Brunel
UNIVERSITY
WEST LONDON

Name: Honnet
Student n°: 0531984

EE2071 Micro electronic workshop:
Gate level systolic multiplier

Example of chip to implement our multiplier

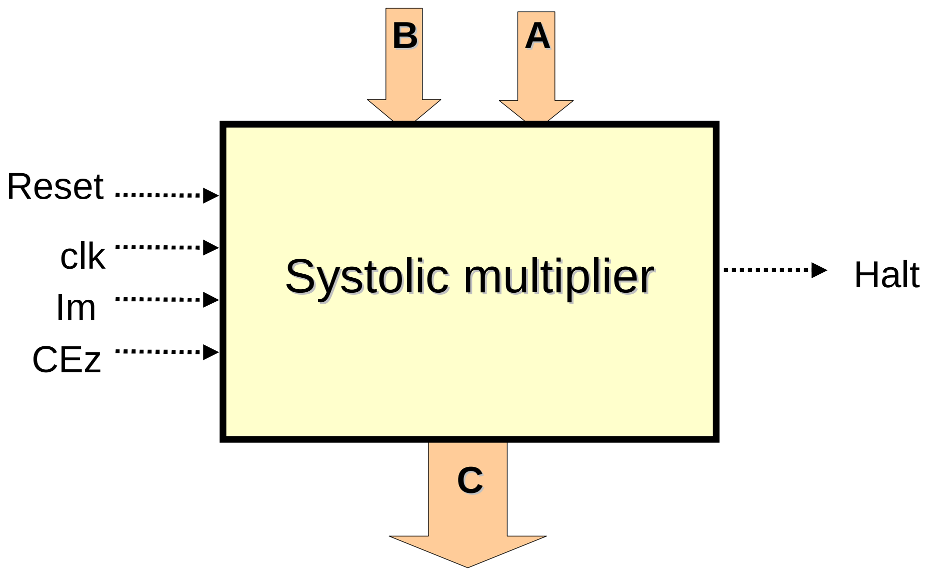
0 ***Purpose***

This laboratory report is our introduction to the principle of manual synthesis for a digital system.
We will try to reproduce a design from Verilog Hardware Description Language (in Register Transfer Level) simulated with the Cadence Simucad Silos software to a graphical gate level using Altera Maxplus.
We are going to meet this challenge by designing a systolic multiplier with two 8bits inputs and a 16bits output.
This multiplier is designed for a digital signal processor and has thus to be able to load 2 input and keep 1 to multiply different values to it.

In a first time we are going to show bloc by bloc the internal components of our design and in a second time we are going to implement the gate level equivalent by reproducing the same behaviour of these components.

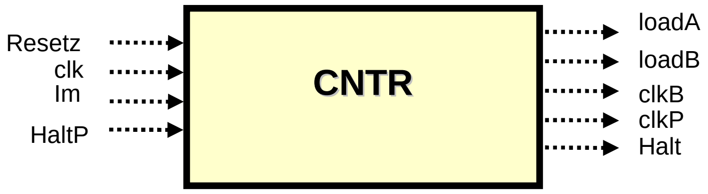
1 ***Verilog multiplier***

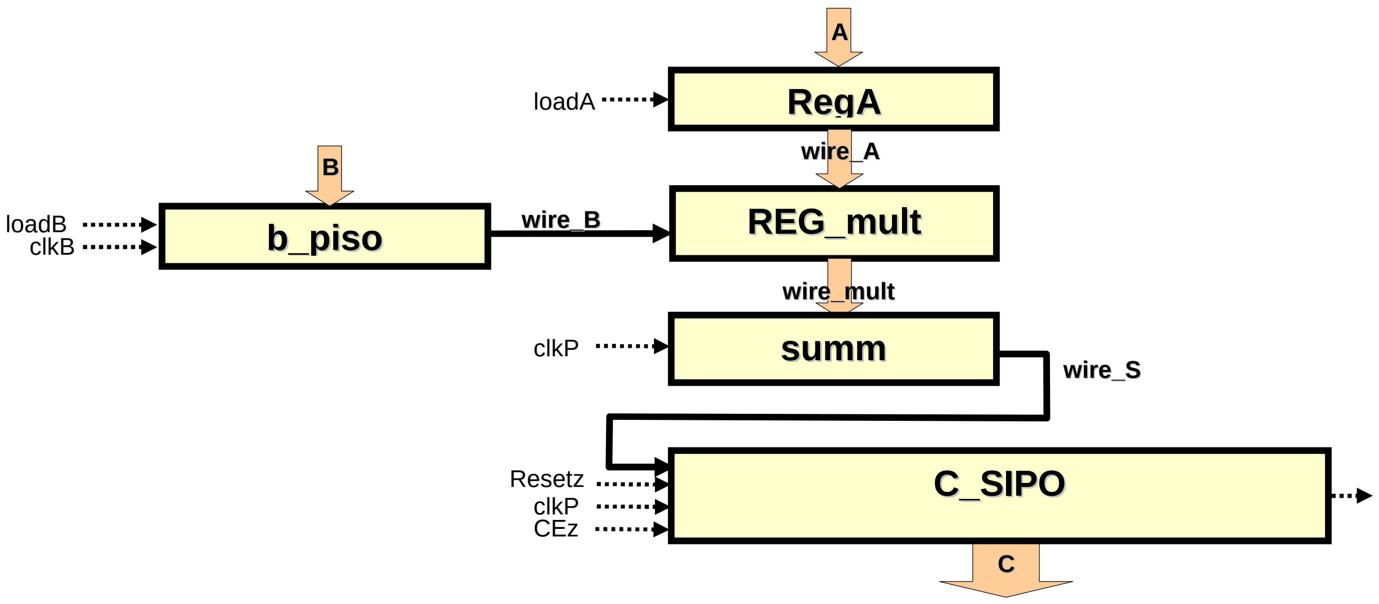
First of all, let's see how this component is interfaced:



Details:
As said previously, A and B are on 8 bits, C is on 16 bits
The signal Im (Input mode) allows selecting if we want to load 1 or 2 inputs (A is not loaded if Im = 0).
The output C is in high impedance state when CEz = 1.
The output Halt is set when the multiplier has completely finished its calculation and is thus ready.

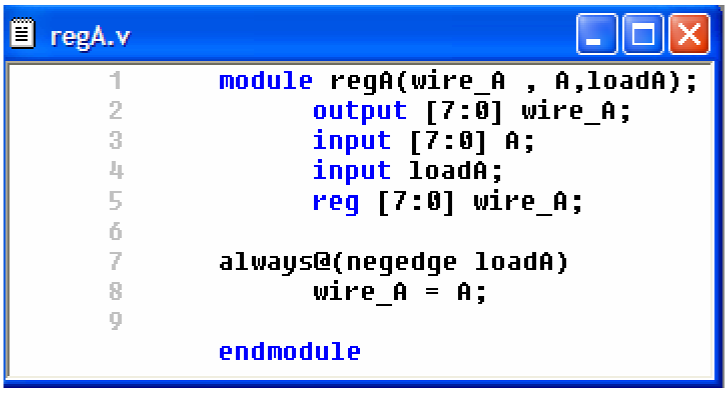
Internal components



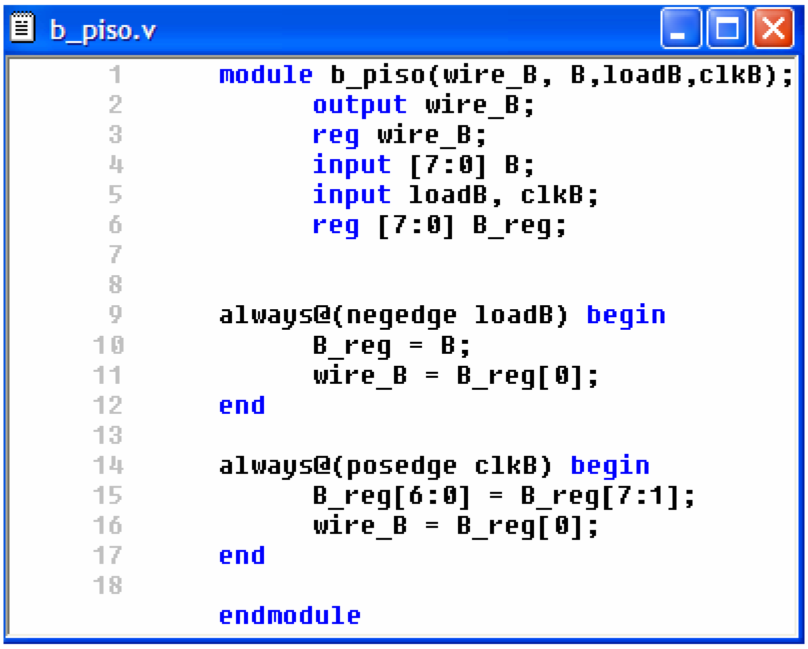


Verilog codes:

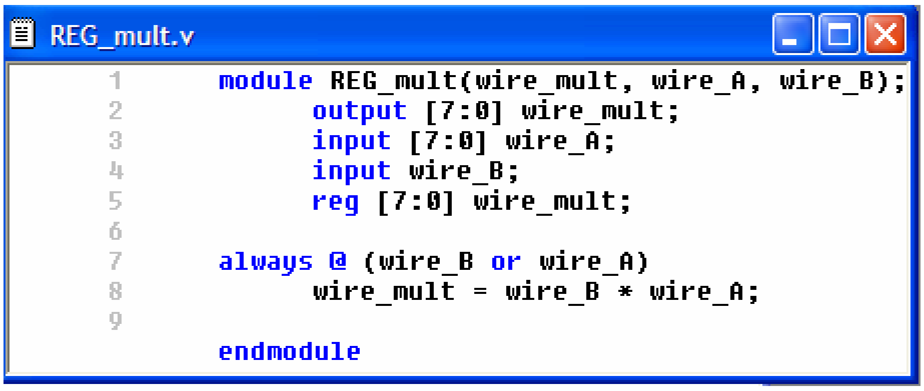
The 3 next components were quite straight forward, I thus detail them too much.



```
module regA(wire_A , A,loadA);
	output [7:0] wire_A;
	input [7:0] A;
	input loadA;
	reg [7:0] wire_A;

always@(negedge loadA)
	wire_A = A;

endmodule
```



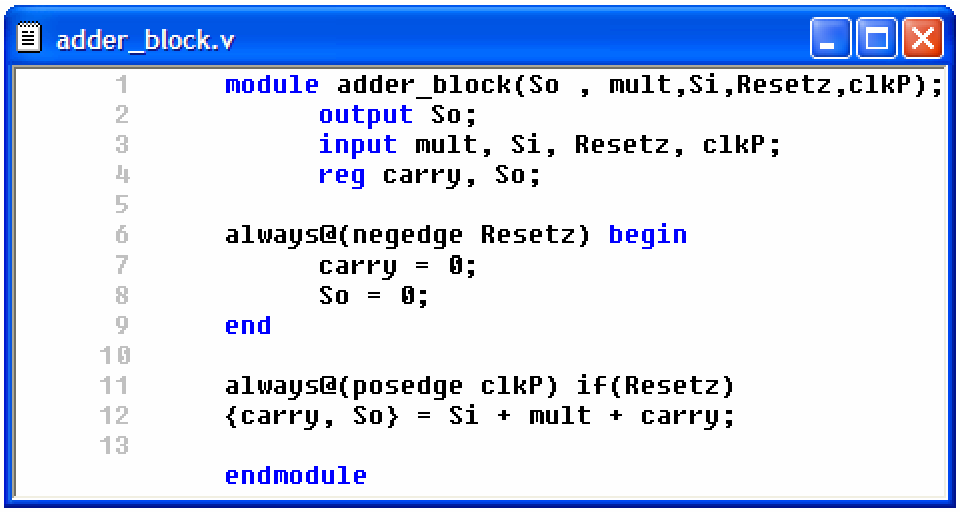
```
module b_piso(wire_B, B,loadB,clkB);
	output wire_B;
	reg wire_B;
	input [7:0] B;
	input loadB, clkB;
	reg [7:0] B_reg;


always@(negedge loadB) begin
	B_reg = B;
	wire_B = B_reg[0];
end

always@(posedge clkB) begin
	B_reg[6:0] = B_reg[7:1];
	wire_B = B_reg[0];
end

endmodule
```



```
module REG_mult(wire_mult, wire_A, wire_B);
	output [7:0] wire_mult;
	input [7:0] wire_A;
	input wire_B;
	reg [7:0] wire_mult;

always @ (wire_B or wire_A)
	wire_mult = wire_B * wire_A;

endmodule
```

In the beginning, I designed a really simple RTL multiplier, but I never succeeded to make it work, I still don't know why. I thus decided to start again and I did it in the same principle than the gate level one:

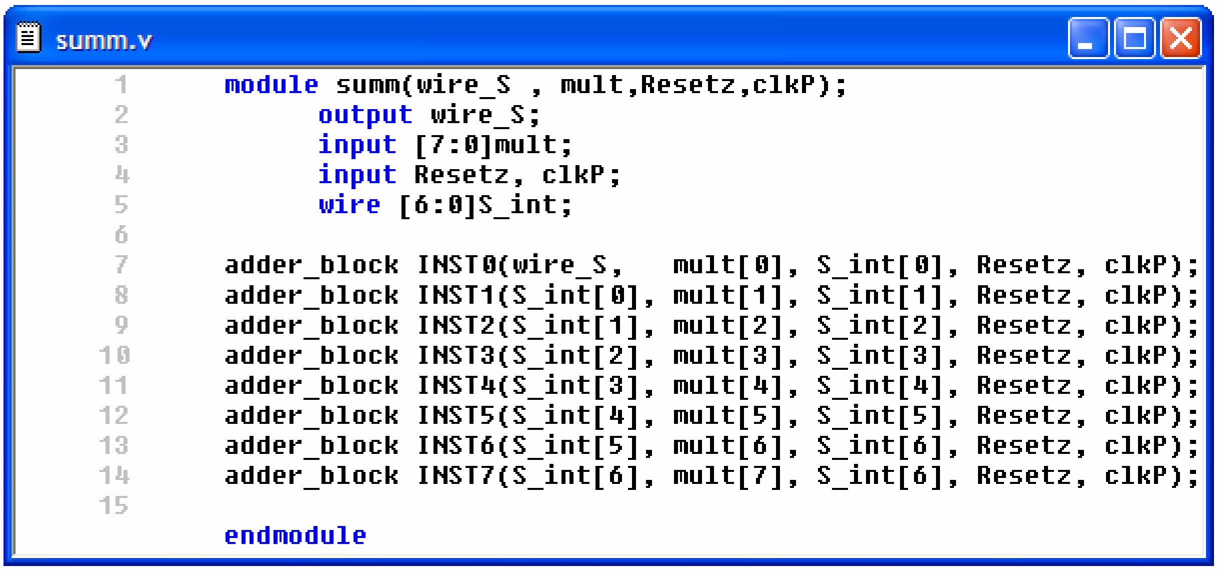
=> It's composed by an adder block (a specialised full adder) where the carry out is fed back in the carry in at the next clock pulse. This is a implicit way to ripple it quickly and efficiently.



adder_block.v

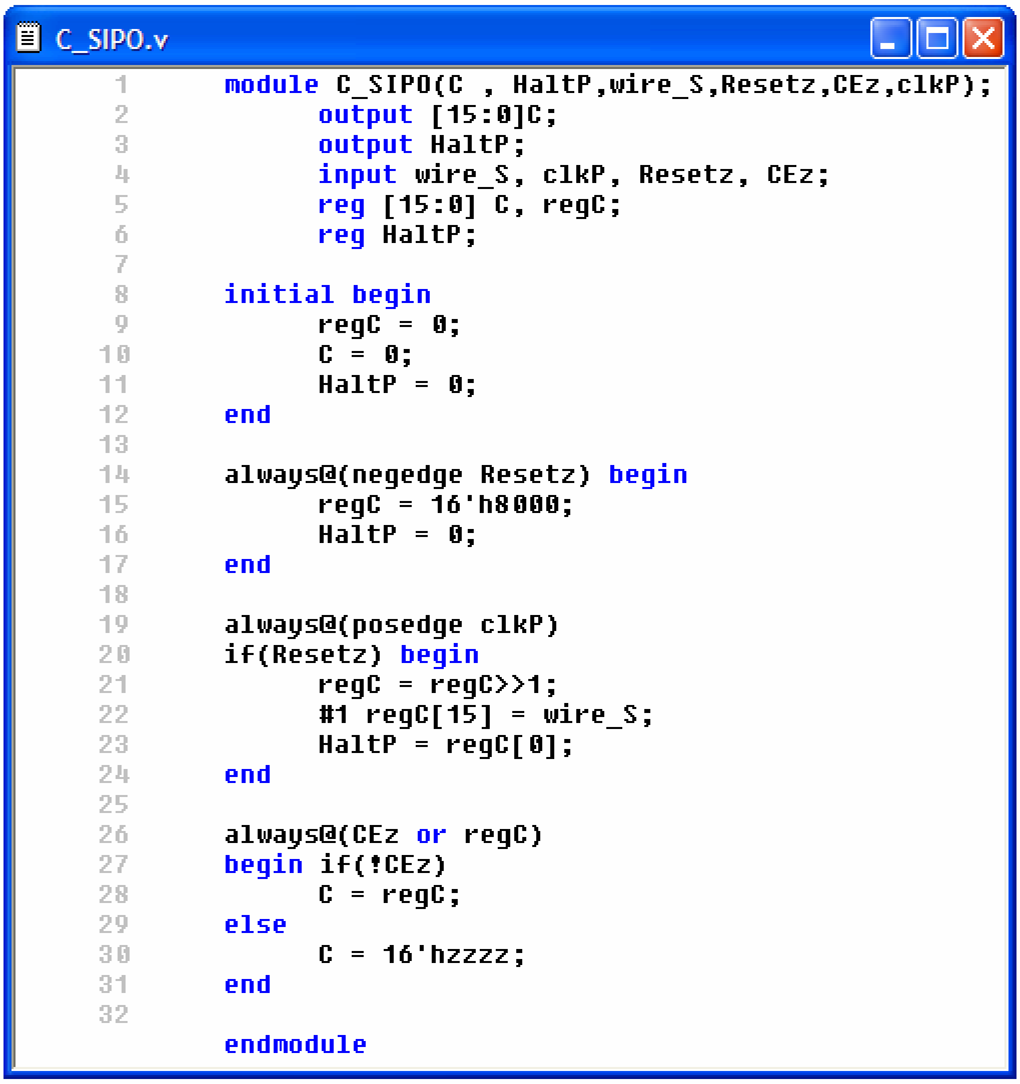
```
module adder_block(So , mult,Si,Resetz,clkP);
      output So;
      input mult, Si, Resetz, clkP;
      reg carry, So;

always@(negedge Resetz) begin
      carry = 0;
      So = 0;
end

always@(posedge clkP) if(Resetz)
{carry, So} = Si + mult + carry;

endmodule
```

…and a module that instantiate it 8 times:

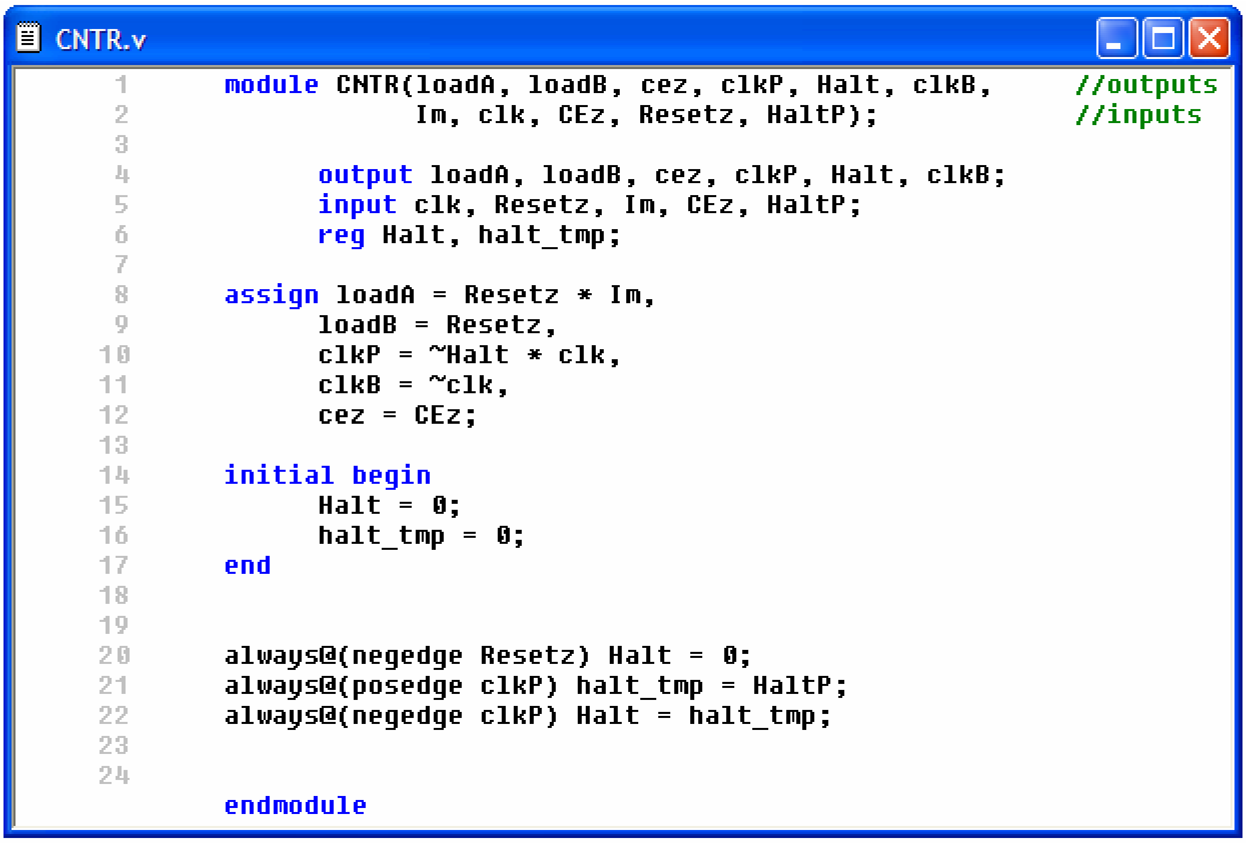


summ.v

```
module summ(wire_S , mult,Resetz,clkP);
      output wire_S;
      input [7:0]mult;
      input Resetz, clkP;
      wire [6:0]S_int;

adder_block INST0(wire_S,   mult[0], S_int[0], Resetz, clkP);
adder_block INST1(S_int[0], mult[1], S_int[1], Resetz, clkP);
adder_block INST2(S_int[1], mult[2], S_int[2], Resetz, clkP);
adder_block INST3(S_int[2], mult[3], S_int[3], Resetz, clkP);
adder_block INST4(S_int[3], mult[4], S_int[4], Resetz, clkP);
adder_block INST5(S_int[4], mult[5], S_int[5], Resetz, clkP);
adder_block INST6(S_int[5], mult[6], S_int[6], Resetz, clkP);
adder_block INST7(S_int[6], mult[7], S_int[6], Resetz, clkP);

endmodule
```

