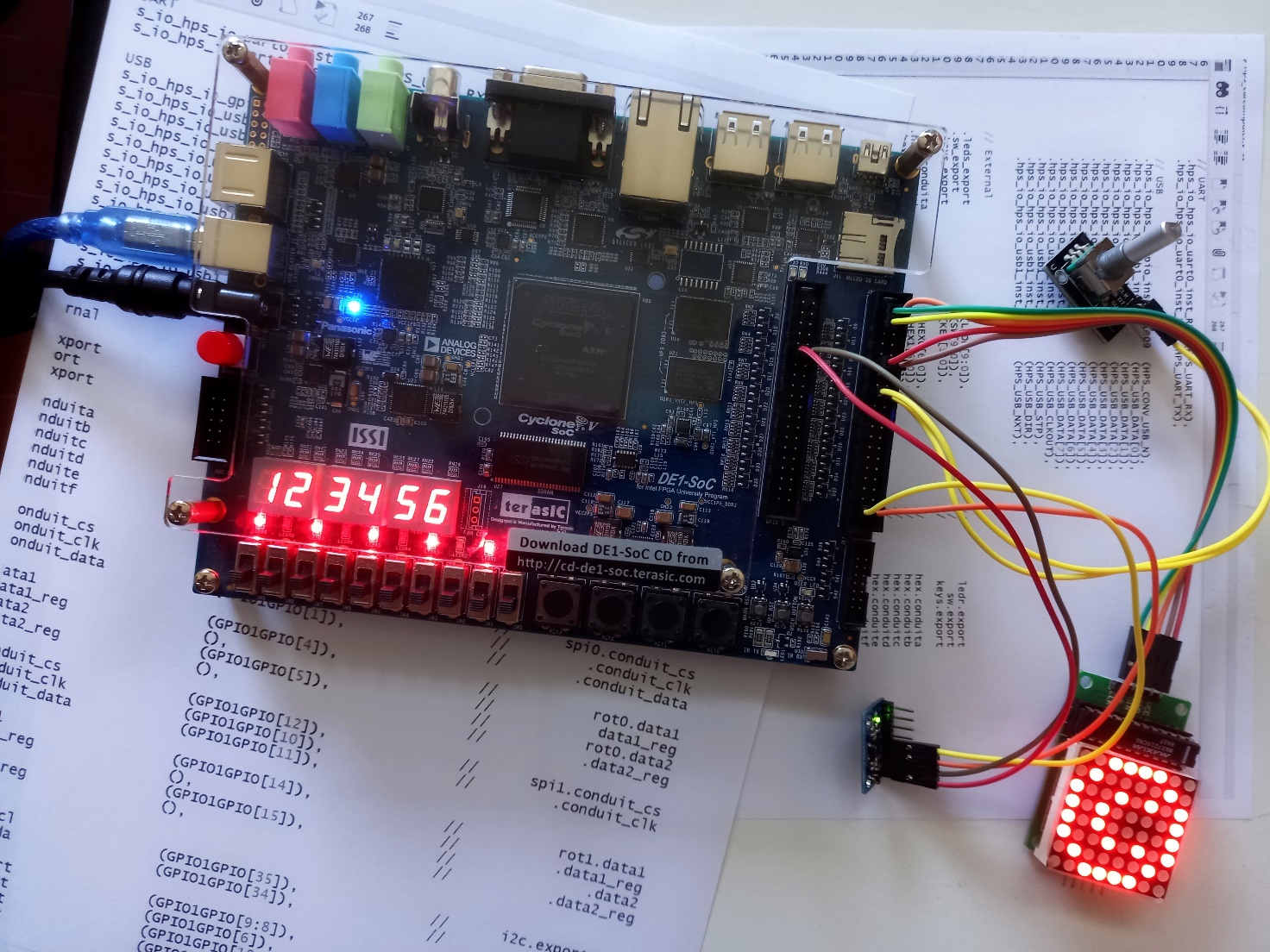
Electronic Systems Design

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

Tutorial 6



**Development of a complete computing system of interfaces based on ARM.**  
**Hardware** used: Terasic DE1-SoC Board  
**Software** used: Quartus 22.1

Tutorial 6

Creation of a computing system equipped with both standard and custom peripherals, based on an ARM processor

Description: In this tutorial you will create a system consisting of an ARM processor and various peripherals both custom and made by third parties.

Purpose: The aim is to see how an entire computing system can be created within a single FPGA and how this system can be used to interact with the outside world through peripherals already developed by third parties or even by developing the most suitable peripheral to create a certain interface to the outside.

Expected learning:

* In-depth study of the Quartus "Platform Designer" tool to create a processor together with its interface system
* Configuring the processor and third-party peripherals
* Creation of "ad hoc" peripherals
* Processor programming and code execution in a Codesign "Hardware-Software" process

# Premise

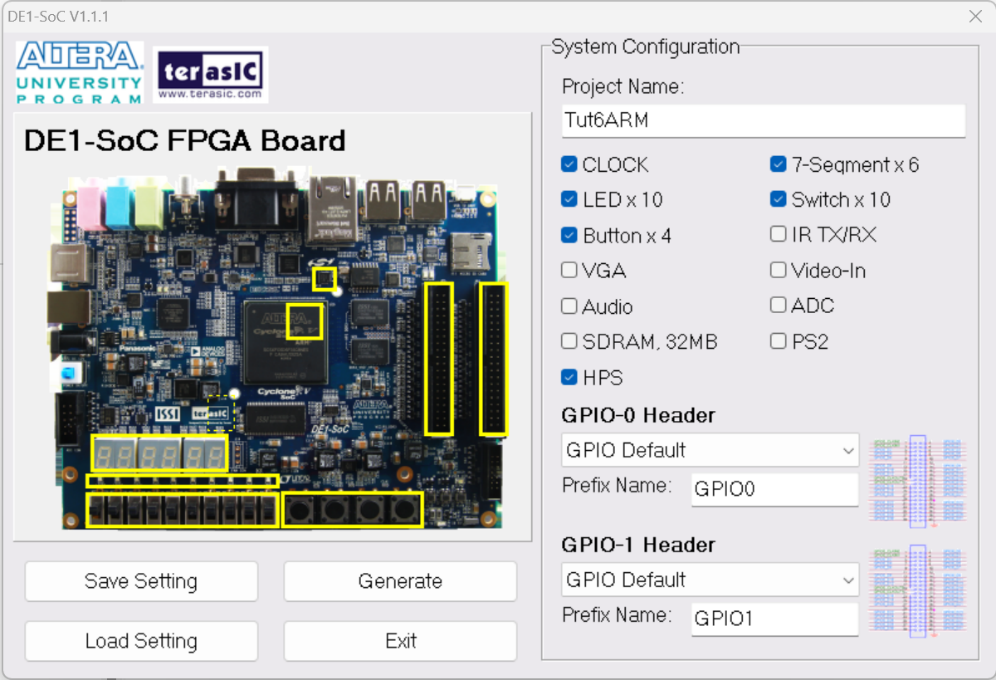
A modern FPGA contains both resources to create dedicated hardware, but also specific resources that create computing, control and interface systems to the outside world according to the most common protocols in order to create what is commonly called System On Chip (SoC). In particular, the FPGA mounted on the DE1-SoC has already integrated a dual Core ARM processor in hardware form as well as EMACs, USB Controllers, I2C Controllers, UARTs, CAN Controllers, SPI Master Controllers, SPI Slave Controllers, GPIO Interfaces etc. etc.

In this tutorial we will see how to interact with these elements to create an ad hoc calculation and interface system, choosing in particular which processing is most appropriate to be carried out in hardware and which in software.

# Realization of the architecture

Start by creating the "skeleton" of a system using the "System Builder" tool that involves the use of

* HPS (Hardware Processor System)
* Leds
* 7-segment display
* Swithes
* Keys
* Clock
* SystemID
* GPIO0 and GPIO1 (in default mode)



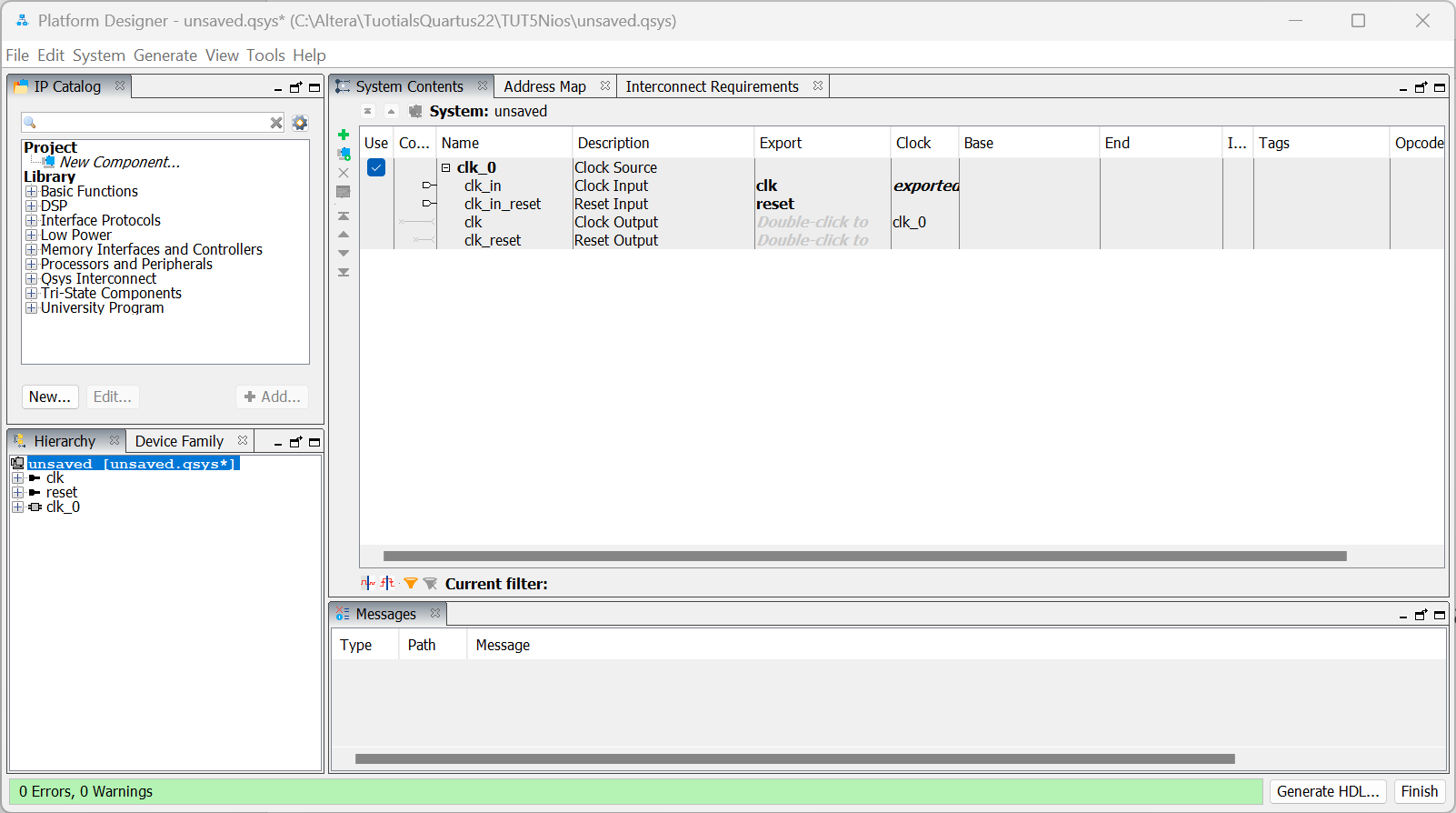
Not all of these peripherals will be used at first, but their presence will be useful later without having to reconsider the project from the beginning.

