Wrocław, 02.06.17r

Realizacja sterownika magistrali i^2c na układzie FPGA

DOKUMENTACJA PROJEKTOWA

WOJCIECH ADLES 209853

FILIP OLIWA 188592

Spis treści

1. Wstęp

Cel i zakres projektu

Celem projektu było napisanie sterownika magistrali i2c, realizując go na układzie FPGA. Do tego celu użyliśmy języka VHDL, pracując w środowisku Xilinx ISE. Utworzony przez nas sterownik jest przeznaczony dla urządzeń pracujących na magistrali jako master tj. urządzenie nadrzędne, które inicjujące i kończy transmisję. Podczas trwania laboratorium projektowego udało nam się napisać pełną obsługę komunikacji typu master-transmiter, oraz niepełną master-receiver, gdzie nieokreślony jest warunek końca transmisji. Projekt nie został przetestowany z użyciem sprzętu, a jedynie w symulatorze ISim, dzięki testbenchowi zamieszczonym na stronie kursu, który zwracał oczekiwane rezultaty.

Sprzęt

Napisany przez nas kod jest syntezowalny i zostaje implementowany na urządzenie XC3S500E, z rodziny Spartan3E. Tak jak napisano powyżej, układ nie został przetestowany bezpośrednio na sprzęcie, zatem opieramy się na wynikach symulacji przeprowadzonych w programie ISim. Docelowa płyta znajdująca się w laboratorium to Xilinx UG230 Spartan-3E FPGA Starter Kit, zawierająca układ XC3S500E którego głównym źródłem zegarowym jest 50MHz oscylator[1], który użyliśmy jako źródło taktowania naszego sterownika.

Zagadnienia teoretyczne

Magistrala i^{2}c jest magistralą szeregową typu half-duplex, gdzie transmisja danych odbywa się za pomocą dwóch linii: SDA czyli linii danych oraz SCL, linii zegara. Każda z nich jest podciągnięta do zasilania poprzez rezystory pull-up. Wszystkie urządzenia podpięte do magistrali posiadają swój własny, unikalny adres. Podstawowym adresowaniem jest adresowanie 7-bitowe, jednakże istnieje też wersja 10-bitowa, pozwalająca na zaadresowanie do 1024 urządzeń. Urządzenia te podzielone są zgodnie z modelem komunikacji master-slave, gdzie zarówno master jak i slave może nadawać i odbierać dane, jednakże komunikacja jest nawiązywana tylko przez urządzenie typu master, które jest również odpowiedzialne za generowanie impulsów zegarowych na linii SCL. Transmisja danych jest zatem dwukierunkowa, przy czym w dany czasie dane mogą być przesyłane tylko w jedną stronę. 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.

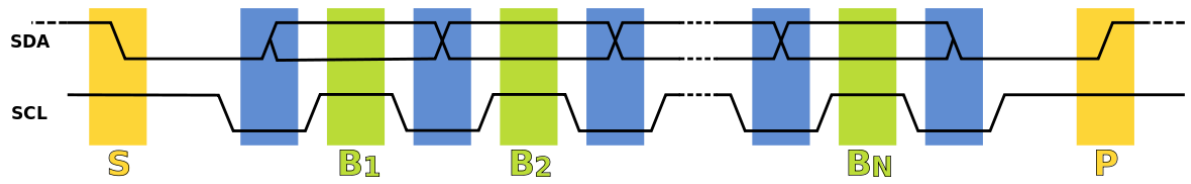
Przechodząc do omówienia warstwy łącza danych, transmisja odbywa się poprzez wysyłanie pojedynczych bajtów, z których każdy z nich jest zakończony bitem potwierdzenia. Transmisja zaczyna się od sygnału START, który polega na zmianie stanu linii SDA z wysokiego na niski, przy jednoczesnym utrzymaniu linii SCL w stanie wysokim. Transmisję terminuje sygnał STOP polegający na analogicznej

[1] https://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf

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, determinującym czy chcemy być nadawcą czy odbiorcą danych (1 - zapis, 0 – odczyt). Po nadaniu ostatniego bitu R/W master zwalnia linię danych, oczekując odpowiedzi od wywoływanego urządzenia. Slave powinien w tym czasie przytrzymać linię SDA w stanie niskim podczas taktu zegara na linii SCL, co rozumiane jest jako nadanie sygnału ACK, potwierdzającego gotowość na nadanie/odbiór danych. Jeśli jednak w tym czasie linia SDA pozostanie w stanie wysokim, urządzenie master uzna to jako brak potwierdzenia (sygnał NACK) i będzie zmuszony zakończyć transmisję sygnałem STOP, bądź nadać sygnał repeated START, ustanawiający kolejną transmisję.

