Kurs Verilog cz.1 – wstęp

Złożoność układów cyfrowych zgodnie z prawem Moore'a, podwaja się co około 18 miesięcy. Liczba tranzystorów znajdujących się w układzie scalonym sięga już dziesiątek, a nawet setek milionów. Choć są to liczby ogromne, to w dzisiejszych czasach nie robią już wielkiego wrażenia.

Przy takiej złożoności układów scalonych stosowane dotychczas narzędzia projektowe przestają być użyteczne. Operują one bowiem na zbyt niskim poziomie abstrakcji (bramki logiczne), co uniemożliwia projektowanie złożonych układów w rozsądnym czasie.

Następstwem dynamicznego rozwoju techniki cyfrowej i rosnących trudności w projektowaniu coraz bardziej złożonych układów, jest powstanie języków opisu sprzętu (**HDL** – Hardware Description Language). Jest to zupełnie nowe podejście do projektowania – języki opisu sprzętu pozwalają na opis zachowania, a nie struktury układu (choć nie wykluczają także opisu struktury, jeśli zachodzi taka potrzeba).

1. HDL

Omawiając język Verilog zaczniemy od tego, czym ogólnie są języki opisu sprzętu (HDL).

Język opisu sprzętu pozwala nam stworzyć abstrakcyjny model rzeczywistego urządzenia elektronicznego. Bardzo ważne jest właśnie pojęcie **abstrakcji** w opisywaniu działania urządzenia, a dokładniej poziom abstrakcji. Możemy wyróżnić trzy takie poziomy:

- poziom połączenia,
- poziom bramek,
- poziom transferu rejestrowego (RTL – register transfer level).

Poziom połączenia - stanowi on opis rozkładu ścieżek, tranzystorów, rezystorów i innych elementów w układzie. Jest to więc poziom najniższy, najbardziej szczegółowy, ale wymagający ogromnego nakładu pracy, aby za jego pomocą stworzyć skomplikowany układ.

Poziom bramek – jest już bardziej przyjazny dla projektanta. Na poziomie tym możemy opisywać bramki logiczne oraz przerzutniki, czyli znając działanie podstawowych bramek (AND, OR, NOT, XOR, itp.) możemy już stworzyć w rozsądnym czasie nieskomplikowany, lecz przydatny układ.

Profesor, który wykładał Układy Cyfrowe na moich studiach, na jednym ze spotkań opowiedział nam jak kiedyś projektowało się układy scalone. Gdy żyły jeszcze dinozaury, schematy rysowane na poziomie bramek logicznych, z powodu ich ogromu wieszane były na ścianie. Następnie każdy mógł popatrzeć i poszukać błędów (pewnie organizowali konkursy). Podobno starzy wyjadacze orientowali się nawet, co się w takim układzie dzieje.

Poziom transferu rejestrowego – nazwa trochę dziwna, ale jest to poziom, który nas najbardziej interesuje. Jest to opis rejestrów oraz transferu sygnałów między nimi.

Podsumowując – **HDL** pozwala nam zaprojektować układ opisując jego działanie. Mało tego – pozwala nam przetestować i zasymulować działanie tworzonego układu zanim zostanie on fizycznie zaimplementowany w strukturze krzemu (układzie CPLD, FPGA, ASIC).

2. Verilog

Wiemy już czym jest HDL, a czym jest Verilog? **Verilog** jest jednym dwóch popularnych języków opisu sprzętu. Jego powstanie datuje się na rok 1984. Patrząc na tempo rozwoju technologi cyfrowej jest to już staruszek! Od tamtego czasu uległ on pewnym modyfikacjom, powstały także nowe wersje. Mimo wszystko w świecie profesjonalnych producentów układów scalonych stosuje się rozwiązania starsze, ale dobrze sprawdzone. Pewność działania przekłada się w tym przypadku na miliony dolarów.

Dlaczego Verilog, a nie VHDL?

No właśnie – dlaczego? Nie dlatego, że nie lubię VHDL. Nawet za dobrze go nie znam. Verilog okazał się po prostu na tyle prosty do nauczenia i stosowania w praktyce, że wygrał z VHDL. VHDL jest mniej czytelny i posiada pewne wady, które utrudniają wykrycie pomyłek.

Verilog w dużym stopniu przypomina język C, a ten zna prawie każdy elektronik. Pewnie Ty też. Dlatego poznanie podstaw języka Verilog zajmie Ci… godzinę?

Zachęcam też do porównania go VHDL – różnice w wygodzie stosowania są łatwo zauważalne.

