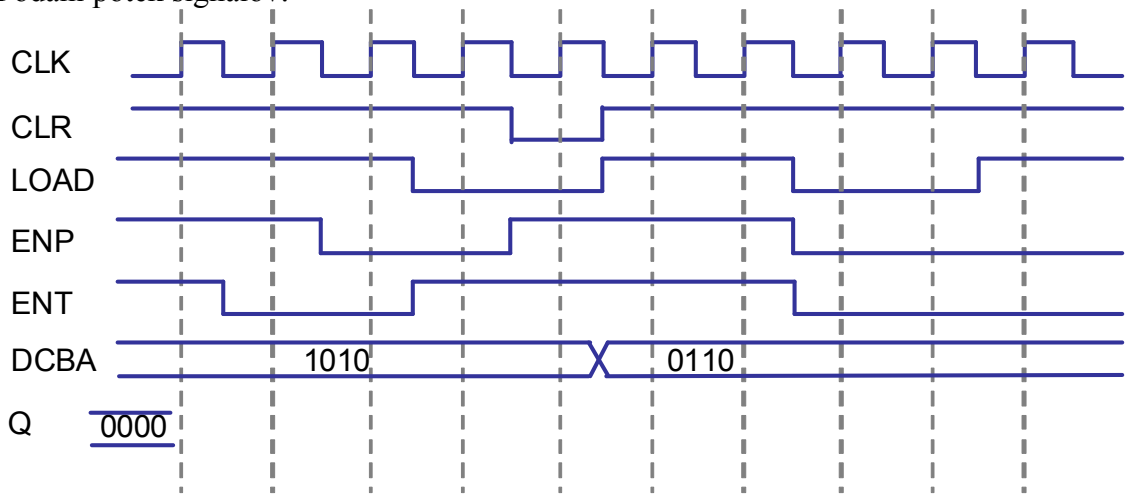
43. Določite 4−bitni izhodni signal (Q_D, Q_C, Q_B, Q_A) MSI števca 74163, če je podan potek vhodnih signalov.

CLR	LOAD	ENP	ENT	CLK	A	B	C	D	QA	QB	QC	QD	RCO
0	X	X	X	POS	X	X	X	X	0	0	0	0	0
1	0	0	0	POS	X	X	X	X	A	B	C	D	*1
1	1	1	1	POS	X	X	X	X	števec šteje				*1
1	1	1	X	X	X	X	X	X	QA0	QB0	QC0	QD0	*1
1	1	X	1	X	X	X	X	X	QA0	QB0	QC0	QD0	*1

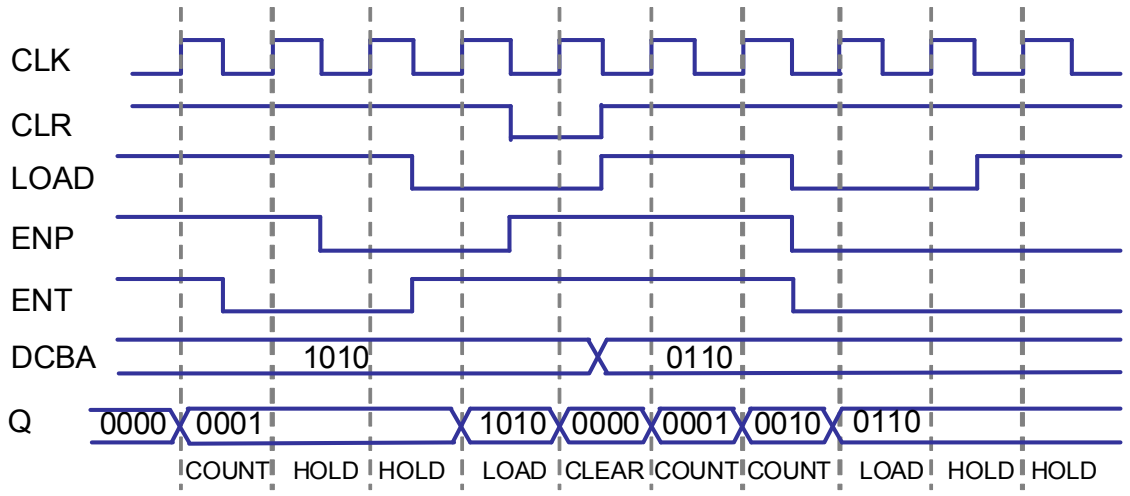


*RCO gre na '1', ko gre štetje iz 15 na 0. Q_A je najmanj pomemben bit štetja.

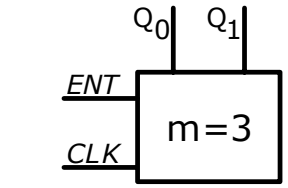
Podani potek signalov:

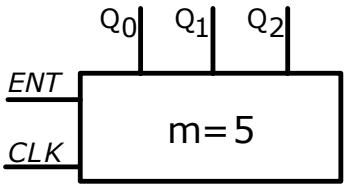


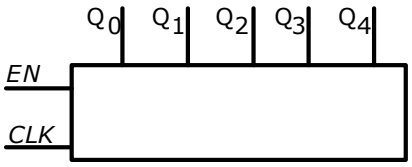
Rešitev dobimo, če opazujemo vrednosti vhodnih signalov ob prednjem robu signala ure (CLK) in iz tabele delovanja razberemo stanje števca. Števec asinhrono nalaga vrednost, čim je CLR='0'. Števec šteje, če sta ENT in ENP='1' in je CLR='1' ter LOAD='1'. Števec sinhrono naloži vrednost z vhodov DCBA, če sta ENP in ENT enaka '0' in je LOAD signal '0'.



44. Imamo števec po modulu 3 in števec po modulu 5. Kako šteje sestavljen števec, če ju vežemo enkrat v kaskado sinhrono in drugič asinhrono? Oba števca imata vhod za omogočenje štetja (*ENT*). Nastala struktura števca ima vhod za omogočanje (*EN*), števni vhod (*CLK*) in števni izhod (Q_4, Q_3, Q_2, Q_1, Q_0)



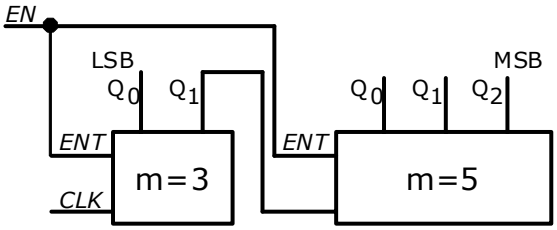




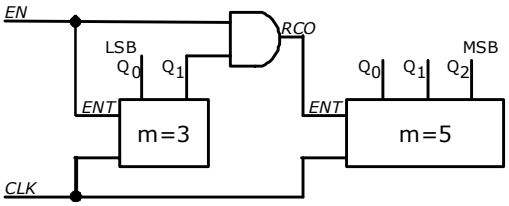
Struktura nastalega števca

Pomnilne celice asinhronega števnika so prožene z izhodi drugih celic in nimajo skupnega urinega signal, zato na vsakega od števcev gledamo kot celoto, ki ima števni vhod (*clk*), števni izhod (Q_2, Q_1, Q_0) in vhod za omogočenje (*ENT*). Če števca vežemo asinhrono, je MSB mesto prvega števca vezano na števni vhod drugega števca. Omogočen števec se poveča na vsak sprednji rob števnega vhoda CLK. Števno zaporedje asinhrone kaskade števcev povzema spodnja tabela.

$\#_{16}$	Q_4	Q_3	Q_2	Q_1	Q_0
	Q_2	Q_1	Q_0	Q_1	Q_0
0_{16}	0	0	0	0	0
1_{16}	0	0	0	0	1
6_{16}	0	0	1	1	0
4_{16}	0	0	1	0	0
5_{16}	0	0	1	0	1
A_{16}	0	1	0	1	0
8_{16}	0	1	0	0	0
9_{16}	0	1	0	0	1
E_{16}	0	1	1	1	0
C_{16}	0	1	1	0	0
D_{16}	0	1	1	0	1
12_{16}	1	0	0	1	0
10_{16}	1	0	0	0	0
11_{16}	1	0	0	0	1
2_{16}	0	0	0	1	0



Če števca vežemo v sinhrono kaskado, bosta štela z istim signalom ure (*clk*). Višji modul štetja je prožen s signalom *RCO* (ang. ripple carry out), ki postane '1', ko gre štetje iz stanja *m−1* v stanje 0_{10}, ob pogoju, da je štetje omogočeno (*ENT='1'*). Uporabljeni števec po modulu 3 nima RCO signala, zato ga moramo izdelati s pomočjo AND vrat. Vrata potrebujemo zato, da lahko sinhroni števec ustavimo (EN='0'). Števec šteje po zaporedju 0_{10}, 1_{10}, 2_{10}, torej lahko MSB izhod tega števca (Q_1) izkoristimo za detekcijo prehoda zaporedja na 0_{10}. Dobljeni signal vodimo na vhod za omogočenje štetja naslednje stopnje.



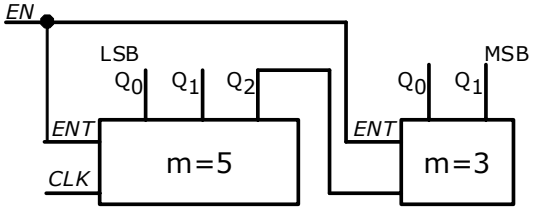
Števno zaporedje se tokrat glasi:

#16	Q4 Q3 Q2 Q1 Q0	
	Q2	Q1 Q0 Q1 Q0
016	0	0 0 0 0
116	0	0 0 0 1
216	0	0 0 1 0
416	0	0 1 0 0
516	0	0 1 0 1
616	0	0 1 1 0
816	0	1 0 0 0
916	0	1 0 0 1
A16	0	1 0 1 0
C16	0	1 1 0 0
D16	0	1 1 0 1
E16	0	1 1 1 0
1016	1	0 0 0 0
1116	1	0 0 0 1
1216	1	0 0 1 0

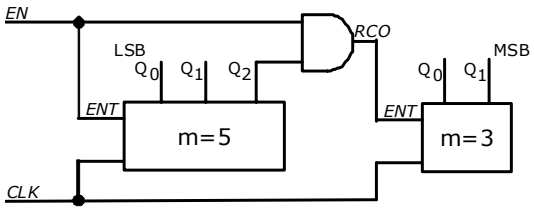
Modula štetja lahko obrnemo, tako da bo na nižjem mestu števec po modulu m=5. Če števca vežemo asinhrono, je MSB mesto števca po modulu 5 vezano na števni vhod drugega števca. Števno zaporedje se glasi:

#16	Q4 Q3 Q2 Q1 Q0	
	Q1 Q0	Q2 Q1 Q0
016	0 0	0 0 0
116	0 0	0 0 1
216	0 0	0 1 0
316	0 0	0 1 1
C16	0 1	1 0 0
816	0 1	0 0 0
916	0 1	0 0 1
A16	0 1	0 1 0
B16	0 1	0 1 1
1416	1 0	1 0 0
1016	1 0	0 0 0
1116	1 0	0 0 1
1216	1 0	0 1 0
1316	1 0	0 1 1
416	0 0	1 0 0

Asinhrona vezava modulov:



Če števca vežemo sinhrono kaskado, bosta štela z istim signalom ure (*clk*). Višji modul štetja je prožen s signalom *RCO* (ang. ripple carry out), ki postane '1', ko gre štetje iz stanja *m−1* v stanje 0_{10}. Števec po modulu 5 šteje po zaporedju 0_{10}, 1_{10}, 2_{10}, 3_{10}, 4_{10} torej lahko MSB izhod tega števca (Q_2) izkoristimo za detekcijo prehoda zaporedja na 0_{10}. Dobljeni signal vodimo na vhod za omogočenje štetja naslednje stopnje preko AND vrat, s pomočjo katerih celoten števec lahko ustavimo (EN='0').

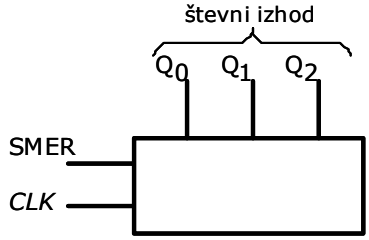


Števno zaporedje se glasi:

#16	Q4 Q3 Q2 Q1 Q0	
	Q1 Q0	Q2 Q1 Q0
016	0 0	0 0 0
116	0 0	0 0 1
216	0 0	0 1 0
316	0 0	0 1 1
416	0 0	1 0 0
816	0 1	0 0 0
916	0 1	0 0 1
A16	0 1	0 1 0
B16	0 1	0 1 1
C16	0 1	1 0 0
1016	1 0	0 0 0
1116	1 0	0 0 1
1216	1 0	0 1 0
1316	1 0	0 1 1
1416	1 0	1 0 0

Vse vezave se nahajajo v Logisim predlogah rešenih nalog na domači strani predmeta v imeniku: Logisim\counter\CASCADING

45. Prikažite sintezo sinhronega dvosmernega 3−bitnega števca z uporabo T flip−flopov: Zapišite tabelo prehajanja stanj in določite enačbe flip−flopov. Števec ima vhod SMER, ki določa smer štetja: Če je SMER='0', števec šteje naraščajoče, sicer padajoče. Imena signalov so razvidna iz spodnje slike.



Postopek sinteze zahteva, da zapišemo tabelo prehajanja stanj števca:

SMER	Q_2	Q_1	Q_0	Q_2	Q_1	Q_0	T_2	T_1	T_0
0	0	0	0	0	0	1	0	0	1
0	0	0	1	0	1	0	0	1	1
0	0	1	0	0	1	1	0	0	1
0	0	1	1	1	0	0	1	1	1
0	1	0	0	1	0	1	0	0	1
0	1	0	1	1	1	0	0	1	1
0	1	1	0	1	1	1	0	0	1
0	1	1	1	0	0	0	1	1	1
1	0	0	0	1	1	1	1	1	1
1	0	0	1	0	0	0	0	0	1
1	0	1	0	0	0	1	0	1	1
1	0	1	1	0	1	0	0	0	1
1	1	0	0	0	1	1	1	1	1
1	1	0	1	1	0	0	0	0	1
1	1	1	0	1	0	1	0	1	1
1	1	1	1	1	1	0	0	0	1

Normalna analiza bi zahtevala, da narišemo Veitch−eve diagrame za štiri spremenljivke za vsak vhod T−FF, vendar ker so T−FF po svoji naravi primerni za realizacijo števcev, so praviloma njihove vhodne enačbe zelo enostavne. Iz tabele prehajanja stanj števca določimo enačbe T−FF:

Iz stolpca T_0 se vidi, da je T_0='1'. Kot zanimivost omenimo dejstvo, ki se ponavlja pri večini sinhronih števcev: Če namesto '1' na T_0 vodimo nek zunanji signal, nam to omogoča štetje števca (ENABLE oz. EN).

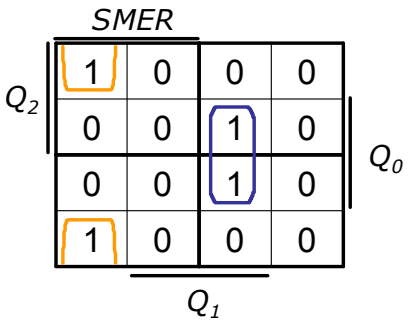
Iz stolpca T_1 se vidi, da se ponavlja vzorec 01, če je SMER='0' in 10, če je SMER ='1'.

SMER	T_1
0	Q_0
1	Q_0'

kar lahko kratko zapišemo kot:

$$T_1 = SMER \cdot \overline{Q_0} + \overline{SMER} \cdot Q_0 = SMER \oplus Q_0$$

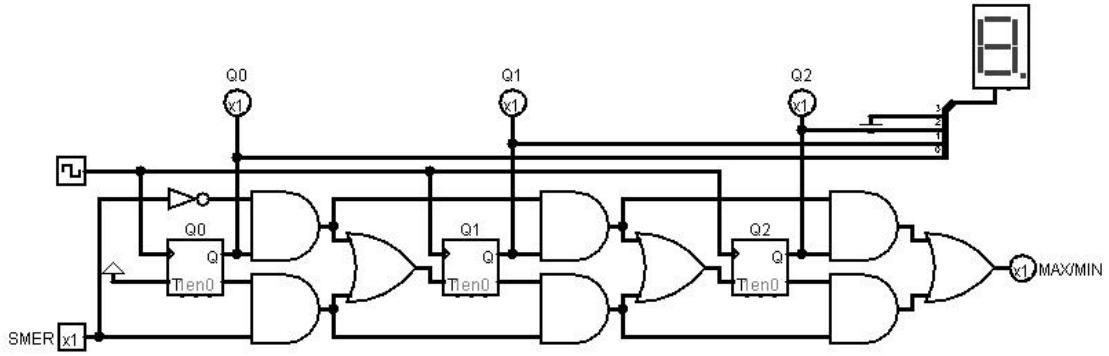
Za T_2 se da enostavno ugotoviti realizacijo iz Veitch−evega diagrama:



$$T_2 = SMER \cdot \overline{Q_1} \cdot \overline{Q_0} + \overline{SMER} \cdot Q_1 \cdot Q_0$$

V enačbi za T_2 poiščemo podobnosti z enačbo za T_1: Enačba za T_1 vsebuje konjunkciji SMER·Q_0' in SMER'·Q_0, ki sta vsebovani tudi v enačbi za T_2, kar nam dodatno poenostavi realizacijo števca. Obenem nam taka realizacija nakazuje osnovno strukturo, ki jo lahko s ponavljanjem razširimo v večbitni dvosmerni sinhroni števec.

