POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Informatyka
SPECJALNOŚĆ: Inżynieria Internetowa

PRACA DYPLOMOWA INŻYNIERSKA

Realizacja w układzie FPGA uniwersalnego sterownika magistrali I^2C

FPGA implementation of a universal host controller for the I^2C bus

AUTOR:
Wojciech Adles

PROWADZĄCY PRACĘ:
dr inż. Jarosław Sugier

OCENA PRACY:

WROCŁAW 2017

Spis treści

1. Wstęp

1.1 Cel i zakres pracy

Celem zrealizowanego projektu inżynierskiego była implementacja uniwersalnego sterownika magistrali I^2C w układzie FPGA (*ang. Field-Programmable Gate Array)*, wykorzystując do tego celu język VHDL. Autor zdecydował się na wybór danego tematu z uwagi na zainteresowanie szeroko pojętą logiką cyfrową i jej praktycznym zastosowaniem w projektowaniu układów cyfrowych. Wykorzystując zdobytą podczas odbytych studiów wiedzę, realizacja niniejszej pracy dyplomowej pozwoliła na dogłębniejsze zrozumienie pracy układów programowalnych i problemów jakimi jest obarczona fizyczna realizacja projektów, napisanych z wykorzystaniem języków opisu sprzętu. Obecnie najszerzej używanymi językami HDL (*ang. Hardware Description Language*) są VHDL i Verilog. Autor zdecydował się na wykorzystanie pierwszego z nich, z uwagi na większą wiedzę i doświadczenie wynikające z częstszego używania języka VHDL podczas toku studiów.

Zakres pracy nad projektem obejmował kolejno:

- Poznanie zasady działania magistrali I^2C w warstwie fizycznej i zastosowanego dla niej protokołu wymiany danych.
- Opracowanie ogólnego zarysu projektu, podziału go na moduły i ich wzajemne skomunikowanie.
- Implementacja projektu w języku VHDL i realizacja modułów za pomocą zsynchronizowanych ze sobą procesów.
- Przeprowadzenie testów napisanego kodu z wykorzystaniem narzędzi symulacyjnych, weryfikując różne scenariusze testowe.
- Weryfikacja sprzętowa poprawności działania zsyntezowanego sterownika na fizycznym układzie FPGA i urządzenia podłączonego do magistrali.
- Analiza otrzymanych rezultatów implementacji, wykorzystania zasobów, uzyskane prędkości pracy i opóźnienia powstałe na układzie.

Głównym założeniem projektu była obsługa transmisji dwukierunkowej jako master (urządzenie inicjujące i synchronizujące transmisję) z wykorzystaniem kolejki FIFO (*ang. First In, First Out)* buforującej wysyłane/odbierane dane. Sterownik realizuje przesył danych z prędkością 400 kb/s (tryb Fast-mode) i adresowanie 7-bitowe. Działać ma on dla wariantu single-master, gdzie do magistrali podłączone jest tylko jedno urządzenie nadrzędne. Nie przewiduje się obsługi funkcjonalności clock stretching, która umożliwia podtrzymanie

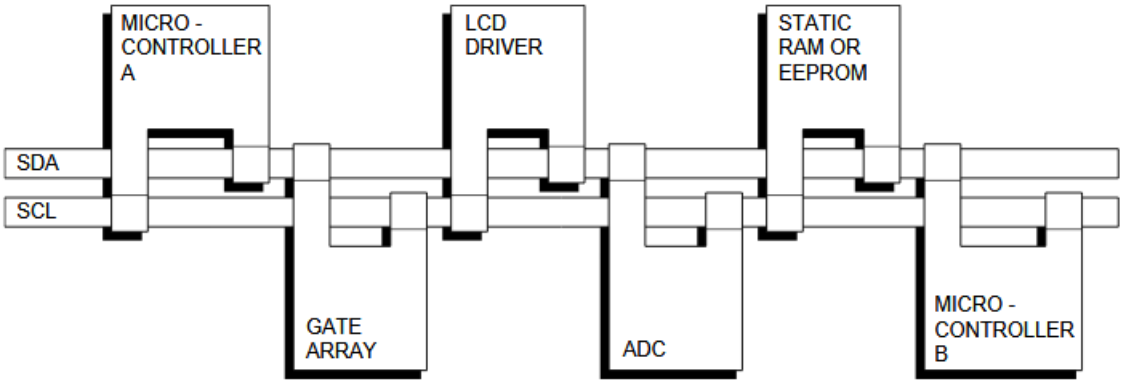
transmisji przez urządzenie podrzędne, gdy nie jest ono gotowe przyjąć kolejne dane. Docelowo sterownik będzie zsyntezowany i zaimplementowany na fizycznym układzie FPGA, gdzie zostanie przetestowany komunikując się z wybranym urządzeniem za pomocą magistrali I^2C. Wymagane zatem było stworzenie osobnego modułu, realizującego logikę komunikacji z danym urządzeniem docelowym, jak również umożliwienie obserwacji poprawności otrzymanych wyników, potwierdzających poprawne działanie sterownika.

1.2 Wykorzystany sprzęt

Do realizacji projektu została wykorzystana płyta Spartan-3E Starter Kit z układem FPGA Xilinx XC3S500E, na który syntezowany jest kod sterownika. Głównym źródłem zegarowym płyty jest 50MHz oscylator[1], który został wykorzystany jako źródło taktowania opisywanego projektu. Jako urządzenie podpięte do magistrali I^2C na którym sterownik był sprzętowo testowany, użyty został 3-osiowy akcelerometr ADXL345. Posiada on szereg rejestrów za pomocą których można sterować pracą urządzenia[2], a które posłużyły do weryfikacji poprawności zrealizowanego przez autora sterownika.

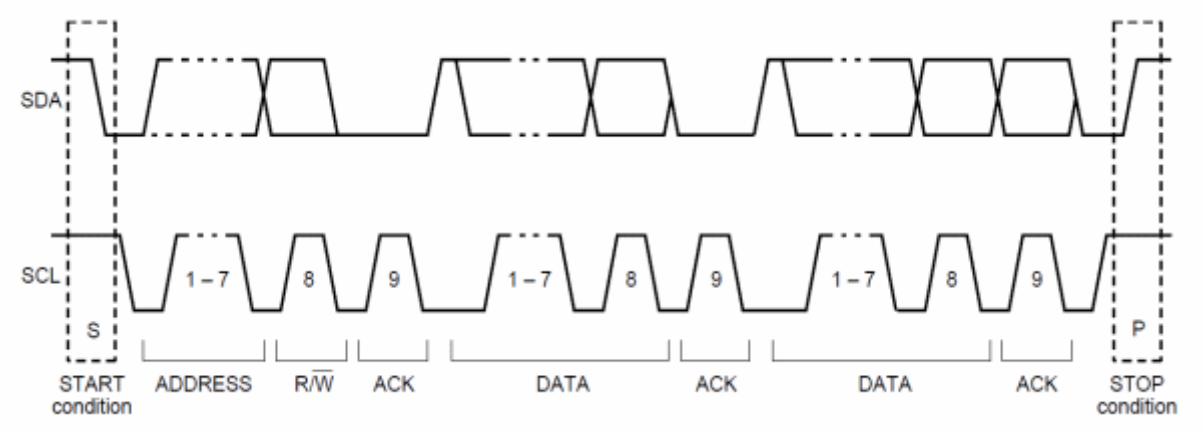
1.3 Zagadnienia teoretyczne

Magistrala I^2C jest magistralą szeregową typu half-duplex (w danym czasie możliwa jest komunikacja tylko w jedną stronę). Transmisja odbywa się za pomocą dwóch linii tj. SDA czyli linii danych oraz SCL, która jest linią zegarową. Urządzenia korzystające z magistrali są podzielone na nadrzędne (*ang. master*) i podrzędne (*ang. slave*), gdzie urządzenia typu master są odpowiedzialne za inicjację i synchronizację przesyłu danych. Jako, że magistrala umożliwia podłączenie do niej wielu urządzeń nadrzędnych, stosowany jest arbitraż mający na celu wykluczenie sytuacji w której wiele urządzeń próbuje skorzystać z niej jednocześnie, jak i synchronizacja zegara na linii SCL pomiędzy urządzeniami nadrzędnymi (Rysunek 1). Implementowany sterownik przewidziany jest dla wariantu pojedynczego urządzenia typu master, dlatego opis tych funkcjonalności został pominięty. Szybkość transmisji reguluje kilka standardów, gdzie zaczynając od najstarszego z nich wynosi ona 100 kb/s w trybie Standard, aż do 3,4 Mb/s w trybie High-speed[3].



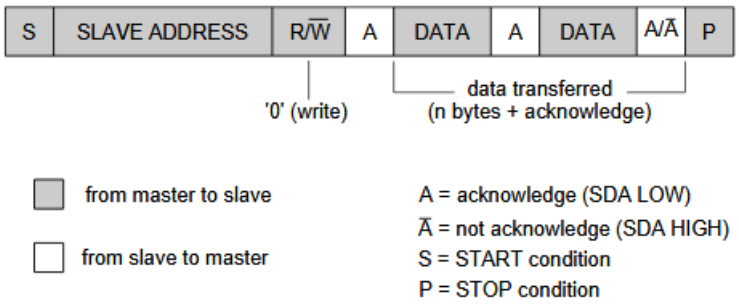
Rysunek 1 - Przykład podłączenia wielu urządzeń do magistrali I^2C (źródło: [3])

Przechodząc do omówienia warstwy łącza danych, transmisja odbywa się poprzez wysyłanie pojedynczych bajtów, gdzie po każdym z nich nadawany jest bit potwierdzenia. Przesył danych zaczyna się od sygnału START, który polega na zmianie stanu linii SDA z wysokiego na niski, przy jednoczesnym utrzymaniu linii SCL w stanie wysokim. Transmisję terminuje sygnał STOP polegający na analogicznej zmianie stanu linii SDA z niskiego na wysoki, przy utrzymaniu SCL w stanie wysokim. Zatem, urządzenie master po nadaniu sygnału START wysyła pierwszy bajt danych w którym pierwsze 7 bitów jest adresem urządzenia z którym próbuje nawiązać połączenie, a ostatni bit jest bitem R/W (*ang. Read/Write),* determinującym kierunek przepływu danych (wartość ‘0’ dla odczytu, ‘1’ dla zapisu). Po nadaniu ostatniego bitu R/W urządzenie nadrzędne zwalnia linię danych, oczekując odpowiedzi od wywołanego urządzenia podrzędnego. Slave powinien w tym czasie podtrzymać linię SDA w stanie niskim podczas taktu zegara na linii SCL, co rozumiane jest jako nadanie sygnału ACK (*ang. Acknowledge)*, potwierdzającego gotowość na nadanie/odbiór danych. Jeśli jednak w tym czasie linia SDA pozostanie w stanie wysokim, urządzenie master uzna to jako brak potwierdzenia, tj. sygnał NACK (*ang. Not Acknowledge)* i będzie zmuszony zakończyć transmisję sygnałem STOP, bądź nadać sygnał REPEATED START, rozpoczynający kolejną transmisję (Rysunek 2).