W przypadku gdy bit R/W wynosi 0 (master nadaje), 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 bądź repeated START w przypadku, gdy wysłane zostały wszystkie bajty lub nie dostał potwierdzenia od urządzenia slave.

Dla transmisji typu read (bit R/W = 1), po pierwszym wysłanym przez mastera bajcie adresowym i otrzymaniu potwierdzenia, odbiera on nadawany przez drugie urządzenie dane, potwierdzając odbiór każdego bajtu sygnałem ACK na linii danych. Gdy master będzie chciał przerwać transmisję, nadaje on sygnał NACK podczas trwania taktu zegara przeznaczonego do potwierdzenia i kończy transmisję sygnałem STOP, bądź repeated START.

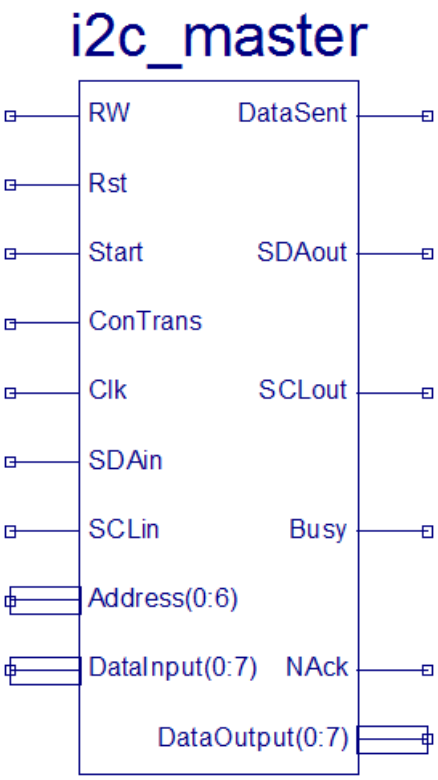


Rysunek 1: Przykładowa transmisja danych

2. Realizacja projektu

Porty I/O

Pierwszym etapem projektowania naszego modułu sterownika jest określenie portów wejścia-wyjścia za pomocą których będzie on komunikował się ze swoim otoczeniem. Poniżej znajduje się schemat modułu wygenerowanego na podstawie kodu VHDL, którego porty następnie omówimy.



Rysunek 2: Schemat portów I/O

Porty wejściowe:

- RW – port na który podawany jest bit R/W który determinuje kierunek transmisji
- Rst – port resetujący moduł do stanu początkowego
- Start – port używany przy inicjacji połączenia
- ConTrans – impuls na tym porcie oznacza kontynuację transmisji po wysłaniu poprzedniego bajtu danych
- Clk – wejście zegarowe
- SDAin – wejście obierające dane z linii SDA
- SCLin – wejście odbierające dane z linii SCL (nie jest używane dla mastera)
- Address – port na który podawany jest wektor zawierający adres urządzenia docelowego
- DataInput – wejście na wysyłany bajt danych

Porty wyjściowe:

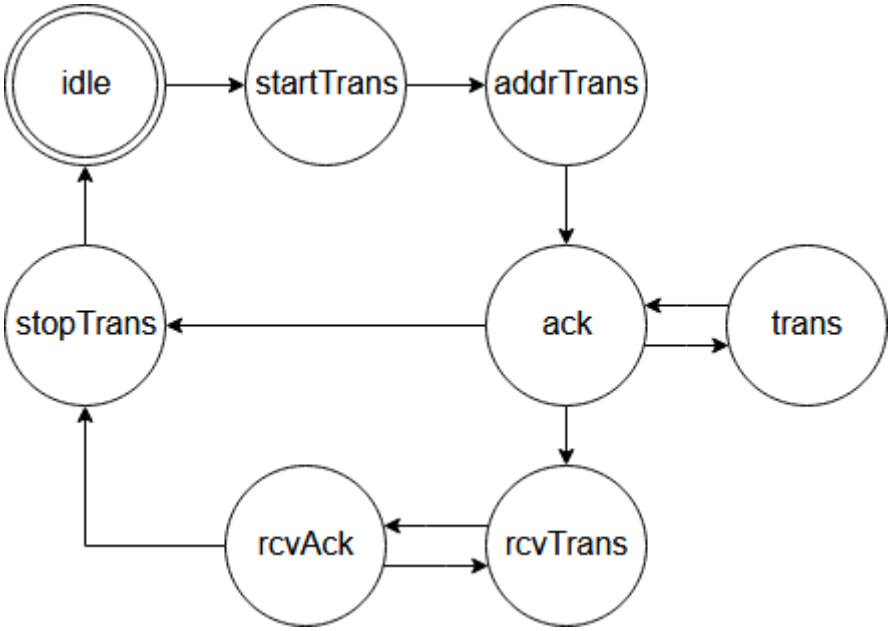
- DataSent – wyjście informujące o wysłaniu bajtu danych
- SDAout – wyjście nadające dane na linię SDA
- SCLout – wyjście na linię SCL
- Busy – wyjśćie informujące o stanie zajetości modułu
- NAck – port używany przy sygnalizacji braku potwierdzenia ze strony odbiorcy
- DataOutput – port na którym zwracany jest odebrany bajt danych

