**Non Restoring Division în Verilog**

*Proiect realizat de către:* **Stuparu Daniel-Mihai,**

**Tomuș Nicolas**

Anul 2, AC, CTI-Româna

Cuprins:

1. Introducere……………………………………………………............……..2
2. Algoritmul..........…………………………………………………………….2
3. Schema de asamblu ……......................................................................……..3
4. Exemplu..........................................................................................................4
5. Implementare..................................................................................................5
6. Registrii ...........………………………………………………...…….5
7. Paraller adder substactor …………………………………...…...…...8
8. Control unit ………….....................……………………...………….9
9. Non restoring division........................................................................12
10. Testbench............................................................................................14
11. Exemple testbench Verilog………........……………..……………...……...15

VII. Bibliografie...................................................................................................18

I. Introducere

Non-Restoring Division este o metodă de împărțire utilizată în arhitecturile de procesoare, care constă în efectuarea unui număr finit de pași pentru a obține rezultatul împărțirii. Această metodă este numită "non-restoring" pentru că nu "restaurează" dividendul la fiecare pas, ci folosește un rest intermediar pentru a progresa în operația de împărțire.

În mod tradițional, algoritmul Non-Restoring Division folosește două registre, numit divizor și rest. În fiecare pas, se determină dacă restul intermediar este mai mic sau mai mare decât divizorul, iar în funcție de această verificare, restul intermediar este ajustat și divizorul este rotit spre stânga cu o poziție. Acești pași se repetă până când divizorul ajunge să aibă aceeași lungime cu dividendul, iar la final se obține atât rezultatul împărțirii cât și restul.

Non-Restoring Division este considerată mai rapidă decât metoda Restoring Division, dar necesită mai mult hardware și are nevoie de un număr fix de pași, indiferent de valoarea dividendului și a divizorului. Implementarea sa poate fi realizată în limbaje de descriere hardware, cum ar fi Verilog sau VHDL, prin crearea unui modul hardware care să efectueze algoritmul de împărțire.

II. Algoritmul

SHIFT: S.A.Q [7:1] = A.Q

if S = 0 then

S.A = S.A - M;

else

S.A = S.A + M;

if S = 0 then

Q[0] = 1

else

Q[0] = 0;

if COUNT 7 = 1 then

goto CORR

else

COUNT ++; go to SHIFT

CORR: if S = 1 then

S.A = S.A + M;

III. Schema de asamblu

AQ = Dividend

M = Divizor

COUNT=0, S=0

Shift Left SAQ

S=?

SQ <- Q - M

SQ <- Q + M

S=?

= 1

= 0

COUNTER++

COUNT=8

S=?

SQ <- Q + M

A -> Restul

Q -> Câtul

STOP

1

0

1

0

No

Yes

0

1

IV. Exemplu

5771/135 = 42 rest 101

5771 = 0001 0110 1000 1011

135 = 1000 0111

-135 = 1 0111 1001

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| COUNT | S | A | Q | M |
| 0 | 0  0  1  1 | 0001 0110  0010 1101 +  0111 1001  1010 0110 | 1000 1011  0001 0110  0001 0110 | 1000 0111 |
| 1 | 1  0  1 | 0100 1100 +  1000 0111  1101 0011 | 0010 1100  0010 1100 |  |
| 2 | 1  0  0 | 1010 0110 +  1000 0111  0010 1101 | 0101 1000  0101 1001 |  |
| 3 | 0  1  1 | 0101 1010 +  0111 1001  1101 0011 | 1011 0010  1011 0010 |  |
| 4 | 1  0  0 | 1010 0111 +  1000 0111  0010 1110 | 0110 0100  0110 0101 |  |
| 5 | 0  1  1 | 0101 1100 +  0111 1001  1101 0101 | 1100 1010  1100 1010 |  |
| 6 | 1  0  0 | 1010 1011 +  1000 0111  0011 0010 | 1001 0100  1001 0101 |  |
| 7 | 0  1  1 | 0110 0101 +  0111 1001  1101 1110 | 0010 1010  0010 1010 |  |
| Corecție | 1  0  0 | 1101 1110 +  1000 0111  0110 0101 |  |  |

Câtul = Q = 0010 1010 = 42

Restul = S + A = 0 0110 0101 = 101

V. Implementare

Verilog

a. Registrii

module reg\_m(

input clk, rst, ld\_in\_bus,

input [7:0] in\_bus,

output reg [7:0] rez Registrul M

); Fig. 1

always @ ( posedge clk or negedge rst )

if ( !rst ) rez <= 0;

else if ( ld\_in\_bus ) rez <= in\_bus;

endmodule

module reg\_q(

input clk, rst, ld\_in\_bus, left\_shift, set\_lsb, lsb,

input [7:0] in\_bus,

output reg [7:0] rez

);

always @ ( posedge clk or negedge rst )

if ( !rst ) rez <= 0;

else if ( ld\_in\_bus ) rez <= in\_bus; Registru Q

else if( left\_shift ) begin Fig. 2

rez <= rez << 1;

end

else if(set\_lsb) begin

rez[0] <= lsb;

end

endmodule

module reg\_sign(

input clk, rst, ld, in,

output reg rez

);

always @ (posedge clk or negedge rst) begin

if(!rst) Registrul pentru semn

rez <= 0; Fig. 3

else

if(ld)

rez <= in;

end

endmodule

module reg\_a(

input clk, rst, ld\_in\_bus, ld\_sum, left\_shift, lsb,

input [7:0] in\_bus,

input [7:0] sum,

output reg [7:0] rez

);