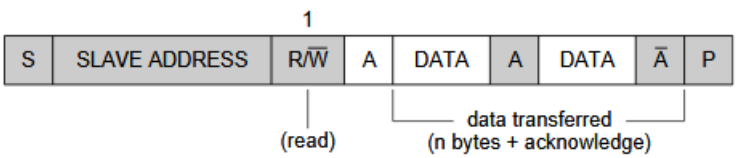
Rysunek 2 - Przebieg przykładowej transmisji za pomocą magistrali I²C (źródło: [3])

W przypadku gdy bit R/W ma wartość ‘0’ (transmisja typu tx), dalsza transmisja odbywa się podobnie jak w przypadku pierwszego bajtu, tj. po każdym wysłanym bajcie master czeka na sygnał zwrotny od skomunikowanego urządzenia, kończąc transmisję sygnałem STOP lub REPEATED START w przypadku, gdy wysłane zostały wszystkie bajty lub nie dostał potwierdzenia od urządzenia slave (Rysunek 3).



Rysunek 3 - Przykładowa transmisja typu tx (źródło: [3])

Dla transmisji przychodzącej (transmisja typu rx) gdzie bit R/W ma wartość ‘1’, po pierwszym wysłanym przez urządzenie nadrzędne bajcie adresowym i otrzymaniu potwierdzenia, odbiera ono nadawane przez drugie urządzenie dane, potwierdzając odbiór każdego bajtu sygnałem ACK na linii danych. Gdy urządzenie master będzie chciało przerwać transmisję, nadaje ono sygnał NACK podczas trwania taktu zegara przeznaczonego do potwierdzenia i kończy transmisję sygnałem STOP, bądź REPEATED START (Rysunek 4).



Rysunek 4 - Przykład transmisji typu rx (źródło: [3])

1.4 Narzędzia programistyczne

Przy tworzeniu tego projektu, głównym narzędziem było środowisko programistyczne Xilinx ISE, za pomocą którego napisany kod był syntezowany i analizowany. Dzięki niemu możliwe było śledzenie przebiegu poszczególnych etapów implementacji pisanego sterownika takich jak podgląd schematu RTL czy dostęp do generowanych, licznych raportów o przebiegu syntezy i implementacji napisanego kodu.

Kolejnym wykorzystanym programem był edytor tekstu Github Atom. Autor zdecydował się na pisanie kodu w zewnętrznym edytorze ze względu na jego szerokie możliwości modyfikacji oraz dostosowywania do potrzeb i przyzwyczajeń użytkownika. Został on odpowiednio przystosowany do pracy nad projektem poprzez instalacje rozszerzeń pozwalających na kolorowanie i sprawdzanie składni języka VHDL oraz zintegrowanie z systemem kontroli wersji Git i serwisem internetowym Github, na którym kod był przechowywany w prywatnym repozytorium.

Do przeprowadzania symulacji użyty został program ISim, który jest zintegrowany z omówionym wcześniej środowiskiem Xilinx ISE. Narzędzie to pozwala na przeprowadzenie m.in. symulacji behawioralnych, Post-Map czy Post-Route. Oprócz parametrów takich jak czas symulacji czy wybór określonych sygnałów poddawanych analizie czasowej, możliwe jest również przeprowadzanie symulacji krok po kroku, z użyciem określanych przez użytkownika punktów przerwania. Dużą zaletą symulatora jest również zmiana opisanego w języku VHDL scenariusza symulacji „w locie” i natychmiastowe przeprowadzenie jej ponownie.

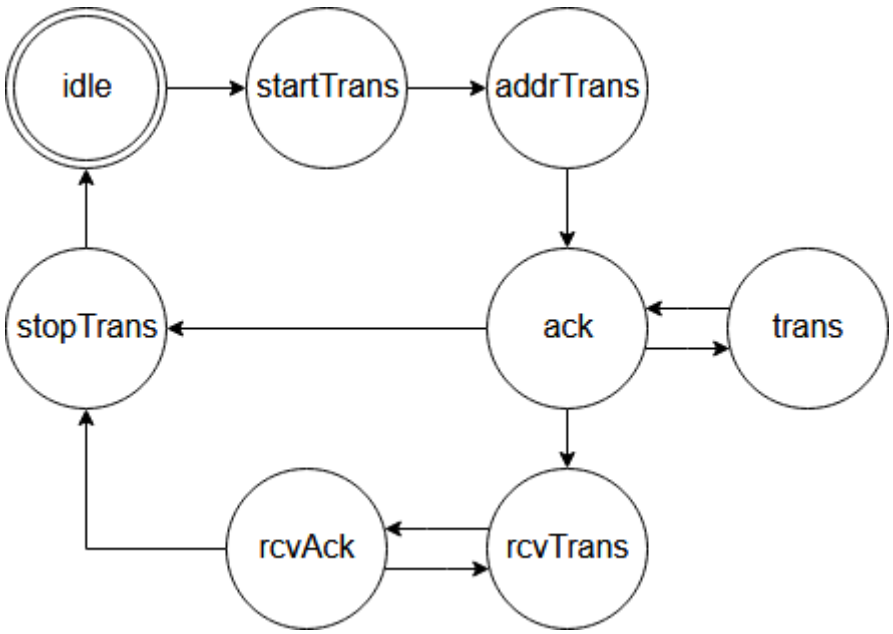
Programem użytym do komunikacji komputera z płytą Spartan-3E Starter Kit i znajdującym się na nim modułem FPGA był iMPACT firmy Xilinx. Tak jak w przypadku symulatora ISim, program ten jest zintegrowany ze środowiskiem Xilinx ISE, co stanowi spore ułatwienie przy wykorzystaniu wygenerowanego uprzednio pliku .bit w konfiguracji danego modułu FPGA. Program iMPACT skanuje porty wyjściowe komputera, w poszukiwaniu podłączonego urządzenia, a po jego znalezieniu proces programowania układu trwa najwyżej parępaście sekund.

2. Projekt sterownika

2.1 Zarys rozwiązania

Aby spełnić założenia postawione dla projektu wymagane było stworzenie ogólnego schematu działania sterownika, tak aby realizował on stawiane mu wymagania, działając jednocześnie w ściśle określonym środowisku. Architektura układów FPGA narzucała implementacje sterownika jako wiele, równolegle działających procesów synchronizowanych ze sobą poprzez proces główny, będący automatem skończenie stanowym. Realizacja takiego automatu, jego poszczególnych stanów oraz grafu opisującego przejścia pomiędzy nimi zależą bezpośrednio od protokołu transmisji magistrali I^2C. W następnym podpunkcie znajduje się zaproponowany w projekcie diagram maszyny stanów, który jest reprezentowany przez graf opisujący możliwe przejścia z danych stanów automatu. Następnie omówiony zostanie ogólny schemat zmodularyzowanego sterownika i jego miejsce w środowisku testowym.

2.2 Diagram maszyny stanów



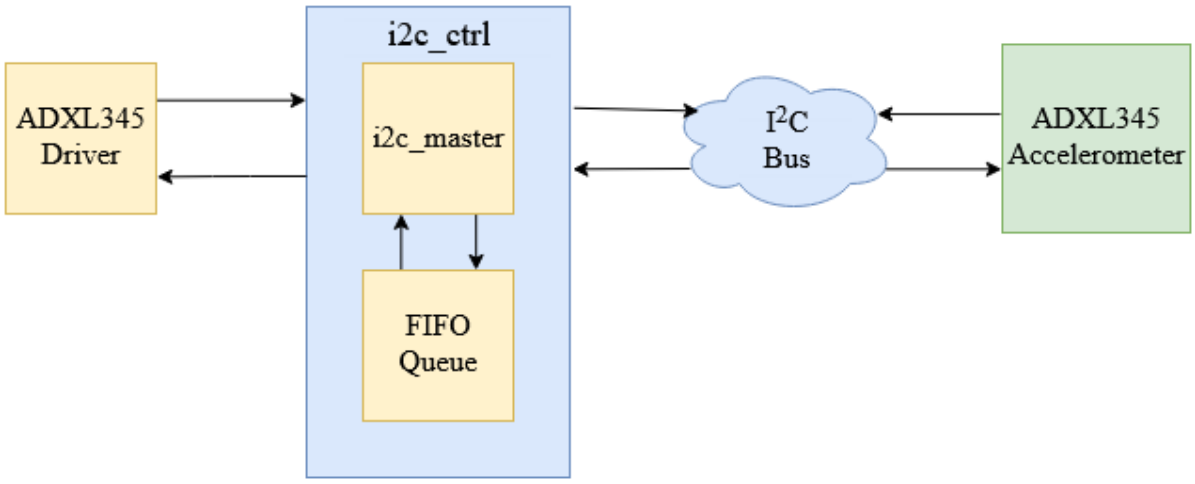
Rysunek 5 - Graf automatu stanowego

Powyższy rysunek przedstawia diagram zaimplementowanej maszyny stanów, przedstawiony w formie grafu. Warunki przejść pomiędzy stanami zostaną teraz opisane, zaczynając od stanu początkowego. Stan *idle* jest stanem, w którym sterownik znajduje się oczekując na sygnał rozpoczęcia transmisji. Po otrzymaniu sygnału startowego maszyna przechodzi w stan *startTrans* w którym na magistrali nadawany jest sygnał START po

zakończeniu którego sterownik przechodzi do stanu *addrTrans*. Wtedy to na linii danych transmitowany jest bajt adresowy, składający się z 7-bitowego adresu wywoływanego urządzenia i bitu kierunku transmisji. Następnie maszyna przechodzi do stanu *ack* w którym oczekuje na potwierdzenie otrzymania wysłanego bajtu przez urządzenie slave. W razie potwierdzenia przez urządzenie podrzędne odbioru danych, następny stan zależny jest od kierunku zainicjowanej uprzednio transmisji. Jeśli wysłany bit R/W miał wartość '0' sterownik przechodzi w stan *trans* w którym wystawia na linii SDA kolejne bajty transmitowane do adresata. Po każdym z wysłanych bajtów automat wraca do stanu *ack* w którym oczekuje potwierdzenia odebrania wysłanych danych. Gdy bit R/W był ustawiony na '1' sterownik przychodzi do stanu *rcvTrans* gdzie odbiera dane wysyłane przez urządzenie podrzędne. Po każdym odebranym bajcie moduł sterownika przechodzi do stanu *rcvAck* w którym to potwierdza odebranie bajtu danych. W razie, gdy chce zakończyć transmisję przychodzącą nadaje w tym stanie sygnał NACK i przechodzi w stan *stopTrans* gdzie nadawany jest sygnał STOP i po zakończeniu transmisji automat przechodzi w stan spoczynku – *idle*. Do stanu *stopTrans* może przejść również ze stanu *ack* w przypadku, gdy przez sterownik kończona jest transmisja nadawcza, bądź gdy urządzenie odbiorcze nada sygnał NACK, który informuje o błędzie w odbiorze wysłanego bajtu po swojej stronie.

