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

Rendszerarchitektúrák házi feladat

Dokumentáció

Készítette
Horváth Dániel
Kalocsai Kristóf
Uzseka Dániel

Konzulens
Fehér Béla

2017. május 17.

Tartalomjegyzék

1. fejezet

Bevezetés

Féléves munkánk során egy perifériaillesztő modult valósítottunk meg System Verilog nyelven. Az illesztő az ARM™Ltd. AMBA™-APB rendszerbusza, és az I2C busz között teremt kapcsolatot. Ezen két buszt ismertetjük a továbbiakban.

1.1. AMBA-APB

Az Advanced Microcontroller Bus Architecture egy nyílt szabvány, amely chipen belüli kommunikációs összeköttetéseket definiál. Ennek a szabványnak része az Advanced Peripheral Bus (APB)[1] , amely alacsony sávszélességű, kis komplexitású, és minimális fogyasztású.

PCLK	Órajel, felfutó éle időzíti az összes átviteli ciklust.
PRESETn	Reset, aktív-alacsony.
PADDR	Címbusz, maximum 32 bit széles.
PSELx	Slave-select, minden egységhez tartozik egy. Kiválasztja az adott slave egységet.
PENABLE	Engedélyező jel, az átviteli ciklus második szakaszát jelzi.
PWRITE	Irányjelző, magas értéke írást, alacsony értéke olvasást jelent.
PWDATA	Adatbusz, maximum 32 bit széles, mindig a perifériabusz bridge hajtja.
PRDATA	Adatbusz, maximum 32 bit széles, a slave egység hajtja az olvasási ciklusban.

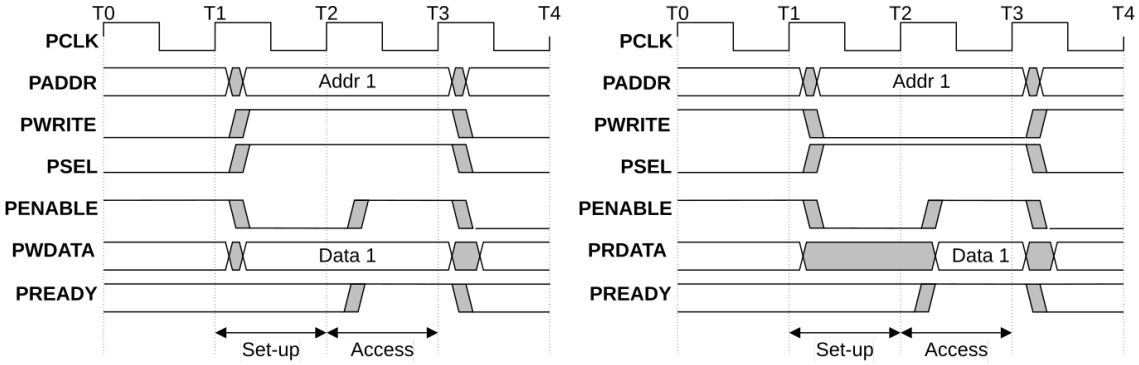
1.1. táblázat. Az APB busz jelei.

A busz jeleinek áttekintése után nézzünk egy írási ciklust a buszon. Minden átviteli ciklus két szakaszból áll, egy *setup* és egy *access* fázisból, ezen fázisok PCLK felfutó élére kezdődnek, ahogy az az 1.1 ábrán [1] megfigyelhető. A setup fázisban a híd eszköz (APB master) a PADDR buszra kiadja a címzett periféria címét, a PWRITE jelet az írásnak megfelelően magas értékűre

[1] Források:
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0367b/graphics/apb_write_transfer_no_wait_states.svg,
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0367b/graphics/apb_read_transfer_no_wait_states.svg

állítja. A slave-hez tartozó PSEL vonalat is magas értékre állítja be, továbbá a PWDATA buszra kapuzza az írni kívánt adatot. Ebben a fázisban PENABLE alacsony értékű, így a slave eszköznek lehetősége van felkészülni az átvitelre. PCLK következő felfutó élére a PENABLE vonalat logikai 1 értékűre állítja, és ezzel a buszciklus végéig bezárólag a slave eszköz mintát vesz a PWDATA buszról, amivel lezárul az átvitel, PSEL, PWRITE és PENABLE visszaállításra kerül a master által.

Az olvasás ciklus a vezérlőjeleket tekintve csak PWRITE értékében tér el. Ha az APB master olvasási ciklust kezdeményez, a PWRITE jelet logikai alacsony értékűre állítja a setup fázisban. Az access fázisban szintén kiadja a PENABLE jelet, mire a slave eszköz a megcímzett regiszterét a PRDATA buszra kapuzza. Az előzőekhez hasonlóan, a master vezérlőjeleket visszaveszi a ciklus befejeztével.



1.1. ábra. Egy írási és olvasási ciklus időzítési viszonyai az APB buszon.

