Unarni operatorji v VHDL

Z uporabo procesnega stavka (process) VHDL programirajte arhitekturo N-bitnih unarnih operatorjev OR, AND in XOR po podani entiteti:

```
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, reduced_AND, reduced_XOR: out STD_LOGIC);
end reduction_operators;
```

Unarni logični operatorji – včasih imenovani tudi operatorji redukcije (ang. reduction operators) se izvajajo nad vsemi elementi vhodnega vektorja, ki je tipa **STD_LOGIC_VECTOR**. V standardu VHDL-2002 so programirani z rekurzivnimi funkcijami (**or** , **and** , **xor** in njihove negacije): **rezultat <= or_reduce vektor;** (glej reduce pack; VHDL.org). V standardu VHDL-2008 imajo malo drugačna imena: **rezultat <= or vektor;**. V standardu VHDL-1993 tovrstni operatorji niso vgrajeni, zato bomo s to vajo pokazali, kako različne HDL izvedbe vplivajo na hitrost delovanja in porabo primitivov v FPGA vezjih.

Unarna OR operacija izvede OR operacijo med posameznimi biti vhodnega vektorja tipa **STD_LOGIC_VECTOR**.
Primer:

```
signal temp : STD_LOGIC_VECTOR(3 downto 0) := "1001";
signal bitni_or : STD_LOGIC; -- rezultat unarne or operacije
-- unarna or operacija - operacija or med vsemi biti vektorja bitni_or
bitni_or  <= temp(0) or temp(1) or temp(2) or temp(3)
```

Rezultat unarne OR operacije bo postal '1', če je vsaj en element vhodnega vektorja enak '1'.

Podobno lahko zapišemo za unarno AND operacijo:

```
signal temp : STD_LOGIC_VECTOR(3 downto 0) := "1001";
signal bitni_and : STD_LOGIC; -- rezultat unarne and operacije
-- unarna and operacija - operacija and med vsemi biti vektorja bitni_and
bitni_and <= temp(0) and temp(1) and temp(2) and temp(3)
```

Rezultat unarne AND operacije bo postal '0', če je vsaj en element vhodnega vektorja enak '0'.

Za XOR operacijo uporabimo lastnost združevanja (ang. associativity), ki velja tudi za AND in OR operaciji:

Za 3-vhodna XOR vrata:	a ⊕ (b ⊕ c) (2 nivoja zakasnitve)	=	(a ⊕ b) ⊕ c (2 nivoja zakasnitve)
Za 4-vhodna XOR vrata:	a ⊕ (b ⊕ (c ⊕ d)) (3 nivoji zakasnitve)	=	(a ⊕ b) ⊕ (c ⊕ d). (2 nivoja zakasnitve)
Za 5-vhodna XOR vrata:	a ⊕ (b ⊕ (c ⊕ (d ⊕ e))) (4 nivoji zakasnitve)	=	((a ⊕ b) ⊕ (c ⊕ d)) ⊕ e. (3 nivoji zakasnitve)
Za 6-vhodna XOR vrata:	a ⊕ (b ⊕ (c ⊕ (d ⊕(e ⊕ f)))) (5 nivojev zakasnitve)	=	((a ⊕ b) ⊕ ((c ⊕ d)) ⊕ (e ⊕ f)). (3 nivoji zakasnitve)

Pri levem načinu izvedbe imamo opravka z zaporedno tvorbo večbitne unarne operacije, katere rezultat je neuravnoteženo dvojiško drevo (ang. unbalanced binary tree), čigar višina *h* predstavlja število nivojev vrat, potrebnih za realizacijo (*h=n-1*). Pri desnem načinu izvedbe, z uporabo lastnosti združevanja je rezultat uravnoteženo dvojiško drevo (ang. balanced binary tree). Prednost desne izvedbe je v združevanju dvočlenikov, ki se lahko izvajajo vzporedno, in v zmanjšanju števila potrebnih nivojev na $h = \lceil \log_2(n) \rceil$, kjer je *h* višina drevesa - število stopenj zakasnitve, *n* pa število elementov vhodnega vektorja.

```
signal temp : STD_LOGIC_VECTOR(3 downto 0) := "1001";
signal bitni_xor : STD_LOGIC; -- rezultat bitne xor operacije
-- unarna xor operacija - operacija xor med vsemi biti
bitni_xor <= temp(0) xor (temp(1) xor (temp(2) xor temp(3)))
```

Najbolj enostavna izvedba unarne XOR operacije je realizacija N-bitnih XOR vrat kot zaporedja 2-vhodnih XOR operacij. Unarna operatorja OR in AND bi lahko realizirali tudi s primerjalnikom enakosti (operator =):

```
architecture vhdl_93_comb of reduction_operators is begin
      reduced_OR <= '0' when (A = (A'range => '0')) else '1';   -- unarni OR
      reduced_AND <= '1' when (A = (A'range => '1')) else '0';  -- unarni AND
end vhdl_93_comb;
```

vendar to ni izvedba s procesnim stavkom, ki bi jo radi programirali, in *ni cilj* naloge.

V nadaljevanju bomo programirali entiteto z imenom `reduction_operators`, ki ima definirane naslednje vhode/izhode:

- **N** : celoštevilski parameter (naravno število), ki označuje velikost vhodnega vektorja nad katerim izvajamo unarno operacijo
- **A**: N-bitni vhodni vektor tipa `std_logic_vector` - predstavlja polje elementov nad katerimi izvajamo unarno operacijo,
- **reduced_OR, reduced_AND, reduced_XOR**: izhodi tipa `std_logic`, ki po vrsti predstavljajo rezultate unarne operacije OR, AND, XOR.

Opisana entiteta se nahaja v datoteki (`unary_operators_vhdl93.vhd`).
Naloge:

1. Ustvarite nov projekt z imenom `Unary_VHDL93` v mapi `VHDL93`.
   Izbrano vezje projekta je odvisno od verzije ISE, ki jo uporabljate (v oknu Hierarchy→Design Properties (desni klik):
   - za ISE 10.1 nastavite Spartan 3 FPGA (Family: Spartan3, Device: xc3s200, Package: tq144, Speed grade: -5).
   - za ISE 14.7 nastavite Spartan 6 FPGA (Family: Spartan6, Device: 6slx4, Package: tqg144, Speed grade: -3).

