Język Verilog w przykładach (2)

Układ sterowania 4-cyfrowym wyświetlaczem 7-segmentowym LED

W tej części kursu opiszemy bloki funkcjonalne układu sterowania 4-cyfrowym wyświetlaczem 7-segmentowym w trybie multipleksowanym. Następnie zaprezentujemy sposób ich połączenia w module nadrzędnym opisu hierarchicznego.

Jest to przykład zastosowania w jednym układzie: rejestru przesuwnego, licznika pierścieniowego (zbudowanego w oparciu o rejestr przesuwający z krążącą jedynką), konwertera kodu binarnego (BIN) liczby na kod BCD (binarny kod dziesiętny) oraz dekodera kodu binarnego na kod wskaźnika 7-segmentowego.

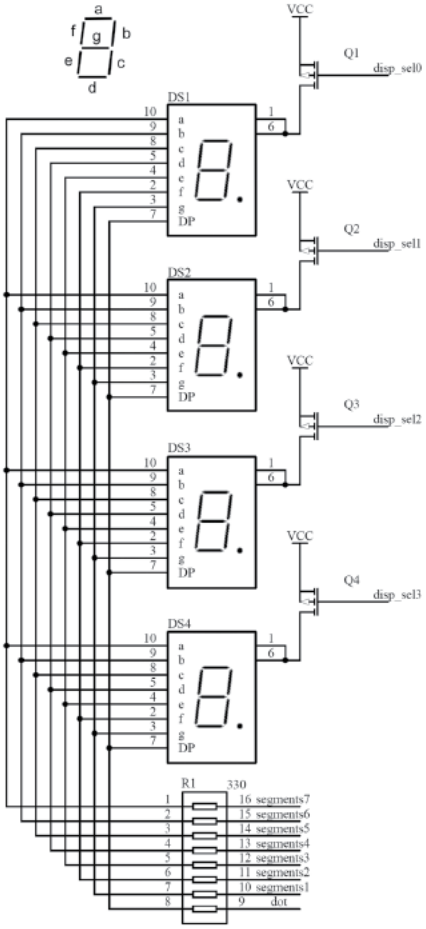
Na **rys. 7** przedstawiono schemat dołączenia wyprowadzeń wyświetlacza. Linie sterujące poszczególnymi segmentami są wspólne dla wszystkich wyświetlaczy, natomiast zasilanie anody każdego z nich jest kontrolowane przez tranzystory typu p-MOS. Takie rozwiązanie umożliwia obsługę wielu modułów wyświetlaczy przy niewielkiej liczbie linii sterujących. Przykładowo, w tym układzie użyto tylko 12 linii, zamiast 32 niezbędnych przy wyświetlaniu statycznym (każdy moduł oddzielnie)

Zaprojektujmy dekoder 4-bitowej liczby binarnej na kod wyświetlacza 7-segmentowego, zakładając wyświetlanie na nim cyfr kodu szesnastkowego (jednak w tym przykładzie cyfry A ... F nie będą wyświetlane). Przyjmijmy, że poziom niski napięcia na linii sterującej powoduje świecenie segmentu (wyświetlacz ze wspólną anoda). Kod wyjściowy dekodera przedstawiono na **list. 8**. Do opisu dekodera użyto instrukcji *case*. W wyniku syntezy układu na podstawie tego opisu powstanie odpowiednia sieć bramek logicznych lub będzie użyta rozproszona pamięci RAM (jeśli docelowy układ FPGA ma taką) skonfigurowana jako pamięć ROM.

Mając dekoder możemy wyświetlić cyfry 0...9. Są jednak cztery moduły wyświetlaczy. Przyjmijmy zatem, że chcemy mieć możliwość wyświetlenia liczb całkowitych 0...9999. Dla takiego zakresu liczb w kodzie BCD potrzebnych jest 14 bitów dla liczby w kodzie BIN. Następnie niezbędna jest konwersja tej liczby na liczbę w kodzie BCD. Spośród znanych metod konwersji, metody arytmetyczne sprawdzają się tylko w systemach mikroprocesorowych, gdyż nie wszystkie układy FPGA mają sprzętowe multiplikatory, a ich synteza w układach programowalnych nie jest efektywna. Konwersja tablicowa jest możliwa, ale konieczne jest wówczas użycie dużych bloków pamięci, a więc powinna być stosowana tylko w przypadku, gdy czas przetwarzania jest krytyczny.

