به نام خدا

عنوان درس

VHDL

عنوان پروژه

راه اندازی پرتکل ارتباطی I2C

(AD5622)

استاد

دکتر ستار میرزاکوچکی

گردآورنده

زهرا داورزنی

زمستان ۱۴۰۱

ارتباط I2C پرتکل ارتباطی سریال از طریق دو سیم SDA و SCl است. انتقال دیتا از مستر به اسلیو و برعکس در مد read و write از طریق پین SDA انجام می‌شود. I2C در سه مد standard mode و fast mode و high speed mode کار می‌کند که تفاوت این مدهای کاری در سرعت انتقال دیتا می‌باشد. فرکانس در standard mode، 100 khz و در fast mode، 400 khz و در high speed mode، 3.4 Mhz می‌باشد. بررسی انجام شده در اینجا در مد standard mode می‌باشد. فرکانس در نظر گرفته شده برای fpga، 80 Mhz می‌باشد.

در حالت دیفالت هر دو سیم SDA و SCl با مقاومتی پول اپ شده است و دارای مقدار ۱ هستند.

در ارتباط I2C زمانی که سطح ولتاژ کلاک ۱ است مقدار دیتا باید ثابت باشد زمانی که کلاک ۰ است دیتا می‌تواند تغییر داشته باشد در واقع setup time یعنی زمانی که قبل از ۱ شدن کلاک I2C، دیتا موردنظر باید روی سیگنال SDA قرار بگیرد و hold time است یعنی زمانی که بعد از ۰ شدن کلاک، مقدار دیتا نباید تغییر کند. رعایت این نکته از اهمیت بالایی برخوردار است.

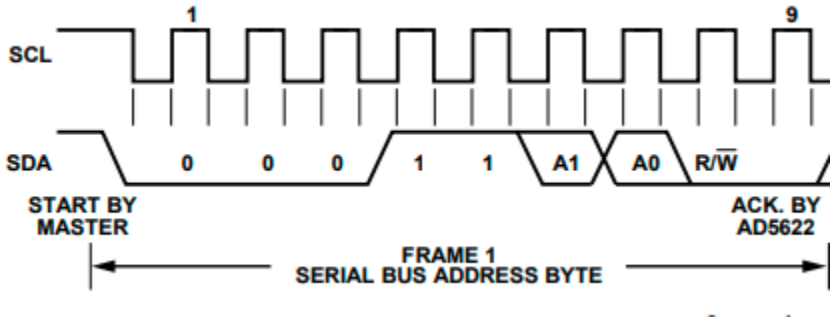
در دیتاشیت AD5622، مقدار setup time، ۲۵۰ نانو ثانیه است و ماکزیمم مقدار hold time، ۳,۴ میکرو ثانیه و مینیمم مقدار ان ۰ می‌باشد. با توجه به اینکه کلاک I2C، 100 khz و دوره تناوب ۱۰ میکروثانیه است و به منظور قرار گرفتن تغییرات SDA در وسط کلاک I2C، SDA_clk ایجاد شده است و کاملا setup time و hold time رعایت شده است.

مستر برای اطلاع به اسلیو برای شروع انتقال دیتا باید به اسلیو سیگنال start ارسال کند و بعد از اتمام انتقال دیتا باید سیگنال stop را ارسال کند.

موارد مربوط به این سیگنال‌ها تحت عنوان start condition و stop condition در دیتاشیت مطرح شده است. start condition و stop condition به این شکل است که زمانی که کلاک ۱ است اگر SDA از ۱ به ۰ تغییر پیدا کند یعنی سیگنال start را ارسال می‌کند در Timing diagram، S نشان دهنده سیگنال start است و اسلیوها باید اماده دریافت دیتا باشند پس از اتمام انتقال دیتا زمانی که کلاک ۱ است اگر SDA از ۰ به ۱ تغییر حالت بدهد stop condition اتفاق افتاده است در Timing diagram، P نشان دهنده سیگنال stop است و انتقال دیتا با اسلیو موردنظر به اتمام رسیده است. لازم به ذکر است که هر دو سیگنال start و stop را مستر ارسال می‌کند.

همانطور که در شکل زیر مشاهده می‌شود ابتدا start bit توسط مستر ارسال می‌شود بعد از ارسال start bit یک ادرس ۷ بیتی که شامل ادرس اسلیو می‌باشد از طرف مستر به اسلیو ارسال می‌شود. در بیت ۸ام مستر، بیت

مربوط به تعیین مد read و write را ارسال میکند. اگر در مد write باشیم یعنی مستر بخواهد دیتا در اسلیو بنویسد بیت R/W باید ۰ باشد.



همانطور که در دیتاشیت تعیین شده است این ادرس بصورت "00011A0A1" می‌باشد. که مقادیر A0 و A1 در دیتاشیت مطابق جدولی تعیین می‌شود که در اینجا بصورت "00" درنظر گرفته شده است.

پس از دریافت ۸ بیت توسط اسلیو، اسلیو SDA را در کلاک ۹ام، 0 می‌کند. این بیت acknowledge نام دارد. اینکار برای اطلاع به مستر است که اطلاعات به درستی دریافت شده است. بیت acknowledge باید توسط گیرنده ارسال شود و این بیت پس از هر بایت باید فرستاده شود. نکته قابل توجه دیگر این است که انتقال دیتا از بیت msb شروع می‌شود و با بیت lsb پایان می‌یابد.

همانطور که در کد و شبیه سازی مشاهده می‌شود بیت r/w، صفر درنظر گرفته شده است که یعنی مستر در مد نوشتن است. بنابراین مد نوشتن در AD5622 بررسی می‌کنیم.

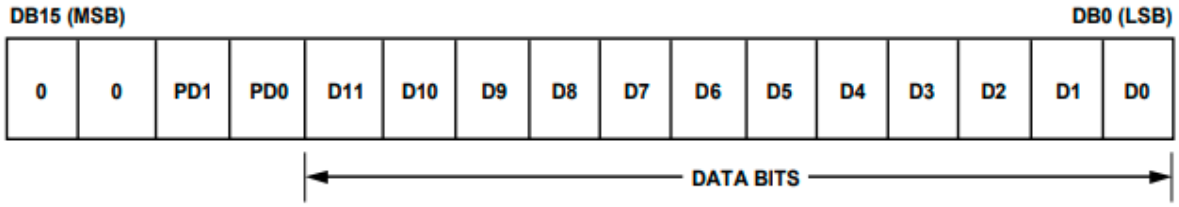


Figure 41. AD5622 Input Register Contents

، ۱۶ بیتی است یعنی در هر بار نوشتن یا خواندن بایستی ۱۶ بیت AD5622 شکل بالا نشان می‌دهد رجیستر است که این دو بیت PD2 و PD1 و دو بیت 0دیتا دریافت یا ارسال گردد. ۴ بیت پرارزش دارای دو بیت ثابت تعیین کننده مد عملکردی دیوایس می‌باشد که در کد و شبیه سازی بصورت "00" در نظر شده است که به معنای مد نرمال عملکردی دیوایس می‌باشد.

در مد نوشتن waveform سیگنال بصورت زیر می‌باشد.

