

Rendszerarchitektúrák Házi Feladat

Wishbone – I2C kommunikáció megvalósítása

Készítette:

Moró Anna (KIHLI2)

Murai János (D0YRUM)

Konzulens:

Wacha Gábor

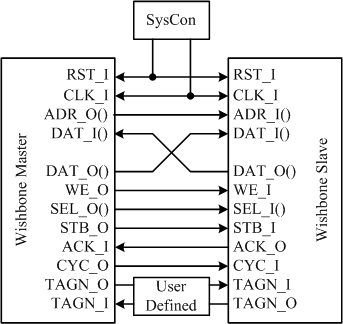
# Specifikáció

A Wishbone egy nyílt forrású, párhuzamos busz, mely ingyenes elérhető bárki számára. A feladatunk, hogy ehhez illesszük a rendkívül elterjed I2C buszt, egy egyszerű, kétirányú, kétvezetékes buszrendszert IC-k közötti vezérlésre, amely soros kommunikációt biztosít a hardware egységek között.

## Wishbone

A Wishbone buszból elérhető 8, 16, 32, 64 bites változat is. Mi ebből a 32 bites buszszélességet választottuk.

A következő jelek érhetőek el a buszon:

* RST\_I: A buszt ezen a jelen keresztül lehet alapállapotba vinni.
* CLK\_I: Rendszerórajel
* ADR\_O(): 32 bites címvezeték
* DAT\_I(): 32 bites adatvezeték bemenet, az adatok fogadására
* DAT\_O(): 32 bites adatvezeték kimenet, az adatok küldésére
* WE\_O/WE\_I: A master itt adja meg a slave felé, hogy az éppen futó buszciklus az READ, vagy WRITE ciklus.
* SEL\_O()/SEL\_I(): A slave kiválasztó jel
* STB\_O/STB\_I: A helyes adattranszfert jelzi a master a slavenek.
* ACK\_I/ACK\_O: A helyes adattranszfert jelzi a slave a masternek.
* CYC\_O/CYC\_I: A master jelzi a slave felé, hogy érvényes buszciklus van érvényben.
* Továbbá van 2 további jel, amiket a felhasználó tud definiálni (TAGN\_I/TAGN\_O). Ezeket mi nem használtuk a fejlesztés során.

## I2C

## Az I2C működéséhez két darab kétirányú buszvezeték szükséges, egy soros adatvonal (SDA) és egy soros órajel (SCL). Az adatforgalom soros, 8 bites rendszerű, melynek sebessége normál üzemmódban 100 kbit/s, gyors üzemmódban 400kbit/s.

Az I2C buszon az adatforgalom kezdetét, végét, illetve az adatáramlás irányának megváltozását külön jelekkel jelzik. Az egyik ilyen eset egy magas-alacsony átmenet az SDA vezetéken, miközben az SCL magas szintű. Ez a szituáció egy START feltételt jelez.

Az SCL magas szintje melletti alacsony-magas átmenet az SDA vezetéken egy STOP feltételt definiál.

A START és STOP feltételeket mindig a master generálja. A busz a START feltétel után foglaltnak tekinthető és később ismét szabadnak tekinthető egy STOP jel után.

Az SDA vezetéken minden byte nyolc bites. Az egy átvitel alatt átvihető byte-ok száma korlátlan. Az adat átvitele a legnagyobb helyiértékű bit (MSB) átvitelével kezdődik.

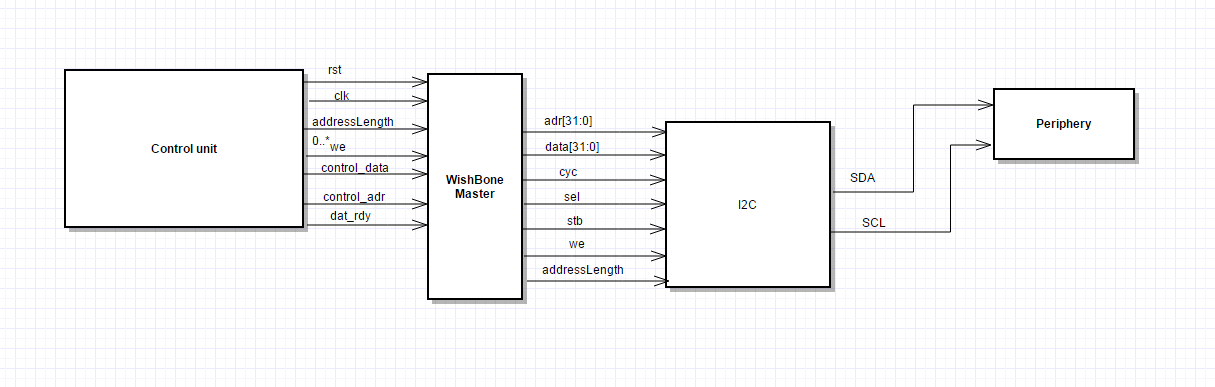
Az átvitel során minden byte-ot egy nyugtázás bit követ. A nyugtázással kapcsolatos órajel impulzust a master generálja. A küldő szabaddá teszi az SDA vezetéket (magas állapottal) a nyugtázás órajel impulzusa alatt. Amennyiben a fogadó elfogadja az adott byte-ot, akkor fel kell húznia az SDA vezetéket a nyugtázás órajelimpulzusa közben úgy, hogy az stabilan magasan maradjon az órajel magas periódusa alatt.