   V arhivu predloge naloge se v mapi `VHDL93` nahaja datoteka `unary_operators_vhdl93.vhd`. Datoteko dodajte k projektu (Project→Add Source). V tej datoteki je izpisana entiteta unarnega operatorja, arhitektura (`vhdl_93_process`) je prazna. Redukcijske operatorje programirajte tako, da v arhitekturo (`vhdl_93_process`) zapišete tri procesne stavke z imeni (`or_reduction, and_reduction, xor_reduction`), s katerimi po vrsti realizirate unarne operacije OR, AND, XOR. Pri realizaciji s procesnim stavkom uporabite začasno spremenljivko (npr. `temp_or`), ki je tipa (`std_logic`). To spremenljivko uporabite v (`for in ... loop`) zanki, s katero preletite vse elemente vhodnega vektorja A (`A'range`). Če izvedba unarne operacije narekuje predčasen izhod iz zanke, to storite z ukazom (`exit when`). V primeru unarne OR operacije iščete *prvi* element vektorja A, katerega vrednost '1'. Če ga najdete, je rezultat unarne OR operacije enak '1'. Idejo programiranja unarnega OR operatorja s procesnim stavkom povzema spodnji procesni stavek:

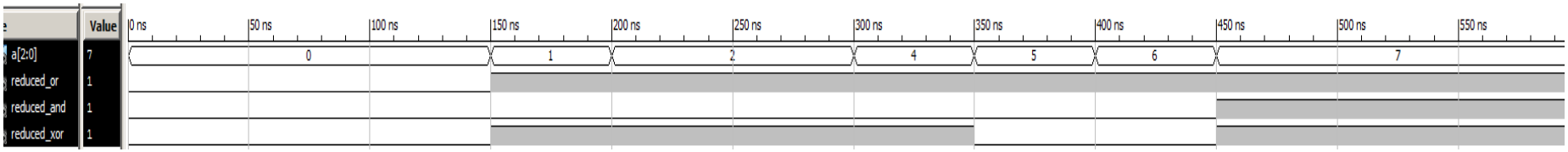
```
or_reduction: process (A)
variable temp_or: std_logic;
begin
      temp_or := '0';
      for i in A'range loop
            temp_or := temp_or or A(i);      -- ce je v A samo en element, je rezultat ta element (or 0)
            exit when temp_or = '1';
      end loop;
      reduced_OR <= temp_or;
end process;
```

2. V arhivu predloge naloge se v mapi `VHDL93` nahaja datoteka testnih vrednosti `unary_operators_vhdl93_tb.vhd`. Datoteko dodajte k projektu (Project→Add Source) in izberite način simulacije: V panelu Sources izberite simulacijo (Behavioral simulation). V datoteki `unary_operators_vhdl93_tb.vhd` programirajte zaporedje testnih vrednosti v jeziku VHDL in ne grafično z nastavljanjem vrednosti v generatorju TBW, kot ste na laboratorijskih vajah v ISE 10.1. Najprej deklarirajte komponento (`component`) enote testiranja (ang. unit under test - UUT). Nato deklarirajte splošne konstante (`constant`) z uporabo agregatov (`others`), ki bodo predstavljale N-bitne vrednosti dvojiških števil od 0_{10} do 7_{10}. Definirajte še notranje signale, ki naj imajo zaradi enostavnosti povezovanja ista imena in tip kot v UUT. S splošnim povezovalnim stavkom (`generic map, port map`) povežite interne signale datoteke testnih vrednosti in priključke enote UUT. S klicem (`generic map`) povežete N bitni parameter (velikost vhodnega vektorja A) datoteke testnih vrednosti na istoimenski N bitni parameter v datoteki UUT. Nato programirajte stimulacijski proces z imenom (`stim_proc`), ki bo zagotavljal časovno zaporedje vhodnega vektorja A po spodnji tabeli:

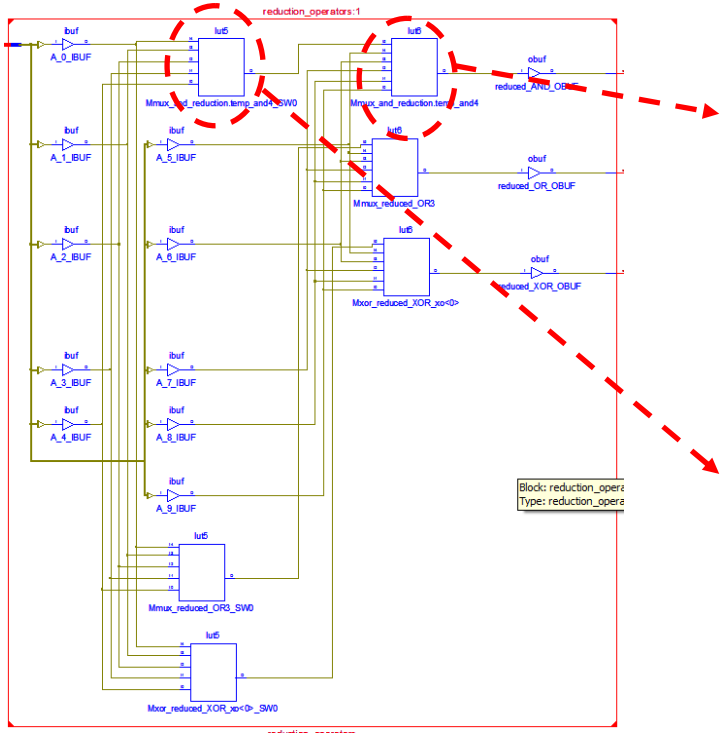
Vrednost A(2 downto 0)	0_{10}	1_{10}	2_{10}	3_{10}	4_{10}	5_{10}	6_{10}	7_{10}
Časovni interval (ns)	0 - 150 ns	150 - 200 ns	200 - 300 ns	ni v uporabi	300 - 350 ns	350 - 400 ns	400 - 450 ns	450 ns - do konca simulacije

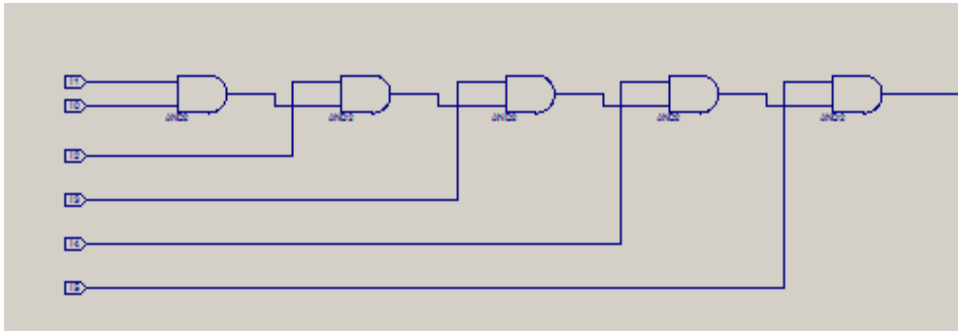
Časovno zaporedje vhodnega vektorja A nastavljate s prireditvijo ustrezni splošni konstanti izbrane vrednosti (0_{10} ... 7_{10}). Vrednosti vhodnih signalov UUT v procesnem stavku zadržite na zadnji izbrani vrednosti z uporabo stavka (`wait for 50 ns;`). Npr. stavek (`wait for 50 ns;`) zadrži vrednosti vhodnih signalov UUT na zadnji nastavljeni vrednosti za (`50 ns`).

Poženite simulacijo (ISIM Simulator→Simulate behavioral model). Če ste procesne stavke programirali pravilno, dobite spodaj prikazan rezultat simulacije unarnih operatorjev:

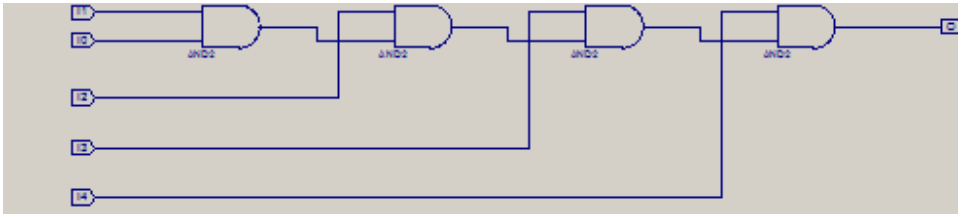


3. Oglejte si **<u>dejansko realizacijo</u>** programirane entitete unarnih operatorjev: V panelu Sources zamenjajte simulacijo (Behavioral simulation) z implementacijo vezja (Implementation) in v panelu Processes razprite "Implement Design→Synthesize XST→View Technology schematic→Rerun All (desni klik)"). Če ste stvari programirali pravilno, se realizacija 10-bitnih unarnih OR, AND, XOR operatorjev sintetizira v izvedbo na spodnji sliki:





Vsebina LUT6 pri realizaciji 10 vhodne unarne AND operacije.