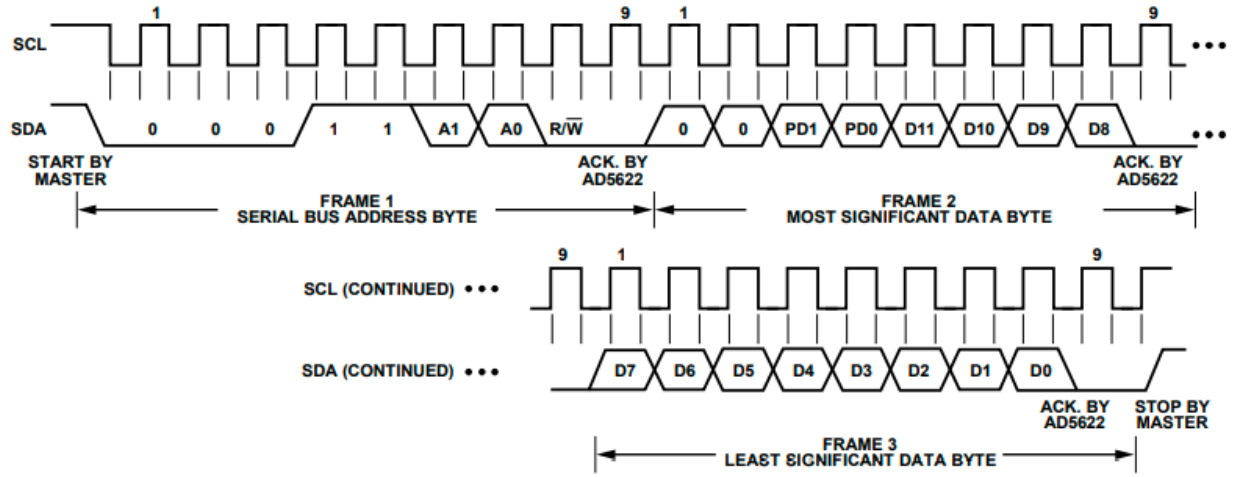


Figure 45. AD5622 Write Sequence

همانطور که در شکل بالا مشاهده می‌شود ارسال از بیت پرارزش رجیستر AD5622 اغاز می‌شود و بعد از ارسال ۸ بیت، بیت اکنالج توسط گیرنده (AD5622) با صفر کردن SDA ارسال می‌شود سپس ۸ بیت دوم ارسال می‌شود و سپس بیت اکنالج ارسال می‌شود.

در مد خواندن waveform سیگنال بصورت زیر می‌باشد.

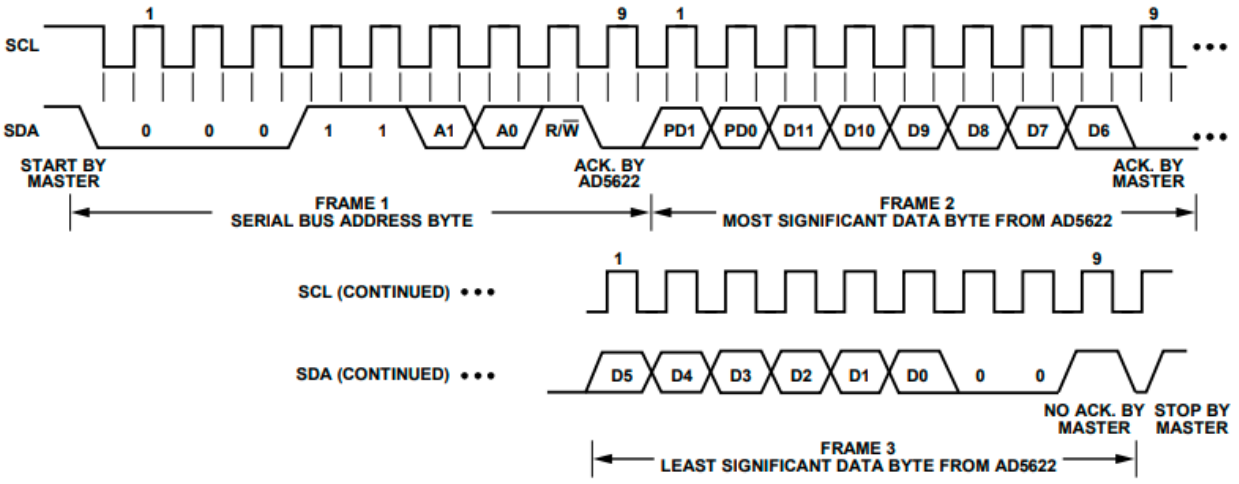


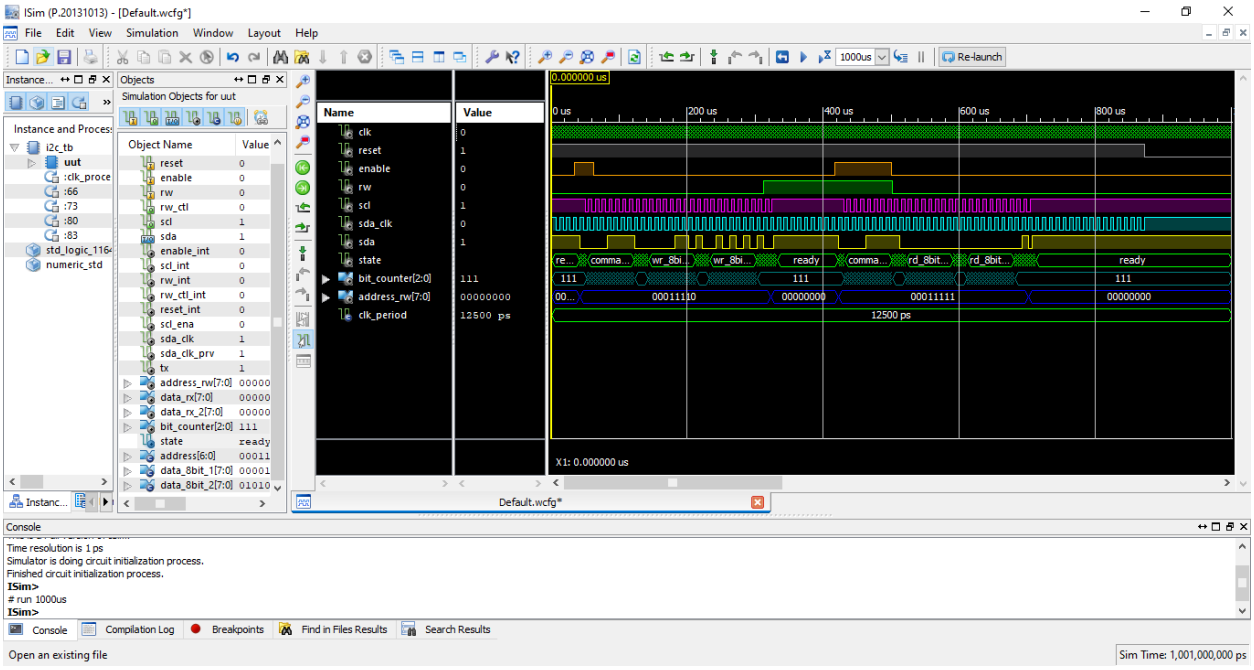
Figure 48. AD5622 Read Sequence

همانطور که در شکل بالا مشاهده می‌شود اسلیو ابتدا دو بیت PD1 و PD2 را ارسال می‌کند که این دو بیت "00" در نظر گرفته شده است که به معنای مد نرمال است و بعد از ارسال ۶ بیت دیگر، بیت اکنالج توسط مستر ارسال می‌شود. سپس ارسال ۸ بیت دوم اغاز می‌شود. دو بیتی که در انتها ارسال می‌شود "00" است.

