

**Budapesti Műszaki és Gazdaságtudományi Egyetem**

**Villamosmérnöki és Informatikai Kar**

**Beágyazott Információs Rendszerek főspecializáció (MIT)**

**Rendszerarchitektúrák**

**AMBA AXI-LITE – SPI**

Készítették:

Cseh Péter (DM5HMB)

Gergely Dániel (T5OCI8)

Varga Ákos (BOA0LG)

Konzulens:

Fehér Béla

2017.05.12.

# 1. SPI Periféria

## Az SPI Protokoll

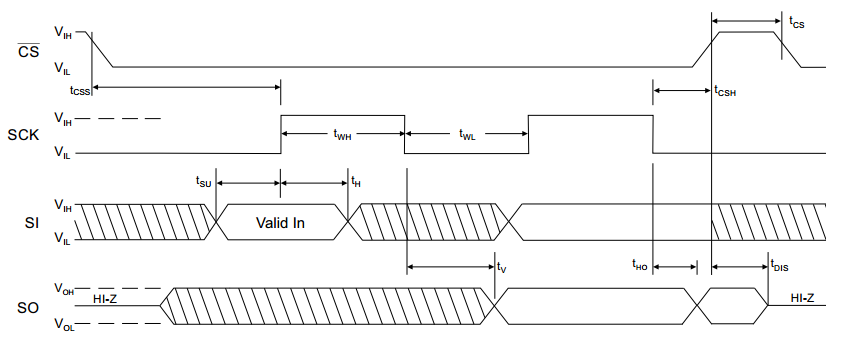
Az SPI (*Serial Peripheral Interface*) egy soros, szinkron *full duplex* kommunikációs protokoll.  
A protokoll négy vezetéket használ, melyek a következők:

* SS: *Slave Select*, amivel a *master* kiválasztja azt a *slave* eszközt, amelyikkel kommunikál
* SCK: *Serial Clock*, az órajel, amely biztosítja a kommunikáció szinkronizáltságát
* MISO: Master Input Slave Output, vagyis a *slave* eszközből küldött adatok a *master* eszköz felé
* MOSI: Master Output Slave Input, a *master* eszköz üzenete a *slave* eszköz számára

A periféria modult úgy terveztük, hogy illeszkedjen a kiválasztott [Atmel AT25010B SPI Serial EEPROM](http://www.microchip.com/wwwproducts/en/AT25010B) memóriához.

## 1.2. Specifikáció

A specifikáció kidolgozása során figyelembe kellett venni az Atmel EEPROM időzítési megkötéseit, illetve egyéb jellemzőket, melyek befolyásolják a kommunikációt.



1. ábra: Az Atmel EEPROM SPI időzítési diagramja

Az *SCK* frekvenciája 0 – 20 MHz közötti értéket vehet fel. 4MHz-es *SCK* jelet választottunk, mert ezt könnyű előállítani a 16MHz-es rendszer órajelből. A memória ilyen kommunikációs sebesség mellett alacsonyabb feszültségen üzemel (1,8V). Az 1. ábrán látható időzítési diagramhoz tartozó időzítési megkötések, az általunk kiválasztott sebesség szerint:

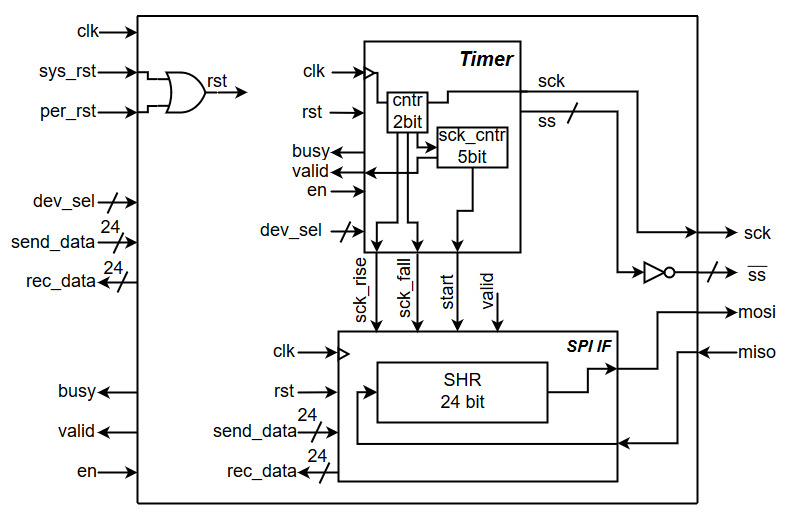
* tcss: CS setup time, minimum 200 ns
* tcsh: CS hold time, minimum 200 ns
* twh: SCK high time, minimum 80 ns
* twl: SCK low time, minimum 80 ns
* tsu: Data In setup time, minimum 80 ns
* th: Data In hold time, minimum 80 ns