Jak można wywnioskować ze schematu, zdecydowaliśmy się na rozdzielenie wejścia danych które chcemy nadać, od wysyłanego pierwszego bajtu z adresem i bitem R/W, które mają swoje dedykowane wejścia. Moduł posiada również wejście resetujące, które pozwala zmienić jego stan wewnętrzny na „idle". Stosujemy też wejście ConTrans, które próbkujemy podczas odbioru sygnału ACK determinujące kontynuację transmisji następnego bajtu danych.

Jeśli chodzi o wyjścia modułu, zdecydowaliśmy się na trzy wyjścia sygnalizujące: DataSent, Busy oraz NAck. Na pierwszy z nich podawany jest impuls gdy transmisja bajtu zakończy się. Wyjście Busy będzie w stanie niskim podczas gdy sterownik znajduję się w stanie oczekiwania, w przeciwnym przypadku na to wyjście wystawiana zostaje '1'. Wyjście NAck będzie sygnalizowało brak potwierdzenia ze strony urządzenia docelowego, które w naszym przypadku kończy transmisję sygnałem STOP.

Maszyna stanów

Następnym krokiem, było zaprojektowanie maszyny stanów dla naszego modułu sterującego. Oparliśmy się tutaj w dużej mierze o przebieg transmisji odbywającej się za pośrednictwem magistrali i^2c, wyróżniając widoczne na poniższym diagramie stany.