Primer podobnega vezja 4−bitnega dvojiškega dvosmernega števca, ki ima še vzporedno nalaganje je 74191[11]. Če boste primerjali našo realizacijo in realizacijo v podatkovnem listu, boste opazili, da je v dejanski realizaciji 74191 precej več večvhodnih AND vrat: Delno je razlog za to v dodani logiki za vzporedno nalaganje, delno pa tudi zato, da zagotovimo enakomerno zakasnitev med posameznimi stopnjami števca.



Ko narišemo vezje dvosmernega števca, zelo spominja na združitev sinhronega števca za štetje navzgor in sinhronega števca za štetje navzdol: Če bi števec vseboval samo zgornja AND vrata (vezanih neposredno na T vhod − brez OR) bi bil to števec navzgor, če pa samo spodnja AND bi bil števec navzdol. Signal SMER določa katera AND vrata so omogočena:

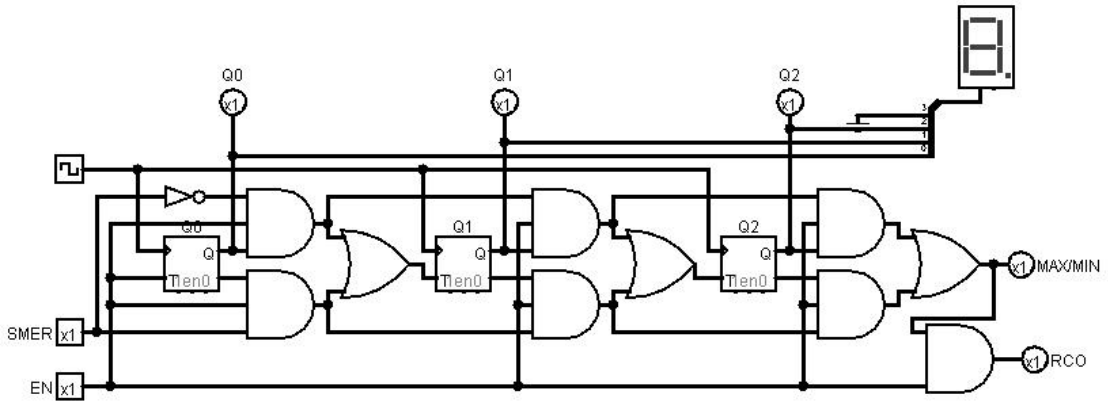
- zgornja AND vrata, ko je SMER='0' − štejemo naraščajoče,
- spodnja AND vrata, ko je SMER='1' − štejemo padajoče.

Signal EN je signal za omogočanje štetja − dokler je EN='0' se vsebina vseh T−FF ne spreminja, ampak ohranja trenutno stanje. Iz sheme števca je razviden tudi način razširitve števca na večje število bitov.
Pri tovrstnih števcih želimo realizirati tudi signal za proženje naslednjih stopenj števca RCO (ang.. ripple carry out) poimenovan včasih tudi TC (ang. terminal count), oz. tudi RC (ang. ripple clock). RCO je signal, ki postane '1', ko števec preide iz najvišjega stanja (v našem primeru "111") v stanje "000" pri štetju navzgor in ko preide iz najnižjega stanja "000" v najvišje stanje ("111") pri štetju navzdol:

SMER	RCO
0	$Q_0 \cdot Q_1 \cdot Q_2$
1	$Q_0' \cdot Q_1' \cdot Q_2'$

Tak signal uporabljamo pri realizaciji večbitnih števcev tako, da izdelane 3 bitne števce vežemo kaskadno − torej da signal RCO vežemo na EN signal naslednjega vezja. Za realizacijo takega signala bi narisali enako kombinacijo AND in OR vrat še na izhodu Q_2, kot kaže naslednja slika:

[11] http://www.alldatasheet.com/view.jsp?Searchword=74191

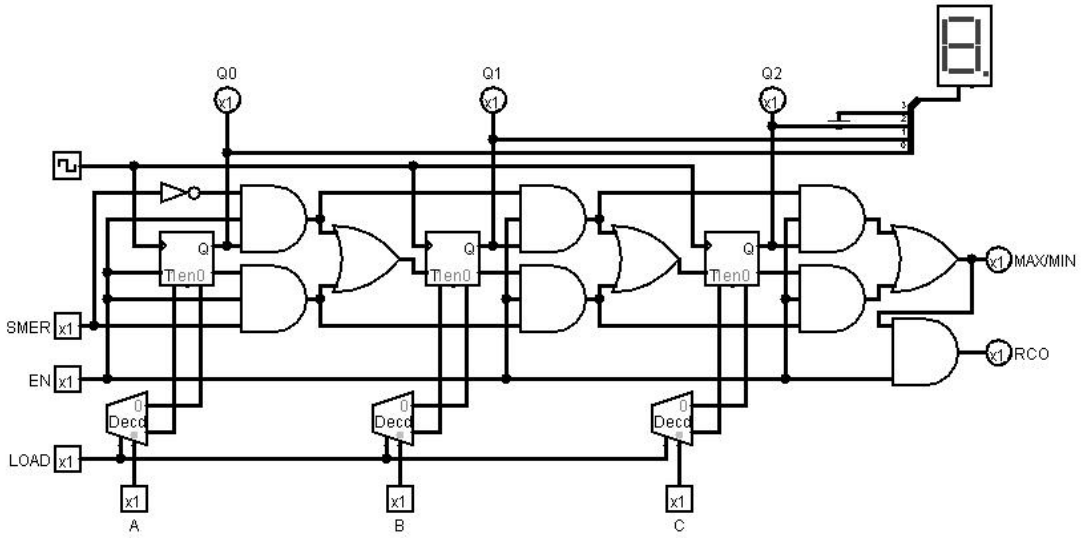


Vezje se nahaja v Logisim predlogah rešenih nalog na domači strani predmeta:
Logisim\counter\counter_up_down_3_bit_using_T_FF_with_enable.circ

Večina števcev je realizirana v 4−bitni zasnovi, tako da glede na vrednost RCO signala ločimo dve skupini števcev:

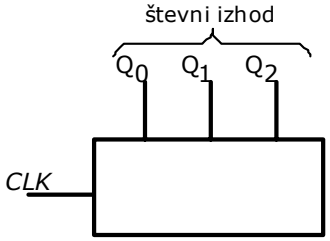
- desetiški (BCD) števci, katerih RCO se postavi na '1' takrat, ko števec preide iz stanja "1001" v "0000" in
- dvojiški (binarni), katerih RCO se postavi na '1' takrat, ko števec preide iz stanja "1111" v "0000". Več o delovanju RCO najdete v opisu delovanja števcev 74161[12].

Števcu dodamo še signal za asinhrono vzporedno nalaganje (LOAD), ki v aktivnem stanju (LOAD='1') asinhrono nastavi vrednosti T−FF na vrednosti vhodov za nalaganje C, B, A. V števcu 74191[13].je izveden z enostavnim dekoderjem, ki postavi ustrezni vhod za asinhrono postavljanje T−FF (ang. preset) na '1', če je vrednost vhoda za nalaganje '1'. Če je vrednost vhoda za nalaganje '0', potem se '1' postavi na asinhroni vhod za brisanje T−FF (ang. clear).
Vezje se nahaja v Logisim predlogah rešenih nalog na domači strani predmeta:
Logisim\counter\counter_up_down_3_bit_using_T_FF_with_enable_with_load.circ



[12] http://www.alldatasheet.com/view.jsp?Searchword=74161
[13] http://www.alldatasheet.com/view.jsp?Searchword=74191

46. Prikažite sintezo 3−bitnega sinhronega števca *navzdol* po Graye−vi kodi s T flip−flopi in logičnimi vrati. Števec ima 3−bitni števni izhod (Q_2, Q_1, Q_0). Uporabite poimenovanje signalov, kot je narisano na spodnji sliki.

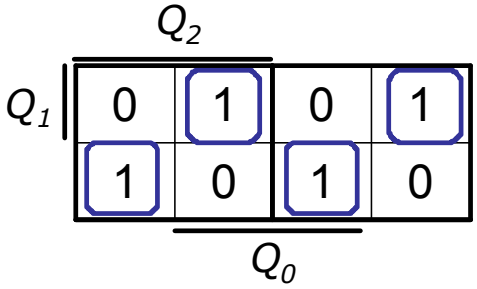


Postopek sinteze zahteva, da zapišemo tabelo prehajanja stanj števca navzdol po Gray−evi kodi. Desetiška števna sekvenca po 3−bitni Grayevi kodi se glasi:
… 0, 4, 5, 7, 6, 2, 3, 1, 0,…

Števno sekvenco zapišemo v tabelo:

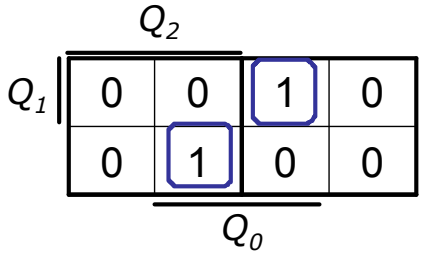
trenutno stanje			naslednje stanje			enačbe T−FF		
Q_2	Q_1	Q_0	Q_2	Q_1	Q_0	T_2	T_1	T_0
0	0	0	1	0	0	1	0	0
0	0	1	0	0	0	0	0	1
0	1	0	0	1	1	0	0	1
0	1	1	0	0	1	0	1	0
1	0	0	1	0	1	0	0	1
1	0	1	1	1	1	0	1	0
1	1	0	0	1	0	1	0	0
1	1	1	1	1	0	0	0	1

Iz tabele prehajanja stanj števca določimo enačbe T−FF:
Za T_0 narišemo Veitchev diagram. Funkcija je funkcija linearna, zato jo bomo izrazili z XOR operacijami.



$$T_0 = Q_2 \oplus Q_1 \oplus Q_0$$

Podobno za T_1 narišemo Veitchev diagram



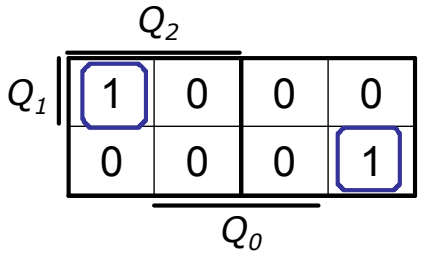
Za T_1 sledi:

$$T_1 = \overline{Q_2} \cdot Q_1 \cdot \overline{Q_0} + Q_2 \cdot \overline{Q_1} \cdot \overline{Q_0}$$
$$T_1 = \left(\overline{Q_2} \cdot Q_1 + Q_2 \cdot \overline{Q_1}\right) \cdot \overline{Q_0}$$

Operacija v oklepajih je XOR, zato enačbo lahko poenostavimo v:

$$T_1 = (Q_2 \oplus Q_1) \cdot Q_0$$

In še za T_2:



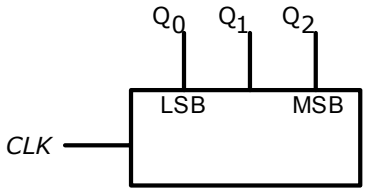
Za T_2 sledi:

$$T_2 = \overline{Q_2} \cdot \overline{Q_1} \cdot \overline{Q_0} + Q_2 \cdot Q_1 \cdot \overline{Q_0}$$
$$T_2 = \left(\overline{Q_2} \cdot \overline{Q_1} + Q_2 \cdot Q_1\right) \cdot \overline{Q_0}$$

Operacija v oklepajih je negacija XOR (za dve spremenljivki je to ekvivalenca), zato enačbo lahko poenostavimo v:

$$T_2 = \left(\overline{Q_2 \oplus Q_1}\right) \cdot Q_0$$

47. Prikažite sintezo 3−bitnega sinhronega števca *navzgor* po Graye−vi kodi s T flip−flopi in logičnimi vrati. Števec ima 3−bitni številni izhod (Q_2, Q_1, Q_0) in vhod za signal ure (CLK). Uporabite poimenovanje signalov, kot je narisano na spodnji sliki.



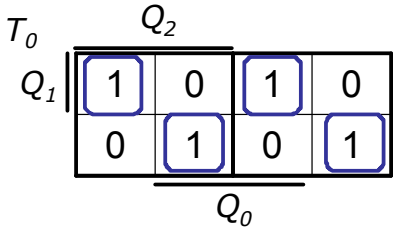
Postopek sinteze zahteva, da zapišemo tabelo prehajanja stanj števca navzgor po Gray−evi kodi. Desetiška številna sekvenca navzgor po 3−bitni Grayevi kodi se glasi: … 0, 1, 3, 2, 6, 7, 5, 4, 0…

Številno sekvenco zapišemo v tabelo:

trenutno stanje			naslednje stanje			enačbe T−FF		
Q_2	Q_1	Q_0	Q_2	Q_1	Q_0	T_2	T_1	T_0
0	0	0	0	0	1	0	0	1
0	0	1	0	1	1	0	1	0
0	1	0	1	1	0	1	0	0
0	1	1	0	1	0	0	0	1
1	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1
1	1	0	1	1	1	0	0	1
1	1	1	1	0	1	0	1	0

Iz tabele prehajanja stanj števca določimo enačbe T−FF:

Za T_0 narišemo Veitchev diagram. Funkcija je funkcija linearna, zato jo bomo izrazili z XOR operacijami.



$$T_0 = \overline{Q_2}\cdot\overline{Q_1}\cdot\overline{Q_0} + \overline{Q_2}\cdot Q_1\cdot Q_0 + Q_2\cdot\overline{Q_1}\cdot Q_0 + Q_2\cdot Q_1\cdot\overline{Q_0}$$
$$T_0 = \overline{Q_2}\cdot\left(\overline{Q_1}\cdot\overline{Q_0} + Q_1\cdot Q_0\right) + Q_2\cdot\left(\overline{Q_1}\cdot Q_0 + Q_1\cdot\overline{Q_0}\right)$$
$$T_0 = \overline{Q_2}\cdot\left(\overline{Q_1\oplus Q_0}\right) + Q_2\cdot\left(Q_1\oplus Q_0\right)$$

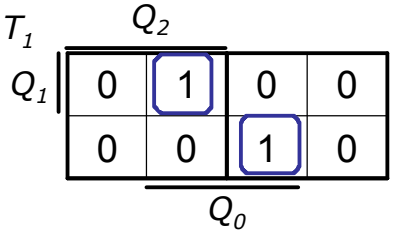
Uvedemo novo spremenljivko *x*:

$$x = Q_1\oplus Q_0$$

in jo vstavimo v izraz za T_0:

$$T_0 = \overline{Q_2}\cdot\overline{x} + Q_2\cdot x = \overline{Q_2\oplus x}$$
$$T_0 = 1\oplus Q_2\oplus Q_1\oplus Q_0$$

Podobno za T_1 narišemo Veitchev diagram:



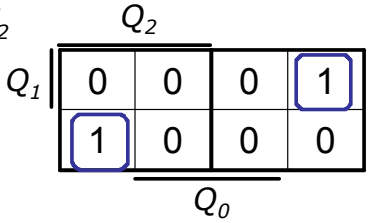
Iz diagrama za T_1 sledi:

$$T_1 = Q_2\cdot Q_1\cdot Q_0 + \overline{Q_2}\cdot\overline{Q_1}\cdot Q_0$$
$$T_1 = \left(Q_2\cdot Q_1 + \overline{Q_2}\cdot\overline{Q_1}\right)\cdot Q_0$$

Operacija v oklepajih je ekvivalenca, zato enačbo lahko poenostavimo v:

$$T_1 = \left(\overline{Q_2\oplus Q_1}\right)\cdot Q_0$$

Podobno storimo še za T_2:



Iz diagrama za T_2 sledi:

$$T_2 = \overline{Q_2}\cdot Q_1\cdot\overline{Q_0} + Q_2\cdot\overline{Q_1}\cdot\overline{Q_0}$$
$$T_2 = \left(\overline{Q_2}\cdot Q_1 + Q_2\cdot\overline{Q_1}\right)\cdot\overline{Q_0}$$
$$T_2 = \left(Q_2\oplus Q_1\right)\cdot\overline{Q_0}$$

