Układy Cyfrowe i Systemy Wbudowane

Implementacja dekodera USB 1.1 Full-Speed na platformie FPGA Spartan 3E

Autorzy :

Jakub Smolarczyk 272924

Kacper Wieczorek 264471

28.05.2025

Gr. śr. P. 13:15 – 16:15

Spis treści

1. Wprowadzenie

1.1. Temat, cel i zakres projektu

Tematem niniejszego projektu była realizacja modułu dekodera dla interfejsu Universal Serial Bus (USB), działającego zgodnie ze standardem USB 1.1. Implementacja została przeprowadzona na platformie sprzętowej opartej o układ FPGA (Field-Programmable Gate Array) z rodziny Spartan 3E.

Głównym celem projektu było zaprojektowanie oraz implementacja w języku opisu sprzętu VHDL systemu zdolnego do poprawnego odbioru i dekodowania danych przesyłanych strumieniowo przez port USB. Projekt koncentrował się na przechwyceniu sygnału cyfrowego na podstawie sygnałów różnicowych D+ i D- zgodnie ze specyfikacją USB 1.1, dla transmisji typu Full-Speed (12 Mbit/s). Kluczowymi elementami implementacji było wierne odtworzenie logiki dekodowania danych w formacie NRZI (Non-Return-to-Zero Inverted) oraz zaimplementowanie mechanizmu usuwania bitów wtrąconych (ang. bit unstuffing), który jest niezbędny do prawidłowej interpretacji ramek danych i utrzymania synchronizacji.

Zakres projektu obejmował następujące etapy:

- Dokładną analizę dokumentacji technicznej standardu USB 1.1, ze szczególnym uwzględnieniem specyfikacji warstwy fizycznej (PHY) dla transmisji Full-Speed, w tym stanów linii (Idle, SE0, J, K), kodowania danych oraz struktury pakietów (SYNC, PID, DATA/ADR, CRC, EOP).
- Zaprojektowanie architektury modułu dekodującego w języku VHDL, uwzględniającej logikę detekcji początku i końca pakietu (SoP, EoP), synchronizacji z przychodzącym strumieniem bitów, dekodowania NRZI oraz algorytmu usuwania bitów wtrąconych.
- Implementację poszczególnych bloków funkcjonalnych oraz ich integrację w kompletny system dekodera.
- Przeprowadzenie symulacji funkcjonalnych oraz, w miarę możliwości, czasowych w celu weryfikacji poprawności działania zaprojektowanego modułu w różnych scenariuszach testowych.
- Syntezę i implementację (place and route) projektu na docelowej platformie FPGA Spartan 3E, wraz z konfiguracją odpowiednich przypisań pinów.
- Przeprowadzenie testów praktycznych z wykorzystaniem rzeczywistego sygnału USB, pochodzącego z podłączonego kabla USB.

Projekt skupiał się na funkcjonalności odbiornika USB i dekodowaniu warstwy fizycznej. Zakres **nie obejmował**: implementacji pełnego stosu protokołu USB (np. obsługi transakcji, transferów, enumeracji urządzeń), obsługi trybu Low-Speed lub High-Speed, mechanizmów zaawansowanej obsługi błędów transmisji takich jak CRC5 i CRC16, implementacji funkcji nadajnika USB, ani obsługi różnych klas urządzeń USB deskryptorów na poziomie aplikacyjnym.

1.2. Wykorzystane interfejsy i protokoły

Kluczowym protokołem, którego dotyczyła implementacja, był Universal Serial Bus (USB) w wersji 1.1, pracujący w trybie Full-Speed (prędkość transmisji 12 Mbit/s). Standard ten definiuje zarówno aspekty elektryczne warstwy fizycznej, jak i formatowanie danych oraz podstawowe typy pakietów (Token, Data, Handshake, Special). W ramach projektu zaimplementowano logikę odbioru i interpretacji sygnałów zgodnie z tą specyfikacją, w szczególności:

- Odbiór i interpretację różnicowych sygnałów D+ i D- w celu odtworzenia sekwencji bitów.
- Dekodowanie NRZI (Non-Return-to-Zero Inverted), gdzie zmiana poziomu logicznego na liniach transmisyjnych reprezentuje bit ’0’, natomiast brak zmiany poziomu logicznego oznacza transmisję bitu ’1’.
- Mechanizm bit unstuffing (odwrotność procesu bit stuffing), polegający na automatycznym usuwaniu bitu ’0’, który jest wstawiany przez nadajnik po każdej sekwencji sześciu kolejnych bitów ’1’. Jest to niezbędne do zapewnienia odpowiedniej liczby przejść sygnału dla utrzymania synchronizacji odbiornika.

W projekcie wykorzystano również wewnętrzne interfejsy typowe dla systemów cyfrowych opartych na FPGA, takie jak sygnały zegarowe do taktowania logiki, sygnały resetujące, a także interfejsy do komunikacji z innymi peryferiami dostępnymi na płytce Spartan 3E takimi jak diody LED do sygnalizacji stanu dekodera oraz przycisk do jego resetu.

1.3. Wykorzystana platforma sprzętowa

Do realizacji zadań projektowych oraz testowania zaimplementowanego dekodera USB wykorzystano następującą platformę sprzętową oraz narzędzia:

- Płytka rozwojowa: Xilinx Spartan 3E Starter Kit, wyposażona w układ FPGA XC3S500E. Płytka ta dostarcza niezbędną infrastrukturę, w tym złącze USB, generatory zegarowe, oraz podstawowe peryferia I/O takie jak diody LED czy przyciski.
- Komputer PC: Wyposażony w oprogramowanie Xilinx ISE Design służące do projektowania (edycji kodu VHDL), symulacji (ISim), syntezy, implementacji (place and route) oraz programowania konfiguracji układu FPGA poprzez interfejs JTAG.

2. Podstawy teoretyczne

Niniejsza sekcja przedstawia kluczowe zagadnienia teoretyczne związane ze standardem USB 1.1 oraz technikami kodowania i transmisji danych, które były niezbędne do zrozumienia i realizacji projektu.

2.1. Standard USB 1.1 – Warstwa fizyczna i format pakietów

Standard Universal Serial Bus (USB) w wersji 1.1 definiuje dwie prędkości transmisji: Low-Speed (1.5 Mbit/s) oraz Full-Speed (12 Mbit/s) [1]. Projekt koncentrował się na trybie Full-Speed.

- Sygnalizacja różnicowa: USB wykorzystuje dwie linie danych, D+ i D-, do transmisji sygnału w sposób różnicowy. Zmniejsza to podatność na zakłócenia. Stany logiczne są definiowane przez różnicę napięć między D+ a D-.

  – Stan J (Full-Speed): D+ > D- (np. D+ wysoki, D- niski)

  – Stan K (Full-Speed): D- > D+ (np. D- wysoki, D+ niski)

  – SE0 (Single Ended Zero): D+ i D- niskie (poniżej progu).

  – SE1 (Single Ended One): D+ i D- wysokie (nieużywane w normalnej transmisji).

  – Idle (Full-Speed): Stan J.

- Struktura pakietu USB: Dane w USB są przesyłane w postaci pakietów. Każdy pakiet Full-Speed rozpoczyna się od sekwencji synchronizacyjnej (SYNC), po której następuje identyfikator pakietu (PID), opcjonalne pole danych (DATA) i/lub adres (ADR), pole sumy kontrolnej (CRC) oraz znacznik końca pakietu (EOP).

  – SYNC (Synchronizacja): Sekwencja 8 bitów (KJKJKJKK) poprzedzająca każdy pakiet, umożliwiająca odbiornikowi synchronizację zegara.

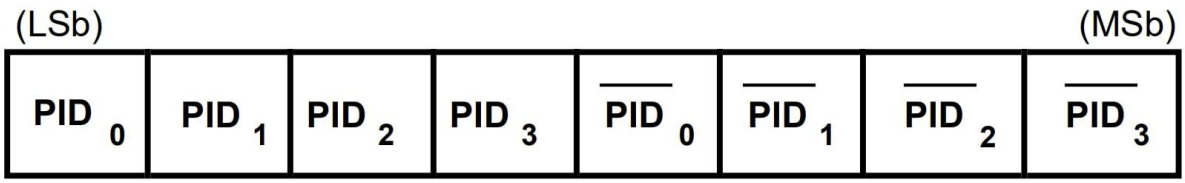
  – PID (Packet Identifier): 8-bitowe pole (4 bity wartości + 4 bity dopełnienia) określające typ pakietu (np. TOKEN, DATA0, DATA1, ACK, NAK, SOF). Jego strukturę przedstawia rysunek 1 poniżej.

