# Język VHDL

## Jednostki i architektury

Poprzednio wspomnieliśmy o tym, że język VHDL pochodzi z początku lat 80-tych. Stworzony był by opisywać specyfikację układów cyfrowych, po części również symulację. Nie był tworzony z myślą o syntezie, do której będzie wykorzystywany na laboratorium i o której będziemy w głównej mierze mówić. Język VHDL pozwala na opis specyfikacji rozbudowanych projektów cyfrowych, które mogą być podzielone na moduły i z tego powodu podział projektu na elementy, czy też mniejsze jednostki jest podstawą. Wszystko co opisuje się w VHDL jest opisywane w postaci jednostki, z którą kojarzy się później kilka architektur.

```
entity HalfAdder is
    port ( A : in  BIT;
           B : in  BIT;
           S : out  BIT;
           C : out  BIT);
end entity HalfAdder;
```

Opis jednostki (entity) rozpoczyna się od listy portów (sygnałów wejściowych i wyjściowych lub interfejsu jednostki). Zaczynamy od słowa kluczowego „entity”, później wpisujemy identyfikator jednostki (HalfAdder), następnie słowo kluczowe „is”. Opis portów rozpoczyna się od słowa „port” po którym pojawia się lista opisanych portów. Każdy port ma unikalną nazwę, po dwukropku opisuje się tryb pracy portu. Porty przenoszą sygnały określonego typu. Tu jest użyty „BIT” (wbudowany w pakiecie standardowym). Tego typu nie używa się w opisach do syntezy, bo nie nadaje się on do opisywania sygnałów w układach rzeczywistych.

Definicji jednostki używa się tylko jeden raz. Do definicji jednostki możemy przypisać szereg architektur. Przydaje się to podczas specyfikacji i symulacji układów, natomiast nie nadaje się do syntezy. Jedną jednostkę można opisać na kilka sposobów.

Istnieją 3 poziomy opisu architektur. Będziemy omawiali poziomy w kierunku od najniższego do najwyższego.

### Opis strukturalny

Najniżej w hierarchii znajduje się opis strukturalny, w którym opisujemy strukturę danego modułu, z jakich elementów i podzespołów się składa. Jest to po prostu lista instancji elementów niższego poziomu. Elementy pochodzą z biblioteki:

```
library UNISIM;
use UNISIM.VComponents.all;

          . . .
architecture Structural of HalfAdder is
begin
    XOR_gate : XOR2 port map ( A, B, S );
    AND_gate : AND2 port map ( A, B, C );
end architecture Structural;
```

Opis architektury rozpoczynamy od słowa kluczowego „architecture”, następnie wpisujemy nazwę architektury i nazwę jednostki, której opis będzie dotyczył. Między „begin” a „end architecture” podajemy właściwy opis wewnętrznej organizacji modułu. Półsumator złożony jest z bramek AND oraz XOR. Bramki te są wzięte z biblioteki, co sprowadza na nas konieczność zastosowania klauzuli użycia biblioteki. W opisywanym przypadku korzystamy z biblioteki „UNISIM”. Następnie umieszczana jest klauzula użycia komponentów z tej biblioteki. Zapis „VComponents.all” oznacza że będziemy używać wszystkich komponentów. Zamiast „all” można by wypisać konkretne elementy (stosuje się to rzadko):

```
use UNISIM.VComponents.XOR2;
use UNISIM.VComponents.AND2;
```

Klauzula użycia biblioteki kojarzyć się nam będzie z dyrektywą „#include” z języka C++, gdzie zadeklarowany plik nagłówkowy jest ważny od momentu jego zadeklarowania do końca pliku z kodem. W przypadku VHDL zadeklarowana biblioteka dotyczy tylko najbliższej jednostki. Jeżeli w pliku mamy zdefiniowanych kilka jednostek, to każdą jednostkę musimy poprzedzić listą bibliotek, z których chcemy korzystać. Zamiast bibliotek można umieszczać instancje jednostek zdefiniowanych przez nas wcześniej. Tak powstaje hierarchiczna struktura projektu.

Tworzenie instancji jakiegoś elementu ma składnię:

```
XOR_gate : XOR2 port map ( A, B, S );
```

Etykieta przed dwukropkiem jest obowiązkowa. Jest to sensowne, ponieważ w procesie symulacji będziemy mogli odwoływać się do wstawianych elementów. Następnie użyta jest nazwa komponentu wstawionego z biblioteki (może to być jakaś jednostka wcześniej przez nas opracowana). Ostatnim elementem jest przypisanie portów. Jednostka „XOR2" zdefiniowana w bibliotece ma w swojej definicji listę trzech portów. My odwzorowujemy porty jednostki „HalfAdder" na porty opisane w definicji jednostki „XOR2".

