VHDL

VHDL - Very high speed integrated circuit Hardware Description Language

VHDL är ett komplext språk, från början avsett för att beskriva digitala system på olika abstraktionsnivåer (beteende- och strukturmässigt). Idag går det att skapa verklig hårdvara med VHDL, men endast en liten del av VHDLs syntax går att syntetisera till hårdvara.

Entity & Architecture

En VHDL-fil som beskriver en krets följer ofta mönstret nedan.

```
entity krets is
      port(
            insignal : in std_logic;
            utsignal : out std_logic
      );
end entity;

architecture beteende of krets is
      signal intern_signaler : std_logic;
begin
      ...
end architecture;
```

Avsnittet `entity` beskriver hur kretsen ser ut från utsidan, det vill säga vilka in- och utsignaler den har. I avsnittet `architecture` beskrivs kretsens funktion beteendemässigt och/eller strukturmässigt.

Signal

Interna signaler deklareras i `architecture` innan ordet `begin`. Signaler syntetiseras till ledningar/sladdar i hårdvara eller ledningar/sladdar ”med minne” om de beskrivs som register (vippor) eller latchar.

std_logic & std_logic_vector

Datatyp som ska efterlikna egenskaper hos elektriska signaler.

Kan anta värdena:

- ’1’,’0’ : driver signalen hög resp. låg.

- 'Z': högimpedansig, tri-state.
- 'L', 'H': svag låg och svag hög.
- 'X', 'W': okänd och svag okänk. Signalen drivs från två ställen med olika nivåer.
- 'U': ej initialiserad.
- '-': don't care.

Allmänt går endast '1' och '0' (och ibland 'Z') att syntetisera.

Datatyperna `std_logic` & `std_logic_vector` är definerade i `IEEE.std_logic_1164.all`. Datatyperna blir tillgängliga att använda i en VHDL-fil med följande syntax.

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

Egna Datyper

Ibland kan det vara motiverat att använda egendefinierade datatyper. Till exempel vid konstruktion av en tillståndsmaskin kan det vara lämpligt att definiera en datatyp som beskriver nuvarande- och nästatillstånd på ett bättre sätt än `std_logic` eller `std_logic_vector`.

```
type state_type is (INSIDE, OUTSIDE);
```

Datatypen `state_type` som definierats ovan kan endast anta två värden, `INSIDE` och `OUTSIDE`. Efter att en egen datatyp definierats så kan den användas på samma sätt som övrgia datatyper.

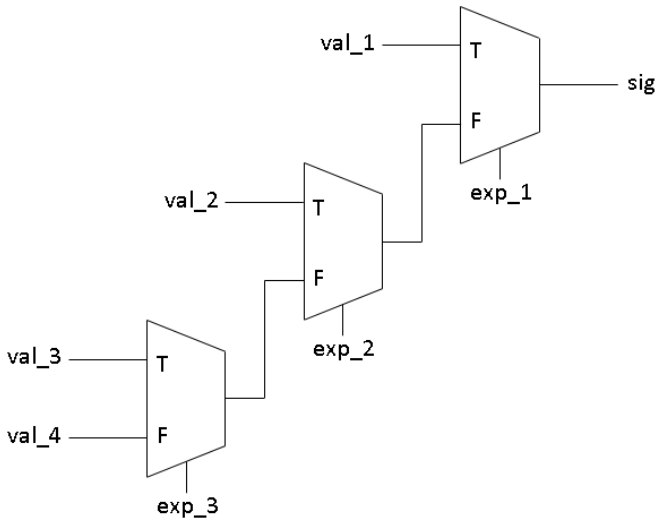
```
signal current_state : state_type;
...
if current_state = INSIDE then
...
```

Conditional Signal Assignment

Syntax:

```
sig <=  val_1 when exp_1 else
        val_2 when exp_2 else
        val_3 when exp_3 else
        val_4;
```

Motsvarande hårdvara



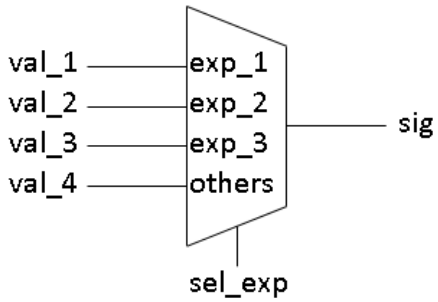
- Kan endast användas utanför en process.
- De booleska uttrycken `exp_i` utvärderas succesivt tills ett sant uttryck hittas och motsvarande värde `val_i` tilldelas då utsignalen. Om inget av uttrycken är sanna tilldelas utsignalen värdet `val_4` i exemplet ovan.
- Flera uttryck kan vara sanna men det som kommer först har högst prioritet.

