Eksperymenty z FPGA (11)

Operacje mnożenia liczb

W artykule zapoznamy się z operacją mnożenia liczb w układach FPGA i przygotujemy moduł wykonujący mnożenie liczb zespolonych. Potem przetestujemy go dla liczb przesyłanych przez port szeregowy, a w kolejnym odcinku będzie to jeden z bloków budujących naszą implementację transformaty Fouriera. Tak jak poprzednio, przed przystąpieniem do wykonywania eksperymentów, zachęcam do aktualizacji repozytorium z przykładami [1] (choćby poprzez wywołanie polecenia `git pull`).

Mnożenie to działanie, którego implementacja z samych tablic LUT jest dość złożona. Z drugiej strony, jest bardzo popularna przy cyfrowym przetwarzaniu sygnałów. Zauważyli to także producenci układów FPGA i postanowili rozwiązać ten problem poprzez dodanie sprzętowych bloków, realizujących funkcje mnożenia. W dokumentacji spotyka się je pod różnymi nazwami. W produktach Intela bardziej zaawansowane bloki noszą nazwę DSP (DSP blocks). W rodzinie MAX10 zaimplementowana została uproszczona wersja, nazwana po prostu „mnożarką" (*embedded multiplier*), której opis znajdziemy na [2].

Uproszczony schemat mnożarki pokazuje **rysunek 1**. Mamy tam dwa wejścia przyjmujące wektory bitów, interpretowane jako czynniki. Mogą one, ale nie muszą, zostać zatrzaśnięte we wbudowanym rejestrze. Jego zastosowanie pozwala na uzyskanie wyższej częstotliwości pracy. Następnie mamy logikę kombinacyjną, realizującą mnożenie. Iloczyn, podobnie jak czynniki, może zostać zatrzaśnięty w rejestrze wyjściowym. Dodatkowo, dla każdego wejścia można skonfigurować liczbę w formie ze znakiem albo bez znaku.

Pojedyncza mnożarka może wykonywać pojedyncze mnożenie liczb 18-bitowych albo dwa mnożenia liczb 9-bitowych. Do przechowywania wyniku mnożenia dwóch czynników, zapisanych odpowiednio na N1 i N2 bitach, potrzebnych jest N1+N2 bitów. Łatwo możemy policzyć, że uzyskane wyniki będą miały odpowiednio 36 albo 18 bitów długości. Mnożarki mogą być użyte dla czynników mniejszej długości. Niewykorzystane bity zostaną wypełnione bitem znaku.

W układach z rodziny MAX10, w zależności od wersji, może być od 16 do 144 niezależnych mnożarek. Płytka Rysino wyposażona jest w FPGA 10M04, który zawiera ich 20.

Mnożenie liczb zespolonych

Aby zademonstrować sposób użycia nowo poznanych bloków, zaimplementujemy mnożenie zespolone. Zaczniemy więc od bardzo krótkiej powtórki z algebry. Mamy dwie liczby zespolone a i b, które rozpiszemy jako:

$$a=a_{re}+ja_{im}$$
$$b=b_{re}+jb_{im}$$

gdzie a_{re} i b_{re} odpowiadają części rzeczywistej, a a_{im} i b_{im} części urojonej; j symbolizuje jednostkę urojoną, czyli pierwiastek z -1. Jeśli rozpiszemy mnożenie, otrzymamy:

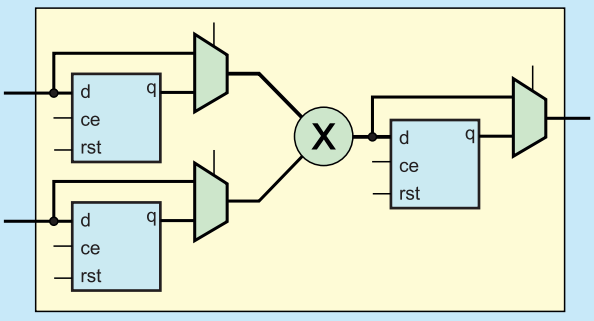
$$ab=(a_{re}+ja_{im})(b_{re}+jb_{im})=a_{re}b_{re}+ja_{re}a_{im}+ja_{im}b_{re}-a_{im}b_{im}=$$
$$(a_{re}b_{re}-a_{im}b_{im})+j(a_{re}a_{im}+a_{im}b_{re})$$

Mnożenie dwóch liczb zespolonych możemy zatem zrealizować jako cztery mnożenia i dwa dodawania. Istnieją wprawdzie sprytne sposoby zmniejszenia liczby mnożeń do 3, lecz pozostaniemy przy bardziej klasycznym sposobie [3].

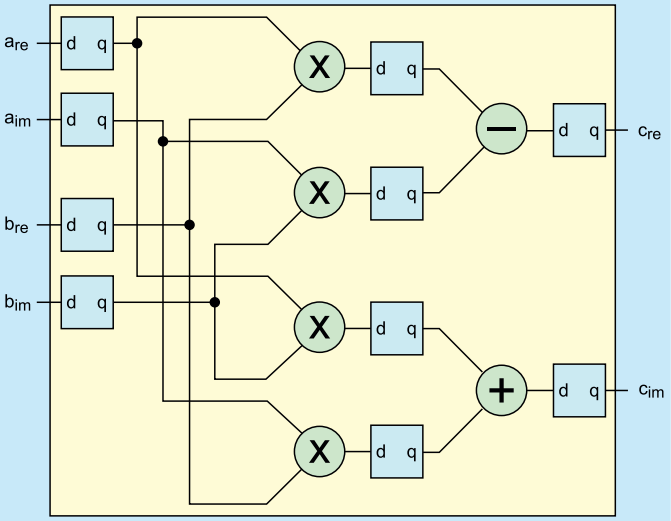
Na **rysunku 2** pokazano sposób implementacji uzyskanego przez nas wzoru w układzie FPGA. Moduł przyjmuje cztery wektory wejściowe. Najpierw są one zatrzaskiwane w rejestrach. W drugim takcie następuje równoległe wykonanie czterech mnożeń rzeczywistych. Uzyskane wyniki są zatrzaskiwane w kolejnych rejestrach. W trzecim takcie następuje wykonanie dodawania i odejmowania. Wynik jest zapisywany w rejestrach wyjściowych. Tak przygotowany moduł będzie miał latencję trzech taktów zegara.

