Eksperymenty z FPGA (6)

W poprzedniej części uruchomiliśmy część nadawczą portu szeregowego, który znajdziemy na płytce „Rysino". Dzisiaj uzupełnimy go o część odbiorczą. A na końcu zajmiemy się luźniejszym tematem: przygotujemy moduł generujący sygnał o zmiennym wypełnieniu (PWM). Tak jak poprzednio przed przystąpieniem do wykonywania eksperymentów zachęcam do aktualizacji repozytorium z przykładami [1] (na przykład poprzez wywołanie polecenia git pull).

Odbieramy

Odbieranie danych jest procesem nieco trudniejszym od nadawania. Nie mamy pełnej kontroli nad sytuacją, musimy bardziej dostosować się do nadawcy. Przyjrzyjmy się jeszcze raz ramce danych (**rysunek 1**). Dodatkowo znalazły się nad nią strzałki, określające momenty, w których musi zareagować nasz odbiornik.

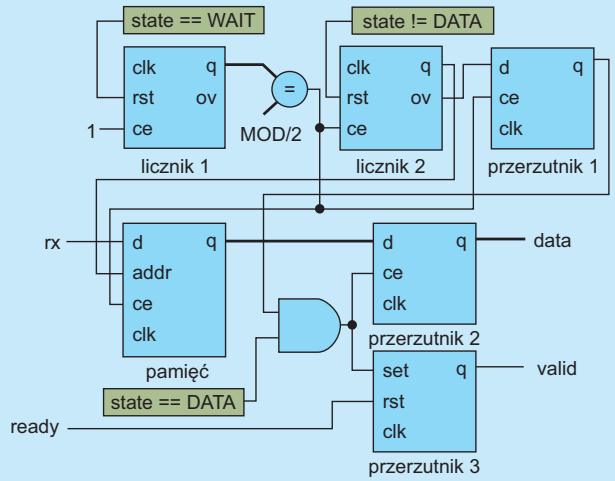
W stanie bezczynności na linii panuje jedynka logiczna. Nasz odbiornik cały czas oczekuje, aż stan zmieni się na zero, co oznacza pojawienie się bitu startu i początek nowej ramki. Wydarzenie to zaznaczone jest czerwoną strzałką. Bit startu pozwala nam zsynchronizować nasz odbiór z nadajnikiem. W tym monecie musimy uruchomić nasz własny zegar, który powie nam, kiedy na linii pojawią się kolejne bity. Chcemy jednak odczytywać stan linii nie w momentach, gdy następuje zmiana, ale gdy jej stan jest stabilny. Dlatego kolejne odczyty będziemy wykonywać nie w momencie gdy licznik się przepełni, ale gdy doliczy do połowy. Pierwszy odczyt (niebieska strzałka) trafi więc w bit startu, a dopiero kolejne osiem (zaznaczone na zielono) pozwoli odebrać dane. Cały cykl odbioru kończymy na bicie stopu, zaznaczonym czarną strzałką. W tym momencie dane zostały odebrane, a odbiornik znowu rozpoczyna nasłuch w celu wykrycia następnego bitu startu.

Podobnie jak w nadajniku, tu także skorzystamy z maszyny stanu. Tym razem wystarczą nam cztery stany: WAIT, START, DATA i STOP. Na **rysunku 2** pokazano graf przejść maszyny stanów. Bezpośrednio wynikają one ze zdarzeń zaznaczonych na **rysunku 1**. Po resecie moduł znajdzie się w stanie WAIT, gdzie oczekuje dopóty, dopóki stan na wejściu *rx* nie zmieni się na niski. Spowoduje to przejście do stanu START, gdzie oczekuje na pierwszy sygnał *rx_clk*, który następuje w połowie bitu startu. Wtedy następuje przejście do stanu DATA, w którym odbiornik znajduje się aż do odebrania 8 bitów transmitowanych w pojedynczej ramce. Następnie poprzez stan STOP wraca do oczekiwania na nową transmisję w stanie WAIT. Jej implementację pokazuje **listing 1**.

