Osnove modeliranja digitalnega vezja v strojno-opisnem jeziku

Uvod

Strojno-opisni jezik (angl. Hardware Description Language - HDL) je računalniški jezik za načrtovanje digitalnih vezij. Predstavili bomo načrtovanje v standardnem jeziku VHDL in poenostavljenem SHDL. Načrt digitalnega vezja tradicionalno predstavlja logična shema, ki prikazuje gradnike vezja in povezave med njimi. Strojno-opisni jezik opisuje zgradbo vezja (strukturni opis) ali pa predstavlja delovanje vezja na bolj abstrakten način.

Funkcijski opis

Kombinacijsko logiko predstavimo z izrazi, ki opisujejo funkcije (npr. seštevanje) in pretok podatkov (dataflow) med vhodom in izhodom vezja.

Opis na ravni registrov

Sekvenčna vezja predstavimo s flip-flopi in registri ter kombinacijskimi prevajalnimi funkcijami (Register Transfer Level - RTL). Iz opisa vezja na ravni registrov dobimo shemo vezja.

Postopkovni opis

Postopkovni (behavioral) opis je najbolj abstrakten, ker predstavlja delovanje vezja v obliki algoritma. Ta opis je zelo podoben programski kodi, vendar ga moramo drugače obravnavati, ker predstavlja vezje z gradniki, ki hkrati izvajajo naloge.

1. Osnovni elementi

Osnovni elementi opisa vezja so signali, ki predstavljajo zunanje povezave (priključke) in notranje povezave. Strojno-opisni jezik najprej predstavi vezje kot enoto (entiteto) z določenim imenom in deklaracijami priključkov. Signale opišemo podobno kot spremenljivke v programskih jezikih: določimo imena in vrsto oz. podatkovni tip, pri zunanjih priključkih pa tudi smer (**in** so vhodi, **out** pa so izhodi).

V jeziku VHDL je opis sestavljen iz entitete (**entity**) s priključki (**port**) in opisom pripadajoče zgradbe (**architecture**). Na vrhu opisa vključimo knjižnice, ki določajo podatkovne tipe in operacije. Sintaksa jezika VHDL je zelo gostobesedna, saj ima veliko rezerviranih besed.

Za lažji začetek bomo uporabljali jezik SHDL, ki pozna enostaven opis entitete in deklaracij, zgradbo vezja pa zapišemo med rezerviranima besedama **begin** in **end**. Mogoče je celo izpustiti stavek **entity**, kar pomeni da bo imel model vezja neko privzeto ime. Načrtovanje vezij v obeh jezikih bomo predstavili na primerih v naslednjih poglavjih, ki obsegajo vse ravni opisovanja digitalnega vezja.

2. Funkcijski opis kombinacijskega vezja

Seštevalnik

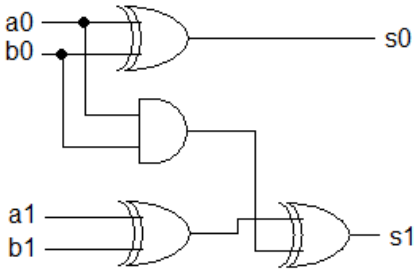


Tabela prikazuje opis dvobitnega seštevalnika, ki je sestavljen iz štirih logičnih vrat v poenostavljenem jeziku SHDL (levo) in standardnem VHDL (desno).

Enobitni signali so deklarirani kot **u1** oz. **std_logic**. Logična vrata opišemo z operatorji **and, or, not, xor**.

```
entity add2
 a0,a1,b0,b1: in u1
 s0,s1: out u1
 p0: u1
begin
 s0=a0 xor b0
 p0=a0 and b0
 s1=p0 xor (a1 xor b1)
end
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity add2 is
 port (
   a0,a1,b0,b1: in std_logic;
   s0,s1: out std_logic
 );
end add2;

architecture RTL of add2 is
 signal p0: std_logic;
begin
 s0 <= a0 xor b0;
 p0 <= a0 and b0;
 s1 <= p0 xor (a1 xor b1);
end RTL;
```

Vrstni red prireditvenih stavkov ni pomemben, saj opisujejo dele vezja, ki sočasno določajo izhodne vrednosti. Vsak stavek predstavlja logiko, katere izhod je signal na levi strani.

Omejitev: posameznemu signalu smemo le **enkrat** prirediti vrednost, sicer bi naredili kratek stik !

Deklaracija vektorskih signalov, npr. 2-bitni vektor:

```
a,b: in u2
```

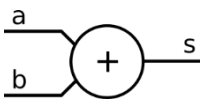
```
a,b : in std_logic_vector(1 downto 0);
```

Posamezen bit (element vektorja) dobimo z uporabo indeksa v okroglih oklepajih.

```
s0 <= a(0) xor b(0);
p0 <= a(0) and b(0);
s1 <= p0 xor (a(1) xor b(1));
```

Omejitev SHDL: elementi vektorja lahko nastopajo le na desni strani izrazov, ker ne preverja prekrivanja indeksov in bi lahko opisali kratek stik !

Seštevalnik z aritmetičnimi operatorji




```
entity add2
 a,b: in u2;
 s: out u2;
begin
 s = a + b
end
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity add2 is
 port (
   a,b: in unsigned(1 downto 0);
   s: out unsigned(1 downto 0) );
end add2;

architecture RTL of add2 is
begin
 s <= a + b;
end RTL;
```

Večbitne signale lahko seštevamo z operatorjem seštevanja. Jezik VHDL ne zna seštevati s standardnimi vektorji, zato moramo vključiti novo knjižnico (**numeric_std**) in deklarirati vektorje kot nepredznačena (**unsigned**) ali predznačena (**signed**) števila.

Omejitev VHDL: pri izrazih z vektorji se mora podatkovni tip signala in število bitov ujemati s tipom signala, ki mu prirejamo izraz.

Pri seštevanju pride do prenosa, zato bi vsoto 2-bitnih vrednosti morali zapisati v 3-bitni vektor. Jezik SHDL ta primer samodejno upošteva, v jeziku VHDL pa moramo vektorja pred seštevanjem razširiti:

```
s <= resize(a,3) + resize(b,3);
```

Funkcija **resize** spremeni velikost celoštevilskega vektorja vrste **unsigned** ali **signed**, tako da poskuša ohraniti vrednost (doda ali odstrani bite na levi strani). V jeziku VHDL bi dejansko bilo dovolj, če bi razširili enega izmed sumandov, ker je operator definiran tudi za različno dolge vektorja. V tem primeru je število bitov vsote enako številu bitov daljšega vektorja.

Kaj pa če želimo prišteti enobitno vrednost, npr. vhodni prenos? VHDL zahteva v tem primeru pretvorbo enobitne v nepredznačeno vektorsko vrednost:

```
c: in u1
a,b: in u2
s: out u3

s = a + b + c
```

```
c: in std_logic;
a,b: in unsigned(1 downto 0);
s: out unsigned(2 downto 0)

s <= (resize(a,3) + resize(b,3)) + unsigned'("" & c);
```

Ostale aritmetične operacije

Podobno kot seštevanje, deluje tudi odštevanje vektorjev. Jezik **VHDL** pozna še množenje in deljenje celoštevilskih vektorjev, ki predstavljata kombinacijski množilnik in delilnik. Programirljiva vezja vsebujejo kombinacijske množilnike, deljenje pa ni vedno dobro podprto z orodji za sintezo vezij. Jezik **SHDL** zna modelirati le množenje vektorjev. Dolžina produkta celoštevilskih vektorjev je enaka vsoti dolžin množenca in množitelja, npr:

`a: u4`
`b: u6`
`p: u10`
 	
`p = a * b`
	`signal a : unsigned(3 downto 0);`
	`signal b : unsigned(5 downto 0);`
	`signal p : unsigned(9 downto 0);`
	
	`p <= a * b;`

Konstantne vrednosti

Kako zapišemo konstantno vrednost v SHDL in VHDL? Jezik SHDL omogoča enostavno pisanje izrazov s celoštevilskimi vrednostmi v desetiškem zapisu, na voljo pa je tudi zapis binarnega vektorja in alternativni dvojiški (0b) in šestnajstiški (0x) zapis celega števila.

