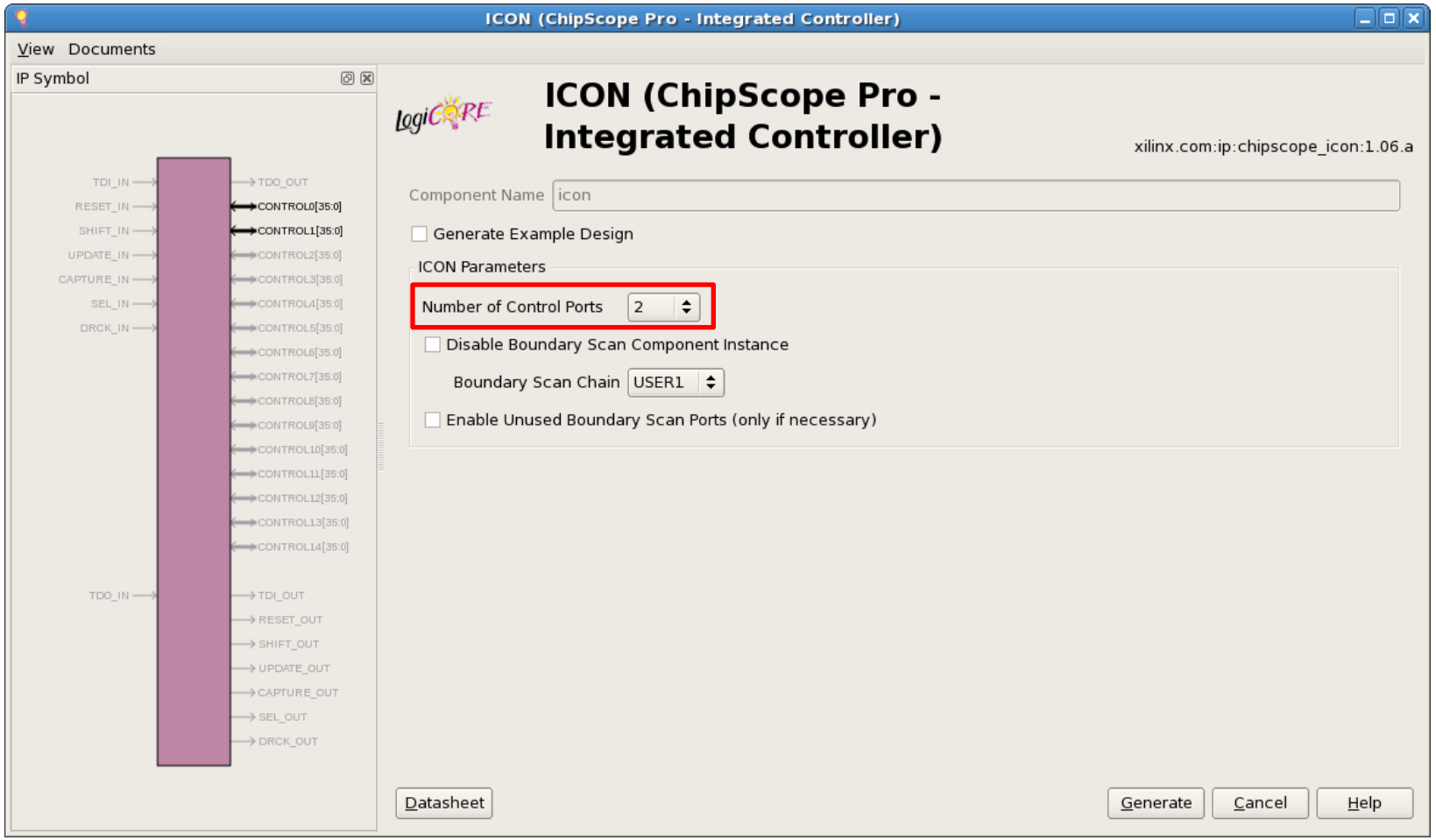
COE 758 - Xilinx ISE 13.4 Tutorial 3

Integrating Virtual I/O Into a Project

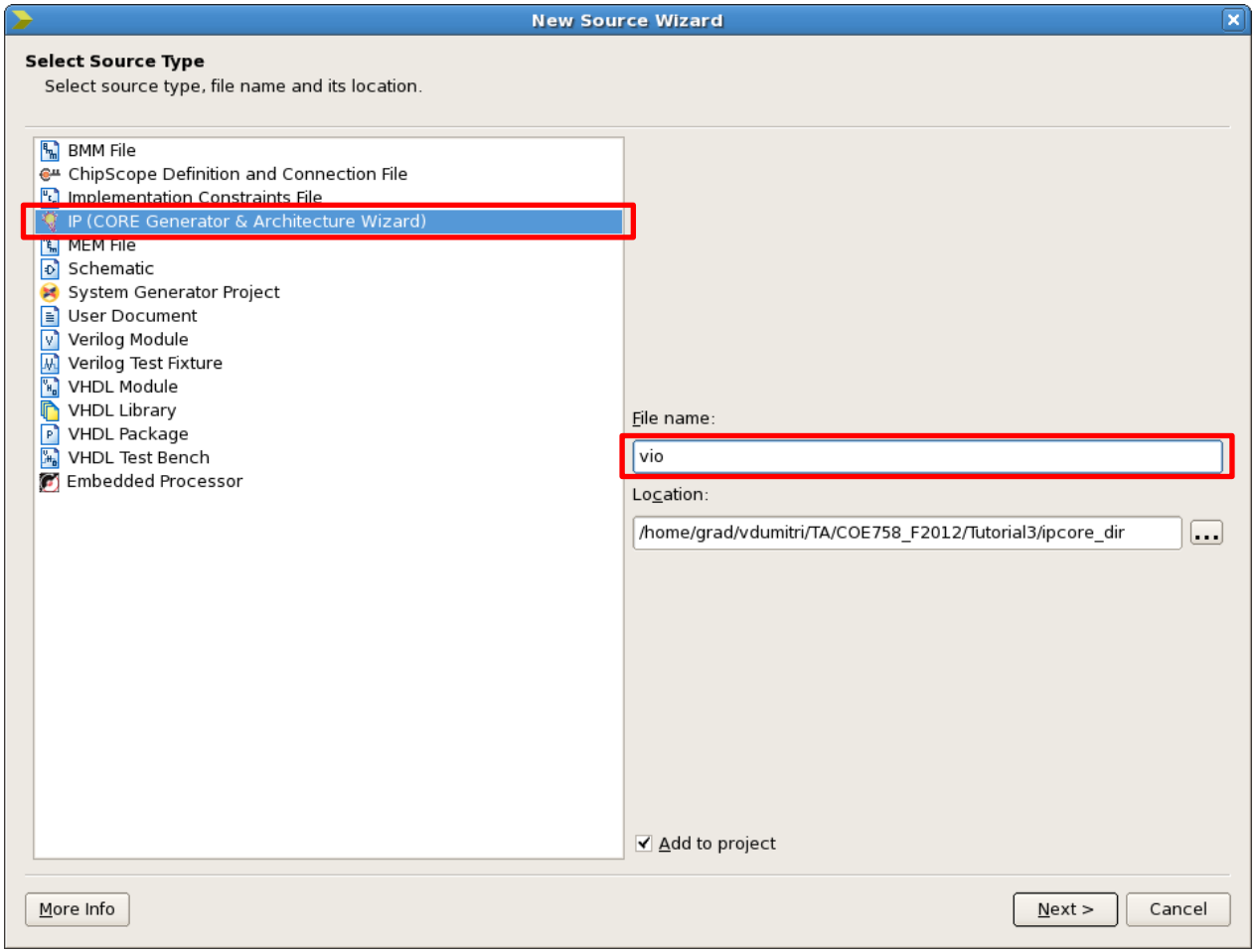
Generating and Using On-Chip BlockRAM Memory

ChipScope VIO Overview

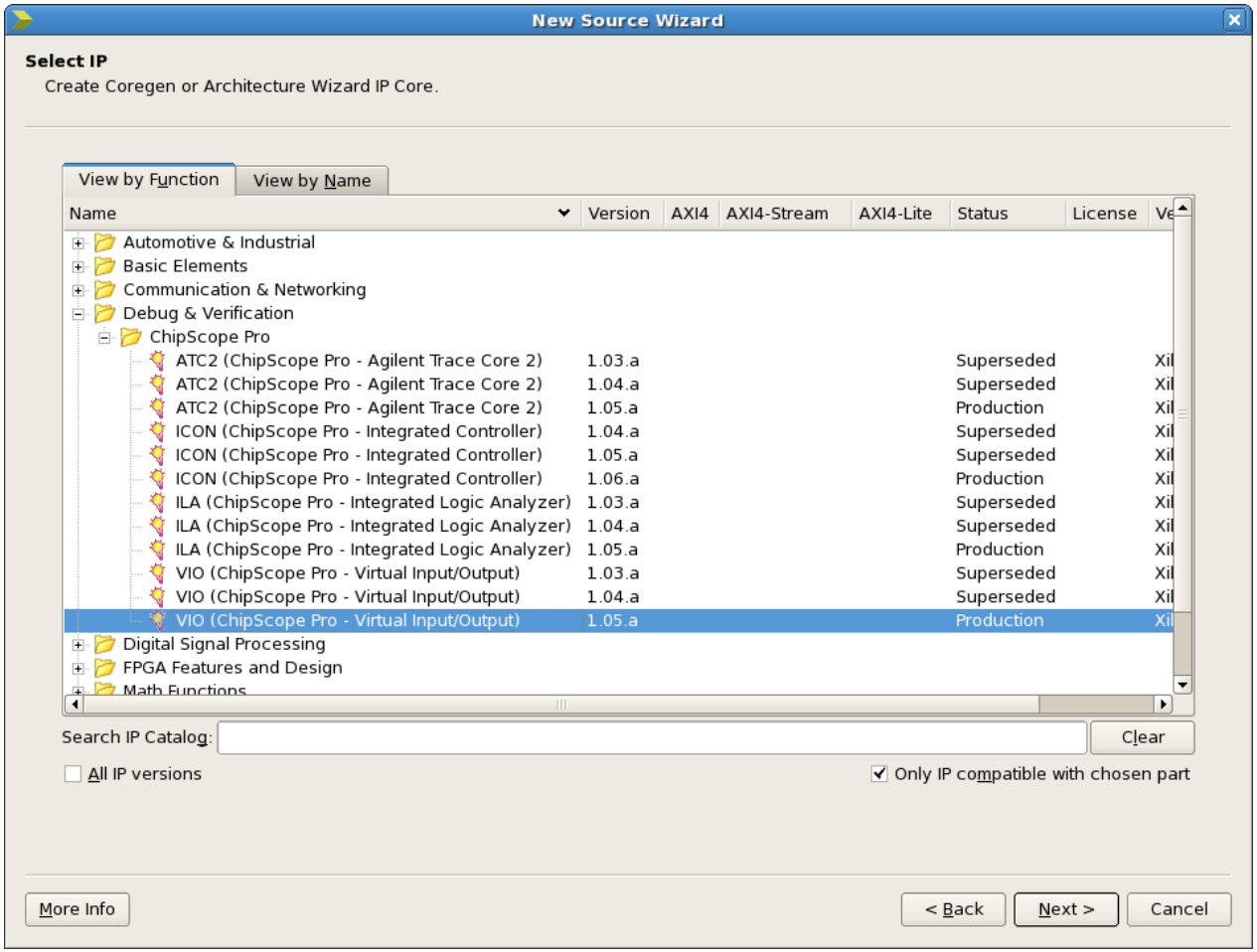
- In addition to the ICON and ILA cores, ChipScope also incorporates the Virtual I/O (VIO) core, which allows a designer to have direct input and output access to internal portions of their design via the use of the JTAG chain.
- The VIO core is instantiated inside a design in the same way as the ILA core, and is controlled by the ICON core.
- The VIO ports can be connected directly to signals inside a design, and will mimic regular (physical) I/O connections.
- This tutorial will describe how a VIO core can be generated, instantiated into a design, and used with the ChipScope Analyzer.
- The tutorial will also describe how to take advantage of the on-chip memory present inside an FPGA by generating and incorporating a Block RAM memory core into your design.
- Note that this project builds onto the subjects covered in Tutorials 1 and 2, and is an augmentation of the Tutorial 2 project. You can either use the same project, or make a new copy of the project for Tutorial 3.



