DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 391 – Computer Systems Design Studio
2018/2019 Term 2

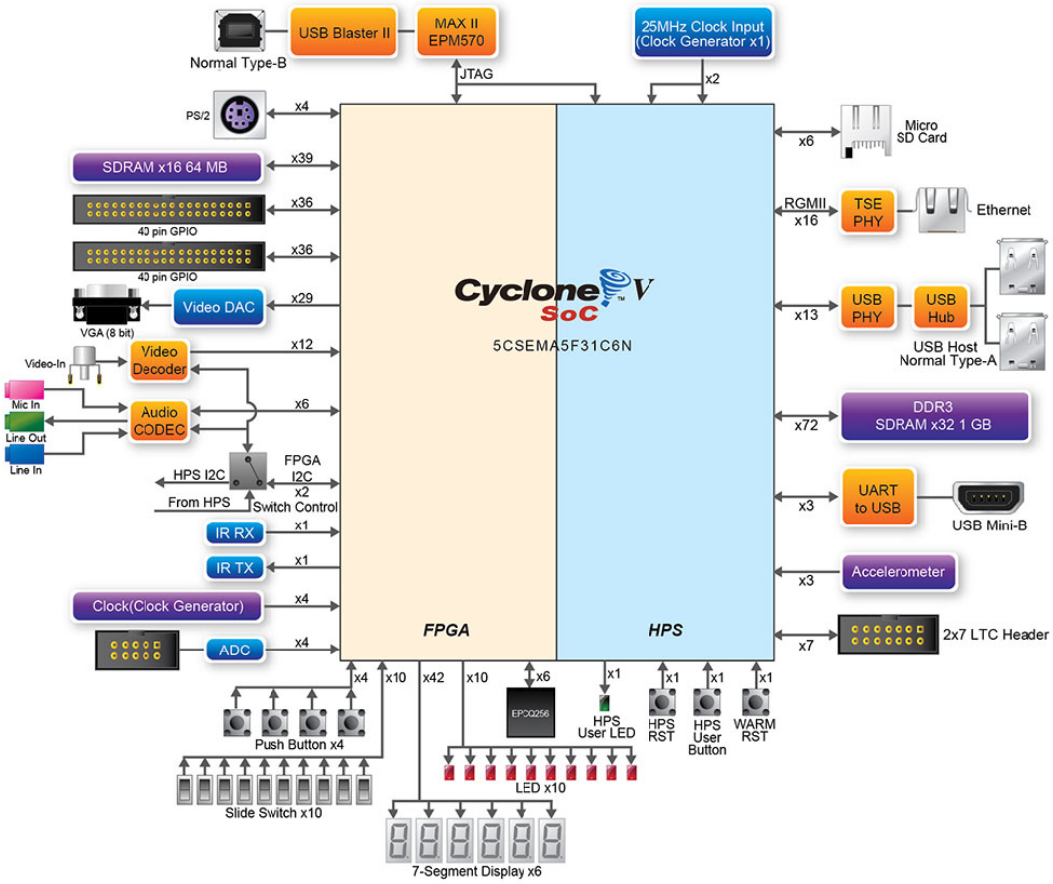
Tutorial 1.2 - Using QSYS to make an ARM HPS (Hard Processor System) on the DE1SoC board

This is quite a long and detail orientated tutorial which is the basis of everything else in module 1 of this course so take your time with it – don't rush, double check things, work slowly and methodically.

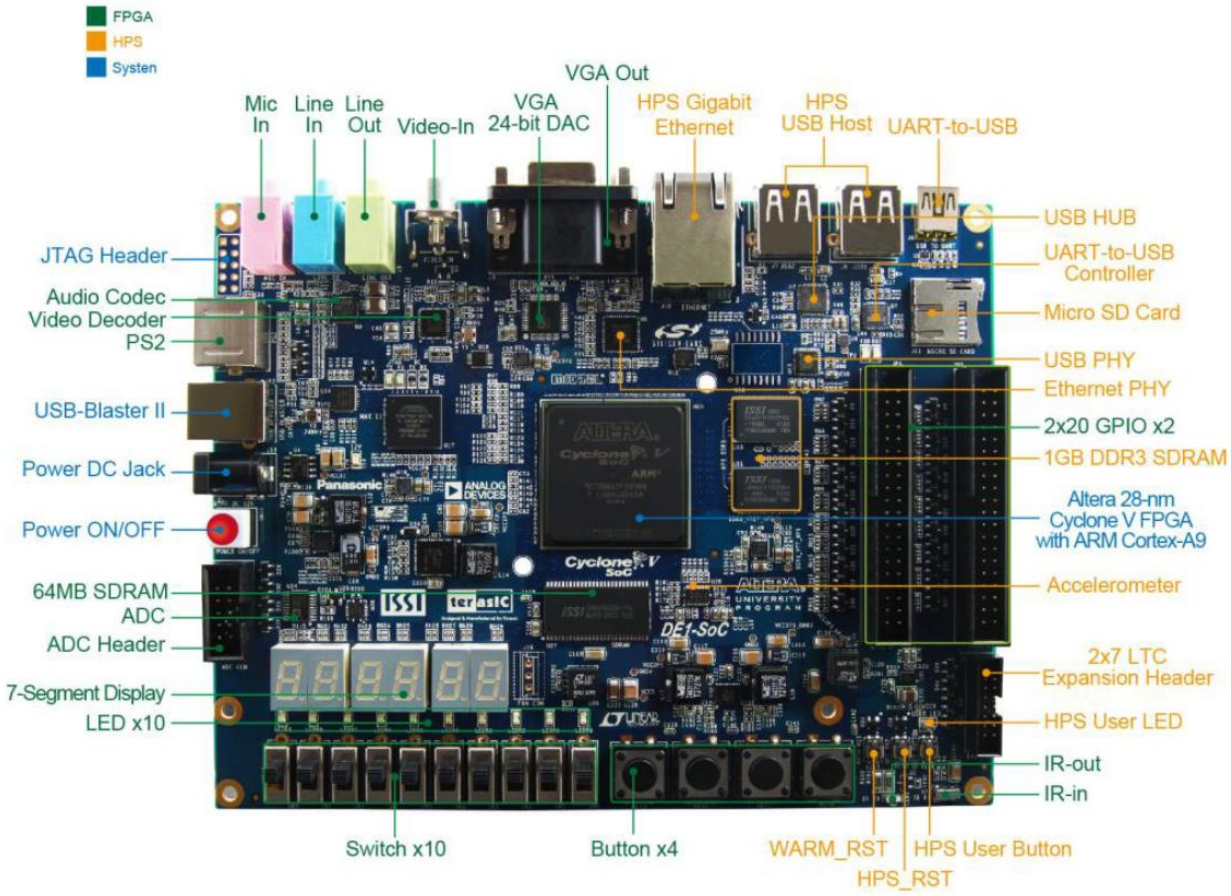
The DE1SoC board with its Cyclone V chip makes an interesting platform to build a system around. In essence it is composed of two separate halves

1. An HPS (Hard Processor System) side with a hardwired, already laid out in Silicon, Dual Core ARM processor running at approx 800Mhz with built in hardware such as Interface to 1Gbyte of Dram on the DE1, plus IO devices such as multiple Can Bus, I^2C, UART, SD-Card controller, USB, Ethernet ports etc. In this sense it can function on its own.
2. A programmable FPGA side where we can design in Verilog or VHDL, additional hardware devices. These can then be interfaced to the ARM hard processor system through *bridges* that connect the two halves together.

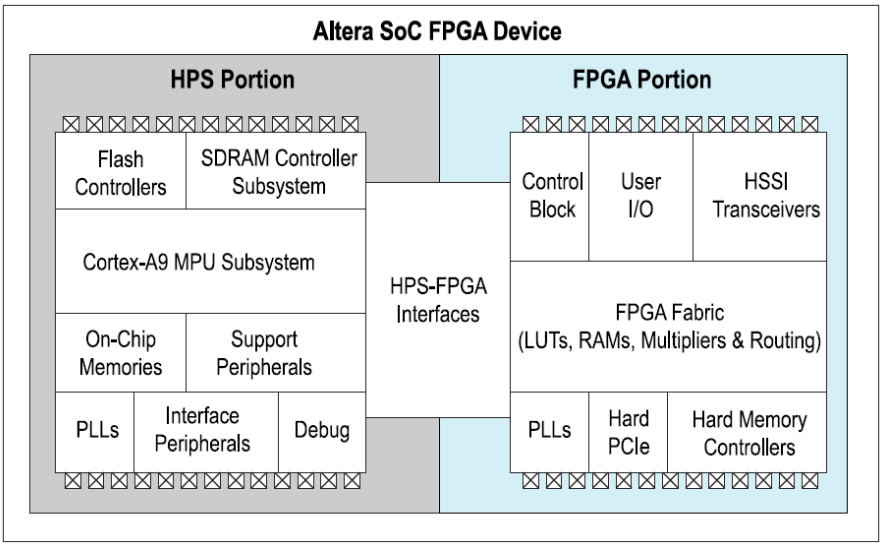
These two halves are illustrated below:-



An illustration of the board itself is shown below (*taken from the user manual*). The items highlighted with Green text are physical devices that are can be accessed via the FPGA part of the board. The things highlighted in Orange are connected to the hard wired ARM processor.



You can see how the two parts are connected together in the illustration below using **HPS-to-FPGA bridges**, which allow things on the FPGA side of the chip to be accessed by the ARM cores.



HPS-FPGA Bridges

The HPS-FPGA bridges provide a variety of communication channels between the HPS and the FPGA fabric. The HPS-FPGA interfaces include:

- **FPGA-to-HPS bridge** – a high performance bus with a configurable data width of 32, 64, or 128 bits. This bridge allows the FPGA side of the chip to access slave devices on the HPS side. It also permits IO device and memory interfaces on the FPGA side of the chip to be accessible to the ARM processors as part of their address space.
- **HPS-to-FPGA bridge** – a high performance bus with a configurable data width of 32, 64, or 128 bits. This bridge allows the HPS to access slave devices on the FPGA side. This is sometimes referred to as the "*heavyweight*" HPS-to-FPGA bridge to distinguish it from its "*lightweight*" counterpart described below.
- **Lightweight HPS-to-FPGA bridge** – a bus with a 32-bit fixed data width that allows the HPS side ARM processor to access 8,16, 32 bit slave devices on the FPGA side of the chip.

For the purpose of this course, this is we will be making a lot of use of the 3rd bridge above, since this is both powerful and fast enough to be able to support pretty much any 8, 16 and 32 bit IO devices that we design on the FPGA part of the board. Through this bridge we can connect the IO devices that we design to the ARM processors and allow high bandwidth communication. You would probably not use it for communicating with Memory, since that usually requires VERY high bandwidth – but you could, it would just be slightly slower. Memory and much higher performance devices would probably be hung off the heavy weight **HPS-to-FPGA bridge.**

Address Map of the Lightweight Bridge

Every IO device and interface that we design within the FPGA side of the chip can be hung off the end of the lightweight bridge and can be accessed in software ('C' code) provided we know where it exists in the memory map of the ARM processor's 4GByte address space.

The **base address** of the lightweight bridge is **0xFFC00000**. This address cannot be changed as it is hard wired into the HPS/ARM chip, thus everything we hang off this bridge is accessed as an **offset** relative to this base address e.g. the address of any device hung of the bridge is (**0xFFC00000 + Some Offset**). In designing our hardware devices, we are thus only interested in defining the *offset*. In C code, we will have to use pointers to access the device by adding the offset to the base address **0xFFC00000**. More on this later, but for now here's an example where we set up a pointer to a 32 bit wide (i.e. integer sized) hardware device which has been designed to occupy an **offset** of 20.

```
volatile unsigned int *MyIODevice = (volatile unsigned int *0xFFC00020) ;
```

Boot Pre-loaders

The ARM processors are complex chips and there is a multitude of HPS specific IO devices attached to them that can be configured in Quartus. In fact there are more devices on the chip than can be brought out to pins in the Cyclone V, so choices and sacrifices have to be made on the DE1 board as to which

devices we want access to. Regardless of choice, these devices need to be carefully initialised after a reset/power-on sequence.

The software to do this is called a boot Pre-loader. It is run after the initial Boot code (*triggered by a reset*) is run. The boot code is fixed in ROM on the Cyclone V and cannot be change, but the pre-loader can. The initial Boot code, attempts to locate and run a user defined pre-loader after a reset/power-on to initialise the system to an application ready state. Once this is done, we can run our application code or perhaps boot an operating system.

The pre-loader typically performs the following actions:

- Initialize the SDRAM interface
- Configure the HPS I/O devices
- Configure pin multiplexing so that the devices we wish to use on the ARM core are brought out to pins on the Cyclone V chip
- Configure HPS clocks
- Initialize the flash controller (NAND, SD/MMC, QSPI) that could contains or firmware or OS
- Load the firmware/OS into SDRAM and run it.

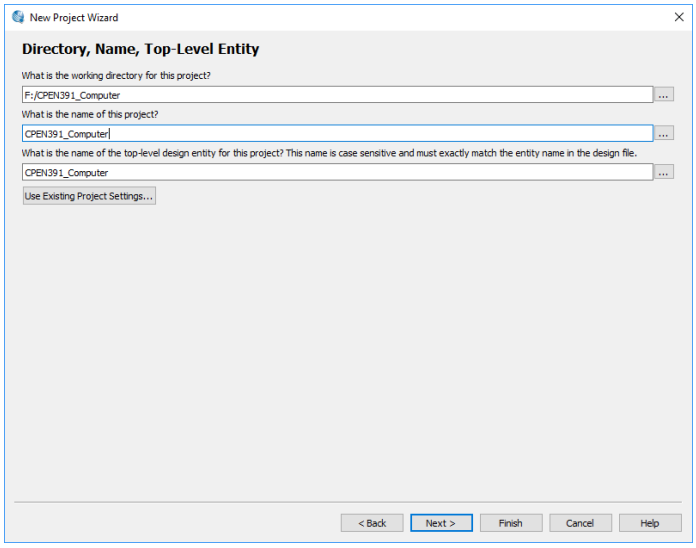
By default the pre-loader holds the 2nd of two ARM cores (CPU1) in a reset state – if you want to use it, you have to release by yourself.

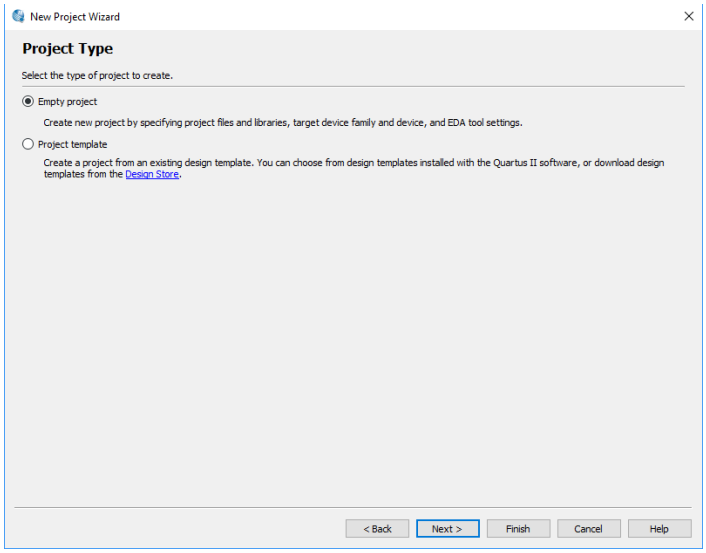
Writing a pre-loader is a reasonably complex job, so we won't be doing that for now, instead we'll be making use of a pre-written one to simplify things at this stage. We will only focus on writing the application (*that's the interesting bit*). We only mention the pre-loader here as it is implicitly used in some parts of this tutorial and we have to download it **before** downloading our application.

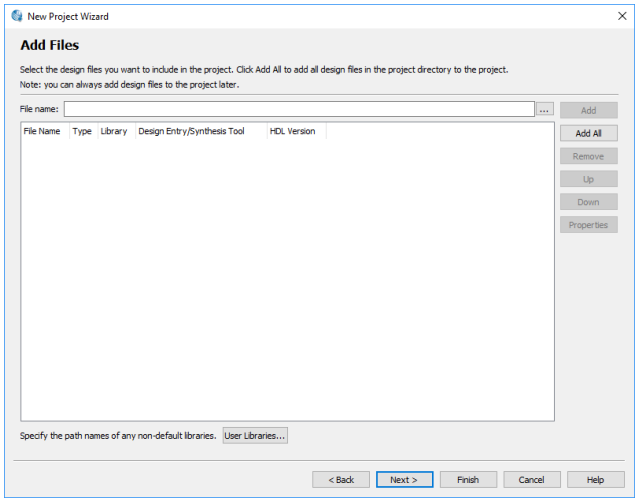
Question: How do we create all our FPGA specific hardware and connect it to the bridges and hence the ARM processors? **Answer**: We use a sophisticated tool in Quartus called **QSYS**. This tool will enable us to build the system configuration, the IO devices we want, and fix their **offset** addresses relative to the bridge base address. We can then write the software ('C' program) for the ARM processor to talk to these devices.

Creating a QSYS generated system

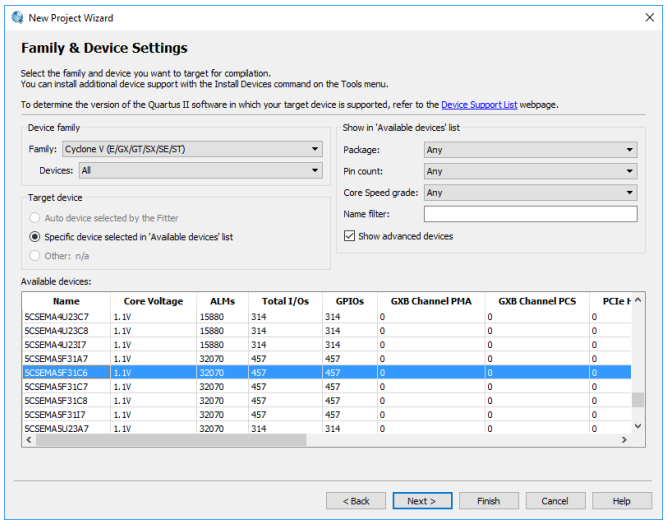
Start by creating a new project in Quartus. I'm using the F: drive in the example below, you use whatever works for you

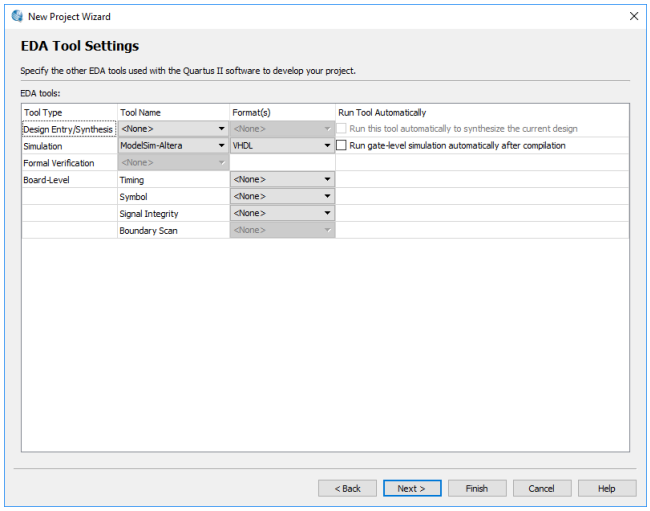


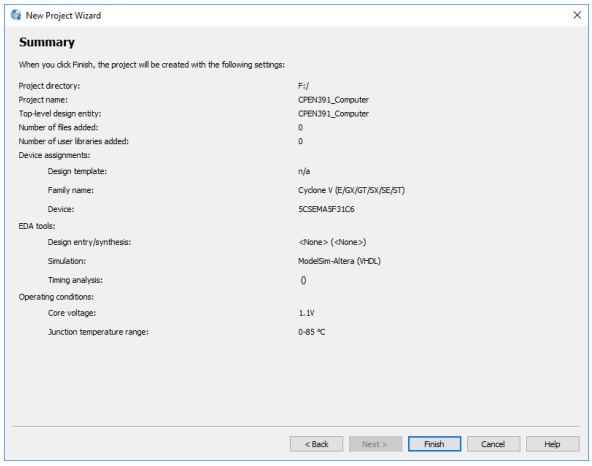




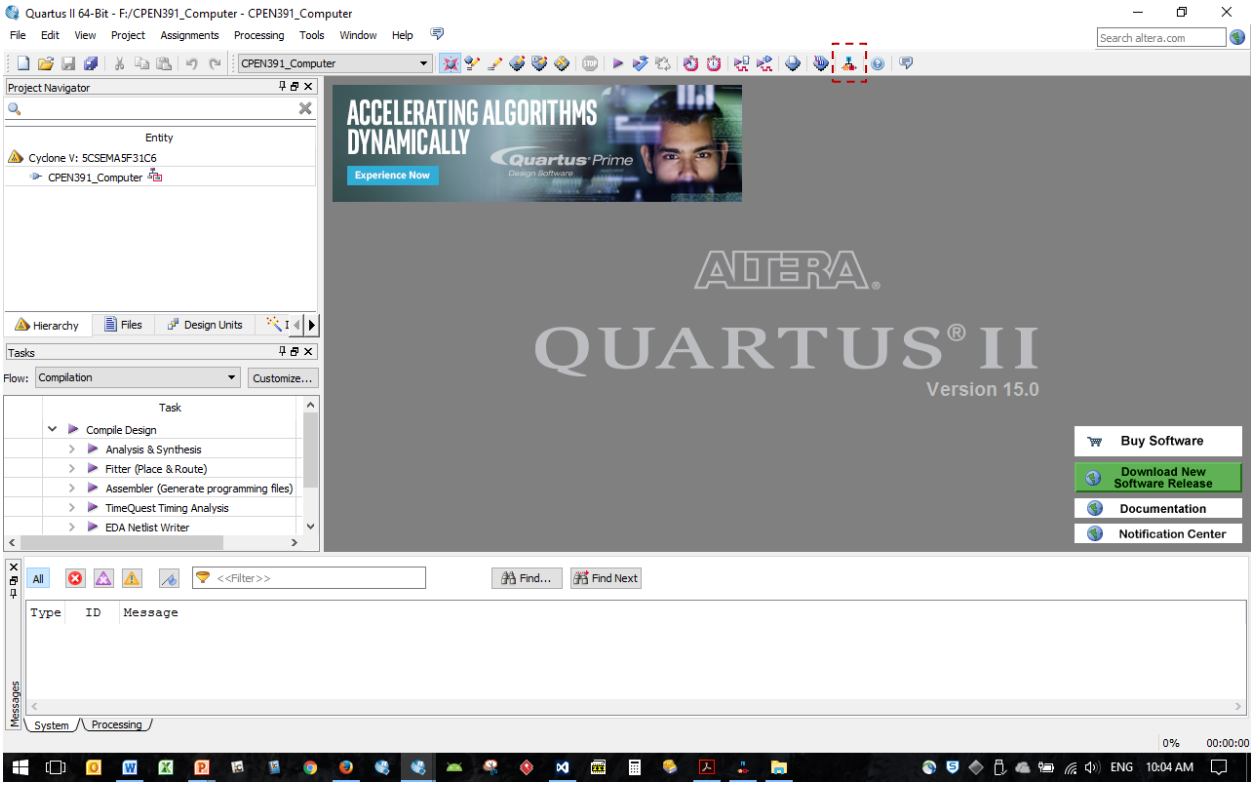
Select device: **5CSEMA5F31C6**



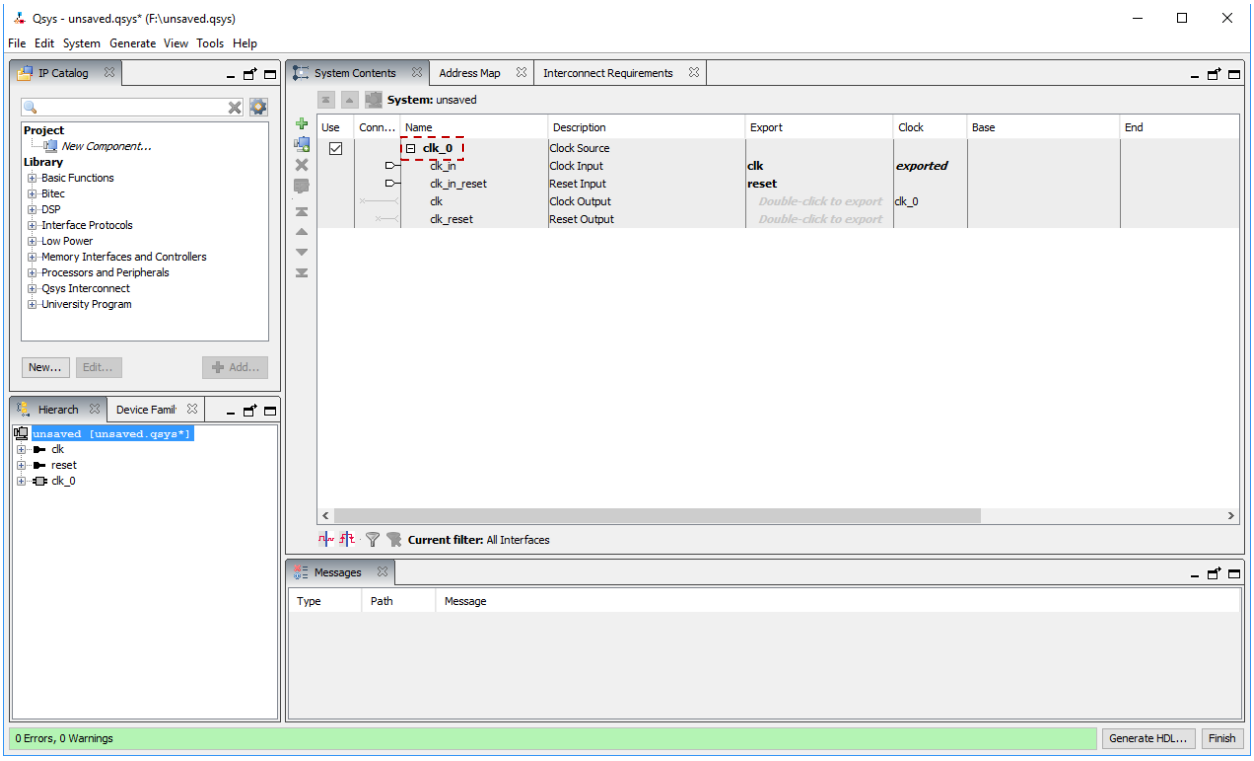




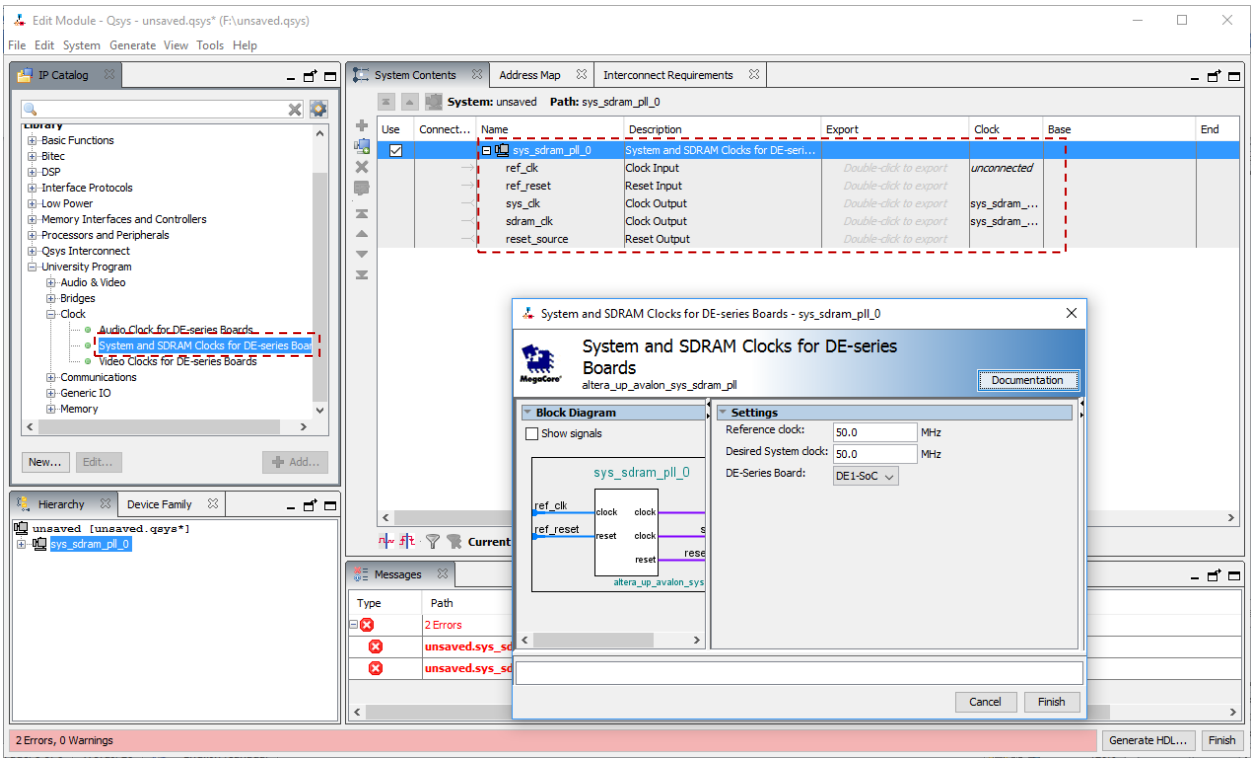
Click **Finish**. Now click on **Qsys** tool on the toolbar highlighted below.