Az I2C buszrendszer kétfajta címzési módot ismer: a 7 bites és a 10 bites címzést. 7 bites formátum esetén a START feltétel után először a megcímzendő slave eszköz címe kerül elküldésre. Ez a cím 7 bit hosszú, kiegészítve egy nyolcadik, adatirányt jelző bittel (R/W). 10 bites címzés esetén első byte-ként egy a 7 bites címzésben nem felhasználható címet, az 1111 0xx címet adja ki a master első byte-ként. Az xx a 10 bites cím felső két bitjét jelenti. Az alsó nyolc bitet a következő byte-ban adja meg a master.

Továbbá szükséges egy busz illesztő áramkör. Ennek legegyszerűbb megvalósítása egy shift regiszterrel történhet.

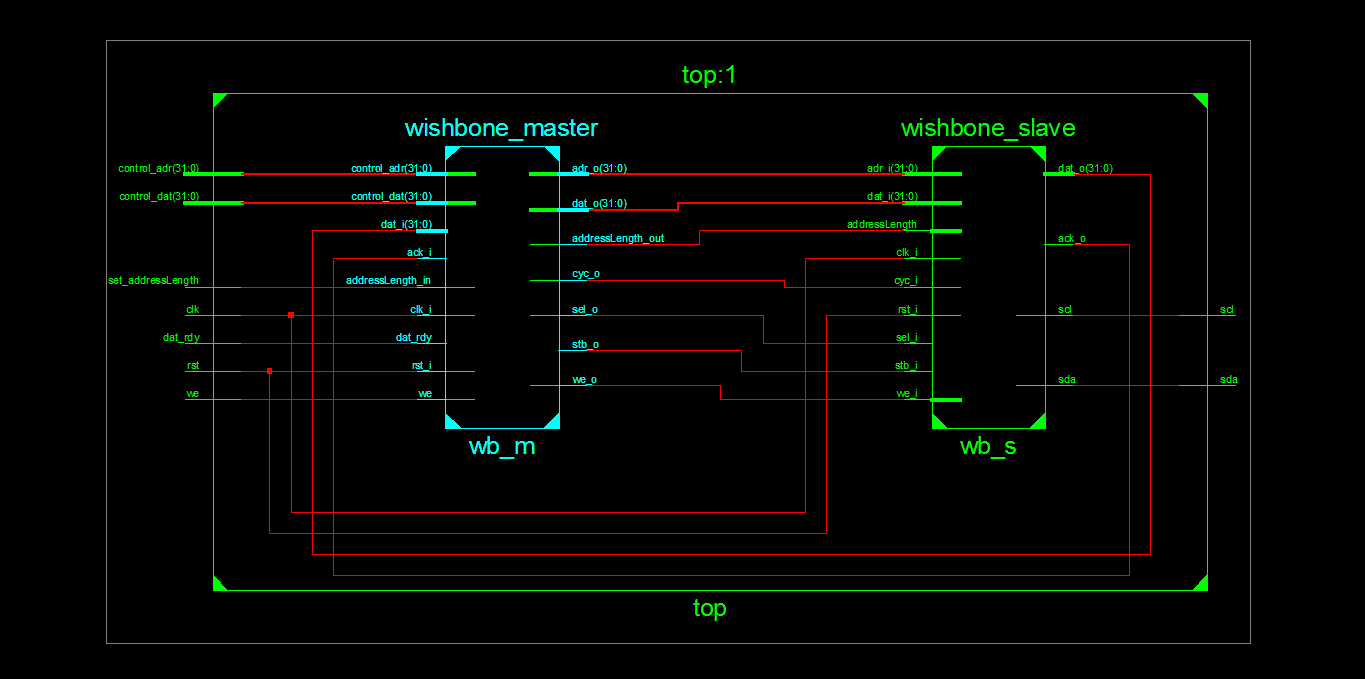
# Schematic

A tervezett blokkvázlat az alábbi ábrán látható:



1. ábra: A tervezett blokkvázlat

A megvalósítás után a Xilinx RTL Schematic funkciója segítségével is legeneráltunk egy blokkvázlatot:



. ábra: A generált blokkvázlat

Az ábrákból jól látható, hogy a tervezett és a kivitelezett blokkvázlat megegyezik.

