Adding and Simulating Custom SoC Peripherals

Version 2.0

March 17, 2014

Corporate HQ & Design Center
380 Stevens Ave. Suite 206
Solana Beach, CA 92075
http://www.macnica-na.com

About Macnica Americas

Macnica Americas is a franchised semiconductor distributor for multiple, high-tech suppliers within North America. Our business model emphasizes unsurpassed technical support and knowledge versus other distribution options at no cost premium. Macnica Americas is the North American based division of Macnica Inc., a $2.4B global leader in semiconductor distribution. We maintain a field support staff as well as centralized design & applications teams.

Optional design services are headquartered in San Diego, CA., USA and offer partial or full turnkey design of FPGAs, power distribution networks, and full PCB design. Our expertise includes all aspects of high speed communications protocols and networking, video broadcast, signal processing, and storage applications. Macnica's specialty is high density, high speed complex FPGA designs utilizing multiple IP cores with fast time to market requirements.

Macnica can help you deliver a winning project with the unique combination of technical support, custom IP, and design services. Setup a meeting today!

http://www.macnica-na.com/web/americas/home

License and Terms of Use

Table of Contents

1 Lab Overview

1.1 Introduction and Goals

This lab is designed as a self-paced-learning tool for basic Altera SoC hardware development. In reaching this goal, only a relatively small subset of SoC features, peripherals, and options are explored. It is highly recommended persons attend additional training, such as that offered by Altera directly, for more detailed education on this rather complex flow and device family: http://www.altera.com/education/training/curriculum/soc/trn-soc.html

The lab is broken into a series of major sections or milestones representing the common phases of design with the Altera SoC product. Unlike other trainings you may have had, this lab does not explicitly indicate every button to push in explicit order. Instead, your goal is described with the necessary information given with exact steps left to the user. If you are having problems, each section concludes with a series of hints related to the tasks proposed. In the event you are still unable to achieve the desired functionality, or you simply wish to double-check your progress, a completed project has been provided with the lab materials in the ***/SoC_3_lab/solution*** folder.

A minimal working knowledge of Quartus is expected. For someone completely new to the Altera FGPA development tools, it is suggested they complete tutorials integrated within Quartus under the ***Help*** *pull-down menu prior to starting this lab.*

All HDL is coded in Verilog HDL, but the logic is so basic VHDL users should not find this at all a limitation in their experience.

1.2 Hardware and Software Requirements

Review vWorkshops_Getting_Started.pdf document for a detailed reference on installing the necessary software and burning the microSD card per following requirement list:

- Macnica Helio SoC Evaluation board w/micro-USB cable
- Quartus 13.1 installed local machine
- ModelSim-ASE 10.1d (Included in Quartus 13.1 installer)
- microSD card loaded with Helio image as documented vWorkshops_Getting_Started.pdf

1.3 Assistance

A dedicated e-mail account has been setup to receive support requests for the vWorkshop series. Please identify the course (in this case: SoC 3 Adding and Simulating Custom SoC Peripherals) in addition to details on the question. workshophelp@macnica.com

1.4 Lab Agenda and Milestones

Below is a listing of the lab agenda and milestones.

Custom Peripheral Creation

Given source code for a rudimentary PWM led driver, you will use the Qsys Component Editor to create a custom component and make it available in the component library of Qsys.

Component Editor Basics

Get just beyond the basics of the Component Editor. You will modify the critical **<component>_hw.tcl** file directly and add a parameterized feature.

BFM Simulation

Using Altera's BFMs in Qsys, you will create a test system for simulation of your custom component in order to verify its operation. Due to limitations of time and scope, a simplified "user test program" driving these BFMs is provided – you can modify it for more features or functions such as automated checking at your leisure.

Project Integration

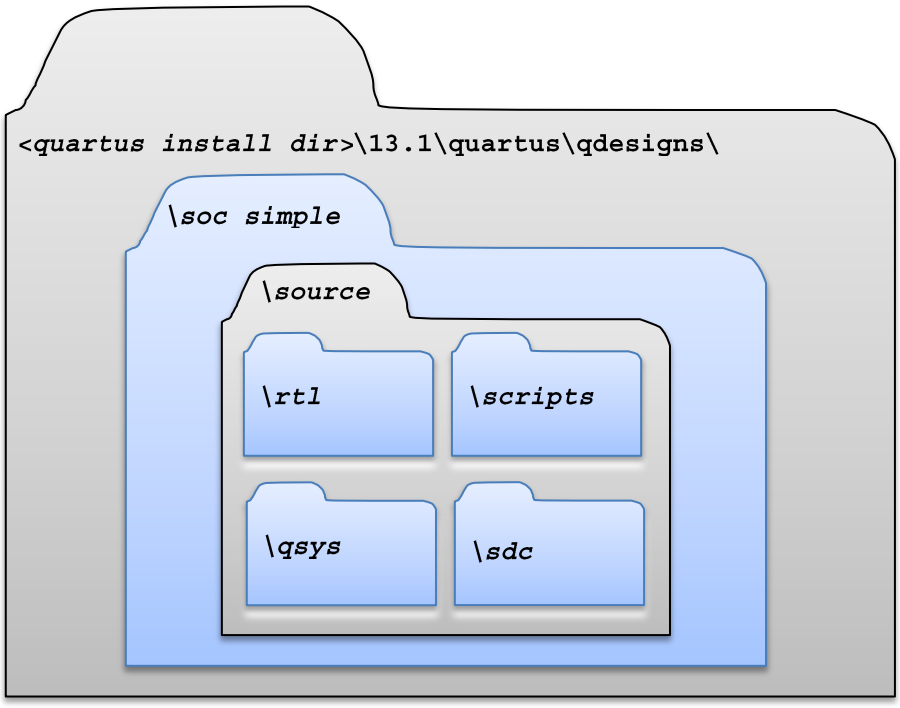
The custom component will then be integrated into our simple SoC Quartus project from the Basic Altera SoC HPS Usage training and compiled. This will require some minor edits to the project files and assignments.

Interaction and Verification

Finally, you will load the compiled design into the FPGA, and use SystemConsole to interact and verify functionality of the custom component in hardware.

2 Lab Instructions

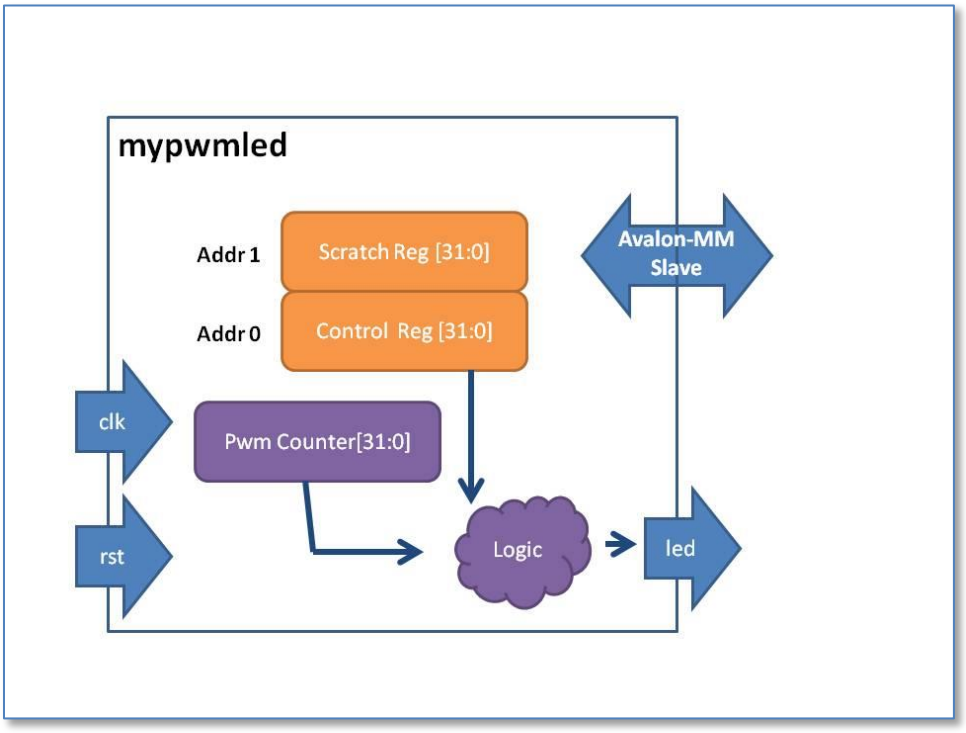
- ☐ We will be using the same folder structure and contents as that used in the prior lab (graphic below). If you did not complete the Basic Altera SoC HPS Usage lab, use the contents of the ***/Soc_3_lab/resources/soc_simple.zip*** to create this structure and align with the completion of the SoC 2 Basic Altera SoC HPS Usage lab.