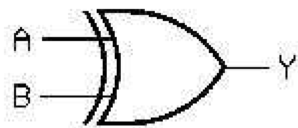
Zmiana sposobu myślenia

Opis składni języka Verilog zamieszczę w kolejnej części tutoriala. Na koniec jedna mała rzecz – zmiana sposobu myślenia o programowaniu.

Najprawdopodobniej masz na swym sumieniu co najmniej jeden programik napisany w C, Delphi, Basicu, Javie, czy jeszcze jakimś innym języku. Wiesz zatem, że instrukcje, z których składa się Twój program wykonywane są jedna po drugiej, **sekwencyjnie**, szeregowo. I to jest właśnie najważniejsza różnica dotycząca tworzenia programów i tworzenia opisu sprzętu. W sprzęcie tzw. procesy wykonują się **jednocześnie**, **równolegle**. Tzn., że jeśli opiszesz jakiś proces, a inny w kolejnych liniach kodu, lub zrobisz to odwrotnie – to efekt będzie identyczny – oba procesy wykonają się w tym samym czasie.

Nie wiem, czy to czujesz… Dla mnie jest to już intuicyjne i zapewniam Cię, że po kilku przykładach na pewno też to ogarniesz.

Kurs Verilog cz.2 – moduł



Najważniejszym elementem każdego projektu w języku **Verilog** jest **moduł**. Wszystkie deklaracje, instrukcje, procesy itp. znajdują się wewnątrz modułu. Stanowi on abstrakcyjny model układu, lub danej części układu. Można go sobie wyobrazić, jako czarną skrzynkę z interfejsem wejściowym, wyjściowym oraz pewną funkcjonalnością.

Moduł może opisywać układ o dowolnej złożoności. Mogą to być podstawowe bramki logiczne, liczniki, sterowniki, a także bardziej skomplikowane układy jak mikroprocesory, sprzętowe kodeki wizyjne itp.

Najczęściej złożone układy dzieli się na mniejsze części, z których każdą można opisać w osobnym module. Dostarcza się w ten sposób prostych bloków funkcjonalnych, które później mogą być wykorzystane w wielu miejscach projektu.

1. Budowa modułu

Deklaracja modułu rozpoczyna się słowem kluczowym `module` po której podawana jest jego nazwa. Koniec modułu wyznacza słowo kluczowe `endmodule`.

Wygląda to mniej więcej tak:

```
module <nazwa_modułu> (<opcjonalna_lista_portów>);
...
endmodule
```

Nazwa modułu

Nazwa modułu nie może być słowem kluczowym języka Verilog oraz nie może zaczynać się od cyfry. Prawidłowa nazwa może się składać z liter, cyfr oraz znaków _ oraz `$`. Ważna jest także wielkości liter w nazwie, dlatego przykładowa nazwa `Adder` nie jest równoważna nazwie `adder`.

Porty

Po nazwie modułu w nawiasie podajemy nazwy portów w jakie ma być wyposażony moduł. Rolę portu tzn. czy ma on być wejściowy, wyjściowy, czy dwukierunkowy określa się w ciele modułu (można też inaczej, ale o tym później). Kolejność deklarowania portów jest dowolna.

```
<funkcja_portu> <nazwa_portu>;
```

Port może pełnić jedną z trzech funkcji:

- **input** – wejście,
- **output** – wyjście,
- **inout** – port dwukierunkowy

Przykład:

```
module and3 (x1, x2, x3, y);

input x1, x2, x3;
output y;

assign y = x1 & x2 & x3;

endmodule
```

Powyższy przykład przedstawia realizację 3-wejściowej funkcji **AND**. Słowo kluczowe `assign` oznacza „przypisz”, czyli przypisujemy sygnałowi y kombinację sygnałów `x1`, `x2` oraz `x3`.

Komentarze

W języku Verilog dozwolone są komentarze znane z języka C, czyli: blokowy znajdujący się pomiędzy znacznikami `/*` oraz `*/`, a także jedno liniowy zaczynający się od `//`.

2. Instancja

Czym jest instancja? Instancja jest, mówiąc kolokwialnie, powołanym do życia modułem. Podobnie jak w języku obiektowym np. C++, mamy osobne pojęcia stanowiące klasę i obiekt. Klasa stanowi pewien opis, a obiekt jest już istniejącym w programie tworem, stworzonym na podstawie tej klasy.