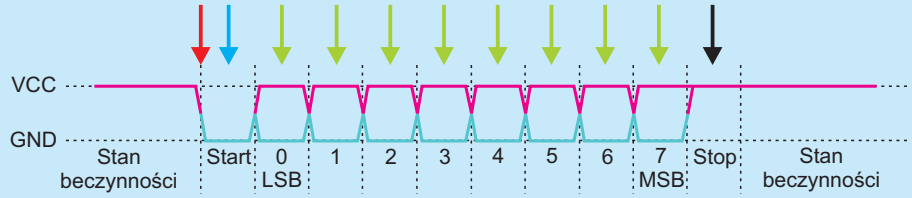
Na **rysunku 3** pokazano poglądowy schemat logiki odbiornika, natomiast **listing 2** prezentuje jej implementację. Jego zachowanie zależy od aktualnego stanu maszyny stanów. Licznik1 służy do odmierzania czasu trwania pojedynczego symbolu. Startuje on, gdy moduł wyjdzie ze stanu WAIT. Jego implementacja znajduje się w liniach 41...48. Tym razem jednak sygnał CE jest aktywny, gdy doliczy do połowy, a nie w momencie przepełnienia. Zostało to zrealizowane za pomocą polecenia *assign* w linii 48. Licznik2 (linie 50...56) zlicza liczbę odebranych bitów. Pracuje, tylko gdy odbiornik jest w stanie DATA. Na podstawie jego wartości wybierana jest pozycja w ośmiobitowej „pamięci", pod którą

Listing 1. Implementacja maszyny stanów (06_UART/uart_rx.sv)

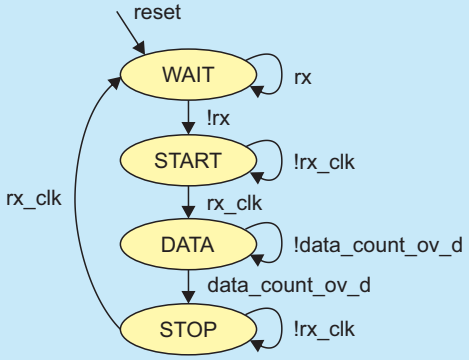
```
29 always_ff @(posedge bus.clk or negedge bus.rst)
30  if (!bus.rst)
31      s <= WAIT;
32  else
33  case (s)
34      WAIT: s <= rx ? WAIT : START;
35      START: s <= rx_clk ? DATA : START;
36      DATA: s <= data_count_ov_d ? STOP : DATA;
37      STOP: s <= rx_clk ? WAIT : STOP;
38      default: s <= WAIT;
39  endcase
```



Rysunek 3. Schemat ścieżki danych



Rysunek 1. Ramka danych w porcie szeregowym, gdzie strzałki określają momenty, w których reaguje nasz odbiornik



Rysunek 2. Przejścia pomiędzy stanami w odbiorniku

Listing 2. Implementacja interfejsu w języku SystemVerilog (06_UART/uart_rx.sv)

```
41 assign ctx_rst = (s == WAIT);
43 counter #(.N(MOD)) crx (
43  .clk(bus.clk),
44  .rst(!ctx_rst),
45  .ce(1'b1),
46  .q(ctx_q),
47  .ov());
48 assign rx_clk = (ctx_q == MOD/2);
49
50 assign data_count_rst = (s != uartPkg::DATA);
51 counter #(.N(8)) data_count (
52  .clk(bus.clk),
53  .rst(!data_count_rst),
54  .ce(rx_clk),
55  .q(i),
56  .ov(data_count_ov));
57
58 always_ff @(posedge bus.clk)
59  if (rx_clk)
60      data_count_ov_d <= data_count_ov;
61
62 always_ff @(posedge bus.clk)
63  if (rx_clk)
64      rxb[i] <= rx;
65
66 always_ff @(posedge bus.clk or negedge bus.rst)
67  if (!bus.rst)
68      bus.valid <= 1'b0;
69  else if (data_count_ov_d && s == DATA)
70      bus.valid <= 1'b1;
71  else if (bus.ready)
72      bus.valid <= 1'b0;
73
74 always_ff @(posedge bus.clk)
75  if (data_count_ov_d && s == DATA)
76      bus.data <= rxb;
```

