

My First Qsys System

PDF created: December 13, 2017
Validated using tools release: 17.0.0

Contents

Overview	1
Prerequisites	1
Creating the Qsys System	2
Integrating the Qsys System into the Intel Quartus Software Project	13

Overview

This tutorial demonstrates how to use the Qsys tool to easily create an FPGA design using IP available in the Intel® Quartus® software IP catalog. Qsys speeds embedded system design by standardizing the interconnect between IP blocks and allowing users to create their own IP blocks for reuse in their systems.

Prerequisites

The following are required:

- Windows* or Linux* development host PC
- Installed Intel Quartus Prime Software. Either the Lite or Standard Edition, but not the Pro Edition.
- Completed Intel Quartus software project from “How to Program Your First FPGA Device”
 - Either follow the tutorial steps presented [here](#).
 - Or, you can download an archive of the required contents from that tutorial to your local file system [here](#).

Note: User experience may vary when using earlier or later versions of Intel Quartus software. Screenshots in this document are based on the 17.0 release.

Creating the Qsys System

This section will describe the steps required to create a Qsys system. It assumes that you have the completed **blink** Intel Quartus software project from the “How to Program Your First FPGA Device” tutorial open.

Step 1. With the **blink** Intel Quartus software project from the previous tutorial open in the tool, start by launching Qsys by clicking the Qsys button on the toolbar, or select Qsys from the **Tools** menu.

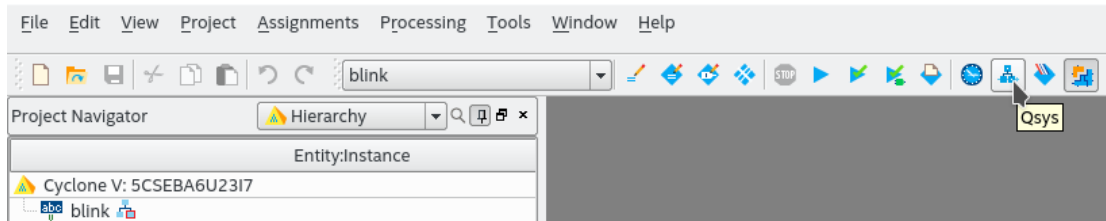


Figure 1: Qsys Button on Intel Quartus Software Toolbar

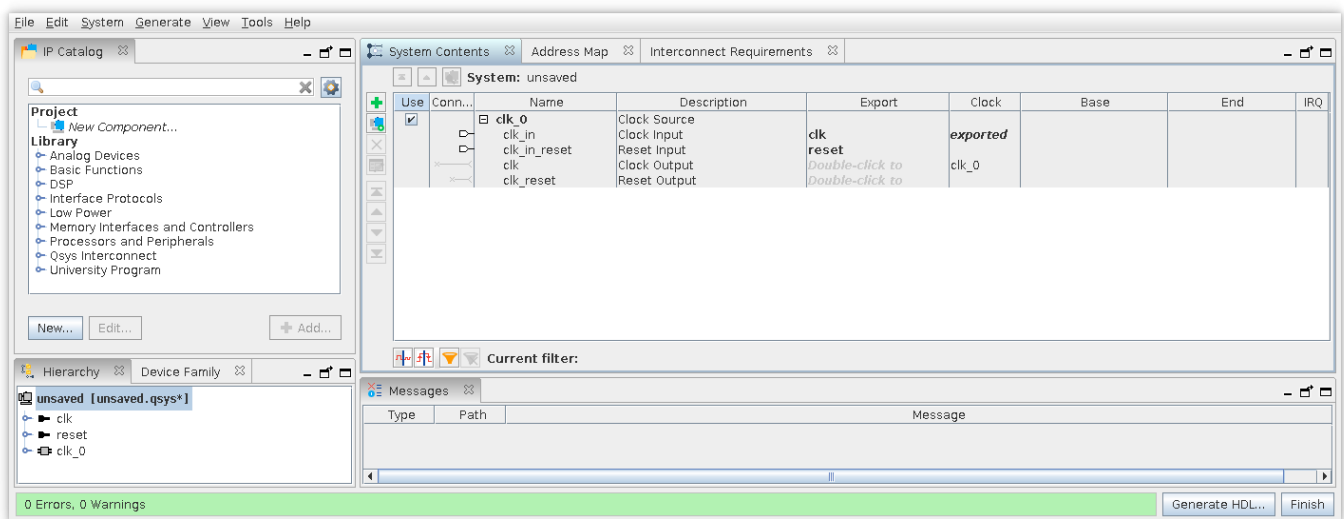


Figure 2: New Empty Qsys System

Step 2. Qsys opens to an unsaved system, with one clock source component already instantiated. To add another IP block, use the **IP Catalog** pane in the left hand column. In the search bar, start typing **on-chip memory** until you see the **On-Chip Memory (RAM or ROM)** IP appear.

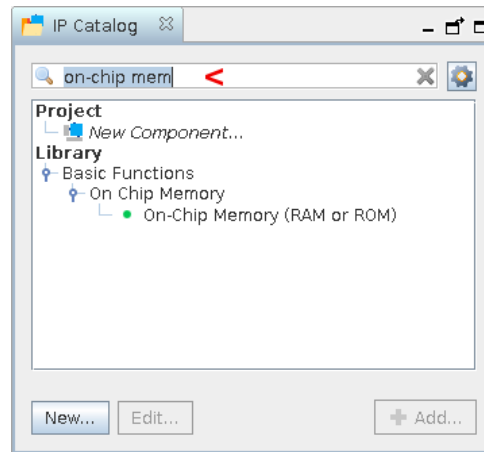


Figure 3: Search for On-Chip Memory Component

Step 3. Either double click on the On-Chip Memory component or select it with the mouse and click the **Add...** button to insert a new instance into the system. This opens the parameter dialog for the On-Chip Memory IP. Change the **Slave S1 Data Width** to **64**, and change the **Total Memory Size** field to **65536** to make this a 64-kilobyte on-chip memory that supports 64-bit data accesses, then click the **Finish** button at the bottom right of the dialog box.