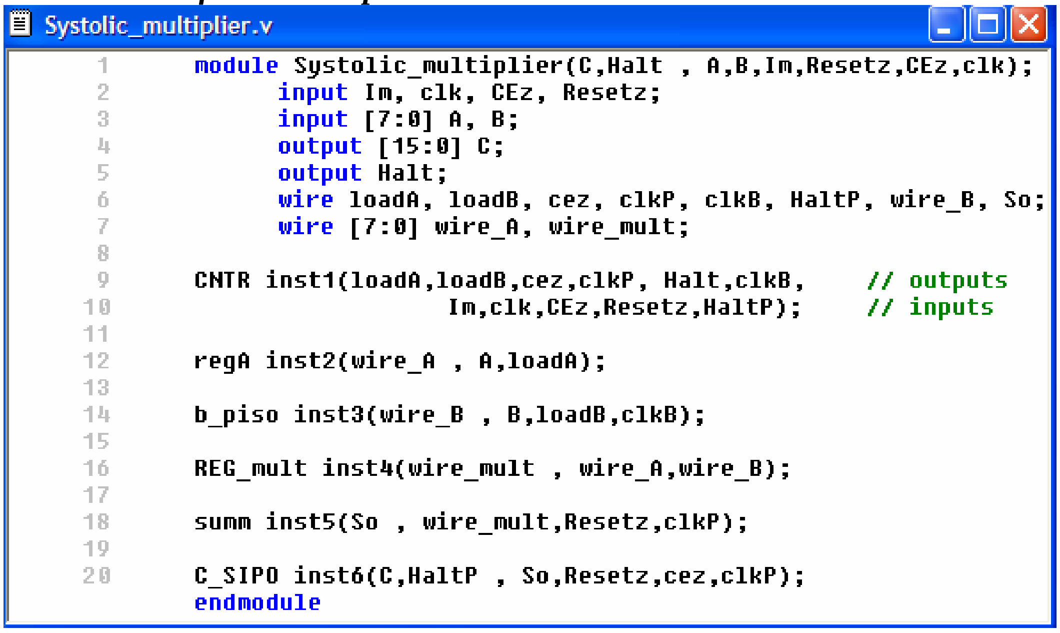


```
module C_SIPO(C , HaltP,wire_S,Resetz,CEz,clkP);
      output [15:0]C;
      output HaltP;
      input wire_S, clkP, Resetz, CEz;
      reg [15:0] C, regC;
      reg HaltP;

initial begin
      regC = 0;
      C = 0;
      HaltP = 0;
end

always@(negedge Resetz) begin
      regC = 16'h8000;
      HaltP = 0;
end

always@(posedge clkP)
if(Resetz) begin
      regC = regC>>1;
      #1 regC[15] = wire_S;
      HaltP = regC[0];
end

always@(CEz or regC)
begin if(!CEz)
      C = regC;
else
      C = 16'hzzzz;
end

endmodule
```



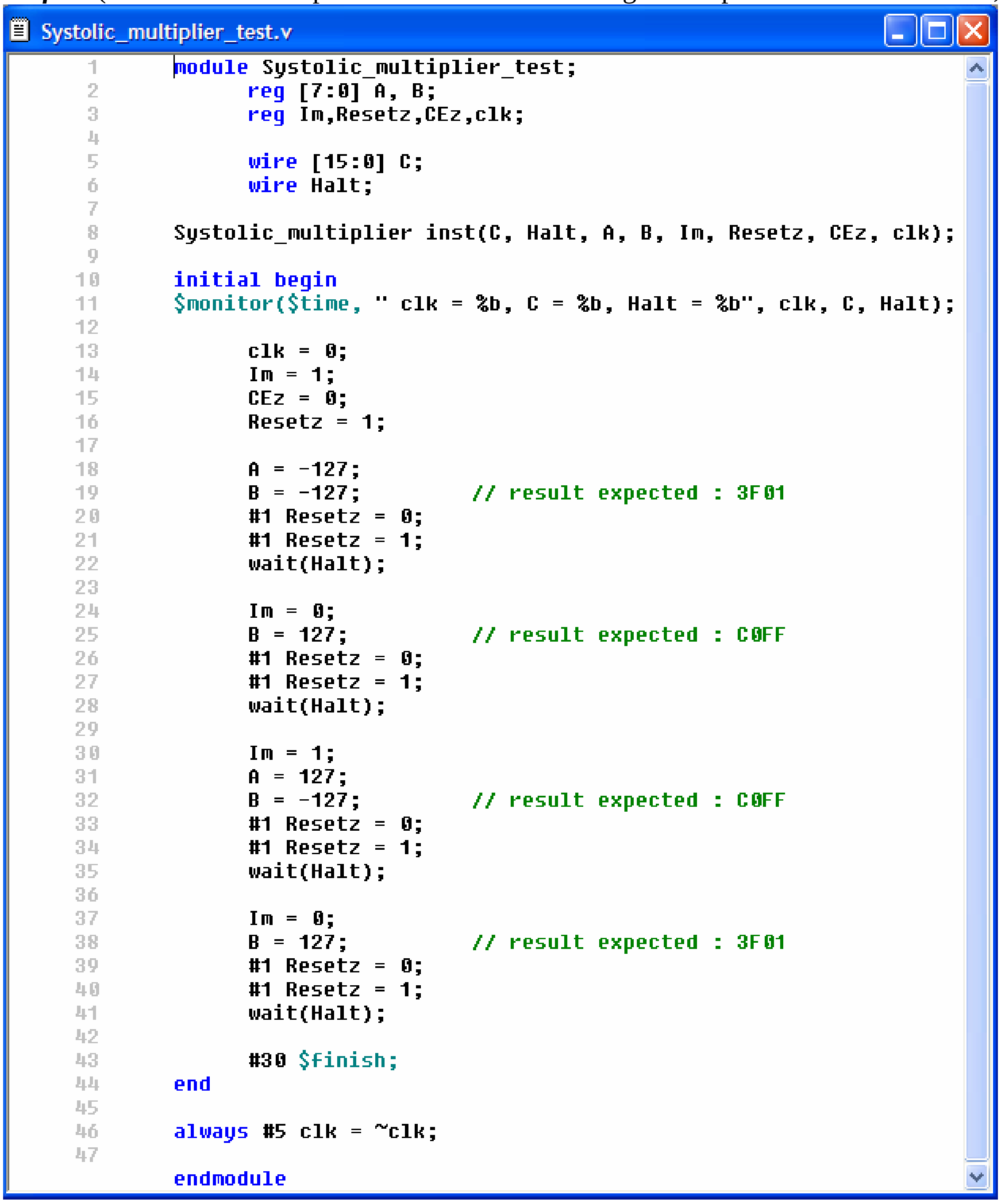
```
module CNTR(loadA, loadB, cez, clkP, Halt, clkB,       //outputs
            Im, clk, CEz, Resetz, HaltP);               //inputs

      output loadA, loadB, cez, clkP, Halt, clkB;
      input clk, Resetz, Im, CEz, HaltP;
      reg Halt, halt_tmp;

assign loadA = Resetz * Im,
      loadB = Resetz,
      clkP = ~Halt * clk,
      clkB = ~clk,
      cez = CEz;

initial begin
      Halt = 0;
      halt_tmp = 0;
end


always@(negedge Resetz) Halt = 0;
always@(posedge clkP) halt_tmp = HaltP;
always@(negedge clkP) Halt = halt_tmp;


endmodule
```

Instantiation of all the components:



```
module Systolic_multiplier(C,Halt , A,B,Im,Resetz,CEz,clk);
      input Im, clk, CEz, Resetz;
      input [7:0] A, B;
      output [15:0] C;
      output Halt;
      wire loadA, loadB, cez, clkP, clkB, HaltP, wire_B, So;
      wire [7:0] wire_A, wire_mult;

CNTR inst1(loadA,loadB,cez,clkP, Halt,clkB,     // outputs
                  Im,clk,CEz,Resetz,HaltP);     // inputs

regA inst2(wire_A , A,loadA);

b_piso inst3(wire_B , B,loadB,clkB);

REG_mult inst4(wire_mult , wire_A,wire_B);

summ inst5(So , wire_mult,Resetz,clkP);

C_SIPO inst6(C,HaltP , So,Resetz,cez,clkP);
endmodule
```

testfile: (this test file is simplified to be able monitoring the output in the result text file, see next page)

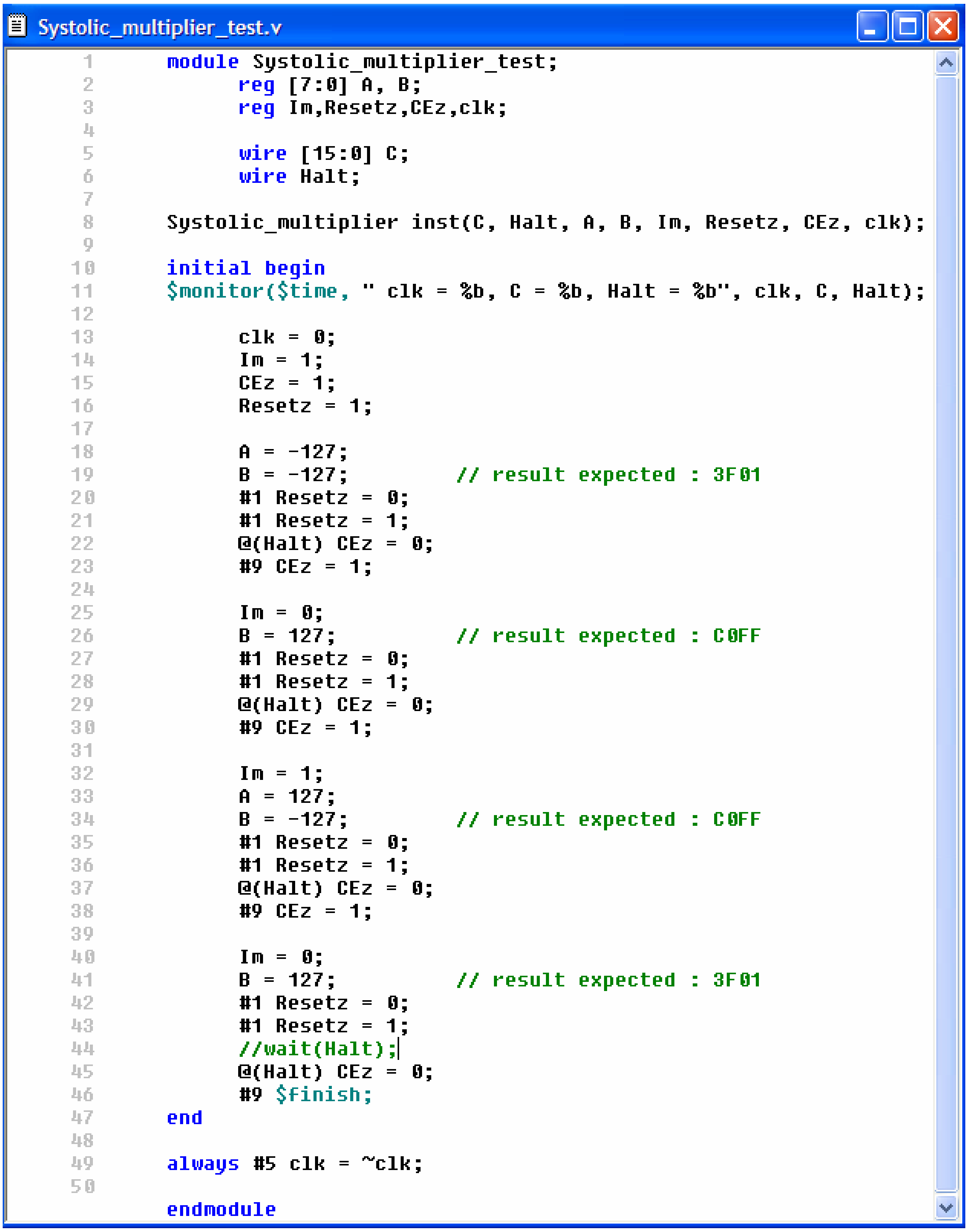


```
module Systolic_multiplier_test;
      reg [7:0] A, B;
      reg Im,Resetz,CEz,clk;

      wire [15:0] C;
      wire Halt;

Systolic_multiplier inst(C, Halt, A, B, Im, Resetz, CEz, clk);

initial begin
$monitor($time, " clk = %b, C = %b, Halt = %b", clk, C, Halt);

      clk = 0;
      Im = 1;
      CEz = 0;
      Resetz = 1;

      A = -127;
      B = -127;         // result expected : 3F01
      #1 Resetz = 0;
      #1 Resetz = 1;
      wait(Halt);

      Im = 0;
      B = 127;          // result expected : C0FF
      #1 Resetz = 0;
      #1 Resetz = 1;
      wait(Halt);

      Im = 1;
      A = 127;
      B = -127;         // result expected : C0FF
      #1 Resetz = 0;
      #1 Resetz = 1;
      wait(Halt);

      Im = 0;
      B = 127;          // result expected : 3F01
      #1 Resetz = 0;
      #1 Resetz = 1;
      wait(Halt);

      #30 $finish;
end

always #5 clk = ~clk;

endmodule
```

For this result text file I thus enabled the chip output (no state Z) to be able to see its evolution:

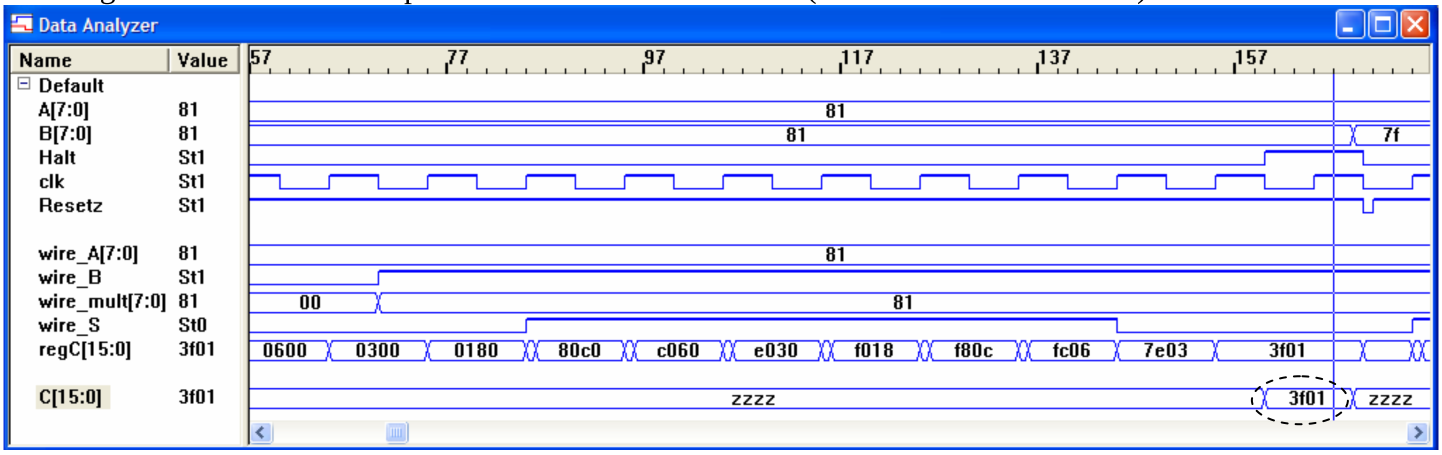
```
  1 clk = 0, C = 1000000000000000, Halt = 0
  5 clk = 1, C = 0100000000000000, Halt = 0
  6 clk = 1, C = 1100000000000000, Halt = 0
 10 clk = 0, C = 1100000000000000, Halt = 0
 15 clk = 1, C = 0110000000000000, Halt = 0
 20 clk = 0, C = 0110000000000000, Halt = 0
 25 clk = 1, C = 0011000000000000, Halt = 0
 30 clk = 0, C = 0011000000000000, Halt = 0
 35 clk = 1, C = 0001100000000000, Halt = 0
 40 clk = 0, C = 0001100000000000, Halt = 0
 45 clk = 1, C = 0000110000000000, Halt = 0
 50 clk = 0, C = 0000110000000000, Halt = 0
 55 clk = 1, C = 0000011000000000, Halt = 0
 60 clk = 0, C = 0000011000000000, Halt = 0
 65 clk = 1, C = 0000001100000000, Halt = 0
 70 clk = 0, C = 0000001100000000, Halt = 0
 75 clk = 1, C = 0000000110000000, Halt = 0
 80 clk = 0, C = 0000000110000000, Halt = 0
 85 clk = 1, C = 0000000011000000, Halt = 0
 86 clk = 1, C = 1000000011000000, Halt = 0
 90 clk = 0, C = 1000000011000000, Halt = 0
 95 clk = 1, C = 0100000001100000, Halt = 0
 96 clk = 1, C = 1100000001100000, Halt = 0
100 clk = 0, C = 1100000001100000, Halt = 0
105 clk = 1, C = 0110000000110000, Halt = 0
106 clk = 1, C = 1110000000110000, Halt = 0
110 clk = 0, C = 1110000000110000, Halt = 0
115 clk = 1, C = 0111000000011000, Halt = 0
116 clk = 1, C = 1111000000011000, Halt = 0
120 clk = 0, C = 1111000000011000, Halt = 0
125 clk = 1, C = 0111100000001100, Halt = 0
126 clk = 1, C = 1111100000001100, Halt = 0
130 clk = 0, C = 1111100000001100, Halt = 0
135 clk = 1, C = 0111110000000110, Halt = 0
136 clk = 1, C = 1111110000000110, Halt = 0
140 clk = 0, C = 1111110000000110, Halt = 0
145 clk = 1, C = 0111111000000011, Halt = 0
150 clk = 0, C = 0111111000000011, Halt = 0
155 clk = 1, C = 0011111100000001, Halt = 0
160 clk = 0, C = 0011111100000001, Halt = 1 => 30F1
161 clk = 0, C = 1000000000000000, Halt = 0
165 clk = 1, C = 0100000000000000, Halt = 0
166 clk = 1, C = 1100000000000000, Halt = 0
170 clk = 0, C = 1100000000000000, Halt = 0
175 clk = 1, C = 0110000000000000, Halt = 0
176 clk = 1, C = 1110000000000000, Halt = 0
180 clk = 0, C = 1110000000000000, Halt = 0
185 clk = 1, C = 0111000000000000, Halt = 0
186 clk = 1, C = 1111000000000000, Halt = 0
190 clk = 0, C = 1111000000000000, Halt = 0
195 clk = 1, C = 0111100000000000, Halt = 0
196 clk = 1, C = 1111100000000000, Halt = 0
200 clk = 0, C = 1111100000000000, Halt = 0
205 clk = 1, C = 0111110000000000, Halt = 0
206 clk = 1, C = 1111110000000000, Halt = 0
210 clk = 0, C = 1111110000000000, Halt = 0
215 clk = 1, C = 0111111000000000, Halt = 0
216 clk = 1, C = 1111111000000000, Halt = 0
220 clk = 0, C = 1111111000000000, Halt = 0
225 clk = 1, C = 0111111100000000, Halt = 0
226 clk = 1, C = 1111111100000000, Halt = 0
230 clk = 0, C = 1111111100000000, Halt = 0
235 clk = 1, C = 0111111110000000, Halt = 0
236 clk = 1, C = 1111111110000000, Halt = 0
240 clk = 0, C = 1111111110000000, Halt = 0
245 clk = 1, C = 0111111111000000, Halt = 0
250 clk = 0, C = 0111111111000000, Halt = 0
255 clk = 1, C = 0011111111100000, Halt = 0
260 clk = 0, C = 0011111111100000, Halt = 0
265 clk = 1, C = 0001111111110000, Halt = 0
270 clk = 0, C = 0001111111110000, Halt = 0
275 clk = 1, C = 0000111111111000, Halt = 0
280 clk = 0, C = 0000111111111000, Halt = 0
285 clk = 1, C = 0000011111111100, Halt = 0
290 clk = 0, C = 0000011111111100, Halt = 0
295 clk = 1, C = 0000001111111110, Halt = 0
300 clk = 0, C = 0000001111111110, Halt = 0
305 clk = 1, C = 0000000111111111, Halt = 0
306 clk = 1, C = 1000000111111111, Halt = 0
310 clk = 0, C = 1000000111111111, Halt = 0
315 clk = 1, C = 0100000011111111, Halt = 0
316 clk = 1, C = 1100000011111111, Halt = 0
320 clk = 0, C = 1100000011111111, Halt = 1 => C0FF
321 clk = 0, C = 1000000000000000, Halt = 0
325 clk = 1, C = 0100000000000000, Halt = 0
326 clk = 1, C = 1100000000000000, Halt = 0
330 clk = 0, C = 1100000000000000, Halt = 0
335 clk = 1, C = 0110000000000000, Halt = 0
336 clk = 1, C = 1110000000000000, Halt = 0
340 clk = 0, C = 1110000000000000, Halt = 0
345 clk = 1, C = 0111000000000000, Halt = 0
346 clk = 1, C = 1111000000000000, Halt = 0
350 clk = 0, C = 1111000000000000, Halt = 0
355 clk = 1, C = 0111100000000000, Halt = 0
356 clk = 1, C = 1111100000000000, Halt = 0
360 clk = 0, C = 1111100000000000, Halt = 0
365 clk = 1, C = 0111110000000000, Halt = 0
366 clk = 1, C = 1111110000000000, Halt = 0
370 clk = 0, C = 1111110000000000, Halt = 0
375 clk = 1, C = 0111111000000000, Halt = 0
376 clk = 1, C = 1111111000000000, Halt = 0
380 clk = 0, C = 1111111000000000, Halt = 0
385 clk = 1, C = 0111111100000000, Halt = 0
386 clk = 1, C = 1111111100000000, Halt = 0
390 clk = 0, C = 1111111100000000, Halt = 0
395 clk = 1, C = 0111111110000000, Halt = 0
396 clk = 1, C = 1111111110000000, Halt = 0
400 clk = 0, C = 1111111110000000, Halt = 0
405 clk = 1, C = 0111111111000000, Halt = 0
410 clk = 0, C = 0111111111000000, Halt = 0
415 clk = 1, C = 0011111111100000, Halt = 0
420 clk = 0, C = 0011111111100000, Halt = 0
425 clk = 1, C = 0001111111110000, Halt = 0
430 clk = 0, C = 0001111111110000, Halt = 0
435 clk = 1, C = 0000111111111000, Halt = 0
440 clk = 0, C = 0000111111111000, Halt = 0
445 clk = 1, C = 0000011111111100, Halt = 0
450 clk = 0, C = 0000011111111100, Halt = 0
455 clk = 1, C = 0000001111111110, Halt = 0
460 clk = 0, C = 0000001111111110, Halt = 0
465 clk = 1, C = 0000000111111111, Halt = 0
466 clk = 1, C = 1000000111111111, Halt = 0
470 clk = 0, C = 1000000111111111, Halt = 0
475 clk = 1, C = 0100000011111111, Halt = 0
476 clk = 1, C = 1100000011111111, Halt = 0
480 clk = 0, C = 1100000011111111, Halt = 1 => C0FF
481 clk = 0, C = 1000000000000000, Halt = 0
485 clk = 1, C = 0100000000000000, Halt = 0
486 clk = 1, C = 1100000000000000, Halt = 0
490 clk = 0, C = 1100000000000000, Halt = 0
495 clk = 1, C = 0110000000000000, Halt = 0
500 clk = 0, C = 0110000000000000, Halt = 0
505 clk = 1, C = 0011000000000000, Halt = 0
510 clk = 0, C = 0011000000000000, Halt = 0
515 clk = 1, C = 0001100000000000, Halt = 0
520 clk = 0, C = 0001100000000000, Halt = 0
525 clk = 1, C = 0000110000000000, Halt = 0
530 clk = 0, C = 0000110000000000, Halt = 0
535 clk = 1, C = 0000011000000000, Halt = 0
540 clk = 0, C = 0000011000000000, Halt = 0
545 clk = 1, C = 0000001100000000, Halt = 0
550 clk = 0, C = 0000001100000000, Halt = 0
555 clk = 1, C = 0000000110000000, Halt = 0
560 clk = 0, C = 0000000110000000, Halt = 0
565 clk = 1, C = 0000000011000000, Halt = 0
566 clk = 1, C = 1000000011000000, Halt = 0
570 clk = 0, C = 1000000011000000, Halt = 0
575 clk = 1, C = 0100000001100000, Halt = 0
576 clk = 1, C = 1100000001100000, Halt = 0
580 clk = 0, C = 1100000001100000, Halt = 0
585 clk = 1, C = 0110000000110000, Halt = 0
586 clk = 1, C = 1110000000110000, Halt = 0
590 clk = 0, C = 1110000000110000, Halt = 0
595 clk = 1, C = 0111000000011000, Halt = 0
596 clk = 1, C = 1111000000011000, Halt = 0
600 clk = 0, C = 1111000000011000, Halt = 0
605 clk = 1, C = 0111100000001100, Halt = 0
606 clk = 1, C = 1111100000001100, Halt = 0
610 clk = 0, C = 1111100000001100, Halt = 0
615 clk = 1, C = 0111110000000110, Halt = 0
616 clk = 1, C = 1111110000000110, Halt = 0
620 clk = 0, C = 1111110000000110, Halt = 0
625 clk = 1, C = 0111111000000011, Halt = 0
630 clk = 0, C = 0111111000000011, Halt = 0
635 clk = 1, C = 0011111100000001, Halt = 0 => 30F1
                         ...
```

...But as the multiplier is supposed to disable its output when the result is not ready, I changed a little the test file to do it (by setting CEz at 1 when the device is busy, which place C in a high impedance state).



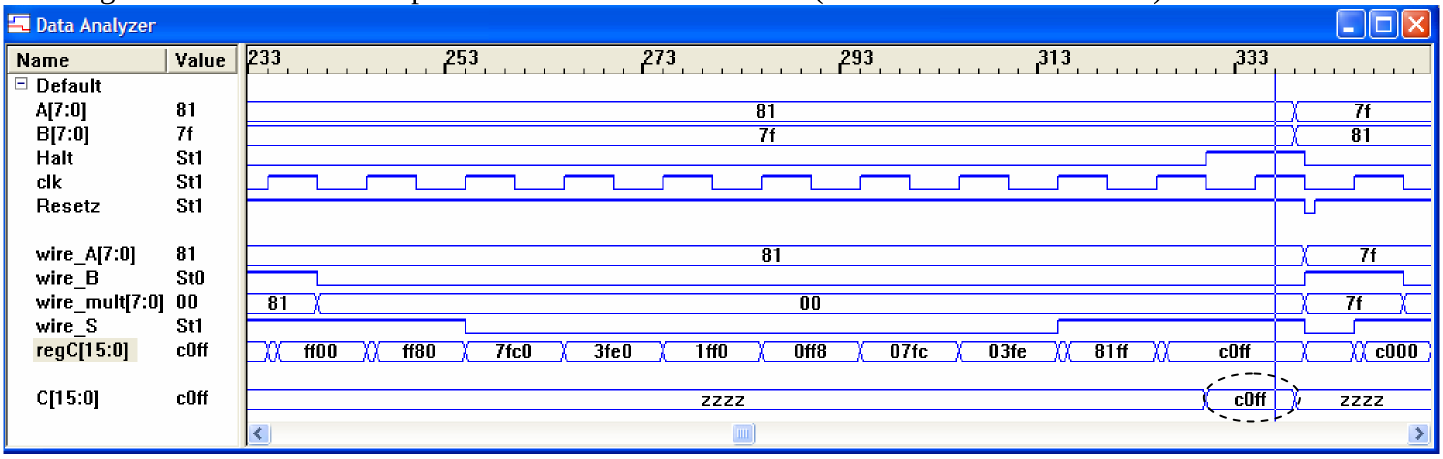
```
module Systolic_multiplier_test;
      reg [7:0] A, B;
      reg Im,Resetz,CEz,clk;

      wire [15:0] C;
      wire Halt;

Systolic_multiplier inst(C, Halt, A, B, Im, Resetz, CEz, clk);

initial begin
$monitor($time, " clk = %b, C = %b, Halt = %b", clk, C, Halt);

      clk = 0;
      Im = 1;
      CEz = 1;
      Resetz = 1;

      A = -127;
      B = -127;            // result expected : 3F01
      #1 Resetz = 0;
      #1 Resetz = 1;
      @(Halt) CEz = 0;
      #9 CEz = 1;

      Im = 0;
      B = 127;             // result expected : C0FF
      #1 Resetz = 0;
      #1 Resetz = 1;
      @(Halt) CEz = 0;
      #9 CEz = 1;

      Im = 1;
      A = 127;
      B = -127;            // result expected : C0FF
      #1 Resetz = 0;
      #1 Resetz = 1;
      @(Halt) CEz = 0;
      #9 CEz = 1;

      Im = 0;
      B = 127;             // result expected : 3F01
      #1 Resetz = 0;
      #1 Resetz = 1;
      //wait(Halt);
      @(Halt) CEz = 0;
      #9 $finish;
end

always #5 clk = ~clk;

endmodule
```

Chronogram result for 1st multiplication: -127 × -127 = 16129 (= 0x81 × 0x81 = 0x3F01)


Data Analyzer

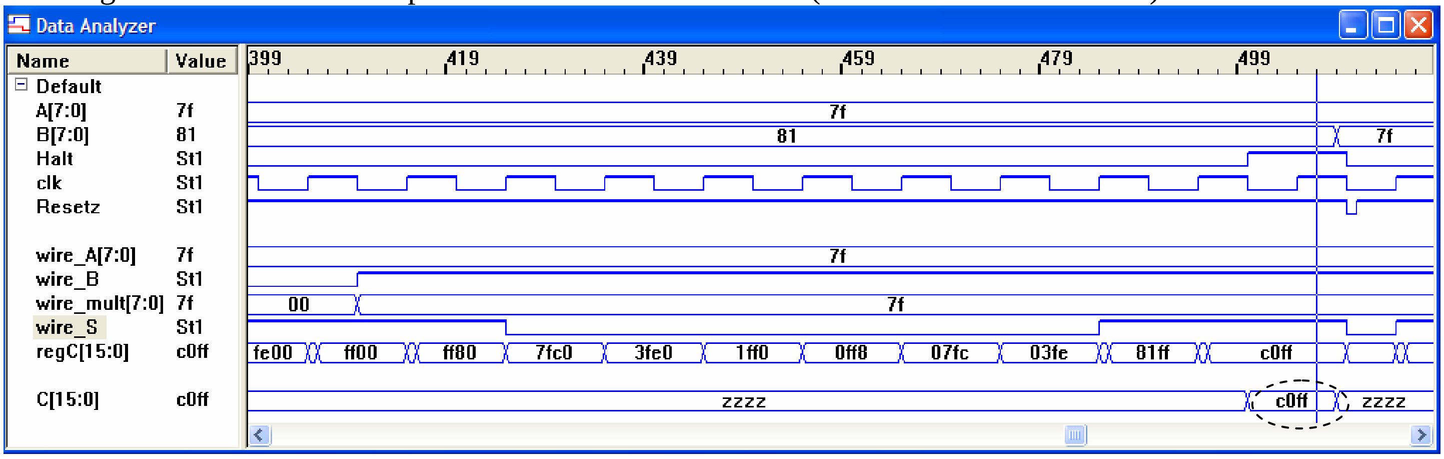
Name	Value
Default	
A[7:0]	81
B[7:0]	81
Halt	St1
clk	St1
Resetz	St1
wire_A[7:0]	81
wire_B	St1
wire_mult[7:0]	81
wire_S	St0
regC[15:0]	3f01
C[15:0]	3f01

Chronogram result for 2nd multiplication: -127 × 127 = -16129 (= 0x81 × 0x7F = 0xC0FF)


Data Analyzer

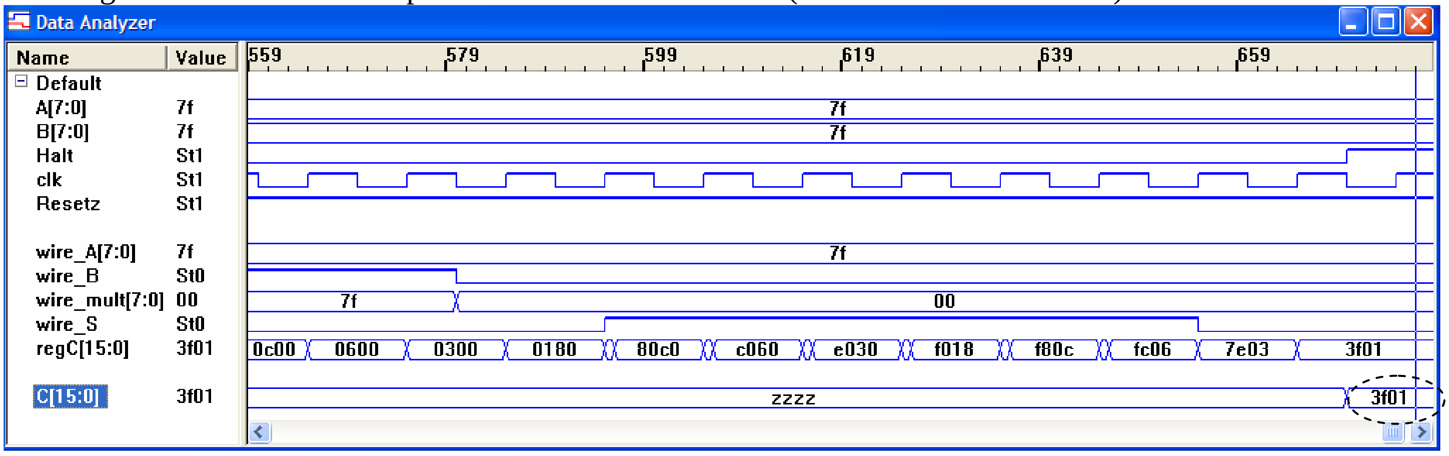
Name	Value
Default	
A[7:0]	81
B[7:0]	7f
Halt	St1
clk	St1
Resetz	St1
wire_A[7:0]	81
wire_B	St0
wire_mult[7:0]	00
wire_S	St1
regC[15:0]	c0ff
C[15:0]	c0ff

Chronogram result for 3rd multiplication: 127 × -127 = -16129 (= 0x7F × 0x81 = 0xC0FF)


Data Analyzer

Name	Value
Default	
A[7:0]	7f
B[7:0]	81
Halt	St1
clk	St1
Resetz	St1
wire_A[7:0]	7f
wire_B	St1
wire_mult[7:0]	7f
wire_S	St1
regC[15:0]	c0ff
C[15:0]	c0ff

Chronogram result for 4th multiplication: 127 × 127 = 16129 (=0x7F × 0x7F = 0x3F01)

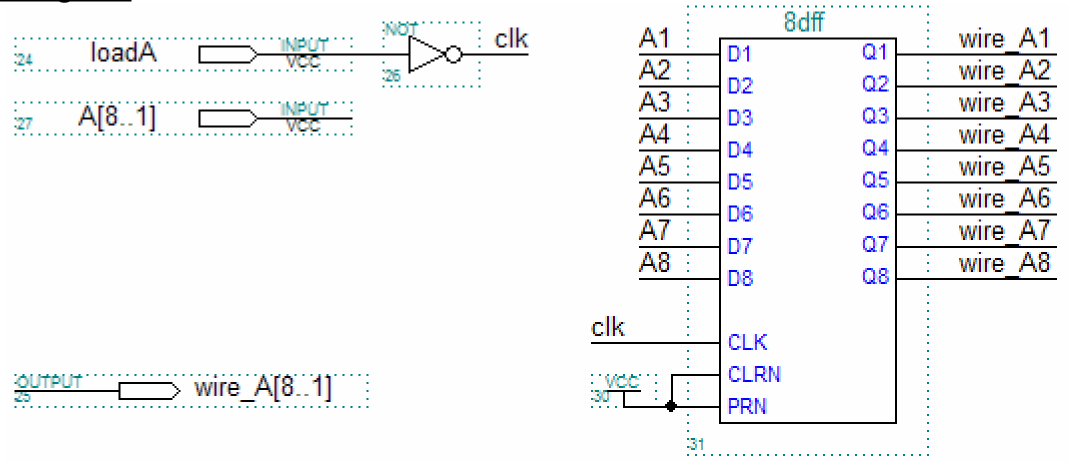

Data Analyzer

Name	Value
Default	
A[7:0]	7f
B[7:0]	7f
Halt	St1
clk	St1
Resetz	St1
wire_A[7:0]	7f
wire_B	St0
wire_mult[7:0]	00
wire_S	St0
regC[15:0]	3f01
C[15:0]	3f01

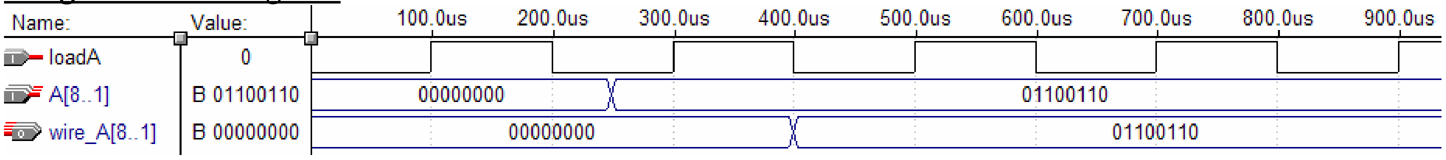
2 *Gate level multiplier*

Now we have seen that the Verilog design is efficient. We are thus going to stick to its principle but all the virtual time management of Silos becomes sometime a little more complex in gate level considering that all the gate delays are not zero and all the behavioural description are not always easy to translate (synthesize).

"RegA" block diagram:

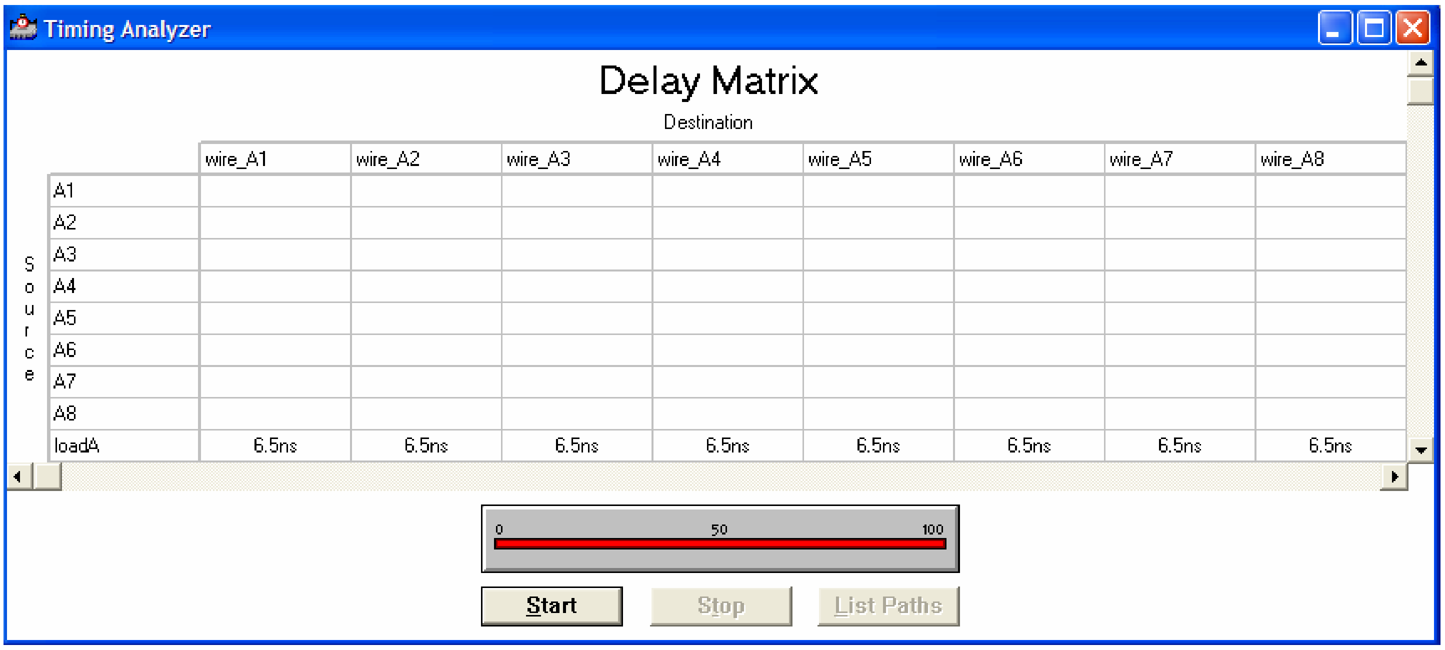


"RegA" test chronogram:



The result expected is obtained: at the clock edge we get the input in output.