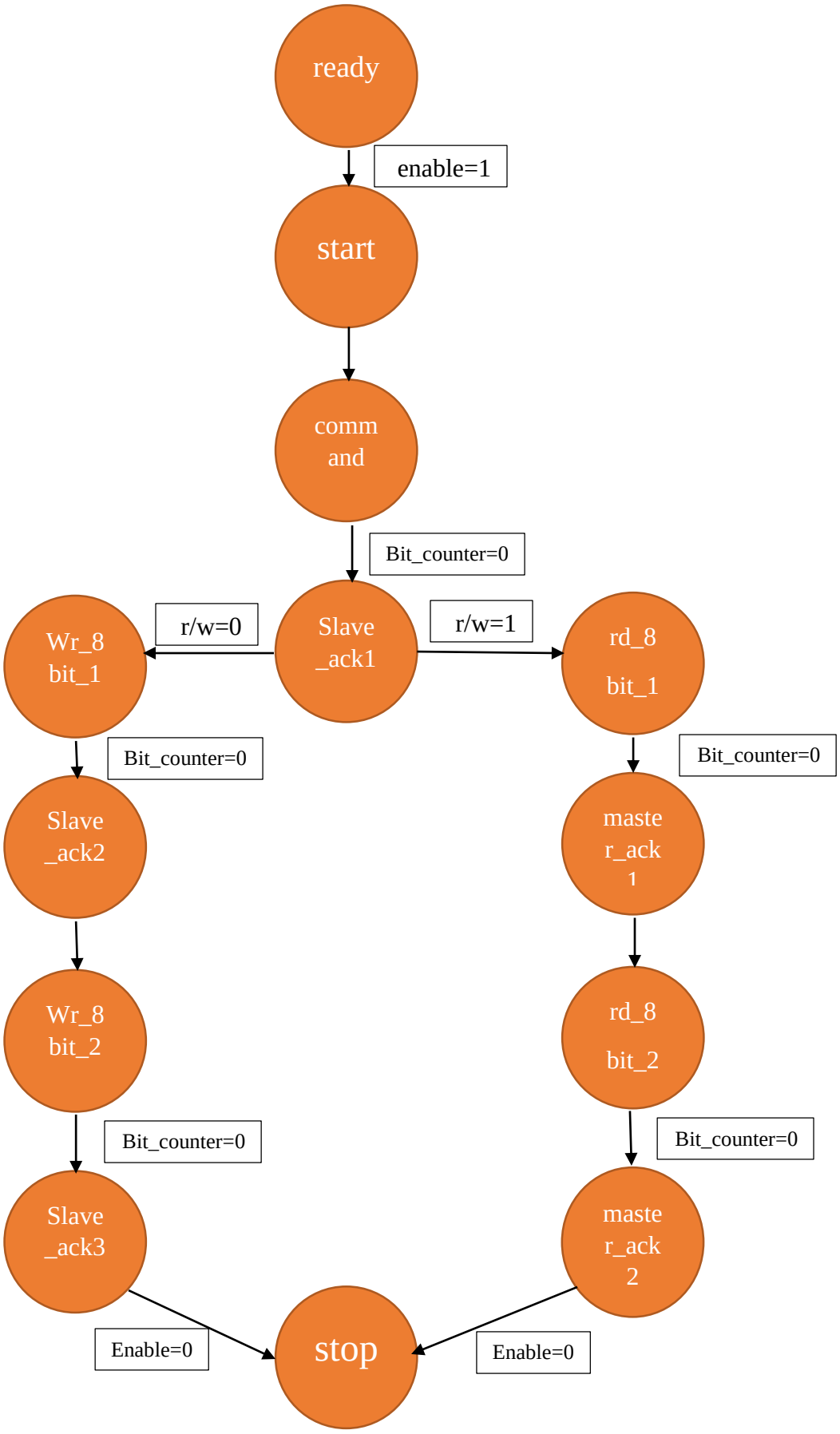
بعد ازاتمام این ۸ بیت باید بیت اکنالج توسط مستر ارسال شود که در دیتاشیت بیان شده است این بیت no acknowledge نام دارد و د رکلاک ۹ام، SDA از 0 به 1 تغییر پیدا می‌کند و قبل از کلاک ۱۰ام باید SDA صفر شود و در کلاک ۱۰ام مجدد 1 شود.

اکنون به شبیه سازی و کد مربوط به راه اندازی این دیوایس می‌پردازیم.

نمای کلی از شبیه سازی در محیط ise در شکل زیر مشاهده می‌شود سپس به جزئیات ان پرداخته خواهد شد.



ابتدا طرح کلی از حالت هایی که در برنامه نویسی برای این دیوایس بکار برده شده است را بیان می‌کنیم.


ready
enable=1
start
comm
and
Bit_counter=0
Slave
_ack1
r/w=0
r/w=1
Wr_8
bit_1
rd_8
bit_1
Bit_counter=0
Bit_counter=0
Slave
_ack2
maste
r_ack
1
Wr_8
bit_2
rd_8
bit_2
Bit_counter=0
Bit_counter=0
Slave
_ack3
maste
r_ack
2
Enable=0
stop
Enable=0

مطابق طرح کلی نشان داده شده مستر انقدر در حالت ready می‌ماند تا enable = 1 شود و استیت تغییر پیدا می‌کند.

در کد مربوطه ابتدا کتابخانه های استفاده شده را تعریف کرده.

```
5  ----------------------------------------------------------------------------------
6  -- Library
7  library IEEE;
8  use IEEE.STD_LOGIC_1164.ALL;
9  use IEEE.NUMERIC_STD.ALL;
10 ----------------------------------------------------------------------------------
```

سپس پورت های ورودی و خروجی تعریف شده است.

```
10 ----------------------------------------------------------------------------
11 entity I2C_AD5622 is
12    PORT(
13          clk               : IN       STD_LOGIC;   -- Fpga clk = 80 MHz
14          Reset             : IN       STD_LOGIC;   -- Fpga reset
15          Enable            : IN       STD_LOGIC;   -- If Enable = '1' the i2c communication will be active and generate SCL
16          RW                : IN       STD_LOGIC;   -- Define write mode or read mode if  RW = 0, mode is write
17          RW_CTL            : OUT      STD_LOGIC;   -- Define when we want to write or read if RW_CTL = 0, mode is tansmit
18          SCL               : OUT      STD_LOGIC;   -- I2C clock
19          SDA               : INOUT    STD_LOGIC    -- Data line for i2c
20
21       );
22 end I2C_AD5622;
```

سپس سیگنال های داخلی و متغییرهای مورد نیاز تعریف شده است.

```
23
24 architecture Behavioral of I2C_AD5622 is
25    -- I/O internal signal
26    SIGNAL Enable_Int        : STD_LOGIC      :='0';
27    SIGNAL SCL_Int           : STD_LOGIC      :='0';
28    SIGNAL RW_Int            : STD_LOGIC      :='0';
29    SIGNAL RW_CTL_Int        : STD_LOGIC      :='0';
30    SIGNAL Reset_Int         : STD_LOGIC      :='0';
31    -- internal signal
32    SIGNAL SCL_Ena           : STD_LOGIC      :='0';                          -- Activate i2c clock when we want i2c
33    SIGNAL SDA_clk           : STD_LOGIC      :='0';                          -- Clock for put data on SDA because setup time
34    SIGNAL SDA_clk_prv       : STD_LOGIC      :='0';                          -- Creat risinge edge
35    SIGNAL Tx                : STD_LOGIC      :='1';                          -- Data that we want to transmit
36    SIGNAL Address_RW        : STD_LOGIC_VECTOR (7 DOWNTO 0)   :=(others=> '0'); -- Slave address & read and write bit
37    SIGNAL Data_Rx           : STD_LOGIC_VECTOR (7 DOWNTO 0)   :=(others=> '0'); -- Data that recieve from slave
38    SIGNAL Data_Rx_2         : STD_LOGIC_VECTOR (7 DOWNTO 0)   :=(others=> '0'); -- Data that recieve from slave
39
40    SIGNAL Bit_counter       : unsigned (2 DOWNTO 0) :=(others=> '1');          -- jount 0 to 7
41    -- State Machine
42    TYPE state_machine   IS (ready, start, command, slave_ack1, wr_8bit_1, wr_8bit_2, rd_8bit_1, rd_8bit_2, slave_ack2,
43                             slave_ack3, sack1_del, sack2_del, sack3_del, master_ack1, master_ack2, mack1_del, mack2_del, stop);
44    SIGNAL state   : state_machine     := ready;
45    -- Constant
46    CONSTANT Address         : STD_LOGIC_VECTOR (6 DOWNTO 0)    := "0001111";   -- Slave address 00011 &  A1A2(A1 = 1, A2 = 1 --->
47    CONSTANT Data_8bit_1     : STD_LOGIC_VECTOR (7 DOWNTO 0)    := "00001101";  -- The fist 8 bit that shouhd transmit on write m
48    CONSTANT Data_8bit_2     : STD_LOGIC_VECTOR (7 DOWNTO 0)    := "01010101";  --The second 8 bit that we want to transmit (defi
```

نحوه ایجاد سیگنال‌های SCL و ... در شکل زیر مشاهده می‌شود.