1.2. I2C

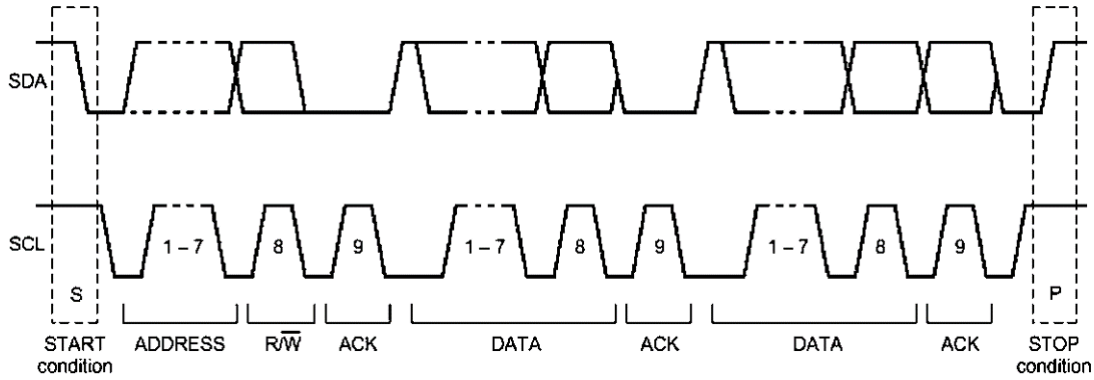
Az Inter-Integrated Circuit (vagy I2C, I^2C, esetleg IIC) egy multi-master, multi-slave, single-ended, félduplex soros kommunikációs busz.[2] Két, nyitott kollektoros (open-drain) vezetéket használ a kétirányú kommunikációhoz, ezek az *SDA* adatvonal és *SCL* órajelvezeték. Fizikai kialakításból adódóan a vezetékeket ellenállásokon keresztül magas logikai (3V3, 5V, 1V8 stb.) feszültségre kell felhúzni. A buszra csatlakozó eszközök a vezetékeket "kényszeríteni" tehát csak lefelé tudják. Ez teszi lehetővé a multi-master struktúrához szükséges kiválasztást és arbitrációt, továbbá mivel különálló kiválasztó jelek (slave-select) nem állnak rendelkezésre, így az üzeneteket címezni kell. Egy üzenet felépítése látható az 1.2 ábrán.[2]

Az *SCL* órajelet mindig a *master* szolgáltatja a buszra csatlakozó eszközöknek.

Minden üzenet a *START* feltétellel (az SCL magas értéke mellett SDA lefutó éle) kezdődik és a *STOP* feltétellel (az SCL magas értéke mellett SDA felfutó éle) végződik. Ezek speciális feltételek, az üzenet belsejében nem fordulhatnak elő, mivel SDA csak SCL alacsony értéke mellett változhat.

A startbitet követi egy 7 bites címmező, amely a címzettet azonosítja, és egy *R/W* bit, amely írás esetén alacsony, olvasás esetén magas értékű. Ezután a küldő "elengedi" az adatbuszt, és a vevő, ha sikeres volt az átvitel, az adatbuszon a következő bit idejéig egy alacsony logikai értékű *ACK* nyugtázó jelet ad az SDA vonalra. Ha ez megtörtént, a master további 8 órajelciklus alatt egy byte-nyi adatot kapuz az adatbuszra. Újabb nyugtázás esetén a byte átvitele befejeződött, és sikeres. A masternek lehetősége van további byteokat küldeni, szintén

[2] Forrás:
https://i2.wp.com/maxembedded.files.wordpress.com/2014/02/data-transfer-timing-diagram.png



1.2. ábra. Az I2C busz időzítési diagrammja.

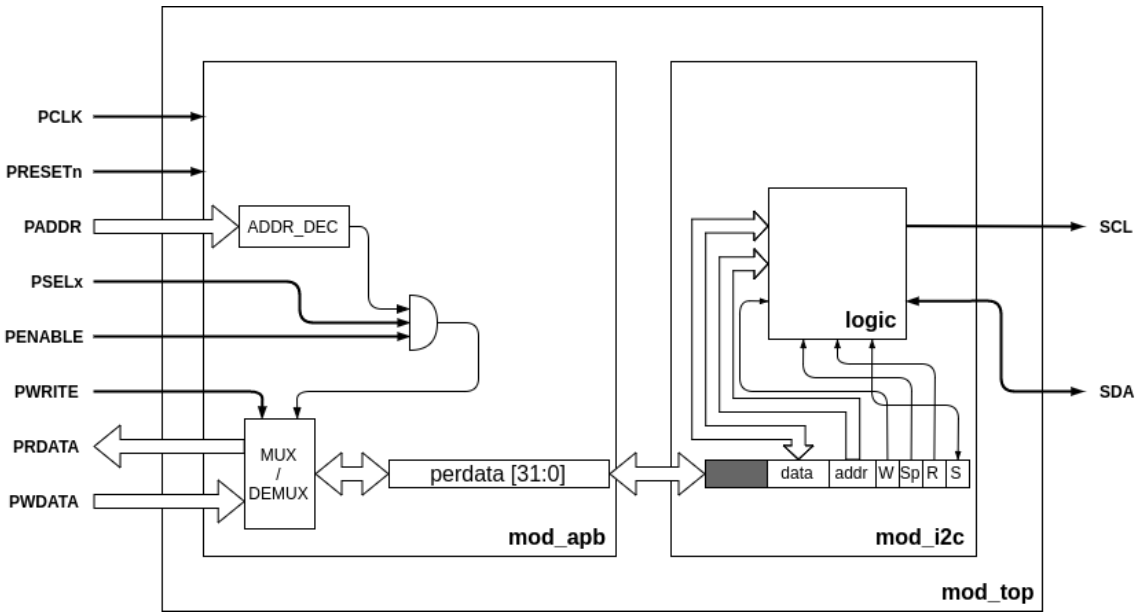
minden 8 bit adat után egy bit nyugtázással. Ha lezajlott a kívánt mennyiségű adat átvitele, a master kiadja a stopfeltételt, amely az üzenet végét jelenti.

2. fejezet

Áttekintés

2.1. A perifériaillesztő modul felépítése

Mint az a 2.1 ábrán is látszik, modulunk (*mod_top*) 2 almodult tartalmaz, a feladatkiírásnak megfelelően: egy *mod_apb* modulból, ami közvetlenül az APB buszra csatlakozik, és egy *mod_i2c* modulból, ami a soros kommunkáció vonalaival van összeköttetésben.



2.1. ábra. A modul magas szintű áttekintő ábrája.

Az APB modul a rendszerbusz vezérlőjeleit dekódolja, és továbbítja a szükséges adatokat az I2C modulnak. Az I2C modul a rendelkezésére bocsátott adatokból lefolytatja soros kommunikációt.

