

Instancje komponentów

Mamy zdefiniowane jakieś jednostki i chcemy je wstawić we wnętrze innej architektury. W przykładzie użyjemy 2-wejściowe bramki XOR i AND i zmontujemy z nich półsumator. Najpierw potrzebować będziemy definicji jednostek („XOR_2WE” oraz „AND_2WE”). Mają one oprócz listy portów listę parametrów generycznych:

<pre>entity XOR_2WE is generic(Tp : DELAY_LENGTH); port (I1, I2 : in STD_LOGIC; O : out STD_LOGIC); end XOR_2WE; architecture A of XOR_2WE is begin O <= I1 xor I2 after Tp; end A;</pre>	<pre>entity AND_2WE is generic(Tp : DELAY_LENGTH); port (I1, I2 : in STD_LOGIC; O : out STD_LOGIC); end AND_2WE; architecture A of AND_2WE is begin O <= I1 and I2 after Tp; end A;</pre>
---	--

Parametrem tym jest „Tp” (czas propagacji bramek). Architektura bramek składa się z jednej instrukcji współbieżnej (przypisanie z użyciem „after”). Pracujemy na typie STD_LOGIC. Wykorzystując definicje jednostek, możemy ich użyć w półsumatorze:

```
entity HalfAdder is
  (...)
architecture Structural of HalfAdder is
  component XOR_2WE is
    generic( Tp : DELAY_LENGTH );
    port ( I1, I2 : in STD_LOGIC; O : out STD_LOGIC);
  end component;
  component AND_2WE is
    generic( Tp : DELAY_LENGTH );
    port ( I1, I2 : in STD_LOGIC; O : out STD_LOGIC);
  end component;
```

Na początku będziemy potrzebowali deklaracji komponentów. Po słowie kluczowym „architecture” wstawiamy definicje sygnałów oraz deklaracje komponentów rozpoczynające się od słowa kluczowego „component”. Później następuje lista argumentów oraz przypisanie portów. Tak zadeklarowane komponenty wstawiamy w treści architektury we współbieżnych instrukcjach instancji komponentów:

```
  generic map( 5 ns ) port map( A, B, S );
  AND_gate : AND_2WE generic map( 3 ns ) port map( O=>C, I1=>A, I2=>B );
end architecture Structural;
```

Każda instancja musi mieć nazwę („XOR_gate : ” oraz „AND_gate : ”). Nazwę tworzymy zgodnie z zasadami tworzenia identyfikatorów w VHDL-u. Nazwa wygląda tak, jak etykieta (posiada dwukropek). Po niej następuje nazwa komponentu z odwzorowaniem parametrów generycznych („generic map”) oraz portów („port map”). Powyżej przedstawiono dwa style odwzorowania. Przy instancji bramki XOR mamy styl pozycyjny (kolejność sygnałów na liście jest identyczna kolejności portów komponentu). Możemy przyporządkowywać według nazw (tak zrobiono przy instancji bramki AND). Druga możliwość pozwala na zmianę kolejności zapisu.

W instancjach niektórych komponentów może zająć konieczność nie dołączenia niektórych portów. Używa się słowa kluczowego „open”. Zdarza się to głównie przy portach wyjściowych. Zapisujemy to tak:

NazwaPortu => open;

Wejścia powinno się dołączać. Jeżeli na schemacie układu pojawiają się komponenty z portami wejściowymi nie połączonymi, to zostaną one przez środowisko Xilinx dołączone do masy układu (podanie '0' na wejście).

Parametry generyczne mogą mieć wartości domyślne. Jeżeli pominiemy parametr, który taką wartość ma, to zostanie ona automatycznie podana. Brak wartości domyślnej zmusza nas do jawnego podawania wartości parametru.

