Język Verilog w przykładach (1)

Wprowadzenie, opis multipleksera i licznika

Na forach internetowych temat Verilog robi furorę. Obok VHDL'a jest alternatywnie używany prawie we wszystkich komercyjnych programach symulacji i syntezy logicznej układów cyfrowych, a także do opisu bloków funkcjonalnych tych układów zamieszczanych w podręcznikach. Postanowiliśmy przybliżyć go Czytelnikom. W kilku artykułach przedstawiamy język opisu sprzętu Verilog. Ponieważ założeniem tego cyklu jest nauka języka na podstawie przykładów, to opis jego instrukcji, ich składni i innych elementów będzie ograniczony do niezbędnego minimum.

Obecnie VHDL i Verilog są najczęściej stosowanymi językami opisu sprzętu (HDL). W Polsce dużą popularnością cieszy się VHDL, jednak Verilog ma coraz więcej zwolenników. Ich możliwości są porównywalne, przy czym Verilog jest znacznie mniej rygorystyczny, jeśli chodzi o kontrolę typów sygnałów. Języki te umożliwiają opis układów cyfrowych w celu ich syntezy logicznej lub symulacji za pomocą odpowiedniego oprogramowania projektowego, a także implementacji zsyntetyzowanego układu w programowalnych strukturach PLD (CPLD i FPGA).

Oprogramowanie projektowe

Active HDL firmy Aldec jest zintegrowanym środowiskiem do opisu (w językach VHDL i Verilog) i symulacji systemów cyfrowych do realizacji w PLD. Jego zaletą jest niezależność wyników syntezy i symulacji od architektury docelowego układu scalonego, a więc i jego producenta (Altera, Xilinx, Actel, Lattice itd.). Program jest wart zainteresowania, gdyż ma przyjazny w użyciu symulator i jest udostępniany za darmo na licencji studenckiej (przy ograniczonych, ale wystarczających dla początkujących, możliwościach funkcjonalnych).

Producenci układów programowalnych także oferują swoje systemy projektowe umożliwiające opisanie projektowanego układu w jednym z języków HDL, jego syntezę logiczną oraz symulację funkcjonalną bądź z uwzględnieniem opóźnień, a także realizację opisanego układu w strukturach PLD (zaprogramowanie PLD plikiem wynikowym odpowiadającym opisowi).

Na przykład firma Xilinx oferuje pakiet *ISE WebPACK*, firma Altera pakiet *Quartus II Web Edition*, natomiast dla układów FPGA firmy Lattice można używać *ISPLever starter*. Jest to oprogramowanie w wersjach darmowych, również z nieco ograniczonymi możliwościami, jednak wystarczające do nauki języka i przygotowywania projektów własnych układów do realizacji w strukturach programowalnych.

Języki opisu sprzętu

Języki opisu sprzętu są specyficzną grupą języków i nie można dla nich stosować wprost metod i rozumowania znanych z języków programowania (C/C++ i innych). Języki te są stosowane do opisania pewnego skonfigurowania elementów fizycznych w układ o określonych cechach funkcjonalnych (sformułowania modelu). Poszczególne wiersze opisu (instrukcje) są więc odpowiednio interpretowane jako połączenia konfigurujące układ, a nie wykonywane sekwencyjnie, jak w programach przygotowanych w językach programowania. Należy pamiętać, że nie każdy opisany model można zsyntezować, ale można w oparciu o ten opis przygotować moduł testujący (tzw. *testbench*), w celu jego symulacji.

Podstawy języka Verilog

Ponieważ założeniem kursu jest nauka języka na podstawie przykładów, opis jego składni i elementów będzie ograniczony do niezbędnego minimum. Podstawowym typem sygnału w Verilog'u jest *wire* będący odpowiednikiem przewodu w układzie. Takiemu sygnałowi może być na stałe przypisany określony stan logiczny. Może on być również deklarowany jako wektor (tablica) – co umożliwia reprezentowanie wielobitowych magistral sygnałowych. W t**ab. 1.** przedstawiono przykładowe sposoby użycia sygnału *wire.* Jak można zauważyć, operatorem występującym z sygnałami typu *wire* jest *assign* (wyłącznie z tym typem sygnałów).