"RegA" time analysis:



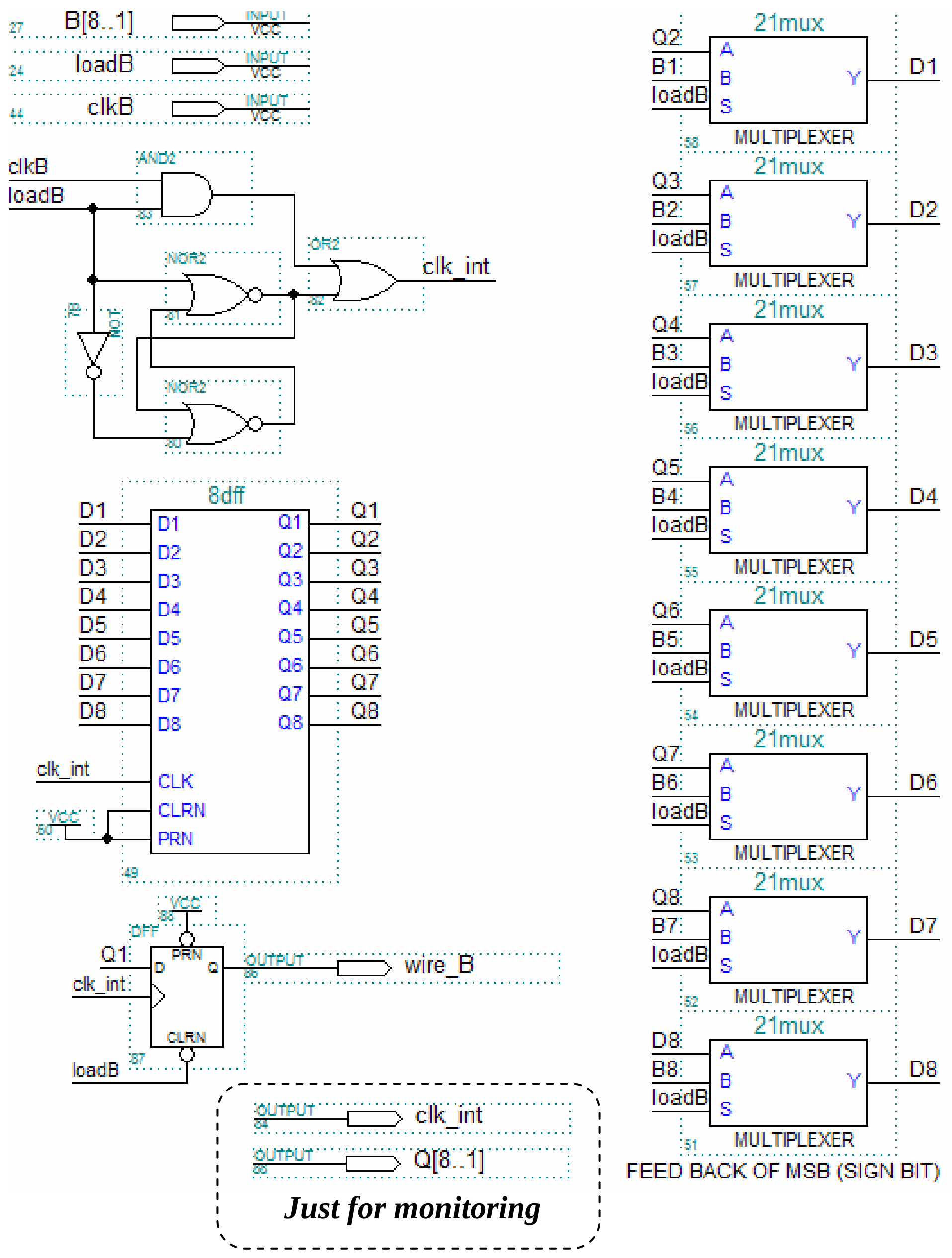
Delay Matrix

Destination

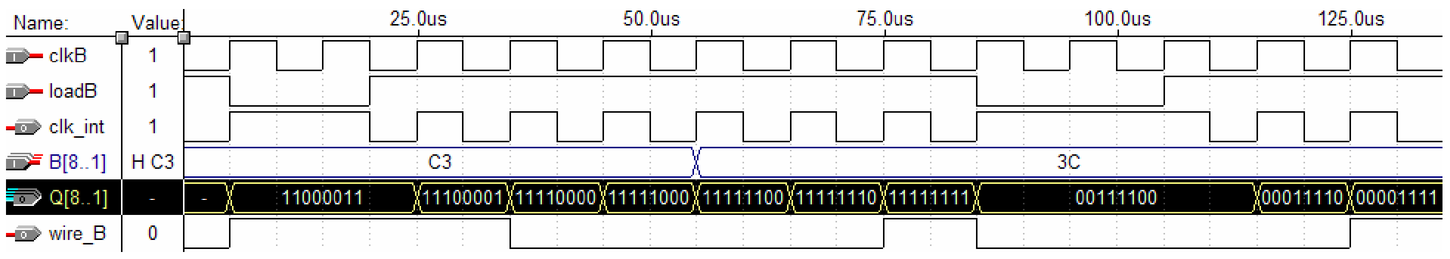
Source	wire_A1	wire_A2	wire_A3	wire_A4	wire_A5	wire_A6	wire_A7	wire_A8
A1								
A2								
A3								
A4								
A5								
A6								
A7								
A8								
loadA	6.5ns	6.5ns	6.5ns	6.5ns	6.5ns	6.5ns	6.5ns	6.5ns

We keep these results for later, to see the speed limit of our final component.

<u>"b_piso"</u> <u>block diagram:</u> (parallel in serial out)



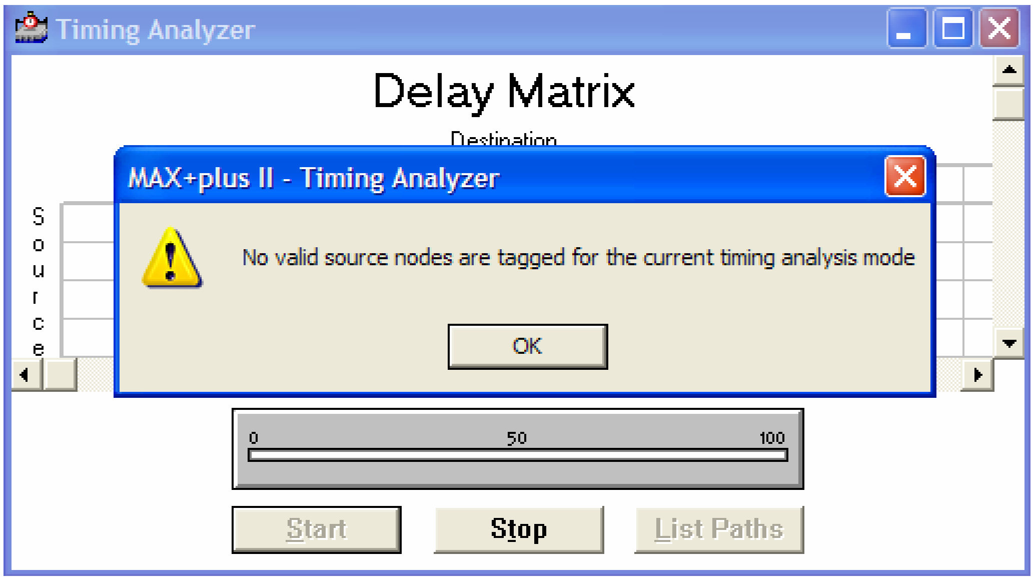
<u>"b_piso"</u> <u>test chronogram:</u>



Note: as we can see, I've added 2 extra outputs to be able to monitoring the DFF values and the internal clock (clk_int). The sign Bit is correctly propagated and the output of the module is the LSB as wanted.

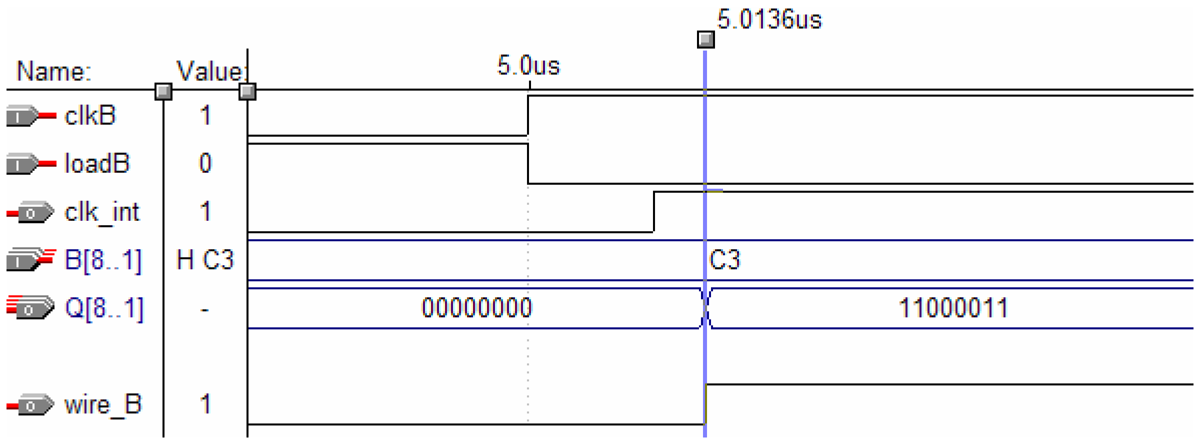
The internal clock was not a piece of cake to create, it seems to be simple but it's a RS flip flop connected with another logical bloc that allows to disable the clock when it's loading.

"b_piso" time analysis: for some reason the analyser doesn't want to simulate this component:



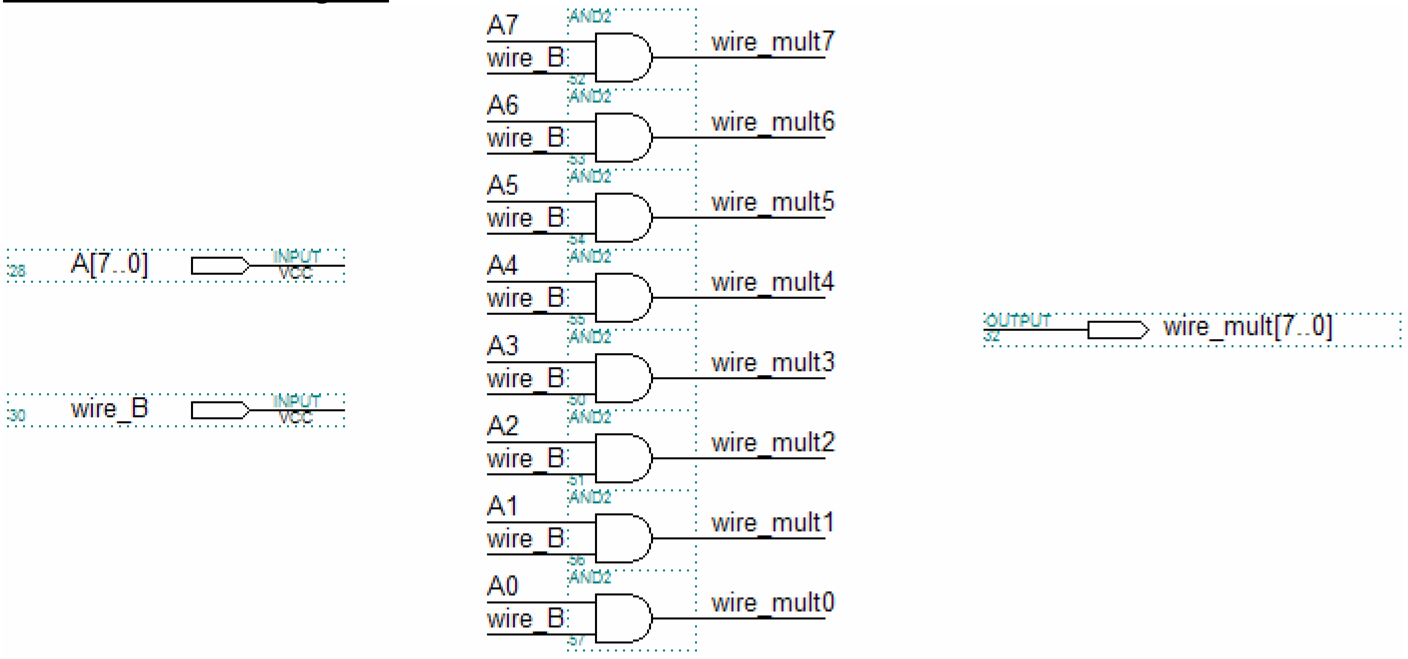
No valid source nodes are tagged for the current timing analysis mode

…but nothing can stop me!
=> I zoomed (a lot) on all transitions of the "b_piso" test chronogram and I found the longest time delay:

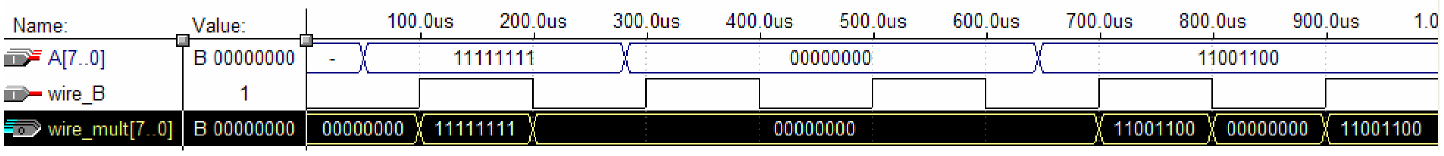


It thus seems that the longest time delay is 13.6ns

"REG_mult" block diagram:

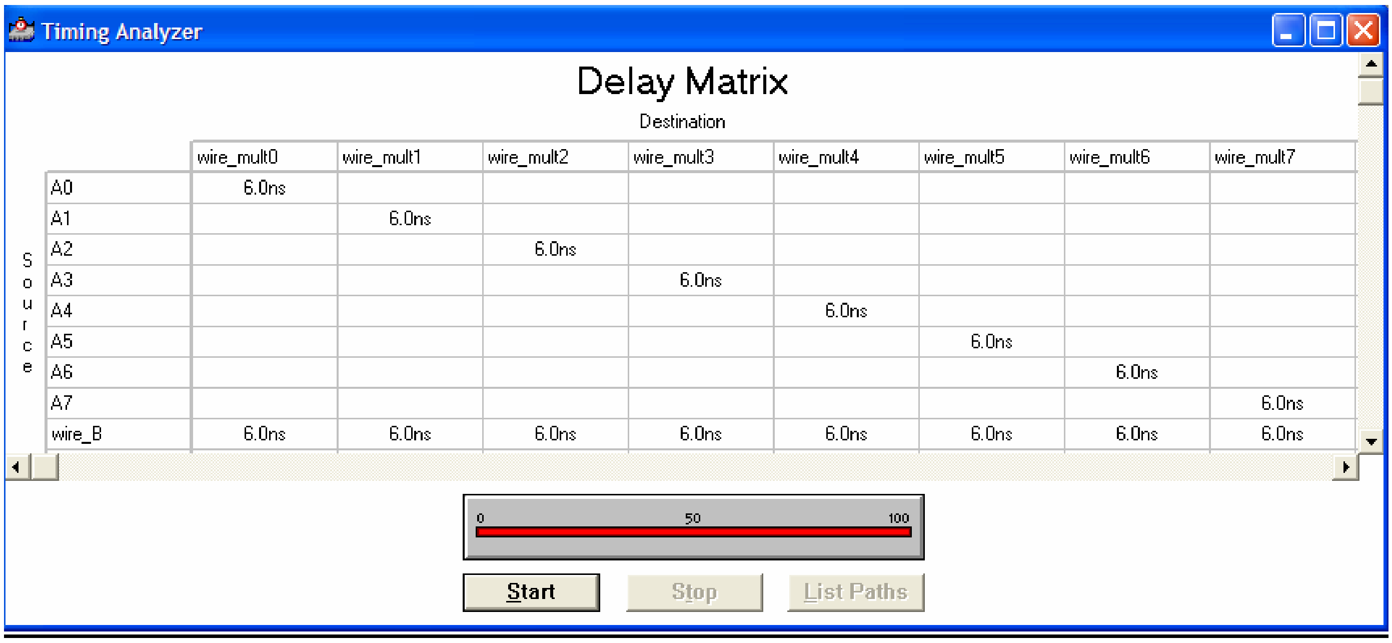


"REG_mult" test chronogram:



Note: This component, completely combinatorial, is definitely the simplest of the multiplier, but the paradox is that it's the only one to perform a real multiplication!

"REG_mult" time analysis



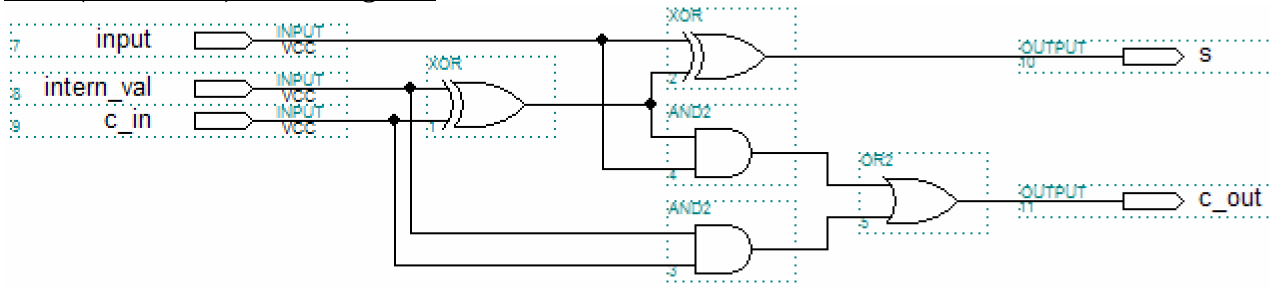
Delay Matrix

Destination

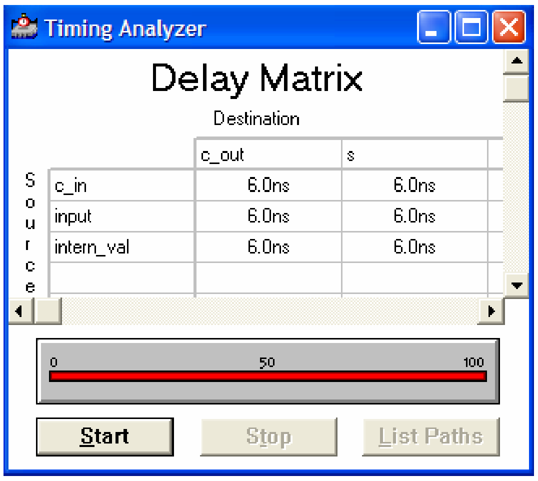
Source	wire_mult0	wire_mult1	wire_mult2	wire_mult3	wire_mult4	wire_mult5	wire_mult6	wire_mult7
A0	6.0ns							
A1		6.0ns						
A2			6.0ns					
A3				6.0ns				
A4					6.0ns			
A5						6.0ns		
A6							6.0ns	
A7								6.0ns
wire_B	6.0ns	6.0ns	6.0ns	6.0ns	6.0ns	6.0ns	6.0ns	6.0ns

As explained in the Verilog design, to make the sum, I used a full adder and I've duplicated it with synchronised feed back of the carry (by a DFF).