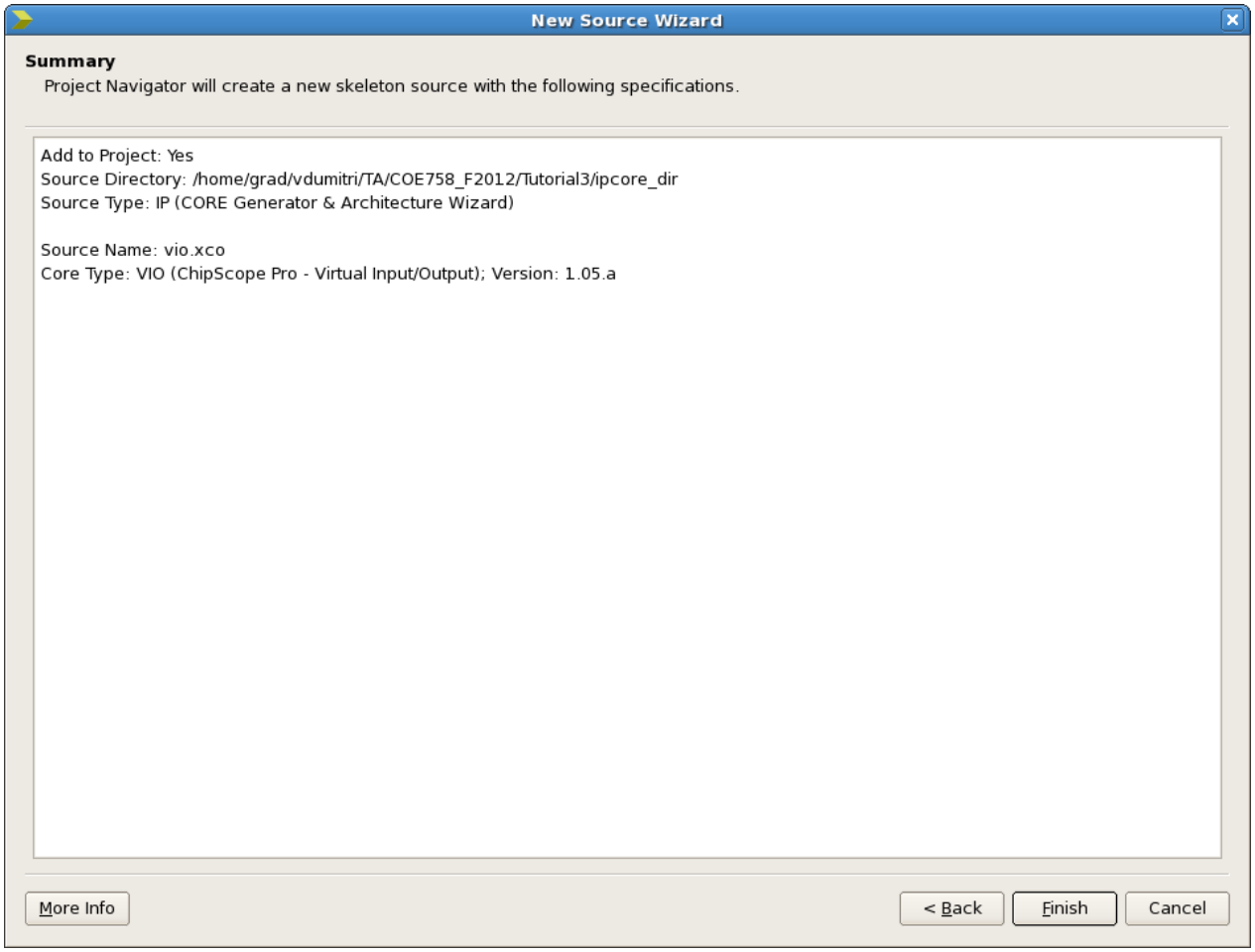
As mentioned previously, the VIO core is controlled by the ICON in the same way as the ILA. The ICON uses a 36-bit bus to control one ILA or VIO core. In order to control two such cores, the ICON must be regenerated to include two such control ports. To do this, double-click on the ICON component in the design navigator. This will bring up the window shown above. Change the **Number of Control Ports** entry to **2** and press Generate. You will receive a warning message indicating that a core with this name already exists and will be overwritten; click **Yes** to overwrite the old core.



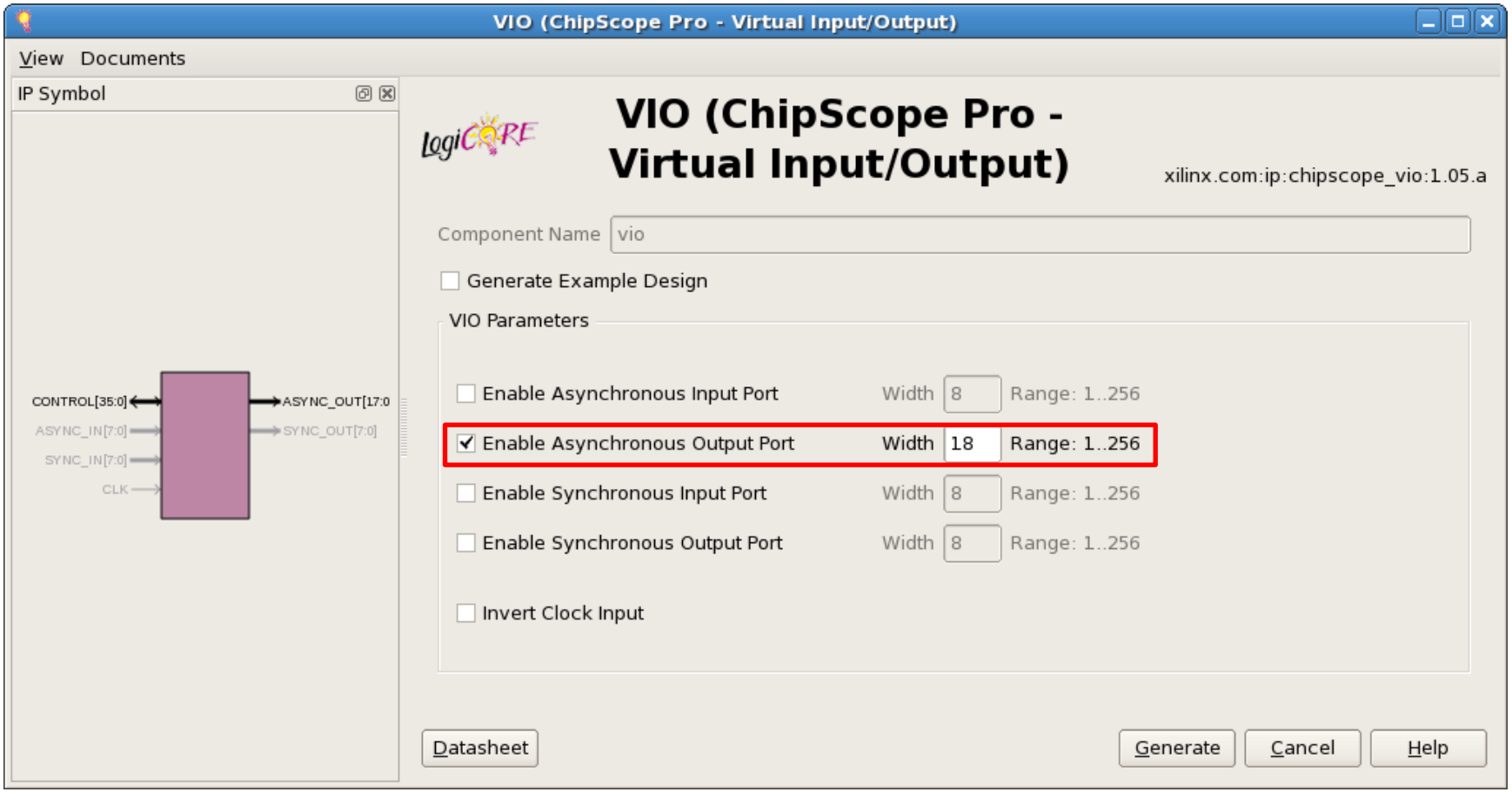
To generate the VIO core for the project, follow the same steps as were used for the ICON and ILA. Add a new source to the project, select the source type as **IP (Core Generator & Architecture Wizard)** and name the core appropriately.



