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

Rendszerarchitektúrák Házi Feladat

Wishbone – I2C kommunikáció megvalósítása

Készítette:

Moró Anna (KIHLI2)

Murai János (D0YRUM)

Konzulens:

Wacha Gábor

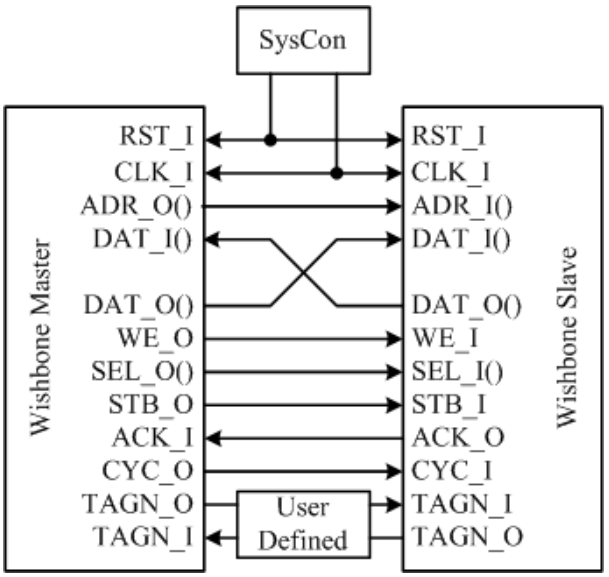
Specifikáció

A Wishbone egy nyílt forrású, párhuzamos busz, mely ingyenes elérhető bárki számára. A feladatunk, hogy ehhez illesszük a rendkívül elterjed I^2C 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.

I^2C

Az I^2C 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 I^2C 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.

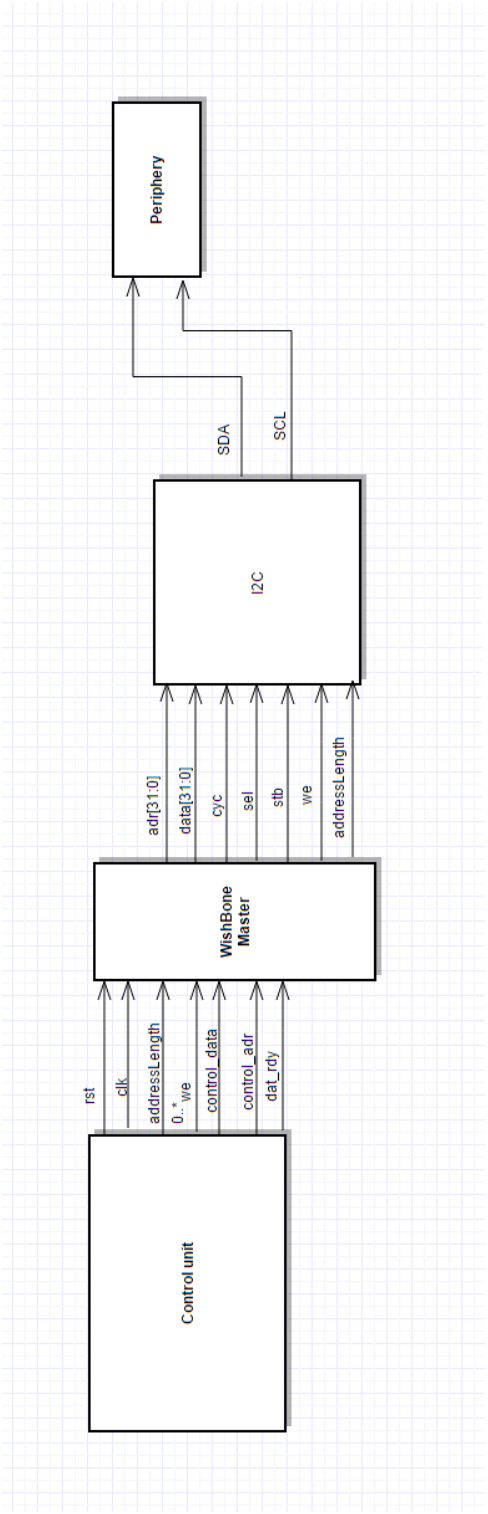
Az átvitel során minden byte-ot egy nyugtázás bit követ. A nyugtázással kapcsolatos órajel impulzust a master generálja. A küldő szabaddá teszi az SDA vezetéket (magas állapottal) a nyugtázás órajel impulzusa alatt. Amennyiben a fogadó elfogadja az adott byte-ot, akkor fel kell húznia az SDA vezetéket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