Az EEPROM *slave* üzemmódban működik.  
A memóriának hat féle kommunikációs módja van, ezek használatához 24 bitnyi információt vár maximum az eszköz, az első 8 bit a művelet típusa, a második 8 bit a memória címe (művelettípus függő), a harmadik 8 bit a címre beírt adat (művelettípus függő).

(Megjegyzés: Az 1. ábrán a *CS* az *SS*-el, az *SI* a *MOSI*-val, az *SO* a *MISO* vonalakkal egyezik meg.)

## 1.3. Tervezés

### 1.3.1. Blokkvázlat



2. ábra: SPI Blokkvázlat

A 2. ábrán látható blokkvázlat interfészén jobb oldalt láthatóan az SPI jelek, bal oldalt a busz modulhoz kapcsolódó vonalak. A bal oldali jelek:

* clk: A rendszer órajel
* sys\_rst: A rendszer reset jele (active high reset)
* per\_rst: A periféria reset jele (active high reset)
* dev\_sel: A slave eszköz kiválasztó jel, szélessége megegyezik a slave eszközök számával, és így az SS vonalak számával
* send\_data: Az AXI LITE busztól érkező adat az EEPROM felé (szélességét az EEPROM specifikációja definiálja)
* rec\_data: Az EEPROM-tól érkező adat a busz felé (érvényes adatot csak az alsó 8 bit tartalmaz)
* busy: A periféria modul magas logikai szintre húzza ezt a vonalat, ha kommunikáció zajlik a memóriával
* valid: Jelzi, amikor a rec\_data adat érvényes
* en: Ennek a jelnek a magas szintje indítja az SPI kommunikációt

A periféria modul három fő blokkból áll, melyek az *spi\_top*, az *spi\_if* és a *timer*.

(Megjegyzés: a memória WP és HOLD jelét nem hajtom meg az SPI interfészből, feltételezem, hogy ez nem az SPI periféria felelőssége)

#### 1.3.2.1. SPI TOP

Az *spi\_top* tartalmazza a benne lévő modulok összeköttetését, illetve a modulok és a külső interfész közötti összeköttetéseket. Ezek mellett előállítja a *rst* jelet, amely a *timer* és az *spi\_if* blokkok reset jele. A *rst* jel a *sys\_rst* és a *per\_rst* jelek vagy kapcsolata. A blokk továbbá negálja a *timer* blokk *ss* jelét.

#### 1.3.2.2. SPI IF

Az *spi\_if* blokk tulajdonképpen egy 24 bites shift regiszter. Feladata beolvasni a *send\_data* adatot, a soros adat *kishiftelése* a *mosi* vonalon, illetve *beshiftelni* a *miso* jelet. Fontos, hogy ez a modul nem tartalmaz időzítést, azt a *timer* biztosítja.  
Mind a 24 bit *kishiftelése* nem minden esetben indokolt (pl olvasás esetén csak a parancsot és a címet kell (16 bit)), de a memória figyelmen kívül hagyja a többlet biteket, és így az *SPI IF* felépítése egyszerűsödik.

