



**Budapesti Műszaki és Gazdaságtudományi Egyetem**  
Villamosmérnöki és Informatikai Kar  
Méréstechnika és Információs Rendszerek Tanszék

# **Rendszerarchitektúrák**

Házi feladat

**AXI - SPI perifériaillesztő**

Kardos Bálint, ZI84PX  
Murányi Péter, A74MW9  
Konzulens: Raikovich Tamás

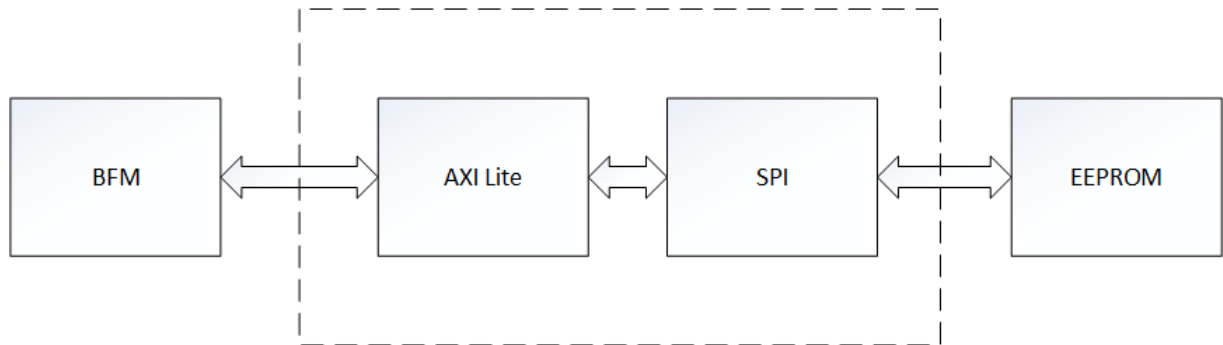
2018. május 18.

# Tartalomjegyzék

<b>1. Feladat</b>	<b>1</b>
<b>2. AXI Lite</b>	<b>1</b>
<b>3. SPI - Serial Peripheral Interface Bus</b>	<b>3</b>
<b>4. Tervezés</b>	<b>5</b>
<b>5. Verilog megvalósítás</b>	<b>6</b>
5.1. AXI Lite illesztő . . . . .	6
5.2. SPI modul . . . . .	7
<b>6. Szimuláció</b>	<b>7</b>
6.1. BFM . . . . .	8
6.2. EEPROM . . . . .	8
6.3. Szimuláció eredménye . . . . .	9
<b>7. Forráskód</b>	<b>11</b>
7.1. SPI . . . . .	11
7.2. Busz illesztő . . . . .	13
7.3. Szimulációs kód . . . . .	20
<b>Hivatkozások</b>	<b>24</b>

## 1. Feladat

A félév során egy SPI Master periféria Verilog nyelven történő fejlesztése volt a feladat, amely egy AXI Lite perifériabusz vezérlőre került illesztésre. Az így elkészített rendszer működőképessége szimulációval került ellenőrzésre. Szimulációban a perifériabusz vezérlését BFM valósítja meg, ami úgy tesz, mintha AXI Lite Master lenne az AXI Lite időzítési kritériumait figyelembe véve. Az SPI Masterre egy Microchip által fejlesztett EEPROM Verilog modul csatlakozik. A szimuláció az EEPROM-ba történő írást és ennek a beírt adatnak a visszaolvasását valósítja meg.

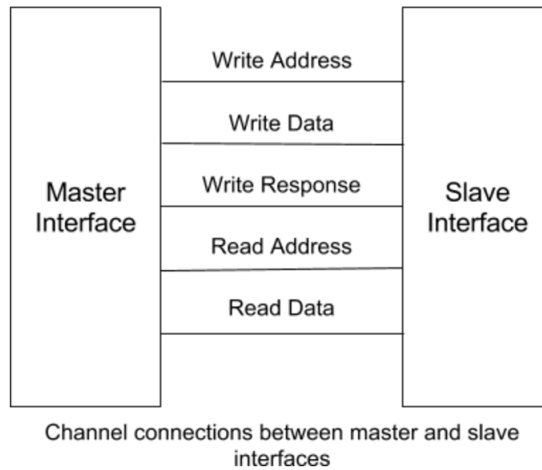


1. ábra. A kitűzött feladat blokkdiagramja

## 2. AXI Lite

Az AXI (Advanced eXtensible Interface) - az AMBA specifikáció része, amely interfészt számos SoC gyártó alkalmazza rendszereiben manapság. A protokoll egyszerű szabályokat fektet le annak érdekében, hogy a chipen található modulok hogyan kommunikáljanak egymással handshake jellegű folyamattal kezdve minden tranzakciót. A protokoll egyszerű, összegezve a legfontosabb jellemzői:

- bármely vezérlő jel/cím/adat átvitele előtt a master és a slave között handshake folyamat zajlik le, amelyre a valid és a ready jelek szolgálnak,
- különböző fázisban kerül átvitelre a vezérlő jel/cím és adat,
- külön csatornák állnak rendelkezésre a vezérlő jel/cím és adat átvitelére,
- burst-jellegű kommunikáció biztosítja a folyamatos adattranszfert.