Open the project thus generated within Quartus.

Inside Quartus:

Tools > Platform Design

This opens up a system that allows you to configure the architecture based on blocks already developed by third parties.

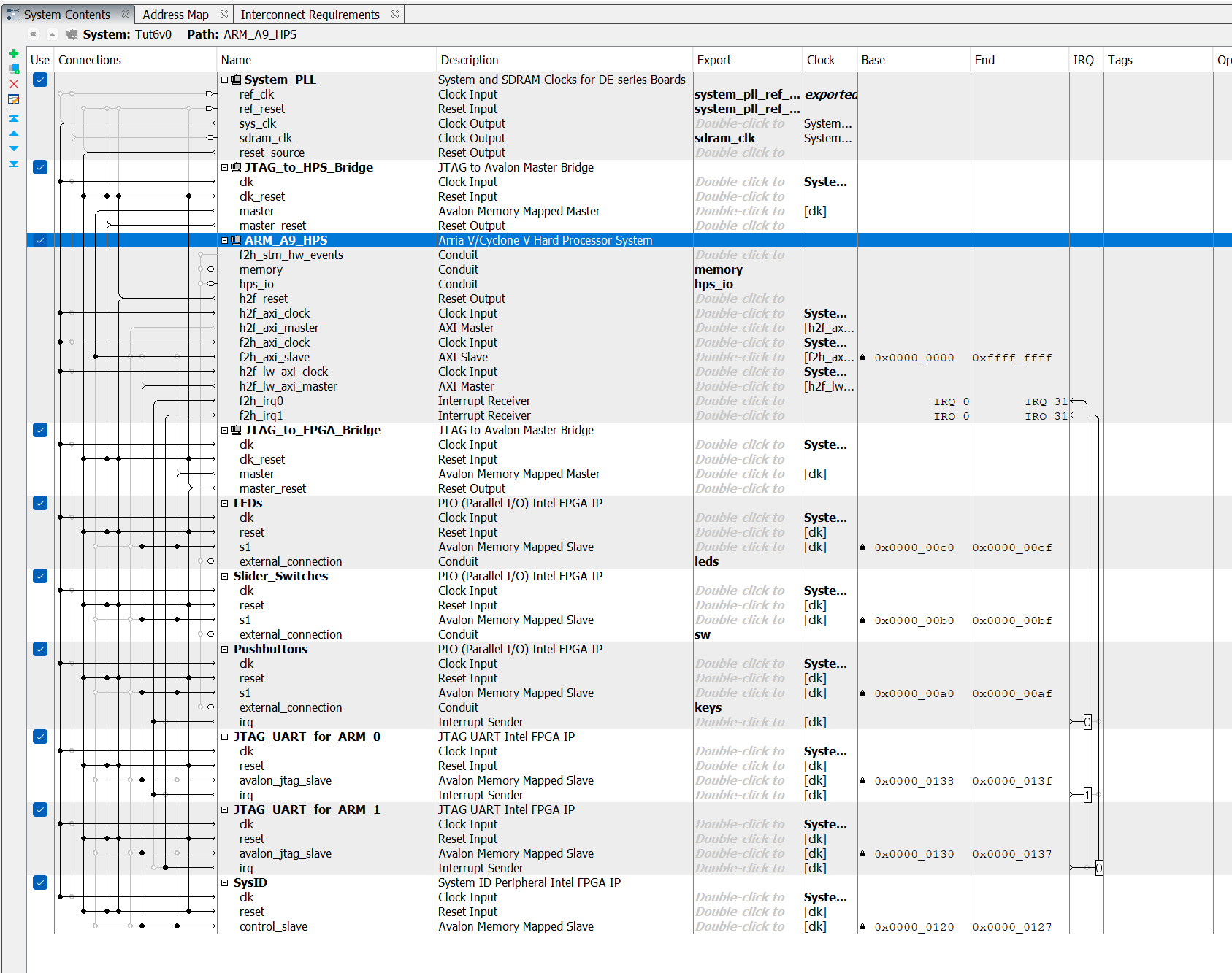


A first block is already present and is a block useful for managing the clock and reset, but this will be replaced with another one that can also be used for projects that should for example use the external memory (SDRAM) present on the DE1-SoC

Therefore, include the following blocks

* **System and SDRAM clock For DE1-series Boards** (the soft processor)
* **JTAG\_to\_HPS\_Bridge** (the interface that allows the JTAG to access HPS resources)
* **ARM\_A9\_HPS** (the ARM processor)
* **JTAG\_to\_FPGA\_Bridge** (the interface that allows the JTAG to access the FPGA's resources)
* Two **JTAG UART INTEL FPGA IP** (The STDIO serial interface to communicate with the processor) to interface with the two ARM processors
* Three suitably sized Intel FPGA IP (Parallel I/O) PIO-type **interfaces (to interface with Switches, LEDs and KEYs, respectively)**
* **System ID Peripheral Intel FPGA IP** (System Identifier – SysID)

The window with the architecture should look more or less like this:



Rename the various blocks appropriately and configure the connections as follows:

* The clock signal output from the "System PLL" is connected to all the blocks and in particular to the 3 interfaces of the ARM
* The Reset signal output from the clock generator is connected to all blocks

System > Create Global Reset Network

* The **ARM h2f\_lw\_axi\_master** line reaches all blocks used as a peripheral   
  (i.e. the ARM reads/writes data from all blocks through its lightweight 32-bit interface)
* Line **h2f\_axi\_master** remains disconnected
* the f2h\_axi\_slave line is connected with the **master port of JTAG\_to\_HPS\_bridge** (this will allow the debugging tool to be able to interact with the processor)
* Memory **and** hps\_io **signals**  are exported to the FPGA
* The **f2h\_stm\_hw\_event** signal (if present) can remain disconnected
* The **f2h\_irq0** and **f2h\_irq1** signals are connected to the two interfaces respectively **JTAG\_UART\_For\_ARM**
* Check that all interfaces (Led,Switches,Button,SysID, Bridge-x2, UART-x2) receive
  + The clock from System PLL
  + The Reset from:
    - SystemPLL
    - JTAG2HPS Bridge
    - JTAG2FPGA Bridge
    - ARM (from port h2f\_reset)
* Check that all interfaces to the outside (Led,Switches,Button,SysID) receive
  + on the slave port (**S1**) the signals coming from
    - JTAG2FPGA Bridge
    - ARM (from port h2f\_lw\_axi\_master)
  + Export all interface signals (conduit) to the outside and assign a name if necessary

Now let's define (manually or automatically) the appropriate memory addresses for the various blocks. The automatic procedure can be useful in this sense

System > Assign Base Address

And define (manually or automatically) the "interrupt numbers"

System > Assign Interrupt Numbers

The architecture defined so far should resemble the one shown in the figure.

Now you have to carefully configure all the various blocks and in particular the ARM Processor which reflects in its structure the use of the interfaces already provided on the De1\_Soc card.

In particular, the

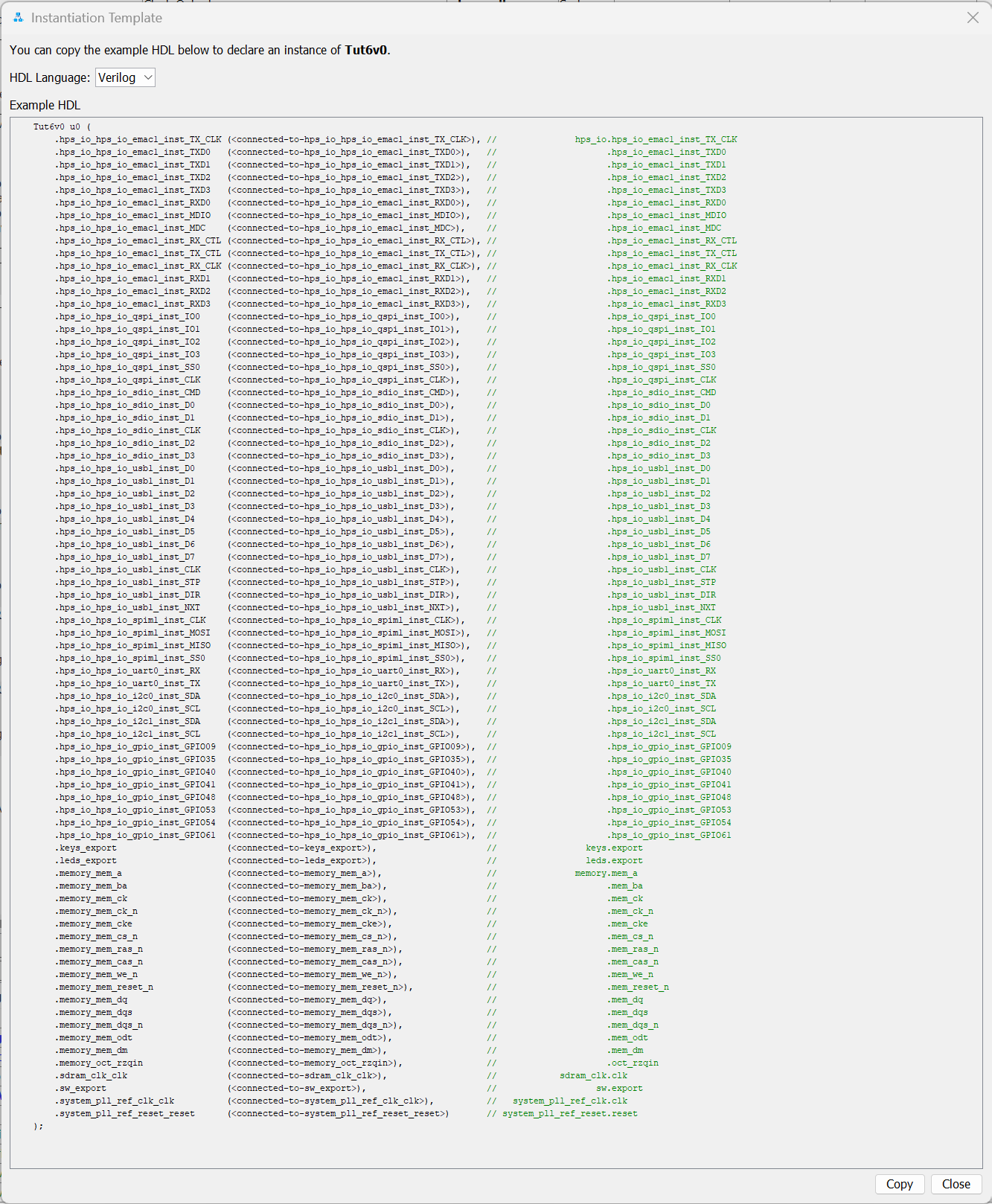
* **Fpga Interface** is used to configure all the various interfaces and interrupts deriving from the various peripherals that make up the HPS
* **Peripherals Pins** define which HPS peripherals are activated and which default pinout setting to refer to. For example: since in the De1-SoC board at the HPS55 and HPS56 pins there is already an accelerometer installed whose I2C port belongs precisely to those pins, obviously if you want to use it you will have to ensure that the I2C Controller I2C0 is assigned to the HPS I/O set0 I/O set. The same applies, for example, to controllers to access the SD memory or the USB interface or even the Ethernet port, etc. etc. Another example are the HPS53 and HPS54 pins that have been connected respectively to an LED and a button accessible only through HPS and therefore if you want to use them they must be exported through the appropriate connection table.
* **HPS Clocks (in turn divided into Input clocks and Output clocks)** defines the frequencies of the clocks pertaining to the various peripherals and eventually exports them
* **SDRAM (in turn divided into subfolders)** defines all the parameters for suitable communication to the SDRAM.

Since the complete setting of all these parameters, which must reflect the physical characteristics of the circuit on which the FPGA is mounted, can be quite expensive, it is advisable to draw inspiration from the file available on the Moodle page of the course, or from the software material provided together with the De1-SoC sheet.