Vsebina LUT5 pri realizaciji 10 vhodne unarne AND operacije.

Bistvo realizacije unarnih operatorjev se skriva v vsebini 5 in 6 vhodnih vpoglednih tabel. Na sliki je predstavljena implementacija vezja v okolju ISE 14.7, na primeru Spartan6, ki uporablja LUT5 in LUT6 vpogledne tabele. Če uporabljate ISE 10.1, je implementacija vezja rahlo drugačna, saj Spartan 3 vsebuje največ 4-vhodne vpogledne tabele (LUT4).
Če kliknete na določeno LUT tabelo, pri OR in AND izvedbi unarnega operatorja opazite 10 vhodna AND/OR vrata, ki so sestavljena iz *zaporedno* vezanih 4 vhodnih in 6 vhodnih AND/OR vrat. Na vhodih/izhodih so dodani vhodni in izhodni ojačevalniki signala (IBUF, OBUF). Največja vpogledna tabela v Spartan6 in v seriji 7 Xilinx FPGA je 6 vhodna (LUT6), zato sta za realizacijo 10 vhodnih vrat

potrebni dve vpogledni tabeli, pri Spartan3 pa tri vpogledne tabele LUT4. Pri realizaciji 10 bitnih XOR vrat je stvar podobna - tudi tu je rabimo enako število vpoglednih tabel in velja podobna logika *zaporednega* povezovanja 2 vhodnih XOR vrat, le da je realizacija povezav neočitna, saj so izvedena v DNO obliki.
Če bi npr. analizirali vsebino LUT5 tabele pri realizaciji unarnih XNOR vrat (levi klik na LUT→zavihek TruthTable), bi prebrali funkcijo v PDNO $f^5 = V(0, 3, 5, 6, 9, 10, 12, 15, 17, 18, 20, 23, 24, 27, 29, 30)$. Funkcija f^5 v LUT5 predstavlja 5 vhodna XNOR vrata. V primeru ISE10.1 bi v Spartan3 LUT4 tabeli prebrali funkcijo $f^4 = V(0, 3, 5, 6, 9, 10, 12, 15)$. Za LUT4 si lahko s klikom na izbrano tabelo ogledate tudi Karnaughov diagram. Funkcijo *f* lahko analiziramo kot kombinacijski generator sode parnosti (ang. even parity generator). Izhod teh vrat postane '1', če je število vhodov na '1' sodo - funkcija 5 oziroma 4-vhodnih XNOR vrat. Podobno analizo bomo v nadaljevanju izvedli za 6 vhodna XOR vrata.

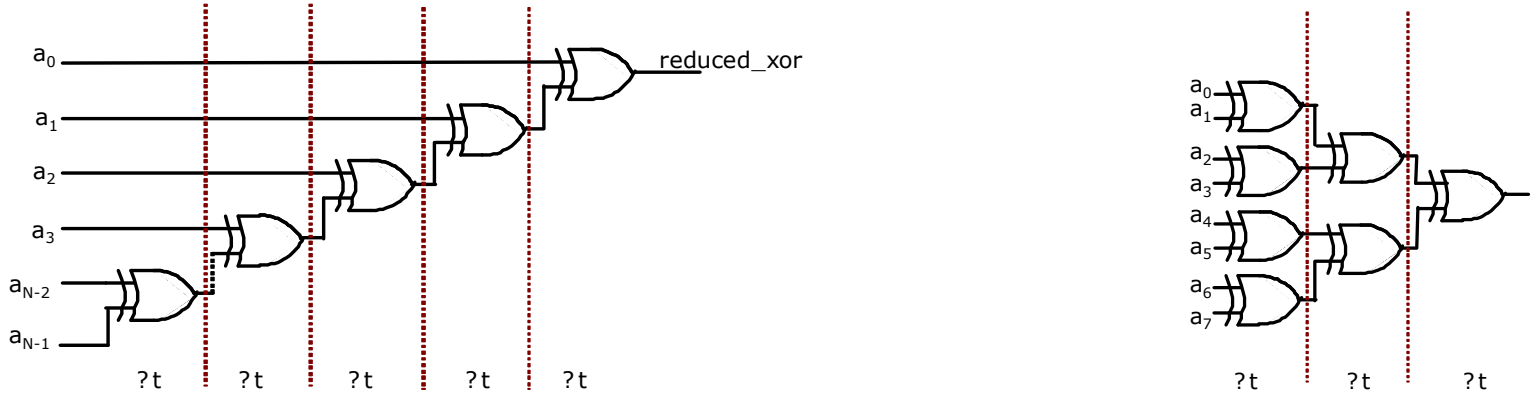
4. V implementaciji povečajte število mest vhodnega vektorja unarne operacije iz (N=10) na (N=4096 za Spartan6 in N=1024 za Spartan3), poženite sintezo (ang. Synthesize-XST→Rerun All (desni klik)) in v konzoli (zavihek Console) določite največjo zakasnitev poti kombinacijskega vezja (ang. Maximum combinational path delay). Če delate z vezjem Spartan 3, ne postavite N večji kot 1024, sicer bo sinteza trajala predolgo (za N=1024 traja sinteza ca. 1226.55 s, odvisno od hitrosti računalnika). Poročilo sintetizatorja XST v konzoli na koncu vsebuje kratek povzetek sinteze (ang. Design Summary):

```
--*********************************Za primer Spartan 3 (1024 vhodna XOR vrata)************************************
--Timing Summary:
----------------
--Speed Grade: -5
--
--   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: 163.402ns

--*********************************Za primer Spartan 6(4096 vhodna XOR vrata)************************************
--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: 1456.874ns
```

V opozorilih v konzoli ISE opazite, da ste porabili več kot 100% kapacitet vezja, kar je logično, saj ne Spartan3, ne Spartan6 nimata 4096 vhodno-izhodnih priključkov, ampak samo 144, saj smo izbrali ohišje TQ144. Največja zakasnitev izdelanega vezja na primeru Spartan6 je 1456.874 ns, pri vezju Spartan3 pa 163 ns, kar lahko razložimo z zaporedno vezavo LUT tabel. Med rezultatom Spartan3 in Spartan6 ni primerjave, saj gre v prvem primeru za 1024 vhodna XOR vrata, v drugem pa za 4096 vhodna. Pomembno je, da smo dobili izhodiščno vrednost zakasnitve, na kateri bomo ocenjevali nadaljnja izboljšanja HDL izvedbe unarnih operatorjev.
Podrobnejše poročilo o zakasnitvah sinteze vezja dobite, če odprete celotno poročilo o projektu (Meni ISE→Project→Design Summary). V oknu poročilu izberete na levem delu (meni) razdelek *poročilo o sintezi* (Detailed report→Synthesis report) in se premaknete na mesto podrobnejše analize časovnih omejitev in poti (ang. Timing constraint: Default path analysis). V časovnih podrobnostih razberete število logičnih nivojev (ang. levels of logic), ki povzročijo končno zakasnitev. Pri 4096 vhodnih XOR vratih je ob sintezi nastalo 1360 nivojev zakasnitev, pri sintezi 1024 vhodnih XOR vrat pa 107 nivojev zakasnitve. Število nivojev zakasnitev v nobenem primeru ni enako 4096 oz. 1024, saj so v uporabi LUT4 in/ali LUT6 tabele, ki *lahko* v enem nivoju zakasnitve realizirajo 4 oz. 6 vhodna XOR vrata. Nastalo število nivojev pripišemo potratni *zaporedni* vezavi vpoglednih tabel in *trivialni* realizaciji s pomočjo 2-vhodnih XOR vrat.