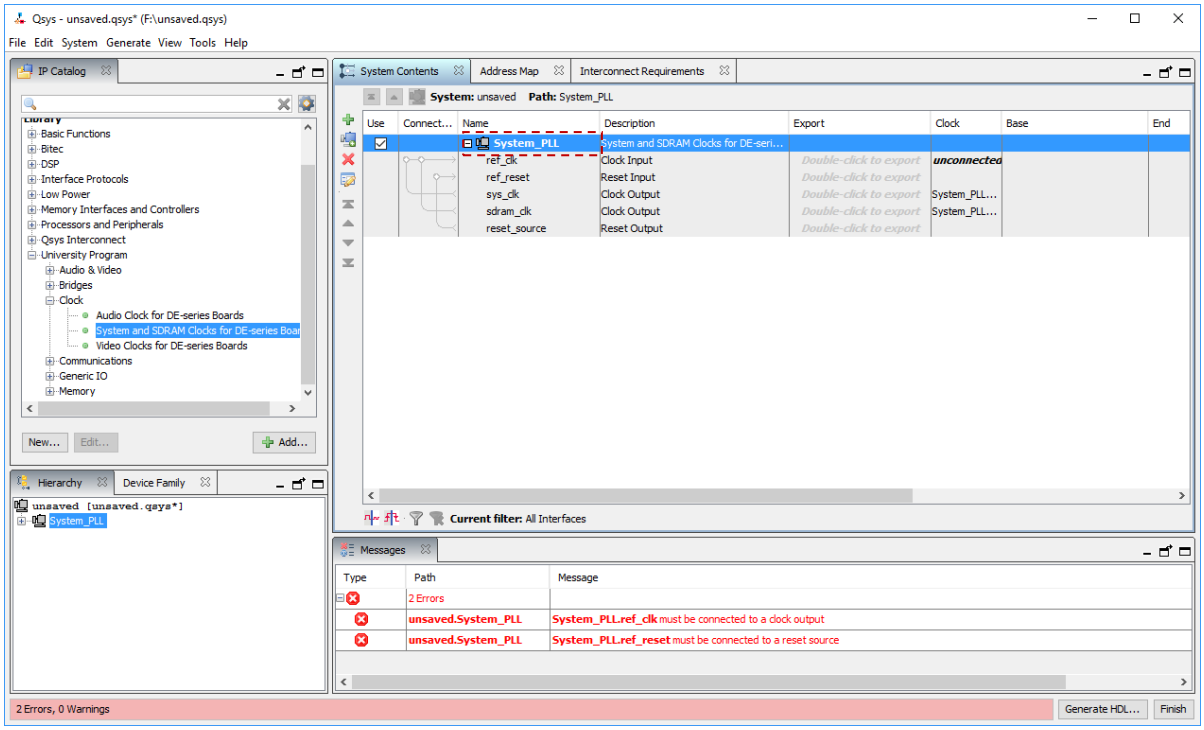
Qsys starts and display this



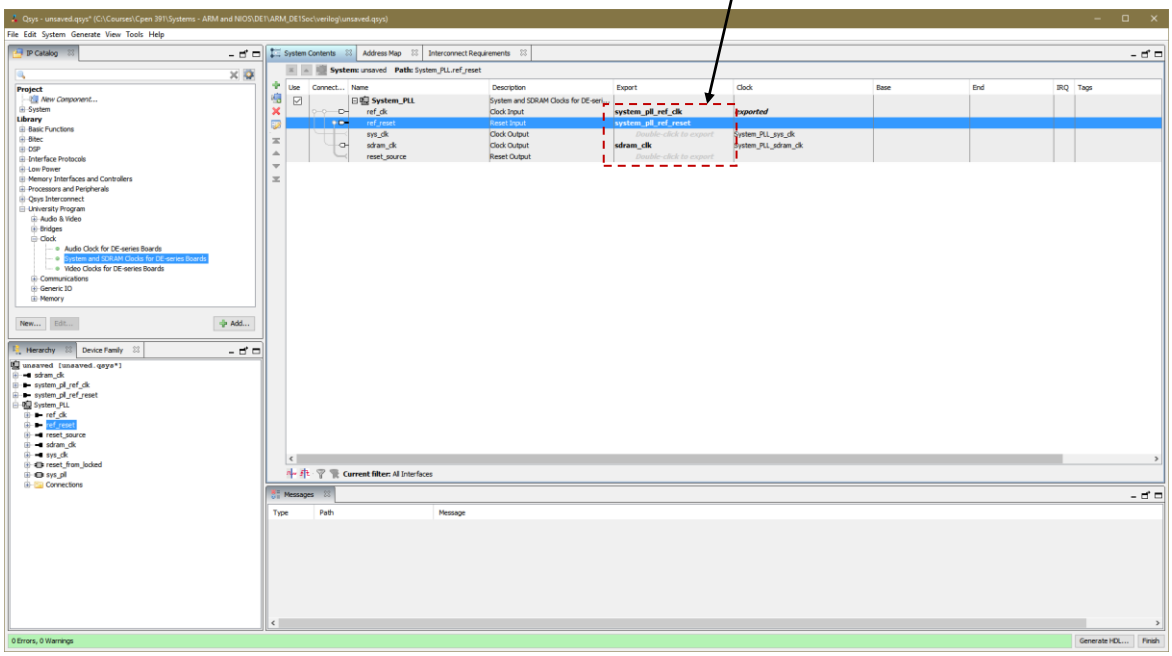
Delete **clk_0** component above and **double-click System and sdram clocks for DE-series board** from the **library->university program->clock** tab in the IP Catalog (see below) to add the new component to our design. A small dialog box will open as shown, keep the default settings shown and click finish.



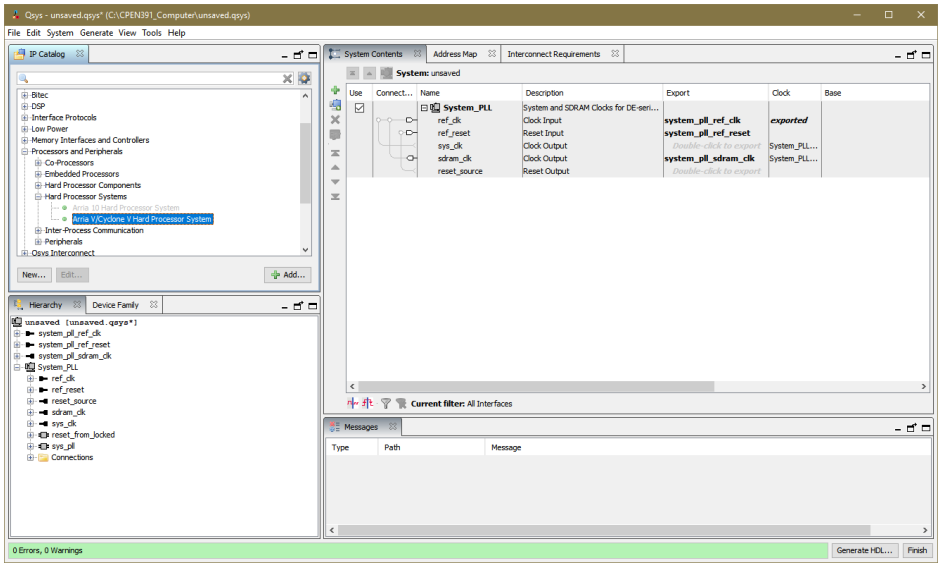
Ignore the Red warnings/errors in the Message Tab at the bottom these will disappear when we complete the design and resolve any conflicts. For now, rename the **clock** to **System_PLL** as shown below by clicking on it.



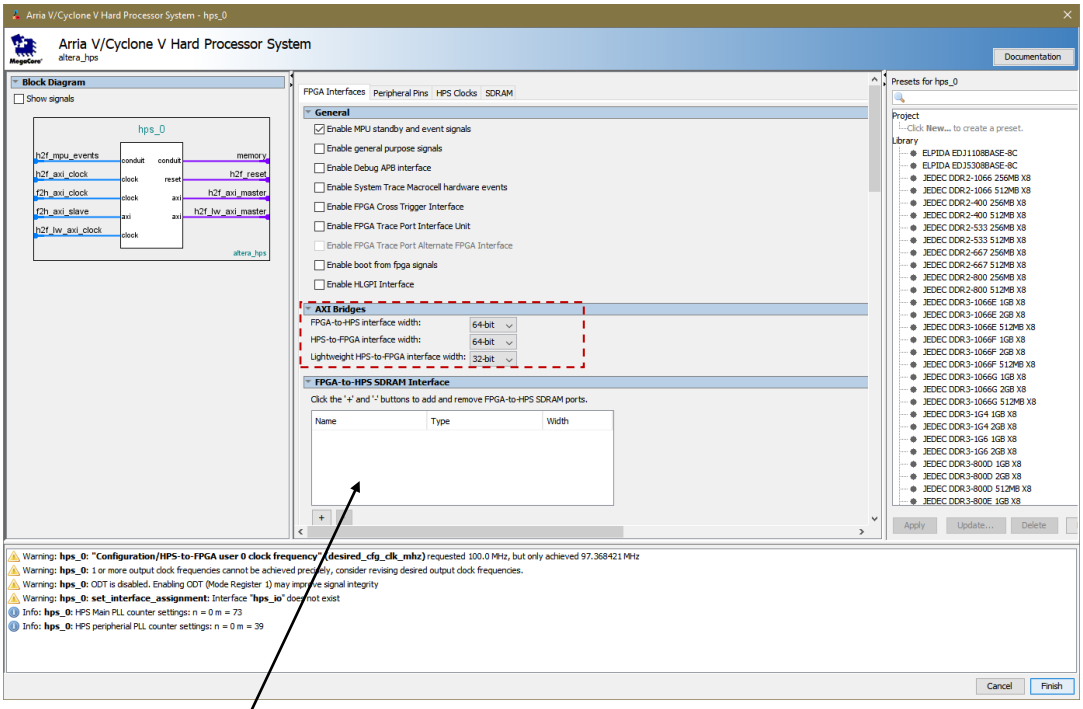
Export the 3 signals highlighted below by clicking the **export settings column** as shown below. Make sure they use the same names shown here. This action allows the clocks and other signals to be brought out for connection to external devices; in particular the **sdram_clk** signal is important.



Now add the **hard processor system,** i.e. the Dual Core ARM processor as shown below.

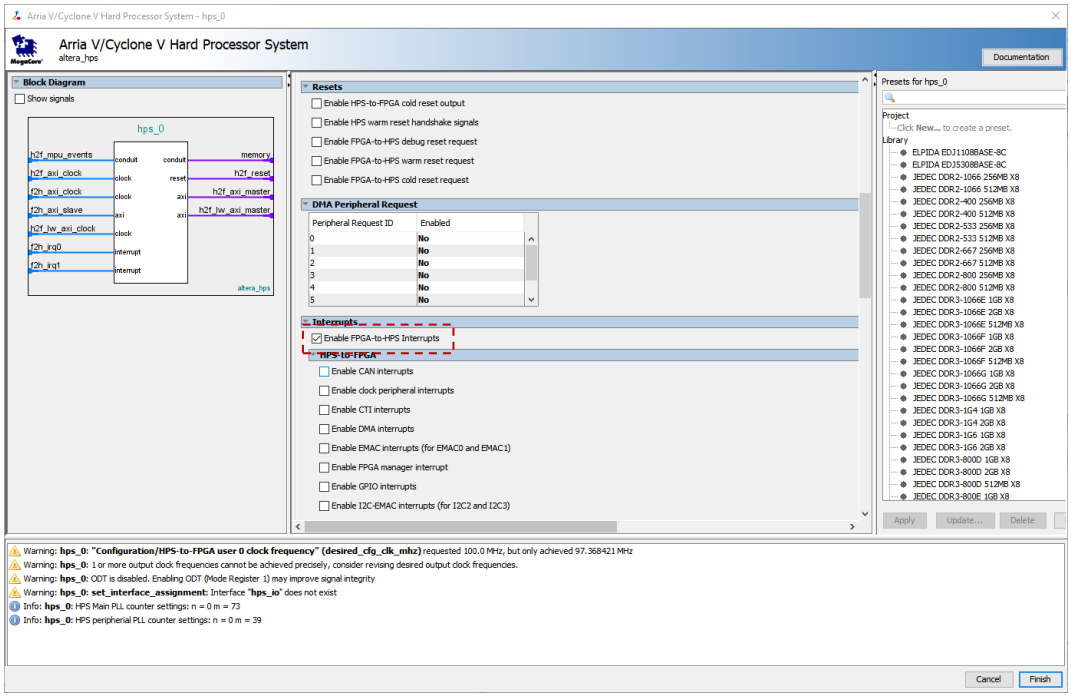


This complex dialog box pops up where we customise and make settings for the processor. In the FPGA interface settings tab **tick/untick** these options as shown below. You can see the **AXI Bridges** shown with the 3 bridges discussed earlier, including the 32 bit **Lightweight HPS-to-FPGA bridge.**

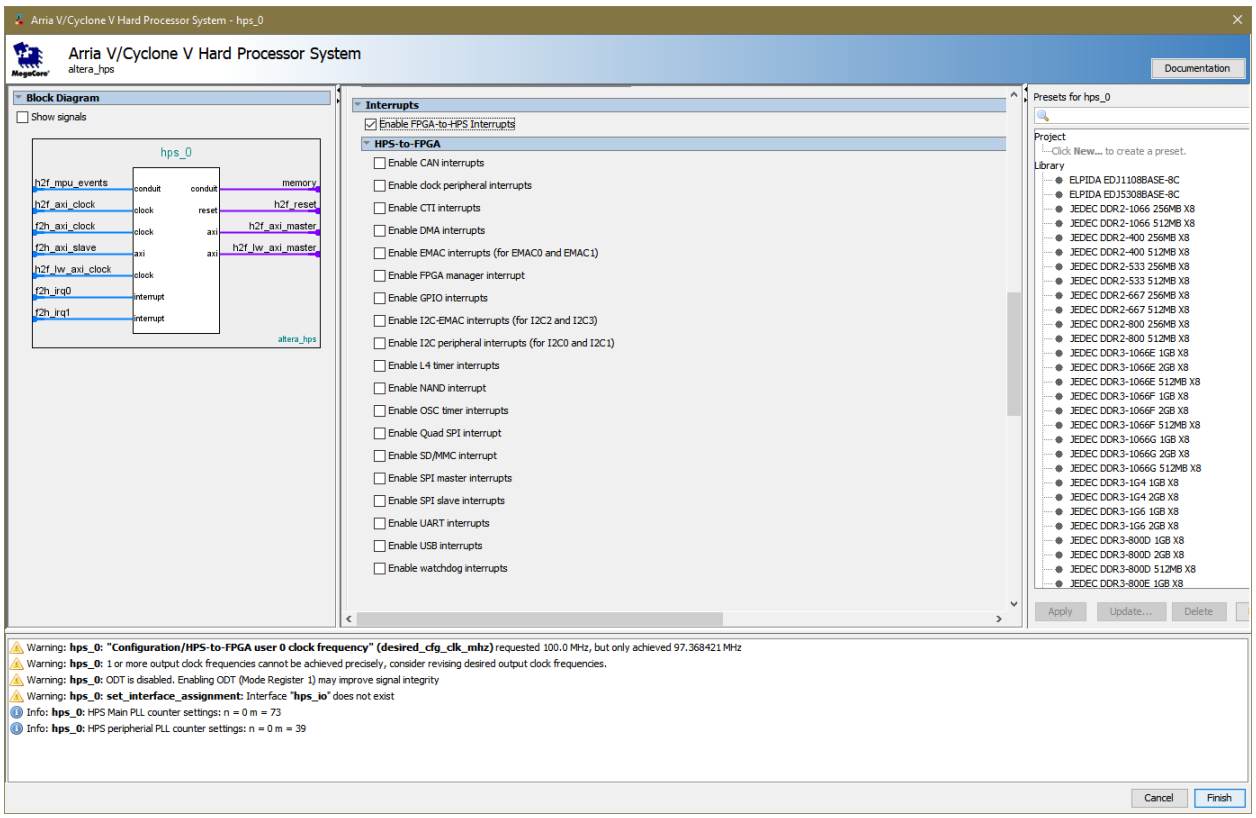


Remove any FPGA-to-HPS SDRAM interfaces from this box by clicking them and the '-' symbol.

Now scroll down and copy the settings **ticked/unticked** here.

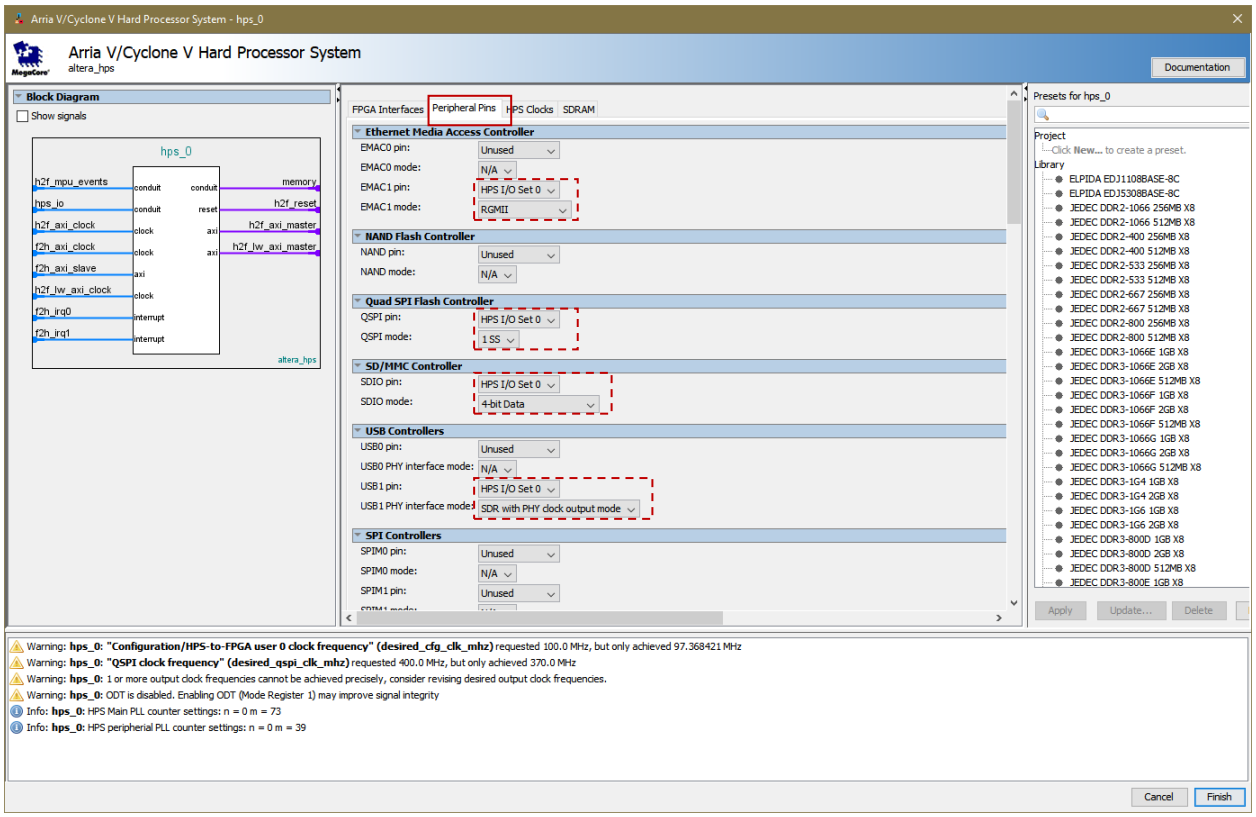


Scroll down further and apply these settings. In this project we are not going to use any of the hard wired HPS components (*that would be too easy*), we are going to design some of our own.



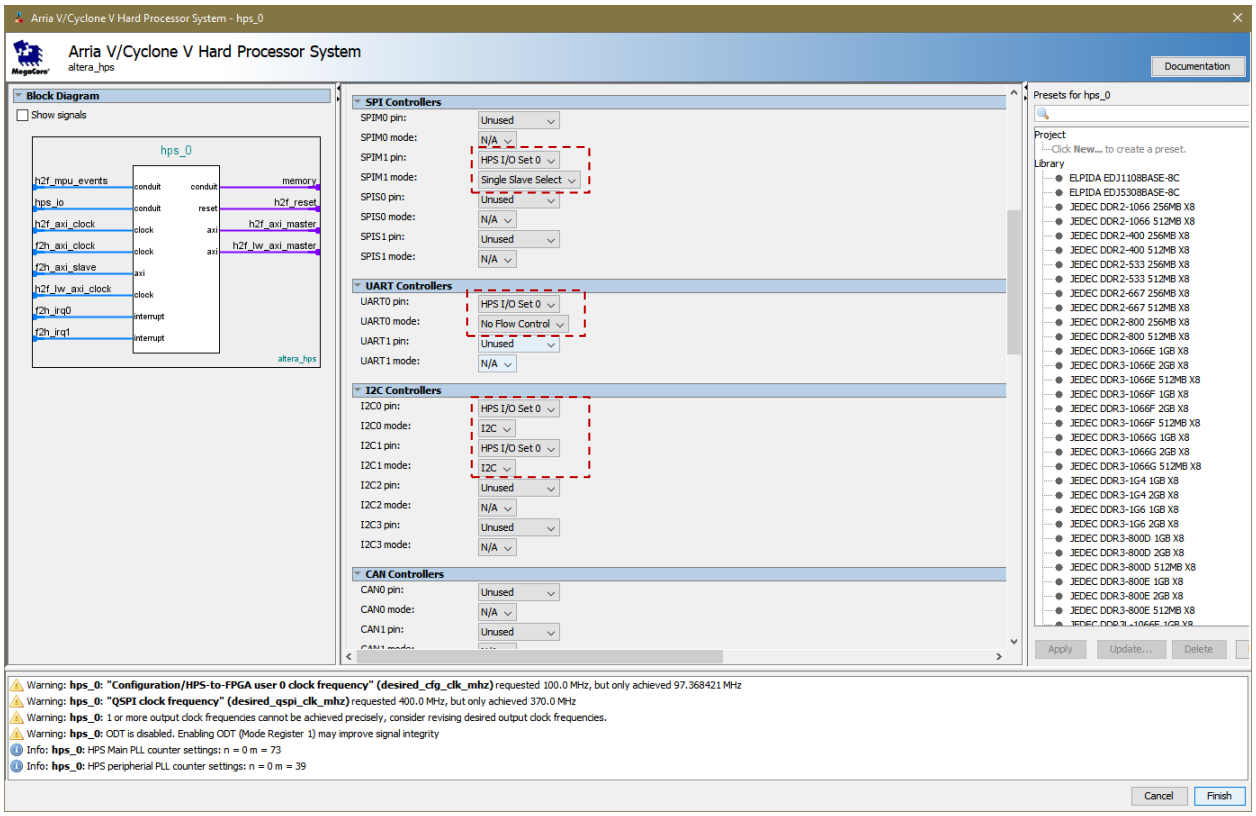
In the **peripheral pins** tabs, make these settings. Most of the settings here are to allow the ARM based IO devices to be **enabled** or **disabled** and **routed** through a **multiplexer** to physical pins on the chip. The reason for the multiplexer is simply that the Cyclone V device on the DE1-Soc does not have enough pins to allow all the built-in ARM devices to be brought out, so some sacrifices have to be made as to which ones we use. In this 391 project course, we will be using very few of them so they are set to **Unused**. This does not increase of decrease the size of the design, since all HPS peripheral/IO devices are already hard wired inside the device, so the only choice is whether to allow them to be used outside the chip.

