Načrtovanje digitalnih vezij

Predstavitev domače strani
http://ndv.fe.uni-lj.si
in izvajanja predmeta

Načrtovanje digitalnih vezij

Uvod v logična vezja:
CAD orodja in VHDL

Pristopa k načrtovanju

- Tradicionalni
  - Se zanaša na matematične modele
  - Analitični pristopi
  - Zahteva popoln vpogled in razumevanje problema
  - Uporaben za manjše izzive (vezja)
  - Nezadosten za reševanje večjih (realnih) problemov
- CAD
  - Programi temeljijo na matematičnih modelih in analitičnih pristopih
  - Pregleden do uporabnika
  - Več podrobnosti je zabrisanih
  - Uporaben za reševanje realnih problemov

Uvod v CAD orodja

- CAD sistem navadno vključuje naslednja orodja za načrtovanje
  - Vnos načrtovanih funkcij
    (Design entry)
  - Sinteza in optimizacija načrtovanih funkcij
    (Synthesis and optimization)
  - Simulacija načrtovanih funkcij
    (Simulation)
  - Fizična izvedba in načrtovanje
    (Physical design and implementation)

Vnos načrtovanih funkcij

- Postopek vnosa opisa načrtovanih funkcij v CAD sistem
  (*design entry)*
- Tipične metode za vnos načrtovanih funkcij
  - Pravilnostne tabele
    (truth tables)
  - Shematski vnos funkcije
    (schematic capture)
  - Jezik strojnega opisa vezja
    (Hardware Description Language)

Jeziki strojnega opisa vezja

- Razširjeni HDL jeziki
  - VHDL (VHSIC Hardware Description Language, kjer je VHSIC: very-high-speed integrated circuit)
  - Verilog
  - MyHDL, PSHDL ...
- VHDL in Verilog sta standardizirana (VHDL: IEEE Std 1076-2008)
- Verilog:VHDL – kateri je boljši?
  VHDL: enoumen, ADA-like
  Verilog: kontekstualen
  (blocking, nonblocking, casex, casez) C-like

Sinteza

- ***Prevajanje***
  (*compiling*)
- **Optimiranje** glede na hitrost in/ali velikost – logična optimizacija
  (logic synthesis/logic optimization)
- **Tehnološka postavitev**
  (technology mapping)
- **Sinteza postavitve**
  (layout synthesis)

Simulacija

- **Funkcionalna simulacija**

  (*functional simulation)*
  praviloma v obliki časovnega diagrama

  Uporabnik sam preveri dobljene izhode vezja s svojimi pričakovanimi izhodi.

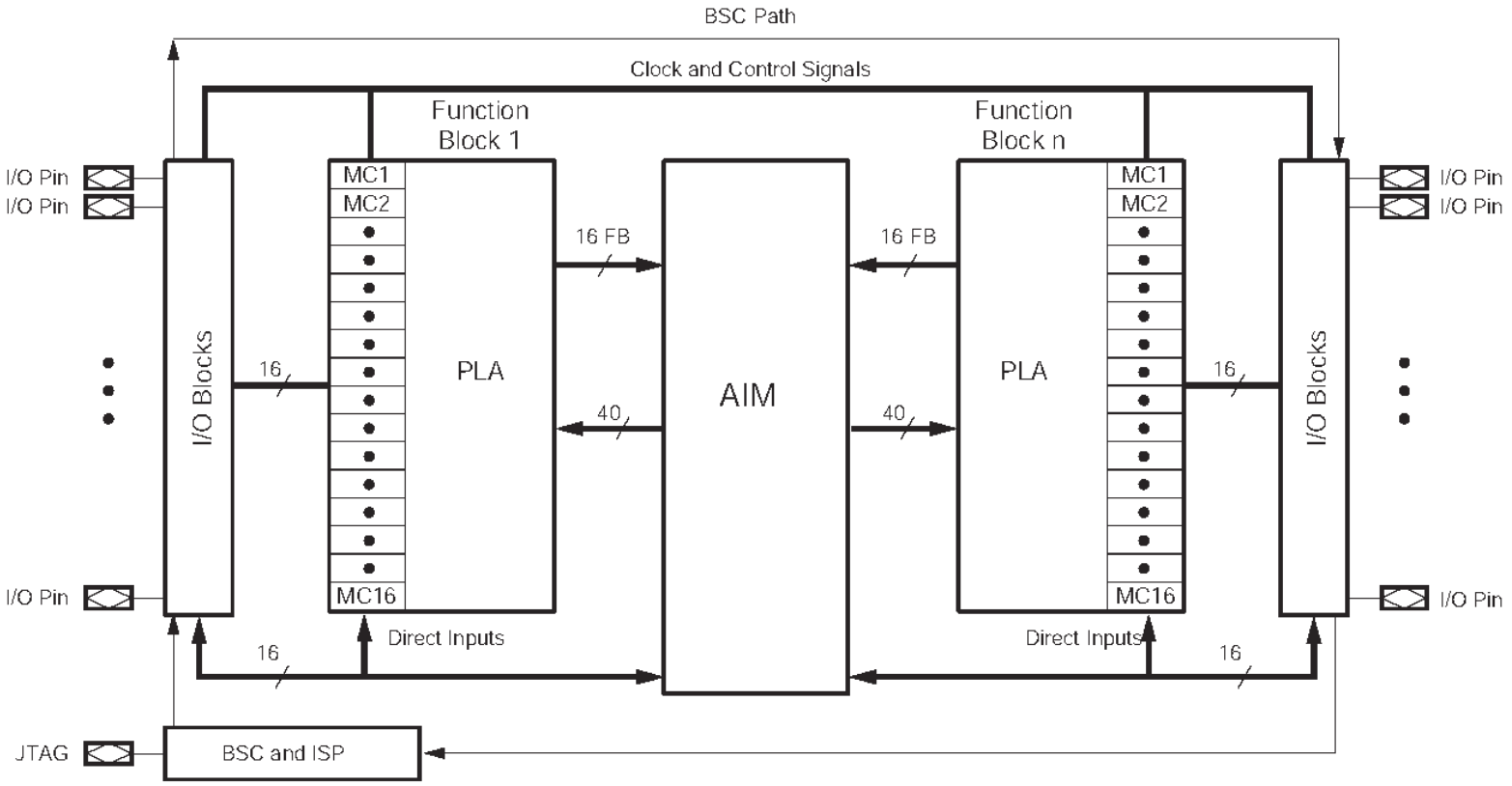
- **Časovna simulacija**

  Simulatorji navadno privzamejo, da je časovna zakasnitev širjenja preko več logičnih vrat zanemarljiva. To lahko modeliramo.

Načrtovanje digitalnih vezij

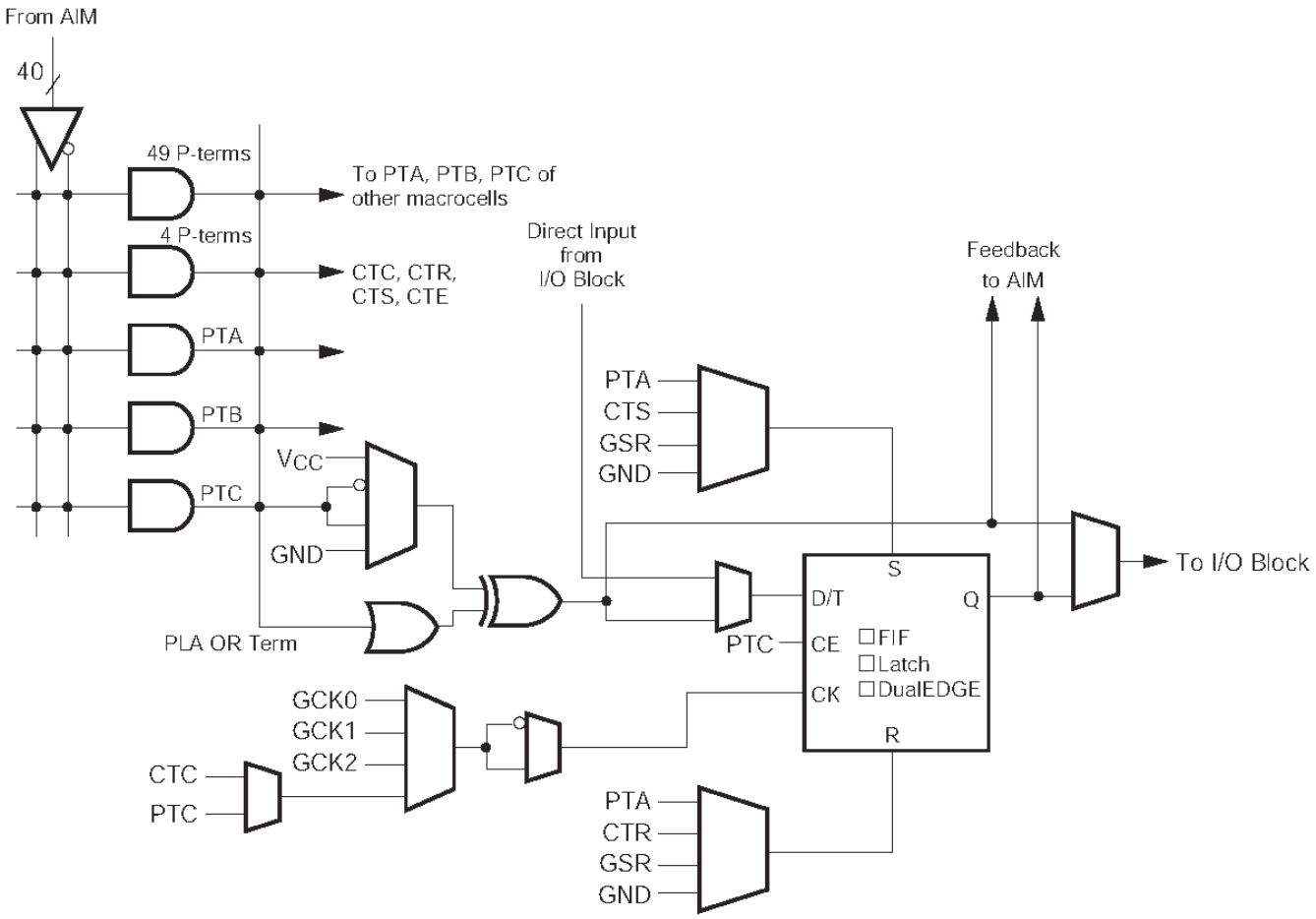
Programabilna vezja
CPLD in FPGA

CoolRunner II CPLD



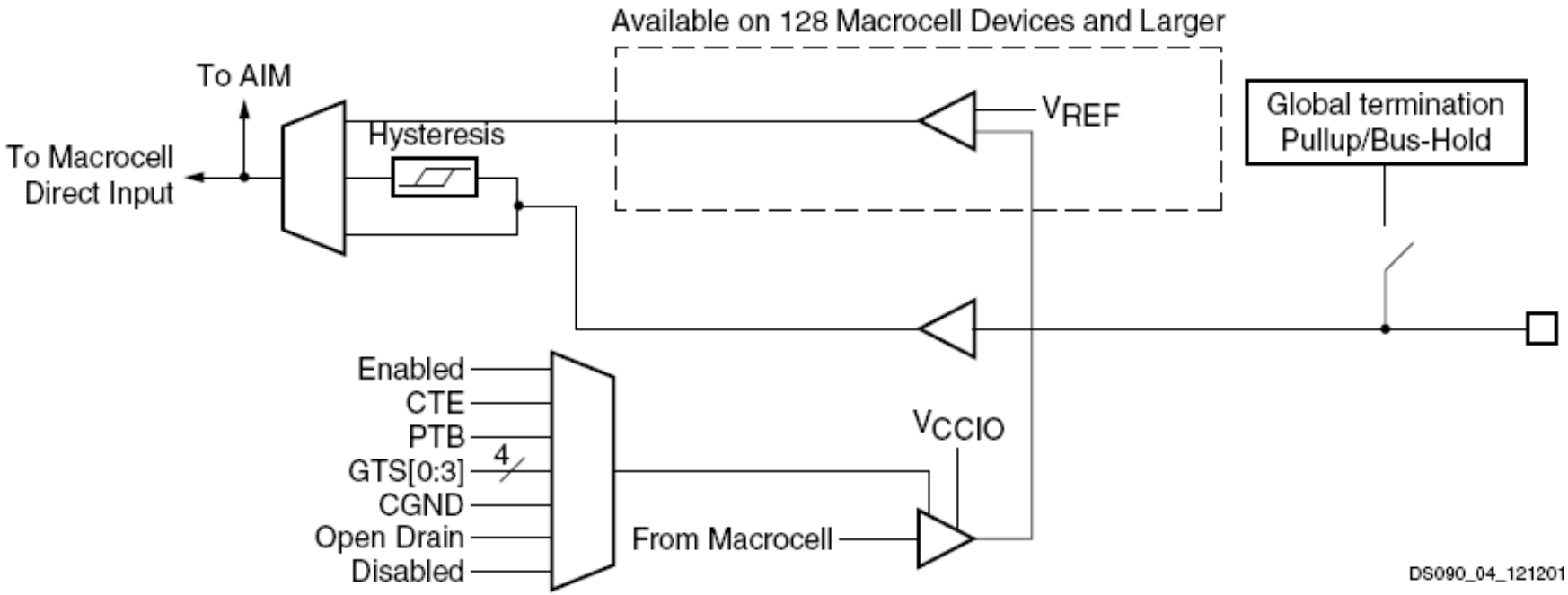
Vir: [Xilinx DS090](#).

CoolRunner II - makrocelica



Vir: Xilinx DS090, XAPP376

I/O priključek CoolRunner II CPLD

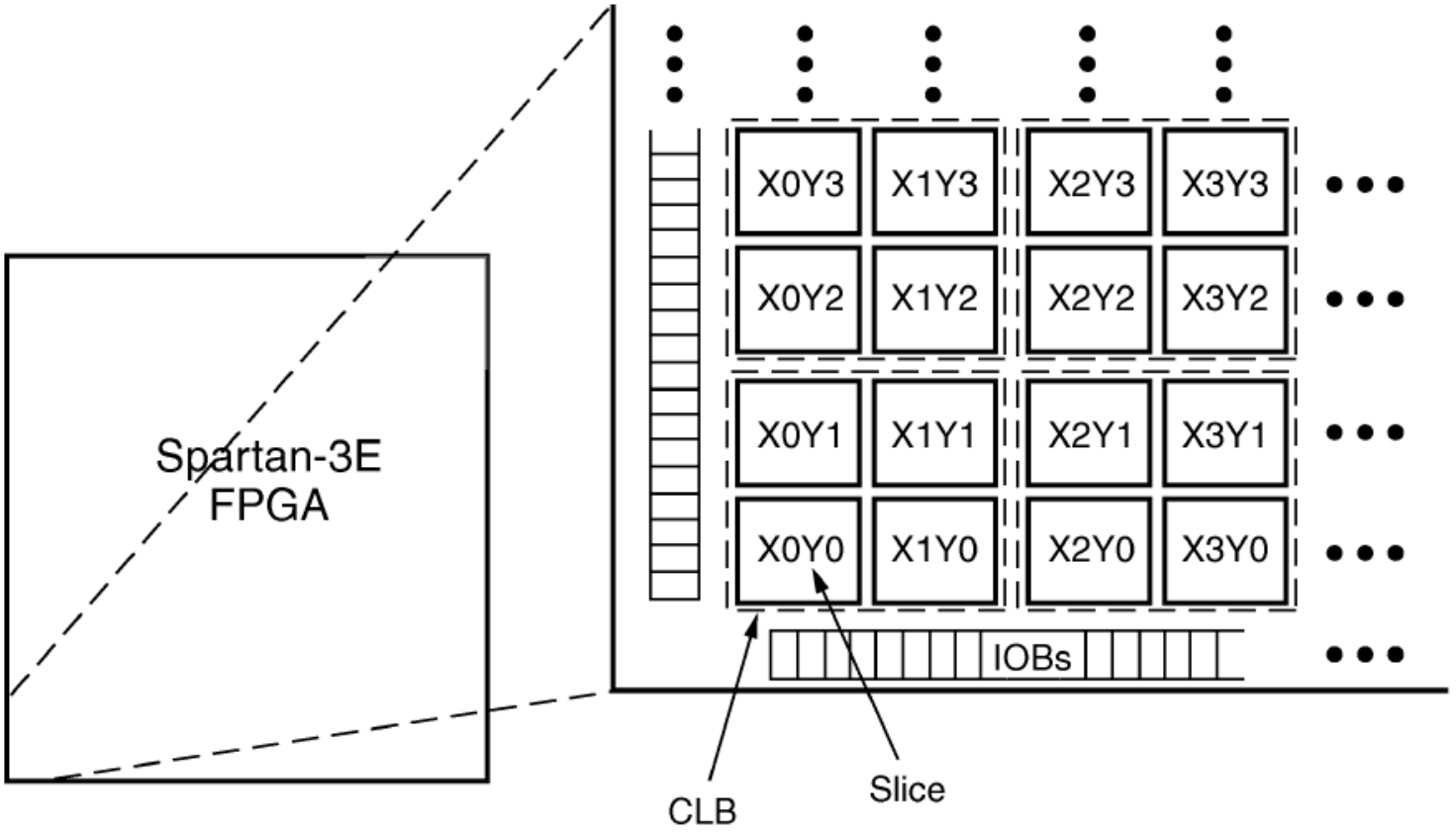


FPGA vezja – Spartan 3E

Osnovni funkcijski bloki:

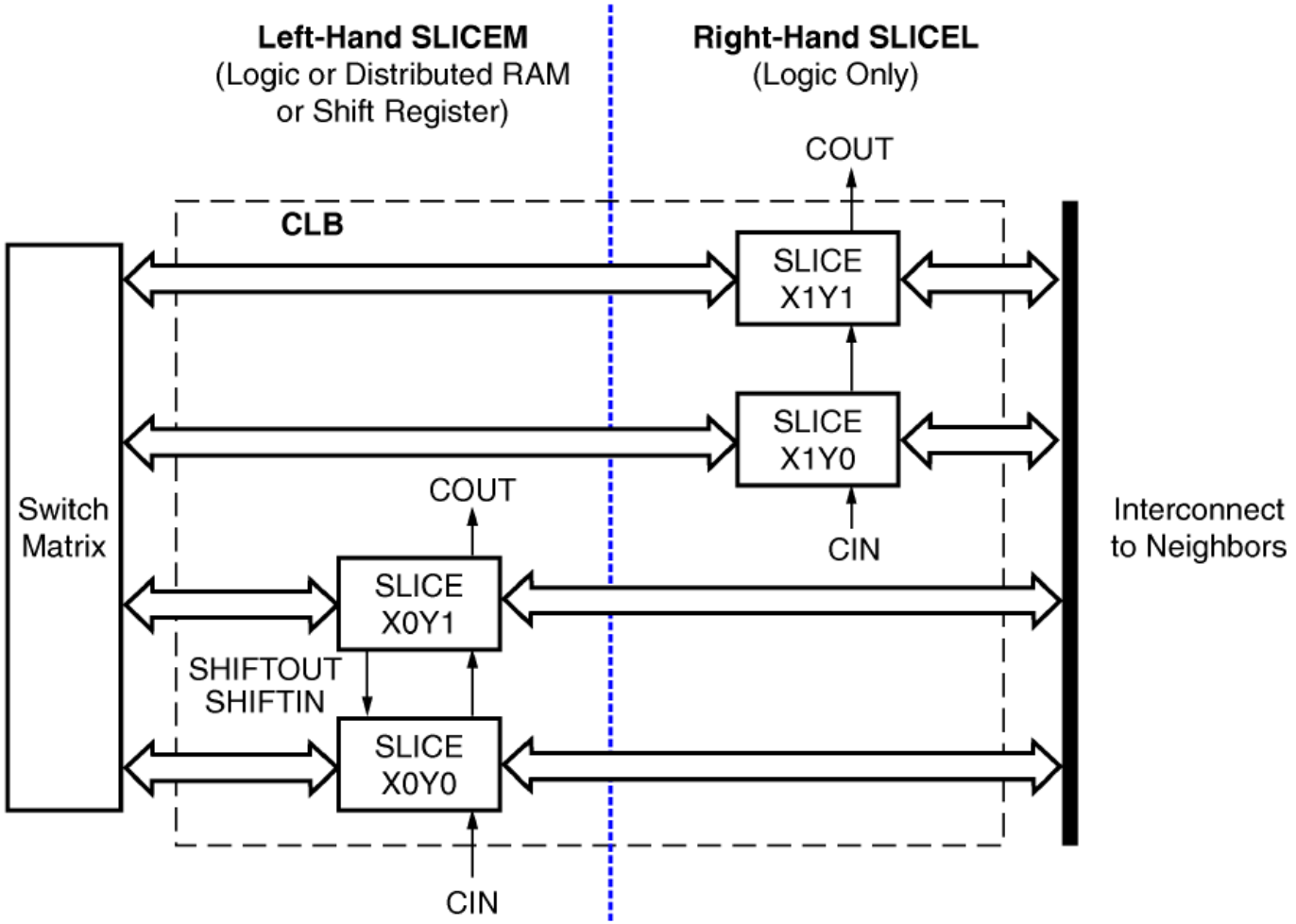
- **vhodno-izhodni bloki** (IOB - Input/Output Blocks) omogočajo prenos podatkov med notranjo logiko čipa in vhodno-izhodnimi priključki.
- **konfigurabilni logični bloki** (CLB - Configurable Logic Blocks) vsebujejo programabilne LUT (Look-Up Table) tabele v katerih implementiramo logične funkcije ter pomnilne celice (flip-flope oz. zatiče),
- **dvokanalni blokovni RAM** (BRAM) pomnilnik velikosti 18 Kbit,
- **množilniki** dveh 18-bitnih števil,
- **vezje za upravljanje z uro** (DCM - Digital clock Manager) omogoča ter skrbi za distribucijo urinega signala v FPGA vezju, za zakasnitev in fazni zamik ure, za množenje in deljenje frekvence urinega signala ter odpravljanje zdrsa urinega signala (clock skew).

Spartan 3E

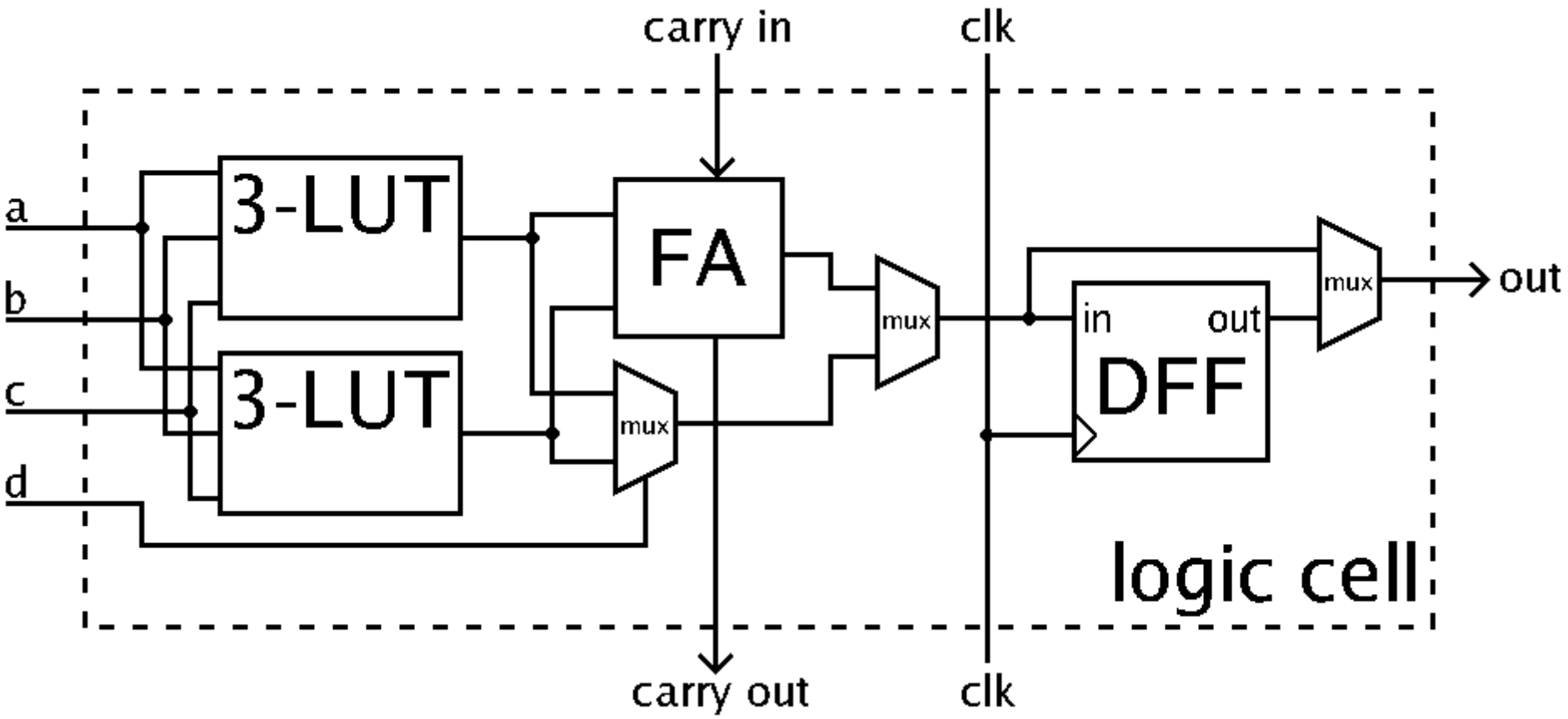


CLB=4 rezine (ang. slice)

Spartan 3E – sestava CLB

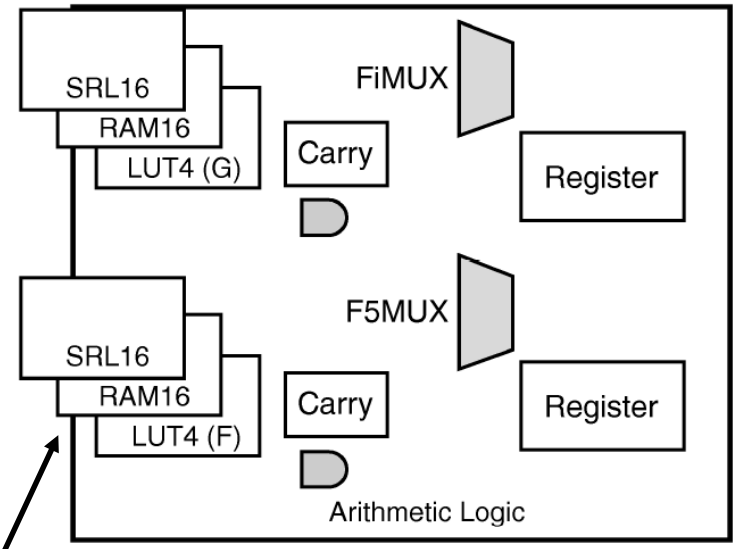


Poenostavljena celica CLB v FPGA vezjih

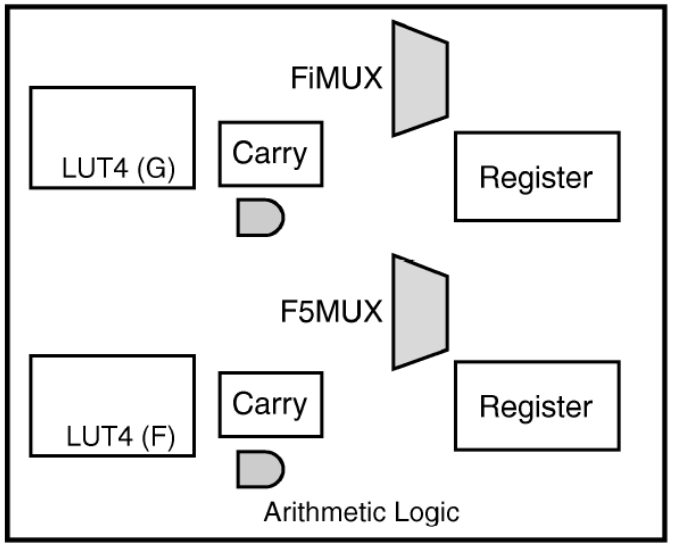


Vir: Wikipedia.

Spartan 3E – sestava rezine

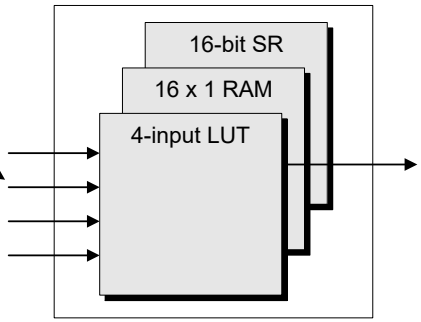


SLICEM



SLICEL

MLUT (multipurpose LUT)



Razlike med CPLD in FPGA

- CPLD temeljijo na PAL organizaciji
  - Enostavna AND/OR struktura – konfiguracija ostane tudi po izklopu napajanja
  - Majhne in predvidljive zakasnitve od priključka do priključka
  - Ko so programirani, lahko vsebino enostavno zaklenemo
  - CPLD arhitekture so si med sabo zelo podobne (enostaven razvoj)
  - Omejena kompleksnost (do 1024 flip-flopov) → uporabo CPLD za tvorbo "glue logic" vezij – tisto, kar smo včasih realizirali z diskretno logiko.
  - Statična poraba npr. CoolRunner CPLD (<50 µA).
- FPGA so bolj kompleksna vezja (do 150,000 flip-flopov Spartan 6)
  - Poraba je odvisna od konfiguracije. Tipična poraba Artix 7 brez kristalnega oscilatorja znaša 0.75W.
  - Konfiguracijo vezja nalagamo vedno po vključitvi napajanja
  - Konfigurabilnost predstavlja problem za varnost
  - Konfigurabilnost predstavlja možnost za dinamične spreminanje povezav vezja MED delovanjem.
  - FPGA vsebujejo več fleksibilnosti in bolj kompleksne enote kot CPLD (vezja za upravljanje s signalom ure, množilniki, vgrajeni mikroprocesorji ...)

CPLD uporabljamo za enostavne, majhne aplikacije, kjer rabimo takojšnje delovanje po vklopu in nizko porabo moči ter varnost. FPGA sicer.

FPGA vezja – stanje danes

Lastnost/Vezje	Virtex-6	Virtex-5	Spartan-6	Extended Spartan-3A
Logične celice	<760 000	<330 000	< 150 000	< 53 000
I/O priključkov	<1200	<1200	< 576	< 519
Blok RAM	<38 Mbits	<18 Mbits	<4.8 Mbits	<1.8 Mbits
Množilniki	(25 x 18 MAC)	(25 x 18 MAC)	(18 x 18 MAC)	(18 x 18 MAC)
Multi-Gigabit High Speed Serial	6.5 Gbps, >11 Gbps	3.75 Gbps,	6.5 Gbps	3.125 Gbps
MicroBlaze™ Soft Processor	Da	Da	Da	Da

Načrtovanje digitalnih vezij

Uvod v logična vezja:

Sinteza kombinacijske funkcije v VHDL

Imena/oznake (ang. identifiers)

- VHDL ne razlikuje velikih/malih črk za oznake.
- Oznake so lahko iz črk, številk in podčrtaja (_).
- Začnejo se s črko, ne s številko, ne s podčrtajem.

Vgrajeni tipi podatkov v VHDL

- Vrste tipov podatkov v VHDL:
  - Skalarni (ang. Scalar):
    - Numerični (ang. numeric),
    - Naštevni (ang. enumeration),
    - Fizikalni (ang. physical)
  - Sestavljeni (ang. Composite)
  - Datotečni (ang. File)
  - Kazalčni (ang. Access (Pointer))

Cela števila (ang. Integer)

- Cela števila v VHDL so vsa pozitivna in negativna števila v 32 bitnem obsegu
- Osnovni celoštevilski tip:
  TYPE integer IS RANGE –2147483648 TO 2147483647;

- Znotraj celoštevilskega tipa lahko naredimo svoj tip
  TYPE type_name IS RANGE int_range_constraint;

- Indeksiranje obsega: naraščajoče - TO oz. padajoče - DOWNTO
  - `TYPE dan IS RANGE 1 TO 31;`
  - `TYPE mesec IS RANGE 1 TO 12;`
  - `TYPE verjetnost IS RANGE 0 DOWNTO 1;`
    `--WARNING:HDLCompiler:746 - Range is empty (null range)`
  - `TYPE voltage IS RANGE -12 DOWNTO 12;`
  - `SIGNAL V1:voltage RANGE 5 DOWNTO 0; -- deklaracija signala V1, ki je tipa voltage, na intervalu [5..0]`
  - `SIGNAL Vout : voltage; -- izhod Vout v polnem obmocju [12..-12]`

Cela števila - podtipi

VHDL pozna tudi naravna števila in pozitivna števila kot podtipa celih števil:

- Obseg naravnih števil je od 0 do 2147483647;
- Obseg pozitivnih števil je od 1 do 2147483647;

- V IEEE1076 sta podtipa definirana takole:

```
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
```

Zg. meja intervala celih števil `integer'high` = 2147483647

Realna števila (ang. real)

- Realna števila so pozitivna in negativna realna števila, ki jih zapišemo v IEEE754 zapisu s plavajočo vejico enojne natančnosti (ang. single precision floating point)
- TYPE type_name IS RANGE range_constraint;
- Obseg:

```
TYPE real IS RANGE -1.79769E308 TO 1.79769E308;
```

Naštevni tipi (ang. enumeration)

- Vsebuje seznam oznak, ki jih lahko zavzame naštevni tip
- Deklaracija:
  TYPE type_name IS (enumeration_ident_list);

- Vgrajeni naštevni tipi:
  - `TYPE bit IS (`0', '1');`
  - `TYPE boolean IS (false, true);`
  - `TYPE severity_level IS (note, warning, error, failure);`
  - `TYPE character IS (`a','b','c',...);`
- Primeri naštevnih tipov:
  - `TYPE tri_nivojska_logika IS (`0','1','Z');`
  - `TYPE op_code IS (ADD, ADC, SBC, SBI, SEI, MUL, DIV );`
  - `TYPE fsm_state IS (init, s0, s1, s2);`
- Preostala tipa navajamo zaradi popolnosti:
  tip `file_open_kind` in tip `file_open_status` za delo z datotekami.

Fizikalni tip (ang. physical)

- Opisuje enote fizikalnih veličin z osnovnimi enotami, njihovimi večkratniki v določenem obsegu.

- Edini fizikalni tip, ki je zaenkrat standardiziran v VHDL, je čas:

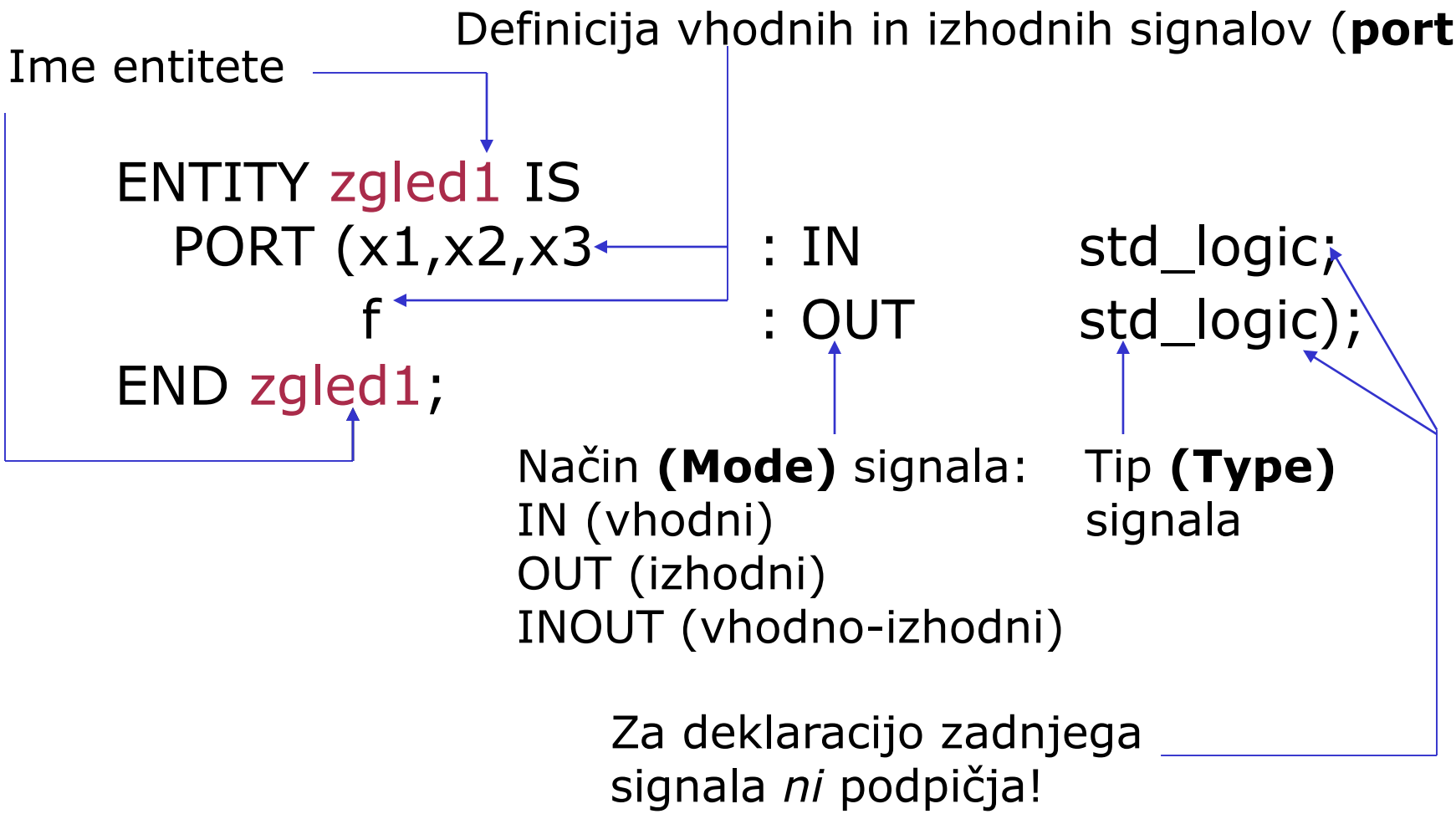
```
type Time is range --
  units
     fs;              -- femto sekunda
     ps  = 1000 fs;   -- piko sekunda
     ns  = 1000 ps;   -- nano sekunda
     us  = 1000 ns;   -- mikro sekunda
     ms  = 1000 us;   -- mili sekunda
     sec = 1000 ms;   -- sekunda
     min = 60  sec;   -- minuta
     hr  = 60  min;   -- urica
  end units;
```

Uvod v VHDL – std_logic_1164

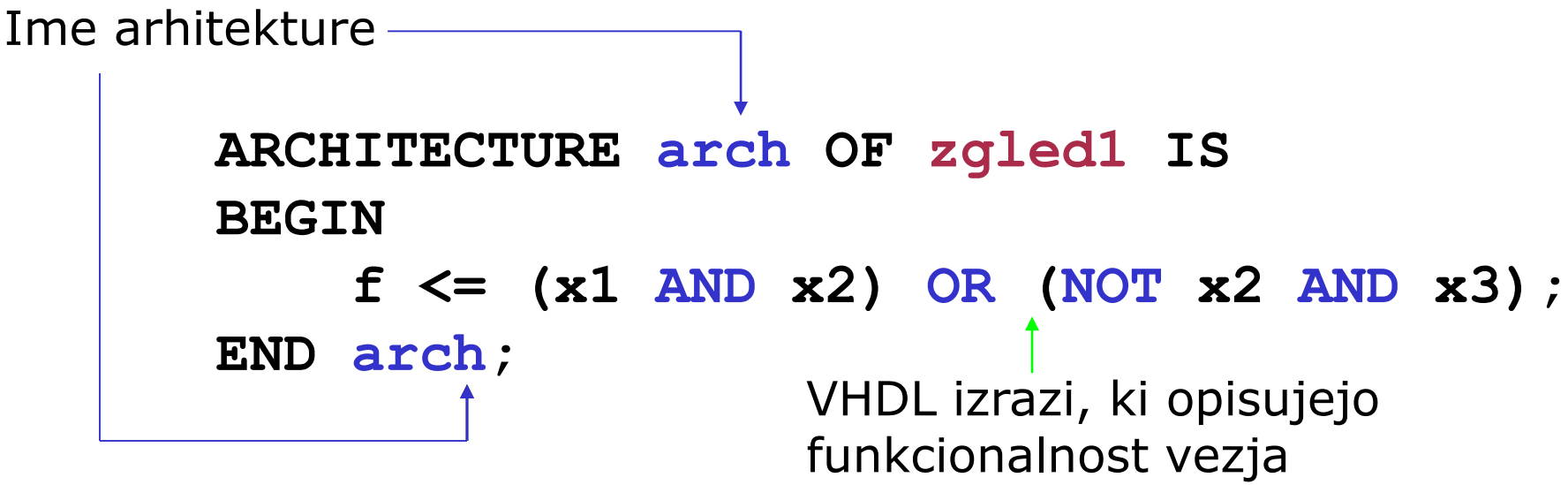
- Predstavitev logičnih signalov v knjižnici std_logic_1164
  - Logični signali tipa **std_logic** lahko zavzamejo **več** vrednosti:
    'U', 'X', '0', '1', 'Z', 'W', 'H', 'L', in '–'

Preostale vrednosti signala std_logic:
•'U': Neinicializirana vrednost. Signal še ni bil postavljen na nobeno vrednost. Vrednost simulacije, ne izvedbe (implementacije)!
•'X': Neznana vrednost:
V simulaciji vrednosti ni bilo mogoče določiti.
•'0': logična 0 (strong driver)
•'1': logična 1 (strong driver)
•'Z': Visoka impedanca (ang. High-Z)
•'W': Weak driver. V simulaciji ni mogoče določiti ali je 0 oz. 1.
•'L': Weak signal, predvidena vrednost naj bi bila v simulaciji 0
•'H': Weak signal, predvidena vrednost naj bi bila v simulaciji 1
•'-': Redundanca (ang. don't care).

Pisanje enostavne VHDL kode



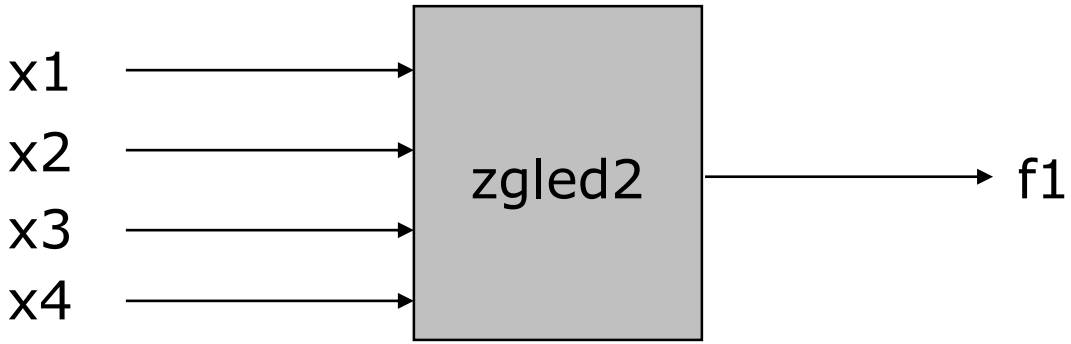
Pisanje enostavne VHDL kode



Pisanje enostavne VHDL kode

interni signal

```
ENTITY zgled2 IS
   PORT (x1, x2, x3       : IN              std_logic;
         x4               : INOUT           std_logic;
         f1               : OUT             std_logic );
END zgled2;
ARCHITECTURE arch OF zgled2 IS
SIGNAL x5 : std_logic := '0';
BEGIN
         x4 <= x1 AND x2;
         x5 <= NOT x2 AND x3;
         f1 <= x4 OR x5;
END arch;
```



Vmesne povezave v VHDL - BUFFER

- Priključkov tipa OUT, ne moremo uporabiti kot vhode (IN) v isto ali drugo entiteto.
- Če je priključek tipa OUT, se namreč nahaja samo na levi strani določitvenega izraza.
- Če je priključek tipa IN potem se nahaja samo na desni strani določitvenega izraza.
- Prva možnost izvedbe notranjih povezav, kjer je izhod obenem vhod v entiteto (povratna vezava):
  - Izhoda ne deklariramo kot OUT ampak kot:

    `BUFFER C <= A or (B and C);`
- Vsi priključki, na katere se veže priključek C morajo biti tipa **`BUFFER`**.

- Razlika med **`BUFFER`** in **`INOUT`** je, da slednji vsebuje tudi stanje ‘Z’, medtem ko prvi ne.

Povratne zanke v VHDL - SIGNAL

- Boljša možnost izvedbe internih povezav je uvedba **žice** (signal):

  Naj bo izhod C tipa **out**, z njim pa želimo izvesti:

  C <= A or (B and C).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity loopback is
        Port ( A, B : in STD_LOGIC;
               C : out STD_LOGIC);
end loopback;
architecture arch of loopback is
signal C_signal : STD_LOGIC;
begin
        C <= C_signal;
        C_signal <= A or ( B and C_signal );
end arch;
```

Boole-ovi operatorji v VHDL

- Nekaj logičnih operatorjev v VHDL:
  - **AND** logična konjunkcija
  - **OR** logična disjunkcija
  - **NOT** logična negacija
  - **NAND**, **NOR**, **XOR**, **XNOR**
- Operator prireditve (<=)
- VHDL NIMA prioritete logičnih operatorjev
- *Enostavna logična izjava (<=)*
  (ang. *simple assignment statement*)

Določitveni izrazi v VHDL

Izrazi za določanje logičnih vrednosti signalom – določitveni izrazi (ang. assignment statements):

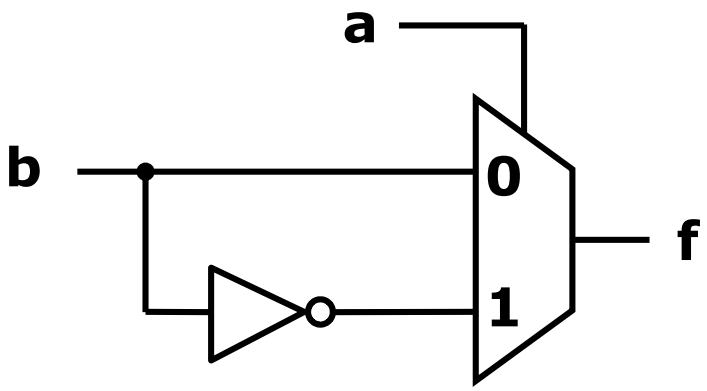
- Enostavni določitveni izrazi
- Izbirni izrazi:
  (when … when others in when … else)
- Pogojni izrazi:
  (if-elseif-else in case)

WHEN ELSE realizacija a⊕b

```
ENTITY exor2 IS
PORT ( a, b : IN STD_LOGIC;
          f : OUT STD_LOGIC
     );
END exor2;

ARCHITECTURE arch OF exor2 IS
BEGIN
    f <= b WHEN a = '0' ELSE (NOT b);
END arch;
```

a	*b*	*f*
0	0	0
0	1	1
1	0	1
1	1	0

Izbirni stavek tipa when-else
je realiziran z izbiralnikom 2/1



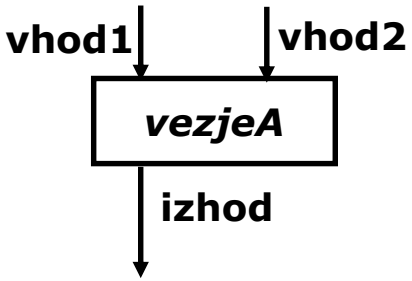
WHEN OTHERS realizacija a⊕b⊕c

```
ENTITY exor3 IS
PORT ( a,b,c : IN STD_LOGIC;
       f : OUT STD_LOGIC);
END exor3;
ARCHITECTURE arch OF exor3 IS
BEGIN
     WITH a SELECT
     f <= (b xor c) WHEN '0',
          (b xnor c) WHEN '1',
          'X' WHEN OTHERS;
END arch;
```

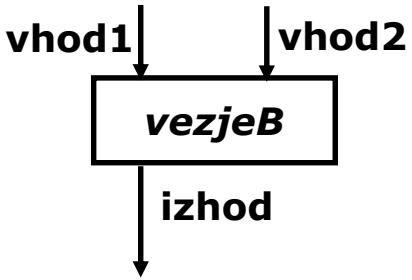
a	*b*	*c*	*f*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Povezovanje komponent v VHDL

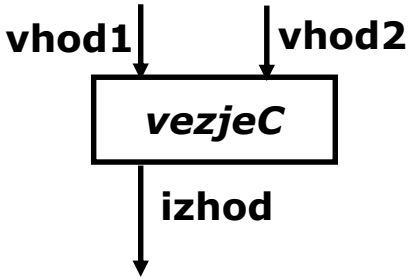
```
entity vezjeA is
Port ( vhod1, vhod2: in STD_LOGIC;
          izhod : out STD_LOGIC);
end vezjeA;
architecture arch of vezjeA is
begin
 izhod  <= vhod1 and vhod2;
end arch;
```

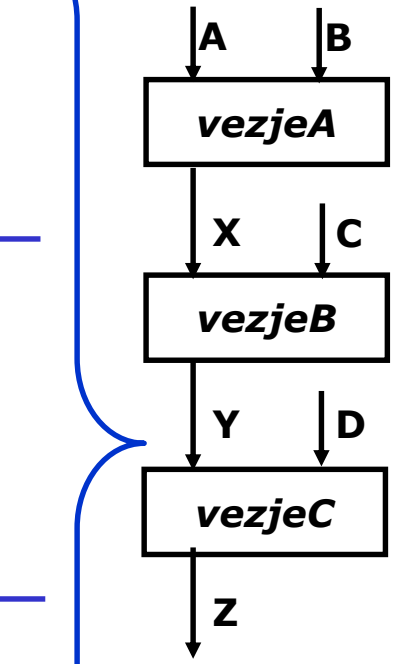



```
entity vezjeB is
Port (vhod1, vhod2: in STD_LOGIC;
        izhod : out STD_LOGIC);
end vezjeB;
architecture arch of vezjeB is
begin
 izhod <= vhod1 xor vhod2;
end arch;
```



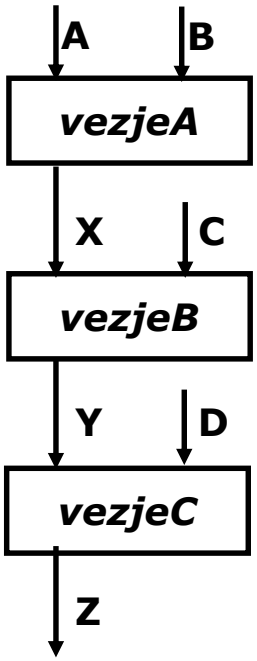

```
entity vezjeC is
Port (vhod1, vhod2: in STD_LOGIC;
        izhod : out STD_LOGIC);
end vezjeC;
architecture arch of vezjeC is
begin
 izhod <= vhod1 or vhod2;
end arch;
```

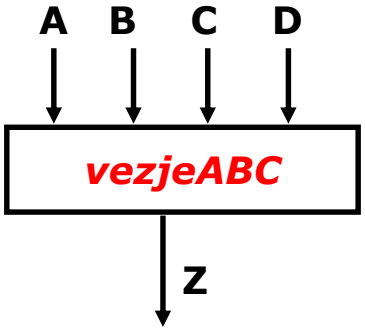




Povezovanje komponent v VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity vezjeABC is
Port ( A, B, C, D : in STD_LOGIC;
          Z : out STD_LOGIC);
end vezjeABC;
architecture arch of vezjeABC is
signal X, Y : STD_LOGIC;
component vezjeA is
Port ( vhod1, vhod2: in STD_LOGIC;
          izhod : out STD_LOGIC);
end component;
component vezjeB is
Port ( vhod1, vhod2: in STD_LOGIC;
          izhod : out STD_LOGIC);
end component;
component vezjeC is
Port ( vhod1, vhod2: in STD_LOGIC;
          izhod : out STD_LOGIC);
end component;
begin

 U1: vezjeA port map( A, B, X );          --položajno povezovanje (positional association)
 U2: vezjeB port map( C, X, Y );
 U3: vezjeC port map( vhod2 => Y,
                      vhod1 => D,
                      izhod => Z);        --imensko povezovanje - lahko zamenjamo vrstni red
end arch;
```





Povezovalni stavek (port map)

- **U1: vezjeA port map(A, B, X);**
  - Definira komponento vrste **vezjeA** z imenom **U1**
  - Uporablja *položajno povezovanje* (ang. *positional association)*
  - Vhodi in izhodi, navedeni v PORT MAP se pojavljajo v enakem zaporedju kot v stavku COMPONENT (Deklaracijo entitete kopiramo in zamenjamo ENTITIY ←→ COMPONENT)
  - **Pisanje je hitrejše, a se hitro lahko zmotimo in zamenjamo vrstni red položajev, zato tega načina v praksi ne uporabljajte!**
- **U3: vezjeC port map (vhod2 => Y, vhod1 => D, izhod => Z);**
  - Komponenta vrste **vezjeC** z imenom **U3**, ki uporablja *imensko povezovanje* (ang. *named association)*
  - Vhodi in izhodi, navedeni v PORT MAP so povezani z določenim poimenovanjem signala v stavku COMPONENT
- Če v seznamu povezav **izhodnega** priključka ne želimo povezati, potem v povezovalnem stavku uporabimo rezervirano besedo (**open**)

  U4p: vezjeD port map (a, b, open, f);
  U4n: vezjeD port map (a => a, b => b, izhod1 => open, izhod2 => f);

Procesni stavek in uvedba nadzora časa

- **Sočasni določitveni izrazi** (v logičnih enačbah) (*concurrent assignment statements)*
  - **Zaporedje** teh izrazov v VHDL kodi **ne vpliva** na pomen kode.
- **Sekvenčni določitveni izrazi** (ang. *sequential assignment statements*) nastopajo izključno v procesnih stavkih
  - Zaporedje teh izrazov v VHDL kodi *lahko vpliva* na pomen kode
  - Primera sekvenčnih določitvenih izrazov sta if-elseif-else in case.
- Kombinacijski stavki in procesni stavki se izvajajo vzporedno.

Realizacija a⊕b⊕c v procesnem stavku

```
ENTITY exor3 IS
PORT (  a, b, c : IN STD_LOGIC;
        f : OUT STD_LOGIC);
END exor3;
ARCHITECTURE arch OF exor3 IS
   BEGIN
   PROCESS ( a, b, c )
   BEGIN
      IF a = '0' THEN
        f <= b xor c;
      ELSE
        f <= b xnor c;
      END IF;
   END PROCESS;
   END arch;
```

a	*b*	*c*	*f*	
0	0	0	0	
0	0	1	1	xor
0	1	0	1	
0	1	1	0	
1	0	0	1	
1	0	1	0	xnor
1	1	0	0	
1	1	1	1	

seznam občutljivosti
(ang. sensitivity list)

Realizacija a⊕b⊕c v procesnem stavku

```
ENTITY exor3 IS
PORT (  a, b, c : IN STD_LOGIC;
        f : OUT STD_LOGIC);
END exor3;
ARCHITECTURE arch OF exor3 IS
   BEGIN
   PROCESS ( a, b, c )
      BEGIN
          CASE a IS
             WHEN '0'     => f <= b xor c;
             WHEN '1'     => f <= b xnor c;
             WHEN OTHERS  => f <= 'X';
          END CASE;
      END PROCESS;
   END arch;
```

Spremenljivke v procesnem stavku

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity xor_v is
Port ( A, B, C : in STD_LOGIC;
       X,Y : out STD_LOGIC);
end xor_v;
architecture arch of xor_v is
begin
       process (A, B, C)
       variable D : STD_LOGIC;
       begin
              D := A; -- D ← A (inicializacija)
              X <= C xor D; -- A se uporabi pri X=C⊕A
              D := B; -- D ← B
              Y <= C xor D; -- B se uporabi pri Y=C⊕B
       end process;
end arch;
```

Spremenljivka (VARIABLE) nima strojnega ekvivalenta! (služi lažjemu programiranju)

Vezana je na določen proces - tam, kjer je definirana. Če želimo isto spremenljivko uporabljati med več procesi jo deklariramo kot deljeno (`shared variable`)

Ovrednoti se **SPROTI**, zato jo lahko uporabimo večkrat v procesnem stavku (zanke ...).

V definiciji spremenljivke (stavek VARIABLE) ne inicializirajte začetne vrednosti! Zaradi preglednosti to storite na začetku procesnega stavka!

Signali v procesnem stavku

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity xor3_s is
Port ( A, B, C : in STD_LOGIC;
       X, Y : out STD_LOGIC);
end xor3_s;
architecture arch of xor3_s is
signal D : STD_LOGIC;
begin
       process (A,B,C)
       begin
              D <= A;  -- se ne upošteva
              X <= C xor D;
              D <= B;        -- velja zadnja prireditev!
              Y <= C xor D; -- Y=C⊕B ter X=C⊕B
       end process;
end arch;
```

WARNING:Xst:647 - Input <A> is never used. This port will be preserved and left unconnected if it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this sub-block is preserved.

Signal ima strojni ekvivalent! **ŽICA**.

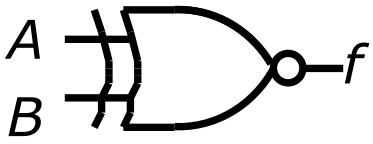
Signal je "globalen".

Ovrednoti se **NA KONCU**.

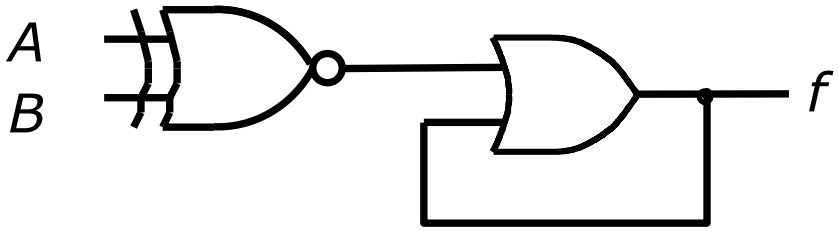
Če isti signal priredimo večkrat, velja zadnja prireditev!

Implicitni spomin v procesu

```
ARCHITECTURE arch OF c1 IS
BEGIN
PROCESS ( A, B )
BEGIN
   f <= '0';
   IF A = B THEN
      f <= '1';
    END IF ;
END PROCESS ;
END arch;
```




```
ARCHITECTURE arch OF c1 IS
BEGIN
PROCESS ( A, B )
BEGIN

   IF A = B THEN
       f <= '1';
   END IF ;
END PROCESS ;
END arch;
```

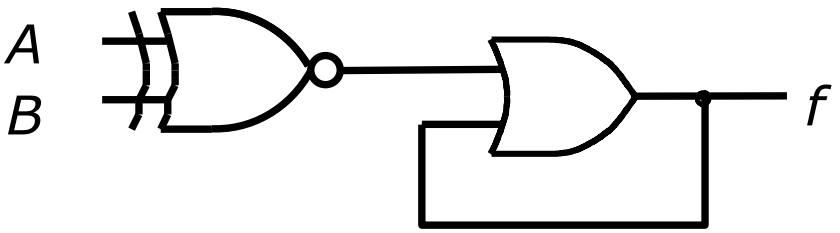


Izogibanje implicitnemu spominu v procesnih stavkih

- Stavek **IF** brez **ELSE** *ohranja staro stanje* spremenljivke (ang. latch), ki ji v **IF** stavku prirejamo vrednost!
- Ohranjanje stanja ➔ izhod vezja je odvisen:
  - od vrednosti vhodov in
  - od stare vrednosti izhoda
- Velja za vse pogojne stavke znotraj procesnega stavka (**IF**, **CASE**)

```
ARCHITECTURE arch OF c1 IS
BEGIN
PROCESS ( A, B )
BEGIN
   IF A = B THEN
        f <= '1';
   END IF ;
END PROCESS;
END arch;
```

```
WARNING:Xst :737 - Found 1-
  bit latch for signal
  <f>.
```



Kako deluje simulator v VHDL?

- Ko sta struktura (entity) in obnašanje vezja (behavior) definirana, moramo obnašanje vezja časovno simulirati.
- Simulator je orodje, ki spremlja obnašanje vezja (vhodi, izhodi) v diskretnih časovnih korakih.
- Naloga programerja je, da zapiše zaporedje vzorcev vzbujanja (ang. stimulation) vhodnih priključkov testiranega vezja (ang. UUT- unit under test), simulator določi izhodno vrednost z ovrednotenjem obnašanja vezja in čez čas vrne vrednost izhoda vezja.
- Temu rečemo transakcija nad opazovanim signalom.
- Če se novo izračunana vrednost razlikuje od stare, se sproži dogodek (ang. event).

Časovni potek simulacije

Simulacija se vedno začne z vzpostavitvijo (ang. initialization phase), nakar sledijo dvostopenjski cikli simulacije.

Ko se proženje novih dogodkov popolnoma zaključi, se simulacija ustavi.

Cilj simulacije je torej zbiranje časovnih dogodkov v enoti, ki jo testiramo.

Vzpostavitev simulacije

- Simulacija se vedno začne z vzpostavitvijo (ang. initialization phase), kjer se vhodnim signalom dodelijo njihove začetne vrednosti, čas simulacije se postavi na 0.
- V tej fazi se enkrat izvrši ovrednotenje obnašanja UUT, da tudi izhodi dobijo svoje začetne vrednosti.
- Internemu signalu ob začetku *simulacije* določimo vrednost:

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

Cikel simulacije

- Cikel simulacije, ki se ponavlja je sestavljen iz:
  - Prve stopnje, kjer se čas simulacije spremeni na trenutek naslednje transakcije. Ob tem se izvršijo vse predvidene transakcije s signali in te lahko povzročijo nadaljnje dogodke.
  - Druge stopnje, kjer se ovrednoti obnašanje modulov, ki so bili proženi z dogodki prve stopnje. Obnašanje teh modulov proži nove dogodke.
  - Ko se ovrednotijo dogodki druge stopnje, se simulacijski cikel zaključi in simulator preide na nov simulacijski cikel.
- Ko se proženje novih dogodkov popolnoma zaključi, se simulacija ustavi.
- Cilj simulacije je torej zbiranje časovnih dogodkov v sistemu in njihov prikaz v simulatorju (ang. monitor).

Modeliranje časovnih zakasnitev vezja - after

- Iz poteka cikla simulacije sledi, da neskončno hitre strukture (z zakasnitvijo 0) ne morejo obstajati.
- Zakaj ne?
- Zakasnitve moramo torej nekako predstaviti, oz. jih privzeti če niso posebej določene.
- Poleg tega, da so določene za uporabljen čip, lahko nastavljamo tudi stopnjo hitrosti (ang. speed grade) simulacije
- Za ta namen v enostavnem določitvenem izrazu uporabimo rezervirano besedo **`after`**.

```
ARCHITECTURE arch OF zgled2 IS
signal na, nb, nna : std_logic :=
   '0';
BEGIN
   na <= NOT a AFTER 10 ns;
   nna <= NOT na AFTER 10 ns;
   fa <= na AFTER 10 ns;
   nb <= NOT b AFTER 10 ns;
   fb <= nb AFTER 10 ns;
   f <= nna AND nb AFTER 10 ns;
END arch;
```

Uporaba procesnega stavka za nadzor nad časom – wait for

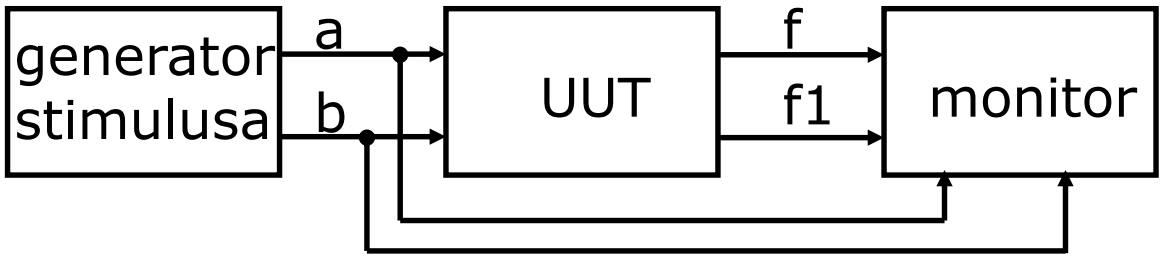
- Poleg **after** lahko zakasnitve v simulatorju modeliramo tudi s stavkom **wait for**, ki zadrži izvajanje procesa simulacije za določeno časovno enoto (npr. 200 ns).
- Če so v procesnem stavku stavki **wait for**, potem procesni stavek **nima** seznama občutljivosti.
- Če uporabimo samo **wait** - brez časovne enote, potem se izvajanje procesa zadrži do konca simulacije (s tem dosežemo enkratno izvajanje procesa).

```
process
    begin
        CLOCK <= '0';
        wait for 200 ns;
        CLOCK <= '1';
        wait for 200 ns;
end process;
```

Procesni stavek zgoraj ustvari pravokotni signal z imenom CLOCK med nivojema ‘0’ in ‘1’, periode 400 ns, razmerja signal/pavza 50%, saj ne vsebuje stavka wait brez časovne enote.

Datoteka testnih vrednosti

- Datoteka testnih vrednosti (ang. testbench) bo vedno nadrejena enoti, ki jo testiramo (ang. unit-under-test oz. UUT).
- Sestavljena je iz:
  - Generatorja vzbujanja oz. stimulusa → wait for
  - Povezave UUT in generatorja → port map
  - Opcijskega zapisa rezultatov simulacije (ang. monitor)



Datoteka testnih vrednosti kombinacijskih vezij

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY time_sim_tb IS
END time_sim_tb;

ARCHITECTURE a OF time_sim_tb IS

COMPONENT zgled2 IS
PORT (
a, b : IN STD_LOGIC;
fa, fb, f : OUT STD_LOGIC);
END COMPONENT;

signal a : std_logic := '0';
signal b : std_logic := '0';
signal fa, fb, f : std_logic;
```

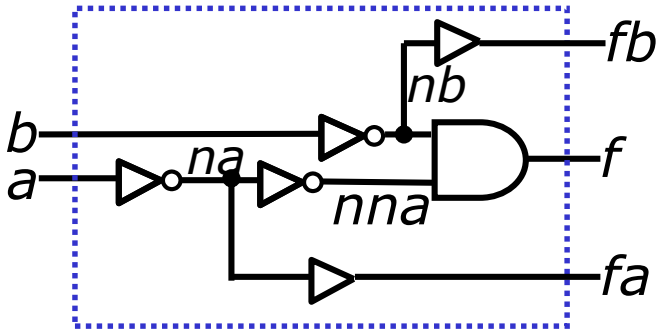
```
BEGIN
   uut: zgled2 PORT MAP (
         a => a,
         b => b,
         fa => fa,
         fb => fb,
         f => f
        );
   stim_proc: process
   begin
         a <= '1';
         wait for 200 ns;
         a <= '0';
         wait for 200 ns;
         b <= '1';
      wait;
   end process;
END;
```

Povezava UUT

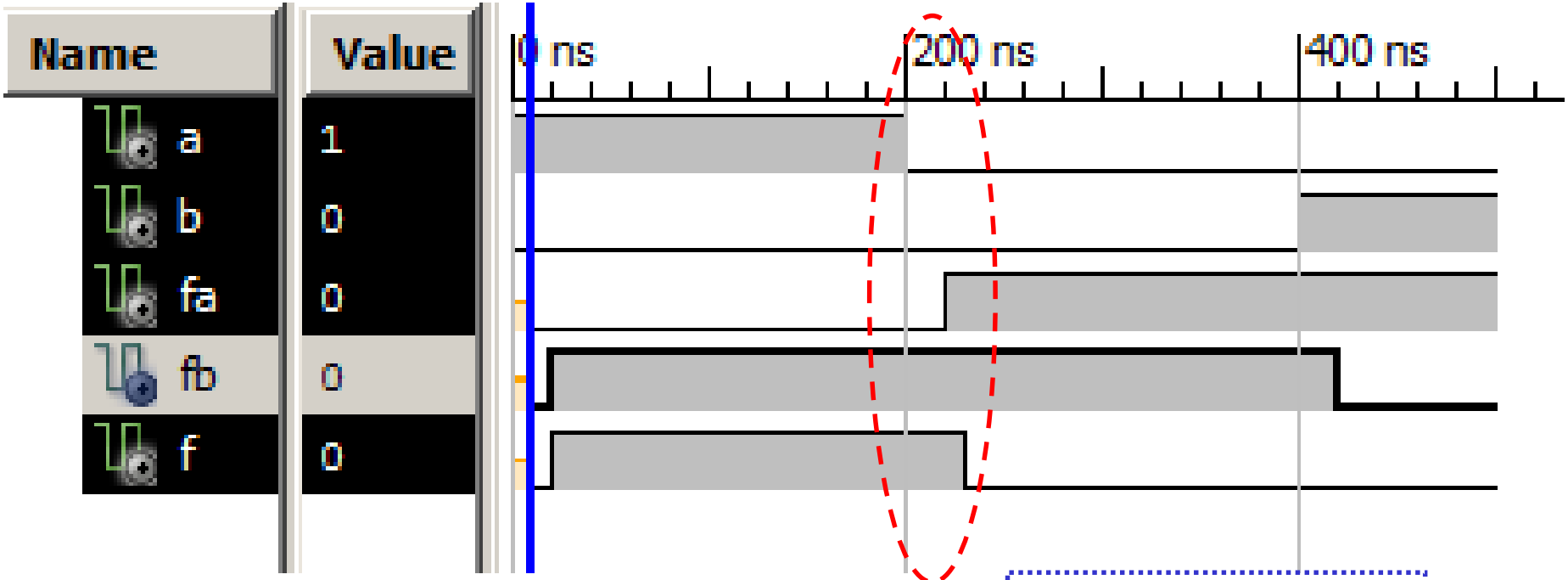
Generator stimulusa

Zakaj je modeliranje časovnih zakasnitev tako pomembno v VHDL?

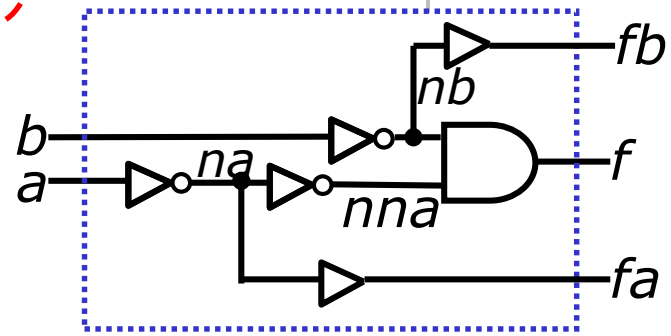
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY zgled2 IS
PORT (  a, b : IN STD_LOGIC;
        fa, fb, f : OUT
   STD_LOGIC);
END zgled2;
ARCHITECTURE arch OF zgled2 IS
signal na, nb, nna : std_logic :=
   '0';
BEGIN
   na <= NOT a AFTER 10 ns;
   nna <= NOT na AFTER 10 ns;
   fa <= na AFTER 10 ns;
   nb <= NOT b AFTER 10 ns;
   fb <= nb AFTER 10 ns;
   f <= nna AND nb AFTER 10 ns;
END arch;
```



Zakaj je modeliranje časovnih zakasnitev tako pomembno v VHDL?

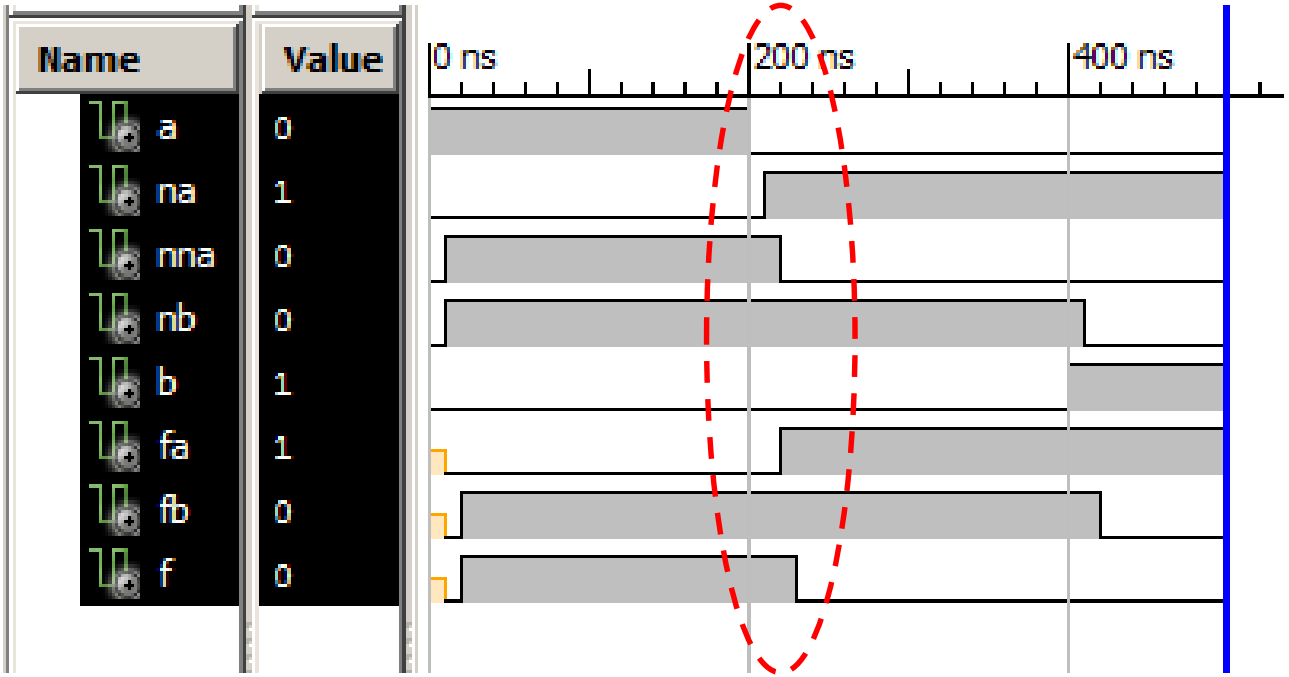


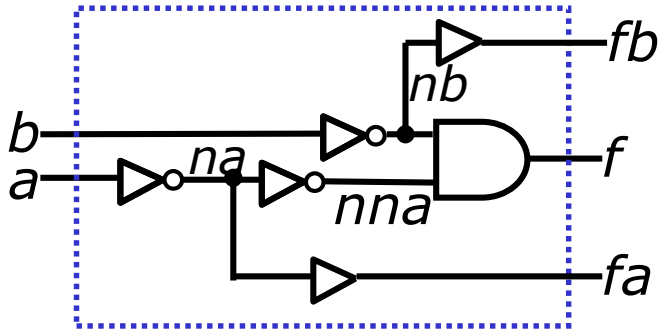

```
a <= '1';
wait for 200 ns;
a <= '0';
wait for 200 ns;
b <= '1';
```



Zakaj je modeliranje časovnih zakasnitev tako pomembno v VHDL?

```
a <= '1';
wait for 200 ns;
a <= '0';
wait for 200 ns;
b <= '1';
```

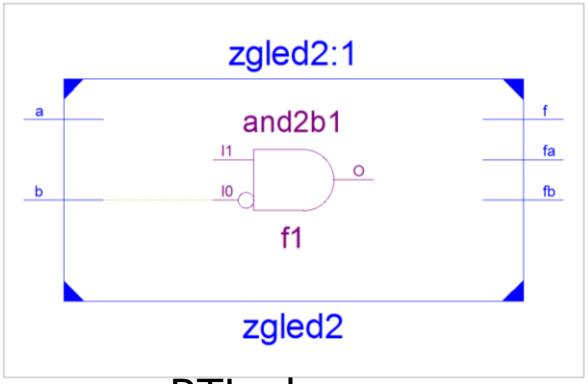




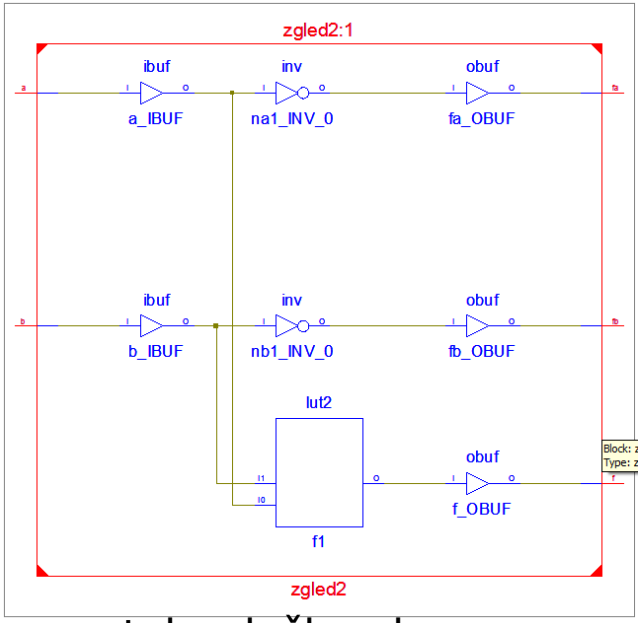
Sinteza in optimizacija HDL opisa

- Kako si ogledamo, kakšna bo pravzaprav izvedba v čipu?

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY zgled2 IS
PORT ( a, b : IN STD_LOGIC;
   fa, fb, f : OUT STD_LOGIC);
END zgled2;
ARCHITECTURE arch OF zgled2 IS
signal na, nb, nna : std_logic :=
   '0';
BEGIN
   na <= NOT a AFTER 10 ns;
   nna <= NOT na AFTER 10 ns;
   fa <= na AFTER 10 ns;
   nb <= NOT b AFTER 10 ns;
   fb <= nb AFTER 10 ns;
   f <= nna AND nb AFTER 10 ns;
END arch;
```



RTL shema



tehnološka shema

Implementacija na čipu (Implementation)

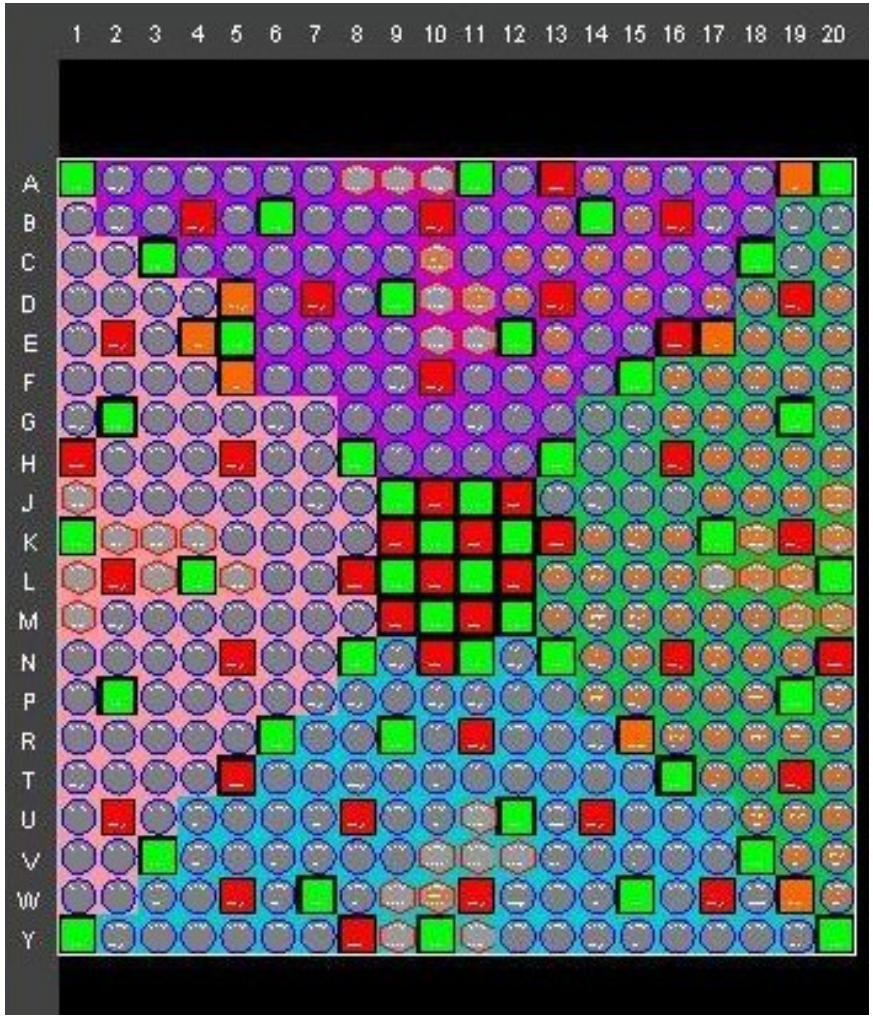
- Za izvedbo predstavljenega zgleda na čipu moramo določiti še vhodne in izhodne priključke vezja, kar storimo s pisanjem datoteke uporabniških omejitev (ang. user constraint file) – UCF datoteke.

Povezavo logičnega priključka (vhoda/izhoda) s fizičnim priključkom določamo kot "omejitev":

NET "x1" LOC ="H6";

s čimer povemo, da smo povezavo x1 priključili na H6.

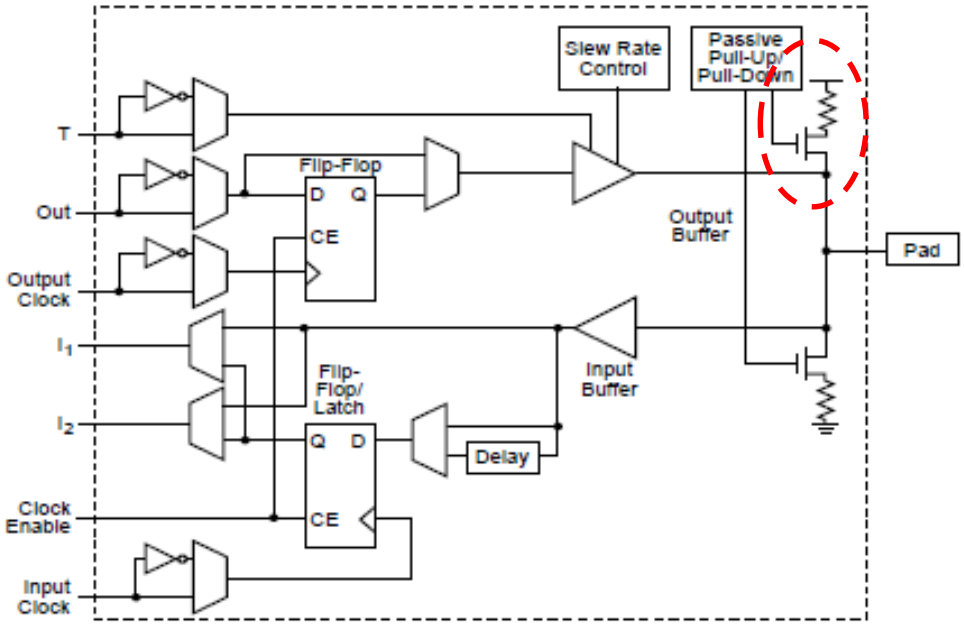
```
# vhodi na stikalih
#NET "x1" LOC = "U9"| IOSTANDARD = "LVCMOS33";
#NET "x2" LOC = "U8"| IOSTANDARD = "LVCMOS33";
#NET "x3" LOC = "R7"| IOSTANDARD = "LVCMOS33";
# izhod na LED
#NET "f" LOC = "T8"| IOSTANDARD = "LVCMOS33";
```



Kaj lahko še nastavljamo v UCF datoteki?

```
#NET "PS2Clk" LOC = "F4"| PULLUP | IOSTANDARD =
     "LVCMOS33";
#NET "PS2Data" LOC = "B2" | PULLUP | IOSTANDARD =
     "LVCMOS33";
```

Z besedo PULLUP vključimo upor za zagotavljanje visokega logičnega nivoja (ang. pullup).



Poenostavljena shema vhodno-izhodnega bloka (IOB) vezja FPGA XC4000E.

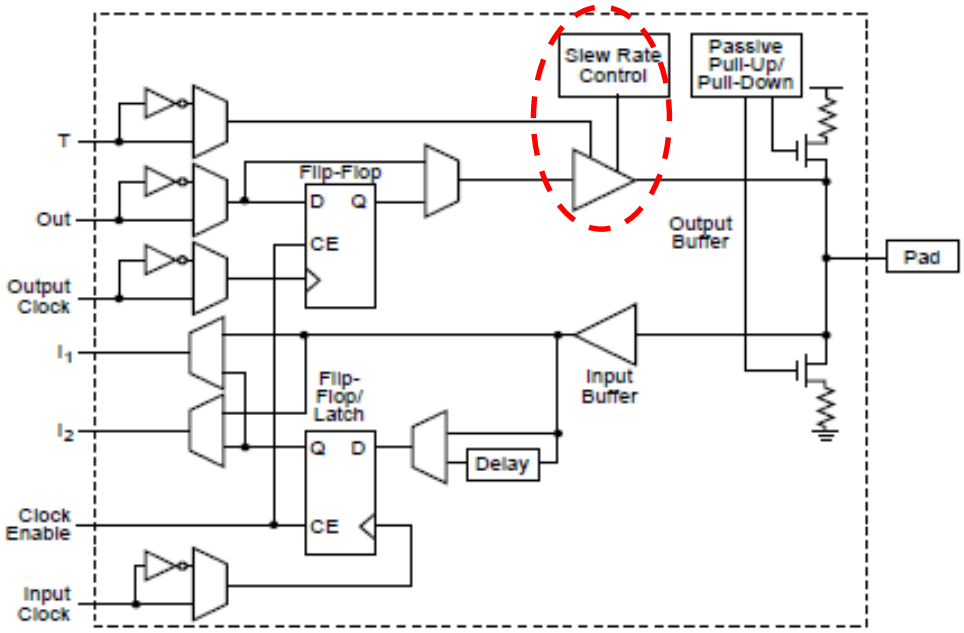
Vir: XC4000E and XC4000X Series Field Programmable Gate Arrays, Xilinx, 1999

Kaj lahko še nastavljamo v UCF datoteki?

```
NET "vsync"   LOC = "T10" | DRIVE=8 | SLEW=FAST;
NET "hsync"   LOC = "R9"  | DRIVE=8 | SLEW=FAST;
```

Z besedo DRIVE=x določamo velikost izhodnega toka priključka v mA za LVCMOS in LVTTL izhode. Večina I/O priključkov FPGA ima moč 12 mA.
Z besedo SLEW=FAST nastavljamo hitrost prehodov med logičnimi nivoji (ang. slew rate).

Več o parametrih UCF datotek najdete v:
"Constraints Guide", 2008 Xilinx



Poenostavljena shema vhodno-izhodnega bloka (IOB) vezja FPGA XC4000E.

Vir: XC4000E and XC4000X Series Field Programmable Gate Arrays, Xilinx, 1999

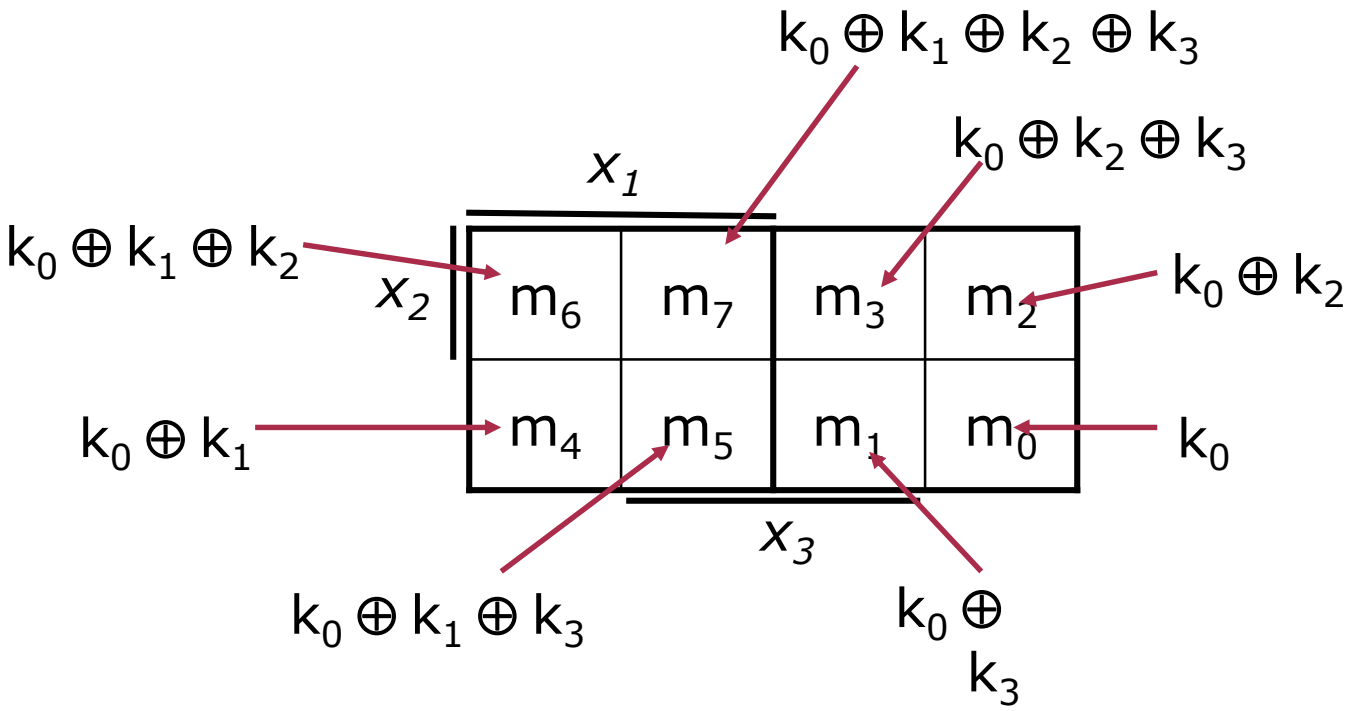
Enostaven zgled VHDL kode

- Napišite VHDL kodo vezja 3-bitne funkcije večine *(majority function)*
- Funkcija večine postavi izhod f='1' ko je večina od vhodov (x1, x2, x3) enaka '1'
- Načrtajte konstrukt entitete in arhitekturni konstrukt.
  - Konstrukt entitete imenujte ***Majority***,
  - Arhitekturni konstrukt imenujte ***MajorityFunc***

Linearnost funkcije

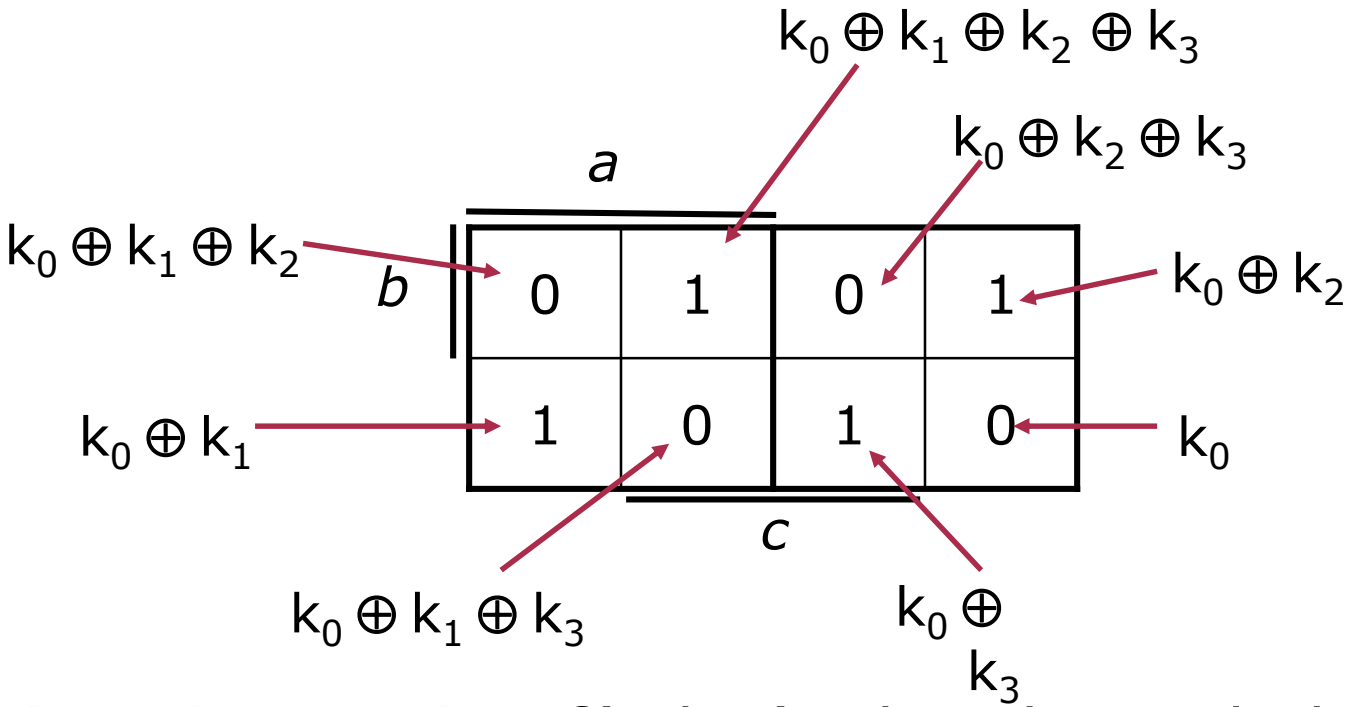
$f(x_1,x_2,x_3 \ldots x_n)=k_0 \oplus k_1x_1 \oplus k_2x_2 \oplus k_3x_3 \ldots \oplus k_nx_n$

Koeficienti $k_0, k_1 \ldots k_n$ so lahko 0 ali 1.



XOR treh spremenljivk

- Kakšna je PDNO oblika naslednjega izraza?
  $f(a, b, c) = a \oplus b \oplus c$



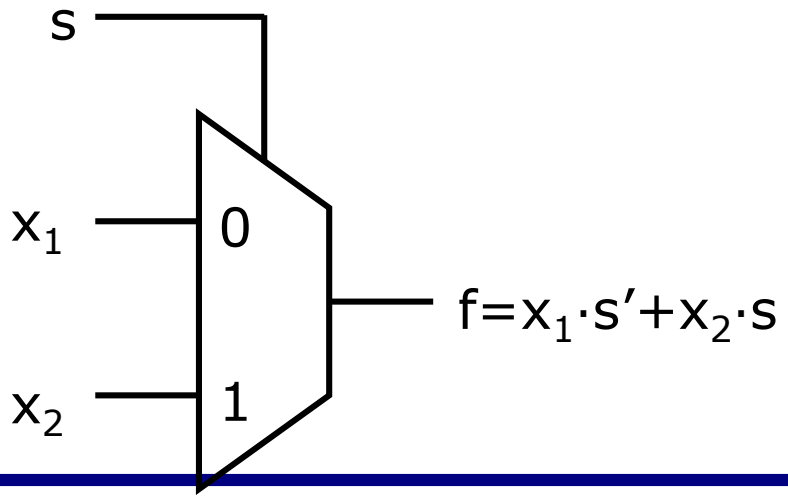
$f(a,b,c) = V(1,2,4,7)$ $f(a,b,c) = k_0 \oplus k_1\, a \oplus k_2\, b \oplus k_3\, c$

Načrtovanje digitalnih vezij

Gradniki kombinacijskih vezij:
Izbiralniki

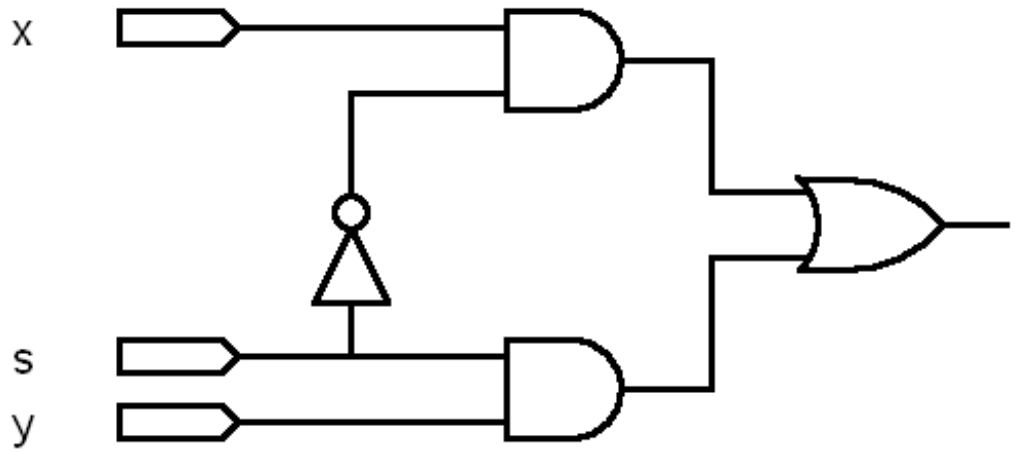
Izbiralniki

- Vezje izbiralnika oz. multiplekserja (MUX) ima
  - Nekaj podatkovnih vhodov (n)
  - Enega (ali več) izbirnih vhodov (2^n)
  - En izhod (f)
- Signal na enem od vhodov *(x_1, x_2)* pošlje na svoj izhod *(f)*.
- Kateri vhod je trenutno izbran določajo izbirni vhodi (ang. *select* ali *s*)

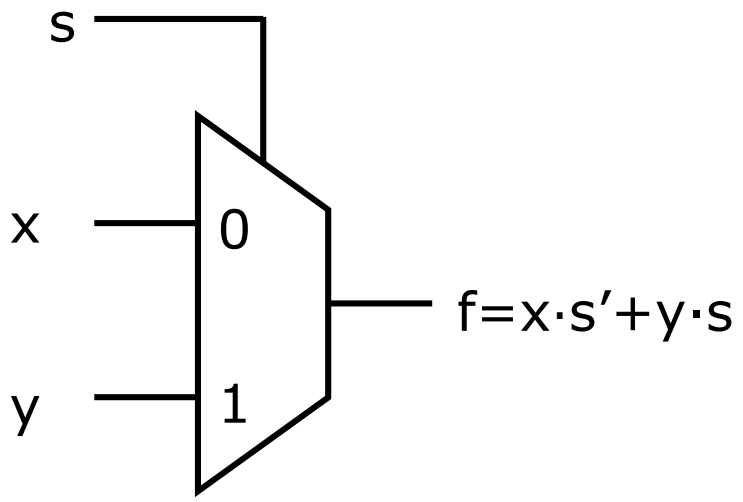


s	*f*
0	x_1
1	x_2

Izvedbe izbiralnikov

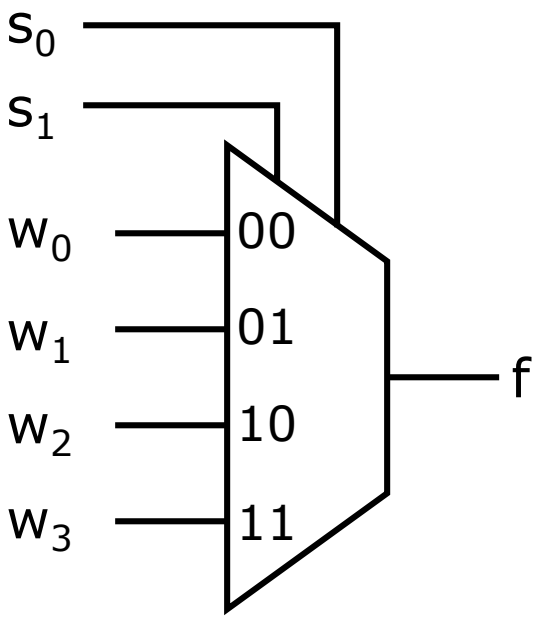


Osnovna izvedba



MUX 4/1

- 4-vhodni izbiralnik (4/1 MUX) na izhodu f 'izbira' enega od 4 podatkovnih vhodov (w_0, w_1, w_2, w_3). Kateri vhod je trenutno izbran določa stanje 2 izbirnih linij (s_1, s_2)

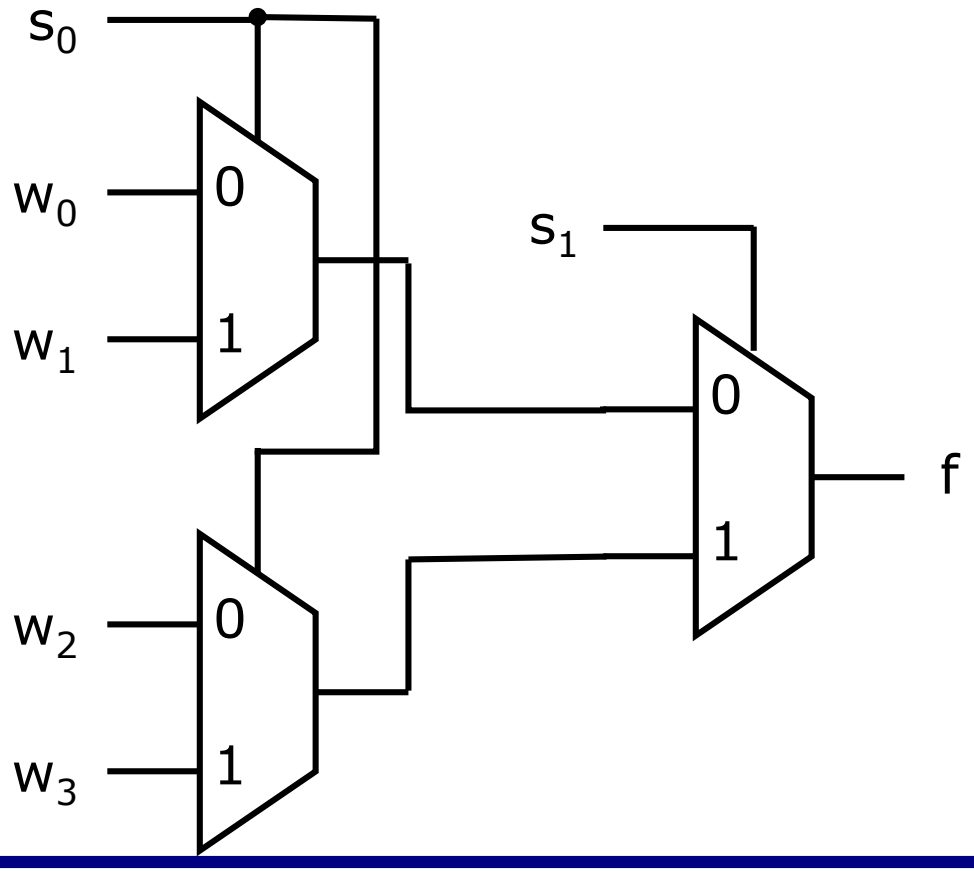


s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

$$f = s_1's_0'w_0 + s_1's_0w_1 + s_1s_0'w_2 + s_1s_0w_3$$

Načrtovanje 4/1 izbiralnika

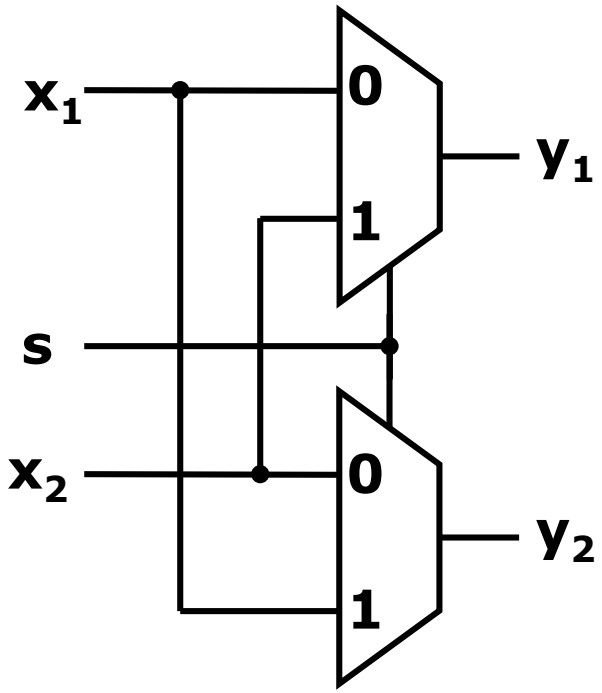
- Izdelamo ga s pomočjo treh MUX 2/1

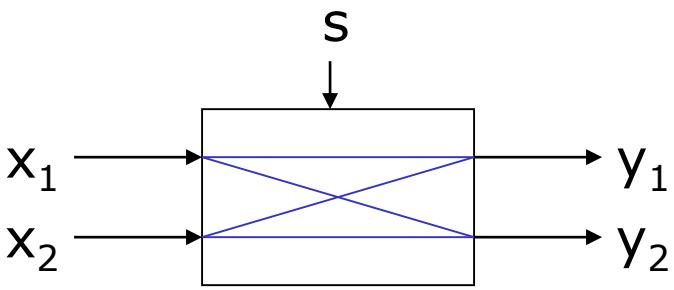


s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

Uporaba MUX: križno stikalo

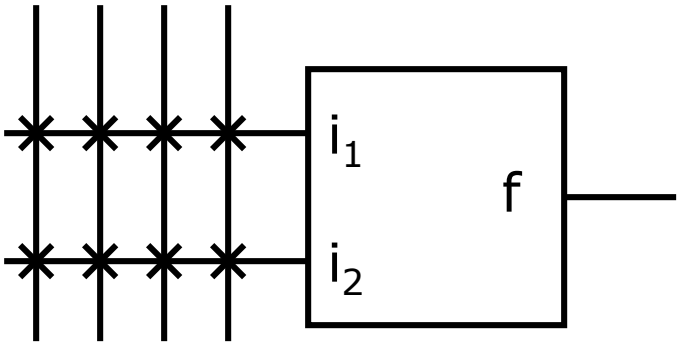
- Vezje z *n* vhodi in *k* izhodi
- Poveže katerikoli vhod na katerikoli izhod
- To je n x k križno stikalo (***n x k crossbar switch***)



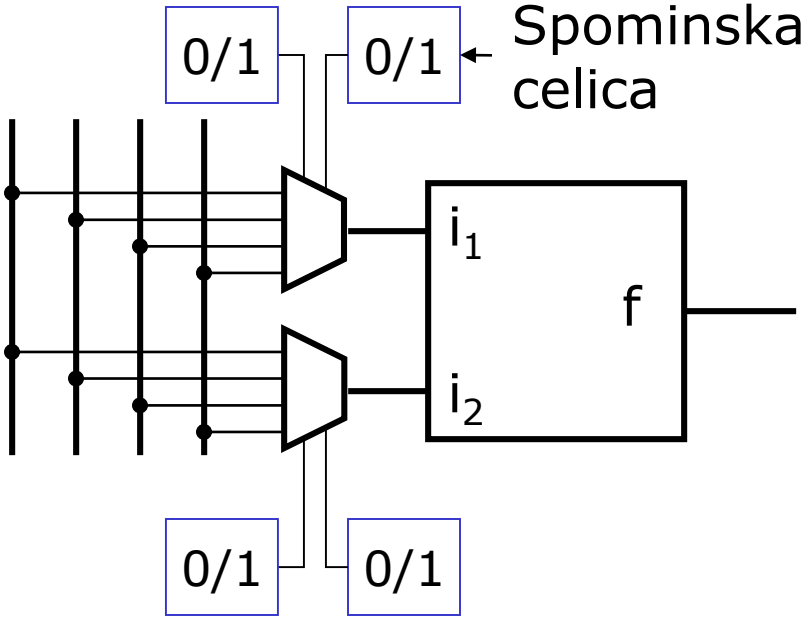


Uporaba MUX: Programabilno stikalo

- V programabilnih vezjih (PLD, CPLD in FPGA) programabilna stikala povezujejo povezave znotraj vezja
  - Tovrstna stikala so izvedena z izbiralniki

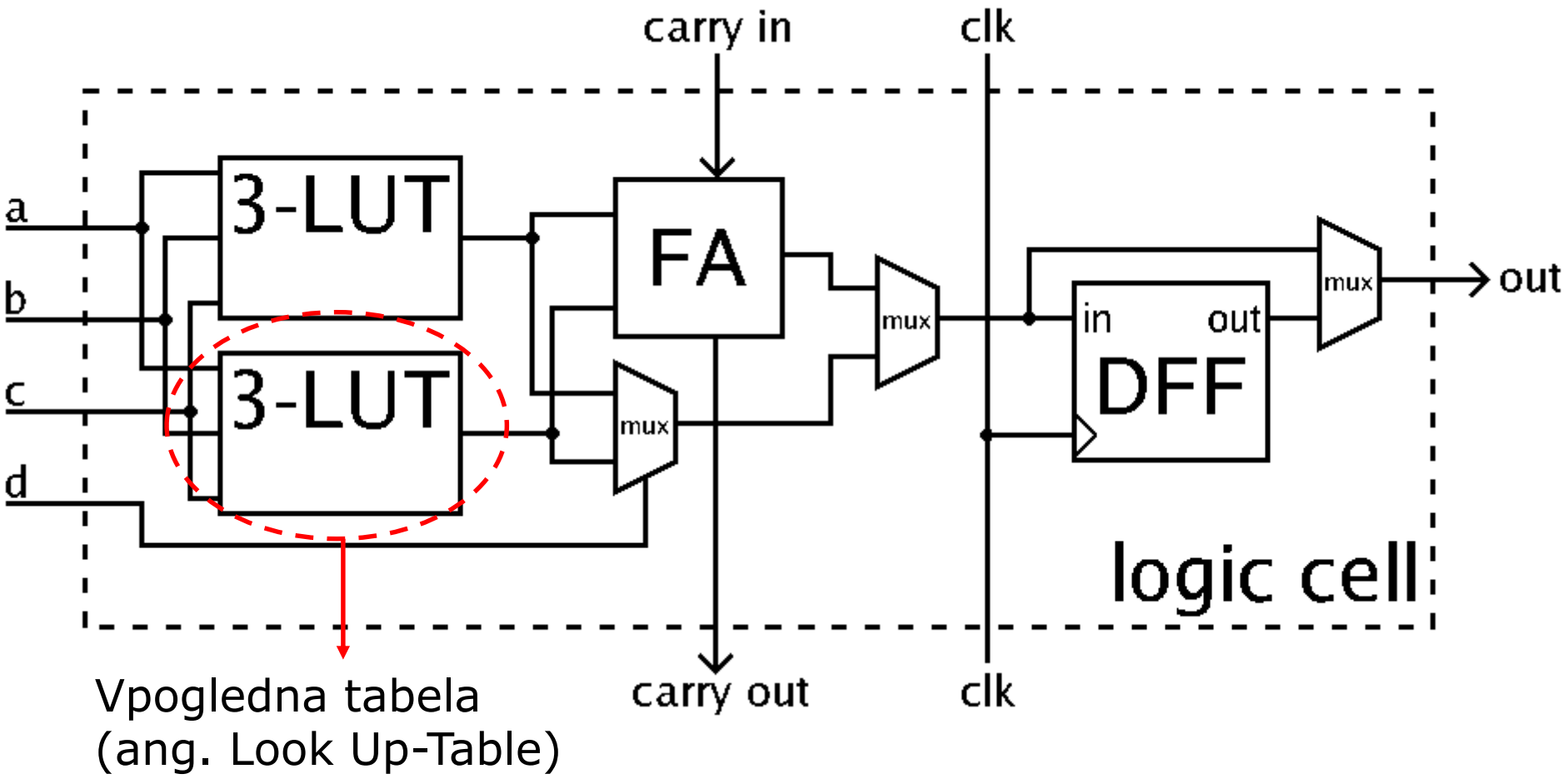


Logični blok vezja FPGA
s programabilnimi vhodi



Izvedba MUX

Poenostavljena celica CLB v FPGA

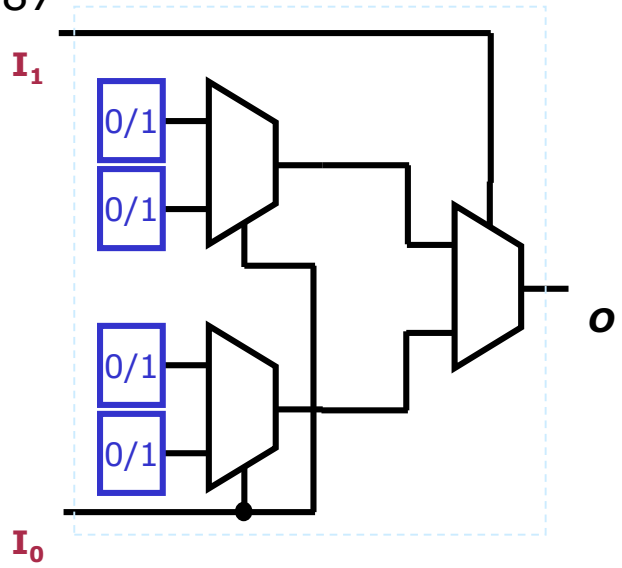


Vir: Wikipedia.

LUT2 v VHDL

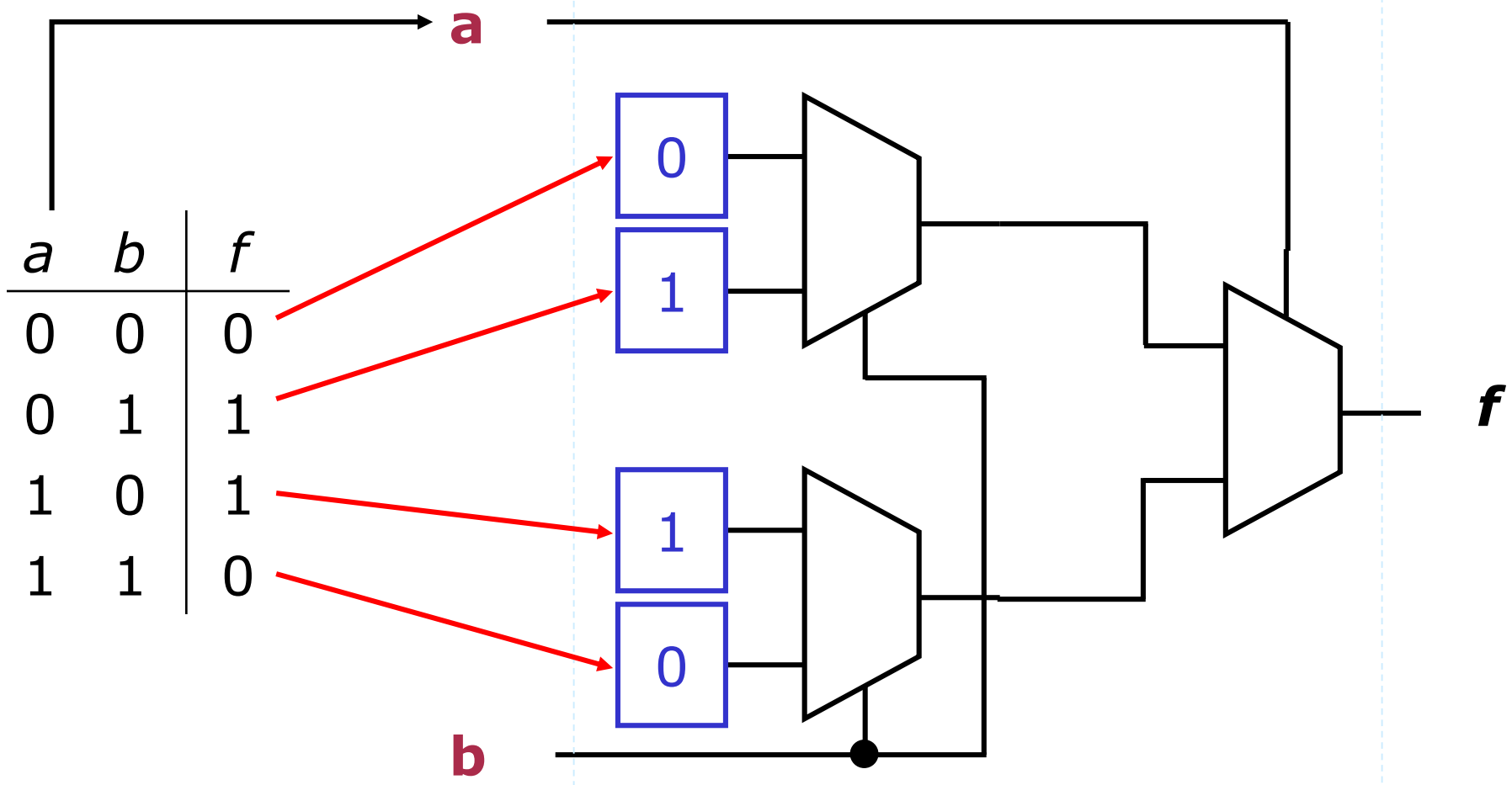
```
library ieee;
use ieee.std_logic_1164.all;
entity lut2_inst is
port( I0, I1 : in std_logic;
      O : out std_logic);
end lut2_inst;
architecture a of lut2_inst is
begin
inst : LUT2 generic map (INIT=>"1")
port map (I0=>I0, I1=>I1, O=>O);
end a;
```

XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices UG687



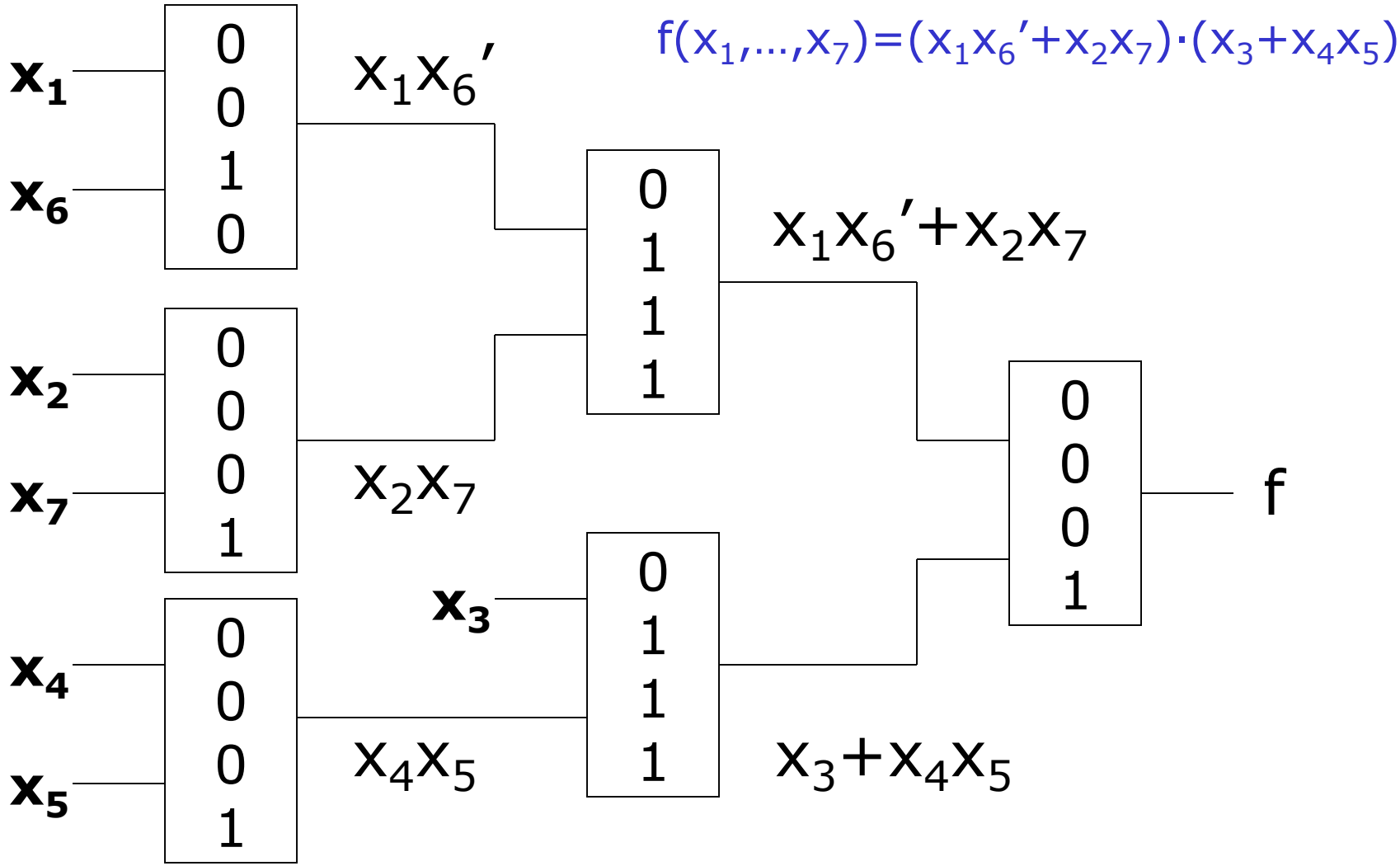
Programirana LUT (f=a'b+ab')

Najbolj pomembna spremenljivka (a) nadzira zadnji multiplekser

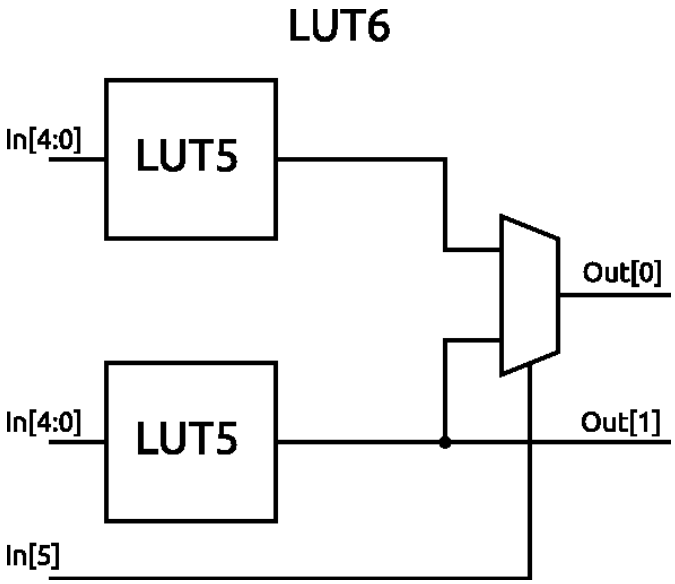


a	b	f
0	0	0
0	1	1
1	0	1
1	1	0

Faktorski zapis funkcije v FPGA z LUT2



LUT6 v vezju Spartan6



Vir: [Spartan-6 FPGA Configurable Logic Block]

Izvedba logičnih funkcij z MUX

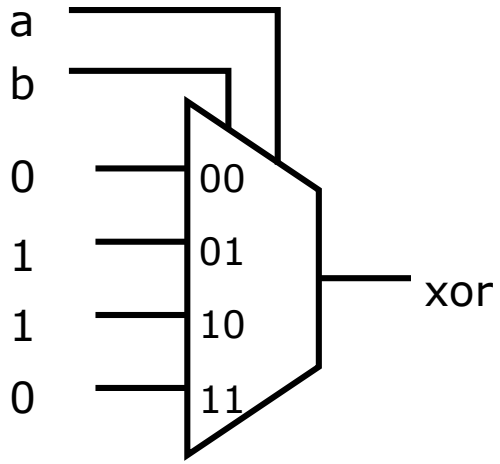
- MUX lahko uporabljamo pri sintezi (tvorbi) logičnih funkcij
  - Vpogledne tabele (LUT) so izvedene z MUX, s katerimi izbiramo (konstantno) vrednost iz vpogledne tabele.
- Vzemimo primer XOR in EQU (XNOR) funkcij

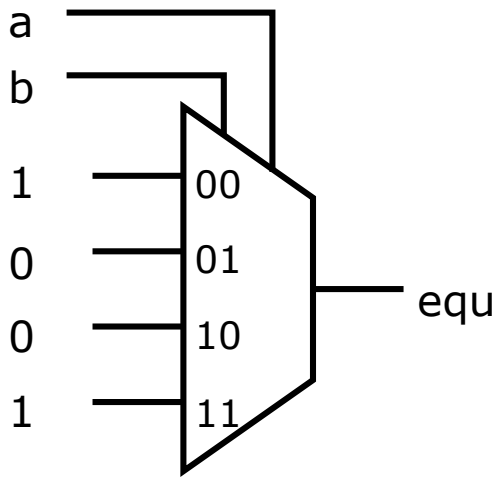
n	**naslovni vhodi**	**podatkovni vhodi**
r	*r* spremenljivk	0, 1 (t.i. trivialna realizacija)
r+1	*r* spremenljivk	funkcije preostale spremenljivke: $0, x_i, x'_i, 1$
r+2	*r* spremenljivk	funkcije preostalih dveh spremenljivk:
		$0, x_i, x'_i, x_j , x'_j , , ..., 1$

Izvedba logičnih funkcij z MUX

- MUX lahko uporabljamo pri sintezi (tvorbi) logičnih funkcij
  - Vpogledne tabele (LUT) so izvedene z izbiralniki, s katerimi izbiramo (konstantno) vrednost iz vpogledne tabele.
- Vzemimo primer XOR in EQU funkcij

a	*b*	*xor*	*equ*
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1



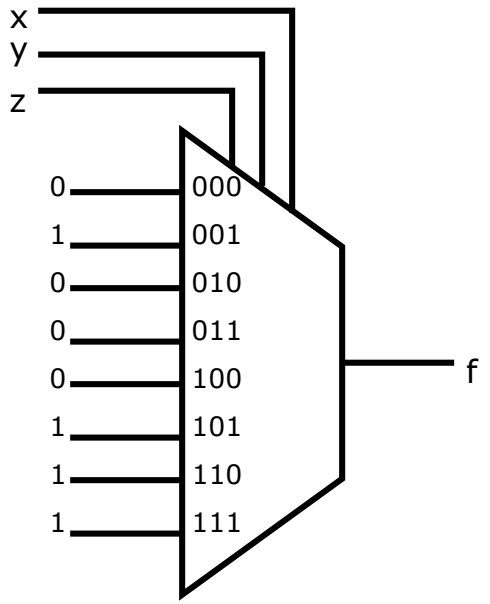


Trivialna realizacija (r=n)

Za funkcijo z n spremenljivkami vzamemo
r = n-naslovni multiplekser.

Primer: Realizacija funkcije
f(x, y, z)=V(1,5,6,7)

x	*y*	*z*	*f*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

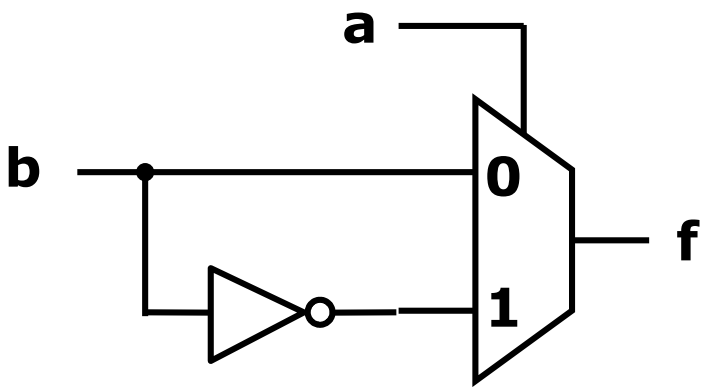


Izvedba z MUX (r=n-1) - primer

- Realizirajte funkcijo XOR dveh spremenljivk s pomočjo inverterjev in 2/1 izbiralnika.

a	*b*	*f*	
0	0	0	Ko je a=0, f=b
0	1	1	
1	0	1	Ko je a=1, f=b’
1	1	0	

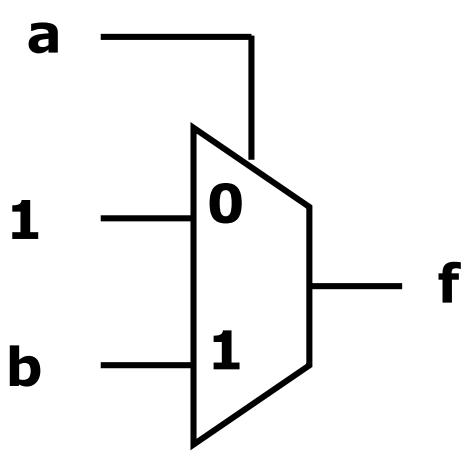
a	*f*
0	b
1	b’



Izvedba z MUX (r=n-1) - primer

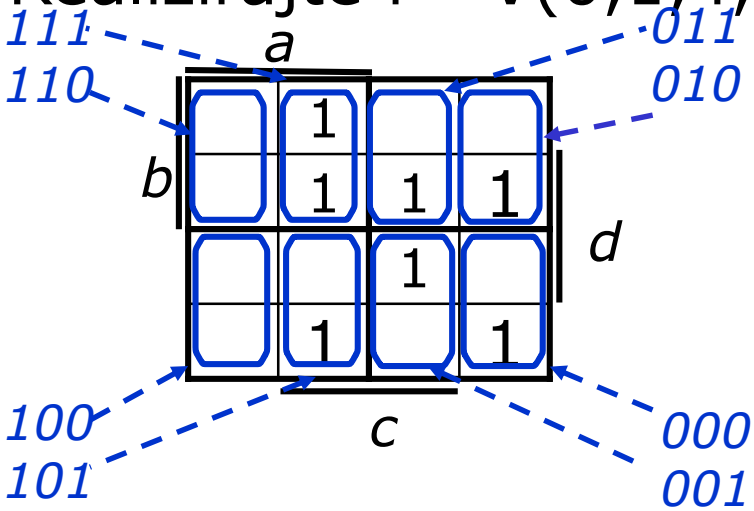
- Z uporabo 2/1 izbiralnikov in logičnih vrat realizirajte spodnjo funkcijo

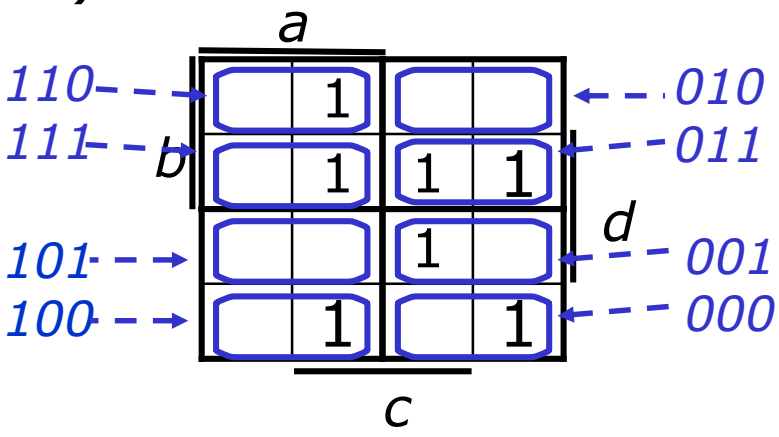
a	*b*	*f*
0	0	1
0	1	1
1	0	0
1	1	1

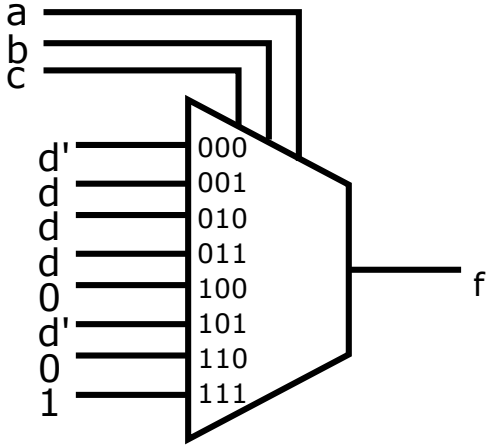


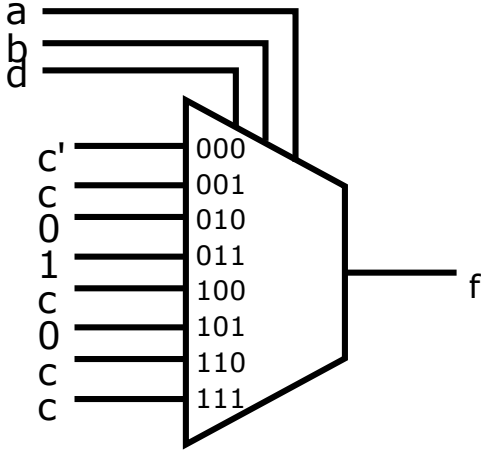
Izvedba funkcij z MUX (r=n-1)

Realizirajte $f^4=V(0,1,4,7,9,10,14)$.





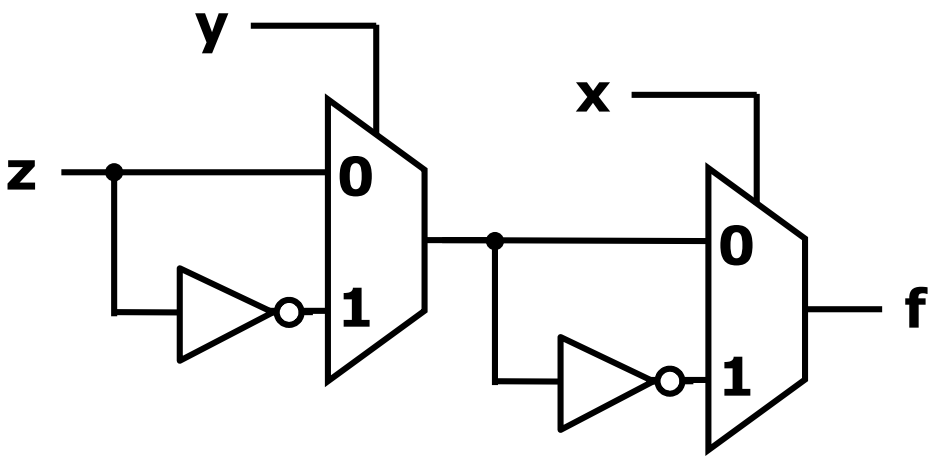




Izvedba funkcije z MUX (r=n-2) – primer

- Realizirajte 3-vhodna XOR vrata z uporabo 2/1 MUX

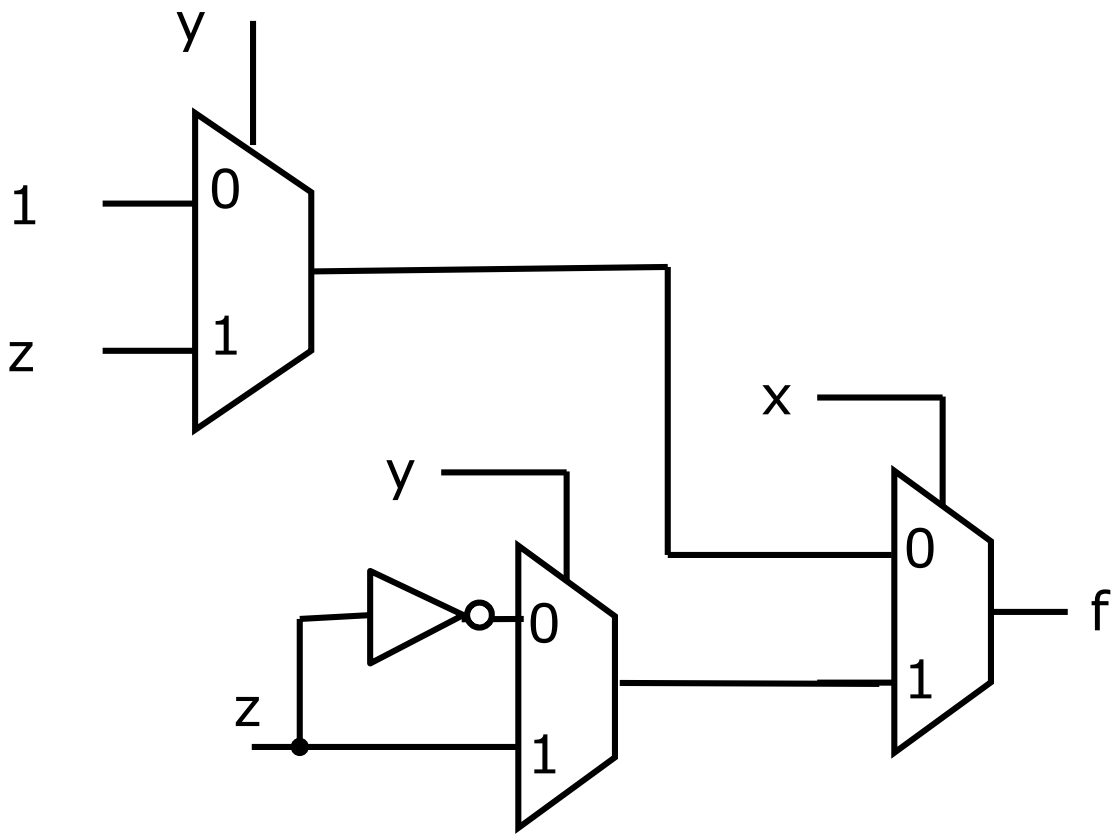
x	*y*	*z*	*f*	
0	0	0	0	y⊕z
0	0	1	1	
0	1	0	1	
0	1	1	0	
1	0	0	1	(y⊕z)’
1	0	1	0	
1	1	0	0	
1	1	1	1	



Izvedba funkcije z MUX (r=n-2)

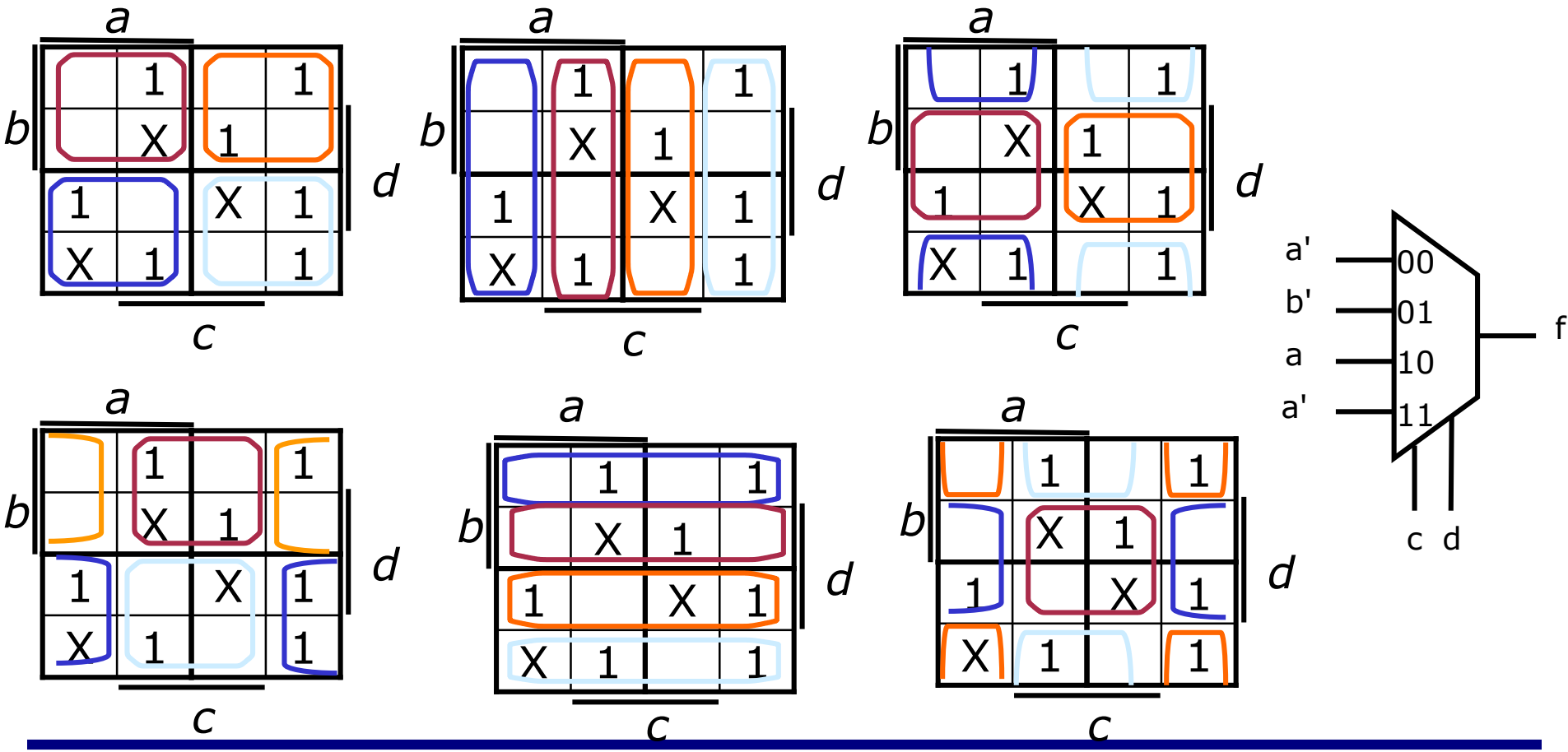
- Z uporabo 2/1 MUX in logičnih vrat realizirajte spodnjo funkcijo

x	y	z	f	
0	0	0	1	'1'
0	0	1	1	
0	1	0	0	z
0	1	1	1	
1	0	0	1	(y⊕z)'
1	0	1	0	
1	1	0	0	
1	1	1	1	



Izvedba funkcije z MUX (r=n-2) – primer

Realizirajte $f^4=V(0,1,4,7,9,10,14)$ in $V_x(3,8,15)$ z enim 4/1 izbiralnikom in inverterji.



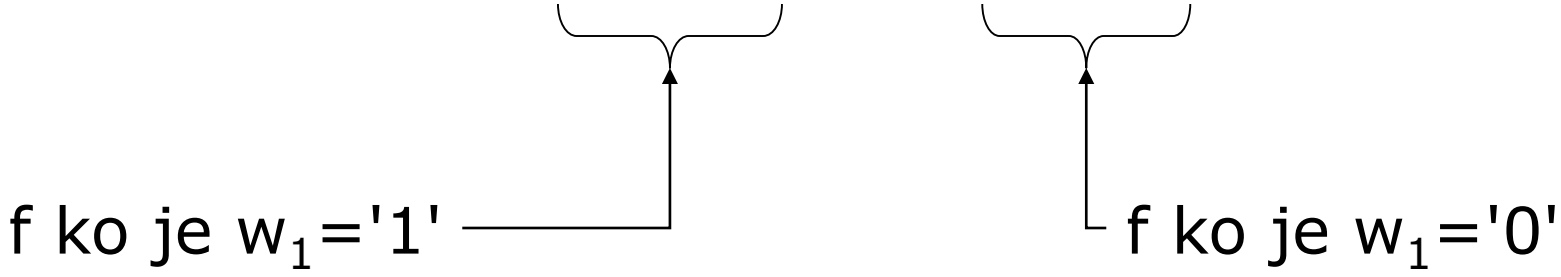
Shannonov razvoj funkcije

- Katerokoli Boolovo funkcijo $f(w_1,\ldots,w_n)$ lahko zapišemo v obliki

  $f(w_1,\ldots, w_n)=(w_1)'\cdot f(0,w_2,\ldots,w_n)+(w_1)\cdot f(1,w_2,\ldots,w_n)$

- Razvoj lahko naredimo po katerikoli izmed *n* spremenljivk funkcije
- Če je $f(w_1, w_2, w_3)= w_1\cdot w_2+w_1\cdot w_3+w_1'\cdot w_2\cdot w_3$
  - Razvoj po w_1 nam da rezultat

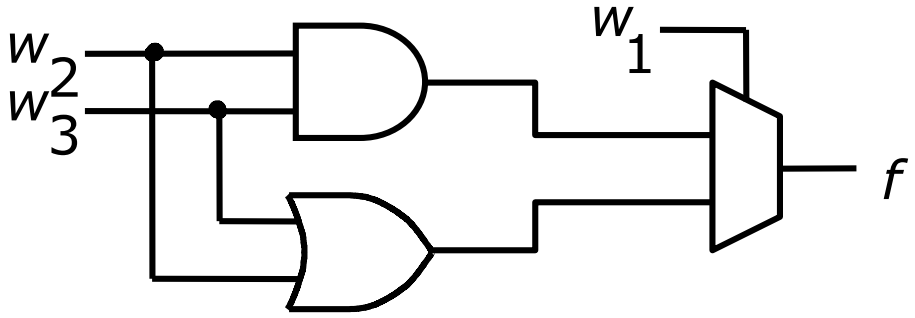
    $f(w_1, w_2, w_3)= w_1\cdot(w_2+w_3)+w_1'\cdot(w_2\cdot w_3)$



Zgled Shannonovega razvoja

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	f
0	$w_2 \cdot w_3$
1	$w_2 + w_3$



Zgled Shannonovega razvoja

$f(x, y, z) = x' \cdot y' \cdot z' + x' \cdot y' \cdot z + x' \cdot y \cdot z + x \cdot y' \cdot z' + x \cdot y' \cdot z$

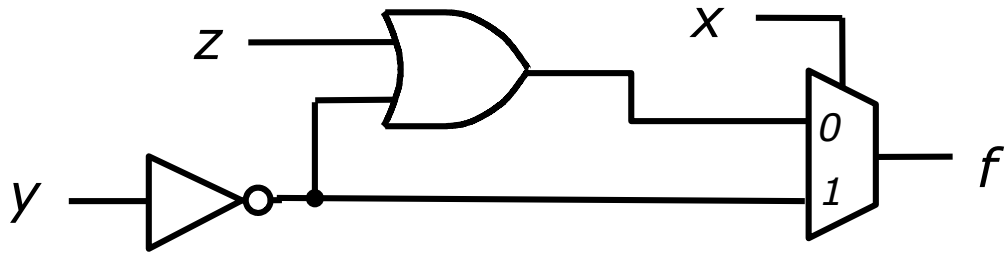
Če izberemo x kot spremenljivko razvoja

$f = x' \cdot (y' \cdot z' + y' \cdot z + y \cdot z) + x \cdot (y' \cdot z' + y' \cdot z)$

$f = x' \cdot (y' \cdot z' + y' \cdot z + y \cdot z + y' \cdot z) + x \cdot (y')$

$f = x' \cdot (y' \cdot (z' + z) + (y + y') \cdot z) + x \cdot (y')$

$f = x' \cdot (y' + z) + x \cdot (y')$



x	y	z	f
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Zgled Shannon-ovega razvoja

$f(x, y, z) = x' \cdot y' \cdot z' + x' \cdot y' \cdot z + x' \cdot y \cdot z + x \cdot y' \cdot z' + x \cdot y' \cdot z$

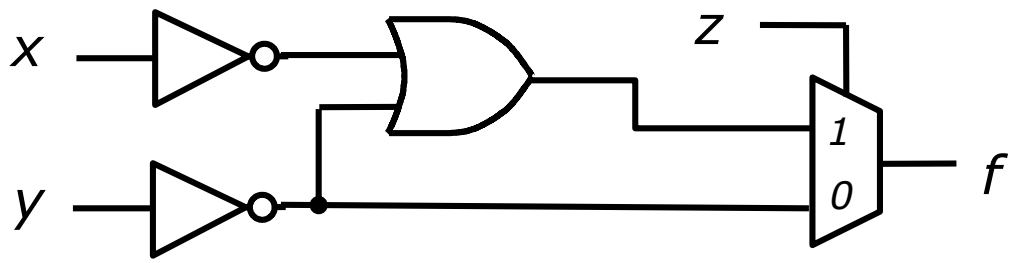
Če izberemo z kot spremenljivko razvoja

$f = z' \cdot (x' \cdot y' + x \cdot y') + z \cdot (x' \cdot y' + x' \cdot y + x \cdot y')$

$f = z' \cdot (x' \cdot y' + x \cdot y') + z \cdot (x' \cdot y' + x' \cdot y + x \cdot y' + x' \cdot y')$

$f = z' \cdot y'(x + x') + z \cdot (x' \cdot (y' + y) + (x' + x) \cdot y')$

$f = z' \cdot y' + z \cdot (x' + y')$



x	y	z	f
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Zgled Shannon-ovega razvoja

$f(x, y, z) = x' \cdot y' \cdot z' + x' \cdot y' \cdot z + x' \cdot y \cdot z + x \cdot y' \cdot z' + x \cdot y' \cdot z$

Če izberemo (x,y) kot spremenljivki razvoja

x	y	z	f	
0	0	0	1	F_0
0	0	1	1	
0	1	0	0	F_1
0	1	1	1	
1	0	0	1	F_2
1	0	1	1	
1	1	0	0	F_3
1	1	1	0	

$f = x' \cdot y' \cdot z' + x' \cdot y' \cdot z + x' \cdot y \cdot z + x \cdot y' \cdot z' + x \cdot y' \cdot z$

$f = x' \cdot y' \cdot (z' + z) + x' \cdot y \cdot z + x \cdot y' \cdot (z' + z)$

$f = x' \cdot y' \cdot (F_0) + x' \cdot y \cdot (F_1) + x \cdot y' \cdot (F_2)$

$f = x' \cdot y' \cdot (1) + x' \cdot y \cdot (z) + x \cdot y' \cdot (1)$

Kaskadna realizacija z MUX

$f(x_1,x_2,x_3,x_4,x_5,x_6)=x_1x_2x_3+x_1'x_2x_4'x_6'+x_1x_2'x_4x_5x_6$

Na I. nivoju izberemo x_1x_2 kot spremenljivko razvoja

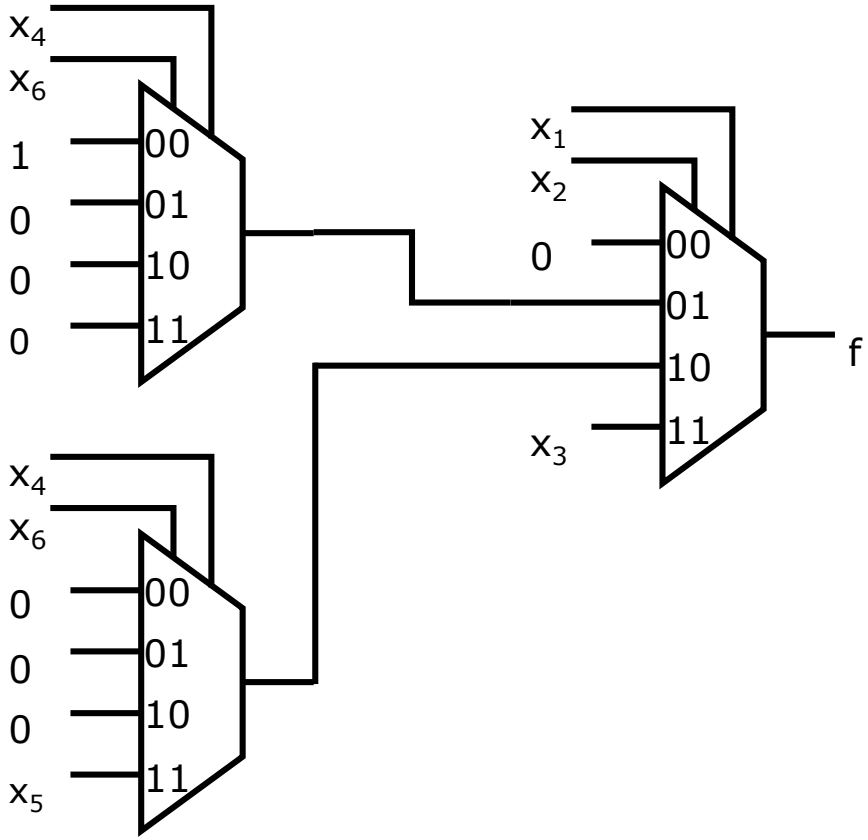
$f(x_1,x_2,x_3,x_4,x_5,x_6)=x_1x_2(x_3)+x_1'x_2(x_4'x_6')+x_1x_2'(x_4x_5x_6)$

Na II. nivoju izberemo x_4x_6 kot spremenljivko razvoja

$f(x_1,x_2,x_3,x_4,x_5,x_6)=x_1x_2(x_3) + x_1'x_2(x_4'x_6') + x_1x_2'(x_4x_5x_6)$

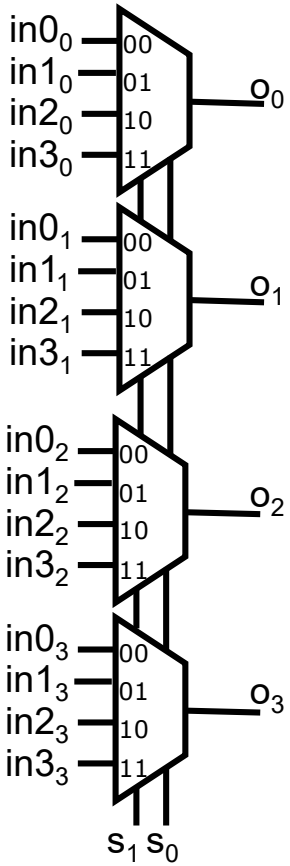
Kaskadna realizacija z MUX

$f(x_1,x_2,x_3,x_4,x_5,x_6)=x_1x_2(x_3)+x_1'x_2(x_4'x_6')+x_1x_2'(x_4x_5x_6)$

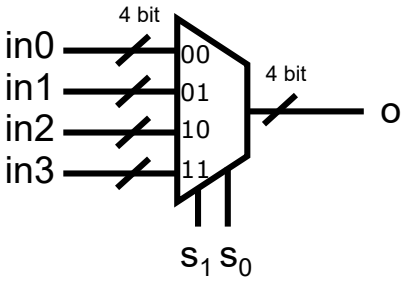


Izvedba večbitnih MUX – izbiralnik vodil (ang. bus multiplexer)

Izvedba bus_mux4x1



Simbol bus_mux4x1



- Izbiralnik vodil (ang. bus multiplexer) izbira med več večbitnimi vrednostmi.
- Če želimo izdelati N bitni izbiralnik vodil MUX, to storimo tako, da vzamemo N N/1 MUX in jih vežemo z naslovnim vodilom skupaj.

Načrtovanje digitalnih vezij

Gradniki kombinacijskih vezij:
Polja in večbitna števila v VHDL

Polja v VHDL (ang. array)

Polja združujejo signale in spremenljivke istega tipa:

```
type ime_tipa is array (range) of element_tipa;
```

Eden (od dveh) osnovnih tipov, definiran v paketu `ieee.std_logic_1164`, je polje elementov `std_logic_vector`, ki je predstavljen kot neomejeno (ang. unconstrained) 1D polje elementov `std_logic`:

```
TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF
std_logic;
```

Elemente 1D polj lahko uredimo v dveh zaporedjih:

- Naraščajoče:
```
type type_array_int is array (integer range <>) of
integer;
variable tc_up : type_array_int(0 to 9);
```
- Padajoče:
```
variable tc_down : type_array_int(9 downto 0);
```

Nastavljanje vrednosti polj - agregati

- Agregat je osnovna operacija, ki združuje enega ali več elementov v sestavljene elemente polja ali zapisa (ang. record).
- Elementi agregata so lahko urejeni položajno, ali imensko:
  - Položajno (ang. *positional* association) :
    ```
    variable x : BIT_VECTOR (0 to 3) :=
    ('0','1','0','1');
    ```
    Navedene so *vse* vrednosti,
    nepovezane preskočimo z besedo `open`
  - Imensko (ang. *named* association):
    ```
    variable y : BIT_VECTOR (0 to 3) :=
    (1=>'1',0=>'0',3=>'1',2=>'0');
    ```
  - Ni nujno (a prihrani marsikatero solzo),
    če so navedene vse vrednosti.

Uporaba agregatov

```
signal x : std_logic_vector (15 downto 0);
x <= (15 downto 8 => '0', 7 downto 0 => '1');

signal x : std_logic_vector (15 downto 0);
x <= (14 downto 8 => '0', others => '1');

signal x : std_logic_vector (15 downto 0);
x <= (others => 'Z');

signal x : std_logic_vector (15 downto 0);
x <= (15 | 7 downto 0 => '1', others => '0');
```

Naraščajoče zaporedje zapisa elementov (to)

- Tak način predstavitve je primeren za organizacijo elementov v polju – v seznamu je prvi element z najnižjim indeksom, zato to ni naraven način za zapis dvojiških števil (normalno pišemo MSB na levi).