# A forráskód

## A topmodul

`timescale 1ns / 1ps  
  
**module** top(  
        **input** rst,  
        **input** clk,  
        **input** [31:0] control\_dat,  
        **input** [31:0] control\_adr,  
        **input** dat\_rdy,  
        **input** set\_addressLength,  
        **input** we,  
          
        **inout** sda,  
        **output** scl  
    );  
  
**wire** [31:0]wb\_m2s\_dat;  
**wire** [31:0]wb\_m2s\_adr;  
**wire** wb\_m2s\_we;  
**wire** wb\_m2s\_sel;  
**wire** wb\_m2s\_stb;  
**wire** wb\_m2s\_cyc;  
  
**wire** wb\_s2m\_dat;  
**wire** wb\_s2m\_ack;  
**wire** addressLength;                     //User defined  
  
wishbone\_master wb\_m(  
        .rst\_i(rst),  
        .clk\_i(clk),  
        .dat\_i(wb\_s2m\_dat),  
        .ack\_i(wb\_s2m\_ack),  
          
        .adr\_o(wb\_m2s\_adr),  
        .dat\_o(wb\_m2s\_dat),  
        .we\_o(wb\_m2s\_we),  
        .sel\_o(wb\_m2s\_sel),  
        .stb\_o(wb\_m2s\_stb),  
        .cyc\_o(wb\_m2s\_cyc),  
        .addressLength\_out(addressLength),  
          
        //master controlling signals  
        .control\_dat(control\_dat),  
        .control\_adr(control\_adr),  
        .dat\_rdy(dat\_rdy),  
        .addressLength\_in(set\_addressLength),  
        .we(we)  
);  
  
  
wishbone\_slave #(  
        .slave\_addr(32'h1000\_0000))   
        wb\_s(  
        .rst\_i(rst),  
        .clk\_i(clk),  
        .dat\_i(wb\_m2s\_dat),  
        .adr\_i(wb\_m2s\_adr),  
        .we\_i(wb\_m2s\_we),  
        .sel\_i(wb\_m2s\_sel),  
        .stb\_i(wb\_m2s\_stb),  
        .cyc\_i(wb\_m2s\_cyc),  
        .addressLength(addressLength),  
          
        .dat\_o(wb\_s2m\_dat),  
        .ack\_o(wb\_s2m\_ack),  
          
        // Periphery signals  
        .sda(sda),  
        .scl(scl)  
);  
          
  
**endmodule**

## A WishBone master

`timescale 1ns / 1ps  
  
**module** wishbone\_master(  
        //wb signals  
        **input** rst\_i,  
        **input** clk\_i,  
        **input** [31:0] dat\_i,  
        **input** ack\_i,  
          
        **output** [31:0] adr\_o,  
        **output** [31:0] dat\_o,  
        **output** we\_o,  
        **output** sel\_o,  
        **output** stb\_o,  
        **output** cyc\_o,  
        **output** addressLength\_out,  
          
        //master controlling signals  
        **input** [31:0] control\_dat,  
        **input** [31:0] control\_adr,  
        **input** dat\_rdy,  
        **input** addressLength\_in,  
        **input** we  
    );  
           
**reg** [31:0]reg\_adr\_o;  
**reg** [31:0]reg\_dat\_o;  
**reg** reg\_we\_o;      
**reg** reg\_sel\_o;  
**reg** reg\_sys\_o;  
**reg** reg\_cyc\_o;  
**reg** addressLength;  
  
**always** @(**posedge** clk\_i)  
**begin**  
        **if**(rst\_i || ack\_i)  
        **begin**  
                reg\_adr\_o <= 0;  
                reg\_dat\_o <= 0;  
                reg\_we\_o <= 0;  
                reg\_sel\_o <= 0;  
                reg\_sys\_o <= 0;  
                reg\_cyc\_o <= 0;  
                addressLength <= 0;  
        **end**  
        **else** **if**(dat\_rdy)  
        **begin**  
                reg\_adr\_o <= control\_adr;  
                reg\_dat\_o <= control\_dat;  
                reg\_we\_o <= we;                                 
                reg\_sel\_o <= 1;  
                reg\_cyc\_o <= 1;  
                addressLength <= addressLength\_in;  
        **end**  
**end**  
  
**assign** adr\_o = reg\_adr\_o;  
**assign** dat\_o = reg\_dat\_o;  
**assign** we\_o = reg\_we\_o;    
**assign** sel\_o = reg\_sel\_o;  
**assign** sys\_o = reg\_sys\_o;  
**assign** cyc\_o = reg\_cyc\_o;  
**assign** addressLength\_out = addressLength;  
                  
  
**endmodule**

## Az I2C

`timescale 1ns / 1ps  
  
**module** wishbone\_slave #(  
        **parameter** slave\_addr = 32'h00000000)  
        (  
        **input** rst\_i,  
        **input** clk\_i,  
        **input** [31:0] dat\_i,  
        **input** [31:0] adr\_i,  
        **input** we\_i,                                            // 0-W, 1-R  
        **input** sel\_i,  
        **input** stb\_i,  
        **input** cyc\_i,  
        **input** addressLength,   // User defined. If (addressLength == 1) -> 10bbit  
          
        **output** [31:0] dat\_o,  
        **output** ack\_o,     
          
        //interface to the periphery  
        **output**         scl,  
        **inout**          sda  
          
        );  
          
**reg** selected;   // true if the slave\_addr and the adr\_i are equal  
**reg** ack;                        // acknowledge signal to master  
**wire** sda\_oe;    // true in write cycle, false in read cycle  
**reg** scl\_reg;    // Contains the value of the SCL  
**wire** scl\_en;    // Used for setting the SCL's frequency  
**wire** scl\_oe;    // indicates an ongoing cycle  
**wire** scl\_low;   // The SDA can change if it's true.  
**wire** scl\_high;  // If SCL is high, START or STOP condition can be indicated.  
  
