School of Engineering and Technology

Course:	Final Year of Beng/CEE Communications & Electronic Engineering
Names of students:	Ignacio Salan
Assignment title:	Embedded Systems Project.
Supervisor:	Mr. Ian Elliot
Date:	09-05-2003

TABLE OF CONTENTS

1 INTRODUCTION

In this assignment is designed an eight bit unsigned divider following the specifications given. Next figure illustrates the process of unsigned binary division using the so-called Restoring method. The algorithm produces a quotient and remainder by dividing a dividend by a divisor. All operands are declared as 8-bit unsigned numbers, for this is used the type unsigned in IEEE package Numeric_std.

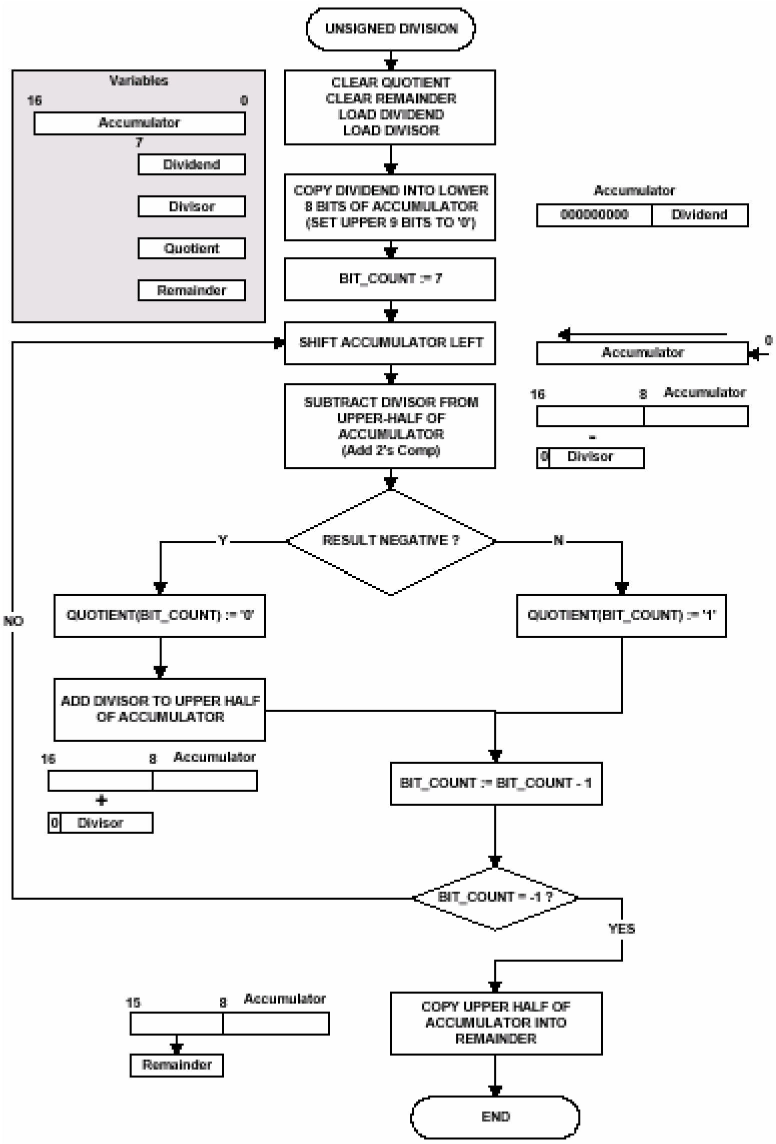

Figure 1. *Flowchart for Binary Division.*

1.1 Demonstration of the Algorithm.

This is a task to demonstrate the algorithm shown in figure 1 by means of a table illustrating each step involved in dividing the number 10011110_2 by 00001111_2. So the following table is calculated.

Bit_Count	Dividend	Divisor	Accumulator	Quotient	Remainder	Operation
0	10011110	00001111	UUUUUUUUUUUUUUUU	00000000	00000000	Load dividend and divisor. Clear Quotient and Remainder.
0	10011110	00001111	0000000010011110	00000000	00000000	Load Dividend into lower 8 bits of accumulator set upper bits 9 to 0 to '0'.
7	10011110	00001111	0000000100111100	00000000	00000000	Shift accumulator left and set bit_count to 7.
7	10011110	00001111	1111001000111100	00000000	00000000	Subtract divisor from upper-half of accumulator.
7	10011110	00001111	0000000100111100	00000000	00000000	Add divisor to upper half of accumulator.
6	10011110	00001111	0000001001111000	00000000	00000000	Shift accumulator left.
6	10011110	00001111	1111001101111000	00000000	00000000	Subtract divisor from upper-half of accumulator.
6	10011110	00001111	0000001001111000	00000000	00000000	Add divisor to upper half of accumulator.
5	10011110	00001111	0000010011110000	00000000	00000000	Shift accumulator left.
5	10011110	00001111	1111010111110000	00000000	00000000	Subtract divisor from upper-half of accumulator.
5	10011110	00001111	0000010011110000	00000000	00000000	Add divisor to upper half of accumulator.
4	10011110	00001111	0000100111100000	00000000	00000000	Shift accumulator left.
4	10011110	00001111	1111101011100000	00000000	00000000	Subtract divisor from upper-half of accumulator.
4	10011110	00001111	0000100111100000	00000000	00000000	Add divisor to upper half of accumulator.
3	10011110	00001111	0001001111000000	00000000	00000000	Shift accumulator left.
3	10011110	00001111	0000010011000000	00000000	00000000	Subtract divisor from upper-half of accumulator.
3	10011110	00001111	0000010011000000	00001000	00000000	Quotient(bit_count)='1'
2	10011110	00001111	0000100110000000	00001000	00000000	Shift accumulator left.
2	10011110	00001111	1111101010000000	00001000	00000000	Subtract divisor from upper-half of accumulator.
2	10011110	00001111	0000100110000000	00001000	00000000	Add divisor to upper half of accumulator.
1	10011110	00001111	0001001100000000	00001000	00000000	Shift accumulator left.
1	10011110	00001111	0000010000000000	00001010	00000000	Subtract divisor from upper-half of accumulator. And Quotient(bit_count)='1'
0	10011110	00001111	0000100000000000	00001010	00000000	Shift accumulator left.
0	10011110	00001111	1111100100000000	00001010	00000000	Subtract divisor from upper-half of accumulator.
0	10011110	00001111	0000100000000000	00001010	00001000	Add divisor to upper half of accumulator. Copy upper half of accumulator into remainder.

Table 1. *Table showing the steps involved in algorithm.*

As it has been shown in the table the algorithm is working as it was expected, next step in this assignment is to implement in VHDL the previous algorithm and simulate it using a random set of input operands.

1.2 Design of a VHDL Algorithm Description of the Divider

Here is described the algorithm designed and simulated in VHDL, using the ModelSim software package.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity Divider is
end entity Divider;

architecture algorithm of Divider is
begin
        do_it : process
        variable accumulator : unsigned(16 downto 0);
        variable dividend : unsigned(7 downto 0);
        variable divisor : unsigned(7 downto 0);
        variable quotient : unsigned(7 downto 0);
        variable remainder : unsigned(7 downto 0);

        begin
        quotient := (others => '0');
        remainder := (others => '0');
        dividend := "11100000";  --158
        divisor := "00011000"; --15 (division is 10.53)
        accumulator(7 downto 0) := (dividend(7 downto 0));
```