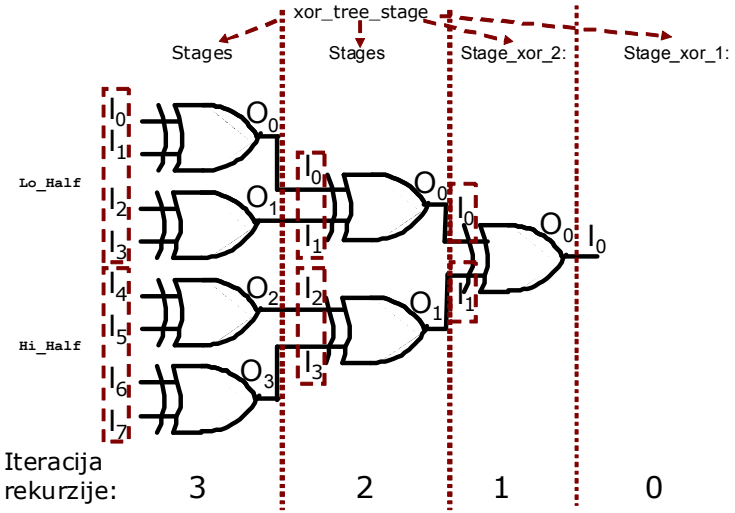
```
--*******************************Za primer Spartan 3 (1024 vhodna XOR vrata)************************************
--Timing Detail:
---------------
--All values displayed in nanoseconds (ns)
--
--=========================================================================
--Timing constraint: Default path analysis
--  Total number of paths / destination ports: 105082 / 3
-------------------------------------------------------------------------
--Delay:               163.402ns (Levels of Logic = 107)

--*******************************Za primer Spartan 6 (4096 vhodna XOR vrata)************************************
--Timing Details:
----------------
--All values displayed in nanoseconds (ns)
--
--=========================================================================
--Timing constraint: Default path analysis
--  Total number of paths / destination ports: 2214747276 / 3
-------------------------------------------------------------------------
--Delay:               1456.874ns (Levels of Logic = 1360)
--=========================================================================
```

5. Kot eno možnih pohitritev realizacije unarnih operatorjev smo uvodoma omenili realizacijo s pomočjo lastnosti združevanja Boole-ove logike, pri kateri lahko na enem nivoju izvedemo operacije več dvočlenikov naenkrat. Združevanje dvočlenikov XOR operacije izkoriščamo npr pri kombinacijskem izračunu pravilnosti prenosa podatka (npr. izračunu parnosti ali pri izračunu CRC) (glej patent US6295626: Symbol based algorithm for hardware implementation of cyclic redundancy check).



Strukturo uravnoteženega dvojiškega drevesa na desni strani slike lahko programiramo na več načinov. En izmed načinov je podoben izvedbi redukcijskih operatorjev v VHDL 2002 z rekurzivnim klicem funkcije (glej reduce pack; VHDL.org).
Podoben način je z rekurzivno redukcijo vhodnega vektorja (`I`) na zgornjo in spodnjo polovico v primeru uporabe 2-vhodnih XOR vrat, vendar brez uporabe funkcij. Na zgornji desni sliki na eni stopnji redukcije vhodni vektor (`I`) na začetku iz velikosti 8 vhodov reduciramo na 4 izhode (`O`), pri čemer med biti vektorja paroma izvajamo XOR operacijo. Izhod te stopnje rekurzivno vodimo na naslednjo stopnjo, pri kateri se štirje vhodi preko 2-vhodnih XOR reducirajo na dva izhoda. Postopek ponavljamo, dokler dolžina vhodnega vektorja ni enaka 2. Takrat preostala elementa vhodnega vektorja vodimo na zadnja XOR vrata in na izhod strukture, kot kaže desni del zgornje slike.
Posamezno iteracijo redukcije z XOR definiramo kot parametrizirano entiteto (`XorTreeStage`), ki ima N bitni vhod (`I`) in 1-bitni izhod (`O`).

```
entity XorTreeStage is
generic (N : natural);
port(  I: in std_logic_vector(N - 1 downto 0);
       O: out std_logic);
end XorTreeStage;
```



Rekurzivno redukcijo razdelimo na tri primere: Prvi primer (`Stages`) se nanaša na vse stopnje rekurzivne redukcije drevesa *razen* zadnjih dveh nivojev. Drugi primer (`Stage_Xor_2`) se nanaša samo na zadnja XOR vrata (na sliki označeno kot iteracija rekurzije 1). Tretji primer (`Stage_Xor_1`) je na sliki označena kot iteracija rekurzije 0, pri čemer vhod (`I`) pravzaprav samo vodimo na izhod (`O`), saj imamo opravka samo z enim bitom (žica). Ideja rekurzivne redukcije drevesa je, da na vsaki stopnji (`Stages`) razdelimo vhodni vektor (`I`) na zgornjo in spodnjo polovico in med sosednimi biti polovic tvorimo 2 vhodne XOR operacije. Nad izhodi rezultirajočih XOR operacij rekurzivno ponovimo 2 vhodne XOR operacije, dokler ne dosežemo dolžine vhodnega vektorja 2, ki jo lahko izračunamo kot enostaven enih samih 2-vhodnih XOR vrat. Tretji primer je dodan zaradi popolnosti, saj smo kot parameter N navedli naravno število - torej v splošnem uporabnik lahko definira tudi eno bitna XOR vrata, kar predstavlja povezavo vhoda na izhod (žica). Za N=1 se rekurzija nikdar ne bi izvedla - če bi ta primer izpustili, sintetizator XST ne bi povezal vhoda na izhod in bi vrnil ustrezno opozorilo.

Koda rekurzivne redukcije z uporabo 2-vhodnih XOR vrat je povzeta v spodnji tabeli. Rekurzijo v VHDL dosežemo s krožnim sklicem (ang. circular reference) tako, da entiteto (`XorTreeStage`) definiramo *tudi* kot komponento (`component`) modula. ISE sicer vrne opozorilo o krožnem sklicu (ang. circular reference warning), a ga ignoriramo, saj je v tem primeru raba rekurzije znana. Če rekurzivnega sklica v splošnem ne bi predvidevali, naj bo to opozorilo, da je s sklici nekaj narobe, saj se z nekontroliranimi krožnimi sklici proces sinteze hitro ujame v neskončno zanko. Privzeta največja globina rekurzivnih sklicev v XST je 64.