SHDL	**VHDL**
- celoštevilske konstante: 0, 1, 2 - binarni vektorji: "0000", "0001", "0010" - alternativni zapis: 0b0000, 0x0, 0xF	- enobitne konstante: '0', '1' - vektorske konstante: "0000", "0001", "1111" - šestnajstiški zapis vektorja: X"0", X"1", X"F" - celoštevilske konstante: 0, 1, 2 - pri pretvorbi celega št. v nepredznačen vektor določimo vrednost in št. bitov: **to_unsigned**(0, 4);

Jezik VHDL zahteva pri prirejanju konstante **ujemanje podatkovnih tipov**, v jeziku SHDL pa za ustrezno pretvorbo poskrbi prevajalnik.

`b: u1`
`c,d: u4`
 	
`b=1`
`c=2`
`d="0011"`
	`signal b: std_logic;`
	`signal c,d: unsigned(3 downto 0);`
	
	`b <= '1';`
	`c <= to_unsigned(2, 4);`
	`d <= "0011";`

Konstante, ki jim vrednosti določimo ob deklaraciji zapišemo v jeziku VHDL z rezervirano besedo **constant** in posebno prireditvijo. V jeziku SHDL pa le določimo začetno vrednost deklariranega signala.

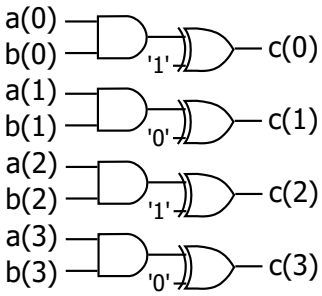
Seštevalnik s konstanto

Aritmetične operacije podpirajo kombiniranje vektorskih in celoštevilskih vrednosti. Npr. prištevanje konstantne vrednosti v obeh jezikih zapišemo na zelo kompakten način:

```
d <= c + 1;
```

Logične operacije z vektorji

Logične operacije z vektorskimi signali delujejo tako, da se izvedejo nad posameznimi biti.



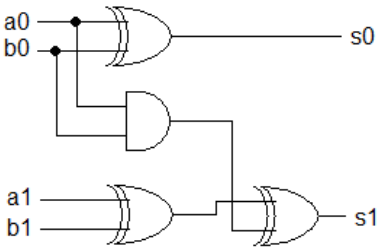
Primer:

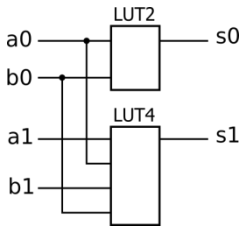
```
a,b,c: u4
c = (a and b) xor "0101";
```

V izrazu s 4-bitnimi vektorji je vsak logični operator ponovljen štirikrat. V jeziku **VHDL** morajo imeti vektorski operandi v logičnih izrazih enako dolžino.

Kombinacijsko vezje z LUT ali ROM

Naredimo dvobitni seštevalnik z vpoglednimi tabelami (Look-Up Table – LUT). Delovanje seštevalnika predstavimo z dvema pravilnostnima tabelama: izhod s1 ima 16 vrstic, ker je odvisen od vseh štirih vhodov, s2 pa le 4 vrstice, ker je odvisen le od a0 in b0. V vezju bomo imeli eno dvovhodno (LUT2) in eno štirivhodno (LUT4) tabelo, ki deluje kot bralni pomnilnik (ROM).





a1	a0	b1	b0	s1
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

a0	b0	s0
0	0	0
0	1	1
1	0	1
1	1	0

V strojno-opisnem jeziku je pomnilnik zbirka vrednosti. Model pomnilnika ROM ali vpogledne tabele je sestavljen iz deklaracije zbirke, inicializacije vrednosti in stavka, ki opisuje branje iz pomnilnika. V jeziku **VHDL** določimo nov podatkovni tip in deklariramo signal, ki mu nastavimo vrednost, v SHDL pa to storimo obenem. Poglejmo primer deklaracije zbirke štirih enobitnih vrednosti:

`lut2: 4u1 = 0,1,1,0;`	`type lut2_type is array (0 to 3) of std_logic;` `signal lut2: lut2_type := ('0','1','1','0');`

Branje pomnilnika opisuje prireditveni stavek v katerem z naslavljanjem zbirke izberemo izhodno vrednost: `izhod = lut(naslov);`

Naslov sestavimo iz vhodnih signalov, pri čemer moramo paziti, da je vrstni red signalov takšen, kot smo ga imeli pri sestavljanju pravilnostne tabele. Sestavljanje signalov iz posameznih bitov opisuje operator &. Primer opisa pomnilnika, ki je narejen kot dvovhodna tabela:

```
adr1: u2

adr1 = a(0) & b(0);
s0 = lut2(adr1);
```

```
signal adr1: unsigned(1 downto 0);

adr1 <= a(0) & b(0);
s0 <= lut2(to_integer(adr1));
```

Naslov adr1 smo deklarirali kot 2-bitni vmesni signal. V jeziku VHDL je naslov (indeks) zbirke določen kot celoštevilski podatek (**integer**), zato potrebujemo funkcijo za pretvorbo iz nepredznačenega vektorja v celo število.

Celoten opis dvobitnega seštevalnika z vpoglednimi tabelami v SHDL in VHDL:

```
entity add2lut
 a,b: in u2
 s1,s0: out u1
 lut2: 4u1 = 0,1,1,0;
 lut4: 16u1 = 0,0,1,1,0,1,1,0,
              1,1,0,0,1,0,0,1;
 adr1: u2
 adr2: u4
begin
 adr1 = a(0) & b(0)
 s0 = lut2(adr1)
 adr2 = a(1) & a(0) & b(1) & b(0)
 s1 = lut4(adr2)
end
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity add2lut is
 port (
   a, b : in unsigned(1 downto 0);
   s1, s0 : out std_logic );
end add2lut;

architecture RTL of add2lut is
 type lut2_type is array (0 to 3) of std_logic;
 signal lut2 : lut2_type := ('0', '1', '1', '0');
 type lut4_type is array (0 to 15) of std_logic;
 signal lut4 : lut4_type := ('0','0','1','1',
 '0','1','1','0','1','1','0','0','1','0','0','1');
 signal adr1 : unsigned(1 downto 0);
 signal adr2 : unsigned(3 downto 0);
begin
 adr1 <= a(0) & b(0);
 s0 <= lut2(to_integer(adr1));
 adr2 <= a(1) & a(0) & b(1) & b(0);
 s1 <= lut4(to_integer(adr2));
end RTL;
```

Vpogledno tabelo z enobitnim izhodom lahko opišemo še bolj preprosto z vektorjem namesto z zbirko. Vektorju priredimo začetno vrednost in ga pri branju naslovimo, podobno kot zbirko. Pri opisu pravilnostne tabele z vektorjem moramo paziti na vrsti red elementov: elemente zbirke običajno določimo od najmanjšega do največjega indeksa (npr. 0 to 3), pri vektorskih signalih pa tečejo indeksi v obratnem vrstnem redu (3 downto 0) !