Listing 3. Testbench umożliwiający sprawdzenie pracy odbiornika (06_UART/uart_rx_tb.sv)

```
28 initial begin
29  bus_rx.ready <= 1'b1;
30  bus_tx.valid <= 1'b1;
31  bus_tx.data <= 8'hab;
32 end
33
34 uart_tx uart_tx(
35  .bus(bus_tx),
36  .tx(rtx));
37
38  uart_rx dut(
39  .rx(rtx),
40  .bus(bus_rx));
```

zostanie zapisana kolejna wartość z wejścia *rx*. Jej implementacja zajmuje trzy linie 62...64. Na końcu Przerzutnik2 (linie 74...76) zatrzaskuje odebraną wartość, dzięki czemu będzie ona dostępna cały czas, aż do odebrania kolejnej. Przerzutnik3 (linie 66...72) obsługuje naszą magistralę. Gdy pojawi się nowa wartość, sygnał *valid* zostanie ustawiony na 1, w którym zostanie, aż do pojawienia się sygnału na wejściu *ready*. W kodzie, za pomocą kolejności warunków w wyrażeniu if/else, zadbaliśmy o to, żeby ustawienie miało wyższy priorytet niż kasowanie. Wtedy nawet jeżeli *ready* będzie cały czas aktywne, *valid* i tak „zapaliło" się na jeden cykl zegara. Natomiast w żaden sposób nie sprawdzamy, czy dana została przez kogoś odczytana. Jeżeli pojawi się kolejna, stara wartość zostanie po prostu nadpisana.

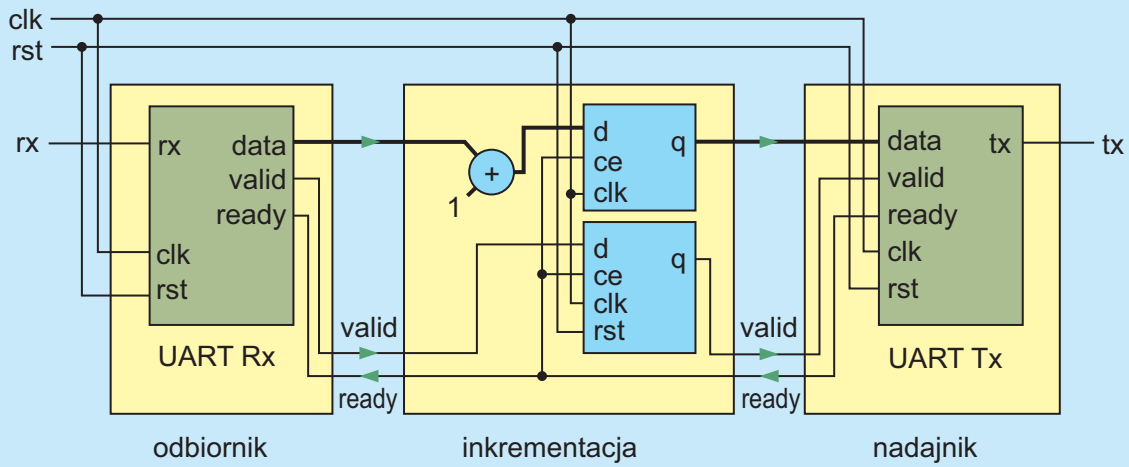
Aby sprawdzić działanie modułu, zastosujemy testbench, którego fragment pokazuje **listing 3**. Do wygenerowania sygnału testowego użyty został nadajnik opisany w poprzedniej części. Jego wyjście *tx* zostało podłączone do wejścia *rx* odbiornika. Aby ją uruchomić, w programie ModelSim wywołujemy polecenie:

```
do uart_rx_sim.do
```

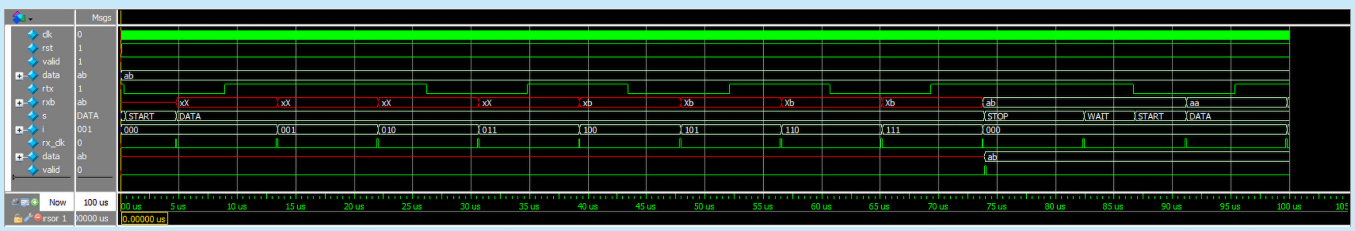
Uzyskany wynik pokazano na **rysunku 4**. Na samej górze widzimy sygnał zegarowy *clk*, lecz jest on na tyle szybki, że w ustawionym przybliżeniu jego zmiany zlewają się ze sobą i widzimy„ciągły prostokąt". Pod nim widzimy sygnał resetu. Niżej widzimy fragment magistrali wejściowej nadajnika. Jako dana wpisana jest liczba 0xAB, natomiast sygnał *valid* jest ciągle ustawiony na 1. Powoduje to, że nadajnik gdy tylko skończy jedną transakcję od razu rozpocznie następną.

