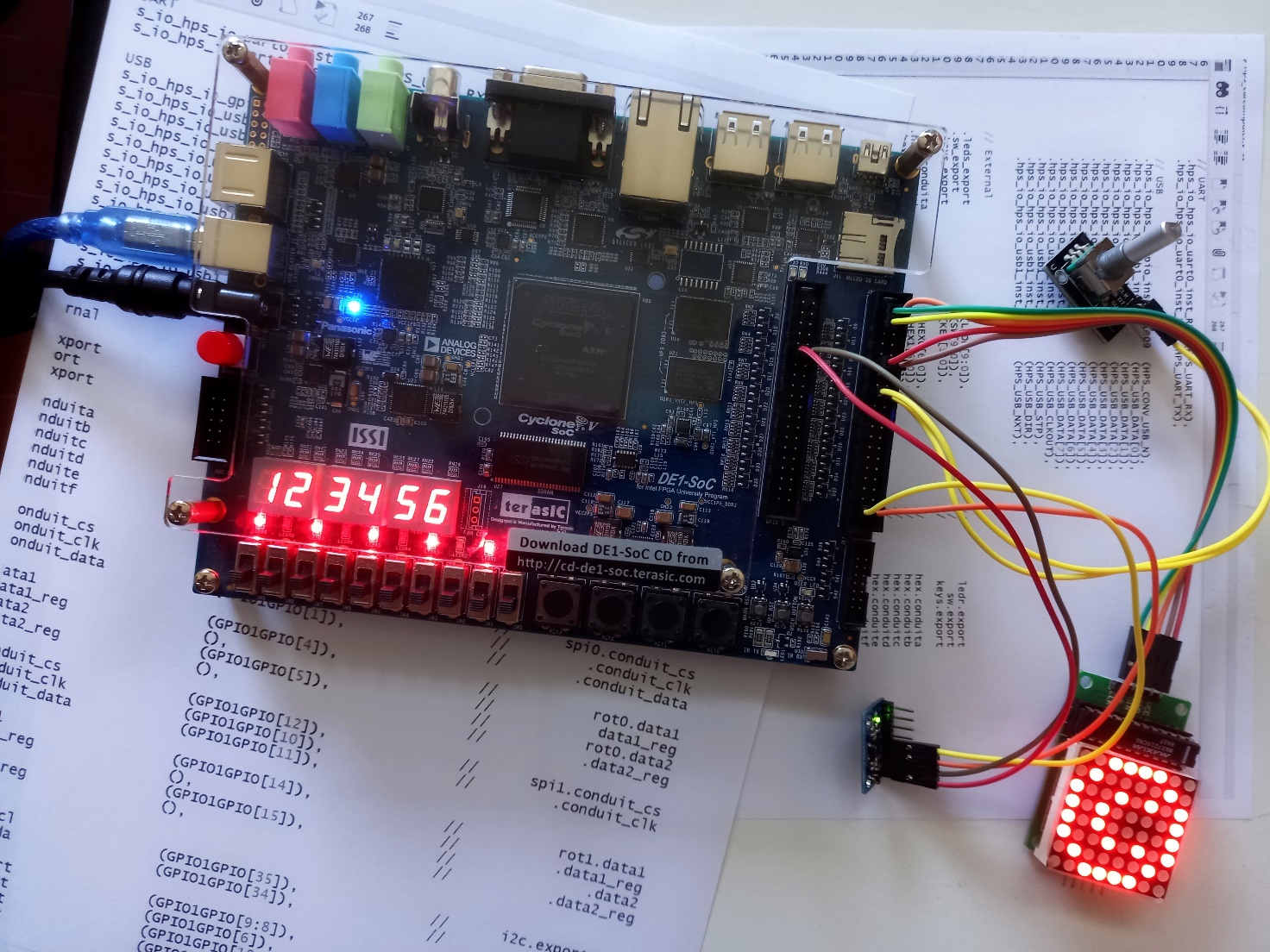
Electronic Systems Design

# Prof. Marsi Stefano - University of Trieste

# Academic Year 2025/26

Tutorial 4



**Design of an SPI interface system.**  
**Hardware** used: Terasic DE1-SoC Board  
**Software** used: Quartus 22.1, System Builder

Tutorial 4

Realization of a system for generating an SPI signal

Description: In this tutorial, we will implement on FPGA a modular system designed to generate an SPI signal for controlling any device that uses this protocol. Specifically, the SPI interface will be employed to program the registers of a system managing an LED matrix driven by the MAX7219 chip.

Purpose: The purpose of this tutorial is to familiarize you with external interfaces and with the generation of signals that underlie particular specifications through the definition of a system suitable for this purpose. The use of a tool, developed within Quartus, useful for monitoring the temporal evolution of signals, will also be explored.

Expected learning:

* Definition of a complete SPI interface system through a hierarchical scheme.
* In-depth study of the Verilog language and the Questa/ ModelSim simulator.
* To familiarize with the "Logic Analyzer" tool to monitor the internal signals of the FPGA.
* To familiarize with the In-System Memory Editor to interact with memories.

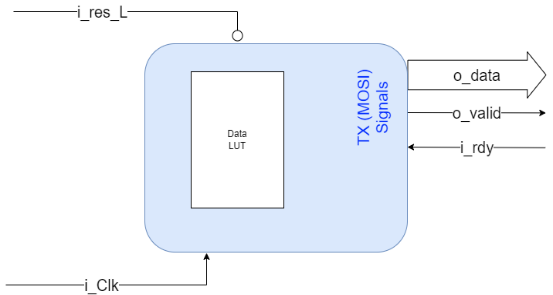
# Premise

Unlike the previous tutorials, this time you will start by defining and simulating the system you want to develop outside the integrated Quartus system, and then import it, once the simulations have given a positive result within the synthesis tool.

Although the SPI protocol can be used for both reading and writing, in this case it will only be used for writing, which leads to a simplification of the complete system.

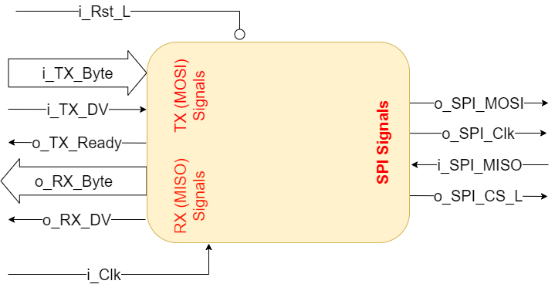
The system that will be implemented essentially consist of two blocks:

* The first one: in it the whole series of data that has to be sequentially transmitted reside. Data is stored in a LUT (Look Up Table). A control system, through a handshaking protocol provides their values (when required through a i\_rdy signal) to a parallel bus interfaced to the second block. The first block provides also a "data valid" information when the data is available and stable.



* The second block, interfaced with the first, on the other hand, generates the request for data (i\_rdy) and when received, generates the sequence of f.d.o that populate the SPI protocol.

This second block is also designed to be able to read data through the SPI protocol, but this part will not be used in this exercise.



# Verilog Files and Simulations

The entire project is described in three VerilogHDL files available on the moodle site of the course:

* SPI\_Master16b.v: Describes the block that generates SPI signals
* SPI\_LUT: Descive block storing data to be sent
* TOP\_Spi: Merges the two blocks into the overall device

Use the simulation tool to analyze and validate the operation.   
(For detailed operation, please refer to the Specific tutorial).

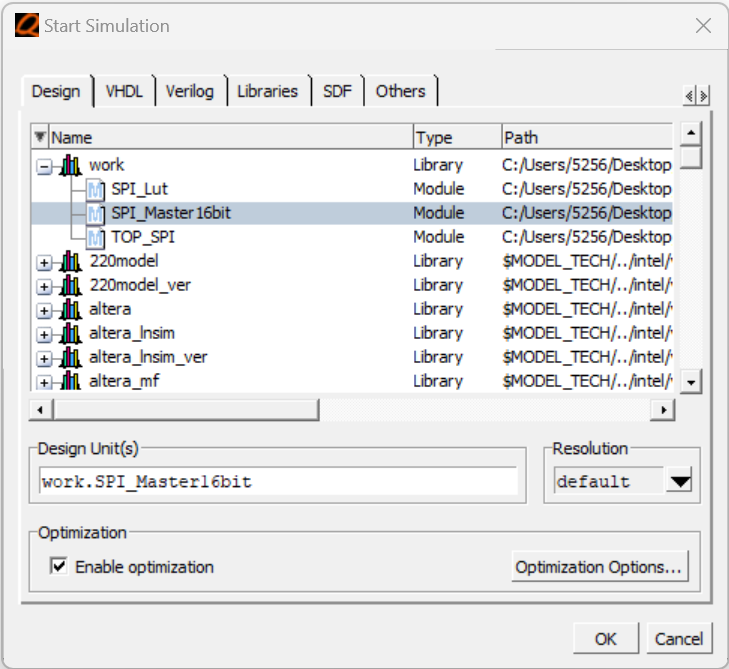
* Open the simulation tool (ModelSim/Questa)
* Create a suitable project
* Import the three files
* Compile all the files

### Simulating SPI\_Master16b

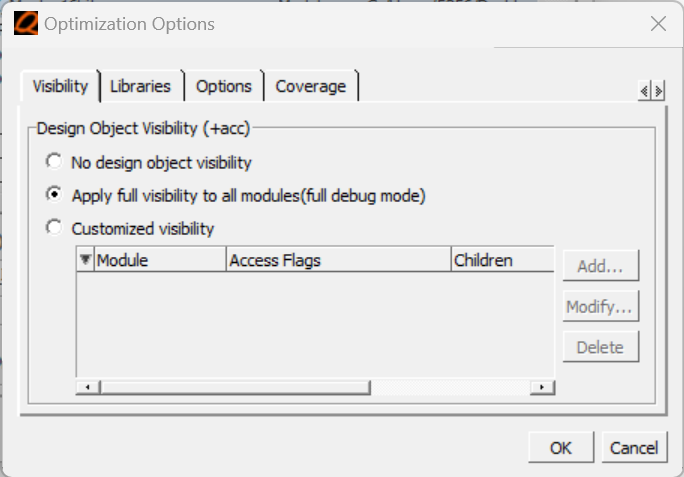
Let's now move on to the simulation of the "SPI\_Master16b" module

> Simulate > Start Simulation

In the window, select the chosen module and click on "**Optimization Options**"