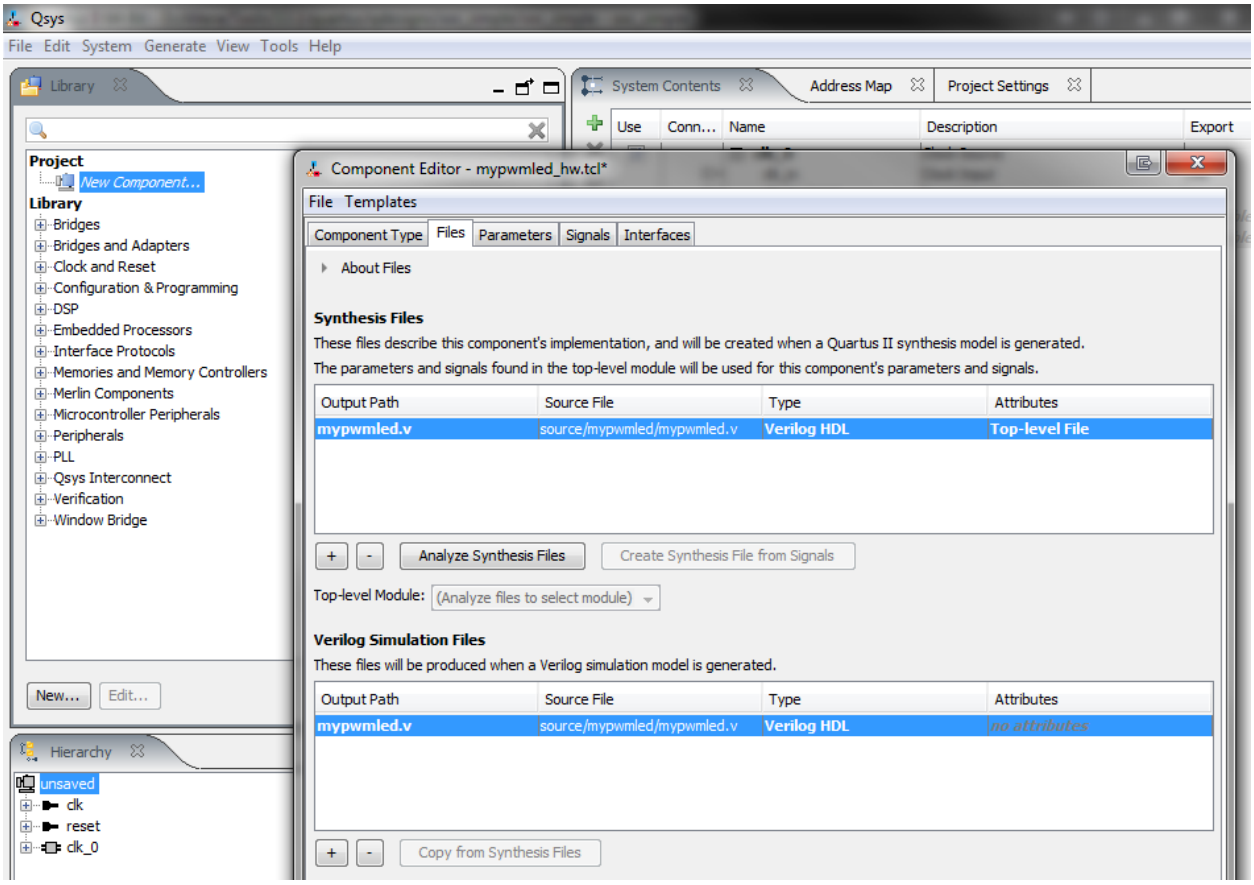
2.1 Custom Component Creation

The HDL source code for a very simple PWM function has been provided for you, but we will need to define how that source code, specifically its ports, are mapped to the Qsys standard interface and signal types.

- ☐ Create a new folder within ***~/soc_simple/source*** called **mypwmled** (not shown in graphic above). Copy the file **mypwmled.v** from the ***/SoC_3_lab/resources*** folder into ***/soc_simple/source/mypwmled***. This will be the source for our custom component in this lab.
- ☐ Review the source file ***~/soc_simple/source/mypwmled.v*** and compare to the simple block diagram below to familiarize yourself with its operation. This is about as simple a custom component as could be created for lab purposes – it has a 1 bit address space and a simple comparison between a running counter and control register to toggle the state of an output signal and "pwm" a LED. You might notice a parameter purposefully commented out – we will activate that later.



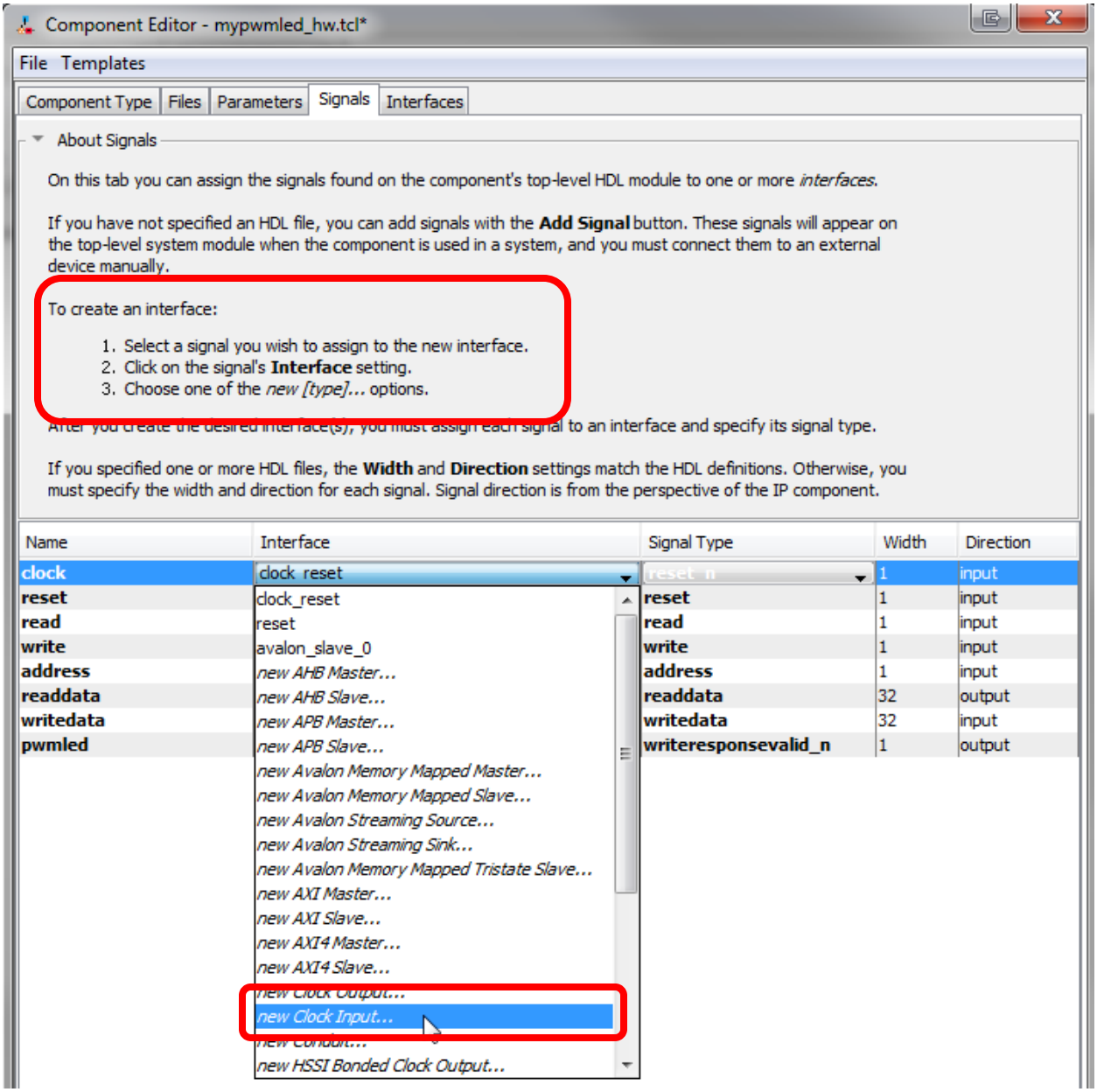
- ☐ Launch Quartus and open the project file ***/soc_simple/soc_simple.qpf***. Launch Qsys, but ***do not*** open the **qsystem.qsys** file as we have done in the past at this time. You will see why this is important later.
- ☐ Double-click **New Component** in the component library to invoke the **Component Editor**. Set **mypwmled** as both the name and display name for the custom component being created leaving all else default. On the **Files** tab, add the source file ***/soc_simple/source/mypwmled/mypwmled.v*** file as the synthesis ***and*** simulation source.