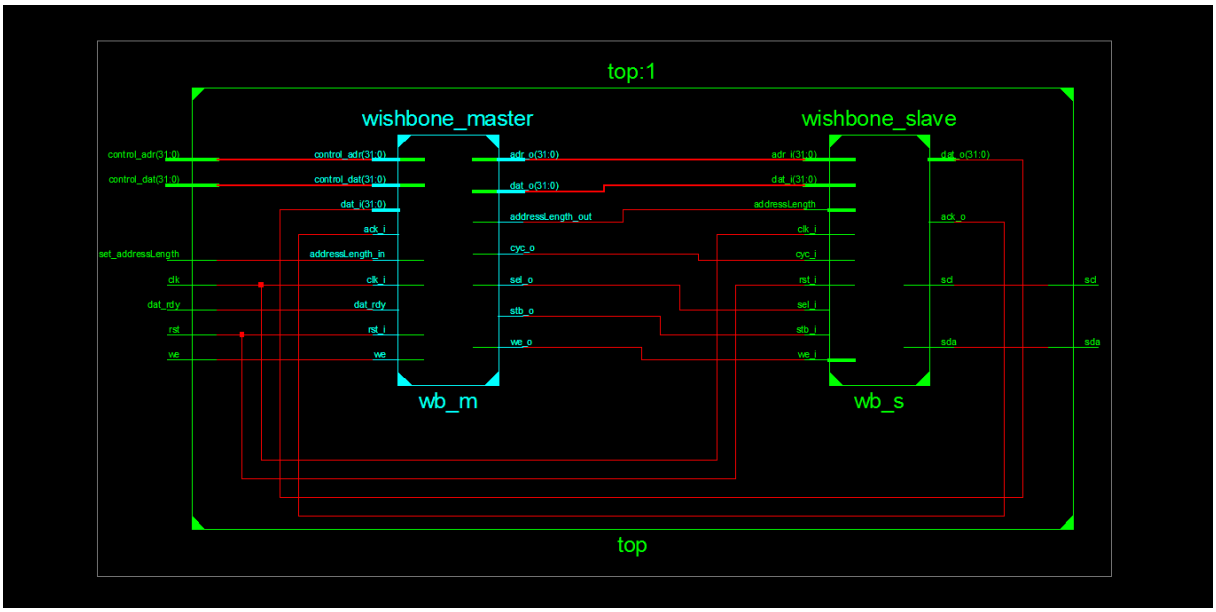
Schematic

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:



2. 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 I^2C-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

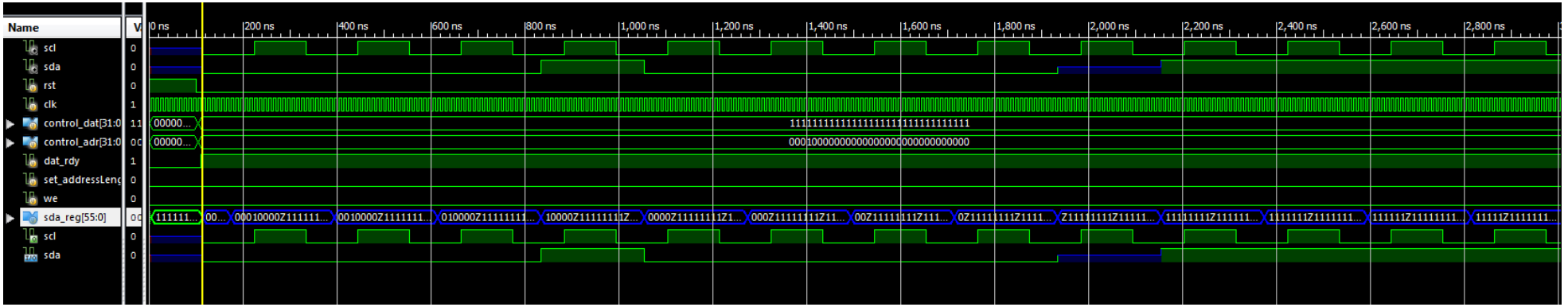
7 bites cím esetén

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 I^2C elengedi az sda-t, hogy a periféria visszajelezhessen.