A továbbiakban tekintsük át részletesebben az almodulok felépítését. A modulokhoz, és a későbbiekben a tesztekhez kapcsolódó forráskódok a dokumentum végén, a függelékben találhatóak.

2.2. mod_apb

Ki és bemenetek

Mivel ez a modul közvetlenül az APB buszra csatlakozik, bemenetei megegyeznek a busz jeleivel, eltekintve a PREADY és PSLVERR jelektől, melyeket nem használunk. Továbbá tar-

talmaz két darab 32 bites ki illetve bemeneti portot, amely tartalmazza az I2C modul számára küldött, illetve attól fogadott összes releváns adatot. A pontos tartalmat lásd az I2C modul tárgyalásánál.

Reset

A modul szinkron resetet valósít meg, az órajel felfutó élére mintát vesz a PRESETn jelből, amelynek alacsony értéke mellett nullára állítja a belső állapotregiszterét, illetve az APB és I2C felé menő kimeneteit.

Állapotok

Ez az almodul 4 belső állapotot különböztet meg az APB vezérlőjelek alapján, ahol *X* az érdektelent (Don't Care), *1* a logikai magasat (illetve helyes címet), *0* pedig ennek ellenkezőjét jelöli. Az alábbi táblázat szemlélteti az állapotokat:

Állapot	PADDR	PSELx	PENABLE	PWRITE
IDLE	X	0	0	X
SETUP	X	1	0	X
READ	1	1	1	0
WRITE	1	1	1	1

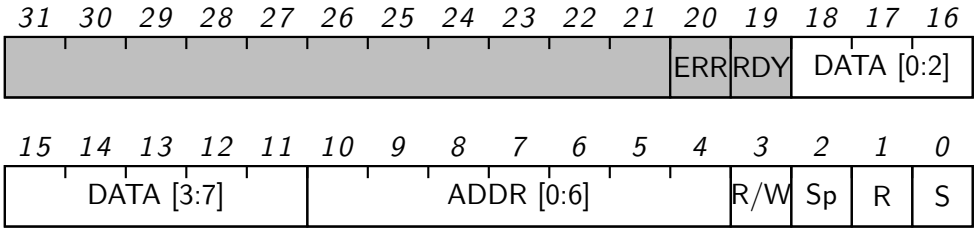
Logika

A vezérlőjelek dekódolása alapján, READ állapotban az I2C modul felől kapott 32 bit széles értéket (*in_perdata*) a PRDATA buszra kapuzza, WRITE állapotban a PWDATA busz tartalmát hozzárendeli az I2C felé tartó, *out_perdata* kimenetéhez. IDLE állapotban nagyimpedanciát kapcsol a PRDATA buszra, hogy a rendszer többi egységét ne zavarja. Ez a logika feltételes értékadással került implementálásra.

2.3. mod_i2c

Ki és bemenetek

Ez a modul közvetlenül az I2C buszra csatlakozik, így rendelkezik egy kétirányú, háromállapotú SDA porttal, mely a nyitott-kollektoros működést valósítja meg, illetve egy SCL órajel kimenettel. Továbbá csatlakozik a mod_apb modulhoz egy 32 bit széles regiszteren keresztül. Ez a regiszter tartalmaz minden információt a modul számára az I2C kommunikáció kezdeményezéséhez. A regiszter kiosztását és tartalmát lásd a 2.2 ábrán.



2.2. ábra. A kommunikációs regiszter kiosztása.