```
                accumulator(16 downto 8) := (others => '0');
                  for k in 7 downto 0 loop
                    accumulator := shift_left(accumulator,1);
                    accumulator(16 downto 8) := accumulator(16 downto 8) + (not('0' & divisor) + 1);
                    if accumulator(16) = '1' then
                          quotient(k) := '0';
                          accumulator(16 downto 8) := accumulator(16 downto 8) + ('0'& divisor);
                      else if accumulator(16)= '0' then
                          quotient(k) := '1';
                      end if;
                    end if;
                  end loop;

                remainder := accumulator(15 downto 8);
                quotient := quotient(7 downto 0);
                assert false report "simulation ended" severity failure;
         end process;
   end architecture algorithm;
```

In the previous code it can be seen that the operands are the same that the ones in the table 1, the operation is the following (158/15) generates a quotient of 10 and a remainder of 8. Now this algorithm is simulated with different numbers to prove that is working properly. The next simulations windows are obtained:

Now the dividend is 224 that in binary is 11100000 and the divisor is 24 that in binary is 00011000, this numbers generates a quotient of 9 and a remainder of 8.

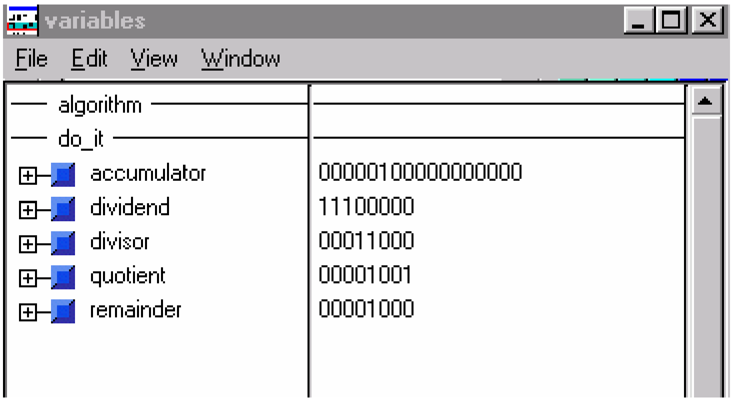

Figure 2. *Simulation window obtained with the previous numbers described.*

Now the dividend is 198 (11000110_2) and the divisor 37 (00100101_2) so the window from the simulation is,

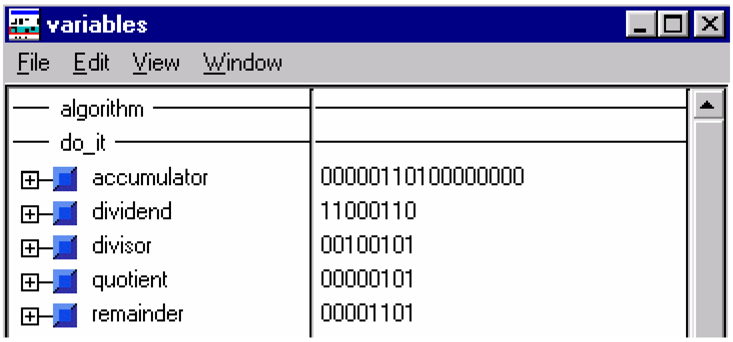

Figure 3 *Simulation window obtained with the previous numbers described.*

The previous numbers generates a quotient of 5 and a remainder of 13, as it has been demonstrated in the previous window.

The last set of numbers are going to be 59 (00111011_2) and 11 (00001011_2) these numbers generate a quotient of 5 and a remainder of 4.

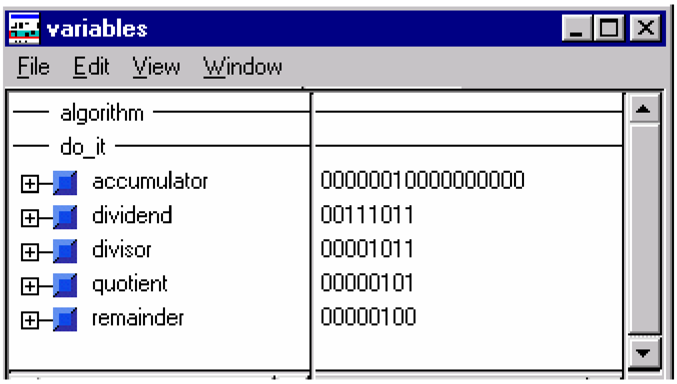

Figure 4 *Simulation window obtained with the previous numbers described.*

Trough all these simulations windows have been demonstrated that the algorithm is working as it was expected. Now is going to be described the synthesis of the register transfer level of the divider.

1.3 Register Transfer Level.

The next diagram is illustrating the description of the divider in terms of registers and all the signals needed to control the divider and have a good performance of the whole system.

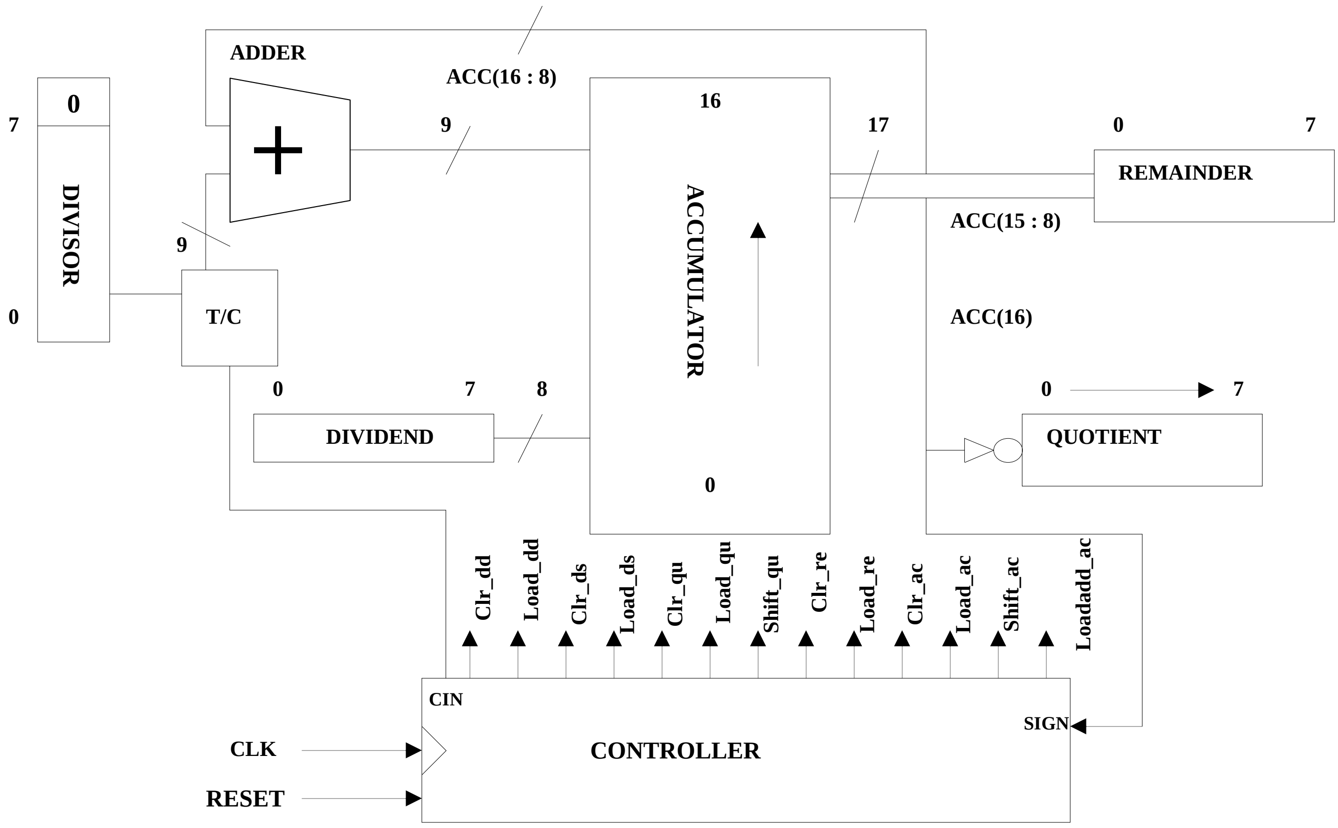

Figure 5 *Description of the RTL block diagram.*

The signal CIN is going to control when to make a subtraction (adding converting to two complement one of the numbers in this case the divisor), so when CIN is '0' the operation made is the sum of the divisor and the upper half of the accumulator, in the other case when CIN is '1' the operation is the subtraction of the divisor and the upper half of accumulator.

With the signal 'Loadadd_ac' is loading the output of the adder, all this is explained in more detail in the description of the controller. The incoming signal 'sign' is to control when to do a subtraction or a simple sum.

Now are going to be described one by one all the components and the controller of this design.

1.3.1 Dividend register

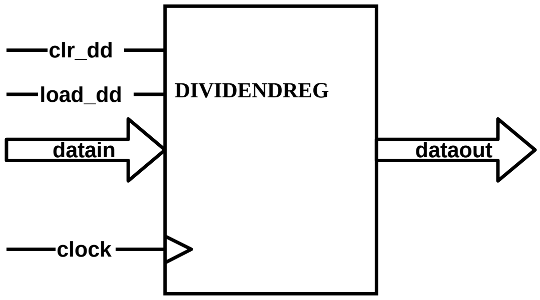

Figure 6 *Dividend Register.*

And the VHDL code for this register is the following,

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity dividendreg is
	port(
		load,
		clr,
		clock : in std_logic;
		datain : in unsigned(7 downto 0);
		dataout : out unsigned(7 downto 0)
	);
end dividendreg;
```

```
architecture v1 of dividendreg is
        signal q : unsigned(7 downto 0);
begin
        process begin
                wait until rising_edge(clock);
                if clr = '1' then
                   q <= (others => '0');
                elsif load = '1' then
                   q <= datain;
                end if;
        end process;
        dataout <= q;
end v1;
```