	Kod BCD					Kod BIN			
	Dziesiatki	Jedności							
Shift						1	1	0	0
Shift					1	1	0	0	
Shift				1	1	0	0		
≥5			1	1	0	0			
Plus 3		0	0	1	1				
Shift		1	0	0	1				
Koniec	1	0	0	1	0				
Dziesiętnie	1	2				12			

Rys. 7. Połączenie wyprowadzeń segmentowych 4-cyfrowego wyświetlacza sterowanego multipleksowo



Rys. 8. Ilustracja algorytmu *shift plus 3*

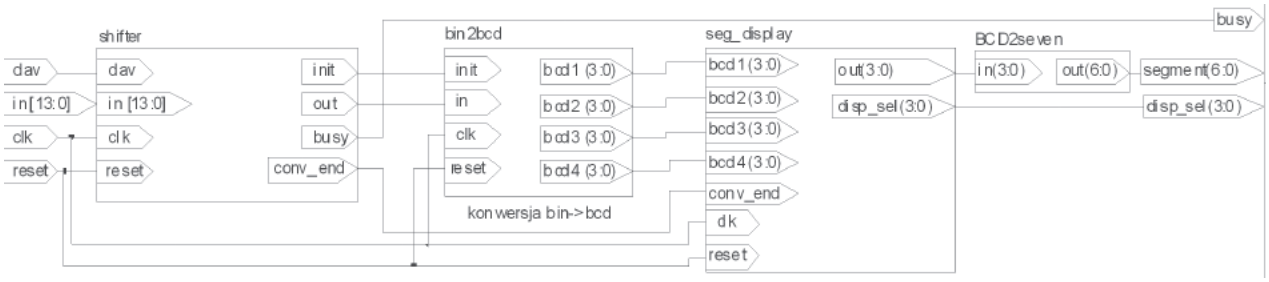
List. 8. Dekoder kodu BIN na kod wskaźnika 7-segmentowego

```
module bin2seven(in,out);
input [3:0] in;  //liczba binarna
output reg [6:0] out;     //linie sterujące wyśw.
//dekoder
always@(in)
case(in)
         4'h0: out<=7'b0000001;
         4'h1: out<=7'b1001111;
         4'h2: out<=7'b0010010;
         4'h3: out<=7'b0000110;
         4'h4: out<=7'b1001100;
         4'h5: out<=7'b0100100;
         4'h6: out<=7'b0100000;
         4'h7: out<=7'b0001111;
         4'h8: out<=7'b0000000;
         4'h9: out<=7'b0000100;
         4'hA: out<=7'b0001000;
         4'hB: out<=7'b1100000;
         4'hC: out<=7'b0110001;
         4'hD: out<=7'b1000010;
         4'hE: out<=7'b0110000;
         4'hF: out<=7'b0111000;
default: out<=7'b0000001;
endcase
endmodule
```