Vsakega od treh prej omenjenih primerov redukcije ločimo s posebnim pogojnim stavkom tvorjenja (`if...generate`). Prva stavka se nanašata na točno določeno dolžino vhodnega vektorja (`I`), zadnji pa pokriva rekurzivne klice za preostale dolžine, ki so večje od 2 (`I'length > 2`). V primeru rekurzivne redukcije drevesa (`Stages`) ob vsakem klicu vhodni vektor (`I`) razdelimo na zgornjo in spodnjo

polovico z uporabo agregatov. V ta namen definiramo signala (`Lo_Half, Hi_Half`), ki po vrsti predstavljata spodnjo in zgornjo polovico trenutnega vhoda (`I`). Če je velikost vhodnega vektorja (`I`) liho število, preostali element pripišemo zgornji polovici (`Hi_Half`). Nad nastalima polovicama rekurzivno ustvarimo komponenti (`XorTreeStage`), pri čemer se prva komponenta nanaša na spodnjo polovico, druga pa na zgornjo. Ko se vsi rekurzivni klici odvijejo (oz. ko je dolžina vhoda = 2), se povratno ovrednotijo tudi vrednosti izhodov posamezne stopnje (`O <= Lo_Xor xor Hi_Xor;`).

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

entity XorTreeStage is
	generic (N : natural);
	port(  I: in std_logic_vector(N - 1 downto 0);	-- vhodni vektor redukcije ima N vhodov
	       O: out std_logic);			-- izhodni bit redukcije
end XorTreeStage;

architecture tree_of_xor2 of XorTreeStage is

component XorTreeStage
	generic (N : natural);
	port(  I: in std_logic_vector(N - 1 downto 0);	-- vhodni vektor redukcije ima N vhodov
	       O: out std_logic);  -- izhodni bit redukcije
end component;

begin

Stage_xor_1:
	if I'length = 1 generate
		begin
		O <= I(I'left);     -- xor enega bita std_logic_vector je kar ta bit (x xor 0) = x
		-- ta stavek obenem opravlja pretvorbo std_logic_vector(0 downto 0) v tip izhoda (O), ki je std_logic
	end generate Stage_xor_1;

Stage_xor_2:
	if I'length = 2 generate
		begin
		O <= I(I'right) xor I(I'left);  -- xor dvobitnega vektorja std_logic_vector: desni bit xor levi bit
	end generate Stage_xor_2;
```

```
Stages: if I'length > 2 generate
	signal Lo_Xor, Hi_Xor: std_logic;
	signal Lo_Half : std_logic_vector(I'length/2 - 1 downto 0);
	signal Hi_Half : std_logic_vector(I'length - 1 downto I'length/2);
	begin
		-- razdelimo vhodni vektor na polovici (lo - spodnja, Hi - zgornja)
		Lo_Half <= I(I'length/2 - 1 downto 0);
		-- pri lihih vrednostih dolžine vhodnega vektorja (I) dodatni element pripišemo zg. polovici
		Hi_Half <= I(I'length - 1 downto I'length/2);

		-- povežemo polovici z rekurzivnim povezovalnim stavkom
		Lo_Tree: XorTreeStage generic map (Lo_Half'length) port map (Lo_Half, Lo_Xor);
		Hi_Tree: XorTreeStage generic map (Hi_Half'length) port map (Hi_Half, Hi_Xor);

		O <= Lo_Xor xor Hi_Xor;   -- polovici združimo z 2-vhodnimi xor vrati

	end generate Stages;

end tree_of_xor2;
```

6. Če si ponovno ogledamo povzetek časovnih zakasnitev za primer (N=4096 za Spartan6 in N=4096 za Spartan3) opazimo občutno izboljšanje zakasnitve vrat. Poleg tega se čas sinteze občutno skrajša in zdaj lahko sintezo N=4096 izvajamo tudi na primeru Spartan3.

```
--************************Za primer Spartan 3 (4096 vhodna XOR vrata) ***************************
--Timing Summary:
-----------------
--Speed Grade: -5
--
--   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: 15.034ns
--
--Timing Detail:
---------------
--All values displayed in nanoseconds (ns)
--
--=========================================================================
--Timing constraint: Default path analysis
--  Total number of paths / destination ports: 4096 / 1
-------------------------------------------------------------------------
--Delay:               15.034ns (Levels of Logic = 8)
--  Source:            I<0> (PAD)
--  Destination:       O_Xor (PAD)
--
--************************Za primer Spartan 6 (4096 vhodna XOR vrata) ***************************
--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: 11.022ns
--
--Timing Details:
```

```
-----------------
--All values displayed in nanoseconds (ns)
--
--=========================================================================
--Timing constraint: Default path analysis
--  Total number of paths / destination ports: 4096 / 1
-------------------------------------------------------------------------
--Delay:               11.022ns (Levels of Logic = 9)
```

Zakasnitev vrat se je izboljšala na 11.022 ns za 4096 bitna XOR vrata v primeru vezja Spartan6 in 15.034 ns v primeru vezja Spartan3. Število nivojev zakasnitve se je zmanjšalo na 9 pri Spartan6 in 8 v primeru Spartan3.

Oglejmo si še porabo primitivov (ang. Primitive Usage) v podrobnem poročilu o izvedbi. V poročilu o porabi primitivov pri Spartan6 opazimo, da so se pojavili bloki LUT3, LUT4, LUT5, LUT6. Tega v VHDL realizaciji nismo predvidevali, saj smo rekurzijo vedno izvajali nad polovicama vhodnega vektorja. To je posledica optimizacije, ki sledi procesu sinteze. V procesu optimizacije lahko 2-vhodna XOR vrata, na katerih vhode so vezana dvoja 2-vhodna XOR vrata nadomestimo z enimi 4-vhodnimi XOR vrati in strukturo realiziramo z eno LUT4 tabelo. Podobno logiko optimizacije (združevanja vrat v večvhodna XOR vrata lahko izvajamo dokler to dovoljujejo osnovni bloki. Največji blok v Spartan6 in seriji 7 Xilinx FPGA je LUT6, torej lahko realiziramo elementarna 6 vhodna vrata z enim primitivom. To nas navede na novo rekurzivno delitev, vendar tokrat ne samo na polovice, ampak na katerokoli delitev - največ na šestine. Podobno velja v primeru Spartan3, kjer je največji blok LUT4, kar pomeni da rekurzivno delitev izvajamo največ po četrtinah.

```
--************************Za primer Spartan 3 (4096 vhodna XOR vrata) ***************************
--Device utilization summary:
---------------------------
--
--Selected Device : 3s200tq144-5
--
-- Number of Slices:                      785  out of   1920    40%
-- Number of 4 input LUTs:               1365  out of   3840    35%
-- Number of IOs:                        4097
-- Number of bonded IOBs:                4097  out of     97   4223% (*)
--
--************************Za primer Spartan 6 (4096 vhodna XOR vrata) ***************************
--Primitive and Black Box Usage:
-------------------------------
--# BELS                             : 866
--#      LUT3                        : 1
--#      LUT4                        : 93
--#      LUT5                        : 46
--#      LUT6                        : 726
--# IO Buffers                       : 4097
--#      IBUF                        : 4096
--#      OBUF                        : 1
--
--Device utilization summary:
---------------------------
--
--Selected Device : 6slx4tqg144-3
--
```