Tab. 1. Przykłady użycia sygnałów *wire*

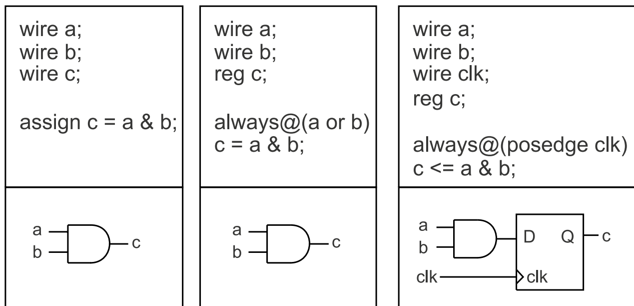
Kod:	Opis operacji:

`wire a;`
`wire b;`
`wire c;`
`assign a=b;`
	Tworzy 1-bitowe sygnały a, b i Cc, z przypisaniem na stałe sygnałowi a wartość sygnału b.

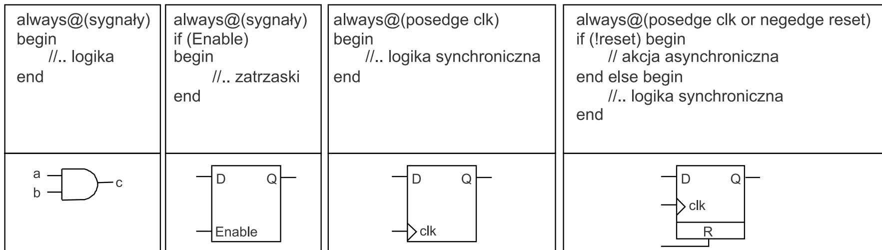
`wire [7:0] a = 8'hA;`	Tworzy 8-bitowy sygnał a z przypisaniem mu stałej wartości 0xA (hex). Zastosowany system liczbowy jest określony odpowiednią literą (d, h, b): dziesiętny: 8'd10, szesnastkowy: 8'hA, binarny: 8'b1010).

`wire [7:0] a;`
`wire [7:0] b;`
`assign a = {4'd0, b[3:0] };`
	Przykład konkatenacji (zestawienia). Tworzy dwa 8-bitowe sygnały a i b. Sygnałowi a jest przypisywany wektor, którego bardziej znacząca tetrada jest zerami, natomiast mniej znaczącą stanowią wybrane bity sygnału b.

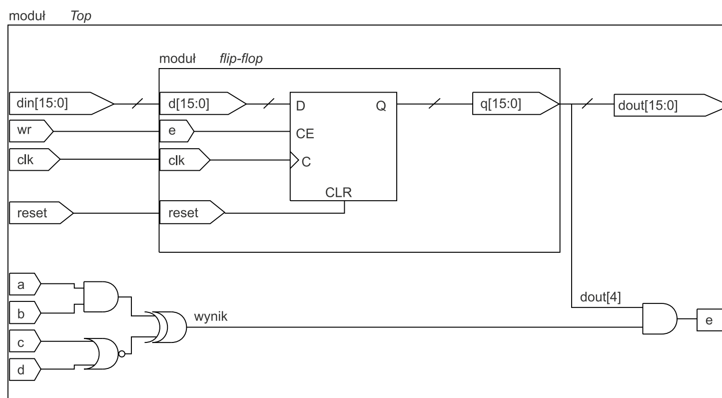
`wire [7:0] a;`
`wire [7:0] b;`
`assign a = b + 1'b1;`
	Tworzy dwa 8-bitowe sygnały a i b. Do sygnału a przypisywana jest suma wartości sygnału b i liczby 1.