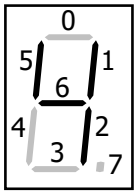
```
lut1: u4 = "0110";
lut2: u16 = "1001001101101100";

s0 = lut1(adr1)
s1 = lut2(adr2)
```

```
constant lut1: unsigned(3 downto 0) :=
"0110";
constant lut2: unsigned(15 downto 0) :=
"1001001101101100";

s0 <= lut1(to_integer(adr1));
s1 <= lut2(to_integer(adr2));
```

Dekodirnik

Pomnilnik ROM je zelo uporaben za opis dekodirnikov, ki jih ne moremo enostavno zapisati s funkcijskimi izrazi. Vzemimo primer dekodirnika za 7-segmentni prikazovalnik, ki pretvarja 4-bitno vhodno vrednost (bcd) v 7-bitno kodo za prižiganje ustreznih segmentov svetlečih diod na prikazovalniku. Dekodirnik opisuje pravilnostna tabela:

število	BCD vrednost	7-bitna koda
0	0000	0111111
1	0001	0000110
2	0010	1011011
3	0011	1001111
4	0100	1100110
5	0101	1101101
6	0110	1111101
7	0111	0000111
8	1000	1111111
9	1001	1101111



Kombinacijsko logiko dekodirnika opišemo v obliki pomnilnika ROM, ki vsebuje deset 7-bitnih konstantnih vrednosti:

```
bcd: in u4;
led: out u7;

rom: 10u7 =
"0111111","0000110",
"1011011","1001111",
"1100110","1101101",
"1111101","0000111",
"1111111","1101111";

led = rom(bcd)
```

```
 bcd : in unsigned(3 downto 0);
 led : out unsigned(6 downto 0);

type rom_type is array (0 to 9) of unsigned(6 downto 0);
signal rom: rom_type := ("0111111", "0000110",
"1011011", "1001111",
"1100110", "1101101",
"1111101", "0000111",
"1111111", "1101111");

led <= rom(to_integer(bcd));
```

Opis dekodirnika na zajame vseh možnih kombinacij. Kadar je vhod bcd večji ali enak 10, bo naslov pomnilnika izven območja vrednosti. Simulator vezja (VHDL in SHDL) bo v tem primeru javil napako:

Error : Array: rom(10) index out of bounds!

Model vezja je mogoče kljub temu sintetizirati. Če poskrbimo, da na vhod ne pridejo vrednosti izven območja, bo vezje delovalo povsem pravilno. Ob nepravilnem vhodu pa ne moremo predvideti vrednosti izhoda. Odziv vezja bi lahko nedvoumno določili z dodatnimi pogoji v opisu, ki pa predstavljajo tudi dodatno logiko in večjo površino vezja.

Primerjalnik

Primerjanje vrednosti dveh signalov ali signala in konstante zapišemo v obliki enostavnega pogoja s primerjalnim operatorjem. S pogojem preverjamo ali sta vrednosti na levi in desni strani enaki (npr: a = b ali c = 1), različni (npr. a /= b) in ali vrednosti vektorskih ustrezata zapisani relaciji (npr: a > b ali b <= a).

Primerjalni operatorji

=	/=	>	>=	<	<=
enako	ni enako	večje	večje ali enako	manjše	manjše ali enako

Omejitve relacijskih operatorjev: operatorja enako in ni enako sta definirana za vse tipe podatkov, ostali operatorji pa le za vektorske podatke. Pri primerjavi dveh signalov morata biti na obeh straneh signala enakega (Npr. signed ali unsigned in enaka dolžina vektorja).

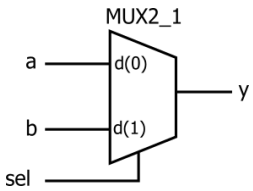
Kombinacijski primerjalnik opišemo s pogojnim prireditvenim stavkom v katerem priredimo 1 ob izpolnjenem pogoju in 0, kadar pogoj ni izpolnjen. Primer primerjalnika dveh štiri-bitnih vrednosti:

```
x,y: in s4;
enako, vecje: out u1;
negx, negy: out u1;

enako = 1 when x=y else 0
vecje = 1 when x>y else 0
negx = 1 when x<0 else 0
negy = 1 when y<0 else 0
```

```
x, y : in signed(3 downto 0);
enako, vecje : out std_logic;
negx, negy : out std_logic;

enako <= '1' when x = y else '0';
vecje <= '1' when x > y else '0';
negx <= '1' when x < 0 else '0';
negy <= '1' when y < 0 else '0';
```

Izbiralnik

Pogojni prireditveni stavek (when…else) je uporaben tudi za funkcijski opis kombinacijskega izbiralnika z dvema vhodoma:




```
y = a when sel=1 else b
```

```
y <= a when sel = '1' else b;
```

Krmilni vhod sel je eno-biten signal, ki določa kateri izmed vhodov bo povezan na izhod. Vhodi in izhod so lahko eno-bitni ali pa vektorski signali.

Kadar imamo eno-bitne vhode, lahko opišemo izbiralnik tudi z operatorjem indeksiranja vhodnega vektorja. Primer opisa izbiralnika, ki z dvo-bitnim krmilnim vhodom sel izbira med enim izmed štirih vhodov:

`d: in u4;`
`sel: in u2;`
`y: out u1;`
 	
`y = d(sel)`
	`d : in unsigned(3 downto 0);`
	`sel : in unsigned(1 downto 0);`
	`y : out std_logic;`
	
	`y <= d(to_integer(sel));`

Dvo-vhodni izbiralnik v jeziku VHDL vseeno raje opišemo s stavkom **when … else**, ker ima takšen izbiralnik eno-biten krmilni signal, ki ga pretvorimo v celoštevilski indeks s precej zapletenim izrazom:

```
y <= d(to_integer(unsigned'("" & sel)));
```

Jezik **VHDL** pozna tudi razširjeno obliko pogojnega prireditvenega stavka v katerem preverjamo več pogojev in je primeren za opis izbiralnikov z več kot dvema vhodoma:

```
y <= d(3) when sel = "11" else
     d(2) when sel = "10" else
     d(1) when sel = "01" else
     d(0);
```

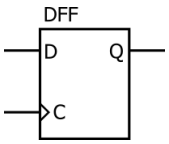
Poenostavljen jezik **SHDL** ne pozna razširjene oblike prireditvenega stavka. Kadar potrebujemo več pogojev, uporabimo pogojni stavek **if … elsif.**

```
if sel=3 then y=d(3)
elsif sel=2 then y=d(2)
elsif sel=1 then y=d(1)
else y=d(0)
end
```

Pogojni stavek spada med konstrukte za postopkovni opis delovanja vezja, ki ga bomo obravnavali v posebnem poglavju.

3. Opis na ravni registrov

Podatkovni flip-flop



Podatkovni ali D flip-flop (DFF) je osnovni pomnilni element sinhronih vezij. Izhod flip-flopa se spreminja le ob naraščajoči ali padajoči fronti ure, vmes pa se njegovo stanje ohranja. Naraščajočo fronto ure definiramo s pogojem: `rising_edge(clk)`. Model flip-flopa v jeziku **VHDL** opišemo s procesom:

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

Stavki znotraj procesa se med simulacijo ovrednotijo le ob spremembi signala clk, pogojni stavek pa dodatno predpisuje spremembo ure iz 0 na 1 oz. naraščajočo fronto ure. Ob naraščajoči fronti ure se vrednost iz podatkovnega vhoda d prenese na izhod q.