```
SIGNAL C: STD_LOGIC_VECTOR (0 to 3);
```

C je 4-bitni STD_LOGIC signal

- Če izvedemo prireditev vrednosti C <= "1110";
- Dostop do bitov:
  C(0) – 1-bitna veličina (LSB) – položaj skrajno levo
  C(3) – 1-bitna veličina (MSB) – položaj skrajno desno
- Zapis elementov v urejenem seznamu od leve proti desni (oz. zaporedju obravnave elementov):
  1(LSB) 1 1 0(MSB)

Padajoče zaporedje zapisa elementov polja (downto)

- Ta način uporabljamo za predstavitev števil, saj imamo MSB mesto v zapisu na skrajno levem mestu (zapis števila z leve proti desni: MSB→LSB).

```
SIGNAL x: STD_LOGIC_VECTOR (3 downto 0);
```

x je 4-bitni STD_LOGIC signal
- X(3) je MSB – položaj skrajno levo
- X(0) je LSB – položaj skrajno desno
- Če izvedemo x <= "1110";
  Dobimo zapis elementov v urejenem seznamu:
  1(MSB) 1 1 0(LSB)

Večbitna števila v VHDL - std_logic_1164

- Naj bosta definirana signala:

```
signal x : std_logic_vector(3 downto 0);
signal y : std_logic_vector(0 to 3);
```

- Pozor: Če izvedemo enostaven določitveni izraz y <= x, to po elementih pomeni prireditve:
  - x(3) → y(0),
  - x(2) → y(1),
  - x(1) → y(2),
  - x(0) → y(3).

-
```
signal little_endian : std_logic_vector(0 to 7) :=
( 0 => '1', others => '0');
```
-
```
signal big_endian : std_logic_vector(7 downto 0) :=
( 0 => '1', others => '0');
```

Preslikava na polje z enim elementom

- Naj bo dano polje:

```
signal single_element_array : std_logic_vector(0 downto 0);
```

- Kako priredimo vrednost elementa polju:

-
```
single_element_array <= '0';
--single_element_array je polje (no-go)!
```
-
```
single_element_array <= ('0');
-- oklepaj ne naredi polja (no-go)!
```
-
```
single_element_array <= (0 => '0');
-- priredimo polje, ki ima na mestu 0 vrednost '0'(go)!
single_element_array <= (others => '0');
-- agregat, ki se prevede na mesto 0 in vrednost '0'(go)!
```
-
```
single_element_array(0) <= '0';
-- prirejamo ELEMENT polja. Priredimo vrednost '0'(go)!
```

Območja z negativnimi indeksi

- Če je tip urejenega seznama predznačen, lahko uporabljamo tudi negativne indekse:
- `type neg_index_array is array (-5 to 5) of integer;`
- `variable my_neg_array : neg_index_array;`
- `my_neg_array(-5) := 15;`
- `my_neg_array(-3 to -1) := 12 & 13 & 14;`
  `-- & je operator sestavljanja (concatenation)`

Naštevni indeksi urejenega seznama

- Tip urejenega seznama ni nujno številčni, važno je da je našteven in diskreten.
- type myEnum is (a,b,c,d,e);
- type myArray is array(a to d) of integer;
- constant myArrayConstant : myArray := (a => 42, others => 0);

Večdimenzionalna polja

- Polja lahko združujemo v večdimenzionalne strukture:

```
type bcd_type is array(3 downto 0) of std_logic;
type bcd_arr_type is array(7 downto 0) of bcd_type;
type bcd_2darr_type is array(5 downto 0, 3 downto 0)
of std_logic;
signal bcd_8_digit_nr : bcd_arr_type ;
signal bcd_6_digit_nr : bcd_2darr_type;
signal bcd_digit0, bcd_digit1, bcd_digit2, bcd_digit3
: bcd_type;
```

Nastavljanje večdimenzionalnih polj

- Nastavljanje vrednosti po *osnovnih* elementih:

```
bcd_8_digit_nr (0)(0) <= '1';
bcd_6_digit_nr (0, 0) <= '1';
```

- Nastavljanje vrednosti po *sestavljenih* elementih:

```
bcd_6_digit_nr <= (bcd_digit0, bcd_digit0,
bcd_digit1, bcd_digit2, bcd_digit3 );
```

- Nastavljanje *več* elementov *naenkrat* z uporabo agregatov:

```
bcd_8_digit_nr <= (others => nibble0);
bcd_6_digit_nr <= (others => nibble5);
bcd_8_digit_nr <= (others => (others => '0'));
bcd_6_digit_nr <= (others => (others => '1'));
```

Osnovne lastnosti polj (vektorjev) v VHDL

Lastnosti (ang. attributes):

```
signal a : std_logic_vector (7 downto 0);
signal b : std_logic_vector (0 to 7);
```

a'range	= 7 downto 0	b'range	= 0 to 7
a'length	= 8	b'length	= 8
a'high	= 7	b'high	= 7
a'low	= 0	b'low	= 0
a'left	= 7 (MSB položaj)	b'left	= 0
a'right	= 0 (LSB položaj)	b'right	= 7

Za naraščajoče zaporedje 1D polj (to) velja:
T'left = T'low, T'right = T'high.

Za padajoče zaporedje zapisa 1D polj (downto) velja:
T'left = T'high, T'right = T'low.

Sestavljanje večbitnih števil v VHDL

```
entity concat_demo is
Port( X : in  STD_LOGIC_VECTOR (2 downto 0);
   Q0, Q1, Q2 : out  STD_LOGIC_VECTOR (2 downto 0) );
end concat_demo;
architecture arch of concat_demo is
signal REG : STD_LOGIC_VECTOR(2 downto 0):= "001";
begin
   REG <= X;
   Q0 <= REG(0) & REG(2 downto 1); -- reg ror 1
   Q1 <= REG(0) & REG(1) & REG(2);
   Q2 <= X;
end arch;
```

Operator sestavljanja (ang. concatenation)

Konstante (CONSTANT)

Konstanta (**CONSTANT**) nima strojnega ekvivalenta! (služi lažjemu programiranju)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity const_demo is
Port (   x1 : in STD_LOGIC;
         x2 : in STD_LOGIC_VECTOR(3 downto 0);
         y1 : out STD_LOGIC;
         y2 : out STD_LOGIC_VECTOR(3 downto 0)      );
end const_demo;
architecture arch of const_demo is
constant ENA : STD_LOGIC := '1';   -- enobitna konstanta
constant PETNAJST_2 : STD_LOGIC_VECTOR(3 downto 0) := "1111";-- bin
constant PETNAJST_16 : STD_LOGIC_VECTOR(3 downto 0) := x"F";-- hex
constant C15 : STD_LOGIC_VECTOR(3 downto 0) := (others => '1');-- hex
constant C1 : STD_LOGIC_VECTOR(3 downto 0) := (0 => '1', others => '0');
-- splošna konstanta "0001" → others je vedno na koncu
constant C3 : STD_LOGIC_VECTOR(3 downto 0) := (0|1 => '1', others => '0');
-- splošna konstanta "0011" → others je vedno na koncu
begin
 y1 <= x1 xor ENA; -- y1 = not x1
 y2 <= x2 xor PETNAJST_16; -- y2 = not x2 (stiribitna operacija)
end arch;
```

Primeri *integer* konstant

```
12343             -- desetiška konstanta
2#10011110#       -- dvojiška konstanta
8#720#            -- osmiška konstanta
16#FFFF0ABC#      -- šestnajstiška konstanta
16#FFFF_0ABC#     -- uporaba separatorja (_)
                     za lažje branje vrednosti
```

Parametriziranje v VHDL

- Večkrat želimo definirati lastnost posameznega primera komponente posebej, a se med seboj različni primeri iste entitete razlikujejo samo po *parametrih* (npr. zakasnitev vrat, bitna širina strukture v smislu *n*-bitni izbiralnik ...)
- Prilagajanje klicev parametrov omogoča parametrizacija z generičnimi konstantami (`generic`)
- Generične konstante omogočajo posplošitev struktur namesto, da bi za vsak primer rabili ustvariti novo strukturo.
- Parametri entitete se lahko nastavljajo v nadrejenih (klicočih) entitetah.

Primer parametrizacije: Unarni logični operatorji nad poljem

- Unarni logični operatorji – včasih tudi *operatorji redukcije* (ang. reduction operators), ki se izvajajo med elementi polja v tipu **STD_LOGIC_VECTOR** so programirani z rekurzivnimi funkcijami:
  - v standardu VHDL-2002 (**or** , **and** , **xor** in njihove negacije):
    npr. **rezultat <= or_reduce vektor;**
    (glej reduce pack; VHDL.org)
  - v standardu VHDL-2008 (**or** , **and** , **xor** in negacije):
    npr. **rezultat <= or vektor;**
- Primer unarne OR operacije:

```
signal temp : STD_LOGIC_VECTOR(3 downto 0) := "1001";
signal bitni_or : STD_LOGIC; -- rezultat bitne or operacije

-- unarna or operacija - operacija or med vsemi biti
bitni_or  <= temp(0) or temp(1) or temp(2) or temp(3)
```

Unarni logični operatorji

V VHDL-93 teh operatorjev ni, zato moramo za njihovo izvedbo zapisati when-else stavek ali zanko v procesnem stavku.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity reduction_operators is
generic ( N: Natural := 10 );
port ( A: in STD_LOGIC_VECTOR ( N-1 DOWNTO 0 );
       reduced_AND, reduced_OR : out STD_LOGIC);
end reduction_operators;
architecture vhdl_93_comb of reduction_operators is
begin
reduced_OR <= '0' when (A = (A'range => '0')) else '1';
      -- redukcijski OR
reduced_AND <= '1' when (A = (A'range => '1')) else '0';
      -- redukcijski AND
end vhdl_93_comb;
```

Unarni logični operatorji - proces

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity reduction_operators  is
generic (      N: Natural := 10 );
port ( A: in STD_LOGIC_VECTOR (N-1 DOWNTO 0);
       reduced_OR: out STD_LOGIC);
end reduction_operators;
architecture vhdl_93_process of reduction_operators is
begin
or_reduction: process (A)
variable result: std_logic;
begin
   for i in A'range loop
        result := result or A(i);
        exit when result = '1';
   end loop;
   reduced_OR <= result;
end process;
end vhdl_93_process;
```

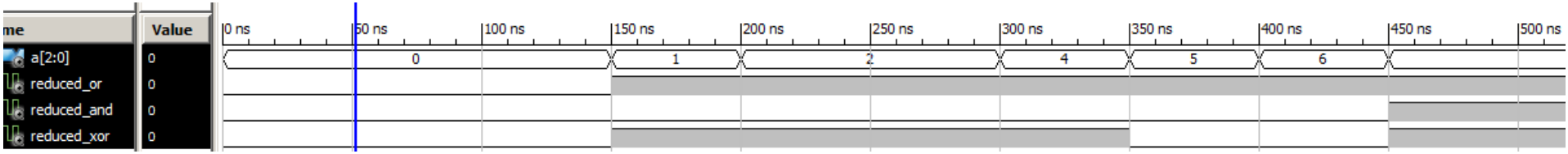
Splošne konstante z uporabo others

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity reduction_operators_tb is
generic (   N: natural := 3 );
end reduction_operators_tb;

architecture test of reduction_operators_tb is
     component reduction_operators  is
     generic ( N: Natural := 10 );
     port ( A: in STD_LOGIC_VECTOR (N-1 DOWNTO 0);
            reduced_OR : out STD_LOGIC);
     end component;

constant ZERO : std_logic_vector(N-1 DOWNTO 0)
:= (others => '0');
constant ONE : std_logic_vector(N-1 DOWNTO 0)
:= (0 => '1', others => '0');
constant TWO : std_logic_vector(N-1 DOWNTO 0)
:= (1 => '1', others => '0');
constant THREE : std_logic_vector(N-1 DOWNTO 0)
:= (0 | 1 => '1', others => '0');
constant FOUR : std_logic_vector(N-1 DOWNTO 0)
:= (2 => '1', others => '0');
constant FIVE : std_logic_vector(N-1 DOWNTO 0)
:= (0 => '1', 2 => '1', others => '0');
constant SIX : std_logic_vector(N-1 DOWNTO 0)
:= (1 | 2 => '1', others => '0');
constant SEVEN : std_logic_vector(N-1 DOWNTO 0)
:= (0 | 1 | 2 => '1', others => '0');
signal A : std_logic_vector (N-1 DOWNTO 0) := ZERO;
signal reduced_OR : STD_LOGIC;
```

```
begin
    U1: reduction_operators
    generic map (N => N)
    port map (A => A,
    reduced_OR=> reduced_OR);
stim_proc: process
begin
          A <= ZERO;                    --0
          wait for 50 ns;
          A <= ONE after 100 ns;        --1
          wait for 50 ns;
          A <= TWO after 100 ns;        --2
          wait for 50 ns;
          A <= THREE after 100 ns;      --3
          wait for 50 ns;
          A <= FOUR after 100 ns;       --4
          wait for 50 ns;
          A <= FIVE after 100 ns;       --5
          wait for 50 ns;
          A <= SIX after 100 ns;        --6
          wait for 50 ns;
          A <= SEVEN after 100 ns;      --7
          wait;
end process;
end test;
```

Unarni logični operatorji - naloga

- Po podanem zgledu unarnega operatorja OR programirajte še logična unarna operatorja XOR in AND v procesnem stavku.
- Pravilnost delovanja preverite na spodaj prikazanih vrednostih



Načrtovanje digitalnih vezij

Gradniki kombinacijskih vezij:
Izbiralniki v VHDL

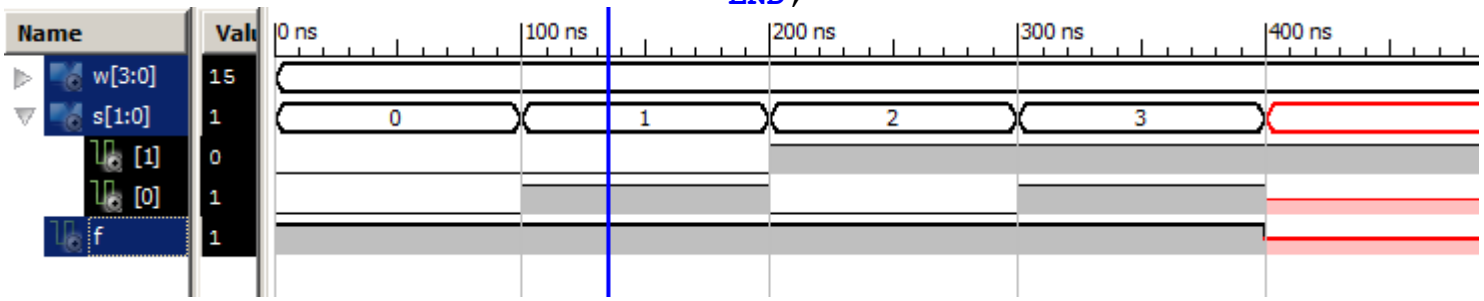
WHEN OTHERS (MUX 4/1)

```
ENTITY mux4to1 IS
PORT (w : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
      s : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
      f : OUT STD_LOGIC );
END mux4to1;
ARCHITECTURE arch OF mux4to1 IS
BEGIN
      WITH s SELECT
      f <=  w(0) WHEN "00",
            w(1) WHEN "01",
            w(2) WHEN "10",
            w(3) WHEN "11",
            'X' WHEN OTHERS;
END arch;
```

Simulacija MUX 4/1

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mux_4_1_tb IS
END mux_4_1_tb;

ARCHITECTURE behavior OF mux_4_1_tb IS
COMPONENT mux4to1 IS
PORT (
    w : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    s : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    f : OUT STD_LOGIC );
END COMPONENT;
signal w : STD_LOGIC_VECTOR(3 DOWNTO 0) :=
    (others => '1');
signal s : STD_LOGIC_VECTOR(1 DOWNTO 0) :=
    (others => '0');
signal f : STD_LOGIC;
```

```
BEGIN
   uut: mux4to1 PORT MAP (
    w => w,
    s => s,
    f => f);

   stim_proc: process
   begin
         s <= (others => '0');
         wait for 100 ns;
         s <= "01";
         wait for 100 ns;
         s <= "10";
         wait for 100 ns;
         s <= "11";
         wait for 100 ns;
         s <= "1X";
         wait;
    end process;
END;
```



Implementacija MUX 4/1

```
Started : "Synthesize - XST".
Running xst...
Command Line: xst -intstyle ise -ifn
    "mux4to1.xst" -ofn "mux4to1.syr"
Reading design: mux4to1.prj
=========================================
* HDL Parsing
=========================================
Parsing VHDL file "mux_4_1.vhd" into library
    work
Parsing entity <mux4to1>.
Parsing architecture <arch> of entity
    <mux4to1>.
=========================================
* HDL Elaboration
=========================================
Elaborating entity <mux4to1> (architecture
    <arch>) from library <work>.
```

```
=========================================
* HDL Synthesis
=========================================
    Synthesizing Unit <mux4to1>.
    Related source file is "mux_4_1.vhd".
    Found 1-bit 4-to-1 multiplexer for
    signal <f> created at line 12.
    Summary:
    inferred   1 Multiplexer(s).
Unit <mux4to1> synthesized.
=========================================
HDL Synthesis Report
Macro Statistics
# Multiplexers                  : 1
 1-bit 4-to-1 multiplexer       : 1
=========================================
    Advanced HDL Synthesis
=========================================
    Advanced HDL Synthesis Report
Macro Statistics
# Multiplexers                  : 1
 1-bit 4-to-1 multiplexer       : 1
```

Kako ocenimo kvaliteto implementiranega kombinacijskega vezja?

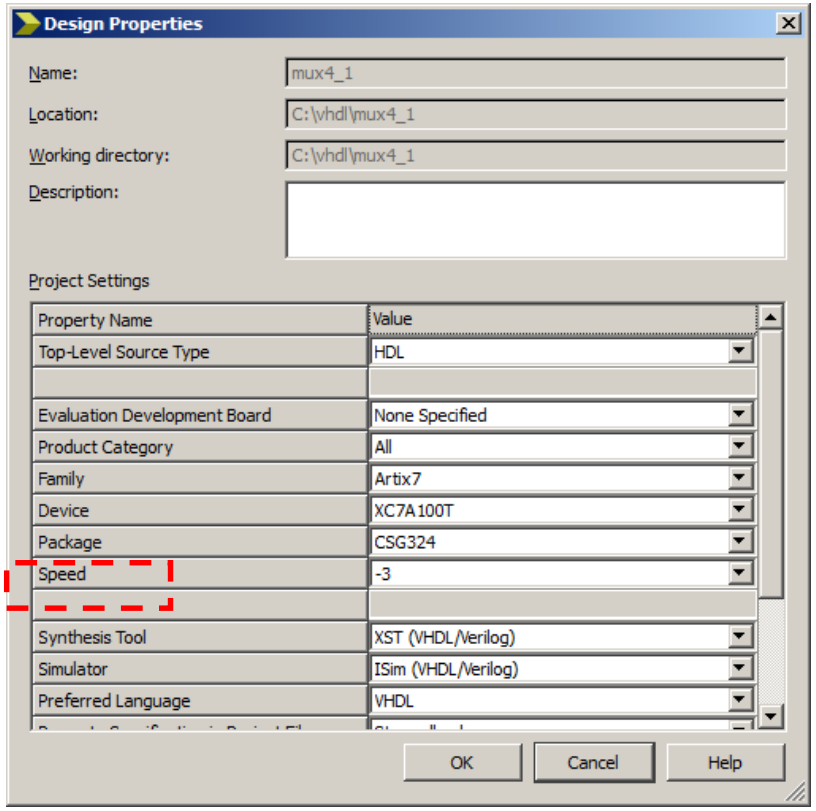
S stališča zakasnitve:

```
=========================================================
Low Level Synthesis
=========================================================
Optimizing unit <mux4to1> ...
Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block
     mux4to1, actual ratio is 0.
Final Macro Processing ...
=========================================================
Final Register Report
Found no macro
=========================================================
```

```
=========================================================
Partition Report
=========================================================
Partition Implementation Status
-------------------------------
  No Partitions were found in this design.
-------------------------------
=========================================================
Design Summary
=========================================================
Clock Information:
------------------
No clock signals found in this design
Asynchronous Control Signals Information:
-----------------------------------------
No asynchronous control signals found in this design
Timing Summary:
---------------
Speed Grade: -3
   Minimum period:
No path found
   Minimum input arrival time before clock:
No path found
   Maximum output required time after clock:
No path found
   Maximum combinational path delay: 1.071ns
=========================================================
Process "Synthesize - XST" completed successfully
```

Speed grade v Xilinx ISE

- Parameter implementacije projekta speed grade je odvisen od vrste čipa, ki ga uporabljamo za izvedbo (FPGA/CPLD).
- Za CPLD predstavlja speed grade čas, ki je potreben za prehod preko čipa od vhoda do izhoda (v smislu enostavnega določitvenega izraza (out<=in).
- Če je za CPLD v projektu izbrani speed grade -10, potem bo čip *gotovo* postavil izhodni priključek z zakasnitvijo največ 10 ns od postavitve vhodnega priključka.
- Za CPLD velja: nižja številka, hitrejši čip. Ta definicija je uveljavljena med različnimi proizvajalci in omogoča primerjavo med CPLD čipi.
- V starejših Xilinx FPGA (pred Virtex družino) je speed grade predstavljal čas prehoda signala preko LUT (nižja vrednost → hitrejši čip).
- Od družine Virtex naprej speed grade predstavlja *prirastek* k pohitritvi (večja številka – hitrejši čip). V Xilinx FPGA vezjih je ta prirastek pohitritve med 10 in 15%. Primer: speed grade=-5 je ca. 10% hitrejši kot speed grade = -4.



WHEN ELSE (MUX 2/1)

```
ENTITY mux2to1 IS
PORT ( w0, w1, s : IN STD_LOGIC;
       f : OUT STD_LOGIC
     );
END mux2to1;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
      f <= w0 WHEN s = '0' ELSE w1;
END Behavior;
```

Procesni stavek (MUX 2/1)

```
ARCHITECTURE arch OF mux2to1 IS
BEGIN
PROCESS ( w0, w1, s )
BEGIN
   IF s = '0' THEN
     f <= w0 ;
   ELSE
     f <= w1 ;
   END IF ;
END PROCESS ;
END arch;
```

Seznam občutljivosti

IF-THEN-ELSE izraz,
ki realizira MUX 2/1

Proces (MUX 2/1)

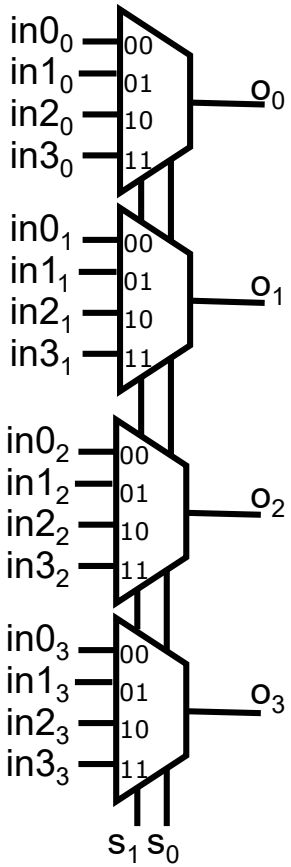
```
ARCHITECTURE arch OF mux2to1 IS
BEGIN
PROCESS ( w0, w1, s )
   BEGIN
      CASE s IS
         WHEN '0'       => f <= w0;
         WHEN OTHERS    => f <= w1;
      END CASE ;
   END PROCESS ;
END arch ;
```

Parametriziran izbiralnik (MUX N/1)

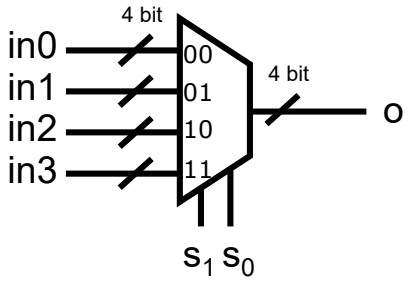
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;
ENTITY muxnto1 IS
   generic( n_addr: natural := 2 );
   PORT ( s : in std_logic_vector(n_addr - 1 downto 0);
        w : in std_logic_vector(2**n_addr - 1 downto 0);
        f : OUT STD_LOGIC
        );
END muxnto1;
ARCHITECTURE rtl OF muxnto1 IS
BEGIN
   f <= w(to_integer(unsigned(s)));
END rtl;
```

Parametriziran izbiralnik vodil

Izvedba bus_mux4x1



Simbol bus_mux4x1




```
library ieee;
use ieee.std_logic_1164.all;
entity bus_mux_4x1 is
generic ( W: integer := 8 );
port (
    in0, in1, in2, in3:
    in std_logic_vector (W-1 downto 0);
    s: in std_logic_vector (1 downto 0);
    o: out std_logic_vector (W-1 downto 0)
);
end;
architecture arch of mux_4x1 is
begin
o <=     in0 when s = "00" else
         in1 when s = "01" else
         in2 when s = "10" else
         in3 when s = "11" else
         (others => 'X');
end arch;
```

Izbiralnik vodil (ang. bus multiplexer) izbira med več večbitnimi vrednostmi. Parametrizirana je samo širina vodila (W).

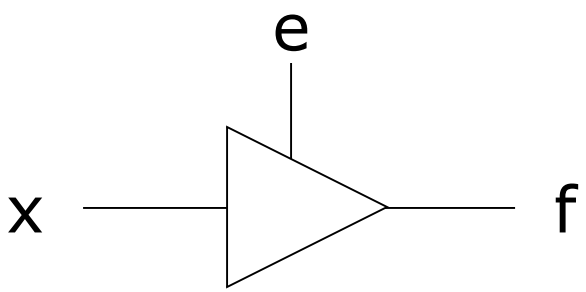
Načrtovanje digitalnih vezij

Gradniki kombinacijskih vezij:

Ojačevalniki in realizacija tri-stanjske logike

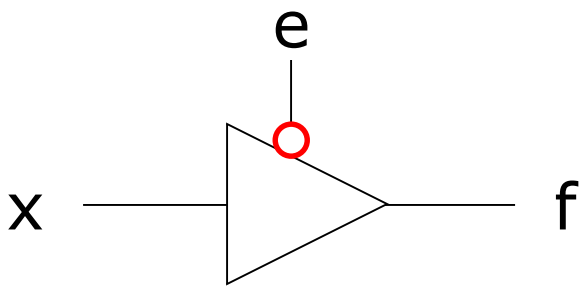
3-stanjski ojačevalniki in vrata

Tri-stanjski ojačevalnik (3-state buffer)



f=x če je e=1, sicer f=Z

f<=x **when** e='1' **else** 'Z';



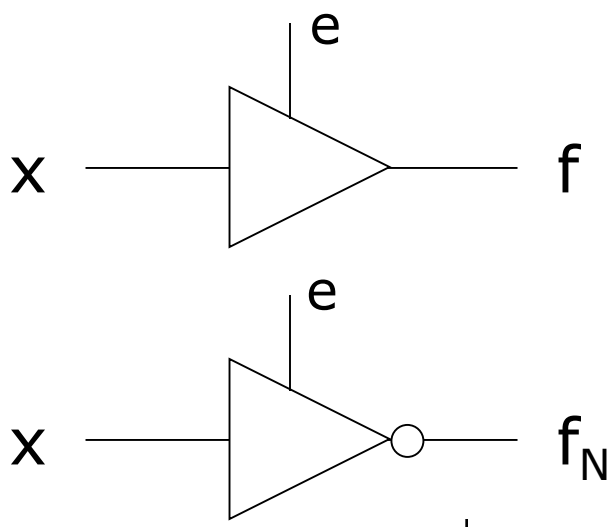
f=x če je e=0, sicer f=Z

f<=x **when** e='0' **else** 'Z';

Če je izhod invertiran,
je to tristanjski inverter (3-state inverter)

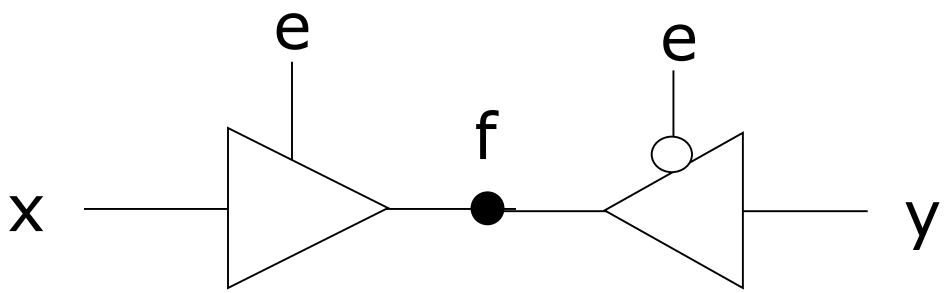
3-stanjski ojačevalniki

- e=1 ➔ f=x
  (ojačevalnik na svojem izhodu dal enako vrednost kot f)
- e=0, ➔ f=Z
  (ojačevalnik svoj izhod izključi od obeh logičnih nivojev).
- Tok takrat ne bo tekel:
  - Ne v izhod (sink),
  - Ne iz izhoda (source)



e	x	f_N	f
0	0	Z	Z
0	1	Z	Z
1	0	1	0
1	1	0	1

Povezovanje več izhodov - vodilo



f=x če je e=1, sicer f=Z

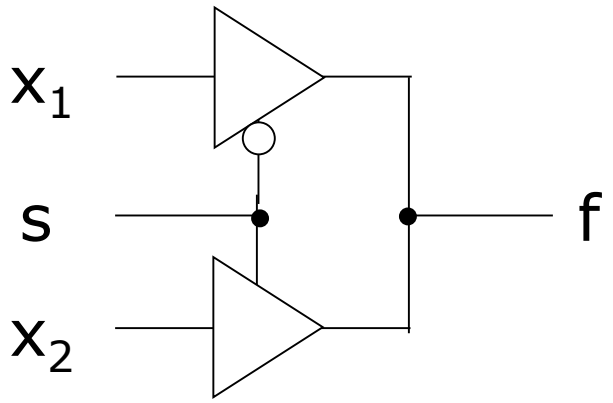
f=y če je e=0, sicer f=Z

VHDL: f<=x **when** e='1' **else** 'Z';

Katerega VHDL tipa je f, če povezujemo priključka kot je narisano?
Kako je treba nastaviti priključek v UCF datoteki? (weak PULL-UP upor)

Uporaba tri-stanjskih ojačevalnikov

MUX 2/1 s tristanjskimi ojačevalniki



s	x1	x2	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Izhodi 3-stanjskih ojačevalnikov so vezani skupaj.

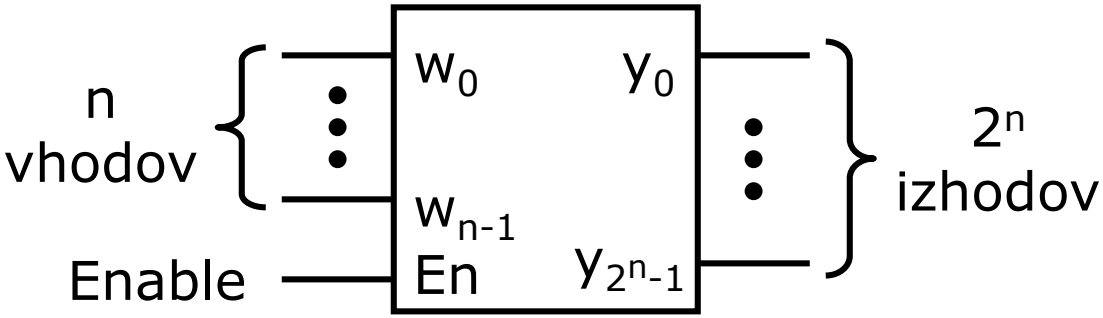
```
f <= x1 when s='0' else x2;
```

Načrtovanje digitalnih vezij

Gradniki kombinacijskih vezij:

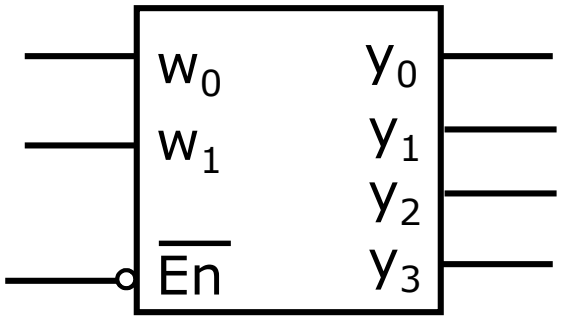
Dekoderji/demultiplekserji,
kodirniki, pretvorniki kode
in primerjalniki velikosti

Dekoderji

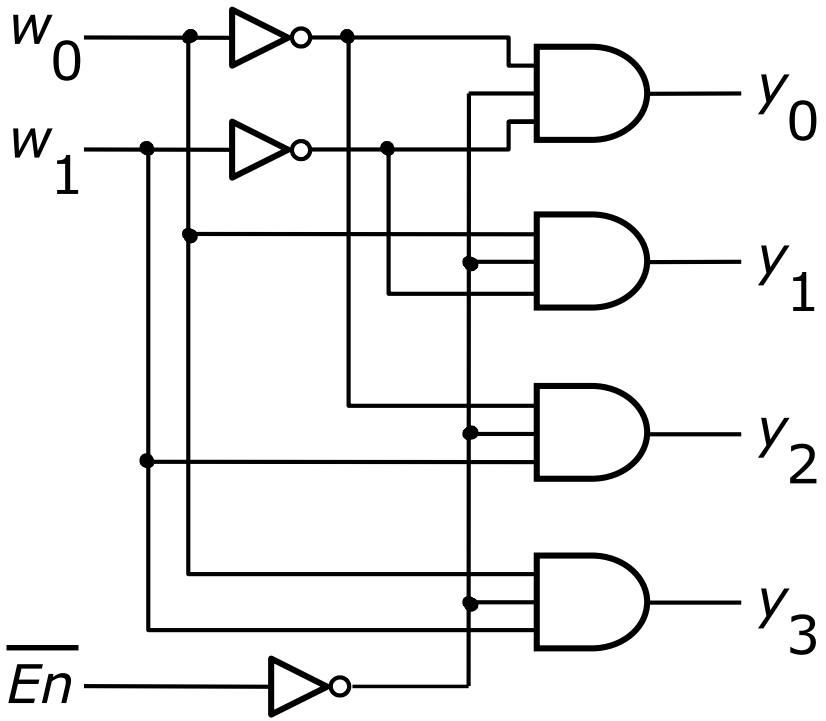


- Enable='0' ➔ ni izbran noben izhod
  (vsi onemogočeni – vsi neaktivni)
- Enable='1' ➔ aktiven (izbran) samo eden od izhodov glede na kombinacijo vhodov
  koda 1-od-N oz. koda "ena naenkrat"
  (ang. one-hot encoding)

Vezje dekoderja 2/4

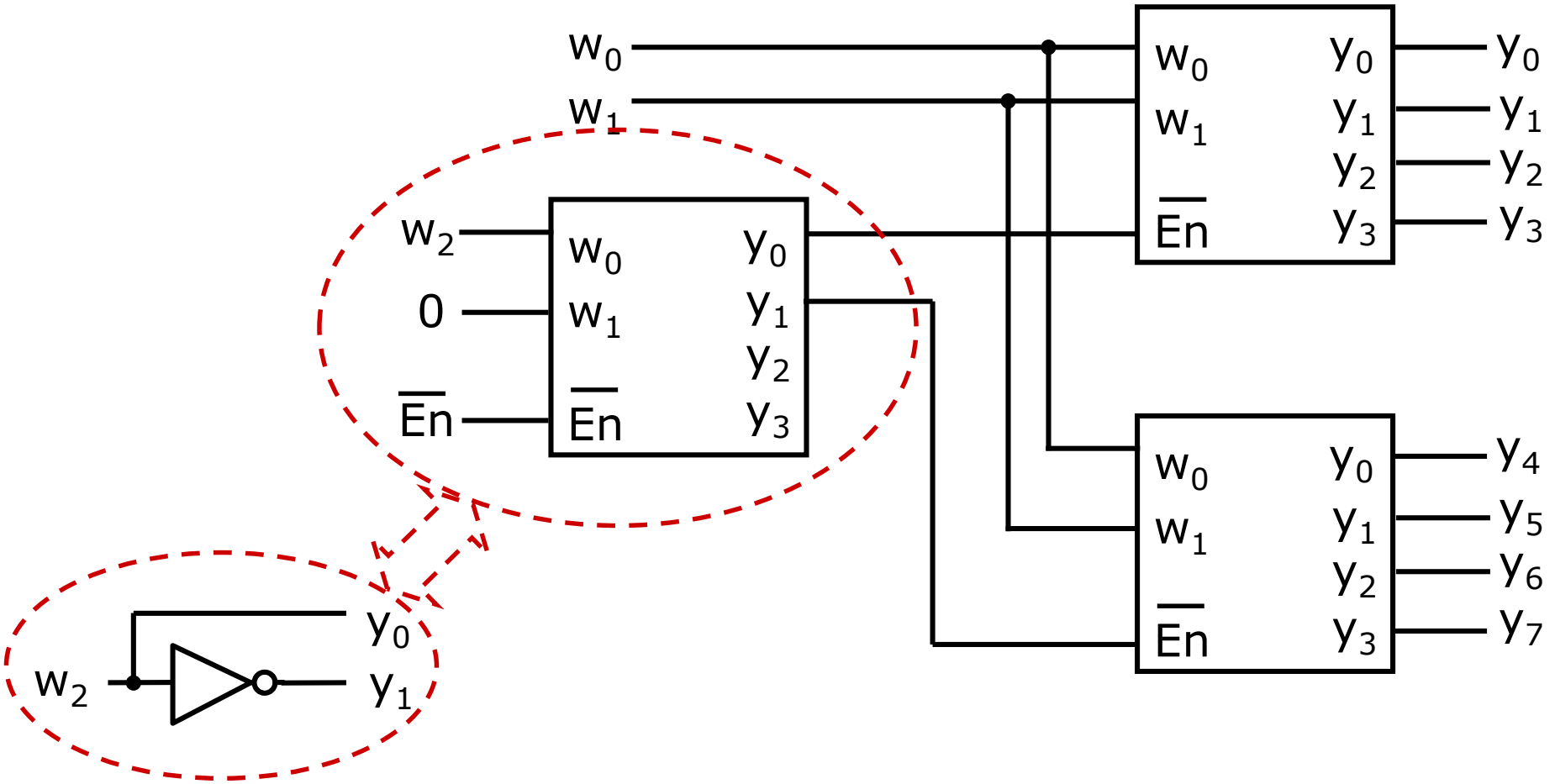


$\overline{En}$	w_1	w_0	y_0	y_1	y_2	y_3
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	1	0	0	0	1	0
0	1	1	0	0	0	1
1	X	X	0	0	0	0

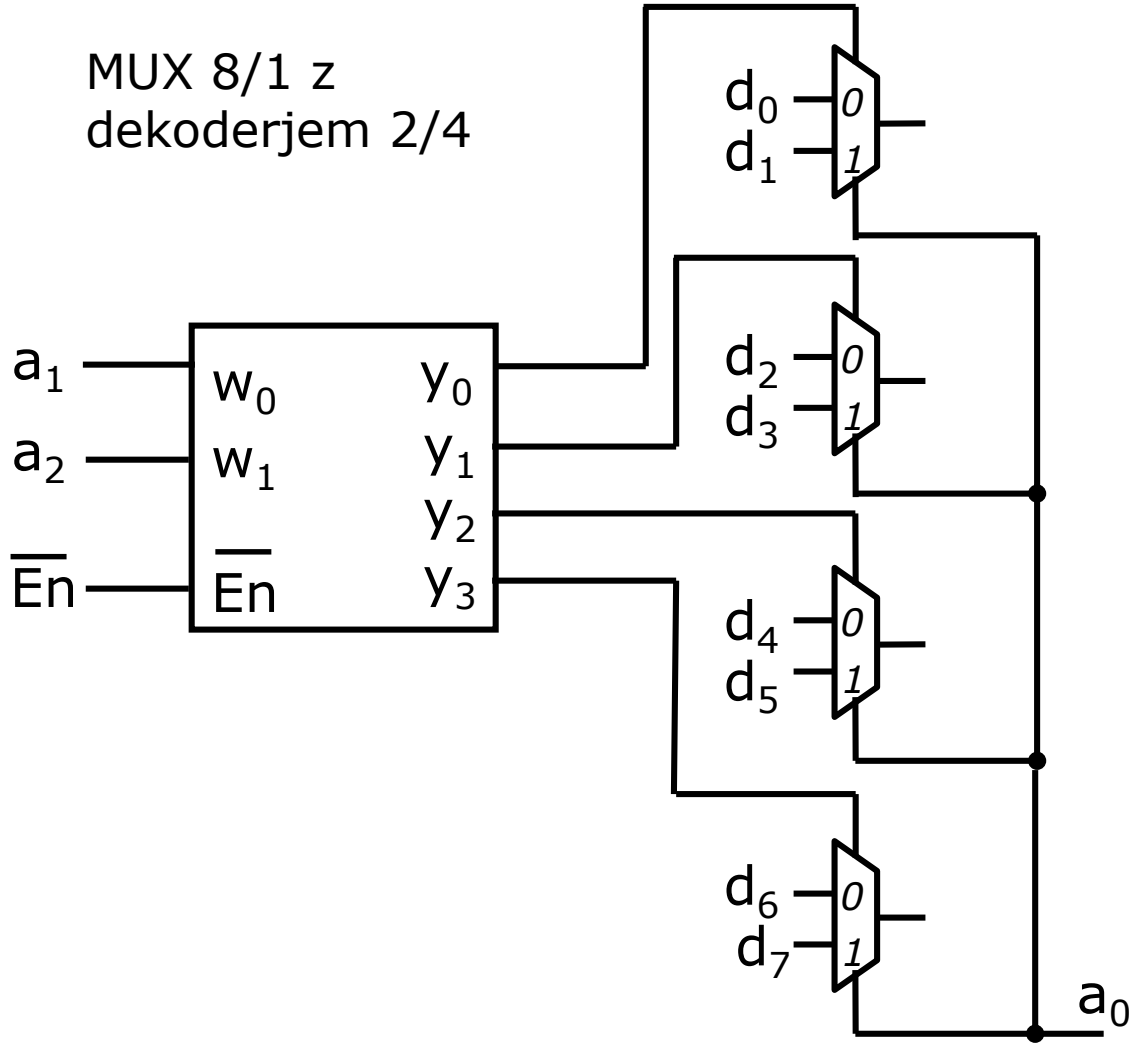


Realizacija večjih dekoderjev iz manjših

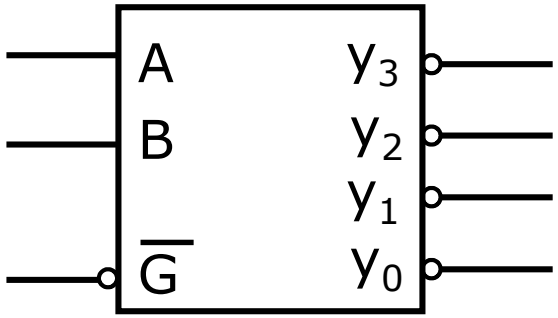
dekoder 3/8



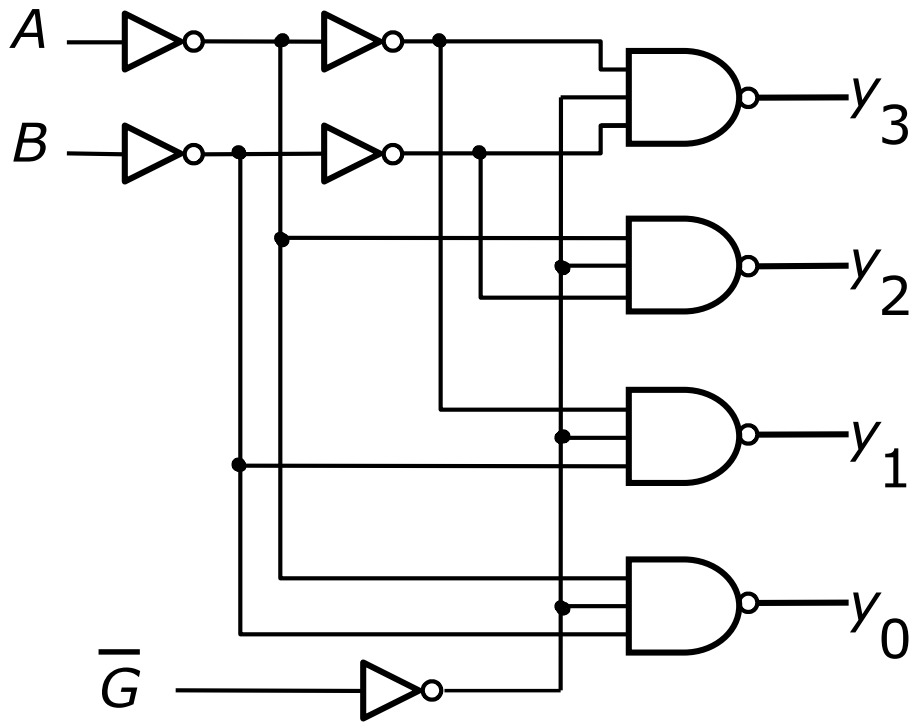
Realizacija večjega MUX z uporabo dekoderjev



Dvojni dekoder 2/4 - 74139

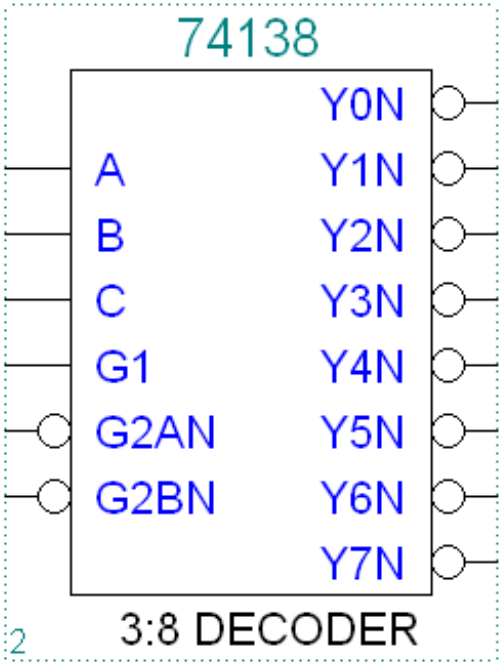


$\overline{G}$	B	A	y_0	y_1	y_2	y_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0



Podobno vezje je 74138, ki realizira 3/8 dekoder.

Dekoder 3/8 - 74138



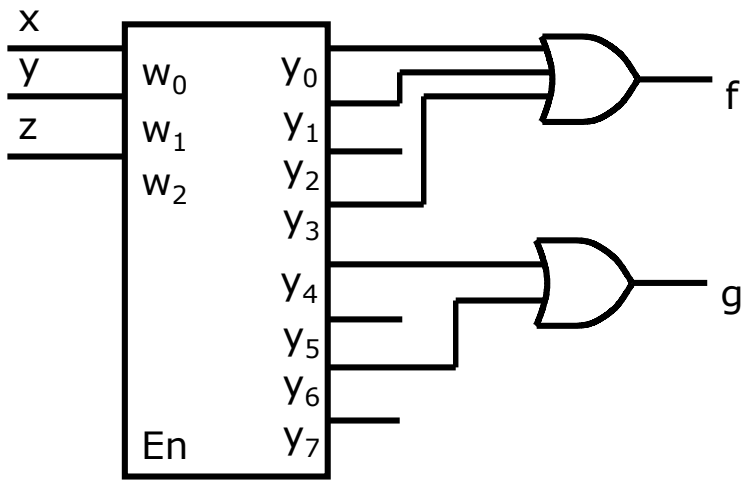
Inputs					Outputs							
Enable		Select										
G1	G2*	C	B	A	Y0N	Y1N	Y2N	Y3N	Y4N	Y5N	Y6N	Y7N
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

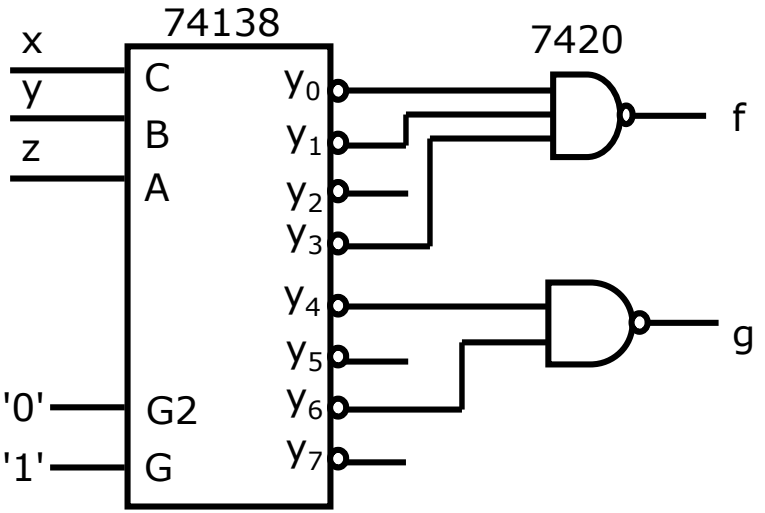
* G2 = G2AN + G2BN

Ko je izhod izbran, je njegova vrednost nizka

Uporaba dekoderjev

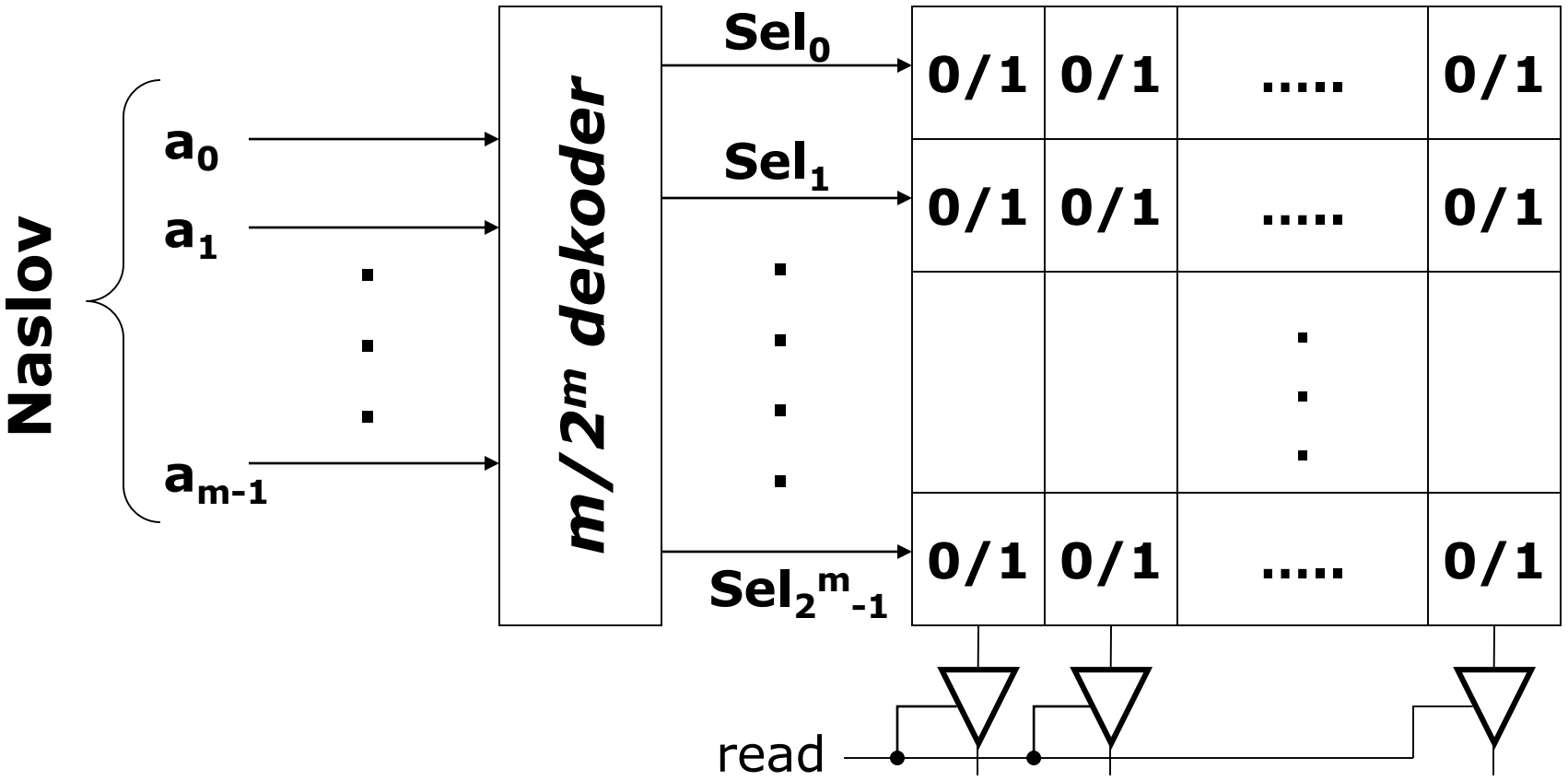
- Tipična uporaba dekoderjev je pri realizaciji funkcij v DNO obliki – generator mintermov
- Z uporabo dekoderjev 3/8 in OR vrat realizirajte funkciji: $f(x, y, z) = x' \cdot y' \cdot z' + x' \cdot y' \cdot z + x' \cdot y \cdot z$ in $g(x,y,z) = x \cdot y' \cdot z' + x \cdot y \cdot z'$



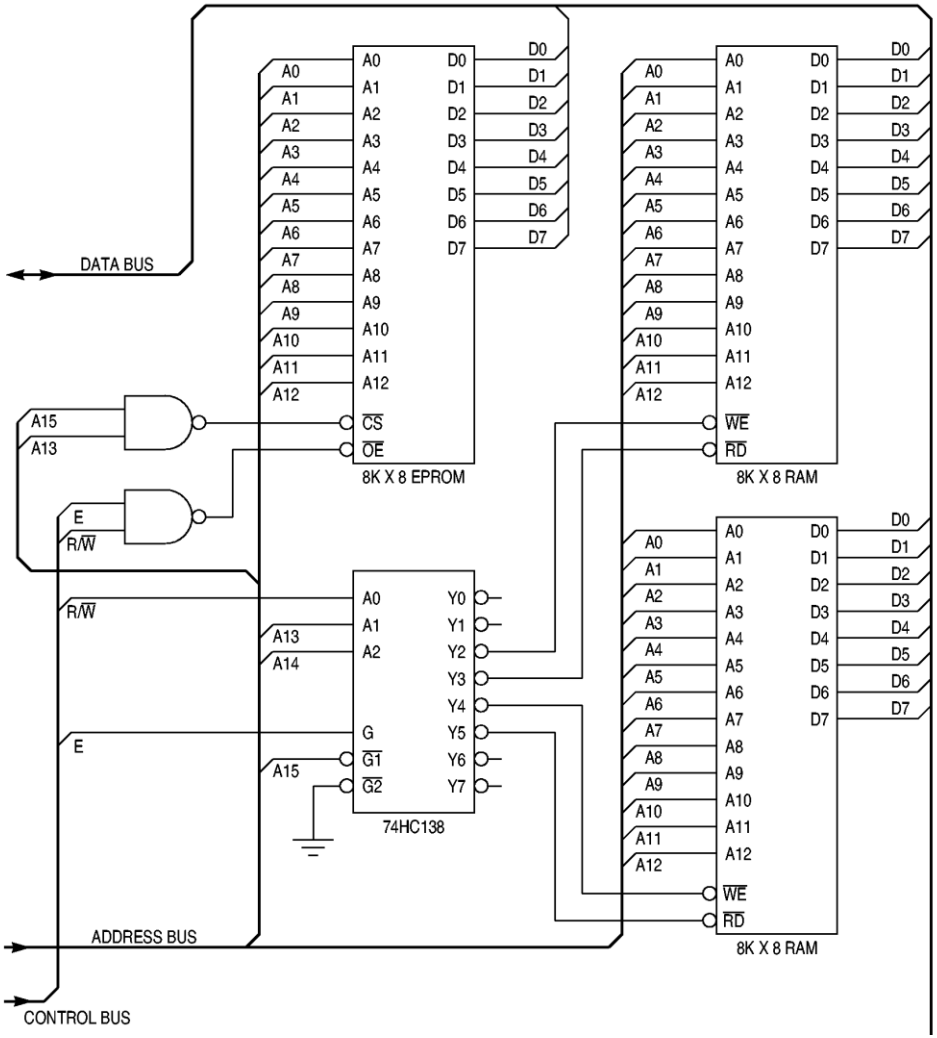


Uporaba dekoderjev

- Dekodiranje naslovnih linij spominskih vezij

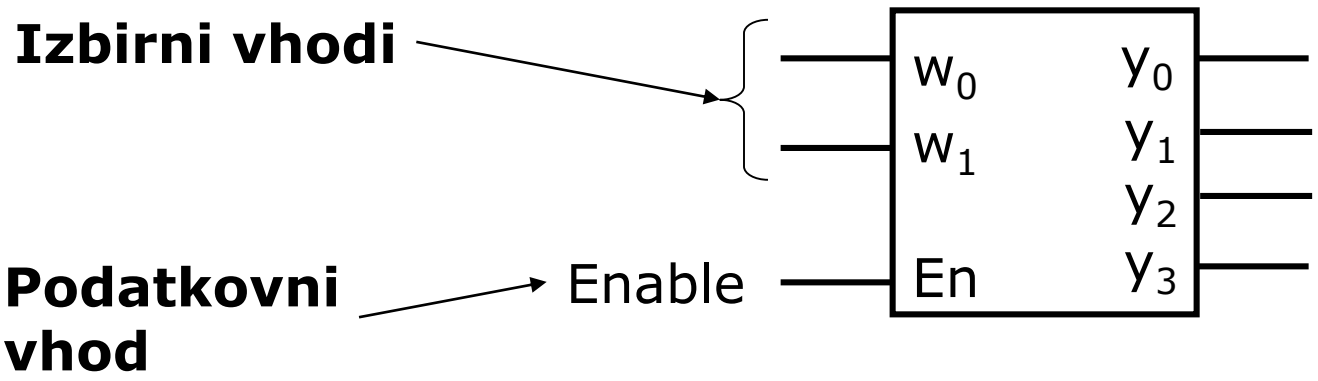


Uporaba dekoderjev: vodilo



Demultiplekser (DMUX)

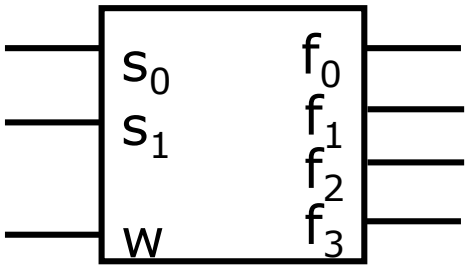
- MUX izbira med *n* podatkovnimi vhodi in enim izhodom. Vezje, ki deluje obratno, je demultiplekser.
- Demultiplekser postavi vrednost enega vhoda na izhod, ki je trenutno izbran.
- Dekoder $n/2^n$ *lahko* realizira $1/n$ demultiplekser, če ima podatkovni vhod (Enable)



DMUX v VHDL

```
ENTITY dec2to4 IS
PORT (      w : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
            En : IN STD_LOGIC;
            y : OUT STD_LOGIC_VECTOR(0 TO 3));
END dec2to4;
ARCHITECTURE arch OF dec2to4 IS
SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
    Enw <= En & w;
    WITH Enw SELECT
            y <=    "1000" WHEN "100",
                    "0100" WHEN "101",
                    "0010" WHEN "110",
                    "0001" WHEN "111",
                    "0000" WHEN OTHERS;
END arch;
```

Parametriziran DMUX




```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;
ENTITY dmuxnto1 IS
    generic( n_addr: natural := 2 );
    PORT ( s : in std_logic_vector(n_addr - 1 downto 0);
           w : in STD_LOGIC;
           f : OUT std_logic_vector(2**n_addr - 1 downto 0)
         );
END dmuxnto1;
ARCHITECTURE rtl OF dmuxnto1 IS
constant zeroes: std_logic_vector(2**n_addr - 1 downto 0) := (others =>'0');
signal  f_sig : std_logic_vector (2**n_addr - 1 downto 0) := (others => '0');
BEGIN
    dmux_process: PROCESS( w, s)
    VARIABLE addr : INTEGER := 0;
    BEGIN
         addr := to_integer(unsigned(s));
         f_sig <= std_logic_vector(to_unsigned( 2**addr, 2**n_addr));

    END PROCESS;
    f <= f_sig when w = '1' else zeroes;
END rtl;
```

Načrtovanje digitalnih vezij

Gradniki kombinacijskih vezij:
Pretvorniki kode

Pretvorniki kode (ang. code converters)

- Pretvorijo kodiranje vhodnih signalov v drug tip kodiranja izhodnih signalov.

- Primera že predstavljenih struktur:
  - 3/8 dekoder:
    dvojiško število pretvori v zaporedno številko izhoda
    imenovano tudi kodiranje "ena naenkrat" oz. "1-od-N"
    (ang. "one hot encoding")

  - 8/3 kodirnik:
    zaporedno številko izhoda pretvori v dvojiško število

BIN ➔ BCD pretvornik kode

00000 → 0 0000	$(0\ 0_{BCD})$	01010 → 1 0000	$(1\ 0_{BCD})$
00001 → 0 0001	$(0\ 1_{BCD})$	01011 → 1 0001	$(1\ 1_{BCD})$
00010 → 0 0010	$(0\ 2_{BCD})$	01100 → 1 0010	$(1\ 2_{BCD})$
00011 → 0 0011	$(0\ 3_{BCD})$	01101 → 1 0011	$(1\ 3_{BCD})$
00100 → 0 0100	$(0\ 4_{BCD})$	01110 → 1 0100	$(1\ 4_{BCD})$
00101 → 0 0101	$(0\ 5_{BCD})$	01111 → 1 0101	$(1\ 5_{BCD})$
00110 → 0 0110	$(0\ 6_{BCD})$	10000 → 1 0110	$(1\ 6_{BCD})$
00111 → 0 0111	$(0\ 7_{BCD})$	10001 → 1 0111	$(1\ 7_{BCD})$
01000 → 0 1000	$(0\ 8_{BCD})$	10010 → 1 1000	$(1\ 8_{BCD})$
01001 → 0 1001	$(0\ 9_{BCD})$	10011 → 1 1001	$(1\ 9_{BCD})$
		10100 →10 0000	$(2\ 0_{BCD})$

Pomik levo eno mesto je množenje z 2

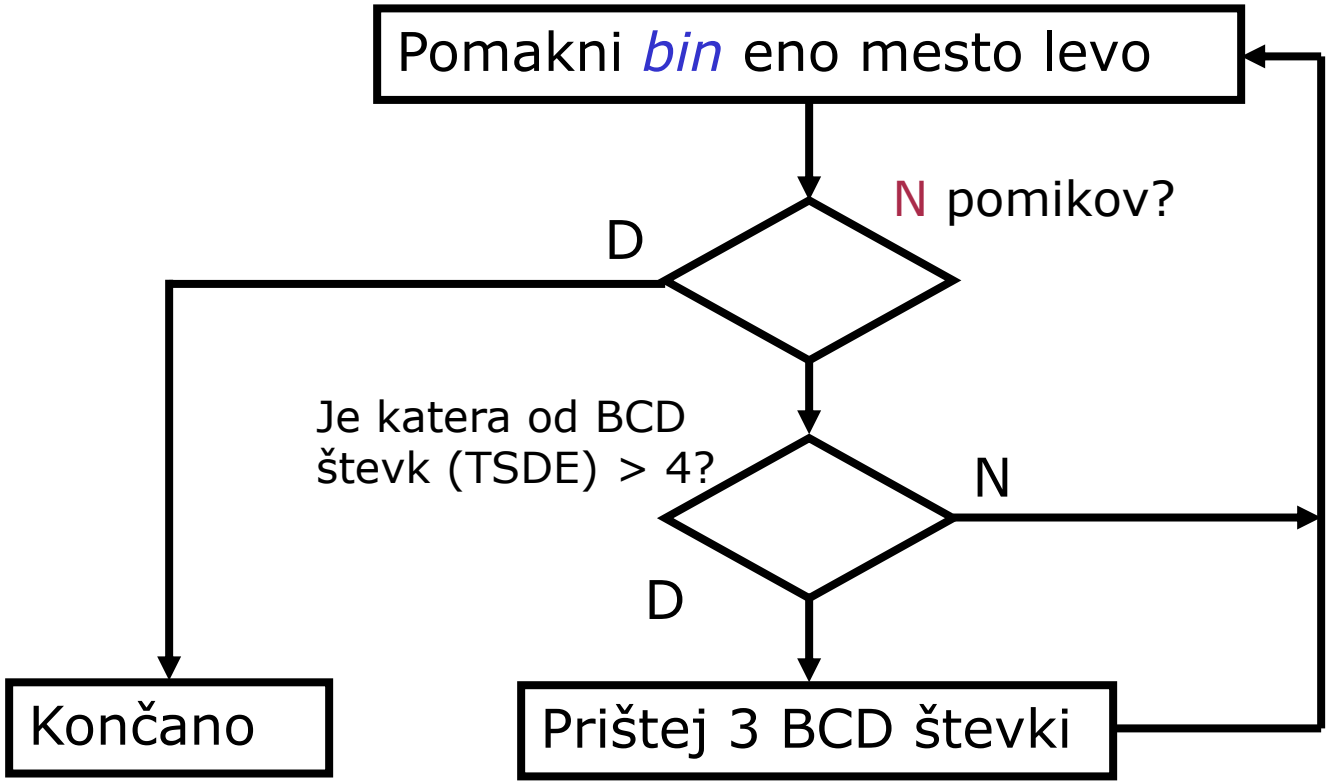
BIN ➔ BCD pretvornik kode

01010 → **1 000**0 (1 0_{BCD})
01011 → **1 000**1 (1 1_{BCD})
01100 → **1 001**0 (1 2_{BCD})
01101 → **1 001**1 (1 3_{BCD})
01110 → **1 010**0 (1 4_{BCD})
01111 → **1 010**1 (1 5_{BCD})
10000 → **1 011**0 (1 6_{BCD})
10001 → **1 011**1 (1 7_{BCD})
10010 → **1 100**0 (1 8_{BCD})
10011 → **1 100**1 (1 9_{BCD})

2^{i+4}	2^{i+3}	2^{i+2}	2^{i+1}	2^{i}					
0	0	0	0	X	0	0	0	0	X
0	0	0	1	X	0	0	0	1	X
0	0	1	0	X	0	0	1	0	X
0	0	1	1	X	0	0	1	1	X
0	1	0	0	X	0	1	0	0	X
0	1	0	1	X	1	0	0	0	X
0	1	1	0	X	1	0	0	1	X
0	1	1	1	X	1	0	1	0	X
1	0	0	0	X	1	0	1	1	X
1	0	0	1	X	1	1	0	0	X

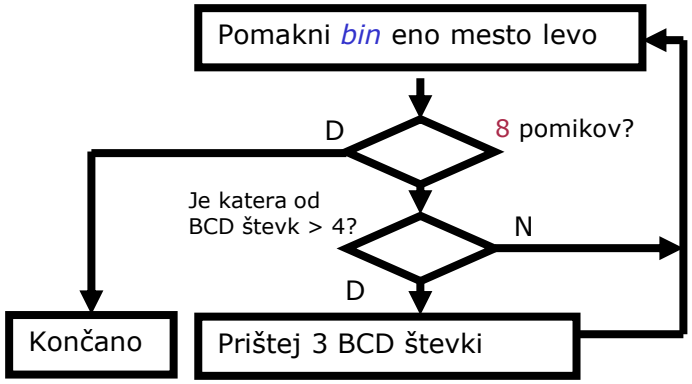
BIN ➔ BCD pretvornik kode

- *"Double dabble"* algoritem za pretvorbo N bitnega števila (*bin*) v BCD zapis (TSDE)



4 bitni BIN ➔ BCD primer 14_{10}

D	E	bin	operacija	opomba
		1110	pomik1	
	1	110	pomik2	<4
	11	10	pomik3	<4
	111	0	+3	>4
	1010	0	pomik4	
1	0100		končano	
1	*4*			



8 bitni BIN ➔ BCD primer 143_{10}

S	D				E					Š	T	E	V	I	L	O	operacija
									1	0	0	0	1	1	1	1	POMIK1
								1	0	0	0	1	1	1	1		POMIK2
							1	0	0	0	1	1	1	1			POMIK3
						1	0	0	0	1	1	1	1				POMIK4
					1	**0**	**0**	**0**	1	1	1	1					**+3**
					1	0	1	1	1	1	1	1					POMIK5
				1	0	1	1	1	1	1	1						+3
				1	**1**	**0**	**1**	**0**	1	1	1						POMIK6
			1	1	0	1	0	1	1	1							+3
			1	1	**1**	**0**	**0**	**0**	1	1							POMIK7
		1	1	1	0	0	0	1	1								+3
	1	**0**	**1**	**0**	0	0	0	1	1								POMIK8
1	0	1	0	0	0	0	1	1									
1			**4**				**3**										

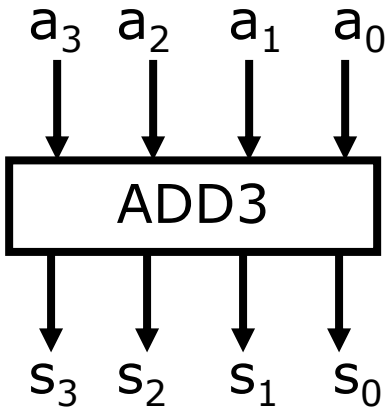
8 bitni BIN ➔ BCD primer 255_{10}

S	D	E	F	F	operacija	opomba
			1111	1111	pomik1	
		1	1111	111	pomik2	
		11	1111	11	pomik3	
		111	1111	1	+3	>4
		1010	1111	1	pomik4	
	1	0101	1111		+3	>4
	1	1000	1111		pomik5	
	11	0001	111		pomik6	
	110	0011	11		+3	>4
	1001	0111	11		pomik7	
1	0010	1010	1		+3	>4
1	0010	0101	1		pomik8	
10	0101	0101			končano	
2	5	5				

11 bitni BIN ➔ BCD primer 999_{10}

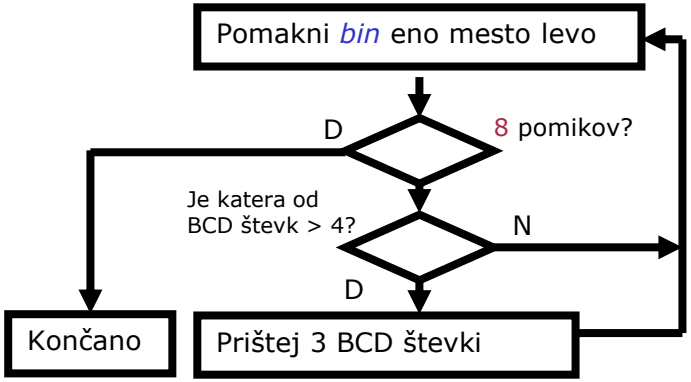
S				D				E				ŠTEVILO											operacija
												0	1	1	1	1	1	0	0	1	1	1	POMIK1
											0	1	1	1	1	1	0	0	1	1	1		POMIK2
										0	1	1	1	1	1	0	0	1	1	1			POMIK3
									0	1	1	1	1	1	0	0	1	1	1				POMIK4
								0	**1**	**1**	**1**	1	1	0	0	1	1	1					**+3**
								1	0	1	0	1	1	0	0	1	1	1					POMIK5
							1	**0**	**1**	**0**	**1**	1	0	0	1	1	1						POMIK6
							1	1	0	0	0	1	0	0	1	1	1						**+3**
						1	1	0	0	0	1	0	0	1	1	1							POMIK7
				0	**1**	**1**	**0**	0	0	1	0	0	1	1	1								**+3**
				1	0	0	1	0	0	1	0	0	1	1	1								POMIK8
			1	0	0	1	0	0	1	0	0	1	1	1									POMIK9
		1	0	0	1	0	0	**1**	**0**	**0**	**1**	1	1										**+3**
		1	0	0	1	0	0	1	1	0	0	1	1										POMIK10
	1	0	0	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	1											**+3**
	1	0	0	1	1	0	0	1	1	0	0	1											POMIK11
1	0	0	1	1	0	0	1	1	0	0	1												
		9				**9**				**9**													

BIN ➔ BCD: blok prištej 3 (ADD3)



a_3	a_2	a_1	a_0	s_3	s_2	s_1	s_0	operacija
0	0	0	0	0	0	0	0	<4
0	0	0	1	0	0	0	1	<4
0	0	1	0	0	0	1	0	<4
0	0	1	1	0	0	1	1	<4
0	1	0	0	0	1	0	0	=4
0	1	0	1	1	0	0	0	>4 (+3)
0	1	1	0	1	0	0	1	>4 (+3)
0	1	1	1	1	0	1	0	>4 (+3)
1	0	0	0	1	0	1	1	>4 (+3)
1	0	0	1	1	1	0	0	>4 (+3)
1	0	1	0	X	X	X	X	ni eno BCD mesto
1	0	1	1	X	X	X	X	ni eno BCD mesto
1	1	0	0	X	X	X	X	ni eno BCD mesto
1	1	0	1	X	X	X	X	ni eno BCD mesto
1	1	1	0	X	X	X	X	ni eno BCD mesto
1	1	1	1	X	X	X	X	ni eno BCD mesto

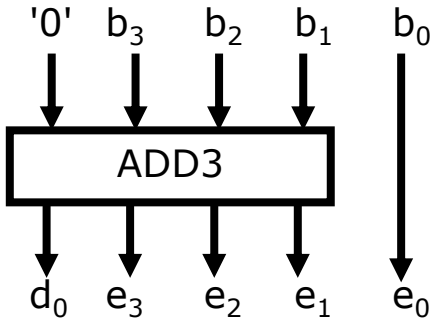
BIN ➔ BCD: realizacija v prostoru



$$B = \sum_{i=0}^{n-1} b_i \cdot 2^i = \left(\left(b_{n-1} \cdot 2 + b_{n-2}\right) \cdot 2 + b_{n-3}\right) \cdot 2 + ... + b_0$$

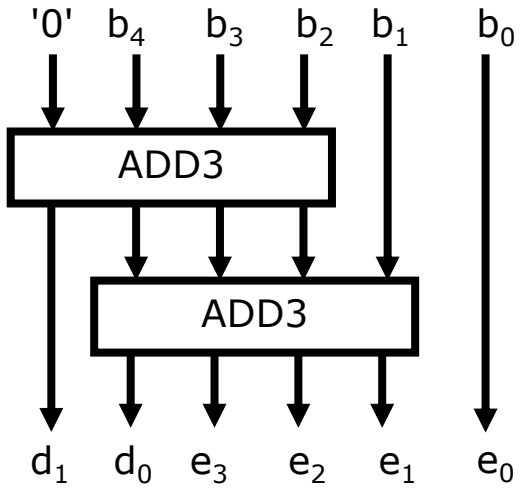
$$B = \left(\left(b_3 \cdot 2 + b_2\right) \cdot 2 + b_1\right) \cdot 2 + b_0$$

BCD		BIN	operacija	opomba
d_0	$e_3e_2e_1e_0$	$b_3b_2b_1b_0$		
		1 1 1 0	pomik1	
	1	1 1 0	pomik2	<4
	1 1	1 0	pomik3	<4
	1 1 1	0	+3	>4
	1 0 1 0	0	pomik4	
1	0 1 0 0		končano	
1	*4*			



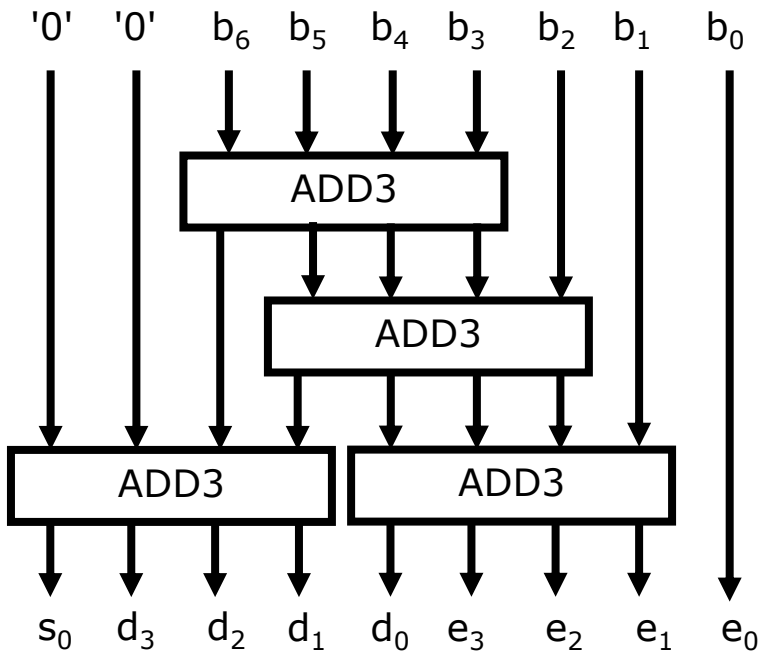
BIN ➔ BCD: realizacija v prostoru

BCD		BIN	operacija	opomba
d_1 d_0	$e_3e_2e_1e_0$	b_4 b_3 b_2 b_1 b_0		
		1 1 1 1 0	pomik1	
	1	1 1 1 0	pomik2	<4
	1 1	1 1 0	pomik3	<4
	1 1 1	1 0	+3	>4
	1 0 1 0	1 0	pomik4	
1	0 1 0 1	0	+3	>4
1	1 0 0 0	0	pomik5	
1 1	*0 0 0 0*		končano	
3	*0*			

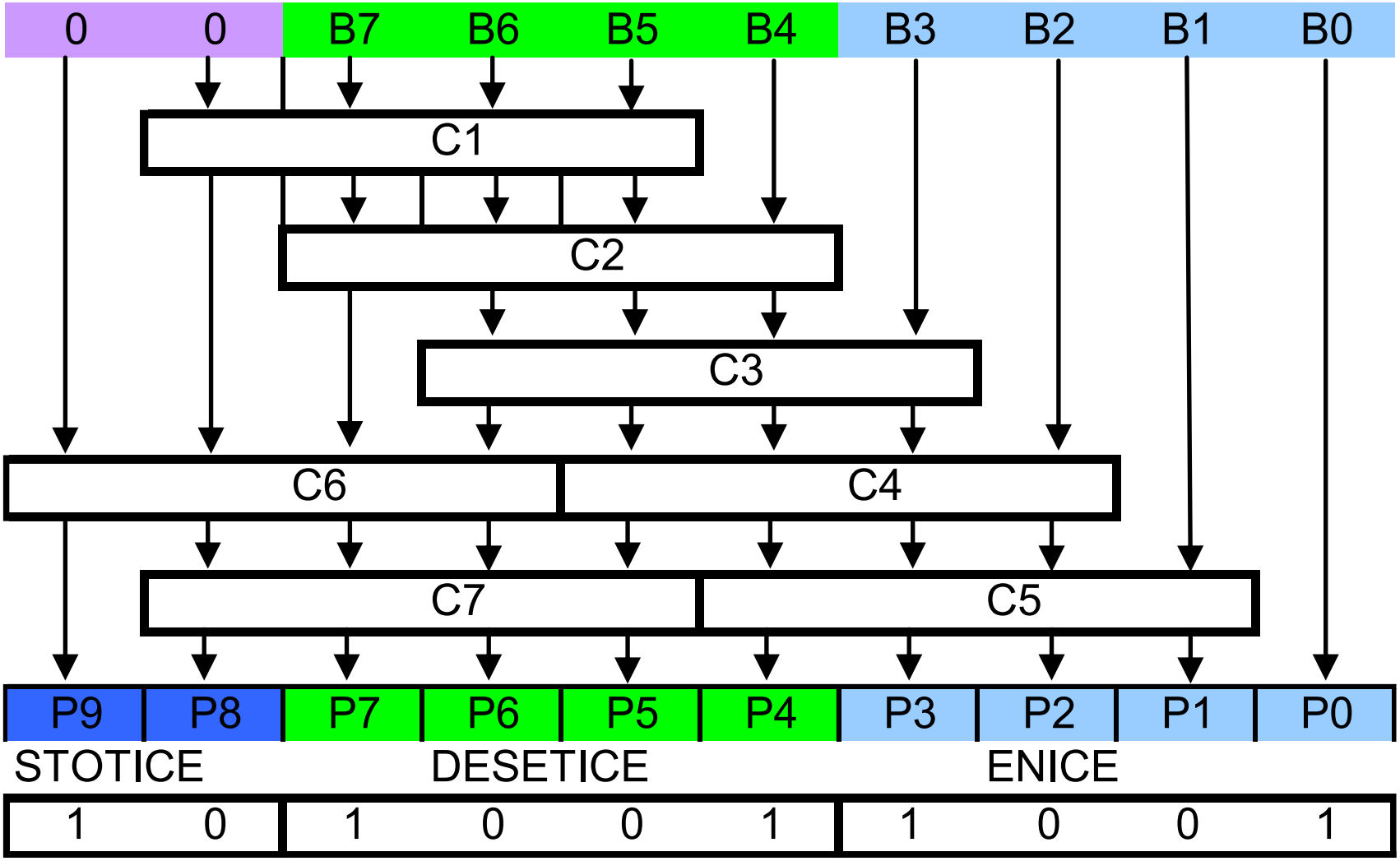


BIN ➔ BCD: realizacija v prostoru

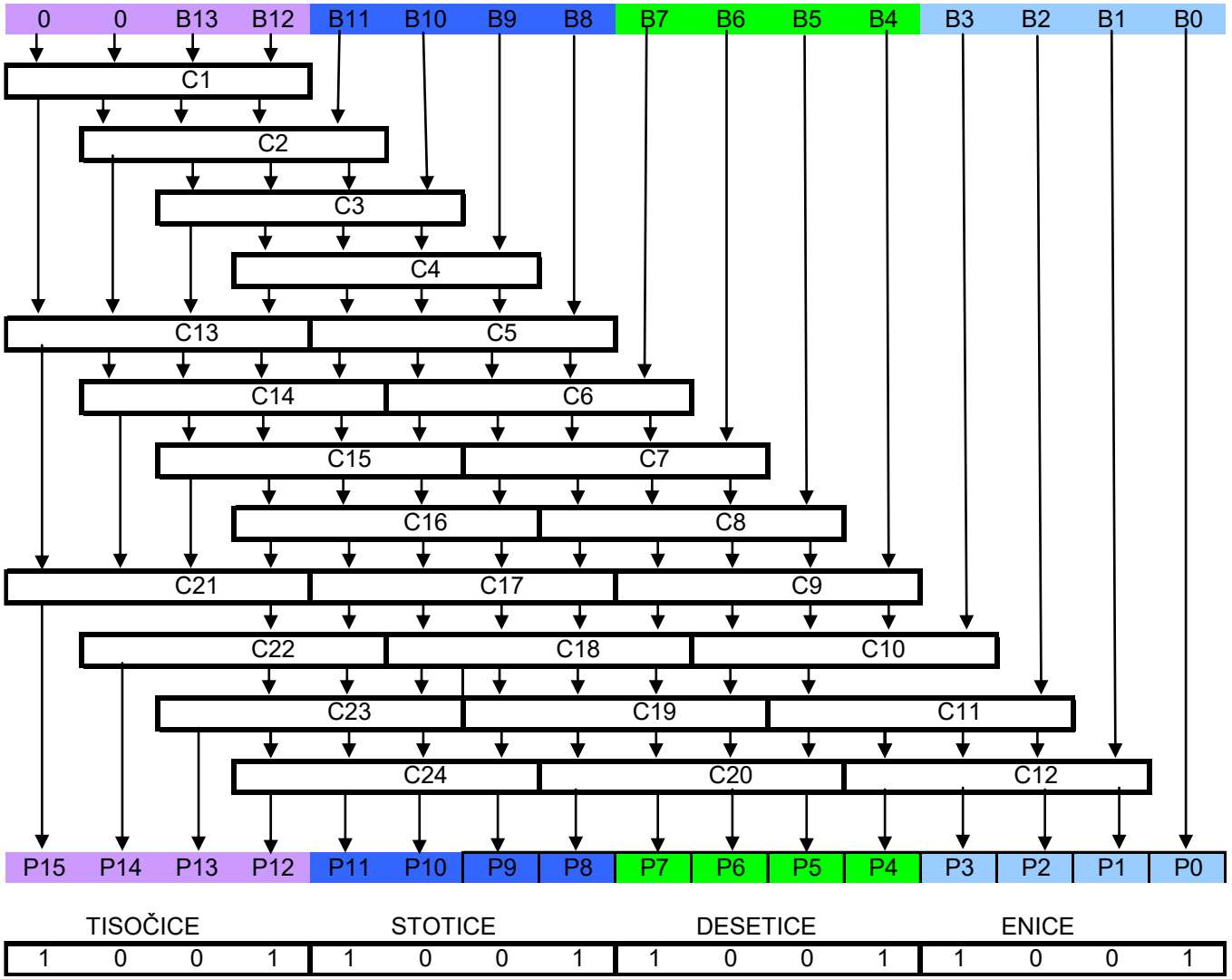
s_0	$d_3d_2d_1d_0$	$e_3e_2e_1e_0$	$b_6b_5b_4b_3b_2b_1b_0$	
			1 1 1 1 1 1 0	pomik1
		1	1 1 1 1 1 0	pomik2
		1 1	1 1 1 1 0	pomik3
		1 1 1	1 1 1 0	+3
		1 0 1 0	1 1 1 0	pomik4
	1	0 1 0 1	1 1 0	+3
	1	1 0 0 0	1 1 0	pomik5
	1 1	0 0 0 1	1 0	pomik6
	1 1 0	0 0 1 1	0	+3
	1 0 0 1	0 0 1 1	0	pomik6
1	*0 0 1 0*	*0 1 1 0*		



8 bit BIN ➔ BCD pretvornik kode



14 bit BIN ➔ BCD pretvornik kode



Razširitve double dabble algoritma

- Osnovni algoritem smo pokazali na primeru pretvorbe vrednosti $0...19_{10}$. Vrednost 19_{10} zahteva 5 bitov v zapisu BCD in 5 bitov v dvojiškem zapisu. Ker je bil LSB enak, je imel blok ADD3 4 bite pogojnega prištevanja. Izhod ADD3 bloka je imel tudi 4 bite.
- Med vsemi možnostmi večbitnih blokov so primerne tiste strukture, ki imajo enako število vhodov in izhodov.
- Zaporedje obsegov pretvorbe večbitnih ADD blokov:
  - 4 bitna → obseg (od $0...\mathbf{0}9_{10}$)
  - 5 bitna → obseg (od $0...\mathbf{1}9_{10}$)
  - 6 bitna → obseg (od $0...\mathbf{3}9_{10}$),
  - 7 bitna → obseg (od $0...\mathbf{7}9_{10}$),
  - 8 bitna → obseg (od $0...\mathbf{15}9_{10}$),
  - 9 bitna → obseg (od $0...\mathbf{31}9_{10}$)
  - 10 bitna → obseg (od $0...\mathbf{63}9_{10}$)
  - 11 bitna → obseg (od $0...\mathbf{127}9_{10}$)
  - 12 bitna → obseg (od $0...\mathbf{255}9_{10}$)

6 bitna BIN2BCD pretvorba ($0..39_{10}$)

00000**0**	00	000**0**	01010**0**	10	000**0**
00000**1**	00	000**1**	01010**1**	10	000**1**
00001**0**	00	001**0**	01011**0**	10	001**0**
00001**1**	00	001**1**	01011**1**	10	001**1**
00010**0**	00	010**0**	01100**0**	10	010**0**
00010**1**	00	010**1**	01100**1**	10	010**1**
00011**0**	00	011**0**	01101**0**	10	011**0**
00011**1**	00	011**1**	01101**1**	10	011**1**
00100**0**	00	100**0**	01110**0**	10	100**0**
00100**1**	00	100**1**	01110**1**	10	100**1**
00101**0**	01	000**0**	01111**0**	11	000**0**
00101**1**	01	000**1**	01111**1**	11	000**1**
00110**0**	01	001**0**	10000**0**	11	001**0**
00110**1**	01	001**1**	10000**1**	11	001**1**
00111**0**	01	010**0**	10001**0**	11	010**0**
00111**1**	01	010**1**	10001**1**	11	010**1**
01000**0**	01	011**0**	10010**0**	11	011**0**
01000**1**	01	011**1**	10010**1**	11	011**1**
01001**0**	01	100**0**	10011**0**	11	100**0**
01001**1**	01	100**1**	10011**1**	11	100**1**

Blok pogojnega prištevanja za 6 bitno BIN2BCD pretvorbo ($0..39_{10}$) – brez LSB

00000	00	000	01010	10	000
00000	00	000	01010	10	000
00001	00	001	01011	10	001
00001	00	001	01011	10	001
00010	00	010	01100	10	010
00010	00	010	01100	10	010
00011	00	011	01101	10	011
00011	00	011	01101	10	011
00100	00	100	01110	10	100
00100	00	100	01110	10	100
00101	01	000	01111	11	000
00101	01	000	01111	11	000
00110	01	001	10000	11	001
00110	01	001	10000	11	001
00111	01	010	10001	11	010
00111	01	010	10001	11	010
01000	01	011	10010	11	011
01000	01	011	10010	11	011
01001	01	100	10011	11	100
01001	01	100	10011	11	100

Blok pogojnega prištevanja za 6 bitno BIN2BCD pretvorbo ($0..39_{10}$)

0	00000	00000	0	20	01010	10000	6
1	00000	00000	0	21	01010	10000	6
2	00001	00001	0	22	01011	10001	6
3	00001	00001	0	23	01011	10001	6
4	00010	00010	0	24	01100	10010	6
5	00010	00010	0	25	01100	10010	6
6	00011	00011	0	26	01101	10011	6
7	00011	00011	0	27	01101	10011	6
8	00100	00100	0	28	01110	10100	6
9	00100	00100	0	29	01110	10100	6
10	00101	01000	3	30	01111	11000	9
11	00101	01000	3	31	01111	11000	9
12	00110	01001	3	32	10000	11001	9
13	00110	01001	3	33	10000	11001	9
14	00111	01010	3	34	10001	11010	9
15	00111	01010	3	35	10001	11010	9
16	01000	01011	3	36	10010	11011	9
17	01000	01011	3	37	10010	11011	9
18	01001	01100	3	38	10011	11100	9
19	01001	01100	3	39	10011	11100	9

ADD3

ADD6

ADD9

Blok pogojnega prištevanja za 6 bitno BIN2BCD pretvorbo ($0..39_{10}$)

```
ENTITY add369 IS
PORT (   x : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
         y : OUT STD_LOGIC_VECTOR(5 DOWNTO 0));
END add369;
ARCHITECTURE arch OF add369 IS
BEGIN
   WITH x(5 DOWNTO 1); SELECT
         y <=  x WHEN (x <= 4),
               (x+3) WHEN (x >= 5) and (x <= 9),
               (x+6) WHEN (x >= 10) and (x <= 14),
               (x+9) WHEN (x >= 15) and (x <= 19),
               "000000" WHEN OTHERS;
   y(0) <= x(0);
END arch;
```

Načrtovanje digitalnih vezij

Gradniki kombinacijskih vezij:
Predstavitev paketnih datotek in konfiguracij
na primeru načrtovanja parametriziranega
BIN→BCD pretvornika

Paketne datoteke v VHDL

- V paketnih datotekah zaobjamemo elemente, ki si jih medsebojno (globalno) deli več entitet.
- V paketni datoteki lahko tudi hranimo podatke, ki si jih deli več entitet.
- Paketno datoteko sestavljata dva dela:
  - deklaracija (ang. package declaration)
  - telo (ang. package body)

Paketna deklaracija

- Paketna deklaracija lahko vsebuje deklaracije različnih elementov:
  - Podprogramov
  - Tipov, podtipov
  - Konstant
  - Komponent
  - Lastnosti (ang. attribute)
  - Datotek

Deklaracija

```
package primer_paketne is
   type t1 is (z0,z1,z2,r0,r1,r2,f0,f1,f2);
   type t2 is array (0 to 15) of t1;
   type t3 is array (natural range <>) of t2;
   function f1 (s : t3) return t2;
   subtype t4 is resolve_cluster t3;
   constant c0 : t4
End primer_paketne;
```

Telo

- Telo paketne datoteke lahko vsebuje deklaracije različnih elementov :
  - Podprogramov
  - Tipov, podtipov
  - Konstant
  - Lastnosti (ang. attribute)
  - Stavkov USE
  - Psevdonimov (ang. alias)
  - Datotek

Telo

```
package body primer_paketne is
constant c7 : t_wclus:= (zx,zx);
function f1 (s: t3) return t2 is
variable return_result : t2;
variable x: integer;
begin
   if s'length = 0 then
        return c0;
   end if;
   for i in s'range loop
        if s(i) /= c0 then
                x := x - 1;
                if x = 1 then
                        return_result := a(i);
                else
                        return_result : = c7
                end if;
        end if;
   end loop
   return return_result; -- izhodna vrednost
end f1;                  -- konec funkcije
end primer_paketne;      -- konec paketne
```

Podprogrami v paketnih datotekah

- Podprogrami so lahko funkcije ali procedure
- Obnašajo se podobno kot procesni stavki, s tem da stavki IF, CASE, LOOP, WAIT v njih niso dovoljeni.
- Bistvena razlika med procesnim stavkom in podprogrami je v nivoju uporabe:
  - Procesni stavki se izvajajo v glavni kodi entitete in se morajo dati sintetizirati.
  - Podprogrami vsebujejo ponavljajoče se dele kode, ki si jih lahko deli več entitet. Koda podprogramov se ne sintetizira, ampak se vrnjene spremenljivke obravnavajo kot konstante v nadrejeni (klicoči) entiteti.
- Podprograme sicer lahko pišemo neposredno v glavno kodo, vendar je taka rešitev zelo nepregledna in krši načelo delitve podatkov med entitetami.

Procedure in funkcije

- Procedura lahko vrne več argumentov
- Procedura ima lahko vhodne (in) parametre, izhodne (out) parametre in vhodno-izhodne (inout) parametre.

- Funkcija vedno vrne samo en parameter
- Vsi ostali parametri so vhodni (in)
- Funkcije služijo npr. pretvorbi tipov podatkov, kompleksnejšim logičnim operacijam, aritmetičnim operacijam, deklaracijam novih operatorjev in lastnosti (ang. attribute)
- Deklaracija signalov in novih komponent v funkcijah ni dovoljena.

Funkcije v paketni datoteki - deklaracija

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.math_real.all;
package bin2bcd_pkg is
   function nr_bcd_digits (bin_bits : integer) return integer;
   function conv_bcd_to_bin (bcd_nr : integer; bin_nr_size :
   integer) return integer;
end bin2bcd_pkg;
```

Funkcije v paketni datoteki - telo

```
package body bin2bcd_pkg is
--    @Function name: nr_bcd_digits
-- @Parameters:
--    a: binary number size
--    @Return:
--    Number of bcd digits required to encode a maximum binary number of size a bits
function nr_bcd_digits (bin_bits : integer) RETURN integer IS
     VARIABLE max_bin_nr   : integer := 2**bin_bits - 1;
begin
     return ( integer(floor(log10(real(max_bin_nr)))) + 1 );
end nr_bcd_digits;
--    @Function name: conv_bcd_to_bin
-- @Parameters:
--    bcd_nr: stevilo v bcd zapisu
--    bin_nr_size: velikost dvojiskega stevila
--    @Return:
--    BCD number pretvori v dvojisko predstavitev s pomocjo zaporednega deljenja s 16. Ta funkcija se ne da sintetizirati.
function conv_bcd_to_bin (bcd_nr : integer; bin_nr_size : integer) RETURN integer IS
     CONSTANT size_of_bcd_nr : integer := nr_bcd_digits(bin_nr_size); --izracunaj st. potrebnih bitov za hranjenje bcd stevk
     variable return_val : integer range 0 to 2**bin_nr_size - 1 := 0;
     variable temp : integer := 0;
     variable i : integer range 0 to size_of_bcd_nr - 1;
     variable bcd_digit : unsigned (3 downto 0) := to_unsigned(0, 4);-- iteracija bcd stevke na 0
begin
     return_val := 0; -- vmesni bcd rezultat na 0
     bcd_digit := to_unsigned(0, 4);
     i := 0; --stevec potenc postavimo na enice
     temp := bcd_nr; -- postavi zacetno bcd vrednost v spremenljivko bcd pretvorbe
     while (temp /= 0) loop              -- j tece po bitih bcd stevila navzgor
             bcd_digit := to_unsigned(temp rem 16, bcd_digit'length); -- pridobi vmesno bcd stevko
             return_val := return_val + to_integer(bcd_digit) * 10**i;
             temp := temp / 16;          -- prehod na naslednjo bcd stevko s celostevilskim deljenjem
             i := i + 1;
     end loop;
     return ( return_val );
end conv_bcd_to_bin;
end bin2bcd_pkg;
```

Parametriziran BIN→BCD pretvornik

```
-- Primer: Ce je bin_nr 12 bitna, algoritem traja 12 iteracij.
-- Vsaka iteracija je 16 bitna (4 bcd stevke: TSDE).
-- Primer pretvorbe (0x3ff=1023):
--  i/bcd_sig       T         S         D         E                 bin_nr

-- 0               0000      0000      0000      0000      0011 1111 1111
-- 1               0000      0000      0000      0000      0111 1111 1110
-- 2               0000      0000      0000      0000      1111 1111 1100
-- 3               0000      0000      0000      0001      1111 1111 1000
-- 4               0000      0000      0000      0011      1111 1111 0000
-- 5               0000      0000      0000      0111      1111 1110 0000
-- 5add            0000      0000      0000      1010      1111 1110 0000
-- 6               0000      0000      0001      0101      1111 1100 0000
-- 6add            0000      0000      0001      1000      1111 1100 0000
-- 7               0000      0000      0011      0001      1111 1000 0000
-- 8               0000      0000      0110      0011      1111 0000 0000
-- 8add            0000      0000      1001      0011      1111 0000 0000
-- 9               0000      0001      0010      0111      1110 0000 0000
-- 9add            0000      0001      0010      1010      1110 0000 0000
-- 10              0000      0010      0101      0101      1100 0000 0000
-- 10add           0000      0010      1000      1000      1100 0000 0000
-- 11              0000      0101      0001      0001      1000 0000 0000
-- 11add           0000      1000      0001      0001      1000 0000 0000
-- 12              0001      0000      0010      0011      0000 0000 0000

--bcd              1         0         2         3         decimalni rezultat
```

bcd_sig je 2D polje std_logic_vector

Parametriziran BIN→BCD pretvornik

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
use work.bin2bcd_pkg.all;
entity bin2bcd is
generic (
   bin_nr_size : integer := 8
--stevilo bitov vhodnega dvojiskega stevila (najmanj 2)
);
port (
bin_nr : in STD_LOGIC_VECTOR (bin_nr_size - 1 downto 0);
bcd : out STD_LOGIC_VECTOR((nr_bcd_digits(bin_nr_size) *
   4) - 1 downto 0) -- bcd stevke
);
end bin2bcd;
```

vključitev paketne datoteke

uporaba funkcije
iz paketne datoteke

Parametriziran BIN→BCD pretvornik

```
architecture nested_loops of bin2bcd is

bcd_proc: process(bin_nr)
constant nr_bcd_bits: integer := nr_bcd_digits(bin_nr_size) * 4;
-- stevilo bitov BCD rezultata izracunano iz najv. dvojiskega stevila
variable bin_temp : STD_LOGIC_VECTOR (bin_nr'range);
-- zacasna spremenljivka
variable bcd_digit : unsigned (3 downto 0) := to_unsigned(0, 4);
-- iteracija bcd stevke na 0
variable j : integer range 0 to nr_bcd_bits;
--tekoci indeks po bitih bcd rezultata
type bcd_sig_type is array (2 to bin_nr_size) of STD_LOGIC_VECTOR
   (bcd'range);
-- polje iteracij bcd stevk
-- ena iteracija vec za shranjevanje rezultata
variable bcd_sig : bcd_sig_type := (others => (others => '0'));
-- polje vseh iteracij bcd stevk na 0
   begin
```

deklaracija spremenljivke
2D polja std_logic_vector
in postavljanje vseh elementov na '0'

Parametriziran BIN→BCD pretvornik

```
-- ce v zacetni iteraciji naredimo pomik dve mesti levo
-- do pogojnega pristevanja ne more priti (ne more biti vec kot 4),
-- cetudi sta oba bita bin_temp(MSB downto MSB-1) enaka '1'
bin_temp := bin_nr(bin_nr'left - 2 downto 0) & "00";
-- shrani pomaknjen vhod v zacasno spremenljivko
bcd_sig(2) := (nr_bcd_bits - 1 downto 2 => '0') & bin_nr(bin_nr'left downto bin_nr'left - 1);
-- pomik bitov bin_temp[MSB:MSB-1] -> bcd_sig[LSB+1:LSB] - preskocimo prvi dve iteraciji
for i in 2 to bin_nr_size - 1 loop
-- i tece po bitih dvojiskega stevila vse do predzadnje stopnje
    j := 0; -- postavi indeks spremenljivke bcd_sig na 0
   while (j < nr_bcd_bits) loop -- j tece po stevkah (korak 4) rezultata bcd_sig
          bcd_digit := unsigned(bcd_sig(i)( j + 3 downto j));
          -- pogojno pristej 3 in vodi rezultat na naslednjo stopnjo (mux in sestevalnik)
          if bcd_digit > 4 then
                    bcd_sig(i)( j + 3 downto j) := std_logic_vector(bcd_digit + 3);
          else
                    bcd_sig(i)( j + 3 downto j) := std_logic_vector(bcd_digit);
          end if;
          j := j + 4;          -- preskoci na naslednjo bcd stevko v signalu bcd_sig
    end loop;
    bcd_sig(i+1) := bcd_sig(i)(nr_bcd_bits - 2 downto 0) & bin_temp(bin_temp'left);
    -- pomik bcd levo, MSB(bin_temp) -> LSB(bcd)
    bin_temp := bin_temp(bin_temp'left - 1 downto 0) & '0'; -- pomik bin_temp levo
end loop;
bcd <= bcd_sig(bin_nr_size);  -- rezultat se nahaja na zadnji stopnji pretvorbe
end process bcd_proc;
end nested_loops;
```

Parametrizirana datoteka testnih vrednosti

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use IEEE.numeric_std.all;
USE work.bin2bcd_pkg.all;
ENTITY bin2bcd_tb IS
generic ( bin_nr_size : integer := 14 ); --stevilo bitov vhodnega dvojiskega stevila (najmanj
    2));
END bin2bcd_tb;

ARCHITECTURE arch OF bin2bcd_tb IS
COMPONENT bin2bcd is
GENERIC ( bin_nr_size : integer := 12 --stevilo bitov vhodnega dvojiskega stevila (najmanj 1)
);
port (    bin_nr : in  STD_LOGIC_VECTOR (bin_nr_size - 1 downto 0);
    bcd : out STD_LOGIC_VECTOR((nr_bcd_digits(bin_nr_size) * 4) - 1 downto 0) -- bcd stevke
);
end COMPONENT;

signal bin_nr : std_logic_vector(bin_nr_size-1 downto 0) := (others => '0');
signal bcd : std_logic_vector((nr_bcd_digits(bin_nr_size) * 4) - 1 downto 0) := (others =>
    '0');

for all: bin2bcd use entity work.bin2bcd(nested_loops);
```

S for.all use stavkom eksplicitno navedemo uporabo določene arhitekture (če jih je več).
To lahko spravimo v datoteko nastavitev projekta (ang. configuration)

Parametrizirana datoteka testnih vrednosti

```
begin
    -- Povezava UUT - Unit Under Test (enote, ki jo testiramo)
   uut: bin2bcd     generic map ( bin_nr_size => bin_nr_size)
                    PORT MAP ( bin_nr => bin_nr, bcd => bcd);
stim_proc: process
    variable i, test: integer range 0 to 2**bin_nr_size - 1;
    --tekoci indeks i po bitih bcd rezultata in testna spremenljivka
begin
    i := 0;
    while (i < 2**bin_nr_size) loop -- j tece po vseh kombinacijah binarnega stevila
          bin_nr <= std_logic_vector(to_unsigned(i, bin_nr'length));
          -- preslikaj celo stevilo i v std_logic_vector bin_nr ustrezne dolzine
          wait for 10 ns;     -- pocakati je potrebno najmanj "max. combinatorial delay" (ca.
    6.396 ns @ Artix 7 FPGA)
          test := conv_bcd_to_bin(to_integer(unsigned(bcd)), bin_nr_size);
          --pretvori bcd v dvojisko predstavitev z uporabo zaporednega deljenja s 16
          if (i /= test) then
                assert false
                report "Napaka pretvorbe: " & CR & " double_dabble = " &
                integer'image(test) & " decimalno, i: " &
                integer'image(i);
          end if;
          i := i + 1; -- naslednja iteracija vhoda
    end loop;
    wait;
end process;
end;
```

Stavka assert in report v datotekah testnih vrednosti

- Stavka **assert** in **report** se v datotekah testnih vrednosti uporabljata za preverjanje pravilnosti izvajanja simulacije.
- Besedo "assert" v tem kontekstu prevedemo kot zahteva oz. trditev.
- Uporabljamo ju za:
  - preverjanje pravilnosti načrtovanja (npr. produkt negativnih števil je pozitivno število) in
  - preverjanje pravilnosti vhodnih vrednosti signalov ter napak na signalih (npr. če se dva signala postavita na '1', pa se ne bi smela).
- Če preverjamo, ali je signal a vedno manjši ali kvečjemu enak signalu b. Če to NI RES (false) simulator vrne kršitev zahteve (ang. assert violation):

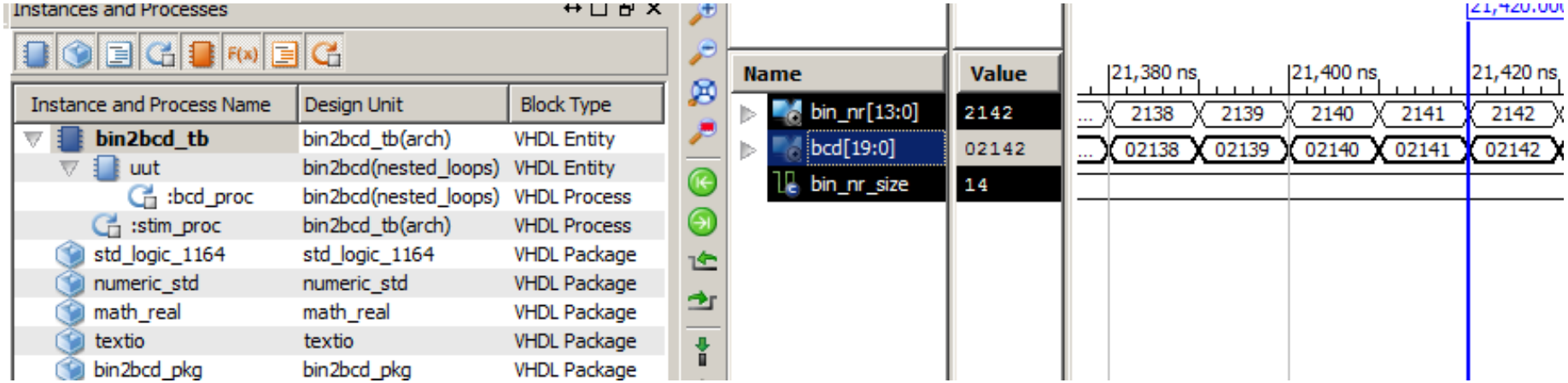
```
assert a.value <= b.value report "a.value prevelik";
```

- Če preverjamo, ali sta signala a in b naenkrat enaka '1':

```
assert a.sig='1' nand b.sig='1'
report "a.sig and b.sig sta oba 1!";
```

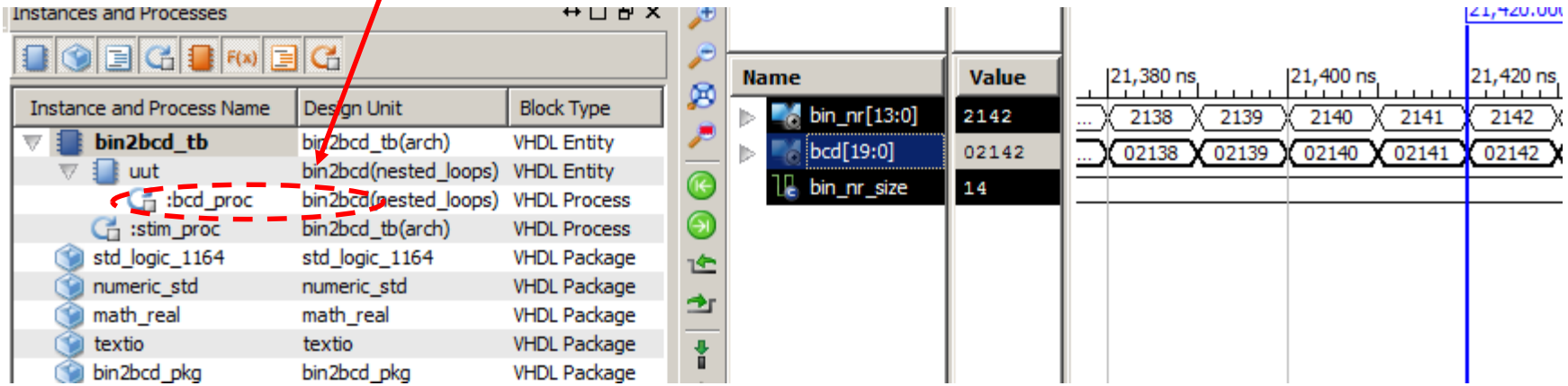
Proces razhroščevanja v Xilinx ISE 14.7

- Pogled na simulacijsko okno.
- Zavihek console

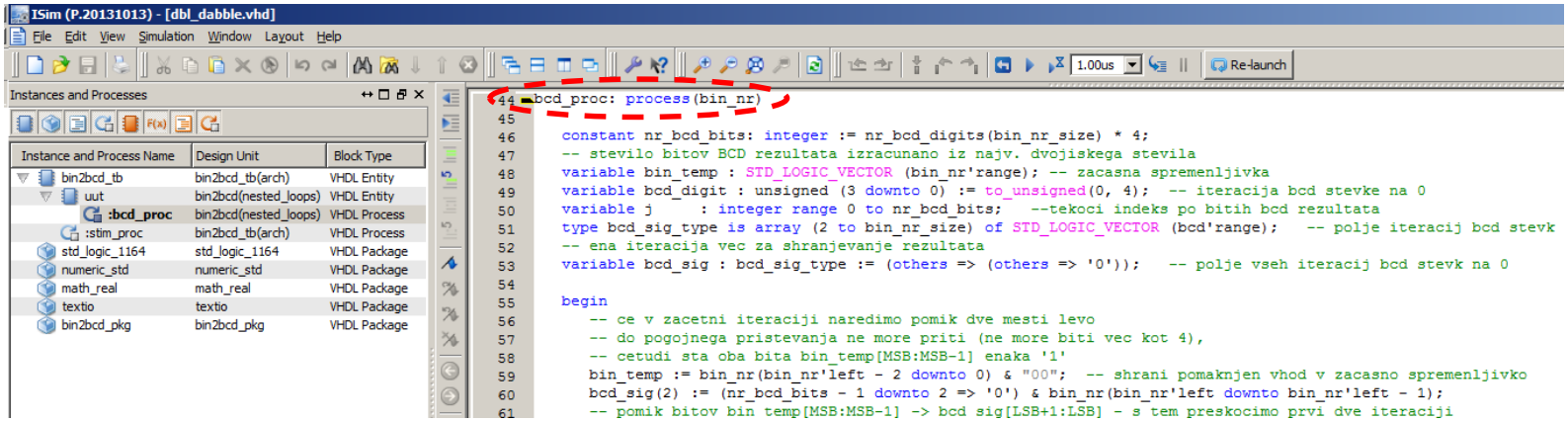


Postavljanje prekinitve

Z dvoklikom izberemo procesni stavek, v katerem želimo postaviti prekinitev.

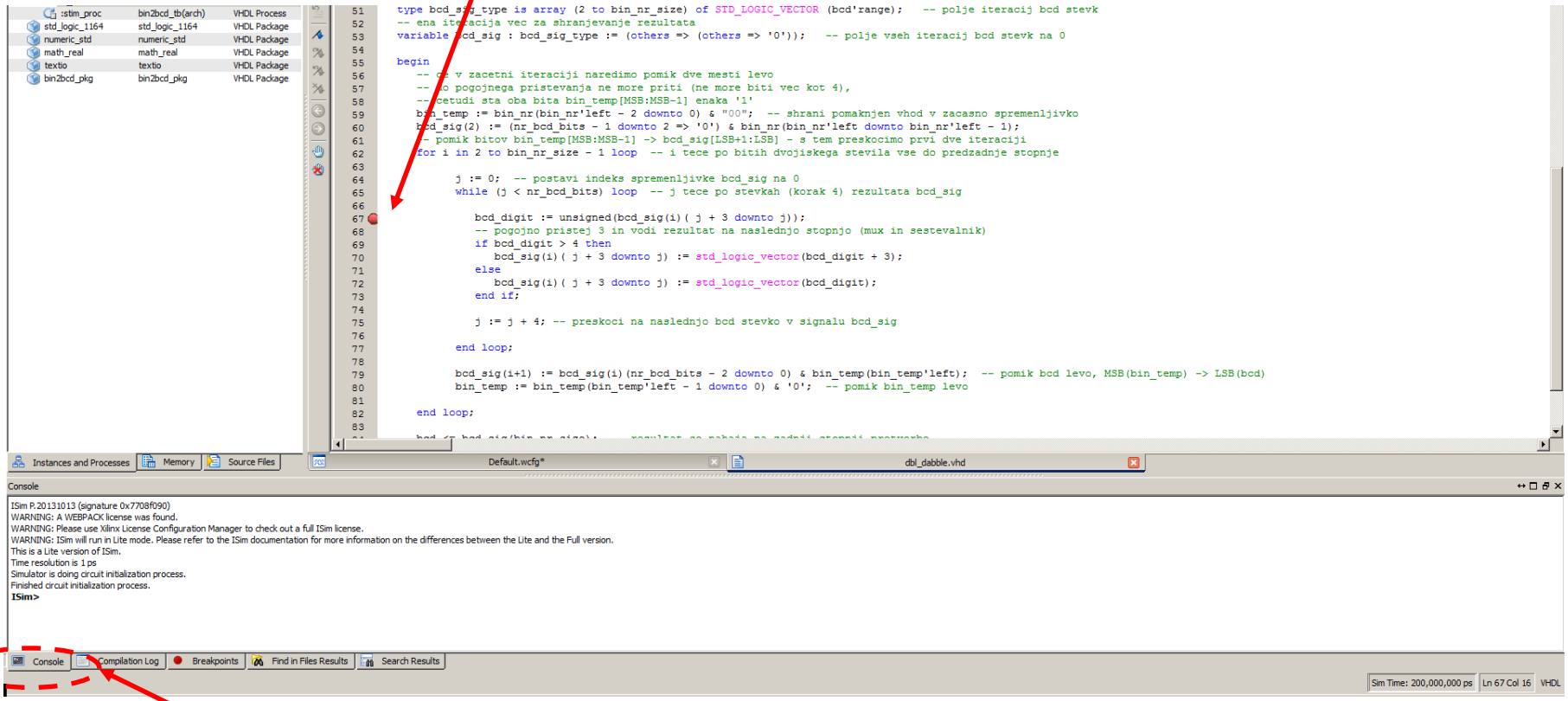


Simulator odpre ustrezno vhd datoteko in nas postavi na začetek izbrane lokacije



Postavljanje prekinitve

Postavimo kurzor na vrstico programa in pritisnemo F9. Pojavi se rdeča pika, ki označuje mesto simulacijske prekinitve



Ostale ukaze pišemo v obliki skripte ali kot zaporedje ukazov v konzoli Isim (Console)

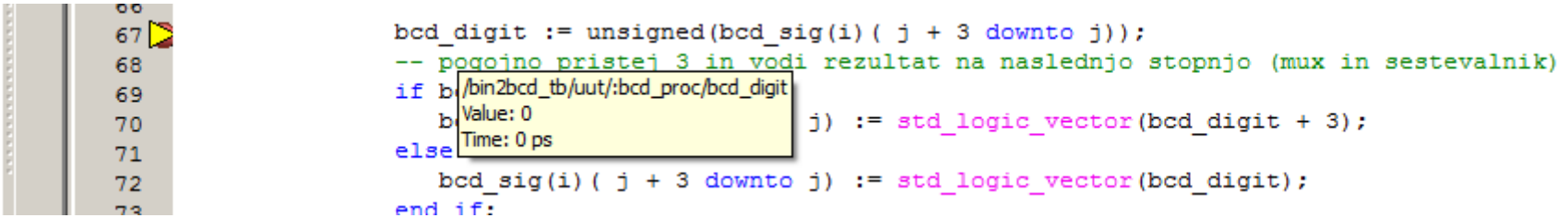
Proces razhroščevanja v Xilinx ISE 14.7

- V konzoli Isim najprej vzpostavimo začetne razmere (čas na 0, postavitev začetnih vrednosti simulacije).
- To storimo tako, da napišemo v Isim konzolo ukaz: restart
- Simulacijo lahko poženemo z F5 ali z ukazom v Isim konzoli: run all
- Simulacija se izvaja, dokler ne naleti na konec predvidenega časa simulacije ali prekinitev simulacije (ang. breakpoint).
- Ko se ustavi, v konzoli Isim javi podobno sporočilo :

```
Simulator is doing circuit initialization process.
Stopped at time : 0 fs : File "dbl_dabble.vhd" Line 67
```

Proces razhroščevanja v Xilinx ISE 14.7

- Ko je simulacija ustavljena zaradi prekinitve lahko pregledujemo vrednosti spremenljivk:
  - Grafično – tako da se postavimo z mišjo nad spremenljivko in se izpiše njena vrednost v plavajočem okencu



  - V konzoli Isim – s spodnjim ukazom Isim sporoči vrednost v zahtevani bazi (ang. radix). Če lastnost –radix izpustimo, je privzeta baza desetiška:

```
ISim> show value bcd_digit -radix hex
```

- Vrednosti spremenljivk (ang. variable) lahko opazujemo izključno v konzoli – grafično ne!

Proces razhroščevanja v Xilinx ISE 14.7

- Ko smo s pregledovanjem vrednosti spremenljivk in signalov zaključili, lahko simulacijo:
  - *Poženemo* do naslednje simulacijske prekinitve (ang. breakpoint) s pritiskom na **F5**.
  - *Poženemo* za določen čas – v konzoli Isim vpišemo run 100 ns.
  - *Preskočimo/korakamo* ukaz za ukazom (ang. step) z **F11**.

Proces razhroščevanja v Xilinx ISE 14.7

- Če želimo v danem trenutku simulacije izpisati vse spremenljivke, to storimo z ukazom: **dump**
- Isim na **dump** odgovori s podobnim sporočilom:

```
ISim> dump
Objects in /bin2bcd_tb/uut
Generic: {bin_nr_size} : 14
Port(IN): {bin_nr[13:0]} : 0
Port(OUT): {bcd[19:0]} : x
Objects in /bin2bcd_tb/uut/:bcd_proc
Constant: {nr_bcd_bits} : 20
Variable: {bcd_digit[3:0]} : 0
Variable: {bcd_sig[2:14][19:0]} : 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0
Variable: {bin_temp[13:0]} : 0
Variable: {j} : 0
```

- Ostali TCL ukazi Isim so povzeti v: Isim user guide (UG660).

Veljavni ukazi Isim

- **bp** < list | <add <file_name> <line_number>> | <remove <file_name> <line_number>> | <del <index_list>> | clear >
- **cls**
- **describe** <object_name>
- **divider add** <name> [-into <id>] [-color <color>]
- **dump**
- **group add** <name> [-into <id>]
- **help** [<command_name>]
- **hwcosim** < set <property> <value> > | < get <property> >
- **init**
- **isim** < < <ptrace|ltrace> <on|off> > | < set <userunit|arraydisplaylength|maxtraceablesize|radix> <value> > | < get <userunit|arraydisplaylength|maxtraceablesize|radix> > | < condition <list | <add <options>> | <remove <options>> > | < force < <add <options>> | <remove <options>> > >
- **marker add** <time>
- **onerror** {<script>}
- **put** <hdl_object_name> <value> [-radix <type>]
- **quit** [-f | -s]
- **restart**
- **resume**
- **run** [all | continue | <<time> [<unit>]>]
- **saif** < <open [-scope <path_name>] [-file <file_name>] [-level <nesting_level>] [-allnets] > | close >
- **scope** [.. | <path_name>]
- **sdfanno** <-min|-typ|-max> <file_name> [-nowarn] [-noerror] [-root <path_name>]
- **show** < time | port | scope | signal | variable | constant | <child [-r]> | <<driver|load|value> <hdl_object_name>> >
- **specify** < <pathpulse <int> [<int>]> | <pulsestyle <onevent|ondetect>> | showcancelled | noshowcancelled >
- **step**
- **stop**
- **test** <hdl_object_name> <value> [-radix <type>]
- **vcd** < <dumpfile <file_name>> | dumpflush | dumpon | dumpoff | <dumpvars -m <path_name> [-l <level>]> | <dumplimit <size>> >
- **virtualbus add** <name> [-into <id>] [-reverse] [-radix <radix>] [-color <color>]
- **wave** <add [options] [<object_name>...]> | <log [options] [<object_name>...]>
- **wcfg** new | open <fileName> | save <fileName> | select <fileName>

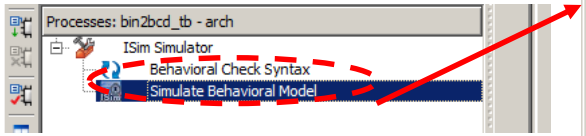
TCL ukazi Isim

- S TCL ukazi se da izvajati marsikaj – npr. da signal prisilimo (ang. force) v dano stanje za nek časovni interval.
- Signal clk postavimo na 1 v trenutku ustavitve simulacije (t_0). Signal clk gre nazaj na ‘0’ čez 20 ns in ponavlja tak cikel (‘0’ → ‘1’→’0’...) na vsakih 40 ns dokler ne mine 1 us od t_0:

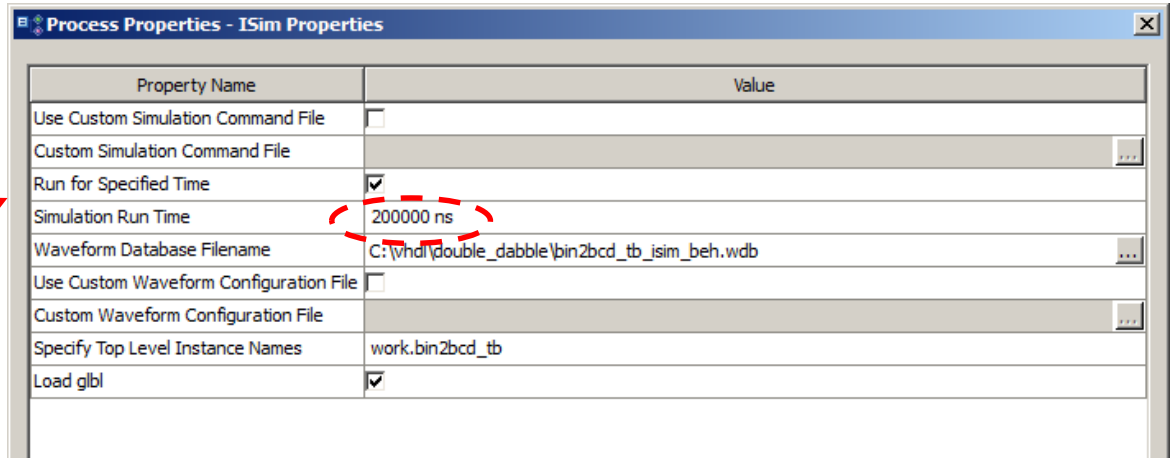
```
isim force clk 1 -value 0 -time 20 ns -repeat
40 ns -cancel 1 us
```

Nastavljanje trajanja simulacije

- V Project Navigatorju nastavimo "Simulation"
- Postavimo se na "Simulate behavioral model"
- Kliknemo desni gumb
- Izberemo "Process properties.



- Odpre se okno "Process properties", kjer nastavljamo "Simulation Run Time" v časovnih enotah VHDL.
- Privzete enote so ns.



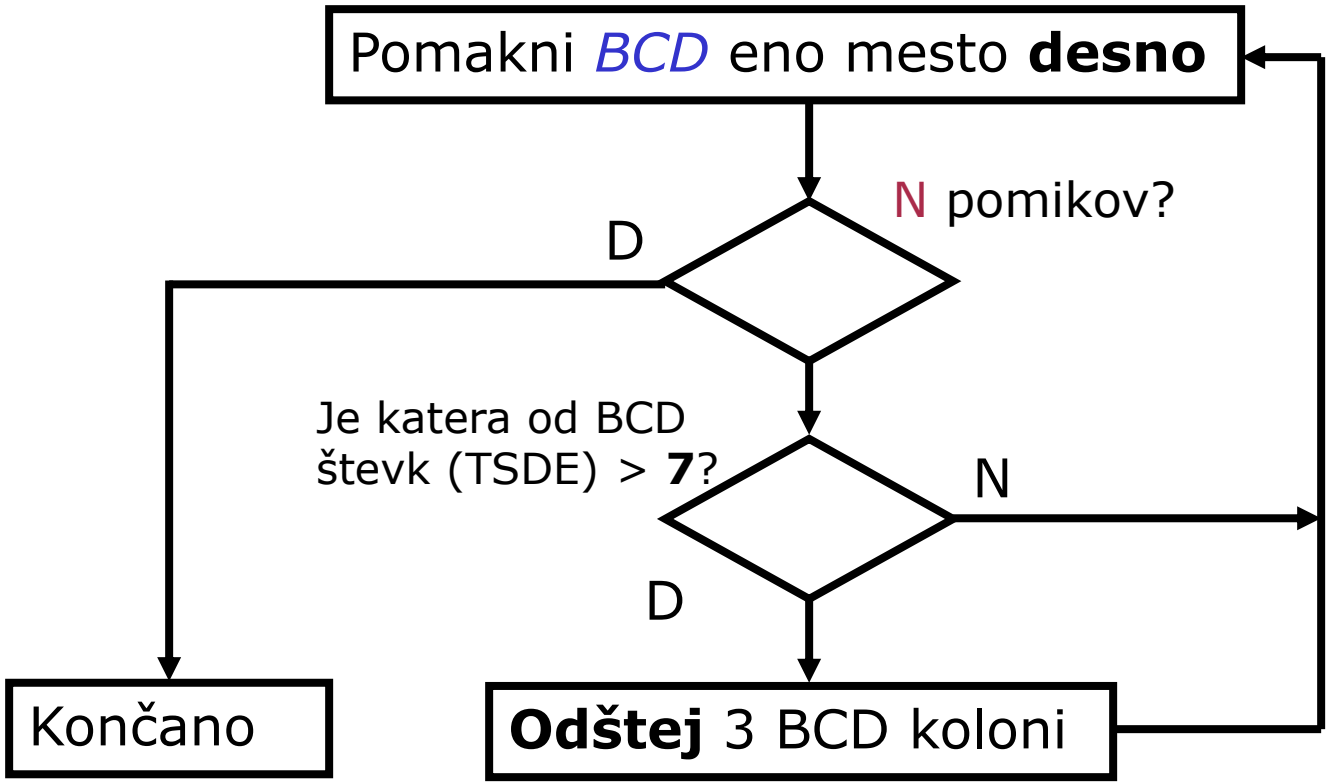
BCD→BIN pretvornik kode

- 4-bitni prekodirnik odšteje 3 od števil, ki so večja ali enaka 8.
- Če je BCD števka >7 odštej 3
- Večje (ali enako) od binarne 8 pomeni večje (ali enako) od 10_{BCD}

$(1\ 0_{BCD})$**1 000**0 → **0101**0
$(1\ 1_{BCD})$**1 000**1 → **0101**1
$(1\ 2_{BCD})$**1 001**0 → **0110**0
$(1\ 3_{BCD})$**1 001**1 → **0110**1
$(1\ 4_{BCD})$**1 010**0 → **0111**0
$(1\ 5_{BCD})$**1 010**1 → **0111**1
$(1\ 6_{BCD})$**1 011**0 → **1000**0
$(1\ 7_{BCD})$**1 011**1 → **1000**1
$(1\ 8_{BCD})$**1 100**0 → **1001**0
$(1\ 9_{BCD})$**1 100**1 → **1001**1

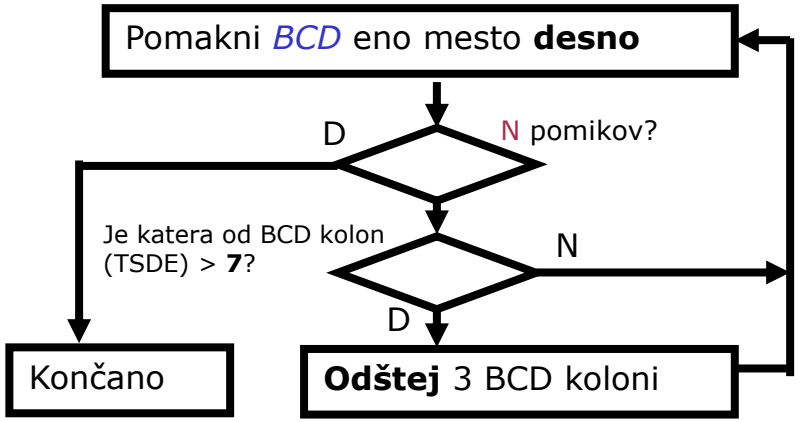
BCD➔BIN pretvornik kode

- Algoritem za pretvorbo BCD števila (TSDE) v N bitno dvojiško število (*bin*)



4 bit BCD ➔ BIN primer 14_{10}

D	E	bin	operacija	opomba
1	0100		pomik1	<7
	1010	0	-3	>7
	0111	0	pomik2	<7
	011	10	pomik3	<7
	01	110	pomik4	<7
	0	1110	končano	



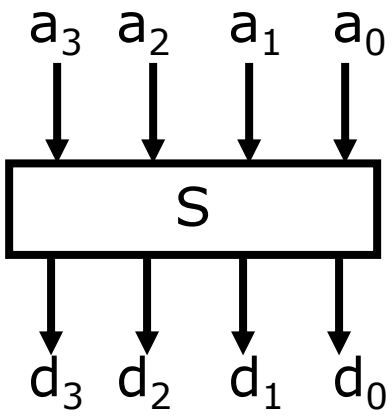
8 bitni BCD ➔ BIN primer $8F_{16}$

S				D				E				ŠTEVILO								operacija
0	0	0	1	0	1	0	0	0	0	1	1									POMIK1
	0	0	0	**1**	**0**	**1**	**0**	0	0	0	1	1								-3
	0	0	0	0	1	1	1	0	0	0	1	1								POMIK2
		0	0	0	0	1	1	**1**	**0**	**0**	**0**	1	1							-3
		0	0	0	0	1	1	0	1	0	1	1	1							POMIK3
			0	0	0	0	1	**1**	**0**	**1**	**0**	1	1	1						-3
			0	0	0	0	1	0	1	1	1	1	1	1						POMIK4
				0	0	0	0	**1**	**0**	**1**	**1**	1	1	1	1					-3
				0	0	0	0	1	0	0	0	1	1	1	1					POMIK5
					0	0	0	0	1	0	0	0	1	1	1	1				POMIK6
						0	0	0	0	1	0	0	0	1	1	1	1			POMIK7
							0	0	0	0	1	0	0	0	1	1	1	1		POMIK8
								0	0	0	0	**1**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	$8F_{16}$

11 bitni BCD ➔ BIN primer $3E7_{16}$

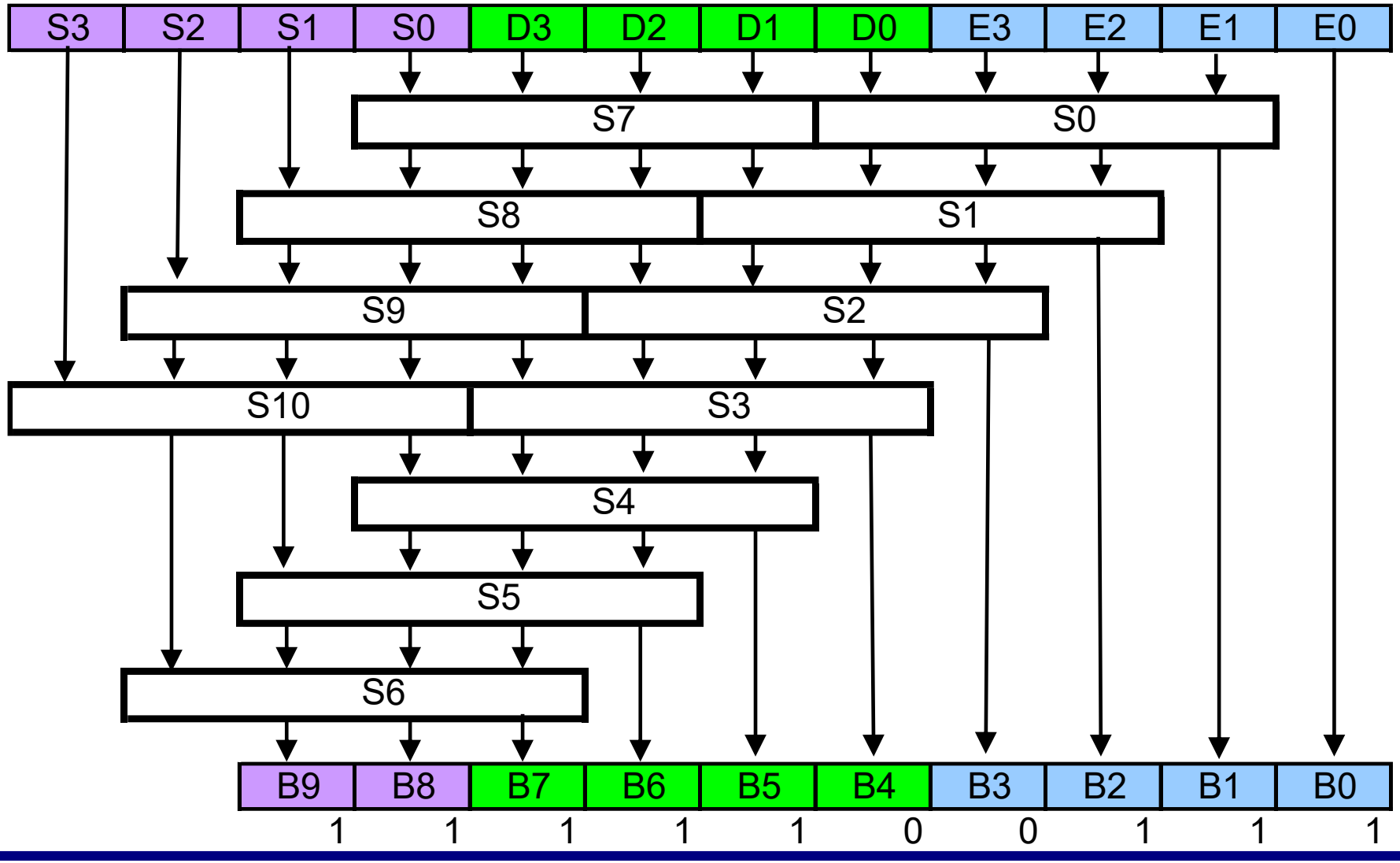
S				D				E				ŠTEVILO										operacija
1	0	0	1	1	0	0	1	1	0	0	1											POMIK1
	1	0	0	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	1										**-3**
	1	0	0	1	0	0	1	1	0	0	1	1										POMIK2
		1	0	0	1	0	0	**1**	**1**	**0**	**0**	1	1									**-3**
		1	0	0	1	0	0	1	0	0	1	1	1									POMIK3
			1	0	0	1	0	0	1	0	0	1	1	1								POMIK4
				1	**0**	**0**	**1**	0	0	1	0	0	1	1	1							**-3**
				0	1	1	0	0	0	1	0	0	1	1	1							POMIK5
					0	1	1	0	0	0	1	0	0	1	1	1						POMIK6
						0	1	**1**	**0**	**0**	**0**	1	0	0	1	1	1					**-3**
						0	1	0	1	0	1	1	0	0	1	1	1					POMIK7
							0	**1**	**0**	**1**	**0**	1	1	0	0	1	1	1				**-3**
							0	0	1	1	1	1	1	0	0	1	1	1				POMIK8
								0	0	1	1	1	1	1	0	0	1	1	1			POMIK9
									0	0	1	1	1	1	1	0	0	1	1	1		POMIK10
										0	0	1	1	1	1	1	0	0	1	1	1	POMIK11
											0	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	$3E7_{16}$

BCD ➔ BIN : blok odštej 3 (SUB3)



a_3	a_2	a_1	a_0	d_3	d_2	d_1	d_0	operacija
0	0	0	0	0	0	0	0	<7
0	0	0	1	0	0	0	1	<7
0	0	1	0	0	0	1	0	<7
0	0	1	1	0	0	1	1	<7
0	1	0	0	0	1	0	0	<7
0	1	0	1	0	1	0	1	<7
0	1	1	0	0	1	1	0	<7
0	1	1	1	0	1	1	1	=7
1	0	0	0	0	1	0	1	>7 (-3)
1	0	0	1	0	1	1	0	>7 (-3)
1	0	1	0	0	1	1	1	>7 (-3)
1	0	1	1	1	0	0	0	>7 (-3)
1	1	0	0	1	0	0	1	>7 (-3)
1	1	0	1	X	X	X	X	ni možno!
1	1	1	0	X	X	X	X	ni možno!
1	1	1	1	X	X	X	X	ni možno!

BCD ➔ BIN : shema pretvorbe 999_{10}



Načrtovanje digitalnih vezij

Predstavitev števil in
aritmetična vezja:

Gray-eva koda

Gray-eva koda

00 0 0	000 0 00	0000 0 000
01 0 1	001 0 01	0001 0 001
10 1 1	010 0 11	0010 0 011
11 1 0	011 0 10	0011 0 010
	100 1 10	0100 0 110
	101 1 11	0101 0 111
	110 1 01	0110 0 101
	111 1 00	0111 0 100
		1000 1 100
		1001 1 101
		1010 1 111
		1011 1 110
		1100 1 010
		1101 1 011
		1110 1 001
		1111 1 000

n_2 ➔ Gray-eva koda

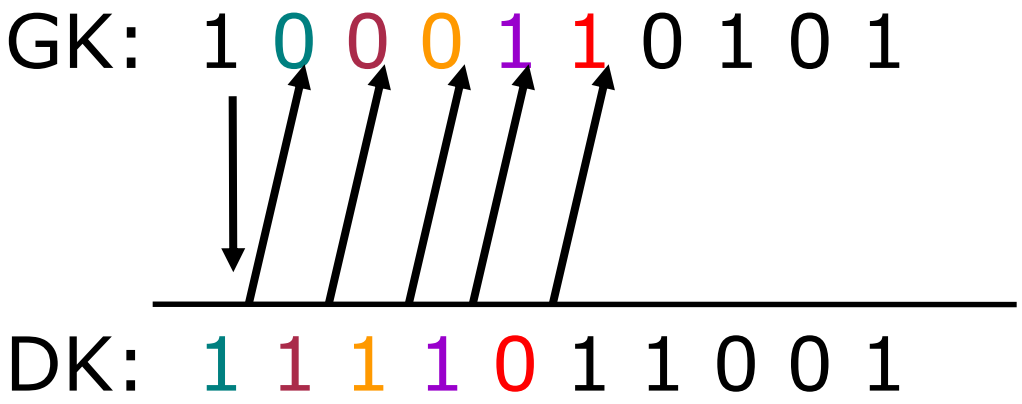
DK: 0 1 1 0 1 0 0 1 1 0

\+ 0 1 1 0 1 0 0 1 1

GK: 0 1 0 1 1 1 0 1 0 1

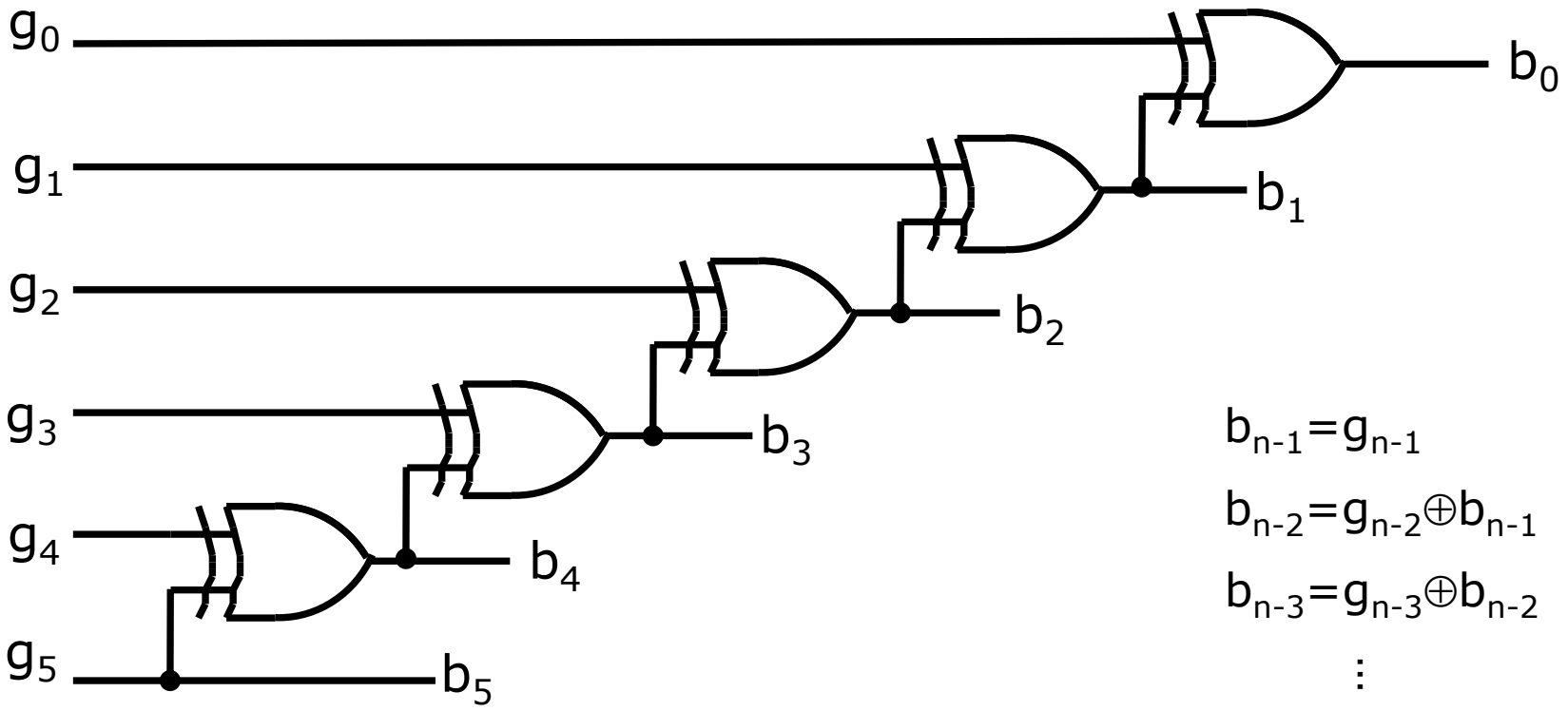
- Dvojiško kodo za eno mesto zamaknemo eno mesto desno
- Dobljeno vrednost prištejemo brez prenosov (XOR med bitoma):
  - Vsoti 0+1=1 in 1+0=1 ➔ 1 v GK
  - Vsoti 0+0=0 in 1+1=0 ➔ 0 v GK.

Gray-eva koda ➔ n_2



- Prepišemo MSB bit Grayeve kode na prvo mesto dvojiške kode,
- XOR z naslednjim bitom Grayeve kode,
- dobimo drugi bit dvojiške kode,
- dobljeni bit prištejemo naslednjemu bitu Grayeve kode,
- postopek ponavljamo do zadnjega bita Grayeve kode.

Strojna izvedba pretvornika $n_2 \leftarrow$ GK



$b_{n-1}=g_{n-1}$

$b_{n-2}=g_{n-2} \oplus b_{n-1}$

$b_{n-3}=g_{n-3} \oplus b_{n-2}$

$\vdots$

Strojna izvedba pretvornika n_2 ➔ GK

$b_{n-1}=g_{n-1}$

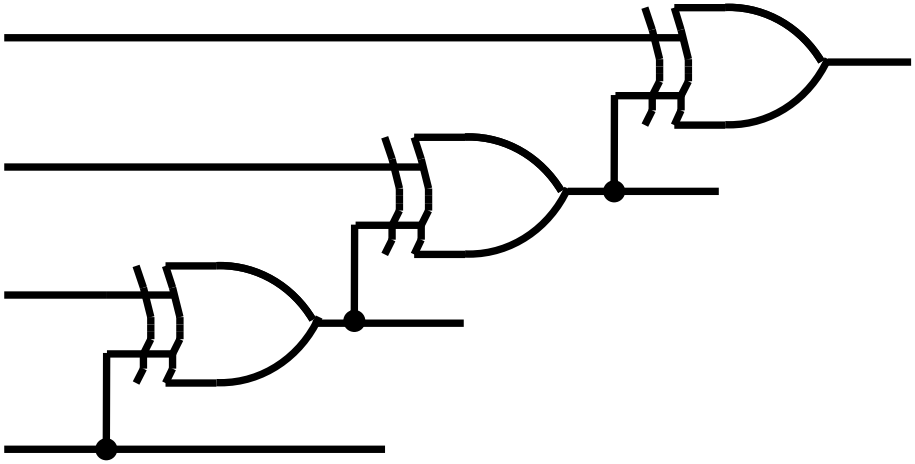
$b_{n-2}=g_{n-2}\oplus g_{n-1}$

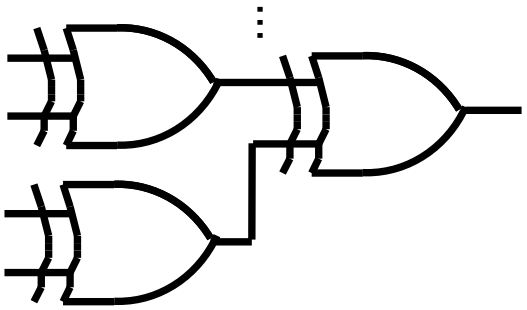
$b_{n-3}=g_{n-3}\oplus g_{n-2}\oplus g_{n-1}$

$b_{n-4}=g_{n-4}\oplus g_{n-3}\oplus g_{n-2}\oplus g_{n-1}$

Lastnost združevanja
(Boole-ova logika)

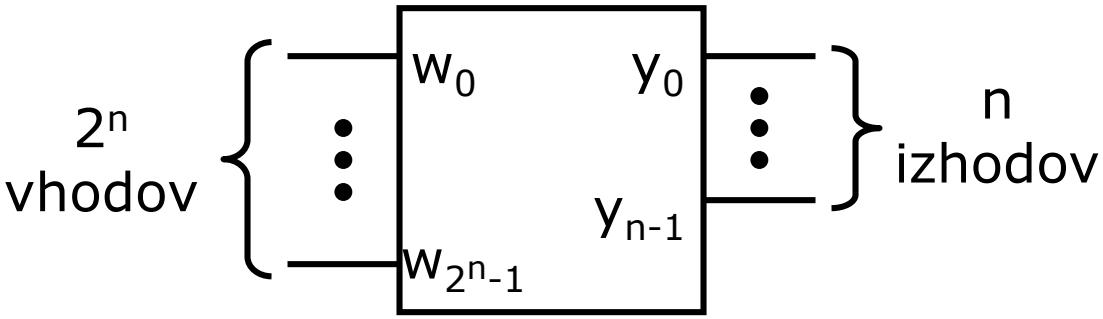
$=(g_{n-4}\oplus g_{n-3})\oplus(g_{n-2}\oplus g_{n-1})$





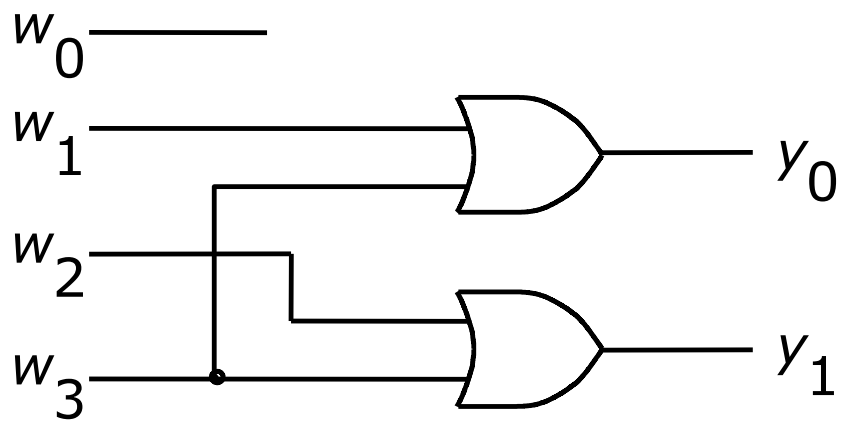
XOR vrata so samo 2-vhodna v TTL ➔ sestavimo drevesno strukturo.
(manjša zakasnitev, manj nivojev)

Kodirniki (ang. encoder)



$2^n/n$ binarni kodirnik

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



To bi delovalo, če bi bil aktiven *vedno samo en vhod*.
Sicer pri w=0110 dobimo y=11, česar ne želimo. Zato imamo prioriteto.

Kodirnik prioritete (Priority encoder)

- Naj ima w_0 najnižjo prioriteto in w_3 najvišjo
- Izhod ***z*** določa kdaj noben od vhodov ni 1
- Zapišemo
  - $i_0 = w_3' \cdot w_2' \cdot w_1' \cdot w_0$
  - $i_1 = w_3' \cdot w_2' \cdot w_1$
  - $i_2 = w_3' \cdot w_2$
  - $i_3 = w_3$

4/2 kodirnik prioritete

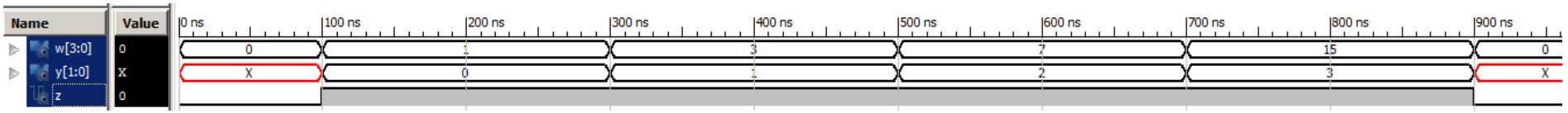
w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

$y_0 = i_1 + i_3$, $y_1 = i_2 + i_3$

$z = i_1 + i_2 + i_3 + i_4$

Kodirnik prioritete (when-else)

```
ENTITY priority IS
PORT (w : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
      y : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
      z : OUT STD_LOGIC);
END priority;
ARCHITECTURE arch OF priority IS
BEGIN
   y <="11" WHEN w(3) = '1' ELSE
       "10" WHEN w(2) = '1' ELSE
       "01" WHEN w(1) = '1' ELSE "00";

   z <='0' WHEN w = "0000" ELSE '1';
END arch;
```



Kodirnik prioritete (when others)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
ENTITY priority IS
PORT ( w : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
       y : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
       z : OUT STD_LOGIC);
END priority;
ARCHITECTURE arch_when_when_others OF priority IS
BEGIN
       WITH w SELECT
       y <=   "11" WHEN "1---",
              "10" WHEN "01--",
              "01" WHEN "001-",
              "00" WHEN "0001",
              "XX" WHEN OTHERS;
        -- redundanc (-) se tu ne da sintetizirati!
       z <='0' WHEN w = "0000" ELSE '1';
END arch_when_when_others;
```

Kodirnik prioritete v VHDL (when others)

- Intuitivno bi kodirnik prioritete radi zapisali z uporabo redundanc (-) in when others VHDL stavka.
- Koda je sicer sintaktično pravilna, vendar se ne da sintetizirati.
- Redundance sintetizator razume kot vrednost tipa std_logic, ki pa je žal ne more sintetizirati kot bi npr. ‘0’ ali ‘1’.

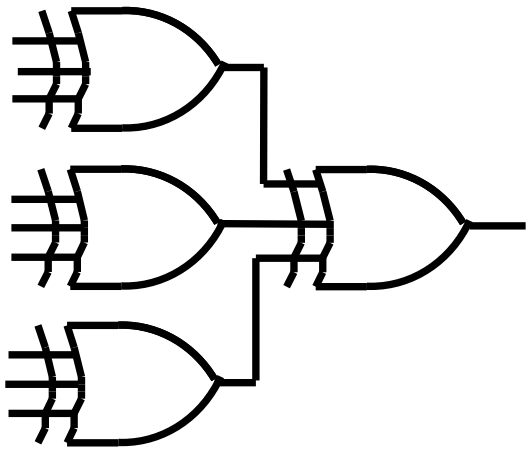
- `WARNING:HDLCompiler:314 - "priority_encoder4.vhd" Line 26: Choice with meta-value "1---" is ignored for synthesis`
- `WARNING:HDLCompiler:314 - "priority_encoder4.vhd" Line 27: Choice with meta-value "01--" is ignored for synthesis`
- `WARNING:HDLCompiler:314 - "priority_encoder4.vhd" Line 28: Choice with meta-value "001-" is ignored for synthesis`

Kodirnik prioritete (if-elsif-else)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
ENTITY priority IS
PORT (    w : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
          y : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
          z : OUT STD_LOGIC);
END priority;
ARCHITECTURE arch_case OF priority IS
BEGIN
    pri_enc : process (w) is
     begin
      if (w(3)='1') then
          y <= "11";
      elsif (w(2)='1') then
          y <= "10";
      elsif (w(1)='1') then
          y <= "01";
      elsif (w(0)='1') then
          y <= "00";
      else
          y <= "XX";
      end if;
end process pri_enc;
z <='0' WHEN w = "0000" ELSE '1';
END arch_case;
```

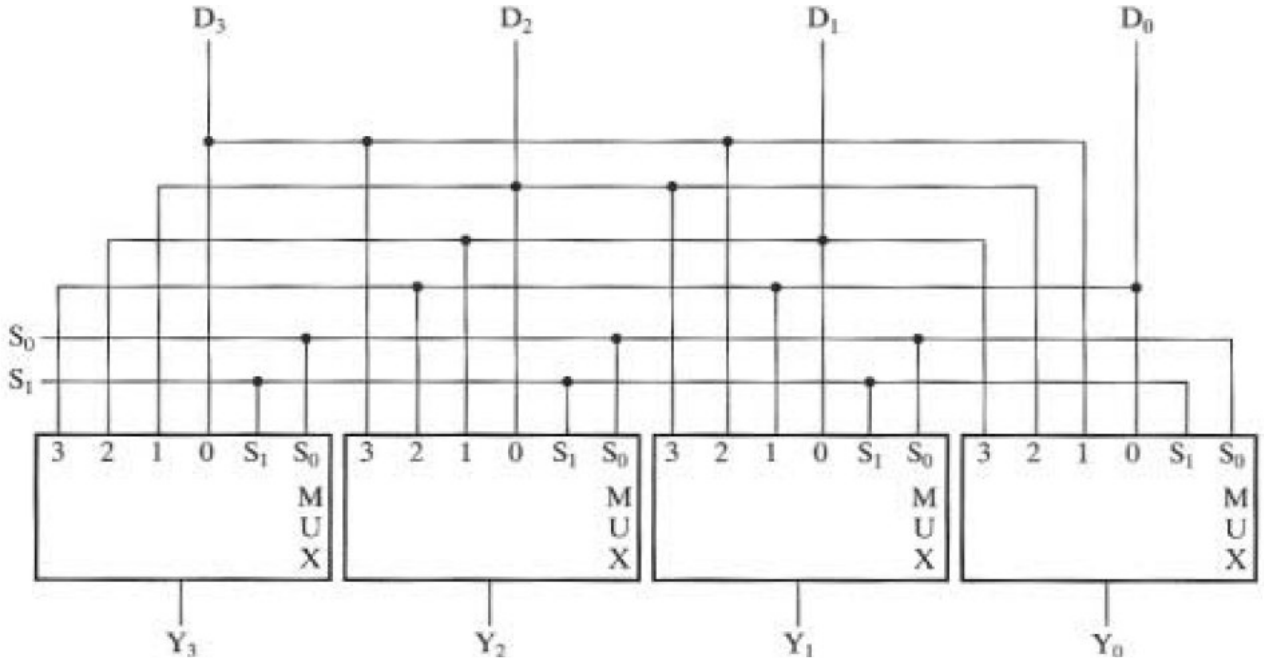
Kombinacijski generator parnosti

- f='1', če je število enic na vhodih liho, sicer je f='0'.
- XOR operacijo več spremenljivk se uporablja tudi kot generator paritete. (74280 ➔ 9-bitni MSI generator paritete)
- Realizira XOR in XNOR 9 spremenljivk.
- Kombinacijsko ga v VHDL realiziramo z unarnim XOR operatorjem.