Rys. 1. Trzy warianty opisu bramki AND



Rys. 2. Wzorce projektowe z *always*



Rys. 3. Przykładowa struktura hierarchiczna

List. 1. Opis przerzutnika

```
//zespół przerzutników o parametryzowanej długości słowa wejściowego
//rejestr równoległy
module flip_flop
#(parameter dlength = 8)            //parametr
(d, clk, reset, e, q);              //lista sygnałów we/wy

 // deklaracja sygnałów
    input [dlength-1:0] d;                   //wejście D
    input clk;                       //zegar
    input reset;           //asynchroniczny reset
    input e;                         //enable
    output reg [dlength-1:0] q;    //wyjście

//właściwy kod
always@(posedge clk or posedge reset)
      if (reset)
             q   <= {dlength{1'b0}};
       else if (e)
              q  <= d;
endmodule
```

Drugim typem sygnału jest *reg*. Przy czym nie należy sugerować się nazwą – nie zawsze jest to rejestr. Sygnał zdefiniowany jako *reg* może (ale nie musi) przechowywać wartość. Nieodłącznym elementem składni towarzyszącym sygnałom typu *reg* jest *always*. Jego działanie można częściowo porównać do procesu w języku VHDL. A zatem kod zawarty w ramach bloku *always* uwzględniany („wykonywany") tylko wtedy, gdy jest spełniony warunek określony w jego liście wrażliwości.

Załóżmy, że chcemy opisać bramkę AND. Ten sam wynik końcowy można uzyskać stosując sygnały zarówno typy *reg* jak i *wire*, jednak ten pierwszy umożliwia dodatkowo implementację rejestru, a więc pamiętania stanu logicznego. Przykładowe opisy wraz z wynikami ich syntezy przedstawiono na **rys. 1.**

Analizując opis bloków *always* (**rys. 2.**) można wyróżnić pewne wzorce projektowe, umożliwiające syntezę konkretnego rodzaju układu. Słowa kluczowe *posedge* i *negedge*

www.wobit.com.pl

CZUJNIKI
www.czujniki.pl
ENKODERY
www.enkodery.com
SILNIKI DC I BLDC
www.silniki.com
SILNIKI KROKOWE
www.silniki.pl
STEROWNIKI
www.silniki.pl
SPRZĘGŁA
www.silniki.pl
STOLIKI XY
www.stolikixy.pl
PROWADNICE
www.prowadnice.pl
ŚRUBY KULOWE
www.emechanika.com
AKTUATORY
www.silniki.pl
MANIPULATORY
www.manipulatory.com
KOMPONENTY ROBOTYKI
www.mobot.pl

P.P.H.WObit Witold Ober
61-474 Poznań, Gruszkowa 4
tel. 061 8350 - 800, - 620
fax 061 8350 - 704, -804
wobit@wobit.com.pl

oznaczają wrażliwość bloku odpowiednio na narastające i opadające zbocze. Warto zapamiętać te schematy, ponieważ opis bloków *always*, mimo iż poprawny z punktu widzenia składni oraz symulatora, może być niesyntezowalny. Jeśli chodzi o umieszczanie słów *begin* i *end*, to ich stosowanie jest konieczne tylko wtedy, gdy do wykonania jest więcej niż jedno polecenie (np. dwa kolejne przypisania) – używanie ich zawsze nie jest jednak błędem, a wybór sposobu przygotowywania opisu (kodu) zależy od indywidualnych upodobań projektanta.

Podstawową jednostką projektową jest w Verilogu'u moduł. Jego część deklaracyjna zawiera listę sygnałów wejściowych i wyjściowych oraz ich typów. Dodatkowo można określić parametry stosowane do tworzenia sparametryzowanych modeli. Umieszczenie ich w odpowiednim miejscu w deklaracji modułu umożliwia nadpisywanie ich z poziomu jednostki projektowej wyższej w hierarchii.