V jeziku VHDL je mogoče katerikoli eno-bitni signal opisati kot uro z uporabo ustrezno zapisanega pogoja. Praktična vezja pa postavljajo številne zahteve: ura pomnilnih gradnikov mora biti brez šuma, od izvora do vseh flip-flopov naj potuje z enakimi zakasnitvami, potrebna je sinhronizacija asinhronih vhodov in signalov, ki prihajajo iz domene druge ure. V praksi najlažje načrtujemo, analiziramo in optimiziramo sinhrona sekvenčna vezja, kjer so vsi flip-flopi povezani na isto uro.

Omejitev: SHDL omogoča le opis pomnilnih gradnikov, pri katerih flip-flopi preklapljajo ob naraščajoči fronti ure z imenom clk. Signal clk ne deklariramo med priključki, saj je kar to privzet signal v modelu vezja s flip-flopi in registri.

Poenostavljen jezik **SHDL** uvaja zelo kompakten način opisa sinhronega pomnilnega gradnika s sekvenčnim prireditvenim stavkom. To je prireditveni stavek z operatorjem <=, ki določa, da se prireditev izvede ob fronti ure clk. Za opis osnovnega flip-flopa potrebujemo le eno vrstico:

```
q <= d
```

Flip-flop nastavlja podatkovni izhod ob vsakem ciklu ure. Če ima flip-flop na vhodu tudi sinhroni signal reset, bo dobil izhod ob vsakem ciklu ure eno izmed dveh vrednosti:

- 0, če je reset=1
- d, sicer

Model vezja flip-flopa s sinhronim signalom reset vsebuje dodaten pogoj:

```
if reset=1 then
  q <= 0
else
  q <= d
end
```

```
process(clk)
begin
 if rising_edge(clk) then
  if reset = '1' then
     q <= '0';
  else
     q <= d;
  end if;
 end if;
end process;
```

Predstavljen model vezja vsebuje najpogosteje uporabljene oznake za imena signalov D flip-flopa (d, reset, clk, q). Signale lahko poljubno poimenujemo (npr. vhod, nastavi, ura, izhod) in prevajalnik bo iz konteksta pogojev razbral, da opisujemo podatkovni flip-flop. Edina omejitev je poimenovanje ure v jeziku SHDL, ki je privzeto signal clk.

Nadgradnja flip-flopa

Če imamo v tehnološki knjižnici na voljo le flip-flop brez signala reset, mu lahko dodamo zunanjo logiko za sinhrono resetiranje stanja. Model flip-flopa s sinhronim signalom reset na ravni registrov (**RTL – Register Transfer Level**) sestavlja opis kombinacijskega dela vezja in pomnilnega elementa:

```
if reset=1 then nq=0 else nq=d end
q <= nq
```

kombinacijski del vezja

pomnilni element: D flip-flop

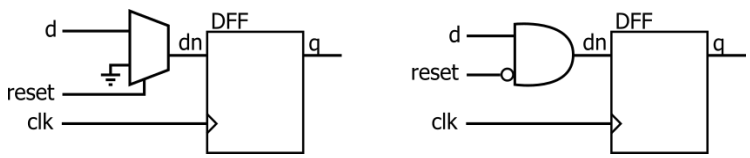
Uvedli smo nov signal (nq) za naslednje stanje flip-flopa, tj. naslednjo vrednost signala q. Kombinacijska logika definira naslednje stanje v odvisnosti od vhodov reset in d. Logiko lahko opišemo tudi s pogojnim prireditvenim stavkom:

```
nq = 0 when reset=1 else d
```

ali pa z logičnim izrazom:

```
nq = d and (not reset)
```

Model vezja na ravni registrov, ki vsebuje le funkcijski opis kombinacijskega vezja in osnovne pomnilne gradnike (DFF), je primeren za shematsko predstavitev. Shema vezja v prvem primeru vsebuje izbiralnik, v drugem pa logična vrata:



Register

Register ima krmilni signal z oznako omogoči (en, ce) ali naloži (load), ki določa ob katerem ciklu ure se vhod prenese na izhod: če je ta signal aktiven, se izhod nastavi na vrednost vhoda, sicer pa izhod ohranja stanje, kar bi v jeziku **SHDL** opisali s pogojem:

```
if en=1 then
  q <= d
else
  q <= q
end
```

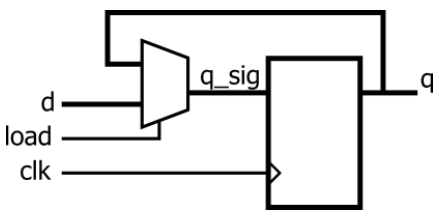
Eksplicitен zapis ohranjanja stanja v opisu pomnilnih gradnikov ni potreben. Če je izhod gradnika tudi izhodni priključek vezja, bi imeli v jeziku **VHDL** še dodatno komplikacijo. Izhodni priključek se namreč ne sme pojavljati na desni strani prireditvenega stavka in bi potrebovali dodaten notranji signal:

```
signal q_sig : std_logic := '0';

process(clk)
begin
 if rising_edge(clk) then
  if en = '1' then
     q_sig <= d;
  else
     q_sig <= q_sig;
  end if;
 end if;
end process;

q <= q_sig;
```

Kadar v strojno-opisnem jeziku ne določimo vrednosti signala, se bo vrednost ohranjala. To pa ravno modelira delovanje registra, ki ga opišemo bolj kompaktno:

`if en=1 then`
`  q <= d`
`end`
	`process(clk)`
	`begin`
	`  if rising_edge(clk) then`
	`  if en = '1' then`
	`  q <= d;`
	`  end if;`
	`  end if;`
	`end process;`

Pomnilni elementi v vezju FPGA že vsebujejo krmilni signal (en). Če bi imeli na voljo le register brez krmilnega vhoda za nalaganje nove vrednosti, bi ga naredili z dodatno logiko na vhodu. Shemo dobimo iz daljšega opisa, ki ga razdelimo na kombinacijski del in pomnilni element. Kombinacijsko vezje predstavlja izbiralnik, ki določa ali bo naslednje stanje enako q ali d:



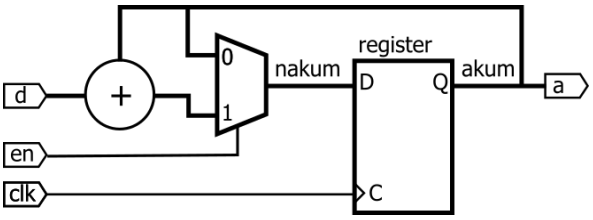
Akumulator

Akumulator je register, ki prišteva (akumulira) vhodno vrednost. Opišemo ga z *registrsko operacijo*, pri kateri je izhod kombinacijskega vezja povezan na register. V jeziku VHDL opišemo registrsko operacijo v sinhronem procesu , v SHDL pa s sekvenčnim operatorjem. Običajno ima akumulator še signal za omogočanje prištevanja (en):

```
entity Akumulator
 d: in u8;
 en: in u1;
 akum:  u8;
 a: out u8;
begin
 if en then
   akum <= akum + d
 end
 a = akum;
end
```

```
entity Akumulator is
 port (
   clk : in std_logic;
   d : in unsigned(7 downto 0);
   en : in std_logic;
   a : out unsigned(7 downto 0) );
end Akumulator;

architecture RTL of Akumulator is
 signal akum : unsigned(7 downto 0) := "00000000";
begin
 process(clk)
 begin
  if rising_edge(clk) then
   if en = '1' then
     akum <= akum + d;
   end if;
  end if;
 end process;
 a <= akum;
end RTL;
```

V opisu akumulatorja uporabimo notranji vektorski signal akum, da se izognemo omejitvam jezika VHDL glede zunanjih priključkov. Ker se pojavlja akum na levi in desni strani prireditvenega izraza, bi moral biti deklariran kot zunanji signal vrste **buffer** za katerega veljajo omejitve pri povezavi v hierarhično vezje. Če uporabljamo notranji vektor, se temu izognemo in ga na koncu le priredimo izhodnemu priključku (a).