// Set bitrate  
**parameter** BITRATE = 10;  
  
//Selection detection  
**always** @(**posedge** clk\_i)  
**begin**  
        **if**(rst\_i) selected <= 0;  
        **else** **if**(adr\_i == slave\_addr) selected <= 1;  
        **else** selected <= 0;  
**end**  
  
// Start cycle   
**reg** cyc\_old;  
**reg** start;  
**always** @(**posedge** clk\_i)  
**begin**  
        **if**(rst\_i)  
        **begin**  
                cyc\_old <= 0;  
                start <= 0;  
        **end**  
        **else**  
        **begin**  
                **if**((cyc\_old == 0) && (cyc\_i == 1)) start <= 1;  
                cyc\_old <= cyc\_i;  
        **end**  
**end**  
  
// Set BitRate  
**reg** [7:0] clk\_counter;  
**always** @(**posedge** clk\_i)  
 **begin**  
        **if** (rst\_i || (clk\_counter == BITRATE) || ((cyc\_old == 0) && (cyc\_i == 1)))  
        **begin**  
                clk\_counter <= 0;  
        **end**  
        **else**   
        **begin**  
                clk\_counter <= clk\_counter + 1;  
        **end**  
**end**  
  
**assign** scl\_en = (clk\_counter == BITRATE);    
  
// Create scl signal  
**always** @(**posedge** clk\_i) **begin**  
        **if**(rst\_i) **begin**  
                scl\_reg <= 0;  
        **end**  
        **else** **if**(scl\_en) **begin**  
                scl\_reg <= ~scl\_reg;  
        **end**  
**end**  
                  
// Fill up the shift register  
// 10bit address is separated  
**reg** [55:0] sda\_reg;  
**reg** start\_condition;  
**reg** stop\_condition;  
**reg**[5:0] bit\_counter;  
**always** @(**posedge** clk\_i) **begin**  
        **if**(rst\_i)**begin**  
                sda\_reg <= {56{1'b1}};  
                bit\_counter <= 0;  
                start\_condition <= 0;  
  
        **end**  
        **else** **if** ((cyc\_old == 0) && (cyc\_i == 1)) **begin**  
                **if**(addressLength)**begin**  
                                sda\_reg[55] <= 0;                                                   // start sign  
                                sda\_reg[54:50] <= 5'b11110;                 // Higher 2 bits of the address  
                                sda\_reg[49:48] <= adr\_i[31:30];  
                                sda\_reg[47] <= we\_i;  
                                sda\_reg[46] <= 1'bz;                                                // ACK from the slave  
                                sda\_reg[45:38] <= adr\_i[29:22];             // Lower 8 bits of the address  
                                sda\_reg[37] <= 1'bz;  
                                **if** (!we\_i)**begin**  
                                        sda\_reg[36:29] <= dat\_i[31:24];  
                                        sda\_reg[28] <= 1'bz;  
                                        sda\_reg[27:20] <= dat\_i[23:16];  
                                        sda\_reg[19] <= 1'bz;  
                                        sda\_reg[18:11] <= dat\_i[15:8];  
                                        sda\_reg[10] <= 1'bz;  
                                        sda\_reg[9:2] <= dat\_i[7:0];  
                                        sda\_reg[1] <= 1'bz;  
                                **end**  
                                **else** **begin**  
                                        sda\_reg[36:29] <= {8{1'bz}};  
                                        sda\_reg[28] <= 1'b1;  
                                        sda\_reg[27:20] <= {8{1'bz}};  
                                        sda\_reg[19] <= 1'b1;  
                                        sda\_reg[18:11] <= {8{1'bz}};  
                                        sda\_reg[10] <= 1'b1;  
                                        sda\_reg[9:2] <= {8{1'bz}};  
                                        sda\_reg[1] <= 1'b0;  
                                **end**  
                                sda\_reg[0] <= 1'b1;                                         //stop condition  
                        **end**  
                        **else** **begin**  
                                sda\_reg[55] <= 0;                                                   //start sign  
                                sda\_reg[54:48] <= adr\_i[31:25];             //7 bit address  
                                sda\_reg[47] <= we\_i;  
                                sda\_reg[46] <= 1'bz;  
                                **if** (!we\_i)**begin**  
                                        sda\_reg[45:38] <= dat\_i[31:24];  
                                        sda\_reg[37] <= 1'bz;  
                                        sda\_reg[36:29] <= dat\_i[23:16];  
                                        sda\_reg[28] <= 1'bz;  
                                        sda\_reg[27:20] <= dat\_i[15:8];  
                                        sda\_reg[19] <= 1'bz;  
                                        sda\_reg[18:11] <= dat\_i[7:0];  
                                        sda\_reg[10] <= 1'bz;  
                                **end**  
                                **else** **begin**  
                                        sda\_reg[45:38] <= {8{1'bz}};  
                                        sda\_reg[37] <= 1'b1;  
                                        sda\_reg[36:29] <= {8{1'bz}};  
                                        sda\_reg[28] <= 1'b1;  
                                        sda\_reg[27:20] <= {8{1'bz}};  
                                        sda\_reg[19] <= 1'b1;  
                                        sda\_reg[18:11] <= {8{1'bz}};  
                                        sda\_reg[10] <= 1'b0;  
                                **end**  
                                sda\_reg[9] <= 1'b1;                                         //stop condition  
                                start\_condition <= 1'b1;  
                        **end**  
        **end**  
        **if** (start && start\_condition && scl\_high)**begin** //send out start condition  
                sda\_reg <= {sda\_reg[54:0], sda\_reg[55]};  
                start\_condition <= 1'b0;  
                bit\_counter <= 1'b0;  
        **end**  
          
