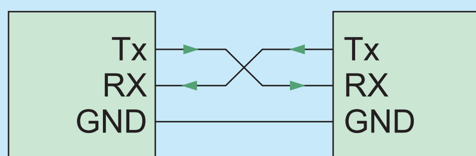
Eksperymenty z FPGA (5)

Dzisiaj zajmiemy się kolejnym peryferium, jakie jest dostępne na płytce „Rysino". Naszym celem będzie uruchomienie komunikacji poprzez port szeregowy. Tak jak poprzednio przed przystąpieniem do wykonywania eksperymentów zachęcam do aktualizacji repozytorium z przykładami (na przykład poprzez wywołanie polecenia `git pull`).



Rysunek 1. Sposób łączenia urządzeń poprzez port szeregowy

Jak działa port szeregowy?

Port szeregowy UART (*Universal Asynchronous Receiver/Transmitter* – uniwersalny asynchroniczny odbiornik/nadajnik) składa się z dwóch linii danych: nadawczej Tx i odbiorczej Rx oraz masy. Obie linie działają niezależnie, więc możliwa jest równoczesna transmisja w obu kierunkach (full-duplex). Jak pokazuje **rysunek 1** łączone są one „na krzyż": wyjście Tx jednego urządzenia łączymy z wejściem Rx drugiego.

Najprostszą ramkę danych przesyłaną przez port szeregowy pokazuje **rysunek 2.** Istnieją także inne warianty różniące się liczbą bitów danych, długością startu, czy obecnością bitu parzystości. Nie będziemy się jednak nimi zajmowali.

W implementowanej przez nas wersji pojedyncza ramka składa się z dziesięciu bitów. W momencie, gdy linia jest wolna panuje na niej stan wysoki. Przesyłanie danych rozpoczyna się od bitu startu. Sygnalizuje on początek transmisji przez zmianę stanu linii na niski. Po nim następuje osiem bitów (jeden bajt) danych zaczynając od najmniej znaczącego bitu. Na końcu znajduje się bit stopu o wartości 1. Dzięki temu linia „płynnie" przechodzi w stan bezczynności. Poprzez zwiększenie liczby bitów stopu, zwiększamy minimalny czas pomiędzy dwoma kolejnymi ramkami.

Już w samym rozwinięciu skrótu UART znajdziemy słowo „asynchroniczny". Nadajnik może rozpocząć przesyłanie nowej ramki w dowolnym momencie. Odbiornik cały czas „nasłuchuje" linie i rozpoczyna zapis ramki po wykryciu bitu startu. Między obydwoma urządzeniami nie ma żadnej innej synchronizacji. Czas trwania pojedynczego stanu jest ustalony parametrem, który musi być znany po obu stronach. Dlatego jednym z głównych czynników ograniczającymi długość pojedynczej ramki danych jest właśnie niedokładność zegarów. Musi być na tyle mała, aby w czasie trwania ramki błąd nie spowodował zgubienia lub dwukrotnego odczytanie któregoś stanu.

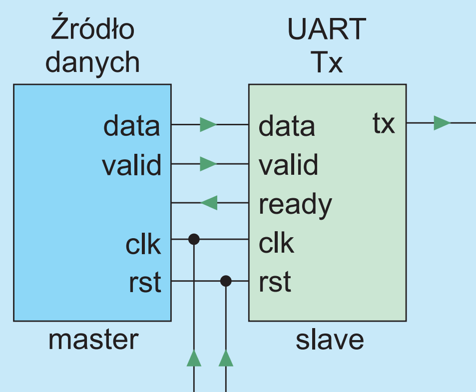
Prędkość przesyłania podawana jest w Bodach, czyli stanach na sekundę. Dane można przesyłać zasadniczo z dowolną prędkością, ale często spotykanymi wartościami są 9600 Bd oraz 115200 Bd. Dla nich jeden stan trwa odpowiednio 104 µs i 8,68 µs. W naszym przypadku jeden stan koduje jeden bit. Jednak ze względu na bit startu i stopu prędkość przesyłu danych to jedynie 80% tej wyrażonej w Bodach. Dlatego dla podanych przykładowych prędkości maksymalne przepustowości wynoszą odpowiednio 7,5 kib/s oraz 90 kib/s (gdzie kib jest skrótem od kibibit, czyli 1024 bity).

„Nadajnik"

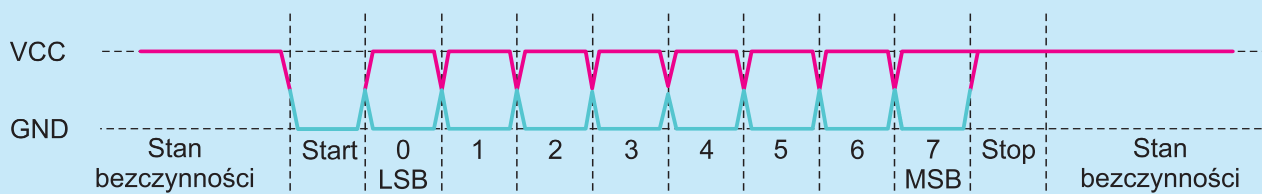
Nasze eksperymenty zaczniemy od łatwiejszego zadania, czyli wysyłania. Przy odbieraniu musimy wykryć początek ramki i dostosować się do niego. Natomiast w części „Tx-owej" po prostu mierzymy czas i ustawiamy odpowiednia stany na linii danych.

Zanim jednak zaczniemy projektować musimy ustalić, w jaki sposób będzie się on komunikował z innymi modułami, które znajdą się w układzie FPGA. Z jednej strony dane do wysłania nie będą dostępne ciągle, więc nadajnik musi wiedzieć kiedy powinien zacząć wysyłać. Z drugiej strony nadawanie zajmuje dość długi czas. Nadanie pojedynczego bajtu przy prędkości 115200 Bd zajmie 86,8 µs co zajmie prawie 700 cykli zamontowanego na płytce zegara. Dlatego musimy także sygnalizować pozostałym modułom, czy nadajnik jest gotowy do przyjęcia kolejnej porcji danych. Musimy więc utworzyć magistralę komunikacyjną, która dzięki dodatkowym sygnałom będzie mogła obsłużyć opisane wyżej przypadki.