Once the processor and peripheral architecture is complete, take note of the addresses to which the various peripherals are mapped.

You might also want to see an example of how to instantiate the component:

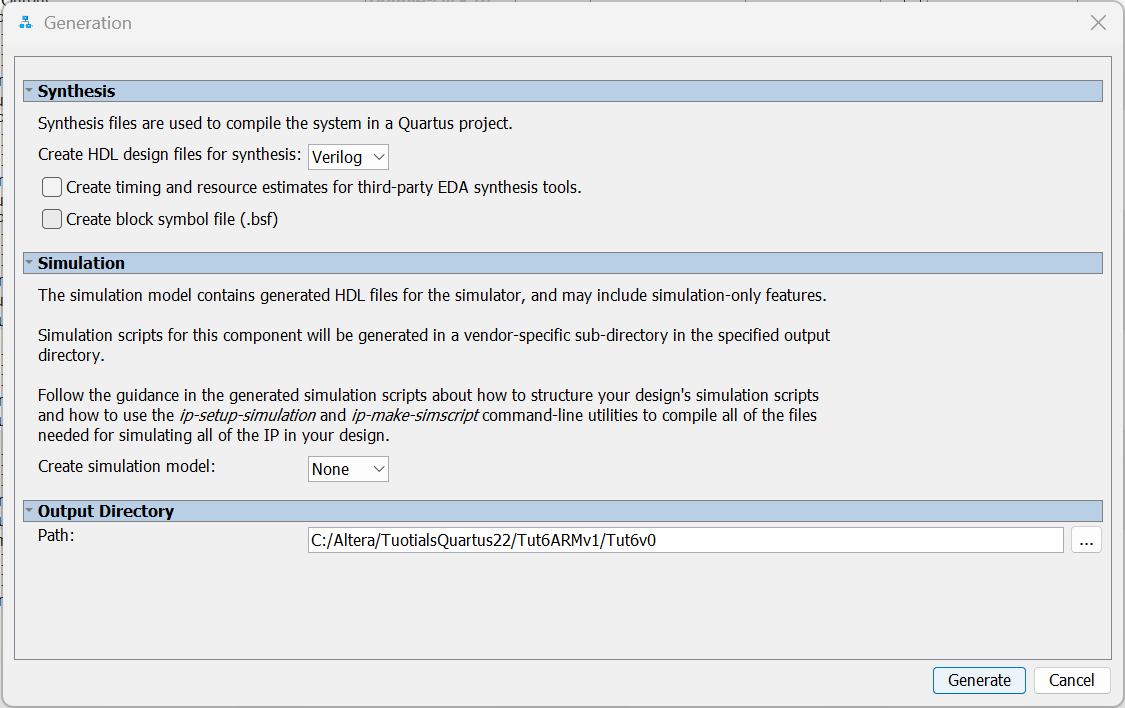
Generate > Show Instanstation Template



Which can be taken as a cue to subsequently instantiate the component within the top level entity.

Now you can build the complete architecture

< Generate HDL>



Choose if you want to create the symbol (useful within a schematic if necessary) or the verilog files to carry out a suitable simulation in Verilog – But both options will not be used in this tutorial

< Generate >

And then

< Close >

< Finish >

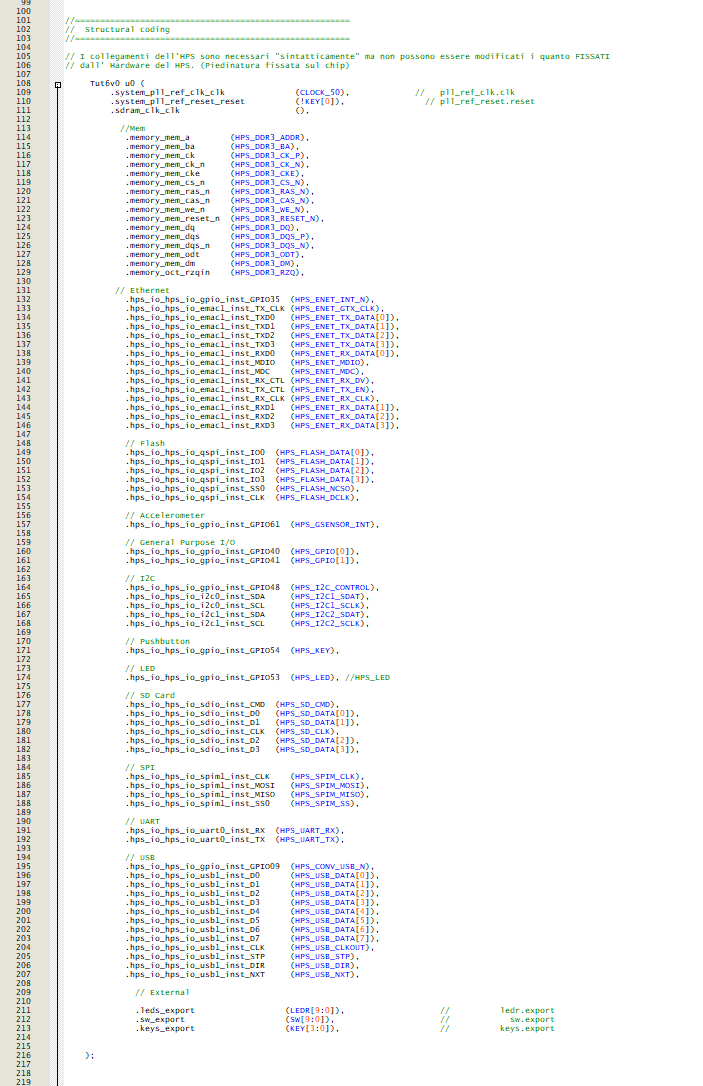
Probably at the end of the process you will see a window reminding you where the files to be manually included in the project have been stored.

# Instantiation of the architecture within the project

The architecture generated in the previous step must now be imported at the global system level and interfaced with any other blocks and signals.

For example, it is necessary to ensure that the clock and an appropriate reset signal arrive at the architecture and its I/O signals related to leds, keys and switches connect to the corresponding pins inside the DE0-SoC board. In addition, it must also be ensured that the various HPS pins dedicated to the connection to external peripherals (such as DDR3, USB, ENET, FLASH, I2C, GPIO, ... ) that are shared externally in the instance of the previously made component, find an appropriate physical connection to the corresponding components mounted on the DE1-SoC board.

So at the level of the "Top Level Entity" generated with the "system builder" tool you will need to create all the appropriate connections:



An example file with the above instantiation is available on the moodle page relating to the course, but you must pay particular attention to the fact that the names of the signals reported are consistent with those assigned to the signals "exported" within the "Platform Designer" tool and if you have opted for different names, it will be advisable to modify the instantiation file accordingly as well. Also note that the signal used for the reset in this case is the inverted KEY[0] signal !

Finally, at the system level, you still need to include the generated ".qip" file in the project.

Project > Add/Remove files in project

Click on the three dots

Go to find the file where it was previously indicated by the dialog box and import it.

<OK>

At this point the entire system can be compiled.

ONLY  **run**  the "Analysis and Synthesis" process

Then you have to execute two scripts to make an appropriate assignment of the parameters and the assignment of the pins related to HPS that impose appropriate constraints regarding the *"Placement", "Mapping" and "Routing" phases*

Tools > TcL scripts

And double-click on

hps\_sdram\_p0\_parameters.tcl

hps\_sdram\_p0\_pin\_assignments.tcl

You only need to do this once per project, and you don't need to run it again if you make changes to the architecture.

At this point you can complete the compilation of the project and download it to the card.

The Hardware Part inherent in the implementation of the system is completed and you can close Quartus.

# Intel FPGA Monitor Program

Launch "Intel FPGA Monitor Program"

File > New Project

Define an appropriate directory (it is suggested to create, within the project created with Quartus, a suitable directory – called for example "software", in which to reside the project that now continues in the creation of an appropriate software for controlling the processor just created). Assign a name to the project and choose **ARM Cortex A9 as the architecture**

> Next

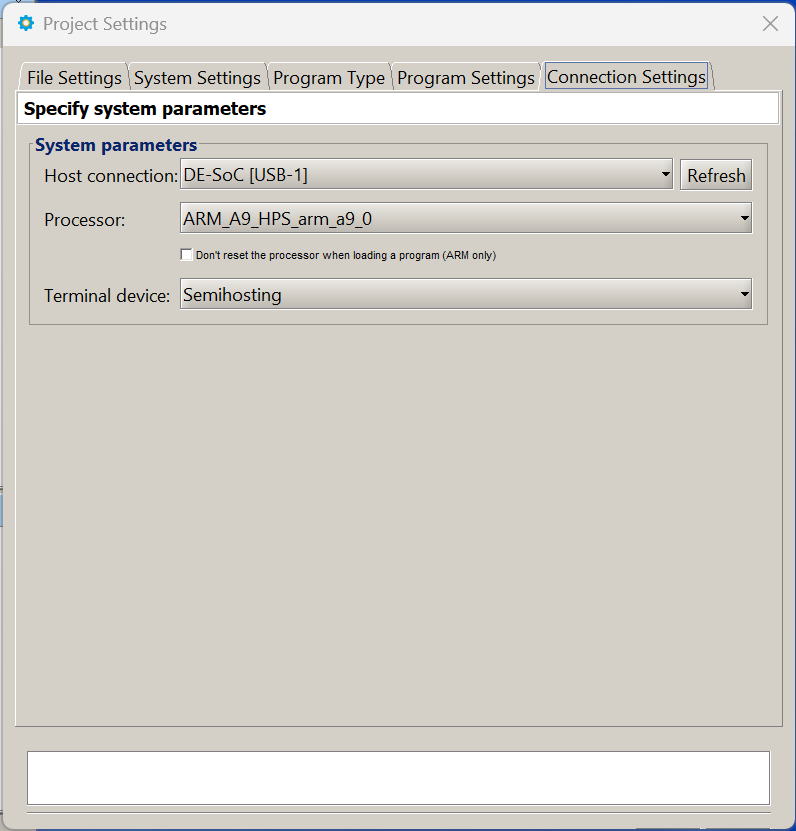
As the type of architecture, choose <Custom>, then indicate the files that characterize the hardware system .sopcinfo and .sof located in the directory where the hardware of the project resides, as a pre-loader the one relating to the DE1-Soc card and as a preloader choose the one relating to the card you are using.

> Next

As a type of programming at the moment choose <No Program>

> Next

Then define the processor and communication interface in "Connection Settings"



> Save

The system will ask if you want to load the configuration file on the board, but if you have a small bug in the system it is better to skip this point and use Quartus to program the board (as already done).