Akumulatorju lahko dodamo izhod zero, ki sporoča, da je trenutna vrednost akumulatorja 0. Izhod je kombinacijski primerjalnik, ki ga opišemo s kombinacijskim prireditvenim stavkom:

```
zero = 1 when akum=0 else 0
```

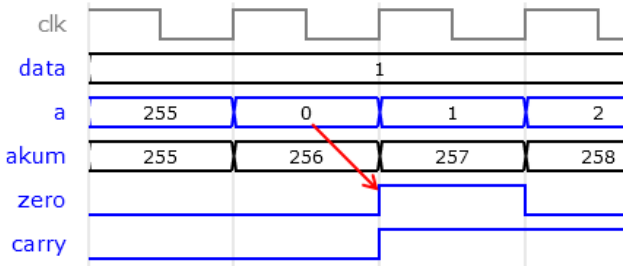
Nadgradimo akumulator še z izhodom carry, ki se postavi na 1, kadar pride po seštevanju vrednosti do prenosa na najvišje mesto. V tem primeru naj bo akum deklariran kot 9-bitni vektor, da bomo na najvišjem bitu dobili izhodni prenos.

```
a: out u8;
akum:  u9;
zero,carry: out u1

a = akum(7:0)
carry = akum(8)
zero = 1 when akum(7:0)=0
         else 0
```

```
a <= akum(7 downto 0);
carry <= akum(8);
zero <= '1' when akum(7 downto 0)=0
            else '0';
```

Razdelitev opisa akumulatorja z zastavicami na pomnilni del in kombinacijsko logiko je pomembna zaradi časovne usklajenosti izhodnih signalov. Če bi želeli narediti opis vezja v jeziku **VHDL** še bolj kompakten in bi dali stavke za izhodne zastavice v proces, bi imela izhoda carry in zero zakasnitev:

```
process(clk)
begin
 if rising_edge(clk) then
  if en = '1' then
     akum <= akum + resize(data,9);
  end if;
  carry <= akum(8);
  if akum(7 downto 0) = 0 then
     zero <= '1';
  else
     zero <= '0';
  end if;
 end if;
end process;
```



Zakasnitev med vrednostjo akum in zero oz. carry!

Prireditev signalu znotraj procesa s pogojem za fronto ure (sinhroni proces) namreč pomeni, da je signal izhod iz flip-flopa in da se njegova vrednost spremeni z zakasnitvijo. Kadar je izhod odvisen od vrednosti signala, ki se nastavlja ob isti uri (v našem primeru signal akum) imamo dodaten cikel zakasnitve in nepotrebne pomnilne elemente (flip-flope) na izhodu. Pri opisu v orodju SHDL pa moramo biti pozorni na vrsto uporabljenega prireditvenega operatorja.

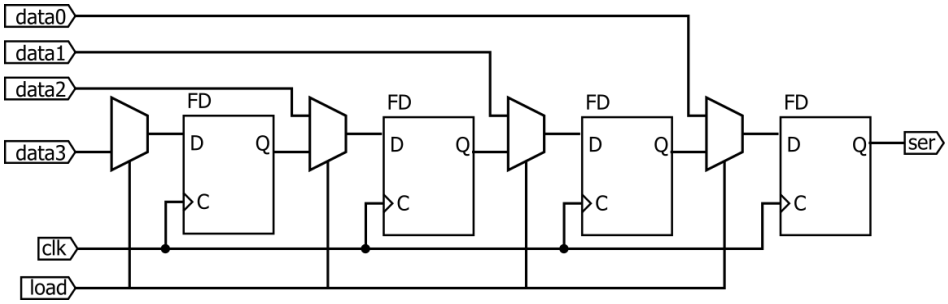
Pomikalni register

Pomikalne registre uporabljamo za pretvorbo paralelnih podatkov v serijske in obratno. Najprej bomo predstavili opis pomikalnega registra s paralelnim vhodom in serijskim izhodom (PISO):

```
entity piso
 data: in u4
 load: in u1
 ser: out u1
 reg: u4
begin
 if load then
   reg <= data
 else
   reg <= reg srl 1
 end
 ser = reg(0)
end
```

```
signal reg : unsigned(3 downto 0) := "0000";

process(clk)
begin
 if rising_edge(clk) then
  if load = '1' then
     reg <= data;
  else
     reg <= shift_right(reg,1);
  end if;
 end if;
end process;

ser <= reg(0);
```

Model vezja vsebuje 4-bitni notranji signal reg, ki nalaga vrednost iz vhoda data, jo ob uri pomika v desno (**srl**) in pošilja na serijski izhod ser. Sestavljen je iz flip-flopov in izbiralnikov.



Pomikalni register, ki pomika vrednosti v desno, bo najprej dal na izhod najmanj pomemben bit, reg(0), nato pa po vrsti še ostale. Če potrebujemo na izhodu najprej najbolj pomemben bit, uporabimo pomikanje v levo. Za pomikanje bitov v levo zapišemo izraz z operatorjem **sll** (v jeziku VHDL pa s funkcijo shift_left)[1].

```
reg <= reg sll 1
```

```
ser = reg(3)
```

Ob pomikanju se na izpraznjeno mesto postavi vrednost 0. Če želimo na izpraznjeno mesto prenesti vrednost iz dodatnega serijskega vhoda (npr. sd), bi lahko naredili takole:

```
reg <= (reg sll 1) or sd
```

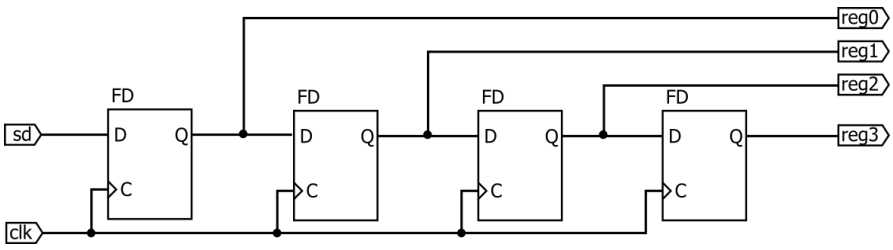
Takšen zapis bi uporabili na primer pri izvedbi pomikanja z mikroprocesorjem. Model vezja pa raje opišemo z operatorjem sestavljanja, ki ne zahteva dodatne logike. Pomik bi opisali takole:

```
reg <= reg(2:0) & sd
```

Delovanje stavka z operatorjem sestavljanja najlažje predstavimo tako, da zapišemo tabelo s posameznimi biti na levi in na desni strani. Vezje je sestavljeno iz štirih zaporedno vezanih flip-flopov:

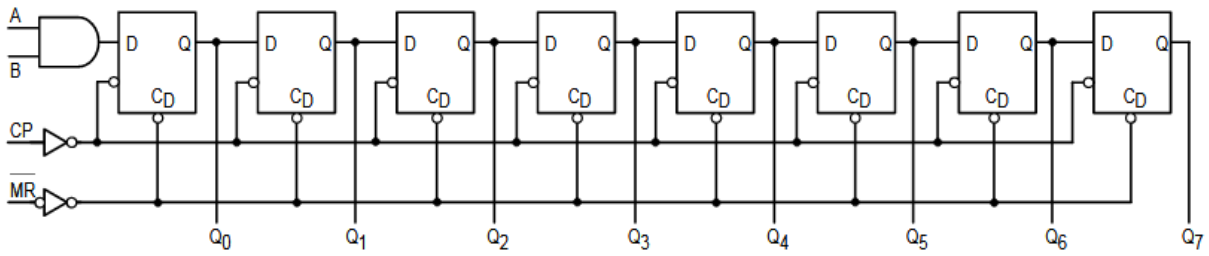
reg(3)	reg(2)
reg(2)	reg(1)
reg(1)	reg(0)
reg(0)	sd

Stavek opisuje 4-bitni pomikalni register s serijskim vhodom in paralelnim izhodom (SIPO), ki je sestavljeno iz štirih flip-flopov.