Rysunek 3: Diagram maszyny stanów

Aby lepiej zobrazować warunki przejść pomiędzy poszczególnymi stanami poniżej wstawiliśmy listing z kodem procesu zaimplementowanej przez nas maszyny stanów.

```
-------------------------------------------------State Machine process
FSM: process(state, clkCount, Start, LaRW, dataCount, SDAin, ConTrans)
begin
	nextState <= state;
	case state is
		when idle =>
			CE <= '0';
			if Start = '1' then
				nextState <= startTrans;
			end if;

		when startTrans =>
			CE <= '1';
			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' =>
						if ConTrans = '1' and LaRW = '0' then
							nextState <= trans;
						elsif ConTrans = '1' and LaRW = '1' then
							nextState <= rcvTrans;
						elsif ConTrans = '0' then
							nextState <= stopTrans;
						end if;
					when '1' =>
						NAck <= '1';
						nextState <= stopTrans;
					when others =>
						nextState <= stopTrans;
				end case;
			end if;

		when rcvAck =>
			if ClkCount = "1111100" then
				if ConTrans = '1' then
					nextState <= rcvTrans;
				elsif 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 maszyny stanów

Naszym stanem początkowym jest „idle” czyli stan spoczynku. Jak można zauważyć w powyższym kodzie, automat wychodzi z tego stanu w przypadku gdy na wejście Start modułu zostaje podany impuls informujący o rozpoczęciu transmisji. Zatrzaskiwane są wtedy sygnały Address i RW, a sama maszyna przechodzi do stanu „startTrans”.

W stanie „startTrans” sygnał wewnętrzny CE (którego wykorzystanie jest złą praktyką, której niestety nie zdążyliśmy poprawić) ustawiany jest na ‘1’ co powoduje że proces generujący sygnał zegarowy, taktujący linię SCL jest uaktywniony. W stanie „startTrans” generowany jest również sygnał START, poprzez zmianę linii SDA w stan niski. Po pełnym okresie wewnętrznego zegara clkCount, przechodzimy do stanu „addrTrans”.

Podczas trwania stanu „addrTrans” wysyłany jest pierwszy bajt danych, zawierający adres docelowego urządzenia i bit RW. Gdy cały bajt zostatnie wysłany przechodzimy do stanu „ack”.

W stanie „ack” zwalniamy linię SDA, oczekując na potwierdzenie odebrania wysłanego przez nas bajtu. Jak można zobaczyć w kodzie, pod koniec okresu zegarowego, sprawdzamy stan wejścia SDAin, które informuje nas o odebraniu sygnału Acknowledge bądź NotAcknowledge. W zależności od zastanej na wejściu SDAin odpowiedzi, oraz sygnału ConTrans kontynuujemy (bądź nie) transmisję. Sprawdzamy tutaj również zatrzaśnięty wcześniej bit RW, który determinuje dalszy sposób prowadzenia wymiany danych. Gdy dane wysyłamy przechodzimy do stanu „trans”, gdy natomiast chcemy dane odebrać, przechodzimy do staniu „rcvTrans”. W przypadku braku potwierdzenia lub ‘0’ na wejściu ConTrans, przechodzimy do stanu „stopTrans” który wysyła sygnał STOP kończąc transmisje.

W stanach „trans” i „rcvTrans” transmisja jest kontynuowana, a bajty są odpowiednio wysyłane, bądź odbierane. Po zakończeniu transmisji całego bajtu w stanie „trans”, maszyna przechodzi z powrotem do stanu „ack”.

W przypadku transmisji odbiorczej, przechodzimy do stanu „rcvAck” w którym to nasz sterownik ma za zadanie potwierdzić odebranie bajtu. Niestety ten fragment protokołu transmisji nie został przez nas do końca dopracowany tj. nie posiadamy warunku kończącego w sposób prawidłowy transmisję wysyłając sygnał NAck. Zastosowany jest tutaj mechanizm końca transmisji polegający na próbkowaniu wejścia conTrans i podejmowania na jego podstawie decyzji o kontynuowaniu transmisji, bądź jej zakończeniu nadaniem sygnału STOP. Decyzja ta podejmowana jest pod koniec taktu zegara kiedy to wysłany już został sygnał ACK, który informuje nadawcę o chęci kontynuowania transmisji, przez co nie jest przygotowany na rychłe nadanie przez nas sygnału STOP. Decyzja o kontynuacji transmisji powinna zostać podjęta jeszcze przed wysłaniem sygnału ACK. Najprostszym rozwiązaniem tego problemu jest odgórne określenie liczby odbieranych bajtów w ramach jednej transmisji, co jednak było by mało elastycznym pomysłem.

Wewnętrzny sygnał zegarowy

Jednym z podstawowych problemów przy projektowaniu układu sterującego transmisją na magistrali i^2c jest fakt różnicy taktowania pomiędzy zegarem taktującym linię SCL, a zegarem układu na którym implementujemy nasz projekt. Jako, ze chcemy otrzymać prędkość transmisji danych rzędu 400 kb/s, częstotliwość zegara taktującego linię SCL musi wynosić 400 kHz. Nasz projekt planowaliśmy docelowo zaimplementować na płycie Xilinx UG230 Spartan-3E FPGA Starter Kit z układem XC3S500E, którego główne źródło taktowania ma częstotliwość 50 MHz. Z tego wynika, że na jeden takt zegara

wewnętrznego sterownika przypada 125 taktów zegara o okresie 20 ns. Zatem musieliśmy zaimplementować licznik wewnętrzny, który zapewni właściwą synchronizację procesów w naszym projekcie. Poniżej znajduje się listing z kodem tego procesu zegarowego.

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

Listing 2: Kod procesu zegarowego

Jak widać, licznik wykorzystuje sygnał wewnętrzny clkCount który jest inkrementowany podczas gdy flaga CE = ‘1’. Jak już wcześniej wspominaliśmy, stosowanie takich flag nie jest dobrą praktyką i taki licznik powinien być zależny od maszyny stanów. Niestety, nie zdążyliśmy poprawić tego fragmentu kodu, a z racji braku testów sprzętowych, musimy opierać się na wynikach symulacji, gdzie taki mechanizm nie wykazywał błędnych zachowań.

Generacja sygnału zegarowego na linii SCL

Jednym z głównych procesów synchronizowanych licznikiem ClockCounter jest proces SclDriver, zajmujący się generowaniem impulsów zegarowych na linii SCL. Służy on do synchtonizacji transmisji szeregowej, odbywającej się na magistrali. Jego kod znajduję się w listingu poniżej.

```
-------------------------------------------------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 3: Kod procesu SclDriver

W zależności od stanu w jakim znajduje się aktualnie sterownik oraz od stanu licznika clkCount, proces podaje na wyjście SCLout logiczne ‘1’ bądź ‘0’. Warto tutaj zauważyć, że stan wysoki jest równoważny ze zwolnieniem linii SCL, która znajdzie się wtedy w stanie wysokiej impedancji, co będzie widocznie w czasie symulacji, znajdującej się dalszej części dokumentacji. W powyższym kodzie można zaobserwować również proporcje stanu wysokiego i niskiego na wyjściu SCLout podczas stanów transmisji i potwierdzania danych. Przez pierwsze 72 takty zegara wyjście jest w stanie niskim, podczas pozostałych 53 taktów znajduję się w stanie wysokim.