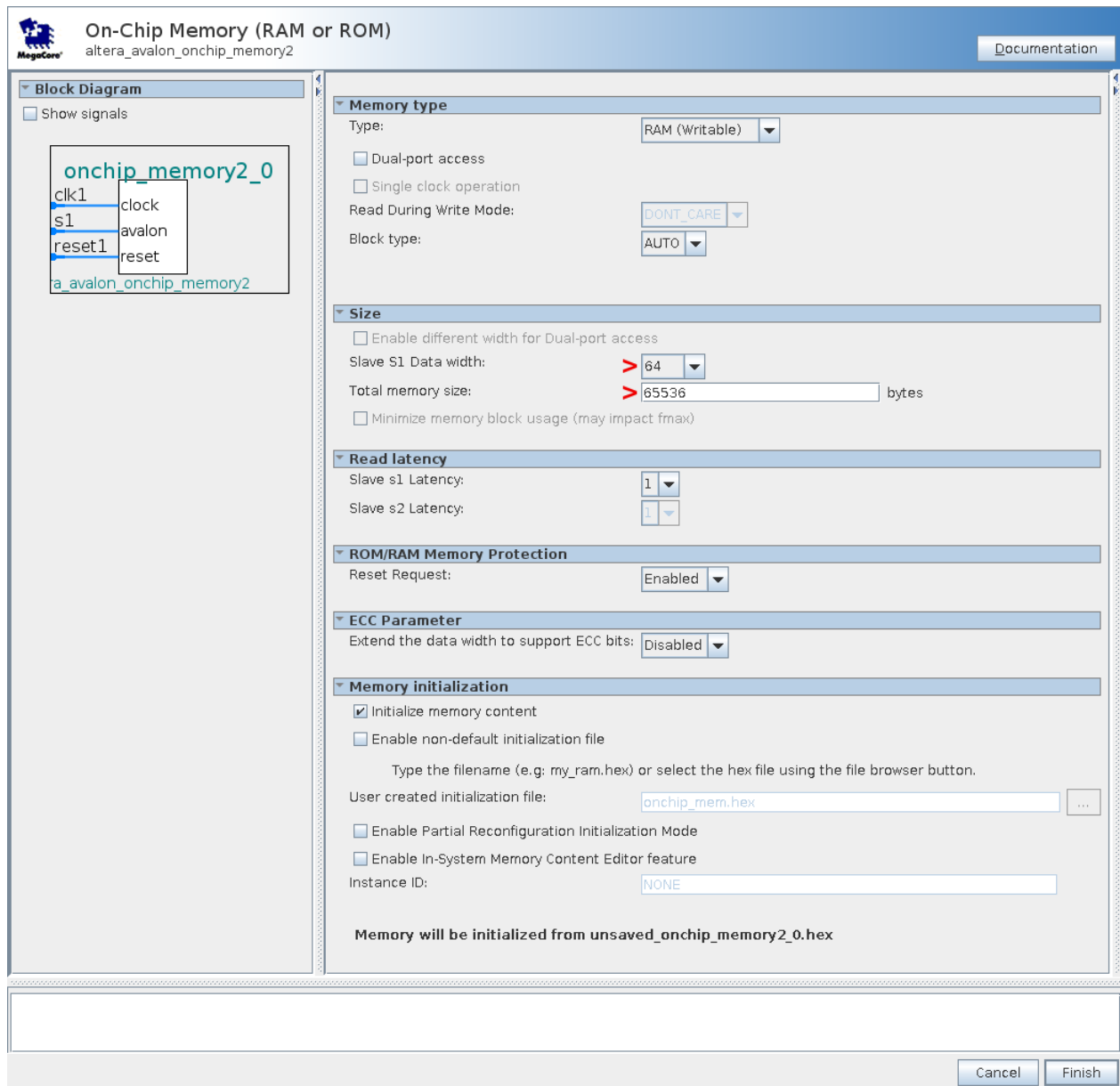


Figure 4: On-Chip Memory Parameters Dialog

Step 4. With the new memory component instantiated in the system, we should give it a more meaningful name to us. To rename components in Qsys, select the component then right click to bring up the context pop-up menu, and select **Rename**. Rename the on-chip memory from **onchip_memory2_0** to **ocram_64k**.

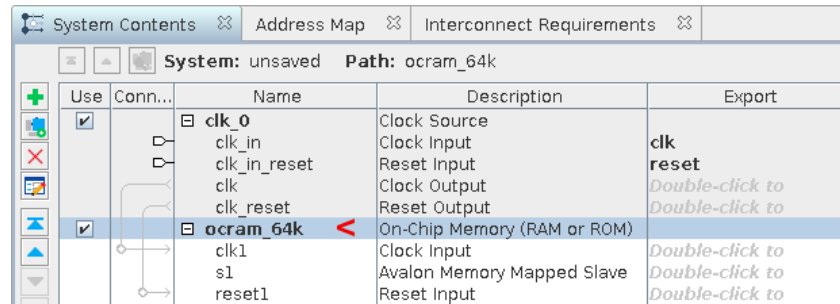


Figure 5: Renamed On-Chip Memory Component

Step 5. To connect components in Qsys, use the **Connections** column in the **System Contents** pane. Possible connections of similar types are presented as circles at the intersection points and shaded in gray. They become active connections when the circle is clicked and turns black. First, connect the **clk_0** clock output with the **ocram_64k** clock input. Then, connect the **clk_0** reset output with the **ocram_64k** reset input. The system should now look like the image below.

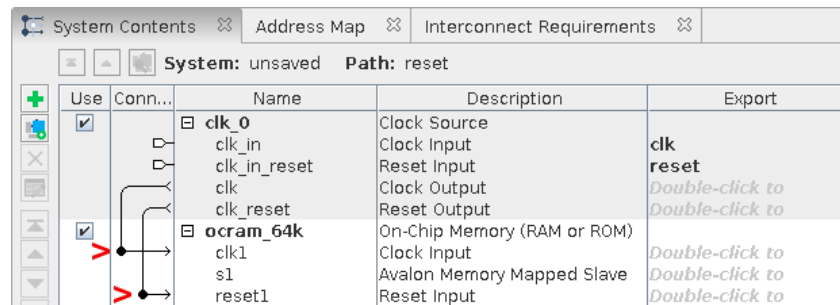


Figure 6: Clock and Reset Connections to On-Chip Memory

Notice how the Avalon Memory Mapped Slave connection on the on-chip RAM does not yet have an available connection. That connection will not appear until an IP with a matching Avalon Memory Mapped Master is added, which will be done in a later step.

Also note that hovering the mouse over a connection point brings up a tool tip that details the interfaces to be connected. This can be useful while building larger systems.

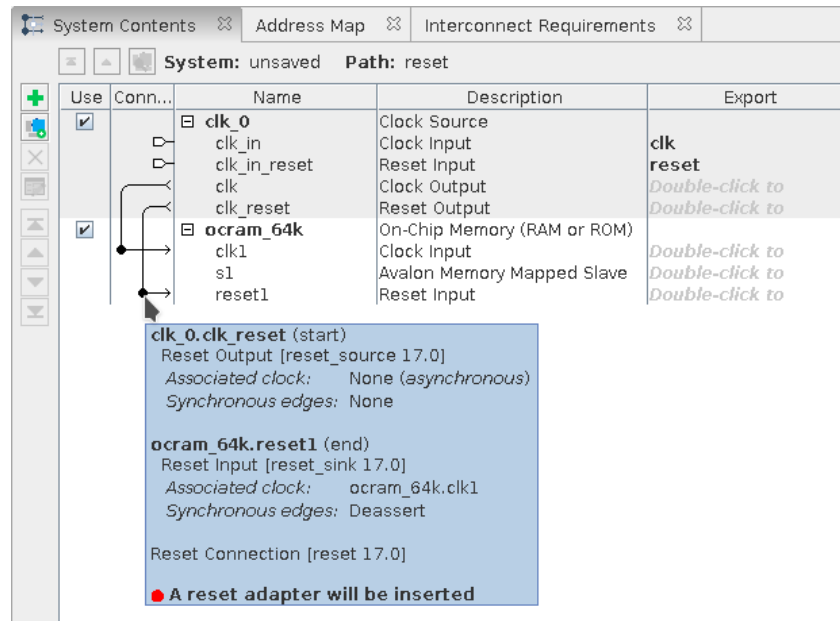


Figure 7: Connection Point Tooltip Pop-ups

Step 6. Adding further IP follows the same process. To complete the system, add the following IP, and connect the clock and reset interfaces for each.

