



Making Qsys Components

For Quartus Prime 16.0

1 Introduction

The Altera Qsys tool allows a digital system to be designed by interconnecting selected Qsys components, such as processors, memory controllers, parallel and serial ports, and the like. The Qsys tool includes many pre-designed components that may be selected for inclusion in a designed system, and it is also possible for users to create their own custom Qsys components. This tutorial provides an introduction to the process of creating custom Qsys components. The discussion is based on the assumption that the reader is familiar with the Verilog or VHDL hardware description language and is also familiar with the material in the tutorial *Introduction to the Altera Qsys System Integration Tool*.

The screen captures in this tutorial were obtained using the Quartus Prime version 16.0 software; if other versions are used, some of the images may be slightly different.

Contents:

- Introduction to Qsys
- What is a Qsys component?
- Avalon Memory-Mapped Interface details
- Adding a new component to Qsys
- Instantiating the new component

2 Introduction to Qsys

The Qsys tool allows users to put together a system using pre-made and/or custom components. Such systems usually comprise one or more processors, memory interfaces, I/O ports and other custom hardware. The Qsys-created system can be included as part of a larger circuit and implemented on an FPGA board, such as the Altera DE-series boards. An example of such a system is depicted in Figure 1, where the part of the system created by the Qsys tool is highlighted in a blue color.

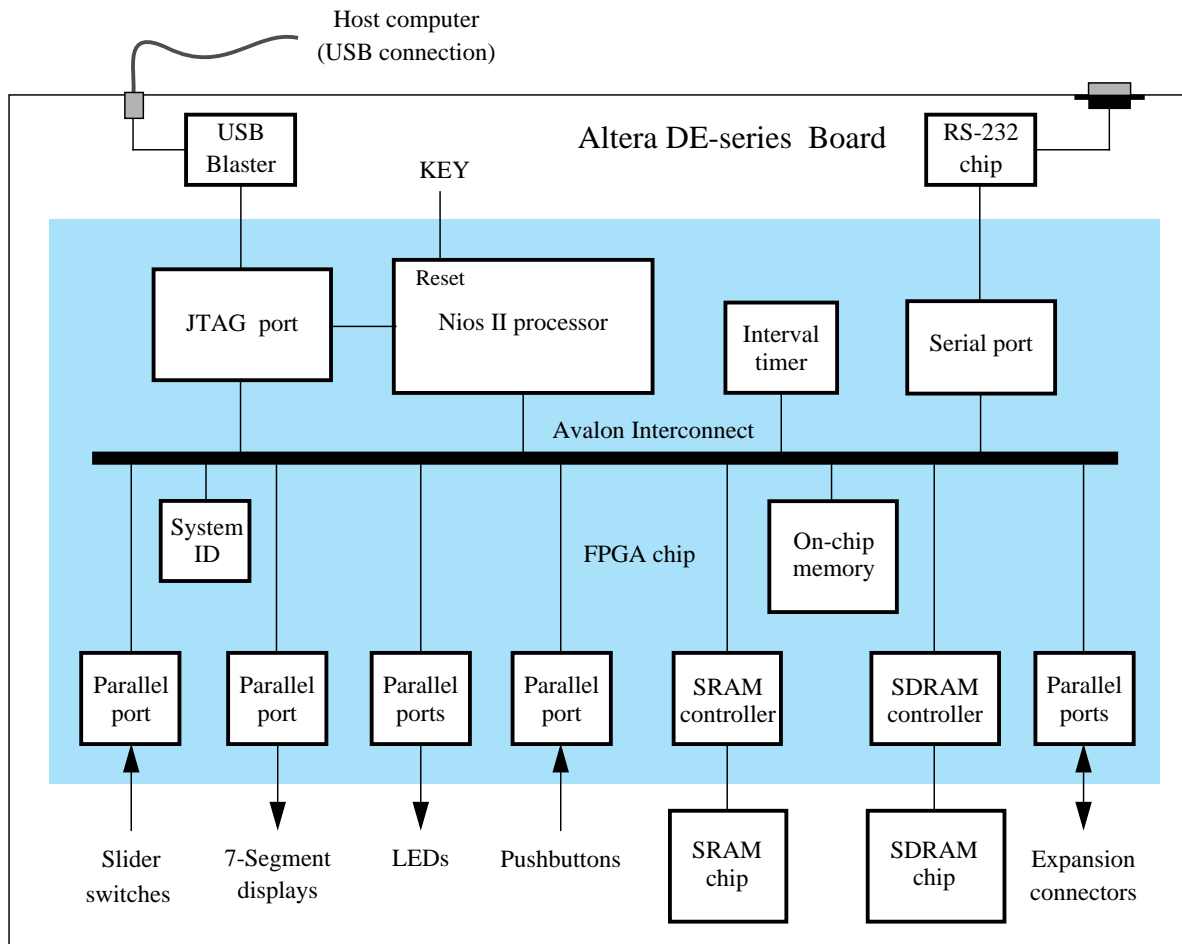


Figure 1. Block diagram of an example Qsys system implemented on an FPGA board.

Each component in the system, referred to as a *Qsys component*, adheres to at least one of the Avalon Interfaces supported by Qsys. With the interface defined for the component, Qsys is able to construct an interconnect structure, called the Avalon Interconnect, which enables components to exchange data. The Qsys tool can generate a system based on the selected set of components and user parameters. The generated system contains Verilog or VHDL code for each component and the interconnect structure, allowing it to be synthesized, placed and routed for an FPGA device.

In this tutorial we explain what we mean by a Qsys component, describe the Avalon Interfaces in more detail, and show how to create a custom component that can be included in the Qsys list of available components.

3 What is a Qsys Component?

A Qsys component is a hardware subcircuit that is available as a library component for use in the Qsys tool. Typically, it contains two parts: the internal hardware modules, and the external Avalon Interfaces. The internal modules are the circuits that implement the desired functionality of the Qsys component, while the Avalon Interfaces are used by the component to communicate with hardware modules that are external to the component.

There are many types of Avalon Interfaces; the most commonly used types are:

- Avalon Clock Interface – an interface that drives or receives clocks
- Avalon Reset Interface – an interface that provides reset capability
- Avalon Memory-Mapped Interface (Avalon MM) – an address-based read/write interface which is typical of master-slave connections
- Avalon Streaming Interface (Avalon-ST) – an interface that supports unidirectional flow of data
- Avalon Conduit Interface – an interface that accommodates individual signals or groups of signals that do not fit into any of the other Avalon Interface types. You can export the conduit signals to make connections external to the Qsys system.

A single component can use as many of these interface types as it requires. For example, a component might provide an Avalon-ST port for high-throughput data, in addition to an Avalon MM slave port for control. All components must include the Avalon Clock and Reset Interfaces. Readers interested in more complete information about the Avalon Interfaces may consult the *Avalon Interface Specifications* document that can be found on the Altera website.

In this tutorial we will show how to develop a Qsys component that has an Avalon Memory-Mapped Interface and an Avalon Conduit Interface. The component is a 32-bit register that can be read or written as a memory-mapped slave device via the Avalon Interconnect and can be visible outside the system through a conduit signal. The purpose of the conduit is to allow the register contents to be displayed on external components such as LEDs or 7-segment displays. Thus, this register is similar to the output parallel ports shown in Figure 1.

If the register is to be used in a system such as the one depicted in Figure 1, then it should respond correctly to Nios II instructions that store data into the register, or load data from it. Let D be the 32-bit input data for the register, *byteenable* be the four-bit control input that indicates which byte(s) will be loaded with new data, and Q be the 32-bit output of the register. In addition, it is necessary to provide clock and reset signals. Figures 2 and 4 show a suitable specification for the desired register, called *reg32*, in Verilog and VHDL, respectively.

Our register will be instantiated in a top-level module that provides the necessary signals for connecting to an Avalon MM Interconnect. Let this module be called *reg32_avalon_interface*. The Avalon MM Interface signals used in this module are:

- *clock*
- *resetn* – active-low reset signal
- *readdata* – 32-bit data read from the register
- *writedata* – 32-bit data to be stored in the register
- *read* – active when a read (load) transaction is to be performed
- *write* – active when a write (store) transaction is to be performed
- *byteenable* – two-bit signal that identifies which bytes are being used
- *chipsselect* – active when the register is being read or written

The *reg32_avalon_interface* module also provides a 32-bit Avalon Conduit Interface signal called *Q_export*. Figures 3 and 5 show how this module can be specified in Verilog and VHDL code, respectively.

```

module reg32 (clock, resetn, D, byteenable, Q);
    input clock, resetn;
    input [3:0] byteenable;
    input [31:0] D;
    output reg [31:0] Q;

    always@(posedge clock)
        if (!resetn)
            Q <= 32'b0;
        else
            begin
                // Enable writing to each byte separately
                if (byteenable[0]) Q[7:0] <= D[7:0];
                if (byteenable[1]) Q[15:8] <= D[15:8];
                if (byteenable[2]) Q[23:16] <= D[23:16];
                if (byteenable[3]) Q[31:24] <= D[31:24];
            end
        end
endmodule

```

Figure 2. Verilog code for the 32-bit register.

```
module reg32_avalon_interface (clock, resetn, writedata, readdata, write, read,
    byteenable, chipselect, Q_export);

    // signals for connecting to the Avalon fabric
    input clock, resetn, read, write, chipselect;
    input [3:0] byteenable;
    input [31:0] writedata;
    output [31:0] readdata;

    // signal for exporting register contents outside of the embedded system
    output [31:0] Q_export;

    wire [3:0] local_byteenable;
    wire [31:0] to_reg, from_reg;

    assign to_reg = writedata;

    assign local_byteenable = (chipselect & write) ? byteenable : 4'd0;

    reg32 U1 ( .clock(clock), .resetn(resetn), .D(to_reg), .byteenable(local_byteenable),
        .Q(from_reg) );

    assign readdata = from_reg;
    assign Q_export = from_reg;
endmodule
```

Figure 3. Verilog code for the Avalon MM Interface.

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY reg32 IS
    PORT ( clock, resetn : IN    STD_LOGIC;
           D              : IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
           byteenable     : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);
           Q              : OUT   STD_LOGIC_VECTOR(31 DOWNTO 0) );
END reg32;

ARCHITECTURE Behavior OF reg32 IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL clock'EVENT AND clock = '1';
        IF resetn = '0'THEN
            Q <= "00000000000000000000000000000000";
        ELSE
            IF byteenable(0) = '1'THEN
                Q(7 DOWNTO 0) <= D(7 DOWNTO 0); END IF;
            IF byteenable(1) = '1'THEN
                Q(15 DOWNTO 8) <= D(15 DOWNTO 8); END IF;
            IF byteenable(2) = '1'THEN
                Q(23 DOWNTO 16) <= D(23 DOWNTO 16); END IF;
            IF byteenable(3) = '1'THEN
                Q(31 DOWNTO 24) <= D(31 DOWNTO 24); END IF;
            END IF;
        END PROCESS;
    END Behavior;

```

Figure 4. VHDL code for the new register.

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY reg32_avalon_interface IS
    PORT (
        clock, resetn      : IN    STD_LOGIC;
        read, write, chipselect : IN    STD_LOGIC;
        writedata           : IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
        byteenable          : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);
        readdata            : OUT   STD_LOGIC_VECTOR(31 DOWNTO 0);
        Q_export            : OUT   STD_LOGIC_VECTOR(31 DOWNTO 0) );
END reg32_avalon_interface;

ARCHITECTURE Structure OF reg32_avalon_interface IS
    SIGNAL local_byteenable : STD_LOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL to_reg, from_reg : STD_LOGIC_VECTOR(31 DOWNTO 0);

    COMPONENT reg32
        PORT (
            clock, resetn : IN    STD_LOGIC;
            D              : IN    STD_LOGIC_VECTOR(31 DOWNTO 0);
            byteenable     : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);
            Q              : OUT   STD_LOGIC_VECTOR(31 DOWNTO 0) );
    END COMPONENT;
BEGIN
    to_reg <= writedata;
    WITH (chipselect AND write) SELECT
        local_byteenable <= byteenable WHEN '1', "0000" WHEN OTHERS;
    reg_instance: reg32 PORT MAP (clock, resetn, to_reg, local_byteenable, from_reg);
    readdata <= from_reg;
    Q_export <= from_reg;
END Structure;

```

Figure 5. VHDL code for the memory-mapped new-register interface.

