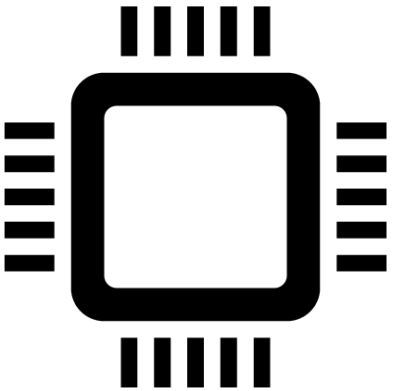
Programowanie struktur cyfrowych



Język VHDL.
Instrukcje sekwencyjne

dr Aleksander Lamża

Uniwersytet J. Kochanowskiego w Kielcach
Uniwersytet Śląski w Katowicach

aleksander.lamza@us.edu.pl

Instrukcje sekwencyjne i współbieżne

Modelowanie struktury i zachowania układów cyfrowych w swojej naturze wymaga podejścia **współbieżnego**. Instrukcje tego typu są wykonywane asynchronicznie. Co istotne, język VHDL w żaden sposób nie definiuje kolejności, w jakiej są wykonywane.

Z kolei algorytmy procesów i podprogramów są definiowane przy pomocy instrukcji **sekwencyjnych**. Oznacza to, że są one wykonywane w kolejności, w jakiej zostały zapisane.

Pełną elastyczność i funkcjonalność układów programowalnych uzyskuje się dzięki umiejętnemu połączeniu obu podejść: współbieżnego i sekwencyjnego.

Przegląd instrukcji sekwencyjnych

W języku VHDL mamy do dyspozycji wiele instrukcji sekwencyjnych:

- instrukcja przypisania do zmiennej (`:=`)
- instrukcja przypisania do sygnału (`<=`)
- instrukcja oczekiwania (`wait`)
- instrukcja warunkowa (`if-then`),
- instrukcja wyboru (`case`),
- instrukcja pętli (`loop`),
- instrukcja wyjścia z pętli (`exit`),
- instrukcja kolejnej iteracji (`next`),
- instrukcje raportu (`report`) i asercji (`assert`),
- instrukcja pusta (`null`).

Instrukcja przypisania do zmiennej

Instrukcja ta, podobnie jak jej odpowiedniki w innych językach, służy do modyfikowania wartości zmiennej.

Jej składnia wygląda następująco:

```
instrukcja_przypisania_zmiennej ::= [etykieta:]
                    cel := wyrażenie;
```

Po lewej stronie przypisania znajduje się cel przypisania, a po prawej wyrażenie, którego wartość zostaje przypisana do celu.

Co istotne, typ wyrażenia musi być taki sam, jak typ zmiennej.

W podstawowej formie przypisanie wygląda np. tak:

```
variable a: Integer;
…
a := 13;
…
a := a + 1;
```

Instrukcja przypisania do zmiennej

Na tym nie kończą się jednak możliwości instrukcji przypisania.

W przypisaniu w VHDL-u możemy też indeksować zmienne, ustalać ich zakres i wskazywać składniki.

Jeżeli mamy np. 8-bitową zmienną typu `bit_vector`:

```
variable a: bit_vector(0 to 7);
```

możemy przypisać wartość do wybranego bitu:

```
a(3) := '1';
```

albo do wybranego zakresu bitów:

```
a(0 to 3) := ('1', '0', '0', '1');
```

Oczywiście indeksowanie można zastosować również po prawej stronie:

```
a(0 to 3) := a(4 to 7)
```

Instrukcja przypisania do zmiennej

Jeżeli mamy do czynienia z typem rekordowym, możemy przypisywać wartości do poszczególnych elementów rekordu.

Przypuśćmy, że zadeklarowaliśmy typ rekordowy i zmienną:

```
type foo_type is record
  a: bit;
  b: integer;
end record;

variable f: foo_type;
```

Zmiany wartości składników rekordu dokonuje się za pomocą notacji „kropkowej” (podobnie jak w wielu innych językach):

```
f.a := '0';

f.b := 13;
```

Instrukcja przypisania do zmiennej

W VHDL-u istnieje jeszcze jedna możliwość przypisywania wartości – do **agregatu zmiennych**.

