Pakiet „standard" c.d.

```
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;
```

Na poprzednim wykładzie zostały omówione typy liczb całkowitych do podtypu liczb dodatnich włącznie. Istnieje typ zmiennoprzecinkowy REAL. Obejmuje swoim zakresem liczby podwójnej precyzji. Typ boole'owski jest zdefiniowany jako typ wyliczeniowy (enumeracja). Typ znakowy CHARACTER również definiuje się jako wyliczeniowy. Stałe znakowe w języku VHDL zapisuje się w pojedynczych nawiasach.

STRING jest typem przechowującym napisy. Używa się tutaj wektora. Może on być wektorem z otwartym zakresem (w nawiasach ostrych nie ma konkretnej liczby określającej długość przechowywanej danej.). Typ TIME jest typem fizykalnym, co oznacza że ma on zdefiniowane jednostki. Jednostką bazową dla TIME jest femtosekunda (10^{-15} sekundy). Typu tego używamy głównie do realizowania procesu symulacji działania układów cyfrowych. Ważne jest, by stosować odstęp między wartością a jednostką czasu. DELAY_LENGHT jest podtypem czasowym.

Typ BIT i napisy bitowe

Typ BIT jest typem wyliczeniowym:

```
type BIT is ('0', '1');
```

W VHDL'u nie ma fizykalnego typu, który mógłby reprezentować stany cyfrowe. Symulacja opiera się na stałych znakowych. Wartości zmiennych typu BIT traktujemy jako wartości sygnałów logicznych.

```
type BIT_VECTOR is array (NATURAL range <>) of BIT;
```

Z typem wektora bitowego spotkaliśmy się na poprzednim wykładzie. Istnieje on jako tablica, której rozmiar określony jest liczbą naturalną. Powyżej pokazana definicja wektora jest tzw. zakresem otwartym indeksu tablicy. Indeksy opisujące poszczególne bity wektora w VHDL mogą być ułożone w kolejności rosnącej lub malejącej. Zapis:

```
signal DataBus  : BIT_VECTOR( 7 downto 0);
```

będzie oznaczał indeksowanie „w standardzie inżynierski", czyli z malejącym indeksem. Zapis:

```
signal DataBus : BIT_VECTOR( 0 to 7);
```

oznacza indeksowanie z indeksem rosnącym.

Czym są napisy bitowe? Jest to specyficzny rodzaj napisów, które umieszcza się w cudzysłowach podwójnych. Napisy bitowe są specyficznym rodzajem napisów. Ich elementami mogą być wyłącznie zera i jedynki. Poniżej przedstawiono różne możliwości przypisania stałej bitowej do sygnału DataBus:

```
DataBus <= "10000000";
DataBus <= B"1000_0000";
DataBus <= X"80";
```

Kod powyżej reprezentuje napis bitowy z podaną podstawą liczbową. Jeżeli stosujemy zapis z podaniem podstawy, możemy korzystać ze znaków podkreślenia, które będą pełniły rolę separatorów przy oddzielaniu tetrad bitowych (by polepszyć widoczność). Należy uważać na to, że zapisem heksadecymalnym można przedstawiać napisy bitowe, których długość jest wielokrotnością czwórki.

Kolejne linijki są przykładami konstruowania agregatów. Tutaj podaje się wartości na poszczególnych pozycjach:

```
DataBus <= ( '1', '0', '0', '0', '0', '0', '0', '0' );
DataBus <= ( '1', others => '0' );
```

Zamiast wypisywać wszystkie pozycje bitowe osobno, można wypisać tylko pierwszą, użyć słowa kluczowego „others" i po nim wpisać wartość na pozostałych pozycjach. Agregaty przydają się gdy chcemy wyzerować jakiś wektor bitowy. Używamy do takiego celu zapisu:

```
DataBus <= ( others => '0' );
```

W agregacie można wskazywać pozycję w sposób jawny. Gdy chcemy na siódmym bicie wpisać jedynkę i na pozostałych bitach zero, posługujemy się kodem:

```
DataBus <= ( 7 => '1', others => '0' );
```

Jeżeli chcemy by na bitach 7 i 3 były jedynki, a zera na pozostałych pozycjach, można użyć składni:

```
DataBus <= (3|7 => '1', others => '0');
```

W wektorze bitowym możemy się odwoływać do jego fragmentu (przykładowo połowa bajtu):

```
HalfByte <= DataBus( 7 downto 4 );
```

Można się odwoływać również do pojedynczych elementów w wektorze bitowym:

```
MSB <= DataBus( 7 );
```

Operatory

Większość z nich działa na wartościach numerycznych. Pogrupowane zostały według priorytetów wykonywania (od najwyższego do najniższego). Operatory w grupach mają jednakowy priorytet wykonywania. Łączność w języku VHDL jest zawsze lewostronna. Nie ma tutaj zwykłego operatora przypisania.