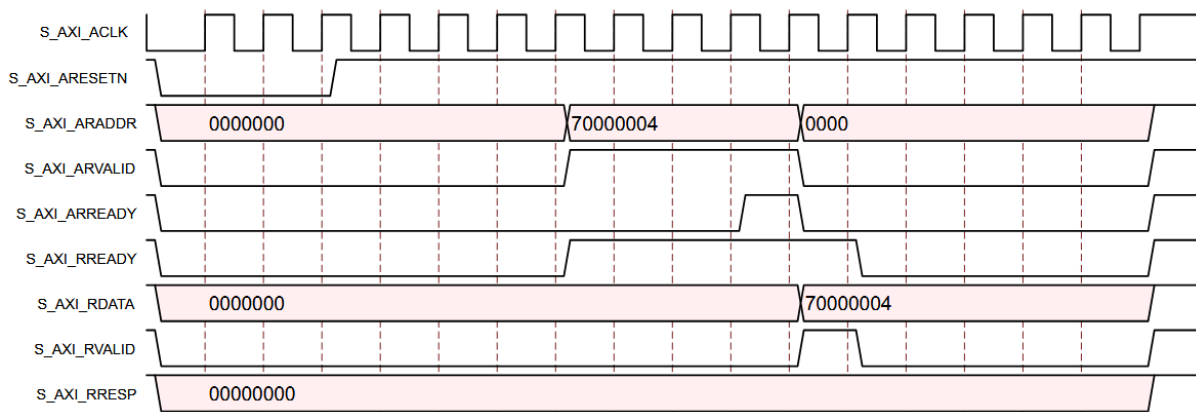


2. ábra. A master és slave eszköz csatorna összeköttetései

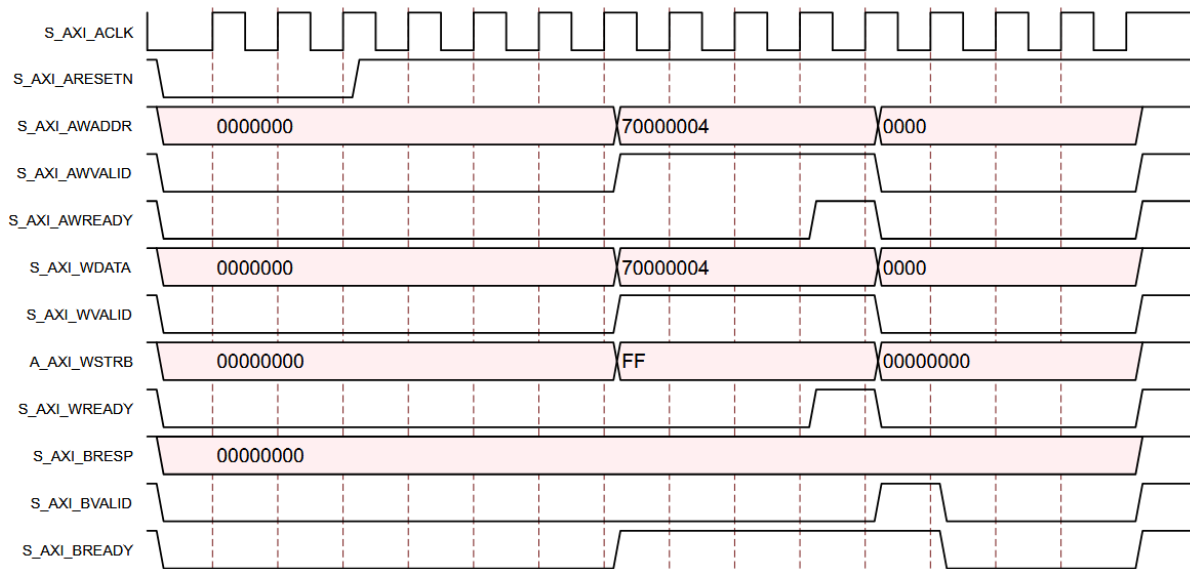
Az interfész először a master és a slave eszköz között kapcsolatot létesít. A két eszközt 5 külön csatorna köti össze:

- Read Address
- Write Address
- Read Data
- Write Data
- Write Response

A valid és ready jelek mindegyik csatornán megvannak így biztosítva a handshake-et. Bármely jeltovábbítása esetén a forrás csatorna aktív valid jelet szolgáltat, amelyre a cél csatorna aktív ready-vel válaszol. Amennyiben mindkét jel aktív, úgy megkezdődhet a kommunikáció. A vezérlő jele/címek és adatok külön fázisokban kerülnek átvitelre, így fontos, hogy megtörténjen a címzés még az adattranszfert megelőző handshake előtt a két eszköz között.



3. ábra. AXI4-Lite Single Read működés



4. ábra. AXI4-Lite Single Write működés

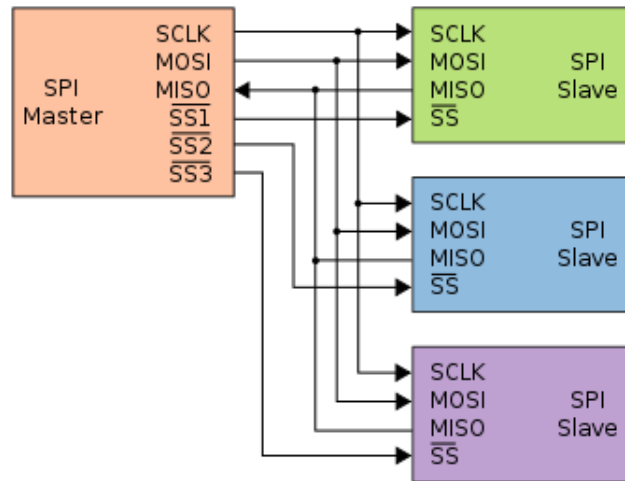
### 3. SPI - Serial Peripheral Interface Bus

A Serial Peripheral Interface busz egy szinkron soros interfész, amely elsősorban beágyazott rendszerekben fordul elő, ahol az egymással kommunikálni kívánó eszközök közötti távolság rövid. Az SPI interfész shift regisztereken alapul. A kommunikáció master-slave jellegű, ahol a master vezérli a kommunikációt: szolgáltatja az órajelet, valamint egy külön vezetéssel engedélyezi a hozzákapcsolt slave perifériákat.

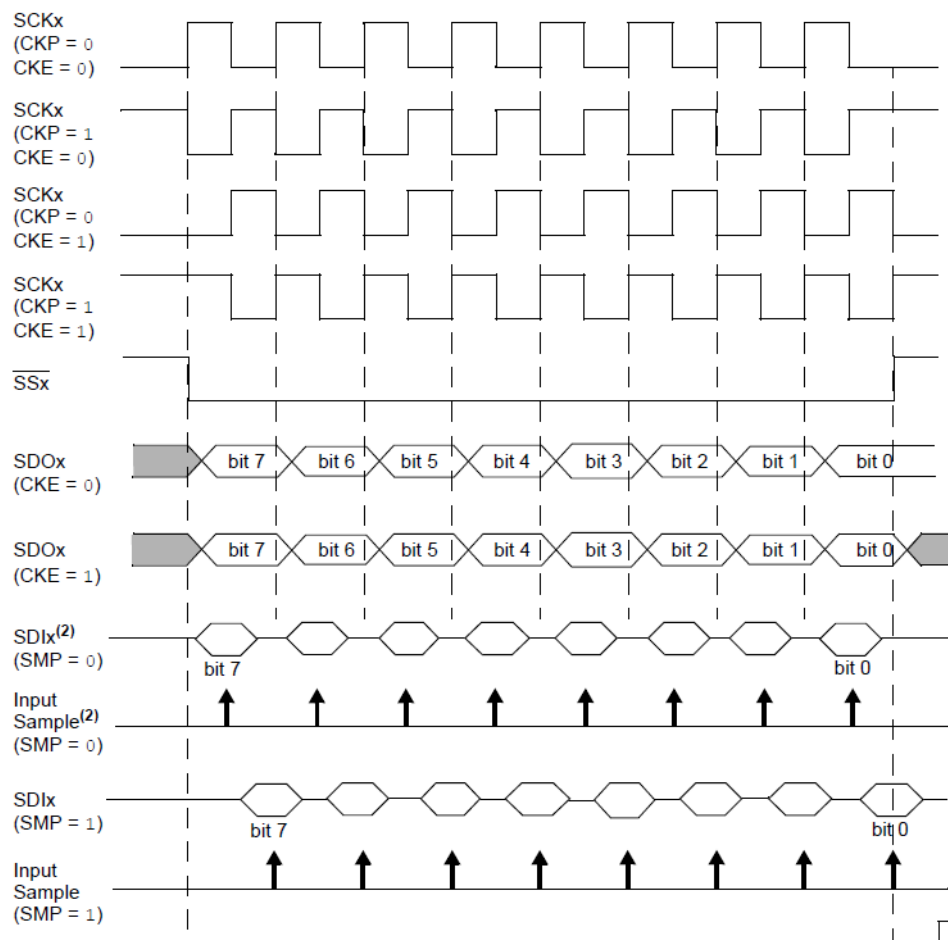
Ennek az a hátránya, hogy minden slave perifériához külön kiválasztó vezetéket kell használni. A szükséges I/O lábak csökkentése érdekében használhatunk dekódert. Az SPI busz jelvezetékei:

- SCLK: órajel, amit a master biztosít,
- MOSI: a master eszköz soros adatkimenete, amely a slave soros adatbemenetére csatlakozik,
- MISO: a master eszköz soros adatbemenete, amely a slave soros adatkimenetére csatlakozik,
- $\overline{CS}_n$ : slave eszköz kiválasztó jele, ahol n az n-edik eszközhöz tartozó engedélyező jele.

A kommunikáció megkezdéséhez a master biztosítja az órajelet és logikai 0-ba állítja a megfelelő  $\overline{CS}_n$  slave kiválasztó jelét. A kommunikáció egyszerre kétirányú - vagyis míg a léptetőregiszter bemenetén sorosan fogadja a biteket ugyanakkor a regiszter másik végén kiléptetjük annak előző tartalmát. Ezért például 16 órajellel a master és slave 16 bites shift regisztereinek a tartalma helyet cserél. Az SPI periféria fontos tulajdonsága az órajel élének és polaritásának megválasztása.