#### 1.3.2.3. Timer

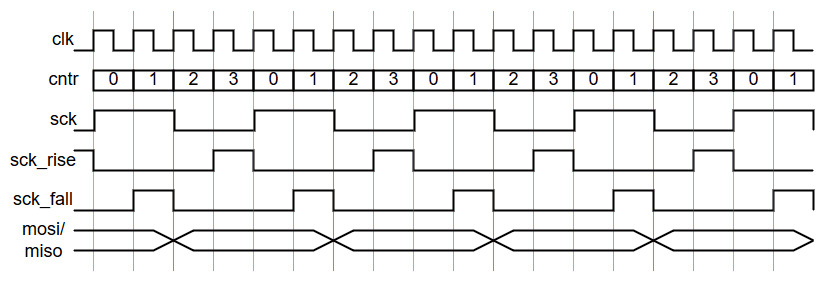
A *timer* tartalmaz minden időzítést, amely a helyes kommunikációhoz szükséges. Ez a blokk, húzza le a kiválasztott slave eszköz *ss* jelét, generálja az *sck* órajelet. Jelzi a *spi\_if*-nek a kommunikáció kezdetét (*start* jel), jelzi az *sck* felfutó- és lefutó élét (*sck\_rise* és *sck\_fall*). Továbbá jelzi a kommunikáció végét a *valid* vonalon, ez mind a *spi\_if* számára, mind az AXI LITE busz számára fontos.

### 1.3.2. Időzítés

A rendszer órajel a feladat kiírás szerint 16 MHz. Az ehhez tartozó periódus idő 1 / 16MHz, ami 62,5 ns, tehát ez a legkisebb időzítés, ami a rendszerben történhet. A specifikációban megadott *sck* órajel frekvenciát a rendszer órajel negyedére választottam (ez alapján twh = twl = 125 ns).  
A tcss = tcsh: min 200 ns-ot 250ns-ra választottam, mivel ez pont négyszerese az órajel periódus idejének. A tsu = th: min 80 ns-ot 125 ns-ra választottam, ez kettő rendszer órajel periódus idő.

Az EEPROM dokumentációja szerint az EEPROM az *sck* órajel felfutó élére mintavételezi a *mosi*-t*(SI*), és az *sck* lefutó élére küldi a *miso*-t(*SO*).

#### 1.3.2.1. SCK Ciklus



3. ábra: SCK ciklus

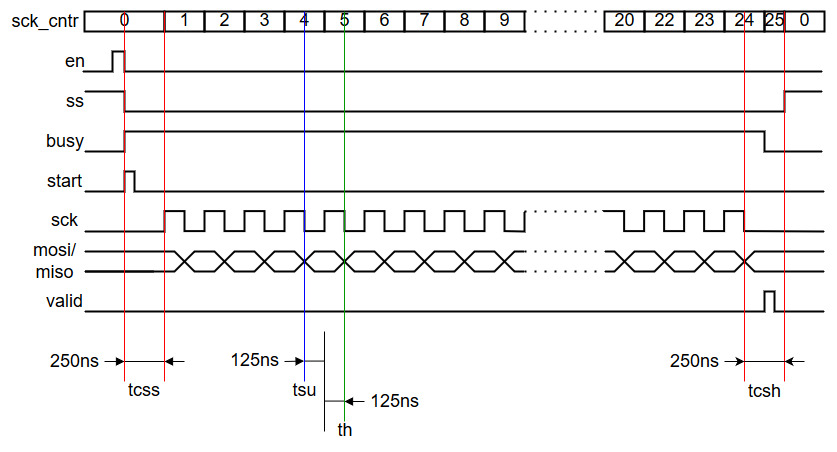
Az időzítési kritériumok alapján egy *sck* ciklust négy részre osztottunk. A négy rész:

1. *sck\_fall* -> magas
2. *sck* je*l* -> alacsony, *sck\_fall* -> alacsony
3. *sck\_rise* -> magas
4. *sck* jel -> magas, *sck\_rise* -> alacsony

Fontos megjegyezni, hogy ezek a jelek egy D-FF-on keresztül terjednek tovább a *timer* blokkon kívülre, tehát egy órajelet késnek. Vagyis például az *sck* felfutó élét egy órajellel korábban kell jeleznünk(*sck\_rise*).