```
--
--Slice Logic Utilization:
-- Number of Slice LUTs:                 866  out of   2400    36%
--    Number used as Logic:              866  out of   2400    36%
--
--Slice Logic Distribution:
-- Number of LUT Flip Flop pairs used:   866
--   Number with an unused Flip Flop:    866  out of    866   100%
--   Number with an unused LUT:            0  out of    866     0%
--   Number of fully used LUT-FF pairs:    0  out of    866     0%
--   Number of unique control sets:        0
```

7. Število nivojev lahko dodatno zmanjšamo, saj FPGA vezja Spartan6 in serije 7 vsebujejo LUT6, s katerimi lahko realiziramo 6-vhodno XOR funkcijo z neposrednim vpisovanjem vrednosti mintermov. Vezja družine Spartan3 vsebujejo največ LUT4. Šest vhodno XOR funkcijo lahko enostavno zapišemo v PDNO obliki s pomočjo štetja logičnih '1' v dvojiškem zapisu številke minterma: Če je število '1' v dvojiški kodi številke minterma liho, je rezultat '1', sicer '0'.
   Povedano strnimo v spodnji tabeli, ki vsebuje številko minterma (m_j), dvojiški zapis številke minterma ($bin(m_j)$), število '1' v dvojiškem zapisu minterma ($št(1)$) in rezultat 6-vhodne XOR funkcije (f).

m_j	$bin(m_j)$	$št(1)$	f	$hex(f)$	m_j	$bin(m_j)$	$št(1)$	f	$hex(f)$	m_j	$bin(m_j)$	$št(1)$	f	$hex(f)$	m_j	$bin(m_j)$	$št(1)$	f	$hex(f)$
0	000000	0	*0*		16	010000	1	*1*		32	100000	1	*1*		48	110000	2	*0*	
1	000001	1	*1*	6	17	010001	2	*0*	9	33	100001	2	*0*	9	49	110001	3	*1*	*6*
2	000010	1	*1*		18	010010	2	*0*		34	100010	2	*0*		50	110010	3	*1*	
3	000011	2	*0*		19	010011	3	*1*		35	100011	3	*1*		51	110011	4	*0*	
4	000100	1	*1*		20	010100	2	*0*		36	100100	2	*0*		52	110100	3	*1*	
5	000101	2	*0*	9	21	010101	3	*1*	6	37	100101	3	*1*	6	53	110101	4	*0*	*9*
6	000110	2	*0*		22	010110	3	*1*		38	100110	3	*1*		54	110110	4	*0*	
7	000111	3	*1*		23	010111	4	*0*		39	100111	4	*0*		55	110111	5	*1*	
8	001000	1	*1*		24	011000	2	*0*		40	101000	2	*0*		56	111000	3	*1*	
9	001001	2	*0*	9	25	011001	3	*1*	6	41	101001	3	*1*	6	57	111001	4	*0*	*9*
10	001010	2	*0*		26	011010	3	*1*		42	101010	3	*1*		58	111010	4	*0*	
11	001011	3	*1*		27	011011	4	*0*		43	101011	4	*0*		59	111011	5	*1*	
12	001100	2	*0*		28	011100	3	*1*		44	101100	3	*1*		60	111100	4	*0*	
13	001101	3	*1*	6	29	011101	4	*0*	9	45	101101	4	*0*	9	61	111101	5	*1*	*6*
14	001110	3	*1*		30	011110	4	*0*		46	101110	4	*0*		62	111110	5	*1*	
15	001111	4	*0*		31	011111	5	*1*		47	101111	5	*1*		63	111111	6	*0*	

   Rezultate 6 vhodne XOR funkcije f preberemo iz tabele od minterma 63 padajoče do minterma 0 v šestnajstiški obliki - stolpec *hex(f)*: $6996\_9669\_9669\_6996_{16}$.

V VHDL realiziramo 6-vhodno funkcijo XOR neposredno s klicem spodnjega primitiva LUT6.

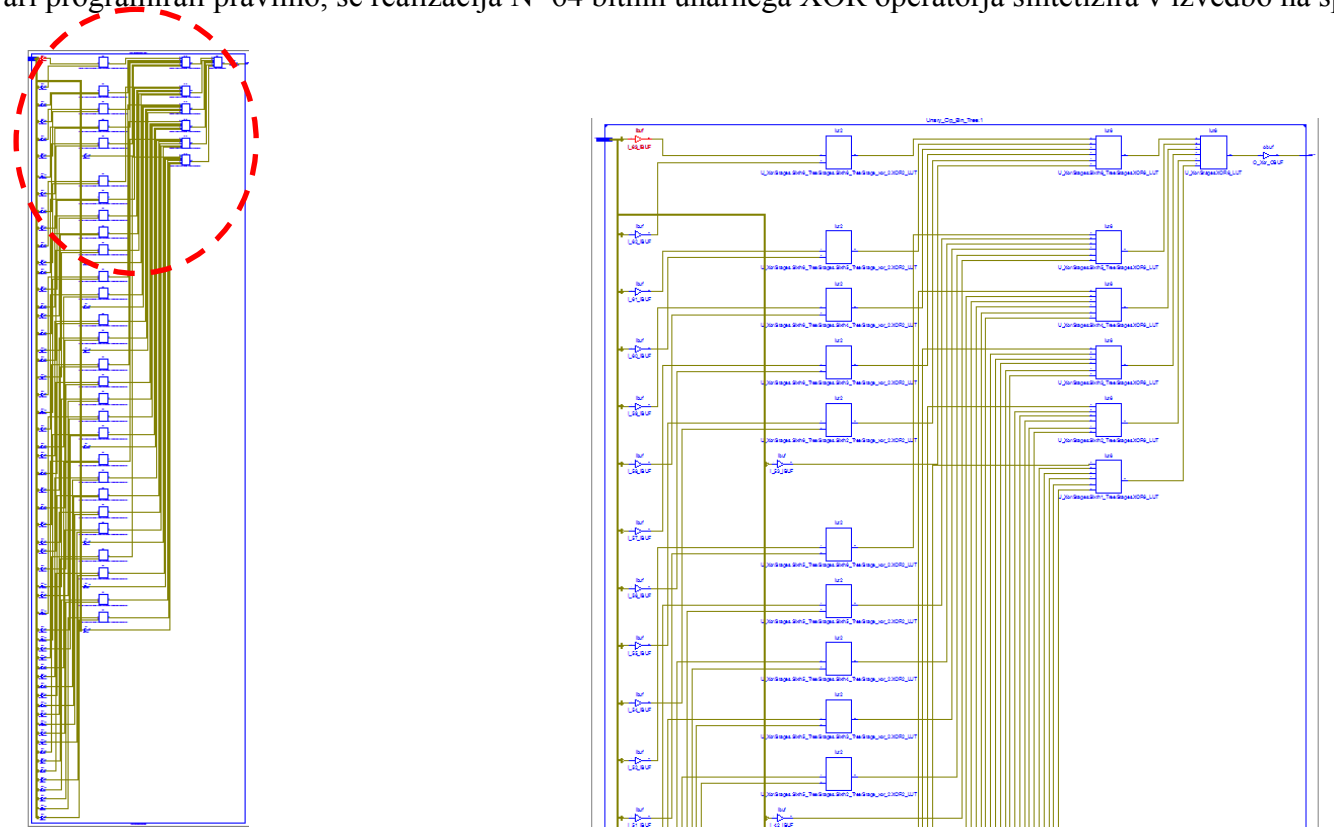
```
XOR6_LUT : LUT6
        generic map(
          INIT => X"6996_9669_9669_6996")
        port map(
          O  => O,
          I0 => in_lut6(0),
          I1 => in_lut6(1),
          I2 => in_lut6(2),
          I3 => in_lut6(3),
          I4 => in_lut6(4),
          I5 => in_lut6(5));
```

Zgornji klic realizira 6 vhodna XOR vrata z 1-bitnim izhodom (`O`) in šestimi vhodi (`in_lut6(5 downto 0)`). Za LUT4 pri vezju Spartan3 bi 4 vhodna XOR vrata realizirali z upoštevanjem spodnjih štirih bitov XOR6 tabele. Primitivi LUT4, LUT5 in LUT6 so definirani v knjižnici `library unisim;` njihovo uporabo omogočimo s stavkom `use unisim.vcomponents.all;`.

V ta namen ustvarite nov projekt z imenom `Unary_Tree` v mapi `tree`. Za vezje projekta ponovno nastavite ustrezno vezje glede na verzijo uporabljenega delovnega okolja ISE. V arhivu predloge naloge se v mapi `tree` nahajajo datoteke vrednosti `unary_op_tree.vhd`, `unary_op_xor_tree_lut6.vhd` in `unary_op_xor_tree_mux2_1.vhd`. Vse datoteke dodajte k projektu (Project→Add Source) in izberite način *izvedbe* (Panel Sources→Implementation).
Datoteka (`unary_op_tree.vhd`) je krovna datoteka in je že izpolnjena s primerom klica N-bitnih unarnega operatorja. Datoteka (`unary_op_xor_tree_mux2_1.vhd`) vsebuje zgoraj predstavljen zgled rekurzivne delitve na polovice. V datoteko (`unary_op_xor_tree_lut6.vhd`) programirajte hitrejšo rekurzijo z uporabo večbitnih vpoglednih tabel (LUT4, LUT5, LUT6) za primer vezja Spartan 6, oziroma z LUT4 za Spartan 3. Primera eno (`Stage_Xor_1`) in dvovhodnih (`Stage_Xor_2`) XOR vrat ostajata enaka kot v primeru rekurzivne delitve na polovice. Posebej ločimo še primere LUT3 (`Stage_xor_3`), LUT4 (`Stage_xor_4`). Realizacija s Spartan6 omogoča še klice LUT5 (`Stage_xor_5`) in LUT6 tabel (`Stage_xor_6`). Z vpoglednimi tabelami realiziramo XOR vrata 3, 4, 5 in 6 operandov. Za način klica LUT5 in LUT6 si oglejte (Spartan-6 Libraries Guide for HDL Designs, stran 153 in 165). Za način klica LUT4 primitiva v Spartan 3 si oglejte (Spartan 3 libraries guide for HDL designs, stran 99). Za vsako LUT tabelo zapišemo pogoj glede na ustrezno dolžino vhodnega vektorja: Za 3-bitni primer vhodnega vektorja velja (`if I'length = 3 generate`). Za vsako LUT tabelo je potrebno določiti vrednost parametra (`INIT => X"vrednost"`), pri čemer parameter INIT določimo z izbiro dela vrednosti INIT parametra LUT6 (glej `XOR6_LUT`) za ustrezno manjše število spremenljivk. Zadnji primer predstavlja rekurzivno redukcijo drevesa (`Stages`). Aktivira se vsakič, ko je dolžina vhodnega vektorja večja od 6 (`if I'length > 6`). Za vezje Spartan3 se aktivira, ko je dolžina vhodnega vektorja večja od 4 (`if I'length > 4`). V tem primeru vhodni vektor (`I`) razdelimo na šestine (četrtine - Spartan3) z uporabo agregatov. V ta

namen definiramo signale (`Sixth1_in, Sixth2_in, Sixth3_in, Sixth4_in, Sixth5_in, Sixth6_in`), ki po vrsti predstavljajo šestine trenutnega vhoda (`I`). V primeru Spartan 3 definiramo signale (`Quarter1_in, Quarter2_in, Quarter3_in, Quarter4_in`), ki po vrsti predstavljajo četrtine trenutnega vhoda (`I`). Če je velikost vhodnega vektorja (`I`) liho število, preostali element pripišemo zadnjem delu vhodnega vektorja (`I`) (`Sixth6_in/Quarter4_in`). Nad nastalimi šestinami/četrtinami s povezovalnimi stavki ustvarimo šest/štiri komponent (`XorTreeStage`), pri čemer vsaka komponenta obsega ustrezno šestino/četrtino vhodnega vektorja. Ko se vsi rekurzivni klici odvijejo, se povratno ovrednoti tudi vseh šest vrednosti izhodov posamezne stopnje v LUT6 tabeli, v kateri so programirana XOR6 vrata (glej `XOR6_LUT`). Za Spartan3 se ovrednotijo štiri vrednosti izhodov posamezne stopnje v LUT4 tabele, v kateri so programirana XOR4 vrata.

8. Oglejte si **dejansko realizacijo** programirane rekurzivne entitete unarnih operatorja XOR z uporabo LUT6 (Spartan6) oz. LUT4 (Spartan3). Če ste stvari programirali pravilno, se realizacija N=64 bitnih unarnega XOR operatorja sintetizira v izvedbo na spodnji sliki:



Levi del slike kaže tehnološko shemo povezovanja 64 bitnih XOR vrat za primer vezja Spartan6, desni del pa je povečava obkroženega dela na levi sliki. Na desnem delu lahko opazite prvi in drugi nivo redukcije z LUT6. Nato se dolžine vhodnih vektorjev hitro zmanjšajo na 1 oz. 2, zato so na tretjem nivoju LUT2, s katerimi so realizirana 2-vhodna XOR vrata.

Če si ponovno ogledate povzetek časovnih zakasnitev za primer N=4096 opazimo malenkostno izboljšanje.