W przypadku języka Verilog moduł stanowi opis, a instancja jego wywołanie. Zobaczmy na przykładzie. Mamy dwa moduły (moduły jest najlepiej definiować w osobnych plikach, ale nie jest to obowiązek), gdzie drugi z nich wywołuje dwie instancje (`a1` oraz `a2`) pierwszego.

```
module add3 (x1, x2, x3, y);

input x1, x2, x3;
output y;

assign y = x1 & x2 & x3;

endmodule
module box(x1, x2, x3, x4, x5, x6, y);

input x1, x2, x3, x4, x5, x6;
output y;

wire y1, y2;
```

```
add3 a1(x1, x2, x3, y1);
add3 a2(x4, x5, x6, y2);

assign y = y1 | y2;

endmodule
```

Każda z instancji musi posiadać własną unikalną nazwę. Wywołując instancję nie musimy połączać wszystkich wejść oraz wyjść. Jeśli któreś z nich opuścimy, syntezer odpowiednio zoptymalizuje układ wycinając niepotrzebną część (jeśli niczego nie podłączymy wycięty zostanie cała instancja).

Ale to jest już wiedza dla średniozaawansowanych.

PS.

Ten inny, według mnie wygodniejszy sposób definiowania portów wygląda następująco:

```
module and3 (
input x1, x2, x3,
output y
);

assign y = x1 & x2 & x3;

endmodule
```

Dzięki takiemu zapisowi wystarczy opisać porty w jednym miejscu, co ułatwia wszelkie zmiany. Zauważ, że poszczególne nazwy portów oddzielone są przecinkiem, natomiast po ostatniej przecinka już nie ma.

Kurs Verilog cz.3 – symulacja

Znamy już najważniejszy element każdego projektu w Verilogu. Gdy moduł jest zaprojektowany możemy (a nawet musimy, chyba że jest on banalnie prosty) sprawdzić jego działanie. Najlepszym sposobem na zweryfikowanie poprawności projektu jest przygotowanie i przeprowadzenie odpowiedniej symulacji.

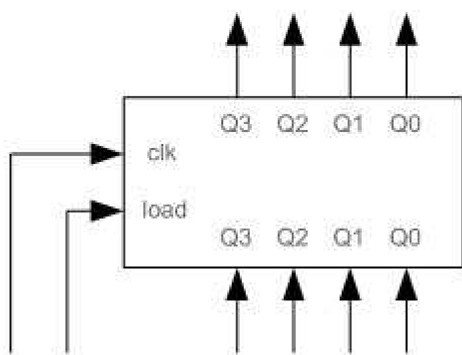
Dla zawziętych elektroników, praktyków symulacja nie kojarzy się najlepiej. W przypadku układów programowalnych jest to jednak nieodzowny element każdego projektu, ale nie ma się czego bać – wygląda to o niebo lepiej niż w przypadku układów analogowych.

O tym, jakie narzędzie do przygotowania symulacji udostępnia opis sprzętu przekonasz się czytając poniższy tekst.

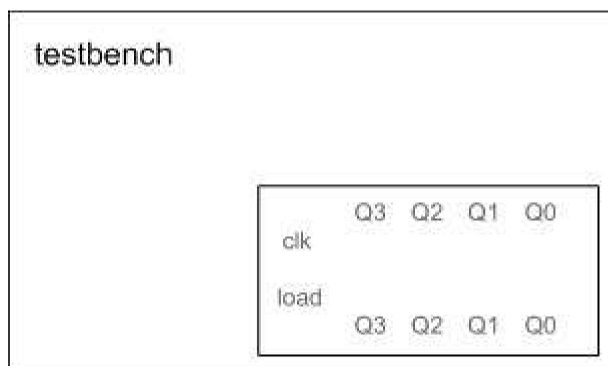
1. Testbench

Aby przeprowadzić symulację niezbędne jest stworzenie dodatkowego bloku, który określi wymuszenia dla testowanego modułu. Blok taki nosi nazwę **testbench** i definiuję się go tak, jak każdy inny moduł w Verilogu. Testbench stanowi niejako opis środowiska zewnętrznego układu.

Najlepiej zrozumieć to analizując przykład. Mamy 4-bitowy licznik z możliwością wpisania wartości początkowej, taki jak na rysunku poniżej.



Rysunek 1. Licznik 4-bitowy



Rysunek 2. Testbench licznika

Finalnie do naszego układu podłączymy inne układy (źródło impulsów zliczanych, sterowanie, odczyt aktualnej wartości). Symulacją takiego właśnie środowiska zajmuje się testbench.