A beérkező adat mintavételezése az *sck\_rise* jelre történik, az *sck* felfutó élénél (függőleges zöld vonal a 3. ábrán). Az adatküldés a *sck\_fall* jelre történik, az *sck* lefutó élénél (függőleges kék vonal a 3. ábrán).

### 1.3.2.2. Kommunikációs ciklus



4. ábra: Kommunikációs ciklus

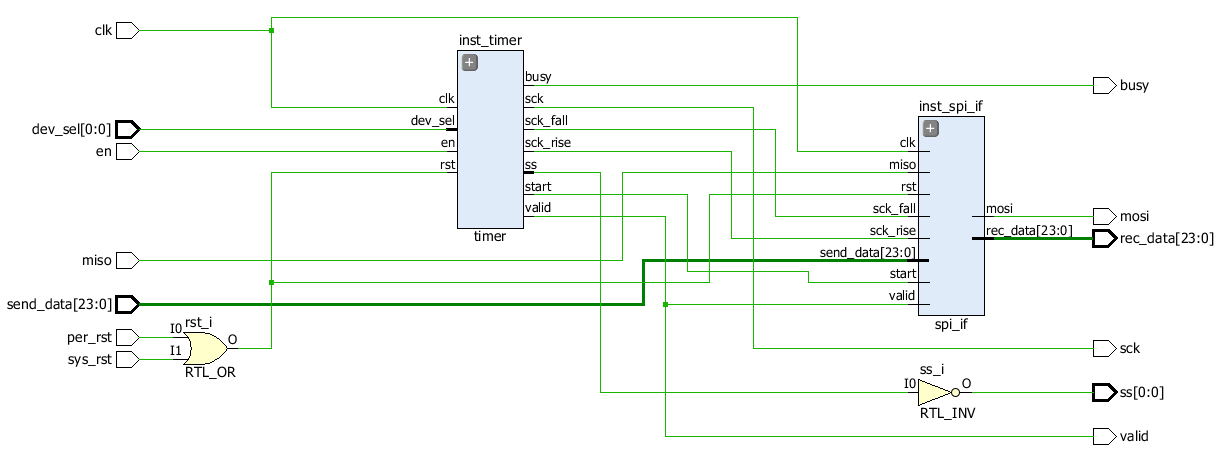
Az SPI kommunikációt az *en* jel magas állapota indítja. Ekkor kiválasztjuk a *slave* eszközt a *ss* jellel, jelezzük a shift regiszternek a kommunikáció kezdetét a *start* pulzussal, valamit jelzést adunk a *busy* vonalon. Ezután 250 ns-mal, a *tcss* betartásával indul a kommunikáció. A kommunikáció végeztével a *valid* vonalon jelzünk, hogy beolvastuk az adatot, a *busy* alacsony logikai szintre állítjuk, valamint a *tcsh* betartásával a *ss* vonalat is visszaállítjuk.

Az állapotok megkülönböztetéséhez egy 5 bites számlálót használunk (*sck\_cntr*) amely 0-tól 25-ig számol, a tényleges adatátvitel a számláló 1 – 24 értékeinél történik, a többi állapot a vezérlő jelekhez szükséges.

# 1.4. Implementáció

Az SPI implementációhoz a Xilinx Vivado 2016.4. fejlesztői környezetet használtuk. A nyelv VHDL.

### 1.4.1. RTL ábra



5. ábra: Generált RTL ábra

Az RTL ábra összehangban van a tervezett blokkvázlattal.