Select "Apply Full Visibility" which allows you to monitor all signals and processes within the module



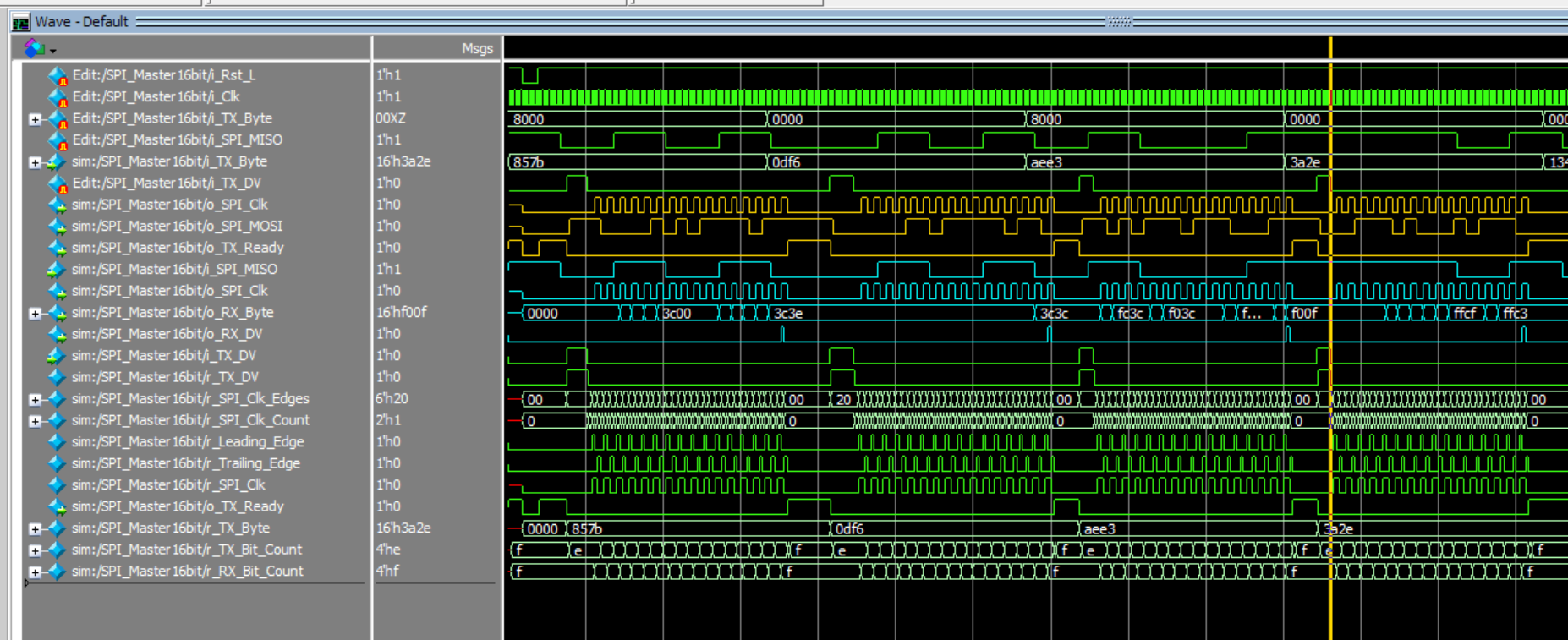
> OK

> OK

Let's now create suitable signals for all the inputs, in particular:

* A unique negative pulse on the i\_Rst\_L signal
* A clock on i\_Clk
* The data that you would like to transmit on the i\_TX\_Byte bus (note that due to a bug in the tool these data, when displayed in Octal or Hexadecimal format, are represented, when they appear as "stimuli", only according to their most significant bit. Therefore, they must be also displayed as an "internal signal" to allow a representation with all the significant bits)
* Pulses for the "data valid" signal i\_TX\_DV interspersed with the previous data (note that these pulses, should be sufficiently spaced to allow the device to complete the transmission before a new transmission is requested .  
  (If you like you could test what happen if "data\_valid" pulses are too close each others)
* Any MISO read signals to test the reading ability of the device i\_SPI\_MISO

It is now possible to simulate the module by highlighting all the signals both present in the various registers and at the outputs.



Note in particular (by zooming in and/or highlighting specific signals) and comparing how it works with the VerilogHDL Source code.

Note about the generation of SPI protocol data, in particular:

* The procedure is initialized when i\_TX\_DV is put at high level (by resetting the counter r\_SPI\_Clk\_Edges) and it starts only when it passes from the high to the low state
* The "r\_SPI\_Clk\_Count" clock cycle counter with respect to which the pulses relating to the arrival of a Leading\_edge or Trailing\_Edge r\_Leading\_Edge are defined, r\_Trailing\_Edge saved in the appropriate registers
* Register signal generation r\_SPI\_Clk
* The (backwards) counter of the fronts r\_SPI\_Clk\_Edges
* The generation of "o\_TX\_Ready" used to request a new data is active only when the previous counter has reached 0

Then note the memorization of two "recorded" versions **r\_TX\_Byte** and **r\_TX\_DV** of the respective input signals **i\_TX\_Byte** and **i\_TX\_DV** in order to keep the data to be transmitted stable even if the inputs change.

Note then the counter related to the bit to be transmitted "r\_TX\_Bit\_Count", its reset to the maximum value (0xF = 15) following a global reset or following the "o\_TX\_Ready" signal and its decrease in correspondence with a "Leading Edge" or a "Trailing edge" depending on the chosen transmission mode, and how based on the value assumed by the latter on the output the desired bit is formed:

1. o\_SPI\_MOSI <= r\_TX\_Byte[r\_TX\_Bit\_Count];

Note the generation of the output signal "o\_SPI\_Clk" which takes the default value in case of reset or alternatively the value present in the "r\_SPI\_Clk" register

Finally, note the reading operation of the SPI protocol: on the i\_SPI\_MISO line, data are sequentially supplied, these, based on a counter r\_RX\_Bit\_Count are continuously supplied directly to the o\_RX\_Byte output, which therefore continues to settle bit by bit with each change in the input data.

221 o\_RX\_Byte[r\_RX\_Bit\_Count] <= i\_SPI\_MISO;

When all the data are present and stable on the output bus, the o\_RX\_DV signal is activated , which could be used by a receiver to determine the instant at which the complete data is read.

### Simulation of SPI\_LUT

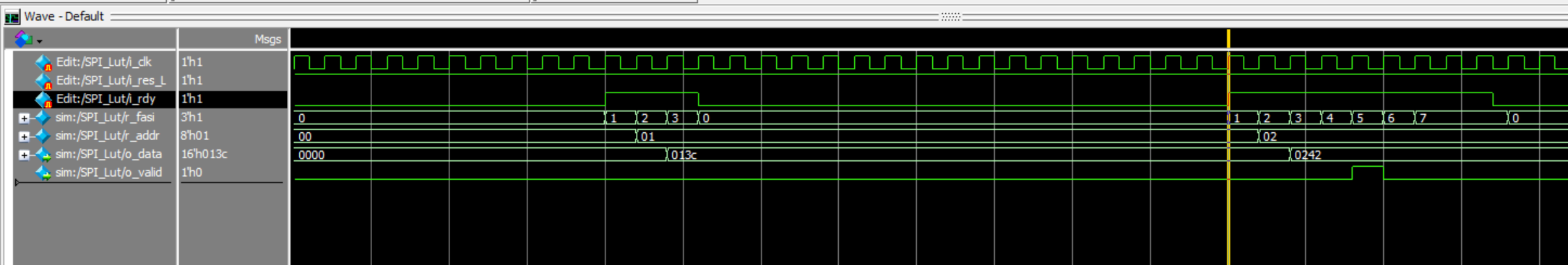
With the procedure suggested in the previous point, let's now consider how to simulate the "SPI\_LUT" module:

Generate the necessary inputs:

* A clock signal
* A negative pulse on the reset
* A series of more or less large pulses on the i\_rdy signal

It can therefore be seen, for the way the system has been designed

* That when a request arrives through i\_rdy, the procedure to provide the data starts
* This starts a "phase counter" r\_fasi that
  + At time 2 increments the address of the ROM r\_addr from which to take the data
  + At time 3 this data is provided as output o\_data
  + At time 5 the o\_valid signal is activated
* When i\_rdy lowers again, the phase counter resets and the next time the procedure is requested. Once the data in memory has been exhausted, the transmission starts again from the first.
* Note again that for a correct behavior the i\_rdy signal must remain high for a sufficient number of clocks to ensure the generation of the o\_valid signal



However, it should be emphasized that the i\_rdy signal will be generated by  ***the Master\_spi16b***  block and will be brought back to the low state only after when the Master\_spi16b block receive from SPI\_Lut the data to be transmitted and the "data\_valid" signal.

Finally, note that the first data in memory at position 0 is not transmitted, so an empty "dummy" data can be introduced at that position.