[1] Operatorja sll in srl sta definirana v jeziku VHDL, vendar se za pomikanje (ne)predznačenih vektorjev priporoča uporaba funkcij shift_left() in shift_right().

Poglejmo še primer opisa vezja 74165, ki je pomikalni register s paralelnim in serijskim vhodom ter serijskim izhodom. Vezje ima dva serijska vhoda, ki sta vezana na logična vrata and in krmilni signal mr za resetiranje flip-flopov.



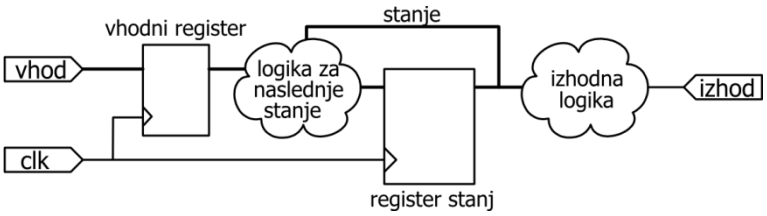

```
entity sipo
 a,b,mr: in u1;
 q: out u8;
 d: u1
begin
 d = a and b
 if mr=0 then
  q <= 0
 else
  q <= q(6:0) & d
 end
end
```

```
d <= a and b;
q <= q_sig;

process(clk)
begin
 if rising_edge(clk) then
  if mr = '0' then
     q_sig <= to_unsigned(0, 8);
  else
     q_sig <= q_sig(6 downto 0) & d;
  end if;
 end if;
end process;
```

4. Postopkovni opis

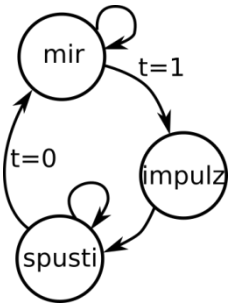
Sekvenčni stroj

Sekvenčni stroji ali končni stroji stanj (finite-state machine) so vezja z registrom in povratno zanko, pri katerih je novo stanje odvisno od predhodnega in od vhodov. V splošnem je takšno vezje sestavljeno iz registra stanj, logike za naslednje stanje in izhodne logike. Če se vhodi spreminjajo hitro in asinhrono, potrebujemo tudi vhodni register:



Sekvenčne stroje načrtujemo s pomočjo **diagrama prehajanja stanj**.

Naredimo vezje, ki ob pritisku tipke generira impulz dolžine ene periode ure. Predpostavljamo, da je ura precej hitrejša kot pritisk tipke. Sekvenčni stroj je v začetku v mirovnem stanju, dokler ni signal t = '1'. Ob fronti ure gre v stanje impulz in ob naslednji fronti v stanje spusti. V tem stanju čaka tako dolgo, dokler ni t = '0', nato pa se vrne spet v mirovno stanje. Vezje naj ob stanju impulz na izhod postavi vrednost '1'.



Stanja opišemo z naštevnim podatkovnim tipom: v jeziku VHDL deklariramo nov podatkovni tip in nato deklariramo signal(e), v SHDL pa to storimo v eni vrstici:

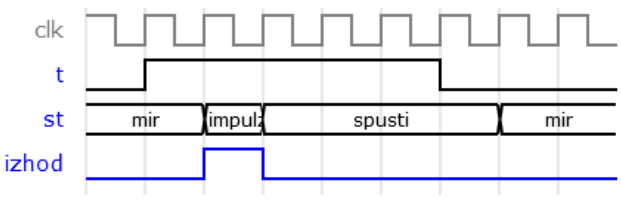
`st: (mir, impulz, spusti)`	`type stanja is (mir, impulz, spusti);` `signal st: stanja;`

Model sekvenčnega stroja sestavlja opis logike za prehajanje stanj, ki vsebuje tudi register, in opis izhodne kombinacijske logike. Opis prehajanja stanj naredimo s pogojnim ali izbirnim stavkom za vsako stanje ter dodatnimi pogoji za prehod. Register stanj je določen z uporabo operatorja <= v SHDL oz. s sinhronim procesom v jeziku VHDL:

```
if st=mir then
 if t=1 then st<=impulz end
elsif st=impulz then
 st<=spusti
elsif st=spusti then
 if t=0 then st<=mir end
end

izhod=1 when st=impulz else 0
```

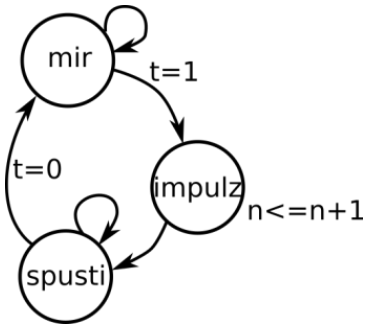
```
process(clk)
begin
 if rising_edge(clk) then
  case st is
   when mir =>
     if t = '1' then
        st <= impulz;
     end if;
   when impulz =>
     st <= spusti;
   when spusti =>
     if t = '0' then
        st <= mir;
     end if;
   when others => null;
  end case;
 end if;
end process;

izhod <= '1' when st = impulz else '0';
```

Če je izhod sekvenčnega stroja aktiven le ob enem stanju, ga najučinkoviteje opišemo s pogojnim prireditvenim stavkom **when**…**else**. Delovanje vezja preverimo s simulacijo.



Sekvenčni stroj za detekcijo pritiska tipke bi lahko uporabili tako, da bi ob vsakem pritisku tipke povečali vrednost sinhronega števca (n). Opis števca naredimo z dodatno vrstico (n <= n + 1) v modelu vezja, ob pogoju za stanje impulz.

```
if st=mir then
 if t=1 then st<=impulz end
elsif st=impulz then
 st <= spusti
 n <= n + 1
elsif st=spusti then
 if t=0 then st<=mir end
end

izhod=1 when st=impulz else 0
```



V implementiranem vezju so stanja shranjena v flip-flopih in kodirana z vektorskim zapisom. Kodiranje stanj izvede orodje za sintezo vezja, ki določi vrsto kodiranja:

- binarno: "00" = mir, "01" = impulz, "10" = spusti,
- kodiranje z eno enico: "001" = mir, "010" = impulz, "100" = spusti,
- Grayevo kodiranje ali kakšno drugo.

Vezje za blokovno šifriranje podatkov

Postopkovni opis vezja je primeren za načrtovanje vezij, ki izvajajo računalniške algoritme v strojni opremi. Vzemimo primer poenostavljenega algoritma za blokovno šifriranje podatkov. Namesto 64- ali več-bitnih blokov bomo uporabili le dva 8-bitna znakovna vhoda in štiri iteracije šifriranja s skrivnimi ključi.