Obsługa linii danych (SDA)

Obsługą linii danych zajmują się dwa procesy, SdaDriver który obsługuje transmisje wychodzące, generując sygnały na wyjściu SDAout, oraz SdaInDriver który zajmuje się odbiorem danych z wejścia SDAin. Zaczniemy od omówienia obszerniejszego z nich, czyli SdaDriver, którego kod znajduję się poniżej.

```
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 or state = trans then
                        if clkCount = "0100011" then
                              SDAout <= LaDataInput(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
                              SDAout <= '0';
                        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 4: Kod procesu SdaDriver

Jak widać jest on głównie zależny od maszyny stanów, jak również używa licznika clkCount do synchronizacji z sygnałem zegarowym na linii SCL. Można zaobserwować generację sygnałów START i STOP, zwolnienie linii w stanie „ack” jak również generację sygnału ACK, w stanie „rcvAck”. Podczas transmisji danych w stanach „addrTrans” i „trans” na wyjście SDAout podawany jest odpowiedni bit wektora danych, który jest ustalany za pomocą licznika dataCount, który jest sterowany w procesie DataCounter o którym później. Na koniec warto zwrócić uwagę na fakt, że zmiana wyjścia SDAout odbywa się tylko gdy linia SCL jest w stanie niskim, co jest wymagane w przypadku obsługi magistrali i^2c.

Przechodząc do procesu SdaInDriver, wartość na wejściu SDAin pobierana jest podczas, gdy linia SCL jest w stanie wysokim i zapisywana do wektora DataRecieved, z użyciem tego samego licznika danych co w poprzednim procesie. Kod znajduje się w listingu poniżej.

```
-------------------------------------------------SDAin signal driver
SdaInDriver: process(state, clkCount, Clk, dataCount)
   begin
            if rising_edge(Clk) then
                  if state = rcvTrans then

                        if clkCount = "1100000" then

                              DataRecieved(dataCount) <= SDAin;
                        end if;
                  end if;
            end if;
   end process SdaInDriver;
```

Listing 5: Kod procesu SdaInDriver

Licznik bitów danych

W poprzednich procesach korzystaliśmy z licznika wysłanych/odebranych bitów danych, dlatego w tej sekcji postaramy się go przybliżyć. Sama zmienna licznikowa jest zdefiniowana jako integer przyjmujący wartości od -1 do 7, z wartością początkową równą -1. Kod procesu zajmującego się obsługą te zmiennej zamieszczamy poniżej.

```
-------------------------------------------------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 6: Kod procesu DataCounter

Jak widać proces przypisuje zmiennej dataCount wartość -1 w stanach w których dane nie są transmitowane, podczas gdy sterownik znajdzie się w stanach transmisji danych licznik jest inkrementowany na początku taktu zegara wewnętrznego. Licznik ten wykorzystywany jest również w procesie maszyny stanów, gdzie jest warunkiem zmiany stanu po nadaniu/odebraniu wszystkich 8 bitów, które wykrywane są właśnie dzięki tej zmiennej.

Zatrzaski

W naszym sterowniku wykorzystywany jest jeden proces, zatrzaskujący dane na wejściach RW, Address oraz DataInput. Są to wejścia na których podawane są dane które chcemy przetransmitować. W przypadku gdy zaczynamy transmisję zatrzaskiwane są wejścia Address i RW, które przypisywane są do swoich odpowiedników LaAddress i LaRW. Są one następnie przypisywane do uniwersalnego wektora danych LaDataInput który jest później transmitowany poprzez linię SDA. W przypadku wysyłania „zwykłych" bajtów, zatrzaskiwane jest wejście DataInput i bezpośrednio przypisywane do wektora LaDataInput. Kod procesu znajduje się w listingu poniżej.

```
-------------------------------------------Latch for Input Data process
InputLatch: process(Clk, Start, ConTrans, state)
   begin
            if Start = '1' and state = idle then
                  LaAddress <= Address;
                  LaRW <= RW;
                  LaDataInput <= LaAddress & LaRW;
            end if;
            if ConTrans = '1' and state = ack then
                  LaDataInput <= DataInput;
            end if;
   end process InputLatch;
```

Listing 7: Proces odpowiadający za zatrzaski

Sterowanie wyjściem DataSent

Za sterowanie wyjściem DataSent odpowiada jeden prosty proces generujący impuls na tym wyjściu dla stanów „ack" i „rcvAck", na początku okresu zegarowego odpowiedzialnego za potwierdzenie przetransmitowanego przedtem bajtu. Oto jego listing.

```
-------------------------------------------------Ack process
WaitAck: process(state, clkCount, Clk)
   begin
            if rising_edge(Clk) then
                  if state = Ack or state = rcvAck then
                        if clkCount = "0000000" then
                              dataSent <= '1';
                        else
                              dataSent <= '0';
                        end if;
                  else
                        dataSent <= '0';
                  end if;
            end if;
   end process WaitAck;
```

Listing 8: Kod procesu WaitAck

Obsługa wyjścia DataOutput

Ostatnim omawianym przez nas procesem jest proces DataOut, który ma za zadanie wystawianie na wspomniane wyjście odebranych bitów. Robi to w podobnej chwili co proces omawiany poprzednio, a mianowicie gdy sterownik znajduje się w stanie „rcvAck", na samym początku okresu zegarowego przypadającego na tej właśnie stan. Poniżej znajduje się prosty kod tego procesu.

