

Ugradbeni računlni sustavi

XILINX – XST – VHDL

Dr.sc. Hrvoje Mlinarić

XST – Xilinx Synthesis Tools

- Osnovna problematika svih alata za sintezu sklopova na temelju jezika za digitalni dizajn sklopova (eng. Hardware Design Languages) je da definicija jezika omogućuje veliku funkcionalnost, dok alat za sintezu ima ograničenja programskom optimizacijom koju može napraviti i samim sklopovljem za koje radi sintezu.
- Kako je XST namijenjen isključivo za sintezu XILINX FPGA sklopova, stoga je i njegova sinteza ograničena mogućnostima takvih sklopova.
- Navedena ograničenja vrlo su bitna i treba ih imati na umu prilikom pisanja VHDL koda za opis sklopa.

XST – Xilinx Synthesis Tools

- VHDL (eng. VHSIC Hardware Definition Language)
 - VHSIC – organizacija osnovana 1980 (Very High Speed Integrated Circuits)
- VHDL je jezik za opis sklopova koji omogućava opis kako složenih tako i jednostavnih sklopova. VHDL je definiran da zadovolji mnoge zahtjeve koji se postavljaju tijekom dizajniranja sklopa:
 - omogućuje opis strukture sklopa, dijeljenjem u jednostavnije podsustave.
 - omogućava opisivanje ponašanja sustava i podsustava korištenjem jezičnih oblika sličnih programskim jezicima.
 - omogućava testiranje sustava prije nego što on bude izveden u stvarnom sklopovlju, što nam omogućava da testiramo i prilagođavamo sklop prije nego što se napravi prototip.
- U nastavku su opisane neke funkcionalnosti XST alata za sintezu sklopova temeljem opisa u VHDL jeziku.

Tipovi podataka

- *BIT* ('0', '1')
- *BOOLEAN* (false, true)
- *REAL*
- *STD_LOGIC* ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-')
 - 'U' – nedefinirano
 - 'X' – nepoznato
 - '0' – niska razina
 - '1' – visoka razina
 - 'Z' – stanje visoke impedancije
 - 'W' – slab nepoznati signal
 - 'L' – slabi niski signal
 - 'H' – slabi visoki signal
 - '-' – signal nema utjecaja
 - XST tretira jednako sljedeće signale:
 - '0' i 'L'
 - '1' i 'H'
 - '-' i 'X'
 - Vrijednosti 'U' i 'W' XST ne može koristiti.

Tipovi podataka

- Vektorski tipovi
 - *BIT_VECTOR*
 - *STD_LOGIC_VECTOR*
- Cjelobrojni tip
 - *INTEGER*
- Predefinirani VHDL tipovi
 - *BIT*
 - *BOOLEAN*
 - *BIT_VECTOR*
 - *INTEGER*
 - *REAL*

Tipovi podataka

- *STD_LOGIC* i *STD_LOGIC_VECTOR* definirani su u *STD_LOGIC_1164* IEEE paketu koji je sastavni dio IEEE biblioteke. Za uključivanje tih tipova podataka u projekt potrebno je dodati sljedeće dvije linije koda:

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;
```

Više dimenzionalna Polja

- Najviše tri dimenzije
- Polja mogu biti signali, konstante ili varijable
- Moguće je pridružiti aritmetičke operacije poljima
- Vršiti prijenos polja u funkciju i proceduru
- Sve dimenzije polja moraju u potpunosti biti navedene.

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);  
type TAB12 is array (11 downto 0) of WORD8;  
type TAB13 is array (2 downto 0) of TAB12;
```

- Moguće je definirati polje i na sljedeći način:

```
subtype TAB13 is array (2 downto 0, 22 downto 0) of  
STD_LOGIC_VECTOR (7 downto 0);
```

Više dimenzionalna Polja

- Slijede primjeri korištenja više dimenzionalnih polja:

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0),  
type TAB05 is array (4 downto 0) of WORD8  
type TAB03 is array (2 downto 0) of TAB05;  
signal WORD_A : WORD8;  
signal TAB_A, TAB_B : TAB05;  
signal TAB_C, TAB_D : TAB03;
```

```
constant CST_A : TAB03 := {  
  («00000000», «00000001», «00000010», «00000011»)  
  («00000100», «00000101», «00000110», «00000111»)  
  («00001000», «00001001», «00001010», «00001011»)  
  («10001000», «10001001», «10001010», «10001011»)};
```


Više dimenzionalna Polja

- Rad sa poljima je u potpunosti podržan, stoga je pridruživanje vrlo jednostavno:

```
TAB_A <= TAB_B;  
TAB_C <= CST_A;
```

- Moguće je izvoditi indeksiranje elemenata u polju.

```
TAB_A(5) <= WORD_A;  
TAB_C(1) <= TAB_A;
```

- Podržana je i mogućnost podindeksiranja elementa u polju.

```
TAB_A(3)(0) <= '0';  
TAB_C(2)(4)(2) <= '1';
```

Više dimenzionalna Polja

- Moguće je korištenje dijela polja

```
TAB_A(4 downto 1) <= TAB_B (3 downto 0);
```

```
TAB_C(2)(5)(3 downto 0) <= TAB_B (3)(4 downto 1);
```

Strukture

- Moguće je koristiti strukture tipova podataka, a unutar strukture može se nalaziti druga struktura, konstanta ili atributi.