$f=x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus \ldots \oplus x_9$

Vzporedni pomikalnik podatkov – barrel shifter



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

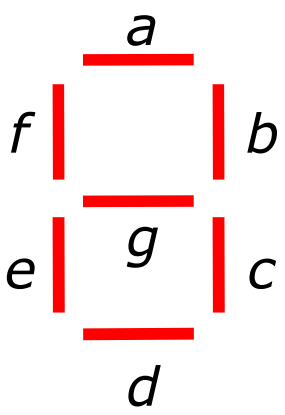
Vzporedni pomikalnik podatkov

```
architecture arch of brlshft4 is
begin
process (I, S)
begin
   case S is
     when "00" => O <= I;
     when "01" =>
              O(3) <= I(0);
              O(2 downto 0) <= I(3 downto 1);
     when "10" =>
              O(3 downto 2) <= I(1 downto 0);
              O(1 downto 0) <= I(3 downto 2);
     when "11" =>
              O(3 downto 1) <= I(2 downto 0);
              O(0) <= I(3);
     when others => O <= I;
   end case;
end process;
end arch;
```

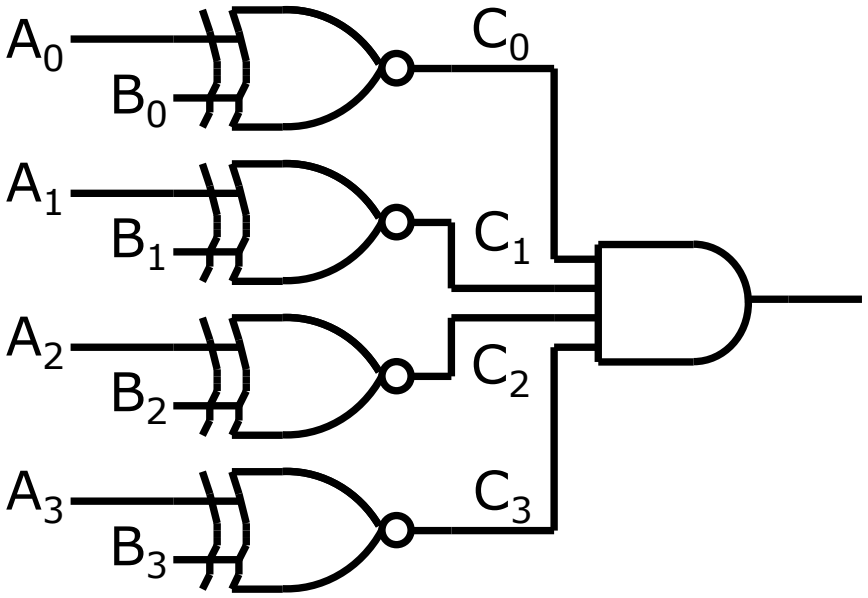
Vhodi						Izhodi			
S1	S0	I0	I1	I2	I3	O0	O1	O2	O3
0	0	a	b	c	d	a	b	c	d
0	1	a	b	c	d	b	c	d	a
1	0	a	b	c	d	c	d	a	b
1	1	a	b	c	d	d	a	b	c

BCD ➔ 7-segmentni dekodirnik v VHDL

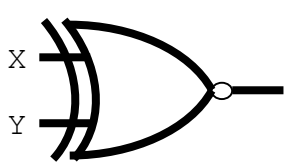
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY seg7 IS
PORT (D       : IN  STD_LOGIC_VECTOR (3 DOWNTO 0);  -- BCD vhod
      S       : OUT STD_LOGIC_VECTOR (6 DOWNTO 0)); -- 7 segmentni izhod,
                                                    -- negativna logika: vrstni red gfedcba
END seg7;
ARCHITECTURE display OF SEG7 IS
BEGIN
s <=  "1000000" WHEN d = "0000" ELSE
      "1111001" WHEN d = "0001" ELSE
      "0100100" WHEN d = "0010" ELSE
      "0110000" WHEN d = "0011" ELSE
      "0011001" WHEN d = "0100" ELSE
      "0010010" WHEN d = "0101" ELSE
      "0000010" WHEN d = "0110" ELSE
      "1111000" WHEN d = "0111" ELSE
      "0000000" WHEN d = "1000" ELSE
      "0010000" WHEN d = "1001" ELSE
      "0001000" WHEN d = "1010" ELSE
      "0000011" WHEN d = "1011" ELSE
      "1000110" WHEN d = "1100" ELSE
      "0100001" WHEN d = "1101" ELSE
      "0000110" WHEN d = "1110" ELSE
      "0001110";
END display;
```



4 bitni primerjalnik enakosti



(EQU)

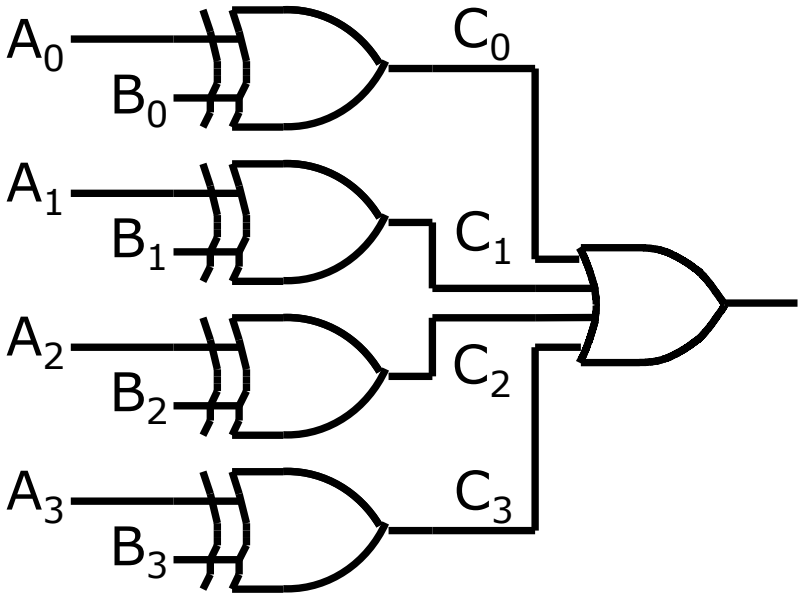


X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

$Z = (X \oplus Y)'$

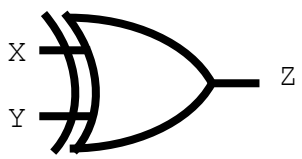
"Equality comparator"

4 bitni primerjalnik enakosti



f=**0**, ko velja
(A == B)

(XOR)



X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

$Z = (X \oplus Y)$

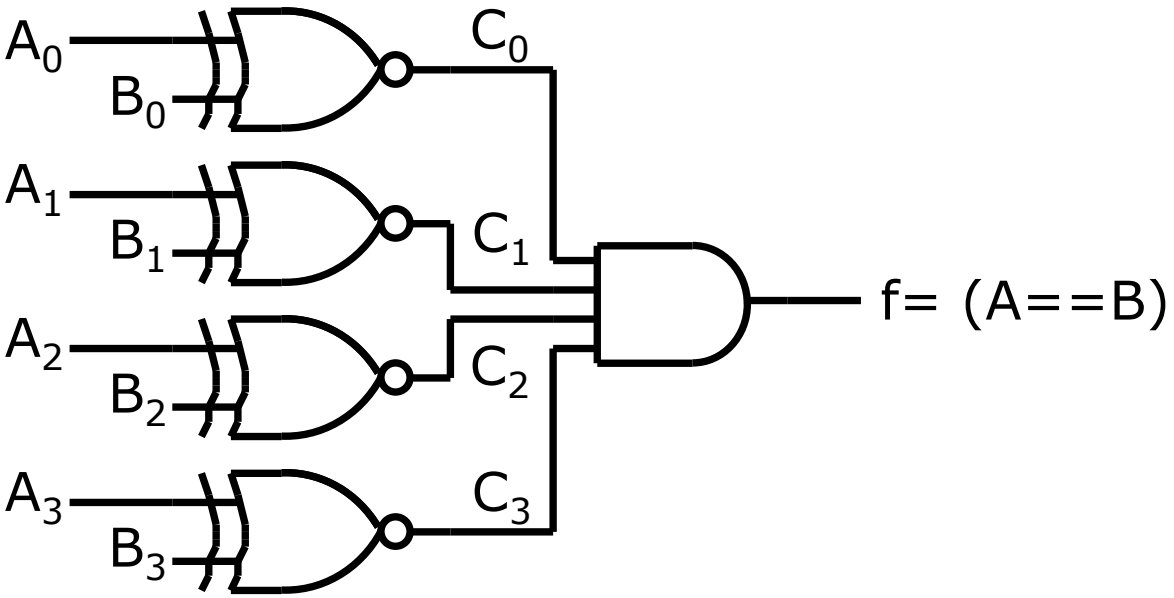
"Equality comparator"

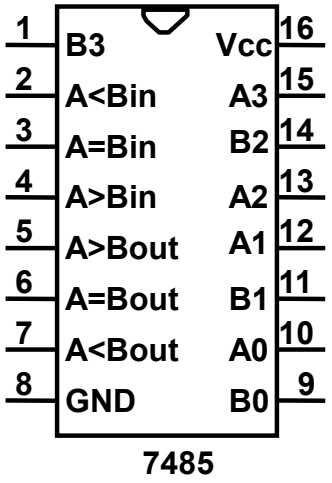
4 bitni primerjalnik velikosti

Primerja števili A in B. Kdaj bo A>B? $A=A_3A_2A_1A_0$ in $B=B_3B_2B_1B_0$

- Če je A=9 in B=7 je jasno A>B. Zakaj?
  A=1001= $A_3A_2A_1A_0$
  B=0111= $B_3B_2B_1B_0$
- Velja $A_3>B_3$, oziroma $A_3 \cdot B_3' = 1$
- Če je A=13 in B=11 je jasno A>B. Zakaj?
  A=1101
  B=1011
- Zato ker velja $A_2>B_2$, torej pišemo $A_2 \cdot B_2' = 1$
  ob pogoju, da sta A_3 in B_3 enaka
- Če je A=11 in B=10 je spet jasno A>B. Zakaj?
  A=1011
  B=1010
- Zato ker velja $A_0>B_0$, torej pišemo $A_0 \cdot B_0' = 1$
  ob pogoju, da sta enaka A_3 in B_3, A_2 in B_2, A_1 in B_1

4 bitni primerjalnik velikosti

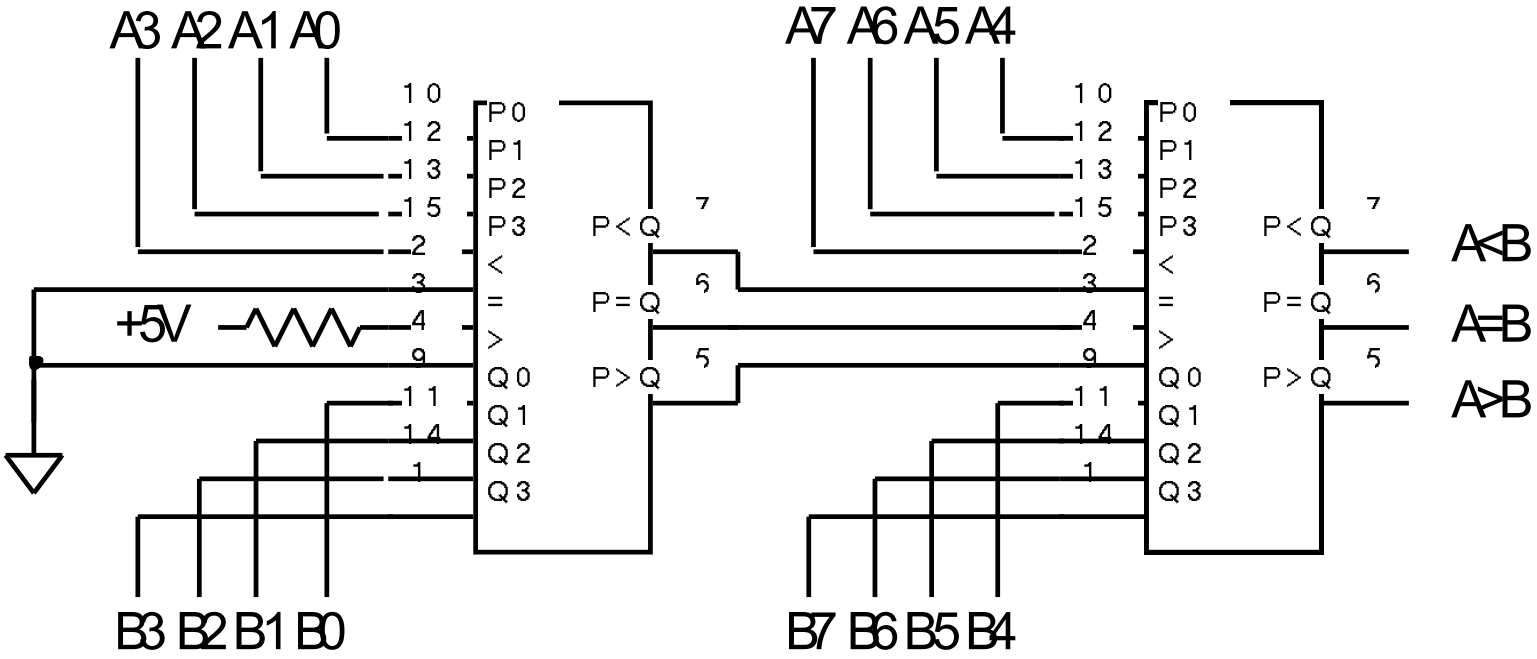




$A > B \rightarrow f = A_3 \cdot B_3' + C_3 \cdot A_2 \cdot B_2' + C_3 \cdot C_2 \cdot A_1 \cdot B_1' + C_3 \cdot C_2 \cdot C_1 \cdot A_0 \cdot B_0'$

Večbitni primerjalnik velikosti

Kako bi določili A>B? Če bi šli neposredno primerjati vse kombinacije (4 bite A + 4 bite B) bi imeli 256 kombinacij.



Kaskadna vezava dveh 7485 primerjalnikov:
8-bitni primerjalnik velikosti

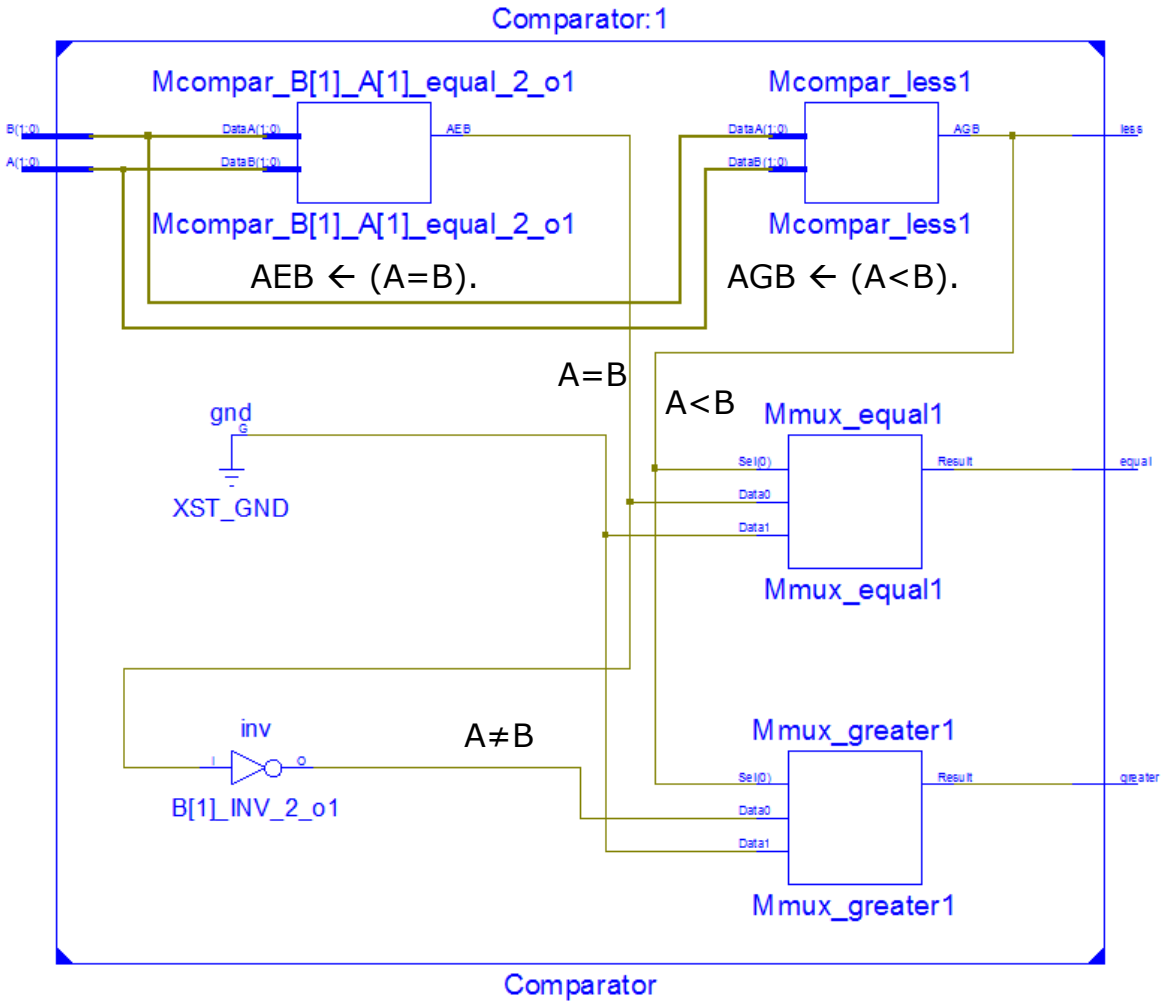
Primerjalnik velikosti v VHDL

```
library ieee;
use ieee.std_logic_1164.all;
entity Comparator is
generic(n: natural :=2);
    port(
            A: in std_logic_vector(n-1 downto 0);
            B: in std_logic_vector(n-1 downto 0);
            less, equal, greater: out std_logic
    );
end Comparator;
architecture arch of Comparator is
begin
    process(A,B)
    begin
            if (A<B) then
                        less <= '1';
                        equal <= '0';
                        greater <= '0';
            elsif (A=B) then
                        less <= '0';
                        equal <= '1';
                        greater <= '0';
            else
                        less <= '0';
                        equal <= '0';
                        greater <= '1';
            end if;
    end process;
end arch;
```

VHDL operator	Opis
=	Enakost
/=	Neenakost
<	Manj kot
<=	Manjši ali enak
>	Večji
>=	Večji ali enak

Primerjalnik velikosti v VHDL

```
architecture arch of Comparator is
begin
    process(A,B)
    begin
            if (A<B) then
                        less <= '1';
                        equal <= '0';
                        greater <= '0';
            elsif (A=B) then
                        less <= '0';
                        equal <= '1';
                        greater <= '0';
            else
                        less <= '0';
                        equal <= '0';
                        greater <= '1';
            end if;
    end process;
end arch;
```

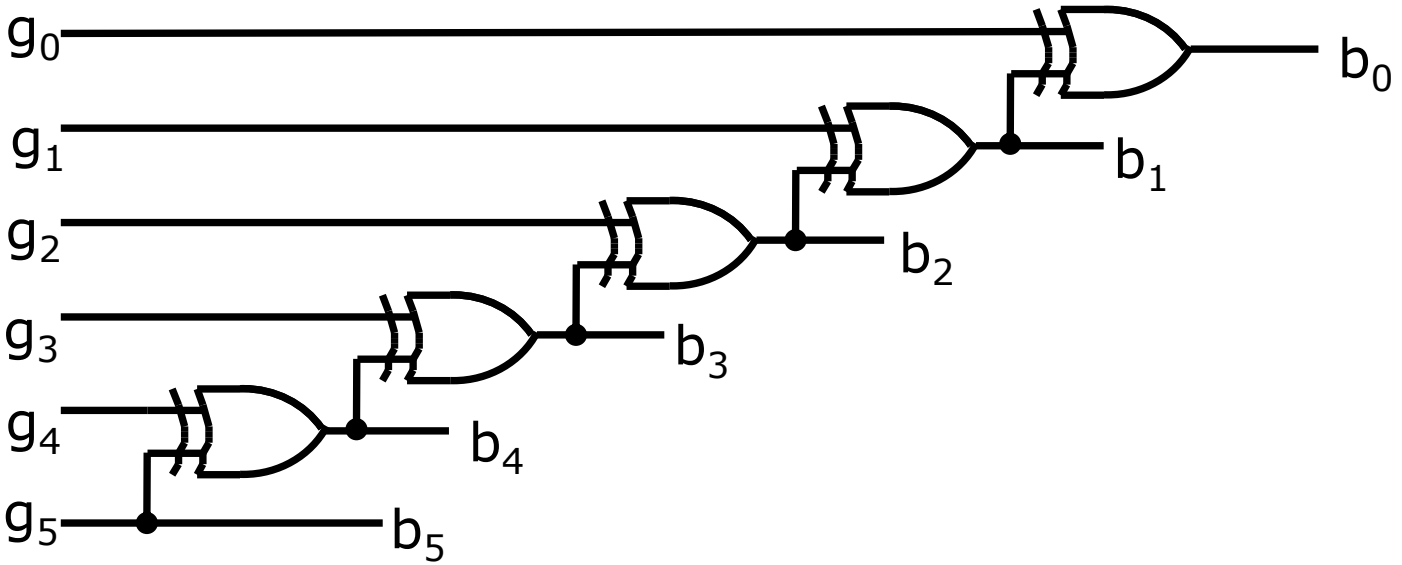


Načrtovanje digitalnih vezij

Realizacija ponavljajočih se struktur v VHDL (for-generate, if-generate)

Realizacija ponavljajočih se struktur v VHDL

- VHDL ima nekaj sočasnih izjav za opis ponavljajočih struktur, kot je npr. pretvornik n_2 ➔ GK
- Uporabljajo se za sestavljanje parametriziranih struktur (r-bitni pretvornik n_2 ➔ GK)
- V ta namen obstajata for-generate zanka in if-generate stavek, ki v kombinaciji z povezovalnim stavkom omogočata algoritemsko povezovanje komponent v ponavljajoče se strukture.



Parametriziran kombinacijski pretvornik n_2 ←→ Gray

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity bin2gray_gray2bin is
generic (n: natural := 4);
port (    X          : in   STD_LOGIC_VECTOR (n-1 downto 0);
          mode       : in   STD_LOGIC; --mode: '1'->bin2gray, '0'->gray2bin
          Y          : out  STD_LOGIC_VECTOR (n-1 downto 0));
end bin2gray_gray2bin;
architecture arch of bin2gray_gray2bin is
signal A: STD_LOGIC_VECTOR (n downto 0);
begin
G1 :      for i in n-1 downto 0 generate - indeks i pada (downto), ne raste (to) !
          begin
          G1a :     if i = n-1 generate
                    begin
                              A(i) <= X(i); -- if-generate nima else pogoja!
                    end generate G1a;
          G1b :     if i < n-1 generate
                    begin
                              A(i) <= X(i+1) xor X(i) when mode = '1' else A(i+1) xor X(i);
                    end generate G1b;
          end generate;

    Y <= A(n-1 downto 0);

end arch;
```

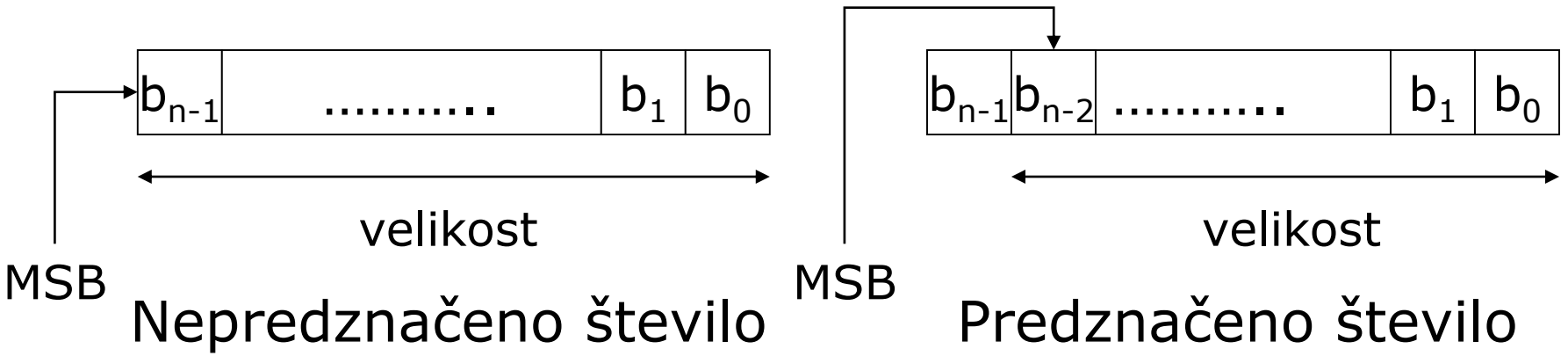
Načrtovanje digitalnih vezij

Gradniki kombinacijskih vezij
za aritmetiko:

Seštevalniki

Predznačena števila

- Za predznačena števila v dvojiškem sistemu je predznak števila (sign) označen z MSB bitom
  - 0 = pozitivno število
  - 1 = negativno število
- Za n-bitno število bo preostalih *n*-1 bitov predstavljalo velikost števila (magnitude)



Negativna števila

- Negativne vrednosti predznačenih števil lahko zapišemo v treh najbolj pogostih oblikah
  - Predznak-velikost
  - Eniški komplement
  - Dvojiški komplement
- Predznak-velikost zapis primer za 4-bitna števila

  +5=0101 -5=1101

  +3=0011 -3=1011

  +7=0111 -7=1111

Predstavitve predznačenih dvojiških števil

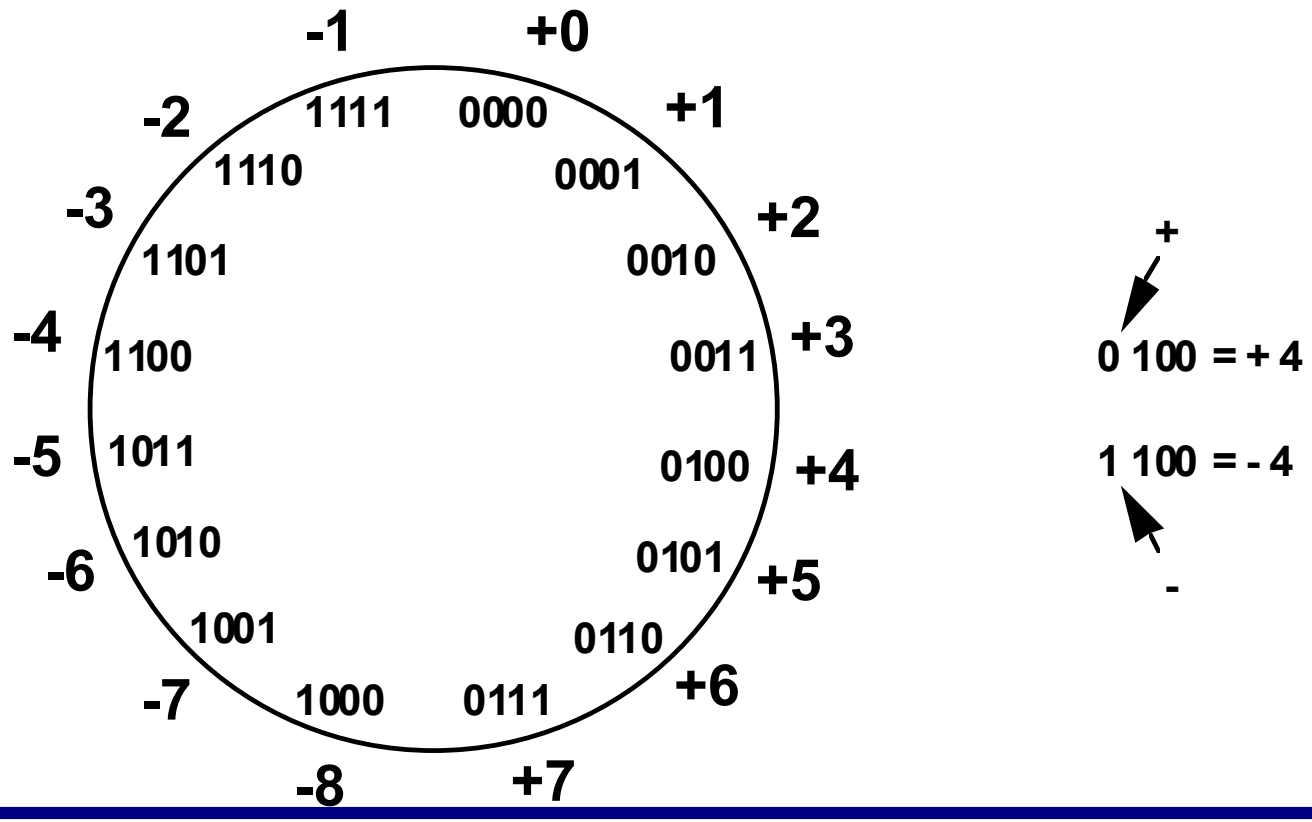
Izmed različnih predstavitev omenimo:

- predznak-veličina (sign-magnitude, SM) – zoprn za seštevanje/odštevanje.
- eniški komplement (1'K),
- dvojiški komplement (2'K) – zelo primeren za seštevanje/odštevanje

b_2	b_1	b_0	SM	1'K	2'K
0	0	0	+0	+0	+0
0	0	1	1	1	1
0	1	0	2	2	2
0	1	1	3	3	3
1	0	0	-0	-3	-4
1	0	1	-1	-2	-3
1	1	0	-2	-1	-2
1	1	1	-3	-0	-1

Dvojiški komplement (2'K)

- Ima samo eno predstavitev za število nič.
- Ima eno negativno število v obsegu več kot pozitivno: obsega zapis n-bitnih števil $[-2^{n-1} \ldots +2^{n-1}-1]$



Dvojiški komplement

- Z eniškim komplementom *n*-bitno negativno število *K* zapišemo v dvojiškem komplementu:

  $K=(2^n)-P$

- Če je *n=4*, potem

  $K=2^4-P=(16)_{10}-P=(10000)_2-P$

  $-5=(16)_{10}-5=(10000)_2-(0101)_2=(1011)_2$

  $-3=(16)_{10}-3=(10000)_2-(0011)_2=(1101)_2$

Ročno določanje dvojiškega komplementa

- Primer
  B=00110100
- Dvojiški komplement B je

B=00110100 ➔ K=11001100

Spremenjeno Nespremenjeno

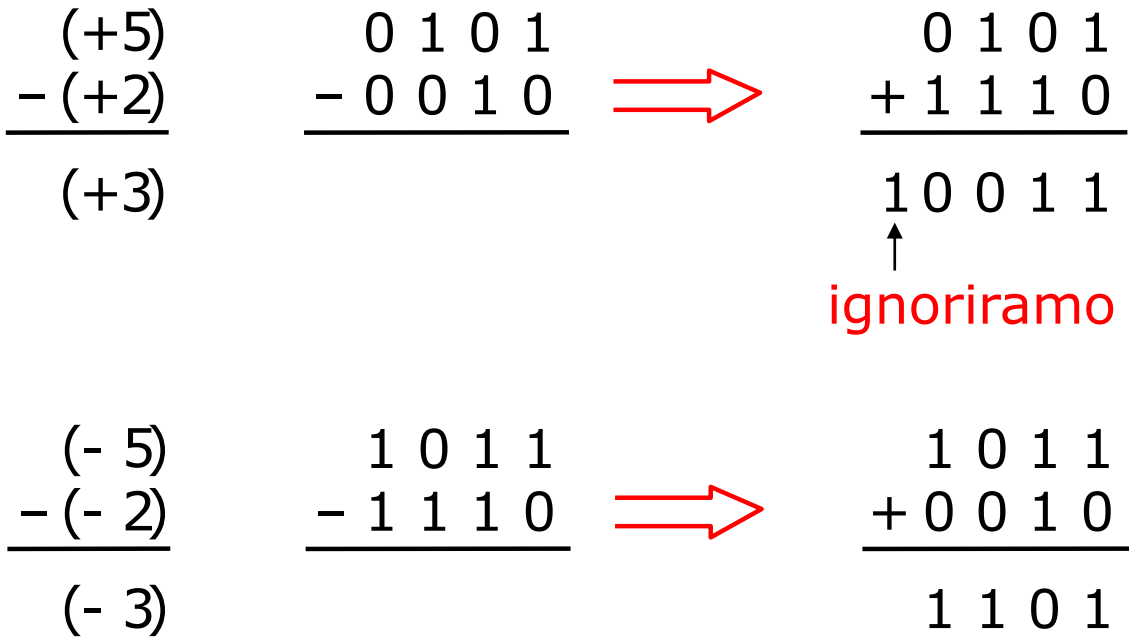
Operacije z dvojiškim komplementom

(+5)	0 1 0 1	(–5)	1 0 1 1
+(+2)	+0 0 1 0	+(+2)	+0 0 1 0
(+7)	0 1 1 1	(–3)	1 1 0 1
(+5)	0 1 0 1	(–5)	1 0 1 1
+ (–2)	+1 1 1 0	+ (–2)	+1 1 1 0
(+3)	1 0 0 1 1	(–7)	1 1 0 0 1

ignoriramo

ignoriramo

Odštevanje z dvojiškim komplementom



Preliv (overflow)

- Za zapis *predznačenega* števila uporabljamo *n* bitov
- Rezultat operacije je v območju $[-2^{n-1} \ldots +2^{n-1}-1]$
- Z *n*-bitnim zapisom v (2'K) lahko predstavimo števila v obsegu $[-2^{n-1} \ldots +2^{n-1}-1]$
  - Za $n=3$ ➔[-4..3], za $n=16$ ➔[-32768.. 32767]
- Če 32767 prištejemo 1, dobimo -32768
- To je neveljaven rezultat.

- Če rezultata ne moremo zapisati v območje predstavitve n-bitnega števila, potem govorimo o prelivu ali prekoračitvi (***arithmetic overflow***).

Primeri aritmetičnega preliva

$x_3\ x_2\ x_1\ x_0$
$+\ y_3\ y_2\ y_1\ y_0$
$c_4\ c_3\ c_2\ c_1$
$s_3\ s_2\ s_1\ s_0$

(+7)	0 1 1 1	(–7)	1 0 0 1
+(+2)	+0 0 1 0	+(+2)	+0 0 1 0
(+9)	1 0 0 1	(–5)	1 0 1 1
	$c_4 = 0$		$c_4 = 0$
	$c_3 = 1$		$c_3 = 0$

(+7)	0 1 1 1	(–7)	1 0 0 1
+(–2)	+1 1 1 0	+(–2)	+1 1 1 0
(+5)	1 0 1 0 1	(–9)	1 0 1 1 1
	$c_4 = 1$		$c_4 = 1$
	$c_3 = 1$		$c_3 = 0$

Če imajo števila različne predznake, se preliv ne zgodi!

Preliv v zapisu z 2'K

Zaznavanje preliva

Pozitivno	Negativno
+ Pozitivno	+ Negativno
= Negativno	= Pozitivno

Določanje preliva

$$OF_{2'K} = \overline{a_{k-1}} \cdot \overline{b_{k-1}} \cdot s_{k-1} + a_{k-1} \cdot b_{k-1} \cdot \overline{s_{k-1}} = c_k \oplus c_{k-1}$$

Načrtovanje digitalnih vezij

Predstavitev števil in
aritmetična vezja:

Predstavitev števil v VHDL

Števila v VHDL – kaj je standardno in kaj ni

- V VHDL za delo s števili obstaja več paketov iz zgodovinskih razlogov. Izpostavili bomo dva:
  - STD_LOGIC_ARITH (vsebuje tudi tipa std_logic_signed, std_logic_unsigned)
  - NUMERIC_STD (standardiziran na IEEE)
- V nadaljevanju se bomo držali ***izključno*** zapisov v paketni datoteki 'NUMERIC_STD' saj je standardiziran in definiran v vseh okoljih (Altera Quartus II, Xilinx ISE, Active-HDL, Synopsis ...)

Števila v NUMERIC_STD

- Podatkovna tipa v NUMERIC_STD:
  - SIGNED, UNSIGNED
- UNSIGNED - samo pozitivne vrednosti
  - **signal** count: unsigned (3 **downto** 0)
    (4-bitni signal, vrednosti od 0 do 15)
- SIGNED
  - predznačena vrednost v 2'K, kjer MSB predstavlja bit predznaka
  - **signal** count: signed (3 **downto** 0)
  - (4-bitni signal, vrednosti od -8 do 7)

Števila v NUMERIC_STD

- Če velikosti signed/unsigned ne definiramo, sintetizator predpostavi 32-bitno velikost, kar je zelo potratno.
- signed/unsigned sta pravzaprav posebna podtipa std_logic_vector, ki dovoljujeta samo vrednosti '0', '1' na mestih zapisa.
  - `type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;`
  - `type SIGNED is array (NATURAL range <>) of STD_LOGIC;`
- S tipom std_logic_vector ne moremo izvajati, primerjav in aritmetičnih operacij ('U' + 'U'= WTF?)
- Zato potrebujemo nabor funkcij za pretvorbo med std_logic_vector in signed/unsigned.

Operatorji v NUMERIC_STD

- Paketna datoteka NUMERIC_STD vsebuje definicije aritmetičnih operatorjev
  - seštevanja (+), odštevanja (-), množenja (*), deljenja (/), celoštevilskega deljenja (**mod**) in ostanka (**rem**) ter potenciranja (**)
  - Primerjave (=, >, <, <=, >=, /=)
- Nekatere operatorje lahko sintetiziramo (+,-,*), drugih ne (/, **mod**, **rem**, **).
- Vključimo jo z ukazom "**use**" na vrhu kode:

```
use ieee.numeric_std.all;
```

Zapis števil v NUMERIC_STD

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

Uporabimo knjižnico **numeric_std – nikakor std_logic_arith**, saj ni standardna (Synopsis)

~~library ieee ;~~
~~use ieee.std_logic_1164.all ;~~
~~use ieee.std_logic_arith.all ;~~
~~use ieee.std_logic_unsigned.all ;~~

```
signal A : unsigned(3 downto 0);
signal B : signed (3 downto 0);
signal C : std_logic_vector (3 downto 0);
A <= "1111" ; -- 15_10
B <= "1111" ; -- (-1)_10
C <= "1111" ; -- ni število, ampak vektor enic
```

Pretvorbe v NUMERIC_STD

- signed -> std_logic_vector

```
signal A : unsigned(3 downto 0);
signal B : signed (3 downto 0);
signal C : std_logic_vector (3 downto 0);
```

Pretvorba std_logic_vector → število

- V aritmetičnih vezjih so običajno vsi priključki tipa std_logic vector, operandi arhitekture pa so števila tipa signed oz. unsigned.
- Prvi korak predstavlja pretvorba a v a_signed

```
signal a : in STD_LOGIC_VECTOR( OP_SIZE-1 downto 0);
signal a_signed : signed (a'range);
signal a_unsigned : unsigned (a'range);
   -- pretvori std_logic_vector v signed
   a_signed <= signed(a);
   -- pretvori std_logic_vector v unsigned
   a_unsigned <= unsigned(a);
```

Tabela pretvorb

Ponor / Izvor	**integer**	**unsigned**	**signed**	**slv (standard logic vector)**
integer		`to_unsigned ( arg,size: natural)`	`to_signed ( arg: integer; size: natural)`	`std_logic_vector(to_unsigned (my_int, my_slv'length) ); Pozor!`
unsigned	`to_integer ( arg: unsigned)`		`signed ( arg: unsigned)`	`to_stdlogicvector ( arg: unsigned)`
signed	`to_integer ( arg: signed)`	`unsigned ( arg: signed)`		`to_stdlogicvector ( arg: signed)`
slv (standard logic vector)	`to_integer(unsigned(arg: unsigned)); Pozor!`	`to_unsigned ( arg: std_logic_vector)`	`to_signed ( arg: std_logic_vector)`	

Pozor: Obstajata dve možnosti.

Primeri pretvorbe: integer→ slv, unsigned, signed

```
signal i1  : integer;
signal o1a : std_logic_vector(3 downto 0);
signal o1b : std_logic_vector(3 downto 0);
--integer → slv samo za pozitivna cela stevila
o1a <= std_logic_vector(to_unsigned(i1, o1a'length));
-- integer → slv za vsa cela stevila
o1b <= std_logic_vector(to_signed(i1, o1b'length));

signal i2  : integer;
signal o2 : unsigned(3 downto 0);
o2 <= to_unsigned(i2, o2'length); -- integer → unsigned

signal i3  : integer;
signal o3 : signed(3 downto 0);
o3 <= to_signed(i3, o3'length); -- integer → signed
```

Primeri pretvorbe: slv→ integer, unsigned, signed

```
signal i4   : std_logic_vector(3 downto 0);
signal o4a : integer;
signal o4b : integer; -- slv → integer
o4a <= to_integer(unsigned(i4)); -- nepredznačen primer
o4b <= to_integer(signed(i4)); -- predznačen primer

signal i5  : std_logic_vector(3 downto 0);
signal o5 : unsigned(3 downto 0);
o5 <= unsigned(i5); -- slv → unsigned

signal i6  : std_logic_vector(3 downto 0);
signal o6 : signed(3 downto 0);
o6 <= signed(i6); -- slv → signed
```

Primeri pretvorbe: unsigned→ integer, signed, slv

```
signal i7  : unsigned(3 downto 0);
signal o7 : integer;
o7 <= to_integer(i7); -- unsigned → integer

signal i8  : unsigned(3 downto 0);
signal o8 : std_logic_vector(3 downto 0);
o8 <= std_logic_vector(i8); -- unsigned → slv

signal i9  : unsigned(3 downto 0);
signal o9 : signed(3 downto 0);
o9 <= signed(i9); -- unsigned → signed
```

Pretvorba število → std_logic_vector

- Obratna operacija v aritmetičnih vezjih zahteva pretvorbo števila (rezultata) operacije nazaj v priključek tipa std_logic vector.

```
signal sum  : std_logic_vector(OP_SIZE-1 downto 0);
signal sum_sig : signed (sum'range);
sum <= std_logic_vector(sum_sig); -- pretvorba;
```

Pretvorba std_logic → število

- Včasih se na vhodu vezja pojavi 1-bitna vrednost (std_logic), s katero moramo izvesti aritmetično operacijo z drugim številom.
- Primer tega vezja je seštevalnik, kjer izvedemo prištevanje vhodnega prenosa pri (c_in) operandoma a in b, ki sta nepredznačeni števili.

```
signal c_in : in std_logic;
signal c_in_ext : signed (OP_SIZE downto 0);
signal c_in_vector : std_logic_vector (OP_SIZE downto 0);
  c_in_vector <= (0 => c_in, others=>'0');
  c_in_ext <= signed(c_in_vector);
  -- pretvori c_in std_logic v signed
```

- Druga možnost je z vpisom std_logic na LSB mesto vektorja:

```
  c_in_ext(c_in_ext'right) <= c_in;
  -- to vrne opozorilo glede neuporabljenih mest [OP_SIZE:1]!
```

Razširjanje mest nepredznačenega števila

- Število a naj bo **nepredznačeno**. Razširiti ga želimo na novo nepredznačeno število, ki zaseda več bitov

```
signal a : unsigned (OP_SIZE-1 downto 0);
signal a_padded : unsigned (OP_SIZE downto 0);
-- razsiri mesta z dodajanjem ene nicle na MSB
   resize('0' & a) (ang. zero padding)
a_padded <= resize(a, a_padded'length);
```

Razširjanje mest predznačenega števila

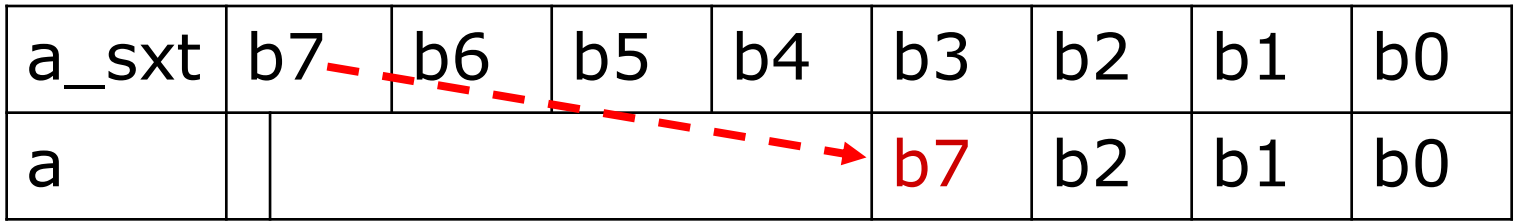
- Število a naj bo **predznačeno**. Razširiti ga želimo na novo predznačeno število, ki zaseda več bitov.
- Temu pravimo razširjanje predznaka (ang. sign extension)

```
signal a : signed (OP_SIZE-1 downto 0);
signal a_sxt : signed (OP_SIZE downto 0);
-- razsiri mesta z replikacijo MSB mesta
  stevila a - isto kot resize(a(a'left) & a)
a_sxt <= resize(a, a_sxt'length);
```

Krajšanje predznačenega števila

- Predznačeno število a želimo skrajšati - mu odvzeti nekaj mest z uporabo funkcije **`resize`**.
- Krajšanje mest števila se izvaja z MSB strani, tako da se MSB originala ohranja!

```
signal a : signed (7 downto 0);
signal a_sxt : signed (3 downto 0);
-- ohrani se predznak a_sxt, brišejo se b6, b5, b4, b3!
a <= resize(a_sxt, a'length);
```



Krajšanje nepredznačenega števila

- Predznačeno število a želimo skrajšati - mu odvzeti nekaj mest z uporabo funkcije resize.
- Krajšanje mest števila se izvaja normalno z MSB strani – MSB originala se ne ohranja!!!

```
signal a : unsigned (7 downto 0);
signal a_sxt : unsigned (3 downto 0);
-- brišejo se b7, b6, b5, b4!
a <= resize(a_sxt, a'length);
```

a_sxt	b7	b6	b5	b4	b3	b2	b1	b0
a					b3	b2	b1	b0

Načrtovanje digitalnih vezij

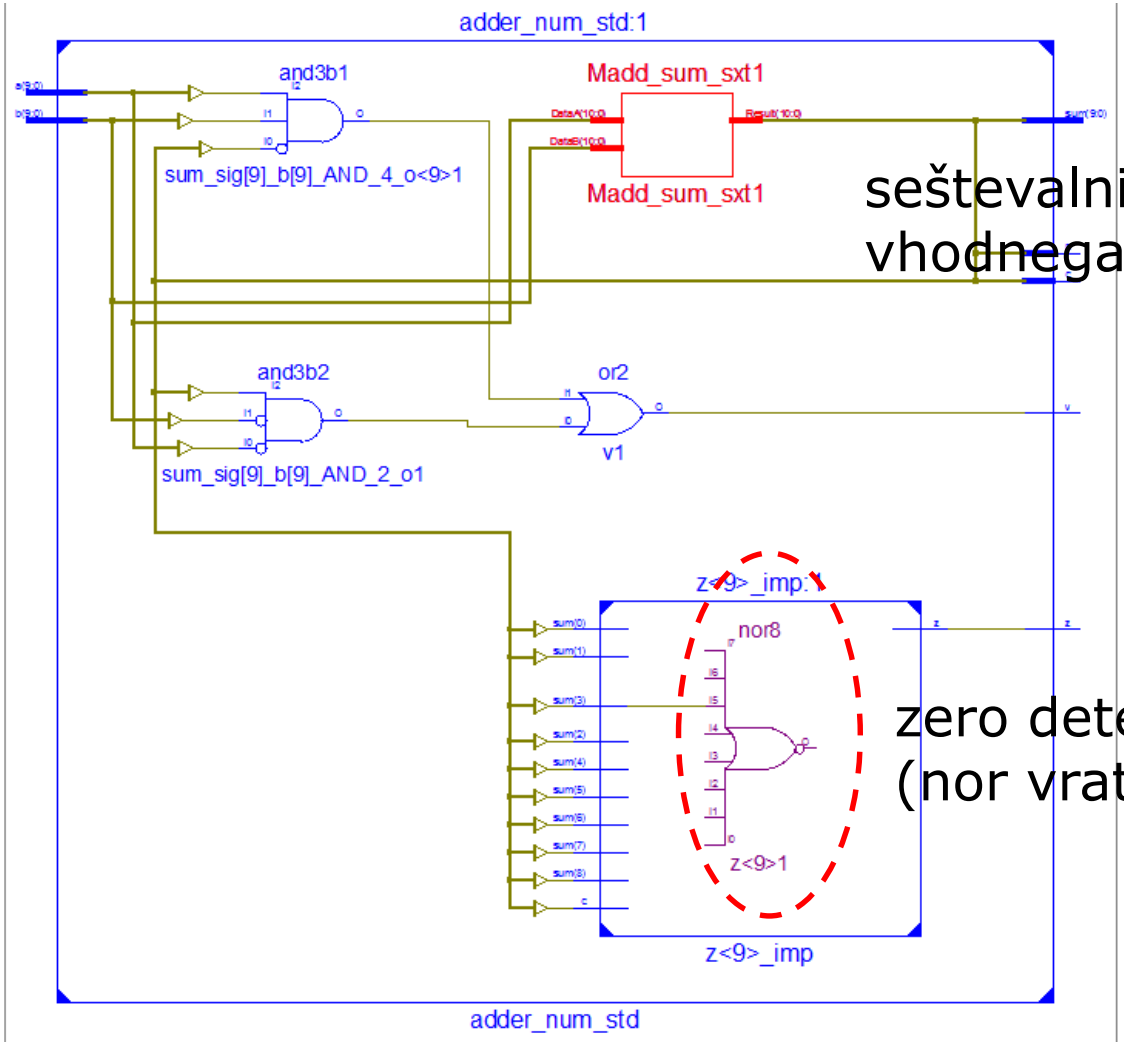
Predstavitev števil in
aritmetična vezja:

Preliv v VHDL

Predznačeno seštevanje in NCVZ

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity adder_num_std is
generic ( OP_SIZE : Natural := 10 );
port(     a, b : in STD_LOGIC_VECTOR(OP_SIZE-1 downto 0);
          sum  : out STD_LOGIC_VECTOR(OP_SIZE-1 downto 0);
          n, c, v, z : out STD_LOGIC );
end adder_num_std;
architecture arch_signed_ncvz of adder_num_std is
signal sum_sxt, a_sxt, b_sxt : signed (OP_SIZE downto 0);
signal sum_sig : signed (a'range);
constant ZERO : signed(a'range) := (others=>'0'); --nicla
begin
    a_sxt <= resize(signed(a), a_sxt'length); -- razsiri predznak pred sestevanjem
    b_sxt <= resize(signed(b), b_sxt'length);
    sum_sxt <= a_sxt + b_sxt;
    sum_sig <= resize(sum_sxt, a'length); -- odstranimo MSB -> vsota na mestih [N-1..0]
    sum <= std_logic_vector(sum_sig);   -- da se izognemo tipu inout za sum;
    n <= std_logic(sum_sig(sum_sig'left));                  -- MSBbit je bit predznaka
    c <= std_logic(sum_sxt(sum_sxt'left));                  -- MSB+1 bit je izhodni prenos
    -- preliv se postavi pri sestevanju, ko sta negativna a, b in je pozitiven rezultat ali
    -- v primeru ko sta pozitivna a, b in je negativen rezultat
    v <= std_logic( (sum_sig(sum_sig'left) and not a(a'left) and not b(b'left)) or
                    (not sum_sig(sum_sig'left) and a(a'left) and b(a'left)));
    z <= '1' when (sum_sig = ZERO) else '0'; -- primerjalnik enakosti (nor vrata v sintezi)
end arch_signed_ncvz;
```

Predznačeno seštevanje in NCVZ



seštevalnik brez vhodnega prenosa!

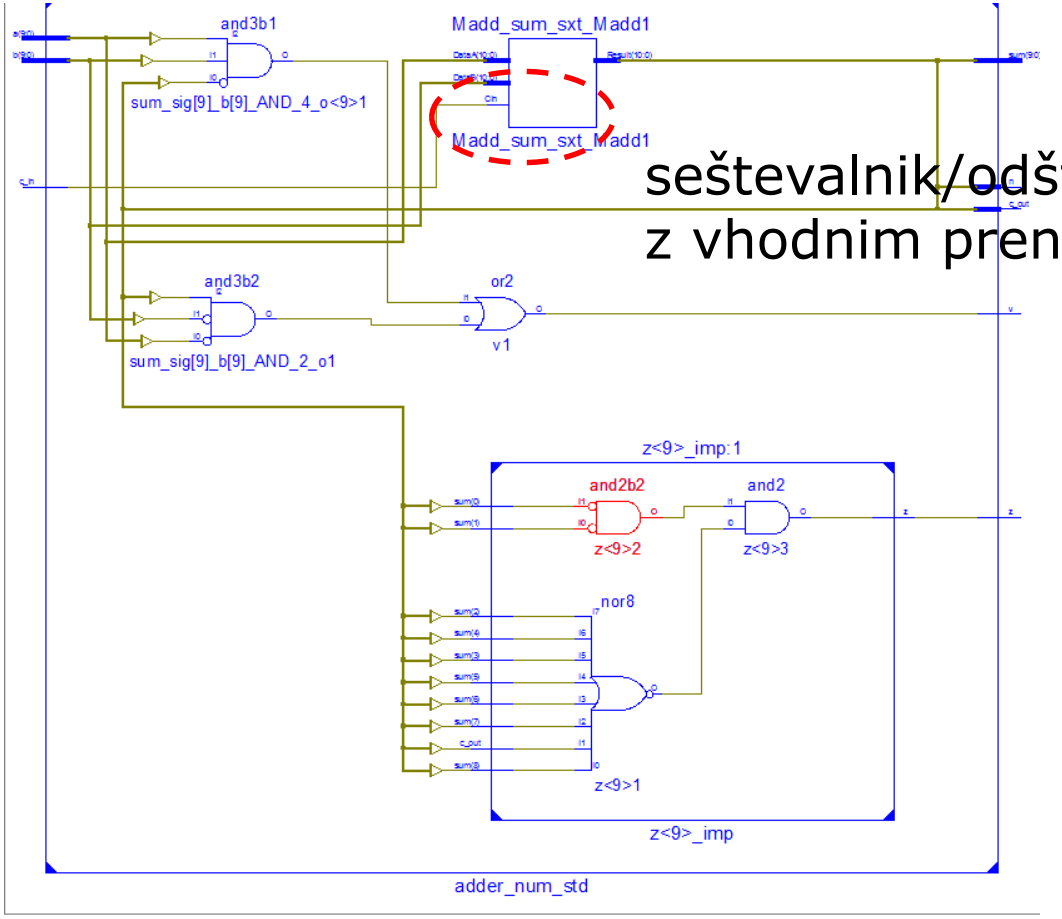
zero detektor (nor vrata)

Predznačeno seštevanje in NCVZ z vhodnim prenosom

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity adder_num_std is
generic (   OP_SIZE : Natural := 10 );
  port(
     c_in : in std_logic;
     a, b : in std_logic_vector(OP_SIZE-1 downto 0);
     sum  : out std_logic_vector(OP_SIZE-1 downto 0);
     n, c_out, v, z : out std_logic
);
end adder_num_std;
architecture arch_signed_ncvz of adder_num_std is
constant ZERO : signed(a'range) := (others=>'0'); --
    nicla
signal sum_sxt, a_sxt, b_sxt, c_in_ext : signed
    (OP_SIZE downto 0);
signal c_in_vector : std_logic_vector (OP_SIZE downto
    0);
signal sum_sig : signed (a'range);
```

```
begin
    c_in_vector <= (0 => c_in, others=>'0');
    c_in_ext <= signed(c_in_vector); -- pretvori c_in
    std_logic v signed
    --c_in_ext(c_in_ext'right) <= c_in;
    --zg. pretvori c_in std_logic v signed - vrže
    opozorilo glede neuporabljenih bitov [10:1]!
    a_sxt <= resize(signed(a), a_sxt'length); --
    razsiri predznak pred sestevanjem
    b_sxt <= resize(signed(b), b_sxt'length);

    sum_sxt <= a_sxt + b_sxt + c_in_ext;

    sum_sig <= resize(sum_sxt, a'length);--
    odstranimo MSB -> vsota na mestih [N-1..0]
    sum <= std_logic_vector(sum_sig);-- da se
    izognemo tipu inout za sum;

    n <= std_logic(sum_sig(sum_sig'left));      --
    MSBbit je bit predznaka
    c_out <= std_logic(sum_sxt(sum_sxt'left)); --
    MSB+1 bit je izhodni prenos
    -- preliv se postavi pri sestevanju, ko sta
    negativna a, b in je pozitiven rezultat ali
    -- v primeru ko sta pozitivna a, b in je
    negativen rezultat
    v <= std_logic((sum_sig(sum_sig'left) and not
    a(a'left) and not b(b'left)) or
    (not sum_sig(sum_sig'left) and a(a'left) and
    b(a'left)));
    z <= '1' when (sum_sig = ZERO) else '0';--
    primerjalnik enakosti (nor vrata v sintezi)
end arch_signed_ncvz;
```

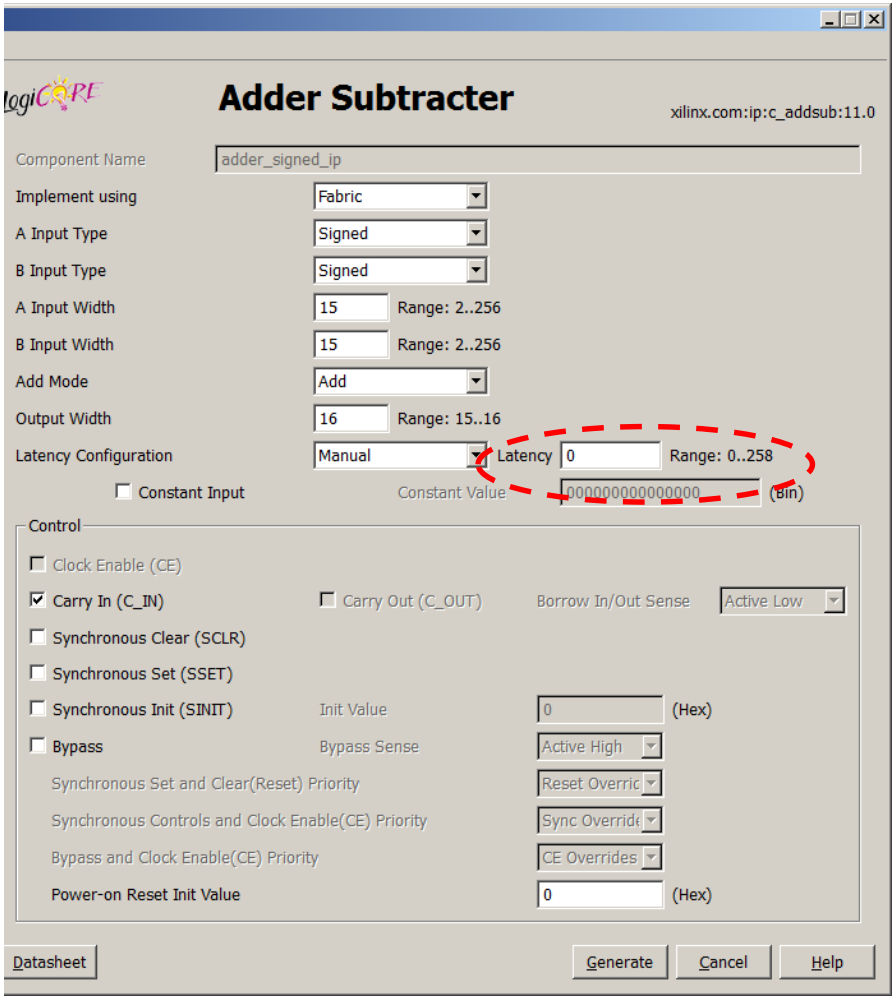
Predznačeno seštevanje in NCVZ



seštevalnik/odštevalnik
z vhodnim prenosom!

Uporaba IP core generatorja

- Vgrajeni seštevalnik postavimo na kombinacijsko izvedenko tako, da postavimo zakasnitev (ang. latency) na 0.
- Sicer je ta struktura sekvenčno vezje (ima CLK signal za FIFO medpomnenje vhodnih in izhodnega podatka).



Uporaba IP core generatorja

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity adder_num_std is
generic (   OP_SIZE : Natural := 10 );
  port(
     c_in : in std_logic;
     a, b : in std_logic_vector(OP_SIZE-1 downto 0);
     sum  : out std_logic_vector(OP_SIZE-1 downto 0);
     n, c_out, v, z : out std_logic
);
end adder_num_std;
arch_signed_ncvz_ip of adder_num_std is
COMPONENT adder_signed_ip
PORT (
      a : IN STD_LOGIC_VECTOR(14 DOWNTO 0);
      b : IN STD_LOGIC_VECTOR(14 DOWNTO 0);
      c_in : IN STD_LOGIC;
      s : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
);
END COMPONENT;

signal a_sxt, b_sxt, sum_sig : STD_LOGIC_VECTOR (14
     downto 0);

signal sum_sxt : STD_LOGIC_VECTOR (15 downto 0);

constant ZERO : std_logic_vector(a_sxt'range) :=
     (others=>'0'); -- nicla
```

```
begin
--IP core generator adder/subtracter deluje z
std_logic_vectorji, zato jih moramo razsiriti predznak
z resize
a_sxt <= std_logic_vector(resize( signed(a),
     a_sxt'length));
b_sxt <= std_logic_vector(resize( signed(b),
     b_sxt'length));

U1 : adder_signed_ip PORT MAP (
a => a_sxt, b => b_sxt, c_in => c_in, s => sum_sxt);
--latenco postavimo na 0, da je kombinacijski!

sum_sig <= sum_sxt(sum_sig'range);  -- odstranimo MSB
     -> vsota na mestih [N-1..0]

n <= std_logic(sum_sig(sum_sig'left));
          -- MSBbit je bit predznaka
c_out <= std_logic(sum_sxt(sum_sxt'left));
          -- MSB+1 bit je izhodni prenos
-- preliv se postavi pri sestevanju, ko sta negativna
     a, b in je pozitiven rezultat ali
     -- v primeru ko sta pozitivna a, b in je
     negativen rezultat
v <= std_logic((sum_sig(sum_sig'left) and not
     a(a'left) and not b(b'left)) or
     (not sum_sig(sum_sig'left) and a(a'left) and
     b(a'left)));
     z <= '1' when (sum_sig = ZERO) else '0';    --
     primerjalnik enakosti (nor vrata v sintezi)
end arch_signed_ncvz_ip;
```

Predznačeno seštevanje z & operatorjem

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity adder_num_std is
generic (   OP_SIZE : Natural := 10 );
port(
) c_in : in std_logic;
a, b : in std_logic_vector(OP_SIZE-1 downto 0);
sum  : out std_logic_vector(OP_SIZE-1 downto 0);
n, c_out, v, z : out std_logic
;
end adder_num_std;
arch_signed_ncvz_concatenate of adder_num_std is
constant ZERO : signed(a'range) := (others=>'0');
signal sum_sxt, a_sxt, b_sxt, c_in_ext : signed
    (OP_SIZE downto 0);
signal c_in_vector : std_logic_vector (OP_SIZE downto
    0) := (others=>'0');
signal sum_sig : signed (a'range);
```

```
begin
c_in_vector(c_in_vector'right) <= c_in;
-- postavi c_in na LSB mesto c_in_vector
c_in_ext <= signed(c_in_vector);
-- pretvori c_in std_logic v signed
a_sxt <= signed(a(a'left) & a);
-- razsiri predznak pred sestevanjem
b_sxt <= signed(b(b'left) & b);
sum_sxt <= a_sxt + b_sxt + c_in_ext;
sum_sig <= sum_sxt(a'range);
-- odstranimo MSB -> vsota na mestih [N-1..0]
sum <= std_logic_vector(sum_sig);
-- da se izognemo tipu inout za sum;
n <= std_logic(sum_sig(sum_sig'left));
-- MSBbit je bit predznaka
c_out <= std_logic(sum_sxt(sum_sxt'left));
-- MSB+1 bit je izhodni prenos
-- preliv se postavi pri sestevanju, ko sta negativna
    a, b in je pozitiven rezultat ali
    -- v primeru ko sta pozitivna a, b in je
    negativen rezultat
v <= std_logic((sum_sig(sum_sig'left) and not
    a(a'left) and not b(b'left)) or
    (not sum_sig(sum_sig'left) and a(a'left) and
    b(a'left)));
z <= '1' when (sum_sig = ZERO) else '0';
-- primerjalnik enakosti (nor vrata v sintezi)
end arch_signed_ncvz_concatenate;
```

Načrtovanje digitalnih vezij

Predstavitev števil in
aritmetična vezja:

Seštevalniki

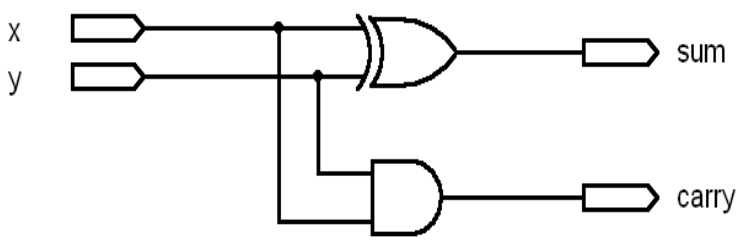
Polovični seštevalnik (HA)

x	0	0	1	1
+ y	+ 0	+ 1	+ 0	+ 1
c s	0 0	0 1	0 1	1 0

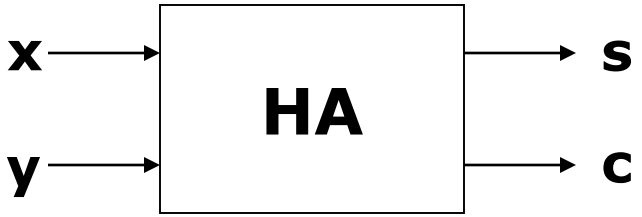
Prenos (Carry)

Vsota (Sum)

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

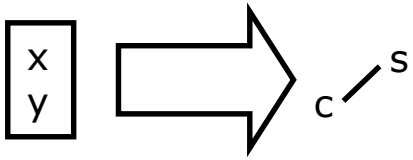


Polovični seštevalnik (Half Adder oz. HA)



$s = (x \oplus y)$ $c = x \cdot y$

Simbolični zapis:

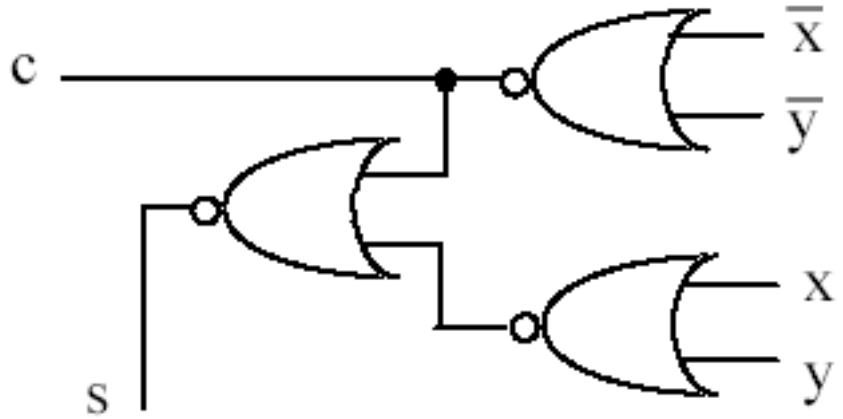


Druge izvedbe HA

NOR izvedba HA

$$c = \overline{\bar{x} + \bar{y}}$$

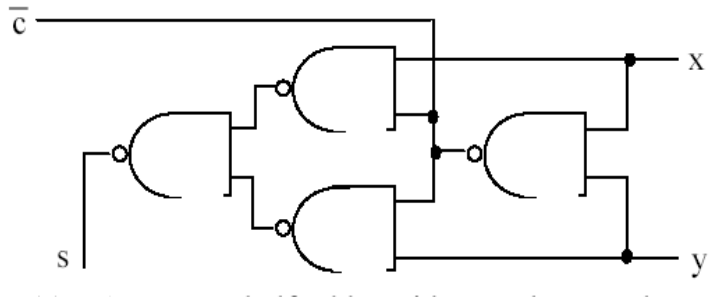
$$s = \overline{xy + \bar{x}\bar{y}}$$



NAND izvedba HA z negiranim prenosom

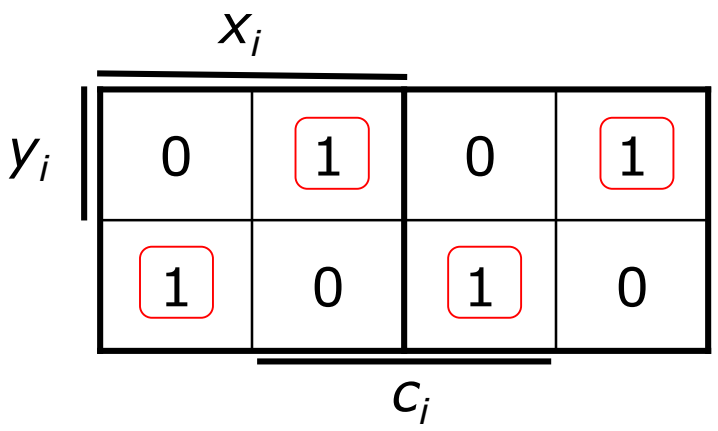
$$\bar{c} = \overline{xy}$$

$$s = x\bar{c} + y\bar{c} = \overline{\overline{x\bar{c}} \cdot \overline{y\bar{c}}}$$

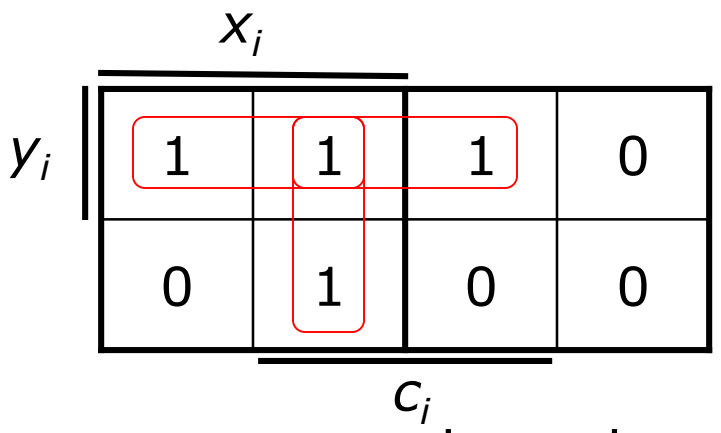


Polni seštevalnik (FA)

c_i	x_i	y_i	c_{i+1}	s_{i+1}
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



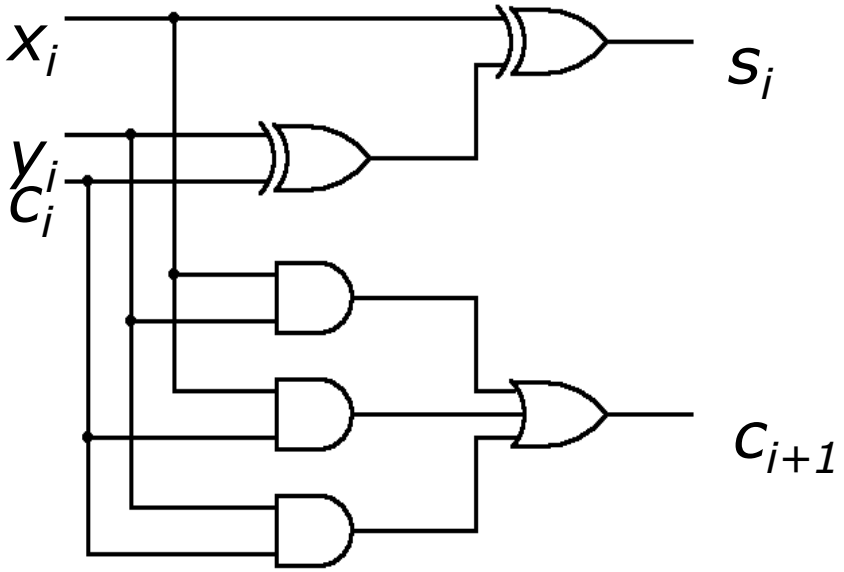
$$s_{i+1} = x_i \oplus y_i \oplus c_i$$

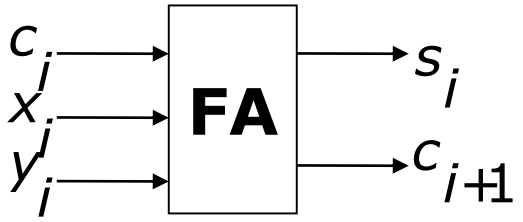


$$c_{i+1} = x_i \cdot y_i + y_i \cdot c_i + x_i \cdot c_i$$

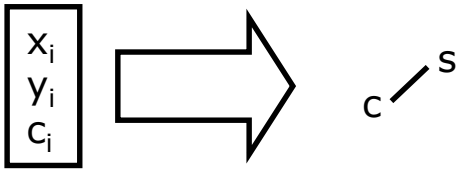
Vezje polnega seštevalnika (FA)

Neposredna izvedba:

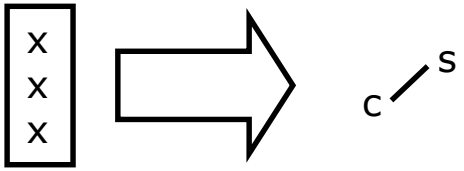




Simbolični zapis:



Včasih kar samo:



Polni seštevalnik v jeziku VHDL

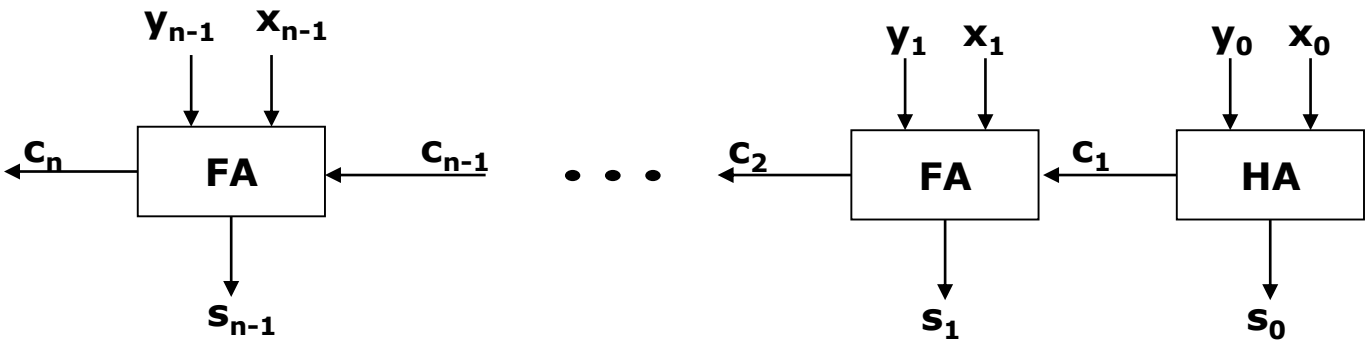
Osnovni enačbi iz analize: $s_{i+1}=x_i \oplus y_i \oplus c_i$

$$c_{i+1}=x_i \cdot y_i + y_i \cdot c_i + x_i \cdot c_i$$

```
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;
```

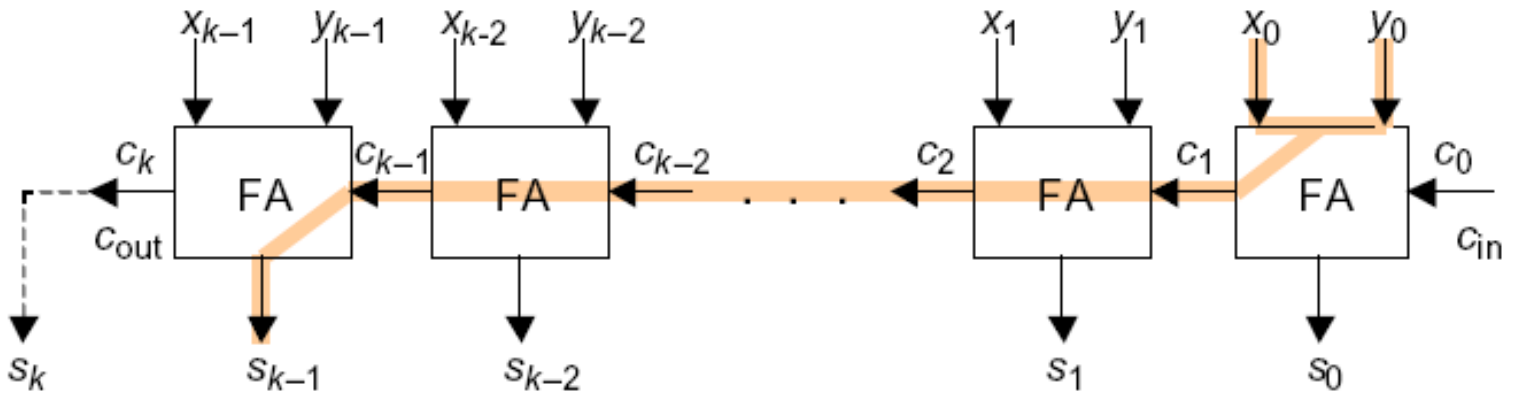
Ripple-carry (RC) seštevalnik

seštevanje n bitnih števil s=x+y



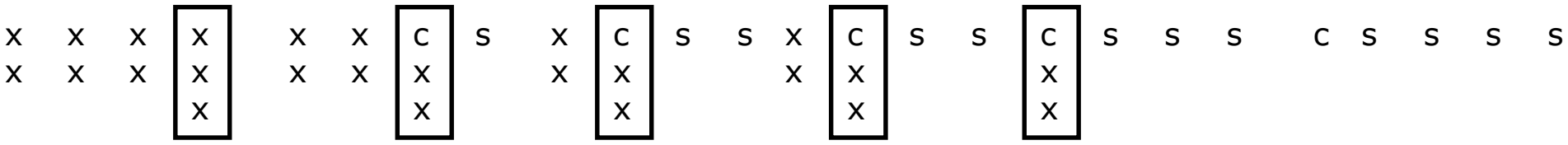
Kritična pot RC seštevalnika

$$T_{\text{ripple-add}} = T_{FA}(x,y \rightarrow c_{out}) + (k-2) \times T_{FA}(c_{in} \rightarrow c_{out}) + T_{FA}(c_{in} \rightarrow s)$$

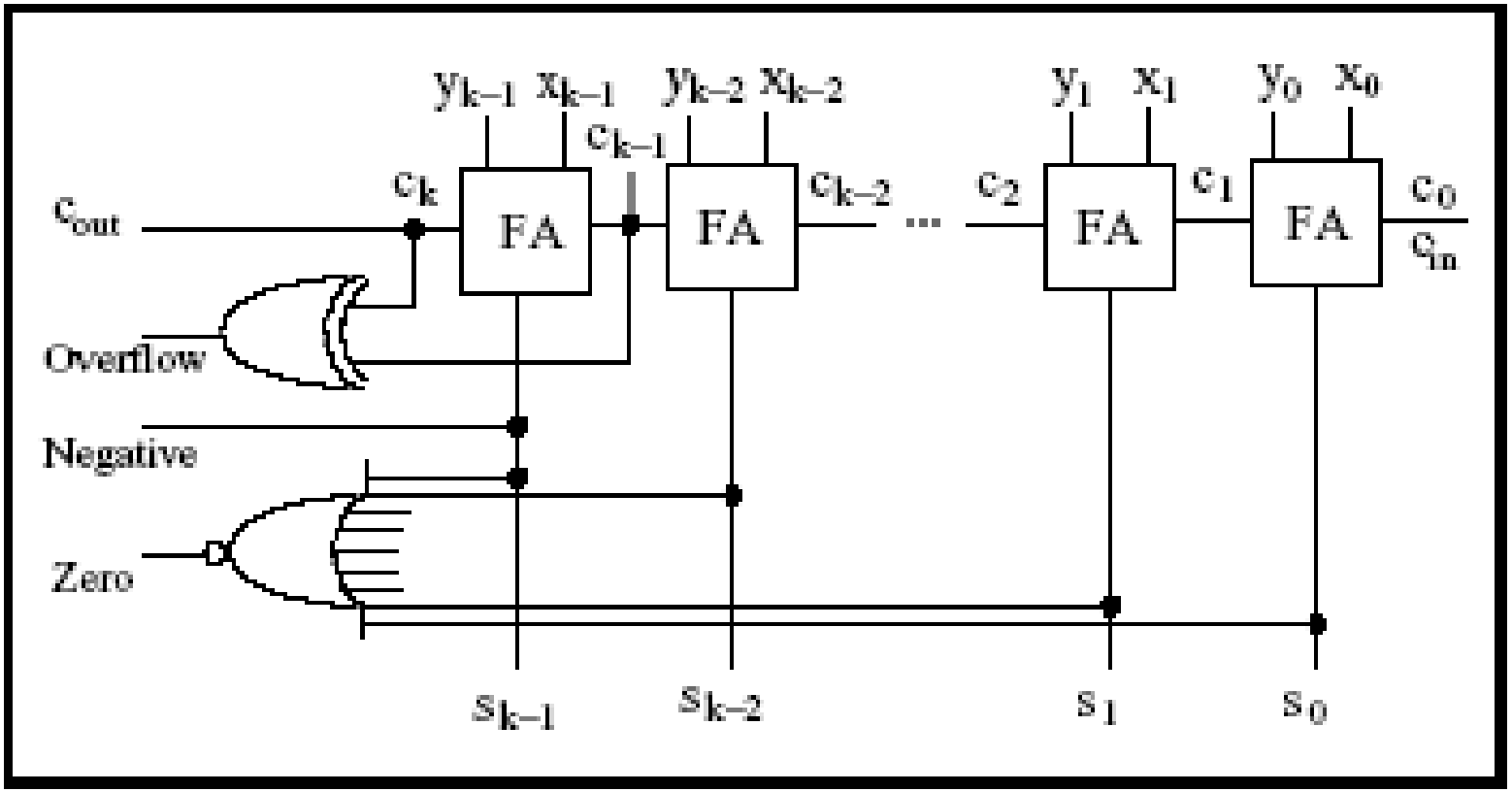


Seštevanje dveh 4-bitnih števil – simbolični zapis:

t=0 1·Δt 2·Δt 3·Δt 4·Δt rezultat

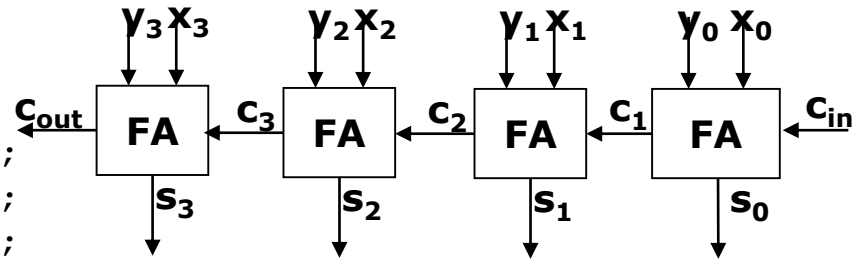


NCVZ biti RC seštevalnika



4-bitni RC seštevalnik v VHDL

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY adder4 IS
    PORT (
    Cin              : IN         STD_LOGIC ;
    x3, x2, x1, x0   : IN         STD_LOGIC ;
    y3, y2, y1, y0   : IN         STD_LOGIC ;
    s3, s2, s1, s0       : OUT STD_LOGIC ;
    Cout         : OUT STD_LOGIC
);
END adder4 ;
ARCHITECTURE arch OF adder4 IS
    SIGNAL c1, c2, c3 : STD_LOGIC ;
    COMPONENT fulladd
        PORT (   Cin, x, y : IN    STD_LOGIC ;
                 Cout      : OUT   STD_LOGIC );
    END COMPONENT ;
BEGIN
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 );
stage3: fulladd PORT MAP (Cin => c3, Cout => Cout, x => x3, y => y3, s => s3  ) ;
END arch ;
```

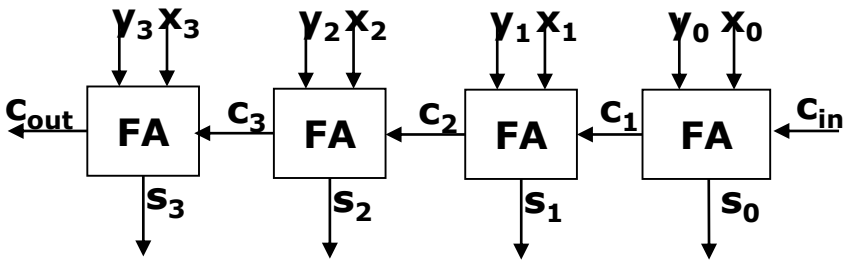


Komponente v VHDL

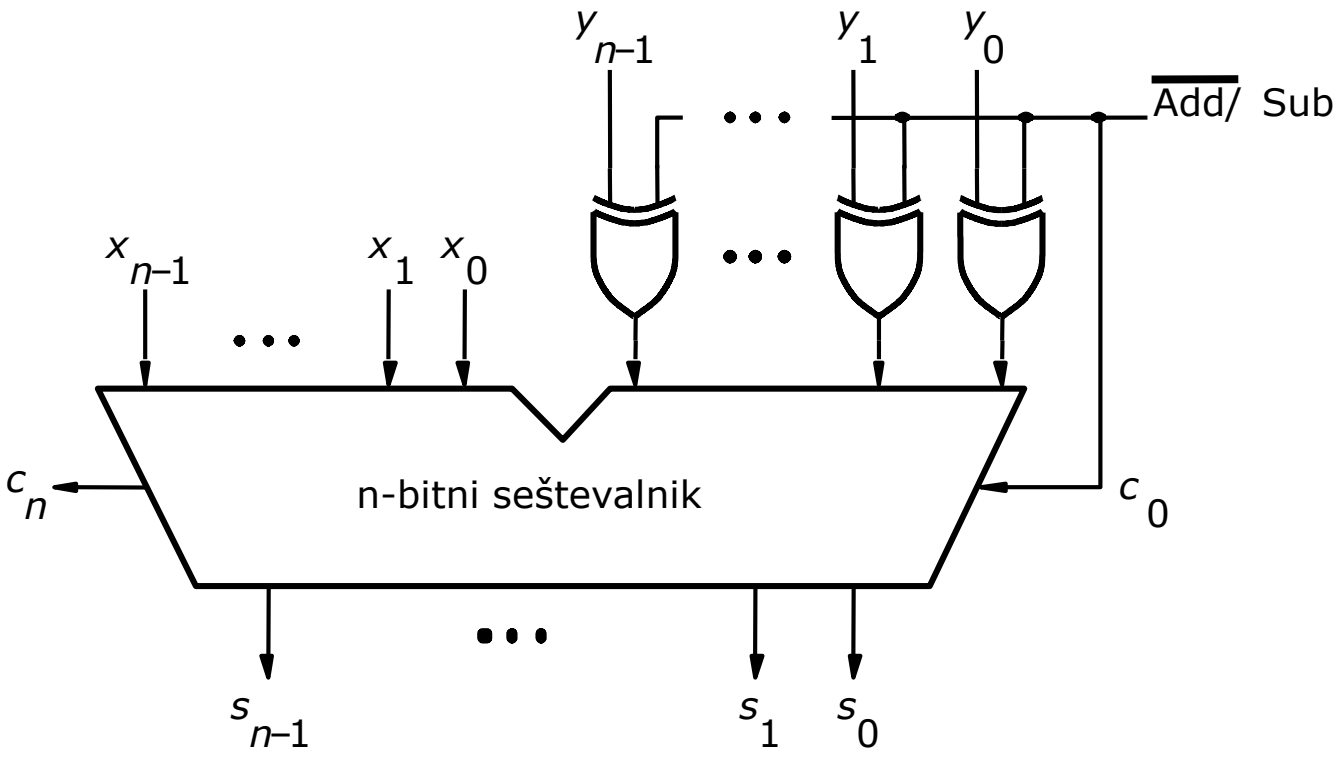
- **stage0: fulladd PORT MAP (Cin, x0, y0, s0, c1) ;**
  - Definira primer komponente fulladd z imenom stage0
  - Uporablja *položajno povezovanje* (*positional association)*
  - Vhodi in izhodi, navedeni v PORT MAP se pojavljajo v enakem zaporedju kot v stavku COMPONENT
- **stage3: fulladd PORT MAP (**
  Cin => c3, Cout => Cout, x => x3, y => y3, s => s3
) ;
  - Definira primer komponente fulladd z imenom stage3
  - Uporablja *imensko povezovanje* (*named association)*
  - Vhodi in izhodi, navedeni v PORT MAP so povezani z določenim poimenovanjem signala v stavku COMPONENT

Organizacija komponent v VHDL
Primer: 4-bitni RC seštevalnik