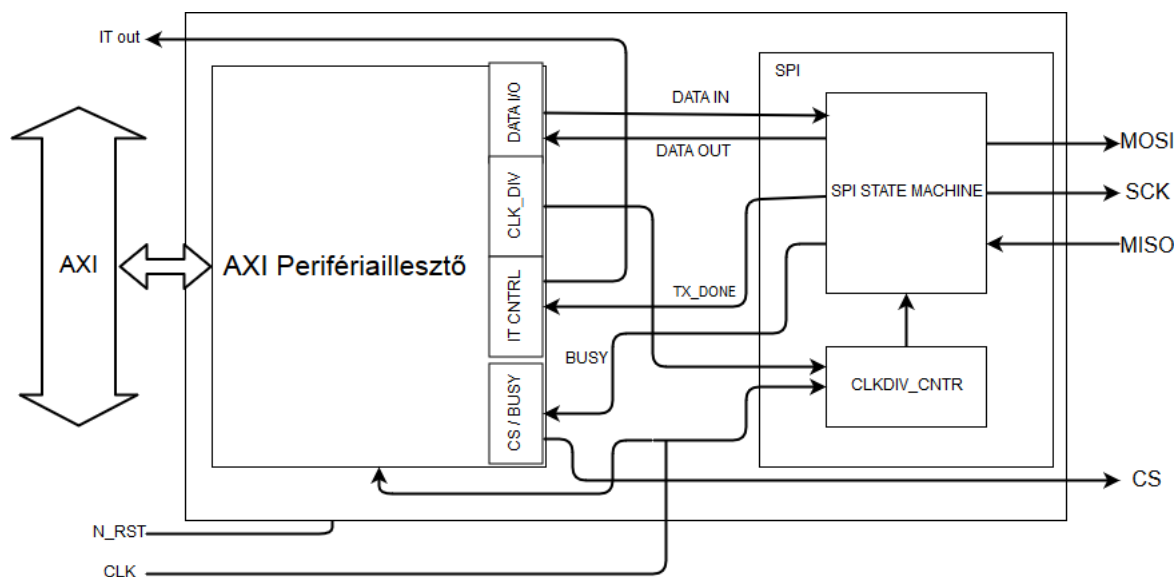
5. ábra. SPI busz: 1 master és 3 független slave eszköz



6. ábra. Az SPI periféria órakonfigurációs beállításai

## 4. Tervezés

Az eszköz két elkülöníthető részből fog állni. Egy AXI Lite 32-bites illesztőből, és magából az SPI kommunikációt lebonyolító master modulból. Mivel az eszköz az AXI Lite interfészen keresztül mint egy memory-mapped eszköz jelenik meg, így az eszköz I/O regisztereken keresztül lesz konfigurálható. Szükség van így egy AXI Lite slave eszközre ami kellő mennyiségű regisztert illeszt az AXI buszra. Az SPI modulban kell egy eszköz ami megvalósítja a kommunikációt, illetve kell egy külön órajelosztó, ami a számunkra szükséges SCK órajelet állítja elő. Mindezen felül az AXI illesztő és SPI modul közé szükséges további logika ami elvégez olyan funkciókat mint pl. az interrupt generálása, SPI START jel generálása.



7. ábra. Az eszköz blokkdiagramja

Az eszköz regisztereken keresztül lesz konfigurálható. Szükség van egy adat be és kimeneti regiszterre, ami úgy legyen megvalósítva hogy íráskor egy új transzfert indít el automatikusan. Szükség van továbbá egy regiszterre amiben megadhatjuk hogy az SCK a rendszer órajelének hányadrésze legyen osztva. Sok mikrovezérlőhöz hasonlóan, és a kivitelezés egyszerűsítése érdekében ez az osztás kettő hatványa lehet csak. A regiszter 3 bites, így az elérhető osztás 2-től 256-ig állítható, ami 100MHz-nél 50Mhz - 390kHz tartományt jelent. Szükség van továbbá egy regiszterre ami tartalmaz egy jelző bitet ami az SPI ciklus befejeződését jelzi, illetve egy bitre amivel a megszakítást lehet ki-be kapcsolni. Szükség van további két bitre, egyre amivel a  $\overline{CS}$  kimenetet lehet vezérelni, illetve egy státuszbitre ami megmutatja hogy jelenleg épp folyamatban van-e egy küldés/fogadás. Az alábbi (8. ábrán) látható a tervezett regiszterkészlet.

	7	6	5	4	3	2	1	0
0	[7:0] DATA I/O							
4	X				[2:0] CLK_DIV			
8	X						IT EN	TX DONE
C	X						CS	BUSY

8. ábra. A periféria regiszterei

## 5. Verilog megvalósítás

### 5.1. AXI Lite illesztő

Az AXI lite illesztő a Xilinx saját példája alapján lett megírva. Minden csatorna elkülöníthető, a funkciójukat önállóan valósítják meg. Mivel általában az ehhez hasonló rendszerek active-low reset-et használnak, így itt is ez lett alkalmazva. A reset és órajel vonalak minden egységhez csatlakoznak, a reset a rendszer minden elemét egy megadott indítási állapotba hozza.

Mivel regisztereket nem szabad egyszerre több helyről is írni, ezért a bemeneti és kimeneti regisztereket el kell különíteni az AXI illesztőn belül. Egyes regiszterek értékét csak az AXI master változtatja, ezeket egyszerűen vissza lehet kötni a kimeneti regiszterekbe, viszont egy csomó regiszter más értéket ad ki olvasáskor, mint amit beleírtunk, ezeknek a kezelésére külön logika szükséges.

Az írási csatornát kezelő logika végzi a bejövő adatok kezelését. Érvényes AWVALID jelre, ha éppen nem folyik írás, jelez az AWREADY vonalon hogy képes érvényes cím fogadására és mintavételezi a AWADDR csatornát. Ezután ugyanez a handshake folyamat megy végbe az írási adat csatornán is. Ha mindkét csatornán érvényes volt a tranzakció, megtörténik a tényleges írás a címzett adatregiszter AXI\_WSTRB által megadott megfelelő bájt vonalaiba. Ezzel együtt történik az írás válasz generálása a saját csatornáján, ami itt mindig nulla: írás OK. Mivel az eszköz úgy lett megtervezve hogy az adat I/O regiszterbe való írás indít egy új SPI tranzakciót, így szükség van arra hogy detektáljuk mikor történik írás a regiszterbe. Erre egy külön regiszter lett létrehozva ami 1-be állítódik amikor az AXI Lite nullás regiszterébe történik írás, 0-ba ha bármi más történik. Ezzel egy egy órajelű start jelet tudunk létrehozni amit az SPI modulnak továbbítva megtörténhet az írás.

Az olvasási csatorna végzi a master felől jövő írási kérések kiszolgálását. Az olvasás mindössze két csatornát igényel. Érvényes olvasási handshake után a mintavételezett címnek megfelelő regiszterből megtörténik az adat kiírása az olvasási válasz státusszal (itt mindig nulla) együtt. A kimenő regiszterek rendre mind wire adattípust használnak, ezek meghajtása vagy egy belső regiszterről történik, aminek az értékét valamilyen logika határozza meg, vagy fixen nullára vannak bekötve (mivel 32 bitesek az AXI Lite regiszterek, míg legfeljebb is csak 8 bit van meghajtva egy regiszterben).