Kod realizujący mnożenie liczb zespolonych pokazuje **listing 1**. Stworzony moduł nosi nazwę `c_mul`. W liniach 11...13 zdefiniowane są trzy parametry. K definiuje długość wektora danych wejściowych. `K_OUT1` i `K_OUT2` decydują, które z bitów końcowego wyniku mają znaleźć się na wyjściu. Dalej (linie 15...22) znajdują się wejścia i wyjścia. Ponieważ chcemy, aby działania zostały wykonane na liczbach ze znakiem, przy definicji wektorów używamy słowa kluczowego `signed`.

Kolejną częścią jest definicja zmiennych pomocniczych. Tutaj także używamy słowa kluczowego `signed`. Pierwsze cztery (linie



Rysunek 1. Budowa mnożarki wbudowanej w układ Max10



Rysunek 2. Realizacja mnożenia liczb zespolonych

24…27) będą tworzyły rejestry zatrzaskujące wejścia. Ich długość jest więc równa K. Kolejne cztery będą przechowywały wyniki poszczególnych mnożeń, stąd ich długość musi być odpowiednio większa i wynosi 2K. Na końcu (34…35) znajdują się rejestry wyjściowe. Zarówno dodawanie, jak i odejmowanie wymaga wydłużenia o jeden dodatkowy bit, stąd otrzymujemy wektory o długościach 2K+1.

Teraz możemy przejść do implementacji samej logiki modułu. Składają się na nią trzy bloki always_ff. Pierwszy (37…42) odpowiada za zatrzaskiwanie danych wejściowych. Najciekawszą funkcję realizuje kolejny blok (44…49), gdzie następuje mnożenie. Jest ono realizowane po prostu poprzez użycie operatora *, podobnie jak w wielu innych językach. Dopiero przy budowaniu projektu środowisko Quartus wykryje, że to działanie może zostać wykonane w specjalnym bloku i wyinferuje mnożarkę. Możemy także, poprzez dodanie odpowiednich ustawień, wymusić, aby zamiast wykorzystania specjalnych modułów, mnożenie zostało złożone ze zwykłych elementów logicznych.

W ostatnim bloku always_ff realizowanie jest dodawanie i odejmowanie. Na samym końcu (linie 56…57) następuje wycięcie zadanych bitów wyniku. Za pomocą polecenia assign są one przypisane do wyjść modułu.

Listing 1. Implementacja mnożenia liczb zespolonych (12_mul/c_mul.sv)

```
10 module c_mul #(
11   parameter K = 9,
12   parameter K_OUT1 = 8,
13   parameter K_OUT2 = 16
14 ) (
15   input wire clk,
16   input wire rst,
17   input wire signed [K-1:0]a_re,
18   input wire signed [K-1:0]a_im,
19   input wire signed [K-1:0]b_re,
20   input wire signed [K-1:0]b_im,
21   output logic signed [K_OUT2-K_OUT1:0]c_re,
22   output logic signed [K_OUT2-K_OUT1:0]c_im
23 );
24   logic signed [K-1:0]a_re_r;
25   logic signed [K-1:0]a_im_r;
26   logic signed [K-1:0]b_re_r;
27   logic signed [K-1:0]b_im_r;
28
29   logic signed [2*K-1:0] c_re_1;
30   logic signed [2*K-1:0] c_re_2;
31   logic signed [2*K-1:0] c_im_1;
32   logic signed [2*K-1:0] c_im_2;
33
34   logic signed [2*K:0] c_re_tmp;
35   logic signed [2*K:0] c_im_tmp;
36
37   always_ff @(posedge clk) begin
38     a_re_r <= a_re;
39     a_im_r <= a_im;
40     b_re_r <= b_re;
41     b_im_r <= b_im;
42   end
43
44   always_ff @(posedge clk) begin
45     c_re_1 <= a_re_r * b_re_r;
46     c_re_2 <= a_im_r * b_im_r;
47     c_im_1 <= a_re_r * b_im_r;
48     c_im_2 <= a_im_r * b_re_r;
49   end
50
51   always_ff @(posedge clk) begin
52     c_re_tmp <= c_re_1 - c_re_2;
53     c_im_tmp <= c_im_1 + c_im_2;
54   end
55
56   assign c_re = c_re_tmp[K_OUT2:K_OUT1];
57   assign c_im = c_im_tmp[K_OUT2:K_OUT1];
58 endmodule
```

Testujemy

Do sprawdzenia działania naszego modułu wykorzystamy testbench, którego najciekawsze fragmenty pokazane są na **listingu 2**. Najpierw definiujemy pomocnicze stałe (linie 12…14). K to długość wektorów reprezentujących dane wejściowe, LATENCY to latencja, a TESTS to liczba działań, które chcemy wykonać w ramach testów. Następnie (wiersze 23…26) tworzymy cztery pomocnicze tablice, o długości równej latencji modułu. Wykorzystamy je do przechowywania danych wejściowych i porównania ich z danymi wyjściowymi, otrzymanymi po ich propagacji przez moduł.

Generowanie przypadków testowych znajdziemy w pętli for. Jeden obieg pętli odpowiada jednemu przypadkowi testowemu. Dodatkowe obiegi odpowiadające latencji pozwalają nam poczekać na ostatnie wyniki. Aktualne wymuszenia są generowane w liniach 52…55 i zapisywane w tablicach pod adresem i%LATENCY. Zostaną odczytane przez kod z wierszy 47…50, po upływie LATENCY cyklów, a następnie wykorzystane do wyznaczenia oczekiwanego wyniku i wyświetlone razem z wyjściem z modułu.

