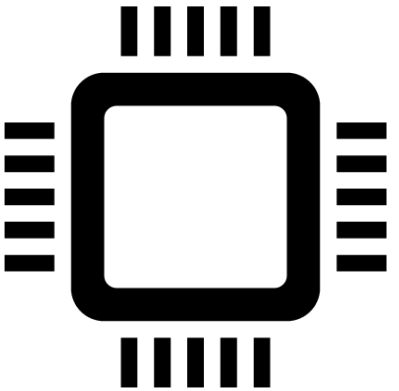
Programowanie struktur cyfrowych



Język VHDL.
Instrukcje współbieżne

dr Aleksander Lamża

Uniwersytet J. Kochanowskiego w Kielcach
Uniwersytet Śląski w Katowicach

aleksander.lamza@us.edu.pl

Instrukcje współbieżne

Poprzedni wykład był poświęcony instrukcjom sekwencyjnym, czyli takim, które są wykonywane jedna po drugiej, w kolejności, w jakiej występują w danym bloku kodu.

W tej prezentacji znajduje się przegląd instrukcji współbieżnych.

Wykonanie instrukcji współbieżnych następuje asynchronicznie. Język nie definiuje kolejności, w której są wykonywane.
Podstawową instrukcją jest **instrukcja procesu** reprezentująca niezależny sekwencyjny proces.

Przegląd instrukcji współbieżnych

W VHDL-u można wyróżnić kilka instrukcji współbieżnych.

W prezentacji zostaną opisane cztery, wybrane instrukcje:

- instrukcja procesu,
- instrukcje przypisania do sygnału,
- instrukcje konkretyzacji składników,
- instrukcja powielania (generowania).

Instrukcja procesu

Tę instrukcję już dobrze znamy. Służy ona do zdefiniowania niezależnego bloku kodu sekwencyjnego, czyli procesu reprezentującego wybranej części projektu.

Składnia instrukcji jest następująca:

```
instrukcja_procesu ::= [etykieta:]
    [postponed] process [(lista_czułości)] [is]
       część_deklaracyjna
    begin
       instrukcje_sekwencyjne
    end [postponed] process [etykieta];
```

Instrukcja procesu

Temat **listy czułości** już się pojawił na poprzednim wykładzie.
Dla przypomnienia – chodzi o wyzwalanie wznawiania procesu.
Jeżeli na liście czułości umieścimy jakiś sygnał, proces zostanie wznowiony po zmianie jego stanu.

Przykładowy proces:

```
process (a, b)
begin
    x <= a and b;
end process;
```

zostanie wznowiony po zmianie co najmniej jednego sygnału: `a` lub `b`.

Instrukcja procesu

W **części deklaracyjnej** można umieścić deklaracje i definicje typów, podtypów, stałych i zmiennych używanych w procesie. Co ważne, nie można tu deklarować sygnałów.

Oto kilka przykładów:

```
process
    type byte is range 0 to 255;
    subtype nibble is range 0 to 15;
    constant eof: byte := 0;
    variable a: nibble;
begin
    …
end process;
```

Instrukcja procesu

Wyjaśnienia wymaga opcjonalne słowo kluczowe **`postponed`**, które zostało wprowadzone w wersji VHDL 93. Wpływa ono na moment wznowienia wykonywania procesu.

Słowo „postponed” oznacza po polsku „przełożony”, „odroczony”. A co jest odraczane? Wykonanie procesu po jego wznowieniu. Ale po kolei...

Jeżeli w definicji procesu nie umieścimy słowa `postponed`, proces zostanie wykonany w tym samym cyklu symulacji, w którym został wznowiony.
W związku z tym będą brane pod uwagę bieżące wartości sygnałów, być może jeszcze nie do końca stabilne.

Zastosowanie słowa kluczowego `postponed` zmienia sposób wznawiania. Po wznowieniu (wyzwolonym listą czułości lub instrukcją oczekiwania) proces czeka na zakończenie bieżącego cyklu symulacyjnego i zostaje uruchomiony dopiero w kolejnym. Dzięki temu ma dostęp do ustalonych, stabilnych wartości sygnałów.

Instrukcja przypisania do sygnału

Współbieżna instrukcja przypisania do sygnału służy do zmiany wartości sygnału. Składnia jest następująca:

```
przypisanie_sygnału ::= [etykieta:][postponed]
        podstawowe_przypisanie_sygnału
    |   warunkowe_przypisanie_sygnału
    |   selektywne_przypisanie_sygnału
```

Mamy więc trzy formy przypisania: podstawowe, warunkowe i selektywne.