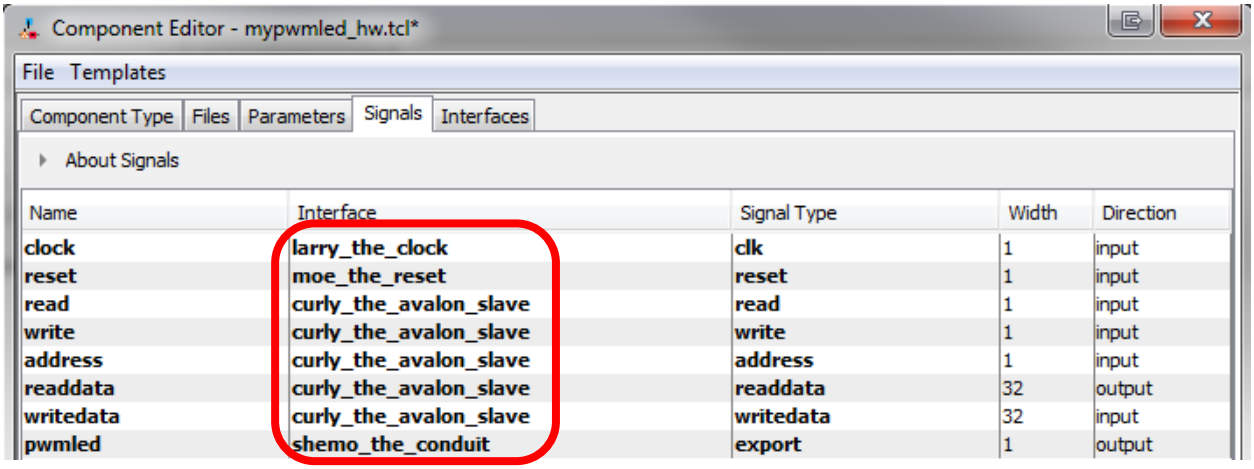
- ☐ Invoke **Analyze Synthesis Files** on the **Files** tab. After it completes, look at the resulting **Interfaces** tab and **Signals** tab. Given your knowledge gained in presentation and demo, fill out the table below and identify what might not "look right."

Interface Name	Signal Type in Interface	HDL port name	Looks "Right"?
		reset	
		read	
		write	
		address	
		readdata	
		writedata	
		clock	
		pwmled	

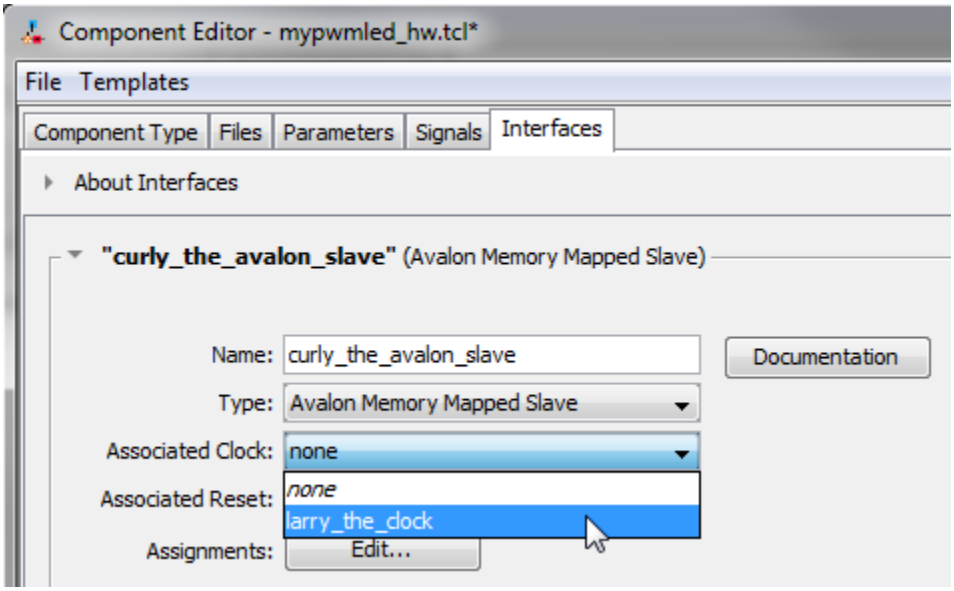
- ☐ Correct the interfaces such that you have at least one each of the following interface *types*: one **Avalon-MM Slave**, one **Clock Input**, one **Reset Input**, and one **Conduit_End**. This can be tricky – the best method is often using the **Signals** tab to create one or more interfaces of the desired type. In the graphic below, a new **Clock Input** interface is being created.



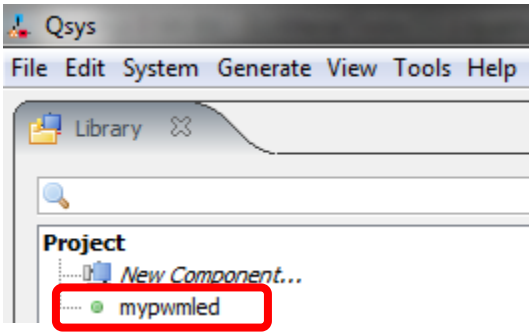
- Once an interface exists, you can rename it on the **Interfaces** tab. The **Signal** tab shows the interface *name*, not necessarily the *type*. Though newly created interfaces get a descriptive name by default the name does not matter -- so you can be creative if you wish and I have done so for purposes of demonstration☺. Make sure all HDL port names shown are mapped to an interface and the signal type within that interface matches the function and polarity you can infer from the HDL port name. Refer to the below picture as a guide. (Note: the conduit signal type is a bit odd and identified as **export**)



- After the prior steps, you might have "left over" interfaces on your **Interfaces** tab with no HDL signals/HDL ports assigned. There is a convenient button **Remove Interfaces with No Signals** on the **Interfaces** tab just for this purpose.
- Finally, most interface types (for example, an Avalon-mm slave) require association with a clock and reset signal. This is done on the **Interfaces** tab you should have error messages regarding them missing. Assign the correct clock and reset to the Avalon slave, reset and conduit interfaces. There should no remaining errors in the status window when complete.