4 Avalon Memory-Mapped Interface Details

The Avalon Memory-Mapped Interface is a bus-like protocol that allows two components to exchange data. One component implements a *master* interface that allows it to request and send data to *slave* components. A slave component can only receive and process requests, either receiving data from the master, or providing the data requested by the master.

Each slave device includes one or more registers that can be accessed for read or write transaction by a master

device. Figures 6 and 7 illustrate the signals that are used by master and slave interfaces. The direction of each signal is indicated by arrows beside it, with \leftarrow indicating an output and \rightarrow indicating an input to a device. All transactions are synchronized to the positive edge of the Avalon *clk* signal. At time t_0 in the figures, the master begins a read transaction by placing a valid address on its *address* outputs and asserting its *read* control signal. The slave recognizes the request because its *chipselect* input is asserted. It responds by placing valid data on its *readdata* outputs; the master captures this data on its *readdata* inputs and the read transaction ends at time t_1 . A second read transaction is shown in the figure starting at time t_2 . In this case, the slave device asserts the *waitrequest* input of the master, which can be used to extend a read transaction by any number of clock cycles. The slave device deasserts the *waitrequest* signal and provides the requested data at time t_3 , and the read transaction ends at time t_4 .

A write transaction is illustrated starting at time t_5 in Figures 6 and 7. The master places a valid address and data on its *address* and *datawrite* outputs, and asserts the *write* control signal. The slave captures the data on its *datawrite* inputs and the write transaction ends at time t_6 . Although not shown in this example, a slave device can assert the *waitrequest* input of the master to extend a write transaction over multiple clock cycles if needed.

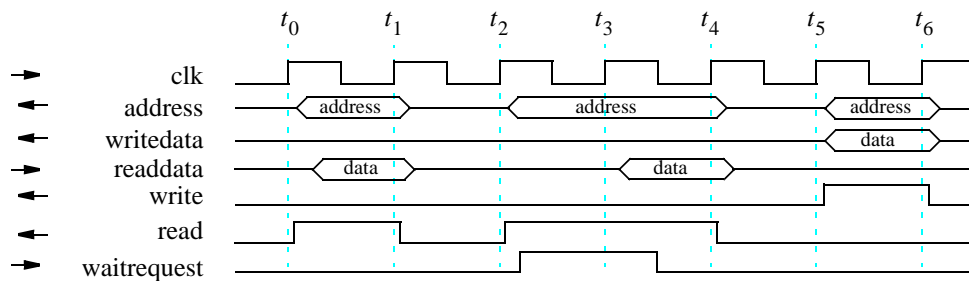


Figure 6. Timing diagram for read/write transactions from the master's point of view.

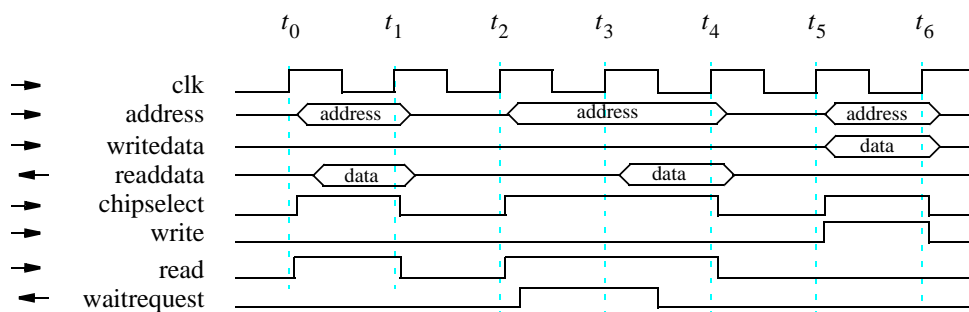


Figure 7. Timing diagram for read/write transactions from the slave's point of view.

Addresses used by master devices are aligned to 32-bit word boundaries. For example, Figure 8 illustrates four 32-bit addresses that could be used to select four registers in a slave device. The address of the first register is 0x10000000, the address of the second register is 0x10000004, and so on. In this example, the slave would have a two-bit address input for selecting one of its four registers in any read or write transaction. Since addresses are word-aligned, the lower two address bits from the master are not seen in the slave. The master provides a four-bit *byteenable* signal, which is used by the slave to control a write transaction for individual bytes. For example, if the master performs a write transaction to only the most-significant byte of the second register in Figure 8 then the

master would write to address 0x10000007 by having its *byteenable* output signal set to the value 0x1000 and its *address* output signal set to the value 0x10000004. The slave device would see its two-bit address input set to 0x01 and would use its *byteenable* inputs to ensure that the write transaction is performed only for the selected byte of the second register. Although the *byteenable* signals are not shown in Figures 6 and 7, they have the same timing as the address signals.

The above examples show the basic transactions between a master and a slave. More advanced transactions can be performed, the procedure for which is described in the *Avalon Interconnect Specifications* document.

Master Address	Slave Address[1..0]	31 32 1 0	
0x10000000	00		First Register
0x10000004	01		Second Register
0x10000008	10		Third Register
0x1000000C	11		Fourth Register

Figure 8. Example for registers in an Avalon MM Interface.

5 Adding a New Component to the Qsys IP Catalog

In this section we show how to create a new Qsys component for our 32-bit register defined in Figures 2 to 4. As a first step, start the Quartus Prime software and make a new project for use with this tutorial. Name the project *component_tutorial*, and choose the settings that are needed for your DE-series board, including the specific FPGA chip.

Next, using your preferred HDL language (Verilog or VHDL), create a new design file (File > New... > Design Files) and add the Verilog code from Figure 2, or the VHDL code from Figure 4, and name the file *reg32.v* (or *.vhd*). Add this file to your current project (Project > Add Current File to Project). Create another file and add the Verilog code from Figure 3 or the VHDL code from Figure 5, name the file *reg32_avalon_interface.v* (or *.vhd*), and add this file to the current project.

Later, we will create a top-level HDL file for the *component_tutorial* project, but first we will use the Qsys tool to generate an embedded system. Open the Qsys tool to get to the window depicted in Figure 9. The Qsys tool automatically includes a clock component in the system, as shown in the figure. Since we will use an active-low reset signal for our system, double-click on the name of the exported reset signal on the clock component and change it to *resetn*.

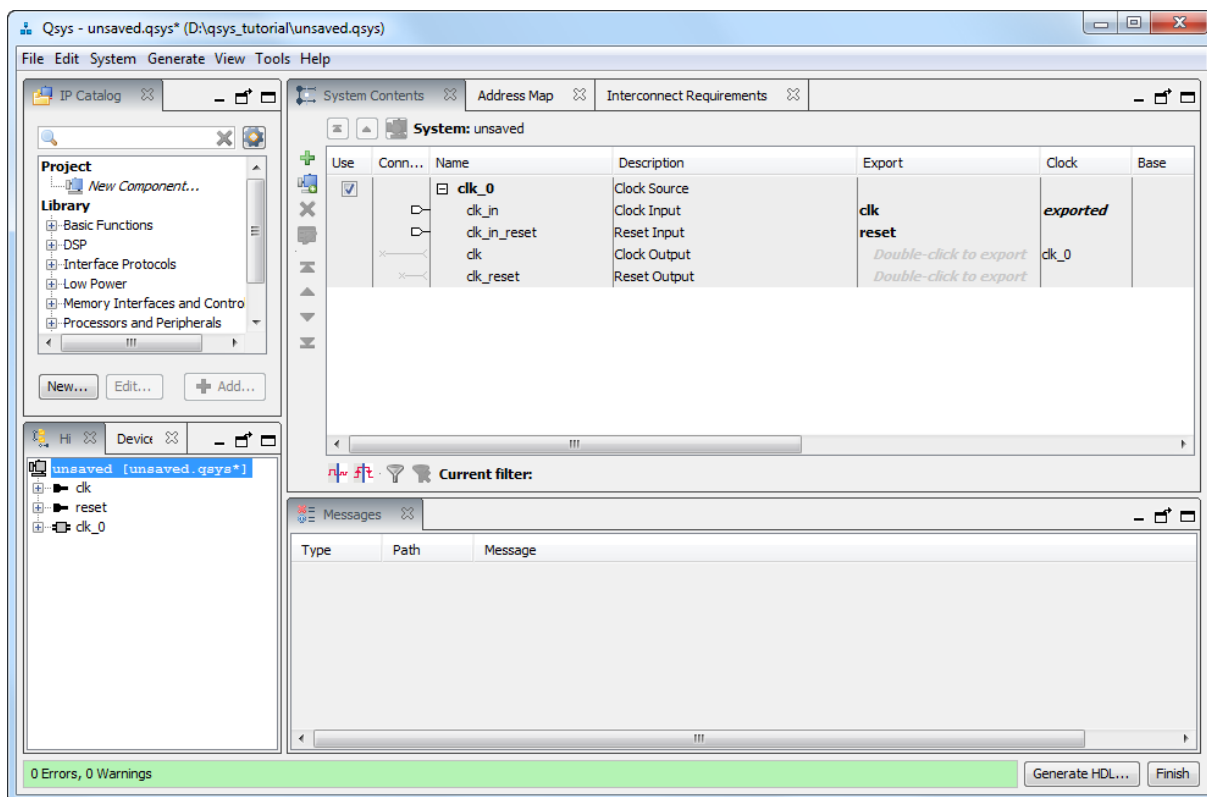


Figure 9. Qsys window.

Before creating the new Qsys component for our 32-bit register, we will first instantiate some other components that will be needed in our system. In the IP Catalog area of the Qsys window expand the Processors and Peripherals > Embedded Processors item and add a Nios II (Classic) Processor to the system. In the Nios II Processor dialog window that opens, select Nios II/e as the type of processor. Next, in the IP Catalog, expand the Basic Functions > On-Chip Memory item and add an On-Chip Memory (RAM or ROM) component. Click Finish to return to the main Qsys window. In the Connections area of the Qsys window, make the connections illustrated in Figure 10 between the clock component, Nios II processor, and on-chip memory module.

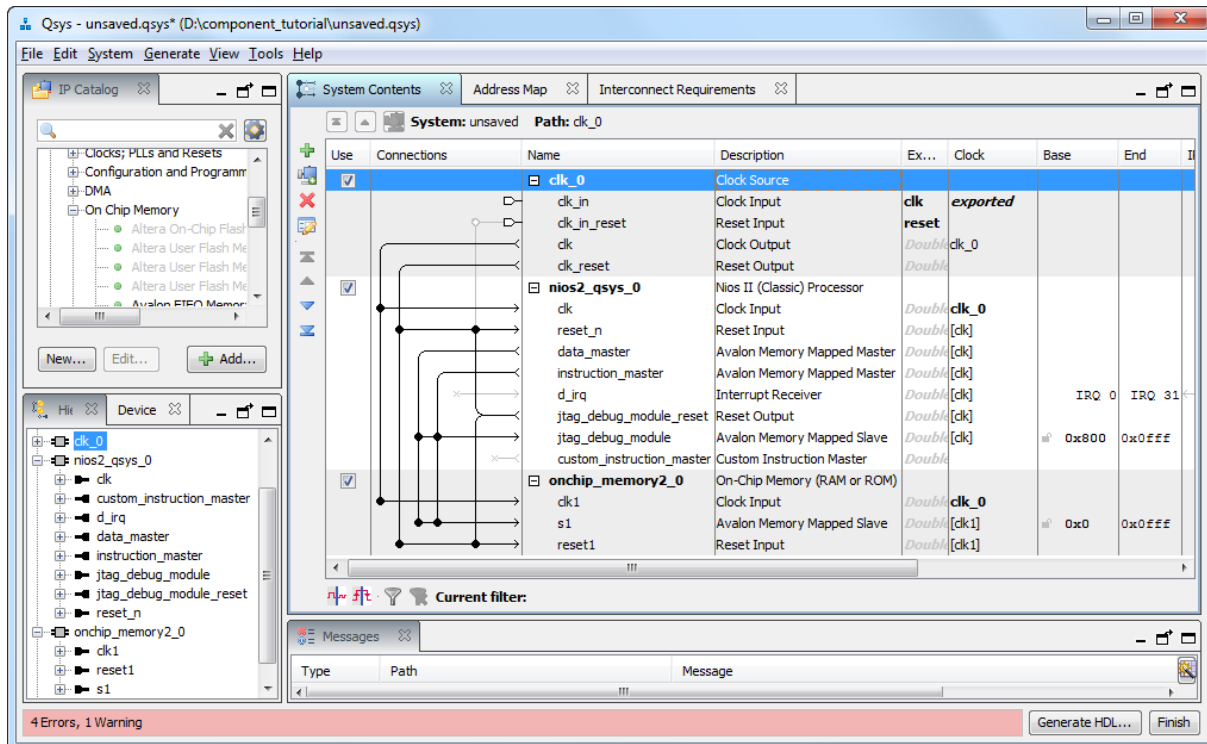


Figure 10. Connections needed between components.