For our purposes, the most important are the **SD/MMC controller** which connects to the SD-Card reader on the L.H.S of the DE1 board and the **Ethernet Controller** and the **Serial and USB Ports**. We can put a file system on the SD-Card and later (in exercise 1.13) we will boot a version of LINUX from it so we will enable it for use later. The Ethernet connection will be used as a **download** and **debug tool** for our software development. The serial port allows us to talk to the board. USB could be used for a USB file system. **I2C** interfaces could be brought out also. The settings below are important and configure the Cyclone 5 chip to connect to the correct Pins as wired to devices via PCB tracks on the DE1 board. Other manufacturers that produce different boards based on the Cyclone V chip might use different MUX settings to connect the IO devices to different pins.

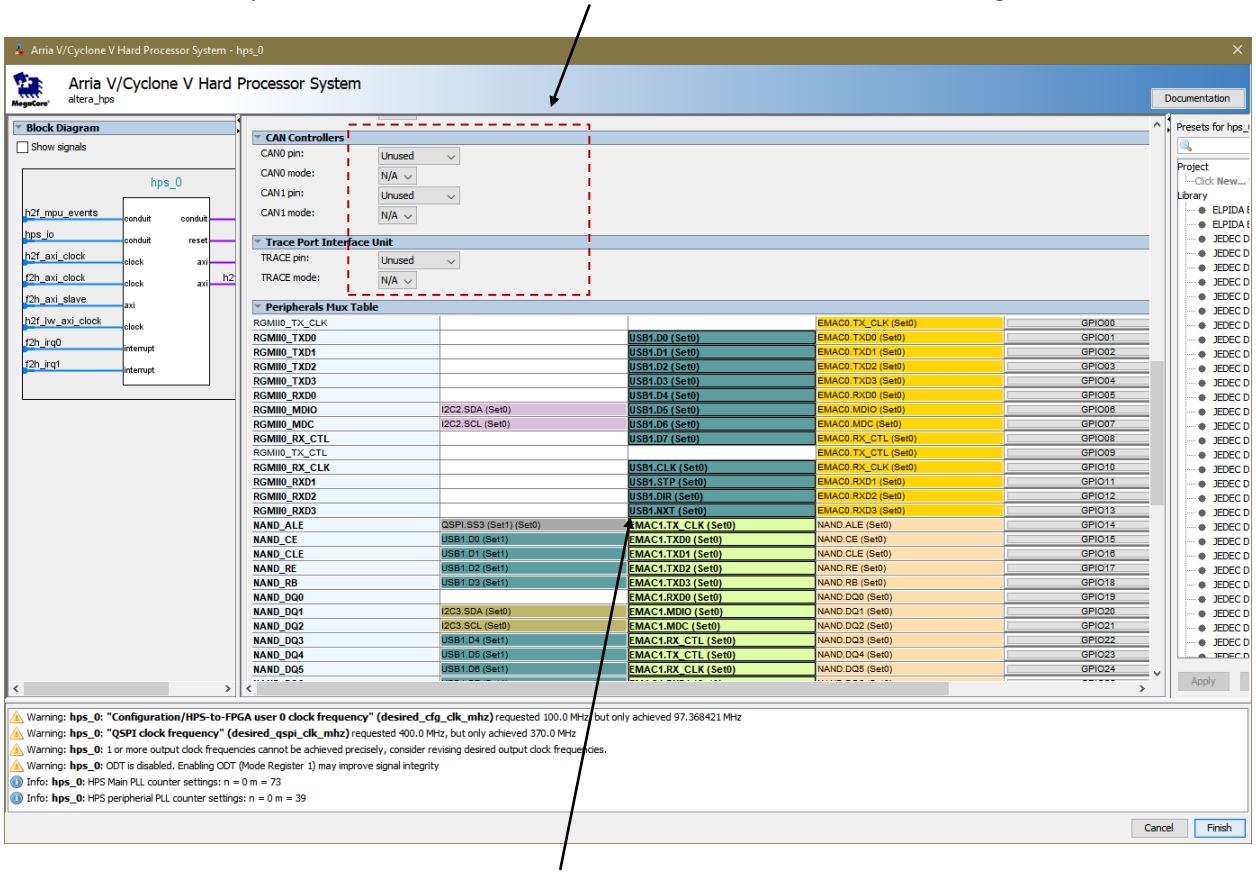


Scroll down further

We also want to enable 1 of the two built in **UART** channel to connect to the top right serial port on the DE1 board. This will allow our ARM cores to communicate through that serial port to the outside world.

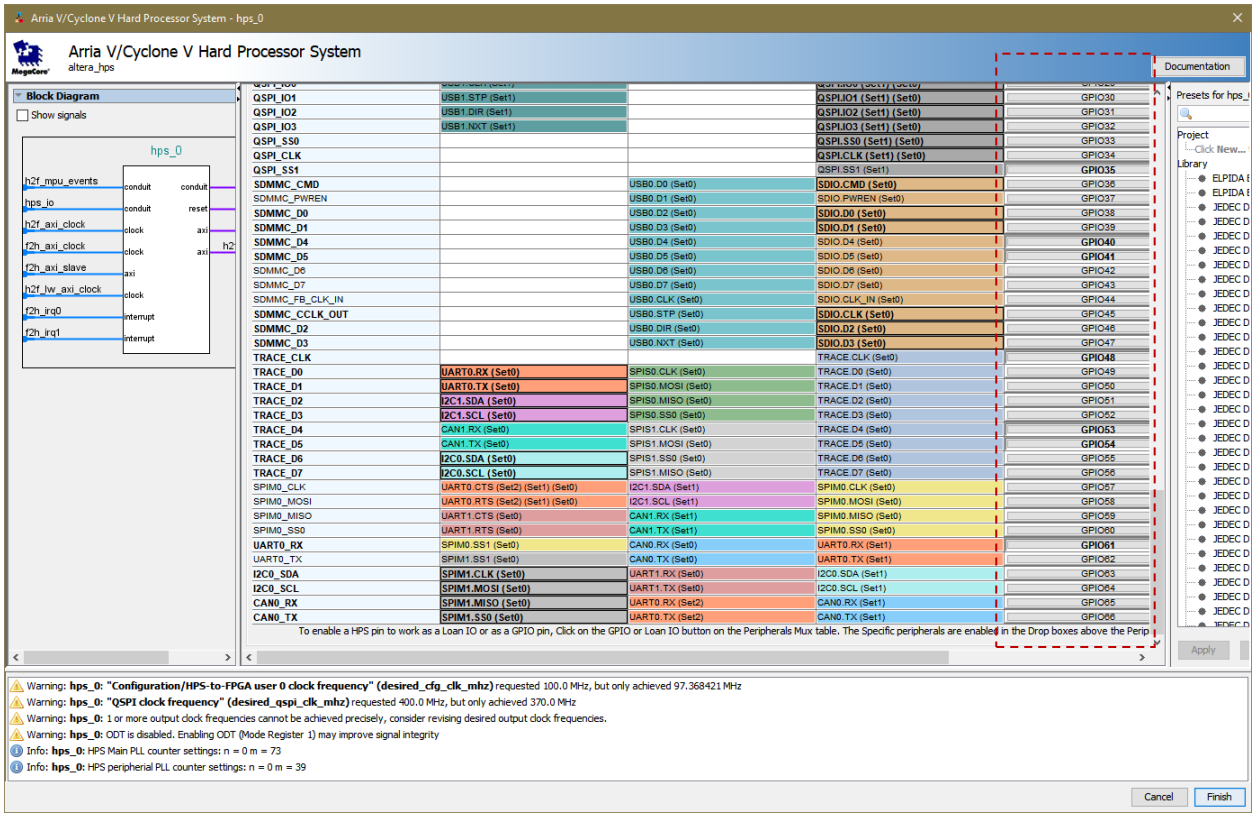


Scroll down and complete the **CAN Controllers** and **Trace Port Interface Unit** settings shown below.



The complicated **Peripheral Mux Table** settings above allow us to bring HPS peripherals such as USB, Ethernet etc. out to "**virtual GPIO**" pins inside the Cyclone V chip. You'll notice above that **USB1** is surrounded by **bolds lines**, which indicates that **USB1** port on the HPS has been Mux'ed onto **GPIO Pins 1-8, 11-13** (rather than **Ethernet0** or **I2C2** – i.e. two other devices which *could* have been Mux'ed to those same pins). We don't get any choices about these as they need to agree with the physical connections made on the PCB for the DE1 so just accept them and don't change any of them.

The only other things we could (*optionally*) do here are to click on the GPIO Pins **9, 35, 40, 41, 48, 53, 54, 61** which allow us to connect unused IO pins (*things for which we have not connected any HPS peripherals such as Ethernet, sdcard, usb etc*) to real Pins on the Cyclone V FPGA. A general purpose parallel IO port on the HPS could then drive these "spare" pins to provide simple single bit IO. This is shown below – go ahead and click the GPIO buttons **9, 35, 40, 41, 48, 53, 54, 61**.



On the DE1 board for example, we could connect these GPIO pins to things like LEDS and Push Buttons, in fact some of them have been (*see illustration Page 1*). You don't have to bring these GPIO pins out to real pins, in fact you could turn them all off.

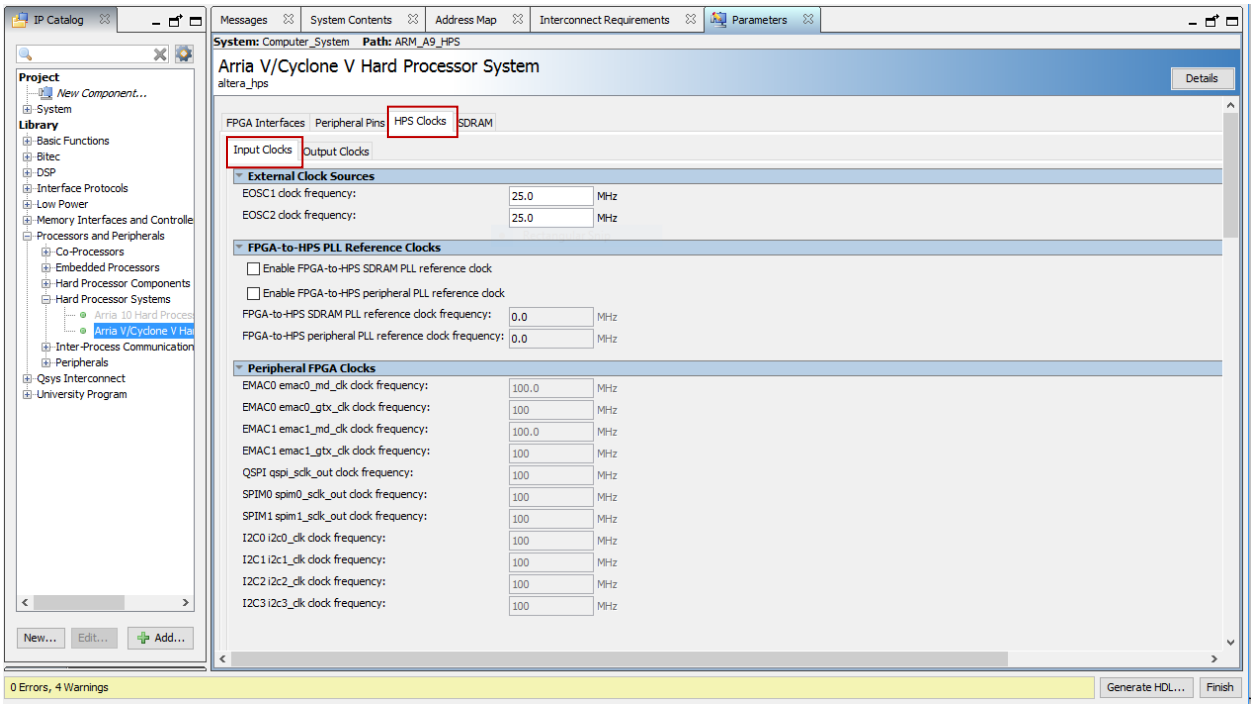
One further option is that you could disconnect the unused pins from the HPS side of the Cyclone V altogether and instead connect them to the FPGA side of the Cyclone V (*by clicking the appropriate LoanIOxx button – these pins are to the right of the GPIO pins but are not visible in the image shown above – scroll across to see them*) and use that associated real Cyclone V pin for connecting to your own FPGA side logic.

In essence, we can "mux" onto these pins in 3 different ways, we can connect

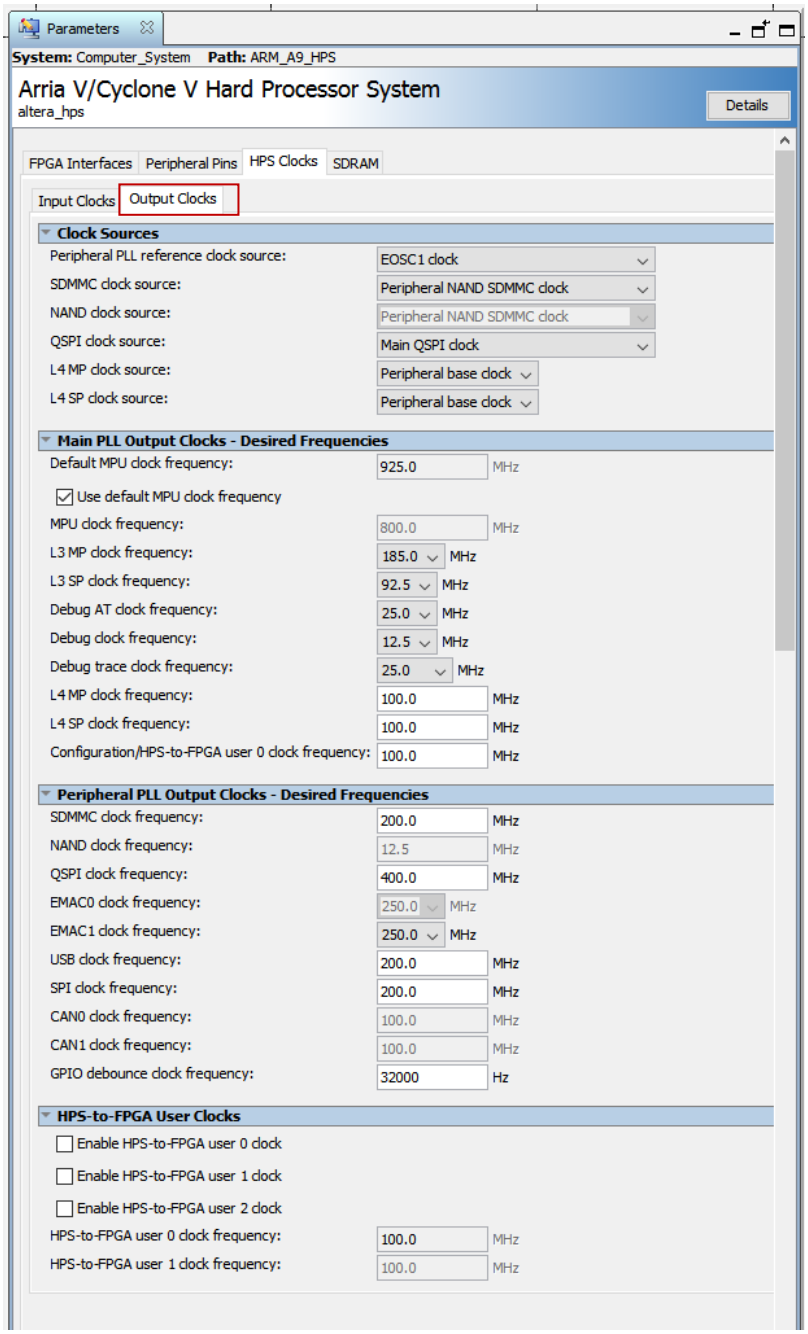
- ARM side complex peripherals such as Ethernet, USB, Flash etc
- ARM side parallel port pins
- FPGA side logic connection

Just accept the overview of what we are doing here. You can read more about it in the **DE1 User manual Chapter 6** and also in the **Cyclone V data sheet** (all 3800 pages of it – if you are feeling brave !!!) and also the **Cyclone V HPS user Guide**.

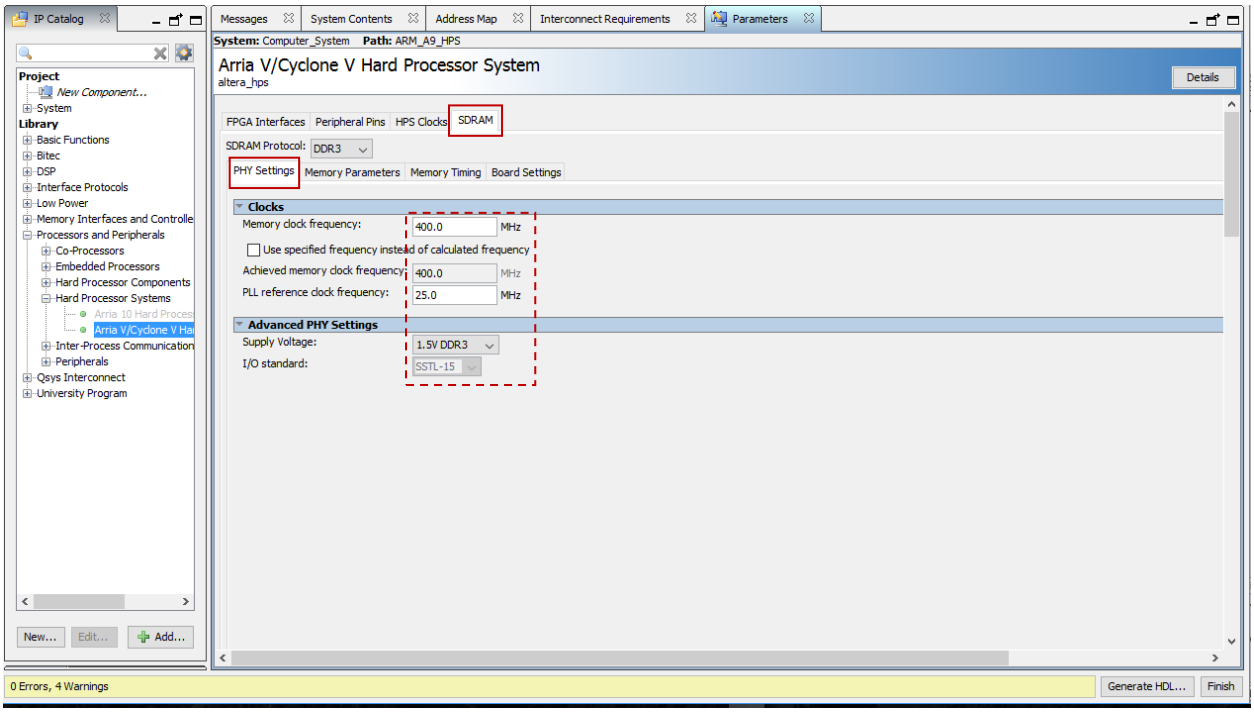
Under the **HPS Clocks Tabs -> Input Clocks** make these settings



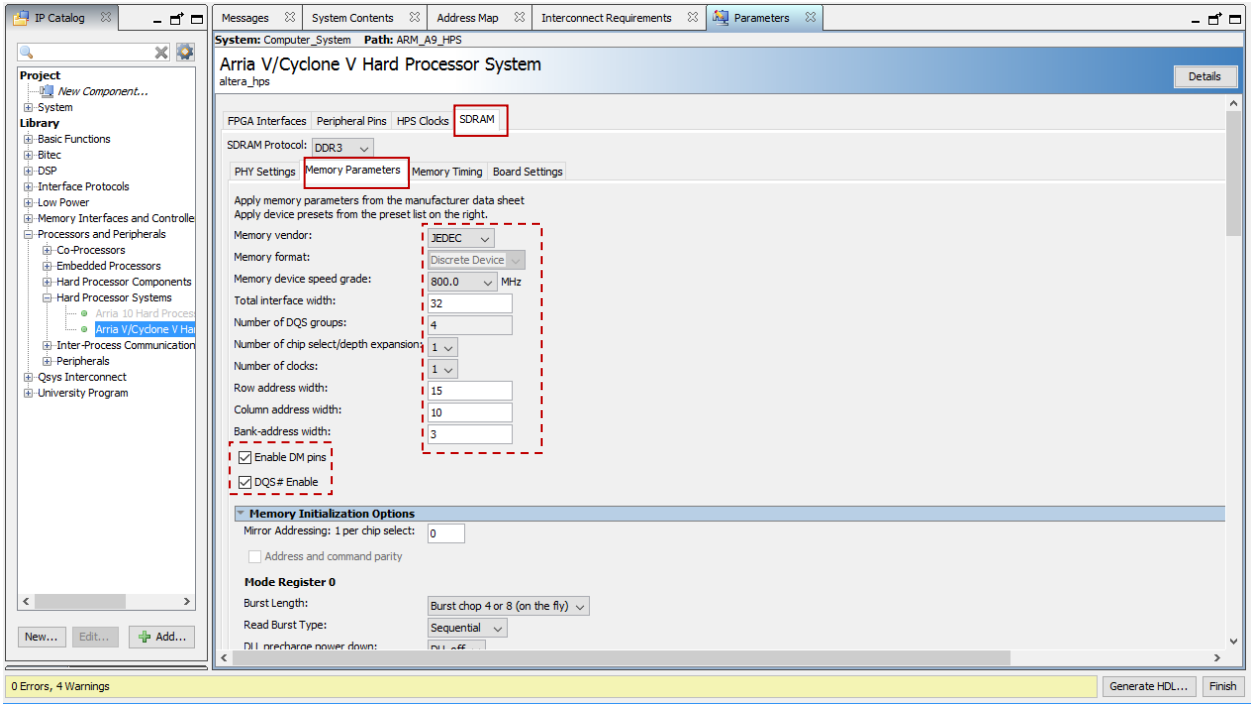
Under the **HPS Clocks Tabs -> Output Clocks**, check these default settings are correct



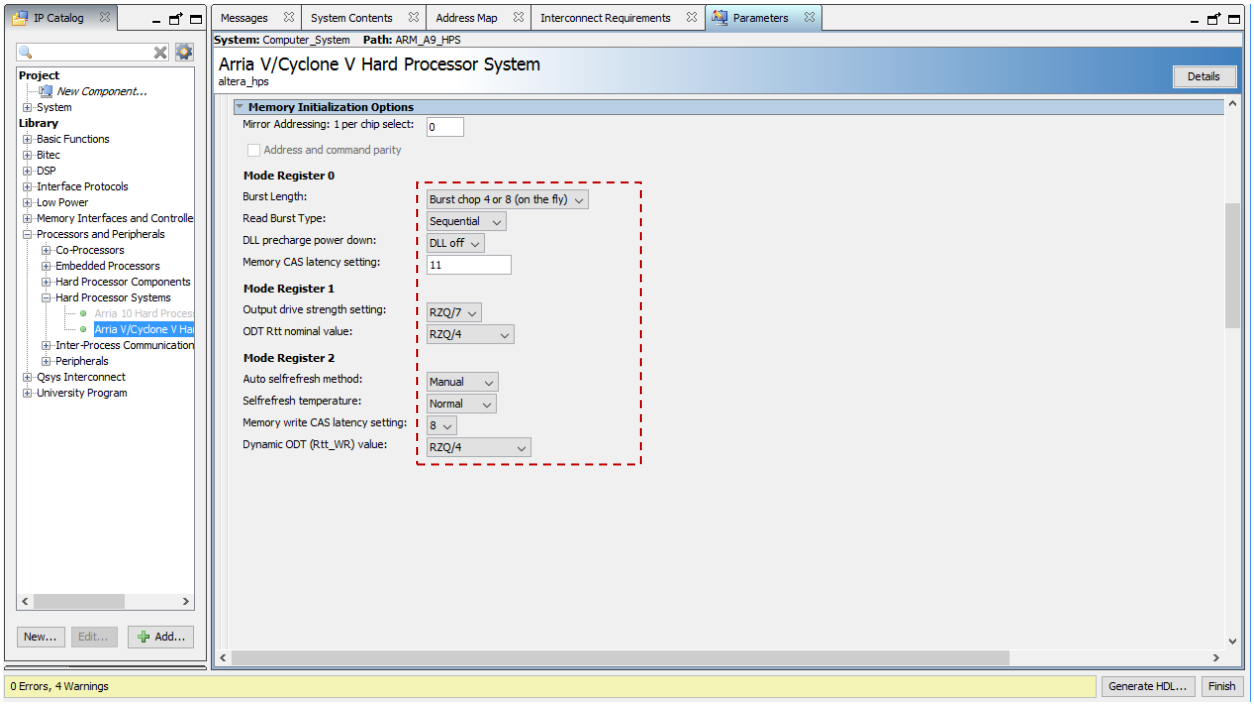
Under the **SDRam Tab->Physical Settings** you will HAVE to make these changes to settings. Again these are settings for the DE1