```
Type REC1 is record
```

```
    Field1 : std_logic
```

```
    Field2 : std_logic_vector (3 downto 0)
```

```
End record;
```

Početne vrijednosti

- Moguće je definirati početno stanje.
- Vrijednost kojom se vrši inicijalizacija mora biti:
 - konstantna
 - ne može ovisiti o prijašnjoj inicijalizaciji
 - ne može biti funkcija ili procedura
- Ako postavimo inicijalnu vrijednost sklopovlje postavlja tu vrijednost na izlaz prilikom globalnog *reset*-a ili paljenja uređaja (eng. Power on reset). Početna vrijednost se postavlja inicijalno za komponentu i neovisna je o lokalnom *reset*-u.

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

```
signal arb_priority : std_logic_vector (3 downto 0)  
:= '1011';
```

Lokalni reset

- Koristi se tako da napišemo proces koji će napraviti inicijalizaciju sklopa.
- Ovako napisana procedura u XST alatu prevest će se kako sklop s dva stanja čiji izlaz se može kontrolirati s *RST* signalom.
- Također treba napomenuti da *globalni reset* nije isto što i *lokalni reset*.

```
Process (ckl,rst)
begin
  If rst='1' then
    Arb_onebit <='0';
  End if;
End process;
```

```
entity Top is
  Port(
    clk, rst : in std_logic;
    a_in : in std_logic;
    dout : out std_logic);
end Top;

architecture Behavioral of Top is
  signal arb_onebit : std_logic := '1';
begin
  process (clk, rst)
  begin
    if rst='1' then
      arb_onebit <= '0';
    elsif (clk'event and clk='1') then
      arb_onebit <= a_in;
    end if;
  end process;

  dout <= arb_onebit;
end Behavioral;
```

Objekti

- **Signal** se definira u dijelu za definiciju arhitekture i moguće ih je koristiti bilo gdje u arhitekturi. Vrijednost se pojedinom signalu pridružuje sa znakom “<=”

```
signal sig1 : std_logic;  
sig1 <= '1';
```

- **Varijable** se definiraju unutar procesa ili podprograma i mogu se koristiti samo unutar navedenog procesa ili potprograma. Pridruživanje vrijednosti varijabli izvodi se znakom “:=”

```
variable var1 : std_logic_vector (7 downto 0);  
var1 := "00110010"
```

- **Konstante** se mogu deklarirati u bilo kojem deklaracijskom području i mogu se koristiti unutar područja u kojem su definirane. Vrijednost konstanti ne može se mijenjati.

```
signal sig1 : std_logic_vector (5 downto 0);  
constant init0 : std_logic_vector (5 downto 0) :=  
"110011";  
sig1 <= init0;
```

Entiteti i opis arhitekture

- Opis sklopa u VHDL-u sastoji se od dva dijela: sučelja (entiteta) i tijela opisa arhitekture.
- Sučelje se opisuje u prvom dijelu i naziva se entitet (eng. entity).
- Tijelo opisuje arhitekturu sklopa i kako se on ponaša (eng. architecture).
- Unutar opisa entiteta opisujemo ulazno izlazne signale. Svaki ulaz/izlaz ima svoje ime i način rada (ulazni, izlazni, dvosmjerni, registarski) te tip podatka.
- Broj ulaza/izlaza mora biti konstantan.
- Tip ulaza/izlaza može imati samo jednu dimenziju.
- Unutar opisa arhitekture nalazi se model koji opisuje ponašanje sklopa.
- Arhitektura se sastoji od deklaracije internih signala, funkcija i procedura, te sekvencijalne i funkcijske logike.

Entiteti i opis arhitecture

```
Library IEEE;  
use IEEE.std_logic_1164.all;
```

```
entity EXAMPLE is  
  port (  
    A,B,C : in std_logic;  
    D,E : out std_logic  
  );  
end EXAMPLE;
```

```
architecture ARCHI of EXAMPLE is  
  signal T : std_logic;  
begin  
  ...  
end ARCHI;
```

Referenciranje komponenti

- Strukturni opis sklopa sastoji se od nekoliko komponenti povezanih u hijerarhijsku strukturu koja čini opis sklopa.
- Osnovna jedinica strukturalne arhitekture sklopa je komponenta, koja je preko svojih ulaza/izlaza unutar arhitekture signalima povezana s drugim komponentama.
- Signali predstavljaju žice između komponenata.
- U VHDL-u komponente su opisane entitetom i arhitekturom.
- Entitet opisuje kako komponenta izgleda prema van, odnosno kako ju druge komponenti vide
- Arhitektura predstavlja unutarnji dizajn same komponente i opisuje njeno ponašanje.
- Povezivanje komponenti ostvareno je signalima.
- Povezivanje može biti direktno ili se može ostvariti korištenjem dodatne logike za povezivanje.
- Samo referenciranje komponente sastoji se od pridruživanja ulaza/izlaza komponente objektu unutar arhitekture koja ga koristi.

