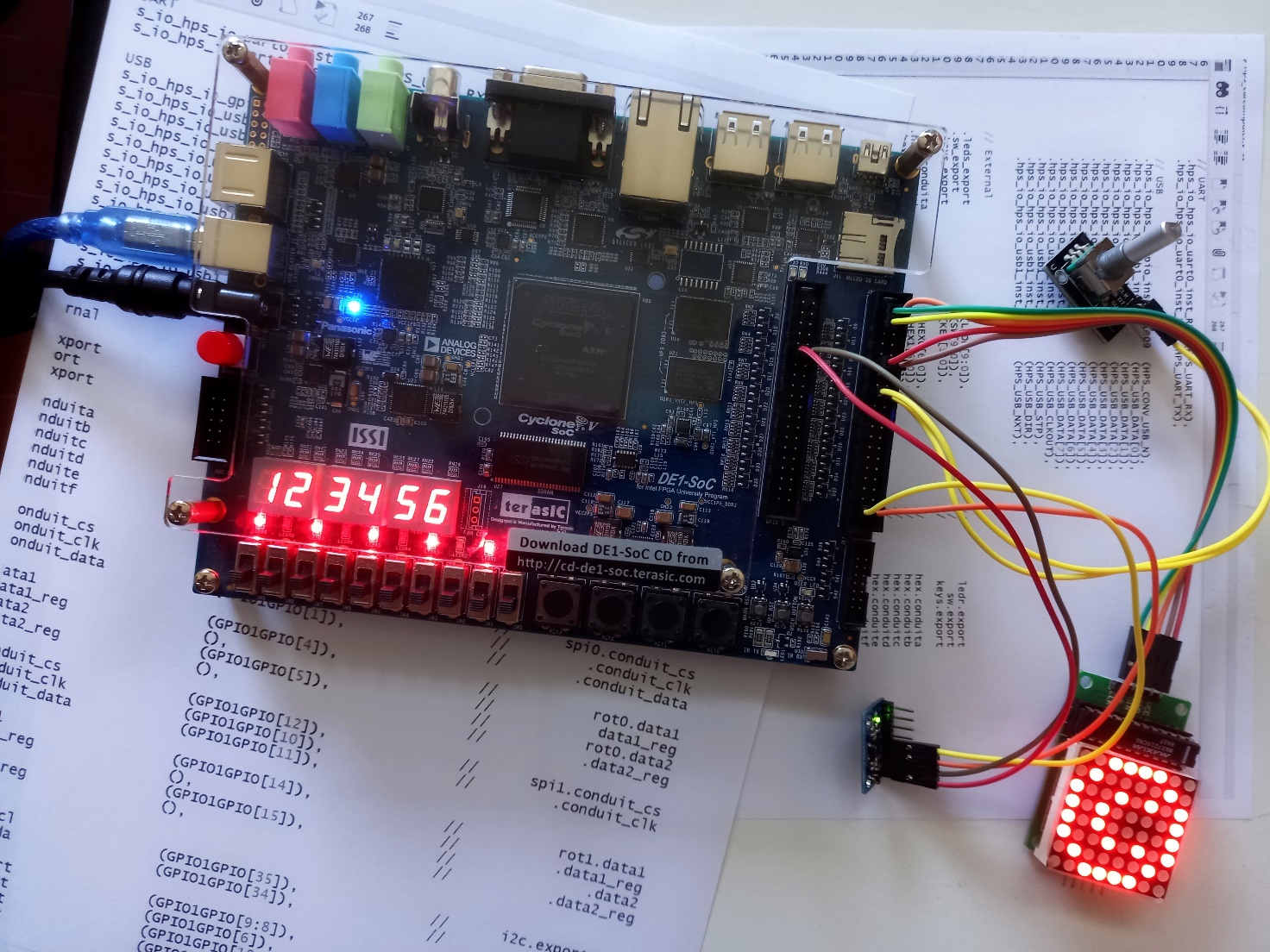
Electronic Systems Design

# Prof. Marsi Stefano - University of Trieste Academic Year 2025/26

Tutorial 2



**Temporal analysis of a system realized on FPGAs.**  
**Hardware** used: Terasic DE1-SoC Board  
**Software** used: Quartus 21.1, System Builder

Tutorial 2

Realization of a hierarchical Sum and Accumulation circuit on DE1-SoC

Description: The projects developed in two parts: In the first part a simple logic circuit is created on DE1-SoC with inputs driven by the switches and output on the LEDs. At each press of a key (Key[0]) system adds the binary value present on the switches to the one present in memory and displays the result on the LEDs. In the second part, we want to investigate the delay times present in a circuit, therefore two adders with different architectures will be created and the results generated by them when the same input data are provided to them will be compared.

Purpose: Introduction to the use of "System Builder" for the realization of systems of a midrange complexity. Familiarize with the realization of a hierarchical system. Explore the tools useful for a temporal analysis of the final device.

Expected learning:

* Using the "System Builder" Tool.
* Hierarchical description and simulation of a system described in Verilog HDL.
* Verification of delay times and implementation constraints (Timing Analyzer)

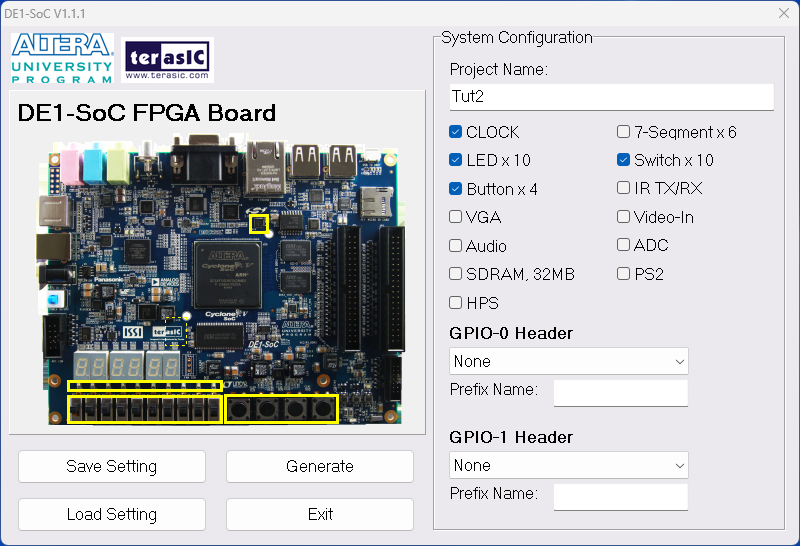
# Procedure

#### Introduction

The company that produces the development board (Terasic) has developed a tool (System Builder), useful for easily starting a project. This software, based on the system required interfaces, creates the skeleton of a project to be developed later on Quartus, but already accompanied by constraint files both for the pinout and for the definition of clock constraints.