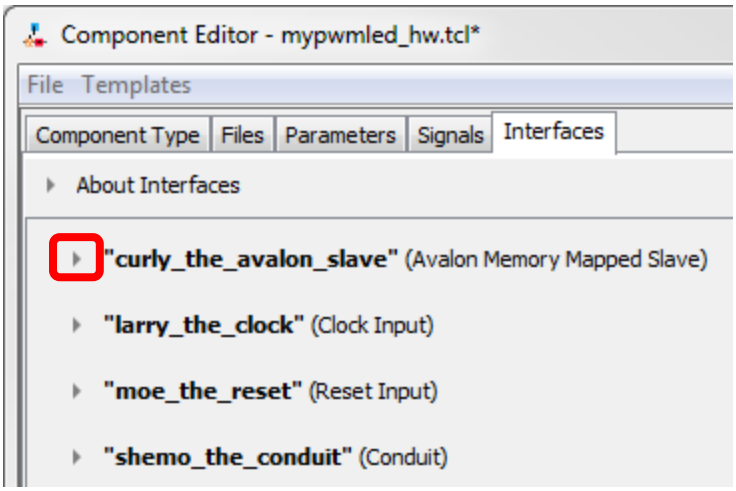


- Aside from association with a clock and reset, you will *not* need to change any other component properties for interfaces from their defaults (for example, words-vs-byte transfers, timing, pipelining, etc). Just be aware that, for a more complex component, you do have access to such parameters to match the functionality of the HDL.
- Close and save your component a final time. Your custom component now appears in the component library. Close Qsys.



Hints: Custom Component Creation

- *The Interfaces tab can be made more manageable by minimizing the detail properties expanded by default. When minimized, you see just the interface **name** and **type**.*



- *Upon first analysis of the HDL, the component editor incorrectly creates two interfaces of type **Reset Input** and no interface of type **Clock Input**. It also doesn't know what to do with our design-specific output port **pwmled**, so it incorrectly associates it with the **Avalon-MM Slave** interface called **avalon_slave_0**. These are the errors we need to correct.*
- *When saved, your component will appear in the **Project** area of the component library because we did not give it a folder/subfolder name for filing within the **Library**.*
- *To edit your saved component, you can use the right-click menu with the component selected.....double-click adds the component to your system and single-click just highlights. You can also use the Edit button at the bottom of the Component Library.*

2.2 Component Editor Basics

Now we will look at the output of the Component Editor and understand how this file interacts with the Qsys component library and HDL source code. We will also enhance our component with the addition of a parameter that can be configured when instanced in a system.

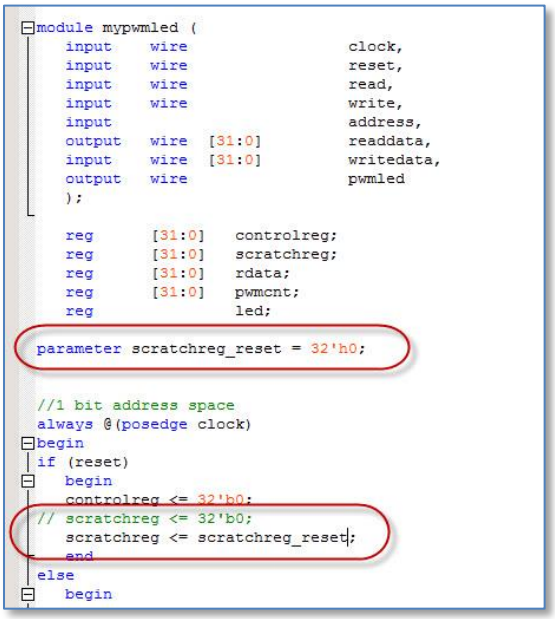
- ☐ Locate the **mypwmled_hw.tcl** – this is the output file of the Component Editor created when you hit **Finish** and save. Notice that it is not in the same folder as the source code. This is a bit undesirable – the location is a result of the "path focus" Qsys had when we invoked the

Component Editor. Specifically, we had no Qsys system open so our "focus" was in the default project directory. Move (don't copy/paste) **mypwmled_hw.tcl** to the ***soc_simple/source/mypwmled*** folder so it is in the same location as its source. Launch Qsys (or do a System Refresh via the **File** pull-down menu if you never closed it). Do you see a change in the availability of your new component in the library? Do you know why?

- ☐ Add the new location of the **mypwmled_hw.tcl** file to the Qsys search path via **Options** under the **Tools** pull-down menu in Qsys. Be sure to hit **Finish** and not just close the window or your changes will not take effect. A refresh of the library occurs automatically. Is the new component available in the library now? What happens if you try to edit the component again? (Close the component without saving when you do this!). Any theories based on the error message? (Try viewing the contents of **mypwmled_hw.tcl** *in a text editor to try and identify the problem).*

- ☐ It will take two simple edits of the **mypwmled_hw.tcl** file for it to operate properly in its new location. After edit, you must perform a **System Refresh** for this file to be re-parsed. Try to fix things and return expected to proper operation – refer to tips if necessary.

- ☐ Now edit the **mypwmled.v** file and, per the picture below, make a total of three changes to comment/uncomment the declaration of **scratchreg_reset** and its use to set the value of **scratch_reg** on a reset. In Qsys, open the **mypwmled** component for editing and reanalyze the source files to bring in this change. Notice the change on the **Parameters** tab of the Component Editor. Click **Preview the GUI** to preview what a user will see adding this component to their system. Click **Finish** and save this change to the component which will create a new version of **mypwmled_hw.tcl**. Review the impact of adding this parameter to the **mypwmled_hw.tcl** with your favorite text editor.



```
module mypwmled (
    input    wire                clock,
    input    wire                reset,
    input    wire                read,
    input    wire                write,
    input                        address,
    output   wire  [31:0]        readdata,
    input    wire  [31:0]        writedata,
    output   wire                pwmled
    );

    reg      [31:0]   controlreg;
    reg      [31:0]   scratchreg;
    reg      [31:0]   rdata;
    reg      [31:0]   pwmcnt;
    reg               led;

  parameter scratchreg_reset = 32'h0;

  //1 bit address space
  always @(posedge clock)
begin
  if (reset)
    begin
    controlreg <= 32'h0;
  // scratchreg <= 32'h0;
    scratchreg <= scratchreg_reset;
    end
  else
    begin
```

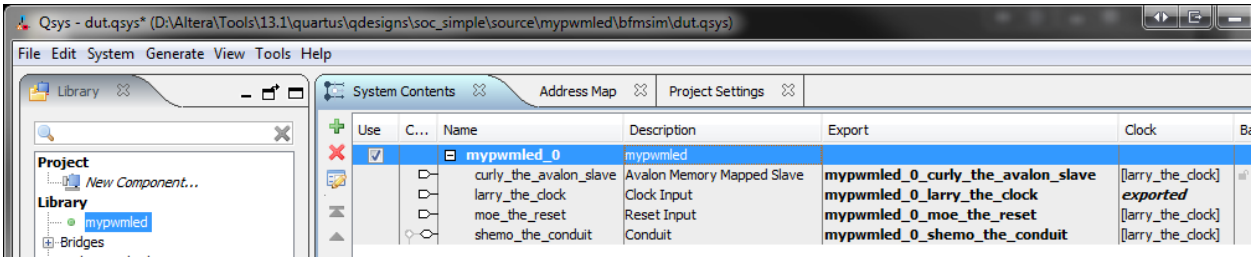
Hints: Component Editor Basics++

- *After moving the file **mypwmled_hw.tcl**, it is no longer in the default search path of Qsys (refer to presentation on paths searched by default). Qsys rebuilds the component library on launch or after a manual refresh, so the component disappears from the library until we include its new location into the search path.*
- *After the search path is fixed, the paths to **mypwmled.v** defined in **mypwmled_hw.tcl** itself need to be updated. Without correction, Qsys locates the component and populates it into the library, but gives errors when we try to edit the file (or if we were to try and generate the system). Delete the leading **source/mypwmled** from **mypwmled_hw.tcl** in two places which are no longer necessary with the **.v** file now in the same directory as the **_hw.tcl** file.*
- *Because the Component Editor is a bit fickle and requires a fair bit of manual corrections, you are probably starting to understand why many users begin to just create/edit **<component>_hw.tcl** files directly. The syntax is relatively straightforward and well defined in documentation: http://www.altera.com/literature/hb/qts/qsys_tcl.pdf*
- *Advanced users can do much more than enable simple parameters. Refer to Altera documentation for tcl-function callback support such as data-entry validation and more.*