A regiszterek funkciójuktól függően különbözően viselkednek. Az I/O regiszter írási oldalról az SPI adatbemenetére kapcsolódik, míg olvasási oldalról az adatkimenetéhez van kötve. Ennek a regiszternek az írása egyben egy új SPI kommunikációt is elindít. Az órajelosztást állító regiszter a beírt értékét adja vissza olvasáskor, illetve közvetlenül csatlakozik az SPI clkdiv bemenetéhez. Az ITENABLE és TXDONE biteket tartalmazó regiszter ennél összetettebben működik. A TXDONE bit minden írás befejeződésekor magas állapotba kerül. Innen nullázható egy 1-es beírásával, illetve egy új SPI ciklus indításával, ami automatikusan nullázza ezt az értéket. A negyedik regiszter a busy és  $\overline{CS}$ -t vezérlő biteket tartalmazza, előbbi csak olvasható, utóbbi írható és olvasható is.

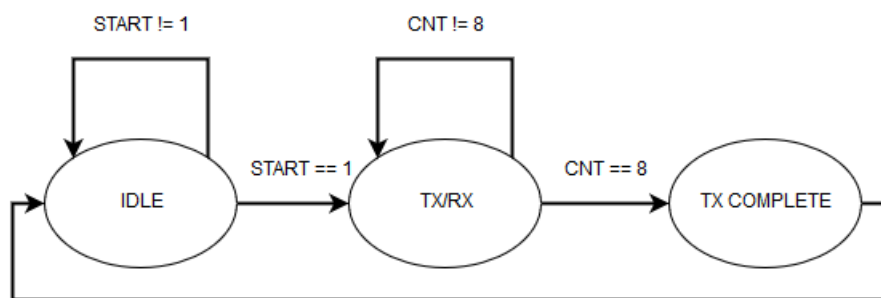
Ezekén túl itt van elhelyezve az SPI-t indító logika, illetve a megszakítás generálása. Az SPI indítása egy egyszerű logikai ÉS kapcsolat, ami az I/O regiszterbe történő írást jelző bitet, az SPI használatát jelölő "busy" bit negáltját és CS bit állását fogja össze. Ebből egy egy órajeses start impulzus jön létre ami elindítja az SPI modult. Az interrupt jel hasonló módon, adott bitek megfelelő állása esetén generálódik. Figyelembe veszi, hogy aktiváltuk e az SPI TX\_ENABLE bitjét, és az SPI modul TXDONE kimeneti bitjét figyeli. Ha ez a kettő egyszerre igaz, egy egy



órajeles megszakítást generál, jelölve, hogy befejeződött az SPI kommunikáció.

## 5.2. SPI modul

Az SPI modul két részből áll. Egy órajel osztó számlálóból, illetve az SPI-t vezérlő állapotgépből. Az órajelosztó számláló egy egyszerű 8-bites számláló, aminek a CLKDIV által megadott bitjét csatoljuk ki az SCK órajel megadásához. Ez a jel lesz egyben a kimeneti SCK órajel, illetve ezt használja az állapotgép az adat küldés ütemezésére. A számláló nem szabadonfutó, csak SPI kommunikáció indításakor indul el, addig reset állapotban vár, aminek következményében a kimenete, így SCK nulla lesz.



9. ábra. SPI állapotgépe

Az SPI ciklust egy állapotgép kezeli. Induláskor az IDLE, várakozó állapotban van. Ebben az állapotban a kimeneteket nullázva egy START jelre várakozik. Ha ezt a jelet megkapja, a BUSY jelzőbitet magasra húzza, beolvassa a bemeneti adatot és egy shift regiszterbe menti melynek az MSB-je a MOSI kimenetre van kötve, majd az állapotváltozót megváltoztatva a TX/RX állapotba kerül. Ebben az állapotban az órajelosztó számlálóból kapott jelet figyeli. Mivel minden a főórajel felfutó élére történik, így külön logika kell amivel ellenőrizzük hogy a leosztott órajel felfutó vagy lefutó élen van-e. Ezt egy átmeneti tárolóval oldottuk meg. Ezt minden ciklusban frissítjük a leosztott órajel aktuális állapotával és ezt a két régi és új jelet vizsgáljuk hogy eldöntsük, felfutó vagy lefutó órajele van-e az SCK jelnek. Felfutó élen a MOSI bemeneten lévő adatot egy bemeneti shift-regiszterbe mintavételezzük. Lefutó élen a kimeneti shift regiszter shiftelődik eggyel balra, illetve egy számláló inkrementálódik, ami számolja a kiküldött bitek számát. Ha ennek a számlálónak az értéke eléri a 8-at, tehát az utolsó felfutó él után lefut az SCK az állapotgép a TX COMPLETE állapotba vált. Ebben az állapotban a bejövő shiftregiszter értékét átmásolja az adat kimenetre, továbbá a TXDONE jelet magas értékre húzza. Ebből az állapotból mindig az IDLE állapotba kerül a következő órajelciklusban, ahol a TXDONE és BUSY jelek nullára állnak vissza jelezve hogy új küldésre készen áll.

## 6. Szimuláció

A létrehozott periféria működése szimulációval lett ellenőrizve. A szimulációhoz a Vivado beépített szimulátorát használtuk. A perifériához az AXI interfész felől egy AXI-LITE Master-t szimuláló BFM lett illesztve, míg az SPI oldalról egy Microchip EEPROM funkcionális verilog modellje lett illesztve. A szimuláció egy az EEPROM-ba való írást, majd onnan a beírt adatok kiolvasását valósítja meg.

A szimuláció 100MHz-s órajelet használ, a reset vonal induláskor 20ns ideig aktív alacsony állapotban van, majd visszatér magas szintre, ami elindítja az összes eszközt. Ezek után futnak le az AXI taszkok, amik inicializálják az SPI perifériát, beírnak 3 bájtot az illesztett EEPROM-ba, majd ezeket ugyanonnan kiolvassa. Az egész szimuláció 20us-ot vesz igénybe.

Az SPI modul /16 órejelosztást használ, amivel az SCK 6,25 MHz lesz. A SPI megszakítás küldője be lett kapcsolva ugyanis ezt használjuk az SPI tranzakciós ciklus befejeződésének figyelésére. A  $\overline{CS}$  jelet manuálisan állítjuk, az SPI megfelelő regiszterébe való írással.

## 6.1. BFM

A busz funkcionális modell egyszerű verilog taszkokkal lett megvalósítva. Ezek a taszkok az AXI LITE interfészt ismertető részben bemutatott időzítési diagram szerint lettek megírva. A taszkokban az órajel felfutó éle után mindenhol egy 1ns-os késleltetés lett elhelyezve, hogy szimuláljuk a rendszerben a flip-flop-ok set-up és hold késleltetésüket. Az írási taszk két bemenettel rendelkezik, írási cím és írási adat. Az olvasási taszk egy bemenet és egy kimenettel rendelkezik: az olvasandó adat címe és a beolvasott adat. Verilog taszkok kimenetein szimulációban csak a taszk befejeződését követően jelenik meg az új adat, attól függetlenül hogy a taszkban hol történt ténylegesen a kimeneti regiszter írása.