Przypisanie podstawowe wygląda następująco:

```
podstawowe_przypisanie_sygnału ::=
    cel_przypisania <= wyrażenie
```

Instrukcja przypisania do sygnału

Przykładem niech będzie jakiś układ kombinacyjny:

```
entity Foo is
    port (
        a, b: in bit;
        y1, y2: out bit;
    )
end entity;

architecture Foo_arch of Foo is
begin
    y1 <= a xor b after 5 ns;
    y2 <= y1 and a after 5 ns;
end;
```

Należy pamiętać, że instrukcje przypisania współbieżnego są wykonywane równolegle, czyli kolejność ich umieszczenia w kodzie nie ma znaczenia.

Instrukcja przypisania do sygnału

Instrukcja warunkowego przypisania dokonuje aktualizacji sygnału na podstawie zestawu warunków logicznych.

Każdy warunek jest powiązany z jedną charakterystyką przypisania:

```
warunkowe_przypisanie ::=
   cel_przypisania <=
    { charakterystyka_przypisania when warunek else }
      charakterystyka_przypisania when warunek;
```

Wszystkie warunki są sprawdzane do momentu znalezienia spełnionego warunku. Wtedy do sygnału zostaje przypisana wartość wynikająca z charakterystyki przypisania powiązanej ze spełnionym warunkiem.

Instrukcja przypisania do sygnału

Przykład:

```
type operation = (ADD, SUB, MUL, DIV);

...

architecture ALU_arch of ALU is
begin
    y <= a + b after 5 ns when op = ADD else
         a - b after 5 ns when op = SUB else
         a * b after 5 ns when op = MUL else
         a / b after 5 ns when op = DIV;
end architecture;
```

Instrukcja ta odpowiada sekwencyjnej instrukcji warunkowej:

```
if (op = ADD) then
    y <= a + b after 5 ns
elseif (op = SUB) then
    y <= a - b after 5 ns
elseif ...
```

Instrukcja przypisania do sygnału

Podobna do instrukcji warunkowej jest instrukcja selektywnego przypisania. Różnica polega na tym, że w pierwszym przypadku o wyborze decyduje warunek logiczny, a w drugim **wartość**.

Składnia jest następująca:

```
selektywne_przypisanie ::=
   with wyrażenie select
      cel_przypisania <=
       { charakterystyka_przypisania when wyrażenie, }
         charakterystyka_przypisania when wyrażenie;
```

Należy pamiętać o tym, że zestaw wyrażeń wyboru musi być kompletny, tzn. muszą być uwzględnione wszystkie przypadki.

W spełnieniu tego warunku pomaga słowo kluczowe **`others`** reprezentujące pozostałe, nieuwzględnione wcześniej warunki.

Instrukcja przypisania do sygnału

Przykład:

```
entity DISP_7SEG is
    port (
        bcd: in bit_vector(3 downto 0);
        seg: out bit_vector(6 downto 0)
    );
end DISP_7SEG;

architecture arch of DISP_7SEG is
begin
    with bcd select
        seg <= "1111110" after 5 ns when "0000",
               "0110000" after 5 ns when "0001",
               "1101101" after 5 ns when "0010",
               …
               "1111011" after 5 ns when "1001",
               "0000000" after 5 ns when others,
end;
```

Instrukcja konkretyzacji składnika

Instrukcja konkretyzacji składnika umożliwia tworzenie projektu w postaci strukturalnego opisu jednostki projektowej.

Składnia tej instrukcji jest następująca:

```
instrukcja_konkretyzacji ::=
   etykieta: jednostka_konkretyzacji
             [wartości_parametrów]
             [porty];

jednostka_konkretyzacji ::=
      entity nazwa_jednostki [(nazwa_ciała_jednostki)]
   | [component] nazwa_składnika
   | configuration nazwa_konfiguracji
```

Instrukcja konkretyzacji zawiera jednostkę konkretyzacji, wartości parametrów i połączenia portów.

Jednostka konkretyzacji reprezentuje część projektu, która jest definiowana oddzielnie.