Identyfikatory w VHDL mogą się składać z liter, cyfr i znaku „_". Nazwa musi rozpoczynać się od litery, znak „_" może znajdować się wewnątrz identyfikatora (nie może on znajdować się na początku ani na końcu). Język VHDL nie rozróżnia wielkości liter. Komentarze obejmują tylko jedną linię począwszy od dwóch myślników. Nie ma tu komentarzy blokowych.

Rysowanie schematu generuje opis na poziomie strukturalnym. Każdy plik .SCH jest automatycznie konwertowany na poziom strukturalny, którego opis znajduje się w pliku .VHF. Zaglądając do tego pliku możemy się dowiedzieć w jaki sposób schemat układu został przełożony na opis w kodzie VHDL.

### Opis przepływowy

Wyższym poziomem opisu jest opis przepływowy (Data Flow). Tutaj opisujemy sposób przenoszenia wartości logicznych pomiędzy poszczególnymi portami:

```
architecture Dataflow of HalfAdder is
begin
    S <= A xor B;
    C <= A and B;
end architecture Dataflow;
```

Zamiast wstawiania komponentów bibliotecznych piszemy że sygnał C jest funkcją AND portów wejściowych A i B. Operatory „and" i „xor" są wbudowanymi operatorami, które działają na typie BIT. Nasze porty również są typu BIT.

Opisując działanie naszego modułu na poziomie przepływowym nie musimy się powoływać na konkretne elementy. Narzędzie syntezy ma większe pole manewru. Może ono zoptymalizować wpisane przez nas równania pod kątem złożoności jak i architektury na której będzie implementowany układ.

### Opis behawioralny

Inaczej nazywany jako opis proceduralny lub sekwencyjny. Opis taki nie ma nic wspólnego ze strukturą układu. Jest to opis najlepiej nadający się do symulacji. Tłumaczy on w jaki sposób komputer ma symulować pracę modułu. Nie ma żadnych danych o zastosowanych bramkach czy też przerzutnikach.

Wszystkie instrukcje, które są podane w architekturze, są wykonywane współbieżnie. Gdy chcemy cokolwiek opisywać sekwencyjnie, musimy użyć instrukcji procesu.

```
architecture Behavioral of HalfAdder is
begin
    process( A, B )
    begin
        -- Sum
        if A /= B then
            S <= '1';
        else
            S <= '0';
        end if;
        -- Carry
        if A = '1' and B = '1' then
            C <= '1';
        else
            C <= '0';
        end if;
    end process;
end architecture Behavioral;
```

Zaczynamy od słowa kluczowego „process" po którym następuje lista wrażliwości procesu, a między „begin" i „end process" mamy listę instrukcji programu sekwencyjnego.

Synteza z poziomu strukturalnego jest najprostsza. Wykonywana jest szybko, bo narzędzie syntezy „ma mało do roboty". W przypadku opisu przepływowego narzędzie syntezy może dokonać optymalizacji układu i jego działanie będzie wolniejsze. W przypadku opisu behawioralnego wszystko zależy od narzędzia syntezy. Opisy behawioralne generują sporo zamieszania. Można się spotkać z opiniami, że opis behawioralny zwalnia projektanta od zajmowania się szczegółami implementacyjnymi. Narzędzia syntezy nie dają sobie rady z optymalnym realizowaniem opisów behawioralnych. Czasami narzędzie musi na podstawie opisu „zorientować się" o jaki układ nam chodzi, następnie dobrać odpowiedni sposób realizacji opisu przy pomocy zasobów sprzętowych by na końcu projekt zrealizować. Nie należy liczyć na to, że wynik syntezy będzie najlepszy z możliwych i będzie optymalny pod względem szybkości działania jak i zajętości zasobów.

Obok omówionych powyżej poziomów opisu istnieje również coś takiego jak:

**Poziom przesłań międzyrejestrowych**

(Register Transfer Level). Jest to poziom, który jest specyficznym połączeniem opisów przepływowego i behawioralnego dla potrzeb syntezy. Opisy na tym poziomie będą używane przez nas do celów syntezy.