Istnieją różne gotowe magistrale danych, takie jak AXI [1], Avalon [2], czy Wishbone [3]. Tutaj wykorzystamy bardzo uproszczoną wersję szyny AXI-Stream. Skupimy się jednak na samej idei, która za nią stoi bez wgłębiania się w wymagania standardu. Szczegółowe informacje można znaleźć tu [4].

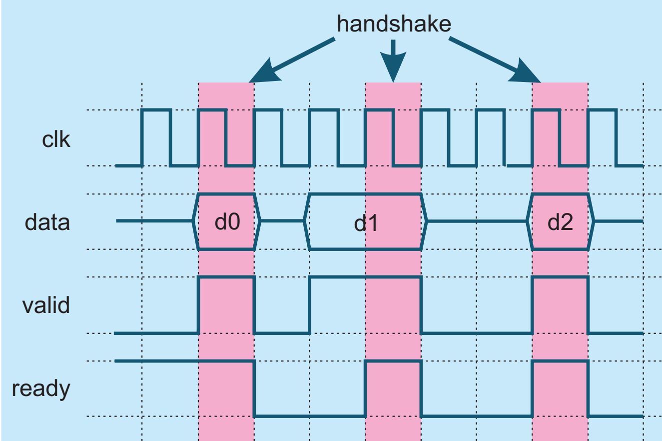


Rysunek 3. Połączenie nadajnika ze źródłem danych



Rysunek 2. Ramka danych w porcie szeregowym

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



Rysunek 4. Przebiegi na magistrali danych, kolorem różowym zaznaczono „handshake”

Rysunek 3 pokazuje połączenie nadajnika ze źródłem danych. Pierwsza linia nazwana `data` (dane), służy po prostu do przekazania danych. W naszym przypadku będzie to 8 bitowy wektor, który pozwoli na przekazanie całej ramki w jednym takcie zegara. Druga linia `valid` (poprawny) informuje, że jeżeli w danym cyklu zegara panuje na niej stan wysoki, to wartości dostępne na linii data są poprawne i należy je obsłużyć. Sygnał `ready` (gotowy) płynie w przeciwnym kierunku niż dwa poprzednie. Za jego pomocą `slave` (układ podrzędny) informuje `mastera` (układ nadrzędny) o gotowości do przyjęcia kolejnej porcji danych.

Przykładowe przebiegi uzyskane na magistrali pokazuje **rysunek 4**. Na początku widzimy, że sygnał `ready` jest w stanie wysokim. Układ podrzędny jest więc gotowy i czeka na nowe dane. Pojawiają się one na kolejnym takcie zegara, gdy sygnał `valid` przechodzi w stan wysoki. Stan gdy oba sygnały: `valid` i `ready` są równocześnie w stanie wysokim nazywa się `handshake` (uścisk dłoni). W momencie tym następuje przekazanie danych z mastera do slavea. Ponieważ dane d0 zostały wysłane, a kolejne jeszcze się nie pojawiły sygnał `valid` zostaje zdjęty. Podobnie zachowuje się nasz nadajnik: otrzymał dane i rozpoczął ich przetwarzanie, dlatego sygnał `ready` przyjął stan niski.

Gdy kolejna porcja danych (oznaczona jako d1) jest gotowa znowu następuje ustawienie linii `valid`. Tym razem jednak `slave` nie jest jeszcze gotowy do obsłużenia kolejnej porcji. Master musi więc potrzymać d1 na szynie dopóki dane nie zostaną obsłużone. Gdy nastąpi kolejny `handshake` d1 zostaną przekazane do urządzenia podrzędnego.

Jak widzimy dla d2 może też zdarzyć się sytuacja, że oba sygnały: `ready` i `valid` „zapalą się” równocześnie. Wtedy taka porcja danych zostanie obsłużona natychmiast.

Listing 1. Implementacja interfejsu w języku SystemVerilog (*05_UART/uart_pkg.sv*)

```
10 interface StreamBus #(
11         parameter N = 8
12     ) (
13         input wire clk,
14         input wire rst
15     );
16     logic [N-1:0] data;
17     logic valid;
18     logic ready;
19
20     modport Master (
21         input clk,
22         input rst,
23         input ready,
24         output data,
25         output valid
26     );
27
28     modport Slave (
29         input clk,
30         input rst,
31         input data,
32         input valid,
33         output ready
34     );
35
36 endinterface
```

Gdy sygnał `valid` jest opuszczony, stan linii danych nie powinien mieć wpływu na działanie układu. Dlatego dla tych chwil stan linii data został symbolicznie oznaczony poziomą linią położoną na wysokości stanu niedozwolonego (w połowie drogi pomiędzy 0 a 1).

Zostały jeszcze dwa sygnały, których znaczenie powinno być łatwe do odgadnięcia. Są to `clk`, czyli zegar oraz `rst`, czyli reset. Są to wspólne linie, które dla obu modułów są traktowane jako wejścia.

Aby ułatwić modelowanie magistral w języku SystemVerilog wprowadzono interfejsy. Na **listingu 1** przedstawiono opis naszej magistrali. Widzimy, że składniowo interfejsy są bardzo podobne do modułów. Definicja rozpoczyna się w linii 10 od słowa kluczowego interface, po którym podajemy wymyśloną przez nas nazwę: `StreamBus`. Interfejsy, podobnie jak moduły mogą być parametryzowane. W naszym przypadku stworzymy jeden parametr N określający długość szyny danych i przypiszemy mu domyślną wartość 8. W liniach 13 i 14 zdefiniowane są wejścia. Są to te sygnały, które są wspólne dla wszystkich elementów magistrali. Jak pamiętamy z rysunku 3 w naszym przypadku jest to reset oraz zegar. W środku interfejsu zdefiniowane są jego wewnętrzne sygnały. U nas są to pozostałe trzy linie: `data`, `valid` i `ready`. W tym miejscu nie ustalamy ich kierunku ale na razie jedynie typ.