Errors will be displayed in the Qsys Messages window about the Reset and Exception vectors memories that are needed for the Nios II Processor. To fix these errors, re-open the Nios II processor component that has already been added to the system by right-clicking on it and selecting Edit. In the window shown in Figure 11 use the provided drop-down menus to set both the Reset vector memory and Exception vector memory to the on-chip memory component. Click Finish to return to the main Qsys window.

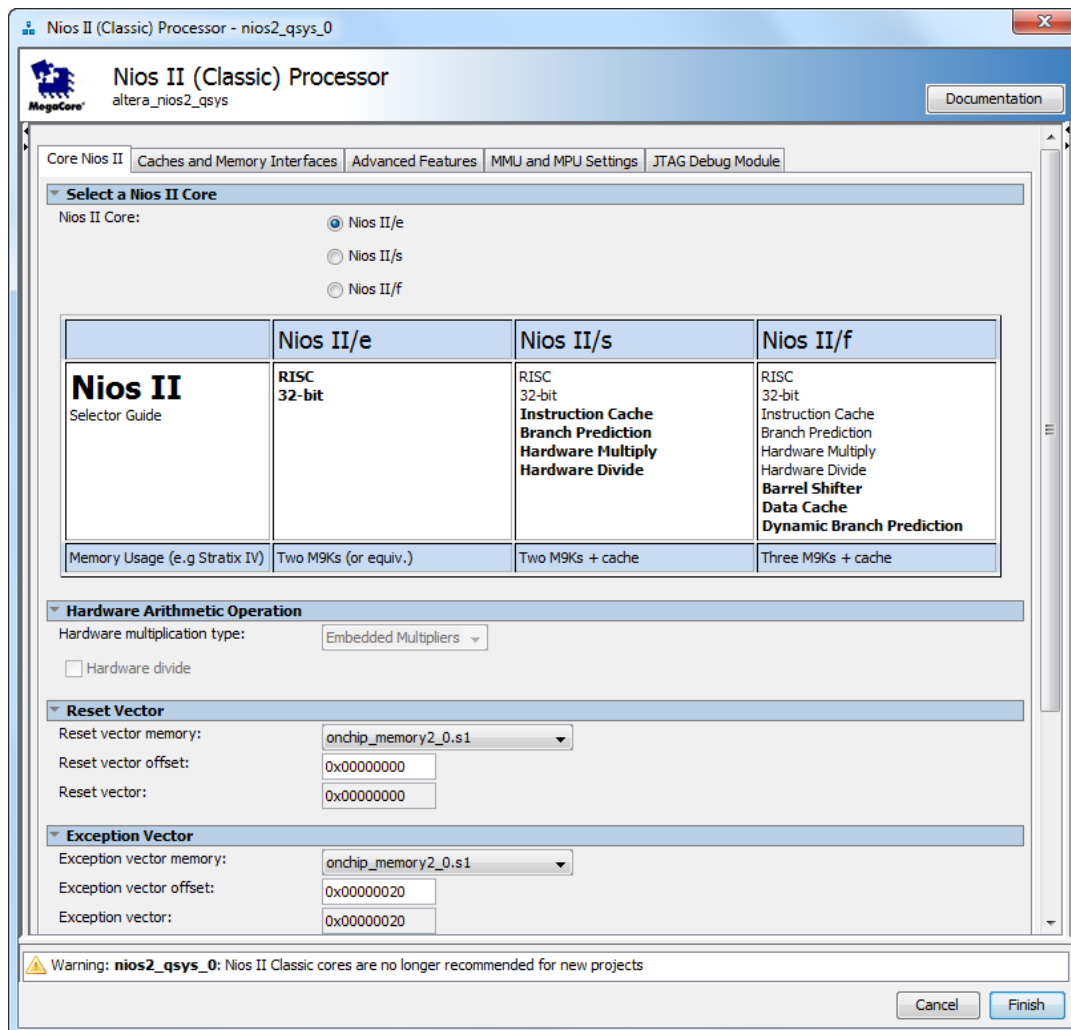


Figure 11. Setting the reset and exception vector memories.

The screenshot shows the Qsys IDE interface for a project named 'Qsys - unsaved.qsys'. The main window displays the 'System: unsaved' configuration with a hierarchical tree of components and their connections. The 'Connections' table lists components like 'clk_0', 'nios2_qsys_0', and 'onchip_memory2_0' with their respective descriptions and clock/base/end values. The 'Messages' pane at the bottom shows '0 Errors, 1 Warning'.

Use	Connections	Name	Description	Ex...	Clock	Base	End	II
<input checked="" type="checkbox"/>		clk_0	Clock Source					
		clk_in	Clock Input	clk	exported			
		clk_in_reset	Reset Input	reset				
		clk	Clock Output		clk_0			
		clk_reset	Reset Output					
<input checked="" type="checkbox"/>		nios2_qsys_0	Nios II (Classic) Processor					
		clk	Clock Input	Double	clk_0			
		reset_n	Reset Input	Double	[clk]			
		data_master	Avalon Memory Mapped Master	Double	[clk]			
		instruction_master	Avalon Memory Mapped Master	Double	[clk]			
		d_irq	Interrupt Receiver	Double	[clk]		IRQ 0	IRQ 31
		jtag_debug_module_reset	Reset Output	Double	[clk]			
		jtag_debug_module	Avalon Memory Mapped Slave	Double	[clk]	2800	0x2fff	
		custom_instruction_master	Custom Instruction Master	Double				
<input checked="" type="checkbox"/>		onchip_memory2_0	On-Chip Memory (RAM or ROM)					
		clk1	Clock Input	Double	clk_0			
		s1	Avalon Memory Mapped Slave	Double	[clk1]	1000	0x1fff	
		reset1	Reset Input	Double	[clk1]			

Current filter:

Messages

Type Path Message

0 Errors, 1 Warning

Generate HDL... Finish

Altera Corporation - University Program
May 2016

Now, we will create the new Qsys component for our 32-bit register, and make this component available in the Qsys IP Catalog. To create a new component, click the **New Component...** button in the IP Catalog area of the Qsys window. The Component Editor tool, shown in Figure 13, will appear. It has four tabs.

The first step in creating a component is to specify where in the IP Catalog our new component will appear. In the current tab, **Component Type**, change the **Name** to *reg32_avalon_interface*, the **Display name** to *reg32_component*, and provide a name for the **Group** setting, such as *My Own IP Cores*.

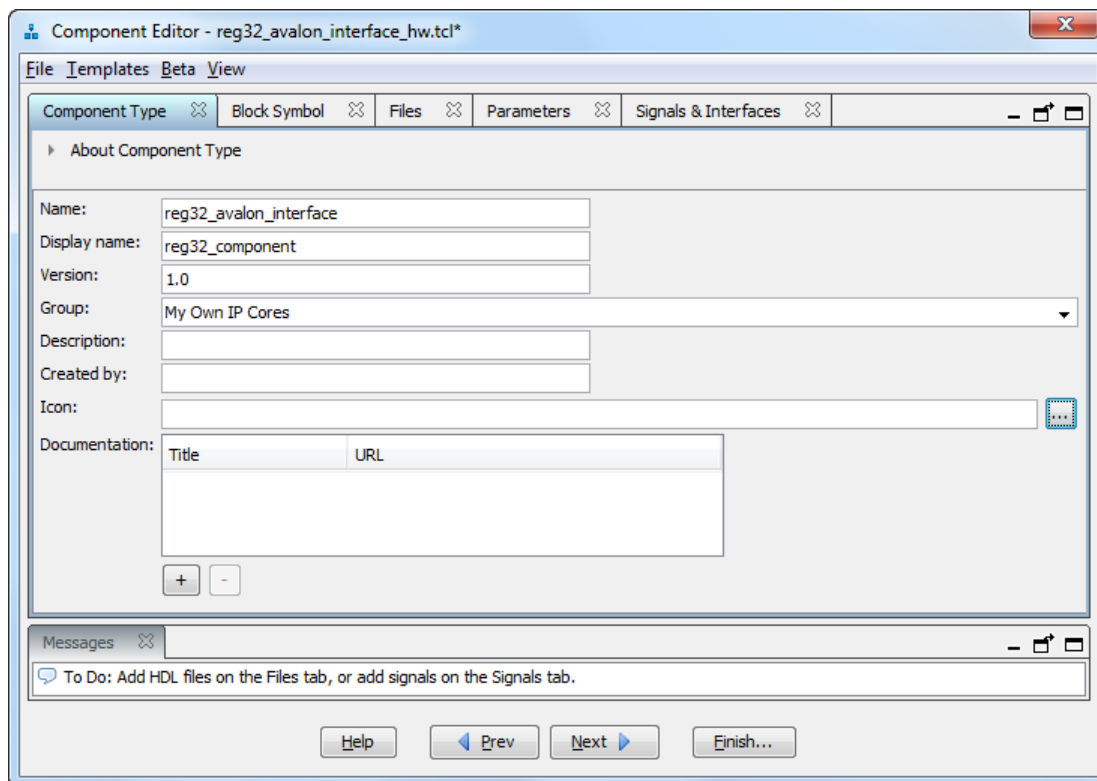


Figure 13. Component Editor window.

Next, we add the files that describe the component. Go to the **Files** tab, depicted in Figure 14, and then click on the **Add File...** button under **Synthesis Files** to browse and select the top-level file `reg32_avalon_interface.v`. Run the analysis of the top-level file by clicking on the **Analyze Synthesis Files** button. Qsys analyzes this file to determine the types of interfaces that are used by the component. Optionally, you can also add the file `reg32.v` to the list of *Synthesis Files*. Then click the **Copy from Synthesis Files** button under **Verilog Simulation Files** to add the files for simulation. If the Component Editor finds any errors when analyzing the top-level file, then they will need to be fixed and the code re-analyzed. Once no syntax errors are present, then the next step is to specify the types of interfaces that are used by the component.