Razvoj algoritma poteka pogosto kar v programskih jezikih, zato bomo opisali naš primer v jeziku C++. Algoritem uporablja pri šifriranju štiri konstantne 8 bitne ključe, ki jih deklariramo kot zbirko s štirimi vrednostmi. V prvem koraku razdelimo vhodni podatek na dva dela (a in b), ki ju v programu preberemo iz standardnega vhoda (cin):

```
int main(void)
{
  char a, b, a_nov, b_nov;
  char k[] = {0xff, 0x0f, 0x03, 0x30};

  cin >> a;
  cin >> b;

  for (int i=0; i<=3; i++) {
    a_nov = b;
    b_nov = (b + k[i]) ^ a;
    a = a_nov;
    b = b_nov;
  }

  cout << a << b;
}
```

Jedro algoritma tvorijo štiri iteracije šifriranja, v katerih se vsakokrat izračunajo nove vrednosti a in b. Nova vrednost a je kar enaka b, nova vrednost b pa se izračuna z izrazom, v katerem je vsakokrat uporabljen del ključa k(i).

Vezje naj ima 16-bitni vhod in izhod, ključi pa so definirani v pomnilniku ROM, v jeziku SHDL:

```
k: 4u8 = 0xff, 0x0f, 0x03, 0x30
```

oz. v jeziku VHDL:

```
type k_type is array (0 to 3) of unsigned(7 downto 0);
signal k : k_type := (x"ff", x"0f", x"03", x"30");
```

1. Algoritem v kombinacijskem vezju

Kombinacijsko vezje za šifriranje naredimo tako, da razvijem zanko. Po vsaki iteraciji zanke moramo shraniti rezultat v nove signale, sicer bi dobili v vezju kombinacijsko zanko.

```
a0 = vhod(15:8);
b0 = vhod(7:0);

a1 = b0;
b1 = (b0 + k(0)) xor a0;

a2 = b1;
b2 = (b1 + k(1)) xor a1;

a3 = b2;
b3 = (b2 + k(2)) xor a2;

a4 = b3;
b4 = (b3 + k(3)) xor a3;

izhod = a4 & b4;
```

Vsak izračunani izraz smo priredili novemu signalu, ki v vezju predstavlja izhod iz kombinacijske logike (seštevalnika in operacije **xor**). Vezje torej vsebuje 4 seštevalnike in 4 vektorske **xor** operatorje, ki predstavljajo kombinacijske zakasnitve med vhodom in izhodom.

Zanimajo nas lastnosti opisanega vezja: poraba površine in hitrost obdelave podatkov. Predstavljajmo si, da podatki prihajajo zaporedoma na vhod vezja kateremu dodamo vhodni in izhodni register. Naredimo RTL sintezo in tehnološko preslikavo modela vezja in poglejmo rezultate. Izbrali smo tehnologijo vezij CPLD Xilinx XC95288XL-10. Rezultat tehnološke preslikave je:

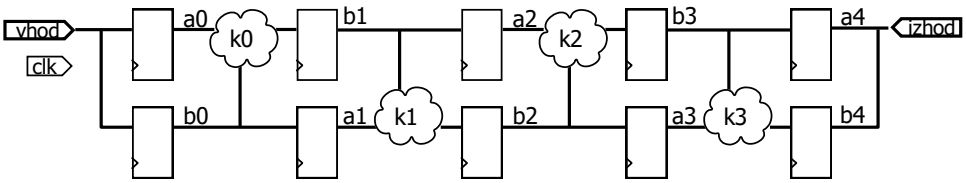
Površina: 56 makrocelic, 32 flip-flopov
Frekvenca: 26.8 MHz

Število uporabljenih flip-flopov se ujema s številom bitov v vhodnem in izhodnem registru, površina in frekvenca pa sta odvisni od lastnosti tehnologije in nastavitev optimizacije. Maksimalno frekvenco vezja izračuna programska oprema na podlagi ocenjenih zakasnitev po tehnološki preslikavi. Frekvenca vezja je očitno omejena z zakasnitvijo med vhodom in izhodom na kombinacijski logiki. Na kombinacijski poti signala so poleg logičnih vrat štiri zaporedni seštevalniki, ki predstavljajo največjo zakasnitev. Pokazali bomo, da z drugačno izvedbo vezja dosežemo precej boljše rezultate.

2. Paralelno sekvenčno vezje (cevovod)

Če prihaja na vhod vezja podatkovni tok, npr. nov podatek ob vsakem urnem ciklu, bomo naredili cevovod (*pipeline*). To je sinhrono sekvenčno vezje, kjer so vsi registri povezani na isto uro. Takšna vezja so zelo primerna za načrtovanje, optimizacijo in analizo časovnih parametrov. Načrtovanje sinhronih vezij je še posebej priporočljivo v tehnologiji **programirljivih vezij**, ki imajo običajno malo globalnih povezav za distribucijo ure do vseh celic.

Sinhrono vezje opišemo v jeziku SHDL zelo enostavno – kombinacijske prireditvene operatorje zamenjamo s sekvenčnimi. V jeziku VHDL bi pa prireditvene stavke prenesli v sinhroni proces.



Sedaj predstavlja vsak prireditveni stavek register in vezje je narejeno v obliki 4 stopenjskega cevovoda. Pri tej izvedbi vezja potrebujemo 4 cikle, da se rezultat pojavi na izhodu.

Površina: 80 makrocelic, 80 flip-flopov
Frekvenca: 90.9 MHz

Površina vezja se je v primerjavi s kombinacijsko izvedbo povečala zaradi dodatnih registrov, vendar je precej višja tudi frekvenca delovanja vezja. Razlog za to je, da najdaljša kombinacijska pot med registroma sedaj vsebuje le eno seštevanje in operacijo **xor**. Vezje lahko obdela podatkovni tok s hitrostjo 181.8 MByte/s (na vhodu in izhodu sta po 2 byta), v primerjavi s kombinacijsko izvedbo, ki obdela le 53.6 MByte/s.

Najvišja frekvenca delovanja je v sinhronem vezju omejena z največjo kombinacijsko zakasnitvijo na poti med dvema registroma. Programska oprema analizira zakasnitve na vseh signalnih poteh med pari registrov in poišče kritično pot, na kateri je zakasnitev največja. Pri načrtovanju na nivoju RTL lahko sami optimiziramo vezje z uravnoteženo postavitvijo registrov.

Če iz modela vezja odstranimo vsak drugi register in naredimo tehnološko preslikavo dobimo:

Površina: 54 makrocelic, 48 flip-flopov
Frekvenca: 46.1 MHz

Površina vezja je manjša, vendar imamo sedaj na kritični poti po dve seštevanji in operaciji **xor**, zato je najvišja frekvenca približno pol manjša.