```
BEGIN
   C(0) <= Cin;
   Cout <= C(4);
   G1: FOR i IN 0 TO 3 GENERATE
       stages: fulladd PORT MAP (
             C(i), X(i), Y(i), S(i), C(i+1)) ;
   END GENERATE;
END arch;
```



Odštevalnik/seštevalnik

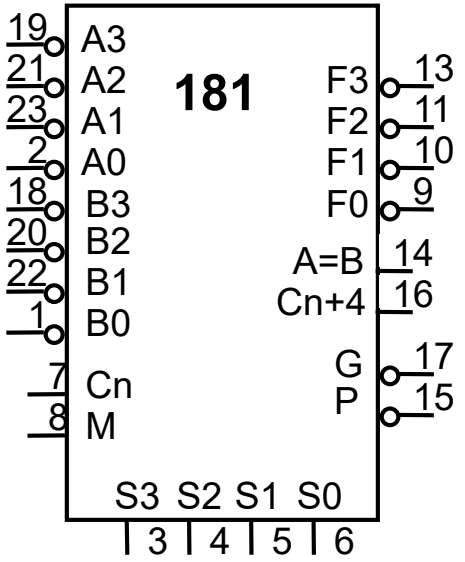


Načrtovanje digitalnih vezij

Aritmetično logična enota

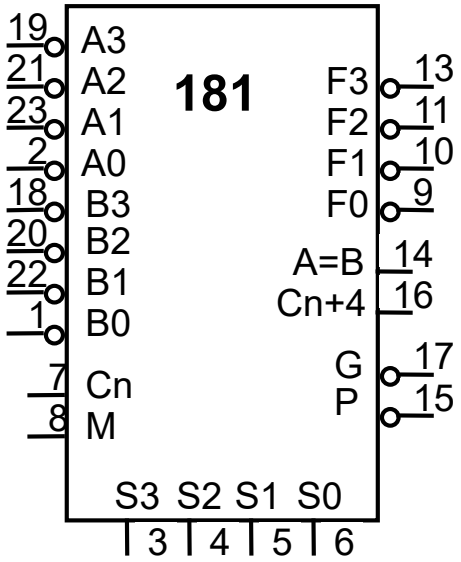
74181 - aritmetično logična enota

- Kombinacijsko vezje, ki izvaja različne aritmetične in logične operacije.
- Vhodi: A[], B[], C_n, S, M
- Izhodi: F, C_{n+4}, P, G, A==B
- M določa tip operacije:
  - M=0 za aritmetične in
  - M=1 za logične operacije.
- Vhodi S(3:0) določajo konkretno operacijo
- Realizira 48 različnih operacij.



74181 - aritmetično logična enota

$S_3S_2S_1S_0$	M=1	C=0	C=1
		M=0	
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



$F_i=f(A_i,B_i)$ i=0..3

Primer: M=1, S=1110

$F_0=A_0$ AND B_0

$F_1=A_1$ AND B_1

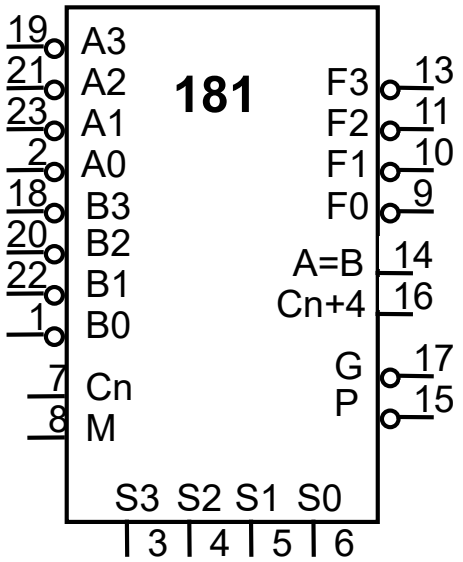
$F_2=A_2$ AND B_2

$F_3=A_3$ AND B_3

74181 - aritmetično logična enota

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 (F = A plus B plus 1) ➔ C_n=1, ne C_n=0.
  - Izhodni prenos invertiramo



Normalno seštevanje/odštevanje

	0011	3_{10}
+	0010	2_{10}
	0101	5_{10}

	0011	3_{10}
-	1110	$(-2)_{10}$
	0101	5_{10}

Seštevanje/odštevanje s 74181 in pozitivno logiko vhodov in izhodov

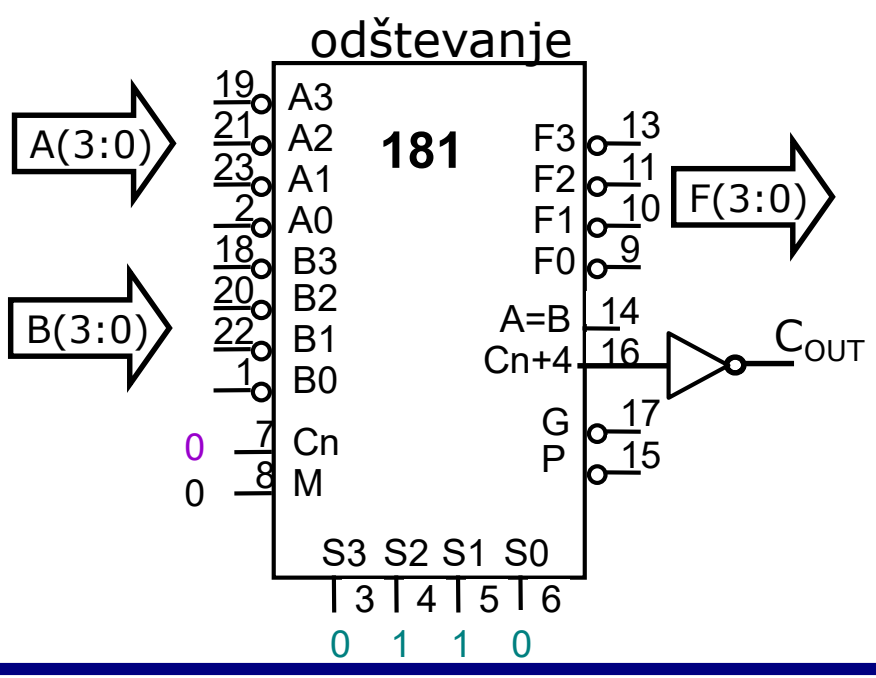
	1100	
+	1101	
+	1	C_n
1	1010	
(C_{n+4}) 0	0101	$= 5_{10}$

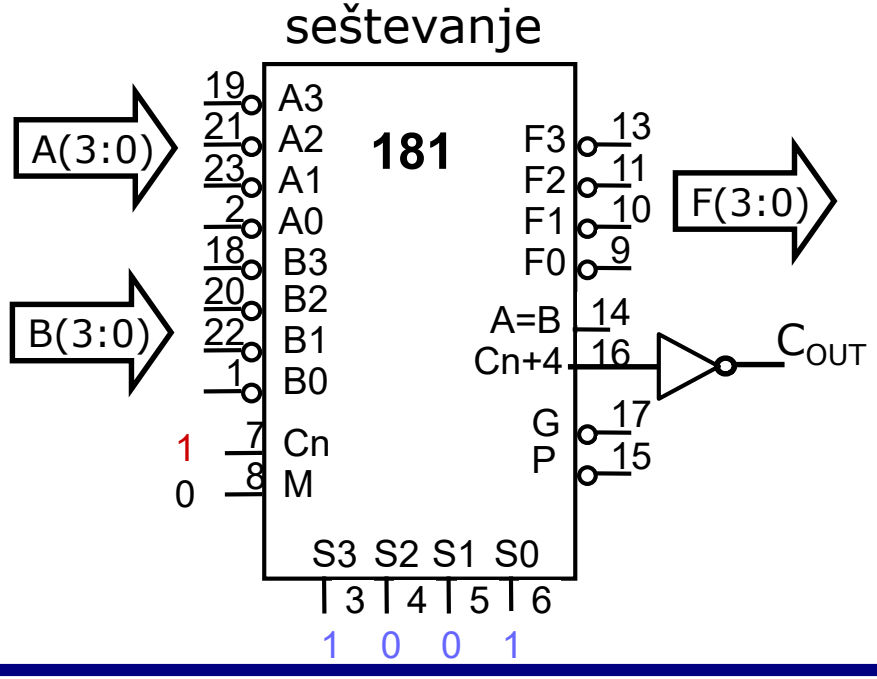
	1100	
-	0001	
0	1011	
(C_{n+4}) 1	0101	$= 5_{10}$

Seštevanje in odštevanje s 74181

Pri aritmetičnih funkcijah vzamemo negirane vrednosti C_n in C_{n+4}:

- Pri seštevanju vzamemo
  S=1001 (F = A plus B plus 1) ➔ C_n=1, ne C_n=0.
- Izhodni prenos C_{n+4} invertiramo.
- Pri odštevanju vzamemo
  S=0110 (F = A minus B) ➔ C_n=0, ne C_n=1.
- Izhodni prenos C_{n+4} invertiramo.





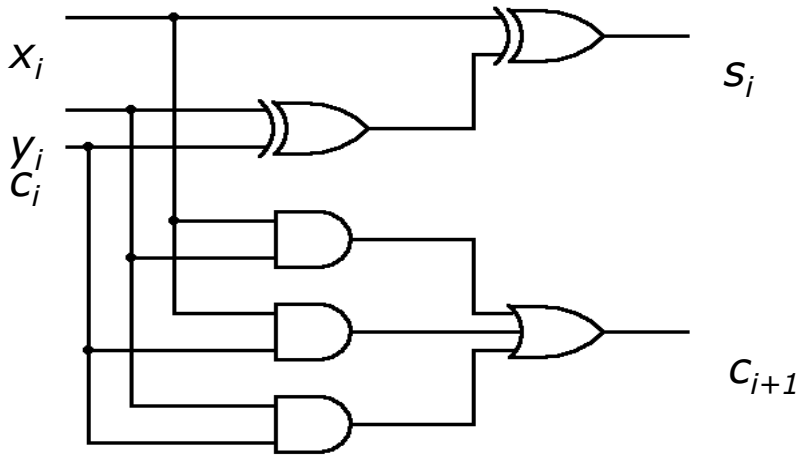
Načrtovanje digitalnih vezij

Predstavitev števil in
aritmetična vezja:

Hitre realizacije seštevalnikov

Hitrost delovanja vezja seštevalnika/odštevalnika

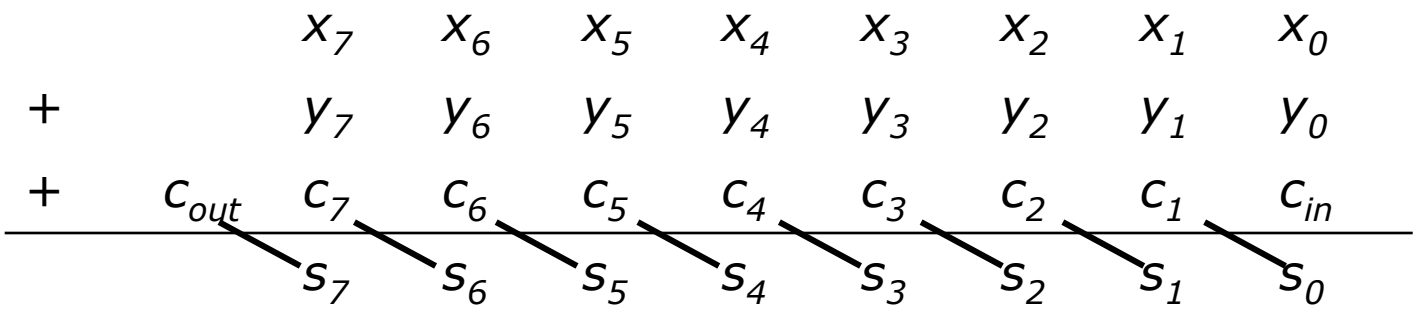
- Zanima nas največja časovna razlika (zakasnitev):
  - od tega da na vhode postavimo operanda *X* in *Y*
  - dokler niso na izhodu veljavni vsi biti vsote *S* in izhodni prenos, c_n
- Opazujmo seštevalnik izveden v RC izvedbi, ki ima vsak bit izveden takole:



zakasnitev kritične poti

kritična pot

Carry Look Ahead (CLA) seštevalnik



prenos z mesta	x_7 y_7	x_6 y_6	x_5 y_5	x_4 y_4	x_3 y_3	x_2 y_2	x_1 y_1	x_0 y_0	c_{in}
ni prenosa z nižjega mesta				1 1	0 X				0
c_4				1 X	1 1	0 X			0
c_3				1 X	1 X	1 1	0 X		0
c_2				1 X	1 X	1 X	1 1	0 X	0
c_1				1 X	1 X	1 X	1 X	1 1	0
c_{in}				1	1	1	1	1	1

Carry Look Ahead (CLA) seštevalnik

- Funkcijo izhodnega prenosa na i-ti stopnji lahko zapišemo kot

  $c_{i+1}=x_i \cdot y_i + x_i \cdot c_i + y_i \cdot c_i = x_i \cdot y_i + (x_i + y_i) \cdot c_i$
- Označimo $g_i = x_i y_i$ in $p_i = x_i + y_i$, tako da je $c_{i+1} = g_i + p_i \cdot c_i$

$g_i=1$ ko sta oba x_i **in** y_i enaka 1, ne glede na vhodni prenos c_i

- V tem primeru bo stopnja *i* zanesljivo tvorila izhodni prenos, zato funkciji *g* pravimo "funkcija tvorjenja prenosa" (***generate*** function)

$p_i=1$ ko sta x_i **ali** y_i enaka 1. Izhodni prenos se zgodi, če je vhodni prenos $c_i=1$

- Vhodni prenos (ki je 1) se širi preko stopnje *i*; zato funkciji *p* pravimo "funkcija širjenja prenosa" (***propagate*** function)

Carry Look Ahead (CLA) seštevalnik

- *Od prej velja* $g_i=x_i \cdot y_i$ in $p_i=x_i+y_i$, ter
  $c_{i+1}= g_i+p_i \cdot c_i$

  $c_{i+1}=x_i \cdot y_i+x_i \cdot c_i+y_i \cdot c_i=x_i \cdot y_i+(x_i+y_i) \cdot c_i=$
  $=x_i \cdot y_i+(x_i \cdot y'_i+x'_i \cdot y_i+x_i \cdot y_i) \cdot c_i$
  $=x_i \cdot y_i+(x_i \cdot y'_i+x'_i \cdot y_i+x_i \cdot y_i) \cdot c_i$

  Člen $x_i \cdot y_i$ *je že vsebovan v g funkciji, zato velja*
  $x+x \cdot y=x$ ➔ $x_i \cdot y_i+(x_i \cdot y_i) \cdot c_i$

  Od tod sledi, da je $\boldsymbol{g_i=x_i \cdot y_i}$ in $\boldsymbol{p_i=x_i \oplus y_i}$
  $=x_i \cdot y_i+(x_i \cdot y'_i+x'_i \cdot y_i) \cdot c_i=g_i+p_i \cdot c_i$

  Poleg tega imamo še funkcijo ***izničenja*** *prenosa (ang. annihilate):* $\boldsymbol{a_i=x'_i \cdot y'_i}$

Carry Look Ahead (CLA) seštevalnik

- Zapišimo izraz izhodnega prenosa za n-bitni seštevalnik

  Če je, $c_n = g_{n-1} + p_{n-1} \cdot c_{n-1}$

  in, $c_{n-1} = g_{n-2} + p_{n-2} \cdot c_{n-2}$

  potem, $c_{n-2} = g_{n-1} + p_{n-1} \cdot (g_{n-2} + p_{n-2} \cdot c_{n-2})$

  $c_{n-3} = g_{n-1} + p_{n-1} \cdot g_{n-2} + p_{n-1} \cdot p_{n-2} \cdot c_{n-2}$

- Če bi to vstavljanje ponovili na preostalih stopnjah, bi dobili

$$c_n = g_{n-1} + p_{n-1} \cdot g_{n-2} + p_{n-1} \cdot p_{n-2} \cdot g_{n-3} + \ldots + p_{n-1} \cdot p_{n-2} \ \ldots$$
$$\ldots + p_1 \cdot g_0 + p_{n-1} \cdot p_{n-2} \ldots p_0 c_0$$

Carry Look Ahead (CLA) seštevalnik

Prenos, ki se tvori na stopnji *n*-2, in se širi prek preostalih stopenj

Prenos, ki se tvori na stopnji n-2, in se širi prek preostalih stopenj

$$c_n = g_{n-1} + p_{n-1}g_{n-2} + p_{n-1}p_{n-2}g_{n-3} + \ldots + p_{n-1}p_{n-2}\ldots p_1 g_0 + p_{n-1}p_{n-2}\ldots p_0 c_0$$

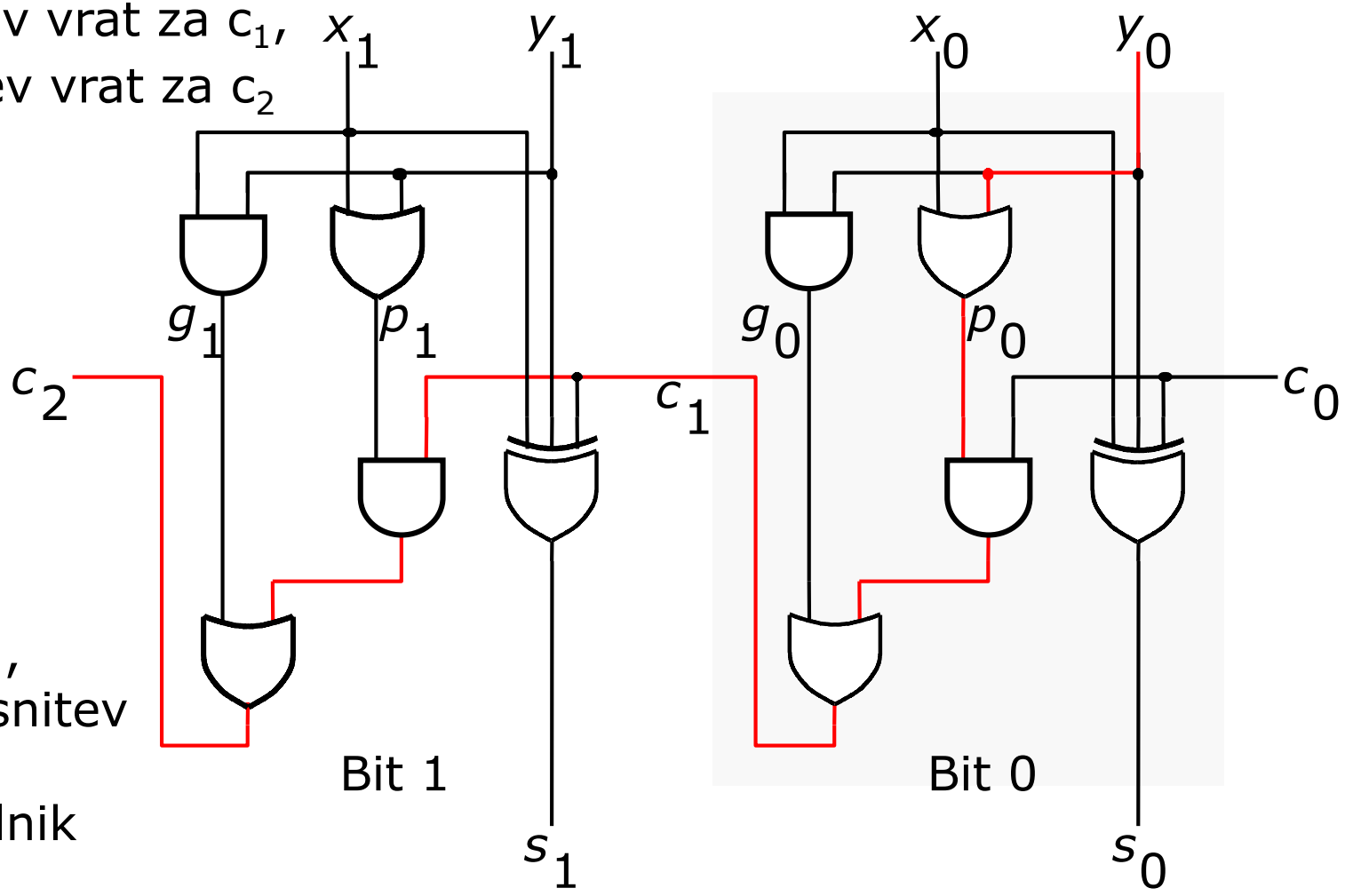
Prenos, ki se tvori na zadnji stopnji

Prenos, ki se tvori na stopnji n-3, in se širi prek preostalih stopenj

Vhodni prenos c_0 ki se širi preko vseh stopenj

Kritična pot RC seštevalnika

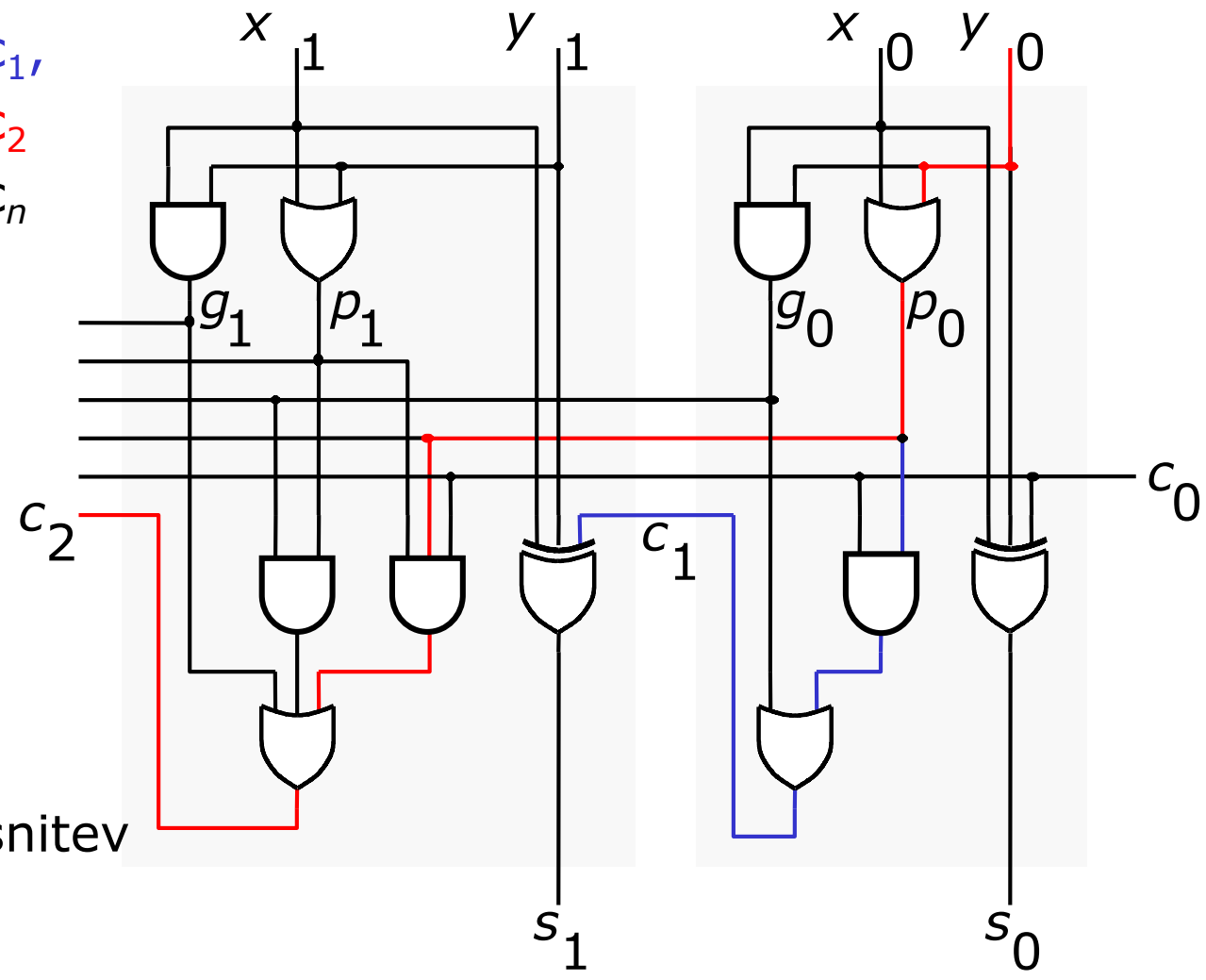
3 zakasnitev vrat za c_1,
5 zakasnitev vrat za c_2



V splošnem,
$2n+1$ zakasnitev
za n-bitni
RC seštevalnik

Kritična pot CLA seštevalnika

3 zakasnitev vrat za c_1,
3 zakasnitev vrat za c_2
3 zakasnitev vrat za c_n



Skupna zakasnitev
n-bitnega
CLA seštevalnika je 4
zakasnitve vrat:
Vsi g_i in p_i, eno zakasnitev
Vsi c_i, dve zakasnitvi
+ zakasnitev za s_i

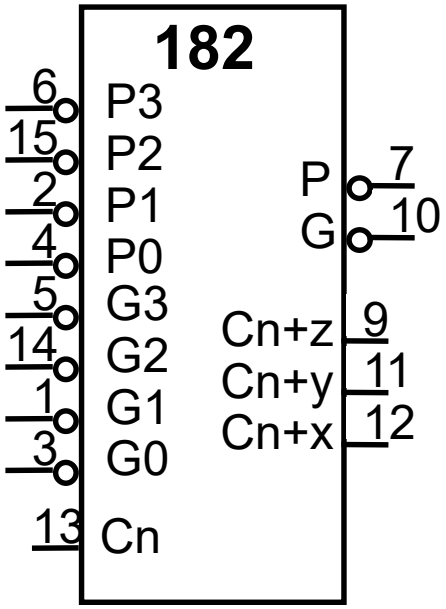
Carry Look Ahead Generator (CLAG)

- Realizira tvorbo izhodnih prenosov C_{n+x}, C_{n+y}, C_{n+z} na osnovi vhodnih funkcij g_0, p_0, g_1, p_1, g_2, p_2 *in* g_3, p_3 ter vhodnega prenosa C_n po enačbah:

$$P = P_3 \cdot P_2 \cdot P_1 \cdot P_0$$

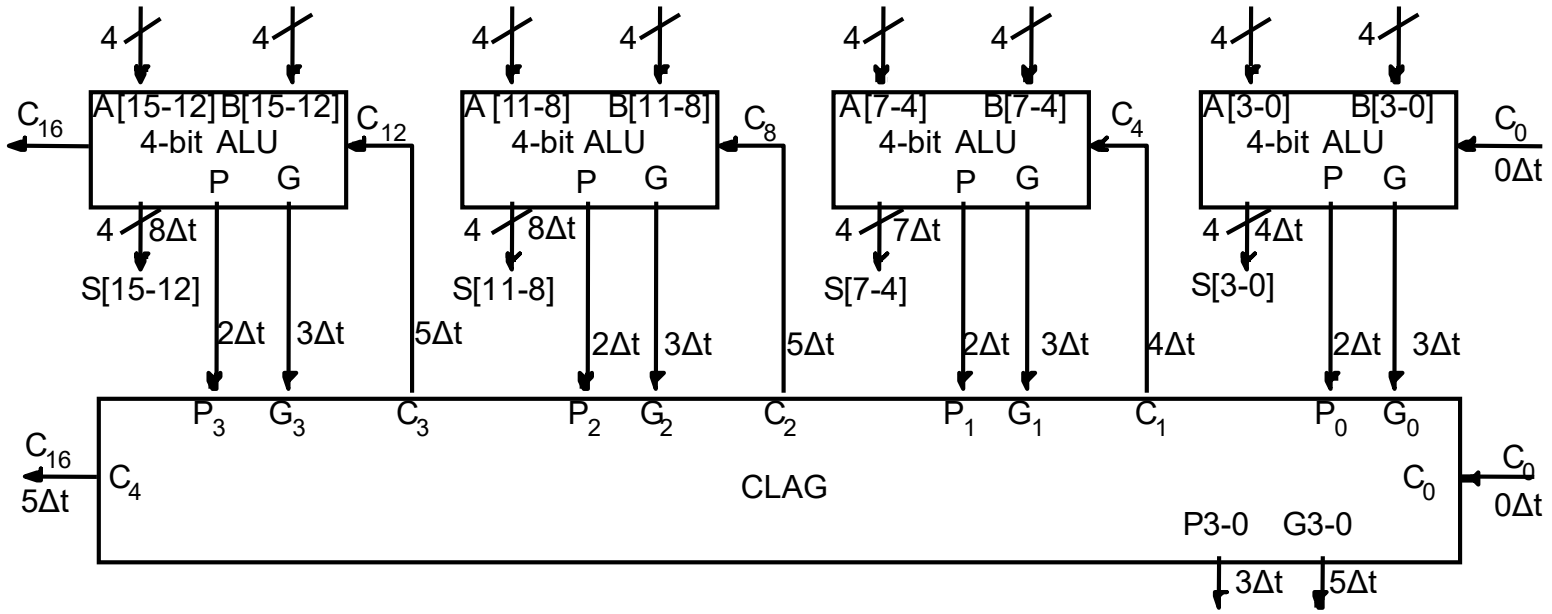
$$G = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0$$

$$C_{n+x} = P_0 \cdot (G_0 + C_n) = G_0 + P_0 \cdot C_n$$
$$C_{n+y} = P_1 \cdot (G_1 + P_0 \cdot (G_0 + C_n)) = G_1 + P_1 \cdot C_{n+x}$$
$$C_{n+z} = P_2 \cdot (G_2 + P_1 \cdot (G_1 + P_0 \cdot (G_0 + C_n))) = G_2 + P_2 \cdot C_{n+y}$$
$$C_{out} = C_{n+4} = G_3 + P_3 \cdot C_{n+z}$$



- Primeren je za neposredno uporabo s 74181 (negirani izhodi in vhodi)

Dvonivojska vezava ALU in CLAG



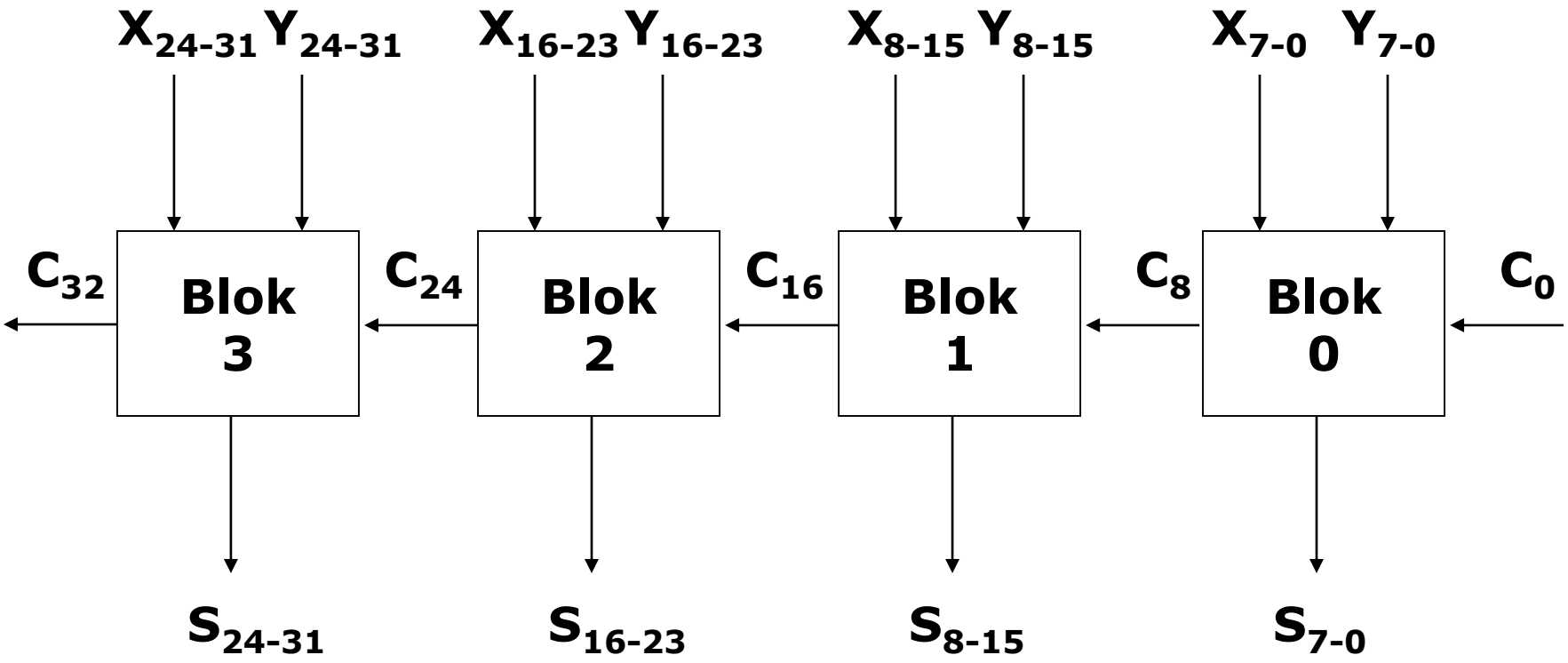
4 bit ALU imajo interno CL logiko

Drugi nivo CLAG razširi CL logiko na 16 bitov

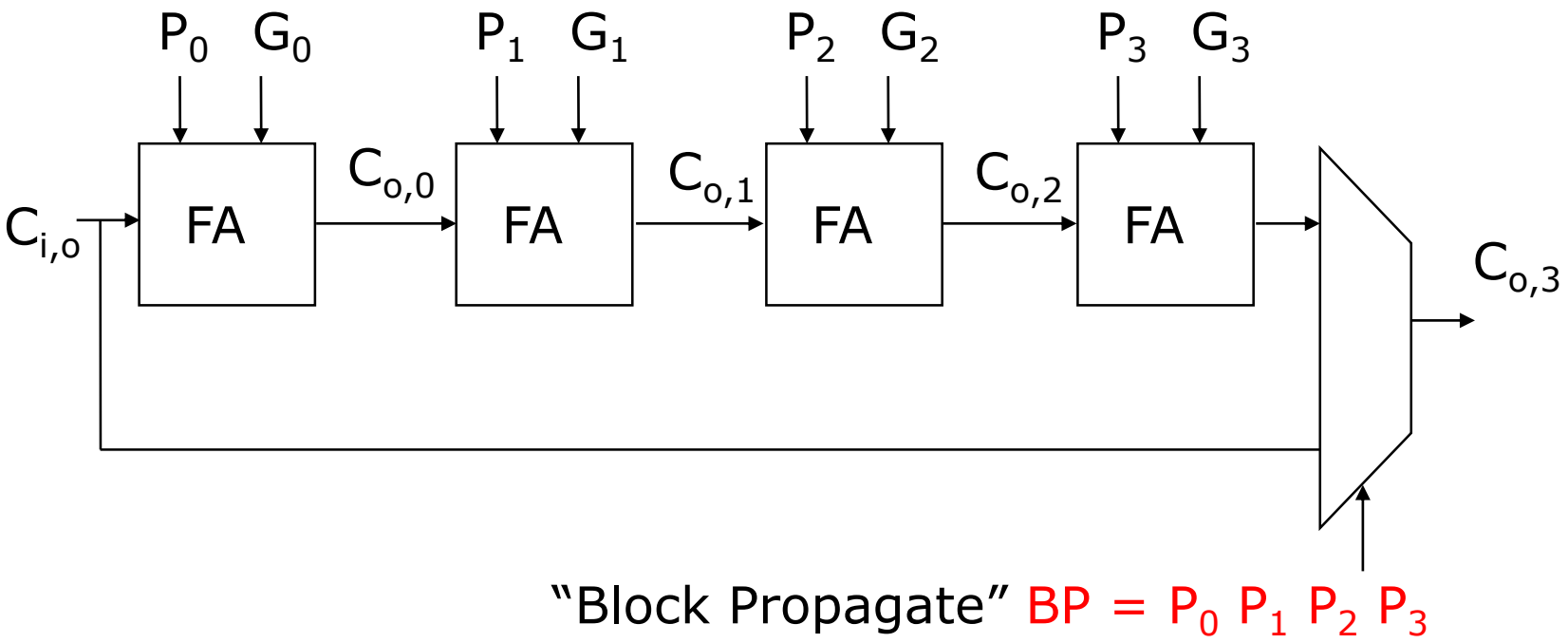
Načrtovanje 32-bitnega seštevalnika

- Seštevalnik razdelimo v 4 bloke tako da
  - Biti $b_7 \ldots b_0$ so v bloku 0
  - Biti $b_{15} \ldots b_8$ so v bloku 1
  - Biti $b_{23} \ldots b_{16}$ so v bloku 2
  - Biti $b_{31} \ldots b_{24}$ so v bloku 3
- Vsak blok načrtujemo kot 8-bitni CLA seštevalnik
  - Izhodni prenosi teh štirih blokov so c_8, c_{16}, c_{24}, in c_{32}
- Uporabljata se dva osnovna pristopa k povezovanju teh štirih blokov
  - RC povezava med bloki
  - CLA vezje na drugem nivoju

RC povezava med bloki

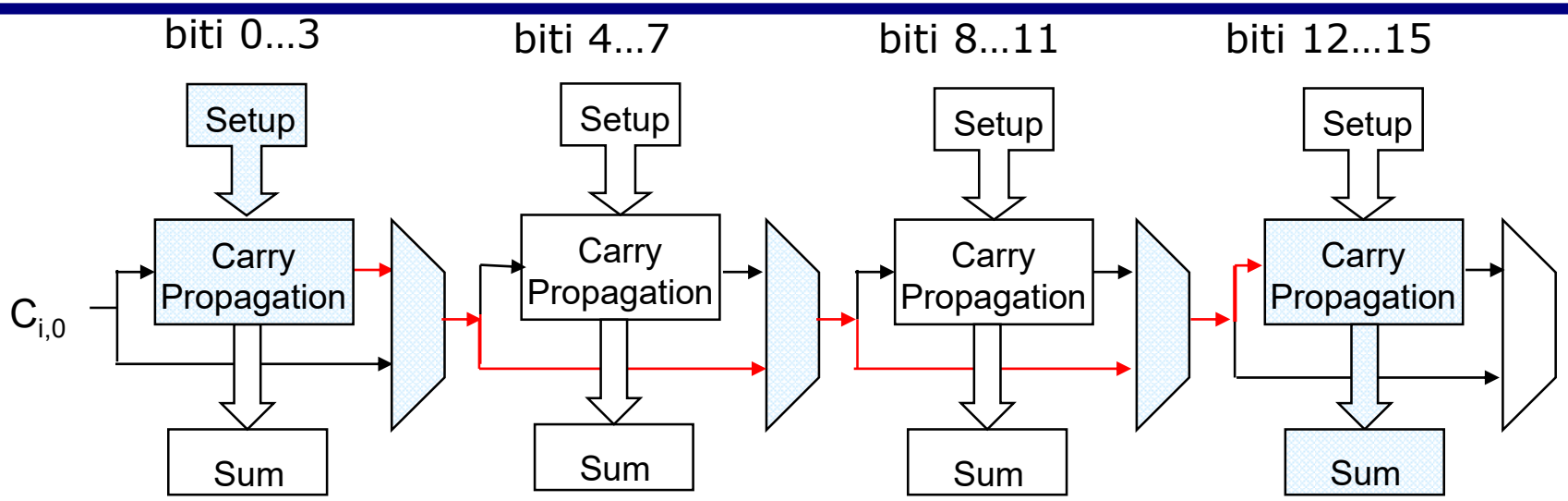


Carry skip seštevalnik



Če je BP = 1, potem lahko preskočimo $C_{o,3} = C_{i,0}$, sicer se prenos znotraj bloka tvori (G_i=1) ali pa se izniči (a_i=1, annihilate).

Carry skip seštevalnik



Worst-case delay → prenos iz bita 0 na bit 15.
Prenos z mesta 0 vzvalovi skozi mesta 1,2,3, nato preskoči mesta 4-11, in vzvalovi skozi mesta 12-15

$$T_p = t_{setup} + M\, t_{carry} + (N/M - 2)\, t_{skip} + (M-1)\, t_{carry} + t_{sum}$$

Carry skip seštevalnik

- Naj ima mesto RC stopnje enako zakasnitev (t_{carry}) kot stopnja preskoka (t_{skip}).
- N = število mest seštevalnika, M= velikost RC stopnje
- Zakasnitev carry skip seštevalnika je potem:

$$T_{CSkA} = \underbrace{1}_{t_{setup}} + \underbrace{M}_{\substack{\text{RC} \\ \text{prvi blok}}} + \underbrace{(N/M-2)}_{\text{preskokov}} + \underbrace{(M-1)}_{\substack{\text{RC} \\ \text{zadnji blok}}} + \underbrace{1}_{t_{sum}}$$

$$= 2M + (N/M) - 1$$

- Optimalna velikost RC stopnje (M) je:

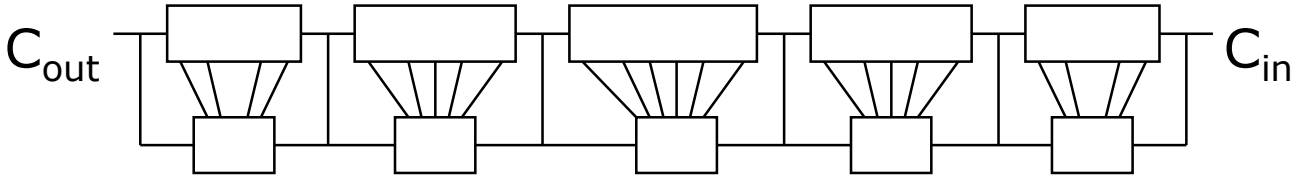
$$dT_{CSkA}/dM = 0 \Rightarrow \sqrt{(N/2)} = M_{opt}$$

- Optimalna zakasnitev:

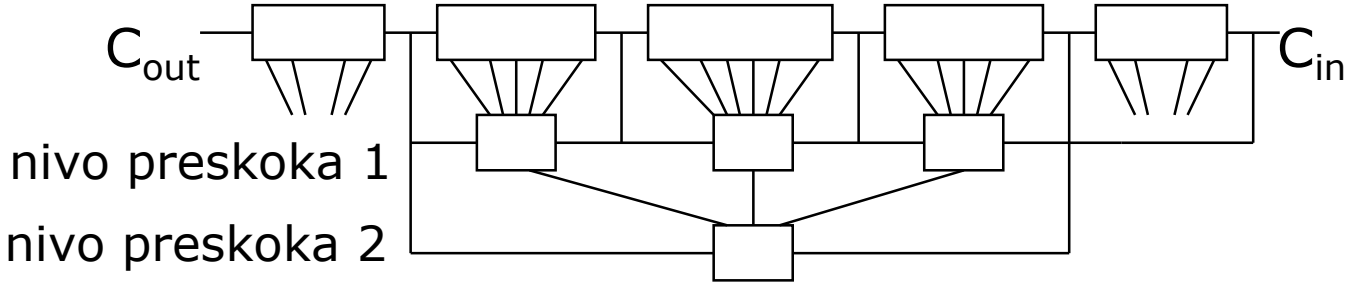
$$T_{CSkA} = 2(\sqrt{(2N)}) - 1$$

Carry skip seštevalnik

- **Spremenljiva velikost RC stopnje**



- **Večnivojska izvedba logike preskakovanja**



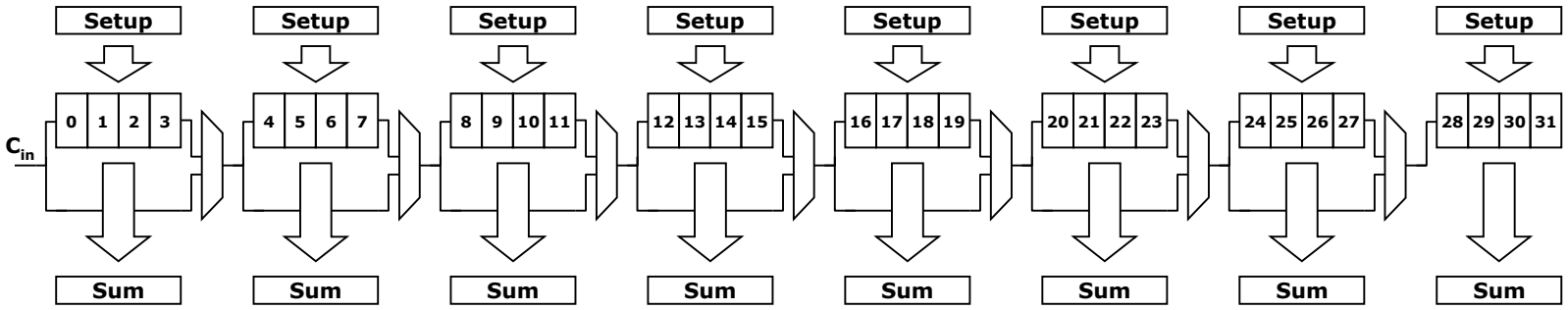
Carry skip seštevalnik

- **RC bloki enake velikosti**

  Če so vse RC stopnje enako velike, bodo zadnji RC bloki končali z izračunom, nato pa čakali, da se prenos razširi preko vseh multiplekserjev do njih.

  (Primer. 32-bitni carry-skip seštevalnik)

  Izhodni prenos bitov 4-7 je na voljo istočasno kot izhodni prenos bitov 0-3, zato bo drugi blok čakal, da se prenos razširi preko prvega izbiralnika.



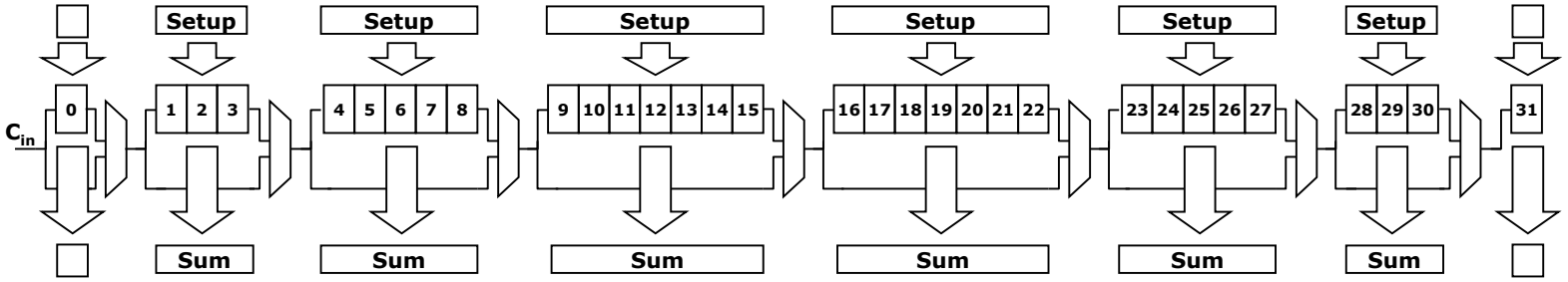
Carry skip seštevalnik

- **RC bloki različne velikosti**

  Za pohitritev lahko spreminjamo velikost RC bloka glede na izračun kritične poti.

  Prvi blok naj bo zato najmanjši, nato pa naj se velikost povečuje.

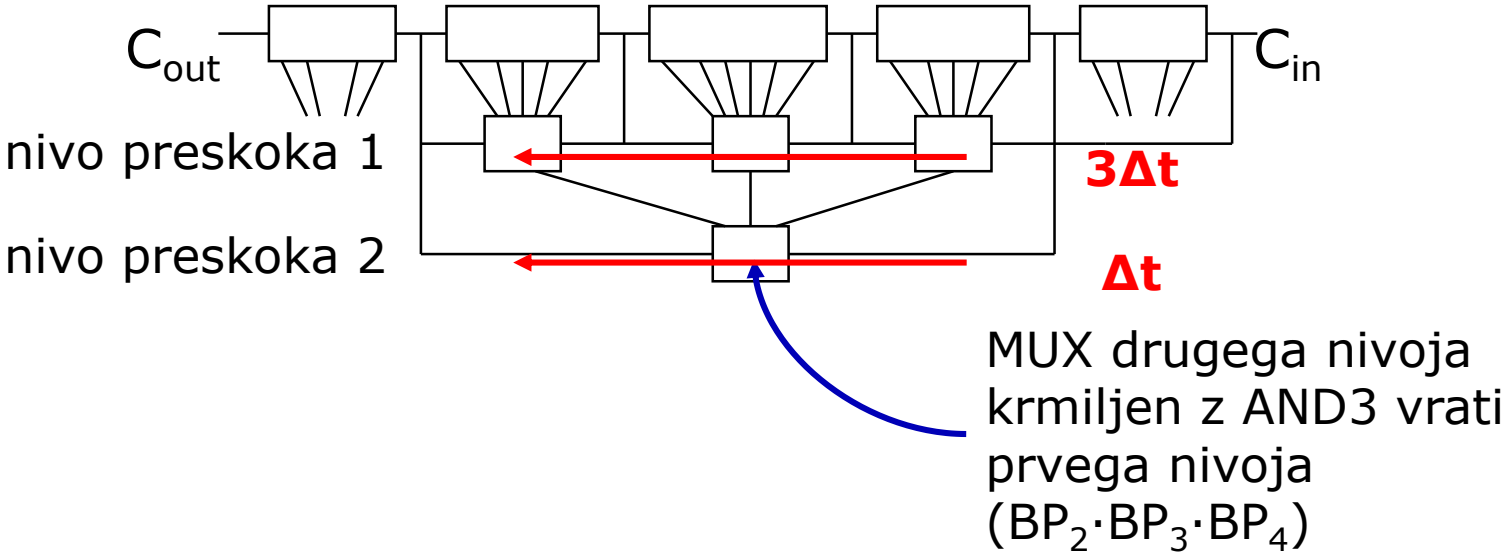
  Kritična pot vsebuje tudi zadnji RC blok, katerega zakasnitev naj bo najmanjša, zato naj se od sredine seštevalnika velikost zmanjšuje.



Carry skip seštevalnik

- **Dvonivojski carry skip seštevalnik**

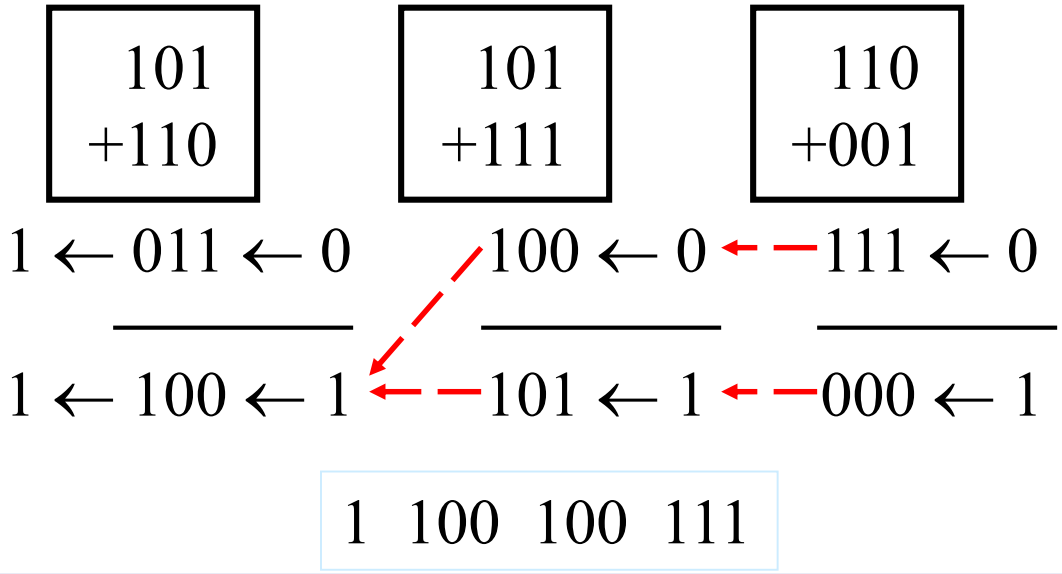
  Če tvorimo AND funkcijo izhodov več prvih nivojev, dobimo naslednji nivo.



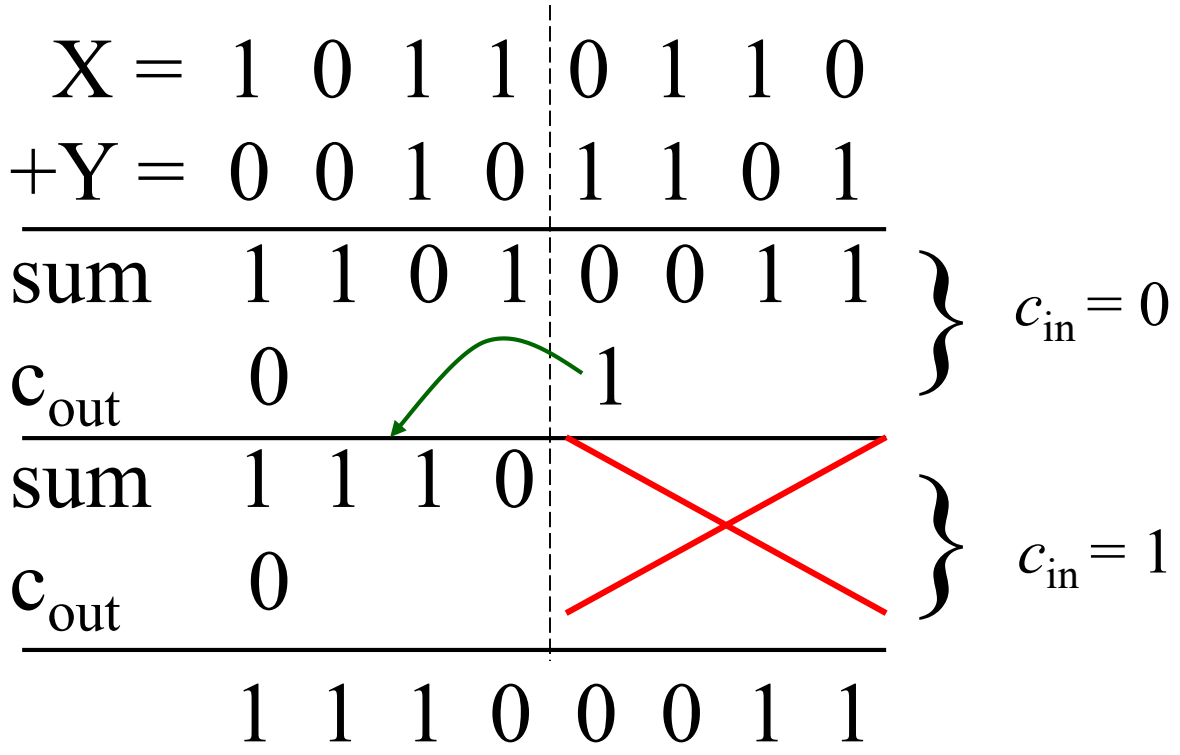
Carry Select seštevalnik

•Primer seštevanja:

$$\begin{array}{r|r|r} 101 & 101 & 110 \\ +\ 110 & 111 & 001 \end{array}$$

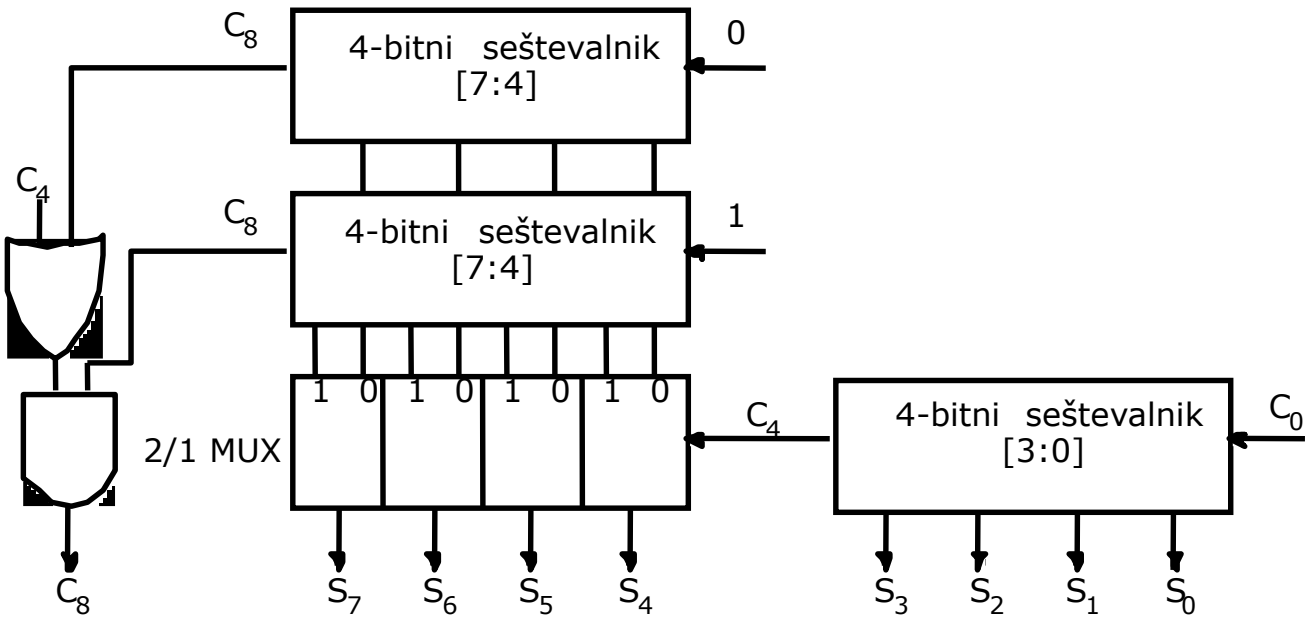


Carry Select seštevalnik



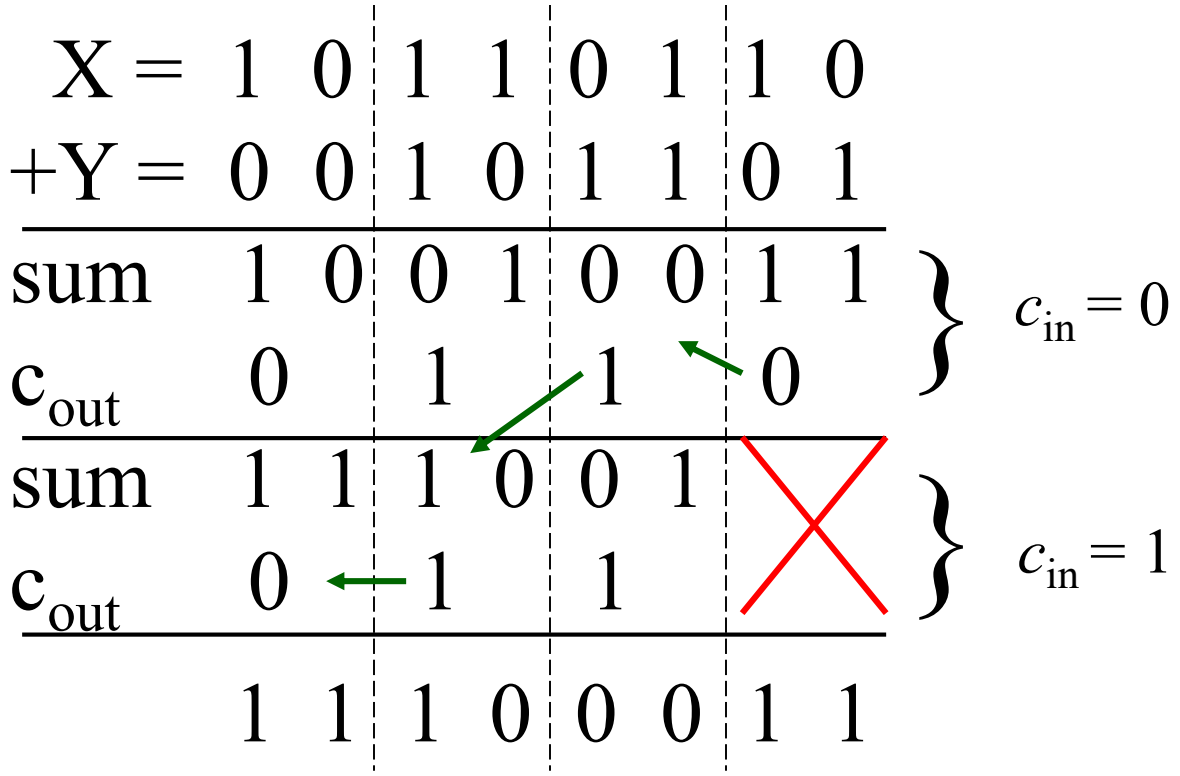
- $c_{in} \neq 1$ za spodnja 4 mesta
- Naslednja stopnja čaka $4\Delta t$ za izbiro rezultata
- Delitev lahko nadaljujemo na manjše kose (2)

Carry Select seštevalnik



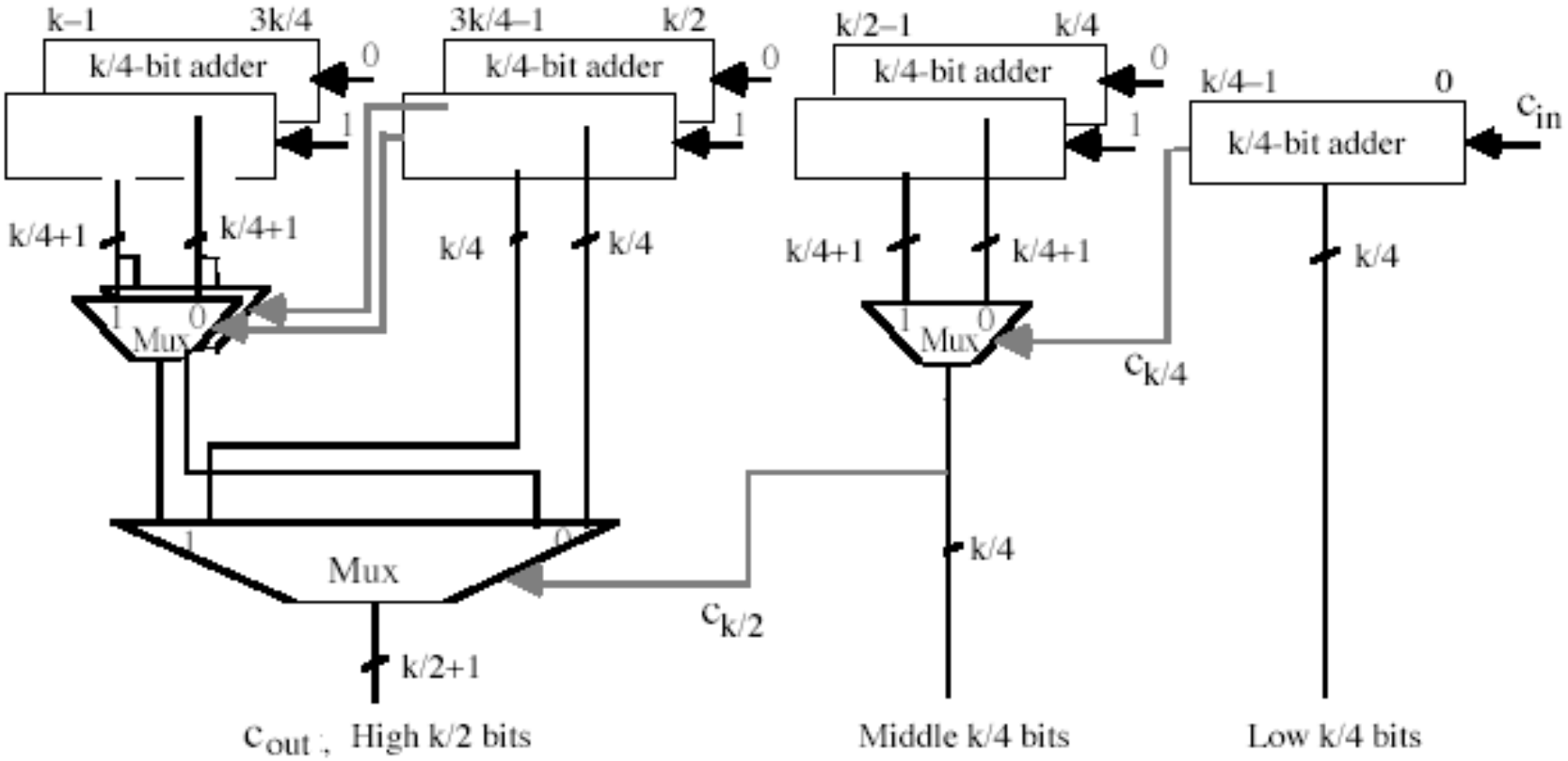
Enonivojski k-bitni carry-select seštevalnik sestavljen iz k/2-bitnih seštevalnikov

Carry Select seštevalnik



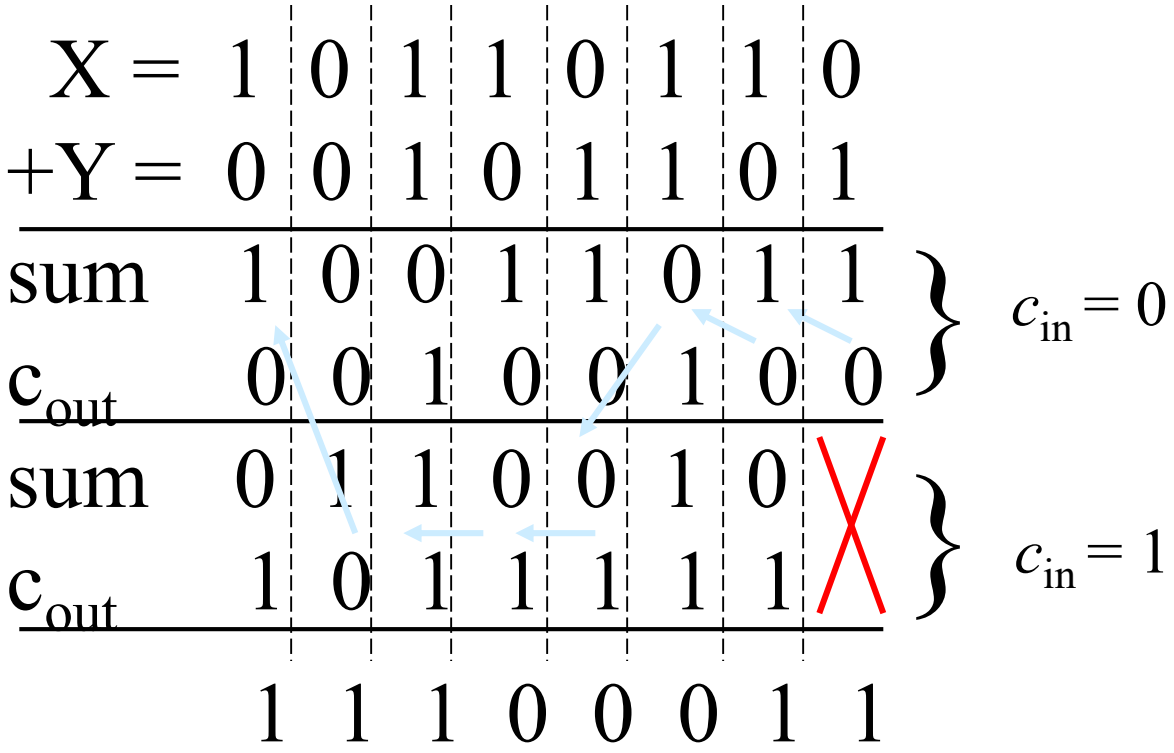
- $c_{in} \neq 1$ za spodnji 2 mesti
- Naslednja stopnja čaka $2\Delta t$ za izbiro rezultata
- Delitev lahko nadaljujemo še na manjše kose (1)

Carry Select seštevalnik



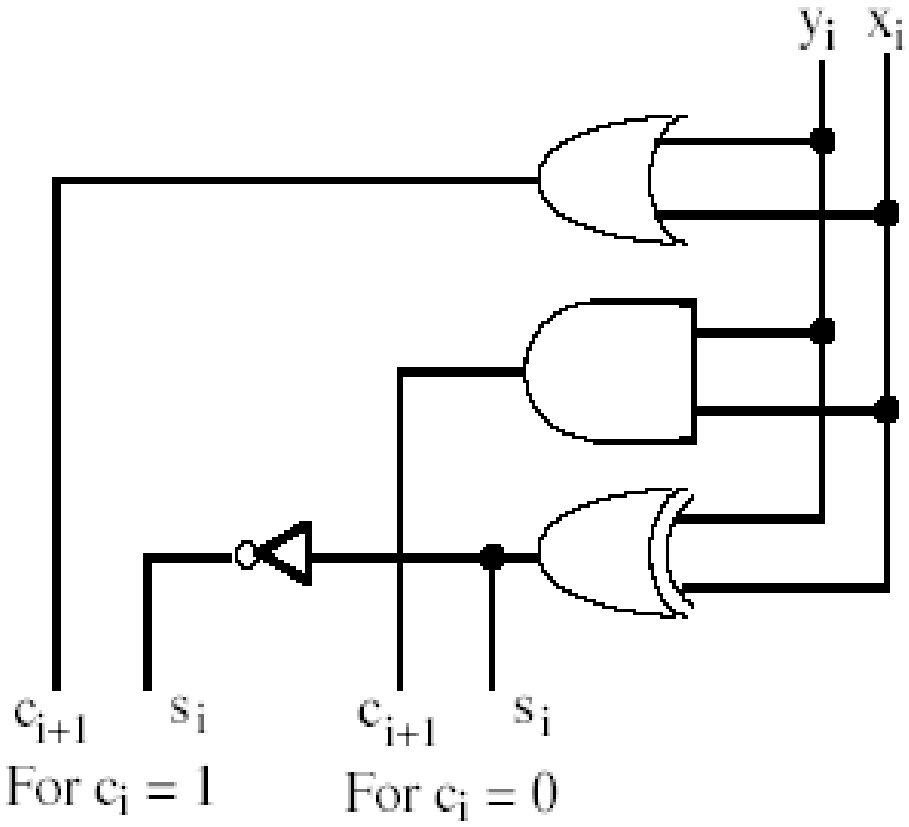
Dvonivojski k bitni carry-select seštevalnik, sestavljen iz k/4 bitnih seštevalnikov

Carry Select seštevalnik



- $c_{in} \neq 1$ za LSB mesto
- Naslednja stopnja čaka Δt za izbiro rezultata
- Delitev ne moremo več nadaljevati (Conditional sum)

Conditional sum seštevalnik



Začetni nivo conditional sum seštevalnika

Conditional sum seštevalnik

	i	7	6	5	4	3	2	1	0
	x_i	1	0	1	1	0	1	1	0
	y_i	0	0	1	0	1	1	0	1
Step 1	s_i^0	1	0	0	1	1	0	1	1
	c_{i+1}^0	0	0	1	0	0	1	0	0
	s_i^1	0	1	1	0	0	1	0	
	c_{i+1}^1	1	0	1	1	1	1	1	
Step 2	s_i^0	1	0	0	1	0	0	1	1
	c_{i+1}^0	0		1		1		0	
	s_i^1	1	1	1	0	0	1		
	c_{i+1}^1	0		1		1			
Step 3	s_i^0	1	1	0	1	0	0	1	1
	c_{i+1}^0	0				1			
	s_i^1	1	1	1	0				
	c_{i+1}^1	0							
Result		1	1	1	0	0	0	1	1

Carry select seštevalnik

- Imamo k=8 bitni "carry select" seštevalnik, izveden v dveh 4-bitnih delih
- Vsak 4-bitni izveden kot CLA seštevalnik
  - Rabi 4Δt za izračun vsote in 3Δt za izračun izhodnega prenosa
- MUX 2/1 ima še 2Δt zakasnitve
- 8-bitna vsota veljavna po: 4Δt+ 2Δt=6Δt
- V primerjavi s 7Δt zakasnitev v CLA izvedbi in 16Δt v RC izvedbi.

Splošni izraz za zakasnitev "conditional sum" seštevalnika:
$T(k)=T(k/2)+1+T(1)=\log_2(k)+T(1)$

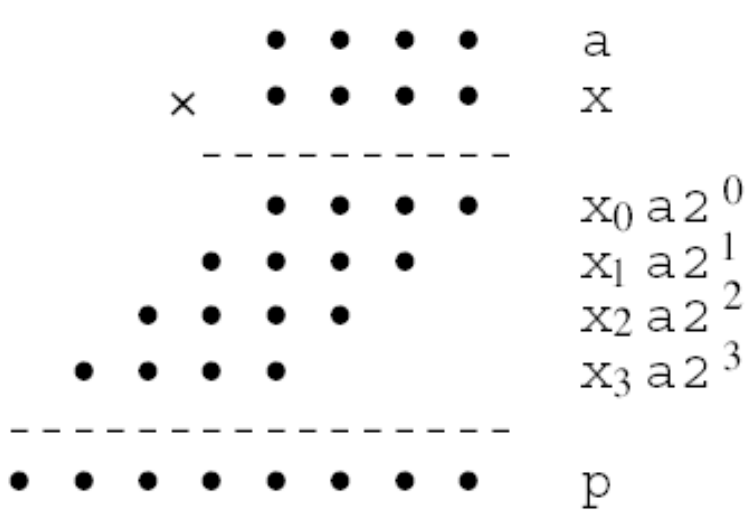
Pohitritve seštevanja

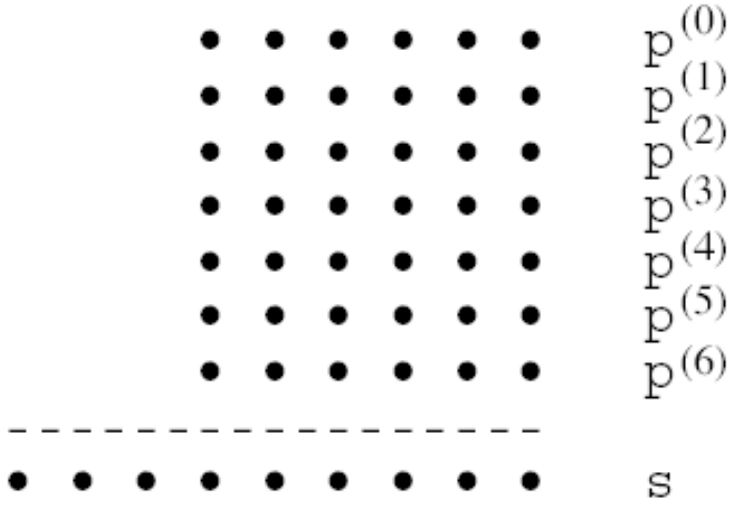
- Rajši zaznavamo konec širjenja prenosa kot pa čakamo največjo možno zakasnitev
- Mehanizmi pohitritve
  - carry look-ahead
  - carry select …
- Omejimo širjenje prenosa za majhno število bitov.
- Izničimo širjenje prenosa s pomočjo redundantnih predstavitev števila
- Seštevamo več operandov (drevesni seštevalniki)

Načrtovanje digitalnih vezij

Vzporedno seštevanje več operandov
(Paralelni števniki)

Uporaba vzporednega seštevanja operandov





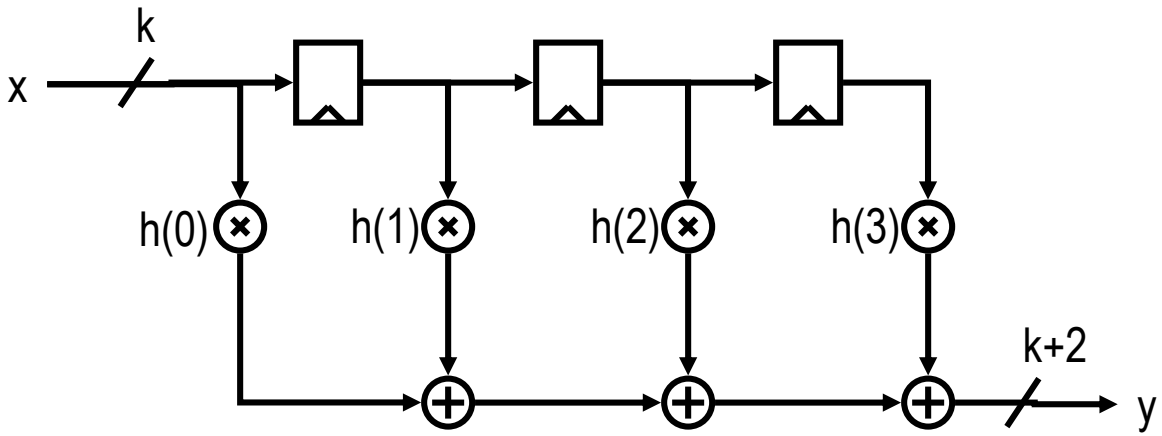
Množenje

$p=a \cdot x$

Skalarni produkt

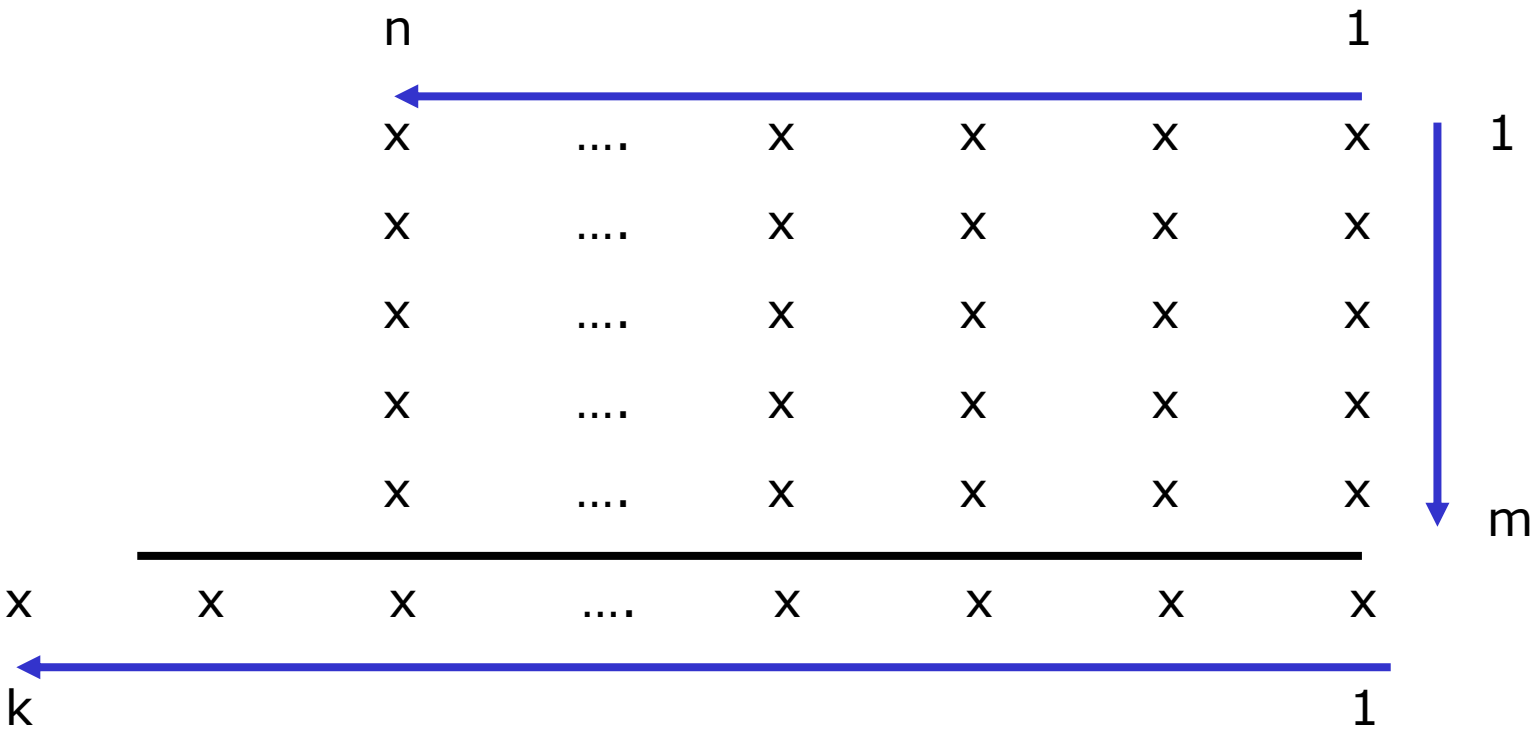
$$s = \sum_{i=0}^{n-1} x^{(i)} y^{(i)} = \sum_{i=0}^{n-1} p^{(i)}$$

Uporaba vzporednega seštevanja operandov

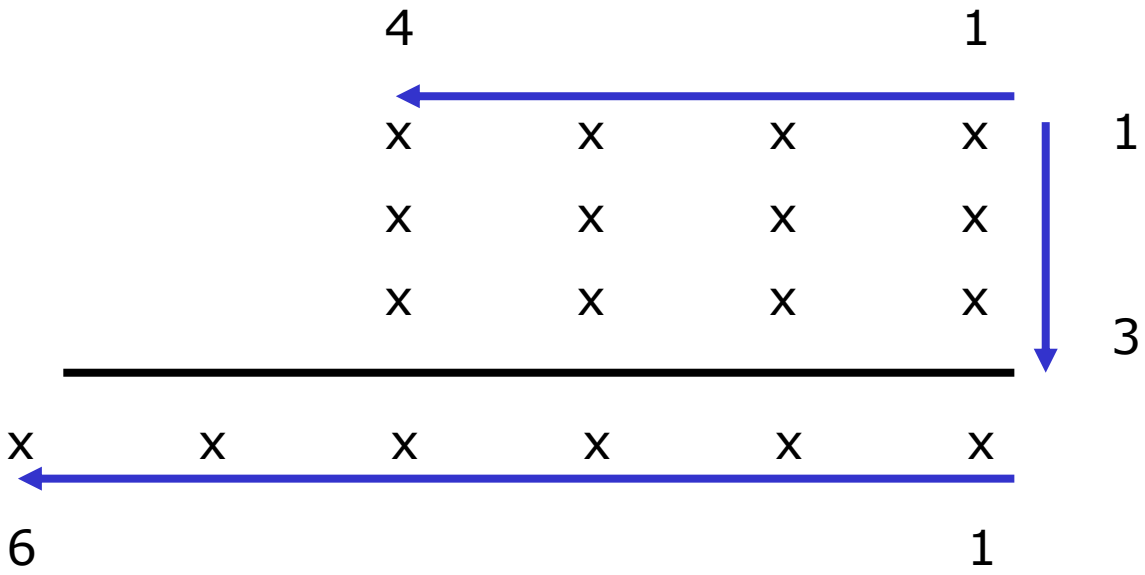


- Finite Impulse Response (FIR) filter
- Filter s 4 odcepi zahteva seštevanje 4 števil
  - Z n-odcepi pa seštevanje n-števil
- V nadaljevanju bomo pokazali vzporedno seštevanje več operandov z dvovhodnimi k-bitnimi seštevalniki
- Zapis z velikim "O" (črka, ne nič) ki predstavlja velikostni razred aproksimacije brez enot
  - Pomaga grobi aproksimaciji za primerjave brez manjših komponent (majhnih konstant …)
- Primer ripple carry: zakasnitev širjenja je reda **O(k)**
- Hitri seštevalniki (carry-select, carry-lookahead, prefix …): zakasnitev širjenja je reda **$O(\log_2(k))$**

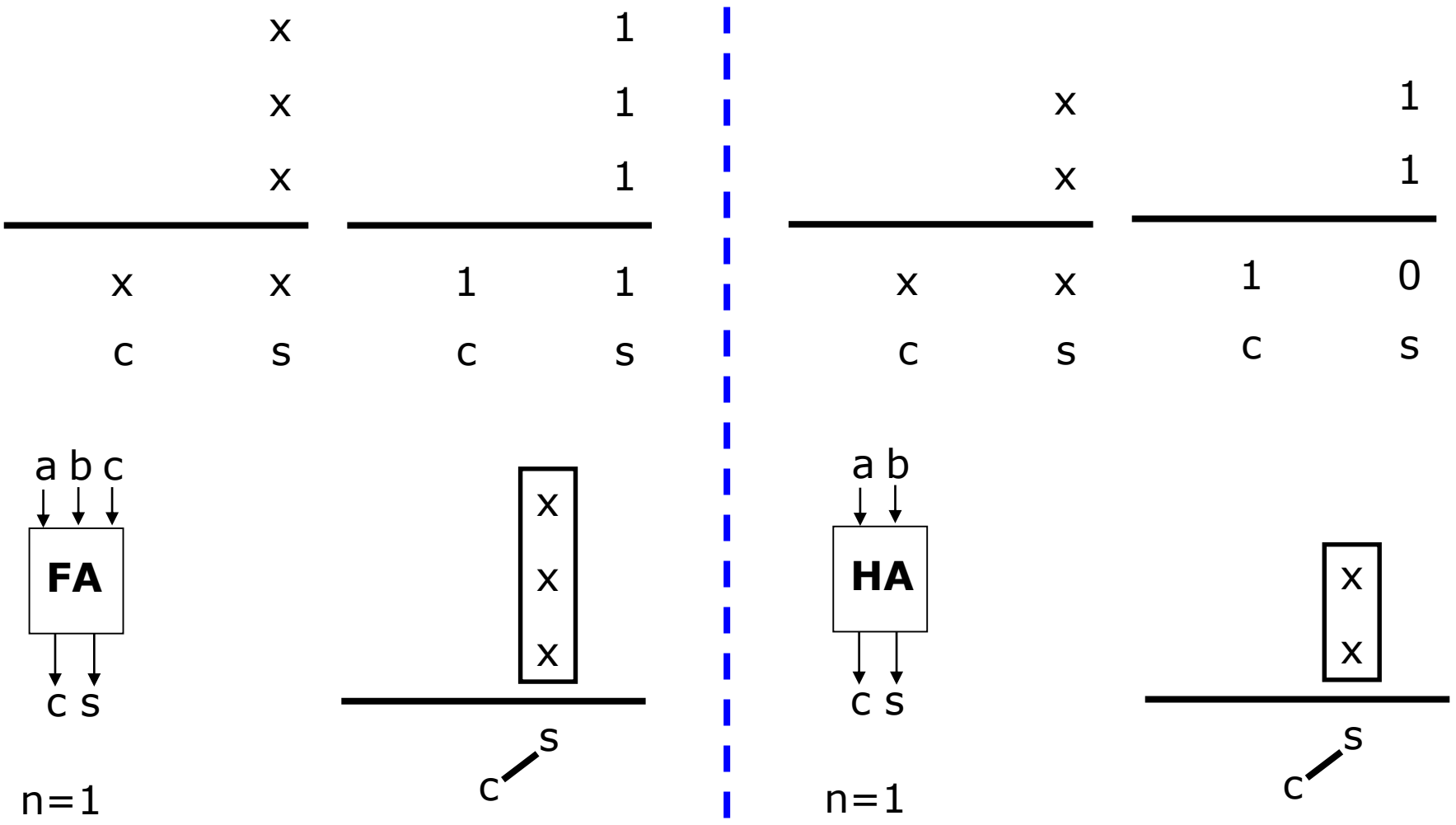
Paralelni števniki PŠ(m,k,n)



Primer PŠ(m,k,n): PŠ(3,6,4)

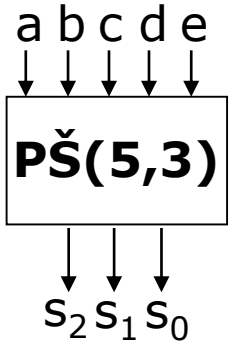


FA = PŠ(3,2), HA = PŠ(2,2)



PŠ(5,3)

		x			1
		x			1
		x			1
		x			1
		x			1
x	x	x	1	0	1

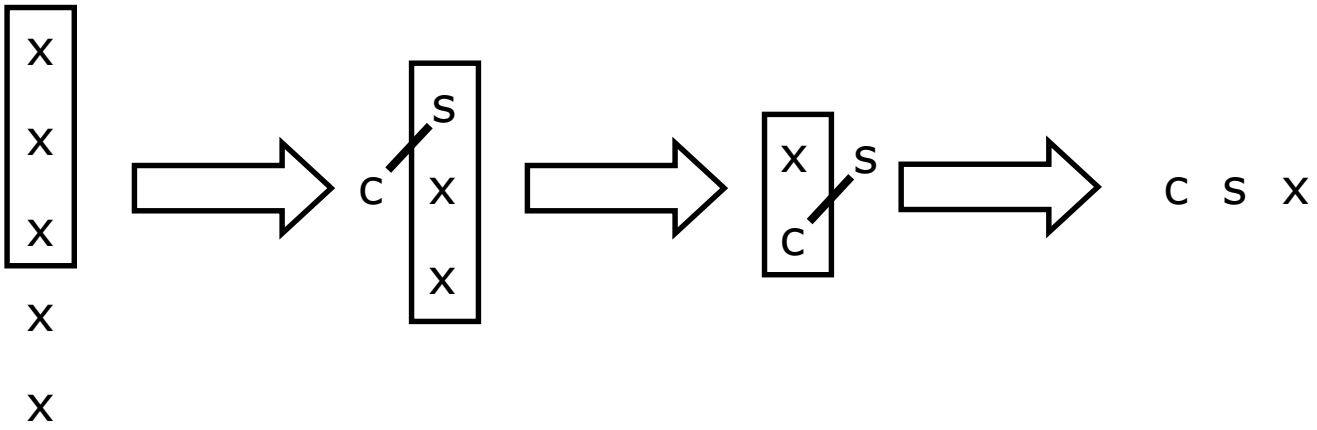


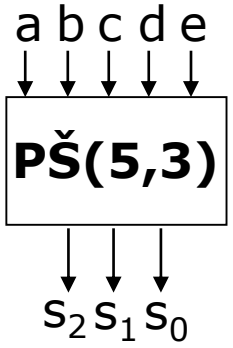
s_0=1 pri lihem številu enic na vhodu ➔ s_0=(a ⊕ b ⊕ c ⊕ d ⊕ e)

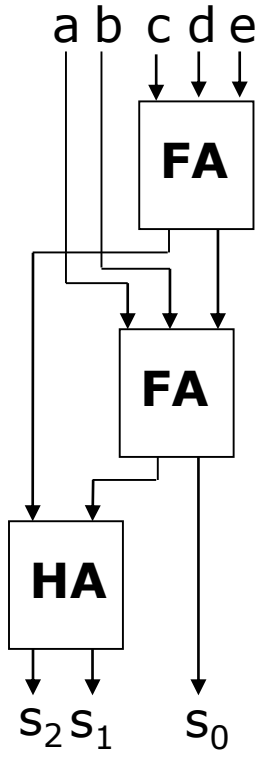
s_1=1 če sta na vhodu 2 ali 3 enice

s_2=1 če sta na vhodu 4 ali 5 enic

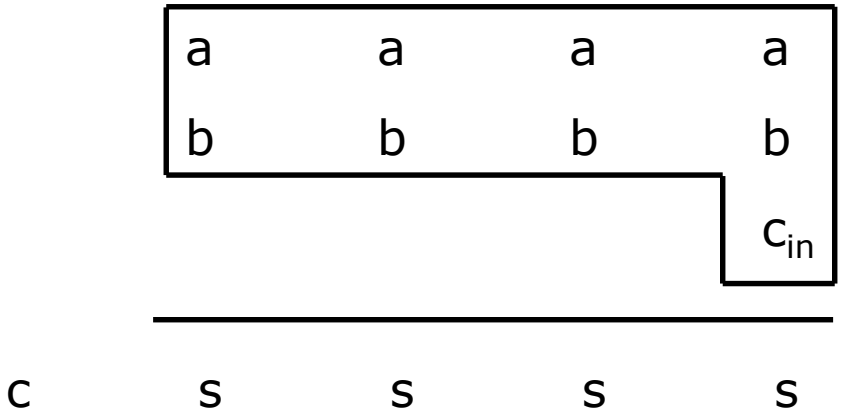
Tvorba PŠ(5,3) iz manjših PŠ



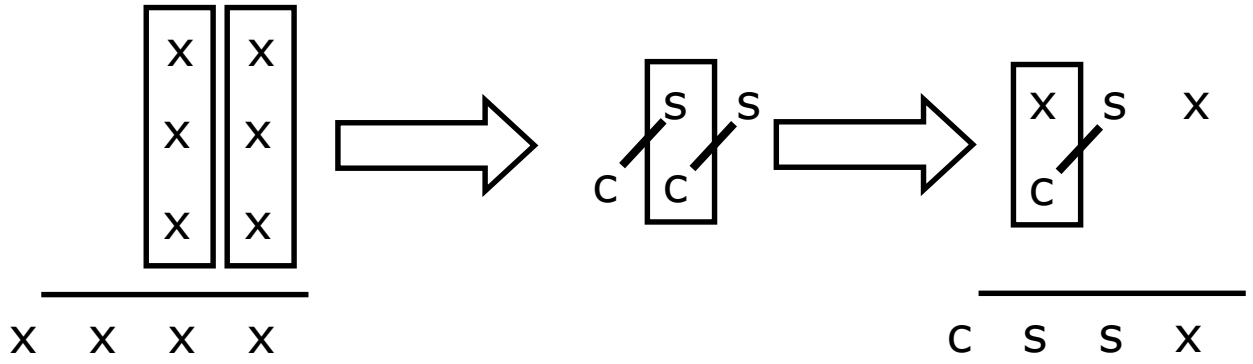




ALU kot 4-bitni PŠ



Primer: Seštevanje treh 4-bitnih števil z ALU (rabimo 2 ALU):



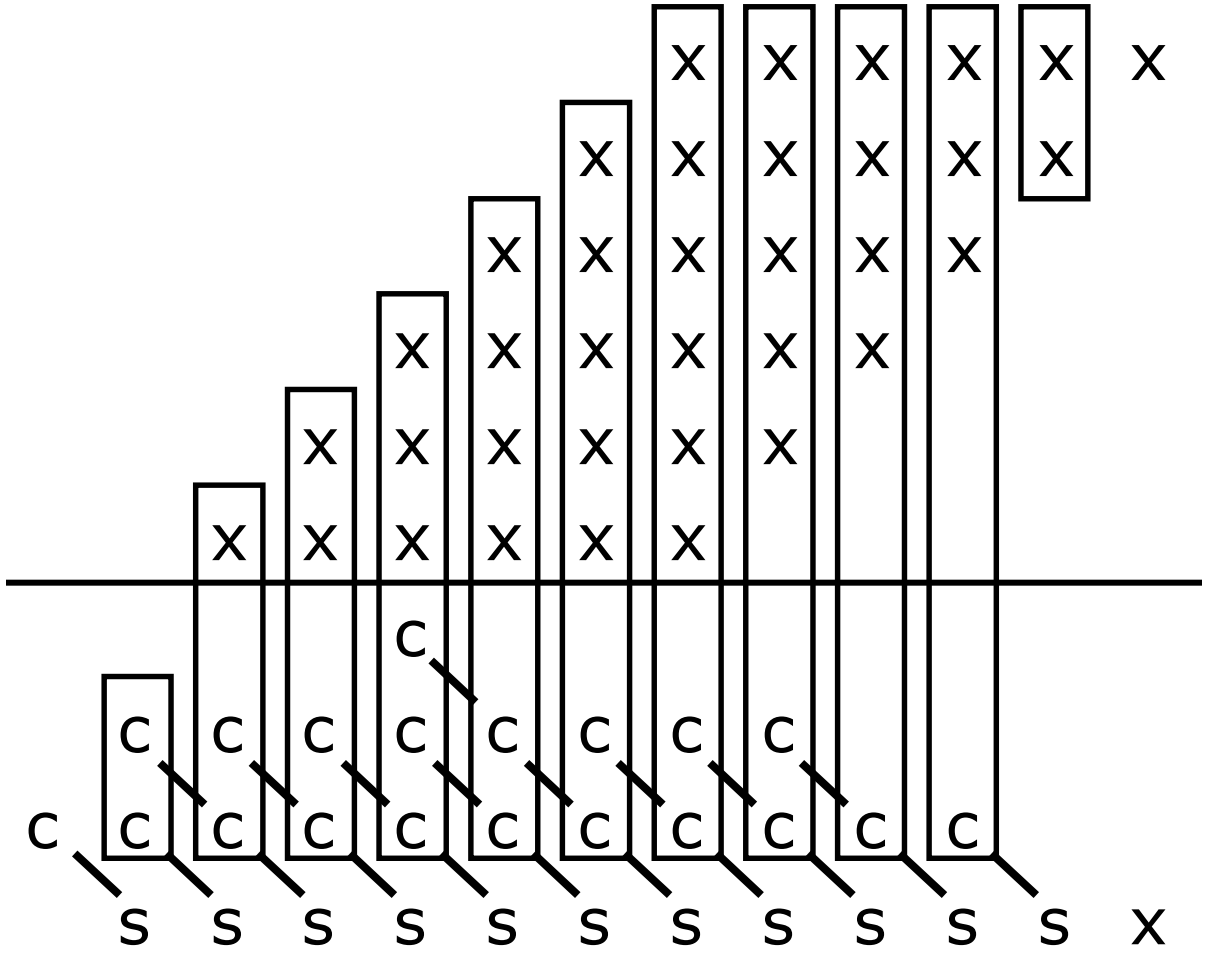
Načrtovanje digitalnih vezij

Sheme za hitro seštevanje

Direktna metoda

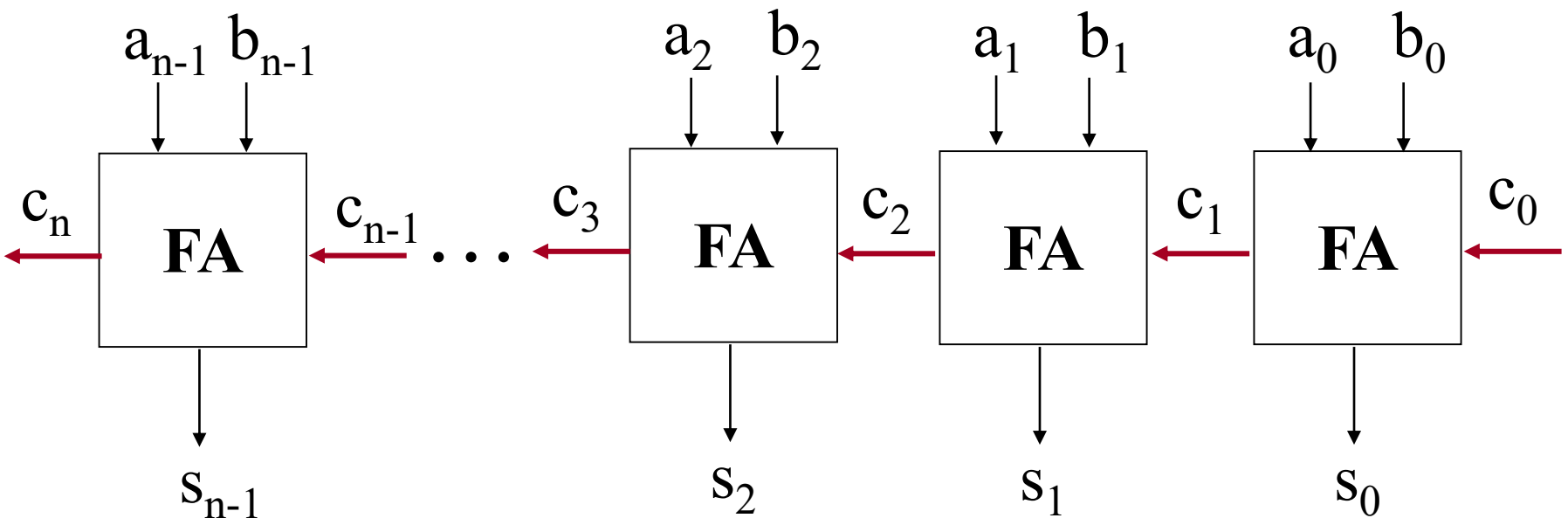
- V vsakem stolpcu vzamemo PŠ(m, $\log_2(m)+1$),
  - m dolžina stolpca + število prenosov iz prejšnjih stopenj.
- Problem:
  - Razširjanje prenosa
  - Potrebujemo PŠ z mnogo vhodi.

Direktna metoda



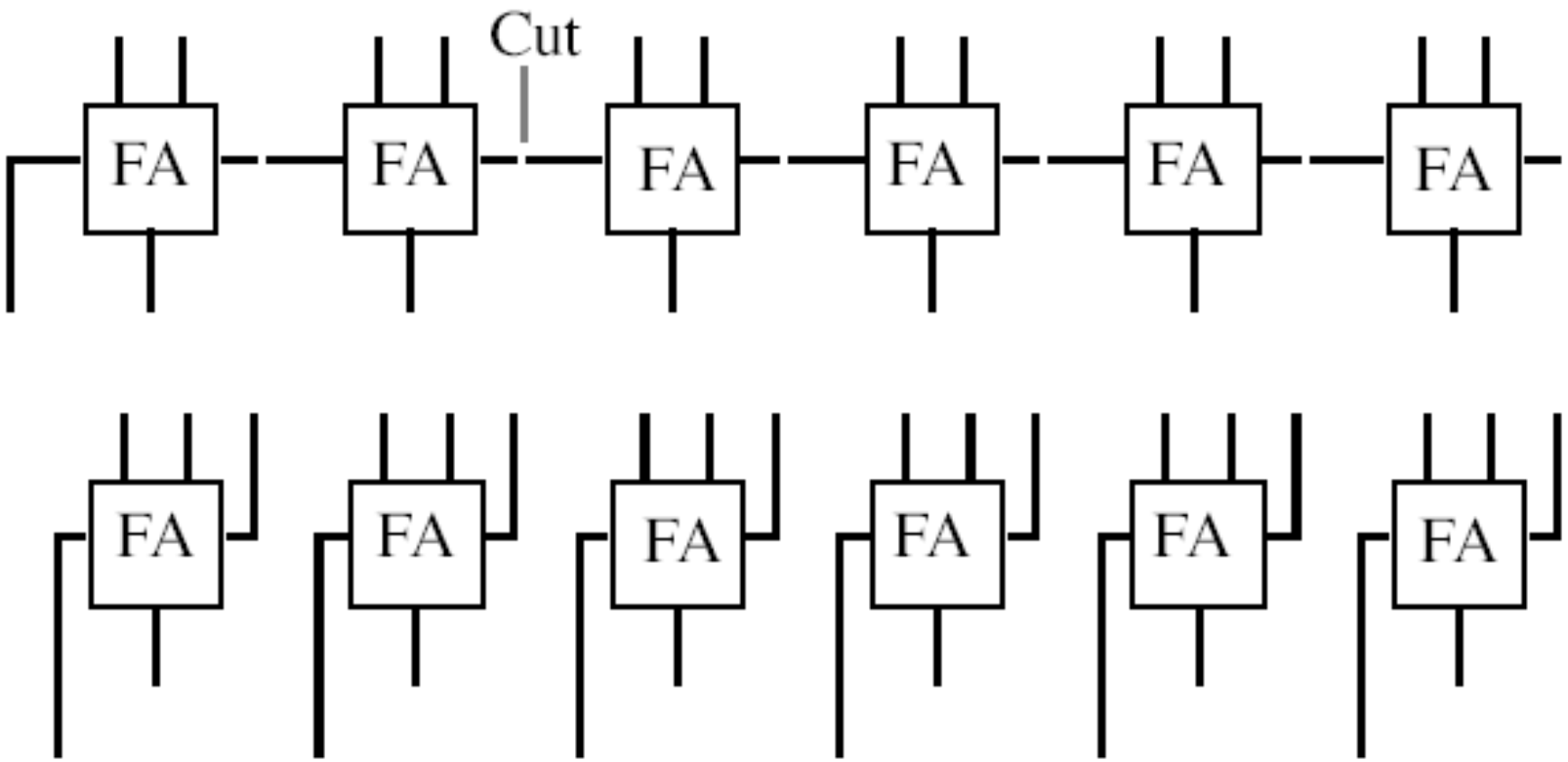
Carry Save Seštevalnik (CSA)

Do sedaj: Ripple-Carry oziroma Carry Propagate Adder (CPA)

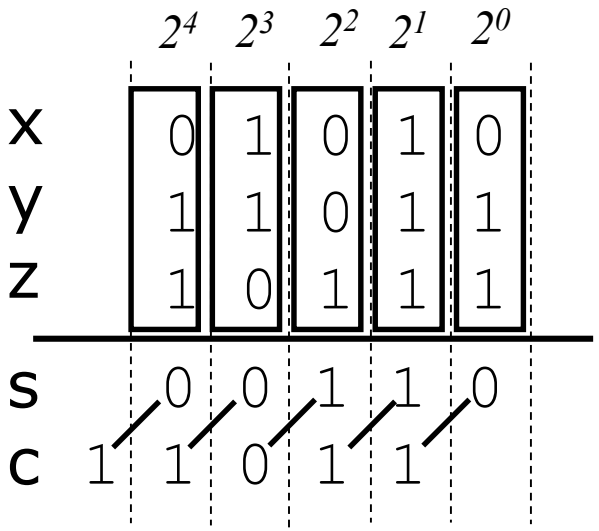


Kritično pot sestavlja n FA

RC seštevalnik : CSA seštevalnik

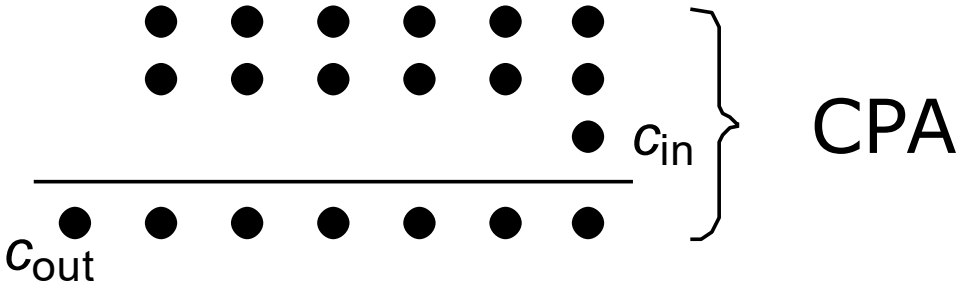


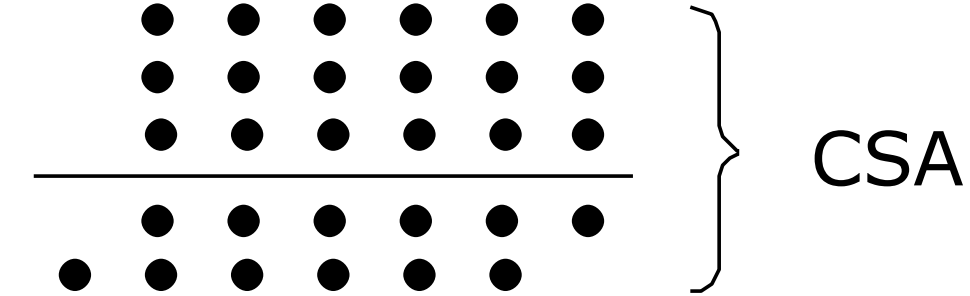
Delovanje CSA

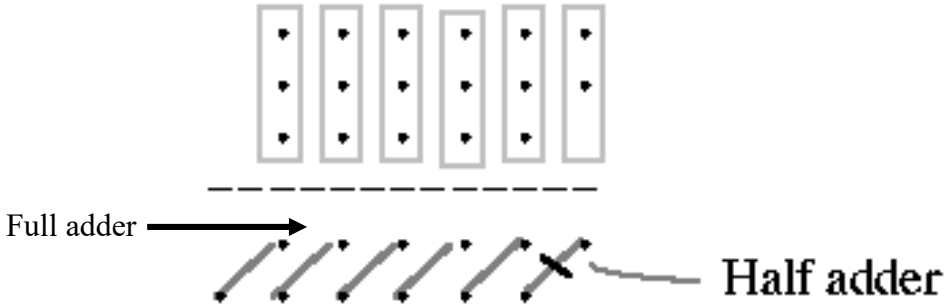


$$x+y+z = s + c$$

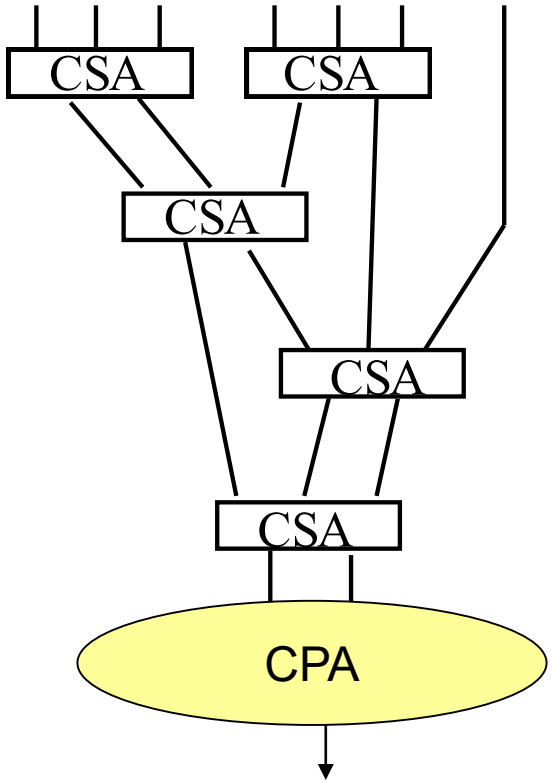
CPA in CSA v zapisu s pikami







Drevesa CSA

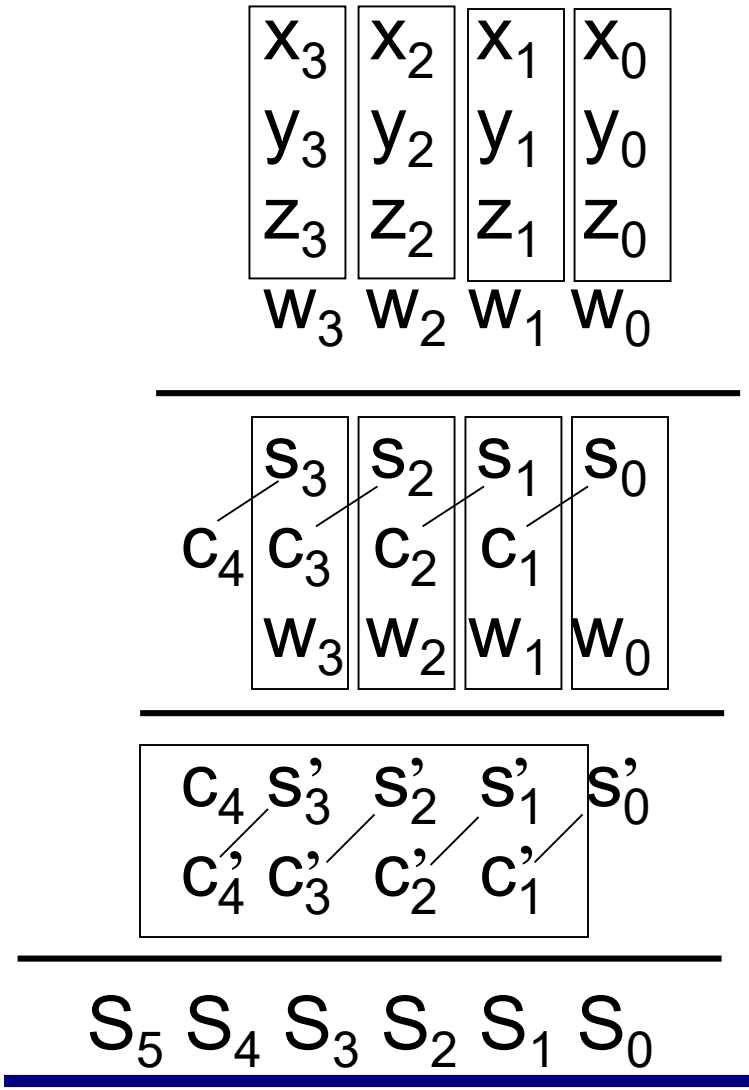


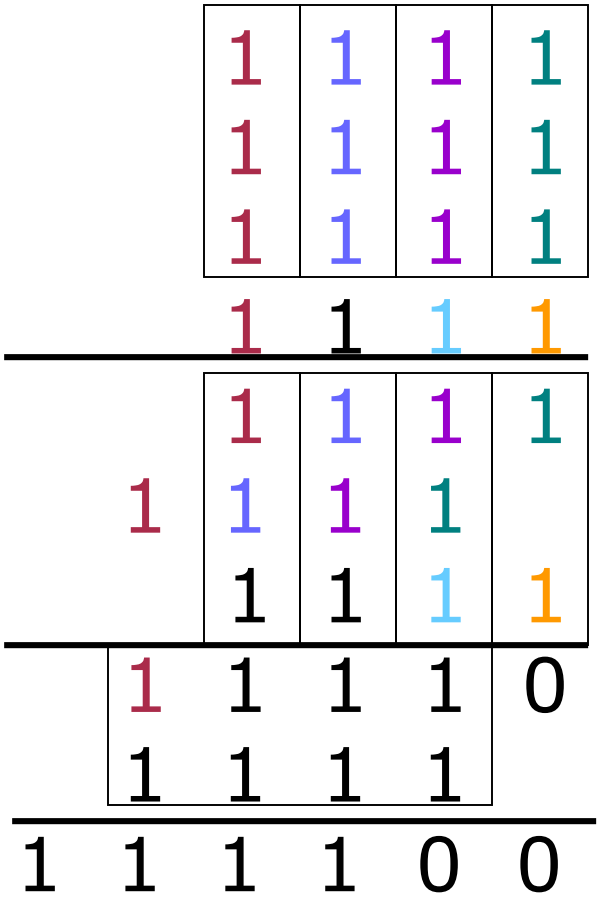
$$T_{\text{carry-save-multi-add}} = O(\text{višina drevesa} + T_{CPA})$$
$$= O(\log n + \log k)$$

$$C_{\text{carry-save-multi-add}} = (n-2)C_{CSA} + C_{CPA}$$

- CSA drevesa sestavljajo:
  - **grobo** $\log_2(n)$ stopenj CSA
    - Zato uporabljamo zapis z "O"
    - Vsaka stopnja predstavlja 1 enoto zakasnitve (eno FA celico), tako da je vseh enot zakasnitve $\log_2(n)$
    - Določili bomo točno število stopenj
  - En k-biten CPA
    - Če uporabljamo hitri seštevalnik, bo to zavzelo še $\log_2(k)$ enot zakasnitve

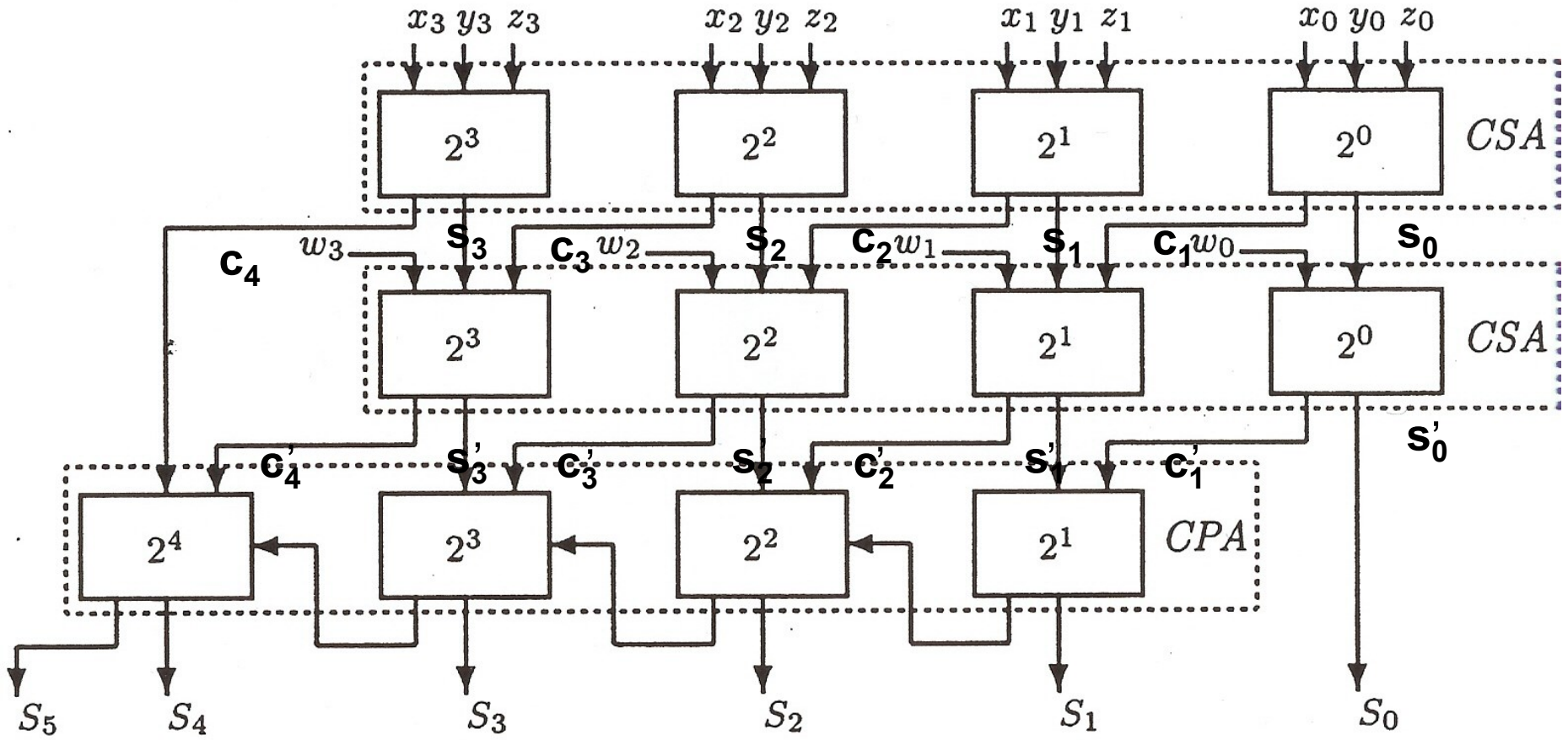
CSA za 4 operande



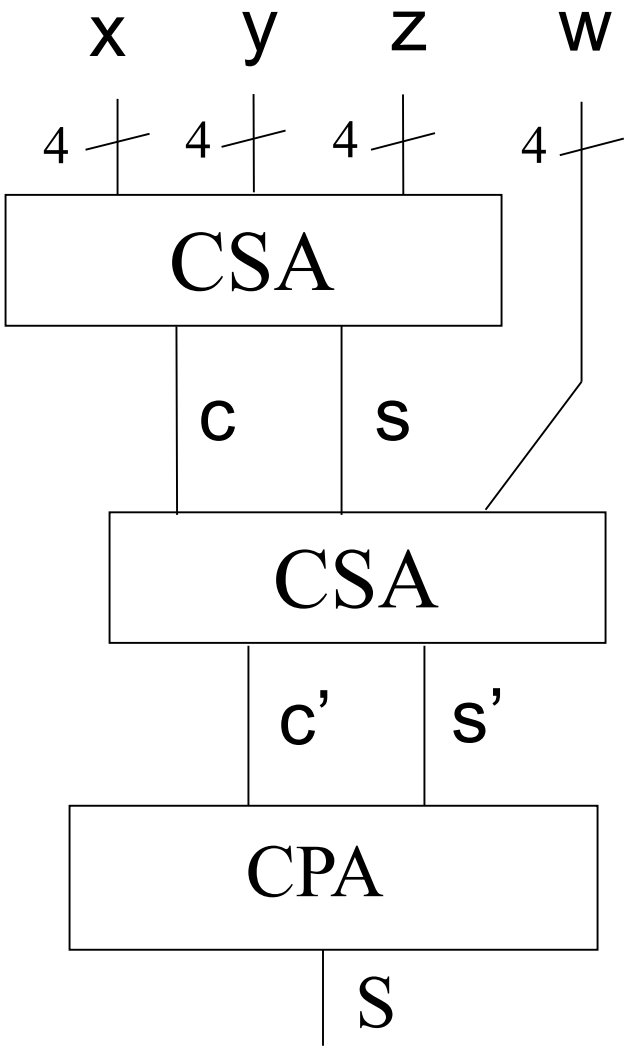


$4_{10} * 15_{10} = 60_{10}$

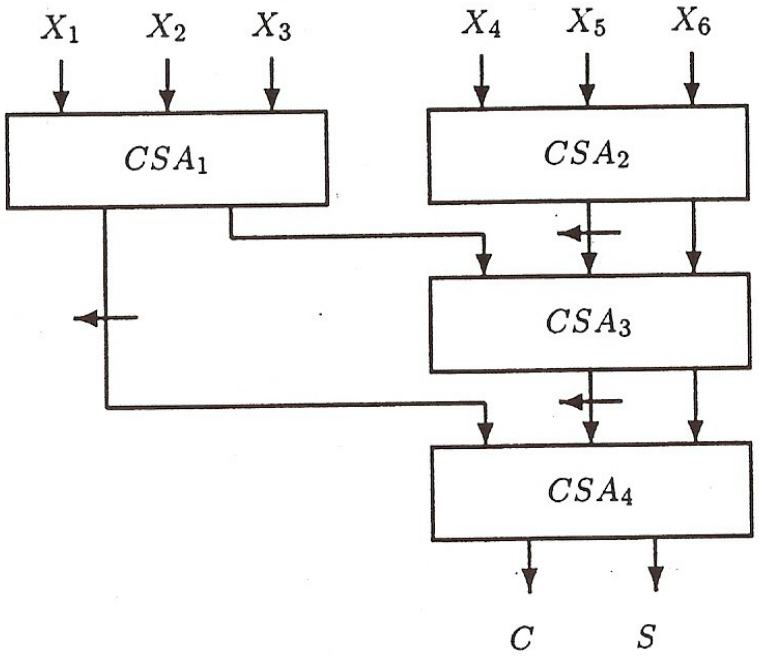
CSA za 4 operande



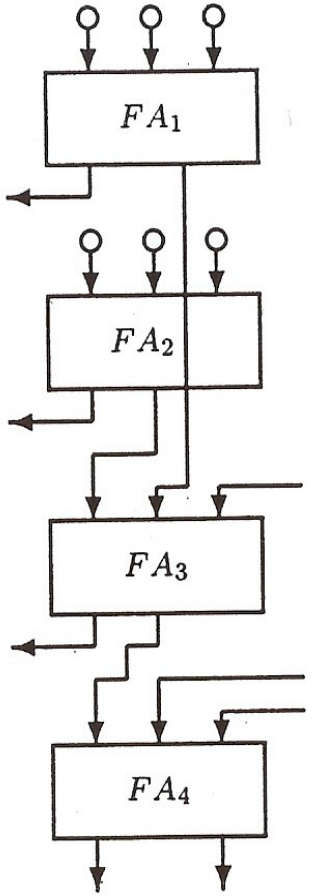
CSA za 4 operande



CSA za 6 operandov

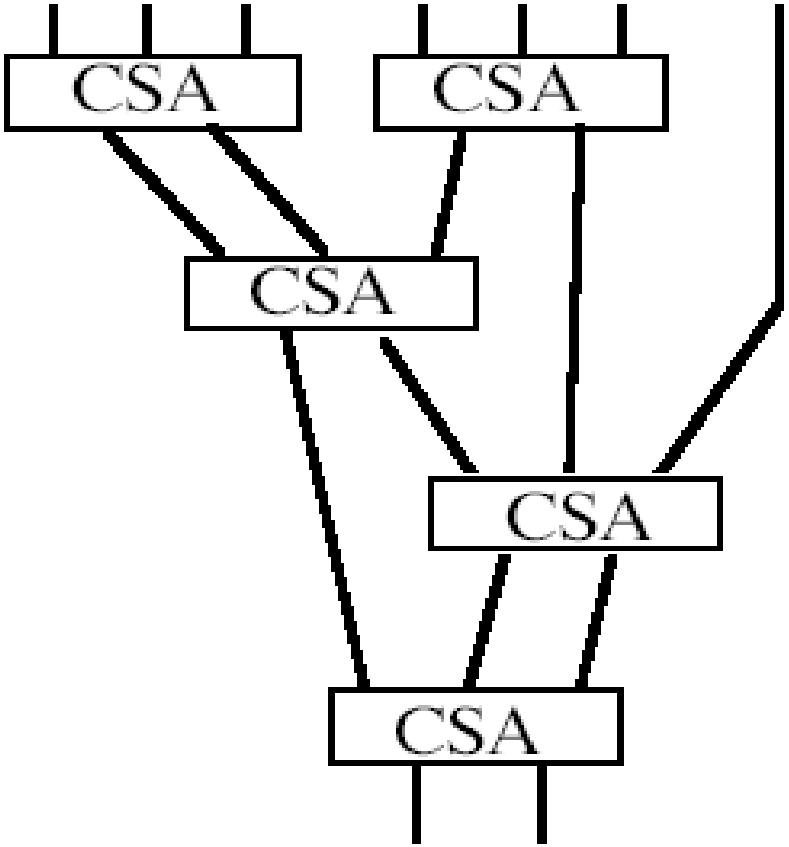


CSA drevo

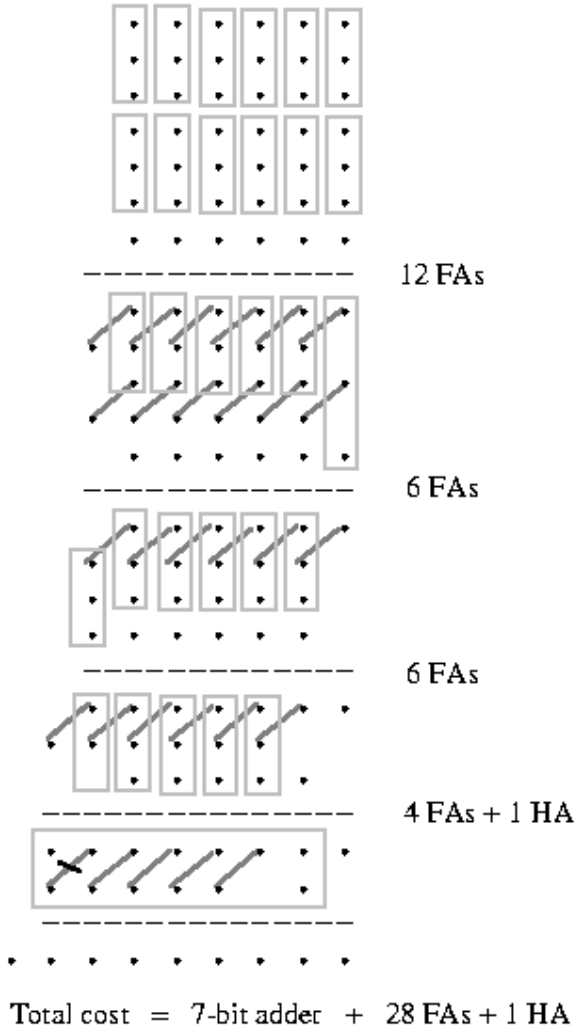


Realizacija enega bita

Drevo CSA, ki reducira 7 števil na 2

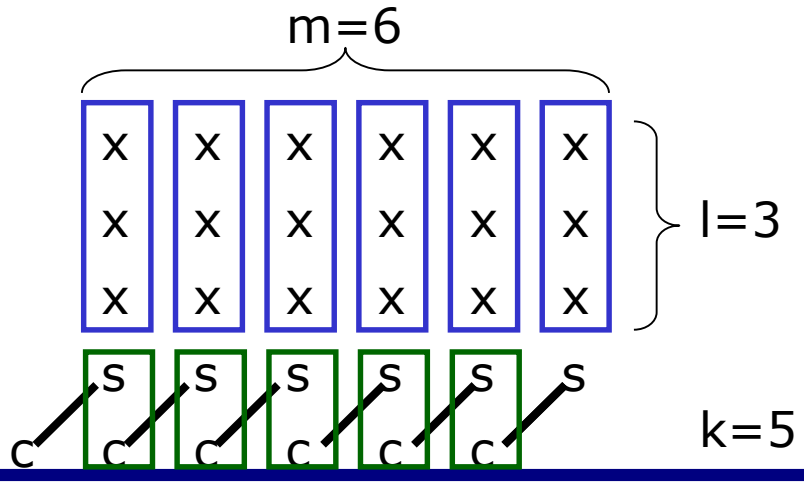


Seštevanje 7 6-bitnih števil v zapisu s pikami

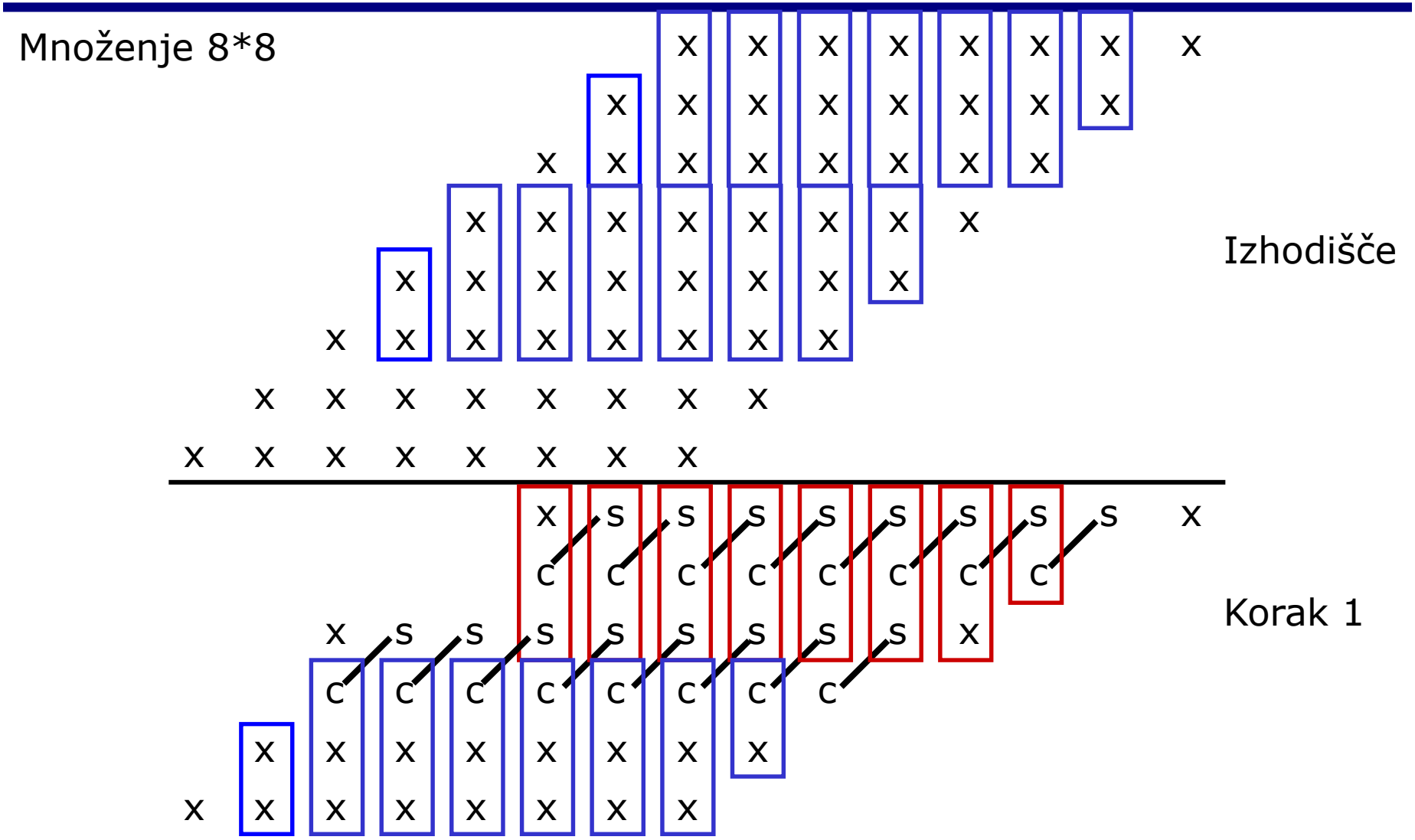


Wallace

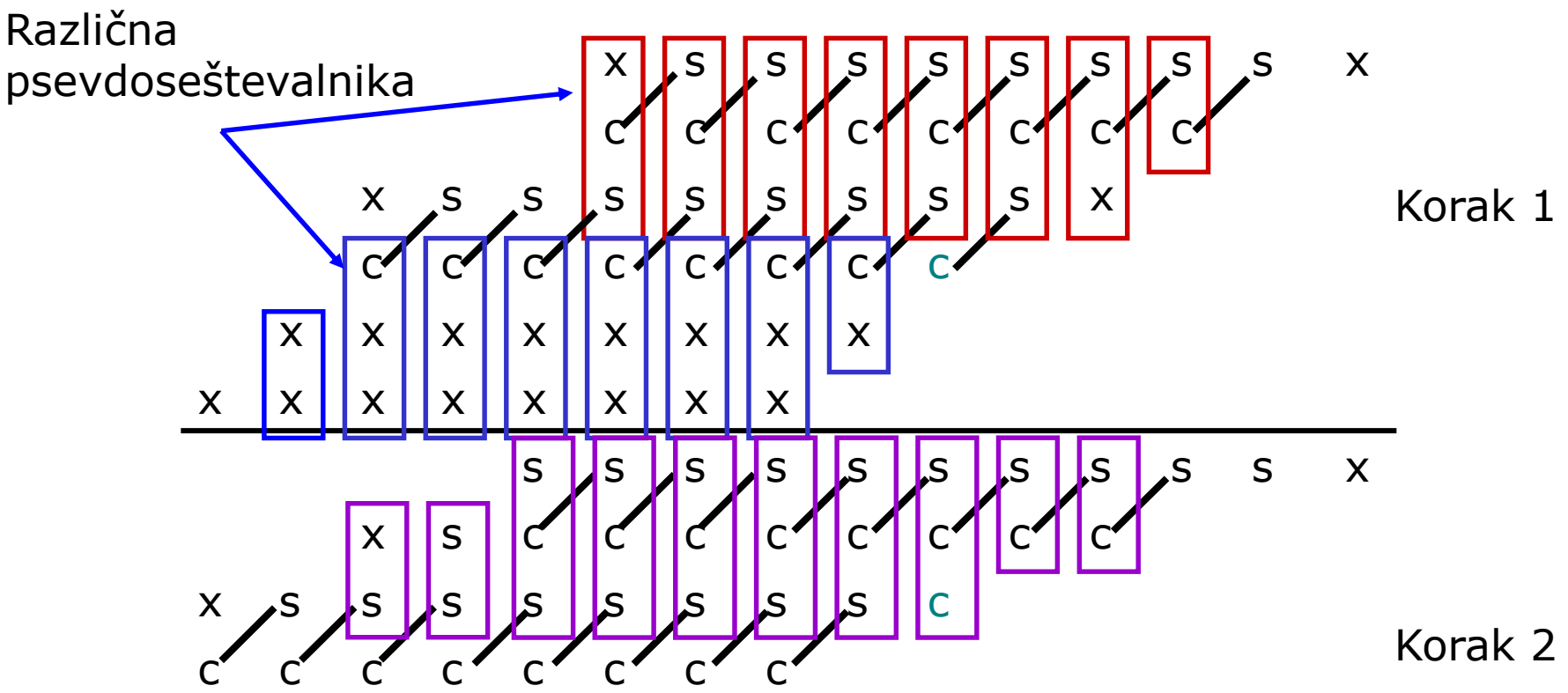
- Uporablja psevdo seštevalnike (Wallace-ovo drevo):
  - CSA, sestavljen iz več enakih PŠ
- Če seštevamo l vrstic, uporabimo v danem koraku $\lfloor l/m \rfloor$ psevdoseštevalnikov
  - m število vektorjev vhoda psevdoseštevalnika,
  - k število vektorjev izhoda psevdoseštevalnika.
- Dobljene in preostale vrstice peljemo v naslednji nivo
- Končamo, ko ostane le k vrstic.
  - V vsakem koraku pride do redukcije vrstic v razmerju m/k.



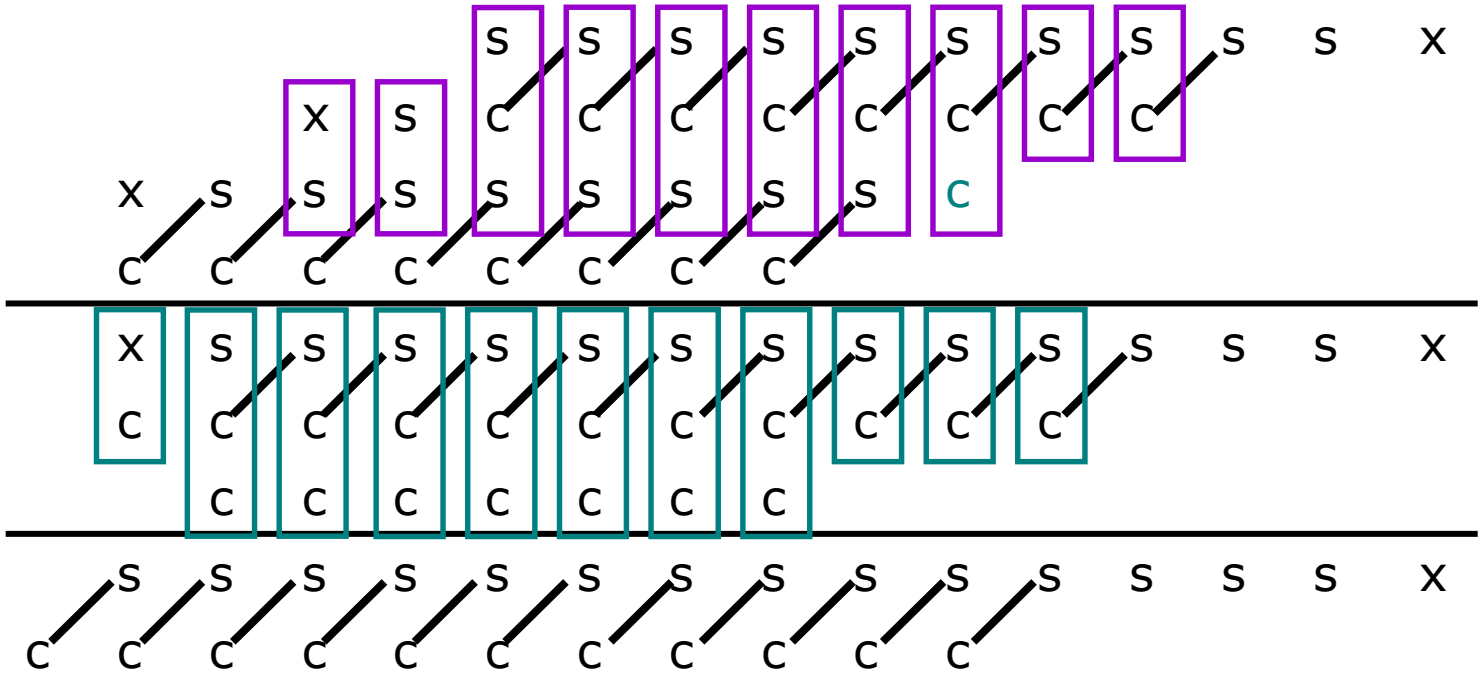
Wallace – primer



Wallace – primer



Wallace – primer



Korak 3

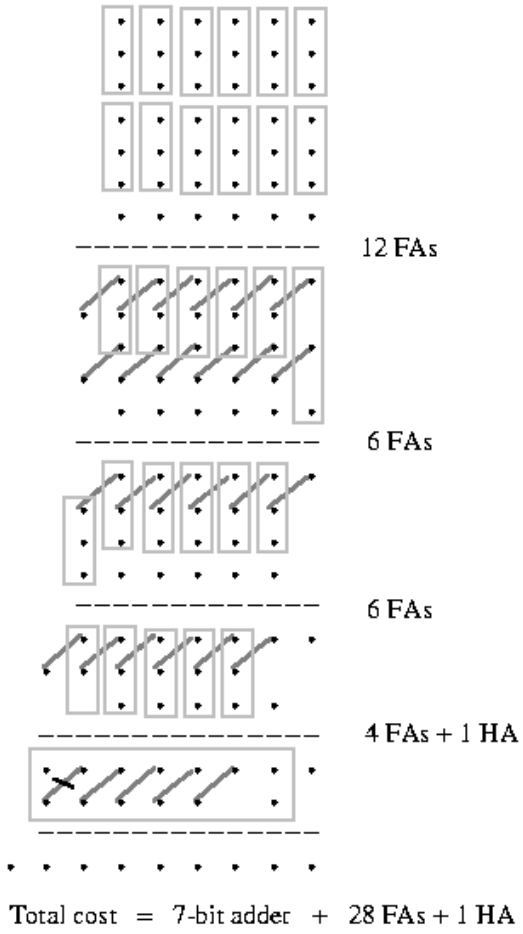
Wallace in Dadda drevesa

- Vsi primeri CSA dreves do sedaj so bila Wallace drevesa
- Wallace drevesa
  - Zmanjša velikost končnega CPA
  - Predstavljajo splošni optimum s stališča **hitrosti**
- Dadda drevesa
  - Zmanjša strošek CSA drevesa
  - Predstavljajo splošni optimum s stališča **površine** (med CSA drevesi)

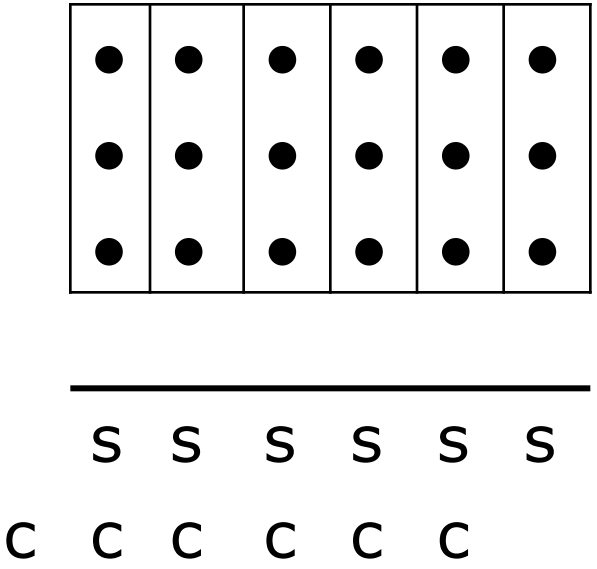
Wallace in Dadda drevesa

- Wallace zmanjša število operandov čim ima možnost
  - Cilj je najmanjše število bitov v CPA seštevalniku
  - Včasih nekaj bitov daljši CPA ne vpliva preveč na zakasnitev širjenja (recimo CLA izvedba CPA)
- Dadda išče minimalno število FA in HA enot
  - Za ceno rahlo večjega končnega CPA

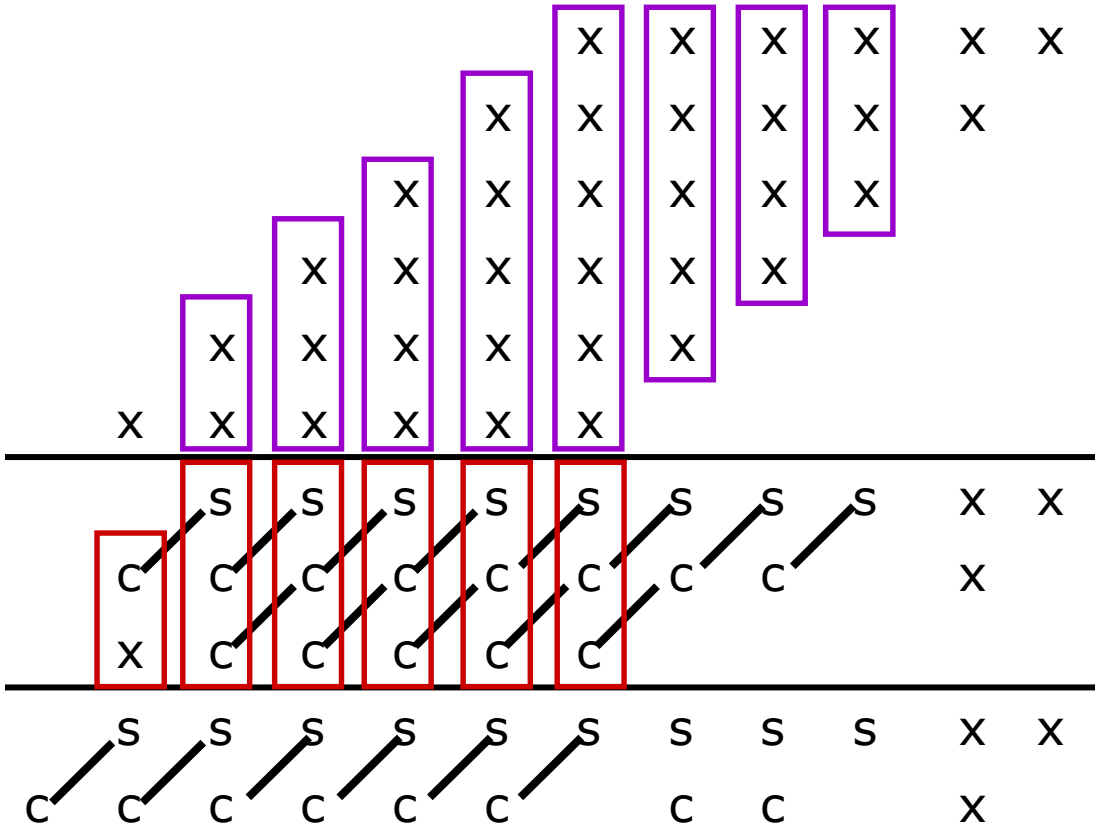
Wallace drevo



Wallace zmanjša število operandov z uporabo psevdoseštevalnikov!



Dadda – osnovna metoda



- generiranje dveh števil
- seštevanje teh dveh števil

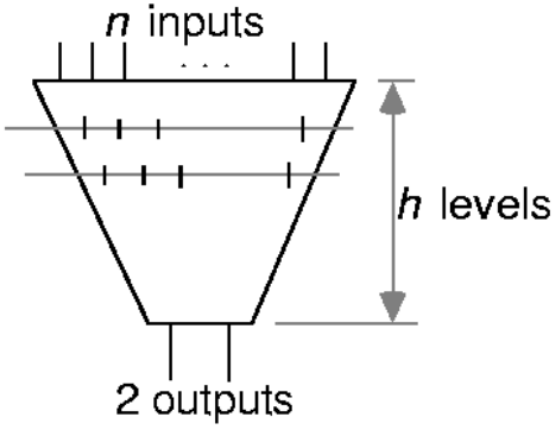
z večbitnim paralelnim seštevalnikom.

V stolpcu s 3 ali več biti vzamemo PŠ take dolžine, kot je stolpec.

V stolpcu z 2 bitoma je PŠ samo če se bo pojavil prenos iz prejšnje stopnje.

Število vhodov (višina drevesa)

n(h): Največje število vhodov,
ki jih lahko reduciramo na 2 z drevesom višine h



$$n(0) = 2$$

$$n(h) = \left\lfloor \frac{3}{2}\, n(h\text{-}1) \right\rfloor$$

n(1) = 3 n(4) = 9
n(2) = 4 n(5) = 13
n(3) = 6 n(6) = 19

Višina drevesa (število vhodov)

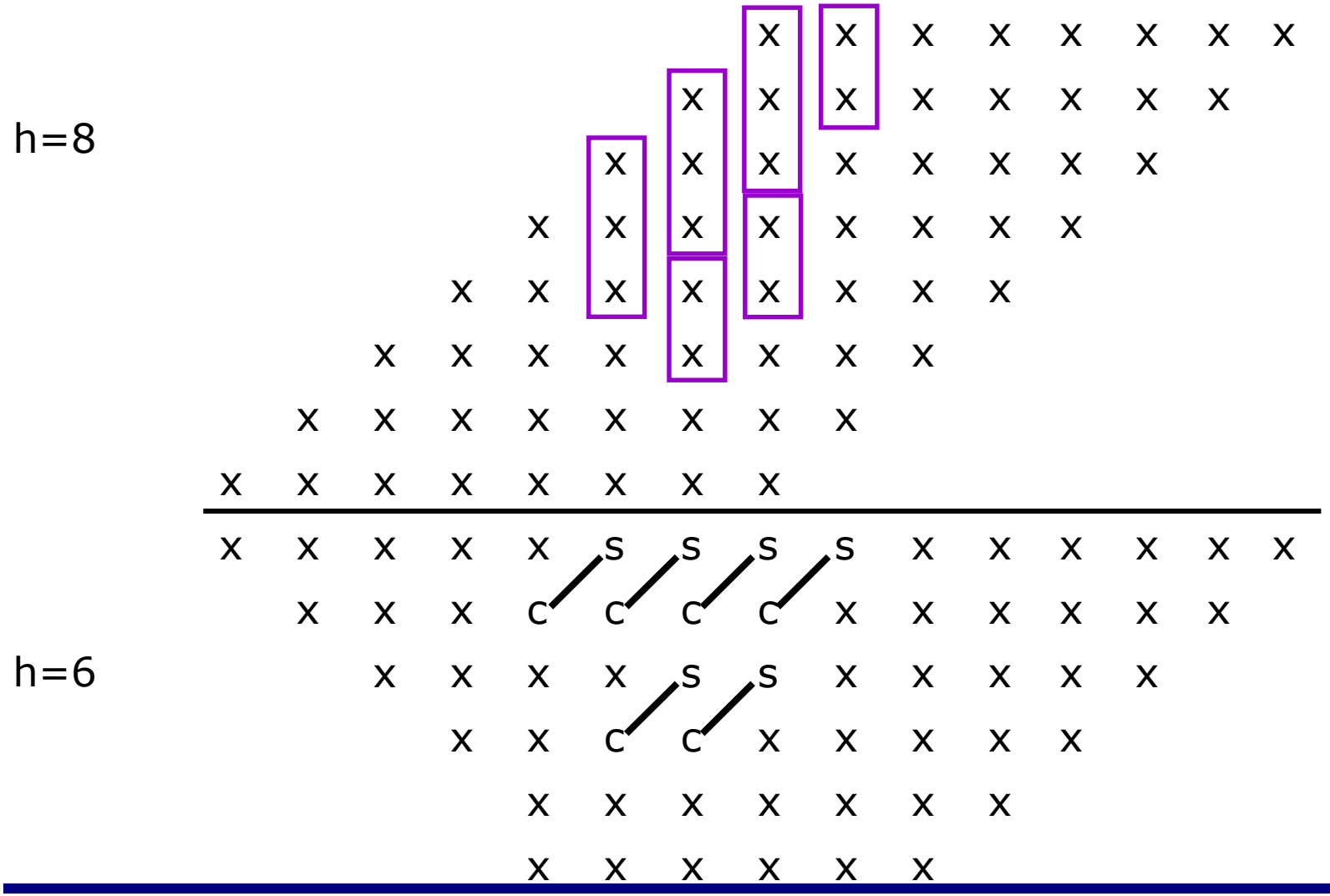
Najmanjša višina CSA drevesa za n operandov h(n)

$$h(n) = 1 + h\left(\left\lceil \frac{2}{3} n \right\rceil\right)$$

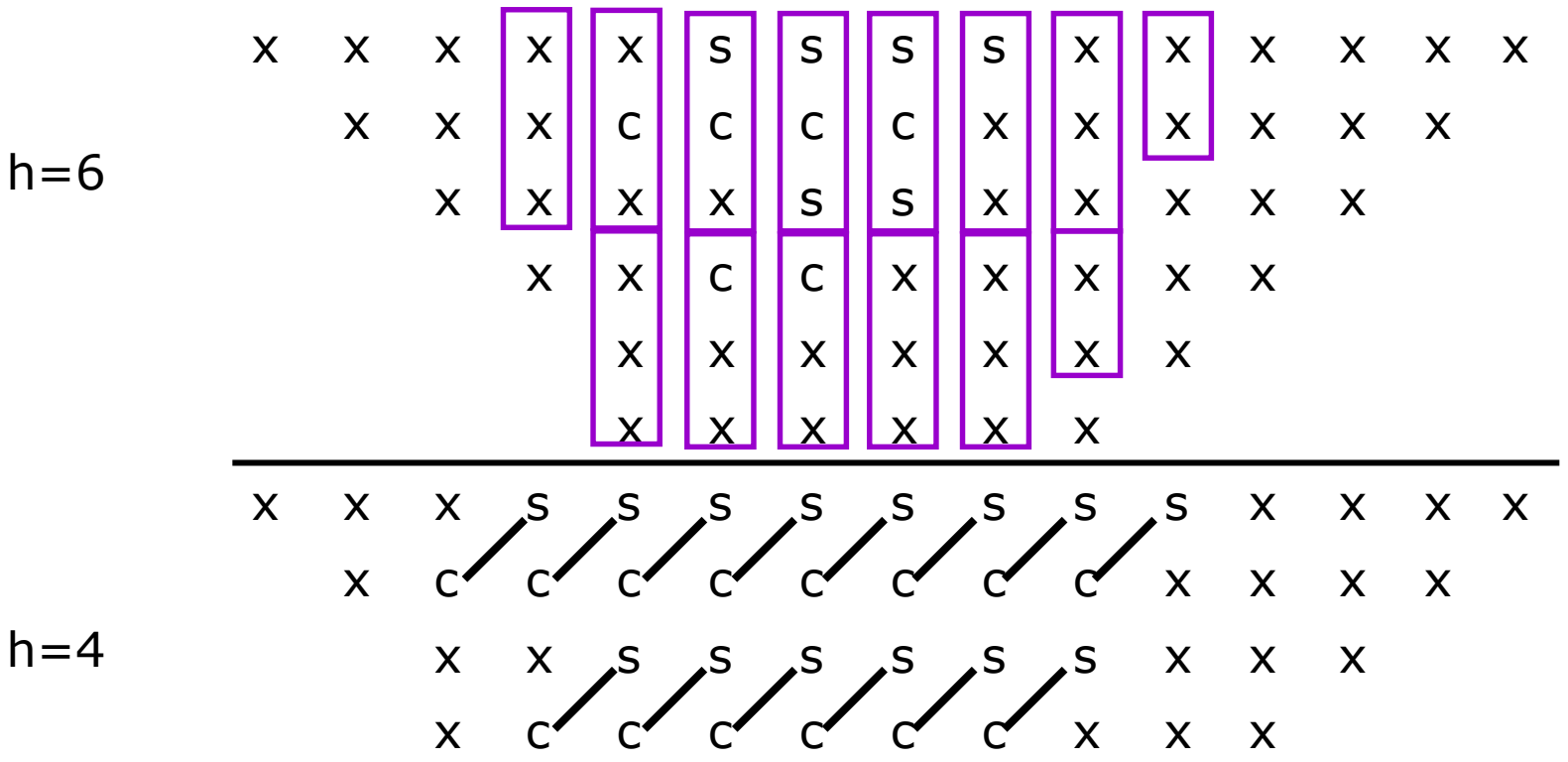
$$h(2) = 0$$

$$h(n) \geq \log_{\frac{3}{2}}\left(\frac{n}{2}\right)$$

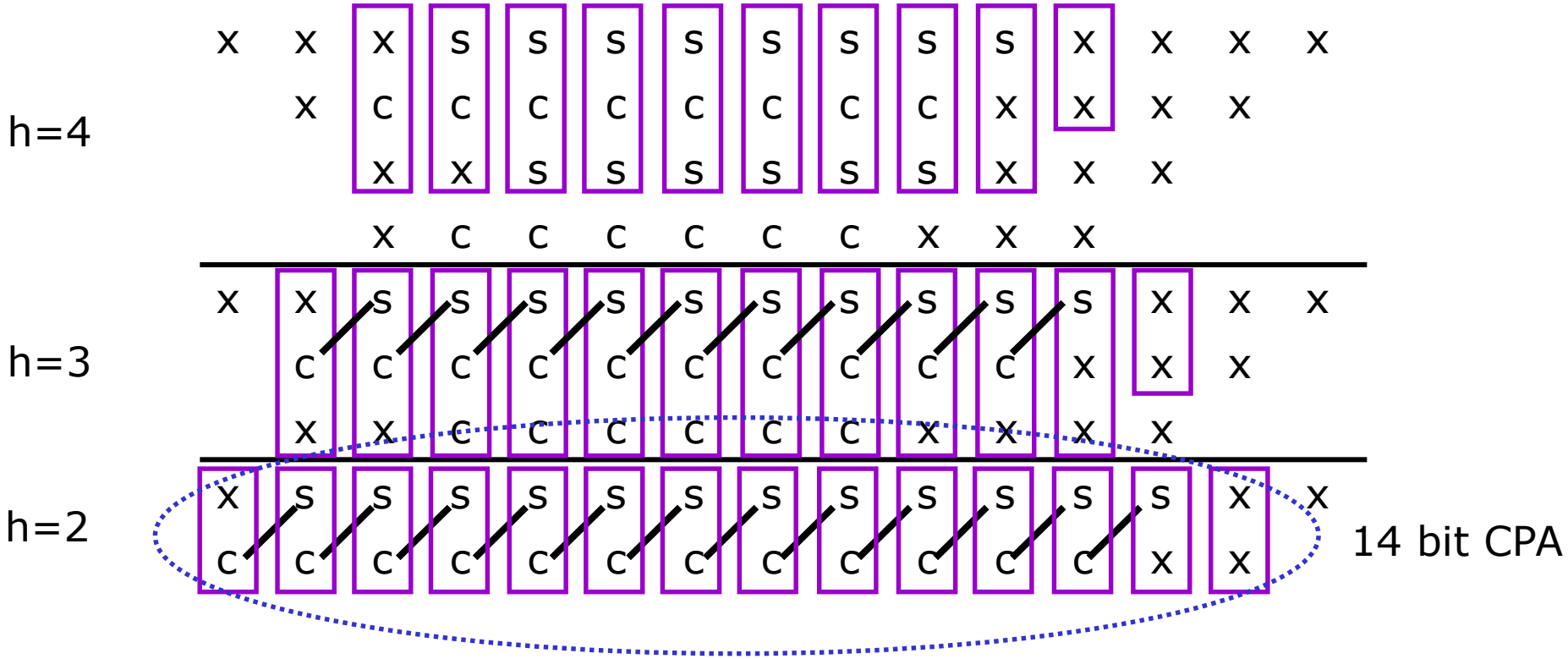
Dadda – modificirana metoda



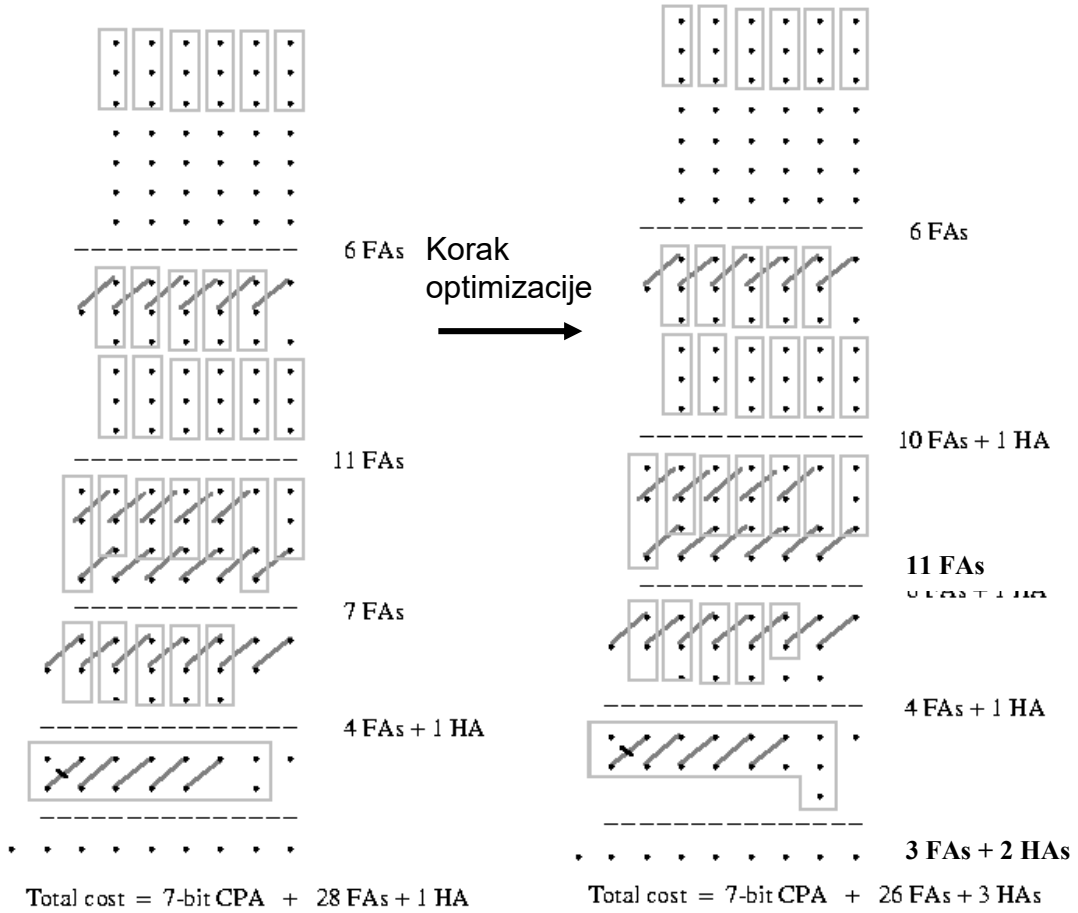
Dadda – modificirana metoda



Dadda – modificirana metoda

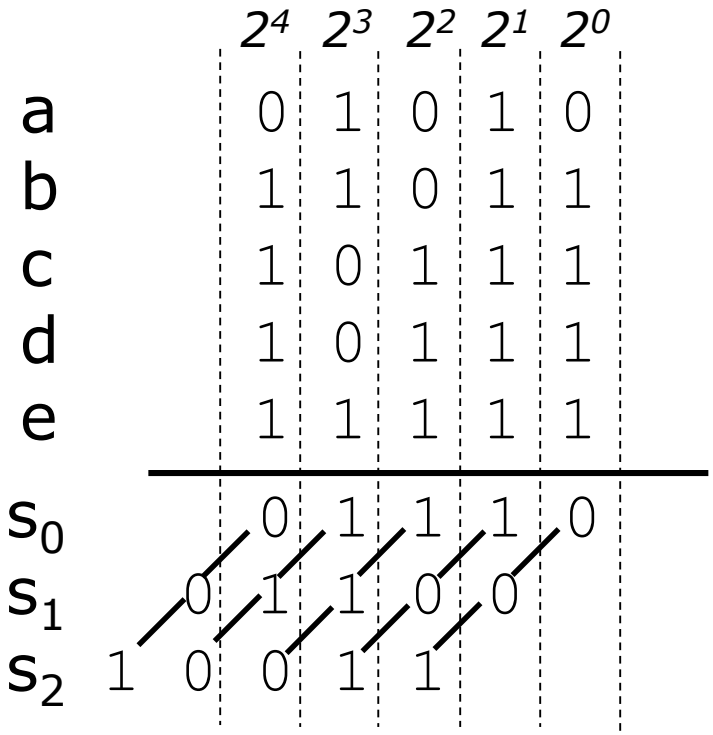


Dadda drevo



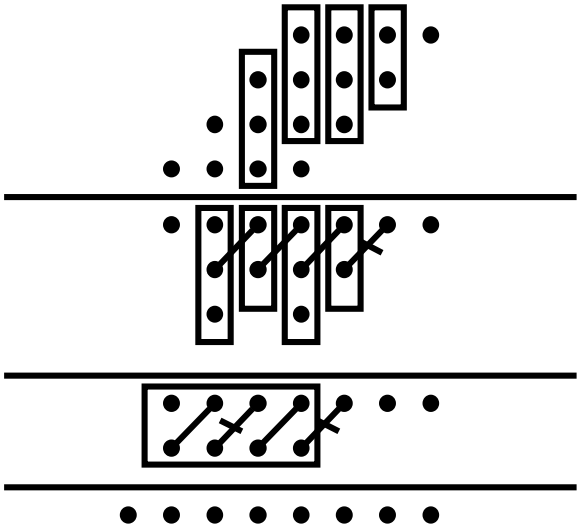
Dadda zmanjša število operandov čim kasneje: S tem ne povzroča dodatnih zakasnitev!

PŠ(5,3)



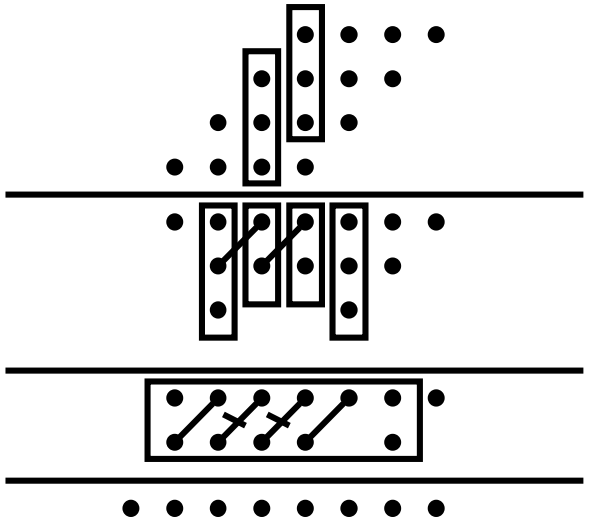
$$a+b+c+d+e = s_0+s_1+s_2$$

Wallace : Dadda drevo (4x4)



Wallace drevo: 2 CSA nivoja,
5 FA, 3 HA, 4-bitni CPA

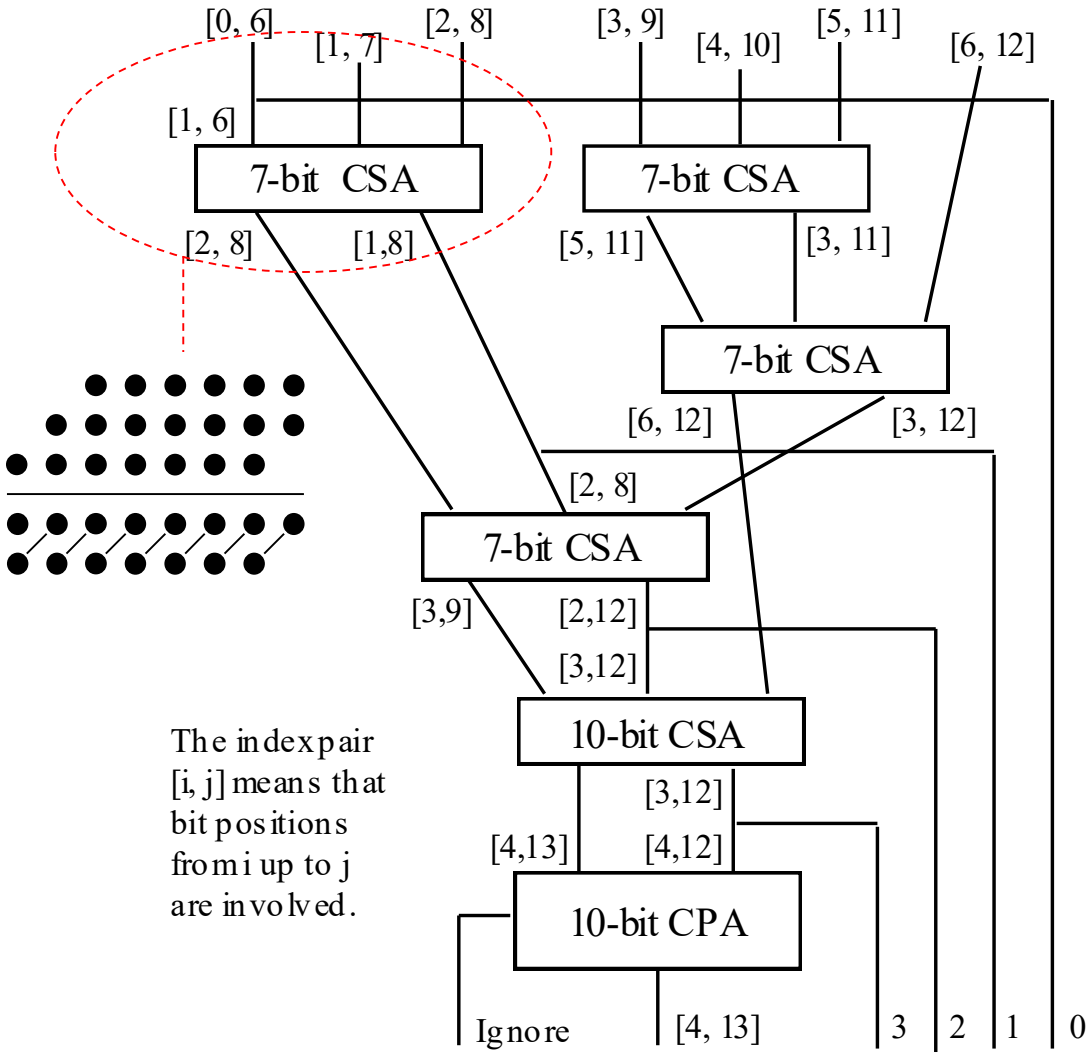
1	2	3	4	3	2	1
		FA	FA	FA	HA	
1	3	2	3	2	1	1
	FA	HA	FA	HA		
2	2	2	2	1	1	1
4-bitni seštevalnik						
1	1	1	1	1	1	1



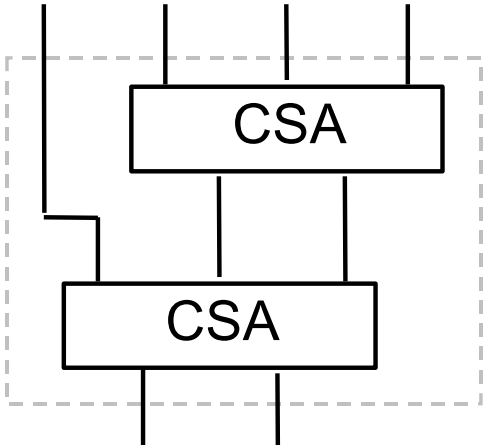
Dadda drevo: 2 CSA nivoja,
4 FA, 2 HA, 6-bitni CPA

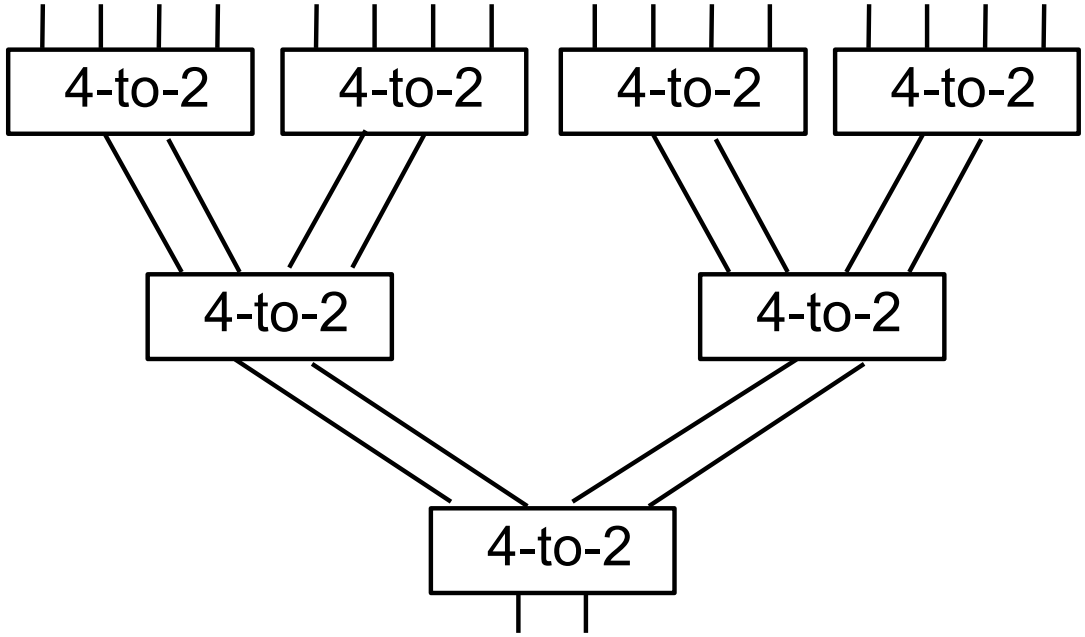
1	2	3	4	3	2	1
		FA	FA			
1	3	2	3	2	1	1
	FA	HA	HA	FA		
2	2	2	2	1	2	1
6-bitni seštevalnik						
1	1	1	1	1	1	1

7 x 7 množilnik



Drugi načini redukcije: Dvojiško drevo iz PŠ(4,2)





Načrtovanje digitalnih vezij

Množenje z drevesnimi strukturami

Množilnik

			*	a_2	a_1	a_0
				b_2	b_1	b_0
				a_2b_0	a_1b_0	a_0b_0
			+	a_2b_1	a_1b_1	a_0b_1
			+	a_2b_2	a_1b_2	a_0b_2
p_5	p_4	p_3	p_2	p_1	p_0	

$7_{10}*4_{10}=28_{10}$

			1	1	1	
		*				
			1	0	0	
			0	0	0	0
+						
		0	0	0		0
+						
	1	1	1			1
28_{10} =	1	1	1	0	0	

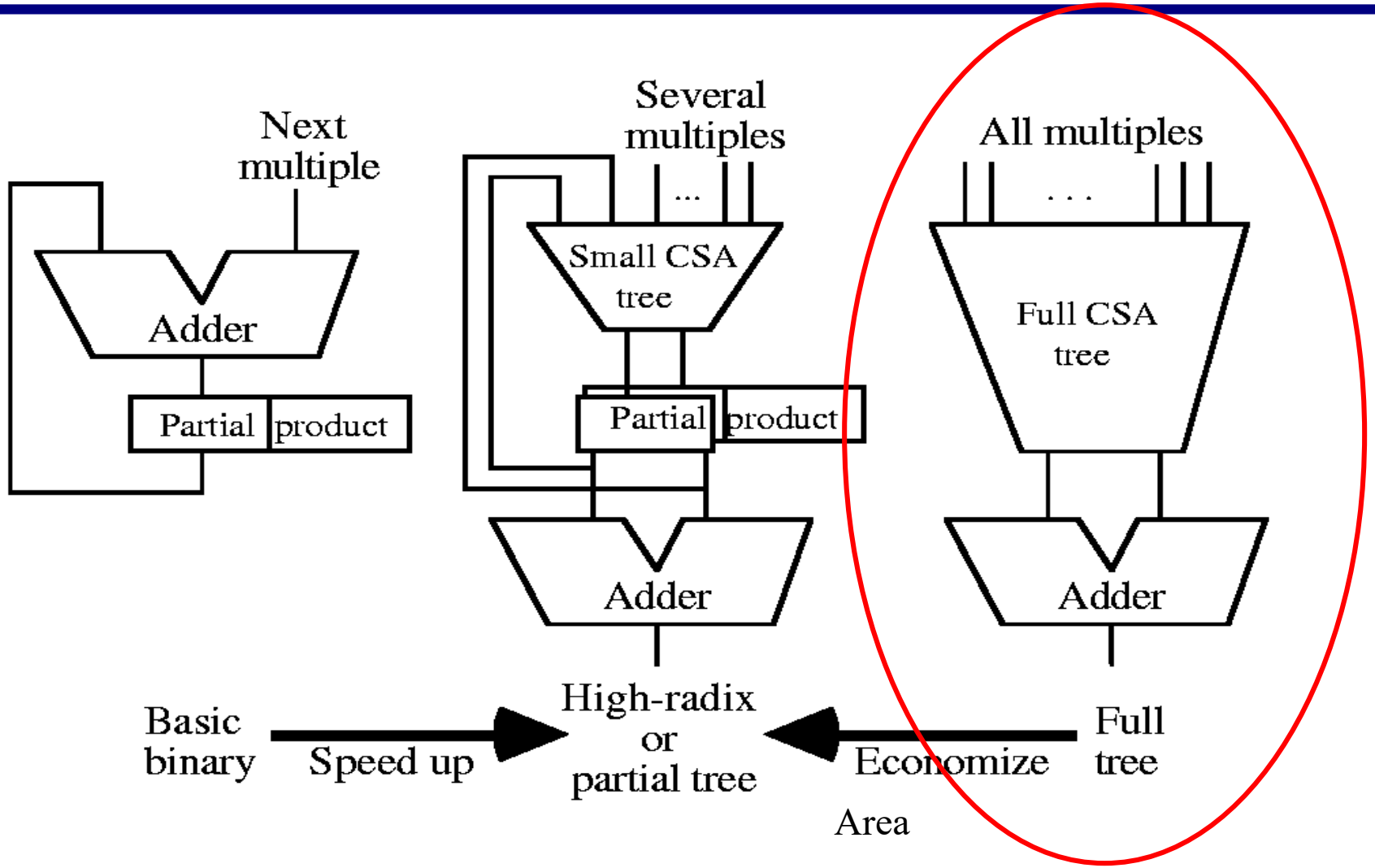
$(-6_{10})*5_{10}=-30_{10}$

					1	0	1	0	*	0	1	0	1
1	1	1	1	1	0	1	0						
0	0	0	0	0	0	0							
1	1	1	0	1	0								
0_1	0_1	0_1	0_1	0_1	0								
1	1	1	0	0	0	1	0						
0	0	0	1	1	1	1	0						

$E2_{16}=-30_{10}$

$1E_{16}=30_{10}$

Arihtekture množenja



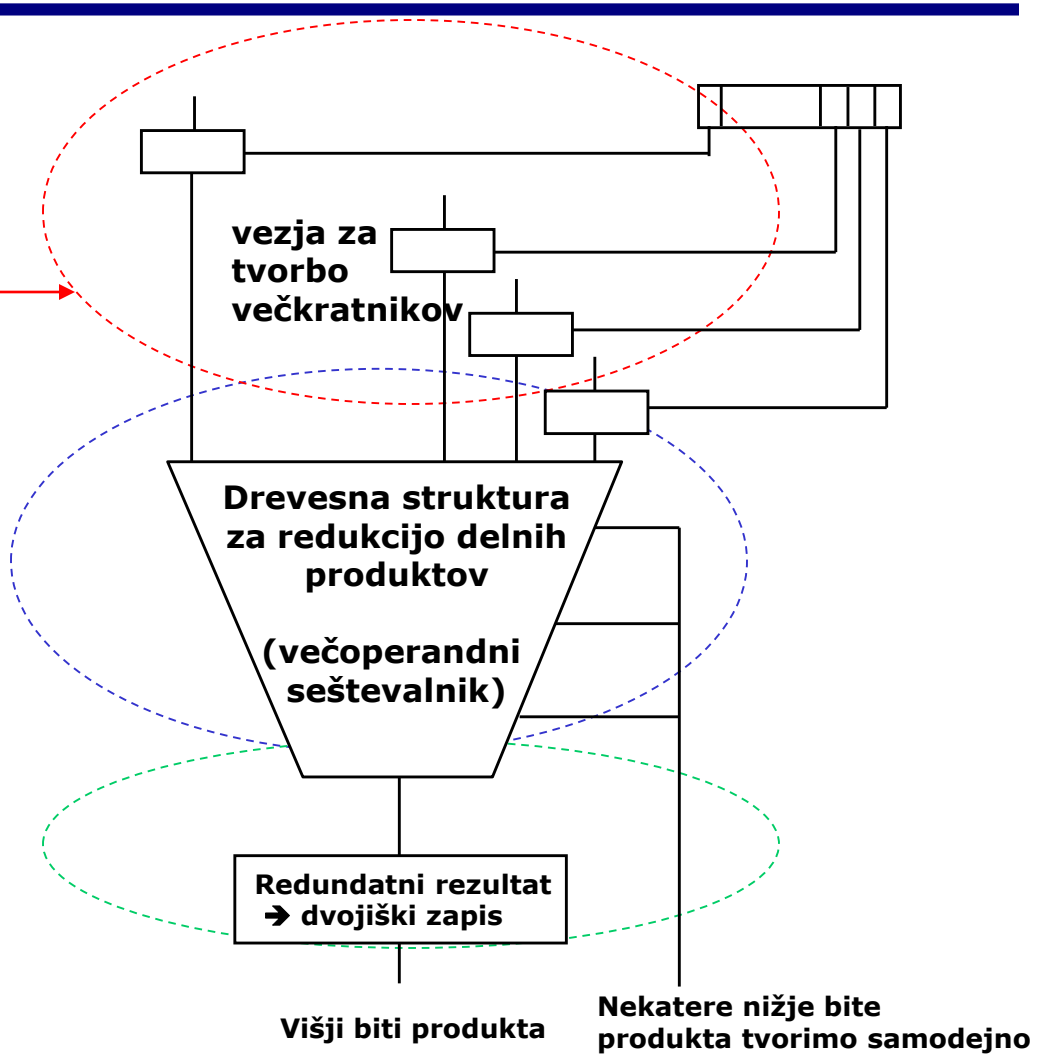
Arhitektura množenja z drevesnimi strukturami

Pristopi se razlikujejo glede na izvedbo treh elementov:

1. Vezja za tvorbo večkratnikov

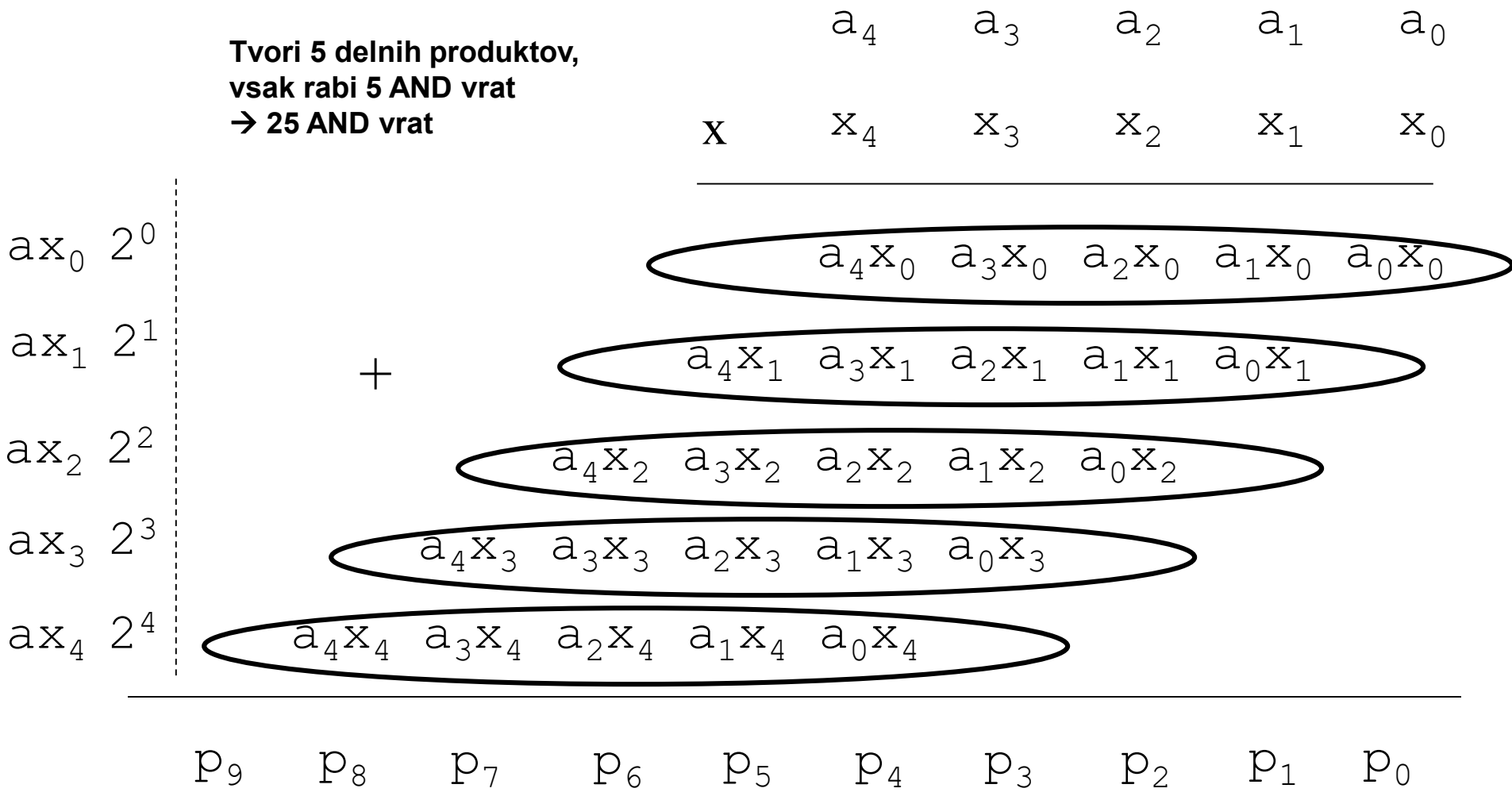
2. Redukcijsko drevo:
Drevesna struktura za redukcijo paralelnih produktov

3. Pretvornik redundantnega rezultata ➔ dvojiški zapis



Vezja za tvorbo večkratnikov

Tvori 5 delnih produktov,
vsak rabi 5 AND vrat
→ 25 AND vrat



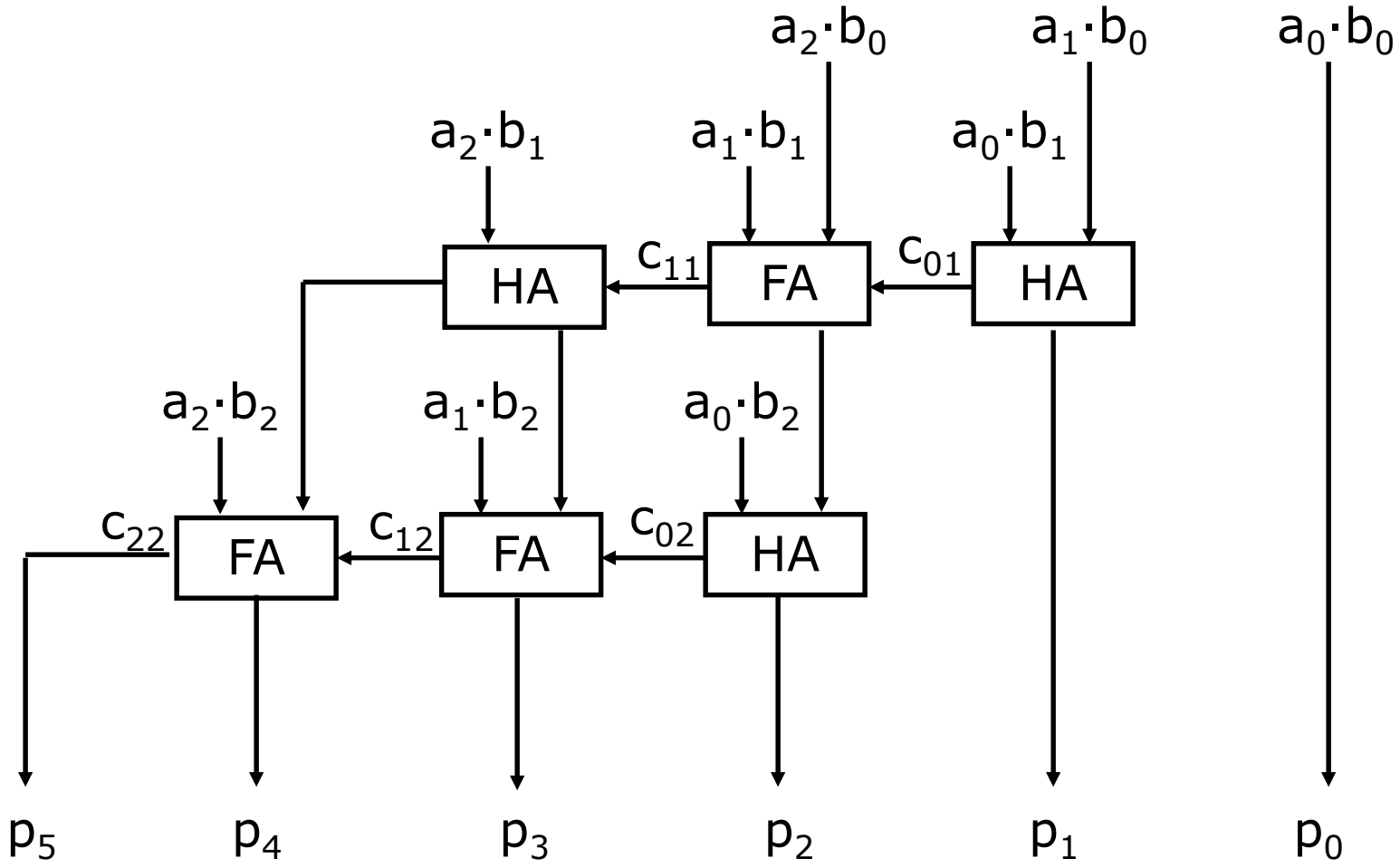
Načrtovanje digitalnih vezij

Matrično množenje

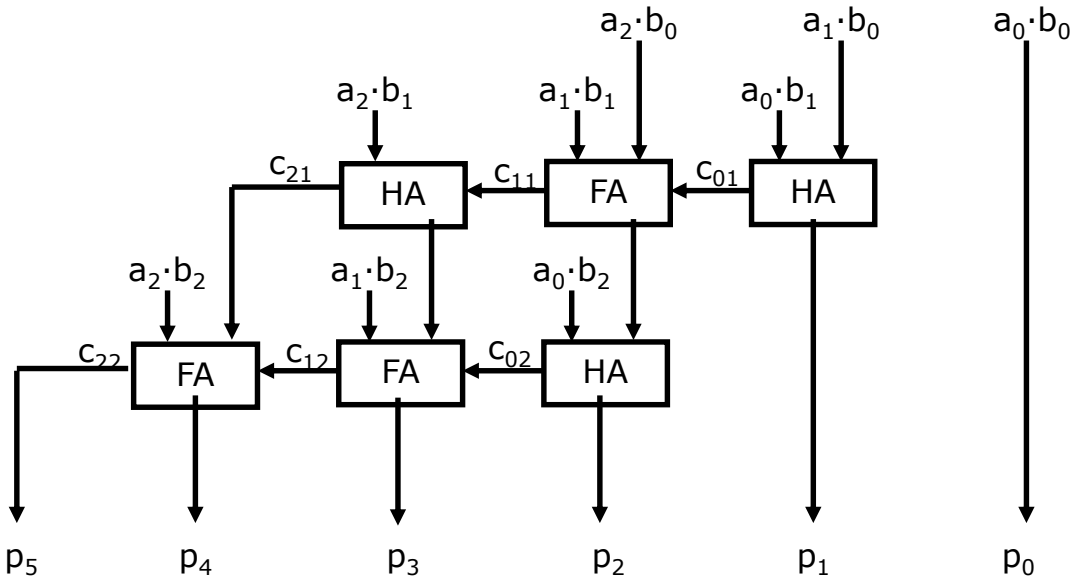
Matrični množilnik

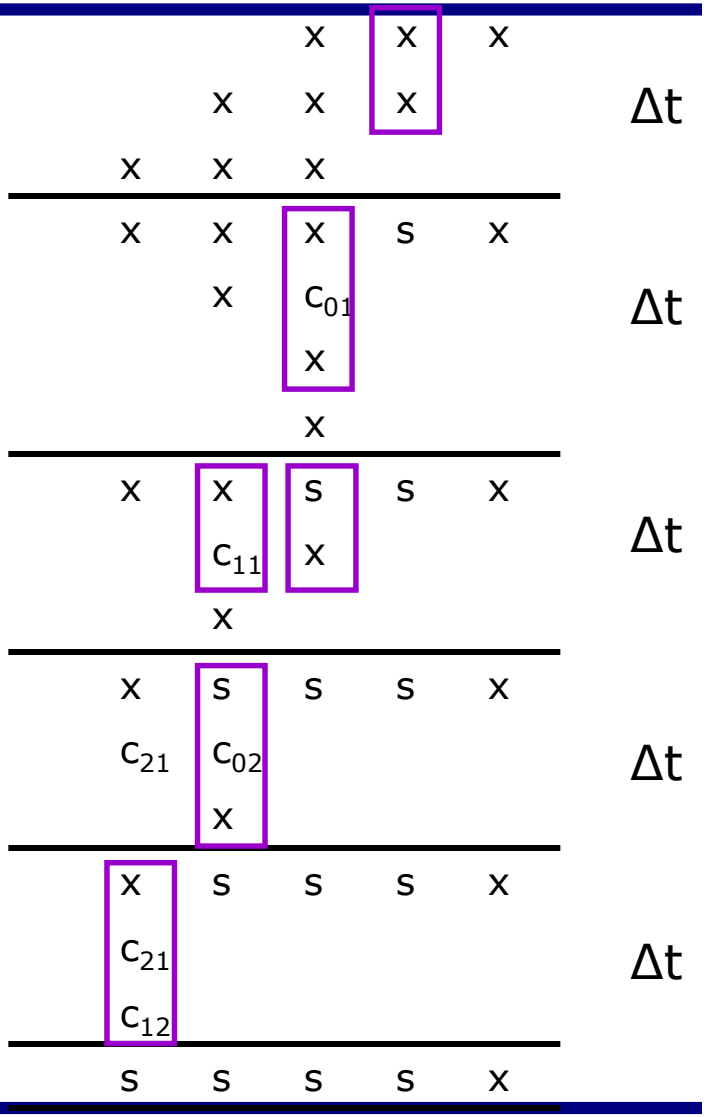
- Redukcijska drevesa, ki temeljijo na CSA so nepravilne strukture in zato povzročajo težave pri VLSI implementaciji (ožičenje, postavitev ….)
- Alternativa so matrični množilniki (array multiplier)
  - So počasnejši zaradi večjega števila stopenj
  - **Zelo primerni za realizacijo v VLSI**

3-bitni matrični množilnik



Zakasnitev 3-bitnega matričnega množilnika



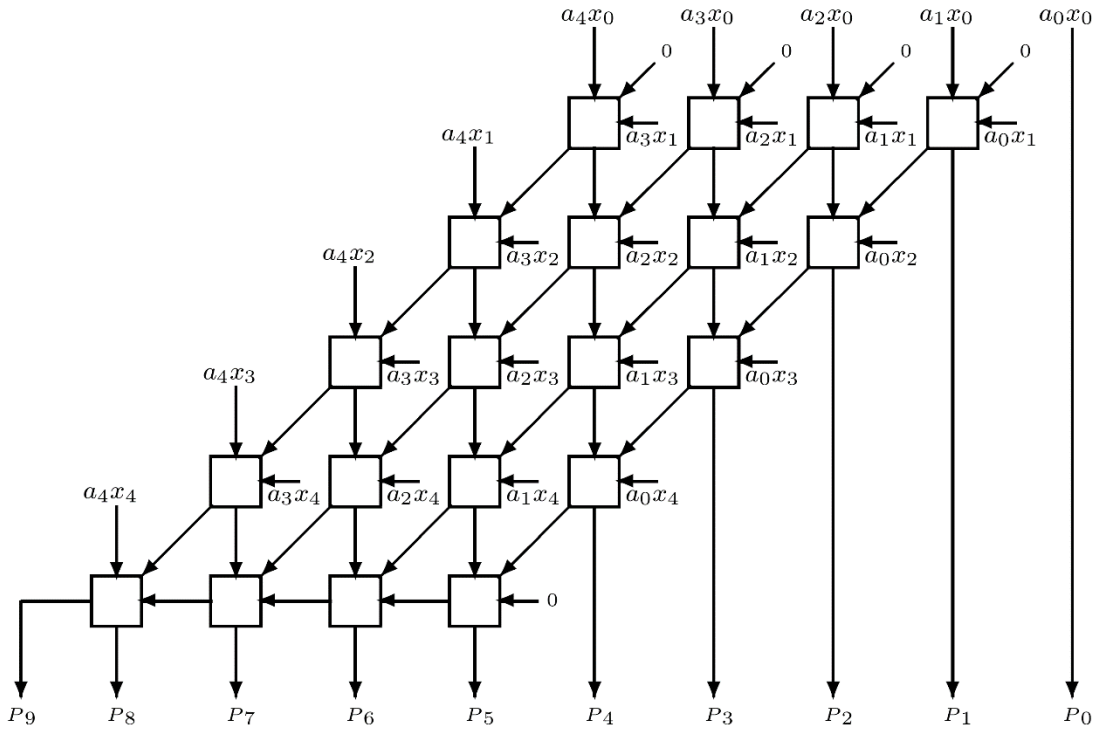


Zakasnitev n-bitnega matričnega množilnika = 3n-4.

Če je n=3 ➔ zakasnitev = 5*Δt

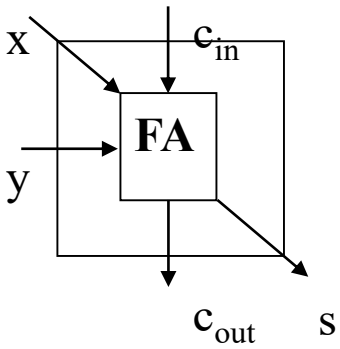
Nepredznačeni 5-bitni matrični množilnik

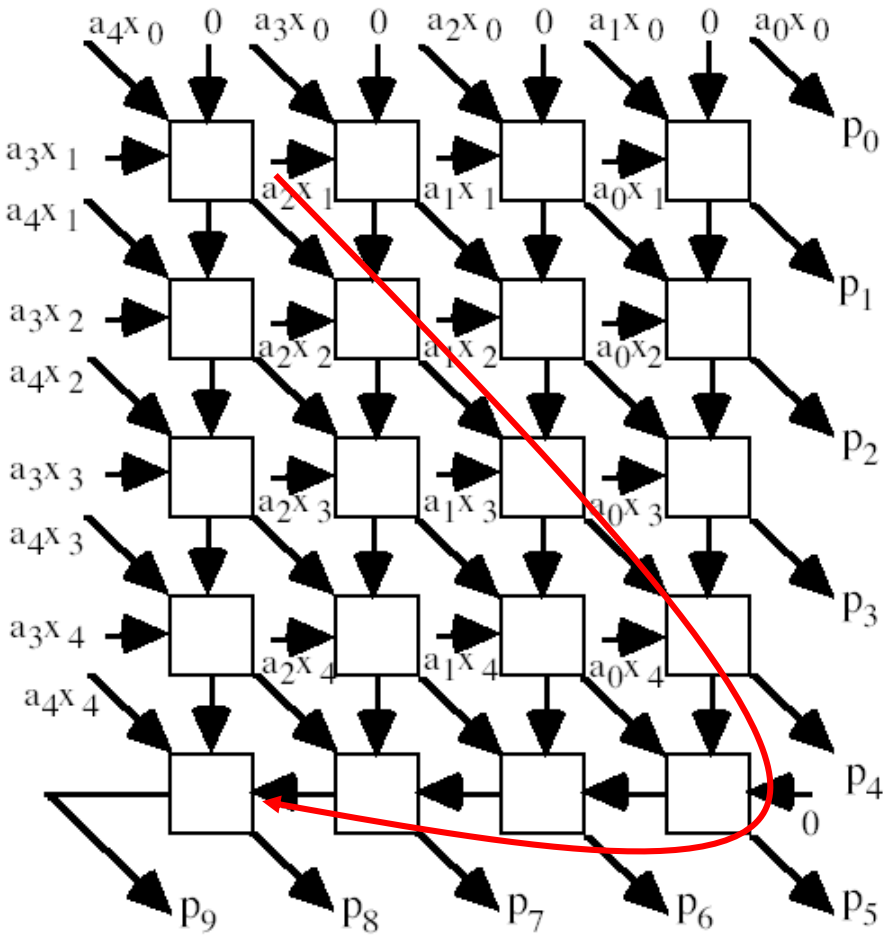
- Osnovna celica je FA,
- Na vhodu celice:
  - en bit delnega produkta aixj +
  - en bit predhodno seštetega delnega produkta +
  - bit vhodnega prenosa.
- Nima horizontalnega širjenja prenosa v prvih 4 vrsticah,
  - celice so carry-save tipa (izhod celice sta vsota in prenos)
- Zadnja vrstica je RC seštevalnik, ki ga lahko nadomestimo s hitrim seštevalnikom (recimo CLA).



Nepredznačeni 5-bitni matrični množilnik

Osnovna celica matričnega množilnika:





Kritična pot (predpostavimo da sta zakasnitvi vsote in prenosa enaki)

Načrtovanje digitalnih vezij

Realizacija delnih produktov

Realizacija večjih množilnikov z uporabo manjših

- Če želimo realizirati n x n bitni množilnik kot eno integrirano vezje lahko uporabimo enega od pristopov
- Recimo da želimo narediti 2n x 2n bitni množilnik, če imamo n x n bitne množilnike. Potrebujemo 4 (n x n) množilnike saj velja:

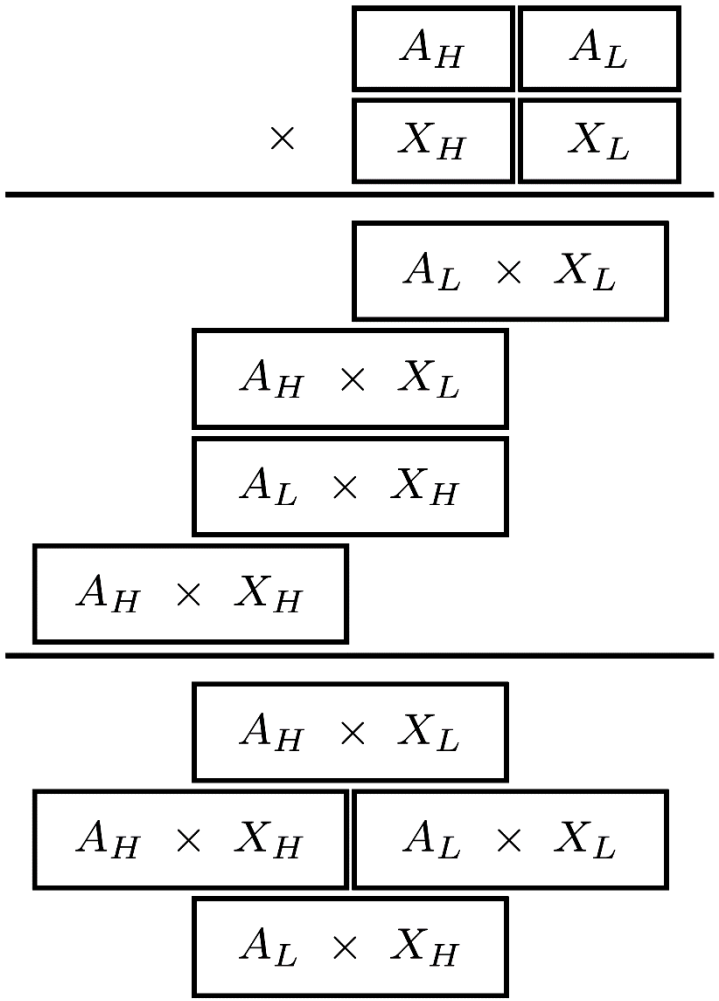
$$A \cdot X = (A_H \cdot 2^n + A_L) \cdot (X_H \cdot 2^n + X_L)$$

$$= A_H \cdot X_H \cdot 2^{2n} + (A_H \cdot X_L + A_L \cdot X_H) \cdot 2^n + A_L \cdot X_L$$

- A_H in A_L predstavljata najbolj in najmanj pomembni polovici množenca A;
- X_H in X_L predstavljata najbolj in najmanj pomembni polovici množitelja X

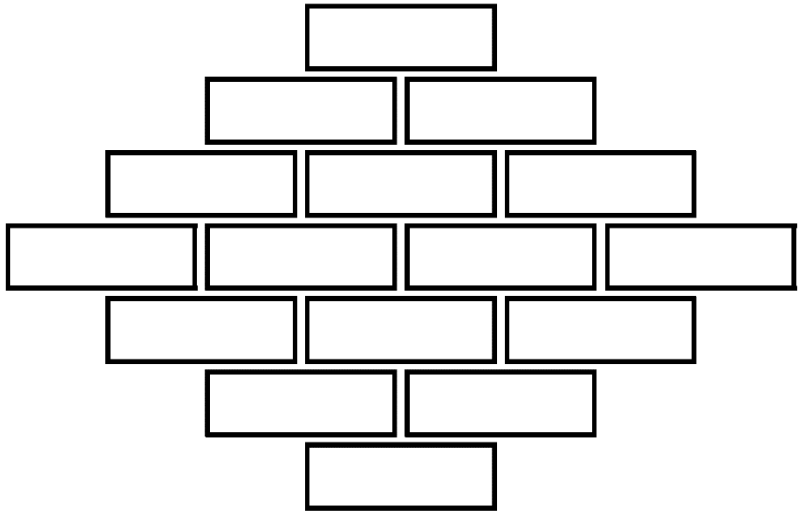
Poravnava delnih produktov

- 4 delne produkte dolžine 2n bitov moramo pravilno poravnati pred seštevanjem
- Zadnja postavitev predstavlja minimalno višino matrike: en nivo CSA in CPA
- n najmanj pomembnih bitov – so že biti v končnem produktu, zato ne rabimo dodatnega seštevanja
- 2n centralnih bitov seštejemo z 2n bitnim CSA, ki ima izhode povezane na CPA.
- n najbolj pomembnih bitov je povezanih na isti CPA, saj centralni biti lahko tvorijo prenos samo na mesta najbolj pomembnih bitov iz česar sledi da rabimo 3n bitni CPA



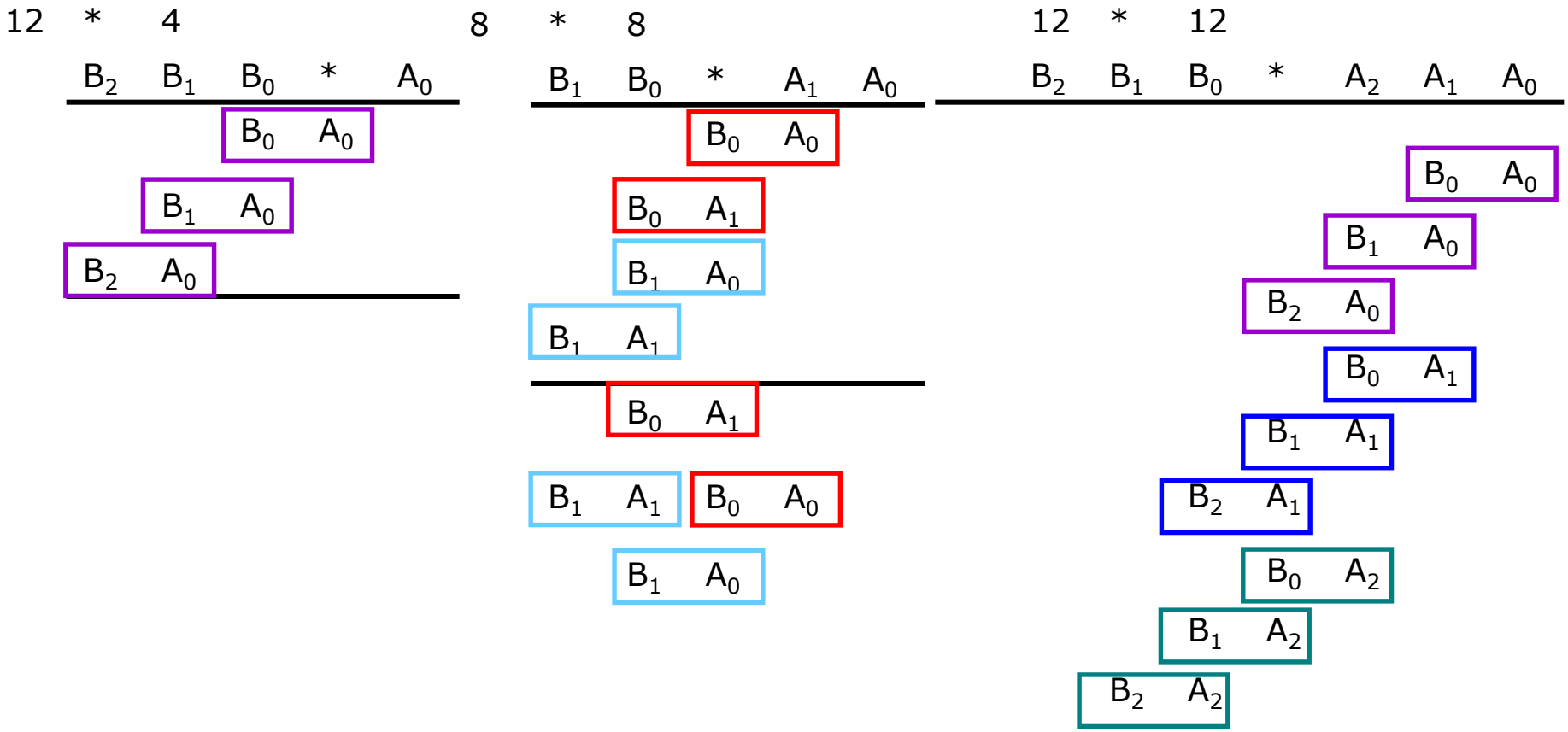
Razstavljanje večjega množilnika v manjše - razširitev

- Imamo osnovni množilnik je n x m biten, kjer velja $n \neq m$
- Z njim lahko realiziramo množilnike večje kot 2n x 2m
- Primer: 4n x 4n bitni množilnik, ki ga realiziramo z n x n bitnimi množilniki
  - 4 n x 4n bitni množilnik zahteva 4 2n x 2n bitne množilnike
  - 2n x 2n bitni množilnik zahteva 4 n x n bitne množilnike
  - Skupno torej 16 n x n bitnih množilnikov
  - 16 delnih produktov moramo poravnati pred seštevanjem



Razširitev: Velja za katerikoli kn x kn bitni množilnik (k je celo število)

Razstavljanje večjega množilnika v manjše - razširitev



Zmanjševanje števila delnih produktov

- Upoštevamo 2 ali več bitov množitelja naenkrat
- Zahteva tvorbo A (množenca), 2A, 3A
- Zmanjša število delnih produktov na n/2 – vsak korak je bolj zahteven
- Poznamo več algoritmov, ki ne povečajo kompleksnosti vezja – Booth-ov algoritem
- Cilj tega algoritma je tvorba manj delnih produktov za skupine *zaporednih ničel in enic*.

Načrtovanje digitalnih vezij

Zmanjšanje števila delnih produktov:
Booth-ov algoritem

Booth-ov algoritem

- Skupina zaporednih ničel v množilniku ne predstavlja novega delnega produkta – dobljeni delni produkt samo pomaknemo desno.
- Skupina m zaporednih enic v množilniku zahteva tvorbo manj kot m delnih produktov
- ...01…110... = ...10...000... - ...00...010...
- Uporaba SD (signed-digit) zapisa =...100...0$\bar{1}$0...
- Primer:
- ...011110... = ...100000... - ...000010... = ...100$\bar{1}$0...
  (decimalna oblika: 15=16-1)
  - Namesto da bi tvorili vseh m delnih produktov tvorimo samo 2
  - S tem da prvi delni produkt prištejemo, drugega odštejemo. Število pomikov za eno mesto se ohranja (m).

Booth-ov algoritem - pravila

- Kodiranje množitelja $x_{n-1}\, x_{n-2} \ldots x_1\, x_0$ v SD zapis
- Rezultat kodiranja množitelja $y_{n-1}\, y_{n-2} \ldots y_1\, y_0$
- Preverimo bita x_i in x_{i-1} množitelja da dobimo en bit y_i
- Predhodni bit x_{i-1} služi samo za referenco
- Naenkrat kodiramo dva sosedna bita, s tem da za soseda prvega bita ($i=0$) vzamemo $x_{-1}=0$
- Rezultat lahko dobimo z odštevanjem (XOR) sosedov $y_i = x_{i-1} - x_i$
- Ta postopek je vzporeden: Ni bistveno katera soseda vzamemo najprej.
- Primer:
  Množitelj $0011110011(0)$ kodiramo kot
  $0100\bar{1}0\bar{1}0\bar{1}$

  To pomeni 4 namesto 6 seštevanj/odštevanj!
  Odštevanje (-1) označimo s črto nad enico.

x_i	x_{i-1}	Operation	Comments	y_i
0	0	shift only	string of zeros	0
1	1	shift only	string of ones	0
1	0	subtract and shift	beginning of a string of ones	$\bar{1}$
0	1	add and shift	end of a string of ones	1

Bit predznaka

- 2'K zapis uporablja MSB bit kot bit predznaka (X_{n-1})
- Odloča o seštevanju/odštevanju – brez pomikanja
- Preverimo ga samo za negativne vrednosti množitelja ($X_{n-1}=1$)
- Ločimo dva primera:

$$A \cdot X = A \cdot \widetilde{X} - A \cdot x_{n-1} \cdot 2^{n-1} \quad \text{where} \quad \widetilde{X} = \sum_{j=0}^{n-2} x_j 2^j$$

1. Množenec A odštevamo – potrebna je korekcija
2. Brez bita predznaka – pregledamo niz enic in prištevamo na mestu n-1
   - Če je $x_{n-1}=1$ potem potrebnega prištevanja ne naredimo
   - Ekvivalentno odštevanju $A \cdot 2^{n-1}$ korekcijskega člena

	x_{n-1}	x_{n-2}	y_{n-1}
(1)	1	0	$\bar{1}$
(2)	1	1	0

Bit predznaka - primer

x_i	1	1	0	1	1	1	(0)	-9_{10}
y_i	0	$\bar{1}$	1	0	0	$\bar{1}$		

$$y_i = x_{i-1} - x_i$$

	x_{n-1}	x_{n-2}	y_{n-1}
(1)	1	0	$\bar{1}$
(2)	1	1	0

Primer množenja Booth radix 2

A		1	0	1	1			-5
X	$\times$	1	1	0	1			-3
Y		0	$\bar{1}$	1	$\bar{1}$			recoded multiplier
Add $-A$		0	1	0	1			
Shift		0	0	1	0	1		
Add A	$+$	1	0	1	1			
		1	1	0	1	1		
Shift		1	1	1	0	1	1	
Add $-A$	$+$	0	1	0	1			
		0	0	1	1	1	1	
Shift		0	0	0	1	1	1	1

Booth-ov algoritem - lastnosti

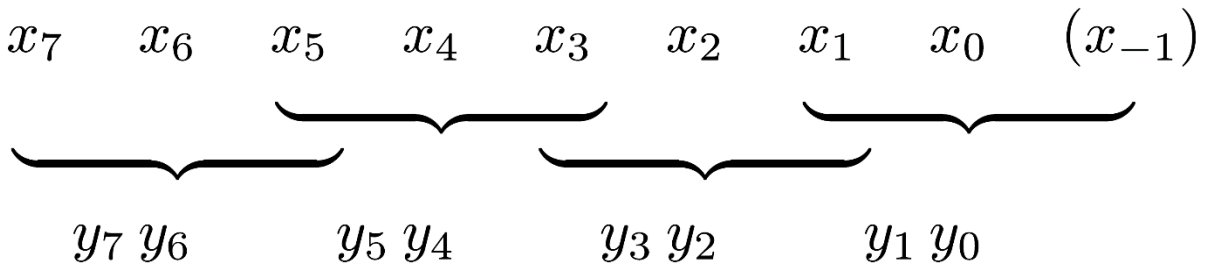
- Množenje se začne pri LSB
- Če začnemo pri MSB to pomeni daljši seštevalnik/odštevalnik pri širjenju prenosa
- Posebnega SD množilnika (ki zahteva 2 bita na mesto) ni treba graditi
  - Bite originalnega množitelja pregledamo in tvorimo kontrolne signale za seštevalnik/odštevalnik
- Pri Booth-ovem algoritmu je lahko množitelj v 2'K
  - Če množimo nepredznačena števila, potem dodamo 0 levo od množitelja ($x_n=0$) za zagotovitev pravilnosti.

Slabosti Booth-ega algoritma

- Spremenljivo število seštevanj/odštevanj in pomikov med dvema operacijama seštevanja/odštevanja
  - Zoprno pri načrtovanju sinhronega množilnika
- Algoritem ni učinkovit za osamljene enice!
- Primer:
- 001010101(0) kodiramo kot $01\bar{1}1\bar{1}1\bar{1}1\bar{1}$, kar zahteva 8 namesto 4 operacije
- Zato raje vzamemo 3 bite množitelja naenkrat namesto 2.
- Do sedaj opisanemu algoritmu pravimo včasih tudi Booth-ov algoritem v bazi 2 (ang. Radix-2)

Booth-ov algoritem v bazi 4

- Baza 4: (ang. Radix-4), uteži mest so 4^i i=0,1,2..
- Bita x_i in x_{i-1} kodiramo v y_i in y_{i-1} pri tem x_{i-2} služi kot referenčni bit
- Podobno x_{i-2} in x_{i-3} kodiramo v y_{i-2} in y_{i-3} pri tem x_{i-4} služi kot referenčni bit
- Skupine 3 bitov se prekrivajo – skrajno desna skupina x_1 x_0 (x_{-1}), sledi ji x_3 x_2 (x_1) ...



Zgled Booth-ovega množilnika: C:\vhdl\mult_booth_8x8\booth_multiplier.vhd

Radix-4 algoritem - pravila

- Opazujmo liha mesta i = 1, 3, 5, 7…
- Osamljene 0/1 so obravnavane učinkovito:
- Če je x_{i-1} osamljena 1, potem je $y_{i-1}=1$, torej rabimo samo eno operacijo
- Če je x_{i-1} osamljena 0 v nizu 1 …$\bar{1}$0(1)… to kodiramo kot …11… ali …0$\bar{1}$..➔ ena operacija
- Da dobimo zahtevano operacijo izračunamo $x_{i-1}+x_{i-2}-2x_i$ za lihe i in rezultat predstavimo kot 2 bitno dvojiško število y_iy_{i-1} v zapisu SD.

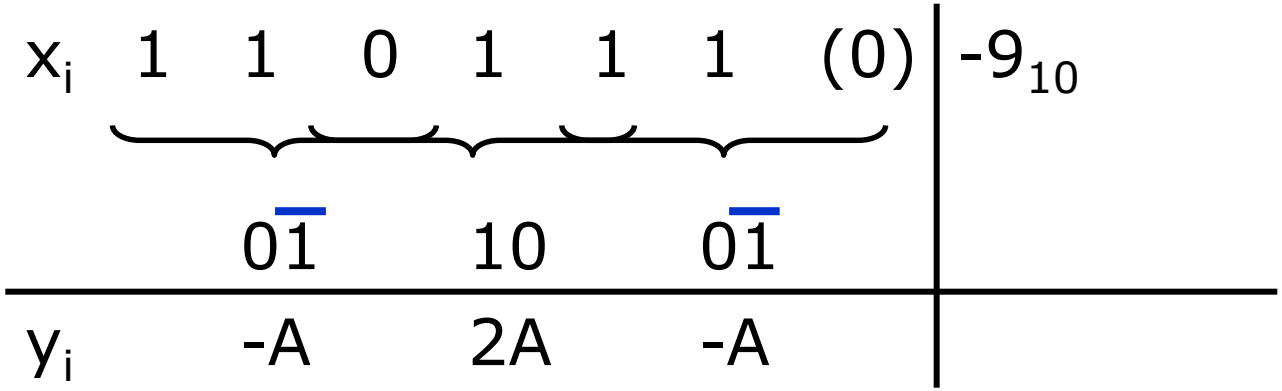
Preslika vrednosti intervala [0..3] na interval [-2..2].
Formula preslikave:
$-2x_i+x_{i-1}+x_{i-2}$

Tabela preslikave:

x_i	x_{i-1}	x_{i-2}	y_i	y_{i-1}	operation	comments
0	0	0	0	0	$+0$	string of zeros
0	1	0	0	1	$+A$	a single 1
1	0	0	$\bar{1}$	0	$-2A$	beginning of 1's
1	1	0	0	$\bar{1}$	$-A$	beginning of 1's
0	0	1	0	1	$+A$	end of 1's
0	1	1	1	0	$+2A$	end of 1's
1	0	1	0	$\bar{1}$	$-A$	a single 0
1	1	1	0	0	$+0$	string of 1's

Zgled Booth-ovega kodirnika:
C:\vhdl\mult_booth_8x8\booth_encoder.vhd

Radix-4 algoritem - primer



Formula preslikave: $-2x_i+x_{i-1}+x_{i-2}$

Tabela preslikave:

x_i	x_{i-1}	x_{i-2}	y_i	y_{i-1}	operation	comments
0	0	0	0	0	$+0$	string of zeros
0	1	0	0	1	$+A$	a single 1
1	0	0	$\bar{1}$	0	$-2A$	beginning of 1's
1	1	0	0	$\bar{1}$	$-A$	beginning of 1's
0	0	1	0	1	$+A$	end of 1's
0	1	1	1	0	$+2A$	end of 1's
1	0	1	0	$\bar{1}$	$-A$	a single 0
1	1	1	0	0	$+0$	string of 1's

Radix-4 in Radix-2 algoritem

- 01|01|01|01|(0) kodiramo kot 01|01|01|01 - število operacij ostaja 4, kar je najmanj.
- 00|10|10|10|(0) kodiramo kot 01|$\bar{0}$1|0$\bar{1}$|$\bar{1}$0, število operacij je 4, namesto 3.
- Radix-4 algoritem ima manj vzorcev z več delnimi produkti, in posledično manjši prirastek števila operacij
- S tem pristopom lahko realiziramo tudi n biten sinhronski množilnik, ki tvori natanko n/2 delnih produktov.
- Sode n – množilnike v 2'K izdelamo brez dodelav algoritma,
- Lihe n – množilnike v 2'K izdelamo z razširitvijo bita predznaka: Dodamo eno ničlo levo od množitelja, če množimo nepredznačene vrednosti in n je lih – dve ničli če je n sod.

Primer

A			01	00	01			17
X	$\times$		11	01	11			-9
Y			$0\bar{1}$	10	$0\bar{1}$			recoded multiplier
			$-A$	$+2A$	$-A$			operation
Add $-A$	$+$		10	11	11			
2-bit Shift		1	11	10	11	11		
Add $2A$	$+$	0	10	00	10			
			01	11	01	11		
2-bit Shift			00	01	11	01	11	
Add $-A$	$+$		10	11	11			
			11	01	10	01	11	-153

- Korakov: n/2=3 ➔ zmanjša 2 bita množitelja v vsakem koraku
- Vsi pomiki so dvomestni
- Potrebujemo dodaten bit da shranimo predznak pri prištevanju 2A

Radix-8 Booth-ov algoritem

- Kodiranje je raztegnjeno na 3 bite naenkrat – prekrivanje skupin 4 bitov
- Tvorimo samo n/3 delnih produktov – vendar potrebujemo večkratnik 3A kar je zahteven korak
- Primer: kodiranje 010(1) da rezultat $y_i\ y_{i-1}\ y_{i-2}=011$
- Obstaja način za poenostavljanje tvorbe in prištevanja ±3A
- Če želimo najti minimalno število operacij prištevanja/odštevanja za dani množitelj moramo najti njegovo minimalen SD zapis.
- Slednje predstavlja SD zapis, ki ima najmanjše število neničelnih mest:

$$min \ \sum_{i=0}^{n-1} |y_i|$$

Določanje minimalnega SD zapisa množitelja

- $y_{n-1}y_{n-2}\ldots y_0$ je minimalna predstavitev števila v SD zapisu če velja $y_i \cdot y_{i-1}=0$ za $1 \leq i \leq n-1$, s tem da MSB biti ustrezajo pogoju $y_{n-1} \cdot y_{n-2} \neq 1$
- Primer: Predstavitev 7 s 3 biti 111 je minimalna kljub temu da velja $y_i \cdot y_{i-1} \neq 0$
- Za katerikoli množitelj izpolnimo zgornji pogoj tako da dodamo 0 mesto višje od MSB

x_{i+1}	x_i	c_i	y_i	c_{i+1}	Comments
0	0	0	0	0	string of 0's
0	1	0	1	0	a single 1
1	0	0	0	0	string of 0's
1	1	0	$\bar{1}$	1	beginning of 1's
0	0	1	1	0	end of 1's
0	1	1	0	1	string of 1's
1	0	1	$\bar{1}$	1	a single 0
1	1	1	0	1	string of 1's

Kanonsko kodiranje

- Bite množitelja jemljemo posamezno z desne s tem da x_{i+1} predstavlja referenčni bit
- Za pravilno obravnavo osamljenih 0/1 v nizu enic/ničel rabimo informacijo o nizu bolj desno od opazovanega mesta
- Bit "prenosa" je 0 če gre za niz ničel in 1 če gre za niz enic
- Kodirani množitelj lahko uporabljamo brez popravkov, če je zapisan v 2'K
- Razširitev bita predznaka

  x_{n-1} - $x_{n-1}x_{n-1}x_{n-2}...x_0$
- Lahko razširimo na dva ali več bitov naenkrat
- Potrebujemo mnogokratnike za 2 bita $\pm A$ in $\pm 2A$

x_{i+1}	x_i	c_i	y_i	c_{i+1}	Comments
0	0	0	0	0	string of 0's
0	1	0	1	0	a single 1
1	0	0	0	0	string of 0's
1	1	0	$\bar{1}$	1	beginning of 1's
0	0	1	1	0	end of 1's
0	1	1	0	1	string of 1's
1	0	1	$\bar{1}$	1	a single 0
1	1	1	0	1	string of 1's

Kanonsko kodiranje - primer

x_i	0	0	1	1	0	1	1	1	0	110_{10}
c_i		1	1	1	1	1	1	0	0	
y_i		1	0	0	$\bar{1}$	0	0	$\bar{1}$	0	128-16-2

x_{i+1}	x_i	c_i	y_i	c_{i+1}	Comments
0	0	0	0	0	string of 0's
0	1	0	1	0	a single 1
1	0	0	0	0	string of 0's
1	1	0	$\bar{1}$	1	beginning of 1's
0	0	1	1	0	end of 1's
0	1	1	0	1	string of 1's
1	0	1	$\bar{1}$	1	a single 0
1	1	1	0	1	string of 1's

Slabosti kanonskega kodiranja

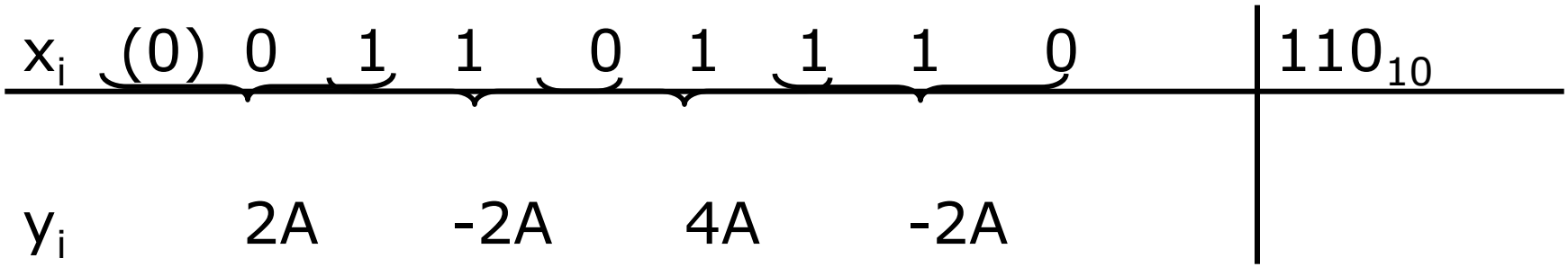
- Bite množitelja pridobivamo zaporedno!
- V Booth-ovem algoritmu ni širjenja "prenosa", kar pomeni da so delni produkti izračunani vzporedno in lahko uporabimo seštevalnik več operandov
- Če želimo izkoristiti minimalno število operacij, mora biti število seštevanj/odštevanj in dolžin pomikov spremenljivo, kar težko dosežemo
- Za pomike enotne dolžine imamo $n/2$ delnih produktov kar je več kot minimum v kanonskem kodiranju

Spremenjen Booth Radix 4 algoritem

- Zmanjšanje števila delnih produktov ob enotnih 2-bitnih pomikih
- x_{i+1} naj bo referenčni bit x_i x_{i-1} če je i lih
- Delne produkte $\pm 2A, \pm 4A$ tvorimo s pomikanjem
  (pomik levo = množenje z 2)
- Delni produkt $4A$ tvorimo takrat ko imamo $(x_{i+1})x_i\, x_{i-1}=(0)11$, kar je niz enic za $(x_{i+3})x_{i+2}\, x_{i+1}$ če imamo 0 na skrajnem desnem mestu
  - Ni kodiranje, ker ne moremo zapisati 4 z dvema bitoma
  - Število delnih produktov je vedno $n/2$
  - Množitelji v 2'K – razširimo bit predznaka
  - Nepredznačena števila: Dodamo eno ali dve ničli levo od množitelja

x_{i+1}	x_i	x_{i-1}	Operation	Comments
0	0	0	$+0$	string of 0's
0	0	1	$+2A$	end of 1's
0	1	0	$+2A$	a single 1
0	1	1	$+4A$	end of 1's
1	0	0	$-4A$	beginning of 1's
1	0	1	$-2A$	a single 0
1	1	0	$-2A$	beginning of 1's
1	1	1	$+0$	string of 1's

Primer



Formula preslikave: $-4x_{i+1}+2x_i+2x_{i-1}$

x_{i+1}	x_i	x_{i-1}	Operation	Comments
0	0	0	$+0$	string of 0's
0	0	1	$+2A$	end of 1's
0	1	0	$+2A$	a single 1
0	1	1	$+4A$	end of 1's
1	0	0	$-4A$	beginning of 1's
1	0	1	$-2A$	a single 0
1	1	0	$-2A$	beginning of 1's
1	1	1	$+0$	string of 1's

Popravek pri LSB

- Za najbolj desni par bitov x_1x_0, če je $x_0 = 1$ predpostavimo začetek niza enic, ki ga dejansko ni (saj je x_0=LSB), kar pomeni da ni bilo odštevanja
- Primer: množitelj 01110111 – delni produkti:

$$\begin{array}{rcccc} & 01 & 11 & 01 & 11 \\ & +2A & +0 & -2A & +0 \\ \text{instead of} & +2A & +0 & -2A & -A \end{array}$$

- Popravek: Ko je x_0=1 postavimo delni produkt na -A namesto na 0
- Imamo 4 možne primere:

x_2	x_1	x_0	Operation
0	0	1	$+2A - A = A$
0	1	1	$+4A - A = 3A$
1	0	1	$-2A - A = -3A$
1	1	1	$0 - A = -A$

Primer

- Predhodni primer:

A			01	00	01			17
X	$\times$	(1)	11	01	11			-9
			0	$-2A$	0			Operation
Initial $-A$			10	11	11			
Add 0	+		00	00	00			
			10	11	11			
2-bit Shift		1	11	10	11	11		
Add $-2A$	+	1	01	11	10			
		1	01	10	01	11		
2-bit Shift			11	01	10	01	11	
Add 0	+		00	00	00			
			11	01	10	01	11	-153

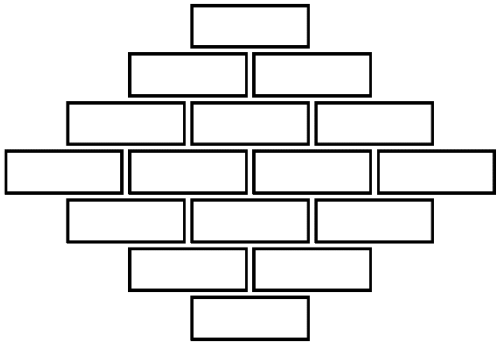
- Bit predznaka množitelja razširimo tako, da nad prvim parom bitov množitelja ni potrebna nobena operacija
- Zaradi mnogokratnikov -2A potrebujemo dodatni bit predznaka da shranimo predznak

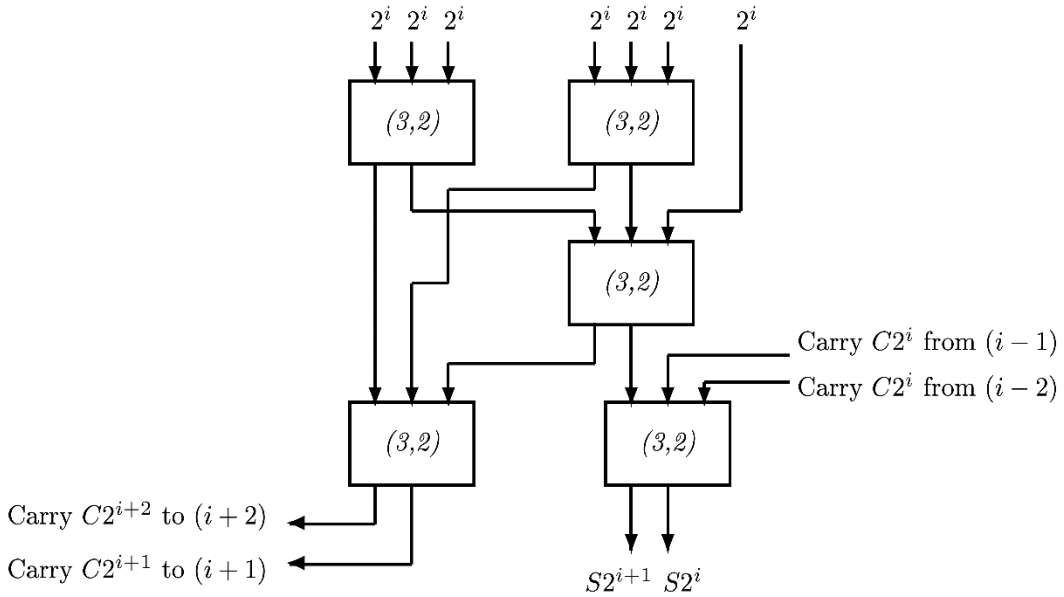
Razširitev Booth radix 4 algoritma

- Opisano metodo lahko razširimo na 3 bite ali več na vsakem koraku
- Vendar potem potrebujemo večkratnike množenca A kot npr. 3A ali celo 6A
  - Pripravimo vnaprej in shranimo
  - Izvajamo dve seštevanji v enem koraku
- Primer:
  - Za (0)101 rabimo 8-2=6,
  - Za (1)001 rabimo -8+2=-6

Seštevanje delnih produktov

Ko smo poravnali 16 produktov dobimo 7 bitov v enem stolpcu, ki jih moramo sešteti.
PŠ(7,3) s katerimi 3 operande seštejemo s PŠ(3,2) kar tvori 2 operanda, ki ju seštejemo s CPA





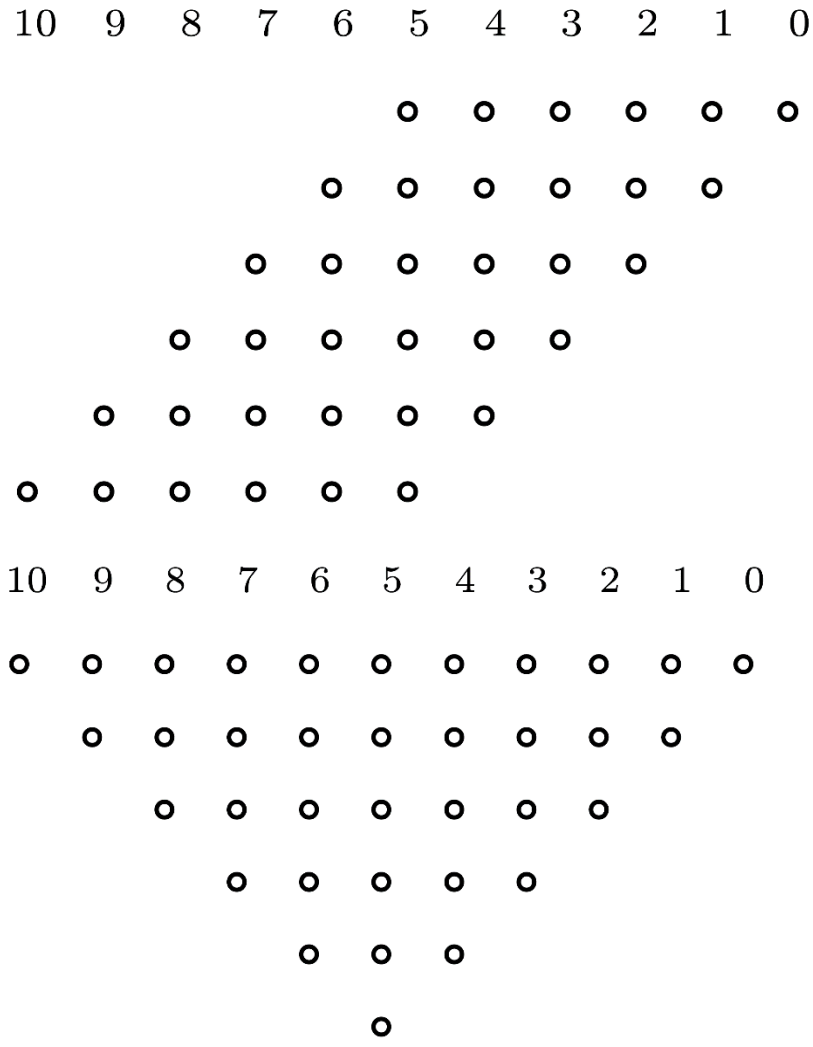
Seštevanje delnih produktov

- Nastopi po tvorbi delnih produktov – neposredno ali z uporabo manjših množilnikov
- Nastale delne produkte moramo sešteti da dobimo končni produkt
  - z uporabo hitrega seštevalnika več operandov
- Izkoristili bomo posebno obliko delnih produktov, saj tako zmanjšamo kompleksnost realizacije
- Delni produkti imajo manj bitov kot končni produkt in jih moramo poravnati pred seštevanjem
- Pričakujemo veliko stolpcev ki imajo mnogo manj bitov kot skupno število delnih produktov. Slednje narekuje uporabo enostavnih števnikov!

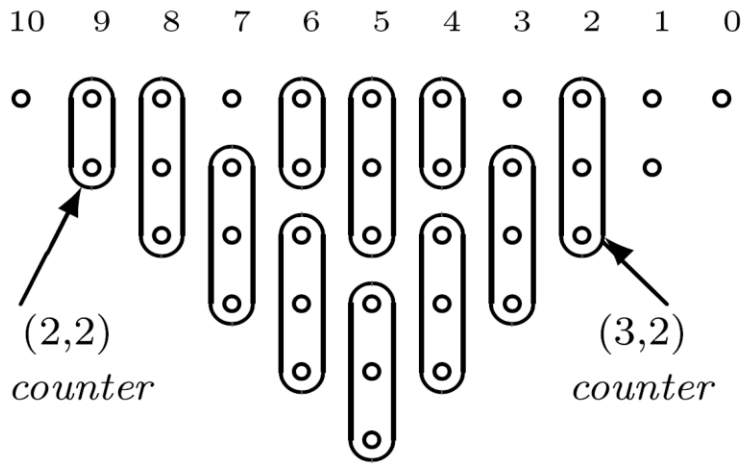
Primer – 6 delnih produktov

- To strukturo dobimo ko množimo nepredznačene 6 bitne operande z algoritmom "en bit naenkrat"
- 6 operandov lahko seštejemo z 3-nivojskim CSA drevesom

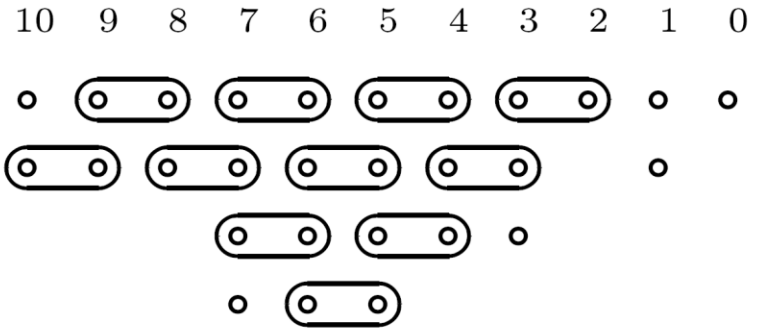
- Število PŠ(3,2) lahko močno zmanjšamo če izkoristimo dejstvo da vsi stolpci (razen enega) vsebujejo manj kot 6 bitov
- Ponovno narišemo matriko bitov seštevanja.



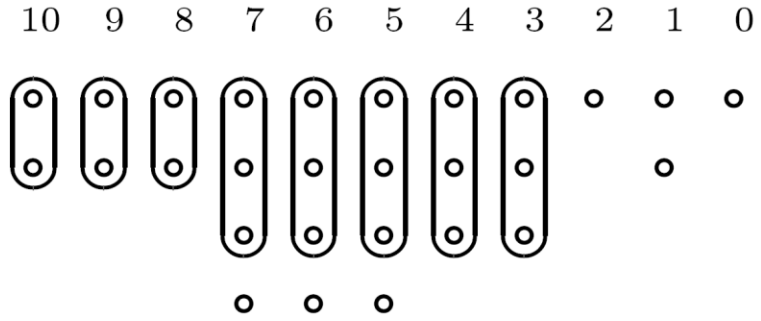
Poenostavljanje z uporabo PŠ(2,2)



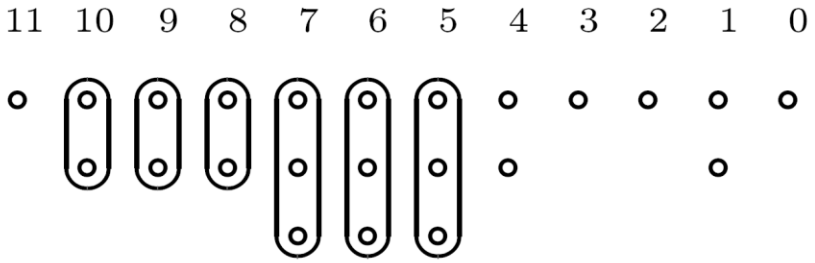
(a) Level 1 carry-save addition.



(b) Results of level 1.



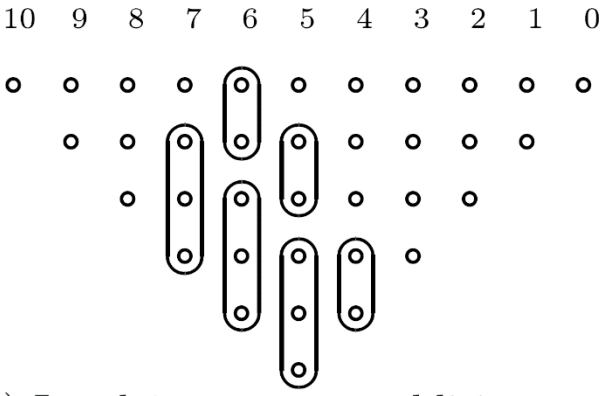
(c) Level 2 carry-save addition.



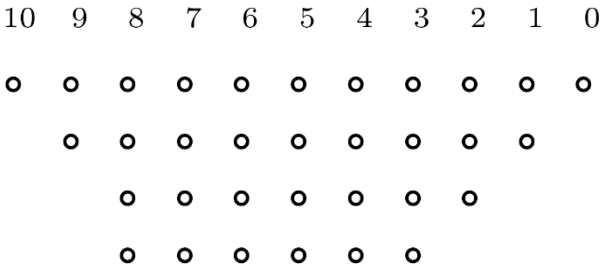
(d) Level 3 carry-save addition.

Dodatno zmanjšanje števila PŠ

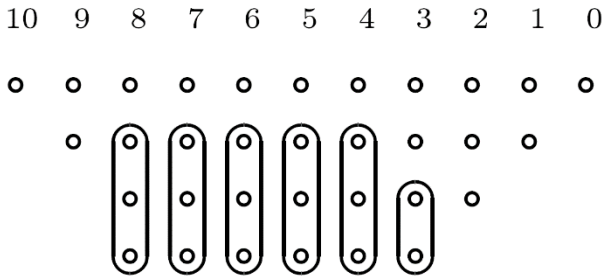
- V vsakem koraku zmanjšamo število bitov stolpca k najbližjemu elementu zaporedja 3, 4, 6, 9, 13, 19, …
- 15 FA in 5HA proti 16 FA in 9 HA



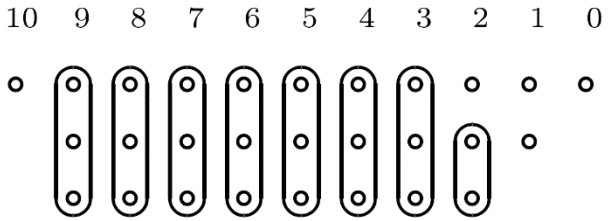
(a) Level 1 carry-save addition.



(b) Results of level 1.



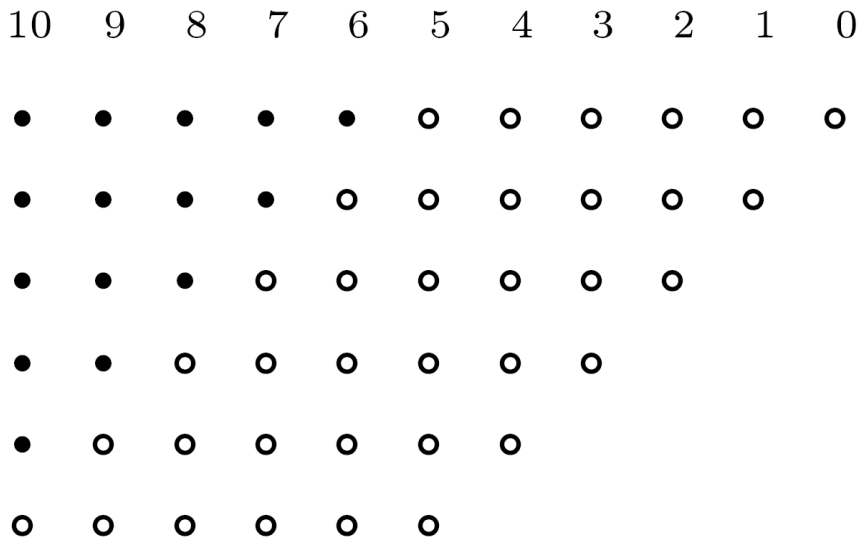
(c) Level 2 carry-save addition.



(d) Level 3 carry-save addition.

Spremenjena matrika za negativna števila

- Bit predznaka moramo primerno razširiti
- V prvi vrstici imamo 11 namesto 6 bitov …
- Poveča zapletenost seštevalnika več operandov
- Če 2'K tvorimo z uporabo 1'K se zapletenost matrike še poveča
- Kako se lahko izognemo seštevanju dodatnih '1' predznaka?



Zmanjševanje števila enic predznaka

- Število zapisano v 2'K

  S S S S S S z_4 z_3 z_2 z_1 z_0

  z vrednostjo

$$-s \cdot 2^{10} + s \cdot 2^9 + s \cdot 2^8 + s \cdot 2^7 + s \cdot 2^6 + s \cdot 2^5 + z_4 \cdot 2^4 + z_3 \cdot 2^3 + z_2 \cdot 2^2 + z_1 \cdot 2^1 + z_0$$

- Zamenjamo s številom

  0 0 0 0 0 (-s) z_4 z_3 z_2 z_1 z_0

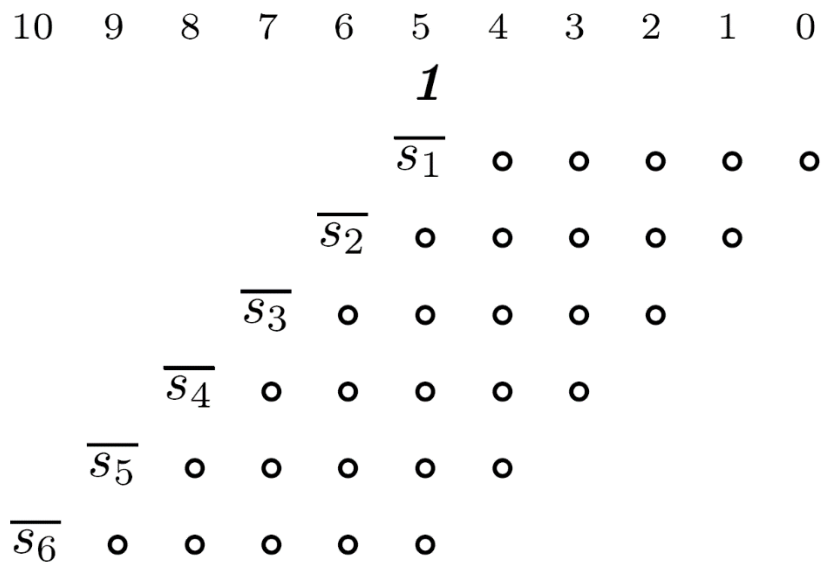
  saj velja

$$-s \cdot 2^{10} + s \cdot (2^9 + 2^8 + 2^7 + 2^6 + 2^5)$$

$$= -s \cdot 2^{10} + s \cdot (2^{10} - 2^5) = \quad -s \cdot 2^5$$

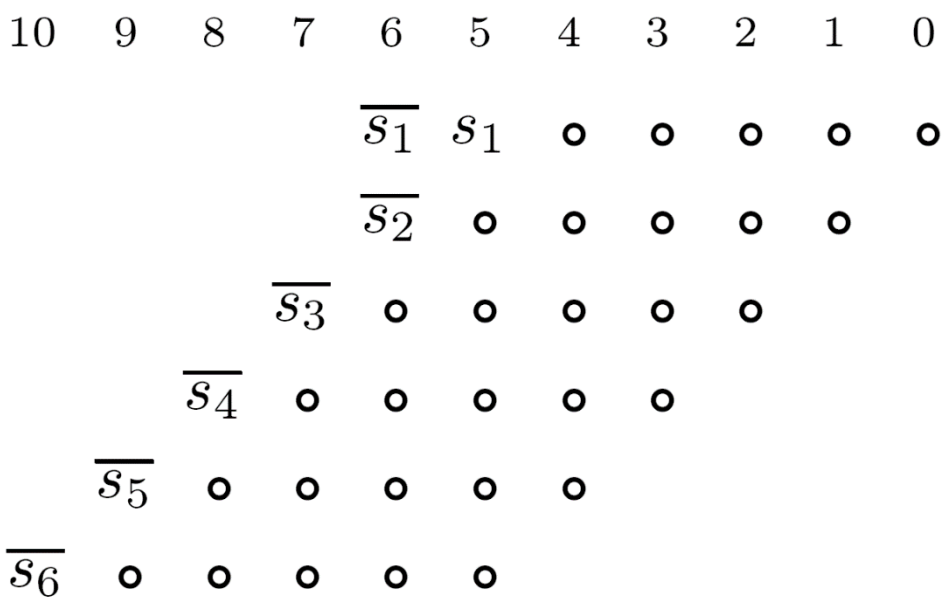
Nova matrika bitov

- Da dobimo vrednost -s v stolpcu 5 komplementiramo s na (1-s) in prištejemo 1
  - Prenos 1 na stolpec 6 služi kot dodatna 1, ki jo potrebujemo za bit predznaka drugega delnega produkta
- Nastala matrika ima manj bitov vendar je bolj visoka (7 namesto 6)



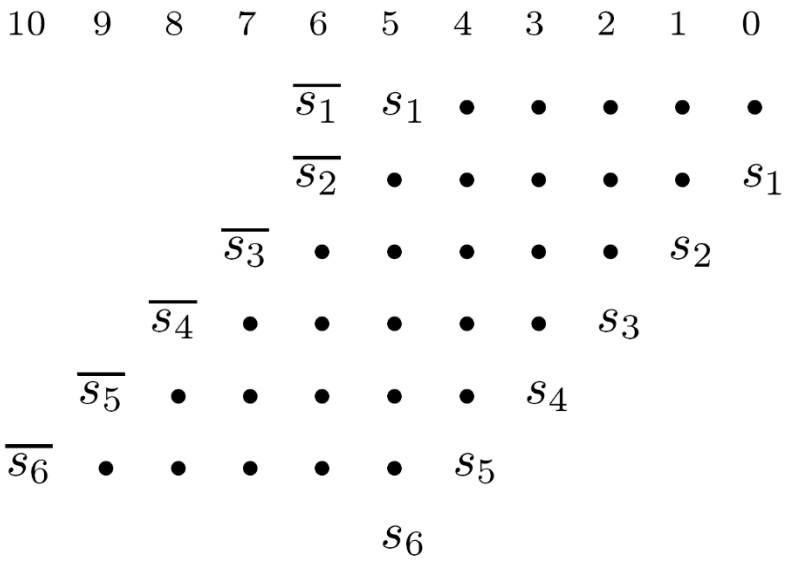
Eliminacija dodatne enice v 5. stolpcu

- Postavimo dva bita predznaka s_1 in s_2 v isti stolpec
- Velja $(1-s_1)+(1-s_2) = 2 - s_1 - s_2$
- 2 predstavlja izhodni prenos na naslednji stolpec
- To dosežemo tako da najprej razširimo bit predznaka s_1



Uporaba 1'K in prenosa

- Dodamo prenose v matriko
- Komplemente ustreznih bitov izračunamo vedno ko je $S_i=1$
- Dodaten S_6 v stolpcu 5 poveča višino drevesa na 7
- Če je zadnji delni produkt vedno pozitiven (množitelj je pozitiven), potem lahko S_6 eliminiramo



Primer

- Množitelj kodiran s kanonskim kodiranjem

A					0	1	0	1	1	0	22
X			$\times$		0	0	1	0	1	1	11
Y					0	1	0	$\bar{1}$	0	$\bar{1}$	Recoded multiplier
1	**1**	**1**	**1**	**1**	**1**	0	1	0	1	0	
0	**0**	**0**	**0**	**0**	0	0	0	0	0		
1	**1**	**1**	**1**	0	1	0	1	0			
0	**0**	**0**	0	0	0	0	0				
0	**0**	1	0	1	1	0					
0	0	0	0	0	0						
0	0	0	1	1	1	1	0	0	1	0	

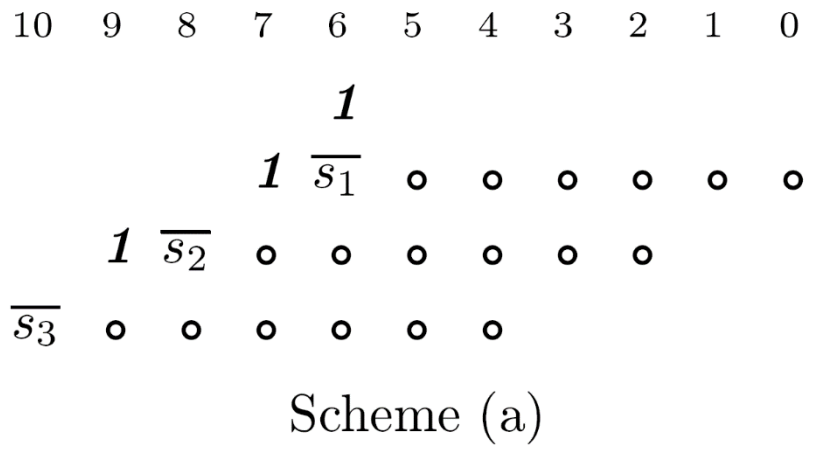
Manjše drevo za izdelani primer

10	9	8	7	6	5	4	3	2	1	0
				0	**1**	0	1	0	1	0
				1	0	0	0	0	0	
			0	0	1	0	1	0		
		1	0	0	0	0	0			
	1	1	0	1	1	0				
1	0	0	0	0	0					
0	0	0	1	1	1	1	0	0	1	0

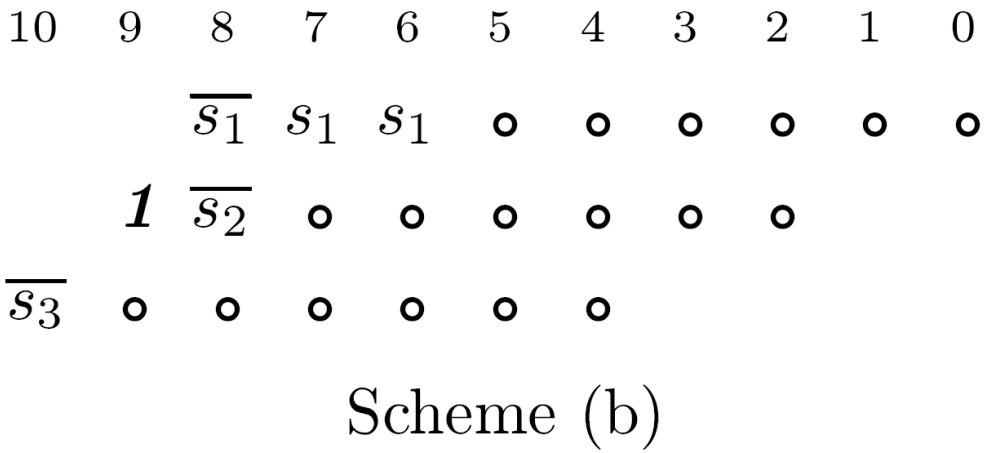
Primer ob uporabi 1'K in prenosa

10	9	8	7	6	5	4	3	2	1	0
				0	**1**	0	1	0	0	1
				1	0	0	0	0	0	**1**
			0	0	1	0	0	1	**0**	
		1	0	0	0	0	0	**1**		
	1	1	0	1	1	0	**0**			
1	0	0	0	0	0	**0**				
					0					
0	0	0	1	1	1	1	0	0	1	0

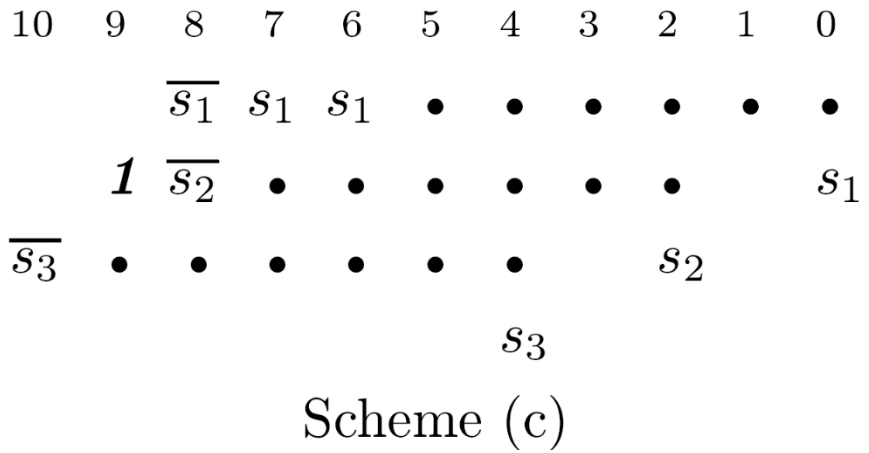
Primer: Radix-4 Booth-ov algoritem



Scheme (a)



Scheme (b)

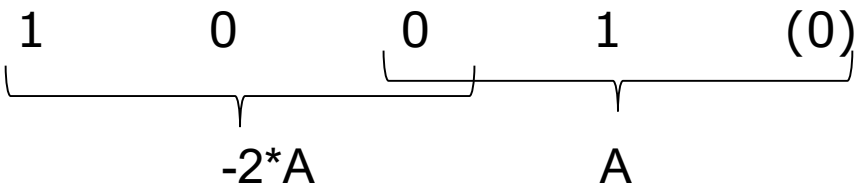


Scheme (c)

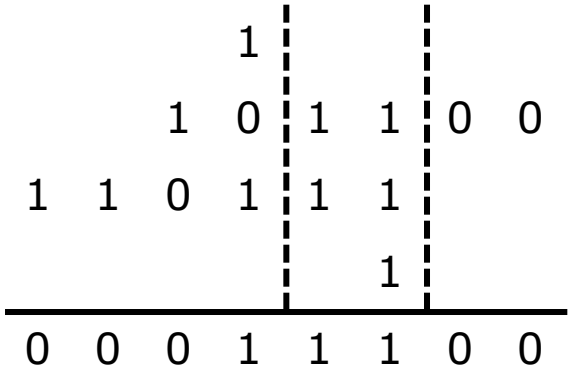
Primer: Radix-4 Booth-ov algoritem

A*X=(-4)*(-7)=28

X=(-7) =



A	1	1	1	1	0	0	0	0
-A	0	0	0	0	1	1	1	1
-2A	0	0	0	1	1	1	1	1



Primer: Radix-4 Booth-ov algoritem

A*X=(-6)*(-10)=60

X=(-10) = 1 1 0 1 1 0

(0)

-A 2A -2A

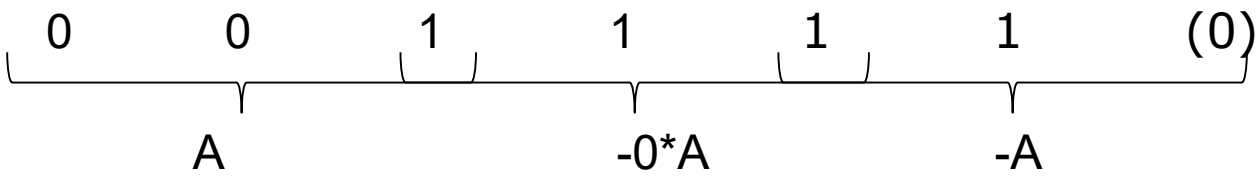
2A	1	1	0	1	0	0	0	0
A	1	1	1	0	1	0	0	0
-A	0	0	0	1	0	1	1	1
-2A	0	0	1	0	1	1	1	1

				1					
			1	1	1	0	1	1	
		1	0	0	1	0	0		1
1	1	0	1	0	1				
					1				
0	0	0	0	1	1	1	1	0	0

Primer: Radix-4 Booth-ov algoritem

A*X=(10)*(15)=
150

X=(15) =



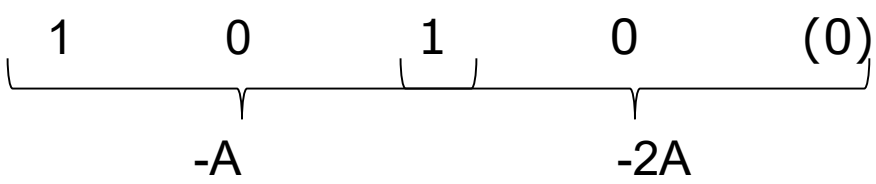
					1			
				1	0	0	1	0	1
		1	0	1	1	1	1		1
1	1	1	0	1	0		1		
0	0	1	0	0	1	0	1	1	0

2A	0	1	0	1	0	0	0	0
A	0	0	1	0	1	0	0	0
0A	0	0	0	0	0	0	0	0
-0A	1	1	1	1	1	1	1	1
-A	1	1	0	1	0	1	1	1
-2A	1	0	1	0	1	1	1	1

Primer: Radix-4 Booth-ov algoritem

A*X=(-6)*(-6)=36

X=(-6) =



			1					
		1	1	1	0	1	1	
1	1	0	1	0	1		1	
					1			
0	0	1	0	0	1	0	0	

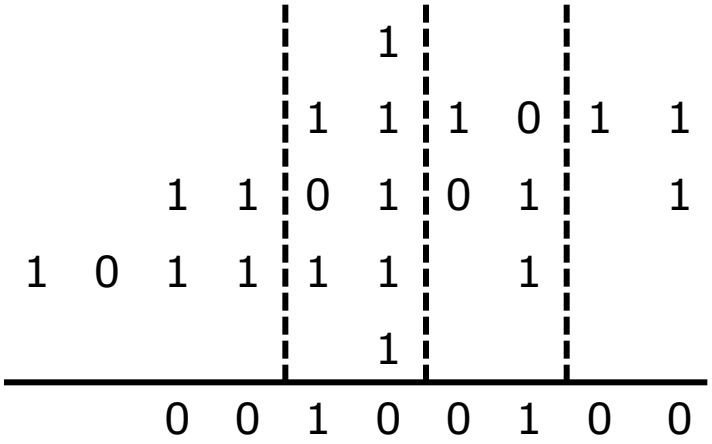
2A	1	1	0	1	0	0	0	0
A	1	1	1	0	1	0	0	0
-A	0	0	0	1	0	1	1	1
-2A	0	0	1	0	1	1	1	1

Primer: Radix-4 Booth-ov algoritem

A*X=(-6)*(-6)=36

X=(-6) = 1 1 1 0 1 0 (0)

-0A -A -2A



2A	1	1	0	1	0	0	0	0
A	1	1	1	0	1	0	0	0
0A	0	0	0	0	0	0	0	0
-0A	1	1	1	1	1	1	1	1
-A	0	0	0	1	0	1	1	1
-2A	0	0	1	0	1	1	1	1

Primer: Radix-4 Booth-ov algoritem

A*X=(-6)*(-31)=186

X=(-31) = 1 0 0 0 0 1 (0)

-2A 0A A

				1				
			1	0	1	0	1	0
		1	1	0	0	0	0	
1	1	1	0	1	1			
					1			
0	0	1	0	1	1	1	0	1	0

2A	1	1	0	1	0	0	0	0
A	1	1	1	0	1	0	0	0
0A	0	0	0	0	0	0	0	0
-0A	1	1	1	1	1	1	1	1
-A	0	0	0	1	0	1	1	1
-2A	0	0	1	0	1	1	1	1

Primer: Radix-4 Booth-ov algoritem

A*X=(-4)*(21)=-84

X=(21) = 0 1 0 1 0 1 (0)

A A A

					1		
				1	0	1	1	0	0
		1	0	1	1	0	0		
1	0	1	1	0	0				
1	1	1	0	1	0	1	1	0	0
0	0	0	1	0	1	0	1	0	0

2A	1	1	1	0	0	0	0	0
A	1	1	1	1	0	0	0	0
-A	0	0	0	0	1	1	1	1
-2A	0	0	0	1	1	1	1	1

Primer: Radix-4 Booth-ov algoritem

A*X=(4)*(-19)=-76

X=(-19)

(0) = 1 0 1 1 0 1

-A -A A

					1			
				1	1	0	1	0	0
		1	0	1	0	1	1		
1	0	1	0	1	1		1		
					1				
1	1	1	0	1	1	0	1	0	0
0	0	0	1	0	0	1	1	0	0

2A	0	0	1	0	0	0	0	0
A	0	0	0	1	0	0	0	0
-A	1	1	1	0	1	1	1	1
-2A	1	1	0	1	1	1	1	1

Primer: Radix-4 Booth-ov algoritem

A*X=(-6)*(13)=-78

X=(13) = 0 0 1 1 0 1 (0)

A -A A

					1				
				1	0	1	0	1	0
			1	1	0	1	0	1	
							1		
	1	0	1	0	1	0			
	1	1	1	0	1	1	0	0	1	0
	0	0	0	1	0	0	1	1	1	0

2A	1	1	0	1	0	0	0	0
A	1	1	1	0	1	0	0	0
-A	0	0	0	1	0	1	1	1
-2A	0	0	1	0	1	1	1	1

Pozor: Vedno je treba preveriti koliko biten je rezultat! Če ima rezultat (78) 6 bitov, ga bomo dosegli s 3 kratnim pomikanjem.

Primer: Radix-4 Booth-ov algoritem

A*X=(5)*(10)=50

X=(10) = 0 0 1 0 1 0 (0)

A -A -2A

2A	0	0	1	0	1	0	0	0
A	0	0	0	1	0	1	0	0
-A	1	1	1	0	1	0	1	1
-2A	1	1	0	1	0	1	1	1

					1				
				1	0	0	1	0	1
		1	0	1	0	1	0		1
1	1	0	1	0	1		1		
0	0	0	0	1	1	0	0	1	0

Primer: Radix-4 Booth-ov algoritem

- Isti množitelj v kodirani obliki: $010\bar{1}0\bar{1}$

9	8	7	6	5	4	3	2	1	0
		0	**1**	**1**	0	1	0	1	0
	1	**0**	0	1	0	1	0		
1	1	0	1	1	0				
0	0	1	1	1	1	0	0	1	0

9	8	7	6	5	4	3	2	1	0
		0	**1**	**1**	0	1	0	0	1
	1	**0**	0	1	0	0	1		**1**
1	1	0	1	1	0		**1**		
					0				
0	0	1	1	1	1	0	0	1	0

Množilnik v VHDL