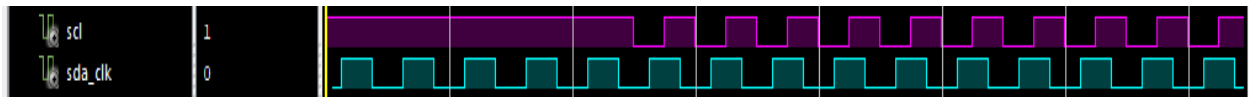
```
49 begin
50    SCL    <= '0' when (SCL_Ena = '1' AND SCL_Int = '0') else '1'; -- Generate i2c clock
51    RW_CTL <= RW_CTL_Int;
52    SDA    <= Tx when RW_CTL_Int = '0' else 'Z'; -- When master are on write mode (RW_CTL_Int = '0'), master transmit Tx on SDA o
53    Reset_Int <= Reset;
```

در این بخش از کد پروسس حساس به کلاک مربوط به ایجاد SDA_clk مشاهده می‌شود.

```
55  Generate_SCL_SDA_clock: process(clk)        -- Generate SCL_clk and SDA_clk
56                                              -- SCL_clk = 100 KHz (standarde mode)
57                                              -- Master put data on SDA in Rising edge of SDA_clk
58
59     variable count : INTEGER RANGE 0 TO 800; -- Fpga clock is 80 MHz and generat on testbench
60
61  begin
62
63        IF(reset_Int = '0') THEN
64           count := 0;
65
66        ELSIF(clk'EVENT AND clk = '1') THEN
67           SDA_clk_prv <= SDA_clk;        -- Detect rising edge of SDA_clk
68           IF(count = 799) THEN            -- (Fpga clock = 80 MHz) / (SCL clock = 100 KHz) = 800 ---> count range is 0 to 799
69              count := 0;
70           ELSE
71              count := count + 1;
72           END IF;
73
```

```
73
74           CASE count IS
75              WHEN 0 TO 199 =>
76
77                 SCL_Int  <= '0';
78                 SDA_clk <= '0';
79
80              WHEN 200 TO 399 =>
81
82                 SCL_Int  <= '0';
83                 SDA_clk <= '1';
84
85              WHEN 400 TO 599 =>
86
87                 SCL_Int  <= '1';
88                 SDA_clk <= '1';
89
90              WHEN OTHERS =>
91
92                 SCL_Int  <= '1';
93                 SDA_clk <= '0';
94
95           END CASE;
96        END IF;
97  END PROCESS;
```

شکل زیر شبیه سازی مربوط به SDA_clk و SCL را نشان می‌دهد. مشاهده می‌شود لبه‌ی بالا رونده SCL در در وسط SDA_clk قرار دارد. در لبه بالارونده SDA_clk دیتا روی SDA قرار می‌گیرد به همین دلیل ایجاد SDA_clk از اهمیت بالایی برخوردار است.



پروسس بعدی کد در ارتباط با انتقال و دریافت دیتا می‌باشد.

```
99  Data_Transmiting_Recieving: process(clk)
100 begin
101
102    IF (reset_Int = '0') THEN
103
104       state       <= ready;
105       Enable_Int <= '0';
106       RW_CTL_Int     <= '0';
107       Bit_counter    <= (others => '1');
108       Data_Rx        <= (others => '0');
109
110    ELSIF (clk'EVENT AND clk = '1') THEN -- Rising edge of fpga clock
111
112       Enable_Int  <= Enable;
113       RW_Int      <= RW;
```

همانطور که در تصویر بالا مشاهده می‌شود در صورت ریست کردن مقادیر سیگنال‌ها به مقادیر اولیه برمی‌گردد که جلوتر در شبیه سازی کد این موضوع مشاهده می‌شود.

وضعیت استیت‌ها در تصاویر زیر مشاهده می‌شود.

```
117         IF (SDA_clk_prv = '0' AND SDA_clk = '1') THEN -- Rising edge of SDA_clk
118
119            CASE state IS
120
121               WHEN ready =>                          -- master/fpga waits on ready state unitl Enable = 1
122
123                  RW_CTL_Int     <= '0';              -- If Enable = 1, master wants to write start bit and address bits on sla
124                  Bit_counter    <= (others => '1');  -- Bit_counter = 111 (7 in decimal)
125                  Data_Rx        <= (others => '0');  -- Data_Rx = "00000000"
126
127                  IF (Enable_Int = '1') THEN
128
129                     state       <= start;
130                     Tx          <= '0';              -- creat start bit(start bit = SDA becomes low when SCL is high)
131                     Address_RW  <= Address & RW_Int; -- put sddress slave with bit write or bit read in Address_RW
132
133                  ELSE
134
135                     state       <= ready;            -- If Enable = 0 master/fpga waits on ready state
136                     Tx          <= '1';              -- SDA is pull up normally and because master does not want to  write Tx =
137                     Address_RW  <= (others => '0');  -- Address_RW = "00000000" because i2c communication is not active and the
138
139                  END IF;
```

```
140
141               WHEN start =>
142
143                  state          <= command;
144                  RW_CTL_Int     <= '0';              -- Because master wants to write start bit and address bits on slave
145                  Tx             <= Address_RW (to_integer (Bit_counter)); -- Put the eighth bit of Address_RW on Tx and should p
146                  Bit_counter    <= Bit_counter - 1;  -- Bit_counter = 110
147                  Address_RW     <= Address_RW;
148                  Data_Rx        <= (others => '0');  -- Data_Rx = "00000000"
149
150               WHEN command =>                        -- Master waits until Bit_counter = 0 and after that state change to slave
151
152                  RW_CTL_Int     <= '0';
153                  Tx             <= Address_RW (to_integer (Bit_counter));
154                  Address_RW     <= Address_RW;
155
156                  Data_Rx        <= (others => '0');  -- Data_Rx = "00000000"
157                  IF (Bit_counter = 0) THEN
158
159                     state       <= slave_ack1;       -- when the all 8 bit send, slave should send acknowledge to master
160                     Bit_counter <= (others => '1');  -- Bit_counter = 111 (7 in decimal)
161
162                  ELSE
163
164                     state       <= command;
165                     Bit_counter <= Bit_counter - 1;
166
167                  END IF;
```

```
169               WHEN slave_ack1 =>
170
171                  state       <= sack1_del;           -- In 9th clock we should see acknowlegde so in 8th clock change the state
172                  RW_CTL_Int     <= '1';              -- Because slave wants to write acknowkedge bit on ADS line that means mas
173                  Tx             <= '1';              -- Set acknowledge bit on sda by reciever(here by testbench)
174                  Bit_counter    <= (others => '1');  -- Bit_counter = 111 (7 in decimal)
175                  Address_RW     <= Address_RW;
176                  Data_Rx        <= (others => '0');  -- Data_Rx = "00000000"
177
```

```
178             WHEN sack1_del =>                       -- On this state acknowledge bit has seen by master
179
180                Bit_counter <= Bit_counter - 1;
181                Address_RW     <= Address_RW;
182
183                IF (Address_RW(0) = '0') THEN        -- If Address_RW(0) = 0, mode is write
184
185                   state       <= wr_8bit_1;
186                   RW_CTL_Int     <= '0';           -- Mode is write
187                   Tx             <= Data_8bit_1 (to_integer (Bit_counter)); -- Write the first bit on sda when RW_CTL_Int = 0
188                   Data_Rx        <= (others => '0'); -- Data_Rx = "00000000"
189
190                ELSE                                 -- If Address_RW(0) = 1, mode is read
191
192                   state       <= rd_8bit_1;
193                   RW_CTL_Int     <= '1';           -- Mode is read
194                   Tx             <= '1';           -- Master reads data from SDA line (here from testbench)
195                   Data_Rx (to_integer (Bit_counter))  <= SDA; -- Read the first bit from SDA line when RW_CTL_Int = 1 slave t
196
197                END IF;
```

```
199             WHEN wr_8bit_1 =>                       -- Write the first 8 bits on slave
200
201                RW_CTL_Int     <= '0';               -- Mode is write
202                Tx             <= Data_8bit_1 (to_integer (Bit_counter)); -- Put data on SDA line
203                Address_RW     <= Address_RW;
204                Data_Rx        <= (others => '0');   -- Data_Rx = "00000000"
205
206                IF (Bit_counter = 0) THEN
207
208                   state       <= slave_ack2;        -- when the all 8 bits send, slave should send acknowledge to master
209                   Bit_counter <= (others => '1');   -- Bit_counter = 111 (7 in decimal)
210
211                ELSE
212
213                   state       <= wr_8bit_1;         -- If Bit_counter \= 0, stay on wr_8bit_1 state
214                   Bit_counter <= Bit_counter - 1;
215
216                END IF;
217
```

