



M Ű E G Y E T E M 1 7 8 2

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Méréstechnika és Információs Rendszerek Tanszék

Rendszerarchitektúrák (VIMIMA08)

APB UART interfész

HOLLÓS ÁDÁM (HX8ROW)

SZABÓ ÁRON ()

2019. MÁJUS 16.

Tartalomjegyzék

1. Bevezetés	2
1.1. A feladat	2
1.2. A választott periféria	2
2. Specifikáció	3
3. Tervezés	4
3.1. Busz bemenetek és kimenetek	4
3.2. Transmit modul	6
3.3. Receive modul	6
4. Megvalósítás	7
5. Tesztelés	11

1. Bevezetés

1.1. A feladat

Kétfős csapatunk részére a feladat egy perifériaillesztő áramkör megtervezése volt:

1. A tervezendő periféria specifikációinak meghatározása
2. Blokkvázlat megtervezése
3. HDL Implementáció
4. Bus Functional Model elkészítése
5. Dokumentáció elkészítése

1.2. A választott periféria

Az általunk választott perifériaillesztő:

- Busz típusa: **AMBA APB 32 bit / 16 MHz**
- Periféria típusa: **UART**

A perifériaillesztőt Xilinx Vivado használatával, Verilog nyelven valósítjuk meg.

2. Specifikáció

1. A periféria bemeneti és kimeneti jelei

- (a) **Rx:** UART fogadás (*bemenet*)
- (b) **Tx:** UART küldés (*kimenet*)
- (c) **PCLK:** AMBA APB busz órajel, a felfutó él ütemez minden átvitelt. A perifériánk belső órajele is ez lesz (*bemenet*)
- (d) **PRESETn:** AMBA APB reset jel, aktív alacsony, a busz reset jeléhez kapcsolódik (*bemenet*)
- (e) **PADDR:** AMBA APB címbusz, 32 bit széles (*bemenet*)
- (f) **PSEL:** AMBA APB select jel a perifériának (*bemenet*)
- (g) **PENABLE:** AMBA APB transzfer engedélyező jel (*bemenet*)
- (h) **PWRITE:** AMBA APB transzfer típus (magas: írás a perifériába, alacsony: olvasás a perifériából) (*bemenet*)
- (i) **PWDATA:** AMBA APB perifériába írt adat, 32 bit széles (*bemenet*)
- (j) **PSTRB:** AMBA APB írt adat érvényes bytejai, 4 bit széles (*bemenet*)
- (k) **PREADY:** A periféria visszajelzése az AMBA APB busznak (*kimenet*)
- (l) **PRDATA:** AMBA APB perifériából olvasott adat, 32 bit széles (*kimenet*)

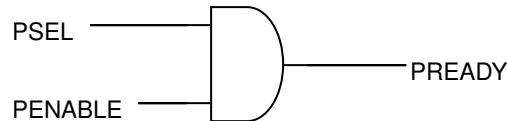
2. A periféria regiszterei

- (a) **0x00000000:** Konfigurációs regiszter
 - i. **[15:0]:** N ($baud = \frac{f_{clk}}{N*16}$)
 - ii. **[16]:** Magas: 8 bites átvitel, Alacsony: 7 bites átvitel
 - iii. **[17]:** Magas: 2 stop bit, Alacsony: 1 stop bit
 - iv. **[18]:** Periféria engedélyezése (Magas: engedélyezve, Alacsony: tiltva)
- (b) **0x00000004:** Státusz regiszter
 - i. **[0]:** Transmit FIFO üres
 - ii. **[1]:** Transmit FIFO tele
 - iii. **[8]:** Receive FIFO üres
 - iv. **[9]:** Receive FIFO tele
- (c) **0x00000008:** Receive regiszter (alsó 8 bit)
- (d) **0x0000000C:** Transmit regiszter (alsó 8 bit)

3. Tervezés

3.1. Busz bemenetek és kimenetek

A PREADY jelet a busz specifikációja szerint a perifériaillesztő adja ki, ha elkészült az írási vagy olvasási művelettel. Mivel a perifériaillesztő minden művelettel egyetlen órajel alatt végez, a PREADY jelet magasba lehet állítani már a tranzakció kezdetekor. A PENABLE, és a perifériára vonatkozó PSEL jel egyértelműen jelzi a tranzakció létét. (1. ábra)

