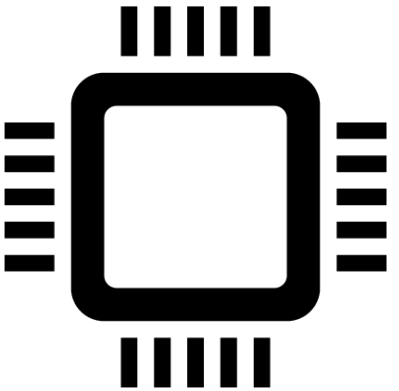
Programowanie struktur cyfrowych



Język VHDL.
Typy

dr Aleksander Lamża

Uniwersytet J. Kochanowskiego w Kielcach
Uniwersytet Śląski w Katowicach

aleksander.lamza@us.edu.pl

Czym są typy?

W VHDL-u mamy do dyspozycji dwie główne grupy typów: **proste** i **złożone**.

W standardzie języka jest zdefiniowanych kilka podstawowych typów, takich jak choćby używany przez nas wcześniej typ `bit` (prosty) czy `bit_vector` (złożony). Mamy również możliwość definiowania własnych typów i podtypów.

Poza standardowymi typami będziemy też korzystać z typów zdefiniowanych w bibliotece IEEE, która zawiera dodatkowe elementy ujęte w standardach, zwłaszcza 1164.

Czym są typy?

Na początek pytanie – czym tak naprawdę jest typ?

Typ to zbiór wartości o pewnych charakterystycznych cechach.

Każdy typ posiada identyfikującą go **nazwę**, za pomocą której wskazujemy, że dany obiekt (sygnał, zmienna) ma przyjąć te cechy i ma się zachowywać w określony sposób.

Deklaracja typu w notacji BNF wygląda następująco:

```
deklaracja_typu := type identyfikator is definicja_typu;
```

Składnia definicji typu zależy od konkretnej grupy typów (szczegóły za chwilę).

Przyjrzymy się wybranym typom, które wykorzystuje się najczęściej. Później przejdziemy do najpowszechniej stosowanego typu `std_logic` i `std_logic_vector`.

Typy proste

Typy proste to takie, których wartości nie mogą się składać z mniejszych elementów.

Dodatkowo poszczególne wartości są uporządkowane według pewnej skali (kolejności). Dzięki temu dla dwóch wartości można określić relację równości, mniejszości i większości.

W VHDL-u zdefiniowane są cztery główne typy proste:

- wyliczeniowe,
- całkowite,
- rzeczywiste,
- fizykalne.

Typy proste

Podstawową własnością typów prostych skalarnych jest **zakres** definiowany pierwszą i ostatnią wartością oraz kierunkiem zmian.

Składnia typowego zakresu w notacji BNF:

```
zakres := wyrażenie kierunek wyrażenie
```

Składnik „wyrażenie” to najczęściej literał (numeryczny).

Kierunek może być wznoszący (**`to`**) lub opadający (**`downto`**).

Kilka przykładów:

```
0 to 7 -- zakres liczb całkowitych od 0 do 7

10 downto 1 -- zakres liczb całkowitych od 10 do 1
```

Możemy mieć też do czynienia z zakresem **pustym**:

```
1 to 0

0 downto 15
```

Typy proste: wyliczeniowe

Typ wyliczeniowy jest typem dyskretnym, czyli jego wartości mają określoną pozycję w zdefiniowanym zakresie równą liczbie całkowitej.

Składa się z **różnych** wartości nienumerycznych pełniących funkcję identyfikatorów:

```
definicja_typu_wyliczeniowego := ( identyfikator {, identyfikator})
```

Każdy element typu wyliczeniowego ma przyporządkowaną liczbę naturalną. Liczba ta określa położenie identyfikatora w zbiorze. Pierwszy identyfikator ma wartość 0, kolejny 1 itd.