Listę otwierają operatory potęgowania, wyznaczania wartości bezwzględnej oraz wyznaczania negacji logicznej:

```
**     exponentiation,       numeric ** integer,   result numeric
abs    absolute value,       abs numeric,          result numeric
not    complement,           not logic or boolean, result same
```

Kolejne operatory opisują mnożenie oraz dzielenie. Polecenie „remainder” służy do wyznaczania reszty z dzielenia:

```
*      multiplication,       numeric * numeric,    result numeric
/      division,             numeric / numeric,    result numeric
mod    modulo,               integer mod integer,  result integer
rem    remainder,            integer rem integer,  result integer
```

Poniżej przedstawione zostały jednoargumentowe operatory zmiany znaku. Operatory można przeciążać.

```
+      unary plus,           + numeric,            result numeric
-      unary minus,          - numeric,            result numeric
```

Kolejnymi operatorami są dodawanie, odejmowania oraz konkatenacja:

```
+      addition,             numeric + numeric,    result numeric
-      subtraction,          numeric - numeric,    result numeric
&      concatenation,        array or element,     result array
```

Ostatni operator jest użyteczny podczas przekształcania napisów bitowych lub innych obiektów będących wektorami. O konkatenacji powiemy nieco później. Kolejne operatory służą do realizowania obrotów i przesunięć na macierzach lub wektorach:

```
sll    shift left logical,   log. array sll integer,       result same
srl    shift right log.,     log. array srl integer,       result same
sla    shift left arith.,    log. array sla integer,       result same
sra    shift right arith.,   log. array sra integer,       result same
rol    rotate left,          log. array rol integer,       result same
ror    rotate right,         log. array ror integer,       result same
```

Grupa poniżej reprezentuje operatory porównywania:

```
=      equality,                                  result boolean
/=     inequality,                                result boolean
<      less than,                                 result boolean
<=     less than or equal,                        result boolean
>      greater than,                              result boolean
>=     greater than or equal,                     result boolean
```

Należy zwrócić uwagę na operator równości, który składa się z jednego znaku równości. Przyzwyczajeni jesteśmy do faktu, że znak „=” w językach programowania reprezentował sobą przypisanie. W VHDL przypisanie realizuje się przez zapis „:=”.

Najniższy priorytet mają operatory logiczne:

```
and    logical and,          log. array or boolean,       result same
or     logical or,           log. array or boolean,       result same
nand   logical nand,         log. array or boolean,       result same
nor    logical nor,          log. array or boolean,       result same
xor    logical xor,          log. array or boolean,       result same
xnor   logical xnor,         log. array or boolean,       result same
```

Wszystkie mają ten sam priorytet. Nie obowiązują tutaj zasady rachunku logicznego, gdzie AND miał większy priorytet od OR. W języku VHDL nie można łączyć ze sobą różnych operatorów logicznych. Zapis:

```
C <= A and B and C;
```

nie będzie błędny, ponieważ operator iloczynu jest asocjacyjny. Można go kaskadowo łączyć. Nie można mie-

szać ze sobą różnych operatorów w jednym wyrażeniu logicznym. Kod:

```
D <= E and B or C;
```

będzie zapisem błędnym. Oczywiście jeżeli powyższy zapis przekształcony zostanie na:

```
D <= (E and B) or C;
```

to będzie on prawidłowy. Używanie wyrażeń złożonych nasuwa za sobą konieczność jawnego używania nawiasów. Prawdopodobnie ma to służyć jawnemu pokazywaniu priorytetów wykonywania poszczególne operatorów logicznych.

VHDL jest rygorystyczny. Można ze sobą łączyć operatory AND, OR, XOR oraz XNOR. Nie można łączyć ze sobą NOR-ów i NAND-ów, ponieważ nie są one operatorami asocjacyjnymi.

