Dziś zakończymy omawianie zagadnień związanych z opisem sekwencyjnym. Należy wspomnieć o tym, że w języku VHDL każdą jednostkę opisujemy przy pomocy architektury. Zamieszczone tam instrukcje są wykonywane współbieżnie. Instrukcjami takimi są instrukcje przypisania, generacji oraz instrukcje procesu. Wewnątrz procesów zawarte są instrukcje wykonywane w sposób sekwencyjny. Mówiliśmy o tym, że procesy mogą mieć jawnie podaną listę wrażliwości (lista sygnałów, których zmiana powoduje wykonanie procesu) lub jawnie podane instrukcje „wait" wstrzymujące działanie procesu „aż coś się stanie". Poruszono również temat instrukcji warunkowych. Odpowiednio użyte instrukcje warunkowe odpowiadają przypisaniu selektywnemu. Należy pamiętać o tym że według standardu każda instrukcja przypisania współbieżnego (występująca w architekturze) jest zamieniana na proces wyzwalany wszystkimi sygnałami występującymi po prawej stronie instrukcji przypisania współbieżnego.

Dzisiaj omówione zostaną kolejne instrukcje wykorzystywane w opisie sekwencyjnym.

Pętle (Iteracje)

W języku VHDL są trzy instrukcje iteracyjne:

```
[ label: ] loop
               statements   -- use exit to abort
           end loop [ label ] ;

[ label: ] for variable in range loop
               statements
           end loop [ label ] ;

[ label: ] while  condition  loop
               statements
           end loop [ label ] ;
```

Pierwsza pętla jest pętlą nieskończoną. Zestaw instrukcji między „loop" oraz „end loop" będzie wykonywany w nieskończoność. Z pętli tej musimy „wydostać się" w sposób jawny używając instrukcji „exit".

Składnia drugiej pętli przypomina tą z poznanych już języków programowania. Różnica polega na zdefiniowaniu „na poczekaniu" zmiennej iterującej „variable" oraz zakresu zmian tej zmiennej. Definicja zakresu w postaci `0 to 7` spowoduje wzrost wartości zmiennej z każdym „przejściem" pętli. Zapis `7 downto 0` będzie oznaczał odliczanie od 7 w dół.

Ostatnia pętla jest pętlą „while" ze sprawdzaniem warunku na początku. Jeżeli chcielibyśmy używać pętli ze sprawdzaniem warunku na końcu, musielibyśmy użyć pętli pierwszej z jawnym użyciem instrukcji wyjścia w przypadku gdy jakiś warunek zostanie spełniony.

Etykiety w pętlach są opcjonalne.

Instrukcje „next" i „exit"

Instrukcja „next" odpowiada poleceniu „continue" z języka C. Oznacza przerwanie bieżącej iteracji i przejście do następnej. Instrukcja może mieć etykietę informującą o pętli, której ta operacja dotyczy (potrzebne gdy pętle są zagnieżdżone):

```
next;
next outer_loop;
next when A>B;
next this_loop when C=D or A>B;
```

Odpowiednikiem „break" z języka C jest „exit":

```
exit;
exit outer_loop;
exit when A>B;
exit this_loop when C=D or A>B;
```

Dotyczą jej te same uwagi co „next". Można wskazać która pętla zostanie przerwana. Spowoduje to wyjście poza pętlę i przejście do następnej instrukcji sekwencyjnej. Przerwanie można uzależnić od spełnienia jakiegoś warunku.

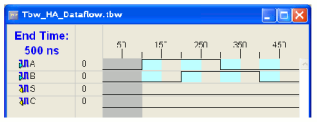
Posługiwanie się opisami sekwencyjnymi wiąże się z pewnym ryzykiem. Czasami takie opisy źle lub wcale się nie syntezują. By tego uniknąć, trzeba używać specyficznych konstrukcji które poprawnie zostaną rozpoznane przez oprogramowanie XST. Nie należy projektować układów w ten sposób, że w architekturze znajduje się instrukcja procesu, w której się „wszystko dzieje". To może działać poprawnie w symulacji, ale synteza może być niewykonywalna.

Instrukcji „return” (powrotu z funkcji/procedury) nie będziemy omawiać. Może się ona przydać podczas tworzenia testbenchy. O instrukcjach sprawdzania asercji i debugowania też nie będziemy mówić.