Selected Signal Assignment

Syntax:

```
with sel_exp select
             sig <=  val_1 when exp_1,
                     val_2 when exp_2,
                     val_3 when exp_3,
                     val_4 when others;
```

Motsvarande hårdvara



- Kan endast användas utanför en process.
- Alla möjliga värden av signalen `sel_exp` måste täckas av ETT OCH ENDAST ETT av uttrycken `exp_i`. Det vill säga, endast ett av uttrycken får vara sant.
- `others` i exemplet ovan täcker alla fall som inte tagits med tidigare.

Process

Syntax:

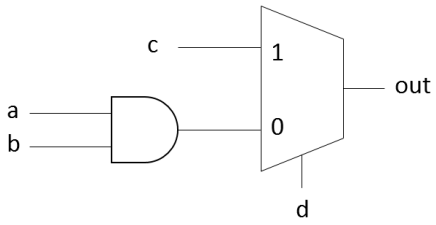
```
process_label:process(sensitivity_list)
begin
              sequential_statement1;
              sequential_statement2;
              ...
end process;
```

- En process innehåller en serie instruktioner som exekveras ”sekventiellt”.
- I VHDL finns en uppsjö av sekventiella instruktioner. Dock så saknar många en tydlig motsvarighet i hårdvara, vilket gör dem svåra att syntetisera.
- Processens känslighetslista bestämmer vilka signaler som triggar processen. Då en signal i känslighetslistan ändrar värde aktiveras processen och nya utsignaler genereras.
- En signal kan tilldelas värden från flera olika ställen i en process. Men den kommer bara anta det värde som den tilldelats sist i processen, inga mellanliggande värden tilldelas.

Exempel: Tilldelning från flera ställen

```
p1:process(a, b, c, d)
begin
       o <= a and b;
       if (d = '1') then;
              o <= c;
       end if;
end process;
```

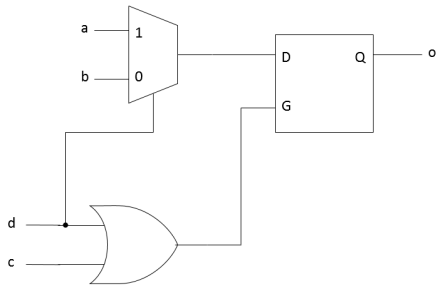
Motsvarande hårdvara



Om en signal inte alltid tilldelas ett värde

```
p1:process(a, b)
begin
      if (d = '1') then;
            o <= a;
      elsif (c = '1') then
            o <= b;
     end if;
end process;
```

Motsvarande hårdvara



Eftersom inget i processen talar om vad `o` ska ha för värde då `c=0` och `d=0` så antar syntetiseringsverktyget att tidigare värde ska kommas ihåg och en latch implementeras i hårdvaran. Latchar bör undvikas då de kan introducera kapplöpning (race) mellan signaler. Ofta varnar syntetiseringsverktyg för latchar i designen vilka ofta beror på att man glömt att tilldela en signal värde i alla lägen.

```
signal a, b : std_logic;
...
p2:process(a, b)
begin
      if (a = '1') then;
            o <= b;
      elsif (a = '0') then
            o <= '0';
     end if;
end process;
```

Även i exemplet ovan så täcks inte alla fall eftersom signalen `a` är av typen `std_logic` vilken kan anta 9 olika värden. Använd istället en else-stats eller standardvärden.

Kod med else-stats.

```
signal a, b : std_logic;
...
p3:process(a, b)
begin
      if (a = '1') then;
            o <= b;
      else
            o <= '0';
     end if;
end process;
```

Kod med standardvärde innan if-stats.

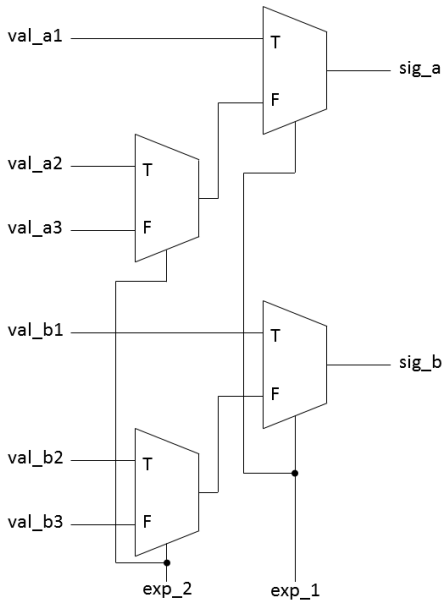
```
signal a, b : std_logic;
...
p2:process(a, b)
begin
      o <= '0';
      if (a = '1') then;
            o <= b;
     end if;
end process;
```