## 6.2. EEPROM

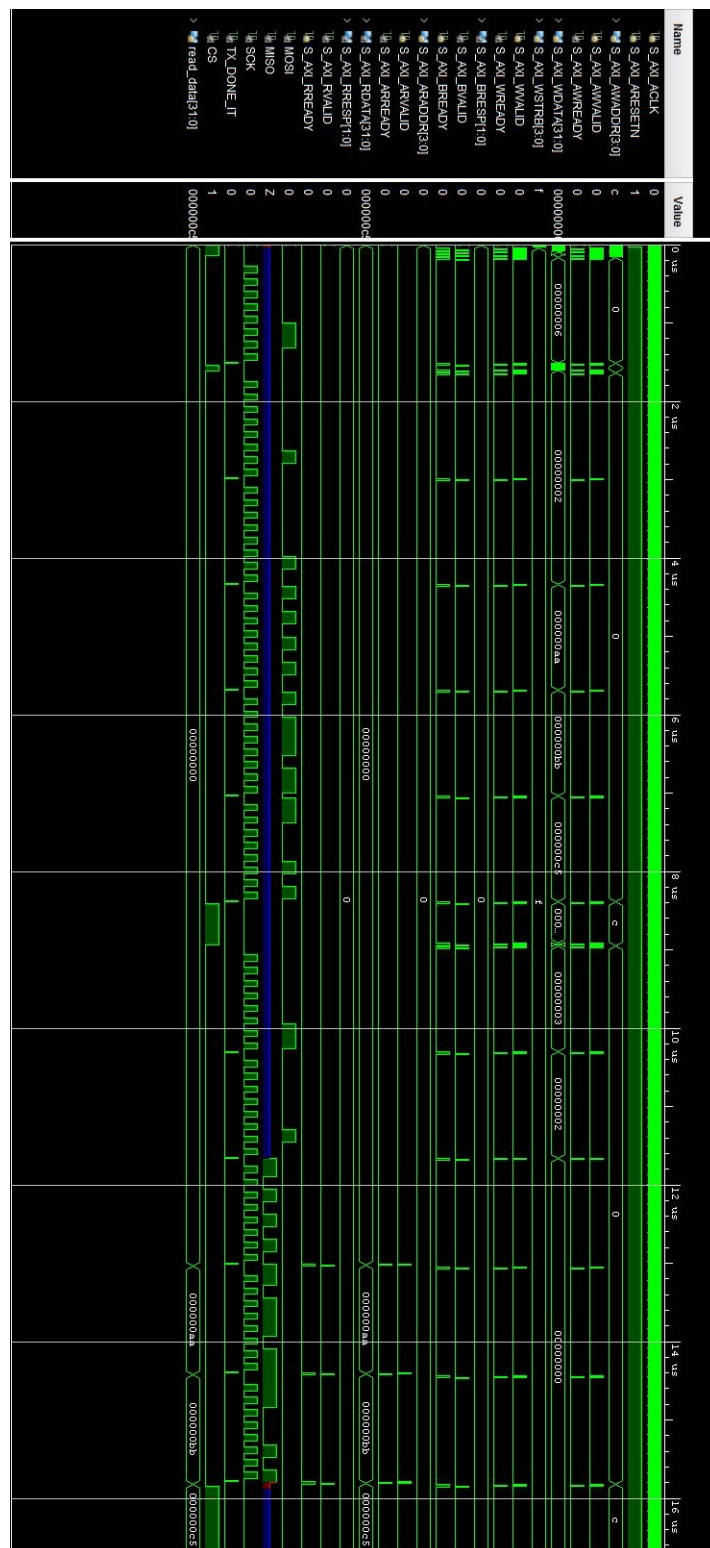
Az eszközhöz egy Microchip 25AA010A EEPROM szimulációs verilog modellje van illesztve, hogy a valós működés is tesztelve legyen. Ez a modul egy 1kbit méretű flash EEPROM-ot valósít meg, valós időzítési értékekkel. Az eszköz leírása [ezen a linken](#) érhető el.

Az eszköz valós időzítéseket is szimulál, ezért például az SCK órajel nem lehet gyorsabb 10MHz-nél, ugyanis az eszköz nem lesz képes elég gyorsan reagálni az órajel változására. Ezt az SPI órajelosztójának megfelelő beállításával lehet elérni.

Hogy az eszközt írassuk, először egy instrukciót kell neki küldeni ami engedélyezi az írást az eszközön belül. Ehhez egy külön írási ciklusra van szükségünk ahol  $\overline{CS}$  jelet aktív alacsonyra kell húzni, majd az írásengedélyező (WREN) parancsot kiküldve a  $\overline{CS}$  jelet egy órajelciklus idejére vissza kell engedni magas szintre. Ha ezt nem tesszük meg, nem fogunk tudni írni az eszközbe. Ezek után lehet ténylegesen írni az eszközt. Íráskor először egy 8-bites írás instrukciót kell kiküldeni, ez után jöhet a cím, szintén 8-biten, majd ezt követően kezdődik meg a tényleges adat kiküldése. Lehet bájtos, illetve burst módban írni, viszont burst módban maximum 16 bájtot lehet kiküldeni, ugyanis az EEPROM csak ekkora bemeneti bufferrel rendelkezik. Az írást a  $\overline{CS}$  magasra húzásával lehet befejezni, ekkor az írásengedélyező belső flag automatikusan 0-ra áll vissza. Az EEPROM verilog modelljében apró változtatásokat kellett eszközölni, ugyanis a valóságot szimulálva 5ms-ig késleltet mielőtt az adatot ténylegesen beírja az adattároló regisztereibe. Ezt hogy a szimuláció rövidebb legyen, és a kapott hullámformák átláthatóbbak legyenek 100ns-re csökkentettük.

Olvasáshoz nem kell speciális parancsot megadni, elég az olvasás instrukció és olvasandó adat kezdőcíme, majd annyi bájtot tudunk kiolvasni amennyit akarunk, az EEPROM minden bájt után automatikusan növeli a pointer értékét. Olvasásnál nincs 16 bájtos limit, amíg el nem érjük a címtartomány végét, folyamatosan tudunk adatot kiolvasni. Az EEPROM verilog modellje a valóságot szimulálva, ha éppen nem küld adatot a MASTER felé a soros adatkimenet (SO) lábát magas impedanciás módba rakja (ez látható is a szimulációban).

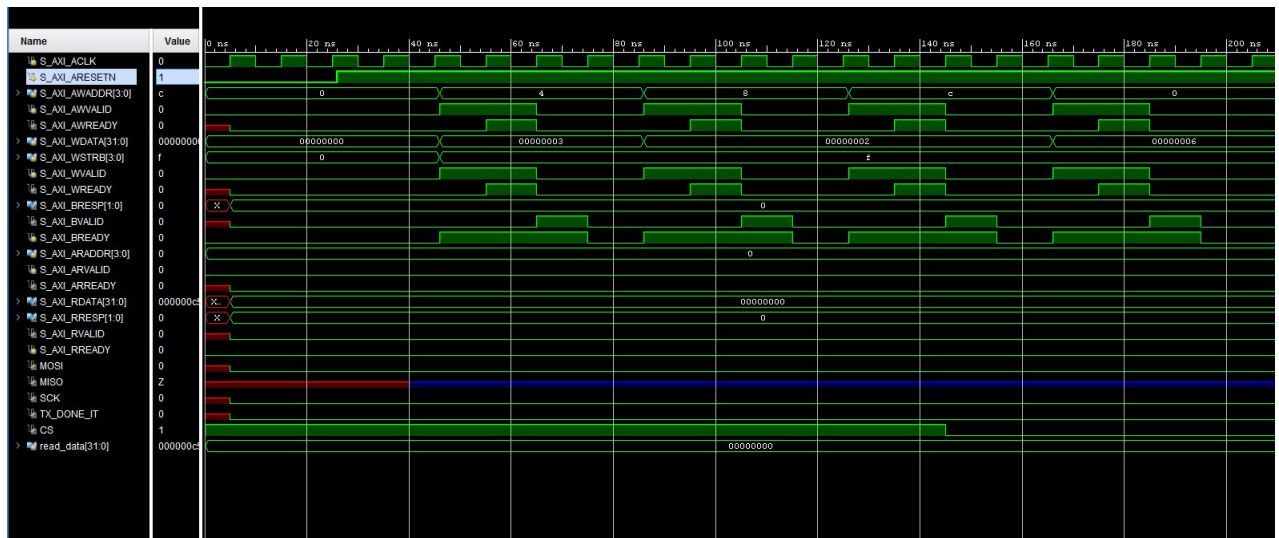
### 6.3. Szimuláció eredménye



10. ábra. A teljes szimuláció

A 10. ábrán látható a teljes szimuláció eredménye. Láthatóak az AXI periféria írások és olvasások, illetve az SPI ciklusok. A beírt 3 adatbájt olvasáskor megjelenik a kimeneten, ebből

látható hogy az SPI illesztő és az EEPROM is megfelelően működik. Ezeken felül a Vivado lehetőséget ad hogy a belső regiszterek állását is ellenőrizni lehessen, ha ezt megtesszük látható lesz hogy az EEPROM-nak valóban beíródott a belső regisztereibe az adat.