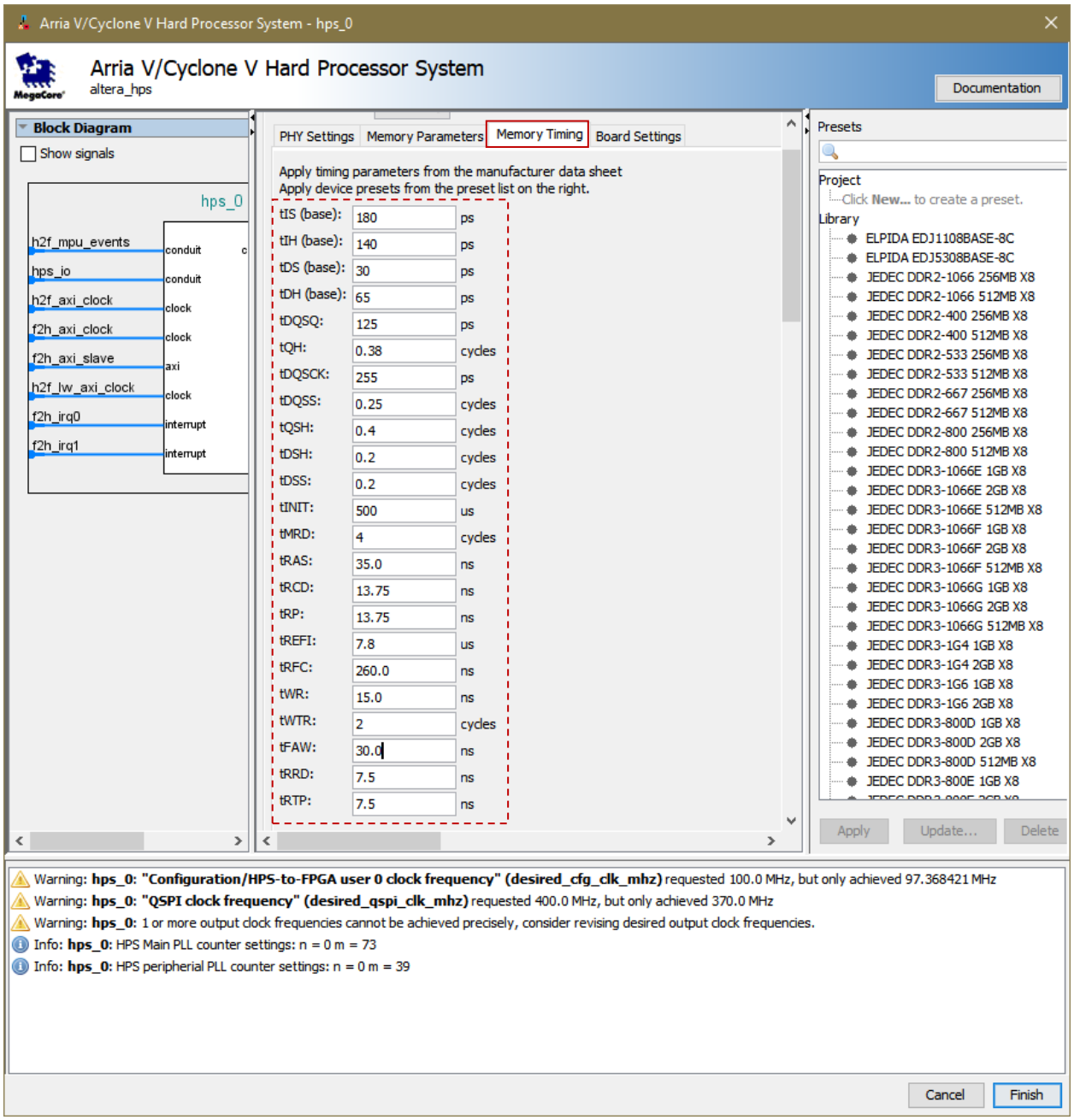
Under the **Memory Parameters** sub-tab make these settings



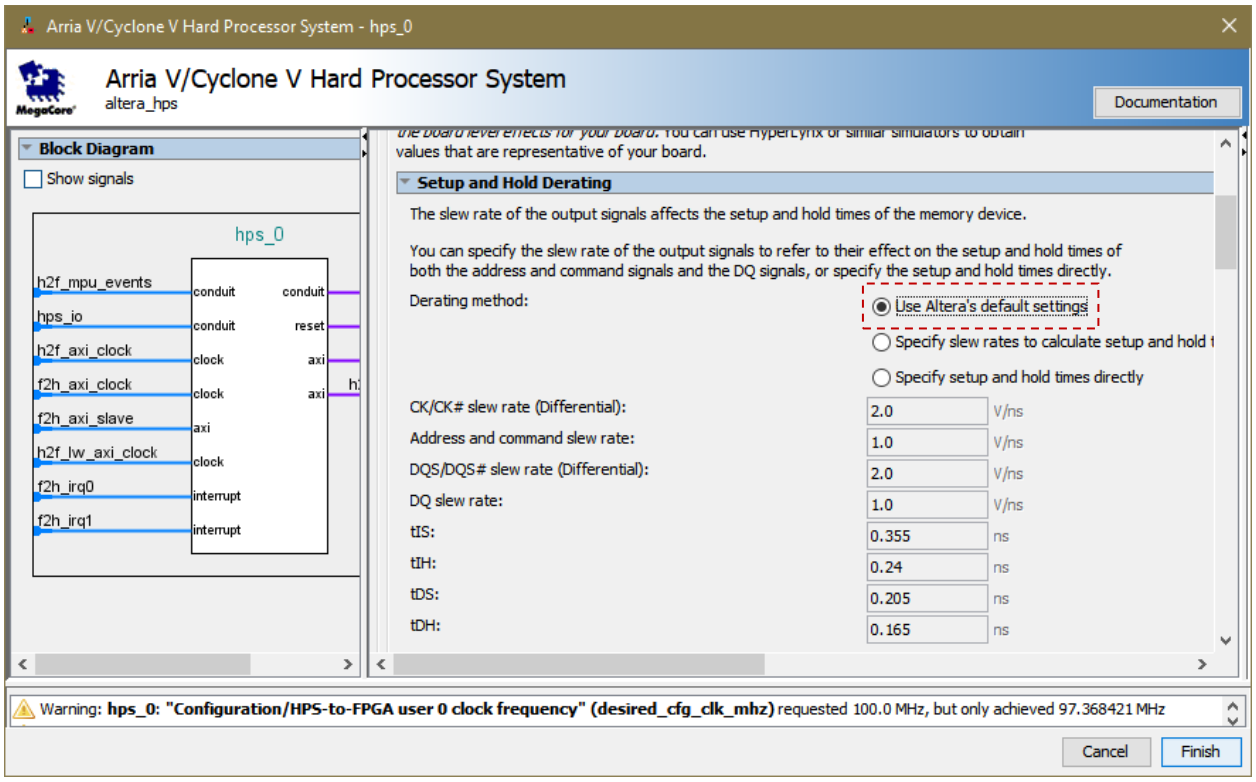
Scroll down further and make these settings



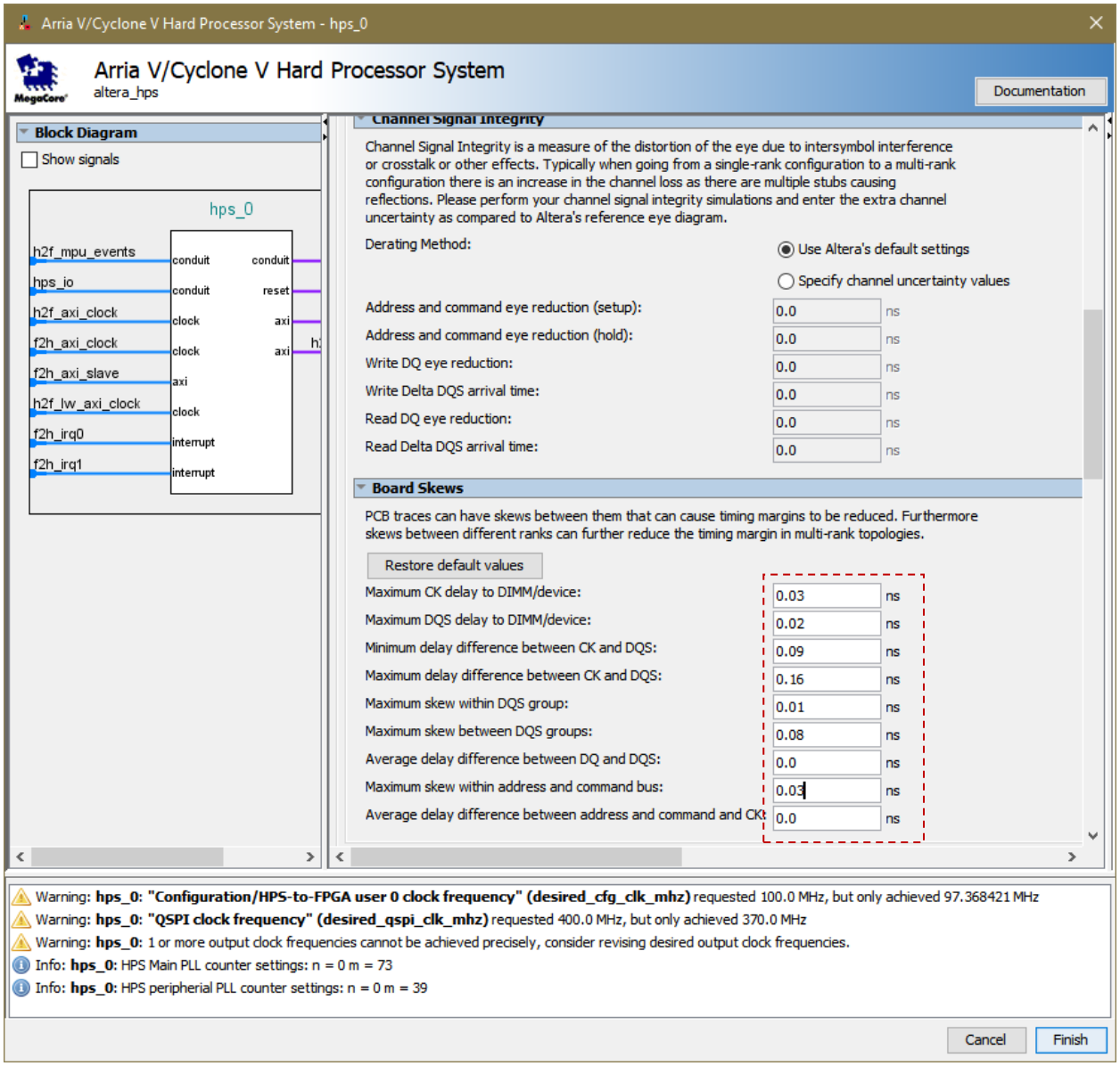
Under the **Memory Timing** Tab make these settings. These timings are specific to the Memory chips used on the DE1 and take account of the timing Skew on the PCB, i.e. they are measured and computed and the values used to account for critical timings related to the Dram memory.


Arria V/Cyclone V Hard Processor System - hps_0
Arria V/Cyclone V Hard Processor System
altera_hps
Documentation
Block Diagram
Show signals
hps_0
h2f_mpu_events
hps_io
h2f_axi_clock
f2h_axi_clock
f2h_axi_slave
h2f_lw_axi_clock
f2h_irq0
f2h_irq1
conduit
conduit
clock
clock
axi
clock
interrupt
interrupt
PHY Settings
Memory Parameters
Memory Timing
Board Settings
Apply timing parameters from the manufacturer data sheet
Apply device presets from the preset list on the right.
tIS (base): 180 ps
tIH (base): 140 ps
tDS (base): 30 ps
tDH (base): 65 ps
tDQSQ: 125 ps
tQH: 0.38 cycles
tDQSCK: 255 ps
tDQSS: 0.25 cycles
tQSH: 0.4 cycles
tDSH: 0.2 cycles
tDSS: 0.2 cycles
tINIT: 500 us
tMRD: 4 cycles
tRAS: 35.0 ns
tRCD: 13.75 ns
tRP: 13.75 ns
tREFI: 7.8 us
tRFC: 260.0 ns
tWR: 15.0 ns
tWTR: 2 cycles
tFAW: 30.0 ns
tRRD: 7.5 ns
tRTP: 7.5 ns
Presets
Project
Click New... to create a preset.
Library
ELPIDA EDJ1108BASE-8C
ELPIDA EDJ5308BASE-8C
JEDEC DDR2-1066 256MB X8
JEDEC DDR2-1066 512MB X8
JEDEC DDR2-400 256MB X8
JEDEC DDR2-400 512MB X8
JEDEC DDR2-533 256MB X8
JEDEC DDR2-533 512MB X8
JEDEC DDR2-667 256MB X8
JEDEC DDR2-667 512MB X8
JEDEC DDR2-800 256MB X8
JEDEC DDR2-800 512MB X8
JEDEC DDR3-1066E 1GB X8
JEDEC DDR3-1066E 2GB X8
JEDEC DDR3-1066E 512MB X8
JEDEC DDR3-1066F 1GB X8
JEDEC DDR3-1066F 2GB X8
JEDEC DDR3-1066F 512MB X8
JEDEC DDR3-1066G 1GB X8
JEDEC DDR3-1066G 2GB X8
JEDEC DDR3-1066G 512MB X8
JEDEC DDR3-1G4 1GB X8
JEDEC DDR3-1G4 2GB X8
JEDEC DDR3-1G6 1GB X8
JEDEC DDR3-1G6 2GB X8
JEDEC DDR3-800D 1GB X8
JEDEC DDR3-800D 2GB X8
JEDEC DDR3-800D 512MB X8
JEDEC DDR3-800E 1GB X8
Apply
Update...
Delete
Warning: hps_0: "Configuration/HPS-to-FPGA user 0 clock frequency" (desired_cfg_clk_mhz) requested 100.0 MHz, but only achieved 97.368421 MHz
Warning: hps_0: "QSPI clock frequency" (desired_qspi_clk_mhz) requested 400.0 MHz, but only achieved 370.0 MHz
Warning: hps_0: 1 or more output clock frequencies cannot be achieved precisely, consider revising desired output clock frequencies.
Info: hps_0: HPS Main PLL counter settings: n = 0 m = 73
Info: hps_0: HPS peripherial PLL counter settings: n = 0 m = 39
Cancel
Finish

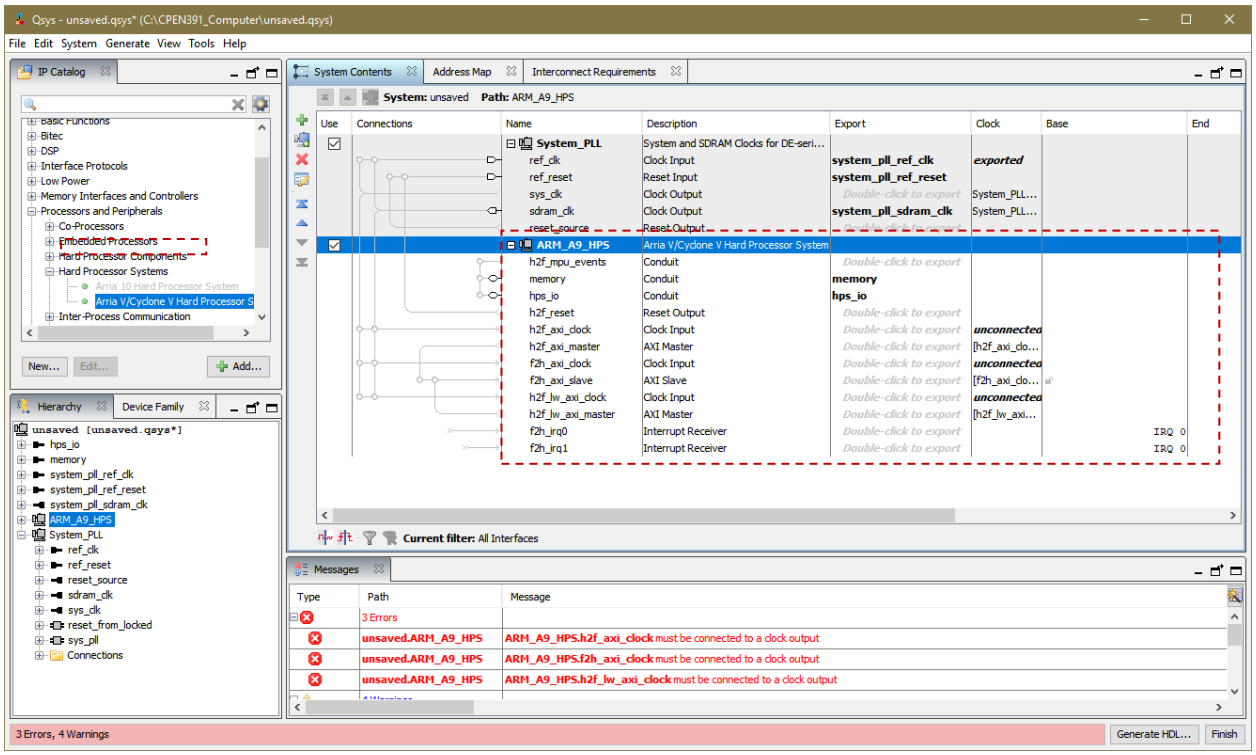
Under Board Settings Tab



Scroll down further and make these settings



Now click Finish and the ARM processor HPS_0 is added to our design. Rename it to **ARM_A9_HPS. Note you can go back and edit these changes at any time**


Qsys - unsaved.qsys* (C:\CPEN391_Computer\unsaved.qsys)
File Edit System Generate View Tools Help
IP Catalog
Basic Functions
Bitec
DSP
Interface Protocols
Low Power
Memory Interfaces and Controllers
Processors and Peripherals
Co-Processors
Embedded Processors
Hard Processor Components
Hard Processor Systems
Arria 10 Hard Processor System
Arria V/Cyclone V Hard Processor S
Inter-Process Communication
New...
Edit...
Add...
Hierarchy
Device Family
unsaved [unsaved.qsys*]
hps_io
memory
system_pll_ref_clk
system_pll_ref_reset
system_pll_sdram_clk
ARM_A9_HPS
System_PLL
ref_clk
ref_reset
reset_source
sdram_clk
sys_clk
reset_from_locked
sys_pll
Connections
System Contents
Address Map
Interconnect Requirements
System: unsaved Path: ARM_A9_HPS
Use
Connections
Name
Description
Export
Clock
Base
End
System_PLL
System and SDRAM Clocks for DE-seri...
ref_clk
Clock Input
system_pll_ref_clk
exported
ref_reset
Reset Input
system_pll_ref_reset
sys_clk
Clock Output
Double-click to export
System_PLL...
sdram_clk
Clock Output
system_pll_sdram_clk
System_PLL...
reset_source
Reset Output
ARM_A9_HPS
Arria V/Cyclone V Hard Processor System
h2f_mpu_events
Conduit
memory
hps_io
h2f_reset
Reset Output
h2f_axi_clock
Clock Input
unconnected
h2f_axi_master
AXI Master
[h2f_axi_clo...
f2h_axi_clock
f2h_axi_slave
AXI Slave
[f2h_axi_clo...
h2f_lw_axi_clock
h2f_lw_axi_master
[h2f_lw_axi...
f2h_irq0
Interrupt Receiver
IRQ 0
f2h_irq1
Current filter: All Interfaces
Messages
Type
Path
Message
3 Errors
unsaved.ARM_A9_HPS
ARM_A9_HPS.h2f_axi_clock must be connected to a clock output
ARM_A9_HPS.f2h_axi_clock must be connected to a clock output
ARM_A9_HPS.h2f_lw_axi_clock must be connected to a clock output
3 Errors, 4 Warnings
Generate HDL...
Finish

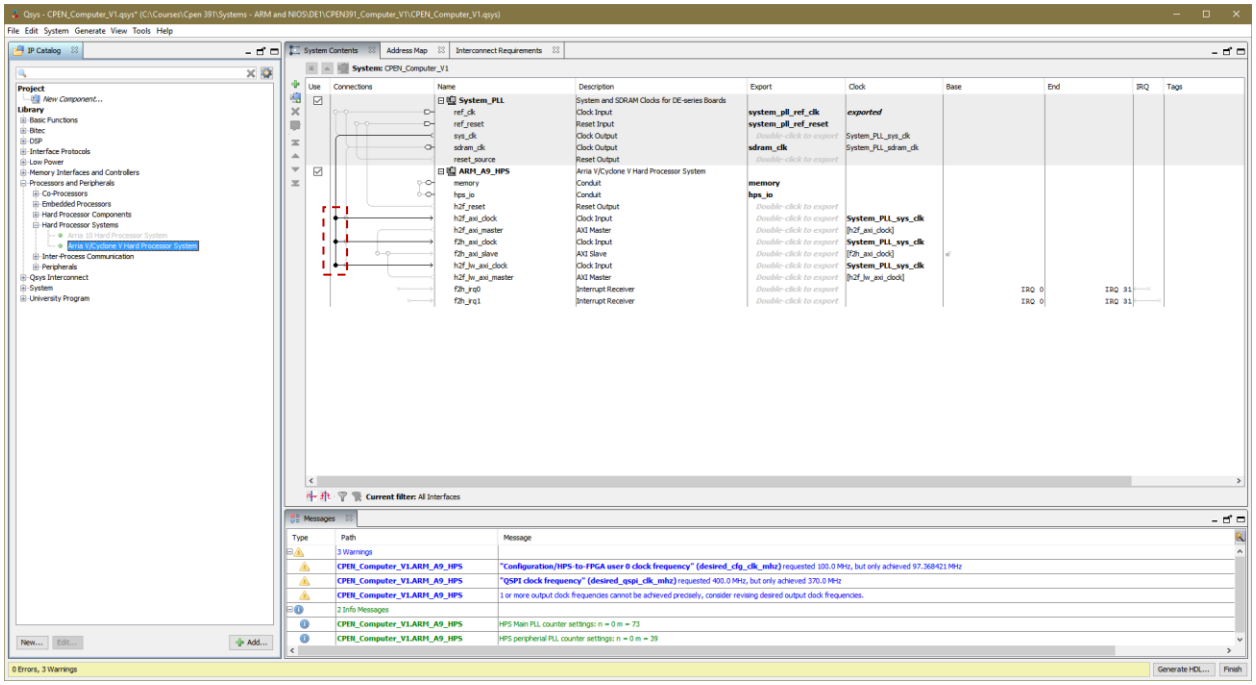
Wiring up the ARM core and other HPS devices to the clocks

Now that we have instantiated and configured the HPS/ARM cores, we need to wire them to the **sys_clock** signal coming from the **System_PLL** component we generated instantiated earlier. Wire the clock as shown below. **You can make the connections by clicking on the tiny circles that are drawn at the intersection of signal, i.e. the filled black dots shown below**. This connects a system clock to the 3 bridges we discussed earlier.

You can see the 3 bridge as part of the ARM/HPS cores in the illustration below.

- H2f refer to the HPS-to-FPGA bus bridge.
- f2h refer to the FPGA-to-HPS bus bridge.
- H2f_lw refer to the lightweight HPS-to-FPGA bus bridge

Using these bridges we can hang our custom FPGA designed cores



Adding FPGA side controllers for the devices on the DE1-Soc board

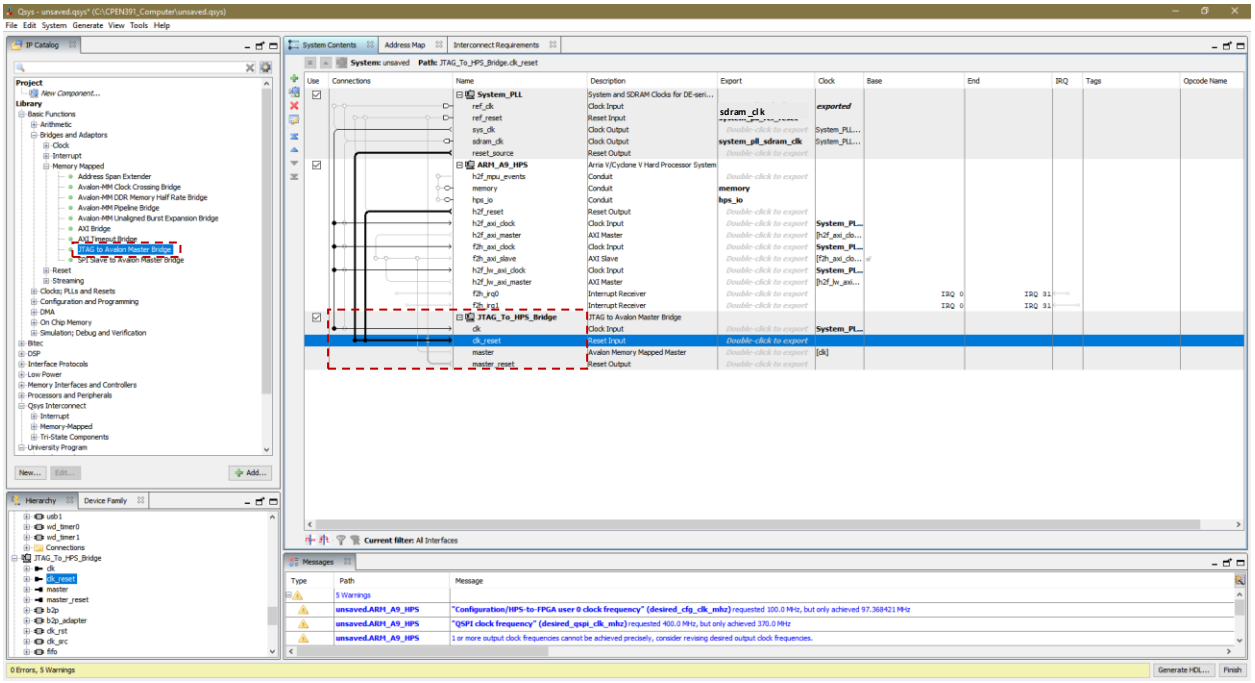
We are now going to add a bunch of controllers to control physical devices on the DE1 board, e.g. Hex displays, more dram memory, switches, Serial ports, Jtag download ports etc. These will **all** be created by Qsys on the **FPGA side** of the chip with connections made to the ARM cores via the lightweight **bridge**.

Add the JTAG Bridges

Add an instance of a **JTAG to Avalon Master Bridge**, accept the default settings from the dialog box, **rename** to **JTAG_To_HPS_Bridge** and wire up the connection as shown below.