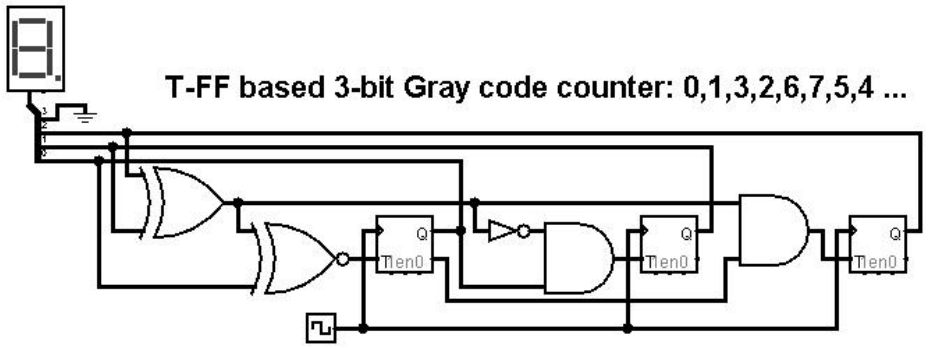
Operacija v oklepajih je negacija XOR, zato enačbo lahko poenostavimo v:

$$T_2 = (Q_2 \oplus Q_1) \cdot \overline{Q_0}$$

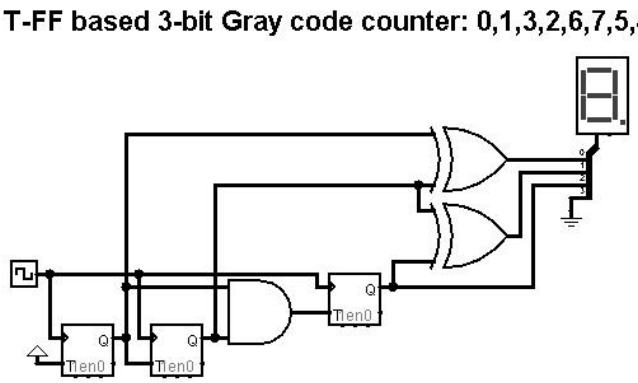
Manj potratno možnost realizacije predstavlja dvojiški 3−bitni sinhroni števec navzgor. Tak števec realiziramo s tremi T−FF in enimi AND vrati.
Nastalemu sinhronemu števcu na izhodu dodamo pretvornik kode iz dvojiškega v Gray−evo kodo z XOR vrati po enačbah:

$$G_{MSB} = B_{MSB}$$
$$G_i = B_{i+1} \oplus B_i$$

Vezje se nahaja v Logisim predlogah rešenih nalog na domači strani predmeta:
Logisim\counter\3-bit Gray code counter_revB.circ



Enostavnejšo izvedbo štetja dosežemo z uporabo sinhronega 3 bitnega dvojiškega števca in pretvornika kode iz dvojiške v Gray−evo kodo.



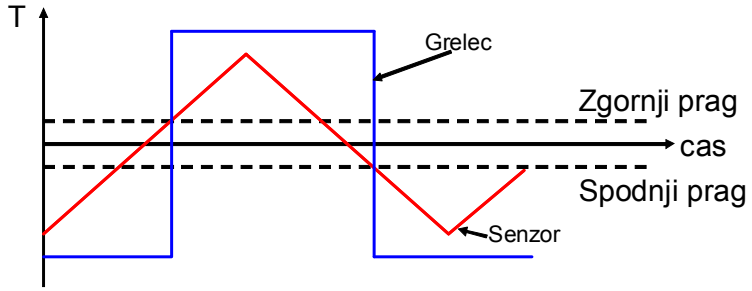
Vezje se nahaja v Logisim predlogah rešenih nalog na domači strani predmeta:
Logisim\counter\3-bit Gray code counter_revC.circ

48. Programirajte pulzno−širinski modulator (PWM) v VHDL. PWM je vezje, s katerim spreminjamo razmerje signal/pavza izhodnega signala − razmerje trajanja vrednosti signala '1' (t_1) proti trajanju vrednosti signala '0' (t_0).
Povprečna vrednost pravokotnega signala preko periode se spreminja z razmerjem $t_1/(t_1+t_0)$.
Osnovo PWM predstavlja števec navzgor po modulu (`M`). Modul števca določa periodo PWM. PWM ima primerjalni vhod (`ref`), katerega vrednost se nahaja znotraj intervala [`0… M−1`]. Trenutno stanje števca (`stevec`) primerjamo z vrednostjo (`ref`): Če je vrednost signala (`ref`) večja od trenutne vrednosti števca, potem bo izhod PWM (`pwm_out <= '1'`), sicer je (`pwm_out <= '0'`). Števec navzgor vsebuje signal za ponastavitev (`rst`) ki postavlja vsebino štetja na 0.,

Poleg osnovne uporabe (modulacija signala), se PWM pogosto uporablja za digitalno spreminjanje povprečne vrednosti signala preko periode (t_1+t_0) s čimer lahko realiziramo enostaven digitalno−analogni pretvornik (DAC), če izhodno vrednost (`pwm_out`) vodimo na vhod nizko prepustnega sita (RC). Če mejno frekvenco RC sita izberemo primerno, bo izhod sita dober približek enosmerni vrednosti povprečne napetosti preko periode PWM. Z uporabo PWM v modernih vgrajenih sistemih realiziramo enostaven DAC, katerega ločljivost je sorazmerna s številom bitov prostotekočega števca.
Večkrat takšno uporabo srečamo tudi brez izhodnega RC sita, saj krmiljeno vezje že samo po sebi ni zmožno slediti hitrim spremembam PWM in bo samo po sebi opravljalo funkcijo povprečenja. Primer takega vezja je LED. Če izhod PWM vodimo na LED, se bo njena svetlost spreminjala glede na nastavljeno PWM razmerje signal/pavza.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity pwm is
  generic( M : natural := 8
       );
  Port ( rst, clk : in STD_LOGIC;
         ref : in STD_LOGIC_VECTOR (M-1 downto 0);
         pwm_out : out STD_LOGIC;
  );
end pwm;

architecture arch of pwm is
signal stevec: STD_LOGIC_VECTOR (M-1 downto 0);
begin
process (stevec, clk, rst)
begin
  if rst = '1' then
    stevec <= (others => '0'); --zbrisi stevec
  elsif rising_edge( clk ) then
    stevec <= stevec + 1;-- pristej
  end if;
end process;
pwm_out <= '1' when ( stevec > ref ) else '0';
end arch;
```

49. V VHDL realizirajte avtomat končnih stanj, ki opravlja nalogo dvopoložajne regulacije temperature s histerezo v peči.
    Regulacija temperature je najbolj ilustrativen primer, na katerem se uporablja dvopoložajna (ON−OFF) regulacija s histerezo. Na vhodu regulatorja je senzor temperature, ki je priključen na analogno−digitalni pretvornik. Z njim zajemamo vrednost temperature.
    Dvopoložajna regulacija *brez histereze* pomeni, da bomo grelec ugasnili, ko je vrednost temperature višja od nastavljene ($T>T_{reg}$) in grelec prižgali, ko je vrednost temperature prenizka ($T<T_{reg}$). Problem te regulacije je, da ko se temperatura enkrat stabilizira ($T \approx T_{reg}$), takrat vrednost temperature zelo malo niha okoli nastavljene vrednosti in grelec krmilna elektronika grelec po nepotrebnem prižiga in ugaša. Zakaj po nepotrebnem? Regulacija je hiter proces, sprememba temperature pa je počasen proces, saj mora biti grelec prižgan nekaj časa, da se to odrazi na temperaturi. Hitro preklapljanje, ki sledi iz krmilne elektronike, hitro uniči rele.
    Da se temu izognemo, uvedemo regulacijo s histerezo, pri kateri se iz ene vrednosti nastavljene temperature oz. T_{reg} malo odmaknemo *nad* in *pod* to vrednost, tako da nastaneta spodnji prag preklopa in zgornji prag preklopa. Časovni potek delovanja regulatorja temperature s histerezo prikazuje spodnja slika, kjer imamo prikazano temperaturo senzorja (rdeče) in stanje grelca (modro). Iz delovanja vidimo, da grelec vklopi šele, ko temperatura preseže zgornji prag in izklopi šele, ko je temperatura pod spodnjim pragom.



Če želimo realizirati delovanje na zgornji sliki, bo imel avtomat končnih stanj krmilne elektronike dve stanji (Mealy−ev tip): "prizgan", "ugasnjen". Temperaturo senzorja beremo z A/D pretvornika in rezultat A/D pretvorbe v nepredznačeni obliki (`sensor`) in predstavlja vhod v regulacijo. Izhod avtomata je spremenljivka (`grelec`) v negativni logiki, s katero krmilimo rele, ki prižiga in ugaša grelec. Avtomat ima interni signal − spremenljivko (`meja`), ki hrani trenutno vrednost meje s katero primerjamo signal (`sensor`). Vrednosti, ki jih lahko (`meja`) zavzema sta dve: `sp_meja` in `zg_meja`. V našem primeru sta ti vrednosti konstantni (recimo da je vrednost T_{reg} fiksna) in ustrezata spodnjemu in zgornjemu pragu histereze na sliki.
Avtomat ima tipko za ponastavitev (`rst`) v negativni logiki s katero ga postavimo v začetno stanje "prizgan". Ob tem grelec tudi prižgemo in trenutno vrednost nastavljene meje nastavimo na `zg_meja`.
V stanju "prizgan" ostanemo toliko časa, dokler temperatura senzorja ne preseže trenutne vrednosti nastavljene meje. Ko jo enkrat preseže, postavimo spremenljivko meja na spodnjo mejo (`sp_meja`) in ugasnemo grelec ter preidemo v stanje "ugasnjen". Če temperatura ni presegla trenutne vrednosti meje, ostanemo v stanju "prizgan". Za stanje "ugasnjen" velja, da če je temperatura senzorja večja od trenutne nastavljene meje, ostajamo v stanju "ugasnjen", sicer spremenljivko trenutne

vrednosti meje postavimo na `zg_meja`, grelec prižgemo in preidemo v stanje "prizgan".

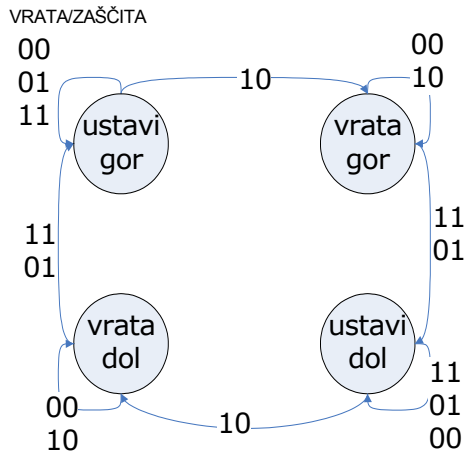
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity hysteresis is
   Port (
      clk, rst: in STD_LOGIC;
      grelec, znotraj_histereze : out STD_LOGIC;
      sensor : in STD_LOGIC_VECTOR (7 downto 0);
   );
end hysteresis;

architecture arch of hysteresis is
   TYPE stanja_regulatorja IS (ugasnjen, prizgan);
   SIGNAL y: stanja_regulatorja;
   constant zg_meja: STD_LOGIC_VECTOR(7 downto 0) := x"D0";
   constant sp_meja: STD_LOGIC_VECTOR(7 downto 0) := x"2F";
   --za hranjenje vrednosti trenutne meje
   signal meja : STD_LOGIC_VECTOR (7 downto 0) := zg_meja;
begin

process(clk, rst, meja, sensor, y)
begin
   if (rst = '0') then
      y <= prizgan; --na zacetku prizgemo grelec
      grelec <= '0'; -- prizgi grelec
      meja <= zg_meja; --in cakamo da se segreje do zg meje
   elsif ( rising_edge(clk) ) then
      CASE y IS
         WHEN prizgan =>
            IF (sensor > meja) THEN
               --odcitek adc vecji od zg. meje - prevroce
               grelec <= '1'; -- ugasni grelec
               meja <= sp_meja; -- nova meja primerjave je spodnja
               y <= ugasnjen;-- novo stanje prizgan
            ELSE
               --odcitek adc manjsi od zg. meje - se vedno prehladno
               y <= prizgan;
            END IF;

         WHEN ugasnjen =>
            IF (sensor > meja) THEN
               --odcitek adc vecji od sp. meje - se vedno prevroce
               y <= ugasnjen;
            ELSE
               --odcitek adc manjsi od sp. meje - prehladno
               grelec <= '0'; -- prizgi grelec
               meja <= zg_meja;
               y <= prizgan;
            END IF;

      END CASE;
   end if;
end process;

-- ali je regulator znotraj histereze [sp_meja ... zg_meja] ?
znotraj_histereze <= '1' when ((sensor > sp_meja ) and (sensor <
zg_meja )) else '0';

end arch;
```

50. Načrtajte krmilje za garažna vrata z uporabo avtomata končnih stanj. Garažna vrata imajo vhod VRATA ter vhod ZAŠČITA, ki postane '1' vedno ko preko motorja steče dovolj velik tok. Z meritvijo toka na motorju obenem izdelamo funkcijo detekcije obeh končnih položajev, kot tudi zaščito proti oviram na poti vrat. Vezje ima 2−biten izhod za enosmerni motor:

Koda operacije		**Operacija motorja**
OP_1	OP_0	
0	0	motor stoji
0	1	motor pomika vrata gor
1	0	motor pomika vrata dol

Če pritisnemo gumb VRATA, se vrata začno pomikati gor. Če na poti naletijo na oviro ali pa pridejo do zgornje končne lege, se motor ustavi. Če pritisnemo gumb VRATA ponovno se začnejo gibati v obratni smeri − torej dol. Krmilnik se obnaša podobno, če so vrata na poti dol.

Narišemo Moore−ov diagram stanj:



ime stanja	OP_1	OP_0
ustavi gor	0	0
vrata gor	0	1
vrata dol	1	0
ustavi dol	1	1

Iz opisa naloge je razvidno, da stanje "ustavi" ni samo eno, ker si moramo zapomniti v katero smer so se gibala vrata, da bi lahko šli v nasprotni smeri. Glede na to imamo stanji "ustavi gor", ki določa, da se bodo vrata ob naslednjem pritisku na gumb gibala gor in stanje "ustavi dol", ki določa, da se bodo vrata ob naslednjem pritisku na gumb gibala dol. Če stanja ločimo tako, potem v stanju "ustavi gor" ostajamo toliko časa, dokler ne pritisnemo VRATA in jasno na motorju ni napake, se pravi kombinacija "10". Vrata se nato pomikajo gor (preidemo v stanje "vrata gor"). V tem stanju lahko tipko spustimo in vrata se pomikajo navzgor. To se dogaja toliko časa, dokler ne naletimo na pogoj ZAŠČITA='1' (se pravi kombinaciji "11" in "01". Ko postane pogoj ZAŠČITA='1' se postavimo v stanje "ustavi dol" in v tem stanju ostajamo dokler vztraja pogoj ZAŠČITA='1' oz. dokler ne pritisnemo tipke VRATA='1' (kombinacija "10"). Takrat na podoben način preidemo v stanje "vrata dol", kjer ostanemo dokler ne naletimo na oviro (tla prostora recimo), ko preidemo v stanje "ustavi gor".
Takšna realizacija še zdaleč ni optimalna: Bolje bi bilo, če bi avtomat realizirali kot Mealy−ev tip. Dejanska realizacija ne vsebuje avtomata, temveč en T−FF in relejno logiko.

Diagram prehajanja stanj izpišemo v tabelo prehajanja stanj:

Trenutno stanje	Naslednje stanje VRATA/ZAŠČITA				Izhod (OP_1OP_0)
	00	01	10	11	
"ustavi gor"	"ustavi gor"	"ustavi gor"	"vrata gor"	"ustavi gor"	00
"vrata gor"	"vrata gor"	"ustavi dol"	"vrata gor"	"ustavi dol"	01
"ustavi dol"	"ustavi dol"	"ustavi dol"	"vrata dol"	"ustavi dol"	10
"vrata dol"	"vrata dol"	"ustavi gor"	"vrata dol"	"ustavi gor"	00

Imamo 4 stanja avtomata, torej bomo za njihovo kodiranje potrebovali dva flip−flopa. Za analizo avtomata izberemo kodiranje stanj:

Stanje	$Q_1(t)$	$Q_0(t)$
"ustavi gor"	0	0
"vrata gor"	0	1
"ustavi dol"	1	0
"vrata dol"	1	1

Predlagano kodiranje uporabimo nad tabelo prehajanja stanj in izhod zapišemo ločeno glede na vse kombinacije vhoda VRATA/ZAŠČITA.