11. ábra. AXI írási ciklusok

A 11. ábrán látható egy AXI írási ciklus. Látható hogy a cím és adatcsatorna egyszerre működtethető, mindkettő jelzi hogy új adat érkezik, majd vár a slave felől a válasz jelre hogy a handshake megtörténjen és az adat beíródjon a slave eszközbe. Az írásra a slave eszköz a write-response csatornán választ ad a megfelelő hibakóddal (esetünkben csak 0 lehet) és ezzel a master befejezi az írási ciklust. Mivel az AXI Lite nem támogat burst átvitelt így itt minden adatot egy komplett írási ciklussal kell átvinni.



12. ábra. SPI kommunikáció

Maga az SPI működése a 12. ábrán látható. Látszik hogy a feladatban megadott üzemmód szerint felfutó SCK jelre történik az adat mintavételezése, míg lefutó élre az új adat kiküldése. Látszik hogy amíg az EEPROM-nak nem küldünk írási utasítást a kimenete (MISO) magas impedanciás állapotban van. Látható hogy a TXCOMPLETE interrupt minden tranzakció végén egy óralejre megjelenik, ezzel jelezve a ciklus befejeződését. Érdekes megfigyelni hogy mivel az EEPROM valós időzítésekkel szimulál, az új adatot csak az SCK lefutása után jelentős idővel küldi ki.



13. ábra. AXI olvasási és írási ciklus

A 13. ábrán látható egy AXI olvasási ciklus. Látszik hogy a címet megadva és a handshake után megjelenik a kiolvasandó adat a megfelelő csatornáján.

## 7. Forráskód

### 7.1. SPI

```
'timescale 1ns / 1ps

module SPI_MASTER( //Clock and reset
    input clk,
    input rstn,
    //Registers
    input [2:0] clk_div,
    input [7:0] data_in,
    output reg [7:0] data_out,
    output reg busy,
    output reg tx_complete,
    input start_tx,
    //IO
    output reg sck,
    output MOSI,
```

```

        input MISO

    );

//Clock divider
reg [7:0] cntr = 0;
reg cntr_enable = 0;

always @(posedge clk)
begin
    if((~rstn)||(~cntr_enable))
        cntr <= 0;
    else
        cntr <= cntr + 1;
end

wire spi_tick = cntr[clk_div];

//Controll state machine
reg [1:0] state = 0;
reg [7:0] TX_BUFFER;
reg [7:0] RX_BUFFER;
reg spi_tick_prev;
reg [3:0] tx_count;

assign MOSI = TX_BUFFER[7];

//State machine
always @(posedge clk)
begin
    if(~rstn)
        begin
            state <= 0;
            TX_BUFFER <= 0;
            data_out <= 0;
            tx_complete <= 0;
            spi_tick_prev <= 0;
            tx_count <= 0;
            sck <= 0;
            cntr_enable <= 0;
        end
    else
        case(state)
            2'b00: //Idle state
                begin
                    tx_complete <= 0;
                    spi_tick_prev <= 0;
                    tx_count <= 0;
                    busy <= 0;
                    if(start_tx)
                        begin
                            state <= 2'b01;
                            busy <= 1;

```

```

        TX_BUFFER <= data_in;
        cntr_enable <= 1;
        RX_BUFFER <= 0;

    end

    end

    2'b01: //Transmit state
    begin
        //Shift data out falling edge
        if((spi_tick_prev) && (~spi_tick))
            begin
                TX_BUFFER <= {TX_BUFFER[6:0],1'b0};
                sck <= 0;
                tx_count <= tx_count + 1;
            end

        //Sample data on rising edge
        if((~spi_tick_prev) && (spi_tick) && (tx_count != 4'd8))
            begin
                RX_BUFFER <= {RX_BUFFER[6:0],MISO};
                sck <= 1;
            end

        //Update tick
        spi_tick_prev <= spi_tick;
        //Terminate transfer
        if(tx_count == 4'd8)
            begin
                state <= 2'b10;
                sck <= 0;
            end

    end

    2'b10: //TX done state
    begin
        tx_complete <= 1;
        state <= 2'b00;
        data_out <= RX_BUFFER;
        TX_BUFFER <= 0;
        cntr_enable <= 0;

    end

    default:
        state <= 0;

    endcase

end

endmodule

```

## 7.2. Busz illesztő

```

`timescale 1ns / 1ps
//AXI Lite controller with 4 32 bit registers

module AXI_LITE_CNTRL(
    //General
    input wire S_AXI_ACLK, //Clock

```

```

        input wire  S_AXI_ARESETN, //Active LOW reset
        //AXI4-Lite wires
        //Write address channel
        input wire [3:0] S_AXI_AWADDR, //Address, 4 bits
        input wire  S_AXI_AWVALID, //Write address valid
        output reg  S_AXI_AWREADY, //Write address ready
        //Write data channel
        input wire [31:0] S_AXI_WDATA, //Input data
        input wire [3:0] S_AXI_WSTRB, //Byte lane select
        input wire  S_AXI_WVALID, // Write address valid
        output reg  S_AXI_WREADY, //Ready for write
        //Write response channel
        output reg [1:0] S_AXI_BRESP, //Write response
        output reg  S_AXI_BVALID, // Write response
        input wire  S_AXI_BREADY, //Response ready
        //Read address channel
        input wire [3:0] S_AXI_ARADDR, //Read address
        input wire  S_AXI_ARVALID, //Read address valid
        output reg  S_AXI_ARREADY, //Read address ready
        //Read data channel
        output reg [31:0] S_AXI_RDATA, //Read data
        output reg [1:0] S_AXI_RRESP, //Read response
        output reg  S_AXI_RVALID, //Read valid
        input wire  S_AXI_RREADY, //Read ready
        //SPI signals
        output CS,
        output MOSI,
        output SCK,
        input MISO,
        //Interrupt
        output reg TX_DONE_IT
    );

    reg aw_enable = 0;

    //Write address latching
    reg [3:0] axi_waddr = 0;

    always @ (posedge S_AXI_ACLK)
    begin
        if(~S_AXI_ARESETN)
        begin
            axi_waddr <= 0;
        end
        else
        begin
            if(~S_AXI_AWREADY && S_AXI_AWVALID && S_AXI_WVALID && aw_enable)
                axi_waddr <= S_AXI_AWADDR;
            end
        end

        //Awready gen
        always @ (posedge S_AXI_ACLK)
        begin

```



```

    if (~S_AXI_ARESETN) //Reset signal
    begin
        S_AXI_AWREADY <= 0;
        aw_enable <= 1;
    end
    else
    begin
        if(~S_AXI_AWREADY && S_AXI_AWVALID && S_AXI_WVALID && aw_enable)
        begin
            S_AXI_AWREADY <= 1;
            aw_enable <= 0;
        end
        else if(S_AXI_BREADY && S_AXI_BVALID)
        begin
            S_AXI_AWREADY <= 0;
            aw_enable <= 1;
        end
        else
            S_AXI_AWREADY <= 0;
        end
    end
end

//Wready gen
always @ (posedge S_AXI_ACLK)
begin
    if (~S_AXI_ARESETN)
    begin
        S_AXI_WREADY <= 0;
    end
    else
    begin
        if(~S_AXI_WREADY && S_AXI_WVALID && S_AXI_AWVALID && aw_enable)
        begin
            S_AXI_WREADY <= 1;
        end
        else
        begin
            S_AXI_WREADY <= 0;
        end
    end
end

//Write logic
//---INPUT REGISTERS-----
reg [31:0] input_reg0 = 0;
reg [31:0] input_reg1 = 0;
reg [31:0] input_reg2 = 0;
reg [31:0] input_reg3 = 0;
//-----
//New TX trigger signal
reg NEW_TX = 0;
reg TX_DONE_CLEAR;
always @ (posedge S_AXI_ACLK)