```
--*************************Za primer Spartan 3 (4096 vhodna XOR vrata) ***************************

--Timing Summary:
-----------------
--Speed Grade: -5
--
--   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: 15.034ns
--
--Timing Detail:
---------------
--All values displayed in nanoseconds (ns)
--
--=========================================================================
--Timing constraint: Default path analysis
--  Total number of paths / destination ports: 4096 / 1
-------------------------------------------------------------------------
--Delay:               15.034ns (Levels of Logic = 8)
--  Source:            I<0> (PAD)
--  Destination:       O_Xor (PAD)
--
--  Data Path: I<0> to O_Xor
--                                Gate     Net
--    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
--    ----------------------------------------  ------------
--     IBUF:I->O             1   0.715   0.976  I_0_IBUF (I_0_IBUF)
--     LUT4:I0->O            1   0.479   0.976
U_Xor/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Stage_xor
_4.XOR4_LUT (U_Xor/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Quarter1_out)
--     LUT4:I0->O            1   0.479   0.976
U_Xor/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Stages.XOR4_LUT
(U_Xor/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Quarter1_out_XDM0001)
--     LUT4:I0->O            1   0.479   0.976
U_Xor/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Stages.XOR4_LUT
```

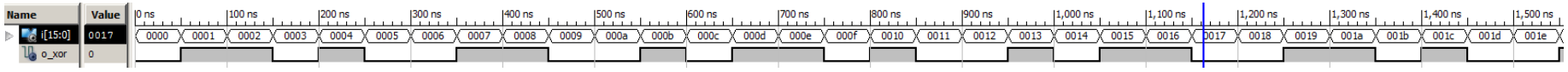
```
(U_Xor/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Quarter1_out)
--     LUT4:I0->O            1   0.479   0.976  U_Xor/Stages.Quarter1_Tree/Stages.Quarter1_Tree/Stages.XOR4_LUT
(U_Xor/Stages.Quarter1_Tree/Quarter1_out)
--     LUT4:I0->O            1   0.479   0.976  U_Xor/Stages.Quarter1_Tree/Stages.XOR4_LUT (U_Xor/Quarter1_out)
--     LUT4:I0->O            1   0.479   0.681  U_Xor/Stages.XOR4_LUT (O_Xor_OBUF)
--     OBUF:I->O                 4.909          O_Xor_OBUF (O_Xor)
--    ----------------------------------------
--    Total                     15.034ns (8.498ns logic, 6.536ns route)
--                                       (56.5% logic, 43.5% route)