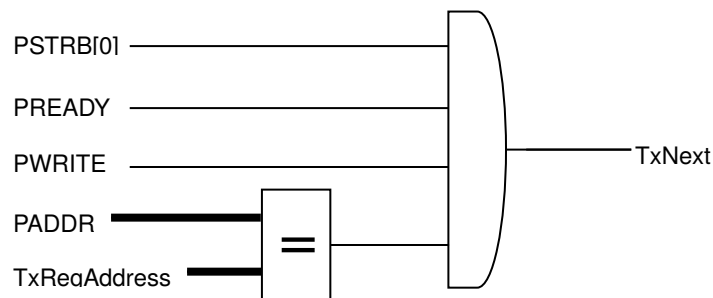


1. ábra. A PREADY jel kombinációs logikája

Ha a megfelelő regiszter van címezve, akkor a perifériaillesztő adatot kell, hogy beolvasson az adatbuszról. A transmit modulban egy FIFO várja az új adatot, így ennek bemenetét kell engedélyezni a beolvasáshoz. A FIFO bemenetére (*TxDData*) így közvetlenül ráköthető az adatbusz alsó 8 bitje, és az engedélyező jelet (*TxNext*), pedig egy címezést vizsgáló logika adja. (2. és 3. ábra)



2. ábra. Küldendő adat továbbítása a transmit modulnak

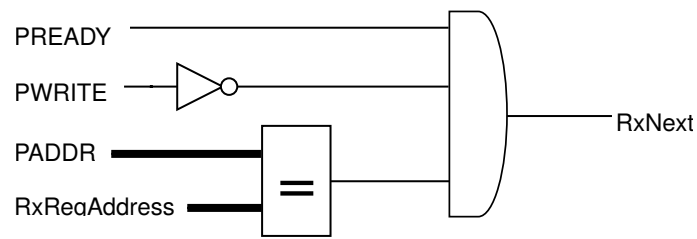


3. ábra. Új adat jelzése a transmit modulnak

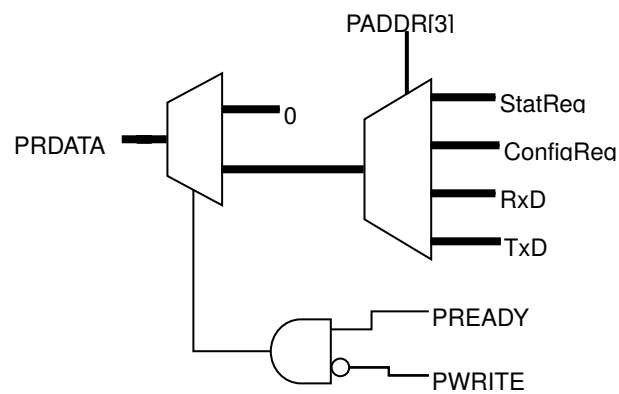
Hasonlóképpen kell jelezni az olvasást is, az *RxNext* jelet a receive modul kapja meg, ahol a FIFO kimeneti engedélyező bemenetére van kapcsolva. (4. ábra)

Az APB buszra a címzés alapján hivatkozott regiszter tartalma kerül. Fontos, hogy ha a busz nem éppen a perifériaillesztőről olvas, akkor az ne hajtsa meg az adatbuszt. (5. ábra)