1.3.2 Divisor register

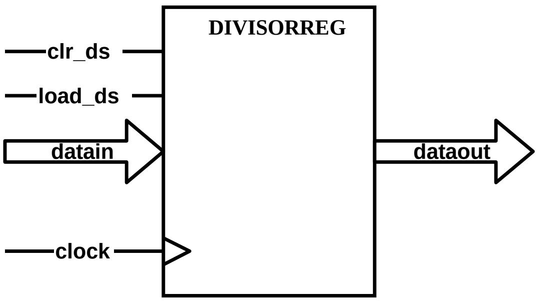

Figure 7 *Divisor Register.*

In the next page appears the program for this register.

```
library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all;

entity divisorreg is

        port(

                load,

                clr,

                clock : in std_logic;

                datain : in unsigned(7 downto 0);

                dataout : out unsigned(7 downto 0)

        );

end divisorreg;


architecture v1 of divisorreg is

        signal q : unsigned(7 downto 0);

begin

        process begin

                wait until rising_edge(clock);

                if clr = '1' then

                        q <= (others => '0');

                elsif load = '1' then

                        q <= datain;

                end if;

        end process;

        dataout <= q;

end v1;
```

1.3.3 Quotient register

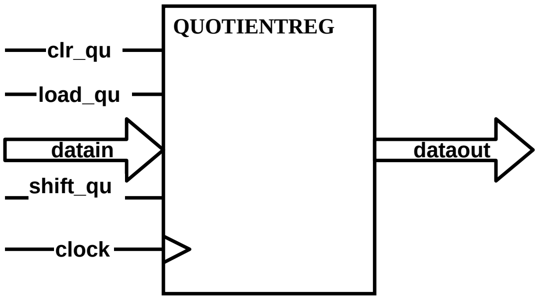

Figure 8 *Quotient Register.*

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity quotientreg is
	port(
		load,
		clr,
		shift,
		clock,
		datain : in std_logic;
		dataout : out unsigned(7 downto 0)
	);
end quotientreg;


architecture v1 of quotientreg is
	signal q : unsigned(7 downto 0);
begin
	process begin
		wait until rising_edge(clock);
		if clr = '1' then
			q <= (others => '0');
		elsif load = '1' then
			q(0) <= not(datain);
```

```
            elsif shift = '1' then
                  q <= shift_left(q, 1);
            end if;
      end process;
      dataout <= q;
end v1;
```

1.3.4 Remainder register

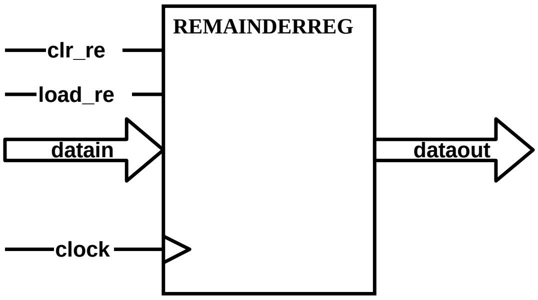

Figure 9 *Remainder Register.*

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity remainderreg is
      port(
            load,
            clr,
            clock : in std_logic;
            datain : in unsigned(7 downto 0);
            dataout : out unsigned(7 downto 0)
      );
```

```
end remainderreg;

architecture v1 of remainderreg is

        signal q : unsigned(7 downto 0);

begin

        process begin

                wait until rising_edge(clock);

                if clr = '1' then

                        q <= (others => '0');

                elsif load = '1' then

                        q <= datain;

                end if;

        end process;

        dataout <= q;

end v1;
```

1.3.5 Accumulator register

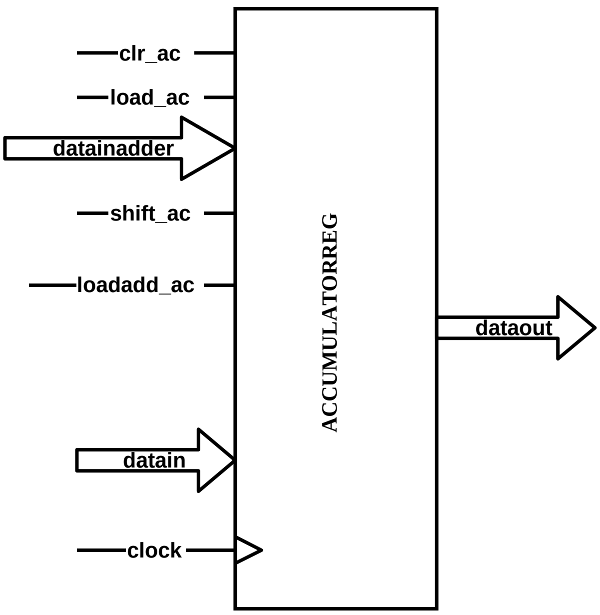

Figure 10 *Accumulator Register.*

```
library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all;

entity accumulatorreg is

        port(

                load,

                clr,

                shift,

                clock,

                loadadd : in std_logic;

                datain : in unsigned(7 downto 0);

                datainadder : in unsigned(8 downto 0);

                dataout : out unsigned(8 downto 0)

        );

end accumulatorreg;


architecture v1 of accumulatorreg is

        signal q : unsigned(16 downto 0);

begin

        process begin

                wait until rising_edge(clock);

                if clr = '1' then

                        q <= (others => '0');

                elsif load = '1' then

                        q (7 downto 0) <= datain;

                elsif loadadd = '1' then

                        q (16 downto 8) <= datainadder;

                elsif shift = '1' then

                        q <= shift_left(q, 1);
```

```
                end if;
        end process;
        dataout <= q (16 downto 8);
end v1;
```

1.3.6 Controller

Here is described the part of the counter,

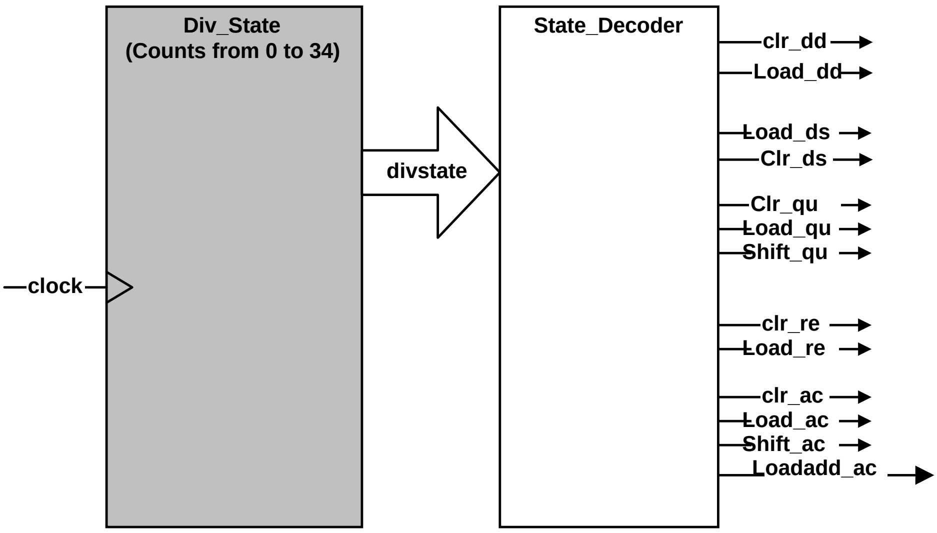

Figure 11 *Controller-Counter.*

```
library ieee;
use ieee.std_logic_1164.all;
entity controller is
        port(
                clock,
                reset,
                sign        : in std_logic;
                clr_dd,                 --dividend reg control
                load_dd,
                clr_ds,                 -- divisor reg control
                load_ds,
```

```
                    clr_qu,                 -- quotient reg control
                    load_qu,
                    shift_qu,
                    clr_re,                 -- remainder reg control
                    load_re,
                    clr_ac,                 -- accumulator reg control
                    load_ac,
                    shift_ac,
                    loadadd_ac,
                    cin,
                    done : out std_logic
          );
end controller;

architecture v1 of controller is
          signal divstate : natural range 0 to 34;
begin
          --state counter
          div_state : process begin
                    wait until rising_edge(clock);
                    if reset = '1' then
                              divstate <= 0;
                    elsif divstate < 34 then
                              divstate <= divstate + 1;
                    end if;
          end process;
```