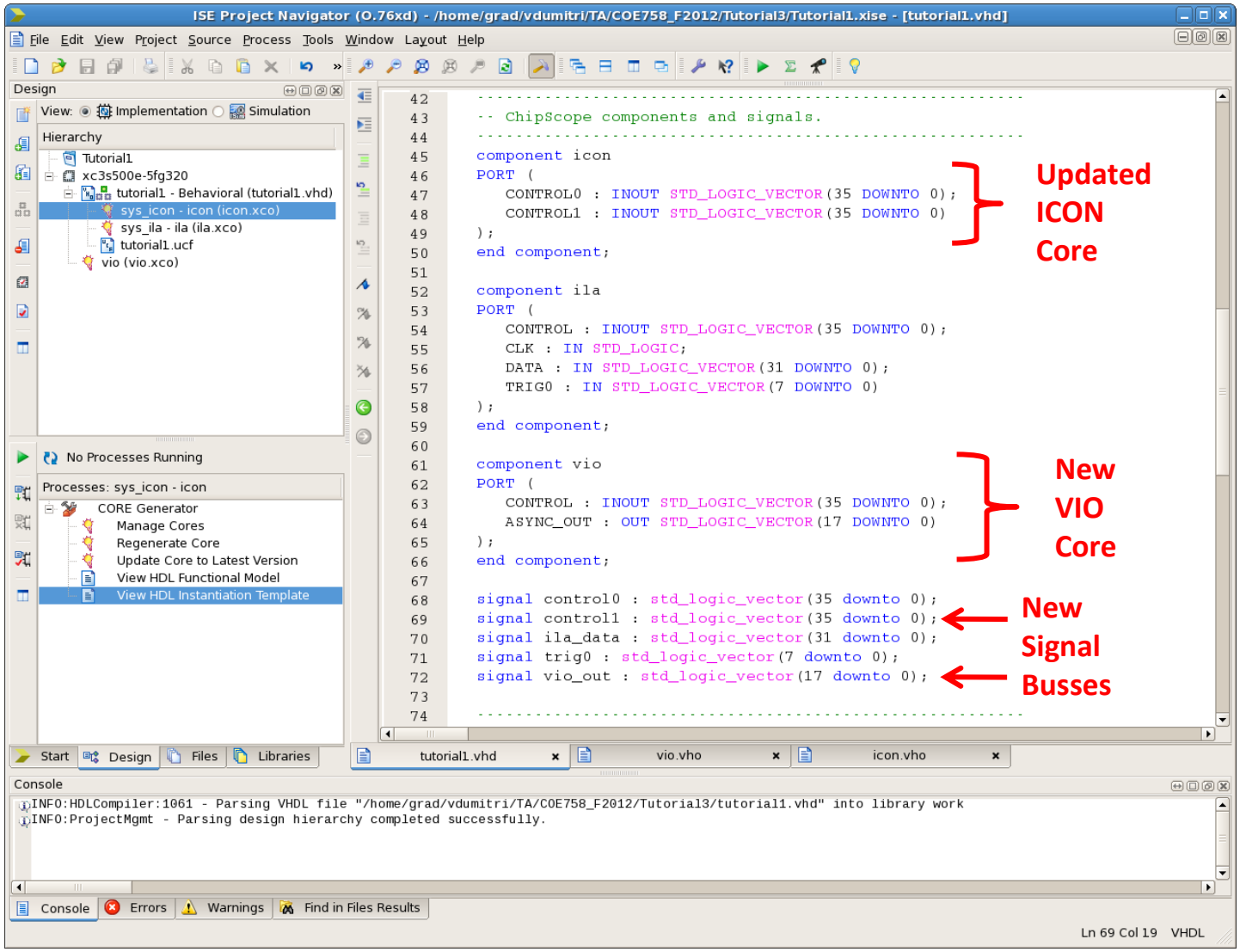
Select **VIO** as the desired IP core, once again ensuring that you selected the core with the status **Production**, and click **Next**.



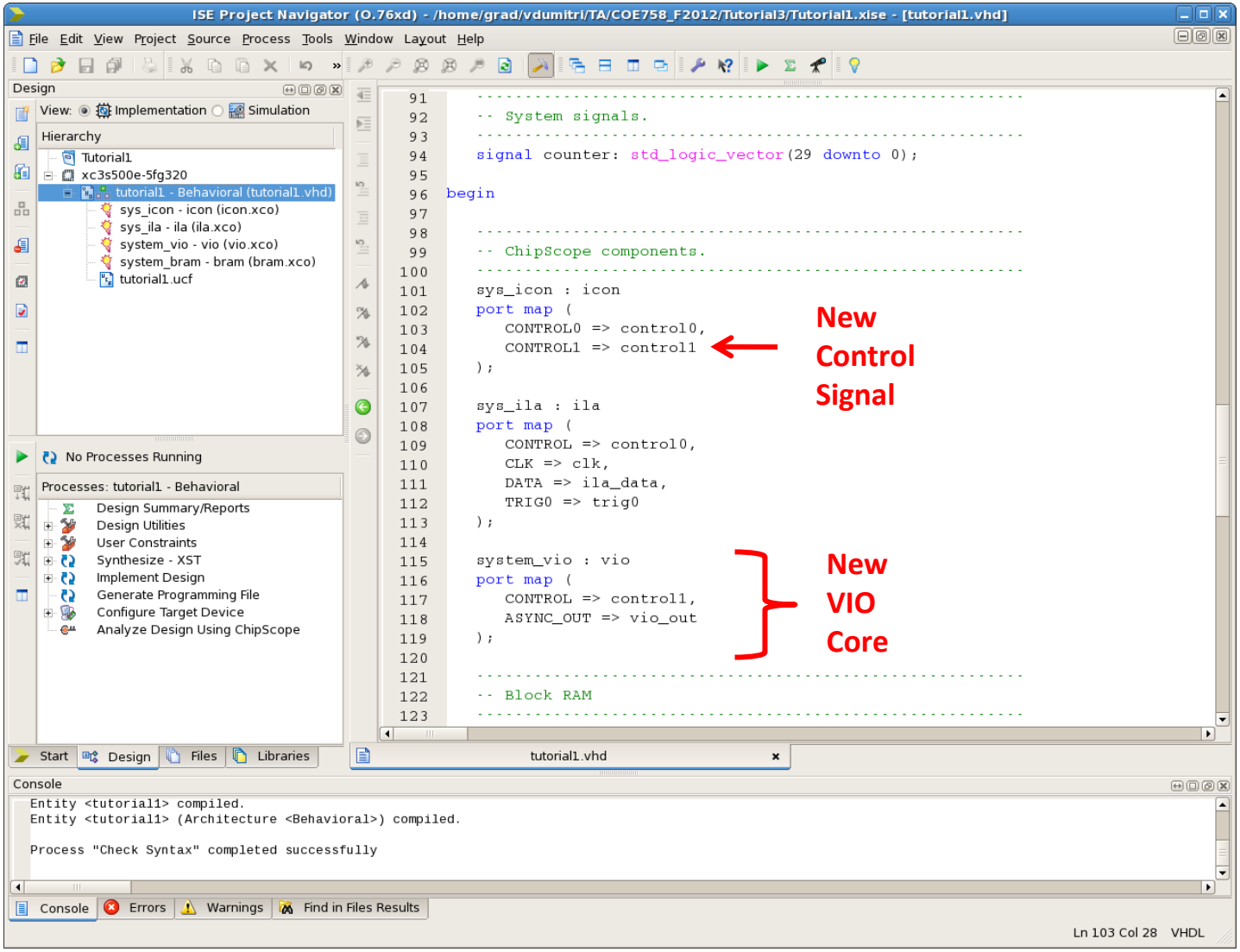
You will once again receive a summary regarding the core you are about to generate. Click **Finish** to proceed to the core generator.



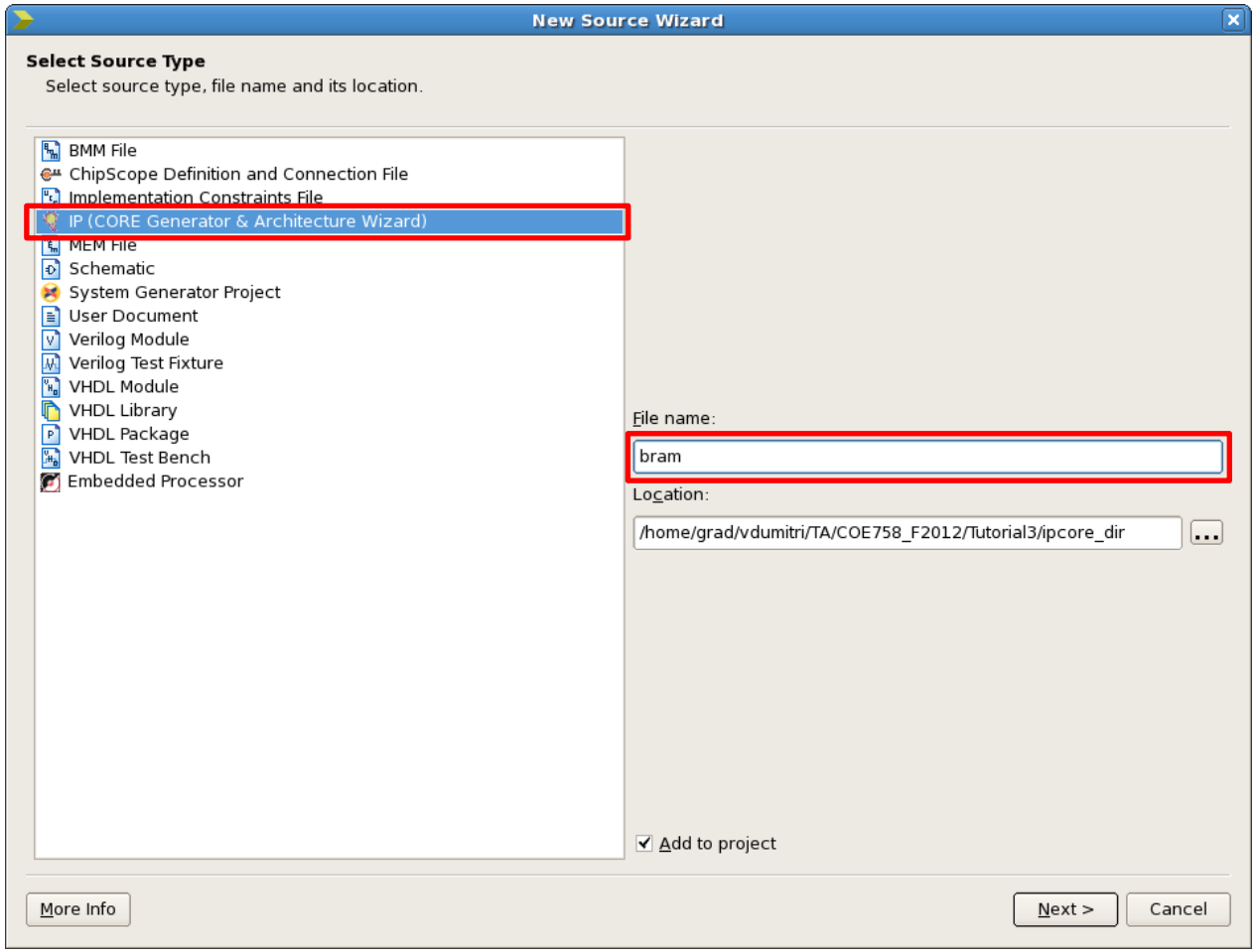
The VIO core has a limited range of parameters available. The user can select to use either asynchronous or synchronous input and/or output ports. For the current tutorial, enable only the option **Enable Asynchronous Output Port**, and set the **Width** to **18**. Finally, click **Generate** to generate the core.