Instrukcja generacji

Występuje ona w wariancie z pętlą i wariancie warunkowym. W wariancie z pętlą używa się słów kluczowych „for” oraz „generate”. W przykładzie posłużymy się sumatorem kaskadowym (zbudowany z 8-miu sumatorów pełnych). Układ będzie posiadał 8-bitowe porty wejściowe A oraz B, 8-bitowy port wyjściowy oraz odpowiednie wejścia i wyjścia przeniesień:

```
entity FullAdder is
  port ( A, B : in STD_LOGIC_VECTOR( 7 downto 0 );
        CI : in STD_LOGIC;
        S : out STD_LOGIC_VECTOR( 7 downto 0 );
        CO : out STD_LOGIC);
end FullAdder;
```

Można by napisać w VHDL-u moduły sumatorów pełnych i powielić je. Zamiast tego posłużymy się pętlą. Potrzebny będzie nam sygnał wewnętrzny dla przeniesień (wektor „Cint”):

```

architecture Dataflow of FullAdder is
    signal Cint : STD_LOGIC_VECTOR( 8 downto 0 );
begin
    lb: for i in 0 to 7 generate
        S(i) <= A(i) xor B(i) xor Cint(i);
        Cint(i + 1) <= ( A(i) and B(i) ) or
            ( A(i) and Cint(i) ) or ( B(i) and Cint(i) );
    end generate;
    Cint( 0 ) <= CI;
    CO <= Cint( 8 );
end Dataflow;

```

Cała architektura będzie składała się z jednej instrukcji generacji i dwóch współbieżnych przypisań. Składnia funkcji generacji zaczyna się od słowa „for” następnie podaje się zakres indeksowania w generacji (coś w rodzaju pętli). Zakres będzie biegł w górę. Zmienna „i” jest zmienną iteracyjną. Tej zmiennej nie definiuje się w VHDL-u. Powołuje się ją ad hoc na potrzeby funkcji generacji. Jest to zmienna całkowitoliczbowa. Po słowie kluczowym „generate” piszemy blok instrukcji współbieżnych które zostaną powielone. Na końcu należy przypisać sygnały wejścia i wyjścia z łańcucha przeniesień:

```

        ( A(i) and Cint(i) ) or ( B(i) and Cint(i) );
    end generate;
    Cint( 0 ) <= CI;
    CO <= Cint( 8 );
end Dataflow;

```

Architektura taka składa się z trzech instrukcji: instrukcji generacji oraz dwóch instrukcji przypisania współbieżnego. Instrukcja generacji generuje 8 bloków składających się z instrukcji przypisania.

Druga wersja instrukcji generacji jest wersją warunkową:

```

label: if condition generate      -- label required
    block_declarative_items      \ optional
begin
    concurrent_statements
end generate label;

```

Jeśli warunek jest spełniony, to generowany jest blok instrukcji współbieżnych. Czasami zagnieżdża się tą instrukcją w iteracyjnej instrukcji generacji. Etykieta (poprzednio „lb”, teraz „label”) jest obowiązkowa. Jeżeli zapomnimy o etykietce, to kompilacja i synteza projektu nie powiedzie się.

Opisy sekwencyjne

Instrukcja procesu

Sama instrukcja procesu jest współbieżna. Hierarchia jest taka że mamy jednostkę, mamy architekturę, wewnątrz architektury umieszczamy instrukcję procesu, wewnątrz instrukcji procesu możemy umieszczać instrukcje sekwencyjne. Wszystkie instrukcje sekwencyjne muszą znajdować się jakimś procesie, nie można ich umieszczać na poziomie architektury. Na poziomie architektury można umieszczać tylko instrukcje współbieżne. Proces jest taką instrukcją współbieżną a w procesie może znajdować się ciąg instrukcji sekwencyjnych. Składnia procesu jest następująca:

```

[label:] process [ ( sensitivity_list ) ] [ is ]
    [ declarative_items ]
begin
    sequential_statements
end process [ label ];

```

Etykieta jest opcjonalna. Po słowie kluczowym „process” znajduje się lista wrażliwości (lista czułości) procesu. Jest to lista sygnałów, których zmiana wartości powoduje wykonanie procesu. Procesy są wyzwalane w chwili wykrycia zmiany wartości sygnału z listy wrażliwości. Lista ta jest opcjonalna. Gdy jej nie ma, to w procesie muszą jawnie wystąpić instrukcje „wait”. Można wstrzymywać wykonanie procesu dzięki poleceniom „wait” lecz jest to niezalecane dla procesu syntezy. Przydaje się to przy generowaniu pobudzeń w testbenchach.

