Stałe - *constant*

- Pojedyncze wartości zadeklarowanego typu
- Ustawiane przed rozpoczęciem symulacji bez możliwości późniejszych zmian
- Deklarowane w ciele architektury
- Widoczne dla całej architektury

```
architecture test_arch of circ is
      constant pi:real:=3.14159;
begin
  ...
  ...
  ...
end test_arch;
```


Zmienne i sygnały

- Wykorzystywane do przenoszenia danych w obrębie elementów
- Traktowane różnie podczas symulacji i syntezy
- Sygnały traktowane są jako połączenia („druty” - *wires*)
- Zmienne i stałe traktowana analogicznie do odpowiedników w językach programowania, wykorzystywane do modelowania zachowania systemu

Zmienne

- Deklaracja wewnątrz sekwencyjnych elementów VHDL (*process*, *procedure*, *function*)
  - Deklaracyjna część konstrukcji *process*
  - Dostępność ograniczona do procesu
  - Zasięg lokalny – brak możliwości deklaracji umożliwiającej dostęp z więcej niż jednego procesu
- Wartości początkowe
  - Zależne od typu zmiennej
    - przy braku inicjalizacji wartość początkowa przyjęta jako najmniejsza wartość typu zmiennej
    - dla typu tablicowego – poszczególne elementy przyjmują wartości jak w punkcie poprzednim

Zmienne

- Zmiana wartości realizowana za pomocą operatora przypisania zmiennej w trakcie symulacji
- Przypisanie zmiennej natychmiast zastępuje jej dotychczasową wartość

```
variable iloczyn: integer := 0;
```

```
process is
      variable zmienna: real;
begin
      ..........
end process;
```

```
iloczyn := iloczyn * 2;
zmienna := 4.332;
```

Uaktualnianie zmiennych

```
process (x) is
      variable licznik_x: integer;
begin
      licznik_a:=licznik_a + 1;
end process;
```

Jeśli zachodzą zdarzenia dotyczące sygnału *x* aktywujące proces zmienna licznik_a jest inkrementowana. Po zakończeniu symulacji zmienna licznik_a zawiera łączną liczbę zdarzeń na sygnale a.

```
signal a,b:integer;
process (x) is
      variable z1,z2: integer;
begin
      z1 := a - z2;
      b <= -z1;
      z2 := b+z1*5;
end process;
```

Jeśli w pierwszym przebiegu procesu z2 otrzyma wartość 2, to z2=2 zostanie wykorzystana do wyznaczenia z1 w pierwszej instrukcji procesu

5

Sygnały

- Reprezentacja portów I/O jednostki projektowej oraz wewnętrznych sygnałów architektury
- Zakres:
  - Sygnał I/O obowiązuje dla wszystkich architektur w ramach jednostki projektowej
  - Sygnały wewnętrzne – ciało architektury
    - Deklaracja w części deklaracyjnej architektury
    - Możliwość inicjalizacji w trakcie deklarowania

6

Przypisanie sygnałów

- Wartość sygnału wyznaczona po obliczeniu wartości wyrażenia (wynik podlega kontroli typów)
- Zależności czasowe
  - Przypisanie sygnału może zostać przesunięte o określony interwał czasowy

```
signal WR: integer;
```

```
WR := dane_we after 20 ns;
```

```
architecure test of rejestr is
       signal WR:integer:=0;
       .........
begin
       process(dane)
              begin
              ..........
              WR<=dane;
```

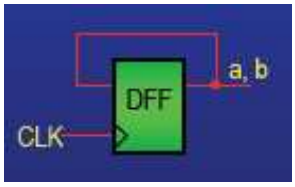

Zmienne vs. Sygnały

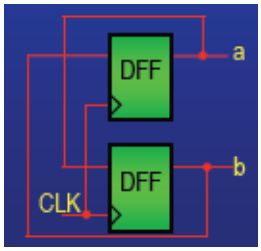
- Składnia VHDL:
  - Zmienne: ograniczony zakres (process)
  - Sygnały: większy zasięg (moduł)
- Symulacja
  - Zmienne: przypisanie natychmiastowe
  - Sygnały: przygotowanie do wykonania w następnym cyklu symulacji, uaktualniane tylko raz przy końcu procesu
- Synteza
  - Zmienne: połączenia („druty”) w netliście
  - Sygnały: w ramach składni reagującej na sygnał zegarowy syntezowane do elementów pamięciowych

Zmienne vs. Sygnały

```
process (CLK)
   variable a, b: std_logic;
begin
   if (CLK'event and CLK = '1') then
      a := b;
      b := a;
   end if;
end process;
```

```
signal a, b : std_logic;
process (CLK)
begin
   if (CLK'event and CLK = '1') then
      a <= b;
      b <= a;
   end if;
end process;
```