  – ADR (Adres): 7-bitowy adres urządzenia i 4-bitowy numer punktu końcowego (end-point) w pakietach typu TOKEN.

  – DATA (Dane): Pole danych o zmiennej długości (0-1023 bajtów dla Full-Speed) w pakietach DATA.

  – CRC (Cyclic Redundancy Check): Suma kontrolna chroniąca pole danych (CRC16) lub pole adresu/PID (CRC5).

  – EOP (End Of Packet): Specyficzna sekwencja sygnalizacyjna (SE0 przez 2 bity, po którym następuje stan J przez 1 bit) oznaczająca koniec pakietu.



Rysunek 1. Struktura bajta PID

2.2. Kodowanie NRZI (Non-Return-to-Zero-Inverted)

USB wykorzystuje kodowanie NRZI do transmisji danych [1]. W tym schemacie:

- Bit ’0’ jest reprezentowany przez zmianę (inwersję) poziomu logicznego na liniach transmisyjnych w stosunku do poprzedniego bitu.
- Bit ’1’ jest reprezentowany przez brak zmiany poziomu logicznego na liniach transmisyjnych.

Dekoder NRZI musi przechowywać poprzedni stan linii, aby poprawnie zinterpretować bieżący bit.

2.3. Bit Stuffing i Unstuffing

Aby zapewnić wystarczającą liczbę przejść sygnału dla utrzymania synchronizacji zegara odbiornika (ponieważ długie sekwencje bitów ’1’ nie powodują zmian w kodowaniu NRZI), standard USB wymaga stosowania techniki bit stuffing [1].

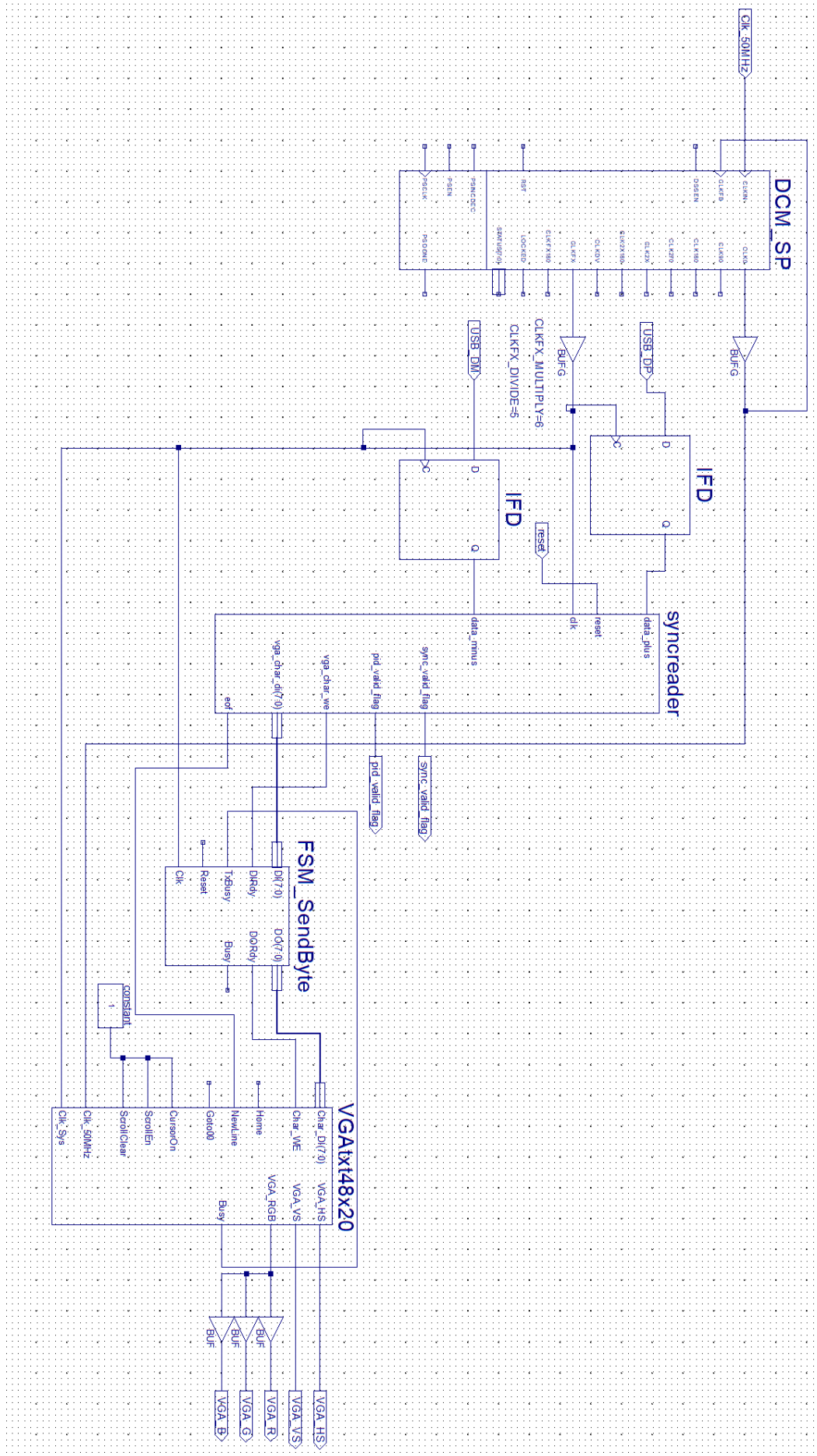
- Bit Stuffing (nadajnik): Po każdej sekwencji sześciu kolejnych bitów ’1’ w strumieniu danych (przed kodowaniem NRZI), nadajnik automatycznie wstawia dodatkowy bit ’0’.
- Bit Unstuffing (odbiornik): Odbiornik, po zdekodowaniu NRZI, musi wykryć sekwencję sześciu kolejnych bitów ’1’, a następnie usunąć następujący po nich bit ’0’ (tzw. stuffed bit).

Mechanizm ten nie dotyczy pola SYNC.

3. Projekt

3.1. Struktura układu

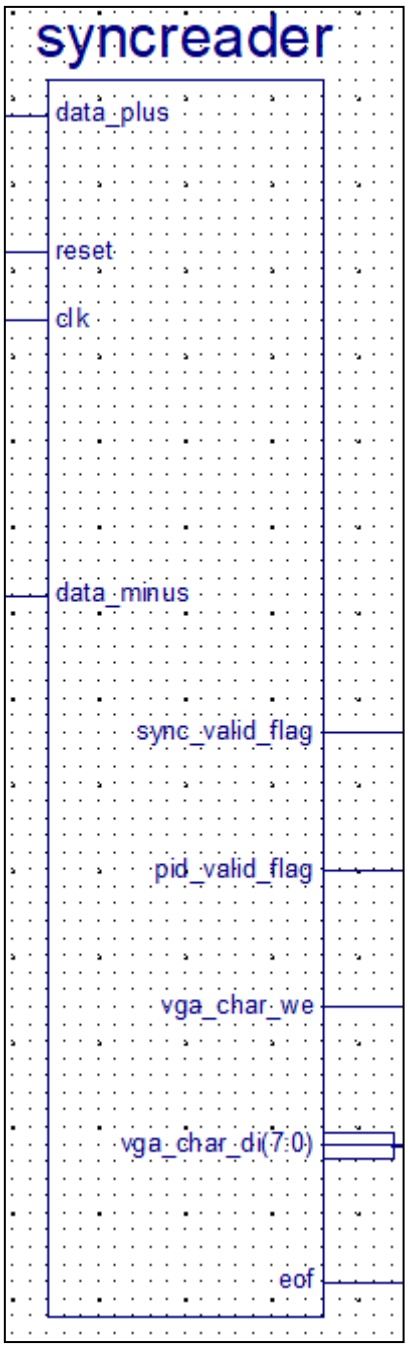
Do realizacji projektu stworzyliśmy moduł, który w połączeniu z gotowymi modułami zewnętrznymi spełniał wszystkie wymagania stawiane w projekcie. Całkowity schemat uwzględniający wykorzystane moduły przedstawia rysunek 2 poniżej.