## Porty

Port może pracować w jednym z trybów pracy:

```
B : in BIT;
S : out BIT;
```

IN – port jest wówczas tylko do odczytu (wprowadzamy nim sygnał do układu)

OUT – port jest tylko do zapisu. Nie można go czytać wewnątrz architektury.

INOUT – praca dwukierunkowa.

BUFFER – to jest właściwy tryb wyjściowy z możliwością odczytu. Port jako taki jest wyjściowy a sygnał wystawiany na tym porcie można odczytywać wewnątrz jednostki.

Istnieje jeszcze tryb LINKAGE który został zgłoszony do usunięcia ze standardu. Nie należy go stosować. Był on używany jako łącznik podczas łączenia hierarchii modułów. Do takiego portu nie można było dokonywać zapisu ani odczytu.

W syntezie układów należy używać portów typu IN oraz OUT. Pozostałe porty mogą być syntezowane w sprzęcie przez narzędzia w dziwny sposób. Jeżeli będziemy chcieli użyć bufora 3-stanowego, to będziemy go musieli opisać go ręcznie. Możliwości pracy dwukierunkowej mogą być używane do specyfikacji i symulacji, ale nie do syntezy.

Jeżeli chcemy mieć port jednokierunkowy z możliwością odczytu, to należy korzystać z trybu BUFFER. Używanie trybu INOUT jest błędem. Poza tym port pracujący w trybie BUFFER może mieć tylko jedno przypisanie sygnału. Pojawia się problem z portami wyjściowymi, gdzie ich odczyt jest nielegalny:

```
entity AndNand is
    port ( A : in  BIT;
           B : in  BIT;
           C : in  BIT;
           WY_And  : out  BIT;
           WY_Nand : out  BIT);
end AndNand;
architecture DataflowBad of AndNand is
begin
    WY_And  <= A and B and C;
    WY_Nand <= not WY_And;
end DataflowBad;
```

Mamy tutaj 2-wyjściową bramkę AND NAND. Można by to było w najprostszy sposób uzyskać poprzez obliczenie iloczynu trzech sygnałów i następnie na jego zanegowaniu. Problem w tym, że taki opis jest nielegalny, ponieważ niechcący napisaliśmy operację odczytu portu wyjściowego (podkreślona linia). Można to obejść w następujący sposób:

```
architecture DataflowOK of AndNand is
    signal Int_And : BIT;
begin
    Int_And <= A and B and C;
    WY_And  <= Int_Ard;
    WY_Nand <= not Int_And;
end DataflowOK;
```

Polega to na zdefiniowaniu dodatkowego sygnału wewnętrznego (podkreślona linia) do którego przypisujemy wartość iloczynu. Następnie sygnał ten kopiujemy do portu wyjściowego (WY_And) oraz po zanegowaniu do drugiego wyjścia (WY_Nand).

Definiowanie sygnałów wewnętrznych polega na wstawieniu słowa „signal”, następnie podaniu nazwy sygnału i po dwukropku typu tego sygnału.

Instrukcję przypisania sygnału realizuje operator „<=”. W ten sposób można przypisywać wartości portom i sygnałom. Należy pamiętać o tym, że sygnały w przeciwieństwie do portów nie mają ściśle określonego trybu pracy. Z operacją przypisania związane jest tworzenie kolejki zdarzeń czasowych.

## Obiekty jawne VHDL

Do obiektów jawnych zalicza się SYGNAŁY, OPERATORY PRZYPISANIA oraz SPECJALNY MECHANIZM WARTOŚCIOWANIA. W układach cyfrowych często zdarza się, że jedna i ta sama linia jest sterowana

kilkoma sterownikami naraz. Jest to uwzględnione w mechanizmie wartościowania sygnałów. Każda instrukcja przypisania jest traktowana jako sterownik sygnału. Jeżeli dany sygnał jest przepisywany w wielu miejscach, to może mieć wiele sterowników. Wtedy jest potrzebne opracowanie sposobu według którego sygnał będzie ustawiany. Realizuje się to przy pomocy FUNKCJI ROZSTRZYGAJĄCYCH. Z mechanizmem wartościowania są związane TRANSAKCJE (uwzględnienie momentu przypisania). Przypisanie sygnały nie musi być natychmiastowe. Jeżeli chcemy wiernie symulować pracę układu cyfrowego, to do każdego przypisania należy dodać pewne opóźnienie. Z każdym sygnałem jest tworzona lista sterowników i skojarzonych z nim transakcji. Podczas symulacji te transakcje są wykonywane.

Sygnały są tworem skomplikowanym, ale za to mogą w miarę wiernie naśladować pracę rzeczywistych sygnałów elektronicznych w układzie.