Action > Connect to System

Access the Memory tab

Goto Address go to the addresses where LEDs, Swithes and SysIDs have been mapped (e.g. 0xff200000), check "Query Device"

> GO

At this point you can see the various peripherals through their memory addresses:

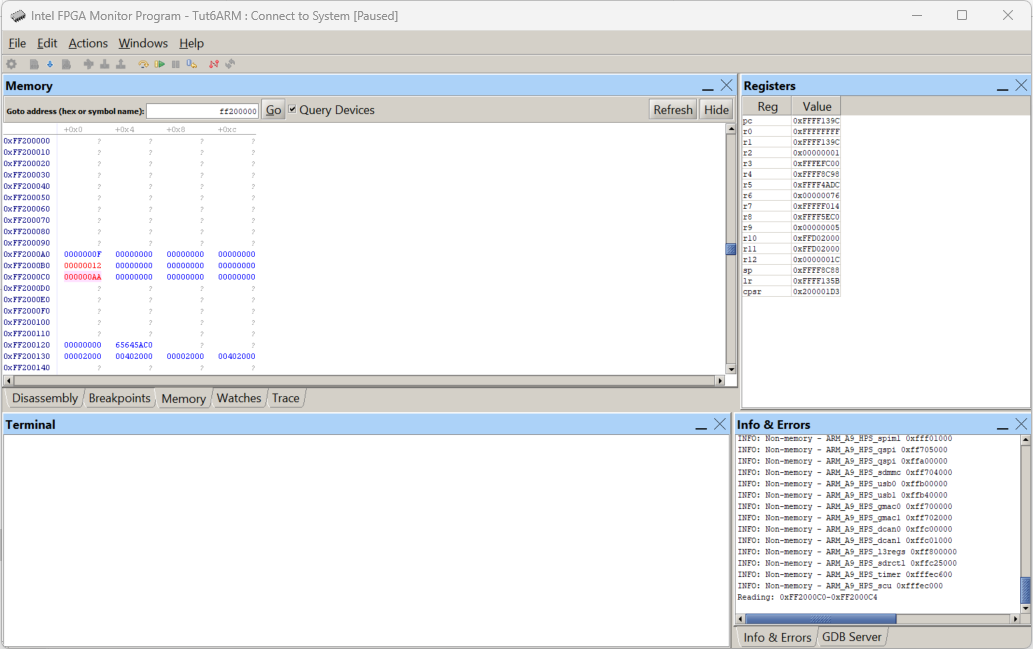
- by writing some data (e.g. 0x2AA to the relative location of the LEDs) you will see the LEDs light up alternately)

- by activating a configuration on the switches and clicking <Refresh> you will see this configuration mapped in memory

- the SysID location will be mapped to the value set via Hardware during the architecture definition phase

- Pressing the Reset button (Key[0]) will turn off the LEDs

The values changed since the previous reading are shown in red.



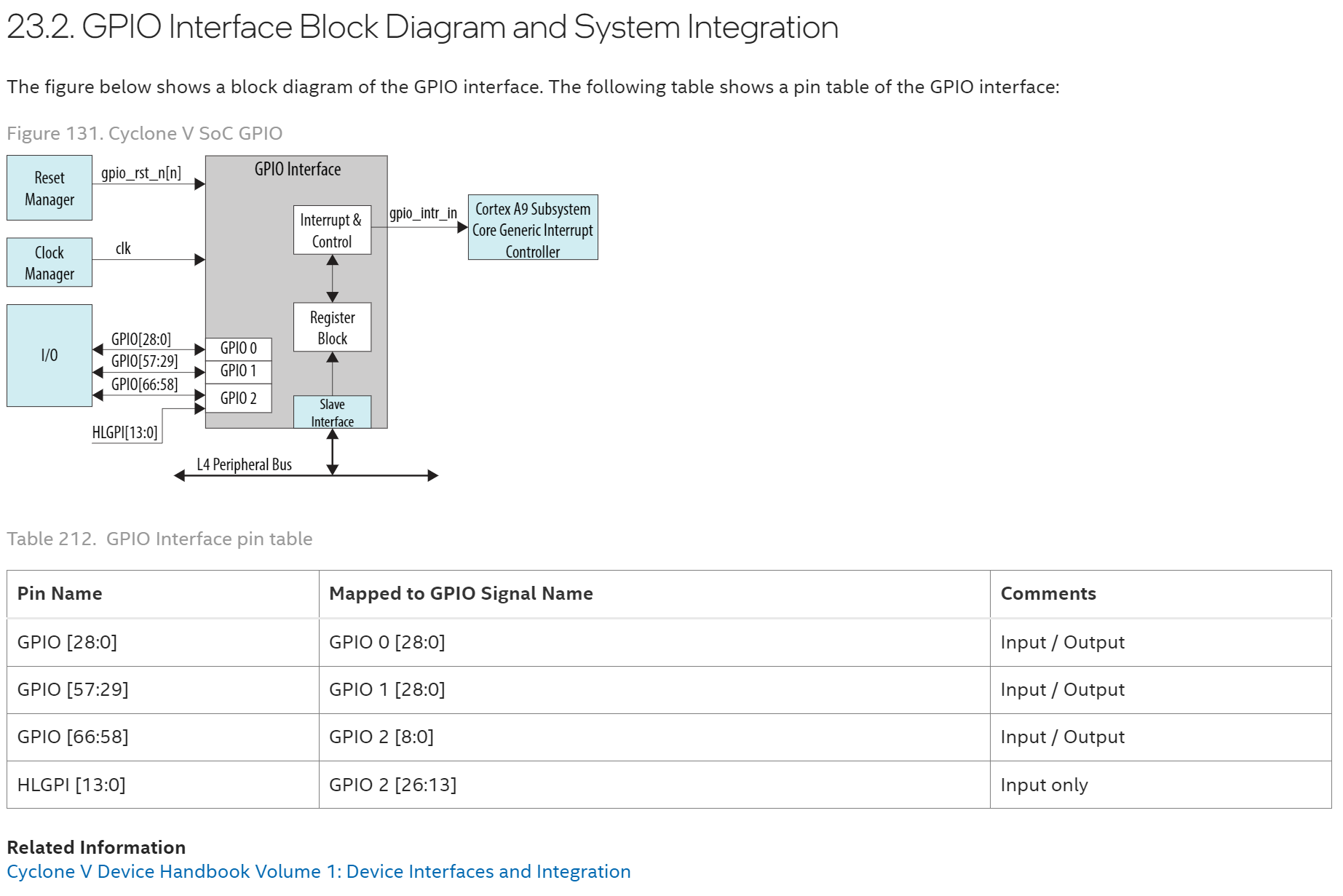
### Access to other embedded peripherals

In addition to the peripherals made on FPGAs and expressly integrated using "Platform Design", the HPS integrated within the CycloneV presents, made in hardware, many other peripherals (ENET, USB, UART, I2C, SPI, ....) of which some have been connected to appropriate compatible devices on the DE1-SoC board, others have remained unused. In particular, we see that two GPIO pins controlled EXCLUSIVELY by the HPS (the *hps\_io\_hps\_io\_gpio\_inst\_GPIO53* and *hps\_io\_hps\_io\_gpio\_inst\_GPIO54)* pins have been connected respectively to a dedicated LED and a dedicated button arranged on the DE1-SoC board.

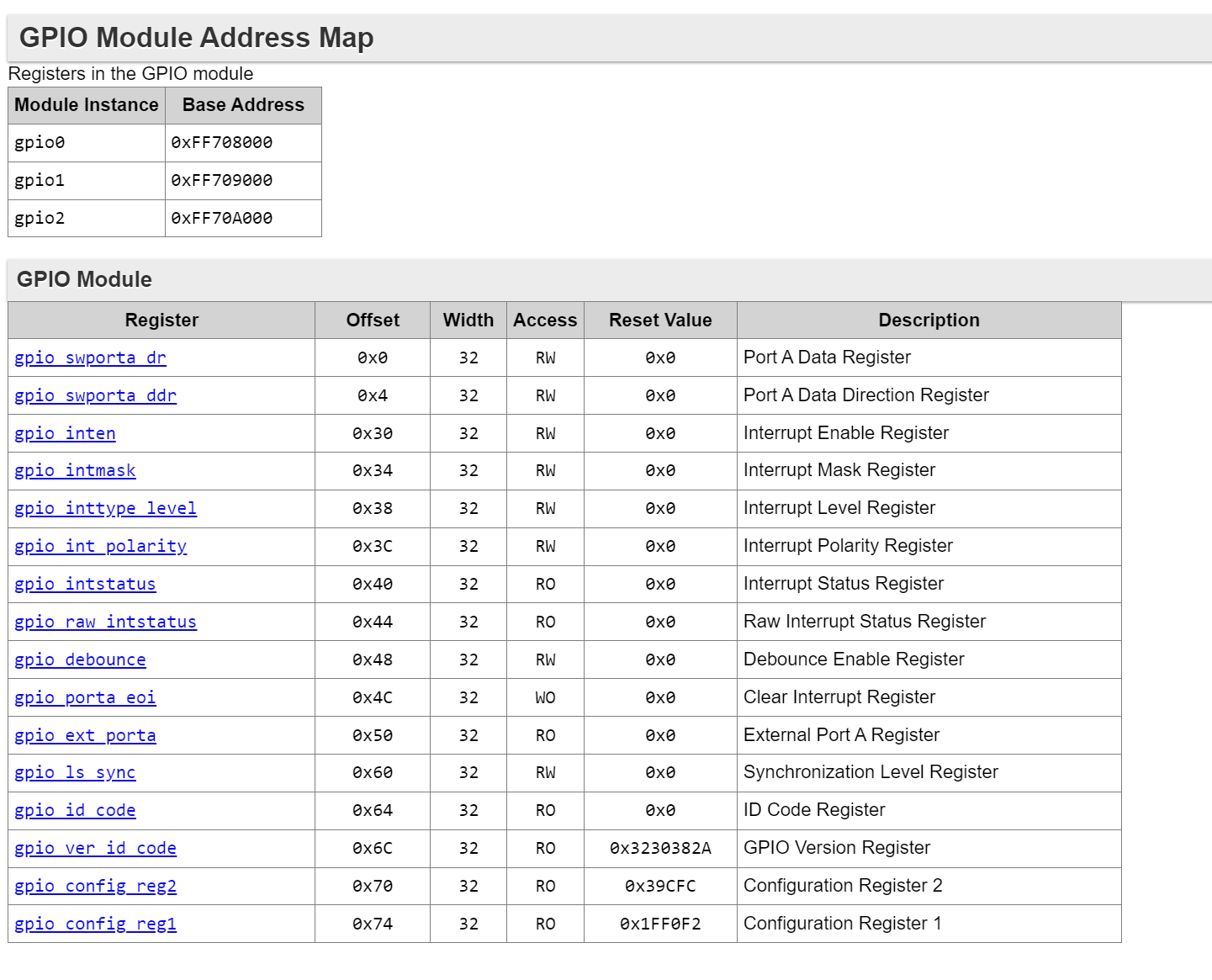
They are therefore visible at an appropriate memory location and controllable/observable through a certain procedure.