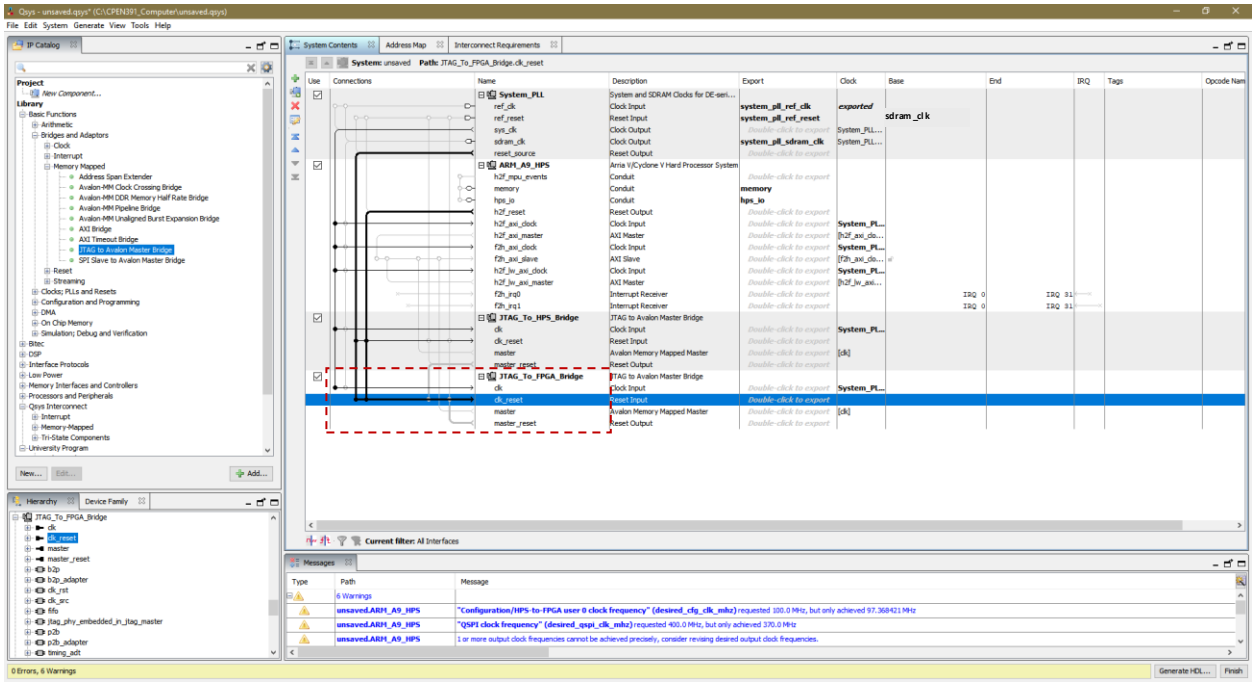
This allows Quartus and Software development environments to communicate with and control the HPS/ARM cores via the USB cable, e.g. for downloading software, resetting the computer etc.

Notice we've added a clock source to the **JTAG bridge** from the **System_PLL** and also two **reset sources**, one from the **System_PLL** and the other from the **HPS/ARM cores** themselves.



Add another instance of the same Bridge and rename it to **JTAG_To_FPGA_Bridge** and connect up as shown below. This allows Quartus to communicate with the FPGA side of the chip, ie. for capturing signals in real-time.

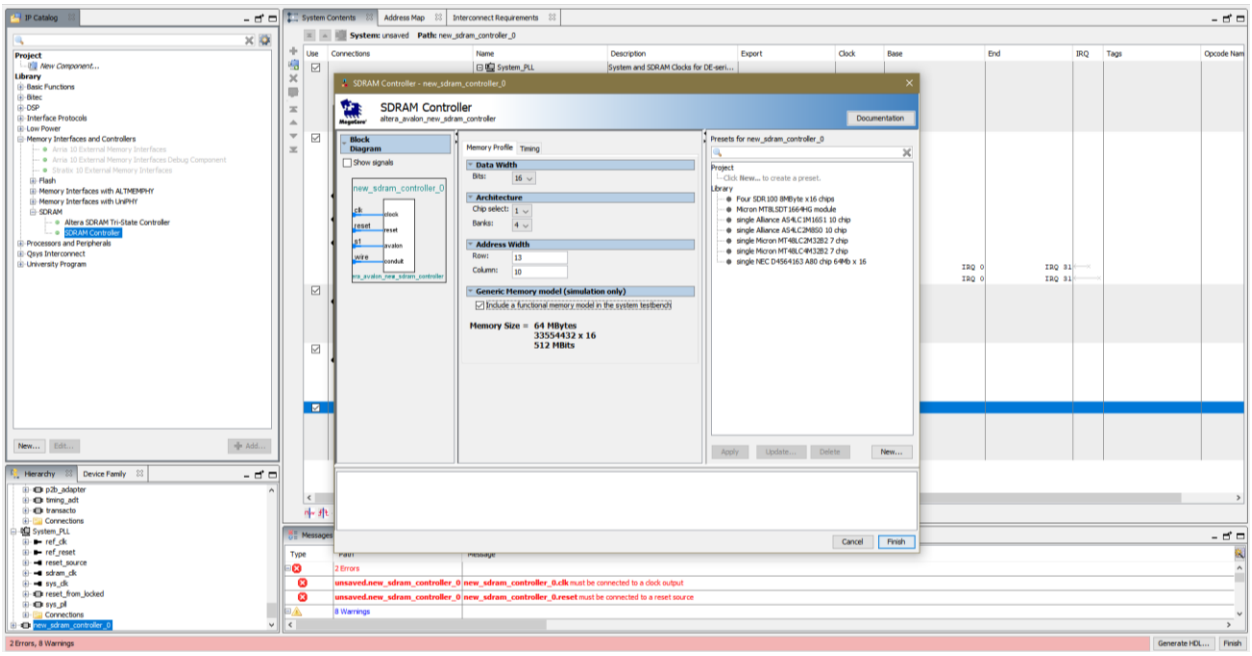
Again we've added a clock source to the **JTAG bridge** from the **System_PLL** and also two **reset sources**, one from the **System_PLL** and the other from the **HPS/ARM cores** themselves.

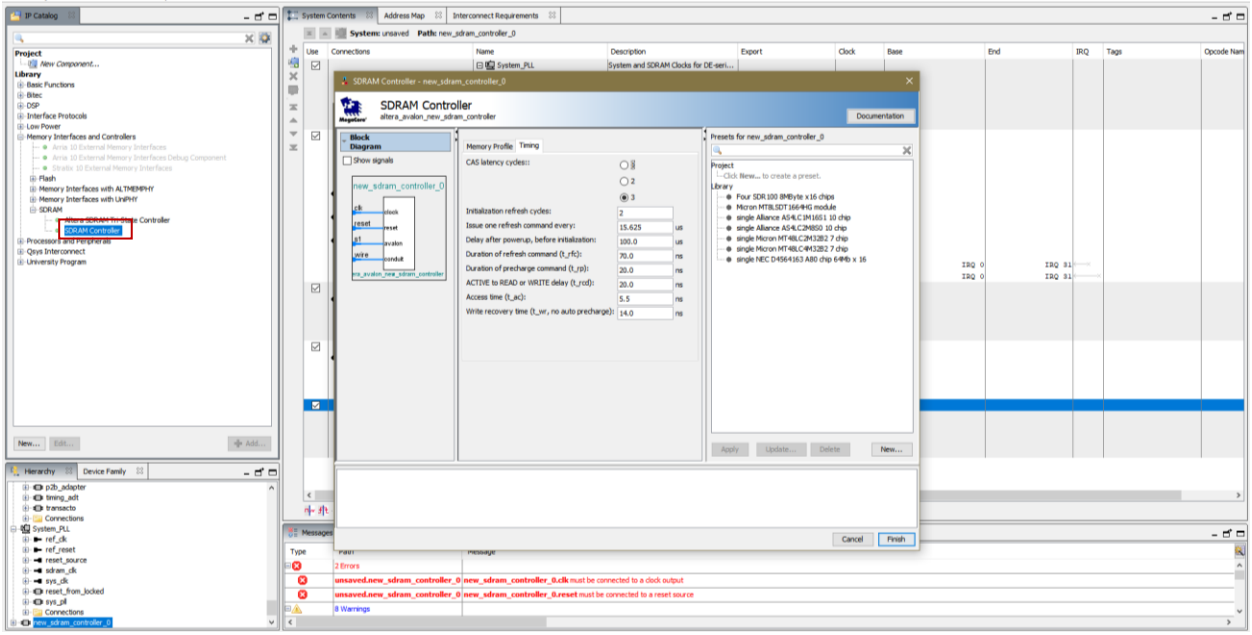


Adding an SDRAM Controller to the FPGA side

The FPGA side of the Cyclone V chip is also connected to 64Mbytes of SDRam (*independent of the DDRDram connected direction to the HPS/ARM processors – see Page 1*). We can add a controller for this memory to our system that would allow the ARM cores to access that memory in addition to its own.

Add an instance of a SDram memory controller and set up the **Memory Profile** and **Timing data** as per the next two illustrations and make the connection shown below. Note that the ARM/HPS has its own dedicated 1GByte of DDR3 memory hard wired on the board and as such cannot be added/removed from our design. The SDRam we are adding here is on the FPGA side of the chip and is a 64Mbyte device.

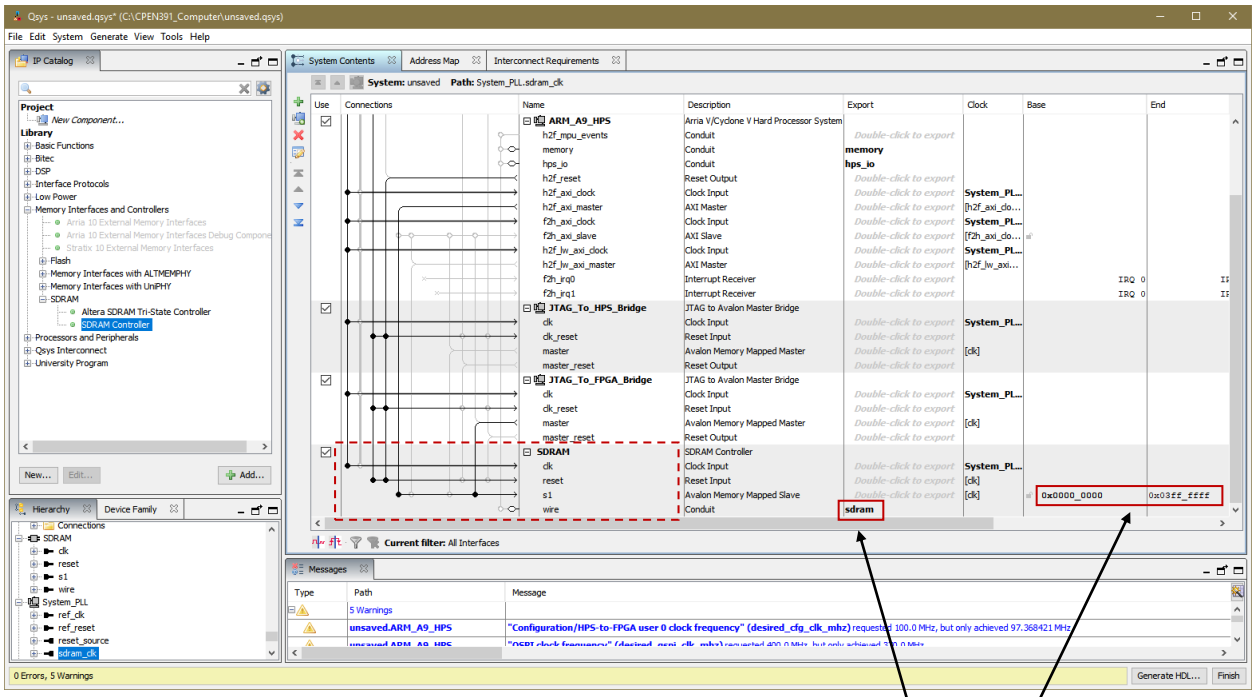




Making this connection allows the ARM/HPS to access a second source of Dram (*not that it really needs it*), but in more sophisticated systems, it could be used as a source of memory for other Soft core processors that we integrate into our system, e.g. a NIOS II processor (Altera's own in house design soft core processor that can be downloaded to the FPGA fabric). Adding this memory to our system would allow the ARM/HPS cores to communicate with any NIOS Cores we might add later through shared memory.

As with the HPS memory, this FPGA side memory has to have its timing information and size configuration set up as a set of parameters for Qsys to instantiate.

Rename the controller to **SDRAM**



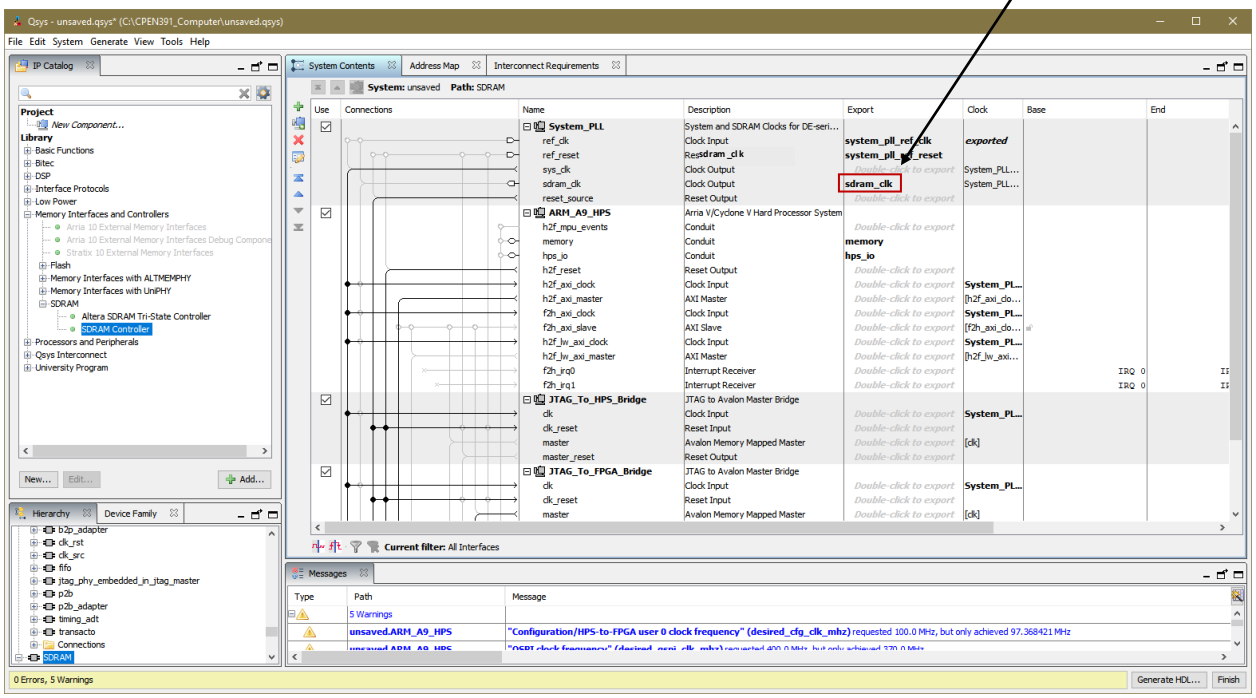
Make sure you have **export** the sdram controller "wire" and rename it to **sdram** shown above and make the other connections as shown above. Notice how a base address/range of **0x0000_0000 – 0x03ff_ffff** (64 Mbytes) has been chosen for the memory in the illustration below.

Notice its primary bus (address/data etc) labelled "**s1**" is connected to the **h2f_axi_master bus**. This means the ARM/HPS cores can access it through a **high bandwidth bus** for maximum speed of operation. This bus has a base address of **0** so the memory sits at the bottom of the ARM cores memory map.

In the previous illustration (above), you'll notice that the **sdram** was also connected to JTAG_to_FPGA Bridge's "**master**" bus. This connection, when instantiated by QSys, will allow Quartus and our software development environment to interrogate and download compiled code directly into the sdram via the JTAG port (USB port on the DE1). That is, we could chose to let the ARM/HPS processor execute their programs from this memory if required, rather than the dedicated 1GByte DDR3 memory chips on the board. Of course this would be slower, but that option now exists because of this connection.

Notice we've also added a clock source to the **SDram controller** from the **System_PLL** (system clock generator) and also two **reset sources**, one from the **System_PLL** and the other from the **HPS/ARM cores** themselves.

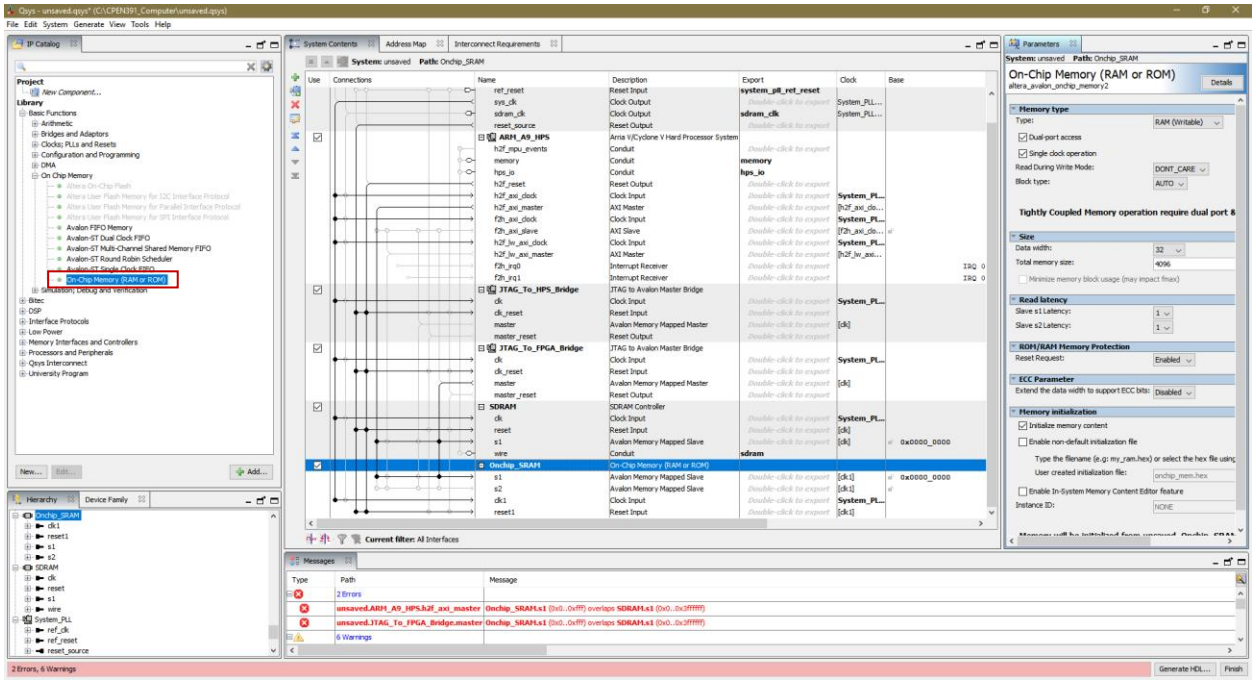
We should have done this earlier, but go back to the top and rename the clock source as **sdram_clk**, it's just a name and as such is *not really important*, but other code we use later references this name so let's be consistent.



Adding On Chip Memory

Add an instance of OnChipRam called **Onchip_SRAM** (4Kbytes in size) and set the parameters as shown. (You can view/change parameter after creating the component by double clicking on the component later – as shown here)

Also make the **connections** shown below

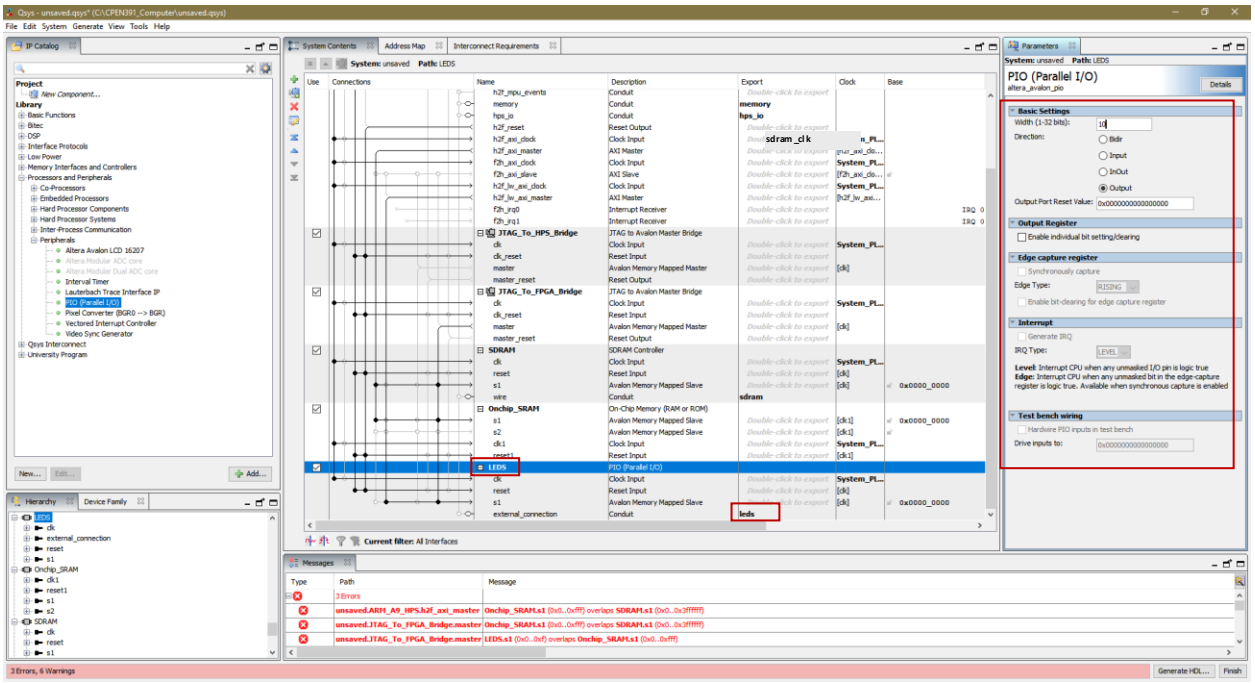


Again the new memory's main address/data bus "**s1**" is connected to the **JTag-to-FPGA Bridge's master bus** to allow downloading of data/code from some 'C' Compiler environment via the USB/JTAG port. The connection to the ARM/HPS h2f_axi_master bus also means the ARM processor can read/write to it. We'll fix the address conflict with the SDRam memory chip we created earlier at the end of this exercise.

Notice we've added a clock source to the **OnChip_SRam controller** from the **System_PLL** (system clock) and also two **reset sources**, one from the **System_PLL** and the other from the **HPS/ARM cores** themselves.

Adding a Parallel Output Port to drive the 10 Red LEDS on the DE1

Add an instance of the **PIO parallel IO port**, rename it to **LEDS** and export the "wire" and call it **leds**. Set the parameters as shown in the illustration (**10 bit output port**). Don't worry about the errors below about overlapping addresses. We'll fix those later for all devices when we have finished adding new devices to the system.



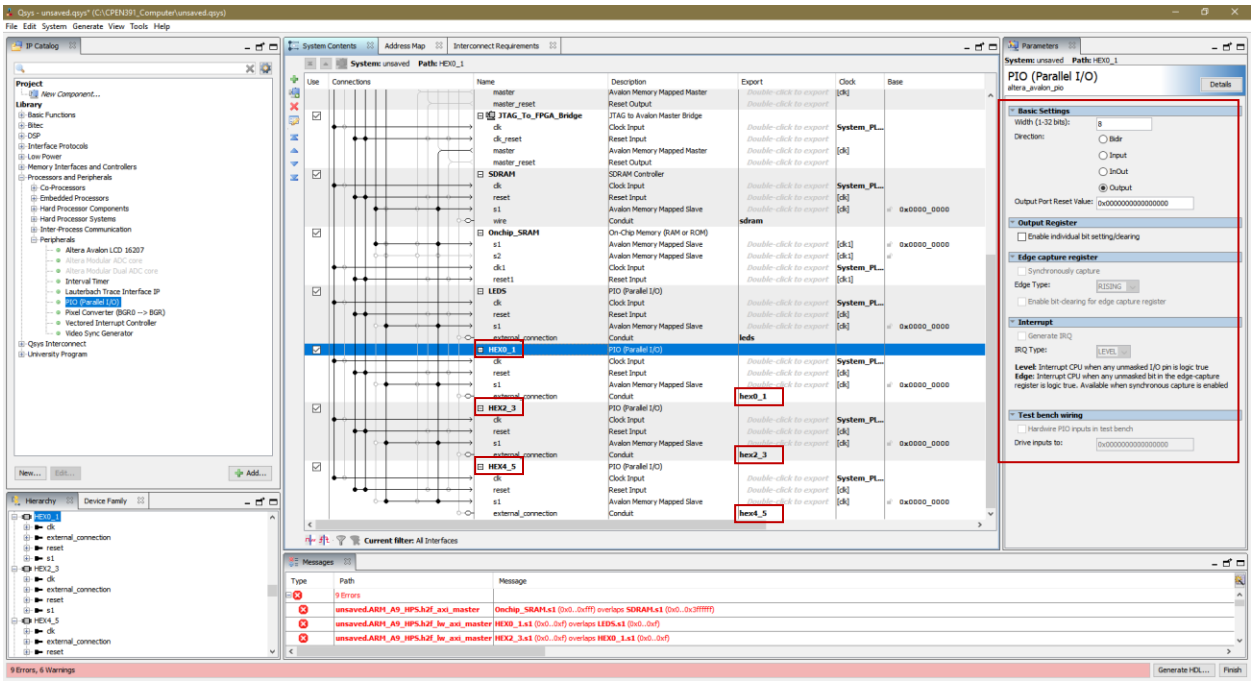
In this case, because we are adding an IO device (*rather than memory*) the connection from the LEDS port is to the ARM/HPS via the "**h2f_lw_axi_master**" i.e. the **HPS to FPGA lightweight bus** for low bandwidth devices. This means it's address is defined as (**0xFFC0000 + Offset**). We'll define the offset later.

Notice we've added a clock source to the **LEDS controller** from the **System_PLL** (system clock) and also two **reset sources**, one from the **System_PLL** and the other from the **HPS/ARM cores** themselves.

Adding Parallel Output Ports to drive the 3 pairs of 7 segment displays on the DE1

Add three more instances of the PIO parallel IO port, rename them to **HEX0_1**, **HEX2_3** and **HEX4_5** to drive the **7 segment displays** on the **DE1** board and **export** the signals from them as **hex0_1, hex2_3**, **hex4_5**.

Set the parameters as shown in the illustration below (8 bit output ports) and wire up as shown. Again don't worry about the errors below about overlapping addresses. We'll fix those later for all devices when we have finished adding new devices to the system.