Trenutno stanje		**Naslednje stanje $Q_i(t+1)$** VRATA/ZAŠČITA								**Izhod**	
		00		01		10		11			
$Q_1(t)$	**$Q_0(t)$**	**Q_1**	**Q_0**	**Q_1**	**Q_0**	**Q_1**	**Q_0**	**Q_1**	**Q_0**	**OP_1**	**OP_0**
0	0	0	0	0	0	0	1	0	0	0	0
0	1	0	1	1	0	0	1	1	0	0	1
1	0	1	0	1	0	1	1	1	0	0	0
1	1	1	1	0	0	1	1	0	0	1	0

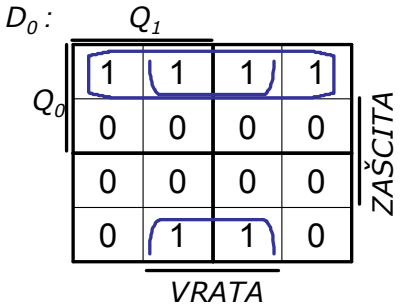
Dobljeno tabelo razdelimo na dva dela, saj bomo izhoda OP_1OP_0 obravnavali posebej. Najprej zapišemo samo aplikacijsko tabelo za enačbe vhodov D−FF pri čemer za vsak FF upoštevamo enačbo $Q_i(t+1)=D_i$:

Trenutno stanje		**Naslednje stanje** VRATA/ZAŠČITA							
		00		01		10		11	
$Q_1(t)$	**$Q_0(t)$**	**D_1**	**D_0**	**D_1**	**D_0**	**D_1**	**D_0**	**D_1**	**D_0**
0	0	0	0	0	0	0	1	0	0
0	1	0	1	1	0	0	1	1	0
1	0	1	0	1	0	1	1	1	0
1	1	1	1	0	0	1	1	0	0

Za vsak vhod FF lahko izrišemo Veitch−ev diagram 4 spremenljivk, saj sta funkciji D_1 in D_0 odvisni od vhodov VRATA in ZAŠČITA in trenutnega stanja $Q_1(t)$ $Q_0(t)$. Če predstavimo funkciji D_1 in D_0 kot funkciji spremenljivk $D_0(Q_1(t), Q_0(t)$, VRATA, ZAŠČITA) in $D_1(Q_1(t), Q_0(t)$, VRATA, ZAŠČITA), tako da je MSB $Q_1(t)$ in LSB ZAŠČITA, potem jo lahko zapišemo v obliki PDNO kot:

$$D_0 = V(2,6,10,14,4,12)$$
$$D_1 = V(7,11,10,14,5,9,8,12)$$

Za D_0 moramo narisati Veitch−ev diagram in funkcijo minimizirati:



Zapišemo enačbo za vhod D_0:

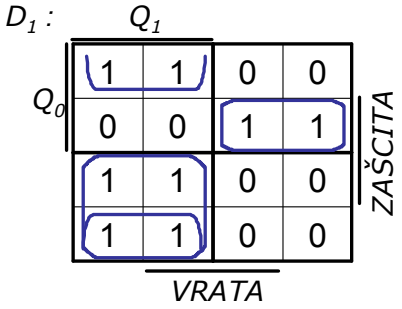
$$D_0 = VRATA \cdot \overline{ZAŠČITA} + Q_0(t) \cdot \overline{ZAŠČITA}$$

Za sintezo izhodov zapišemo samo del aplikacijske tabele, ki zajema izhoda OP_1 in OP_0:

Trenutno stanje		**Izhod**	
$Q_1(t)$	**$Q_0(t)$**	**OP_1**	**OP_0**
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	0

$$OP_0 = \overline{Q_1(t)} \cdot Q_0(t)$$
$$OP_1 = Q_1(t) \cdot Q_0(t)$$

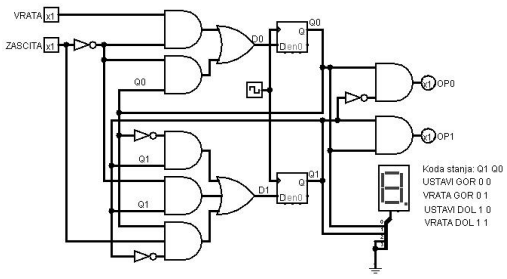
Podobno storimo za D_1:



Zapišemo enačbo za vhod D_1:

$$D_1 = Q_1(t) \cdot \overline{Q_0(t)} + Q_1(t) \cdot \overline{ZAŠČITA} +$$
$$+Q_0(t) \cdot ZAŠČITA \cdot \overline{Q_1(t)}$$

Vezje narišemo:



Vezje se nahaja v predlogah avditornih vaj na domači strani predmeta:
Logisim\fsm\garazna_vrata.circ.circ

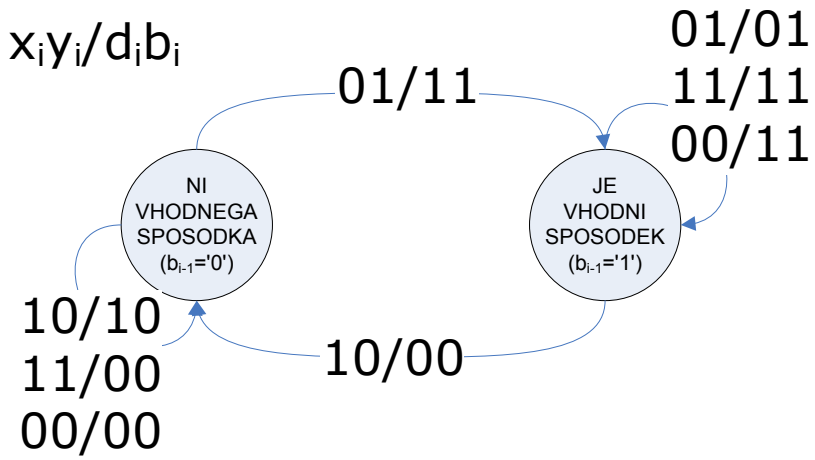
Preizkusite, ali se vezje res obnaša glede na prikazani časovni diagram v besedilu naloge. Vezje najprej postavite v začetno stanje (CTRL+R), nato poženete generator ure (pritisnete CTRL+K).

Nalogo realizirajmo še v VHDL:

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY garazna_vrata IS
   PORT(  clk, rst, vrata, zascita : IN STD_LOGIC;
          op : OUT STD_LOGIC_VECTOR (1 downto 0)
   );
END garazna_vrata;

ARCHITECTURE arch OF garazna_vrata IS
TYPE stanja_vrat IS (ustavi_gor, ustavi_dol, vrata_gor, vrata_dol);
SIGNAL y: stanja_vrat ;
BEGIN