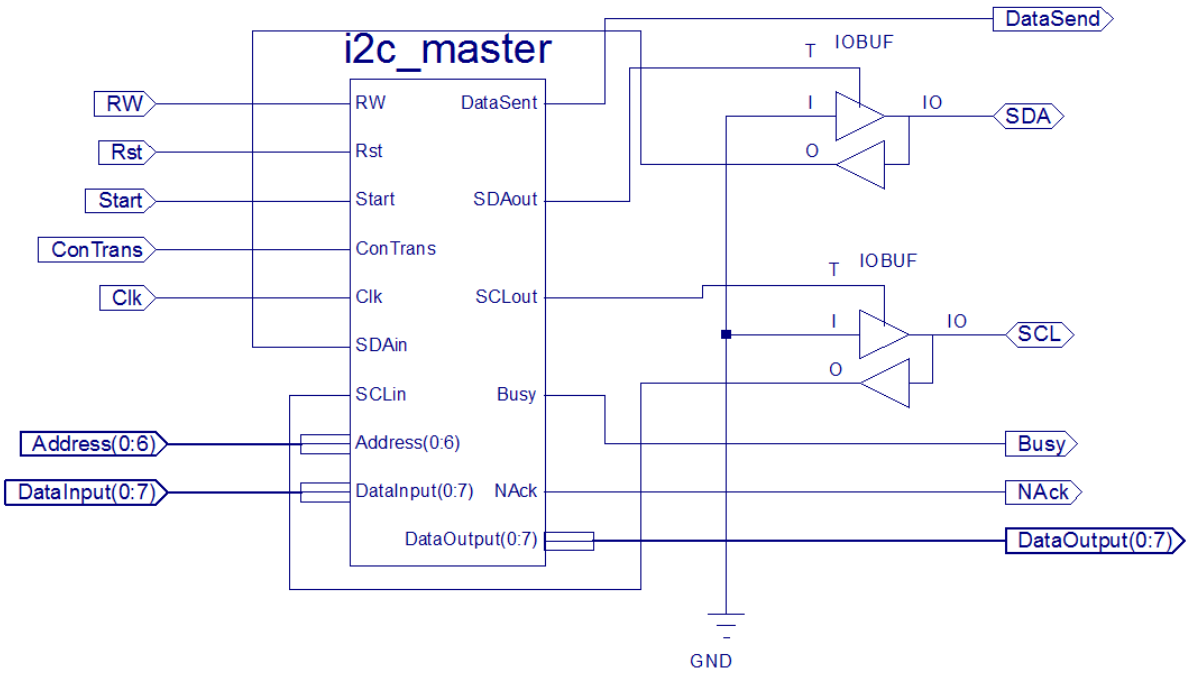
```
-------------------------------------Process for presenting recieved data

DataOut: process(Clk, state, clkCount)
      begin
            if rising_edge(Clk) then
                  if state = rcvAck and clkCount = "0000000" then
                        DataOutput <= DataRecieved;
                  end if;
            end if;
      end process DataOut;
```

Listing 9: Kod procesu DataOut

Schemat podłączenia modułu sterownika

Przed rozpoczęciem symulacji należało odpowiednio połączyć wyjścia na schemacie głównym. Z racji dwukierunkowości linii SDA i SCL byliśmy zmuszeni podczas pisania kodu, do odwoływania się do portów wejściowych i wyjściowych tych linii, które jednak fizycznie występują jako jedność. Dlatego wymagane było zastosowanie buforów I/O, które są zbudowane z wykorzystaniem bramek trójstanowych. To właśnie one będą symulowały dodatkowy stan wysokiej impedancji (Z). Resztę portów wejścia wyjścia podłączono bezpośrednio. Poniżej znajduje się schemat który był testowany z użyciem symulatora ISim.



