Wydział Elektroniki
i Technik Informacyjnych

Politechnika Warszawska

PUF — Programowanie Układów FPGA

Symulacja implementacji interfejsu SPI i modułu akcelerometru ADXL345 na FPGA Cmod S7

CMOD + ADXL345 + SPI

Wiktor Chocianowicz
318501

Michał Jędrzejczyk
318519

29 KWIETNIA 2024

Spis treści

1 Informacje wstępne

1.1 Opis projektu

Opisywana część projektu zakłada zaimplementowanie funkcjonalnego interfejsu SPI, w tym wszystkich wymaganych przez interfejs modułów, jak również zaimplementowanie symulatora wybranego pmoda akcelerometru. Celem, pozaimplementacyjnym, jest wykonanie odpowiednich symulacji i określenie poprawności przebiegu przesyłu danych między modułami.

1.2 Zakładana funkcjonalność

Zakładamy, że wgrany na FPGA kod poskutkuje prawidłowym odbiorem danych z wybranego pmoda, które następnie będzie można wyświetlić na konsoli debugującej ILA w środowisku Vivado.

1.3 Podział zadań

Tabela 1: Zaimplementowane modułu i odpowiadający im wykonawca.

Moduł	Wykonawca
SPImaster	Wiktor Chocianowicz
PmodACL	Michał Jędrzejczyk
SPIconverter	Michał Jędrzejczyk
debouncer	Wiktor Chocianowicz
top	Michał Jędrzejczyk
testbench	Wiktor Chocianowicz

1.4 Wstęp teoretyczny

1.4.1 Zaimplementowany interfejs

W projekcie wykorzystany został interfejs SPI w trybie 4-Wire o polaryzacji CPOL = 1 i fazie CPHA = 1. Zasada działania interfejsu opiera się na dwóch rejestrach przesuwnych — odbioru i przesyłu. Proces rozpoczyna się wraz z ustaleniem wartości 0 na pinie CS. Dane aktualizowane są na zboczu opadającym i próbkowane na zboczu narastającym. Pierwsze 8 bitów wystawianych jest przez moduł mastera na linię MOSI, pierwszy, zaczynając od MSB, to bit zapisu lub odczytu z rejestru, drugi to wybór transmisji „multi-byte", następne 6 bitów przeznaczone jest dla adresu rejestru. W zależności od wybranego trybu następuje zapis kolejnych 8 bitów do rejestru wysyłanych po linii MOSI lub odczyt 8 bitów z linii MISO, obsługiwanej przez pmod. Koniec transmisji zawiadamia ustalenie wartości 1 na pinie CS.

Zaimplementowany moduł mastera odczytuje dane z pmoda cyklicznie od zerowego rejestru dla osi x, po pierwszy rejestr osi z. Po zakończeniu proces powtarza się dla każdego rejestru. Pmod nie jest konfigurowany przed rozpoczęciem procesu czytania, co oznacza, że w rejestrach konfiguracyjnych znajdują się domyślne wartości, które można odczytać z karty katalogowej akcelerometru.

Planowo interfejs zaimplementowany jest na częstotliwość 2,5 MHz.

1.4.2 Schemat ramki danych

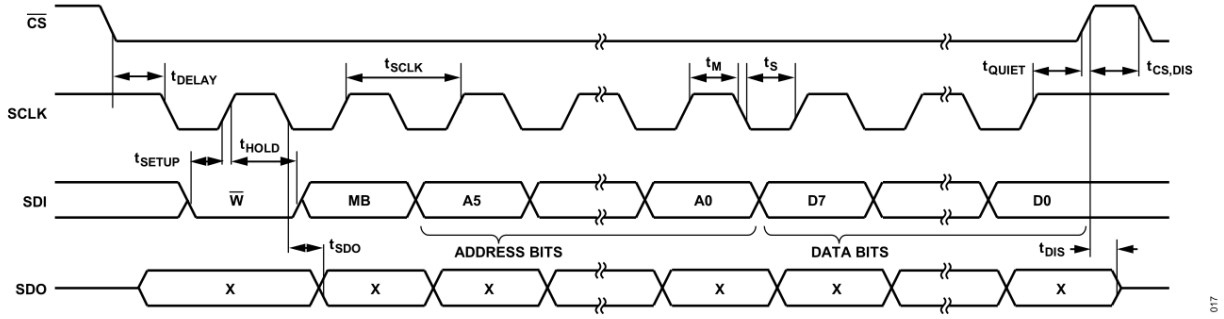


Figure 37. SPI 4-Wire Write

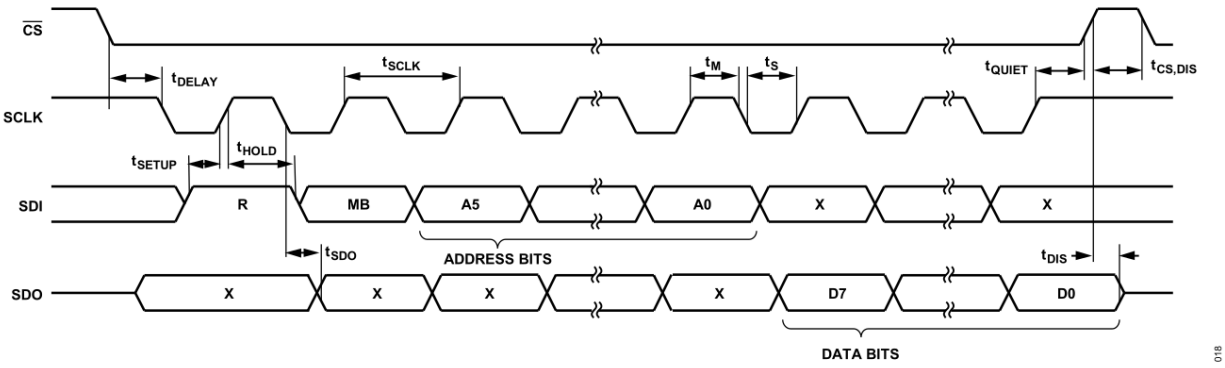


Figure 38. SPI 4-Wire Read

Rysunek 1: Ramka danych z karty katalogowej akcelerometru, s. 15. `https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf`

Jedna ramka przesyłu danych składa się z 16 zboczy narastających i opadających zegara `sclk`. Przykładową ramkę danych zaimplementowanego interfejsu można zobaczyć na Rys. (9).

