EHB436E

DIGITAL SYSTEM DESIGN APPLICATIONS

Muhammed Erkmen

040170049

Final Project REPORT

A/B with ALU

Abstract

In this project i designed an ALU, a register block and a control unit. My challange is dividing 8 bit unsigned integer A to 4 bit unsigned integer B (Group MSB). I implemented 2 different methods. First one is **Non-Restoring Division Algorithm.** Second one i designed which works faster in small numbers but slower in big numbers, makes substractions till the answer is negative and counts this substraction steps as quotient. When the answer is negative, algorithm adds a dividor to this answer and new answer is remainder.

Non-Restoring Division Algorithm

This algorithm is used to implement unsigned integer division. If the N bit integer A dividing by another M bit integer B, there will be N bit steps in this algorithm (Attention: A>B).

First step is shifting number A through R. Then we look the first bit of R.

If it is 0, then we subtract B from R and save to R.

If it is 1, then we add R and B and save to A.

In next step we look Cout of the addition/substraction.

If it is 0, then Q[STEP] = 1

If it is 1, then Q[STEP] = 0

I added next page an example for this algorithm step by step.

You can follow how bits of R effect the action(ACT) from looking the colors.

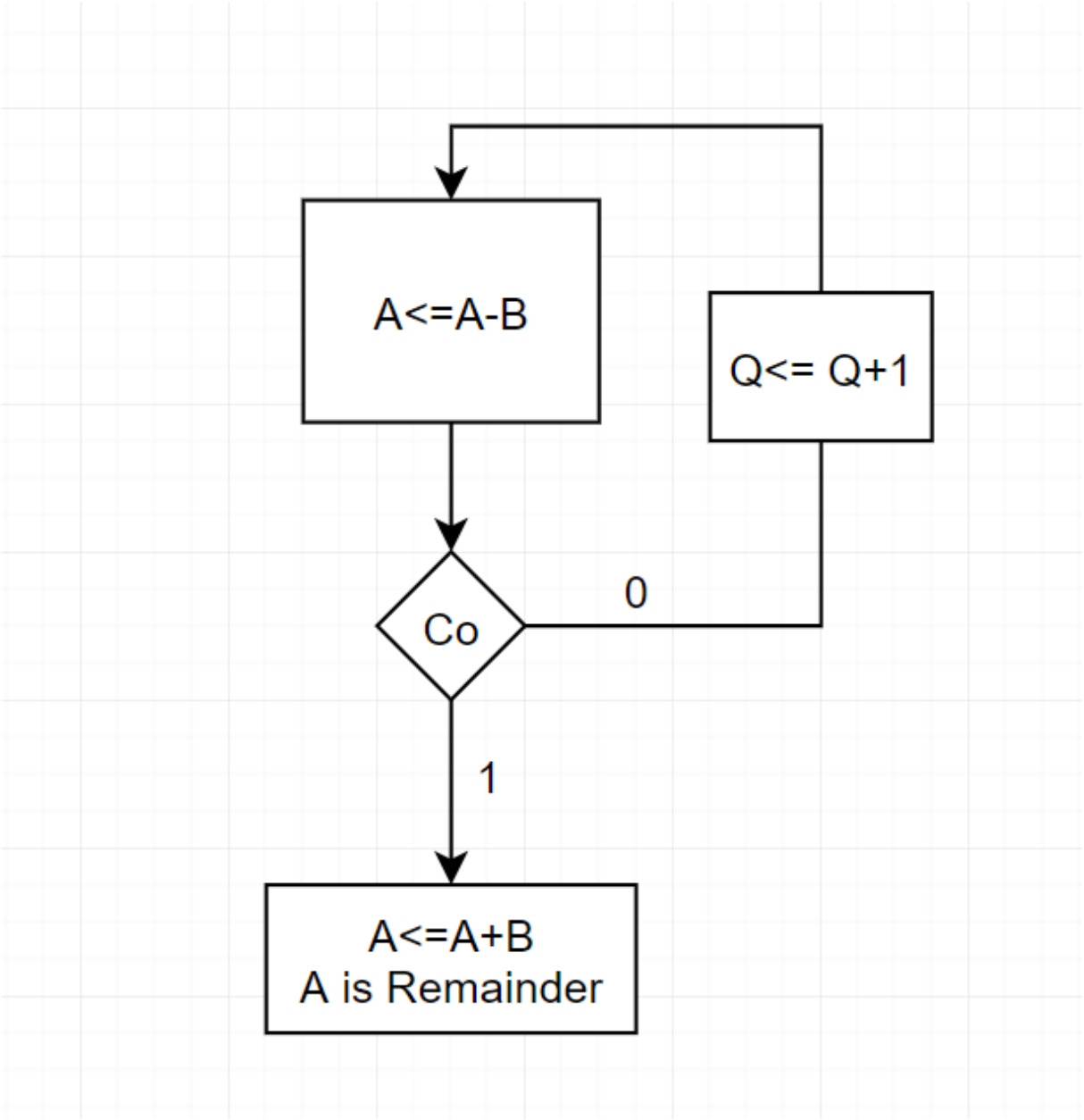
An example of non-restoring division method with all steps made by me

A = 0000 1001 (Q)
B = 0000 0011
-B = 1111 1101

N	B	R	Q	ACT
8	0000 0011	0000 0000	0000 1001	BEGIN
	0000 0011	0000 0000	0001 001_	SHIFT LEFT AQ
	0000 0011	1111 1101	0001 001_	R= R-B
7	0000 0011	1111 1101	0001 0010	Q[0]=0
	0000 0011	1111 1010	0010 010_	SHIFT LEFT AQ
	0000 0011	1111 1101	0010 010_	R= R+B
6	0000 0011	1111 1101	0010 0100	Q[0]=0
	0000 0011	1111 1010	0100 100_	SHIFT LEFT AQ
	0000 0011	1111 1101	0100 100_	R= R+B
5	0000 0011	1111 1101	0100 1000	Q[0]=0
	0000 0011	1111 1010	1001 000_	SHIFT LEFT AQ
	0000 0011	1111 1101	1001 000_	R= R+B
4	0000 0011	1111 1101	1001 0000	Q[0]=0
	0000 0011	1111 1011	0010 000_	SHIFT LEFT AQ
	0000 0011	1111 1110	0010 000_	R= R+B
3	0000 0011	1111 1110	0010 0000	Q[0]=0
	0000 0011	1111 1100	0100 000_	SHIFT LEFT AQ
	0000 0011	1111 1111	0100 000_	R= R+B
2	0000 0011	1111 1111	0100 0000	Q[0]=0
	0000 0011	1111 1110	1000 000_	SHIFT LEFT AQ
	0000 0011	0000 0001	1000 000_	R= R+B
1	0000 0011	0000 0001	1000 0001	Q[0]=1
	0000 0011	0000 0011	0000 001_	SHIFT LEFT AQ
	0000 0011	0000 0000	0000 001_	R= R-B
0	0000 0011	0000 0000	0000 0011	Q[0]=1
		kalan 0	bölüm 3	

My Simple Algorithm

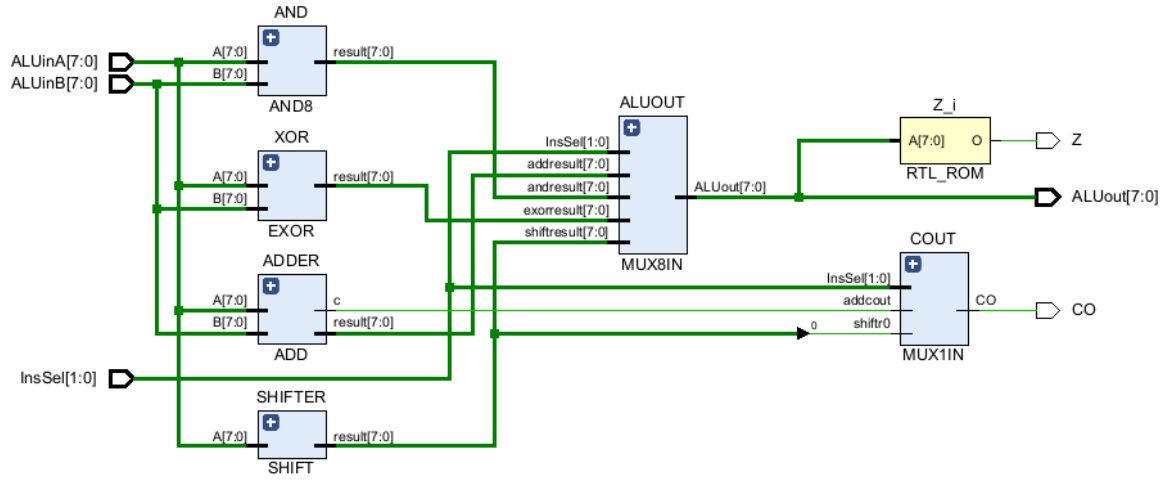
As i explained before, i make substractions till the result becomes negative and counts these substractions as quotient. When result becomes negative, it adds divider to dividend so that is remainder. Here is simplest block of my algorithm:



ALU

I designed the ALU same as given in experiment Figure-3 ALU sheet.

Here is RTL schematic of my ALU block:



Codes of my ALU Block:

8-bit ADD

```
`timescale 1ns / 1ps

module ADD(
input [7:0] A,
input [7:0] B,
output reg [7:0] result,
output c
    );

reg [8:0] ADDITION;
wire [8:0] fakeA = {1'b0,A};
wire [8:0] fakeB = {B[7],B};
assign c = ADDITION[8];

always @(A,B) begin
result <= A+B;
ADDITION <= fakeA + fakeB;
end
endmodule
```

8-bit AND

```
`timescale 1ns / 1ps


module AND8(
input [7:0] A,
input [7:0] B,
output reg [7:0] result
    );

always @(*) begin
result <= A&B;
end
endmodule
```

8-bit XOR

```
`timescale 1ns / 1ps

module EXOR(
input [7:0] A,
input [7:0] B,
output reg [7:0] result
    );

always @(A,B) begin
result <= A ^ B ;
end

endmodule
```

8-bit Circular Left Shift

```
`timescale 1ns / 1ps


module SHIFT(
input [7:0] A,
output reg [7:0] result
    );

always @(*) begin
result[7:1] <= A[6:0];
result[0] <= A[7];
end
endmodule
```

CO Mux:

```
`timescale 1ns / 1ps

module MUX1IN(
input addcout,
input shiftr0,
input [1:0] InsSel,
output reg CO
    );

always @(*) begin

case(InsSel)
0: CO <= 1'b0;
1: CO <= 1'b0;
2: CO <= addcout;
3: CO <= shiftr0;
endcase

end
endmodule
```

8-bit ALUOut MUX:

```
`timescale 1ns / 1ps


module MUX8IN(
input [7:0] andresult,exorresult,shiftresult,addresult,
input [1:0] InsSel,
output reg [7:0] ALUout
    );

always @(*) begin

case(InsSel)
0: ALUout <= andresult;
1: ALUout <= exorresult;
2: ALUout <= addresult;
3: ALUout <= shiftresult;
endcase

end
endmodule
```