Ostatnią częścią są tak zwane modport. Dla naszego interfejsu są przygotowane dwa: `Master` (linie 20 do 26) oraz `Slave` (linie 28 do 34). Wewnątrz nich określamy kierunki poszczególnych linii. Później w definicji modułu, w którym chcemy użyć interfejsu określamy której wersji chcemy użyć. Decydujemy o tym umieszczając w liście sygnałów odpowiednio `StreamBus.Master` albo `StreamBus.Slave`.

Przykładowe wykorzystanie interfejsu pokazuje **listing 2**. Jest to nagłówek modułu `uart_tx`, czyli naszego nadajnika. Najpierw przyjmuje on dwa parametry. `F` odpowiada częstotliwości zegara wyrażonej w hercach, a `BAUD` liczbę symboli na sekundę. Następnie w liniach 14 i 15 znajdziemy listę sygnałów. Pierwszym z nich jest nasz interfejs pracujący w trybie `Slave`, a drugim linia nadawcza portu szeregowego oznaczona jako `tx`. Widzimy, że dzięki wykorzystaniu interfejsu liczba wejść/wyjść uległa dużemu skróceniu.

Maszyna stanów

Zastanówmy się teraz co tak naprawdę powinien robić nasz nadajnik. Przez większość czasu będzie utrzymywał na wyjściu `Tx` stan wysoki i czekał na pojawienie się na interfejsie danych do wysłania. Gdy takowe się pojawią powinien przejść do procedury wysyłania: najpierw bit startu, potem kolejne bity danych i na końcu bit stopu. Musi przy tym wziąć pod uwagę, że czas trwania symbolu nie jest równy jednemu taktowi zegara, lecz musi być zgodny z zadanym. Po wysłaniu całego symbolu powinien znowu ustawić na wyjściu stan wysoki

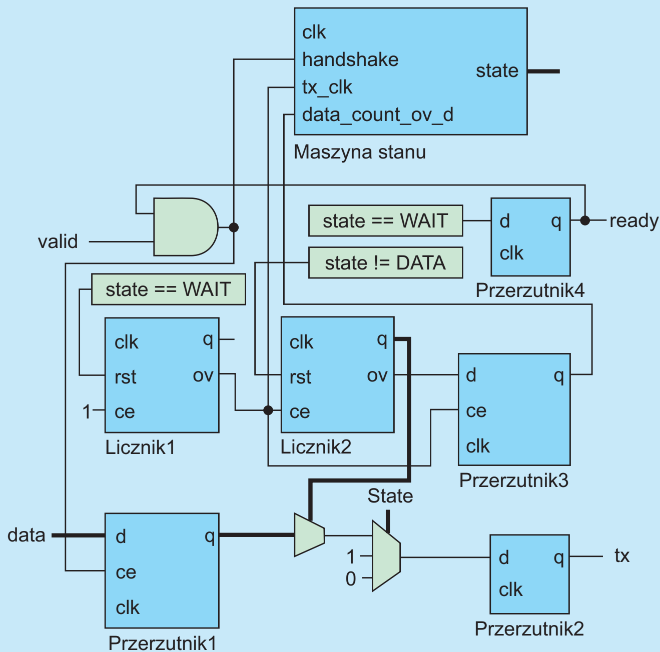
Listing 2. Wykorzystanie interfejsu w module (*05_UART/uart_tx.sv*)

```
10 module uart_tx #(
11     parameter F = 8000000,
12     parameter BAUD = 115200
13 ) (
14     StreamBus.Slave bus,
15     output logic tx
16 );
```

Listing 3. Definicja typu wyliczeniowego state umieszczona w pakiecie uartPkg (*05_UART/uart_pkg.sv*)

```
38 package uartPkg;
39
40     typedef enum logic [1:0] {
41         WAIT,
42         START,
43         DATA,
44         STOP
45     } state;
46
47 endpackage : uartPkg
```


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



Rysunek 5. Uproszczony schemat nadajnika. Wszystkie wejścia clk są podłączone do sygnału zegarowego

i rozpocząć oczekiwanie na kolejną daną. Możemy więc przygotować listę stanów, w których może znaleźć się nasz moduł oraz podać przepis na wyznaczenie kolejnego stanu w zależności od aktualnego oraz wejść naszego modułu. W naszym przypadku będzie ich cztery:

- `WAIT` – oczekiwanie na dane,
- `START` – wysłanie bitu startu,
- `DATA` – przesyłanie kolejnych bitów danych
- `STOP` – wysyłanie bitu stopu.

Aby posługiwać się w kodzie nazwami zdefiniujemy typ wyliczeniowy `state`, pokazany na **listingu 3**. Jak widzimy w linii 4 będzie on przechowywany jako wektor typu `logic`. Ponieważ może przyjmować 4 różne wartości wystarczą nam więc 2 bity do jego przechowania. Aby używać naszego nowego typu danych w różnych modułach umieścimy go w pakiecie nazwanym `uartPkg`. Dzięki temu w projektach do których dołączymy nasz pakiet będziemy mogli odnieść się do naszego typu poprzez odwołanie postaci: `uartPkg::state`.

Mamy już sposób na reprezentację poszczególnych stanów, teraz zaprojektujmy hardware, którym będzie sterował. Pokazany został na **rysunku 5**, dla lepszej czytelności pominięte zostały połączenia wejść `clk` (wszystkie są przyłączone do sygnału zegarowego). Na początku, gdy pojawią się nowe dane zostaną one zapisane w przerzutniku 1. Jak widzimy, jego sygnał `CE` jest w stanie wysokim, gdy nastąpi handshake. Dzięki temu dane nie ulegną zmianie w czasie wysyłania.