always @ ( posedge clk or negedge rst ) begin

if ( !rst ) rez <= 0;

else if ( ld\_in\_bus ) begin

rez <= in\_bus; Registrul A

end Fig. 4

else if(ld\_sum) begin

rez <= sum;

end

else if( left\_shift ) begin

rez <= rez << 1;

rez[0] <= lsb;

end

end

endmodule

module reg\_counter(

input clk, rst, increment,

output reg [2:0] rez,

output reg count\_is\_7

);

always @ (posedge clk or negedge rst) begin

if(!rst) begin

rez <= 3'b000; Registrul counter

count\_is\_7 <= 0; Fig. 5

end

else if(rez == 3'b111) begin

rez <= 3'b000;

count\_is\_7 <= 1;

end

else if(increment)

rez <= rez + 1;

end

endmodule

module reg\_out(

input clk, rst, ld\_in\_bus,

input [7:0] in1,

input [7:0] in2,

output reg [7:0] rez

);

reg aux;

always @ (posedge clk or negedge rst) begin

if(!rst) begin Registrul de iesire

rez <= 0; Fig. 6

aux <= 0;

end

else if(ld\_in\_bus)

if(aux == 0) begin

rez <= in1;

aux <= 1;

end

else begin

rez <= in2;

aux <= 0;

end

end

endmodule

1. Modulul **reg\_m** (din Fig. 1) implementează un registru cu o intrare **in\_bus** și o ieșire **rez**. Intrarea **rst** pune la început semanlul **rez** pe 0 iar în celelalte cazuri semanlul **ld\_in\_bus** controlează dacă noul conținut al registrelor trebuie încărcat cu **in\_bus**. Acestea sunt actualizate în mod sincron cu semnalul **clk**.

2. Modulul **reg\_q** (din Fig. 2) implementează un registru cu intrarea **in\_bus** și ieșirea **rez**. Intrarea **rst** pune la început semanlul **rez** pe 0, intrărea **ld\_in\_bus** controlează intrarea **in\_bus** și **set\_lsb** dacă LSB-ul trebuie încărcată iar intrarea **left\_shift** controlează dacă trebuie făcută o mutare la stânga a registrelor.

3. Modulul **reg\_a** (din Fig. 3) implementează un registru cu intrările **in\_bus** și **sum** și ieșirea **rez**. . Intrarea **rst** pune la început semanlul **rez** pe 0, intrarea **ld\_in\_bus** controlează intrarea **in\_bus**, **ld\_sum** controlează intrarea pentru **sum** iar **left\_shift** controlează dacă trebuie făcută o mutare la stânga a registrelor și incarcă **lsb** pe pozitia LSB.

4. Modulul **reg\_out** (din Fig.4) implementează un registru cu intrările **in1** și **in2** și ieșirea **rez**. Intrarea **ld\_in\_bus** controlează dacă intrarea trebuie încărcată cu **in1** sau **in2**, în funcție de starea internă a registrelor.

5. Modulul **reg\_sign** (din Fig. 5) implementează un registru cu intrarea **in** și ieșirea **rez**. . Intrarea **rst** pune la început semanlul **rez** pe 0 daca este negativ, respectiv îl setează pe **rez** cu valoarea lui **in**.

6. Modulul **reg\_counter** (din Fig. 6) implementează un contor de 3 biți. Intrarea **increment** controlează dacă contorul trebuie să fie incrementat. Ieșirea **rez** reprezintă valoarea curentă a contorului. Semnalul **count\_is\_7** este activat atunci când contorul atinge valoarea 7 și se resetează la 0.

b.Parallel adder substactor

module FAC(

input x, y, c\_in,

output z, c\_out

);

Full adder cell

assign z = x ^ y ^ c\_in; Fig. 7

assign c\_out = (x & y) | (x & c\_in) | (y & c\_in);

endmodule

module parallel\_adder\_subtractor(

input operation\_type,

input sign\_in,

input [7:0] x,

input [7:0] y,

output [7:0] result,

output sign\_out

);

wire [9:0] carries;

wire [8:0] sum;

reg [8:0] x\_ext;

reg [8:0] y\_ext;

assign carries[0] = ~operation\_type; Parallel adder

subtractor

always @(\*) begin Fig. 8

x\_ext = {sign\_in, x};

y\_ext = {1'b0, y};

end

genvar i;

generate

for (i = 0; i < 9; i = i + 1) begin

FAC add\_inst(.x(x\_ext[i]), .y((operation\_type) ? y\_ext[i] : ~y\_ext[i]), .c\_in(carries[i]), .z(sum[i]), .c\_out(carries[i+1]));

end

endgenerate

assign result = sum[7:0];

assign sign\_out = sum[8];

endmodule

7. Modulul **FAC** (din Fig. 7) este o implementare a unui full-adder-cell. Acest modul primește trei intrări **x**, **y** și **c\_in** și două ieșiri **z** și **c\_out**. Intrările **x** și **y** reprezintă biții care trebuie adunați, iar **c\_in** reprezintă bitul de transport de la poziția mai puțin semnificativă. Ieșirile **z** și **c\_out** reprezintă rezultatul adunării și bitul de transport pentru poziția mai semnificativă.