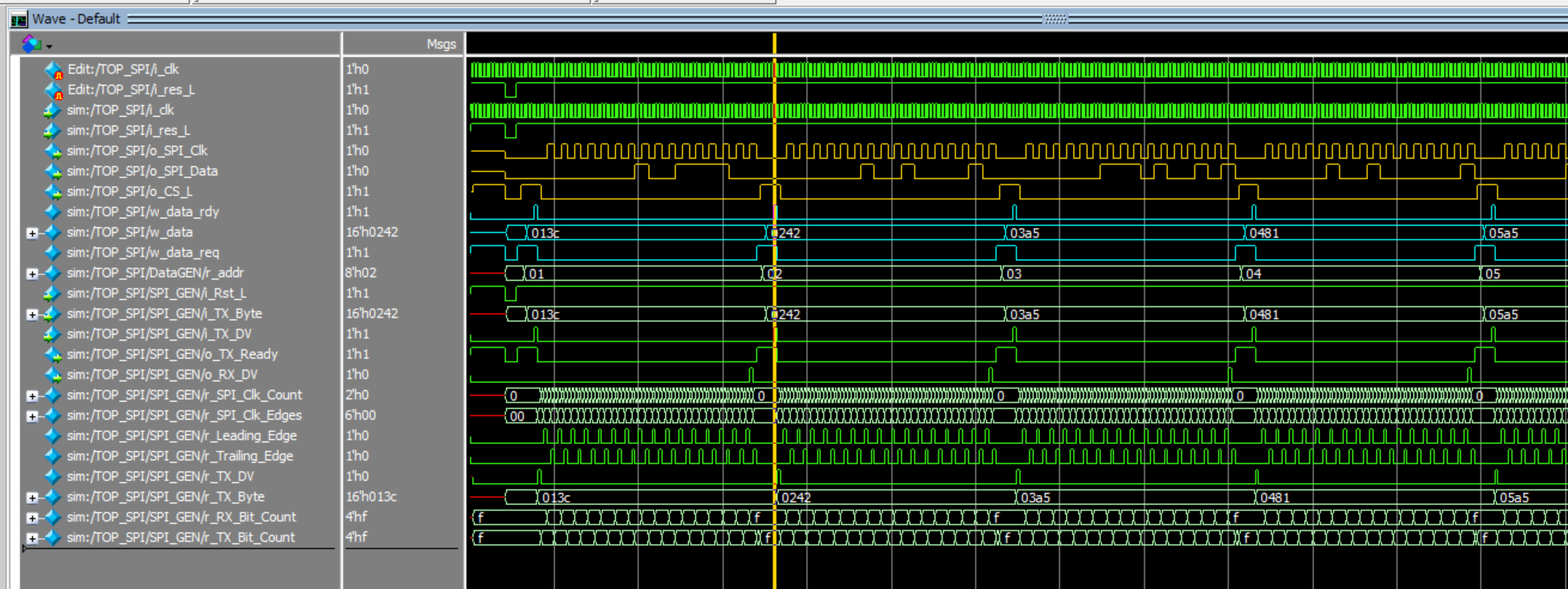
To see the joint operation of the two blocks, it is therefore possible to move on to the simulation of the TOP\_SPI block which brings together the two previously developed and studied blocks in a single system.

### Simulation of TOP\_SPI

With the procedure suggested in the previous point, let's now move on to simulate the "SPI\_LUT" module:

Generate the necessary inputs

* A clock signal
* A negative pulse on the reset



And display the signals considered most significant such as

* The data to be transmitted w\_data
* the various synchronisms of handshaking w\_data\_rdy, w\_data\_req
* The output signals of the SPI protocol o\_SPI\_Clk, o\_SPI\_Data, o\_SPI\_CS
* Also display the adequacy of the data to be transmitted with its relative serial protocol, for example the first data 0x013C = 0000\_0001\_0011\_1100

# Implementation on FPGAs

Using the "SystemBuilder" tool, create the skeleton of a system that uses

* Clock
* Swithes + Buttons
* GPIO1 (Default Mode)

Choose a directory in which to save the project and give it a name (e.g. Tut4SPI)

Copy or move the three VerilogHDL files into the newly generated project directory.

Open the project within Quartus.

Import the three VerilogHDL files

Project > Add/Remove Files in project

Edit the "Project Navigator" Window to Files mode.

Right-click on the file TOP\_Spi 🡪 Create Verilog Instanstation Template File from Current File

The file: TOP\_SPI\_inst.v appears in the Project Directory and can be examined to see how to instantiate the module.

ReModify the "Project Navigator" Window by putting it in Hierarchy mode

Double click on the Top Level Entity (Tut4SPI)

Include the TOP\_SPI\_inst module instantiation in the Verilog file (seen in the previous steps)

Change the connections to reflect the connections with the devices. Especially

* Connect the clock with the CLOCK\_50 signal
* Connect the reset with the KEY signal[0]
* Connect respectively
  + o\_SPI\_Clk with GPIO1GPIO[0]
  + o\_SPI\_Data with GPIO1GPIO[1]
  + o\_SPI\_CS\_L\_sig with GPIO1GPIO[2]

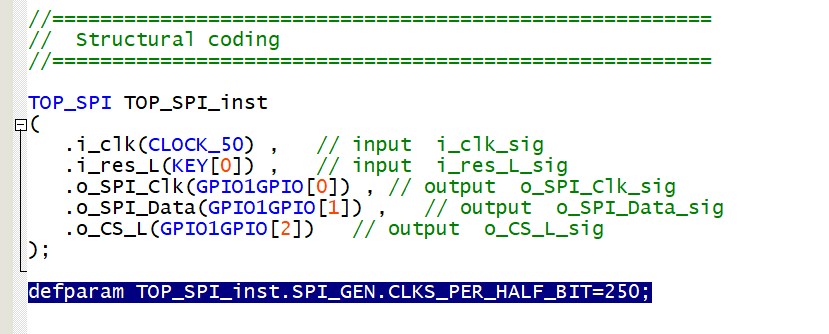
Also change the working frequency of the SPI output: considering that the SPI working frequency goes from about 10KHz to 10MHz you need to set the latter by changing the parameter CLKS\_PER\_HALF\_BIT contained inside the SPI\_master16b module. For example, by bringing the value to 250 there will be 500 primary clock cycles (at 50MHz) for each SPI clock cycle, thus providing an SPI frequency of 100KHz

This can be done in various ways, but the most convenient is by introducing the

defparam TOP\_SPI\_inst. SPI\_GEN. CLKS\_PER\_HALF\_BIT=250;

which overrides the default parameters in hierarchical form for the modules concerned.

In the end, the code to be provided to the Top Level Entity could be the following:



# Links & Downloads

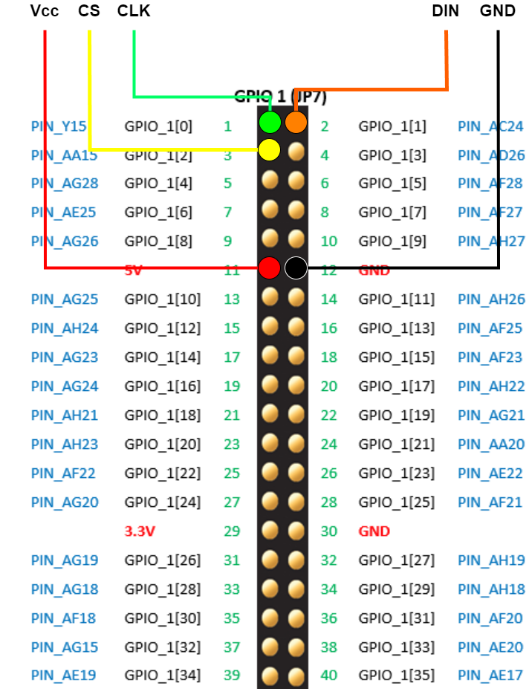
The complete system can then be compiled to the final ".sof" file ready to configure the FPGA. Before that, however, the device must be properly connected.

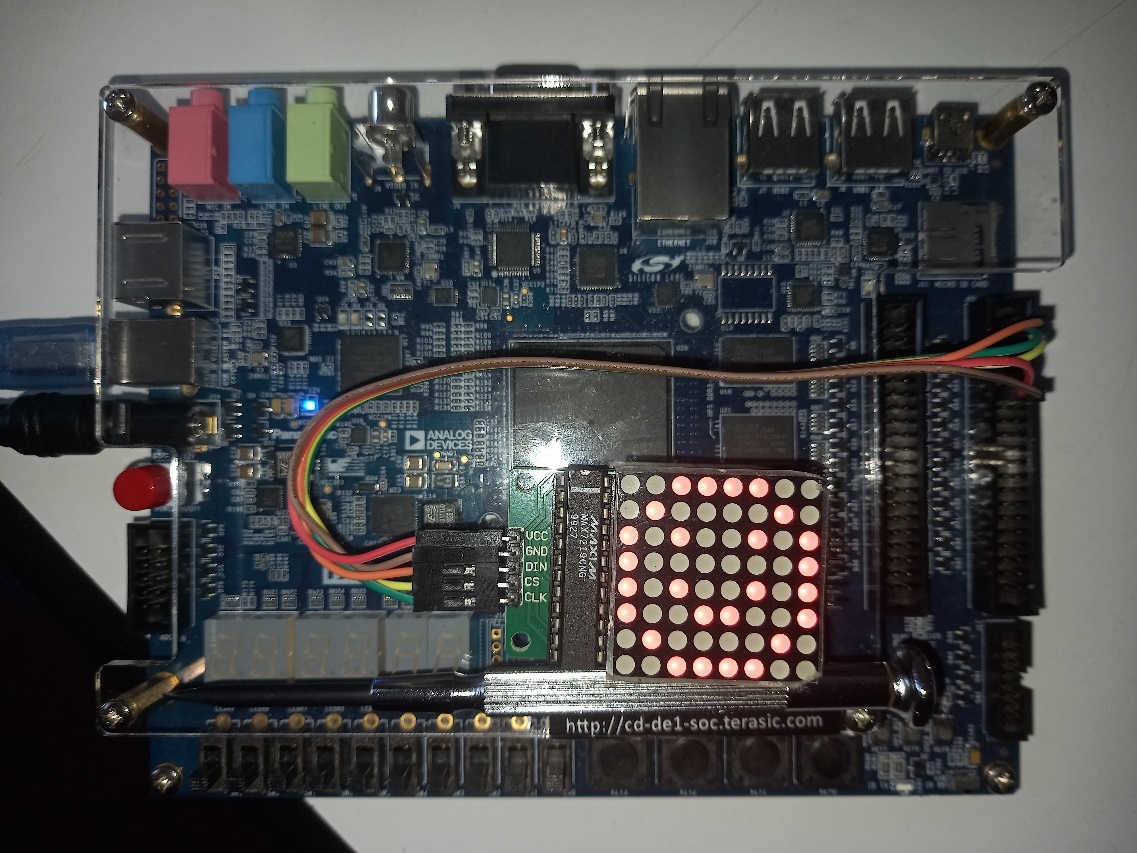
It is controlled through 3 SPI signals (CLK, DIN, CS) and of course is powered by VCC and GND.

The Vdc and GND power supplies can be taken directly from the DE1-SoC board on pins 11 and 12 of the GPIO1 connector respectively, while as arranged in the Verilog code made in the previous step, connect

* CLK with GPIO1GPIO[0]
* DIN with GPIO1GPIO[1]
* CS with GPIO1GPIO[2]

As per the image below





Note: the system configured in this way continuously sends the configuration data of the 15 registers through the SPI protocol, these values are contained in the ROM included in the SPI\_LUT module.