Now the decoder,

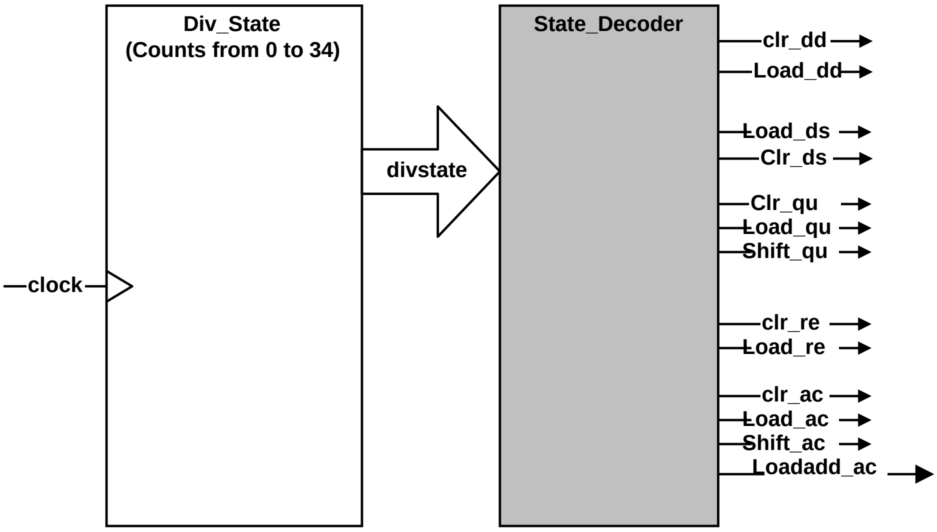

Figure 12 *Controller-Decoder.*

```
--state decoder (combinational logic)

        decoder : process(divstate)

        begin

                clr_dd     <= '0';

                load_dd    <= '0';

                clr_ds     <= '0';

                load_ds    <= '0';

                clr_qu     <= '0';

                load_qu  <= '0';

                shift_qu  <= '0';

                clr_re     <= '0';

                load_re   <= '0';

                clr_ac     <= '0';

                load_ac   <= '0';

                shift_ac    <= '0';

                loadadd_ac    <= '0';

                done   <= '0';

                case divstate is

                        when 0 =>
```

```
                load_dd <= '1';

                load_ds <= '1';

                clr_qu <= '1';

                clr_re <= '1';

                clr_ac <= '1';

            when 1 =>

                load_ac <= '1';

            when 2|6|10|14|18|22|26|30 =>

                shift_ac <= '1';

                cin <= '1';

            when 3|7|11|15|19|23|27|31 =>

                loadadd_ac <= '1';

            when 4|8|12|16|20|24|28|32    =>

                cin <= '0';
                load_qu <= '1';

            when 5|9|13|17|21|25|29|33    =>

                if sign = '1' then

                    loadadd_ac <= '1';

                end if;
                if divstate < 33 then shift_qu <= '1';

                end if;

            when 34 =>

                load_re <= '1';

                done <= '1';

        end case;

    end process;

end v1;
```

1.3.7 Unsigned Divider

Here is the description of the RTL Divider.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity divider8rtl is
        port(   dividend: in unsigned(7 downto 0);
                divisor : in unsigned(7 downto 0);
                quotient : out unsigned(7 downto 0);
                remainder : out unsigned(7 downto 0);
                clock, reset : in std_logic;
                done : out std_logic);
end divider8rtl;

architecture rtl of divider8rtl is

        component dividendreg
                port(
                        load,
                        clr,
                        clock : in std_logic;
                        datain : in unsigned(7 downto 0);
                        dataout : out unsigned(7 downto 0)
                        );
        end component;

        component divisorreg
                port(
                        load,
```

```
            clr,
            clock : in std_logic;
            datain : in unsigned(7 downto 0);
            dataout : out unsigned(7 downto 0)
            );
end component;

component quotientreg
      port(
            load,
            clr,
            shift,
            clock,
            datain : in std_logic;
            dataout : out unsigned(7 downto 0)
            );
end component;

component remainderreg
      port(
            load,
            clr,
            clock : in std_logic;
            datain : in unsigned(7 downto 0);
            dataout : out unsigned(7 downto 0)
            );
end component;
```

```
component accumulatorreg
        port(
                load,
                clr,
                shift,
                clock,
                loadadd : in std_logic;
                datain : in unsigned(7 downto 0);
                datainadder : in unsigned(8 downto 0);
                dataout : out unsigned(8 downto 0)
                );
end component;

component controller
        port(
                clock,
                reset,
                sign        : in std_logic;
                clr_dd,
                load_dd,
                clr_ds,
                load_ds,
                clr_qu,
                load_qu,
                shift_qu,
                clr_re,
                load_re,
                clr_ac,
                load_ac,
                shift_ac,
                loadadd_ac,
```

```
                cin,
                done : out std_logic
                );
        end component;

        signal dividendrego, divisorego : unsigned(7 downto 0);
        signal adderout : unsigned(8 downto 0);
        signal adderin : unsigned(8 downto 0);
        signal accumulatorego : unsigned(16 downto 8);
        signal cin : std_logic;
        signal load_dd ,clr_dd : std_logic;
        signal load_ds ,clr_ds : std_logic;
        signal load_qu ,clr_qu ,shift_qu : std_logic;
        signal load_re ,clr_re : std_logic;
        signal load_ac ,clr_ac, shift_ac ,loadadd_ac : std_logic;

begin

        --dividend register
        dividend_reg : dividendreg
            port map(
                load => load_dd,
                clr => clr_dd,
                clock => clock,
                datain => dividend,
                dataout => dividendrego
                );
```

```
--divisor register
divisor_reg : divisorreg
        port map(
                load => load_ds,
                clr => clr_ds,
                clock => clock,
                datain => divisor,
                dataout => divisorego
                );

--quotient register
quotient_reg : quotientreg
        port map(
                load => load_qu,
                clr => clr_qu,
                shift => shift_qu,
                clock => clock,
                datain => accumulatorego(16),
                dataout => quotient
                );

--remainder register
remainder_reg : remainderreg
        port map(
                load => load_re,
                clr => clr_re,
                clock => clock,
                datain => accumulatorego(15 downto 8),
                dataout => remainder
                );
```

```
--accumulator register
accumulator_reg : accumulatorreg
        port map(
                load => load_ac,
                clr => clr_ac,
                shift => shift_ac,
                loadadd => loadadd_ac,
                clock => clock,
                datain => dividendrego,
                datainadder => adderout,
                dataout => accumulatorego
                );

--controller
cont : controller
        port map(
                clock => clock,
                reset => reset,
                sign => accumulatorego(16),
                clr_dd => clr_dd,
                load_dd => load_dd,
                clr_ds => clr_ds,
                load_ds => load_ds,
                clr_qu => clr_qu,
                load_qu   => load_qu,
                shift_qu => shift_qu,
                clr_re => clr_re,
                load_re    => load_re,
                clr_ac => clr_ac,
```