Obiektami jawnymi są również stałe (słowo kluczowe „constant”):

```
constant Pi : REAL :=3.14;
```

Istnieją też obiekty jawne w postaci zmiennych. Kod opisujący zmienną typu czasowego z przypisaną wartością początkową wygląda następująco:

```
variable SetupTime : TIME := 2.5ns
```

Mamy tu również przykład stałej typu fizycznego (nanosekundy) typu TIME. Zmienne nie są syntezowalne. Występują głównie w procesach. Kojarzyć je należy z opisami sekwencyjnymi. Ze zmiennymi nie ma takiego zamieszania jak z sygnałami. Nie dotyczą ich takie rzeczy jak wiele sterowników, funkcje rozstrzygające, transakcje...

Obiektami jawnymi są również PLIKI. Można je wykorzystywać do przechowywania pobudzeń układów i zapisywania ich odpowiedzi.

## Klauzula „generic”

Jest to sposób na parametryzowanie jednostek. Typowym zastosowaniem może być zdefiniowanie bufora z magistralami o różnej szerokości.

Opis pseudoformalny:

```
entity identifier is
   generic ( parameter_declarations ); -- opticnal
   port ( port_declarations );         -- opticnal
   [ declarations ]                    -- opticnal
begin                                  \__ optional
   [ statements ]                      /
end entity identifier ;
```

Przed specyfikacją portów może wystąpić klauzula „generic” która określa parametry. Po klauzuli „port” mogą wystąpić dodatkowe deklaracje, gdzie można zdefiniować sygnały i zmienne, które będą globalne we wszystkich architekturach dla danej jednostki (rzadko się to stosuje). Po „begin” definiuje się część opisową jednostki. Przykład użycia klauzuli „generic”:

```
entity Buf is
   generic ( N : POSITIVE := 8;          -- data width
             Delay : DELAY_LENGTH := 2.5 ns );
   port ( Input  : in BIT_VECTOR( N-1 downto 0 );
          OE     : in BIT;
          Output : out BIT_VECTOR( N-1 downto 0 ) );
end entity Buf;
```

Tu mamy podane dwa parametry: N określa szerokość wektorów wejściowego i wyjściowego a parametr DELAY_LENGHT będzie używany w modelu czasowym jako wartość opóźnienia wnoszona przez bufor. Jest to bufor 3-stanowy (ma zdefiniowane wejście OE). Widzimy tu sposób zdefiniowana wektorów. Typ wektorowy konkretyzuje się podając w nawiasach okrągłych zakres indeksów (u nas indeksy od N-1 do 0). Gdyby zaszła potrzeba numerowania z indeksem rosnącym, to zamiast „N-l downto 0” należy napisać „0 to N-1”. W klauzuli „generic” są podane domyślne wartości parametrów.

## Pakiet „standard”

W kodzie źródłowym podanym przy okazji omawiania opisu strukturalnego była dołączona biblioteka UNISIM. Istnieje standardowy pakiet zdefiniowany od razu w języku VHDL, który jest domyślnie dołączany. Zawiera on definicje standardowych typów danych:

```
type INTEGER is range --usually typical INTEGER-- ;
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
type REAL is range --usually double precision f.p.-- ;
type BOOLEAN is (FALSE, TRUE);
type CHARACTER is ( --256 characters-- );
type STRING is array (POSITIVE range <>) of CHARACTER;
type TIME is range --implementation defined-- ;
  units
     fs;            -- femtosecond
     ps  = 1000 fs; -- picosecond
     ns  = 1000 ps; -- nanosecond
     us  = 1000 ns; -- microsecond
     ms  = 1000 us; -- millisecond
     sec = 1000 ms; -- second
     min = 60  sec; -- minute
     hr  = 60  min; -- hour
  end units;
subtype DELAY_LENGTH is TIME range 0 fs to TIME'HIGH;
```

6

Typ INTEGER jest tym, który znamy z innych języków programowania. W języku VHDL możemy tworzyć podtypy: z typu INTEGER zostały stworzone dwa podtypy (liczb naturalnych i dodatnich NATURAL i POSITIVE). Pojawia się tu również użycie atrybutu (przykładowo „INTEGER'HIGH”). Po lewej stronie apostrofu znajduje się coś, czego atrybut odczytujemy. Po prawej stronie – nazwa atrybutu. Jest to wyrażenie, które może zwracać różne rzeczy: mogą to być wartości, mogą to być podtypy.