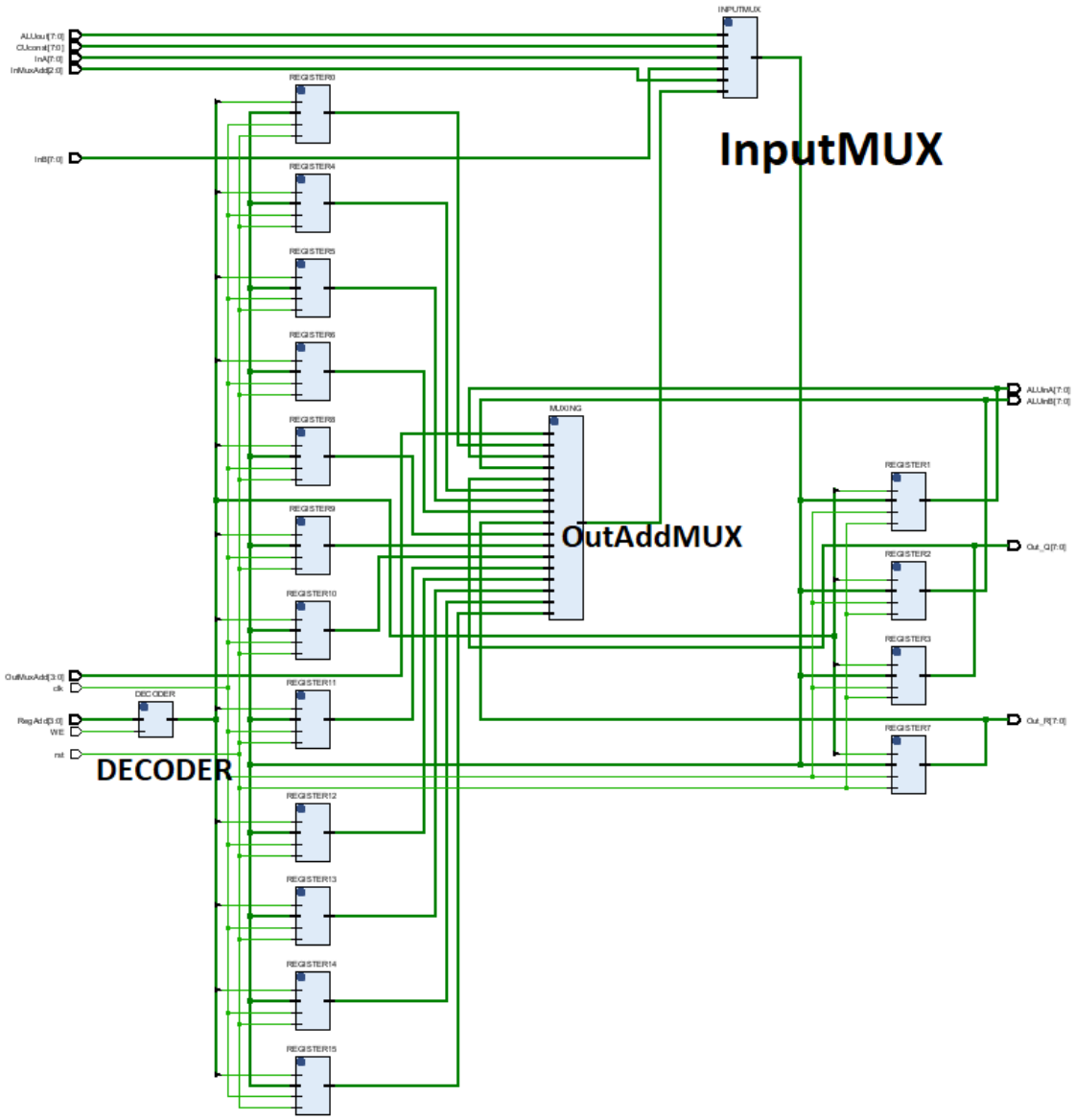
Top ALU module with Zero Comparator:

```
`timescale 1ns / 1ps

module ALU(
input [7:0] ALUinA,
input [7:0] ALUinB,
input [1:0] InsSel,
output [7:0] ALUout,
output CO,
output reg Z
    );
wire [7:0] addresult, exorresult,shiftresult,andresult;
wire cadder;

ADD ADDER (.A(ALUinA),.B(ALUinB),.result(addresult),.c(cadder));

AND8 AND (.A(ALUinA),.B(ALUinB),.result(andresult));

EXOR XOR (.A(ALUinA),.B(ALUinB),.result(exorresult));

SHIFT SHIFTER (.A(ALUinA),.result(shiftresult));

MUX8IN ALUOUT (.addresult(addresult),
                .exorresult(exorresult),
                .shiftresult(shiftresult),
                .andresult(andresult),
                .InsSel(InsSel),
                .ALUout(ALUout));

MUX1IN COUT (.addcout(cadder),
             .shiftr0(shiftresult[0]),
             .InsSel(InsSel),
             .CO(CO));

always @(*) begin

if(ALUout == 8'd0)
Z<=1;
else
Z<=0;

end
endmodule
```

RB(Register Block)

I designed this block excatly the same as given in Figure-2-RB.pdf.

Here is RTL schematic of my RB block:



There are 16 8-bit registers, 1 decoder, 1 mux selects input and 1 mux selects output register to go input.

Codes of my RB block:

Register Module:

```
`timescale 1ns / 1ps

module Register(
input [7:0] Rin,
input En,
input clk,
input rst,
output reg [7:0] Rout
    );


always @(posedge clk or posedge rst) begin
    if(rst)
        Rout <= 8'd0;
    else
    begin
        if(En)
            begin
            Rout <= Rin;
            end
    end
end
endmodule
```

Input Selection Mux:

```
module InMuxAdd_(
input [2:0] InMuxAdd,
input [7:0] InA,
input [7:0] InB,
input [7:0] CUConst,
input [7:0] ALUout,
input [7:0] RegOut,
output reg [7:0] Rin);

always @(*) begin
case(InMuxAdd)
0: Rin <= InA;
1: Rin <= InB;
2: Rin <= CUConst;
3: Rin <= ALUout;
4: Rin <= RegOut;
5: Rin <= RegOut;
6: Rin <= RegOut;
7: Rin <= RegOut;
endcase
end
endmodule
```

Output to Input Selection MUX:

```
module OutMuxAdd_ (
                        input [3:0] OutMuxAdd,
                        input [7:0] Rout0,
                        input [7:0] Rout1 ,
                        input [7:0] Rout2 ,
                        input [7:0] Rout3 ,
                        input [7:0] Rout4 ,
                        input [7:0] Rout5 ,
                        input [7:0] Rout6 ,
                        input [7:0] Rout7 ,
                        input [7:0] Rout8 ,
                        input [7:0] Rout9 ,
                        input [7:0] Rout10 ,
                        input [7:0] Rout11 ,
                        input [7:0] Rout12 ,
                        input [7:0] Rout13 ,
                        input [7:0] Rout14 ,
                        input [7:0] Rout15 ,
                        output reg [7:0] RegOut);


always @(*) begin

case(OutMuxAdd)
0: RegOut<= Rout0;
1: RegOut<= Rout1;
2: RegOut<= Rout2;
3: RegOut<= Rout3;
4: RegOut<= Rout4;
5: RegOut<= Rout5;
6: RegOut<= Rout6;
7: RegOut<= Rout7;
8: RegOut<= Rout8;
9: RegOut<= Rout9;
10: RegOut<= Rout10;
11: RegOut<= Rout11;
12: RegOut<= Rout12;
13: RegOut<= Rout13;
14: RegOut<= Rout14;
15: RegOut<= Rout15;
endcase
end
endmodule
```

Decoder:

```
module RegAddDecoder (
                        input WE,
                        input [3:0] RegAdd,
                        output reg [15:0] En);


always @(*) begin
if(WE)
begin
case(RegAdd)
0: En<= 16'b0000_0000_0000_0001;
1: En<= 16'b0000_0000_0000_0010;
2: En<= 16'b0000_0000_0000_0100;
3: En<= 16'b0000_0000_0000_1000;
4: En<= 16'b0000_0000_0001_0000;
5: En<= 16'b0000_0000_0010_0000;
6: En<= 16'b0000_0000_0100_0000;
7: En<= 16'b0000_0000_1000_0000;
8: En<= 16'b0000_0001_0000_0000;
9: En<= 16'b0000_0010_0000_0000;
10: En<= 16'b0000_0100_0000_0000;
11: En<= 16'b0000_1000_0000_0000;
12: En<= 16'b0001_0000_0000_0000;
13: En<= 16'b0010_0000_0000_0000;
14: En<= 16'b0100_0000_0000_0000;
15: En<= 16'b1000_0000_0000_0000;
endcase
end
end
endmodule
```

RB Module:

```
`timescale 1ns / 1ps


module RB(

input clk,
input rst,
//
input [2:0] InMuxAdd,
input [7:0] InA,
input [7:0] InB,
input [7:0] CUconst,
input [7:0] ALUout,
//
input WE,
input [3:0] RegAdd,
//
output [7:0] ALUinA,
output [7:0] ALUinB,
output [7:0] Out_Q,
output [7:0] Out_R,
input [3:0] OutMuxAdd
    );


wire [15:0] En;
wire [7:0] Rout [15:0];
wire [7:0] RegOut;
wire [7:0] Rin;

//  Input MUX
InMuxAdd_ INPUTMUX (
            .InMuxAdd(InMuxAdd),
            .InA(InA),
            .InB(InB),
            .CUConst(CUconst),
            .ALUout(ALUout),
            .RegOut(RegOut),
            .Rin(Rin) );

// Enable Decoder
RegAddDecoder DECODER (.WE(WE),.RegAdd(RegAdd),.En(En));


// REGISTERS:
Register REGISTER0 (.Rin(Rin),.En(En[0]),.clk(clk),.rst(rst),.Rout(Rout[0]));
Register REGISTER1 (.Rin(Rin),.En(En[1]),.clk(clk),.rst(rst),.Rout(Rout[1]));
Register REGISTER2 (.Rin(Rin),.En(En[2]),.clk(clk),.rst(rst),.Rout(Rout[2]));
Register REGISTER3 (.Rin(Rin),.En(En[3]),.clk(clk),.rst(rst),.Rout(Rout[3]));
Register REGISTER4 (.Rin(Rin),.En(En[4]),.clk(clk),.rst(rst),.Rout(Rout[4]));
Register REGISTER5 (.Rin(Rin),.En(En[5]),.clk(clk),.rst(rst),.Rout(Rout[5]));
Register REGISTER6 (.Rin(Rin),.En(En[6]),.clk(clk),.rst(rst),.Rout(Rout[6]));
Register REGISTER7 (.Rin(Rin),.En(En[7]),.clk(clk),.rst(rst),.Rout(Rout[7]));
Register REGISTER8 (.Rin(Rin),.En(En[8]),.clk(clk),.rst(rst),.Rout(Rout[8]));
Register REGISTER9 (.Rin(Rin),.En(En[9]),.clk(clk),.rst(rst),.Rout(Rout[9]));
Register REGISTER10 (.Rin(Rin),.En(En[10]),.clk(clk),.rst(rst),.Rout(Rout[10]));
Register REGISTER11 (.Rin(Rin),.En(En[11]),.clk(clk),.rst(rst),.Rout(Rout[11]));
Register REGISTER12 (.Rin(Rin),.En(En[12]),.clk(clk),.rst(rst),.Rout(Rout[12]));
Register REGISTER13 (.Rin(Rin),.En(En[13]),.clk(clk),.rst(rst),.Rout(Rout[13]));
```

```
Register REGISTER14 (.Rin(Rin),.En(En[14]),.clk(clk),.rst(rst),.Rout(Rout[14]));
Register REGISTER15 (.Rin(Rin),.En(En[15]),.clk(clk),.rst(rst),.Rout(Rout[15]));


// Out Mux

OutMuxAdd_ MUXING (.OutMuxAdd(OutMuxAdd),
            .Rout0(Rout[0]),
            .Rout1(Rout[1]),
            .Rout2(Rout[2]),
            .Rout3(Rout[3]),
            .Rout4(Rout[4]),
            .Rout5(Rout[5]),
            .Rout6(Rout[6]),
            .Rout7(Rout[7]),
            .Rout8(Rout[8]),
            .Rout9(Rout[9]),
            .Rout10(Rout[10]),
            .Rout11(Rout[11]),
            .Rout12(Rout[12]),
            .Rout13(Rout[13]),
            .Rout14(Rout[14]),
            .Rout15(Rout[15]),
            .RegOut(RegOut)
);

assign Out_Q = Rout[3];
assign Out_R = Rout[7];
assign ALUinA = Rout[1];
assign ALUinB = Rout[2];

endmodule
```

Control Unit of Non-Rest. Division

So there are 2 control units for each methods. First i'll explain the Non-restoring division algorithm control unit.