```
                    load_ac => load_ac,
                    shift_ac => shift_ac,
                    loadadd_ac => loadadd_ac,
                    cin => cin,
                    done => done
                    );
            --adder
            adderin <= ('0' & divisorego);
            adderout <= (accumulatorego(16 downto 8)) + (not(adderin) + 1) when cin = '1' else
                        (accumulatorego(16 downto 8)) + (adderin);

    end rtl;
```

This is the description of the adder

1.4 Simulation

In this section is shown the simulation made with the previous code, when everything was compiled. The number of states described in the controller to complete the division is 34. The loop of the divider has been implemented in 4 states.

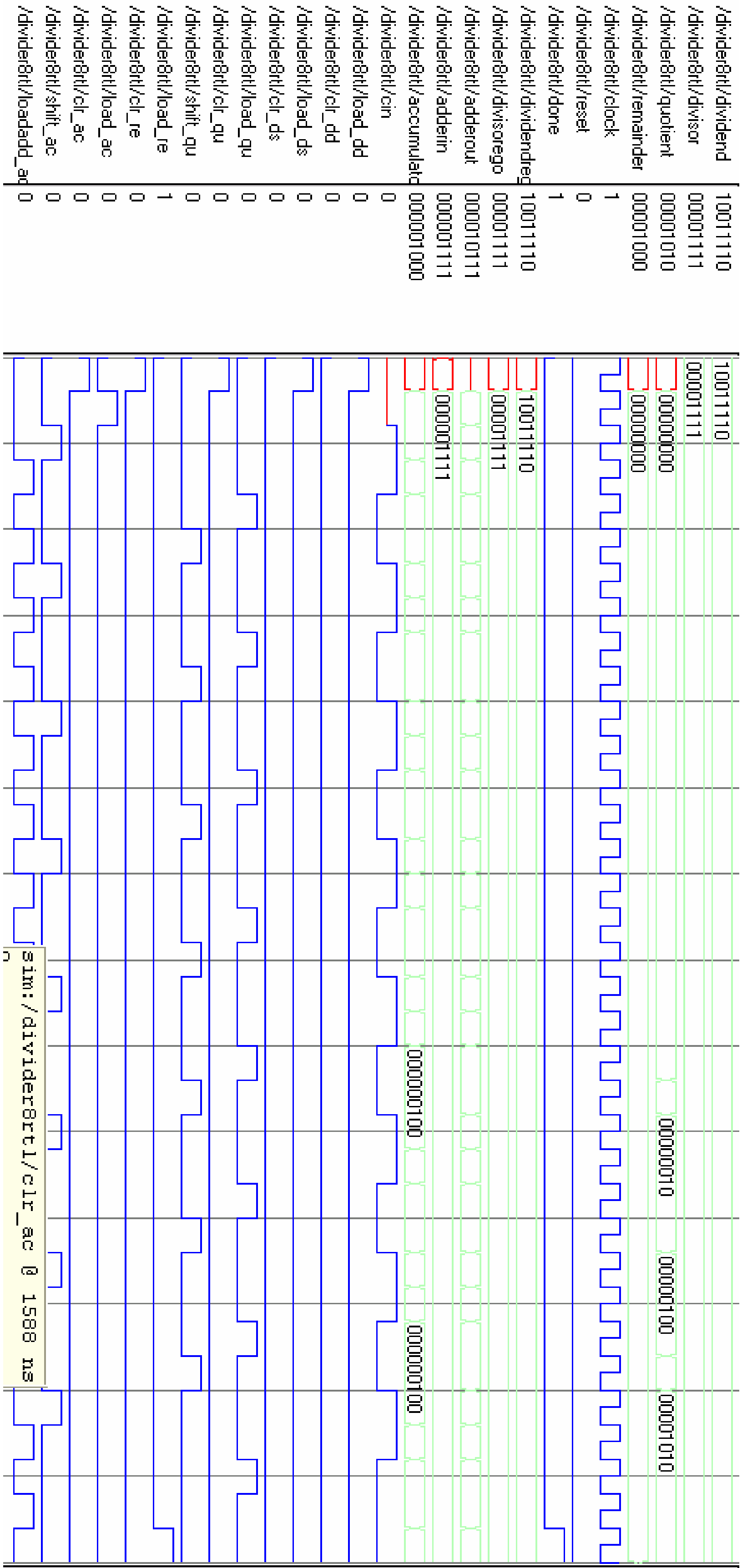

Figure 13 *Simulation Window of the divider.*

As it can be seen in the simulation window all the signals of the divider are represented, and everyone is working properly at correct time. Also it can be observed the final result in the quotient register and the remainder, this example is the first one when the dividend is 158 and the divisor is 15, that produces a quotient of 10 and a remainder of 8. In the next page is shown another example with different numbers, 224 as a dividend and the divisor is 24, that produces a quotient of 9 and a remainder of 8.

With the next window simulation is proved that the system is working properly, the last appendix corresponds to the lab sessions.

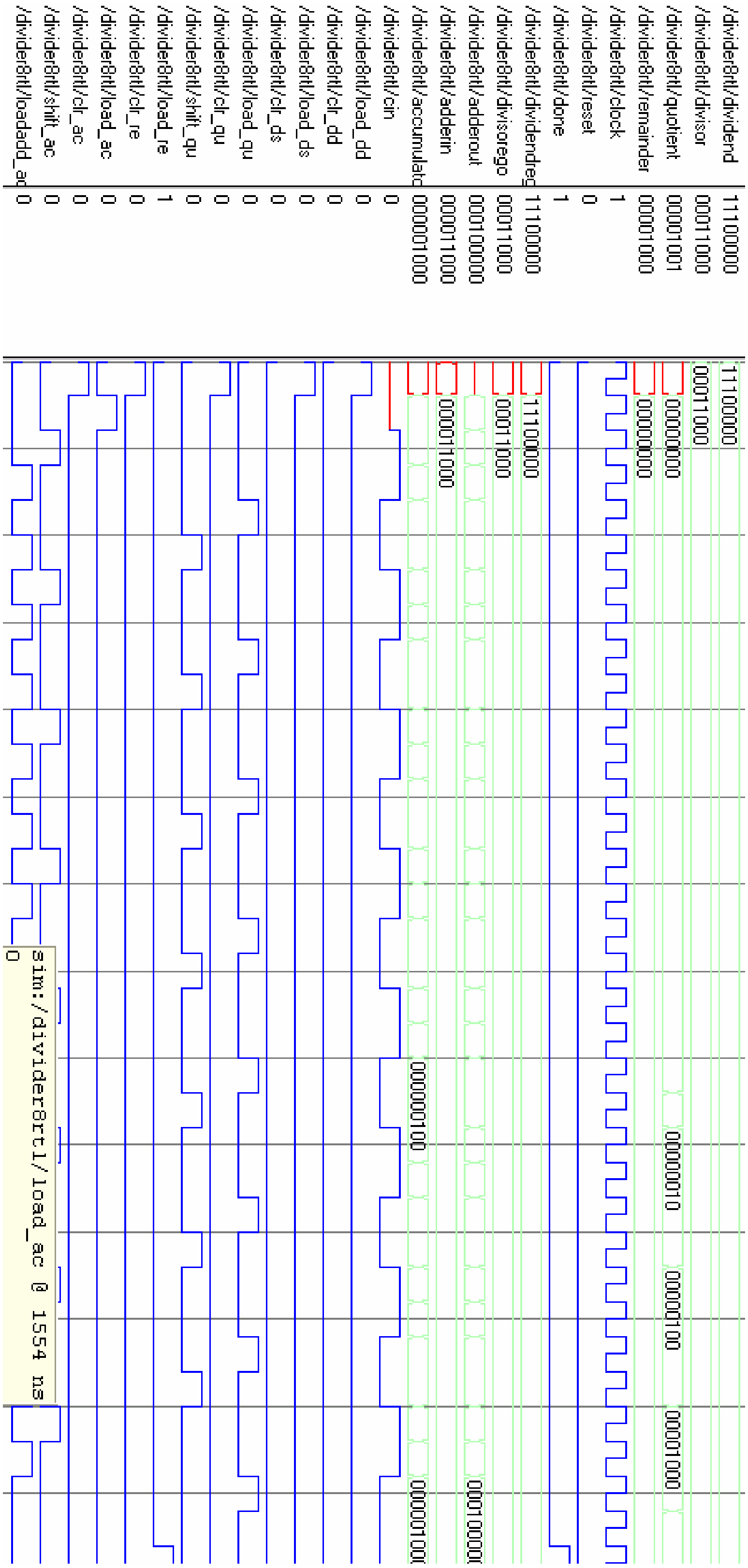

Figure 14 *Simulation Window of the divider.*

Appendix I

1. Lab Session 1

The first design is a two input exclusive-or gate, so here is paste the code in VHDL,

```
-- 2 input exclusive-or gate.
-- Modeled at the structural level.

entity and_gate is
 port (
        a  : in  bit  ;
        b  : in  bit  ;
        c  : out bit) ;
end and_gate;

architecture behavior of and_gate is
begin
   process(a,b)
   begin
        c <= a and b after 15 ns;
   end process;
end behavior;

entity or_gate is
 port (
        d  : in  bit  ;
        e  : in  bit  ;
        f  : out bit) ;
end or_gate;
```