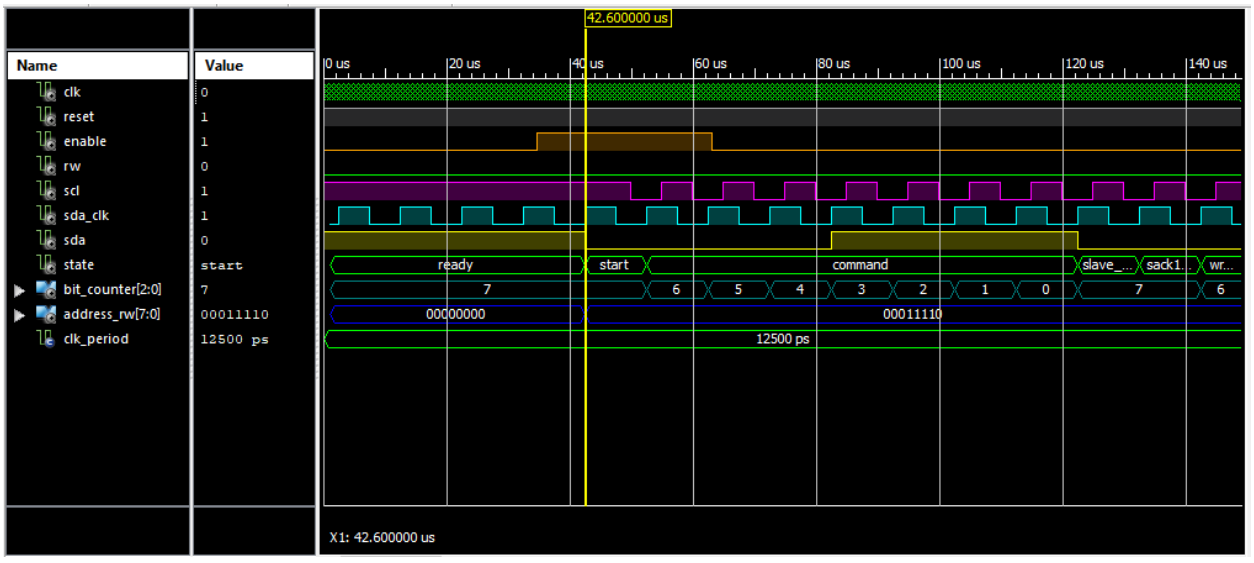
```
218             WHEN slave_ack2 =>
219
220                state       <= sack2_del;           -- State changes to sack2_del
221                RW_CTL_Int     <= '1';              -- Mode is read
222                Tx             <= '1';
223                Bit_counter    <= (others => '1');  -- Bit_counter = 111 (7 in decimal)
224                Address_RW     <= Address_RW;
225                Data_Rx        <= (others => '0');  -- Data_Rx = "00000000"
226
227             WHEN sack2_del =>
228
229                state       <= wr_8bit_2;           -- State changes to wr_8bit_2 because master wants to write the second 8
230                Bit_counter <= Bit_counter - 1;
231                RW_CTL_Int     <= '0';              -- Mode is write
232                Tx             <= Data_8bit_2 (to_integer (Bit_counter)); -- Put data on SDA line
233                Data_Rx        <= (others => '0');  -- Data_Rx = "00000000"
234
```

```
235             WHEN wr_8bit_2 =>                       -- Write the second 8 bits on slave
236
237                RW_CTL_Int     <= '0';               -- Mode is write
238                Tx             <= Data_8bit_2 (to_integer (Bit_counter));
239                Address_RW     <= Address_RW;
240                Data_Rx        <= (others => '0');   -- Data_Rx = "00000000"
241
242                IF (Bit_counter = 0) THEN
243                   state       <= slave_ack3;        -- when the all 8 bits send, slave should send acknowledge to master
244                   Bit_counter <= (others => '1');   -- Bit_counter = 111 (7 in decimal)
245
246                ELSE
247
248                   state       <= wr_8bit_2;         -- If Bit_counter \= 0, stay on wr_8bit_1 state
249                   Bit_counter <= Bit_counter - 1;
250
251                END IF;
252
```

```
253             WHEN slave_ack3 =>
254
255                state       <= sack3_del;             -- State changes to sack3_del
256                RW_CTL_Int  <= '1';                   -- Mode is read
257                Tx          <= '1';
258                Bit_counter <= (others => '1');       -- Bit_counter = 111 (7 in decimal)
259                Address_RW  <= Address_RW;
260                Data_Rx     <= (others => '0');       -- Data_Rx = "00000000"
261
262             WHEN sack3_del =>
263
264                Bit_counter <= Bit_counter - 1;
265                Data_Rx     <= (others => '0');       -- Data_Rx = "00000000"
266
267                IF (Enable_Int = '1') THEN            -- If Enable_Int = 1 communication
268
269                   Address_RW <= Address & RW_Int;
270
271                   IF (Address_RW = Address & RW_Int) THEN
272
273                      state       <= wr_8bit_1;       -- Stay on wr_8bit_1 state
274                      RW_CTL_Int  <= '0';             -- Mode is write
275                      Tx          <= Data_8bit_1 (to_integer (Bit_counter));
276                      Bit_counter <= Bit_counter - 1;
277
278                   ELSE
```

```
278                   ELSE
279
280                      state       <= start;           -- State changes to start
281                      RW_CTL_Int  <= '1';             -- Mode is read
282                      Tx          <= '1';
283                      Bit_counter <= (others => '1'); -- Bit_counter = 111 (7 in decimal)
284
285                   END IF;
286
287                ELSE                                  -- If Enable_Int \= 1, State changes to stop
288
289                   state       <= stop;
290                   Bit_counter <= (others => '1');    -- Bit_counter = 111 (7 in decimal)
291                   Address_RW <= (others => '0');     -- Address_RW = "00000000"
292                   Data_Rx <=  (others => '0');       -- Data_Rx = "00000000"
293
294                END IF;
```

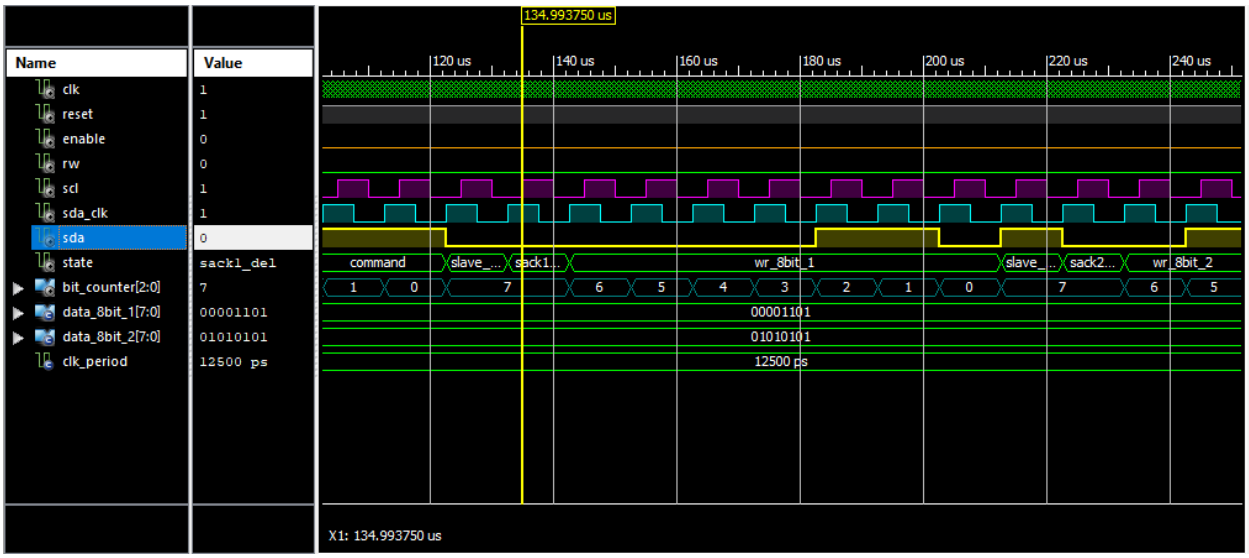
اکنون به بررسی کدهایی که نشان داده شد در شبیه سازی می‌پردازیم.