When reset is positive, it resets all case registers inside, as other modules resets everything.

If reset is negative,

First it catches a start signal (Should be 0_1_0 rectangle signal). After it catches this signal, Busy goes and starts to load needed values to the registers **(REGISTER_LOADS <= 1)**

CHECK STEP PART

If **STEP case register is <=7, do followings**. If STEP==8, the circuit will **stop, reset case registers and gives true outputs.**

REGISTER LOAD PART

1 - It compares **InA and 0 in ALU**. If it is zero, **Q and R goes to 0 directly and Busy goes 0. If it is not zero, loads InA to Reg3.**

2- It compares **InB and 0 in ALU.** If it is zero**, Q and R goes to FF** directly which **means ERROR** and **Busy goes 0**. If it is not zero**, loads InB to Reg4**.

3- It loads **8'b1111_1111 to REG5**. That will be needed in XOR operations and also it equals to -1 in signed which value i'll need to clear least significant bits in next steps.

4- It loads **8'b0000_0001 to REG6** which i'll need to get 2s complement of B.

5- It loads **8'b0000_0000 to REG7** which will be my remainder output.

6- It makes **twoscomplement signal 1 and REGISTER_LOAD signal 0**. This part is done and every needed value is stored in registers.

TWOSCOMPLEMENT PART

In this part, i'll get 2s complement of B to make substractions.

1- It selects **OutMuxAdd as 4** which is output of **REG4 (divisor),** selects **InMuxAdd as 4** which selects output value of output mux and selects **RegAdd as 1 which is ALUinA**.

2- It selects **OutMuxAdd as 5** which is **8'b1111_1111 to XOR** and selects **RegAdd as 2** which is **ALUinB**. InMuxAdd is already 4. And it selects **InsSel as 2'b01 which is XOR operation**

3- It selects **InMuxAdd as 3** which is **ALUResult (B XOR 1)** and selects **RegAdd as 1** which is **ALUinA**.

4- It selects **OutMuxAdd as 6** which is **8'd1** and **InMuxAdd as 4** which is OutMuxValue. Selects **RegAdd as 2 which is ALUinB** and **selects InsSel as 2'd2 which is ADD in ALU**.

5- It selects **InMuxAdd as 3 which is ALUResult (B XOR 1 +1 = 2s COMPLEMENT OF B)** and selects **RegAdd as 8**. **Now Reg[8] stores –B**. Sets **twoscomplement signal 0** and clean casetwocomplement register which controls the cases of this operation. **Sets shiftoperationA as 1**. This part is ended and **next part is shifting A**.

SHIFTOPERATIONA PART

1- Selects **OutMuxAdd as 7** which is **remainder**, selects **InMuxAdd as 4** which selects **OutMuxValue as input**. RegAdd selected as 1 which **is ALUinA**. **InsSel set to 2'b11** which is **shift**.

2- Selects **InMuxAdd as 3 which is ALUout**. Selects **RegAdd as 7 which is remainder**. Shifted remainder loaded in remainder now but our operation is circular shift. **So we should check the least significant bit which comes as CO in shift operation.**

3- In this part it **checks CO**. If CO is 0, it sets directly to shiftoperationQ to 1 and shiftoperationA and case register of shiftoperationA to 0. If CO is 1, it sets CLEARLSB to 1 and shiftoperationA and case register of shiftoperationA to 0.

(shiftoperationA means shiftoperationR)

CLEARLSB_A PART (CLEAR Least Significant Bit (R[0]) OF CIRCULAR SHIFTED R)

If the circular shifted bit of R is 1, **algorithm should clear that bit. It does that cleanse operation as R – 1'b1.**

1- It sets **OutMuxAdd as 7** which is **shifted R, InMuxAdd as 4** which is **OutMuxValue** and **RegAdd as 1** which is **ALUinA.**
2- It sets **OutMuxAdd as 5** which is **8'b1111_1111** (-1), **InMuxAdd as 4** which is **OutMuxValue** and **RegAdd as 2** which is **ALUinB**. **Sets InsSelection 2'b10 which is ADD.**
3- It sets **InMuxAdd as 3** which is **ALUout** and **RegAdd as 7** which is **R**. Now LSB of R is cleansed from 1'b1. Sets CLEAR_LSB 0, caseclearlsb 0 and shiftoperationQ as 1.

SHIFTOPERATIONQ PART (Shifts Q 1 bit left and stores Cout value to define A)

In this part, algorithm shifts Q with circular shift operation in ALU. Then it saves the COUT data as K which needed to implemented in two next step because we should clear LSB of this shifted Q too. So in this part, algorithm **just does the shifting operation of Q and saves the K data**.

1- Sets **OutMuxAdd as 3** which is **Q**. Sets **InMuxAdd as 4** which is **OutMuxValue**. Sets **RegAdd** as **1** which is **ALUinA**. And sets **InsSel as 2'b11**. Now R is shifted.

2- Sets **InMuxAdd as 3** which is **ALUOutput** and sets **RegAdd as 3** which is **Q**. Shifted Q saved.

3- Checks CO. If CO = 1 that means first bit of Q is 1 and it should clear that 1. So

It saves CO as K.

If CO = 1, sets CLEARQLSB control signal to 1.

If CO = 0, sets Aupdate(Means Rupdate) to add a value (0000_0000 or 0000_0001) to R.

Sets caseshiftq and shiftoperationQ to 0.

CLEARQLSB PART (It clears LSB of Q from 1 if it is.)

Algorithm does the same thing as CLEARLSB which clears LSB of R register.

1- Sets **OutMuxAdd as 3** which is **circular shifted Q**, **InMuxAdd as 4** which is **OutMuxValue** and sets **RegAdd as 1** which is **ALUinA**.

2- Sets **OutMuxAdd as 5** which is **8'b1111_1111**(-1), **RegAdd as 2** which is **ALUinB** and **InsSel as 2'b10** which is Add. ALU will make operation of Q-1.

3-Sets **InMuxAdd as 3** which is **ALUout(Q-1)**, **RegAdd as 3** which is address of **Q register**. So cleared Q is in Q register after this operation. Sets CLEARQLSB and caseCLEARQLSB 0 and sets Aupdate as 1.

Aupdate PART (Add Shifted Q bit to R.)

In this part algorithm adds the shifted Q bit to R. Then it shifts R and looks to CO which is MSB of R. If it is 0, algorithm does the R+B, if it is 1, algorithm does the R-B.

1- It sets **OutMuxAdd as 7** which is **R**, **InMuxAdd as 4** which is **OutMuxValue**, **RegAdd as 1** which is **ALUinA**.

2- It looks to K value which we stored 2 step before declares the value of shifted Q bit.

If K == 1 , sets **OutMuxAdd as 6** which is 8**'b0000_0001**, **InMuxAdd as 4** which is **OutMuxValue** and **InsSel as 2'b10** which is **ADD** operation.

If K==0, sets **OutMuxAdd as 13** which is **8'b0000_0000**, **InMuxAdd as 4** which is **OutMuxValue** and **InsSel as 2'b10** which is **ADD** operation

3- Set **InMuxAdd as 3** which is **ALUResult** and **RegAdd as 7** which is **R**. Now R updated.

4- Set **OutMuxAdd as 7** which is **updated R**, **InMuxAdd as 4** which is **OutMuxValue** and **InsSel as 2'b11** which is **circular shifting.**

5- Checks CO. **If CO == 1, sets ADD 1**. **If CO==0, sets SUBSTRACT 1**. Sets Aupdate and caseAupdate as 0.

SUBSTRACT PART (R = R-B)

1- Sets **OutMuxAdd as 7** which is **R**, **InMuxAdd as 4** which is **OutMuxValue**, **RegAdd as 1** which is **ALUinA**.

2- Sets **OutMuxAdd as 8** which is **–B**, **InMuxAdd as 4** which is **OutMuxValue**, **RegAdd as 2** which is **ALUinB**, **InsSel as 2'b10** which is **ADD**.

3- Sets **InMuxAdd as 3** which is **ALUresult(R-B),** RegAdd as 7 which is R. (R = R-B)

4- Sets Qupdate as 1 and SUBSTRACT and caseSUBSTRACT as 0.

ADD PART (R = R+B)

1- Sets **OutMuxAdd as 7** which is **R**, **InMuxAdd as 4** which is **OutMuxValue**, **RegAdd as 1** which is **ALUinA**.

2- Sets **OutMuxAdd as 4** which is **B, InMuxAdd as 4** which is **OutMuxValue**, **RegAdd as 2** which is **ALUinB**, **InsSel as 2'b10** which is **ADD**.

3- Sets **InMuxAdd as 3** which is **ALUresult(R+B)**, **RegAdd as 7** which is **R. (R = R+B)**

4- Sets Qupdate as 1 and ADD and caseADD as 0.

QUPDATE PART (CHECK MSB of R and UPDATE Q[0])

1- Set **OutMuxAdd as 7** which is **R**, **InMuxAdd as 4** which is **OutMuxValue**, **RegAdd as 1** which is **ALUinA**. Set **InsSel as 2'11** which is **circular shift**.

2- Check CO.

> If CO == 0, set **OutMuxAdd as 3** which is **Q**, **InMuxAdd as 4** which is **OutMuxValue**, **RegAdd as 1** which is **ALUinA**. caseQupdate <=caseQupdate+1.
>
> If CO==1, set Qupdate and caseQupdate as 0, **set STEP <= STEP+1**, and set **shiftoperationA as 1**. That means a loop is done. The circuit will check STEP first then goes to shiftoperationA or it stops.