Information on the various HPS peripherals and in particular on the GPIO peripherals can be found in the documents:

* Cyclone® V Hard Processor System Technical Reference Manual
* Cyclone V HPS Register Address Map and Definitions

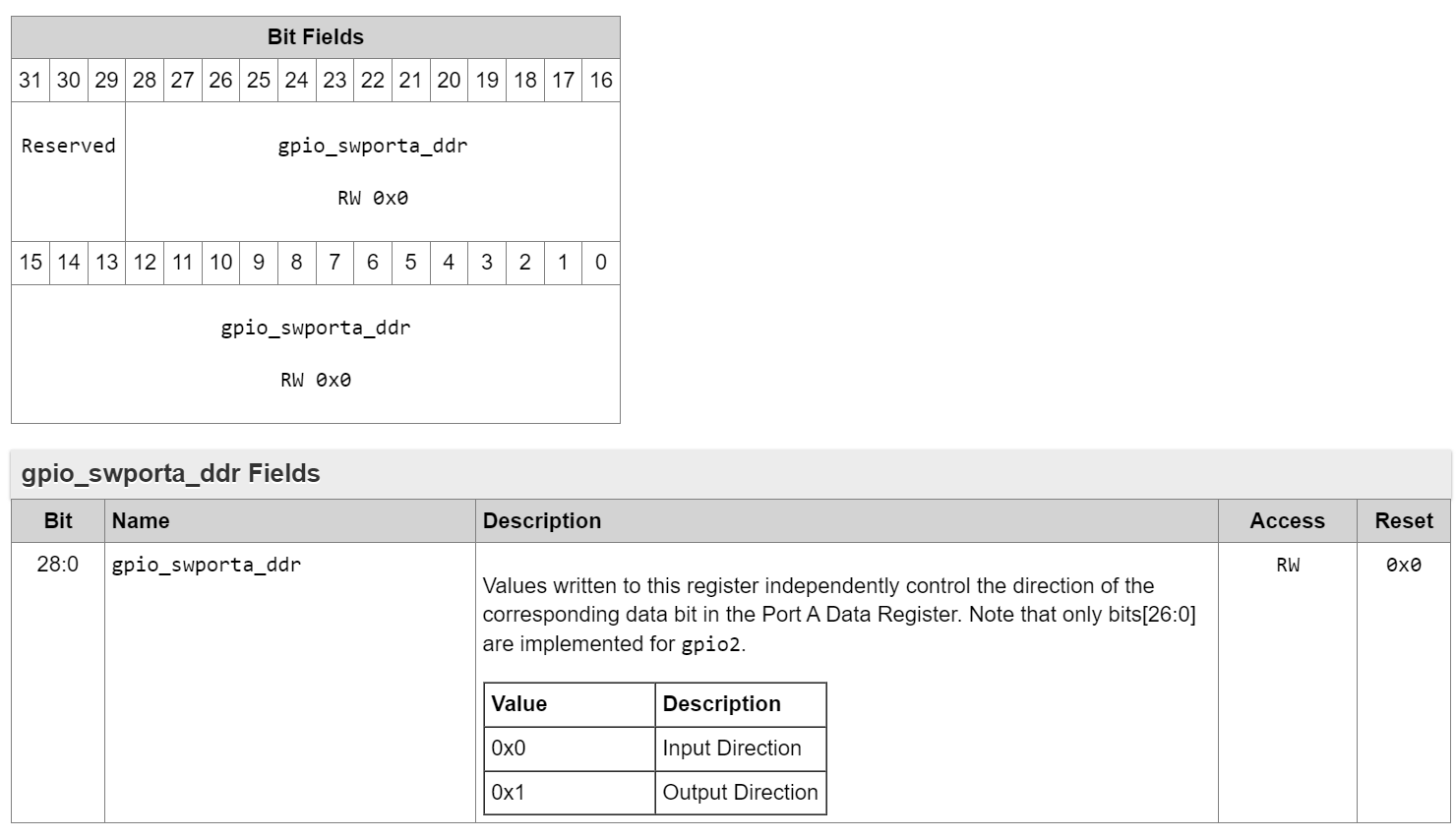


In particular, they find that the different GPIO pins accessible through HPS are grouped in three different banks whose driver is accessible in a precise memory location



While the various registers used to control the driver are accessible in adjacent memory locations.

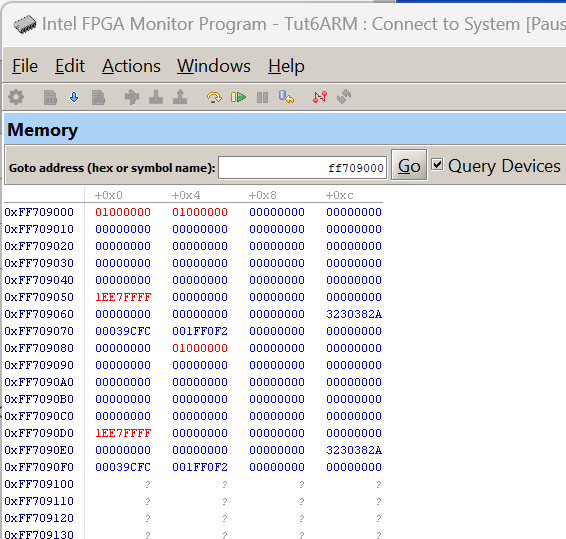
Now taking into consideration the LED, connected to the HPS\_GPIO53, it can be deduced that it is accessible through the gpio1 driver mapped to the 0xff709000 and subsequent locations.The documentation shows that in order to turn on the LED, you must first define the direction of the I/O port for HPS\_GPIO53 and configure it as an output.



This can be done by activating the position bit 24 at address 0xff709004 and then activating bit 24 at address 0xff709000 (i.e. by writing the value 0x01000000 to both locations).

First of all, note that this operation changes the state of bit 24 also at location 0xff709050 (dedicated to reading GPIOs) which passes from the value 0x1FE7FFFF to the value 0x1EE7FFFF (after activating the output mode but with bit24 still at 0) and then returns to the value 0x1FE7FFFF after turning on the LED.

While the reading of the state of the button can be operated similarly based on the value assumed by the position bit 25 always in the location responsible for reading the GPIO 0xff709050 whose value changes from 0x1FE7FFFF when the button is not pressed to 0x1DE7FFFF when the button is pressed.

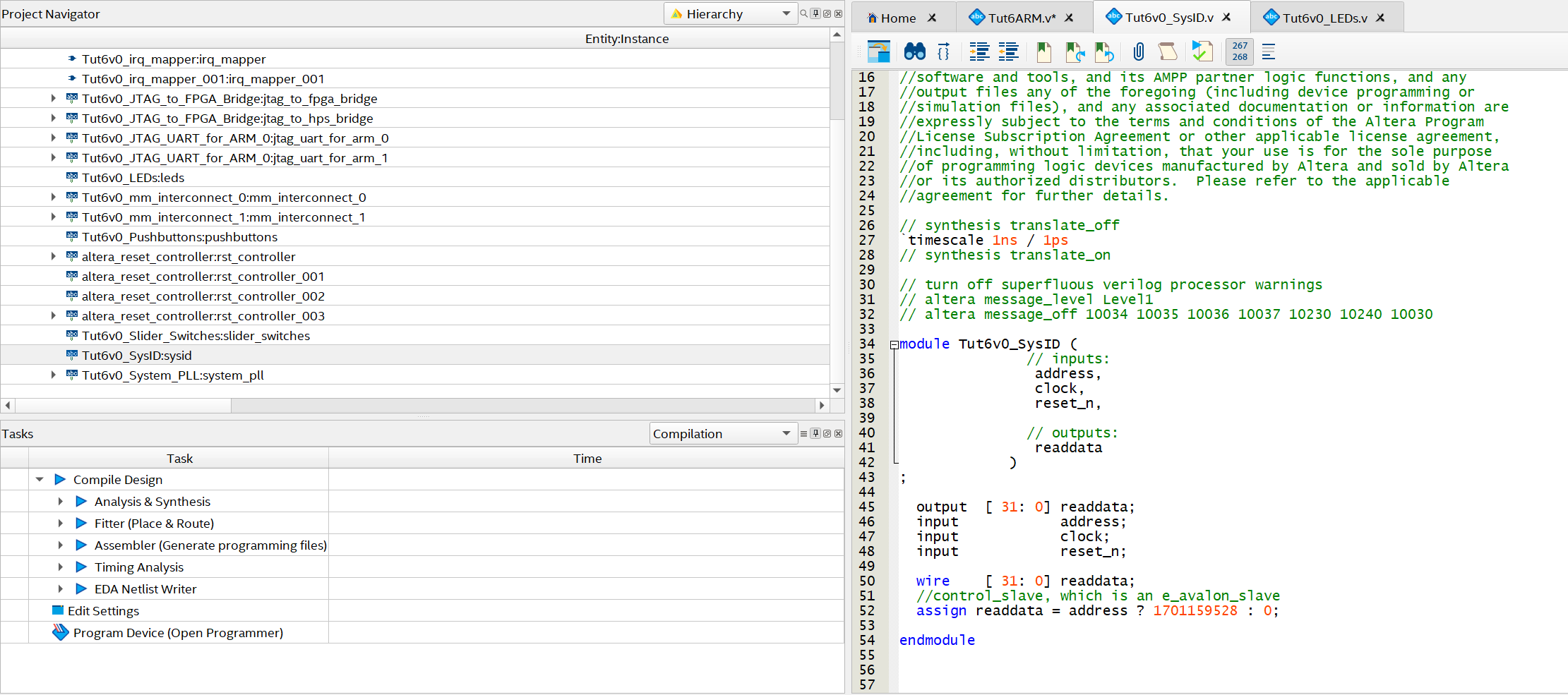


### Creation of generic peripherals to be interfaced with the processor.

Just as dedicated peripherals, developed by third parties, available in the "IP Catalog" of the "Platform Design" tool, have been included in the processor architecture, so Custom peripherals made ad hoc to handle specific problems can be integrated into the processor. These peripherals will use the "AvalonMM" bus as a means of communication to the processor and to the outside world of "conduit" connections. To get ideas on how to write a dedicated peripheral, you can draw inspiration from the codes relating to the peripherals used so far.

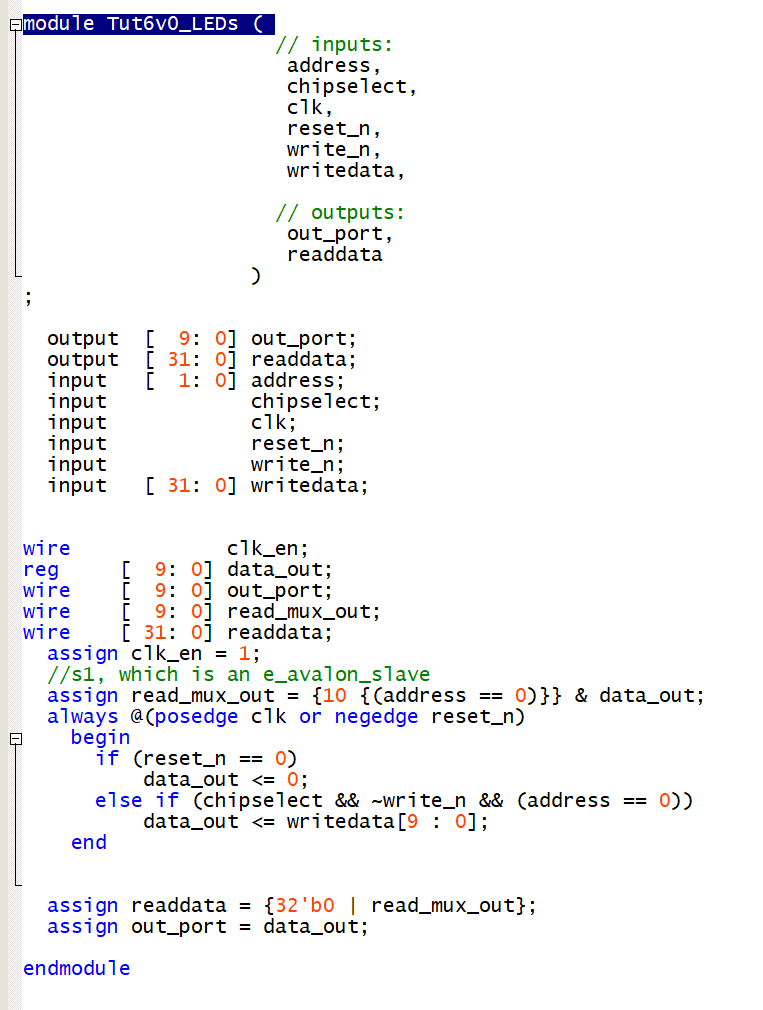
Open Quartus by importing the previous project.

Within the "Project Navigator" window put in "Hierarchy" mode, by opening the subblocks constituting the processor developed in the previous step, certain interface blocks (Led, Switch, SySID) can be recognized. Clicking on them gives you access to their code which can be useful as inspiration to see what signals are involved and how they are managed to obtain the dedicated results.



Focusing on the SysID, it can be seen that the signals with which the processor queries this peripheral is essentially the address bus (which in this case is a single bit) and that when this bit is set to 1 the interface interprets this value as a query, consequently providing the value of the SysID itself on the "readdata" bus.