* run System Builder
* check the interfaces you intend to use in particular (LED / SWITCHES / BUTTONS / CLOCK )

>> Generate



* Save the project in an appropriate directory

#### Definition of the system architecture

* Open the newly generated project with Quartus   
  (double click on <Your file>.qpf)

The project already has a VerilogHDL file that represents the "TOP Level Entity" of the final project and a .SDC (Synopsys Design Constraints) which contains and will contain the various design constraints. These Files will be used later.   
  
Let's now create a new Verilog file containing 3 modules:

* A Half Adder
* A Full Adder that instantiates the above half-adder
* A 10-bit adder that instantiates the above generated elements.

File > New ( Ctrl-N)

Choose Verilog HDL File

**module** HA ( input a, input b, output s, output c );

assign s = a ^ b;

assign c = a & b;

**endmodule**

**module** FA ( input a, input b, input c\_in, output s, output c\_out );

wire somma\_par;

wire carry1,carry2;

HA HA1(a,b,somma\_par,carry1);

HA HA2(c\_in,somma\_par,s,carry2);

assign c\_out = carry1 | carry2;

**endmodule**

**module** Nbit\_adder(input1,input2,answer);

parameter NBIT=10;

input [NBIT-1:0] input1,input2;

output [NBIT-1:0] answer;

wire carry\_out;

wire [NBIT-1:0] carry;

genvar i;

generate

for(i=0;i< NBIT;i=i+1)

begin: generate\_N\_bit\_Adder

if(i==0)

HA f(input1[0],input2[0],answer[0],carry[0]);

else

FA f(input1[i],input2[i],carry[i-1],answer[i],carry[i]);

end

assign carry\_out = carry[NBIT -1];

endgenerate

**endmodule**

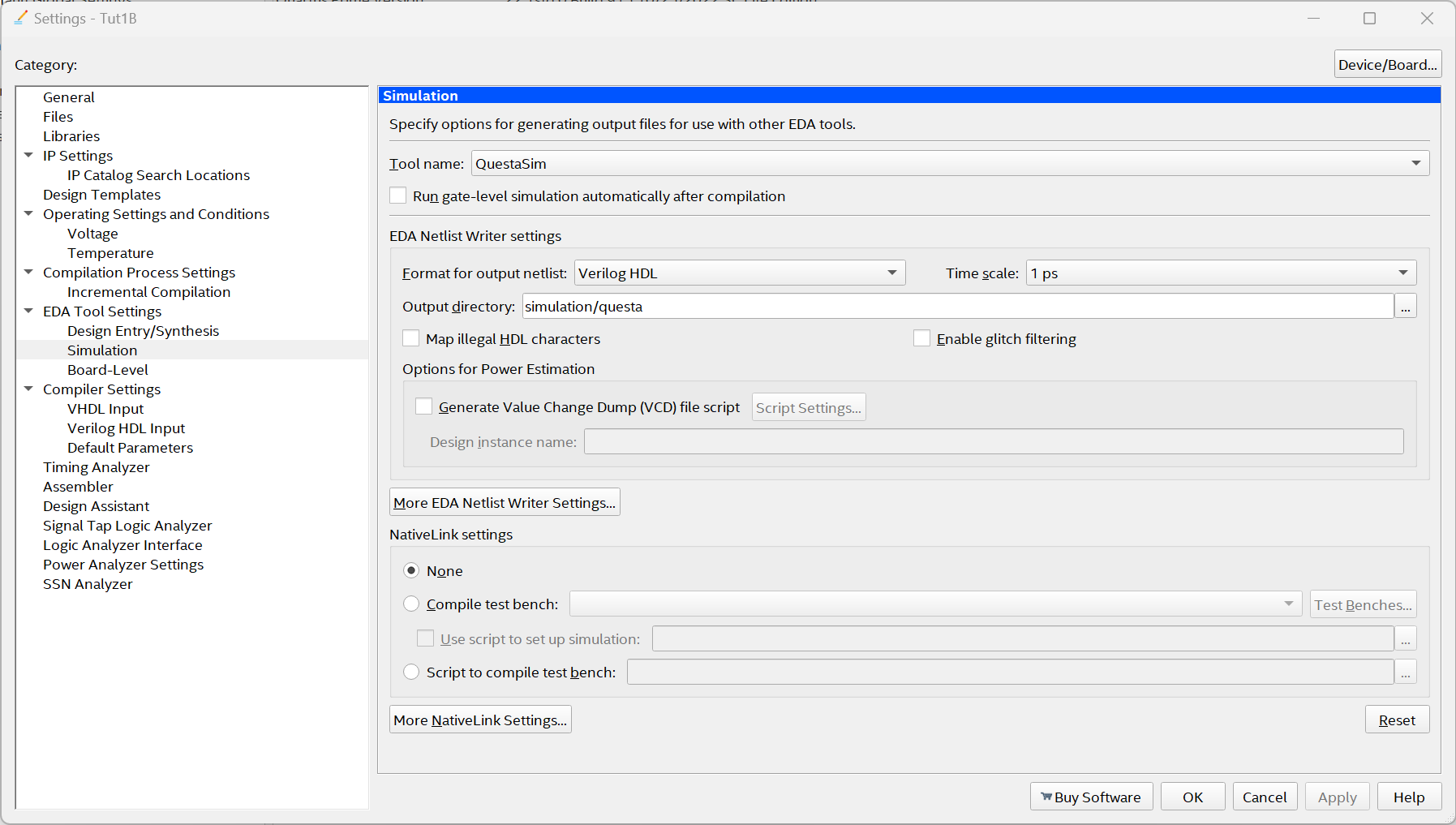
Save the file **WITH THE SAME NAME** as the module at the highest hierarchical level (Nbit\_adder) and run Analisys and Syntesis.