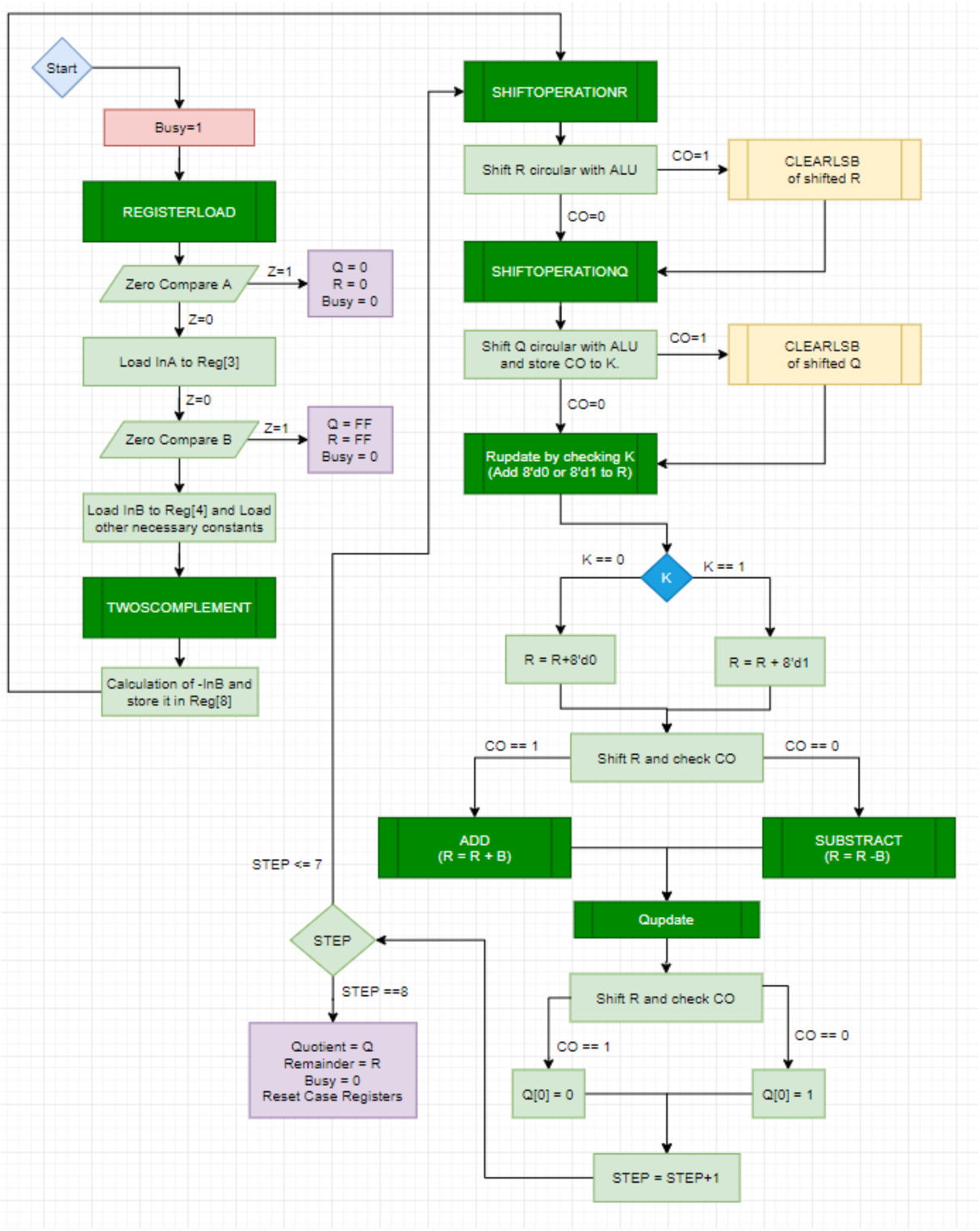
3- Set **OutMuxAdd as 6** which is **8'b0000_0001(1), RegAdd as 2** which is **ALUinB**, **InsSel as 2'b10** which is **ADD**. That means if MSB of R is 0, set Q[0] 1'b1.

4- Set **InMuxAdd as 3** which is **ALUresult**, **RegAdd as 3** to **update Q with new value (Q = Q+1'b1)**. Set Qupdate and caseQupdate as 0, shiftoperationA as 1 and STEP<=STEP+1. That means a loop is done. The circuit will check STEP first then goes to shiftoperationA or it stops.

These are the all of the steps of Non-Restoring Division Algorithm Control Unit.

In next page, i drew a block diagram of this control unit which shows steps clearly.

Here is block diagram of Non-Restoring Division Algorithm which i designed:



I named this project as Non-Restoring Division in MY ZIP.

CU of Non-Restoring Algorithm:

```
`timescale 1ns / 1ps



module CU(
    input clk,
    input rst,
    input Start,
    output reg Busy,
    output reg [7:0] CUconst,
    output reg [2:0] InMuxAdd,
    output reg [3:0] OutMuxAdd,
    output reg [3:0] RegAdd,
    output reg WE,
    output reg [1:0] InsSel,
    input CO,
    input Z
    );

// STATE REGISTERS
reg K =0;
reg ADD = 0;
reg [4:0] STEP = 5'd0;
reg CLEARQLSB=0;
reg [2:0] caseCLEARQLSB=3'd0;
reg SUBSTRACT = 0;
reg twoscomplement = 0;
reg CLEAR_LSB= 0;
reg REGISTER_LOADS = 0;
reg shiftoperationQ = 0;
reg startprocess = 0;
reg shiftoperationA =0;
reg Aupdate = 0;
reg Qupdate = 0;
reg [2:0] caseADD = 3'd0;
reg [2:0] caseQupdate = 3'd0;
reg [2:0] caseSUBSTRACT =3'd0;
reg [2:0] caseAupdate = 3'd0;
reg [2:0] caseshiftQ = 3'd0;
reg [1:0] caseclearlsb = 2'd0;
reg [5:0] shiftingA = 6'd0;
reg [6:0] casereg = 7'd0;
reg [2:0] shiftB =2'd0;
reg [2:0] casetwoscomplement= 3'd0;
reg [3:0] casezero=4'd0;

always @(posedge clk or posedge rst) begin
    if(rst)
        begin
            Busy<=0;
            CUconst<=0;
            WE<=0;
            startprocess = 0;
        end
    else if(STEP <=7)
        begin
                if(Start & !startprocess)
                begin
                startprocess <=1;
                Busy<=1;
                end
                if(startprocess)
                begin
                WE <=1;
                REGISTER_LOADS<=1;
                end

                if(REGISTER_LOADS)
                begin
                    case(casereg)
                    0:  begin
                            case(casezero)
                            0: begin
                                InMuxAdd<=3'd0;
                                RegAdd<=4'd1;
                                casezero<=casezero+1;
                               end
                            1:  begin
                                InsSel<=2'd3;
                                casezero<=casezero+1;
                                end
                            2: begin
                                if(Z)
                                    begin
                                        CUconst<=8'd0;
```

```
                        InMuxAdd<=3'd2;
                        RegAdd<=4'd3;
                        casezero<=casezero+1;
                    end
                else
                    begin
                       InMuxAdd<=3'd0;
                       RegAdd <= 4'd3;       // REG3'e b□□en (Q'YA D®ğECEK)
                       casereg<=casereg+1;
                       casezero<=0;
                    end
               end
            3: begin
                RegAdd<=4'd7;
                casezero<=casezero+1;
               end
            4:begin
                  K <=0;
                  ADD <= 0;
                  STEP <= 5'd0;
                  CLEARQLSB<=0;
                  caseCLEARQLSB<=3'd0;
                  SUBSTRACT <= 0;
                  twoscomplement <= 0;
                  CLEAR_LSB<= 0;
                  REGISTER_LOADS <= 0;
                  shiftoperationQ <= 0;
                  startprocess <= 0;
                  shiftoperationA <=0;
                  Aupdate <= 0;
                  Qupdate <= 0;
                  caseADD <= 3'd0;
                  caseQupdate <= 3'd0;
                  caseSUBSTRACT <=3'd0;
                  caseAupdate <= 3'd0;
                  caseshiftQ <= 3'd0;
                  caseclearlsb <= 2'd0;
                  shiftingA <= 6'd0;
                  casereg <= 7'd0;
                  shiftB <=2'd0;
                  casetwoscomplement<= 3'd0;
                  casezero<=0;
                  Busy<=0;
              end
            endcase

        end
    1:  begin
        case(casezero)
            0: begin
                InMuxAdd<=3'd1;
                RegAdd<=4'd1;
                casezero<=casezero+1;
               end
            1:  begin
                InsSel<=2'd3;
                casezero<=casezero+1;
                end
            2: begin
                if(Z)
                    begin
                        CUconst<=8'b1111_1111;
                        InMuxAdd<=3'd2;
                        RegAdd<=4'd3;
                        casezero<=casezero+1;
                    end
                else
                    begin
                       InMuxAdd<=3'd1;
                       RegAdd <= 4'd4;       // REG3'e b□□en (Q'YA D®ğECEK)
                       casereg<=casereg+1;
                       casezero<=0;
                    end
               end
            3: begin
                RegAdd<=4'd7;
                casezero<=casezero+1;
               end
            4:begin
             casezero<=0;
             K <=0;
             ADD <= 0;
             STEP <= 5'd0;
             CLEARQLSB<=0;
             caseCLEARQLSB<=3'd0;
             SUBSTRACT <= 0;
             twoscomplement <= 0;
```

```
                    CLEAR_LSB<= 0;
                    REGISTER_LOADS <= 0;
                    shiftoperationQ <= 0;
                    startprocess <= 0;
                    shiftoperationA <=0;
                    Aupdate <= 0;
                    Qupdate <= 0;
                    caseADD <= 3'd0;
                    caseQupdate <= 3'd0;
                    caseSUBSTRACT <=3'd0;
                    caseAupdate <= 3'd0;
                    caseshiftQ <= 3'd0;
                    caseclearlsb <= 2'd0;
                    shiftingA <= 6'd0;
                    casereg <= 7'd0;
                    shiftB <=2'd0;
                    casetwoscomplement<= 3'd0;
                    Busy<=0;
                     end
                   endcase

               end
           2:  begin                    // REG5'e XOR i穗 full 1 (Ayn zamanda -1)
                 CUconst<=8'b1111_1111;
                 InMuxAdd <= 3'd2;
                 RegAdd <= 4'd5;
                 casereg <= casereg+1'b1;
               end
           3:  begin                    // REG6ya 2s complement i穗 1
                  CUconst <= 8'b0000_0001;
                  InMuxAdd<=3'd2;
                  RegAdd <= 4'd6;
                  casereg <= casereg+1'b1;
               end
            4: begin
                   CUconst<= 8'd0;          // REG7'ye zerinde ilem yapmak i穗 full 0 (A: remainder)
                   InMuxAdd <= 3'd2;
                   RegAdd <= 4'd7;
                   casereg <= casereg+1'b1;
               end
            5: begin
               twoscomplement<=1;
               casereg <= casereg+1;
               end

           default: ;

           endcase
           end

       if(twoscomplement)
           begin
               case(casetwoscomplement)

                    0:begin
                         OutMuxAdd <= 4'd4;
                         InMuxAdd <= 4'd4;
                         RegAdd <= 4'd1;          // B□□ A'ya yazld
                         casetwoscomplement<=casetwoscomplement+1;
                      end
                   1:begin
                         OutMuxAdd<=4'd5;     //full 1 XOR i穗
                         InMuxAdd<=4'd4;
                         RegAdd <= 4'd2;
                         casetwoscomplement<=casetwoscomplement+1;
                         InsSel<=2'd1;        // XOR OPERATION
                     end
                   2:begin
                         InMuxAdd<=4'd3;      // ALU result se穣di
                         RegAdd<=4'd1;        //B□□'in XORu ALUinA'ya yazld
                         casetwoscomplement<=casetwoscomplement+1;
                     end
                   3:begin
                         OutMuxAdd<=4'd6;
                         InMuxAdd<=4'd4;
                         RegAdd<=4'd2;        // 0000_0001 B'ye yazld
                         InsSel<=2'd2;
                         casetwoscomplement<=casetwoscomplement+1;
                     end
                   4:begin
                         InMuxAdd<=4'd3; //ALUresult se穣di
                         RegAdd<=4'd8; // BJEN2 ـد's COMPLEMENTىARTIK REG10DA
                         shiftoperationA<=1;
                         casetwoscomplement<=0;
```