Rysunek 2. Schemat ogólny projektu

3.2. Opis napisanych modułów

3.2.1. Moduł syncreader



Rysunek 3. Symbol modułu do odczytywania pakietów USB

Porty wejściowe

- data_plus – jednobitowe wejście, odpowiadające za linię D+ w USB
- data_minus – jednobitowe wejście, odpowiadające za linię D- w USB
- clk – wejście zegarowe o niestandardowej częstotliwości 60 MHz
- reset

Porty wyjściowe

- sync_valid_flag – jednobitowe wyjście informujące o powodzeniu odczytu bajta synchronizującego
- pid_valid_flag – jednobitowe wyjście informujące o powodzeniu odczytu bajta PID
- vga_char_di – 8 bitowy wektor, reprezentujący bajt do wyświetlenia przez VGA
- vga_char_we – bit do sygnalizacji momentu, w którym bajt ma być wypisany przez VGA
- eof – bit do sygnalizacji wykrycia końca pakietu

Wszystkie opisane wyżej porty przedstawia rysunek 4.

```
entity syncreader is
    Port (
        -- Inputs
        data_plus  : in  STD_LOGIC; -- linia D+
        data_minus : in  STD_LOGIC; -- linia D-
        reset      : in  STD_LOGIC; -- asynchroniczny reset układu
        clk        : in  STD_LOGIC; -- zegar

        -- Outputs
        sync_valid_flag : out STD_LOGIC := '0'; -- flaga do potwierdzenia poprawności odczytu bajtu synchronizującego
        pid_valid_flag : out STD_LOGIC := '0'; -- flaga do potwierdzenia poprawności odczytu bajtu PID
        vga_char_di    : out unsigned(7 downto 0) := (others => '0'); -- znak do wyświetlenia
        vga_char_we    : out STD_LOGIC := '0'; -- sygnał write enable dla kolejnych modułów
        eof          : out STD_LOGIC := '0' -- sygnał end of packet - do przejścia do nowej linii w VGA
    );
end syncreader;
```

Rysunek 4. Porty modułu syncreader

Ponadto moduł zawiera zestaw sygnałów wewnętrznych oraz zmiennych procesu przedstawionych na rysunkach 5 i 6 poniżej. Zastosowanie zmiennych procesu umożliwia natychmiastową aktualizację kluczowych danych zamiast aktualizować je dopiero przed ponownym rozpoczęciem procesu.

```
  -- Internal signals
  signal shift_reg    : std_logic_vector(7 downto 0) := (others => '1'); -- rejestr do przechowania najnowszych 8 bitów
  signal bit_count    : integer range 0 to 8 := 0;  -- licznik bitow - do okreslenia kiedy odczytano kompletny bajt
  signal sample_count : integer range 0 to 4 := 2;  -- zmienna do ignorowania 4/5 cykli zegara (USB 12MHz - układ FPGA 60MHz)

 signal consecutive_ones : integer range 0 to 6 := 1;  -- licznik kolejnych '1'
  signal skip_next_bit    : std_logic := '0';          -- flaga pomijania bitu stuffing
```

Rysunek 5. Sygnały modułu syncreader

```
process(clk, reset)
  variable eof_first_bit_flag : std_logic := '0'; -- flaga do wykrycia pierwszego bitu sekwencji końca pakietu
  variable decoded_bit : std_logic;
  variable new_shift_reg : std_logic_vector(7 downto 0) := (others => '1'); -- zmienna lokalna (procesu) rejestru
  variable prev_bit : std_logic := '1'; -- ostatnio odczytany bit - potrzebny do dekodowania następnego
  variable v_consecutive_ones : integer range 0 to 6 := 1;                -- licznik kolejnych '1' lokalny - lic
begin
```

Rysunek 6. Zmienne procesu modułu syncreader

Moduł do odczytywania pakietów USB 1.1. Pierwotnie jego celem było wykrycie bajta synchronizującego, a następnie przekazanie odpowiednich informacji do kolejnych modułów, lecz w trakcie wykonywania projektu uznano, że efektywniejszym rozwiązaniem będzie zawarcie całego procesu odczytywania pakietów w pojedynczym module. Główna logika modułu opiera się na maszynie stanów, reprezentowanym przez graf z rysunku 12.

Maszyna składa się z następujących stanów:

- S_SEARCH_SYNC – stan początkowy
- S_READ_PID – stan osiągany po wykryciu sekwencji, odpowiadającej bajtowi synchronizującemu
- S_READ_DATA – stan osiągany po wykryciu poprawnego PID

Moduł rozpoczyna pracę od odczytywania bitów z magistrali D+ USB, co piąty cykl zegara. Jest to spowodowane niestandardową częstotliwością pracy USB wynoszącą 12 MHz w porównaniu do częstotliwości zegara płytki FPGA (50 MHz), przekonwertowanej przed dotarciem tego sygnału do modułu do wartości 60 MHz. Podczas każdego cyklu, w którym moduł odczytuje bit, najpierw go dekoduje (rysunek 7), gdyż pochodzący z magistrali sygnał jest zakodowany metodą NRZI. Zdekodowany bit jest następnie umieszczany na początku 8 bitowego wektora (działającego jako rejestr przesuwny), chyba że bit następuje bezpośrednio po sekwencji sześciu logicznych jedynek (w zdekodowanej postaci), wtedy taki bit zostaje pominięty.

W stanie początkowym S_SEARCH_SYNC (rysunek 8) po każdej aktualizacji rejestru porównywany jest on ze stałą, odpowiadającą poprawnej sekwencji bajta synchronizującego. W przypadku wykrycia tej sekwencji w rejestrze, maszyna stanów przechodzi do stanu S_READ_PID (rysunek 9), gdzie rejestr zostaje zaktualizowany o kolejne 8 bitów z magistrali. Następnie sprawdzana jest poprawność tej sekwencji, czyli czy najstarsze 4 bity tworzą dopełnienie czterech najmłodszych bitów. W przypadku błędnej sekwencji, maszyna stanów powraca do stanu S_SEARCH_SYNC, a w przeciwnym wypadku na wyjście przekazywana jest sekwencja tworząca bajt PID oraz sygnał vga_char_we, a maszyna stanów przechodzi do S_READ_DATA (rysunek 10). W tym stanie moduł odczytuje bajty i przekazuje je na wyjście tak długo aż nie pojawi się sekwencja kończąca pakiet.

Wykrywanie sekwencji kończącej pakiet odbywa się zawsze pod koniec procesu co piąty cykl zegara (rysunek 11). W przypadku wykrycia sekwencji dwóch logicznych ‘0’ na obu liniach (D+ oraz D-) moduł przekazuje na wyjście bit eof oraz maszyna przechodzi do stanu S_SEARCH_SYNC bez względu na to, w którym stanie obecnie się znajduje.

```
elsif rising_edge(clk) then

    -- reset sygnałów przekazywanych do kolejnych modułów krótko po ich przesłaniu
    vga_char_we <= '0';
    eof <= '0';

    if sample_count = 4 then
       sample_count <= 0;

       -- dekodowanie NRZI
       if data_plus = prev_bit then
          decoded_bit := '1';
       else
          decoded_bit := '0';
       end if;
       prev_bit := data_plus;
```

Rysunek 7. Dekodowanie bitów

```
case current_state is
      -- wykrycie bajtu synchronizującego
     when S_SEARCH_SYNC =>
            new_shift_reg := shift_reg(6 downto 0) & decoded_bit;
            shift_reg <= new_shift_reg;
            -- wykrycie sekwencji
            if new_shift_reg = SYNC_PATTERN then
                current_state <= S_READ_PID;
                bit_count <= 0;
                sync_valid_flag <= '1';
            else
                current_state <= S_SEARCH_SYNC;
            end if;
```

Rysunek 8. Stan szukania sekwencji bajta synchronizującego (S_SEARCH_SYNC)

```
new_shift_reg := shift_reg(6 downto 0) & decoded_bit;
shift_reg <= new_shift_reg;

if bit_count = 7 then
   bit_count <= 0;
      -- wykrycie sekwencji
      if new_shift_reg(7 downto 4) = not new_shift_reg(3 downto 0) then
         pid_valid_flag <= '1';

         vga_char_di <= unsigned(new_shift_reg);

         vga_char_we <= '1';
         current_state <= S_READ_DATA;
      else
         current_state <= S_SEARCH_SYNC;
      end if;
else
   bit_count <= bit_count + 1;
   current_state <= S_READ_PID;
end if;
```