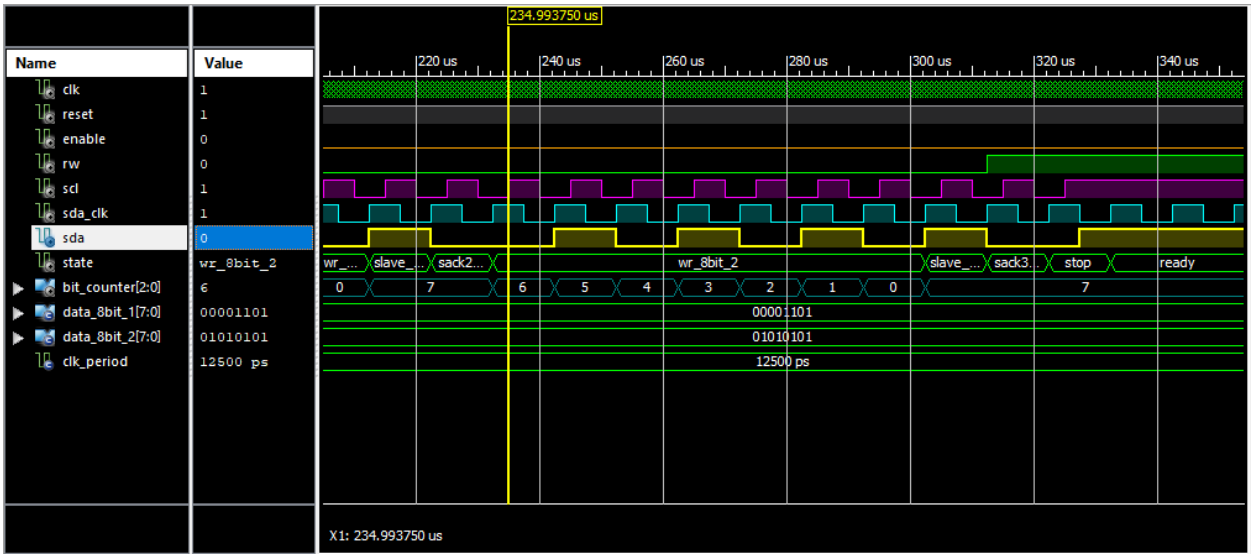
در تصویر بالا مشاهده می‌شود تا زمانی که enable = 0 است مستر در استیت ready منتظر مانده است.

مشاهده می‌شود در استیت start، SDA صفر شده است که نشان دهنده ارسال start bit می‌باشد. سپس مستر

ادرس به همراه بیت r/w را روی SDA قرار داده است و سپس اسلیو بیت اکنالج را ارسال کرده است و چون بیت کم ارزش address_rw، 0 است پس مستر در مد نوشتن می‌رود.



در شکل بالا مشاهده می‌شود در مد نوشتن، مستر ۸ بیت اول را به اسلیو ارسال می‌کند که این ۸ بیت data_8bit_1 است و سپس اسلیو بیت اکنالج را روی SDA قرار می‌دهد. همانطور که تصویر زیر مشاهده می‌شود سپس مستر ۸ بیت دوم یعنی data_8bit_2 را به اسلیو ارسال می‌کند و اسلیو مجدد بعد از دریافت ۸ بیت دوم بیت اکنالج ارسال می‌کند.



چون enable = 0 یعنی مستر نمی‌خواهد به نوشتن ادامه دهد، stop bit توسط مستر به اسلیو ارسال شده است که به معنای پایان ارتباط با اسلیو است.

مجدد در تست بنچ در زمان 417480 ns برای تست حالت خواندن، enable = 1 شده است. این بار می‌خواهیم مد خواندن را بررسی کنیم. ابتدا کد مربوط به مد خواندن را تصاویر بعدی مشاهده می‌کنید.