```

entity HALFADDER is port (
    X,Y : in BIT;
    C,S : out BIT) ;
end HALFADDER;

architecture ARC of HALFADDER is
    component NAND2 port (
        A,B : in BIT;
        Y: out BIT) ;
    end component;

```

```

    for all : NAND use entity NAND(ARC)

```

```

        signal S1, S2, S3 : BIT;

```

```

    begin

```

```

        NANDA : NAND port map (X,Y,S3);

```

```

        NANDB : NAND port map (X,S3,S1);

```

```

        NANDC : NAND port map (S3,Y,S2);

```

```

        NANDD : NAND port map (S1,S2,S);

```

```

        C <= S3 ;

```

```

    end ARCH1;

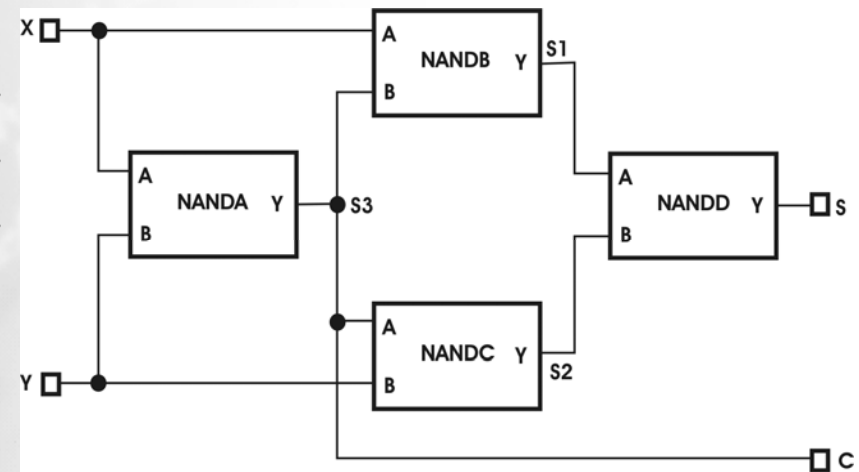
```

```

entity NAND is port (
    A,B : in BIT;
    Y    : out BIT
) ;
end NAND;

architecture ARC of NAND is
begin
    Y <= A nand B;
end ARC;

```



Konfiguracija komponenti

- Pridruživanje odgovarajućeg entiteta i arhitekture određenoj komponenti definira komponentu. XST podržava konfiguraciju komponenata u deklaracijskom području arhitekture.

```
for lista : ime komponente use entity  
Biblioteka.ime-entiteta(ime_arhitekture);
```

- U primjeru polu-zbrajala prikazano je kako referencirati komponentu. Ukoliko sve komponente koriste istu arhitekturu određenog entiteta moguće je koristiti naredbu “**for all**”

```
for all : NAND use entity work.NAND(ARCHI);
```

- Gornja definicija referencira sve komponente tipa NAND da koriste entitet NAND i arhitekturu ARCHI.
- Kada konfiguracijski parametar referenciranja entiteta i arhitekture nedostaje, XST automatski referencira entitet s istim imenom i istim sučeljem, te odabire zadnje prevedenu arhitekturu. Ukoliko nije u mogućnosti naći odgovarajući entitet i arhitekturu, tijekom sinteze sklopa generira se prazan blok koji ništa ne radi.

Generičke vrijednosti

- Generičke vrijednosti definirane su u deklaracijskom dijelu entiteta.
- XST podržava sve oblike generičkih parametara tako da oni mogu biti cjelobrojni, logičke vrijednosti, tekstovi, realne vrijednosti, `std_logic` i ostali tipovi podataka.
- Jedna od najčešćih primjena generičkih parametara je zadavanje širine sabirnice.
- Korištenjem generičkih parametara može se postići da se isti dizajn komponente može koristiti za više primjena.

Generičke vrijednosti

```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all

entity addern is generic (width : integer := 8);
  port (
    A,B : in std_logic_vector (width-1 downto 0);
    Y : out std_logic_vector (width-1 downto 0)
  );
end addern;

architecture bhv of addern is
begin
  Y <= A + B;
end bhv;
```

```

Library IEEE;
use IEEE.std_logic_1164.all;

entity top is port (
    X, Y, Z : in std_logic_vector (12 downto 0);
    A, B : in std__logic_vector (4 downto 0) ;
    S : out std_logic_vector (16 downto 0)) ;
end top;

architecture bhv of top is
    component addern generic (width : integer := 8);
        port ( A,B : in std_logic_vector (width-1 downto 0);
                Y : out std_logic_vector (width-1 downto 0));
    end component;

    for all : addern use entity work.addern(bhv);

    signal C1 : std_logic_vector (12 downto 0) ;
    signal C2, C3 : std_logic_vector (16 downto 0);

begin
    U1 :    addern generic map (n=>13), port map (X,Y,C1);
           C2 <= C1 & A;
           C3 <= Z & B;
    U2 :    addern generic map (n=>17), port map (C2,C3,S);
end bhv;

```

Kombinacijska logika

- Kombinacijskom logikom opisuju se jednoznačna pridruživanja, koja moraju biti definirana unutar tijela arhitekture.
- VHDL opisuje tri osnovna tipa kombinacijske logike:
 - jednostavnu
 - odabirnu
 - uvjetnu.
- Veličina kombinirane logike nije ograničena.
- Redoslijed kombinacijske logike nije bitan.
- Opis kombinacijske logike se sastoji od dva dijela, lijeve strane koja predstavlja rezultat i desna strana koja predstavlja logiku arhitekture.
- Jednostavno pridruživanje signala:

`T <= A and B;`

Pridruživanje signala odabirom (multiplesor)

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

entity select_bhv is generic (width: integer := 8) ;
  port (
    a, b, c, d : in std_logic_vector (width-1 downto 0);
    selector : in std_logic_vector (1 downto 0);
    T : out std_logic_vector (width-1 downto 0) ) ;
end select_bhv;

architecture bhv of select__bhv is
  begin
    with selector select
      T <= a when "00",
          b when "01",
          c when "10",
          d when others;
  end bhv;
```