Instrukcja konkretyzacji składnika

Najlepiej będzie to zilustrować przykładami.

```
counter: entity lib.counter_bin(counter_arch)
            port map (a, b, c);
```

Instrukcja konkretyzacji tworzy jednostkę projektową `counter_bin` z ciałem architektonicznym `counter_arch`. Zarówno jednostka, jak i architektura znajdują się w bibliotece `lib`.

Dochodzi tu również do powiązania portów jednostki z sygnałami `a`, `b` i `c`.

W tym przykładzie wskazujemy wcześniej zdefiniowaną jednostkę projektową (`counter_bin`).

Jest to przykład tzw. konkretyzacja bezpośrednia.

Instrukcja konkretyzacji składnika

Jeśli instrukcja konkretyzacji składnika zawiera nazwę **komponentu**, instrukcja musi być poprzedzona deklaracją tego komponentu.

Deklaracja składnika definiuje raczej wirtualny interfejs jednostki projektowej, a nie wskazuje bezpośrednio określonej jednostki. Połączenie jednostki z komponentem można zrealizować później, np. deklaracji konfiguracji.

Deklaracja komponentu wygląda następująco:

```
component id [is]
    [sekcja_parametrów]
    [sekcja_portów]
end component [id];
```

Komponent można zdefiniować w pakiecie, deklaracji jednostki projektowej, deklaracji architektury bądź deklaracji bloku.

Instrukcja konkretyzacji składnika

Za przykład może posłużyć konkretyzacja dwuwejściowej bramki XOR:

```
architecture struct_arch of Adder is

    -- deklaracja komponentu
    component xor_gate is
        port (
            a, b: in bit;
            y: out bit
        )
    end component xor_gate;

    signal s1, s2, s3: bit;

begin
    -- tworzenie instancji komponentu
    g1: xor_gate port map(s1, s2, s3);
end struct_arch;
```

Instrukcja powielania (generowania)

Instrukcja powielania ułatwia definiowanie powtarzających się struktur projektowych. Składnia instrukcji jest następująca:

```
etykieta: schemat_powielania generate
             [sekcja_deklaracyjna
          begin]
             instrukcje_współbieżne
          end generate [etykieta];
```

Schemat powielania określa sposób generowania struktury instrukcji współbieżnych. Istnieją dwa schematy: iteracyjny (pętlowy) i warunkowy:

```
schemat_powielania ::=
     for id in zakres
   | if warunek
```

Instrukcja powielania (generowania)

Za przykład wykorzystania instrukcji powielania w schemacie iteracyjnym może posłużyć wygenerowania przerzutników na potrzeby licznika.

Zakładamy, że D_FF jest jednostką projektową ze zdefiniowaną architekturą. Deklaracja 4-bitowego licznika będzie wyglądała następująco:

```
entity BIN_COUNTER_4 is
   port (Q: out bit_vector(0 to 3); C: in bit);
end entity;

architecture ARCH of BIN_COUNTER_4 is
   component D_FF
      port (D, CLK: in bit; Q, NQ: out bit);
   end component;

   signal s: bit_vector(0 to 3);
begin
   s(0) <= C;
   g: for i in 0 to 3 generate
      d_ff_i: D_FF port map (s(i+1), s(i), q(i), s(i+1));
   end generate;
end architecture;
```

Instrukcja powielania (generowania)

Instrukcję powielania w schemacie warunkowym najczęściej zagnieżdża się w instrukcji powielania w pętli.

Fragment kodu ilustrujący zastosowanie tej instrukcji:

```
g1: for i in 0 to 3 generate
    ...
    g2: if i=1 or i=2 generate
        ...
    end generate g2;
end generate g1;
```

Daje to możliwość budowania złożonych powtarzalnych struktur.

Podsumowanie

Można uznać, że instrukcje współbieżne są nadrzędne względem instrukcji sekwencyjnych. Przede wszystkim natura układów cyfrowych jest współbieżna, więc musi to znaleźć odzwierciedlenie w języku i stosowanych konstrukcjach.

Projektowanie układów cyfrowych w VHDL-u polega na zgrabnym łączeniu elementów współbieżnych z sekwencyjnymi.

W przedstawionych prezentacjach starałem się nakreślić zarys tematyki. Umiejętności w tym zakresie należy zdobywać – tak jak w „zwykłych” językach programowania – w praktyce.

Wszystkich zainteresowanych odsyłam do licznych źródeł internetowych poświęconych układom programowalnym, językowi VHDL, a także innym językom modelowania struktur cyfrowych (np. Verilogowi).