2.3 BFM Simulation

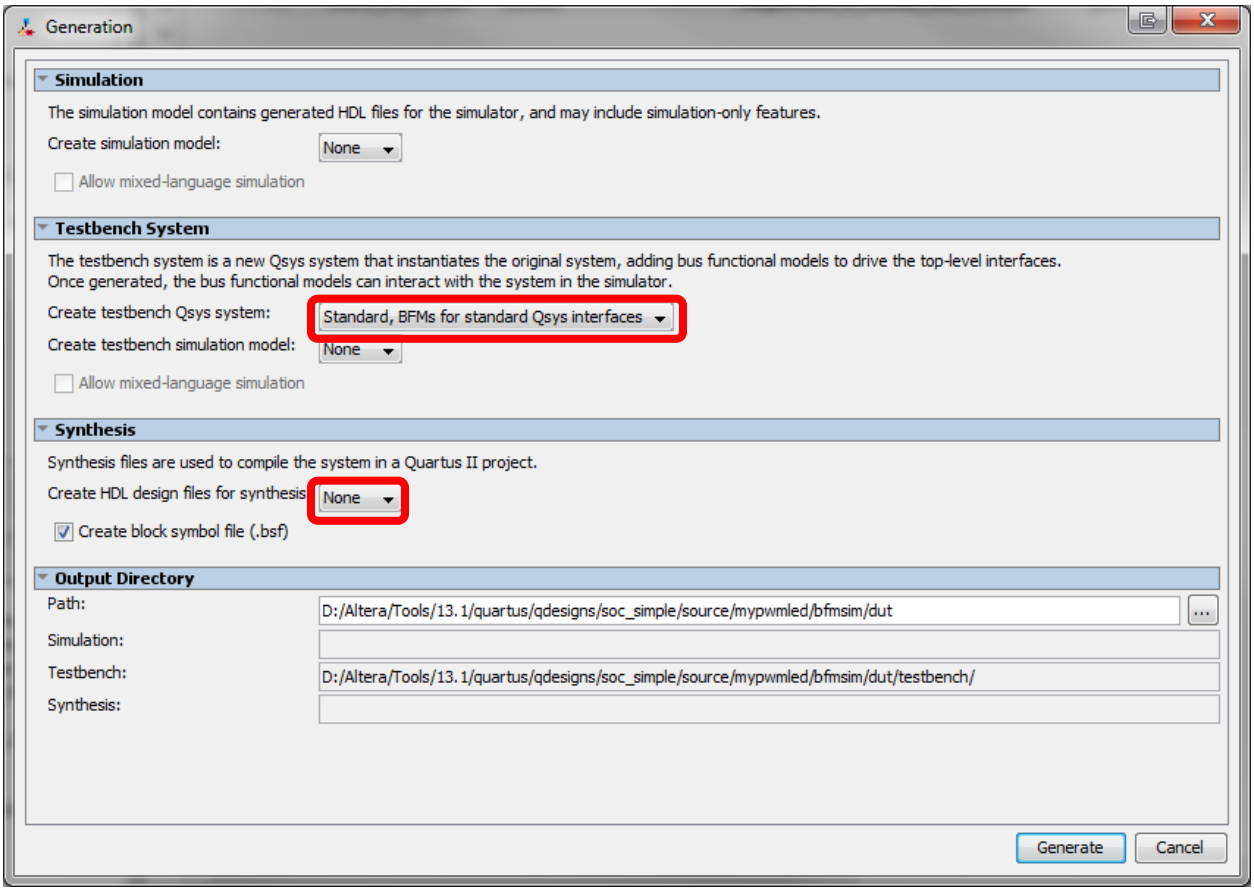
We will now use Qsys to create a BFM-based test bench for our component so that we can verify its functionality in a simulation software package. Since we are using a standardized interface, it is typically sufficient (and certainly much faster) to simulate our component alone with BFM modeling of its connections into a system rather than simulating all system components fully and simultaneously.

☐ Create a folder ***/bfmsim*** inside ***/soc_simple/source/mypwmled***. Invoke Qsys, create a "new" system, and save it as **dut.qsys** in the folder ***/soc_simple/source/mypwmled/bfmsim***. Delete the default **clock** component present and add the **mypwmled** component to the system. (You are prompted by your previewed GUI for the **scratchreg_reset** parameter value. Enter something memorable and take note for later comparison). Now export all signals with their default names by double-clicking in the **Export** column. See graphic below for this step completed – and at the hints section if your attempt looks wildly different and you can't figure out why!

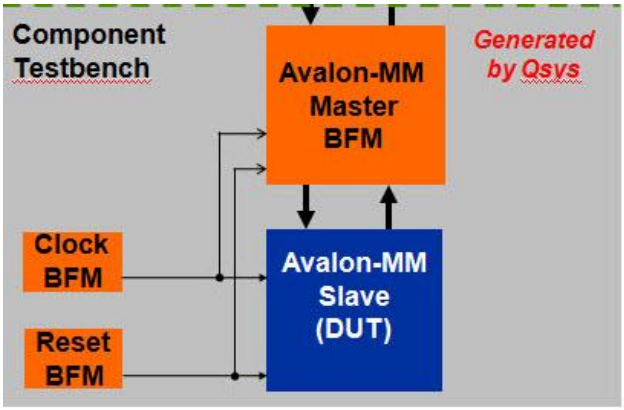


☐ Use the **Generation** pull-down menu and set options to **Create a Testbench Qsys System** = **Standard, BFMs for standard Qsys interfaces** in **Verilog**. (We can also disable generation of files for synthesis by selecting **None** in that section to save time; we are doing sim only for this system and not synthesizing the design to put in to the FPGA itself). You might also notice the

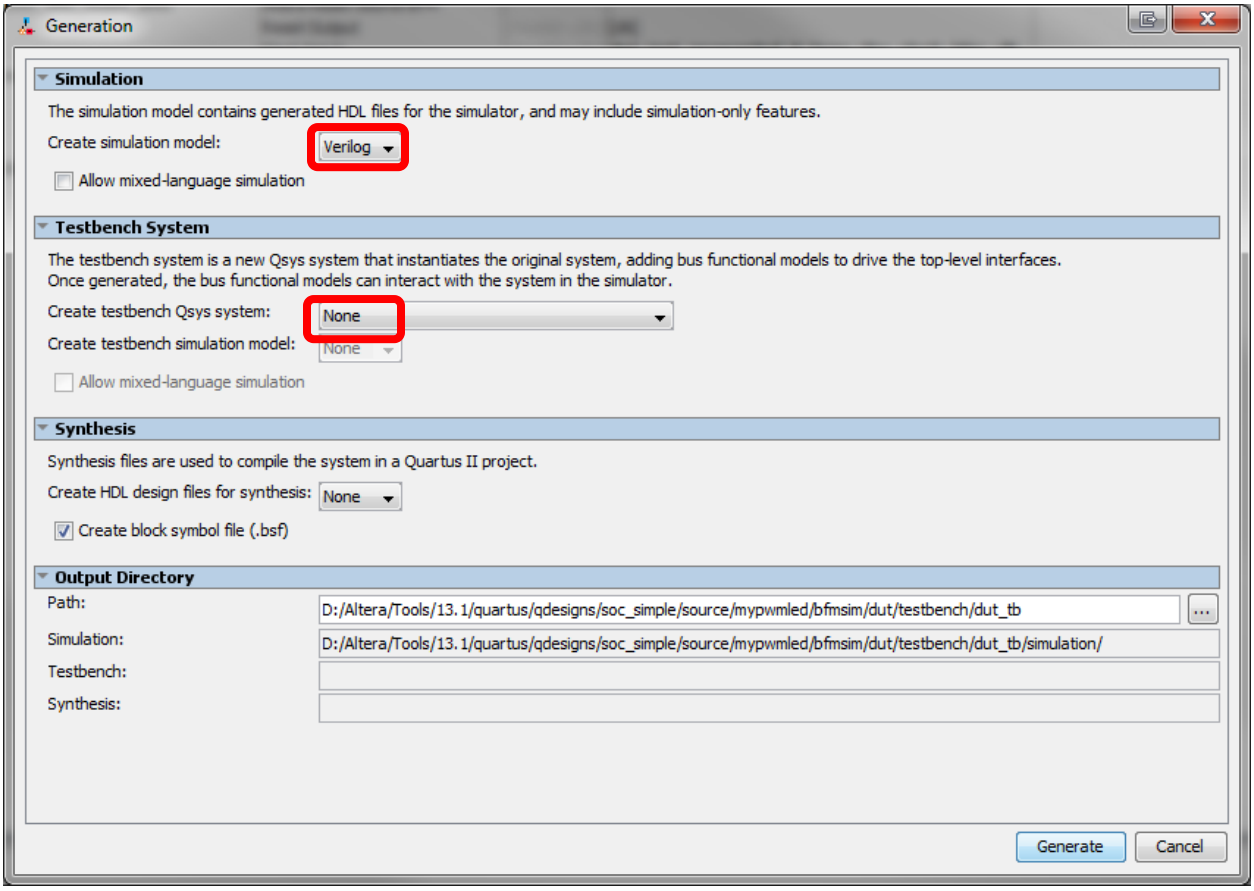
output directories identified – we are accepting the defaults. Press **Generate** to create the output files selected.