FSM: PROCESS( rst, clk )
BEGIN
   IF rst = '1' THEN
      y <= ustavi_gor;
   ELSIF rising_edge(clk) THEN
      CASE y IS
         WHEN ustavi_gor =>
            IF (vrata = '1') and (zascita = '0') THEN
               y <= vrata_gor;
            ELSE
               y <= ustavi_gor;
            END IF;
         WHEN vrata_gor =>
            IF zascita = '1' THEN
               y <= ustavi_dol;
            ELSE
               y <= vrata_gor;
            END IF;
         WHEN ustavi_dol =>
            IF (vrata = '1') and (zascita = '0') THEN
               y <= vrata_dol;
            ELSE
               y <= ustavi_dol;
            END IF;
         WHEN vrata_dol =>
            IF zascita = '1' THEN
               y <= ustavi_gor;
            ELSE
               y <= vrata_dol;
            END IF;
         END CASE;
   END IF;
END PROCESS;
--Moore-ova izvedba
op(1) <= '1' WHEN (y = vrata_dol) or (y = ustavi_dol) ELSE '0';
op(0) <= '1' WHEN (y = vrata_gor) ELSE '0';
END arch;
```

51. Realizirajte avtomat končnih stanj, ki opravlja funkcijo serijskega odštevalnika dveh n−bitnih števil $X=x_{n-1}, \ldots. x_2, x_1, x_0$ in $Y=y_{n-1}, \ldots. y_2, y_1, y_0$, ki se pojavljata na vhodu vezja v zaporedju od LSB do MSB (i=0,1,2,…n−1), skladno s neaktivnim robom ure *CLK*.
Avtomat na izhodu generira zaporedno sekvenco razlike $D= d_{n-1}, \ldots. d_2, d_1, d_0$. =A−B. Vhoda v avtomat končnih stanj sta x_i in y_i, izhoda avtomata naj bosta razlika odštevanja d_i in izhodni sposodek b_i. Pri operaciji odštevanja stanja avtomata določite na to, kakšen je bil vhodni sposodek (b_{i-1}) pri odštevanju z nižjega mesta. Uporabite D flip−flope, ki so proženi na sprednji rob signala ure *CLK*.

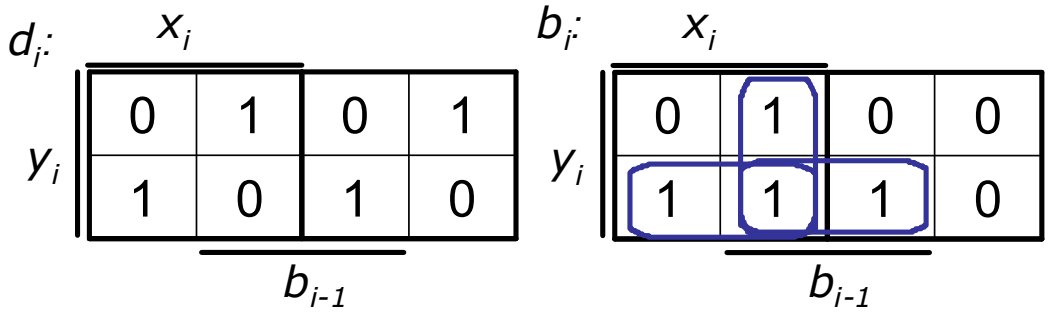
Narišemo diagram stanj Mealy-eve izvedbe avtomata:



Opis diagrama stanj: Pri odštevanju dveh bitov števil $x_i - y_i$ imamo na vhodu tudi morebitni vhodni sposodek z mesta nižje b_{i-1}.
Rezultata odštevanja dveh bitov sta izhodni sposodek b_i in bit razlike d_i. Stanji avtomata nam pravzaprav nakazujeta spremembo sposodka b_{i-1} na b_i, saj je to edina informacija, ki si jo mora avtomat "zapomniti". Razlika je kombinacijski izhod, neodvisen od izhodnega sposodka.

b_{i-1}	x_i	y_i	b_i	$d_i= (x_i - y_i) - b_{i-1}$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	0
1	0	1	0	1
1	1	0	0	0
1	1	1	1	1

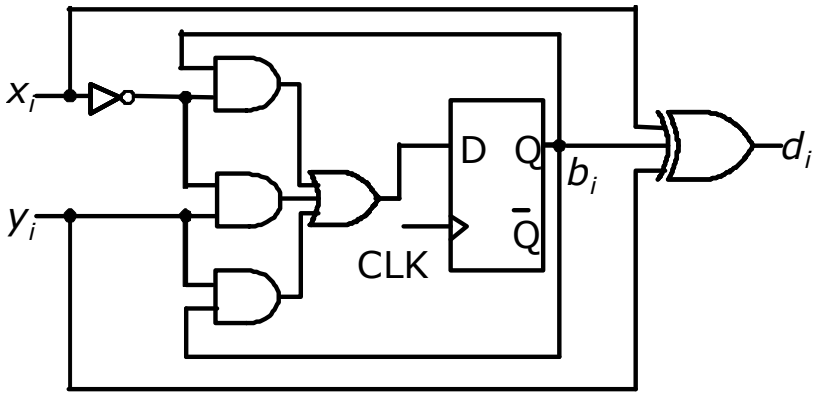
Za izhodni sposodek b_i in bit razlike d_i. narišemo v Veitch−eva diagram ter funkciji minimiziramo.



Dobimo enačbi MDNO oblik funkcij:

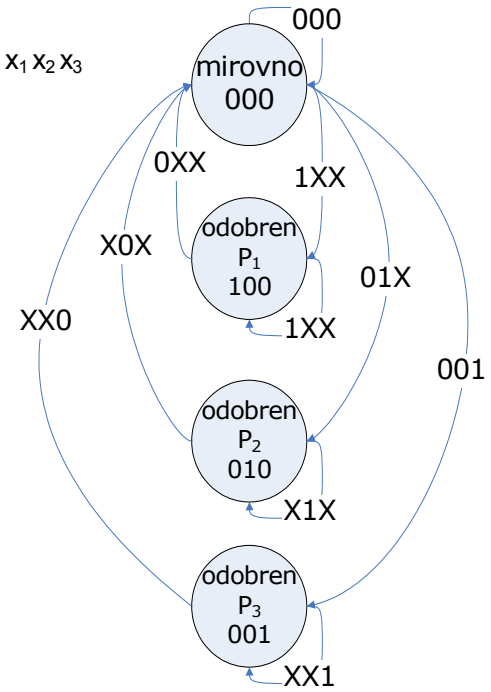
$$b_i = \overline{x_i} \cdot y_i + \overline{x_i} \cdot b_{i-1} + y_i \cdot b_{i-1}$$
$$d_i = x_i \oplus y_i \oplus b_{i-1}$$

Iz dobljenih enačb narišemo realizacijo vezja serijskega enobitnega odštevalnika:



52. Narišite diagram prehajanja stanj avtomata, ki bdi nad izvajanjem treh procesov P_1, P_2 in P_3 (ang. arbiter state machine). Avtomat skrbi, da se lahko izvaja le en proces naenkrat. Procesi imajo med seboj različne prioritete izvajanja: P_1 ima najvišjo prioriteto, sledi mu P_2, medtem ko ima P_3 najnižjo prioriteto. Avtomat ne odobri izvajanja procesa z nižjo prioriteto, če se hkrati pojavita zahtevi za izvajanje dveh procesov. Za vzdrževanja stanja izvajanja procesa in prenehanje izvajanja procesa te omejitve ni.
Avtomat ima tri vhode x_1 x_2 x_3, ki pomenijo zahtevo (ang. request) po izvajanju ustreznega procesa in tri izhode y_1 y_2 y_3. Ustrezen y_i, postane '1' takrat ko avtomat odobri (ang. grant) izvajanje i−tega procesa. Proces se izvaja dokler obstaja zahteva za to (x_i='1'). Naslednji proces se lahko začne izvajati šele, ko zahteva po izvajanju trenutno odobrenega procesa izgine (x_i='0').

Narišemo Moore−ov diagram stanj:



Iz opisa naloge je razvidno, da ima vsak proces svoje stanje: "odobren". Imamo tudi mirovno stanje, v katerega se avtomat postavi ob resetiranju. V mirovnem stanju ni odobren noben proces. Izhodi v tem procesu so zato $y_1y_2y_3$=000. Izhodi preostalih stanj "odobren" so kodirani po kodi ena naenkrat.

V stanje "odobren" določenega procesa preidemo, če je ustrezni vhod x_i='1', vendar ob pogoju, da imajo vsi nadrejeni procesi x_i='0'. Zato, da bi odobrili izvajanje P_3, morata biti zahtevi za P_1 in P_2 enaki '0' − od tod sledi pogoj, $x_1x_2x_3$=001. Za izvajanje procesa P_2 pa je pomembno samo, da je x_1='0'. Stanje P_3 takrat ni bistveno − od tod pogoj $x_1x_2x_3$=01X. Ko v določeno stanje "odobren" že preidemo, tam vztrajamo, dokler vztraja x_i='1', četudi se vmes pojavita zahtevi po ostalih (podrejenih ali nadrejenih) procesih. Iz določenega stanja se premaknemo v mirovno stanje takrat, ko postane x_i='0'. Tudi takrat velja, da stanja zahtev po izvajanju ostalih procesov niso bistvena.

53. Z uporabo D flip−flopov, ki so proženi na sprednji rob signala ure *CLK*, načrtajte Moore−ov avtomat končnih stanj, ki *sproti* izračunava ostanek pri deljenju dvojiškega števila na vhodu s 7. Na vhodu se število pojavlja kot zaporedje bitov (najprej MSB).
Če ima število x na vhodu pri deljenju s 7 ostanek r_i, potem se novi ostanek števila (r_{i+1}), ki ima dodan en bit n, izračuna kot: $r_{i+1}=(2\cdot r_i+n) \bmod 7$, kjer je mod operator izračuna ostanka.
Primer: Naj bo število, katerega ostanek pri deljenju s 7 iščemo, $x=183_{10}=10110111_2$.
Začetni ostanek pri deljenju je $r_0=0$.
Na vhodu avtomata se pojavi MSB $n=1$. Novi ostanek $r_1=(2\cdot r_0+n) \bmod 7=1_{10}$.
Sledi bit $n=0$: Prejšnji ostanek je ($r_1=1_{10}$), novi je $r_2=(2\cdot 1_{10}+0) \bmod 7= 2_{10}$.
Sledi bit $n=1$: Prejšnji ostanek je ($r_2=2_{10}$), novi je $r_3=(2\cdot 2_{10}+1) \bmod 7= 5_{10}$.
Sledi bit $n=1$: Prejšnji ostanek je ($r_3=5_{10}$) novi je $r_4=(2\cdot 5_{10}+1) \bmod 7= 4_{10}$.
Sledi bit $n=0$: Prejšnji ostanek je ($r_4=4_{10}$) novi je $r_5=(2\cdot 4_{10}+0) \bmod 7= 1_{10}$.
Sledi bit $n=1$: Prejšnji ostanek je ($r_5=1_{10}$) novi je $r_6=(2\cdot 1_{10}+1) \bmod 7= 3_{10}$.
Sledi bit $n=1$: Prejšnji ostanek je ($r_6=3_{10}$) novi je $r_7=(2\cdot 3_{10}+1) \bmod 7= 0_{10}$.
Sledi bit $n=1$: Prejšnji ostanek je ($r_7=0_{10}$) novi je $r_8=(2\cdot 0_{10}+1) \bmod 7= 1_{10}$.
Če bi izračunali ostanek pri deljenju po ustaljeni poti, bi dobili:
$183_{10} \bmod 7=1_{10}$.

Najprej pokažimo trditev iz navodila naloge: Naj ima število x na vhodu pri deljenju s 7 ostanek r in kvocient q. Torej: $x=7\cdot q+r$. Če temu številu dodamo bit (n) na zadnjem mestu, potem moramo vsebino x pomakniti eno mesto v levo (x množimo z 2). Novo število zapišemo kot $2\cdot x+n$. Če v dobljeni izraz vstavimo izraz za x, potem dobimo $2\cdot x+n=2\cdot(7\cdot q+r)+n=14\cdot q+2\cdot r+n$. Novi kvocient je očitno deljiv z 2 *(14/7)*, torej je novi ostanek $(2\cdot r+n)$.
Stanja Moore−ovega avtomata naj predstavljajo vsi možni ostanki pri deljenju s 7 − označimo jih od 0 do 6. Tabelo prehajanja stanj lahko zapišemo tako, da imamo na vhodu trenutni ostanek (r_i), iz tega stanja preidemo v novo stanje − nov ostanek (r_{i+1}) ob pogoju, če se na vhodu pojavi bit $n=0$ ali $n=1$. Naslednja stanja avtomata določimo kar iz formule $r_{i+1}=(2\cdot r_i+n) \bmod 7$.

Tabela 2: Tabela naslednjih stanj avtomata $r_{i+1}=(2\cdot r_i+n) \bmod 7$.

r_i	$n=0$	$n=1$
0_{10}	0_{10}	1_{10}
1_{10}	2_{10}	3_{10}
2_{10}	4_{10}	5_{10}
3_{10}	6_{10}	0_{10}
4_{10}	1_{10}	2_{10}
5_{10}	3_{10}	4_{10}
6_{10}	5_{10}	6_{10}

Tabelo naslednjih stanj obrnemo tako, da je vhod (n) najbolj pomembna spremenljivka. Za kodiranje stanj izberemo kar dvojiško kodiranje in zapišemo novo tabelo prehajanja stanj.

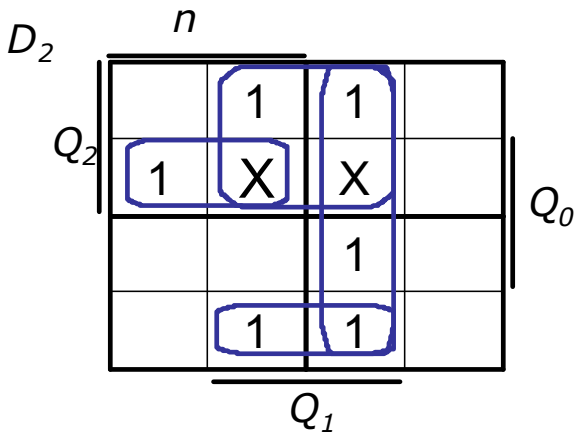
Tabela 3: Tabela prehajanja stanj avtomata $r_{i+1}=(2\cdot r_i+n)$ mod 7.

n	Q_2	Q_1	Q_0	Q_2	Q_1	Q_0	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0	1	0
0	0	1	0	1	0	0	1	0	0
0	0	1	1	1	1	0	1	1	0
0	1	0	0	0	0	1	0	0	1
0	1	0	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	0	1
0	1	1	1	X	X	X	X	X	X
1	0	0	0	0	0	1	0	0	1
1	0	0	1	0	1	1	0	1	1
1	0	1	0	1	0	1	1	0	1
1	0	1	1	0	0	0	0	0	0
1	1	0	0	0	1	0	0	1	0
1	1	0	1	1	0	0	1	0	0
1	1	1	0	1	1	0	1	1	0
1	1	1	1	X	X	X	X	X	X

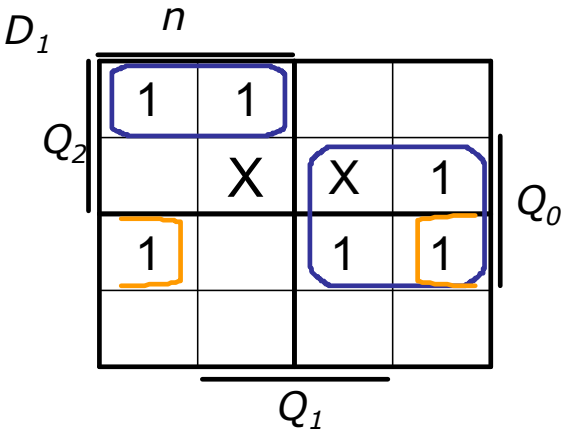
Iz tabele prehajanja stanj zapišemo enačbe D−FF:

$$D_2 = V(2, 3, 6, 10, 13, 14) \text{ in } V_x(7, 15)$$
$$D_1 = V(1, 3, 5, 9, 12, 14) \text{ in } V_x(7, 15)$$
$$D_0 = V(4-6, 8-10) \text{ in } V_x(7, 15)$$

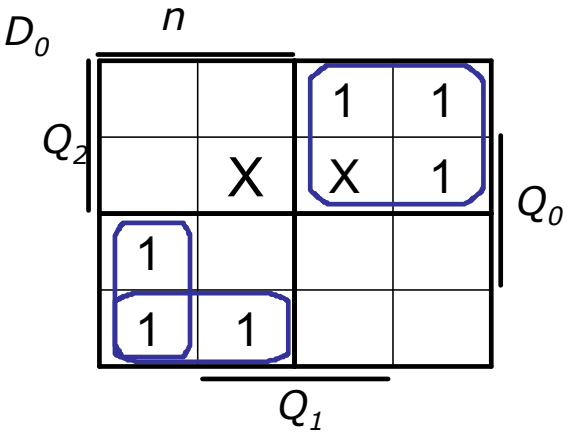
Za D_2 narišemo Veitchev diagram.



Podobno za D_1 narišemo Veitchev diagram.



In tudi za D_0 narišemo Veitchev diagram.



54. Minimizirajte podani avtomat končnih stanj z uporabo metode z razdelki ter zapišite tabelo prehajanja stanj nastalega minimalnega avtomata.

Trenutno stanje	***Naslednje stanje***		***Izhod***
	x=0	*x=1*	*z*
A	B	D	0
B	A	C	1
C	B	D	0
D	E	C	1
E	E	D	1

V prvi iteraciji zberemo skupaj vsa stanja z različnimi izhodi: P_1=(AC)(BDE)

- Pregledamo vsa naslednja stanja pri vhodu 0 in 1 v vsakem bloku:
  - Blok (AC):
    - Naslednja stanja pri x=0 (BB)
    - Naslednja stanja pri x=1 (DD)
      Vsa naslednja stanja so iz istega razdelka. Naslednja stanja so celo enaka, torej lahko stanji A in C združimo.
  - Blok (BDE):
    - Naslednja stanja pri x=0 (AEE)
      Stanja niso v istem razdelku: Problem je pri stanju A in stanju E. Stanje B ima za naslednje stanje A, ki je iz drugega razdelka, torej bo stanje B v novi iteraciji v svojem razdelku.
    - Naslednja stanja pri x=1 (CCD)
      Stanje E ima za naslednje stanje D, ki je iz drugega razdelka,. torej bo stanje E v novi iteraciji v svojem razdelku.

Naslednja iteracija bo torej P_2=(A)(B)(D)(E). Ta iteracija je tudi zadnja, saj so vsa stanja vsaka v svojem razdelku in nadaljnje razdeljevanje ni možno. Novo dobljena tabela prehajanja stanj se glasi:

Trenutno stanje	***Naslednje stanje***		***Izhod***
	x=0	*x=1*	*z*
A	B	D	0
B	A	A	1
D	E	A	1
E	E	D	1

55. Z uporabo T flip−flopov, ki so proženi na sprednji rob signala ure *CLK*, načrtajte Moore−ov avtomat končnih stanj, ki deluje kot 3−bitni števec, ki ima vhod za način štetja M (ang. mode); če je M=0, potem števec šteje po dvojiškem zaporedju navzgor, če je M=1 po Gray−evem zaporedju navzdol.

Postopek sinteze zahteva, da zapišemo tabelo prehajanja stanj števca navzdol po Gray−evi kodi, če je M=1 in dvojiško, če je M=0. Desetiška števna sekvenca po 3−bitni Grayevi kodi navzdol se glasi: … 4, 5, 7, 6, 2, 3, 1, 0, 4, …

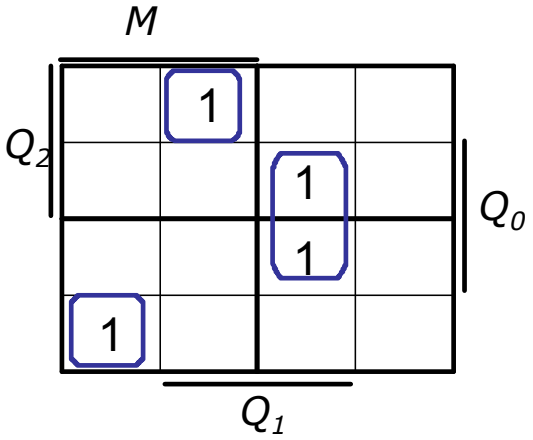
Števno sekvenco zapišemo v tabelo:

M	Q_2	Q_1	Q_0	Q_2	Q_1	Q_0	T_2	T_1	T_0
0	0	0	0	0	0	1	0	0	1
0	0	0	1	0	1	0	0	1	1
0	0	1	0	0	1	1	0	0	1
0	0	1	1	1	0	0	1	1	1
0	1	0	0	1	0	1	0	0	1
0	1	0	1	1	1	0	0	1	1
0	1	1	0	1	1	1	0	0	1
0	1	1	1	0	0	0	1	1	1
1	0	0	0	1	0	0	1	0	0
1	0	0	1	0	0	0	0	0	1
1	0	1	0	0	1	1	0	0	1
1	0	1	1	0	0	1	0	1	0
1	1	0	0	1	0	1	0	0	1
1	1	0	1	1	1	1	0	1	0
1	1	1	0	0	1	0	1	0	0
1	1	1	1	1	1	0	0	0	1

Iz tabele prehajanja stanj števca zapišemo enačbe T−FF:

$$T_2 = V(3, 7, 8, 14)$$
$$T_1 = V(1, 3, 5, 7, 11, 13)$$
$$T_0 = V(0-7, 9, 10, 12, 15)$$

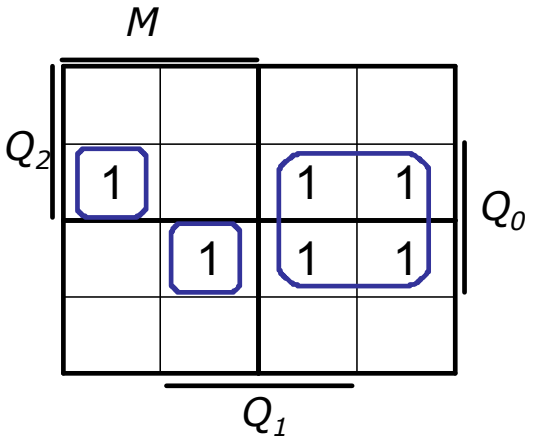
Za T_2 narišemo Veitchev diagram.



Funkcijo poskusimo minimizirati tako, da poiščemo diagonalni vzorec, ki bi lahko pomenil XOR ali EQU operacijo:

$$T_2 = Q_1 \cdot Q_0 \cdot \overline{M} + Q_2 \cdot Q_1 \cdot \overline{Q_0} \cdot M + \overline{Q_2} \cdot \overline{Q_1} \cdot \overline{Q_0} \cdot M$$
$$T_2 = Q_1 \cdot Q_0 \cdot \overline{M} + \left(Q_2 \cdot Q_1 + \overline{Q_2} \cdot \overline{Q_1}\right) \cdot \overline{Q_0} \cdot M$$
$$T_2 = Q_1 \cdot Q_0 \cdot \overline{M} + \left(\overline{Q_2 \oplus Q_1}\right) \cdot \overline{Q_0} \cdot M$$

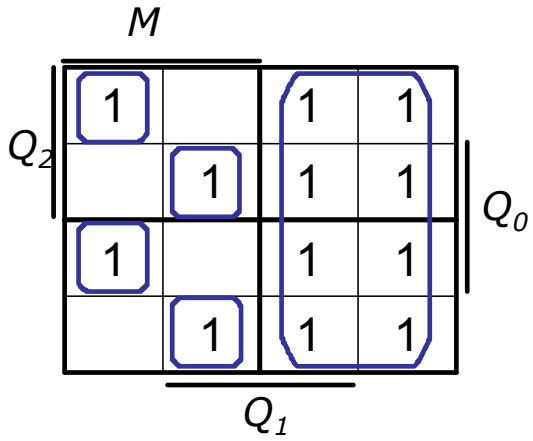
Podobno za T_1 narišemo Veitchev diagram.



$$T_1 = Q_0 \cdot \overline{M} + Q_2 \cdot \overline{Q_1} \cdot Q_0 \cdot M + \overline{Q_2} \cdot Q_1 \cdot Q_0 \cdot M$$
$$T_1 = Q_0 \cdot \overline{M} + \left(Q_2 \cdot \overline{Q_1} + \overline{Q_2} \cdot Q_1\right) \cdot Q_0 \cdot M$$
$$T_1 = Q_0 \cdot \overline{M} + (Q_2 \oplus Q_1) \cdot Q_0 \cdot M$$
$$T_1 = Q_0 \cdot \left(\overline{M} + (Q_2 \oplus Q_1) \cdot M\right)$$

Za T_1 naredimo preizkus pravilnosti: Iz diagrama za T_1 sledi, da če je M=0 in če je Q_0=1 je rezultat '1' (leve 4 enice v diagramu) in če je M=1 in če je Q_0=1 (desni dve enici) je funkcija kar XOR spremenljivk Q_2 in Q_1.

Tudi za T_0 narišemo Veitchev diagram:



Enačbo za T_0 lahko zapišemo z malo razmišljanja. Če je M=0, potem je T_0 kar enak '1', če je M=1 pa je XOR spremenljivk Q_2, Q_1 in Q_0. Pokažimo vseeno, da napisano drži.

$$T_0 = \overline{M} + M \cdot \left(Q_2 \cdot \overline{Q_1} \cdot \overline{Q_0} + Q_2 \cdot Q_1 \cdot Q_0 + \overline{Q_2} \cdot \overline{Q_1} \cdot Q_0 + \overline{Q_2} \cdot Q_1 \cdot \overline{Q_0}\right)$$

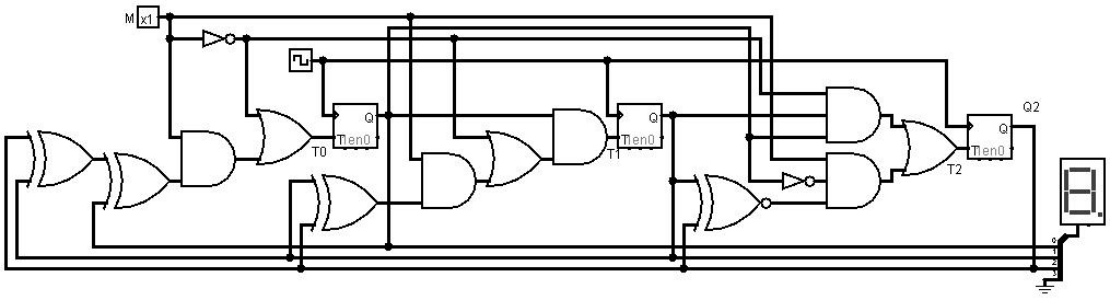
$$T_0 = \overline{M} + M \cdot \left(Q_2 \cdot \left(\overline{Q_1} \cdot \overline{Q_0} + Q_1 \cdot Q_0\right) + \overline{Q_2} \cdot \left(\overline{Q_1} \cdot Q_0 + Q_1 \cdot \overline{Q_0}\right)\right)$$

$$T_0 = \overline{M} + M \cdot \left(Q_2 \cdot \left(\overline{Q_1 \oplus Q_0}\right) + \overline{Q_2} \cdot \left(Q_0 \oplus Q_1\right)\right)$$

$$T_0 = \overline{M} + M \cdot \left(Q_2 \oplus \left(Q_1 \oplus Q_0\right)\right)$$

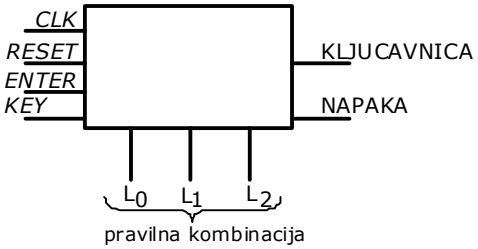
Iz enačb za T_2, T_1 in T_0 narišemo realizacijo vezja.

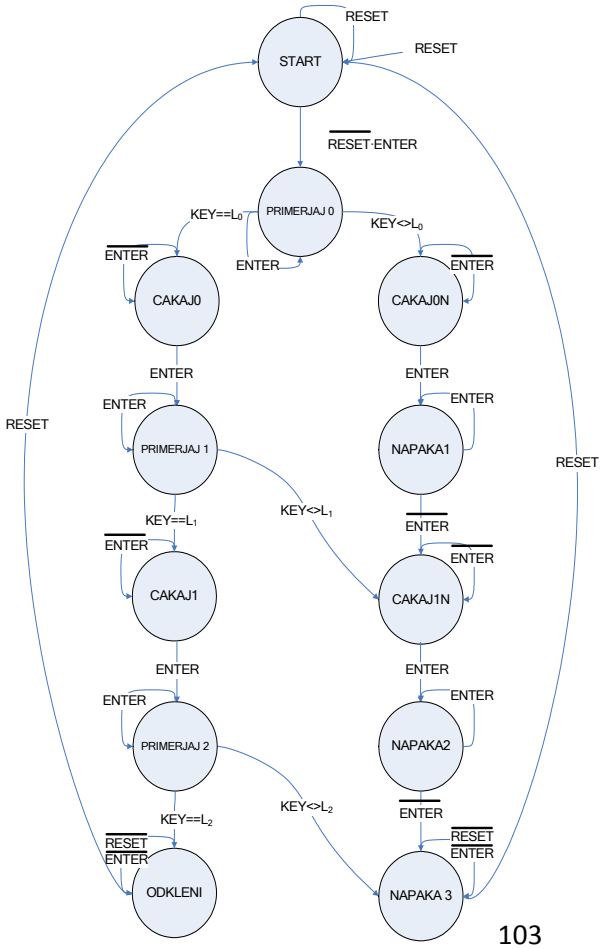
Opis delovanja in vezje števca, ki navzgor šteje dvojiško, navzdol po Graye−evi kodi je v predlogah vaj na domači strani predmeta v imeniku Logisim\counter\ counter_3_bit_up_binary_down_gray_using_T_FF.circ



56. Narišite diagram stanj Moore−ovega avtomata končnih stanj, ki deluje kot 3−bitna sekvenčna ključavnica, sestavljena iz mest (L_0, L_1 in L_2). Ključavnica ima tipko za ponastavitev (*RESET*), ki postavlja ključavnico v začetno stanje, tipko (*ENTER*) za vnos mesta kombinacije in preklopnik (*KEY*) za vrednost mesta kombinacije. Izhod je (*KLJUCAVNICA='1'*), ko uporabnik vnese pravilno kombinacijo. Izhod je (*NAPAKA='1'*), če je vnesena kombinacija napačna. Kombinacijo, ki odklene ključavnico, nastavljamo s preklopniki L_0, L_1 in L_2. Uporabo ključavnice povzema spodnje zaporedje:
    1.) Uporabnik pritisne *RESET*
    2.) S preklopnikom *KEY* nastavi vrednost mesta kombinacije odklepanja
    3.) Pritisne *ENTER*
    4.) Ko *ENTER* spusti, se izvede se primerjava i−tega bita ($KEY \equiv L_i$)
    5.) Dvakrat ponovi korake 2 do 4 za mesto primerjave *i = 1 in 2*
    6.) Če je vnesena tribitna kombinacija pravilna, se postavi izhod *KLJUCAVNICA='1'*, sicer se takrat postavi izhod *NAPAKA='1'*.
    Izhod ostane '1' samo po vnosu vseh bitov kombinacije, sicer je '0'.
    7.) Samo pritisk na *RESET* vrne uporabnika na prvi korak.

    Če se uporabnik med vnosom kombinacije zmoti, mora vsakič vnesti vse tri bite kombinacije in pritisniti *RESET* preden lahko vnaša novo kombinacijo.





Enostavni avtomati imajo malo vhodov, zato lahko po teoriji opišemo vse vhodne kombinacije v vsakem stanju avtomata, kot smo naredili v primeru detekcije zaporedja.

V kruti realnosti imamo opravka z avtomati končnih stanj, ki imajo veliko vhodov, zato namesto vseh kombinacij pišemo samo logični pogoj, ki je potreben za prehajanje iz enega stanja v drugo in pogoj, ki mora biti izpolnjen da avtomat ostane v istem stanju. Tak logični pogoj predstavlja novo spremenljivko, ki jo vodimo na vhod avtomata. Opisani primer ima veliko vhodov (*RESET, ENTER, KEY, L_0, L_1, L_2*). Kot ilustracijo povedanega za prehod iz stanja *START* pišemo pogoj, da uporabnik *ni* pritisnil tipke *RESET* **in** *je* pritisnil tipko

ENTER, kar zapišemo kot dvovhodno konjunkcijo (*AND*) spremenljivk *RESET='0' AND ENTER='1'*, oz. kot novo spremenljivko (RESET'·ENTER='1').
Naslednji primer uvedbe nove spremenljivke, ki zajema prehod v novo stanje in obstanek v stanju, je primerjava enakosti na treh mestih ($KEY==L_i$) oz. neenakosti ($KEY<>L_i$), kar izvedemo z dvovhodnimi XOR (različen) oz. z XNOR vrati (enak). Izhod posameznih XOR oz. XNOR vrat tvori novo vhodno spremenljivko, ki jo zapišemo kot ($KEY==L_i$) oz. ($KEY<>L_i$) in predstavlja nov vhod v avtomat. Iz stanja *PRIMERJAJ0* preidemo v stanje *CAKAJ0*, ko postane izhod operacije (*KEY XNOR* L_0) enak '1'. Avtomat ostane v danem stanju primerjave, dokler uporabnik ni spustil tipke *ENTER*. Šele ob tem, ko tipko *ENTER* spusti (‾_ prehod), se izvede primerjava vnosa: Od tod sledi zaporedje izvajanja pogojev − pogoj, da je *ENTER* spuščen, je *nadrejen* pogoju primerjave.
Omenjeni postopek se ponavlja za vsa tri mesta kombinacije. Pred vsako primerjavo je stanje čakanja, v katerem avtomat stoji in čaka ponoven pritisk tipke *ENTER* (_/‾ prehod) nakar preide v stanje primerjave naslednjega mesta kombinacije. Na zadnjem mestu primerjave preide v stanje *ODKLENI*, kjer postavi izhod *KLJUCAVNICA='1'*.
Če se uporabnik pri vnašanju zmoti, preide iz stanja *PRIMERJAJ0* v sekvenco stanj napake (*CAKAJ0N*), po kateri mora vnesti še dve mesti kode (lahko samo dvakrat pritisne *ENTER*) in šele nato preide v stanje *NAPAKA3*, v katerem postavi izhod *NAPAKA='1'*. Ta izhod je sicer enak '0', a je zaradi preglednosti avtomata izpuščen. Podobno velja za izhod *KLJUCAVNICA*. Realni avtomati so za klasično analizo res prevelik zalogaj, zato jih praviloma opisujemo v VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity three_bit_keylock is
   Port (  KLJUCAVNICA, NAPAKA : out STD_LOGIC;
           RESET, KEY, ENTER, L0, L1, L2, CLK : in STD_LOGIC
     );
end three_bit_keylock;
architecture arch of three_bit_keylock is
   type fsm_states is (
     START, PRIMERJAJ0, CAKAJ0, PRIMERJAJ1, CAKAJ1, PRIMERJAJ2,
     ODKLENI, CAKAJ0N, NAPAKA1, CAKAJ1N, NAPAKA2, NAPAKA3);
   signal state: fsm_states;
begin
   PROCESS( RESET, KEY, ENTER, L0, L1, L2, CLK )
     BEGIN
       IF RESET = '1' THEN
            state <= START;
       ELSIF rising_edge(CLK) THEN
         CASE state IS
           WHEN START =>
             IF ( RESET= '0' ) AND ( ENTER = '1' ) THEN
               state <= PRIMERJAJ0; --pritisnil enter
             ELSE
               state <= START; -- ostani tu
             END IF;
           WHEN PRIMERJAJ0 =>
             IF (ENTER = '0') THEN
               IF ( KEY XNOR L0 ) = '1' THEN
                 state <= CAKAJ0; --pravilno vnesel prvi bit
               ELSE
                 state <= CAKAJ0N;  --napacno vnesel prvi bit
               END IF;
             ELSE
                  state <= PRIMERJAJ0; --ni spustil enter
             END IF;
           WHEN CAKAJ0 =>
             IF (ENTER = '1') THEN
               state <= PRIMERJAJ1; --ni spustil enter
             ELSE
               state <= CAKAJ0; -- ostani tu
             END IF;
```

```
            WHEN PRIMERJAJ1 =>
               IF (ENTER = '0') THEN
                  IF ( KEY XNOR L1 ) = '1' THEN
                     state <= CAKAJ1; --pravilno vnesel drugi bit
                  ELSE
                     state <= CAKAJ1N;  --napacno vnesel drugi bit
                  END IF;
               ELSE
                  state <= PRIMERJAJ1; --ni spustil enter
               END IF;
            WHEN CAKAJ1 =>
               IF (ENTER = '1') THEN
                  state <= PRIMERJAJ2; --ni spustil enter
               ELSE
                  state <= CAKAJ1; -- ostani tu
               END IF;
            WHEN PRIMERJAJ2 =>
               IF (ENTER = '0') THEN
                  IF ( KEY XNOR L2) = '1' THEN
                     state <= ODKLENI;  --pravilno vnesel tretji bit
                  ELSE
                     state <= NAPAKA3;  --napacno vnesel tretji bit
                  END IF;
               ELSE
                  state <= PRIMERJAJ2; --ni spustil enter
               END IF;
            WHEN ODKLENI =>
               IF ( RESET = '1') THEN
                  state <= START;  -- ko pritisne reset
               ELSE
                  state <= ODKLENI;  -- ostani tu
               END IF;
            WHEN CAKAJ0N =>
               IF (ENTER = '1') THEN
                  state <= NAPAKA1;  --ni spustil enter
               ELSE
                  state <= CAKAJ0N;  --ostani tu
               END IF;
            WHEN NAPAKA1 =>
               IF (ENTER = '0') THEN
                  state <= CAKAJ1N;  --ni spustil enter
               ELSE
                  state <= NAPAKA1;  --ostani tu
               END IF;
            WHEN CAKAJ1N =>
               IF (ENTER = '1') THEN
                  state <= NAPAKA2;  --ni spustil enter
               ELSE
                  state <= CAKAJ1N;  --ostani tu
               END IF;
            WHEN NAPAKA2 =>
               IF (ENTER = '0') THEN
                  state <= NAPAKA3;  --ni spustil enter
               ELSE
                  state <= NAPAKA2;  --ostani tu
               END IF;
            WHEN NAPAKA3 =>
               IF (RESET = '1') THEN
                  state <= START;  -- dokler ne pritisne reset
               ELSE
                  state <= NAPAKA3;  --ostani tu
               END IF;
         END CASE;
      END IF;
   END PROCESS;
   KLJUCAVNICA <= '1' WHEN state = ODKLENI ELSE '0';
   NAPAKA <= '1' WHEN (state = NAPAKA3) ELSE '0'; -- Moore-ov avtomat
end arch;
```