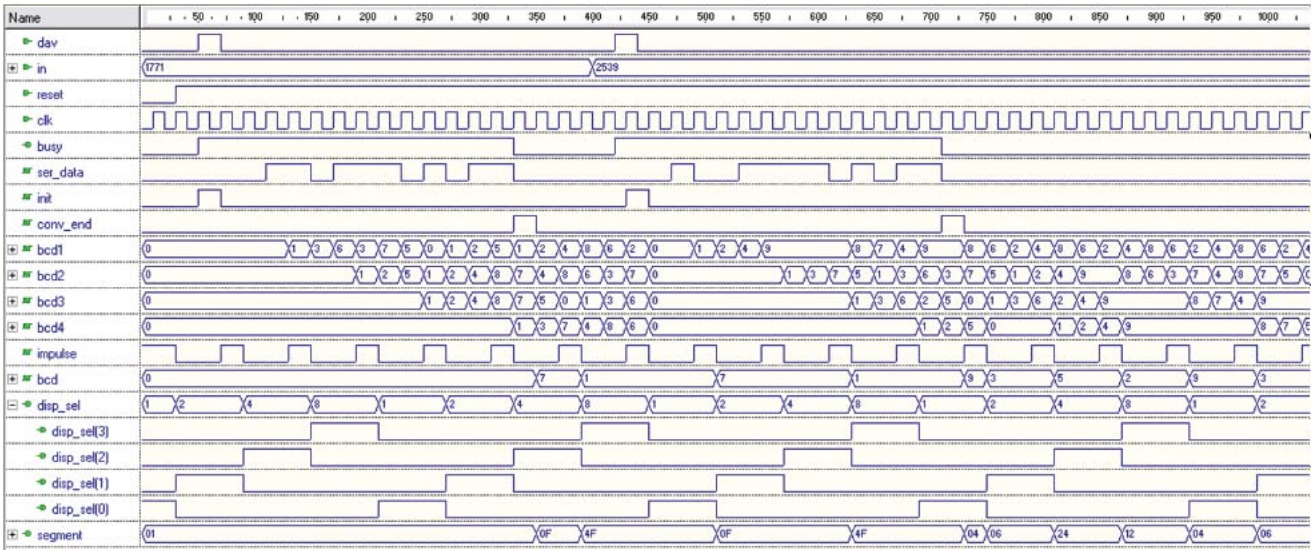
List. 9. Moduł konwertera kodu BIN na kod BCD

```
module bin2bcd_cell(clk,reset,init,mod_in,mod_out,Q);
input clk;                //zegar
input reset;              //reset
input mod_in;    //wejście bitu przeniesienia
input init;               //początek konwersji
output mod_out;  //wyjście bitu przeniesienia
output [0:3] Q;  //wyjście BCD

reg [0:3] Q_actual;       //rejestr
reg [0:2] Q_next;//wyjście dekodera następnej wartości
reg mod_out_next;//pośrednia wartość bitu przeniesienia

always@(Q_actual)//dekoder następnej wartości
         case (Q_actual)
                  4'd0: Q_next <= 3'd0;
                  4'd1: Q_next <= 3'd1;
                  4'd2: Q_next <= 3'd2;
                  4'd3: Q_next <= 3'd3;
                  4'd4: Q_next <= 3'd4;
                  4'd5: Q_next <= 3'd0;
                  4'd6: Q_next <= 3'd2;
                  4'd7: Q_next <= 3'd4;
//                4'd8: Q_next <= 3'd3;
//                4'd9: Q_next <= 3'd4;
                  default: Q_next <= 3'd0;
         endcase

always@(Q_actual)//kontrola bitu przeniesienia
         case (Q_actual)
                  4'd5: mod_out_next <= 1'b1;
                  4'd6: mod_out_next <= 1'b1;
                  4'd7: mod_out_next <= 1'b1;
//                4'h8: mod_out_next <= 1'b1;
//                4'h9: mod_out_next <= 1'b1;
                  default: mod_out_next<= 1'b0;
         endcase

assign mod_out = mod_out_next;     //opóźnienie bitu przeniesienia
assign Q = Q_actual;               //ciągłe przepisanie na wyście

always@(posedge clk or negedge reset)
         if(!reset)
                  Q_actual<= 4'b0;          //reset
         else
                  if (init) begin  //początek konwersji
                           Q_actual <= 4'b0;
                  end else         //krok konwersji
                           Q_actual <= {Q_next ,mod_in};
endmodule
```

W tym przykładzie do konwersji kodu BIN na BCD zastosujemy prosty algorytm *shift plus 3* (stosowany także w kalkulatorach) polegający na przesuwaniu w lewo (mnożenie przez 2) z rejestru równoległo-szeregowego (na przykład 4-bitowego) wpisanej do niego liczby w kodzie BIN. Rejestry te będą po zakończeniu konwersji zawierały wynik w postaci kodu BCD. Jeżeli po przesunięciu wartość dziesiętna w rejestrze docelowym (w jego czterech bitach) jest większa lub równa 5 (dziesiętnie), to do zawartości tego rejestru należy dodać 3 (dwójkowo 0011), po czym następuje kolejne przesunięcie w lewo i wówczas występuje bit przeniesienia 1 (do dekady dziesiątek), a zawartość tetrady jest skorygowana do właściwej postaci BCD (jedności). Dodanie korygującej trójki przed mnożeniem przekształcanej liczby przez 2 (przesunięcie w lewo o 1 bit) odpowiada odjęciu od korygowanej tetrady jedności liczby dziesiętnej 10 (to znaczy dodania jej kodu uzupełnieniowego do 2, to jest 0110 dwójkowo). Na przykład, gdy aktualna zawartość rejestru BCD wynosi 6, to po dodaniu 3 będzie 9, a po przesunięciu w lewo 18, czyli dwójkowo 1 0010 (prawidłowy kod po konwersji do BCD liczby 1100