"fa" (Full Adder) block diagram:



"fa" (Full Adder) time analysis:

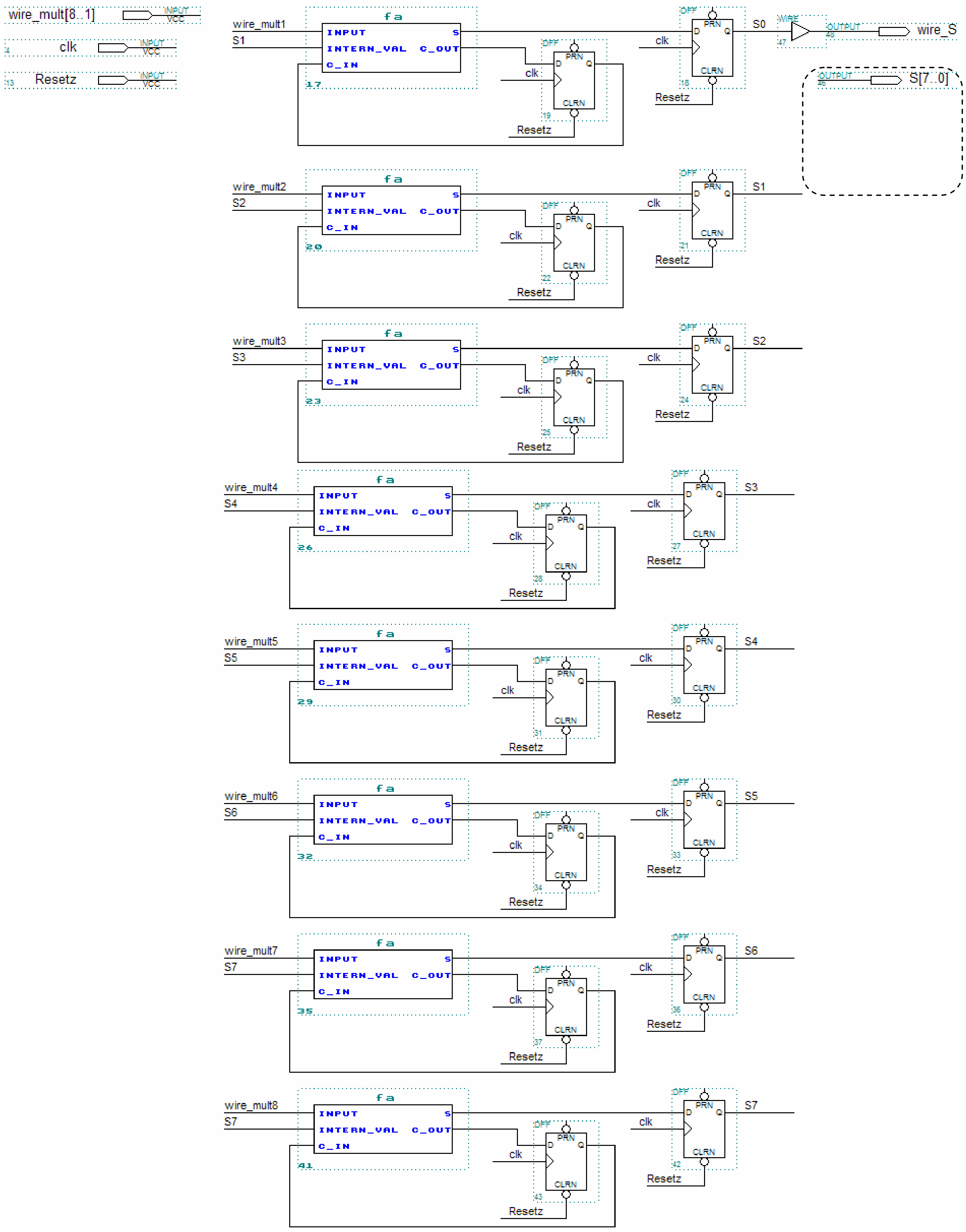


Delay Matrix

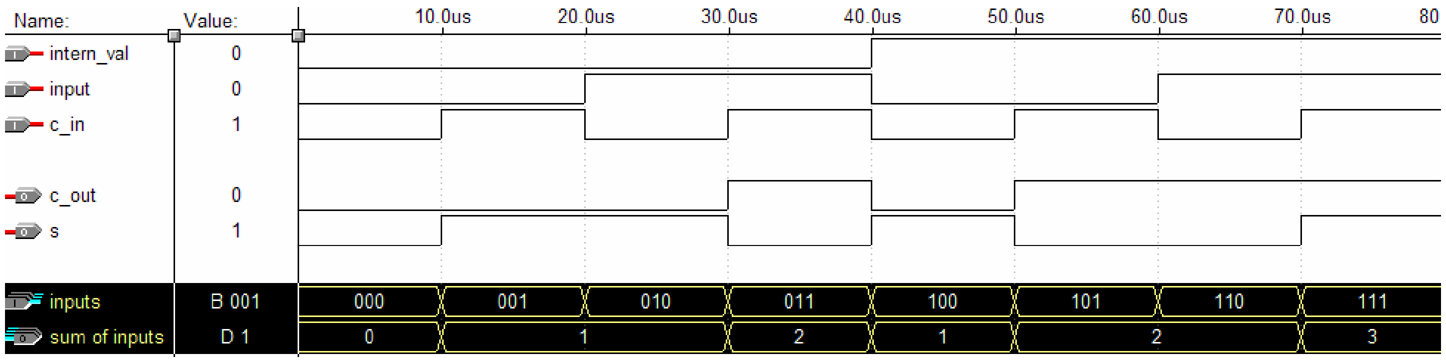
Destination

Source	c_out	s
c_in	6.0ns	6.0ns
input	6.0ns	6.0ns
intern_val	6.0ns	6.0ns

"summm" block diagram: (here is the component that effectuate the instantiation of the full adder)

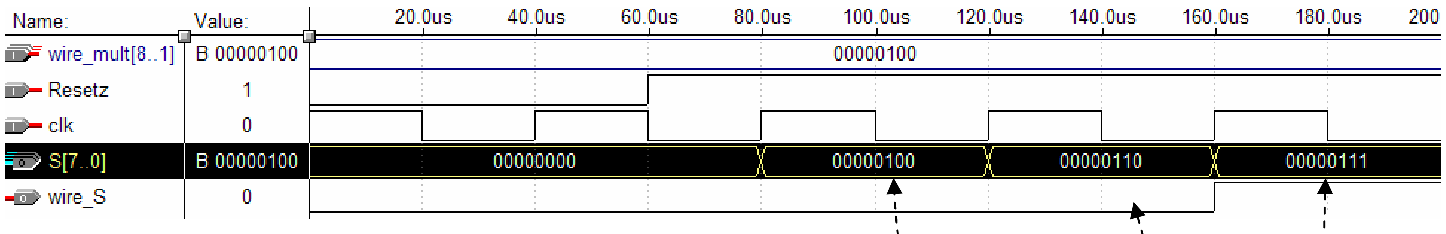


"**fa**" (Full Adder) test chronogram:



To try making the test more readable I used the "group" function that allows taking several pins to make a bus. I displayed the value of the inputs in binary to see the number of "1" in the bus created and in output, the display is in decimal to se the result directly.

"**summ**" test chronogram:



In the extra output called "S" we obtain *half* of the sum of the input "wire_mult" and the previous "S" state.
=> *half* because of the **shift** action (which gives the entire part of the half to be more accurate).
We thus obtain as expected:

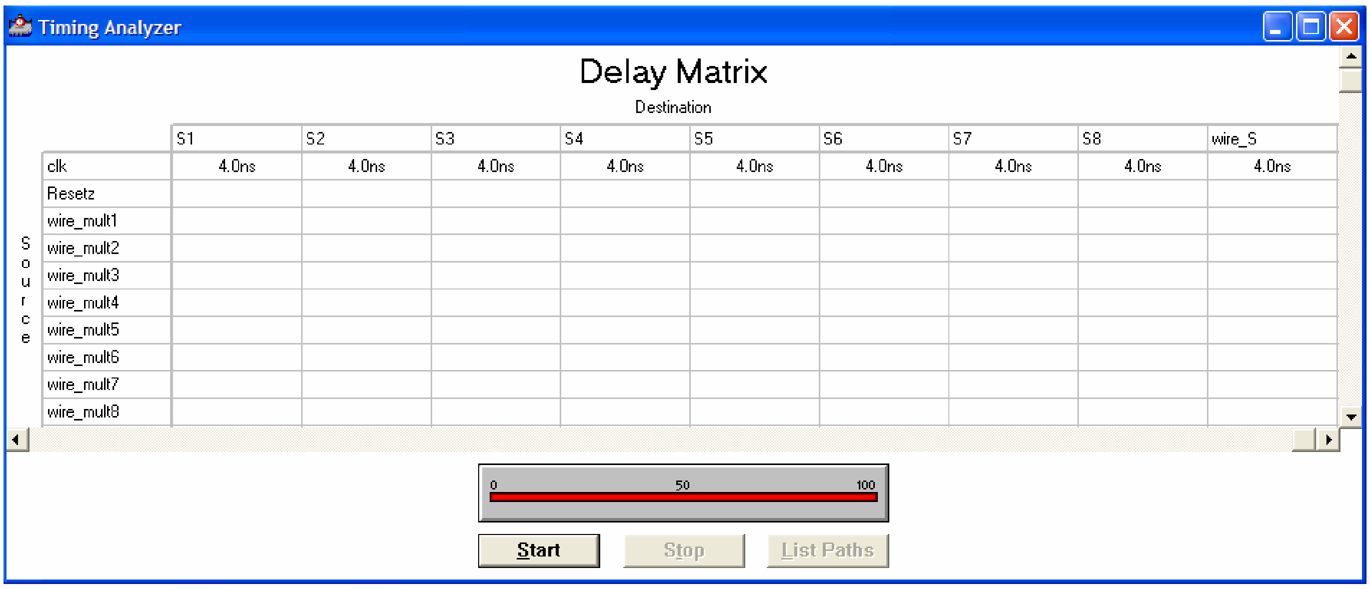
$$\begin{array}{r} 0 \\ \underline{+\ 4} \\ 4 \end{array}$$

SHIFT => 4/2 = 2
$$\begin{array}{r} \underline{+\ 4} \\ 6 \end{array}$$

SHIFT => 6/2 = 3
$$\begin{array}{r} \underline{+\ 4} \\ 7 \end{array}$$

…

" **summ** " time analysis:

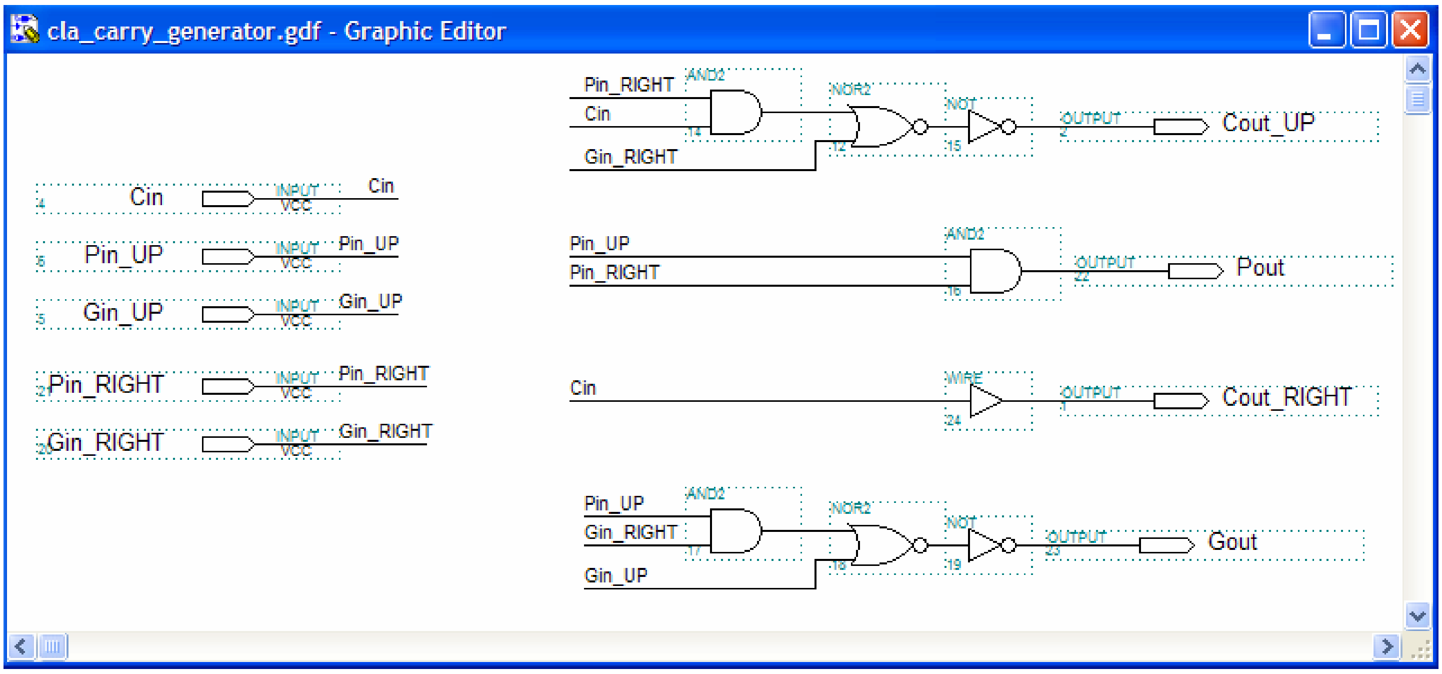

Timing Analyzer

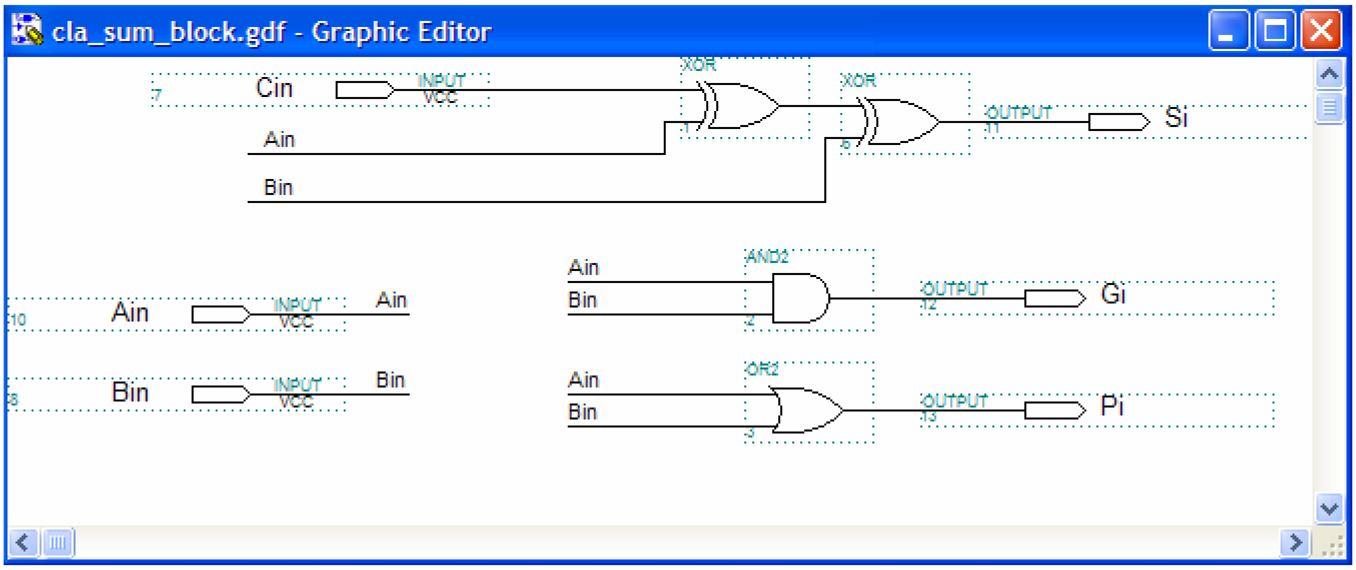
Delay Matrix

Destination

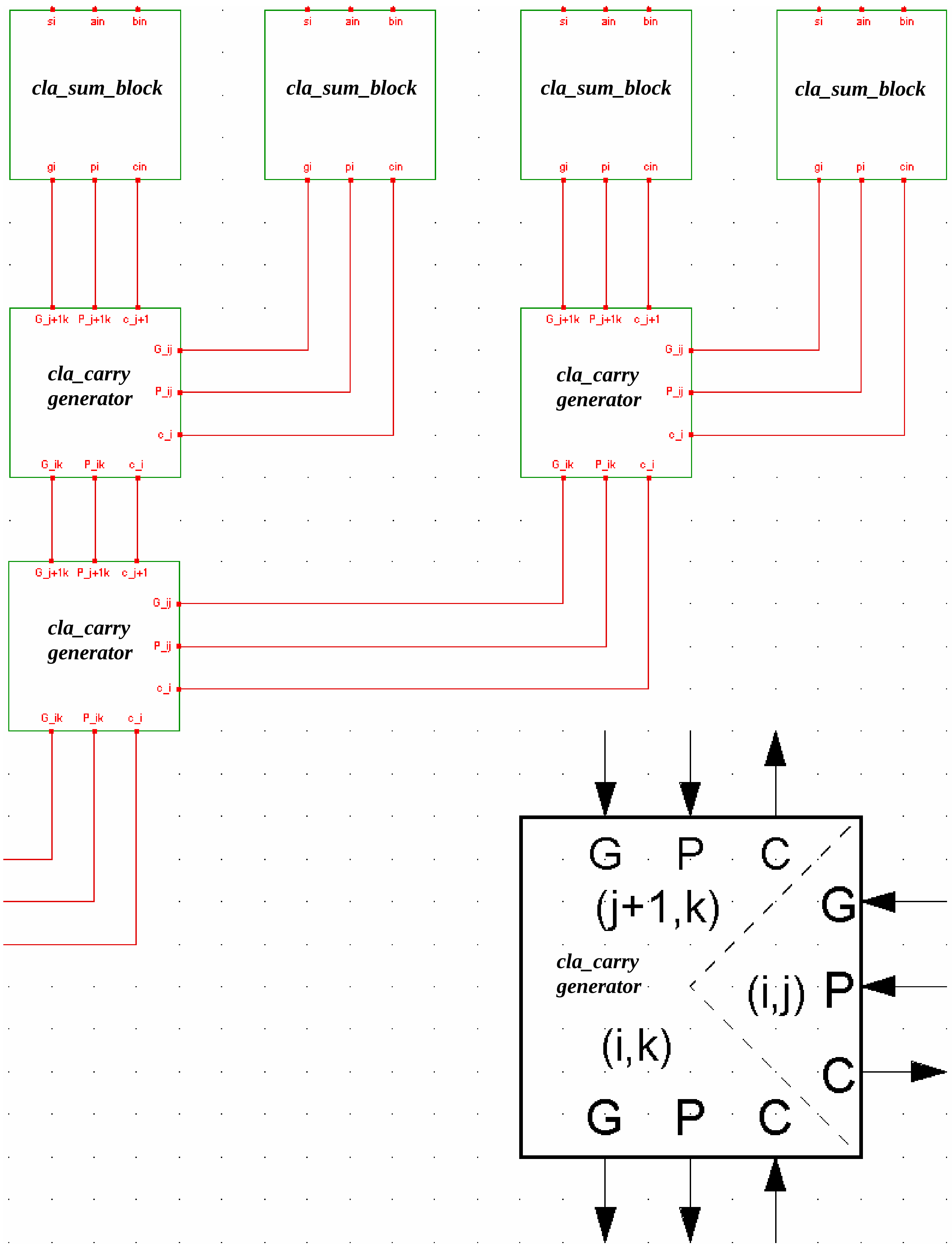
Source	S1	S2	S3	S4	S5	S6	S7	S8	wire_S
clk	4.0ns	4.0ns	4.0ns	4.0ns	4.0ns	4.0ns	4.0ns	4.0ns	4.0ns
Resetz									
wire_mult1									
wire_mult2									
wire_mult3									
wire_mult4									
wire_mult5									
wire_mult6									
wire_mult7									
wire_mult8									

I've had the idea to use a Carry-Look-Ahead adder before I've chosen this design (to go quicker).
In my 1st shot have not thought that the carry can be "sequentially rippled" then doesn't take that much time! …however, the CLA adder was really complex and didn't allow saving a lot of time, it's just interesting from more than 16bits additions. But I've implemented (in a long night) then I show it, for the souvenir:



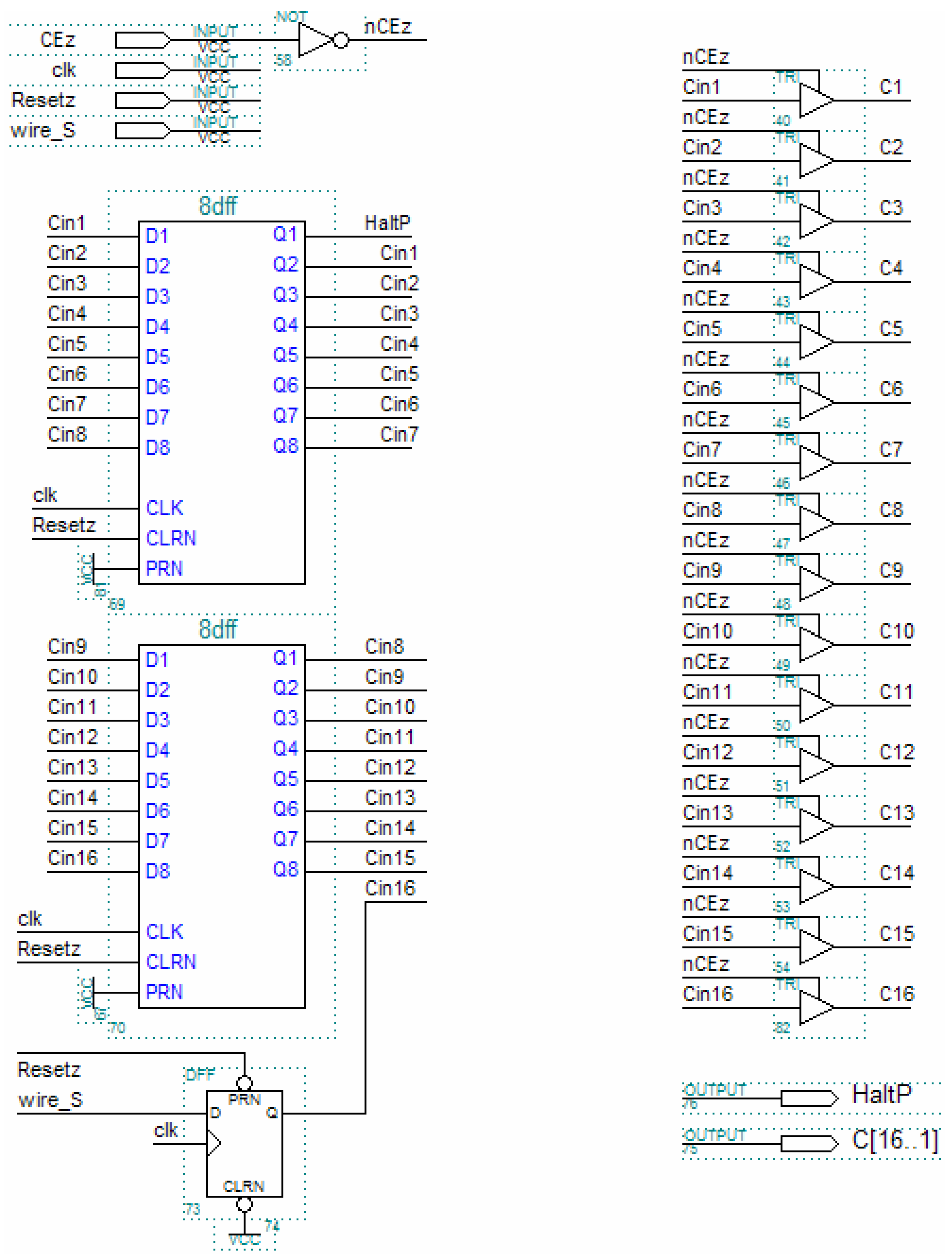


Here is a part of the CLA adder design (just for 4 bits) but the tree is just doubled.



source: http://tams-www.informatik.uni-hamburg.de

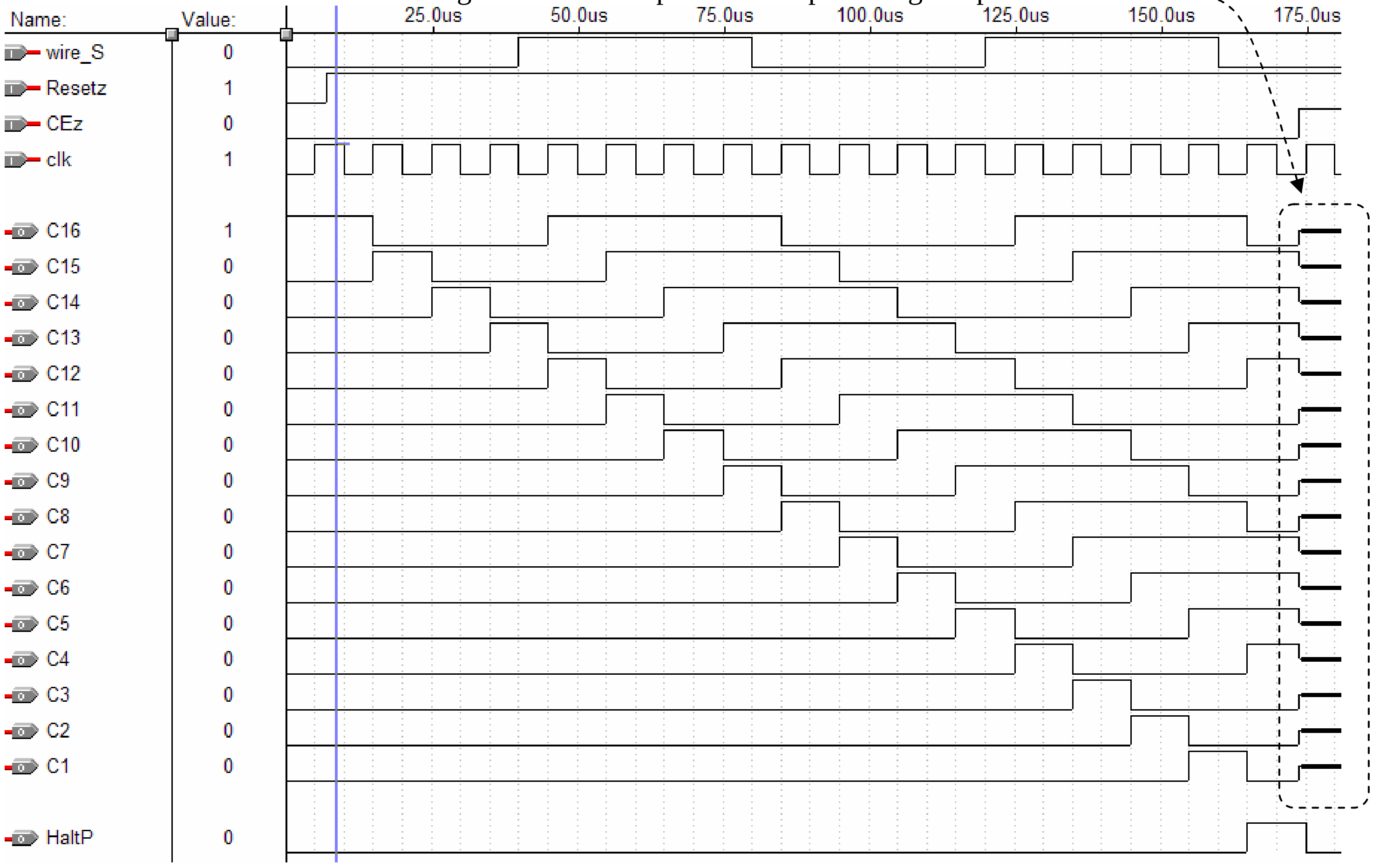
"C_SIPO" block diagram: (serial in parallel out)



This component allows implementing the high impedance state by using the tristate gate, it also allows resetting with the MSB at 1 (it's the marker that will count the 16 clock edges to raise the halt signal when the multiplication is finished) and finally it contains the halt memory cell.

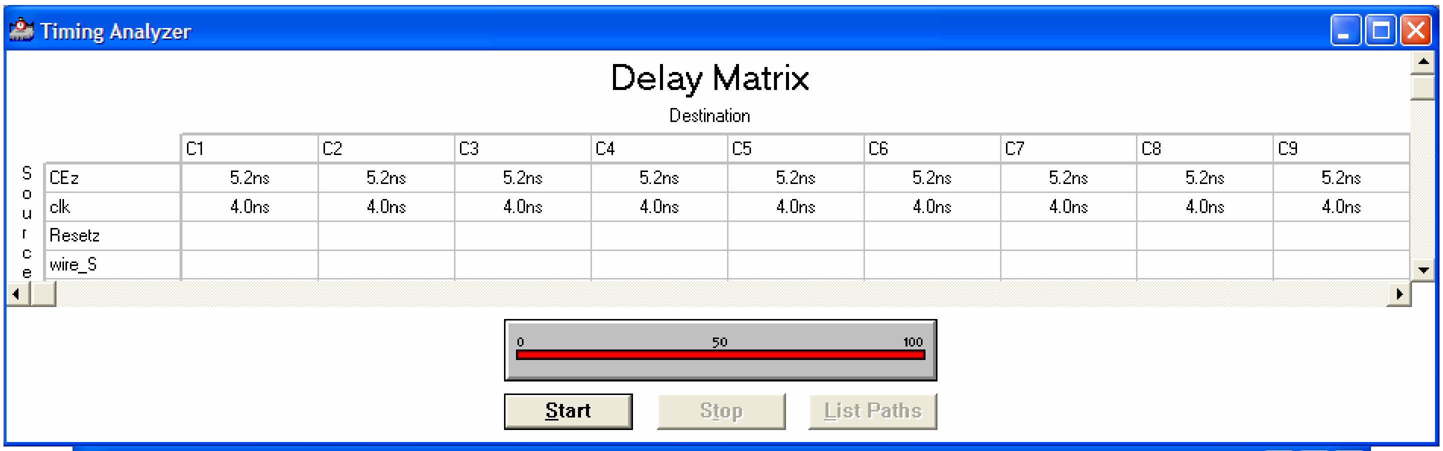
"C_SIPO" test chronogram:

Just to show ho it works I've set the signal CEz at 1 to place the output in high impedance state…

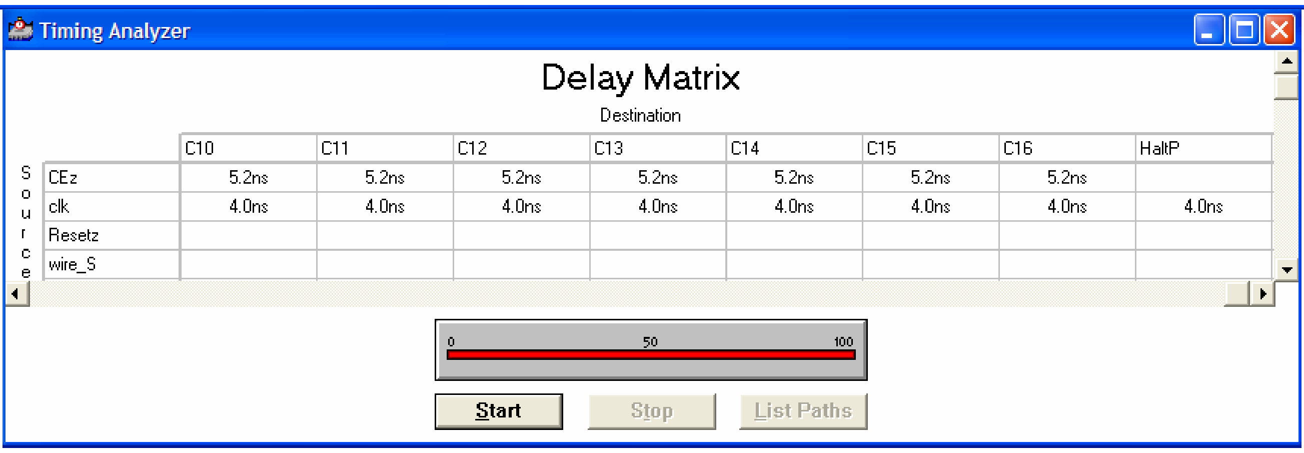


…and we can see the halt signal raised at the 16th clock edges.

" C_SIPO " time analysis:

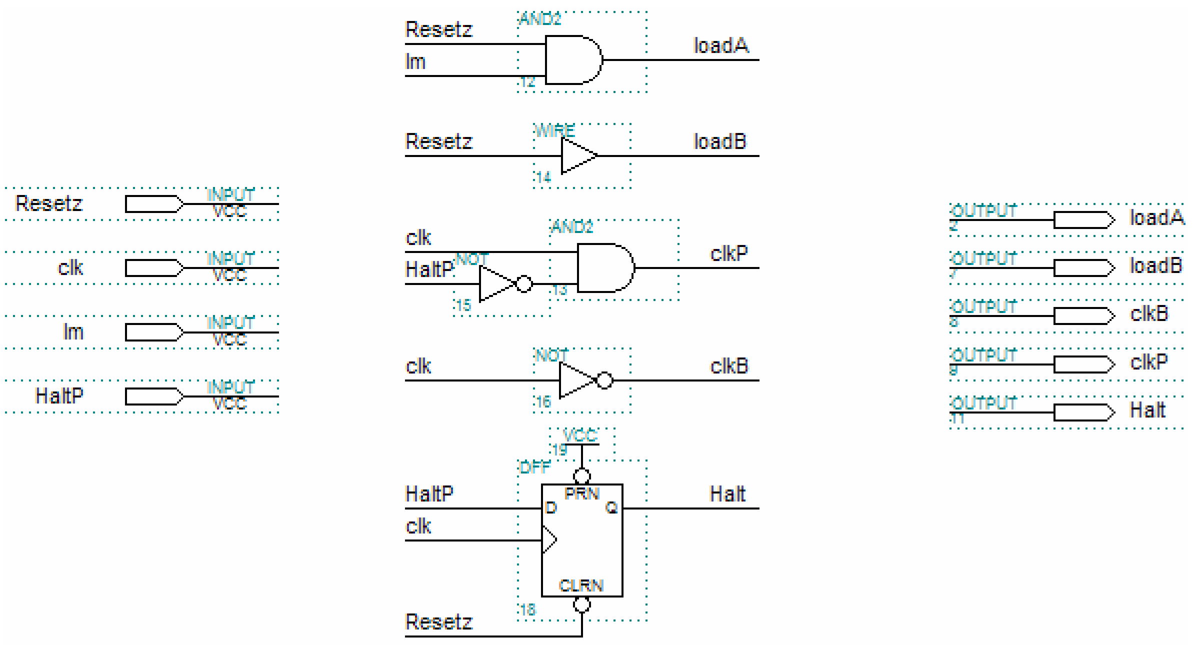


	C1	C2	C3	C4	C5	C6	C7	C8	C9
CEz	5.2ns	5.2ns	5.2ns	5.2ns	5.2ns	5.2ns	5.2ns	5.2ns	5.2ns
clk	4.0ns	4.0ns	4.0ns	4.0ns	4.0ns	4.0ns	4.0ns	4.0ns	4.0ns
Resetz									
wire_S									

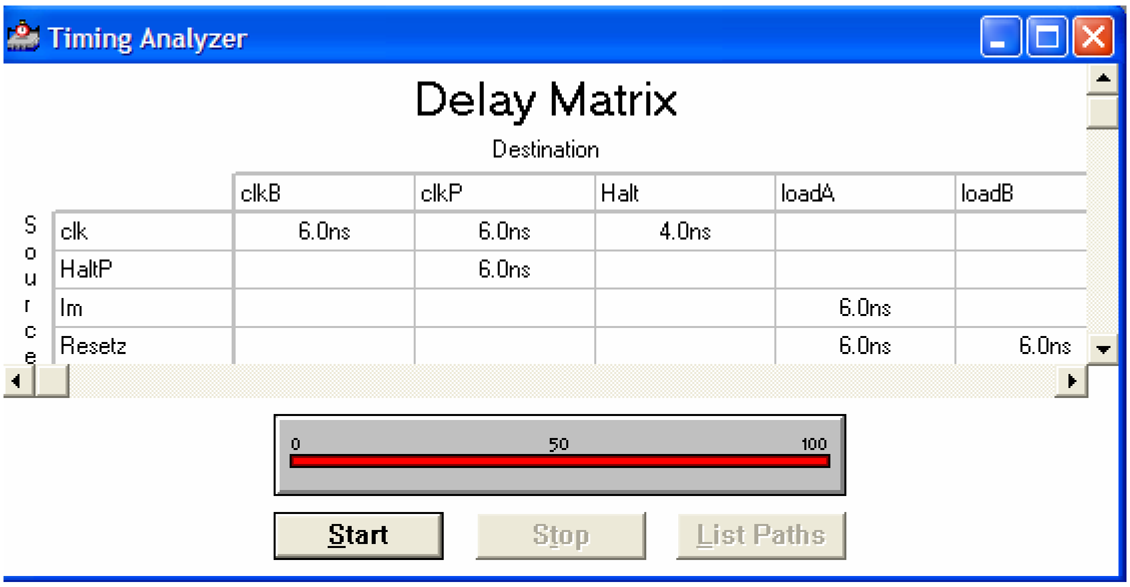


	C10	C11	C12	C13	C14	C15	C16	HaltP
CEz	5.2ns	5.2ns	5.2ns	5.2ns	5.2ns	5.2ns	5.2ns	
clk	4.0ns	4.0ns	4.0ns	4.0ns	4.0ns	4.0ns	4.0ns	4.0ns
Resetz								
wire_S								

"CNTR" block diagram: (control unit)



"CNTR" time analysis:

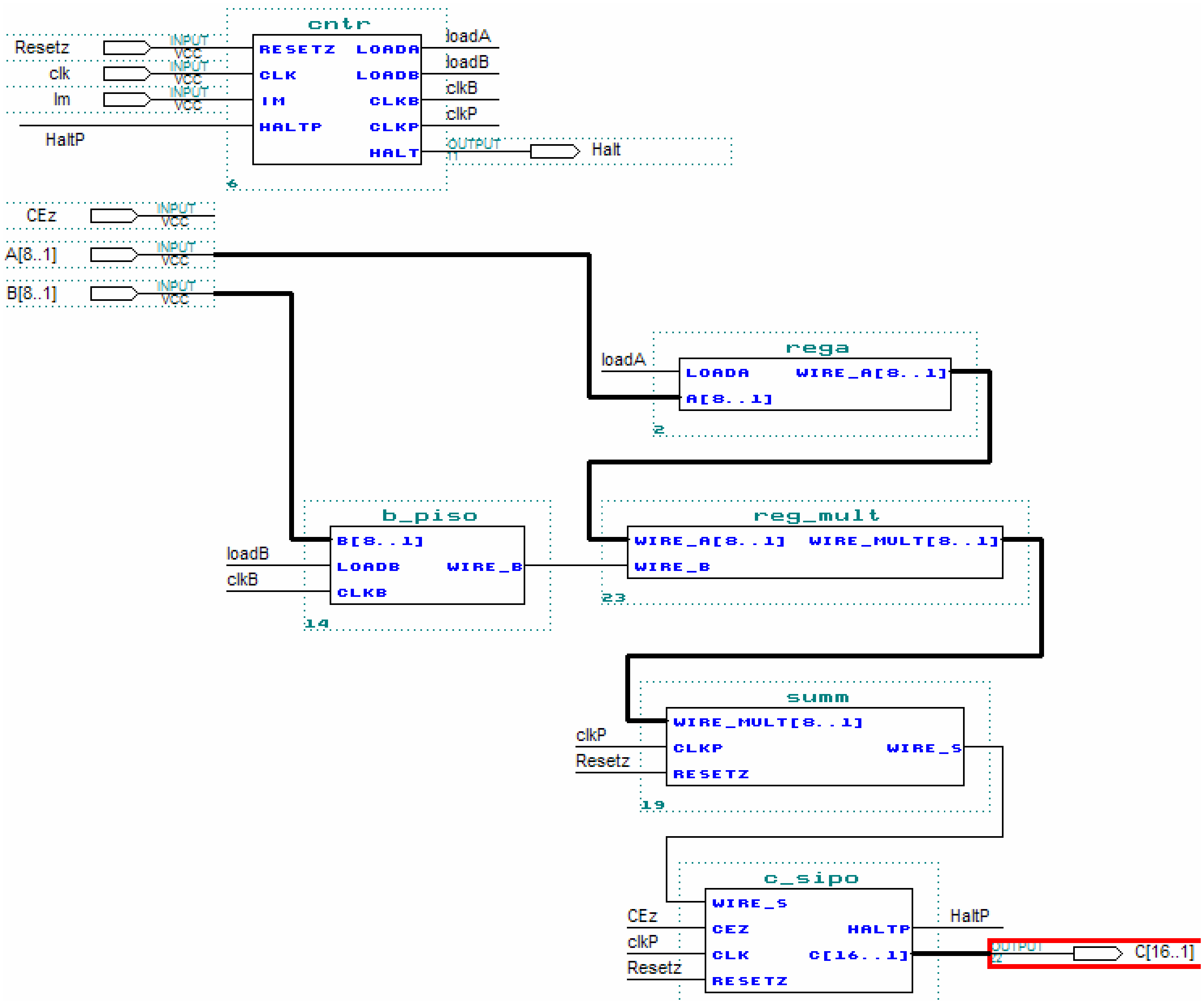


Delay Matrix

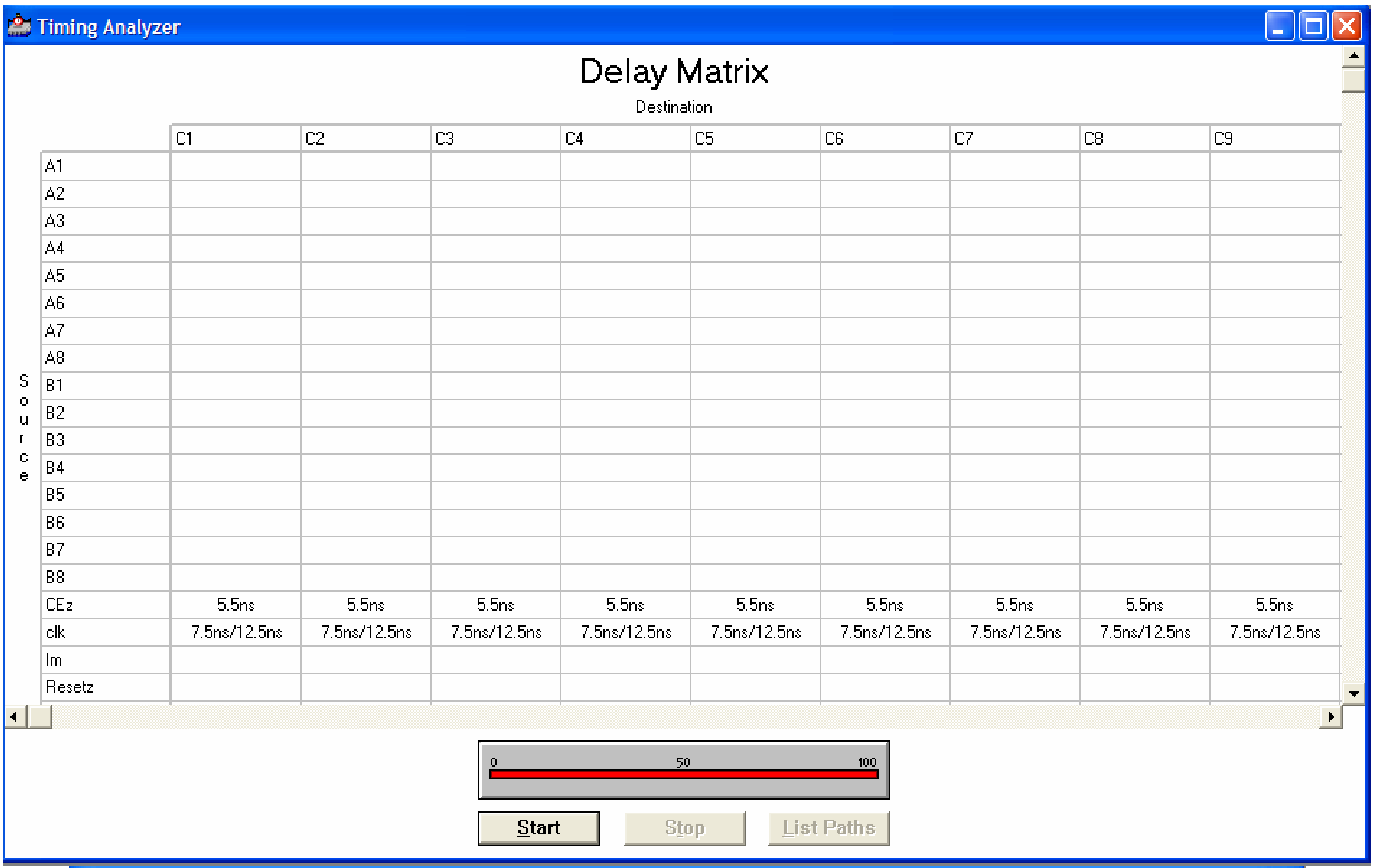
Destination

Source	clkB	clkP	Halt	loadA	loadB
clk	6.0ns	6.0ns	4.0ns		
HaltP		6.0ns			
Im				6.0ns	
Resetz				6.0ns	6.0ns