Step 6a. Add a second, small **On-Chip Memory (RAM or ROM)**, with 32-bit data width and 16 byte memory size. The importance of this seemingly redundant component will be explained in a later step. Rename this component to **default_16b**.

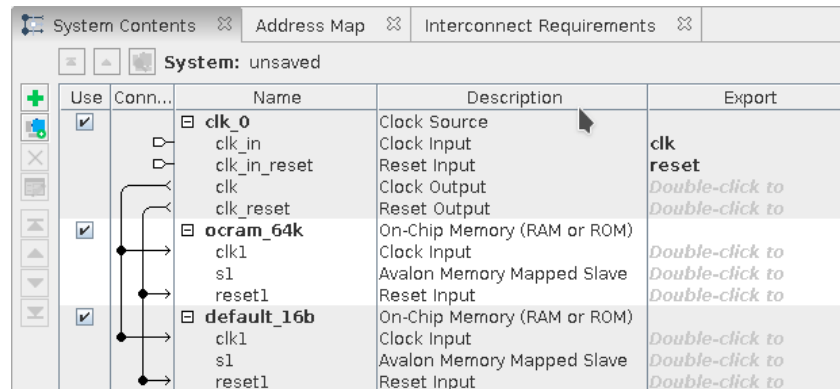


Figure 8: default_16b On-Chip Memory Component

- Step 6b.** Add a **PIO (Parallel I/O)** component with an 8-bit output only. Rename this component to **led_pio**. Then export the **external_connection** interface as **led_pio** by double clicking next to the **external_connection** interface in the **Export** column and then edit the default name catenation to just read **led_pio**.

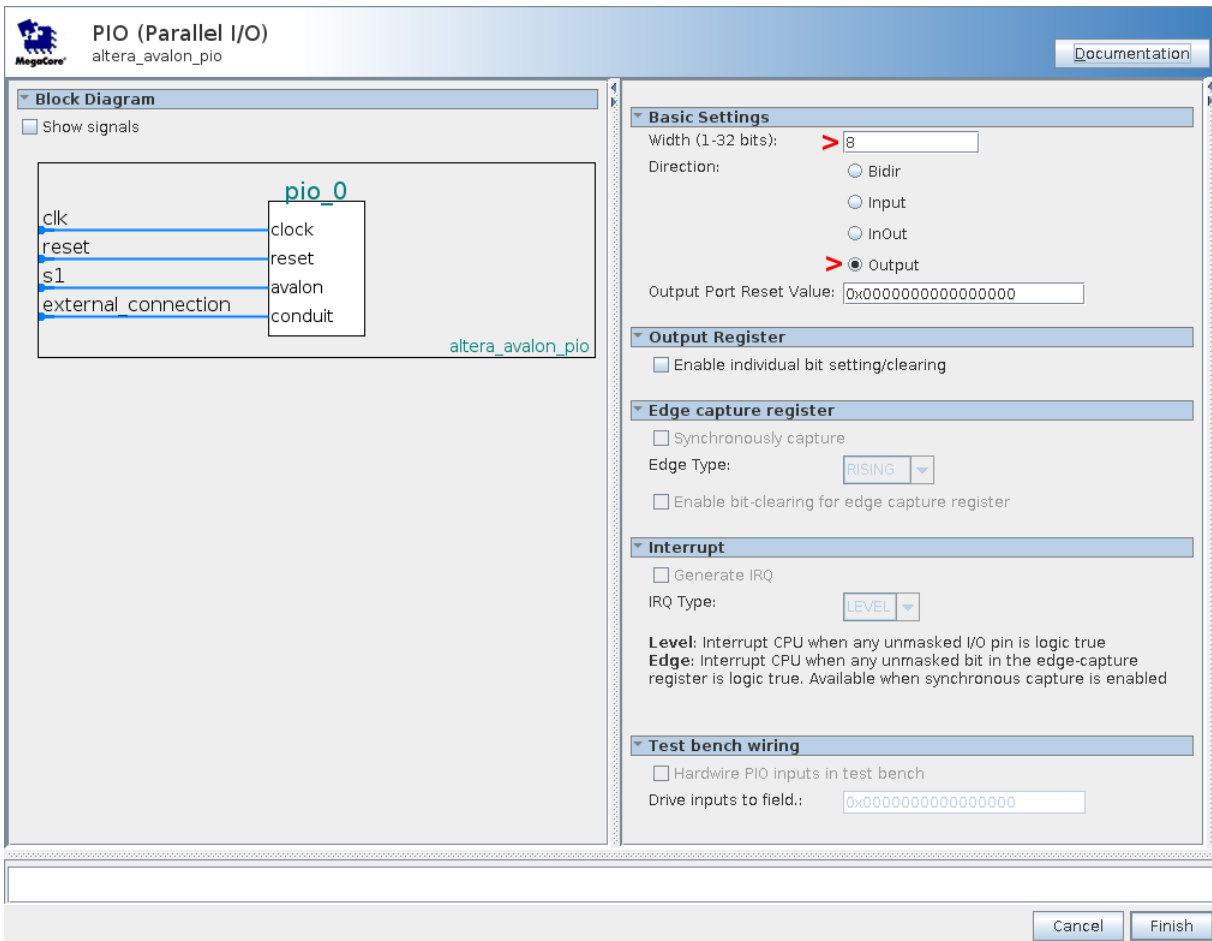


Figure 9: PIO Component Parameters Dialog

- Step 6c.** Add a PIO component with a 1-bit input only. Rename this component to **button_pio** and export the **external_connection** interface as **button_pio**.
- Step 6d.** Add a PIO component with a 4-bit input only. Rename this component to **switch_pio** and export the **external_connection** interface as **switch_pio**.

System: unsaved					
Use	Connec...	Name	Description	Export	
<input checked="" type="checkbox"/>		clk_0	Clock Source	clk reset <i>Double-click to</i> <i>Double-click to</i>	
		clk_in	Clock Input		
		clk_in_reset	Reset Input		
		clk	Clock Output		
		clk_reset	Reset Output		
<input checked="" type="checkbox"/>		ocram_64k	On-Chip Memory (RAM or ROM)	<i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i>	
		clk1	Clock Input		
		s1	Avalon Memory Mapped Slave		
		reset1	Reset Input		
		reset1	Reset Input		
<input checked="" type="checkbox"/>		default_16b	On-Chip Memory (RAM or ROM)	<i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i>	
		clk1	Clock Input		
		s1	Avalon Memory Mapped Slave		
		reset1	Reset Input		
		reset1	Reset Input		
<input checked="" type="checkbox"/>		led_pio	PIO (Parallel I/O)	<i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> led_pio	
		clk	Clock Input		
		reset	Reset Input		
		s1	Avalon Memory Mapped Slave		
		external_connection	Conduit		
<input checked="" type="checkbox"/>		button_pio	PIO (Parallel I/O)	<i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> button_pio	
		clk	Clock Input		
		reset	Reset Input		
		s1	Avalon Memory Mapped Slave		
		external_connection	Conduit		
<input checked="" type="checkbox"/>		switch_pio	PIO (Parallel I/O)	<i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> switch_pio	
		clk	Clock Input		
		reset	Reset Input		
		s1	Avalon Memory Mapped Slave		
		external_connection	Conduit		

Figure 10: PIO Components Added to System

Step 6e. Add a **System ID** component with a recognizable system ID value. Use **0xde10de10** as the ID value and rename this component to **system_id**.