A little more complex is the peripheral intended to access the row of LEDs. This provides a few more control signals that allow both writing and reading on the peripheral, as well as the signal that physically connects to the LEDs (*out\_port)*

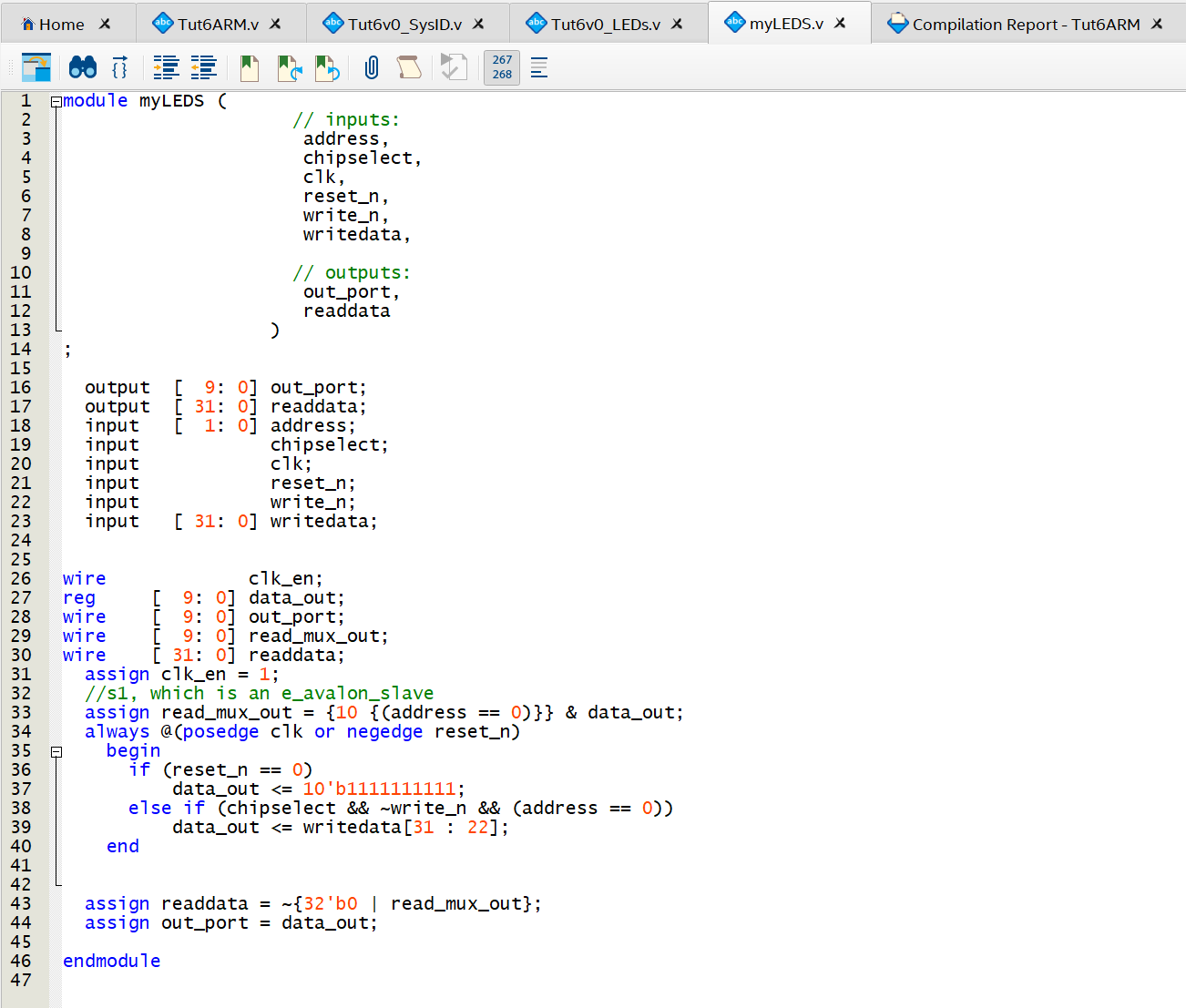


Basically, in order to "write" to the LEDs, the system verifies that "*chipselect*" is activated, that the "*write\_n*" signal is precisely at the low level (indicating the write operation) and that "*address*" is zero, all obviously synchronized on the clock and with an appropriate reset signal. When all conditions are met, the data on the writedata line (of 32bit) relating to the bits [9:0] is reported on the intermediate lines "*data out*" and from this to "*out\_port".* At the same time, the data that the processor reads on the *readdata*  bus is the *read\_mux\_out* extended to 32bit with the addition of an appropriate number of zeros. Where the *signal read\_mux\_out* itself is composed (if the address is equal to 0) of the intermediate signal "*data out*" just calculated, otherwise it will be null.

A certainly interesting practice from a didactic-educational point of view could be to modify this peripheral to create a slightly different one to be replaced with the one used in the previous part of the tutorial.

For example, let's create a peripheral where the value with which to control the LEDs resides in the most significant positions of the register that controls their ignition (instead of the least significant ones), the reset signal instead of turning off the LEDs, turns them on and where the value read is the opposite (i.e. the bit-by-bit negation) of what it would report in the current case.

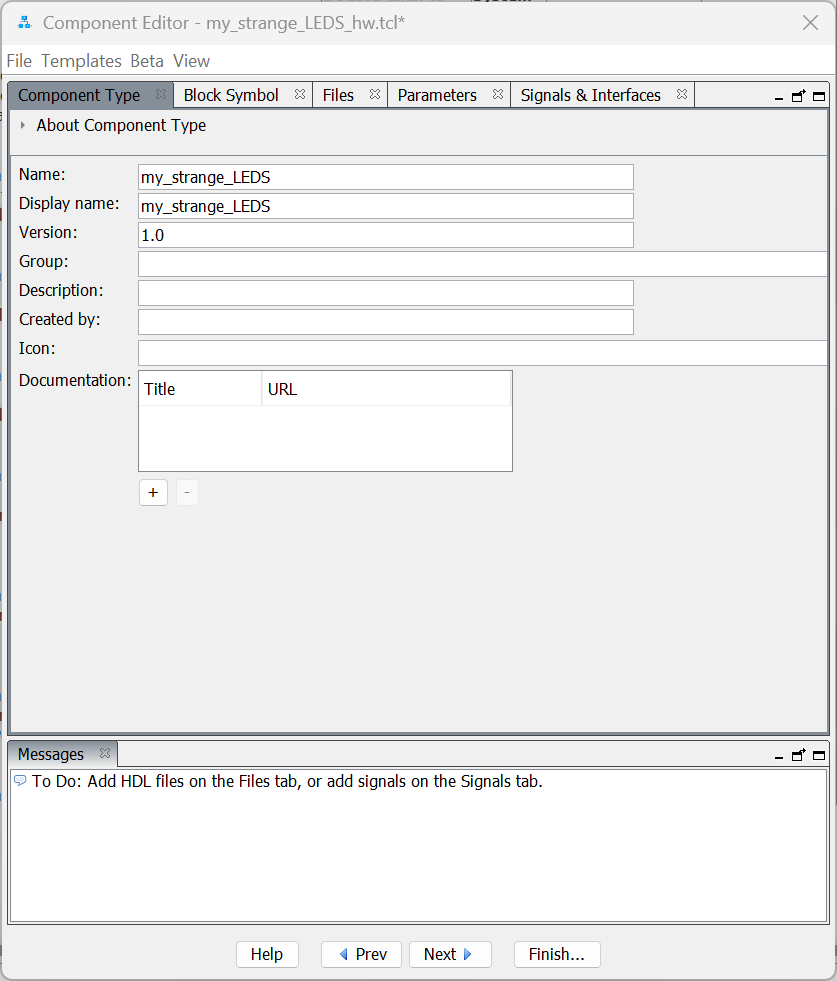
Create a new Verilog file and copy the structure of the device you just scanned into it. Change the name of the module and make the changes suggested in lines 37, 39 and 43 respectively



Save the file appropriately.

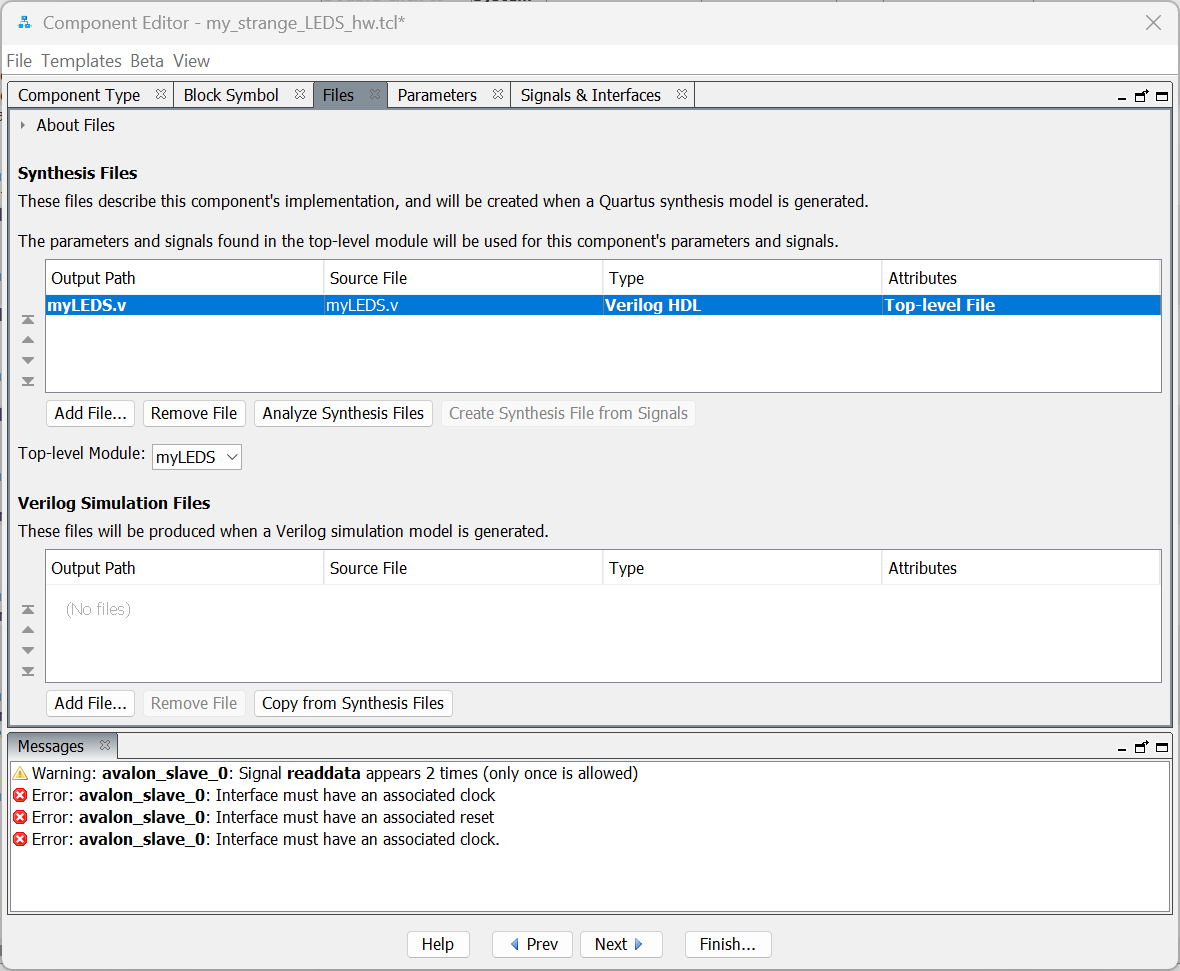
Then open the "Platform Design" tool

In the left window, double-click on "New Component"