Aby zilustrować powyższe informacje napiszemy moduł licznika, oraz testbench dla niego. Zacznijmy od licznika, ta część jest dość prosta:

```
module licznik(
input wire clk,
input wire load,
input wire [3:0] i_Q,
output reg [3:0] o_Q
);

always @(posedge clk or posedge load)
   if(load)
      o_Q <= i_Q;
   else if(clk)
      o_Q <= o_Q + 1;

endmodule
```

Licznik będzie zliczał impulsy na narastającym zboczu sygnału `clk`, ładowanie wartości początkowej nastąpi na narastającym zboczu sygnału `load`.

Testbench będzie symulował źródło zliczanych impulsów, oraz zajmie się wpisaniem początkowej wartości równej 6.

```
module licznik_tb();

reg sys_clk;
reg load;
reg [3:0] Q;
wire [3:0] cnt;

licznik UUT(
.clk(sys_clk),
.load(load),
.i_Q(Q),
.o_Q(cnt)
);

initial
   begin
      sys_clk = 0;
      Q = 8'd6;
      load = 0;
      load = #100 1;
      load = #100 0;
   end

initial
   begin
   // wyświetlenie wyników symulacji
      $monitor ($time, " | %d", cnt);
   end

//-----------------------------------

always #25 sys_clk <= ~sys_clk;

//-----------------------------------
 endmodule
```

Szczegółowo testbench omówimy w kolejnych częściach kursu, tutaj przedstawiona jest pewna idea.

Wyniki można obserwować jako przebiegi czasowe oraz w konsoli symulacji (dzięki zastosowaniu funkcji $monitor).

2. Kilka uwag

- warto stosować nazwy modułów testbenchy z przyrostkiem _tb, jest to dobry zwyczaj ułatwiający orientację w plikach projektu,
- testowanie układu przed jego implementacją jest jednym z najważniejszych i nabardziej czasochłonnych etapów projektowania,
- testowanie odbywa się na dwóch poziomach: behawioralnym oraz tzw. post-route czyli uwzględniającą rozmieszczenie elementów (oraz opóźnienia z tego wynikające) wewnątrz układu programowalnego.

W następnej części przyjrzymy się bliżej składni języka Verilog.

Kurs Verilog cz.4 – start w jeden dzień

W tej części kursu znacznie przyspieszymy. Chciałbym pokazać, że możliwe jest opanowanie podstaw języka Verilog w jeden dzień (właściwie kilka godzin). Będzie to więc omówienie najważniejszych pojęć, zilustrowane prostymi przykładami. Wiedza, którą posiądziesz czytając tą część, pozwoli Ci podjąć samodzielne próby projektowania urządzeń cyfrowych w języku Verilog.

Zakładam, że miałeś już do czynienia z programowaniem w jakimkolwiek języku i znasz podstawy działania układów cyfrowych.

W kolejnych wpisach, nie będących częścią kursu, opiszę jak zamienić opis w plik, który możemy wpisać do układu programowalnego, oraz przedstawię prosty zestaw startowy, na którym uruchomisz przykłady i własne projekty.

1. Cykl projektowania

- specyfikacja
- schemat blokowy
- projekt niskiego poziomu
- kodowanie RTL
- weryfikacja
- synteza

Zastosujmy się do cyklu projektowego – zacznijmy więc od specyfikacji projektu.

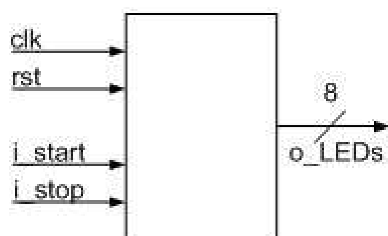
2. Specyfikacja

Co zamierzamy zaprojektować? Na początek spróbujemy ze stosunkowo prostym układem. Dodatkowo chciałbym, aby możliwe było obejrzenie wyników naszej pracy. Dlatego spróbujemy z prostym efektem przesuwającego się punktu na diodach LED.

Zakładamy, że mamy do dyspozycji 8 LEDów i chcemy, aby punkt przemieszczał się co 0,2s. Użyjemy także dwóch przycisków: jeden będzie włączał, a drugi wyłączał generowanie efektu.

Przejdźmy więc do kolejnego etapu projektowania – schematu blokowego.