```
-- Nepredznačeni 8x4-bitni množilnik
-- Download:
   ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip
-- File: HDL_Coding_Techniques/multipliers/multipliers_1.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity multipliers_1 is
generic (
   WIDTHA : integer := 8;
   WIDTHB : integer := 4);
port(
   A : in std_logic_vector(WIDTHA-1 downto 0);
   B : in std_logic_vector(WIDTHB-1 downto 0);
   RES : out std_logic_vector(WIDTHA+WIDTHB-1 downto 0));
end multipliers_1;
architecture a of multipliers_1 is
begin
   RES <= A * B;
end a;
```

Enota za seštevanje in množenje (ang. multiply-accumulate) MAC

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity mac_demo is
     generic (p_width: integer:=8);
     port ( clk, rst: in std_logic;
            A, B: in std_logic_vector(p_width - 1 downto 0);
            RES: out std_logic_vector(p_width * 2 - 1 downto 0));
end mac_demo ;
architecture a of mac_demo is
constant ZERO : unsigned(RES'range) := (others=>'0'); --nicla
signal mult, accum : unsigned(RES'range) := ZERO;
begin
process (clk)
begin
     if rising_edge(clk) then
            if (rst = '1') then
                        accum <= ZERO;
                        mult <= ZERO;
            else
                        accum <= accum + mult;
                        mult <= unsigned(A) * unsigned(B);
            end if;
     end if;
end process;
RES <= std_logic_vector(accum);
end a;
```

```
Macro Statistics
# MACs                : 1
 8x8-to-16-bit MAC  : 1
```

Enota za seštevanje in množenje (ang. multiply-accumulate) MAC

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity mac_demo_comb is
   generic (p_width: integer:=8);
   port ( A, B: in std_logic_vector(p_width - 1 downto 0);
          RES: out std_logic_vector(p_width * 2 - 1 downto 0));
end mac_demo_comb ;
architecture a2 of mac_demo_comb is
begin
   process (A, B)
   constant ZERO : unsigned(RES'range) := (others=>'0'); --nicla
   variable mult, accum : unsigned(RES'range) := ZERO;
   begin
        mult := unsigned(A) * unsigned(B);
        accum := accum + mult;
        RES <= std_logic_vector(accum);
   end process;
end a2;
```

```
Macro Statistics
# MACs                : 1
 8x8-to-16-bit MAC  : 1
```

Načrtovanje digitalnih vezij

Flip-flopi, registri in števci:
Registri

Kako določimo rob prehoda?

Funkciji rising_edge in falling_edge sta definirani v knjižnici std_logic_1164

```
  FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN IS
BEGIN
    RETURN (s'EVENT AND
            (To_X01(s) = '1') AND
            (To_X01(s'LAST_VALUE) = '0'));
END;
FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN IS
BEGIN
    RETURN (s'EVENT AND
            (To_X01(s) = '0') AND
            (To_X01(s'LAST_VALUE) = '1'));
END;
```

D-FF v VHDL

Primitivi D-FF v FPGA seriji 7:

- FDCE (Clock Enable, Asynchronous Clear)
- FDPE (Clock Enable, Asynchronous Preset)
- FDRE (Clock Enable, Synchronous Reset)
- FDPE (Clock Enable, Synchronous Set)

Asinhroni ali sinhroni SET/RESET?

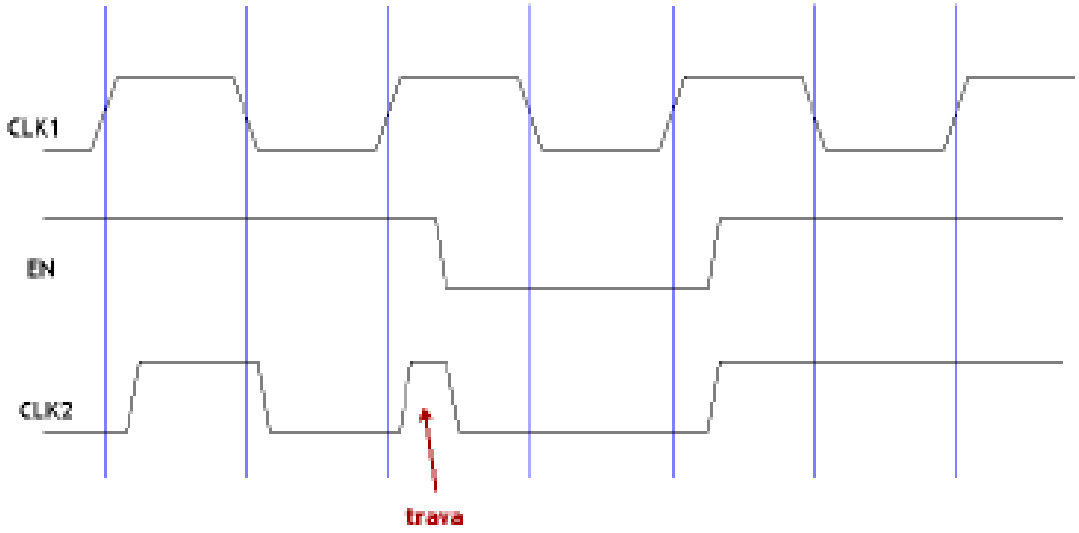
```
library IEEE;
use IEEE.std_logic_1164.all;
entity clock_gating is
    port (clk, -- signal ure
          rst, -- asinhroni reset
          en, -- postavi na '1' za omogocenje vpisa
         d_in: in STD_LOGIC;  -- vhodni podatek
          d_out: out STD_LOGIC -- izhod - rezultat vpisa
    );
end clock_gating;
architecture no_gating_using_fdcpe of clock_gating is
begin
process (clk, rst, set) is
begin
    if (rst = '1') then
          d_out <= '0';        -- asinhrona ponastavitev
    elsif (set = '1') then
          d_out <= '1';        -- asinhrona postavitev
    elsif rising_edge(clk) then
          if (en = '1') then   -- sinhrono omogocenje
                    d_out <= d_in; -- vpis ob prednjem robu signala ure
          end if;
    end if;
end process;
end no_gating_using_fdcpe;
```

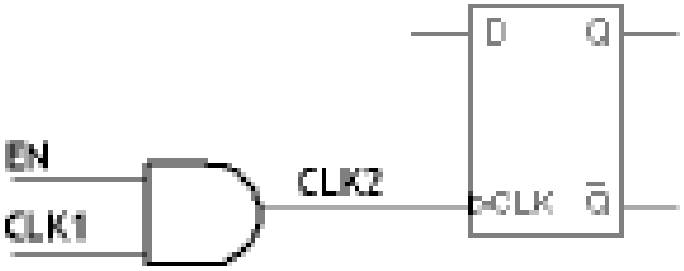
Asinhroni ali sinhroni SET/RESET?

- Naenkrat lahko uporabimo samo set ali samo reset – ne obeh. Sinteza sicer da FDCPE primitiv, vendar ta je počasnejši!
- Smiselno je uporabljati sinhrone SET/RESET vhode

Clock gating

- Trava v signalu ure (ang. glitches) nastane, ko želimo v vezju tvoriti/prepuščati signal ure ob določenem pogoju (ang. gating).
- To bi lahko storili npr. s kombinacijskim signalom (en).
- Takrat se pojavijo asinhrone spremembe na signalu ure - trava – ker krmilni signal ni sinhroniziran s signalom ure.

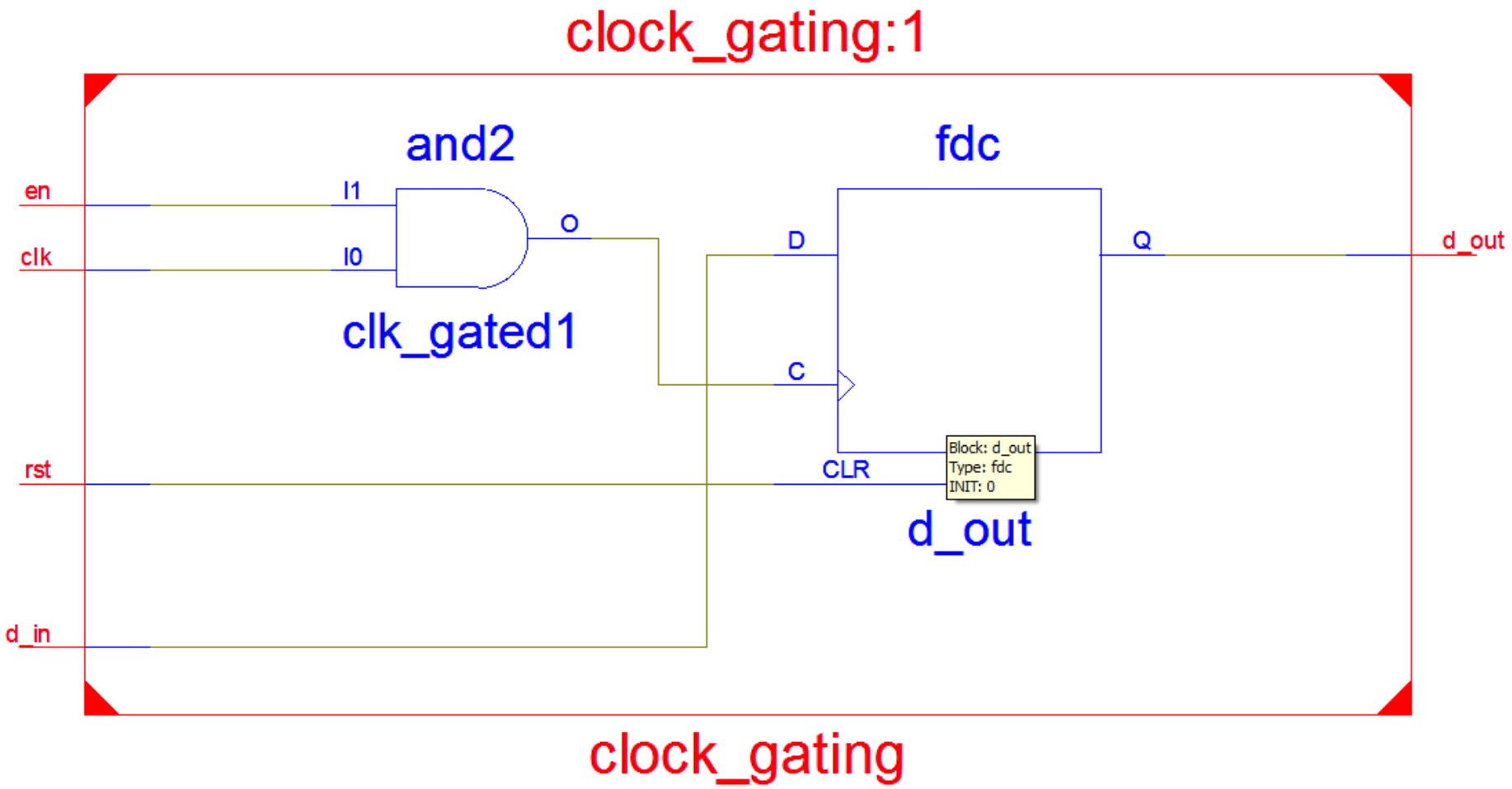




Clock gating – primer česa ne delamo

```
library IEEE;
use IEEE.std_logic_1164.all;
entity clock_gating is
    port (          clk,      -- signal ure
                    rst,      -- asinhroni reset
                    en,       -- postavi na '1' za omogocenje vpisa
                    d_in: in STD_LOGIC; -- vhodni podatek
                    d_out: out STD_LOGIC      -- izhod - rezultat vpisa
     );
end clock_gating;
architecture clk_gating_using_comb_signal of clock_gating is
signal clk_gated : STD_LOGIC;
begin
    clk_gated <= clk and en; -- kombinacijska funkcija, ki krmi uro
    process (clk_gated, rst) is
    begin
          if (rst = '1') then
                    d_out <= '0';     -- asinhrona ponastavitev
          elsif rising_edge(clk_gated) then
                    d_out <= d_in;    -- vpis ob prednjem robu signala ure
          end if;
    end process;
end clk_gating_using_comb_signal;
```

Clock gating – primer česa ne delamo



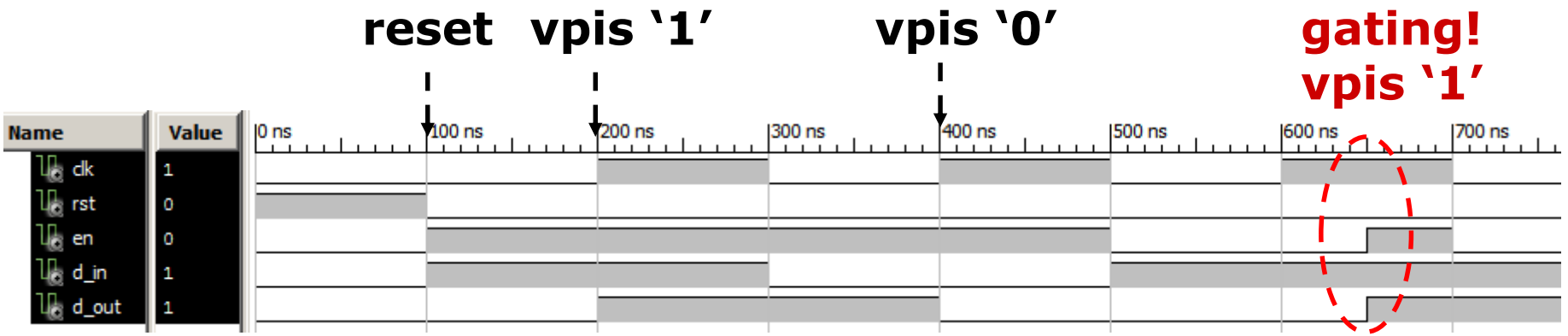
Datoteka testnih vrednosti

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use IEEE.numeric_std.all;
ENTITY clock_gating_tb IS
generic ( delay            : time := 5 ns            );
END clock_gating_tb;
ARCHITECTURE arch OF clock_gating_tb IS
COMPONENT clock_gating is
     port (clk, -- signal ure
      rst,    -- asinhroni reset
      en,     -- postavi na '1' za omogocenje vpisa
     d_in: in STD_LOGIC;   -- vhodni podatek
      d_out: out STD_LOGIC -- izhod - rezultat vpisa
    );
end COMPONENT;
signal clk : STD_LOGIC := '0';-- signal ure
signal rst : STD_LOGIC := '0';-- asinhroni reset
signal en : STD_LOGIC := '0'; -- postavi na '1' za
     omogocenje vpisa
signal d_in : STD_LOGIC := '0'; -- vhod
signal d_out : STD_LOGIC := '0'; -- izhod
constant clock_period : time := 200 ns;
constant clock_duty_cycle : real := 0.5;
constant clock_glitch_duty_cycle : real := 0.25;
constant clock_offset : time := 100 ns;
begin
process     -- proces, ki tvori signal ure clk
begin
     wait for clock_offset;
     clock_loop : loop
     clk <= '0';
     wait for (clock_period - (clock_period *
     clock_duty_cycle));
     clk <= '1';
     wait for (clock_period * clock_duty_cycle);
     end loop clock_loop;
end process;
```

```
     -- Povezovanje testne enote
    uut: clock_gating port map (
clk => clk, rst => rst, en => en, d_in => d_in, d_out
     => d_out);
     stim_proc: process
     begin
     rst <= '1';
     wait for clock_offset;
     rst <= '0';
     en <= '1'; -- omogoci vpis
     d_in <= '1'; -- podatek na '1'
     wait for clock_period;          -- pocakaj cikel
     signala ure - normalen vpis enice
     d_in <= '0'; -- podatek na '0'
     wait for clock_period;          -- pocakaj cikel
     signala ure - normalen vpis nicle
     en <= '0'; -- onemogoci vpis
     d_in <= '1';-- pripravi podatek na '1'
     wait for (clock_period *
     clock_glitch_duty_cycle);       -- pocakaj cetrt
     cikla signala ure
     wait for (clock_period * clock_duty_cycle); --
     pocakaj, da signal ure pride na '1'
     en <= '1'; -- omogoci vpis - prednji rob gated
     signala ure
     wait for (clock_period *
     clock_glitch_duty_cycle);       -- pocakaj cetrt
     cikla signala ure
     en <= '0'; -- onemogoci vpis - zadnji rob gated
     signala ure
     wait for (clock_period *
     clock_glitch_duty_cycle);       -- pocakaj cetrt
     cikla signala ure
     wait;
     end process;
end;
```

Rezultati simulacij - Clock gating

```
rst <= '1';
wait for clock_offset;
rst <= '0';
en <= '1'; -- omogoci vpis
d_in <= '1'; -- podatek na '1'
wait for clock_period;     -- pocakaj cikel signala ure - normalen vpis enice
d_in <= '0'; -- podatek na '0'
wait for clock_period;     -- pocakaj cikel signala ure - normalen vpis nicle
en <= '0'; -- onemogoci vpis
d_in <= '1';-- pripravi podatek na '1'
wait for (clock_period * clock_glitch_duty_cycle); -- pocakaj cetrt cikla signala ure
wait for (clock_period * clock_duty_cycle);    -- pocakaj, da signal ure pride na '1'
en <= '1'; -- omogoci vpis - prednji rob gated signala ure
wait for (clock_period * clock_glitch_duty_cycle); -- pocakaj cetrt cikla signala ure
en <= '0'; -- onemogoci vpis - zadnji rob gated signala ure
wait for (clock_period * clock_glitch_duty_cycle); -- pocakaj cetrt cikla signala ure
```



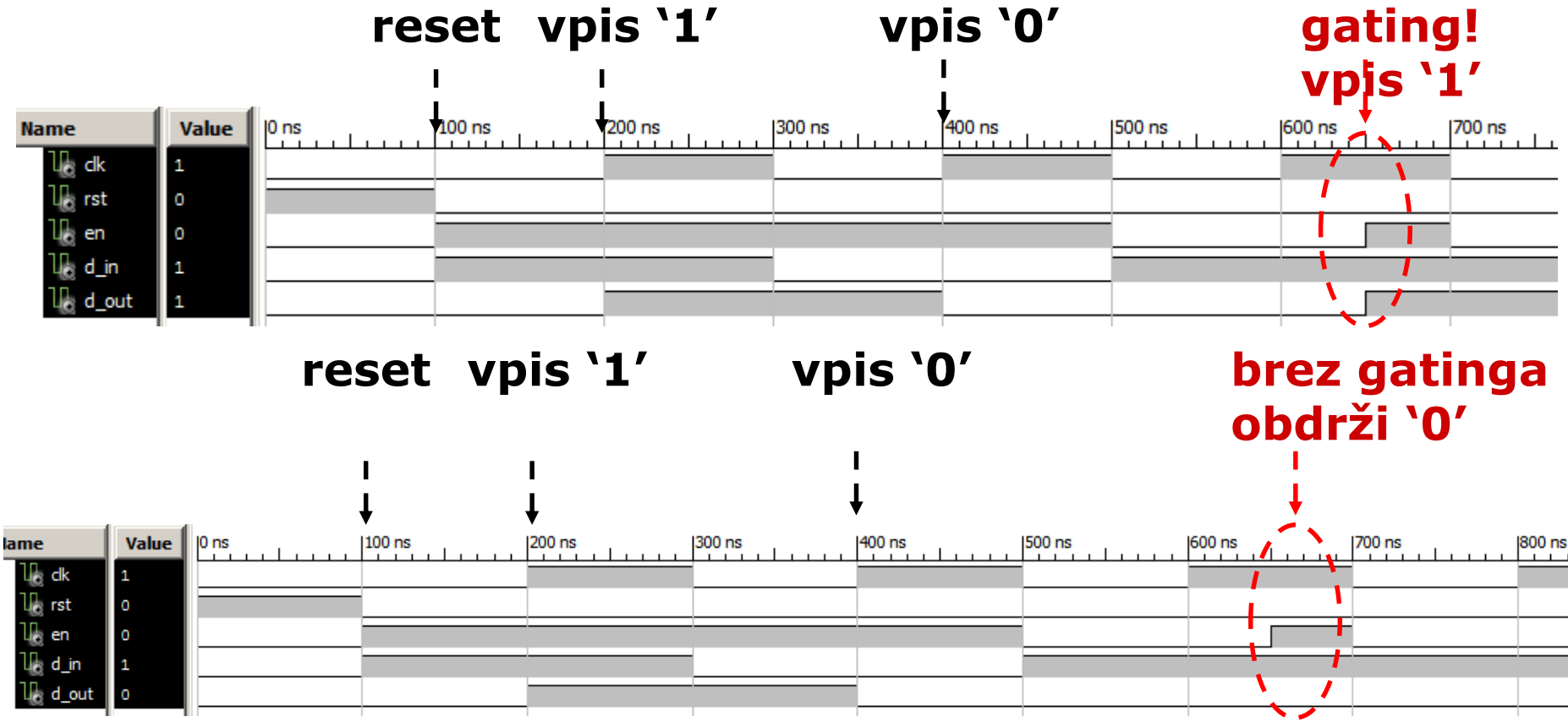
Kako pa potem?

```
library IEEE;
use IEEE.std_logic_1164.all;
entity clock_gating is
   port (clk,     -- signal ure
         rst,     -- asinhroni reset
         en,      -- postavi na '1' za omogocenje vpisa
        d_in: in STD_LOGIC;        -- vhodni podatek
         d_out: out STD_LOGIC -- izhod - rezultat vpisa
    );
end clock_gating;
architecture no_gating_using_ce of clock_gating is
begin
process (clk, rst) is
begin
   if (rst = '1') then
        d_out <= '0';     -- asinhrona ponastavitev
   elsif rising_edge(clk) then
      if (en = '1') then -- omogoči vpis s sinhronim signalom!
        d_out <= d_in; -- vpis ob prednjem robu signala ure
        end if;
   end if;
   end process;
end no_gating_using_ce;
```

Načelo snovanja sinhronskih vezij: Signal ure pride do vseh vezij kot tak – brez kakršnihkoli vrat na svoji poti! Omogočanje delovanja dosežemo s kombinacijsko funkcijo nad signalom omogočenja v FF!

V čem je pravzaprav razlika?

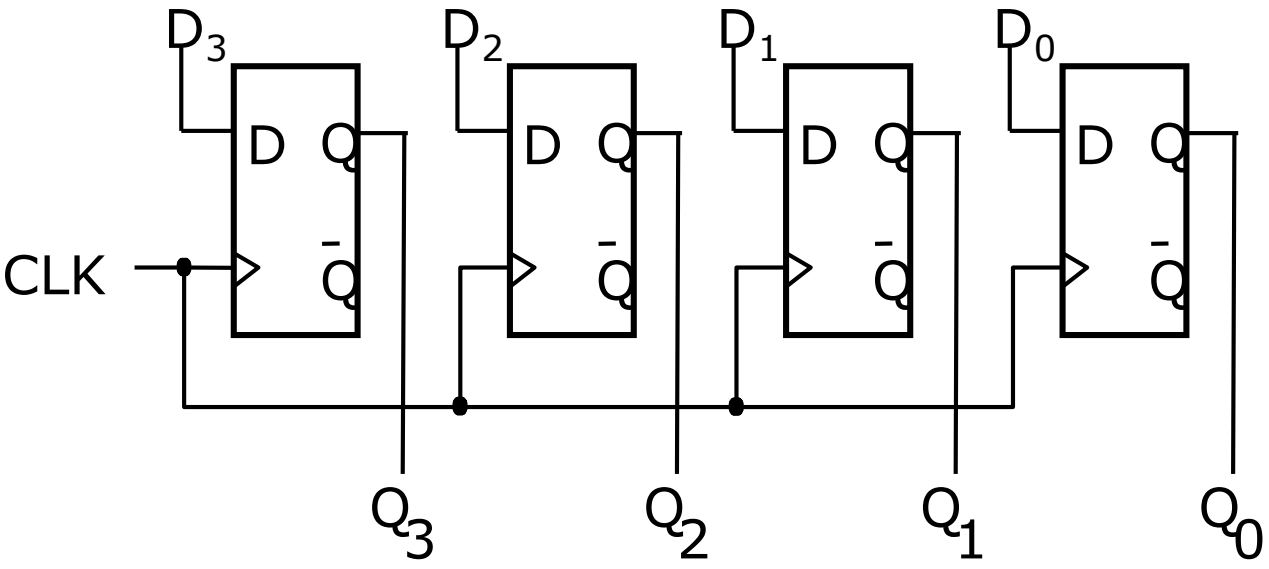
- Primerjava rezultatov iste datoteke testnih vrednosti:



Registri

- FF hrani 1 bit informacije
- ***Register*** je niz *n* FF, ki hrani *n* bitov podatkov
- Vrste registrov:
  - Shranjevalni
  - Pomikalni
  - Splošni oz. univerzalni
- Razdelitev glede na možnost vzporednega vpisa in dostopnost izhodov celic:
  - SISO (zaporedni vhod, zaporedni izhod),
  - SIPO (zaporedni vhod, vzporedni izhod)
  - PISO (vzporedni vhod, zaporedni izhod)
  - PIPO (vzporedni vhod, vzporedni izhod)

Shranjevalni register



Polje registrov (ang. register file)

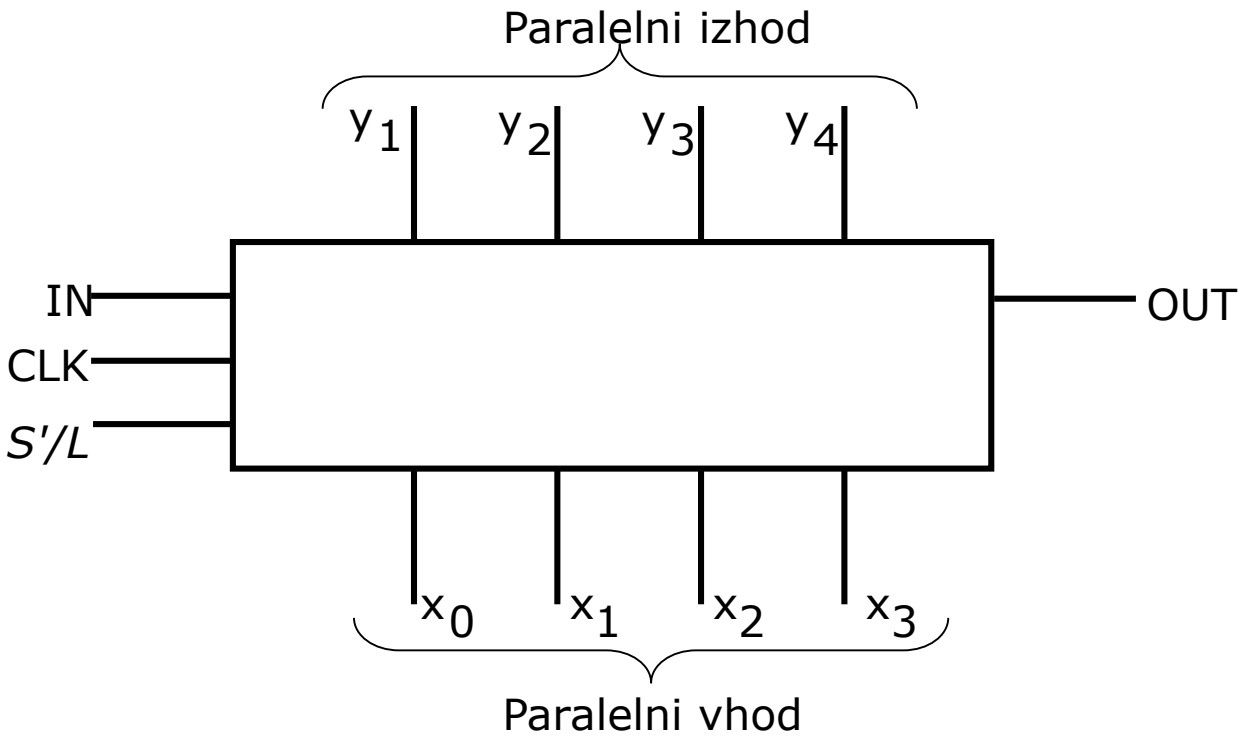
- Polje registrov je več registrov, ki imajo ena vhodna vrata in vsaj ena izhodna vrata.
- Signal za naslov vpisa (w_addr) določa kam se bo podatek shranil
- Signal za naslov branja (r_addr) določa register iz katerega se podatek bere.
- Polje registrov uporabljajo procesorji za hitro, začasno shranjevanje.
- Parametriziran 2^W-krat-B:
  - W ➔ število naslovnih bitov (2^W registrov v polju)
  - B ➔ število bitov v registru.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity reg_file is
generic (
B     : natural := 8;--sirina besede v polju registrov
W     : natural := 3);--stevilo registrov v polju (2**W)
port (
      r_data  : out std_logic_vector(B - 1 downto 0);
      w_data : in  std_logic_vector(B - 1 downto 0);
      w_addr, r_addr
      : in  std_logic_vector(W-1 downto 0);
      wr, clk  : in  std_logic );
end reg_file;
architecture arch of reg_file is
type array_reg_type is array(0 to 2**W - 1) of
      std_logic_vector(B - 1 downto 0);
signal array_reg : array_reg_type;
begin
process (clk) is
begin
if rising_edge(clk) then
      if wr = '1' then
            array_reg(to_integer(unsigned(w_addr)))
            <= w_data;
      end if;
end if;
end process;
r_data <= array_reg(to_integer(unsigned(r_addr)));
end arch;
```

Pomikalni register s paralelnim dostopom

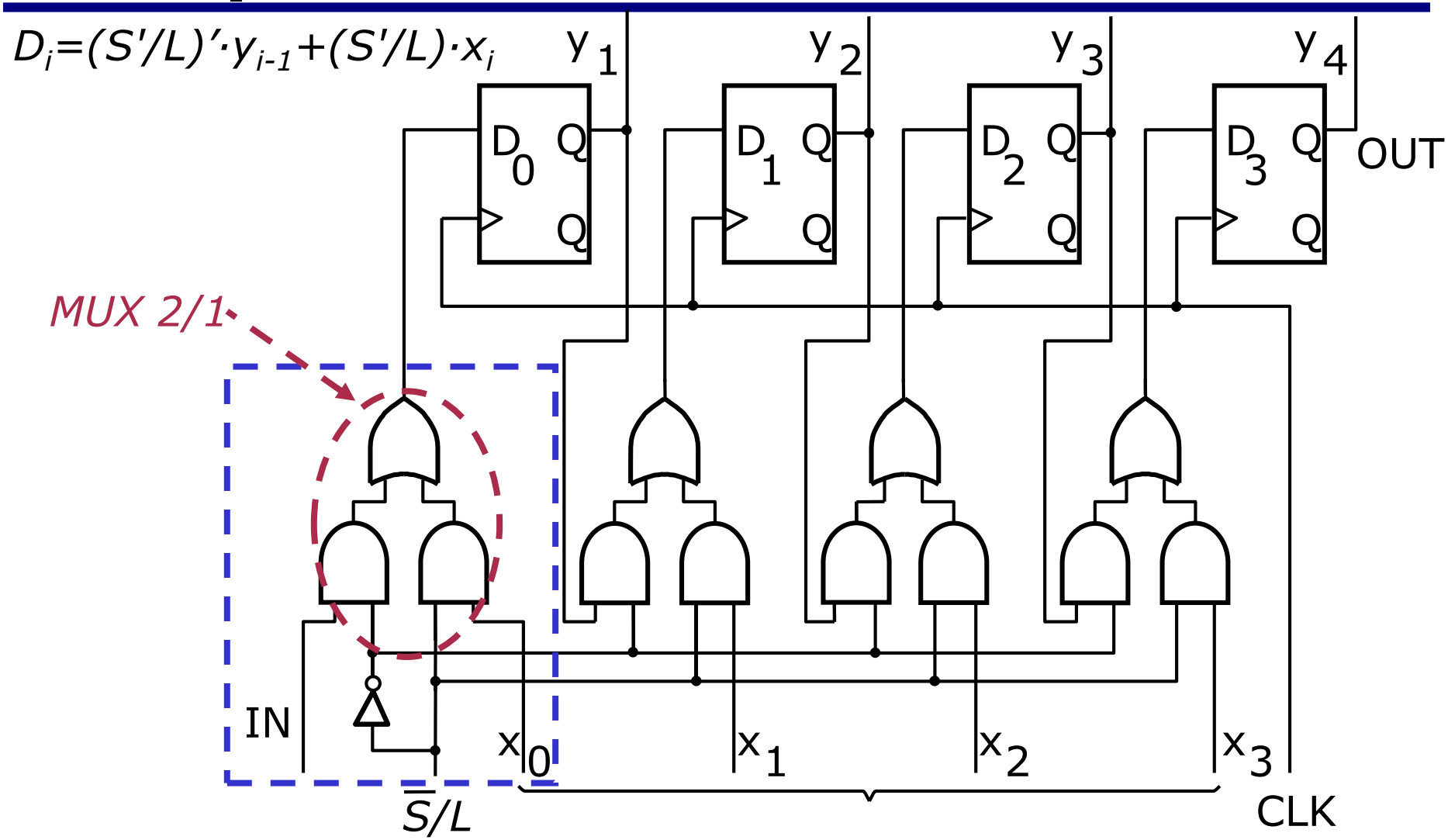
- Prenos *n*-bitov naenkrat ➔ vzporedni (***paralelni***) prenos
- Prenos *1*-bit naenkrat ➔ zaporedni (***serijski***) prenos
- Vzporedno-zaporedna pretvorba podatkov (PISO)
  - Podatke najprej naložimo v register vzporedno (v enem ciklu signala ure) in nato vsebino registra pomikamo zaporedno (sinhroni serijski oddajnik - SPI)
- Zaporedno-vzporedna pretvorba podatkov (SIPO)
  - Podatki so sprejeti zaporedno, potem po *n* ciklih signala ure lahko paralelno preberemo vsebino registra kot *n*-bitno informacijo (sinhroni serijski sprejemnik - SPI)

Pomikalni register s paralelnim dostopom



S'/L	$y_i(t)$	$y_i(t+1)$
0	0	$y_{i-1}(t)$
0	1	$y_{i-1}(t)$
1	0	$x_i(t)$
1	1	$x_i(t)$

Pomikalni register s paralelnim dostopom



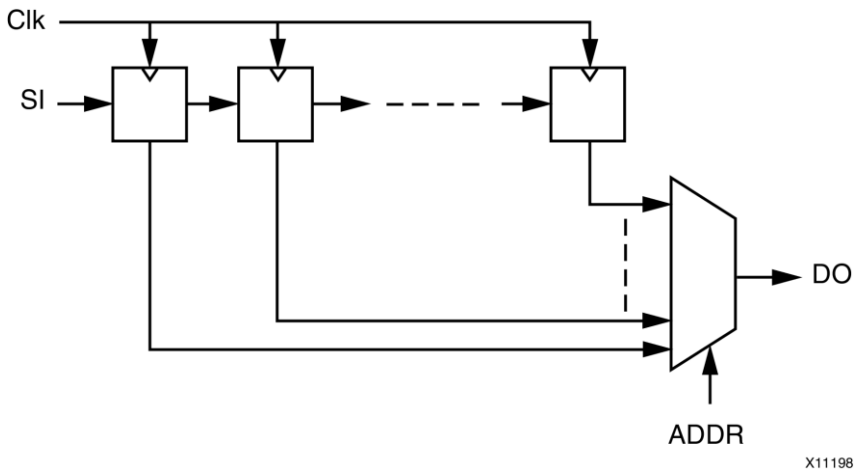
4-bitni pomikalni register v VHDL

```
ENTITY shift4 IS
PORT (R      : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
      Clock, L, w : IN STD_LOGIC;
      Q      : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END shift4;
ARCHITECTURE arch OF shift4 IS
BEGIN
   PROCESS
   BEGIN
    WAIT UNTIL Clock'EVENT AND Clock = '1';
     IF L = '1' THEN
      Q <= R;
     ELSE
      Q <= w & Q(3 DOWNTO 1); --operator sestavljanja
     END IF;
   END PROCESS;
END arch;
```

če je v procesnem stavku stavek **WAIT UNTIL** , potem procesni stavek nima seznama občutljivosti!

Dinamični pomikalni register v VHDL

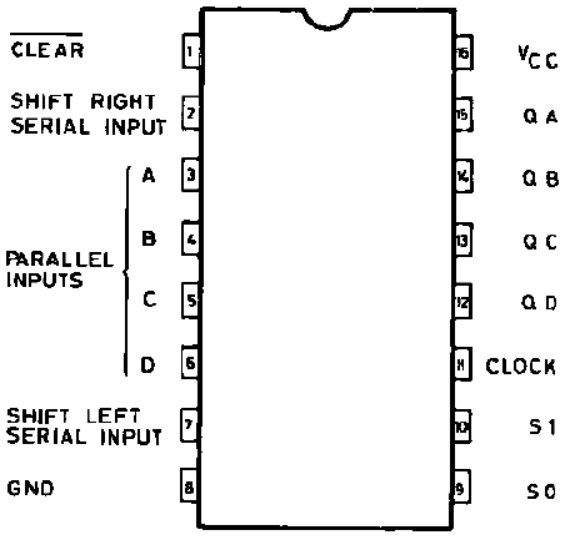
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity dyn_sreg is
generic (
     DEPTH : integer := 32;
     BUS_WIDTH : integer := 8;
     SEL_WIDTH : integer := 5
);
port(
     CLK,
     CE : in std_logic;
     SI : in std_logic_vector(BUS_WIDTH - 1 downto 0);
     A : in std_logic_vector(SEL_WIDTH - 1 downto 0);
     DO : out std_logic_vector(BUS_WIDTH - 1 downto 0)
);
end dyn_sreg;
architecture a1 of dyn_sreg is
type dyn_sr_array is array (0 to DEPTH-1) of std_logic_vector(BUS_WIDTH - 1 downto 0);
signal dyn_sr : dyn_sr_array := (others => (others => '0'));
begin
process (CLK)
begin
     if rising_edge(CLK) then
          if CE = '1' then
               dyn_sr <= SI & dyn_sr(0 to DEPTH - 2); -- pomik mesto desno
          end if;
     end if;
end process;
DO <= dyn_sr(to_integer(unsigned(A)));
end a1;
```




```
Advanced HDL Synthesis Report
Macro Statistics
# Shift Registers                 : 8
 32-bit dynamic shift register    : 8
```

Univerzalni register MSI 74194

PRIKLJUČEK	FUNKCIJA
CLEAR	Asinhrono brisanje (negativna logika)
SR	Zaporedni vhod (ob pomikanju desno)
A,B,C,D	Vzporedni vhodi
SL	Zaporedni vhod (ob pomikanju levo)
S0, S1	vhoda za določanje načina delovanja
CLOCK	Clock Input (negative edge triggered)
QA,QB,QC,QD	Vzporedni izhodi



Delovanje 74194

VHODI										IZHODI			
CLEAR	NAČIN		CLOCK	ZAPOREDNI		VZPOREDNI				QA	QB	QC	QD
	S1	S0		LEVO	DESNO	A	B	C	D				
L	X	X	X	X	X	X	X	X	X	L	L	L	L
H	X	X	↓	X	X	X	X	X	X	QA0	QB0	QC0	QD0
H	H	H	↑	X	X	a	b	c	d	a	b	c	d
H	L	H	↑	X	H	X	X	X	X	H	QAn	QBn	QCn
H	L	H	↑	X	L	X	X	X	X	L	QAn	QBn	QCn
H	H	L	↑	H	X	X	X	X	X	QBn	QCn	QDn	H
H	H	L	↑	L	X	X	X	X	X	QBn	QCn	QDn	L
H	L	L	X	X	X	X	X	X	X	QA0	QB0	QC0	QD0

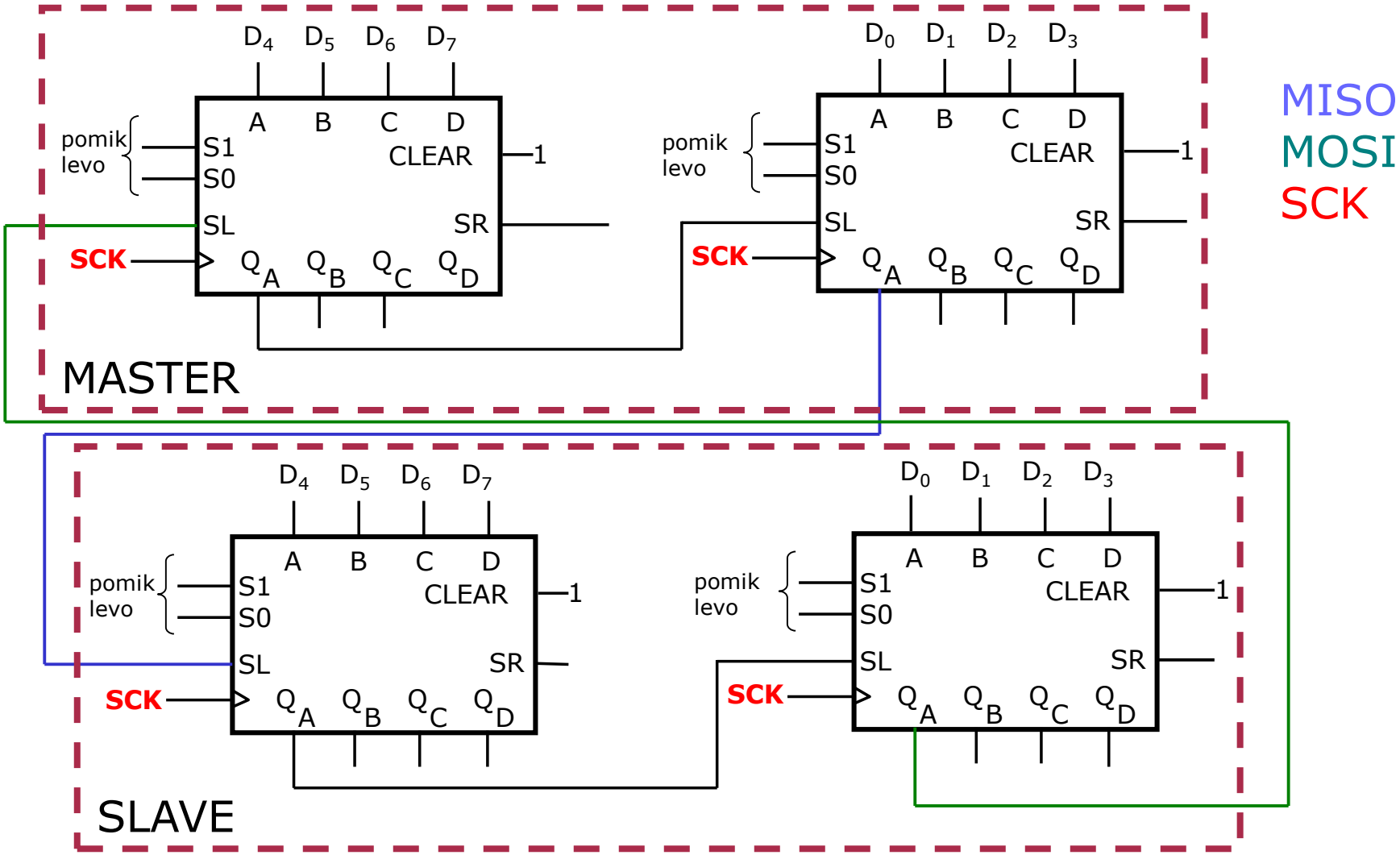
↓ - zadnji rob signala CLOCK
↑ - prednji rob signala CLOCK
4 operacije:

- S0='H' S1='H':Vzporedno nalaganje (a,b,c,d)➔ (Qa, Qb, Qc, Qd)
- S0='H' S1='L':Pomikanje levo (v smeri Qd➔Qa)
- S0='L' S1='H':Pomikanje desno (v smeri Qa➔Qd)
- S0='L' S1='L':Držanje vsebine

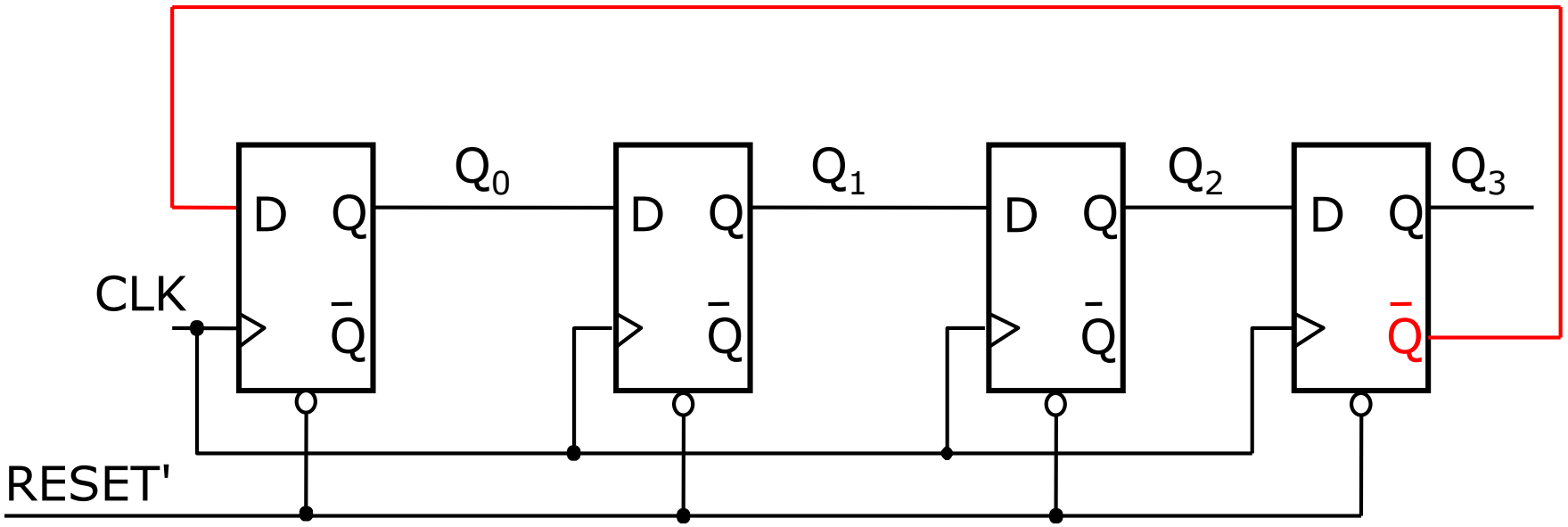
Uporaba pomikalnih registrov

- Vzporedno/zaporedna pretvorba in obratno
- Števci na osnovi pomikalnih registrov
- FIFO in LIFO pomnilnik
- LFSR (Linear Feedback Shift Register)
  - psevdonaključni generator
  - tvorba CRC (ostanek v XOR deljenju)

Vzporedno/zaporedna pretvorba – SPI protokol



Drugi tipi števcev – Johnsonov števec



Sekvenca štetja:
0000➔1000➔1100➔1110➔1111➔0111➔0011➔0001➔0000

Lastnosti Johnson-ovega števca

Dobre lastnosti:

- vezja za detekcijo stanja ne izkazujejo hazarda
- detekcija stanja pri poljubnem modulu štetja je možna na osnovi opazovanja samo dveh mest v registru,
- izhodi posameznih pomnilnih celic v registru so fazno pomaknjeni z *m* deljeni poteki urinega signala.

Slabe lastnosti:

- neracionalna izraba stanj za večje module štetja
- če se zaradi napake pojavi vrednost, ki ni v števni sekvenci, je potrebno izvršiti korekcijo z dodatno logiko.

Prehod v osnovno števno sekvenco (n=3)

Kaj če se v registru Johnson-ovega števca pojavi vrednost, ki ni del števne sekvence?

n=3

trenutno			naslednje		
Q_0	**Q_1**	**Q_2**	**Q_0**	**Q_1**	**Q_2**
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0			
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1			
1	1	0	1	1	1
1	1	1	0	1	1

$Q_0(t+1)$:

Q_1 \ Q_0				
	1	0	0	0
	1	0	0	1

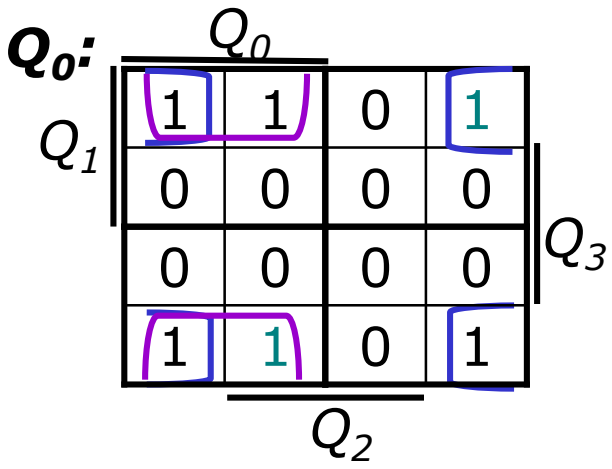
Q_2

$$Q_0(t+1)=Q_1'\cdot Q_2'+Q_0\cdot Q_2'$$

$$Q_0(t+1)=(Q_1'+Q_0)\cdot Q_2'$$

Prehod v osnovno števno sekvenco (n=4)

trenutno				naslednje			
Q_0	Q_1	Q_2	Q_3	Q_0	Q_1	Q_2	Q_3
0	0	0	0	**1**	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	1	0	0	1
0	0	1	1	0	0	0	1
0	1	0	0	**1**	0	1	0
0	1	0	1	0	0	1	0
0	1	1	0	1	0	1	1
0	1	1	1	0	0	1	1
1	0	0	0	**1**	1	0	0
1	0	0	1	0	1	0	0
1	0	1	0	**1**	1	0	1
1	0	1	1	0	1	0	1
1	1	0	0	**1**	1	1	0
1	1	0	1	0	1	1	0
1	1	1	0	**1**	1	1	1
1	1	1	1	0	1	1	1



Ti dve enici lahko vključimo v realizacijo korekturnega vezja, saj se števno zaporedje iz teh dveh stanj prej ko slej izide v osnovno števno zaporedje.

$$Q_0(t+1)=Q_3'\cdot Q_2'+Q_0\cdot Q_3'$$

$$Q_0(t+1)=(Q_2'+Q_0)\cdot Q_3'$$

Realizacija lihih modulov štetja (n=3)

Sodi modul štetja (0,4,6,7,3,1)

n=3

trenutno			naslednje		
Q_0	Q_1	Q_2	Q_0	Q_1	Q_2
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0			
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1			
1	1	0	1	1	1
1	1	1	0	1	1

opazujemo zadnje $Q_n(t)$ in predzadnje mesto $Q_{n-1}(t)$

trenutno		naslednje
Q_1	Q_2	Q_0
0	0	1
0	1	0
1	0	0
1	1	0

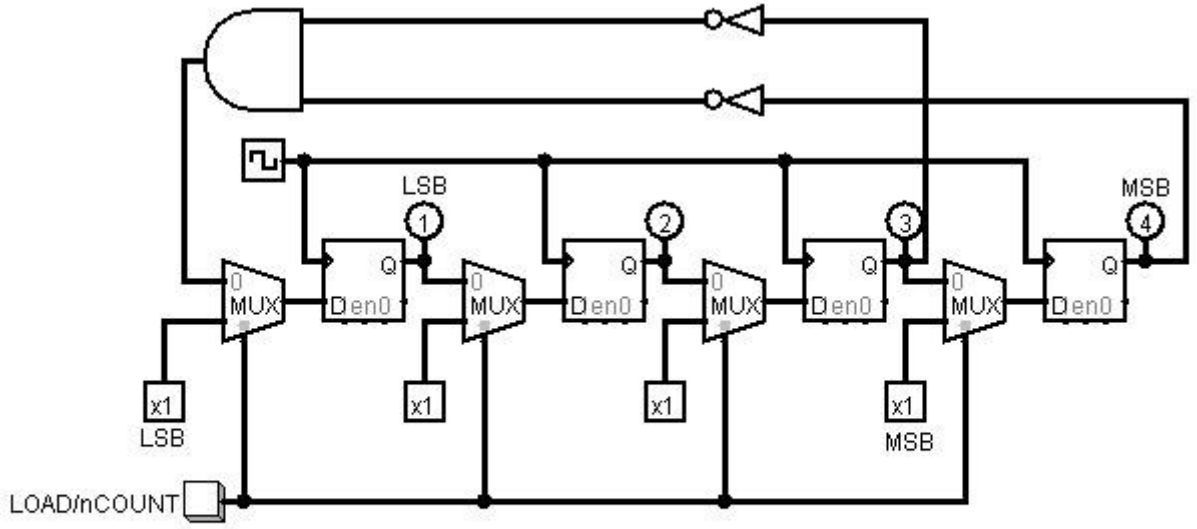
Lihi modul štetja (0,4,2,1,0)

n=3

trenutno			naslednje		
Q_0	Q_1	Q_2	Q_0	Q_1	Q_2
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	0	0	1
0	1	1			
1	0	0	0	1	0
1	0	1			
1	1	0			
1	1	1			

Lihi modul štetja ➔ izločimo eno stanje

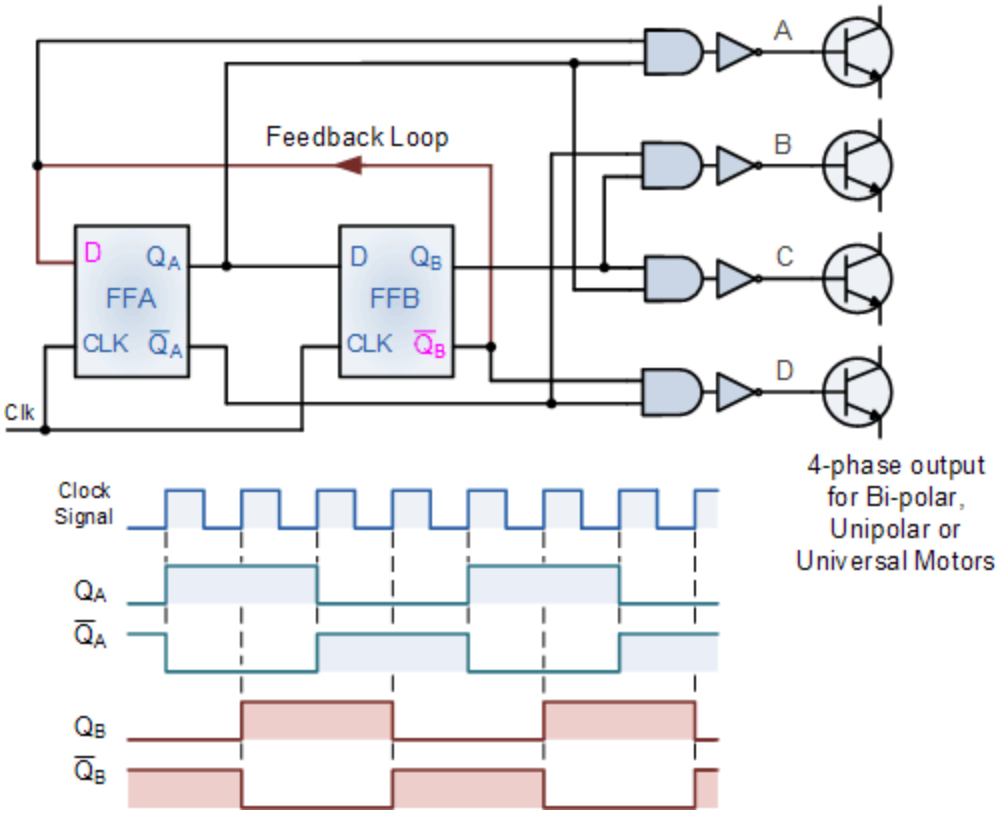
Realizacija lihih modulov štetja (n=4)



Module 7 Johnson counter: 1000;1100;1110;0111;0011;0001;0000

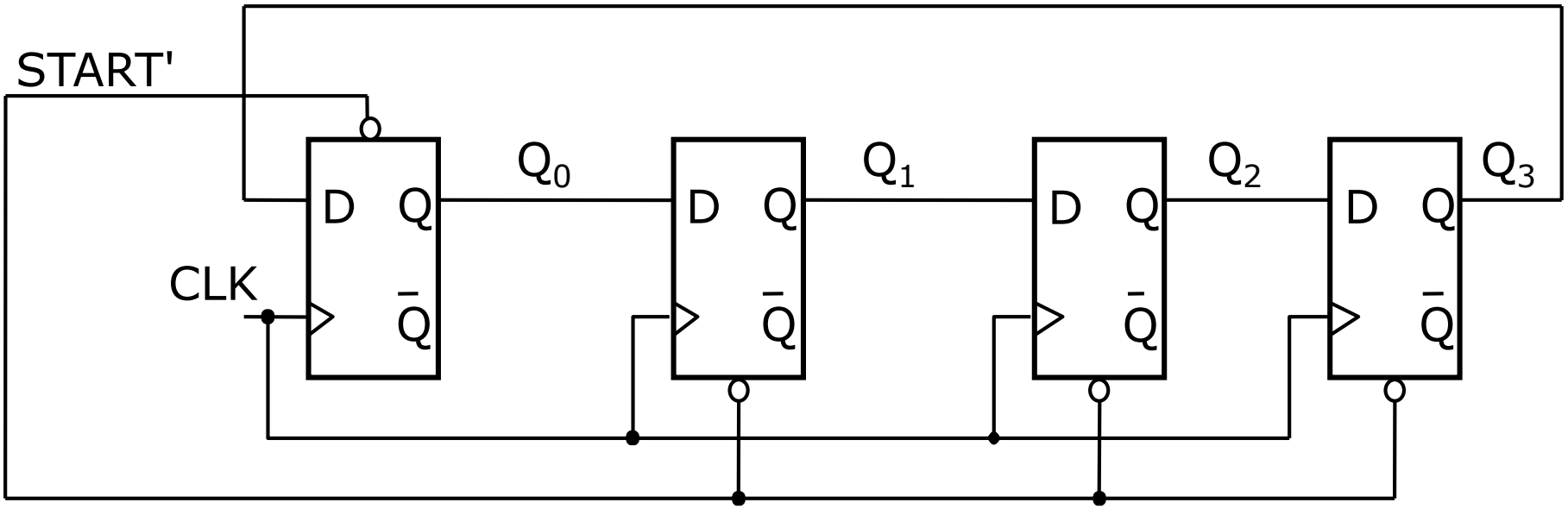
Odd number of states also guarantees return to original count sequence! Example (0101)

Johnsonov števec kot kvadraturni generator



IZHOD	A	B	C	D
Q_A+Q_B'	1	0	0	0
$Q_A'+Q_B$	0	1	0	0
Q_A+Q_B	0	0	1	0
$Q_A'+Q_B'$	0	0	0	1

Krožni števec (ang. ring counter)



Sekvenca štetja: 1000➔0100➔0010➔0001
To je kodiranje stanj "ena naenkrat"
(v knjigi "1-od-N") (ang. one hot encoding)

Pomiki in rotacije v VHDL

Pomike in rotacije lahko v VHDL izvajamo na dva načina:

- *Z operatorjem sestavljanja* (ang. concatenation):

```
tmp <= tmp(14 downto 0) & '0';
```

- *Z uporabo vgrajenih funkcij*
  (Tu velja opozoriti na različne nize funkcij ter operatorjev, ki jih srečamo na spletu).

Vgrajene funkcije pomika v VHDL

```
function shift_left  (  ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
-- Operacija :   pomik levo nad nepredznačenim vektorjem COUNT krat.
--               Izpraznjena mesta se dopolnijo z bitom '0'.
--               COUNT bitov z leve se izgubi

function shift_right (  ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;
-- Operacija :   pomik desno nad nepredznačenim vektorjem COUNT krat.
--               Izpraznjena mesta se dopolnijo z bitom '0'.
--               COUNT bitov z desne se izgubi.

function shift_left  (  ARG: SIGNED; COUNT: NATURAL) return SIGNED;
-- Operacija :   pomik levo nad predznačenim vektorjem COUNT krat.
--               Vsa mesta ARG se pomaknejo za COUNT mest levo.
--               Izpraznjena mesta se dopolnijo z bitom '0'.
--               COUNT bitov z leve se izgubi

function shift_right (  ARG: SIGNED; COUNT: NATURAL) return SIGNED;
-- Operacija:    pomik desno nad predznačenim vektorjem COUNT krat.
--               Izpraznjena mesta se dopolnijo z mestom ARG'LEFT.
--               COUNT bitov z desne se izgubi.
```

Vgrajene funkcije rotacije v VHDL

```
function rotate_left  (  ARG: UNSIGNED; COUNT: NATURAL) return
   UNSIGNED;
-- Operacija:          rotacija levo nad nepredznačenim
                       vektorjem COUNT krat
function rotate_right (  ARG: UNSIGNED; COUNT: NATURAL) return
   UNSIGNED;
-- Operacija:          rotacija desno nad nepredznačenim
                       vektorjem COUNT krat
function rotate_left  (  ARG: SIGNED; COUNT: NATURAL) return SIGNED;
-- Operacija:          rotacija levo nad predznačenim
                       vektorjem COUNT krat
function rotate_right (  ARG: SIGNED; COUNT: NATURAL) return SIGNED;
-- Operacija:          rotacija desno nad predznačenim
                       vektorjem COUNT krat
```

Nestandardni operatorji rotacije in pomika

- Operatorji sll, sla, srl, sra, rol, ror nikdar niso bili definirani v celoti in so jih končno umaknili iz standarda IEEE 1076.
- Definicije teh operatorjev vsebujejo drugi operand kot *predznačeno* število, zato omogočajo tudi negativne pomike.
- Negativni pomiki pri vgrajenih funkcijah niso možni, (možen je le pomik/rotacija za naravno število mest)
- Vgrajena funkcija shift_left() za predznačena števila opravlja funkcijo aritmetičnega pomika levo.
- Če pa za pomik uporabite sll oz. srl operatorja z negativnimi vrednostmi pomika, pride do zmede, saj se bi pričakovali, da se aritmetični pomik levo obnaša kot aritmetični pomik desno – pa se ne!

Nestandardni operatorji rotacije in pomika

Če je original:

```
A = "10010101"    -- kot je definiran v IEEE.NUMERIC_STD
```

Logična pomika:

```
A sll 2 = "01010100"-- logični pomik levo,
                    -- dopolni z leve z '0'
A srl 3 = "00010010"-- logični pomik desno,
                    -- dopolni z desne z '0'
```

Aritmetična pomika:

```
A sla 3 = "10101111"-- aritmetični pomik levo,
                    -- dopolni z leve z MSB (bitom predznaka)
A sra 2 = "11100101"-- aritmetični pomik desno,
                    -- dopolni z desne z LSB
```

Rotaciji:

```
A rol 3 = "10101100"-- rotacija levo za 3 mesta
A ror 5 = "10101100"-- rotacija desno za 5 mest
```

Število mest je lahko tudi negativno: A srl -2 je ista operacija kot A sll 2

Prikaz delovanja standardnih funkcij pomika in rotacije

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use IEEE.numeric_std.all;
use work.shift_rotate_pkg.all;
ENTITY what_is_a_shifter_tb IS
generic (     delay : time := 5 ns;
              size : integer := 17);
END what_is_a_shifter_tb;

ARCHITECTURE arch OF what_is_a_shifter_tb IS
     COMPONENT signed_shifter is
     generic (
              size : natural := 8
     );
     port(    x : in signed(size - 1 downto 0);
              mode : in unsigned(1 downto 0); --4 operacije: 00 lsl, 01 lsr, 10 asl, 11 asr
              s : in unsigned(sizeof(size - 1) - 1 downto 0); -- stevilo bitov pomika
              y : out signed(size - 1 downto 0) -- rezultat pomika/rotacije kot predznaceno stevilo
     );
     end COMPONENT;
     COMPONENT unsigned_shifter is
     generic (
              size : natural := 8
              );
     port(    x : in unsigned(size - 1 downto 0);
              mode : in unsigned(1 downto 0); --4 operacije: 00 lsl/asl, 01 lsr/asr, 10 rol, 11 ror
              s : in unsigned(sizeof(size - 1) - 1 downto 0); -- stevilo bitov pomika
              y : out unsigned(size - 1 downto 0) -- rezultat pomika/rotacije kot nepredznaceno stevilo
              );
     end COMPONENT;
     constant ONE : unsigned(size - 1 downto 0) := to_unsigned(1, size);         -- ena
     constant MINUS_ONE : signed(size - 1 downto 0) := to_signed(-1, size);      --minus ena
     signal mode : unsigned(1 downto 0) := to_unsigned(0, 2); --4 operacije: 00 lsl/asl, 01 lsr/asr, 10 rol, 11 ror
     signal s : unsigned(sizeof(size - 1) - 1 downto 0) := to_unsigned(0, sizeof(size - 1));  -- stevilo bitov pomika
     signal y_unsigned : unsigned(size - 1 downto 0); -- rezultat pomika/rotacije kot nepredznaceno stevilo
     signal y_signed : signed(size - 1 downto 0); -- rezultat pomika/rotacije kot predznaceno stevilo
```

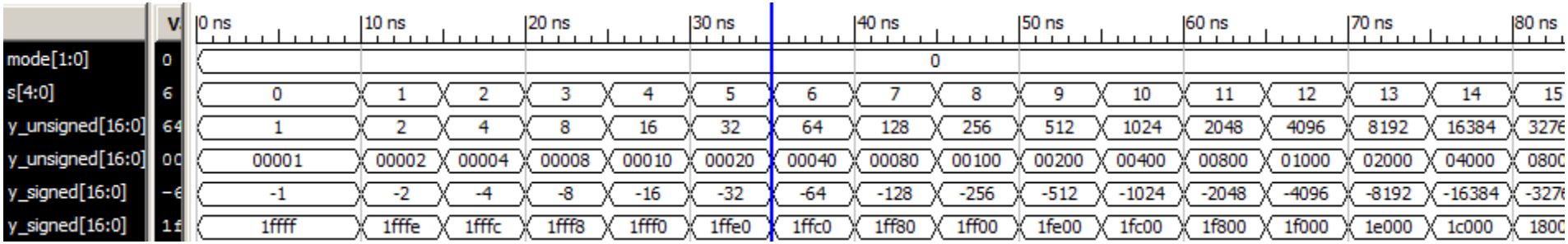
Prikaz delovanja standardnih funkcij pomika in rotacije

```
begin
-- Povezovanje testnih enot Units Under Test (UUTs)
uut_unsigned: unsigned_shifter generic map (size =>
    size) PORT MAP (  x => ONE, mode => mode,
    s => s, y => y_unsigned);
uut_signed: signed_shifter generic map (size => size)
    PORT MAP (  x => MINUS_ONE, mode => mode, s => s,
    y => y_signed);
stim_proc: process
begin
for mode_nr in 0 to 2**mode'length - 1 loop
    mode <= to_unsigned(mode_nr, mode'length);
    wait for delay;
    for shift_nr in  0 to size - 1 loop
            s <= to_unsigned(shift_nr, s'length);
            -- nastavi stevilo pomikov
            wait for delay;
    shift_rotate_case: case mode is
    when "00" =>
            report "lsl(" &
    integer'image(to_integer(ONE))  & ", " &
    integer'image(shift_nr) & ") = " &
            integer'image(to_integer(y_unsigned)) &
            " asl(" &
    integer'image(to_integer(MINUS_ONE))        & ", "
    & integer'image(shift_nr) & ") = " &
            integer'image(to_integer(y_signed));
    when "01" =>
            report "lsr(" &
    integer'image(to_integer(ONE))  & ", " &
    integer'image(shift_nr) & ") = " &
    integer'image(to_integer(y_unsigned)) &
    " asr(" & integer'image(to_integer(MINUS_ONE)) &
    ", " & integer'image(shift_nr) & ") = "&
    integer'image(to_integer(y_signed));
```

```
    when "10" =>
        report "rol(" &
        integer'image(to_integer(ONE)) & ", " &
        integer'image(shift_nr) & ") = " &
        integer'image(to_integer(y_unsigned)) &
        " signed rol(" &
        integer'image(to_integer(MINUS_ONE)) & ",
        " & integer'image(shift_nr) & ") = " &
        integer'image(to_integer(y_signed));
    when "11" =>
        report "ror(" &
        integer'image(to_integer(ONE)) & ", " &
        integer'image(shift_nr) & ") = " &
        integer'image(to_integer(y_unsigned)) &
        " signed ror(" &
        integer'image(to_integer(MINUS_ONE)) & ",
        " & integer'image(shift_nr) & ") = "&
        integer'image(to_integer(y_signed));
    when others =>
        assert false report "Operacija:
        neznana!";
    end case shift_rotate_case;
    end loop;
    end loop;
    wait;
    end process;
    end;
```

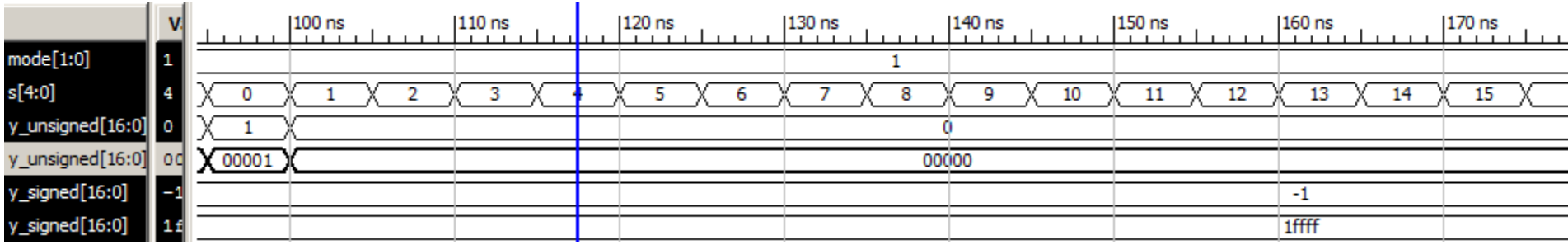
Prikaz delovanja standardnih funkcij pomika

pomik 17 bitnega števila 1 za nepredznačena števila in števila -1 za predznačena števila



pomik levo (LSL, ASL)

→ za nepredznačene se na prazno mesto vpiše '0'
→ za predznačene se prazna mesta dopolnijo z '0'

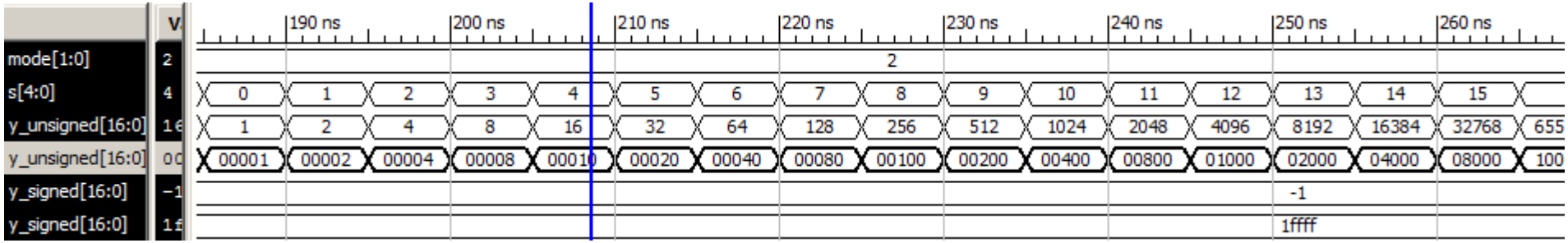


pomik desno (LSR, ASR)

→ za nepredznačene se na prazno mesto vpiše '0'
→ za predznačene se prazna mesta dopolnijo z MSB (-1 so same '1')

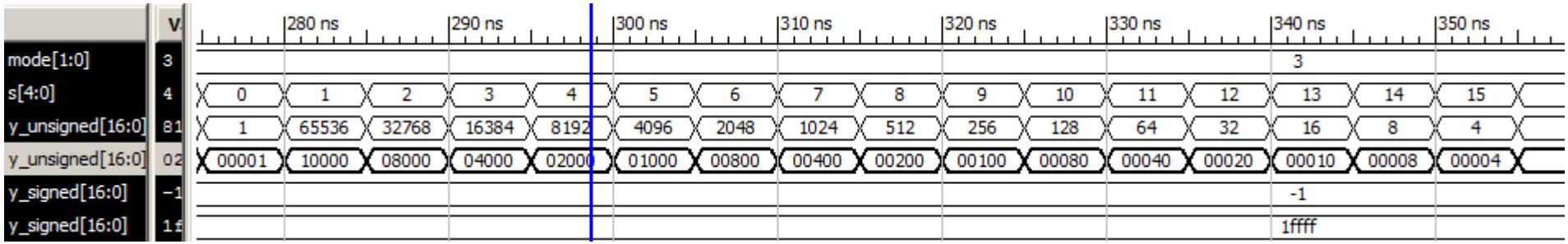
Prikaz delovanja standardnih funkcij rotacije

pomik 17 bitnega števila 1 za nepredznačena števila in števila -1 za predznačena števila



rotacija levo (ROL)

za predznačene ni kaj rotirati (-1 so same '1')



rotacija desno (ROR)

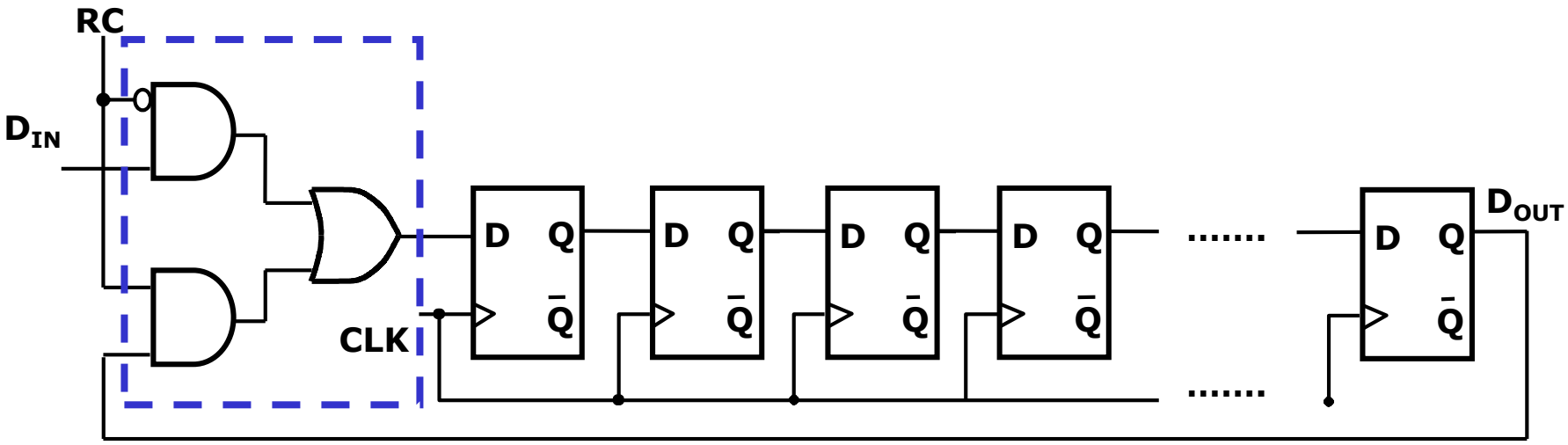
za predznačene ni kaj rotirati (-1 so same '1')

Načrtovanje digitalnih vezij

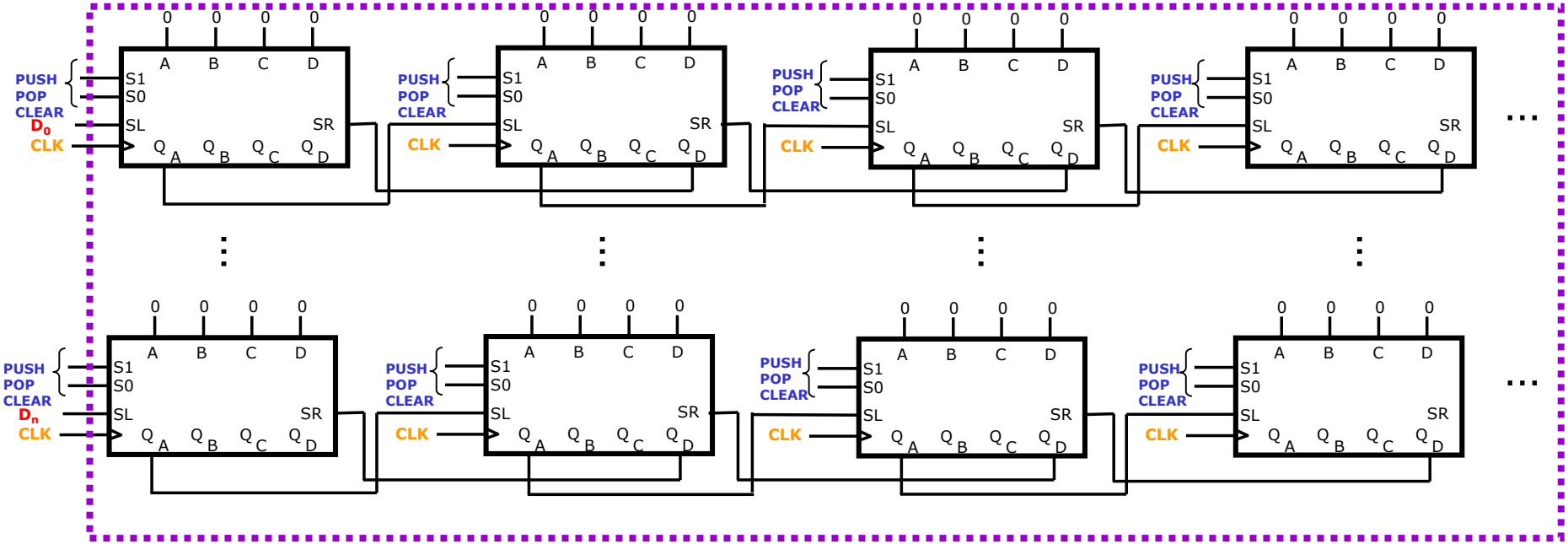
Uporaba pomikalnih registrov:
FIFO, LIFO

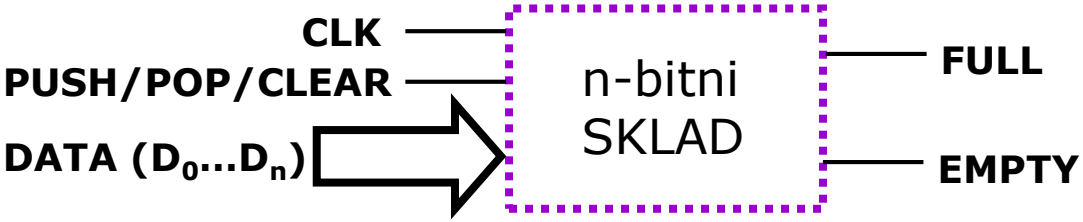
Enosmerni ciklični pomikalni registri

- Uporaba kot FIFO pomnilniki (pomnilnik z zaporednim dostopom).
- Običajne dolžine so 32, 64, 128, 256, 512, 1024.
- Krmilni signal RC krmili MUX 2/1:
  - RC=0 ➔ zaporedni vpis podatka z vhoda D_{IN}
  - RC=1 ➔ register je vezan kot cikličen pomikalni register oz. kot krožni števnik.



Dvosmerni ciklični pomikalni registri



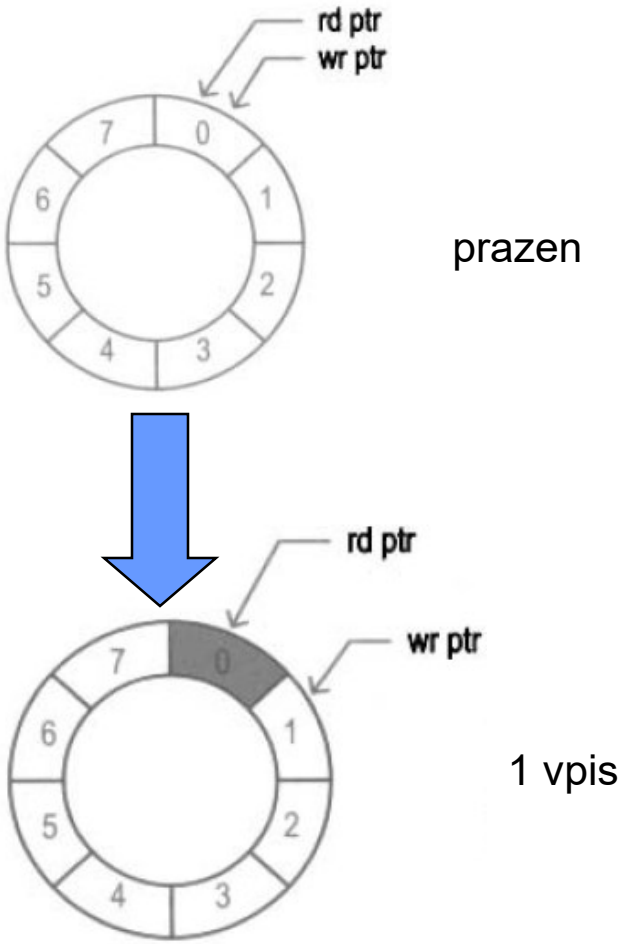


FIFO

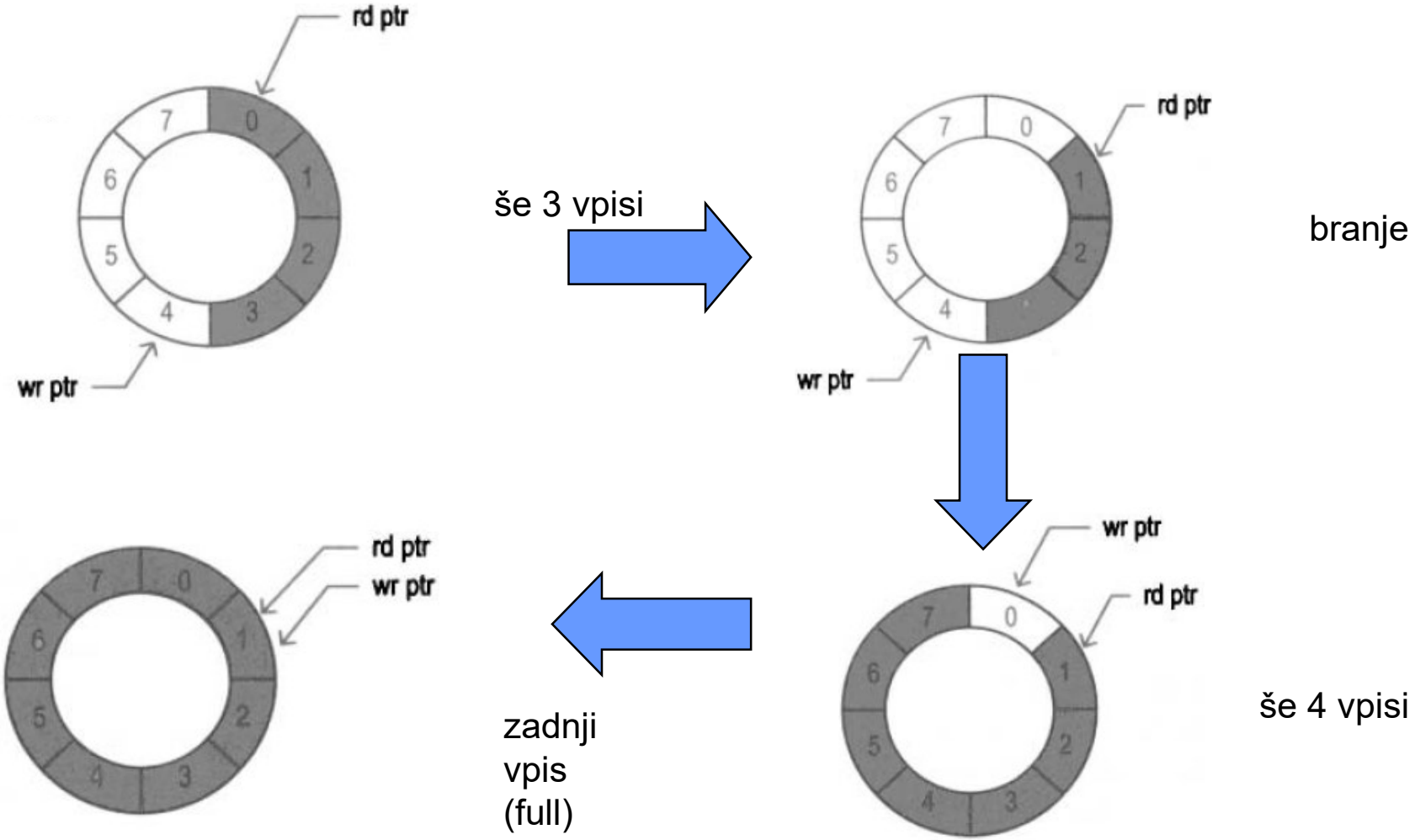
- FIFO – prilagodljiv vmesnik med dvema sistemoma
- Ima dva kontrolna signala wr in rd za operaciji pisanja in branja.
- Ko aktiviramo **wr**, se podatek z vhoda vpiše v polje registrov na ustrezno mesto.
- Signal **rd** ne služi branju, ampak "pomnjenju" da lahko neko lokacijo polja registrov ponovno vpišemo.
- Ima zastavici **full** in **empty**, ki služita detekciji stanja FIFO (poln/prazen).
  - Če se postavi "**full**" nadaljnji vpis ni možen.
  - Pri postavljenem "**empty**" branje ni možno.

FIFO s pomočjo krožnega vmesnika

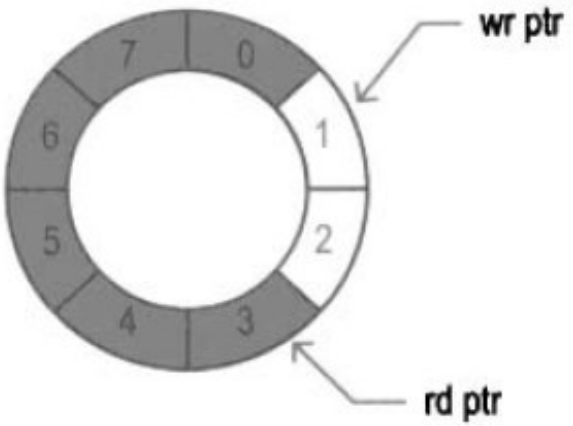
- Izvedba FIFO s pomočjo krožnega vmesnika (ang. circular buffer)
- Najenostavnejša možnost izvedbe FIFO je z uporabo polja registrov in krmilnega vezja.
- Polje registrov si predstavljamo urejene v krogu – krožni pomnilnik, ki ima dva kazalca:
- Pisalni kazalec: w_ptr kaže na začetek vmesnika
- Bralni kazalec: r_ptr kaže na konec vmesnika
- Ob operaciji vpisa/branja se ustrezni kazalec pomakne na naslednje mesto.
- Glede na položaj kazalcev, se kombinacijsko postavita zastavici full (ni možen vpis) in empty (ni možno branje).



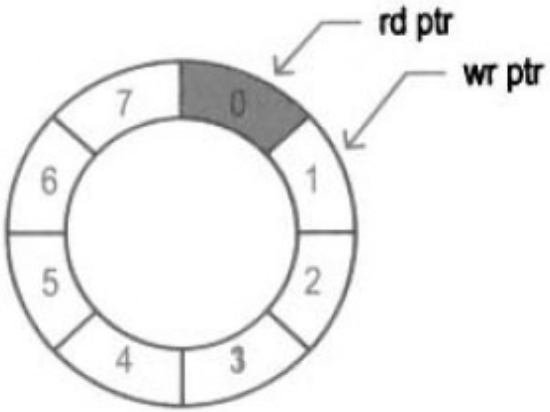
FIFO s pomočjo krožnega vmesnika



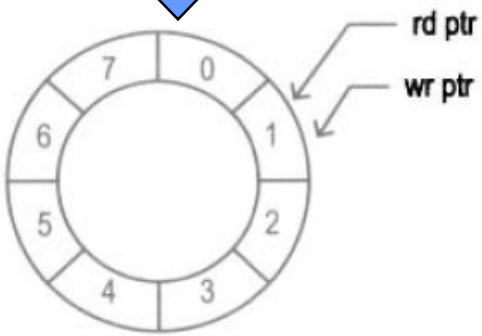
FIFO s pomočjo krožnega vmesnika



2 branji



še 5 branj



še zadnje branje (empty)

FIFO s krožnim vmesnikom

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fifo is
generic (     FIFO_WIDTH : natural := 8;
              FIFO_SIZE : natural := 16
              );
port ( clk, rst : in std_logic;
rd : in std_logic;    -- omogoci branje
wr : in std_logic;     -- omogoci vpis
r_data : out std_logic_vector(FIFO_WIDTH - 1 downto 0); --
  izhod FIFO
w_data : in std_logic_vector (FIFO_WIDTH - 1 downto 0); --
  vhod FIFO
empty : out std_logic;    -- postane '1', ko je FIFO prazen
full : out std_logic      -- postane '1', ko je FIFO poln
       );
end fifo;
```

Deklaracija simbolov v FIFO

```
type memory_type is array (0 to FIFO_SIZE - 1) of std_logic_vector(FIFO_WIDTH - 1 downto 0);
signal memory : memory_type := (others => (others => '0'));   -- 2D polje RAM spomina
signal    wr_ptr, wr_ptr_next,  --kazalca vpisa
          rd_ptr, rd_ptr_next : integer range 0 to FIFO_SIZE - 1; -- kazalca branja
signal    wr_en, rd_en : std_logic;     -- omogocitev vpisa in branja
signal    full_sig, empty_sig,
          full_sig_next, empty_sig_next : std_logic := '0'; -- indikatorja poln/prazen
begin
wr_en <= wr and not full_sig;           -- vpis polnega fifo ni mozen
rd_en <= rd and not empty_sig;          -- branje praznega fifo ni mozno
```

Kazalca branja in pisanja (r_ptr, w_ptr):

- w_ptr oz. r_ptr → **izhoda** iz podatkovnega registra, ki hranita vrednost
- w_ptr_next oz. r_ptr_next → **vhoda** v podatkovni register (vrednost, ki se bo vpisala v register ob naslednjem CLK)
- w_pt _succ oz. r_ptr_succ → **spremenljivki**, ki določata vrednost (w_ptr_reg + 1 oz. r_ptr_reg + 1)

Zastavici (full, empty):

- full oz. empty → **izhoda** iz FF, ki hranita (full oz. empty) vrednost
- full_next oz. empty_next → **vhoda** v FF (vrednost, ki se bo vpisala v FF ob naslednjem CLK)

Branje in vpis polja registrov v FIFO

```
process(clk, wr_en)
begin
if rising_edge(clk) then
   if(wr_en = '1') then
   memory(wr_ptr) <= w_data;    -- vpis je mogoc
   end if;
end if;
end process;
r_data <= memory(rd_ptr); -- asinhron izhod za branje
```

FIFO krmilnik

- Krmilnik skrbi za:
  - povečevanje kazalcev wr_ptr, rd_ptr
  - krmiljenje zastavic full, empty s pomočjo ustreznih FF

Krmiljenje zastavic:

- wr_en, rd_en = 00: → ni operacije - ohranita se trenutni vrednosti kazalcev (wr_ptr_next <= wr_ptr) in (rd_ptr_next <= rd_ptr)
- wr_en, rd_en = 01 : → branje
  - če ni empty, povečaj rd_ptr in zbriši full.
  - Povečaj vrednost kazalca branja (rd_ptr_next <= rd_ptr_succ <= rd_ptr+1)
  - če bo *naslednje branje* povzročilo stanje empty (rd_ptr_succ =w_ptr), potem postavi empty_next – da se empty postavi ob naslednjem CLK.
- wr_en, rd_en = 10 : → vpis
  - če ni full, povečaj wr_ptr in zbriši empty.
  - Povečaj vrednost kazalca pisanja (wr_ptr_next = wr_ptr_succ <= wr_ptr+1)
  - če bo *naslednji vpis* povzročil stanje full (wr_ptr_succ = rd_ptr_reg), potem postavi full_next – da se full postavi ob naslednjem CLK.
- wr_en, rd_en = 11 : → hkratni vpis in branje
  - povečaj oba kazalca (wr_ptr_next = wr_ptr_succ) in (rd_ptr_next = rd_ptr_succ)

FIFO krmilnik – logika naslednjih stanj

```
process(clk, rst)
begin
   if rising_edge(clk) then
        if (rst = '1')  then
                rd_ptr <= 0;
                wr_ptr <= 0;
                full_sig <= '0';
                empty_sig <= '1';-- ob ponastavitvi je fifo prazen
        else
        -- ob aktivnem robu signala ure shrani nove vrednosti
                rd_ptr <= rd_ptr_next;
                wr_ptr <= wr_ptr_next;
                full_sig <= full_sig_next;
                empty_sig <= empty_sig_next;
        end if;
   end if;
end process;

end Behavioral;
```

FIFO krmilnik - vpis

```
process(wr_en, rd_en, wr_ptr, rd_ptr, empty_sig, full_sig)
variable wr_ptr_succ, rd_ptr_succ : integer range 0 to FIFO_SIZE - 1;   -- spremenljivki naslednjih lokacij
begin
if wr_ptr = (FIFO_SIZE - 1) then
     wr_ptr_succ := 0; -- roll over -> ne smemo prekoraciti obmocja stevil 0 to FIFO_SIZE - 1
else
     wr_ptr_succ := wr_ptr + 1;
end if;

if rd_ptr = (FIFO_SIZE - 1) then
     rd_ptr_succ := 0; -- roll over -> ne smemo prekoraciti obmocja stevil 0 to FIFO_SIZE - 1
else
     rd_ptr_succ := rd_ptr + 1;
end if;

wr_ptr_next <= wr_ptr; -- privzeto: brez sprememb kazalcev in indikatorjev prazen/poln
rd_ptr_next <= rd_ptr;
full_sig_next <= full_sig;
empty_sig_next <= empty_sig;

if(wr_en = '1' and rd_en = '0') then -- vpis
     if (full_sig = '0') then        -- vpis polnega sklada ni mozen
            wr_ptr_next <= wr_ptr_succ; -- povecaj kazalec vpisa - pozor: avtomatski rollover iz 1111 na 0000
            empty_sig_next <= '0';  -- ce vpisujemo, fifo zagotovo ni prazen
            if (wr_ptr_succ = rd_ptr ) then
                        full_sig_next <= '1';-- ce kazalec vpisa ujame kazalec branja ni vec kam pisati - poln
            end if;
     end if;
end if;
```

FIFO krmilnik – branje

```
if(wr_en = '0' and rd_en = '1') then -- branje
   if (empty_sig = '0') then -- branje praznega sklada ni mozno
        rd_ptr_next <= rd_ptr_succ;
        full_sig_next <= '0'; -- ce beremo, fifo zagotovo ni poln
        if ( rd_ptr_succ = wr_ptr ) then
                empty_sig_next <= '1'; -- ce kazalec branja ujame
   kazalec vpisa ni vec kaj brati - fifo prazen
        end if ;
   end if ;
end if;

if(wr_en = '1' and rd_en = '1') then -- hkraten branje in vpis
   wr_ptr_next <=  wr_ptr_succ; -- ohrani stanje zastavic full in
   empty (ce je bil fifo prej poln, je tudi zdaj - prazen pa ne more
   biti)
   rd_ptr_next <= rd_ptr_succ;
end if;
end process;
end Behavioral;
```

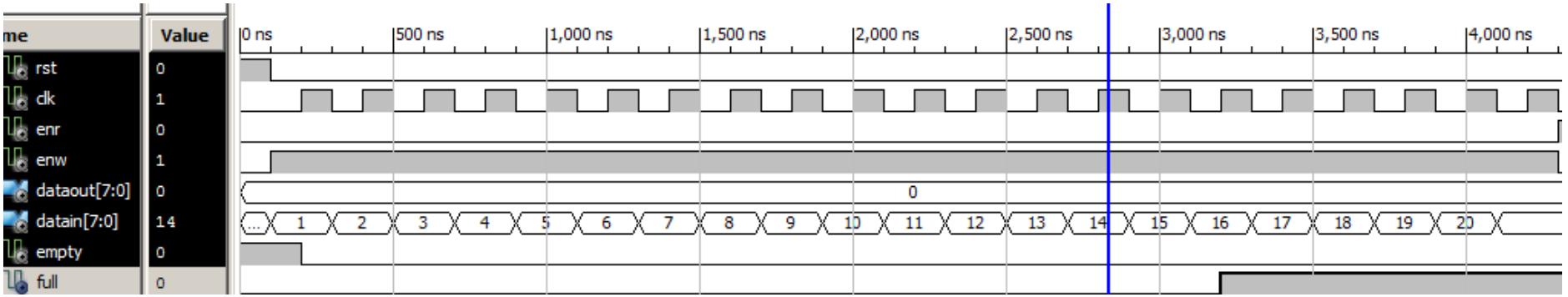
FIFO z uporabo core generatorja

FIFO z uporabo core generatorja

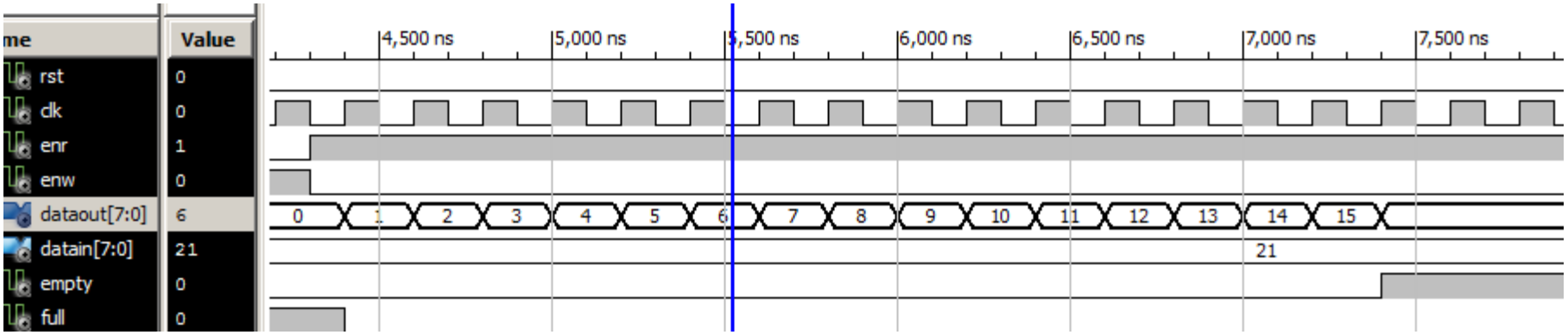
```
COMPONENT fifo
  PORT (
    clk, srst : IN STD_LOGIC;
    din : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    wr_en, rd_en : IN STD_LOGIC;
    dout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    full, empty : OUT STD_LOGIC
  );
END COMPONENT;
```

FIFO z uporabo core generatorja

Vpis 16 podatkov



Branje 16 podatkov

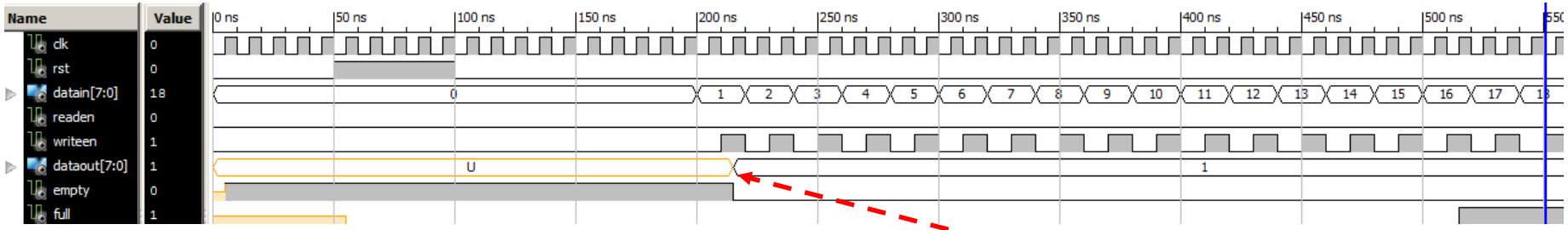


FWFT/lookahead FIFO vmesnik

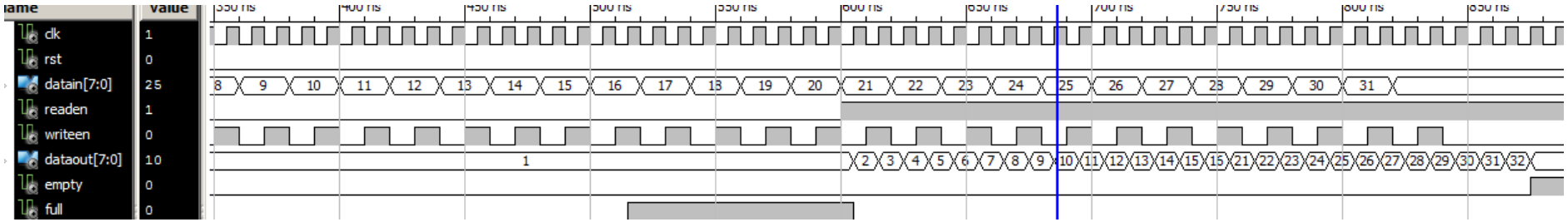
- Glavna razlika med klasičnim FIFO in FWFT (first word fall-through) FIFO je, da se pri slednjem prvovpisani podatek *takoj* (asinhrono) pojavi na izhodu.
- To omogoča, da podatek preberemo ob naslednjem robu signal ure ne da bi zahtevali branje FIFO.
- Poleg tega FWFT FIFO omogoča hkratno branje in vpis. Vpisani podatek je na voljo ob naslednjem robu signala ure.

FWFT/lookahead FIFO vmesnik

Ponastavitev in vpis 16 elementov v FIFO.



Nadaljevanje: Branje FIFO



Načrtovanje digitalnih vezij

Uporaba pomikalnih registrov:
LFSR / psevdonaključni
generatorji

Polinomi po modulu 2

- Odcepe lahko izrazimo z aritmetiko končnih polj kot polinom po modulu 2.
- Polinom po modulu 2 ima samo koeficiente '1' ali '0'.
- Takemu polinomu pravimo značilni polinom (characteristic polynomial)
- Če so odcepi na **16. 14. 13.** in **11.** mestu potem se značilni polinom glasi:

$$x^{16} + x^{14} + x^{13} + x^{11} + 1$$

- Člen "+1" v polinomu ni odcep, ampak ustreza vhodu v prvo mesto: (x_0 je ekvivalenten 1)
- Potence ustrezajo mestom odcepov, šteto od prvega mesta.
- Prvo mesto je vedno povezano na *vhod*, zadnje na *izhod*

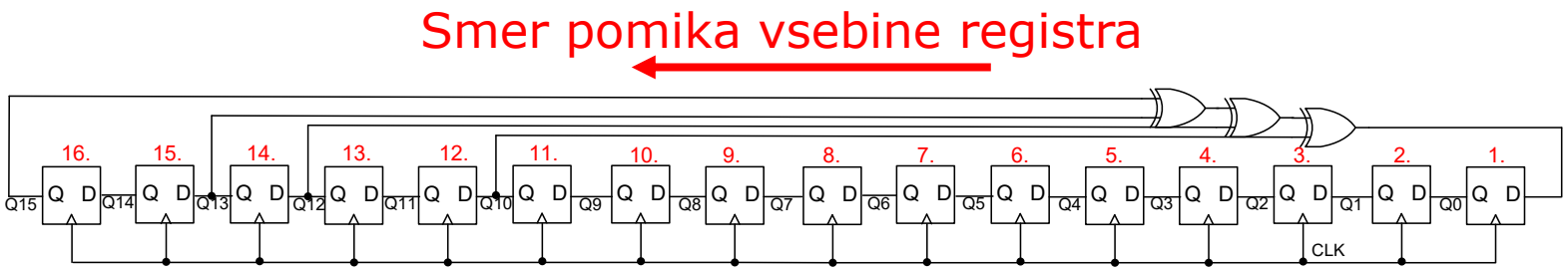


Tabela primitivnih polinomov

- Števna sekvenca LFSR bo najdaljša, če je število odcepov **sodo**.
- Zaporedje odcepov ne sme imeti nobenega skupnega delitelja
- Obstaja lahko več kot ena najdaljša števna sekvenca za dano dolžino LFSR:
  - Če je zaporedje odcepov v n-bitnem LFSR [n, A, B, C, 0], kjer 0 ustreza členu $x_0 = 1$,
  - potem je ustrezno **zrcalno zaporedje** [n, n – C, n – B, n – A, 0].
- Zaporedje odcepov [32, 7, 3, 2, 0] ima **zrcalno zaporedje** [32, 30, 29, 25, 0]. Obe dajeta najdaljši števni zaporedji.

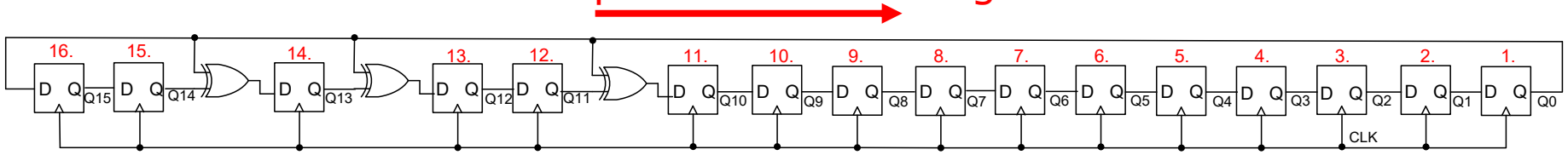
Mest (n)	Značilni polinom	Perioda ($2^n - 1$)
2	$x^2 + x + 1$	3
3	$x^3 + x^2 + 1$	7
4	$x^4 + x^3 + 1$	15
5	$x^5 + x^3 + 1$	31
6	$x^6 + x^5 + 1$	63
7	$x^7 + x^6 + 1$	127
8	$x^8 + x^6 + x^5 + x^4 + 1$	255
9	$x^9 + x^5 + 1$	511
10	$x^{10} + x^7 + 1$	1023
11	$x^{11} + x^9 + 1$	2047
12	$x^{12} + x^{11} + x^{10} + x^4 + 1$	4095
13	$x^{13} + x^{12} + x^{11} + x^8 + 1$	8191
14	$x^{14} + x^{13} + x^{12} + x^2 + 1$	16383
15	$x^{15} + x^{14} + 1$	32767
16	$x^{16} + x^{14} + x^{13} + x^{11} + 1$	65535
17	$x^{17} + x^{14} + 1$	131071
18	$x^{18} + x^{11} + 1$	262143
19	$x^{19} + x^{18} + x^{17} + x^{14} + 1$	524287

Galois-ev LFSR

- Imenovanje po francoskem matematiku: Évariste Galois
- Alternativna izvedba LFSR
  Sinonimi: "modularna", "XOR znotraj",
  "ena na več" (ang. one-to-many) izvedba
- Funkcijsko ekvivalenten standardnemu LFSR
- Biti, ki niso odcepi, se vedno pomaknejo za eno mesto nespremenjeni.
- Na mestih odcepov izvedemo XOR z izhodnim mestom registra preden jih shranimo na naslednje mesto.
- Novo izhodno mesto postane naslednje vhodno mesto.
- Delovanje:
  - Če je izhodno mesto '0', potem se vsa mesta registra pomaknejo **nespremenjeno** in vhodno mesto postane **'0'**
  - Če je izhodno mesto '1', potem se vsa mesta registra pomaknejo **komplementirano** (1➔0, 0➔1) in vhodno mesto postane **'1'**

16-bitni Galois-ev LFSR

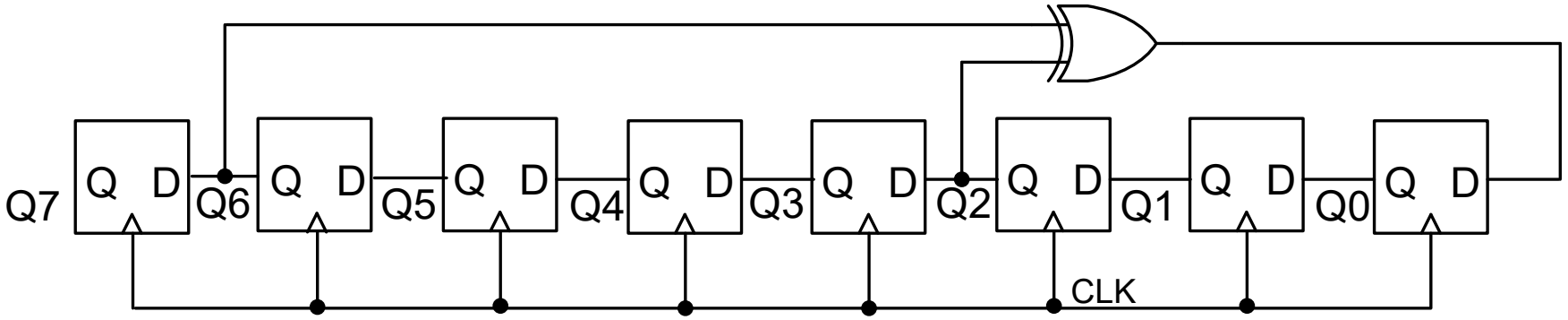
- Števna sekvenca je sestavljena iz 2^{16}-1 stanj
  (razen stanja samih ničel)
- Za enako števno sekvenco moramo pri Galois-evem LFSR uporabiti *zrcalno zaporedje* odcepov glede na Fibonacci-jevm LFSR, sicer bo števna sekvenca obrnjena (MSB↔LSB)
- Spodnja realizacija ima enako števno sekvenco kot prej predstavljeni Fibonacci-jev LFSR, kljub temu da vsebina pomikalnega registra praviloma ni enaka.
- Galois-ev LFSR *ne združuje vseh odcepov* da bi z XOR tvoril novo vrednost vhoda:
  - XOR se izvaja znotraj registra
  - XOR vrata niso vezana zaporedno
- Manjše zakasnitve celotnega LFSR
  (zakasnitev celotnega LFSR je reda zakasnitve *enih* XOR vrat, ne pa vsota zakasnitev verige XOR vrat)
- Galois-ev LFSR je hitrejši ➔ izračun lahko izvajamo vzporedno



Galois-ova polja (GF)

Primeri odcepov LFSR z XOR vrati
(brez zrcalnih zaporedij odcepov)

n	3	4	5	6	7	8	9	10
	1, 3	1, 4	2, 5	1, 6	4, 7	1, 2, 7, 8	5, 9	7, 10



Vir:http://www.newwaveinstruments.com/resources/articles/m_sequence_linear_feedback_shift_register_lfsr.htm

Galois-ova polja (GF)

Bistvena pogoja za naključnost:

- Verjetnost pojavljanja ničel in enic je približno enaka,
- Verjetnost '0' oz. '1' ni odvisna od prejšnje vrednosti. Torej vrednosti v zaporedju medsebojno niso korelirane.