Wewnątrz procesu można używać instrukcji sekwencyjnych. Są nimi:

- Instrukcja wstrzymania („wait”)
- Instrukcja asercji
- Instrukcja raportu (generuje komunikaty diagnostyczne podczas pracy symulatora)
- Instrukcja sekwencyjnego przypisania sygnału („<=”)
- Instrukcja przypisania zmiennej („:=”) Zmienne istnieją tylko wewnątrz procesów. Nie ma ich na poziomie

architektury. Wyjątkiem są zmienne dzielone [odpowiedniki zmiennych globalnych], ale są to obiekty niewidoczne dla syntezy)

- Instrukcja wywołania procedury (nie będzie omawiana)
- Instrukcja warunkowa „if”
- Instrukcja wyboru „case”
- Instrukcje „next”, „exit”, „return”
- Instrukcja pusta „null” (odpowiednik znaku średnika z C++ lub Javy)

Instrukcja przypisania sygnału

Notacja jest taka sama jak przy instrukcji współbieżnego przypisania z tym wyjątkiem, że w procesach nie można używać przypisywania warunkowego i selektywnego. Można używać przypisania prostego z „after”. Przykładowo:

```
process (We1, We2) is
  begin
    Wy <= We1 and We2 after 4 ns;
  end;
```

Standard VHDL opisując pracę instrukcji współbieżnych przenosi wszystko do procesów. Każda prosta instrukcja przypisania jest przekładana na proces, który jest jawnie wyzwalany sygnałami występującymi po prawej stronie instrukcji przypisania.

Nielegalne są przypisania warunkowe oraz selektywne. Błędem byłoby:

```
process (We1, We2) is
  begin
    Wy <= We1 when We2 = '1'
      else '0';
  end;
```

Instrukcja przypisania zmiennej

Najpierw musimy sobie stworzyć zmienną. Zmienne tworzymy na poziomie procesu:

```
process (A, B) is
  variable Var1, Var2:STD_LOGIC := 'u';
  begin
    . . .
    Var1 <= '0';
    . . .
    Var2 <= Var1 xor '1';
    . . .
  end;
```

Po słowie kluczowym „variable” występuje lista nazw zmiennych i po dwukropku nazwa typu zmiennych (tutaj STD_LOGIC). Później można jeszcze przypisać wartości początkowe (w przykładzie przypisano 'u'). Zmienne tworzymy tak samo jak sygnały (w sensie zapisu) ale zmienne tworzy się **tylko** na poziomie procesów. Później można użyć jakichś instrukcji przypisania zmiennych (jest to natychmiastowe).

Instrukcja „wait”

Proces wykonuje się sekwencyjnie i jego wykonanie można wstrzymać do określonego momentu czasowego:

```
wait for 10 ns;           -- timeout
wait until clk='1';       -- warunek logiczny
wait until A>B and S1 or S2;
wait on sig1, sig2;       -- lista wrażliwości
```

Wstrzymanie może na określony odcinek czasu („wait for”), albo tak długo aż zostanie spełniony warunek logiczny („wait until”), albo może być podana jawna lista wrażliwości. Wtedy wstrzymanie trwa do chwili zmiany wartości sygnału z listy wrażliwości („wait on”). Instrukcje „wait on” są syntezywalne i niektóre narzędzia pozwalają na ich stosowanie.


Instrukcja warunkowa „if”

Składnia jest standardowa. Etykieta jest opcjonalna. Konieczne jest stosowanie zakończenia w postaci „endif”:

```
[ label: ] if condition1 then
    statements
elseif condition2 then \ optional
    statements
...
else \ optional
    statements
end if [ label ] ;
```