Uvjetno pridruživanje (multipleksor)

```
entity when_ent is
  generic (width: integer := 8);
  port (
    a, b, c, d : in std_logic_vector (width-1 downto 0);
    selector : in std_logic_vector (1 downto 0);
    T : out std_logic_vector (width-1 downto 0)
  );
end when_ent ;

architecture bhv of when_ent is begin

  T <= a when selector = "00" else
    b when selector = "01" else
    c when selector = "10" else
    d;

end bhv;
```

Generirane strukture

- Strukture koje se višestruko ponavljaju moguće je automatski generirati korištenjem “*generate*” direktive:

```
for i in 1 to N generate;
```

- Tako navedena forma znači da se struktura ponavlja N puta.
- Sljedeći primjer pokazuje generiranje 8 bitnog zbrajala korištenjem segmenta polu-zbrajala.

```
entity EXAMPLE is
```

```
  port (
```

```
    A,B : in BIT_VECTOR (0 to 7);
```

```
    CIN : in BIT;
```

```
    SUM : out BIT_VECTOR (0 to 7);
```

```
    COUT: out BIT
```

```
  );
```

```
end EXAMPLE;
```

```
architecture ARCH1 of EXAMPLE is
```

```
  signal C : BIT_VECTOR (0 to 8);
```

```
begin
```

```
  C(0) <= CIN;
```

```
  COUT <= C(8);
```

```
  LOOP_ADD: for I in 0 to 7 generate
```

```
    SUM (I) <= A(I) xor B(I) xor C(I);
```

```
    C(I+1) <= (A(I) and B(I)) or (A(I) and C(I) or  
              (B(I) and C(I)));
```

```
  end generate;
```

```
end ARCH1;
```

Generirane strukture

- Naredba “*If condition generate*” može se koristiti za statičku definiciju komponenti arhitektura.
- Navedenu direktivu nije moguće koristiti dinamički unutar dizajna, što znači da nije moguće mijenjati broj i način definiranja komponenti tijekom rada.
- Sljedeći primjer predstavlja arhitekturu N-bitnog zbrajala koji podržava širinu od 4 do 32 bita.

```

entity EXAMPLE is generic (N : INTEGER : = 8 ) ;
    port(
        A,B : in BIT_VECTOR (N downto 0);
        CIN : in BIT;
        SUM : out BIT_VECTOR (N downto 0);
        COUT : out BIT
    ) ;
end EXAMPLE;

architecture ARCHI of EXAMPLE is
    signal C : BIT_VECTOR (N+1 downto 0);
begin
    L1: if (N>=4 and N<=32) generate
        C(0) <= CIN;
        COUT <= C(N+1);
        LOOP_ADD : for I in 0 to N generate
            SUM(I) <= A(I) xor B(I) xor C(I);
            C(I + 1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I)
                        and C(I));
        end generate;
    end generate;
end ARCHI;

```

Procesna logika

- Korištenje procesne logike razlikuje se od kombinacijske logike po tome što se pridruživanje radi određenim redoslijedom.
- Zadnje pridružena vrijednost unutar procesne logike poništava prethodnu pridruženu vrijednost.

```
entity EXAMPLE is port (  
    A, B : in BIT;  
    S : out BIT) ;  
end EXAMPLE;
```

```
architecture ARCHI of EXAMPLE is  
begin  
    process (A, B)  
    begin  
        S <= '0' ;  
        if ((A and B) = '1') then  
            S <= '1' ;  
        end if;  
    end process;  
end ARCHI;
```

Procesna logika

- Procesna logika može biti kombinacijska ili slijedna.
- Ukoliko sklopovska izvedba ne zahtijeva upotrebu memorijskih elemenata, odnosno kada su svi signali korišteni u procesu jednoznačno definirani tada je proces kombinacijski.
- Svaki proces ima listu događaja koja slijedi nakon direktive “*process*”. Proces se aktivira u koliko se jedan od elemenata liste događaja promijeni. Lista događaja mora sadržavati sve signale koji se pojavljuju u uvjetima (if, case, i sličnim...) i sve signale koji se nalaze s desne strane pridruživanja.
- Ukoliko jedan ili više signala nedostaje u listi događaja XST ga automatski dodaje. Treba posebno pripaziti na tu činjenicu jer tako generirana logika može imati različitu funkciju od prvobitno napisane.


```
library ASYL;
use ASYL.ARITH.all;

entity ADDSUB is
  port (
    A,B : in BIT_VECTOR (3 downto 0);
    ADD_SUB : in BIT;
    S : out BIT_VECTOR (3 downto 0)
  ) ;
end ADDSUB;

architecture ARCHI of ADDSUB is begin
  process (A, B, ADD_SUB)
    variable AUX : BIT_VECTOR (3 downto 0) ;
  begin
    if ADD_SUB = '1' then
      AUX := A + B ;
    else
      AUX := A - B ;
    end if;
    S <= AUX;
  end process;
end ARCHI;
```

IF ... ELSE

- Naredba *IF...ELSE* provjerava istinitost uvjeta.
- Ako je uvjet istinit izvršava se prvi dio uvjeta, a ukoliko uvjet nije istinit ili je rezultat uvjeta “x” ili “z” izvršava se “*else*” uvjet.