Symulację możemy uruchomić, przechodząc w programie ModelSim do folderu *12_mul* i po wywołaniu rozkazu:

```
do c_mul.do
```

Po zakończeniu symulacji w konsoli pojawią się oczekiwane oraz uzyskane wyniki. Tak jak widzimy na **listingu 3**, najpierw pojawia się wynik z symulacji (oczekiwany), a potem z kodu rtl. W naszym przypadku są one zgodne. Uzyskane przebiegi czasowe pokazuje **rysunek 3**. W liniach 3…6 widzimy dane wejściowe. Na trzecim narastającym zboczu zegara, po pojawieniu się przebiegów, na wyjściu widać odpowiadający im wynik.

Listing 2. Moduł testowy dla mnożenia zespolonego (12_mul/c_mul_tv.sv)

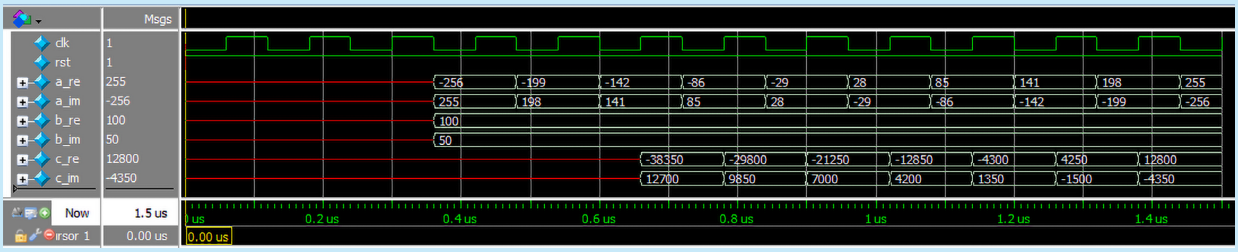
```
11 module c_mul_tb;
12   parameter K = 9;
13   parameter LATENCY = 3;
14   parameter TESTS = 10;

23   int ra_re[LATENCY-1:0];
24   int ra_im[LATENCY-1:0];
25   int rb_re[LATENCY-1:0];
26   int rb_im[LATENCY-1:0];

39   initial begin
40     int rc_re, rc_im;
41     @(posedge clk);
42     @(posedge clk);
43     @(posedge clk);
44     for (int i = 0; i < TESTS+LATENCY; i++) begin
45       @(negedge clk);
46       if (i >= LATENCY) begin
47         rc_re = ra_re[i%LATENCY]*rb_re[i%LATENCY]-ra_im[i%LATENCY]*rb_im[i%LATENCY];
48         rc_im = ra_re[i%LATENCY]*rb_im[i%LATENCY]+ra_im[i%LATENCY]*rb_re[i%LATENCY];
49         $display("sim: %0d+%0j rtl: %0d+%0j",
50           rc_re, rc_im, c_re, c_im);
51       end
52       rb_re[i%LATENCY] = 100;
53       rb_im[i%LATENCY] = 50;
54       ra_re[i%LATENCY] = i*511.0/(TESTS-1)-256;
55       ra_im[i%LATENCY] = 255-i*511.0/(TESTS-1);
56
57       a_re = ra_re[i%LATENCY];
58       a_im = ra_im[i%LATENCY];
59       b_re = rb_re[i%LATENCY];
60       b_im = rb_im[i%LATENCY];
61       @(posedge clk);
62     end
63   $stop;
64 end
```

Listing 3. Wynik symulacji

```
# sim:      -38350+      12700j rtl:  -38350+  12700j
# sim:      -29800+       9850j rtl:  -29800+   9850j
# sim:      -21250+       7000j rtl:  -21250+   7000j
# sim:      -12850+       4200j rtl:  -12850+   4200j
# sim:       -4300+       1350j rtl:   -4300+   1350j
# sim:        4250+      -1500j rtl:    4250+  -1500j
# sim:       12800+      -4350j rtl:   12800+  -4350j
# sim:       21200+      -7150j rtl:   21200+  -7150j
# sim:       29750+     -10000j rtl:   29750+ -10000j
# sim:       38300+     -12850j rtl:   38300+ -12850j
```

Dodajemy UART

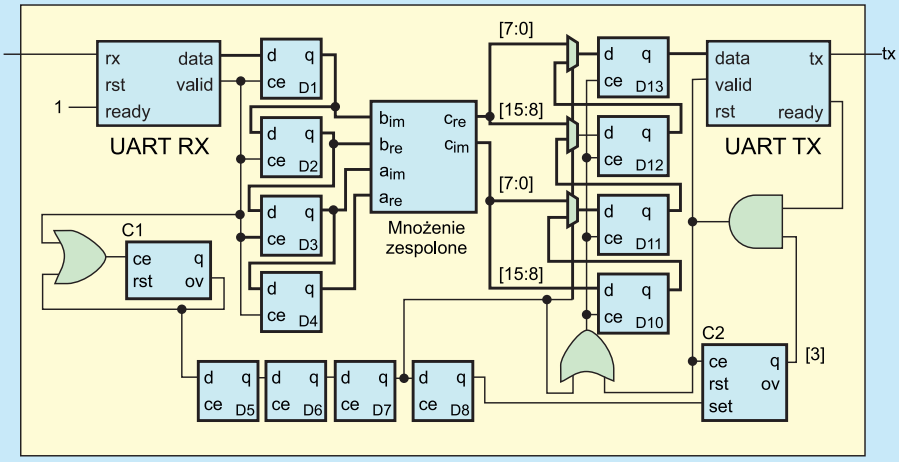
Aby przetestować nasz nowy moduł, w sprzęcie musimy jeszcze dodać moduł, odpowiedzialny za dostarczanie danych wejściowych i odbieranie wyjścia. Do komunikacji wykorzystamy port szeregowy. Schemat przygotowanego modułu pokazuje **rysunek 4**. Aby



Rysunek 3. Przebiegi czasowe uzyskane dla symulacji mnożenia zespolonego

uprościć odbieranie danych, liczby wejściowe są 8-bitowe ze znakiem. Na rysunku, dla uproszczenia, pominięty został sygnał zegarowy – należy pamiętać, że wszystkie przerzutniki są taktowane z jednego zegara. Niektóre bloki mają także wyprowadzone złącze reset. Wszystkie one są połączone razem i resetowane jednym sygnałem.