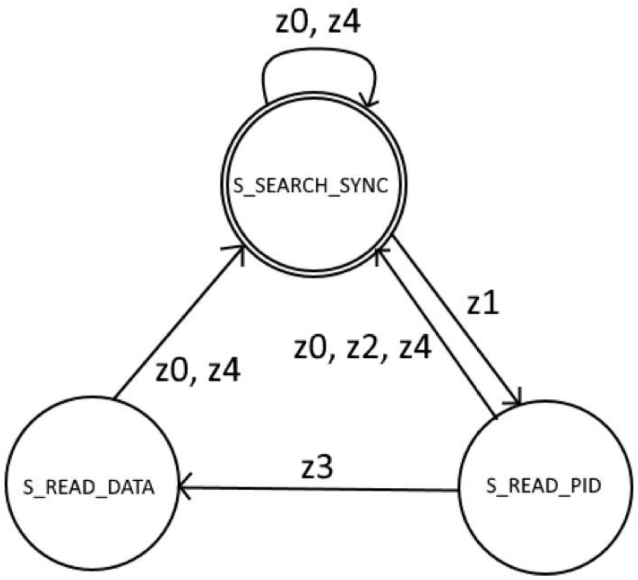
Rysunek 9. Stan szukania poprawnej sekwencji PID (S_READ_PID)

```
-- odczyt reszty pakietu
when S_READ_DATA =>
   if skip_next_bit = '1' then
      skip_next_bit <= '0';  -- pominecie bitu - unstuffing
   else
      v_consecutive_ones := consecutive_ones;
      if decoded_bit = '1' then
         v_consecutive_ones := v_consecutive_ones + 1;
      else
         v_consecutive_ones := 0;
      end if;

      -- ustawienie flagi do pominiecia kolejnego bitu - po wystapieniu 6 kolejnych '1' następny nalezy zignorować
      if v_consecutive_ones = 6 then
         skip_next_bit <= '1';
         consecutive_ones <= 0;

         new_shift_reg := shift_reg(6 downto 0) & decoded_bit;
         shift_reg <= new_shift_reg;
         bit_count <= bit_count + 1;
      else
         new_shift_reg := shift_reg(6 downto 0) & decoded_bit;
         shift_reg <= new_shift_reg;
         consecutive_ones <= v_consecutive_ones;
         bit_count <= bit_count + 1;
      end if;

      -- przekazanie nowego bajtu na wyjscie
      if bit_count = 7 then
         vga_char_di <= unsigned(new_shift_reg);
         vga_char_we <= '1';
         bit_count <= 0;
      end if;
   end if;
   current_state <= S_READ_DATA;
```

Rysunek 10. Stan odczytu danych (S_READ_DATA)

```
  -- wykrycie sekwencji kończącej pakiet 2 bity '0' na obu liniach D+ i D-
  if data_plus = '0' and data_minus = '0' then
       if eof_first_bit_flag = '1' then
          -- w przypadku wykryciu '0' na obu liniach 2 razy pod rząd reset zmiennych mnodułu
          new_shift_reg := "11111111";
          shift_reg <= "11111111";
          eof <= '1';
          current_state <= S_SEARCH_SYNC;
          eof_first_bit_flag := '0';

          v_consecutive_ones := 1;
          consecutive_ones <= 1;
          skip_next_bit <= '0';
          prev_bit := '1';
       else
          eof_first_bit_flag := '1';
       end if;
    else
       eof_first_bit_flag := '0';
 end if;
```

Rysunek 11. Wykrywanie sekwencji końca pakietu



Rysunek 12. Maszyna stanów modułu syncreader

Przejścia przedstawione na rysunku 12 można opisać w następujący sposób

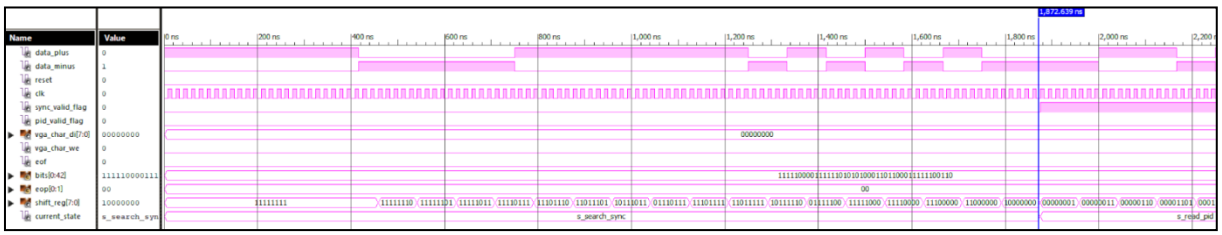
- Z0 – RESET
- Z1 – REGISTER[7:0] == SYNC_PATTERN
- Z2 – BIT COUNT == 7 && (REGISTER[7:4] != NOT REGISTER[3:0])
- Z3 – BIT COUNT == 7 && (REGISTER[7:4] == NOT REGISTER[3:0])
- Z4 – END OF PACKET

3.2.2. Symulacja

W celu weryfikacji sprawności naszego modułu przeprowadzone zostały przykładowe symulacje dla kilku zestawów danych. W pierwszej kolejności zostało sprawdzone działanie modułu dla poprawnych danych z rysunku 13.

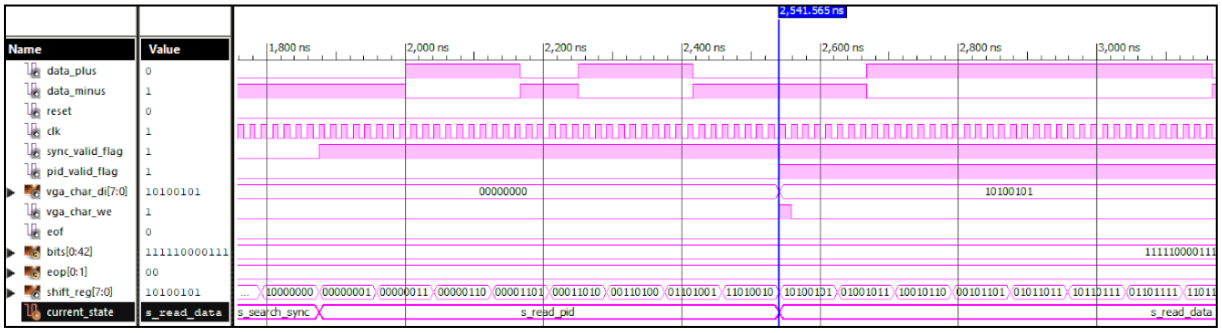
```
constant Bits : std_logic_vector  :=
B"1111100001111111" & -- stan idle
 "01010100" &  -- poprawny bajt synchronizujacy
 "01101100" &  -- poprawny bajt PID
 "01111110" &  -- kolejne bity danych
 "0110";       -- 1 kompletny i 1 częściowy bajt danych
constant EOP : std_logic_vector := B"00"; -- sekwencja końca pakietu
```

Rysunek 13. Dane do pierwszego testu



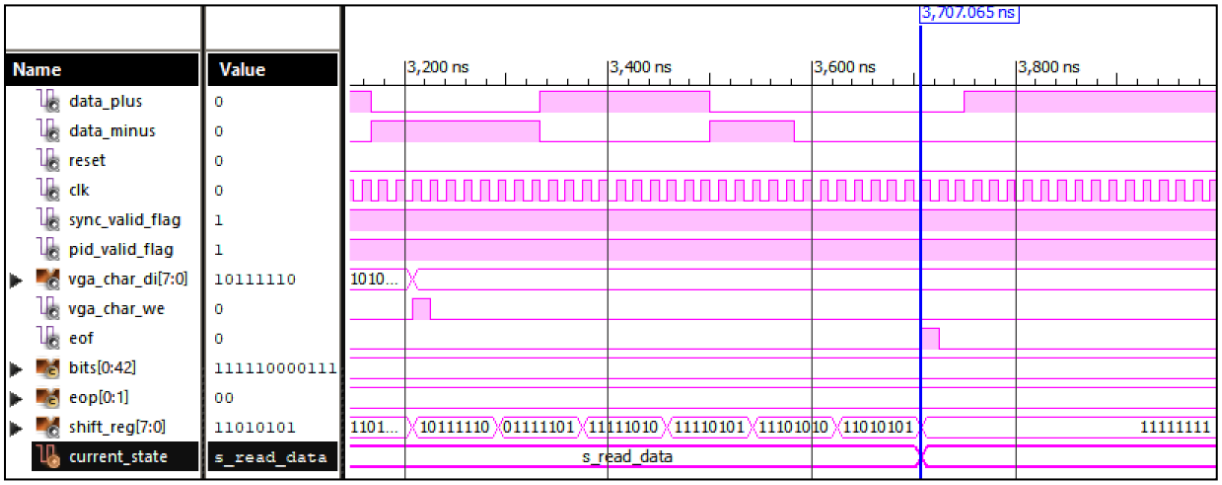
Rysunek 14. Początek symulacji – wykrycie początku pakietu