        **if** (start && stop\_condition && scl\_high)**begin**  //send out stop condition  
                sda\_reg <= {sda\_reg[54:0], sda\_reg[55]};  
                bit\_counter <= 1'b0;  
        **end**  
          
        **if**(start && scl\_low && !start\_condition && !stop\_condition)**begin**       //start shifting  
                sda\_reg <= {sda\_reg[54:0],sda\_reg[55]};  
                bit\_counter <= bit\_counter + 1;  
        **end**  
**end**  
  
// Set the stop condition  
**always** @(**posedge** clk\_i)  
**begin**  
        **if**(rst\_i) stop\_condition <= 0;  
        **else** **if**(addressLength)                                                                 // 10bit address   
                **if**(bit\_counter == 56)  
                        stop\_condition <= 1;  
                **else**   
                        stop\_condition <= 0;  
        **else**  
                **if**(bit\_counter == 47)  
                        stop\_condition <= 1;  
                **else**   
                        stop\_condition <= 0;  
**end**  
  
  
**assign** scl\_oe = start ? 1'b1: 1'b0;  
**assign** sda\_oe = start ? 1'b1: 1'b0;  
**assign** scl = (scl\_oe)? scl\_reg : 1'bz;  
  
**assign** sda = (sda\_oe) ? sda\_reg[55] : 1'bz;  
  
**assign** scl\_low = ((clk\_counter == (BITRATE/2)) & (!scl\_reg));  
**assign** scl\_high = ((clk\_counter == (BITRATE/2)) & ( scl\_reg));  
  
// Fill out dat\_o register with data from the periphery  
**reg** [31:0]reg\_dat\_o;  
**always** @(**posedge** clk\_i)**begin**  
        **if**(start && stop\_condition && we\_i)  
        **if** (addressLength)**begin**  
                reg\_dat\_o[31:24] <= sda\_reg[36:29];     
                reg\_dat\_o[23:16] <= sda\_reg[27:20];  
                reg\_dat\_o[15:8] <= sda\_reg[18:11];  
                reg\_dat\_o[7:0] <= sda\_reg[9:2];  
        **end**  
        **else** **begin**  
                reg\_dat\_o[31:24] <= sda\_reg[36:29];  
                reg\_dat\_o[23:16] <= sda\_reg[27:20];  
                reg\_dat\_o[15:8] <= sda\_reg[18:11];  
                reg\_dat\_o[7:0] <= sda\_reg[9:2];  
        **end**  
**end**  
  
**assign** dat\_o = reg\_dat\_o;  
  
// Set the acknowledge signal   
**reg** ack\_fb;  
**always** @(**posedge** clk\_i)  
**begin**  
        **if**(rst\_i) ack\_fb <= 0;  
        **else** **if**(stop\_condition) ack\_fb <= 1;  
        **else** ack\_fb <= 0;  
**end**  
  
**assign** ack\_o = ack\_fb;  
**endmodule**

# A szimulációhoz felhasznált testbench-ek

## A vezérlő egység adatküldése a periféria felé:

A set\_addressLength-el lehet beállítani az I2C-re kiküldött cím szélességét. Ha 1 akkor 10 bites, ha 0 akkor 7.

`timescale 1ns / 1ps  
  
**module** tb\_top;  
  
        // Inputs  
        **reg** rst;  
        **reg** clk;  
        **reg** [31:0] control\_dat;  
        **reg** [31:0] control\_adr;  
        **reg** dat\_rdy;  
        **reg** set\_addressLength;  
        **reg** we;                //we == 0 -> write  
          
          
        // Outputs  
        **wire** scl;  
          
        // Bidirs  
        **wire** sda;  
          
        // Instantiate the Unit Under Test (UUT)  
        top uut (  
                .rst(rst),   
                .clk(clk),  
                .control\_dat(control\_dat),  
                .control\_adr(control\_adr),  
                .set\_addressLength(set\_addressLength),  
                .we(we),  
                .sda(sda),  
                .scl(scl)  
        );  
  