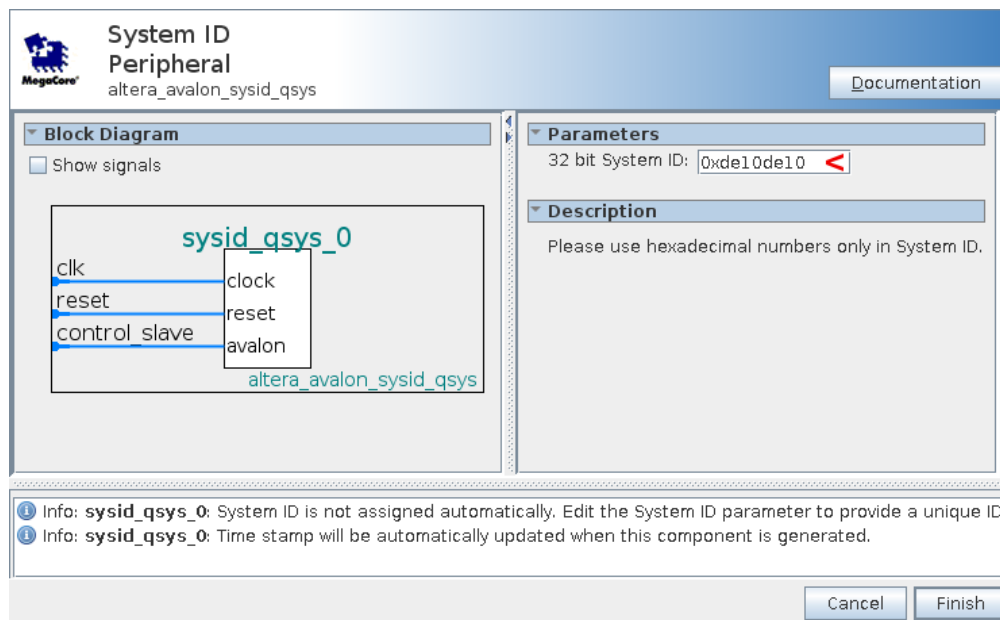


Figure 11: System ID Parameters Dialog

Step 6f. Add a **JTAG to Avalon Master Bridge**. Keep the default instance name **master_0**.

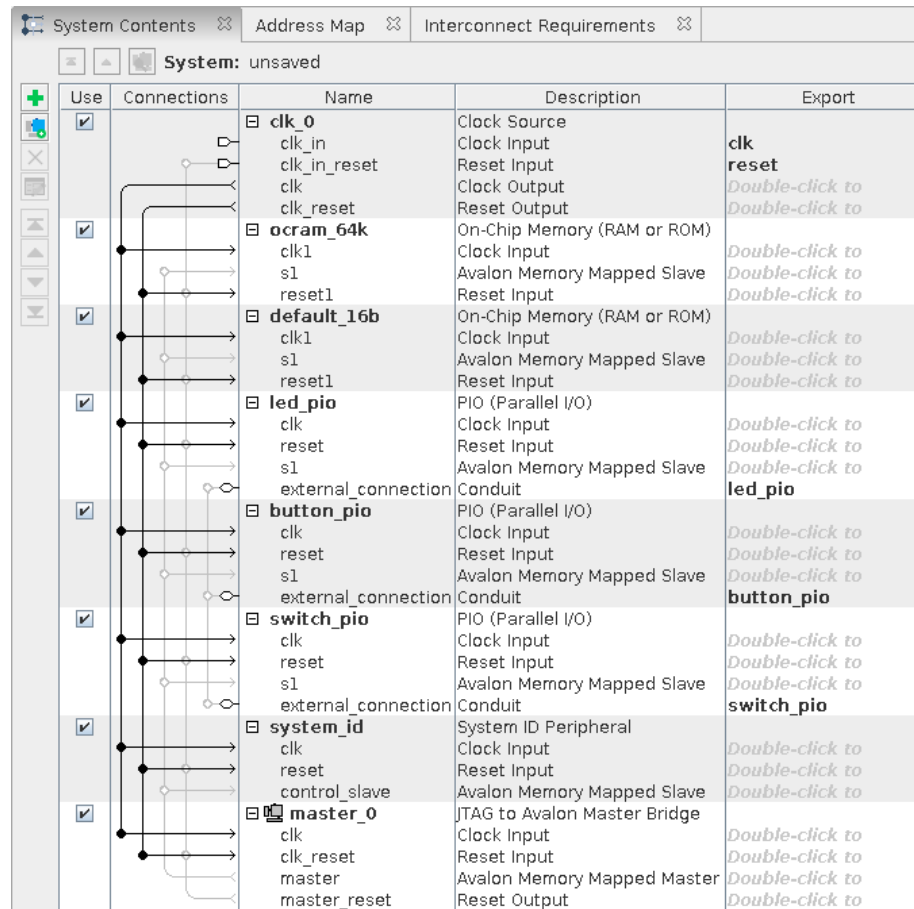


Figure 12: System ID and JTAG Master Added to System

Step 7. At this point, the clocks and resets should all be connected, but it was not until the JTAG master bridge was added that any of the other IPs' Avalon Slave interfaces had any connections available. Now that the JTAG bridge presents an Avalon master, Qsys identifies the similar interfaces and presents the possible connections. Connect the JTAG bridge master to the other IP slaves, while also connecting the **master_reset** reset output to the reset inputs of the other IP blocks.