Agregat jest złączeniem kilku zmiennych. Oto kilka przykładów:

```
variable a, b: bit_vector(3 downto 0);
variable c: bit;

(c, a) := a + b
```

W tym przypadku do zmiennej `c` zostanie przypisana wartość przeniesienia w tej 4-bitowej operacji sumowania.

Instrukcja przypisania do sygnału

W języku modelowania sprzętu kluczowa jest możliwość wpływania na **sygnały**, czyli wartości dostępne „fizycznie” (np. na pinach układu FPGA).

Właśnie ze względu na tę „fizyczność”, możliwości instrukcji przypisania do sygnału są znacznie bogatsze niż instrukcji przypisania do zmiennej.

Składnia wygląda następująco:

```
instrukcja_przypisania_sygnału ::= [etykieta:]
    cel <= [opóźnienie] kształt;

kształt ::= element_k { , element_k}
                    | unaffected

element_k ::= wyrażenie [after wyrażenie_czasowe]
                    | null [after wyrażenie_czasowe]

opóźnienie ::= transport
                    | [reject wyrażenie_czasowe] inertial
```

Instrukcja przypisania do sygnału

Jak widać, składnia jest dosyć złożona. Poszczególne elementy najlepiej będzie opisać na przykładach.

Podstawowa postać przypisania zawiera **cel** (sygnał) i co najmniej jeden element **kształtu sygnału**:

```
signal q: bit;

q <= '1';
```

Spowoduje to przypisanie stanu wysokiego do sygnału `q`.

Najczęściej definiuje się w takich sytuacjach **czas**, po jakim stan ma zostać przypisany:

```
q <= '1' after 2 ns;
```

Jeżeli pominiemy czas opóźnienia, w trakcie symulacji zakładany jest zerowy czas reakcji.

Instrukcja przypisania do sygnału

Z definicji

```
kształt ::= element_k { , element_k}
```

jasno wynika, że kształt sygnału można definiować za pomocą sekwencji elementów. Oto przykład:

```
q <= '0' after 2 ns, '1' after 10 ns;
```

Sygnał `q` przyjmie stan niski po 2 ns, a po kolejnych 8 ns (10 ns od bieżącego czasu) zmieni się na stan wysoki.

Instrukcja przypisania do sygnału

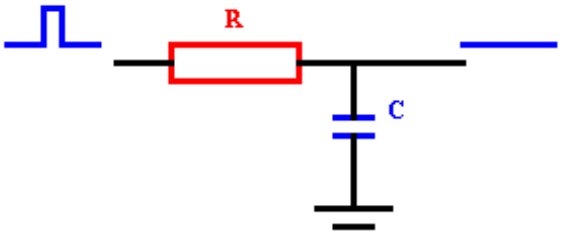
W definicji przypisania do sygnału występuje opcjonalny parametr typu opóźnienia:

```
cel <= [opóźnienie] kształt
```

W VHDL-u występują dwa typu opóźnień: **inercyjne** i **transportowe**.

Opóźnienie transportowe jest związane z przenoszeniem sygnałów w liniach transmisyjnych i występuje w specyficznych sytuacjach.

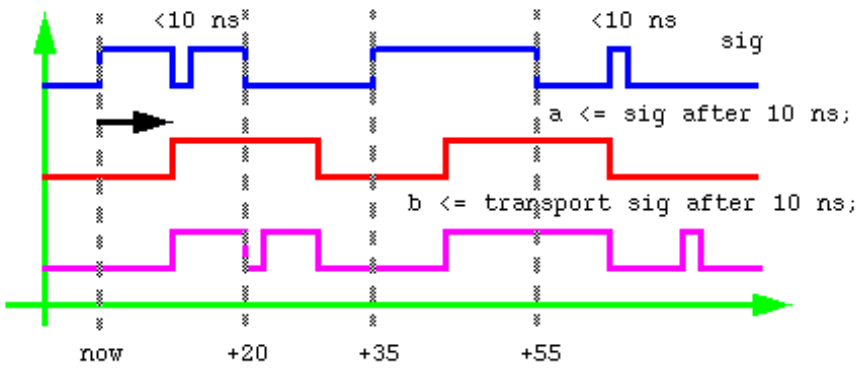
W rzeczywistych układach cyfrowych najczęściej występują opóźnienia związane z bezwładnością (inercją) fizycznego układu elektronicznego. Wynikają one przede wszystkim z pojemności w tych układach:



Kilka rysunków w tej prezentacji pożyczyłem ze strony http://www.pldworld.com/_hdl/1/www.ireste.fr/fdl/vcl/lesd/les_4.htm

Instrukcja przypisania do sygnału

Różnice między typami opóźnień najlepiej będzie zilustrować przykładem:



Oryginalny sygnał (sig) jest przypisywany do sygnału a z domyślnym opóźnieniem inercyjnym, a do sygnału b z opóźnieniem transportowym. W obu przypadkach czas opóźnienia to 10 ns.

Jak widać, „piki” o czasie trwania poniżej 10 ns są eliminowane w wyniku zadziałania opóźnienia inercyjnego, a w opóźnieniu transportowym są przenoszone.

Należy też zwrócić uwagę, że w obu przypadkach dochodzi do jednakowego przesunięcia czasu odpowiedzi o 10 ns (czarna strzałka).

Instrukcja przypisania do sygnału

W przypadku inercyjnego modelu opóźnień mamy możliwość bardziej precyzyjnego zdefiniowania bezwładności. Służy do tego słowo kluczowe **`reject`**:

```
q <= reject 1 ns inertial '1' after 10 ns;
```

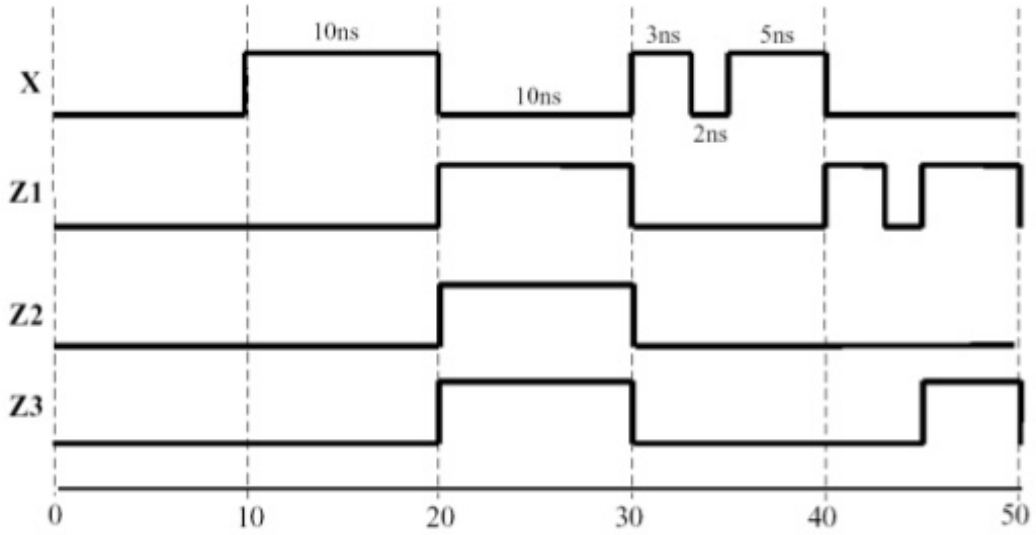
Opóźnienie przypisania sygnału jest równe 10 ns, ale wszystkie zmiany krótsze niż 1 ns są „wchłaniane” przez bezwładność.

W przykładach pojawiały się czasy wyrażane w nanosekundach. Mamy oczywiście możliwość zastosowania innych jednostek czasu zdefiniowanych w typie `Time`:

```
type Time is range
  units
     fs;              -- femtosekundy
     ps  = 1000 fs;   -- pikosekundy
     ns  = 1000 ps;   -- nanosekundy
     us  = 1000 ns;   -- mikrosekundy
     ms  = 1000 us;   -- milisekundy
     sec = 1000 ms;   -- sekundy
     min = 60  sec;   -- minuty
     hr  = 60  min;   -- godziny
  end units;
```

Instrukcja przypisania do sygnału

Na koniec ilustracja zestawiająca możliwości definiowania opóźnień:




```
Z1 <= transport X after 10 ns;    -- transport delay
Z2 <= X after 10 ns;              -- inertial delay
Z3 <= reject 4 ns X after 10 ns;  -- delay with specified rejection pulse width
```

Rysunek pochodzi z prezentacji https://www.slideshare.net/BhupendraPratapSingh1/behavioral-modelling-in-vhdl

Instrukcja oczekiwania

Instrukcja oczekiwania powoduje zawieszenie wykonywania procesu lub procedury. Wznowienie wykonywania jest zależne od spełnienia warunku.