3. Schemat blokowy

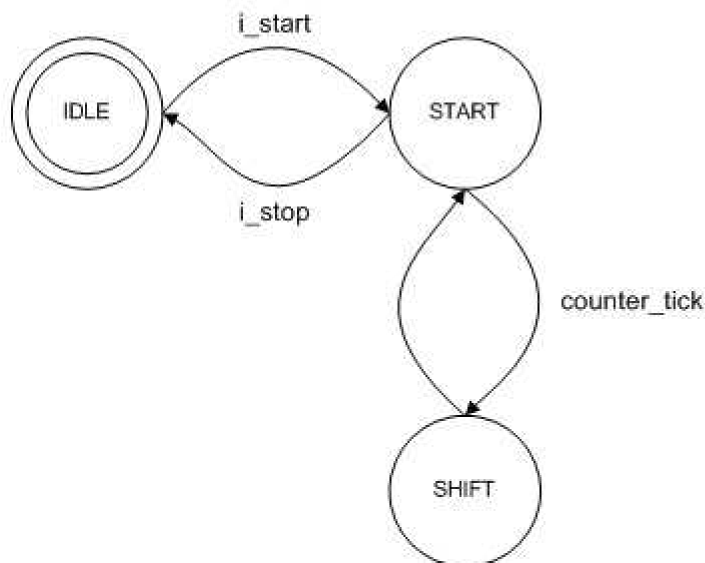


Rysunek 1. Schemat blokowy

W tym momencie jedyne co musimy zrobić, to określić porty (wejścia, wyjścia, porty dwukierunkowe), w które wyposażony będzie nasz układ. Schemat blokowy stanowi reprezentację przepływu sygnałów w naszym projekcie. Na tym etapie nie zastanawiamy się, co będzie wewnątrz naszej czarnej skrzynki.

4. Projekt niskiego poziomu

Teraz spróbujemy narysować automat realizujący założone funkcje.



Rysunek 2. Automat

Od razu powiem – można to zrobić inaczej, ale my pozostaniemy przy takiej konstrukcji.

W stanie `IDLE` automatu wpisywany jest początkowy stan LEDów, w naszym przypadku wartość `8'hFE`, ponieważ chcemy, aby poruszał się jeden punkt (przy aktywnym stanie niskim). Licznik mierzący czas 0,2s nie będzie startowany ani zatrzymywany – będzie on działał cały czas. Stan `START` osiągany jest po wystąpieniu logicznej jedynki na wejściu `i_start`. Gdy automat jest w stanie `START`, może przejść do stanu `SHIFT` (pod wpływem osiągnięcia odpowiedniej wartości przez licznik). W stanie `SHIFT` następuje przesunięcie zawartości rejestru sterującego diodami, a następie powrót do stanu `START`.

Po narysowaniu automatu tradycyjny proces projektowania zakłada: stworzenie tablicy prawdy z przejściami stanów dla każdego przerzutnika, mapy Karnaugh i kolejne nużące kroki.

Na szczęście w Verilogu po narysowaniu automatu (w małych projektach może wydawać się to zbędne, jednak warto przyzwyczajać się do tego, gdyż ich przydatność znacznie wzrasta wraz ze złożonością układu), ten etap się kończy.

Jeśli nie wiesz jak stworzyć i odczytać automat stanów, musisz na własną rękę uzupełnić tę wiedzę.

5. Kodowanie RTL

Tutaj rozpoczynamy już przygodę z Verilogiem.

Kodowanie rozpoczynamy od stworzenia modułu. O modułach w Verilogu pisałem w drugiej części kursu – jeśli jej nie czytałeś najwyższy czas, abyś to zrobił. W tym miejscu posłużymy się schematem blokowym z rysunku 1.

```
module leds_effect(
input clk,
input rst,
input i_start,
input i_stop,
output reg [7:0] o_LEDs
);

...

endmodule
```

W naszym projekcie występują tylko dwa rodzaje portów: wejściowe oraz wyjściowe. W drugiej części kursu przeczytałeś, że istnieją także porty dwukierunkowe `inout`. W przyszłości będziemy korzystać także z nich np. do komunikacji z układem FT245.

Wektor sygnałów

W naszym przykładzie napotykamy:

```
output reg [7:0] o_LEDs
```

Zapis `[7:0]` przed nazwą portu, oznacza 8-bitowy wektor, gdzie najstarszy bit ma numer 7 a najmłodszy 0. Możliwa jest także konstrukcja

```
output reg [0:7] o_LEDs
```

W takim przypadku najstarszy bit ma numer 0, a najmłodszy 7.

Jeśli chcemy się odwołać do jednego z sygnałów wektora, piszemy:

```
assign x = o_LEDs[0] | o_LEDs[3];
```

co w tym przypadku oznacza, przypisanie do `x` sumy logicznej najmłodszego oraz trzeciego bitu wektora `o_LEDs`.

Typy danych

W układach logicznych mamy do czynienia z dwoma rodzajami wymuszeń:

- wymuszenie, które przechowuje wartość,

- wymuszenie, które łączy ze sobą dwa lub więcej punktów.