2.3 Schemat ideowy projektu

W celu umożliwienia przetestowania realizowanego projektu wymagane było opracowanie schematu, który uwzględniałby skorzystanie ze sterownika pod kątem komunikacji z konkretnym urządzeniem podłączonym do magistrali, zachowując uniwersalny charakter opracowywanego modułu. Należało zatem rozgraniczyć poszczególne jednostki



Rysunek 6 - Ogólny schemat realizowanego projektu

składowe, stosując odpowiednią modularyzacje projektu i zaprojektować logikę komunikacji pomiędzy nimi. Opracowany schemat znajduje się poniżej.

Głównym przedmiotem projektu jest moduł zaznaczony na powyższym rysunku niebieskim prostokątem, będzie on nazywany jako *i2c_ctrl*. Zawiera on w sobie rdzeń sterownika magistrali I^2C (nazywany dalej jako *i2c_master*), a który jest bezpośrednio skomunikowany z modułem kolejki FIFO i pośredniczy w komunikacji pomiędzy nią, a modułami odpowiedzialnymi za logikę transmisji z urządzeniami zewnętrznymi, w tym przypadku z modułem sterownika akcelerometru ADXL345. Moduł *i2c_master,* wraz z modułem kolejki FIFO są „widziane" z poziomu pozostałych modułów jako jedność i stanowią kompletny sterownik, nazwany wcześniej *i2c_ctrl*. Dzięki takiemu rozwiązaniu, obsługa sterownika przez zewnętrzne moduły jest znacząco uproszczona i sprowadza się w przypadku transmisji wychodzącej do załadowania danych przeznaczonych do wysłania do kolejki i podaniu adresu oraz bitu R/W na odpowiednie wejścia modułu, inicjując następnie jego pracę sygnałem podanym na wejście startujące.

Przebieg transmisji został częściowo opisany w podrozdziale 2.2, jednak należy ten opis rozszerzyć biorąc pod uwagę integralną rolę kolejki FIFO, którą odgrywa ona w logice pracy sterownika. Przechodząc do opisu samej kolejki, ma ona pojemność 16 bajtów, a komunikacja z nią odbywa się za pośrednictwem sprzężonego z nią submodułu *i2c_master*. Z punktu widzenia zewnętrznego użytkownika sterownika *i2c_ctrl* transmisja odbywa się poprzez jego zewnętrzne porty. Jednak walidacją poprawności żądanych operacji zajmuje się *i2c_master*, który w zależności od stanu w jakim znajduje się sterownik może zignorować pewne komendy. Przykładem takiej ignorowanej operacji może być żądanie zapisu wystawionego na odpowiednie wejście bajtu, podczas gdy sterownik jest w trakcie realizacji transmisji przychodzącej. W takim wypadku dozwolony jest jedynie odczyt danych z kolejki FIFO, ponieważ w przypadku zapisu mogłyby wystąpić problemy czasowe wynikające z próby jednoczesnego zapisu danych odbieranych w trakcie transmisji, jak i danych podawanych na zewnętrzne porty sterownika przez użytkownika. Moduł zachowuje się podobnie w analogicznym przypadku, kiedy to sterownik jest w trakcie wysyłania danych do skomunikowanego urządzenia, a zewnętrzny użytkownik próbowałby jednocześnie te dane odczytywać z zewnętrznych portów sterownika *i2c_ctrl*. Ignorowane są również próby czytania z pustej kolejki, jak i próby zapisu w stanie, gdy kolejka jest już pełna. Te niebezpieczne operacje mogłyby naruszyć poprawną pracę modułu kolejki i doprowadzić do wewnętrznych stanów automatu, które nie powinny mieć miejsca.

Transmisja danych w obu kierunkach może zostać zakończona w dwóch następujących przypadkach:

- Podczas próbkowania sygnałów *EmptyFifo/FullFifo* w stanach odpowiednio *ack/rcvAck* są one ustawione na ‘1’.
- Podczas próbkowania sygnału *ConTrans* w stanach *ack/rcvAck* jest on ustawiony na ‘0’.

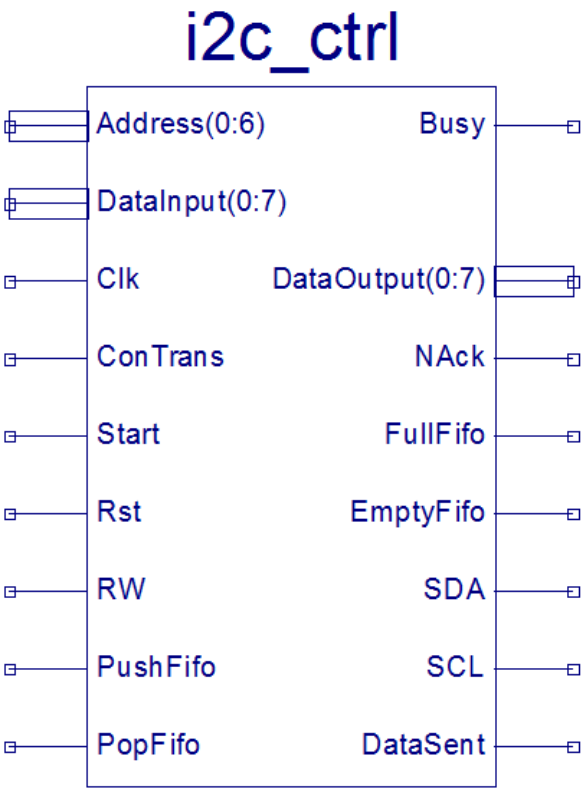
Pierwsza sytuacja ma miejsce gdy podczas transmisji wychodzącej kolejka FIFO jest pusta i tym samym nie ma danych do wysłania. Dla transmisji przychodzącej sytuacja ta ma miejsce gdy kolejka FIFO jest przepełniona i nie ma miejsca na zapis kolejnych bajtów przesłanych przez magistrale.

W drugim przypadku zewnętrzny użytkownik modułu sterownika decyduje się na przerwanie transmisji i ustawia sygnał na wejściu *ConTrans* modułu w stan niski podczas jego próbkowania gdy sterownik znajduje się w stanach *ack/rcvAck*.

3. Implementacja projektu

3.1 Schemat sterownika

Dalej znajduje się schemat modułu sterownika *i2c_ctrl*, zawierającego dwa podmoduły tj. kolejkę FIFO (moduł *fifo_queue*) jak i właściwy moduł sterownika magistrali I^2C (*i2c_master)*. W następnym podrozdziale zostaną opisane porty jego podmodułu *i2c_master*, połączenia oraz wyprowadzenia zewnętrzne.



Rysunek 7 - Schemat modułu i2c_ctrl

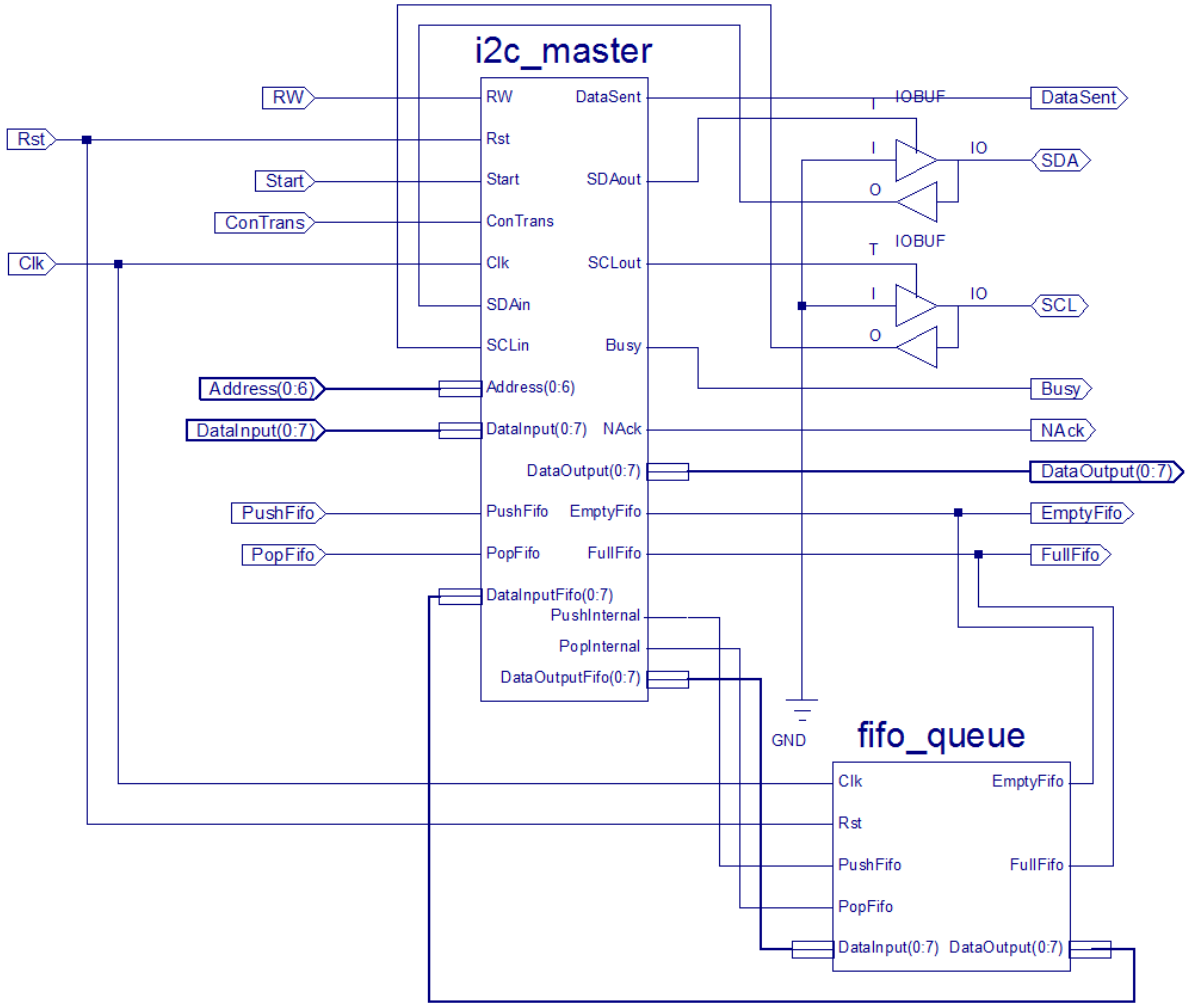
Zaczynając od portów wejściowych modułu *i2c_masteri,* widocznego na rysunku 8, są to kolejno:

- **Clk** – Wejście zegarowe na który podawany jest 50MHz sygnał taktujący moduł sterownika, używany do synchronizacji procesów.
- **Rst** – Wejście służące do resetowania stanów wewnętrznych modułu do wartości początkowych.
- **Start** – Wejście na które podawany jest impuls startowy sterownika.
- **Address(0:6)** – Wejście na wektor 7-bitowego adresu urządzenia, z którym sterownik podejmie komunikacje.
- **RW** – Wejście na bit R/W, definiujący kierunek inicjowanej transmisji.