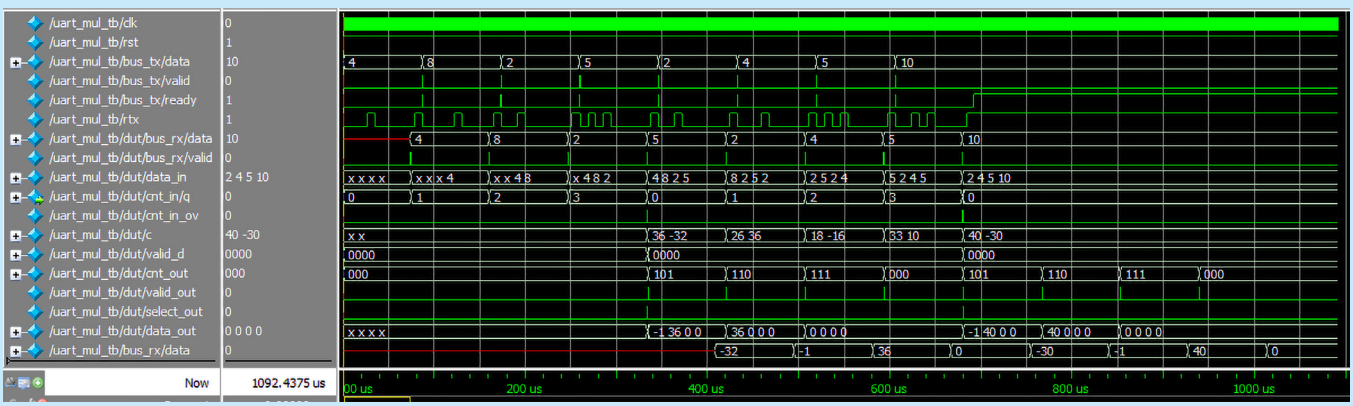
Dane odbierane są przez znany nam już moduł odbiornika. Następnie trafiają na rejestr przesuwny, złożony z przerzutników D1…D4. Przez port po kolei przesyłamy część rzeczywistą i urojoną, najpierw dla czynnika a, później b. Dzięki temu, po wysłaniu wszystkich czterech wartości, znajdą się w rejestrach podłączonych do odpowiednich wejść modułu. Równocześnie dla każdej odebranej danej następuje inkrementacja licznika C1. Zlicza on modulo 5. Kiedy odebrane zostaną wszystkie cztery dane, na jego wyjściu `OV` pojawi się stan 1, który utrzyma się dokładnie przez jeden takt zegara. Będzie się on propagował przez linię opóźniającą, złożoną z przerzutników D6, D7 i D8. Dzięki temu płynie on razem z obliczeniami w module `c_mux`. Dlatego, gdy na wyjściu pojawi się wynik, multipleksery spowodują, że zostanie on przekazany na wejście rejestrów D10…13 i zapisany do nich. Uzyskany wynik ma długość dwa razy po 17 bitów. Aby zmieścić się w 4 bajtach, najstarsze bity nie będą przesyłane. W kolejnym takcie zegara rejestr D9 zostanie ustawiony i wpisze do licznika C2 liczbę 4, czyli 100 (w zapisie binarnym). Rozpocznie to proces wysyłania danych i multipleksery przestawią się w konfiguracji rejestru przesuwnego. Kiedy wszystkie dane zostaną przekazane, licznik C2 „przekręci się”, co spowoduje wyzerowanie bitu `q[3]` i zakończenie transmisji. Najpierw wysłana będzie część rzeczywista, później część urojona wyniku. Kolejność wysyłania to tak zwane littleendian. Oznacza to, że najpierw zostanie wysłanych 8 mniej znaczących, a później 8 bardziej znaczących bitów.



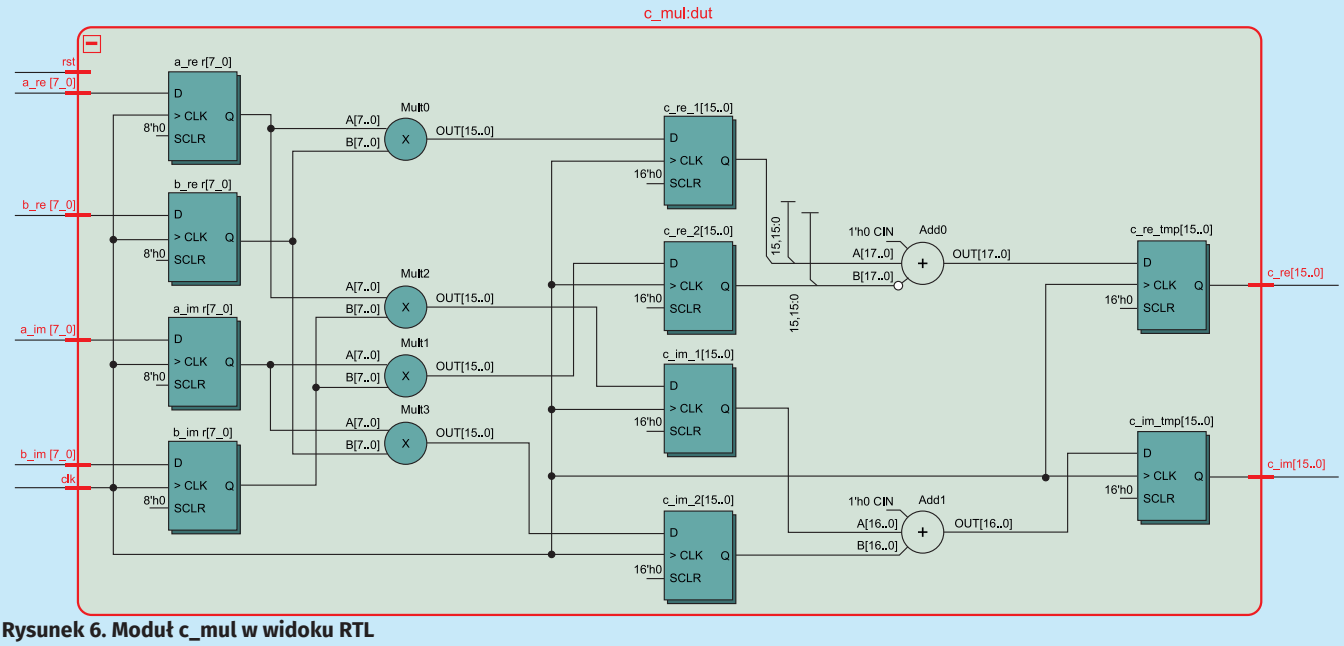
Rysunek 4. Odbieranie i wysyłanie danych przez port szeregowy