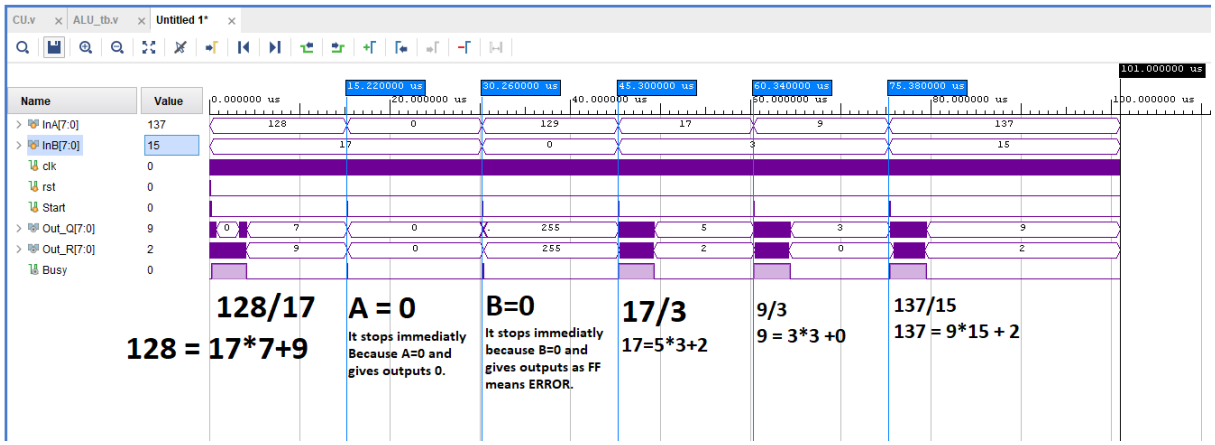
```
                            twoscomplement<=0;
                        end
                    endcase
                end
// SHIFTING A
            if(shiftoperationA)
              begin
                  case(shiftingA)
                  0:begin
                    OutMuxAdd <= 4'd7;      // A SHQTLEMEK óZ A REGISTERI SEəLDY
                    InMuxAdd <=3'd4;        // REGOUTPUT ZPUT OLARAK ALINDI
                    RegAdd<= 4'd1;          // ALUinA'YA YAZILDI
                    shiftingA <= shiftingA +1;
                    InsSel <= 2'd3;
                    end
                  1:begin
                              // SHIFT INSTRUCTION SEəLDY
                        shiftingA <= shiftingA + 1;
                    end
                  2:begin
                      InMuxAdd <= 3'd3;     // Shiftlenen A kaydedilmek i穗 ALUout se穡di
                      RegAdd <= 4'd7;       // A registerna kaydedildi.
                          if(CO)
                              begin
                                  CLEAR_LSB<=1;
                                  shiftoperationA<=0;
                                  shiftoperationQ<=0;
                                  shiftingA<=0;
                              end
                          else
                               begin
                                    CLEAR_LSB<=0;
                                    shiftoperationA<=0;
                                    shiftoperationQ<=1;
                                    shiftingA <= 0;
                               end
                    end
                  endcase
                  end

            if(CLEAR_LSB)
            begin
                case(caseclearlsb)
                0: caseclearlsb<=caseclearlsb+1;
                1:  begin
                        OutMuxAdd <= 4'd7;  // Shiftlenen A se穡di
                        InMuxAdd <= 3'd4;
                        RegAdd <= 4'd1;     //Shiftlenen A AluinA'ya yolland
                        caseclearlsb<=caseclearlsb+1;
                     end
                2:  begin
                        OutMuxAdd <= 4'd5;  // -1 se穡di
                        InMuxAdd <= 3'd4;
                        RegAdd <= 4'd2;     // -1 ALUinB'ye yolland
                        caseclearlsb <= caseclearlsb +1;
                        InsSel <=  2'd2;    // Add instruction se穡di
                    end
                3:  begin
                        InMuxAdd <= 3'd3;   // ALUdan gelen A-1 sonucu se穡di
                        RegAdd <= 4'd7; // A, ALUinA 'ya yazld.
                        caseclearlsb <=0;
                        CLEAR_LSB<=0;
                        shiftoperationQ<=1;
                    end
                endcase
            end

            if(shiftoperationQ)
            begin
                case(caseshiftQ)
                0:  begin
                        OutMuxAdd <=4'd3;
                        InMuxAdd <=3'd4;
                        RegAdd <= 4'd1;         // Q Shfitlemek i穗 A'ya alnd ve shiftlendi
                        InsSel <= 2'd3;
                        caseshiftQ<=caseshiftQ+1;
                    end
                1:  begin
                        InMuxAdd <=3'd3;
                        RegAdd <=3'd3;          // Shiftlenen Q, Q'ya yazld
                        caseshiftQ<=caseshiftQ+1;
                    end
                2:  begin
                        K <= CO;
```

```
                            if(CO)
                                CLEARQLSB<=1;
                            else
                                Aupdate<=1;
                            caseshiftQ<=0;
                            shiftoperationQ<=0;
                        end
                    endcase
                end

                     if(CLEARQLSB)
                    begin
                    case(caseCLEARQLSB)
                    0:begin
                        OutMuxAdd<=4'd3;
                        InMuxAdd<=4'd4;
                        RegAdd<=4'd1;
                        caseCLEARQLSB <=caseCLEARQLSB+1;
                      end
                    1:begin
                        OutMuxAdd<=4'd5;
                        RegAdd<=4'd2;
                        InsSel<=2'd2;
                        caseCLEARQLSB <=caseCLEARQLSB+1;
                      end
                    2:begin
                        InMuxAdd<=3'd3;
                        RegAdd<=4'd3;
                        caseCLEARQLSB <=0;
                        CLEARQLSB<=0;
                        Aupdate<=1;
                      end

                    endcase
                    end

                if(Aupdate)
                begin
                    case(caseAupdate)
                    0:  begin
                            OutMuxAdd<=4'd7;
                            InMuxAdd<=3'd4;
                            RegAdd<=4'd1;
                            caseAupdate<=caseAupdate+1;
                        end
                    1:  begin
                        if(K==1)
                            begin
                            OutMuxAdd<=4'd6;
                            RegAdd<=4'd2;
                            InsSel<=2'd2;
                            end
                        else
                            begin
                                OutMuxAdd<=4'd13;
                                RegAdd<=4'd2;
                                InsSel<=2'd2;
                            end
                        caseAupdate<=caseAupdate+1;
                        end
                    2:  begin
                            InMuxAdd<=3'd3;
                            RegAdd<=4'd7;           // A UPDATED
                            caseAupdate<=caseAupdate+1;
//                            caseAupdate<=0;
//                            Aupdate<=0;
                        end
                    3: begin
                            OutMuxAdd<=4'd7;
                            InMuxAdd<=3'd4;
                            InsSel<= 2'd3;
                            caseAupdate<=caseAupdate+1;
                        end
                    4:begin
                      if(CO)
                        ADD<=1;
                      else
                        SUBSTRACT<=1;
                      caseAupdate<=0;
                      Aupdate<=0;
                      end
                    endcase
                end


                if(SUBSTRACT)
                    begin
                        case(caseSUBSTRACT)
```

```
                0:begin
                    OutMuxAdd<=4'd7;
                    InMuxAdd<=3'd4;
                    RegAdd<=4'd1;
                    caseSUBSTRACT<=caseSUBSTRACT+1;
                  end
                1:begin
                    OutMuxAdd<=4'd8;
                    RegAdd<=4'd2;
                    InsSel<=2'd2;
                    caseSUBSTRACT<=caseSUBSTRACT+1;
                  end
                2:begin
                    InMuxAdd<=3'd3;
                    RegAdd<=4'd7;
                    caseSUBSTRACT<=caseSUBSTRACT+1;
                  end
                3:begin
                    caseSUBSTRACT<=0;
                    SUBSTRACT<=0;
                    Qupdate<=1;
                  end
                endcase
            end


            if(ADD)
            begin
                case(caseADD)
                0:begin
                    OutMuxAdd<=4'd7;
                    InMuxAdd<=3'd4;
                    RegAdd<=4'd1;
                    caseADD<=caseADD+1;
                  end
                1:begin
                    OutMuxAdd<=4'd4;
                    RegAdd<=4'd2;
                    InsSel<=2'd2;
                    caseADD<=caseADD+1;
                  end
                2:begin
                    InMuxAdd<=3'd3;
                    RegAdd<=4'd7;
                    caseADD<=caseADD+1;
                  end
                3:begin
                    caseADD<=0;
                    ADD<=0;
                    Qupdate<=1;
                  end
                endcase
            end




        if(Qupdate)
            begin
                case(caseQupdate)
                0:caseQupdate<=caseQupdate+1;
                1:begin
                    OutMuxAdd<=4'd7;
                    InMuxAdd<=3'd4;
                    RegAdd<=4'd1;
                    caseQupdate<=caseQupdate+1;
                  end
                2:begin
                    InsSel<= 2'd3;      // A shifted
                    caseQupdate<=caseQupdate+1;
                    end
                3:begin
                    if(!CO)
                        begin
                            OutMuxAdd<=4'd3;
                            InMuxAdd<=3'd4;
                            RegAdd<=4'd1;
                            caseQupdate<=caseQupdate+1;
                        end
                    else
                        begin
                        caseQupdate<=0;
                        Qupdate<=0;
                        shiftoperationA<=1;
                        STEP <= STEP+1;
```

```
                                end
                            end
                        4:  begin
                                OutMuxAdd<=4'd6;
                                RegAdd<=4'd2;
                                InsSel <= 2'd2;
                                caseQupdate<=caseQupdate+1;
                            end
                        5:  begin
                                InMuxAdd <=3'd3;
                                RegAdd<=4'd3;
                                caseQupdate<=0;
                                Qupdate<=0;
                                shiftoperationA<=1;
                                STEP <= STEP+1;
                            end
                        endcase
                    end
                        //
                    end
        else if(STEP == 8)
        begin
        K <=0;
        ADD <= 0;
        STEP <= 5'd0;
        CLEARQLSB<=0;
        caseCLEARQLSB<=3'd0;
        SUBSTRACT <= 0;
        twoscomplement <= 0;
        CLEAR_LSB<= 0;
        REGISTER_LOADS <= 0;
        shiftoperationQ <= 0;
        startprocess <= 0;
        shiftoperationA <=0;
        Aupdate <= 0;
        Qupdate <= 0;
        caseADD <= 3'd0;
        caseQupdate <= 3'd0;
        caseSUBSTRACT <=3'd0;
        caseAupdate <= 3'd0;
        caseshiftQ <= 3'd0;
        caseclearlsb <= 2'd0;
        shiftingA <= 6'd0;
        casereg <= 7'd0;
        shiftB <=2'd0;
        casetwoscomplement<= 3'd0;
        Busy<=0;
        end


end


endmodule
```

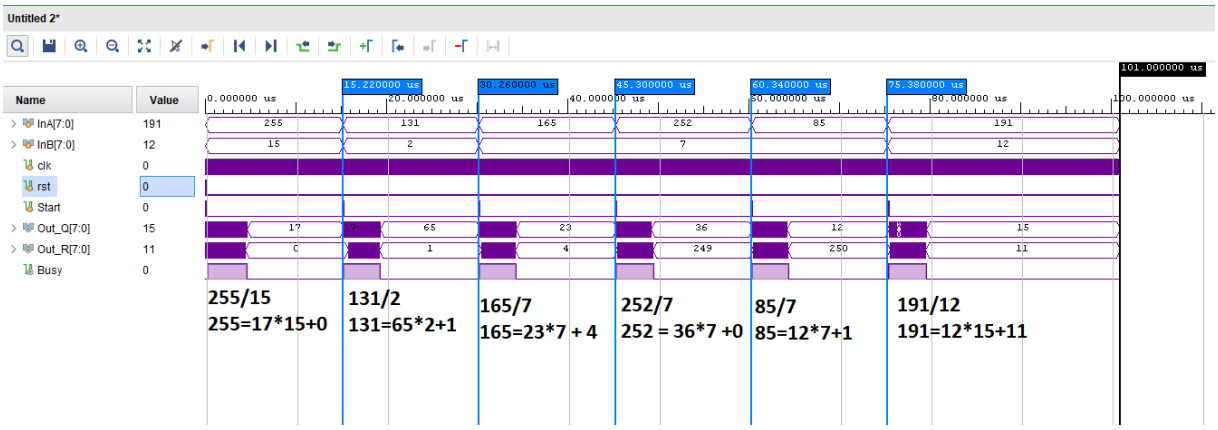
Simulation outputs, Non-Restoring Division Algorithm:



As we can see, when a start signal comes circuit starts division operation and gives Busy output HIGH. When the operation ends, gives the right outputs, Busy goes 0 and can be usable again and again.