8. Modulul **parallel\_adder\_subtractor** (din Fig. 8) este o implementare a unui adunător/substractor paralel care utilizează modulul **FAC** pentru fiecare operație de aducare/scădere. Acest modul primește patru intrări **operation\_type** este un semnal care indică dacă se face o adunare sau o scădere, **sign\_in** reprezintă semnul numărului de intrare, **x** și **y** sunt numerele care trebuie adunate/scăzute iar **result** și **sign\_out** sunt iesirile modulului.

c. Control unit

`define IDLE 4'b0000

`define LD\_A 4'b0001

`define LD\_Q 4'b0010

`define LD\_M 4'b0011

`define CHECK\_SIGN 4'b0100

`define ADD 4'b0101

`define SUB 4'b0110

`define SET\_LSB\_Q 4'b0111

`define LEFT\_SHIFT 4'b1000

`define CHECK\_COUNT 4'b1001

`define END 4'b1010

module control\_unit( Modulul de control

input clk, rst, Fig. 9

input begin\_div,

input sign, cnt7,

output reg ld\_a,

output reg ld\_m,

output reg ld\_q,

output reg ld\_sign,

output reg operation,

output reg left,

output reg set\_lsb,

output reg ld\_sum,

output reg increment,

output reg fin

);

reg [3:0] state\_next, state\_reg;

always @(posedge clk or negedge rst) begin

if(!rst)

state\_reg <= 4'b0000;

else

state\_reg <= state\_next;

end

initial begin

ld\_a = 0;

ld\_q = 0;

ld\_m = 0;

ld\_sign = 0;

operation = 0;

set\_lsb = 0;

increment = 0;

ld\_sum = 0;

left = 0;

fin = 0;

end

always @(state\_reg, begin\_div) begin

state\_next = state\_reg;

case(state\_reg)

`IDLE:

begin

state\_next = `IDLE;

if(begin\_div) begin

state\_next = `LD\_A;

ld\_a = 1;

end

end

`LD\_A:

begin

ld\_a = 0;

state\_next = `LD\_Q;

ld\_q = 1;

end

`LD\_Q:

begin

ld\_q = 0;

state\_next = `LD\_M;

ld\_m = 1;

end

`LD\_M:

begin

ld\_m = 0;

left = 1;

state\_next = `LEFT\_SHIFT;

end

`LEFT\_SHIFT:

begin

increment = 0;

left = 0;

state\_next = `CHECK\_SIGN;

end

`CHECK\_SIGN:

begin

ld\_sum = 1;

ld\_sign = 1;

if(sign) begin

state\_next = `ADD;

operation = 1;

end

else begin

state\_next = `SUB;

operation = 0;

end

end

`ADD:

begin

operation = 0;

ld\_sum = 0;

ld\_sign = 0;

if(cnt7) begin

set\_lsb = 1;

state\_next = `END;

end

else begin

set\_lsb = 1;

state\_next = `SET\_LSB\_Q;

end

end

`SUB:

begin

operation = 1;

ld\_sum = 0;

ld\_sign = 0;

set\_lsb = 1;

state\_next = `SET\_LSB\_Q;

end

`SET\_LSB\_Q:

begin

ld\_sum = 0;

set\_lsb = 0;

state\_next = `CHECK\_COUNT;

end

`CHECK\_COUNT:

begin

set\_lsb = 0;

if(cnt7)

begin

if(sign) begin

ld\_sum = 1;

operation = 1;

state\_next = `ADD; //correction step

end

else

state\_next = `END;

end

else begin

state\_next = `LEFT\_SHIFT;

left = 1;

increment = 1;

end

end

`END:

begin

set\_lsb = 0;

ld\_sum = 0;

fin = 1;

increment = 0;

state\_next = `IDLE;

end

endcase

end

endmodule

9. Modulul **control\_unit** (din Fig. 9). Acest modul reprezintă o unitate de control pentru un procesor aritmetic, care controlează fluxul de date și operațiile efectuate de unitatea de procesare.

Modulul control\_unit are 11 porturi, 5 intrări și 10 ieșiri. Intrările sunt semnalul de **clk**, **rst, begin\_div, sign, cnt7**. Ieșirile sunt semnalul **ld\_a**, **ld\_m**, **ld\_q**, **ld\_sign**, **operation**, **left**, **set\_lsb**, **ld\_sum**, **increment** și **fin**.

**state\_reg** și **state\_next** sunt două registre care stochează starea curentă și următoarea stare a modulului, respectiv. Acestea sunt actualizate în funcție de semnalul de **clk** și **rst** și sunt utilizate într-o structură case pentru a determina starea următoare a modulului.

Apoi, pentru fiecare stare, modulul decide ce semnale să seteze pe ieșiri și care este următoarea stare a modulului. De exemplu: când starea este **LD\_A**, semnalul **ld\_a** este setat la 1, astfel încât valoarea aflată în registrul A poate fi încărcată în unitatea de procesare. După aceea, starea următoare este **LD\_Q**, iar semnalul **ld\_a** este setat la 0, iar semnalul **ld\_q** este setat la 1.

De menționat este și faptul ca semnalele de control aferente unei anumite stări se face in prealabil față de acea stare (**cu o stare înainte**). Acest lucru se datorează faptului că modificările lui **state\_reg** se întamplă pe **posedge clk.**