- **DataInput(0:7)** – Wejście wektorowe na bajt danych przeznaczony do zapisania w pamięci kolejki FIFO i wysłania do urządzenia docelowego.
- **PushFifo** – Impuls na tym wejściu powoduje zapisanie bajtu podanego na wejściu DataInput(0:7) do kolejki FIFO.
- **PopFifo** – Impuls na tym wejściu powoduje wystawienie następnego bajtu z kolejki FIFO na wyjście DataOutput(0:7).
- **EmptyFifo** – Wejście przeznaczone do sygnalizacji pustej kolejki FIFO.
- **FullFifo** – Wejście przeznaczone do sygnalizacji pełnej kolejki FIFO.
- **DataInputFifo(0:7)** – Wejście wektorowe służące do komunikacji wewnętrznej pomiędzy sterownikiem a kolejką FIFO. Na nim odbierane są kolejne bajty odczytywane z kolejki.
- **SDAin** – Wejście na którym próbkowany jest stan linii danych magistrali, wykorzystywane przy transmisji przychodzącej.
- **SCLin** – Wejście na którym próbkowany jest stan linii zegarowej magistrali.
- **ConTrans** – Wejście służące do kontroli długości transmisji.

Wyjścia modułu sterownika:

- **DataSent** – Wyjście na którym sygnalizowane impulsem jest wysłanie bajtu przy transmisji wychodzącej.
- **Busy** – Wyjście na którym sygnalizowana jest zajętość sterownika.
- **NAck** – Wyjście służące do sygnalizacji otrzymania braku potwierdzenia ze strony urządzenia podrzędnego.
- **DataOutput(0:7)** – Wyjście wektorowe na którym wystawiane są dane z kolejki FIFO.
- **PushInternal** – Wyjście służące do wewnętrznej komunikacji pomiędzy sterownikiem, a kolejką FIFO. Impuls na tym wyjściu powoduje zapisanie bajtu podanego na wyjściu wewnętrznym *DataOutputFifo(0:7)* do kolejki.
- **PopInternal** - Wyjście służące do wewnętrznej komunikacji pomiędzy sterownikiem, a kolejką FIFO. Impuls na tym wyjściu powoduje wystawienie następnego bajtu z kolejki FIFO na wewnętrzne wejście *DataInputFifo(0:7).*
- **DataOutputFifo(0:7)** – Wyjście wektorowe służące do wewnętrznej komunikacji pomiędzy sterownikiem, a kolejką FIFO. Za jego pomocą wysyłane są kolejne bajty zapisywane do kolejki.



Rysunek 8 - Schemat sterownika z widoczną modularyzacją

Jak można zauważyć na powyższej ilustracji większość portów modułu *i2c_master* jest wyprowadzona „na zewnątrz” utworzonego schematu i jest dostępna do wykorzystania dla potencjalnych zewnętrznych modułów korzystających ze sterownika *i2c_ctrl*. Tylko 6 wyjść/wejść sterownika wykorzystywanych jest do komunikacji z kolejką FIFO znajdujących się w osobnym module. Sama kolejka posiada dwa porty wyjściowe widoczne zewnętrznie (*EmptyFifo* i *FullFifo*) przeznaczone do sygnalizacji jej pustego/pełnego stanu. Ostatnim ważnym aspektem powyższego schematu, jest sposób połączenia linii magistrali SDA i SCL. Widoczne są one zewnętrznie jako porty dwukierunkowe. Aby było to możliwe, zostały one podłączone do modułu sterownika za pomocą buforów trójstanowych. Jako, że obie linie magistrali I^2C są podłączone na stałe do dodatniego napięcia poprzez tzw. rezystory pull-up, kiedy magistrala nie jest zajęta panuje na niej stan wysoki. Dlatego sterowanie liniami danych i zegara, polega na „przytrzymaniu” ich w stanie niskim przez określony czas. To zapewniają

właśnie bufory trójstanowe, gdzie podłączone do nich wyjścia *SDAout* i *SCLout* pełnią role sygnałów sterujących. Gdy są one w stanie wysokim, wyjścia buforów znajdują się w stanie wysokiej impedancji (Z), natomiast gdy są one w stanie niskim, na wyjście przekazywany jest sygnał wejściowy, który na schemacie jest zwarty do masy, co powoduje utrzymanie linii w stanie niskim.

3.2 Opis modułu i2c_master

Moduł *i2c_master* stanowi kluczową jednostkę realizowanego projektu. Odpowiada on za obsługę transmisji danych poprzez magistralę I^2C i stanowi interfejs ułatwiający zewnętrznym modułom komunikację za pośrednictwem tejże magistrali. Poniżej znajduje się opis działania modułu, w szczególności zawartych w nim procesów.

3.2.1 Proces FSM

Głównym procesem modułu, jest proces realizujący maszynę stanów, identyczną z tą opisaną w podrozdziale 2.2. Na poniższym listingu można dokładnie zaobserwować warunki przejść pomiędzy stanami, które zostały pokrótce opisane w poprzednim rozdziale. Poza stanem początkowym *idle*, przejście z każdego innego stanu jest uzależnione od wartości sygnału *clkCount*, który jest odpowiedzialny za synchronizację działania procesów pracujących w oparciu o 400 kHz tryb pracy magistrali I^2C. Proces sterujący tym sygnałem zostanie opisany jako następny.

```
-------------------------------------------------------------------------------
-- State Machine process
-------------------------------------------------------------------------------
FSM: process(state, clkCount, Start, transDirection, dataCount, SDAin, ConTrans,
             EmptyFifo, FullFifo)
begin
   NAck <= '0';
   nextState <= state;
   case state is
      when idle =>
         if Start = '1' then
            nextState <= startTrans;
         end if;
      when startTrans =>
         if clkCount = "1111100" then
            nextState <= addrTrans;
         end if;
      when addrTrans =>
         if dataCount = 7 and clkCount = "1111100" then
            nextState <= ack;
         end if;
      when trans =>
         if dataCount = 7 and clkCount = "1111100" then
            nextState <= ack;
         end if;
      when rcvTrans =>
         if dataCount = 7 and clkCount = "1111100" then
            nextState <= rcvAck;
         end if;
      when ack =>
         if clkCount = "1111100" then
            case SDAin is
               when '0' => -- data ACK by slave
                  if ConTrans = '1' and transDirection = '0' then
                     nextState <= trans;
                  elsif ConTrans = '1' and transDirection = '1' then
                     nextState <= rcvTrans;
                  elsif ConTrans = '0' then
                     nextState <= stopTrans;
                  end if;
               when '1' => -- data NACK by slave
                  NAck <= '1';
                  nextState <= stopTrans;
               when others =>
                  nextState <= stopTrans;
            end case;
            if transDirection = '0' and EmptyFifo = '1' then
               nextState <= stopTrans;
            end if;
         end if;
      when rcvAck =>
         if ClkCount = "1111100" then
            if FullFifo = '0' and ConTrans = '1' then
               nextState <= rcvTrans;
            elsif FullFifo = '1' or ConTrans = '0' then
               nextState <= stopTrans;
            end if;
         end if;
      when stopTrans =>
         if clkCount = "1111100" then
            nextState <= idle;
         end if;
   end case;
end process FSM;
```

Listing 1 - Kod procesu FSM

3.2.2 Proces ClockCounter

Proces ten odpowiada za inkrementację sygnału *ClkCount*. Licznik zlicza zbocza narastające wejścia zegarowego w stanach aktywnej pracy modułu, zerując sterowany sygnał co 125 taktów. Dzięki temu osiągnięta częstotliwość cyklu wynosi 400 kHz, co odpowiada wykorzystywanemu przez projekt standardowi prędkości transmisji Fast-mode[3].

```
-------------------------------------------------------------------------------
-- Clock Counter process (0-124 clock ticks)
-------------------------------------------------------------------------------
ClockCounter: process(Clk, Rst, state)
begin
   if rising_edge(Clk) then
      if Rst = '1' or state = idle then
         clkCount <= "0000000";
      else
         if clkCount = "1111100" then
            clkCount <= "0000000";
         else
            clkCount <= clkCount + 1;
         end if;
      end if;
   end if;
end process ClockCounter;
```

Listing 2 - Kod procesu ClockCounter

3.2.3 Proces DataCounter

Jest on odpowiedzialny za zliczanie wysyłanych bądź odbieranych za pomocą magistrali I^2C bitów. W stanach poprzedzających transmisję, wartość sygnału *dataCount* ustawiana jest na -1, aby liczone bity były numerowane od zera.

```
-------------------------------------------------------------------------------
-- Counter of bits sent
-------------------------------------------------------------------------------
DataCounter: process(state, Clk, clkCount)
begin
   if rising_edge(Clk) then
      if state = startTrans or state = ack or state = rcvAck then
         dataCount <= -1;
      end if;
      if (state = addrTrans or state = trans or state = rcvTrans)
            and clkCount = "00000000" then
         dataCount <= dataCount + 1;
      end if;
   end if;
end process DataCounter;
```

