Latvijas Universitāte
Datorikas fakultāte

Aparatūras apraksta valodas. VHDL.

Kurss "Ievads digitālajā projektēšanā"

Lekcija 13.11.2010

Autors: **Artis Mednis**

Rīga 2010

VHDL

- ***V****HSIC* ***h****ardware* ***d****escription* ***l****anguage*
- ***v****ery-****h****igh-****s****peed* ***i****ntegrated* ***c****ircuit*

- Izmanto gan digitālo, gan jaukto signālu shēmu projektēšanai

- Izstrādāta pēc ASV Aizsardzības ministrijas (*US Department of Defense*) pasūtījuma

- Sākotnējais pielietojums – jau esošu elektronisko shēmu dokumentēšana

Papildus funkcionalitāte

- Ja mums ir dokumentācija, kas apraksta shēmu, nebūtu slikti, ja to varētu izmantot shēmas darbības simulēšanai

- Nākošais solis – ņemam dokumentāciju un sintezējam shēmas implementāciju fiziskā līmenī, piemēram:
  - Atmiņas blokus (RAM)
  - Skaitītājus
  - Arimētiskas operācijas veicošus blokus

- Sintezējot shēmu, var iet dažādus ceļus:
  - Minimāla aizņemtā vieta
  - Minimāls energijas patēriņš
  - Maksimāls darbības ātrums (CLK frekvence)

VHDL un ADA

- ADA – radīta pēc iepriekš minētās iestādes pasūtījuma 20.gs. 80to gadu sākumā, lai aizstātu līdz šim izmantoto valodu *zvērudārzu* ☺

- VHDL kopīgais ar ADA:
  - Paralēlu procesu apraksts
  - Sintakse (daļēja atbilstība)
  - Stingri noteikti datu tipi (*strongly typed*)
  - Reģistru neatkarība (*non case-sensitive*)

- VHDL atšķirības no ADA:
  - Loģiskās operacijas NAND, NOR
  - Masīvus var indeksēt gan augošā, gan dilstošā secībā

Simulēšana

- Divi atšķirīgi VHDL pielietojumi
  - Elektronisko shēmu simulēšana
  - Elektronisko shēmu sintezēšana
- Funkcionālā simulācija
  - Tiek ņemta vērā tikai loģika
- Laika (*timing*) simulācija
  - Tiek ņemtas vērā arī aiztures, kuras radīsies fiziskajā shēmas implementācijā

Sintezēšana

- Sintezēšanas izmaksas atšķirībā no mērķa platformas:
  - Priekš FPGA – rīki pieejami par brīvu vai *pa lēto*
  - Priekš ASIC – rīki ir, bet lielākajā daļā gadījumu *pa dārgo*

- Tikai daļa VHDL konstrukciju pakļaujas sintezēšanai
  - *wait for 10 ns* – simulēsies, bet nesintezēsies ☹
  - dažādiem rīkiem – dažādas sintezēšanas iespējas
  - eksistē *oficiālā sintezējamā valodas apakškopa*, pie kuras pieturēties nosaka *labā prakse*

VHDL koda paraugi I (AND)

- *Entity* – apraksta saskarni
- *Port* – apraksta katru ieeju/izeju
- *Architecture* – apraksta implemetāciju
- Var tikt importētas papildus bibliotēkas

```
-- (this is a VHDL comment)

-- import std_logic from the IEEE library
library IEEE;
use IEEE.std_logic_1164.all;

-- this is the entity
entity ANDGATE is
   port (
         IN1 : in std_logic;
         IN2 : in std_logic;
         OUT1: out std_logic);
end ANDGATE;

architecture RTL of ANDGATE is
begin

  OUT1 <= IN1 and IN2;

end RTL;
```

Mazliet par *std_logic*

- Deviņi iespējamie stāvokļi:
  - 'U' - uninitialized
  - 'X' - strong drive, unknown logic value
  - **'0' - strong drive, logic zero**
  - **'1' - strong drive, logic one**
  - **'Z' - high impedance**
  - 'W' - weak drive, unknown logic value
  - 'L' - weak drive, logic zero
  - 'H' - weak drive, logic one
  - '-' - don't care