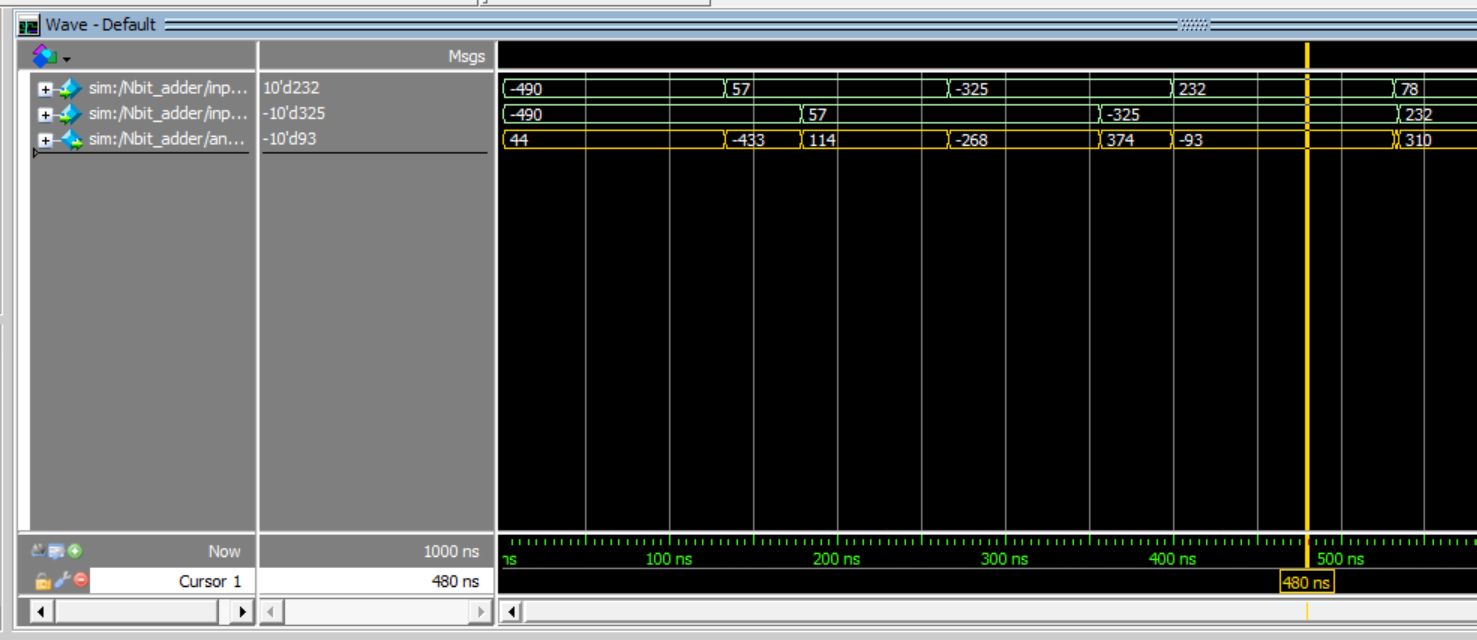
>> Right click on the file you just saved

>> Set as Top Level Enity (Ctrl-Shift-V)

Assignement > Settings (Ctrl-shift-E)

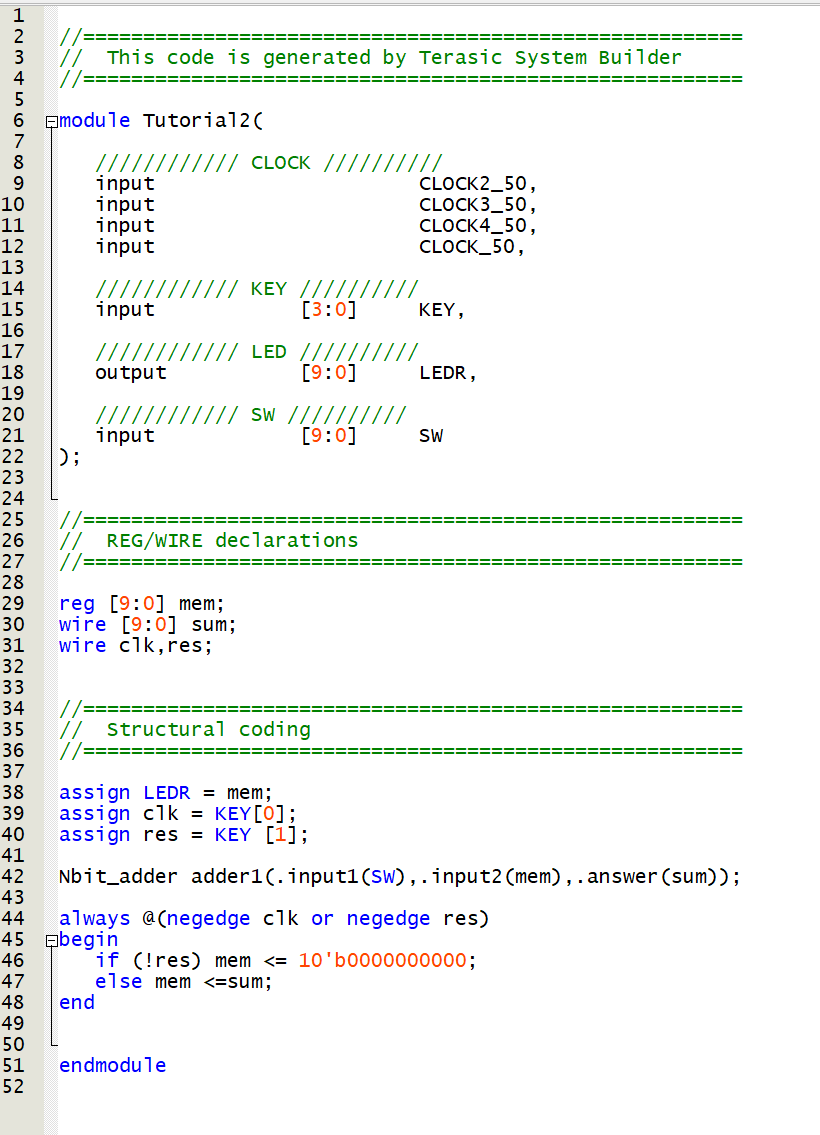
And configure the simulation tool with QuestaSim



Through the simulator (as in the dedicated tutorial) generate appropriate stimuli and verify the correct functioning of the developed adder  
  


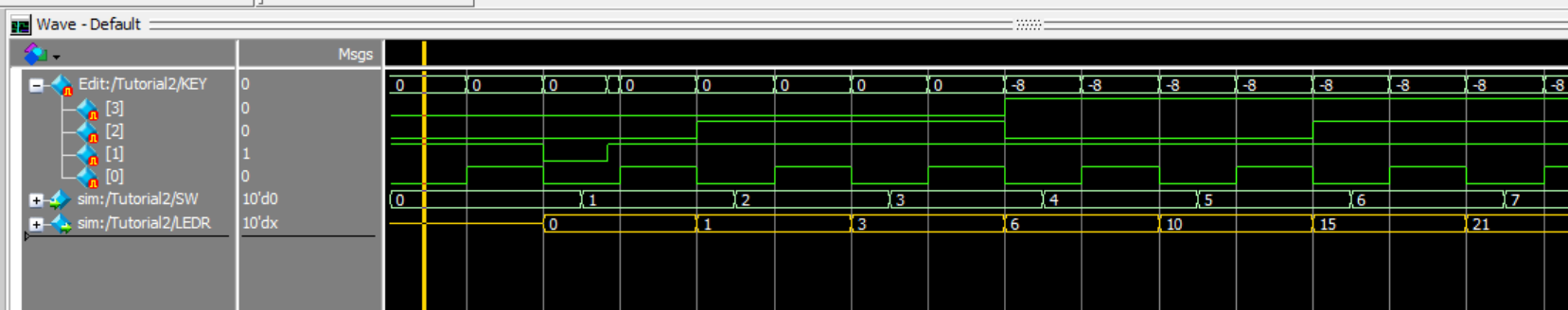
Now it is possible to instantiate the adder just created inside the main block by adding the logic that stores the result of the sum and then provide it as input to the adder in a "add and accumulate" operation using the switches as inputs and the LEDs as outputs.