Stan linii wyjściowej jest ustalany za pomocą dwóch multiplekserów. Pierwszy z nich pozwala na wybranie jednego z 8 bitów danych. Jest on sterowany za pomocą licznika 2. Drugi z nich wybiera czy obecnie na wyjściu mają pojawić się dane, bit startu (0), czy stan wolnej linii (1). Tutaj decyzja jest podejmowana na podstawie aktualnego stanu. Sygnał `Tx` na wyjściu jest zatrzaskiwany w przerzutniku 2. Implementację „ścieżki danych” pokazuje **listing 4**.

Listing 4. Implementacja ścieżki danych (*05_UART/uart_tx.sv*)

```
41 always_ff @(posedge bus.clk)
42     if (handshake)
43         txb <= bus.data;
44
45 always_ff @(posedge bus.clk)
46     case (state)
47         uartPkg::START: tx <= 1'b0;
48         uartPkg::DATA: tx <= data_count_ov_d ? 1'b1 : txb[i];
49         default: tx <= 1'b1;
50     endcase
```

Listing 5. Implementacja liczników (*05_UART/uart_tx.sv*)

```
52      assign ctx_rst = (state == uartPkg::WAIT);
53      counter #(.N((F+BAUD/2)/BAUD)) ctx (
54          .clk(bus.clk),
55          .rst(!ctx_rst),
56          .ce(1'b1),
57          .q(),
58          .ov(tx_clk)
59      );
60
61     assign data_count_rst = (state != uartPkg::DATA);
62     counter #(.N(8)) data_count (
63         .clk(bus.clk),
64         .rst(!data_count_rst),
65         .ce(tx_clk),
66         .q(i),
67         .ov(data_count_ov)
68     );
69
70     always_ff @(posedge bus.clk)
71         if (tx_clk)
72             data_count_ov_d <= data_count_ov;
```

Listing 6. Wyznaczanie sygnału ready (*05_UART/uart_tx.sv*)

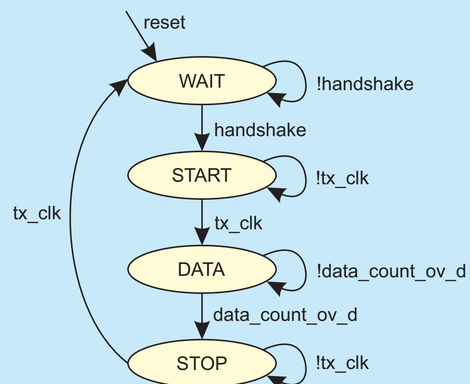
```
39 assign bus.ready = (state == uartPkg::WAIT);
```

Ponieważ prędkość wysyłania jest mniejsza od częstotliwości taktowania potrzebujemy odpowiedniego sygnału zegarowego. Jest on generowany przez licznik 1, uruchamiany, gdy stan jest inny niż `WAIT`. Dzięki temu licznik czasu jest zsynchronizowany z początkiem transmisji. Jego przepełnienie jest wykorzystane przy wyznaczaniu następnego stanu oraz jako sygnał `CE` dla licznika 2. Wyznacza on, który bit ma być obecnie wysłany. Pracuje jedynie wtedy, gdy moduł znajduje się w stanie `DATA`. Przerzutnik 3 dostarcza informacji o zakończeniu wysyłania ostatniego bitu. Jest to sygnał przepełnienia licznika 2 opóźniony o czas trwania jednego symbolu. Implementacja tego fragmentu jest pokazana na **listingu 5**. Jak widzimy wykorzystane zostały stworzone wcześniej moduły licznika.

Zatrzymajmy się chwilę nad ustaleniem zakresu zliczania dla pierwszego licznika. Ma on postać:

`(F+BAUD/2)/BAUD`

gdzie `F` to częstotliwość zegara taktującego moduł, a `BAUD` to oczekiwana szybkość transmisji. Wyjaśnienia wymaga dodanie wartości `BAUD/2` przed wykonaniem dzielenia. Gdyby wykonywać obliczenia na liczbach rzeczywistych jej obecność wydaje się wręcz błędem. Jednak tutaj obliczeń dokonujemy na liczbach całkowitych, a uzyskana część ułamkowa jest obcinana. Jest to tak zwane zaokrąglanie w dół. Dodając do dzielnej połowę dzielnika zmieniamy to zachowanie. Można to rozumieć jako dodane 0,5 przed wykonaniem zaokrąglenia w dół, co spowoduje że dla części ułamkowej mniejszej niż 0,5 nastąpi zaokrąglenie w dół, a dla większej niż 0,5 w górę. Aby uzyskać efekt zaokrąglenia w górę zamiast `BAUD/2` musielibyśmy dodać `BAUD-1`.



Rysunek 6. Maszyna stanów nadajnika

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

Ostatnim nieomówionym elementem generowanie sygnału ready dla naszej magistrali. Jego implementacja jest prosta: nowe dane możemy przyjąć tylko gdy jesteśmy w stanie bezczynności WAIT. Jak pokazuje **listing 6** implementacja tego warunku jest króciutka.