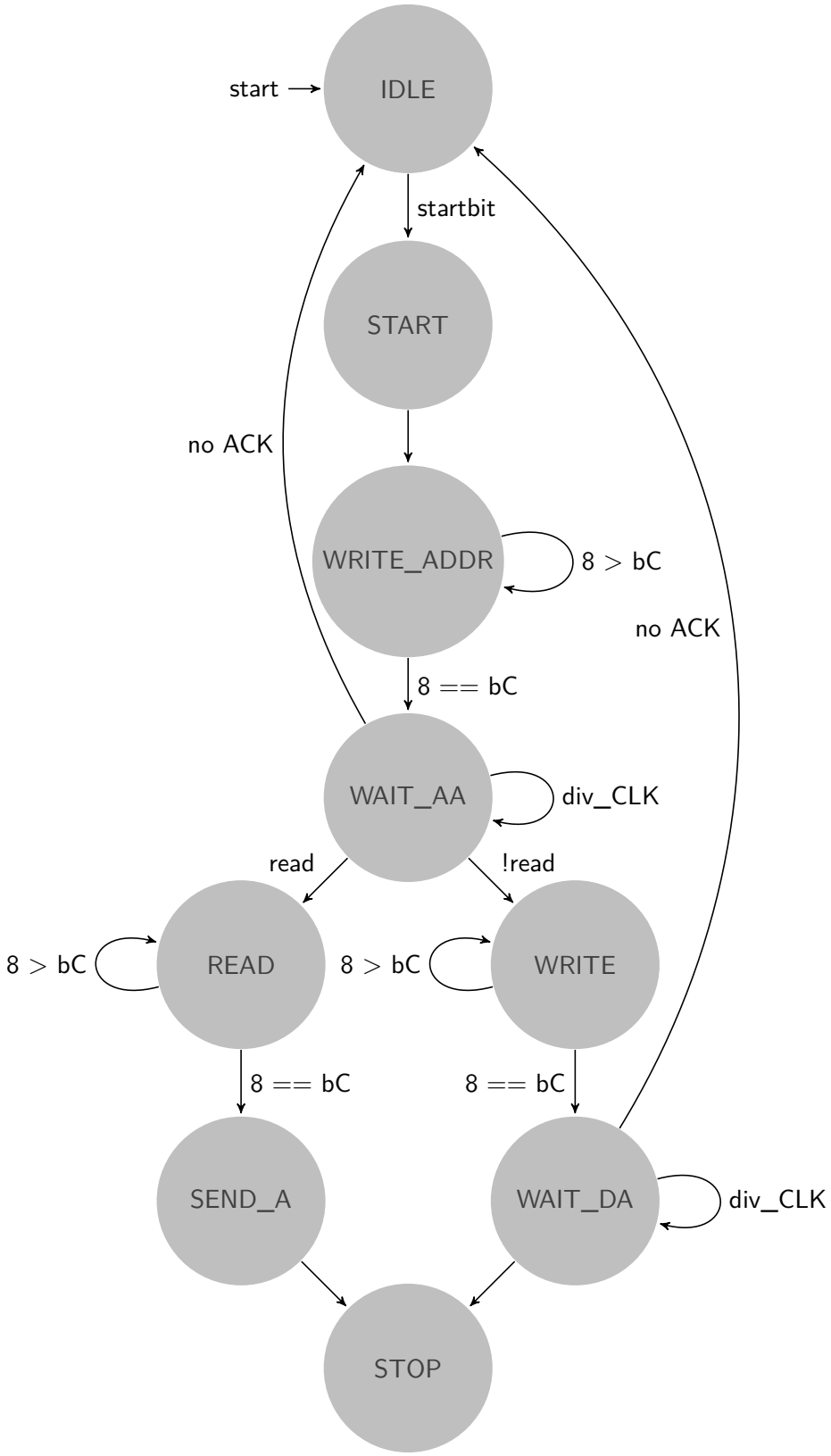
Reset

Ez az almodul is szinkron resetet valósít meg, az órajel felfutó élére vizsgálja a PRESETn vonalat és a regiszterének második bitjét (regData[1]). Ha ezek közül PRESETn-t 0-nak, vagy a bitet pedig 1-nek találja, elengedi az SDA és SCL vonalakat, és belső állapotváltozóit és számlálóit alaphelyzetbe állítja.

Állapotok és logika

A modul állapotátmenetei, és mechanizmusa látható a 2.3 ábrán. A 2.2 ábrán látható start bit (**S**) 1 értékbe állításakor elindul a soros kommunikáció. A sebességet tároló bit (**Sp**) 1-re állításával az I2C kommunikácó órajele 400 kHz-es, míg 0-ra állításával 100 kHz-es frekvenciával rendelkezik. Az írás/olvasás bit (**R/W**) 1 értéke esetén a perifériától való olvasás valósul meg, míg fordított esetben ennek az ellenkezője. Amint a kommunikáció elindul, a modul a készenlét jelzőbitjét (**RDY**) 0-ra állítja, ezzel jelezve, (ha a felhasználó lekérdezi, kiolvassa az APB buszon keresztül) hogy átvitel van folyamatban. Hasonlóképpen, ha hiba történik az átvitelben (nem érkezik ACK jel), azt az **ERR** bittel jelzi a modul.

Modulunk az I2C specifikáció csak egy részét teljesíti, ahogy sok más eszköz is. Csak master funkcióban működőképes, az SCL órajelet mindig ő szolgáltatja. A slave-k (esetlegesen masterek) számára fenntartott órajelnyújtási funkció nem került megvalósításra. Továbbá egy APB kérés hatására mindig 1 byteot olvas vagy ír, a parancsnak megfelelően.