Składnia instrukcji oczekiwania wygląda następująco:

```
instrukcja_oczekiwania ::= [etykieta:]
  wait [sekcja_warunku] [sekcja_czułości] [sekcja_czasu]

sekcja_warunku ::= until warunek

sekcja_czułości ::= on lista_czułości

lista_czułości ::= sygnał {, sygnał}

sekcja_czasu ::= for wyrażenie_czasowe
```

Instrukcja oczekiwania

Pierwszym sposobem określenia warunku wznowienia wykonywania (czyli przerwania oczekiwania) jest zdefiniowanie warunku logicznego, np.:

```
wait until enable = '1';
```

Instrukcja ta spowoduje zawieszenie wykonywania procesu (lub procedury) do momentu, gdy sygnał `enable` przyjmie stan wysoki.

Kolejną możliwością jest wskazanie sygnałów, których zmiana powoduje wznowienie wykonywania. Lista tych sygnałów to tzw. **lista czułości** (ang. *sensitivity*):

```
signal a, b: std_logic;

...

wait on a, b;
```

Co istotne, nawet jeśli nie określimy listy czułości, jest ona niejawnie tworzona. Na przykład dla instrukcji `wait until enable = '1'` na liście czułości znajduje się sygnał `enable`.

Instrukcja oczekiwania

Warunek czasu pozwala na wstrzymanie wykonywania procesu na ustalony okres, np.:

```
wait for 20 us;
```

Instrukcja ta zawiesi wykonywanie na czas 20 mikrosekund.

W jednej instrukcji `wait` można umieścić kilka różnych warunków.
Wznowienie procesu następuje po spełnieniu **wszystkich** warunków.

Przykładem ilustrującym połączenie warunków będzie dwuwejściowy komparator binarny sprawdzający równość dwóch stanów.
Ma to być układ synchroniczny zmieniający stan wyjścia przy wysokim stanie sygnału zegarowego.

Instrukcja oczekiwania

Przykładowy opis behawioralny takiego komparatora wygląda następująco:

```
comp: process
begin
  wait on a, b until clk = '1';
  if a = b then
    y <= '1' after 2 ns;
  else
    y <= '0' after 2 ns;
  end if;
end process;
```

Proces `comp` jest wznawiany przy zmianie stanu sygnałów `a`, `b` lub `clk`, ale pod warunkiem, że `clk` ma stan wysoki.

Instrukcja oczekiwania

Często stosuje się rozwiązanie polegające na umieszczeniu listy czułości w definicji procesu, np.:

```
xor_gate: process (a, b)
begin
    y <= a xor b;
end process;
```

W procesie kompilacji na końcu procesu jest wstawiana niejawna instrukcja oczekiwania na wskazane sygnały.

```
xor_gate: process (a, b)
begin
    y <= a xor b;
    wait on a, b; -- to jest wstawiane przez kompilator
end process;
```

W związku z tym proces zostanie wznowiony (a w zasadzie zakończony) w momencie zmiany któregokolwiek z tych sygnałów.

Instrukcja warunkowa

W każdym języku programowania obecna jest instrukcja warunkowa. Nie inaczej jest w VHDL-u. Mamy do dyspozycji instrukcję if-then o następującej składni:

```
instrukcja_warunkowa ::= [etykieta:]
    if warunek then instrukcje
    { elseif warunek then instrukcje }
    [ else instrukcje ]
    end if [etykieta];
```

Działania tej instrukcji raczej nie trzeba tłumaczyć. Zwróćcie jednak uwagę na konstrukcję `elseif-then`, dzięki której można w prosty sposób tworzyć instrukcje wielokrotnego wyboru.