To change the image displayed, please note that:

* Registers between 1 to 8 contain the data to be displayed in each column, respectively
* Register 9 represent the decoding Mode (Normal = 0x00)
* Register 10 control the light intensity (on the 4 least significant bits) between 0x00 and 0x0F
* Register 11 is the number of columns to be scanned during the led visualization
* Reg 12 is used for Shutdown/Normal mode register controlling the least significant bit (Normal=0x01)
* Unused registers 13 and 14
* The 15 Register is used for Test/Normal Mode controlling the least significant bit (Normal=0x00)

# Logic Analizer

A particularly interesting tool available within Quartus is the Signal Tap Logic Analyzer. This system is designed to monitor both internal and, if necessary, external signals in the FPGA, in order to validate their correct format and temporal behavior.

In short, once the signals to be monitored are selected, the tool stores their data internally in the FPGA and displays their evolution graphically through an interface window. To achieve this, however, the system must define a sampling clock and, most importantly, reserve suitable internal FPGA resources both to store the sampled signals and to transfer them to the interface tool. These resources can significantly impact the implementation of the final circuit; therefore, careful use of this tool is strongly recommended.not to exceed the memory depth or sampling rate to use this tool only during debugging and to delete it from the final system when it has been verified that it has worked correctly.

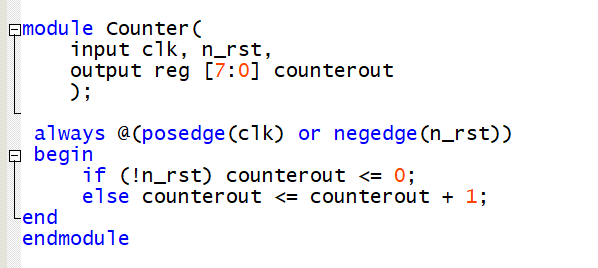
Let's start by taking note of the resources currently used for the implementation of the system

* analyzing both the various "report logs"
* using the "Chip Planner" tool

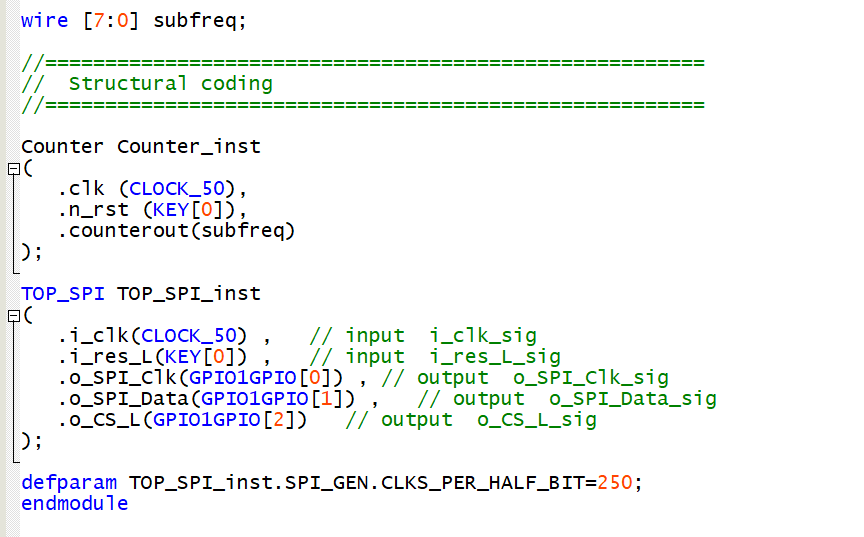
(in particular how many ALMs, Memories and registers were used)

First of all, to avoid having to sample the signals of interest too densely (i.e. at 50MHz), create a "frequency divider" or a cyclic binary counter which, by receiving the 50MHz system clock as input, provides different signals in outputs, each with half the frequency compared to the previous one, so at the fifth output of the counter we will find a signal at about 1.5 MHz useful for sampling the SPI signals that we have chosen to have a frequency of about 100 KHz.

Therefore, modify the code by inserting a new module, either within one of the files already part of the project or through a new file



and then instantiating it in the Top Level Entity by creating appropriate signals (subfreq) and appropriately connecting all the signals.

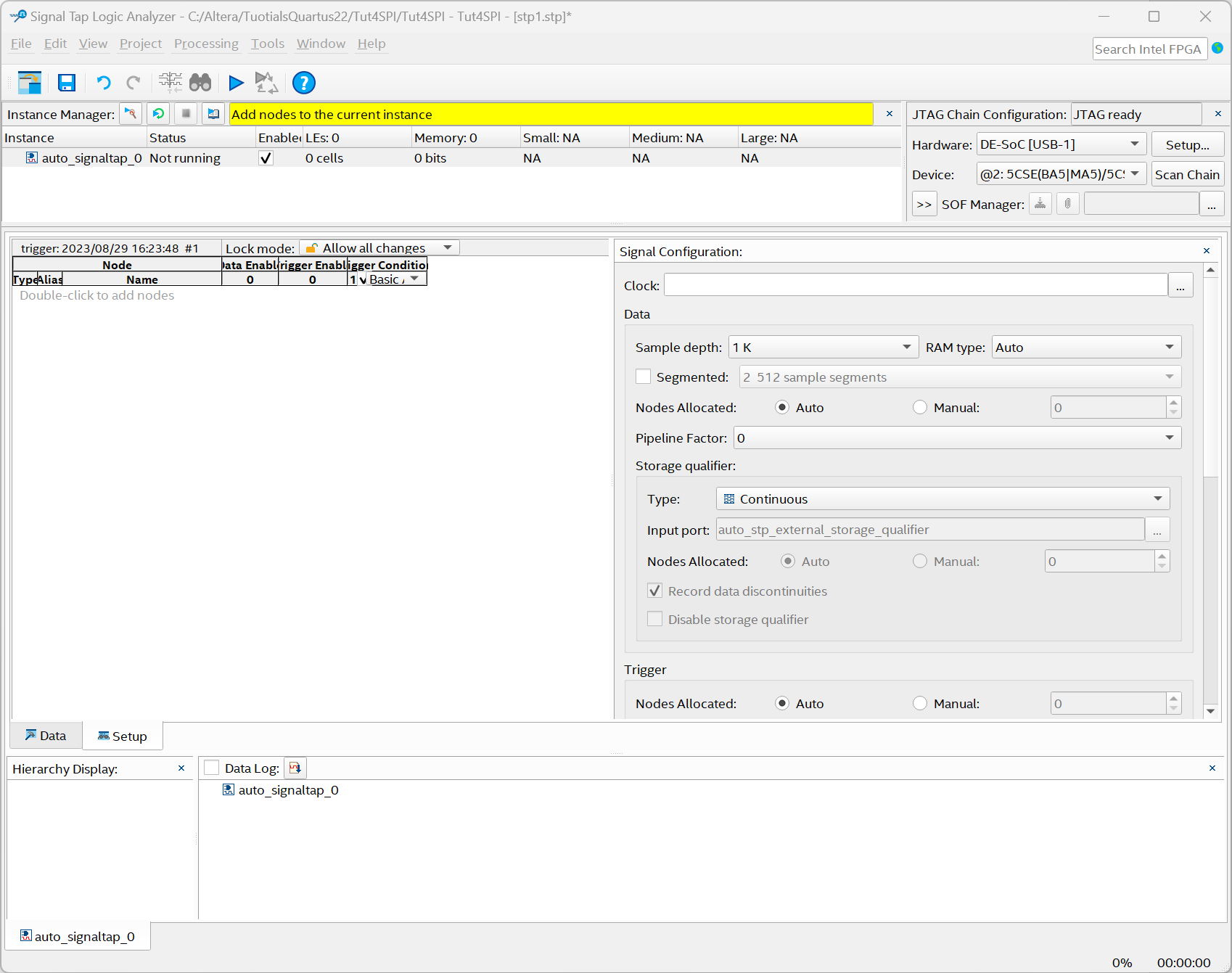


> Processig > Start > Start Analysis and Syntesis (Ctrl-K)

Then open the "Signal Tap Logic Analyzer" tool:

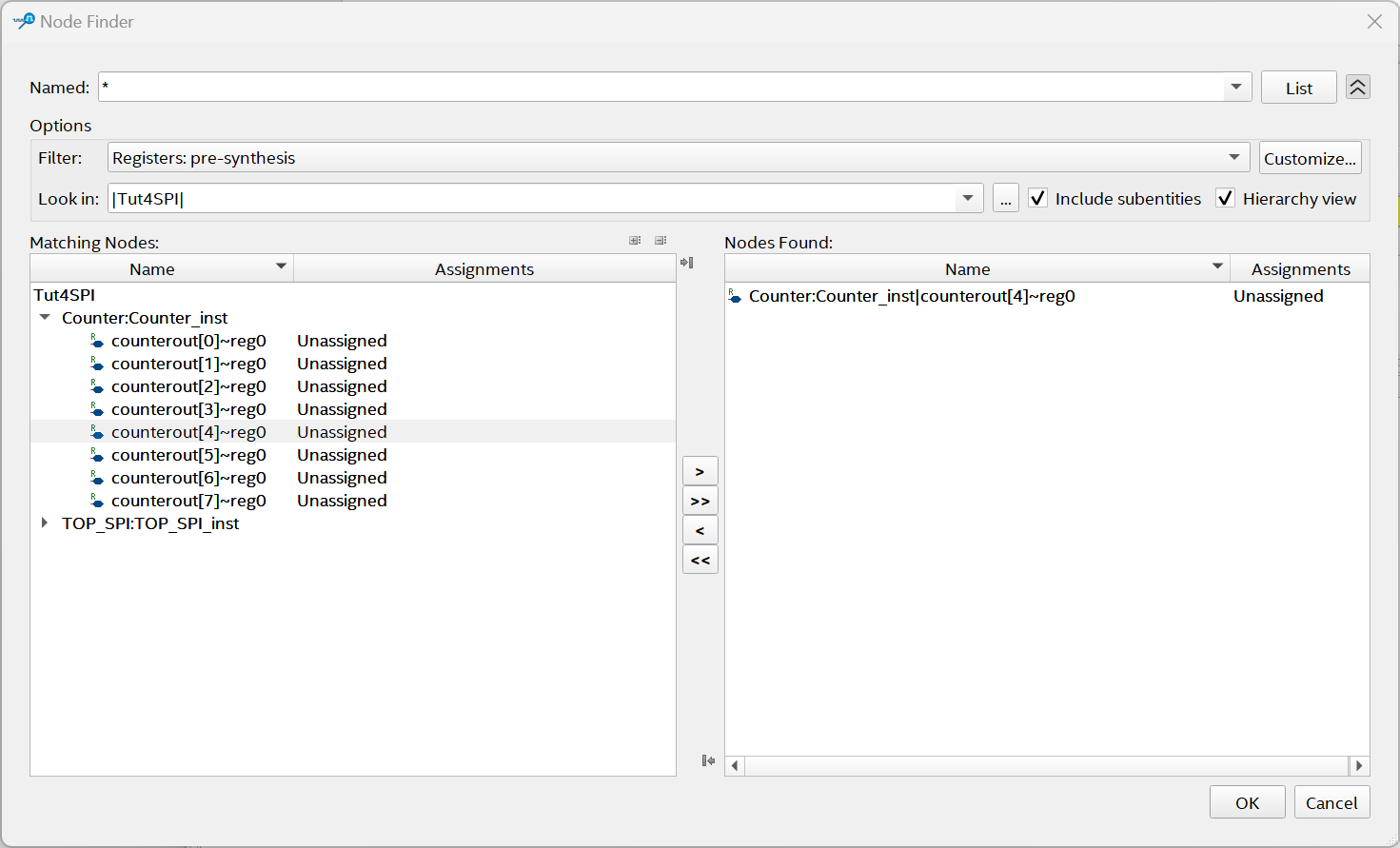
> Tools > Signal Tap Logic Analyzer

In the window that initially opens, define which signal to use as a clock (i.e. to sample the signals).