Na przykład typ `SeverityLevel` stosowany w asercjach jest typem wyliczeniowym o następującej definicji:

```
type SeverityLevel is (NOTE, WARNING, ERROR, FAILURE)
```

Wartości wyliczenia: 0 1 2 3

Typy proste: całkowity

Typ całkowity (**`integer`**) jest dyskretnym typem numerycznym, który składa się z liczb całkowitych należących do ustalonego zakresu:

```
definicja_typu_całkowitego := range zakres
```

Z typem `integer` są powiązane podtypy **`natural`** (liczby naturalne) i **`positive`** (całkowite liczby dodatnie).

Można tworzyć własne typy całkowite, zawężając pełny zakres 32-bitowego typu `integer`. Na przykład definicja typu reprezentującego 4-bitową liczbę całkowitą bez znaku może wyglądać tak:

```
type uint4 is range 0 to 15;
```

Jeżeli teraz zadeklarujemy sygnał tego typu:

```
signal a: uint4;
```

i przypiszemy mu wartość spoza zdefiniowanego zakresu:

```
a <= 16;
```

dostaniemy błąd kompilacji „Value 16 out of range 0 to 15”.

Typy proste: całkowity

W takiej sytuacji można też zastosować podtyp typu `integer`:

```
subtype uint4 is integer range 0 to 15;
```

W ten sposób jawnie określamy pokrewieństwo obu typów, przy czym `uint4` jest zawężeniem zakresu.

Podtypów używa się również wtedy, gdy chcemy zdefiniować odmianę typu tablicowego o ustalonym rozmiarze, np.:

```
subtype uint4 is bit_vector(3 downto 0);
```

Uzyskujemy w ten sposób wektor czterech bitów.

Typy proste: rzeczywiste

Typ fizykalny (`real`) jest typem numerycznym, na który składają się liczby rzeczywiste z ustalonego zakresu:

```
definicja_typu_rzeczywistego := range zakres
```

Wartości typu rzeczywistego nie są dyskretne. Co prawda mają skończoną rozdzielczość dyktowaną implementacją, ale należy przyjąć, że w miarę możliwości odzwierciedlają „prawdziwe” liczby rzeczywiste z pewną dokładnością.

W standardzie określono gwarantowany zakres na -1E38 do +1E38.

Oczywiście, podobnie jak w przypadku liczb całkowitych, możemy definiować własne typy i podtypy rzeczywiste:

```
type coeff is range 0.0 to 1.0;
```

Typy proste: fizykalne

Typ fizykalny służy do reprezentowania wielkości fizycznych, a dokładniej – do definiowania jednostek miary danej wielkości.

Jeżeli w procesie, który staramy się zaimplementować w strukturze programowalnej mamy do czynienia z napięciem, możemy zdefiniować typ fizykalny `voltage` na przykład w ten sposób:

```
type voltage is range -1e6 to 1e6
    units
        mV;                 --miliwolty
        V = 1000 mV;        --wolty
        kV = 1000 V;        --kilowolty
    end units;
```

Wartość napięcia będzie wyrażana w miliwoltach (najmniejszej zdefiniowanej jednostce), ale można wykonywać operacje na dowolnych jednostkach:

```
1kV + 230V - 5mV
```

Wartość odpowiadająca temu wyrażeniu to **1 229 995 mV**.

Typy złożone

Teraz przyszedł czas na **typy złożone**.

Można się domyślić, że – w odróżnieniu od typów prostych – typy złożone składają się z innych typów. Mogą to być typy proste, ale też typy złożone.

Wyróżniamy dwa główne typy złożone: **tablice** (wektory) i **rekordy**.

Zaczynamy od tablic...

Typy złożone: tablice

Typ tablicowy (**`array`**) to uporządkowany zbiór elementów tego samego typu. Każdy element ma jednoznacznie przyporządkowany indeks (lub indeksy). Indeksy są typu (lub podtypu) całkowitego.