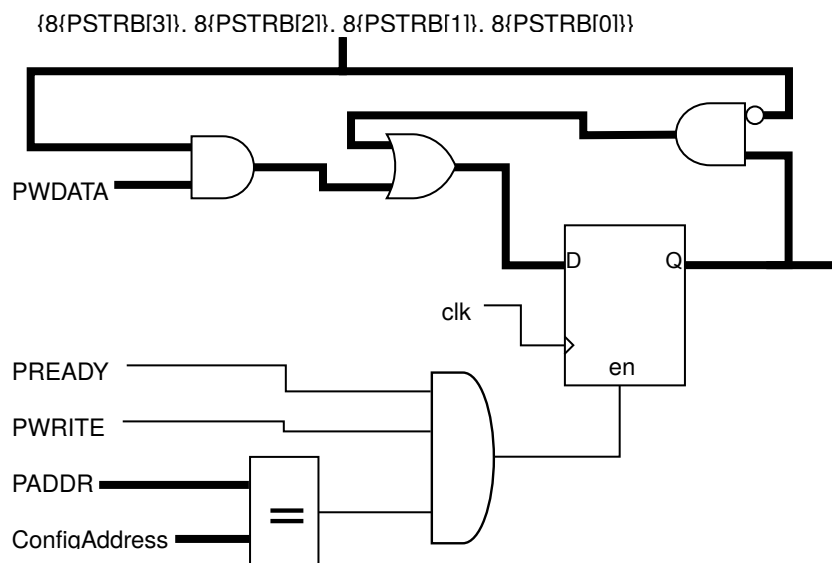
A konfigurációs regiszternél meg kellett oldani, hogy csak a kívánt byteok módosuljanak a regiszter írásakor. A négybites *PSTRB* bemenetből egy bitmaszkot hozunk létre, és ennek alapján töltjük be az új értéket, illetve tartjuk meg az előzőt. A regiszter engedélyező bemenetére egy címzési logikát kötöttünk. (6. ábra)



4. ábra. Adat olvasásának jelzése a receive modulnak



5. ábra. Buszra kerülő adatok

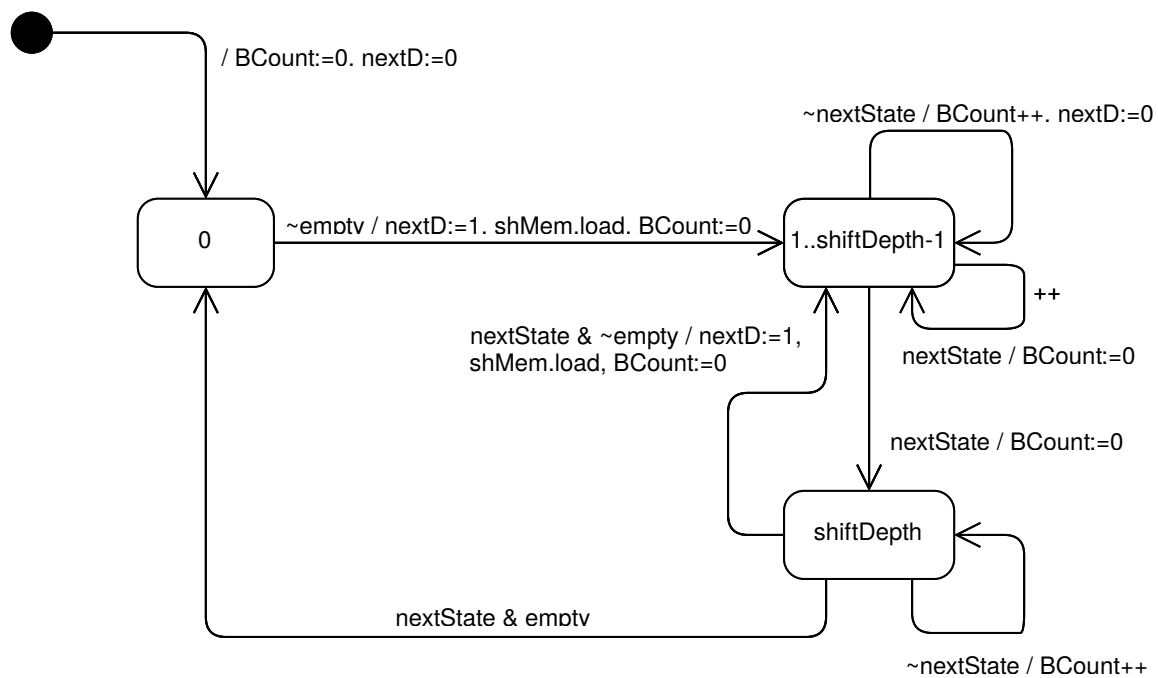


6. ábra. Konfigurációs regiszter beállítása

3.2. Transmit modul

Az UART küldő modul logikailag különválasztható, így külön almodulként valósítjuk meg. Bemenetként megkapja a szükséges konfigurációs regiszterek értékeit, illetve a buszról érkező küldendő adatot. A fő modul felé jelzi, ha a FIFO tele van, vagy üres, illetve az UART kimenet közvetlenül az egész perifériaillesztő kimenete is.