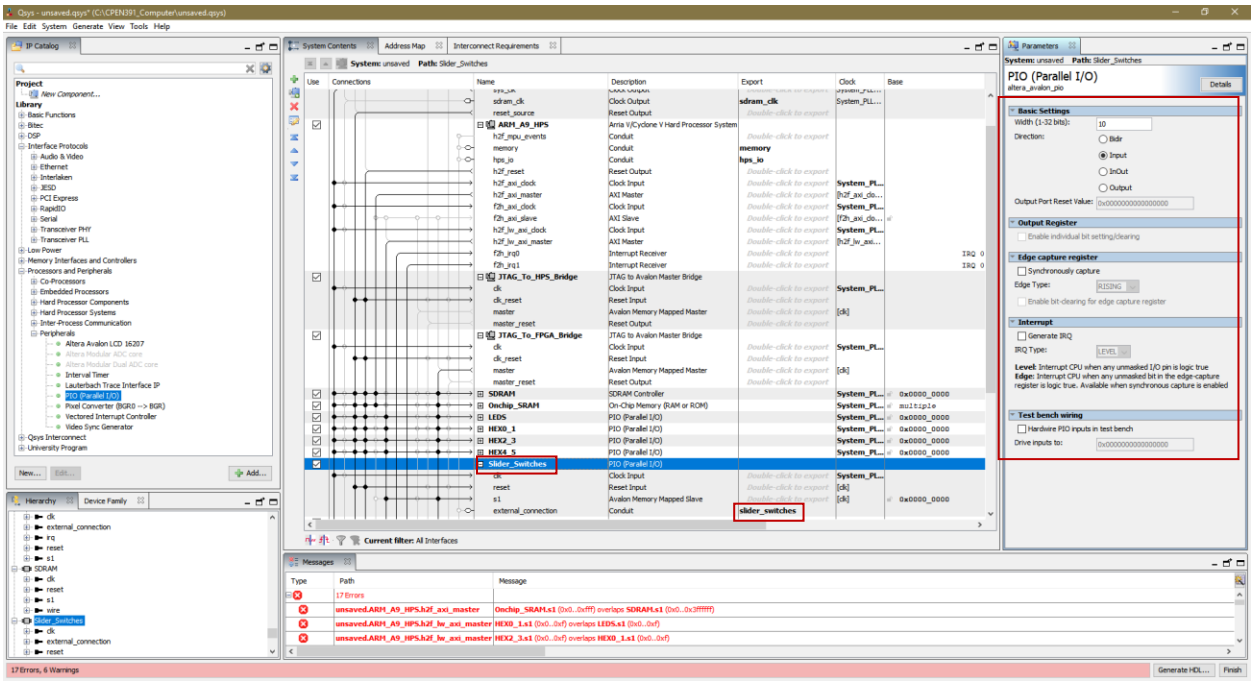


As before, because we are adding an IO device (rather than memory) the connection from the HEX ports is to the ARM/HPS via the "**h2f_lw_axi_master**" i.e. the **HPS to FPGA lightweight bus** for low bandwidth devices. This means it's address is defined as (**0xFFC0000 + Offset**). We'll define the offset later.

Notice we've added a clock source to the **Hex display controllers** from the **System_PLL** (system clock) and also two **reset sources**, one from the **System_PLL** and the other from the **HPS/ARM cores** themselves.

Adding a Parallel Input Port to read from the 10 slider switches on the DE1

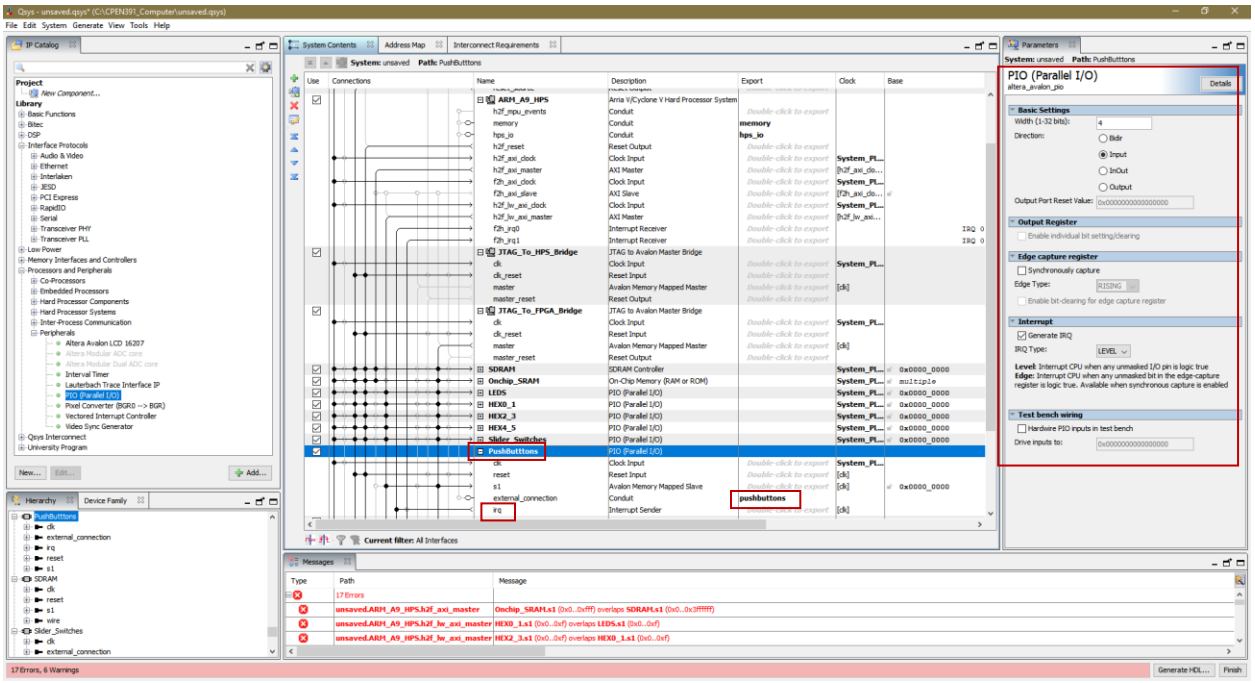
Add an instance of the **PIO parallel IO port**, rename it to **Slider_Switches** and export the signal as **slider_switches**. Set the parameters as shown in the illustration (**10 bit input port**) with IRQ capability and wire up as shown.



As before, because we are adding an IO device (rather than memory) the connection from the Slider Switches port is to the ARM/HPS via the "**h2f_lw_axi_master**" i.e. the **HPS to FPGA lightweight bus** for low bandwidth devices. This means it's address is defined as (**0xFFC0000 + Offset**). We'll define the offset later.

Adding an Input Port for the 4 Push Buttons on the DE1

Add an instance of the PIO parallel IO port, rename it to **PushButtons** and export the **pushbuttons**. Set the parameters as shown in the illustration (4 bit input port with ***synchronous capture*** and ***Interrupt generate capability*** on ***falling edge*** of push button signal, i.e. when pushed) and wire up as shown (don't forget to wire the **IRQ** signal to the interrupt received on the **hps_0** called **f2h_irq0**).



As before, because we are adding an IO device (rather than memory) the connection from the Push Button port is to the ARM/HPS via the "**h2f_lw_axi_master**" i.e. the **HPS to FPGA lightweight bus** for low bandwidth devices. This means it's address is defined as (**0xFFC0000 + Offset**). We'll define the offset later.

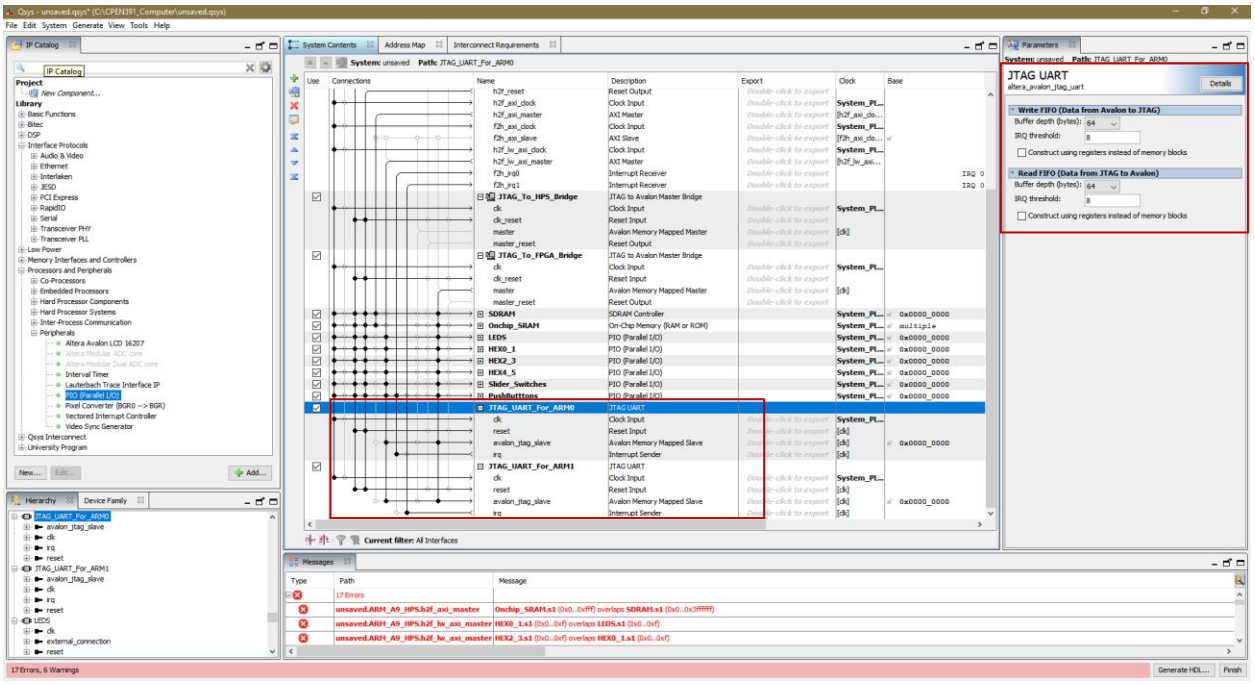
Notice we've added a clock source to the **Push Buttons controller** from the **System_PLL** (system clock) and also two **reset sources**, one from the **System_PLL** and the other from the **HPS/ARM cores** themselves.

We've also added an interrupt signal **irq** to allow the push buttons to interrupt **Core 0** on the Dual core ARM processor.

Adding a Serial Port to Communicate with the Dual ARM Cortex A9 Processor Cores in HPS

Add two instances of a **JTAG_UART** to allow us to communicate with the 2 separate ARM cores inside the Cydone V FPGA chip. Rename them to **JTAG_UART_For_ARM_0** and **JTAG_UART_For_ARM_1**.

Set each up as shown in the parameters and wire up as shown below. Remember to connect the interrupt signals from the UARTS. Also remember to connect their **IRQ signals** to the HPS/ARM cores IRQ receiver (labelled **f2h_irq0**)

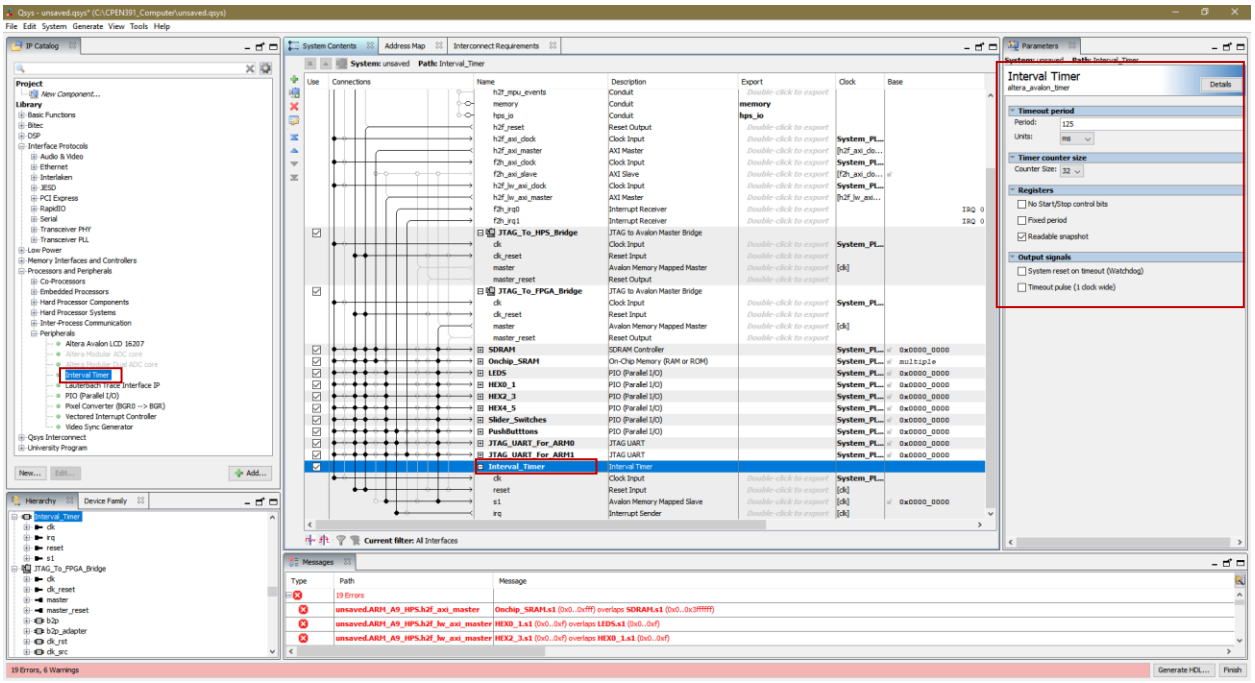


As before, because we are adding an IO device (rather than memory) the main bus connection from the JTAG_UART ports is to the ARM/HPS via the "**h2f_lw_axi_master**" i.e. the **HPS to FPGA lightweight bus** for low bandwidth devices. This means it's address is defined as (**0xFFC0000 + Offset**). We'll define the offset later.

Note that the IRQ connections for each UART go to **DIFFERENT** Cores on the A9 processor which makes sense since the UARTS provide a communication link for 1 core **each.**

Adding an Interval Timer to our System.

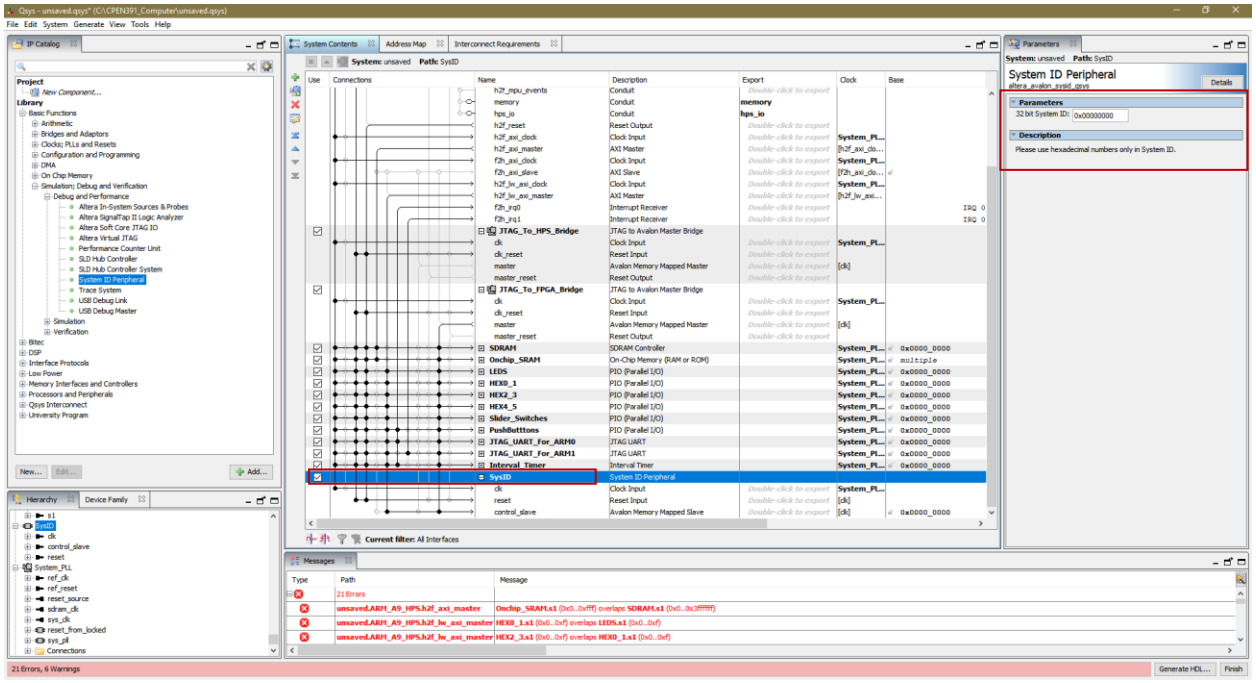
Add an instance of the **Interval Timer** with readable snapshot and interrupt capability. Rename it to **Interval_Timer**, wire up as shown below and set according to the parameters. Remember to wire up the Interrupt output from the timer "**irq**" to the **f2h_irq0 receiver** on the HPS/ARM core.



As before, because we are adding an IO device (rather than memory) the connection from the Push Button port is to the ARM/HPS via the "**h2f_lw_axi_master**" i.e. the **HPS to FPGA lightweight bus** for low bandwidth devices. This means it's address is defined as (**0xFFC0000 + Offset**). We'll define the offset later. This timer can also interrupt ARM Core 0.

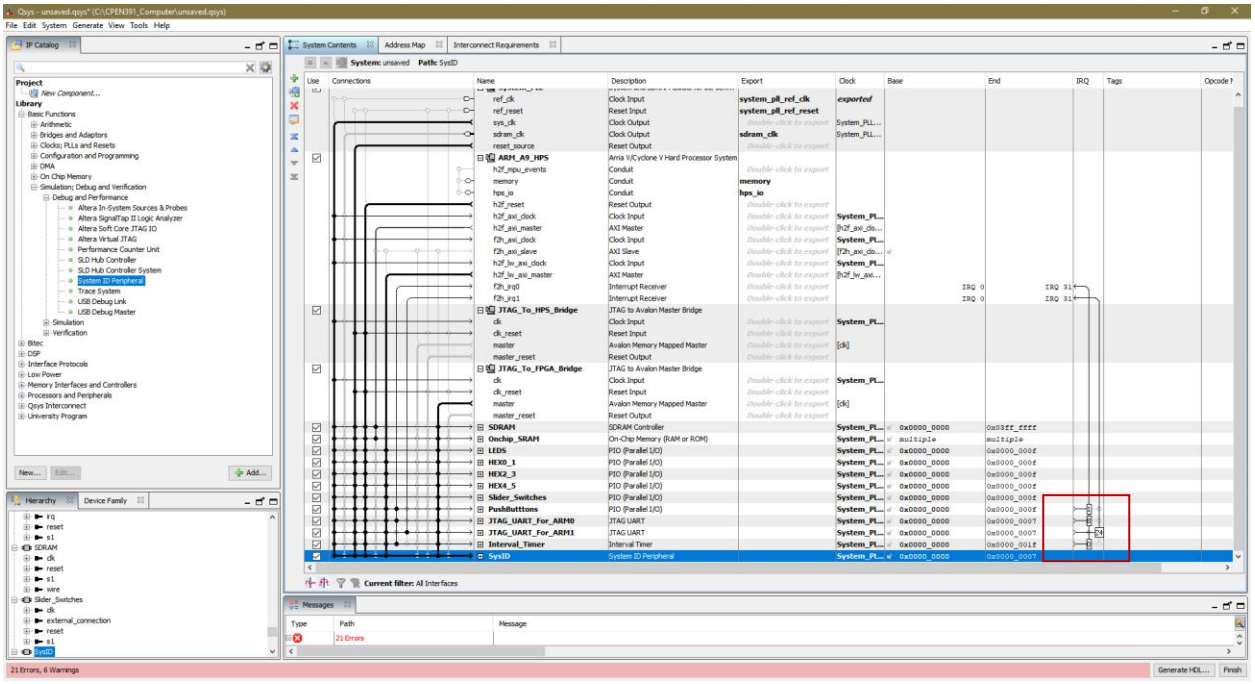
Adding a System Identification Device to the System

Add a System ID Peripheral. Rename it to **SysID** and wire up as shown below. The sytem ID component allows a software download tool to interrogate an ID number given to this particular configuration of system. This would enable it to determine if it is downloading to the correct system, e.g. imagine a firmware update of some critical system such as the Engine management system in a car. The last thing you would want is for software that is incompatible with that version of hardware to be downloaded and "flashed" into the system memory since that would render it useless i.e.it would "Brick it". The ability to read a system ID thus prevents that happening (*if used properly*). For this tutorial however, we'll just set the ID to 0.



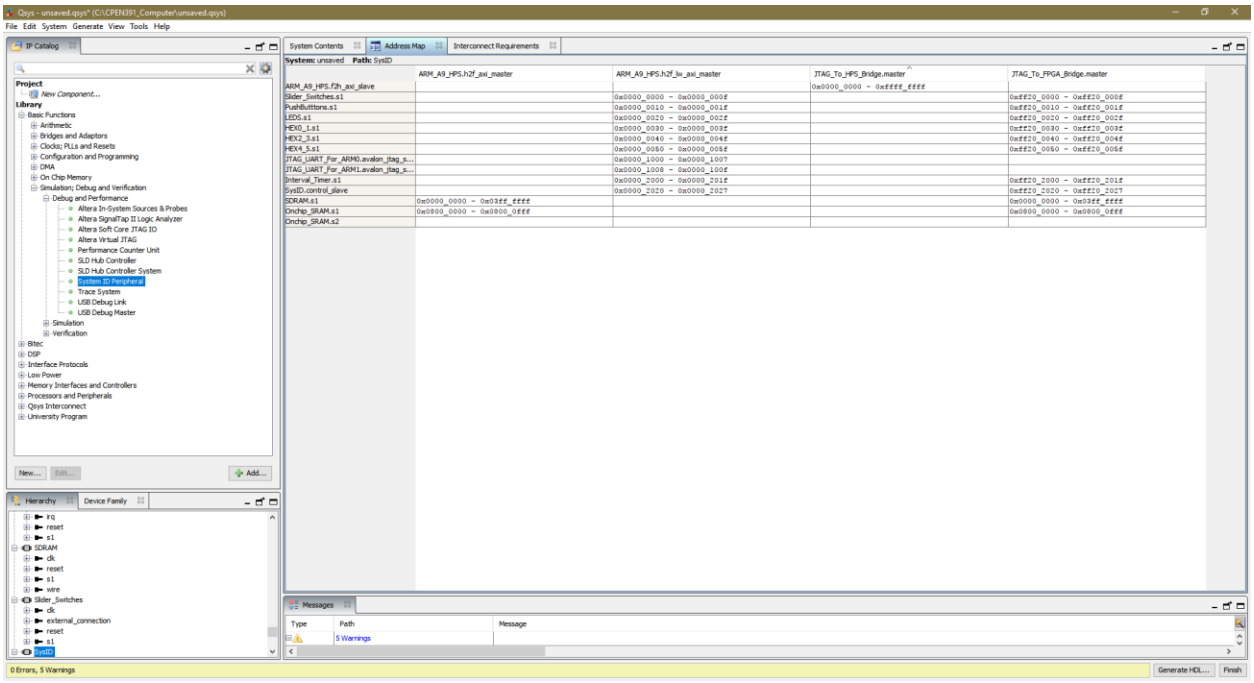
Connecting up Interrupt Capable devices to the ARM HPS

Connect the IO devices to use the **interrupt levels** shown below. These are somewhat arbitrary, but stick with them.



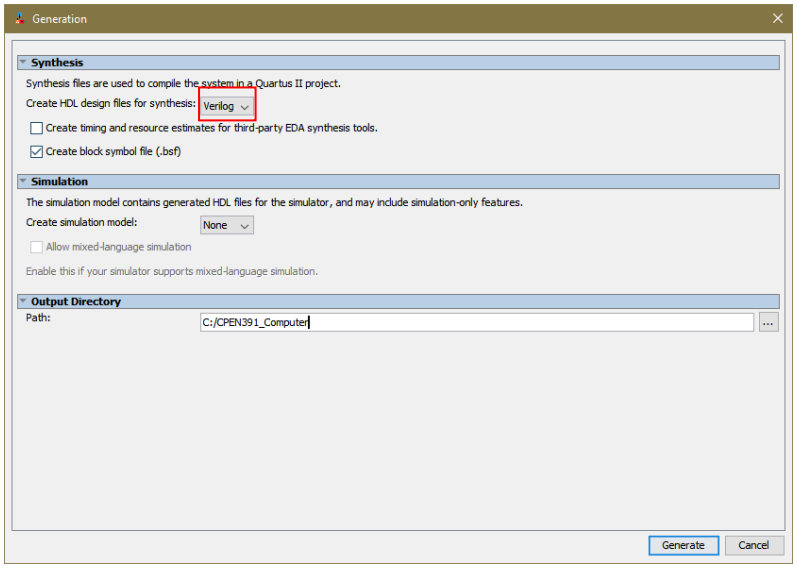
Setting the devices base addresses.

Click on the **Address Map Tab** and set the addresses as shown in the table below by clicking on an entry. MAKE SURE you do this correctly as it **sets base address that we will use in our C code.** The choices are somewhat arbitrary (we the designer get to chose them). Note one of the address is **removed**.



When done correctly the error messages/ address conflicts disappear.

Now click the **Generate HDL** button and choose **Verilog** top level system design (*see below*). That is, the top level in the circuit hierarchy. Save the ".qsys" file with a name matching your project e.g. "**CPEN391_Computer.qsys**".