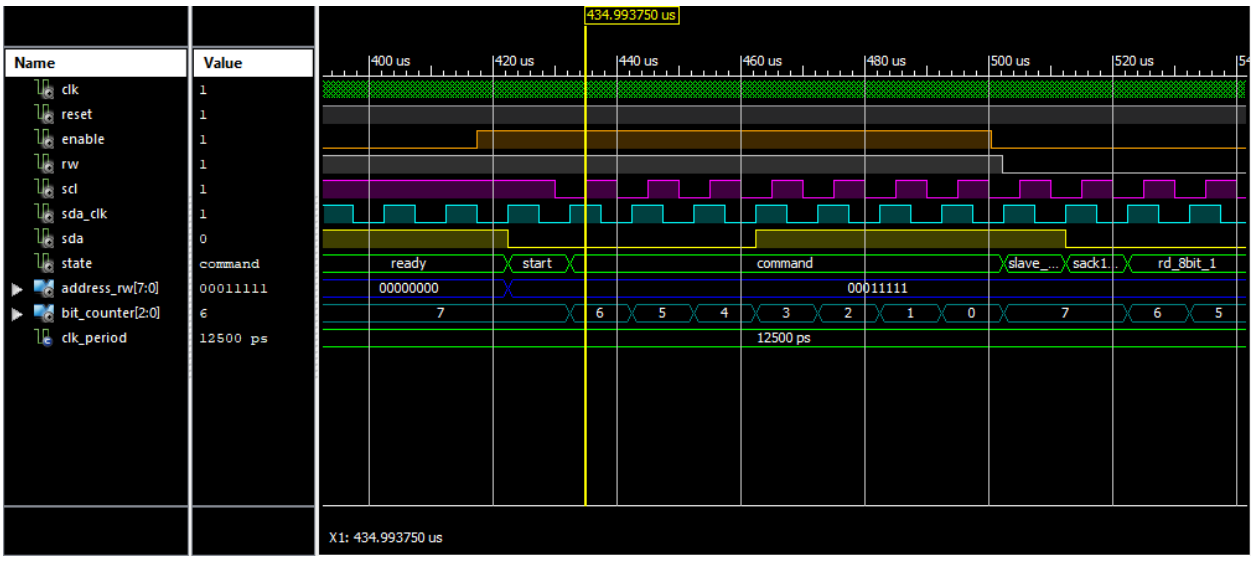
```
295
296             WHEN rd_8bit_1 =>                       -- Read the first 8 bits on slave
297
298                RW_CTL_Int     <= '1';               -- when we want recive data or read data from slave
299                Tx             <= '1';               -- set data from testbench
300                Address_RW     <= Address_RW;
301                Data_Rx (to_integer (Bit_counter))  <= SDA; --Data that master reads from slave store in Data_Rx
302
303                IF (Bit_counter = 0) THEN
304
305                   state       <= master_ack1;       -- State changes to master_ack1
306                   Bit_counter <= (others => '1');   -- Bit_counter = 111 (7 in decimal)
307
308                ELSE
309
310                   state       <= rd_8bit_1;         -- If Bit_counter \= 0, stay on rd_8bit_1
311                   Bit_counter <= Bit_counter - 1;
312
313                END IF;
314
315             WHEN master_ack1 =>
316
317                state       <= mack1_del;
318                RW_CTL_Int     <= '0';               -- Mode is write beacause master wants to creat acknowledge
319                Tx             <= '0';               -- SDA line becomes low by master because of acknowledge bit
320                Bit_counter    <= (others => '1');   -- Bit_counter = 111 (7 in decimal)
321                Address_RW     <= Address_RW;
322                Data_Rx        <= Data_Rx;
323
324             WHEN mack1_del =>
325
326                state       <= rd_8bit_2;            -- State changes to rd_8bit_2 because master wants to read the second 8 b
327                Bit_counter <= Bit_counter - 1;
328                RW_CTL_Int     <= '1';               -- Mode is read
329                Tx             <= '1';
330                Data_Rx_2 (to_integer (Bit_counter))  <= SDA;
331
332             WHEN rd_8bit_2 =>
333
334                RW_CTL_Int     <= '1';               -- when we want recive data or read data from slave
335                Tx             <= '1';               -- set data from test bench
336                Address_RW     <= Address_RW;
337                Data_Rx_2 (to_integer (Bit_counter))  <= SDA; -- Data that master reads from slave store in Data_Rx_2
338
339                IF (Bit_counter = 0) THEN
340
341                   state       <= master_ack2;       -- State changes to master_ack2
342                   Bit_counter <= (others => '1');   -- Bit_counter = 111 (7 in decimal)
343
344                ELSE
345
346                   state       <= rd_8bit_2;         -- If Bit_counter \= 0, stay on rd_8bit_2
347                   Bit_counter <= Bit_counter - 1;
348
349                END IF;
350
351             WHEN master_ack2 =>
352
353                state       <= mack2_del;
354                RW_CTL_Int     <= '0';               -- Mode is write beacause master wants to creat acknowledge
355                Tx             <= '1';               -- SDA line becomes high by master because of no acknowledge bit (refer t
356                Bit_counter    <= (others => '1');   -- Bit_counter = 111 (7 in decimal)
357                Address_RW     <= Address_RW;
358                Data_Rx        <= Data_Rx;
359                Data_Rx_2      <= Data_Rx_2;
```

```
361             WHEN mack2_del =>
362
363                Data_Rx        <= (others => '0');     -- Data_Rx = "00000000"
364
365                IF (Enable_Int = '1') THEN
366
367                   Address_RW <= Address & RW_Int;
368
369                   IF (Address_RW = Address & RW_Int) THEN
370
371                      state       <= rd_8bit_1;        -- Stay on rd_8bit_1 state
372                      RW_CTL_Int     <= '1';           -- Mode is read
373                      Tx             <= '1';
374                      Bit_counter <= Bit_counter - 1;
375                      Data_Rx (to_integer (Bit_counter))  <= SDA;
376
377                   ELSE
378
379                      state       <= start;
380                      RW_CTL_Int     <= '0';           -- Mode is write because master wants send start bit
381                      Tx             <= '0';           -- SDA line becomes low by master because of start bit
382                      Bit_counter <= (others => '1'); -- Bit_counter = 111 (7 in decimal)
383                      Data_Rx        <= (others => '0'); -- Data_Rx = "00000000"
384                      Data_Rx_2      <= (others => '0'); -- Data_Rx_2 = "00000000"
385
386                   END IF;
```

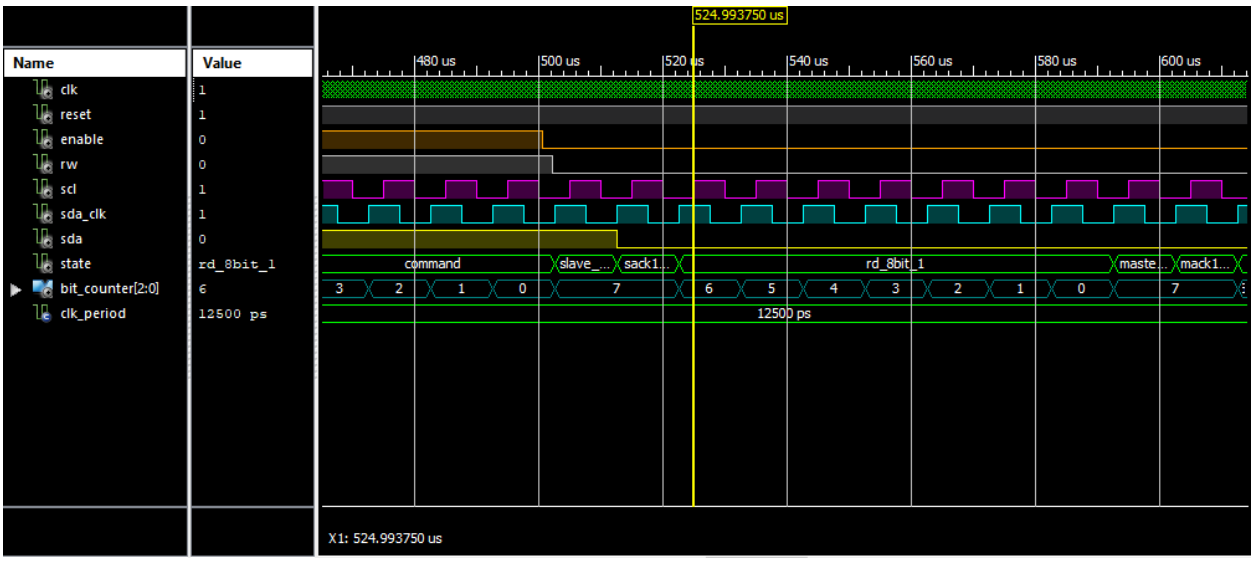
```
388                ELSE                                -- If Enable_Int \= 1, State changes to stop
389
390                   state <= stop;
391                   RW_CTL_Int     <= '0';             -- Mode is write because send stop bit by master(reciever)
392                   Tx             <= '0';             -- As mentioned in datasheet SDA line should become low between no acknow
393                   Bit_counter <= (others => '1');    -- Bit_counter = 111 (7 in decimal)
394                   Address_RW  <= (others => '0');    -- Address_RW = "00000000"
395                   Data_Rx        <= (others => '0'); -- Data_Rx = "00000000"
396
397                END IF;
398
399             WHEN stop =>
400
401                state <= ready;
402                Bit_counter <= (others => '1');       -- Bit_counter = 111 (7 in decimal)
403                Address_RW  <= (others => '0');       -- Address_RW = "00000000"
404                Data_Rx        <= (others => '0');    -- Data_Rx = "00000000"
405
406          END CASE;
```

```
408       ELSIF (SDA_clk_prv = '1' AND SDA_clk = '0') THEN -- falling edge  SDA_clk
409
410          CASE state IS
411
412             WHEN start =>
413
414                IF (SCL_Ena = '0') THEN
415                   SCL_Ena <= '1';                    -- allow to generate SCL_clk
416
417                ELSE
418
419                   NULL;
420
421                END IF;
422
423             WHEN stop =>
424
425                SCL_Ena <= '0';                       -- will not generte SCLC_clk
426                RW_CTL_Int     <= '0';                -- Mode is write because master wants to write stop bit on sda
427                Tx             <= '1';                -- Set 1 on sda because of stop bit (stop bit = SDA becomes high when SCI
428
429             WHEN OTHERS =>
430
431                NULL;
432
433          END CASE;
434       END IF;
435    END IF;
```