Poniżej przykładowa definicja typu tablicowego `byte`:

```
type byte is array (7 downto 0) of bit;
```

W ten sposób definiujemy bajt składający się z 8 bitów uporządkowanych malejąco od 7 do 0 (o zakresach malejących i rosnących tablic bitów będzie mowa w dalszej części).

Poza jednowymiarowymi tablicami możemy definiować również wielowymiarowe:

```
type bitmap is array (0 to 15, 0 to 7) of bit;
```

W ten sposób zdefiniowaliśmy typ dla bitmapy o wymiarach 16x8.

Typy złożone: tablice

Poza tablicami o ustalonym rozmiarze (tzw. ograniczonych) możemy definiować również tablice nieograniczone.

Dobrym przykładem tego typu tablicy jest predefiniowany w VHDL-u typ `string`:

```
type string is array (positive range <>) of character;
```

oraz `bit_vector`:

```
type bit_vector is array (natural range <>) of bit;
```

W praktycznych realizacjach często ogranicza się takie typu poprzez zdefiniowanie zakresu w podtypie. Przykład takiej definicji już pokazałem wcześniej:

```
subtype uint4 is bit_vector(3 downto 0);
```

Typy złożone: tablice

Dostęp do elementów tablicy jest realizowany przez indeksy.

W przypadku tablicy jednowymiarowej wygląda to np. tak:

```
signal b: byte;

…

byte(5) <= '0';
```

Jeżeli mamy do czynienia z tablicą wielowymiarową, kolejne indeksy podajemy po przecinku:

```
variable image: bitmap;

…

image(3,0) <= '1';
```

(Tematem zmiennych – `variable` – zajmiemy się trochę później).

Typy złożone: rekordy

Typ rekordowy – w odróżnieniu od typu tablicowego – pozwala na łączenie elementów różnych typów.

Dla każdego elementu trzeba określić identyfikator i typ.

Poniżej przykład rekordu przechowującego informacje o dokonanym pomiarze (czas pomiaru i zmierzona wartość):

```
type measure is
    record
        timestamp: time;
        value: real;
end record;
```

Dostęp do poszczególnych pól rekordu jest realizowany poprzez składnię „z kropką”:

```
variable m: measure;

…

m.value = 5.73;
```

Inne typy

Pozostało jeszcze kilka typów, których nie opisałem, np. typy plikowe czy wskaźnikowe.

Ponieważ nie są tak powszechnie stosowane w typowych projektach, pominę je na tym etapie. Zainteresowanych odsyłam do literatury.

Teraz omówimy często stosowane typy `std_logic` i `std_logic_vector`, które są „lepszymi” odpowiednikami standardowych typów `bit` i `bit_vector`.

Typ `std_logic`

W większości przypadków zamiast typu `bit` stosuje się `std_logic`.
Dlaczego? Dlatego, że typ `bit` „mieści” tylko dwie wartości: '0' i '1'.
Są to oczywiście dwa stany występujące w układach cyfrowych. Trzeba jednak wziąć pod uwagę sytuacje, których te dwa stany nie uwzględniają, a które mogą się zdarzyć w trakcie projektowania układu.

Wyobraźmy sobie układ, w którym w dwóch miejscach jest ustalany stan jednego sygnału:

```
signal s0: bit;
…
s0 <= '0';
…
s0 <= '1';
```

Odpowiada to sytuacji, w której wyjścia dwóch elementów są połączone i ustalają sprzeczne stany, co jest niedopuszczalne w rzeczywistych układach.

Typ `std_logic`

Inną sytuacją, z którą możemy mieć do czynienia, jest układ z elementami trójstanowymi. Są to elementy cyfrowe, które poza stanem wysokim i niskim mają jeszcze tzw. stan wysokiej impedancji, który pozwala tym elementom „odciąć się” od pozostałych.

Jest to bardzo częsta sytuacja w układach, w których wiele podsystemów korzysta ze wspólnych magistral.