IF Statement

Syntax:

```
if (exp_1) then
      sig_a1 <= val_a1;
      sig_b1 <= val_b1;
elsif (exp_2) then
      sig_a1 <= val_a2;
      sig_b1 <= val_b2;
else
      sig_a1 <= val_a3;
      sig_b1 <= val_b3;
end if;
```

Motsvarande hårdvara



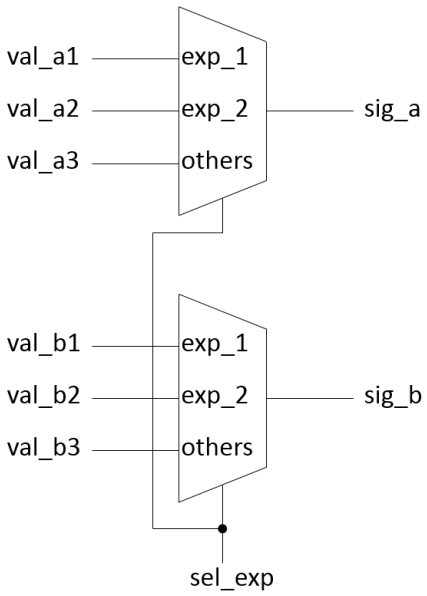
- Kan endast användas i en process.
- Flera uttryck `exp_i` kan vara sanna men det första har högst prioritet.

CASE Statement

Syntax:

```
case sel_exp is
      when exp_1 =>
            sig_a <= val_a1;
            sig_b <= val_b1;
      when exp_2 =>
            sig_a <= val_a2;
            sig_b <= val_b2;
      when others =>
            sig_a <= val_a3;
            sig_b <= val_b3;
end case;
```

Motsvarande hårdvara



- Kan endast användas i en process.
- Endast ett uttryck `exp_i` kan vara sant. Alla fall måste täckas.
- `others` i exemplet ovan täcker alla fall som inte tagits med tidigare.

Register

Syntax D-vippa, positivt flanktriggad:

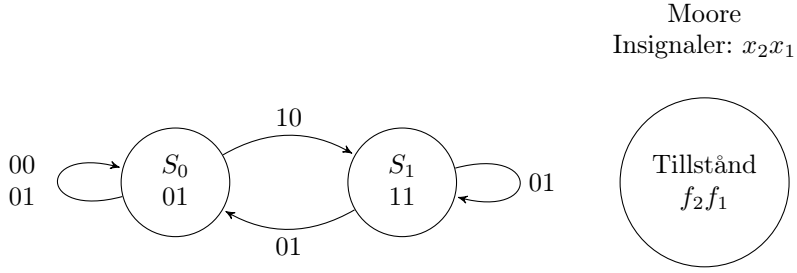
```
reg:process(clk)
begin
      if (clk'event and clk='1') then
           Q <= D;
      end if;
end process;
```

Syntax D-vippa med asynkron aktiv hög nollställning:

```
reg:process(clk, rst)
begin
      if (rst = '1') then
           Q <= '0';
      elsif (rising_edge(clk)) then
           Q <= D;
      end if;
end process;
```

Observera att `(clk'event and clk='1') = rising_edge(clk)`.

Expempel på sekvensnät



```
architecture Behavioral of exempel is

     type state_type is (S0, S1);
     signal current_state, next_state : state_type;

      register_update:process(clk, rst)
      begin
           if (rst = '1') then
                current_state <= S0;
           elsif (rising_edge(clk)) then
                current_state <= next_state;
           end if;
     end process;

      next_state_logic:process(current_state, x1, x2)
      begin
           --Default value
           next_state <= current_state;

           case current_state is
                when S0 =>
                     if ((x1 = '0') and (x2 = '1')) then
                          next_state <= S1;
                     end if;
                when S1 =>
                     if ((x1 = '1') and (x2 = '0')) then
                          next_state <= S0;
                     end if;
           end case;
      end process;

      moore_output_logic:process(current_state)
      begin
           case current_state is
                when S0 =>
                     f1 <= '0';
                     f2 <= '1';
                when S0 =>
                     f1 <= '1';
                     f2 <= '1';
           end case;
      end process;
end Behavioral;
```