Implementacja logiki z rysunku 4 znajduje się na **listingu 4**. W wierszach 10…17 widzimy definicję modułu. Ma on trzy wejścia: zegarowe, reset i rx oraz jedno wyjście tx. Deklaracje wewnętrznych połączeń zostały pominięte – można je znaleźć w całym pliku dostępnym w repozytorium. Dalej w liniach 31…34 znajduje się instancja odbiornika portu szeregowego. Jej wyjście wchodzi na 4-elementowy rejestr przesuwny (wiersze 36…42). Na rysunku 4 składa się on z przerzutników D1…D4. Następnie znajdziemy moduł licznika modulo 5, który zlicza bajty odebrane przez port szeregowy. W liniach 51…59 jest nasz moduł mnożenia liczb zespolonych. Dalej w wierszach 61…65 widzimy rejestr przesuwny, opóźniający sygnał odebrania danych o latencję mnożenia (przerzutniki D6…D9). Kolejny blok `always_ff` (67…73) to wyjściowy rejestr przesuwny z zewnętrznym ładowaniem. Multipleksery zostały zrealizowane za pomocą operatora `?:`. W liniach 81…87 znajduje się 3-bitowy licznik danych wyjściowych. Tym razem nie wykorzystaliśmy naszego starego modułu `counter`, ponieważ potrzebne nam było dodatkowe wejście, wpisujące do licznika liczbę 4. Na samym końcu znajdziemy instancję nadajnika portu szeregowego.

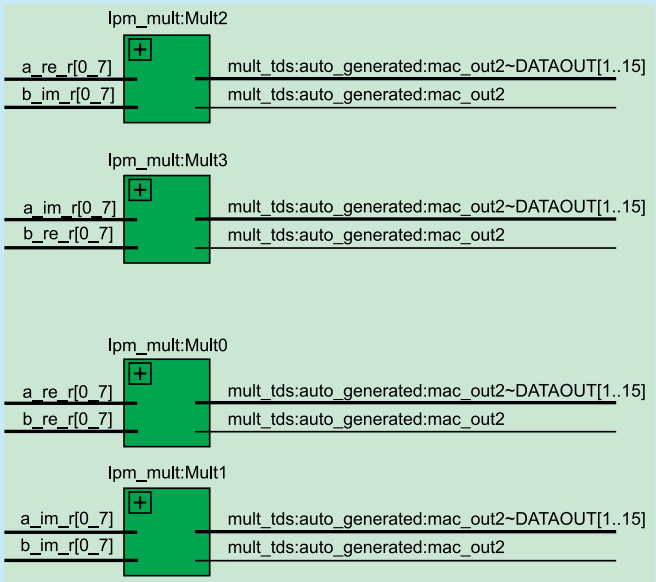
Fragmenty kodu służącego do symulacji znajdziemy na **listingu 5**. W wierszu 15 widzimy listę bajtów, które zostaną wysłane. Odpowiadają one dwóm przypadkom testowym. Pierwszy z nich to mnożenie (4+8j)(2+5j), którego oczekiwanym wynikiem jest –32+36j. Drugi to (2+4j)(5+10j)=–30+40j. Do generowania sygnału testowego



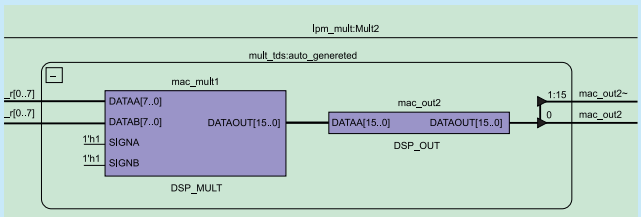
Rysunek 5. Symulacja przedstawiająca komunikację przez port szeregowy



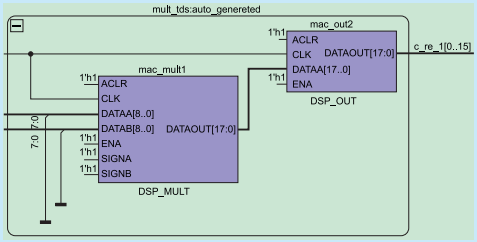
Rysunek 6. Moduł c_mul w widoku RTL



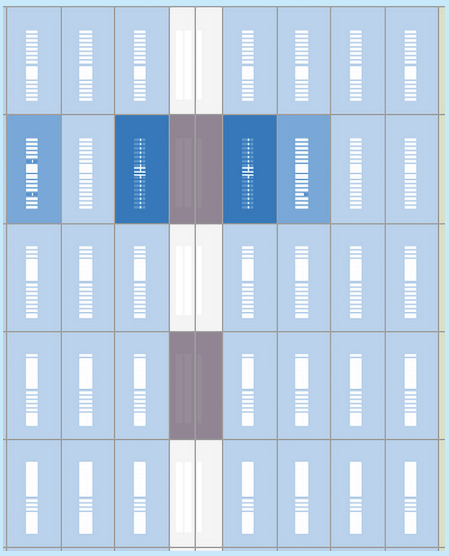
Rysunek 7. Moduł c_mul w widoku Technology Map Viewer (Post--Mapping)



Rysunek 8. Reprezentacja modułu mnożenia w widoku Technology Map Viewer (Post-Mapping)



Rysunek 9. Reprezentacja modułu mnożenia w widoku Technology Map Viewer (Post-Fitting)



Rysunek 10. Lokalizacja bloków DSP w widoku Chip Planner

oraz odbierania wyniku wykorzystane są dodatkowe instancje nadajnika i odbiornika portu szeregowego. Samo wysyłanie odbywa się w pętli przedstawionej w wierszach 31…40. Po przesłaniu wszystkich danych odczekujemy jeszcze 400 µs. Po tym czasie symulacja zostanie zakończona.