Jednostki symulacji

Opisy sekwencyjne występują w jednostkach symulacji. Języka VHDL używa się do opisywania środowiska symulacji. Jak to wygląda?

ISE Testbench




```
entity HA_Tbench is
end HA_Tbench;
architecture TB_Arch of HA_Tbench is
  file RESULTS: (...);
  component HalfAdder
      port (
          A : in STD_LOGIC;
          B : in STD_LOGIC;
          S : out STD_LOGIC;
          C : out STD_LOGIC
      );
  end component;
  signal A : STD_LOGIC := '0';
  signal B : STD_LOGIC := '0';
  signal S : STD_LOGIC := '0';
  signal C : STD_LOGIC := '0';
```

```
begin
  UUT : HalfAdder
  port map ( A => A,
             B => B,
             S => S,
             C => C );
  process
  begin
    -- Current TIME:  100ns
    wait for 100 ns;
    A <= '1';
    -- Current TIME:  200ns
    wait for 100 ns;
    B <= '1';
    -- Current TIME:  300ns
    wait for 100 ns;
    A <= '0';
    -- Current TIME:  400ns
    wait for 100 ns;
    B <= '0';
    -- --------------------
    wait for 100 ns;
    assert (...);-- NOT in v.9 ?!
  end process;
end TB_Arch;
```

Będziemy posługiwać się układem półsumatora. Układ ma 2 wejścia A i B oraz dwa wyjścia. Zwykle testbench generujemy rysując wykresy poprzez klikanie myszą. Wszystko dalej dzieje się na poziomie języka VHDL. To, co widzi ModelSIM wygląda tak, że jednostki dla symulacji nie mają zdefiniowanych wejść i wyjść. Definiowana jest jednostka `HA_Testbench`. Jednostka ta ma jedną architekturę w której dzieje się wszystko. Można zdefiniować plik służący do wyprowadzania rezultatów (**`file`** `RESULTS`) ale nie będziemy z tego korzystać. Najbardziej interesuje nas definicja komponentu. Powielone są tutaj porty, które są typu STD_LOGIC. Następna sekcja to definicja sygnałów, które nazywają się identycznie jak porty. W przypadku automatycznej generacji modelu sygnały te będą miały przypisane wartości początkowe w postaci zer.

Po prawej stronie slajdu znajduje się treść architektury. Składa się ona z dwóch instrukcji współbieżnych: instrukcji instancji komponentu oraz instrukcji procesu. Komponent przez nas symulowany jest wstawiany pod nazwą `UUT` (Unit Under Test). Będziemy się do tej nazwy odwoływać w ModelSIM-ie podczas poszukiwań sygnałów wewnętrznych lub portów symulowanego modułu. W instrukcji procesu zawarte są instrukcje odpowiedzialne za generowanie pobudzeń (przypisywanie wartości do portów wejściowych).

Struktura procesu jest trywialna. Nie posiada on listy wrażliwości. Jego działanie jest wstrzymywane w sposób jawny poprzez użycie instrukcji „wait”. Odpowiednie odcinki czasowe uzyskuje się dzięki `wait for 100 ns`. Po każdym wstrzymaniu jest wykonywane przypisanie.

Na końcu procesu wstawiało się instrukcję asercji, której zadaniem było zatrzymanie symulacji wyświetlenie komunikatu „This is not an error”. W starszych wersjach ISE było to stosowane zawsze. Od wersji 9 skasowano asercję. ModelSIM zawsze wykonuje symulację o długości 1µs po czym się zatrzymuje. Trzeba jawnie wydać polecenie „run” oraz wartość określająca jak długo będzie trwała symulacja.