Rys. 9. Schemat połączeń opisywanych bloków



Rys. 10. Przebiegi czasowe wybranych sygnałów w module wyświetlacz

w kodzie BIN) – **rys. 8**. Nieco zmodyfikowany opis tego algorytmu (nie ma dodawania 3 przed przesunięciem, natomiast zastosowano odpowiednie przypisania modyfikujące) przedstawiono na **list. 9**. Łącząc kaskadowo opisane wyżej rejestry w większe zespoły, można przetwarzać dowolnie długie liczby binarne. Konwersja tą metodą jest zakończona po wprowadzeniu do rejestru BCD wszystkich bitów przekształcanej liczby, a więc trwa zawsze *n* cykli zegarowych, gdzie *n* jest ilością jej bitów. Należy pamiętać aby przed konwersją wyzerować rejestry sygnałem *init*. Układ konwersji działający według tego algorytmu zajmuje niewielką część zasobów FPGA i jest szybki.

W tym układzie konwersji dane muszą być wprowadzane w postaci szeregowej, dlatego należy zaprojektować moduł, którego zadaniem będzie odbieranie danych w postaci równoległej i ich przekształcenie do postaci szeregowej, następnie dekodowanie otrzymanego kodu do postaci kodu wyświetlacza 7-segmentowego oraz wygenerowanie odpowiednich sygnałów dla wyświetlacza. Układ sterowania wyświetlaczem składa się z pięciu części. Pierwszy (*shifter*) oczekuje na wpis danych sygnalizowany sygnałem *dav*. Są one równolegle wpisywane do rejestru przesuwającego *sreg*. Moduł generuje sygnał zajętości (*busy*), zeruje konwertery *bin2bcd* i odlicza 14 taktów zegarowych dla rejestru przesuwającego, po czym generuje sygnał zakończenia konwersji (*conv_end*)

List. 10. Moduł rejestru przesuwającego kodu przekształcanej liczby dwójkowej *(Shifter)*

```
module shifter (clk,reset,in,dav,busy,out,init,conv_end);
        input clk;
        input reset;
        input [13:0] in;
        input dav;
        output busy;
        output out;
        output reg init;
        output reg conv_end;

parameter s_idle = 1'b0;  //stany
parameter s_shift = 1'b1;
reg state;                                          //rejestr stanu

reg [13:0] sreg; //rejestr przesuwny
reg [4:0]counter;//licznik przesunięć

//wpis i przesunięcie
always@(posedge clk or negedge reset)
if(!reset) begin
        counter  <= 5'b0;
        sreg     <= 15'b0;
        init     <= 1'b0;
        state    <= s_idle;
        conv_end <= 1'b0;
end else begin
        init     <= 1'b0;
        conv_end<= 1'b0;
        counter <= counter - 4'b1;
        sreg    <= {sreg[12:0],1'b0};
        case (state)
        s_idle: //czeka na nową daną
                if (dav) begin
                        state <= s_shift;
                        sreg <= in;
                        init <= 1'b1;
                        counter <= 4'd13;
                end else
                        state   <= s_idle;
        s_shift: //wsuwanie danych do konwertera
                if(counter==4'h0) begin
                        state          <= s_idle;
                        conv_end <= 1'b1;
                end
        default: state<= s_idle;
        endcase
end
//sygnalizacja gotowości
assign busy=dav||(state==s_shift);
assign out=sreg[13];

endmodule
```

List. 11. Moduł zbiorczy konwertera kodu BIN na kod BCD

```
module bin2bcd (in, clk, init,reset,bcd1,bcd2,bcd3,bcd4);
        input in;          //wejście z rejestru przesuwnego
        input clk;
        input reset;
        input init;
        output [3:0] bcd1;        //wyjścia w kodzie bcd
        output [3:0] bcd2;
        output [3:0] bcd3;
        output [3:0] bcd4;
//sygnały pośrednie
wire  mod1_2;
wire  mod2_3;
wire  mod3_4;

//instancjonowanie komórek konwertera BIN2BCD
bin2bcd_cell bin2bcd_cell1(
                                .clk(clk),
                                .reset(reset),
                                .init(init),
                                .mod_in(in),
                                .mod_out(mod1_2),
                                .Q(bcd1));

bin2bcd_cell bin2bcd_cell2(
                                .clk(clk),
                                .reset(reset),
                                .init(init),
                                .mod_in(mod1_2),
                                .mod_out(mod2_3),
                                .Q(bcd2));

bin2bcd_cell bin2bcd_cell3(
                                .clk(clk),
                                .reset(reset),
                                .init(init),
                                .mod_in(mod2_3),
                                .mod_out(mod3_4),
                                .Q(bcd3));

bin2bcd_cell bin2bcd_cell4(
                                .clk(clk),
                                .reset(reset),
                                .init(init),
                                .mod_in(mod3_4),
                                .mod_out(),
                                .Q(bcd4));
endmodule
```