Tego typu sytuacji nie da się zrealizować i zasymulować z wykorzystaniem typu bit. Potrzebny jest typ bitowy rozszerzony o inne „stany”.

I tu wkracza typ **`std_logic`**. Aby z niego skorzystać, musimy dodać informację o wykorzystaniu pakietu `std_logic_1164` z biblioteki `ieee`:

```
library ieee;
use ieee.std_logic_1164.all;
```

Typ `std_logic`

Typ ten oferuje aż dziewięć możliwych stanów:

- **`'U'`** – stan niezainicjalizowany,
- **`'X'`** – stan nieznany,
- **`'0'`** – stan niski (tak jak w typie `bit`),
- **`'1'`** – stan wysoki (tak jak w typie `bit`),
- **`'Z'`** – stan wysokiej impedancji (odcięcie),
- **`'W'`** – stan nieznany (podobnie jak 'X', ale stan nie jest wymuszany),
- **`'L'`** – stan niski niewymuszony (tzw. pulldown, czyli podciągnięcie do 0),
- **`'H'`** – stan wysoki niewymuszony (tzw. pullup, czyli podciągnięcie do 1),
- **`'-'`** – stan dowolny.

Typ `std_logic`

Jeżeli teraz wrócimy do wcześniejszego przykładu z przypisywaniem sprzecznych stanów do sygnału i zastosujemy typ `std_logic`, układ zachowa się inaczej:

```
signal s0: std_logic;
…
s0 <= '0';
s0 <= '1';
```

Stan, jaki uzyskamy po uruchomieniu symulacji, to `X`, czyli stan nieznany (działający przykład: https://www.edaplayground.com/x/5TQR). Jest to wyraźny sygnał, że układ nie jest zaprojektowany prawidłowo.

Co innego, jeżeli zastosujemy elementy trójstanowe i w jednym miejscu do sygnału przypiszemy stan `Z`:

```
s0 <= '0';
s0 <= 'Z';
```

W takim przypadku sygnał przyjmie stan `0`.

→ Pobawcie się i przetestujcie różne kombinacje stanów.

Typ `std_logic_vector`

Jeśli w danej sytuacji wygodniej jest operować wieloma liniami naraz (przykładem może być magistrala danych), przydaje się tablica (wektor) bitów.

W przypadku standardowego typu `bit` typem tablicowym jest `bit_vector`. Można się domyślić, że dla typu `std_logic` będzie to `std_logic_vector`.

Przykład deklaracji 4-bitowej magistrali danych:

```
signal data_bus: std_logic_vector(3 downto 0);
```

Po co w deklaracji pojawia się `3 downto 0`?
W ten sposób wyznaczamy rozmiar wektora (4 elementy typu `std_logic`).

W tym przypadku mamy zakres 3..0. Gdybyśmy napisali `0 to 3`, mielibyśmy zakres 0..3. Czy ma to jakieś znaczenie?

Typ `std_logic_vector`

Kolejność bitów jest istotna.

Jeżeli ustalimy kolejność malejącą (`3 downto 0`), przykładowa liczba 13 zapisana na bitach wygląda tak:

3	2	1	0
1	**1**	**0**	**1**
MSB			LSB

MSB to *Most Significant Bit*, czyli najbardziej znaczący bit (o największej wadze).
LSB to *Least Significant Bit*, czyli najmniej znaczący bit.

Liczba 13 zapisana w wektorze o kolejności rosnącej (`0 to 3`) wygląda prawie tak samo:

0	1	2	3
1	**1**	**0**	**1**
MSB			LSB

Ważne jest to, by konsekwentnie stosować jedną kolejność bitów w wektorach. Wymieszanie ich może powodować trudne w wykryciu błędy.

Typ `std_logic_vector`

Zobaczmy teraz, jak ustawiać wartość wektorów i jak ją wyświetlać podczas symulacji (działający przykład: https://www.edaplayground.com/x/p3E).