```
architecture behavior of or_gate is
begin
   process(d,e)
         begin
         f <= d or e after 4 ns;
   end process;
end behavior;


entity inverter is
 port (
         g  : in  bit  ;
         h  : out bit) ;
end inverter;


architecture behavior of inverter is
begin
   process(g)
   begin
         h <= not g after 3 ns;
   end process;
end behavior;


entity x_or is
 port (
         in1  : in  bit  ;
         in2  : in  bit  ;
         out1 : out bit);
end x_or;
```

```
architecture structural of x_or is

   -- signal declarations
   signal t1, t2, t3, t4 : bit;

   -- local component declarations
   component and_gate
         port (a, b : in bit; c : out bit) ;
   end component;

   component or_gate
         port (d, e : in bit; f : out bit) ;
   end component;

   component inverter
         port (g : in bit; h : out bit) ;
   end component;

begin

   -- component instantiation statements
   u0: and_gate port map ( a => t1, b => in2, c => t3);

   u1: and_gate port map ( a => in1, b => t2, c => t4);

   u2: inverter port map ( g => in1, h => t1);

   u3: inverter port map ( g => in2, h => t2);

   u4: or_gate  port map ( d => t3, e => t4, f => out1);
end structural;
```

Now it is shown the code of the test bench of the exclusive or gate,

```
--Test bench for xor gate
entity Test_xor is
end Test_xor;

architecture V1 of Test_xor is
        component x_or
                port(in1, in2 : in bit;
                out1 : out bit);
        end component;

        --local signals
        signal in1, in2, out1 : bit;

begin

        --component instantiation
        dut : x_or port map (in1 => in1,  in2 => in2,  out1 => out1);

        --generate some input waveforms
                process
                begin
                        in1 <= '0'; in2 <= '0';
                        wait for 100 ns;
                        in1 <= '1';
                        wait for 100 ns;
                        in2 <= '1';
                        wait for 100 ns;
```

```
                    in1 <= '0';
                    wait for 100 ns;
                    in2 <= '0';
                    wait for 100 ns;
                    assert false report "Simulation Ended"
                          severity failure;
              end process;
end V1;
```

Now is shown the simulation window of the file Test_xor.vhd,

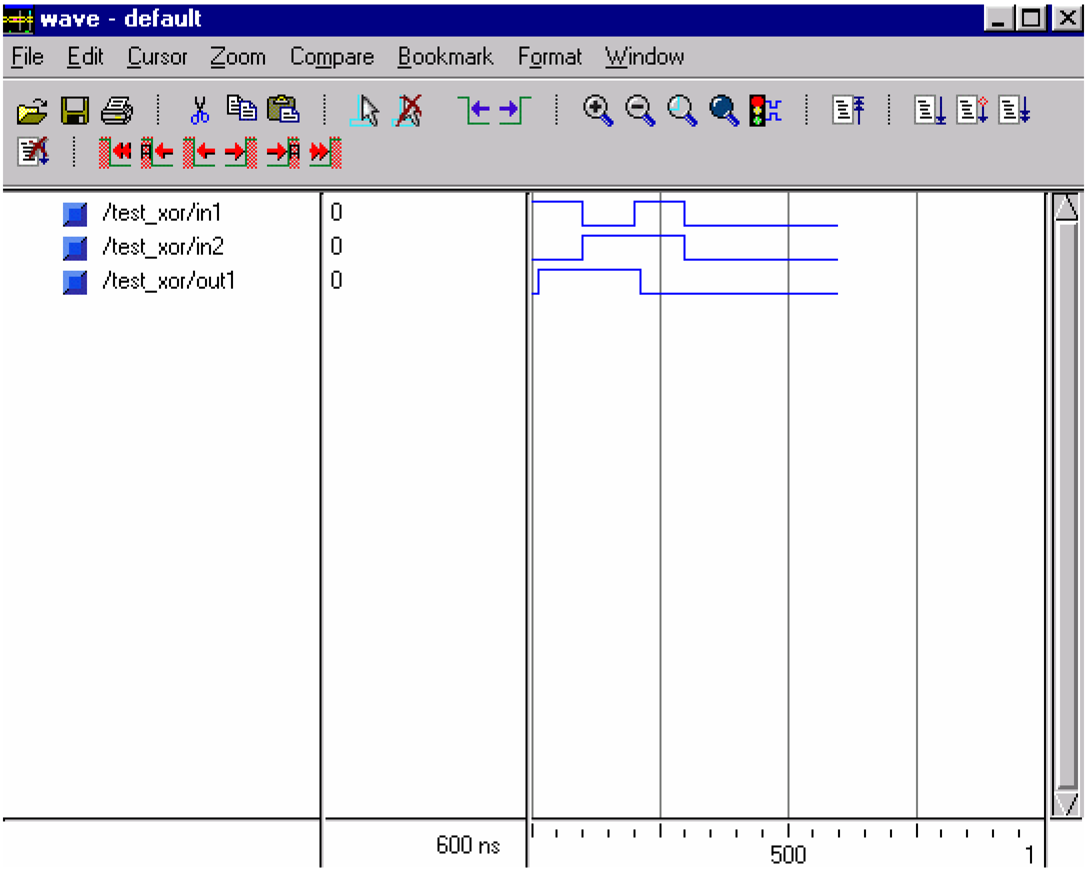

Figure 15 *Simulation Window of the exclusive-or gate.*

Now is illustrated the code for the parity check design,

```
entity parchk is
      port(a,b,c,d,e,f,g,h : in bit;
```

```
                oddbits : out boolean);
end parchk;

architecture v1 of parchk is

        signal w1, w2, w3, w4, w5, w6, w7 : bit;

        component
                x_or port(in1, in2 : in bit;
                        out1 : out bit);
        end component;

begin
                --component instantiations
                x1 : x_or port map (a, b, w1);
                x2 : x_or port map (c, d, w2);
                x3 : x_or port map (e, f, w3);
                x4 : x_or port map (g, h, w4);
                x5 : x_or port map (w1, w2, w5);
                x6 : x_or port map (w3, w4, w6);
                x7 : x_or port map (w5, w6, w7);
                oddbits <= true when  (w7 = '1')  else false;
end v1;
```

The previous code corresponds to the description of the entity and the architecture of the parity check circuit. And the next one is the test bench for this design.

```
entity Test_parchk is
end Test_parchk;

architecture V1 of Test_parchk is

        component parchk
            port(a,b,c,d,e,f,g,h : in bit;
                                        oddbits : out boolean);
        end component;

        signal ODD : boolean;
        signal D : bit_vector(7 downto 0);

begin

        dut : parchk port map (
                    a => D(0),
                    b => D(1),
                    c => D(2),
                    d => D(3),
                    e => D(4),
                    f => D(5),
                    g => D(6),
                    h => D(7),
                    oddbits => ODD);
        --apply inputs
        process
```

```
        begin
                D <= "00000000";
                wait for 100 ns;
                D <= "00000001";
                for i in 1 to 7 loop
                        wait for 100 ns;
                        D <= D(6 downto 0) & '1';
                end loop;
                wait for 100 ns;
                assert false report "Simulation Ended"
                severity failure;
        end process;