A transmit modul egy állapotgép alapján működik. (7. ábra) A megfelelő baud rate-et a *BCount* számlálóval érjük el, ami a konfiguráció alapján kiszámolt végértéknél lépteti az állapotgépet. A kimeneti regiszter (*shMem*) értékének betöltését az ábrán *shMem.load*-dal jelöltük, ennek értéke a küldendő adat, kiegészítve a megfelelő start és stop bitekkel. 0-tól különböző állapot esetén az UART kimenetre ennek a regiszternek az állapot számával címzett bitje kerül.

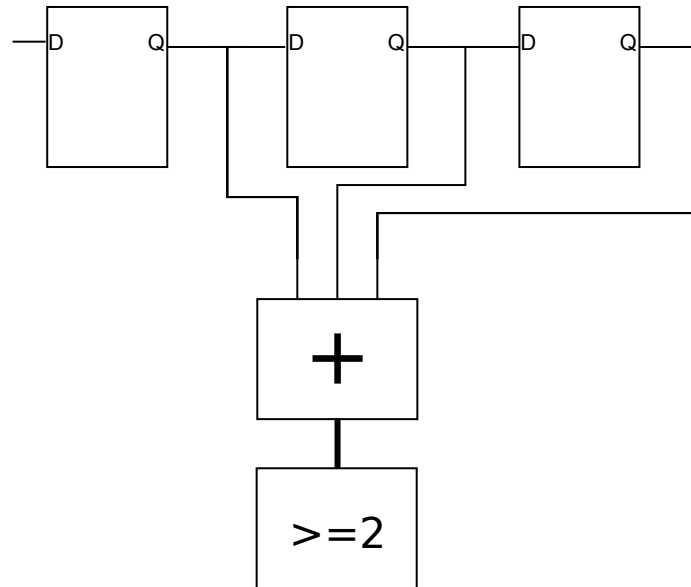


7. ábra. A küldő modul állapotgépe

Rst jelre a modul 0 állapotba kerül, ez az állapot jelenti az UART küldés felfüggesztését. Ha a FIFO nem üres, a kimeneti regiszterbe bekerül belőle egy adat (a *nextD* jel kivesz a FIFO-ból egy elemet), és az állapotgép 1-es állapotba kerül. Az állapotok 1-től *shiftDepth*-ig (ami a konfiguráció alapján a kimeneti regiszter adatmérete). Az utolsó bit után az állapotgép újraindul, ha a FIFO-ban van még adat, illetve 0-ás állapotba kerül, ha nincs. 0-ás állapotban a kimenet magasban van (nincs adat).

3.3. Receive modul

A fogadó modul egy hasonló állapotgéppel működik, mint a küldő, az állapotok itt is az egyes bitek helyét kódolják a byteon belül. A modul *SClk* órajellel vesz mintát az UART bemenetről, ez az órajel az APB órajel N-ed része (ld. konfigurációs regiszter), a baud rate 16-szorosa. Az állapotgép kezdő állapota a 0, ez jelenti, hogy a fogadó a start bitre vár. Ekkor alacsony jel észlelésekor vár fél bitidőt, és ha a jel ezalatt mindvégig alacsony, akkor elindul az állapotgép, és a konfigurációnak megfelelően beolvassa a byteot. A *shMem* regiszterbe a bemenet egy többségi szavazással (8. ábra) beolvasott értéke kerül, a bitidő közepéről mintavételezve.



8. ábra. A beérkező bit többségi szavazással beolvasva

4. Megvalósítás

```

module APB_UART (
    output Tx,
    input Rx,
    input PCLK,
    input PRESETn,
    input [31:0] PADDR,
    input PSEL,
    input PENABLE,
    input PWRITE,
    input [31:0] PWDATA,
    input [3:0] PSTRB,
    output PREADY,
    output [31:0] PRDATA
);

assign PREADY = (PSEL & PENABLE);

wire TxNext, TxEmpty, TxFull;
assign TxNext = (PSTRB[0] & PREADY & PWRITE & (PADDR == 32'd12));
transmitter tr (
    .clk(PCLK),
    .rst(!PRESETn),
    .N(config[15:0]),
    .eight_bit(config[16]),
    .two_stop(config[17]),
    .data(PWDATA[7:0]),
    .next(TxNext),
    .Tx(Tx),
    .empty(TxEmpty),
    .full(TxFull)
);

wire [7:0] RxData;
wire RxNext, RxEmpty, RxFull;
assign RxNext = (PREADY & !PWRITE & (PADDR == 32'd8));
receiver rec(
    .clk(PCLK),
    .rst(!PRESETn || (!config[18])), //Hold in reset if disabled