Wiemy już co ma się dziać w poszczególnych stanach. Teraz musimy ustalić w jaki sposób nasz moduł będzie przechodził pomiędzy nimi. Zostało to pokazane na **rysunku 6**. Na samym początku, zaraz po resecie, znajdziemy się w stanie WAIT. Decyzję o wyborze następnego stanu musimy podjąć na każdym narastającym zboczu zegara. W stanie WAIT decyzję podejmujemy na podstawie wystąpienia handshakeu. Ponieważ ustawienie sygnału ready jest równoznaczne z przebywaniem w stanie WAIT wystarczy, że sprawdzimy sygnał valid. Jeśli pojawią się nowe dane do wysłania, przejdziemy do stanu START, a w przeciwnym razie pozostaniemy w WAIT. Następnie musimy nadać bit startu, dlatego w stanie START będziemy tak długo, aż minie czas nadawania jednego symbolu. Zostaniemy o tym poinformowani poprzez wystąpienie przepełnienia od licznika, które w naszej maszynie trafi na wejście `tx_clk`. W stanie DATA będziemy przez czas trwania kolejnych 8 stanów. O wysłaniu ostatniego bitu poinformuje naszą maszynę stan wysoki na wejściu `data_count_ov`. W stanie STOP znowu będziemy przebywać przez czas jednego symbolu, czyli do pojawienia się stanu wysokiego na sygnale `tx_clk`. Spowoduje on powrót do stanu WAIT i oczekiwanie na nowe dane.

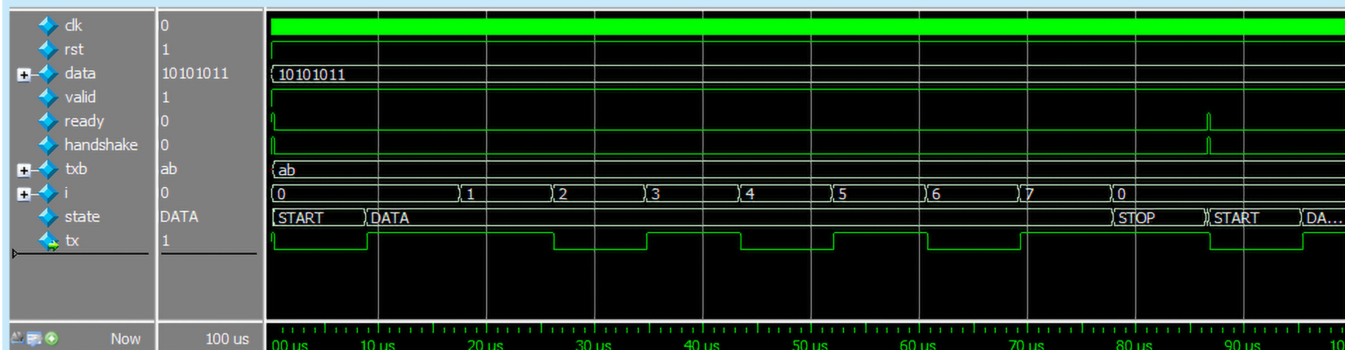
Implementację maszyny stanów prezentuje **listing 7**. Zaczynamy od asynchronicznego resetu znajdującego się w liniach 28 i 29.

Listing 7. Implementacja maszyny stanów (05_UART/uart_tx.sv)

```
17 uartPkg::state state;
27 always_ff @(posedge bus.clk or negedge bus.rst)
28  if (!bus.rst)
29      state <= uartPkg::WAIT;
30  else
31      case (state)
32          uartPkg::WAIT: state <= bus.valid ? uartPkg::START :uartPkg::WAIT;
33          uartPkg::START: state <= tx_clk ? uartPkg::DATA : uartPkg::START;
34          uartPkg::DATA: state <= data_count_ov_d ? uartPkg::STOP : uartPkg::DATA;
35          uartPkg::STOP: state <= tx_clk ? uartPkg::WAIT : uartPkg::STOP;
36          default: state <= uartPkg::WAIT;
37      endcase
```

Listing 8. Test bench dla nadajnika UART (*05_UART/uart_tx_tb.sv*)

```
11 module uart_tx_tb;
12  logic clk;
13  logic rst;
14  StreamBus bus(clk, rst);
15
16  initial begin
17      clk <= '0;
18      forever #625 clk <= ~clk;
19  end
20
21  initial begin
22      rst <= 1'b0;
23      #1250 rst <= 1'b1;
24  end
25
26  initial begin
27      bus.valid <= 1'b0;
28      #1250;
29      bus.valid <= 1'b1;
30      bus.data <= 8'hab;
31  end
32
33  uart_tx dut(
34      .bus(bus),
35      .tx()
36  );
37 endmodule
```



Rysunek 7. Wynik symulacji nadajnika

Następnie funkcja przejścia została zrealizowana w bloku `case`, w którym po kolei dla każdego stanu następuje sprawdzenie warunków z rysunku 6. Dla skrócenia zapisu zamiast polecenia if wykorzystany został trójargumentowy `operator ?:`.

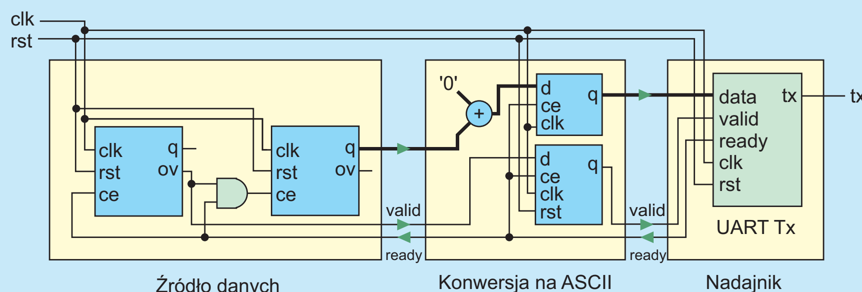
Mamy gotowy moduł. Potrzebujemy teraz test bench'u do sprawdzenia jego działania w symulacji. Fragment kodu został pokazany na **listingu 8**. W liniach 12 i 13 zdefiniowaliśmy dwa sygnały które posłużą do generowania sygnału zegarowego i resetu. Natomiast w linii 14 korzystającą z nich stworzyliśmy instancję naszego interfejsu `StreamBus` o nazwie `bus`. Dalej trzech kolejnych blokach initial generujemy sygnał zegarowy, reset oraz dane. Szczególnie ciekawy jest ten trzeci z nich ponieważ korzystając z kropki odwołujemy się do poszczególnych sygnałów wewnątrz interfejsu. Na końcu umieszczamy instancję modułu `uart_tx` i podłączamy do niej interfejs. Symulację uruchamiamy w programie ModelSim za pomocą polecenia:

```
do uart_tx_sim.do
```

(pamiętając o wcześniejszym przejściu do folderu zawierającego źródła). Powinniśmy zobaczyć wynik podobny do tego z **rysunku 7**. Tutaj widoczna jest kolejna zaleta wykorzystania typu wyliczeniowego. Zamiast wartości liczbowych wypisane zostały nazwy kolejnych stanów.