In the "Signal Configuration" tab, click on the "..." next to the clock cell, a new window will open.

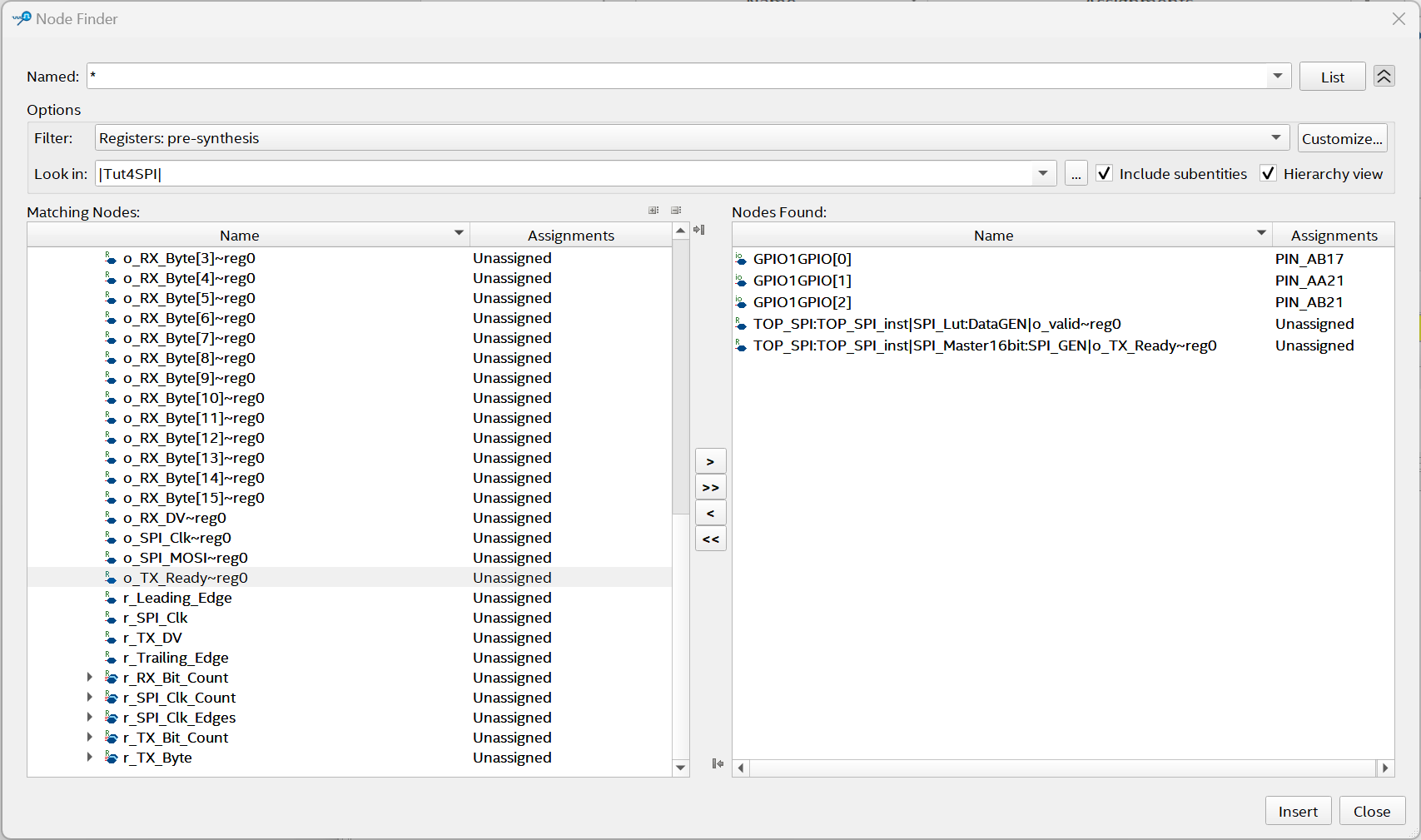
By setting the "Filter" parameter in "Registes:pre-synthesis" mode and then clicking on <List> a series of signals will appear that can be chosen as a sampling clock.



For example, choose the output[4] of the counter

<OK>

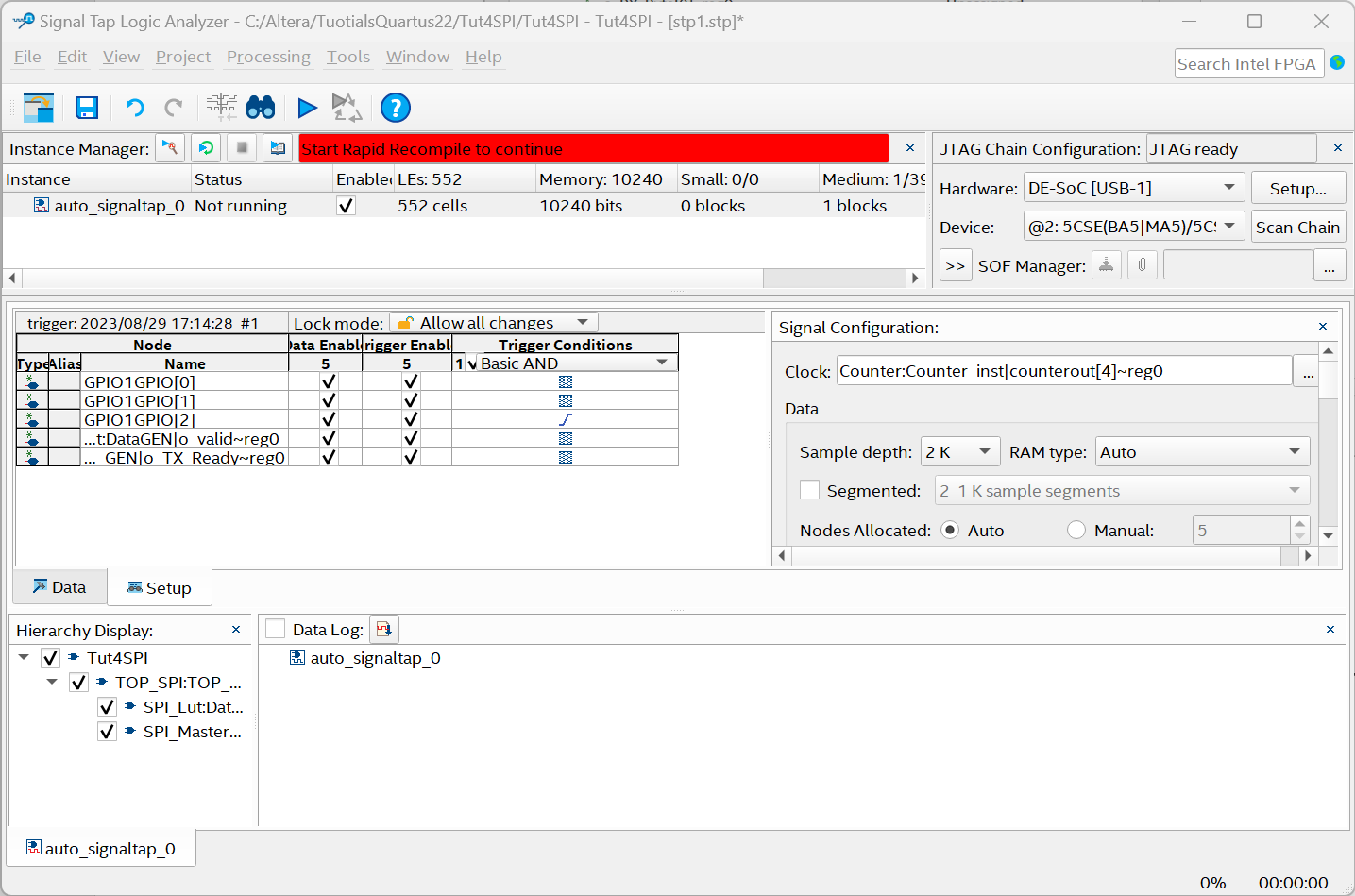
Complete the previous screen by fixing the memory depth (e.g. 2K) and which signals you want to analyze (e.g. the three SPI outputs and the internal handshaking parameters between the two blocks) by double-clicking in the left window and using the same mechanism seen above to filter and identify the signals of interest.