Symulację uruchamiamy poleceniem:

```
do uart_mul.do
```

Uzyskany wynik pokazuje **rysunek 5**. W trzecim wierszu widzimy wchodzące dane. Kolejna linia to sygnał `valid` wyzwalający ich przesyłanie. Samą komunikację możemy zaobserwować w wierszu szóstym. Linia siódma to kolejne bajty odbierane przez port szeregowy. Wiersz dziewiąty (data_in) pokazuje aktualny stan wejściowego rejestru przesuwnego. Kolejny sygnał (q) to licznik wejściowy. Ciekawe informacje możemy także wyczytać z linii 12 – jest to wyjście z mnożenia. Widzimy, że nowe wyniki pojawiają się co takt zegara i bazują na kolejnych wartościach z wejściowego rejestru przesuwnego. Jednak zatrzaśnięte i wysłane z powrotem będą tylko wartości uzyskane dla poprawnego stanu wejść.

Zwracane dane widzimy w ostatnim wierszu. Dla pierwszego mnożenia otrzymujemy –32, –1, 36, 0. Gdy zapiszemy te liczby w bitach, otrzymamy: 1110 0000, 1111 1111, 0010 1000, 0000 0000. Ponieważ najpierw przesyłamy mniej znaczący bajt, po złożeniu otrzymamy

Poprzednie części kursu i dodatkowe materiały dostępne są na stronie www.media.avt.pl

Listing 4. Implementacja komunikacji przez port szeregowy (12_mul/uart_mul.sv)

```
module uart_mul #(
  parameter F = 8000000
) (
  input wire clk,
  input wire rst,
  input wire rx,
  output logic tx
);

  assign bus_rx.ready = 1'b1;
  uart_rx #(.F(F), .BAUD(BAUD)) urx (
    .rx(rx),
    .bus(bus_rx));

  always_ff @(posedge clk)
    if (bus_rx.valid) begin
      data_in[0] <= bus_rx.data;
      data_in[1] <= data_in[0];
      data_in[2] <= data_in[1];
      data_in[3] <= data_in[2];
    end

  counter #(.N(5)) cnt_in (
    .clk(clk),
    .rst(rst),
    .ce(bus_rx.valid | cnt_in_ov),
    .q(),
    .ov(cnt_in_ov));

  c_mul #(.K(8), .K_OUT1(0), .K_OUT2(15)) dut (
    .clk(clk),
    .rst(rst),
    .a_re(data_in[3]),
    .a_im(data_in[2]),
    .b_re(data_in[1]),
    .b_im(data_in[0]),
    .c_re(c[0]),
    .c_im(c[1]));

  always_ff @(posedge clk or negedge bus_rx.rst)
    if (!bus_rx.rst)
      valid_d <= '0;
    else
      valid_d <= {valid_d[2:0], cnt_in_ov};

  always_ff @(posedge clk)
    if (select_out | valid_out) begin
      data_out[0] <= c[1][15:8];
      data_out[1] <= select_out ? c[1][7:0] : data_out[0];
      data_out[2] <= select_out ? c[0][15:8] : data_out[1];
      data_out[3] <= select_out ? c[0][7:0] : data_out[2];
    end

  assign bus_tx.valid = valid_out;
  assign bus_tx.data = data_out[3];

  assign valid_out = cnt_out[2] & bus_tx.ready;
  assign select_out = valid_d[2];

  always_ff @(posedge clk or negedge bus_rx.rst)
    if (!bus_rx.rst)
      cnt_out <= 3'b000;
    else if (valid_d[3])
      cnt_out <= 3'b100;
    else if (valid_out)
      cnt_out <= cnt_out + 3'b1;

  uart_tx #(.F(F), .BAUD(115200)) utx (
    .bus(bus_tx),
    .tx(tx));
```

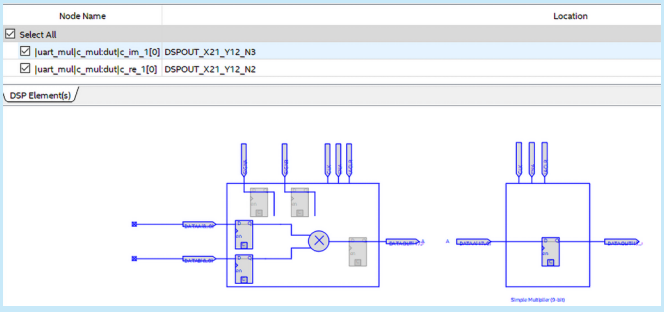
liczby: 1111 1111 1110 0000 i 0000 0000 0010 1000. Po zamianie na system dziesiętny, zgodnie z oczekiwaniami otrzymamy –32+36j. Wynik drugiego mnożenia możemy odczytać analogicznie.