Uporaba LFSR

- Delovanje:
  - V splošnem so XOR vrata 2-vhodna in se nikdar ne vežejo zaporedno
  - Minimalna perioda ure teh vezij je:

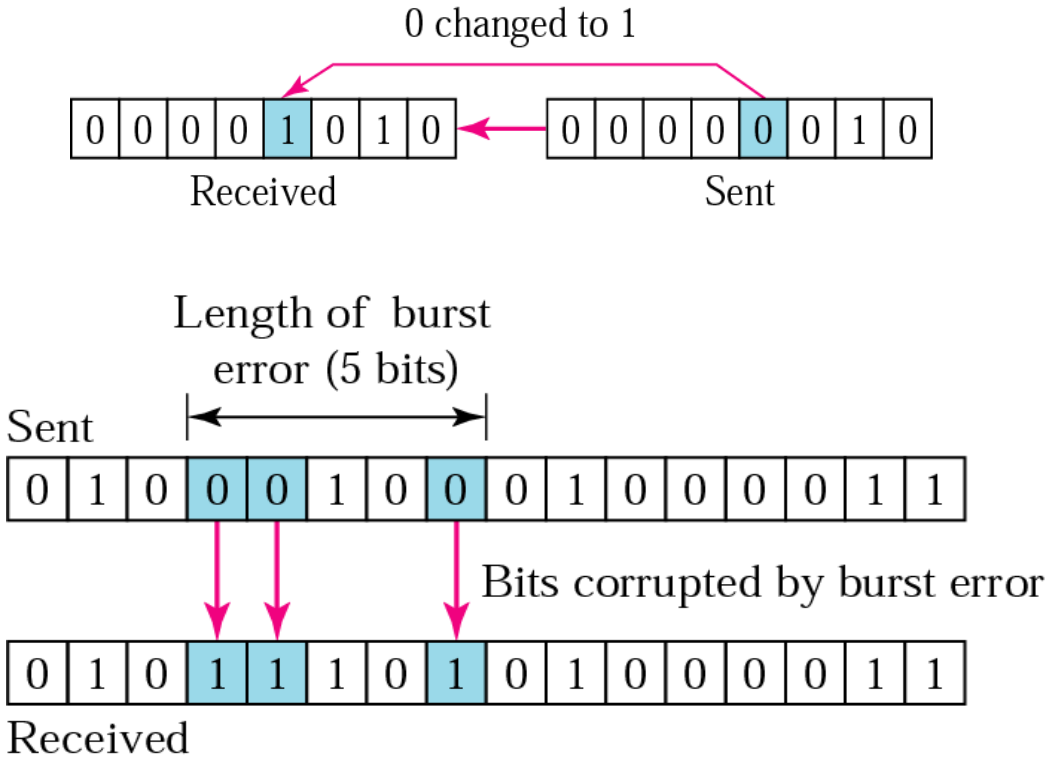
    $T > T_{2\text{-vhod-XOR}}$ + perioda CLK
  - Zelo majhne zakasnitve
  - Zakasnitev neodvisna od n!
- To lahko uporabljamo za *hitre števce*, pri katerih sekvenca štetja ni pomembna.
  - Primer: mikro-koda
- Uporaba kot *generator naključnih vrednosti*.
  - Sekvenca je psevdonaključna:
    - Število se pojavi v naključni sekvenci
    - Ponovi se vsakih 2^n-1 vzorcev
  - Naključna števila uporabljamo za:
    - Računalniška grafika
    - Kriptografija
    - Avtomatsko testiranje
- Uporaba za detekcijo in odpravljanje napak pri prenosu
    - CRC (cyclic redundancy codes)
    - Ethernet

Načrtovanje digitalnih vezij

Določanje in odpravljanje napak
pri prenosu informacije

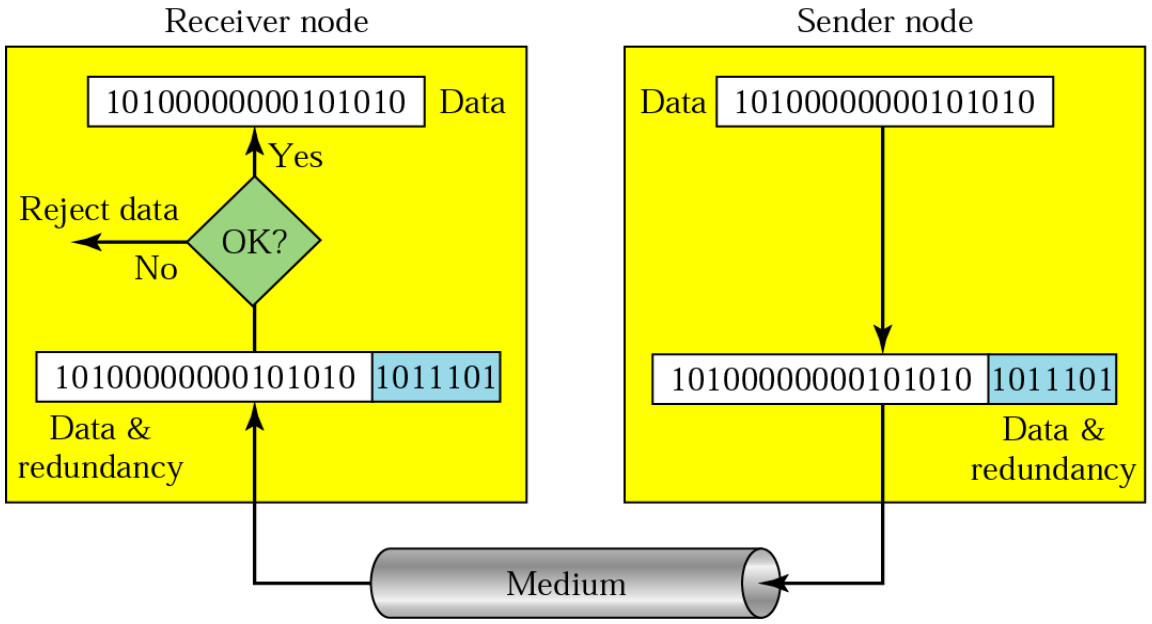
Vrste napak pri prenosu

- Enobitna napaka (single-bit error)
- Rafalna napaka (burst error) pomeni, da sta se v preneseni enoti spremenila 2 bita ali več.



Določanje napak

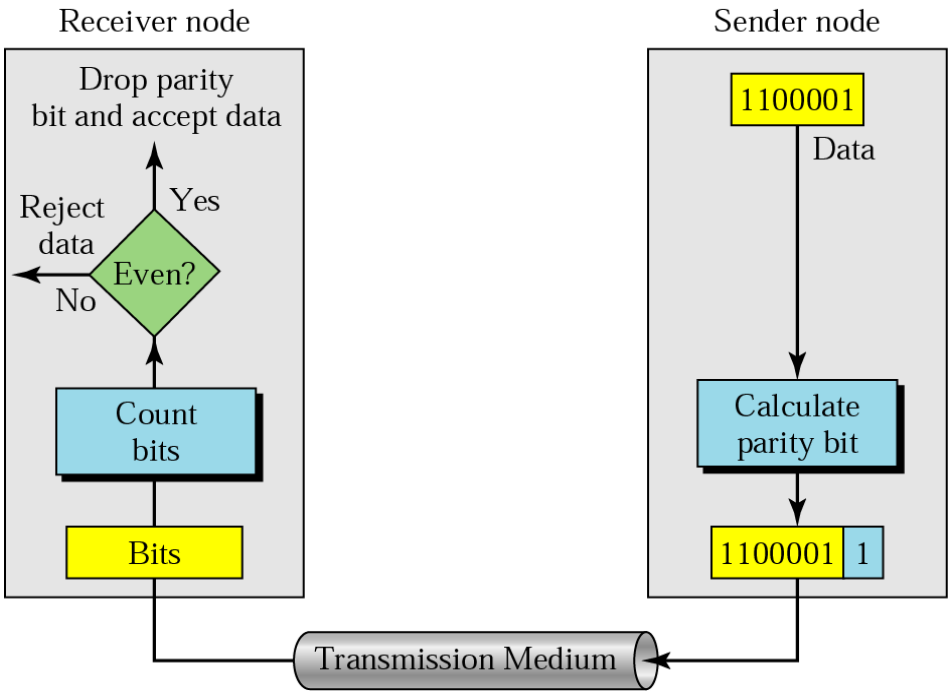
- Mehanizmi za določanje napak temeljijo na konceptu redundantnosti, po katerem pri prenosu informacije dodamo bite, ki jih bomo lahko na sprejemni strani uporabili za preverjanje pravilnosti prenosa.



- Kontrola parnosti (parity check)
- Cyclic redundancy check (CRC)

PARNOST (Parity)

- Enostavno ali dvodimenzionalno
- V preverjanju parnosti dodamo preneseni enoti informacije bit parnosti (oz. paritete, ang. parity bit), tako da je celotno število enic v informaciji liho – če govorimo o lihi parnosti, oziroma sodo, če govorimo o sodi parnosti.



Zgled

- Oddajnik želi oddati besedo "world". V ASCII so znaki oddane besede kodirani kot:
  `1110111 1101111 1110010 1101100 1100100`

- Dejansko so bili oddani naslednji biti:
  `1110111`**`0`** `1101111`**`0`** `1110010`**`0`** `1101100`**`0`** `1100100`**`1`**

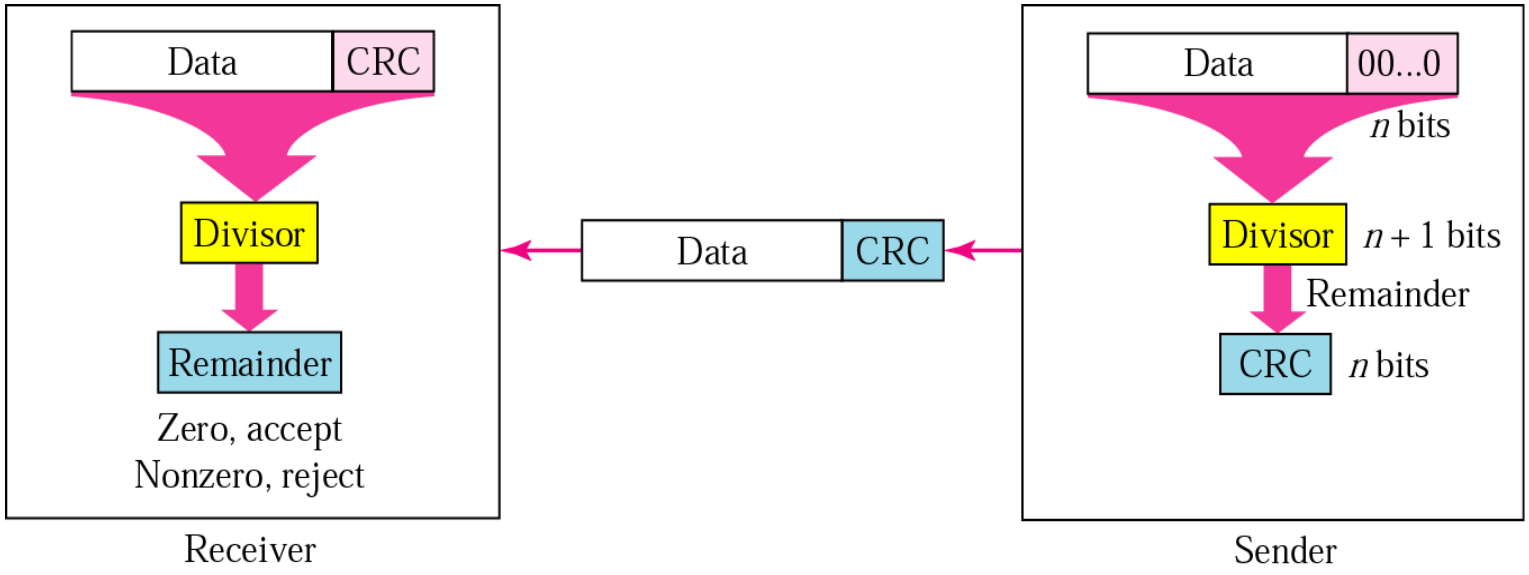
Recimo, da sprejemnik sprejme oddano besedo "world" brez napak pri prenosu.

`11101110 11011110 11100100 11011000 11001001`

Sprejemnik šteje enice v vsakem sprejetem znaku in dobi zaporedje sodih števil enic (6, 6, 4, 4, 4) ➔ Podatek se je prenesel pravilno.

CRC – cyclic redundancy check

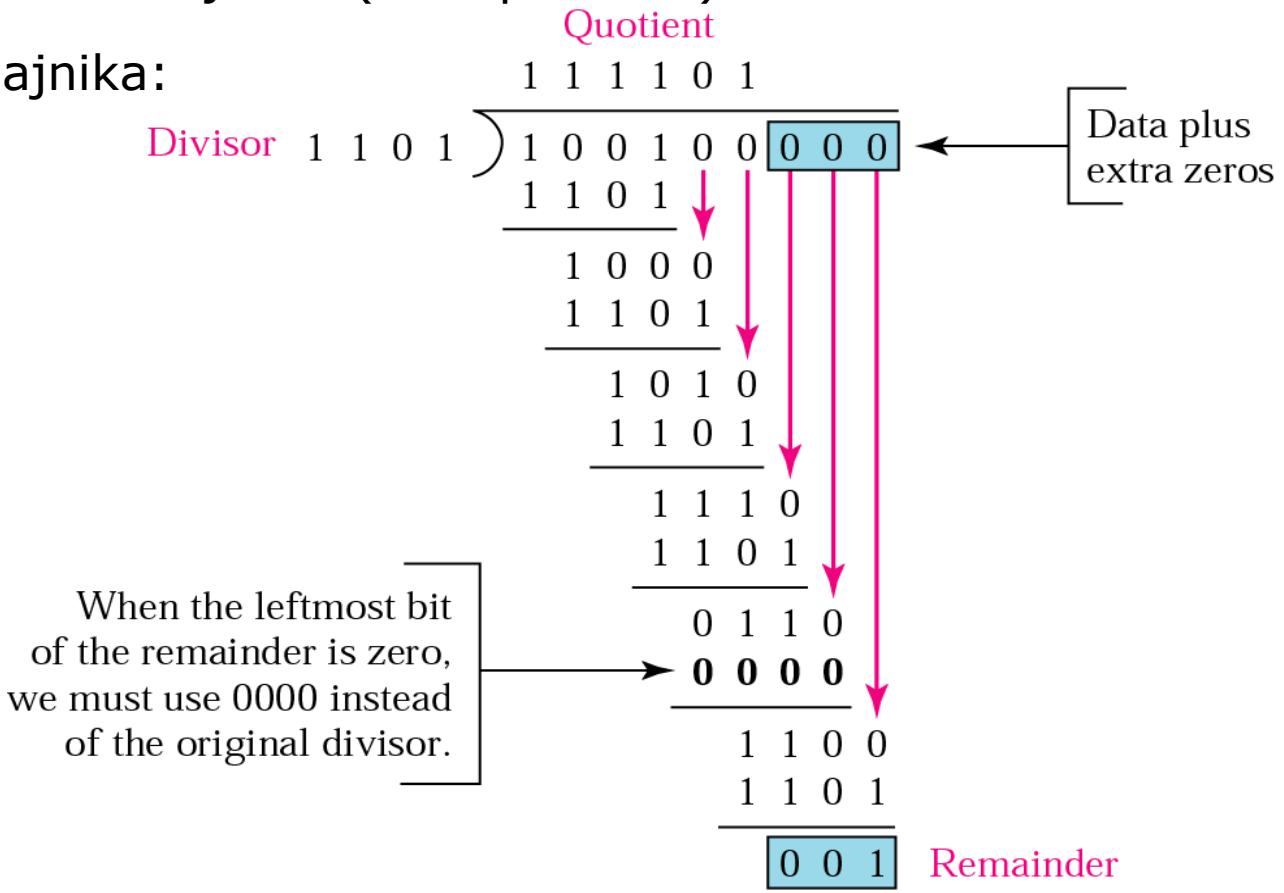
- Uporablja binarno deljenje
- Doda CRC ostanek prenesenim podatkom
- Število dodanih ničel je eno manj kot delitelj (ang. divisor)
- Dodani CRC ostanek naredi celotno zaporedje bitov (informacija + CRC ostanek) deljiv z deliteljem.



Dvojiško deljenje v CRC generatorju

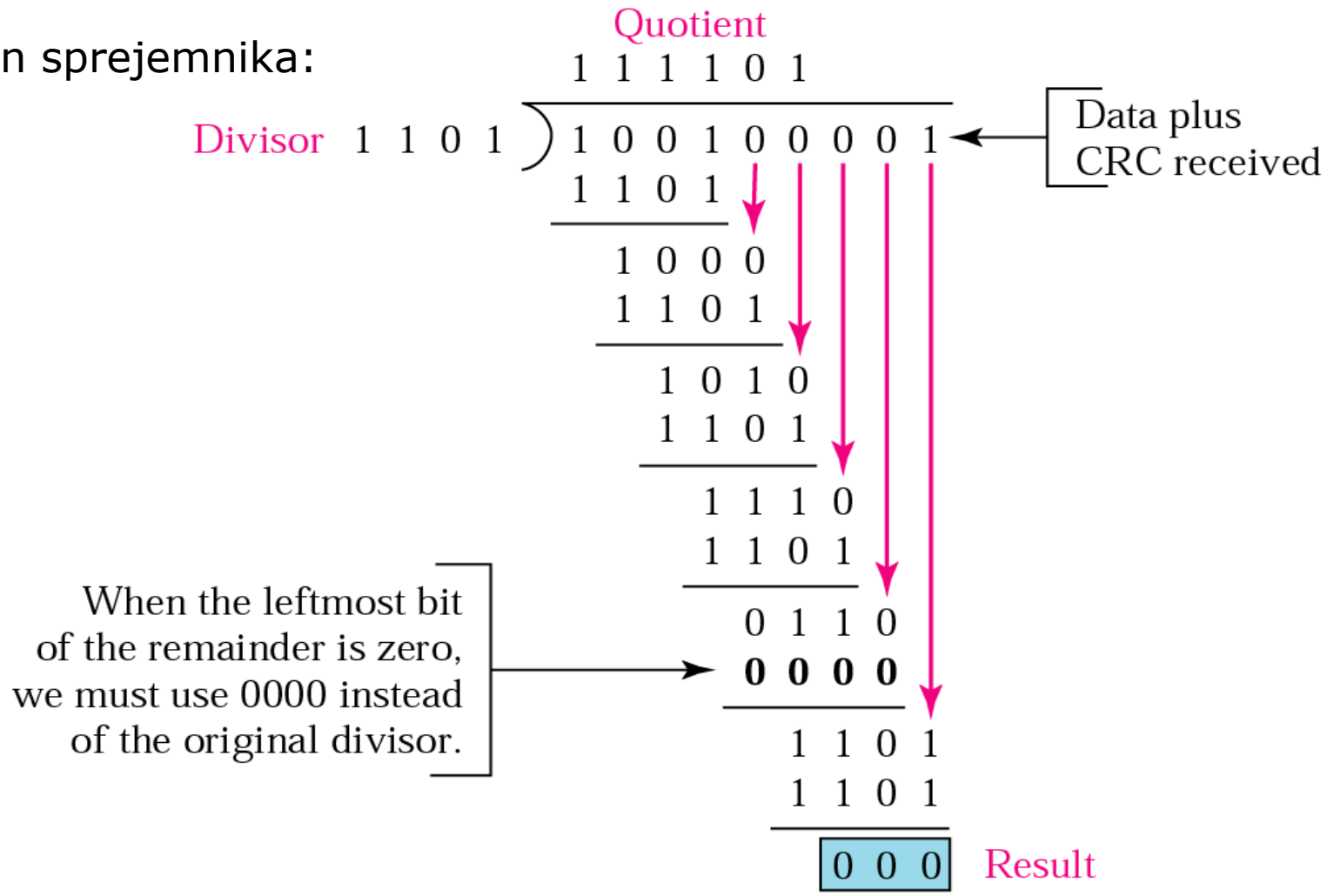
- Uporablja deljenje po modulu 2
- Vsak bit delitelja odštejemo od ustreznega bita deljenca, ne da bi to vplivalo na višje bite (brez sposodkov).

Izračun oddajnika:



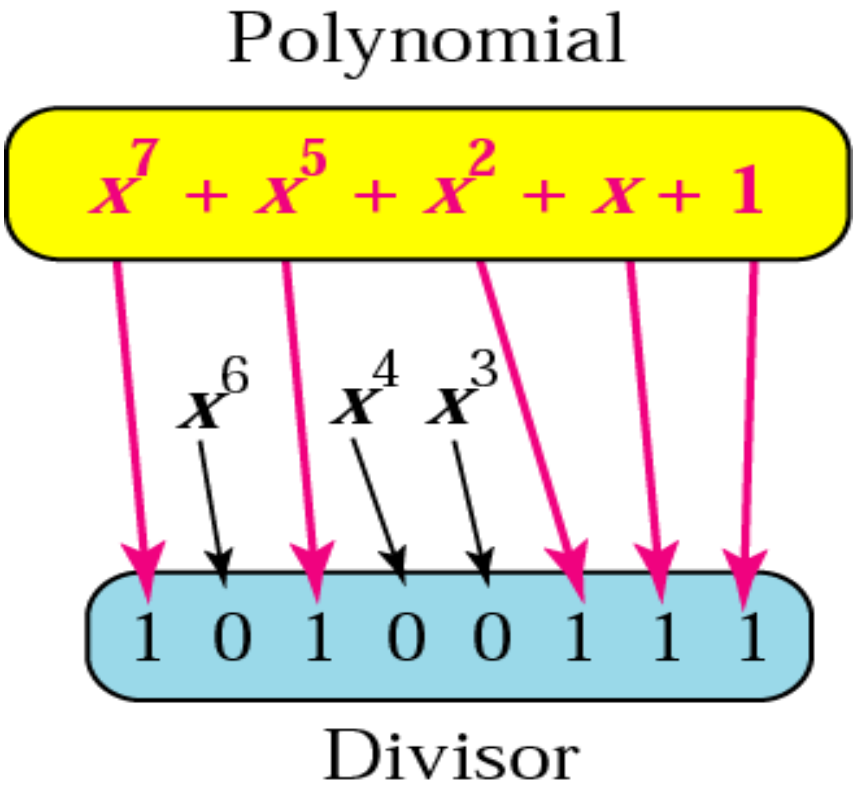
Dvojiško deljenje v CRC generatorju

Izračun sprejemnika:



Polinomi, ki predstavljajo CRC delitelje

- Polinom naj ne bi bil deljiv z x.
- Polinom naj bi bil deljiv z x+1



Standardni CRC polinomi

CRC-1	$x + 1$	*bit parnosti*	0x1
CRC-5-USB	$x^5 + x^2 + 1$	USB token paketi	0x05
CRC-7	$x^7 + x^3 + 1$	telekomunikacijski sistemi, MMC, SD	0x09
CRC-8-CCITT	$x^8 + x^2 + x + 1$	ISDN Header Error Control	0x07
CRC-8-Dallas/Maxim	$x^8 + x^5 + x^4 + 1$	1-Wire vodilo	0x31
CRC-8	$x^8 + x^7 + x^6 + x^4 + x^2 + 1$	0xD5	0xAB
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + x + 1$	telekomunikacijski sistemi	0x80F
CRC-15-CAN	$x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$	CAN	0x4599
CRC-16-IBM	$x^{16} + x^{15} + x^2 + 1$	Bisync, Modbus, USB, ANSI X3.28,	0x8005
CRC-16-CCITT	$x^{16} + x^{12} + x^5 + 1$	X.25, V.41, HDLC, XMODEM, Bluetooth, SD	**0x1021**
CRC-16-T10-DIF	$x^{16} + x^{15} + x^{11} + x^9 + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$	SCSI DIF	0x8BB7
CRC-16-DECT	$x^{16} + x^{10} + x^8 + x^7 + x^3 + 1$	cordless telephones	0x0589
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$	Ethernet, SATA, MPEG-2, Gzip, PKZIP, POSIX, PNG	0x04C11DB7
CRC-40-GSM	$x^{40} + x^{26} + x^{23} + x^{17} + x^3 + 1$	GSM nadzorni kanal	0x0004820009

CRC-12

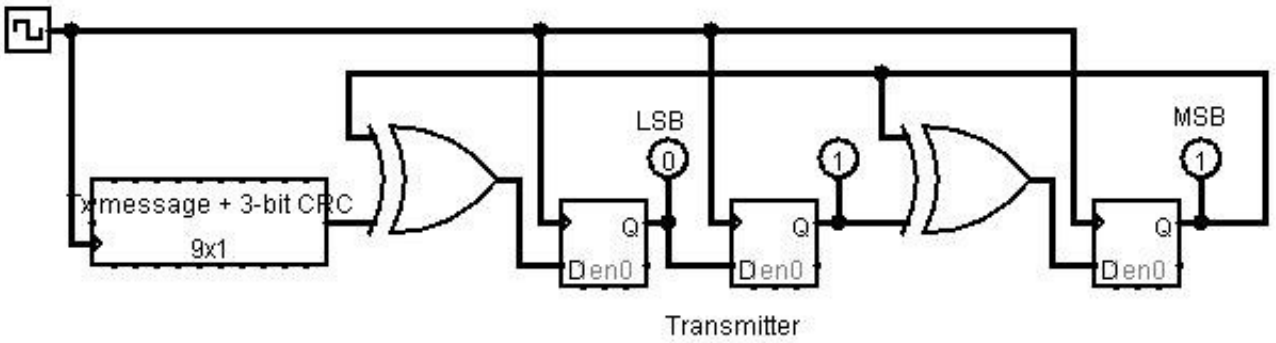
- CRC-12 polinom se glasi:

$$x^{12} + x^{11} + x^3 + x + 1$$

- Tak polinom bo:
  - omogočal zaznavanje vseh rafalnih napak, ki vplivajo na sodo število bitov.
  - omogoča zaznavanje rafalnih napak dolžine manj ali enako 12,
  - omogoča zaznavanje rafalnih napak dolžine več kot 12 v 99.97%

Izračun CRC z uporabo LFSR

CRC using Galois LFSR: polynomial x^3+x^2+1 or (1101)



Transmitter

Set the initial message value (eg. 011110) and three zero CRC bits
in the TX shift register by pressing CTRL+R and clicking the reg values from right to left (MSB rightmost).
After shifting out the entire register (CTRL+T - single clock toggle), the CRC remainder will appear on diodes
In case message is 011110, the CRC remainder is 011.

In order to verify the calculated CRC, enter the same message
and append the CRC remainder in MSB rightmost order to the end of the message
and clock the entire contents of register - if CRC remainder is correct, the result should be 000

Logisim\CRC\CRC_Galois_1101_transmitter_only.circ

Zaporedni izračun CRC v VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity crc_serial is
generic( crc_size : natural := 8; --16;    --16 --8; --32;
    POLY : integer := 16#07#; --16#1021#; --16#8005#; --16#32#; --16#04C11DB7#;
    SEED : integer := 0    --0
    );
port (clk,-- signal ure
    rst,-- asinhroni reset
    en, -- omogoci izracun; postavi na '1' za zacetek izracuna
    d_in : in STD_LOGIC;   -- vhodni niz podatkov
    crc_out: out STD_LOGIC_VECTOR(crc_size - 1 downto 0);    -- izracunan crc
    done_tick: out STD_LOGIC        -- se postavi na '1', ko je crc izracunan
);
end crc_serial;
architecture a1 of crc_serial is
signal crc_reg : STD_LOGIC_VECTOR(crc_size - 1 downto 0) :=
    std_logic_vector(to_unsigned(SEED, crc_size));
constant poly_const : STD_LOGIC_VECTOR(crc_size - 1 downto 0) :=
    std_logic_vector(to_unsigned(POLY, crc_size));
signal crc_next : STD_LOGIC_VECTOR(crc_reg'range);
```

Zaporedni izračun CRC v VHDL

```
begin
crc_reg_proc: process (clk, rst, en)
begin
     if (rst = '1') then
               crc_reg <= std_logic_vector(to_unsigned(SEED, crc_reg'length));
elsif rising_edge(clk) then
     if (en = '1') then
               crc_reg <= crc_next;        -- vpis podatkovnega registra
     end if;
end if;
end process;
     crc_next <= (crc_reg(crc_reg'left - 1 downto 0) & d_in) xor ((crc_reg'range => crc_reg(crc_reg'left)) and poly_const);
     -- ce je MSB crc_reg enak '1', naredi xor na mestih, kjer je poly enak '1'. Kjer je poly '0' naredi samo pomik.
     crc_out <= crc_reg;  -- izhodni crc
ctr_proc: process (clk, rst, en)
variable i : integer range 0 to (2 * crc_size) := 0; -- pretvorba je koncana, ko se izvede (2 * crc_size) pomikov
begin
if (rst = '1') then
     i := 0; -- ponastavi stevec iteracij izracuna
     done_tick <= '0';
elsif rising_edge(clk) then
     if (en = '1') then
               if (i = 2 * crc_size) then
                         i := 0;       -- stevec iteracij izracuna nazaj na 0
               else
                         i := i + 1;   -- povecaj stevec iteracij izracuna
               end if;
     end if;
end if;
done_tick <= '0';          -- privzeta (ang. default) vrednost signala done_tick = '0'
if (i = 2 * crc_size) then -- kombinacijski primerjalnik velikosti
     done_tick <= '1'; -- pretvorba se konca !!po!! 2 * crc_size iteracijah
end if;
end process;
end a1;
```

Zaporedni izračun CRC je pravzaprav izveden z enim stavkom!
Spodnji proces je števec pomikov.

Zaporedni izračun CRC v VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use IEEE.numeric_std.all;
use work.crc_calc_serial_pkg.all;
ENTITY crc_serial_tb IS
generic (delay : time := 5 ns; TEST_DATA : integer := 16#5C#; crc_size : natural := 8; POLY : integer := 16#32#; SEED : integer := 0 );
END crc_serial_tb;
ARCHITECTURE arch OF crc_serial_tb IS
COMPONENT crc_serial is
      generic( crc_size : natural := 8; POLY : integer := 16#07#; SEED : integer := 0
      );
      port (clk, -- signal ure
      rst,       -- asinhroni reset
      en,        -- omogoci izracun; postavi na '1' za zacetek izracuna
      d_in : in STD_LOGIC;      -- vhodni zaporedni niz podatkov
      crc_out : out STD_LOGIC_VECTOR(crc_size - 1 downto 0); -- izracunan crc
      done_tick: out STD_LOGIC -- se postavi na '1', ko je crc izracunan
    );
end COMPONENT;
signal clk : STD_LOGIC := '0';-- signal ure
signal rst : STD_LOGIC := '0';-- asinhroni reset
signal en : STD_LOGIC := '0';  -- omogoci izracun; postavi na '1' za zacetek izracuna
signal d_in : STD_LOGIC:= '0';-- vhodni zaporedni niz podatkov
constant slv_poly : STD_LOGIC_VECTOR(crc_size - 1 downto 0) := std_logic_vector(to_unsigned(POLY, crc_size));
constant slv_test_data : STD_LOGIC_VECTOR(crc_size - 1 downto 0) := std_logic_vector(to_unsigned(TEST_DATA, crc_size));
constant slv_shifted_test_data : STD_LOGIC_VECTOR(2 * crc_size - 1 downto 0) := slv_test_data & std_logic_vector(to_unsigned(0, crc_size));
-- testni vektor je sestavljen iz podatka (TEST_DATA) in vektorja crc_size nicel.
-- poseben primer: ce je seme (ang. seed) enako 0, potem dopolnjevanje ni potrebno,
-- ampak v crc register nalozimo kar TEST_DATA in izvedemo samo crc_size pomikov.
signal crc_out_serial : STD_LOGIC_VECTOR(crc_size - 1 downto 0);            -- zaporedno izracunan crc
signal crc_out_parallel : STD_LOGIC_VECTOR(crc_size - 1 downto 0);          -- vzporedno izracunan crc
signal done_serial_tick : STD_LOGIC;          -- se postavi na '1', ko je zaporedni crc izracunan
signal done_parallel_tick : STD_LOGIC;        -- se postavi na '1', ko je vzporedni crc izracunan

constant clock_period : time := 200 ns;
constant clock_duty_cycle : real := 0.5;
constant clock_offset : time := 100 ns;
for all: crc_serial use entity work.crc_serial(a1);
```

Zaporedni izračun CRC v VHDL

```
for all: crc_serial use entity work.crc_serial(a1);
begin
process    -- proces, ki tvori signal ure clk
begin
   wait for clock_offset;
   clock_loop : loop
        clk <= '0';
        wait for (clock_period - (clock_period * clock_duty_cycle));
        clk <= '1';
        wait for (clock_period * clock_duty_cycle);
   end loop clock_loop;
end process;
-- Povezovanje testne enote
uut: crc_serial generic map (
   crc_size => crc_size, POLY => POLY, SEED  => SEED)
port map ( clk => clk, rst => rst, en => en, d_in => d_in, crc_out =>
   crc_out_serial, done_tick => done_serial_tick);
end;
```

Zaporedni izračun CRC v VHDL

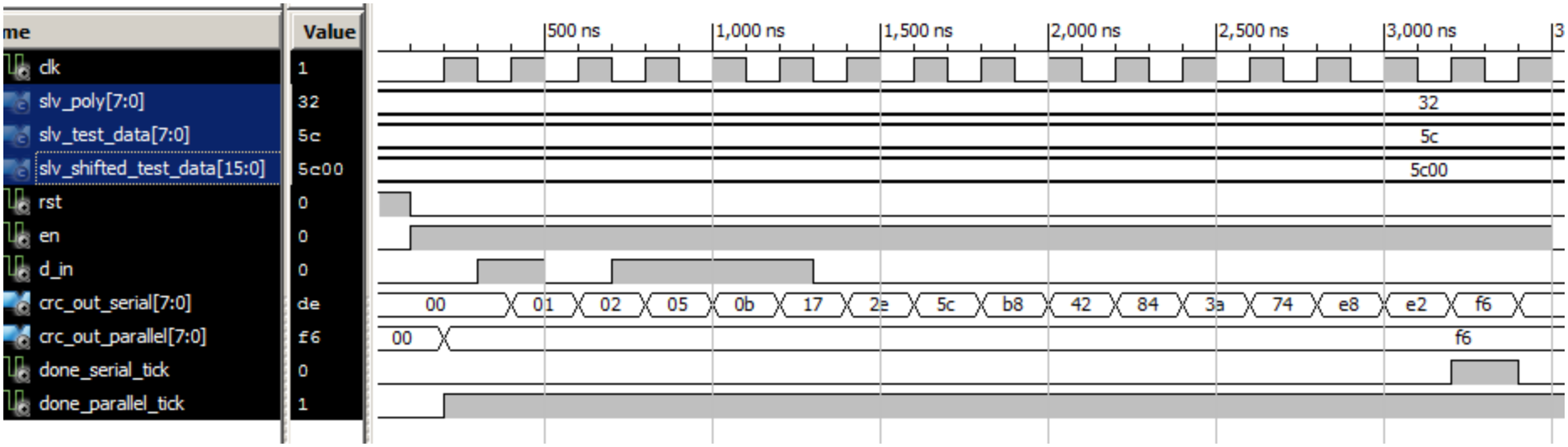
```
tx_proc: process
begin
     rst <= '1';
     en <= '0';
     wait for clock_offset;
     rst <= '0';
     en <= '1';
     for i in slv_shifted_test_data'range loop
            d_in <= slv_shifted_test_data(i);    -- pretvori vzporedni podatek TEST_DATA v zaporedno obliko
            wait for clock_period;  -- pomakni vsebino eno mesto
     end loop;
     wait for clock_period;          -- pocakamo se vsaj eno periodo
     en <= '0';
     wait;
end process;
done_serial_tick_proc: process(clk)
begin
     -- done_serial_tick signal je kombinacijski (izveden z and vrati - glej sintezo),
     -- zato ga moramo sinhronizirati s signalom ure, kar storimo z rising_edge
     -- ce ga ne, postane done_serial_tick '1' pred zadnjim ciklom pretvorbe!
     if rising_edge(clk) then
      if (done_serial_tick = '1') then
        report " Serial CRC = " & slv_image(crc_out_serial) & " (0x" & slv_hex_image(crc_out_serial)& ")";

      end if;
     end if;
end process;
```

Ti dve funkciji je potrebno programirati posebej!

Zaporedni izračun CRC v VHDL

```
crc_parallel_proc: process (clk, rst, en) is
variable crc_buf : STD_LOGIC_VECTOR (crc_out_parallel'range):= (others => '0');
begin
if (rst = '1') then
       crc_buf := (others => '0');
       done_parallel_tick <= '0';       -- privzeta (ang. default) vrednost signala done_tick = '0'
elsif rising_edge(clk) then
      if (en = '1' and done_parallel_tick = '0' ) then
       -- spodnja zanka se izvede samo enkrat (dokler je done_parallel_tick ='0')
       -- v enem ciklu signala ure po tem, ko gre en na '1'!
       for i in slv_shifted_test_data'range loop
             crc_buf := (crc_buf(crc_buf'left - 1 downto 0) & slv_shifted_test_data(i)) xor
             -- pomik mesto crc_buf levo, vpis slv_shifted_test_data na lsb(crc_buf) in xor
             (slv_poly and (crc_out_parallel'range => crc_buf(crc_buf'left)));
             -- ce je MSB crc_buf enak '1', naredi xor s poly, sicer naredi xor z 0 (kar je original)
       end loop;
       done_parallel_tick <= '1'; -- pretvorba se je koncala z enkratno izvedbo zanke
     end if;
end if;
crc_out_parallel <= crc_buf;
end process;
done_parallel_tick_proc: process(clk)
begin
     -- done_parallel_tick signal je kombinacijski (izveden z and vrati - glej sintezo),
     -- zato ga moramo sinhronizirati s signalom ure, kar storimo z rising_edge
     -- ce ga ne, postane done_parallel_tick '1' pred zadnjim ciklom pretvorbe!
     if rising_edge(clk) then
      if (done_parallel_tick = '1') then
       report " Parallel CRC = " & slv_image(crc_out_parallel) & " (0x" & slv_hex_image(crc_out_parallel)& ")";
      end if;
     end if;
end process;
end;
```

Rezultat simulacije



Kako dosežemo izpis slv v konzoli ISE v dvojiški in šestnajstiški obliki besedila v tem smislu:

at 10 us(1): Note: Parallel CRC = 11110110 (0xF6).

Dvojiški izpis slv v datoteki testnih vrednosti

```
function slv_image(arg : std_logic_vector) return string is
constant arg_norm        : std_logic_vector(1 to arg'length) := arg;
constant center          : natural := 2; -- polozaj mesta vrednosti podobe bita je drugi
    znak: podoba bita je taka '1' (prvi je ', drugi 1 in tretji ')
variable just_the_number : character;  -- vrednost bita kot znak
variable bit_image       : string(1 to 3);        -- vmesni string za podobo bita
begin
    if (arg'length > 0) then
      bit_image       := std_logic'image( arg_norm(1) );  -- string bita (image) je v taki
    obliki: "'0'"
      just_the_number := bit_image(center);               -- iz stringa bita izlušèimo samo
    število: "0"
      return just_the_number                              -- vstavi trenutni bit slv
            & slv_image(arg_norm(2 to arg_norm'length)); -- rekurzivno obdelaj ostala mesta
    else
      return "";                      -- dokler je se kaj za pretvarjati
    end if;
end function slv_image;

function slv_image(arg : unsigned) return string is
begin
  return  slv_image(std_logic_vector(arg));
end function slv_image;
end crc_calc_serial_pkg;
```

Šestnajstiški izpis slv v datoteki testnih vrednosti

```
function slv_hex_image(arg: std_logic_vector) return string is
variable hexlen: integer;
variable longslv : std_logic_vector(67 downto 0) := (others => '0');
variable hex : string(1 to 16); variable hex_nibble : std_logic_vector(3 downto 0);
begin
    hexlen := (arg'left+1)/4;
          if (arg'left+1) mod 4 /= 0 then
                    hexlen := hexlen + 1;
          end if;
    longslv(arg'left downto 0) := arg;
    for i in (hexlen -1) downto 0 loop
         hex_nibble := longslv(((i*4) + 3) downto (i*4));
         case hex_nibble is
           when "0000" => hex(hexlen -i) := '0';  when "0001" => hex(hexlen -i) := '1';
           when "0010" => hex(hexlen -i) := '2';  when "0011" => hex(hexlen -i) := '3';
           when "0100" => hex(hexlen -i) := '4';  when "0101" => hex(hexlen -i) := '5';
           when "0110" => hex(hexlen -i) := '6';  when "0111" => hex(hexlen -i) := '7';
           when "1000" => hex(hexlen -i) := '8';  when "1001" => hex(hexlen -i) := '9';
           when "1010" => hex(hexlen -i) := 'A';  when "1011" => hex(hexlen -i) := 'B';
           when "1100" => hex(hexlen -i) := 'C';  when "1101" => hex(hexlen -i) := 'D';
           when "1110" => hex(hexlen -i) := 'E';  when "1111" => hex(hexlen -i) := 'F';
           when "ZZZZ" => hex(hexlen -i) := 'Z';  when "UUUU" => hex(hexlen -i) := 'U';
           when "XXXX" => hex(hexlen -i) := 'X';  when others => hex(hexlen -i) := '?';
         end case;
    end loop;
return hex(1 to hexlen);
end slv_hex_image;
```

Vzporedni izračun CRC v VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity crc_calculator is
     generic( POLY : integer := 16#04C11DB7#; crc_size : natural := 32 );
     port (clk,          -- signal ure
     rst,                -- asinhroni reset
     en : in STD_LOGIC;  -- postavi na '1' za zacetek izracuna
     d_in: in STD_LOGIC_VECTOR(crc_size - 1 downto 0);       -- vhodni podatek
     crc_out: out STD_LOGIC_VECTOR (crc_size - 1 downto 0)   -- CRC rezultat
    );
end crc_calculator;
architecture arch of crc_calculator is
begin
process (clk, rst) is
variable crc_buf : STD_LOGIC_VECTOR (crc_out'range):= (others => '0');
begin
if (rst = '1') then
     crc_buf := (others => '0');
elsif rising_edge(clk) then
     if (en = '1') then
     for i in d_in'reverse_range loop
            crc_buf := (crc_buf(crc_buf'left - 1 downto 0) & d_in(i)) xor
            -- pomik mesto crc_buf levo, vpis d_in na lsb(crc_buf) in xor
            (std_logic_vector(to_unsigned(POLY, crc_out'length)) and (crc_out'reverse_range =>
            crc_buf(crc_buf'left)));
            -- ce je MSB crc_buf enak '1', naredi xor s poly, sicer naredi xor z 0 (kar je original)
     end loop;
     end if;
end if;
crc_out<=crc_buf;
end process;
end arch;
```

Simulacija vzporednega izračuna CRC

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use IEEE.numeric_std.all;
ENTITY crc_calculator_tb IS
generic (    delay        : time := 5 ns;
POLY : integer :=  16#04C11DB7#;
crc_size : natural := 32 );
END crc_calculator_tb;
ARCHITECTURE arch OF crc_calculator_tb IS
COMPONENT crc_calculator is
     generic( POLY : integer :=  16#04C11DB7#;
              crc_size : natural := 32 );
port (clk, -- signal ure
rst, -- asinhroni reset
en : in STD_LOGIC;          -- postavi na '1' za zacetek
     izracuna
d_in: in STD_LOGIC_VECTOR(crc_size - 1 downto 0);     --
     vhodni podatek
crc_out: out STD_LOGIC_VECTOR (crc_size - 1 downto 0) -- CRC
     rezultat
);
end COMPONENT;
signal clk : STD_LOGIC := '0';           -- signal ure
signal rst : STD_LOGIC := '0';           -- asinhroni reset
signal d_tx_out : STD_LOGIC := '0';-- zaporedni vhod za
     podatke v oddajnik
signal crc_out : STD_LOGIC_VECTOR (crc_size - 1 downto 0);
-- CRC rezultat
constant clock_period : time := 200 ns;
constant clock_duty_cycle : real := 0.5;
constant clock_offset : time := 100 ns;
```

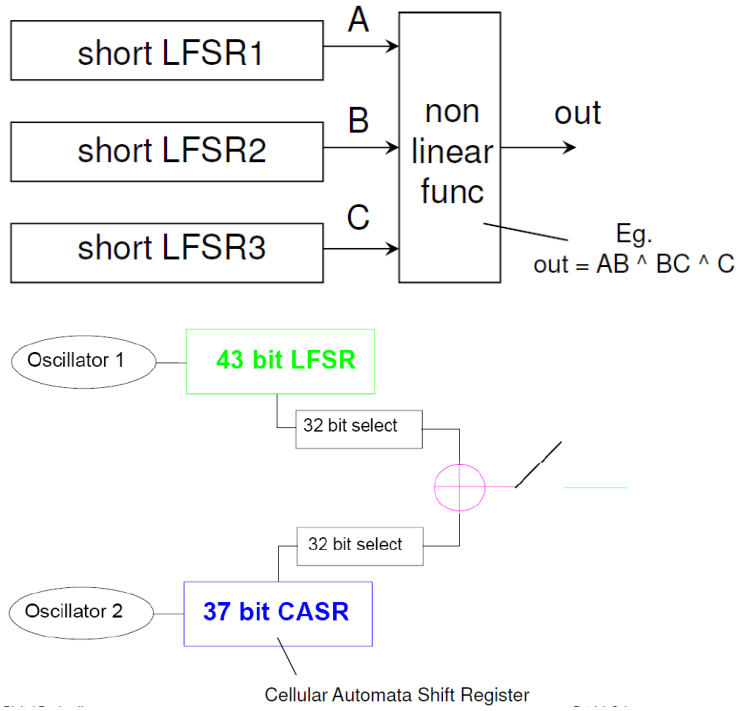
```
begin
process    -- proces, ki tvori signal ure clk
begin
wait for clock_offset;
clock_loop : loop
     clk <= '0';
     wait for (clock_period - (clock_period *
     clock_duty_cycle));
     clk <= '1';
     wait for (clock_period * clock_duty_cycle);
end loop clock_loop;
end process;

-- Povezovanje testnih enot oddajnika in sprejemnika
tx_uut: crc_calculator generic map (
     POLY => POLY, crc_size => crc_size)
port map (clk => clk,
     rst => rst, en => '1', d_in => (1|2|3 => '1', others =>
     '0'), crc_out => crc_out);
tx_proc: process
begin
     rst <= '0';
     wait for clock_offset;
     rst <= '1';
     for i in 0 to crc_size - 1 loop
     wait for clock_period; -- pomakni vsebino eno mesto
     end loop;
     wait;
end process;
d_tx_out <= crc_out(crc_out'left);      --extract serial msb

end;
```

Ali je LFSR uporaben za tvorbo naključnih števil?

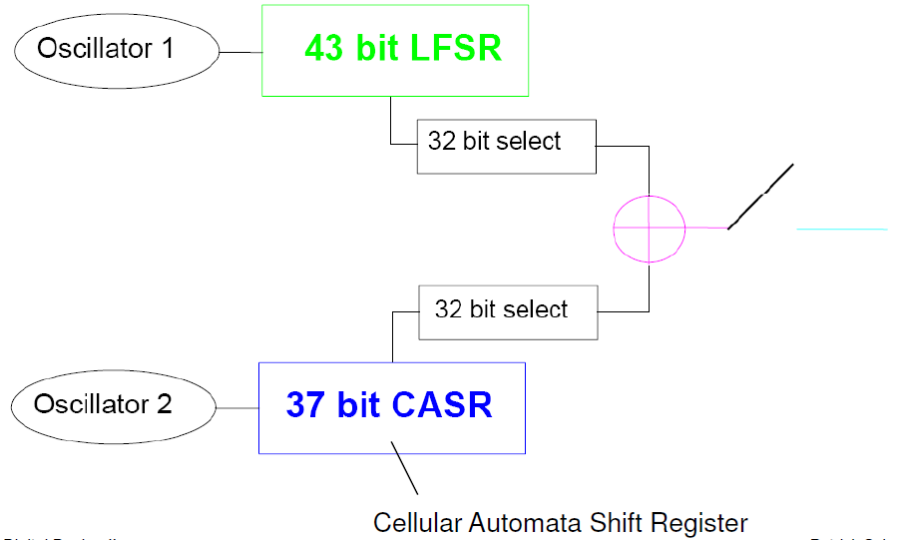
- Matematika (Berlekamp-Massey) sta pokazala, da za N-bitni LFSR z neznano povratno vezavo opazujemo samo 2N bitov, da lahko predvidimo naslednjega (2N+1)
- **LFSR so zato neprimerni za tvorbo naključnih pojavov**
- Kako potem?
- Podaljšamo LFSR in upamo na najboljše N=65536
  P(x) = x^N + ... + 1 (potratna rešitev 64k FF)
- Uporabimo nelinearni kombinacijski generator (ang. Non-linear Combination Generator)

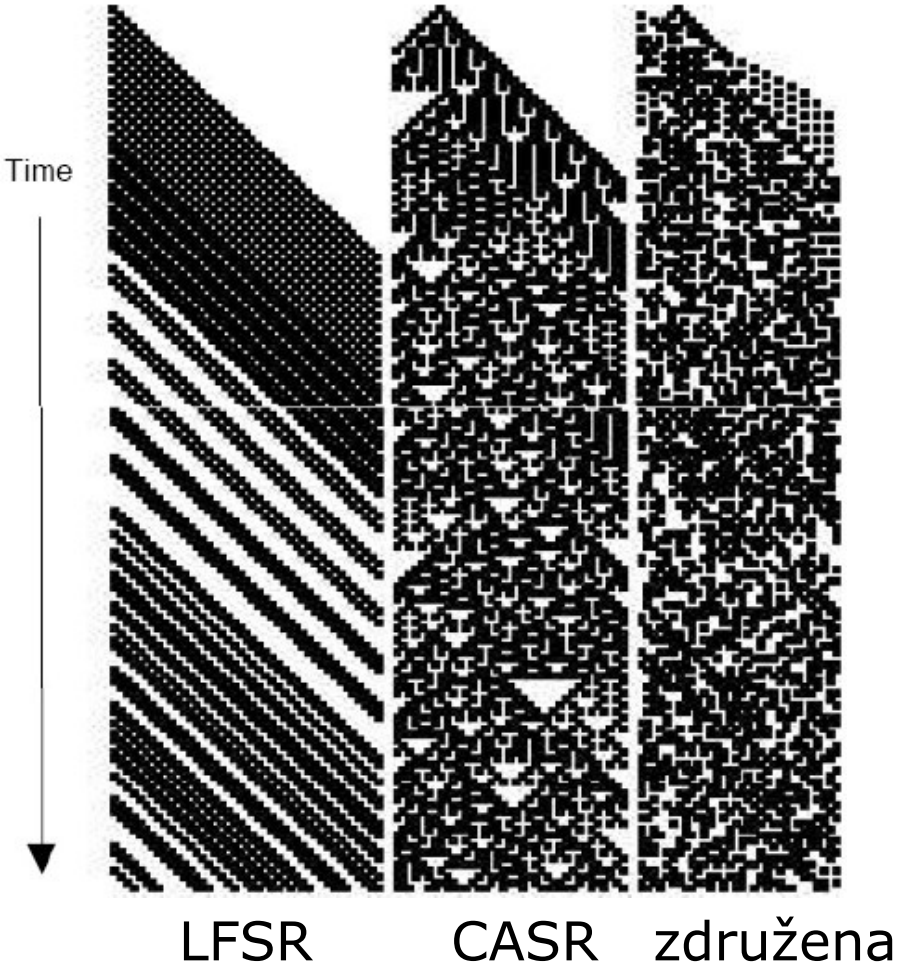
Nelinearni kombinacijski generator (Tkacik, 2002)

- 43-bitni LFSR: $x^{43}+ x^{41}+x^{20}+x+1$
- Največja dolžina: $2^{43}-1$
- 37-bitni avtomat s pomikalnimi registri
- Združuje trenutno in prejšnje stanje v novo stanje, podobno kot 37 prepletenih avtomatov.
- Največja dolžina: $2^{37}-1$



Nelinearni kombinacijski generator (Tkacik, 2002)





Načrtovanje digitalnih vezij

Flip-flopi, registri in števci:
Števci

Števci

- Števci so poseben primer aritmetičnih vezij, katerim se izhod povečuje/zmanjšuje za 1
- Števci štejejo število sprememb številnega signala (signala ure)!
- Vezja števcev se uporabljajo za več namenov
  - Štetje dogodkov (Counter)
  - Sekvenčno naslavljanje pomnilnika (DMA)
  - Časovnik:
    - Tvorba časovnih zakasnitev (timer)
    - Spremljanje pretečenega časa med dogodki (time delay)
- Modul štetja (M=8, pomeni da ima števec na izhodu vsa stanja od 0 do 7)
- Vrste števcev (asinhroni, sinhroni)

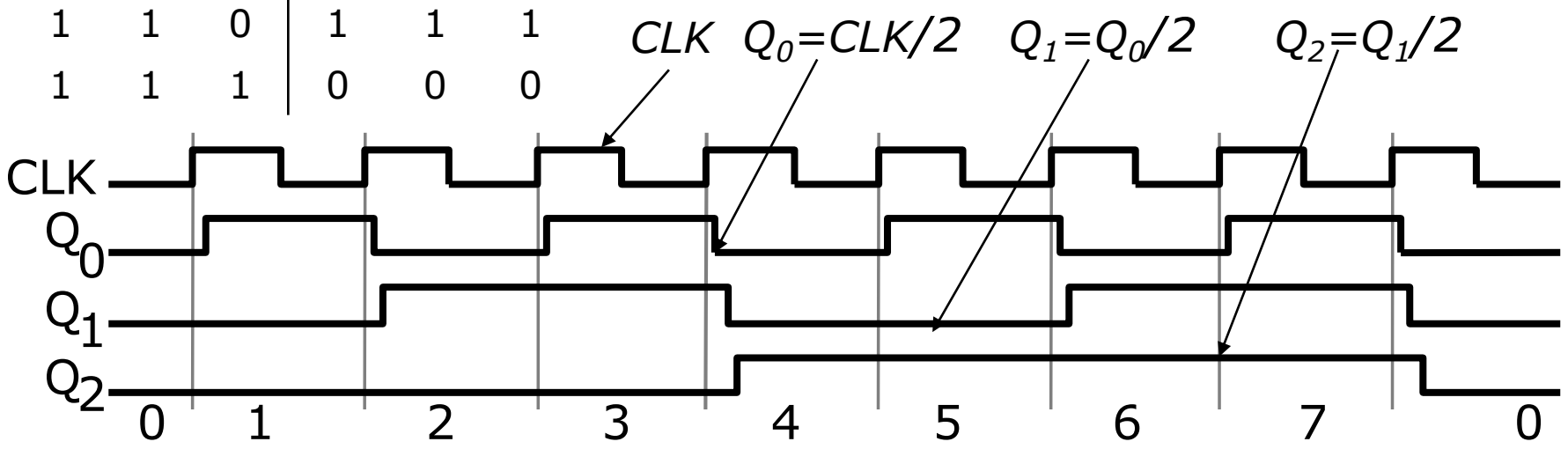
Asinhroni števci

- Asinhronim števcem pravimo tudi ***ripple*** števci. Zakaj?
  - *Vhodna* ura je vedno povezana samo na *en* FF
  - Signali ure za druge FF so izvedeni iz izhodov predhodnih FF
- Asinhrona oblika števca je počasna, zaradi kaskadne povezave signala ure – vezava povečuje zdrs !!!
  - Izvor signala ure valovi (ripple) od stopnje do stopnje
  - Efekt valovanja (ripple effect) je podoben kot pri ripple carry seštevalniku
- Dobra lastnost: so enostavni ker ne potrebujejo dodatne kombinacijske logike.

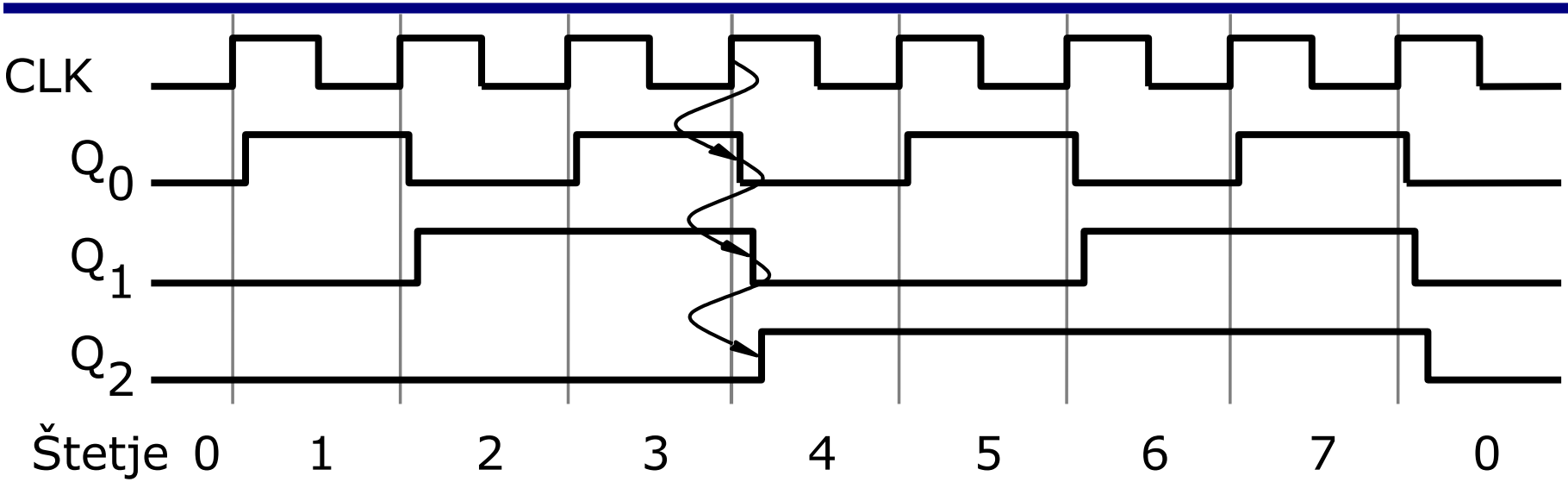
Asinhrono štetje s T-FF

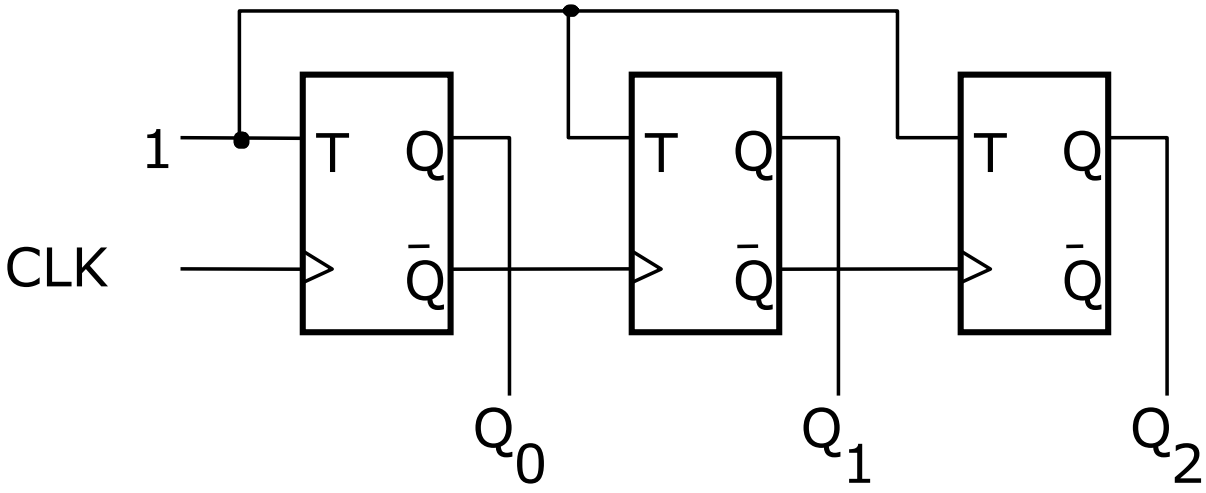
trenutno			*naslednje*		
Q_2	**Q_1**	**Q_0**	**Q_2**	**Q_1**	**Q_0**
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

T-FF že po svoji naravi delovanja deli frekvenco signala ure z 2, čim je vhod T='1'!

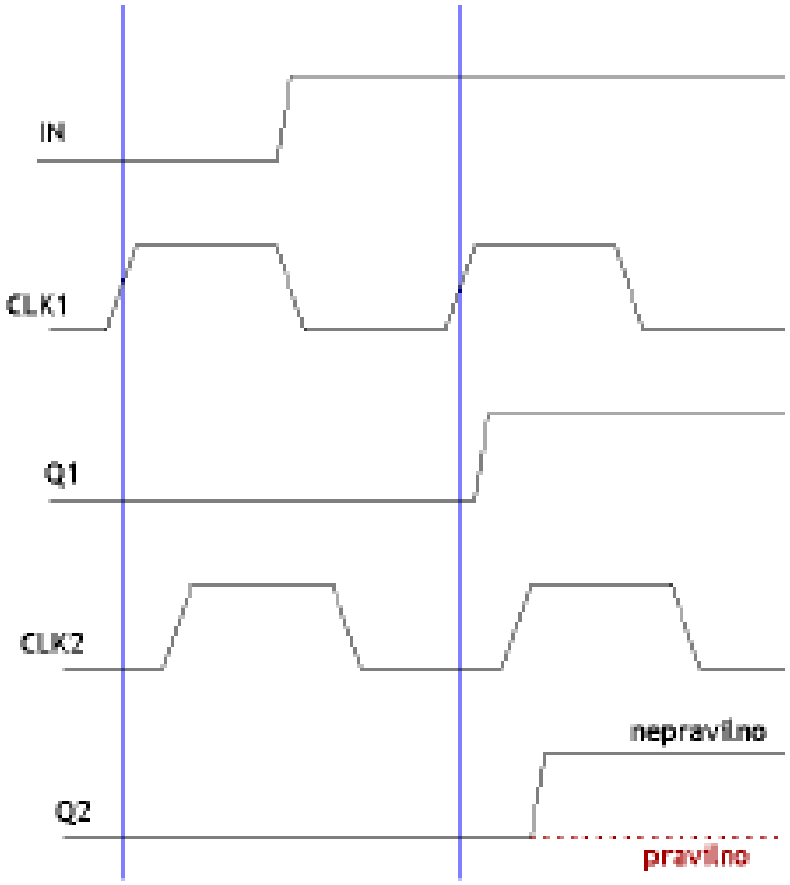


Naraščajoče štetje s T-FF





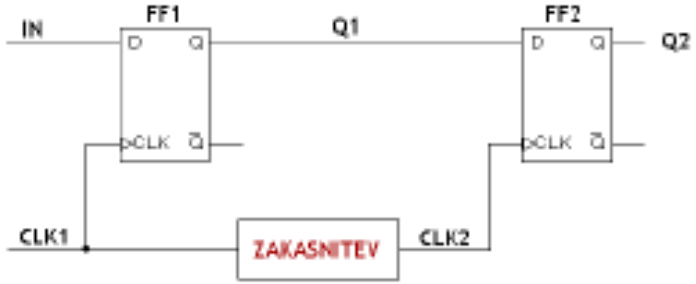
Zdrs signala ure (ang. clock skew)



V sinhronih digitalnih sistemih pričakujemo, da bodo vsi FF spremenili stanje v istem trenutku, torej da bo urina fronta do vsakega flip-flopa prispela v istem trenutku.

Temu ni tako. Pojavu, da se fronta signala ure zaradi zakasnitev pojavi pri dveh FF v dveh različnih časih se imenuje zdrs signala ure (clock skew).

Vstavljanje elementov v pot signala ure vodi do zdrsa signala ure.



T-FF v VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
library UNISIM;
use UNISIM.vcomponents.all; -- tukaj je definiran primitiv FDCE
entity t_ff is
    generic( INIT : bit := '0' -- zacetna vrednost ('0' se postavi ff na '0', '1' na '1')
    );
    port ( C,                       -- signal ure
           CE,                      -- signal omogocenja ure
           CLR,                     -- asinhrono brisanje
           T: in STD_LOGIC;         -- vhod t-ff
           Q, nQ: out STD_LOGIC     -- originalni in invertirani (nQ) izhod t-ff
    );
end t_ff;
architecture t_ff_xor of t_ff is
signal d_in : STD_LOGIC := '0' ; -- vhod d-ff
signal q_sig : STD_LOGIC := '0' ; -- izhod t-ff
begin
    FDCE_inst : FDCE
    generic map ( INIT => INIT)
    port map (
           Q => q_sig,              -- Podatkovni izhod
           C => C,                  -- Vhod za signal ure
           CE => CE,                -- signal omogocenja ure
           CLR => CLR,              -- asinhrono brisanje
           D => d_in                -- vhod d-ff
    );
    d_in <= q_sig xor T;            -- povratna vezava z xor vrati
    Q <= q_sig;                     -- povezovanje izhodnega signala
    nQ <= not q_sig;                -- realizacija invertiranega izhoda
end t_ff_xor;
```

Asinhroni števec navzgor v VHDL

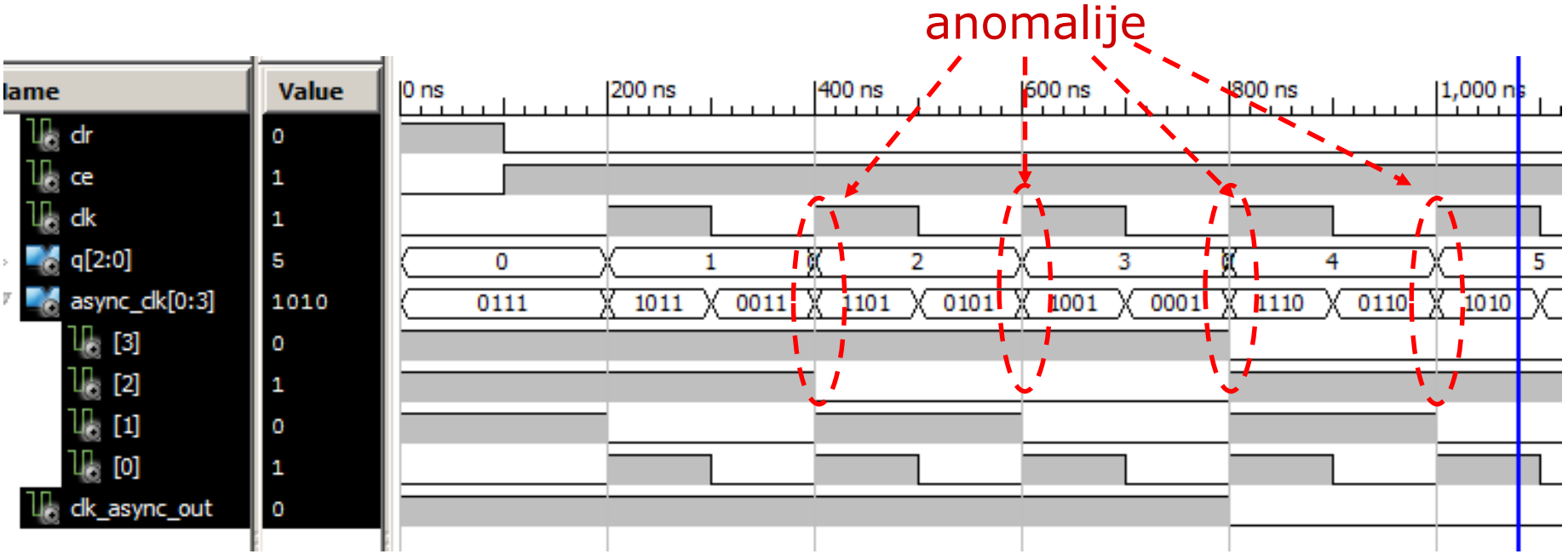
```
library IEEE;
use IEEE.std_logic_1164.all;
library UNISIM;
use UNISIM.vcomponents.all;
entity clock_skew is
generic( INIT : bit := '0';
-- zacetna vrednost ('0' se postavi ff na '0', '1' na '1')
ctr_size : natural := 4
);
port (clk,    -- signal ure
     clr,                  -- asinhrono brisanje
     ce : in STD_LOGIC;    -- signal za omogocenje stetja
     q : out STD_LOGIC_VECTOR(ctr_size - 1 downto 0);
     -- izhodno stetje asinhronega stevca
     clk_async_out: out STD_LOGIC -- izhodni signal ure
     zadnje stopnje asinhronega stevca
    );
end clock_skew;
architecture asyn_ctr_up of clock_skew is
component t_ff is
     generic( INIT : bit := '0' -- zacetna vrednost ('0' se
     postavi ff na '0', '1' na '1')
     );
     port (  C,            -- signal ure
     CE,     -- signal omogocenja ure
     CLR,    -- asinhrono brisanje
     T: in STD_LOGIC;      -- vhod t-ff
     Q, nQ: out STD_LOGIC  -- originalni in invertirani (nQ)
     izhod t-ff
    );
end component;

signal ctr : STD_LOGIC_VECTOR (ctr_size - 1 downto 0);
signal async_clk : STD_LOGIC_VECTOR (ctr_size downto 0);
```

```
begin
async_clk(0) <= clk;-- na prvo stopnjo se
    veže vhodni signal ure
G1 :      for i in 0 to ctr_size - 1
    generate
    begin
    t_ff_stage : t_ff
    generic map ( INIT => INIT)
    port map (
    C => async_clk(i),          -- signal ure
    CE => ce,  -- signal za omogocenje stetja
    CLR => clr, -- asinhrono brisanje
    T => '1', -- Podatkovni vhod t-ff
    Q => ctr(i), -- izhodni bit stetja
    asinhronega stevca
    nQ => async_clk(i + 1) -- invertirani
    izhod je signal ure naslednje stopnje
    );
    end generate;
    clk_async_out <=
    async_clk(async_clk'left);-- zadnji
    signal je izhodni signal ure
    q <= ctr;
end asyn_ctr_up;
```

Uro naslednje stopnje določa izhod prejšnje stopnje!
Zakaj je to narobe?

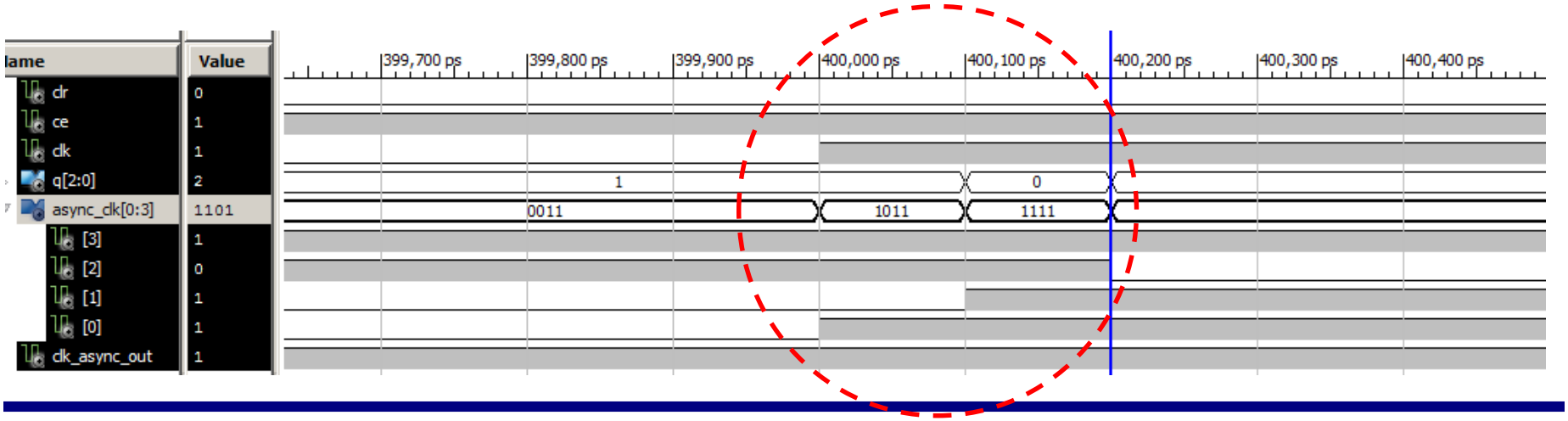
Rezultat simulacije asinhronega števca navzgor v VHDL

- Pri prehodih med številkami opazimo anomalije

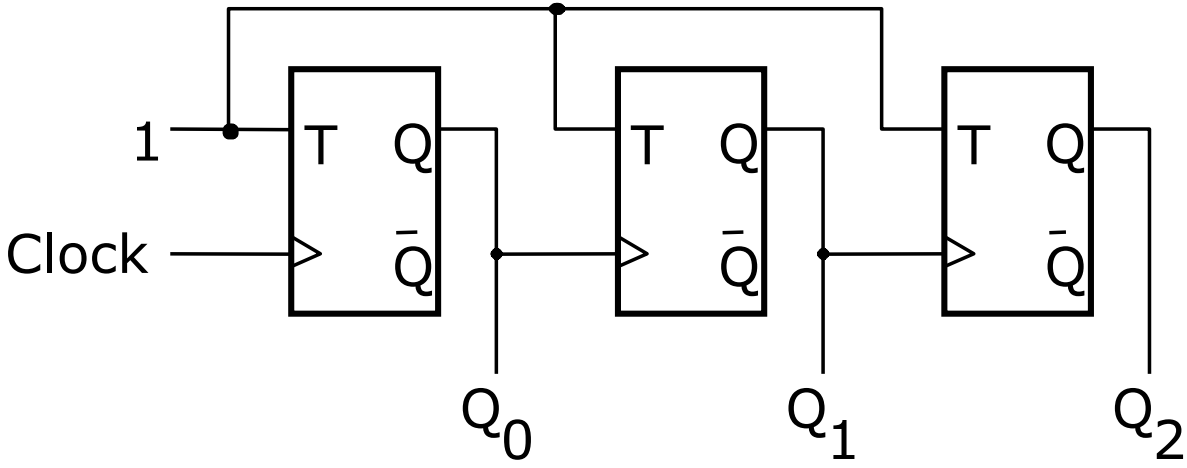


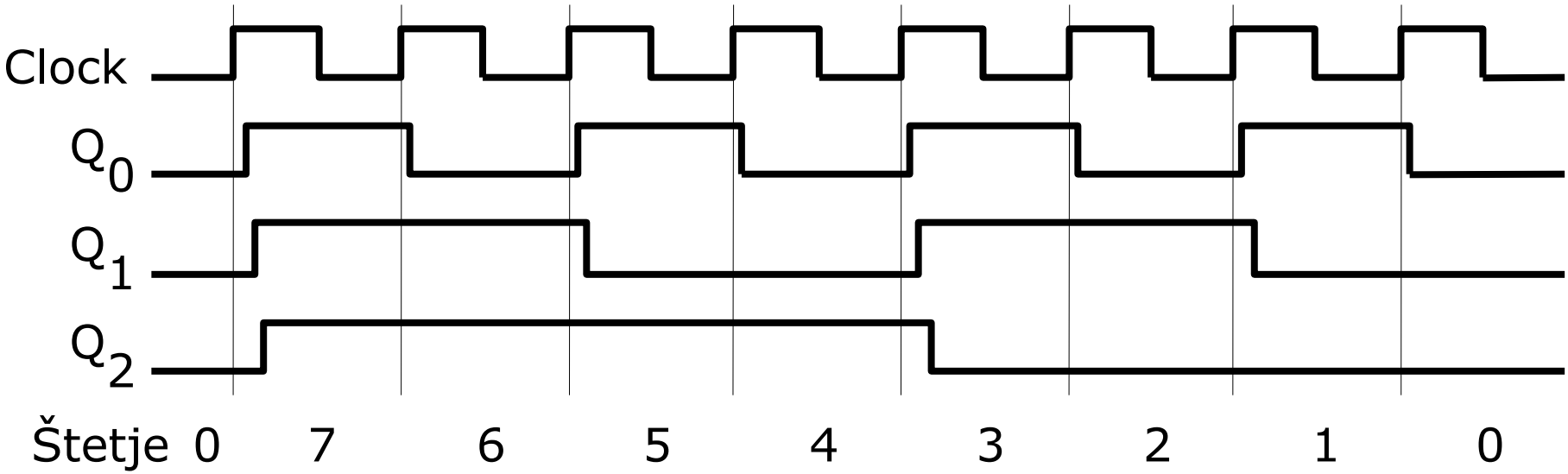
Analiza anomalij

- Na prehodu štetja 1→2 dejansko ob clk pride še do sprememb vseh internih signalov ure.
- Signali ure stopenj asinhronega števca so zakasnjeni za zakasnitev širjenja preko T-FF.
- Če uro pospešimo, se lahko zgodi, da preostale stopnje števca niti ne preidejo v naslednje stanje in števec ne šteje več pravilno.
- Večbitni asinhroni števci so zato za realizacijo v VHDL neprimerni, ker so počasni in kršijo načelo sinhronega načrtovanja vezij.

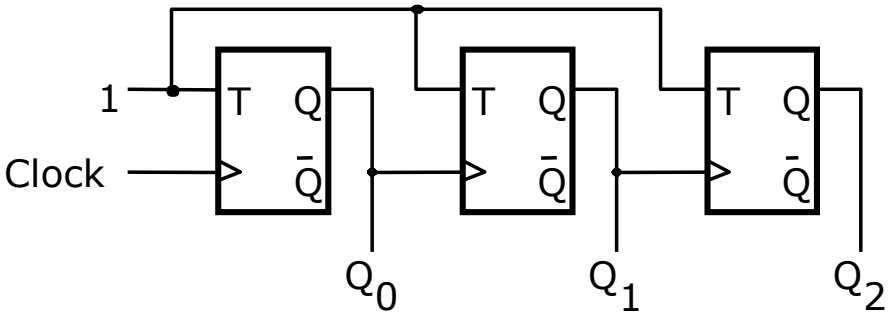


Padajoče štetje s T-FF



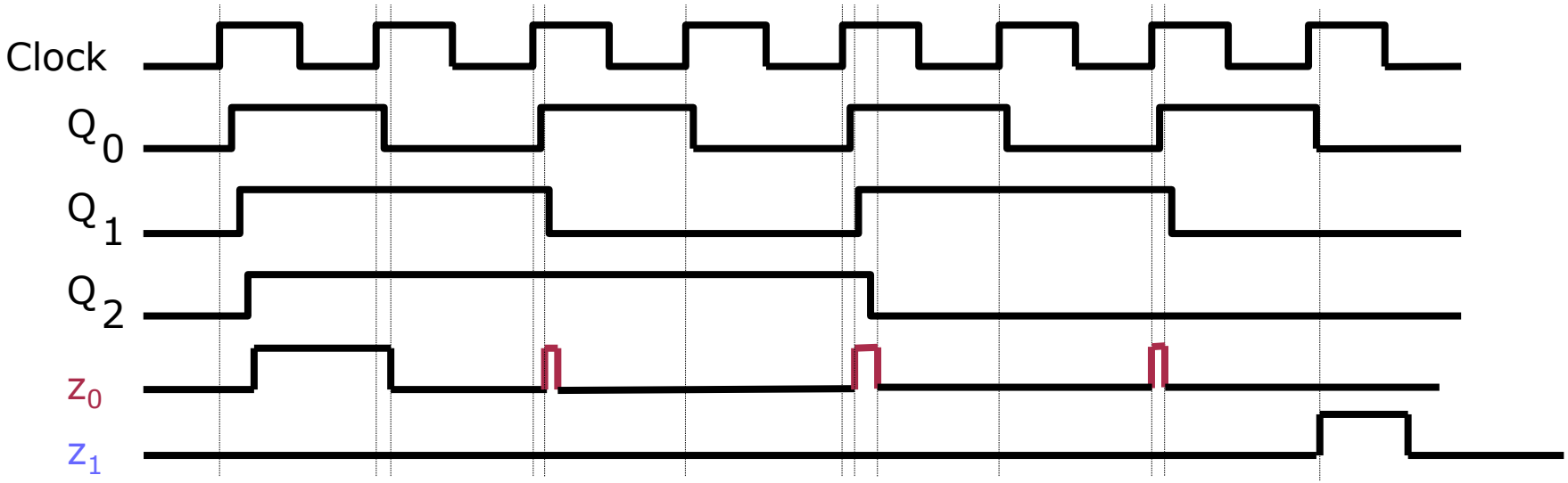


Problem detekcije izbranega stanja pri asinhronih števcih



$z_0 = Q_2 \cdot Q_1 \cdot Q_0$

$z_1 = Q_2' \cdot Q_1' \cdot Q_0'$



Na signalu z_0 se pojavijo motnje ➔ trava, zato je signal uporaben samo kot statičen (torej neuporaben).

Sinhroni števec navzgor s T-FF

trenutno			*naslednje*					
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	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

T_2:

Q_1 \ Q_2				
	0	1	1	0
	0	0	0	0

Q_0

$T_0=1$

$T_1=Q_0$

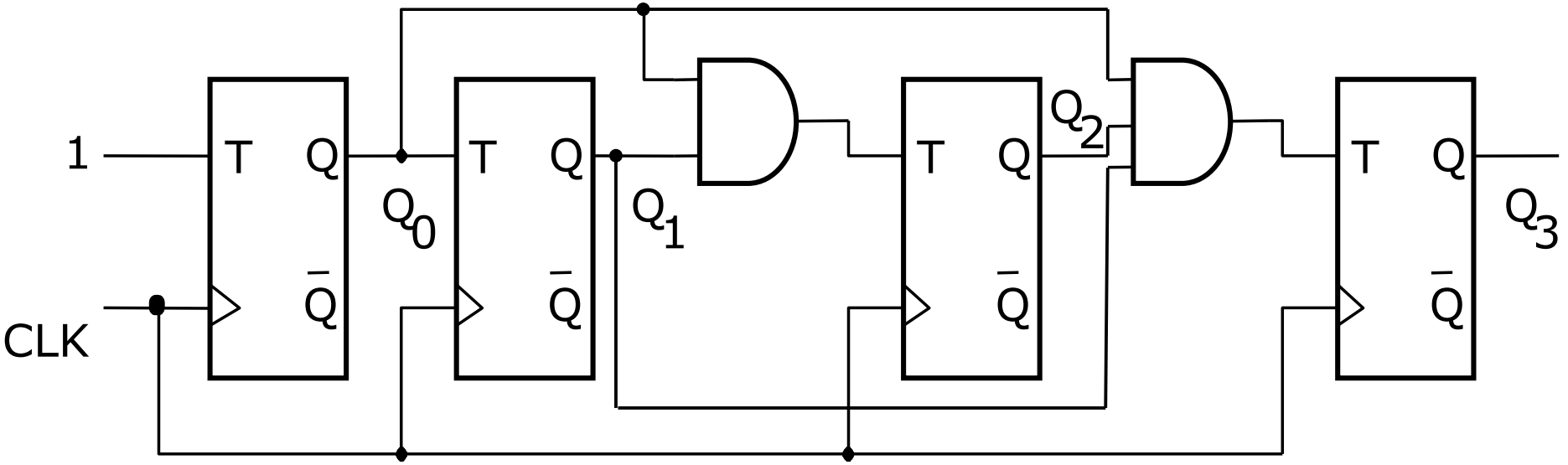
$T_2=Q_1 \cdot Q_0$

$T_n=Q_{n-1} \cdot Q_{n-1} \cdot \ldots \cdot Q_1 \cdot Q_0$

Enonivojska realizacija sinhronega števca navzgor s T-FF

$T_0=1$ $T_1=Q_0$ $T_2=Q_1 \cdot Q_0$ $T_3=Q_2 \cdot Q_1 \cdot Q_0$

$T_0=1$ $T_1=Q_0 \cdot T_0$ $T_2=Q_1 \cdot T_1$ $T_3=Q_2 \cdot T_2$

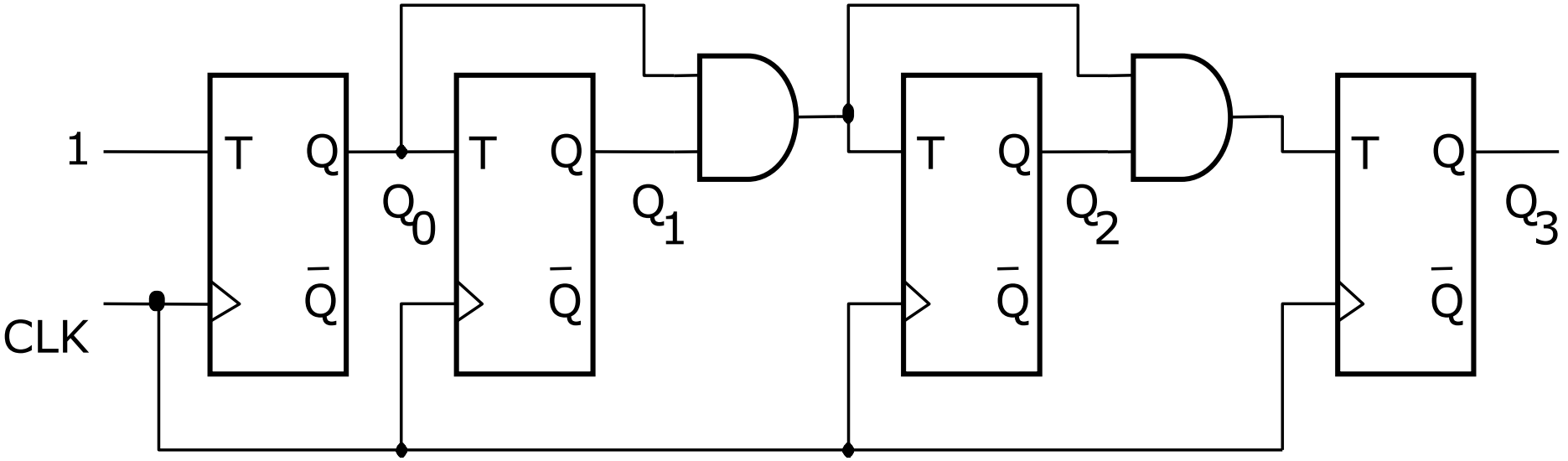


4-bitni sinhronski števec (naraščajoče štetje)
Izhodi števca so $Q_3Q_2Q_1Q_0$.
Problem: FAN-IN za n-vhodna AND vrata, zato števec raje realiziramo večnivojsko.

Večnivojska realizacija sinhronega števca navzgor s T-FF

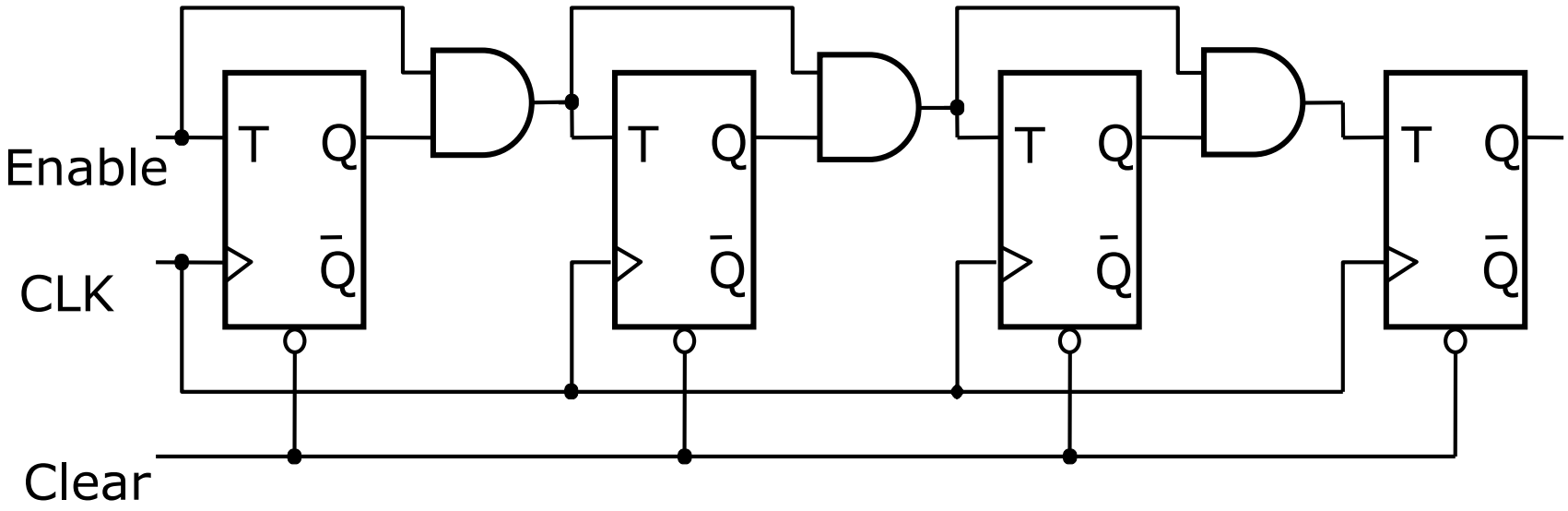
$T_0=1$ $T_1=Q_0$ $T_2=Q_1 \cdot Q_0$ $T_3=Q_2 \cdot Q_1 \cdot Q_0$

$T_0=1$ $T_1=Q_0 \cdot T_0$ $T_2=Q_1 \cdot T_1$ $T_3=Q_2 \cdot T_2$



- Vsi FF zamenjajo stanja z eno zakasnitvijo širjenja vrat (propagation delay) ➔ Ta realizacija ne upočasnjuje števca.
- Hitrost štetja je omejena s hitrostjo širjenja sprememb Q_0, katerega sprememba se mora širiti preko vseh stopenj.
- Čas širjenja ne sme biti daljši od ene periode signala ure.
- Problem so veliki moduli štetja

Omogočanje štetja in brisanje števca



Sinhroni števci z D-FF

Q_3	Q_2	Q_1	Q_0	Q_3	Q_2	Q_1	Q_0	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	1	0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	1	0	0	1	1
0	0	1	1	0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	1	0	1	0	1
0	1	0	1	0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	1	0	1	1	1
0	1	1	1	1	0	0	0	1	0	0	0
1	0	0	0	1	0	0	1	1	0	0	1
1	0	0	1	1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	1	1	0	1	1
1	0	1	1	1	1	0	0	1	1	0	0
1	1	0	0	1	1	0	1	1	1	0	1
1	1	0	1	1	1	1	0	1	1	1	0
1	1	1	0	1	1	1	1	1	1	1	1
1	1	1	1	0	0	0	0	0	0	0	0

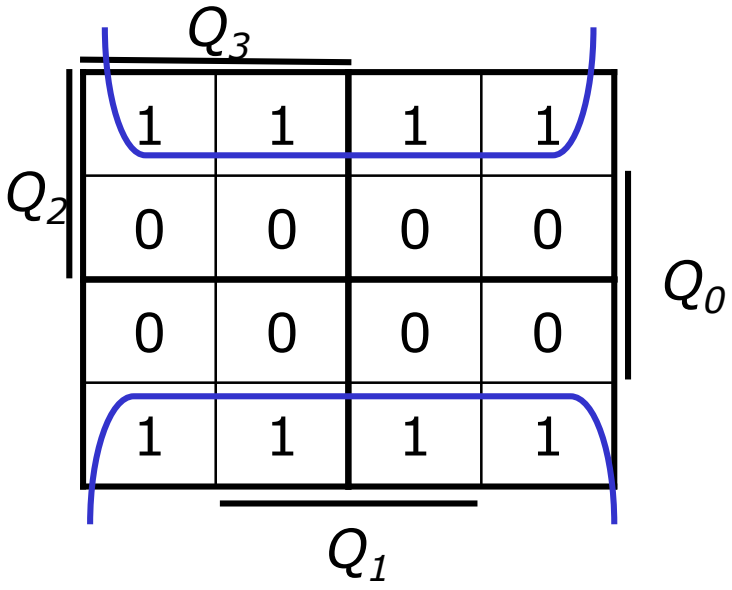
$D_0 = Q_0'$

$D_1 = Q_1 \oplus Q_0$

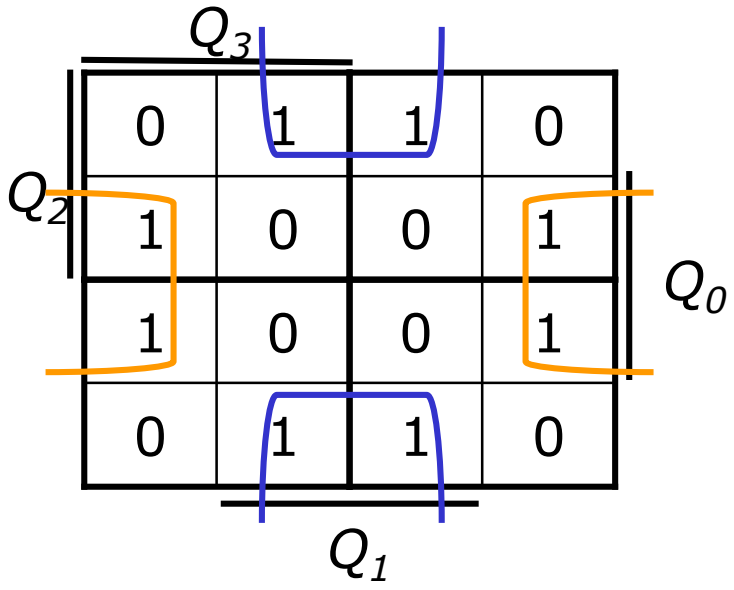
D_2= ? ➔ Veitchev diagram

D_3= ? ➔ Veitchev diagram

Sinhroni števci z D-FF



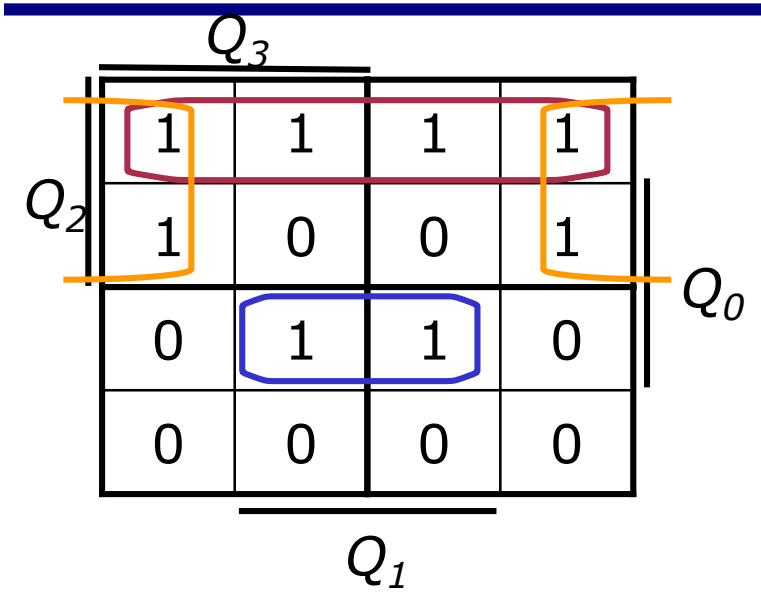
$D_0=Q_0'$



$D_1=Q_1\cdot Q_0'+Q_1'\cdot Q_0$

$D_1=Q_1 \oplus Q_0$

Sinhroni števci z D-FF

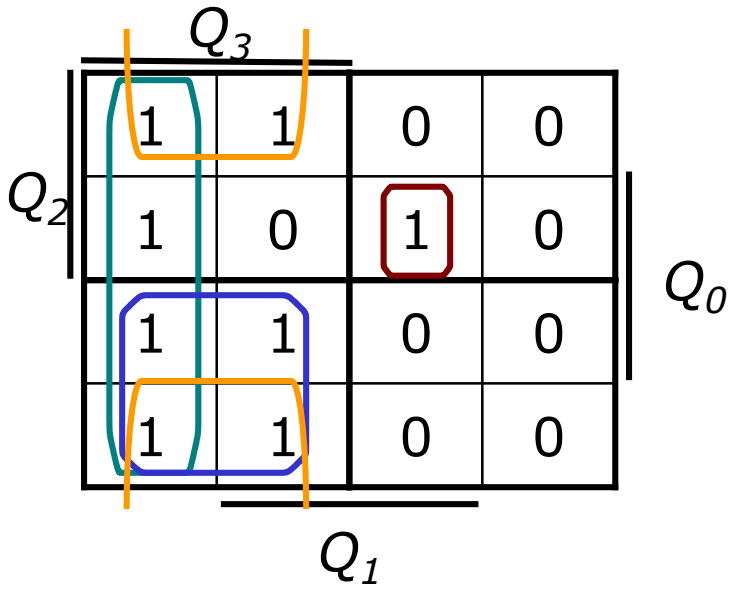


$D_2=Q_2\cdot Q_0'+Q_2\cdot Q_1'+Q_2'\cdot Q_1\cdot Q_0$

$D_2=Q_2\cdot (Q_0'+Q_1')+Q_2'\cdot Q_1\cdot Q_0$

$D_2=Q_2\cdot (Q_0\cdot Q_1)'+Q_2'\cdot (Q_1\cdot Q_0)$

$D_2= Q_2 \oplus Q_1\cdot Q_0$



$D_3=Q_3\cdot Q_1'+Q_3\cdot Q_2'+Q_3\cdot Q_0'+Q_3'Q_2Q_1Q_0$

$D_3=Q_3\cdot (Q_1'+Q_2'+Q_0')+Q_3'\cdot Q_2\cdot Q_1\cdot Q_0$

$D_3=Q_3\cdot (Q_1\cdot Q_2\cdot Q_0)'+Q_3'\cdot (Q_2\cdot Q_1\cdot Q_0)$

$D_3= Q_3 \oplus Q_2\cdot Q_1\cdot Q_0$

Sinhroni števci z D-FF

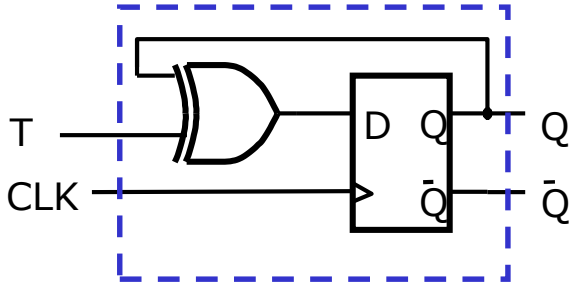
- 4-bitni števec gor šteje v sekvenci 0, 1, 2,…, 15, 0, 1…
- Štetje je predstavljeno na 4 izhodih FF $Q_3Q_2Q_1Q_0$
- D vhodi so podani kot:

$$D_0 = Q_0 \oplus \text{Enable}$$

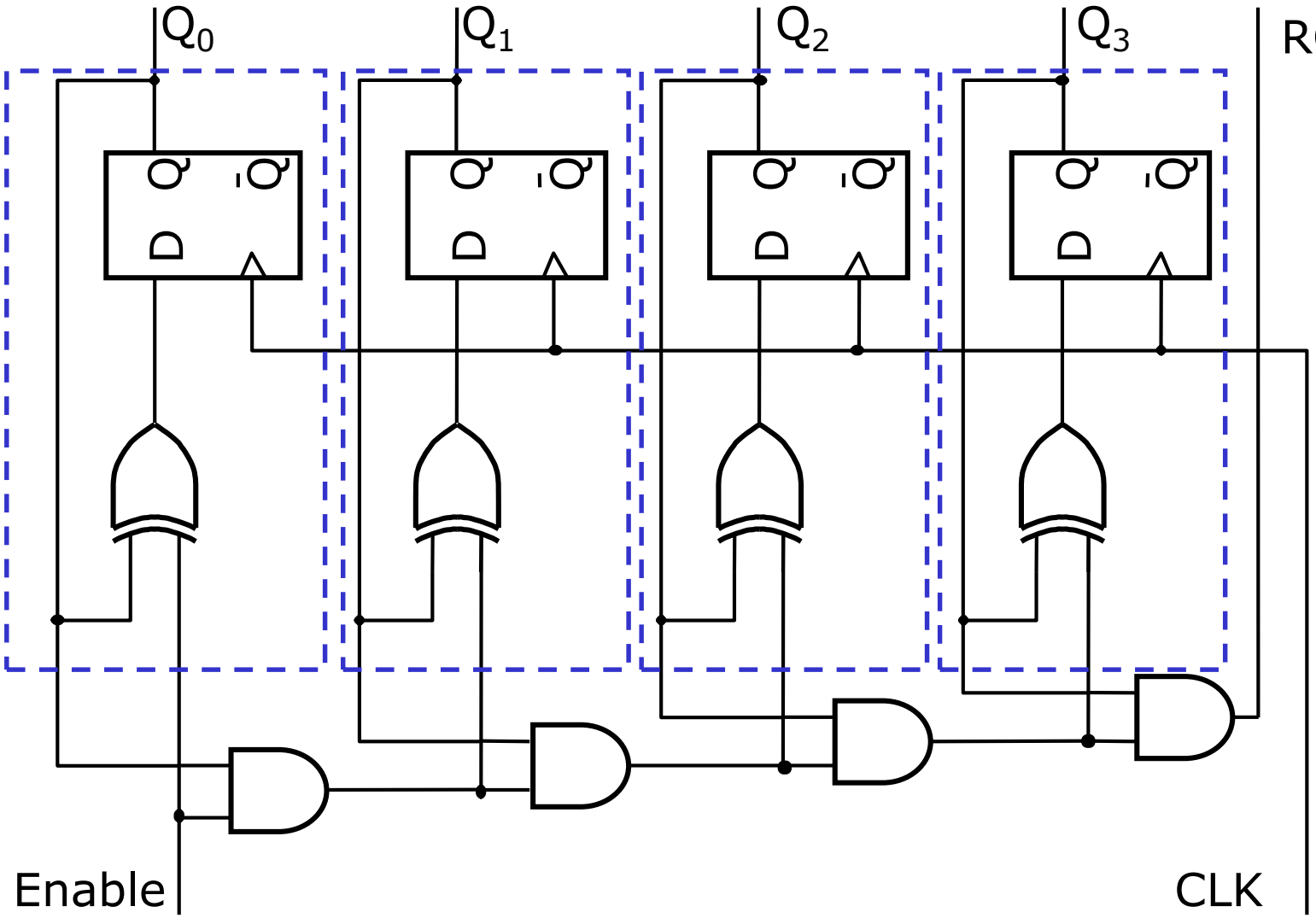
$$D_1 = Q_1 \oplus Q_0 \cdot \text{Enable}$$

$$D_2 = Q_2 \oplus Q_1 \cdot Q_0 \cdot \text{Enable}$$

$$D_3 = Q_3 \oplus Q_2 \cdot Q_1 \cdot Q_0 \cdot \text{Enable}$$



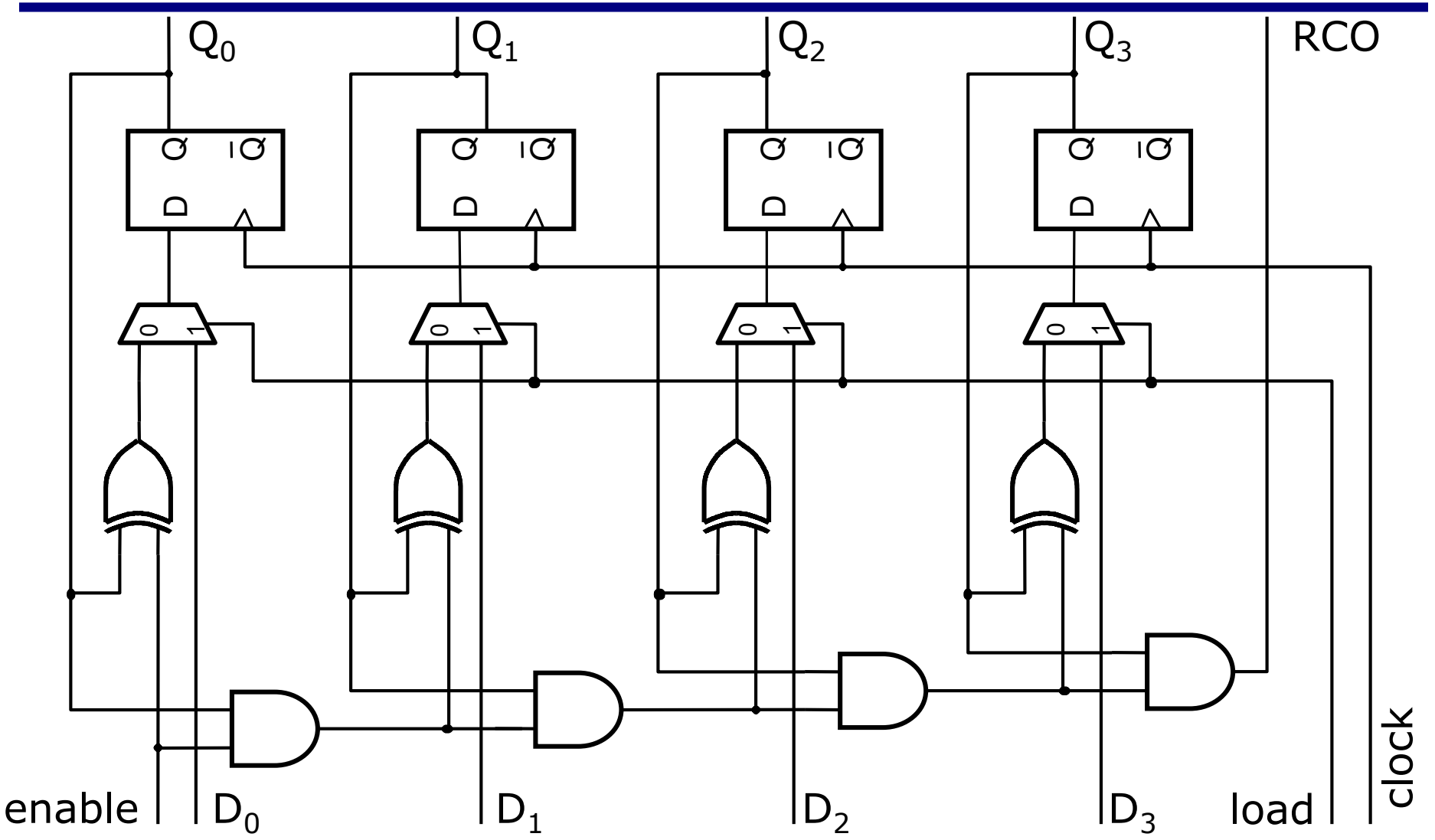
4 bitni števec navzgor z D-FF



Števci z vzporednim vpisom

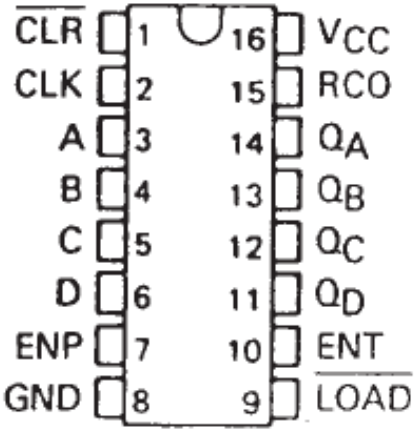
- Običajno števci štejejo od vrednosti 0 dalje
  - Postavljanje števca na 0 NE IZVAJAMO z uporabo *clear* vhoda na FF!
- Štejemo pa lahko tudi *od* vrednosti, ki ni 0.
- Dodati moramo vezje, ki služi **vzporednemu nalaganju** (*parallel load*)
- Kontrolni vhod, *load*, uporabljamo za določanje **načina** (*mode*) delovanja
  - Load=0, štej
  - Load=1, naloži vzporedno vrednost v števec

Števci z vzporednim vpisom



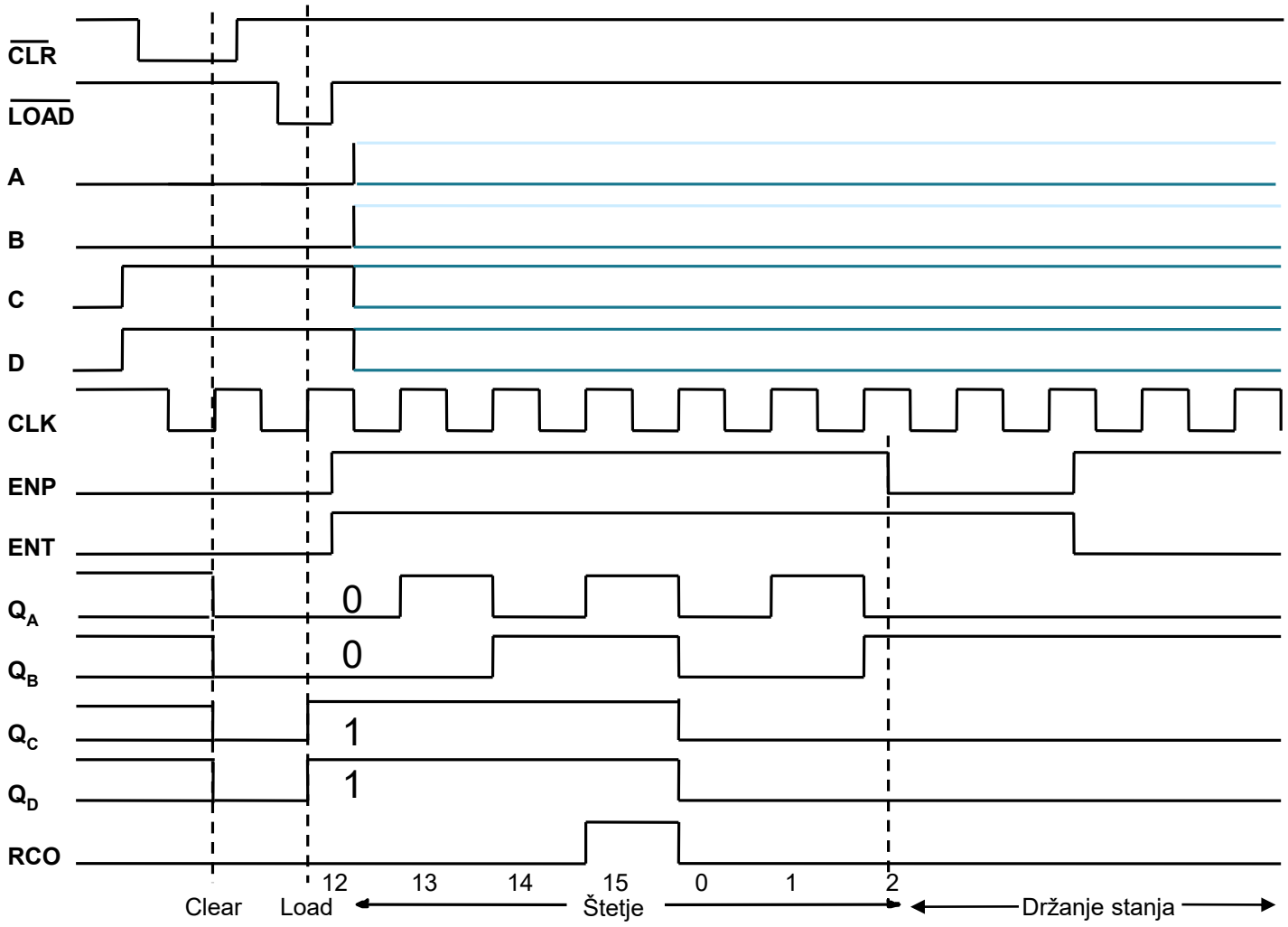
4-bitni sinhroni MSI števec 74163

- MSI števec 74163 ➔ 4-bitni sinhroni števec z vzporednim vpisom :
  - CLR='0' ➔ zbriše števec
    (iz vhodov DCBA na izhode QD, QC, QB, QA)
  - LOAD ='0' ➔ naloži stanje
    (iz vhodov DCBA na izhode QD, QC, QB, QA)
  - ENP in ENT ='1' ➔ šteje
  - Če ni pozitivnega roba CLK ➔ ohranja stanje
  - RCO postane '1'
    pri prehodu iz stanja 1111➔0000

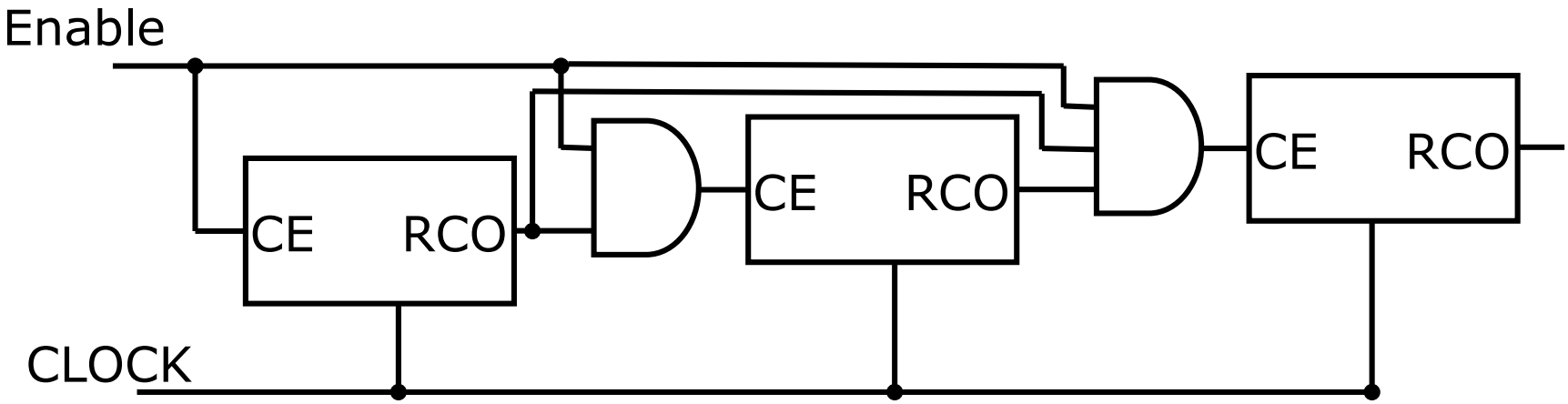


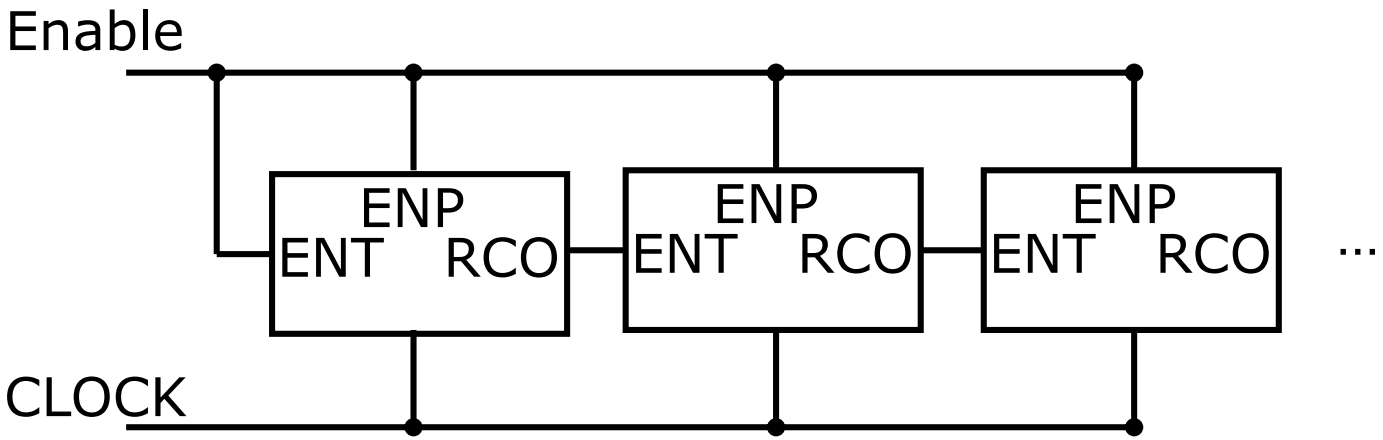
CLR	LOAD	ENP	ENT	CLK	A	B	C	D	QA	QB	QC	QD	RCO
0	X	X	X	POS	X	X	X	X	0	0	0	0	0
1	0	0	0	POS	X	X	X	X	A	B	C	D	*1
1	1	1	1	POS	X	X	X	X	števec šteje				*1
1	1	1	X	X	X	X	X	X	QA0	QB0	QC0	QD0	*1
1	1	X	1	X	X	X	X	X	QA0	QB0	QC0	QD0	*1

74163: Časovni diagram

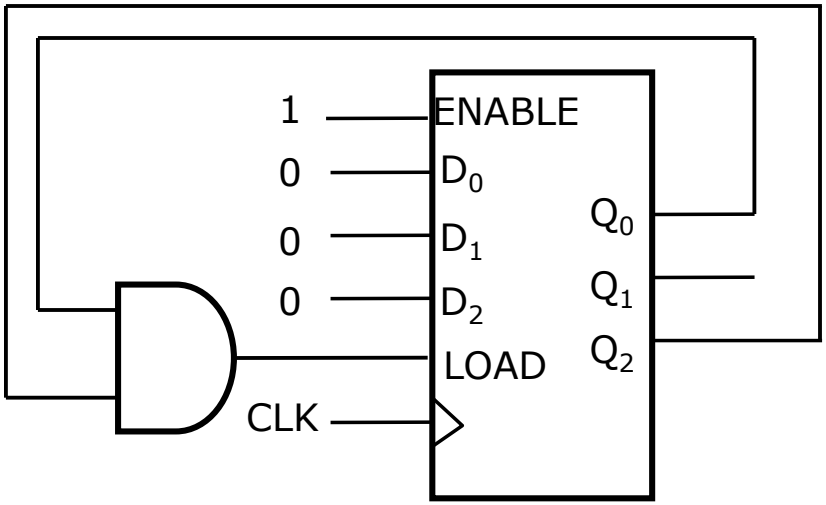


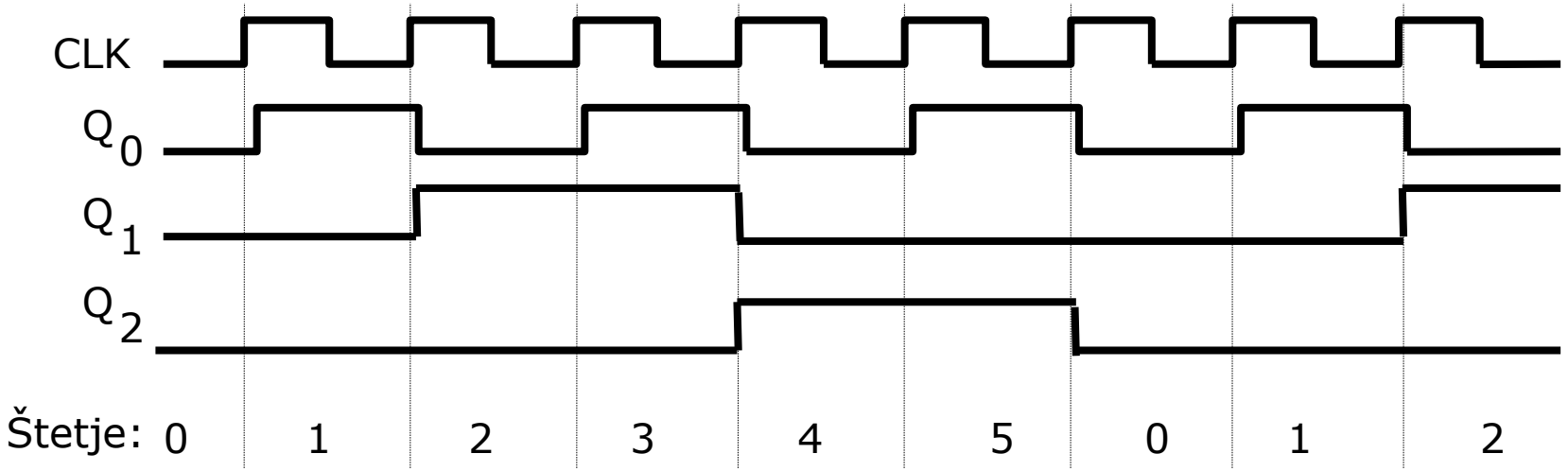
Kaskadna vezava MSI števcev





Štetje po modulu, ki ni 2^n





4-bitni števec gor z load signalom

```
ENTITY upcount IS
    PORT ( R : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
       Clock, nRst, Load : IN STD_LOGIC;
       Count : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)) ;
END upcount ;
ARCHITECTURE arch OF upcount IS
BEGIN
    PROCESS ( Clock, nRst )
    BEGIN
      IF nRst = '0' THEN
       Q <= "0000";
      ELSIF (Clock'EVENT AND Clock = '1') THEN
       IF Load = '1' THEN
              Q <= R;
       ELSE
              Q <= Q + 1;
       END IF;
      END IF;
    END PROCESS;
    Count <= Q;
END arch;
```

74163 v VHDL

```
entity counter163 is
    port( nLOAD, nCLEAR, ENP, ENT, clk :   in std_logic;
        D: in std_logic_vector(3 downto 0);
        Q: out std_logic_vector(3 downto 0);
        RCO :   out std_logic);
end entity counter163;
architecture arch of counter163 is
begin
process (clk, nCLEAR, nLOAD)
begin
    if (clk'EVENT AND clk = '1') then
        if ( nCLEAR = '0' ) then  Q <= "0000"; -- asinhroni reset
        elsif ( nLOAD = '0' ) then    Q <= D; -- vzporedno nalaganje
        elsif ( ENP = '1' and ENT = '1' ) then Q <= Q + 1;   -- stej
        end if;
    end if;
   RCO <= Q[3] and Q[2] and Q[1] and Q[0] and ENT; --tvorba RCO
end process;
end architecture arch;
```

Kaskada sinhronih števnikov za modul 3

trenutno		*naslednje*					
Q_1	Q_0	Q_1	Q_0	J_1	K_1	J_0	K_0
0	0	0	1	0	X	1	X
0	1	1	0	1	X	X	1
1	0	1	1	X	1	0	X
1	1	0	0	?	?	?	?

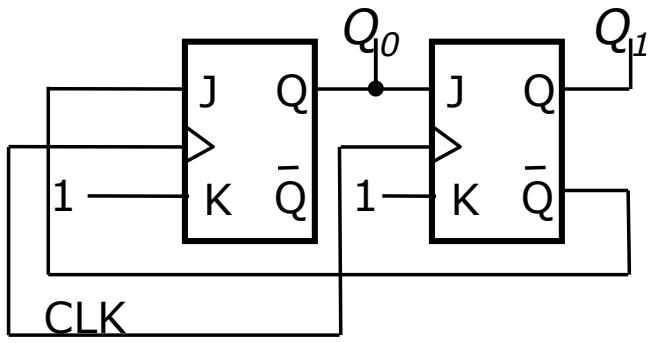
Q_1	Q_0	Q_1	Q_0	J_1	K_1	J_0	K_0
0	0	0	1	0	1	1	1
0	1	1	0	1	1	1	1
1	0	1	1	0	1	0	1
1	1	0	0	1	1	0	1

$J_1=Q_0 \quad K_1=1$

$J_0=Q_1' \quad K_0=1$

Z '?' vplivamo na števno zaporedje,
Z 'x' ne vplivamo na števno zaporedje.

Problem: Q_1Q_0="11" →"11"



Kaskada sinhronih števnikov za modul 5

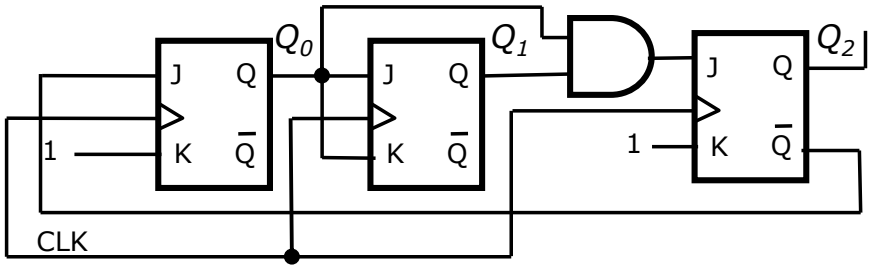
trenutno			*naslednje*								
Q_2	Q_1	Q_0	Q_2	Q_1	Q_0	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	0	1	1	0	X	X	0	1	X
0	1	1	1	0	0	1	X	X	1	X	1
1	0	0	0	0	0	X	1	0	X	0	X

$J_2=Q_1Q_0$ $K_2=1$

$J_1=Q_0$ $K_1=Q_0$

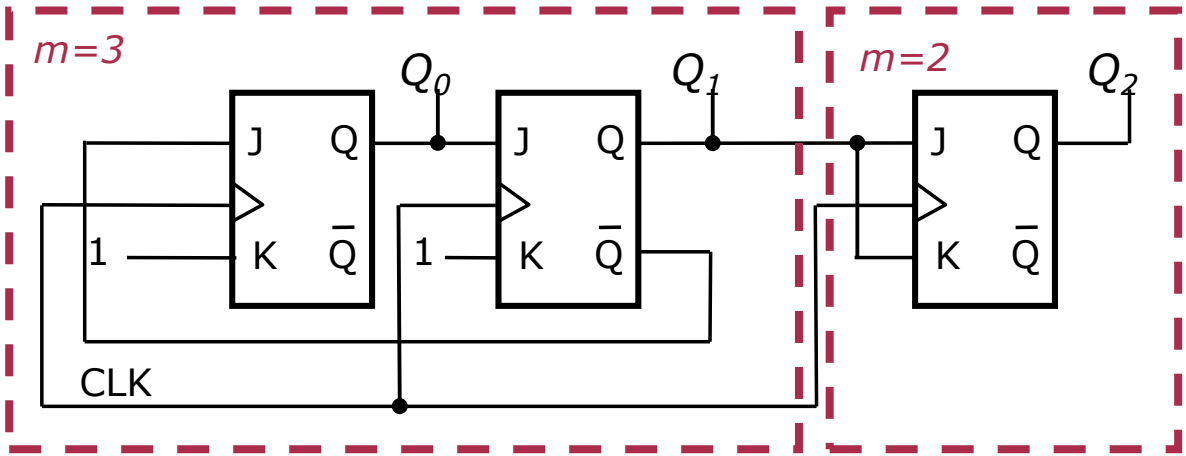
$J_0=Q_2'$ $K_0=1$

Q_2	Q_1	Q_0	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	1	0	0	1	1
0	0	1	0	1	1	1	1	1
0	1	0	0	1	0	0	1	1
0	1	1	1	1	1	1	1	1
1	0	0	0	1	0	0	0	1



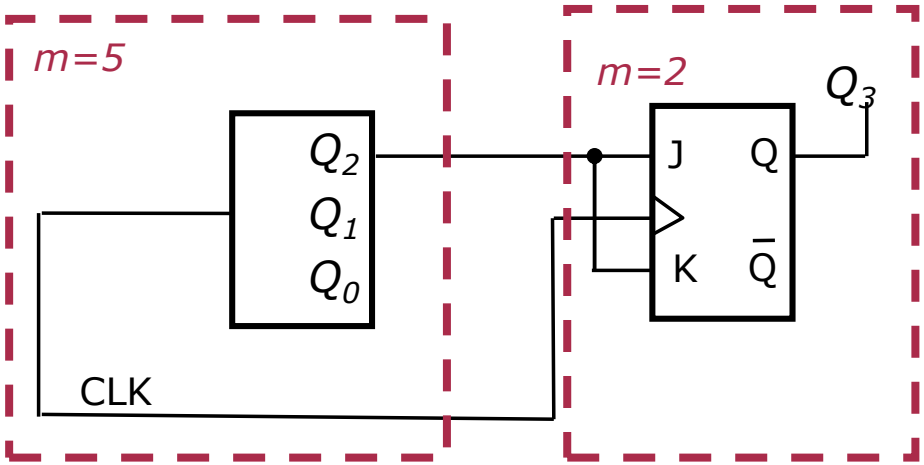
Kaskada sinhronih števnikov za modul 6

	trenutno		*naslednje*		
Q_2	**Q_1**	**Q_0**	**Q_2**	**Q_1**	**Q_0**
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	0	0	0



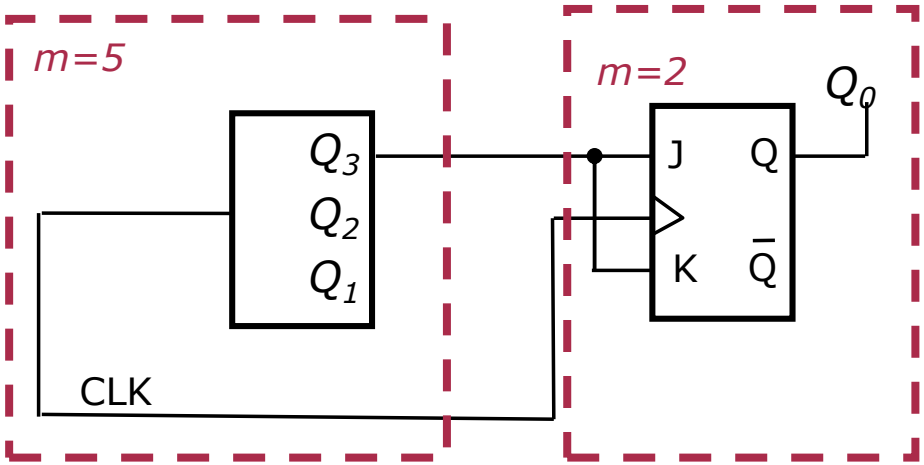
Kaskada sinhronih števnikov za modul 10

	trenutno			*naslednje*			
Q_3	Q_2	Q_1	Q_0	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	1	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	1	1	0	0
1	1	0	0	0	0	0	0



Biternarni/bikvinarni števec

		trenutno		*naslednje*			
Q_3	Q_2	Q_1	Q_0	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0	0	1	0
0	0	1	0	0	1	0	0
0	1	0	0	0	1	1	0
0	1	1	0	1	0	0	0
1	0	0	0	0	0	0	1
0	0	0	1	0	0	1	1
0	0	1	1	0	1	0	1
0	1	0	1	0	1	1	1
0	1	1	1	1	0	0	1
1	0	0	1	0	0	0	0

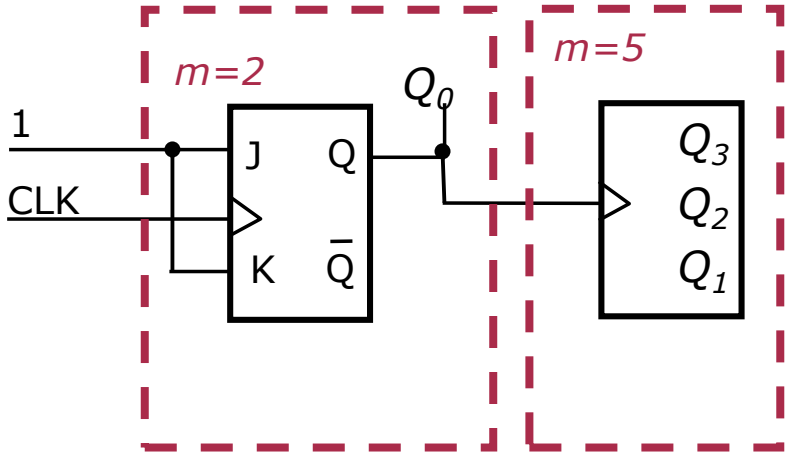


Bikvinarni (m=10):
Šteje najprej po sodih (0,2,4,6,8), nato po lihih (1,3,5,7,9)

Biternarni (m=6)
Šteje najprej po sodih (0,2,4), nato po lihih (1,3,5)

Sinhrono-asinhroni števec

Q_3	Q_2	Q_1	Q_0	števni signal
0	0	0	0	
0	0	1	1	
0	0	1	0	
0	1	0	1	
0	1	0	0	
0	1	1	1	
0	1	1	0	
1	0	0	1	
1	0	0	0	
0	0	0	1	



Ima isti problem kot vsa asinhrona vezja – z velikim modulom štetja bi postal počasen.

Števec m=5 je v osnovi sinhron, le vezan je asinhrono. Prožen je namreč na prednji rob asinhronega števnega signala (⌐‾|), dobljenega iz Q_0

Načrtovanje digitalnih vezij

Načrtovanje FSM
z uporabo
CAD orodij

Sinteza FSM z uporabo CAD orodij

- VHDL vsebuje nekaj pristopov za načrtovanje FSM
- Standardiziran način definicije FSM v VHDL ne obstaja
- Osnovni pristop
  - Definiramo uporabniški podatkovni tip (user-defined data type) ki bo predstavljal stanja v FSM
  - Signal predstavlja izhode FF (spremenljivke stanj), s katerimi realiziramo stanja v FSM
  - VHDL prevajalnik izbere primerno število FF med sintezo FSM
  - Kodiranje stanj lahko izvedemo sami, ali pa ga namesto nas naredi prevajalnik.

Napotki pri snovanju FSM v VHDL

- Izhodi FSM naj bodo vedno Mooreovega tipa, ne Mealyevega (ker so asinhroni)
- Uporabljajte registre, ne zapahov (ker so stanjski, ne robni)
- Vhodi za nastavitev in ponastavitev (set/reset) naj bodo sinhroni, ne asinhroni (ker tehnologija ne omogoča obojega).
- Za točnejše modeliranje uporabljajte after (svet ni lep – zakasnitve obstajajo)
- Zmanjšajte zdrs signala ure kot se le da.

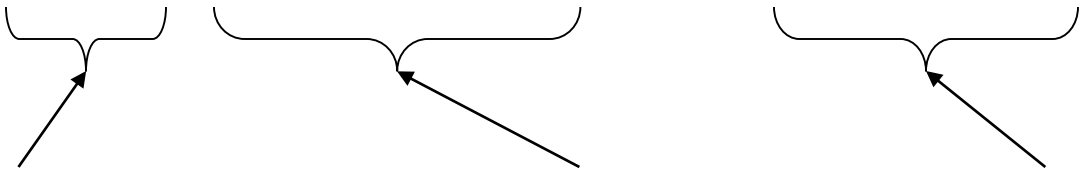
Povzeto po: VLSI Technology (1997)

- Če prehajate med prostori različnih signalov ure uporabljajte za to namenjene strukture (FIFO, dual port RAM). Krmilne signale vedno sinhronizirajte s prostorom signala ure.
- Ne imejte neuporabljenih stanj v FSM
- Ne uporabljajte logike z asinhronimi povratnimi vezavami.
- Ne verjemite simulatorju in tistemu kar *želite* videti. Svet je vedno realen in bistvo je očem vedno prikrito.

Uporabniški podatkovni tipi

- Rezervirana beseda **TYPE** označuje začetek novega tipa, ki definira stanja v FSM

```
TYPE State_type IS (A, B, C);
```



definicija uporabniškega podatkovnega tipa

Ime tipa

Veljavne vrednosti tipa

Definira uporabniški podatkovni tip (z imenom State_type) ki zavzame 3 različne vrednosti: A, B ali C).

Predstavitev stanj

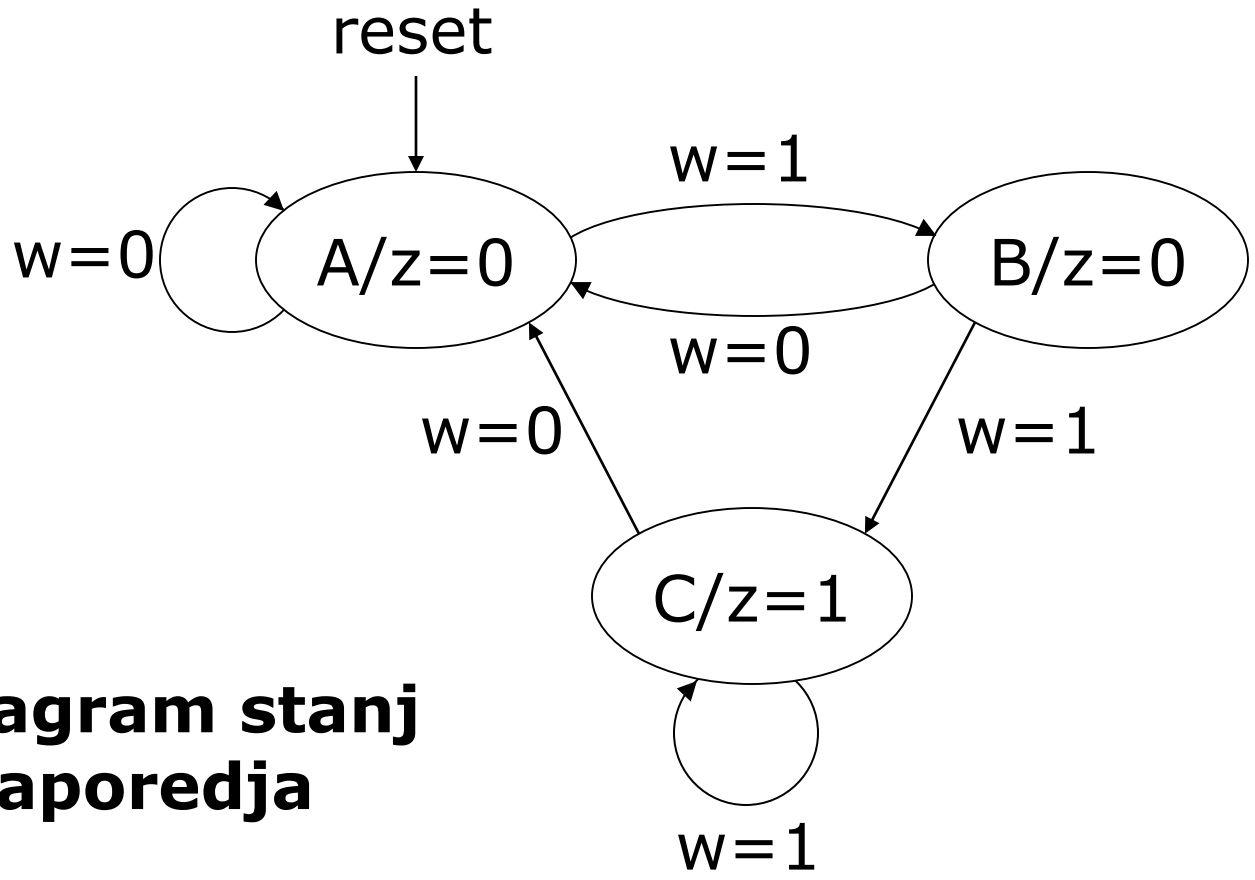
- Rezervirana beseda **SIGNAL** označuje interno povezavo v PLD vezju. Z njo predstavimo izhode FF, saj je naštevnega tipa **State_type**

```
TYPE State_type IS (A, B, C);
SIGNAL y: State_type;
```

Signal y predstavlja izhode FF za FSM ki ima 3 stanja

Primer načrtovanja

- Načrtajte VHDL opis vezja, ki zazna vhodno zaporedje ’11’. Ko jo zazna, postane izhod **w=1**



Moore-ov diagram stanj detektorja zaporedja

Sinteza v VHDL – entiteta

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY detect IS
PORT(clk, resetn, w : IN STD_LOGIC ;
     z : OUT STD_LOGIC) ;
END detect ;
ARCHITECTURE Behavior OF detect IS
  TYPE State_type IS (A,B,C) ;
  SIGNAL y: State_type ;
BEGIN
```

Sinteza v VHDL – arhitektura

```
PROCESS ( resetn, clk )
BEGIN
IF resetn = '0' THEN
   y <= A;
ELSIF (clk'EVENT AND clk='1') THEN
    CASE y IS
       WHEN A =>
         IF w='0' THEN
                  y <= A;
         ELSE
                  y <= B;
         END IF;
       WHEN B =>
         IF w='0' THEN
                  y <= A;
         ELSE
                  y <= C;
         END IF;
       WHEN C =>
            IF w='0' THEN
                     y <= A;
            ELSE
                     y <= C;
            END IF;
       END CASE;
   END IF;
   END PROCESS
    z <= '1' WHEN y = C ELSE '0';
   END Behavior;
```

To definira Moore-ov avtomat. Izhod je funkcija stanja – ne pa tudi vhodov! Izhodi so sinhroni (spremenijo se kvečjemu takrat, ko se spremeni stanje)

Drugačno kodiranje v VHDL

- Druga oblika opisa vezja v VHDL je, da definiramo dva signala, ki predstavljata stanje FSM
  - En signal, **y_present**, definira trenutno stanje FSM
  - Drug signal, **y_next**, definira naslednje stanje FSM
- Zgornji zapis sovpada z zapisom **y** (trenutno stanje) and **Y** (naslednje stanje), ki smo ga uporabljali prej
- Uporabljali bomo **dva** PROCESS stavka, katerima opišemo delovanje FSM
  - Prvi definira tabelo stanj (STATE TABLE) kot kombinacijsko vezje
  - Drugi definira FF, tako da spremenljivka **y_present** dobi vrednost **y_next** po vsakem pozitivnem robu signala ure

Drugačno kodiranje v VHDL

```
ARCHITECTURE Behavior OF detect IS
TYPE State_type IS (A,B,C);
SIGNAL y_present, y_next: State_type;
BEGIN
PROCESS(w,y_present)
BEGIN
CASE y_present IS
    WHEN A =>
       IF w='0' THEN y_next <= A;
       ELSE y_next <= B;
    WHEN B =>
       IF w='0' THEN y_next <= A;
       ELSE y_next <= C;
    WHEN C =>
        IF w='0' THEN y_next <= A;
        ELSE y_next <= C;
        END CASE;
  END PROCESS;
```

```
PROCESS(clk,resetn)
BEGIN
    IF resetn='0' THEN
          y_present <= A;
    ELSIF(clk'EVENT AND clk='1') THEN
          y_present <= y_next;
    END IF;
END PROCESS
```

```
z <='1' WHEN y_present = C ELSE '0';
```

```
END Behavior;
```

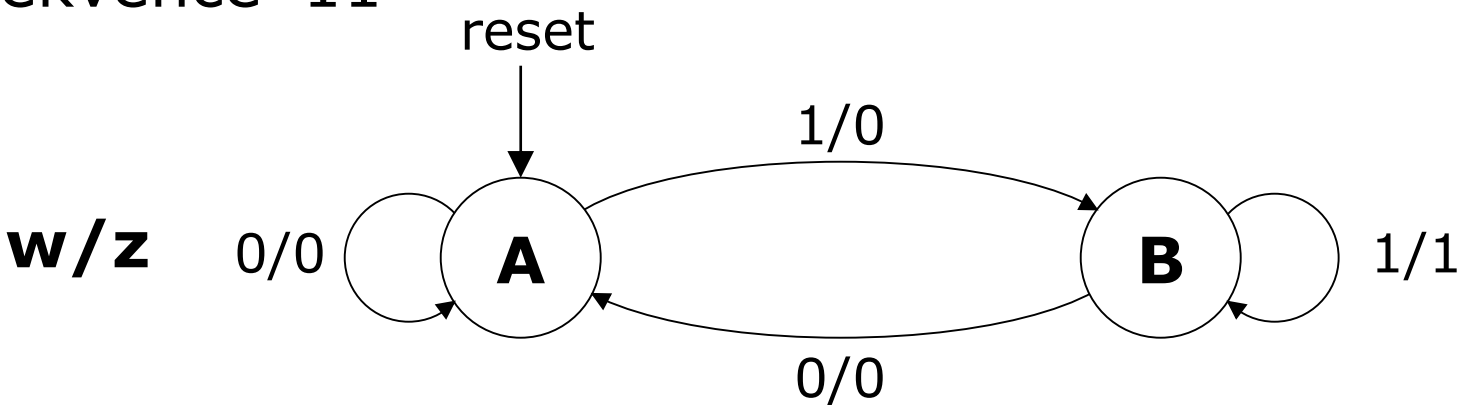
Uporabniško kodiranje stanj

- V preteklem zgledu je kodiranje stanj potekalo avtomatsko – izvaja ga VHDL prevajalnik
- Kodiranje stanj lahko zahtevamo ob tvorbi podatkovnega tipa stanj:

```
ARCHITECTURE Behavior OF simple IS
TYPE State_type IS (A, B, C) ;
ATTRIBUTE ENUM_ENCODING : STRING ;
ATTRIBUTE ENUM_ENCODING OF State_type : TYPE IS "00 01 11" ;
SIGNAL y_present, y_next : State_type ;
BEGIN
```

VHDL koda Mealy-evaga FSM

- Mealy-ev FSM opišemo na podoben način kot Moore FSM
- Prehodi stanj so opisani na isti način kot v prvem VHDL zgledu
- Bistvena razlika v primeru Mealy-evega FSM je v načinu pisanja kode za izhode vezja
- Če ponovimo Mealy-ev diagram stanj za detektor sekvence ’11’



Mealy-ev detektor sekvence ’11’ VHDL

```
ARCHITECTURE Behavior OF detect IS
TYPE State_type IS (A,B) ;
SIGNAL y: State_type ;
BEGIN
PROCESS(resetn,clk)
BEGIN
IF resetn='0' THEN
   y <= A;
ELSIF(clk'EVENT AND clk='1') THEN
   CASE y IS
        WHEN A =>
                IF w='0' THEN y<= A;
                ELSE y<= B;
        END IF;
   WHEN B =>
        IF w='0' THEN y<= A;
        ELSE y<= B;
        END IF;
   END CASE;
END IF;
END PROCESS;
```

```
PROCESS(y,w)
BEGIN
   CASE y IS
        WHEN A =>
                z <='0';
        WHEN B =>
                z <= w;
        END CASE;
END PROCESS;
END Behavior;
```

Pozor – izhodi Mealyevega FSM so asinhroni!

Načrtovanje digitalnih vezij

Sinhronska sekvenčna vezja:
Realizacija zapletenejših FSM

Kodiranje stanj "ena naenkrat"

- Ena možnost kodiranja stanj je, da izberemo toliko spremenljivk stanja kot je stanj v sekvenčnem vezju
- Za vsako stanje so vse spremenljivke stanja razen ene enake 0
- Spremenljivka, katere vrednost je 1 je aktivna
  - ***Od tod ime "ena naenkrat" (ang. one-hot)***
- Poveča število FF, ki se rabijo za realizacijo, vendar zagotavlja enostavnejše izhodne izraze
  - Enostavnejši izhodni izrazi omogočajo hitrejše delovanje vezja, saj je zakasnitev širjenja manjša od izhodov FF do končnih izhodov sekvenčnega vezja

Realizacija avtomatov s kodiranjem stanj “ena naenkrat".

- Število pomnilnih celic = številu stanj.
- Prednost je lažje načrtovanje,
- pomanjkljivost pa poraba pomnilnih celic, zato pri avtomatih z velikim številom stanj ne pride v poštev.
- Kodiranje stanj: Izhod D-FF je Q=1, kadar je avtomat v stanju S, sicer 0.
- Če je avtomat podoben števcu, je realizacija podobna pomikalnemu registru (v limitnem primeru je kar pomikalni register)
- Realizacija avtomata – števca je kar krožni števec.

Realizacija avtomatov s kodiranjem stanj "ena naenkrat".

	x_1	x_2	x_3	x_4	y
S_0	S_1	S_0	S_0	S_4	y_1
S_1	S_1	S_2	S_0	S_0	y_1
S_2	S_1	S_2	S_3	S_0	y_1
S_3	S_1	S_2	S_3	S_4	y_1
S_4	S_0	S_2	S_3	S_4	y_2

	ξ_0	ξ_1
x_1	0	0
x_2	0	1
x_3	1	0
x_4	1	1

trenutno							naslednje				
ξ_0	ξ_1	Q_0	Q_1	Q_2	Q_3	Q_4	Q_0	Q_1	Q_2	Q_3	Q_4
0	0	1	0	0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0	1	0	0	0
0	0	0	0	0	0	1	1	0	0	0	0
0	1	1	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	1	0	0
0	1	0	0	1	0	0	0	0	1	0	0
0	1	0	0	0	1	0	0	0	1	0	0
0	1	0	0	0	0	1	0	0	1	0	0
1	0	1	0	0	0	0	1	0	0	0	0
1	0	0	1	0	0	0	1	0	0	0	0
1	0	0	0	1	0	0	0	0	0	1	0
1	0	0	0	0	1	0	0	0	0	1	0
1	0	0	0	0	0	1	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	0	1
1	1	0	1	0	0	0	1	0	0	0	0
1	1	0	0	1	0	0	1	0	0	0	0
1	1	0	0	0	1	0	0	0	0	0	1
1	1	0	0	0	0	1	0	0	0	0	1

Realizacija avtomatov s kodiranjem stanj "ena naenkrat".

- Do istega lahko pridemo na hitrejši način (brez pisanja aplikacijske tabele) s pomočjo **tabele prednikov**.

Tabela prehajanja stanj

	x_1	x_2	x_3	x_4	y
S_0	S_1	S_0	S_0	S_4	y_1
S_1	S_1	S_2	S_0	S_0	y_1
S_2	S_1	S_2	S_3	S_0	y_1
S_3	S_1	S_2	S_3	S_4	y_1
S_4	S_0	S_2	S_3	S_4	y_2

Tabela prednikov

	x_1	x_2	x_3	x_4	prednikov
S_0	S_4	S_0	$S_0\ S_1$	$S_1\ S_2$	6
S_1	$S_0\ S_1\ S_2\ S_3$	–	–	–	4
S_2	–	$S_1\ S_2\ S_3\ S_4$	–	–	4
S_3	–	–	$S_2\ S_3\ S_4$	–	3
S_4	–	–	–	$S_0\ S_3\ S_4$	3

Tabelo prednikov dobimo iz tabele prehajanja stanj (ali diagrama prehajanja stanj) če si zapišemo vsa stanja, ki vodijo v določeno stanje.

Realizacija avtomatov s kodiranjem stanj "ena naenkrat".

- *V*arčno kodiranje stanj (z manj pomnilnimi celicami): Stanja z največ predniki naj imajo najmanj enic.
- Stanje z največ predniki kodiramo s samimi ničlami.
- Naslednje stanje glede na število prednikov imata eno '1' v kodi stanja.
- Zlato pravilo pri izbiranju kod stanja je: Hammingova razdalja = 1 (med sosednima stanjema naj se spremeni največ eno mesto)

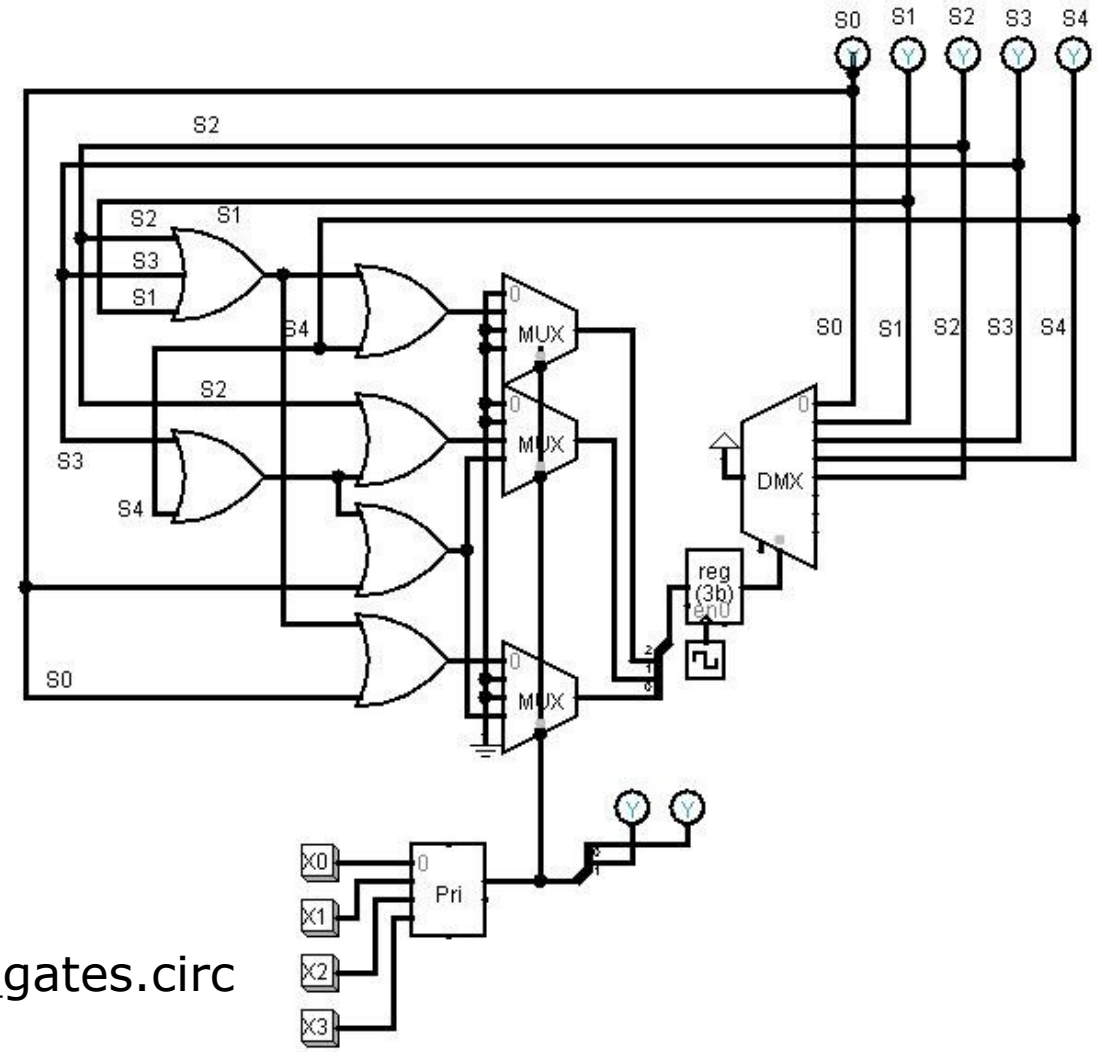
Tabela prednikov

	x_1	x_2	x_3	x_4	prednikov
S_0	S_4	S_0	$S_0\ S_1$	$S_1\ S_2$	6
S_1	$S_0\ S_1\ S_2\ S_3$	–	–	–	4
S_2	–	$S_1\ S_2\ S_3\ S_4$	–	–	4
S_3	–	–	$S_2\ S_3\ S_4$	–	3
S_4	–	–	–	$S_0\ S_3\ S_4$	3

Možna izbira kodiranja stanj

	Q_0	Q_1	Q_2
S_0	0	0	0
S_1	0	0	1
S_2	1	0	0
S_3	0	1	0
S_4	0	1	1

Realizacija avtomatov s kodiranjem stanj "ena naenkrat".