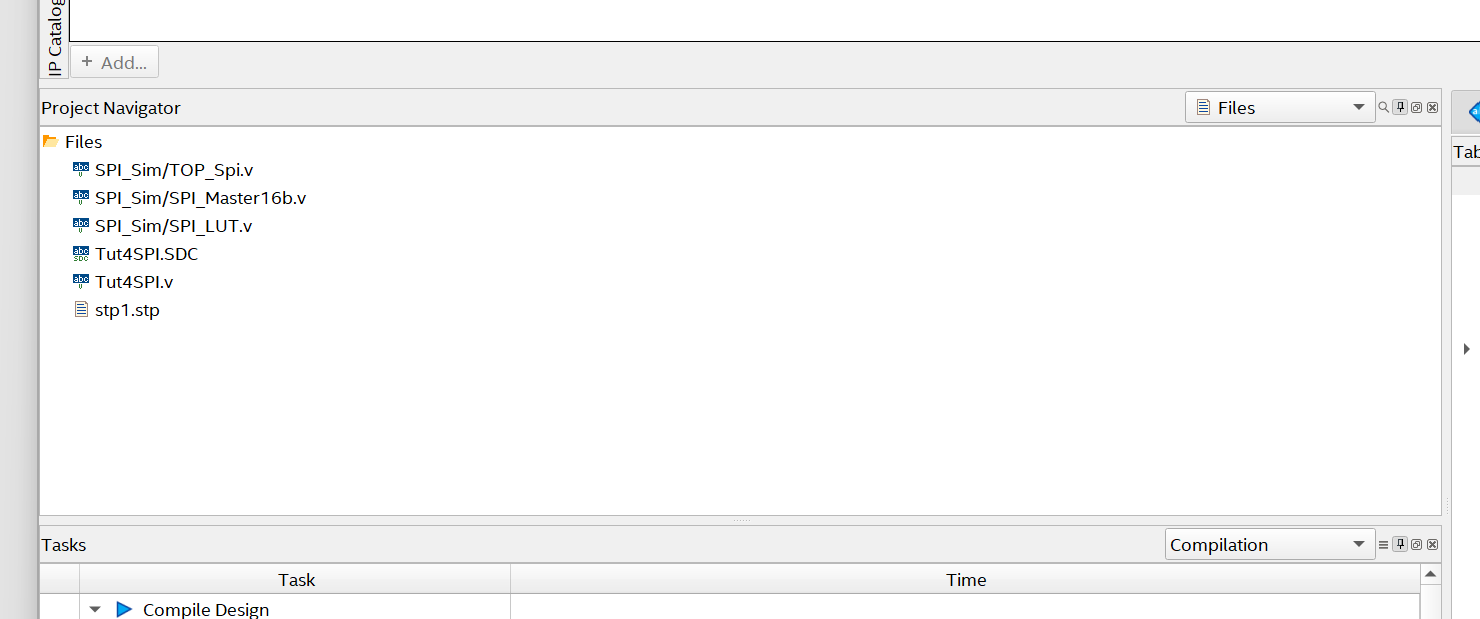


<Insert>

In the next window, right-clicking on the "Trigger condition" cell of the GPIO1GPIO[2] signal connected to the CS of the SPI signal sets this condition to the Raising Edge state.

Save the file (automatically suggested name stp1.stp) and when you ask if you want to include it in the Quartus project, answer in the affirmative way (Alternatively, manually include the file you just saved among the project files)



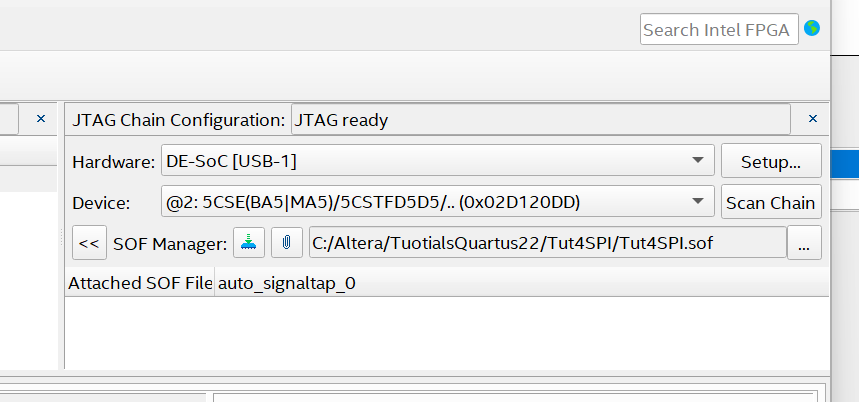


* Complete the project (The operation can be carried out directly within the "Signal Tap Analyzer" tab).

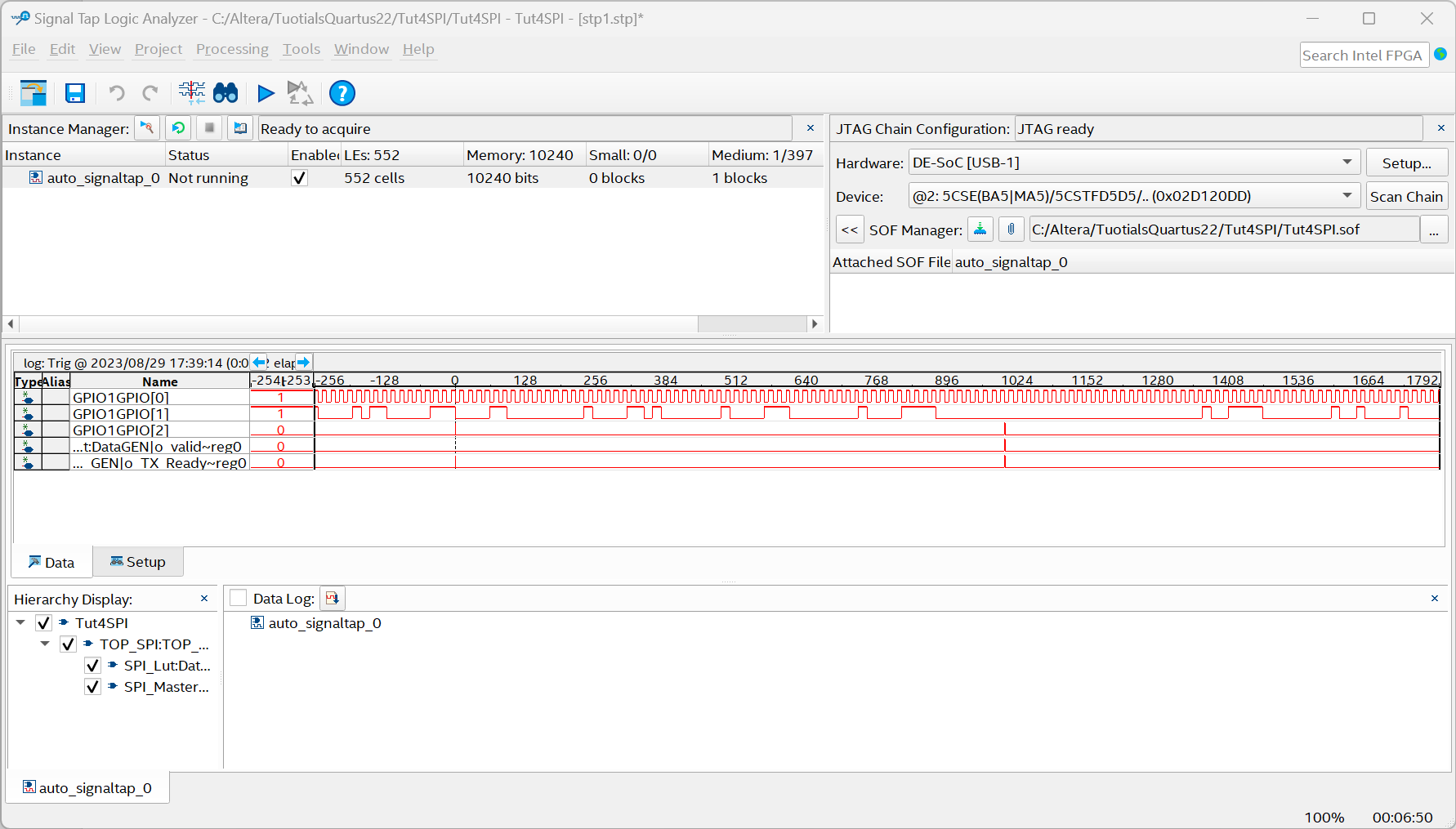
Processing > Start Compilation

* Re-download to the DE1-SoC board

(this part can also be executed directly from within the "Signal Tap Analyzer" window) by configuring the JTAG chain and the configuration file.



At this point (possibly re-opening the Signal Tap Analyzer Tool) if you have closed it by going to the Data tab and using the shortcuts F5 for a point analysis and F6 for a continuous analysis, the data physically read on the FPGA will be available synchronized as required on the CS rising edge



It should be noted in particular, however, that the two hand-shaking signals are sometimes sampled, and sometimes they are not detected, moreover even their temporal succession is not well defined. This is due to the fact that the sampling frequency chosen is too low to have a good sampling, on the other hand increasing the frequency would have decreased the number of samples.

As a last step, it is suggested to verify the resources used for the implementation of the system

* analyzing both the various "log reports" (in particular looking at how many ALMs, memories and logs were used)
* using the "Chip Planner" tool

Comparing the results with those obtained previously, it will be noted how the resources used have increased considerably and that the system use blocks of BRAM memory to store and evaluate continuously the analyzed data before sending them to the visualization tool through the JTAG protocol.

# Memory Editor

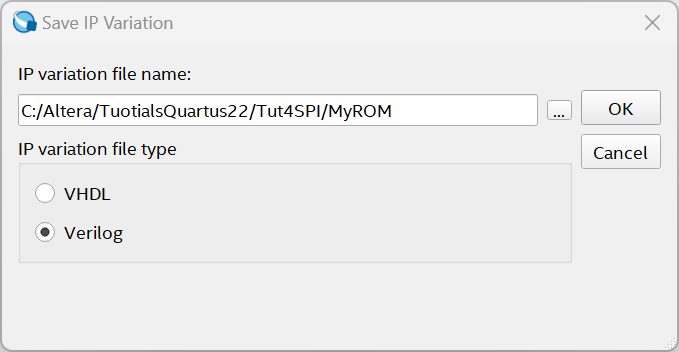
The project developed so far generates a repetitive sequence of SPI signals based on the contents of a memory. Whenever a change to the transmitted code was required, the only option was to modify the memory contents and recompile the entire project.

Quartus, however, provides a tool called In-System Memory Content Editor, which allows interaction with memories via the JTAG protocol, enabling read and write operations. Since this tool effectively converts single-port memories into dual-port memories—using one of the ports to update the contents—it requires specific types of memories designed for this purpose. The goal, therefore, is to modify the design by integrating a general-purpose memory, which can then be updated through this tool to dynamically alter the transmitted signal.