Listing 3 - Kod procesu DataCounter

3.2.4 Proces FifoQueue

Proces *FifoQueue* realizuje komunikacje pomiędzy modułem *i2c_master*, a sprzężonym modułem *fifo_queue*. Na poniższym listingu można wyróżnić dwie części omawianego procesu, jedna dla transmisji rx, druga dla tx. Przy transmisji nadawczej, dozwolone jest pisanie

do kolejki przez zewnętrznego użytkownika, o ile nie jest ona zapełniona (*FullFifo = '0'*). Dodatkowo, po każdorazowym wysłaniu bajtu, do kolejki wysyłany jest sygnał *PopInternal*, przesuwający wewnętrzy wskaźnik na „ogon" kolejki, usuwający z punktu widzenia użytkownika przetransmitowany bajt danych z pamięci. Proces zachowuje się analogicznie dla transmisji odbiorczej, ograniczając możliwe do wykonania operacje do czytania przychodzących danych, o ile kolejka nie jest pusta. Ponadto przesuwa on wskaźnik na „głowę" kolejki po każdym odebranym bajcie danych. Warto zauważyć, że gdy sterownik jest w stanie *idle*, obie operacje czytania i pisania są dozwolone, co pozwala na wygodny dostęp do danych podczas gdy przez moduł nie jest realizowana transmisja.

```
-------------------------------------------------------------------------------
-- Fifo queue driver process
-------------------------------------------------------------------------------
FifoQueue: process(state, Clk, clkCount)
begin
   if rising_edge(Clk) then
      PushInternal <= '0';
      PopInternal <= '0';
      -- Outgoing transmission
      if transDirection = '0' or state = idle then
         if PushFifo = '1' and FullFifo = '0' then
            DataOutputFifo <= DataInput;
            PushInternal <= '1';
         end if;
         if state = trans then
            if clkCount = "1111100" and dataCount = 7 then
              PopInternal <= '1';
            end if;
         end if;
      end if;
      -- Incomming transmission
      if transDirection = '1' or state = idle then
         DataOutput <= DataInputFifo;
         if PopFifo = '1' and EmptyFifo = '0' then
            PopInternal <= '1';
         end if;
         if state = rcvTrans then
            if clkCount = "1111100" and dataCount = 7 then
               DataOutputFifo <= dataRecieved;
               PushInternal <= '1';
            end if;
         end if;
      end if;
   end if;
end process FifoQueue;
```

Listing 4 - Kod procesu FifoQueue

3.2.5 Proces SdaDriver

Ten proces steruje portem wyjściowym *SDAout*, które jest połączone przez bufor trójstanowy do linii danych magistrali I^2C. W zależności od stanu automatu i wartości licznika *clkCount* linia SDA jest odpowiednio sterowana w czasie, gdy linia SCL jest w stanie niskim. Wyjątek stanowi przypadek, gdy proces generuje sygnały START i STOP, zgodnie

z dokumentacją magistrali[3]. W przypadku potwierdzenia transmisji przychodzącej, warunkowo generowany jest sygnał NACK, gdy zewnętrzny użytkownik chce zakończyć transmisję utrzymując port wejściowy *ConTrans* w stanie niskim, bądź ustawiona jest flaga *FullFifo* sygnalizująca brak miejsca do zapisu następnego bajtu w kolejce FIFO. Dodatkowo, w przypadku sygnalizowania na wejściu *ConTrans* chęci przerwania transmisji, należy utrzymać na nim stan niski co najmniej do końca stanu *rcvAck*, kiedy będzie ono powtórnie próbkowane przez proces maszyny stanów. W przeciwnym razie sterownik przejdzie do stanu *rcvTrans*, pomimo wysłania sygnału NACK i zerwania połączenia.

```
-------------------------------------------------------------------------------
-- SDAout signal driver
-------------------------------------------------------------------------------
SdaDriver: process(state, clkCount, Clk, dataCount)
begin
   if rising_edge(Clk) then
      if state = idle then
         SDAout <= '1';
      end if;
      if state = startTrans then
         SDAout <= '0';
      end if;
      if state = addrTrans then
         if clkCount = "0100011" then
            SDAout <= addressByte(dataCount);
         end if;
      end if;
      if state = trans then
         if clkCount = "0100011" then
            SDAout <= DataInputFifo(dataCount);
         end if;
      end if;
      if state = rcvTrans then
         SDAout <= '1';
      end if;
      if state = ack then
         if clkCount = "0001100" then
            SDAout <= '1';
         end if;
      end if;
      if state = rcvAck then
         if clkCount = "0001100" then
            if FullFifo = '1' or ConTrans = '0' then
               SDAout <= '1'; -- NACK signal
            else
               SDAout <= '0'; -- ACK signal
            end if;
         end if;
      end if;
      if state = stopTrans then
         if clkCount = "0000000" then
            SDAout <= '0';
         end if;
         if clkCount = "1111100" then
            SDAout <= '1';
         end if;
      end if;
   end if;
end process SdaDriver;
```

Listing 5 - Kod procesu SdaDriver

3.2.6 Proces SclDriver

Proces *SclDriver* steruje portem wyjściowym *SCLout* i jest odpowiedzialny za generowanie na linii SCL magistrali sygnałów zegarowych. Dla stanu *idle* wyjście *SCLout* utrzymywane jest w stanie wysokim, co przekłada się de facto na zwolnienie magistrali. Zdefiniowane tutaj zostało również zachowanie linii zegarowej przy nadawaniu sygnałów START i STOP. W pozostałych przypadkach proces generuje impulsy zegarowe z częstotliwością 400kHz.

```
-------------------------------------------------------------------------------
-- Scl driver process
-------------------------------------------------------------------------------
SclDriver: process(state, Clk, clkCount)
begin
   if rising_edge(Clk) then
      if state = idle then
         SCLout <= '1';
      end if;
      if state = startTrans then
         if clkCount = "1111100" then
            SCLout <= '0';
         end if;
      end if;
      if state = trans or state = addrTrans or state = rcvTrans
            or state = ack or state = rcvAck then
         if clkCount = "1001000" then
            SCLout <= '1';
         end if;
         if clkCount = "1111100" then
            SCLout <= '0';
         end if;
      end if;
      if state = stopTrans then
         if clkCount = "1001000" then
            SCLout <= '1';
         end if;
      end if;
   end if;
end process SclDriver;
```

Listing 6 - Kod procesu SclDriver

3.2.7 Pozostałe procesy

Z uwagi na niski stopień złożoności pozostałych procesów, zostaną one pokrótce opisane w tym punkcie pracy, bez prezentowania ich kodu w postaci listingów.

- **SdaInDriver** – proces odpowiedzialny za odbiór danych z portu *SDAin*. Stan tego wejścia próbkowany jest podczas transmisji rx i zapisywany.
- **WaitAck** – proces generuje impuls na wyjściu *dataSent* po zakończeniu przesyłu/odbioru pełnego bajtu danych.
- **InputLatch** – proces zatrzaskujący sygnały na wejściach *Address* i *RW*, w chwili podania impulsu startowego.

- **BusyDriver** – proces sterujący sygnałem wyjściowym *Busy*. Sygnalizuje stan zajętości sterownika.

3.3 Opis modułu fifo_queue

Moduł *fifo_queue* realizuje 16-bajtową kolejkę FIFO, przeznaczoną do buforowania danych używanych podczas transmisji z wykorzystaniem magistrali I^2C. Jest on zintegrowana wraz z modułem *i2c_master* i razem z nim tworzy moduł sterownika *i2c_ctrl* będący przedmiotem tej pracy. Przy realizacji kolejki, zaczerpnięto z implementacji podanej w źródle[4], przystosowując kod do specyfiki sterownika. Cała kolejka jest napisana przy użyciu jednego procesu i zawiera następujące sygnały wewnętrzne:

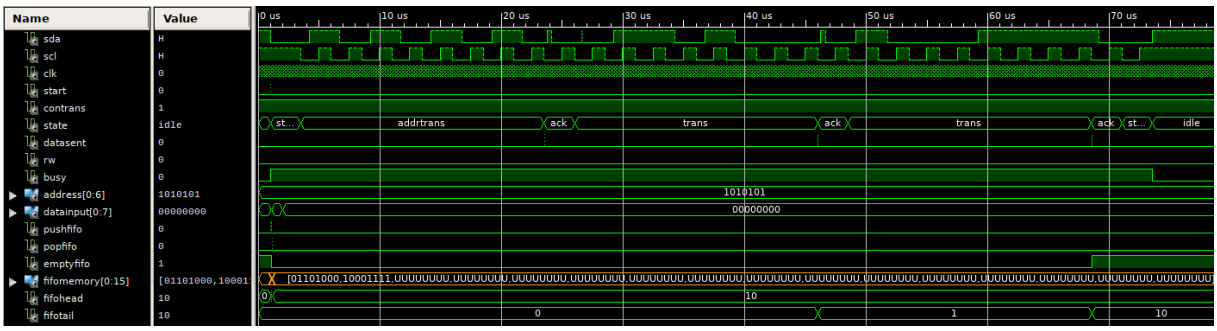
- **fifoMemory** – jest to sygnał niestandardowego typu, będący 16-elementową tablicą 8-bitowych wektorów. Jest on pamięcią kolejki.
- **fifoHead/fifoTail** – wskaźniki na „głowę" i „ogon" kolejki.
- **fifoEmpty/fifoFull** – sygnały informujące o stanie zajętości kolejki.
- **fifoLooped** – 1-bitowy sygnał informujący, czy wartość wskaźnika na „głowę" kolejki jest mniejsza niż wartość wskaźnika „ogona" kolejki. Jest on wymagany ze względu na potrzebę rozróżnienia sytuacji w której wartości obu wskaźników są sobie równe (jest to możliwe w przypadku gdy kolejka jest pusta, bądź pełna).

Kolejka oczekuje na zewnętrzne sygnały na portach wejściowych *PushFifo/PopFifo*, następnie sprawdza, czy dla odpowiedniej operacji są spełnione warunki określone sygnałami *fifoFull/fifoEmpty*. Gdy warunki są spełnione, w przypadku sygnału *PushFifo* podany na wejście *DataInput* bajt zostaje zapisany do tablicy *fifoMemory* pod indeks wskazywany przez *fifoHead*, a sam wskaźnik zostaje inkrementowany. Dla *PopFifo* wskaźnik *fifoTail* jest inkrementowany, a na wyjściu *DataOutput* od razu pojawia się bajt wskazywany przez ten wskaźnik. Przed inkrementacją wskaźników, sprawdzana jest ich wartość, aby w przypadku gdy wynoszą one '15' (wskazując na koniec kolejki FIFO), po inkrementacji przypisać im wartość '0' i ustawić odpowiednio flagę *fifoLooped*. Po wykonaniu tych operacji, uaktualniane są sygnały bitowe *fifoEmpty/fifoFull,* po czym proces czeka na następne sygnały typu Push/Pop.