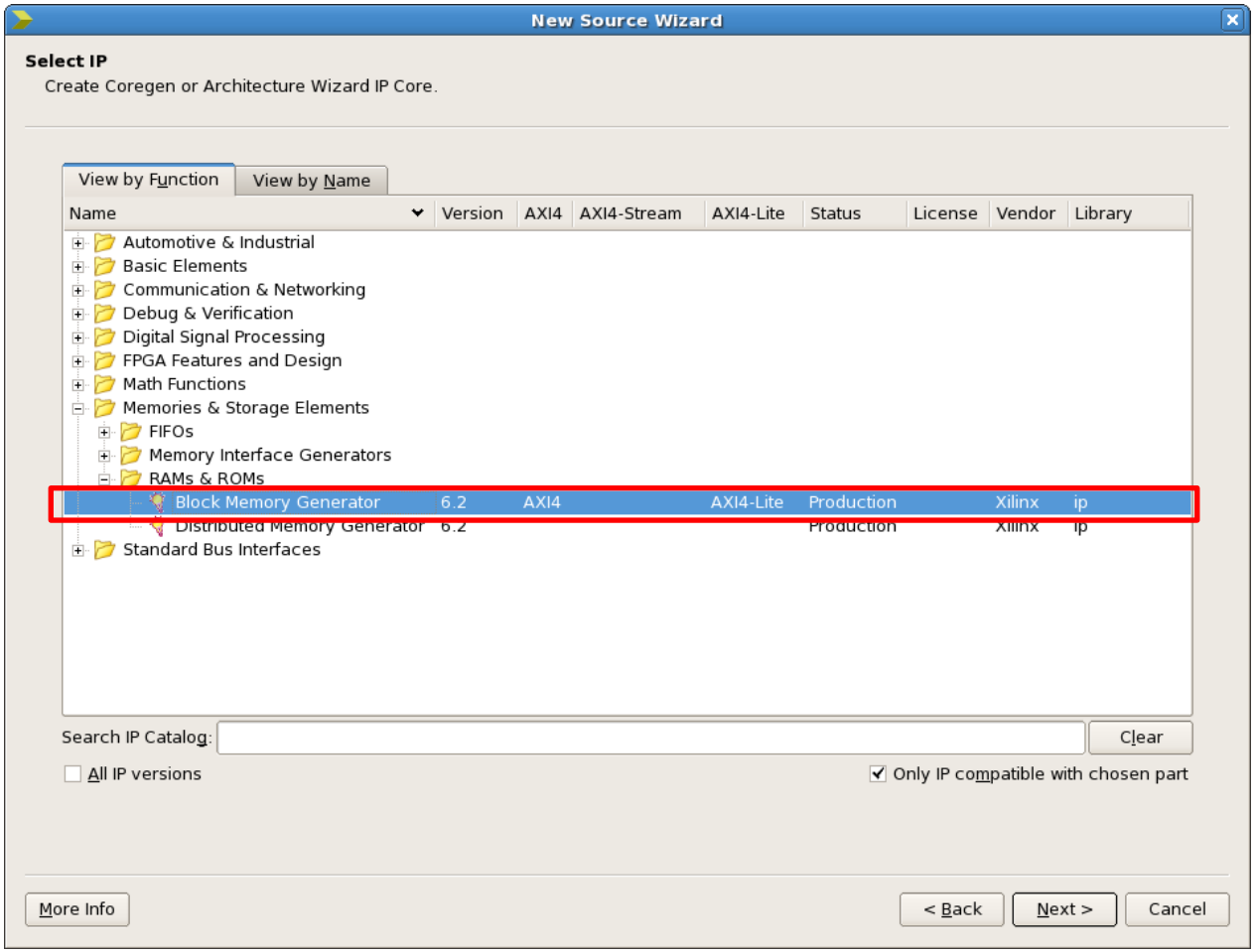
Once the VIO core is generated, use the HDL Instantiation Template to copy the declaration and instantiation statements into your design file. You must also repeat this process for the ICON core, given that is has changed since the last tutorial. You will also need additional signal busses: another control bus (used by the ICON and VIO) and a bus for the VIO output port. The final declarations for your VIO components should look similar to what is shown in the figure above.



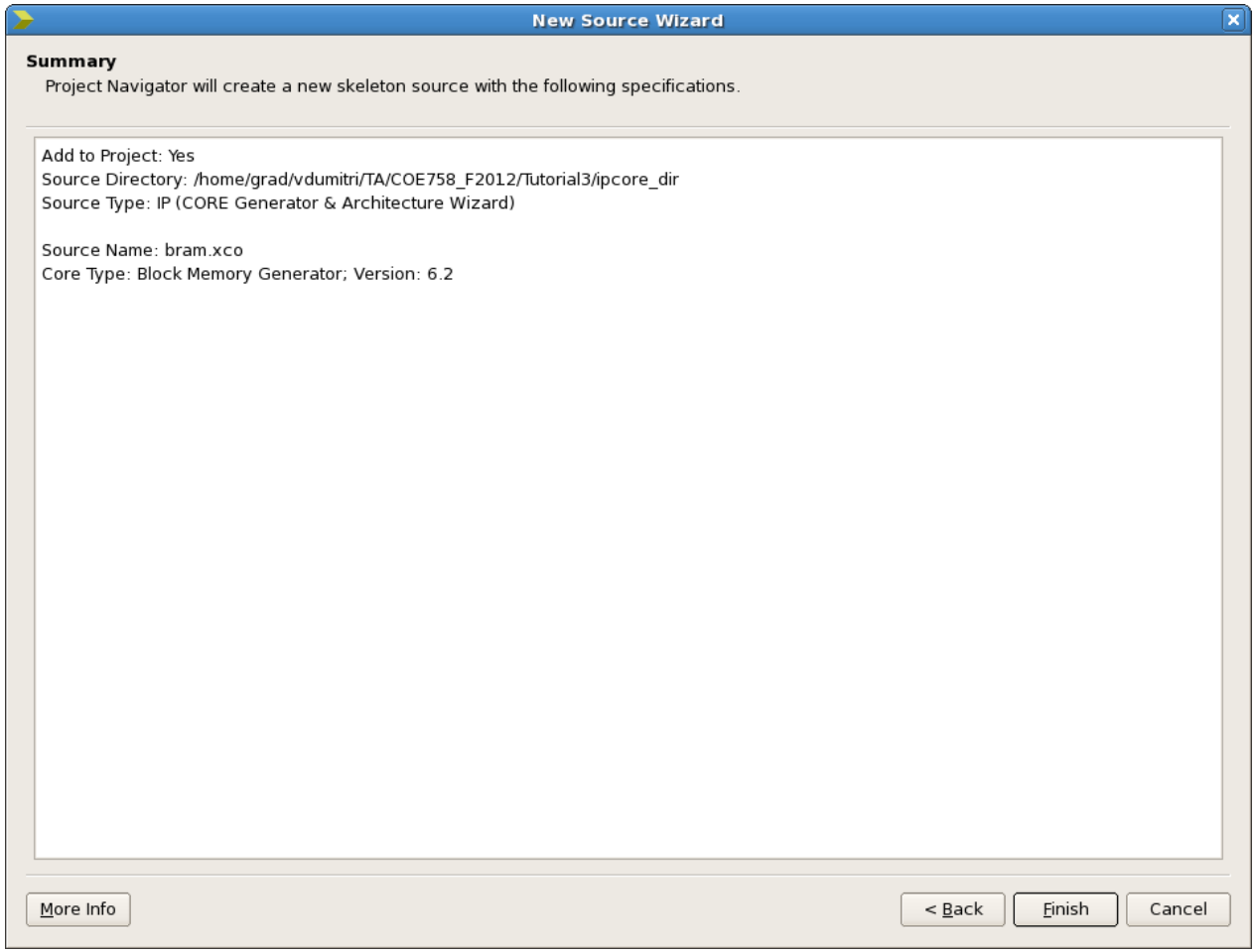
The new VIO core, as well as the updated ICON core must also be instantiated inside the design. The VIO core will be controlled using the second control bus, ***control1,*** which will also be connected to the second ICON control port. The final ChipScope instantiations should look similar to what is shown above.



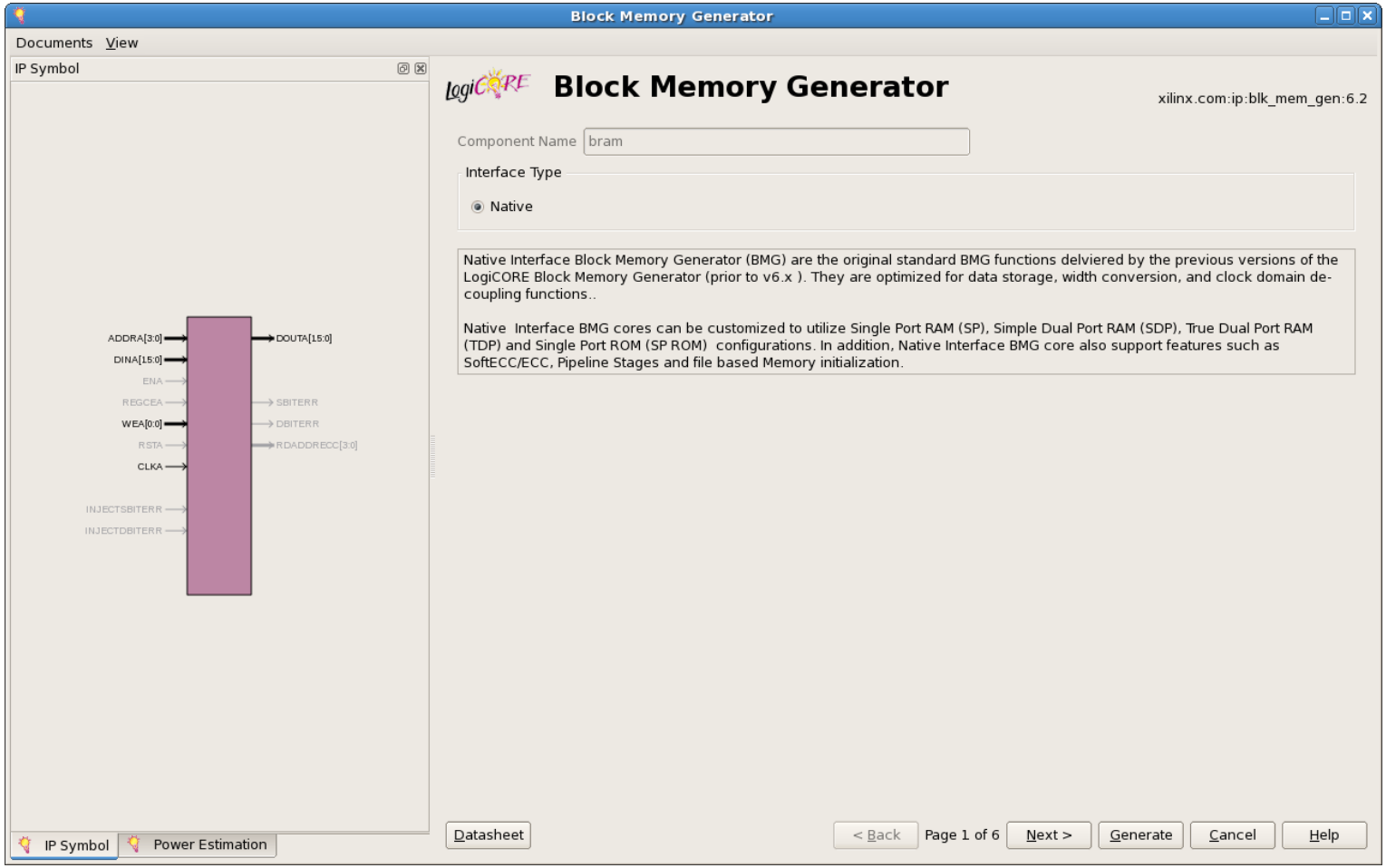
A block memory core is also used in this project, and must be generated and added to the project. Begin by adding a new source to the project. For the source type, once again select **IP (Core Generator & Architecture Wizard)**, and name the core. Finally, click **Next**.