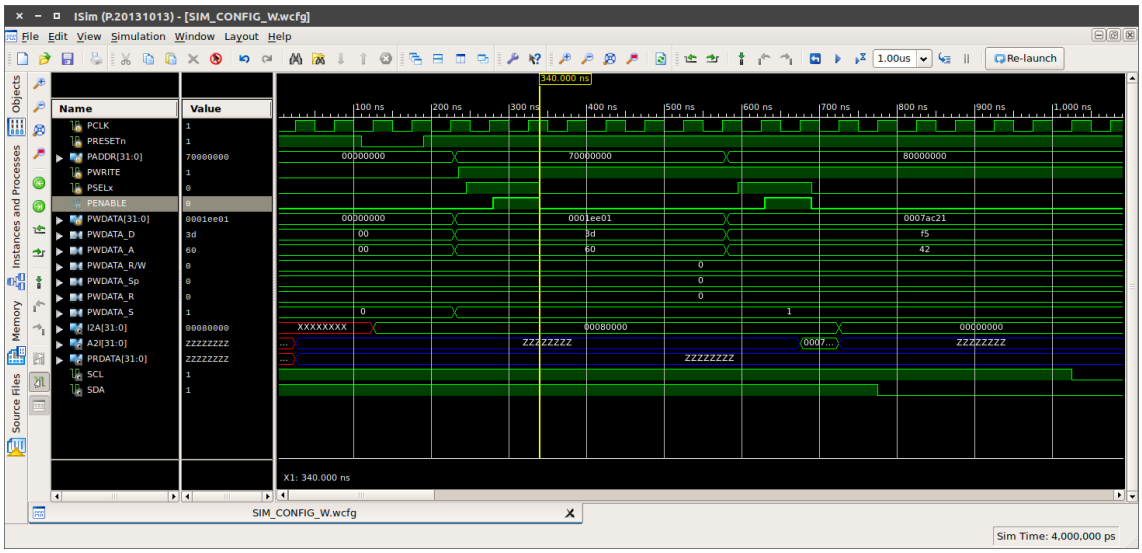


2.3. ábra. Az I2C modul állapotátmenetei

3. fejezet

Tesztelés

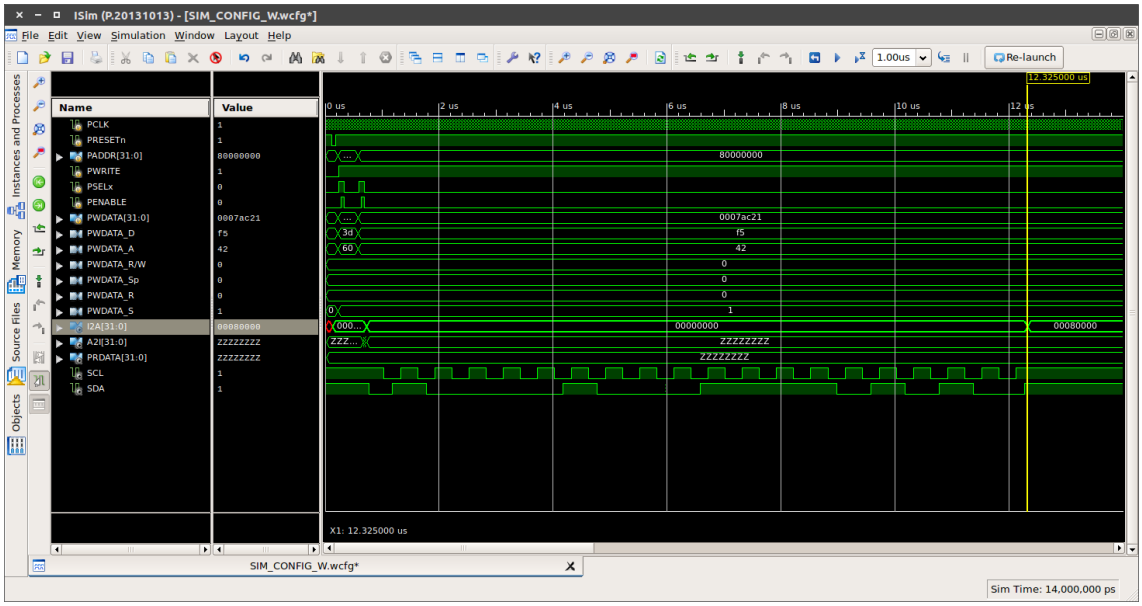
A 3.1 ábrán látható egy szimulációs hullámforma, amely a helyes APB vezérlőjel dekódolást hivatott demonstrálni. A kurzorral megjelölt hely előtti PCLK felfutóélre már minden szükséges vezérlő és engedélyezőjel ki volt adva az APB master részéről, az APB modul mégsem küld semmiféle adatot az I2C modul felé, mivel a PADDR buszon kiadott cím nem egyezik a perifériánk 0x80000000 címével. Viszont az ábrán 675ns - nál megfigyelhető egy helyes címre történő írás, így az A2I vezetéken megjelenik PWDATA tartalma, amit az I2C modul fel is dolgoz, és kezdi is a kommunikációt.



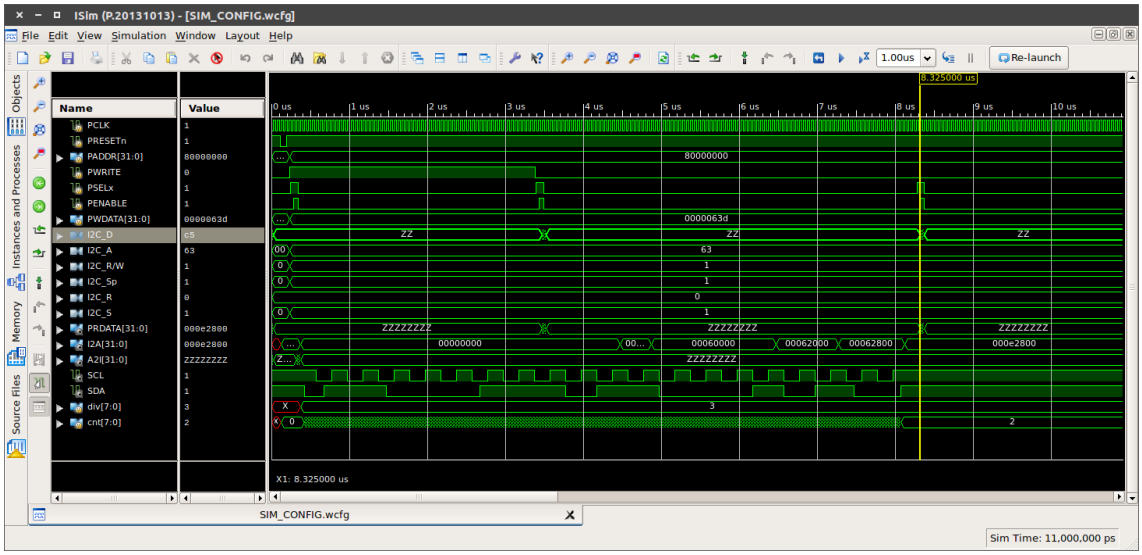
3.1. ábra. APB írás ciklus

A szimulációt tovább futtatva megfigyelhetünk egy rendben lezajló I2C írásműveletet a 3.2 ábrán. A szimulációhoz a slave felől érkező ACK jeleket kézzel, a szimulációs fájlokban szereplő módon adtuk ki, hogy igazolhassuk a modul megfelelő működését. A fájl természetesen megtalálható az F.5 függelékben. A hullámforma egy byte-nyi 0xF5 adat 0x42 címre történő írását ábrázolja.

A következő, 3.3 ábrán egy I2C olvasási ciklust szimuláltunk. Az ehhez tartozó szimulációs fájl szintén megtalálható a F.6 függelékben. Megfigyelhető egy lekérdezés az átvitel közben, de az olvasott adat ekkor **RDY** bit hiányában érvénytelen. Még egy APB olvasás látható a hullámformán, ezúttal már az átvtel lezajlása után, mikor is megfigyelhető az I2A, illetve a PRDATA buszokon az SDA vonalra kézzel felvezetett 0xC5 adat.