- Vai mums tas ir vajadzīgs, ja strādājam tikai ar 0 un 1?
- Izrādās, ka šādā ceļā var panākt labāku simulāciju un atkļūdošanu

VHDL koda paraugi II (MUX)

- Var rakstīt kodu īsu un nepārskatāmu
- Var rakstīt kodu garu, bet parskatāmu
- *Process* bloks – tā iekšienē kods tiek izpildīts pēc kārtas

```
-- template 1:
X <= A when S = '1' else B;

-- template 2:
with S select
  X <= A when '1',
       B when others;

-- template 3:
process(A,B,S)
begin
  case S is
    when '1'    => X <= A;
    when others => X <= B;
  end case;
end process;

-- template 4:
process(A,B,S)
begin
  if S = '1' then
    X <= A;
  else
    X <= B;
  end if;
end process;

-- template 5 - 4:1 MUX, where S is a 2-bit std_logic_vector :
process(A,B,C,D,S)
begin
  case S is
    when "00"   => X <= A;
    when "01"   => X <= B;
    when "10"   => X <= C;
    when others => X <= D;
  end case;
end process;
```

VHDL koda paraugi III (Latch)

- Bez RESET

```
-- latch template 1:
Q <= D when Enable = '1' else Q;

-- latch template 2:
process(D,Enable)
begin
  if Enable = '1' then
    Q <= D;
  end if;
end process;
```

Ar RESET

```
-- SR-latch template 1:
Q <= '1' when S = '1' else
  '0' when R = '1' else Q;

-- SR-latch template 2:
process(S,R)
begin
  if S = '1' then
    Q <= '1';
  elsif R = '1' then
    Q <= '0';
  end if;
end process;
```

- Pārtraukums 10 minūtes

VHDL koda paraugi IV (Flip-flop)

- Noteiktos gadījumos nepieciešams veikt darbību, balstoties uz CLK signālu:

```
-- simplest DFF template (not recommended)
Q <= D when rising_edge(CLK);

-- recommended DFF template:
process(CLK)
begin
  -- use falling_edge(CLK) to sample at the falling edge instead
  if rising_edge(CLK) then
    Q <= D;
  end if;
end process;

-- alternative DFF template:
process
begin
  wait until rising_edge(CLK);
  Q <= D;
end process;

-- alternative template:
process(CLK)
begin
   if CLK = '1' and CLK'event then--use rising edge, use  "if CLK = '0' and CLK'event" instead for falling edge
      Q <= D;
   end if;
end process;
```

VHDL koda paraugi V (Flip-flop)

- Kombinējam CLK ar citiem signāliem, piemēram, RESET:

```
-- template for asynchronous reset with clock enable:
process(CLK, RESET)
begin
  if RESET = '1' then   -- or '0' if RESET is active low...
    Q <= '0';
  elsif rising_edge(CLK) then
    if Enable = '1' then  -- or '0' if Enable is active low...
      Q <= D;
    end if;
  end if;
end process;

-- template for synchronous reset with clock enable:
process(CLK)
begin
  if rising_edge(CLK) then
    if RESET = '1' then
      Q <= '0';
    elsif Enable = '1' then  -- or '0' if Enable is active low...
      Q <= D;
    end if;
  end if;
end process;
```

VHDL izmantošanas veidi

- Aprakstīt aparatūras struktūru (velkam paralēles ar shēmu zīmēšanu)
- Aprakstīt aparatūras datu plūsmu
- Aprakstīt aparatūras *izturēšanos (behavior)*

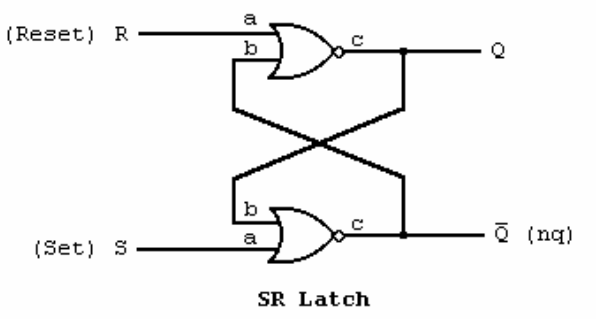
- *Parasti* tiek izmantota kombinācija no visām trim minētajām metodēm