Na rysunku 14 powyżej, prezentującym początkowy etap symulacji, zauważyć można w momencie oznaczonym na niebiesko ustawienie portu sync_valid_flag (piąty wiersz), sygnalizującego poprawność wykrycia bajta synchronizującego (01010100 w NRZI) oraz przejście maszyny stanów do kolejnego stanu (ostatni wiersz). Dzieje się tak nawet, jeżeli stan IDLE nie składa się wyłącznie z samych '1', co widać w pierwszej części rysunku.



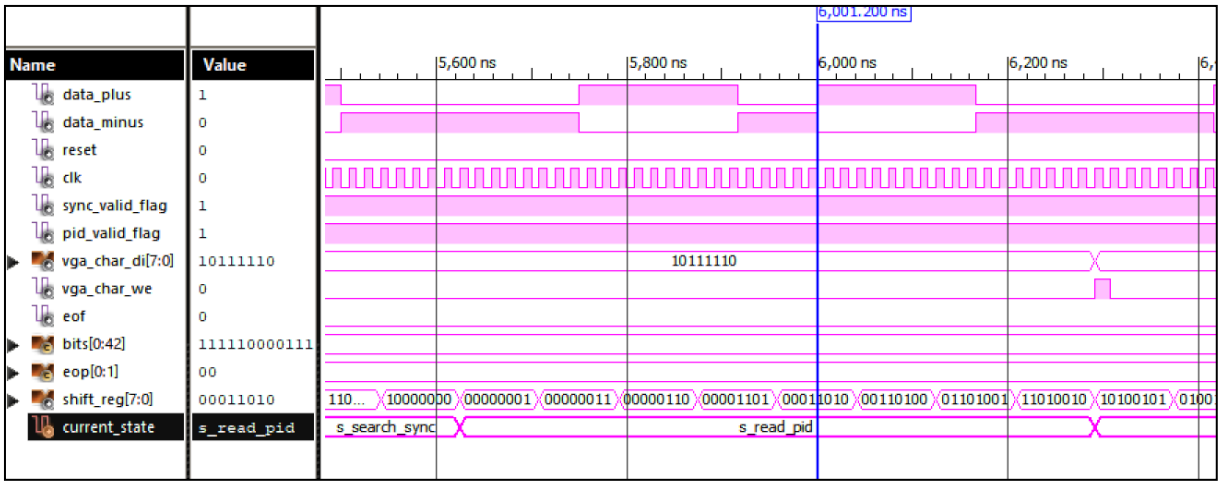
Rysunek 15. Wykrycie poprawnego PID

Na rysunku 15, będącym kontynuacją tej samej symulacji, w momencie oznaczonym na niebiesko ustawiany jest port pid_valid_flag (szósty wiersz), sygnalizujący wykrycie poprawnego bajta PID, czyli pierwsze 4 bity stanowią dopełnienie pozostałych czterech bitów, co zauważyć można po wartości Shift_reg w następnym cyklu zegara (1010 0101 w przedostatnim wierszu) oraz następuje przejście maszyny stanów do kolejnego stanu (ostatni wiersz). Dodatkowo ustawiana jest nowa wartość znaku do wyświetlenia (siódmy wiersz) oraz flaga sygnalizująca kolejnym modułom gotowość do wyświetlania znaku (ósmy wiersz).



Rysunek 16. Wykrycie końca pakietu

W końcowym etapie symulacji z rysunku 16 wykrycie sekwencji końca pakietu ('00' zarówno na linii D+ jak i D-) powoduje powrót do stanu szukania bajta synchronizującego następnego pakietu, ignorując przy tym aktualny stan odczytywania kolejnego bajta danych oraz zwracając na wyjście bit eof (dziewiąty wiersz) jako informacja dla modułu VGA o przygotowanie nowej linii dla nowego pakietu.



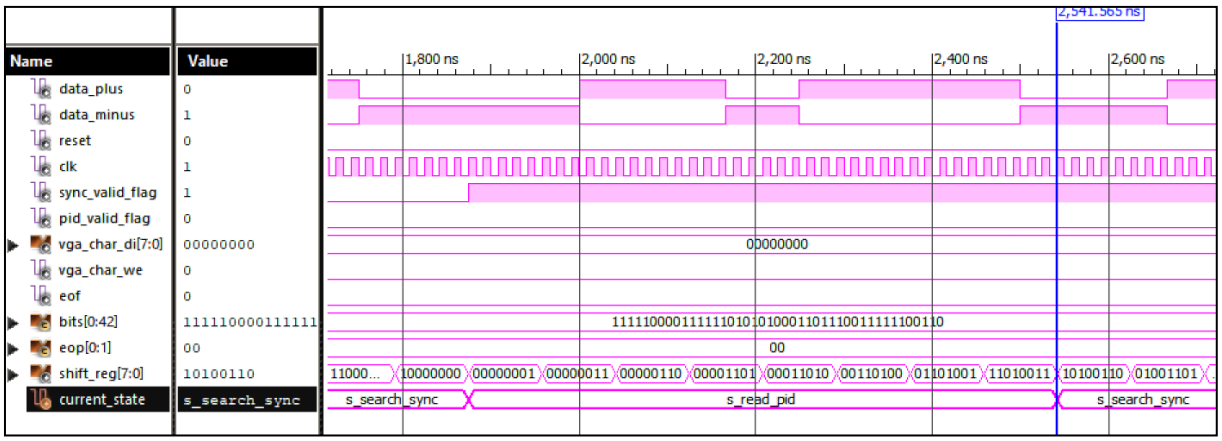
Rysunek 17. Odczyt bajtów SYNC oraz PID dla następnego pakietu

Zgodnie z oczekiwaniami, po powrocie do stanu szukania bajta synchronizującego moduł ponownie wykrywa poprawne sekwencje bajta synchronizującego oraz PID, co widać powyżej na rysunku 17 (ostatni wiersz).

Następnie sprawdzone zostało działanie dla błędnego PID z rysunku 18. Podmieniony bit został oznaczony na szaro.

```
constant Bits : std_logic_vector  :=
B"111110000111111" & -- stan idle
 "01010100" &   -- poprawny bajt synchronizujacy
 "01101110" &   -- poprawny bajt PID
 "01111110" &   -- kolejne bity danych
 "0110";        -- 1 kompletny i 1 częściowy bajt danych
constant EOP : std_logic_vector := B"00"; -- sekwencja końca pakietu
```

Rysunek 18. Dane do drugiego testu



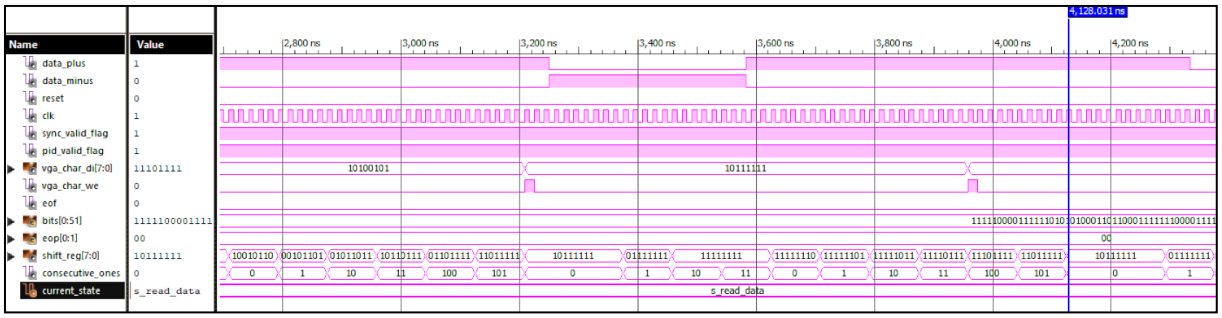
Rysunek 19. Wykrycie błędnego PID

Zgodnie z oczekiwaniami maszyna stanów powraca do szukania nowego pakietu, jeśli PID aktualnego okaże się błędne, co przedstawia powyżej rysunek 19.