Top of Form

d. Non restoring division

module non\_res\_div\_3\_0(

input [7:0] in\_bus,

input begin\_div,

input clk, rst,

output fin,

output [7:0] out\_bus

);

wire [7:0] a;

wire [7:0] q;

wire [7:0] m;

wire [7:0] rez;

wire [2:0] count;

wire left;

wire ld\_a;

wire ld\_q;

wire ld\_m;

wire ld\_sign;

wire sign\_in;

wire sign\_out;

wire ld\_sum;

wire set\_lsb;

wire increment;

wire cnt7;

wire finish;

wire operation;

wire [7:0] out;

reg\_sign inst0(.clk(clk), .rst(rst), .ld(ld\_sign), .in(sign\_out), .rez(sign\_in));

reg\_a inst1(.clk(clk), .rst(rst), .ld\_in\_bus(ld\_a), .ld\_sum(ld\_sum), .left\_shift(left), .lsb(q[7]), .in\_bus(in\_bus), .sum(rez), .rez(a));

reg\_q inst2(.clk(clk), .rst(rst), .ld\_in\_bus(ld\_q), .left\_shift(left), .set\_lsb(set\_lsb), .lsb(~sign\_in), .in\_bus(in\_bus), .rez(q));

reg\_m inst3(.clk(clk), .rst(rst), .ld\_in\_bus(ld\_m), .in\_bus(in\_bus), .rez(m));

reg\_counter inst4(.clk(clk), .rst(rst), .increment(increment), .rez(count), .count\_is\_7(cnt7));

parallel\_adder\_subtractor pa(.operation\_type(operation), .sign\_in(sign\_in), .x(a), .y(m), .result(rez), .sign\_out(sign\_out));

control\_unit cu(.clk(clk), .rst(rst), .begin\_div(begin\_div), .sign(sign\_in), .cnt7(cnt7), .ld\_a(ld\_a), .ld\_m(ld\_m), .ld\_q(ld\_q), .ld\_sign(ld\_sign), .operation(operation),

.left(left), .set\_lsb(set\_lsb), .ld\_sum(ld\_sum), .increment(increment), .fin(finish));

reg\_out inst5(.clk(clk), .rst(rst), .ld\_in\_bus(fin), .in1(a), .in2(q), .rez(out));

assign fin = finish;

assign out\_bus = out;

endmodule

Non restoring division - Fig. 10

10. Modulul **non\_res\_div\_3\_0** (din Fig. 10) primește un număr de intrare de 8 biți (in\_bus), o semnalizare de începere a operației de împărțire (begin\_div), semnalizările de ceas (clk) și resetare (rst) și returnează semnalul de finalizare a operației (fin) și rezultatul împărțirii (out\_bus) de 8 biți.

e. Testbench

module non\_res\_div\_tb;

localparam CLK\_PERIOD = 20;

localparam CLK\_CYCLES = 100;

reg clk;

initial begin

clk = 0; repeat (CLK\_CYCLES \* 2) #(CLK\_PERIOD / 2) clk = ~clk;

end

reg [7:0] in\_bus;

reg rst, begin\_div;

wire fin;

wire [7:0] out\_bus;

initial begin

rst = 0;

in\_bus = 8'd0;

begin\_div = 0;

#10;

rst = 1;

#10

rst = 0;

#10;

rst = 1;

#20;

begin\_div = 1;

#20;

begin\_div = 0;

in\_bus = 8'b0001\_1111; //Upper half of dividend, change at will

#20;

in\_bus = 8'b1010\_1001; //Lower half of dividend, change at will

#20;

in\_bus = 8'b0100\_1111; //Divisor

#20;