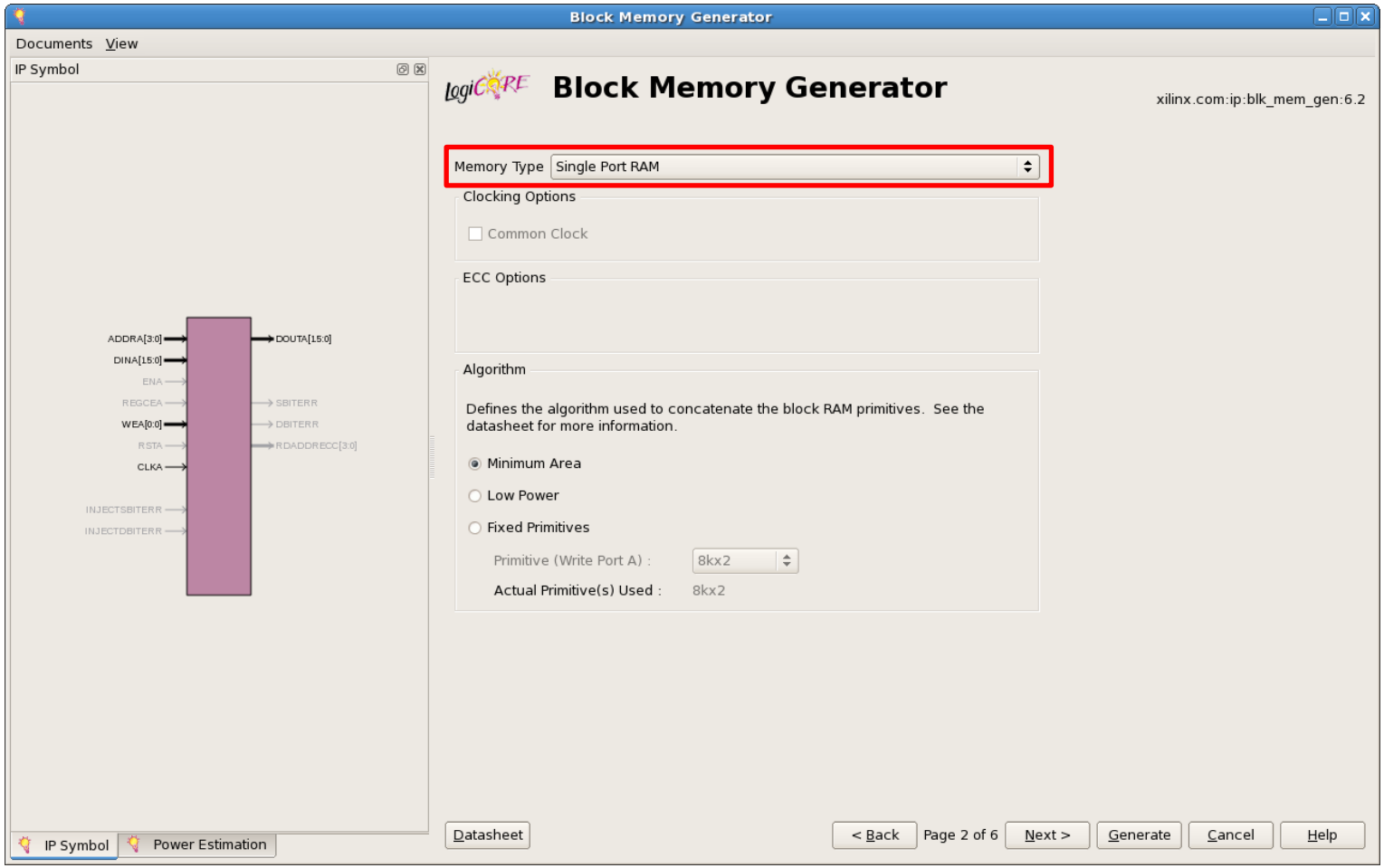
When selecting the core type, expand the **Memories & Storage Elements** folder, followed by the **RAMs & ROMs** folder, and select the **Block Memory Generator**. Once selected, click **Next**.



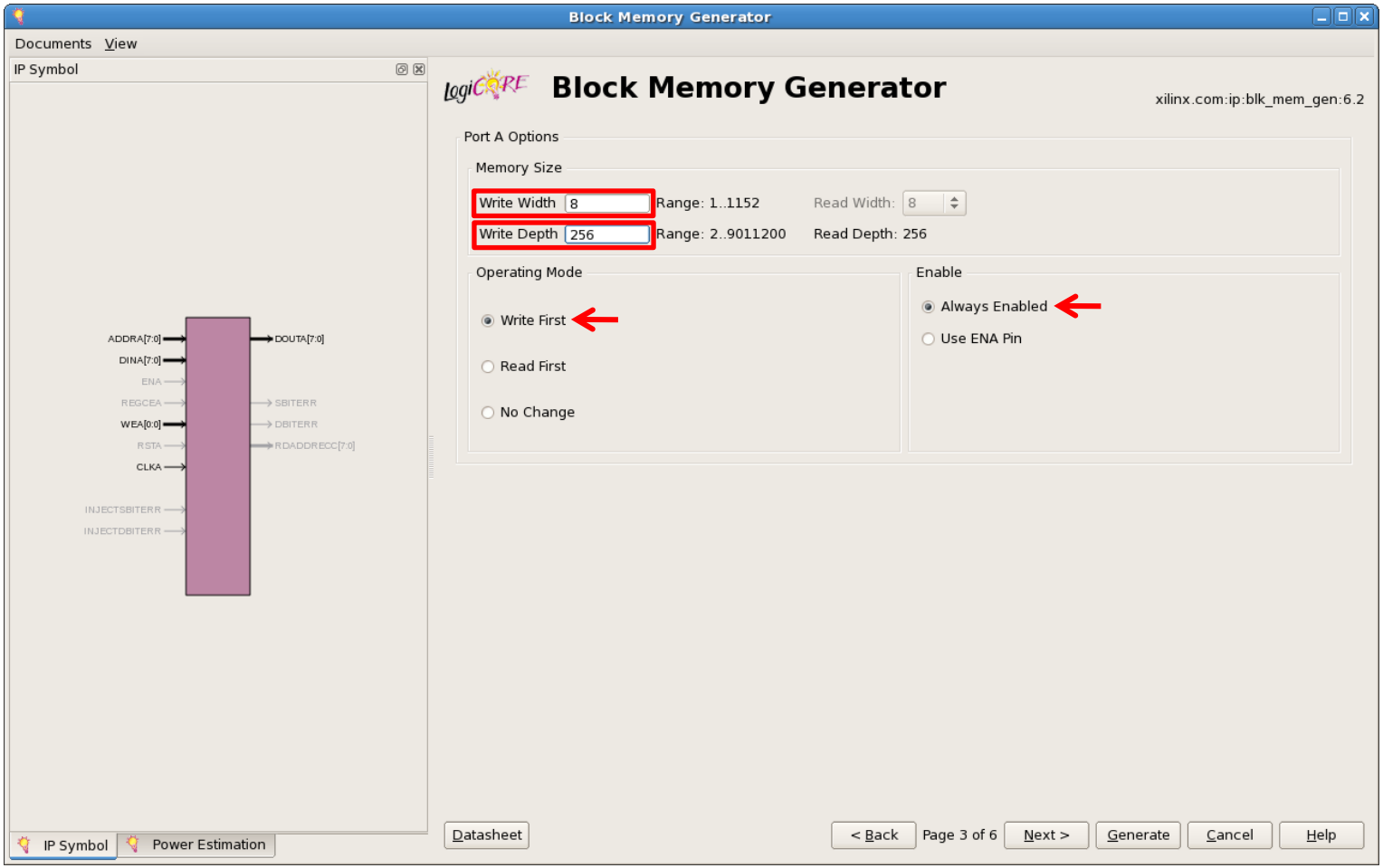
As before, you will be presented with a summary window for the core about to be created. Click **Finish** to launch the core generator.



On the first page of the core generator wizard, no changes are needed. The **Interface Type** will be left to **Native**; click **Next**.



On the second page, select the Memory Type to be Single Port RAM. Leave all other options as they are, and click Next.



Memory Port Settings:

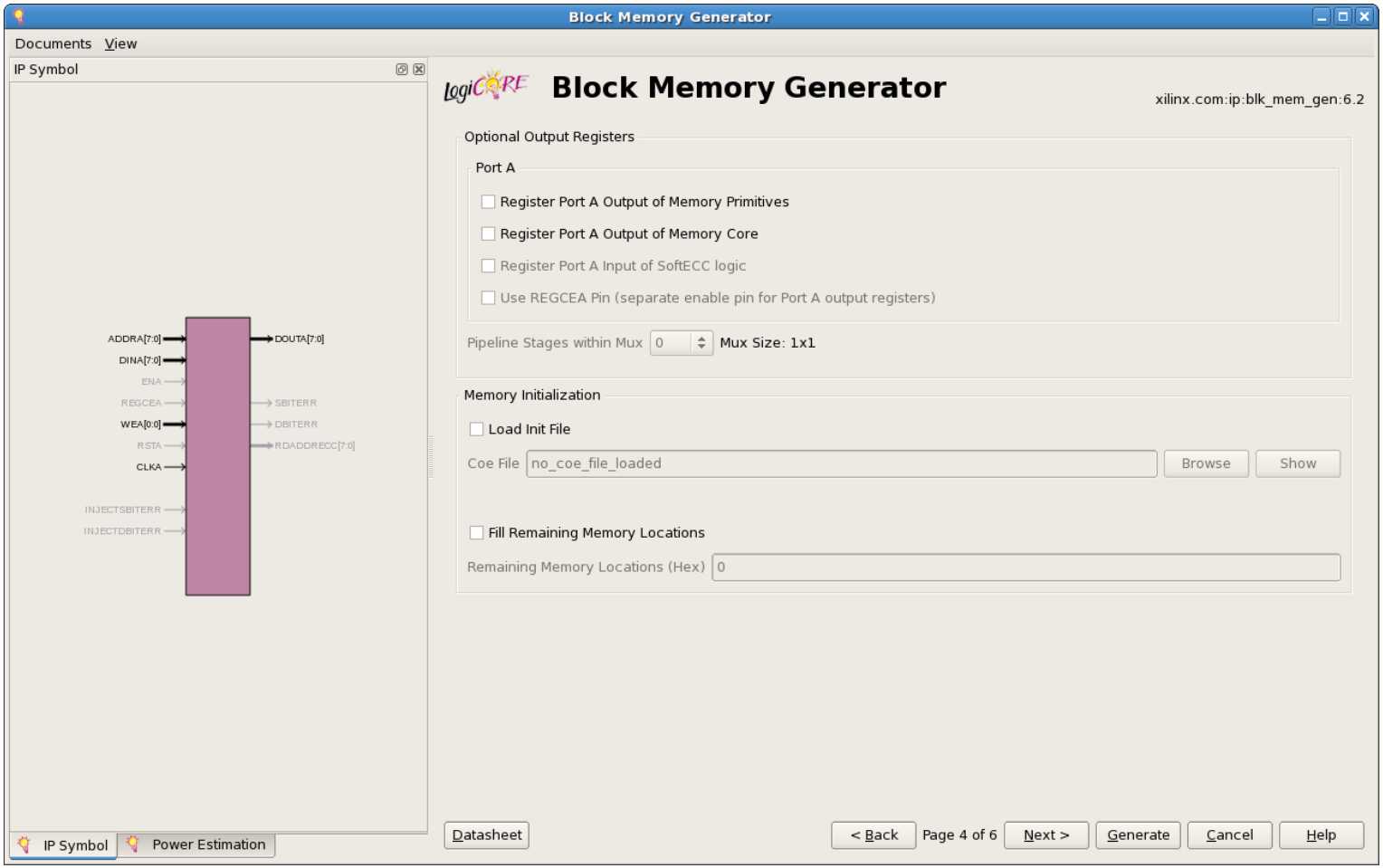
Write Width: ***8***

Write Depth: ***256***

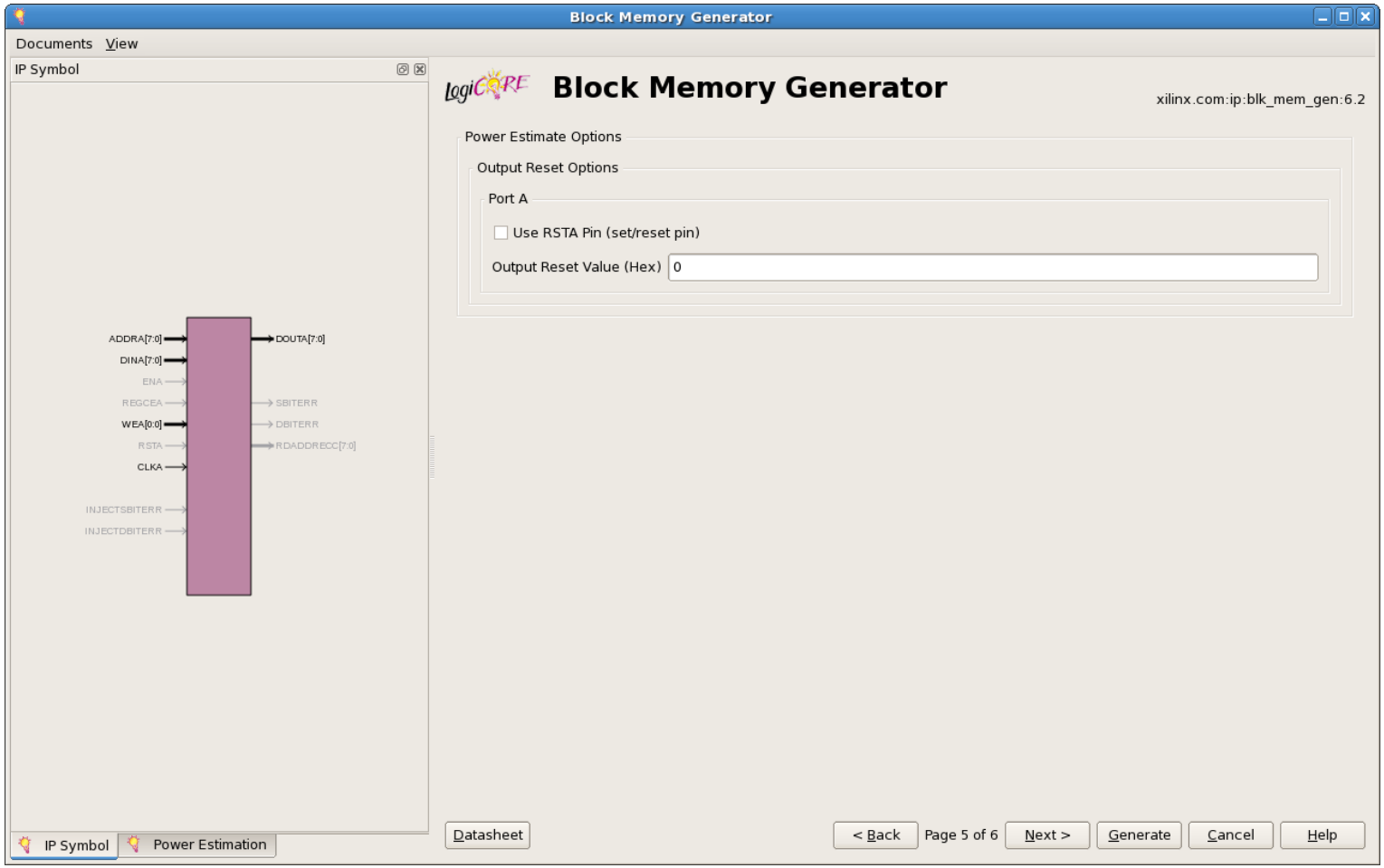
Operating Mode: ***Write first***

Enable: Always ***Enabled***

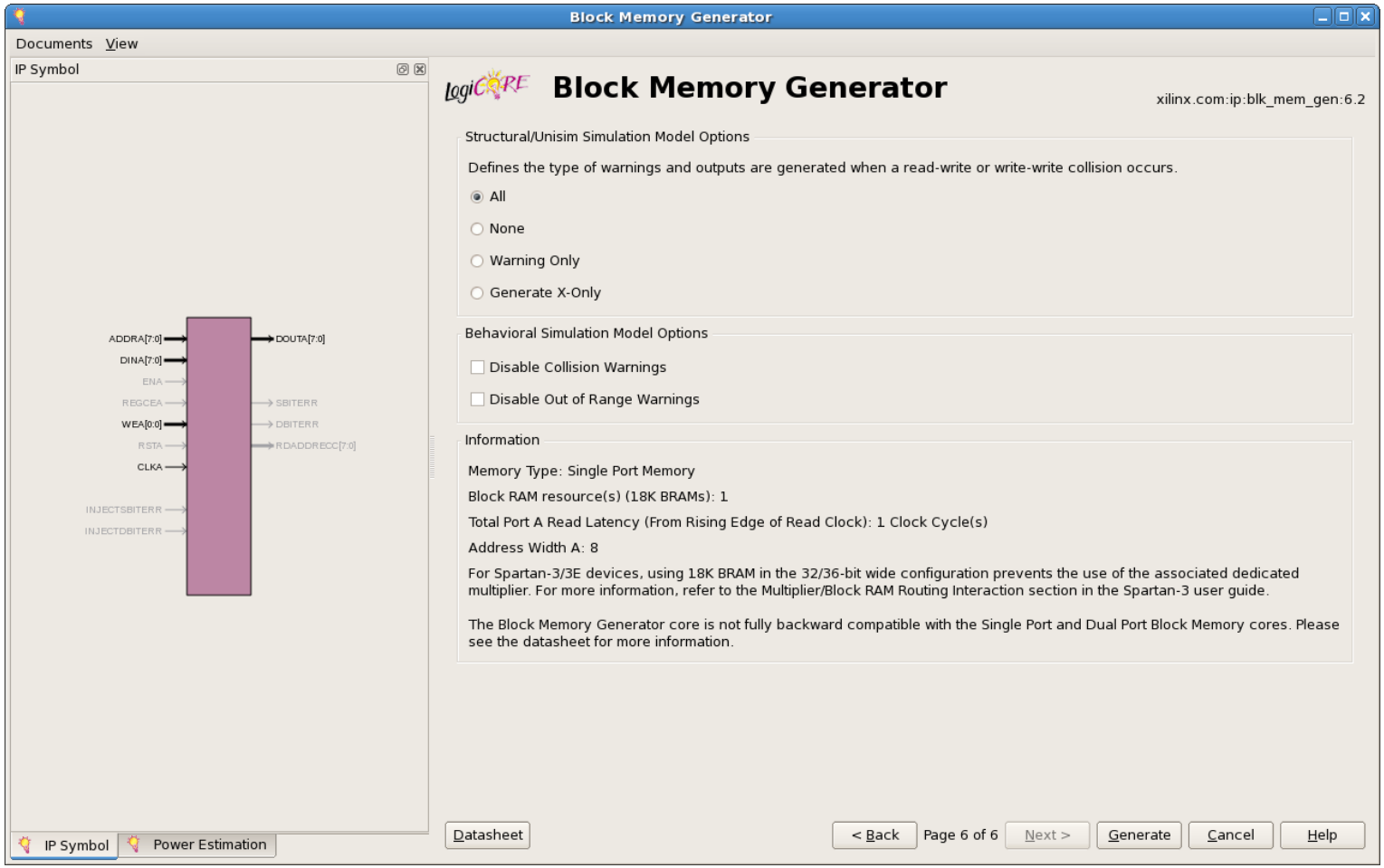
On the third page, you can select the characteristics of the memory port, and the amount of storage available. For the current tutorial, set the **Write Width** to **8**, **Write Depth** to **256**, enable **Write First** mode, and select the **Always Enabled** option. Finally, click **Next**.



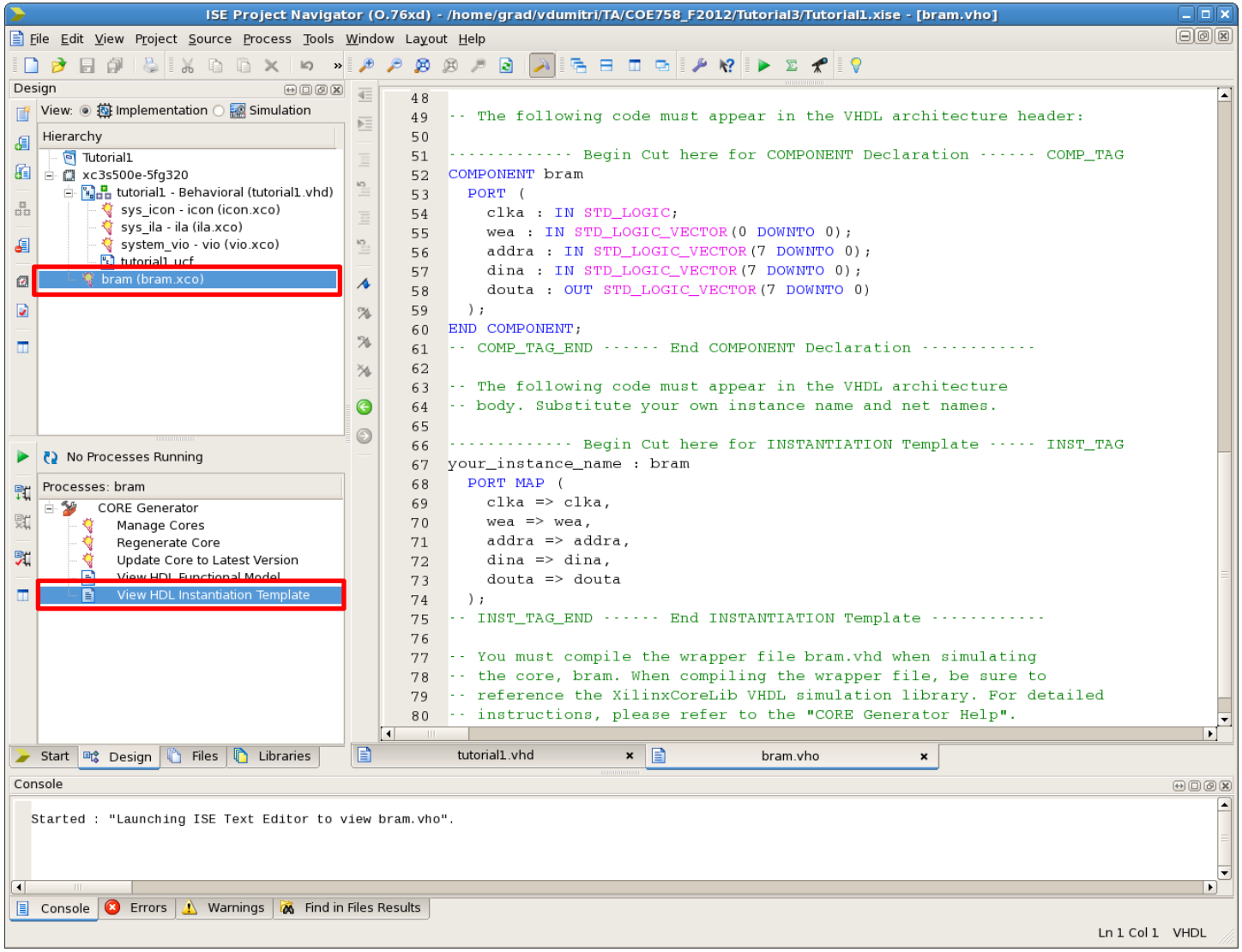
No changes are needed on the fourth page. Click **Next**.