Powiemy teraz więcej o operatorze konkatenacji:

```
signal ASCII : BIT_VECTOR ( 7 downto 0);
signal Digit : BIT_VECTOR ( 3 downto 0);
```

Mamy dwa sygnały: 8-bitowy „ASCII” oraz 4-bitowy „Digit”. Aby dla danej cyfry dziesiętnej można było utworzyć kod ASCII, to należy dokleić na starszą tetradę trójkę, a na młodszą podać wprost zakodowaną cyfrę:

```
ASCII <= "0011" & Digit;  -- X"3" & Digit;
```

Można powyższą operację opisać inaczej, odwołując się do pojedynczych bitów wektora:

```
ASCII <= X"3" & Digit ( 3 ) & Digit ( 2 ) &
                  Digit ( 1 ) & Digit ( 0 );
```

Jeżeli mamy jakiś wektor i chcemy go przesunąć w prawo, to podstawiamy wektor mający na najstarszej pozycji zero a pozostałe bity bierzemy z wektora „ASCII”:

```
-- Shift right (this must be synchronous!):
ASCII <= '0' & ASCII( 7 downto 1 );
```

Przypisanie sygnału opisane powyżej musi być synchroniczne. Tak będziemy opisywać pracę rejestrów przesuwnych, które muszą działać synchronicznie, bo asynchroniczna praca takich układów byłaby bezsensowna.

Analogicznie do poprzedniego przykładu będzie działało przesuwanie w lewo:

```
-- Shift left:
ASCII <= ASCII( 6 downto 0 ) & '0';
```

Pakiet „stdlogic”

Wspomniano wcześniej, że typ BIT jest dwuwartościowym typem abstrakcyjnym. Dla rzeczywistego sygnału logicznego dwa stany to za mało. Typ, który opisywałby właściwości sygnałów cyfrowych, został opisany w standardzie 1164. Pakiet dołączany jest automatycznie do każdego nowego modułu, opisywanego w VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

W tym pakiecie zdefiniowano typ wyliczeniowy 9-wartościowy, który będzie reprezentatywnie przedstawiać zachowanie się sygnałów w układach cyfrowych:

```
type STD_ULOGIC is ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak     Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'   -- Don't care
                   );
```

Typ ten składa się ze stałych znakowych, co pozwala na łatwe stosowanie ich w napisach bitowych i wektorach. Synteza i symulacja układów cyfrowych opiera się na tym typie. Z punktu widzenia jest to świat, w którym analizuje się stałe znakowe. Nie ma żadnej zaszytej implementacji fizykalnej tych stanów.

'U' – reprezentuje sygnał nie zainicjalizowany. Z takimi wartościami startuje symulacja. Wartości ich są nieznane.

'X', '0', '1' – grupa sygnałów „silnych” (silna wartość nieznana, silne/wymuszone zero i silna jedynka). Wartość nieznana pojawia się w sytuacji konfliktu sygnałów (gdy na jedną linię zostaną podane równocześnie stany '0' i '1').

'Z' – oznacza stan wysokiej impedancji. Nie ma czegoś takiego jak silna (lub słaba) impedancja.

'W', 'L', 'H' – grupa sygnałów o „słabych” wartościach. Pojawiają się one „w sąsiedztwie” rezystorów Pull-Up

(daje słabą jedynkę) oraz Pull-Down (daje słabe zero). Podanie tych sygnałów na jedną linię równocześnie wygeneruje stan 'W'.

'-' – to wartość, której interpretacja jest specyficzna. Nigdy nie jest ona generowana podczas pracy układu. Może być ona wpisana przez nas.

Omówiony zestaw 9-ciu stanów stał się standardem przemysłowym i na tym zestawie wszyscy pracują. Zdefiniowany został typ wektorowy:

```
type STD_ULOGIC_VECTOR is array ( NATURAL range <> ) of
STD_ULOGIC;
```

Nie pracuje się na typie BIT (może być on używany w specyficznych przypadkach). Korzysta się z ULOGIC. Typ ten pociągnął za sobą konieczność opracowania funkcji rozstrzygającej oraz tabeli zdefiniowanej w standardzie:

```
function resolved ( s : STD_ULOGIC_VECTOR ) return STD_ULOGIC;

constant resolution_table : stdlogic_table := (
--      ---------------------------------------------------------
--        'U'   X    0    1    Z    W    L    H    -        |   |
--      ---------------------------------------------------------
        ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
        ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
        ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
        ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
        ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
        ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
        ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
    );
```

Nigdzie nie jest generowana wartość „Don't care”. Odpowiedziami dla silnych sygnałów są albo sygnały '0' i '1' lub 'X'. Tablica jest symetryczna względem przekątnej. Stan wysokiej impedancji powiela sygnał z drugiego sterownika (za wyjątkiem sygnału '-').