Rysunek 4: Schemat wykorzystany w symulacji

Symulacja w programie ISim

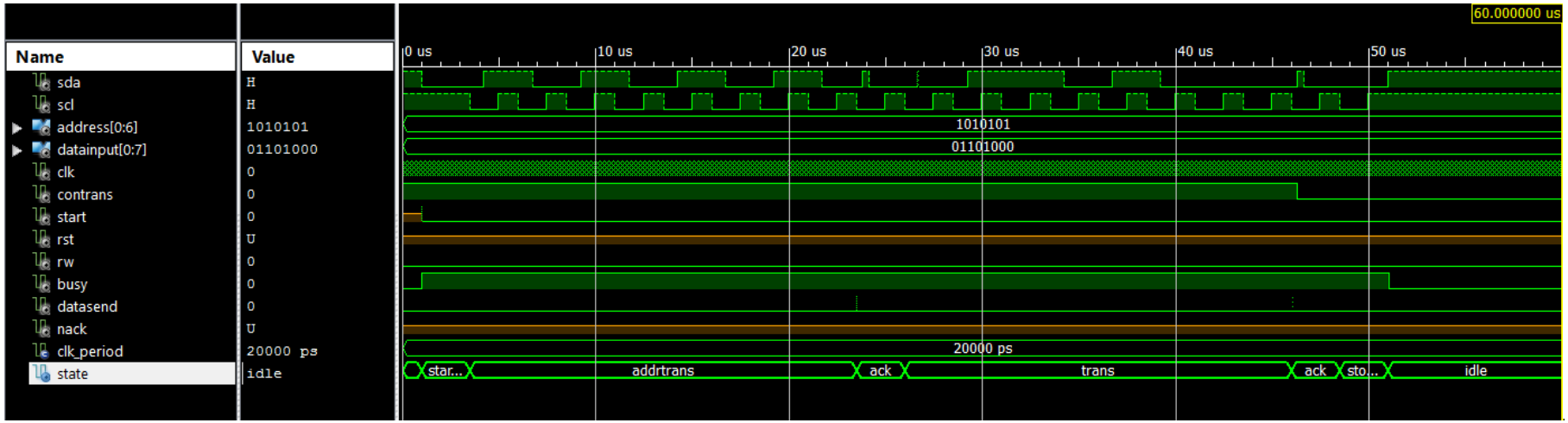
Tak jak już poprzednio wspominaliśmy, działanie naszego projektu przetestowaliśmy wyłącznie za pomocą symulatora. Początkowo moduł testowaliśmy za pomocą własnoręcznie napisanego testbencha, jednakże w końcowej fazie projektu użyliśmy do tego celu pliku znajdującego się na stronie[2]. Istotny fragment kodu testującego znajduje się poniżej, dla symulacji zapisu zmieniona została jedynie wartość wejścia RW na '1'.

```
Address <= "1010101";
      RW <= '0';
      Start <= '1' after 1000 ns, '0' after 1020 ns;
      ConTrans <= '1', '0' after 46.3 us;
      DataInput <= "01101000";
```

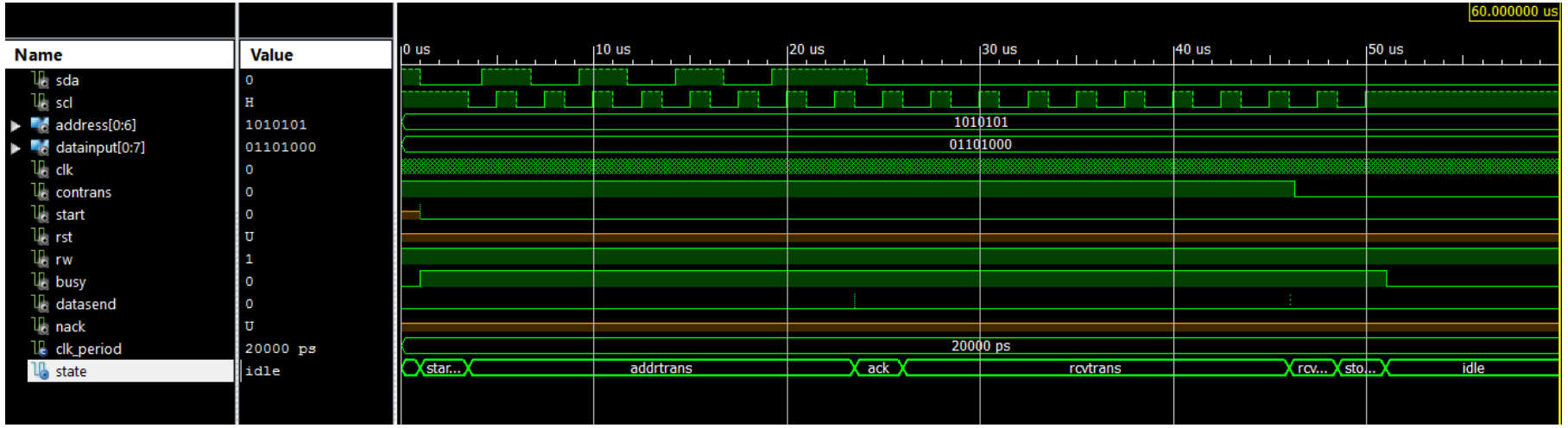
Listing 10: Fragment kodu symulacji

Poniżej znajdują się dwa zrzuty ekranu przedstawiające symulacje nadawania i odbierania danych za pomocą modułu naszego sterownika.