Testbenche można pisać samodzielnie w języku VHDL. Wówczas wskazuje się jednostkę, która będzie symulowana. Spowoduje to wygenerowanie odpowiedniego szablonu z komponentem, sygnałami, instrukcją instancji na słowie `begin` (rozpoczynającym opis procesu) kończąc. Tak wygenerowany plik można modyfikować. Można dopisać instrukcję generującą sygnał zegarowy:

```
architecture
 . . .
begin
UUT:(...);
 . . .
Clk <= not Clk after 10 ns;
```

Da nam to sygnał zegarowy o częstotliwości 50 MHz o nieskończonej długości. Gdybyśmy chcieli dodać jeszcze jeden sygnał zegarowy, którego przebieg będzie przesunięty w fazie o 3 ns od poprzedniego, to potrzebne

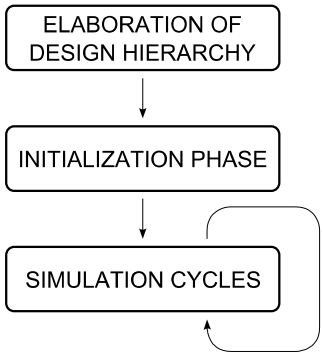
będzie stworzenie procesu:

```
process
begin
   wait for 3 ns;
   petla:
   Clk2 <= not Clk2;
   wait for 10 ns;
   end petla;
end process;
```

Opisywanie długich testbenchy przy pomocy języka VHLD sprawdza się szczególnie wtedy, gdy mamy do czynienia z „długimi" plikami pobudzeń.

Model symulacji VHDL

Standard VHDL opisuje sposób symulacji opisu. Standard nic nie wspomina o syntezie i działaniu projektu w układzie po przeprowadzeniu procesu syntezy. Standard określa się terminami:



Opracowanie hierarchii projektu (Elaboration of design hierarchy) jest etapem wstępnym. Kolejnym krokiem jest etap inicjalizacji (Initialization phase). Na końcu mamy cykle symulacji (Simulation cycles) które się zapętlają (symulator działa w nieskończoność).

OPRACOWANIE HIERARCHII polega na kompilacji opisu modelu i stworzeniu kodu, który będzie nadawał się do symulacji. Każdy kompilator przekłada kod VHDL na własny, wewnętrzny model wykonywalny. Wykonywanie kodu oznacza symulacji. Generowanie kodu do symulacji wiąże się z ustaleniem sterowników wszystkich sygnałów. Efektem symulacji jest określenie w jaki sposób będą zmieniały się wartości sygnałów i portów. Sterowniki są tworzone przez instrukcje przypisania sygnału. Gdy jeden sygnał ma wiele sterowników, używa się funkcji rozstrzygającej. Każdy sterownik jest tak naprawdę pamiętany w postaci listy transakcji. Transakcja to para „wartość/czas" (co oznacza: przypisz „wartość" po takim „czasie"). Jest to przewidywana fala prostokątna skojarzona z danym sterownikiem.

Musimy wprowadzić pojęcia ogólne:

Sygnał aktywny to sygnał którego sterownik ma transakcję w danym cyklu symulacji (w danym cyklu symulacji zostało wykonane przypisanie z danego sterownika do sygnału). To czy sygnał jest aktywny, czy nie można rozpoznać dzięki atrybutowi `ACTIVE`. Ma on wartość „prawda" gdy w danym cyklu symulacji sygnał jest aktywny (miał transakcję)

Zdarzenie to zmiana wartości sygnału w danym cyklu. Słowo „zmiana" jest ważne, ponieważ nie mówimy o zmianie wtedy, gdy do sygnału została przypisana jego poprzednia wartość. Zdarzenia wykrywa się przy użyciu atrybutu `EVENT` (zwraca „prawdę" gdy w danym cyklu symulacji sygnał zmienił swoją wartość). Atrybut ten spotkaliśmy na poprzednich zajęciach.

Będziemy posługiwać się oznaczeniami: $\mathbf{T_C}$ oznacza bieżący czas cyklu symulacji. Każdy symulator startuje od wartości 0, a potem ta wartość rośnie. Oznaczeniem $\mathbf{T_N}$ będziemy opisywać czas cyklu następnego.

INICJALIZACJA to przypisanie wartości początkowych wszystkim sygnałom. W omawianym wcześniej modelu mieliśmy jawne przypisane wartości początkowych. Wartości tych sygnałów będą propagować przez porty

wejściowe do wstawionej tam jednostki. Dalej propagacja obejmie sygnały wewnętrzne. W języku VHDL nie ma sygnałów niezainicjowanych. Sygnały mają albo wartości podane w sposób jawny (operator przypisania) albo są to domyślnie przypisane wartości „skrajnie lewe w danym typie sygnału". W przypadku wartości liczbowych będzie to najmniejsza wartość z zakresu. W przypadku typów wyliczeniowych bierze się pierwszą pozycję. W typie STD_ULOGIC pierwszą wartością jest 'U'. Należy pamiętać o tym, że sygnały, które nie zostaną przez nas jawnie zainicjowane będą miały przypisaną domyślnie wartość 'U'.