Poniżej po lewej stronie mamy opisany multiplekser 4 na 1 z jawnie wyróżnionymi 4-ma kombinacjami sterującymi i sprowadzający inne pobudzenia ('W' 'L' 'H' ...) do stanu nieznanego 'X'. Zapis pokazany po lewej jest równoważny procesowi opisanemu po prawej stronie:

```
architecture DF of MUX_4 is
begin
  Y <= A when Sel = "00" else
    B when Sel = "01" else
    C when Sel = "10" else
    D when Sel = "11" else
    'X';
end DF;
```




```
architecture DF_Eq of MUX_4 is
begin
  process ( Sel, A, B, C, D )
  begin
    if Sel = "00" then
      Y <- A;
    elsif Sel = "01" then
      Y <- B;
    elsif Sel = "10" then
      Y <- C;
    elsif Sel = "11" then
      Y <- D;
    else
      Y <= 'X';
    end if;
  end process;
end DF_Eq;
```

Proces składa się z kaskady warunków. Wygląda to podobnie jak przy instrukcji przypisania warunkowego. Na liście wrażliwości procesu będą występowały wszystkie sygnały, które znajdowały się po prawej stronie polecenia przypisania współbieżnego (sygnały wejściowe A, B, C, D oraz sygnał Sel). Warunki nie muszą być rozłączne. Spełnienie jakiegoś warunku implikuje to, że żaden inny nie jest spełniony.

[Skok przez plot do slajdu 25]

Znamy procesy, znamy listy wrażliwości, znamy instrukcje warunkowe, więc znamy wszystkie mechanizmy potrzebne do opisywania przerzutników. W VHDL nie ma czegoś takiego, jak deklaracja że dany sygnał jest sygnałem synchronicznym (przypisywanym np. do narastającego zbocza sygnału zegarowego). Opisy sygnałów synchronicznych wymagają specjalnych konstrukcji. Musi istnieć uzależnienie przypisania sygnału od zbocza narastającego sygnału zegarowego. Jak w VHDL-u opisać przerzutnik typu D? Przykład znajduje się po lewej stronie obrazka:

```
entity DFF is
port ( D : in STD_LOGIC;
      Clk : in STD_LOGIC;
      Q : out STD_LOGIC );
end DFF;
architecture RTL of DFF is
begin
  process ( Clk )
  begin
    if Clk'Event and Clk = '1' then
      Q <= D;
    end if;
  end process;
end architecture;
```



```
entity DFF is
port ( D : in STD_LOGIC;
      Clk : in STD_LOGIC;
      Q : out STD_LOGIC );
end DFF;
architecture Behavioral of DFF is
  signal Q_int;
begin
  Q <= Q_int;
  process ( Clk )
  begin
    if Clk'Event and Clk = '1' then
      Q_int <= D;
    end if;
  end process;
end architecture;
```

Będziemy potrzebowali procesu wyzwalanego zmianą sygnału „Clk”. Konieczna jest instrukcja warunkowa, testująca czy mamy do czynienia z narastającym zboczem zegarowym. Jest to złożone wyrażenie logiczne:

if Clk'Event and Clk = '1' then . . .

„Event” to atrybut mający wartość „TRUE” gdy sygnał po jego lewej stronie zmieni wartość w bieżącym cyklu symulacji. Drugi warunek jest konieczny do wykrycia zbocza narastającego. Jeżeli chcielibyśmy by przerzutnik reagował na zbocze opadające, to zamiast **and Clk = '1'** trzeba napisać **and Clk = '0'**. Działanie będzie poprawne gdy „Clk” będzie sygnałem typu BIT. W naszym przypadku „Clk” jest typem sygnału o 9 wartościach, ale nikt nie grzebie się w szczegółach, bo sygnały zegarowe z założenia są dwuwartościowe.

Proces jest wyzwalany tylko przy pomocy „Clk”. Zmiany sygnału na wejściu „D” nie powodują przypisania nowej wartości sygnału Q. Tylko zbocze sygnału zegarowego powoduje zmianę stanu przerzutnika. Dlatego na

liście wrażliwości nie ma innych sygnałów niż „Clk”. Jest to konieczne, by pokazać że uzależniamy się od sygnału zegarowego.