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

entity mux4 is port (
    a, b, c, d : in std_logic_vector (7 downto 0);
    sel1, sel2 : in std_logic;
    outmux : out std__logic_vector (7 downto 0) );
end mux4;

architecture behavior of mux4 is
begin
    process (a, b, c, d, sel1, sel2)
    begin
        if (sel1 = '1') then
            if (sel2 = '1') then
                outmux <= a;
            else
                outmux < = b;
            end if;
        else
            if (sel2 = '1') then
                outmux < = c;
            else
                outmux <= d;
            end if;
        end if;
    end process;
end behavior;
```

CASE

- *CASE* naredba vrši usporedbu na temelju kojeg određuje u kojoj će grani završiti.
- Ako ni jedna grana ne zadovoljava uvjet izvršava se zadnja grana koja nosi naziv “*others*”.

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

entity mux4 is port (
    a, b, c, d : in std_logic_vector (7 downto 0);
    sel : in std_logic_vector (1 downto 0);
    outmux : out std_logic_vector (7 downto 0)
) ;
end mux4;

architecture behavior of mux4 is
begin
    process (a, b, c, d, sel)
    begin
        case sel is
            when "00" => outmux <= a;
            when "01" => outmux <= b;
            when "10" => outmux <= c;
            when others => outmux <= d;
        end case;
    end process;
end behavior;
```

FOR ... LOOP

- Naredba FOR ... LOOP mora zadovoljavati sljedeće uvijete:
 - konstantne vrijednosti petlje
 - zaustavljanje petlje korištenjem operacija “<=”, “<”, “>” ili “>=”
 - “next” i “exit” naredbe su podržane.

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

entity countzeros is port (
    a : in std_logic_vector (7 downto 0) ;
    Count : out std_logic_vector (2 downto 0));
end mux4;

architecture behavior of mux4 is
    signal Count_Aux: std_logic_vector (2 downto );
begin
    process (a)
    begin
        Count_Aux <= "000";
        for i in a'range loop
            if (a[i] = '0' ) then
                Count_Aux <= Count_Aux + 1;
            end if;
        end loop;
        Count <= Count_Aux;
    end process;
end behavior;
```

Procesna sekvencijalna logika

- XST podržava dvije vrste procesne sekvencijalne logike.
 - procesnu logiku sa listom događaja
 - procesnu logiku bez liste događaja.

Sekvencijalni proces s listom događaja

- Proces je sekvencijalan u koliko mora koristiti memorijske elemente, odnosno u koliko neki od signala nije eksplicitno definiran u svima granama procesa. U takvom slučaju XST generira memorijske elemente.
- Asinkroni signali moraju biti definirani u listi događaja, a ako signal nije naveden u listi događaja XST dodaje signal automatski i generira upozorenje. Jednako kako i kod kombinacijske logike, ova upozorenja treba uzeti u obzir, jer mogu promijeniti primarnu funkcionalnost logike.

```
process (CLK, RST)
begin
    If RST = '1' then
        --asinkroni dio koda
        ...
    elsif <CLK'EVENT and CLK = '1' then
        -- sinkroni dio koda
    end if;
end process;
```

Sekvencijalni proces bez liste događaja

- Sekvencijalni proces bez liste događaja mora sadržavati “*WAIT*” naredbu, koja mora biti prva naredba unutar procesa.
- Uvjet unutar “*WAIT*” naredbe mora biti uvjet na signalu koji je takt vremenskog vođenja.
- Asinkrone dijelove nije moguće definirati bez liste događaja.
- Višestruke “*WAIT*” naredbe mogu se pojaviti unutar procesa uz određene uvijete.

Sekvencijalni proces bez liste događaja

```
process
begin
    wait until CLK'EVENT and CLK = '1';
    -- sinkroni dio koda
end process;
```

- XST ne podržava upotrebu takta vremenskog vođenja i signala koji ga kontrolira unutar iste *WAIT* naredbe:

Nije podržano:

```
wait until CLOCK'EVENT and CLOCK='0' and ENABLE='1';
```

Treba pisati:

```
wait until CLOCK'EVENT and CLOCK='0';
if ENABLE='1' then ...
```

8-bitni registar definiran korištenjem liste događaja

```
entity EXAMPLE is
  port (
    DI   : in BIT_VECTOR (7 downto 0);
    CLK  : in BIT;
    DO   : out BIT_VECTOR (7 downto 0)
  ) ;
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
  process (CLK)
  begin
    if CLK'EVENT and CLK = '1' then
      DO <= DI ;
    end if;
  end process;
end ARCHI;
```

8-bitni registar definiran korištenjem "WAIT"-a

```
entity EXAMPLE is port (  
    DI   : in BIT_VECTOR (7 downto 0);  
    CLK  : in BIT;  
    DO   : out BIT_VECTOR (7 downto 0)) ;  
end EXAMPLE;  
  
architecture ARCHI of EXAMPLE is  
begin  
    process  
    begin  
        wait until CLK'EVENT and CLK = '1';  
        DO <= DI;  
    end process;  
end ARCHI;
```

8-bitni registar s asinkronim resetom