9

Zmienne vs. Sygnały – przykład

```
entity wrong_parity is
   port(in1: in std_logic_vector(3 downto 0);
        parity_out: out std_logic);
end wrong_parity;

architecture examp1 of wrong_parity is
signal parity_tmp: std_logic;

begin
   process(in1, parity_tmp)
      begin
         parity_tmp <= '0';
         for i in in1'range loop
            parity_tmp <= parity_tmp xor in1(i);
         end loop;
   end process;
   parity_out <= parity_tmp;
end examp1;
```

10

Zmienne vs. Sygnały – analiza przykładu

- Sygnały uaktualniane przy końcu procesu
- Dla sygnału ma miejsce tylko jedno przypisanie
- Po ostatnim rozwinięciu pętli, ostatnie przypisanie dotyczy niezdefiniowanej wartości
- Sytuacja powtarza się przy każdym wywołaniu procesu
- Inicjalizacja sygnału parity_tmp powoduje nieskończoną pętlę (parity_tmp) występuje w liście czułości procesu
- Każda zmiana parity_tmp powoduje kolejne wywołanie procesu

Zmienne vs. Sygnały – synteza przykładu

- Sygnał parity_tmp traktowany jest jak połączenie
- parity_tmp<=parity_tmp XOR in1 powoduje sprzężenie wokół bramki XOR; zamiast układu kombinacyjnego mamy sekwencyjny

Zmienne vs. Sygnały – przykład poprawny

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

entity correct_parity is
   port(in1: in std_logic_vector(3 downto 0);
        parity_out: out std_logic);
end correct_parity;

architecture examp2 of correct_parity is

begin
   process(in1)
   variable parity_var: std_logic;
      begin
         parity_var := '0';
         for i in in1'range loop
            parity_var := parity_var xor in1(i);
         end loop;
         parity_out <= parity_var;
   end process;
end examp2;
```

13

Inicjalizacja zmiennych

- Inicjalizacja jeśli jest wymagana, przeprowadzana tylko raz
- Narzędzia syntezy ignorują inicjalizację zmiennych i sygnałów w części deklaracyjnej
- Inicjalizacja może być wykonana za pomocą przypisania sygnałów lub zmiennych
- Zachowanie wartości zmiennych niezależnie od aktywacji i zatrzymania procesu (cecha przydatna przy tworzeniu pamięci ROM)

14

Zmienne w procesach zawierających sygnał taktujący

- Zmienne i sygnały traktowane są odmiennie przez narzędzia syntezy
- Przypisanie wartości sygnałom odpowiada wykorzystaniu przerzutnika

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

entity reg_ex is
    port(din: in std_logic;
         clk: in std_logic;
         q_out, q_out_bar: out std_logic);
end reg_ex;

architecture exampl of reg_ex is

begin
    process(clk)
    variable q_tmp: std_logic;
       begin
          if (clk'event and clk = '1') then
             q_tmp := din;
          end if;
          q_out <= q_tmp;
          q_out_bar <= not(q_tmp);
    end process;
end exampl;
```


Przykład mieszany

```
Library IEEE;
Use std_logic_1164.all;

entity sub_4bit is
    port( a_in, b_in: in std_logic_vector(3 downto 0);
          . . . . . );
end sub_4bit;

architecture mix_struc_behav of sub_4bit is
signal b_in_invers: std_logic_vector(3 downto 0);

component adder_4bit
      port(a, b: in std_logic_vector(3 downto 0);
           . . . . . .
begin
    process(b_in)   begin
           b_in_invers <= not(b_in);
     end process;

    U0: adder_4bit
         port map(a => a_in, b => b_in_invert,
    . . . . . .
```

Synteza VHDL

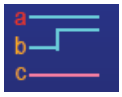
- Pierwotne zastosowanie VHDL - symulacja
- Narzędzia syntezy automatycznie decydują o wykorzystaniu standardowych bloków cyfrowych (przerzutniki, rejestry trójstanowe, logika kombinacyjna)
- Niektóre elementy składniowe działają poprawnie tylko w symulacji i są ignorowane przy syntezie

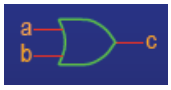
Wartości początkowe: trudności w realizacji w sprzęcie
Wyrażenia czasowe: jw.
Lista czułości procesu: tylko symulacja

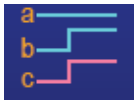
Synteza VHDL

```
process (x) is
begin
       c<=a OR b;
end process;
```