"Systolic multiplier" block diagram: (instantiation of all the components)



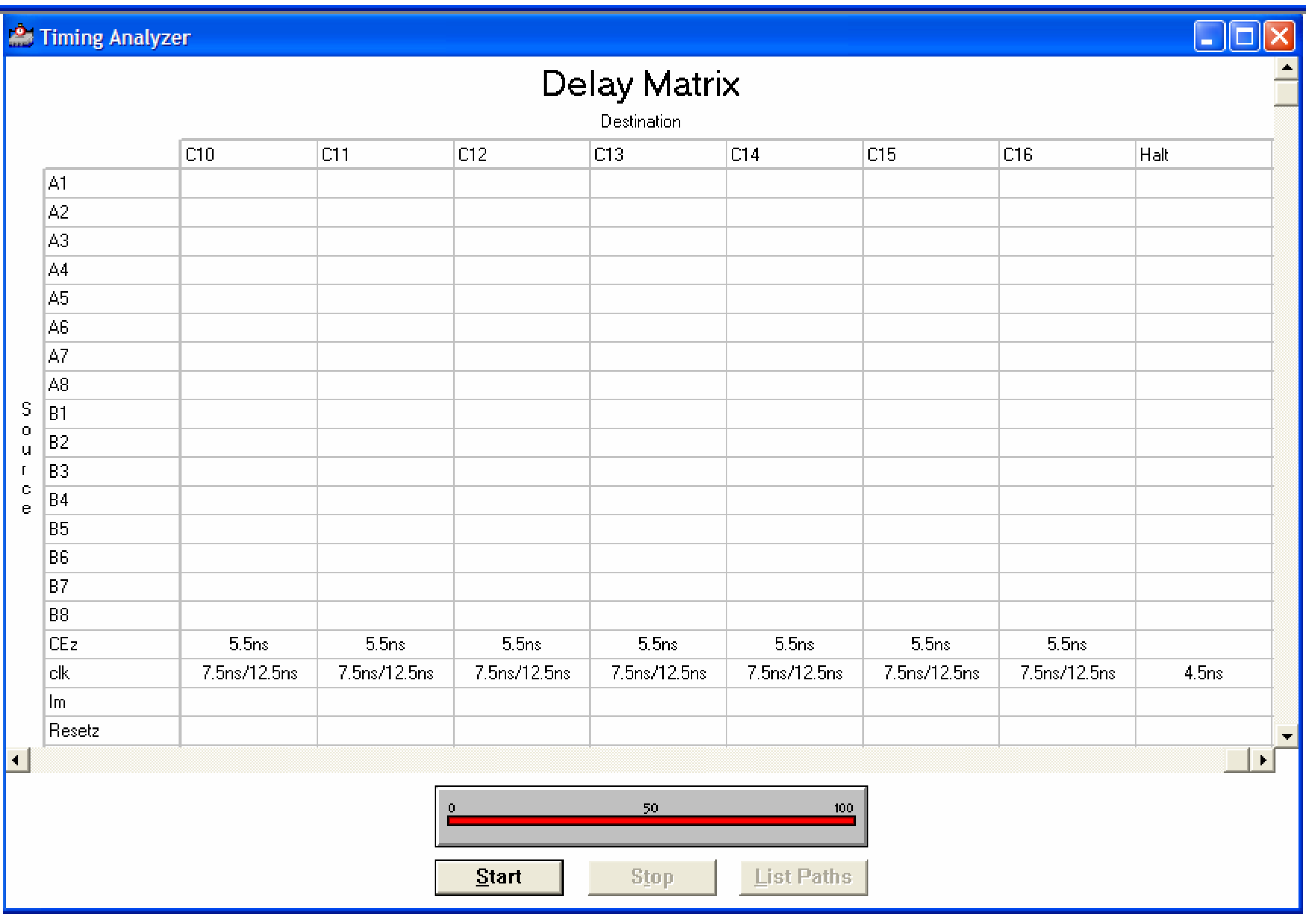
"Systolic multiplier" time analysis:



Delay Matrix

Destination

Source	C1	C2	C3	C4	C5	C6	C7	C8	C9
A1									
A2									
A3									
A4									
A5									
A6									
A7									
A8									
B1									
B2									
B3									
B4									
B5									
B6									
B7									
B8									
CEz	5.5ns	5.5ns	5.5ns	5.5ns	5.5ns	5.5ns	5.5ns	5.5ns	5.5ns
clk	7.5ns/12.5ns	7.5ns/12.5ns	7.5ns/12.5ns	7.5ns/12.5ns	7.5ns/12.5ns	7.5ns/12.5ns	7.5ns/12.5ns	7.5ns/12.5ns	7.5ns/12.5ns
Im									
Resetz									



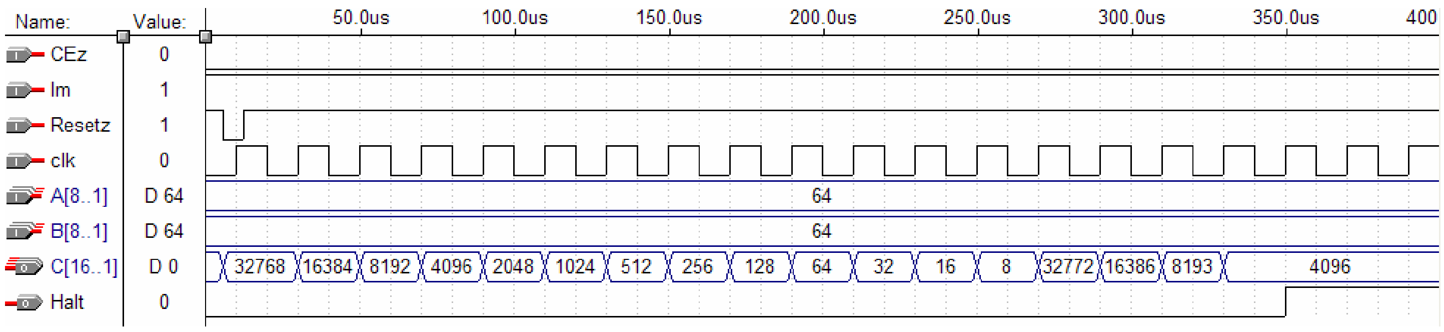
Delay Matrix

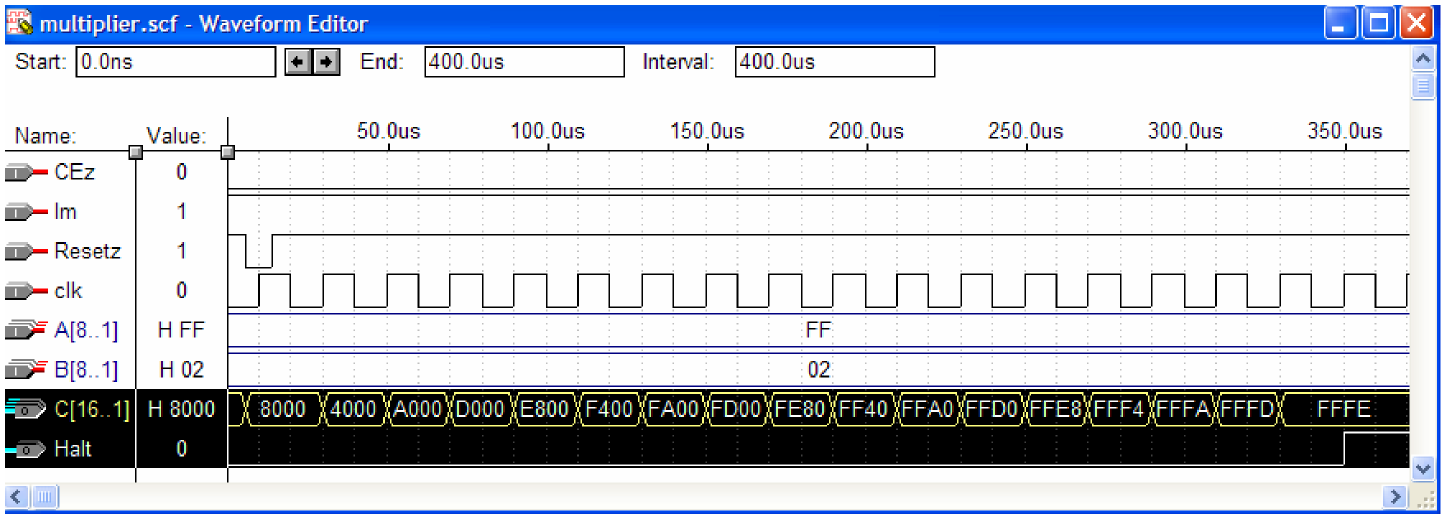
Destination

Source	C10	C11	C12	C13	C14	C15	C16	Halt
A1								
A2								
A3								
A4								
A5								
A6								
A7								
A8								
B1								
B2								
B3								
B4								
B5								
B6								
B7								
B8								
CEz	5.5ns	5.5ns	5.5ns	5.5ns	5.5ns	5.5ns	5.5ns	
clk	7.5ns/12.5ns	7.5ns/12.5ns	7.5ns/12.5ns	7.5ns/12.5ns	7.5ns/12.5ns	7.5ns/12.5ns	7.5ns/12.5ns	4.5ns
Im								
Resetz								

This analysis seems to give a maximum time of 12.5ns in hot conditions but I found 13.6ns in the component b_piso. The maximum speed is thus around 1/13.6ns ≈ 73MHz (but this value is just an estimation)

"Systolic multiplier" chronogram: I used simple values to show the result, for the positive multiplication I display in decimal and for the negative one, I used the hexadecimal display.





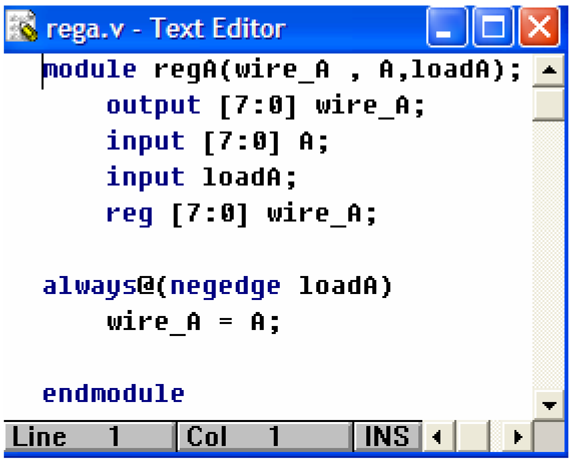
Note: in this report file (*.rpt) we can see, among other things, the element used in the FPGA chip

```
                                                     Shareable     External
Logic Array Block     Logic Cells   I/O Pins         Expanders   Interconnect

A:     LC1 - LC16      4/16( 25%)  12/12(100%)   1/16(  6%)   6/36( 16%)
B:    LC17 - LC32     16/16(100%)   8/12( 66%)   3/16( 18%)  28/36( 77%)
C:    LC33 - LC48     16/16(100%)   4/12( 33%)  12/16( 75%)  26/36( 72%)
D:    LC49 - LC64     16/16(100%)  11/12( 91%)   4/16( 25%)  20/36( 55%)


Total dedicated input pins used:                 2/4      ( 50%)
Total I/O pins used:                            35/48     ( 72%)
Total logic cells used:                         52/64     ( 81%)
Total shareable expanders used:                  4/64     (  6%)
Total Turbo logic cells used:                   52/64     ( 81%)
Total shareable expanders not available (n/a):  16/64     ( 25%)
Average fan-in:                                  5.11
Total fan-in:                                   266

Total input pins required:                      20
Total output pins required:                     17
Total bidirectional pins required:               0
Total logic cells required:                     52
Total flipflops required:                       50
Total product terms required:                  180
Total logic cells lending parallel expanders:    0
Total shareable expanders in database:           3
```

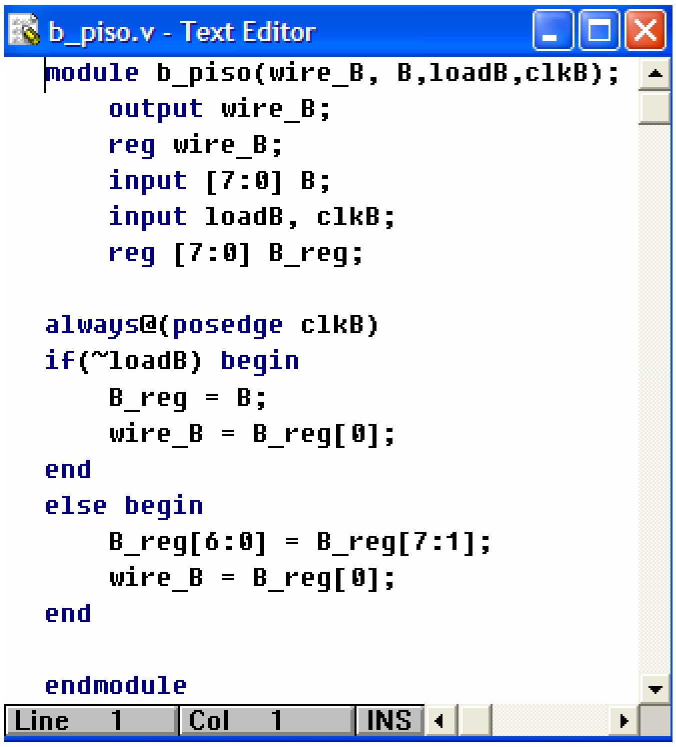
…and we can see the chip selected by Maxplus: (the speed limit depends also on the FPGA selected):

```
** DEVICE SUMMARY **

Chip/                          Input    Output    Bidir          Shareable
POF         Device             Pins     Pins      Pins     LCs   Expanders   % Utilized

multiplier
      EPM7064LC68-7             20       17        0       52      4           81 %

User Pins:                      20       17        0
```

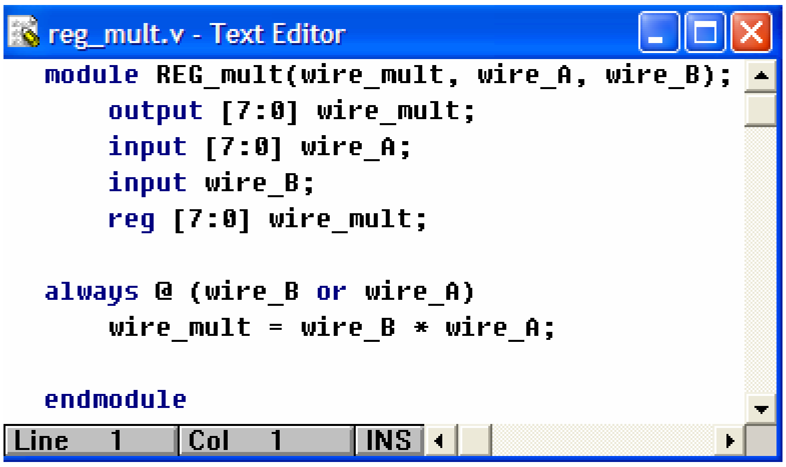
<u>BONUS :</u> As I still have a few "seconds" before I return this assignment, I've done a simulation of the Verilog code (sometimes modified) in Maxplus.



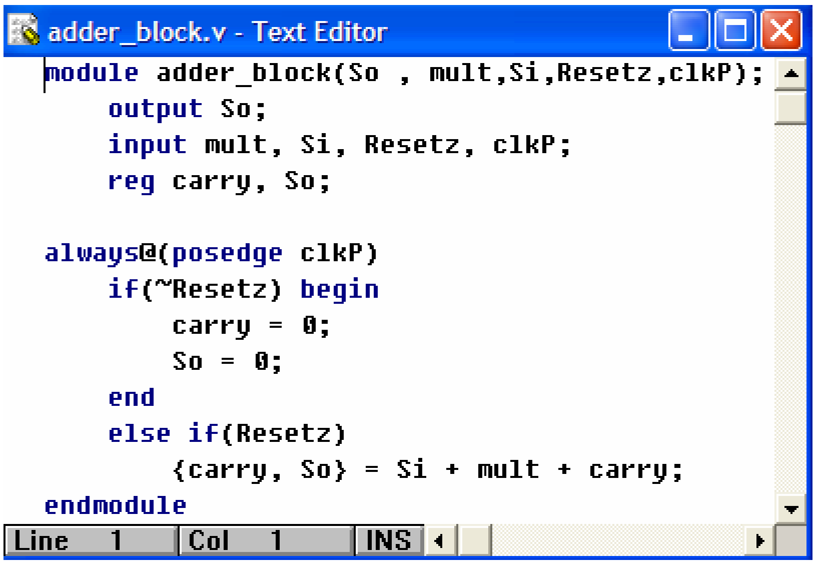
```
module regA(wire_A , A,loadA);
    output [7:0] wire_A;
    input [7:0] A;
    input loadA;
    reg [7:0] wire_A;

always@(negedge loadA)
    wire_A = A;

endmodule
```



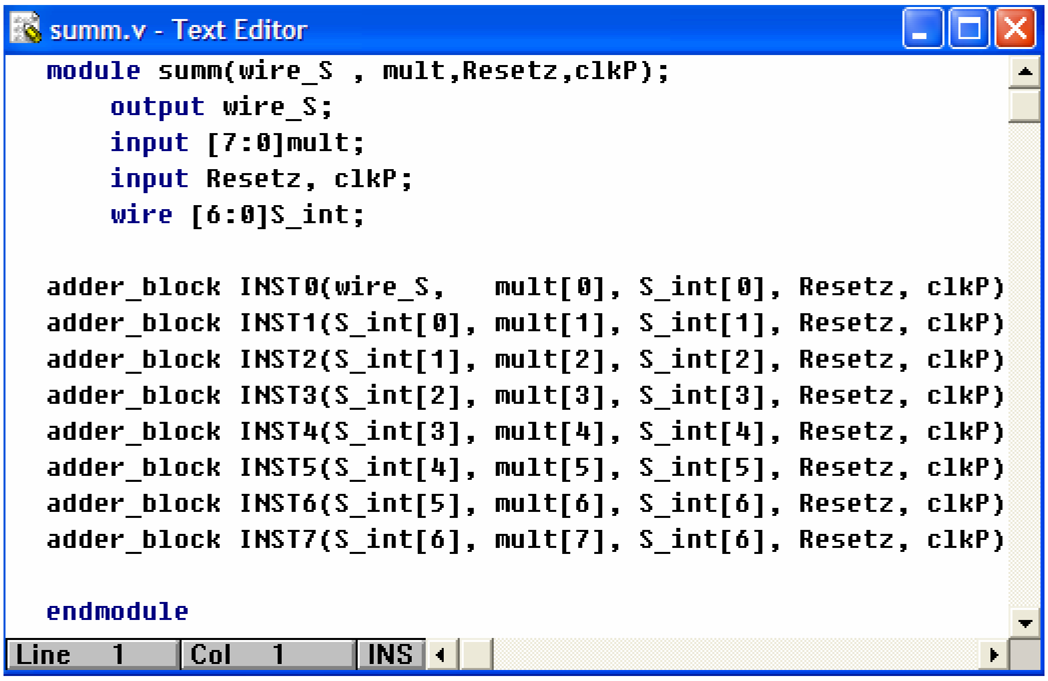
```
module b_piso(wire_B, B,loadB,clkB);
    output wire_B;
    reg wire_B;
    input [7:0] B;
    input loadB, clkB;
    reg [7:0] B_reg;

always@(posedge clkB)
if(~loadB) begin
    B_reg = B;
    wire_B = B_reg[0];
end
else begin
    B_reg[6:0] = B_reg[7:1];
    wire_B = B_reg[0];
end

endmodule
```