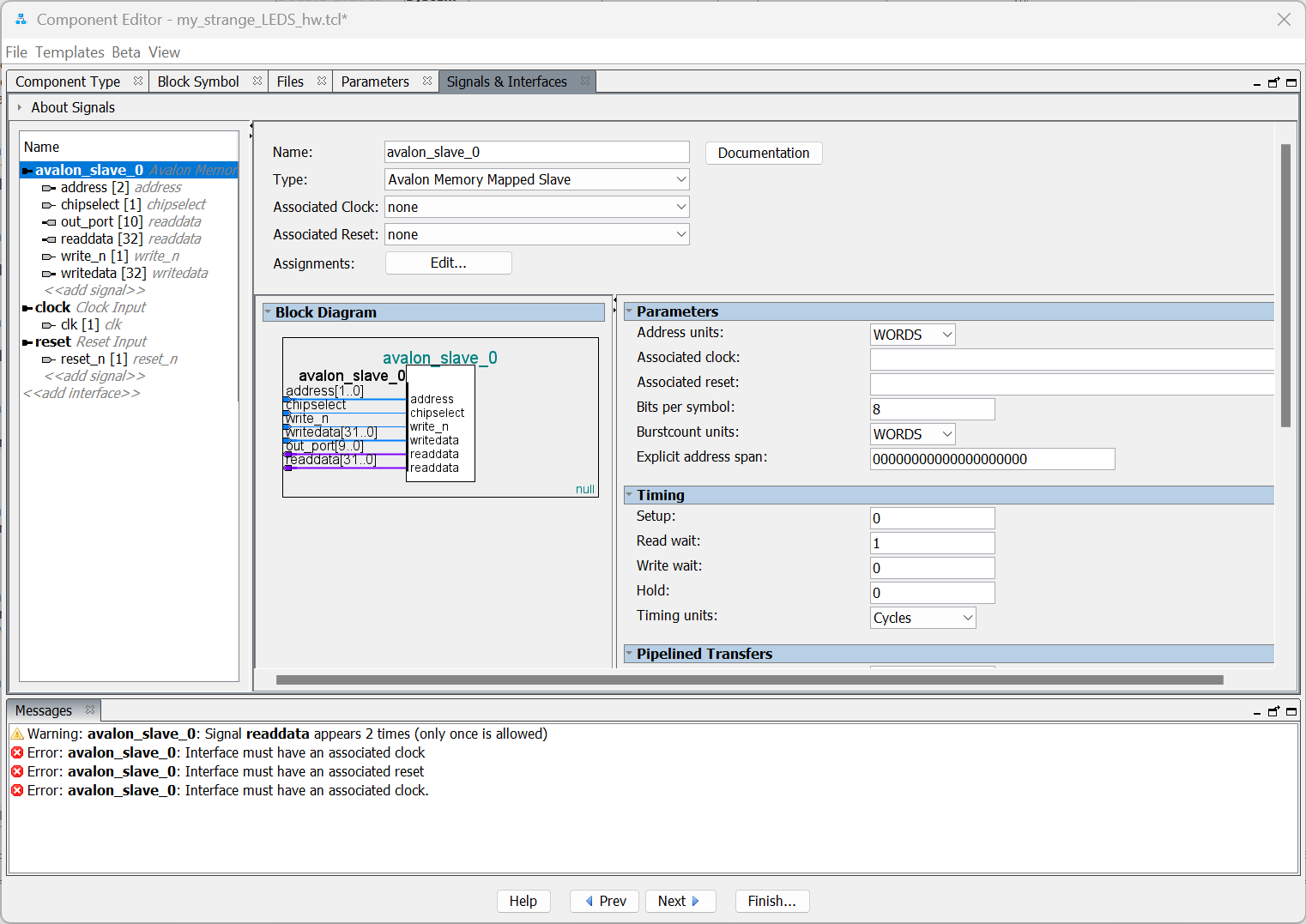


In the window, provide a name for the component (interface) that you are going to make.

In the Files tab, click on <AddFile> add the File you have just created and then click on <AnalyzeFile>. If there are syntactic errors, error messages should appear here



On the other hand, some errors related to the Interfaces appear. Then move to the interface tab and you will notice how the system has already automatically recognized, thanks to their names, some interface signals and has associated specific interfaces with them. If other names were used, it is plausible that the assignment of interfaces and signal types may have to be done manually.

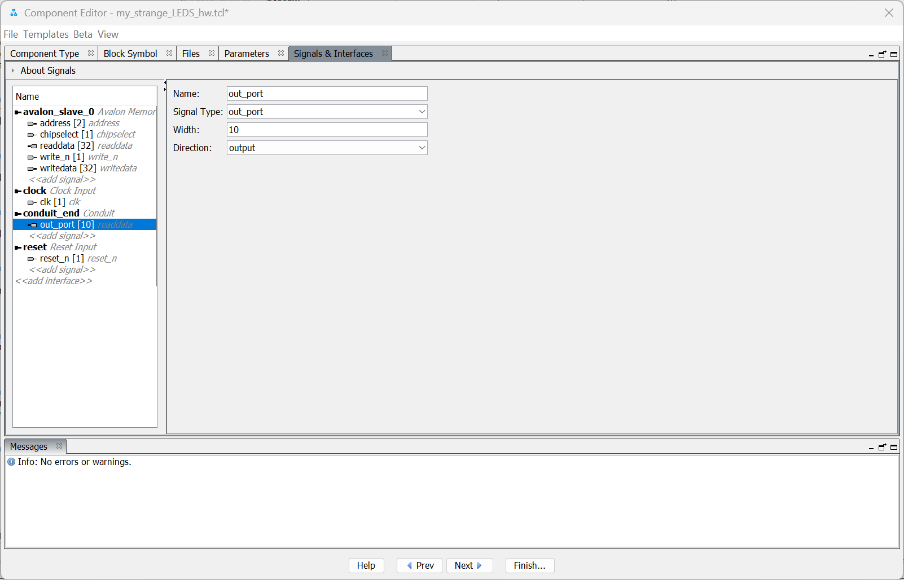


There are currently three interfaces:

* avalon\_slave\_0 (with different signals)
* clock
* Reset

Note, however, that port out\_port has been (mistakenly) recognized as a read data bus

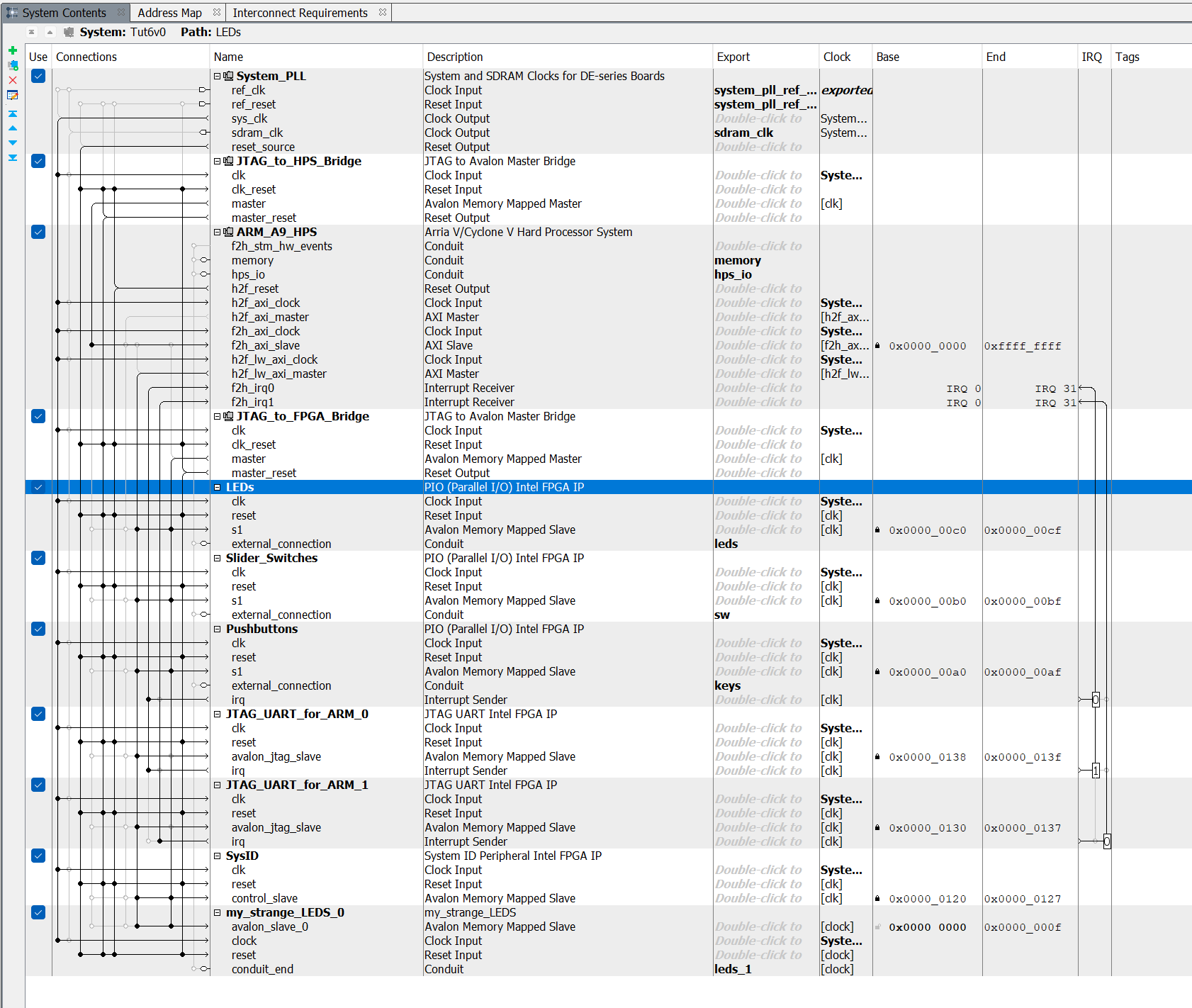
It is therefore necessary to create a new "Conduit" type interface, on which to drag the signal port\_out



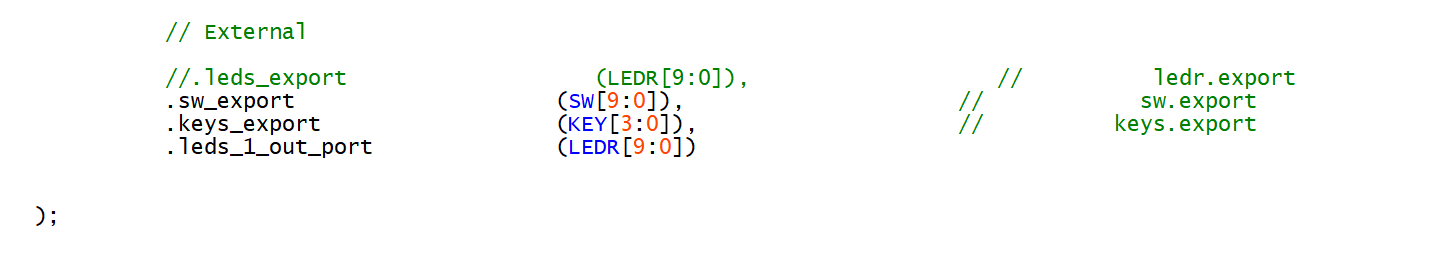
And in the right window establish the "type" of signal, for example by creating an ad hoc name, in this case out\_port (which will then be the name with which the signal will be located within the instance).

In addition, by clicking on the various interfaces, you can determine which are the reset and clock signals that control them.

Once the process is complete, the new component will be available among the interfaces to be included in the processor. And it can be integrated into architecture. In the image below, the new component is added to the existing ones, suitably connected to the AvalonMM and the interface signals to the outside exported and renamed.



At this point the processor can be generated and instantiated at the Top Level Entity level, but altering the connections in order to reflect the connection of the LEDs with this new interface:



By downloading the card you will immediately notice that in the reset phase all the LEDs are on, moreover the data displayed on the LEDs are related this time to the most significant bits and the reading that takes place automatically after writing returns a different value from what was written.