3.4 Symulacja behawioralna modułu i2c_ctrl

Jednym z celów pracy było przetestowanie napisanego sterownika za pomocą narzędzi symulacyjnych. W czasie tworzenia modułu sterownika, zostały przeprowadzone liczne symulacje, sprawdzające poprawność implementowanych rozwiązań. Dzięki takiej formie weryfikacji pisanego kodu, błędy i nieścisłości pojawiające się w trakcie opracowywania

sterownika były eliminowane na bieżąco i poprawiane niemal natychmiastowo, co bardzo ułatwiło proces tworzenia sterownika. W tym podrozdziale zostanie przedstawiona jedna z symulacji transmisji typu tx, z wykorzystaniem kodu symulującego zachowanie urządzenia podrzędnego, pobranego ze strony internetowej dr inż. Jarosława Sugiera[7].

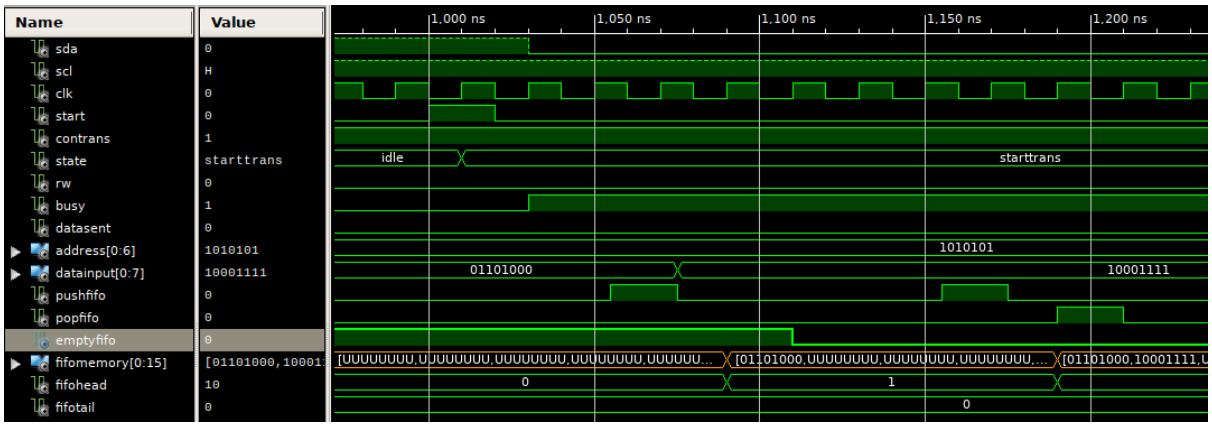


Rysunek 9 - Przebieg transmisji wychodzącej

```
[I2C 1.03 us] START condition
[I2C 23.51 us] address byte: AA
[I2C 46.01 us] byte received: 68
[I2C 68.51 us] byte received: 8F
[I2C 73.51 us] STOP condition
```

Rysunek 10 – Konsola programu iSim po wykonaniu symulacji

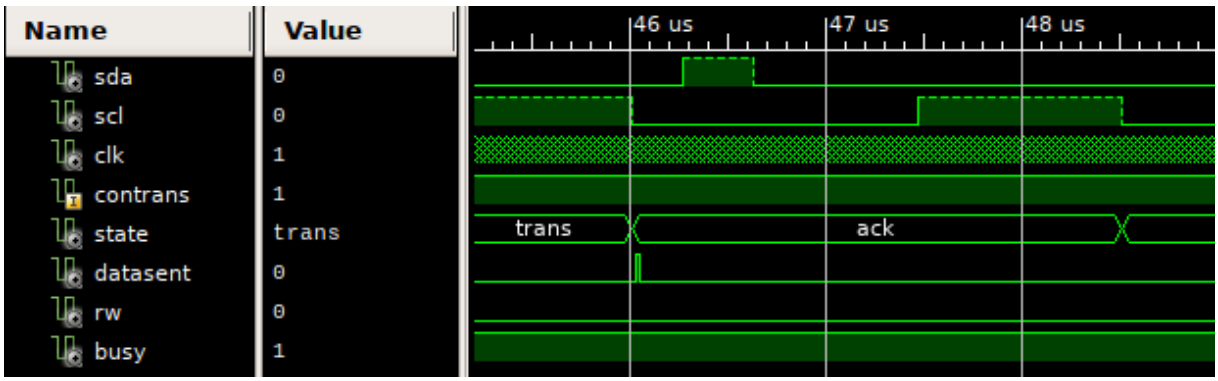
Powyżej przestawiono pełny przebieg symulacji transmisji tx oraz konsolę programu iSim, w której kod symulujący urządzenie docelowe potwierdzał otrzymane po magistrali I^2C dane i sygnały START/STOP.



Rysunek 11 - Inicjacja transmisji

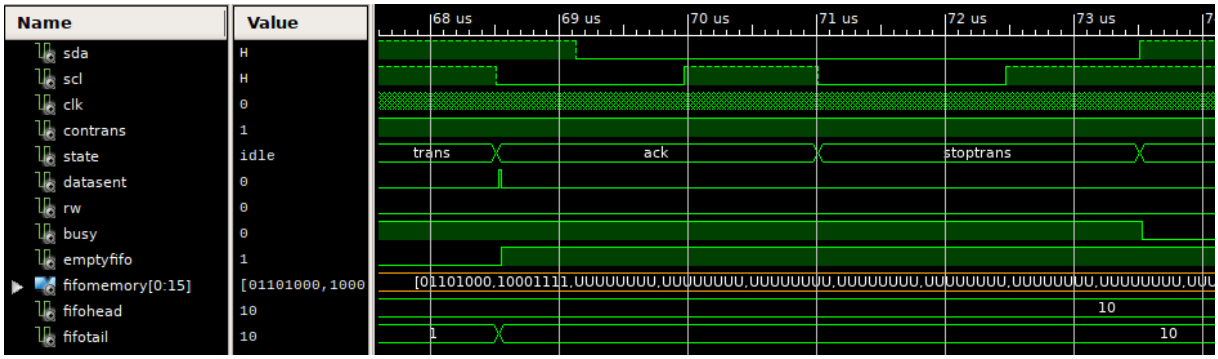
Powyższe zbliżenie przedstawia zapoczątkowanie transmisji przez sterownik. Widoczny jest impuls na wejściu *Start*, który powoduje przejście modułu w stan *startTrans*. Jednocześnie próbkowane są wejścia *Address* oraz *RW*, definiujące adres urządzenia docelowego oraz kierunek inicjowanej transmisji. W tym samym czasie odbywa się zapis transmitowanych do kolejki FIFO bajtów. W dolnej części zrzutu ekranu, można zauważyć

impulsy *PushFifo*, które zapisują wektor bitowy podany na wejściu *DataInput* do pamięci kolejki *fifoMemory* pod wskazany przez wskaźnik *fifoHead* indeks tablicy. Pokazana została również próba niedozwolonej podczas transmisji nadawczej operacji *PopFifo*, której impuls został zignorowany, co widać po niezmienionych wartościach wskaźników.



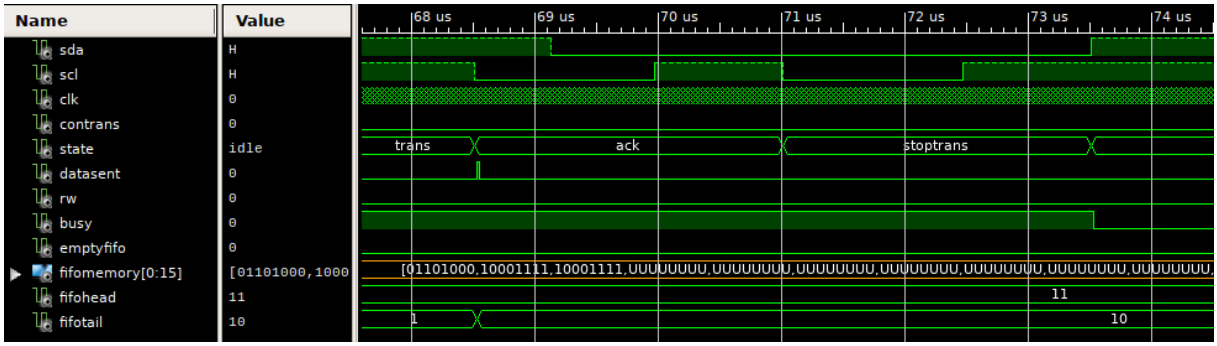
Rysunek 12 - Potwierdzenie odebrania wysłanego bajtu

Na powyższym przebiegu pokazany jest sygnał ACK wysłany przez urządzenie odbiorcze. Na początku stanu *ack* sterownik wysyła impuls na wyjściu *DataSent* informując o wysłaniu bajtu danych. Około 46,6ns symulacji linia SDA zostaje „podtrzymana” w stanie niskim przez symulowane urządzenie i zostaje w takim stanie do końca stanu *ack*.



Rysunek 13 - Zakończenie transmisji

Na rysunku 13, zakończenie transmisji jest spowodowane brakiem danych do wysyłania w kolejce FIFO, co sygnalizuje wyjście *EmptyFifo* będące w stanie wysokim. Dzieje się tak pomimo faktu, że *ConTrans* ma wartość logicznej ‘1’. Sterownik przechodzi do stanu *stopTrans* w którym nadaje sygnał STOP, ustawiając linię SDA w stan wysoki, podczas gdy linia SCL znajduje się również w stanie wysokim. Wraz z przejściem w stan *idle*, sygnał *busy* ustawiany jest w stan niski i moduł przechodzi w tryb oczekiwania na zapoczątkowanie kolejnej transmisji. Poniżej przedstawiona jest sytuacja w której zakończenie transmisji jest spowodowane sygnałem *ConTrans* w stanie niskim pomimo faktu, że w pamięci kolejki FIFO znajduje się bajt danych gotowych do wysłania.