Użycie modułów zaprezentujemy na prostym przykładzie. Moduł nadrzędny zawierać będzie opis funkcji logicznej oraz zespołu przerzutników typu D, do których wpisywane są dane wejściowe przy narastającym zboczu sygnału *clk*, przy jednocześnie wysokim poziomie na linii *wr*. W celu edukacyjnym opiszemy sparametryzowany model zespołu przerzutników (*flip-flop*) tworzących rejestr równoległy oraz umieścimy go w module nadrzędnym (*top*). Schemat układu, który chcemy opisać przedstawiono na **rys. 3**.

Najpierw przeanalizujmy opis modułu *flip-flop* (**list. 1**). Ponieważ chcemy, by długość słowa wejściowego była parametryzowana (ustalana na wyższym poziomie hierarchii), to przyjmijmy parametr *dlength*, ustawmy jego wartość na 8 i umieśćmy go w odpowiedniej części deklaracji modułu, w celu umożliwienia zewnętrznego nadpisania go. Dzięki temu w innych projektach lub innych modułach tego samego projektu będzie można użyć raz napisanego kodu. Następnie parametryzujemy długości wektorów wejściowych i wyjściowych. Zastosowany wzorzec bloku *always* pozwolił utworzyć przerzutnik typu D z asynchronicznym resetem oraz sygnałem *clock-enable*. Należy zwrócić uwagę na sposób zapisu wyzerowania przerzutnika – użyto tzw. replikacji. Ten wiersz można przeczytać następująco: do wektora q przypisuje się wektor tworzony przez *dlength*-krotne powielenie wartości 1'b0.

Moduł nadrzędny *top* definiujemy podobnie, jednak deklarujemy długość słów stałymi wartościami. W opisie występuje dyrektywa *'timescale* (**list. 2**). Pomijając znaczenie pierwszej liczby, druga definiuje rozdzielczość czasową, którą – by przyspieszyć proces symulacji kosztem dokładności – można zmniejszyć. Aby użyć utworzony już moduł *flip-flop*, najpierw należy podać nazwę tego modułu – w tym przypadku *flip_flop*. Następnie nadpisujemy parametr *dlength* wartością 16. Nazwa *flip_flop_i* jest nazwą konkretnej instancji (może ich być wiele), po której następuje już lista połączeń portów modułu instancjonowanego (osadzanego) z sygnałami modułu nadrzędnego. Funkcja logiczna została zrealizowana dwustopniowo z użyciem wyniku przejściowego (*wynik*), chociaż można było ją opisać w jed-

List. 2. Kod modułu nadrzędnego
```
`timescale 1ns / 1ns //dyrektywa rozdzielczości czasowej

module top(din, clk, reset, wr, dout, a,b,c,d,e);
    input [15:0] din;      //wejście danych
    input clk;             //zegar
    input reset; //reset
    input wr;              //wyzwalanie wejścia
    output [15:0] dout;    //wyjście
         input a;
         input b;
         input c;
         input d;
         output e;

//instancjonowanie modułu przerzutników
flip_flop
#(.dlength(16))  //nadpisanie parametru
flip_flop_i
        (
        .d(din), //łączenie sygnałów
        .clk(clk),
        .reset(reset),
        .e(wr),
        .q(dout)
        );

//funkcja logiczna
wire wynik;
assign wynik=(a & b) ^ !(c || d);
assign e=wynik & dout[4];

endmodule
```

List. 3. Opis przy użyciu instrukcji warunkowej if.. else if .. else:
```
module mux1(sel, din1, din2, din3, din4, dout);
    input [1:0] sel;      //selektor
    input [7:0] din1;     //wektory wejściowe
    input [7:0] din2;
    input [7:0] din3;
    input [7:0] din4;
    output reg [7:0] dout;        //wyjście

always@(sel or din1 or din2 or din3 or din4)
        if (sel==2'b00)
                dout = din1;
        else if (sel==2'b01)
                dout = din2;
        else if (sel==2'b10)
                dout = din3;
        else
                dout = din4;
endmodule
```