[2] http://www.zsk.ict.pwr.wroc.pl/zsk_ftp/fpga/Tbw_I2Cslave.zip



Rysunek 5: Symulacja transmisji nadawczej



Rysunek 6: Symulacja transmisji odbiorczej

Poniżej znajdują się odpowiedzi dla kolejno transmisji nadawczej i odbiorczej.

```
ISim>
# run 60.00us
Simulator is doing circuit initialization process.
Finished circuit initialization process.
[I2C 1.03 us] START condition
[I2C 23.51 us] address byte: AA
[I2C 46.01 us] byte received: 68
[I2C 51.01 us] STOP condition
ISim>
```

Rysunek 7: Odpowiedź dla nadawania

```
ISim>
# run 60.00us
Simulator is doing circuit initialization process.
Finished circuit initialization process.
[I2C 1.03 us] START condition
[I2C 23.51 us] address byte: AB
[I2C 49.345 us] byte transmitted: 00 with positive ACK
ISim>
```

Rysunek 8: Odpowiedź dla odbioru

Widać tutaj, że dla transmisji nadawczej program wykrył odpowiednio każdy aspekt przesyłanych danych, począwszy od sygnału START, poprzez nadane bajty, aż do sygnału STOP.

Natomiast w przypadku transmisji odbiorczej po transmisji ostatniego bajtu, został nadany sygnał ACK, przez co zakończenie transmisji sygnałem STOP nie zostało rozpoznane, ponieważ musi być ono poprzedzone sygnałem NACK, co z wyjaśnionych wcześniej przyczyn nie było możliwe.

3. Zakończenie

Design summary

Poniżej znajdują się informacje dotyczące implementacji przez środowisko ISE naszego projektu na konkretny model układu, w tym wypadku wspomnianą wcześniej płytę, używaną na laboratorium. Mówią one o wykorzystaniu przez implementacje naszego projektu zasobów sprzętowych danego układu.

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Total Number Slice Registers	50	9,312	1%
Number used as Flip Flops	34		
Number used as Latches	16		
Number of 4 input LUTs	97	9,312	1%
Number of occupied Slices	71	4,656	1%
Number of Slices containing only related logic	71	71	100%
Number of Slices containing unrelated logic	0	71	0%
Total Number of 4 input LUTs	97	9,312	1%
Number of bonded IOBs	33	232	14%
IOB Latches	1		
Number of BUFGMUXs	2	24	8%
Average Fanout of Non-Clock Nets	3.04		

Rysunek 9: Wykorzystanie zasobów sprzętowych przez nasz projekt

Z raportu generowanego przez środowisko można również wydobyć informacje na temat maksymalnego opóźnienia powstałego na wygenerowanym układzie, w naszym przypadku jest ono rzędu prawie 6 ns.

	Met	Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
1	Yes	Autotimespec constraint for clock net Start_IBUF	SET...	1.210ns	1.889ns	0	00
2	Yes	Autotimespec constraint for clock net Clk_BUFGP	SET...	0.860ns	5.816ns	0	00

Rysunek 10: Opóźnienia powstałe na układzie

Podsumowanie

Podczas realizacji projektu na przestrzeni semestru koncepcja budowy sterownika magistrali i^2c zmieniała się wielokrotnie. Było to spowodowane w dużej części słabym rozeznaniem w paradygmatach programowania sprzętowego układów FPGA. Pomimo przygotowania do projektu w postaci laboratorium mającego miejsce w poprzednim semestrze, dużo niejasnych rzeczy „wychodziło" dopiero teraz. Jednakże cały ten czas był bardzo pomocny w zrozumieniu jak zachowują się układy cyfrowe, których programowanie diametralnie różni się od programowania w warstwie aplikacji.

Moduł sterownika, który udało nam się zaprojektować, nie implementuje wielu opcji które przewiduje protokół transmisji i^2c, takich jak np. clock streching, czy 10-bitowe adresowanie. Jest on jednak dla nas całkiem satysfakcjonującym osiągnięciem biorąc pod uwagę ograniczony czas jaki przypadł realizacji tego projektu. Na pewno brakuje tutaj przetestowania sterownika na sprzęcie, czy też dopracowania trybu master-reciever. Mimo wszystko wynieśliśmy z tych zajęć sporą wiedzę i doświadczenie, zdobytą podczas praktycznego zaprojektowania modułu „od zera".

Bibliografia

1. https://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf
2. http://www.nxp.com/documents/user_manual/UM10204.pdf