- Generation creates a new Qsys file called **dut_tb.qsys**. Locate, open and examine the contents of this system with Qsys. Notice that our original system of single component under test (**dut.qsys**) is instanced in this new system (**dut_tb.qsys**). This is an example of a Qsys system instancing another Qsys system – neat! Notice also the component called **dut_inst** has the appropriate Altera BFM components instanced and connected on each of its interfaces. Below is a simplified picture from the presentation slides of this test bench system. (Aside: we could have achieved the same result by manually instancing and connecting the appropriate BFMs in the original system **dut.qsys**, but this also requires manually configuring many parameters of each BFM component to match the mode used. Having Qsys generate the test bench system for us does this "automagically"!).



- Now generate the simulation model of the test bench system. Using the **Generation** pull-down menu option of the **dut_tb.qsys** system select **Verilog** as a simulation model and disable **Synthesis** and **Testbench** per the below screenshot (the latter two not actually necessary, but speeds up generation since we are using this system for simulation purposes only).



- The generated simulation model creates a compilation script for most popular simulators (Aldec, Cadence, Mentor, Synopsys). We will use the provided Mentor compilation script as a base and tweak it just slightly. Specifically, edit ***/bfmsim/dut/testbench/dut_tb/simulation/mentor/msim_setup.tcl*** so that all design files (not

device library files) compile into the same working library called **work**. The below image shows the necessary edit. *(Caution: if you regenerate the testbench your edits here will be overwritten, but it is unusual to repeatedly generate the testbench).*

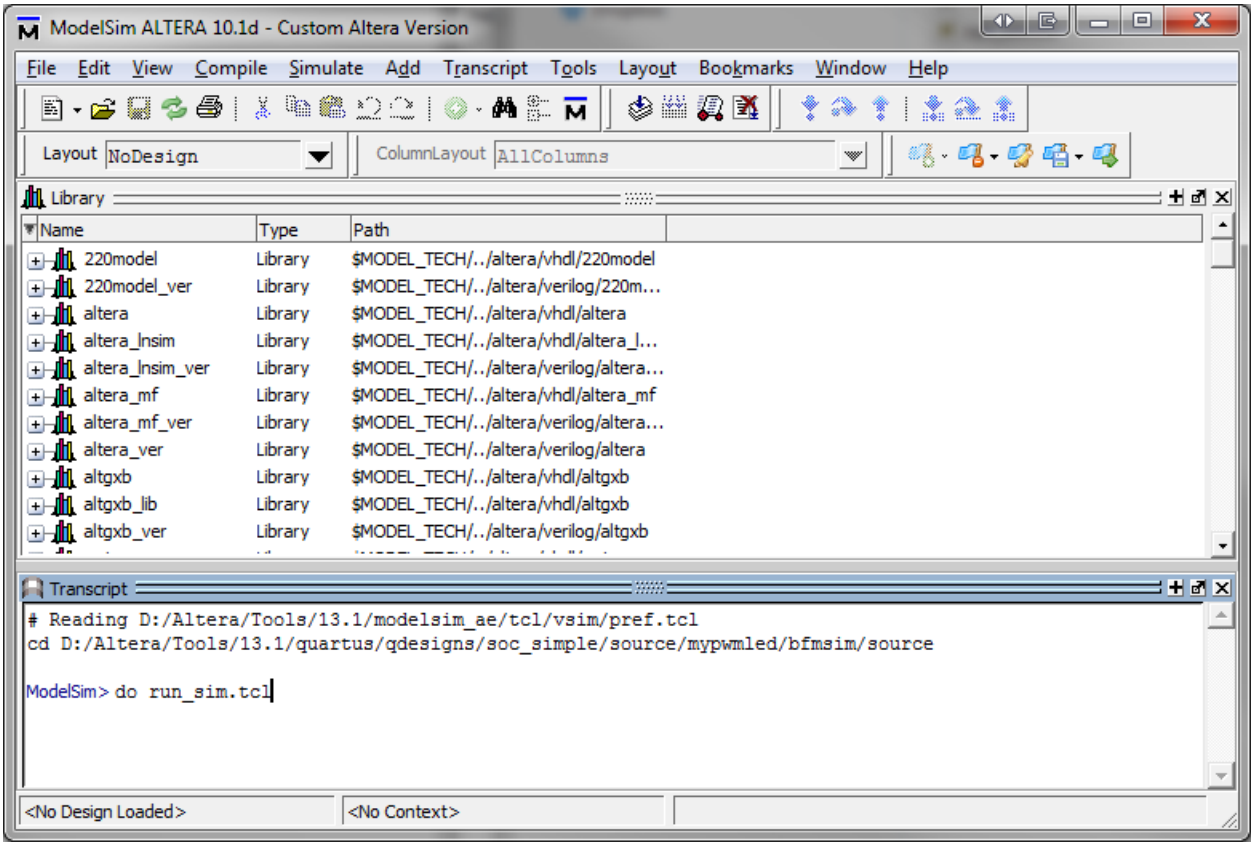
```
114  # ----------------------------------------
115  # Compile the design files in correct order
116  alias com {
117    echo "\[exec\] com"
118    vlog -sv "$QSYS_SIMDIR/submodules/verbosity_pkg.sv"                                    -work work
119    vlog -sv "$QSYS_SIMDIR/submodules/avalon_mm_pkg.sv"                                    -work work
120    vlog -sv "$QSYS_SIMDIR/submodules/avalon_utilities_pkg.sv"                             -work work
121    vlog     "$QSYS_SIMDIR/submodules/mypwmled.v"                                          -work work
122    vlog -sv "$QSYS_SIMDIR/submodules/altera_conduit_bfm.sv"          -L altera_common_sv_packages -work work
123    vlog -sv "$QSYS_SIMDIR/submodules/altera_avalon_mm_master_bfm.sv" -L altera_common_sv_packages -work work
124    vlog -sv "$QSYS_SIMDIR/submodules/altera_avalon_reset_source.sv"  -L altera_common_sv_packages -work work
125    vlog -sv "$QSYS_SIMDIR/submodules/altera_avalon_clock_source.sv"  -L altera_common_sv_packages -work work
126    vlog     "$QSYS_SIMDIR/submodules/dut.v"                                               -work work
127    vlog     "$QSYS_SIMDIR/dut_tb.v"
128  }
129
```

- ☐ We must now have a test program that interacts with the BFMs of the **dut_tb.qsys** using their API calls. People make careers out of verification and creating a robust test program with automated checking is well beyond the scope of this lab ... but an ultra-simple test program has been provided for you. Create the folder ***/source*** inside ***/soc_simple/source/mypwmled/bfmsim***. Copy **test_program.sv** into this folder from ***SoC_3_lab/resource***. Review the copied file in a text editor; you should be able to follow its basic operation using the comments provided.