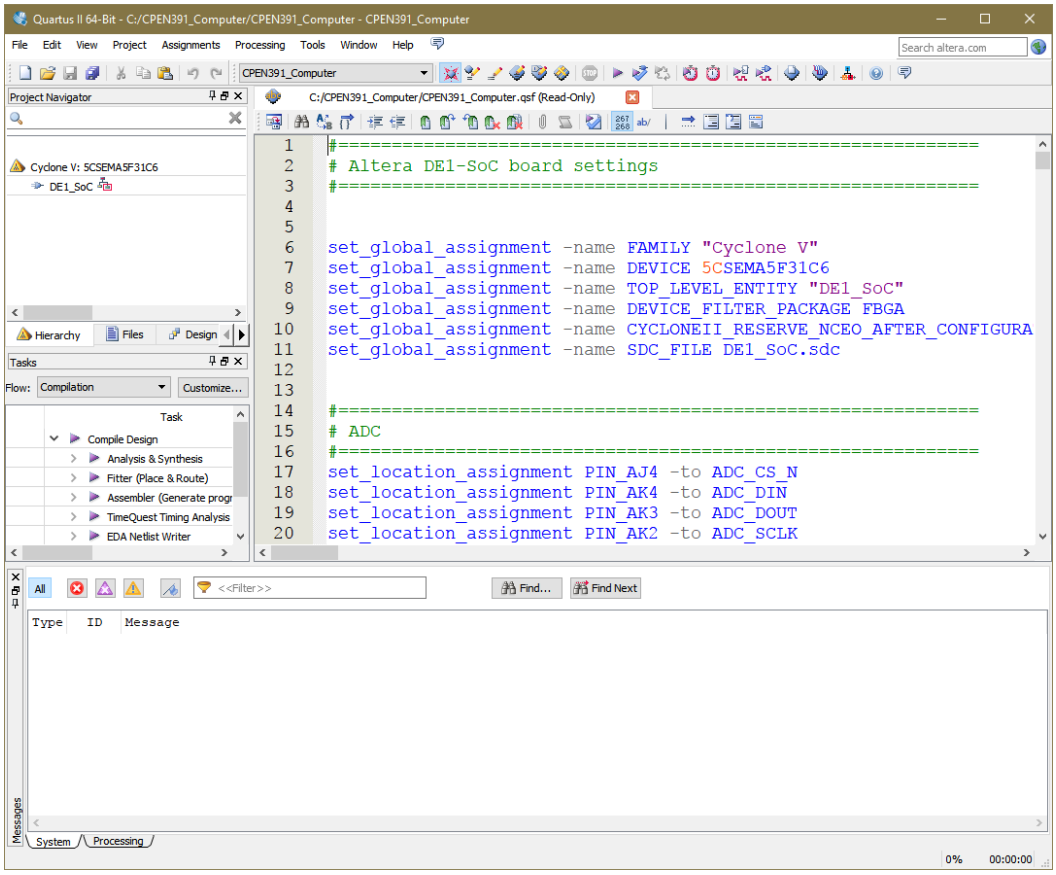


When qsys has generated the system, press "**finish**" to exit. You will then see a dialog that states you have to add the .**qip** files to the project which we will do in a moment. **Note**: if you need to make changes at any time in the future, go back into qsys and re-open the ".qsys" file.

Add Various System Files - Add the Quartus Settings file

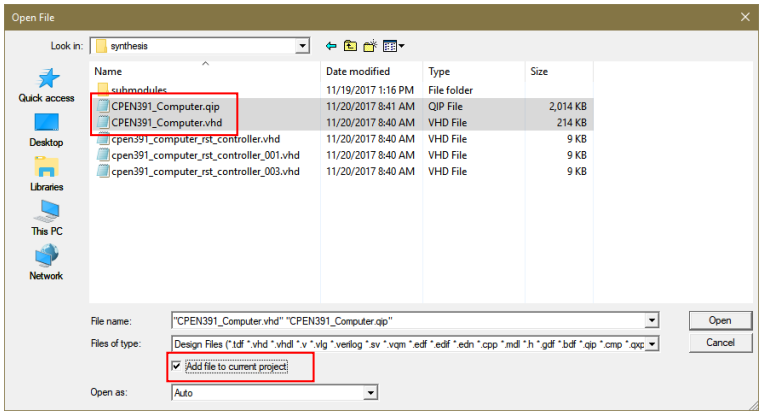
Your instructor will give you a copy of a Quartus settings file (**DE1_Soc_Quartus_Settings_File.QSF**) which contains name-to-pin assignments. Rename it to match your project name i.e. **CPEN391_Computer.qsf**.

Now copy it to your Quartus Project folder (the same folder where the **".qpf"** – project settings file is located). If one is already present, then simply **replace** it with the one given. If you open it, it will look (*similar*) to this:-

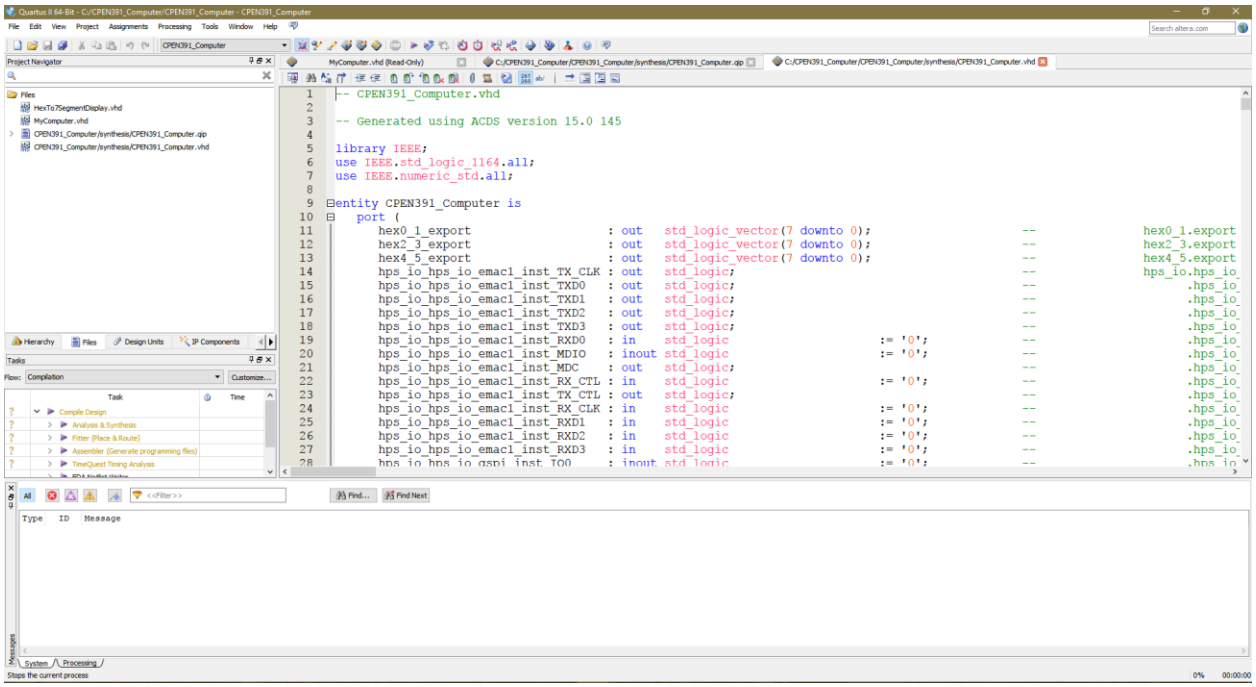


Add the ".qip" file and ".vhd" file produced by Qsys

Now **locate**, **open** and **tick** the "**add to file**" check box, the files "**CPEN391_Computer.qip**" and "**CPEN391_Computer.vhd"** created by Qsys to the project (it should be in a sub-folder called *synthesis*). If you ticked the box to generate simulation files, QSys will have generated a "**.sip**" file too, so add that file to the project in Quartus.

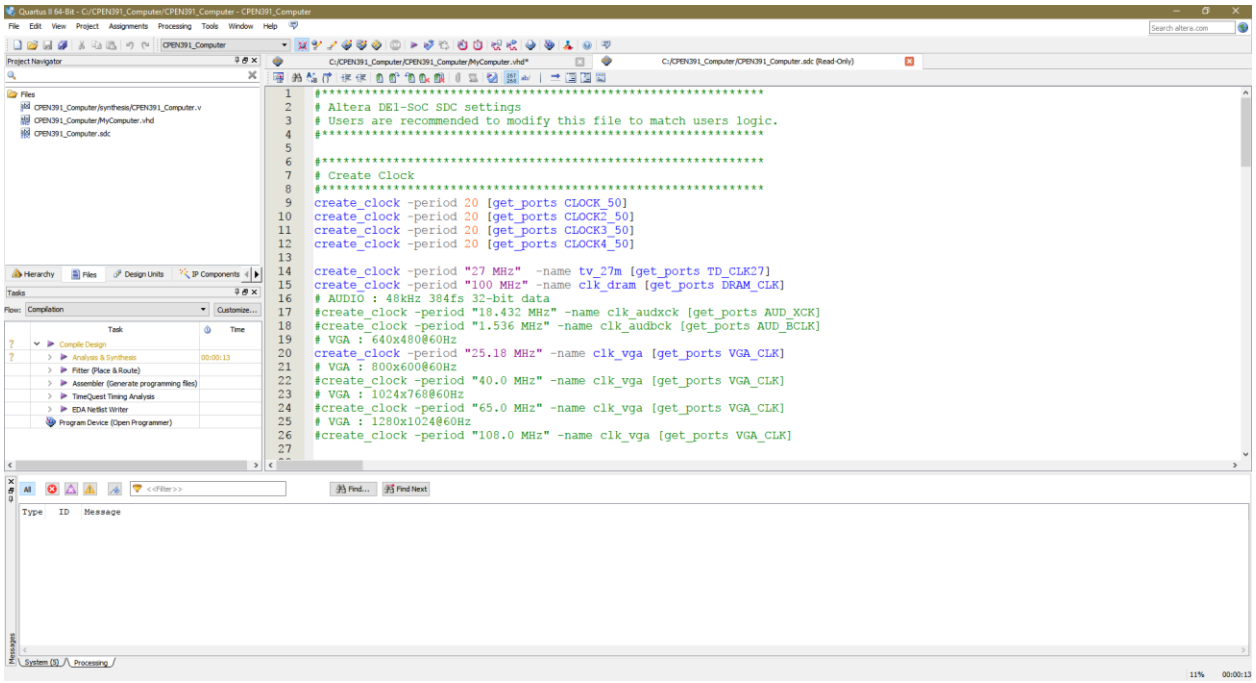


Look at the "**CPEN391_Computer.vhd**" file to see what Qsys generated. It's a bit of a **monster** really. The file lists all the **inputs** and **outputs** of our system as well as **declarations** of all the **components** we added to the system using QSys, e.g. the **slider switches**, **System ID, Hex displays** etc. The important thing is the declarations of the port settings shown below. We'll use these later.



Add the Synopsis Design Constraints (.SDC) File to the Project

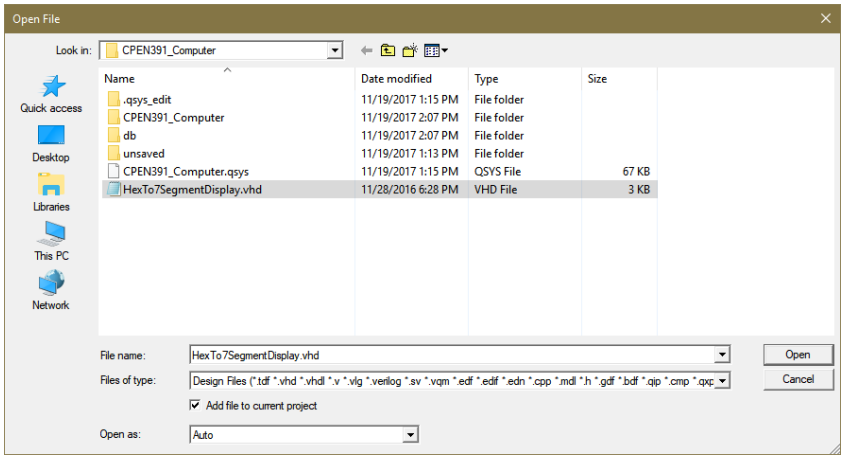
Now copy the "**DE1_SoC.sdc**" (*Synopsis Design constraints*) file given to you by your instructor to the project folder and **open/add that file** to the project. If you open it, it looks like this, with critical *constraints* defined for the DE1 board, e.g. timing variations etc. This name matches the **SDC file name** given in the "**.qsf**" file previously.



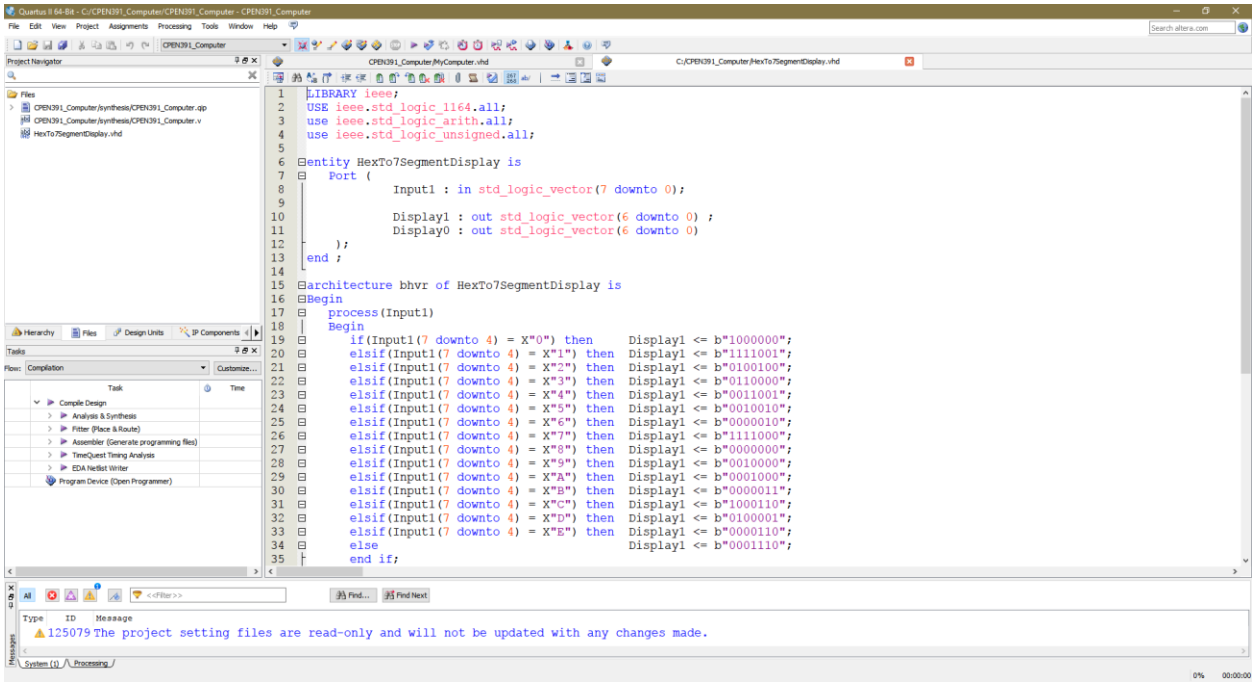
```
#**************************************************************
# Altera DE1-SoC SDC settings
# Users are recommended to modify this file to match users logic.
#**************************************************************

#**************************************************************
# Create Clock
#**************************************************************
create_clock -period 20 [get_ports CLOCK_50]
create_clock -period 20 [get_ports CLOCK2_50]
create_clock -period 20 [get_ports CLOCK3_50]
create_clock -period 20 [get_ports CLOCK4_50]

create_clock -period "27 MHz"  -name tv_27m [get_ports TD_CLK27]
create_clock -period "100 MHz" -name clk_dram [get_ports DRAM_CLK]
# AUDIO : 48kHz 384fs 32-bit data
#create_clock -period "18.432 MHz" -name clk_audxck [get_ports AUD_XCK]
#create_clock -period "1.536 MHz" -name clk_audbck [get_ports AUD_BCLK]
# VGA : 640x480@60Hz
create_clock -period "25.18 MHz" -name clk_vga [get_ports VGA_CLK]
# VGA : 800x600@60Hz
#create_clock -period "40.0 MHz" -name clk_vga [get_ports VGA_CLK]
# VGA : 1024x768@60Hz
#create_clock -period "65.0 MHz" -name clk_vga [get_ports VGA_CLK]
# VGA : 1280x1024@60Hz
#create_clock -period "108.0 MHz" -name clk_vga [get_ports VGA_CLK]
```

Add the BCD to 7 Segment decoder design file

Copy from Connect the HexTo7SegmentDisplay.vhd design and open/add it to the project project.



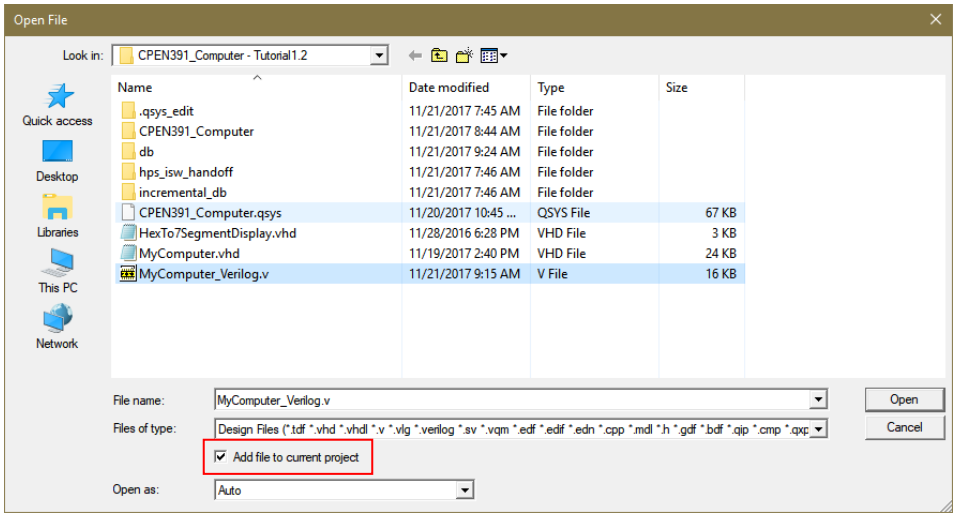
This file looks like this. Yes it's **VHDL** not **Verilog**, but apart from Syntax, the concepts are identical. You could easily *translate* and we live in a **mixed** HDL world anyway, so some familiarity with both is useful.



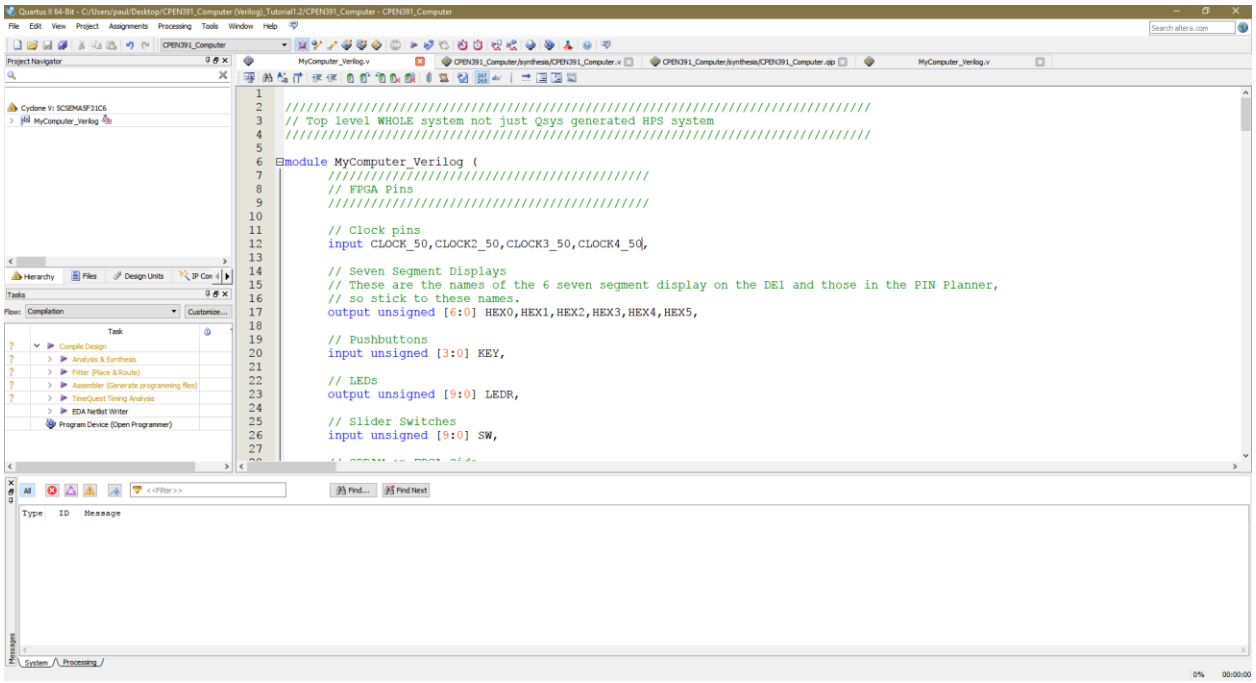
Instantiating the QSYS generated System in Quartus

This is the **important bit**. Now we need to *instantiate* the new **QSys** generated system and make it **THE** system we are going to use in our Project. We can then add other hardware to this and write application software based on it. First step however in the *instantiation* is to **create a new HDL file**. I'm using Verilog here so I've called mine **MyComputer_Verilog.v**. (There's also a VHDL version on Connect also called MyComputer.vhd, if you wanted to take a look at structural VHDL coding)

IMPORTANT: This file is on Connect so you don't have to type it in *(it's a lot of detailed typing)* we'll just go through it here. Make sure you copy the file to your project, open and **add** it to the project.



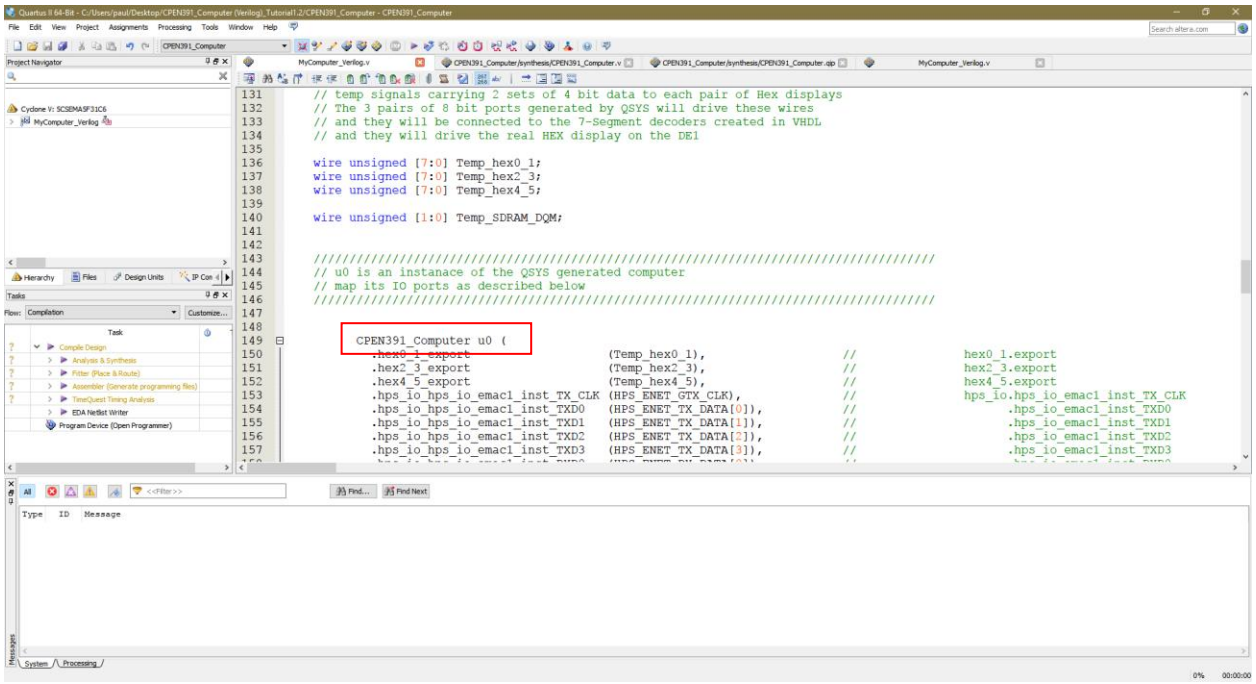
The **MyComputer_Verilog.v** file looks like this



In this file, we describe the **top level entity**, that is, our **total computer system** (top down), with all inputs and outputs to the outside/real word. As stated above, this file is posted on Connect for you to see in its entirety so you only need to copy it to your project folder and add it to the project.

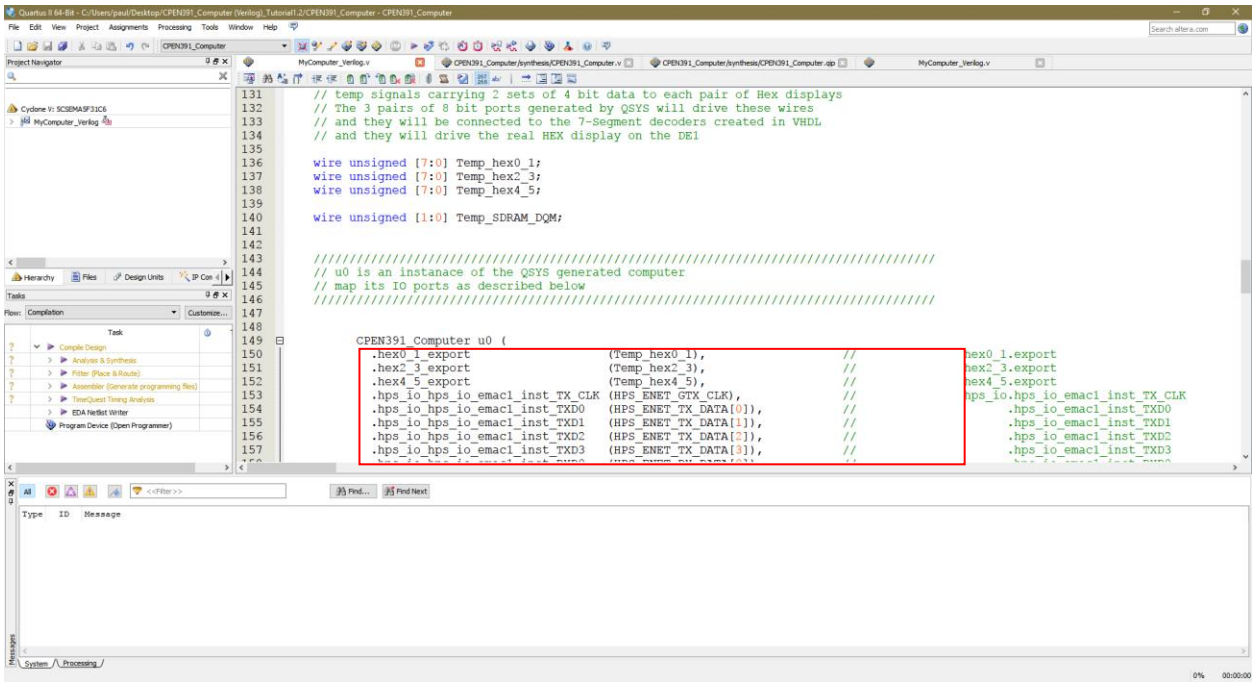
It also contains a declaration of the **Qsys** generated component **CPEN391_Computer** (*see below*) and some temporary signals/wires which we'll come back to for connecting the hex displays to some decoders. We are going to connect various components using VHDL **structural modelling** syntax.