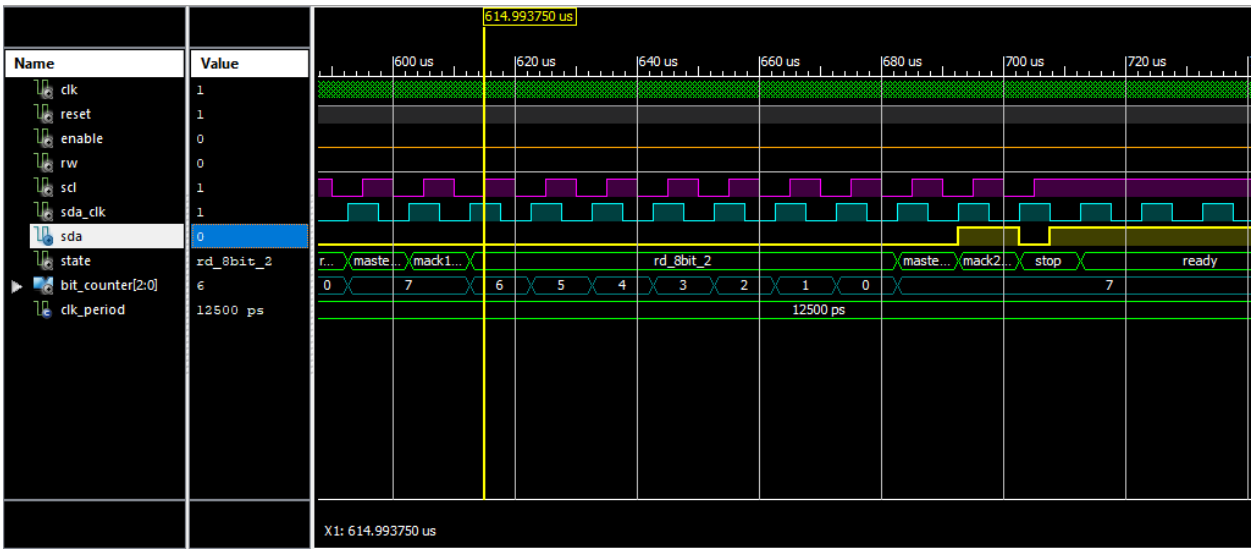
اکنون به بررسی کدهایی که نشان داده شد در شبیه سازی می‌پردازیم.



مشاهده می‌شود در استیت start، SDA صفر شده است که نشان دهنده ارسال start bit می‌باشد. سپس مستر ادرس به همراه بیت r/w را روی SDA قرار داده است و سپس اسلیو بیت اکنالج را ارسال کرده است و چون بیت کم ارزش address_rw، ۱ است پس مستر در مد خواندن می‌رود.

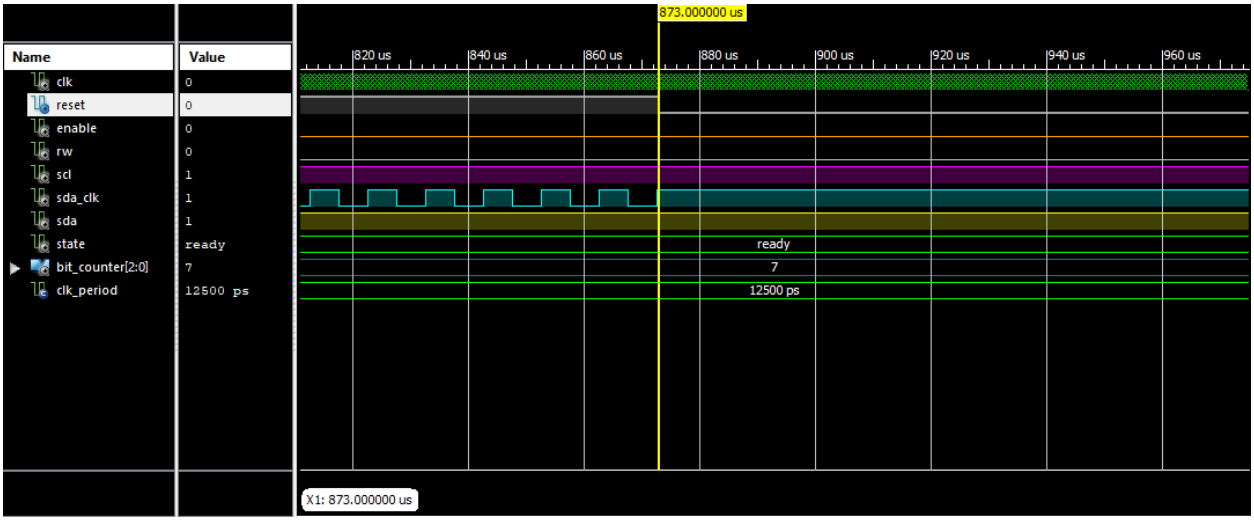


در مد خواندن باید از تست بنچ به SDA مقدار داده شود که در تست بنچ گفته شده است که مقدار 0 به مستر ارسال شود. همانطور که در شکل بالا مشاهده می‌شود مستر ۸ تا 0 خوانده در واقع اسلیو ۸ تا صفر روی SDA قرار داده است. سپس مستر اکنالج به اسلیو ارسال کرده است و مستر شروع به خواندن ۸ بیت دوم می‌کند.



همانطور که مشاهده می‌شود مستر ۸ بیت دوم را نیز خوانده است و مستر باید بیت اکنالج و سپس چون enable = 0 باید stop bit را ارسال کند. در اینجا ارسال بیت اکنالج متفاوت از قبل خواهد بود. در دیتا شیت بیان شده است، مستر برای ارسال بیت اکنالج باید SDA را 1 کند سپس قبل از کلاک بعدی SDA را 0 کند و سپس برای ارسال stop bit مجدد SDA را 1 کند که این موضوع به خوبی در شبیه سازی مشاهده می‌شود.

برای تست حالتی که reset = 0 می‌شود در تست بنچ بیان شده است در زمان 873000 ns ریست اتفاق بیافتد. در شکل زیر در شبیه سازی مشاهده می‌شود که با صفر شدن ریست همه سیگنال‌ها به مقادیر اولیه باز می-گردد.



کد مربوط به تست بنچ در تصاویر زیر مشاهده می‌شود.

```
1
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.ALL;
4  use IEEE.NUMERIC_STD.ALL;
5
6  ENTITY I2C_TB IS
7  END I2C_TB;
8
9  ARCHITECTURE behavior OF I2C_TB IS
10
11     -- Component Declaration for the Unit Under Test (UUT)
12
13     COMPONENT I2C_AD5622
14     PORT(
15          clk : IN  std_logic;
16          Reset : IN  std_logic;
17          Enable : IN  std_logic;
18          RW : IN  std_logic;
19          RW_CTL : OUT  std_logic;
20          SCL : OUT  std_logic;
21          SDA : INOUT  std_logic
22         );
23     END COMPONENT;
24
```

```
26    --Inputs
27    signal clk : std_logic := '0';
28    signal Reset : std_logic := '0';
29    signal Enable : std_logic := '0';
30    signal RW : std_logic := '0';
31
32    --BiDirs
33    signal SDA : std_logic;
34
35
36    --Outputs
37    signal SCL : std_logic;
38    signal RW_CTL : std_logic := '0';
39
40    -- Clock period definitions
41    constant clk_period : time := 12.5 ns;
42
```

```
43 BEGIN
44
45     -- Instantiate the Unit Under Test (UUT)
46     uut: I2C_AD5622 PORT MAP (
47            clk => clk,
48            Reset => Reset,
49            Enable => Enable,
50            RW => RW,
51            RW_CTL => RW_CTL,
52            SCL => SCL,
53            SDA => SDA
54          );
55
56     -- Clock process definitions
57     clk_process :process
58     begin
59        clk <= '0';
60        wait for clk_period/2;
61        clk <= '1';
62        wait for clk_period/2;
63     end process;
64
```

```
64
65     -- Enable Generator
66     process
67     begin
68        Enable <= '0', '1' after 34719 ns, '0' after 63057 ns, '1' after 417480 ns, '0' after 500480 ns;
69        wait;
70     end process;
71
72     -- RW Generator
73     process
74     begin
75        RW <= '0', '1' after 312520 ns, '0' after 502125 ns;
76        wait;
77     end process;
78
79     -- SDA Generator
80     SDA <= 'Z' WHEN RW_CTL = '0' ELSE '0';
81
82     -- Reset
83     Reset <= '1', '0' after 873000 ns;
84
85 END;
```

کد و شبیه سازی مربوط به پروژه در فایل I2C_AD5622 قرار دارد.