In the example below, when writing 0xAAAAAAAA it is noted that the value read simultaneously is 0xFFFFFD55, i.e. the 22 most significant bits are all set to 1, the 10 least significant bits are shown inverted and the LEDs light up alternately on the LED bar in the board.

## Creation of other peripherals to interface with the processor.

You want to equip the processor with three additional peripherals: one that drives the seven-segment display and displays a hexagesimal code based on the data received from the processor, a timer that increases the value every microsecond, but with the possibility of setting the starting value, a rotary encoder driver that increases/decrements the value of a register based on the steps performed by the encoder in one direction or the other

The code of these devices are available on the moodle page of the course

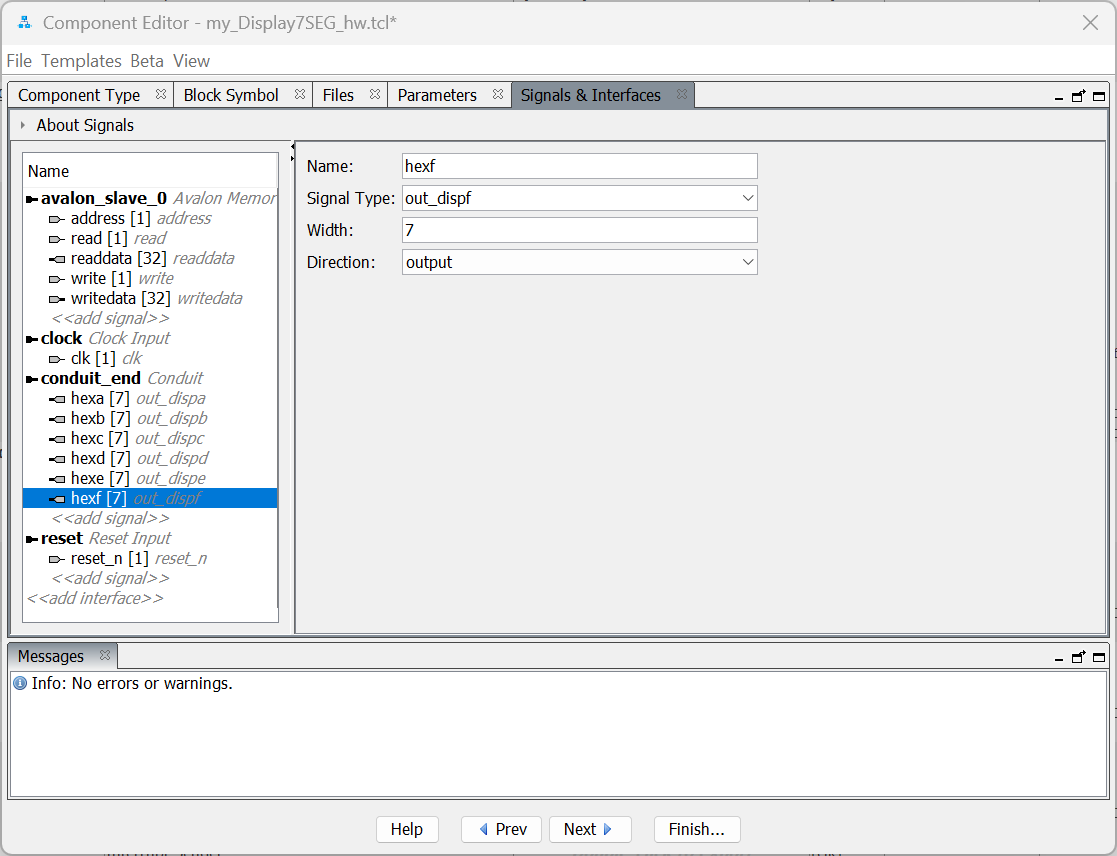
#### 7-segment display

The code consists of two modules:

* SEG7\_LUT is a synchronous circuit that receives a 4-bit signal as input and generates the 7 signals as output to display the corresponding digit on a single display
* my\_led7seg is a higher-level system that when it receives the write signal saves the input data in an internal register (reg\_data). The data in this register, grouped 4 by 4, provides the input for the previous block. They also provide the data to be read when the read line is active. At the output there are 6 separate 7-segment buses, one for each display.

Import the component into the "Platform Design" following the procedure defined above: adding the Verilog file and analyzing it, but this time, since the file contains two modules, you have to define in the same window which is the highest module in the hierarchy (obviously "my\_led7seg").

In the "interface" tab create a new output interface in which to bring the 6 buses destined for the 6 displays and give each of them a unique "Type". Also connect clock and reset to all the interfaces that need it.



#### Timer

This component does not require any connection outside the processor, but it can be very useful for estimating the time elapsed between two program execution points. It is basically a counter that is incremented at a constant interval Furthermore, through the SUBSAMP parameter it is possible to establish the frequency with which the counter is incremented.

As previously done

* create a new component
* Import the description Verilog file and analyze it
* the interfaces
* on the parameters tab it is highlighted how the component has been created leaving 3 parameters adjustable: DATA\_WIDTH, ADDR\_WIDTH, SUBSAMP. But since the data bus must be 32-bit for a correct interface with the processor, this parameter can be deselected from the adjustable parameters. The ADDR\_WIDTH is also quite redundant for operation, but we can leave it to see how the interface behaves if we were to change it. Finally, SUBSAMP is needed to set the timer frequency.

#### Rotary encoder

This component is actually a counter with the ability to increase/decrement the count by analyzing two periodic input signals and their relative phase shift. The count can also be set to a certain value using the write operation on the avalonMM bus, while the read operation returns the value of the counter itself.

As in the case of the Timer, this component also provides 3 adjustable parameters which, however, in practice modify the behavior of the system minimally.

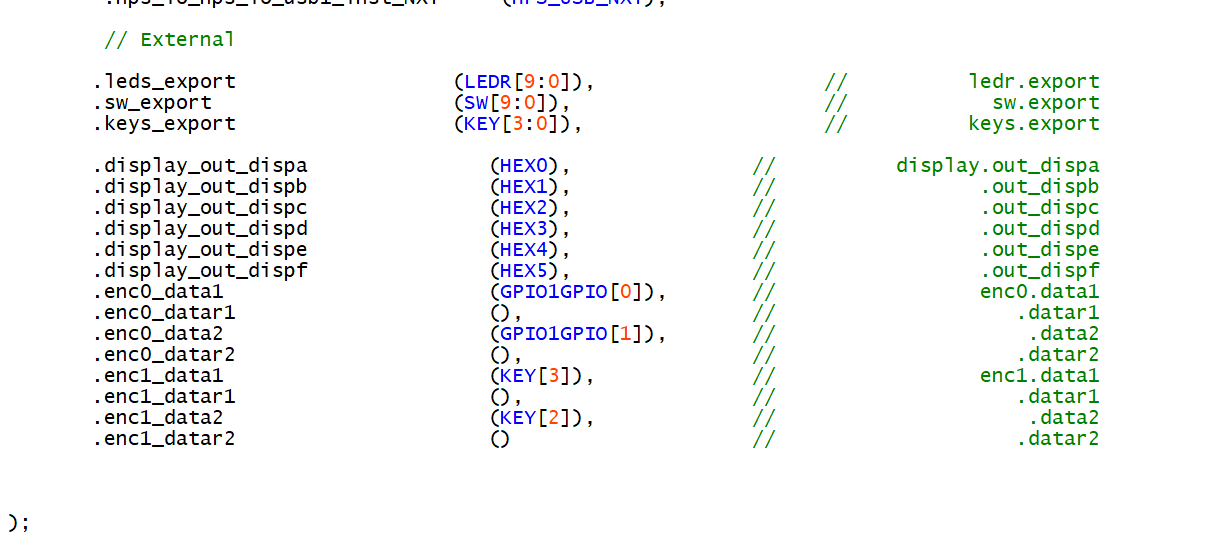
As in the previous steps, create a new component by importing the available Verilog file.

#### Overall HPS

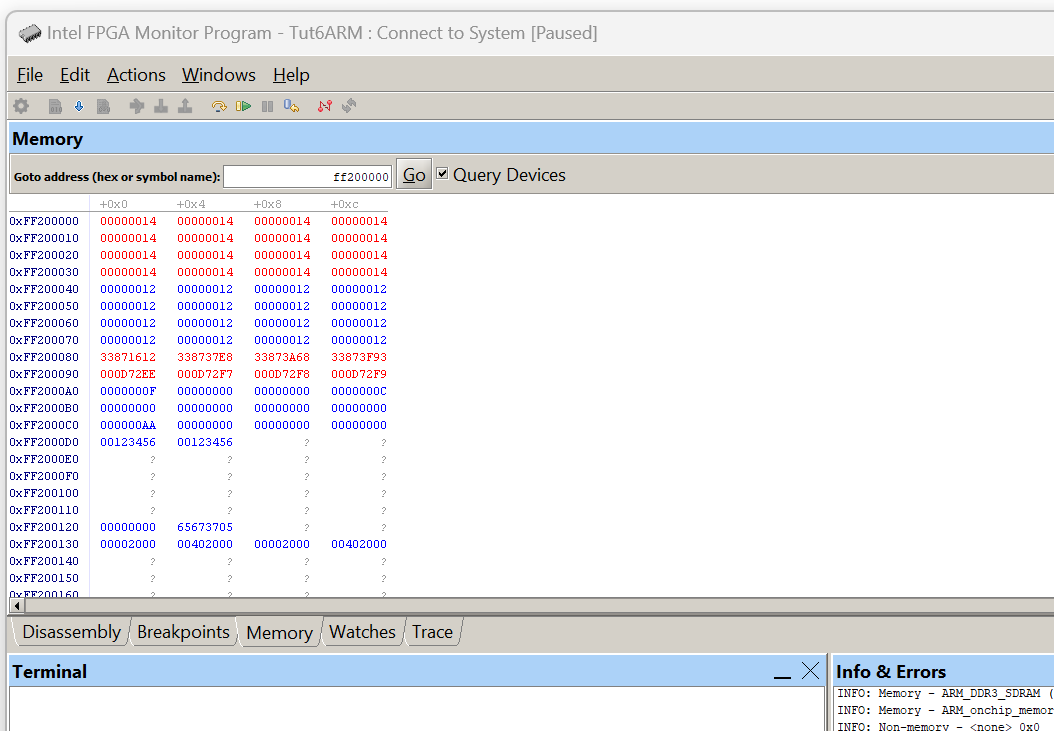
Import now through Platform Designer

* two Timers and adjust the parameters so that one has a period of 1 millisecond (SUBSAMP = 50000) and the other of one microsecond (SUBSAMP=50)
* a seven-segment display
* two rotary encoders – Two are made so that while one takes the signals through the GPIO from a real external rotary encoder, the other takes the signals from the buttons (KEY) in order to emulate their operation through the combined pressing of the keys.

At the Top-Level-Entity level, the component is instantiated by adding all the appropriate connections with the outside world.



Once the project has been completely compiled and downloaded to the card, it can be accessed through the "Altera Monitor Program" and all the interface data can be found at the appropriate memory locations



Note in particular

* The data relating to rotary encoders whose ADDR\_WIDTH is by default equal to 4 actually occupy 16 memory positions, although with all the same data
* The Data relating to Timers are different for each reading and even when a data is written to it, the time elapsed between writing and reading means that the data is different from what is written
* The data in position 0xFF2000A0 are relative to the buttons
* The data in position 0xFF2000B0 are related to the switches
* The data in position 0xFF2000C0 are relative to the LEDs
* The data written in position 0xFF2000D0 is shown in the same form on the seven-segment display.

#### Software

At this point you can write the system code in C which, by interfacing with the peripherals, performs the desired function.

<Continue>