### 1.4.2. SPI TOP kódja

|  |  |
| --- | --- |
| ----------------------------------------------------------------------------------  -- Company: BME  -- Engineer: Cseh Péter(DM5HMB), Gergely Dániel(T5OCI8), Varga Ákos(BOA0LG)  --  -- Create Date: 11.04.2017 21:53:46  -- Design Name: spi  -- Module Name: spi\_top - behavioral  ----------------------------------------------------------------------------------  library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use work.spi\_pkg.all;  entity spi\_top is  port  (  clk : in std\_logic;  per\_rst : in std\_logic;  sys\_rst : in std\_logic;    dev\_sel : in std\_logic\_vector((SPI\_DEVICES - 1) downto 0);  send\_data : in std\_logic\_vector((SPI\_DATA\_WIDTH - 1) downto 0);  rec\_data : out std\_logic\_vector((SPI\_DATA\_WIDTH - 1) downto 0);    busy : out std\_logic;  valid : out std\_logic;  en : in std\_logic;    miso : in std\_logic;  mosi : out std\_logic;  ss : out std\_logic\_vector((SPI\_DEVICES - 1) downto 0);  sck : out std\_logic  );  end spi\_top;  architecture behavioral of spi\_top is  signal rst : std\_logic := '0';  signal sck\_rise : std\_logic := '0';  signal sck\_fall : std\_logic := '0';  signal start : std\_logic := '0';  signal w\_valid : std\_logic := '0';  signal p\_ss : std\_logic\_vector((SPI\_DEVICES - 1) downto 0) := (others => '0'); | begin  --RESET  rst <= (per\_rst or sys\_rst);  --Active 0 slave select  ss <= not(p\_ss);  valid <= w\_valid;  inst\_timer : entity work.timer(behavioral)  port map  (  clk => clk,  rst => rst,      valid => w\_valid,  sck\_rise => sck\_rise,  sck\_fall => sck\_fall,  start => start,    en => en,  busy => busy,  dev\_sel => dev\_sel,    sck => sck,  ss => p\_ss  );    inst\_spi\_if : entity work.spi\_if(behavioral)  port map  (  clk => clk,  rst => rst,    start => start,  sck\_rise => sck\_rise,  sck\_fall => sck\_fall,  valid => w\_valid,    send\_data => send\_data,  rec\_data => rec\_data,    miso => miso,  mosi => mosi  );  end behavioral; |

1. táblázat: SPI TOP kódja

### 1.4.2. SPI IF kódja

|  |  |
| --- | --- |
| ----------------------------------------------------------------------------------  -- Company: BME  -- Engineer: Cseh Péter(DM5HMB), Gergely Dániel(T5OCI8), Varga Ákos(BOA0LG)  --  -- Create Date: 11.04.2017 21:53:46  -- Design Name: spi  -- Module Name: spi\_if - behavioral  ----------------------------------------------------------------------------------  library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use work.spi\_pkg.all;  entity spi\_if is  port  (  clk : in std\_logic;  rst : in std\_logic;    start : in std\_logic;  sck\_rise : in std\_logic;  sck\_fall : in std\_logic;  valid : in std\_logic;    send\_data : in std\_logic\_vector ((SPI\_DATA\_WIDTH - 1) downto 0);  rec\_data : out std\_logic\_vector ((SPI\_DATA\_WIDTH - 1) downto 0);    miso : in std\_logic;  mosi : out std\_logic  );  end spi\_if;  architecture behavioral of spi\_if is  type t\_spi\_if\_fsm is (idle, shift);  signal spi\_if\_fsm : t\_spi\_if\_fsm := idle;    signal shift\_reg : std\_logic\_vector((SPI\_DATA\_WIDTH - 1) downto 0) := (others => '0'); | begin  rec\_data <= shift\_reg;    proc\_comm : process(clk)  begin  if(rising\_edge(clk)) then  if(rst = '1') then  spi\_if\_fsm <= idle;  else  case spi\_if\_fsm is    when idle =>    if(start = '1') then  shift\_reg <= send\_data;    spi\_if\_fsm <= shift;  end if;    when shift =>    --SEND--  if(sck\_fall = '1') then --AT25010B EEPROM is sampling the mosi at sck rising edge -> shifting mosi out at falling edge to keep the setup time  mosi <= shift\_reg(SPI\_DATA\_WIDTH - 1);  end if;    --RECEIVE--  if(sck\_rise = '1') then --received data is valid on the rising edge of the sck  shift\_reg <= shift\_reg((SPI\_DATA\_WIDTH - 2) downto 0) & miso;  end if;    --DONE--  if(valid = '1') then  spi\_if\_fsm <= idle;  end if;    end case;  end if;  end if;  end process proc\_comm;  end behavioral; |