Dodatkowo sprawdzone zostało działanie mechanizmu bit unstuffing dla podanego na rysunku 20 zestawu danych. Na szaro oznaczono zmodyfikowane bity.

```
constant Bits : std_logic_vector  :=
B"1111100000111111" & -- stan idle
 "01010100" &  -- poprawny bajt synchronizujacy
 "01101100" &  -- poprawny bajt PID
 "01111111" &  -- kolejne bity danych
 "000011111" & -- kompletny bajt (pierwszy bit jest nadmiarowy)
 "1111"; -- trzeci bit jest nadmiarowy
constant EOP : std_logic_vector := B"00"; -- sekwencja końca pakietu
```

Rysunek 20. Dane do trzeciego testu



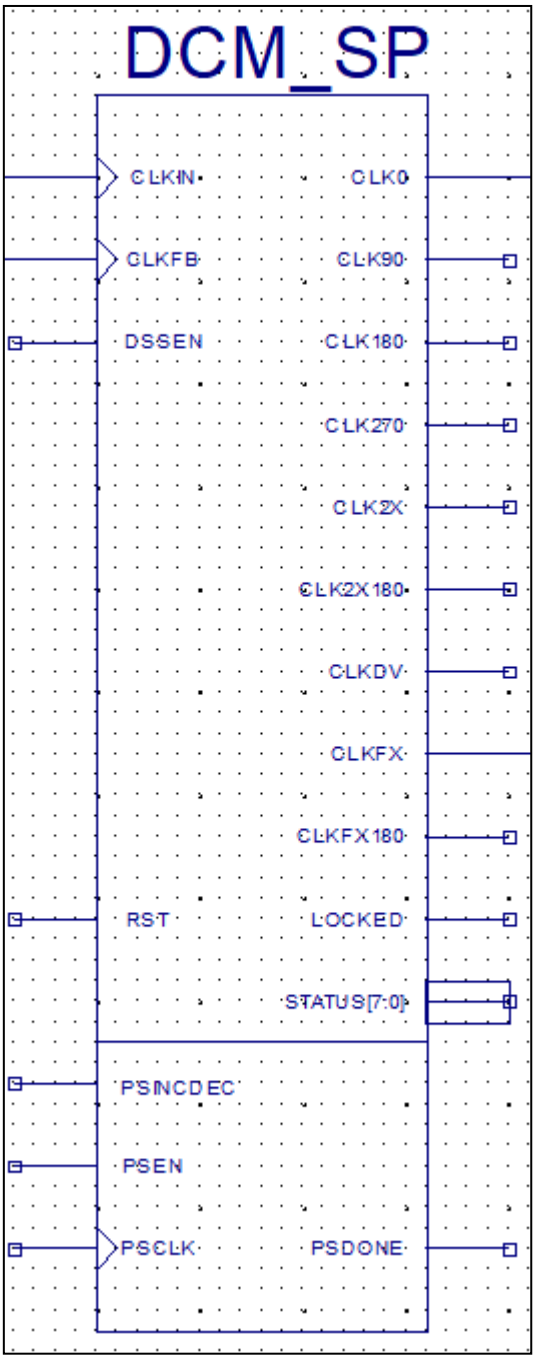
Rysunek 21. Usuwanie bitów nadmiarowych

Na fragmencie powyższej symulacji z rysunku 21 z odczytywaniem kolejnych bajtów po PID zauważyć można, że w momentach około 3.212 µs oraz 4.218 µs pomijane są bity nadmiarowe, czyli takie które występują bezpośrednio po ciągu sześciu logicznych ‘1’ (w zdekodowanej formie), co pokrywa się z przewidywaniami. W przedostatnim wierszu widoczne jest potwierdzenie poprawnego zliczania ciągów następujących po sobie ‘1’.

3.3. Opis wykorzystanych gotowych modułów

3.3.1. DCM_SP

Moduł służący do syntezy częstotliwości zegara. W naszym projekcie jest potrzebny do konwersji częstotliwości sygnału zegara w płytce FPGA z 50 MHz do wielokrotności częstotliwości pracy USB 1.1 12 MHz (60 MHz). Lista wejść oraz wyjść dla tego modułu znajduje się pod adresem [4]. Do wejść CLKIN oraz CLKFB doprowadzany jest standardowy sygnał zegarowy 50 MHz, na wyjściach CLK0 również 50 MHz potrzebny do modułu VGA, natomiast z CLKFX wychodzi zsyntezowany sygnał do wartości 60 MHz. Sygnał ten jest wykorzystywany we wszystkich pozostałych modułach.



Rysunek 22. Symbol modułu do syntezy częstotliwości zegara

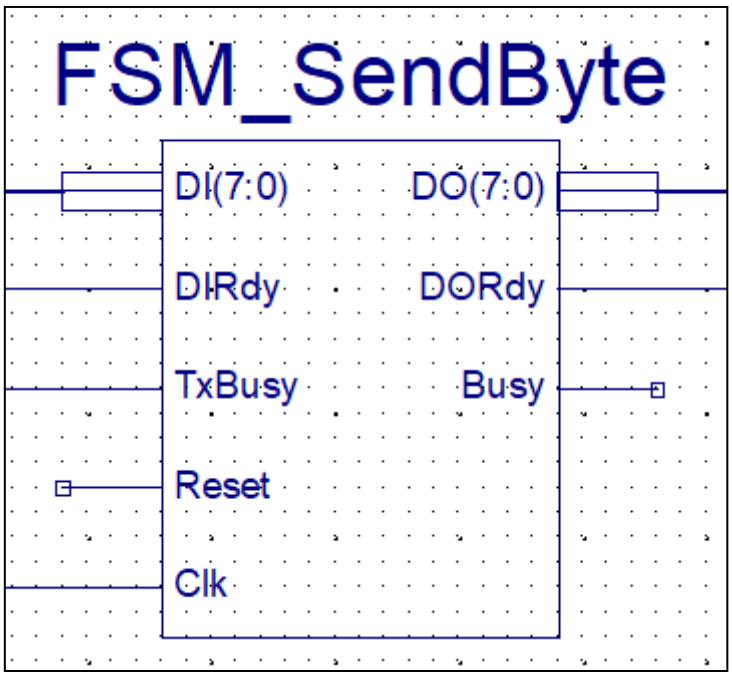
3.3.2. FSM_SendByte

Moduł służący do konwersji wektora 8-bitowego do postaci dwóch cyfr szesnastkowych. Na wejściu moduł przyjmuje:

- sygnał zegarowy 60 MHz
- bit TxBusy z modułu VGA
- bit DIRdy sygnalizujący moment odczytu
- wektor 8-bitowy DI będący wektorem do konwersji

Na wyjściu moduł zwraca:

- bit DORdy sygnalizujący modułowi VGA moment wyświetlenia znaku
- wektor 8-bitowy DO będący przekonwertowanym znakiem do wyświetlenia



Rysunek 23. Symbol modułu do konwersji 8-bitowego wektora na dwie cyfry szesnastkowe