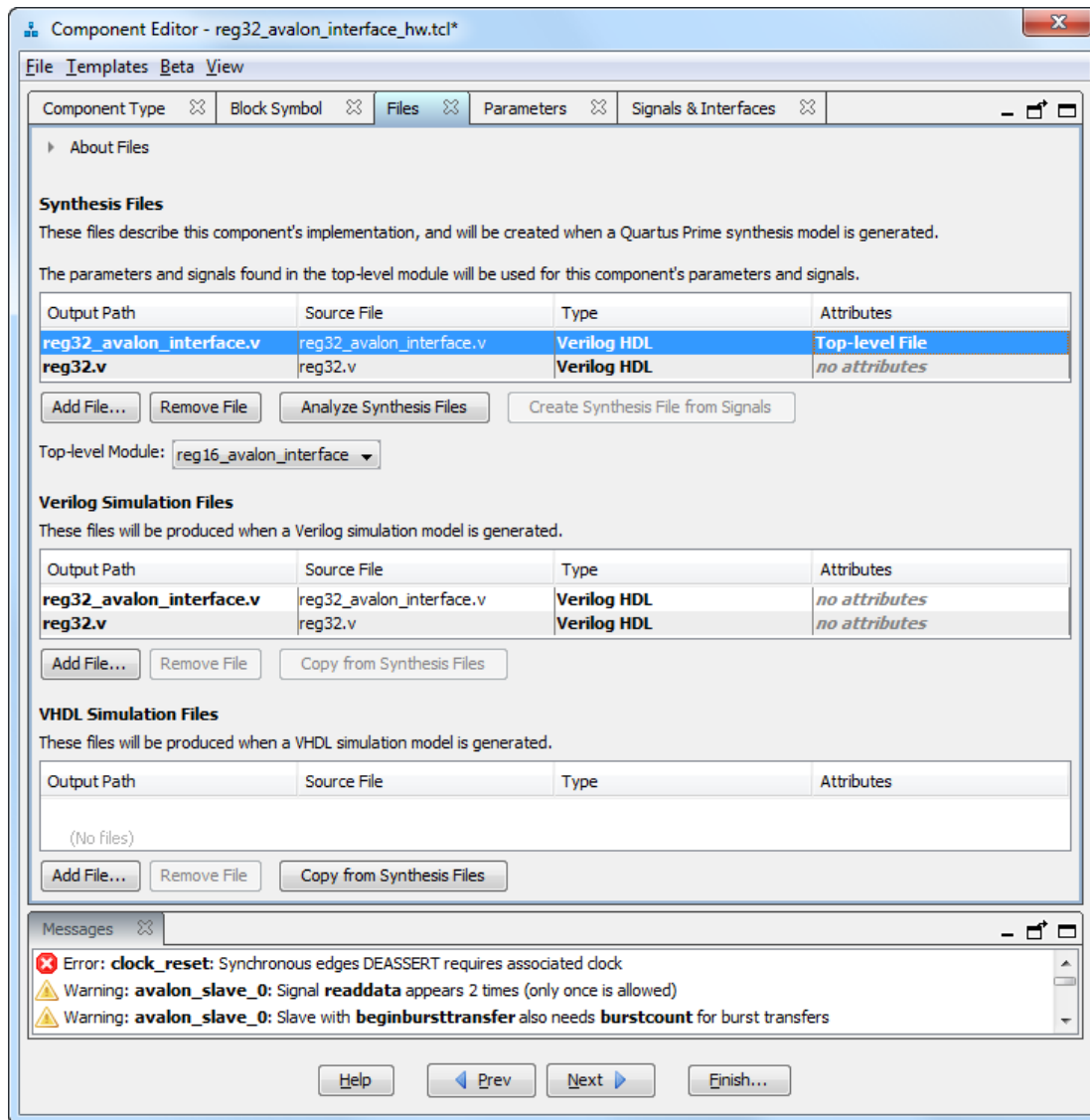


Figure 14. Adding HDL files that define the new component.

Click on the **Signals & Interfaces** tab to specify the meaning of each interface port in the top-level entity. This leads to the window in Figure 15.

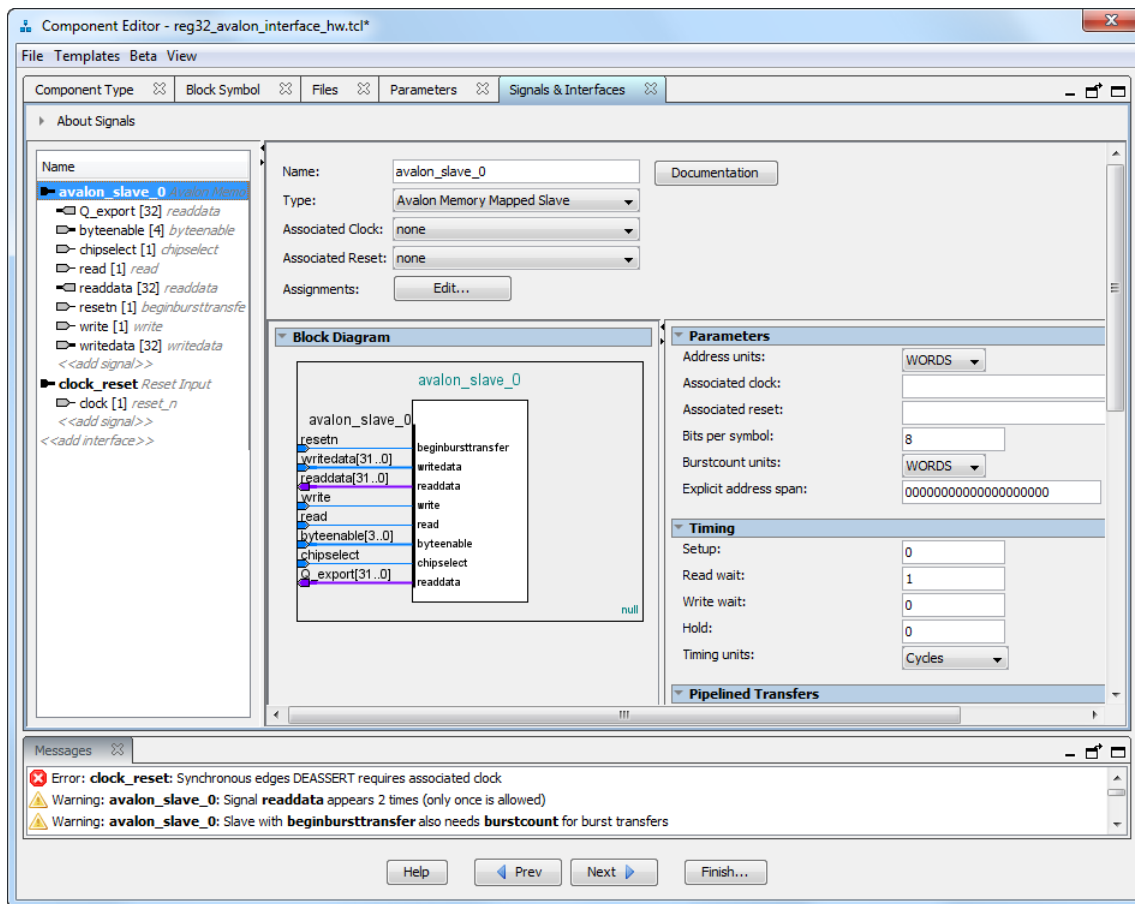


Figure 15. Initial settings for component signals.

To define correctly the meaning of each signal, it is necessary to specify the correct types of interface, put the signals in the correct interface, and specify the signal type for each signal. For the *clock [1]* signal, select **<<add interface>>** and select *Clock Input* as in Figure 16. Now drag and drop the signal *clock [1]* into *Clock Input* interface and change its *Signal Type* to *clk*, as shown in Figure 17.

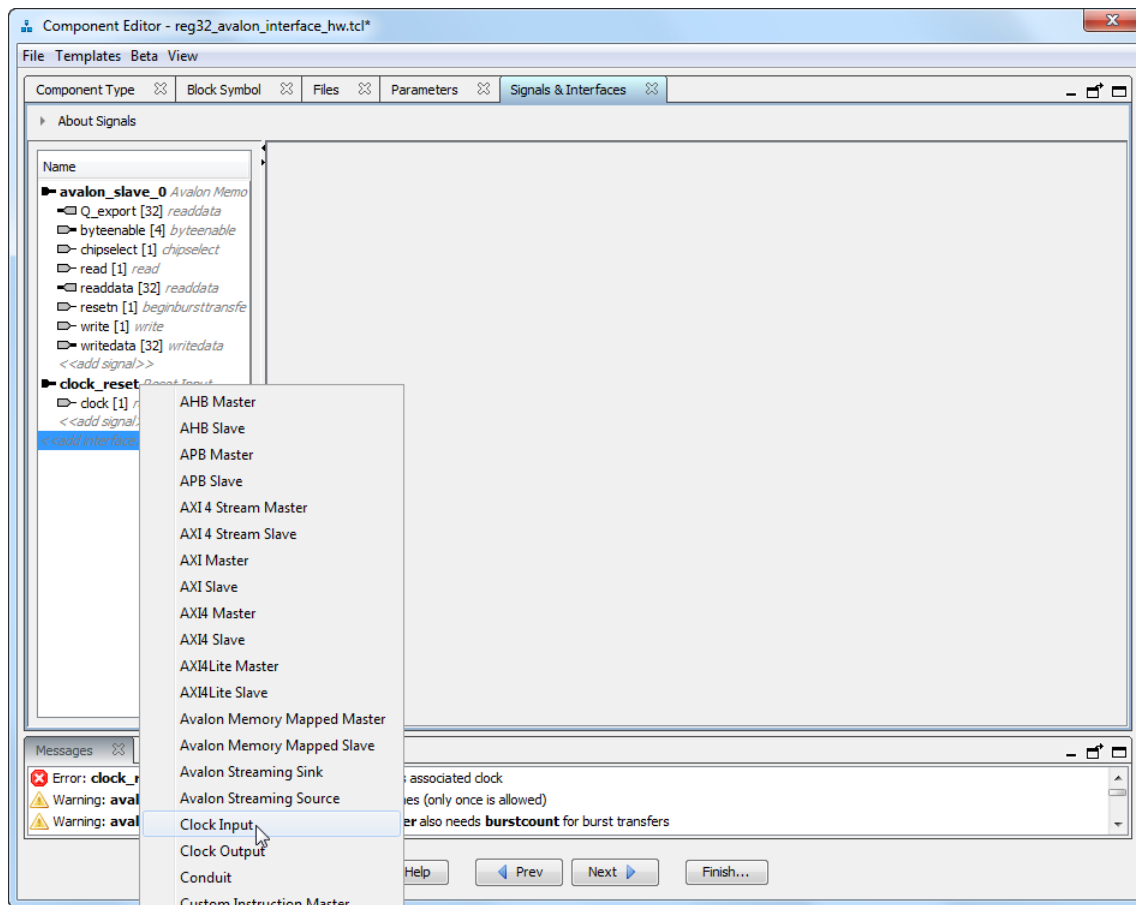
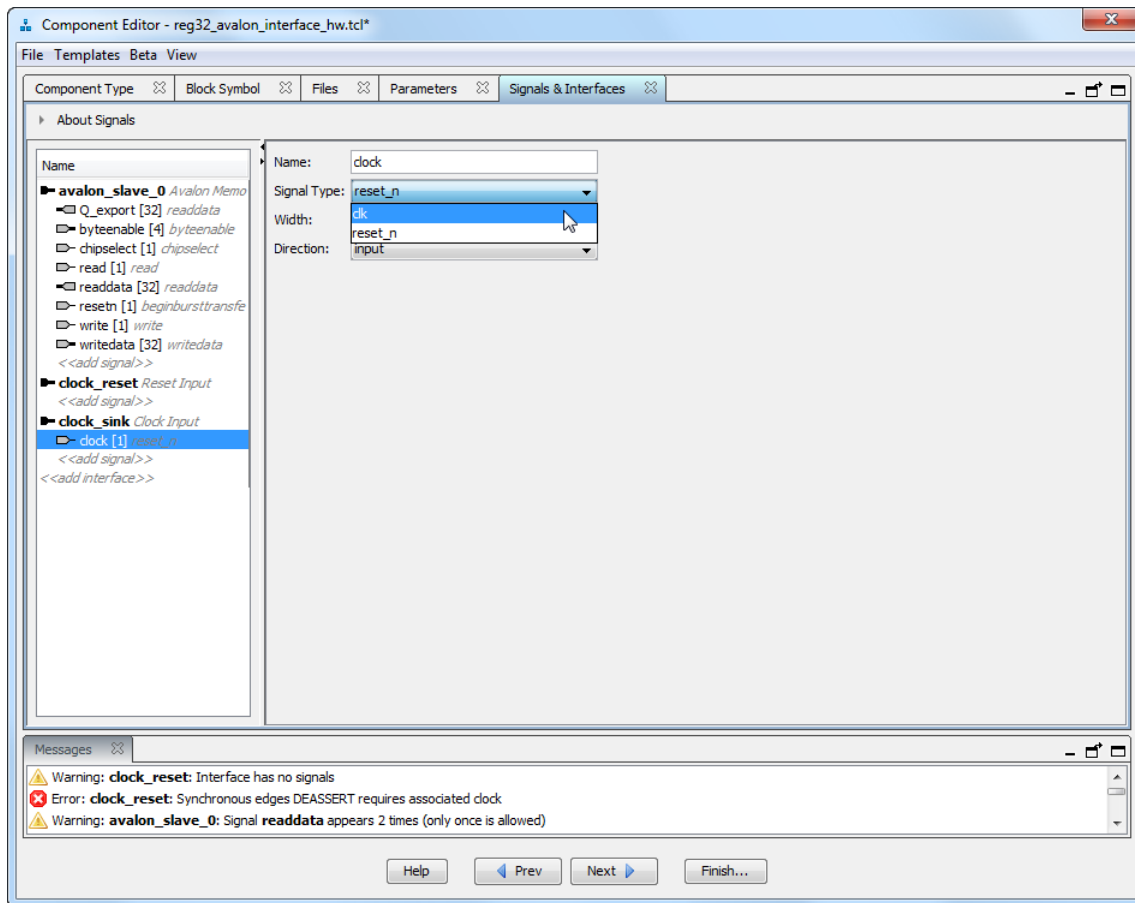


Figure 16. Creating the *Clock Input* interface.