2. táblázat: SPI IF kódja

### 1.4.3. Timer kódja

|  |  |
| --- | --- |
| ----------------------------------------------------------------------------------  -- Company: BME  -- Engineer: Cseh Péter(DM5HMB), Gergely Dániel(T5OCI8), Varga Ákos(BOA0LG)  --  -- Create Date: 11.04.2017 21:53:46  -- Design Name: spi  -- Module Name: timer - behavioral  ----------------------------------------------------------------------------------  library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  use IEEE.NUMERIC\_STD.ALL;  use work.spi\_pkg.all;  --Clock freq: 16MHz -> 1 clock period time: 1/16MHz = 0.0625us = 62.5 ns  --SCK freq: max 5 Mhz -> clk / 4 = 16MHz / 4 = 4MHz -> SCK period time: 1 / 4MHz = 250 ns -> 4 clock period time  --SS setup time: min 200 ns -> 250 ns -> 1 SCK period time  --SS hold time: min 200 ns -> 250 ns -> 1 SCK period time  --data setup time: min 80 ns -> 125 ns -> 2 clk period time  --data hold time: min 80 ns -> 125 ns -> 2 clk period time  --CPOL: 0  --CPHA: 0  entity timer is  port  (  clk : in std\_logic;  rst : in std\_logic;    valid : out std\_logic;  sck\_rise : out std\_logic;  sck\_fall : out std\_logic;  start : out std\_logic;    en : in std\_logic;  busy : out std\_logic;  dev\_sel : in std\_logic\_vector ((SPI\_DEVICES - 1) downto 0);    sck : out std\_logic;  ss : out std\_logic\_vector((SPI\_DEVICES - 1) downto 0)  );  end timer;  architecture behavioral of timer is  signal cntr : integer range 0 to 3 := 3;  type t\_spi\_timer\_fsm is (idle, work);  signal spi\_timer\_fsm : t\_spi\_timer\_fsm := idle;  signal w\_sck : std\_logic := '0';  signal sck\_cntr : integer range 0 to SPI\_DATA\_WIDTH + 1 := 0;  begin  sck <= w\_sck;  proc\_div : process(clk)  begin  if(rising\_edge(clk)) then  if(rst = '1') then  cntr <= 3;  else  if(cntr = 3) then  cntr <= 0;  else  cntr <= cntr + 1;  end if;  end if;  end if;  end process proc\_div; | proc\_timer : process(clk)  begin  if(rising\_edge(clk)) then  if(rst = '1') then  spi\_timer\_fsm <= idle;  else    case spi\_timer\_fsm is    when idle =>  valid <= '0';  sck\_rise <= '0';  sck\_fall <= '0';  w\_sck <= '0';  busy <= '0';  sck\_cntr <= 0;    if(en = '1') then  spi\_timer\_fsm <= work;    start <= '1';  ss <= dev\_sel;  busy <= '1';  else  start <= '0';  ss <= (others => '0');  busy <= '0';  end if;    when work =>  if(cntr = 2) then  sck\_cntr <= sck\_cntr + 1;  end if;    case sck\_cntr is  when 0 =>  start <= '0';    when (SPI\_DATA\_WIDTH + 1) =>  sck\_rise <= '0';  sck\_fall <= '0';  w\_sck <= '0';  if(cntr = 0) then  spi\_timer\_fsm <= idle;  busy <= '0';  valid <= '1';  end if;    when others =>  case cntr is  when 0 => sck\_fall <= '1';  when 1 => w\_sck <= '0'; sck\_fall <= '0';  when 2 => sck\_rise <= '1';  when 3 => w\_sck <= '1'; sck\_rise <= '0';  end case;    end case;    end case;    end if;  end if;  end process proc\_timer;  end behavioral; |

3. táblázat: Timer kódja

### 1.4.4. Package kód

|  |
| --- |
| ----------------------------------------------------------------------------------  -- Company: BME  -- Engineer: Cseh Péter(DM5HMB), Gergely Dániel(T5OCI8), Varga Ákos(BOA0LG)  --  -- Create Date: 11.04.2017 21:53:46  -- Design Name: spi  -- Module Name: spi\_pkg - behavioral  ----------------------------------------------------------------------------------  library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  package spi\_pkg is  constant SPI\_DATA\_WIDTH : integer := 24;  constant SPI\_DEVICES : integer := 1;  end package spi\_pkg; |

4. táblázat Package kód

## 1.5. Verifikáció

A periféria modul megfelelő működését viselkedési szimulációval ellenőriztük.