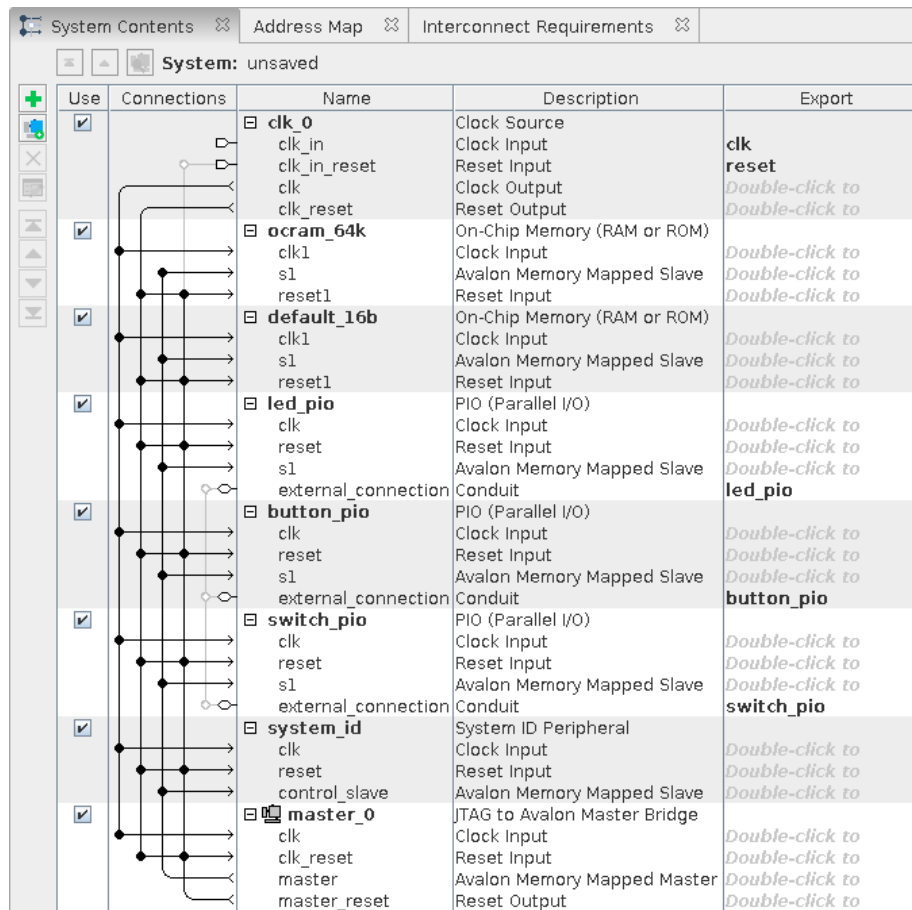


Figure 13: Fully Connected System

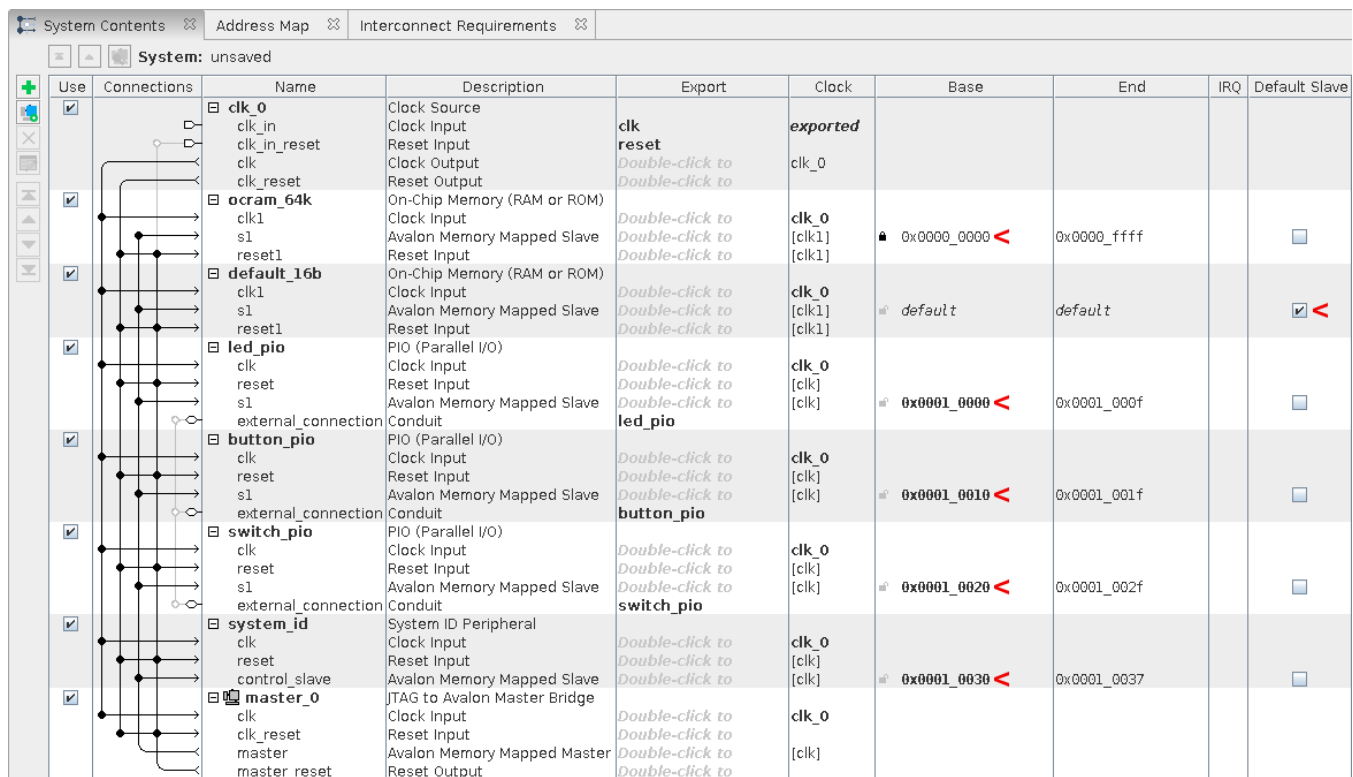
Step 8. Qsys adds a preliminary address span to each slave, but initially they overlap, creating errors in the **Messages** pane.

Type	Path
5 Errors	
unsaved.master_0.master	default_16b.s1 (0x0..0xf) overlaps ocram_64k.s1 (0x0..0xffff)
unsaved.master_0.master	led_pio.s1 (0x0..0xf) overlaps default_16b.s1 (0x0..0xf)
unsaved.master_0.master	button_pio.s1 (0x0..0xf) overlaps led_pio.s1 (0x0..0xf)
unsaved.master_0.master	switch_pio.s1 (0x0..0xf) overlaps button_pio.s1 (0x0..0xf)
unsaved.master_0.master	system_id.control_slave (0x0..0x7) overlaps switch_pio.s1 (0x0..0xf)

Figure 14: Address Span Overlap Errors

To resolve the address span overlap errors, first lock the **ocram_64k** On-Chip Memory address range to start at **0x0000_0000** by clicking the lock button next to the address in the **Base** column. Then, click the **System** menu, and choose **Assign Base Addresses**. Observe the new base addresses for all the IP blocks, and that the address for the On-Chip memory did not change. In general, **Assign Base Addresses** will create the most compact address map by packing all of the peripherals in order from largest address span to smallest address span. There is no guarantee that generated addresses will remain constant over multiple invocations of the

Assign Base Addresses command, however locked address spans are not adjusted by the **Assign Base Addresses** command. For the simplicity of later tutorials, ensure the base address for each IP matches the below image; they can be edited by double-clicking the address itself. Finally, right-click any of the column names in the **System Contents** view, and choose the **Show Default Slave Column** option. Scroll to the right, and check the **Default Slave** box for the **default_16b** on-chip memory.



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Default Slave
<input checked="" type="checkbox"/>		clk_0	Clock Source	clk	exported				
<input checked="" type="checkbox"/>		ocram_64k	On-Chip Memory (RAM or ROM)	reset	clk_0	0x0000_0000	0x0000_ffff		<input type="checkbox"/>
<input checked="" type="checkbox"/>		default_16b	On-Chip Memory (RAM or ROM)	reset	clk_0	default	default		<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>		led_pio	PIO (Parallel I/O)	reset	clk_0	0x0001_0000	0x0001_000f		<input type="checkbox"/>
<input checked="" type="checkbox"/>		button_pio	PIO (Parallel I/O)	reset	clk_0	0x0001_0010	0x0001_001f		<input type="checkbox"/>
<input checked="" type="checkbox"/>		switch_pio	PIO (Parallel I/O)	reset	clk_0	0x0001_0020	0x0001_002f		<input type="checkbox"/>
<input checked="" type="checkbox"/>		system_id	System ID Peripheral	reset	clk_0	0x0001_0030	0x0001_0037		<input type="checkbox"/>
<input checked="" type="checkbox"/>		master_0	JTAG to Avalon Master Bridge	reset	clk_0				<input type="checkbox"/>

Figure 15: Properly Addressed System

Default slaves are an important concept. If a memory mapped read or write transaction is initiated into an unmapped address span, Qsys will route it to the default slave for that master in that interconnect zone. If you do not specify a default slave, then Qsys chooses by default the peripheral with the largest available address span on that interconnect, which in most cases will be some form of memory since those are typically the largest address span peripherals in a system. Choosing a default slave allows the designer to explicitly state that all unmapped address accesses should go to that slave. In this example tutorial, these accesses will be routed to a tiny on-chip memory that is not used for anything else, removing the possibility that data in the main memory or any other peripheral could be corrupted and allowing the designer a space to check whether errant accesses are occurring.