Sygnał wejściowy odbiornika pokazany jest w linii r*tx*. Najciekawszy jest sygnał *rxb*, w którym widzimy kolejne odebrane bity. Na początku jego stan jest nieustalony, co symbolizuje kolor czerwony oraz znaki X. Przy okazji warto zobaczyć, że raz mamy mały znak x, a raz duży X. x oznacza, że wszystkie 4 bity, które opisuje, są nieustalone, a X – że część z nich ma już przypisaną wartość. Na samym końcu widzimy, że poprawnie została odebrana wartość 0xAB.

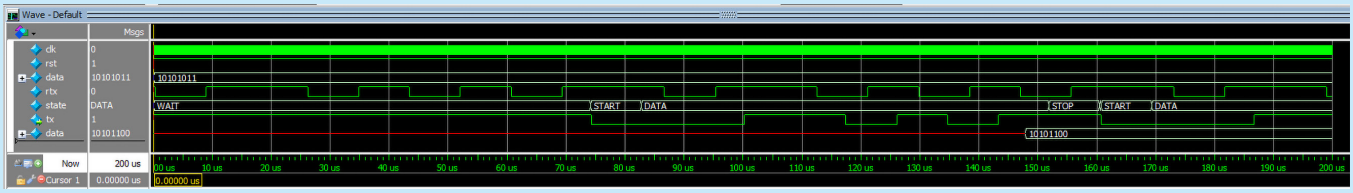
Trzy kolejne sygnały to informacje o wnętrzu odbiornika: s to aktualny stan odbiornika, i – numer odbieranego bitu, a *rx*_clk



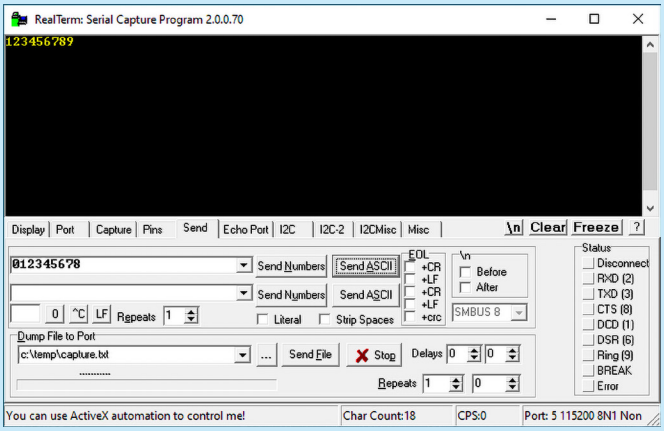
Rysunek 5 Schemat modułu, który odbiera liczbę z portu szeregowego i odsyła liczbę o jeden większą



Rysunek 4. Wyniki symulacji odbiornika



Rysunek 6. Symulacja modułu z listingu 4



Rysunek 7. Okno programu RealTerm przedstawiające wynik działania modułu

– zegara odbiornika liczący czas trwania symbolu. Na samym końcu widzimy magistralę wyjściową: jej sygnał data i *valid*.

Echo

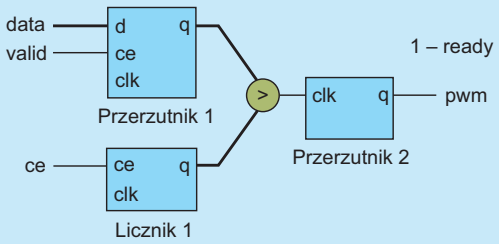
Aby przetestować działanie całości, uruchomimy układ działający podobnie jak „echo", ale żeby sprawdzić, czy poprawnie przetwarza dane pomiędzy odbiornikiem a nadajnikiem dodamy blok zwiększający odebraną wartość o jeden. Układ pokazano na **rysunku 5**.