1.4.3 Chronometraż

Table 10. SPI Timing (T_A = 25°C, V_S = 2.5 V, $V_{DD\ I/O}$ = 1.8 V)[1]

	Limit[2, 3]			
Parameter	**Min**	**Max**	**Unit**	**Description**
f_{SCLK}		5	MHz	SPI clock frequency
t_{SCLK}	200		ns	1/(SPI clock frequency) mark-space ratio for the SCLK input is 40/60 to 60/40
t_{DELAY}	5		ns	$\overline{CS}$ falling edge to SCLK falling edge
t_{QUIET}	5		ns	SCLK rising edge to $\overline{CS}$ rising edge
t_{DIS}		10	ns	$\overline{CS}$ rising edge to SDO disabled
$t_{CS,DIS}$	150		ns	$\overline{CS}$ deassertion between SPI communications
t_S	0.3 × t_{SCLK}		ns	SCLK low pulse width (space)
t_M	0.3 × t_{SCLK}		ns	SCLK high pulse width (mark)
t_{SETUP}	5		ns	SDI valid before SCLK rising edge
t_{HOLD}	5		ns	SDI valid after SCLK rising edge
t_{SDO}		40	ns	SCLK falling edge to SDO/SDIO output transition
t_R[4]		20	ns	SDO/SDIO output high to output low transition
t_F[4]		20	ns	SDO/SDIO output low to output high transition

[1] The $\overline{CS}$, SCLK, SDI, and SDO pins are not internally pulled up or down; they must be driven for proper operation.

[2] Limits based on characterization results, characterized with f_{SCLK} = 5 MHz and bus load capacitance of 100 pF; not production tested.

[3] The timing values are measured corresponding to the input thresholds (V_{IL} and V_{IH}) given in Table 9.

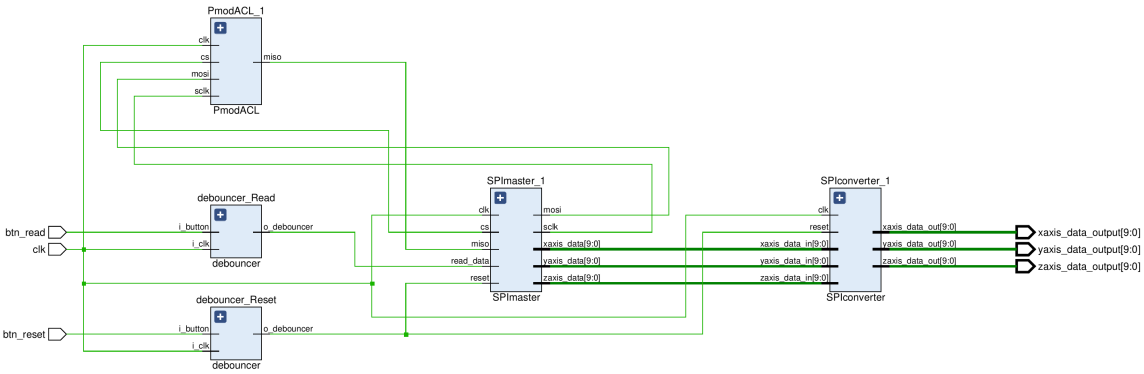
[4] Output rise and fall times measured with capacitive load of 150 pF.

Rysunek 2: Chronometraż z karty katalogowej akcelerometru, s. 16. `https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf`

1.5 Środowisko

Projekt został napisany z użyciem środowiska **Visual Studio Code** i kompilatora **ghdl** na platformach Windows i Ubuntu Linux. Końcowe symulacje przeprowadzone zostały na platformie Linux z użyciem **GTKWave**. Do uzyskania schematu blokowego i blokowej reprezentacji modułów wykorzystane zostało środowisko **Xilinx Vivado**.

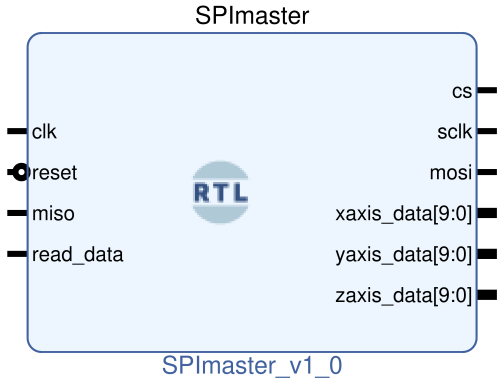
2 Schemat blokowy



Rysunek 3: Schemat blokowy układu.

3 Opisy modułów

3.1 SPImaster



Rysunek 4: Blokowe przedstawienie modułu.

3.1.1 Funkcjonalność

Moduł, poza implementacją funkcjonalności mastera, implementuje również interfejs SPI. Komponent steruje odczytem danych z rejestrów pmoda, generuje sygnał sclk, zarządza procesami nadawania i odczytu danych z linii mosi i miso. Zebrane dane w jednostkach LSB kierowane są do syntezowalnego modułu konwertera.