Sprzęt

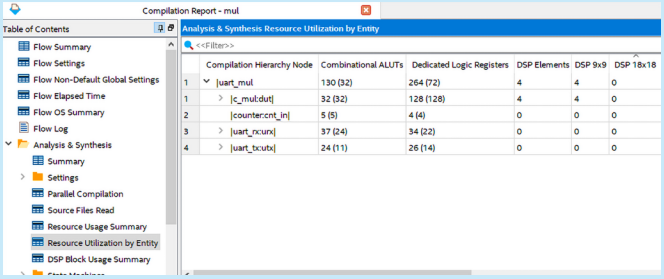
Mamy już gotowy projekt. Ostatnim etapem będzie budowa w środowisku Quartus i testy w sprzęcie. Przy okazji zobaczymy, w jaki sposób zostanie zinterpretowane mnożenie. Otwieramy więc projekt `12_mul/mul.qpf` i rozpoczynamy jego budowę. Kiedy się zakończy, możemy w panelu Tasks wybrać opcję: Compile Design/Analysis&Synthesis/Netlist Viewers/RTL Viewer. Jeśli znajdziemy i otworzymy w nim realizację modułu c_mul, zobaczymy wynik podobny do tego z **rysunku 6.** Jest on bardzo podobny do naszego początkowego projektu z rysunku 2.

Listing 5. Testbench sprawdzający działanie komunikacji przez port szeregowy (12_mul/uart_mul_tb.sv)

```
  logic [7:0]data[7:0] = {8'd10, 8'd5, 8'd4, 8'd2, 8'd5, 8'd2, 8'd8, 8'd4};

  initial begin
    bus_tx.valid = 1'b0;
    #5000;

    for (int i = 0; i < $size(data); i++) begin
      @(negedge clk);
      bus_tx.data = data[i];
      bus_tx.valid = 1'b1;
      @(posedge clk);
      @(posedge clk);
      bus_tx.valid = 1'b0;
      while (!bus_tx.ready)
        @(posedge clk);
    end

    #400us;
    $stop;
  end
```



Rysunek 11. Blok mnożenia w widoku Resource Property



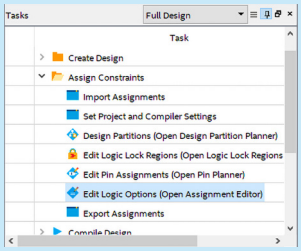
Rysunek 12. Lista wykorzystanych zasobów

Jeżeli przejdziemy do widoku Technology Map Viewer (Post-Mapping), zaobserwujemy (**rysunek 7**), że tym razem znak mnożenia został zastąpiony nowym modułem. Kiedy go otworzymy, naciskając symbol (+), zobaczymy widok podobny do tego z **rysunku 8**. Widzimy na nim bloki DSP_MULT oraz DSP_OUT. Odpowiadają one sprzętowemu blokowi, który znajduje się wewnątrz układu FPGA. Poza wejściami danych DATAA i DATAB mamy także konfigurację znaku. W naszym przypadku oba z nich są na stałe ustawione na 1. Środowisko przyporządkowało tę stałą na podstawie parametru `signed`, w definicji wejściowych sygnałów.

Jeżeli przejdziemy do widoku Technology Map Viewer (Post-Fitting), zobaczymy schemat podobny do **rysunku 9**. Widzimy, że główną różnicą są dodatkowe sygnały kontrolne enable oraz reset. Jak można zobaczyć w narzędziu Chip Planner, moduły mnożenia rozmieszczone są w układzie FPGA w jednej kolumnie. Jest ona oznaczona kolorem jasnoszarym. Wykorzystane moduły zostały zaznaczone kolorem ciemniejszym. Jak widzimy, nasz projekt zużył 2 bloki. Dokładniejszy opis bloku znajdziemy w programie Resource Property Editor. Fragment okna tego programu prezentuje **rysunek 11**. Na liście u góry rysunku, w pojedynczym module (oznaczonym jako X21_Y12), umieszczone zostały dwa 9-bitowe mnożenia.

Warto jeszcze zobaczyć, jakie zasoby zużywa nasz moduł. W tym celu otwieramy raport kompilacji. Jak widzimy na **rysunku 12,** z zakładki Analysis & Synthesis wybieramy Resource Utilization by Entity. Na liście w centralnej części okna rozwijamy uart_mul. Zobaczymy, że blok c_mul zużył 32 bloki ALUT, 128 rejestrów

i 4 elementy DSP (czyli mnożarki). Możemy zmienić konfigurację i wymusić, aby mnożenie zostało zrealizowane z elementów kombinacyjnych, a nie w odpowiednich blokach. W tym celu z panelu Tasks otwieramy Edit Logic Options. Pojawi się nowe okno podobne do tego z **rysunku 14**. Widzimy, że w tym miejscu także pojawia się konfiguracja wejść/wyjść, którą stworzyliśmy w narzędziu Pin Planner. Aby dodać nową pozycję, klikamy <<new>> w kolumnie Assignment Name. Z listy wybieramy opcję DSP_BLOCK_BALANCING. W kolumnie Value, z dostępnych na liście możliwości, wybieramy Logic Elements. Cała konfiguracja jest pokazana na **rysunku 15**. Możemy zapisać i uruchomić budowę powtórnie. **Rysunek 16** pokazuje zużycie zasobów. Widzimy, że obecnie liczba elementów DSP jest równa 0. Liczba zużytych rejestrów nie uległa zmianie. Natomiast liczba elementów logicznych wzrosła ponaddziesięciokrotnie z 32 do 398.



Rysunek 13. Dodawanie ograniczeń

Przywrócimy wykorzystanie sprzętowych mnożarek poprzez zmianę wartości w kolumnie Enabled (rysunek 14) na – No. Teraz zbudujmy projekt jeszcze raz i zaprogramujmy płytkę Rysino.

Do komunikacji użyjemy programu RealTerm. Po wyborze numeru prędkości i ustawieniu prędkości na 115200, przejdźmy do zakładki Display. Tutaj wybieramy tryb wyświetlania (Display As) na int16 – liczba 2-bajtowa ze znakiem. Na końcu jeszcze wyłączamy tryb Big Endian, odklikując znacznik.