Figure 17. Specifying the signal type for *clock*.

For the *resetn* signal, drag and drop it into the *Reset Input* interface and change its signal type to *reset_n*, as indicated in Figures 18 and 19.

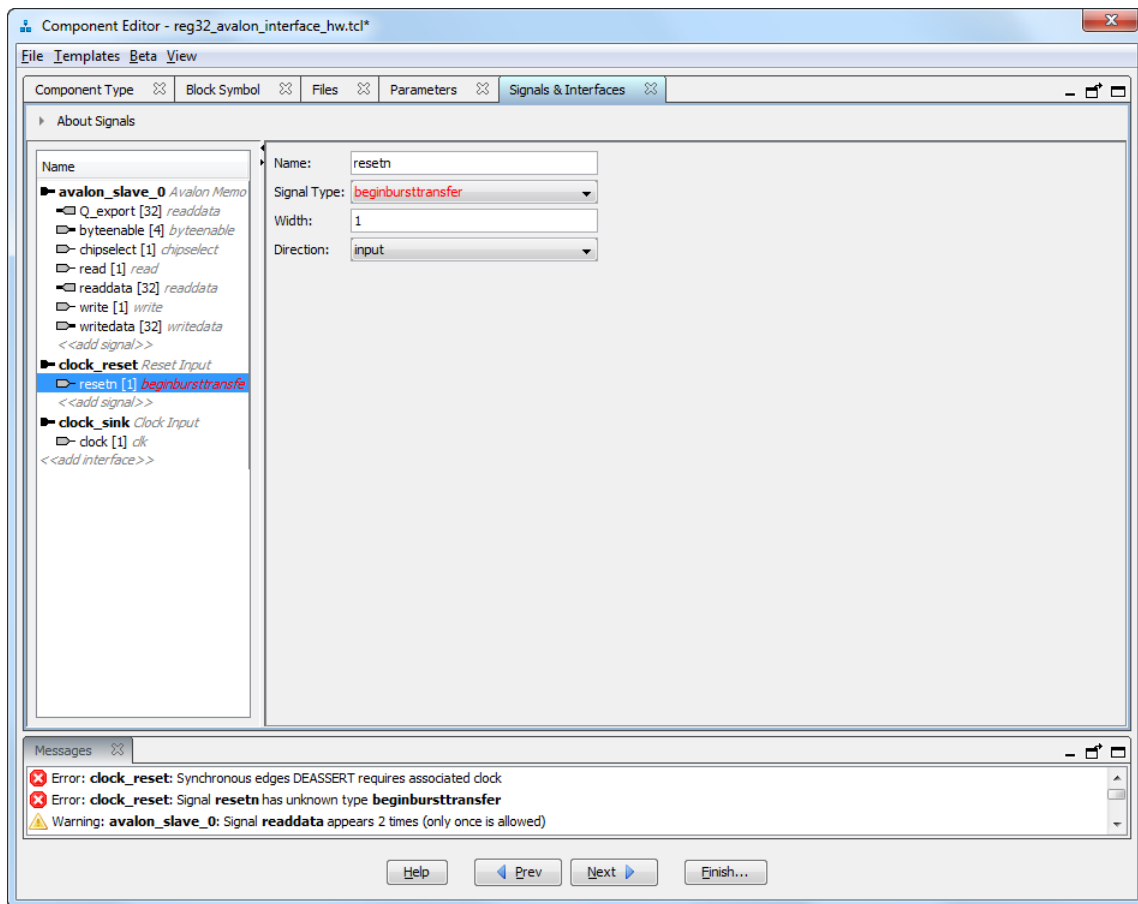
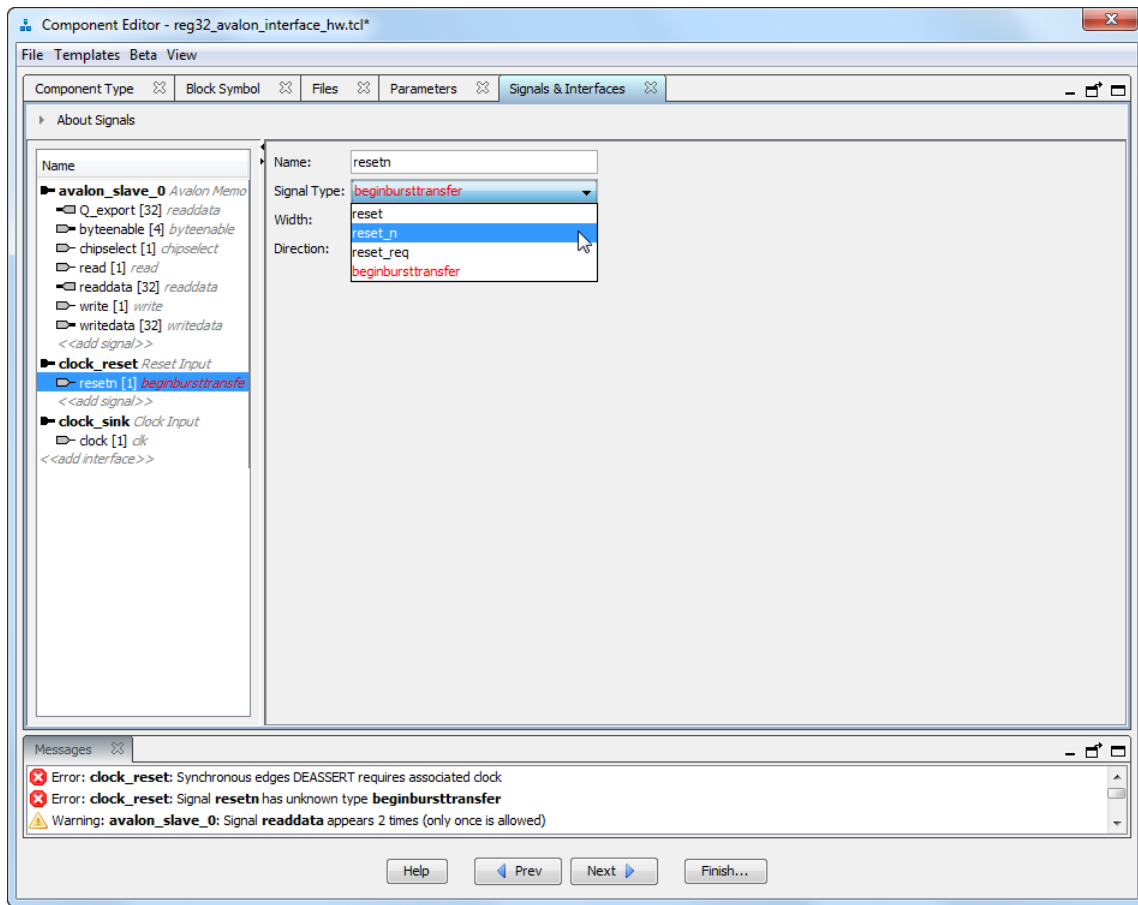


Figure 18. Changing the interface for *resetn*.

Figure 19. Specifying the signal type for *resetn*.

Finally, the *Q_export* signal must be visible outside the Qsys-generated system; it requires a new interface which is not a part of the Avalon Memory-Mapped Interface. Click on the <<add interface>>, name it *conduit_end*, and specify its type as *Conduit*, as shown in Figure 20. Drag and drop *Q_export* into the newly created conduit interface. The *Signal Type* for a conduit signal does not matter, so *Q_export* does not need to be edited. The rest of the signals shown in the Component Editor already have correct interface types as their names are recognizable as specific Avalon signals. The Component Editor window should now appear as shown in Figure 21.

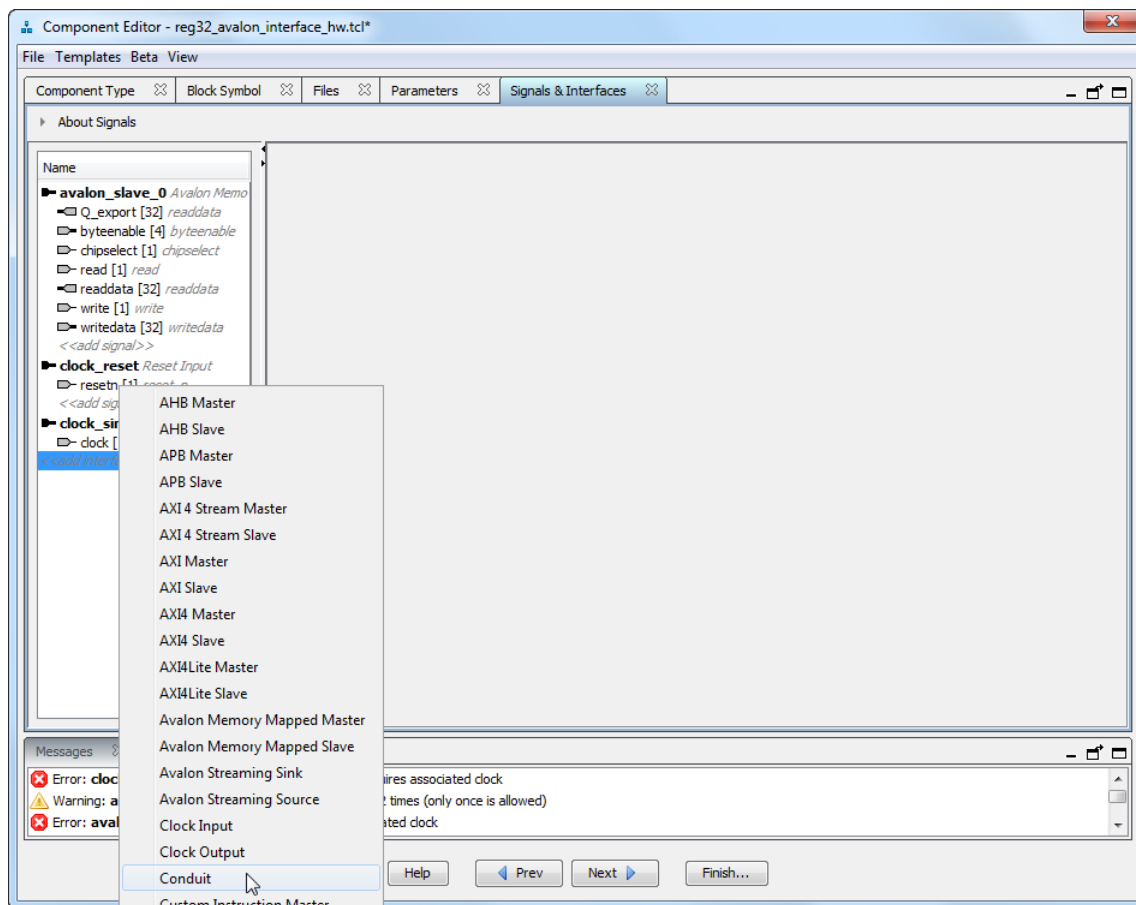


Figure 20. Creating an external interface for *Q_export*.