3.1.2 Porty

```
21  port (
22    -- SPI signals
23    clk         : in STD_LOGIC;                                    -- Clock signal
24    reset       : in STD_LOGIC;                                    -- Reset signal
25    cs          : out STD_LOGIC := '1';                            -- Chip select
      signal
26    sclk        : out STD_LOGIC := '1';                            -- Serial clock
      signal
27    mosi        : out STD_LOGIC := '0';                            -- Master Out Slave
       In signal
28    miso        : in STD_LOGIC;                                    -- Master In Slave
      Out signal
29    read_data   : in STD_LOGIC;                                    -- Start reading
      data from adxl
30    xaxis_data : out STD_LOGIC_VECTOR(9 downto 0) := (others => '0'); -- X axis data
31    yaxis_data : out STD_LOGIC_VECTOR(9 downto 0) := (others => '0'); -- Y axis data
32    zaxis_data : out STD_LOGIC_VECTOR(9 downto 0) := (others => '0')  -- Z axis data
33  );
```

3.1.3 Implementacja

Jednym z procesów mastera jest generacja sygnału sclk wykorzystywanego jako zegar interfejsu. Proces implementuje prosty prescaler.

```
173             -- What to do in running state
174             if r_sclk_counter = PRESCALER then
175               r_sclk         <= not r_sclk;
176               r_sclk_counter <= (others => '0');
177             else
178               r_sclk_counter <= r_sclk_counter + '1';
179             end if;
```

Skrypt 1: Generacja sclk.

Do poprawnego działania interfejsu wymagana jest również poprzednia, względem ostatniego cyklu zegara clk, wartość sygnału sclk:

```
181         r_sclk_prev <= r_sclk;
```

Wartość prescalera jest parametryzowalna i domyślnie przyjmuje wartość pozwalającą na uzyskanie zegara 2,5 MHz.

```
18    PRESCALER     : STD_LOGIC_VECTOR(7 downto 0) := X"28"; -- 2.5 MHz (Int. clock over
      100 MHz)
```

Zegar generowany jest tylko w momencie ustalenia 0 na cs. W pozostałych przypadkach zegar sclk ustalony jest na wartość 1.

```
156           when idle =>
157             -- Go to running state when
158             if cs = '0' then
159               SCLK_STATE <= running;
160             end if;
```

Proces wysyłania danych zaimplementowany jest w zależności od opadającego zbocza zegara `sclk` i ilości zliczonych zboczy opadających. Dane wysyłane są na linię `mosi` z ostatniego bitu bufora, który następnie zostaje przesunięty poprzez dodanie 0 na końcu wektora. Po zakończeniu procesu wysyłania zwracana jest flaga `r_transmit_done`.

```
220             -- What to do in transmitting state
221             if r_sclk_prev = '1' and r_sclk = '0' and r_falling_edge_counter <
        NR_OF_EDGES then
222               mosi                  <= transmit_buffer(15);
223               transmit_buffer       <= transmit_buffer(14 downto 0) & '0';
224               r_falling_edge_counter <= r_falling_edge_counter + '1';
225             elsif r_falling_edge_counter = NR_OF_EDGES then
226               r_transmit_done <= '1';
227             end if;
```

Proces odbioru danych zaimplementowany jest w zależności od narastającego zbocza zegara `sclk` i ilości zliczonych zboczy narastających. Dane odbierane są z linii `miso` do zerowego bitu bufora poprzez dodanie odebranego bitu na koniec wektora. Po zakończeniu procesu odbioru zwracana jest flaga `r_read_done`.

```
267             -- What to do in receiving state
268             if r_sclk_prev = '0' and r_sclk = '1' and r_rising_edge_counter <= X"7" then
269               r_rising_edge_counter                                          <=
        r_rising_edge_counter + '1';
270             elsif r_sclk_prev = '0' and r_sclk = '1' and r_rising_edge_counter > X"7"
        then
271               receive_buffer       <= receive_buffer(14 downto 0) & miso;
272               r_rising_edge_counter <= r_rising_edge_counter + '1';
273             elsif r_rising_edge_counter = NR_OF_EDGES then
274               r_read_done <= '1';
```

Proces odbioru jest również uzależniony od aktualnego cyklu. Dany cykl informuje o aktualnie odczytywanym rejestrze. Po zakończeniu odczytu w bloku `case` zawartość odczytanego bufora kopiowana jest do buforów wyjściowych do konwertera.

```
276               case DATA_CYCLE is
277                 when X"0" =>
278                   if r_can_copy_data = '0' then
279                     xaxis_data(7 downto 0) <= receive_buffer(7 downto 0);
280                   end if;
281                   r_can_copy_data <= '1';
282
283                 when X"1" =>
284                   if r_can_copy_data = '0' then
285                     xaxis_data(9 downto 8) <= receive_buffer(1 downto 0);
286                   end if;
287                   r_can_copy_data <= '1';
```

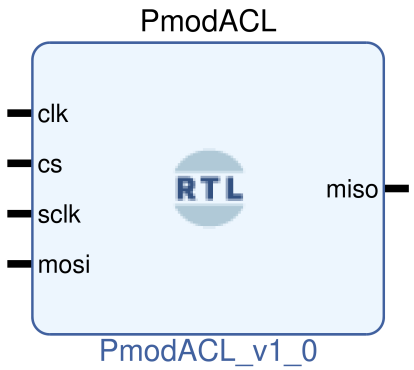
Wybór odczytywanego rejestru, jak również obsługa linii `cs` rozpoczynającej proces przesyłu danych obsługiwana jest poprzez oddzielny proces. Odczyt danych jest zależny od