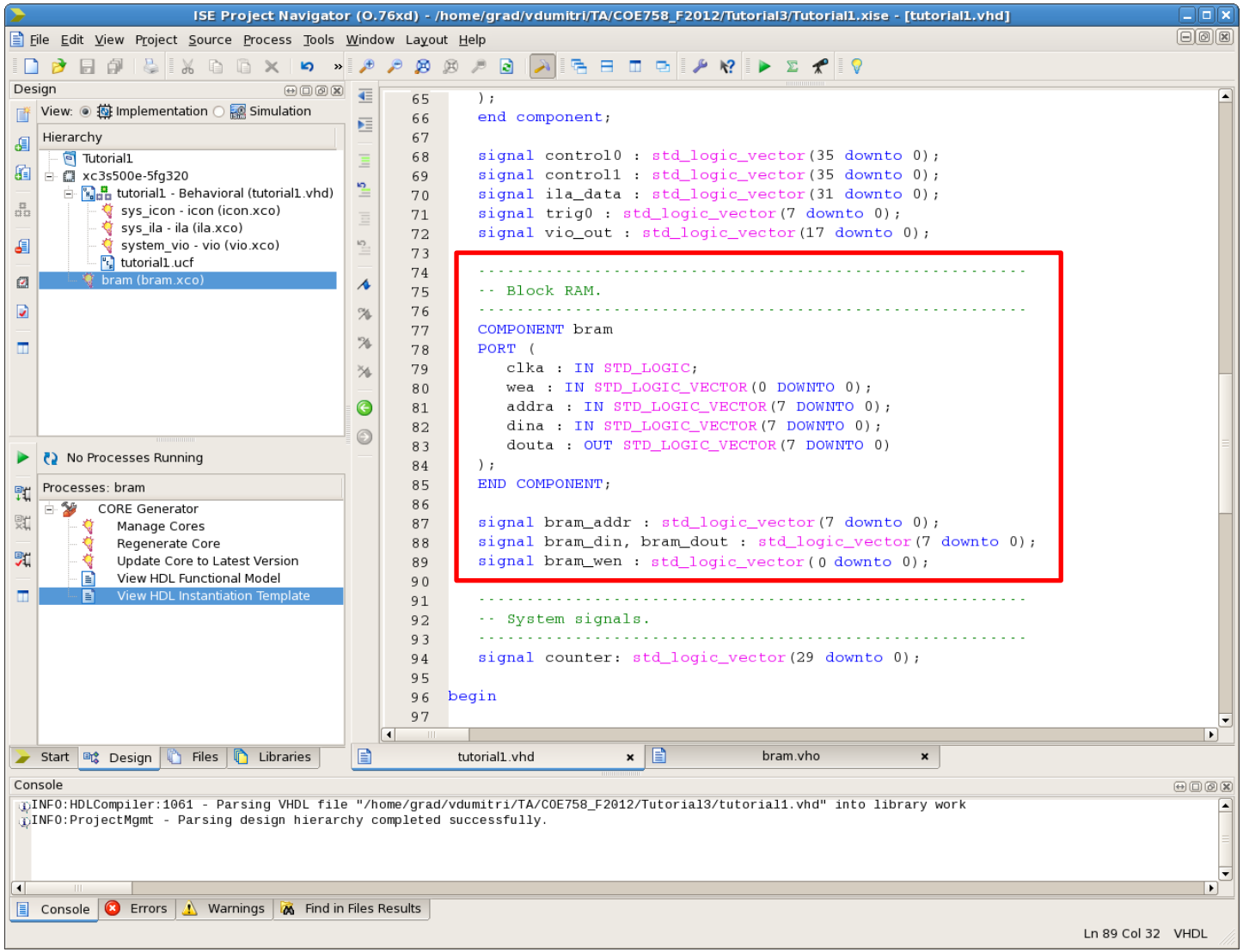
Likewise, no changes are needed on page 5. Click **Next**.



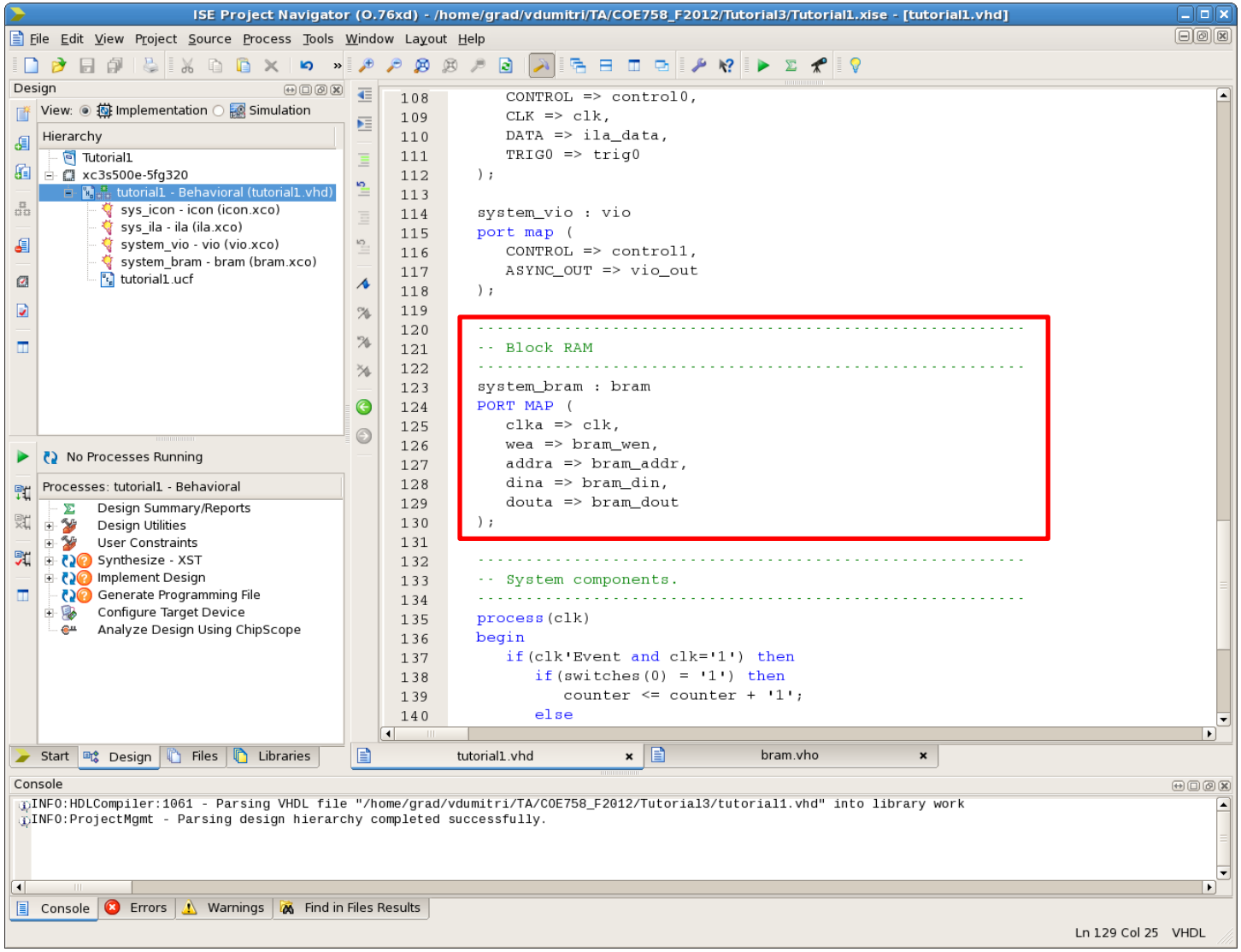
The final page specifies parameters for the simulation model for the memory core. Leave the page as is, and click **Generate**.



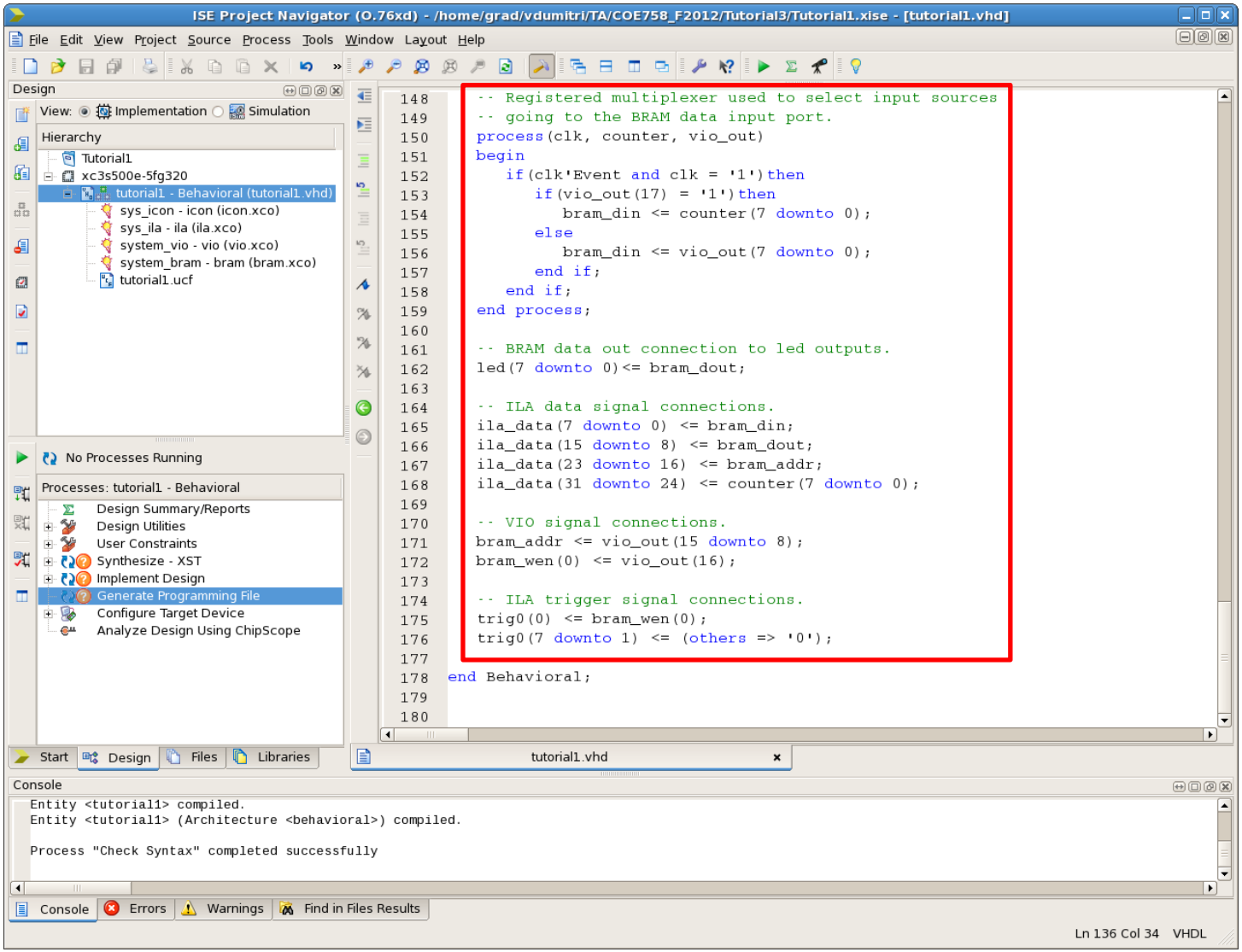
As with the ChipScope cores, use the **HDL Instantiation Template** to copy the BRAM declaration and instantiation statements into your design.