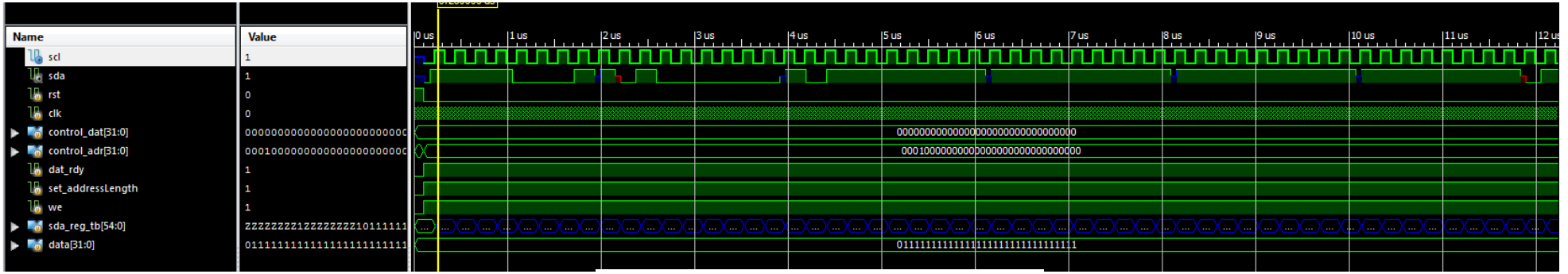
A 4. ábrán pedig az olvasás művelet látható. Látszik, ahogy az I^2C felhúzza az sda vezetéket, amikor a periféria a visszajelzésre vár. Mivel az I^2C a low érték felénél vált, ezért néhány órajelnyi inkonzisztens állapot figyelhető meg az sda vonalon.

10 bites cím esetén

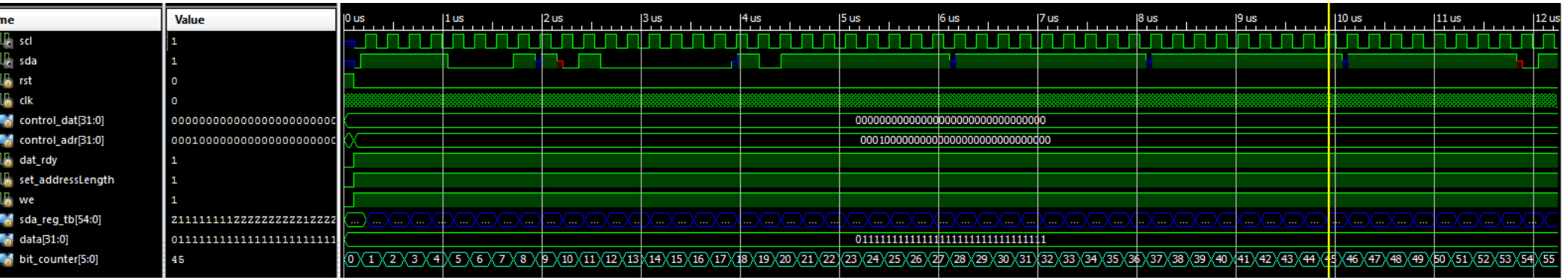
10 bit esetén hasonló működés várható. Az I^2C specifikációja szerint a 10 bites címet 2 keretben lehet kiküldeni. Az elsőnél a cím 2 legnagyobb helyiértékű bitje kerül kiküldésre, az 11110 bitsorozat után. Ezt követi a $r/\overline{w}$ jel, majd a következő keretben a maradék 8 címbit.



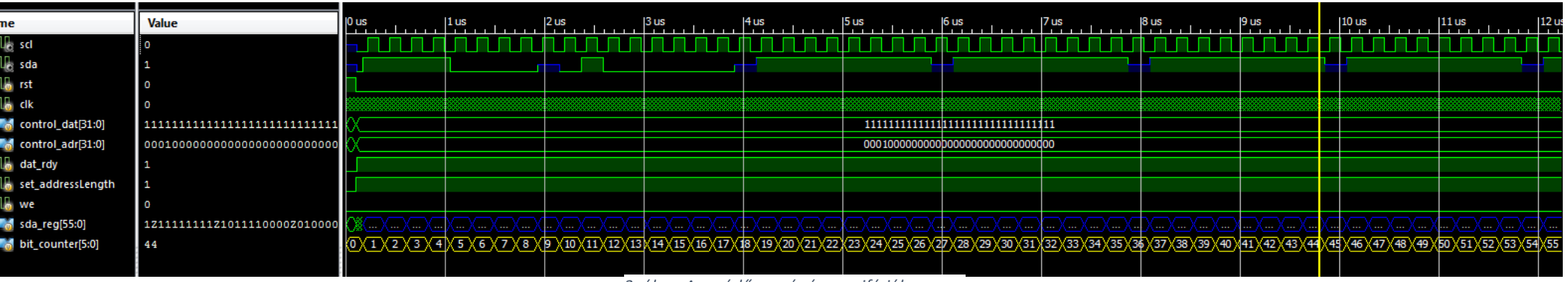
3. ábra: A vezérlő egység ír a perifériába



4. ábra: A vezérlő egység olvas a perifériából



5. ábra: A vezérlő egység olvas a perifériából



3. ábra: A vezérlő egység ír a perifériába