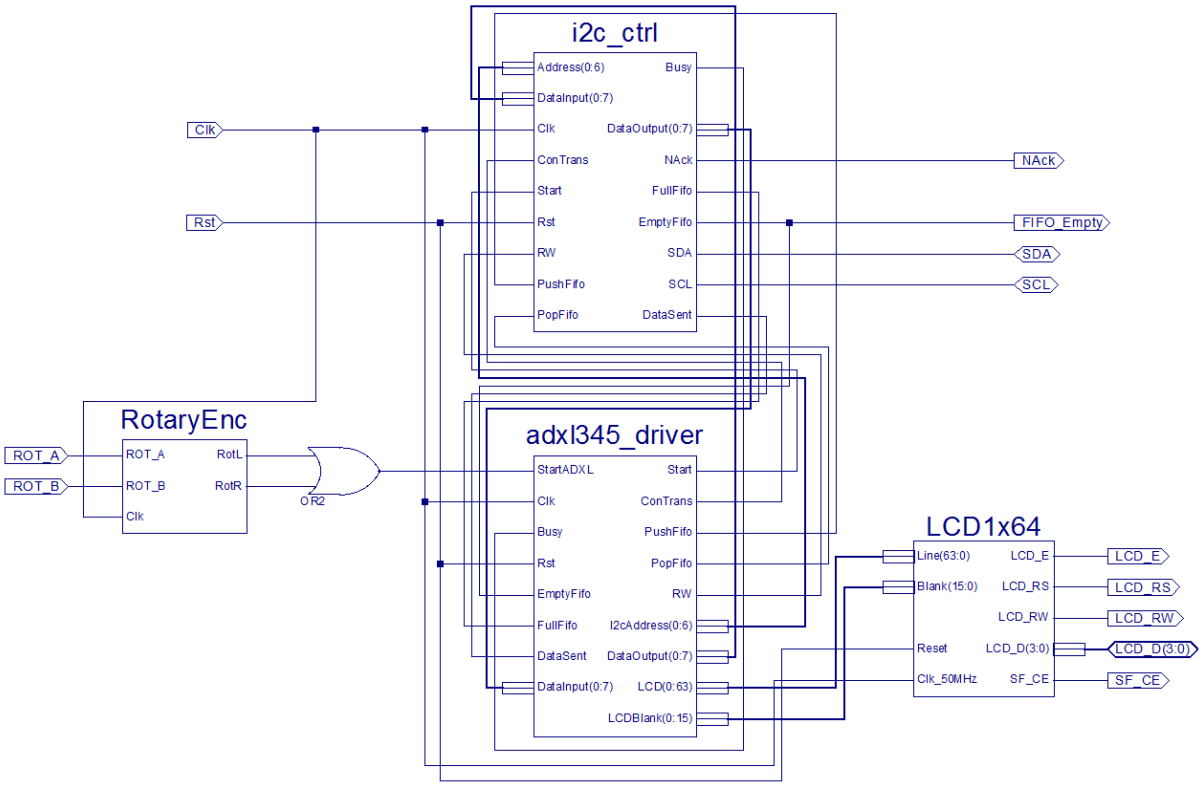


Rysunek 14 - Zakończenie transmisji sygnałem ConTrans = '0'

4. Weryfikacja sprzętowa

4.1 Schemat i pliki ucf

Po przeprowadzeniu symulacyjnych testów sterownika, następnym krokiem była sprzętowa weryfikacja poprawności jego działania. W tym celu został opracowany schemat, pozwalający na przeprowadzenie testów na fizycznym układzie.



Rysunek 15 - Schemat do testów sprzętowych

Przy realizacji tego schematu zostały wykorzystane trzy dodatkowe moduły, z czego dwa z nich, zostały pobrane ze strony[7]. Pobrane moduły to *LCD1x64* oraz *RotaryEnc*. Pierwszy z nich umożliwia komunikacje z wbudowanym w płytę Spartan-3E FPGA Starter Kit wyświetlaczem LCD, na którym wyświetlane są odebrane podczas transmisji dane. Drugi wykorzystuje zamontowany na płycie enkoder obrotowy, który posłuży za generator impulsu startowego. Trzecim modułem, jest napisany przez autora *adxl345_driver*, który odpowiada za logikę komunikacji z akcelerometrem, wykorzystywanym w testach.

Moduł *adxl345_driver* zrealizowany jest na trzech procesach tj. procesie maszyny stanów, procesie zliczającym impulsy *DataSent*, oraz procesie wyświetlającym na wyświetlaczu LCD odebrane z sensora dane. Działanie modułu sprowadza się do przeczytania zawartości jednego z rejestrów akcelerometru i wyświetlenie go na wyświetlaczu, co odbywa się następująco:

- Gdy moduł dostaje sygnał startowy generowany przez enkoder, wystawia sterownikowi adres urządzenia docelowego, ustawia sygnały *ConTrans* = ‘1’ i *RW* = ‘0’ oraz zapisuje bajt z adresem czytanego rejestru do kolejki FIFO.
- Po przesłaniu przez sterownik bajtu zarówno adresu urządzenia jak i czytanego rejestru, zmienia wyjście *ConTrans* na równe ‘0’ i kończy transmisję, czekając aż sygnał *Busy* przejdzie w stan niski.
- Następnie inicjowana jest transmisja rx, poprzez podanie na wyście *RW* = ‘1’. Podczas niej, akcelerometr prześle zawartość jednobajtowego rejestru, który został zażądany.
- Gdy bajt ten zostanie odebrany transmisja jest kończona, a rezultat zostaje wyświetlony na wyświetlaczu LCD.

Aby przypisać widoczne na schemacie porty I/O (*ang. Input/Output*) do konkretnych wyprowadzeń znajdujących się na płycie należało stworzyć pliki ucf (*ang. User Constraints File*). W projekcie skorzystano z dostępnych na stronie[7] plików, odkomentowując potrzebne wyprowadzenia. Zawartość plików znajduje się na listingach poniżej.

```
# Soldered 50MHz Clock.
NET "Clk" LOC = "C9" | IOSTANDARD = LVTTL | PERIOD = 20.0ns HIGH 50%;
# I2C Bus
NET "SCL" LOC = "D7" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 6;
NET "SDA" LOC = "C7" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 6;
# Rotary encoder
NET "ROT_A" LOC = "K18" | IOSTANDARD = LVTTL | PULLUP;
NET "ROT_B" LOC = "G18" | IOSTANDARD = LVTTL | PULLUP;
# Push-buttons (Press = Hi)
NET "Rst" LOC = "K17" | IOSTANDARD = LVTTL | PULLDOWN;
# Simple LEDs (Hi = On)
NET "FIFO_Empty" LOC = "F9" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 4;
NET "NAck" LOC = "F12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 4;
```

Listing 7 - GenIO.ucf

```
# Character LCD
NET "LCD_E" LOC = "M18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "LCD_RS" LOC = "L18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "LCD_RW" LOC = "L17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
# LCD data connections are shared with StrataFlash connections SF_D<11:8>
NET "LCD_D<0>" LOC = "R15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "LCD_D<1>" LOC = "R16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "LCD_D<2>" LOC = "P17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "LCD_D<3>" LOC = "M15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "SF_CE" LOC = "D16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
```

Listing 8 - LCD.ucf

4.2 Opóźnienia i wykorzystanie zasobów

Poniżej znajduje się wyciąg z raportów podsumowujących syntezę i implementację zaprojektowanego układu, wygenerowanych przez środowisko Xilinx ISE. Przedstawione zostało wykorzystanie zasobów układu FPGA XC3S500E, na którym uruchamiany będzie opisywany projekt, oraz opóźnienia czasowe które wygenerował.

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	188	9,312	2%
Number of 4 input LUTs	298	9,312	3%
Number of occupied Slices	218	4,656	4%
Number of Slices containing only related logic	218	218	100%
Number of Slices containing unrelated logic	0	218	0%
Total Number of 4 input LUTs	332	9,312	3%
Number used as logic	280		
Number used as a route-thru	34		
Number used for Dual Port RAMs	16		
Number used as Shift registers	2		
Number of bonded IOBs	16	232	6%
Number of BUFGMUXs	1	24	4%
Average Fanout of Non-Clock Nets	3.25		

Rysunek 16 - Wykorzystanie zasobów układu FPGA

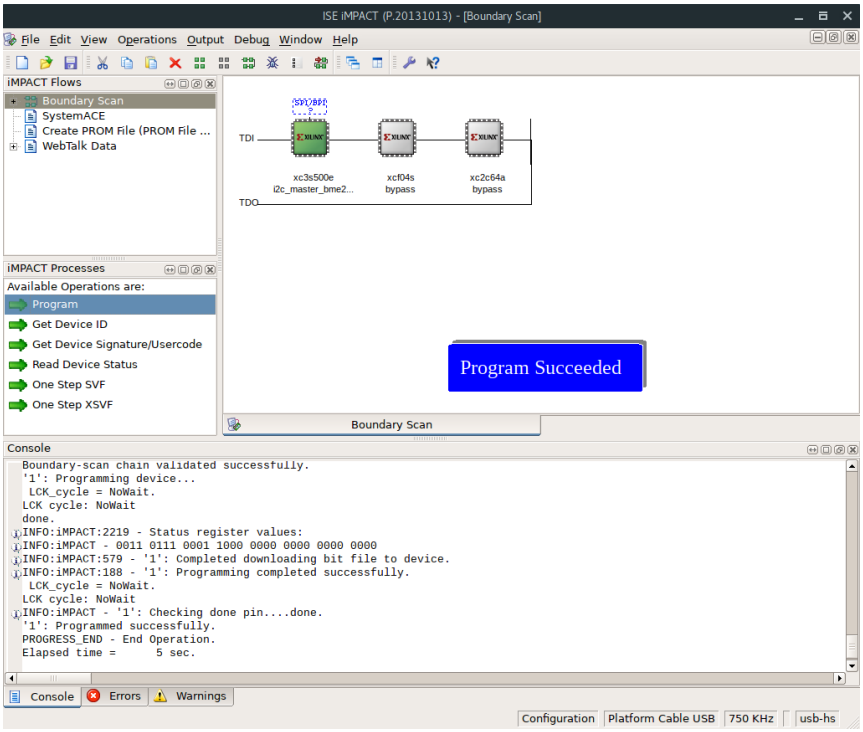
	Met	Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
1	Yes	NET "Clk BUFGP/IBUFG" PERIOD = 20 ns HIGH 50%	SETUPHOLD	11.935ns...	8.065ns	00	00

Rysunek 17 - Opóźnienia powstałe na układzie

Jak wynika z powyższych ilustracji, wykorzystanie zasobów FPGA jest stosunkowo niewielkie, a powstałe opóźnienie wynosi 8.065ns, co w przypadku 20ns okresu źródła zegarowego jest wystarczająco małe. Warto również wspomnieć, że pamięć kolejki opisanej w module *fifo_queue*, zaimplementowana została na LUT (*ang. Look-Up Table)*, co pozwoliło na ograniczenie wykorzystania przerzutników w projekcie. Można to zaobserwować na rysunku 16.