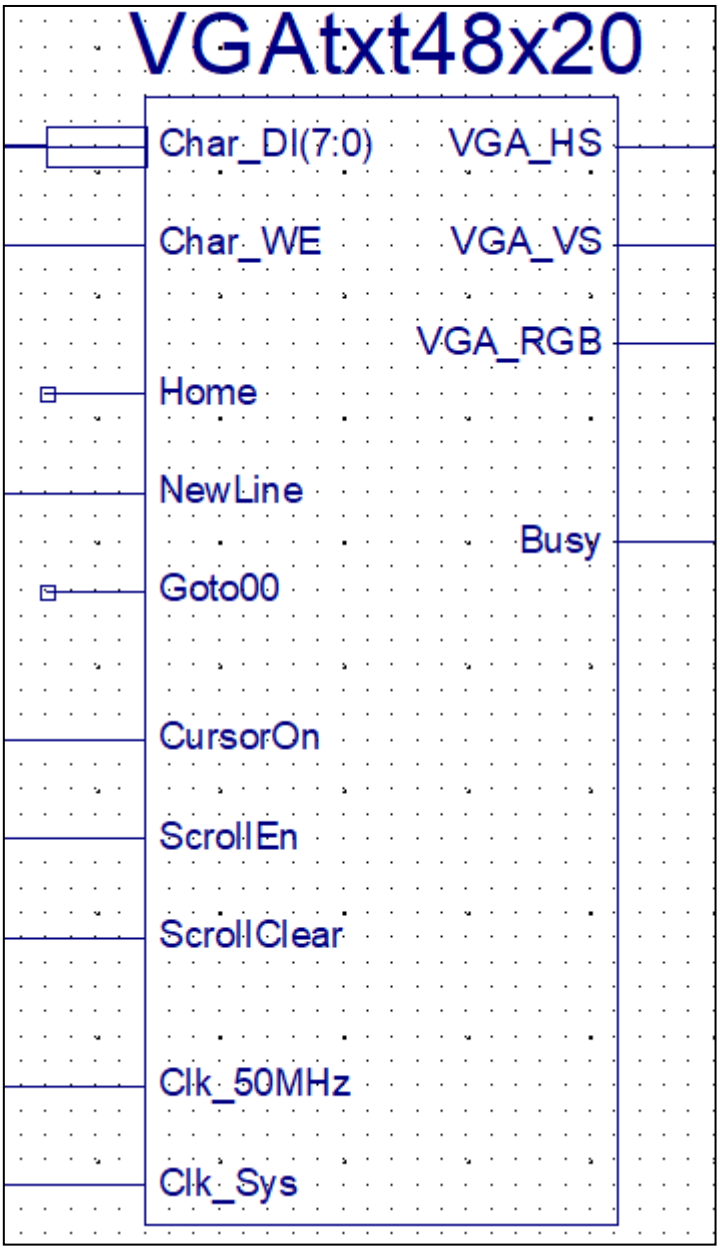
3.3.3. VGAtxt48x20

Moduł pozwalający na wyświetlanie znaków na terminalu znakowym. Zawiera w sobie podstawowe funkcje przenoszenia kursora oraz przewijania ekranu. Lista wejść oraz wyjść znajduję się pod adresem [4]. Wejścia modułu w naszym projekcie:

- Char_Di znak do wyświetlenia, pochodzący z modułu FSM_SendByte
- Char_WE sygnalizujący moment wypisania znaku
- NewLine do utworzenia nowej linii w momencie wykrycia końca pakietu w module syncreader
- CursorOn, ScrollEn oraz ScrollClear podłączone do stałej ‘1’
- wejścia zegarowe Clk_50MHz z sygnałem zegarowym 50 MHz oraz Clk_Sys z zsyntezowanym sygnałem zegarowym o wartości 60 MHz

Wyjścia modułu w naszym projekcie:

- bit Busy do wstrzymania wyświetlania nowych znaków na czas 48 taktów, gdy czyszczona jest nowa linia
- VGA_HX, VGA_VS, VGA_RGB do wyświetlenia znaków



Rysunek 24. Symbol modułu do wyświetlania cyfr

4. Implementacja

4.1. Informacje szczegółowe

Po zaimplementowaniu układu otrzymaliśmy poniższe wyniki:

- zajętość plastrów: 169 na 4656 dostępnych
- zajętość LUT: 295 na 9312 dostępnych
- maksymalna szybkość pracy: 10.063ns (Fz = 99.37MHz)

Wyniki te można również zauważyć na rysunkach 25 i 26 poniżej

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	145	9,312	1%	
Number of 4 input LUTs	251	9,312	2%	
Number of occupied Slices	169	4,656	3%	
Number of Slices containing only related logic	169	169	100%	
Number of Slices containing unrelated logic	0	169	0%	
Total Number of 4 input LUTs	295	9,312	3%	
Number used as logic	248			
Number used as a route-thru	44			
Number used as Shift registers	3			
Number of bonded IOBs	11	232	4%	
IOB Flip Flops	2			
Number of RAMB16s	2	20	10%	
Number of BUFGMUXs	2	24	8%	
Number of DCMs	1	4	25%	
Average Fanout of Non-Clock Nets	3.51			

Rysunek 25. Fragment raportu z Xilinx ISE

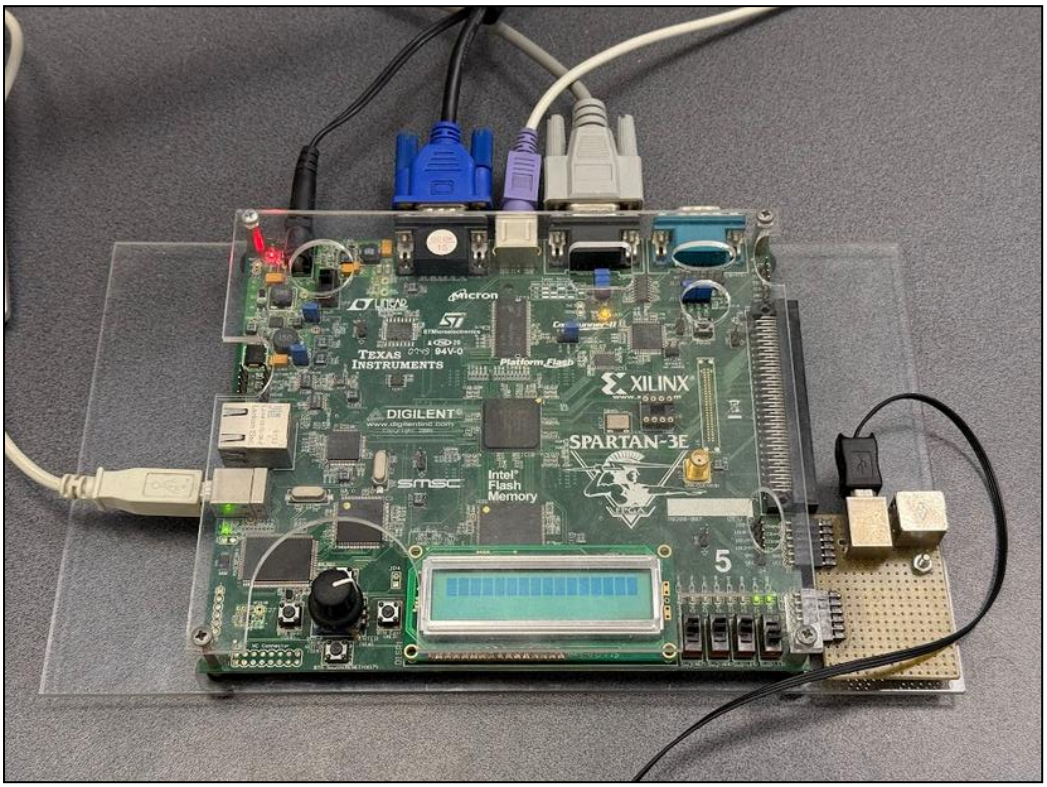
	Met	Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
1	Yes	PERIOD analysis for net "XLXN_8" derived from NET "Clk_50MHz_IBUFG" PERIOD = 20 ns HIGH 50%	SETUP HOLD	6.603ns 1.117ns	10.063ns	0 0	0 0
2	Yes	PERIOD analysis for net "XLXN_7" derived from NET "Clk_50MHz_IBUFG" PERIOD = 20 ns HIGH 50%	SETUP HOLD	10.908ns 1.001ns	9.092ns	0 0	0 0
3	Yes	NET "Clk_50MHz_IBUFG" PERIOD = 20 ns HIGH 50%	MINLOWP...	14.000ns	6.000ns	0	0
4	Yes	PATH "TS_12_path" TIG	SETUP		7.250ns		0
5	Yes	PATH "TS_13_path" TIG					

Rysunek 26. Raport Timing Constraints

Tak niska zajętość układów, w stosunku do całkowitej liczby dostępnych zasobów, pozwala na swobodne rozszerzanie układu o kolejne funkcjonalności. Maksymalna szybkość pracy mieści się w oczekiwanych limitach ze znacznym zapasem, więc układ powinien działać bez żadnych opóźnień lub błędów związanych z desynchronizującymi się modułami.