flagi sterowanej przez zewnętrzny przycisk. W momencie ustalenia flagi proces ustawia cs na 0 i czeka na zgłoszenie flag przez procesy odczytu i nadawania. Wówczas cs ustawiane jest na 1, odczekiwany jest minimalny czas pomiędzy transmisją danych i ponownie uruchamiany jest proces, tym razem dla innego rejestru do odczytu danych.

```
84  -- Read data process
85  read_acl_process : process (clk)
86  begin
87    if rising_edge(clk) then
88      if reset = '1' then
89        cs          <= '1';
90        READ_STATE <= idle;
91      else
92        case READ_STATE is
93          when idle =>
94            if read_data = '1' then
95              READ_STATE <= reading;
96            end if;
97            cs <= '1';
98
99            case DATA_CYCLE is
100             when X"0" =>
101               r_transmit_buffer <= DATAX0;
102
103             when X"1" =>
104               r_transmit_buffer <= DATAX1;
105
106             when X"2" =>
107               r_transmit_buffer <= DATAY0;
108
109             when X"3" =>
110               r_transmit_buffer <= DATAY1;
111
112             when X"4" =>
113               r_transmit_buffer <= DATAZ0;
114
115             when X"5" =>
116               r_transmit_buffer <= DATAZ1;
117
118             when others =>
119               r_transmit_buffer <= DATAX0;
120           end case;
121
122         when reading =>
123           if r_delay_counter = DELAY_COUNTER then
124             cs              <= '0';
125             READ_STATE      <= waiting;
126             r_delay_counter <= (others => '0');
127           else
128             r_delay_counter <= r_delay_counter + '1';
129           end if;
130
131         when waiting =>
132           if r_transmit_done = '1' and r_read_done = '1' then
133             READ_STATE <= idle;
134             if DATA_CYCLE = X"5" then
135               DATA_CYCLE <= (others => '0');
136             else
137               DATA_CYCLE <= DATA_CYCLE + '1';
138             end if;
```

```
139             end if;
140         end case;
141       end if;
142     end if;
143   end process;
```

3.2 PmodACL



Rysunek 5: Blokowe przedstawienie modułu.

3.2.1 Funkcjonalność

Moduł PmodACL jest emulatorem akcelerometru, z którym docelowo komunikujemy się. Jest on połączony przez interfejs SPI z modułem SPImaster, odbiera on adresy rejestrów od modułu SPImaster i zwraca ustawione wartości dla danych osi.

3.2.2 Porty

```
16  port (
17    clk  : in STD_LOGIC;         -- Emulator clock signal 100MHz
18    cs   : in STD_LOGIC;         -- Chip select signal
19    sclk : in STD_LOGIC;         -- Serial clock signal 2,5MHz
20    mosi : in STD_LOGIC;         -- Master Out Slave In signal
21    miso : out STD_LOGIC := '0' -- Master In Slave Out signal
22  );
```

Port clk jest tutaj implementowany wyłącznie na potrzeby symulacyjne, w rzeczywistości akcelerometr ADXL345 nie posiada takiego wyprowadzenia.

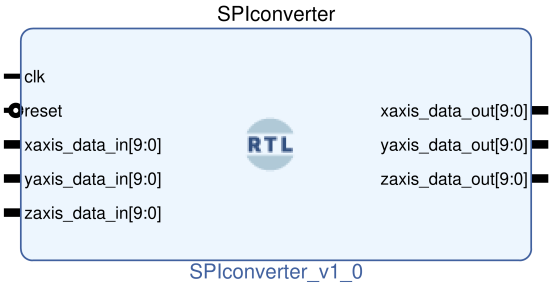
3.2.3 Implementacja

Moduł posiada jeden proces, w którym, z wykorzystaniem bloku `case`, odbiera pierwsze 8 bitów z linii `mosi`, a następnie, w zależności od odebranego adresu, wstawia do bufora wychodzącego odpowiednie dane. Ponieważ w projekcie nie wykorzystywana jest opcja zapisu do rejestrów akcelerometru, w przypadku odebrania ciągu bitów nieoznaczających odbioru danych z rejestrów poszczególnych osi, zgłaszana jest wartość 01 flagi `status`.

```
58  -- Send data process
59  process (clk)
60  begin
61    if rising_edge(clk) then
62      r_sclk <= sclk;
63
64      case STATE is
65        when idle =>
66          -- Go to receiving state when
67          if cs = '0' then
68            STATE <= receiving;
69          end if;
70
71          -- What to do in idle state
72          recieve_buffer  <= (others => '0');
73          miso            <= '0';
74          r_can_copy_data <= '0';
75          status <= "00";
76
77        when receiving =>
78          -- Back to idle state when
79          if cs = '1' then
80            STATE <= idle;
81          end if;
82
83          -- What to do in receiving state
84          if r_sclk = '0' and sclk = '1' and r_rising_edge_counter <= 7 then
85            recieve_buffer                                          <= recieve_buffer(6
    downto 0) & mosi;
86            r_rising_edge_counter                                   <=
    r_rising_edge_counter + '1';
87
88            -- Go to transmitting state when
89          elsif r_rising_edge_counter > 7 then
90            r_rising_edge_counter <= "0000";
91            STATE                 <= transmitting;
92          end if;
93
94        when transmitting =>
95          -- Back to idle state when
96          if cs = '1' then
97            STATE <= idle;
98
99            -- What to do in transmitting state
100         else
101           case recieve_buffer is
102             when DATAX0 =>
103               if r_can_copy_data = '0' then
104                 transmit_buffer <= TO_BE_SENT_DATAX0;
105               end if;
106               r_can_copy_data <= '1';
```

```
107
108             when DATAX1 =>
109               if r_can_copy_data = '0' then
110                 transmit_buffer <= TO_BE_SENT_DATAX1;
111               end if;
112               r_can_copy_data <= '1';
113
114             when DATAY0 =>
115               if r_can_copy_data = '0' then
116                 transmit_buffer <= TO_BE_SENT_DATAY0;
117               end if;
118               r_can_copy_data <= '1';
119
120             when DATAY1 =>
121               if r_can_copy_data = '0' then
122                 transmit_buffer <= TO_BE_SENT_DATAY1;
123               end if;
124               r_can_copy_data <= '1';
125
126             when DATAZ0 =>
127               if r_can_copy_data = '0' then
128                 transmit_buffer <= TO_BE_SENT_DATAZ0;
129               end if;
130               r_can_copy_data <= '1';
131
132             when DATAZ1 =>
133               if r_can_copy_data = '0' then
134                 transmit_buffer <= TO_BE_SENT_DATAZ1;
135               end if;
136               r_can_copy_data <= '1';
137
138             when others =>
139               status          <= "01";
140               r_can_copy_data <= '1';
141           end case;
142           if r_sclk = '1' and sclk = '0' then
143             miso            <= transmit_buffer(7);
144             transmit_buffer <= transmit_buffer(6 downto 0) & '0';
145             if transmit_buffer = X"0" then
146               status <= "10";
147             end if;
148           end if;
149         end if;
150       when others =>
151         status <= "01";
152     end case;
153   end if;
154 end process;
```