Logisim:
fsm_register_gates.circ

Avtomat s skočnim števnikom

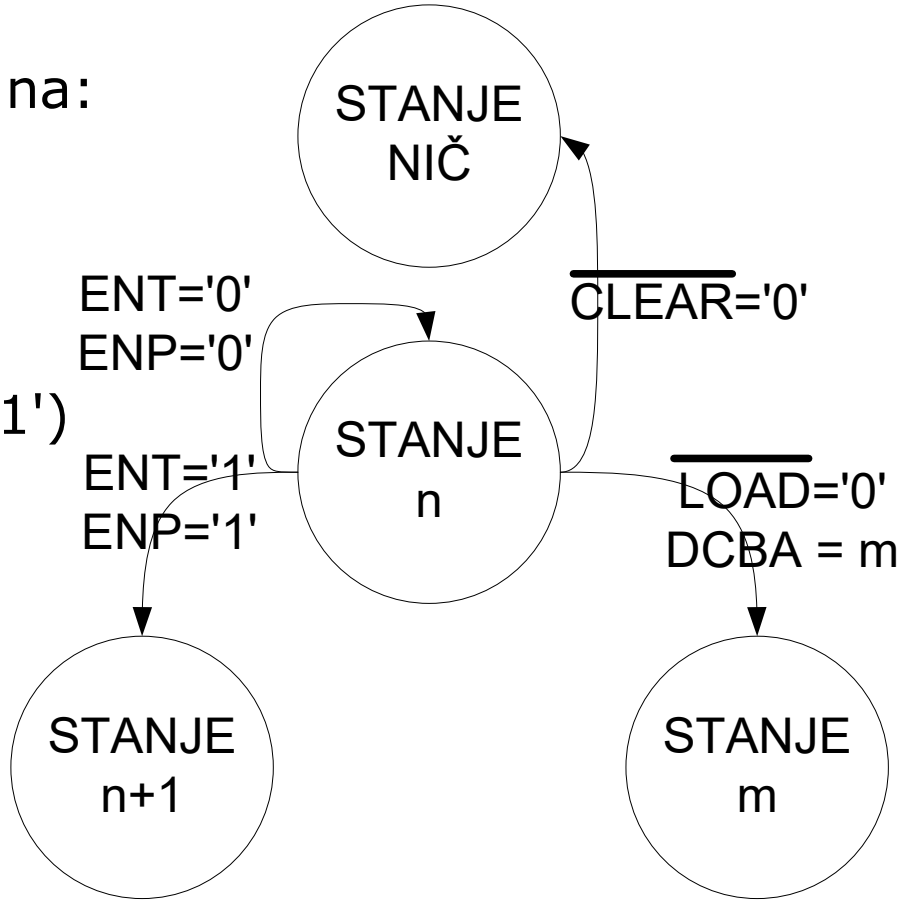
- Skočni števnik (“jump counter").
- Uporaben je v primerih, ko je avtomat podoben števcu, torej ko večina prehodov sledi fiksnemu zaporedju.
- Ostali prehodi pa so izvedeni:
  - z resetiranjem števnika (prehodi v začetno stanje),
  - z nalaganjem novega stanja (“load”).

Avtomat s skočnim števnikom

Uporabljamo sinhrone števce, ki imajo $\overline{\text{CLEAR}}$, $\overline{\text{LOAD}}$, COUNT (npr. 74163)

Iz nekega stanja n lahko preidemo na:

1. Na stanje nič ($\overline{\text{CLEAR}}$='0')
   (števec zbrišemo)
2. Na naslednje stanje n+1
   (števec prišteje 1 ➔ ENT,ENP='1')
3. Na poljubno naslednje stanje m
   ($\overline{\text{LOAD}}$='0', na vhodih DCBA
   je koda novega stanja m)
4. Čakanje v trenutnem stanju
   (ENT, ENP = '0' ➔ števec stoji)



Pomembno je kodiranje stanj,
da se čimbolj prilagodimo delovanju števca!

Realizacija avtomata s skočnim števnikom

V diagramu prehajanja stanj poiščemo vse primere prehodov v stanje nič ($S_x \rightarrow S_0$) in to zapišemo z logičnim izrazom za $\overline{CLEAR}$

	x_1	x_2	x_3	y
S_0	S_0	S_0	S_1	0
S_1	S_2	S_2	S_3	0
S_2	S_4	S_3	S_1	0
S_3	S_3	S_4	S_0	0
S_4	S_0	S_4	S_4	1

- Števec bomo zbrisali če smo v:
  - stanju S_0 in je na vhodu x_1
  - stanju S_0 in je na vhodu x_2
  - stanju S_3 in je na vhodu x_3
  - stanju S_4 in je na vhodu x_1

$$\overline{CLEAR} = S_0 \cdot x_1 + S_0 \cdot x_2 + S_3 \cdot x_3 + S_4 \cdot x_1$$
$$\overline{CLEAR} = S_0 \cdot (x_1 + x_2) + S_3 \cdot x_3 + S_4 \cdot x_1$$

Realizacija avtomata s skočnim števnikom

V diagramu prehajanja stanj poiščemo vse primere prehodov nalaganja stanja ($S_n \rightarrow S_m$) in to zapišemo z logičnim izrazom za $\overline{LOAD}$.

	x_1	x_2	x_3	y
S_0	S_0	S_0	S_1	0
S_1	S_2	S_2	S_3	0
S_2	S_4	S_3	S_1	0
S_3	S_3	S_4	S_0	0
S_4	S_0	S_4	S_4	1

- Števec bomo naložili s kodo stanja S_m če smo v:
  - stanju S_1 in je na vhodu x_3 (naložimo stanje S_3)
  - stanju S_2 in je na vhodu x_1 (naložimo stanje S_4)
  - stanju S_2 in je na vhodu x_3 (naložimo stanje S_1)

$$\overline{LOAD} = S_1 \cdot x_3 + S_2 \cdot x_1 + S_2 \cdot x_3$$
$$\overline{LOAD} = S_1 \cdot x_3 + S_2 \cdot (x_1 + x_3)$$

Realizacija avtomata s skočnim števnikom

Določiti moramo še izraze za vzporedno nalaganje števca DCBA

- Števec bomo naložili s kodo stanja S_m če smo v:
  - stanju S_1 in je na vhodu x_3 (takrat naložimo kodo stanja S_3 ➔ DCBA=0011)
  - stanju S_2 in je na vhodu x_1 (takrat naložimo kodo stanja S_4 ➔ DCBA=0100)
  - stanju S_2 in je na vhodu x_3 (takrat naložimo kodo stanja S_1 ➔ DCBA=0001)
- Vhod A bo '1' vedno, ko je:
  - $\overline{LOAD}$='0' (aktiven) **in**
  - ko smo v stanju S_1 **in** je na vhodu x_3 **ali**
  - ko smo v stanju S_2 **in** je na vhodu x_3
- Vhod B bo '1' vedno, ko je
  - $\overline{LOAD}$='0' (aktiven) **in**
  - ko smo v stanju S_1 **in** je na vhodu x_3
- Vhod C bo '1' vedno, ko je
  - $\overline{LOAD}$='0' (aktiven) **in**
  - ko smo v stanju S_4

$$A=\overline{\overline{LOAD}}\cdot(S_1\cdot x_3+S_2\cdot x_3)=LOAD\cdot(S_1+S_2)\cdot x_3$$

$$B=\overline{\overline{LOAD}}\cdot S_1\cdot x_3$$

$$C=\overline{\overline{LOAD}}\cdot S_2\cdot x_1$$

$$D='0'$$

Kode stanj pri nalaganju:

	D	C	B	A
S_0	0	0	0	0
S_1	0	0	0	1
S_2	0	0	1	0
S_3	0	0	1	1
S_4	0	1	0	0

Realizacija avtomata s skočnim števnikom

V diagramu prehajanja stanj poiščemo vse primere prehodov zadrževanja stanja ($S_n \rightarrow S_n$) in to zapišemo z logičnim izrazom za ENP, ENT.

	x_1	x_2	x_3	y
S_0	S_0	S_0	S_1	0
S_1	S_2	S_2	S_3	0
S_2	S_4	S_3	S_1	0
S_3	S_3	S_4	S_0	0
S_4	S_0	S_4	S_4	1

- Števec bomo zadržali če smo v:
  - stanju S_3 in je na vhodu x_1
  - stanju S_4 in je na vhodu x_2
  - stanju S_4 in je na vhodu x_3

$$ENT=ENP='0'=S_3 \cdot x_1 + S_4 \cdot x_2 + S_4 \cdot x_3$$
$$ENT=ENP='0'=S_3 \cdot x_1 + S_4 \cdot (x_2 + x_3)$$

Realizacija avtomata s skočnim števnikom

V diagramu prehajanja stanj poiščemo vse primere prehodov štetja ($S_n \rightarrow S_{n+1}$) in to zapišemo z logičnim izrazom za ENP, ENT.

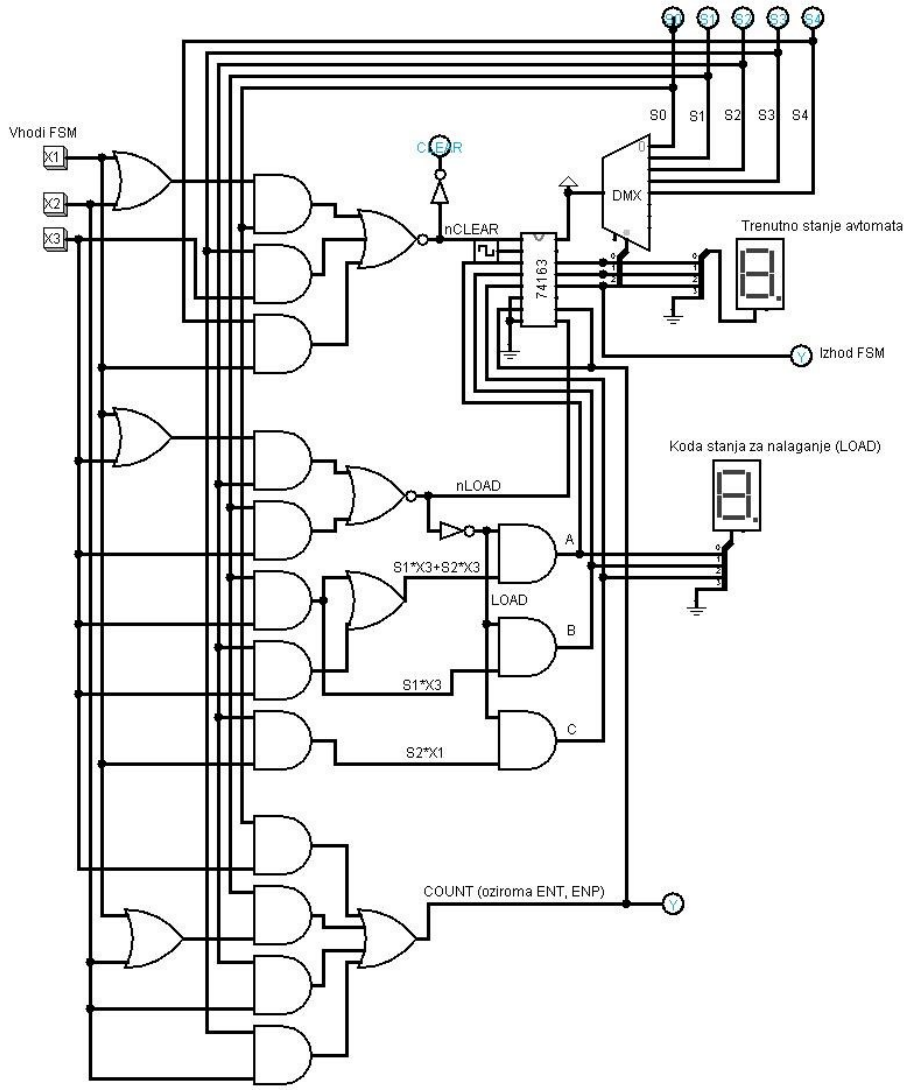
	x_1	x_2	x_3	y
S_0	S_0	S_0	S_1	0
S_1	S_2	S_2	S_3	0
S_2	S_4	S_3	S_1	0
S_3	S_3	S_4	S_0	0
S_4	S_0	S_4	S_4	1

- Števec bo štel če smo v:
  - stanju S_0 in je na vhodu x_3
  - stanju S_1 in je na vhodu x_1
  - stanju S_1 in je na vhodu x_2
  - stanju S_2 in je na vhodu x_2
  - stanju S_3 in je na vhodu x_2

$$ENT=ENP='1'=S_0 \cdot x_3 + S_1 \cdot x_1 + S_1 \cdot x_2 + S_2 \cdot x_2 + S_3 \cdot x_2$$

$$ENT=ENP='1'=S_0 \cdot x_3 + S_1 \cdot (x_1 + x_2) + S_2 \cdot x_2 + S_3 \cdot x_2$$

Realizacija avtomata s skočnim števnikom



Logisim simulacija:
jump_counter.circ

	x_1	x_2	x_3	y
S_0	S_0	S_0	S_1	0
S_1	S_2	S_2	S_3	0
S_2	S_4	S_3	S_1	0
S_3	S_3	S_4	S_0	0
S_4	S_0	S_4	S_4	1

Izhod avtomata y bo '1' samo v stanju S_4, kar je samo pri Q_C='1'.

Načrtovanje digitalnih vezij

Spominska vezja:

Realizacija vgrajenih spominov v FPGA vezjih

Ojačevalnik signala ure (ang. clock buffer)

- Ojačevalnik signala ure uporabljamo zato, da povečamo zmožnost poganjanja izhoda (ang. fanout) in za minimiziranje zdrsa signala ure.
- Z deklaracijo ojačevalnika signala ure (BUFG) sintetizatorju napovemo, da bo dani signal uporablja za globalni signal ure (GCLK) v kompleksnejših vezjih.
- Ojačevalnik signala ure napovemo v Xilinx ISE takole:

```
component BUFG
port (    O : out STD_ULOGIC;
          I : in STD_ULOGIC);
end component;
-- zgornjo deklaracijo komponente postavimo med is in begin
  arhitekture vezja
BUFG_INSTANCE_NAME : BUFG
port map (O => user_O, I => user_I);
```

Tristanjski izhodi v FPGA

- Tristanjski ojačevalniki omogočajo več enotam krmiljenje istih signalov ob različnih časih, ne da bi se enote pri tem medsebojno motile.
- Če bi na iste signala priključili dve enoti, ki bi brez tristanjskega ojačevalnika krmilile iste signale, bi ena lahko vsiljevala ‘1’ na isti liniji, ko bi druga vsiljevala ‘0’.

```
entity tribuf is
generic(w : positive);
Port (      pass : in STD_LOGIC;
            d_in : in STD_LOGIC_VECTOR (w-1 downto 0);
            d_out : out STD_LOGIC_VECTOR (w-1 downto 0));
end tribuf ;
architecture a of tribuf is
constant hi_Z : STD_LOGIC_VECTOR(w-1 downto 0) := (others => 'Z');
begin
d_out <= d_in when pass ='0' else hi_Z;
end a;
```

Vrste spomina v FPGA

V večini modernih FPGA vezij lahko realiziramo dve vrsti spominskih elementov:

- Porazdeljeni RAM (ang. distributed RAM) je realiziran kot polje FF, ki so sestavni del CLB. Tovrstni spomin je primeren za hranjenje majhnih količin podatkov, izvedbo spominskih, pomikalnih registrov, ...
- Blok RAM (BRAM) je namenski, nastavljiv element, ki obstaja kot vgrajena enota znotraj FPGA. Pri tej vrsti RAM elementa nastavljamo širino naslova, podatka in obliko nadzora. BRAM je namenjen hranjenju velikih količin podatkov, kjer bi se z realizacijo s porazdeljenim RAM elementom hitro zgodilo, da bi porabili vse CLB.

Blok RAM elementi v FPGA

- Tipična velikost elementa BRAM v FPGA je 18kbit (Virtex4), ki ga lahko nastavljamo v poljubni geometriji (ang. aspect ratio):
  - Od 16k 1-bitnega elementa,
  - preko 8k 2-bitnega elementa do
  - 512 36-bitnega elementa.
- BRAM bloke lahko enostavno širimo in daljšamo v večje strukture. Za 1k 36 bitnega RAM elementa bo sintetizator uporabil 2 BRAM bloka.
- BRAM elementi podpirajo tudi dvovratni dostop (ang. dual-port RAM), pri čemer imajo vsaka vrata lastno naslovno vodilo, signal ure in nadzorno vodilo (EN, WE). Tudi širini vrat se lahko razlikujeta.
- Da sintetizator spozna, da želimo uporabiti BRAM element namesto porazdeljenega, moramo uporabiti poseben način deklaracije entitete in arhitekture.

Blok RAM element z enojnimi vrati v FPGA

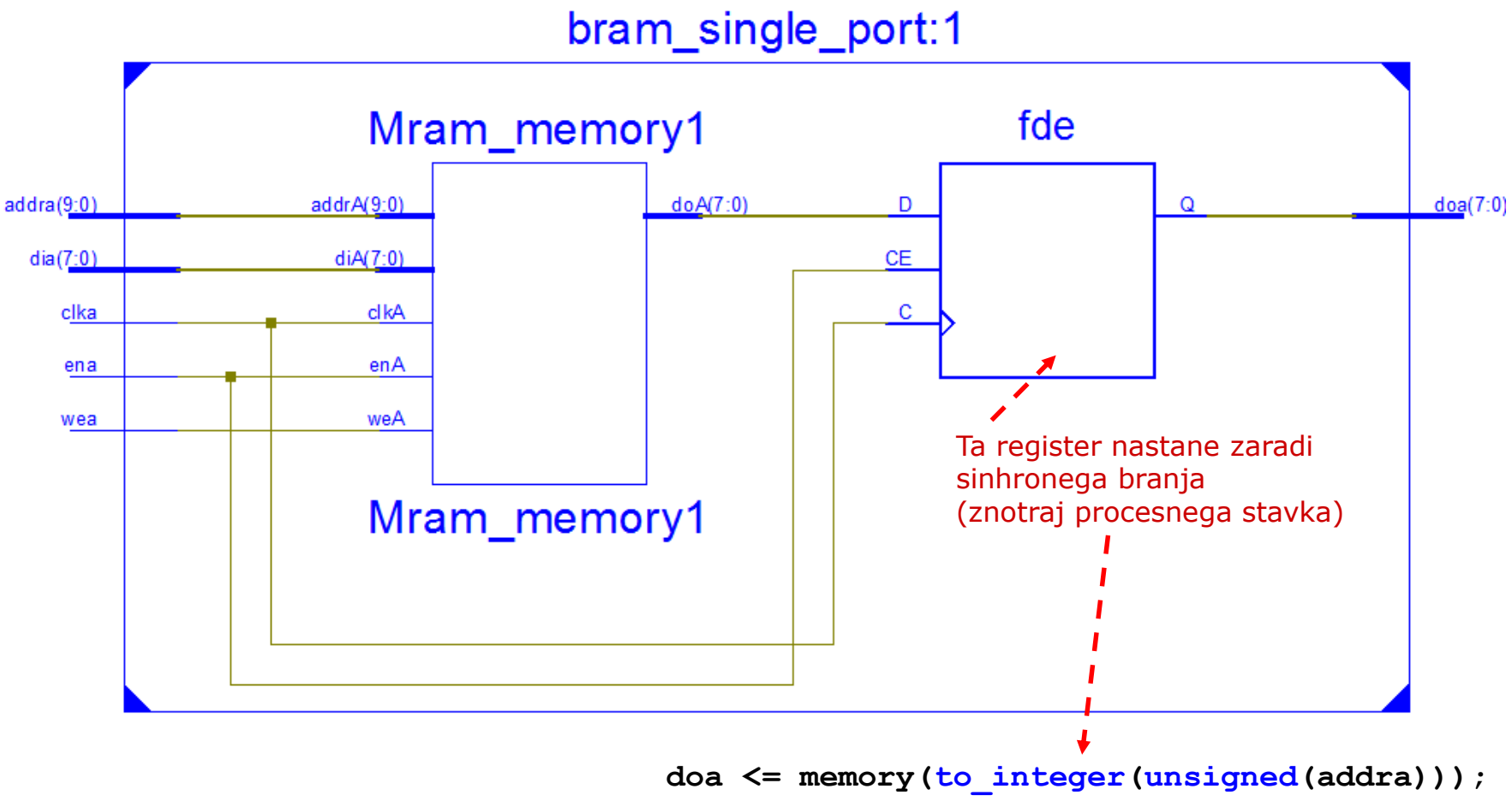
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.math_real.all;
entity bram_single_port is
  generic ( DEPTH : positive := 1024;
            WIDTH : positive := 8);
  port ( clka, ena, wea    : in  std_logic;
    addra : in  std_logic_vector(integer(ceil(log2(real(DEPTH))))-1 downto 0);
    dia   : in  std_logic_vector(WIDTH-1 downto 0);
    doa   : out std_logic_vector(WIDTH-1 downto 0));
end entity bram_single_port;
architecture a of bram_single_port is
type MEM_TYPE is array(0 to DEPTH-1) of std_logic_vector(WIDTH-1 downto 0);
signal memory : MEM_TYPE;
begin
  process( clka )
  begin
    if ( rising_edge(clka) ) then
      if ( ena = '1' ) then
        if ( wea = '1' ) then
          memory(to_integer(unsigned(addra))) <= dia;
        end if;
        doa <= memory(to_integer(unsigned(addra)));
      end if;
    end if;
  end process;
end architecture a;
```

Primer BRAM elementa z enojnimi vrati (1k*8bit).
Ta BRAM najprej vpiše (ang. write-first),
nato postavi podatke na izhodno vodilo.
Možno je izvesti tudi obratno (ang. Read-then-write) ali
vpis brez sprememb (ang. no-change)

Macro Statistics
RAMs : 1
1024x8-bit single-port block RAM : 1

Sinhrono branje
(znotraj procesnega stavka)

Blok RAM element z enojnimi vrati v FPGA



Blok RAM element z dvojnimi vrati v FPGA

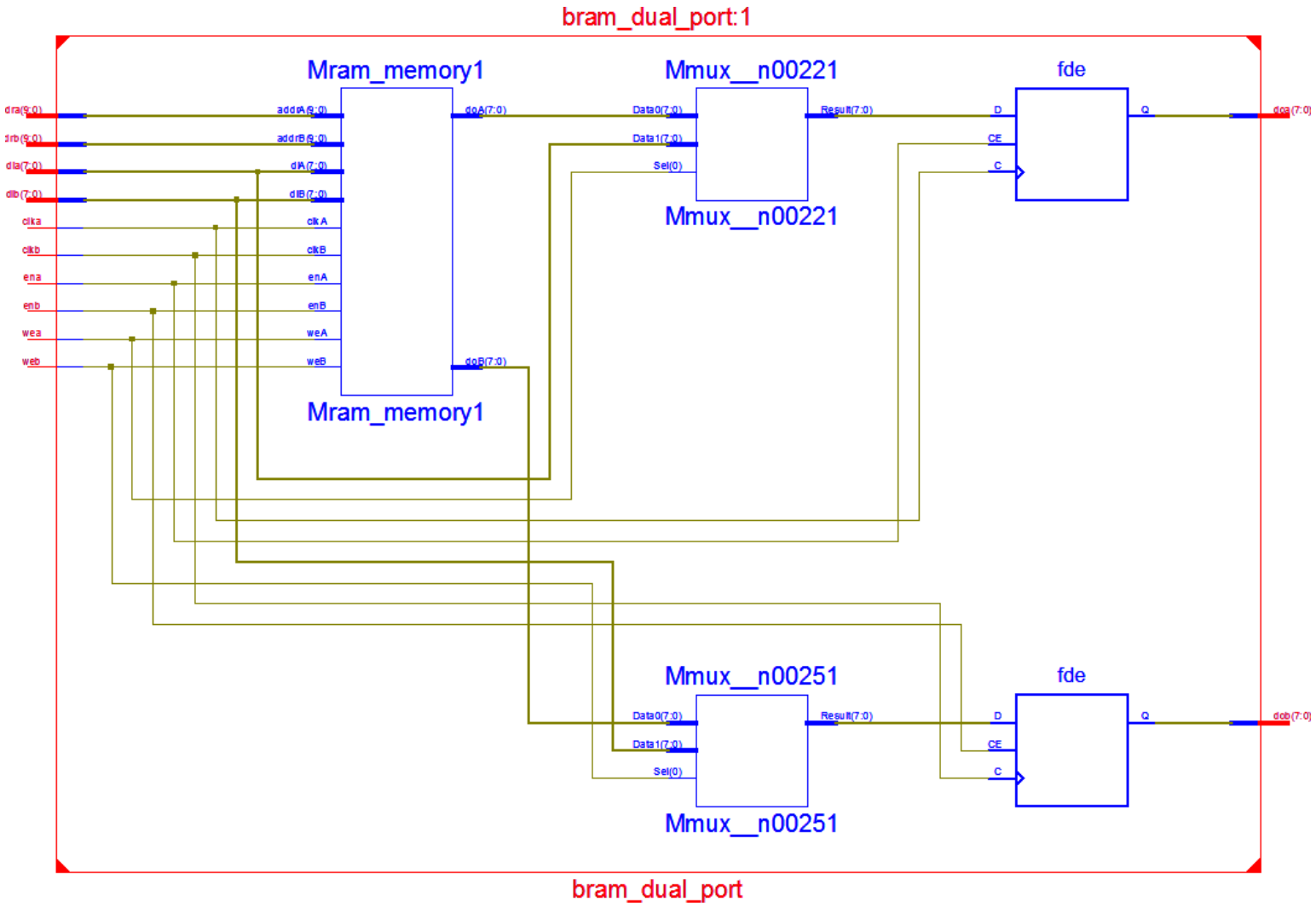
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.math_real.all;
entity bram_dual_port is
  generic (
    DEPTH : positive := 1024;
    WIDTH : positive := 8
  );
  port (
    -- vrata a (read/write)
    clka, ena, wea   : in  std_logic;
    addra : in  std_logic_vector(integer(ceil(log2(real(DEPTH))))-1 downto 0);
    dia   : in  std_logic_vector(WIDTH-1 downto 0);
    doa   : out std_logic_vector(WIDTH-1 downto 0);
    -- vrata b (read/write)
    clkb, enb, web   : in  std_logic;
    addrb : in  std_logic_vector(integer(ceil(log2(real(DEPTH))))-1 downto 0);
    dib   : in  std_logic_vector(WIDTH-1 downto 0);
    dob   : out std_logic_vector(WIDTH-1 downto 0)
  );
end entity bram_dual_port;
```

Blok RAM element z dvojnimi vrati v FPGA

```
architecture a of bram_dual_port is
type MEM_TYPE is array(0 to DEPTH-1) of std_logic_vector(WIDTH-1 downto 0);
shared variable memory : MEM_TYPE;
begin
  vrata_A: process( clka )
  begin
    if ( rising_edge(clka) ) then
      if ( ena = '1' ) then
        if ( wea = '1' ) then
          memory(to_integer(unsigned(addra))) := dia;
        end if;
        doa <= memory(to_integer(unsigned(addra)));
      end if;
    end if;
  end process vrata_A;
  vrata_B: process( clkb )
  begin
    if ( rising_edge(clkb) ) then
      if ( enb = '1' ) then
        if ( web = '1' ) then
          memory(to_integer(unsigned(addrb))) := dib;
        end if;
        dob <= memory(to_integer(unsigned(addrb)));
      end if;
    end if;
  end process vrata_B;
end architecture a;
```

ram_type	Block	
Port A		
aspect ratio	1024-word x 8-bit	
mode	write-first	
clkA	connected to signal <clka>	rise
enA	connected to signal <ena>	high
weA	connected to signal <wea>	high
addrA	connected to signal <addra>	
diA	connected to signal <dia>	
doA	connected to signal <doa>	
optimization	speed	
Port B		
aspect ratio	1024-word x 8-bit	
mode	write-first	
clkB	connected to signal <clkb>	rise
enB	connected to signal <enb>	high
weB	connected to signal <web>	high
addrB	connected to signal <addrb>	
diB	connected to signal <dib>	
doB	connected to signal <dob>	
optimization	speed	

Blok RAM element z dvojnimi vrati v FPGA



ROM v VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.math_real.all;
entity rom_simple is
  generic (
    DEPTH : positive := 256;
    WIDTH : positive := 12
  );
port (
clk, en    : in  std_logic;
addr : in std_logic_vector(integer(ceil(log2
    (real(DEPTH))))-1 downto 0);
    do    : out std_logic_vector(WIDTH-1
    downto 0)
);
end entity rom_simple;
```

ROM element, izveden kot blok RAM element s konstantno vsebino!

```
architecture a of rom_simple is
type MEM_TYPE is array(0 to DEPTH-1) of
   std_logic_vector(WIDTH-1 downto 0);
constant memory : MEM_TYPE := (
    0      => x"0C4",
    1      => x"4D2",
    2      => x"4DB",
    16#38#      => x"6C2",
    16#3A#      => x"0F1",
    16#3F#      => x"7D6",
    16#7F#      => x"4D0",
    16#AF#      => x"F9F",
    others => x"FFF");
begin
  process( clk )
  begin
    if ( rising_edge(clk) ) then
      if ( en = '1' ) then
            do <=
    memory(to_integer(unsigned(addr)));
      else
            do <= (others => 'Z');
        end if;
    end if;
  end process;
end architecture a;
```

ROM v VHDL

ROM element kot kodirnik

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.math_real.all;
entity rom_async is
  generic (DEPTH : positive := 256;
           WIDTH : positive := 12);
  port (
    en   : in  std_logic;
    addr : in  std_logic_vector(integer(ceil(log2(real(DEPTH))))-1 downto 0);
    do   : out std_logic_vector(WIDTH-1 downto 0));
end entity rom_async;
architecture a of rom_async is
type MEM_TYPE is array(0 to DEPTH-1) of std_logic_vector(WIDTH-1 downto 0);
constant memory : MEM_TYPE := (
    0      => x"0C4",
    1      => x"4D2",
    2      => x"4DB",
    16#38#     => x"6C2",
    16#3A#     => x"0F1",
    16#3F#     => x"7D6",
    16#7F#     => x"4D0",
    16#AF#     => x"F9F",
    others => x"FFF");
begin
  do <= memory(to_integer(unsigned(addr))) when (en = '1') else (others => 'Z');
end architecture a;
```

Inicializacija spominskih elementov v VHDL

- S poljem konstant, ki se vpiše ob aktivnem signalu ponastavitve (ang. reset).
- Z branjem zunanje datoteke
- S funkcijo

Incializacija s poljem konstant

```
type ram_type is array (0 to 31) of std_logic_vector(19 downto 0);
signal RAM : ram_type :=
   ( X"0200A", X"00300", X"08101", X"04000", X"08601",
   X"0233A", X"00300", X"08602", X"02310", X"0203B", X"08300",
   X"04002", X"08201", X"00500", X"04001", X"02500", X"00340",
   X"00241", X"04002", X"08300", X"08201", X"00500", X"08101",
   X"00602", X"04003", X"0241E", X"00301", X"00102", X"02122",
   X"02021", X"0030D", X"08201" );

type ram_type is array (255 downto 0) of std_logic_vector (15 downto
   0);
signal RAM : ram_type:= ( 196 downto 110 => X"B8B8",
100 => X"FEFC", 99 downto 0 => X"8282", others => X"3344");
```

Incializacija z datoteko

```
type RamType is array(0 to 7) of
   bit_vector(31 downto 0);
impure function InitRamFromFile
    (RamFileName : in string) return
   RamType is
FILE RamFile : text is in RamFileName;
variable RamFileLine : line;
variable RAM : RamType;
begin
    for I in RamType'range loop
         readline (RamFile,
   RamFileLine);
         read (RamFileLine, RAM(I));
    end loop;
    return RAM;
end function;
signal RAM : RamType :=
   InitRamFromFile("rams_20c.bin");
```

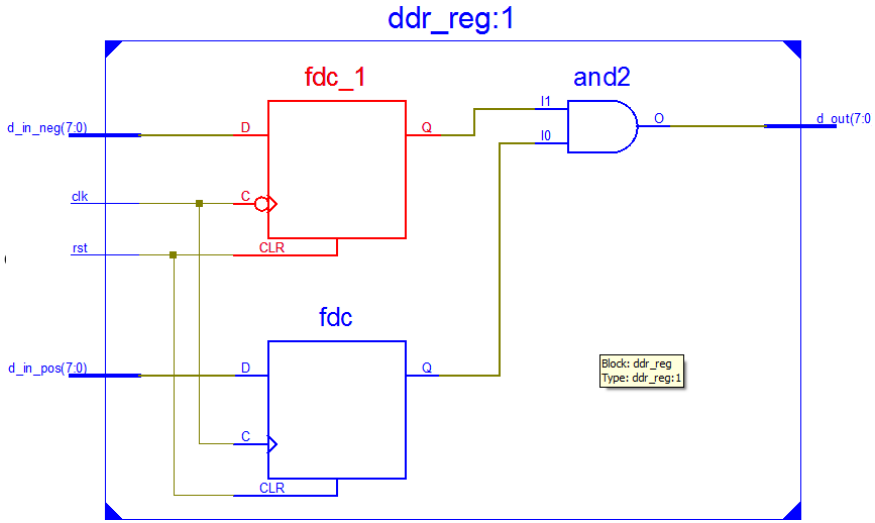
```
Datoteka je v takem zapisu:
00001111000011110000111100001111
01001010001000001100000010000100
00000000001111100000000001000001
11111101010000011100010000100100
00001111000011110000111100001111
01001010001000001100000010000100
00000000001111100000000001000001
11111101010000011100010000100100
```

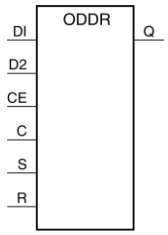
Incializacija s funkcijo

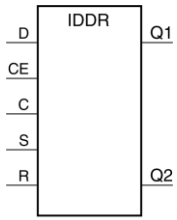
```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
use ieee.math_real.all;
entity rom_vhd is
    generic (
        ADDR_WIDTH       : integer := 8;
        DATA_WIDTH       : integer := 8
    );
    port (
        clk_i     : in  std_logic;
        addr_i    : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
        data_o    : out signed(DATA_WIDTH-1 downto 0)
    );
end rom_vhd;
architecture rtl of rom_vhd is
constant MEM_DEPTH : integer := 2**ADDR_WIDTH;
type mem_type is array (0 to MEM_DEPTH-1) of signed(DATA_WIDTH-1 downto 0);

function init_mem return mem_type is
    constant SCALE : real := 2**(real(DATA_WIDTH-2));
    constant STEP  : real := 1.0/real(MEM_DEPTH);
    variable temp_mem : mem_type;
begin
    for i in 0 to MEM_DEPTH-1 loop
        temp_mem(i) := to_signed(integer(cos(2.0*MATH_PI*real(i)*STEP)*SCALE), DATA_WIDTH);
    end loop;
    return temp_mem;
end;
constant mem : mem_type := init_mem;

begin
process (clk_i)
begin
    if rising_edge(clk_i) then
          data_o <= mem(to_integer(unsigned(addr_i)));
    end if;
end process;
end rtl;
```

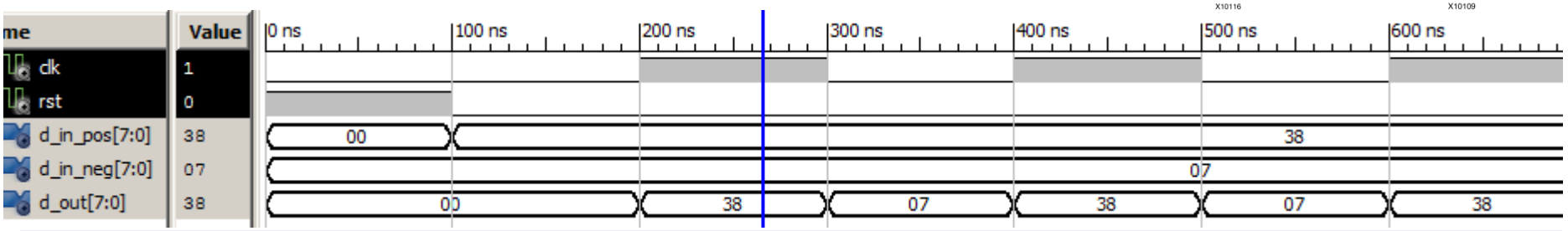
DDR prenos v vhodno-izhodnih blokih FPGA

DDR prenos podatkov omogoča prenos podatkov ob pozitivni in negativni fronti urinega signala.
Za tvorbo DDR prenosa potrebujemo:
•signala ure (vezja FPGA imajo dve urini domeni), ki sta fazno zamaknjena za 180' OTCLK1 in OTCLK2 ter
•izvora podatkov (O1, O2), ki sta sinhronizirana vsak s svojo uro.
• Vhodna oz. izhodna DDR vrata realiziramo s primitivom ODDR/IDDR (za serijo 7 Xilinx FPGA).
•Več o primitivih serije 7 najdete v "Libraries Guide, xapp462".







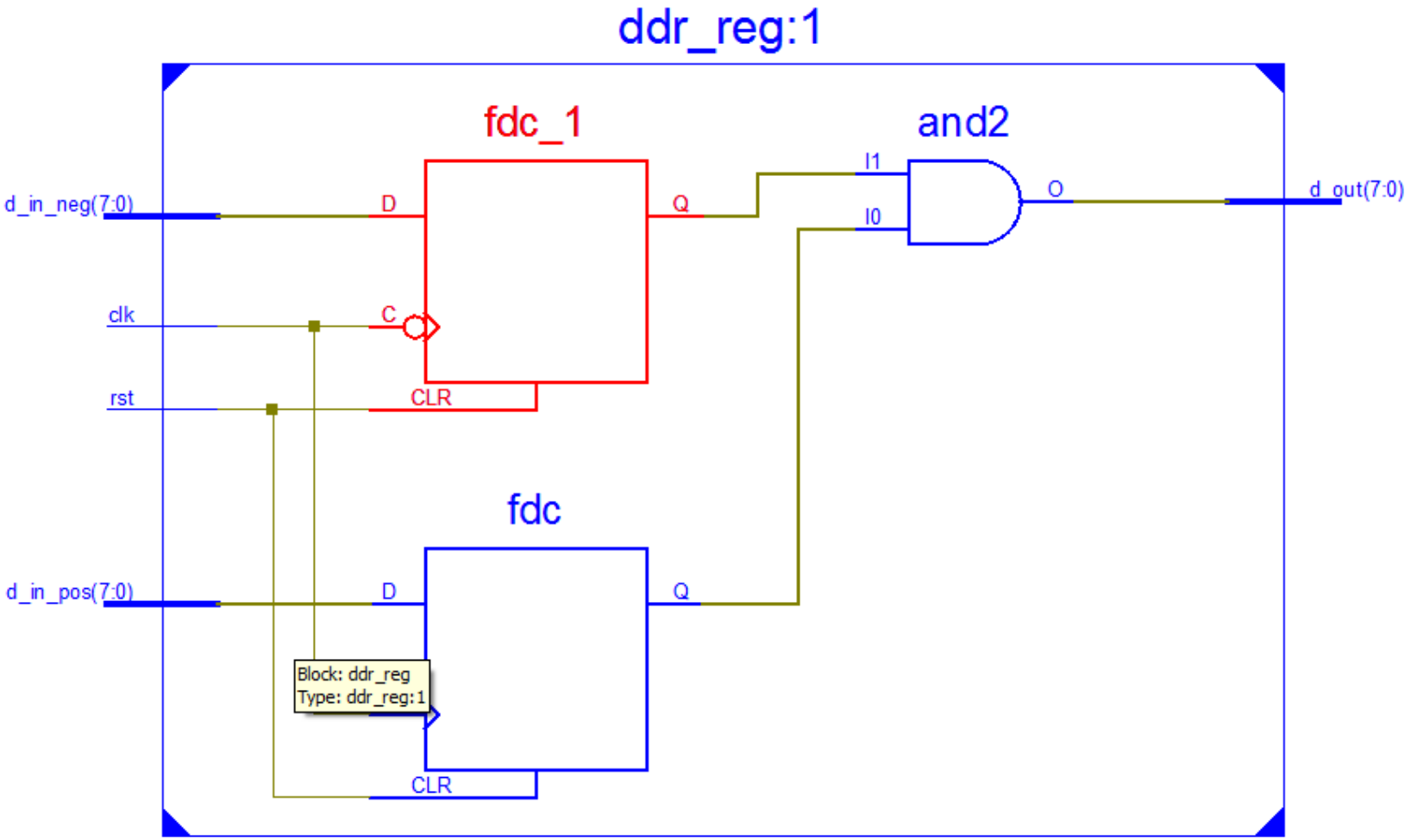


DDR prenos v vhodno-izhodnih blokih FPGA

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library UNISIM;
use UNISIM.vcomponents.all;
entity ddr_reg is
generic( ddr_reg_size : natural := 8
     );
port (clk, -- signal ure
      rst : in STD_LOGIC;  -- asinhroni reset
     d_in_pos, d_in_neg : in STD_LOGIC_VECTOR(ddr_reg_size -
     1 downto 0);          -- vhodni niz podatkov
     d_out: out STD_LOGIC_VECTOR(ddr_reg_size - 1 downto 0)
             -- izhodni niz podatkov
    );
end ddr_reg;
architecture a0 of ddr_reg is
begin
G1 : for i in d_in_pos'range generate
     begin
             ODDR_inst : ODDR
             generic map(
             DDR_CLK_EDGE => "OPPOSITE_EDGE",
             INIT => '0',
             RTYPE => "SYNC")
             port map (
             Q => d_out(i), -- 1-bit DDR izhod
             C => clk, -- 1-bit signal ure
             CE => '1', -- 1-bit omogoci uro
             D1 => d_in_pos(i), -- (pozitivni rob)
             D2 => d_in_neg(i), -- (negativni rob)
             R => rst, -- 1-bit vhod za ponastavitev
             S => '0' -- 1-bit vhod za postavitev
             );
     end generate;
end a0;
```

```
architecture a1 of ddr_reg is
begin
     process(clk, rst)
     begin
             if (rst = '1') then
                     d_out <= (others => '0');
             elsif rising_edge(clk) then
                     d_out <= d_in_pos;
             elsif falling_edge(clk) then
                     d_out <= d_in_neg;
             end if;
     end process;
end a1;
architecture a2 of ddr_reg is
signal r, f : STD_LOGIC_VECTOR(ddr_reg_size - 1 downto
     0);
begin
     process (clk, rst) begin
             if (rst = '1') then
                     r <= (others => '0');
             elsif rising_edge(clk) then
                     r <= d_in_pos;
             end if;
     end process;
     process (clk, rst) begin
             if (rst = '1') then
                     f <= (others => '0');
             elsif falling_edge(clk) then
                     f <= d_in_neg;
             end if;
     end process;
     d_out <= r and f;
end a2;
```

DDR prenos v vhodno-izhodnih blokih FPGA



Načrtovanje digitalnih naprav

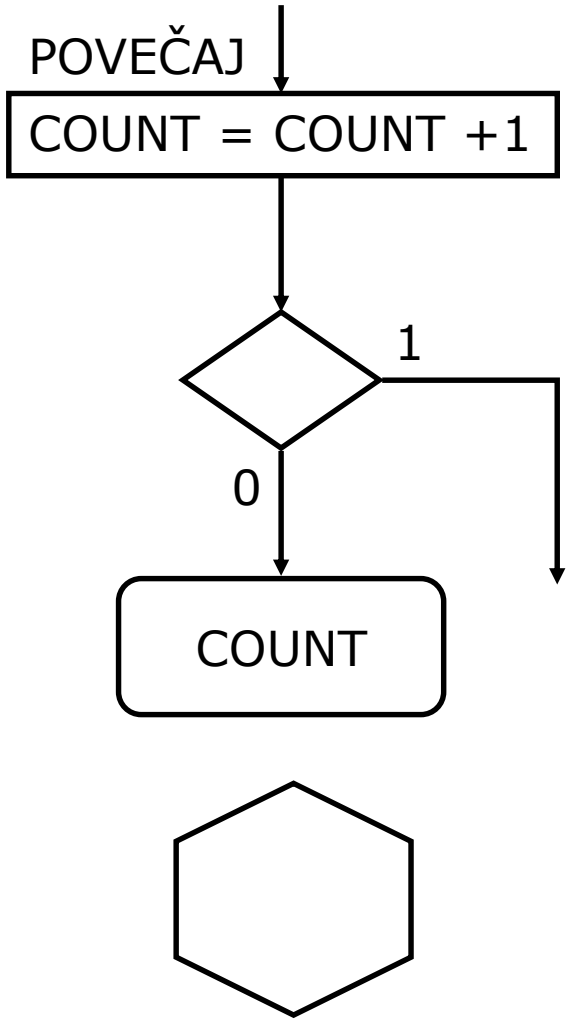
Algoritemski avtomati stanj
ASM
(Algorithmic state machines)

Algoritemski diagrami stanj

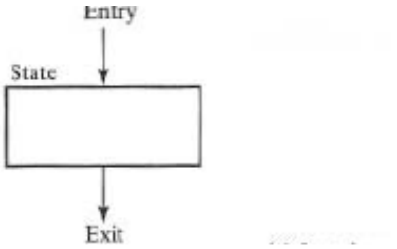
- Diagrami in tabele prehajanja stanj so primerni za opis manjših avtomatov končnih stanj.
- Drugačna oblika predstavitve je ASM diagram.
- ASM diagram je vrsta diagrama poteka (ang. flowchart), s katerim predstavimo prehajanja stanj.

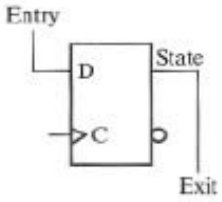
ASM diagrami

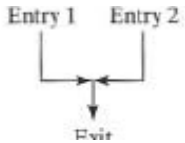
- **Stanje** – pravokotnik, ki opisuje stanje FSM. Ekvivalent v diagramu prehajanja stanj je mehurček, oziroma vrstica v tabeli prehajanja stanj. Izhodi Moore-ovega tipa so vpisani v pravokotnik. Običajno pišemo vanj samo signale, ki se spreminjajo.
- **Odločitev**
  - **Skalarna**
  - **Vektorska**
- **Pogojni izhod**
  oz. izhod Mealy-evega tipa (odvisen od stanja in vhodov FSM).

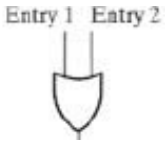


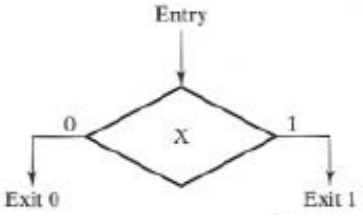
ASM diagrami

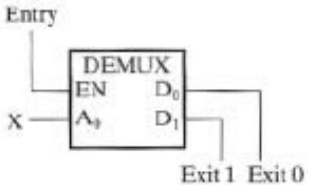


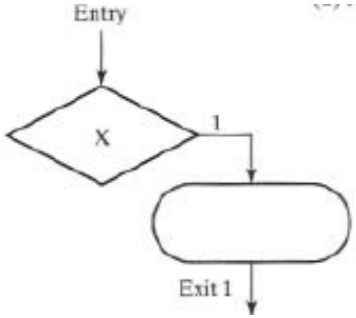


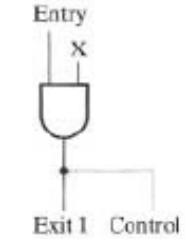


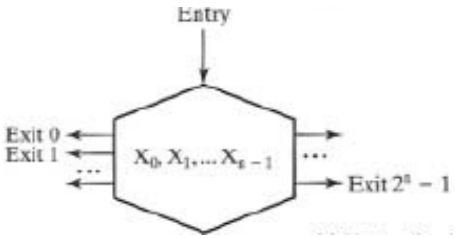


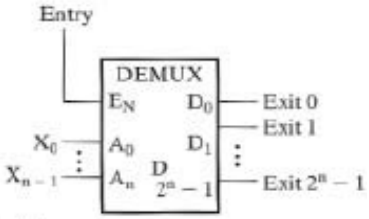




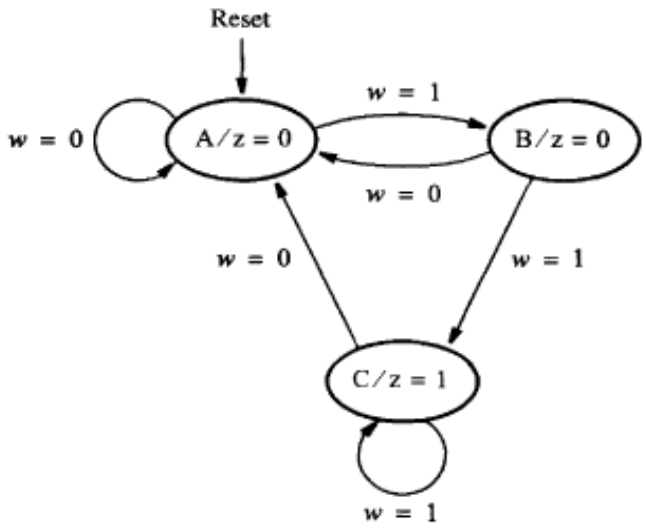




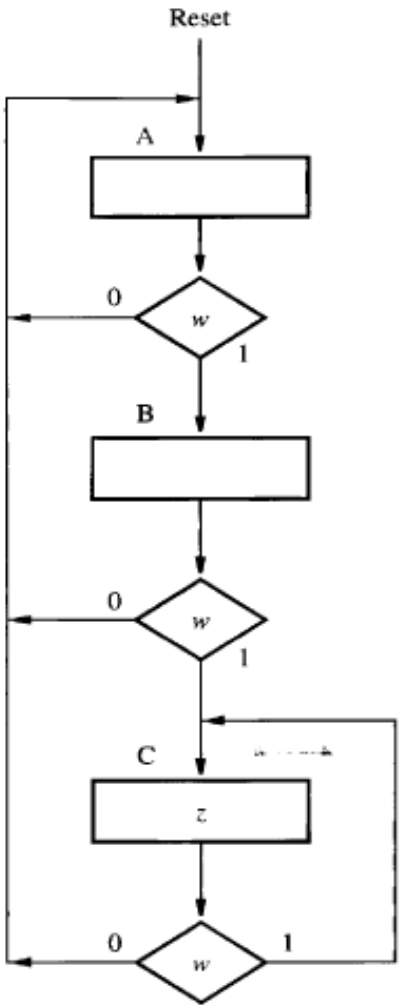




Opis FSM z ASM diagramom



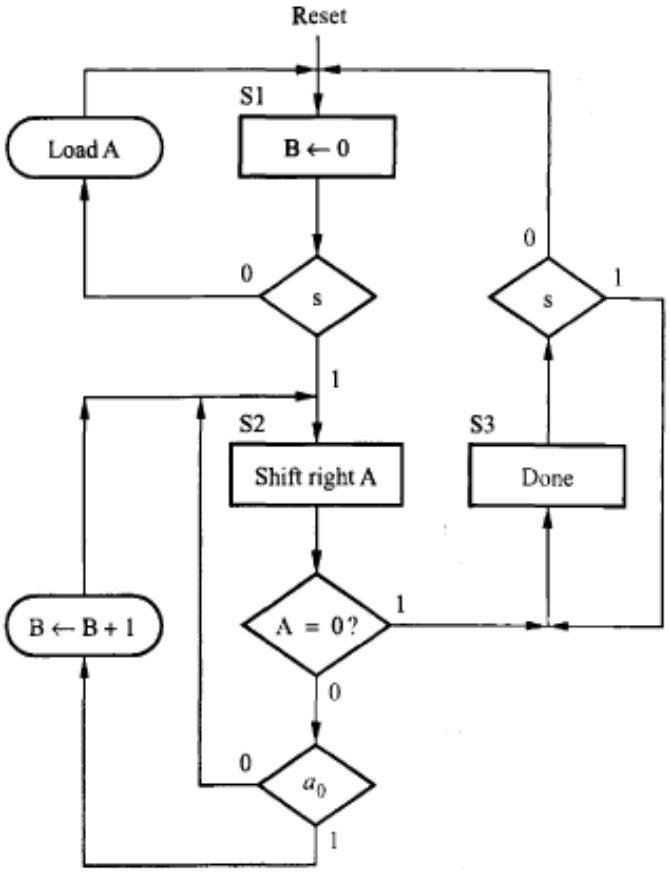
Zgornji Moore-ov avtomat je podan z ASM diagramom na desni.



Psevdo-koda in ASM diagram

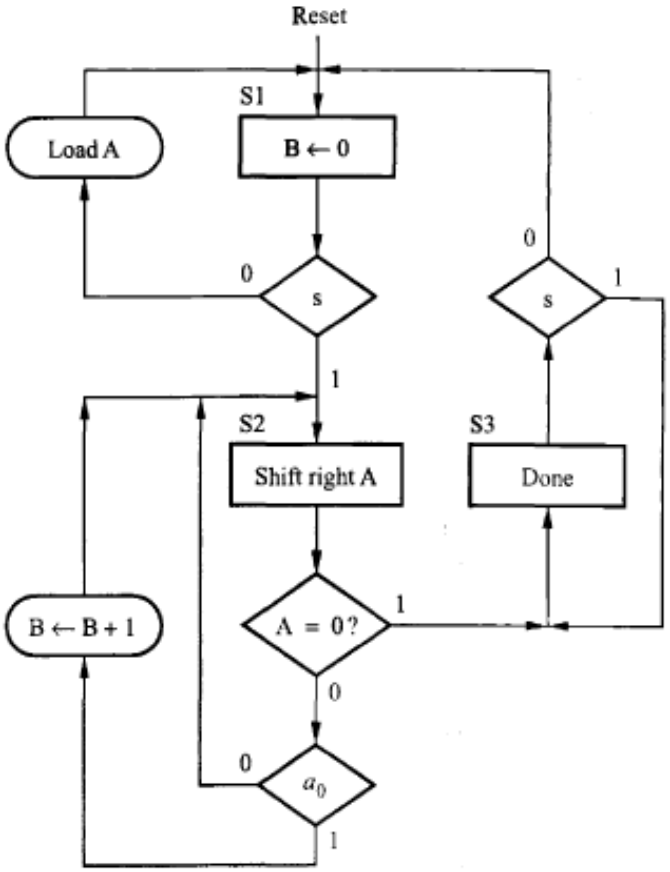
Vezje, ki šteje število '1' v registru A.

```
S1  B=0;
    while A <> 0 do
          if a(0) = 1 then
                B←B+1 ;
          end if;
S2        Right-shift A ;
S3  end while;
```



Izvajanje ASM diagrama

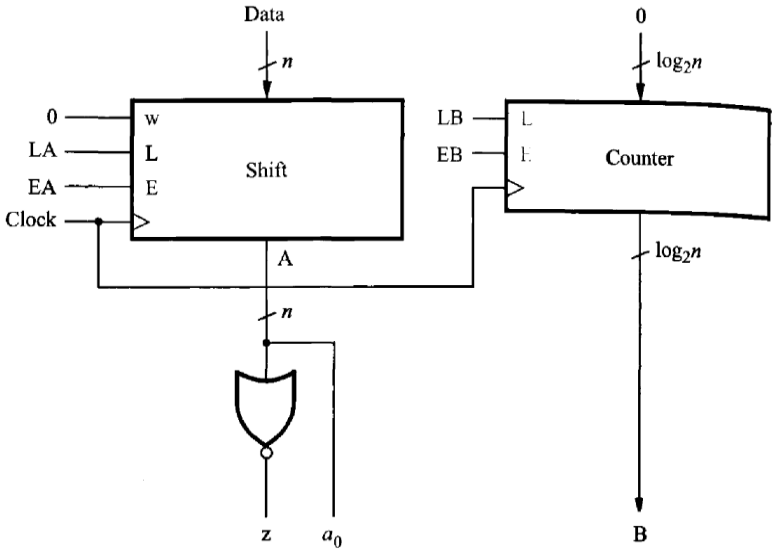
- Razlika med navadnim diagramom poteka in ASM diagramom je v časovnem poteku izvajanja.
- V navadnem diagramu poteka se vrednost A *najprej* pomakne desno, nato se ovrednoti a(0) in *nato* prišteje B←B+1.
- Prehode v ASM diagramu določa *aktivni rob signala ure*. Ta določa spremembo obeh registrov A in B kot tudi prehajanje stanj.
- V stanju S2 se odločitvi *A=0* in *a(0)=1* izvedeta kombinacijsko (hkrati).
- To se lahko zgodi, ker ob pomikanju desno z MSB strani postavimo '0', zato bo vseeno kateri pogoj se izvede prvi.

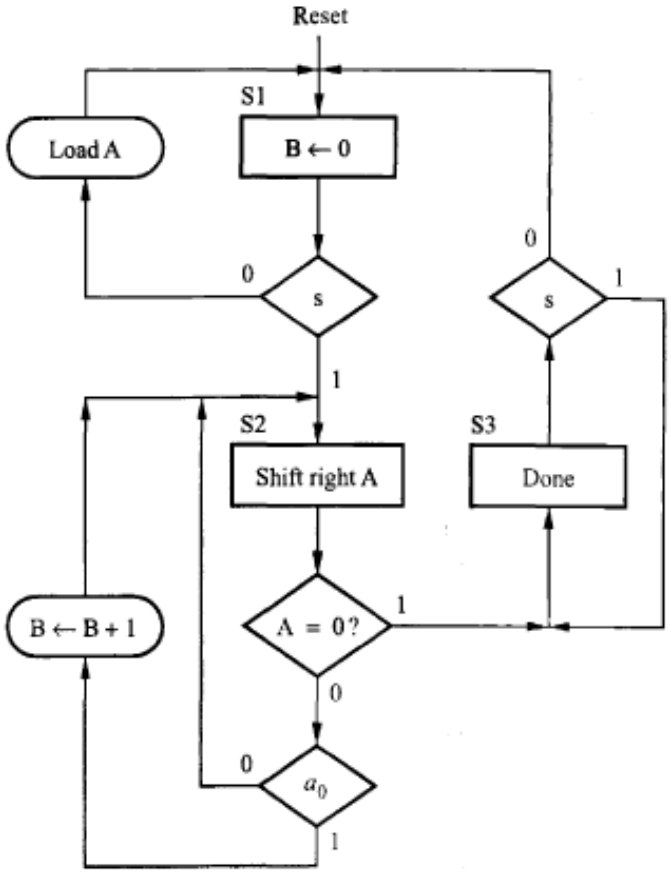


Podatkovna pot (datapath)

Potrebni vezji:

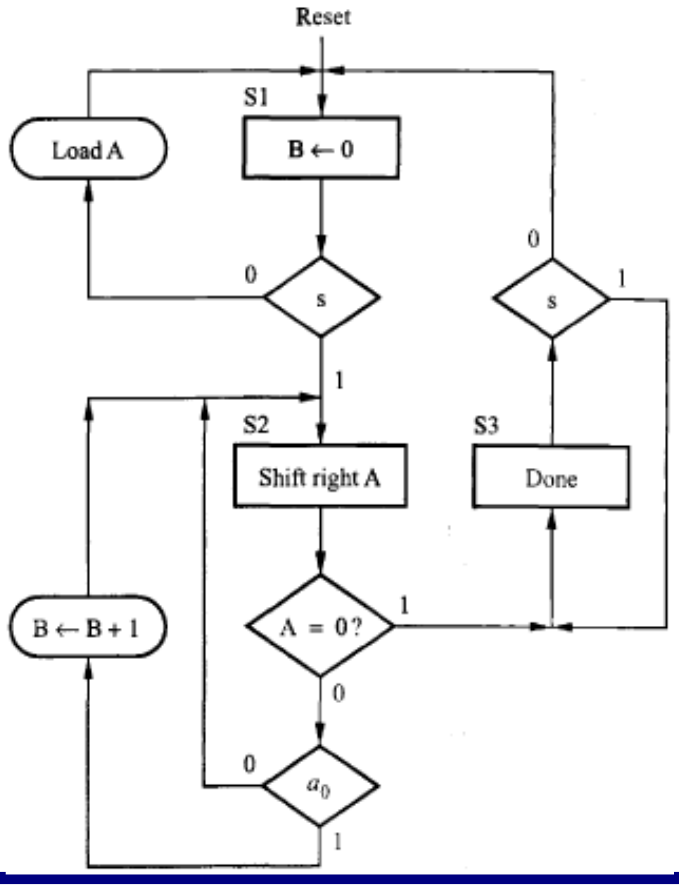
- *pomikalni register* desno z LOAD (LA) in ENABLE (EA)
- *števec* z LOAD (LB) in ENABLE (EB) (ne nRESET).

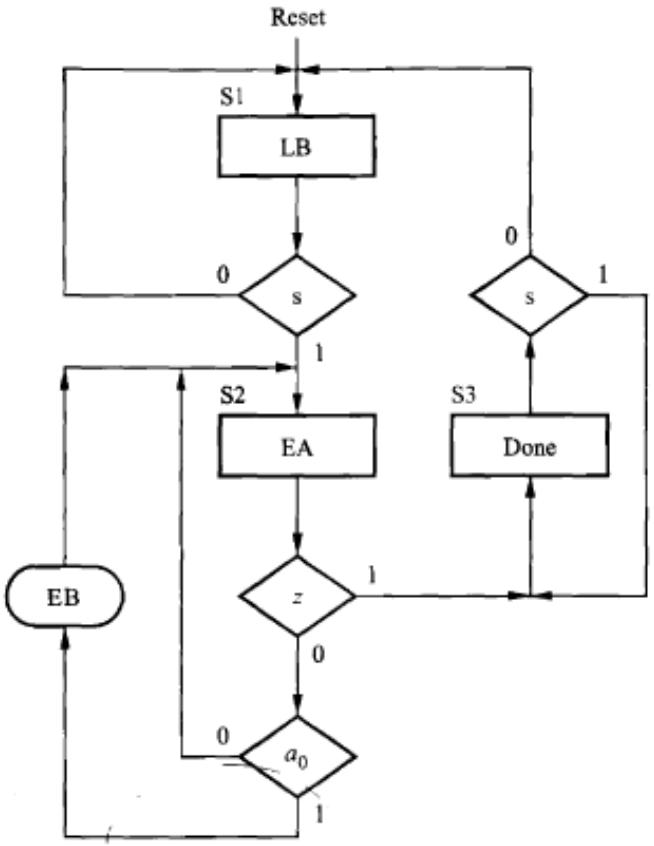




Nadzorno vezje (control circuit)

Nadzorno vezje je avtomat opisan z ASM diagramom, ki realizira krmilne signale pomikalnega registra (EA, LA) in števca (EB, LB)





Realizacija ASM v VHDL - entiteta

```
ENTITY bit_counter IS
generic( bit_counter_size : natural := 4);
PORT(    clk,
         nRST,
         LOAD_DATA,
         START    : IN STD_LOGIC;
         Data     : IN STD_LOGIC_VECTOR (2**bit_counter_size - 1 DOWNTO 0);
         B        : OUT STD_LOGIC_VECTOR (bit_counter_size - 1 DOWNTO 0);
         Done     : OUT STD_LOGIC);
END bit_counter;
```

Vezja podatkovne poti

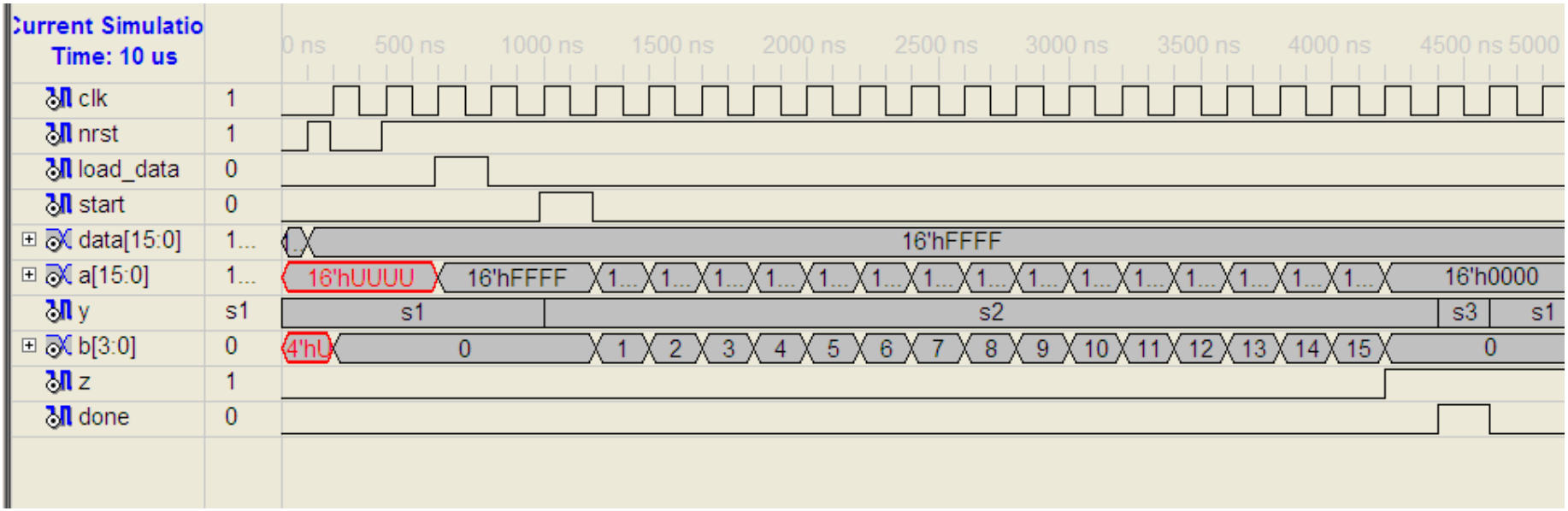
```
SR: shift_reg
GENERIC MAP( reg_size => 2**bit_counter_size)
PORT MAP (            clk =>      clk,
          nCLR =>   '1',
          sr_in => '0',
          sl_in => '0',
          s => sr_mode,
          x => Data,
          Q => A );

    LA <= LOAD_DATA;
    sr_ctrl <= LA & EA;

    with sr_ctrl select
    sr_mode <=      "11" when "11" | "10",          -- 1  X: Vzporedno nalaganje x => Q
                    "01" when "01",       -- 0  1: Pomikanje desno (v smeri od MSB do LSB)
                    "00" when others;     -- 0  0: Držanje vsebine

    z <= '1' WHEN A = zeroes ELSE '0';     -- (A=0?)

    CTR: counter
    GENERIC MAP( ctr_size => bit_counter_size)
    PORT MAP (      clk => clk,             -- signal ure
          nCLR => '1',           -- signal za brisanje števca (aktiven '0')
          nLOAD  => not(LB), -- signal za nalaganje števca (aktiven '0')
          ENP => EB,  ENT => EB,          -- signala za omogočanje štetja
          x => (others => '0'), -- vhod za vzporedno nalaganje
          Q => B);   -- izhodno štetje
```

Kontrolno vezje

```
FSM_transitions: PROCESS (nRST, clk)
BEGIN
    IF nRST = '0' THEN
        y <= S1;
        ELSIF (rising_edge(clk)) THEN
        CASE Y IS
        WHEN S1 =>
                IF START = '0' THEN
                        y <= S1;
                ELSE
                        y <= S2;
                END IF;
        WHEN S2 =>
                IF z = '0' THEN
                        y <= S2;
                ELSE
                        y <= S3;
                END IF;
        WHEN S3 =>
                IF START = '1' THEN
                        y <= S3;
                ELSE
                        y <= S1;
                END IF;
        END CASE;
    END IF;
    END PROCESS;
```

```
FSM_outputs: PROCESS ( y, A(0) )
BEGIN
                EA <= '0';
                LB <= '0';
                EB <= '0';
                Done <= '0';
                CASE y IS
                WHEN S1 =>
                        LB <= '1';
                WHEN S2=>
                        EA <= '1';
                        IF A(0) = '1' THEN
                                EB <= '1';
                        ELSE
                                EB <= '0';
                        END IF;
                WHEN S3 =>
                        Done <= '1';
                END CASE;
END PROCESS;
```

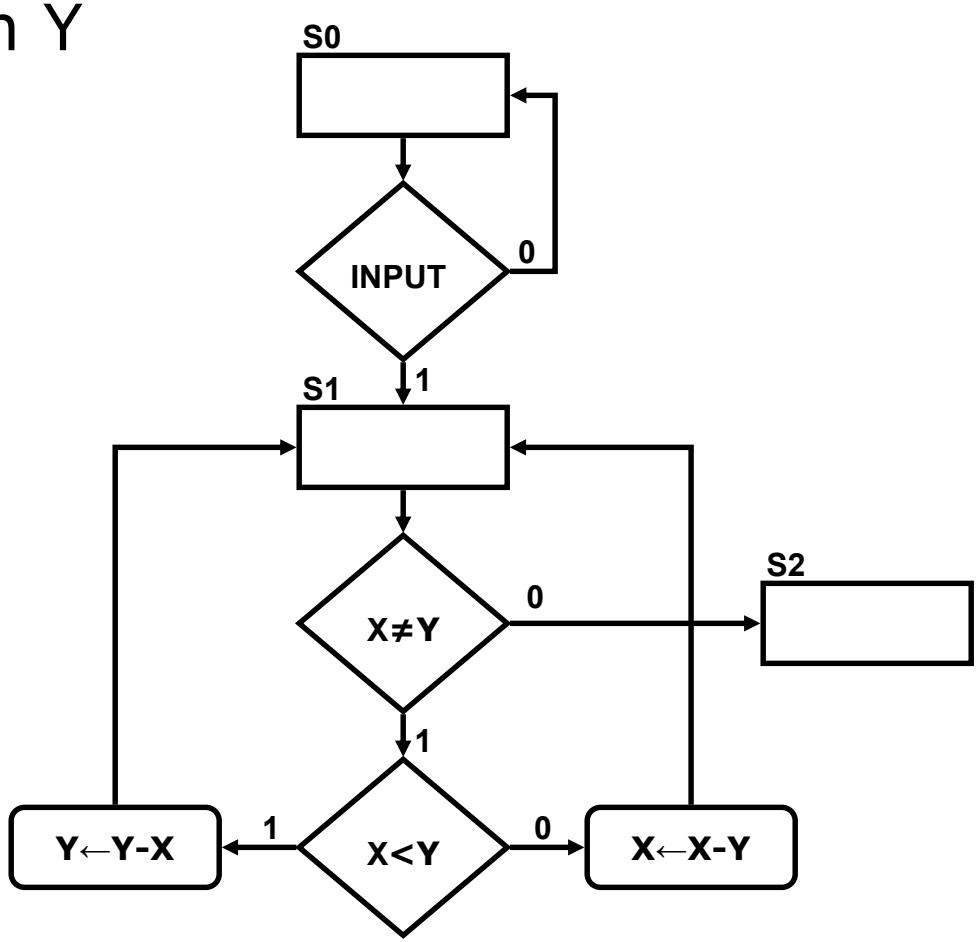
Rezultat simulacije v VHDL



ASM primer - Največji skupni delitelj (GCD)

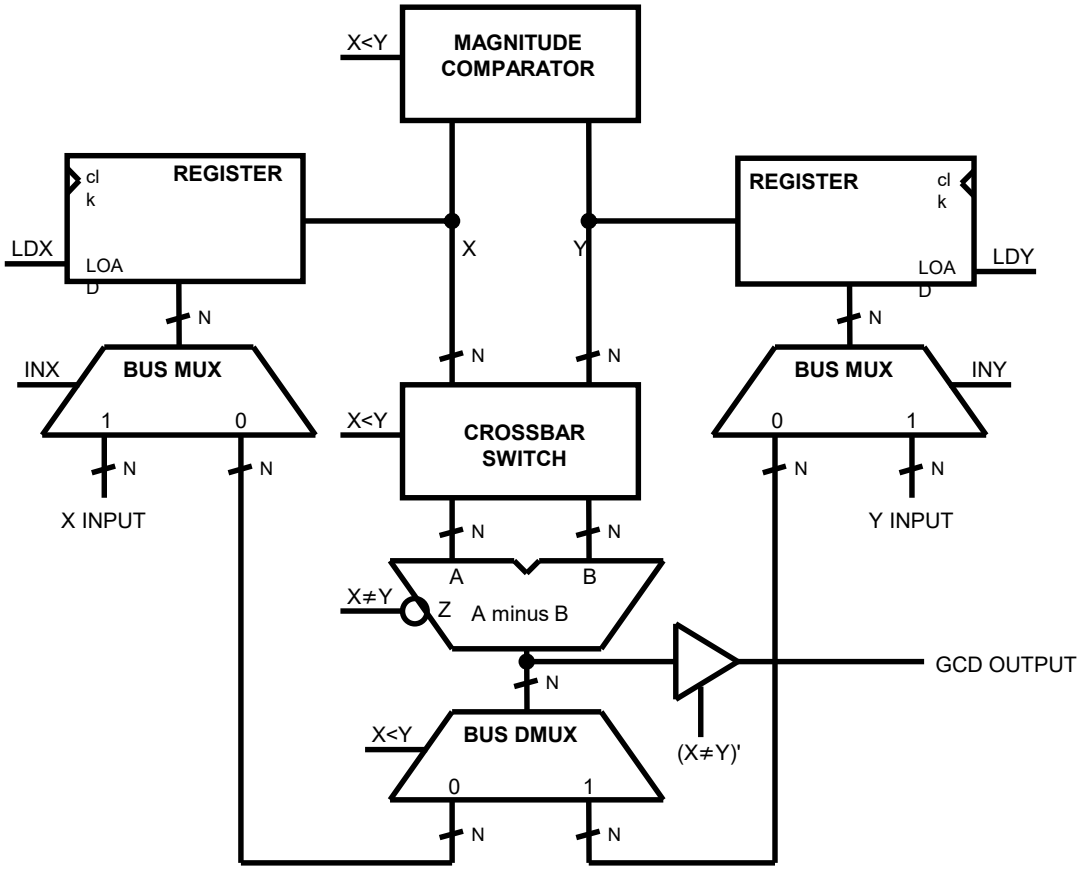
Vezje, ki izračuna največji skupni delitelj števil X in Y

```
S0:     INPUT X, Y
        WHILE (X ≠ Y) {
S1:             IF (X < Y) THEN
                        Y = Y - X;
                ELSE
                        X = X - Y;
                END IF
        }
S2:     OUTPUT X
```



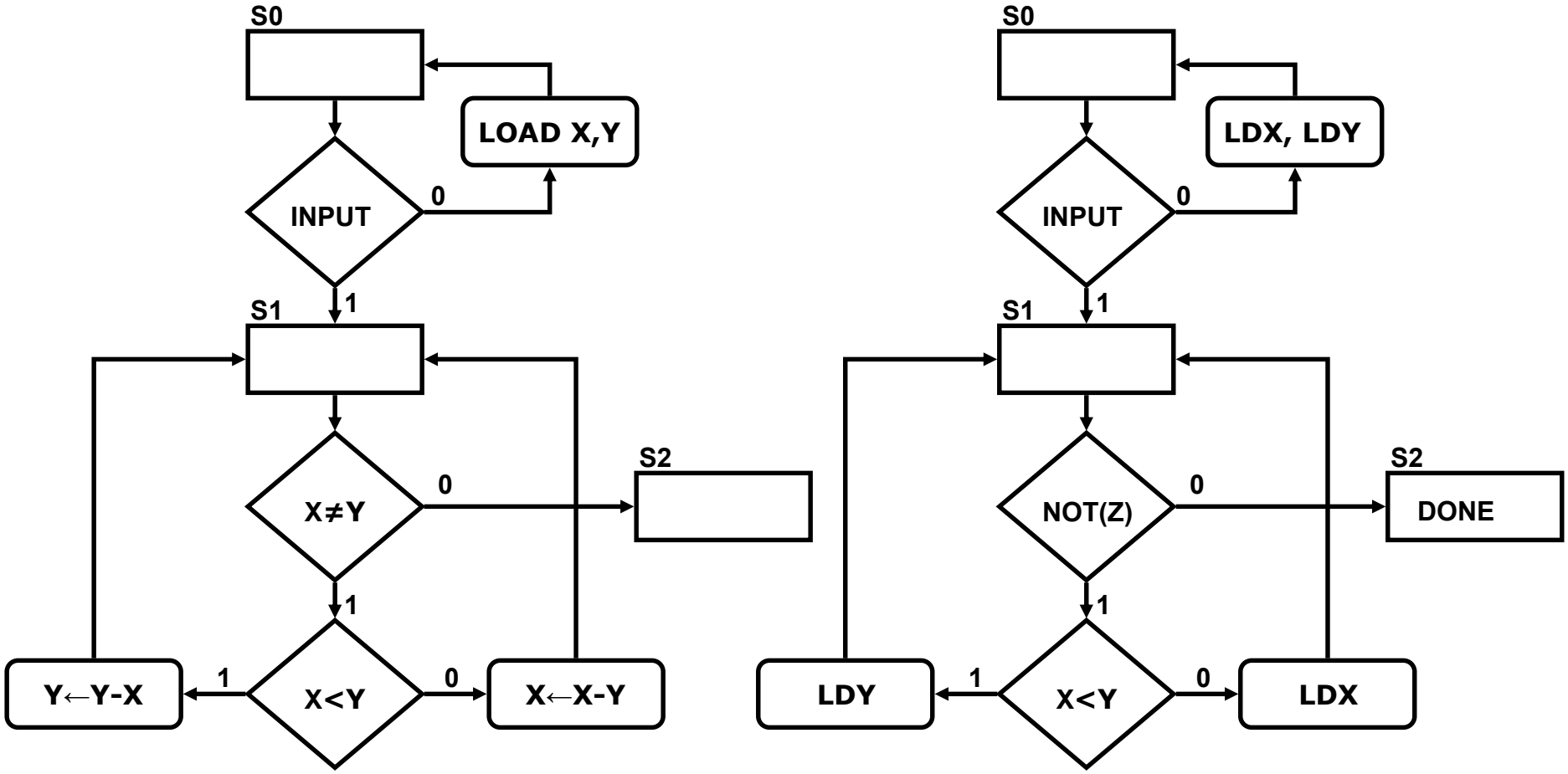
GCD-podatkovna pot (datapath)

- Za vnos (input) števil (x,y) služi krmilni signal (IN).
- Primerjavi enakosti števil (x) in (y) služi signal (XeqY),
- primerjavi x "manjše od" y služi signal (XltY).
- Za omogočanje izhoda (ukaz output) rezultata (result) uporabimo krmilni signal (DONE).



GCD - nadzorno vezje (control circuit)

Nadzorno vezje je avtomat opisan z ASM diagramom, ki realizira krmilne signale registra (X, Y)



GCD v VHDL - entiteta

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

   entity gcd is
   generic(n   : natural := 8);          -- register size parameter
   PORT (      clk,      -- clock input
         load,   -- parallel data load signal   (active '1')
         nRST    : in std_logic; -- reset input  (active '0')
         xinput, -- x data input
         yinput : in      std_logic_vector(n - 1 downto 0);   -- y data
   input
         result  : out    std_logic_vector(n - 1 downto 0);   -- result
   output
         done    : out std_logic          -- done (active '1')
   );
   end gcd;
```

GCD - Vezja podatkovne poti (I. del)

```
architecture ideal of gcd is
signal XeqY, XltY : std_logic;
signal ldx, ldy              : std_logic;
signal done_sig              : std_logic;
signal x_reg_mode, y_reg_mode       : std_logic_vector( 1 downto 0 );
signal Xload : std_logic_vector(n - 1 downto 0);     -- parallel load input for x register
signal Yload : std_logic_vector(n - 1 downto 0);     -- parallel load input for y register
signal X                     : std_logic_vector(n - 1 downto 0);     -- x register contents
signal Y                     : std_logic_vector(n - 1 downto 0);     -- y register contents
signal A                     : std_logic_vector(n - 1 downto 0);     -- ALU input A
signal B                     : std_logic_vector(n - 1 downto 0);     -- ALU input B
signal D                     : std_logic_vector(n - 1 downto 0);     -- ALU subtractor output (difference A - B)

type gcd_asm_state_type is ( S0, S1, S2 );
signal gcd_asm_state: gcd_asm_state_type;
```

GCD - Vezja podatkovne poti (II. del)

```
type gcd_asm_state_type is ( S0, S1, S2 );
signal gcd_asm_state: gcd_asm_state_type;

component shift_reg is
        generic( reg_size: natural := 4);
        PORT (clk, nCLR, sr_in, sl_in : IN std_logic;
                s : in std_logic_vector(1 downto 0);
                x : in std_logic_vector(reg_size - 1 downto 0);
                Q : out std_logic_vector(reg_size - 1 downto 0)
                );
end component;

component ndn_alu
        generic( n: natural := 8 );
        port(   M                       :       in      std_logic;
                --naèin delovanja ('0' => aritmetièni, '1' => logièni)
                F                       :       in      std_logic_vector(2 downto 0);
                -- funkcijski vhod za operacije
                X, Y            :       in      std_logic_vector(n-1 downto 0);
                S                       :       out std_logic_vector(n-1 downto 0);
                Negative,
                Cout,
                Overflow,
                Zero,
                Gout,
                Pout            :       out     std_logic );
end component;
```

GCD - Povezovanje podatkovne poti (I. del)

```
ALU:        ndv_alu
            generic map(n => n)
            port map(
                        M => '0',     -- arithmetic mode
                        F => "001", -- subtract operation       (X - Y)
                        X => A,       -- first operand
                        Y => B,       -- second operand
                        S => D,       -- result
                        Zero => XeqY); -- zero bit is set to '1' when X equals Y

x_reg_mode <= (others => ldx);       -- load ("11") when (ldx = '1'), else hold ("00")
XREG: shift_reg
            generic map(reg_size => n)
            port map(
                        clk => clk,
                        nCLR => nRST,
                        sr_in => '0',
                        sl_in => '0',
                        s => x_reg_mode,
                        x => Xload,
                        Q => X);

y_reg_mode <= (others => ldy);       -- load ("11") when (ldy = '1'), else hold ("00")
YREG: shift_reg
            generic map(reg_size => n)
            port map(
                        clk => clk,
                        nCLR => nRST,
                        sr_in => '0',
                        sl_in => '0',
                        s => y_reg_mode,
                        x => Yload,
                        Q => Y);
```

GCD - Povezovanje podatkovne poti (II. del)

```
XltY <=  '1'          when (X < Y)       else '0';  --magnitude comparator
result <= Y           when (done_sig = '1') else (others => 'Z'); -- 3-state output buffer
Xload <= Xinput       when (load = '1') else D;     -- bus multiplexer for x register
Yload <= Yinput       when (load = '1') else D;     -- bus multiplexer for y register
A <=      Y           when XltY = '1'    else X;    -- crossbar bus switch
B <=      X           when XltY = '1'    else Y;    -- crossbar bus switch
```

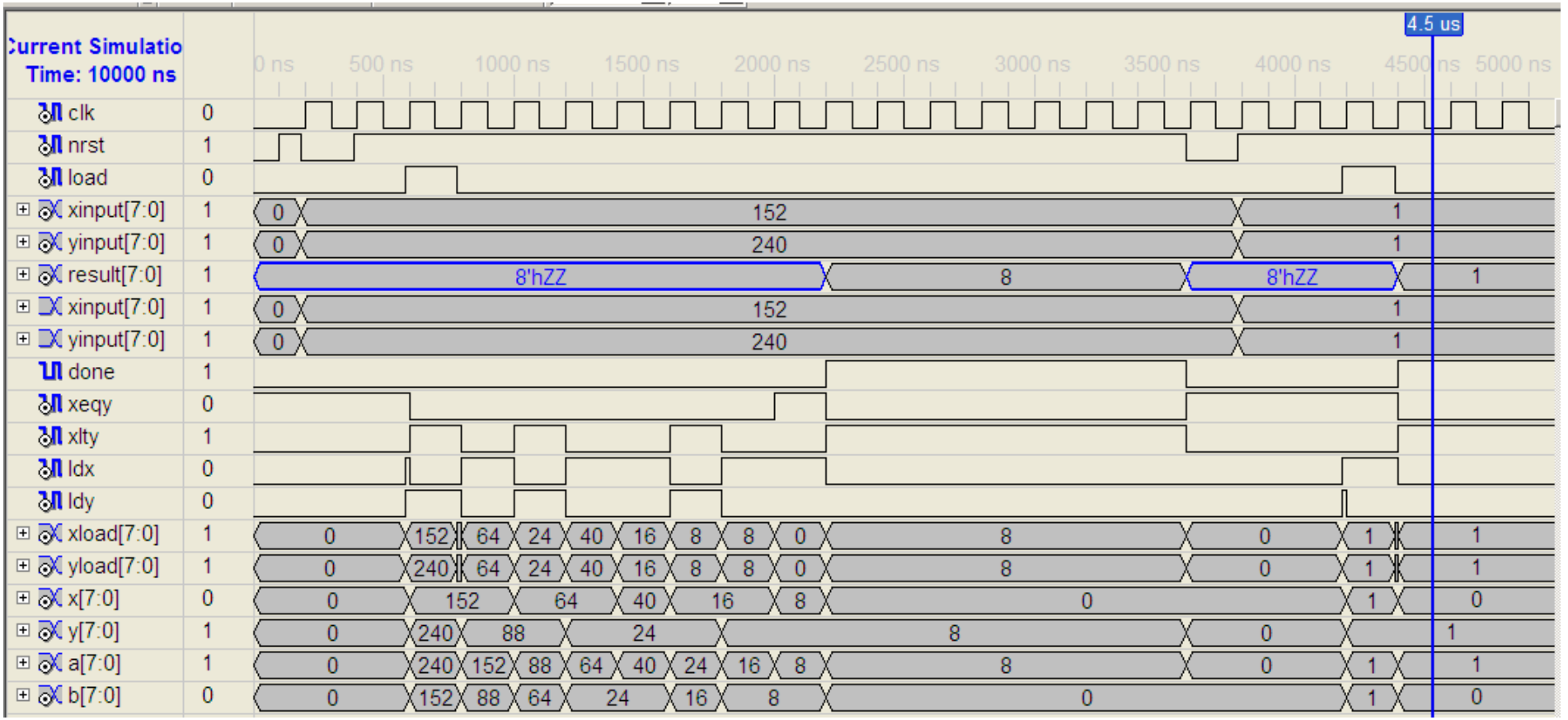
GCD - Nadzorno vezje (I. del)

```
GCD_ASM: process(clk, nRST, load, XeqY)
begin
        if (nRST = '0') then
        gcd_asm_state <= S0;        -- initial state is S0
        elsif rising_edge(clk) then
        case gcd_asm_state is
                when S0 =>
                if (load = '1') then
                        gcd_asm_state <= S1;
                        -- upon load goto S1, else wait for load in S0
                end if;
                when S1 =>
                if (XeqY = '1') then
                        gcd_asm_state <= S2;
                        -- done state, output result
                end if;
                when others =>
                gcd_asm_state <= S2;
                -- done, stay put
        end case;
        end if;
end process;
```

GCD - Nadzorno vezje (II. del)

```
--Mealy type outputs for loading registers X and Y
ldx <= '1' when   ((gcd_asm_state = S1) and (XltY = '0')) or
                  ((gcd_asm_state = S0) and (load = '1')) else '0';

ldy <= '1' when   ((gcd_asm_state = S1) and (XltY = '1')) or
                  ((gcd_asm_state = S0) and (load = '1')) else '0';

--Moore type output for signaling finished state
done_sig <= '1' when (gcd_asm_state = S2) else '0';
done <= done_sig; -- relay last equality to output
```

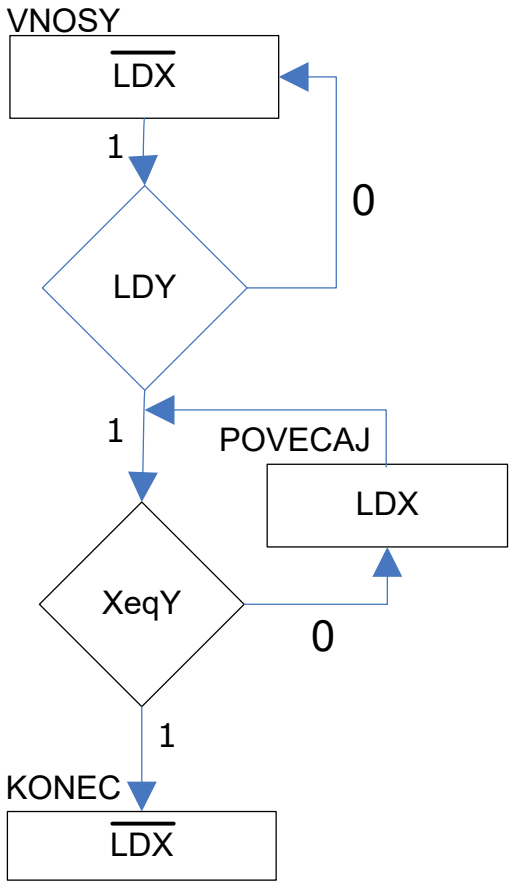
GCD - rezultat simulacije v VHDL



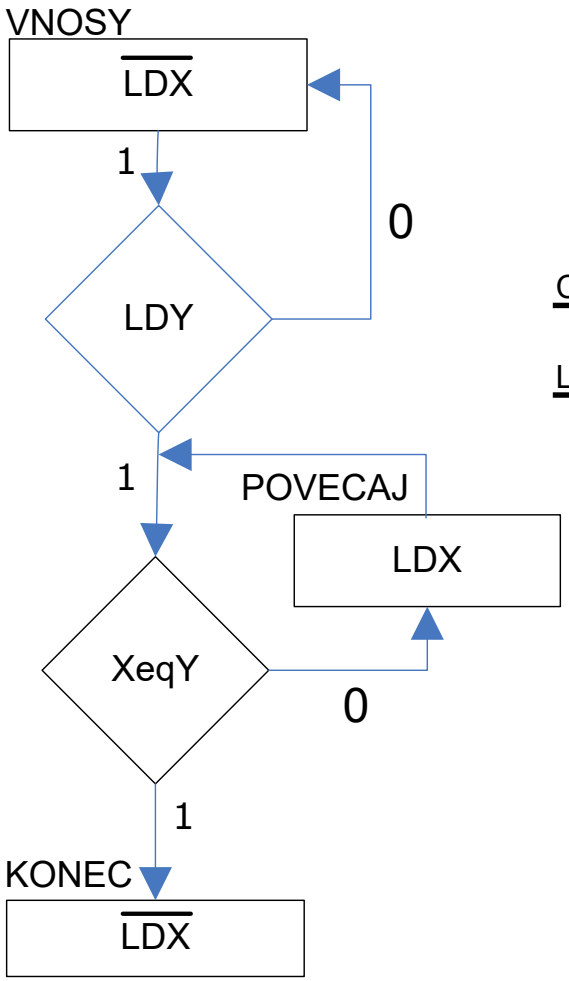
Generator zaporedja:: nadzorno vezje

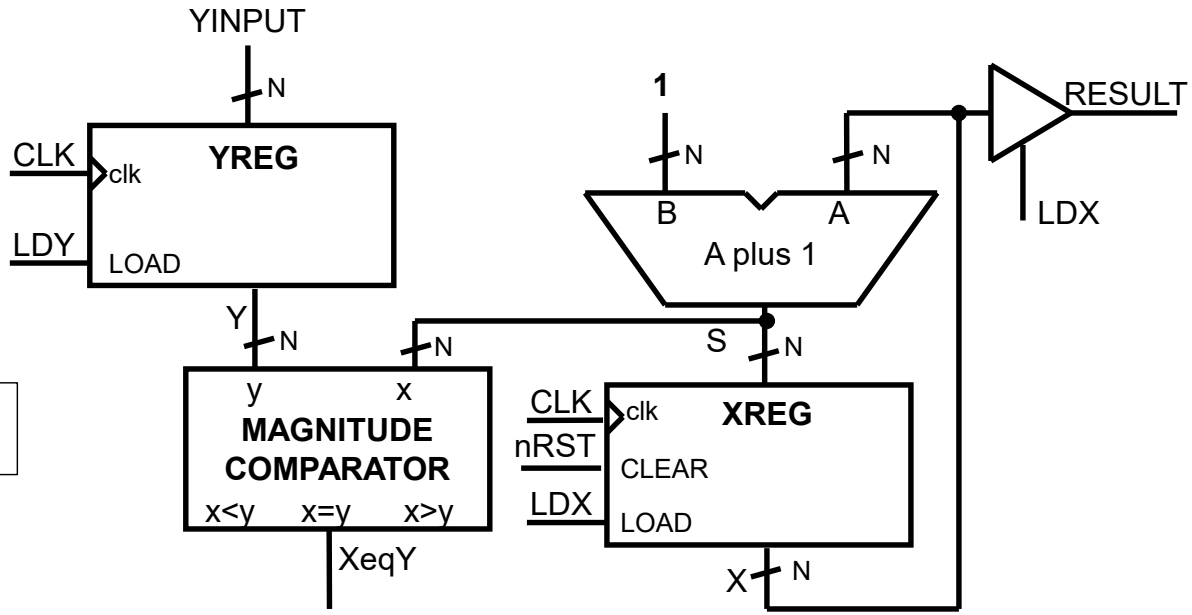
Vezje, ki na izhodu x tvori zaporedje 0… y-1,
če vnesemo y.
Po zadnji vrednosti izhod x preide v stanje visoke impedance.

```
x = 0;
input y;
while (x ≠ y) {
      x = x + 1;
      output x;
}
```



Generator zaporedja::podatkovna pot





Generator zaporedja v VHDL

```
ibrary IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity cnt_seq is
          generic( n : natural := 8);   -- size parameter
          PORT (    clk, -- clock input
                    ldy, -- parallel data load signal        (active '1')
                    nRST      : in std_logic; -- reset input(active '0')
                    yinput    : in        std_logic_vector(n - 1 downto 0); -- y data input
                    result    : out       std_logic_vector(n - 1 downto 0)
                    -- sequence (0 ... y-1)
          );
end cnt_seq;
```

Generator zaporedja – deklaracija komponent podatkovne poti

```
architecture ideal of cnt_seq is
signal XeqY          : std_logic;          --equality comparator (X=Y) signal (active '1')
signal ldx           : std_logic;          -- x register parallel load signal (active '1')
signal x_reg_mode, y_reg_mode     : std_logic_vector( 1 downto 0 );
signal X  : std_logic_vector(n - 1 downto 0);  -- x register contents
signal Y  : std_logic_vector(n - 1 downto 0);  -- y register contents
signal S  : std_logic_vector(n - 1 downto 0);  -- incrementer output (X + 1)
type gcd_asm_state_type is ( vnosy, povecaj, konec );
signal gcd_asm_state: gcd_asm_state_type;
component shift_reg is
          generic( reg_size: natural := 4);
          PORT (clk, nCLR, sr_in, sl_in : IN std_logic;
                    s : in std_logic_vector(1 downto 0);
                    x : in std_logic_vector(reg_size - 1 downto 0);
                    Q : out std_logic_vector(reg_size - 1 downto 0)
                    );
end component;
```

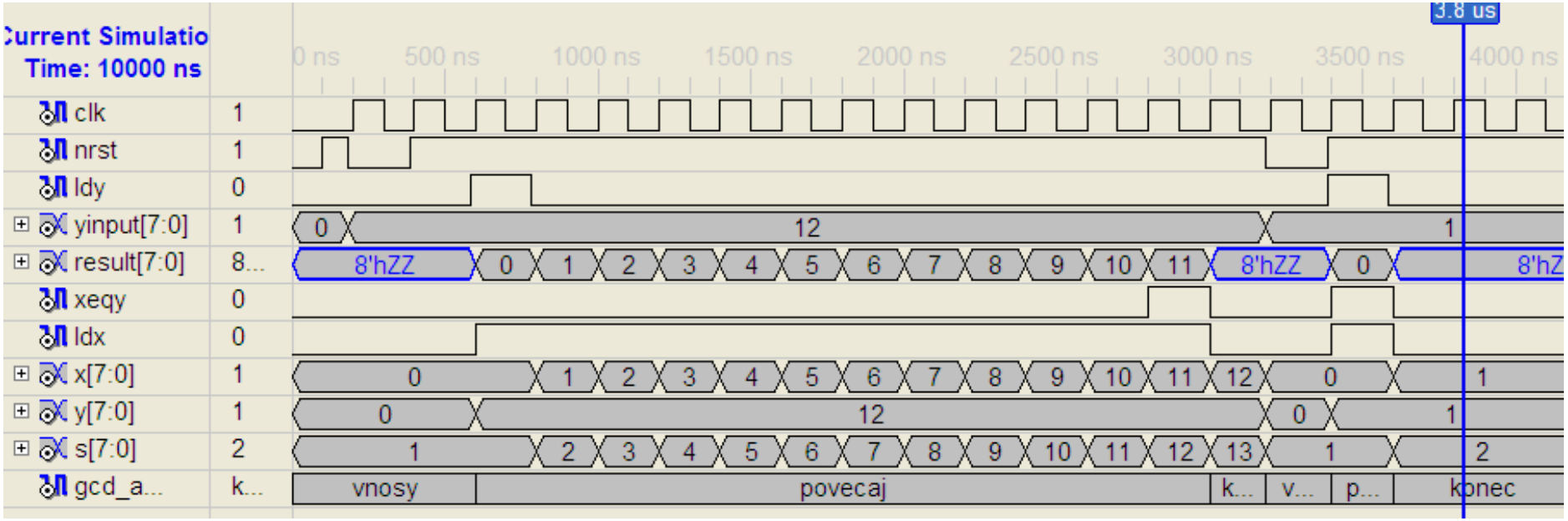
Generator zaporedja-povezovanje elementov podatkovne poti

```
x_reg_mode <= (others => ldx); -- load ("11") when (ldx = '1'), else hold ("00")

XREG: shift_reg
        generic map(reg_size => n)
        port map(
                clk => clk, nCLR => nRST, sr_in => '0', sl_in => '0',
                s => x_reg_mode,
                x => S,
                Q => X);

y_reg_mode <= (others => ldy); -- load ("11") when (ldy = '1'), else hold ("00")

YREG: shift_reg
        generic map(reg_size => n)
        port map(
                clk => clk,  nCLR => nRST,  sr_in => '0', sl_in => '0',
                s => y_reg_mode,
                x => yinput,
                Q => Y);

S <= X + 1;         -- incrementer / adder
XeqY <= '1'         when (S = Y)       else '0';  -- equality comparator
result <= X         when (ldx = '1') else (others => 'Z'); -- 3-state output buffer
```

Generator zaporedja - nadzorno vezje

```
CNT_SEQ_ASM: process(clk, nRST, ldy, XeqY)
begin
        if (nRST = '0') then
                gcd_asm_state <= vnosy;  -- initial state is vnosy
        elsif rising_edge(clk) then
        case gcd_asm_state is
                when vnosy =>
                if (ldy = '1') then
                        gcd_asm_state <= povecaj;-- upon load goto povecaj
                end if;
                when povecaj =>
                if (XeqY = '1') then
                        gcd_asm_state <= konec;  -- finished, disable output
                end if;
                when others =>
                gcd_asm_state <= konec;          -- stay here until new reset
        end case;
        end if;
end process;

ldx <= '1' when (gcd_asm_state = povecaj) else '0';    -- Moore type ldx output
```

Generator zaporedja - rezultat simulacije v VHDL



Načrtovanje digitalnih naprav

Mikroprocesor

Namenski:Splošni procesor

- Do sedaj smo načrtovali namenske mikroprocesorje z eno nalogo:
  - Nadzorno vezje is "fiksno ožičeno" in opravlja eno nalogo
  - Primer: povprečje, sortiranje, deljenje, CORDIC
- Splošni procesor lahko izvaja več nalog glede na program
  - Nadzorna enota ni fiksno ožičena
  - Bere ukaze in izvaja ukaze
  - Splošnost pa ima ceno v hitrosti – praviloma so ti procesorji počasnejši
    - "Programska implementacija" → funkcija se izvaja na splošnem procesorju
    - "Strojna implementacija" → funkcija se izvaja na namenskem procesorju
- Knjiga o procesorjih v VHDL:
  - Vahid and Givargis, "Embedded System Design: A Unified Hardware/Software Introduction"

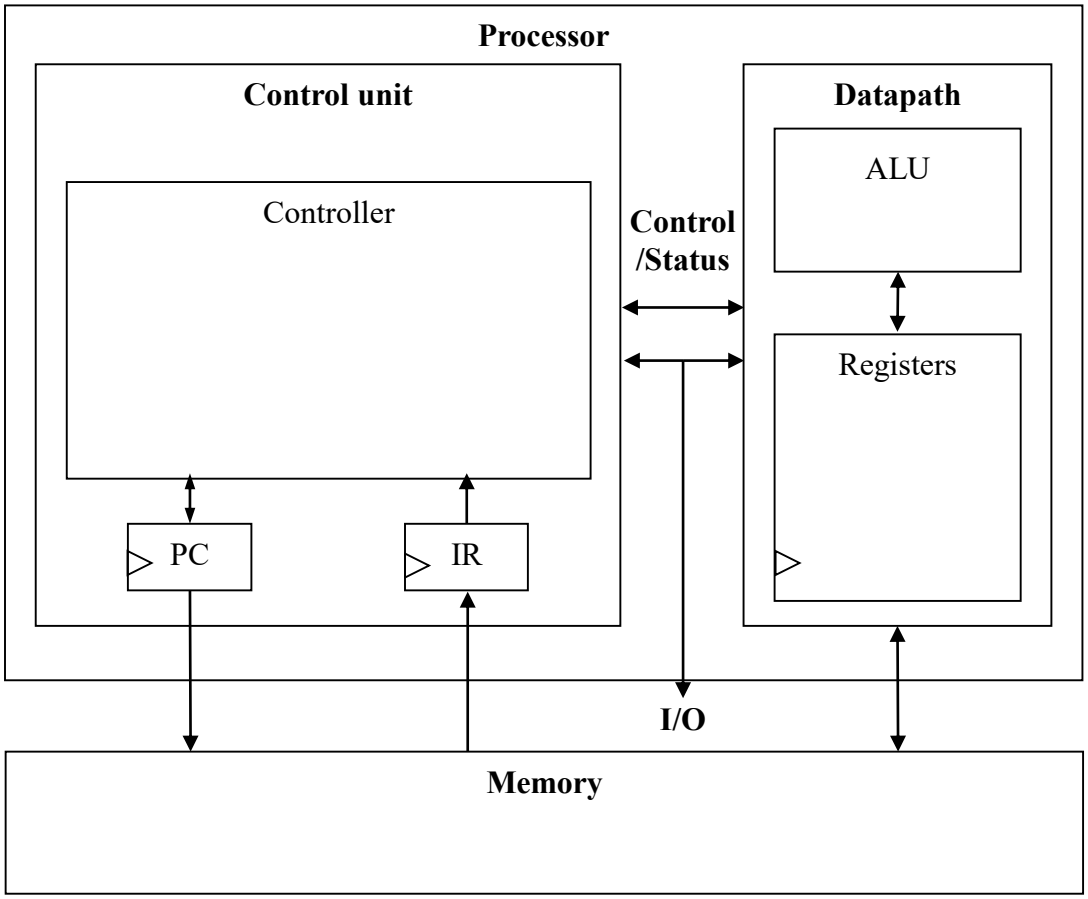
Uvod

- Splošni procesor
  - Namenjen raznolikim programskim nalogam
  - Poceni, saj se NRE stroški porazdelijo (non-recurring engineering cost) preko količine proizvedenih
    - Motorola je prodala pol milijarde 68HC05 microkrmilnikov v letu *1996*
  - Lahko si privoščimo zapleteno načrtovanje zaradi NRE
    - Zapleteno načrtovanje omogoči boljše delovanje, velikost, porabo moči
  - Nizki NRE (Non-recurring engineering) stroški, kratek čas "time-to-market", enostavna izvedba prototipa, visoka prilagodljivost
    - Uporabnik le piše program → ne načrtuje procesorja

Source: Vahid and Givargis, "Embedded System Design: A Unified Hardware/Software Introduction"

Osnovna arhitektura

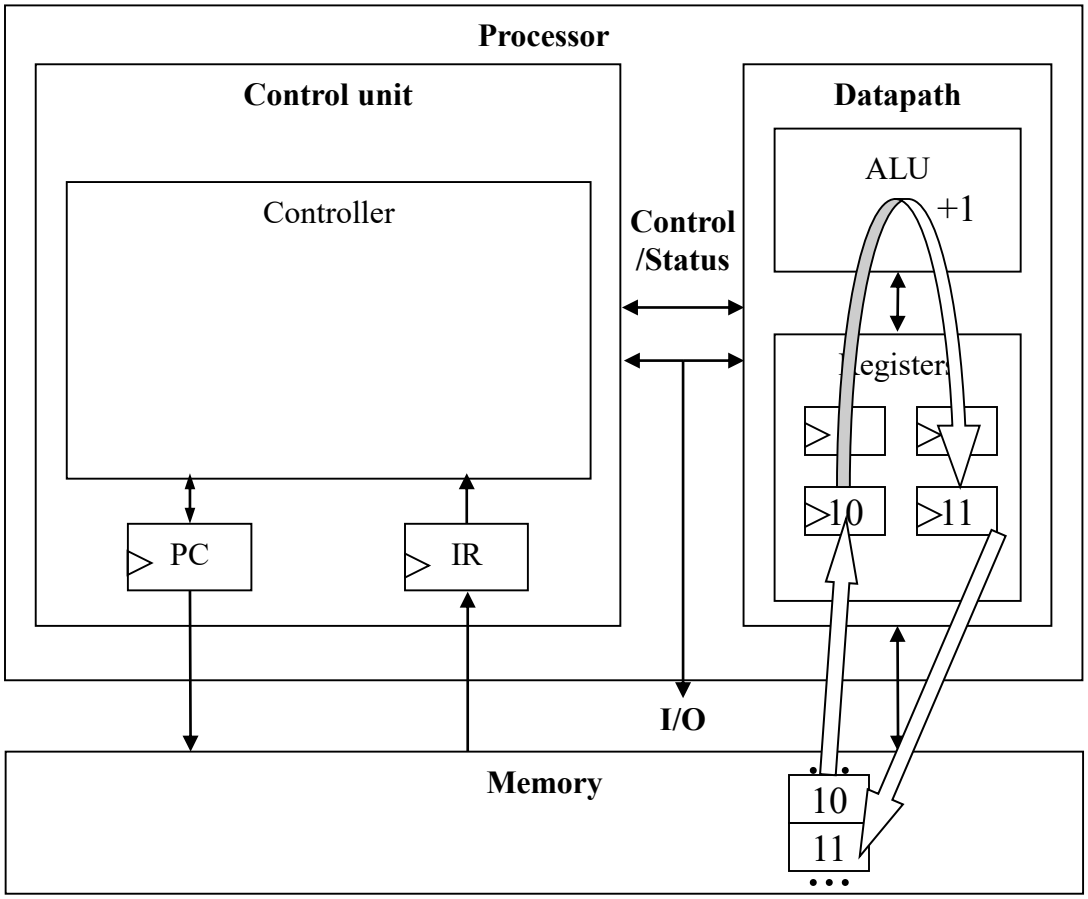
- Nadzorna enota (control unit) in podatkovna pot (datapath)
  - Podobnost z namenskim procesorjem
- Razlike
  - Podatkovna pot je splošna
  - Nadzorna enota ne hrani algoritma. Algoritem je programiran v spomin.



Source: Vahid and Givargis, "Embedded System Design: A Unified Hardware/Software Introduction"

Operacije podatkovne poti

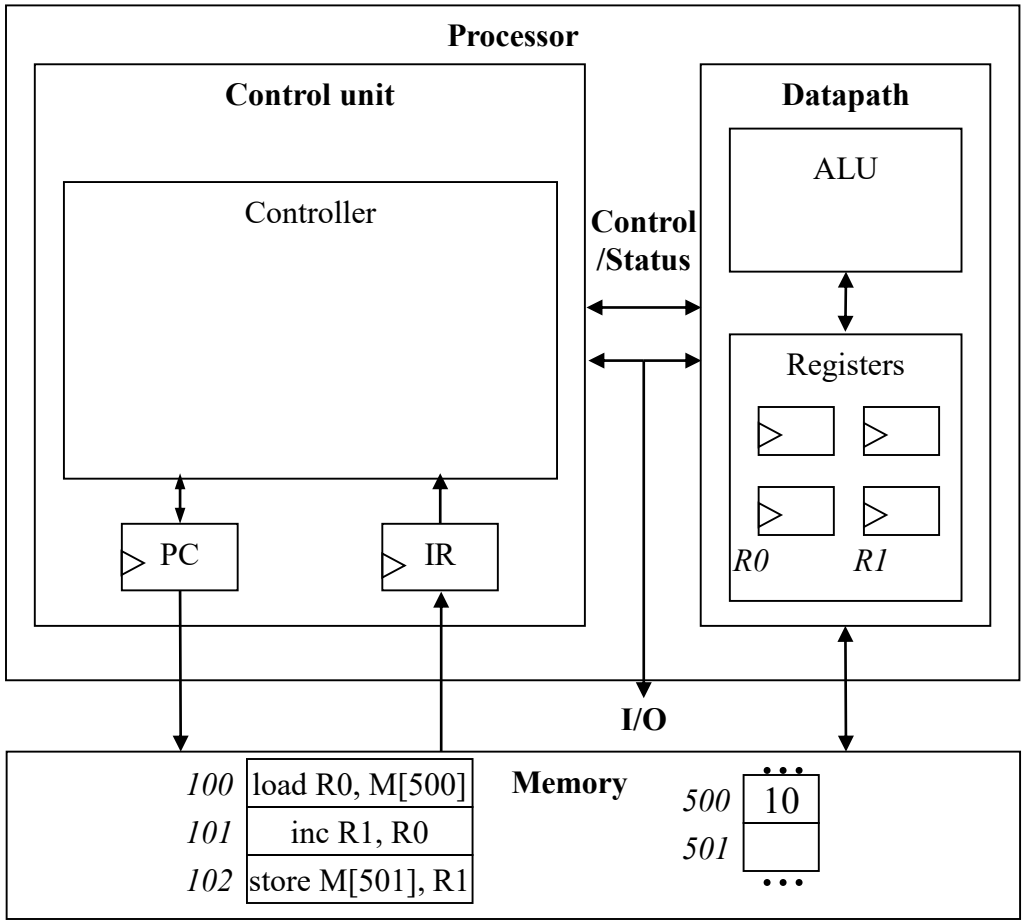
- Load
  - Beri spominsko lokacijo in piši v register
- ALU operacija
  - Na vhodu so registri, ALU izvede operacijo in rezultat se vpiše v register
- Store
  - Zapiši register na spominsko lokacijo



Source: Vahid and Givargis, "Embedded System Design: A Unified Hardware/Software Introduction"

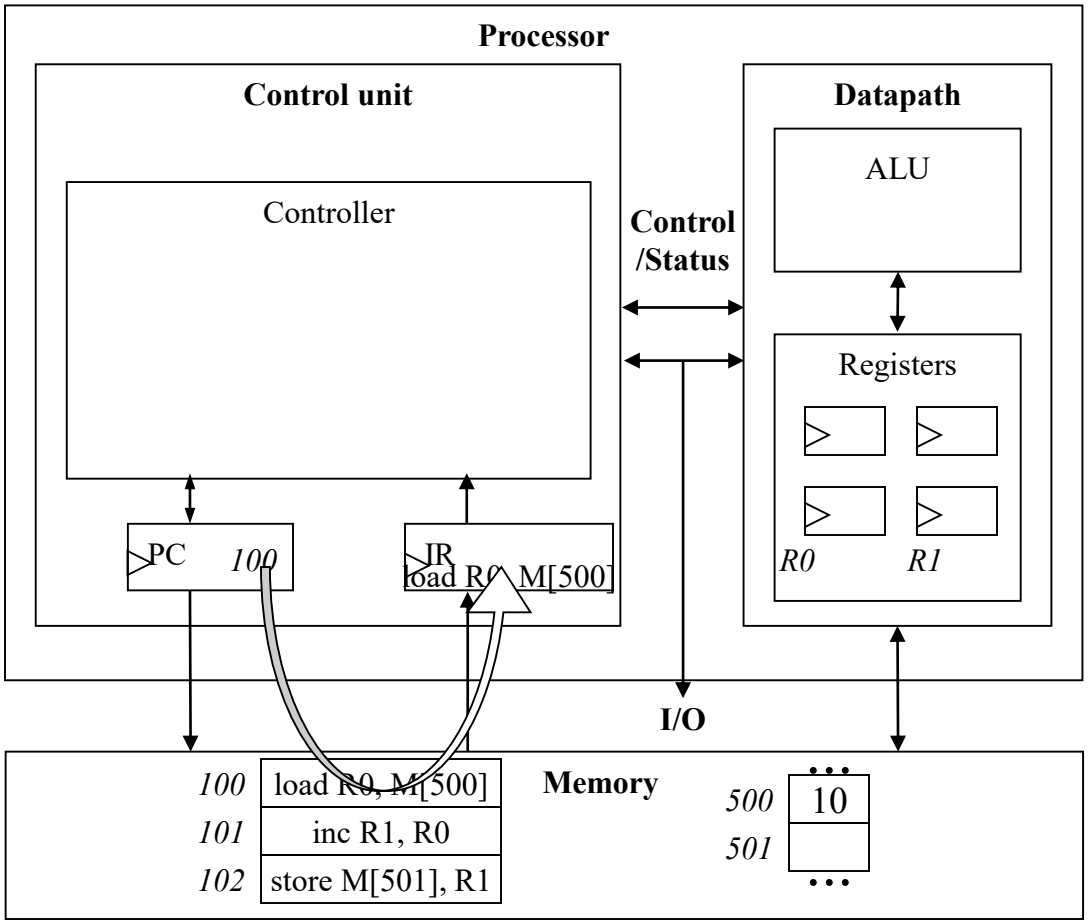
Nadzorna enota

- Nastavlja operacijo podatkovne poti
  - Zaporedje želenih operacij (ukazov), ki so shranjeni v programskem spominu
- Ukazni cikel – razdeljen v mikrooperacije, vsaka traja en cikel signala ure:
  - **Fetch**: Pridobi naslednji ukaz iz IR
  - **Decode**: Dekodiraj pomen ukaza
  - **Fetch operandov**: Premakni podatke iz spomina v ustrezne podatkovne registre
  - **Execute**: Izvedi operacijo z ALU
  - **Store**: Shrani rezultat iz registra nazaj v spomin



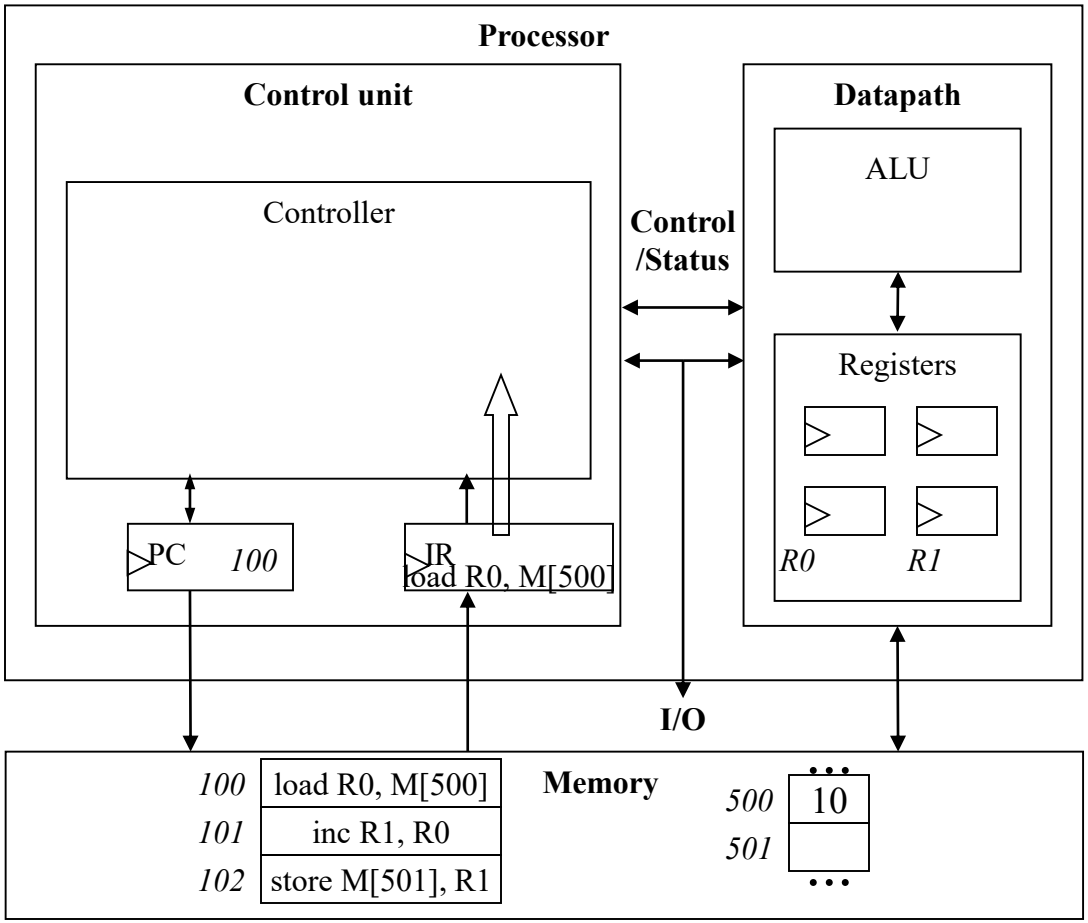
Mikrooperacije nadzorne enote

- Fetch
  - Naslednja operacija ➔ IR
  - PC: programski števec, vedno kaže na naslednji ukaz
  - IR: hrani trenutni ukaz (FETCHED instruction)



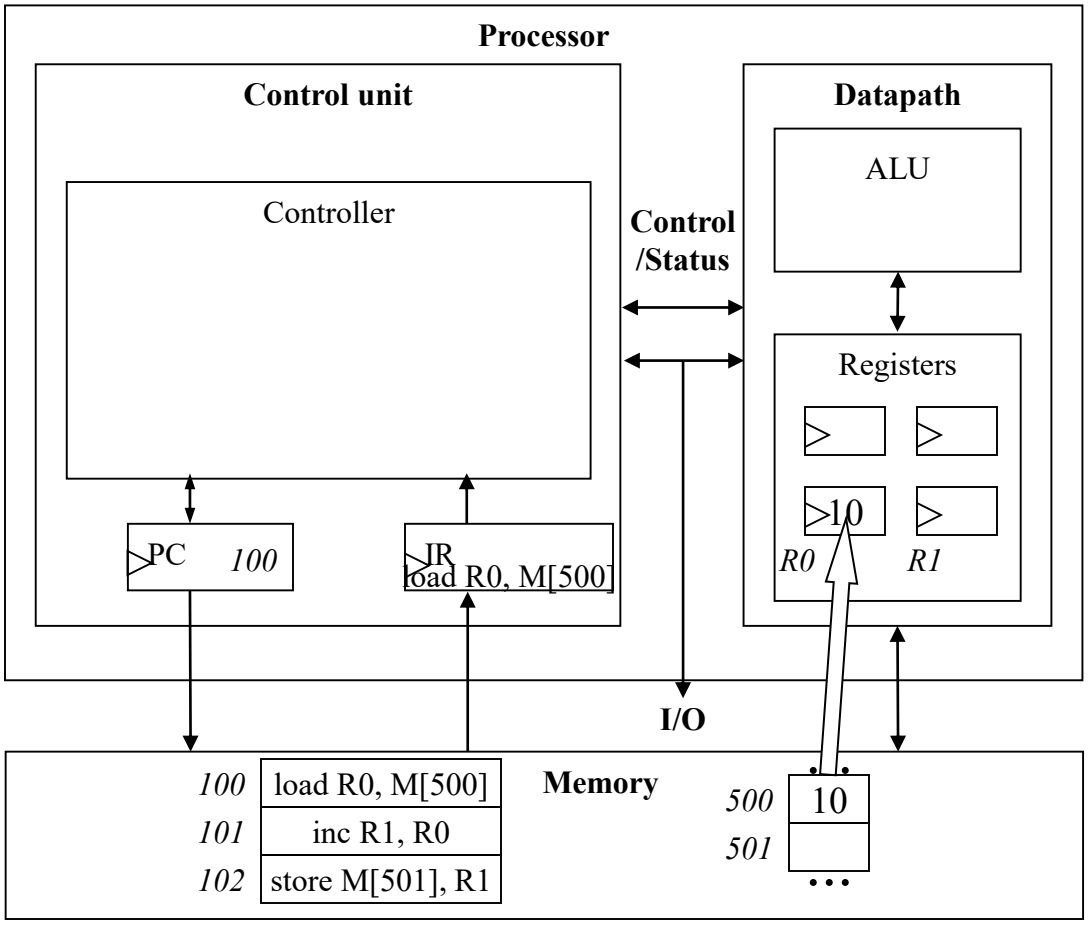
Mikrooperacije nadzorne enote

- Decode
  - Določi (dekodiraj) pomen ukaza



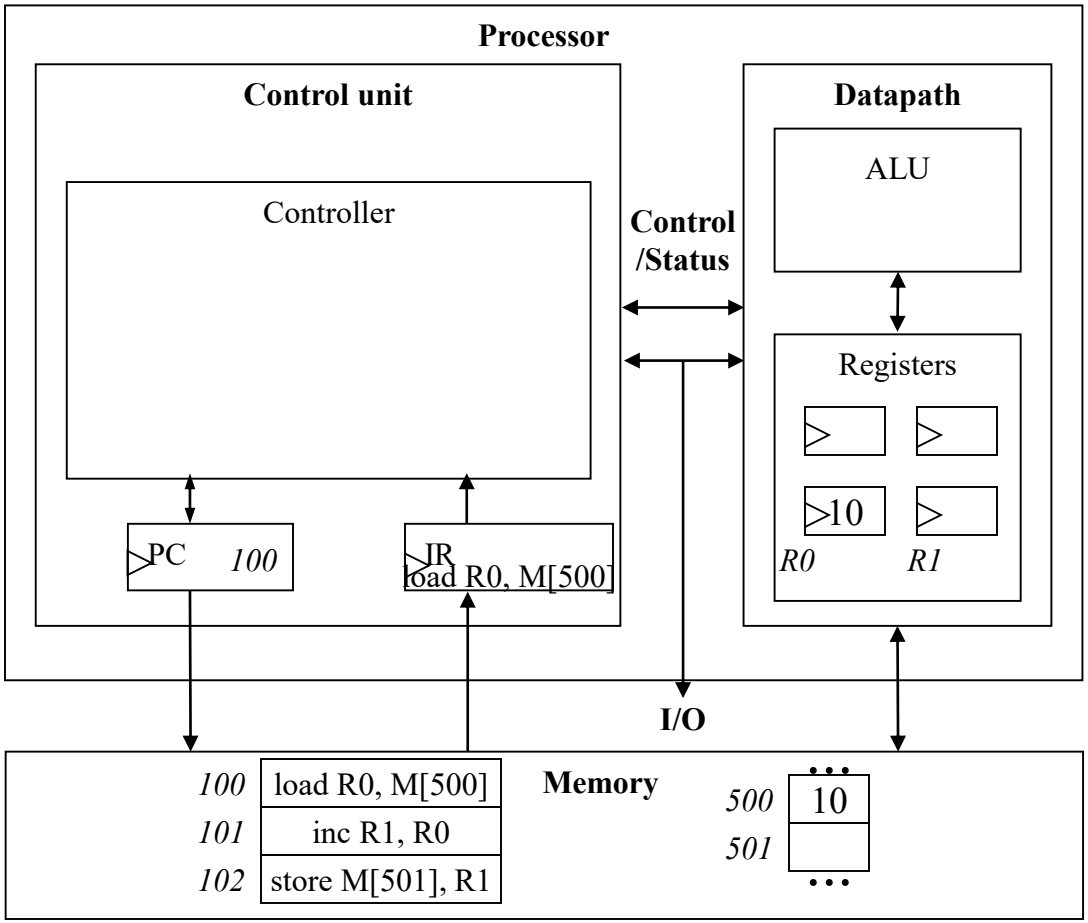
Mikrooperacije nadzorne enote

- Fetch operandov ukaza
  - Pomakni podatke iz spomina v podatkovni register



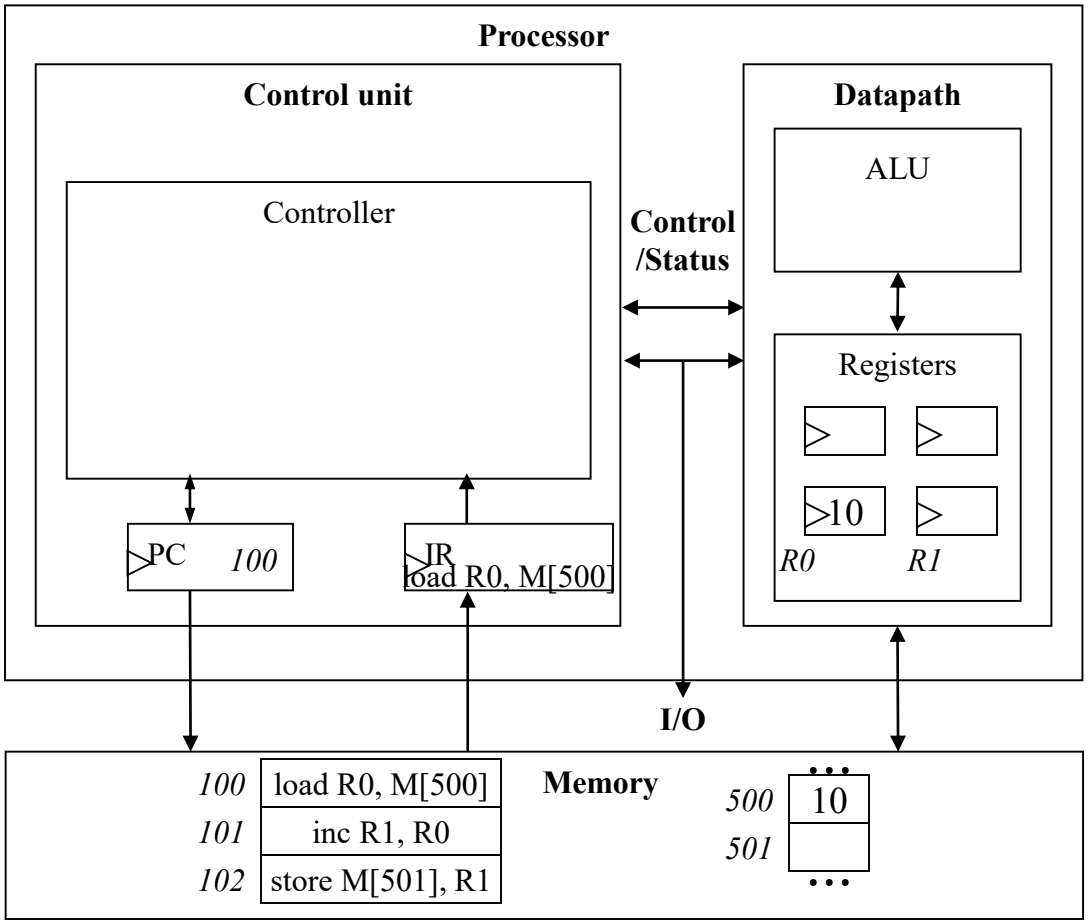
Mikrooperacije nadzorne enote

- Execute
  - Izvedi operacijo z ALU
  - Izvajani ukaz (LOAD) na prosojnici ne naredi nič med tem delom operacije



Mikrooperacije nadzorne enote

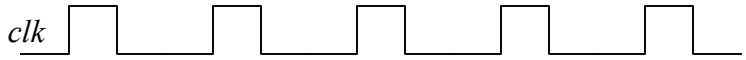
- Store results
  - Shrani rezultat operacije nazaj v spomin
  - Izvajani ukaz (LOAD) na prosojnici ne naredi nič med tem delom operacije

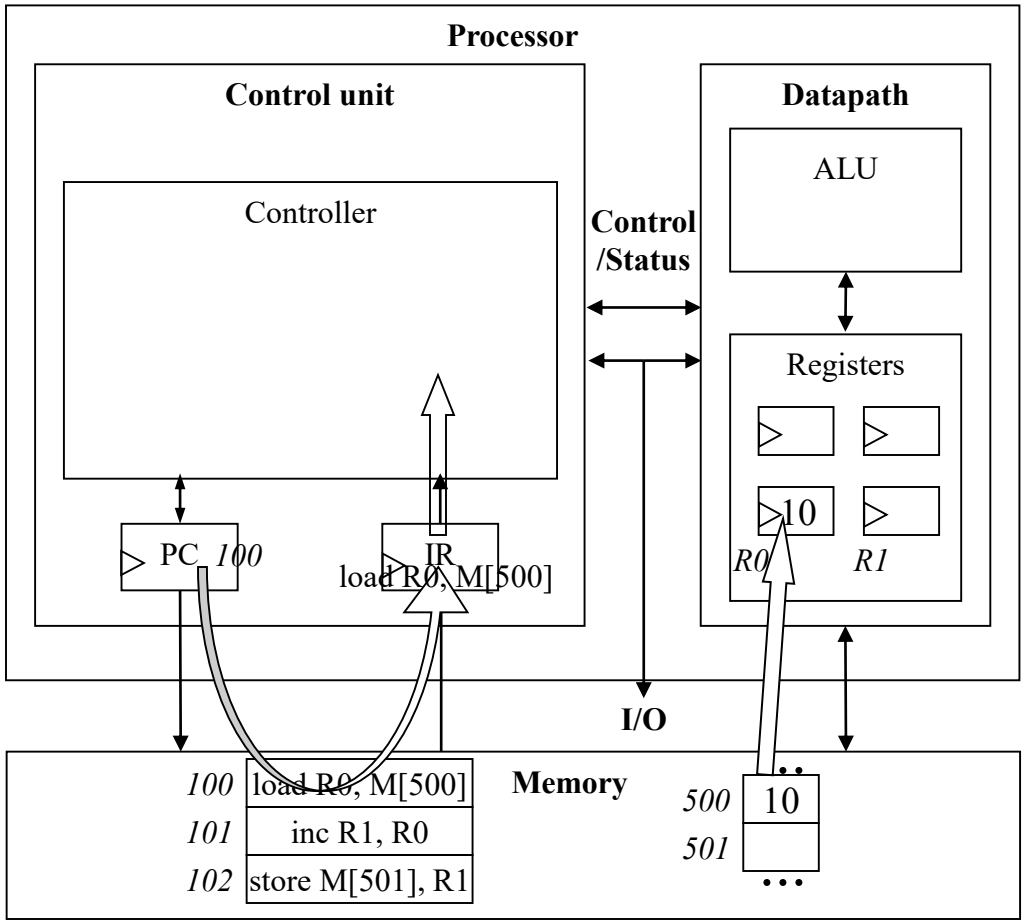


Ukazni cikel

PC=100

Fetch Decode Fetch ops Exec. Store results



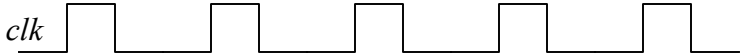


Source: Vahid and Givargis, "Embedded System Design: A Unified Hardware/Software Introduction"

Ukazni cikel

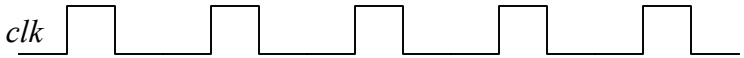
PC=100

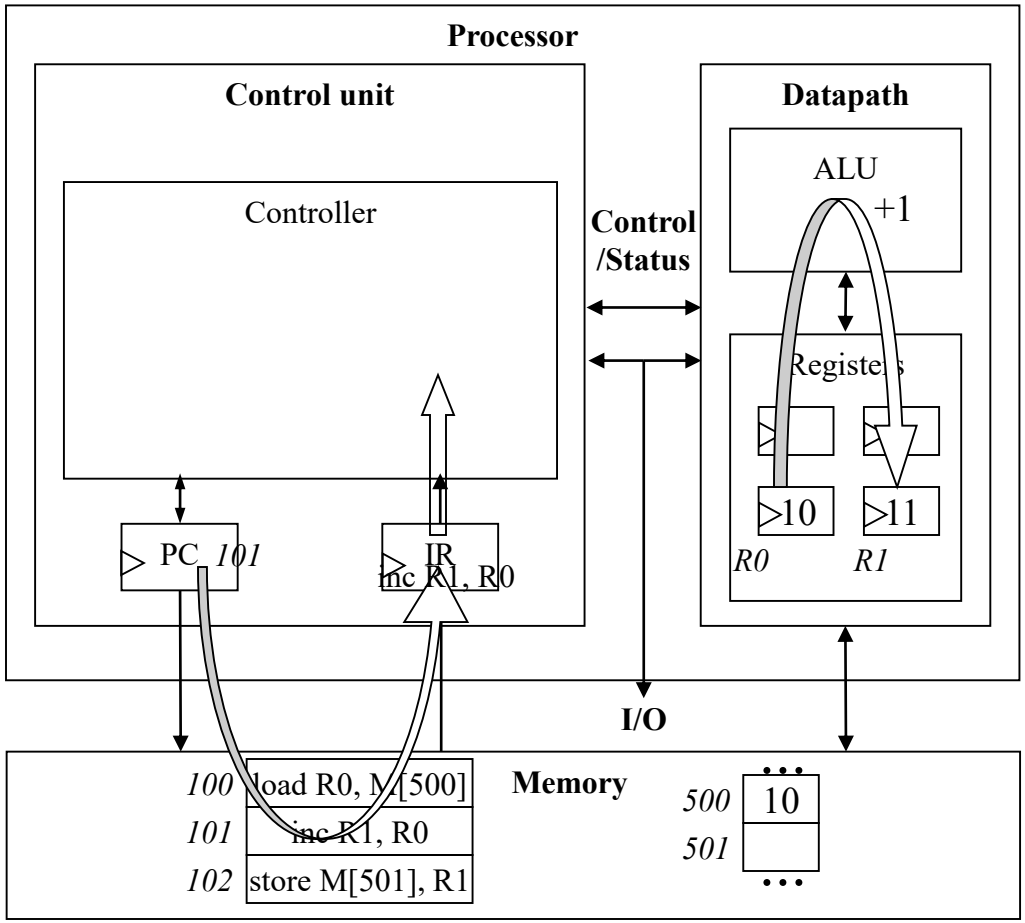
Fetch Decode Fetch ops Exec. Store results



PC=101

Fetch Decode Fetch ops Exec. Store results

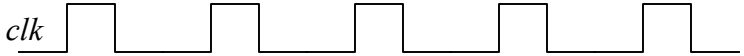




Ukazni cikel

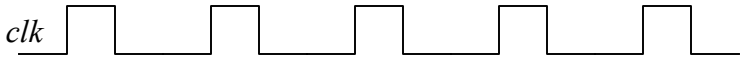
PC=100

Fetch Decode Fetch ops Exec. Store results



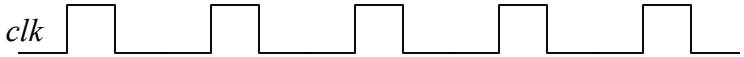
PC=101

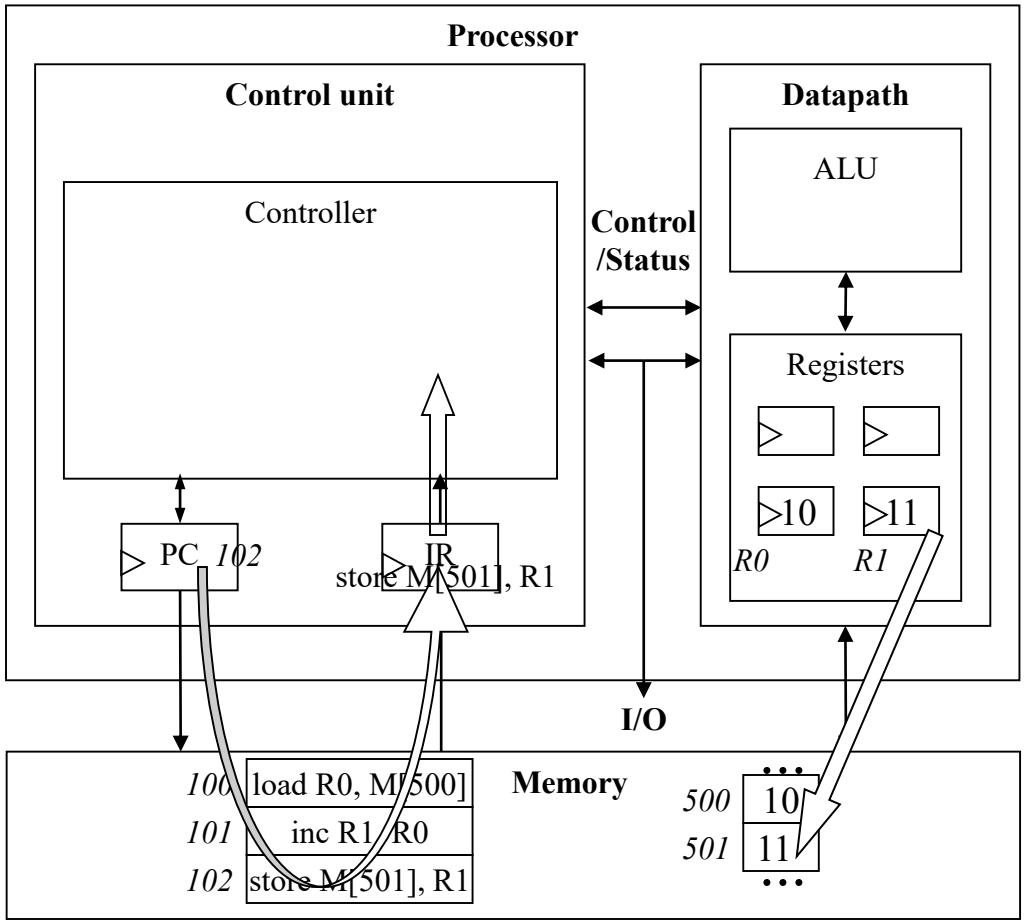
Fetch Decode Fetch ops Exec. Store results



PC=102

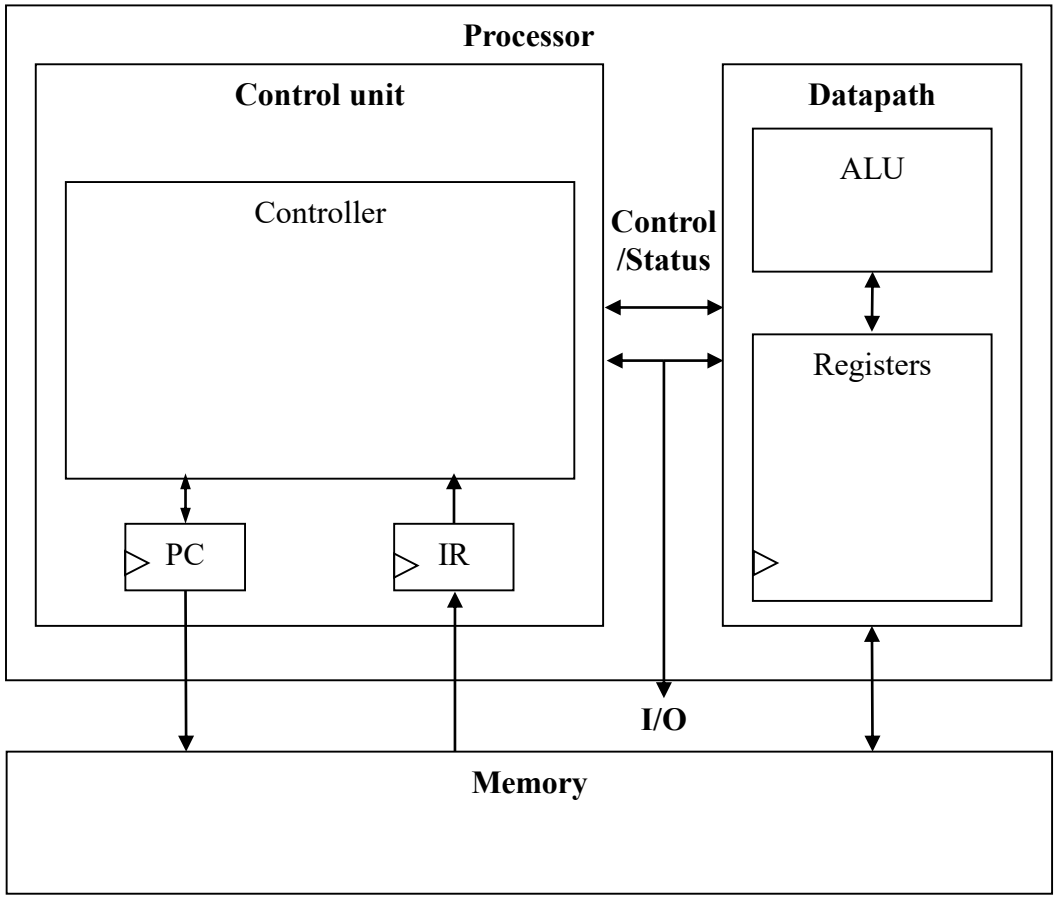
Fetch Decode Fetch ops Exec. Store results





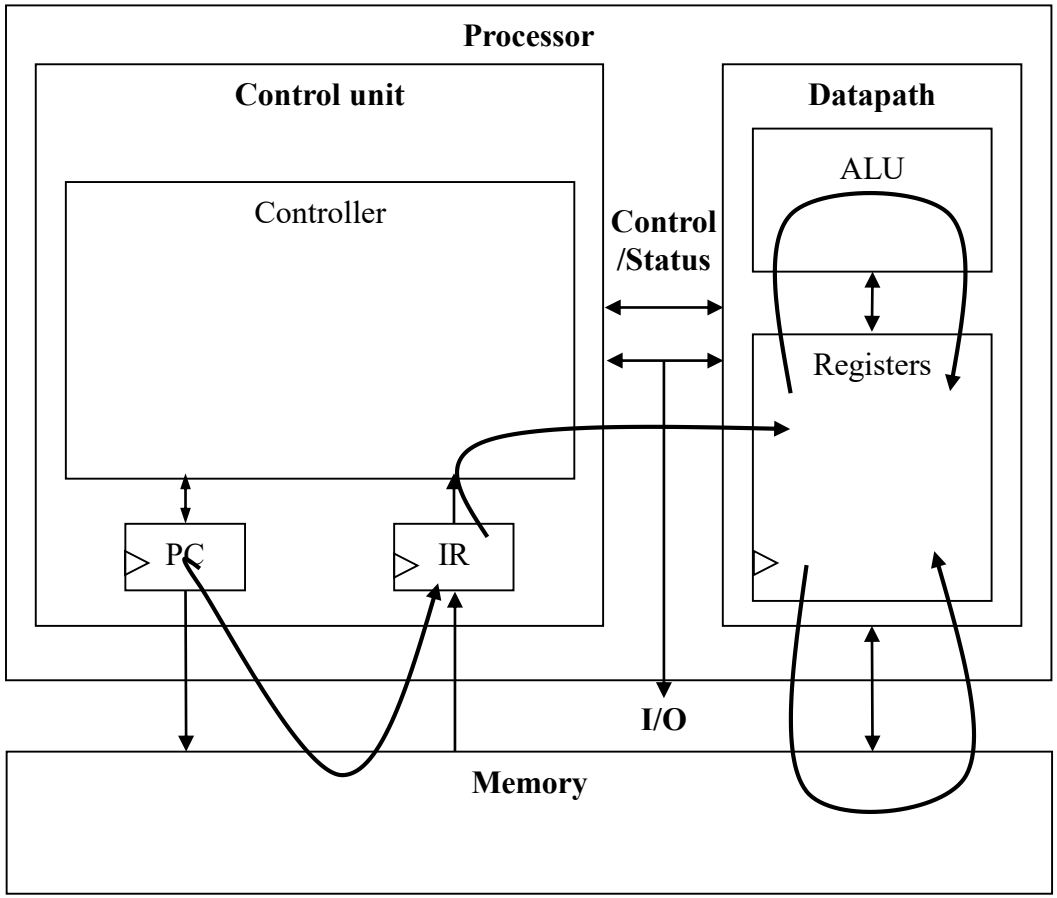
Arhitektura mikroprocesorja

- *N-bitni* procesor
  - N-bitni ALU,
  - N-bitni registri,
  - N-bitna vodila,
  - N-bitni vmesnik za spomin
- Velikost programskega števca (PC) določa velikost programskega prostora

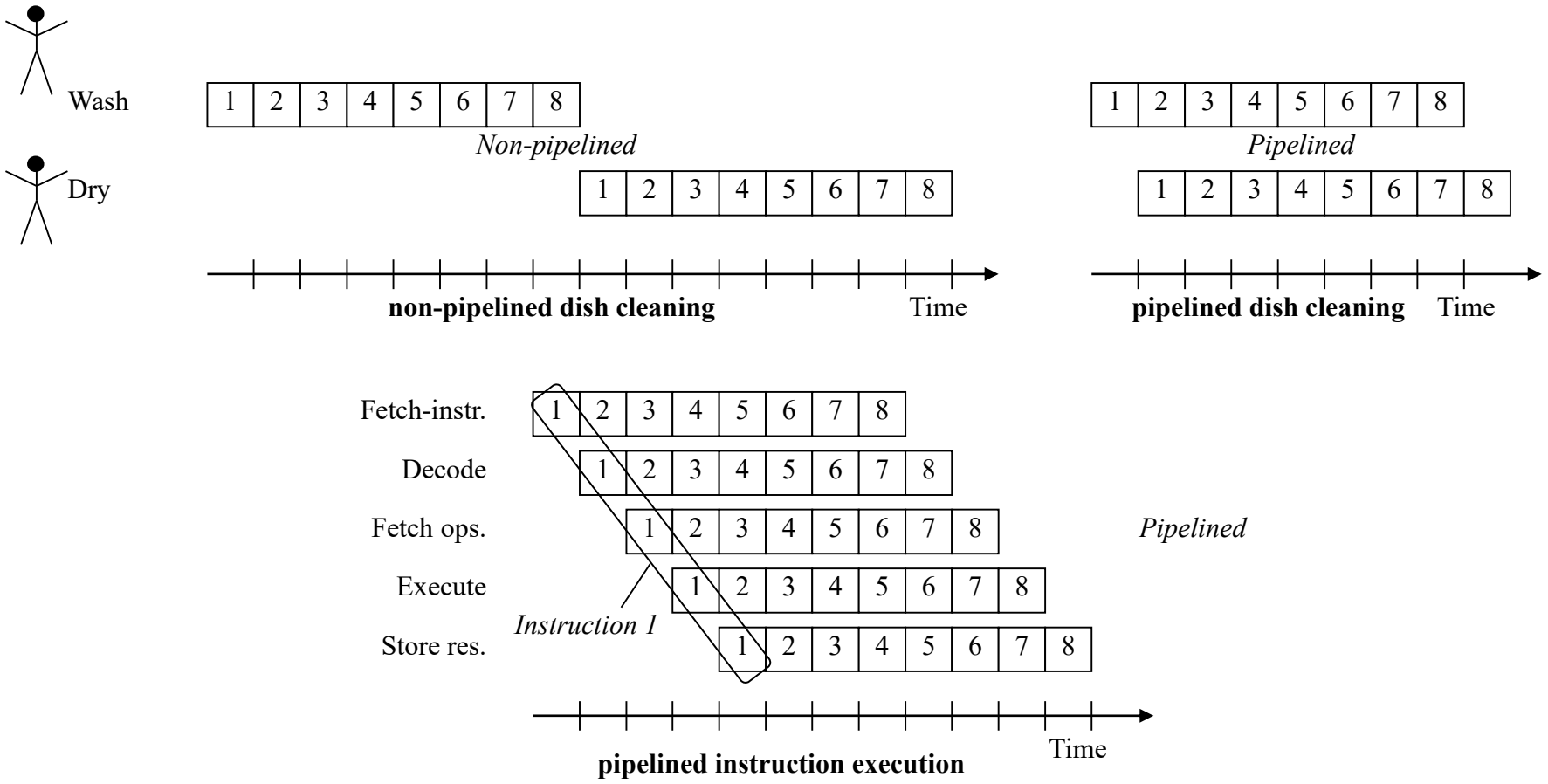


Arhitektura mikroprocesorja

- Frekvenca delovanja
  - Perioda signala ure mora biti večja kot je največja zakasnitev vpisovanja iz registra v register
  - Problem niso registri, ampak je dostop do spomina



Pipelining: Povečevanje učinkovitosti delovanja (Instruction *Throughput)*



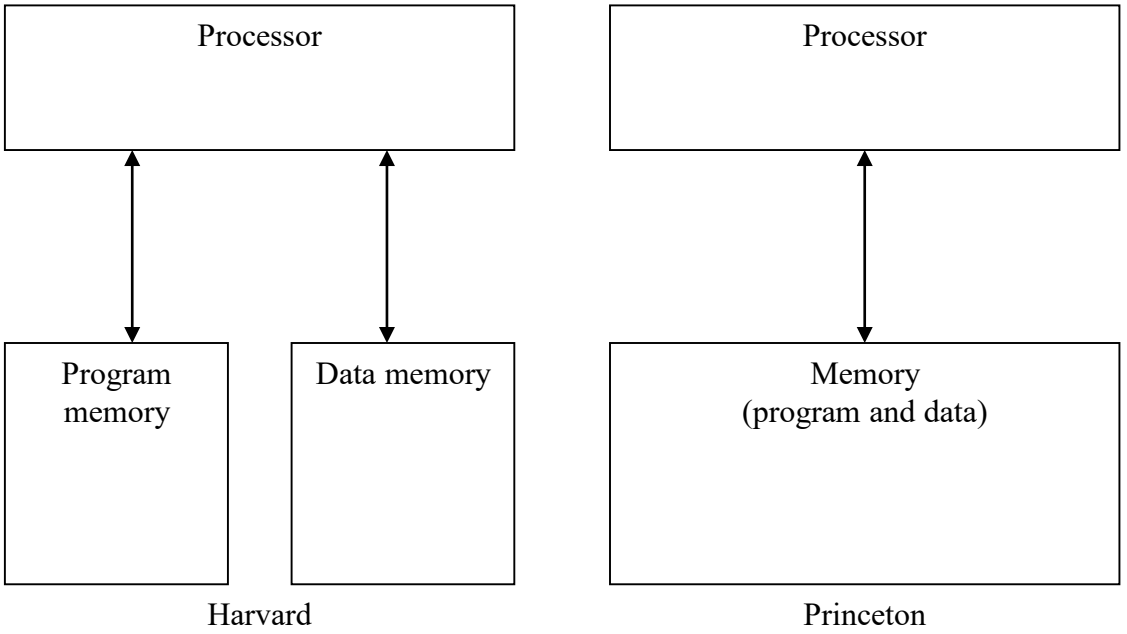
Superskalarne in VLIW arhitekture

- Delovanje lahko izboljšamo z:
  - Hitrejšim signalom ure (sky's not the limit)
  - Pipelining: razdelimo ukaze na stopnje (ang. stages) in stopnje prekrijemo (ang. overlap)
  - *Uporabimo več ALU* in tako izvajamo hkrati več tokov inštrukcij (instruction stream)
    - Superskalarni
      - Skalarni: nima vektorskih operacij
      - Zgrabi (Fetches) ukaze v paketih (batches) in jih izvede vzporedno kar se da največ
        - » Problem je detekcija *neodvisnih* ukazov
      - VLIW (***Very long instruction word***): vsaka beseda v spominu ima več neodvisnih inštrukcij
        - » Prevajalnik zagotavlja zaznavanje in razvrščanje ukazov

SHARC DSP ➔ VLIW ukaz: f12=f0*f4, f8=f8+f12, f0=dm(i0,m3), f4=pm(i8,m9);
en cikel: FP množenje, FP seštevanje in
dvoje nalaganj s samodejnim povečevanjem naslova

Arhitekturi spominskega prostora

- Princeton
  - Manj žic spomina
- Harvard
  - Hkraten dostop do programskega in podatkovnega spomina



Source: Vahid and Givargis, "Embedded System Design: A Unified Hardware/Software Introduction"

Načrtovanje digitalnih naprav

Harvard "single cycle"
mikroprocesor
s fiksno ožičeno enoto

Instruction Set Architecture

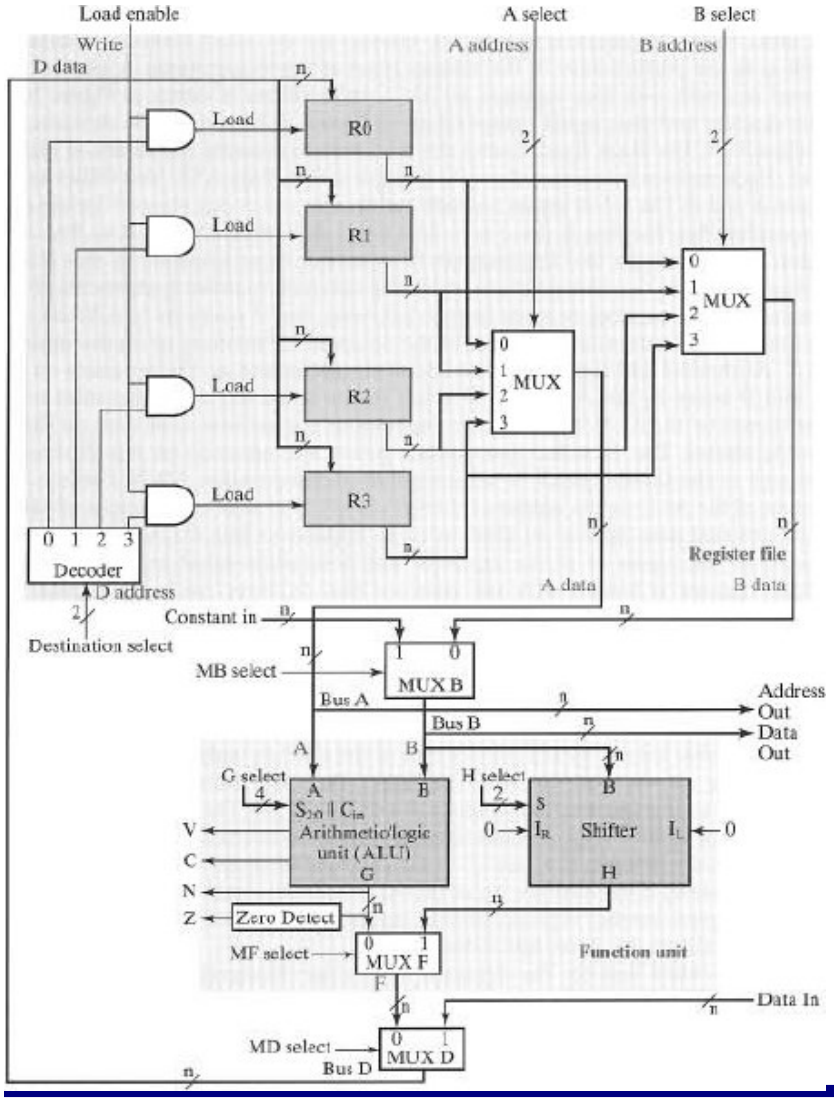
CPU:

- *Podatkovna pot (datapath)*
  - *Niz registrov*
  - *Mikrooperacije, ki se izvajajo nad vsebino registrov*
  - *Nadzorni vmesnik*
- *Nadzorna enota (control unit)*
  (signali, ki krmilijo mikrooperacije v podatkovni poti in ostalih komponentah sistema – spominu)

Podatkovna pot

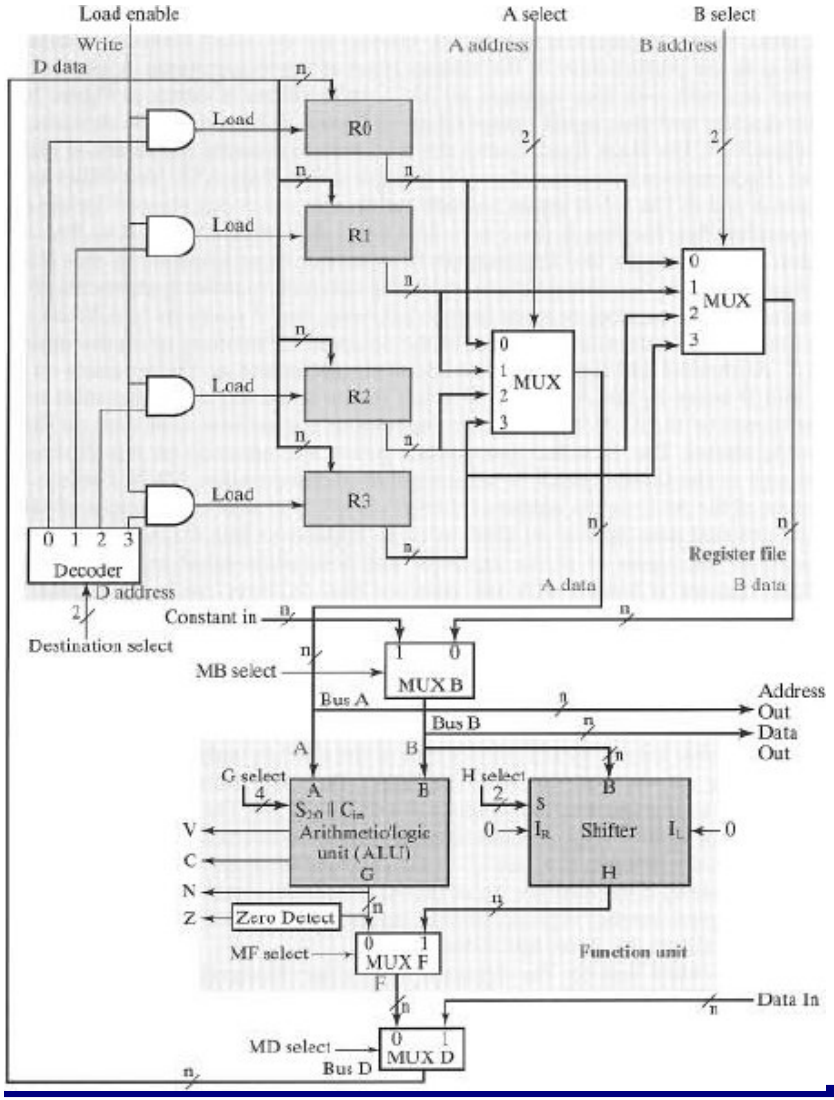
- *Registri običajno ne izvajajo sami mikrooperacij, ampak so povezani preko aritmetično-logične enote (ALU).*
- *Tipična mikrooperacija:*
  - *Podatek iz registra se pojavi na vhodu ALU*
  - *ALU izvede mikrooperacijo*
  - *Rezultat na izhodu ALU se shrani v enega od registrov*

Podatkovna pot



- *4 registri*
- *ALU*
- *pomikalna enota*
- *vodilo A, B*
- *krmiljenje I/O enot*

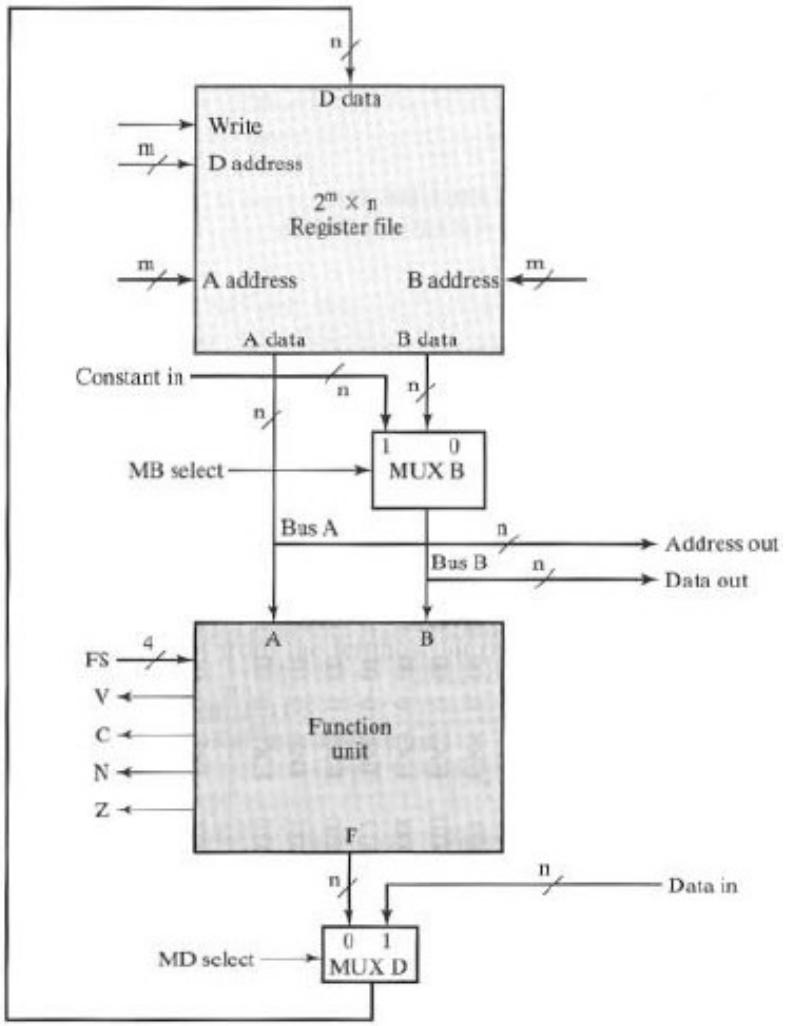
Operacija: R1←R2+R3

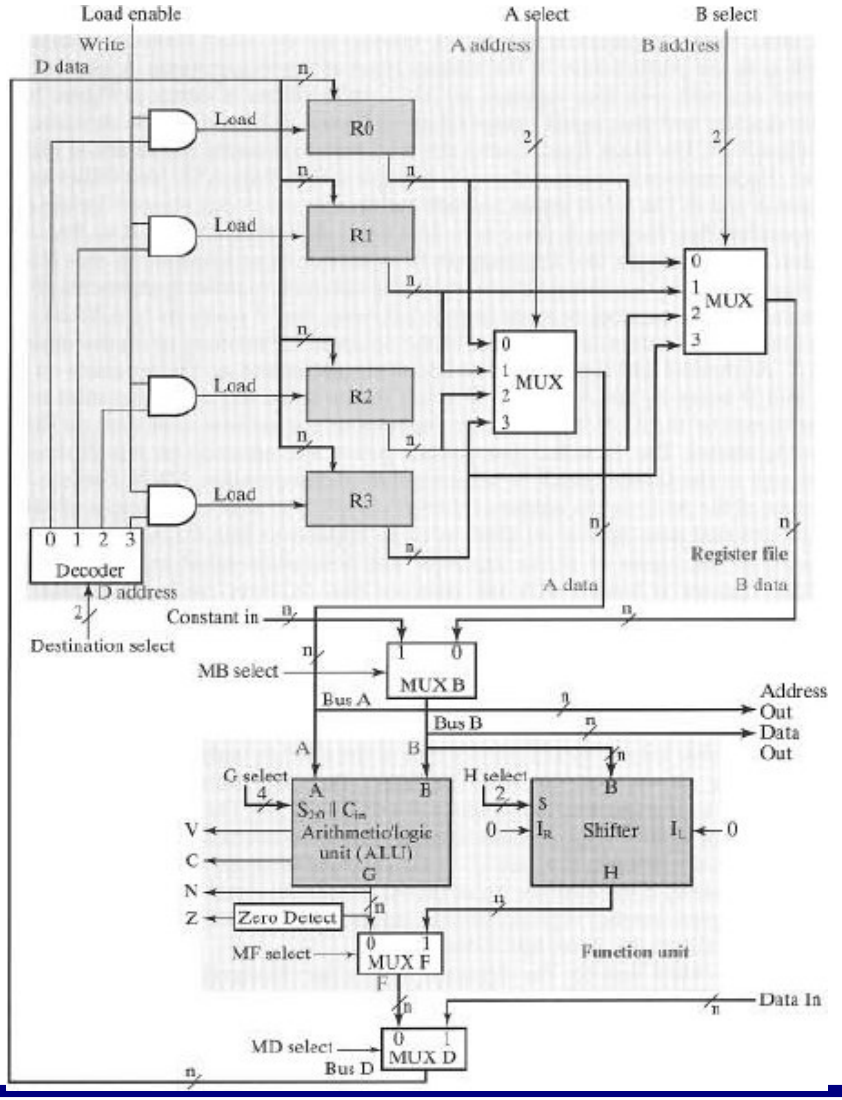


Potek operacije – kontrolna enota:

- *Aselect=2, postavi R2 na vodilo A*
- *Bselect=3, postavi R3 na vodilo B (MBselect=0, brez konstante)*
- *Nastavi G (ALU operacija A+B)*
- *Postavi MFselect=0 (rezultat ALU gre naprej)*
- *Postavi MDselect (izhod MUX F gre na vodilo)*
- *Destination Select = 1, Load Enable = 1*

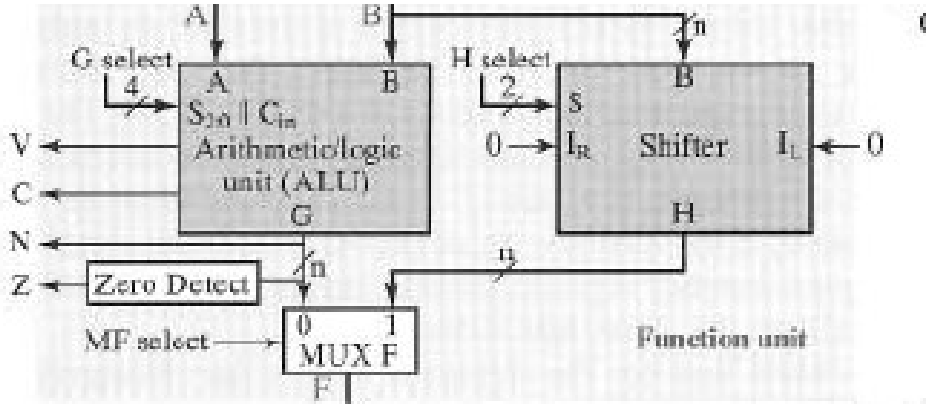
Predstavitev podatkovne poti



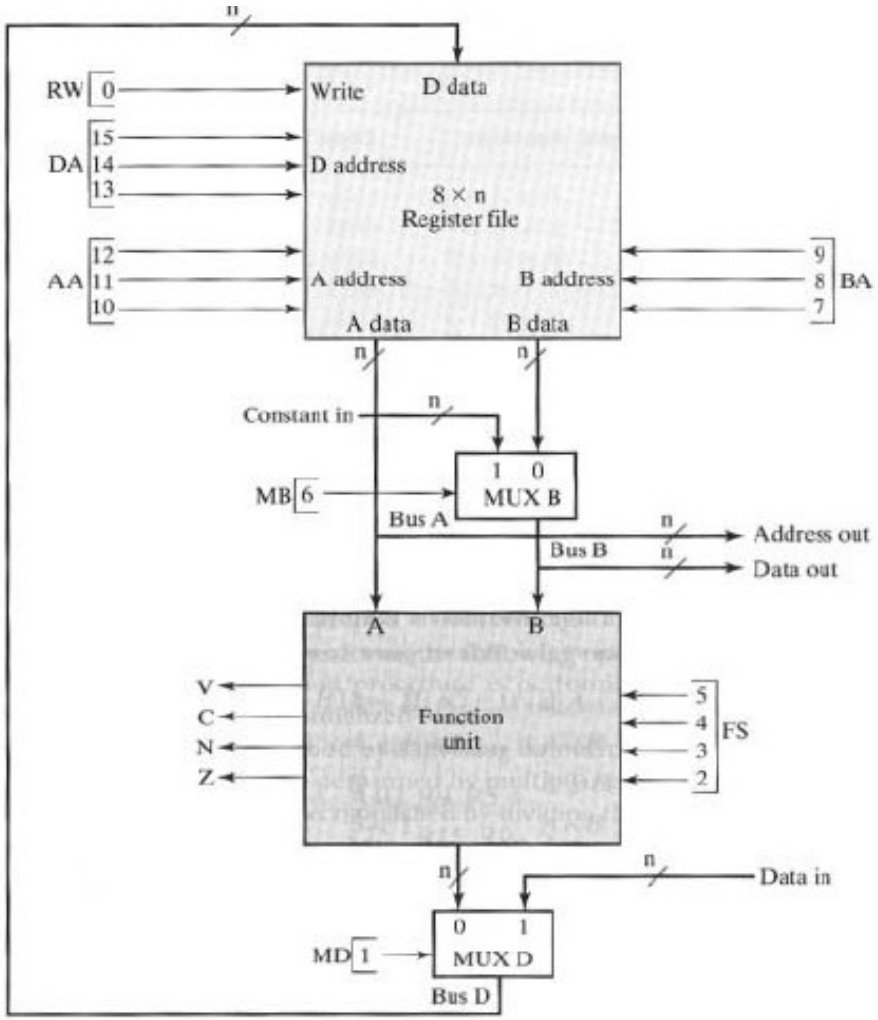


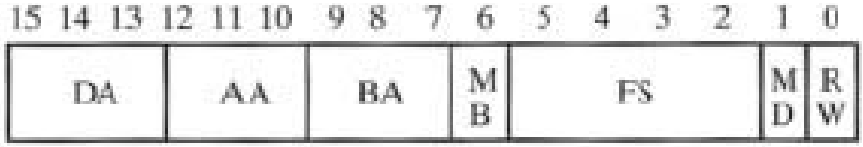
Predstavitev podatkovne poti

FS(3:0)	MF Select	G Select(3:0)	H Select(3:0)	Microoperation
0000	0	0000	XX	$F = A$
0001	0	0001	XX	$F = A + 1$
0010	0	0010	XX	$F = A + B$
0011	0	0011	XX	$F = A + B + 1$
0100	0	0100	XX	$F = A + \overline{B}$
0101	0	0101	XX	$F = A + \overline{B} + 1$
0110	0	0110	XX	$F = A - 1$
0111	0	0111	XX	$F = A$
1000	0	1X00	XX	$F = A \wedge B$
1001	0	1X01	XX	$F = A \vee B$
1010	0	1X10	XX	$F = A \oplus B$
1011	0	1X11	XX	$F = \overline{A}$
1100	1	XXXX	00	$F = B$
1101	1	XXXX	01	$F = \text{sr } B$
1110	1	XXXX	10	$F = \text{sl } B$



Nadzor ukaznih sekvenc





kontrolna beseda	pomen
DA	naslov rezultata operacije
AA	naslov registra na vodilu A
BA	naslov registra na vodilu B
MB	MUX B: konstanta/ vodilo B
FS	izbira funkcije funk. enote
MD	MUX D: ext. podatek/rez. operacije
RW	rezultat se shrani v register ali ne

Nadzorna beseda

DA, AA, BA		MB		FS		MD		RW	
Function	Code	Function	Code	Function	Code	Function	Code	Function	Code
$R0$	000	Register	0	$F = A$	0000	Function	0	No write	0
$R1$	001	Constant	1	$F = A + 1$	0001	Data In	1	Write	1
$R2$	010			$F = A + B$	0010				
$R3$	011			$F = A + B + 1$	0011				
$R4$	100			$F = A + \overline{B}$	0100				
$R5$	101			$F = A + \overline{B} + 1$	0101				
$R6$	110			$F = A - 1$	0110				
$R7$	111			$F = A$	0111				
				$F = A \wedge B$	1000				
				$F = A \vee B$	1001				
				$F = A \oplus B$	1010				
				$F = \overline{A}$	1011				
				$F = B$	1100				
				$F = \text{sr } B$	1101				
				$F = \text{sl } B$	1110				

Tvorba ukaznih sekvenc

Primer: Operacija odštevanja R1←R2-R3 oz. R1←R2+R3'+1

DA, AA, BA		MB		FS		MD		RW	
Function	Code	Function	Code	Function	Code	Function	Code	Function	Code
$R0$	000	Register	0	$F = A$	0000	Function	0	No write	0
$R1$	001	Constant	1	$F = A + 1$	0001	Data In	1	Write	1
$R2$	010			$F = A + B$	0010				
$R3$	011			$F = A + B + 1$	0011				
$R4$	100			$F = A + \overline{B}$	0100				
$R5$	101			$F = A + \overline{B} + 1$	0101				
$R6$	110			$F = A - 1$	0110				
$R7$	111			$F = A$	0111				
				$F = A \wedge B$	1000				
				$F = A \vee B$	1001				
				$F = A \oplus B$	1010				
				$F = \overline{A}$	1011				
				$F = B$	1100				
				$F = \text{sr } B$	1101				
				$F = \text{sl } B$	1110				

DA	AA	BA	MB	FS	MD	RW
$R1$	$R2$	$R3$	Register	$F = A + \overline{B} + 1$	Function	Write
001	010	011	0	0101	0	1

Primeri ukaznih sekvenc

Micro-operation	DA	AA	BA	MB	FS	MD	RW
$R1 \leftarrow R2 - R3$	$R1$	$R2$	$R3$	Register	$F = A + \overline{B} + 1$	Function	Write
$R4 \leftarrow$ sl R6	$R4$	—	$R6$	Register	$F =$ sl B	Function	Write
$R7 \leftarrow R7 + 1$	$R7$	$R7$	—	Register	$F = A + 1$	Function	Write
$R1 \leftarrow R0 + 2$	$R1$	$R0$	—	Constant	$F = A + B$	Function	Write
Data out $\leftarrow R3$	—	—	$R3$	Register	—	—	No Write
$R4 \leftarrow$ Data in	$R4$	—	—	—	—	Data in	Write
$R5 \leftarrow 0$	$R5$	$R0$	$R0$	Register	$F = A \oplus B$	Function	Write

Micro-operation	DA	AA	BA	MB	FS	MD	RW
$R1 \leftarrow R2 - R3$	001	010	011	0	0101	0	1
$R4 \leftarrow$ sl R6	100	XXX	110	0	1110	0	1
$R7 \leftarrow R7 + 1$	111	111	XXX	0	0001	0	1
$R1 \leftarrow R0 + 2$	001	000	XXX	1	0010	0	1
Data out $\leftarrow R3$	XXX	XXX	011	0	XXXX	X	0
$R4 \leftarrow$ Data in	100	XXX	XXX	X	XXXX	1	1
$R5 \leftarrow 0$	101	000	000	0	1010	0	1

Arhitektura enostavnega mikroračunalnika

Programabilni sistem:

Nadzorna enota vsebuje:

- *Programski števec (PC) (števec z vzporednim nalaganjem)*
- *Logiko za interpretiranje in izvedbo (execution) ukazov kot sekvenco mikroukazov*
- *Logiko za nalaganje ukazov*

Ukazi so shranjeni v spominu ROM, RAM

Neprogramabilni sistem:

Spomina ni ➔ ni nadzorne enote.