3.2. ábra. I2C írás ciklus



3.3. ábra. I2C olvasás ciklus

Ábrák jegyzéke

Táblázatok jegyzéke

Irodalomjegyzék

[1] ARM Limited: *AMBA 3 APB Protocol Specification*. ARM Limited, 2004. 09. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0024b/index.html.

[2] NXP Semiconductors: *I2C-bus specification and user manual*. NXP Semiconductors, 2014. 04. http://www.nxp.com/documents/user_manual/UM10204.pdf.

Függelék

F.1. mod_top.v

```
`timescale 1ns / 1ps

`include "macros.vh"


module mod_top
(
        input                                   PCLK,
        input                                   PRESETn,
        input   [`addrWidth-1:0]                PADDR,
        input                                   PSELx,
        input                                   PENABLE,
        input                                   PWRITE,
        output  [`dataWidth-1:0]                PRDATA,
        input   [`dataWidth-1:0]                PWDATA,
        inout                                   SDA,
        output                                  SCL
);

// apb_mod

// Outputs
        wire [`dataWidth-1:0] prdata;
        wire startbit;
        wire resetbit;
        wire it_enable;
        wire [`dataWidth-1:0] per_addr; //azert datawith nem pedig addrwith mert ez az i2c periferiaira
             vonatkozik
        wire [`dataWidth-1:0] per_data;

        wire [`dataWidth-1:0] A2I;
        wire [`dataWidth-1:0] I2A;

// Instantiate the module
apb_mod inst_APB (
    .clk(PCLK),
    .reset(PRESETn),
    .paddr(PADDR),
    .pwdata(PWDATA),
    .prdata(PRDATA),
    .pwrite(PWRITE),
    .psel(PSELx),
    .penable(PENABLE),
    .in_perdata(I2A),
    .out_perdata(A2I)
    );

// Instantiate the module
mod_I2C inst_I2C (
    .SDA(SDA),
    .SCL(SCL),
    .dataIn(A2I),
    .dataOut(I2A),
    .clk(PCLK),
    .rst(PRESETn)
    );

endmodule
```

F.2. macros.vh

```
`ifndef macros_vh
`define macros_vh

`define addrWidth 32
`define dataWidth 32

`endif
```

F.3. apb_mod.v

```
`timescale 1ns / 1ps
`include "macros.vh"

module apb_mod
(
   input clk,
    input reset,
    input [`addrWidth-1:0] paddr,
    input [`dataWidth-1:0] pwdata,
    output [`dataWidth-1:0] prdata,
    input pwrite,
    input psel,
    input penable,
         input [`dataWidth-1:0] in_perdata,
         output [`dataWidth-1:0] out_perdata
    );

reg [1:0] apb_status;

parameter IDLE      = 2'b00;
parameter SELECTED = 2'b01;
parameter READ      = 2'b10;
parameter WRITE     = 2'b11;

assign out_perdata = (WRITE == apb_status)? pwdata : 'bz;
assign prdata = (READ == apb_status)? in_perdata : 'bz;

always @ (posedge clk) begin     // clocked
        if(0 == reset) begin
                apb_status <= IDLE;      // reset state
        end
        else if (psel == 0) // not selected
                apb_status <= IDLE;
        else begin       // selected
                if (penable == 0) // but not enabled
                        apb_status <= SELECTED;
                else begin       // selected and enabled
                        if (32'h80000000 == paddr) begin // base address is 0x80000000
                                if (pwrite == 0)
                                        apb_status <= READ;      //read
                                else
                                        apb_status <= WRITE;     //write
                        end
                end
        end

end

endmodule
```