If you are interested in seeing a more capable default slave component, checkout the **trivial_default_avalon_slave** component available in this public GitHub* repository <https://github.com/intel/supplemental-reset-components-for-qsys>. That default slave can be parameterized with a variety of responses to errant accesses, like never responding which essentially holds the master in an infinite wait request, reply with a specific data pattern, generate an Avalon error response, generate an interrupt, or generate a reset.

Step 9. The Qsys system is complete, and now it is time to integrate it back into the Intel Quartus software project. First, save the Qsys system and choose the name **soc_system.qsys**. Close the save window when the save completes. Then, from the **Generate** menu, choose **Generate HDL....** This brings up the **Generate** window, uncheck the **Create block symbol file (.bsf)** box and click the **Generate** button.

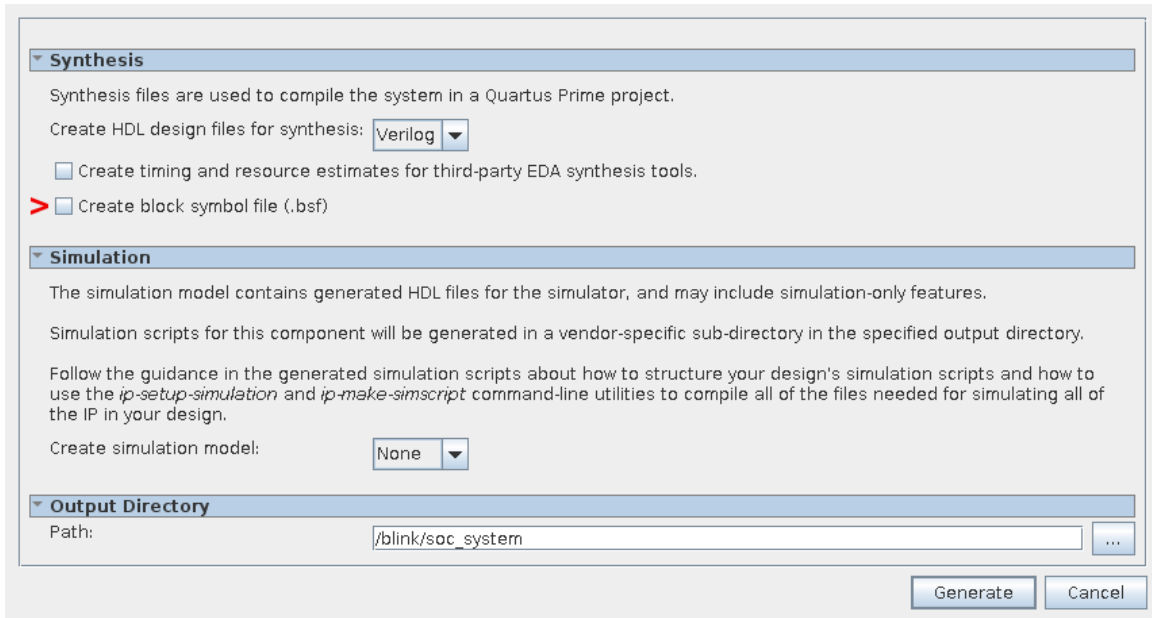


Figure 16: Qsys Generate Dialog

Step 10. Qsys will now generate the HDL files for the IP and interconnect. Once it finishes, click the **Close** button, and then click the **Finish** button at the bottom right of the Qsys window. This will close Qsys, and present instructions to add the **.qip** file to the Intel Quartus software project.

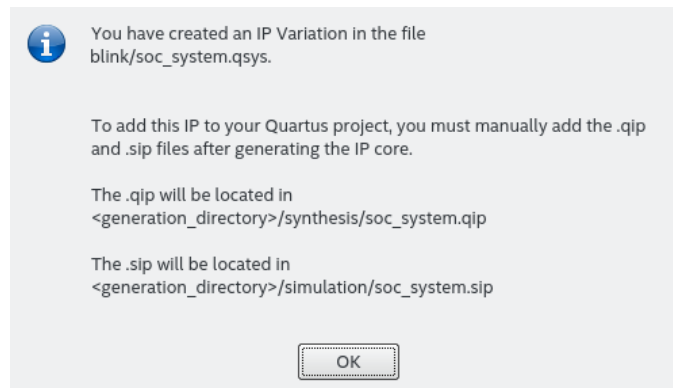


Figure 17: QIP Instruction Dialog

Integrating the Qsys System into the Intel Quartus Software Project

This section will describe how to incorporate the newly generated Qsys output into the existing **blink** Intel Quartus software project.

- Step 1.** Add the Qsys generated QIP file to the **blink** project. Adding the **.qip** file follows the same process used to add the **.sdc** file in the prior tutorial. From the **Assignments** menu, choose **Settings** to bring up the **Settings** window, then select **Files** on the left **Category** list. Click the ... button to the right of the **File Name** box to browse for the file, and navigate to the **.qip** file using the path described in the dialog when you closed Qsys, select that file and click the **Open** button to add the file to the project. Click the **Apply** button and then click the **Ok** button to close the settings window.

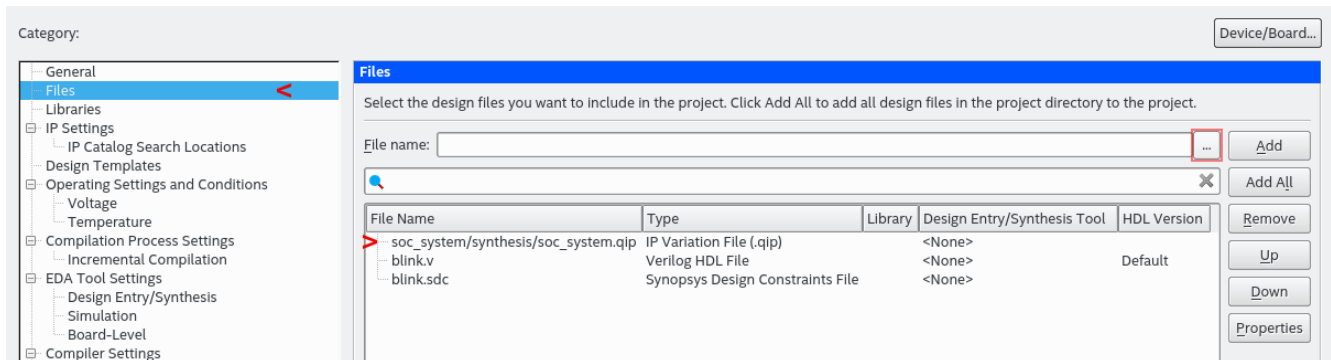


Figure 18: Adding QIP File to Intel Quartus Software Project

- Step 2.** Qsys creates an instantiation template file for the generated system, in this case it is called **soc_system_inst.v** in the **soc_system** folder that is now in the Intel Quartus software project folder. It is convenient to use this to correctly instantiate the Qsys-generated system in the top-level HDL file. Replace the existing code in **blink.v** with the code below:

Download the **blink.v** file [here](#).