* If necessary, include the Main file (which present the same name as the project)
* Set the "Top Level Entity" property back to it
* Edit the code as shown below



Note in particular that in this case the signal coming from a button (Key[0]) is used as a clock for storing the results and the one connected to the Key[1] as a reset, both connected in negated logic (normally high). The outputs, stored in the "mem" register, are connected directly with the LEDs on the board.

Re-simulate everything with Modelsim to verify its congruity by providing the KEYS bus with signals that are congruent with what is expected, i.e. a periodic signal on the KEY[0] and a low pulse on the KEY[1]. The signals on KEY[2] and KEY[3] have no effect on the circuit.



#### Download on board

If the simulation is correct, you can complete the compilation up to the generation of the .sof file and then download it to the board to verify that the entire system works correctly.

#### Time Analysis

We now want to examine a different circuit that can highlight the problems related to the creation of a Ripple Carry adder, and in particular its propagation delays.

Let's now modify the project to create a system that presents critical issues regarding signal propagation and see what tools we have available to analyze these critical issues.

The adder realized in the previous point uses a "Ripple Carry" architecture so under particular conditions the adder may have to wait for the "carry" signal to propagate for many consecutive cells, thus reducing the maximum frequency at which the device can operate.

To highlight this criticality, the idea is to create a circuit equipped with two adders: The first will be made with the "ripple carry" technique just seen, while the second will be automatically synthesized by Quartus using an "a+b" (inferred) operation which exploit the internal resources made available by the FPGA and controlled by "Quartus". Both adders will be fed with the same inputs provided as input at the maximum frequency of the board (50MHz) and the results will be compared with each other at the next cycle. If they are different from each other, an error signal will be generated that will be accumulated in a suitable counter.

Therefore, let's first create a module that compares two NBIT bit buses, generates a signal equal to 1 if they differ, and accumulates the results in a suitable register.

**module** Comparator(clk,rst\_n,data1,data2,accum,err);

parameter NBITS=128;

input clk,rst\_n;

input [NBITS-1:0] data1,data2;

output reg [9:0] accum;

output reg err;

reg [NBITS-1:0] diff;

always @(posedge clk or negedge rst\_n)

if (~rst\_n)

begin

err <= 0;

accum <= 0;

end

else

begin

err <= (data1 != data2);

accum <= accum + err;

end

**endmodule**

Then we should also create a system to generate the test signals.

On this point, however, some clarifications must be made: the purpose of the exercise is to verify the criticality of a RC adder. However, this is highly dependent on the data provided as input and not all data are processed with long propagation delays: for example, if you have to add the following two values:

0 1 0 0 1 0 1 1 1 0 0 0 1 0 0 0 1 1 0 1 0 0 1 0 +

1 0 1 1 0 1 1 0 0 1 1 0 0 0 0 1 1 1 1 0 0 1 0 1

The propagation delay in this case corresponds to the time required for the carry signal to traverse 7 cells (despite the data being 24 bits wide). This occurs because, when the two input bits of a given cell are equal, the carry for that cell is immediately known without waiting for the propagation from the preceding cells. As a result, subsequent cells can determine their outputs without depending on earlier carry signals.

This illustrates that when truly random numbers are used, the probability of encountering pairs of operands whose sum requires propagation across many cells is quite low. To clarify with an analogy: if we consider **N-bit** words and ask for the probability that the carry must propagate through more than **m** consecutive cells before settling, this probability is equivalent to flipping a coin N times and obtaining m consecutive identical outcomes (i.e. )

One might therefore consider generating on each test special pairs of words in which the bits at corresponding positions are never equal, thereby forcing the carry to propagate across the entire adder before the final result is available. While this approach could work, an important caveat must be noted: the propagation delay only becomes visible when the adder’s output differs from that of the previous operation (otherwise no propagation can be noted). Consequently, the test data must be carefully crafted to ensure that successive words not only differ in their bit patterns, but also produce a different carry in the first stage, guaranteeing that the carry chain is exercised throughout.

To do this, taking a cue from a generator of pseudo random Fibonacci numbers, the following code was drawn up.

**module** fibonacci\_double(clk,rst\_n,data1\_out,data2\_out);

parameter NBITS=16;

input clk,rst\_n;

output [NBITS-1:0] data1\_out,data2\_out;

reg [NBITS-1:0] data1,data2;

wire feedback,firstbit;

assign feedback = data1[NBITS-2]^ data1[10] ^ data1[4] ^ data1[2] ;

assign firstbit = data1[13];

always @(posedge clk or negedge rst\_n)

if (~rst\_n)

data1 <= 1;

else

begin

data1 <= {data1[NBITS-2:0], ~feedback,firstbit} ;

data2 <= {~data1[NBITS-2:0], feedback,firstbit} ;

end

assign data1\_out=data1;

assign data2\_out=data2;

**endmodule**

This code generates a pseudorandom word and its conjugate at each clock pulse, but with the bit placed in the least significant position the same for both words. By doing so, adding the two words will activate the entire chain of the carry for the entire length of the adder, but this will be (pseudorandomly) sometimes composed only of zeros and sometimes composed completely of ones.

We can now define the complete circuit that performs the sum of the two words through the two different methods and compares the results.

Keeping the "skeleton" provided by the "System Builder" you can modify the source as described below:

//=======================================================

// REG/WIRE declarations

//=======================================================

parameter NBITS = 256; //128 OK //256 Error

wire clk,clk\_num,res\_n;

reg error;

wire[NBITS-1:0] data1,data2;

wire[NBITS-1:0] datafast,dataslow;

reg[NBITS-1:0] diff\_r,data1\_r,data2\_r;

reg[NBITS-1:0] datafast\_r,dataslow\_r;

//=======================================================

// Structural coding

//=======================================================

assign clk=CLOCK\_50;

assign clk\_num=KEY[1];

assign res\_n=KEY[0];

fibonacci\_double #(.NBITS(NBITS)) randgen (clk\_num,res\_n,data1,data2);

always @(posedge clk)

begin

data1\_r <= data1;

data2\_r <= data2;

datafast\_r <= datafast;

dataslow\_r <= dataslow;

end

// Adders

Nbit\_adder #(.NBIT(NBITS)) RC\_adder(.input1(data1\_r),.input2(data2\_r),.answer(dataslow));

assign datafast=data1\_r+data2\_r;