This declaration came from a **Verilog** file generated by **Qsys** and was pasted into the **MyComputer_Verilog.v** file in the appropriate place, so Quartus knows about this component/module when compiling **MyComputer_Verilog.v**.



```
131     // temp signals carrying 2 sets of 4 bit data to each pair of Hex displays
132     // The 3 pairs of 8 bit ports generated by QSYS will drive these wires
133     // and they will be connected to the 7-Segment decoders created in VHDL
134     // and they will drive the real HEX display on the DE1
135
136     wire unsigned [7:0] Temp_hex0_1;
137     wire unsigned [7:0] Temp_hex2_3;
138     wire unsigned [7:0] Temp_hex4_5;
139
140     wire unsigned [1:0] Temp_SDRAM_DQM;
141
142
143     ////////////////////////////////////////////////////////////////////////////////
144     // u0 is an instanace of the QSYS generated computer
145     // map its IO ports as described below
146     ////////////////////////////////////////////////////////////////////////////////
147
148
149         CPEN391_Computer u0 (
150             .hex0_1_export                  (Temp_hex0_1),                  //              hex0_1.export
151             .hex2_3_export                  (Temp_hex2_3),                  //              hex2_3.export
152             .hex4_5_export                  (Temp_hex4_5),                  //              hex4_5.export
153             .hps_io_hps_io_emac1_inst_TX_CLK (HPS_ENET_GTX_CLK),            //          hps_io.hps_io_emac1_inst_TX_CLK
154             .hps_io_hps_io_emac1_inst_TXD0  (HPS_ENET_TX_DATA[0]),          //                 .hps_io_emac1_inst_TXD0
155             .hps_io_hps_io_emac1_inst_TXD1  (HPS_ENET_TX_DATA[1]),          //                 .hps_io_emac1_inst_TXD1
156             .hps_io_hps_io_emac1_inst_TXD2  (HPS_ENET_TX_DATA[2]),          //                 .hps_io_emac1_inst_TXD2
157             .hps_io_hps_io_emac1_inst_TXD3  (HPS_ENET_TX_DATA[3]),          //                 .hps_io_emac1_inst_TXD3
```

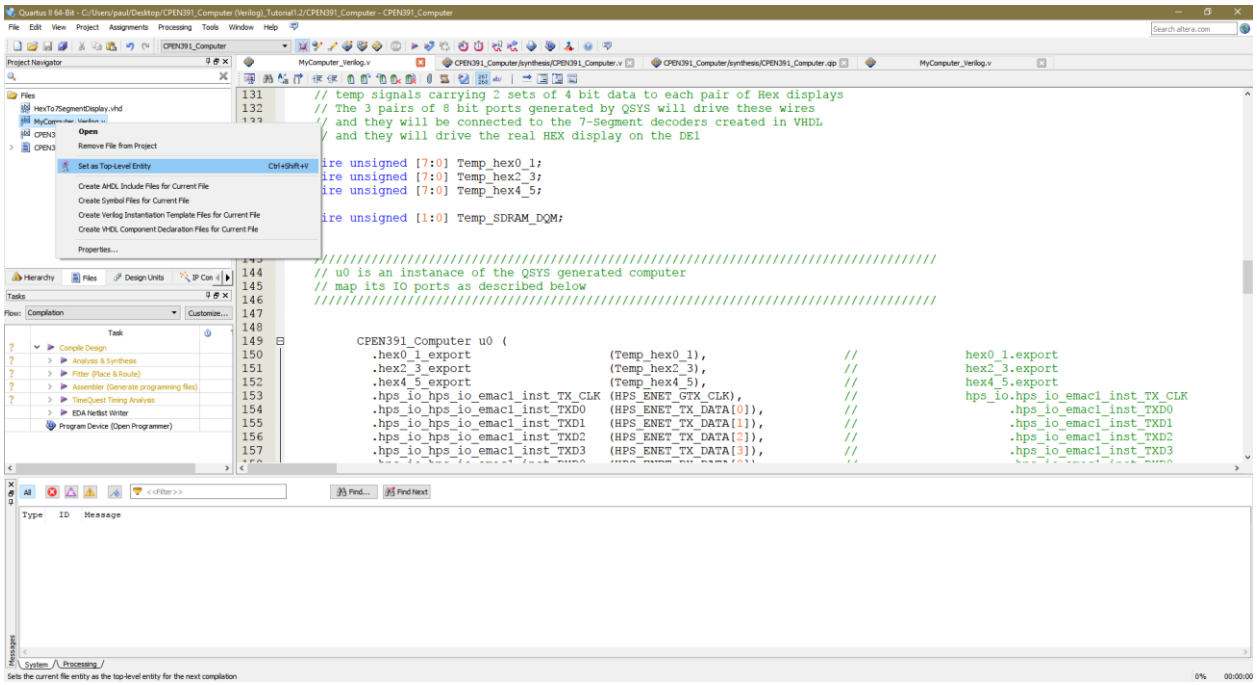
MyComputer_Verilog.v also *instantiates* an instance of the component/module **CPEN391_Computer** called **u0** (*see above*) and makes the IO connections from the component/module **u0** to the signals, inputs and outputs in the top level entity (see below) :-



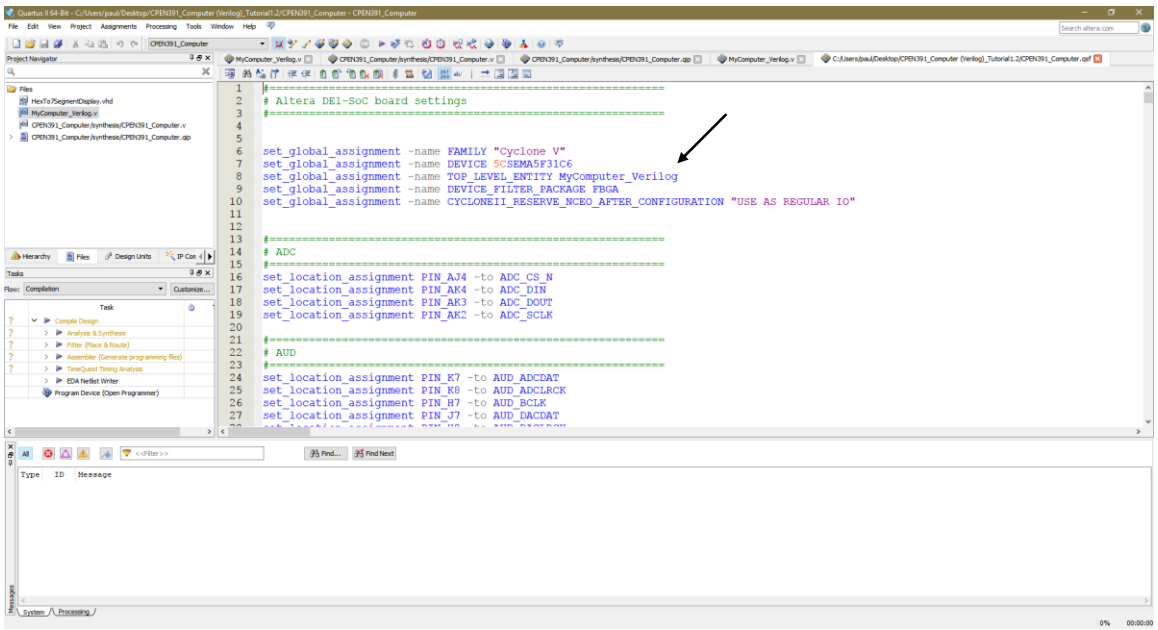
```
131     // temp signals carrying 2 sets of 4 bit data to each pair of Hex displays
132     // The 3 pairs of 8 bit ports generated by QSYS will drive these wires
133     // and they will be connected to the 7-Segment decoders created in VHDL
134     // and they will drive the real HEX display on the DE1
135
136     wire unsigned [7:0] Temp_hex0_1;
137     wire unsigned [7:0] Temp_hex2_3;
138     wire unsigned [7:0] Temp_hex4_5;
139
140     wire unsigned [1:0] Temp_SDRAM_DQM;
141
142
143     ////////////////////////////////////////////////////////////////////////////////
144     // u0 is an instanace of the QSYS generated computer
145     // map its IO ports as described below
146     ////////////////////////////////////////////////////////////////////////////////
147
148
149         CPEN391_Computer u0 (
150             .hex0_1_export                  (Temp_hex0_1),                  //              hex0_1.export
151             .hex2_3_export                  (Temp_hex2_3),                  //              hex2_3.export
152             .hex4_5_export                  (Temp_hex4_5),                  //              hex4_5.export
153             .hps_io_hps_io_emac1_inst_TX_CLK (HPS_ENET_GTX_CLK),            //          hps_io.hps_io_emac1_inst_TX_CLK
154             .hps_io_hps_io_emac1_inst_TXD0  (HPS_ENET_TX_DATA[0]),          //                 .hps_io_emac1_inst_TXD0
155             .hps_io_hps_io_emac1_inst_TXD1  (HPS_ENET_TX_DATA[1]),          //                 .hps_io_emac1_inst_TXD1
156             .hps_io_hps_io_emac1_inst_TXD2  (HPS_ENET_TX_DATA[2]),          //                 .hps_io_emac1_inst_TXD2
157             .hps_io_hps_io_emac1_inst_TXD3  (HPS_ENET_TX_DATA[3]),          //                 .hps_io_emac1_inst_TXD3
```

Defining the Top Level Entity: MyComputer_Verilog.v

Quartus needs to know which file in the project represents the highest level design file in your project hierarchy (i.e. the one at the top of our design hierarchy). To do this, right mouse click the **MyComputer_Verilog.v** design file and set it as the **top level entity** as shown below.

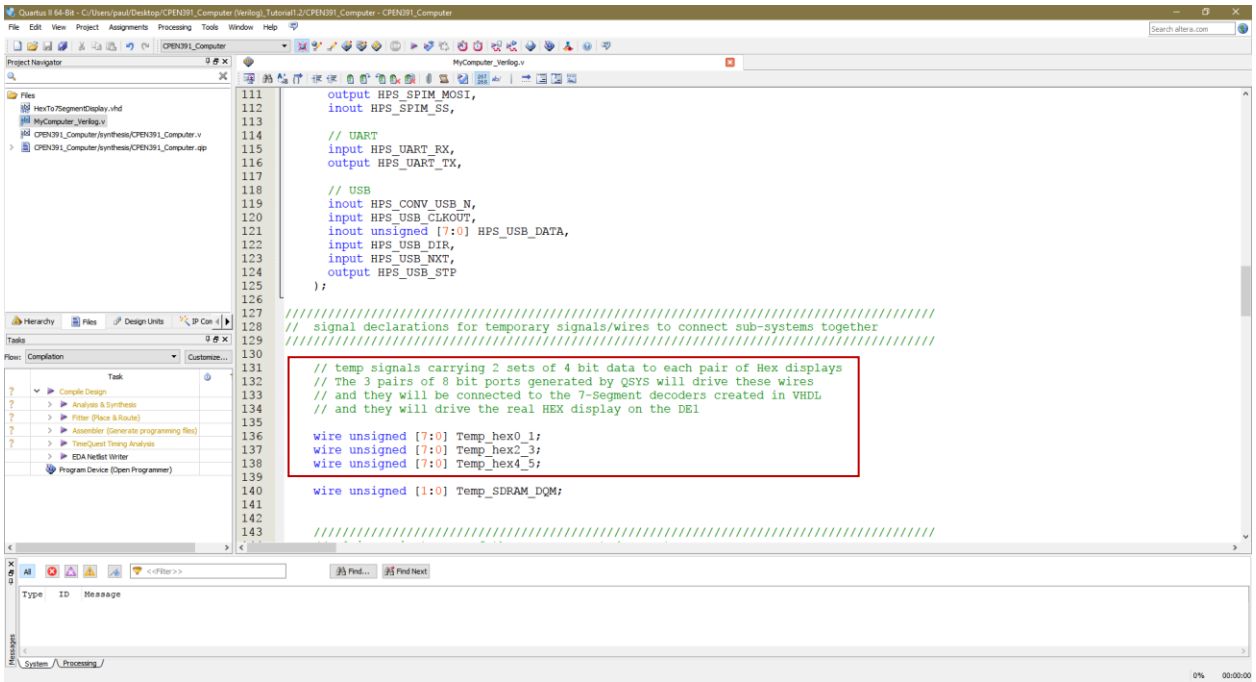


Make sure the ".qsf" file maintained by Quartus updates itself to show the MyComputer_Verilog.v file as the top level entity (*see below*). You may have to quit and restart Quartus for the effect to take place.



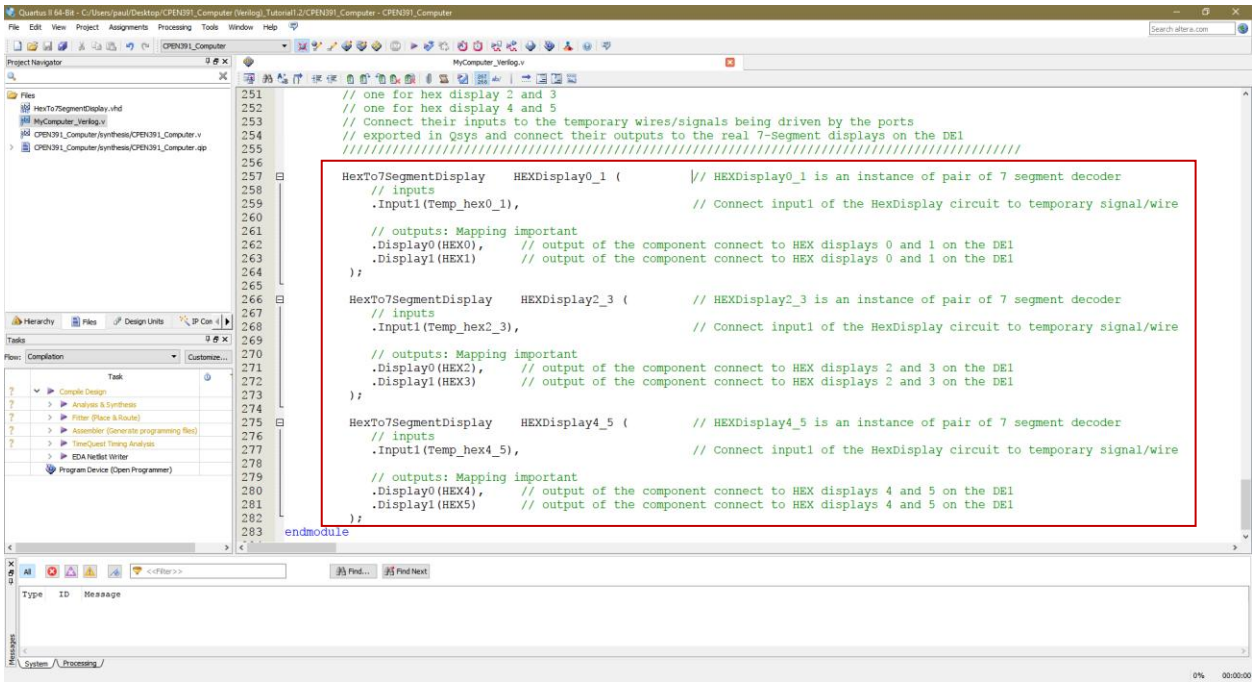
Connecting the HEX displays

MyComputer_Verilog .v top level file also creates 3 instances of the hex to 7 segment decoder and connects them up in. Here are the temporary signals/wires to make the connections.



```
111      output HPS_SPIM_MOSI,
112      inout HPS_SPIM_SS,
113
114      // UART
115      input HPS_UART_RX,
116      output HPS_UART_TX,
117
118      // USB
119      inout HPS_CONV_USB_N,
120      input HPS_USB_CLKOUT,
121      inout unsigned [7:0] HPS_USB_DATA,
122      input HPS_USB_DIR,
123      input HPS_USB_NXT,
124      output HPS_USB_STP
125    );
126
127  ////////////////////////////////////////////////////////////////////////////////
128  //  signal declarations for temporary signals/wires to connect sub-systems together
129  ////////////////////////////////////////////////////////////////////////////////
130
131     // temp signals carrying 2 sets of 4 bit data to each pair of Hex displays
132     // The 3 pairs of 8 bit ports generated by QSYS will drive these wires
133     // and they will be connected to the 7-Segment decoders created in VHDL
134     // and they will drive the real HEX display on the DE1
135
136     wire unsigned [7:0] Temp_hex0_1;
137     wire unsigned [7:0] Temp_hex2_3;
138     wire unsigned [7:0] Temp_hex4_5;
139
140     wire unsigned [1:0] Temp_SDRAM_DQM;
141
142
143     ////////////////////////////////////////////////////////////////////////////////
```

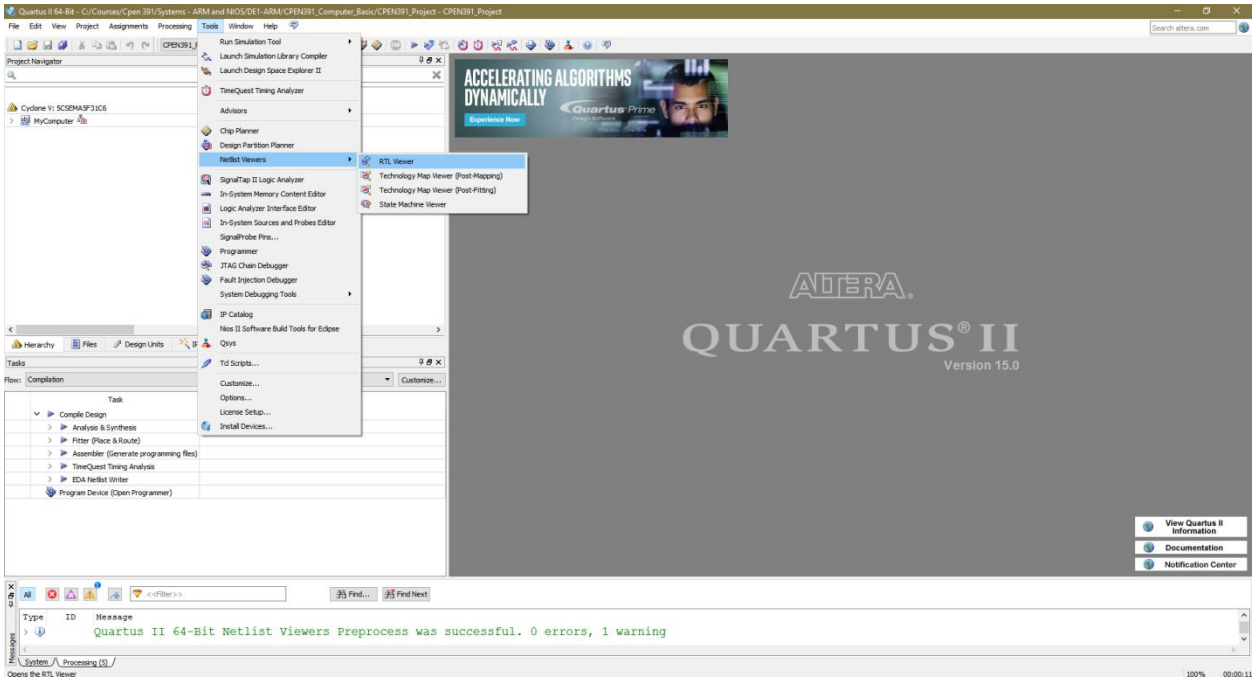
And finally **MyComputer_Verilog.v** creates the 3 instances of that component and wires them to temporary signals above (right at the end of the file).



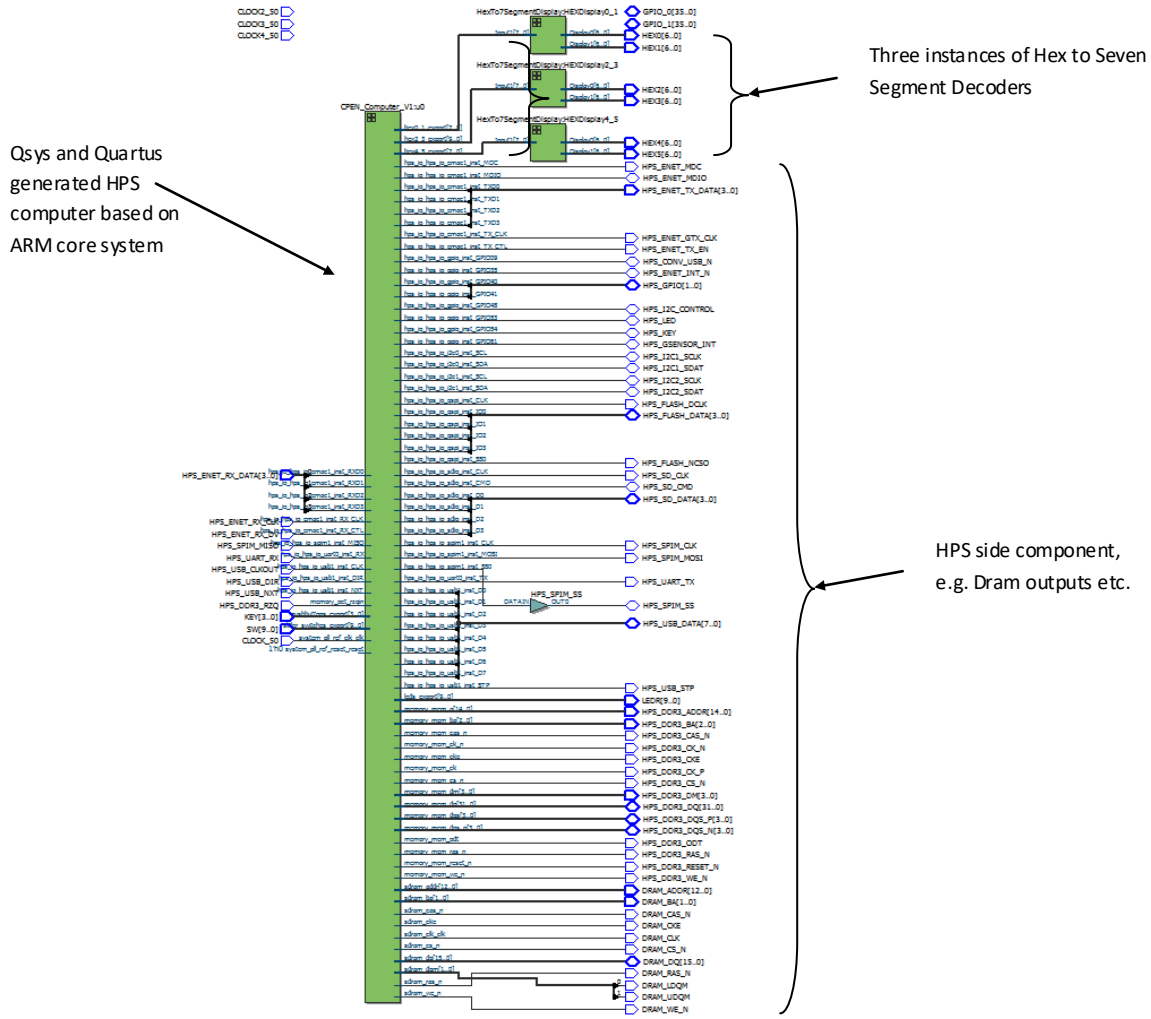
```
251        // one for hex display 2 and 3
252        // one for hex display 4 and 5
253        // Connect their inputs to the temporary wires/signals being driven by the ports
254        // exported in Qsys and connect their outputs to the real 7-Segment displays on the DE1
255        ////////////////////////////////////////////////////////////////////////////////
256
257        HexTo7SegmentDisplay    HEXDisplay0_1 (          // HEXDisplay0_1 is an instance of pair of 7 segment decoder
258            // inputs
259            .Input1(Temp_hex0_1),                        // Connect input1 of the HexDisplay circuit to temporary signal/wire
260
261            // outputs: Mapping important
262            .Display0(HEX0),      // output of the component connect to HEX displays 0 and 1 on the DE1
263            .Display1(HEX1)       // output of the component connect to HEX displays 0 and 1 on the DE1
264         );
265
266         HexTo7SegmentDisplay    HEXDisplay2_3 (          // HEXDisplay2_3 is an instance of pair of 7 segment decoder
267            // inputs
268            .Input1(Temp_hex2_3),                        // Connect input1 of the HexDisplay circuit to temporary signal/wire
269
270            // outputs: Mapping important
271            .Display0(HEX2),      // output of the component connect to HEX displays 2 and 3 on the DE1
272            .Display1(HEX3)       // output of the component connect to HEX displays 2 and 3 on the DE1
273         );
274
275         HexTo7SegmentDisplay    HEXDisplay4_5 (          // HEXDisplay4_5 is an instance of pair of 7 segment decoder
276            // inputs
277            .Input1(Temp_hex4_5),                        // Connect input1 of the HexDisplay circuit to temporary signal/wire
278
279            // outputs: Mapping important
280            .Display0(HEX4),      // output of the component connect to HEX displays 4 and 5 on the DE1
281            .Display1(HEX5)       // output of the component connect to HEX displays 4 and 5 on the DE1
282         );
283  endmodule
```

Compiling the Project

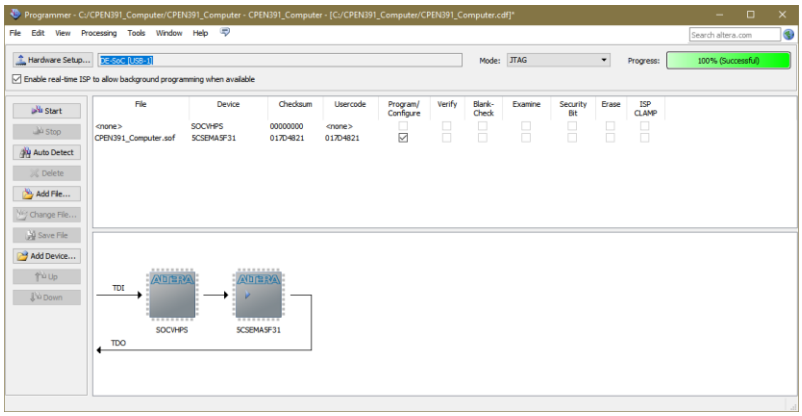
If you have done all this correctly, the project should now compile in Quartus (*with no errors – fingers crossed*). If you examine the RTL viewer (*see below*)



It should present a schematic diagram of the computer we have built (see below)



To **download** the compiled design open the Quartus Programmer and set up the programmer chain like this with the default ".sof" file for the project and click **start**



In the next Tutorial we'll write a simple 'C' program to flash the LEDs and read the switches to check everything is working.