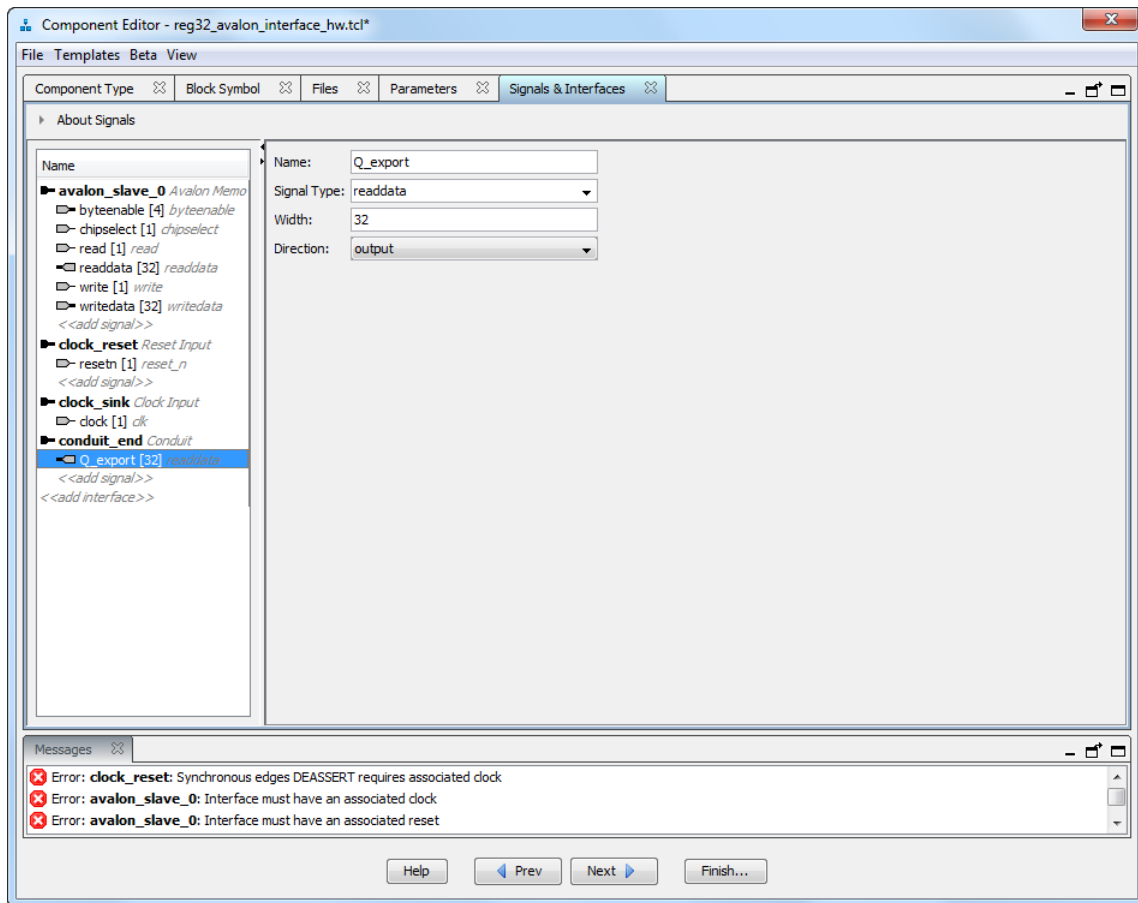


Figure 21. Final settings for component signals.

Note that there are still some error messages. The first error message states that the *avalon_slave_0* interface must have an Associated Clock and an Associated Reset. Select *clock_sink* as this clock and *clock_reset* as the reset, as indicated in Figure 22. Also note in Figure 22 that under the Timing heading we have changed the parameter called Read wait for the *avalon_slave_0* interface from its default value, which was 1, to the value 0. This parameter represents the number of Avalon clock signals that the component requires in order to respond to a read request. Our register can respond immediately, so we do not need to use the default of 1 wait cycle.

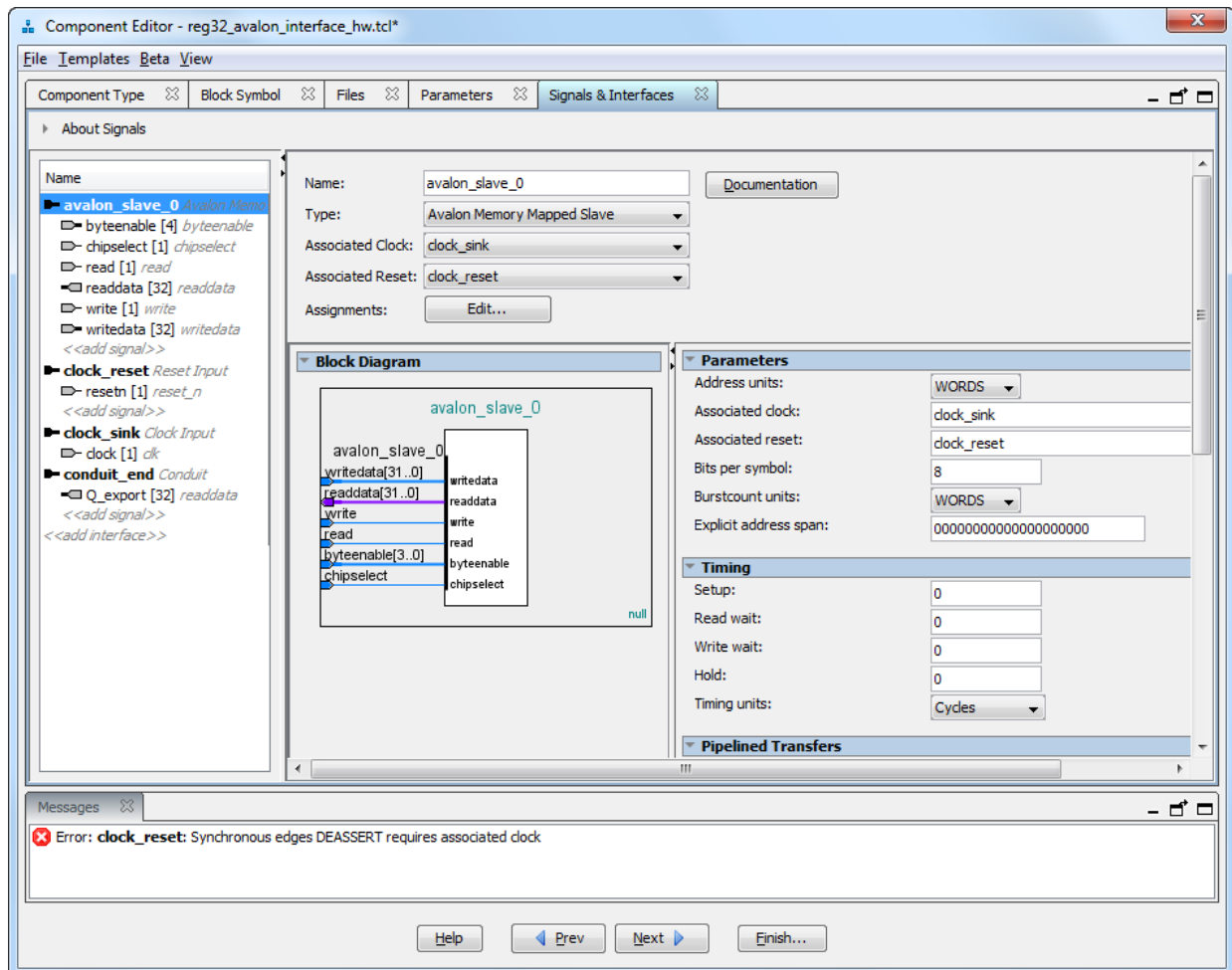


Figure 22. Specifying the clock and reset associated with the Avalon Slave interface.

The remaining error messages state that the *clock_reset* interface must have an associated clock. Set this clock to *clock_sink*, as depicted in Figure 23. Now, there should be no error messages left. Click Finish to complete the creation of the Qsys component, and save the component when prompted to do so.

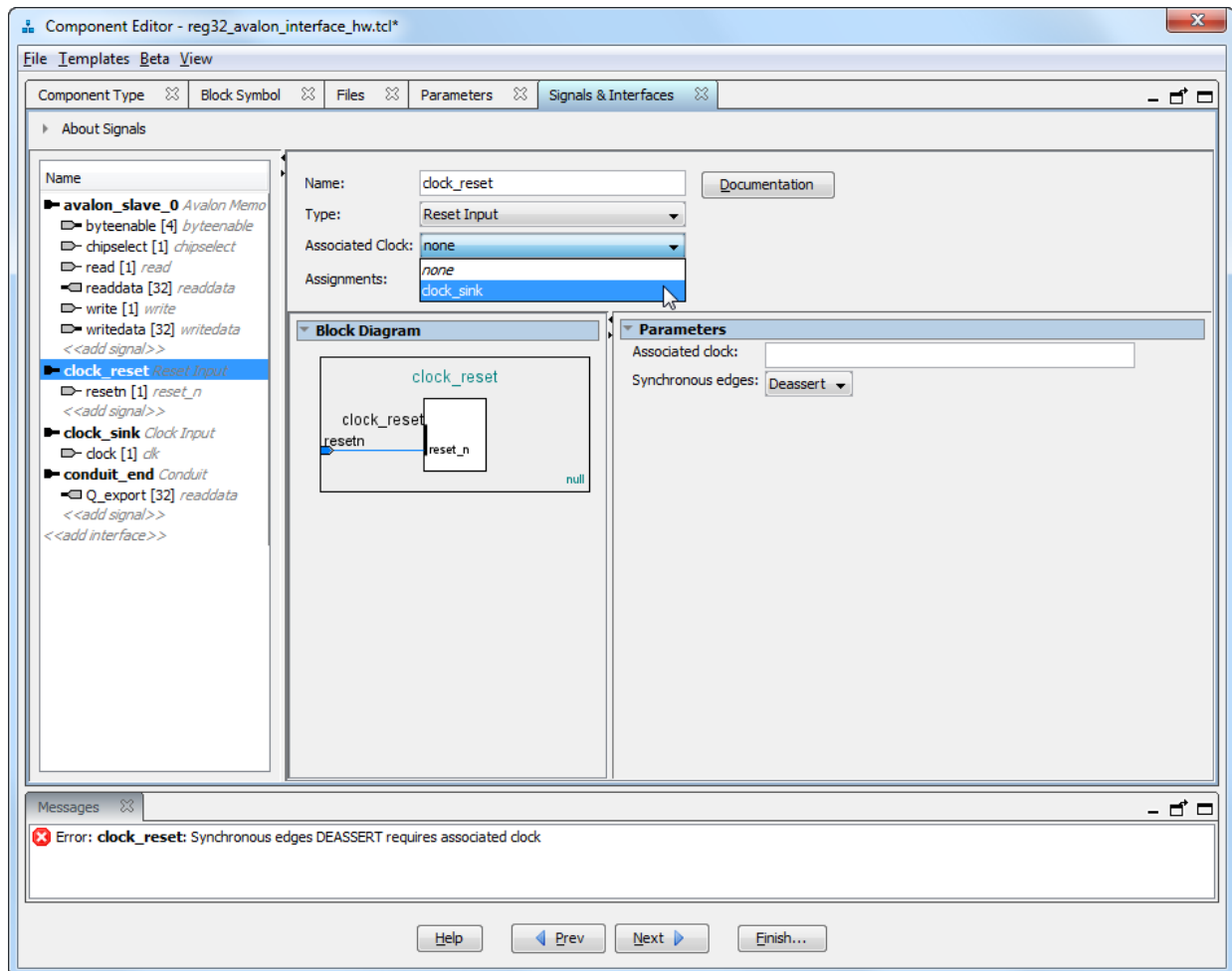


Figure 23. Specifying the clock associated with the reset interface.

6 Instantiating the New Component

In the Qsys IP Catalog, expand the newly-created item *My Own IP Cores*. Add an instance of the *reg32_component*, to open the window shown in Figure 24. Click Finish to return to the main Qsys window. Next, make the connections shown in Figure 25 to attach the register component to the required clock and reset signals, as well as to the data master port of the Nios II processor. Finally, as indicated in the Export column in Figure 25, click on Double-click to export for the Conduit and specify the name *to_hex*. Notice in the Base address column in Figure 25 that the assigned address of the new register component is 00000000. This address can be directly edited by the user, or it can be assigned automatically by using the Assign Base Addresses command in the System menu. In this

tutorial, we will leave the address as 00000000.

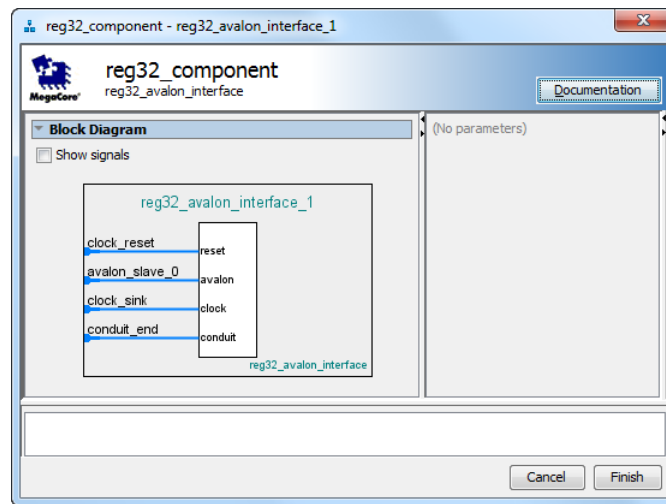


Figure 24. Adding the *reg32_component* to the base system.