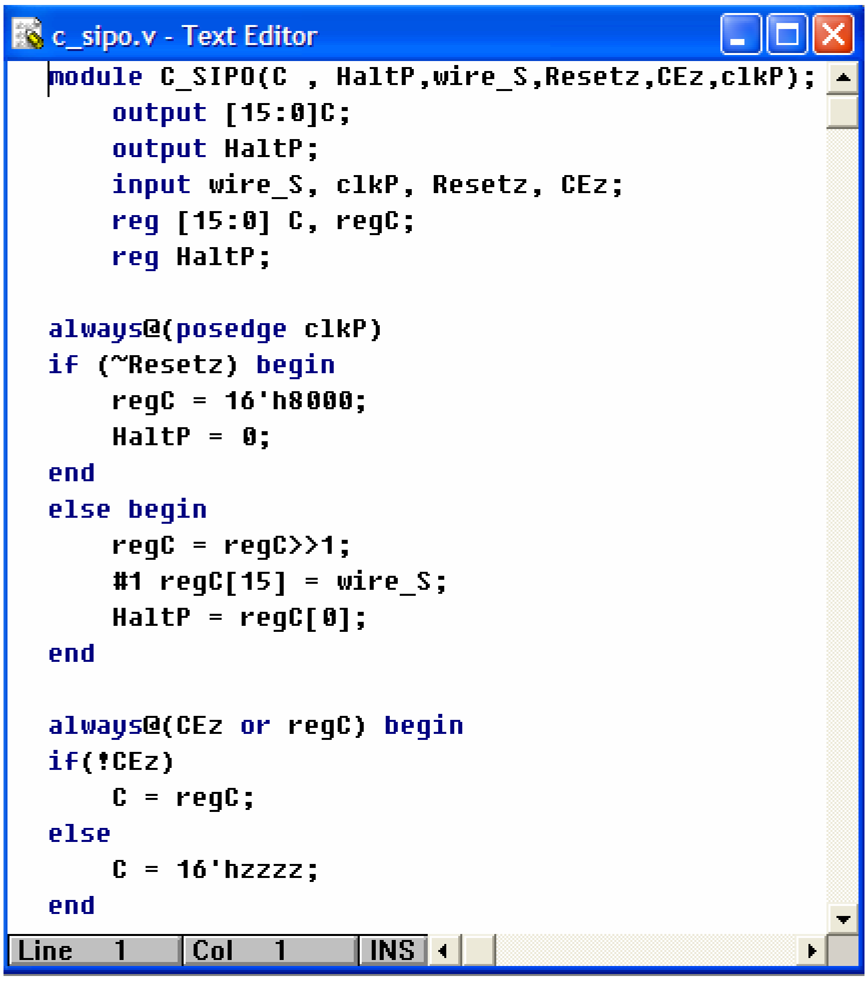
```
module REG_mult(wire_mult, wire_A, wire_B);
    output [7:0] wire_mult;
    input [7:0] wire_A;
    input wire_B;
    reg [7:0] wire_mult;

always @ (wire_B or wire_A)
    wire_mult = wire_B * wire_A;

endmodule
```



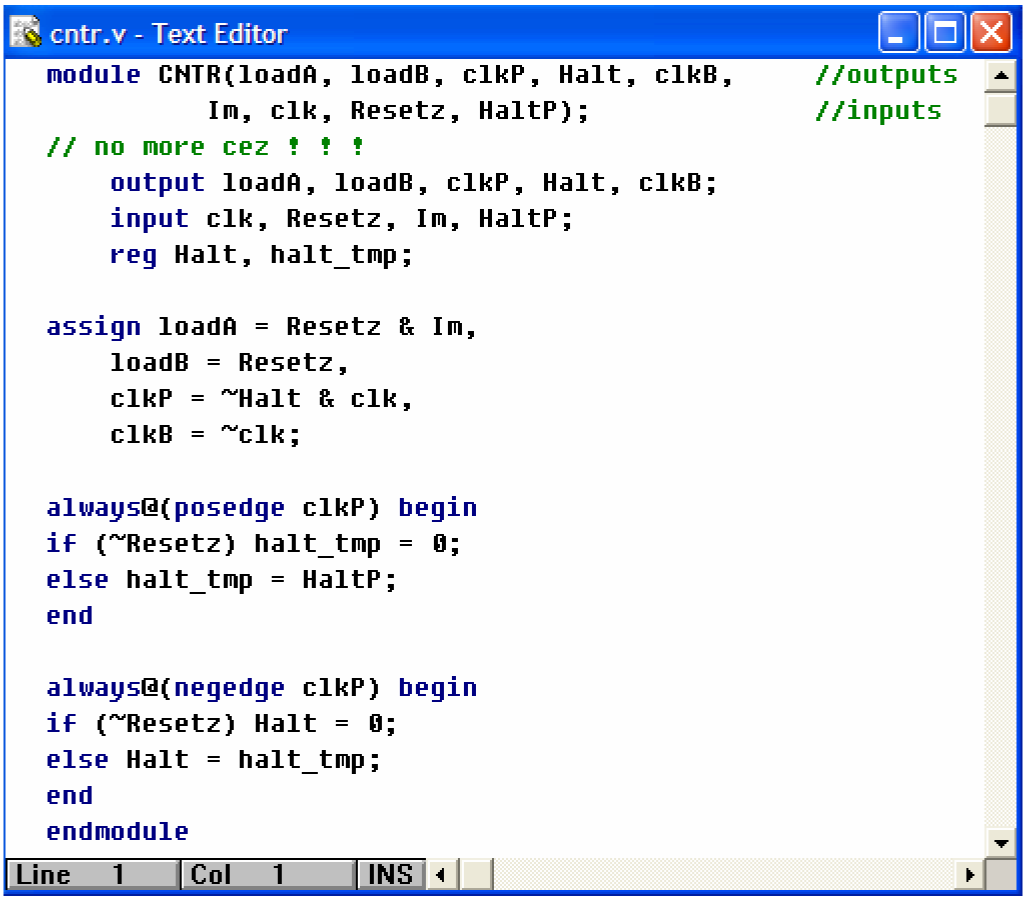
```
module adder_block(So , mult,Si,Resetz,clkP);
    output So;
    input mult, Si, Resetz, clkP;
    reg carry, So;

always@(posedge clkP)
    if(~Resetz) begin
        carry = 0;
        So = 0;
    end
    else if(Resetz)
        {carry, So} = Si + mult + carry;
endmodule
```



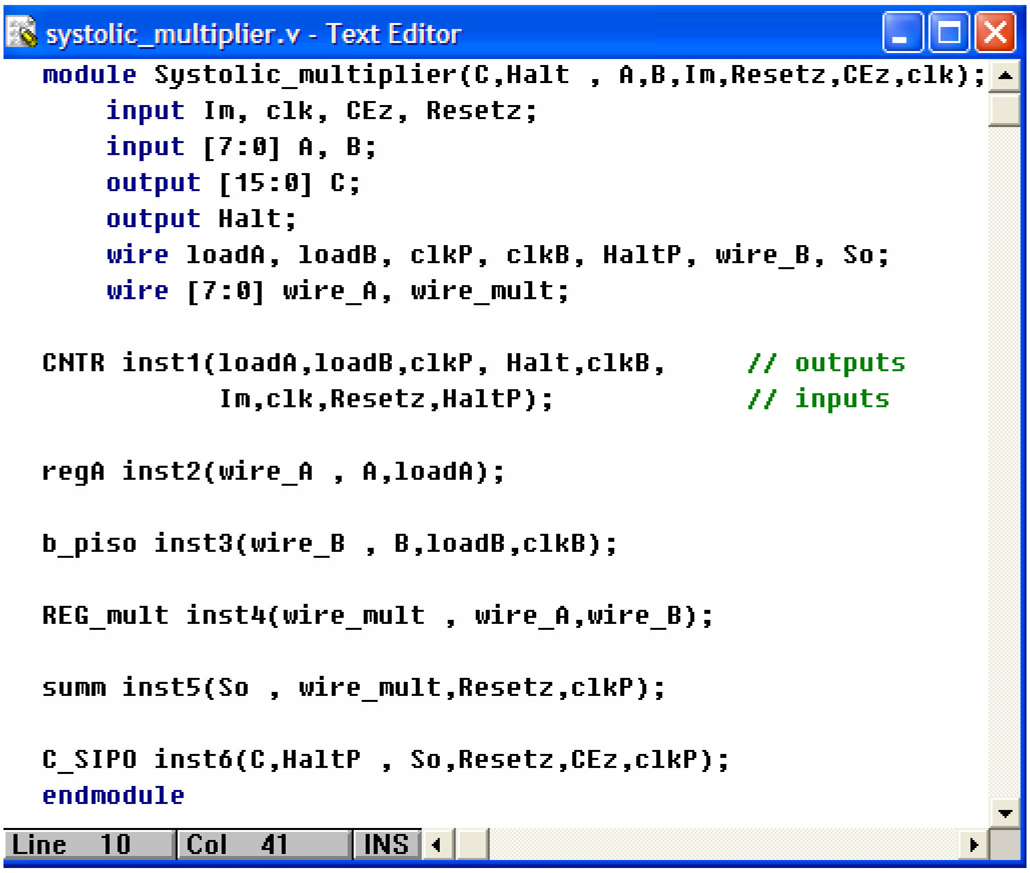
```
module summ(wire_S , mult,Resetz,clkP);
    output wire_S;
    input [7:0]mult;
    input Resetz, clkP;
    wire [6:0]S_int;

adder_block INST0(wire_S,   mult[0], S_int[0], Resetz, clkP)
adder_block INST1(S_int[0], mult[1], S_int[1], Resetz, clkP)
adder_block INST2(S_int[1], mult[2], S_int[2], Resetz, clkP)
adder_block INST3(S_int[2], mult[3], S_int[3], Resetz, clkP)
adder_block INST4(S_int[3], mult[4], S_int[4], Resetz, clkP)
adder_block INST5(S_int[4], mult[5], S_int[5], Resetz, clkP)
adder_block INST6(S_int[5], mult[6], S_int[6], Resetz, clkP)
adder_block INST7(S_int[6], mult[7], S_int[6], Resetz, clkP)

endmodule
```



```
module C_SIPO(C , HaltP,wire_S,Resetz,CEz,clkP);
    output [15:0]C;
    output HaltP;
    input wire_S, clkP, Resetz, CEz;
    reg [15:0] C, regC;
    reg HaltP;

always@(posedge clkP)
if (~Resetz) begin
    regC = 16'h8000;
    HaltP = 0;
end
else begin
    regC = regC>>1;
    #1 regC[15] = wire_S;
    HaltP = regC[0];
end

always@(CEz or regC) begin
if(!CEz)
    C = regC;
else
    C = 16'hzzzz;
end
```

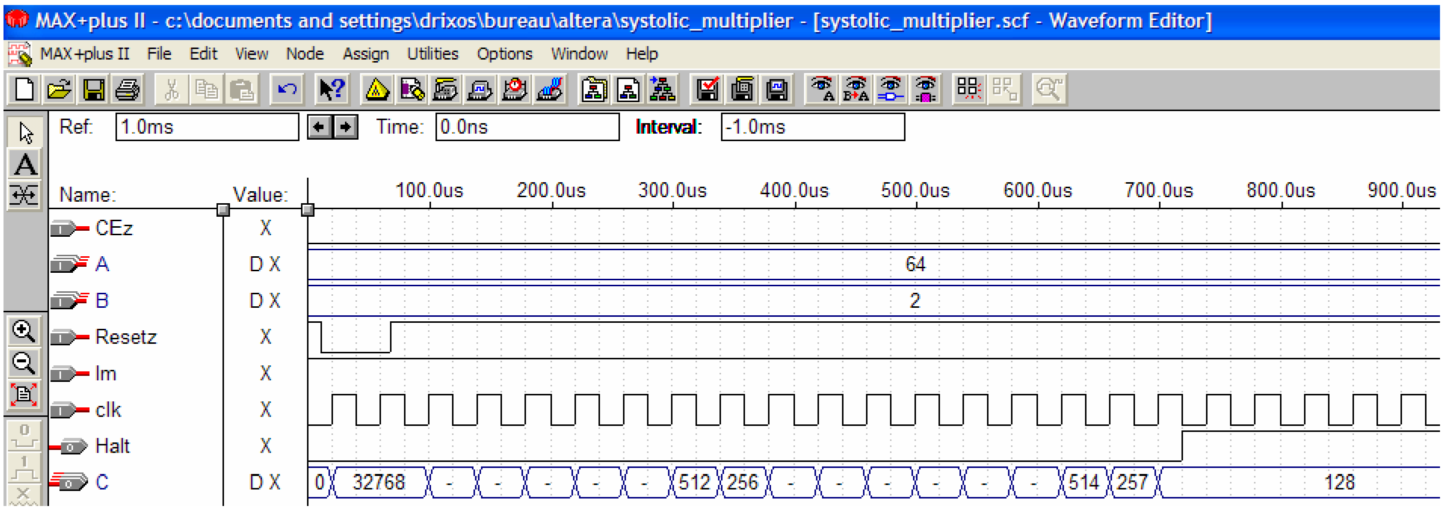


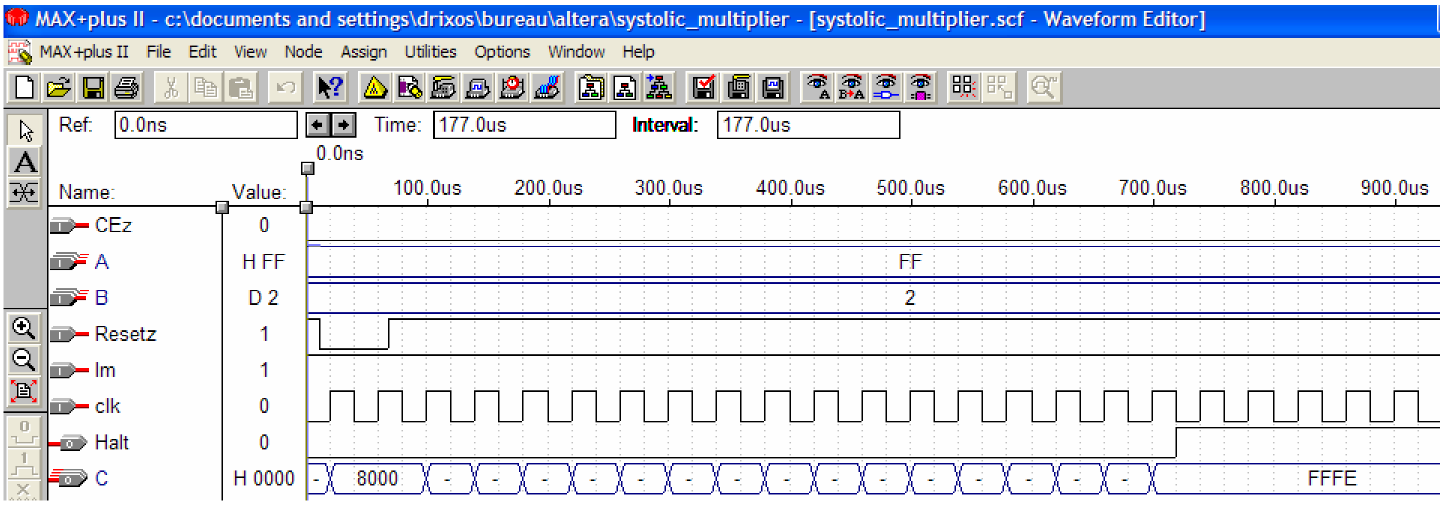
```
module CNTR(loadA, loadB, clkP, Halt, clkB,          //outputs
          Im, clk, Resetz, HaltP);                   //inputs
// no more cez ! ! !
    output loadA, loadB, clkP, Halt, clkB;
    input clk, Resetz, Im, HaltP;
    reg Halt, halt_tmp;

assign loadA = Resetz & Im,
    loadB = Resetz,
    clkP = ~Halt & clk,
    clkB = ~clk;

always@(posedge clkP) begin
if (~Resetz) halt_tmp = 0;
else halt_tmp = HaltP;
end

always@(negedge clkP) begin
if (~Resetz) Halt = 0;
else Halt = halt_tmp;
end
endmodule
```

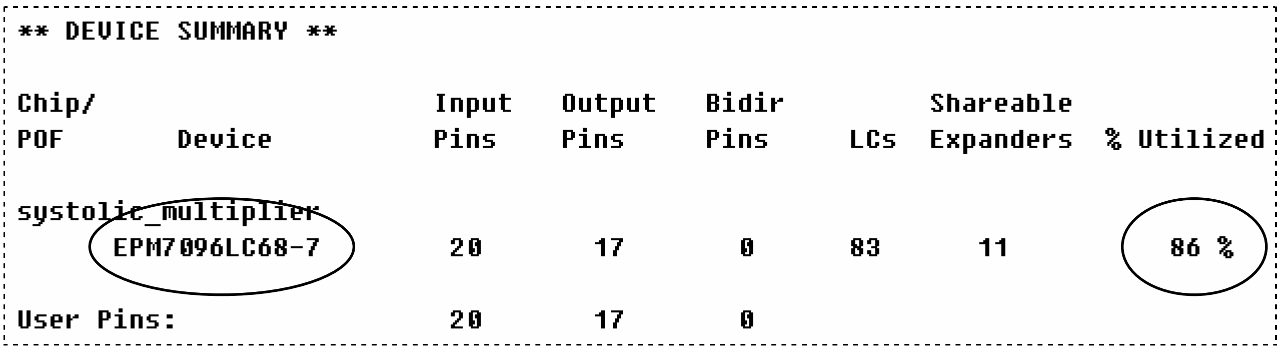


```
module Systolic_multiplier(C,Halt , A,B,Im,Resetz,CEz,clk);
    input Im, clk, CEz, Resetz;
    input [7:0] A, B;
    output [15:0] C;
    output Halt;
    wire loadA, loadB, clkP, clkB, HaltP, wire_B, So;
    wire [7:0] wire_A, wire_mult;

CNTR inst1(loadA,loadB,clkP, Halt,clkB,      // outputs
           Im,clk,Resetz,HaltP);              // inputs

regA inst2(wire_A , A,loadA);

b_piso inst3(wire_B , B,loadB,clkB);

REG_mult inst4(wire_mult , wire_A,wire_B);

summ inst5(So , wire_mult,Resetz,clkP);

C_SIPO inst6(C,HaltP , So,Resetz,CEz,clkP);
endmodule
```

SIMPLE EXAMPLE VALUES FOR THE SIMULATION:







```
** DEVICE SUMMARY **

Chip/                         Input   Output   Bidir        Shareable
POF        Device             Pins    Pins     Pins    LCs  Expanders  % Utilized

systolic_multiplier
     EPM7096LC68-7             20      17       0      83      11         86 %

User Pins:                     20      17       0
```

AS WE CAN SEE THE AUTOMATIC SYNTHESIS FROM VERILOG FILES TAKES MORE PLACE (86% OF THE SAME COMPONENT INSTRAD OF 81)

I'm really happy to have found the time to compare the Verilog synthesis by Maxplus and the gate level synthesis also by Maxplus. This is a good finalisation of the comparison of automatic and manual synthesis. I already knew that the occupation ratio is better in gate level, but now I've seen it for real.

It would have been interesting to test the synthesis size of the behavioural component but time is finished now.

3 Conclusion

This report was a really good approach to the manual synthesis and the gate level fight. The big deal stays in the delay problems, the requirement analysis and the full testing, but we have a good overview of these.

This assignment made me discover a lot of tricks with Maxplus simulator, Silos simulator and also Verilog HDL (definitively confusing considering that I've seen VHSIC HDL last year). However, I have discovered a good overview of these complex uses, but I'm obviously still very far of the full potentials.

My programs are definitively not the only means to reach the aim and obviously, improvements exist but anyway, I'm really proud to have discovered a new design technique and a new HDL, I know it also exists SystemC, A_{ltera}HDL, and more than ten or so but I still have the time…

4 References:

BOOKS: *Fundamentals of DIGITAL LOGIC with Verilog design (Brown Vanesic - Mc Graw Hill)*
DIGITAL FUNDAMENTALS 8th ed. (Thomas FLOYD - Pearson Education International)
The Verilog Hardware Description Language 5th ed. (Thomas & Moorby's - Kluwer Academic)

WEBSITES: http://en.wikipedia.org
http://www.wordreference.com/fren
http://www.cours.polymtl.ca/ele2300/acetates.htm
http://ieeexplore.ieee.org
http://tams-www.informatik.uni-hamburg.de