List. 12. Moduł obsługi wyświetlaczy

```
module seg_display (clk, conv_end, reset, impulse,
                                bcd1,bcd2,bcd3,bcd4,bcd_out,disp_sel);
        input clk;
        input reset;
        input impulse;   //impulsy z preskalera
        input conv_end;  //sygnał końca konwersji
        input [3:0] bcd1; //wejścia w bcd
        input [3:0] bcd2;
        input [3:0] bcd3;
        input [3:0] bcd4;
        output [3:0] bcd_out; //wybrana cyfra
        output [3:0] disp_sel;//wybór wyświetlacza

        reg [3:0] selector; //bufor cykliczny wyboru wyświetlacza
        always@(posedge clk or negedge reset)
        if (!reset)
                selector <= 4'b1;
        else if (impulse)
                selector <= {selector[2:0],selector[3]};

        assign disp_sel = selector;

        //rejestry zatrzaskujące
        reg [3:0] bcd1_reg;
        reg [3:0] bcd2_reg;
        reg [3:0] bcd3_reg;
        reg [3:0] bcd4_reg;

        always@(posedge clk or negedge reset)
                if (!reset) begin
                        bcd1_reg <= 4'b0;
                        bcd2_reg <= 4'b0;
                        bcd3_reg <= 4'b0;
                        bcd4_reg <= 4'b0;
                end else if (conv_end) begin
                        bcd1_reg <= bcd1;
                        bcd2_reg <= bcd2;
                        bcd3_reg <= bcd3;
                        bcd4_reg <= bcd4;
                 end

        assign bcd_out = (selector==4'b0001)?bcd1_reg:
                                        (selector==4'b0010)?bcd2_reg:
                                        (selector==4'b0100)?bcd3_reg:
                                        (selector==4'b1000)?bcd4_
reg:4'b0000;
        endmodule
```

i powraca do stanu oczekiwania na nowy wpis. Należy zauważyć, że zarówno licznik jak i rejestr przesuwny działają niezależnie od aktualnego stanu przerzutnika *state*. Wybierając odpowiednio warunki przełączania tego przerzutnika można zmniejszyć liczbę użytych elementów nie realizując obwodów zatrzymania licznika i rejestru przesuwającego na czas oczekiwania na nową daną. Opis tego modułu jest zamieszczony na **list. 10**. Ponieważ wyświetlacz ma cztery cyfry, to potrzebnych jest tyle samo dekad (tetrad dla zapisu poszczególnych cyfr w postaci dwójkowej) po konwersji na BCD. W opisie na **list. 11** instancjonowane są te komponenty i odpowiednio łączone między sobą. Na wejście pierwszego konwertera podawany jest najbardziej znaczący bit z rejestru przesuwnego – modułu *shifter* (*sreg[13]*). W kolejnym module (*seg_display*) za pomocą sygnału końca konwersji (*conv_end*) zatrzaskiwane są dane wyjściowe – cztery cyfry w kodzie BCD (**list. 12**). Do przełączania wyświetlaczy zastosowano licznik pierścieniowy liczący w kodzie 1 z 4 (*selector*). Wpisana po zerowaniu do tego licznika jedynka jest cyklicznie przesuwana, wskutek czego są wybierane kolejne wyświetlacze. Na podstawie aktualnego stanu tego licznika wyświetlana jest na odpowiednim wyświetlaczu cyfra BCD. Na końcu dołączono konwerter kodu cyfry BCD na kod wyświetlacza 7-segmentowego (*bin2seven*) tak, aby można było bezpośrednio sterować liniami wyświetlacza. Ponieważ wyświetlaczy nie można przełączać ze zbyt dużą częstotliwością, zastosowano preskaler (cz. 1. kursu, list. 7), który „spowalnia" taktowanie licznika pierścieniowego (*selector*). Na **list. 13** połączono te moduły w jeden blok. Graficzne przedstawienie połączeń jest zaprezentowane na **rys. 9**.