end

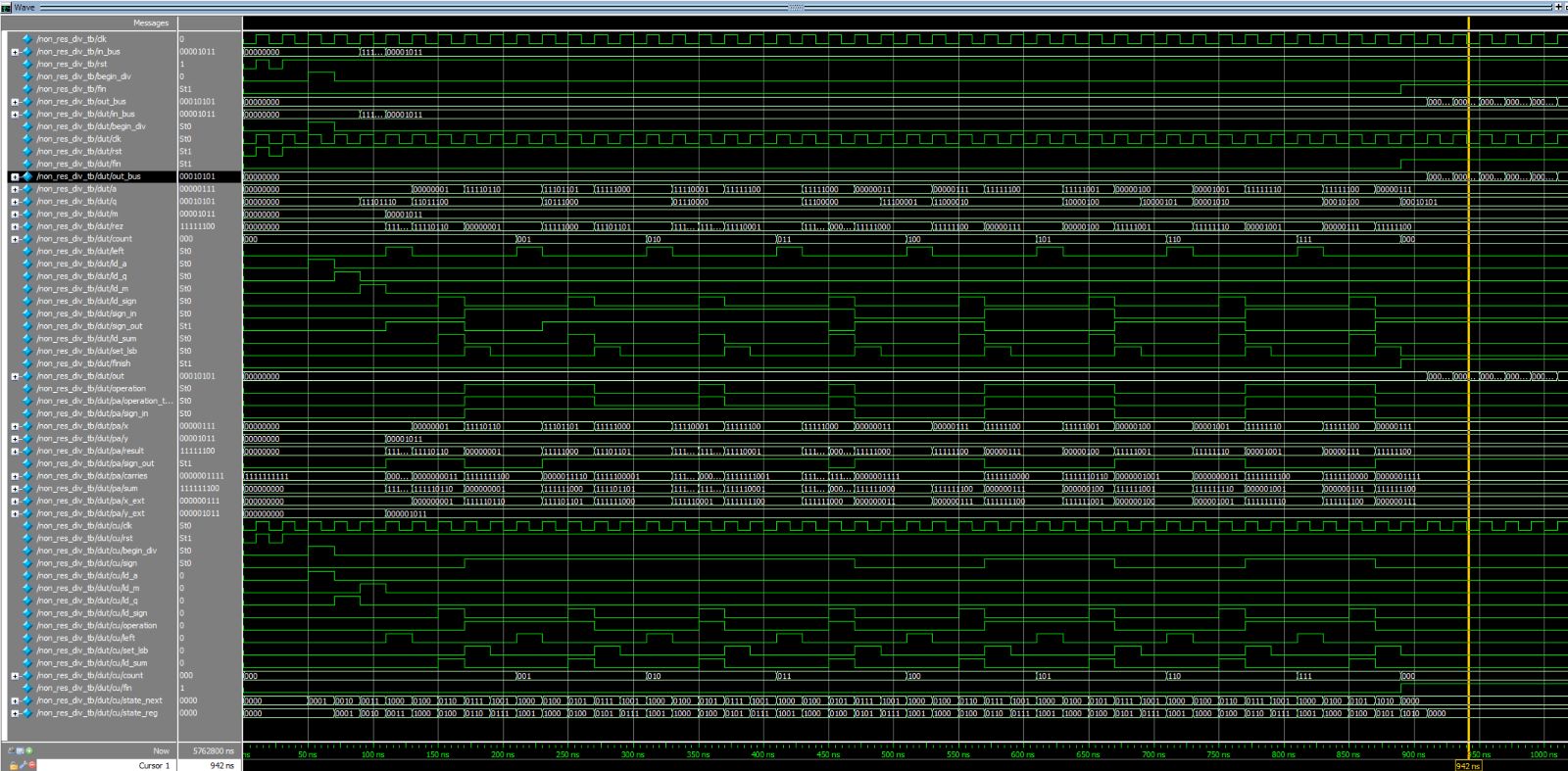
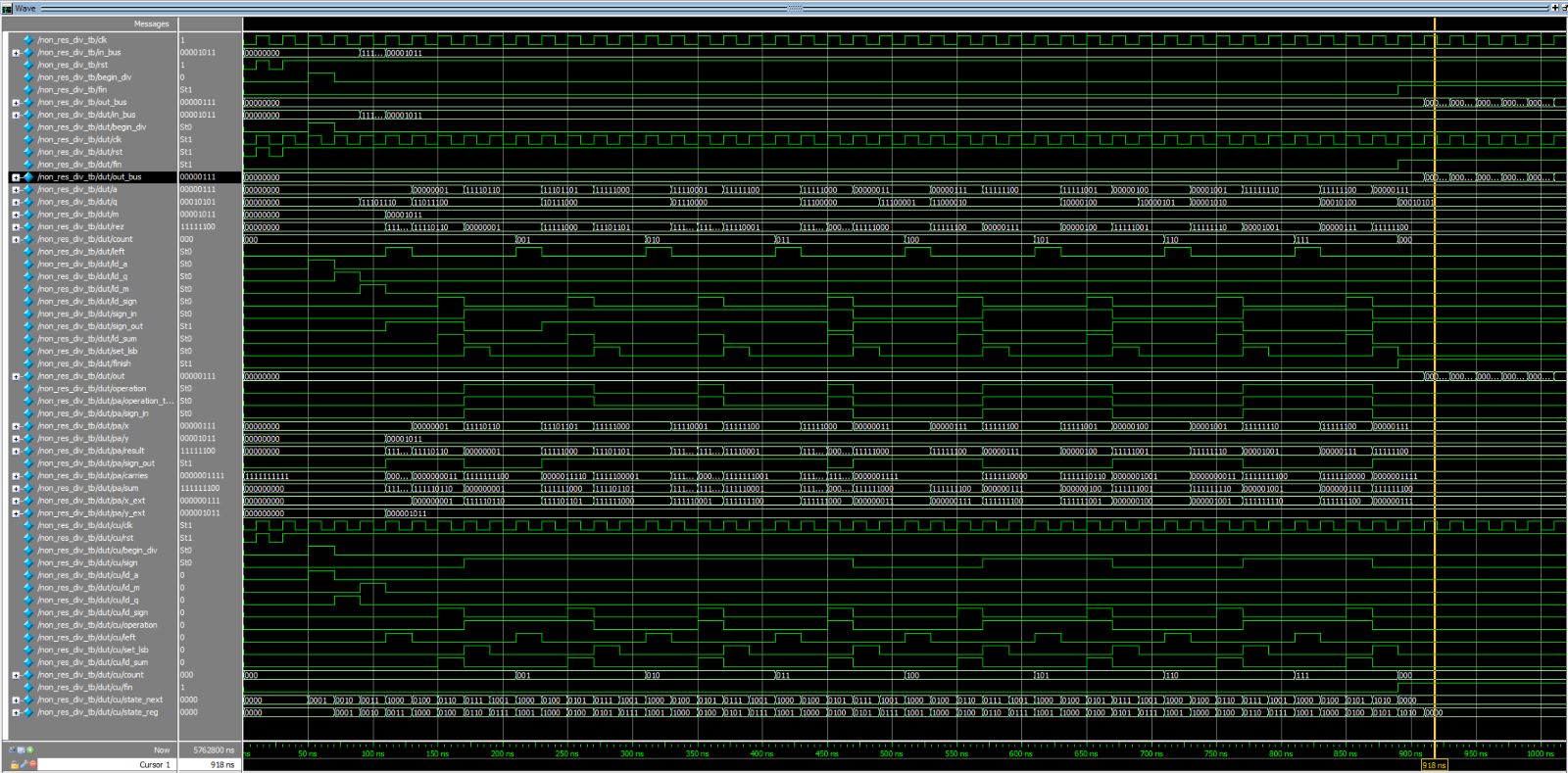
non\_res\_div\_3\_0 dut(.in\_bus(in\_bus), .begin\_div(begin\_div), .clk(clk), .rst(rst), .fin(fin), .out\_bus(out\_bus));