Pierwszym elementem jest odbiornik. Jego wyjście przepływa przez blok realizujący inkrementację. Warto zwrócić uwagę, że sygnał valid ma taką samą latencję jak data. Ostatnim elementem jest opracowany w poprzedniej części nadajnik. Jego implementację znajdziemy na **listingu 4**. Nasz główny model nosi nazwę uart_inc. Ma on trzy wejścia: zegar clk, reset rst i rx oraz jedno wyjście: tx. W liniach 18 i 19 tworzymy dwie magistrale. Pierwsza bus_rx posłuży do połączenia odbiornika z modułem inkrementującym, a druga bus_tx posłuży do przekazania uzyskanego wyniku do nadajnika. Następnie znajdują się instancje wykorzystanych pod modułów: odbiornik (linie 21...23), inkrementacji (25...27) i nadajnik (29...31).

Moduł można przetestować za pomocą testbenchu, który znajduje się w pliku *06_UART/uart_inc_tb.sv*. Aby wygenerować

Listing. 4. Moduł odbierający, inkrementujący i odsyłający liczbę po porcie szeregowym (06_UART/uart_inc.sv)

```
10 module uart_inc #(
11  parameter F = 8000000
12 ) (
13  input wire clk,
14  input wire rst,
15  input wire rx,
16  output logic tx
17 );
18 StreamBus #(.N(8)) bus_rx (.clk(clk), .rst(rst));
19 StreamBus #(.N(8)) bus_tx (.clk(clk), .rst(rst));
20
21 uart_rx #(.F(F), .BAUD(115200)) urx (
22  .rx(rx),
23  .bus(bus_rx));
24
25 inc increment (
26  .bus_in(bus_rx),
27  .bus_out(bus_tx));
28
29 uart_tx #(.F(F), .BAUD(115200)) utx (
30  .bus(bus_tx),
31  .tx(tx));
32
33 endmodule
```



Rysunek 8. Schemat generatora PWM

i sprawdzić sygnały testowe zostały w nim dodane dodatkowe moduły Rx i Tx. Można go uruchomić za pomocą polecenia:

```
do uart_inc_sim.do
```

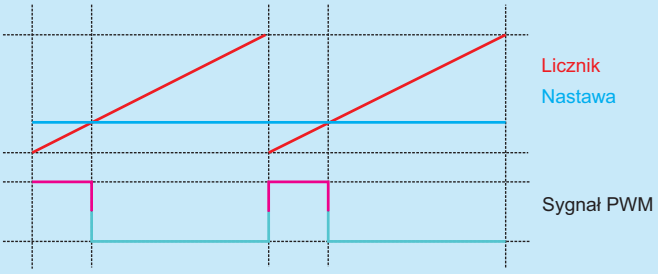
Jej wynik możemy zobaczyć na **rysunku 6**. W trzeciej linii widzimy dane wejściowe. W zapisie binarnym jest to liczba 10101011, co dziesiętnie daje liczbę 171. W ostatniej linii widzimy dane, które wychodzą z bloku uart_inc. Uzyskana wartość to 10101100 binarnie, czyli 172. Oznacza to, że wynik symulacji jest poprawny.

Teraz możemy przejść do ostatniej części, czyli uruchomienia projektu w sprzęcie. W tym celu przygotowany został projekt *06_UART/06_uart_inc.qpf*. Możemy go otworzyć, zbudować i zaprogramować płytkę. Do przetestowania użyjemy znanego nam już programu RealTerm. Najpierw w zakładce *Port* wybieramy numer portu i ustawiamy prędkość na 115200. Następnie przechodzimy do zakładki *Send*, co zostało zaprezentowane na **rysunku 7**. W pole tekstowe możemy wpisać kilka znaków ASCII (u mnie 012345678). Po kliknięciu przycisku *Send ASCII* zostaną one wysłane przez port szeregowy. W konsoli zobaczymy wyniki, czyli ciąg znaków o kodach o 1 większych. W przedstawionym przypadku jest to 123456789.

Sygnał PWM

Kolejnym eksperymentem jest generator sygnału PWM. Skrót ten pochodzi od *pulse-width modulation*, czyli modulacja szerokości impulsu. Jest to sygnał prostokątny o zadanym wypełnieniu.