### 1.5.1. A szimuláció kódja

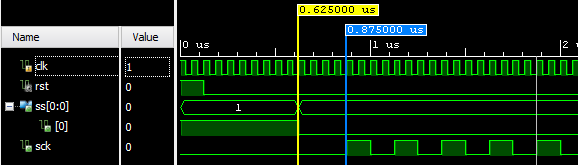
|  |  |
| --- | --- |
| ----------------------------------------------------------------------------------  -- Company: BME  -- Engineer: Cseh Péter(DM5HMB), Gergely Dániel(T5OCI8), Varga Ákos(BOA0LG)  --  -- Create Date: 14.04.2017 14:46:37  -- Design Name: spi  -- Module Name: spi\_top\_sim - sim  ----------------------------------------------------------------------------------  use work.spi\_pkg.all;  library IEEE;  use IEEE.STD\_LOGIC\_1164.ALL;  entity spi\_top\_sim is  end spi\_top\_sim;  architecture sim of spi\_top\_sim is  constant tick : time := 62.5 ns; --16Mhz -> 1/16Mhz = 0.0625 us = 62.5 ns    signal tb\_clk : std\_logic := '0';  signal tb\_per\_rst : std\_logic := '0';  signal tb\_sys\_rst : std\_logic := '0';    signal tb\_dev\_sel : std\_logic\_vector((SPI\_DEVICES - 1) downto 0) := (others => '0');  signal tb\_send\_data : std\_logic\_vector((SPI\_DATA\_WIDTH - 1) downto 0) := x"AA\_55\_AA";  signal tb\_rec\_data : std\_logic\_vector((SPI\_DATA\_WIDTH - 1) downto 0) := (others => '0');  signal tb\_busy : std\_logic := '0';  signal tb\_valid : std\_logic := '0';  signal tb\_en : std\_logic := '0';    signal tb\_miso : std\_logic := '0';  signal tb\_mosi : std\_logic := '0';  signal tb\_ss : std\_logic\_vector((SPI\_DEVICES - 1) downto 0) := (others => '0');  signal tb\_sck : std\_logic := '0';  begin | inst\_dut : entity work.spi\_top(behavioral)  port map  (  clk => tb\_clk,  per\_rst => tb\_per\_rst,  sys\_rst => tb\_sys\_rst,    dev\_sel => tb\_dev\_sel,  send\_data => tb\_send\_data,  rec\_data => tb\_rec\_data,  busy => tb\_busy,  valid => tb\_valid,  en => tb\_en,    miso => tb\_miso,  mosi => tb\_mosi,  ss => tb\_ss,  sck => tb\_sck  );    proc\_clk : process  begin  tb\_clk <= '1'; wait for (tick / 2);  tb\_clk <= '0'; wait for (tick / 2);  end process proc\_clk;    proc\_miso : process  begin  tb\_miso <= '1'; wait for 4\*tick;  tb\_miso <= '0'; wait for 4\*tick;  tb\_miso <= '0'; wait for 4\*tick;  tb\_miso <= '1'; wait for 4\*tick;  end process;    proc\_stim : process  begin  tb\_per\_rst <= '1', '0' after 2\*tick;  tb\_dev\_sel <= "1";  tb\_en <= '0', '1' after 10\*tick, '0' after 11\*tick;  wait;  end process;  end sim; |

5. táblázat: A szimuláció kódja

A szimuláció során az órajel frekvenciájának megfelelő legkisebb időzítéssel dolgoztunk (62.5 ns).