end V1;
```

Now in the next page it is illustrated the simulation window of the parity check, the file is tested is test_parchk.vhd.

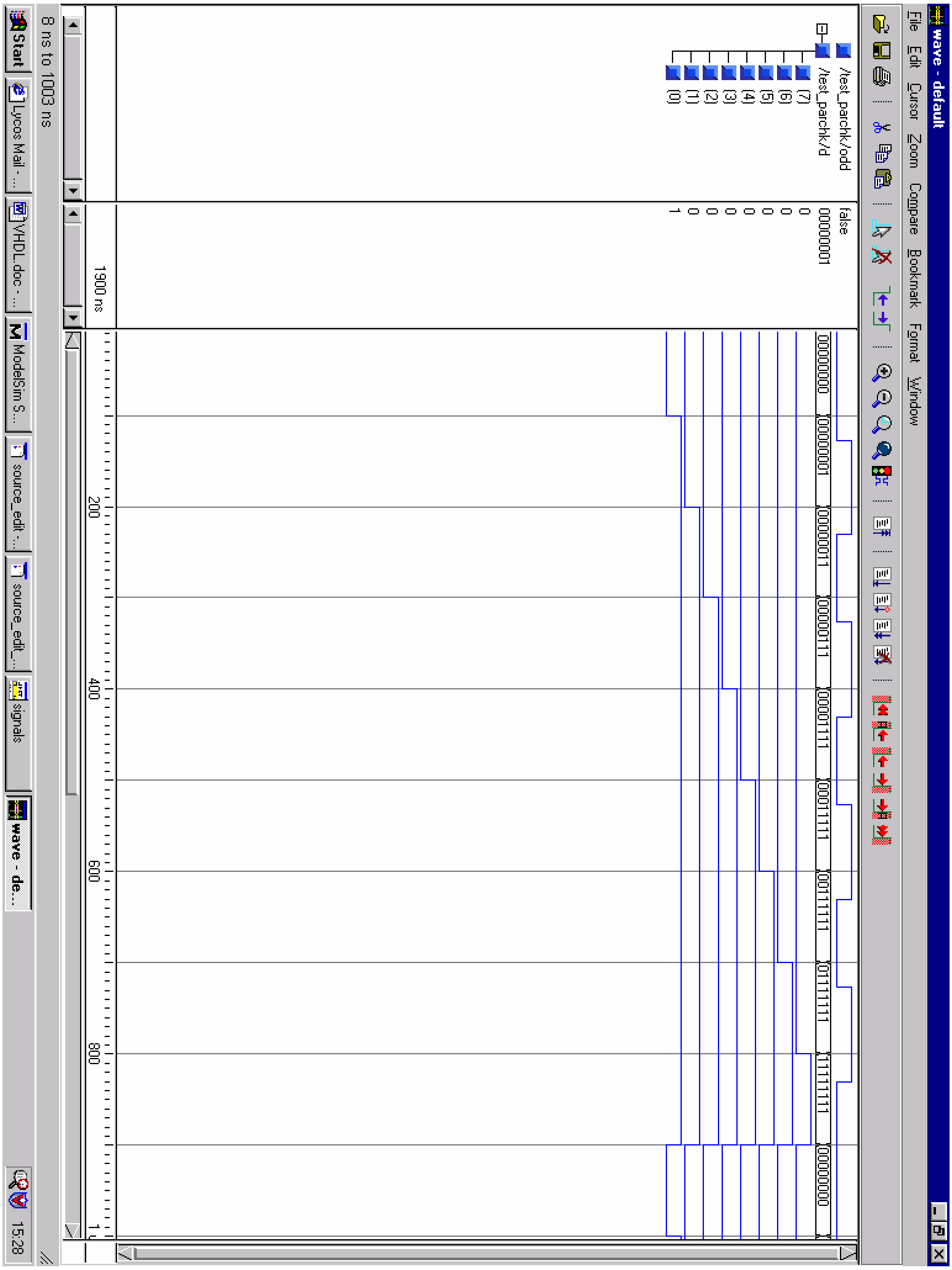

Figure 16 *Simulation Window of the parity check design.*

2. Lab Session 2

In this session was designed an 8 bit multiplier, so there is going to be described each part, only the algorithm and the RTL description are simulated.

- **Algorithm**

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mul8x8a is
end entity mul8x8a;

architecture algorithm of mul8x8a is
begin
        do_it : process
                variable pp : unsigned(16 downto 0);
                variable multiplier : unsigned(7 downto 0);
                variable multiplicand : unsigned(7 downto 0);
                variable product : unsigned(15 downto 0);

          begin
                multiplier := "10001111";  --181
                multiplicand := "11010111"; --118 (prod is 21358)
                pp := (others => '0');
                product := (others => '0');
                  for k in 0 to 7 loop
                        if multiplier(k) = '1' then
                        pp(16 downto 8) :=  ('0' & pp(15 downto 8)) +  ('0' & multiplicand);
                        end if;
```

```
                    pp := shift_right(pp, 1);
                end loop;
                product := pp(15 downto 0);
               assert false report "simulation ended" severity failure;

        end process;
end architecture algorithm;
```

And the simulation for this algorithm when the multiplier is 181, the multiplicand is 118 and the product is 21358 becomes as follows in the next figure.

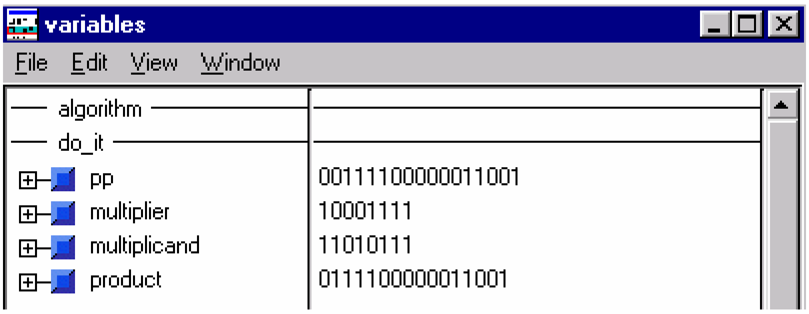

Figure 17 *Algorithm of the Multiplier.*

- **RTL description for the multiplier**

All the registers of the multiplier are described as it is illustrated in the next figure.

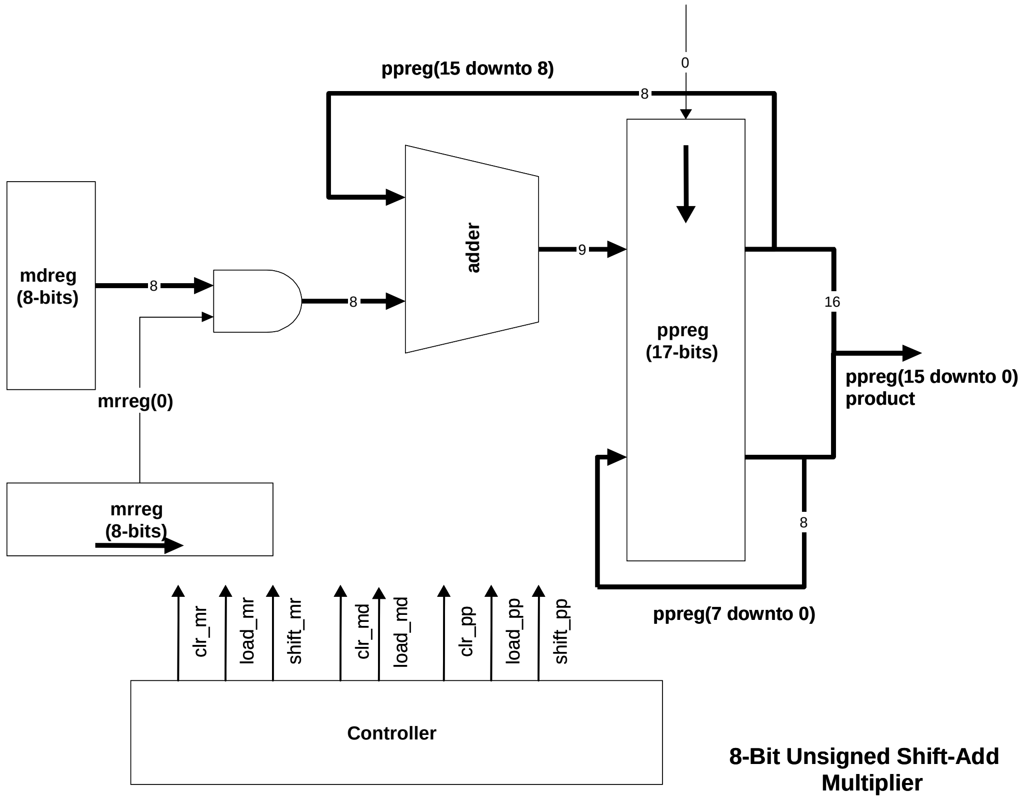

Figure 18 *Block diagram of the Multiplier.*

Multiplicand Register

```
library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all;

entity mdreg is

        port(

                load,

                clr,

                clock : in std_logic;

                datain : in unsigned(7 downto 0);

                dataout : out unsigned(7 downto 0)

                );

end mdreg;
```

```
architecture v1 of mdreg is
        signal q : unsigned(7 downto 0);
begin
        process begin
            wait until rising_edge(clock);
            if clr = '1' then
                q <= (others => '0');
            elsif load = '1' then
                q <= datain;
            end if;
        end process;
        dataout <= q;
end v1;
```

Multiplier Register

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mrreg is
        port(
            load,
            clr,
            shift,
            clock : in std_logic;
            datain : in unsigned(7 downto 0);
            dataout : out unsigned(7 downto 0)
            );
end mrreg;
```