Faza inicjalizacji ma jeszcze jeden ważny element. Pozwala na przepropagowanie się wartości początkowych przez procesy. Wykonywane są jednokrotnie wszystkie procesy do momentu wstrzymania (gdy trafi na instrukcję „wait" lub na koniec procesu). Przed wejściem w pętlę symulacyjną pod wartość T_N podstawia się najbliższy moment transakcji (przeglądane są wszystkie listy transakcji i wybierany jest najbliższy moment wystąpienia transakcji) lub najbliższy moment wznowienia procesu (`wait for cośtam`).

CYKL SYMULACJI składa się z następujących operacji:

- Podczas bieżącej symulacji jest podstawiany T_N
- Aktualizacja sygnałów aktywnych w danym cyklu (konsekwencją będzie pojawienie się zdarzeń)
- Wykonanie procesów wyzwolonych zdarzeniami (procesy z listami wrażliwości) lub wstrzymanych do T_C instrukcją `wait`
- Podstawienie najbliższego momentu transakcji/wznowienia procesu pod T_N

Mówi się, że gdy w ostatnim kroku wyznaczony $T_N = T_C$ (następne zdarzenie jest dla tej samej chwili czasowej jak bieżący cykl symulacji), to następny cykl będzie tzw. „cyklem Δ". Cykle Δ, to takie cykle, w których czas nie posuwa się do przodu. Efekt jest taki, że symulator pętli się w jednej i tej samej chwili czasowej. Cykle Δ opisują propagację zmian w układzie cyfrowym. Bierze się to z rozdzielenia aktualizacji wartości sygnałów i wykonywania procesów. Nie ma współbieżności w wywoływaniu instrukcji przypisania i aktualizowaniu wartości sygnałów. W standardzie powiedziane jest że instrukcja przypisania sygnału nie modyfikuje jego wartości, tylko listę transakcji tego sygnału. Zapis `Cośtam <= '0'` oznacza nieformalnie `Cośtam <= '0' after Δ`. Należy więc pamiętać o tym, że przypisanie wartości jakiegoś sygnału będzie widoczne dopiero w następnym cyklu symulacji. Zmienne nie mają takich własności, ponieważ są inaczej traktowane. Poza tym na zmiennych nie da się opisać pracy układu cyfrowego. Podsumowując:

Każdy cykl symulacji składa się z dwutaktu. Najpierw aktualizuje się sygnały a potem wykonuje procesy. Oba kroki są rozdzielone. Jest to zgodne z rzeczywistym działaniem układów cyfrowych. Sygnały potrzebują czasu na propagację.

Zajmiemy się teraz przykładem. Przedstawione są instrukcje przypisania współbieżnego (na poziomie architektury):

Cykle symulacji - przykład

```
signal S1, S2 : STD_LOGIC;
(...)
S1 <= A and B;   -- A, B = porty in
S2 <= B xor S1;
```

Zakładamy że wartości początkowe sygnałów były następujące:

```
Czas      A    B   S1   S2
(...)    '1'  '1'  '1'  '0'
```

W chwili 10 ns port wejściowy A zmienił swoją wartość na '0':

```
Czas      A    B   S1   S2
(...)    '1'  '1'  '1'  '0'
10 ns    '0'  '1'  '1'  '0'   A'Event + transakcja dla S1
```

Co będzie działo się z instrukcjami przypisania współbieżnego? Będziemy na nie patrzeć jak na procesy. Zmiana na porcie A oznacza aktualizację wartości A w pierwszym etapie cyklu (dla 10 ns) i zostało wygenerowane zdarzenie (sygnał A będzie miał zdarzenie). Dalej zostanie wyzwolony proces przypisujący do sygnału S1. Zostanie obliczona wartość ('0') dla S1 i zostanie ona umieszczona na liście transakcji (również z czasem 10 ns). Właściwe przypisanie nastąpi dopiero w następnym cyklu (będzie to cykl Δ):

```
   Czas      A    B   S1   S2
   (...)    '1'  '1'  '1'  '0'
   10 ns    '0'  '1'  '1'  '0'   A'Event + transakcja dla S1
10 ns + Δ   '0'  '1'  '0'  '0'  S1'Event + transakcja dla S2
10 ns + 2Δ  '0'  '1'  '0'  '1'  S2'Event
```