Uproszczony schemat pokazano na **rysunku 8**. W przerzutniku 1 zapamiętane jest zadane wypełnienie. Jest ono zatrzaskiwane w momencie, gdy sygnał valid jest ustawiony. Zapis jest zawsze możliwy, dlatego sygnał ready został na stałe ustawiony na 1. Licznik 1 w kółko zlicza od 0 do zadanej wartości maksymalnej. Sygnał CE pozwala na regulację częstotliwości generowanego przebiegu. Sygnał wyjściowy jest obliczany poprzez porównanie zadanego wypełnienia z obecną wartością licznika. Dopóki wartość licznika jest mniejsza niż wartość zadana, dopóty na wyjściu



Rysunek 9. Wartość zadana wypełnienia (nastawa), stan licznika oraz wyjściowy sygnał PWM

Listing. 5. Implementacja generatora sygnału PWM (07_PWM/pwm.sv)

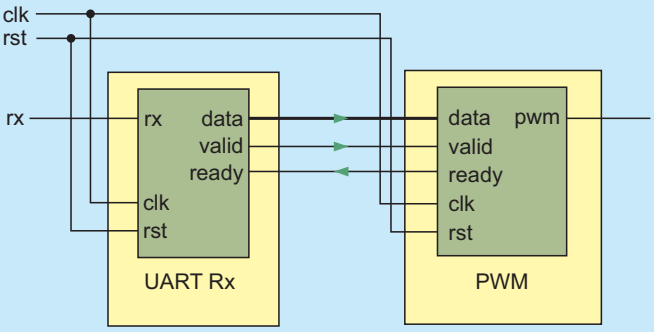
```
11 module pwm #(
12  parameter MAX = 255,
13  parameter MAX_LOG = $clog2(MAX)
14 ) (
15  StreamBus.Slave bus,
16  input wire ce,
17  output logic pwm
18 );
19  logic [MAX_LOG-1:0] count;
20  logic [MAX_LOG-1:0] value;
21
22  assign bus.ready = 1'b1;
23
24  counter #(.N(MAX)) ctx (
25      .clk(bus.clk),
26      .rst(bus.rst),
27      .ce(ce),
28      .q(count),
29      .ov());
30
31 always_ff @(posedge bus.clk or negedge bus.rst)
32  if (!bus.rst)
33      value <= '0;
34  else if (bus.valid)
35      value <= bus.data;
36
37 always_ff @(posedge bus.clk)
38  pwm <= (count < value);
39
40 endmodule
```

Listing 6. Fragmenty testbench dla modułu pwm (07_PWM/pwm_tb.sv)

```
26 initial begin
27  bus.valid <= 1'b1;
28  bus.data <= 4'd0;
29  for (int i = 1; i <= 10; i++)
30      #5us bus.data <= i;
31  #5us $stop;
32 end
33
34 pwm #(.MAX(10)) dut (
35  .bus(bus),
36  .ce(1'b1),
37  .pwm());
```

Listing 7. Generator sygnału PWM sterowany przez UART (07_PWM/pwm_uart.sv)

```
10 module pwm_uart #(
11 parameter F = 8000000
12 ) (
13  input wire clk,
14  input wire rst,
15  input wire rx,
16  output logic [7:0]led
17 );
18  StreamBus #(.N(8)) bus (.clk(clk), .rst(rst));
19  logic pwm_out;
20
21  uart_rx #(.F(F), .BAUD(115200)) urx (
22  .rx(rx),
23  .bus(bus));
24
25  pwm #(.MAX(255)) dut(
26  .bus(bus),
27  .ce(1'b1),
28  .pwm(pwm_out));
29
30 genvar i;
31 generate
32  for (i = 0; i < $bits(led); i++) begin : LED_PWM
33      always_ff @(posedge bus.clk)
34      led[i] <= pwm_out;
35  end
36  endgenerate
37
38 endmodule
```



Rysunek 11. Połączenie odbiornika UART z modułem PWM

panuje stan wysoki. Natomiast od momentu, gdy się zrównają, aż do przepełnienia licznika wyjście znajduje się w stanie niskim.

Sytuacja została pokazana na przebiegu czasowym z **rysunku 9**. Górny wykres pokazuje zadane wypełnienie (kolor niebieski) oraz zmieniający się stan licznika (czerwony). Dolny wykres pokazuje sygnał wyjściowy. Widzimy, że gdy stan licznika jest niższy niż wartość zadana, na wyjściu panuje stan wysoki. W chwili przecięcia następuje zmiana i na wyjściu mamy stan niski. Wypełnienie jest więc równe stosunkowi nastawy do maksymalnej wartości, która może zostać osiągnięta przez licznik.