        **initial** **begin**  
                // Initialize Inputs  
                rst = 1;  
                clk = 0;  
                control\_adr = 0;  
                control\_dat = 0;  
                dat\_rdy = 0;  
                set\_addressLength = 0;  
                we = 0;  
  
                // Wait 100 ns for global reset to finish  
                #100;  
                          
                rst = 0;  
                #5  
                control\_adr = 32'h1000\_0000;  
                control\_dat = 32'hffff\_ffff;  
                set\_addressLength = 1;                        //10 bit periphery adress  
                  
                #5  
                dat\_rdy = 1;  
                $display(dat\_rdy);  
                  
  
        **end**  
          
        **always** #5  
        clk <= ~clk;  
        
**endmodule**

## A periféria adatküldése a vezérlő egység felé 7 bites címen

`timescale 1ns / 1ps  
  
**module** tb\_top\_in;  
  
        // Inputs  
        **reg** rst;  
        **reg** clk;  
        **reg** [31:0] control\_dat;  
        **reg** [31:0] control\_adr;  
        **reg** dat\_rdy;  
        **reg** set\_addressLength;  
        **reg** we;  
  
        // Outputs  
        **wire** scl;  
  
        // Bidirs  
        **wire** sda;  
          
        // Data reg  
        **reg** [45:0]sda\_reg\_tb;  
        **reg** [31:0]data = {31{1'b1}};  
  
        // Instantiate the Unit Under Test (UUT)  
        top uut (  
                .rst(rst),   
                .clk(clk),   
                .control\_dat(control\_dat),   
                .control\_adr(control\_adr),   
                .dat\_rdy(dat\_rdy),   
                .set\_addressLength(set\_addressLength),   
                .we(we),   
                .sda(sda),   
                .scl(scl)  
        );  
  
        **initial** **begin**  
                // Initialize Inputs  
                rst = 1;  
                clk = 0;  
                control\_dat = 0;  
                control\_adr = 0;  
                dat\_rdy = 0;  
                set\_addressLength = 0;  
                we = 0;  
  
                // Wait 100 ns for global reset to finish  
                #100;  
          
                rst = 0;  
                control\_adr = 32'h1000\_0000;  
                we = 1;  
                set\_addressLength = 1;  
                dat\_rdy = 1;  
  
        **end**  
          
        **always** #5  
        clk <= ~clk;  
          
        **reg** [15:0]check\_addr;  
        **reg** old\_sda;  
        **reg** old\_scl;  
        **reg** frame\_start;  
        **reg** we\_feedback;  
        **reg** ack\_feedback;  
          
        **always** @(**posedge** clk)  
        **begin**  
        **if**(rst) old\_scl <= 0;  
   **else** old\_scl <= scl;  
        **end**  
          
        **always** @(**posedge** clk)  
        **begin**  
                **if**(rst) check\_addr <= 0;  
                **else** **if**(((sda == 0) && (old\_sda == 1)) && ((scl == 0) && (old\_scl == 1)))   
                **begin**  
                        frame\_start <= 1;  
                **end**  
                **else** frame\_start <= 0;  
        **end**  
          
        **always** @(**posedge** clk)  
        **begin**  
                **if**(rst)  
                **begin**  
                        sda\_reg\_tb[45] <= 1'bz;  
                        sda\_reg\_tb[44:37] <= {8{1'bz}};  
                        sda\_reg\_tb[36] <= 1'b1;  
                        sda\_reg\_tb[35:28] <= data[31:24];  
                        sda\_reg\_tb[27] <= 1'bz;  
                        sda\_reg\_tb[26:19] <= data[23:16];  
                        sda\_reg\_tb[18] <= 1'bz;  
                        sda\_reg\_tb[17:10] <= data[15:8];  
                        sda\_reg\_tb[9] <= 1'bz;  
                        sda\_reg\_tb[8:1] <= data[7:0];  
                        sda\_reg\_tb[0] <= 1'bz;  
                **end**  
        **end**  
          
  
        **reg** tmp;  
        **always** @(**posedge** clk)  
        **begin**  
                **if**((old\_scl == 0) && (scl == 1))  
                **begin**  
                        tmp <= 1;  
                        sda\_reg\_tb <= {sda\_reg\_tb[53:0],sda\_reg\_tb[54]};  
                **end**  
                **else** tmp <= 0;  
        **end**  
          
  
        **assign** sda = sda\_reg\_tb[54];  
        
**endmodule**

## A periféria adatküldése a vezérlő egység felé 10 bites címen

`timescale 1ns / 1ps  
  
  
**module** tb\_top\_in;  
  
        // Inputs  
        **reg** rst;  
        **reg** clk;  
        **reg** [31:0] control\_dat;  
        **reg** [31:0] control\_adr;  
        **reg** dat\_rdy;  
        **reg** set\_addressLength;  
        **reg** we;  
  
        // Outputs  
        **wire** scl;  
  
        // Bidirs  
        **wire** sda;  
          
        // Data reg  
        **reg** [54:0]sda\_reg\_tb;  
        **reg** [31:0]data = {31{1'b1}};  
  