Pierwszy typ reprezentowany jest w Verilogu przez rejestr (słowo kluczowe `reg` od ang. register), natomiast drugi to przewód (`wire`).

Istnieje spora liczba innych typów danych dostępnych w Verilogu, jednak ich znajomość wymagana jest dopiero przy zaawansowanych projektach.

Przykłady deklaracji:

```
reg y; // rejestr 1-bitowy
reg [15:0] counter; // rejestr 16-bitowy
wire a, b, c; // trzy sygnały typu wire
```

Operatory

Operatory w Verilogu podobne są do tych znanych z innych języków programowania i nie będziemy ich szczegółowo omawiać. Dostępne operatory zebrane zostały w poniższej tabeli.

<table>
<tr><th>Typ operatora</th><th>Symbol</th><th>Znaczenie</th></tr>
<tr><td rowspan="5">Arytmetyczny</td><td>*</td><td>Mnożenie</td></tr>
<tr><td>/</td><td>Dzielenie</td></tr>
<tr><td>+</td><td>Dodawanie</td></tr>
<tr><td>-</td><td>Odejmowanie</td></tr>
<tr><td>%</td><td>Modulo</td></tr>
<tr><td rowspan="3">Logiczny</td><td>!</td><td>Negacja logiczna</td></tr>
<tr><td>&&</td><td>Logiczne AND</td></tr>
<tr><td>||</td><td>Logiczne OR</td></tr>
<tr><td rowspan="4">Relacji</td><td>></td><td>Większe niż</td></tr>
<tr><td><</td><td>Mniejsze niż</td></tr>
<tr><td>>=</td><td>Większe niż lub równe</td></tr>
<tr><td><=</td><td>Mniejsze niż lub równe</td></tr>
<tr><td rowspan="2">Równości</td><td>==</td><td>Równe</td></tr>
<tr><td>!=</td><td>Różne</td></tr>
<tr><td rowspan="5">Bitowy</td><td>~</td><td>Negacja bitowa</td></tr>
<tr><td>&</td><td>Bitowy AND</td></tr>
<tr><td>|</td><td>Bitowy OR</td></tr>
<tr><td>^</td><td>Bitowy XOR</td></tr>
<tr><td>^~ lub ~^</td><td>Bitowy XNOR</td></tr>
<tr><td rowspan="6">Redukcji</td><td>~</td><td>Redukcja NOT</td></tr>
<tr><td>~&</td><td>Redukcja NAND</td></tr>
<tr><td>|</td><td>Redukcja OR</td></tr>
<tr><td>~|</td><td>Redukcja NOR</td></tr>
<tr><td>^</td><td>Redukcja XOR</td></tr>
<tr><td>^~ lub ~^</td><td>Redukcja XNOR</td></tr>
<tr><td>Przesunięcia</td><td>>></td><td>Przesunięcie w prawo</td></tr>
</table>

Typ operatora	Symbol	Znaczenie
	<<	Przesunięcie w lewo
Złożenia (konkatenacji)	{ }	Złożenie
Warunkowy	?	Warunek

Wyrażenia warunkowe

Popatrzmy na słowa kluczowe związane z wyrażeniami warunkowymi: `if`, `else`, `repeat`, `while`, `for`, `case`. Dla osób, które znają jakiś język programowania (nawet jeśli jest to np. PHP) słowa te na pewno zabrzmią znajomo. Zakładam, że i Ty jesteś taką osobą. Nie będziesz więc miał problemów ze zrozumieniem idei wyrażeń warunkowych.

Jedyne na co należy zwrócić uwagę, to fakt, że opis w języku Verilog zostanie przetłumaczony na układ logiczny i nie każda konstrukcja może się nadawać do zaimplementowania w sprzęcie.

if, else

Wyrażenie `if`, `else` pozwala na warunkowe wykonanie kodu. Jeśli warunek jest spełniony wykonywana jest część za `if`, w przeciwnym wypadku część za `else`, przy czym część `else` nie musi się pojawić.

Przykład:

```
always @(posedge clk or posedge rst)
   if(rst)
      counter <= 0;
   else if(a[7])
      counter <= counter + 1;
```

Konstrukcji `if`, `else` nie stosuje się poza blokiem `always`. Jeśli instrukcji, które mają się wykonać jest więcej, należy objąć je blokiem `begin`, `end` (analogia do { } z języka C).

case

Wyrażenie `case` stosowane jest, gdy jedna zmienna musi zostać sprawdzona pod kątem wielu wartości. Wyrażenie `case` zastępuje wielokrotne wykorzystanie konstrukcji `if`, `else`.