### 1.5.2. Időzítések betartása

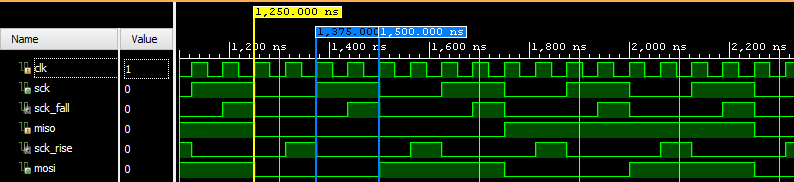
#### 1.5.2.1. tcss



6. ábra: Szimuláció, a tcss betartása

A kurzorok közti *tcss* időtartam 250 ns, ami megfelel a specifikációnak.

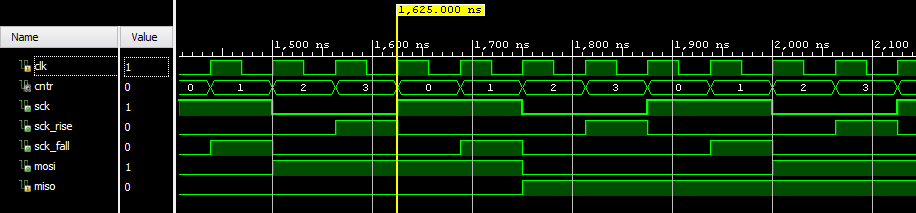
#### 1.5.2.2. tsu és th



7. ábra: Szimuláció, a tsu és th betartása

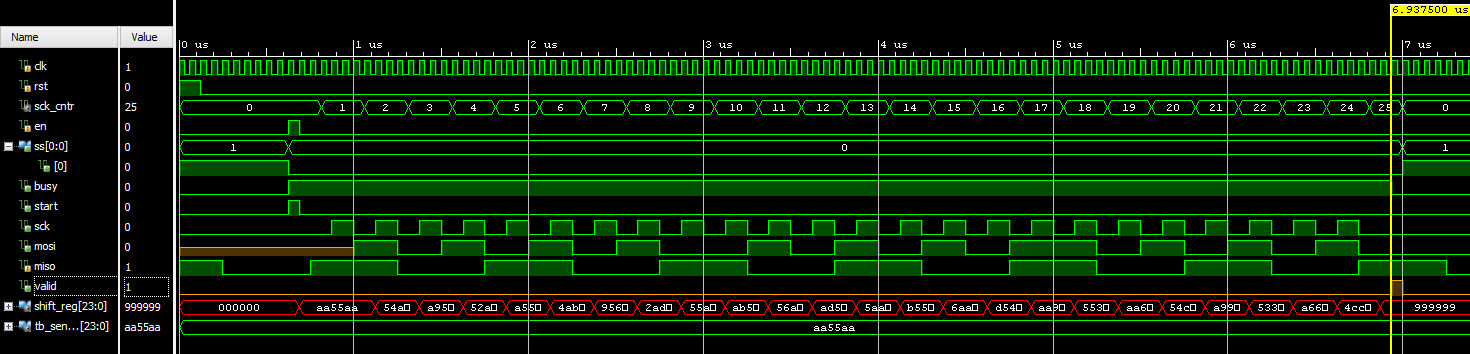
A kurzorok közötti időtartamok, tsu = th = 125 ns.

### 1.5.3. SCK ciklus



8. ábra: Szimuláció, az sck ciklus

### 1.5.4 Kommunikációs ciklus



9. ábra: Szimuláció, kommunikációs ciklus

A szimuláció kódjában a küldendő adat a hexadecimális *0xAA55AA*, tehát a *mosi* vonalon ennek kell megjelennie.

A *miso* vonalra betöltök ’1’-et, majd 4 órajel periódust várakozok, betöltök ’0’-t, várakozok 8 órajelnyi időt, betöltök ’1’-et és várakozok 4 órajelnyi időt; és ezt a ciklust ismételgetem. Tehát a *valid* jelzéskor a beshiftelt értéke a *shift\_reg*-nek csupa bináris *0b1001* mintákból áll, ami a hexadecimális *0x9*-nek felel meg, a szimulációs ábrán ezért lett a *shift\_reg* értéke *0x999999*.