Instrukcja wyboru

Instrukcję wyboru można zrealizować za pomocą instrukcji `if-elseif`. W VHDL-u istnieje jednak instrukcja do tego przeznaczona – `case-when`:

```
instrukcja_wyboru ::= [etykieta:]
    case wyrażenie is
        when wartości => instrukcje
        { when wartości => instrukcje }
    end case [etykieta];
```

Wybór jest zależny od wartości wyrażenia umieszczonego po słowie kluczowym `case`. Musi być typu dyskretnego, ale może być też jednowymiarową tablicą złożoną ze znaków ASCII.

Głównym elementem instrukcji jest zestaw bloków odpowiadających poszczególnym przypadkom wprowadzanych słowem kluczowym `when`. Po nim następuje co najmniej jedna wartość wyboru:

```
wartości ::= wartość { | wartość }

wartość ::= wyrażenie | zakres | nazwa | others
```

Instrukcja wyboru

Przyjrzyjmy się przykładom instrukcji wyboru, które zilustrują wybrane możliwości tej instrukcji.

Wartości typu dyskretnego

```
variable a: integer range 1 to 50;

case a is
    when 1 => operacja_A;
    when 2 | 3 => operacja_B;
    when 13 to 28 => operacja_C;
    when others => operacja_D;
end case;
```

Wartości w postaci tablicy

```
variable b: bit_vector(0 to 1);

case b is
    when "00" | "11"=>  y <= 1
    when "01" | "10" => y <= 0
end case;
```

Instrukcja wyboru

Wartości w postaci typu wyliczeniowego

```
type Dni is (pn, wt, sr, cz, pt, so, nd);
variable d: Dni;

case d is
   when pn to pt => robota;
   when so | nd => wolne;
end case;
```

Instrukcja pętli

W VHDL-u mamy do dyspozycji również **pętle**. Cel ich stosowania jest taki sam jak w innych językach – wielokrotne wykonanie pewnych instrukcji.

Składnia pętli jest następująca:

```
instrukcja_pętli ::= [etykieta:]
    [schemat_iteracji] loop
       instrukcje
    end loop [etykieta];
```

Pętle mogą wystąpić w dwóch różnych formach zależnych od schematu iteracji, który definiuje się przed słowem kluczowym `loop`:

```
schemat_iteracji ::= while warunek
                   | for identyfikator in zakres
```

Instrukcja pętli

Jeżeli nie wskażemy żadnego schematu, uzyskamy pętlę nieskończoną:

```
signal clk: bit := '0';
process
begin
    loop
        clk <= not clk after 1 ms;
    end loop;
end process;
```

Ten proces będzie nieustannie zmieniał stan sygnału `clk` na przeciwny w odstępach 1-milisekndowych, czyli będzie generował sygnał prostokątny.

Instrukcja pętli

Jeżeli nie wskażemy żadnego schematu, uzyskamy pętlę nieskończoną:

```
signal clk: bit := '0';
process
begin
    loop
        clk <= not clk after 1 ms;
    end loop;
end process;
```

Ten proces będzie nieustannie zmieniał stan sygnału `clk` na przeciwny w odstępach 1-milisekndowych, czyli będzie generował sygnał prostokątny.

Pytanie na rozruszanie: jaką częstotliwość ma sygnał `clk`?

Instrukcja pętli

Po zastosowaniu schematu `while` uzyskujemy pętlę podobną do pętli `while` znanej z innych języków programowania:

```
variable i: positive := 0;
signal y: bit;
signal x: bit_vector(0 to 7);

while i < 8 loop
    y <= x(i) after 10 ns;
    i = i + 1;
end loop;
```

Pętla będzie wykonywana do momentu, gdy warunek **`i < 8`** przestanie być spełniony.

Instrukcja pętli

Po zastosowaniu schematu `while` uzyskujemy pętlę podobną do pętli `while` znanej z innych języków programowania:

```
variable i: positive := 0;
signal y: bit;
signal x: bit_vector(0 to 7);

while i < 8 loop
    y <= x(i) after 10 ns;
    i = i + 1;
end loop;
```

Pętla będzie wykonywana do momentu, gdy warunek **`i < 8`** przestanie być spełniony.