```
entity EXAMPLE is
  port (
    DI   : in BIT_VECTOR (7 downto 0);
    CLK  : in BIT;
    RST  : in BIT;
    DO   : out BIT_VECTOR (7 downto 0)
  );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
  process (CLK, RST)
  begin
    if RST = '1' then
      DO <= "00000000";
    elsif CLK'EVENT and CLK = '1' then
      DO <= DI ;
    end if;
  end process;
end ARCHI;
```

8-bitni brojilo s asinkronim resetom

```
library ASYL;
use ASYL.PKG_ARITH.all;

entity EXAMPLE is port (
    CLK : in BIT;
    RST : in BIT;
    DO  : out BIT_VECTOR (7 downto 0)) ;
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
    process (CLK, RST)
        variable COUNT : BIT_VECTOR (7 downto 0);
    begin
        if RST = ' 1' then
            COUNT := "00000000";
        elsif CLK'EVENT and CLK = '1' then
            COUNT := COUNT + "00000001";
        end if;
        DO <= COUNT;
    end process;
end ARCHI;
```

Višestruka “WAIT” naredba

- Unutar sekvencijalne strukture moguće je koristiti višestruku “*WAIT*” naredbu.
- Za korištenje višestruke “*WAIT*” naredbe, XST postavlja sljedeće uvjet:
 - proces može sadržati samo jednu petlju
 - prva naredba unutar petlje mora biti *WAIT*
 - nakon svakog *WAIT*-a mora biti *NEXT* ili *EXIT*
 - uvjet unutar *WAIT*-a mora biti isti u svim *WAIT*-ovima
 - uvjet mora koristiti samo jedan signal i to signal vremenskog vođenja
- Sljedeći primjer koristi višestruku “*WAIT*” naredbu. Primjer opisuje četiri različite operacije koje se izvršavaju jedna za drugom uz mogućnost ponovnog pokretanja procesa generiranjem signala *RST*. Proces naizmjenično pridružuje vrijednosti *DATA1*, *DATA2*, *DATA3* i *DATA4* izlaznom signalu *RESULT*.


```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity EXAMPLE is port (
    DATA1, DATA2, DATA3, DATA4 : in STD_LOGIC_VECTOR (3 downto 0);
    RESULT : out STD_LOGIC_VECTOR (3 downto 0);
    CLK : in STD_LOGIC;
    RST : in STD_LOGIC) ;
end EXAMPLE;

architecture ARCH of EXAMPLE is
begin
    process begin
    SEQ_LOOP : loop
        wait until CLK'EVENT and CLK = '1';
        exit SEQ_LOOP when RST = ' 1' ;
        RESULT <= DATA1;

        wait until CLK'EVENT and CLK = '1';
        exit SEQ_LOOP when RST = ' 1' ;
        RESULT <= DATA2;

        wait until CLK'EVENT and CLK = '1';
        exit SEQ_LOOP when RST = '1';
        RESULT <= DATA3;

        wait until CLK'EVENT and CLK = '1';
        exit SEQ_LOOP when RST = '1';
        RESULT <= DATA4;
    end loop;
    end process;
end ARCH;
```

Funkcije i procedure

- Funkcije i procedure omogućuju korištenje blokova koji se višestruko ponavljaju u dizajnu.
- Moraju biti definirane unutar deklaracijskog dijela entiteta, unutar arhitekture ili paketa.
- Zaglavlje sadrži ulazne parametre za funkciju i ulazno izlazne parametre za procedure.
- Sljedeći primjer prikazuje deklaraciju unutar paketa. Tijelo paketa sadrži jednobitno zbrajalo «ADD».

Definicija Funkcije

```
package PKG is
  function ADD (A,B, CIN : BIT )
    return BIT_VECTOR(1 downto 0);
end PKG;
```

```
package body PKG is
  function ADD (A,B, CIN : BIT )
    return BIT_VECTOR(1 downto 0) is
    variable S, COUT : BIT;
    variable RESULT : BIT_VECTOR (1 downto 0);
  begin
    S := A xor B xor CIN;
    COUT := (A and B) or (A and CIN) or (B and CIN);
    RESULT := COUT & S;
    return RESULT;
  end ADD;
end PKG;
```

Poziv funkcije

```
use work.PKG.all;

entity EXAMPLE is port (
    A,B : in BIT_VECTOR (3 downto 0);
    CIN : in BIT;
    S : out BIT_VECTOR (3 downto 0);
    COUT : out BIT);
end EXAMPLE;

architecture ARCHI of EXAMPLE is
    signal S0, S1, S2, S3 : BIT_VECTOR (1 downto 0);
begin
    S0 <= ADD (A(0), B(0), CIN) ;
    S1 <= ADD (A(1), B(1), S0(1));
    S2 <= ADD (A(2), B(2), S1(1));
    S3 <= ADD (A(3), B(3), S2(1)) ;
    S <= S3(0) & S2(0) & S1(0) & S0(0);
    COUT <= S3 (1) ;
end ARCHI;
```

Deklaracija procedure