W drugim przykładzie mamy przedstawiony opis przerzutnika typu T (prawa strona):

```

entity TFF is
  port ( T : in STD_LOGIC;
         Clk : in STD_LOGIC;
         Q : out STD_LOGIC );
end TFF;

architecture RTL of TFF is
  signal Q_int : STD_LOGIC := '0';
begin
  Q <= Q_int;
  process ( Clk )
  begin
    if Clk'Event and Clk = '1' then
      if T = '1' then
        Q_int <= not Q_int;
      end if;
    end if;
  end process;
end architecture;

```

Różnica jest taka, że przerzutnik T zmienia swoją wartość tylko wtedy gdy na wejściu T jest stan '1'. Konieczne jest powielenie portu wyjściowego w postaci sygnału wewnętrznego i dlatego opis jest nieco dłuższy. Definiujemy sygnał wewnętrzny „Q_int”, dodajemy instrukcję przypisania współbieżnego mówiącą o tym, że na porcie wyjściowym ma się znaleźć to samo, co na sygnale wewnętrznym „Q_int”. Potem można dodać blok z warunkiem.

Ponieważ notacje `if Clk'Event and Clk = '1'` występują dość często, to w pakiecie `STD_LOGIC_1164` zdefiniowano funkcje, które uwzględniają specyfikę logiki 9-wartościowej:

```

• Pakiet STD LOGIC 1164:
function rising_edge (signal s : STD_ULOGIC) return BOOLEAN;
function falling_edge (signal s : STD_ULOGIC) return BOOLEAN;
if Clk'Event and Clk = '1' then... => if rising_edge(Clk) then...

```

Należy z tych funkcji („rising_edge” oraz „falling_edge”) korzystać. Są one bezpieczniejsze w używaniu i ich używanie jest bardziej czytelne.

Opis przerzutników można rozszerzyć o sygnał Clock Enable. Proces zależy od o sygnału zegarowego (nie zależy od CE). Sygnał na CE sprawdzany jest po wykryciu zbocza narastającego sygnału zegarowego. Jeżeli na CE jest 0, to nic się nie dzieje (lewa strona obrazka):

```

entity DFF_E is
  port( D : in STD_LOGIC;
        CE : in STD_LOGIC;
        Clk : in STD_LOGIC;
        Q : out STD_LOGIC );
end DFF_E;

architecture RTL of DFF_E is
begin
  process ( Clk )
  begin
    if rising_edge( Clk ) then
      if CE = '1' then
        Q <= D;
      end if;
    end if;
  end process;
end architecture;

```

```

entity DFF_CE is
  port( D : in STD_LOGIC;
        Clr : in STD_LOGIC;
        CE : in STD_LOGIC;
        Clk : in STD_LOGIC;
        Q : out STD_LOGIC );
end DFF_CE;

architecture RTL of DFF_CE is
begin
  process ( Clk, Clr )
  begin
    if Clr = '1' then
      Q <= '0';
    elsif rising_edge(Clk) then
      if CE = '1' then
        Q <= D;
      end if;
    end if;
  end process;
end architecture;

```

Wesalej się robi gdy mamy do czynienia z asynchronicznymi sygnałami ustawiania lub kasowania. Po prawej stronie obrazka mamy kod przerzutnika z dodatkowym wejściem asynchronicznego kasowania. Na liście wrażliwości musi pojawić się sygnał „Clr”, bo jego wartość aktywna ma wyzerować natychmiast przerzutnik. Sygnał ten ma priorytet większy od „Clk”. Proces może wykonać się w trzech przypadkach (zmiana obu sygnałów, zmiana tylko „Clk” lub zmiana tylko „Clr”).

Nieco inaczej sprawy się mają gdy mamy do czynienia z synchronicznym sygnałem resetującym. Na liście wrażliwości procesu nie ma sygnału „Reset”. Sygnał „Reset” jest próbkowany tylko w momentach wystąpienia zbocza narastającego sygnału zegarowego. Tutaj „Reset” ma większy priorytet od wejścia „D”. Gdy „Reset” ma wartość aktywną, to następuje wyzerowanie sygnału „Q”. Jeżeli „Reset” jest nieaktywny, to sprawdza się jaką wartość ma sygnał „D”:

```

entity DFF_RE is
  port( D   : in  STD_LOGIC;
        Rst : in  STD_LOGIC;
        CE  : in  STD_LOGIC;
        Clk : in  STD_LOGIC;
        Q   : out STD_LOGIC );
end DFF_RE;

architecture RTL of DFF_RE is
begin
  process ( Clk )
  begin
    if rising_edge( Clk ) then
      if Rst = '1' then
        Q <= '0';
      elsif CE = '1' then
        Q <= D;
      end if;
    end if;
  end process;
end architecture;