```



```

        .N(config[15:0]),
        .eight_bit(config[16]),
        .Rx(Rx),
        .data(RxData),
        .next(RxNext),
        .empty(RxEmpty),
        .full(RxFull)
    );

    reg [18:0] config;
    wire config_en;
    wire [18:0] M;
    assign M = {{3{PSTRB[2]}}, {8{PSTRB[1]}}, {8{PSTRB[0]}}};
    assign config_en = (PREADY & PWRITE & (PADDR == 32'd0));
    always @(posedge PCLK) begin
        if(!PRESETn)
            config <= {3'b001, 16'd100}; //defaults: 8N1, receiver disabled, baud=62'500
        else begin
            if(config_en) begin
                config <= ((config & (~M)) | (PWDATA[18:0] & M)); //config[n]=config[n-1]+M*(PWDATA[n-1]-config[n-1])
            end
        end
    end
end

wire [31:0] mux;
assign mux = (
    (PADDR == 32'd0) ? ({13'd0, config}) : (
        (PADDR == 32'd4) ? ({16'd0, {6'b0, RxFull, RxEmpty}, {6'b0, TxFull, TxEmpty}}) : (
            (PADDR == 32'd8) ? ({24'd0, RxData}) : (
                32'd0 //default: 000000000000000000000000
            )
        )
    );

assign PRDATA = ((PREADY && (!PWRITE)) ? mux : 32'd0);
endmodule

```

Forráskód 1. APB UART top modul

```

module receiver(
    input clk,
    input rst,
    input [15:0] N, //f_sampling = f_osc / N = 16 * baud
    input eight_bit,
    input Rx,
    output [7:0] data,
    input next,
    output empty,
    output full
);

    wire SClk;
    reg [15:0] SClk_count;
    assign SClk = (N == 16'd1) ? clk : (SClk_count == N-1);
    always @(posedge clk) begin
        if(rst)
            SClk_count <= 0;
        else if(SClk)
            SClk_count <= 0;
        else
            SClk_count <= SClk_count + 1;
    end

    reg sample;
    always @(posedge clk)
        if(SClk)
            sample <= Rx;

    reg buff[1:0];
    wire dec;
    assign dec = ((sample + buff[0] + buff[1]) >= 2'd2);

```

```

always @(posedge clk) begin
    if(SClk) begin
        buff[0] <= buff[1];
        buff[1] <= sample;
    end
end

reg [7:0] shMem;
reg rec;
FIFO #(.depth(32)) mem(
    .clk(clk),
    .rst(rst),
    .in(shMem),
    .in_next(rec),
    .out(data),
    .out_next(next),
    .empty(empty),
    .full(full)
);

reg [3:0] state; //0:wait, 1:start bit, 2:data bits
reg [3:0] cnt;
always @(posedge clk) begin
    if(rst) begin
        state <= 4'd0;
        rec <= 1'b0;
    end
    else if(SClk) begin
        if(state == 4'd0) begin
            if(sample == 1'b0) begin //start condition
                state <= 4'd1;
                cnt <= 4'd0;
            end
            end
            else if(state == 4'd1) begin
                if(sample == 1'b1)
                    state <= 4'd0;
                else begin
                    if(cnt == 8) begin //if the start condition lasted for half a bit time + 1 sample
                        cnt <= 4'd0;
                        state <= state + 1;
                    end
                    else
                        cnt <= cnt + 1;
                end
            end
            end
            else if(state == (4'd8 + eight_bit)) begin //last bit
                if(cnt == 4'd15) begin
                    state <= 4'd0;
                    shMem[state - 4'd2] <= dec;
                    rec <= 1'b1;
                end
                else
                    cnt <= cnt + 1;
            end
            else begin
                if(cnt == 4'd15) begin
                    cnt <= 4'd0;
                    state <= state + 4'd1;
                    shMem[state - 4'd2] <= dec;
                end
                else
                    cnt <= cnt + 1;
            end
        end
        end
        else
            rec <= 1'b0;
    end
endmodule