F.4. mod_I2C.v

```
`timescale 1ns / 1ps

module mod_I2C(
        inout                   SDA,             // I2C Data
        output                  SCL,             // I2C clock
        input [31:0]     dataIn,
                //0: start
                //1: reset
                //2: speed
                //3: read/write
                //4-10: address
                //11-18: data
        output [31:0]    dataOut,
        input                   clk, //16MHz clk
        input                           rst
    );


reg rSDA = 1;
reg rSCL = 1;

reg i2c_clk;

reg [31:0] regDataIn;
reg [31:0] regDataOut;


parameter [31:0] bSTART = 0;
parameter [31:0] bRESET = 1;
parameter [31:0] bSPEED = 2;
parameter [31:0] bRW    = 3;
parameter [31:0] bADDR  = 10;
parameter [31:0] bDATA  = 18;
parameter [31:0] bRDY   = 19;
parameter [31:0] bERR   = 20;


reg             [3:0]  states;
parameter [3:0] IDLE                            = 0;
```

```
parameter [3:0] START                   = 1;
parameter [3:0]         STOP                            = 2;
parameter [3:0]         WRITE_ADDR              = 3;
parameter [3:0] READ                            = 4;
parameter [3:0] WRITE                           = 5;
parameter [3:0] WAIT_ADDR_ACK   = 6;
parameter [3:0]         WAIT_DATA_ACK   = 7;
parameter [3:0] SEND_ACK                        = 8;

parameter SPEED_100kBPS         = 1'b0;
parameter SPEED_400kBPS         = 1'b1;

reg [3:0]       byteCounter = 0;

reg     [7:0]   div;
reg     [7:0] cnt;

reg read;


always @(posedge clk)
begin
        if (!rst | dataIn[1]) //reset
        begin
                states <= IDLE;
                rSDA <= 1;
                rSCL <= 1;
                cnt <= 0;
                regDataOut <= 32'b0;
                regDataOut[bRDY] <= 1;
        end
        else
        begin
        case (states)
                IDLE :
                begin
                        if (1 == dataIn[bSTART]) //start
                        begin
                                regDataIn <= dataIn;
                                regDataOut[bRDY] <= 0; //i2c is not ready for another communication

                                //set the speed of the communication
                                if (dataIn[bSPEED] == SPEED_100kBPS) //get the speed
                                begin
                                        div <= 5; //16Mbps to 100kbps (x2) - 80
                                end
                                else if (dataIn[bSPEED] == SPEED_400kBPS)
                                begin
                                        div <= 3; //16Mbps to 400kbps (x2) - 20
                                end

                                cnt <= 0;
                                states <= START;
                        end
                        else
                        begin
                                regDataOut[bRDY] <= 1; //ready
                        end
                end

                START :
                begin
                        rSDA <= 0; //pull down the wire

                        if (cnt == div) //divided clock -> toggle the scl
                        begin
                                cnt <= 0;

                                rSCL <= ~rSCL; //pull scl down

                                byteCounter <= 0;
                                read <= regDataIn[bRW];
                                states <= WRITE_ADDR; //go to the next state
                        end
                        else
                        begin
                                cnt <= cnt + 1;
                        end
                end

                WRITE_ADDR :
                begin
                        if (cnt == div) //divided clock -> toggle the scl
                        begin
                                cnt <= 0;

                                rSCL <= ~rSCL; //pull scl down
                        end
                        else
                        begin
                                cnt <= cnt + 1;
                        end

                        if(cnt == (div >> 1))
                        begin
                                if(0 == rSCL)
                                begin
                                        if(8 == byteCounter)
```

```
                        begin
                                states <= WAIT_ADDR_ACK;
                                rSDA <= 1;
                        end
                        else
                        begin
                                rSDA <= regDataIn[bADDR-byteCounter];
                                //from the 7th to the 1st bit
                                byteCounter <= byteCounter + 1;
                        end
                end
        end
end

WAIT_ADDR_ACK :
begin
        if(cnt == div)
        begin
                cnt <= 0;
                rSCL <= ~rSCL; //pull scl down

                if(1 == rSCL)
                begin
                        if(0 == SDA) //for tests, otherwise: 0
                        begin
                                byteCounter <= 0;
                                if(1 == read)
                                        states <= READ;
                                else
                                        states <= WRITE;
                        end
                        else
                        begin
                                //error - no Ack
                                states <= IDLE;
                                regDataOut[bERR] <= 1;
                        end
                end
        end
        else
        begin
                cnt <= cnt + 1;
        end
end

READ :
begin
        if (cnt == div)
        begin
                cnt <= 0;
                rSCL <= ~rSCL;
                if(8 == byteCounter)
                begin
                        states <= SEND_ACK;
                end
        end
        else
        begin
                cnt <= cnt + 1;
        end
        if(cnt == (div >> 1))
        begin
                if(1 == rSCL)
                begin
                        regDataOut[bDATA-byteCounter] <= SDA; //save the incoming data
                        byteCounter <= byteCounter + 1;
                end
        end
end

WRITE :
begin
        if (cnt == div)
        begin
                cnt <= 0;
                rSCL <= ~rSCL;
        end
        else
        begin
                cnt <= cnt + 1;
        end
        if(cnt == (div >> 1))
        begin
                if(0 == rSCL)
                begin
                        if(8 == byteCounter)
                        begin
                                states <= WAIT_DATA_ACK;
                                rSDA <= 1;
                        end
                        else
                        begin
                                rSDA <= regDataIn[bDATA-byteCounter]; //send the data
                                byteCounter <= byteCounter + 1;
                        end
                end
        end
end
```

```
                WAIT_DATA_ACK :
                begin
                        if(cnt == div)
                        begin
                                cnt <= 0;
                                rSCL <= ~rSCL; //pull scl down
                        end
                        else
                        begin
                                cnt <= cnt + 1;
                        end
                        if(cnt == (div))
                        begin
                                if(1 == rSCL)
                                begin
                                        if(0 == SDA) //for tests, otherwise: 0
                                        begin
                                                byteCounter <= 0;
                                                states <= STOP;
                                                rSDA <= 0; //if got ACK, pulling on SDA for release in STOP
                                        end
                                        else
                                        begin
                                                //error - no Ack
                                                states <= IDLE;
                                                regDataOut[bERR] <= 1;
                                        end
                                end
                        end
                end

                SEND_ACK :
                begin
                        if(cnt == div)
                        begin
                                cnt <= 0;
                                rSCL <= ~rSCL; //pull scl down
                        end
                        else
                        begin
                                cnt <= cnt + 1;
                        end
                        if(cnt == (div >> 1))
                        begin
                                if(0 == rSCL)
                                begin
                                        rSDA <= 0;
                                end
                                else
                                begin
                                        states <= STOP;
                                end
                        end
                end

                STOP :
                begin
                        if(cnt == div)
                        begin
                                cnt <= 0;
                                rSCL <= ~rSCL; //pull scl down
                        end
                        else
                        begin
                                cnt <= cnt + 1;
                        end
                        if(cnt == (div >> 1))
                        begin
                                if(1 == rSCL)
                                begin
                                        rSDA <= 1;
                                        states <= IDLE;
                                        regDataOut[bERR] <= 0;
                                end
                        end
                end
        endcase
        end
end


// Open Drain assignment
pullup(SDA); //for simulation only!
assign SDA = rSDA ? 1'bz : 1'b0;
assign SCL = rSCL;

assign dataOut = regDataOut;

endmodule
```