Typ STD_LOGIC jest typem STD_ULOGIC z dołączoną funkcją rozstrzygającą:

```
-------------------------------------------------------------------
-- *** industry standard logic type ***
-------------------------------------------------------------------
subtype STD_LOGIC is resolved STD_ULOGIC;
type STD_LOGIC_VECTOR is array ( NATURAL range <>) of STD_LOGIC;
```

Tego typu należy używać przy opisie układów cyfrowych. Nie używa się BIT-u.

Pozostała jeszcze kwestia przeciążonych operatorów logicznych. Rozwiązano to przy użyciu tablic, które opisują działanie operatorów:

```
constant and_table : stdlogic_table := (
--      ---------------------------------------------------
--      |  'U'   X    0    1    Z    W    L    H    -
        ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ),  --   U
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  --   X
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  --   0
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  --   1
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  --   Z
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  --   W
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  --   L
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  --   H
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )   --   -
);

constant or_table : stdlogic_table := (
--      ---------------------------------------------------
--      |  'U'   X    0    1    Z    W    L    H    -
        ( 'U', 'U', 'U', '1', 'U', 'U', 'U', '1', 'U' ),  --   U
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ),  --   X
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  --   0
        ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ),  --   1
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ),  --   Z
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ),  --   W
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  --   L
        ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ),  --   H
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' )   --   -
```

Instrukcje współbieżne

Są to instrukcje, które mogą wystąpić w ciele architektury (między słowami kluczowymi „begin” i „end” określajacymi treść architektury). Ich kolejność jest nieistotna. Wystąpić mogą instrukcje:

- Współbieżne przypisanie sygnału („<=”)
- Instancja komponentu
- Instrukcja generacji („generate”)
- Instrukcja procesu („process”)
- Instrukcja bloku (nie będzie omawiana)

- Współbieżne wywołanie procedury
- Współbieżne obliczenie asercji

Asercję używa się do przedstawiania warunków, które muszą być spełnione by program działał poprawnie. Gdy asercja nie jest spełniona, to wywoływana może być jakaś funkcja systemowa przerywająca działanie programui wydrukowanie jakiegoś komunikatu. Asercje w VHDL są na bieżąco współbieżnie sprawdzane. Nie spełniona asercja w VHDL może generować ostrzeżenie, błąd lub przerwać działanie symulatora. Asercje nie są syntezowalne i służą tylko do debugowania modelu.

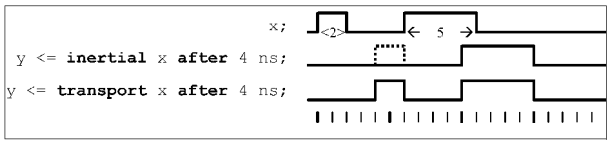
Przypisanie sygnału

Podstawowa operacja, którą poznaliśmy na pierwszym wykładzie z VHDL-a. X oraz Y muszą być sygnałami. Mogą być one również portami:

```
y <= x;
y <= ( x1 and x2 ) or ( ( x3 nand x4 ) xor x5 );
```

UWAGA: Przypisanie zmiennej opisuje się inną instrukcją. Przypisania zmiennych nie można umieszczać w ciele architektury. Zmienne istnieją wewnątrz procesów.

W języku VHDL na potrzeby symulacji stworzono dwa modele przypisania sygnałów. Wiążą się one z sytuacją gdy przypisujemy sygnał z podaniem czasu opóźnienia (po klauzuli „after”):



Modele opóźnienia to model inercyjny i transportowy. W modelu **inercyjnym** tłumione są (nie są przypisywane) sygnały X, które są krótsze od czasu propagacji. W przypadku powyżej pokazanym przeniesiony zostanie tylko impuls o szerokości 5 ns. W modelu **transportowym** przenoszone są wszystkie impulsy bez względu na ich długość. Domyślnie stosowany jest model inercyjny z zerowym czasem opóźnienia:

```
-- Domyślnie:
y <= x;              <=>  y <= inertial x after 0 ns;
y <= x after 4 ns;   <=>  y <= inertial x after 4 ns;
```