--************************Za primer Spartan 6 (4096 vhodna XOR vrata) ***************************
--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: 10.106ns
--
--Timing Details:
-----------------
--All values displayed in nanoseconds (ns)
--
--=========================================================================
--Timing constraint: Default path analysis
--  Total number of paths / destination ports: 4096 / 1
-------------------------------------------------------------------------
--Delay:               10.106ns (Levels of Logic = 7)
```

Zakasnitev vrat se je izboljšala na 10.106 ns za 4096 bitna XOR vrata - število nivojev zakasnitve se je zmanjšalo na 7. Poraba primitivov (ang. Primitive Usage) pa je skoraj **dvakrat** večja:

```
--************************Za primer Spartan 3 (4096 vhodna XOR vrata) ***************************
--Device utilization summary:
-----------------------------
--Selected Device : 3s200tq144-5
-- Number of Slices:                      785  out of   1920    40%
-- Number of 4 input LUTs:               1365  out of   3840    35%
-- Number of IOs:                        4097
-- Number of bonded IOBs:                4097  out of     97   4223% (*)
--
--************************Za primer Spartan 6 (4096 vhodna XOR vrata) ***************************
--Primitive and Black Box Usage:
-------------------------------
--# BELS                              : 1556
--#      GND                           : 1
--#      LUT6                          : 1555
--# IO Buffers                        : 4097
--#      IBUF                          : 4096
--#      OBUF                          : 1
--
--Device utilization summary:
----------------------------
--Selected Device : 6slx4tqg144-3
--
--Slice Logic Utilization:
-- Number of Slice LUTs:                1555  out of   2400    64%
--    Number used as Logic:             1555  out of   2400    64%
--
--Slice Logic Distribution:
-- Number of LUT Flip Flop pairs used:  1555
--   Number with an unused Flip Flop:   1555  out of   1555   100%
--   Number with an unused LUT:            0  out of   1555     0%
--   Number of fully used LUT-FF pairs:    0  out of   1555     0%
--   Number of unique control sets:        0
```

Zaradi popolnosti omenimo, da je v Xilinx vezjih serije 7 možno rekurzije še dodatno nadgraditi, saj lahko v enem CLB združimo dve LUT6 preko enega izbiralnika 2/1 F7BMUX, (glej 7 Series FPGAs Configurable Logic Block, Xilinx, stran 41, slika 2-22) s čimer opisani primer razširimo na redukcije z 7-vhodnima XOR vrati v enem CLB. Z dvema LUT6 v tem neposredno programiramo XOR6 in XNOR6 vrata (podobno kot je prikazano v XOR6_LUT), nato pa izhod XOR6 vrat vodimo na podatkovni vhod 0, izhod XNOR6 pa na podatkovni vhod 1 izbiralnika MUX2/1 (na sliki 2-22 označen kot F7AMUX).

9. V arhivu predloge naloge se v mapi `tree` nahaja datoteka testnih vrednosti `unary_op_tree_tb.vhd`. Datoteko dodajte k projektu (Project→Add Source) in izberite način simulacije. V datoteki `unary_op_tree_tb.vhd` programirajte zaporedje testnih vrednosti v jeziku VHDL in ne grafično z nastavljanjem vrednosti v generatorju TBW, kot ste na laboratorijskih vajah v ISE 10.1. Komponenta (`component Unary_Op_Bin_Tree`) enote testiranja je že deklarirana. Nato deklarirajte splošno konstanto ZERO, podobno kot v primeru prve datoteke testnih vrednosti.
   S splošnim povezovalnim stavkom (`generic map, port map`) povežite interne signale datoteke testnih vrednosti in priključke enote UUT. S klicem (`generic map`) povežete N bitni parameter (velikost vhodnega vektorja `I`) datoteke testnih vrednosti na istoimenski N bitni parameter v datoteki UUT. Nato programirajte stimulacijski proces z imenom (`stim_proc`), ki bo zagotavljal časovno zaporedje vhodnega vektorja (`I`) za *vse* vhodne kombinacije. V našem primeru v datoteki testnih vrednosti nastavimo N=16, kar pomeni 2^{16} kombinacij vhodov vrat. Tolikšnega števila kombinacij vhodnega vektorja ne bomo nastavljali ročno kot v prvem primeru, ampak v procesnem stavku uporabimo zanko (`for in ... loop`), v kateri definiramo indeks (`idx`), ki bo tekel v območju (`0 to 2**N - 1`). Vrednost vhodnega vektorja (`I`) nastavljamo s pomočjo pretvorbe celoštevilskega indeksa (`idx`) v obliko (`std_logic_vector`) z ukazom: `std_logic_vector(to_unsigned(idx, I'length))`. Za pretvorbo dodamo še zakasnilni stavek (`wait for 50 ns;`). Za uporabo funkcije (`to_unsigned`) je v datoteki že vključena knjižnica (`use ieee.numeric_std.all;`).
   Prikazovanje 2^{16} kombinacij z zakasnitvijo 50 ns zahteva veliko daljši čas simulacije, kot je privzeta vrednost ISE (1000 ns). Čas simulacije podaljšate tako, da izberete (ISIM Simulator→Simulate behavioral model (desni klik). Odpre se okno lastnosti ISIM procesa, kjer nastavimo lastnost trajanja simulacije (ang. Simulation Run Time) na daljšo vrednost, pri čemer lahko uporabljamo VHDL časovne enote (ns, us, ms). Pozor - podaljševanje časa poteka simulacije nesorazmerno podaljša dejansko izvajanje simulacije!
   Poženite simulacijo (ISIM Simulator→Simulate behavioral model (desni klik)→Rerun All). Če ste procesne stavke in datoteko testnih vrednosti programirali pravilno, dobite spodnji rezultat simulacije 16 vhodnega unarnega XOR operatorja (prikazan je izsek prvih 30 vrednosti):



Na strežnik na domači strani predmeta naložite *vhd datoteke, ki ste jih programirali, pod kategorijo OSTALO_1. Ostalih datotek ne nalagajte!

Držite se poimenovanja v navodilih. Upoštevajte točno navedbo signalov v podanih entitetah, sicer naloge ne morem popraviti.

Upoštevajte opisano delovanje, ki ustreza opisanim logičnim vrednostim signala (glej opise signalov v entiteti in navodilih).

Pri poimenovanju signalov se držite pravila, da črka "n" pred imenom signala pomeni negativno logiko poimenovanega signala (primer: nCLR je signal, ki je aktiven '0').

Za n-bitne strukture teče indeks elementov tipa std_logic_vector vedno od n-1 (MSB mesto) downto 0 (LSB mesto).

Če naloga zahteva uporabo že izdelanih datotek, zaradi skladnosti uporabljajte podane predloge in ne lastnih.