Ustvarite nov projekt, kopirajte kodo v VHD datoteko in sestavite (*.tbw) datoteko. V simulaciji dodajte interno spremenljivko "`state`" in preverite pravilnost delovanja.

57. Realizirajte filter, ki bo na izhodu tvoril tekoče povprečje (ang. moving average filter) zadnjih 8 vzorcev 8−bitnih vhodnih podatkov z analogno−digitalnega pretvornika. Rezultat tekočega povprečja pretvorbe vzorcev z analogno−digitalnega pretvornika prikažite na LED diodah.

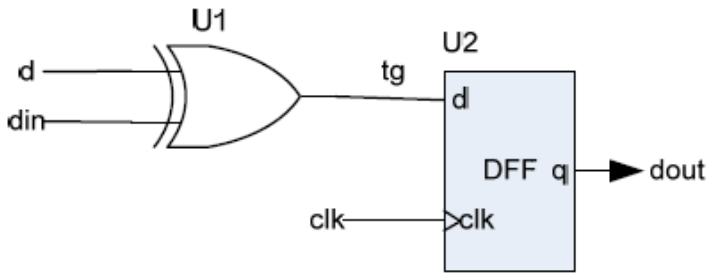
Filter tekočega povprečja vsebuje polje vhodnih podatkov (`mav_filter`). Zato, da bi izračunali trenutno vrednost izhodnega podatka, moramo vsebino polja (`mav_filter`) sešteti, ter deliti s številom vzorcev. Ker je deljenje običajno kompleksna operacija, dimenzioniramo tovrstne filtre tako da je število vzorcev v polju filtra 2^N zato, da deljenje lahko izvedemo kot pomikanje v desno. Po opravljeni operaciji tvorbe tekočega povprečja moramo vsebino filtra premakniti za eno mesto: Zadnji vzorec v polju zato zavržemo, na mesto prvega pa vstavimo novo zajet vzorec z analogno−digitalnega pretvornika.
Vsoto elementov polja filtra shranite v spremenljivki (`vsotafiltra`)
Preberite vrednost A/D pretvornika (`adc`), jo vstavite v polje vhodnih podatkov (`mav_filter`) na najnižje mesto, izračunajte tekoče povprečje po opisanem postopku. Rezultat tekočega povprečja shranite v spremenljivko (`leds`). Vezje vsebuje še signal (`rst`), s katerim postavite vsebino polja vhodnih podatkov (`mav_filter`) na ničelne vrednosti. Do i−tega elementa polja vhodnih podatkov (`mav_filter`) dostopate z uporabo oklepajev in indeksa (`mav_filter(i)`).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity filter is
 port (
     rst : in STD_LOGIC;
     vzorec : in  STD_LOGIC_VECTOR (7 downto 0); -- adc odcitek
     nTRIGGER : in STD_LOGIC;  --signal za zajem vzorca
     izhod_filtra : out  STD_LOGIC_VECTOR (7 downto 0)
     );
end;
architecture arch of filter is
type vzorci is array (0 to 7) of STD_LOGIC_VECTOR(7 downto 0);
signal mav_filter : vzorci;
signal vsotafiltra : STD_LOGIC_VECTOR(10 downto 0); -- 8 bitov + 3
biti

begin
   process(rst, nTRIGGER)
   begin
     if (rst = '0') then
       clearall: FOR i IN 0 TO 7 GENERATE
            mav_filter(i) <= (others =>'0'); --ponastavitev vseh
       END GENERATE;
     elsif(falling_edge(nTRIGGER)) then
       mav_filter(0) <= vzorec;       --pomikalni register - vstavi
nov vzorec
       mav_filter(7 downto 1) <= mav_filter(6 downto 0);
     end if;
   end process;
vsotafiltra <=("000" & mav_filter(0)) + ("000" & mav_filter(1)) +
     ("000" & mav_filter(2)) + ("000" & mav_filter(3)) +
     ("000" & mav_filter(4)) + ("000" & mav_filter(5)) +
     ("000" & mav_filter(6)) + ("000" & mav_filter(7)); --tvori vsoto
--deljenje z 8 je pomik za 3 bite desno
izhod_filtra <= vsotafiltra(10 downto 3);
end arch;
```

58. Narišite rezultat sinteze podane kode VHDL. Arhitekturi elementov "`dff`" in "`xor2`" sta podani drugje. Komponente realizacije predstavite s pravokotniki, če njihova realizacija ni razvidna iz podane kode. Nad posamezne komponente realizacije vpišite ime komponente, v pravokotnik pa ime entitete. Nad morebitnimi večbitnimi vodili označite koliko bitov imajo.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
ENTITY test IS
PORT( din, clk: in STD_LOGIC;
   dout : out STD_LOGIC);
END test
ARCHITECTURE a OF test IS
   COMPONENT dff IS
      PORT( d,clk : in STD_LOGIC;
         q : OUT STD_LOGIC);
   END COMPONENT;
   COMPONENT xor2 IS
      PORT( a,b : in STD_LOGIC;
         y : OUT STD_LOGIC);
   END COMPONENT;
SIGNAL tg : STD_LOGIC;
BEGIN
   u1: xor2 PORT MAP(a=>dout, b=>din, y=>tg);
   u2: dff PORT MAP(d=>tg, clk=>clk, q=>dout);
END a;
```

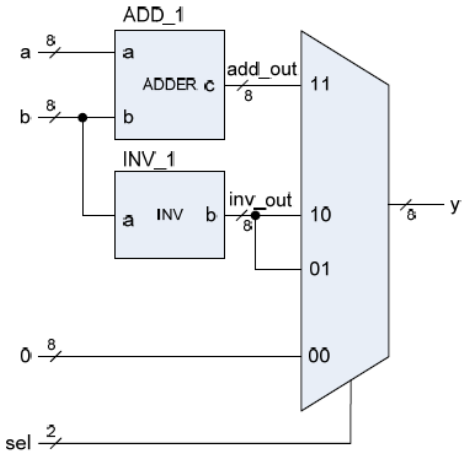
Kodo VHDL si lahko predstavljamo kot povezavo dvovhodnih XOR vrat (U1) in D−FF (U2). V povezovalnem stavku je uporabljeno imensko povezovanje (ang. named association).