If you wish to view the **blink.v** file in the GitHub repo you can look [here](#).

```
1 //
2 // Copyright (c) 2017 Intel Corporation
3 //
4 // Permission is hereby granted, free of charge, to any person obtaining a copy
5 // of this software and associated documentation files (the "Software"), to
6 // deal in the Software without restriction, including without limitation the
7 // rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
8 // sell copies of the Software, and to permit persons to whom the Software is
9 // furnished to do so, subject to the following conditions:
10 //
11 // The above copyright notice and this permission notice shall be included in
12 // all copies or substantial portions of the Software.
13 //
14 // THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
15 // IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
16 // FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
17 // AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
18 // LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
19 // FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
20 // IN THE SOFTWARE.
21 //
22 //
```

```

23 // create module
24 module blink(
25     input    clk,                // 50MHz FPGA input clock
26
27     input    [1:0] push_button,  // KEY[1:0]
28     input    [3:0] switch,       // SW[3:0]
29
30     output   [7:0] leds          // LED[7:0]
31 );
32
33 // Create a power on reset pulse for clean system reset on entry into user mode
34 // We create this with the altera_std_synchronizer core
35 wire sync_dout;
36 altera_std_synchronizer #(
37     .depth (20)
38 ) power_on_reset_std_sync_inst (
39     .clk      (clk),
40     .reset_n  (1'b1),
41     .din      (1'b1),
42     .dout     (sync_dout)
43 );
44
45 // Create a qsys system reset signal that is the logical AND of the power on
46 // reset pulse and the KEY[0] push button
47 wire qsys_system_reset;
48 assign qsys_system_reset = sync_dout & push_button[0];
49
50 /*
51
52 Qsys system instantiation template from soc_system/soc_system_inst.v:
53
54 soc_system u0 (
55     .button_pio_export (<connected-to-button_pio_export>), // button_pio.export
56     .clk_clk            (<connected-to-clk_clk>),           //      clk.clk
57     .led_pio_export     (<connected-to-led_pio_export>),     //      led_pio.export
58     .reset_reset_n      (<connected-to-reset_reset_n>),     //      reset.reset_n
59     .switch_pio_export  (<connected-to-switch_pio_export>)  // switch_pio.export
60 );
61
62 */
63
64 soc_system u0 (
65     .button_pio_export (push_button[1]), // button_pio.export
66     .clk_clk           (clk),            //      clk.clk
67     .led_pio_export    (leds),           //      led_pio.export
68     .reset_reset_n     (qsys_system_reset), //      reset.reset_n
69     .switch_pio_export (switch)          // switch_pio.export
70 );
71
72 endmodule

```

Note the additional instantiated synchronizer in this top level HDL. When the FPGA is released from configuration mode into user mode, it is done so asynchronously, which can violate design timing constraints and lead to design instability. The synchronizer provides a brief and stable reset pulse that is synchronously released, allowing the design to cleanly enter operation after power on and configuration. If you are interested

in seeing a Qsys component that can provide the power on reset functionality, checkout the **power_on_reset** component available in this public Git repository <https://github.com/intel/supplemental-reset-components-for-qsys>.

- Step 3.** Click the **Start Analysis & Elaboration** button on the toolbar to have the Intel Quartus software process the new HDL. There will be many more messages as the Intel Quartus software processes the Qsys-generated system. This process will identify any errors in the HDL source files or how they are configured in the project as well as inform the Intel Quartus software of the new top level ports that we declared in our top module.

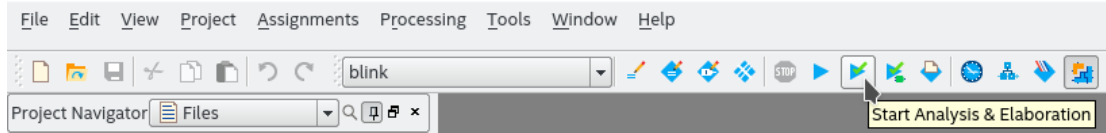


Figure 19: Start Analysis and Elaboration Button on Intel Quartus Software Toolbar

- Step 4.** The new top level module declares many more inputs and outputs than the original design, so new pin constraints will need to be assigned to those inputs and outputs and the stale pin constraints can be removed. The Terasic DE10-Nano user manual contains the diagrams and schematics where these pin assignments are derived from. From the **Assignments** menu, open the **Pin Planner** tool and assign the pins as in the following image. The important columns to match to the image are **Location**, **I/O Standard**, **Current Strength**, and **Slew Rate**.

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate
altera_reserved_tck	Input				PIN_AB5	2.5 V (default)		12mA (default)	
altera_reserved_tdi	Input				PIN_W10	2.5 V (default)		12mA (default)	
altera_reserved_tdo	Output				PIN_Y9	2.5 V (default)		12mA (default)	1 (default)
altera_reserved_tms	Input				PIN_AC7	2.5 V (default)		12mA (default)	
clk	Input	PIN_V11	3B	B3B_NO	PIN_V11	3.3-V LVTTL		16mA (default)	
leds[7]	Output	PIN_AA23	5A	B5A_NO	PIN_AA23	3.3-V LVTTL		16mA	1
leds[6]	Output	PIN_Y16	5A	B5A_NO	PIN_Y16	3.3-V LVTTL		16mA	1
leds[5]	Output	PIN_AE26	5A	B5A_NO	PIN_AE26	3.3-V LVTTL		16mA	1
leds[4]	Output	PIN_AF26	5A	B5A_NO	PIN_AF26	3.3-V LVTTL		16mA	1
leds[3]	Output	PIN_V15	5A	B5A_NO	PIN_V15	3.3-V LVTTL		16mA	1
leds[2]	Output	PIN_V16	5A	B5A_NO	PIN_V16	3.3-V LVTTL		16mA	1
leds[1]	Output	PIN_AA24	5A	B5A_NO	PIN_AA24	3.3-V LVTTL		16mA	1
leds[0]	Output	PIN_W15	5A	B5A_NO	PIN_W15	3.3-V LVTTL		16mA	1
push_button[1]	Input	PIN_AH16	4A	B4A_NO	PIN_AH16	3.3-V LVTTL		16mA (default)	
push_button[0]	Input	PIN_AH17	4A	B4A_NO	PIN_AH17	3.3-V LVTTL		16mA (default)	
switch[3]	Input	PIN_W20	5B	B5B_NO	PIN_W20	3.3-V LVTTL		16mA (default)	
switch[2]	Input	PIN_W21	5B	B5B_NO	PIN_W21	3.3-V LVTTL		16mA (default)	
switch[1]	Input	PIN_W24	5B	B5B_NO	PIN_W24	3.3-V LVTTL		16mA (default)	
switch[0]	Input	PIN_Y24	5B	B5B_NO	PIN_Y24	3.3-V LVTTL		16mA (default)	
<<new node>>									

Figure 20: Pin Assignments in Pin Planner

- Step 5.** Update the timing constraints in **blink.sdc** to match the newly declared ports by replacing the contents of **blink.sdc** with the below code.