3.3 SPIconverter



Rysunek 6: Blokowe przedstawienie modułu.

3.3.1 Funkcjonalność

Zadaniem modułu jest konwersja otrzymanych danych z akcelerometru z U2 na binarne z bitem znaku. SPIconverter otrzymuje dane pomiarowe bezpośrednio od modułu SPImaster zaraz po zakończeniu transmisji, po wykonaniu konwersji zwraca dane na wyjściu modułu top.

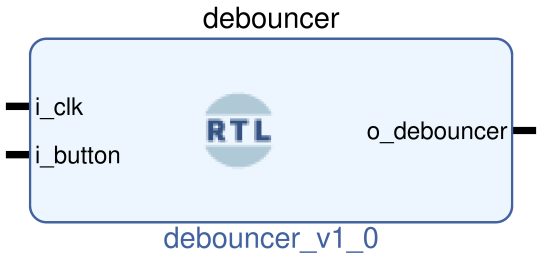
3.3.2 Porty

```
17  port (
18    clk            : in STD_LOGIC;                                    -- Clock signal
19    reset          : in STD_LOGIC;                                    -- Reset signal
20    xaxis_data_in  : in STD_LOGIC_VECTOR(9 downto 0);                 -- X axis data
21    yaxis_data_in  : in STD_LOGIC_VECTOR(9 downto 0);                 -- Y axis data
22    zaxis_data_in  : in STD_LOGIC_VECTOR(9 downto 0);                 -- Z axis data
23    xaxis_data_out : out STD_LOGIC_VECTOR(9 downto 0) := (others => '0'); -- X axis data
24    yaxis_data_out : out STD_LOGIC_VECTOR(9 downto 0) := (others => '0'); -- Y axis data
25    zaxis_data_out : out STD_LOGIC_VECTOR(9 downto 0) := (others => '0')  -- Z axis data
26  );
```

3.3.3 Implementacja

```
40        -- If data is negative in U2, it is converted to positive number and
41        -- the MSB is set to 1 for further conversion in ILA.
42        if xaxis_data_in(9) = '1' then
43          xaxis_data_out <= not xaxis_data_in(9 downto 0) + '1';
44          xaxis_data_out(9) <= '1';
45        else
46          xaxis_data_out <= xaxis_data_in;
47        end if;
```

3.4 debouncer



Rysunek 7: Blokowe przedstawienie modułu.

3.4.1 Funkcjonalność

Zadaniem modułu debouncer jest zminimalizowanie wpływu zjawiska drgania styków po wciśnięciu przycisku na płytce. Moduł wykrywa zmiany stanu na linii przycisku a następnie odczekuje zadaną ilość cylki zegara i sprawdza, czy wartość po odliczeniu jest taka sama jak w momencie rozpoczęcia odliczania, jeśli tak — moduł przełącza wartość wyjścia z użyciem funkcji `not`.

3.4.2 Porty

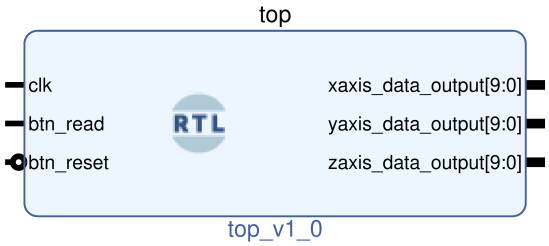
```
17  port (
18    i_clk       : in STD_LOGIC;
19    i_button    : in STD_LOGIC;
20    o_debouncer : out STD_LOGIC
21  );
```

3.4.3 Implementacja

```
31  process (i_clk)
32  begin
33    if rising_edge(i_clk) then
34      if (i_button /= r_button) then
35        r_count_flag <= '1';
36      end if;
37      if (r_count_flag = '1' and r_cnt < DEBOUNCER_CNT_LIMIT) then
38        r_cnt <= r_cnt + 1;
39      elsif (r_count_flag = '1' and r_cnt >= DEBOUNCER_CNT_LIMIT and i_button = '1') then
40        r_count_flag <= '0';
41        r_cnt        <= 0;
```

```
42          r_debouncer  <= not r_debouncer;
43        elsif (r_count_flag = '1' and r_cnt >= DEBOUNCER_CNT_LIMIT and i_button = '0') then
44          r_count_flag <= '0';
45          r_cnt        <= 0;
46        end if;
47        r_button <= i_button;
48
49      end if;
50    end process;
51
52    o_debouncer <= r_debouncer;
```

3.5 top



Rysunek 8: Blokowe przedstawienie modułu.