```

Umieszczanie w treści architektury różnych konstrukcji odpowiada rysowaniu na schemacie różnych fragmentów projektu cyfrowego. Jeżeli mamy do stworzenia projekt zawierający przerzutnik z podłączoną na jego wejście logiką kombinacyjną.

W architekturze tworzymy sygnały wewnętrzne:

```
architecture ...
```

```

. . .
signal D1 : STD_LOGIC;

```

Sygnały kombinacyjne nie muszą mieć podanych wartości początkowych. W opisie należy dodać port wyjściowy:

```
signal Q1 : STD_LOGIC := '0';
```

Sygnałom synchronicznym należy przypisywać wartość początkową. Jeżeli nie przypiszemy wartości początkowej sygnałowi „Q1”, to będzie on miał wartość 'U', a to może nieźle namieszać w symulacji. Na syntezę nie będzie miało to żadnego wpływu.

Później opisuje się bloki z których składa się architektura. Przykładowo zaczynamy od opisu części kombinacyjnej:

```

begin
D1 <= (We1 and We2) or ( ... );

```

Niżej można przypisać jakąś wartość do „Q1”. Możemy używać przypisań różnego rodzaju. Po załatwieniu części kombinacyjnej umieszczamy w treści architektury procesy odpowiedzialne za działanie przerzutnika:

```

process (clk) is
begin
. . .
end process;

```

Proces będzie wyzwalany od sygnały „clk”. Tak jak poprzednio umieszczamy odpowiednie opisy zależnie od rodzaju przerzutnika który chcemy umieścić. Narzędzie XST potrafi rozpoznać rejestry, liczniki, różnego rodzaju układy akumulacyjne. W „XST User Guide” w rozdziale 2 znajdują się wskazówki jak należy w VHDL opisywać standardowe bloki tak, by narzędzie XST sobie poradziło z poprawną synteza.

[A teraz wielki powrót do slajdu 21]

Instrukcja wyboru „case”

```

[ label: ] case expression is
  when choice1 =>
    statements
  when choice2 => \_ opt.
    statements    /
  ...
  when others => \_ opt. if all choices
    statements  / covered
end case [ label ] ;


```

Instrukcja przypomina tą, znaną z C++. Obliczana jest wartość wyrażenia, która jest porównywana z opcjami wyboru „choice1”, „choice2”. **Opcje wyboru muszą być rozłączne.** Muszą być kompletne. Występuje opcja

„others” zawierająca pozostałe możliwości. Notacja „=>” pełni tylko rolę składniową i nie jest jakkolwiek instrukcją przypisania. Nie ma czegoś takiego jak „wyskakiwanie” z opcji („break” znany z C++ i Javy), ponieważ po wykonaniu się danej opcji wychodzimy z instrukcji „case”.

Instrukcja „case” jest równoważna selektywnemu przypisywaniu:

<pre>architecture DF2 of MUX is begin with Sel select Y <= A when "00", B when "01", C when "10", D when "11", 'X' when others; end DF2;</pre>	<pre>architecture DF2_Eq of MUX_4 is begin process (Sel, A, B, C, D) begin case Sel is when "00" => Y <= A; when "01" => Y <= B; when "10" => Y <= C; when "11" => Y <= D; when others => Y <= 'X'; end case; end process; end DF2_Eq;</pre>
---	--



1

Jeśli mamy multiplekser 4 na 1 opisany funkcją przypisania selektywnego, to jest on z definicji równy procesowi w którym występuje jedna instrukcja „case”. Lista wrażliwości zawiera wszystkie sygnały występujące z prawej strony wyrażenia przypisania oraz sygnał selekcji.

Czasami zamiast „others” należy używać instrukcji „null” (jeżeli nie chcemy nic robić).