```
package PKG is
  procedure ADD (
    A,B, CIN : in BIT;
    C : out BIT_VECTOR (1 downto 0));
end PKG;
```

```
package body PKG is
  procedure ADD (
    A,B, CIN : in BIT;
    C : out BIT_VECTOR (1 downto 0)) is

    variable S, COUT : BIT;

  begin
    S := A xor B xor CIN;
    COUT := (A and B) or (A and CIN) or (B and CIN);
    C := COUT & S;
  end ADD;
end PKG;
```

Poziv procedure

```
use work.PKG.all;

entity EXAMPLE is port (
    A,B : in BIT_VECTOR (3 downto 0);
    CIN : in BIT;
    S : out BIT_VECTOR (3 downto 0);
    COUT : out BIT);
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
    process (A,B,CIN)
        variable S0, S1, S2, S3 : BIT_VECTOR (1 downto 0)
    begin
        ADD (A(0) , B(0) , CIN, S0) ;
        ADD (A(1) , B(1) , S0 (1) , S1) ;
        ADD (A(2) , B(2) , S1 (1) , S2) ;
        ADD (A(3) , B(3) , S2 (1) , S3) ;
        S <= S3(0) & S2(0) & S1(0) & S0(0);
        COUT <= S3(1);
    end process;
end ARCHI;
```

Iznimke

- Rad sa iznimkama podržan je u XST alatu.
- Korištenjem iznimaka dizajner može detektirati i kontrolirati situacije unutar svog VHDL dizajna kao što su neispravni parametri poslani nekoj funkciji ili neispravne početne vrijednosti inicijalizirane komponente.
- Za svako generiranje iznimke XST kreira upozorenje koje se može definirati na nekoliko nivoa.
- Nivoi nam omogućuju generiranje upozoravajućih poruka zaustavljanje cijeli proces sinteze. XST podržava iznimke samo s statičkim uvjetima.
- Sljedeći primjer prikazuje blok koji opisuje pomični registar SING_SLR. Širina pomičnog registra definirana je s SRL_WIDTH generičkom varijablom. Unutar bloka koristi se iznimka da bi se ograničila maksimalna veličina pomičnog registra na 17 bitova. Arhitektura TOP referencira dvije komponente SING_SLR, prvu širine 13 a drugu širine 18. Prevođenjem takvog dizajna u XST dobivamo obavijest o pogrešci nastaloj prilikom prevođenja.

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

entity SINGE_SRL is generic (SRL_WIDTH : integer := 16);
  port (
    elk : in std_logic ;
    inp : in std_logic;
    outp : out std_logic
  );
end SINGE_SRL;

architecture beh of SINGE_SRL is
  signal shift_reg : std_logic_vector (SRL_WIDTH-1 downto 0);
begin
  assert SRL_WIDTH <= 17
    report "The size of Shift Register exceeds the size of a single SRL"
    severity FAILURE;

  process (clk)
  begin
    if (clk'event and clk = '1') then
      shift_reg <= shift_reg (SRL_WIDTH-1 downto 1) & inp;
    end if;
  end process;

  outp <= shift_reg(SRL_WIDTH-1);
end beh;
```



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

entity TOP is port (
    clk : in std_logic;
    inp1, inp2 : in std_logic;
    outp1, outp2 : out std_logic );
end TOP;

architecture beh of TOP is
    component SINGE_SRL is generic (SRL_WIDTH : integer := 16);
    port (
        clk : in std_logic;
        inp : in std_logic;
        outp : out std_logic );
    end component;
begin
    inst1: SINGE_SRL generic map (SRL_WIDTH => 13)
        port map(
            clk => clk, inp => inp1, outp => outp1 ) ;
    inst2: SINGE_SRL generic map (SRL_WIDTH => 18)
        port map(
            clk => clk, inp => inp2, outp => outp2 ) ;
end beh;
```

Rezultat izvođenja

```
=====
*                               HDL Analysis                               *
=====
```

```
Analyzing Entity <top> (Architecture <beh>).
  Entity <top> analyzed. Unit <top> generated.
```

```
Analyzing generic Entity <singe_srl> (Architecture <beh>).
  SRL_WIDTH =13
Entity <singe_srl> analyzed. Unit <singe_srl> generated.
```

```
Analyzing generic Entity <singe_srl> (Architecture <beh>).
  SRL_WIDTH =18
```