F.5. tb_top_W.v

```
`timescale 1ns / 1ps
```

```
module tb_top_W;

        // Inputs
        reg PCLK;
        reg PRESETn;
        reg [31:0] PADDR;
        reg PSELx;
        reg PENABLE;
        reg PWRITE;
        reg [31:0] PWDATA;

        // Outputs
        wire [31:0] PRDATA;
        wire SCL;

        // Bidirs
        wire SDA;

        // Instantiate the Unit Under Test (UUT)
        mod_top uut (
                .PCLK(PCLK),
                .PRESETn(PRESETn),
                .PADDR(PADDR),
                .PSELx(PSELx),
                .PENABLE(PENABLE),
                .PWRITE(PWRITE),
                .PRDATA(PRDATA),
                .PWDATA(PWDATA),
                .SDA(SDA),
                .SCL(SCL)
        );

        reg rSDA = 1'bz;

        initial begin
                // Initialize Inputs
                PCLK = 0;
                PRESETn = 1;
                PADDR = 0;
                PSELx = 0;
                PENABLE = 0;
                PWRITE = 0;
                PWDATA = 0;

                // Wait 100 ns for global reset to finish
                #100;

                // Add stimulus here

                //reset
                #10 PRESETn = 0;
                #80 PRESETn = 1;

                // helytelen cimre iras
                // DATA | ADDR |R/W|Sp|R|S|
                #40 PADDR = 32'h70000000;
                PWDATA[18:11] = 8'h3D; //data
                PWDATA[10:4] = 7'h60; // address
                PWDATA[3] = 0; // write
                PWDATA[2] = 0; // speed
                PWDATA[0] = 1; // start

                #5 PWRITE = 1; // AMBA write
                #10 PSELx = 1;  // selecting
                #25 //penable csak kovetkezo ciklusban!
                #10 PENABLE = 1;  //enabling

                //to idle
                #60 PENABLE = 0; PSELx = 0;

                #200;

                // DATA | ADDR |R/W|Sp|R|S|
                #40 PADDR = 32'h80000000;
                PWDATA[18:11] = 8'hF5; //data
                PWDATA[10:4] = 7'h42; // address
                PWDATA[3] = 0; // write
                PWDATA[2] = 0; // speed
                PWDATA[0] = 1; // start

                #5 PWRITE = 1; // AMBA write
                #10 PSELx = 1;  // selecting
                #25 //penable csak kovetkezo ciklusban!
                #10 PENABLE = 1;  //enabling

                //to idle
                #60 PENABLE = 0; PSELx = 0;

                #5288;
                // addr ACK (for speed=0)
                rSDA = 0;
                # 600;
                rSDA = 'bz;

                #4797;
                // data ACK (for speed=0)
                rSDA = 0;
                # 600;
```

```
                rSDA = 'bz;


        end

        assign SDA = rSDA;

        always begin
                #25 PCLK = ~PCLK;
        end

endmodule
```

F.6. tb_top_R.v

```
`timescale 1ns / 1ps

module tb_top_R;

        // Inputs
        reg PCLK;
        reg PRESETn;
        reg [31:0] PADDR;
        reg PSELx;
        reg PENABLE;
        reg PWRITE;
        reg [31:0] PWDATA;

        // Outputs
        wire [31:0] PRDATA;
        wire SCL;

        // Bidirs
        wire SDA;

        // Instantiate the Unit Under Test (UUT)
        mod_top uut (
                .PCLK(PCLK),
                .PRESETn(PRESETn),
                .PADDR(PADDR),
                .PSELx(PSELx),
                .PENABLE(PENABLE),
                .PWRITE(PWRITE),
                .PRDATA(PRDATA),
                .PWDATA(PWDATA),
                .SDA(SDA),
                .SCL(SCL)
        );

        reg rSDA = 1'bz;

        initial begin
                // Initialize Inputs
                PCLK = 0;
                PRESETn = 1;
                PADDR = 0;
                PSELx = 0;
                PENABLE = 0;
                PWRITE = 0;
                PWDATA = 0;

                // Wait 100 ns for global reset to finish
                #100;

                // Add stimulus here

                //reset
                #10 PRESETn = 0;
                #80 PRESETn = 1;

                // issuing i2c write command
                // DATA | ADDR |R/W|Sp|R|S|
                #40 PADDR = 32'h80000000; // peripheral address
                PWDATA[10:4] = 7'h63; // I2C address
                PWDATA[3] = 1; // read
                PWDATA[2] = 1; // speed
                PWDATA[0] = 1; // start

                #5 PWRITE = 1; // AMBA write
                #10 PSELx = 1;  // selecting
                #25 //penable csak kovetkezo ciklusban!
                #10 PENABLE = 1;  //enabling

                //to idle
                #60 PENABLE = 0; PSELx = 0;

                // polling peripheral
                # 3000;
                // DATA | ADDR |R/W|Sp|R|S|
                #40 PADDR = 32'h80000000; // peripheral address

                #5 PWRITE = 0; // AMBA read
                #10 PSELx = 1;  // selecting
                #25 //penable csak kovetkezo ciklusban!
                #10 PENABLE = 1;  //enabling
```

```
                //to idle
                #60 PENABLE = 0; PSELx = 0;

                // I2C is not ready (PRDATA[19] == 0), data is not reliable!


                // addr ACK (for speed=1)
                # 285;
                rSDA = 0;
                # 400;
                rSDA = 'bz;

                // slave data 8'hC5; MSB first
                rSDA = 'bz;  // 1
                # 400 rSDA = 'bz; // 1
                # 400 rSDA = 0; // 0
                # 400 rSDA = 0; // 0
                # 400 rSDA = 0; // 0
                # 400 rSDA = 'bz; // 1
                # 400 rSDA = 0; // 0
                # 400 rSDA = 'bz; // 1


                // polling peripheral
                # 1250;
                // DATA | ADDR |R/W|Sp|R|S|
                #40 PADDR = 32'h80000000; // peripheral address

                #5 PWRITE = 0; // AMBA read
                #10 PSELx = 1;  // selecting
                #25 //penable csak kovetkezo ciklusban!
                #10 PENABLE = 1;  //enabling

                //to idle
                #60 PENABLE = 0; PSELx = 0;

                // I2C was ready! (PRDATA[19] == 1)


        end

        assign SDA = rSDA;

        always begin
                #25 PCLK = ~PCLK;
        end

endmodule
```