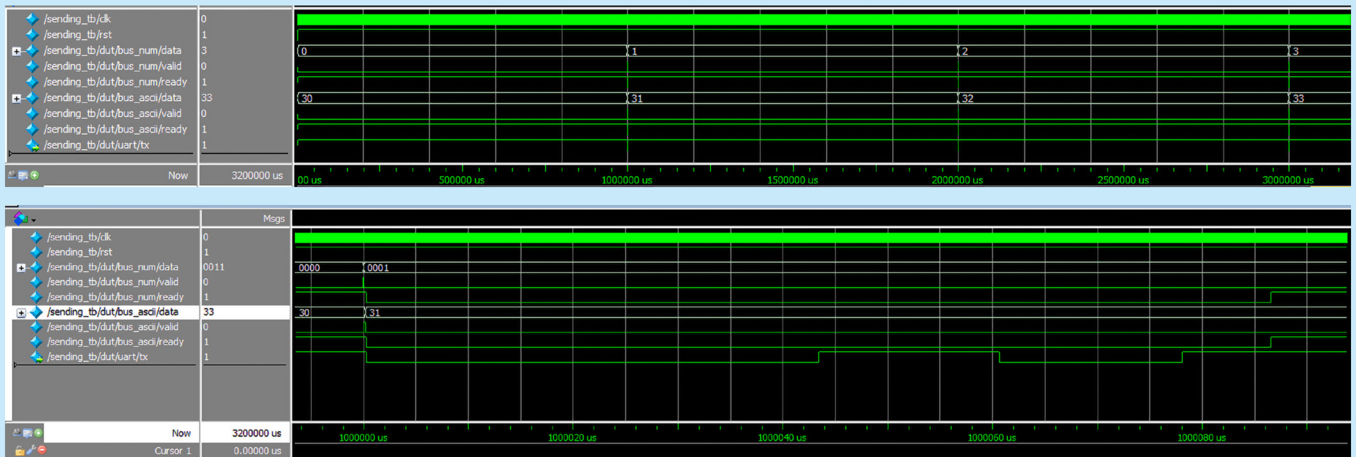
Źródło danych

Mamy już gotowy i sprawdzony nadajnik. Aby jednak wykorzystać go w sprzęcie potrzebujemy jeszcze źródła danych. Stworzymy więc bardzo prosty projekt, którego zadaniem będzie wysłanie na port szeregowy kolejnych liczb od 0 do 9 zakodowanych w formacie ASCII.



Rysunek 8. Moduł wypisujący kolejne liczby

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



Rysunek 9. Wynik symulacji modułu sending

Jego schemat pokazuje **rysunek 8**. Przy okazji jego budowy zapoznamy się z działaniem naszej magistrali w praktyce.

Zaczynając od lewej strony pierwszym modułem jest źródło danych. Składa się ono z dwóch liczników. Pierwszy z nich zapewnia opóźnienie pomiędzy kolejnymi danymi, a drugi generuje kolejne liczby do wysłania. Sygnał `valid` jest wyzwalany w momencie przepełnienia pierwszego licznika. Ciekawa jest obsługa sygnału ready – steruje on sygnałami `CE`. Jeżeli druga strona magistrali nie jest gotowa następuje zablokowanie pracy źródła. Jego implementacja jest pokazana na **listingu 9**. Posiada tylko jedno wejście – naszą magistralę (linia 13). Następnie znajdują się dwie instancje licznika. Pierwszy z nich (linie 17...23) generuje sygnał przepełnienia co około sekundę. Drugi natomiast realizuje zliczanie od 0 do 9 (linie od 25 do 31).

Kolejnym elementem jest konwersja na ASCII. Jest to bardzo prosty moduł. Na danych przepływających przez magistralę realizuje on dodanie 48 czyli wartości '0' w ASCII. Tak naprawdę jest to po prostu doklejenie do wejściowych bitów wartości `4'b0011`. Wyjściowy wynik jest następnie zapisany w rejestrze. Warto zwrócić uwagę, że sygnał `valid`, musi „płynąć" razem z danymi, których dotyczy. Brak sygnału ready, podobnie jak poprzednio, powodują zatrzymanie pracy modułu. Natomiast wyjście `ready` jest po prostu zwarte z jego wejściem. Jest to wada tego typu magistrali, ponieważ powstaje sygnał, który steruje stosunkowo dużą ilość logiki i nie może zostać „zpipelinowany" poprzez dodanie dodatkowych rejestrów. Może on powodować problemy z domknięciem relacji czasowych podczas budowania projektu. Więcej na ten temat można znaleźć między innymi w [5].

Jego implementacja jest pokazana na **listingu 10**. Jako wejścia podane są dwie magistralę. Wejściowa pracuje w trybie `Slave`, a wyjściowa `Master`. Wewnątrz modułu znajdziemy dwa bloki `always_ff`. Pierwszy z nich (linie 15...17) realizuje operację na danych, a drugi (linie 19...23) opóźnia sygnał `valid`. Widzimy, że resetowany jest jedynie ten drugi. Jeżeli sygnał `valid` jest w stanie niskim dane znajdujące się na magistrali i tak nie zostaną zinterpretowane. Nie ma więc potrzeby ich resetować.