```

Forráskód 2. Receive modul

```

module transmitter(
    input clk,
    input rst,
    input [15:0] N, //f_sampling = f_clk / N = 16 * baud
    input eight_bit,
    input two_stop,
    output empty,
    output full,
    input [7:0] data,
    input next,
    output Tx
);

wire [7:0] D;
reg nextD;
FIFO #(.depth(32)) mem(
    .clk(clk),
    .rst(rst),
    .in(data),
    .in_next(next),
    .out(D),
    .out_next(nextD),
    .empty(empty),
    .full(full)
);

reg [10:0] shMem;
reg [3:0] state; //0: wait, 1-19: bit_num-stop_num: send
reg [19:0] BCount;
wire nextState;
assign nextState = (BCount == ({N, 4'd0} - 1));
wire [3:0] shiftDepth;
assign shiftDepth = (4'd1 + 4'd7 + {3'd0, eight_bit} + 4'd1 + {3'd0, two_stop}); //start_bit + 7
assign Tx = (state == 4'd0) ? 1'b1 : shMem[state - 4'd1];

always @(posedge clk) begin
    if(rst) begin
        BCount <= 0;
        state <= 0;
        nextD <= 0;
    end
    else begin
        if(state == 0) begin
            if(!empty) begin
                nextD <= 1;
                shMem <= {2'b11, (eight_bit ? D[7] : 1'b1), D[6:0], 1'b0}; //11 - eight_bit or stop
                state <= 4'd1;
                BCount <= 19'd0;
            end
        end
        else begin
            if(state == shiftDepth) begin
                if(nextState) begin
                    if(!empty) begin
                        nextD <= 1;
                        shMem <= {2'b11, (eight_bit ? D[7] : 1'b1), D[6:0], 1'b0}; //11 - eight_bit
                        state <= 4'd1;
                        BCount <= 19'd0;
                    end
                else
                    state <= 4'd0;
                end
            else begin
                BCount <= BCount + 1;
            end
        end
        else begin
            if(nextState) begin

```

```

        state <= state + 1;
        BCount <= 19'd0;
    end
    else begin
        BCount <= BCount + 1;
        nextD <= 0;
    end
end
end
end
end
endmodule

```

Forráskód 3. Transmit modul

```

module FIFO #(parameter depth = 32) (
    input clk,
    input rst,
    input [7:0] in,
    input in_next,
    output [7:0] out,
    input out_next,
    output empty,
    output full
);

    reg [$clog2(depth+1)-1:0] count;
    reg [7:0] mem [depth-1:0];

    assign empty = (count == 0);
    assign full = (count == depth);
    assign out = mem[0];

    integer ii;
    always @(posedge clk) begin
        if(rst)
            count <= 0;
        else begin
            if(in_next && !out_next) begin //write
                count <= count + 1;
                mem[count] <= in;
            end

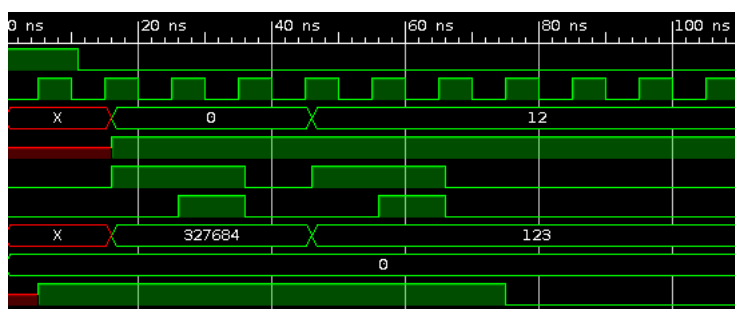
            if(!in_next && out_next) begin //read
                count <= count - 1;
                for(ii = 0; ii < (depth-1); ii = ii + 1 )
                    mem[ii] <= mem[ii+1];
            end

            if(in_next && out_next) begin //write and read
                mem[count] = in;
                for(ii = 0; ii < (depth-1); ii = ii + 1 )
                    mem[ii] <= mem[ii+1];
            end
        end
    end
endmodule

```

Forráskód 4. FIFO verilog modul

5. Tesztelés



9. ábra. Szimulációs jelalak