Pytanie na rozruszanie: jaką funkcję realizuje przedstawiony tu układ?

Instrukcja pętli

Dzięki kolejnemu schematowi iteracji możemy zrealizować klasyczne pętle typu `for`. Poprzedni przykład zrealizowany z tą pętlą wygląda następująco:

```
variable i: positive;
signal y: bit;
signal x: bit_vector(0 to 7);

for i in 0 to 7 loop
   y <= x(i) after 10 ns;
end loop;
```

Zmienna i przyjmuje kolejne wartości z zakresu od 0 do 7.

Instrukcja wyjścia z pętli

Przed chwilą omawialiśmy pętle i sposoby definiowania warunków ich zakończenia. Istnieje jeszcze jedna możliwość – instrukcja wyjścia:

```
instrukcja_wyjścia ::= [etykieta:]
    exit [etykieta_pętli] [when warunek];
```

Przypuśćmy, że mamy nieskończoną pętlę:

```
loop
    instrukcja_A
    exit when reset = '1';
    instrukcja_B
end loop;
```

Jeśli sygnał `reset` będzie miał stan niski, pętla będzie się wykonywać. Jeśli przyjmie stan wysoki, nastąpi natychmiastowe wyjście z pętli, tzn. w danej iteracji `instrukcja_A` zostanie wykonana, ale `instrukcja_B` już nie.

Instrukcja kolejnej iteracji

Istnieje jeszcze jedna instrukcja związana z pętlami – instrukcja kolejnej iteracji:

```
instrukcja_kolejnej_iteracji ::= [etykieta:]
    next [etykieta_pętli] [when warunek];
```

Przyjrzyjmy się takiemu przykładowi:

```
signal a, b: bit := 0;

for i in 1 to 99 loop
    a <= '1' after 1 ms, '0' after 2 ms;
    next when ((i mod 3) = 0);
    b <= '1' after 1 ms, '0' after 2 ms;
end loop
```

Pytanie na rozruszanie: co tu się dzieje?

Instrukcja raportu

Instrukcja raportu służy do wyświetlania komunikatów w czasie działania symulacji:

```
instrukcja_raportu ::= [etykieta:]
    report wyrażenie [severity wyrażenie];
```

Wyrażenie po słowie kluczowym report musi być typu `String`, z kolei wartość opcjonalnego parametru `severity` musi być typu `SEVERITY_LEVEL` (czyli musi przyjąć jedną z wartości: `NOTE`, `WARNING`, `ERROR` i `FAILURE`).

Przykłady:

```
report "Pozdrowienia z procesu!";

report "Coś złego się podziało..." severity WARNING;

report "Za chwilę coś wybuchnie!" severity FAILURE;
```

Instrukcja asercji

W jednym z poprzednich wykładów temat asercji już się pojawił.

Asercja jest podobna do raportu, z tym że komunikat się pojawi tylko wtedy, gdy wskazany **warunek nie jest spełniony**.

Składnia instrukcji:

```
instrukcja_asercji ::= [etykieta:]
    assert warunek
    [report wyrażenie] [severity wyrażenie];
```

Przykład:

```
assert (y = '0') report ”Bledna wartosc wyjscia”;
```

To samo można zapisać w inny, często bardziej czytelny sposób:

```
assert not (y = '1') report ”Bledna wartosc wyjscia”;
```

Instrukcja pusta

I na koniec najciekawsza z instrukcji – instrukcja pusta, która nic nie robi :)

Jej składnia wygląda następująco:

```
instrukcja_pusta ::= [etykieta:] null;
```

Przydaje się sporadycznie, np. w instrukcji przypadku:

```
case a is
   when 0 => null;
   when others => a := a + 1;
end case;
```

Podsumowanie

Omówiłem instrukcje sekwencyjne, czyli te, które są stosowane w procesach i procedurach.

Jak wspomniałem na wstępie, modelowanie układów cyfrowych wymaga umiejętnego połączenia instrukcji sekwencyjnych – tam, gdzie jest to wskazane – z instrukcjami współbieżnymi, które są niejako naturalne dla rzeczywistych układów cyfrowych.

W związku z tym kolejny temat to **instrukcje współbieżne**.