        // Instantiate the Unit Under Test (UUT)  
        top uut (  
                .rst(rst),   
                .clk(clk),   
                .control\_dat(control\_dat),   
                .control\_adr(control\_adr),   
                .dat\_rdy(dat\_rdy),   
                .set\_addressLength(set\_addressLength),   
                .we(we),   
                .sda(sda),   
                .scl(scl)  
        );  
  
        **initial** **begin**  
                // Initialize Inputs  
                rst = 1;  
                clk = 0;  
                control\_dat = 0;  
                control\_adr = 0;  
                dat\_rdy = 0;  
                set\_addressLength = 0;  
                we = 0;  
  
                // Wait 100 ns for global reset to finish  
                #100;  
          
                rst = 0;  
                control\_adr = 32'h1000\_0000;  
                we = 1;  
                set\_addressLength = 1;  
                dat\_rdy = 1;  
  
        **end**  
          
        **always** #5  
        clk <= ~clk;  
          
        **reg** [15:0]check\_addr;  
        **reg** old\_sda;  
        **reg** old\_scl;  
        **reg** frame\_start;  
        **reg** we\_feedback;  
        **reg** ack\_feedback;  
          
        **always** @(**posedge** clk)  
        **begin**  
        **if**(rst) old\_scl <= 0;  
   **else** old\_scl <= scl;  
        **end**  
          
        **always** @(**posedge** clk)  
        **begin**  
                **if**(rst) check\_addr <= 0;  
                **else** **if**(((sda == 0) && (old\_sda == 1)) && ((scl == 0) && (old\_scl == 1)))   
                **begin**  
                        frame\_start <= 1;  
                **end**  
                **else** frame\_start <= 0;  
        **end**  
          
        **always** @(**posedge** clk)  
        **begin**  
                **if**(rst)  
                **begin**  
                        sda\_reg\_tb[54] <= 1'bz;  
                        sda\_reg\_tb[53:46] <= {8{1'bz}};  
                        sda\_reg\_tb[45] <= 1'b1;  
                        sda\_reg\_tb[44:37] <= {8{1'bz}};  
                        sda\_reg\_tb[36] = 1'b1;  
                        sda\_reg\_tb[35:28] <= data[31:24];  
                        sda\_reg\_tb[27] <= 1'bz;  
                        sda\_reg\_tb[26:19] <= data[23:16];  
                        sda\_reg\_tb[18] <= 1'bz;  
                        sda\_reg\_tb[17:10] <= data[15:8];  
                        sda\_reg\_tb[9] <= 1'bz;  
                        sda\_reg\_tb[8:1] <= data[7:0];  
                        sda\_reg\_tb[0] <= 1'bz;  
                **end**  
        **end**  
          
  
        **reg** tmp;  
        **always** @(**posedge** clk)  
        **begin**  
                **if**((old\_scl == 0) && (scl == 1))  
                **begin**  
                        tmp <= 1;  
                        sda\_reg\_tb <= {sda\_reg\_tb[53:0],sda\_reg\_tb[54]};  
                **end**  
                **else** tmp <= 0;  
        **end**  
          
  
        **assign** sda = sda\_reg\_tb[54];  
        
**endmodule**

# A teszt eredményei

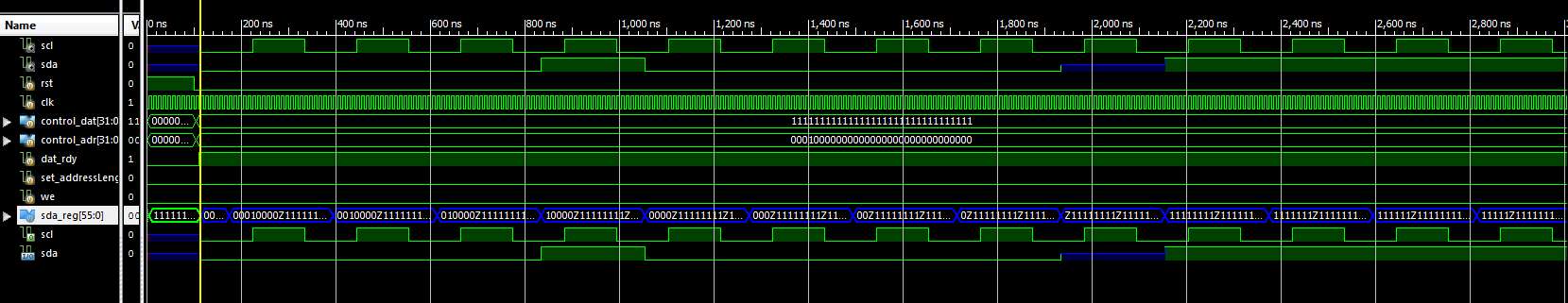
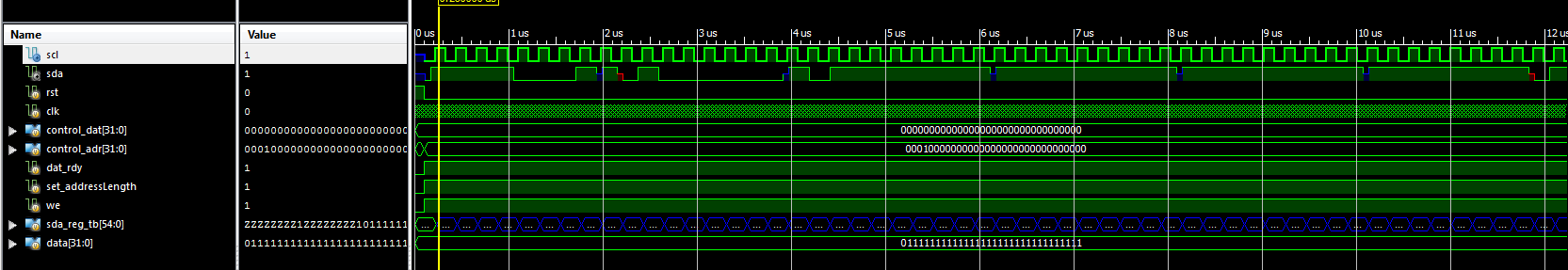
## 7 bites cím esetén