Wszystkie trzy przedstawione bloki zostały połączone w module `sending`. Jest to główny moduł naszego projektu, który uruchomimy w układzie FPGA. Jego kod pokazuje **listing 11**. Ma on dwa wejścia: zegar i reset oraz jedno wyjście `tx`. W liniach 17 i 18 zdefiniowane są dwa interfejsy, którymi połączymy modułu. Następnie znajdują się kolejne instancję omówionych wcześniej modułów. Prędkość transmisji została ustalona na 115200 Bodów.

Przed rozpoczęciem budowania warto jeszcze uruchomić symulację, którą znajdziemy w pliku *05_UART/sending_tb.sv*. Możemy w tym celu wykorzystać skrypt:

```
do sending_sim.do
```

Listing 9. Implementacja źródła danych (*05_UART/data_source.sv*)

```
10 module data_source #(
11     parameter F = 8000000
12 ) (
13     StreamBus.Master bus
14 );
15 logic c_1s_ov;
16
17 counter #(.N(F)) c_1s (
18     .clk(bus.clk),
19     .rst(bus.rst),
20     .ce(bus.ready),
21     .q(),
22     .ov(c_1s_ov)
23 );
24
25 counter #(.N(10)) c_data (
26     .clk(bus.clk),
27     .rst(bus.rst),
28     .ce(c_1s_ov & bus.ready),
29     .q(bus.data),
30     .ov()
31 );
32
33 assign bus.valid = c_1s_ov;
34
35 endmodule
```

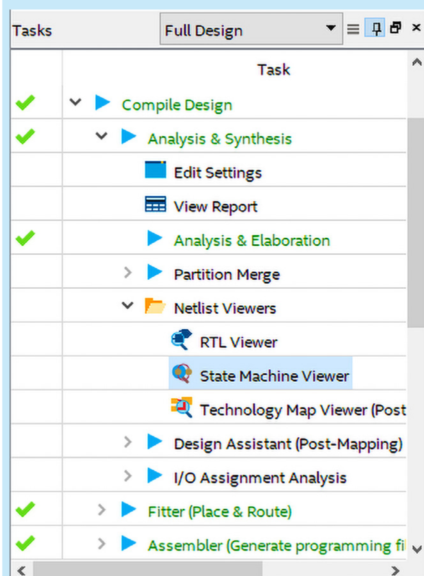
Listing 10. Implementacja źródła danych (*05_UART/to_ascii.sv*)

```
10 module to_ascii (
11  StreamBus.Slave bus_in,
12  StreamBus.Master bus_out
13 );
14
15  always_ff @(posedge bus_in.clk)
16  if (bus_out.ready)
17      bus_out.data <= 8'd48 + bus_in.data;
18
19  always_ff @(posedge bus_in.clk or negedge bus_in.rst)
20  if (!bus_in.rst)
21      bus_out.valid = 1'b0;
22  else if (bus_out.ready)
23      bus_out.valid <= bus_in.valid;
24
25  assign bus_in.ready = bus_out.ready;
26 endmodule
```

Listing 11. Moduł sending (*05_UART/sending.sv*)

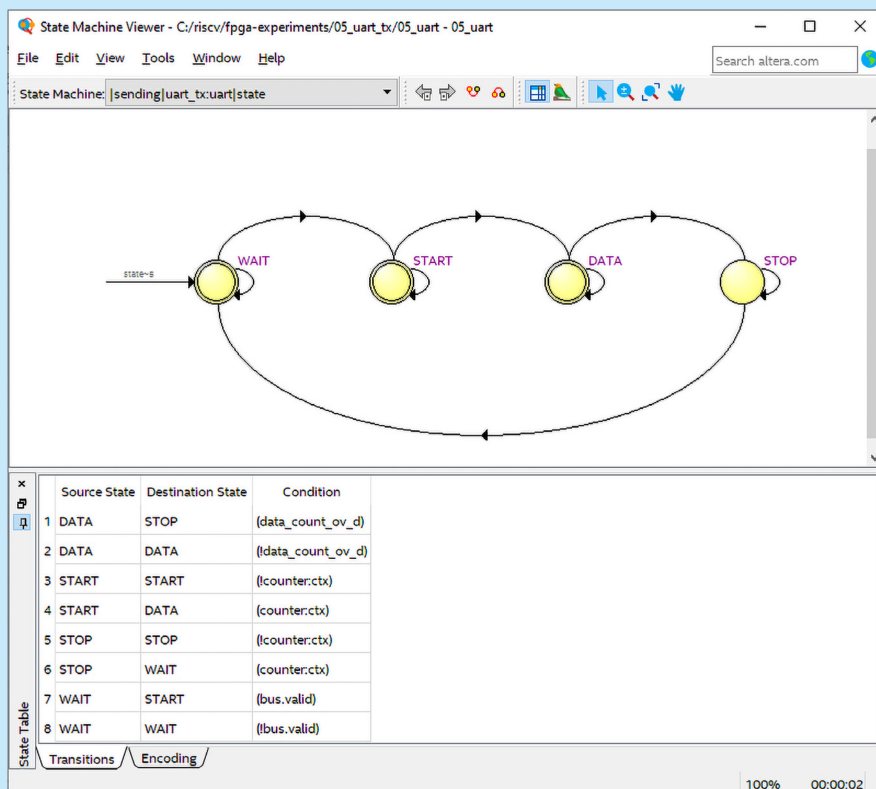
```
10 module sending #(
11  parameter F = 8000000
12 ) (
13  input wire clk,
14  input wire rst,
15  output logic tx
16 );
17  StreamBus #(.N(4)) bus_num (.clk(clk), .rst(rst));
18  StreamBus #(.N(8)) bus_ascii (.clk(clk), .rst(rst));
19
20  data_source #(.F(F)) source (
21      .bus(bus_num)
22  );
23
24  to_ascii ascii (
25      .bus_in(bus_num),
26      .bus_out(bus_ascii)
27  );
28
29  uart_tx #(
30      .F(F),
31      .BAUD(115200)
32  ) uart (
33      .bus(bus_ascii),
34      .tx(tx)
35  );
34
35 endmodule
```

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



Rysunek 10. Otwieranie podglądu maszyny stanów

Ponieważ czas symulacji to nieco ponad 3 sekundy, jej wykonanie może zająć kilka minut. Gdy już dojdzie do końca zobaczymy efekt podobny do pokazanego na **rysunku 9**.