3. Sekvenčni procesor

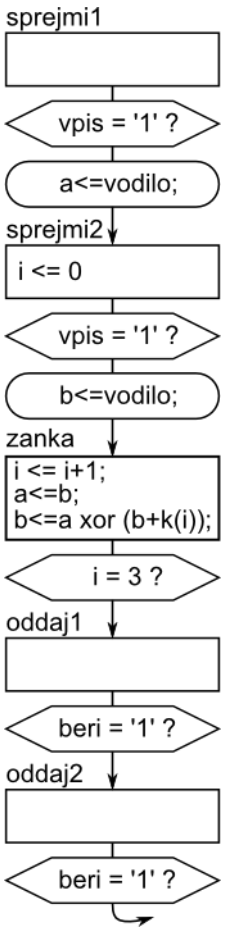
Najmanjše vezje bo v obliki procesorja, ki sekvenčno (zaporedno) šifrira podatke. Vezje naredimo v obliki algoritmičnega stroja stanj (*Algorithmic State Machine*), ki ga modeliramo s posplošenim diagramom stanj s tremi grafičnimi elementi:

- pravokotnik opisuje stanje vezja in vsebuje stavke, ki se izvršijo v tem stanju,
- v rombu ali šestkotniku je zapisan pogoj,
- v ovalu pa so stavki, ki se izvršijo ob pogoju.

Tokrat bomo podatke prenašali po enem 8-bitnem vodilu. Algoritem začnemo izvajati v stanju sprejmi1, kjer čakamo na signal za vpis podatka iz vodila. Po dveh zaporednih vpisih gremo v stanje zanka, v katerem se indeks povečuje od 0 do 3. V zanki vsakokrat naredimo eno iteracijo šifriranja in rezultat shranimo v registra a in b. Nato gremo v stanje oddaj1, kjer čakamo na potrditev, da je prvi rezultat prebran, v stanju oddaj2 pa še beremo še drugi rezultat. Na koncu se vrnemo v prvotno stanje in vezje je pripravljeno na ponovni cikel šifriranja. Poglejmo diagram ASM in jedro opisa vezja v jezikih SHDL in VHDL :

```
if st=sprejmi1 then
  if vpis then
    a<=vhod;
    st<=sprejmi2
  end
elsif st=sprejmi2 then
  i <= 0
  if vpis then
     b<=vhod;
     st<=zanka
  end
elsif st=zanka then
  i <= i + 1
  a <= b;
  b <= a xor (b + k(i))
  if i=3 then
    st<=oddaj1
  end
elsif st=oddaj1 then
  if beri then
    st<=oddaj2
  end
else
  if beri then
    st<=sprejmi1
  end
end
```

```
process(clk)
begin
   if rising_edge(clk) then
      case stanje is
         when sprejmi1 =>
           if vpis='1' then
             a <= vodilo;
             stanje <= sprejmi2;
           end if;
         when sprejmi2 =>
           i <= 0;
           if vpis='1' then
             b <= vodilo;
             stanje <= zanka;
           end if;
         when zanka =>
           i <= i + 1;
           a <= b;
           b <= (b + k(i)) xor a;
           if i=3 then
             stanje <= oddaj1;
           end if;
         when oddaj1 =>
           if beri='1' then
             stanje <= oddaj2;
           end if;
         when others =>
           if beri='1' then
             stanje <= sprejmi1;
           end if;
      end case;
   end if;
end process;
```



Rezultat sinteze in tehnološke preslikave:

Površina: 31 makrocelic, 22 flip-flopov
Frekvenca: 84.7 MHz

Vezje zasede manjšo površino, kljub temu, da je opis daljši in kompleksnejši od kombinacijske ali cevovodne izvedbe! Na velikost vezja vpliva predvsem število in vrsta kombinacijskih operatorjev.

5. Pravila načrtovanja

Načrtovanje modela vezja se razlikuje od razvoja programske opreme, čeprav so stavki strojno-opisnega jezika podobni stavkom programskih jezikov. Razlika je v pomenu in načinu izvajanja stavkov, ki predstavljajo v vezju sočasno delujoče gradnike. Navedli bomo nekaj pravil oz. najpogostejših začetniških napak pri modeliranju sinhronih digitalnih vezij.

1. Signalu priredimo vrednost le enkrat v posameznem bloku

   Če sta dve ali več zaporednih prireditev v istem bloku, se bo izvedla le zadnja prireditev, npr.

```
a <= 0
a <= a + 1
```

   V programskem jeziku, bi se spremenljivka najprej postavila na 0, potem pa povečala na 1. Strojno-opisni jezik pa prvi stavek ignorira, če je zapisan v procesnem bloku (npr. znotraj pogojnega stavka), ali pa predstavlja kratek stik!

2. Ne uporabljajmo zapahov

   Zapah (*latch*) dobimo, kadar mora nek signal ohranjati vrednost brez prisotnosti ure. Zapah deluje asinhrono, zato lahko povzroča težave v realnem vezju. Primer:

```
if en then
  y = d
end
```

   Signal y mora ohranjati vrednost, ko je en=0, zato je v vezju zapah. Kombinacijsko vezje mora imeti izhod določen v vseh primerih (**if** in **else** ali pa privzeta vrednost).

3. Izogibajmo se kombinacijskih zank

   S kombinacijsko povratno zanko opišemo zapah ali pa vezje, ki nekontrolirano spreminja vrednosti (npr. oscilira z neznano frekvenco). Primer kombinacijske zanke (SHDL):

```
c = not c
a = a + 1
```

   Kombinacijski zanki se izognemo tako, da v opis povratne zanke dodamo register – v jeziku VHDL zapišemo prireditveni stavek znotraj pogoja za fronto ure, v SHDL pa z operatorjem <=

4. Vezje naj bo sinhrono

   Jezik SHDL ne omogoča opisa asinhronih pomnilnih elementov, v jeziku VHDL je to mogoče, vendar ni priporočljivo. Asinhroni signali povzročajo težave, npr. metastabilnost flip-flopov, ki jo s simulacijo ne zaznamo. Asinhrone vhode v sekvenčne stroje je priporočljivo sinhronizirati z uro tako, da jih povežemo na vhodni register.

   V sinhronem vezju so vsi flip-flopi vezani na isto uro. Kadar potrebujemo v delu vezja nižjo frekvenco, ne uporabimo klasični delilnik, ki bi imel na izhodu uro, ampak delilnik z izhodom za omogočanje (enable).

5. Stanje pomnilnih elementov

   Pomnilnim elementom (flip-flopi, RAM) stanje nastavljamo z enim stavkom. Če imamo več prireditvenih stavkov za nastavljanje stanja, se morajo pogoji izključevati, v nasprotnem primeru se izvede le zadnja prireditev (pravilo 1).

   Branje običajnega (*single-port*) pomnilnika ROM ali RAM modeliramo le z enim stavkom, da bo sintetizator uporabil ustrezen gradnik strojne opreme.

6. Zakasnitve naj določa ura

   V jeziku VHDL lahko modeliramo zakasnitve (npr. v testni strukturi), vendar jih ne moremo sintetizirati. Zakasnitve signala naredimo s flip-flopi: vsak flip-flop zakasni vhod za en cikel ure. Najhitrejšo spremembo izhoda dobimo s kombinacijskim vezjem, daljše zakasnitve pa naredimo s sinhronimi števci.

7. Operacije in optimizacija

   Operacije z vektorji predstavljajo različno velika vezja. Kadar je mogoče, nadomestimo kompleksnejše operacije z enostavnejšimi. Glede na računsko zahtevnost (velikost vezja, zakasnitve) jih razdelimo na:
   - množenje (najzahtevnejše)
   - seštevanje, odštevanje (zakasnitve zaradi prenosa)
   - logične operacije in splošno pomikanje (izbiralniki)
   - sestavljanje vektorjev in izbiranje bitov (le povezave, brez logičnih elementov)

   Velikost vezja zmanjšamo z izbiro operacij in ponovno uporabo računskih gradnikov vezja (zaporedna obdelava podatkov). Hitrost delovanja povečamo z vzporednim računanjem in uporabo registrov (cevovod).