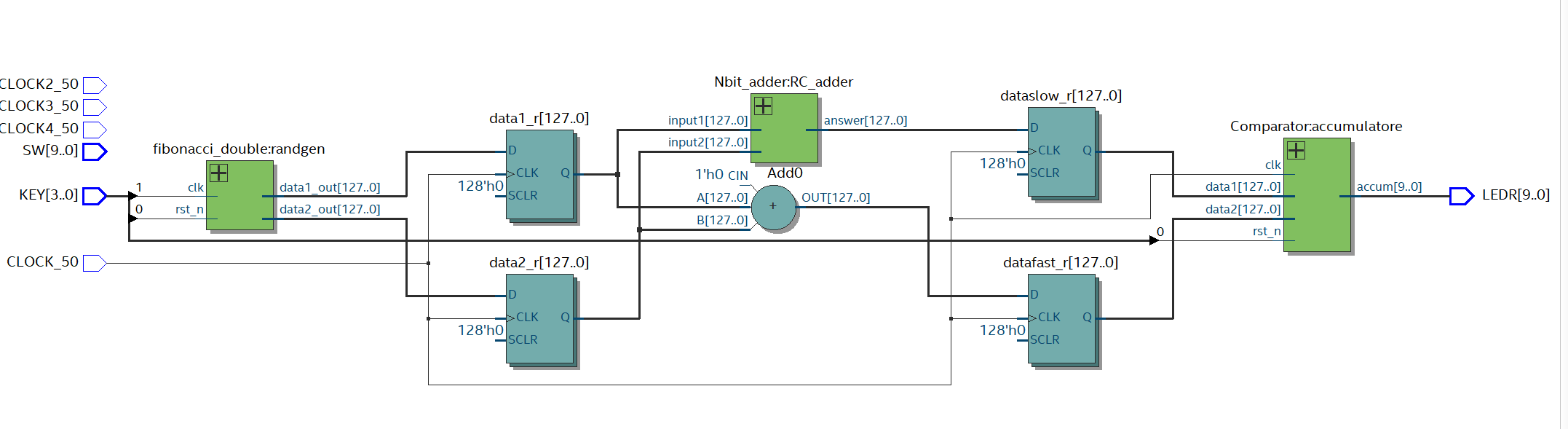
// Comparator

Comparator #(.NBITS(NBITS)) accumulatore(.clk(clk),.rst\_n(res\_n),.data1(dataslow\_r),.data2(datafast\_r),.accum(LEDR),.err());

endmodule

The entire project structure can be depicted using:

>> Tools > Netlist Viewers > RTL Viewer



Build now the entire project. And test it on the board:

* KEY[0] resets the system and puts it in its initial condition
* KEY[1] modify the input samples providing pseudo random inputs

It will be noted that the error counter (which provides its output directly on the LEDs) will sometimes detect an error when generating samples, i.e. the two adders, operating with a frequency of 50MHz, were not able to provide the same result, as at least one of the two was driven with a frequency higher than its maximum working frequency.

Now try to change the number of BITs of the adders, bringing it to 128 for example and recompile the system. During the on-board test you will notice that now the system does NOT detect any errors and both adders work correctly.

#### Using the "Timing Analyzer"

The system created is particularly suitable for analysis through a tool that helps the designer to identify the critical issues related to signal propagation: the "Timing Analyzer".

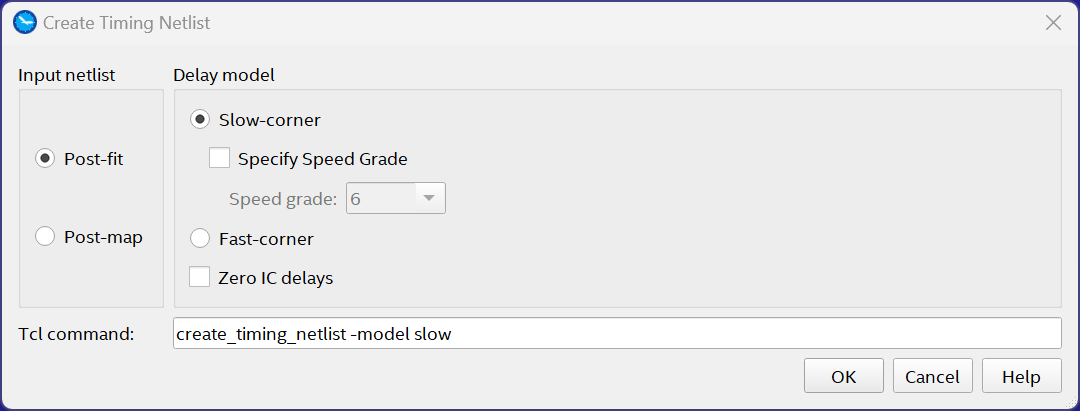
Start by removing from the project any files with the \*.sdc extension that may already be present, usually generated by the "System Builder". These are constraint files that are used in the synthesis phase to verify whether the time constraints are respected and possibly to guide the synthesis and "Place & Route" process and recompile the complete system.

Open the "Timing Analyzer" tool

> Tools > Timing Analyzer

Inside the Tool:

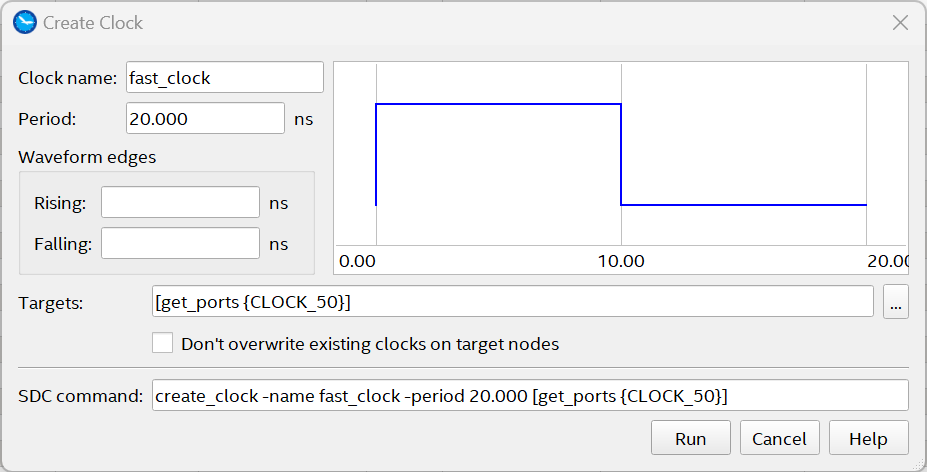
> Netlist > Create Timing Netlist



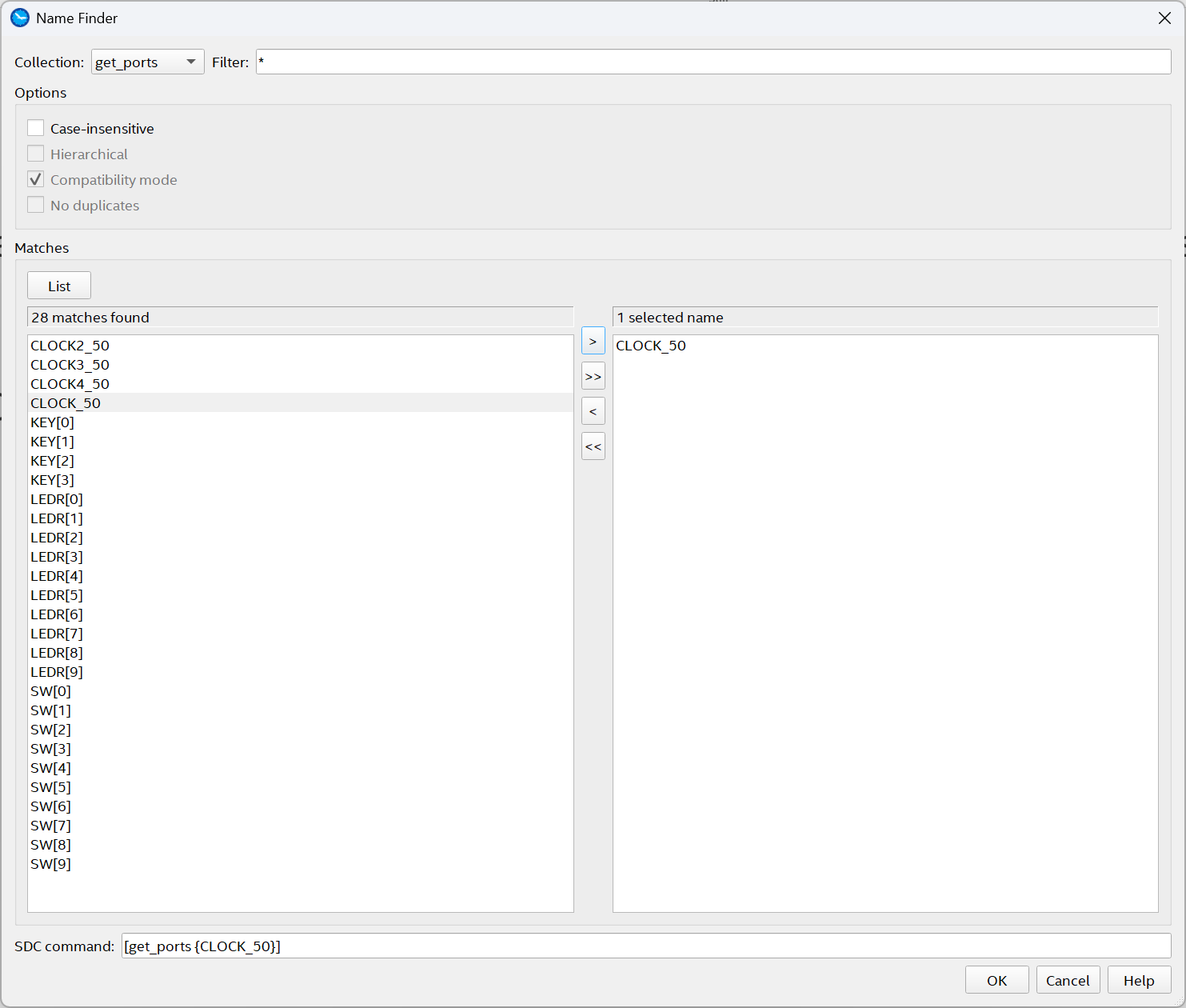
Choose the condition "Slow Corner" and "post-fitting" - <OK>

It is now necessary to indicate to the system what the implementation constraints are, first of all what the clock signals are and at what frequency they act:

> Constraints > Create Clock

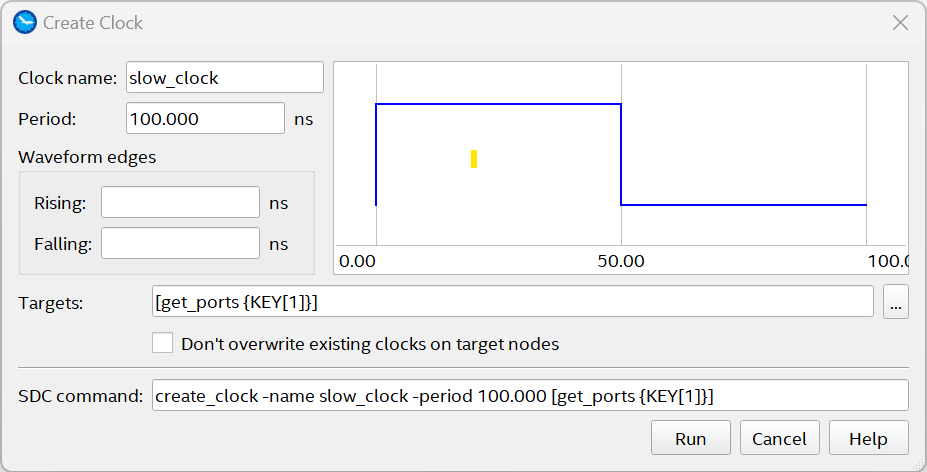


Provide a name for the clock network (e.g.: fast\_clock), its specifications (Period = 20 ns) and the signal to which it is associated (Target = CLOCK\_50) To do this click on " ..." in the window that opens, click on the "List" button and select the appropriate target <OK" with the ">" button>



<OK> <Run>

Repeat the operation by creating a second clock network associated with the clock that generates the random inputs (assuming its period is 100 ns) associated with the KEY[1] port.