W celu sprawdzenia działania całego układu sterującego przeanalizujemy wyniki symulacji z **rys. 10**. Liczby 1771 i 2539 są wpisywane do modułu, odpowiednio w 50. i 420. nanosekundzie procesu symulacji, a konwersja jest ukończona odpowiednio w 330. i 710. nanosekundzie. Przez cały czas symulacji widoczna jest praca sterownika wyświetlaczy (sygnały *disp_sel* i *bcd*). O ile w opisanym przykładzie przełączanie wyświetlaczy odbywa się co 20 ms, to na rys. 2.4 przebiegi te zostały „przyspieszone" w celu umożliwienia ich zaprezentowania na jednym ekranie (przy określonej częstotliwości sygnału taktującego *clk*.

W celu zwiększenia czytelności kodu, cztery moduły dekodera zostały zainstancjonowane oddzielnie. Jest to jednak niepraktyczne w przypadku konieczności stosowania większej ich liczby. Język Verilog umożliwia bardziej zwarty opis wielokrotnego instancjonowania tego samego modułu. Na **list. 14** jest zamieszczony fragment kodu prezentujący użycie tej konstrukcji w opisanym module.

List. 13. Moduł nadrzędny

```
module wyswietlacz (clk, reset, in, dav, busy, disp_sel, segment);
        input clk;       //50Mhz
        input reset;
        input [13:0] in;
        input dav;
        output busy;
        output [3:0] disp_sel;    //wybór wyświetlacza
        output [6:0] segment;     //sterowanie segmentami

        wire ser_data,init,conv_end;
        //shifter
        shifter shifter_i(
                .clk(clk),
                .reset(reset),
                .in(in),
                .dav(dav),
                .busy(busy),
                .out(ser_data),
                .init(init),
                .conv_end(conv_end));

        wire [3:0] bcd1,bcd2,bcd3,bcd4;
        //konwerter bin->bcd
        bin2bcd bin2bcd_i(
        .in(ser_data),
        .clk(clk),
        .init(init),
        .reset(reset),
        .bcd1(bcd1),
        .bcd2(bcd2),
        .bcd3(bcd3),
        .bcd4(bcd4));

        //preskaler 20ms
        wire impulse;
        licznik #(.length(20), .count(1000000))
        prescaler_20ms (.clk(clk), .reset(reset), .counter(),
.impulse(impulse));

        wire [3:0] bcd;
        //moduł wyświetlający
        seg_display seg_display_i(
                .clk(clk),
                .conv_end(conv_end),
                .reset(reset),
                .impulse(impulse),
                .bcd1(bcd1),
                .bcd2(bcd2),
                .bcd3(bcd3),
                .bcd4(bcd4),
                .bcd_out(bcd),
                .disp_sel(disp_sel));

        //konwerter BCD na kod 7-segmentowy
        BCD2seven BCD2seven(
        .in(bcd),
        .out(segment));

endmodule
```

List. 14. Fragment kodu z przykładem wielokrotnego instancjonowania (bardziej zwarty opis niż na list. 11)

```
wire [3:0] mod1; //sygnały wejściowe
wire [3:0] mod2; //sygnały wyjściowe
assign mod1[3:1]=mod2[2:0];       //połączenia pomiędzy komórkami
assign mod1[0]=temp[13];  //przypisanie wejścia do pierwej komórki
wire [15:0] bcd_int;              //opis wyjść BCD
//instancjonowanie czterech komórek konwertera
bin2bcd_cell bin2bcd_celli [3:0](.clk(clk),.reset(clrn),.init(init),.mod_
in(mod1),.mod_out(mod2),.Q(bcd_int));

assign bcd1_int = bcd_int[3:0];   //odpowiednie rozdzielenie sygnałów
assign bcd2_int = bcd_int[7:4];
assign bcd3_int = bcd_int[11:8];
assign bcd4_int = bcd_int[15:12];
```

Ukończyliśmy pierwszy prawdziwy projekt. Zajmuje on około 7% zasobów najmniejszego układu FPGA z rodziny Spartan 3E firmy Xilinx. W kolejnej, trzeciej części przedstawimy przykład opisu pamięci RAM implementowanej w FPGA.

Krzysztof Kasiński
krzysztof.kasinski@o2.pl