1- 128/17 given,outputs are Q = 7 and R=9. >> 128 = 7*17+9 is correct.
2- 0/17 given,outputs are Q=0 and R=0. >> 0 = 17*0 + 0 is correct
3- 129/0 is given,outputs are Q=FF and R=FF >> If Q and R FF, that means error. Correct.
4- 17/3 is given,outputs are Q=5 and R=2 >> 17 = 3*5 + 2 is correct.
5- 9/3 is given,outputs are Q=3 and R=0 >> 9=3*3 +0 is correct.
6- 137/15 is given, outputs are Q=9 and R=2 >>137=15*9+2 is correct.

Another simulation outputs of Non-Restoring Division Algorithm:



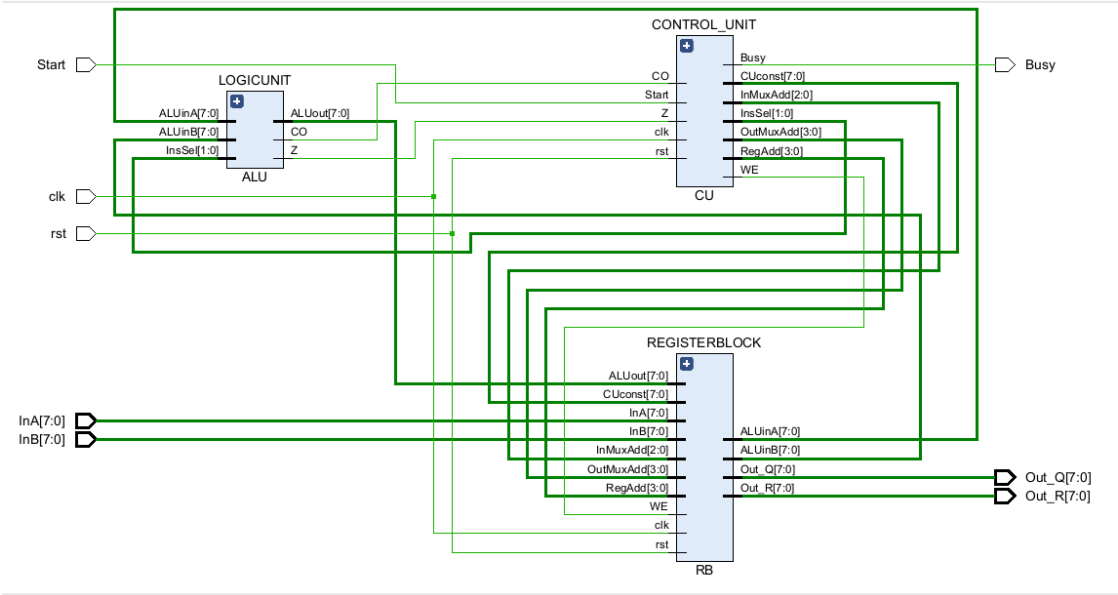
As we can see, when a start signal comes circuit starts division operation and gives Busy output HIGH. When the operation ends, gives the right outputs, Busy goes 0 and can be usable again and again. But some outputs are not correct in this simulation.

1- 255/15 given,outputs Q = 17 and R=0. >> 255 = 17*15+0 is correct.
2- 131/2 given,outputs Q=65 and R=1. >> 131 = 65*2 + 1 is correct.
3- 165/7 is given,outputs Q=23 and R=4 >> 165 = 23*7 + 4 is correct.
4- 252/7 is given,outputs Q=36 and R=249 >> 252 = 7*36 + 0. R isn't correct. Q is okay.
5- 85/7 is given,outputs Q=12 and R=250 >> 85 = 12*7+1. R isn't correct. Q is okay.
6- 191/12 is given, outputs Q=15 and R=11 >>191=15*12+11 is correct.

So in second simulation part, i see that Quotient output of the circuit works fine but remainder gives faulty outputs sometimes. I probably think that because of the CLEARLSB and shifting operations at first step. That is because of my ALU limited to shift as circular. If it was logical shift, everything would be easier and probably every output will become true.

Any other state can be execute with extracting Non-Restoring Divison Algorithm Project and editing testbench.

RTL Schematic TOPMODULE of Non-Restoring Division Algorithm:



Testbench is included in .ZIP file, but in experiment sheet it says that pdf should includes testbench. So here is testbench of non-restoring division algorithm:

```
`timescale 1ns / 1ps

module TOP_tb;

reg [7:0] InA;
reg [7:0] InB;
reg clk = 0;
reg rst=0;
reg Start =0;
wire [7:0] Out_Q,Out_R;
wire Busy;
TOP DUT     (.clk(clk),
             .rst(rst),
             .Start(Start),
             .Out_Q(Out_Q),
             .Out_R(Out_R),
             .Busy(Busy),
             .InA(InA),
             .InB(InB));

always begin
#10;
clk <= ~clk;
end
```

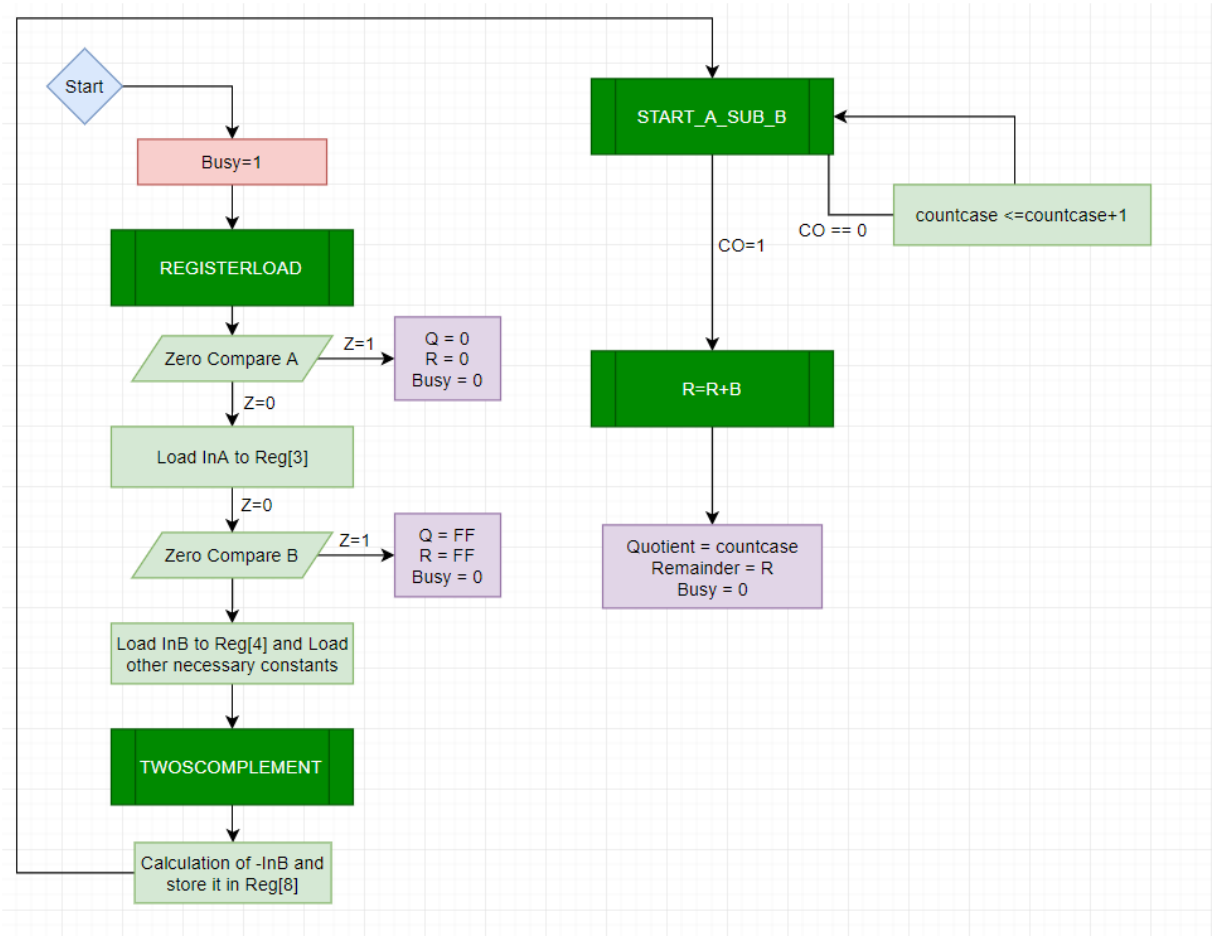
```
initial begin
rst=1;
InA = 8'b1111_1111;
InB = 8'b0000_1111;
#100;
rst=0;
#100;
Start=1;
#20;
Start=0;
#15000;
InA = 8'b1000_0011;
InB = 8'b0000_0010;
#20;
Start=1;
#20;
Start=0;
#15000;
InA = 8'b1010_0101;
InB = 8'b0000_0111;
#20;
Start=1;
#20;
Start=0;
#15000;
InA = 8'b1111_1100;
InB = 8'b0000_0111;
#20;
Start=1;
#20;
Start=0;
#15000;
InA = 8'b0101_0101;
InB = 8'b0000_0111;
#20;
Start=1;
#20;
Start=0;
#15000;
InA = 8'b1011_1111;
InB = 8'b0000_1100;
#20;
Start=1;
#20;
Start=0;
end
endmodule
```

Top Module code of Non-Restoring Division algorithm:

```
module TOP(
    input clk,
    input rst,
    input Start,
    input [7:0] InA,
    input [7:0] InB,
    output [7:0] Out_Q,
    output [7:0] Out_R,
    output Busy
    );

wire [7:0] CUconst;
wire [2:0] InMuxAdd;
wire [3:0] OutMuxAdd;
wire [3:0] RegAdd;
wire WE;
wire [1:0] InsSel;
wire CO;
wire Z;
wire [7:0] ALUinA;
wire [7:0] ALUinB;
wire [7:0] ALUout;

CU CONTROL_UNIT (.clk(clk),
                 .rst(rst),
                 .Start(Start),
                 .Busy(Busy),
                 .CUconst(CUconst),
                 .InMuxAdd(InMuxAdd),
                 .OutMuxAdd(OutMuxAdd),
                 .RegAdd(RegAdd),
                 .WE(WE),
                 .InsSel(InsSel),
                 .CO(CO),
                 .Z(Z));

RB REGISTERBLOCK (.clk(clk),
                  .rst(rst),
                  .InMuxAdd(InMuxAdd),
                  .InA(InA),
                  .InB(InB),
                  .CUconst(CUconst),
                  .ALUout(ALUout),
                  .WE(WE),
                  .RegAdd(RegAdd),
                  .ALUinA(ALUinA),
                  .ALUinB(ALUinB),
                  .Out_Q(Out_Q),
                  .Out_R(Out_R),
                  .OutMuxAdd(OutMuxAdd));

ALU LOGICUNIT ( .ALUinA(ALUinA),
                .ALUinB(ALUinB),
                .InsSel(InsSel),
                .ALUout(ALUout),
                .CO(CO),
                .Z(Z)
                );
endmodule
```

My Simple Division Algorithm

Here is block diagram of my simple division algorithm:



REGISTER LOAD PART

1 - It compares **InA and 0 in ALU**. If it is zero, **Q and R goes to 0 directly and Busy goes 0. If it is not zero, loads InA to Reg3.**

2- It compares **InB and 0 in ALU.** If it is zero**, Q and R goes to FF** directly which **means ERROR** and **Busy goes 0**. If it is not zero**, loads InB to Reg4**.