3.5.1 Funkcjonalność

Zadaniem modułu jest połączenie wszystkich komponentów w gotowy układ do wgrania na fpga.

3.5.2 Porty

```
18    port (
19      clk               : in STD_LOGIC;                     -- clk
20      btn_read          : in STD_LOGIC;                     -- start reading button
21      btn_reset         : in STD_LOGIC;                     -- reset button
22      xaxis_data_output : out STD_LOGIC_VECTOR(9 downto 0); -- X axis data
23      yaxis_data_output : out STD_LOGIC_VECTOR(9 downto 0); -- Y axis data
24      zaxis_data_output : out STD_LOGIC_VECTOR(9 downto 0)  -- Z axis data
25    );
```

3.5.3 Implementacja

```
90  debouncer_Reset : entity work.debouncer
91    generic map(DEBOUNCER_CNT_LIMIT => 100 - 1)
92    port map(
93      i_clk       => clk,
94      i_button    => btn_reset,
95      o_debouncer => btn_reset_D
96    );
97
98  debouncer_Read : entity work.debouncer
99    generic map(DEBOUNCER_CNT_LIMIT => 100 - 1)
100   port map(
101     i_clk       => clk,
102     i_button    => btn_read,
103     o_debouncer => btn_read_D
104   );
105
106 SPImaster_1 : entity work.SPImaster
107   generic map(
108     PRESCALER     => X"28", -- 2.5 MHz (Int. clock over 100 MHz)
109     DELAY_COUNTER => X"15"  -- 210 ns + 10 ns
110   )
111   port map(
112     clk        => clk,
113     reset      => btn_reset_D,
114     cs         => cs_SPImaster_out,
115     sclk       => sclk_SPImaster_out,
116     mosi       => mosi_SPImaster_out,
117     miso       => miso_SPImaster_out,
118     read_data  => btn_read_D,
119     xaxis_data => xaxis_data_SPImaster_out,
120     yaxis_data => yaxis_data_SPImaster_out,
121     zaxis_data => zaxis_data_SPImaster_out
122   );
123
124 SPIconverter_1 : entity work.SPIconverter port map (
125   clk            => clk,
126   reset          => btn_reset_D,
127   xaxis_data_in  => xaxis_data_SPImaster_out,
128   yaxis_data_in  => yaxis_data_SPImaster_out,
129   zaxis_data_in  => zaxis_data_SPImaster_out,
130   xaxis_data_out => xaxis_data_output,
131   yaxis_data_out => yaxis_data_output,
132   zaxis_data_out => zaxis_data_output
133   );
134
135 PmodACL_1 : entity work.PmodACL port map(
136   clk  => clk,
137   cs   => cs_SPImaster_out,
138   sclk => sclk_SPImaster_out,
139   mosi => mosi_SPImaster_out,
140   miso => miso_SPImaster_out
141   );
```

4 Symulacja

4.1 Makefile

```
# ------------------------------------------------------------
# University: Warsaw University of Technology
# Author:     Wiktor Chocianowicz
# ------------------------------------------------------------
# Create Date:    13/01/2024
# Description:    Makefile for compiling PUF project.
# ------------------------------------------------------------

MODULES=debouncer.vhd SPImaster.vhd SPIconverter.vhd PmodACL.vhd
TOP=top.vhd
TESTBENCH=testbench.vhd

.PHONY: waves clean

$(patsubst %.vhd,%.vcd,$(TESTBENCH)): $(patsubst %.vhd,%,$(TESTBENCH))
  @echo "Remaking $@"
  @./$< --vcd=$@

$(patsubst %.vhd,%,$(TESTBENCH)): $(patsubst %.vhd,%.o,$(TESTBENCH))
  @echo "Remaking $@"
  @ghdl -e --std=08 -fsynopsys $@

$(patsubst %.vhd,%.o,$(TESTBENCH)): $(patsubst %.vhd,%.o,$(TOP)) TESTBENCH/$(TESTBENCH)
  @echo "Remaking $@"
  @ghdl -a --std=08 -fsynopsys TESTBENCH/$(TESTBENCH)

$(patsubst %.vhd,%.o,$(TOP)): $(patsubst %.vhd,%.o,$(MODULES)) TOP/$(TOP)
  @echo "Remaking $@"
  @ghdl -a --std=08 -fsynopsys TOP/$(TOP)

$(patsubst %.vhd,%.o,$(MODULES)): %.o: MODULES/%.vhd
  @echo "Remaking $@"
  @ghdl -a --std=08 -fsynopsys $<

waves: $(patsubst %.vhd,%.vcd,$(TESTBENCH))
  @echo "Wave'ing $<"
  @gtkwave -o $(patsubst %.vhd,%.vcd,$(TESTBENCH)) 2> /tmp/gtkwave

clean:
  @echo "Cleaning up..."
  @rm -f *.o testbench *.vcd work*.cf *.vcd.fst
```

Skrypt 2: Makefile użyty do kompilacji projektu na platformie Ubuntu Linux.