List. 4. Opis użyciem instrukcji wybiru case().
```
module mux2(sel, din1, din2, din3, din4, dout);
    input [1:0] sel;      //selektor
    input [7:0] din1;     //wektory wejściowe
    input [7:0] din2;
    input [7:0] din3;
    input [7:0] din4;
    output reg [7:0] dout;        //wyjście

always@(sel or din1 or din2 or din3 or din4)
case(sel)
        2'd0: dout <= din1;
        2'd1: dout <= din2;
        2'd2: dout <= din3;
        2'd3: dout <= din4;
        default: dout <= din1;
endcase
endmodule
```

List. 5. Użycie instrukcji przypisania warunkowego.
```
module mux3(sel, din1, din2, din3, din4, dout);
    input [1:0] sel;      //selektor
    input [7:0] din1;     //wektory wejściowe
    input [7:0] din2;
    input [7:0] din3;
    input [7:0] din4;
    output [7:0] dout;    //wyjście

assign dout = (sel==2'd0)? din1 :
           (sel==2'd1)? din2 :
           (sel==2'd2)? din2 : din4;
endmodule
```

List. 6. Dwukierunkowy licznik rewersyjny z synchronicznym wpisem równoległym i zerowaniem asynchronicznym

```
module licznik
#(parameter length = 4)
(clk, reset, dir, counter, load, data);
   input clk;           //zegar
   input reset;    //asynchroniczny reset
   input dir;           //kierunek zliczania
   output reg [length-1:0] counter;
   input load;     //strobe wpisu
   input [length-1:0] data; //dane do wpisu

always@(posedge clk or negedge reset)
if (!reset)      //asynchroniczny reset
    counter <= {length{1'b0}};
else if (load)
     counter <= data;     //wpis równoległy
else
     if (dir)             //zliczanie w górę
            counter <= counter + 1'b1;
      else                //zliczanie w dół
            counter <= counter - 1'b1;
endmodule
```

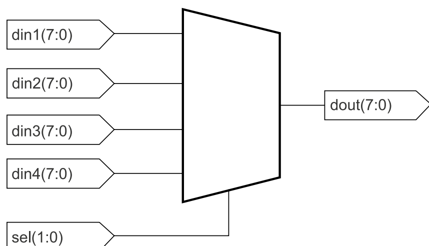
List. 7. Opis układu preskalera

```
module preskaler
#(parameter length = 4, parameter count=5)
(clk, reset, counter, impulse);
    input clk;             //zegar
    input reset; //asynchroniczne zerowanie
    output reg [length-1:0] counter;
     output impulse;      //impuls preskalera

always@(posedge clk or negedge reset)
if (!reset)               //asynchroniczny reset
        counter <= {length{1'b0}};
else if (impulse)         //wpis do licznika
        counter <= count;
else                      //dekrementacja licznika
        counter <= counter - 1'b1;

//warunek na reset licznika
assign impulse=(counter==0) ? 1'b1 : 1'b0;

endmodule
```



Rys. 4. Symbol multipleksera w RTL

nej linijce. Ma to sens wtedy, gdy podczas symulacji układu chcemy wyświetlić przebieg właśnie w węźle nazwanym *wynik*, to tylko opisując ten sygnał jawnie jesteśmy w stanie obejrzeć go w symulatorze.

Po lekturze powyższej części artykułu można już zorientować się w ogólnej strukturze języka. W dalszej części przedstawimy opis podstawowych bloków funkcjonalnych, takich jak multipleksery i liczniki.