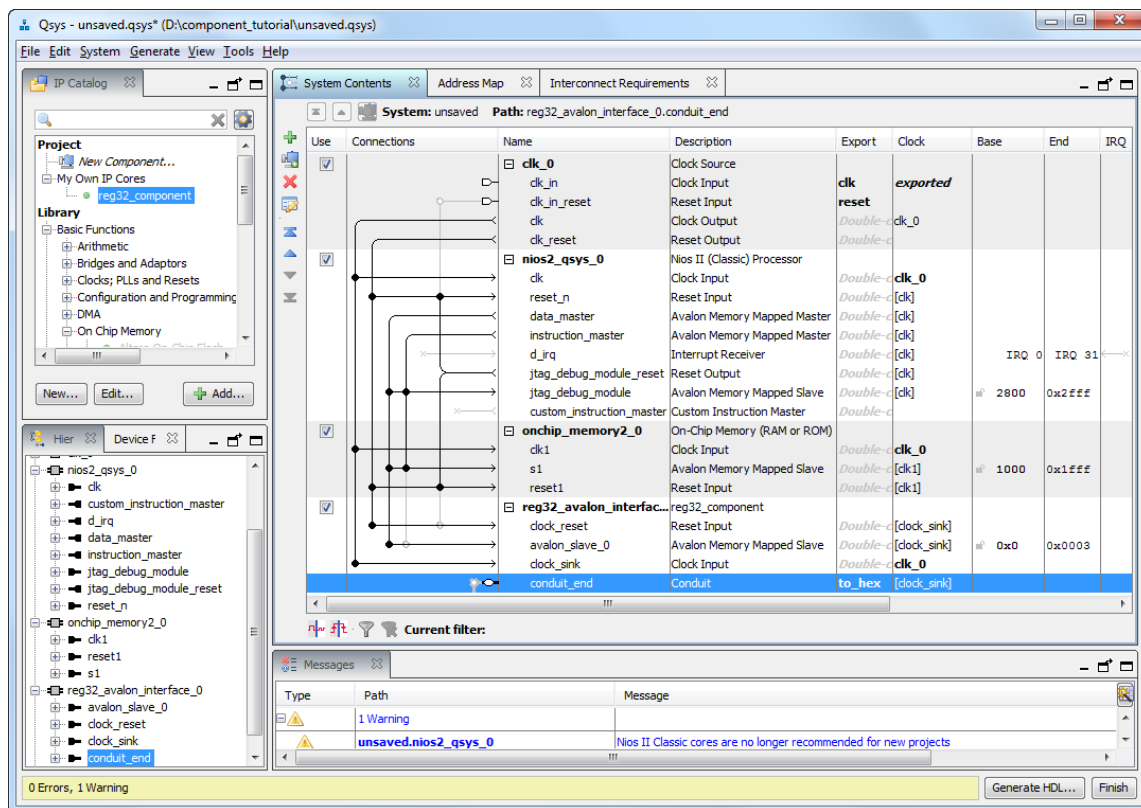


Figure 25. Required connections for the new component.

Use the **Save** command in the **File** menu to save the defined Qsys system using the name *embedded_system*. Next, in the Qsys window select **Generate > Show Instantiation Template...**, the window in Figure 26 will show up. This window gives an example of how the embedded system defined in the Qsys tool can be instantiated in HDL code. Note that the clock input of our embedded system is called *clk_clk*, the reset input is called *resetn_reset_n*, and the conduit output is named *to_hex_readdata*.

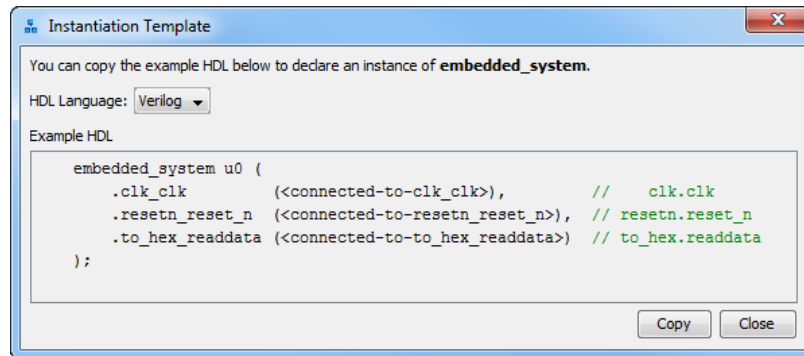


Figure 26. The HDL Example tab.

Finally, open the Generation window in the Qsys tool, shown in Figure 27 by selecting **Generate > Generate HDL**, and then click the **Generate** button. This action causes the Qsys tool to generate HDL code that specifies the contents of the embedded system, including all of the selected components and the Avalon interconnection fabric.

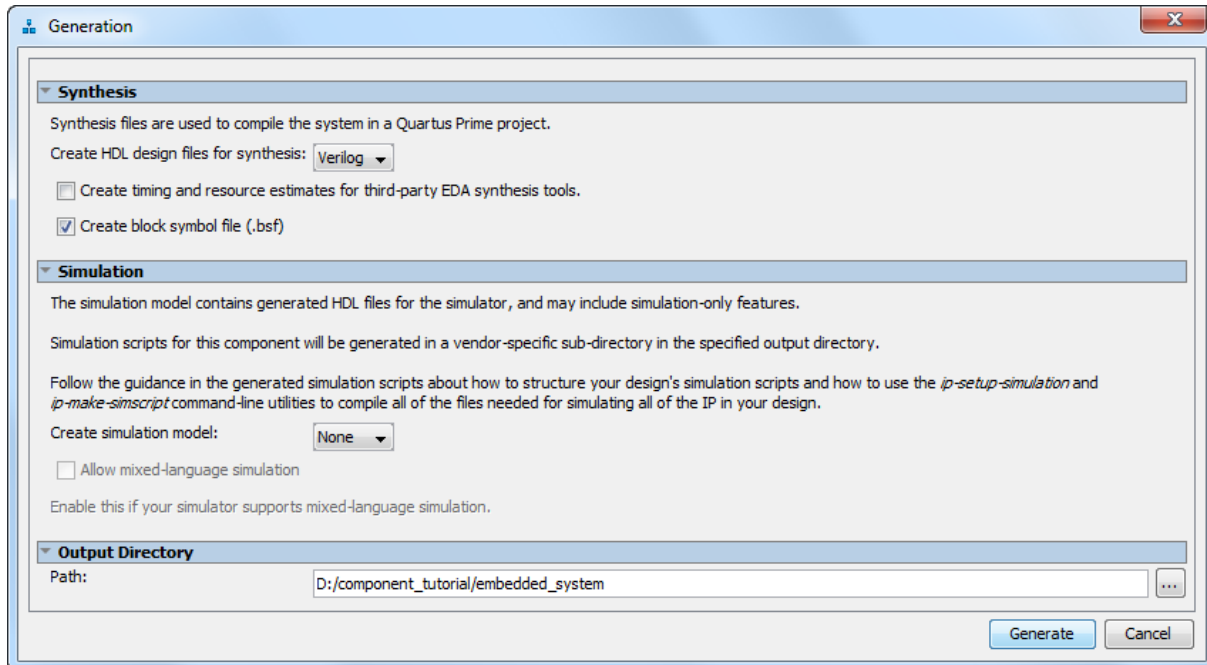



Figure 27. The Generation tab.

Close the Qsys tool to return to the main Quartus Prime window. Next, select **Project > Add/Remove Files in Project...** from the main Quartus Prime window, and then browse on the  button to open the window in Figure 28. Browse to the folder called *embedded_system/synthesis* and then select the file named *embedded_system.qip*. This file provides the information needed by the Quartus Prime software to locate the HDL code generated by the Qsys tool. In Figure 28 click **Open** to return to the **Settings** window and then click **Add** to add the file to the project. Click **OK** to return to the main Quartus Prime window.

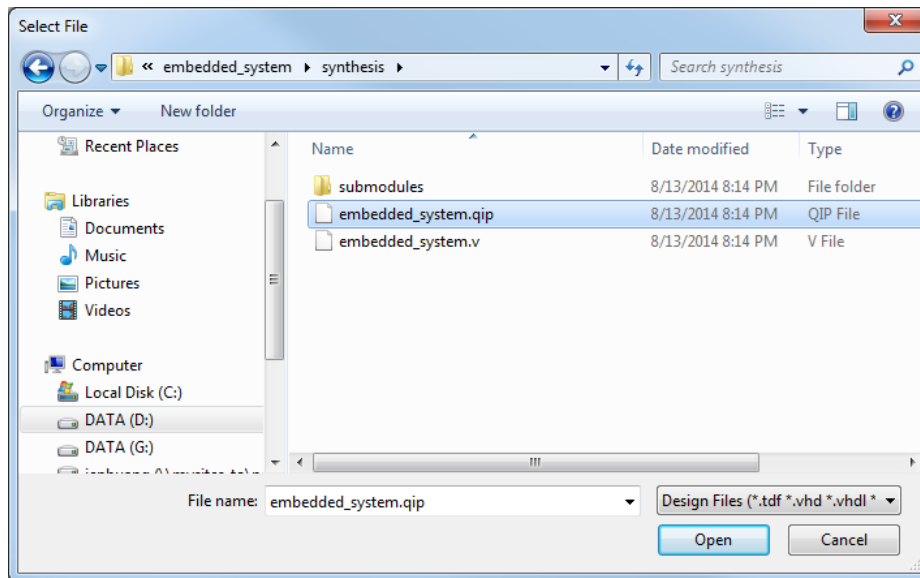


Figure 28. Adding the .qip file to the Quartus Prime project.

7 Implementing the Embedded System in an FPGA Chip

To implement the Qsys-generated embedded system in an FPGA chip, we need to create a top-level HDL module which instantiates the embedded system and has the appropriate input and output signals. A suitable HDL module is given in Figures 29 and 30, in Verilog and VHDL. The module connects the 50 MHz clock signal, *CLOCK_50*, on the DE-series board to the clock input of the embedded system, and connects *KEY₀* to the reset input. The external conduit from the embedded system is connected to the seven segment displays *HEX0*, ..., *HEX3*. The HDL code for the 7-segment display code converter, called *hex7seg*, is provided in Appendix A, in Figures 35 and 36.

Store the code for the top-level module in a file called *component_tutorial.v* (or *.vhd*), and store the code for the seven-segment code converter in a file called *hex7seg.v* (or *.vhd*). Include appropriate pin assignments in the Quartus Prime project for the *CLOCK_50*, *KEY₀*, and *HEX0*, ..., *HEX3* signals on the DE-series board. If all the necessary pin assignments are not made, it may not be possible to connect to the board (from the Quartus Prime software, or the Altera Monitor Program). For instructions on adding pin assignments, see the Quartus Introduction tutorial.

Compile the project. After successful compilation, download the circuit onto the DE-series board by using the Quartus Prime Programmer tool. See the Quartus Introduction tutorial for instructions on downloading a circuit to a board.

```
module component_tutorial (CLOCK_50, KEY, HEX0, HEX1, HEX2, HEX3);
    input CLOCK_50;
    input [0:0] KEY;
    output [0:6] HEX0, HEX1, HEX2, HEX3;

    wire [15:0] to_HEX;

    embedded_system U0 (
        .clk_clk(CLOCK_50), .resetn_reset_n(KEY[0]), .to_hex_readdata(to_HEX) );

    hex7seg h0(to_HEX[3:0], HEX0);
    hex7seg h1(to_HEX[7:4], HEX1);
    hex7seg h2(to_HEX[11:8], HEX2);
    hex7seg h3(to_HEX[15:12], HEX3);
endmodule
```

Figure 29. Verilog code for the top-level module.