	Source State	Destination State	Condition
1	DATA	STOP	(data_count_ov_d)
2	DATA	DATA	(!data_count_ov_d)
3	START	START	(!counter.ctx)
4	START	DATA	(counter.ctx)
5	STOP	STOP	(!counter.ctx)
6	STOP	WAIT	(counter.ctx)
7	WAIT	START	(bus.valid)
8	WAIT	WAIT	(!bus.valid)

Rysunek 11. Okno programu *State Machine Viewer*

Czas na sprzęt

Ostatnim krokiem jest zbudowanie projektu i uruchomienie go w układzie FPGA. Uruchamiamy środowisko Quartus i otwieramy projekt *05_uart_tx/05_uart.qpf*. Podobnie jak poprzednio rozpoczynamy budowę. Po jej zakończeniu wrócimy jeszcze na chwilę do naszej maszyny stanów. Okazuje się, że środowisko potrafi wykryć jej obecność w kodzie i wykonać dodatkowe optymalizacje.

Aby się o tym przekonać z panelu *Tasks* wybieramy opcję *Analysis & Synthesis*, następnie *Netlist Viewers*, gdzie znajdziemy *State Machine Viewer* (**rysunek 10**). Pojawi się okno podobne to tego z **rysunku 11**. W środkowej części okna widzimy graf przejść naszej maszyny. Nazwy stanów zostały wyciągnięte z użytego trybu wyliczeniowego. W dolnej części okna w zakładce *Transitions* widzimy listę możliwych przejść z podanymi warunkami. W zakładce *Encodeing* (**rysunek 12**) widzimy w jaki sposób stan będzie reprezentowany logice. Widzimy, że zamiast 2 bitów są wykorzystane 4.

Wróćmy jednak do sprzętu. Gdy wsad dla układu FPGA zostanie wygenerowany możemy podłączyć „Rysino" do komputera i rozpocząć programowanie. Gdy zakończy się ono powodzeniem otwieramy dowolny monitor portu szeregowego. Ja wykorzystałem program Realterm [6]. W zakładce Port wybieramy numer portu szeregowego oraz ustawiamy prędkość transmisji na 115200. Po zatwierdzeniu przyciskiem *Change* w polu danych powinny pojawiać się kolejne cyfry co zostało pokazane na **rysunku 13**.

Podsumowanie

W tym odcinku uruchomiliśmy nadajnik portu szeregowego, a przy okazji przyglądnęliśmy się koncepcji maszyny stanów oraz magistrali. W następnej części zajmiemy się drugą stroną portu szeregowego, czyli odbieraniem danych.

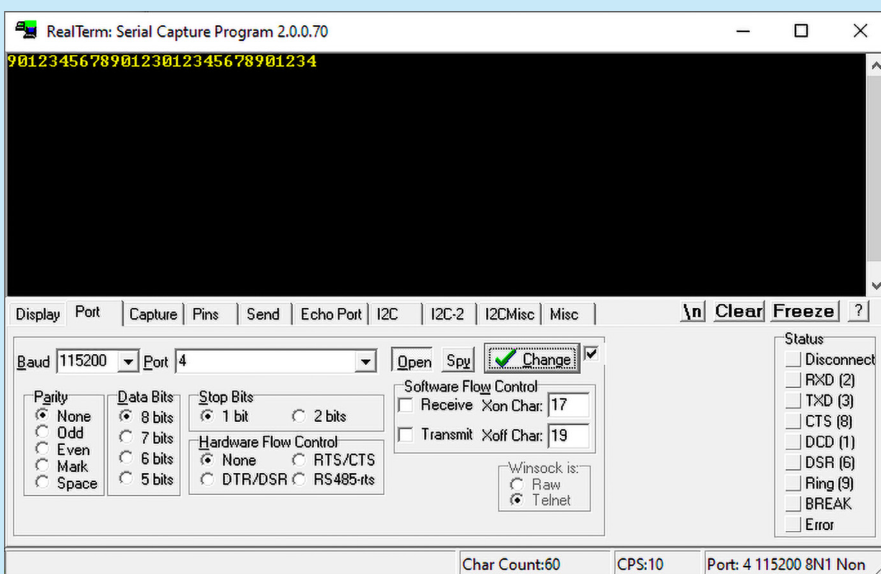
Rafał Kozik
rafkozik@gmail.com

[1] AMBA® AXI™ and ACE™ Protocol Specification, ARM 2019, ***http://bit.ly/38NHTVk***
[2] Avalon® Interface Specifications, Intel 2019, ***https://intel.ly/2Qaz1CI***
[3] WISHBONE System-on--Chip (SoC)Interconnection Architecturefor Portable IP Cores, Open Cores 2010, ***http://bit.ly/2wVy3n1***
[4] AMBA® 4 AXI4-Stream Protocol, ARM 2010, ***http://bit.ly/2W6coTL***
[5] Building a Skid Buffer for AXI processing, ***http://bit.ly/2TXWIQ0***
[6] ***http://bit.ly/2TJx3LJ***

	Name	STOP	DATA	START	WAIT
1	WAIT	0	0	0	0
2	START	0	0	1	1
3	DATA	0	1	0	1
4	STOP	1	0	0	1

Transitions | Encoding

Rysunek 12. Zakładka *Encoding*



Rysunek 13. Okno programu Realterm

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