3- It loads **8'b1111_1111 to REG5**. That will be needed in XOR operations and also it equals to -1 in signed which value i'll need to clear least significant bits in next steps.

4- It loads **8'b0000_0001 to REG6** which i'll need to get 2s complement of B.

5- It loads **8'b0000_0000 to REG7** which will be my remainder output.

6- It makes **twoscomplement signal 1 and REGISTER_LOAD signal 0**. This part is done and every needed value is stored in registers.

TWOSCOMPLEMENT PART

In this part, i'll get 2s complement of B to make substractions.

1- It selects **OutMuxAdd as 4** which is output of **REG4 (divisor),** selects **InMuxAdd as 4** which selects output value of output mux and selects **RegAdd as 1 which is ALUinA**.

2- It selects **OutMuxAdd as 5** which is **8'b1111_1111 to XOR** and selects **RegAdd as 2** which is **ALUinB**. InMuxAdd is already 4. And it selects **InsSel as 2'b01 which is XOR operation**

3- It selects **InMuxAdd as 3** which is **ALUResult (B XOR 1)** and selects **RegAdd as 1** which is **ALUinA**.

4- It selects **OutMuxAdd as 6** which is **8'd1** and **InMuxAdd as 4** which is OutMuxValue. Selects **RegAdd as 2 which is ALUinB** and **selects InsSel as 2'd2 which is ADD in ALU**.

5- It selects **InMuxAdd as 3 which is ALUResult (B XOR 1 +1 = 2s COMPLEMENT OF B)** and selects **RegAdd as 8**. **Now Reg[8] stores –B**. Sets **twoscomplement signal 0** and clean casetwocomplement register which controls the cases of this operation. **Sets START_A_SUB_B as 1**. This part is ended and **next part is START_A_SUB_B.**

START_A_SUB_B PART

1- It sets **OutMuxAdd as 3** which is **A**, **InMuxAdd as 4** which is **OutMuxValue**, **RegAdd as 1** which is **ALUinA**. Now ALUinA has value A.

2- It sets **OutMuxAdd as 8** which is **–B**, **InMuxAdd as 4** which is **OutMuxValue**, **RegAdd as 2** which is **ALUinB**. Now ALUinB has value –B . Sets InsSel as 2'b10 which is ADD. (A – B)

3- Checks CO.

> If CO == 0, that means the remainder of substraction is **positive**. Sets **InMuxAdd as 3** which is **ALUresult**, **RegAdd as 1** which is **ALUinA**. **InsSel is 2'b10 already** so it will make the operation **(R-B).** Sets **countcase <= countcase+1 which counts the substraction number**. Sets **casesubab<=3'b1 which makes it to go step 2 in this PDF**.
>
> If CO==1, that means remainder of substraction is **negative**. Sets **InMuxAdd as 3** which is **ALUresult**, **RegAdd as 1** which is **ALUinA**. Sets **START_A_SUB_B 0** and **case register remainder 1.**

REMAINDER PART

Now we have a negative number which is the result of all substraction and it is on ALUinA. We need to just add it a B to get true remainder result.

1- It sets **CUconstant as countcase** which is the **quotient**, **InMuxAdd as 2** which is **CUconstant**, **RegAdd as 0** which is the **quotient output** register.
2- It sets **OutMuxAdd as 4** which is **B**, **InMuxAdd as 4** which is **OutMuxValue**, **RegAdd as 2** which is **ALUinB**. **(R = R + B)**
3- It sets **InMuxAdd as 3** which is **ALUout**, **RegAdd as 14** which is **output of Remainder.** Then it clears all case registers and sets Busy to 0.

This project has almost same files as the Non-Restoring Algorithm. A couple lines in RB is different. Even testbenches are the same. Just CU module and RB module's couple line is different. So i'll just declare this modules again. RTL schematic of ALU,RB and TopModule is same as before because nothing is different.

Code Of RB:

```
`timescale 1ns / 1ps



module CU(
    input clk,
    input rst,
    input Start,
    output reg Busy,
    output reg [7:0] CUconst,
    output reg [2:0] InMuxAdd,
    output reg [3:0] OutMuxAdd,
    output reg [3:0] RegAdd,
    output reg WE,
    output reg [1:0] InsSel,
    input CO,
    input Z
    );
reg remainder=0;
reg [3:0] caseremainder = 4'd0;
reg [7:0] countcase=8'd0;
reg START_A_SUB_B;
reg [3:0] casesubab = 4'd0;
reg [4:0] STEP = 5'd0;
reg twoscomplement = 0;
reg REGISTER_LOADS = 0;
reg startprocess = 0;
reg [6:0] casereg = 7'd0;
reg [2:0] casetwoscomplement= 3'd0;
reg zerocompare = 0;
reg [3:0] casezero = 4'd0;
always @(posedge clk or posedge rst) begin
    if(rst)
        begin
            Busy<=0;
            CUconst<=0;
            WE<=0;
            startprocess <= 0;
            remainder <=0;
            caseremainder <=0;
            countcase <=0;
            START_A_SUB_B <=0;
            casesubab<=0;
            STEP <=0;
            twoscomplement<=0;
            REGISTER_LOADS<=0;
            startprocess <=0;
            casereg <=0;
            casetwoscomplement<=0;
        end
    else if(STEP <=7)
        begin
                if(Start & !startprocess)
                begin
                startprocess <=1;
                Busy<=1;
                end
                if(startprocess)
                begin
                WE <=1;
                REGISTER_LOADS<=1;
                end
                if(REGISTER_LOADS)
                begin
                    case(casereg)
                    0:  begin
                            case(casezero)
                            0: begin
                                InMuxAdd<=3'd0;
                                RegAdd<=4'd1;
                                casezero<=casezero+1;
                               end
                            1:  begin
                                InsSel<=2'd3;
                                casezero<=casezero+1;
                                end
```

```
                    2: begin
                        if(Z)
                            begin
                                CUconst<=8'd0;
                                InMuxAdd<=3'd2;
                                RegAdd<=4'd14;
                                casezero<=casezero+1;
                            end
                        else
                            begin
                               InMuxAdd<=3'd0;
                               RegAdd <= 4'd3;       // REG3'e b□□en (Q'YA D®ﻮECEK)
                               casereg<=casereg+1;
                               casezero<=0;
                            end
                       end
                    3: begin
                        RegAdd<=4'd0;
                        casezero<=casezero+1;
                       end
                    4:begin
                        Busy<=0;
                        WE<=0;
                        startprocess <= 0;
                        remainder <=0;
                        caseremainder <=0;
                        countcase <=0;
                        START_A_SUB_B <=0;
                        casesubab<=0;
                        STEP <=0;
                        twoscomplement<=0;
                        REGISTER_LOADS<=0;
                        startprocess <=0;
                        casereg <=0;
                        casetwoscomplement<=0;
                        casezero<=0;
                      end
                    endcase

                end
            1:  begin
                case(casezero)
                    0: begin
                        InMuxAdd<=3'd1;
                        RegAdd<=4'd1;
                        casezero<=casezero+1;
                       end
                    1:  begin
                        InsSel<=2'd3;
                        casezero<=casezero+1;
                        end
                    2: begin
                        if(Z)
                            begin
                                CUconst<=8'b1111_1111;
                                InMuxAdd<=3'd2;
                                RegAdd<=4'd14;
                                casezero<=casezero+1;
                            end
                        else
                            begin
                               InMuxAdd<=3'd1;
                               RegAdd <= 4'd4;       // REG3'e b□□en (Q'YA D®ﻮECEK)
                               casereg<=casereg+1;
                               casezero<=0;
                            end
                       end
                    3: begin
                        RegAdd<=4'd0;
                        casezero<=casezero+1;
                       end
                    4:begin
                        Busy<=0;
                        WE<=0;
                        startprocess <= 0;
                        remainder <=0;
                        caseremainder <=0;
                        countcase <=0;
                        START_A_SUB_B <=0;
                        casesubab<=0;
                        STEP <=0;
                        twoscomplement<=0;
                        REGISTER_LOADS<=0;
                        startprocess <=0;
                        casereg <=0;
                        casetwoscomplement<=0;
                        casezero<=0;
```