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY component_tutorial IS
    PORT ( CLOCK_50 : IN STD_LOGIC;
          KEY       : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
          HEX0      : OUT STD_LOGIC_VECTOR(0 TO 6);
          HEX1      : OUT STD_LOGIC_VECTOR(0 TO 6);
          HEX2      : OUT STD_LOGIC_VECTOR(0 TO 6);
          HEX3      : OUT STD_LOGIC_VECTOR(0 TO 6);
    END component_tutorial;

ARCHITECTURE Structure OF component_tutorial IS
    SIGNAL to_HEX : STD_LOGIC_VECTOR(31 DOWNTO 0);
    COMPONENT embedded_system IS
        PORT ( clk_clk      : IN STD_LOGIC;
              resetn_reset_n : IN STD_LOGIC;
              to_hex_readdata : OUT STD_LOGIC_VECTOR (31 DOWNTO 0) );
    END COMPONENT embedded_system;

    COMPONENT hex7seg IS
        PORT ( hex      : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
              display   : OUT STD_LOGIC_VECTOR(0 TO 6) );
    END COMPONENT hex7seg;
BEGIN
    U0: embedded_system PORT MAP (
        clk_clk      => CLOCK_50,
        resetn_reset_n => KEY(0),
        to_hex_readdata => to_HEX );
    h0: hex7seg PORT MAP (to_HEX(3 DOWNTO 0), HEX0);
    h1: hex7seg PORT MAP (to_HEX(7 DOWNTO 4), HEX1);
    h2: hex7seg PORT MAP (to_HEX(11 DOWNTO 8), HEX2);
    h3: hex7seg PORT MAP (to_HEX(15 DOWNTO 12), HEX3);
END Structure;

```

Figure 30. VHDL code for the top-level module.

8 Testing the Embedded System

One way to test the circuit is to use the Altera Monitor Program. Open the Monitor Program and create a new project called *component_tutorial* and select Nios II as the architecture. In the New Project Wizard, for the Specify a system screen choose <Custom System>. As shown in the figure, under System details browse to select the system description file called *embedded_system.sopcinfo*. Also, browse to select the Quartus Prime programming file called *component_tutorial.sof*, as illustrated in Figure 31. For the screen titled Specify a program type in the New Project Wizard, choose No Program. Press Finish. When prompted to download the system, as shown in Figure 32, press No since we have already done so with the Quartus Prime Programmer.

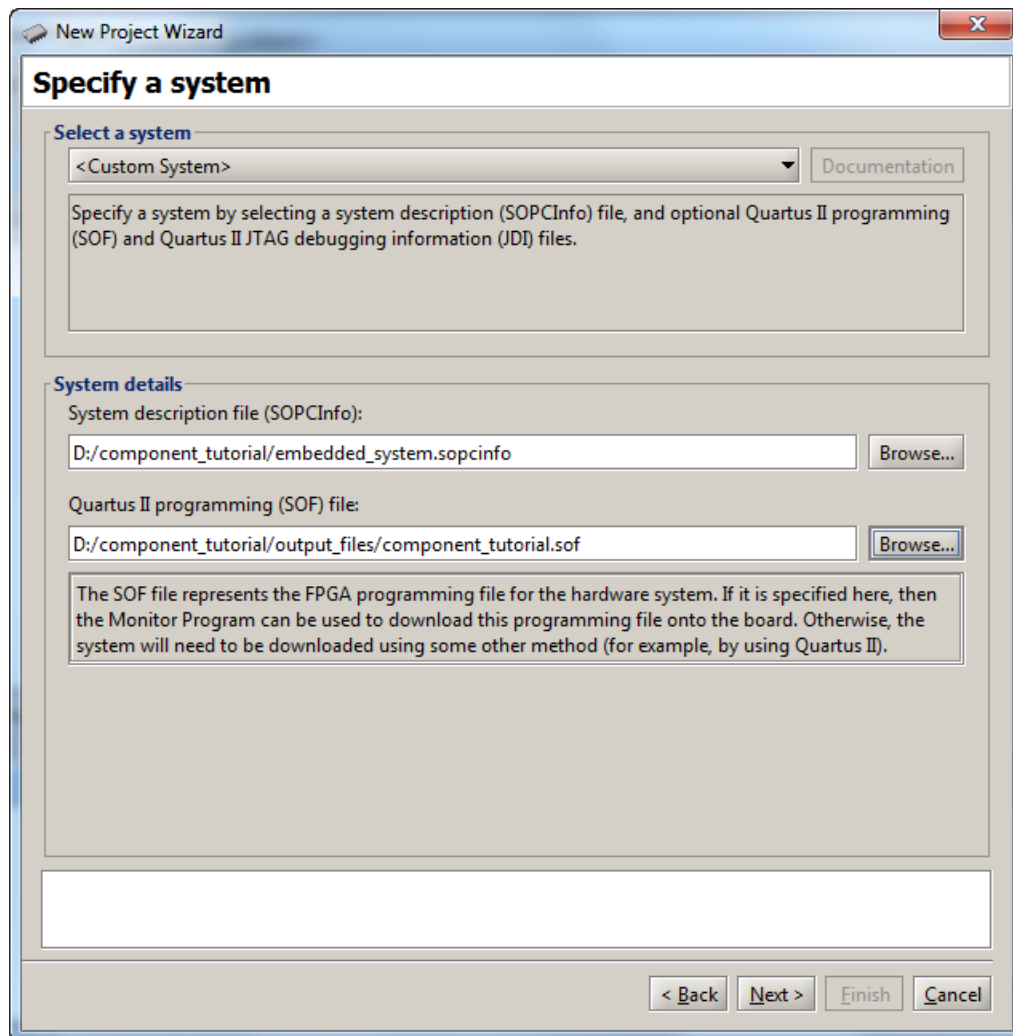


Figure 31. Specifying the system description file and Quartus Prime programming file.

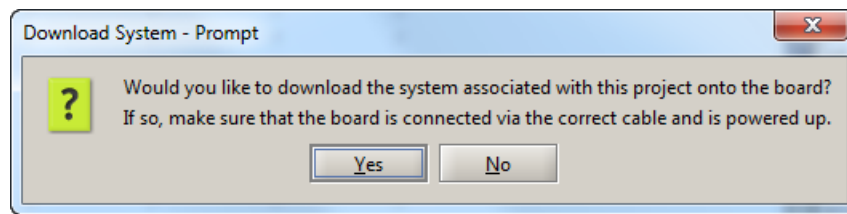


Figure 32. Press No to this prompt.

After successfully creating the Monitor Program project, click on the command **Connect to System** in the **Actions** menu. Open the **Memory** tab in the Monitor Program, and click the setting **Query Devices**, as indicated in Figure 33. Now, click the **Refresh** button to see that the content of address 0x00000000, which represents the 32-bit register component, has the value 00000000. Edit the value stored in the register, as illustrated in Figure 34, and observe the changes on the seven-segment displays on the DE-series board.

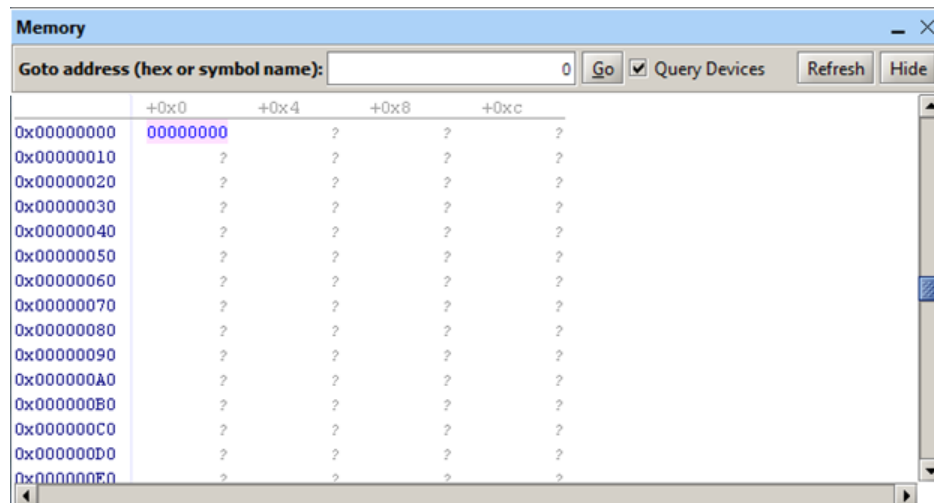


Figure 33. Using the Memory tab in the Monitor Program.

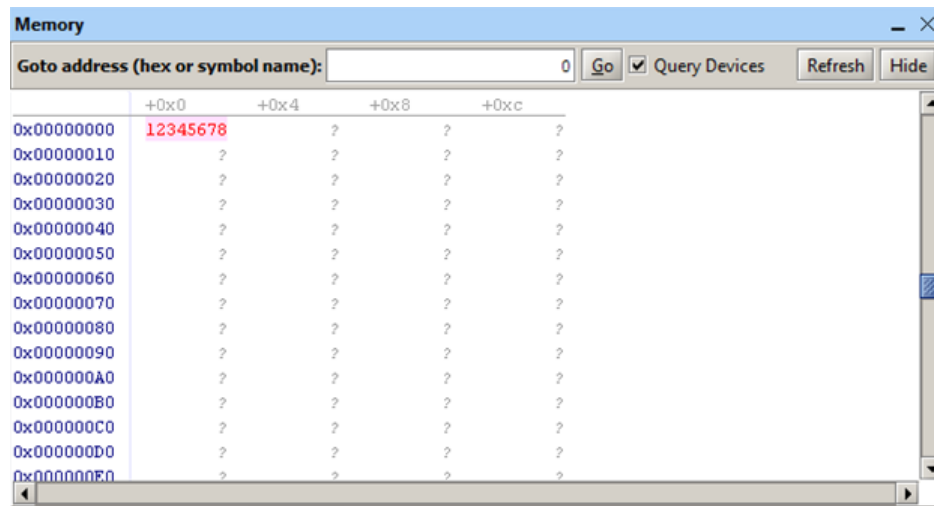


Figure 34. Changing the value stored in the 32-bit register.

9 Concluding Remarks

In this tutorial we showed how to create a component for use in a system designed by using the Qsys tool. Although the example is for a slave interface, the same procedure is used to create a master interface, with the only difference being in the type of an interface that is created for the component.

10 Appendix A

The HDL code for the seven-segment code converter that is instantiated in Figures 29 and 30 is shown in Figures 35 and 36.

```

module hex7seg (hex, display);
    input [3:0] hex;
    output [0:6] display;

    reg [0:6] display;
    /*
    *      - 0 -
    *    5 |   | 1
    *      - 6 -
    *    4 |   | 2
    *      - 3 -
    */
    always @ (hex)
        case (hex)
            4'h0: display = 7'b0000001;
            4'h1: display = 7'b1001111;
            4'h2: display = 7'b0010010;
            4'h3: display = 7'b0000110;
            4'h4: display = 7'b1001100;
            4'h5: display = 7'b0100100;
            4'h6: display = 7'b0100000;
            4'h7: display = 7'b0001111;
            4'h8: display = 7'b0000000;
            4'h9: display = 7'b0001100;
            4'hA: display = 7'b0001000;
            4'hb: display = 7'b1100000;
            4'hC: display = 7'b0110001;
            4'hd: display = 7'b1000010;
            4'hE: display = 7'b0110000;
            4'hF: display = 7'b0111000;
        endcase
    endmodule

```

Figure 35. Verilog code for the seven-segment display code converter.

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY hex7seg IS
    PORT ( hex      : IN STD_LOGIC_VECTOR(3 DOWNT0 0);
          display   : OUT STD_LOGIC_VECTOR(0 TO 6) );
END hex7seg;

ARCHITECTURE Behavior OF hex7seg IS
BEGIN
    --      - 0 -
    --    5 |    | 1
    --      - 6 -
    --    4 |    | 2
    --      - 3 -
    PROCESS (hex)
    BEGIN
        CASE hex IS
            WHEN "0000" => display <= "0000001";
            WHEN "0001" => display <= "1001111";
            WHEN "0010" => display <= "0010010";
            WHEN "0011" => display <= "0000110";
            WHEN "0100" => display <= "1001100";
            WHEN "0101" => display <= "0100100";
            WHEN "0110" => display <= "0100000";
            WHEN "0111" => display <= "0001111";
            WHEN "1000" => display <= "0000000";
            WHEN "1001" => display <= "0001100";
            WHEN "1010" => display <= "0001000";
            WHEN "1011" => display <= "1100000";
            WHEN "1100" => display <= "0110001";
            WHEN "1101" => display <= "1000010";
            WHEN "1110" => display <= "0110000";
            WHEN "1111" => display <= "0111000";
        END CASE;
    END PROCESS;
END Behavior;

```

Figure 36. VHDL code for the seven-segment display code converter.

Copyright © 2016 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

This document is being provided on an “as-is” basis and as an accommodation and therefore all warranties, representations or guarantees of any kind (whether express, implied or statutory) including, without limitation, warranties of merchantability, non-infringement, or fitness for a particular purpose, are specifically disclaimed.