Blok case zaczyna się słowem kluczowym `case` a kończy słowem `endcase`.

Dobrym zwyczajem jest stosowanie opcji `default`. Dlaczego? Rozważmy przypadek automatu, który z jakiegoś powodu (np. silne zakłócenia elektromagnetyczne) wchodzi w nieistniejący stan. Bez opcji `default` automat taki pozostanie w nieistniejącym stanie zawieszając działanie urządzenia (lub jego części). Opcja `default` stanowi zbiór wszystkich możliwych, a nieopisanych stanów i pozwala wybrnąć z opisanej sytuacji.

Przykład:

```
always @(posedge clk or posedge rst)
   if(rst)
      rx_state <= 0;
   else
      case(rx_state)
      0: rx_state <= i_stb ? 1 : 0;
      1: rx_state <= i_stb ? 1 : 2;
      2: rx_state <= 0;
      default: rx_state <= 0;
      endcase
```

Konstrukcji `case`, `endcase` nie stosuje się poza blokiem `always`.

Układy kombinacyjne oraz synchroniczne

W układach cyfrowych występują dwa typy bloków, są to: bloki kombinacyjne oraz bloki sekwencyjne.

Dla nas, istotne jest, w jaki sposób modelować te elementy w Verilogu:

- elementy kombinacyjne mogą być modelowane za pomocą przypisania (`assign`) oraz procesów (blok `always`),
- elementy sekwencyjne mogą być modelowane tylko jako procesy (bloki `always`)

Blok always

Słowo `always` (z ang. zawsze) sugeruje nam sposób działania tego bloku. Mianowicie, jest on wykonywany zawsze, gdy spełniony jest warunek z listy czułości. **Lista czułości** informuje blok `always`, kiedy wykonać kod zawarty w jego wnętrzu. Symbol `@` (at) występujący po słowie kluczowym `always` tworzy z nim wyrażenie **always at**, czyli „zawsze gdy".

Blok `always` nie może sterować daną typu `wire`. Dopuszczalne są tylko dane typu `reg` oraz `integer`. Kolejną ważną zasadą konstrukcji bloków `always`, jest sterowanie konkretną daną tylko z jednego bloku. Jeśli przypisania znajdą się w dwóch lub więcej blokach, synteza zakończy się niepowodzeniem.

Lista czułości może być:

- listą czułą na poziom (dla układów kombinacyjnych),
- listą czułą na zbocze (dla układów sekwencyjnych).

Przykład:

```
always @(a or b or sel)
   if(sel == 0)
      y <= a;
   else
      y <= b;
```

Przykład:

```
always @(posedge clk or posedge rst)
   if(rst)
```

```
         y <= 0;
      else if(sel == 0)
         y <= a;
      else
         y <= b;
```

Możliwe jest także reagowanie na zbocze opadające stosując słowo kluczowe `negedge`. Nie można natomiast umieszczać na jednej liście, wyrażeń reagujących na poziom i wyrażeń reagujących na zbocze.
Przypisania wewnątrz bloków `always` mogą być dwojakiego typu. Pierwszy typ, to przypisania nieblokujące wykorzystujące zapis `<=` np.:

```
   always @(posedge clk or posedge rst)
      if(rst)
      begin
         x <= 0;
         y <= 0;
         z <= 0;
      end
      else if(flag)
      begin
         x <= 1;
         y <= x;
         z <= y;
      end
```

Drugi typ, to przypisania blokujące:

```
   always @(posedge clk or posedge rst)
      if(rst)
      begin
         x = 0;
         y = 0;
         z = 0;
      end
      else if(flag)
      begin
         x = 1;
         y = x;
         z = y;
      end
```

W pierwszym przypadku, gdy kod zależny od zmiennej `flag`, wykona się po raz pierwszym, zawartość zmiennych będzie następująca `x = 1`, `y = 0`, `z = 0`, ponieważ w `x` oraz `y`, w poprzednim cyklu zegarowym były zera i ta wartośc zostanie wpisana do rejestrów `y` oraz `z`.
Natomiast w drugim przypadku, będzie to `x = 1`, `y = 1`, `z = 1`, ponieważ instrukcję wykonują się sekwencyjnie (kolejne przepisania są opóźniane).

Stosowanie przypisań blokujących jest niezalecane i ich stosowanie powinno być ograniczone do niezbędnego minimum. Zabronione jest mieszanie typów przypisań w jednym bloku `always`.