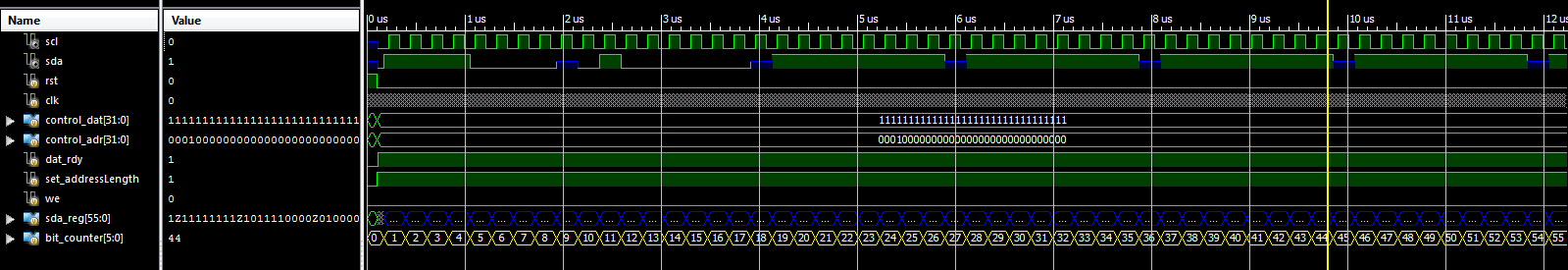
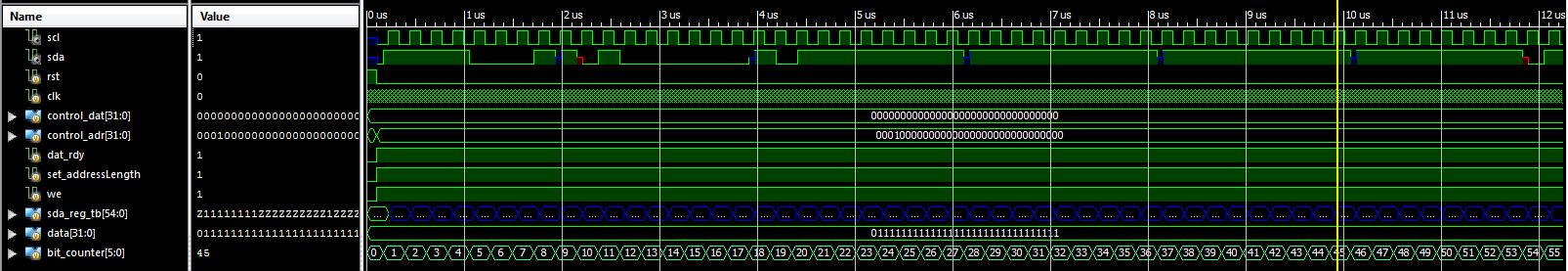
Először a vezérlő egység által kiküldött jeleket vizsgáltuk meg. A teszt során, amint az a testbench-ekből is látszik, egy csupa 1-esből álló adatregisztert használtunk. A WishBone slave címe a 0x100000000. A 3. ábrán az látszik, hogy a vezérlő egység ír a perifériába. Az sda vezeték minden scl low értékekor változhat. Az ack jelnél az I2C elengedi az sda-t, hogy a periféria visszajelezhessen.

A 4. ábrán pedig az olvasás művelet látható. Látszik, ahogy az I2C felhúzza az sda vezetéket, amikor a periféria a visszajelzésre vár. Mivel az I2C a low érték felénél vált, ezért néhány órajelnyi inkonzisztens állapot figyelhető meg az sda vonalon.

## 10 bites cím esetén

10 bit esetén hasonló működés várható. Az I2C specifikációja szerint a 10 bites címet 2 keretben lehet kiküldeni. Az elsőnél a cím 2 legnagyobb helyiértékű bitje kerül kiküldésre, az 11110 bitsorozat után. Ezt követi a jel, majd a következő keretben a maradék 8 címbit.





3. ábra: A vezérlő egység ír a perifériába

5. ábra: A vezérlő egység olvas a perifériából