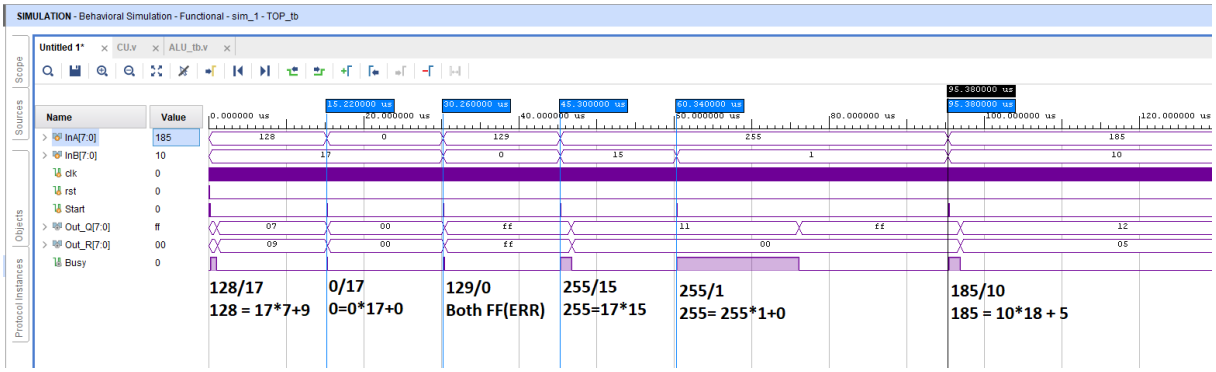
```
                    end
                endcase

            end
        2:  begin                   // REG5'e XOR i穩 full 1 (Ayn zamanda -1)
              CUconst<=8'b1111_1111;
              InMuxAdd <= 3'd2;
              RegAdd <= 4'd5;
              casereg <= casereg+1'b1;
            end
        3:  begin                   // REG6ya 2s complement i穩 1
               CUconst <= 8'b0000_0001;
               InMuxAdd<=3'd2;
               RegAdd <= 4'd6;
               casereg <= casereg+1'b1;
            end
         4: begin
                CUconst<= 8'd0;         // REG7'ye zerinde ilem yapmak i穩 full 0, A: remainder
                InMuxAdd <= 3'd2;
                RegAdd <= 4'd7;
                casereg <= casereg+1'b1;
            end
         5: begin
            twoscomplement<=1;
            casereg <= casereg+1;
            end

        default: ;

        endcase
        end

    if(twoscomplement)
        begin
            case(casetwoscomplement)

                0:begin
                     OutMuxAdd <= 4'd4;
                     InMuxAdd <= 4'd4;
                     RegAdd <= 4'd1;          // B□□ A'ya yazld
                     casetwoscomplement<=casetwoscomplement+1;
                   end
                1:begin
                     OutMuxAdd<=4'd5;     //full 1 XOR i穩
                     InMuxAdd<=4'd4;
                     RegAdd <= 4'd2;
                     casetwoscomplement<=casetwoscomplement+1;
                     InsSel<=2'd1;        // XOR OPERATION
                   end
                2:begin
                     InMuxAdd<=4'd3;      // ALU result se穡di
                     RegAdd<=4'd1;        //B□□'in XORu ALUinA'ya yazld
                     casetwoscomplement<=casetwoscomplement+1;
                   end
                3:begin
                     OutMuxAdd<=4'd6;
                     InMuxAdd<=4'd4;
                     RegAdd<=4'd2;        // 0000_0001 B'ye yazld
                     InsSel<=2'd2;
                     casetwoscomplement<=casetwoscomplement+1;
                   end
                4:begin
                     InMuxAdd<=4'd3; //ALUresult se穡di
                     RegAdd<=4'd8; // B╞EN2 ┘'s COMPLEMENT┘ARTIK REG10DA
                     START_A_SUB_B<=1;
                     casetwoscomplement<=0;
                     twoscomplement<=0;
                   end
            endcase
        end


        if(START_A_SUB_B)
        begin
            case(casesubab)
            0:  begin
                     OutMuxAdd<=4'd3;         // B╞□EN ALUinA'ya
                     InMuxAdd<=3'd4;
                     RegAdd<=4'd1;
                     casesubab<=casesubab+1;
                end
            1:  begin
                     OutMuxAdd<=4'd8;
                     InMuxAdd<=3'd4;
```

```
                                RegAdd<=4'd2;
                                InsSel <=2'd2;          // ADD A - B
                                casesubab<=casesubab+1;
                            end
                        2:  begin
                                if(!CO)
                                    begin
                                    countcase<=countcase+1'd1;
                                    casesubab<=1'b1;
                                    InMuxAdd<=3'd3;
                                    RegAdd<=4'd1;
                                    end
                                else
                                    begin
                                    remainder<=1;
                                    InMuxAdd<=3'd3;
                                    RegAdd<=4'd1;
                                    START_A_SUB_B<=0;
                                    end
                            end
                        endcase
                    end

                     if(remainder)
                                    begin
                                    case(caseremainder)
                                        0:begin
                                            CUconst<= countcase;
                                            InMuxAdd<=3'd2;
                                            RegAdd<=4'd0;
                                            caseremainder<=caseremainder+1;
                                          end
                                        1:begin
                                            OutMuxAdd<=4'd4;
                                            InMuxAdd<=3'd4;
                                            RegAdd<=4'd2;
                                            caseremainder <= caseremainder+1;
                                          end
                                        2:begin
                                            InMuxAdd<=3'd3;
                                            RegAdd<=4'd14;
                                            caseremainder <= caseremainder+1;
                                            Busy<=0;
                                          end
                                        3:begin
                                            CUconst<=0;
                                            WE<=0;
                                            startprocess <= 0;
                                            remainder <=0;
                                            caseremainder <=0;
                                            countcase <=0;
                                            START_A_SUB_B <=0;
                                            casesubab<=0;
                                             STEP <=0;
                                            twoscomplement<=0;
                                            REGISTER_LOADS<=0;
                                            startprocess <=0;
                                            casereg <=0;
                                            casetwoscomplement<=0;

                                          end


                                    endcase
                                    end


                        end

end


endmodule
```

RB module:

```
module RB(

input clk,
input rst,
//
input [2:0] InMuxAdd,
input [7:0] InA,
input [7:0] InB,
input [7:0] CUconst,
input [7:0] ALUout,
//
input WE,
input [3:0] RegAdd,
//
output [7:0] ALUinA,
output [7:0] ALUinB,
output [7:0] Out_Q,
output [7:0] Out_R,
input [3:0] OutMuxAdd
    );


wire [15:0] En;
wire [7:0] Rout [15:0];
wire [7:0] RegOut;
wire [7:0] Rin;

//  Input MUX
InMuxAdd_ INPUTMUX (
            .InMuxAdd(InMuxAdd),
            .InA(InA),
            .InB(InB),
            .CUConst(CUconst),
            .ALUout(ALUout),
            .RegOut(RegOut),
            .Rin(Rin) );

// Enable Decoder
RegAddDecoder DECODER (.WE(WE),.RegAdd(RegAdd),.En(En));

// REGISTERS:
Register REGISTER0 (.Rin(Rin),.En(En[0]),.clk(clk),.rst(rst),.Rout(Rout[0]));
Register REGISTER1 (.Rin(Rin),.En(En[1]),.clk(clk),.rst(rst),.Rout(Rout[1]));
Register REGISTER2 (.Rin(Rin),.En(En[2]),.clk(clk),.rst(rst),.Rout(Rout[2]));
Register REGISTER3 (.Rin(Rin),.En(En[3]),.clk(clk),.rst(rst),.Rout(Rout[3]));
Register REGISTER4 (.Rin(Rin),.En(En[4]),.clk(clk),.rst(rst),.Rout(Rout[4]));
Register REGISTER5 (.Rin(Rin),.En(En[5]),.clk(clk),.rst(rst),.Rout(Rout[5]));
Register REGISTER6 (.Rin(Rin),.En(En[6]),.clk(clk),.rst(rst),.Rout(Rout[6]));
Register REGISTER7 (.Rin(Rin),.En(En[7]),.clk(clk),.rst(rst),.Rout(Rout[7]));
Register REGISTER8 (.Rin(Rin),.En(En[8]),.clk(clk),.rst(rst),.Rout(Rout[8]));
Register REGISTER9 (.Rin(Rin),.En(En[9]),.clk(clk),.rst(rst),.Rout(Rout[9]));
Register REGISTER10 (.Rin(Rin),.En(En[10]),.clk(clk),.rst(rst),.Rout(Rout[10]));
Register REGISTER11 (.Rin(Rin),.En(En[11]),.clk(clk),.rst(rst),.Rout(Rout[11]));
Register REGISTER12 (.Rin(Rin),.En(En[12]),.clk(clk),.rst(rst),.Rout(Rout[12]));
Register REGISTER13 (.Rin(Rin),.En(En[13]),.clk(clk),.rst(rst),.Rout(Rout[13]));
Register REGISTER14 (.Rin(Rin),.En(En[14]),.clk(clk),.rst(rst),.Rout(Rout[14]));
Register REGISTER15 (.Rin(Rin),.En(En[15]),.clk(clk),.rst(rst),.Rout(Rout[15]));
// Out Mux

OutMuxAdd_ MUXING (.OutMuxAdd(OutMuxAdd),
            .Rout0(Rout[0]),
            .Rout1(Rout[1]),
            .Rout2(Rout[2]),
            .Rout3(Rout[3]),
            .Rout4(Rout[4]),
            .Rout5(Rout[5]),
            .Rout6(Rout[6]),
            .Rout7(Rout[7]),
            .Rout8(Rout[8]),
            .Rout9(Rout[9]),
            .Rout10(Rout[10]),
            .Rout11(Rout[11]),
            .Rout12(Rout[12]),
            .Rout13(Rout[13]),
            .Rout14(Rout[14]),
            .Rout15(Rout[15]),
            .RegOut(RegOut)
);

assign Out_Q = Rout[0];
assign Out_R = Rout[14];
assign ALUinA = Rout[1];
assign ALUinB = Rout[2];

endmodule
```

Simulation outputs of My Division Algorithm:

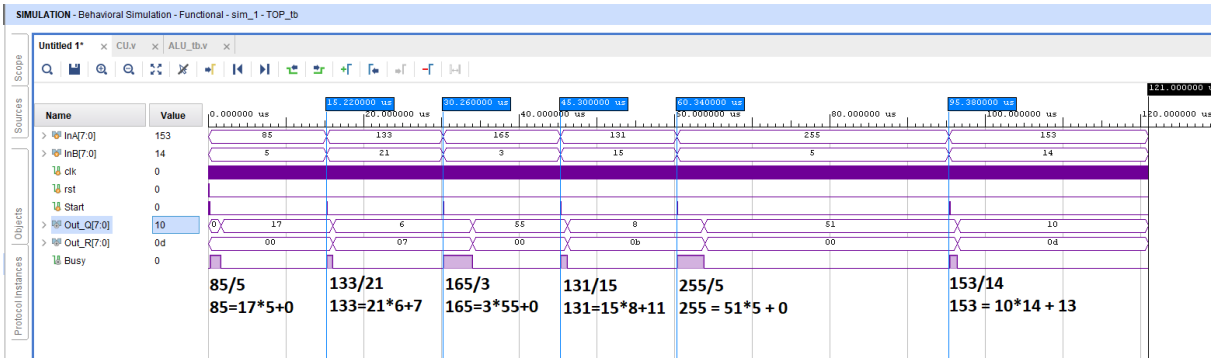


As we can see, when a start signal comes circuit starts division operation and gives Busy output HIGH. When the operation ends, gives the right outputs, Busy goes 0 and can be usable again and again. But some outputs are not correct in this simulation. **Q and R values in this simulation is hexadecimal.**

1- 128/17 given,outputs Q = 7 and R=9. >> 128 = 7*17+9 is correct.
2- 0/19 given,outputs Q=0 and R=0. >> 0 = 0*19 + 0 is correct.
3- 129/0 is given,outputs Q=FF and R=FF >> INTEGER/0 => ERROR(BOTH FF)
4- 255/15 is given,outputs Q=17 and R=0 >> 255 = 15*17 + 0 is correct.
5- 255/1 is given,outputs Q=255 and R=0 >> 255 = 255*1 + 0 is correct.
6- 185/10 is given, outputs Q=18 and R=5 >> 185 =10*15 + 5 is correct.

My algorithm works in 8 bit A and 7 bit B too. That's why it works okay in InB=19 and InB=17.

Second simulation of My Division Algoirthm:



Q is decimal, R is hexadecimal in this simulation.

1- 85/5 given,outputs Q = 17 and R=0. >> 85 = 5*17+0 is correct.
2- 133/21 given,outputs Q=6 and R=7. >> 133 = 6*21 + 7 is correct.
3- 165/3 is given,outputs Q=55 and R=0 >> 165 = 3*55 + 0 is correct
4- 131/15 is given,outputs Q=8 and R=11 >> 131 = 15*8 + 11 is correct.
5- 255/5 is given,outputs Q=51 and R=0 >> 255 = 51*5 + 0 is correct.
6- 153/14 is given, outputs Q=10 and R=13 >> 153 =14*10 + 13 is correct.

My Division Algorithm works as i expected.

My second algorithm works perfect and there is no faulty case output in 8 bit A and 4 bit B.

But their difference is, second algorithm is slower in cases like 255/1, 254/1 250/1 etc.

I uploaded to the ninova an archive which includes both of them.

MyProject.rar includes My Division Algorithm's project files.

Non-Restoring Division Project.rar includes the non-restoring division algorithm which i implemented.

Also i added RTL schematics of Control Units to the Appedix section.

1st page is My Division Algorithm's RTL Schematic,
2nd page is Non-Restoring Division Algorithm's RTL Schematic.

In project sheet, there is no declaration of timing and utilization analysis, so i didn't do these.

Thanks for everything you've teached.