Możliwa jest także konstrukcja `always` bez listy czułości, stosowana w symulacji.

```
   always #5 sys_clk = ~sys_clk;
```

Zapis `#5` oznacza opóźnienie (w jednostkach określonych w projekcie) i nie jest syntezowalny, tzn. nie doprowadzi do wygenerowania sygnału zegarowego w zaimplementowanym urządzeniu. Sygnał taki wystąpi tylko w symulacji.

Przypisanie (assign)

Przypisanie służy do modelowania układów kombinacyjnych w Verilogu.

Przykład:

```
assign y = a & b;
assign not_q = ~q;
```

Blok inicjujący (initial)

Blok `initial` jest niesyntezowalny, tzn. wykorzystywany jest tylko do symulacji. Jest on wykonywany przy starcie symulacji.

Przykład:

```
initial
begin
   sys_clk = 0;
   sys_rst = 0;
   data = 8'h41;
end
```

Uwaga, ważne! Blok `initial` nie powoduje wpisania wartości do rejestru lub pamięci w fizycznie zaimplementowanym układzie. Jest to konstrukcja, która służy jedynie do symulacji programowej projektu.

6. Weryfikacja projektu

Dobrze, napiszmy opis naszego efektu świetlnego i spróbujmy zasymulować jego działanie. Przesunięcie punktu na LEDach ma następować co 0,2s, załóżmy więc, że nasz układ działa z oscylatorem 10MHz – jest to nam potrzebne do obliczenia wartości wpisywanej do licznika. Jeśli częstotliwość będzie inna, należy odpowiednio zmodyfikować wspomnianą wartość.

```
module leds_effect(
input clk,
input rst,
input i_start,
input i_stop,
output reg [7:0] o_LEDs
);

integer main_state;
reg [18:0] cnt;

// licznik zliczający w dół
always @(posedge clk or posedge rst)
   if(rst)
      cnt <= 19'h30D3E;
   else if(cnt[18])
      cnt <= 19'h30D3E;
```

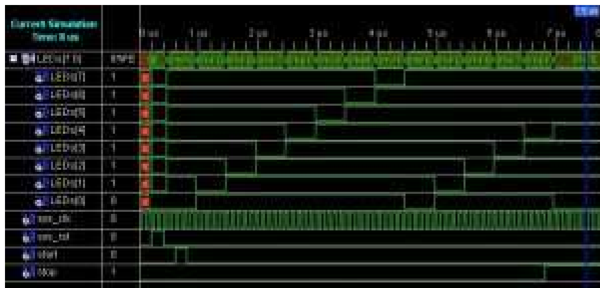
```
   else
      cnt <= cnt - 1;

// automat
always @(posedge clk or posedge rst)
   if(rst)
      main_state <= 0;
   else
      case(main_state)
      0: main_state <= i_start ? 1 : 0;
      1: if(i_stop) main_state <= 0; else if(cnt[18]) main_state <=
2;
      2: main_state <= 1;
      default: main_state <= 0;
      endcase

// LEDy
always @(posedge clk or posedge rst)
   if(rst)
      o_LEDs <= 0;
   else if(main_state == 0)
      o_LEDs <= 8'hFE;
   else if(main_state == 2)
   begin
      o_LEDs[7:1] <= o_LEDs[6:0];
      o_LEDs[0] <= o_LEDs[7];
   end

endmodule
```

Mamy już opis naszego efektu świetlnego, teraz spróbujmy zasymulować jego działanie. Napiszmy więc testbench:

```
module leds_effect_tb();

reg sys_clk, sys_rst;
reg start, stop;
wire [7:0] LEDs;

initial
begin
   sys_clk = 0;
   sys_rst = 0;
   start = 0;
   stop = 0;

   sys_rst = #200 1;
   sys_rst = #200 0;

   start = #200 1;
   start = #200 0;

   stop = #6000 1;
end

always #50 sys_clk = ~sys_clk; // opóźnienie w nanosekundach,
częstotliwość 10MHz

leds_effect UUT(
.clk(sys_clk),
```

```
.rst(sys_rst),
.i_start(start),
.i_stop(stop),
.o_LEDs(LEDs)
);
endmodule
```

Wynik symulacji (wartość wpisywana do licznika została zmieniona na 3, aby na przebiegach czasowych widoczne były zmiany):



Rysunek 3. Wynik symulacji

Jest kod, widzisz przebiegi czasowe, ale jak czy wiesz jak uruchomić symulację? O tym w kolejnym wpisie (poza kursem), a w następnych częściach przyjrzymy się bliżej elementom języka Verilog.