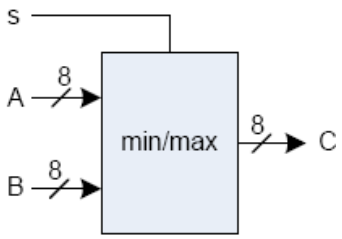
59. Narišite rezultat sinteze podane kode VHDL. Arhitekturi elementov *"adder"* in *"inv"* sta podani drugje. Komponente realizacije predstavite s pravokotniki, če njihova realizacija ni razvidna iz podane kode. Nad posamezne komponente realizacije vpišite ime komponente, v pravokotnik pa ime entitete. Nad morebitnimi večbitnimi vodili označite koliko bitov imajo.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY vezje IS
PORT( a, b : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
   sel : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
   y : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END vezje;

ARCHITECTURE struct OF vezje IS
COMPONENT adder IS
PORT( a, b : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
   c : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;

COMPONENT inv IS
PORT( a : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
   b : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;

SIGNAL add_out, inv_out : STD_LOGIC_VECTOR(7 DOWNTO 0);

BEGIN
ADD_1: adder PORT MAP(a=>a, b=>b, c=>add_out);
INV_1: inv  PORT MAP(b, inv_out);

WITH sel SELECT
   y <= add_out WHEN "11",
   inv_out WHEN "10" | "01",
   (OTHERS => '0') WHEN OTHERS;
END struct;
```

Koda VHDL realizira dvonivojsko vezje: Na prvem nivoju se nahaja izbiralnik 4/1, ki izbira med štirimi 8−bitnimi podatkovnimi vhodi. Naslovni vhod izbiralnika je vektor `sel`, na podatkovnih vhodih pa je izhod seštevalnika (`sel="11"`) izhod inverterja (`INV_1`) za kombinaciji (`sel="01"` ali `sel="10"`) ter 8−bitni vektor (`x"00"`) v primeru (`sel="00"`). Na vhod izbiralnika sta priključeni entiteti z imenom `adder`, z imenom `ADD_1` in entiteta `inv` z imenom `INV_1`. Entiteta `adder` ima dva 8−bitna vhoda (`a, b`) in en izhod (`add_out`). Entiteta `inv` ima en 8−bitni vhod (`a`) in en izhod (`inv_out`).



60. Zapišite *kombinacijsko inačico* (procesnega stavka ne smete uporabiti) entitete in arhitekture VHDL opisa primerjalnika dveh 8−bitnih nepredznačenih števil `A` in `B`. Vezje ima še vhod `s`, s katerim določamo način primerjave: Izhod vezja bo večje število, če je `s ='1'` in manjše število, če je `s ='0'`. Ime entitete je `"min_max"`, arhitekture pa `"arch"`.




```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
ENTITY min_max IS
PORT( s : IN STD_LOGIC;
   A,B : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
   C : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END min_max;

ARCHITECTURE arch OF min_max IS

SIGNAL AU, BU : UNSIGNED(7 DOWNTO 0);

BEGIN
AU = UNSIGNED(A);
BU = UNSIGNED(B);
C <= A WHEN ((s = '0' AND AU < BU) OR (s = '1' AND AU > BU)) ELSE B;
END arch;
```

61. Zapišite entiteto in arhitekturo VHDL opisa strukture D-FF, ki je prožen na sprednji rob signala ure (`Clock`). Struktura ima vhod (`D`) in izhod (`Q`) Ime entitete je `"DFF"`, arhitekture pa `"arch"`

Osnovna izvedba D–FF vsebuje funkcijo `rising_edge`, s katero zapišemo pogoj, da (`Clock'EVENT AND Clock = '1'`), kot včasih opazite v različnih zgledih. Obratno polariteto signala ure dosežemo s funkcijo `falling_edge`, s katero zapišemo pogoj, da (`Clock'EVENT AND Clock = '0'`).

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY DFF IS
PORT ( D, Clock : IN STD_LOGIC;
       Q : OUT STD_LOGIC );
END DFF ;

ARCHITECTURE arch OF DFF IS
BEGIN
PROCESS
BEGIN
  IF rising_edge(Clock) THEN
    Q <= D;
  END IF;
END PROCESS;
END arch;
```

Izvedenemu D–FF dodamo še asinhrona signala za brisanje (`nClr`) in postavljanje (`nPreset`). Oba signala imata negativno logiko.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY dff IS
PORT (D, Clock, nClr, nPreset : IN STD_LOGIC;
       Q : OUT STD_LOGIC);
END dff ;
ARCHITECTURE Behavior OF dff IS
BEGIN
PROCESS ( Clock, nClr, nPreset )
BEGIN
     IF nClr = '0' THEN
       Q <= '0';      --brisi
     ELSIF nPreset = '0' THEN
       Q <= '1';      --postavi
     ELSIF Clock'EVENT AND Clock = '1' THEN
       Q <= D;      -- ohranja trenutno stanje
     END IF;
END PROCESS;
END Behavior;
```

Alternativna izvedba čakanja na sprednji rob signala ure:

```
LIBRARY IEEE;
use ieee.std_logic_1164.all;
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY DFF IS
PORT ( D, Clock : IN STD_LOGIC;
       Q : OUT STD_LOGIC );
END flipflop ;
ARCHITECTURE arch OF DFF IS
BEGIN
PROCESS
  BEGIN
    WAIT UNTIL Clock'EVENT AND Clock = '1';
       Q <= D;
  END PROCESS;
END arch;
```

62. Zapišite entiteto in arhitekturo VHDL opisa strukture T-FF, ki je prožen na sprednji rob signala ure (`Clock`). Pomnilnik ima vhod (`T`) in izhod (`Q`) Ime entitete je `"tff"`, arhitekture pa `"arch"`. T−FF ima še asinhrona signala za brisanje (`nClr`) in postavljanje (`nPreset`). Oba signala imata negativno logiko.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY tff IS
   PORT (T, Clock, nClr, nPreset : IN STD_LOGIC;
         Q : OUT STD_LOGIC);
END tff ;
ARCHITECTURE arch OF tff IS
signal output: STD_LOGIC;
BEGIN
PROCESS ( Clock, nClr, nPreset )
BEGIN
   IF nClr = '0' THEN
      Q <= '0';      -- asinhrono brisanje
   ELSIF nPreset = '0' THEN
      Q <= '1';       -- asinhrono postavljanje
   ELSIF Clock'EVENT AND Clock = '1' THEN
      IF (T = '0') THEN
         output <= output;  -- trenutno stanje
      ELSE
         output <= NOT(output);  -- invertiranje
      END IF;
   END IF;
   Q <= output;
END PROCESS;
END arch;
```

Zakaj smo uporabili signal `output`, ko pa bi lahko isto stvar realizirali tudi brez, tako da bi nastavili tip signala `Q` kot `inout`? Signal `output` nam omogoča, da realiziramo inverzijo v vrstici: `output <= NOT(output); -- invertiranje`

V tej vrstici se namreč signal pojavlja na levi in desni strani: Če bi realizacijo izvedli brez vmesnega signala `output` bi bil izhod (`Q`) tipa `inout`. Nasploh se tipu `inout` poskušamo ogniti če se le da, ker trenutno smer signala tega tipa določajo drugi signali. Če ta smer ni pravilno določena, se zgodi da naenkrat povežemo skupaj dva vhoda, kar pomeni, da stanje ni enolično določeno ampak plava (ang. floating).
Druga možnost je hujša: Če povežemo skupaj dva izhoda, lahko nastopi stanje, pri katerem je en izhod enak '0', drugi '1'. Med izhodoma je povezava, preko katere bo stekel tok iz visokega izhoda proti nizkemu (pozitivna logika). Upornost povezave je običajno nizka ($R \approx 10^{-3}\Omega$), zato bo po Ohmovem zakonu stekel *velik tok*, kar povzroči, da se povezava trajno prekine in izhod vezja ni več uporaben.
Obstajajo sicer tipi izhodov, ki jih je možno vezati skupaj: To so tristanjski izhodi, (stanja '0', '1', 'Z') vendar moramo pri njih zelo paziti na trenutno smer (vhod, izhod), da ne uničimo katere od povezav.
Druga možnost so izhodi tipa odprt kolektor (*ang. open collector*) oz. odprt ponor (*ang. open drain*) pri katerih imamo PUN (*ang. pull−up network*) sestavljen iz uporovnega bremena, realiziranega v različnih tehnologijah. Tovrstne izhode je možno vezati skupaj v zvezano ALI operacijo (*ang. wired OR*), vendar so cena za to slabše zakasnitve širjenja (*ang. propagation delay*).[14]

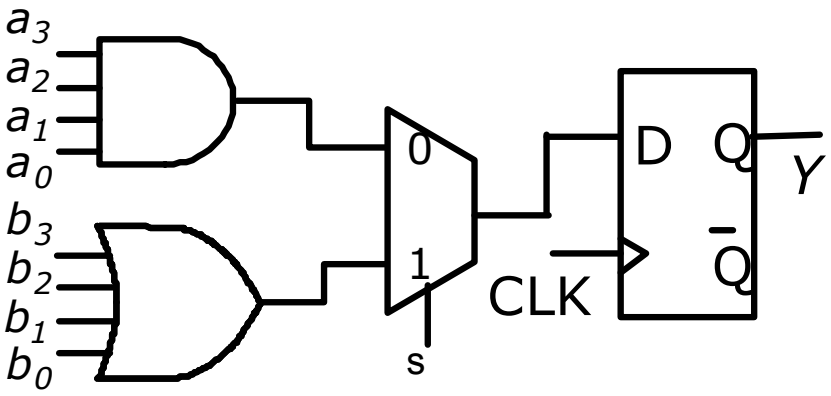
[14] http://en.wikipedia.org/wiki/Open_collector

63. Zapišite entiteto in arhitekturo VHDL opisa strukture JK-FF, ki je prožen na sprednji rob signala ure (`Clock`). Pomnilnik ima vhoda (`J`) in (`K`) in izhod (`Q`) Ime entitete je `"jkff"`, arhitekture pa `"arch"`. T−FF ima še asinhrona signala za brisanje (`nClr`) in postavljanje (`nPreset`). Oba signala imata negativno logiko.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY jkff IS
   PORT (J, K, Clock, nClr, nPreset : IN STD_LOGIC;
         Q : OUT STD_LOGIC);
END jkff ;
ARCHITECTURE arch OF jkff IS
signal output: STD_LOGIC;
BEGIN
PROCESS ( Clock, nClr, nPreset )
BEGIN
   IF nClr = '0' THEN
   -- asinhrono brisanje
     Q <= '0';
   ELSIF nPreset = '0' THEN
   -- asinhrono postavljanje
        Q <= '1';
   ELSIF Clock'EVENT AND Clock = '1' THEN
   -- trenutno stanje
        IF (J = ‘0’ AND K = ‘0’) THEN  output <= output;
   -- postavljanje
        ELSIF (J= ‘1’ AND K= ‘0’) THEN output <= ‘1’;
   -- brisanje
        ELSIF (J= ‘0’ AND K= ‘1’) THEN output <= ‘0’;
   -- invertiranje
        ELSIF (J= ‘1’ AND K= ‘1’) THEN output <= NOT(output);
     END IF;
   END IF;
   Q <= output;
END PROCESS;
END arch;
```

64. Narišite realizacijo podane VHDL kode.

```
LIBRARY IEEE;
use ieee.std_logic_1164.all;
entity vezje is
port ( clk, s : in std_logic;
     a, b : in std_logic_vector(3 downto 0);
     y : out std_logic);
end vezje;

architecture arch of vezje is
begin
process(clk)
begin
if(clk'EVENT and clk='1') then
  if s='0' then
    y<=a(0) and a(1) and a(2) and a(3);
  else
    y<=b(0) or b(1) or b(2) or b(3);
  end if;
end if;
end process;
end arch;
```



65. Zapišite entiteto in arhitekturo VHDL opisa strukture 8 bitnega ROM pomnilnika velikosti 64 bytov. Pomnilnik beremo na prednji rob signala ure (`clk`). Pomnilnik ima 3−stanjske izhode, ki jih omogočimo s signalom (`oe='1'`). Vezje izberemo s signalom (`cs='1'`). Če vezje ni izbrano (`cs='0'`) ali če izhodi vezja niso omogočeni (`oe='0'`), so vsi podatkovni vhodi v stanju visoke impedance (`data="ZZZZZZZZ"`). Spremenljivka (`addr`) hrani naslov pomnilnika, podatkovni vhod/izhod pa poimenujte (`data`). Ime entitete je `"rom"`, arhitekture pa `"Behavioral"`.

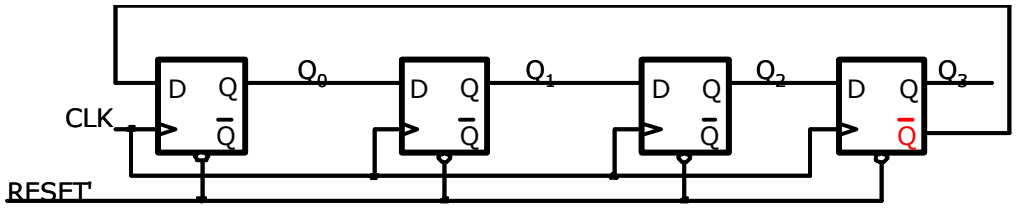
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity rom is
    Port (
    addr     : in std_logic_vector(5 downto 0);
    data     : out std_logic_vector(7 downto 0);
    oe, cs, clk   : in std_logic
   );
end rom;
architecture Behavioral of rom is
type memory is array (63 downto 0) of std_logic_vector (7 downto 0);
signal memoryk : memory;
begin
   memoryk(0) <= "10000100";
   memoryk(1) <= "00101111";
   memoryk(2) <= "10000111";
   memoryk(3) <= "00000000";
   memoryk(4) <= "10000100";
   memoryk(5) <= "11110000";
   memoryk(6) <= "10000111";
   memoryk(7) <= "00000001";
   memoryk(8) <= "00000001";
   memoryk(9) <= "10000111";
   memoryk(10) <= "00000010";
   memoryk(11) <= "10000101";
   memoryk(12) <= "00000011";
   memoryk(13) <= "10000111";
   memoryk(14) <= "00000011";
   memoryk(15) <= "00000010";
   memoryk(16) <= "10000111";
   memoryk(17) <= "00000100";
   memoryk(18) <= "10000110";
   memoryk(19) <= "00000001";
   memoryk(20) <= "10000111";
   memoryk(21) <= "00000101";
   process (clk)
   begin
     if rising_edge(clk) then
       if oe = '1' and cs = '1' then
         data <= memoryk(conv_integer(addr));
       else
         data <= (others => 'Z');
       end if;
     end if;
   end process;
end Behavioral;
```

66. Zapišite entiteto in arhitekturo VHDL opisa strukture 8 bitnega statičnega RAM pomnilnika velikosti 64 bytov. Pomnilnik beremo na prednji rob in vpisujemo na zadnji rob signala ure (`clk`). Pomnilnik ima 3−stanjske izhode, ki jih omogočimo s signalom (`oe='1'`). Vezje izberemo s signalom (`cs='1'`). Če vezje ni izbrano (`oe='0'`) ali če izhodi vezja niso omogočeni (`cs='0'`), so vsi podatkovni vhodi v stanju visoke impedance (`data="ZZZZZZZZ"`). Spremenljivka (`addr`) hrani naslov pomnilnika, podatkovni vhod/izhod pa poimenujte (`data`). Ime entitete je `"ram"`, arhitekture pa `"Behavioral"`.