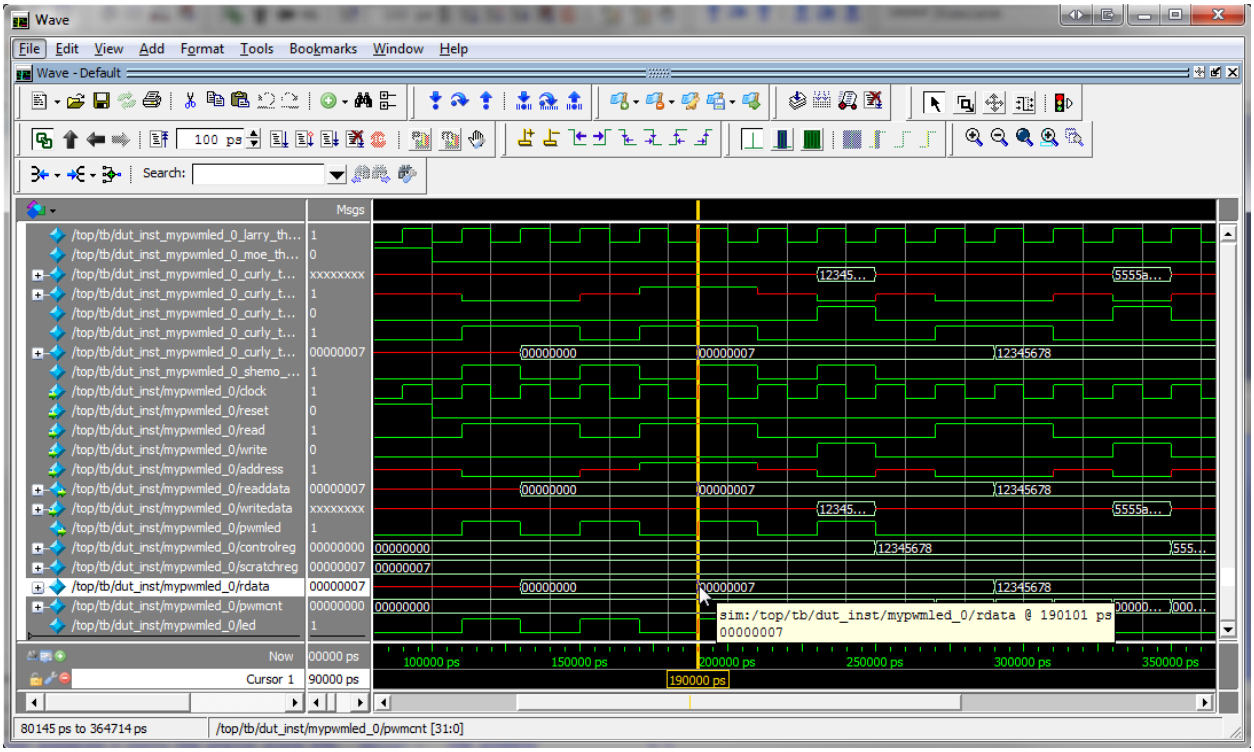
- ☐ Next, we need a single top-level file that instances both our test program ***test_program.sv*** and the testbench **dut_tb.qsys**. This is a very, very simple task...but to prevent typos a top-level has been created for you. Copy **top.v** from the folder ***/SoC_3_lab/resource*** folder into ***/mypwmled/bfmsim/source***. Review the file contents to fully comprehend its simplicity.

- ☐ Lastly, we will use a short, custom tcl script to execute our simulation from the ModelSim prompt. Copy **run_sim.tcl** from ***/SoC_3_lab/resource*** folder into ***/soc_simple/source/mypwmled/bfmsim/source*** and review its contents. It is largely calling macros defined by the Qsys compilation script **msim_setup.tcl** to compile the **tb_dut.qsys** simulation files generated prior.

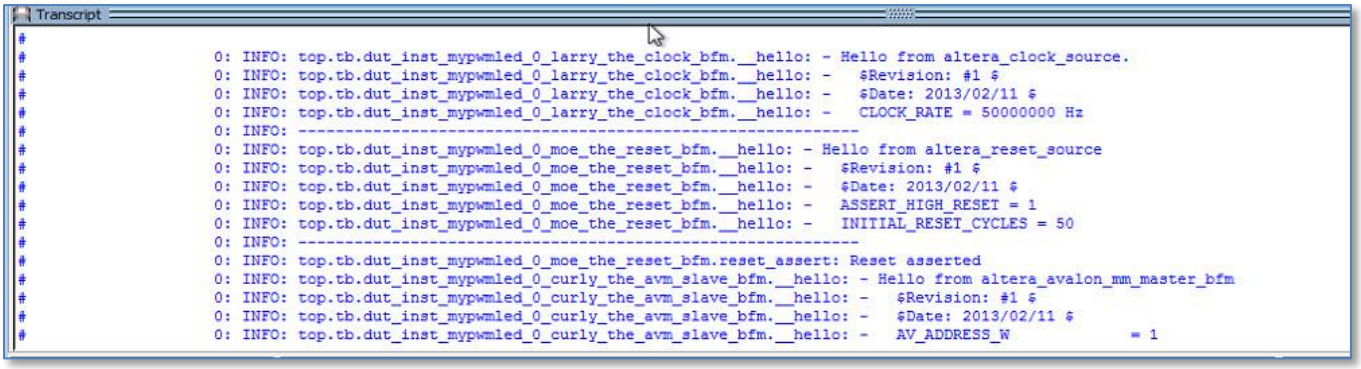
- ☐ Invoke **ModelSim-ASE** and change-directory to ***/soc_simple/source/mypwmled/bfmsim/source*** which is where the sim execution script **run_sim.tcl** was placed. Source this execution script with the command **do run_sim.tcl** at the ModelSim prompt.



- ☐ You should see compilation messages scroll by and then the simulation run. In the waveform window, select all signal names and use the right-click menu to change their radix to hex for easier reading (they are binary by default). You will probably want to zoom in/out as well…..but you should be able to identify your custom scratch register reset value and the response of the custom component to the writes and reads defined in the file **test_program.sv**.



☐ View the transcript window and take note of the messaging from the BFM models. The amount of information output is controlled by the verbosity setting applied in the user test program **test_program.sv**.



Hints: BFM Simulation

- *When creating your test bench system and adding your component, if you don't see all your component interfaces for export it is most likely because you still have an interconnect filter applied. For example, you could be hiding Clock and Reset interface types. Apply the interconnect filter **Default** to resolve.*
- *The Qsys interface names match the naming made with the component editor. In the project solution and screenshots I call attention to this by including "larry, moe, curly, and shemp" in my interface naming during component creation. The order of the interfaces in the GUI also depends on their definition order in the **myledpwm_hw.tcl** file. Your component may be*

published with different interface naming or interface order. As long as the interface number and types are the same, the order or name themselves do not matter and just impact system readability.