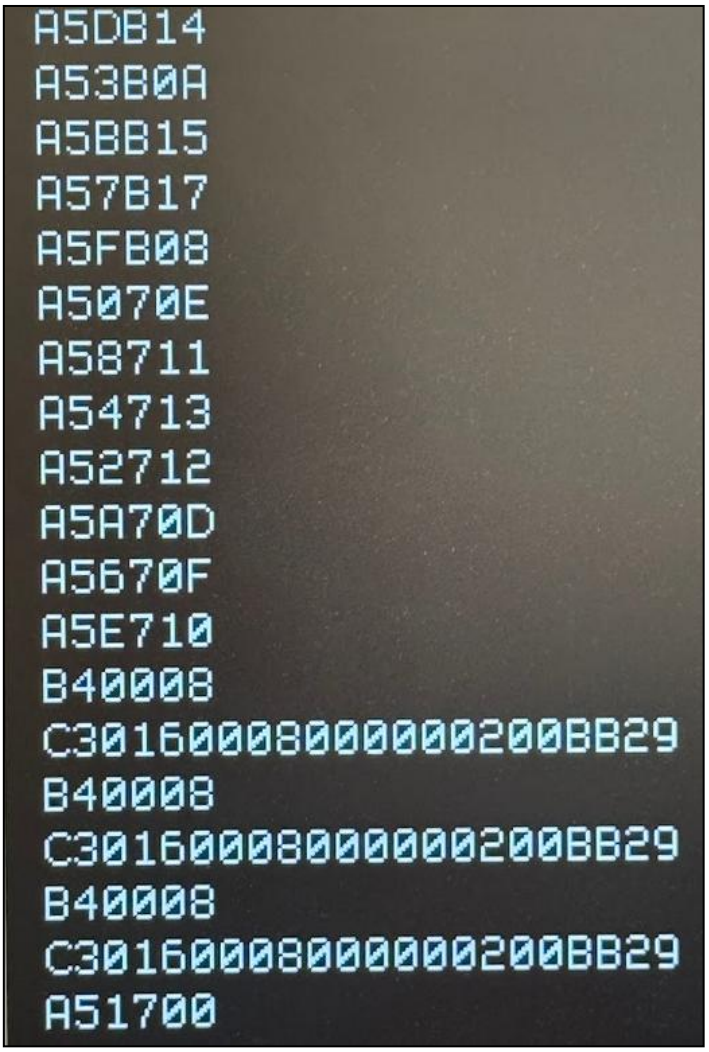
4.2. Uzytkowanie modułu

Aby zaobserwować działanie modułu syncreader należy przygotować monitor podłączony przez VGA do płytki FPGA oraz na płytkę FPGA musi być wgrany poprzez iMpact plik z rozszerzeniem .bit wygenerowany wcześniej w Xilinx ISE. Po wgraniu pliku należy podłączyć kabel USB (czarny po prawej na rysunku) do płytki tak jak przedstawia rysunek 27 poniżej. Kabel powinien uprzednio być podłączony do komputera drugim końcem.



Rysunek 27. Płytka FPGA z podłączonym USB (po prawej)

Po podłączeniu powinny się zaświecić dwie prawe diody tak jak na powyższym rysunku w jego prawej dolnej części (prawa sygnalizuje wykrycie sekwencji bajta synchronizującego a lewa wykrycie poprawnej sekwencji PID), po czym przez około dwie sekundy będą się wyświetlać odczytane pakiety w tempie około 1000 na sekundę, a następnie odczyt zostanie przerwany. Wtedy monitor powinien wyświetlać kilkanaście ostatnich linii podobnych do tych widocznych na rysunku 28 poniżej.



Rysunek 28. Przykładowy wynik wyświetlany na monitorze, po podłączeniu USB do płytki

Powyższe linie przedstawiają różne pakiety przechwycone z linii USB. Pierwsze 2 znaki to pole PID identyfikujące rodzaj pakietu. Najwięcej jest pakietów A5, czyli typu Start of Frame. Każdy taki pakiet posiada swój numer ramki, który obejmuje pierwsze 11 bitów po bajcie PID, czyli (pomijając pierwsze 2 znaki) 2 i ¾ trzeciego znaku. Pozostałe bity stanowi CRC pakietu. Poprawnie zdekodowany pakiet Start of Frame powinien posiadać numer ramki o 1 większy od poprzedniego pakietu, przy czym najmłodszą cyfrą jest tu najbardziej znaczący bit, a najstarszą cyfrą jest najmniej znaczący bit. Poniżej przedstawiony został przykład wizualny (za pomocą systemu binarnego) numeracji kolejnych pakietów na podstawie kilku pierwszych pakietów z rysunku 28 powyżej. Kolorem szarym oznaczone zostało pole numeru ramki oraz kolorem białym bity, które ulegają zmianie w porównaniu z poprzednim pakietem.

- DB14 – 1101 1011 0001 0100
- 3B0A – 0011 1011 0000 1010
- BB15 – 1011 1011 0001 0101
- 7B17 – 0111 1011 0001 0111
- FB08 – 1111 1011 0000 1000
- 070E – 0000 0111 0000 1110

5. Podsumowanie

5.1. Ocena pracy

W ramach projektu udało nam się zrealizować wszystkie postawione wymagania. Wymagania te obejmowały następujące funkcjonalności:

- dostrojenie częstotliwości układu FPGA (domyślnie 50 MHz) do częstotliwości USB (12 MHz)
- odczytywanie bitów oraz ich dekodowanie z NRZI
- wykrywanie sekwencji bitowych oznaczających początek pakietu i jego koniec oraz określających poprawność jego identyfikatora (PID)
- pozbywanie się bitów nadmiarowych (bit unstuffing)
- przekazywanie gotowych bajtów do zewnętrznego modułu, który przetwarza je do postaci 2 cyfr szesnastkowych
- wyświetlenie tych cyfr przez VGA

Jedyną wadą naszego projektu jest to, że w obecnej wersji działa on poprawnie tylko dla pakietów Start of Frame, w których liczony jest numer ramki. Spowodowane jest to sprawdzaniem nadmiarowości tylko wewnątrz stanu S_READ_DATA, przez co ignorowane są dane z poprzedniego stanu, co może mieć znaczenie, gdy bajt PID kończy się ciągiem jednej lub więcej logicznych ‘1’.

5.2. Możliwe kierunki rozwoju

W pierwszej kolejności należałoby zoptymalizować mechanizm bit unstuffing do takiej postaci, aby działał on poprawnie również dla pozostałych rodzajów pakietów. W tym celu wystarczyłoby powtórzyć mechanizm sprawdzania nadmiarowości bitów wewnątrz stanu S_READ_PID, a zmienną do liczenia długości ciągów jedynek zainicjować wartością 0 zamiast 1.

Poza tym z uwagi na bardzo małą zajętość projektu względem całkowitej liczby dostępnych zasobów, projekt można rozwijać o wiele różnych funkcjonalności. Przykładowo warto byłoby rozwinąć projekt o lepszą wizualizację zawartości pakietów. W obecnej formie sprawdzanie numeru ramki w pakietach Start of Frame jest mało intuicyjne z powodu 11 bitowego pola, które nie pasuje do 4 bitowych wektorów, których reprezentacje są wyświetlane przez VGA oraz odwrotnego liczenia ramek, gdzie najstarszy bit zachowuje się jak najmłodszy bit, a najmłodszy bit zachowuje się jako najstarszy bit. Dodatkowo warto byłoby również dodać opcję do sprawdzenia poprawności kodu CRC każdego pakietu, tak aby móc pominąć pakiety z zakłóconą informacją, czyli gdzie kod CRC nie odpowiada reszcie pakietu.

6. Literatura

[1] Universal Serial Bus Specification, Revision 1.1, September 23, 1998.
http://esd.cs.ucr.edu/webres/usb11.pdf

[2] Xilinx, Spartan-3E FPGA Family Data Sheet (DS312), December 14, 2018.
https://docs.amd.com/v/u/en-US/ds312

[3] Xilinx, Spartan-3E FPGA Starter Kit Board User Guide (UG230), January 20, 2011.
https://docs.amd.com/v/u/en-US/ug230

[4] Jarosław Sugier, Strona płyty Spartan-3E Starter (FPGA)
https://indyk.ict.pwr.wroc.pl/ucyfr/fpga/