4.2 Struktura katalogów projektu

```
.
|--- makefile
|--- MODULES
|    |--- debouncer.vhd
|    |--- PmodACL.vhd
|    |--- SPIconverter.vhd
|    +--- SPImaster.vhd
|--- TESTBENCH
|    +--- testbench.vhd
+--- TOP
     +--- top.vhd
```

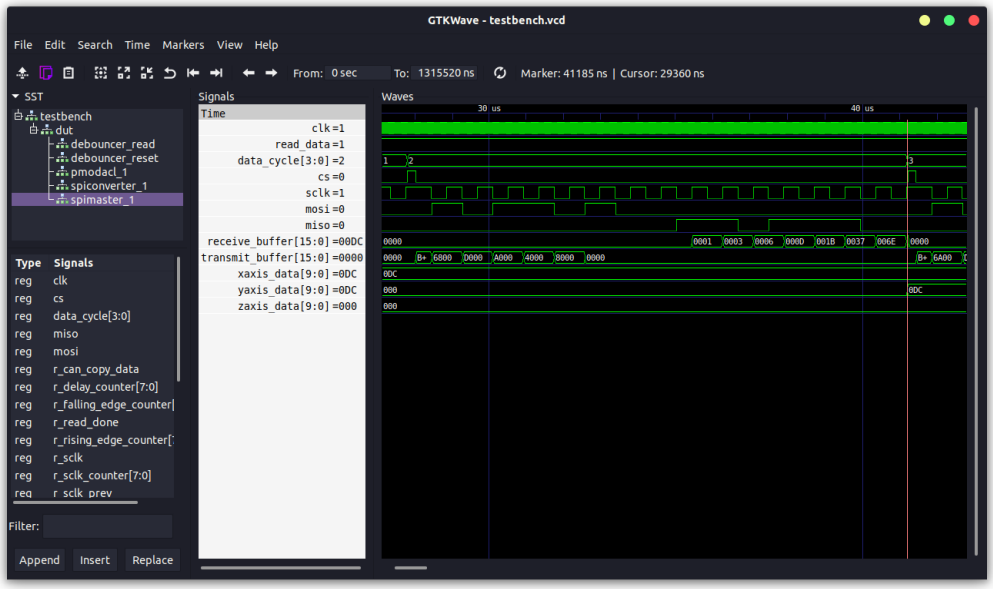
4.3 Testbench

```
39  -- Clock
40  clk <= not clk after 5 ns;
41
42  dut : top port map(
43    clk               => clk,
44    btn_read          => btn_read,
45    btn_reset         => btn_reset,
46    xaxis_data_output => xaxis_data,
47    yaxis_data_output => yaxis_data,
48    zaxis_data_output => zaxis_data
49  );
50
51  stimulus :
52  process begin
53    wait for 20 ns;
54    for i in 0 to 800 loop
55      btn_read <= '0';
56      wait for 500 ps;
57      btn_read <= '1';
58      wait for 500 ps;
59    end loop;
60    wait for 300 ns;
61    btn_read <= '0';
62
63    wait for 200 us;
64    for i in 0 to 800 loop
65      btn_reset <= '0';
66      wait for 500 ps;
67      btn_reset <= '1';
68      wait for 500 ps;
69    end loop;
70    wait for 300 ns;
71    btn_reset <= '0';
72
73    wait for 50 us;
74    for i in 0 to 800 loop
75      btn_reset <= '0';
76      wait for 500 ps;
77      btn_reset <= '1';
78      wait for 500 ps;
79    end loop;
80    wait for 300 ns;
81    btn_reset <= '0';
82
83    wait for 20 us;
84    for i in 0 to 800 loop
85      btn_read <= '0';
86      wait for 500 ps;
87      btn_read <= '1';
88      wait for 500 ps;
89    end loop;
90    wait for 300 ns;
91    btn_read <= '0';
92
93    wait for 40 us;
94    for i in 0 to 800 loop
95      btn_read <= '0';
96      wait for 500 ps;
97      btn_read <= '1';
98      wait for 500 ps;
99    end loop;
100   wait for 300 ns;
101   btn_read <= '0';
102
103   wait for 1 ms;
104   stop;
105 end process stimulus;
106 end behavioural; -- behavioural
```

Skrypt 3: Testbench użyty do symulacji projektu.

Testbench generuje zegar z częstotliwością 100 MHz. W procesie stymulującym generowane są naciśnięcia przycisków odczytu i resetu. Zdarzenia zwarcia przycisków symulują w pętli drgania styku.

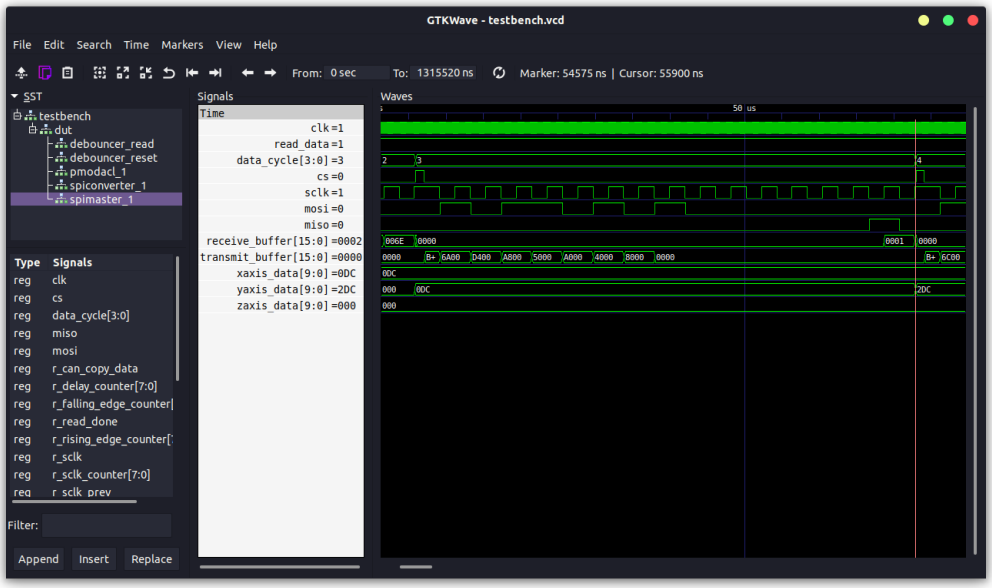
4.4 Ramka danych



Rysunek 9: Symulacja dla odczytu danych rejestru zerowego osi y.