First of all, access the IP Catalog window (Alt-7) where it is not already open and search, perhaps using filters, for the primitive ROM: -1PORT



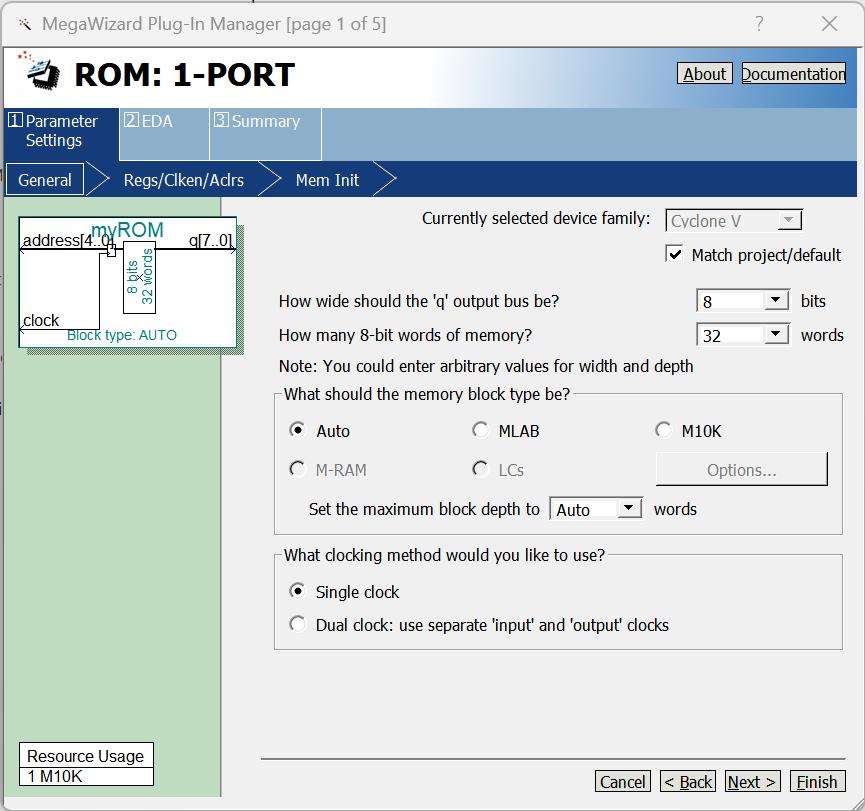
By double-clicking the generation window opens:



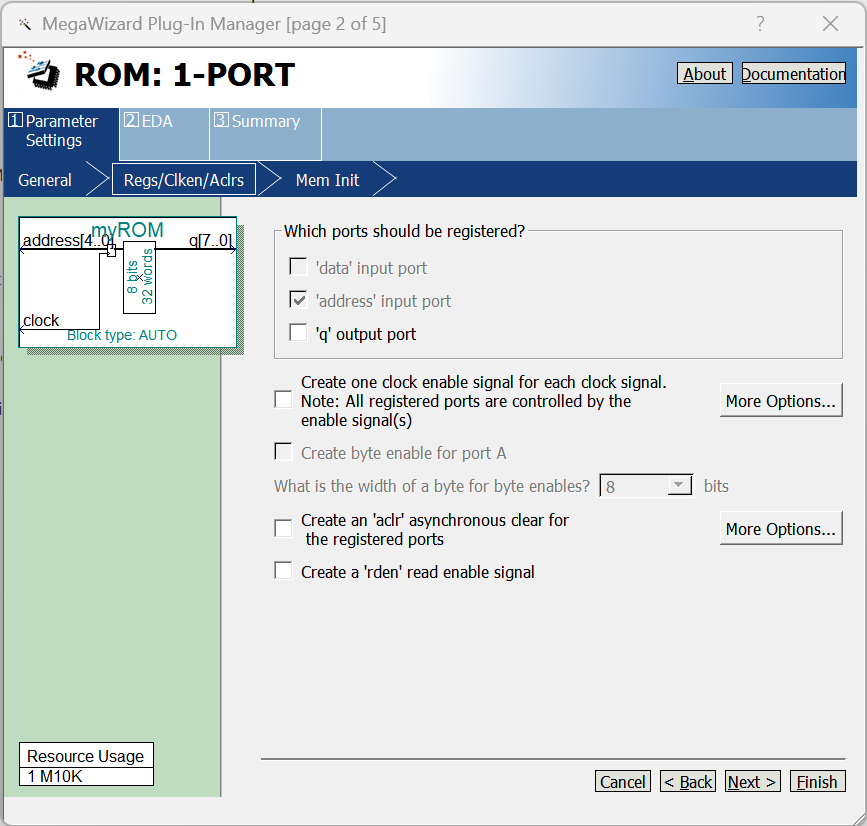
Provide an appropriate name and choose to generate the Verilog code

>OK

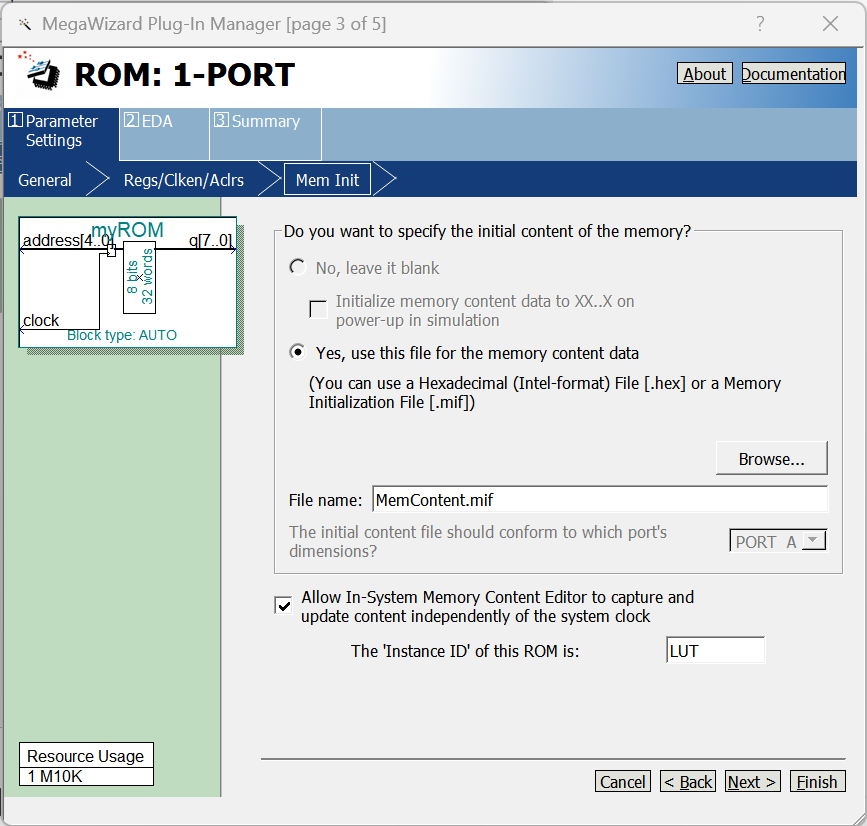
In the configuration windows choose the size of the memory, in this case 32 words of 8 bits each (it is not possible to use memories with less than 32 words, so we will use only the first 16)



Configure the output to be register-free



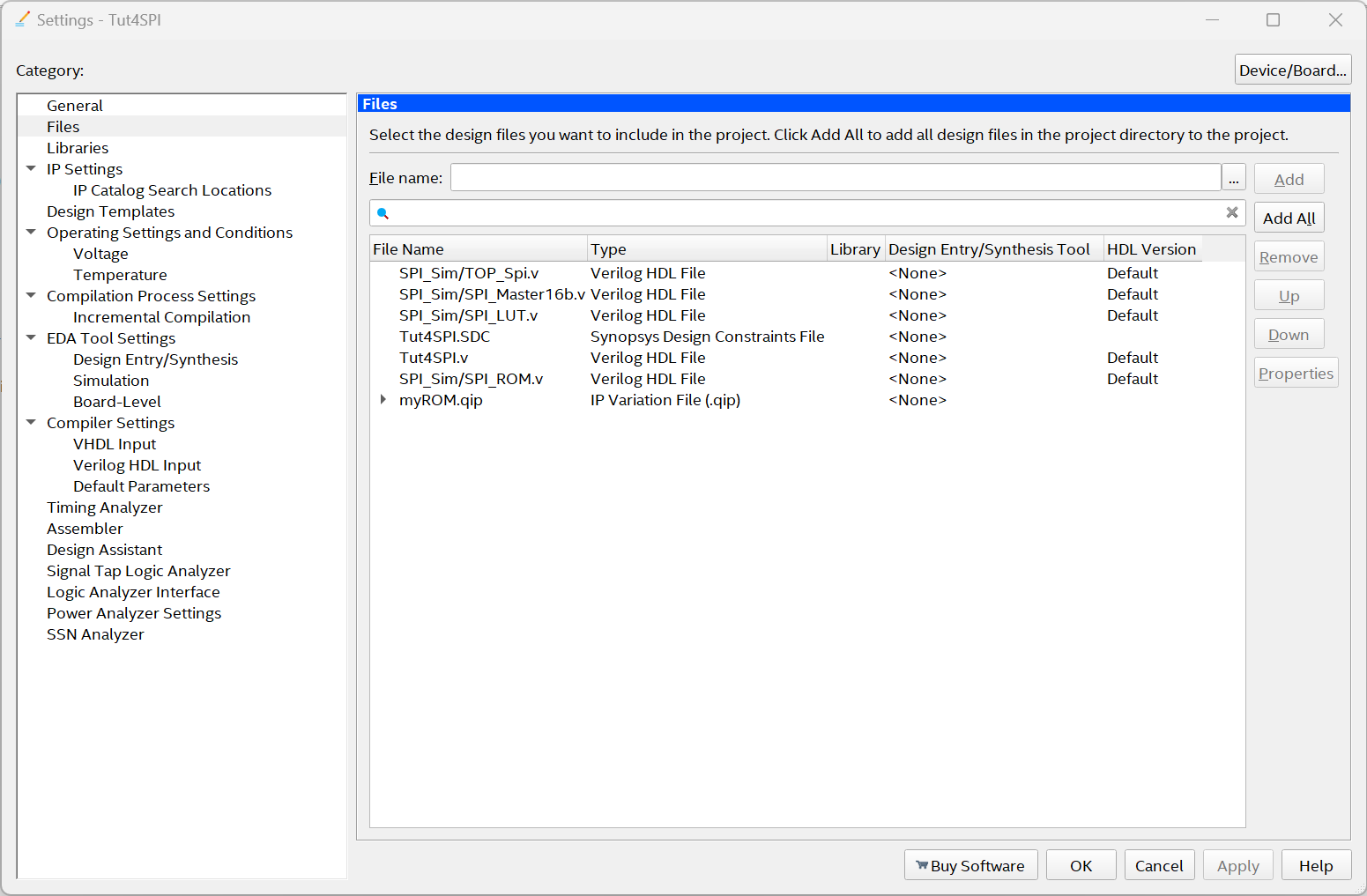
Configure the memory to be accessible to the system "In-System Memory Control Editor" by providing a 4-character mnemonic name and provide a memory configuration file (An example is available on the course page).