Multipleksery

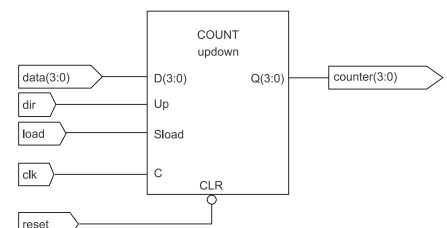
Przyjmijmy, że 8-bitową magistralą chcemy przesyłać kolejno cztery 8-bitowe słowa pochodzące z różnych źródeł. Potrzebnych jest do tego 8 multiplekserów 4-wejściowych (4x1), czyli multiplekser grupowy o symbolu graficznym przedstawionym na **rys. 4**. W tym punkcie przedstawimy opis takiego multipleksera z użyciem trzech różnych konstrukcji językowych (**list. 3**, **list. 4**, **list. 5**). Każdy z tych opisów jest syntezowany do tego samego układu, więc wybór jednego z nich może zależeć od indywidualnych preferencji projektanta.

Przy projektowaniu multiplekserów należy zwracać uwagę, aby opisać go dla wszystkich możliwych kombinacji wartości sygnałów przełączających (adresujących). W przeciwnym przypadku podczas syntezy zostanie utworzony zatrzask i dodatkowa struktura logiczna – w oprogramowaniu firmy Xilinx wykrycie zatrzasku jest sygnalizowane ostrzeżeniem.

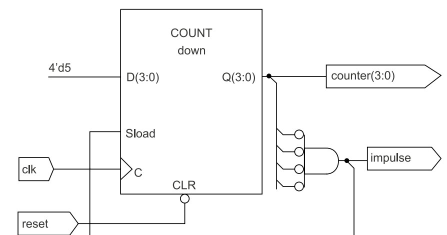
Liczniki

Przedstawimy dwa przykładowe wzorcowe opisy, które mogą być modyfikowane dla własnych aplikacji.

Pierwszym jest licznik dwukierunkowy o sparametryzowanej długości, z asynchronicznym zerowaniem i możliwością wpisu równoległego (**list. 6**). Symbol graficzny tego licznika z *RTL schematic* przedstawiono **rys. 5**. Należy zwrócić uwagę na sposób przypisania zer podczas zerowania do wektora o programowalnej długości. Konstrukcja {*length*{*1'b0*}} jest replikacją. Jest ona przydatna szczególnie przy używaniu modeli sparametryzowanych. Tworzy ona wektor będący powieleniem wartości znajdującej się w wewnętrznych nawiasach {}, a liczba znajdująca się przed nim (tu *length*) określa krotność powielenia. Należy przy tym pamiętać, że krotność musi być wartością stałą – w innym przypadku struktura ta jest niesyntezowalna.



Rys. 5. Symbol RTL licznika przedstawionego na list. 6



Rys. 6. Schemat RTL generatora przedstawionego na list. 7

Powyższy wzorzec licznika może być użyty do utworzenia sparametryzowanego generatora impulsów, który po określonej liczbie cykli zegara będzie generować impuls trwający jeden cykl zegarowy (**list. 7**). Nazwijmy taki układ preskalerem (**rys. 6**). Zastosowań takiego impulsu jest wiele – jednym z nich jest obsługa zdarzeń wolnozmiennych w stosunku do zegara systemowego. Licznik (rys. 6) odlicza w dół (jest dekrementowany) od pewnej wartości (w tym przypadku parametr *count* = *5*). Gdy licznik osiągnie stan zerowy, to na wyjście impuls podawany jest poziom wysoki i wpisywana jest sparametryzowana wartość *count* do licznika (5). Ponieważ model ma dwa parametry: liczbę bitów licznika (*length*) i liczbę, od której licznik ma liczyć (*count*), to należy zwrócić uwagę na wzajemną zgodność wartości obu parametrów.

Mamy nadzieję, że to wprowadzenie i pierwsze, łatwe w interpretacji, przykłady opisu bloków funkcjonalnych, zachęcą Czytelników do zainteresowania się kolejnymi artykułami tego cyklu.

Krzysztof Kasiński
krzysztof.kasinski@o2.pl
home.agh.edu.pl/kasinski

forum.ep.com.pl