endmodule

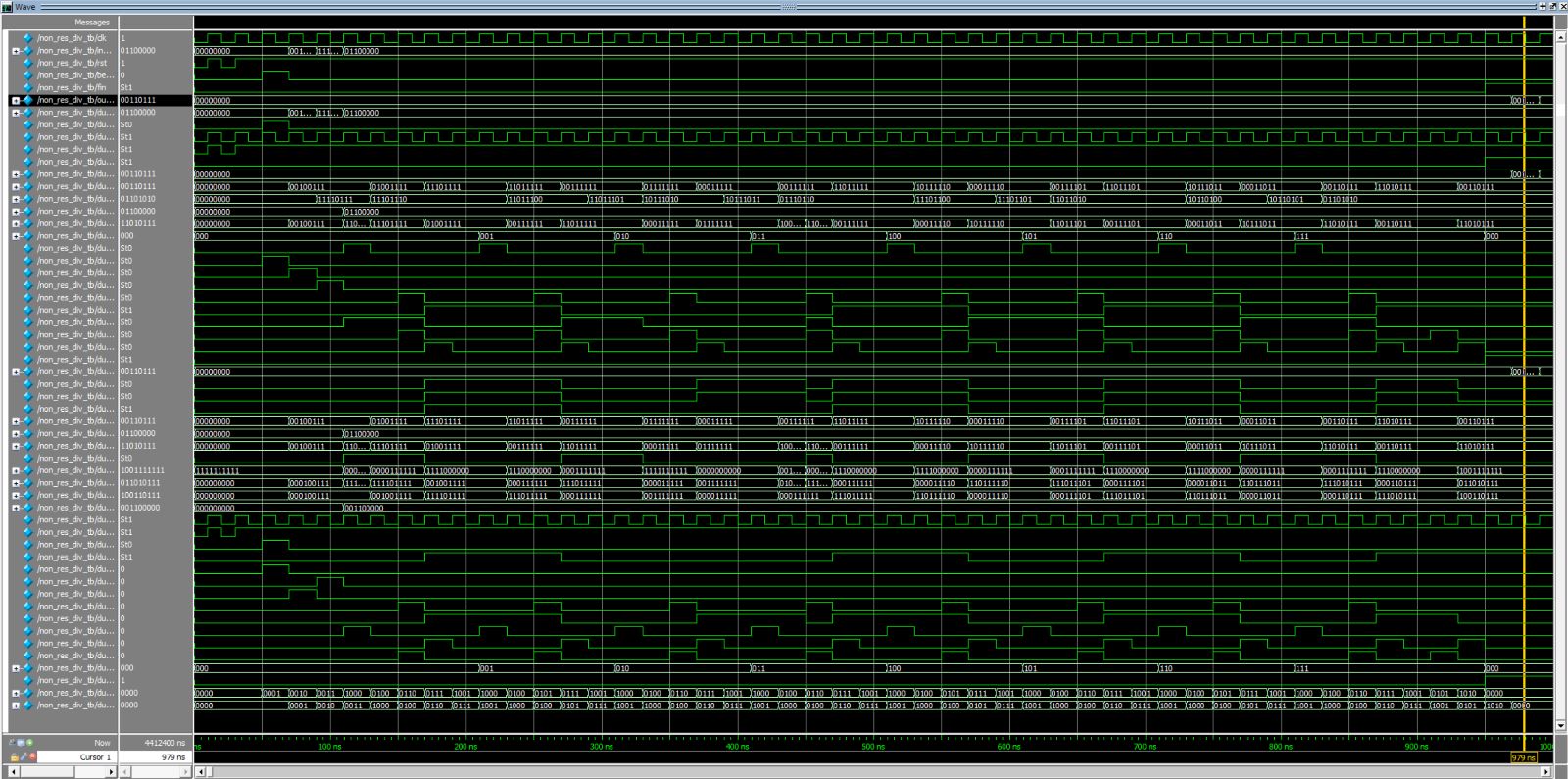
Testbench - Fig. 11

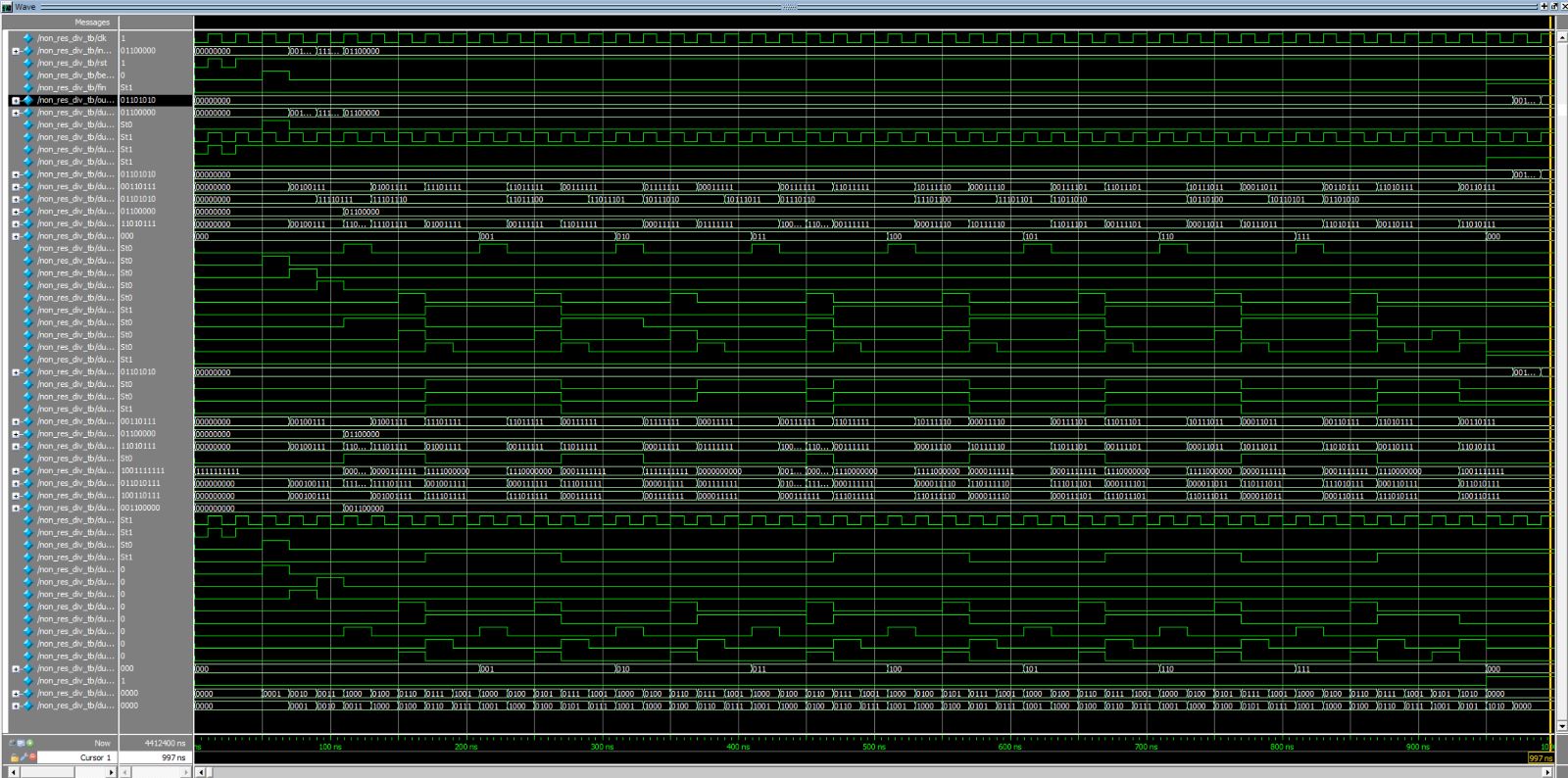
VI. Exemple testbench Verilog:

238 / 11 = 21 rest 7

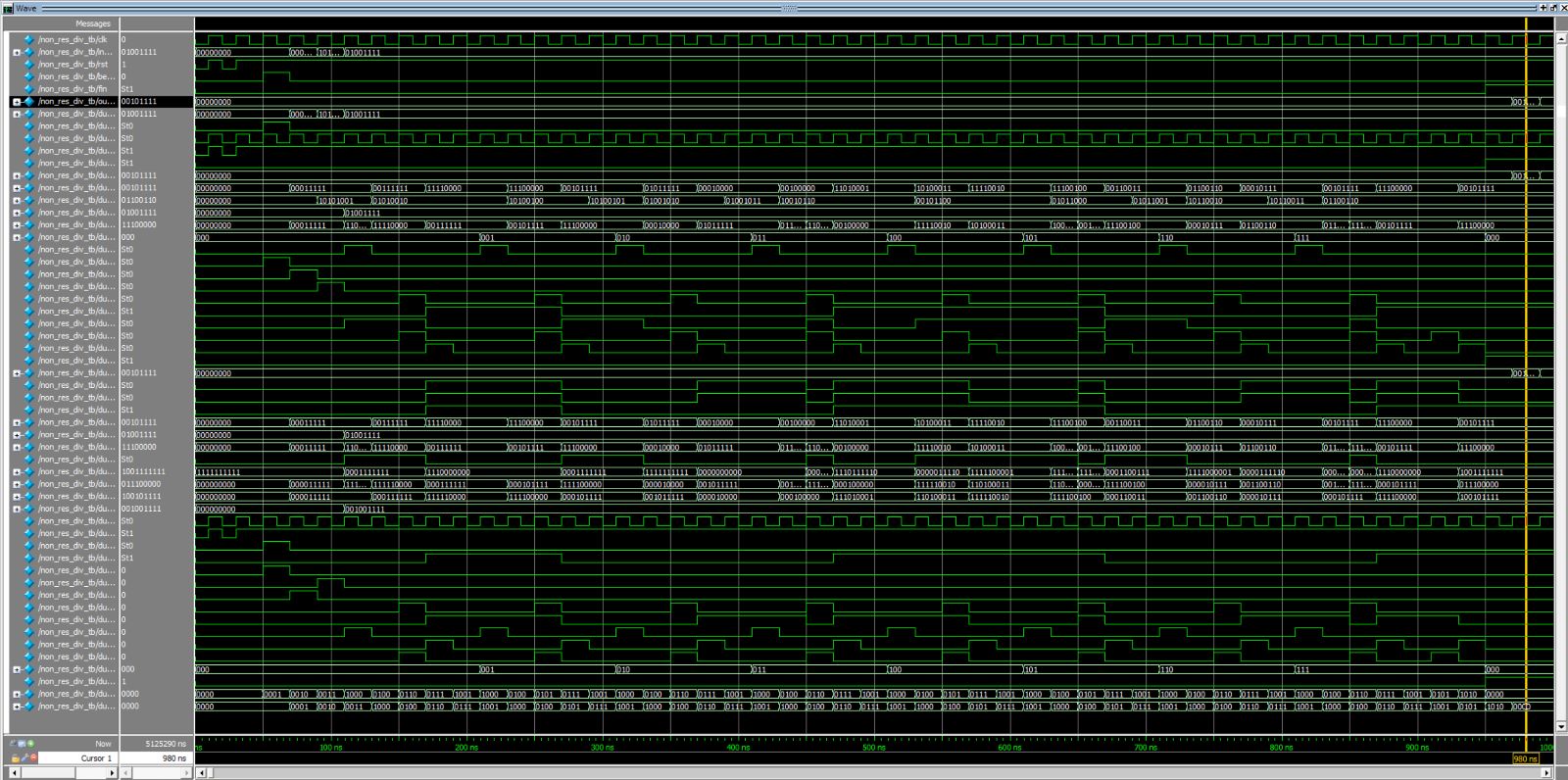


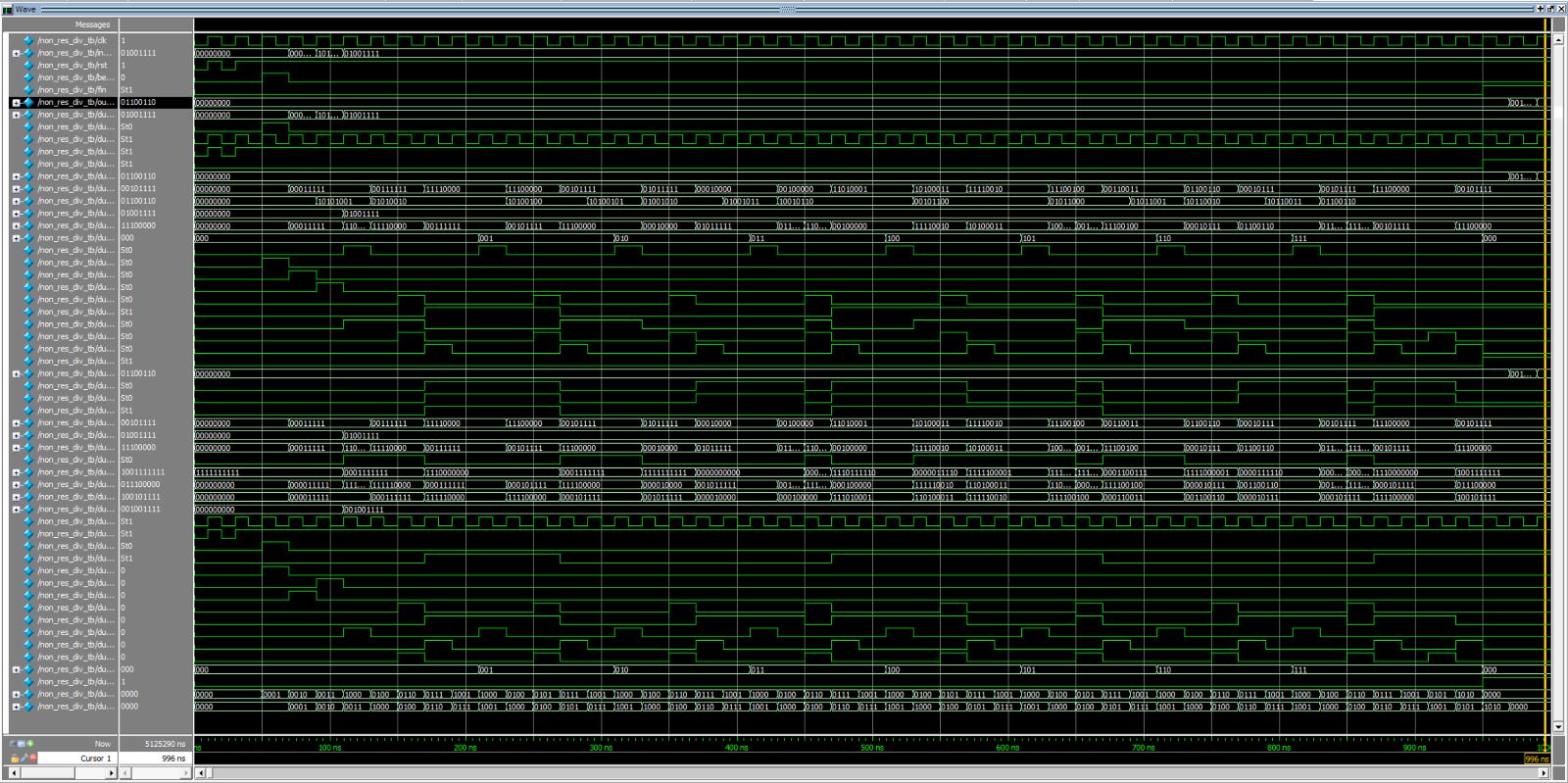
10231 / 96 = 106 rest 55





8105 / 79 = 102 rest 47





VII. Bibliografie

1. Wikipedia - Non-restoring division:

https://en.wikipedia.org/wiki/Division\_algorithm#Non-restoring\_division

1. Fixed Point Arithmetic - Division:

<https://witscad.com/course/computer-architecture/chapter/fixed-point-arithmetic-division>

1. PROGRAMAREA CU LIMBAJE DE DESCRIERE HARDWARE: <https://dokumen.pub/programarea-cu-limbaje-de-descriere-hardware-aplicaii-n-limbajul-vhdl-informatica.html>
2. TRANSLATION OF DIVISION ALGORITHM INTO VERILOG HDL: <http://www.arpnjournals.org/jeas/research_papers/rp_2017/jeas_0517_6036.pdf>
3. Non Restoring Division Algorithm for Unsigned Integer: <https://www.youtube.com/watch?v=f6A3ySUdT80>

**Link github:** https://github.com/stuparu-daniel/Non-Restoring-Division-Verilog