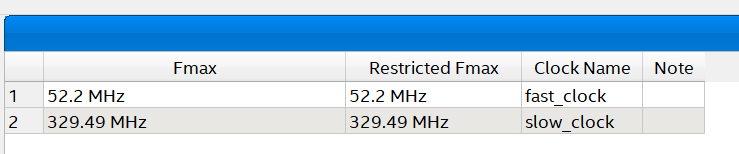
<Run>

> Netlist > Update Timing Netlist

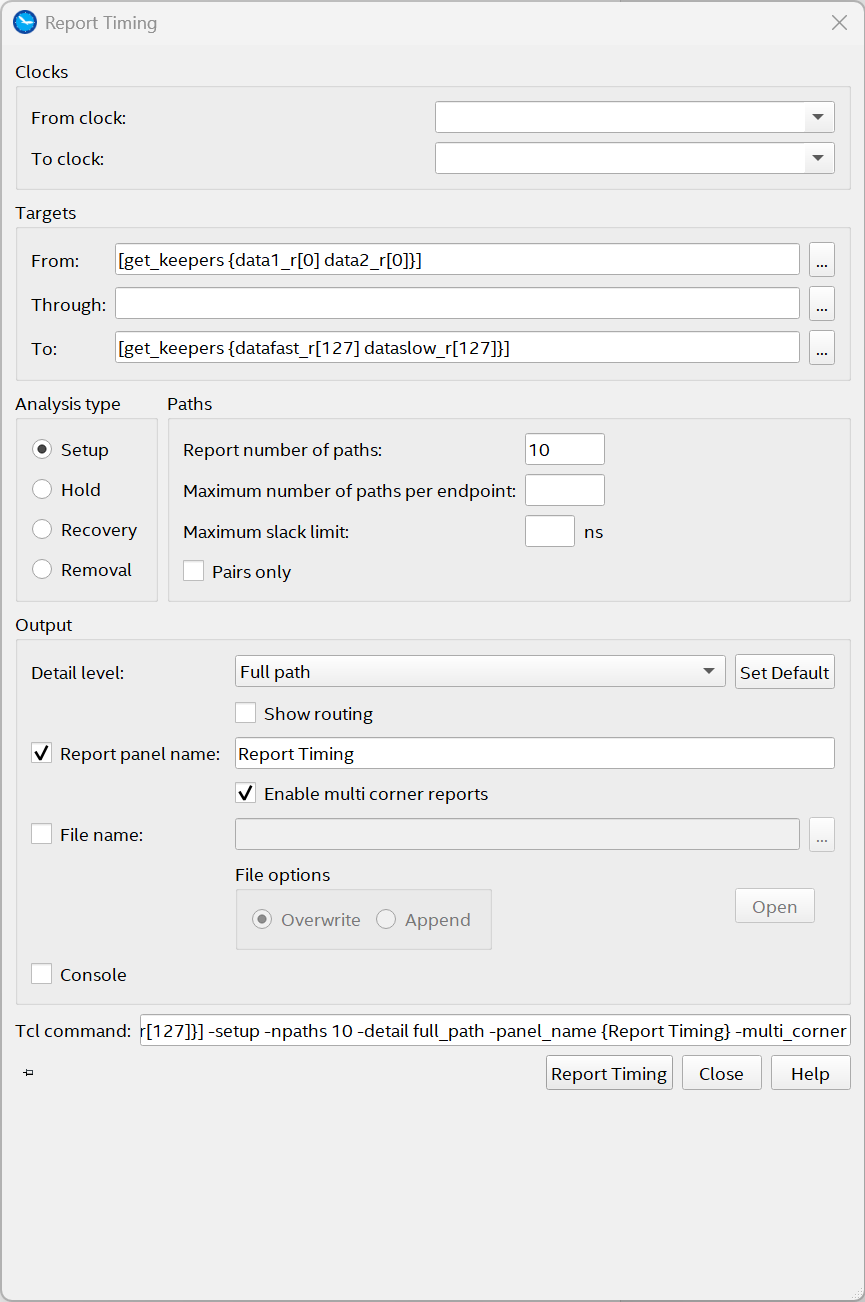
Now there are several "reports" related to the project made, of which we will analyze some of them, however leaving the reader the faculty and advice to analyze other information that the tool provides.

> Report > Datasheet > Fmax Summary Report

Which returns information on the maximum working frequency related to the two clock circuits



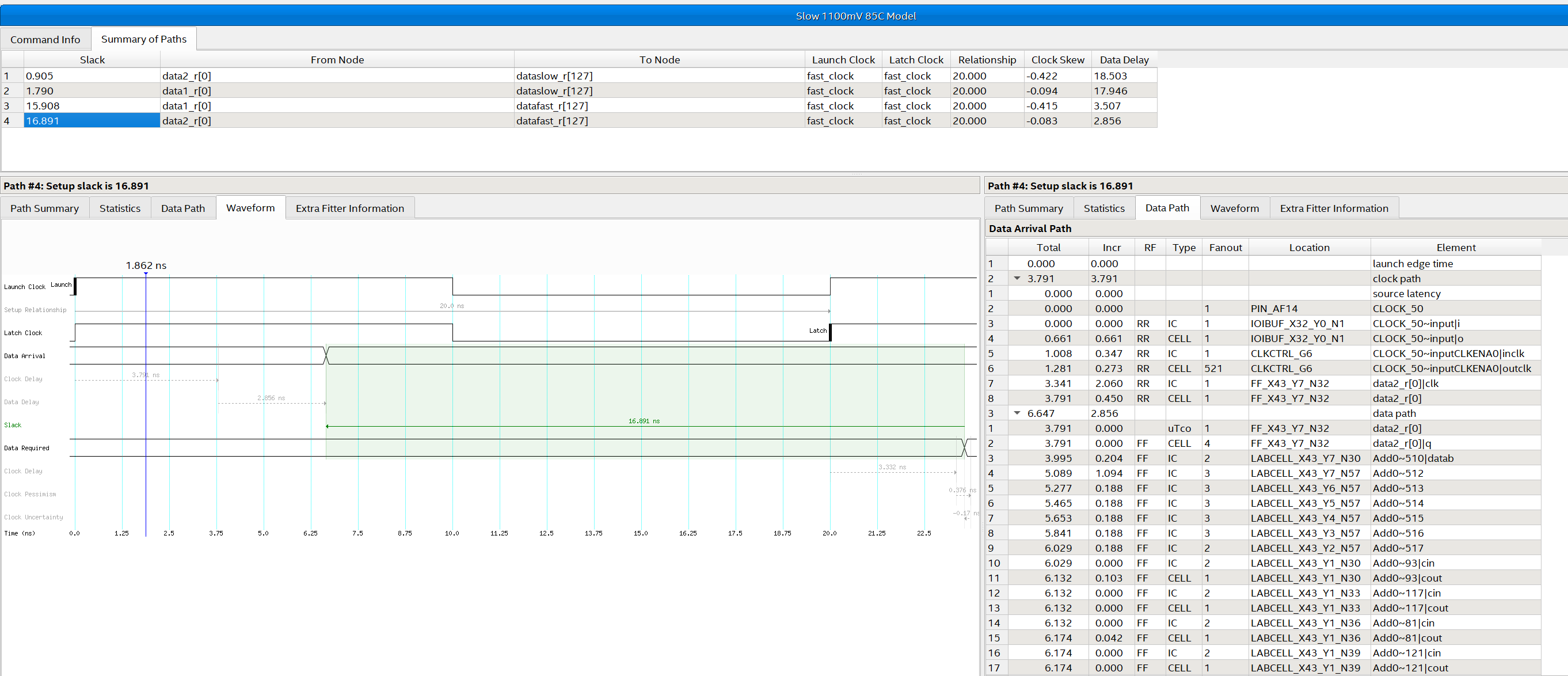
> Report > Report Timing



Choose some paths of interest, such as those that involve both adders from the least significant bits as inputs to the most significant bits as outputs (as in the example)