```
ERROR:Xst - assert_1.vhd line 15: FAILURE: The size of Shift Register
exceeds the size of a single SRL
```

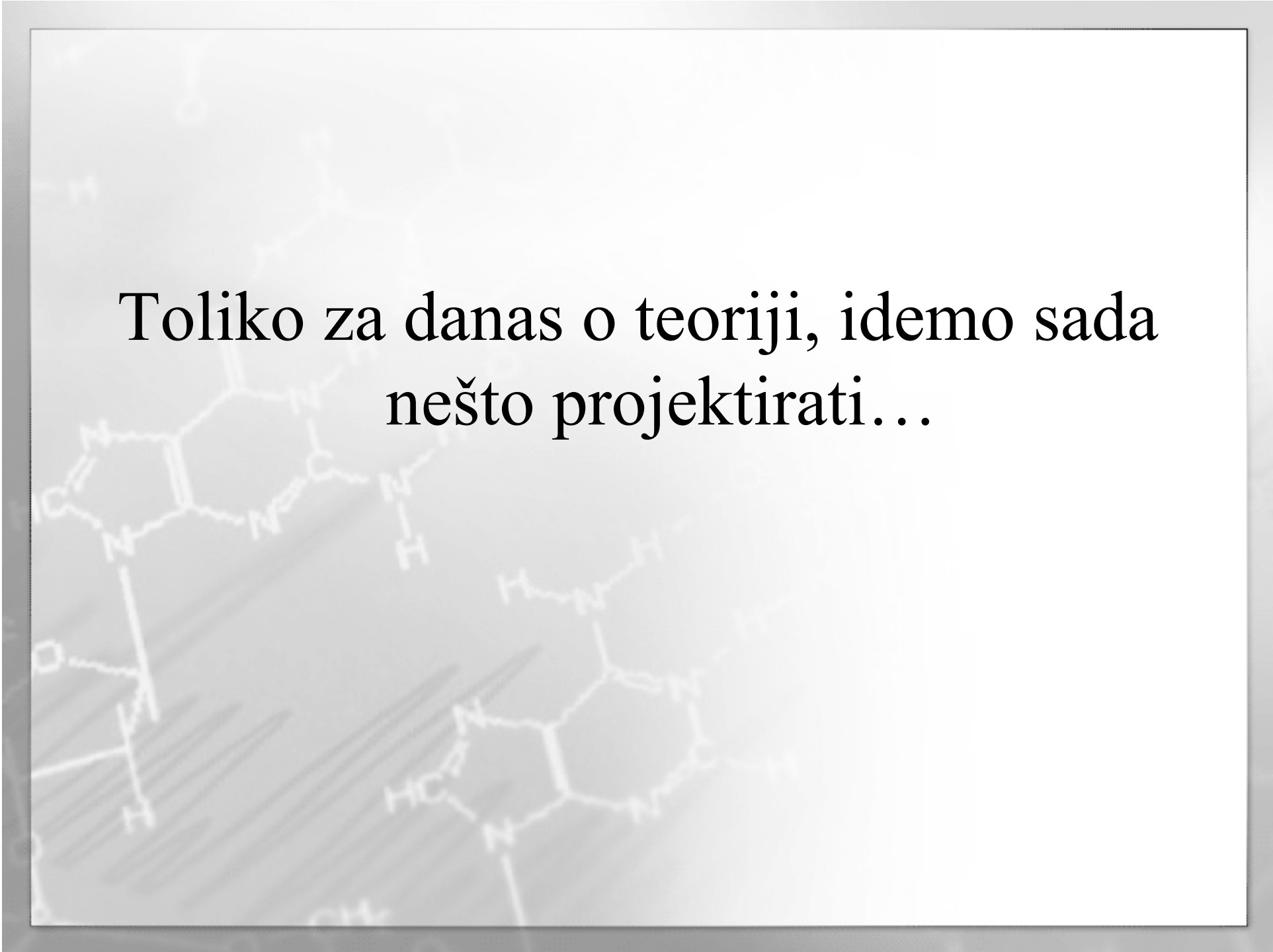
Paketi

- VHDL arhitekture mogu biti definirane korištenjem paketa.
- Paketi mogu sadržati definicije tipova i podtipova, deklaraciju konstanti, funkcije, procedure i opise komponenti.
- Paketi nam omogućuju jednostavniju kontrolu i promjenu dizajna a sastoje se od dva dijela: deklaracije paketa i tijela paketa.
- Tijelo paketa sadrži opis funkcija koje su deklarirane u deklaraciji paketa.
- Da bi koristili pojedine pakete potrebno ih je uključiti u projekt. Paketi se uključuju u projekte dodavanjem sljedećih linija koda u dizajn:

```
library lib_pack;  
-- lib_pak je ime datoteke paketa koji se uključuje  
use lib_pack.pack_name.all  
-- pack_name je ime paketa iz datoteke koji se želi koristiti.  
-- all predstavlja da se sve što je definirano u paketu koristiti
```

Paketi

- XST također podržava predefinirane pakete, to su paketi koji su već prevedeni i mogu se uključiti u VHDL dizajn. Takvi paketi namijenjeni su za upotrebu prilikom sinteze, ali se mogu koristiti i tijekom simulacije.
- Standardni paketi sadrže definiciju osnovnih tipova (bit, bit_vector i integer), te ih nije potrebno posebno dodavati u projekt, oni su automatski pridružen projektu.
- IEEE paketi
 - Std_logic_1164: definira tipove std_logic, std_ulogic, std_logic_vector, std_logic_uvector i pripadajuće funkcije bazirane na tim tipovima podataka.
 - Numeric_bit: podržava vektore sa i bez predznaka zasnovanih na tipu podatka bit i sve pripadajuće funkcije za rad s njima.
 - Numeric_std: : podržava vektore sa i bez predznaka zasnovanih na tipu podatka std_logic i sve pripadajuće funkcije za rad s njima.
 - Math_real

The background of the slide features several faint, overlapping chemical structures. These include a complex polycyclic aromatic system with multiple nitrogen atoms (possibly a nucleic acid base or a complex organic molecule) and a smaller, simpler structure that appears to be a substituted benzene or a similar ring system. The structures are rendered in a light gray color, providing a scientific context for the text.

Toliko za danas o teoriji, idemo sada
nešto projektirati...