```
architecture v1 of mrreg is
        signal q : unsigned(7 downto 0);
begin
        process begin
            wait until rising_edge(clock);
                if clr = '1' then
                    q <= (others => '0');
                elsif load = '1' then
                    q <= datain;
                elsif shift = '1' then
                    q <= shift_right(q, 1);
                end if;
        end process;
        dataout <= q;
end v1;
```

Product Register

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ppreg is
        port(
            load,
            clr,
            shift,
            clock : in std_logic;
            datain : in unsigned(8 downto 0);
            dataout : out unsigned(16 downto 0)
```

```
        );
end ppreg;

architecture v1 of ppreg is
    signal q : unsigned(16 downto 0);
begin
    process begin
        wait until rising_edge(clock);
        if clr = '1' then
            q <= (others => '0');
        elsif load = '1' then
            q(16 downto 8) <= datain;
        elsif shift = '1' then
            q <= shift_right(q, 1);
        end if;
    end process;
    dataout <= q;
end v1;
```

Controller

```
library ieee;
use ieee.std_logic_1164.all;
entity controller is
    port(
        clock,
        reset : in std_logic;
        clr_mr,         --multiplier reg control
        load_mr,
        shift_mr,
```

```
                    clr_pp, -- pp reg control
                    load_pp,
                    shift_pp,
                    clr_md,  --multiplicand reg control
                    load_md,
                    done : out std_logic
                    );
end controller;

architecture v1 of controller is
            signal mulstate : natural range 0 to 17;
begin

            --state counter
            mul_state : process begin
                    wait until rising_edge(clock);
                    if reset = '1' then
                            mulstate <= 0;
                    elsif mulstate < 17 then
                            mulstate <= mulstate + 1;
                    end if;
            end process;
            --state decoder (combinational logic)
            decoder : process(mulstate)
            begin
                    clr_mr <= '0';
                    load_mr <= '0';
                    shift_mr <= '0';
                    clr_pp <= '0';
```

```
            load_pp <= '0';
            shift_pp <= '0';
            clr_md <= '0';
            load_md <= '0';
            done <= '0';
            case mulstate is
                when 0 =>
                    load_mr <= '1';
                    load_md <= '1';
                    clr_pp <= '1';
                when 2|4|6|8|10|12|14|16 =>
                    shift_mr <= '1';
                    shift_pp <= '1';
                when 1|3|5|7|9|11|13|15 =>
                    load_pp <= '1';
                when 17 =>
                    done <= '1';
            end case;
        end process;
end v1;
```

RTL description

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mul8x8rtl is
    port(multiplier : in unsigned(7 downto 0);
        multiplicand : in unsigned(7 downto 0);
```

```
                product : out unsigned(15 downto 0);
                clock, reset : in std_logic;
                done : out std_logic);
end mul8x8rtl;

architecture rtl of mul8x8rtl is

        component mdreg
                port(
                        load_md,
                        clr_md,
                        clock : in std_logic;
                        datain : in unsigned(7 downto 0);
                        dataout : out unsigned(7 downto 0)
                        );
        end component;

        component mrreg
                port(
                        load_mr,
                        clr_mr,
                        shift_mr,
                        clock : in std_logic;
                        datain : in unsigned(7 downto 0);
                        dataout : out unsigned(7 downto 0)
                        );
        end component;

        component ppreg
```

```
        port(
                load_pp,
                clr_pp,
                shift_pp,
                clock : in std_logic;
                datain : in unsigned(8 downto 0);
                dataout : out unsigned(16 downto 0)
                );
end component;

component controller
        port(
                clock, reset : in std_logic;
                clr_mr,
                load_mr,
                shift_mr,
                clr_md,
                load_md,
                clr_pp,
                load_pp,
                shift_pp,
                done : out std_logic
                );
end component;

signal mdrego, mrrego : unsigned(7 downto 0);
signal adderin : unsigned(7 downto 0);
signal adderout : unsigned(8 downto 0);
signal pprego : unsigned(16 downto 0);

signal clr_mr ,load_mr ,shift_mr : std_logic;
```

```
        signal clr_md ,load_md : std_logic;
        signal clr_pp ,load_pp ,shift_pp : std_logic;


begin
        --multiplicand register
        md_reg : mdreg
                port map (
                        load_md => load_md,
                        clr_md => clr_md,
                        clock => clock,
                        datain => multiplicand,
                        dataout => mdrego
                        );


        --multiplier register
        mr_reg : mrreg
                port map (
                        load_mr => load_mr,
                        clr_mr => clr_mr,
                        shift_mr => shift_mr,
                        clock => clock,
                        datain => multiplier,
                        dataout => mrrego
                        );
          --product register
          pp_reg : ppreg
                port map (
                        load_pp => load_pp,
                        clr_pp => clr_pp,
```

```
                shift_pp => shift_pp,
                clock => clock,
                datain => adderout,
                dataout => pprego
                );

        --controller
        cont : controller
            port map (
                clock => clock,
                reset => reset,
                clr_mr => clr_mr,
                load_mr => load_mr,
                shift_mr => shift_mr,
                clr_md => clr_md,
                load_md => load_md,
                clr_pp => clr_pp,
                load_pp => load_pp,
                shift_pp => shift_pp,
                done => done
                );

        --adder
        adderin <= mdrego when mrrego(0) = '1' else (others => '0');
        adderout <= ('0' & pprego(15 downto 8)) + ('0' & adderin);
        --connect pprego to product output
        product <= pprego(15 downto 0);

    end rtl;
```

And finally in the next page is illustrated the simulation of the multiplier.

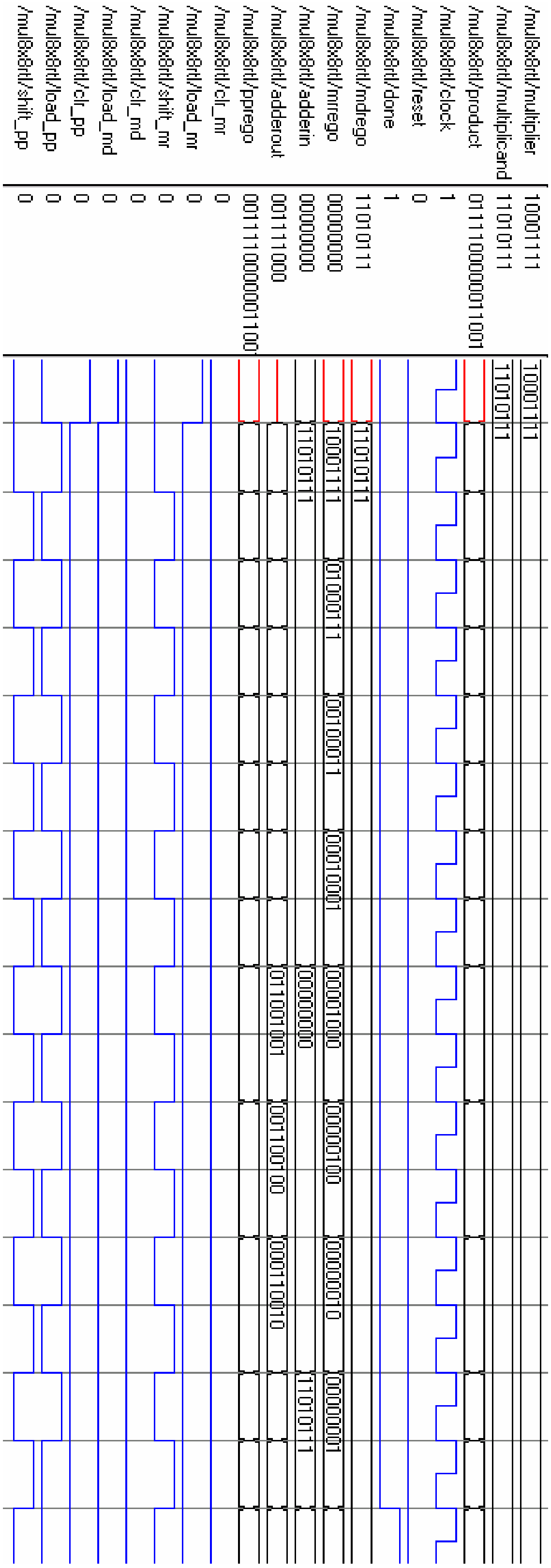

Figure 19 *Simulation of the Multiplier.*