Dažādu metožu pielietojums

- *Entity*

```
entity latch is
  port (s,r: in bit;
        q,nq: out bit);
end latch;
```

Shēma



SR Latch

- *Architecture I*

```
architecture dataflow of latch is
  signal q0 : bit := '0';
  signal nq0 : bit := '1';
begin
  q0<=r nor nq0;
  nq0<=s nor q0;

  nq<=nq0;
  q<=q0;
end dataflow;
```

Architecture II

```
architecture structure of latch is
  component nor_gate
    port (a,b: in bit;
          c: out bit);
  end component;
begin
  n1: nor_gate
    port map (r,nq,q);
  n2: nor_gate
    port map (s,q,nq);
end structure;
```

Konstrukcijas tikai simulācijai

- Lielu daļu VHDL konstrukciju **NEVAR** sintezēt:
  - Izmanto prototipēšanai, simulēšanai, atkļūdošanai
  - Var veidot sarežģītus signālus testēšanas vajadzībām

```
process
begin
  CLK <= '1'; wait for 10 ns;
  CLK <= '0'; wait for 10 ns;
end process;
```

```
process
begin
  wait until START = '1'; -- wait until START is high

  for i in 1 to 10 loop -- then wait for a few clock periods...
    wait until rising_edge(CLK);
  end loop;

  for i in 1 to 10 loop          -- write numbers 1 to 10 to DATA, 1 every cycle
    DATA <= to_unsigned(i, 8);
    wait until rising_edge(CLK);
  end loop;

  -- wait until the output changes
  wait on RESULT;

  -- now raise ACK for clock period
  ACK <= '1';
  wait until rising_edge(CLK);
  ACK <= '0';


  -- and so on...
end process;
```

Mazliet par īpašiem datu tipiem

- bit_vector

```
entity demux is
  port (e: in bit_vector (3 downto 0);      -- enables for each output
        s: in bit_vector (1 downto 0);      -- select signals
        d: out bit_vector (3 downto 0));    -- four output signals
end demux;

architecture rtl of demux is
  signal t : bit_vector(3 downto 0);        -- an internal signal
begin
  t(3)<=s(1) and s(0);
  t(2)<=s(1) and not s(0);
  t(1)<=not s(1) and s(0);
  t(0)<=not s(1) and not s(0);
  d<=e and t;
end rtl;
```

- time
  - sastāv no divām daļām, skaitļa un mērvienības

Datu uzglabāšana

- Atceramies Verilog – ar ko atšķiras *reg* un *wire*?
- VHDL izmanto *signal* un *variable*

- *Signal*
  - izmanto struktūru un datu plūsmu aprakstos
  - piešķiršana rada notikumu (*event*)
  - ja notikuma rašanās brīdī *iet* process, vispirms tiek pabeigts tas, tikai pēc tam apstrādats notikums
- *Variable*
  - izmanto tikai *process* bloku iekšienē
  - *uzvedas* kā mainīgais programmatūras izstrādē
  - nereaģē un nerada notikumus

Izvade

- Izmantojam papildus bibliotēku *std.textio.all*
- Ērta iespēja atkļūdošanai

```
use textio.all;
architecture behavior of check is
begin
  process (x)
    variable s : line;
    variable cnt : integer:=0;
  begin
    if (x='1' and x'last_value='0') then
      cnt:=cnt+1;
      if (cnt>MAX_COUNT) then
        write(s,"Counter overflow - ");
        write(s,cnt);
        writeline(output,s);
      end if;
    end if;
  end process;
end behavior;
```

Praktiskie darbi

- Sākam darbu pie kursa projekta KP3

Pateicos par uzmanību!

Jautājumi?