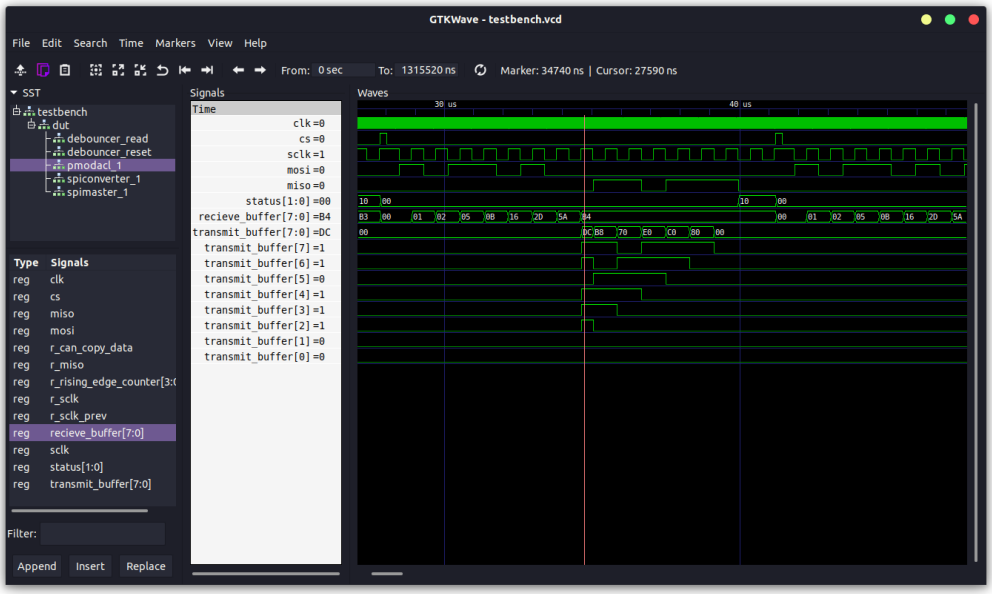
Przy wykonywaniu symulacji sprawdzano czy zachowany jest chronometraż z Rys. (2). Przy odpowiedniej implementacji „timingi" zgadzały się z tabelą z karty katalogowej. Dodane zostało jedynie opóźnienie między następnymi transmisjami $t_{CS,DIS}$.

Na Rys. (9) i (10) można zaobserwować pełne cykle transmisji danych po interfejsie SPI. Transmisja rozpoczyna się w momencie opuszczenia linii `cs`, po czym rozpoczyna się generacja sygnału `sclk`. Na zboczach opadających obserwujemy zmianę danych na poszczególnych liniach. Dane próbkowane są na zboczach narastających. W momencie wysłania adresu rejestru do odczytu `SPImaster` zaczyna odbierać dane z linii `miso`. Dane przenoszone są do bufora i wysyłane komplementarnie do `SPIconverter`.



Rysunek 10: Symulacja dla odczytu danych rejestru pierwszego osi y.

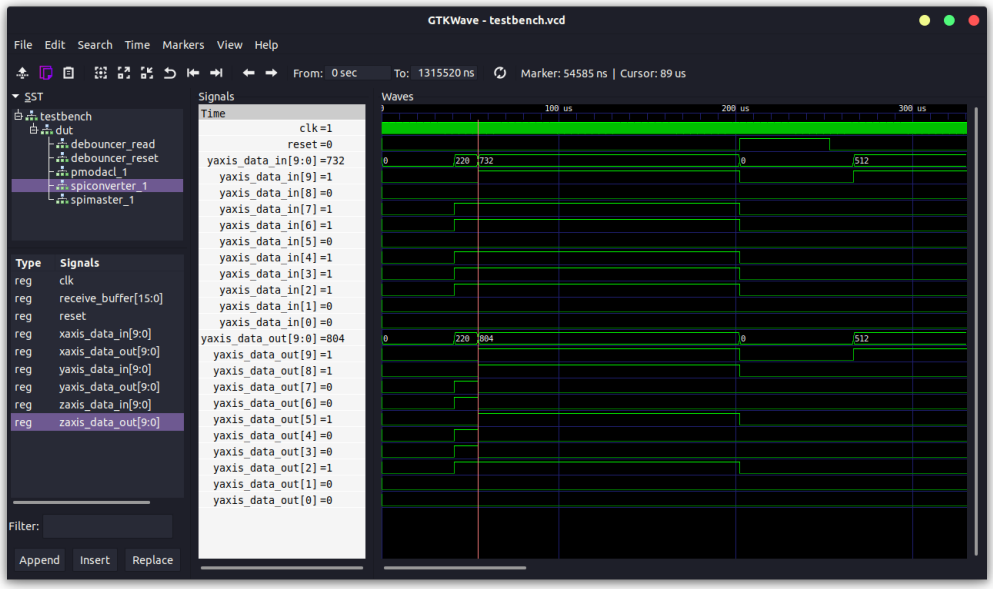
4.5 Emulator akcelerometru



Rysunek 11: Odbiór adresu rejestru i zapełnienie bufora transmisji.

Emulator pmoda czeka na otrzymanie adresu rejestru, z którego mają zostać wysłane dane. Następnie w bloku `case` zapełniany jest bufor danych, które mają zostać wystawione na linię `miso`.

4.6 Konwersja na wartość bitową ze znakiem



Rysunek 12: Symulacja konwersji U2 na kod binarny z bitem znaku.

Konwersja odbywa się przez negację logiczną bitów wektora i dodanie binarnej 1. Następnie na MSB ustawiana jest 1. Tak przygotowany wektor można będzie przekształcić poza syntezowalnym układem w wartość przyspieszenia ziemskiego po zidentyfikowaniu wartości ujemnych i pozytywnych, a następnie pomnożenie wartości przez 3,9 mg/LSB.