> Finish

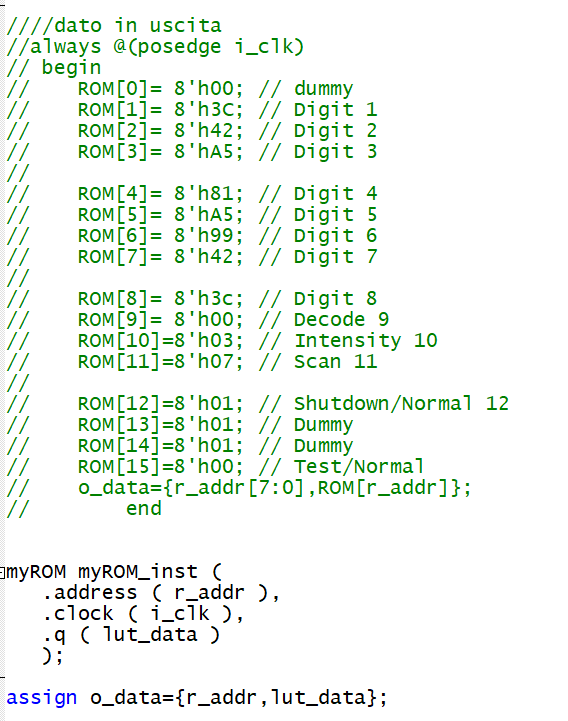
If necessary, it may also be useful to generate an example of "instantiation" of the newly generated element.

Make sure that the .qip file you just generated belongs to the overall project in which you want to include it.



We must now move on to modify the design to replace the LUT created as a combinatorial circuit with the newly generated ROM.

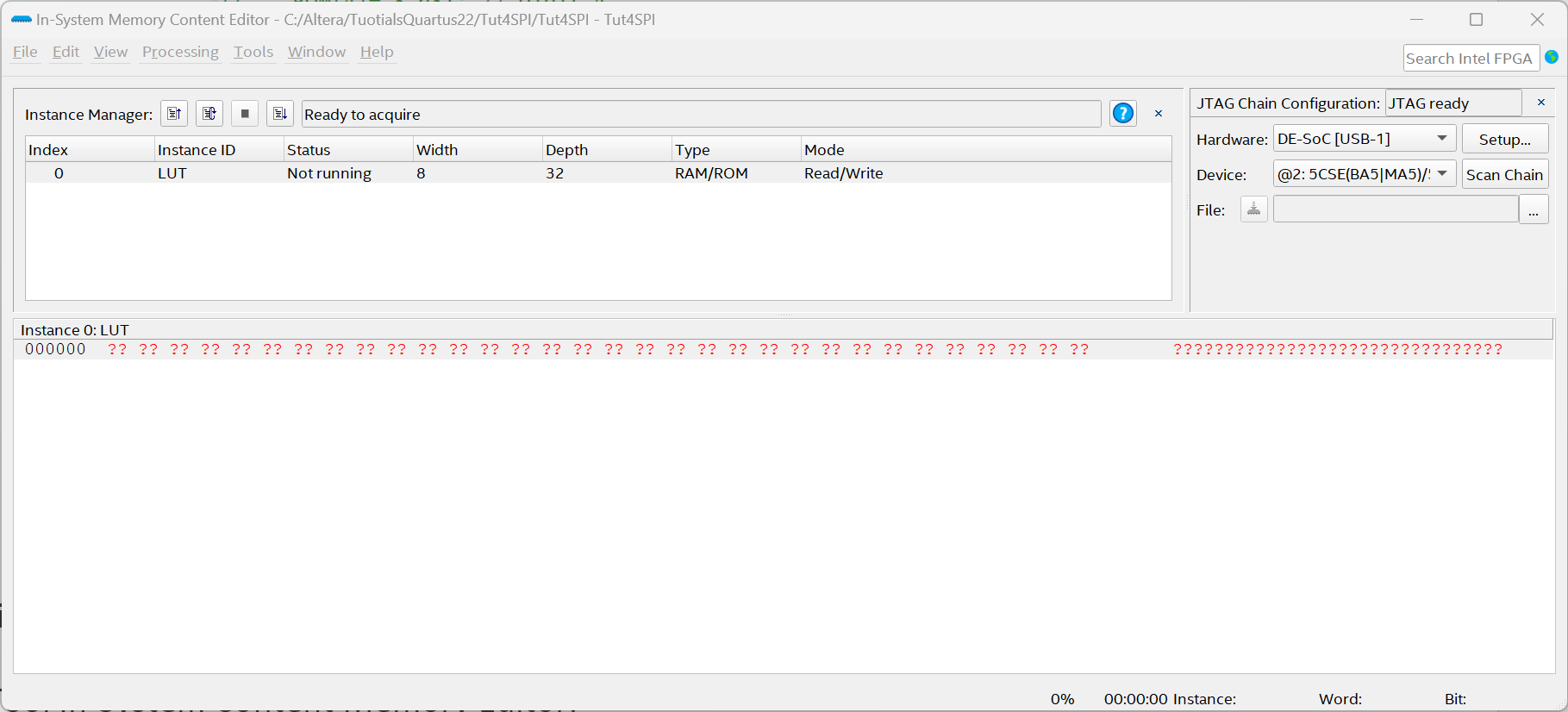
The Verilog file "SPI\_LUT" for the part relating to the data memory by instantiating the ROM just generated and concatenating the address with the output data, not before having changed the type of the variable o\_data from "reg" to "net" and having declared the new intermediate signal "lut\_data".



Then recompile the entire project and use it to configure the Board.

You can now open the In-System Content Memory Editor Tool:

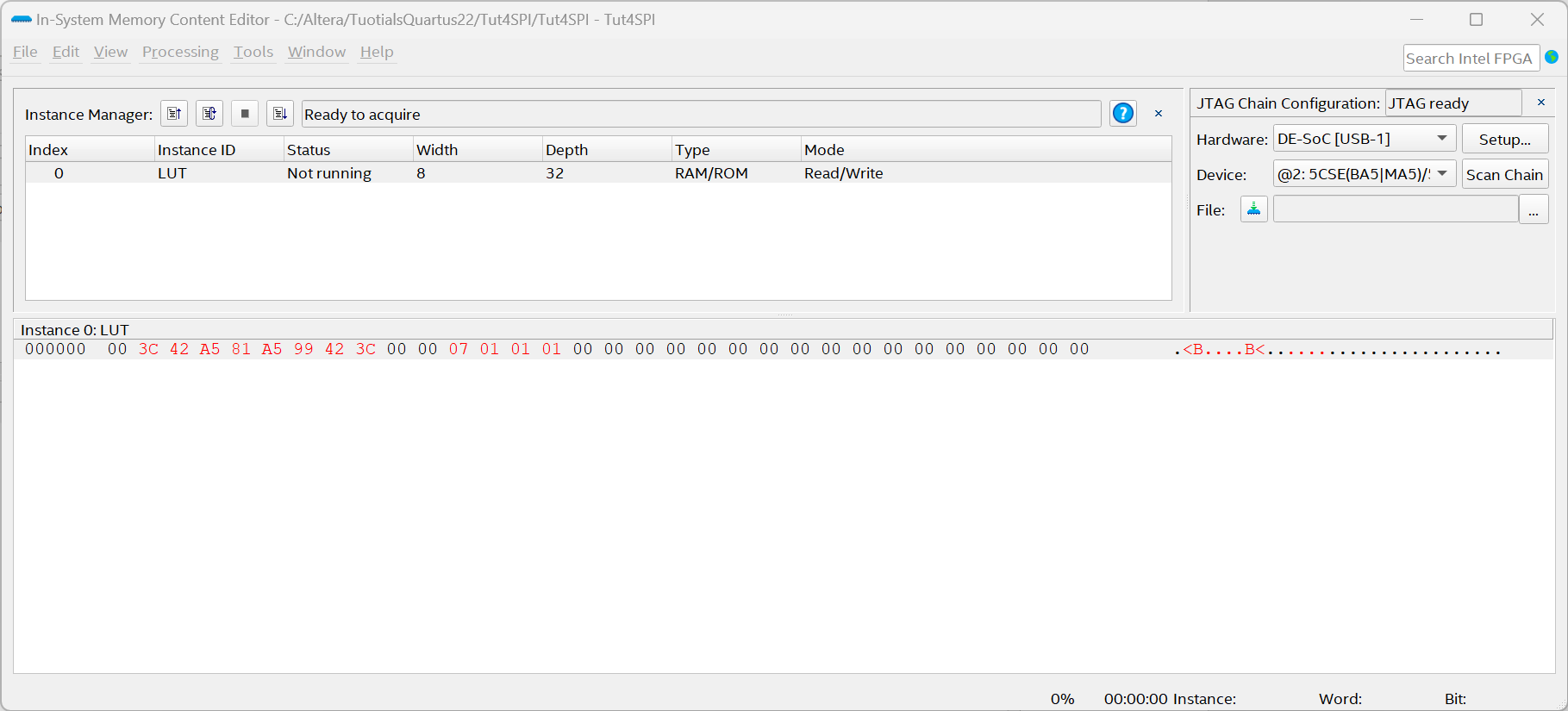
> Tool > In-System Content Memory Editor



Click on the appropriate "Instance ID" in this case only one and choose

> Processing > Read Data From In-System Memory (F5)

And the window is populated with data obtained from memory



Or modify the data (Writing congruous data) and activate the writing of the data:

> Processing > Write Data To In-System Memory (F7)

By doing so, you can try to alter

- the data displayed (in the [1:8] positions) - the brightness of the display [10]- in the number of columns on which to write the data [11]- the Shutdown/Normal mode [12]- the Normal/Test mode [14]