- *When editing **msim_setup.tcl** there is a **device_library_files** section and **design_files** section. Only edit the **design_files** section with the library **–work work** modification.*
- *If you used your own naming convention for the interfaces, you will need to modify the test_program.sv file to reflect those names. Lines 15 -17.*
- *ModelSim-ASE (Altera Starter Edition) requires no license. This is a separate simulation tool by 3rd party (Mentor) and separate from Quartus though Altera distributes/provides the install files. You can get a comparison of ModelSim-ASE to other variants here: http://www.altera.com/products/software/quartus-ii/modelsim/qts-modelsim-index.html*

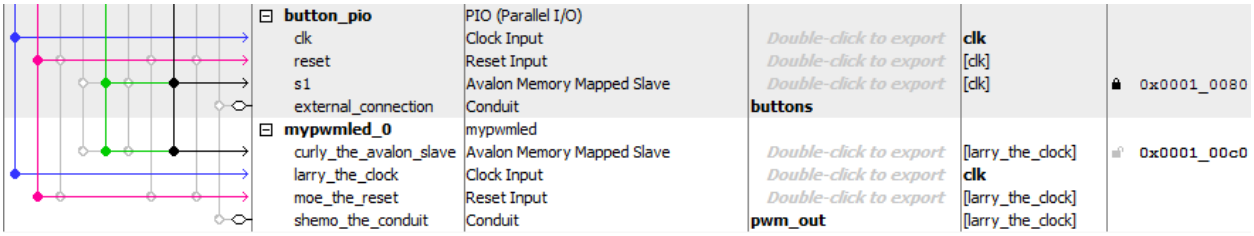
2.4 Project Integration

We will now bring the custom component into our simple SoC system. Since we are limited in the number of LEDs available, we will rob the most significant led bit (LED3) from GPIO control and instead drive it with the new custom component **mypwmled**.

- ☐ Open the **qsystem.qsys** file located in ***soc_simple/source/qsys*** system which was the focus of the previous lab. (Don't forget you can use your interconnect filters to simplify the system view). Add the **mypwmled** component and accept the default component name. Make connections from this component into the system using the info in below table and graphic:

Component Name	scratchreg_on _reset	Avalon-MM slave interface	Clock Interface	Reset Interface	Conduit Interface
mypwmled_0	something memorable	Same as button_pio components	Same as button_pio components	Same as button_pio components	Export as **pwm_out** (not the default)

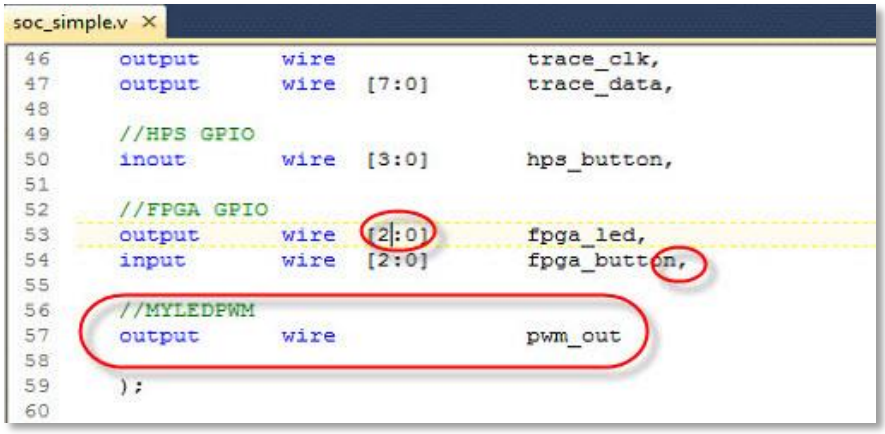
- ☐ Edit the base address for the **mypwmled** component to be **0x000100C0** which is important to note for our peeking/poking later.

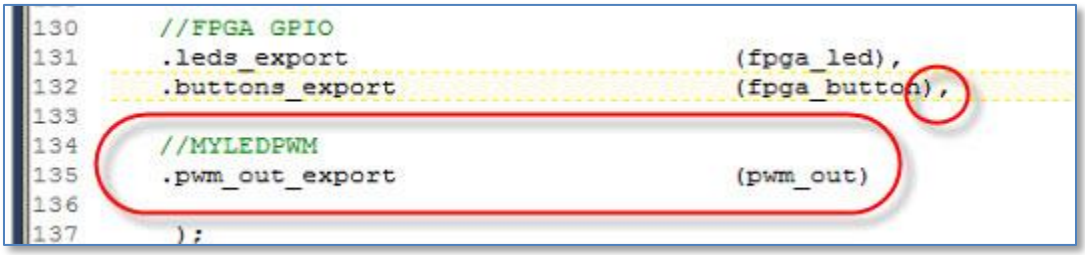


- ☐ Use the **Generate** pull-down menu to create the HDL Example and confirm that there is now a top level HDL connection called **pwm_out_export**. This didn't exist prior, so we'll need to update our top-level Quartus project soon.

- ☐ You should have no errors identified in the message window. Use the **Generate** pull-down menu to enable **Verilog** output in the **Synthesis** section. Simulation and Testbench generation can be disabled to save time. Click **Generate**, saving the system if necessary. Hopefully you have no errors!

- ☐ Edit the **soc_simple.v** file located in ***/soc_simple/source/rtl*** to add the new output **pwm_out** to both the top-level port list and the instance of the Qsys system. We will also reduce the top-level **fpga_led** bus from 3 to 2 bits. For those unfamiliar with Verilog, use the below pictures as a guide. Don't forget the trailing comma modification circled.





- ☐ We now need to assign this new output to a pin. Launch the **Assignment Editor** (under the **Assignments** pull-down menu in Quartus) and sort by name. Find the location assignment for **fpga_led[3]** which is currently to **PIN_AC4**. Edit the name from **fpga_led[3]** to **pwm_out**. This pin/led will now be driven by our new **mypwmled** component instead of the **led_pio** component. We'll leave this bit of the **led_pio** dangling – it won't harm anything.

- ☐ Perform a full compilation of the project. Verify that a new **soc_simple.sof** file was created by viewing its timestamp. In the compilation report notice that the Timequest section is now red – we'll ignore for purposes of this lab, but do you know why?

Hints: Project Integration

- *Revert your interconnection filter to **Default** at least once to view all connections*
- *We don't use the default export name for the **mypwmled** conduit because everyone may have named the interface differently in component generation -- this impacts default naming.*
- *Astute students will notice we didn't make an I/O voltage assignment to pwm_out. The project is set to have 2.5V be the default unless otherwise specified....we're relying on that.*

- *Timequest shows red because not all I/O are fully constrained. Specifically, we did not make any timing assignments for the new pwm_out port. This will function because this output is largely static. We would make assignments to remove this path from timing analysis to resolve the error.*

2.5 Interaction and Verification

Now let's use SystemConsole as we did in SoC 2 lab to peek/poke our system and verify operation of our new component hardware at location.

- ☐ This is largely a repeat of the previous lab with the added bit of detail that our custom component is at location **0x000100C0** and a memory space of two 32-bit words.
- ☐ Perform a 32bit peek to address **0x000100C4**. This is the scratch register. Does its value match your expectations?
- ☐ Now do a poke to address **0x000100C0** of **0x02FAF080** and observe LED3. Any idea on why this value specifically? (What is it in decimal, what frequency is our **myledpwm** component operating, and why is this significant? Maybe look again at the **myledpwm.v** source file....)

Hints: Interaction and Verification

- *If you haven't completed the previous lab, you can probably get good results by following the screenshot above.....but you should refer to that lab for more detail.*
- *SystemConsole (as a master) uses byte-addressing even though the slave is only lone address bit (word addressed). This is why our addressing isn't 0x000100C0 and 0x000100C1 when peeking/poking from SystemConsole.*
- *Remember we used a parameter for the reset value of the scratch register that could be user-edited using the Qsys GUI. You should be reading your custom value here.*
- *0x02FAF080 is 50M in decimal which is also the clock frequency driving the comparison counter. To get a 1sec LED toggle, we want the comparison counter equal to the operating frequency. You might try writing in a value in 2x or 4x to get a slower flash.*

3 Notes

Document Revision History

Revision	Date	Comments
0.1	May 8, 2013	Initial Draft
0.2	May 27, 2013	Internal Review
1.0	May 29, 2013	Customer Release
2.0	March 17, 2014	Updated to Q-II 13.1