Klauzula „after” jest niesyntezowalna. Odgrywa rolę tylko podczas symulacji projektu. Możliwe jest opisywanie wielokrotnej zmiany w reakcji na pojedynczą zmianę sygnału X:

```
-- Wielokrotna zmiana w reakcji na pojedyńczą
-- zmianę sygnału:
y <= x after 4 ns, not x after 8 ns;
```

Przecinkami można oddzielać całą listę przypisań sygnałów z rosnącymi czasami opóźnień. Tego również się nie syntezuje. Oczywiście podczas symulacji się przydaje. Kod przedstawiony poniżej opisuje generację sygnału zegarowego:

```
-- W opisach sekwencyjnych (np. powtarzanych w pętlach):
Clk <= '1', '0' after ClkPeriod / 2;
```

„ClkPeriod” jest stałą typu DELAY_LENGHT.

Przypisanie warunkowe

Często używane w opisie układów kombinacyjnych. Posłużymy się multiplekserem 4 na 1:

```
entity MUX_4 is
        port( A, B, C, D : in STD_LOGIC;
              Sel : in STD_LOGIC_VECTOR( 1 downto 0 );
              Y : out STD_LOGIC );
end MUX_4;
```

Ma on 4 wejścia (A, B, C, D), jedno wyjście (Y) oraz wejście selekcji będące 2-bitowym wektorem. Składnia przypisania warunkowego:

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

Stosując przypisanie warunkowe trzeba być ostrożnym. Przypisanie to nie żąda wyczerpującej listy warunków (może mieć miejsce sytuacja, że żaden z warunków nie będzie spełniony). Wektor „Sel” jest wektorem 2-ch sy-

gnałów o 9 wartościach. Wypisane zostały tylko 3 możliwości (z 81). Pozostałe 78 zostało skomasowane przy przypisaniu D. Narzędzie syntezy zinterpretuje prawidłowo układ jako multiplekser. Symulator działa inaczej i tak opisany multiplekser może dla 78 możliwości do wyjścia Y zostanie przypisane wejście D. Aby się przed tym zabezpieczyć, należy skorygować kod:

```
. . .
C when Sel = „10” else
D when Sel = „11” else
'X';
```

Nie należy tu używać 'U' ani '-'.

Jeżeli lista warunków jest niewyczerpująca możliwości, to zostanie wygenerowany przez narzędzie zatrzask dla sygnału przypisywanego warunkowo. Gdy jeden z wypisanych warunków będzie spełniony, to zostanie przypisana nowa wartość sygnału. Gdy żaden z warunków nie zachodzi, to sygnał nie zmieni swojej wartości (zamykanie się zatrzasku). Pojawianie się zatrzasków projekcie bywa niebezpieczne, ponieważ burzy ono zależności czasowe. Dobrze jest pisać kompletne listy warunków w instrukcjach przypisywania warunkowego. Pojawienie się zatrzasków generuje ostrzeżenia w raporcie syntezy.

Instrukcja przypisywania warunkowego z „when” „else” działa tak, że wyszukiwany jest pierwszy warunek, który jest spełniony (**ważna jest kolejność warunków**), wobec czego każdy warunek na pozostałych pozycjach zawiera sobie negację wszystkich warunków poprzednich. Jeżeli narzędzia syntezy nie dostrzegą w opisie „gotowca” i synteza przebiega na piechotę w postaci bramek logicznych, to pociąga to za sobą dużą zasobożerność projektu. Instrukcję tą należy stosować z rozwagą. Wady tej nie ma

Przypisanie selektywne

Składnia polecenia jest inna:

```
architecture Dataflow2 of MUX_4 is
begin
    with Sel select
        Y <= A when "00",
             B when "01",
             C when "10",
             D when others;
end Dataflow2;
```

15

Instrukcja jest bezpieczniejsza, ponieważ jej składnia wymaga kompletnej listy warunków, które wzajemnie się wykluczają. Jeżeli lista warunków nie jest wyczerpująca, to należy używać słowa kluczowego „others”. Ponieważ warunki są rozłączne **nie jest istotna kolejność opcji** (za wyjątkiem „others”, która jeżeli występuje, to musi być ostatnia).