<Report Timing>

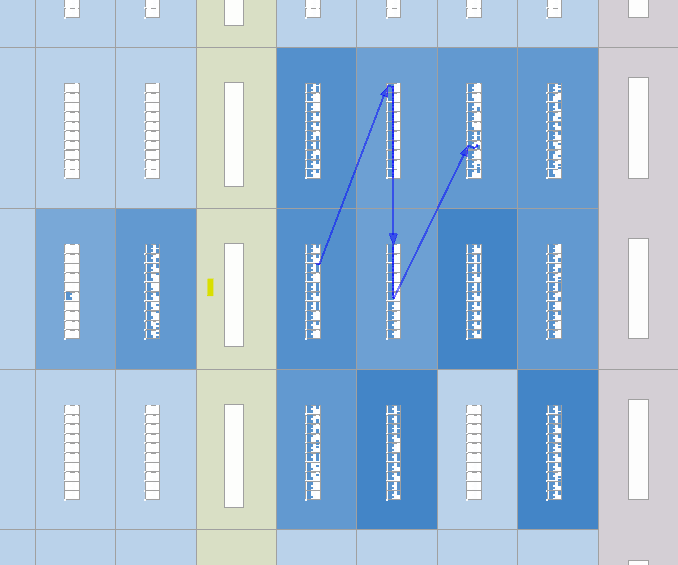
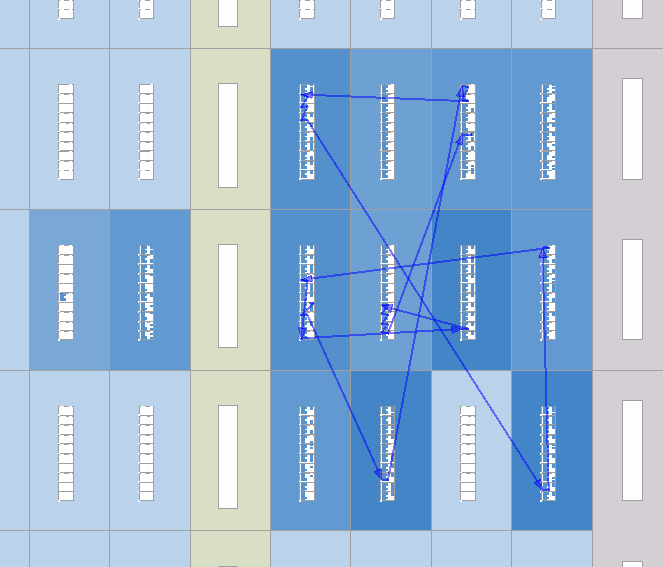
On the following page you will be reported the 4 paths identified and by clicking on each of them the timing of signal propagation, any violations, the location of the path, statistics etc. etc.



It can be seen how the "fast" adder is able to switch the entire carry chain in less than 4ns, while the Ripple Carry adder made in a hierarchical form with separate elements takes more than 18ns.

By clicking with the right button on the path of interest you can locate the path in the various views (Chip Planner – Technology Map Viever – Resources Property Editor)

For example, compare the different paths in the "Chip Planner"



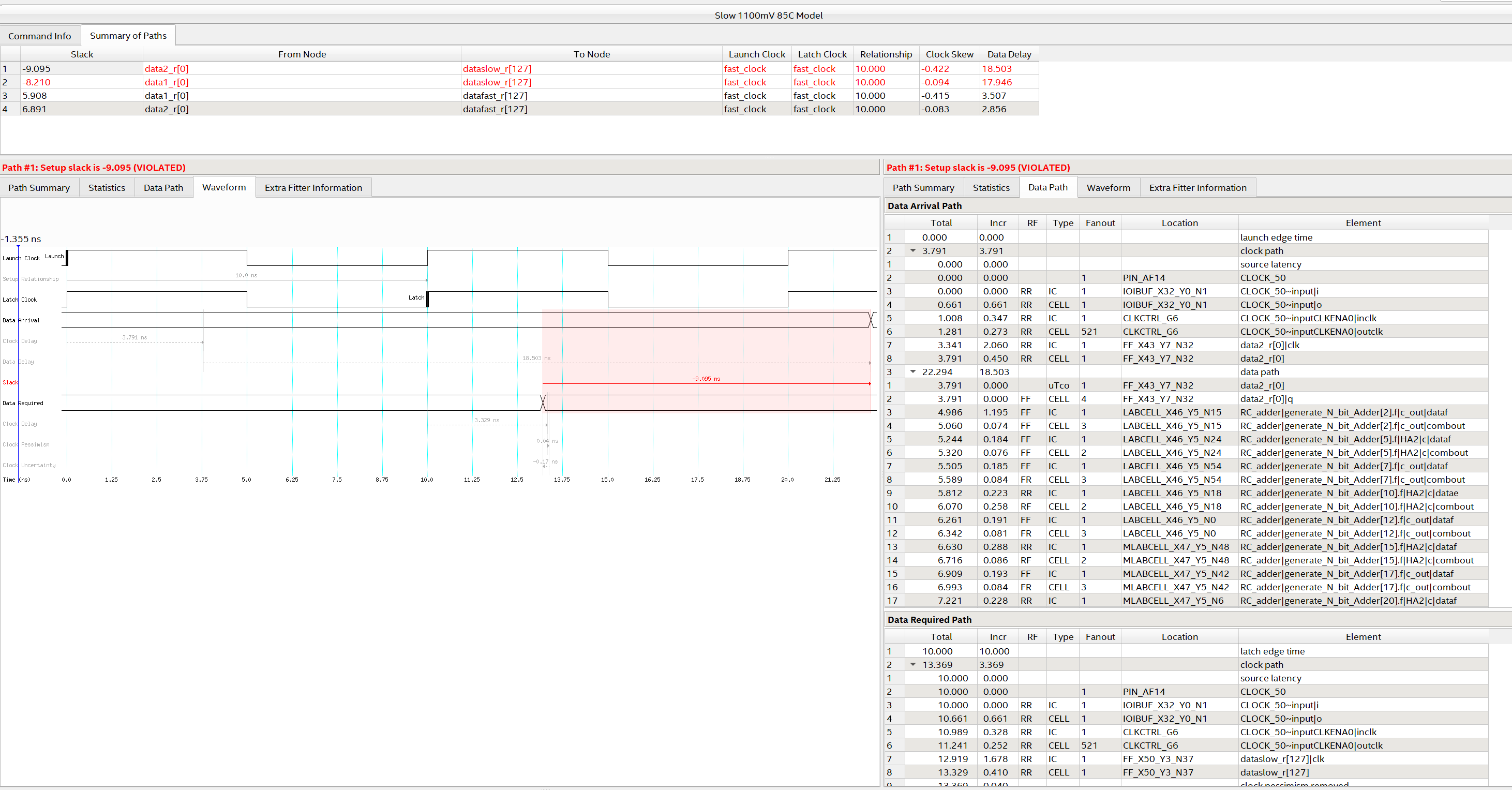
Now suppose that our system, instead of being driven by a 50MHz frequency, was synchronized by a 100MHz frequency.

Erase the time data found so far

> Netlist > Delete Timing Netlist

And repeat the above operations but modifying, for example, the period of the "fast\_clock" setting it to 10ns.

Now the report will highlight the violations that will highlight how certain signals, in particular those that travel through the "ripple carry" adder, arrive too late compared to the clock front that has to sample them:



Particularly interesting is also the tool:

> Reports > Custom Reports > Report Timing Closure Recommendations...

Which opens a page where, for the most critical routes, it recommends the most suitable actions to be taken to avoid critical issues.

The file containing all the constraints set (with the extension .sdc) can now be saved and included in the project.

> Constraints > Write SDC File ...

Its usefulness within the project is twofold:

* on the one hand, it serves to guide the "mapping and fitting" process to better organize the positioning of the logic within the FPGA in an attempt to reach in imposed constraints
* on the other hand, the system, if it is ever unable to meet the constraints, will report this lack with special messages.

It should also be emphasized that the use of a constraint file cannot improve so much the performance of a system. More or less it will be able to improve by a maximum factor of among 10% (and always assuming that the constraints imposed are reasonable). To achieve significantly higher performance, it is necessary to start from the project itself by using, for example, pipelined systems or parallelizing the operations to be performed, or using at the best the available resources.