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity ram is
    Port (
   addr:  in std_logic_vector(5 downto 0);
   data   : inout std_logic_vector(7 downto 0);
   w_nr   : in std_logic;
   clk, oe, cs:  in std_logic
   );
end ram;

architecture Behavioral of ram is
   type mem_array is array(63 downto 0) of std_logic_vector (7 downto
0);
   signal memory : mem_array;
   signal read, write : std_logic;
begin

   read <= oe and cs and (not w_nr);
   -- beremo lahko, ko je vezje izbrano, vhodi omogoceni in ko je
   -- nastavljena operacija branja
   write <= w_nr and cs;
   -- pisemo lahko, ko je vezje izbrano, in ko je
   -- nastavljena operacija pisanja
ram: process(clk) is
   begin
     if rising_edge(clk) then
       --preberi podatek na prednji rob signala ure
       if read = '1' then  --beri
         data <= memory(conv_integer(addr));
       elsif write = '1' then --pisi na zadnji rob signala ure
         memory(conv_integer(addr)) <= data;
       else
         data <= (others => 'Z');  --sicer so izhodi Hi-Z
       end if;
     end if;
   end process;
end Behavioral;
```

67. Zapišite entiteto in arhitekturo VHDL opisa 4−bitnega Johnsonovega števca, ki je prožen na sprednji rob signala ure (`CLK`). Števec ima 4−bitni števni izhod (`Q`) in asinhron signal za ponastavljanje (`nRESET`), ki ima negativno logiko. Ime entitete je `"johnson"`, arhitekture pa `"arch"`.

Strukturo Johnson−ovega števca povzema spodnja slika:




```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity johnson is
port (
        Q : out std_logic_vector(3 downto 0);
        nRESET, CLK : in std_logic
        );
end johnson;
architecture arch of johnson is
signal temp : unsigned(3 downto 0);
begin
process(CLK)
begin
    if( rising_edge(CLK) ) then
        if (nRESET = '0') then
            temp <= (others => '0'); --brisanje vsebine
        else
            temp(1) <= temp(0);  --pomikanje
            temp(2) <= temp(1);
            temp(3) <= temp(2);
            temp(0) <= not temp(3);
        end if;
    end if;
end process;
Q <= temp; --kombinacijska prireditev
end arch;
```

Z ukaznim zaporedjem izvedemo dejansko pomikanje v registru spremenljivke (temp):

```
            temp(1) <= temp(0);  --pomikanje
            temp(2) <= temp(1);
            temp(3) <= temp(2);
            temp(0) <= not temp(3);
```

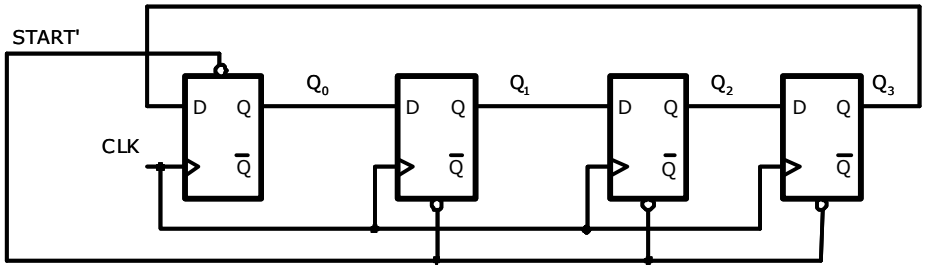
Vrstni red ukazov, ki skrbijo za pomikanje je bistven, ker se zaporedje nahaja znotraj procesnega stavka.

Enak pomen kot zgornje zaporedje stavkov lahko zapišemo s pomočjo operatorja lepljenja (&), ki sestavi več bitov skupaj v vektor:

```
            temp <= temp(2 downto 0) & not temp(3);
```

68. Zapišite entiteto in arhitekturo VHDL opisa 4−bitnega krožnega števca (ang. ring counter), ki je šteje na sprednji rob signala ure (`CLK`). Števec ima 4−bitni številni izhod (`Q`) in asinhron signal za postavljanje začetnega stanja (`nSTART`), ki ima negativno logiko. Ob signalu (`nSTART='0'`) se v verigo FF naložijo same '0', razen na prvo mesto, kamor se naloži '1'. Ime entitete je `"ring"`, arhitekture pa `"arch"`.

Strukturo krožnega števca povzema spodnja slika:




```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity ring is
port (
        Q : out std_logic_vector(3 downto 0);
        nSTART,CLK : in std_logic
        );
end ring ;
architecture arch of ring is
signal temp : unsigned(3 downto 0);
begin
process(CLK)
begin
    if( rising_edge(CLK) ) then
        if (nSTART = '1') then
            --postavi LSB na '1', ostalo na '0'
            temp <= (others => '0') & '1';
        else
            temp(1) <= temp(0);  --pomikanje
            temp(2) <= temp(1);
            temp(3) <= temp(2);
            --razlika je samo negacija pri Johnson števcu
            temp(0) <= temp(3);
        end if;
    end if;
end process;
Q <= temp; --kombinacijska prireditev izhoda
end arch;
```

Z ukaznim zaporedjem izvedemo dejansko pomikanje v registru spremenljivke (temp):

```
            temp(1) <= temp(0);  --pomikanje
            temp(2) <= temp(1);
            temp(3) <= temp(2);
            temp(0) <= temp(3);
```

Vrstni red ukazov, ki skrbijo za pomikanje je bistven, ker se zaporedje nahaja znotraj procesnega stavka.. Enak pomen kot zgornje zaporedje stavkov lahko zapišemo s pomočjo operatorja lepljenja (&), ki sestavi več bitov skupaj v vektor:

```
            temp <= temp(2 downto 0) & temp(3);
```

69. Zapišite entiteto in arhitekturo VHDL opisa D zapaha, ki je transparenten (Q=D) na visoko stanje signala ure. D–zapah ima vhod `D` in izhod `Q`. Ime entitete je `"latch"`, arhitekture pa `"arch"`.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY latch IS
PORT (D, Clk : IN STD_LOGIC;
     Q : OUT STD_LOGIC);
END latch;
ARCHITECTURE arch OF latch IS
BEGIN
PROCESS ( D, Clk )
BEGIN
   IF Clk = '1' THEN
      Q <= D;
   END IF;
END PROCESS;
END arch;
```

Tak D–zapah uporablja implicitni spomin v procesu, saj izid `IF` stavka ni določen v obeh primerih. Implicitni spomin v VHDL ni zaželen, ker velikokrat nakazuje na nedokončan zapis pogoja `IF` ali `CASE` stavka. Orodje za sintezo pregleduje vse izide `IF` stavka in čim kateri izid manjka uporabnika opozori:

"`WARNING:Xst:737 - Found 1-bit latch for signal <q>. Latches may be generated from incomplete case or if statements. We do not recommend the use of latches in FPGA/CPLD designs, as they may lead to timing problems.`"

70. Zapišite entiteto in arhitekturo VHDL opisa T−FF, ki je prožen na sprednji rob signala ure (`Clock`). Flip−flop ima vhod `T` in izhod `Q`. T−FF ima tudi signala za asinhrono brisanje (`nCLR`) in asinhrono postavljanje (`nPRESET`). Oba signala imata negativno logiko (aktiven `nCLR='0'`) in (aktiven `nPRESET='0'`). Ime entitete je `"tff"`, arhitekture pa `"Behavior"`.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY tff IS
   PORT (T, Clock, nClr, nPreset : IN STD_LOGIC;
         Q : OUT STD_LOGIC);
END tff ;
ARCHITECTURE Behavior OF tff IS
signal output: STD_LOGIC;
BEGIN
PROCESS ( Clock, nClr, nPreset )
BEGIN
   IF nClr = '0' THEN
      Q <= '0';       -- asinhrono brisanje
   ELSIF nPreset = '0' THEN
      Q <= '1';       -- asinhrono postavljanje
   ELSIF Clock'EVENT AND Clock = '1' THEN
      IF (T = '0') THEN
         output <= output;  -- trenutno stanje
      ELSE
         output <= NOT(output);  -- invertiranje
      END IF;
   END IF;
   Q <= output;
END PROCESS;
END Behavior;
```

71. Zapišite entiteto in arhitekturo VHDL opisa JK−FF ki je prožen na sprednji rob signala ure (`Clock`). Flip−flop ima vhoda `J` in `K` in izhod `Q`. JK−FF ima tudi signala za asinhrono brisanje (`nCLR`) in asinhrono postavljanje (`nPRESET`). Oba signala imata negativno logiko (aktiven `nCLR='0'`) in (aktiven `nPRESET='0'`). Ime entitete je `"jkff"`, arhitekture pa `"Behavior"`.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY jkff IS
   PORT (J, K, Clock, nClr, nPreset : IN STD_LOGIC;
         Q : OUT STD_LOGIC);
END jkff ;
ARCHITECTURE Behavior OF jkff IS
signal output: STD_LOGIC;
BEGIN
PROCESS ( Clock, nClr, nPreset )
BEGIN
   IF nClr = '0' THEN
      Q <= '0';
      -- asinhrono brisanje
   ELSIF nPreset = '0' THEN
      Q <= '1';
      -- asinhrono postavljanje
   ELSIF Clock'EVENT AND Clock = '1' THEN
      IF (J = ‘0’ AND K = ‘0’) THEN  output <= output;
         -- trenutno stanje
         ELSIF (J= ‘1’ AND K= ‘0’) THEN output <= ‘1’;
         -- postavljanje
         ELSIF (J= ‘0’ AND K= ‘1’) THEN output <= ‘0’;
         -- brisanje
         ELSIF (J= ‘1’ AND K= ‘1’) THEN output <= NOT(output);
         -- invertiranje
      END IF;
   END IF;
   Q <= output;
END PROCESS;
END Behavior;
```

72. Zapišite entiteto in arhitekturo VHDL opisa izmišljenega "M−N" flip−flopa, ki je prožen na zadnji rob signala ure. Flip−flop ima vhoda `M` in `N`, komplementarna izhoda `Q` in `nQ` in signal za asinhrono brisanje (`nCLR`), ki ima negativno logiko (aktiven `nCLR='0'`). Ime entitete je `"MNFF"`, arhitekture pa `"arch"`.
Delovanje flip−flopa povzema spodnja tabela:

M	N	Q(t+1)	nQ(t+1)	Pomen stanja
0	0	nQ(t)	Q(t)	Invert
0	1	1	0	Set
1	0	0	1	Reset
1	1	Q(t)	nQ(t)	Hold

```
library ieee;
use ieee.std_logic_1164;
entity MNFF is
port( clk, nCLR, M, N: in std_logic;
     Q, nQ: out std_logic);
end MNFF;
architecture arch of MNFF is
signal TEMP :std_logic;
begin
process(clk, nCLR)
begin
  if(nCLR = '0') then
    TEMP <= '0';
  elsif(clk'EVENT and clk='0') then
    if ( M = '0' and N = '0' ) then  TEMP <= not TEMP;
    elsif ( M = '0' and N = '1' ) then  TEMP <= '1';
    elsif ( M = '1' and N = '0' ) then  TEMP <= '0';
    end if;
  end if;
end process;
Q <= TEMP;
nQ <= not TEMP;
end arch;
```

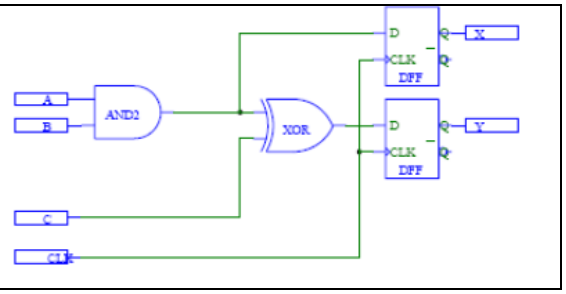
Sicer gre za JK−FF, le vhoda JK sta negirana. Zakaj smo uporabili signal `TEMP`, ko pa bi lahko isto stvar realizirali tudi brez, tako da bi nastavili tip signala `Q` kot `inout`?
Signal `TEMP` nam omogoča, da realiziramo inverzijo v vrstici:

```
TEMP <= not TEMP;
```

V tej vrstici se namreč signal pojavlja na levi in desni strani: Če bi realizacijo izvedli brez vmesnega signala `TEMP` bi bil izhod (`Q`) tipa `inout`. Nasploh se tipu `inout` poskušamo ogniti če se le da, ker trenutno smer signala tega tipa določajo drugi signali. Če ta smer ni pravilno določena, se zgodi da naenkrat povežemo skupaj dva vhoda, kar pomeni, da stanje ni enolično določeno ampak plava (ang. floating).
Druga možnost je hujša: Če povežemo skupaj dva izhoda, lahko nastopi stanje, pri katerem je en izhod enak '0', drugi '1'. Med izhodoma je povezava, preko katere bo stekel tok iz visokega izhoda proti nizkemu (pozitivna logika). Upornost povezave je običajno nizka ($R \approx 10^{-3}\Omega$), zato bo po Ohmovem zakonu stekel *velik tok*, kar povzroči, da se povezava trajno prekine in izhod vezja ni več uporaben.

73. Podana je entiteta `e1` nekega vezja v VHDL in diagram realizacije, ki bo predstavljal arhitekturo tega vezja (`arch`). Zapišite VHDL kodo arhitekture tega vezja.

```
entity e1 is
port( clk, a, b, c: in std_logic;
   x, y: out std_logic);
end e1;
```



Entiteta `e1` je sestavljena iz dveh D–FF, ki ju realizira procesni stavek. Iz sheme se vidi, da je D–FF prožen na sprednji rob signala ure `clk`. Definiramo še vmesna signala `d, e` s katerima realiziramo povezavo AND vrat na vhod prvega D–FF in XOR vrat na vhod drugega D–FF.

```
library ieee;
use ieee.std_logic_1164;
entity e1 is
port( clk, a, b, c: in std_logic;
   x, y: out std_logic);
end e1;

architecture arch of e1 is

signal d, e :std_logic;

begin
process(clk)
begin
   if(clk'EVENT and clk='1') then
      x <= d;
      y <= e;
   end if;
end process;

d <= a and b;
e <= d xor c;

end arch;
```

74. Podani sta dve arhitekturi vezja a_1 in a_2 in entiteta e_1 v VHDL. V kaj se sintetizira spodnja koda VHDL?

```
ENTITY e1 IS
   PORT(clk, d : in std_logic;
     q : out std_logic);
END e1;
ARCHITECTURE a1 OF e1 IS
BEGIN
PROCESS(clk, d)
   BEGIN
   IF(clk='0') THEN
      q <= d;
   ELSE
      q <= not d;
   END IF;
   END PROCESS;
END a1;
```

```
ENTITY e1 IS
   PORT(clk, d : in std_logic;
     q : out std_logic);
END e1;
ARCHITECTURE a2 OF e1 IS
BEGIN
PROCESS(clk, d)
   BEGIN
      IF(clk='1') THEN
         q <= d;
      END IF;
   END PROCESS;
END a2;
```

Arhitektura a_1 se sintetizira v XOR vrata, saj ima if stavek izraženi obe možnosti (TRUE, FALSE). Naredimo razvoj po spremenljivki `clk` in zapišemo pravilnostno tabelo:

clk	*d*	*q*
0	0	0
0	1	1
1	0	1
1	1	0

Iz pravilnostne tabele lahko zapišemo:

$$q = \begin{cases} d & clk = '0' \\ \bar{d} & clk = '1' \end{cases}$$

$$q = clk \oplus d$$

Arhitektura a_2 se sintetizira v D zatič (ang. D−latch), ki je občutljiv na nivo *clk='1'*. Arhitektura a_2 prikazuje implicitni spomin v procesu, saj izid `IF` stavka ni določen v obeh primerih. Implicitni spomin v VHDL ni zaželen, ker velikokrat nakazuje na nedokončan zapis pogoja `IF` ali `CASE` stavka. Orodje za sintezo pregleduje vse izide `IF` stavka in čim kateri izid manjka uporabnika opozori:

"`WARNING:Xst:737 - Found 1-bit latch for signal <q>. Latches may be generated from incomplete case or if statements. We do not recommend the use of latches in FPGA/CPLD designs, as they may lead to timing problems.`"

Sinteza arhitekture a_1

Sinteza arhitekture a_2