Symulacja przed syntezą | Synteza | Weryfikacja po syntezie



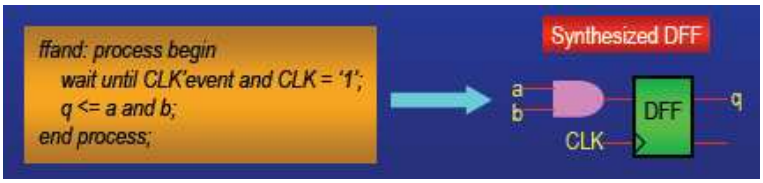




Wykorzystanie w syntezie przerzutników

- Przerzutniki wymagają sygnału taktującego, operują na jego zboczach
- Zbocza sygnału modelowane następująco:
  CLK'event and CLK='1' - zbocze narastające
  CLK'event and CLK='0' - zbocze opadające

- Sygnały i zmienne wewnątrz bloku zamkniętego zegarem są traktowane jako przerzutniki



19

Wykorzystanie w syntezie przerzutników

- Należy unikać bramkowania oraz wewnętrznego generowania sygnału zegarowego

  Sygnał zegarowy jest silnie zależny od technologii i jego bramkowanie może powodować zakłócenia

20

Resetowanie przerzutników

- **Synchroniczny RESET**

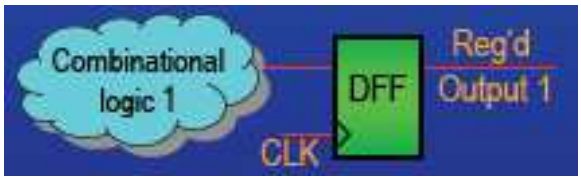
```
process (CLK) begin
  if CLK'event and CLK = '1' then
    if sRST = '1' then
      . . .
  end if; end if;
end process;
```

- **Asynchroniczny RESET**

```
process (CLK, sRST) begin
  if aRST = '1' then
      . . .   -- reset;
  end if;
end process;
```


Wyjścia rejestrowe

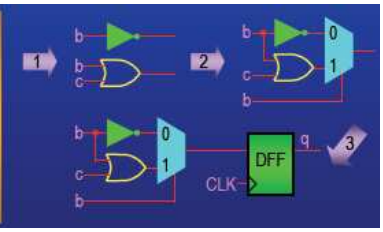
- **Wyjścia determinowane przez logikę kombinacyjną**
- **Zalety wyjścia rejestrowego:**
  - **Przewidywalne opóźnienia**
  - **Jednoczesna zmiana wyjść**



Logika kombinacyjna

- Logika kombinacyjna w syntezie jest wynikiem zastosowania wyrażeń boolowskich
- Instrukcje warunkowe determinują zastosowanie multiplekserów

```
process (CLK) begin
  if (CLK'event and CLK = '1') then
    if (a = '1') then
      d <= b or c;
    else
      d <= not b;
    end if;
  end if;
end process;
```

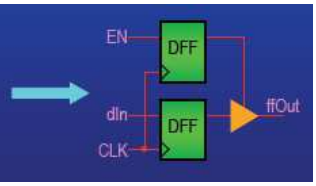


23

Połączenie buforów i przerzutników

- Konstrukcja wyjść uwzględniająca bufory trójstanowe i rejestry może wykorzystywać rejestrowe wyjście bramkujące bufor

```
process (EN, CLK) begin
  if EN = '0' then
    ffOut <= 'Z';
  elsif (CLK'event and CLK = '1' )then
    ffOut <= dIn;
  end if;
end process;
```



24

Modelowanie behawioralne

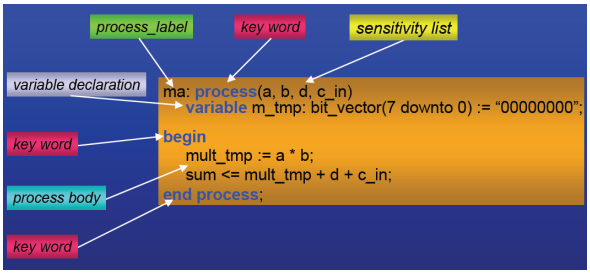
Architektura behawioralna

- Opisuje funkcjonalność w sposób abstrakcyjny
- Zawiera składnię *process*
  - Zbiór instrukcji wykonywanych sekwencyjnie w przeciwieństwie do instrukcji poza składnią *process*
  - Analogia do języków programowania

- Składnia sekwencyjna zawiera obliczenie wyrażenia, przypisanie wartości zmiennym i sygnałom, instrukcje warunkowe, wywołanie podprogramów