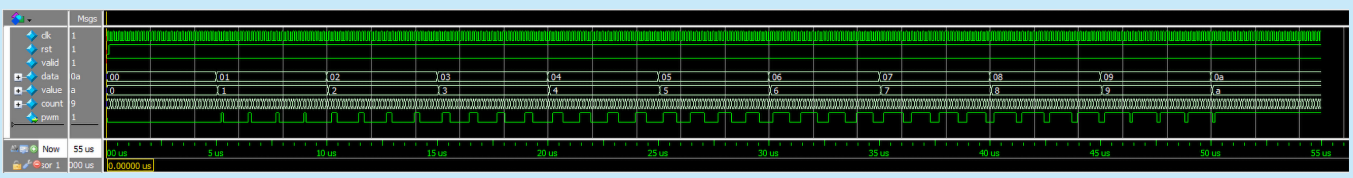
Przejdźmy teraz do implementacji, którą znajdziemy na **listingu 5** (oraz w pliku *07_PWM/pwm.sv*). Parametr MAX określa wartość, do której będzie odliczał licznik, więc równocześnie także wartość nastawy oznaczającej wypełnienie 100%. Sam moduł wystawia znany nam interface StreamBus dzięki czemu będziemy mogli go podłączyć bezpośrednio do wyjścia odbiornika portu szeregowego. Poza tym dostępne jest jeszcze wejście ce oraz wyjście pwm z generowanym przebiegiem. W liniach 24...29 znajduje się instancja licznika. Natomiast w liniach od 31...35 znajduje się przerzutnik, w którym zatrzaskiwana jest nastawa generatora. Samo tworzenie przebiegu ma miejsce w liniach 37 i 38.

Fragment testbench dla modułu znajduje się na **listingu 6** (a całość w pliku *07_PWM/pwm_tb.sv*). W liniach od 34...37 znajduje się instancja generatora PWM. Wartość MAX została ustalona na 10, co oznacza, że wpisanie 0 równa się 0%, a wpisanie 10 odpowiada wypełnieniu 100%. W liniach 26...32 następuje generowanie wymuszenia. Sygnałowi valid na stałe została przypisana wartość 1. Nie jest to problemem, ponieważ na wejściu data cały czas utrzymywana jest zadana wartość. Na początku jest ona równa zero, a następnie co pięć mikrosekund jest inkrementowana. Gdy osiągnie maksymalną wartość, czyli 10, nastąpi odczekanie kolejnych pięciu mikrosekund i symulacja zostanie zakończona.

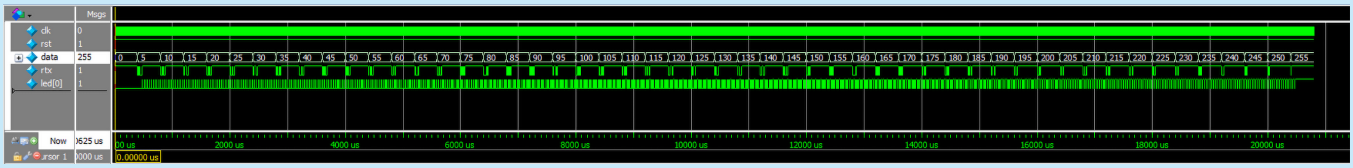
Aby zobaczyć wynik symulacji w programie ModelSim, uruchamiamy skrypt poleceniem:

```
do ./pwm_sim.do
```

Wyniki symulacji są pokazane na **rysunku 10**. W pierwszych trzech liniach widzimy sygnały zegara (clk), resetu (rst) oraz valid. W dwóch kolejnych wierszach mamy wartość data z magistrali oraz opóźnioną o jeden takt zatrzaśniętą wartość zadaną value. Przedostatni wiersz zawiera aktualny stan licznika (count), jednak na załączonym powiększeniu nie da się go odczytać (zachęcam jednak do uruchomienia symulacji i przyglądnięcia mu się na ekranie komputera). Na samym końcu widzimy wyjściowy