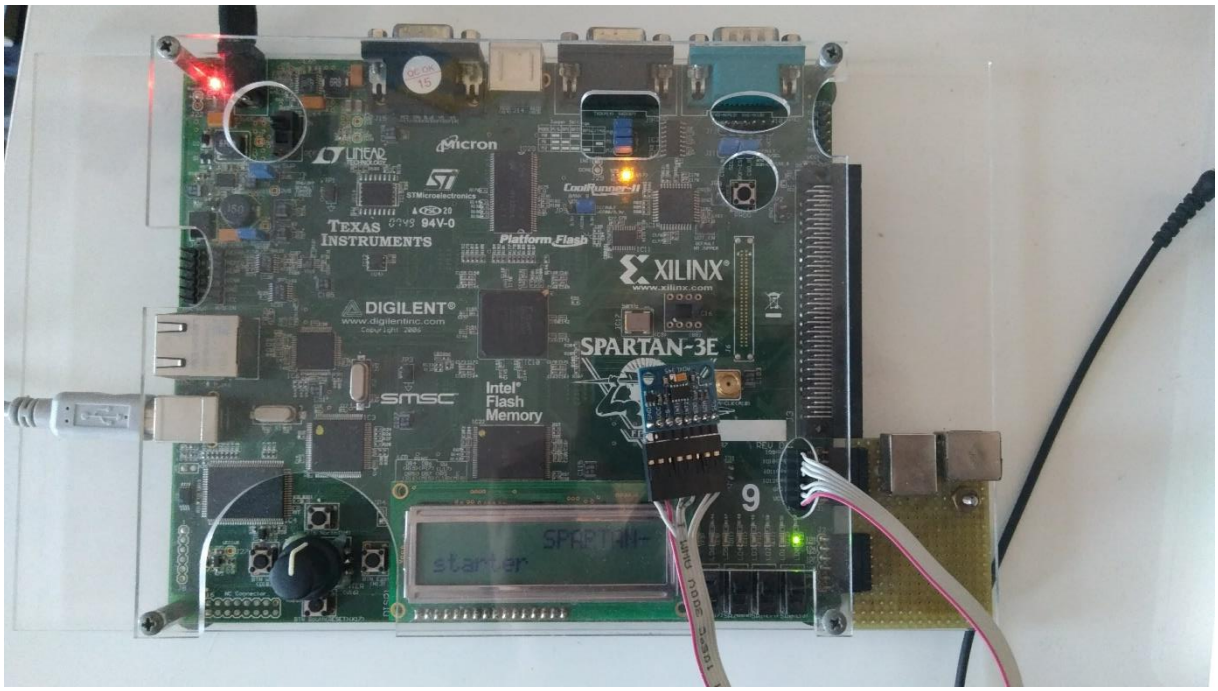
4.3 Testy sprzętowe

Aby zaprogramować układ, wygenerowano za pomocą środowiska Xilinx ISE plik o rozszerzeniu .bit. Następnie za pomocą programu iMPACT znaleziono podłączoną do komputera płytę z układem FPGA i skonfigurowano go z użyciem wygenerowanego wcześniej pliku z projektem. Poniżej znajduje się zrzut ekranu przedstawiający okno programu iMPACT po zaprogramowaniu układu.



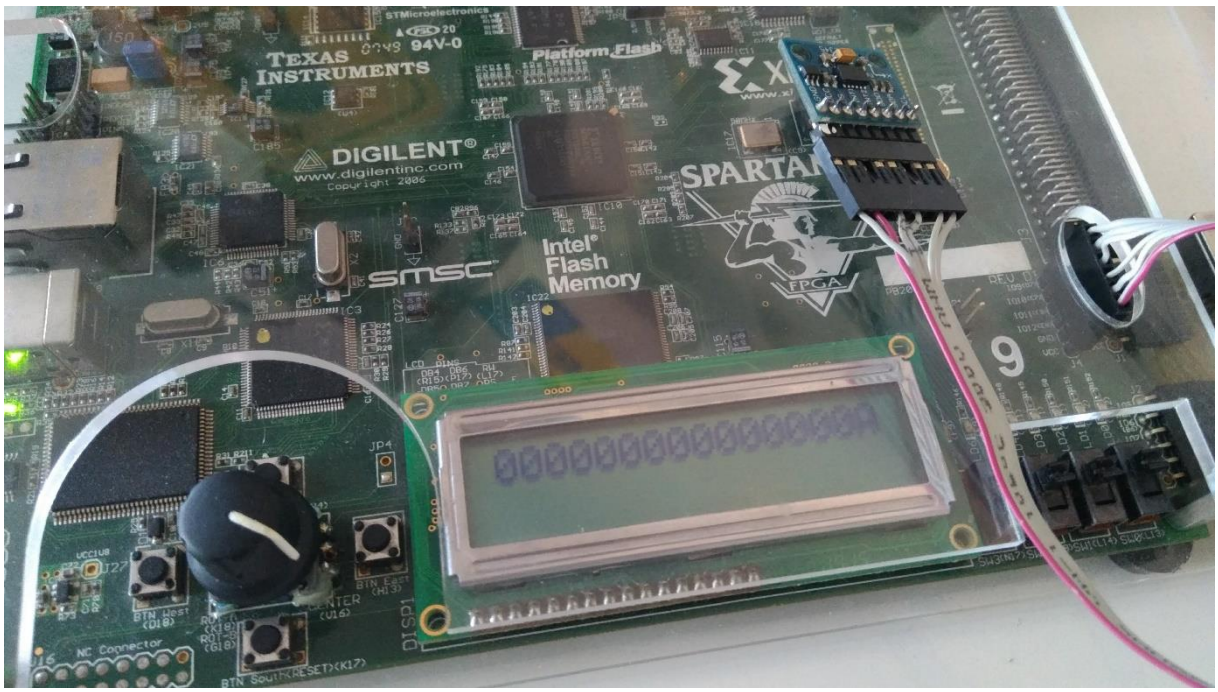
Rysunek 18 - Program iMPACT

Po skonfigurowaniu płyty Spartan-3E Starter Kit został przeprowadzony test sprzętowy, polegający na odczytaniu z podłączonego do wyprowadzeń płyty akcelerometru ADXL345, wartości rejestru o adresie 0x2C. Według dokumentacji domyślna wartość tego rejestru wynosi 0b0000101 binarnie, czyli 0x0A w zapisie heksadecymalnym[3].



Rysunek 19 - Płyta Spartan z podłączonym akcelerometrem

Aby zainicjować działanie sterownika, należało wysłać sygnał startowy do modułu *adxl345_driver*, używając do tego celu pokrętła enkodera. Po przekręceniu pokrętła, na wyświetlaczu LCD wyświetlona została zawartość odebranego rejestru, która zgodnie z dokumentacją wnosiła 0x0A.



Rysunek 20 - Ekran wyświetlacza po przeprowadzeniu testu

Dzięki przetestowanemu w ten sposób projektowi wykazane zostało poprawne działanie stworzonego sterownika magistrali I^2C. Z racji ograniczonego czasu przewidzianego na wykonanie projektu inżynierskiego, nie zostały przeprowadzone bardziej skomplikowane procedury weryfikacyjne z wykorzystaniem sprzętu. Jednakże powyżej opisany test realizuje jeden z celów pracy, jakim była weryfikacja działania sterownika na fizycznym układzie FPGA.

5. Podsumowanie

Realizacja niniejszej pracy pozwoliła autorowi na poszerzenie praktycznej wiedzy z dziedziny projektowania i implementacji złożonych projektów w układach programowalnych. Zrealizowane zostały cele zdefiniowane w rozdziale pierwszym, tj.:

- Opracowany został projekt sterownika, wraz z diagramem maszyny stanów. Poddano go modularyzacji i określono logikę komunikacji pomiędzy poszczególnymi modułami (rozdział 2)
- Projekt został napisany w języku VHDL, a całość została poddana symulacjom podczas których zostało sprawdzone zachowanie modułu sterownika w różnych sytuacjach testowych (rozdział 3).
- Działanie kompletnego projeku zostało przetestowane na fizycznym układzie, co zostało opisane w rozdziale 4.

Praca w obecnej postaci nadaje się do dalszego rozwoju, jest należycie udokumentowana, a kod modułów przejrzyście napisany. Możliwe jest rozszerzenie projektu i przygotowanie go do obsługi przypadków typu multi-master, czyli sytuacji kiedy do magistrali podłączone jest kilka urządzeń nadrzędnych. Wymagałoby to dodanie takich funkcjonalności jak clock stretching, synchronizacja zegarów pomiędzy urządzeniami nadrzędnymi i zaimplementowanie mechanizmów detekcji kolizji. Kolejną możliwością rozwoju projektu jest dodanie 10-bitowego adresowania, co wymagałoby jednak sporego nakładu pracy, zważając na fakt, że obecna maszyna stanów modułu *i2c_master* wymagałaby całkowitego przeprojektowania. Biorąc pod uwagę małą popularność stosowania takiej metody adresacji, poświęcony teoretycznie czas i nakład pracy jest niewspółmierny do potencjalnych zysków wynikających z implementacji tego rozwiązania. Ostatnią rozważaną drogą rozwoju jest zwiększenie szybkości transmisji danych tak, aby sterownik spełniał wyższe standardy zdefiniowane w specyfikacji magistrali I^2C[3]. Jednakże wymagałoby to sprawdzenia, czy opóźnienia powstałe na zmodfikowanym, a następnie zsyntezowanym module, nie wymuszają zastosowania do tego celu innego źródła zegarowego, niż użyte w tej pracy. Sama modyfikacja pociągnęła by za sobą zmiany w procesach korzystających z dotychczasowej wewnętrznej domeny zegarowej.

Bibliografia

[1] Xilinx Inc., *Spartan-3E FPGA Starter Kit Board User Guide (v1.2)*, (https://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf, data dostępu: 08.12.17r.);

[2] Analog Devices, *ADXL345 Digital Accelerometer Data Sheet (rev. E)*, (http://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf, data dostępu: 08.12.17r.);

[3] NXP Semiconductors, *UM10204 I^2C-bus specification and user manual (rev. 6)*, (https://www.nxp.com/docs/en/user-guide/UM10204.pdf, data dostępu: 08.12.17r.);

[4] *VHDL: Standard FIFO*, (http://www.deathbylogic.com/2013/07/vhdl-standard-fifo/, data dostępu: 08.12.17r.);

[5] Mark Zwoliński, *Projektowanie układów cyfrowych z wykorzystaniem języka VHDL*, WKŁ, 2017;

[6] Xilinx Inc., *Spartan-3E FPGA Family Data Sheet (v4.1)*, (https://www.xilinx.com/support/documentation/data_sheets/ds312.pdf, data dostępu: 08.12.17r.);

[7] dr inż. Jarosław Sugier, *Zestawy Digilent S3E-Starter*, (http://www.zsk.ict.pwr.wroc.pl/zsk_ftp/fpga/, data dostępu: 08.12.17r.);

[8] Kevin Skahill, *VHDL for Programmable Logic*, Wydawnictwa Naukowo-Techniczne, 2006;

[9] Pong P. Chu, *RTL hardware design using VHDL*, Wiley-IEEE Press, 2006;

[10] Jerzy Pasierbiński, Piotr Zbysiński, *Układy programowalne w praktyce*, WKŁ, 2002;

[11] Jacek Majewski, Piotr Zbysiński, *Układy FPGA w przykładach*, Wydawnictwo BTC, 2007;