Proces

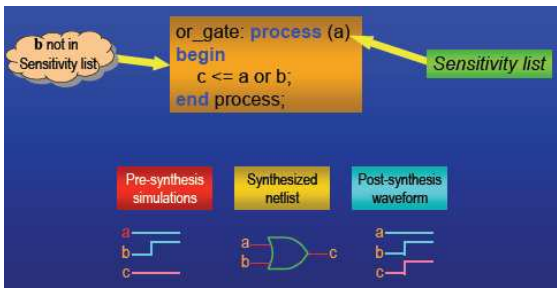
- Typowy element składni VHDL
- Wykorzystywana do lepszej wizualizacji zmian zachodzących w projekcie
- Elementy procesu wykonywane są sekwencyjnie
- Ważna jest kolejność kodowania



27

Lista czułości procesu

- Lista sygnałów, na które proces „reaguje"
- Zdarzenie na dowolnym sygnale z listy czułości powoduje uruchomienie („wykonanie") procesu
- Instrukcje w procesie wykonywane są sekwencyjnie
- Po wykonaniu instrukcji proces jest wstrzymywany do momentu wystąpienia kolejnego zdarzenia z listy

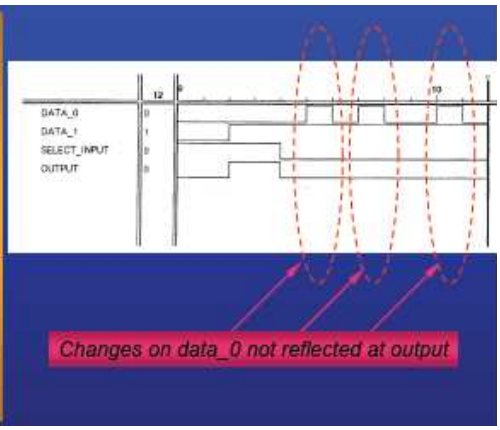


28

Lista czułości procesu

```
entity mux is
   port(data_0,
        data_1, sel: in std_logic;
        output: out std_logic);
end mux;

architecture incom of mux is
begin
  process(data_1, sel)
    begin
      if (sel = '0') then
        output <= data_0;
      else
        output <= data_1;
  end process;
end incom;
```



29

Lista czułości procesu i synteza

- Proces jest brany pod uwagę w symulacji
- Niepoprawna symulacja z powodu braku prawidłowej listy czułości jest przezroczysta z punktu widzenia sytnezowanego układu

- Kod zawierający błędy symulacji może zostać zsyntezowany do poprawnego układu. Może to jednak stwarzać problemy z weryfikacją

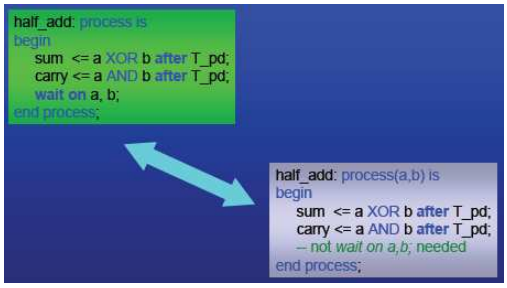
30

Lista czułości procesu i instrukcja WAIT

- Lista czułości i WAIT – wzajemne wykluczenie
- WAIT – alternatywna metoda zawieszania procesu

```
wait on sensivity list;
wait until sensivity list;
wait for sensivity list;

wait on sensivity_list until boolean_expr for time_expr;
```



Generowanie sygnału zegarowego

- Tylko symulacja (testbench)

Brak wejścia do procesu

```
entity clk is
    port(clk: out std_logic);
end clk;

architecture cyk of clk is
begin
  process
      begin
          clk <= '0';
          wait for 10 ns;
          clk <= '1';
          wait for 10 ns;
    end process;
end cyk;
```

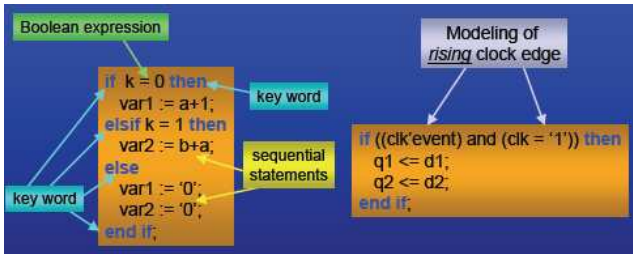
Dopuszczalne instrukcje

- if
- case
- wait
- pętla (for, while, loop)