W momencie „10 ns + Δ” zostanie zdjęta transakcja z listy dla S1 i sygnał ten zostanie zaktualizowany. Zostanie dla niego wygenerowane zdarzenie, które wyzwoli przypisanie dla S2. Proces ten wygeneruje transakcję dla sygnału S2. Ponieważ w procedurze nie ma żadnego opóźnienia, to transakcja dostanie czas 10 ns. Symulator będzie cały czas trwał w chwili 10 ns. Kolejny cykl będzie również cyklem Δ. Wówczas zostanie zdjęta z listy i wykonana transakcja dla sygnały S2. Jeżeli zmiana S2 wywoływałaby kolejne instrukcje przypisania, to mielibyśmy kolejne instrukcje przypisania i kolejne cykle Δ. W przykładzie mieliśmy 3 cykle potrzebne do opisania chwili 10 ns.

Należy pamiętać o tym że w VHDL-u nie ma natychmiastowych przypisań sygnałów. Istnieje opóźnienie Δ. Przypisanie (fizyczna zmiana sygnały po lewej stronie instrukcji przypisania) nastąpi dopiero w następnym cyklu.

Co się stanie jeżeli instrukcje współbieżne zamienimy na instrukcje sekwencyjne, umieszczone w procesie, który wyzwalany jest zmianami sygnałów A, B oraz S1?

```
signal S1, S2 : STD_LOGIC;
(...)
process (A, B, S1)
begin
  S1 <= A and B;
  S2 <= B xor S1;
end process;
```

Czy coś się zmieni? W kwestii cyklów symulacji nic się nie zmieni. Analiza powyższego zapisu przebiegałaby identycznie jak poprzedniego. Gdyby wewnątrz procesu umieścić instrukcje testujące wartość sygnału S1, to wyniki testów będą zwracały poprzednią wartość tego sygnału. Nie powinno się mieszać wewnątrz jednego procesu przypisań i odczytów jednego i tego samego sygnału. Najlepiej będzie rozdzielać procesy na osobne kawałki, które pozwolą nad wszystkim zapanować. Najlepiej poświęcać osobne procesy do przypisywania wartości osobnych sygnałów. Podobnie z odczytywaniem.

[Tymczasowy powrót do tematu układów synchronicznych]

Na poprzednich zajęciach omówione zostały sposoby opisu układów synchronicznych. Pozostał jeszcze do opisania układ rejestru przesuwnego SIPO (Serial In Parallel Out):

```
entity SReg8b is
    port ( Din : in  STD_LOGIC;
           Clk : in  STD_LOGIC;
           Q   : out STD_LOGIC_VECTOR( 7 downto 0 ) );
end SReg2b;
architecture RTL of SReg8b is
  signal iQ : STD_LOGIC_VECTOR( 7 downto 0 );
begin
  Q <= iQ;
  process ( Clk )
  begin
    if rising_edge( Clk ) then
      iQ( 7 downto 0 ) <= iQ( 6 downto 0 ) & Din;
    end if;
  end process;
end architecture;
```

Konieczne jest pracowanie na kopii wewnętrznej portu wyjściowego (odczyt portu wyjściowego jest postępowaniem błędnym). Zasada działania rejestru przesuwnego oparta jest na instrukcji przypisania:

```
iQ (7 downto 0) <= iQ (6 downto 0) & Din;
```

Z taką konstrukcją spotkaliśmy się przy omawianiu operatora konkatenacji. Instrukcja ta musi być wykonywana synchronicznie. Objęta jest warunkiem testującym zbocze narastające zegara. Dalej bierzemy 6 młodszych bitów wektora „Q”. Z prawej strony doklejamy wartość „Din”. Tak powstanie przesunięcie w lewo. Przesuwanie w prawo wymagałoby modyfikacji kodu:

```
iQ (7 downto 0) <= Din & iQ (7 downto 0);
```