```

```

begin
    if(~S_AXI_ARESETN)
        begin
            input_reg0 <= 0;
            input_reg1 <= 0;
            input_reg2 <= 0;
            input_reg3 <= 0;
        end
    else
        begin
            if(S_AXI_WREADY && S_AXI_WVALID && S_AXI_AWREADY && S_AXI_AWVALID)
                begin
                    case(axi_waddr[3:2])
                        2'h0:
                            begin
                                if(S_AXI_WSTRB[0] == 1)
                                    begin
                                        input_reg0[7:0] <= S_AXI_WDATA[7:0];
                                        NEW_TX <= 1;
                                    end
                                if(S_AXI_WSTRB[1] == 1)
                                    input_reg0[15:8] <= S_AXI_WDATA[15:8];
                                if(S_AXI_WSTRB[2] == 1)
                                    input_reg0[23:16] <= S_AXI_WDATA[23:16];
                                if(S_AXI_WSTRB[3] == 1)
                                    input_reg0[31:24] <= S_AXI_WDATA[31:24];
                                end
                            2'h1:
                                begin
                                    if(S_AXI_WSTRB[0] == 1)
                                        input_reg1[7:0] <= S_AXI_WDATA[7:0];
                                    if(S_AXI_WSTRB[1] == 1)
                                        input_reg1[15:8] <= S_AXI_WDATA[15:8];
                                    if(S_AXI_WSTRB[2] == 1)
                                        input_reg1[23:16] <= S_AXI_WDATA[23:16];
                                    if(S_AXI_WSTRB[3] == 1)
                                        input_reg1[31:24] <= S_AXI_WDATA[31:24];
                                    end
                                2'h2:
                                    begin
                                        if(S_AXI_WSTRB[0] == 1)
                                            begin
                                                input_reg2[7:0] <= S_AXI_WDATA[7:0];
                                                TX_DONE_CLEAR <= 1;
                                            end
                                        if(S_AXI_WSTRB[1] == 1)
                                            input_reg2[15:8] <= S_AXI_WDATA[15:8];
                                        if(S_AXI_WSTRB[2] == 1)
                                            input_reg2[23:16] <= S_AXI_WDATA[23:16];
                                        if(S_AXI_WSTRB[3] == 1)
                                            input_reg2[31:24] <= S_AXI_WDATA[31:24];
                                        end
                                    2'h3:
                                        begin

```

```

        if(S_AXI_WSTRB[0] == 1)
            input_reg3[7:0] <= S_AXI_WDATA[7:0];
        if(S_AXI_WSTRB[1] == 1)
            input_reg3[15:8] <= S_AXI_WDATA[15:8];
        if(S_AXI_WSTRB[2] == 1)
            input_reg3[23:16] <= S_AXI_WDATA[23:16];
        if(S_AXI_WSTRB[3] == 1)
            input_reg3[31:24] <= S_AXI_WDATA[31:24];
    end
    default:
        begin
        end
    endcase
end
else
    begin
        NEW_TX <= 0;
        TX_DONE_CLEAR <= 0;
    end
end
end

//Write response-----
always @ (posedge S_AXI_ACLK)
begin
    if(~S_AXI_ARESETN)
        begin
            S_AXI_BVALID <= 0;
            S_AXI_BRESP <= 0;
        end
    else
        begin
            if(S_AXI_AWREADY && S_AXI_AWVALID && ~S_AXI_BVALID && S_AXI_WREADY && S_AXI_WVALID)
                begin
                    S_AXI_BVALID <= 1;
                    S_AXI_BRESP <= 0;
                end
            else
                begin
                    if(S_AXI_BREADY && S_AXI_BVALID)
                        S_AXI_BVALID <= 0;
                    end
                end
        end
    end
end

//Implement axi_arready generation

reg [3:0] axi_araddr = 0;

always @ (posedge S_AXI_ACLK)
begin
    if(~S_AXI_ARESETN)
        begin
            S_AXI_ARREADY <= 0;

```

```

        axi_araddr <= 0;
    end
else
    begin
        if (~S_AXI_ARREADY && S_AXI_ARVALID)
            begin
                S_AXI_ARREADY <= 1;
                axi_araddr <= S_AXI_ARADDR;
            end
        else
            begin
                S_AXI_ARREADY <= 0;
            end
        end
    end
end

//Implement axi_arvalid generation
always @ (posedge S_AXI_ACLK)
begin
    if (~S_AXI_ARESETN)
        begin
            S_AXI_RVALID <= 0;
            S_AXI_RRESP <= 0;
        end
    else
        begin
            if (S_AXI_ARREADY && S_AXI_ARVALID && ~S_AXI_RVALID)
                begin
                    S_AXI_RVALID <= 1;
                    S_AXI_RRESP <= 0;
                end
            else if (S_AXI_RVALID && S_AXI_RREADY)
                begin
                    S_AXI_RVALID <= 0;
                end
        end
    end
end

//Read logic
//-----Output registers-----
wire [31:0] output_reg0;
wire [31:0] output_reg1;
wire [31:0] output_reg2;
wire [31:0] output_reg3;
//-----

always @ (posedge S_AXI_ACLK)
begin
    if (~S_AXI_ARESETN)
        begin
            S_AXI_RDATA <= 0;
        end
    else
        begin

```

```

        if(S_AXI_ARREADY && S_AXI_ARVALID && ~S_AXI_RVALID)
            begin
                case(axi_araddr[3:2])
                    2'h0 : S_AXI_RDATA <= output_reg0;
                    2'h1 : S_AXI_RDATA <= output_reg1;
                    2'h2 : S_AXI_RDATA <= output_reg2;
                    2'h3 : S_AXI_RDATA <= output_reg3;
                endcase
            end
        end

    end

end

//-----
//SPI controll logic

//Data register IN trigger
wire SPI_busy;
wire TX_START = ~SPI_busy & NEW_TX & (~CS);

//IT generation & TX_COMPLETE flag
reg TX_DONE_REG;
wire TX_DONE_SPI;
assign output_reg2[0] = TX_DONE_REG;
assign output_reg2[1] = input_reg2[1];

always @ (posedge S_AXI_ACLK)
begin
    if(~S_AXI_ARESETN)
        begin
            TX_DONE_REG <= 0;
            TX_DONE_IT <= 0;
        end
    else
        begin
            if(TX_DONE_SPI)
                begin
                    TX_DONE_REG <= 1;
                    if(input_reg2[1])
                        TX_DONE_IT <= 1;
                end
            else if(((TX_DONE_CLEAR) && (input_reg2[0])) || (TX_START))
                begin
                    TX_DONE_REG <= 0;
                    TX_DONE_IT <= 0;
                end
            else
                TX_DONE_IT <= 0;
        end
    end

end

//REG4 CS BUSY
assign CS = ~input_reg3[1];
assign output_reg3[1] = input_reg3[1];

```

```

assign output_reg3[0] = SPI_busy;

//SPI connection
SPI_MASTER SPI( .clk(S_AXI_ACLK),
                .rstn(S_AXI_ARESETN),
                .clk_div(input_reg1[2:0]),
                .data_in(input_reg0[7:0]),
                .data_out(output_reg0[7:0]),
                .busy(SPI_busy),
                .tx_complete(TX_DONE_SPI),
                .start_tx(TX_START),
                .MOSI(MOSI),
                .MISO(MISO),
                .sck(SCK)
                );

//Etc
assign output_reg0[31:8] = 24'h0;
assign output_reg1 = {29'h0,input_reg1[2:0]};
assign output_reg2[31:2] = 30'h0;
assign output_reg3[31:2] = 30'h0;

endmodule