- Instrukcje współbieżne (when, with) są zabronione wewnątrz procesów

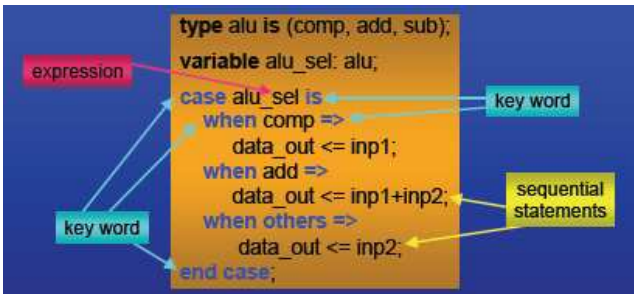
If

```
if boolean-expression then
   sequential-statements
{elsif boolean-expression then  -- elsif clause; if can have 0 or
   sequential-statements}       -- more elsif clauses
[else
   sequential-statements]
end if;
```



Case

```
Case expression is
   when choices => sequential-statements   -- branch 1
   when choices => sequential-statements   -- branch 2
   -- any number of branches can be specified
   [when others => sequential-statements]   -- last branch
end case;
```



35

If czy Case?

- **If/else – jeśli zachowanie układu zależy od więcej niż jednego sygnału**
- **Case – jeśli zachowanie układu może zależeć od jednego sygnału**
- **If/else syntezuje się do kaskadowych multiplekserów**
- **Case syntezuje się do pojedynczego multipleksera**

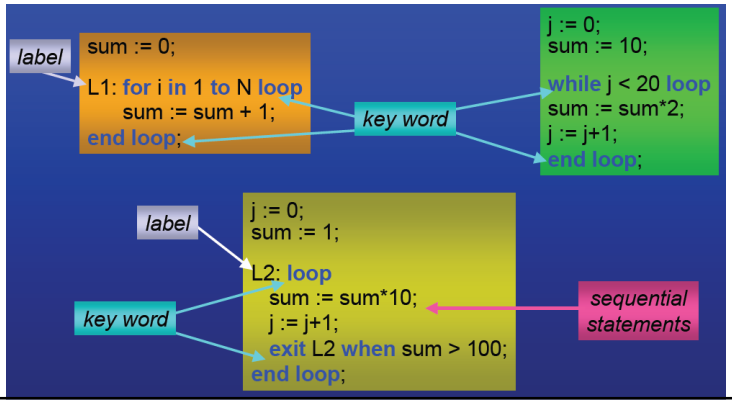
- **If/else – jeśi istotna jest powierzchnia układu (układ mały i wolny)**
- **Case – jeśli szybkość jest istotna (układ duży i szybki)**

36

Ifnstrukcje pętli

- **Wyklorzystywane do iterowania zestawu instrukcji sekwencyjnych**

```
[loop label: ] iteration-scheme loop
    sequential-statements
end loop [loop label];
```



For

- **Przykład: obliczenie y^n, N>2**

```
power_N := y;
for iter in 2 to N loop
   power_N := power_N * y;
end loop;
```

- **Pętla wykonuje się N-1 razy**
- **Inkrementacja licznika przy końcu pętli**
- **Brak konieczności bezpośredniej deklaracji licznika pętli**
- **Zakres może być całkowity lub wyliczeniowy**

While

- **Przykład: obliczenie y^n, N>2**

```
power_N := y;
while i < N loop
   power_N := power_N * y;
end loop;
```

- **Zakończenie pętli, gdy warunek przyjmuje wartość false**

Loop

- **Brak schematu iteracji, wyjście z pętli, gdy wystąpi warunek jej zakończenia**

```
power_N := y;
L: loop
   power_N := power_N * y;
   exit when power_N > 250;
end loop L;
```

39

Komunikacja między procesami

```
    signal address: integer;
    signal mem_read: word;
    signal mem_write: word;
    signal data_read: std_logic;
    signal data_write: std_logic;
    signal mem_ready: std_logic;
begin
    cpu: process
         variable PC: integer;
         variable instr_reg: word;
    begin
      loop
         address <= PC;
         data_read <= '1';
         wait until mem_ready = '1';
         instr_reg := mem_read;
         wait until mem_ready = '0';
         PC:= PC+2;
      end loop;
    end process cpu;

    memory: process
      type data_type is array (0 to 63) of word;
      variable store: data_type;
    begin
      wait until data_read = '1' or data_write = '1';
      if data_read = '1' then
        data_read <= store(PC/2);
        data_out <= store(PC/2);
        mem_ready <= '1';
        wait until data_read = '0';
        mem_ready <= '0';
     elsif data_write = '1' then
        store(PC/2) <= data_in;
        mem_ready <= '1';
        wait until data_write = '0';
        mem_ready <= '0';
      end if;
   end process memory;
```

40