Nadzorna enota je običajno FSM (delovanje je odvisno od ALU in zunanjih vhodov).

ISA – instruction set architecture

Ukaz (inštrukcija): Skupina bitov, ki je potrebna da računalnik izvede določeno operacijo.

Niz vseh ukazov (instruction set)

Arhitektura, ki sledi iz niza ukazov:

- *Spominski prostor*
- *Oblika ukaza*
- *Določila ukaza*

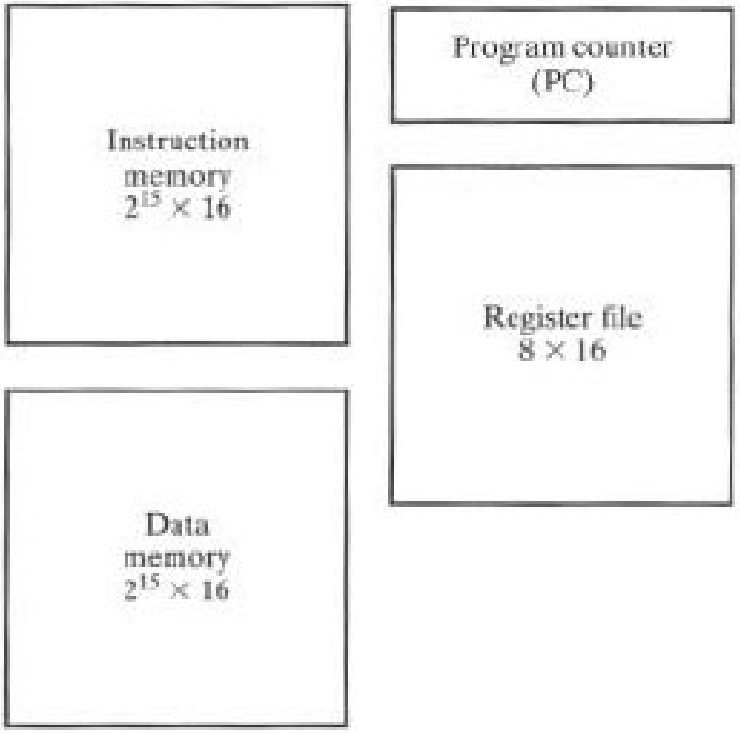
ISA – spominski prostor

Predstavlja spominsko strukturo, ki jo ima na voljo programer:

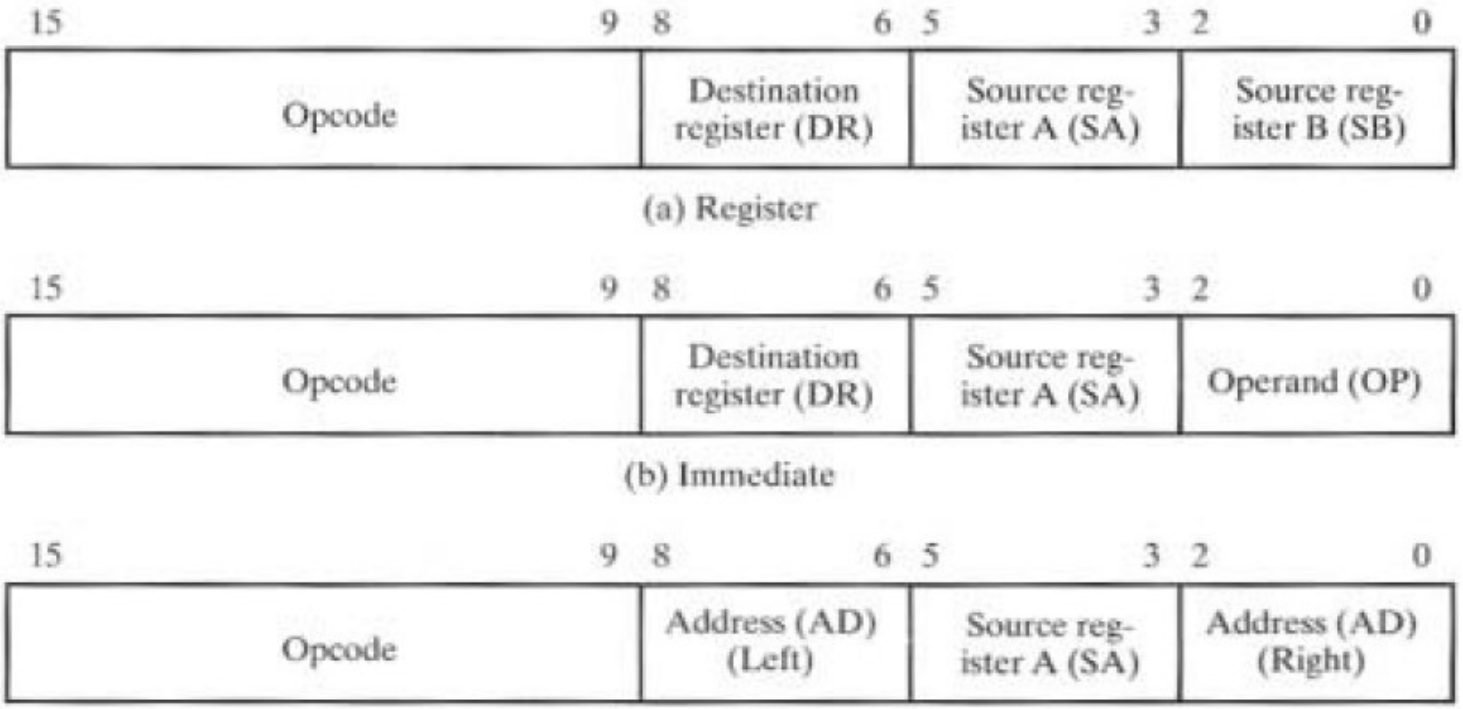
- *Programski spominski prostor*
- *Podatkovni spominski prostor*

Sta lahko:

- *ločena (program v EEPROMU, delovni spomin v RAM) ali*
- *enotna (program, podatki v RAM)*



ISA – oblika ukaza

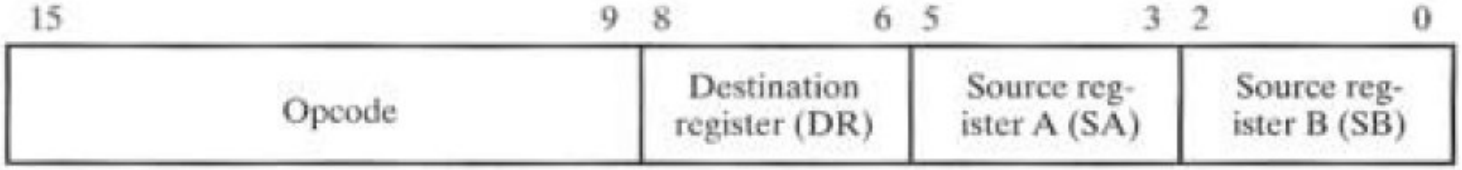


Bitno polje

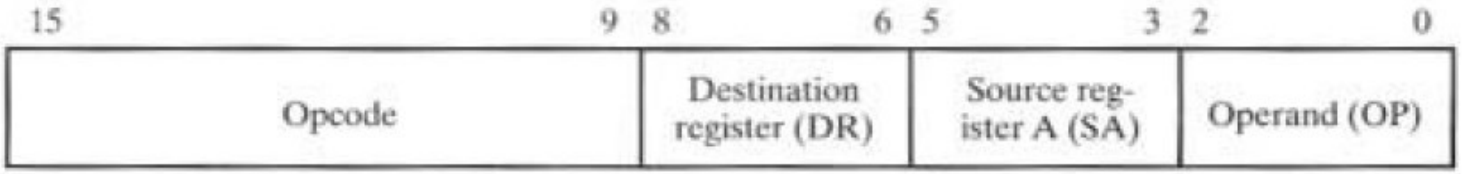
ISA – oblika ukaza

Oblika ukaza : (operandi, cilj)

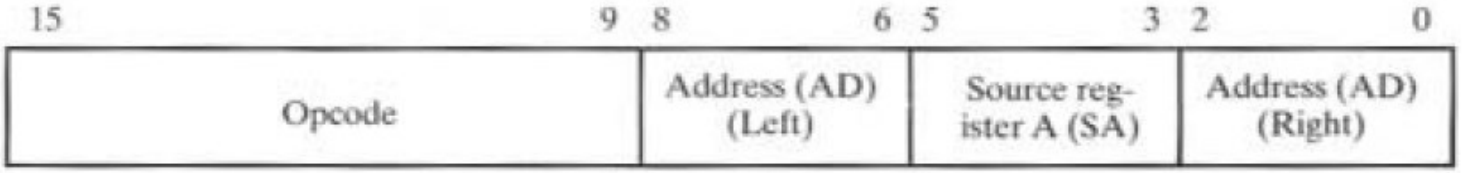
- *Eksplicitno podan operand (iz **kode ukaza** se da razbrati operand) primer – seštevanje, operand podan v kodi ukaza*
- *Implicitno podan operand (iz kode operacije se da razbrati operand) primer – povečaj za 1 (INC), operand (1) podan v kodi operacije same*



(a) Register



(b) Immediate



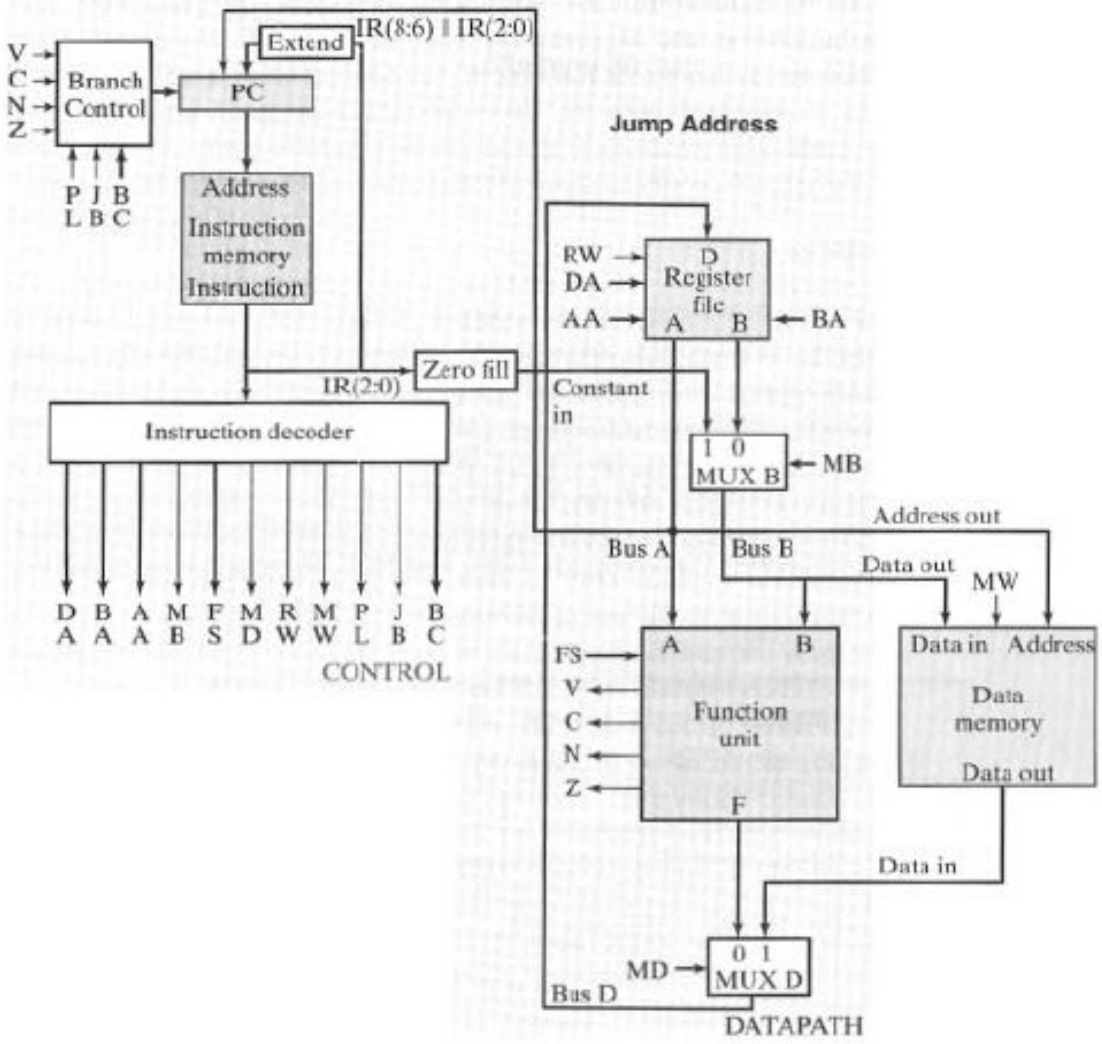
ISA – določila ukaza

Instruction	Opcode	Mnemonic	Format	Description	Status Bits
Move A	0000000	MOVA	RD,RA	R[DR] ← R[SA]	N, Z
Increment	0000001	INC	RD,RA	R[DR] ← R[SA] + 1	N, Z
Add	0000010	ADD	RD,RA,RB	R[DR] ← R[SA] + R[SB]	N, Z
Subtract	0000101	SUB	RD,RA,RB	R[DR] ← R[SA] − R[SB]	N, Z
Decrement	0000110	DEC	RD,RA	R[DR] ← R[SA] − 1	N, Z
AND	0001000	AND	RD,RA,RB	R[DR] ← R[SA] ∧ R[SB]	N, Z
OR	0001001	OR	RD,RA,RB	R[DR] ← R[SA] ∨ R[SB]	N, Z
Exclusive OR	0001010	XOR	RD,RA,RB	R[DR] ← R[SA] ⊕ R[SB]	N, Z
NOT	0001011	NOT	RD,RA	R[DR] ← $\overline{\text{R[SA]}}$	N, Z
Move B	0001100	MOVB	RD,RB	R[DR] ← R[SB]	
Shift Right	0001101	SHR	RD,RB	R[DR] ← sr R[SB]	
Shift Left	0001110	SHL	RD,RB	R[DR] ← sl R[SB]	
Load Immediate	1001100	LDI	RD, OP	R[DR] ← zf OP	
Add Immediate	1000010	ADI	RD,RA,OP	R[DR] ← R[SA] + zf OP	
Load	0010000	LD	RD,RA	R[DR] ← M[SA]	
Store	0100000	ST	RA,RB	M[SA] ← R[SB]	
Branch on Zero	1100000	BRZ	RA,AD	if (R[SA] = 0) PC ← PC + se AD	
Branch on Negative	1100001	BRN	RA,AD	if (R[SA] < 0) PC ← PC + se AD	
Jump	1110000	JMP	RA	PC ← R[SA]	

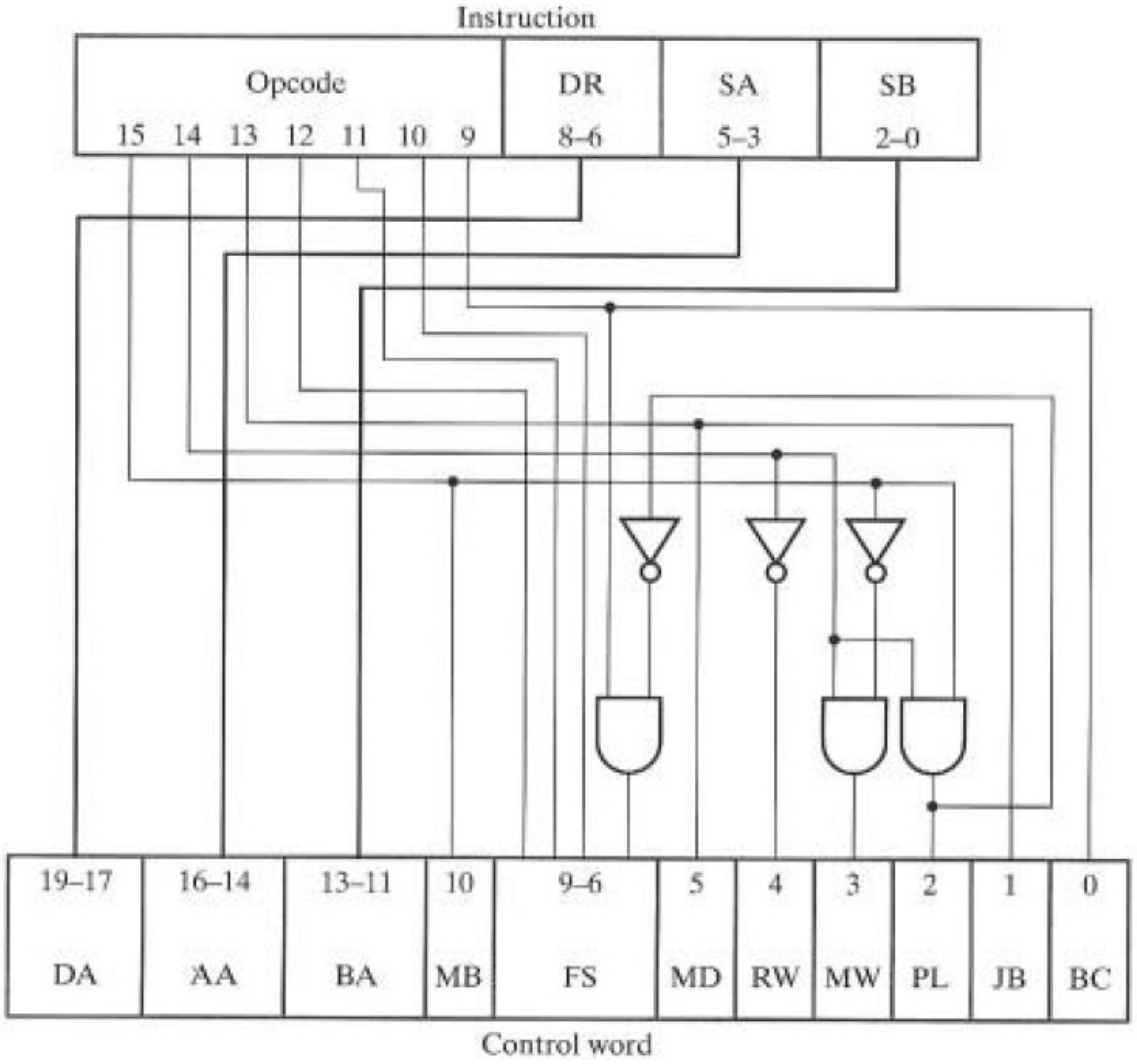
ISA – primer delovanja ukazov

Decimal Address	Memory Contents	Decimal Opcode	Other Fields	Operation
25	0000101 001 010 011	5 (Subtract)	DR:1, SA:2, SB:3	R1 ← R2 – R3
35	0100000 000 100 101	32 (Store)	SA:4, SB:5	M[R4] ← R5
45	1000010 010 111 011	66 (Add Immediate)	DR:2, SA:7, OP:3	R2 ← R7 + 3
55	1100000 101 110 100	96 (Branch on Zero)	AD: 44, SA:6	If R6 = 0, PC ← PC – 20
70	0000000011000000	Data = 192. After execution of instruction in 35, Data = 80.		

Arhitektura enostavnega mikroračunalnika



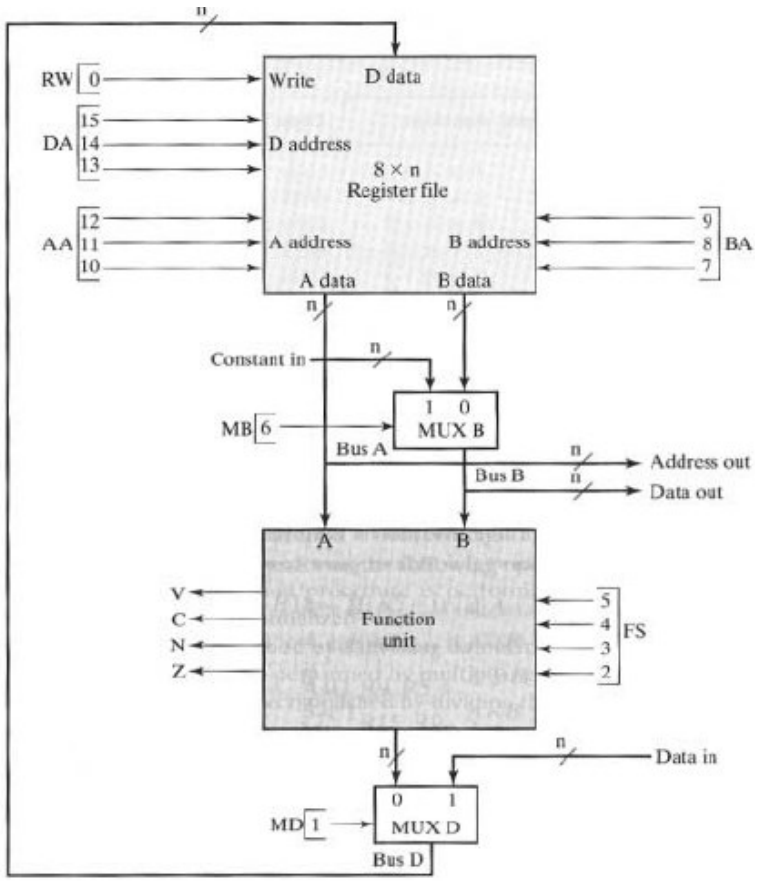
Dekoder ukazov



PL	JB	BC	PC
0	X	X	PC+1
1	0	0	BRZ
1	0	1	BRN
1	1	1	JMP

Pravilnostna tabela dekoderja ukazov

	Instruction Bits				Control Word Bits						
Instruction Function Type	**15**	**14**	**13**	**9**	**MB**	**MD**	**RW**	**MW**	**PL**	**JB**	**BC**
Function unit operations using registers	0	0	0	X	0	0	1	0	0	X	X
Memory read	0	0	1	X	0	1	1	0	0	X	X
Memory write	0	1	0	X	0	X	0	1	0	X	X
Function unit operations using register and constant	1	0	0	X	1	0	1	0	0	X	X
Conditional branch on zero (Z)	1	1	0	0	X	X	0	0	1	0	0
Conditional branch on negative (N)	1	1	0	1	X	X	0	0	1	0	1
Unconditional Jump	1	1	1	X	X	X	0	0	1	1	X



Enostaven program

Operation code	Symbolic name	Format	Description	Function	MB	MD	RW	MW	PL	JB	BC
1000010	ADI	Immediate	Add immediate operand	$R[\mathrm{DR}] \leftarrow R[\mathrm{SA}] + \mathrm{zf}\ I(2{:}0)$	1	0	1	0	0	0	0
0010000	LD	Register	Load memory content into register	$R[\mathrm{DR}] \leftarrow M[R[\mathrm{SA}]]$	0	1	1	0	0	1	0
0100000	ST	Register	Store register content in memory	$M[R[\mathrm{SA}]] \leftarrow R[\mathrm{SB}]$	0	1	0	1	0	0	0
0001110	SL	Register	Shift left	$R[\mathrm{DR}] \leftarrow \mathrm{sl}R[\mathrm{SB}]$	0	0	1	0	0	1	0
0001011	NOT	Register	Complement register	$R[\mathrm{DR}] \leftarrow \overline{R[\mathrm{SA}]}$	0	0	1	0	0	0	1
1100000	BRZ	Jump/Branch	If $R[\mathrm{SA}] = 0$, branch to PC + se AD	If $R[\mathrm{SA}] = 0$, $PC \leftarrow PC + \mathrm{se}AD$, If $R[\mathrm{SA}] \neq 0$, $PC \leftarrow PC + 1$	1	0	0	0	1	0	0

Vejitveni ukazi v zbirnem jeziku

Test	Boolean	Mnemonic	Complementary		Comment
$r > m$	$Z + (N \oplus V) = 0$	BGT	$r \leq m$	BLE	Signed
$r \geq m$	$N \oplus V = 0$	BGE	$r < m$	BLT	Signed
$r = m$	$Z = 1$	BEQ	$r \neq m$	BNE	Signed
$r \leq m$	$Z + (N \oplus V) = 1$	BLE	$r > m$	BGT	Signed
$r < m$	$N \oplus V = 1$	BLT	$r \geq m$	BGE	Signed
$r > m$	$C + Z = 0$	BHI	$r \leq m$	BLS	Unsigned
$r \geq m$	$C = 0$	BHS/BCC	$r < m$	BLO/BCS	Unsigned
$r = m$	$Z = 1$	BEQ	$r \neq m$	BNE	Unsigned
$r \leq m$	$C + Z = 1$	BLS	$r > m$	BHI	Unsigned
$r < m$	$C = 1$	BLO/BCS	$r \geq m$	BHS/BCC	Unsigned
Carry	$C = 1$	BCS	No Carry	BCC	Simple
Negative	$N = 1$	BMI	Plus	BPL	Simple
Overflow	$V = 1$	BVS	No Overflow	BVC	Simple
$r = 0$	$Z = 1$	BEQ	$r \neq 0$	BNE	Simple
Always	-	BRA	Never	BRN	Unconditional

Načrtovanje digitalnih naprav

CORDIC
(COordinate Rotation DIgital Computer)

Uporaba CORDIC

- Realizacija funkcije z rotacijo vektorja
- Izračun trigonometričnih funkcij
- Izračun obratnih trigonometričnih funkcij $\tan^{-1}(a/b)$
- Izračun $\sqrt{(a^2 + b^2)}$ itd.
- Razširitev na hiperbolične funkcije
- Deljenje in množenje
- Izračun $\sqrt{}$, LOG, e^x
- Izračun linearnih transformacij, digitalnih filtrov in reševanje sistemov linearnih enačb
- Področja uporabe: DSP, obdelava slik, 3D grafika, robotika

Rotacija vektorja v krožnem koordinatnem sistemu

- Realizacija funkcije z rotacijo vektorja
- Izračun trigonometričnih funkcij
- Izračun obratnih trigonometričnih funkcij $\tan^{-1}(a/b)$
- Izračun $\sqrt{(a^2 + b^2)}$ itd.

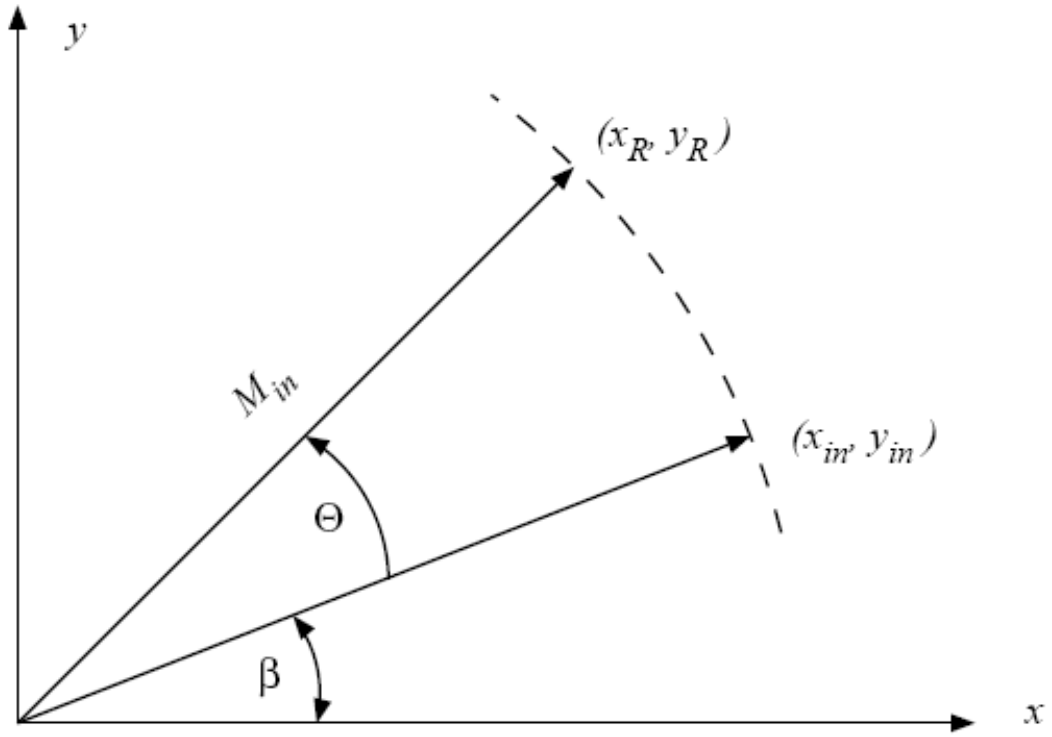
$$x_R = M_{in} \cdot \cos(\beta + \theta) = x_{in}\cos(\theta) - y_{in}\sin(\theta)$$
$$y_R = M_{in} \cdot \sin(\beta + \theta) = x_{in}\sin(\theta) + y_{in}\cos(\theta)$$

$$\begin{bmatrix} x_R \\ y_R \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} x_{in} \\ y_{in} \end{bmatrix}$$

Rotacija vektorja v krožnem koordinatnem sistemu

$$x_R = M_{in} \cdot \cos(\beta + \theta) = x_{in} \cos(\theta) - y_{in} \sin(\theta)$$

$$y_R = M_{in} \cdot \sin(\beta + \theta) = x_{in} \sin(\theta) + y_{in} \cos(\theta)$$



Rotacija vektorja v krožnem koordinatnem sistemu

- Rotacijo vektorja za dani kot (fi) izvajamo zaporedoma z mikrorotacijami po elementarnih kotih $alfa_i$.

$$\theta = \sum_{j=0}^{\infty} \alpha_j$$

- Za operator rotacije velja:

$$ROT(\theta) = \prod_{j=0}^{\infty} ROT(\alpha_j)$$

- Kar lahko za j-to iteracijo mikrorotacije ROT($alfa_i$) pišemo kot:

$$\begin{aligned} x_R[j+1] &= x_R[j]\cos(\alpha_j) - y_R[j]\sin(\alpha_j) \\ y_R[j+1] &= x_R[j]\sin(\alpha_j) + y_R[j]\cos(\alpha_j) \end{aligned}$$

Rotacija vektorja v krožnem koordinatnem sistemu

- Kako se izognemo množenju pri mikrorotacijah?

$$\begin{aligned} x_R[j+1] &= x_R[j]\cos(\alpha_j) - y_R[j]\sin(\alpha_j) \\ y_R[j+1] &= x_R[j]\sin(\alpha_j) + y_R[j]\cos(\alpha_j) \end{aligned}$$

- V zgornjem izrazu izpostavimo:

$$\begin{aligned} x_R[j+1] &= \cos(\alpha_j)(x_R[j] - y_R[j]\tan(\alpha_j)) \\ y_R[j+1] &= \cos(\alpha_j)(y_R[j] + x_R[j]\tan(\alpha_j)) \end{aligned}$$

- Izberemo elementarne kote mikrorotacij kot:

$$\alpha_j = \tan^{-1}(\sigma_j(2^{-j})) = \sigma_j \tan^{-1}(2^{-j}) \qquad \sigma_j \in \{-1, 1\}$$

Rotacija vektorja v krožnem koordinatnem sistemu

- Izbira kotov mikrorotacij kot potence indeksa 2^{-j} (v radianih)

$$\alpha_j = \tan^{-1}(\sigma_j(2^{-j})) = \sigma_j \tan^{-1}(2^{-j}) \qquad \sigma_j \in \{-1, 1\}$$

- Poenostavi operator mikrorotacije na pomik (2^{-j}) in seštevanje

$$\begin{aligned} x[j+1] &= x[j] - \sigma_j 2^{-j} y[j] \\ y[j+1] &= y[j] + \sigma_j 2^{-j} x[j] \end{aligned}$$

- Pri iteraciji mikrorotacije se vrtenja vektorja spremeni iz z(j) na z(j+1):

$$z[j+1] = z[j] - \alpha_j = z[j] - \sigma_j \tan^{-1}(2^{-j})$$

Rotacija vektorja v krožnem koordinatnem sistemu

- V iteraciji mikrorotacije se vektor podaljša za člen cos(alfa$_j$)

$$x_R[j+1] = \cos(\alpha_j)(x_R[j] - y_R[j]\tan(\alpha_j))$$
$$y_R[j+1] = \cos(\alpha_j)(y_R[j] + x_R[j]\tan(\alpha_j))$$

- Zato moramo dolžino vektorja primerno popraviti, kar storimo s konstanto K(j)=1/cos(alfa$_j$)

$$M[j+1] = K[j]M[j] = \frac{1}{\cos\alpha_j}M[j] = (1+\sigma_j^2 2^{-2j})^{1/2}M[j] = (1+2^{-2j})^{1/2}M[j]$$

- Korekcije K(j) lahko obravnavamo posebej (jih izpostavimo) in dobimo konvergentno vrsto za splošni korekcijski faktor K:

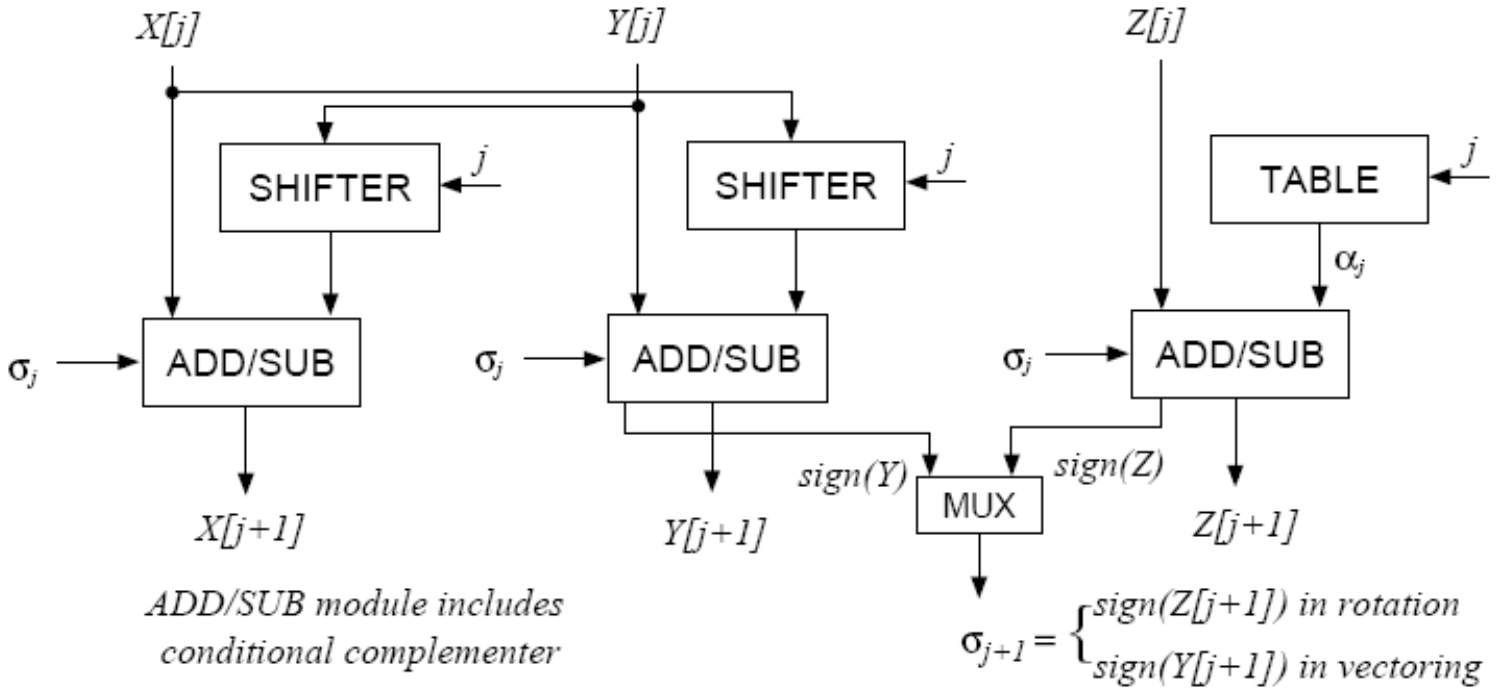
$$K = \prod_{j=0}^{\infty}(1+2^{-2j})^{1/2} \approx 1.6468$$

Rotacija vektorja v krožnem koordinatnem sistemu

- Vrtenje vektorja predstavlja iterativno zmanjševanje razlike z(j) med neznanim kotom in seštevkom elementarnih kotov alfa$_j$

$$
\begin{aligned}
x[j+1] &= x[j] - \sigma_j 2^{-j} y[j] \\
y[j+1] &= y[j] + \sigma_j 2^{-j} x[j] \\
z[j+1] &= z[j] - \sigma_j \tan^{-1}(2^{-j})
\end{aligned}
$$

Mikrorotacija – podatkovna pot



$$
\begin{aligned}
x[j+1] &= x[j] - \sigma_j 2^{-j} y[j] \\
y[j+1] &= y[j] + \sigma_j 2^{-j} x[j] \\
z[j+1] &= z[j] - \sigma_j \tan^{-1}(2^{-j})
\end{aligned}
$$

Iteracije mikrorotacij

- Če zavrtimo začetni vektor (x_{in}, y_{in}) za nek kot fi
- Kot fi postavimo kot začetno iteracijo z(0) in iterativno zmanjšujemo razliko z(j+1) proti 0. Pri iteriranju se spreminjajo samo predznaki kotov $sigma_j$.

$$
\begin{aligned}
x[j+1] &= x[j] - \sigma_j 2^{-j} y[j] \\
y[j+1] &= y[j] + \sigma_j 2^{-j} x[j] \\
z[j+1] &= z[j] - \sigma_j \tan^{-1}(2^{-j})
\end{aligned}
$$

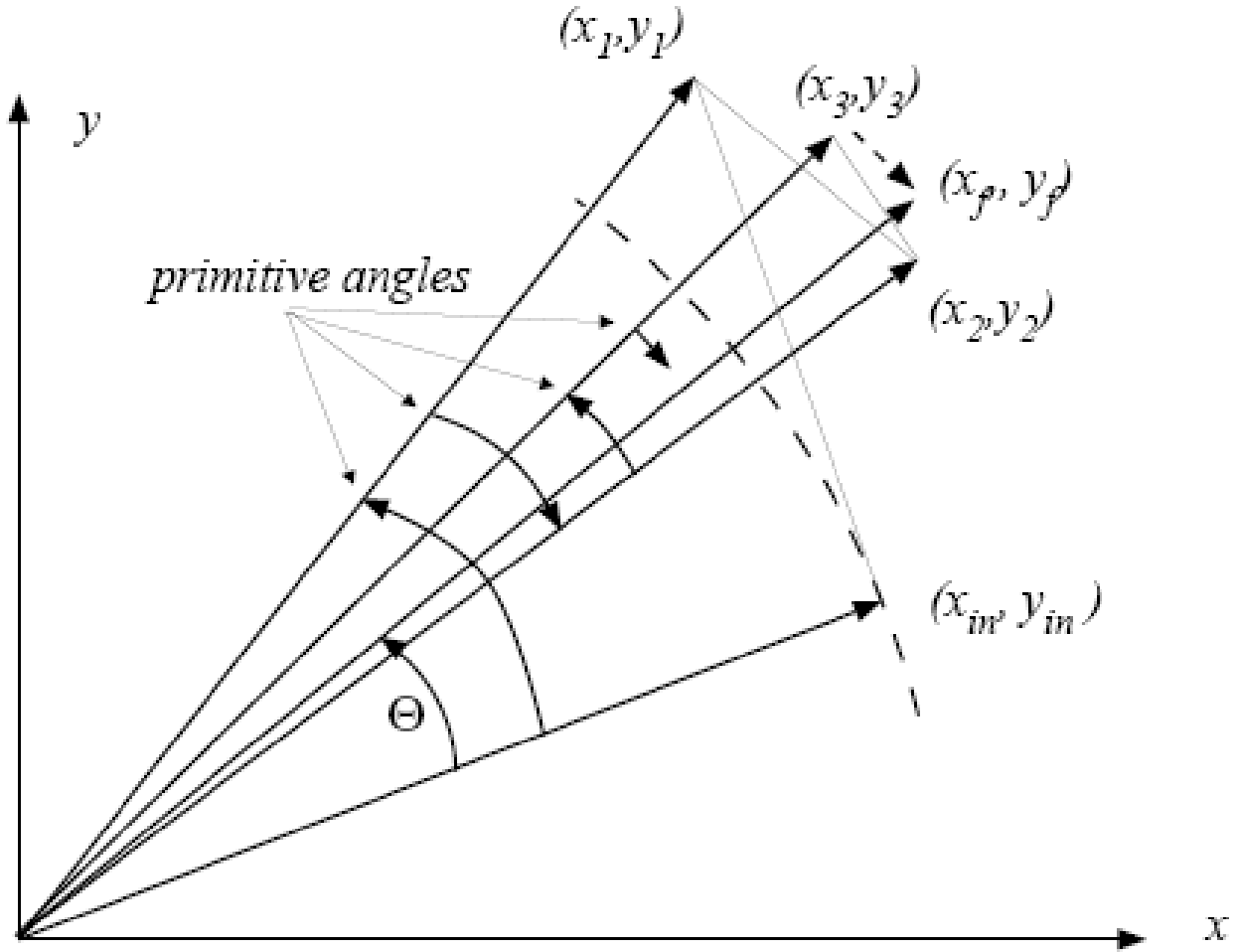
$$
z[0] = \theta \quad x[0] = x_{in} \quad y[0] = y_{in}
$$

$$
\sigma_j = \begin{cases} 1 & if \ z[j] \geq 0 \\ -1 & if \ z[j] < 0 \end{cases}
$$

- Po n iteracijah se nove koordinate vektorja (x_f,y_f) nahajajo na:

$$
\begin{aligned}
x_f &= K(x_{in}\cos\theta - y_{in}\sin\theta) \\
y_f &= K(x_{in}\sin\theta + y_{in}\cos\theta) \\
z_f &= 0
\end{aligned}
$$

Grafična predstavitev iteracije mikrorotacij



Primer iteracije mikrorotacij

- Zavrtimo začetni vektor (x_{in}, y_{in}) za fi=67° v 12 iteracijah.
- Začetne koordinate vektorja (x_{in}, y_{in})=(1, 0.125)

j	$z[j]$	σ_j	$x[j]$	$y[j]$
0	1.1693	1	1.0	0.125
1	0.3839	1	0.875	1.125
2	-0.0796	-1	0.3125	1.1562
3	0.1653	1	0.7031	1.4843
4	0.0409	1	0.5175	1.5722
5	-0.0214	-1	0.4193	1.6046
6	0.0097	1	0.4694	1.5915
7	-0.0058	-1	0.4445	1.5988
8	0.0019	1	0.4570	1.5953
9	-0.0019	-1	0.4508	1.5971
10	0.0000	1	0.4539	1.5962
11	-0.0009	-1	0.4524	1.5967
12	-0.0004	-1	0.4531	1.5965
13			0.4535	1.5963

$$x[j+1] = x[j] - \sigma_j 2^{-j} y[j]$$
$$y[j+1] = y[j] + \sigma_j 2^{-j} x[j]$$
$$z[j+1] = z[j] - \sigma_j \tan^{-1}(2^{-j})$$

$$z[0] = \theta \quad x[0] = x_{in} \quad y[0] = y_{in}$$

$$\sigma_j = \begin{cases} 1 & if \ z[j] \geq 0 \\ -1 & if \ z[j] < 0 \end{cases}$$

(x_f,y_f) = (0.2756,0.9693)

$$x_f = K(x_{in}\cos\theta - y_{in}\sin\theta)$$
$$y_f = K(x_{in}\sin\theta + y_{in}\cos\theta)$$
$$z_f = 0$$

Primer iteracije mikrorotacij

- Zavrtimo začetni vektor (x_{in}, y_{in}) za fi=67° v 12 iteracijah.
- Začetne koordinate vektorja (x_{in}, y_{in})=(1, 0.125)

j	$z[j]$	σ_j	$x[j]$	$y[j]$
0	1.1693	1	1.0	0.125
1	0.3839	1	0.875	1.125
2	-0.0796	-1	0.3125	1.1562
3	0.1653	1	0.7031	1.4843
4	0.0409	1	0.5175	1.5722
5	-0.0214	-1	0.4193	1.6046
6	0.0097	1	0.4694	1.5915
7	-0.0058	-1	0.4445	1.5988
8	0.0019	1	0.4570	1.5953
9	-0.0019	-1	0.4508	1.5971
10	0.0000	1	0.4539	1.5962
11	-0.0009	-1	0.4524	1.5967
12	-0.0004	-1	0.4531	1.5965
13			0.4535	1.5963

- Pravilen rezultat dobimo po korekciji zadnje iteracije:
  x_f= 0.2756 = x(13)/K
  y_f= 0.9693 = y(13)/K
- kjer je K=1.64676

Uporaba mikrorotacij za izračun trigonometričnih funkcij sin, cos

- Začetni vektor (x_{in}, y_{in}) izberemo tako: x(0)=1/K in y(0)=0
- Takrat se splošni operator ROT(fi) spremeni (a=1, b=0):

$$a\cos\theta - b\sin\theta$$

$$a\sin\theta + b\cos\theta$$

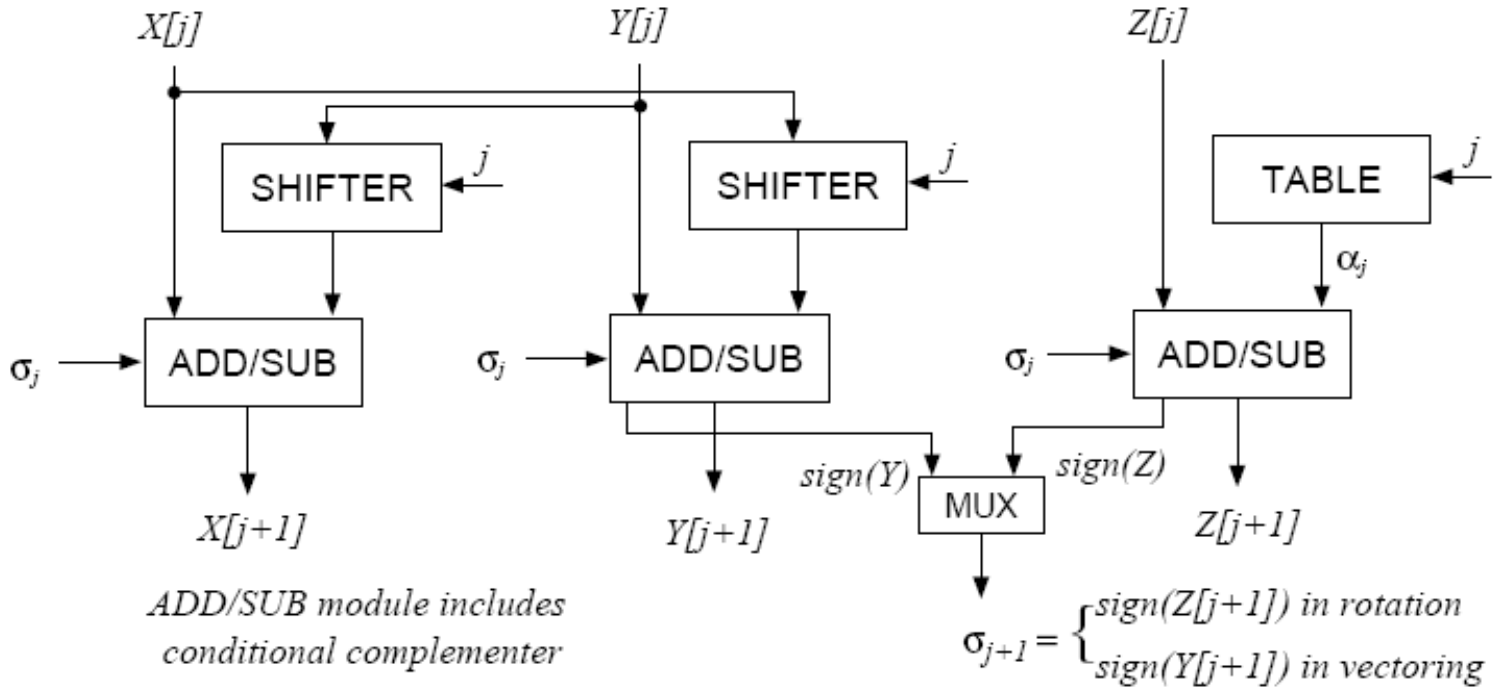
- V iteracijah x dobimo kot končni rezultat cos(fi),
- V iteracijah y dobimo kot končni rezultat sin(fi)

Vektorski način

- Začetni vektor (x_{in}, y_{in}) rotiramo tako, da postane y=0 x(0)=xin y(0)=yin in z(0)=zin
- Enačbe rotacije vektorja

$$
\begin{aligned}
x_f &= K(x_{in}^2 + y_{in}^2)^{1/2} \\
y_f &= 0 \\
z_f &= z_{in} + \tan^{-1}(\frac{y_{in}}{x_{in}})
\end{aligned}
$$

Vektorski način – podatkovna pot



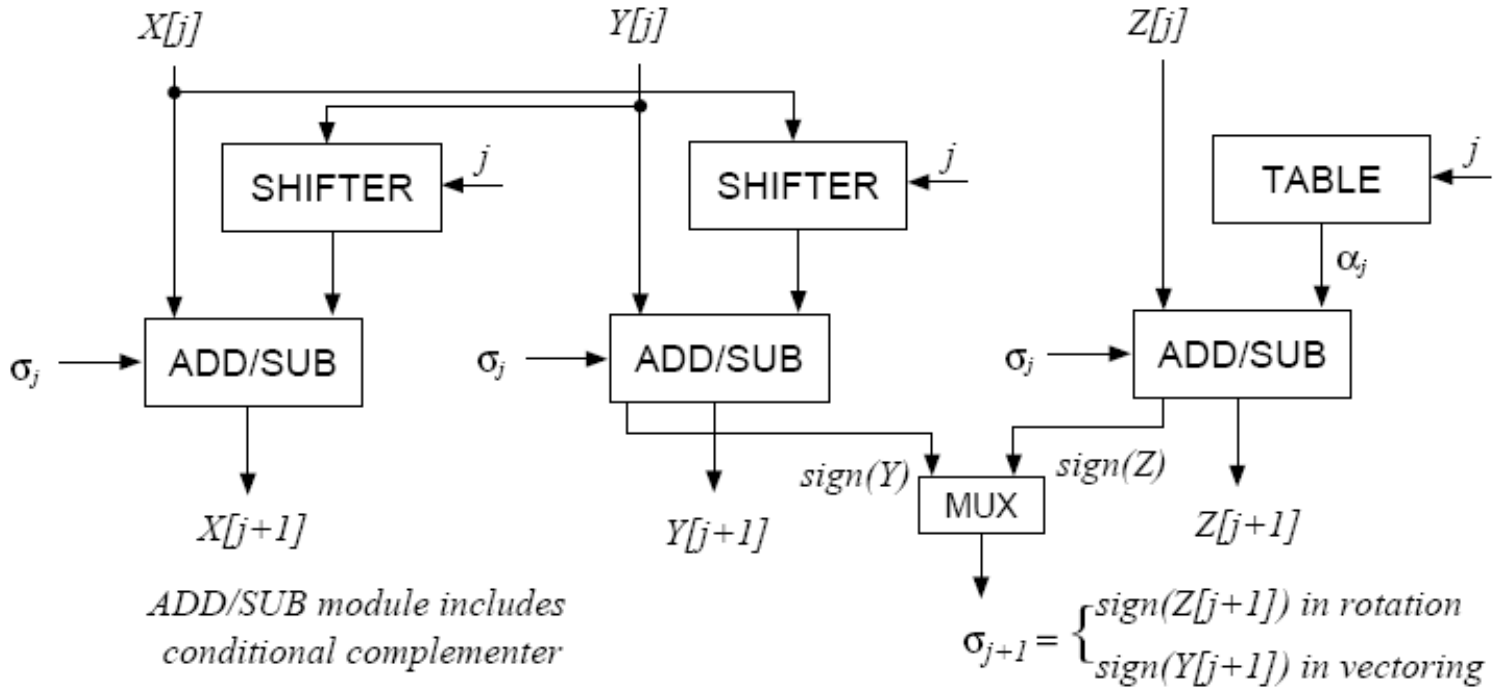
$$\begin{bmatrix} x_R \\ y_R \end{bmatrix} = \begin{bmatrix} \cosh\theta & \sinh\theta \\ \sinh\theta & \cosh\theta \end{bmatrix} \begin{bmatrix} x_{in} \\ y_{in} \end{bmatrix}$$

$$K_h[j] = (1 - 2^{-2j})^{1/2}$$

$$\begin{aligned} x[j+1] &= x[j] + \sigma_j 2^{-j} y[j] \\ y[j+1] &= y[j] + \sigma_j 2^{-j} x[j] \\ z[j+1] &= z[j] - \sigma_j \tanh^{-1}(2^{-j}) \end{aligned}$$

$\tanh^{-1} 2^0 = \infty$ (and $K_h[0] = 0$) zato začne pri j=1

Vektorski način – podatkovna pot



$$\begin{bmatrix} x_R \\ y_R \end{bmatrix} = \begin{bmatrix} \cosh\theta & \sinh\theta \\ \sinh\theta & \cosh\theta \end{bmatrix} \begin{bmatrix} x_{in} \\ y_{in} \end{bmatrix}$$

$$K_h[j] = (1 - 2^{-2j})^{1/2}$$

$$\begin{aligned} x[j+1] &= x[j] + \sigma_j 2^{-j} y[j] \\ y[j+1] &= y[j] + \sigma_j 2^{-j} x[j] \\ z[j+1] &= z[j] - \sigma_j \tanh^{-1}(2^{-j}) \end{aligned}$$

$\tanh^{-1} 2^0 = \infty$ (and $K_h[0] = 0$) zato začne pri j=1

Vektorski način – podatkovna pot

$$\begin{bmatrix} x_R \\ y_R \end{bmatrix} = \begin{bmatrix} \cosh\theta & \sinh\theta \\ \sinh\theta & \cosh\theta \end{bmatrix} \begin{bmatrix} x_{in} \\ y_{in} \end{bmatrix}$$

$$K_h[j] = (1 - 2^{-2j})^{1/2}$$

$$\begin{aligned} x[j+1] &= x[j] + \sigma_j 2^{-j} y[j] \\ y[j+1] &= y[j] + \sigma_j 2^{-j} x[j] \\ z[j+1] &= z[j] - \sigma_j \tanh^{-1}(2^{-j}) \end{aligned}$$

$\tanh^{-1} 2^0 = \infty$ (and $K_h[0] = 0$) zato začne pri j=1

$$K_h \approx 0.82816$$

$$\theta_{max} = 1.11817$$

Način vrtenja

$$\begin{aligned} x_f &= K_h(x_{in}\cosh\theta + y_{in}\sinh\theta) \\ y_f &= K_h(x_{in}\sinh\theta + y_{in}\cosh\theta) \\ z_f &= 0 \end{aligned}$$

Vektorski način

$$\begin{aligned} x_f &= K_h(x_{in}^2 - y_{in}^2)^{1/2} \\ y_f &= 0 \\ z_f &= z_{in} + \tanh^{-1}(\frac{y_{in}}{x_{in}}) \end{aligned}$$

Vektorski način – podatkovna pot

$$\begin{bmatrix} x_R \\ y_R \end{bmatrix} = \begin{bmatrix} \cosh\theta & \sinh\theta \\ \sinh\theta & \cosh\theta \end{bmatrix} \begin{bmatrix} x_{in} \\ y_{in} \end{bmatrix}$$

$$K_h[j] = (1 - 2^{-2j})^{1/2}$$

$$\begin{aligned} x[j+1] &= x[j] + \sigma_j 2^{-j} y[j] \\ y[j+1] &= y[j] + \sigma_j 2^{-j} x[j] \\ z[j+1] &= z[j] - \sigma_j \tanh^{-1}(2^{-j}) \end{aligned}$$

$\tanh^{-1} 2^0 = \infty$ (and $K_h[0] = 0$) zato začne pri j=1

$$K_h \approx 0.82816$$

$$\theta_{max} = 1.11817$$

Način vrtenja

$$\begin{aligned} x_f &= K_h(x_{in}\cosh\theta + y_{in}\sinh\theta) \\ y_f &= K_h(x_{in}\sinh\theta + y_{in}\cosh\theta) \\ z_f &= 0 \end{aligned}$$

Vektorski način

$$\begin{aligned} x_f &= K_h(x_{in}^2 - y_{in}^2)^{1/2} \\ y_f &= 0 \\ z_f &= z_{in} + \tanh^{-1}(\frac{y_{in}}{x_{in}}) \end{aligned}$$

Poenoteni CORDIC algoritem

- m=1 ➔ krožni koordinatni sistem
- m=-1 ➔ hiperbolični koordinatni sistem
- m=0 ➔ linearni koordinatni sistem

$$
\begin{aligned}
x[j+1] &= x[j] - m\sigma_j 2^{-j} y[j] \\
y[j+1] &= y[j] + \sigma_j 2^{-j} x[j] \\
z[j+1] &= \begin{cases} z[j] - \sigma_j \tan^{-1}(2^{-j}) & \text{if } m = 1 \\ z[j] - \sigma_j \tanh^{-1}(2^{-j}) & \text{if } m = -1 \\ z[j] - \sigma_j (2^{-j}) & \text{if } m = 0 \end{cases}
\end{aligned}
$$

$$K_m[j] = (1 + m2^{-2j})^{1/2}$$

$$z[j+1] = z[j] - \sigma_j m^{-1/2} \tan^{-1}(m^{1/2} 2^{-j})$$

Poenoteni CORDIC algoritem

Coordinates	Rotation mode $\sigma_j = sign(z[j])^+$	Vectoring mode $\sigma_j = -sign(y[j])^+$
Circular $(m = 1)$ $\alpha_j = \tan^{-1}(2^{-j})$ initial $j = 0$ $j = 0, 1, 2, .., n$ $K_1 \approx 1.64676$ $\theta_{max} \approx 1.74329$	$x_f = K_1(x_{in}\cos(z_{in}) - y_{in}\sin(z_{in}))$ $y_f = K_1(x_{in}\sin(z_{in}) + y_{in}\cos(z_{in}))$ $z_f = 0$	$x_f = K_1(x_{in}^2 + y_{in}^2)^{1/2}$ $y_f = 0$ $z_f = z_{in} + \tan^{-1}(\frac{y_{in}}{x_{in}})$
Linear $(m = 0)$ $\alpha_j = 2^{-j}$ initial $j = 0$ $j = 0, 1, , 2, ..., n$ $K_0 = 1$ $\theta_{max} = 2 - 2^{-n}$	$x_f = x_{in}$ $y_f = y_{in} + x_{in}z_{in}$ $z_f = 0$	$x_f = x_{in}$ $y_f = 0$ $z_f = z_{in} + \frac{y_{in}}{x_{in}}$
Hyperbolic $(m = -1)$ $\alpha_j = \tanh^{-1}(2^{-j})$ initial $j = 1$ $j = 1, 2, 3, 4, 4, 5...13, 13, ...$ $K_{-1} \approx 0.82816$ $\theta_{max} \approx 1.11817$	$x_f = K_{-1}(x_{in}\cosh(z_{in}) + y_{in}\sinh(z_{in}))$ $y_f = K_{-1}(x_{in}\sinh(z_{in}) + y_{in}\cosh(z_{in}))$ $z_f = 0$	$x_f = K_{-1}(x_{in}^2 - y_{in}^2)^{1/2}$ $y_f = 0$ $z_f = z_{in} + \tanh^{-1}(\frac{y_{in}}{x_{in}})$

$^+$ $sign(a) = 1$ if $a \geq 0$, $sign(a) = -1$ if $a < 0$.

Poenoteni CORDIC algoritem

m	Mode	Initial values			Functions	
		x_{in}	y_{in}	z_{in}	x_R	y_R or z_R
1	rotation	1	0	θ	$\cos\theta$	$y_R = \sin\theta$
-1	rotation	1	0	θ	$\cosh\theta$	$y_R = \sinh\theta$
-1	rotation	a	a	θ	ae^{θ}	$y_R = ae^{\theta}$
1	vectoring	1	a	$\pi/2$	$\sqrt{a^2+1}$	$z_R = \cot^{-1}(a)$
-1	vectoring	a	1	0	$\sqrt{a^2-1}$	$z_R = \coth^{-1}(a)$
-1	vectoring	$a+1$	$a-1$	0	$2\sqrt{a}$	$z_R = 0.5\ln(a)$
-1	vectoring	$a+\frac{1}{4}$	$a-\frac{1}{4}$	0	$\sqrt{a}$	$z_R = \ln(\frac{1}{4}a)$
-1	vectoring	$a+b$	$a-b$	0	$2\sqrt{ab}$	$z_R = 0.5\ln(\frac{a}{b})$

Načrtovanje digitalnih vezij

Asinhrona sekvenčna vezja

Asinhrona sekvenčna vezja

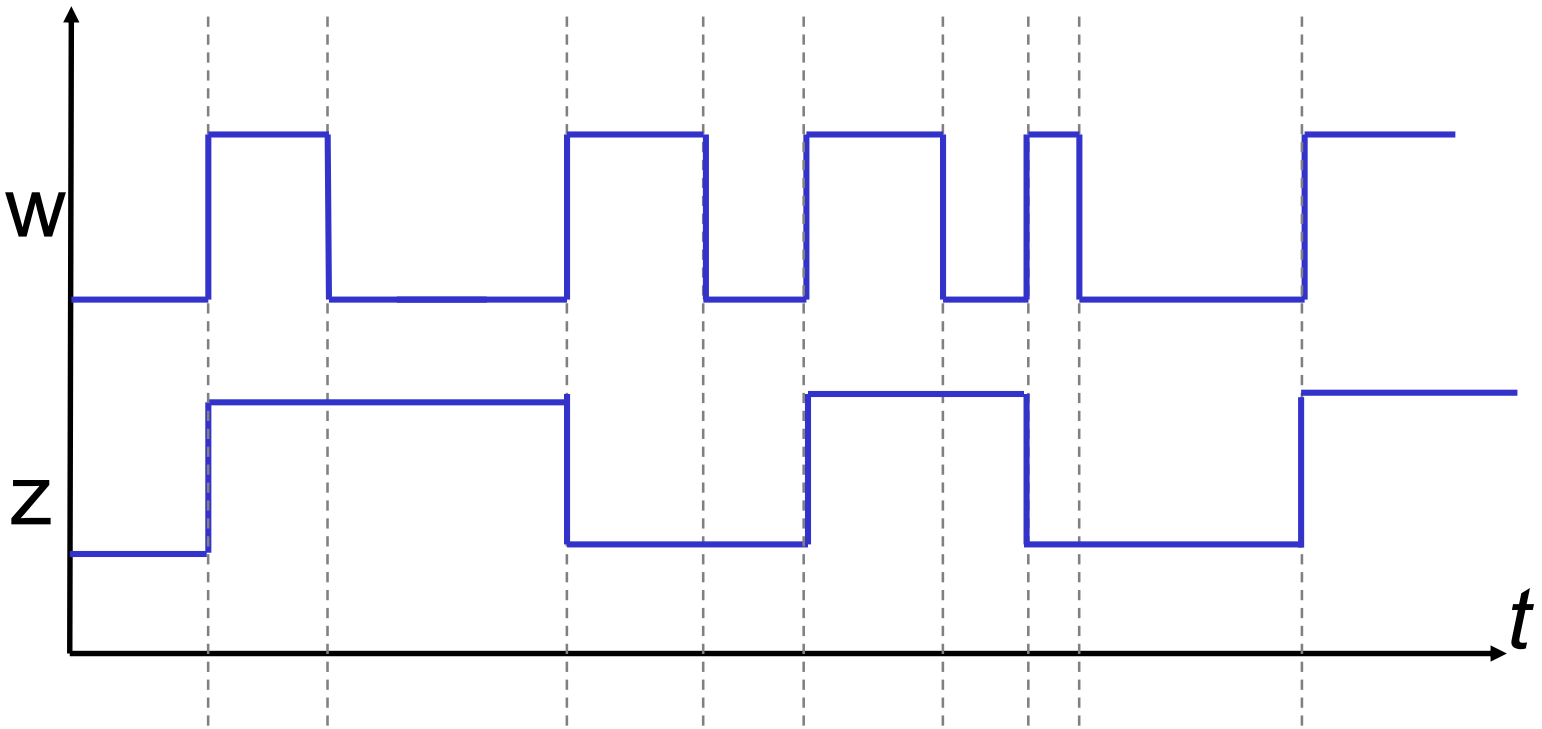
- Nivojsko (ne robno) krmiljena sekvenčna vezja.
- Sprememba stanja sekvenčnega vezja zaradi spremembe logičnega nivoja neke vhodne spremenljivke.
- Istočasna sprememba je nemogoča, lahko pa se zgodi sprememba več spremenljivk v časovnem intervalu, ki je krajši od preklopnega časa pomnilnih celic.

Uporaba asinhronih sekvenčnih vezij

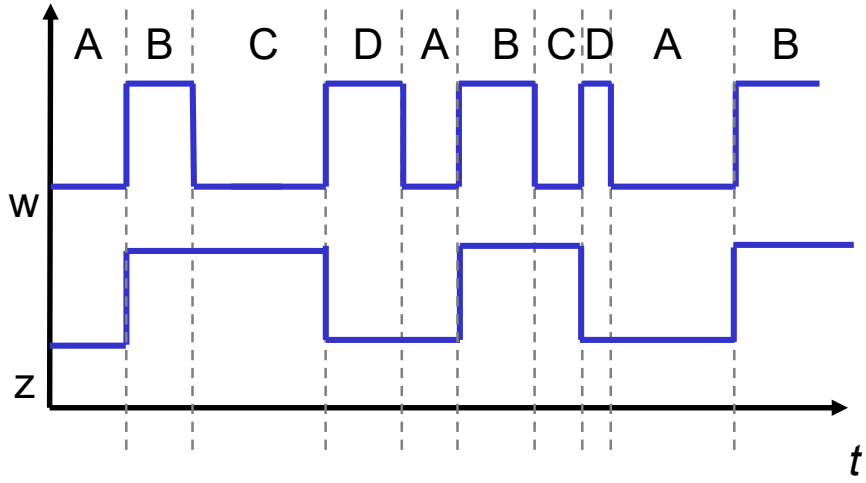
- Kaj se dogaja v sekvenčnem vezju sestavljenem iz pomnilnih celic, ki ne uporabljajo signala ure ampak samo podatkovne vhode?
- Zaradi spremembe vhodnih spremenljivk se spremeni lahko vrednost vhodnih funkcij pomnilnih celic.
- Stanje pomnilnih celic se prične spreminjati. Če ob spremenjenih stanjih vrednosti vhodnih funkcij ostanejo nespremenjene, pravimo da je tako stanje *stabilno*.

Sinteza asinhronega generatorja paritete

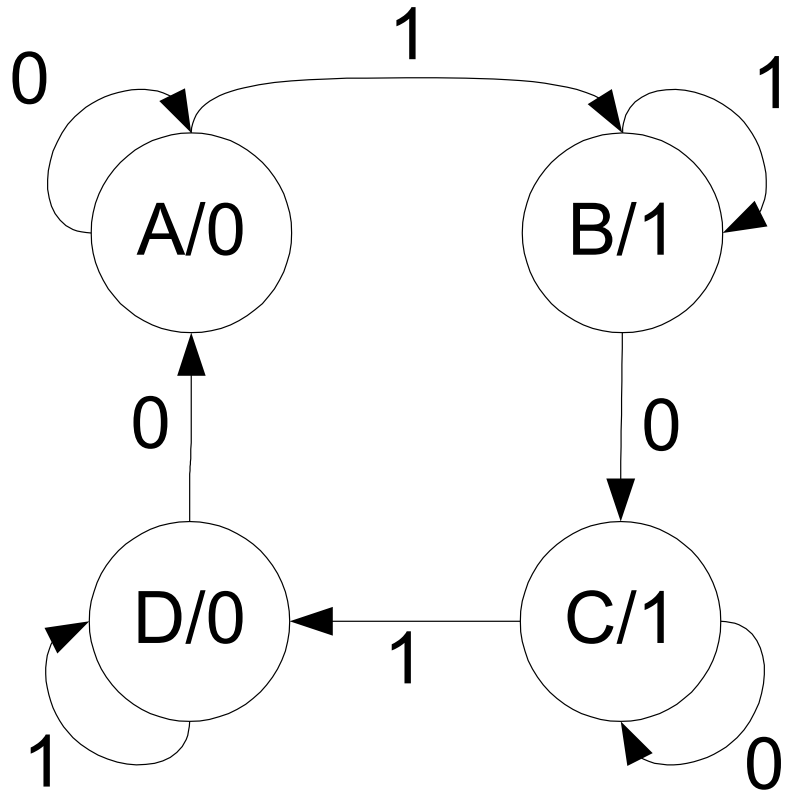
- Želimo realizirati asinhrono vezje, ki ima vhod w in izhod z. Izhod vezja bo z='0', če je bilo število predhodnih pulzov na vhodu liho in z='1', če je število predhodnih pulzov sodo.



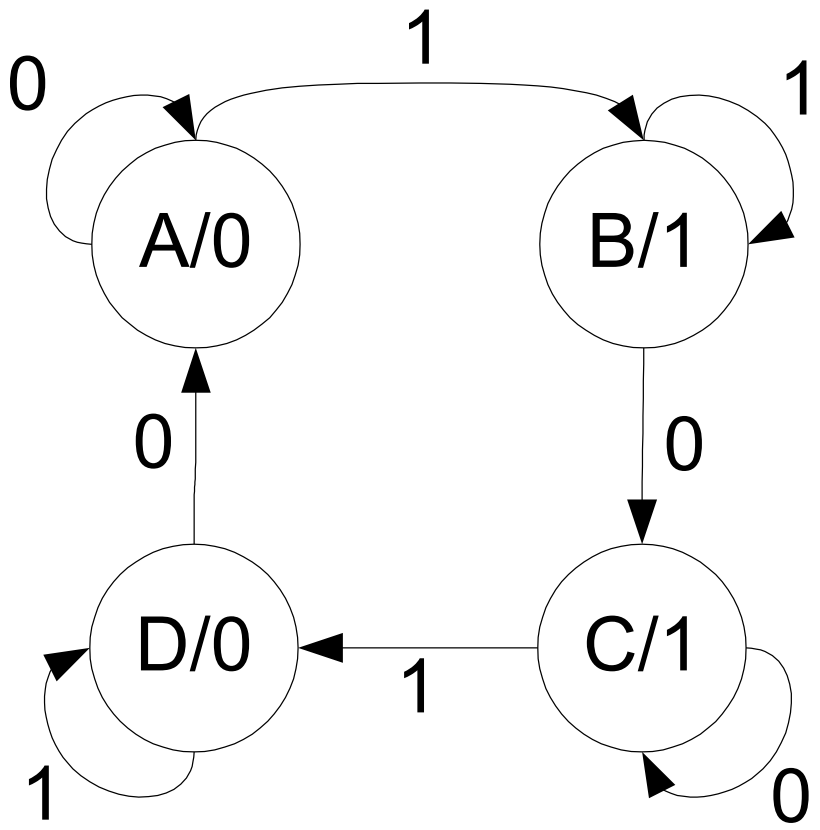
Sinteza asinhronega generatorja paritete



Zakaj ima Moore-ova izvedba avtomata 4 stanja in ne 2?



Sinteza asinhronega generatorja paritete



trenutno stanje	naslednje stanje		z
	0	1	
A	A	B	0
B	C	B	1
C	C	D	1
D	A	D	0

Kodiranje stanj asinhronega avtomata

trenutno stanje	naslednje stanje		z
	w=0	w=1	
A	A	B	0
B	C	B	1
C	C	D	1
D	A	D	0

Kodiranje stanj:

A	00
B	01
C	10
D	11

trenutno stanje	naslednje stanje		z
	w=0	w=1	
00	00	01	0
01	10	01	1
10	10	11	1
11	00	11	0

Če je trenutno stanje '11' in je na vhodu w='0' bi bilo novo stanje po tabeli '00'. Obe celici ne spremenita stanja ob istem času, zato imamo lahko ali stanje '10' ali '01'.

- '11'➔'10'. Ko pride v C bo z='0', kar je *narobe.*
- '11'➔'01'. Ko pride v B, je recimo vhod w='0', preskoči v C ➔ ponovna sprememba obeh mest kode stanja 01➔10. Od tod sledita spet dve možnosti … Takemu pojavu rečemo pobeg ali **race**.

Kodiranje stanj asinhronega avtomata

Pobegu se lahko izognemo,
če stanja obravnavamo kot vhode:

Največ ena spremenljivka stanja
se sme spremeniti naenkrat.

Novo
kodiranje stanj:

A	00
B	01
C	11
D	10

trenutno stanje y_2y_1	naslednje stanje Y_2Y_1		z
	w=0	w=1	
00	00	01	0
01	11	01	1
11	11	10	1
10	00	10	0

Statični hazard!

Kodiranje stanj asinhronega avtomata

trenutno stanje y_2y_1	naslednje stanje Y_2Y_1		z
	w=0	w=1	
00	00	01	0
01	11	01	1
11	11	10	1
10	00	10	0

$Y_1 = w \cdot y_2' + w' \cdot y_1 + y_1 \cdot y_2'$

$Y_2 = w \cdot y_2 + w' \cdot y_1 + y_1 \cdot y_2$

$z = y_1$

Funkcije asinhronih avtomatov vedno minimiziramo v Karnaugh-jevem diagramu!

Y_2:

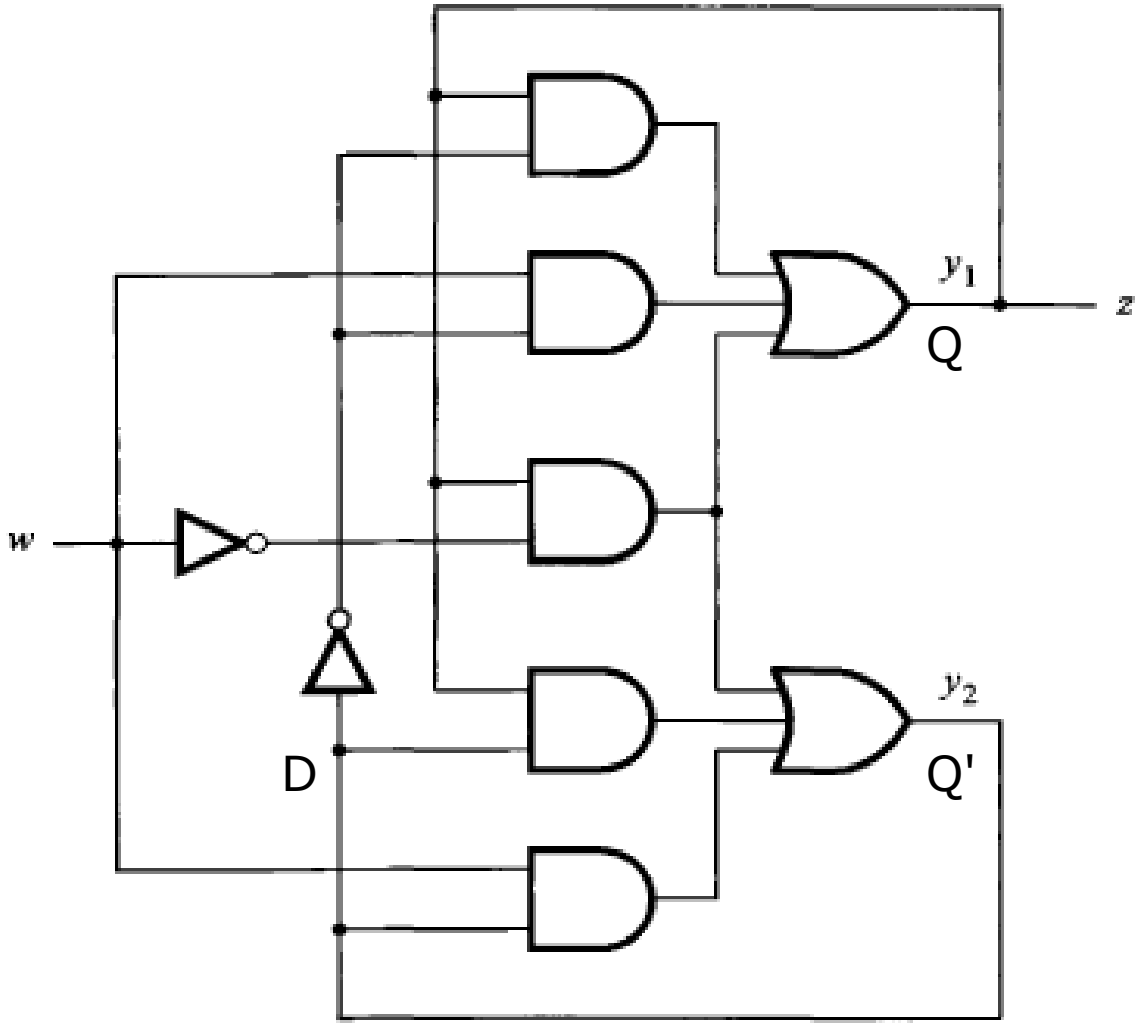
w \ y_2y_1	00	01	11	10
0	0	1	1	0
1	0	0	1	1

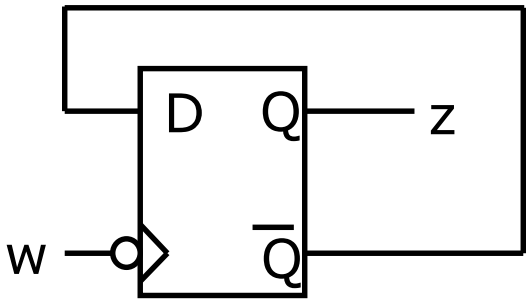
Y_1:

w \ y_2y_1	00	01	11	10
0	0	1	1	0
1	1	1	0	0

Statični hazard!

Kodiranje stanj asinhronega avtomata

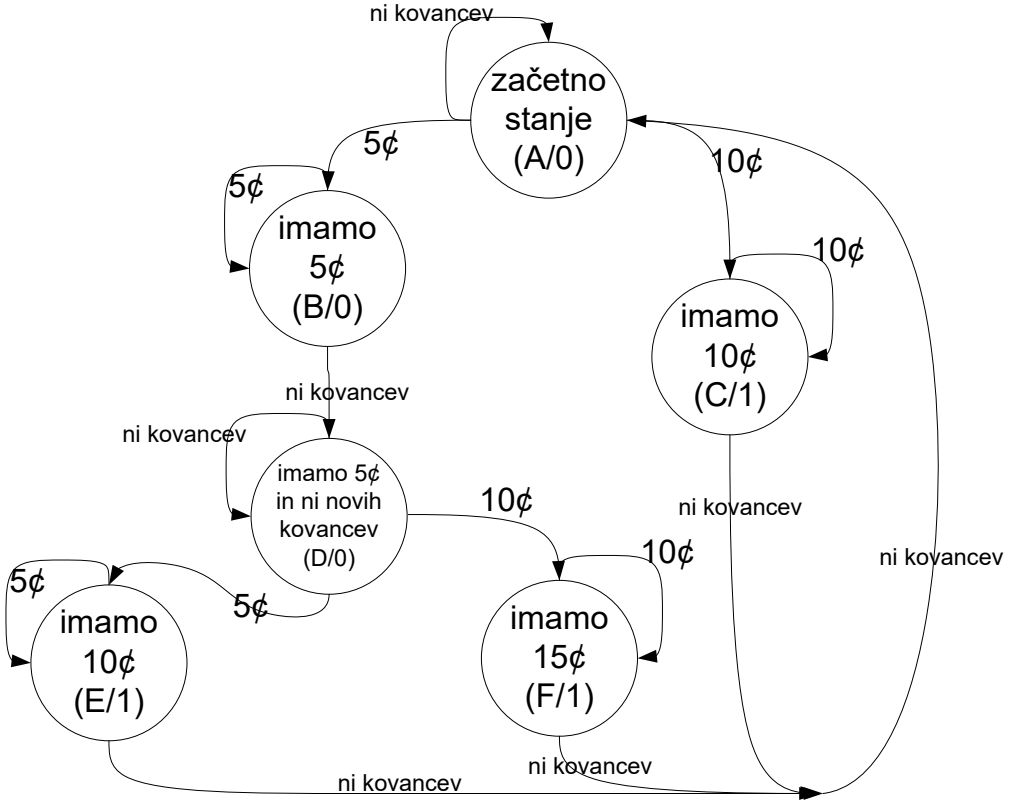




Master-slave D-FF prožen na negativni rob signala ure.

Minimizacija stanj asinhronega avtomata

- Asinhrono vezje avtomata za kavo, ki ima vhoda za 10¢ in 5¢. Cena kave je 10¢. Ko se nabere dovolj denarja (≥10¢) postane izhod kava='1', sicer je '0'. Avtomat ne vrača denarja.

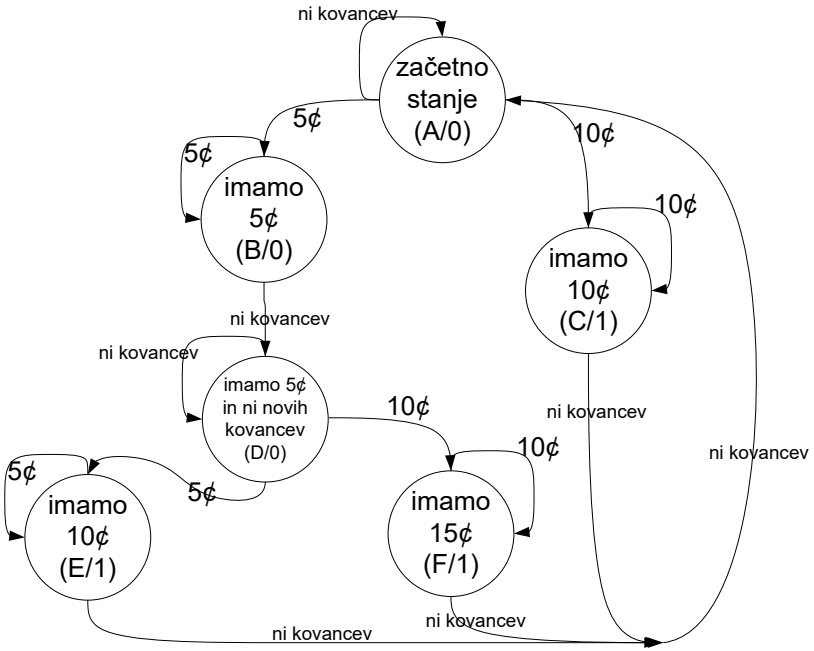


vhod	koda
ni kovancev	00
5¢	01
10¢	10
ni možno*	11

* ker ne moremo naenkrat vreči oba kovanca v avtomat

Redukcija stanj asinhronega avtomata

- Tabeli, ki ima samo eno stabilno stanje v vrstici, pravimo **primitivna tabela stanj**.



	naslednje stanje				kava
trenutno stanje	00	01	10	11	
A	A	B	C	–	0
B	D	B	–	–	0
C	A	–	C	–	1
D	D	E	F	–	0
E	A	E	–	–	1
F	A	–	F	–	1

Redukcija stanj asinhronega avtomata

- Iz definicije ekvivalentnosti stanj sledi, če sta stanji S_i and S_j ekvivalentnim potem so ekvivalentni tudi ustrezni ***k-nasledniki***
- Ob tem lahko zasnujemo postopek minimizacije, ki jemlje FSM kot niz. V postopku minimizacije tak niz razdelimo v ***razdelke,*** ki so sestavljeni iz neekvivalentnih podnizov
- **Definicija**: **Razdelek** sestoji iz enega ali več blokov, kjer vsak blok sestoji iz podniza stanj, ki so lahko ekvivalentna, vendar so stanja v enega bloka neekvivalentna stanjem v drugih blokih

Redukcija stanj asinhronega avtomata

	naslednje stanje				kava
trenutno stanje	00	01	10	11	
A	A	B	C	–	0
B	D	B	–	–	0
C	A	–	C	–	1
D	D	E	F	–	0
E	A	E	–	–	1
F	A	–	F	–	1

Najprej minimizacija z razdelki:

1. Ekvivalentni vrstici v tabeli imata enak izhod.
2. Stabilna stanja in nedoločene kombinacije (-) morajo biti v istih stolpcih naslednjih stanj.

Začetni razdelek vsebuje vsa stanja v eni grupi

$P_1=(ABCDEF)$

Redukcija stanj asinhronega avtomata

- Naslednji razdelek loči stanja, ki imajo različne izhode:
  P_2=(AD)(B)(CF)(E)

- Stanji A in D imata stabilni stanji ob kombinaciji vhoda 00, izhod 0 in nedoločena stanja na istih mestih.

- Stanji C in F imata stabilni stanji ob kombinaciji vhoda 10, izhod 1 in nedoločena stanja na istih mestih.

- Stanji B in E imata stabilni stanji ob kombinaciji vhoda 01, vendar je izhod različen.

	naslednje stanje				kava
trenutno stanje	00	01	10	11	
A	A	B	C	–	0
B	D	B	–	–	0
C	A	–	C	–	1
D	D	E	F	–	0
E	A	E	–	–	1
F	A	–	F	–	1

Redukcija stanj asinhronega avtomata

- Naslednika stanj (AD) sta (AD) za vhod 00, (BE) za 01 in (CF) za vhod 11.
- Par (BE) ni v istem razdelku kot (AD) v iteraciji P_2 zato A in D nista ekvivalentna: P_2=(AD)(B)(CF)(E)
- Naslednika stanj (CF) sta (AA) za vhod 00 in (CF) za vhod 11.
- Naslednja iteracija razdelkov bo torej: P_3=(A)(D)(B)(CF)(E)

	naslednje stanje				kava
trenutno stanje	00	01	10	11	
A	A	B	C	–	0
B	D	B	–	–	0
C	A	–	C	–	1
D	D	E	F	–	0
E	A	E	–	–	1
F	A	–	F	–	1

Iteracija P_3=(A)(D)(B)(CF)(E)
je tudi zadnja, ker nimamo več česa razdružiti.
Zato sledi, da sta stanji C in F ekvivalentni!

Redukcija stanj asinhronega avtomata

- Novo tabelo dobimo tako, da zamenjamo vse F ➔ C.
- Naslednji korak je združevanje vrstic.
- Edina vrstica, ki jo lahko združimo je C:
- Vrstic A in C ne moremo združiti, ker imata različen izhod.
- Združimo lahko C in E. Stabilno stanje pojavi pri vhodu 01 in 10.

	naslednje stanje				kava
trenutno stanje	00	01	10	11	
A	A	B	C	–	0
B	D	B	–	–	0
C	A	–	C	–	1
D	D	E	C	–	0
E	A	E	–	–	1

	naslednje stanje				kava
trenutno stanje	00	01	10	11	
A	A	B	C	–	0
B	D	B	–	–	0
C	A	C	C	–	1
D	D	C	C	–	0

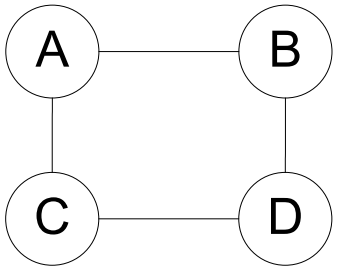
Redukcija stanj asinhronega avtomata

- Stanji S_i in S_j sta združljivi, ko ni sporov za vsako vhodno kombinacijo.
- Za vsako vhodno kombinacijo mora držati vsaj ena izmed trditev:
  - stanji S_i in S_j imata istega naslednika ali
  - stanji S_i in S_j sta stabilni
  - naslednik S_i ali S_j (ali obeh) je nedoločen.
- Stanji S_i in S_j morata imeti enak izhod.

Kodiranje stanj asinhronega avtomata

- Pobeg (race) se lahko zgodi, če se med prehajanjem stanj spremenita več kot ena notranja spremenljivka (Hamming-ova razdalja = 1)
- To zagotovimo z **diagrami prehodov** (transition diagram, state adjacency diagram)
- Diagram prehoda narišemo kot nov diagram stanj iz minimizirane tabele prehodov, ki vsebuje samo prehode med stanji:
  - Smeri prehodov nas ne zanimajo (A➔B je isto kot B➔A)
  - Pogoji, kjer stanja ostajajo sama v sebi (zank nazaj na izvorno stanje ne rišemo)

	naslednje stanje				kava
trenutno stanje	00	01	10	11	
A	A	B	C	–	0
B	D	B	–	–	0
C	A	C	C	–	1
D	D	C	C	–	0



A=00
B=01
D=11
C=10

Hamming-ova razdalja
med dvema številoma je enaka številu bitov ki se spremenijo
če primerjamo dve števili med sabo:
0110 in 0100 ➔ razdalja je 1
0110 in 1101 ➔ razdalja je 3

Kodiranje stanj asinhronega avtomata – označevanje poti prehodov

- Na prejšnji prosojnici smo določili kodiranja stanj precej enostavno, ker je imel diagram prehajanja samo 4 stanja. Kaj če jih je več?
- Podana je minimalna oblika tabele prehajanja stanj za asinhron avtomat:

	naslednje stanje				izhod
trenutno stanje	00	01	10	11	
A	A	B	C	–	0
B	D	B	–	–	0
C	G	C	C	–	0
D	D	C	F	–	0
F	A	F	F	–	1
G	G	F	F	–	0

Kodiranje stanj asinhronega avtomata – označevanje poti prehodov

- Minimalno tabelo prehajanja stanj sledimo po vrsticah:
- Na stabilna stanja postavimo številke poti (1…8)
- Če se isto stabilno stanje nahaja v drugem stolpcu mu damo drugo številko
- V stolpcih tabele zamenjamo preostala nestabilna stanja s številkami ustreznih stabilnih stanj v istem stolpcu

	naslednje stanje				izhod
trenutno stanje	00	01	10	11	
A	A	B	C	–	0
B	D	B	–	–	0
C	G	C	C	–	0
D	D	C	F	–	0
F	A	F	F	–	1
G	G	F	F	–	0

	naslednje stanje				izhod
trenutno stanje	00	01	10	11	
A	1	2	4	–	0
B	5	2	–	–	0
C	8	3	4	–	0
D	5	C	F	–	0
F	1	6	7	–	1
G	8	6	7	–	0

Kodiranje stanj asinhronega avtomata – označevanje poti prehodov

- Iz minimalne tabele prehajanja stanj narišemo
  Diagram prehoda narišemo kot nov diagram stanj, ki vsebuje samo prehode med stanji:
  - Smeri prehodov nas ne zanimajo (A➔B je isto kot B➔A)
  - Poti, kjer stanja ostajajo sama v sebi ne rišemo (zank nazaj na izvorno stabilno stanje ne rišemo)

	naslednje stanje				izhod
trenutno stanje	00	01	10	11	
A	A	B	C	–	0
B	D	B	–	–	0
C	G	C	C	–	0
D	D	C	F	–	0
F	A	F	F	–	1
G	G	F	F	–	0

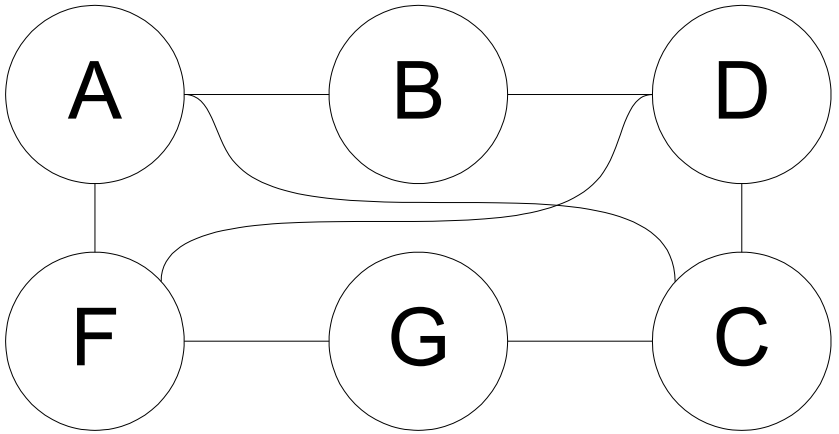
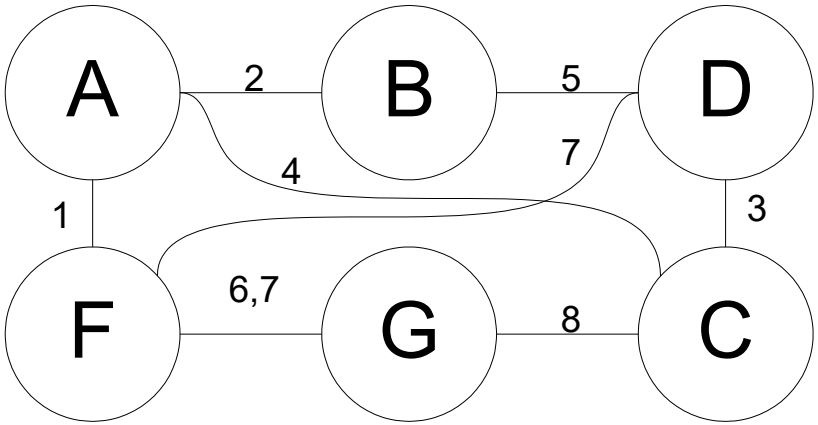
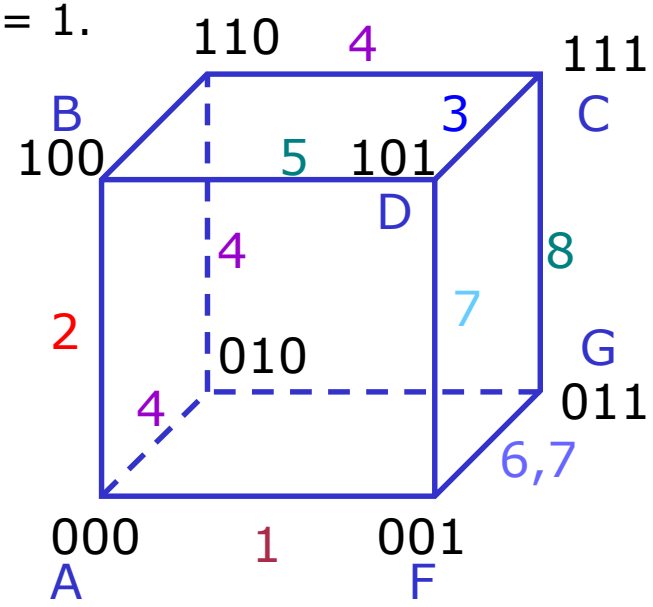
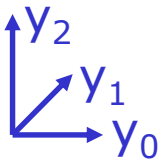


Diagram prehodov

- Stanja kodiramo tako, da poskušamo "napeti" dobljen diagram prehodov na n-dimenzionalno kocko.
- Na danem zgledu se to ne da enostavno narediti (recimo pot 4 A➔C), zato vpeljemo dodatni stanji (010, 110) preko katerih preidemo iz A➔C da bo Hamming-ova razdalja = 1.





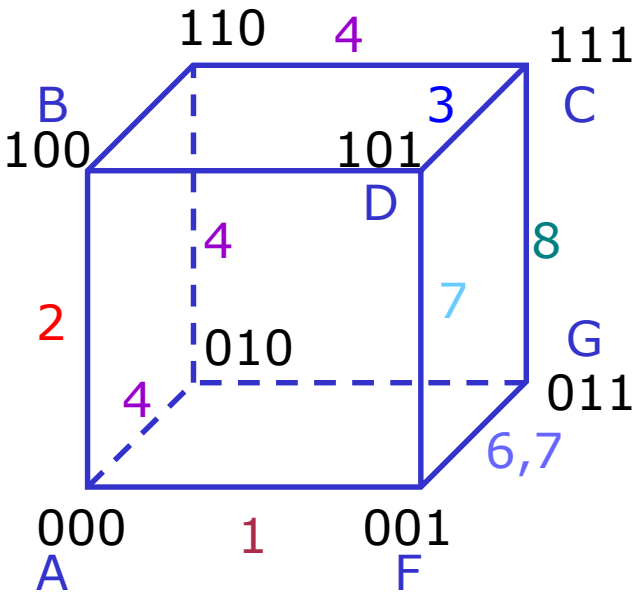


Koda stanja: $y_2y_1y_0$

Diagram prehodov

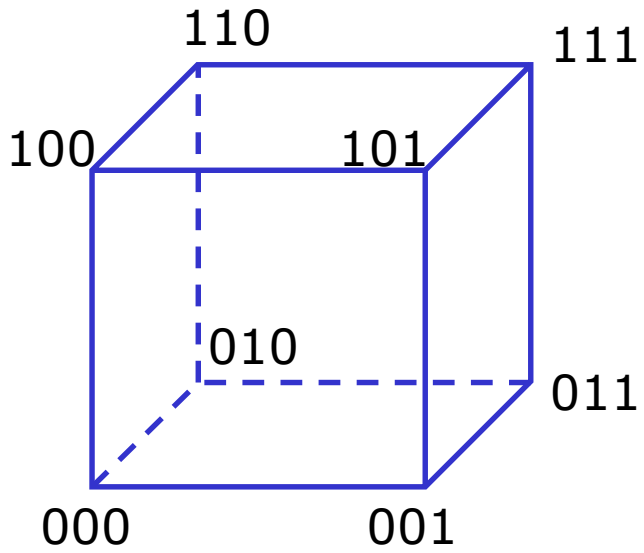
- Dobljene kode stanj vpišemo v minimalno tabelo prehajanja stanj in avtomat realiziramo do konca s Karnaugh-jevimi diagrami
- Pot 4 se od ostalih razlikuje po hitrosti, saj bo avtomat rabil 3·Δt da pride iz A➔C!

		naslednje stanje				izhod
	trenutno stanje	00	01	10	11	
A	000	000	100	111	–	0
B	100	101	100	–	–	0
C	111	011	111	111	–	0
D	101	101	111	001	–	0
F	001	000	001	001	–	1
G	011	011	001	001	–	0



Kodiranje stanj asinhronega avtomata

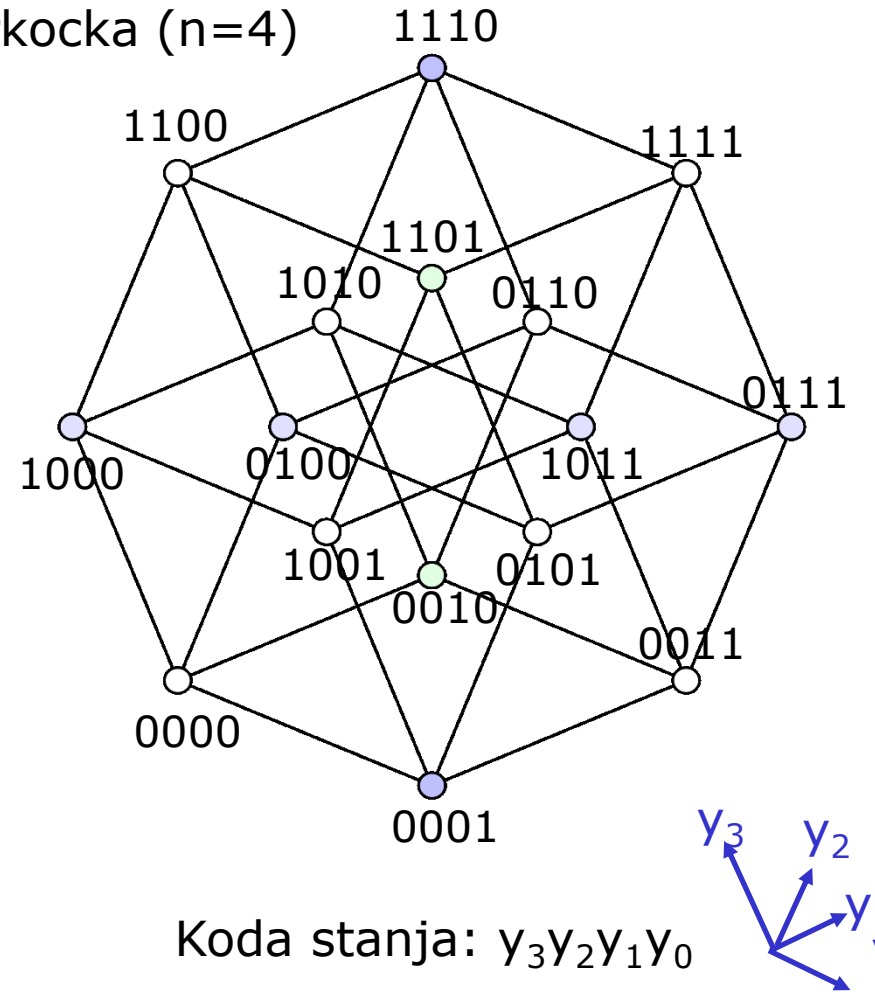
kocka (n=3)



Koda stanja: $y_2y_1y_0$

Vir: Wikipedia: Hypercube graph

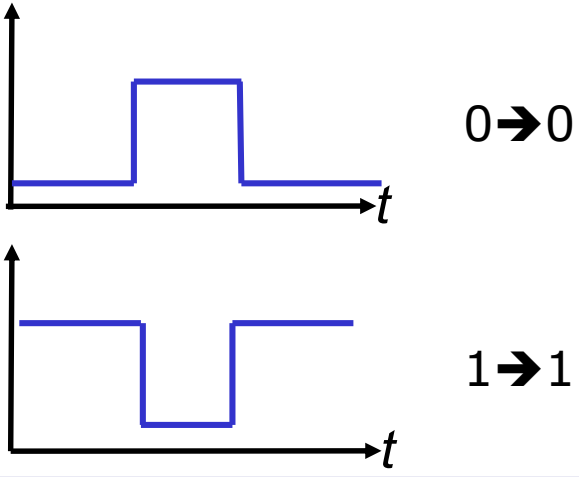
hiperkocka (n=4)



Koda stanja: $y_3y_2y_1y_0$

Hazard

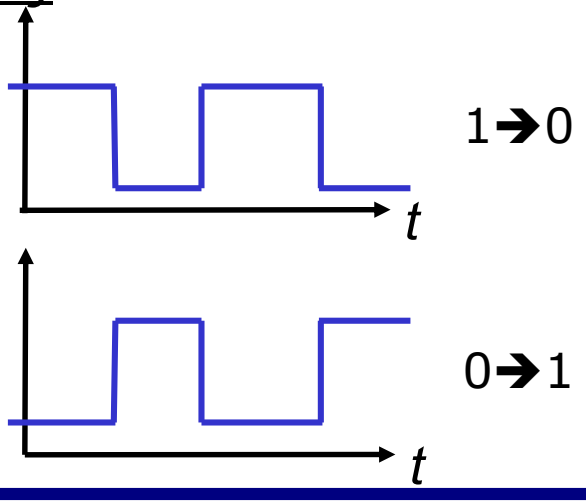
- *Statični hazard* obstaja, ko pričakujemo da naj bi nek signal ostal na logičnem nivoju medtem ko se vhodna spremenljivka spreminja, vendar za kratek čas spremeni logično vrednost, (oscilira) nato pa *se vrne nazaj* na izhodiščni logični nivo.



0➔0

1➔1

- *Dinamični hazard* obstaja, ko pričakujemo da naj bi nek signal ostal na logičnem nivoju medtem ko se vhodna spremenljivka spreminja, vendar za kratek čas spremeni logično vrednost (oscilira), nato pa se *ne vrne nazaj*.



1➔0

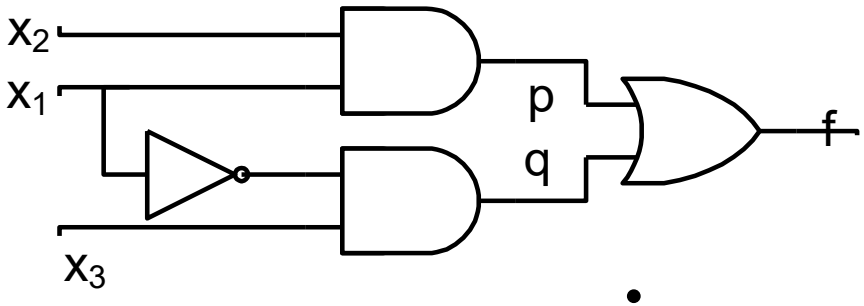
0➔1

Statični hazard

- Podana je funkcija:

$$f=x_1 \cdot x_2 + x_1' \cdot x_3$$

- Ob t=0 je na njenem vhodu $x_1=x_2=x_3=1$, kar pomeni da je f=1
- Po nekem času se x_1 spremeni iz 1➔0.
- Izhod bi moral biti še vedno 1.
- Če ne upoštevamo zakasnitve vrat – ni problema ➔ f=1
- Če pogledamo z upoštevanjem zakasnitve širjenja pa …

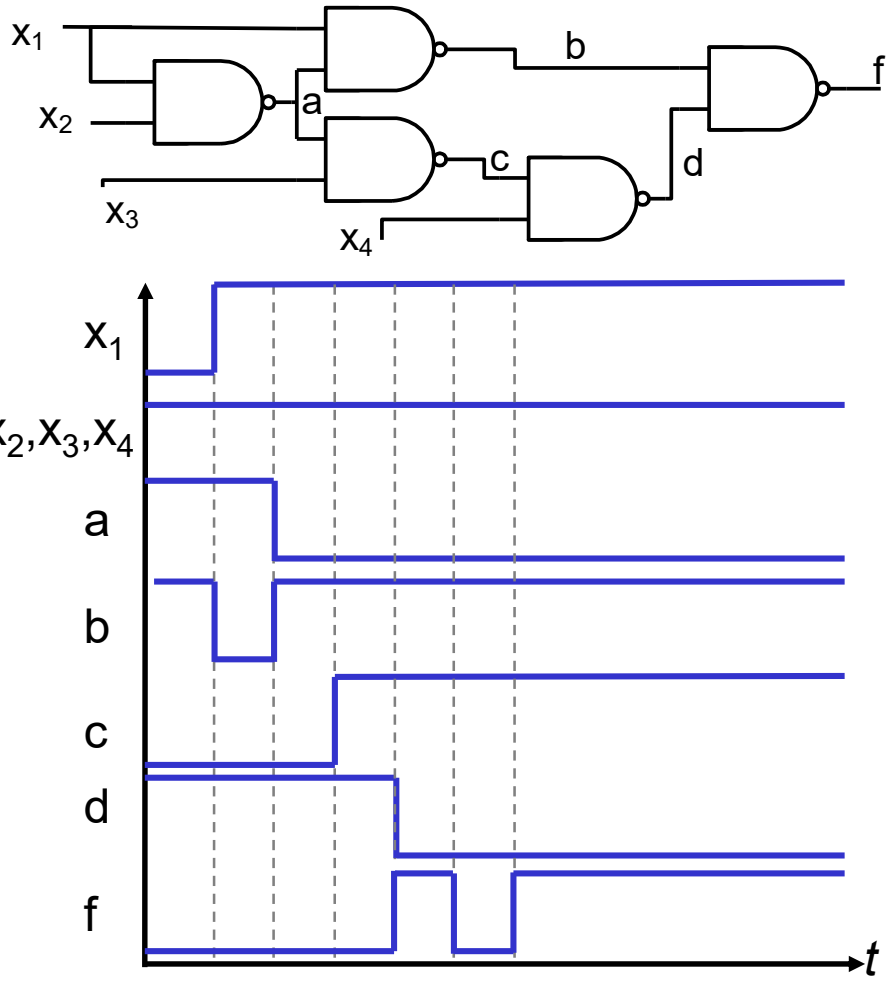


- Od vhodov do OR vrat se poti signalov razlikujejo (x_2 ima inverter)
- Točka *p* bo zato prej postala 0 preden *q* postane 1.
- Za kratek čas bosta *p=q=0* kar pomeni da bo f=0, preden zopet postane f=1.
- Dobimo špico (glitch) 1➔0➔1 katere vzrok je statični hazard.

Dinamični hazard

- Podana je funkcija:

  $f=x_1 \cdot x_2' + x_3' \cdot x_4 + x_1' \cdot x_4$
- Če bi jo realizirali neposredno – ni nevarnosti hazarda (ne statičnega, ne dinamičnega)
- Dinamični hazard povzroča *zasnova vezja*, po kateri obstajajo različne poti širjenja signala
- Vezja, ki vsebujejo dinamični hazard, vsebujejo tudi statičnega.
- Na dinamični hazard običajno naletimo med dekompozicijo funkcije (večnivojska vezja)
- Enostavno se mu izognemo z dvonivojskimi vezji (kadar je mogoče).



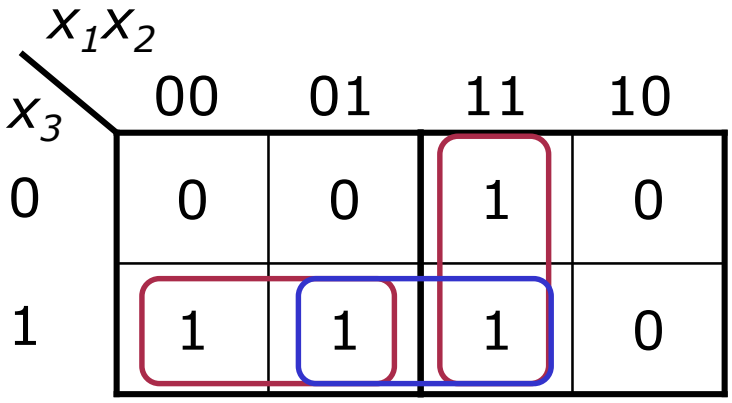
Statični hazard

- Narišemo Karnaugh-jev diagram in označimo vsebovalnike ki tvorijo realizacijo

$$f=x_1 \cdot x_2 + x_1' \cdot x_3$$

- Ta dva člena predstavljata vsebovalnike in obenem vzrok hazarda: Ni namreč člena, ki bi povezoval oba vsebovalnika.

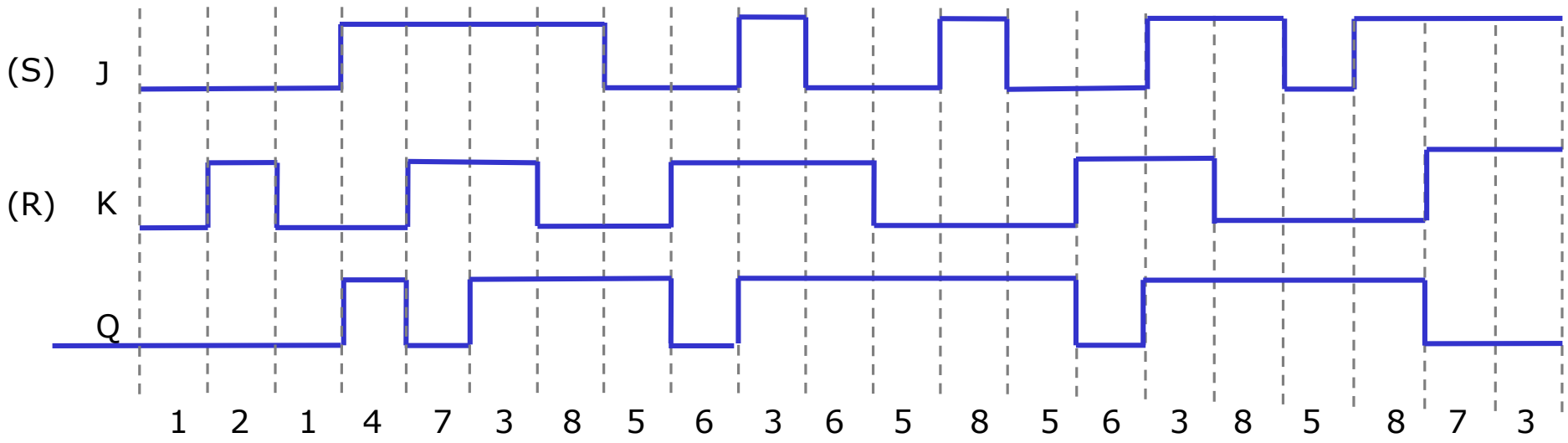
$$f=x_1 \cdot x_2 + x_1' \cdot x_3 + x_2 \cdot x_3$$



Potencialni statični hazard obstaja, če dve sosednji '1' nista pokriti samo z enim členom.

Hazard odpravimo tako da dodamo še tretji člen, ki pokrije vmesni '1'.

Časovni diagram asinhronega JK-FF



	Q(t)	JK	Q(t+1)	Pomen stanja
1	0	00	0	HOLD 0➔0
2	0	01	0	RESET 0➔0
3	0	11	1	INVERT 0➔1
4	0	10	1	SET 0➔1
5	1	00	1	HOLD 1➔1
6	1	01	0	RESET 1➔0
7	1	11	0	INVERT 1➔0
8	1	10	1	SET 1➔1

Q(t)	JK				Q(t+1)
	00	01	11	10	
1	1	2	3	4	0
2	1	2	3	4	0
3	5	6	3	4	1
4	5	2	7	4	1
5	5	2	7	4	1
6	5	6	3	4	1
7	1	2	7	8	0
8	1	2	7	8	0

Sinteza asinhrone JK pomnilne celice

- Primer sinteze JK pomnilne celice, občutljive na prednjo fronto signala na vhodih J in K s pomočjo zapahov (latch) in kombinacijskih vezij.
- Primitivna tabela prehajanja stanj ima 5 stolpcev:
  - (4 za kombinacije 00, 01, 10, 11 vhodov J in K) in
  - 1 stolpec za vrednost izhoda Q.

Q(t)	JK				Q
	00	01	11	10	
1	1	2	3	4	0
2	1	2	3	4	0
3	5	6	3	4	1
4	5	2	7	4	1
5	5	2	7	4	1
6	5	6	3	4	1
7	1	2	7	8	0
8	1	2	7	8	0

Sinteza asinhrone JK pomnilne celice

- Stanja:
- stabilna ➔ *k*
- nestabilna ➔ *k*
- neopredeljena ➔ –

Q(t)	JK				Q
	00	01	11	10	
1	1	2	3	4	0
2	1	2	3	4	0
3	5	6	3	4	1
4	5	2	7	4	1
5	5	2	7	4	1
6	5	6	3	4	1
7	1	2	7	8	0
8	1	2	7	8	0

Sinteza asinhrone JK pomnilne celice

Q(t)	JK				Q
	00	01	11	10	
1	1	2	–	4	0
2	1	2	3	–	0
3	–	6	3	4	1
4	5	–	7	4	1
5	5	2	–	4	1
6	5	6	3	–	1
7	–	2	7	8	0
8	1	–	7	8	0

Najprej določimo kateri prehodi se lahko sploh zgodijo:
2 vhoda naenkrat se ne moreta spremeniti.

Sinteza asinhrone JK pomnilne celice

Naslednji korak je določanje združljivih vrstic:

- Vrstici sta združljivi, če :
  - jima pripada ista izhodna črka in
  - imata v vsakem stolpcu enako stanje, ali
  - pri eni ali obeh vrsticah stanje ni definirano ➔ –
- Paroma združljive vrstice nadomestimo z novo vrstico.
- V posameznih stolpcih ima ta vrstica vrednosti vrstic, ki smo jih združili s tem da prevlada oznaka po naslednji prioriteti:
  - stabilno stanje,
  - nestabilno stanje,
  - nedoločeno stanje
- Rezultat združevanja je reducirana tabelo prehajanja stanj.

Q(t)	JK				Q
	00	01	11	10	
1	(1)	2	–	4	0
2	1	(2)	3	–	0
3	–	6	(3)	4	1
4	5	–	7	(4)	1
5	(5)	2	–	4	1
6	5	(6)	3	–	1
7	–	2	(7)	8	0
8	1	–	7	(8)	0

Združljiva stanja: 1 in 2 ➔ a 3 in 6 ➔ b 4 in 5 ➔ c 7 in 8 ➔ d

Sinteza asinhrone JK pomnilne celice

JK				Q
00	01	11	10	
a	*a*	*b*	*c*	*0*
c	*b*	*b*	*c*	*1*
c	*a*	*d*	*c*	*1*
a	*a*	*d*	*d*	*0*

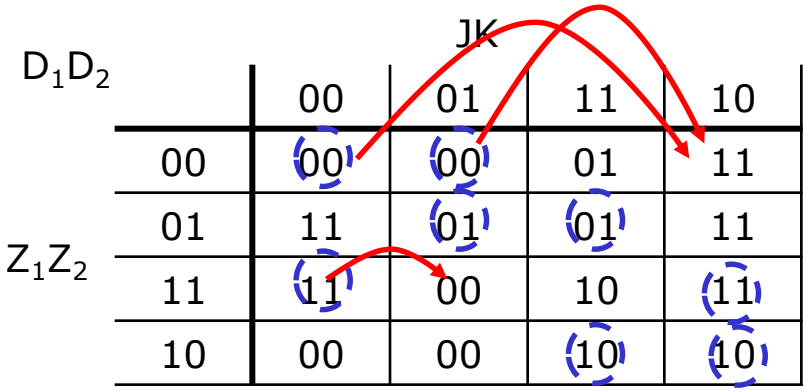
Recimo da izberemo tako kodiranje stanj:

z_1	z_2	stanje
0	*0*	*a*
0	*1*	*b*
1	*1*	*c*
1	*0*	*d*

Sinteza asinhrone JK pomnilne celice

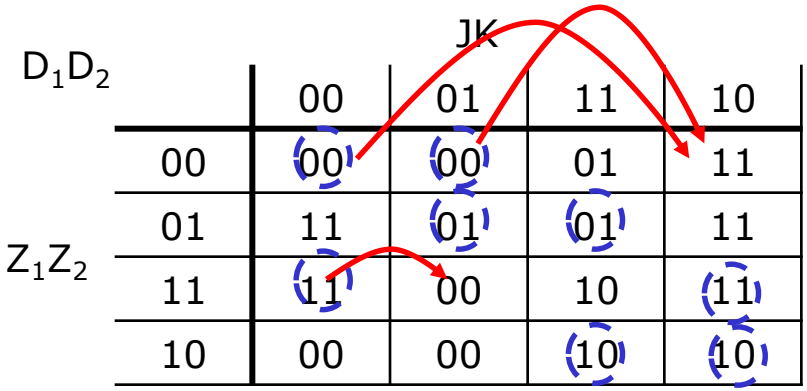
Reducirana tabelo prehajanja stanj:

- Če se pri prehodu iz enega stabilnega stanja v drugo stabilno stanje spremeni samo vrednost ***ene*** pomnilne celice ➔ *prehod* (transition).
- Prehod traja dokler opazovana spremenljivka ne doseže končne vrednosti.
- Kaj če se spremenita vrednosti dveh ali več pomnilnih celic?
- V našem primeru sta to dva primera:
  1. Če se ob JK=00 ali JK=01 pojavi sprememba na JK=10 (prehod 00 ➔ 11)
  2. Če se ob JK=00 pojavi sprememba na JK=01 (prehod 11 ➔ 00)



Sinteza asinhrone JK pomnilne celice

- Pomnilne celice niso enako hitre. Zaradi razlike v hitrosti se za kratek čas pojavijo neka vmesna stanja (Stanji 01 in 10).
- Kaj če je katero od teh dveh stanj stabilno za nove vrednosti vhodnih spremenljivk?
- V tem primeru se proces spreminjanja vrednosti notranjih spremenljivk lahko konča in vezje preide v nepravilno stanje.
- Prehodom, pri katerih se spreminja več notranjih spremenljivk hkrati, pravimo **kritični prehodi** (race).
- V prvem primeru konča vezje v napačnem stabilnem stanju 10, v drugem pa v napačnem stabilnem stanju 01.

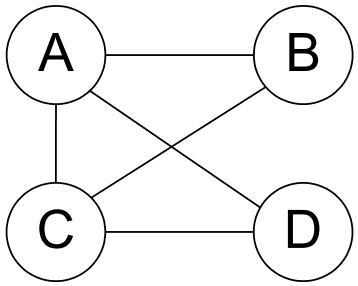


Sinteza asinhrone JK pomnilne celice

JK				Q
00	01	11	10	
(a)	(a)	b	c	0
c	(b)	(b)	c	1
(c)	a	d	(c)	1
a	a	(d)	(d)	0

z_1	z_2	stanje
0	0	a
0	1	b
1	1	c
1	0	d

Ali lahko ta diagram napnemo na stranice kvadrata?

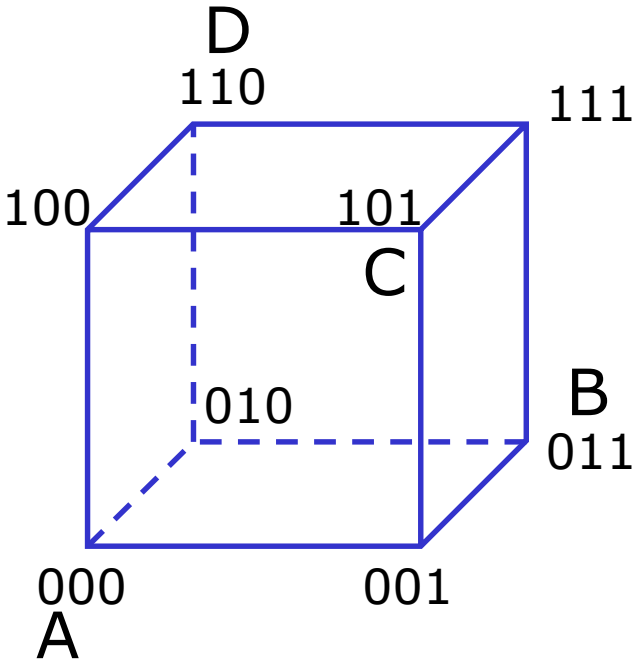


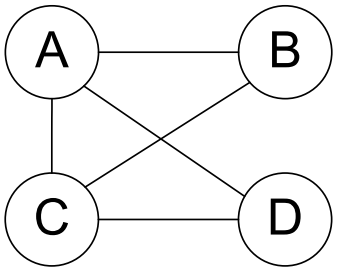
A=00
B=01
D=11
C=10

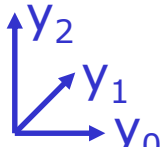
Ne moremo, zato bo število FF 3, in ne 2!

Sinteza asinhrone JK pomnilne celice

Q(t)	JK				Q
	00	01	11	10	
a	a	a	b	c	0
b	c	b	b	c	1
c	c	a	d	c	1
d	a	a	d	d	0







A=000
B=011
D=110
C=101

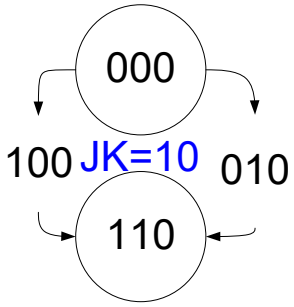
Koda stanja: $y_2y_1y_0$

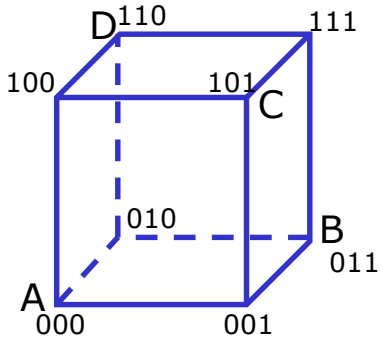
Sinteza asinhrone JK pomnilne celice

	Q(t)	JK			
		00	01	11	10
a	000	000	000	011	101
	001	101	000	011	101
b	011	101	011	011	101
	010	000	000	011	xxx
d	110	000	000	110	110
	111	101	xxx	110	101
c	101	101	000	110	101
	100	000	000	110	101

a=000
b=011
d=110
c=101

Q(t)	***JK***				***Q***
	00	***01***	***11***	***10***	
a	***a***	***a***	***b***	***c***	***0***
b	***c***	***b***	***b***	***c***	***1***
c	***c***	***a***	***d***	***c***	***1***
d	***a***	***a***	***d***	***d***	***0***





Sinteza asinhrone JK pomnilne celice

	$y_2y_1y_0$	JK			
		00	01	11	10
a	000	000	000	011	101
	001	101	000	011	101
b	011	101	011	011	101
	010	000	000	011	xxx
d	110	000	000	110	110
	111	101	xxx	110	101
c	101	101	000	110	101
	100	000	000	110	101

Y_0	$y_2y_1y_0$	JK			
		00	01	11	10
a	000	0	0	1	1
	001	1	0	1	1
b	011	1	1	1	1
	010	0	0	1	X
d	110	0	0	0	0
	111	1	X	0	1
c	101	1	0	0	1
	100	0	0	0	1

$$Y_0 = y_2 \cdot J + y'_2 \cdot y_1 \cdot y_0 + K' \cdot y_2 \cdot y_0 \cdot + y_2 \cdot y'_1 \cdot J \cdot K'$$

Sinteza asinhrone JK pomnilne celice

	$y_2y_1y_0$	JK			
		00	01	11	10
a	000	000	000	011	101
	001	101	000	011	101
b	011	101	011	011	101
	010	000	000	011	xxx
d	110	000	000	110	110
	111	101	xxx	110	101
c	101	101	000	110	101
	100	000	000	110	101

Y_1	$y_2y_1y_0$	JK			
		00	01	11	10
a	000	0	0	1	0
	001	0	0	1	0
b	011	0	1	1	0
	010	0	0	1	x
d	110	0	0	1	1
	111	0	x	1	0
c	101	0	0	1	0
	100	0	0	1	0

$$Y_1 = y'_0 \cdot y_1 \cdot J + J \cdot K + y'_2 \cdot y_1 \cdot y_0 \cdot K$$

Sinteza asinhrone JK pomnilne celice

	$y_2y_1y_0$	JK			
		00	01	11	10
a	000	000	000	011	101
	001	101	000	011	101
b	011	101	011	011	101
	010	000	000	011	xxx
d	110	000	000	110	110
	111	101	xxx	110	101
c	101	101	000	110	101
	100	000	000	110	101

Y_2	$y_2y_1y_0$	JK			
		00	01	11	10
a	000	0	0	0	1
	001	1	0	0	1
b	011	1	0	0	1
	010	0	0	0	X
d	110	0	0	1	1
	111	1	X	1	1
c	101	1	0	1	1
	100	0	0	1	1

$$Y_2 = y'_2 \cdot y_0 \cdot K' + J \cdot K' + y_2 \cdot y_0 \cdot K' + y_2 \cdot J = y_0 \cdot K' + J \cdot K' + y_2 \cdot J$$

Izhod asinhrone JK celice

	$y_2y_1y_0$	JK 00	01	11	10
a	000	0	0	1	1
	001	1	0	1	1
b	011	1	1	1	1
	010	0	0	1	X
d	110	0	0	0	0
	111	1	X	0	1
c	101	1	0	0	1
	100	0	0	0	1

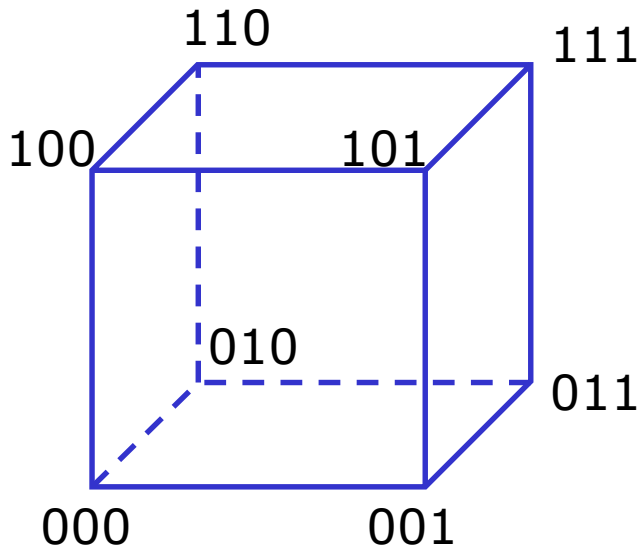
Q(t)	*JK* 00	01	11	10	*Q*
a	*a*	*a*	*b*	*c*	*0*
b	*c*	*b*	*b*	*c*	*1*
c	*c*	*a*	*d*	*c*	*1*
d	*a*	*a*	*d*	*d*	*0*

a in d zamenjamo z 0,
b in c zamenjamo z 1

$$Q=y'_2\cdot J+y'_2\cdot y_1\cdot y_0+y'_2\cdot y_0\cdot K'+y_2\cdot y_0\cdot J+y'_1\cdot J\cdot K'$$

Kodiranje stanj asinhronega avtomata

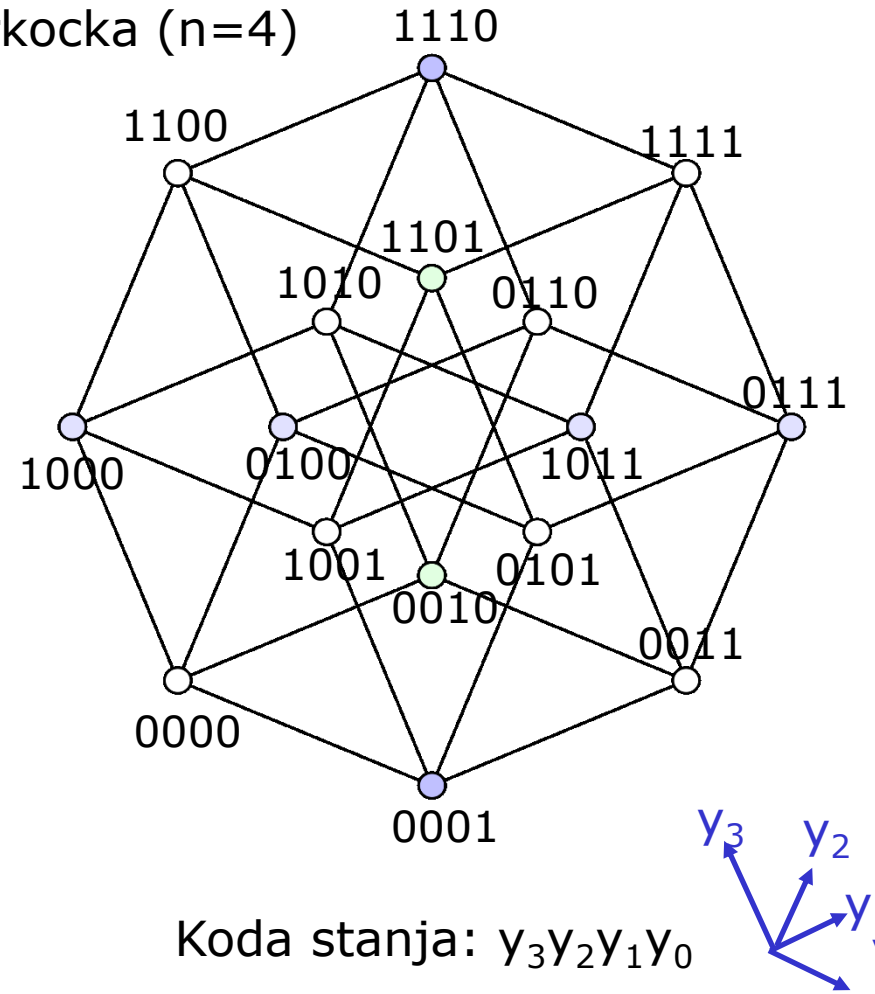
kocka (n=3)



Koda stanja: $y_2y_1y_0$

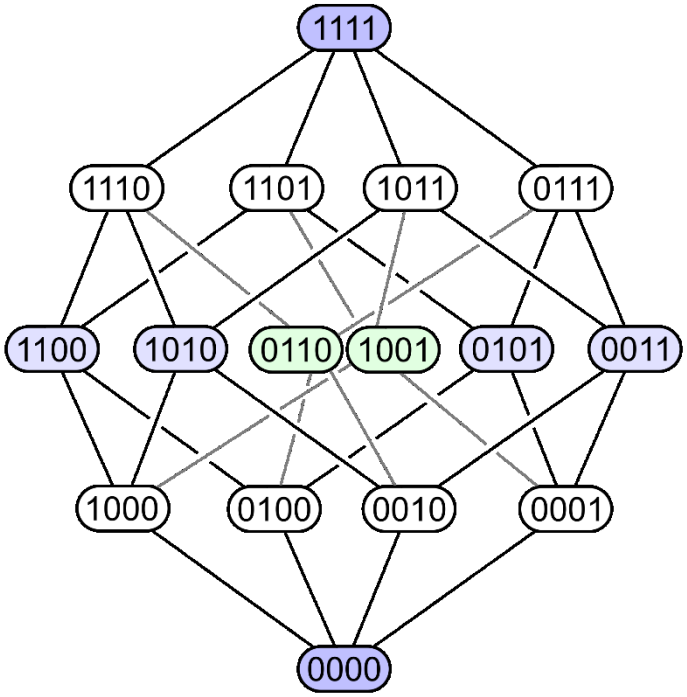
Vir: Wikipedia: Hypercube graph

hiperkocka (n=4)

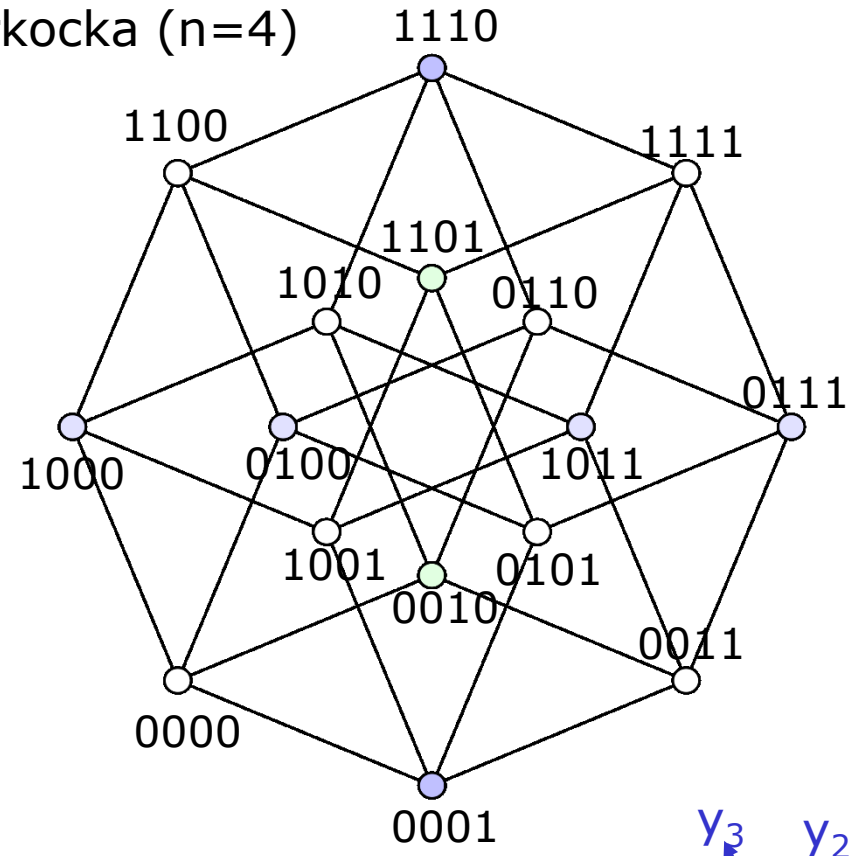


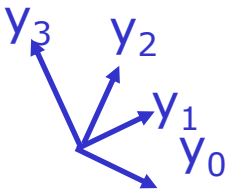
Koda stanja: $y_3y_2y_1y_0$

Kodiranje stanj asinhronega avtomata



hiperkocka (n=4)





Koda stanja: $y_3y_2y_1y_0$

Vir: Wikipedia: Hasse diagram