Download the blink.sdc file [here](#).

If you wish to view the blink.sdc file in the GitHub repo you can look [here](#).

```

1  #
2  # Copyright (c) 2017 Intel Corporation
3  #
4  # Permission is hereby granted, free of charge, to any person obtaining a copy
5  # of this software and associated documentation files (the "Software"), to
6  # deal in the Software without restriction, including without limitation the
7  # rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
8  # sell copies of the Software, and to permit persons to whom the Software is
9  # furnished to do so, subject to the following conditions:
10 #
11 # The above copyright notice and this permission notice shall be included in
12 # all copies or substantial portions of the Software.
13 #
14 # THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
15 # IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

```

```

16 # FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
17 # AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
18 # LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
19 # FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
20 # IN THE SOFTWARE.
21 #
22
23 # inform quartus that the clk port brings a 50MHz clock into our design so
24 # that timing closure on our design can be analyzed
25
26 create_clock -name clk -period "50MHz" [get_ports clk]
27 derive_clock_uncertainty
28
29 # inform quartus that the PIO inputs and outputs have no critical timing
30 # requirements. These signals are driving LEDs and reading discrete push button
31 # and switch inputs, there are no timing relationships that are critical for any
32 # of this
33
34 set_false_path -from [get_ports {switch[0]] -to *
35 set_false_path -from [get_ports {switch[1]] -to *
36 set_false_path -from [get_ports {switch[2]] -to *
37 set_false_path -from [get_ports {switch[3]] -to *
38 set_false_path -from * -to [get_ports {leds[0]]
39 set_false_path -from * -to [get_ports {leds[1]]
40 set_false_path -from * -to [get_ports {leds[2]]
41 set_false_path -from * -to [get_ports {leds[3]]
42 set_false_path -from * -to [get_ports {leds[4]]
43 set_false_path -from * -to [get_ports {leds[5]]
44 set_false_path -from * -to [get_ports {leds[6]]
45 set_false_path -from * -to [get_ports {leds[7]]
46 set_false_path -from [get_ports {push_button[0]] -to *
47 set_false_path -from [get_ports {push_button[1]] -to *
48
49 # Define timing constraints for the JTAG IO pins so that Quartus properly closes
50 # timing on these signal paths. Otherwise we could have unreliable JTAG
51 # communication with the device over the USB Blaser II connection.
52 # NOTE: the 'altera_reserved_tck' clock is automatically defined by Quartus
53
54 set_input_delay -clock altera_reserved_tck -clock_fall 3 [get_ports {altera_reserved_tdi}]
55 set_input_delay -clock altera_reserved_tck -clock_fall 3 [get_ports {altera_reserved_tms}]
56 set_output_delay -clock altera_reserved_tck 3 [get_ports {altera_reserved_tdo}]

```

Step 6. Click the **Start Compilation** button in the top toolbar to compile the entire design.

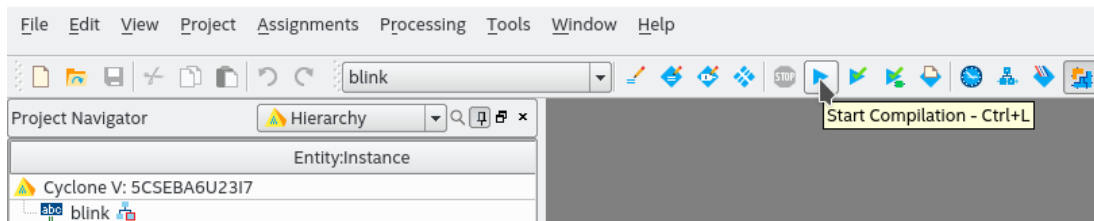


Figure 21: Start Compilation Button on Intel Quartus Software Toolbar

Step 7. After your design has successfully compiled, we will perform one last activity in the Intel Quartus software to prepare for the next tutorial that will actually make use of this system on the Terasic DE10-Nano board. Since we have defined a memory mapped embedded system in Qsys with a JTAG master connected to a number of slave peripherals, it will be necessary for us to know what the base addresses are of the slave peripherals so we can interact with them, performing read and write transactions to them. That base address information is captured in Qsys. You can visualize them in Qsys a number of ways, but that is not convenient for software developers or other users of this system to write code for it. Each time you generate a Qsys system, Qsys outputs a database file called **<your-system-name>.sopcinfo**. The Intel Quartus software tools installation provides a utility that can be used to translate the SOPCINFO database information into a usable macro

format that can be used for various purposes called **sopc-create-header-files**. The default functionality of **sopc-create-header-files** is to create C style header macros from each masters' perspective in the Qsys system. We will perform this operation in the Intel Quartus software TCL Console which is located in the lower middle of the default Intel Quartus software GUI. If you do not see the TCL Console pane, you can open it by selecting the **View > Utility Windows > TCL Console** menu.



Figure 22: Intel Quartus Prime Software TCL Console

We will first create a directory to output the header files into, then we will create the default header file output using **sopc-create-header-files** and finally we will extract the base address entries out of the JTAG master header file for the FPGA peripherals that it is connected to. We will perform all of this with the following TCL commands:

```
Quartus Prime Tcl Console

# make a directory called 'qsys_headers' to store the header files
tcl> file mkdir qsys_headers

# create a TCL variable SCHF_PATH to hold the path to the executable program
# socp-create-header-files on your host PC using the environment variables
# provided by Quartus.
tcl> set SCHF_PATH [glob -join $quartus(quartus_rootpath) socp_builder bin socp-create-header-files]

# create a TCL variable BAT_PATH to hold the path to the Nios II Command Shell
# batch file on Windows platforms. The following code sequence will work on
# either Windows or Linux. For Linux this variable will just be set to NULL.
tcl> set BAT_PATH {}
tcl> if {$tcl_platform(platform) == "windows"} {
    > set BAT_PATH [glob -join $quartus(quartus_rootpath) .. nios2eds {Nios II Command Shell.bat}]
    > }

# execute socp-create-header-files to generate the header files
tcl> eval exec -ignorestderr ${BAT_PATH} ${SCHF_PATH} soc_system.sopcinfo --output-dir qsys_headers

# read the header file for master_0 into a TCL variable
tcl> set master_0_header [read [open [glob -join qsys_headers master_0.h] r]]

# output the C macro lines for the FPGA peripheral base addresses
tcl> foreach line [split ${master_0_header} "\n"] { \
    > if {[string match "*_BASE*" ${line}]} {puts ${line}}}
```

If you have put your Qsys system together properly and executed the above commands correctly, you should see the following output in the Intel Quartus software TCL console representing the base address definitions for the five Qsys peripherals in the FPGA fabric connected to master_0, the JTAG master bridge component.

```
#define OGRAM_64K_BASE 0x0
#define LED_PIO_BASE 0x10000
#define BUTTON_PIO_BASE 0x10010
#define SWITCH_PIO_BASE 0x10020
#define SYSTEM_ID_BASE 0x10030
```

That's it! You have designed and compiled your first Qsys system. This basic design is intended as a stepping stone to the next tutorials and it won't do much on its own when programmed into your device. Continue to the "Interacting with FPGA Designs Using System Console" tutorial where we demonstrate how to use the System Console tool to program the FPGA and interact with this design.