```

### 7.3. Szimulációs kód

```

`timescale 1ns / 1ps

module EEPROM_SPI_TB(
);

//Clock generation
reg S_AXI_ACLK = 0;

always
begin
    #5 S_AXI_ACLK <= ~S_AXI_ACLK;
end

//Reset signal generation
reg S_AXI_ARESETN = 0; //Initial state
always
begin
    #20;
    @(posedge S_AXI_ACLK);
    #1 S_AXI_ARESETN <= 1;
end

//Connections
//AXI4-Lite wires
//Write address channel

```

```

reg [3:0] S_AXI_AWADDR = 0;
reg S_AXI_AWVALID = 0;
wire S_AXI_AWREADY;
//Write data channel
reg [31:0] S_AXI_WDATA = 0;
reg [3:0] S_AXI_WSTRB = 0;
reg S_AXI_WVALID = 0;
wire S_AXI_WREADY;
//Write response channel
wire [1:0] S_AXI_BRESP;
wire S_AXI_BVALID;
reg S_AXI_BREADY = 0;
//Read address channel
reg [3:0] S_AXI_ARADDR = 0;
reg S_AXI_ARVALID = 0;
wire S_AXI_ARREADY;
//Read data channel
wire [31:0] S_AXI_RDATA;
wire [1:0] S_AXI_RRESP;
wire S_AXI_RVALID;
reg S_AXI_RREADY = 0;
//SPI IO
wire MOSI;
wire MISO;
wire SCK;
wire TX_DONE_IT;
wire CS;

//AXI Lite transaction tasks
//Write transaction
task axi_write;
    input [3:0] address;
    input [31:0] data;
    begin
        @(posedge S_AXI_ACLK);
        #1;
        S_AXI_AWADDR <= address;
        S_AXI_AWVALID <= 1;
        S_AXI_WDATA <= data;
        S_AXI_WSTRB <= 4'b1111;
        S_AXI_WVALID <= 1;
        S_AXI_BREADY <= 1;
        wait(S_AXI_AWREADY);
        wait(S_AXI_WREADY);
        @(posedge S_AXI_ACLK);
        S_AXI_WVALID <= 0;
        S_AXI_AWVALID <= 0;
        wait(S_AXI_BVALID);
        @(posedge S_AXI_ACLK);
        S_AXI_BREADY <= 0;
    end
endtask

//Read transaction

```

```

task axi_read;
    input [3:0] address;
    output reg [31:0] data;
    begin
        @(posedge S_AXI_ACLK);
        #1;
        S_AXI_ARADDR <= address;
        S_AXI_ARVALID <= 1;
        S_AXI_RREADY <= 1;
        wait(S_AXI_ARREADY);
        @(posedge S_AXI_ACLK);
        S_AXI_ARVALID <= 0;
        wait(S_AXI_RVALID);
        @(posedge S_AXI_ACLK);
        data <= S_AXI_RDATA;
        #1;
        S_AXI_RREADY <= 0;
    end
endtask

//Read buffer
reg [31:0] read_data = 0;

//Test cases
always
begin
    #40;
    //Set up SPI
    axi_write(4,3); //CLK div /16
    axi_write(8,8'b10); //IT enable
    //Pull CS low
    axi_write(12,8'b10);
    //WRITE LATCH ON
    axi_write(0,8'b00000110);
    //CS high
    wait(TX_DONE_IT);
    axi_write(12,8'b00);
    #50;
    //CS low
    axi_write(12,8'b10);
    //Write data
    axi_write(0,8'b00000010); wait(TX_DONE_IT); //Write instruction
    axi_write(0,8'b00000010); wait(TX_DONE_IT); //ADDRESS
    axi_write(0,8'hAA); wait(TX_DONE_IT); //DATA
    axi_write(0,8'hBB); wait(TX_DONE_IT);
    axi_write(0,8'hC5); wait(TX_DONE_IT);
    axi_write(12,8'b00); //CS high
    #500;
    //Read data
    axi_write(12,8'b10); //CS low
    axi_write(0,8'b00000011); wait(TX_DONE_IT); //Read instruction
    axi_write(0,8'b00000010); wait(TX_DONE_IT); //ADDRESS
    axi_write(0,8'b0); wait(TX_DONE_IT); //Start transfer
    axi_read(0,read_data); //READ

```



```

    axi_write(0,8'b0); wait(TX_DONE_IT); //Start transfer
    axi_read(0,read_data); //READ
    axi_write(0,8'b0); wait(TX_DONE_IT); //Start transfer
    axi_read(0,read_data); //READ
    axi_write(12,8'b00); //CS high
    #5000000;

end

//Connect to UUT
AXI_LITE_CNTRL UUT( .S_AXI_ACLK(S_AXI_ACLK),
                    .S_AXI_ARESETN(S_AXI_ARESETN),
                    .S_AXI_AWADDR(S_AXI_AWADDR),
                    .S_AXI_AWVALID(S_AXI_AWVALID),
                    .S_AXI_AWREADY(S_AXI_AWREADY),
                    .S_AXI_WDATA(S_AXI_WDATA),
                    .S_AXI_WSTRB(S_AXI_WSTRB),
                    .S_AXI_WVALID(S_AXI_WVALID),
                    .S_AXI_WREADY(S_AXI_WREADY),
                    .S_AXI_BRESP(S_AXI_BRESP),
                    .S_AXI_BVALID(S_AXI_BVALID),
                    .S_AXI_BREADY(S_AXI_BREADY),
                    .S_AXI_ARADDR(S_AXI_ARADDR),
                    .S_AXI_ARVALID(S_AXI_ARVALID),
                    .S_AXI_ARREADY(S_AXI_ARREADY),
                    .S_AXI_RDATA(S_AXI_RDATA),
                    .S_AXI_RRESP(S_AXI_RRESP),
                    .S_AXI_RVALID(S_AXI_RVALID),
                    .S_AXI_RREADY(S_AXI_RREADY),
                    .MOSI(MOSI),
                    .MISO(MISO),
                    .SCK(SCK),
                    .TX_DONE_IT(TX_DONE_IT),
                    .CS(CS)
                    );

//EEPROM
M25AA010A EEPROM( .SI(MOSI),
                   .SCK(SCK),
                   .CS_N(CS),
                   .WP_N(1'b1),
                   .HOLD_N(1'b1),
                   .RESET(~S_AXI_ARESETN),
                   .SO(MISO));

endmodule

```

## Hivatkozások

- [1] *SPI leírása* [https://en.wikipedia.org/wiki/Serial\\_Peripheral\\_Interface\\_Bus](https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus)  
2018. május 18.
- [2] *AXI Lite leírása és időzítési diagramja*  
[https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_lite\\_ipif/v3\\_0/pg155-axi-lite-ipif.pdf](https://www.xilinx.com/support/documentation/ip_documentation/axi_lite_ipif/v3_0/pg155-axi-lite-ipif.pdf)  
2018. május 18.
- [3] *AXI Lite példa*  
[https://www.so-logic.net/documents/knowledge/tutorial/Basic\\_FPGA\\_Tutorial\\_Verilog/sec\\_designing\\_with\\_ips.html](https://www.so-logic.net/documents/knowledge/tutorial/Basic_FPGA_Tutorial_Verilog/sec_designing_with_ips.html)  
2018. május 18.
- [4] *EEPROM leírása*  
<http://ww1.microchip.com/downloads/en/DeviceDoc/21832H.pdf>  
2018. május 18.