Rysunek 10. Wyniki symulacji modułu PWM



Rysunek 12. Wynik symulacji generatora PWM sterowanego poprzez port szeregowy

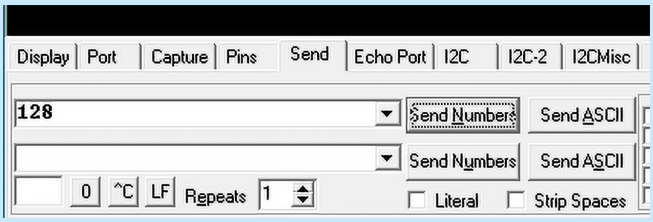
sygnał pwm. Jak możemy zauważyć, że na samym początku dla wypełnienia 0 jest on ciągle w stanie niskim, następnie wypełnienie rośnie, aby dla wartości 10 być ciągle w stanie wysokim.

Aby przetestować działanie modułu PWM, połączymy go z odbiornikiem portu szeregowego zgodnie z **rysunkiem 11**. Dzięki temu będziemy mogli za pomocą portu szeregowego sterować jasnością diod LED. Implementacja tego projektu znajduje się na **listingu 7**. Nasz główny moduł nosi nazwę pwm_uart. W liniach 21...23 znajduje się instancja odbiornika, a dalej w wierszach 25...28 moduł generatora sygnału PWM. Maksymalne wypełnienie jest reprezentowane jako 255. Na końcu w bloku generate zostało stworzone osiem przerzutników, z których każdy będzie sterował jedną z diod świecących. Warto zwrócić uwagę, że obecna tu instrukcja for nie opisuje typowej pętli. Intuicyjnie możemy ją potraktować jako ośmiokrotne wklejenie jej zawartości. Kod ten nie opisuje kolejnych iteracji, ale 8 niezależnych, działających równolegle przerzutników D.

Działanie projektu możemy sprawdzić w symulacji. W programie ModelSim uruchamiamy skrypt:

```
do ./pwm_uart_sim.do
```

Wyniki powinny być podobne do tych z **rysunku 12**. W 3 wierszu widzimy kolejne wartości wysyłane przez port szeregowy. Na początku zaczynamy od 0, a następnie zwiększając co 5, dochodzimy do maksimum, czyli 255. W czwartym rzędzie widoczna jest linia portu szeregowego, a w piątym sygnał sterujący jedną z diod LED. Rozpoczyna się on od ciągłego stanu niskiego. Widzimy, że ulega zmianom po kolejnych zakończonych transmisjach. Na końcu, po odebraniu wartości 255, otrzymujemy ciągły stan wysoki.



Rysunek 13. Okno programu RealTerm z widocznym przyciskiem *Send Numbers*

Ostatnim krokiem jest zbudowanie i uruchomienie projektu w sprzęcie. W tym celu otwieramy projekt *07_PWM/07_pwm.qpf*. Po kompilacji i zaprogramowaniu płytki wracamy do programu RealTerm. Tym razem do przesyłania danych użyjemy przycisku *Send Numbers* (wyślij liczby) pokazany na **rysunku 13**. Dzięki temu wpisany tekst zostanie zinterpretowany jako liczba dziesiętna, a nie ciąg znaków ASCII. Gdy wyślemy 0, diody zgasną, a gdy 255, będą świecić z pełną jasnością.

Podsumowanie

W tym odcinku uruchomiliśmy odbiornik portu szeregowego, dzięki czemu dysponujemy już pełną obsługą tego interfejsu. Następnie przygotowaliśmy prosty generator sygnału PWM i przetestowaliśmy jego działanie. W kolejnej części użyjemy go do zbudowania generatora przebiegu sinusoidalnego, za pomocą którego będziemy sterować membraną piezoelektryczną.

Rafał Kozik
rafkozik@gmail.com

[1] Repozytorium z przykładami ***http://bit.ly/33uYPxs***

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

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

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:
Wiesław Marciniak

Redaktor Programowy, Przewodniczący Rady Programowej:
Piotr Zbysiński

Zastępca Redaktora Naczelnego, Redaktor Prowadzący:
Damian Sosnowski

Zastępca Redaktora Naczelnego, Menedżer Magazynu
Marcin Karbowniczek

Szef Pracowni Konstrukcyjnej:
Grzegorz Becker, tel.: 22 257 84 58

Redaktor strony internetowej *www.ep.com.pl*
Dariusz Welik

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:
Jacek Bogusz, 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**