Deklaracja wektora:

```
signal data_bus: std_logic_vector(3 downto 0);
```

Wartość przechowywaną w wektorze możemy wyświetlić tak:

```
process begin
    wait for 1 ns;   --czekamy na ustalenie się stanów

    report to_string(data_bus);

end process;
```

Po uruchomieniu symulacji w konsoli pojawi się komunikat:

```
# EXECUTION:: NOTE    : UUUU
```

W typie `std_logic` wartość `'U'` oznacza stan niezainicjalizowany (niezdefiniowany). Oznacza to, że nie przypisaliśmy do niego żadnej wartości. To prawda...

Typ `std_logic_vector`

Jak przypisać wartość do wektora?

Można przypisać wartość do każdego bitu z osobna:

```
data_bus(3) <= '1';
data_bus(2) <= '1';
data_bus(1) <= '0';
data_bus(0) <= '1';
```

Można też ustawić je za jednym razem:

```
data_bus <= "1101";
```

Jeżeli chcemy posługiwać się nie pojedynczymi bitami, ale reprezentacją liczbową, możemy napisać:

```
data_bus <= std_logic_vector(to_unsigned(13, data_bus'length));
```

Wymaga to jednak dołączenia pakietu `numeric_std`:

```
use ieee.numeric_std.all;
```

Typ `std_logic_vector`

VHDL pozwala na wprowadzanie wartości liczbowych w wielu systemach pozycyjnych. Najbardziej typowe to: dwójkowy, ósemkowy, dziesiętny (domyślny) i szesnastkowy.

Przykłady tego typu literałów:

- **`16#9B#`** – szesnastkowa liczba 9B, czyli 155,
- **`8#36#`** – ósemkowa liczba 36, czyli 30,
- **`2#1011_0011#`** – dwójkowa liczba 10110011, czyli 179.

Nic nie stoi jednak na przeszkodzie, by liczby zapisywać np. w systemie trójkowym ;)

`3#120#`

Ile to jest?

Wyświetlanie wartości

Jeżeli chcemy wyświetlić wartość inaczej niż to oferuje funkcja `to_string()`, możemy skorzystać z funkcji `image()` dostępnej w większości typów.

Funkcję tę wywołuje się na typie wartości, której postać chcemy przedstawić w postaci łańcucha znaków:

```
report integer'image(5);
```

Jak się można spodziewać, powyższy kod spowoduje wyświetlenie w konsoli wartości 5.

Typ `std_logic_vector` nie dysponuje funkcją `image`. Co w takiej sytuacji zrobić? Trzeba przekonwertwać wartość typu `std_logic_vector` na `integer`. Okazuje się, że nie da się tego zrobić bezpośrednio – pośredniczyć musi typ `unsigned` (lub `signed`, jeżeli chodzi o wektory reprezentujące liczby ze znakiem).

Wyświetlanie wartości

Zakładamy, że mamy sygnał data_bus zadeklarowany w ten sposób:

```
signal data_bus: std_logic_vector(3 downto 0);
```

Użyjemy więc najpierw rzutowania `std_logic_vector` na `unsigned`:

```
unsigned(data_bus)
```

Wynik tej operacji przekażemy funkcji `to_integer()`, która zamieni wektor bitów na wartość liczbową:

```
to_integer(unsigned(data_bus))
```

Tak przygotowaną wartość (już typu `integer`) możemy przekazać funkcji `image` typu `integer`:

```
report integer'image(to_integer(unsigned(data_bus)))
```

Zadanie

Zdefiniujcie przykładowe typy opisane w prezentacji:

- typ wyliczeniowy,
- podtyp typu całkowitego,
- podtyp typu rzeczywistego,
- typ fizykalny,
- typ tablicowy złożony z elementów dowolnego typu wyliczeniowego,
- typ rekordowy o co najmniej dwóch polach.