	Status	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag
1	Ok		clk	Location	PIN_27	Yes			
2	Ok		rst	Location	PIN_121	Yes			
3	Ok		rx	Location	PIN_70	Yes			
4	Ok		tx	Location	PIN_69	Yes			
5		<<new>>	<<new>>	<<new>>					

Rysunek 14. Konfiguracja projektu

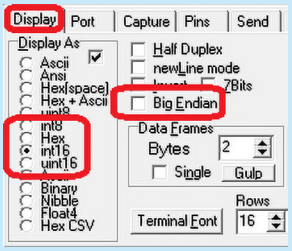
5	Ok			DSP Block Balancing	Logic Elements	Yes	c_mul

Rysunek 15. Konfiguracja wymuszająca syntezę mnożenia z elementów logicznych

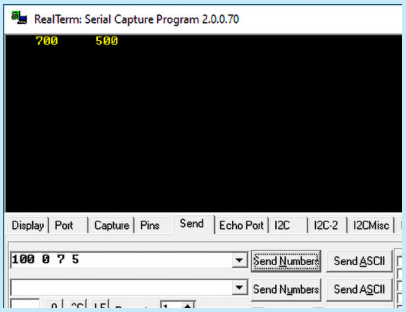
	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	DSP Elements	DSP 9x9	DSP 18x18
1	|uart_mul|	495 (32)	264 (72)	0	0	0
1	|c_mul:dut|	398 (96)	128 (128)	0	0	0
2	|counter:cnt_in|	5 (5)	4 (4)	0	0	0
3	|uart_rx:urx|	36 (23)	34 (22)	0	0	0
4	|uart_tx:utx|	24 (11)	26 (14)	0	0	0

Rysunek 16. Zużycie zasobów przy mnożeniu w logice

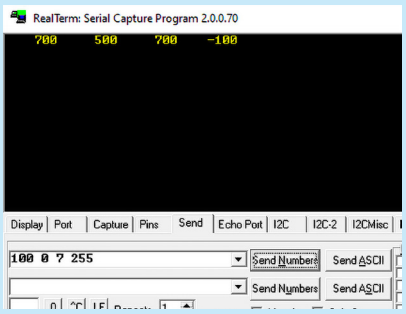
Aby wysłać dwie liczby zespolone, przechodzimy do zakładki Send (**rysunek 18**). W polu wpisujemy cztery liczby. Po kolei są to część rzeczywista i urojona obu czynników. W moim przypadku mnożę liczby 100+0j i 7+5j. Po kliknięciu Send Numbers, w konsoli pojawił się wynik: 700+500j. Niestety, nie jest obsługiwane przesyłanie liczb ujemnych, dlatego trzeba samemu „zmienić" liczbę ujemną, zapisaną w kodzie dopełnienia do dwóch, na odpowiadającą jej liczbę dodatnią. Przykładowo, na **rysunku 19** przesłano liczby 100+0j i 7–1j. Przy wyświetlaniu zgodnie z konfiguracją, liczby ujemne są już obsługiwane.



Rysunek 17. Konfiguracja programu RealTerm



Rysunek 18. Wysyłanie danych



Rysunek 19. Przesyłanie liczby ujemnej

Podsumowanie

Poznaliśmy nowy element układu FPGA – mnożarkę, nazywaną także blokiem DSP. Użyliśmy jej do implementacji mnożenia zespolonego. W kolejnym odcinku blok stanie się fragmentem naszej implementacji dyskretnej transformaty Fouriera.

Rafał Kozik
rafkozik@gmail.com

Bibliografia

[1] Repozytorium z przykładami – http://bit.ly/33uYPxs
[2] MAX 10 Embedded Multiplier Block Overview, https://intel.ly/2ZZDuxe
[3] Complex Multiplication, https://bit.ly/2EpkKzo

Miesięcznik **„Elektronika Praktyczna"**
(12 numerów w roku) jest wydawany
przez AVT-Korporacja Sp. z o.o. we współpracy
z wieloma redakcjami zagranicznymi.

Wydawnictwo:
AVT-Korporacja Sp. z o.o.
03-197 Warszawa, ul. Leszczynowa 11
tel. 22 257 84 99, faks 22 257 84 00

Wydawca:
Wiesław Marciniak

Adres redakcji:
03-197 Warszawa, ul. Leszczynowa 11
tel. 22 257 84 60
faks 22 257 84 00
e-mail: redakcja@ep.com.pl
www.ep.com.pl

Redaktor Naczelny:
Damian Sosnowski

Redaktor Programowy,
Przewodniczący Rady Programowej:
Piotr Zbysiński

Zastępca Redaktora Naczelnego,
Menedżer Magazynu:
Katarzyna Gugała

Szef Pracowni Konstrukcyjnej:
Grzegorz Becker

Redakcja strony internetowej www.ep.com.pl
MAD Sp. z o.o.

Zespół marketingu i reklamy:
Katarzyna Gugała, tel. 22 257 84 64
Bożena Krzykawska, tel. 22 257 84 42
Grzegorz Krzykawski, tel. 22 257 84 60

Sekretarz Redakcji:
Grzegorz Krzykawski, tel. 22 257 84 60

DTP i okładka:
MAD Sp. z o.o.

Stali Współpracownicy:
Nikodem Czechowski, Jakub Tyburski, Lucjan Bryndza, Jarosław Doliński, Andrzej Gawryluk, Krzysztof Górski, Tomasz Jabłoński, Michał Kurzela, Szymon Panecki, Sławomir Skrzyński, Ryszard Szymaniak, Adam Tatuś, Robert Wołgajew

Uwaga!
Kontakt z wymienionymi osobami jest możliwy via e-mail, według schematu: imię.nazwisko@ep.com.pl

Prenumerata w Wydawnictwie AVT
www.avt.pl/prenumerata
lub tel. 22 257 84 22
e-mail: prenumerata@avt.pl
www.sklep.avt.pl, tel. 22 257 84 66

Prenumerata w RUCH S.A.
www.prenumerata.ruch.com.pl
lub tel. 801 800 803, 22 717 59 59
e-mail: prenumerata@ruch.com.pl

Wydawnictwo
AVT-Korporacja Sp. z o.o.
należy do **Izby Wydawców Prasy**