In addition to its declaration, the BRAM component also needs busses for its data input and output ports, as well as the address port. A write enable signal is also required.



Finally, the BRAM should be instantiated as shown in the figure above.

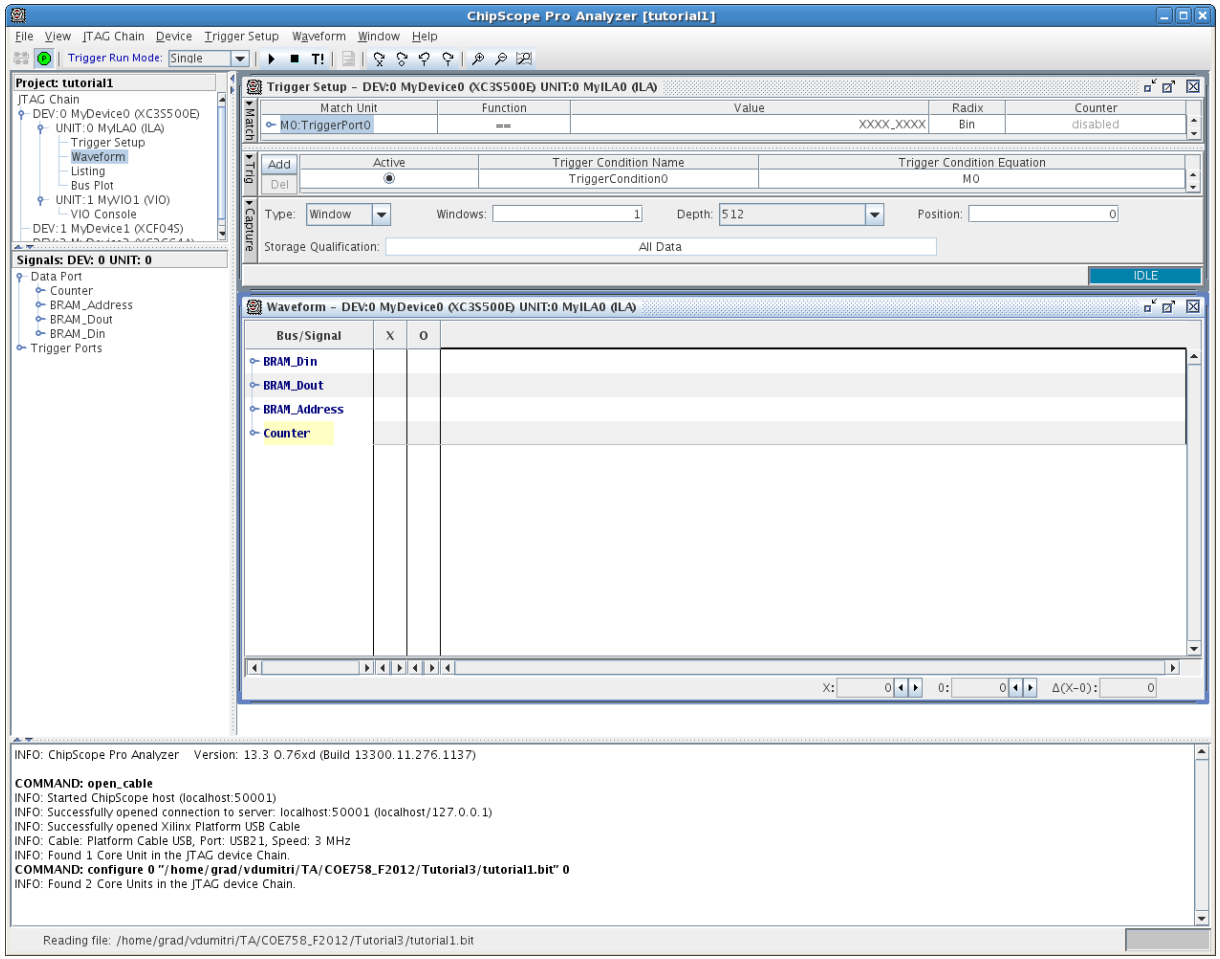


Finally, connect the BRAM input and output signals as shown above. Likewise, add a multiplexer which can select what data is sent to the BRAM data input.

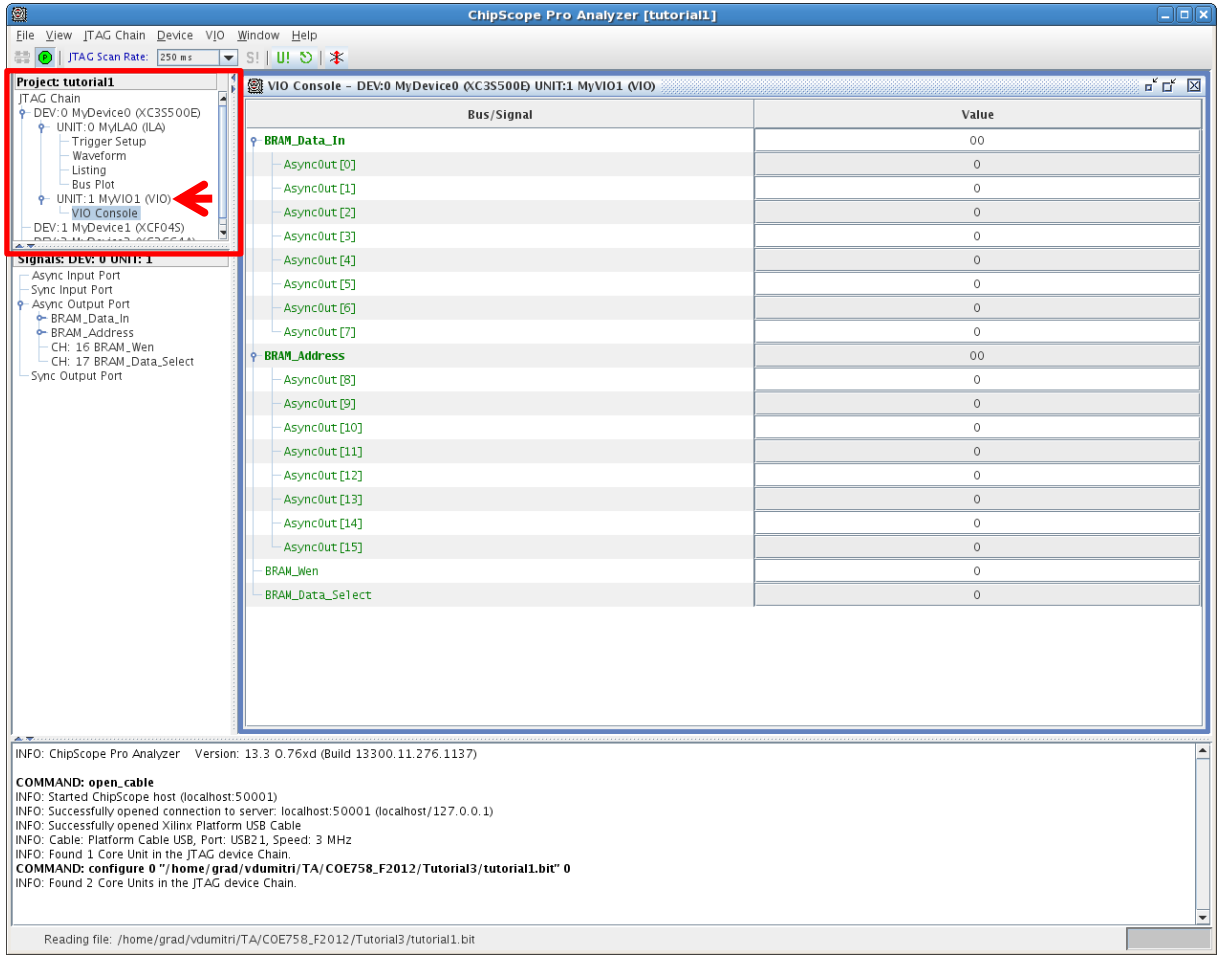
The complete VHDL code for this project is shown below and on the following page.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity tutorial1 is
   Port ( clk : in  STD_LOGIC;
          led : out  STD_LOGIC_VECTOR (7 downto 0);
          switches : in  STD_LOGIC_VECTOR (3 downto 0));
end tutorial1;

architecture Behavioral of tutorial1 is

          ------------------------------------------------------------
          -- ChipScope components and signals.
          ------------------------------------------------------------
          component icon
          PORT (
                    CONTROL0 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
                    CONTROL1 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0)
          );
          end component;

          component ila
          PORT (
                    CONTROL : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
                    CLK : IN STD_LOGIC;
                    DATA : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
                    TRIG0 : IN STD_LOGIC_VECTOR(7 DOWNTO 0)
          );
          end component;

          component vio
          PORT (
                    CONTROL : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
                    ASYNC_OUT : OUT STD_LOGIC_VECTOR(17 DOWNTO 0)
          );
          end component;

          signal control0 : std_logic_vector(35 downto 0);
          signal control1 : std_logic_vector(35 downto 0);
          signal ila_data : std_logic_vector(31 downto 0);
          signal trig0 : std_logic_vector(7 downto 0);
          signal vio_out : std_logic_vector(17 downto 0);

          ------------------------------------------------------------
          -- Block RAM.
          ------------------------------------------------------------
          COMPONENT bram
          PORT (
                    clka : IN STD_LOGIC;
                    wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
                    addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                    dina : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                    douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
          );
          END COMPONENT;

          signal bram_addr : std_logic_vector(7 downto 0);
          signal bram_din, bram_dout : std_logic_vector(7 downto 0);
          signal bram_wen : std_logic_vector(0 downto 0);

          ------------------------------------------------------------
          -- System signals.
          ------------------------------------------------------------
          signal counter: std_logic_vector(29 downto 0);

begin

          ------------------------------------------------------------
          -- ChipScope components.
          ------------------------------------------------------------
          sys_icon : icon
          port map (
                    CONTROL0 => control0,
                    CONTROL1 => control1
          );

          sys_ila : ila
          port map (
                    CONTROL => control0,
                    CLK => clk,
                    DATA => ila_data,
                    TRIG0 => trig0
          );
```

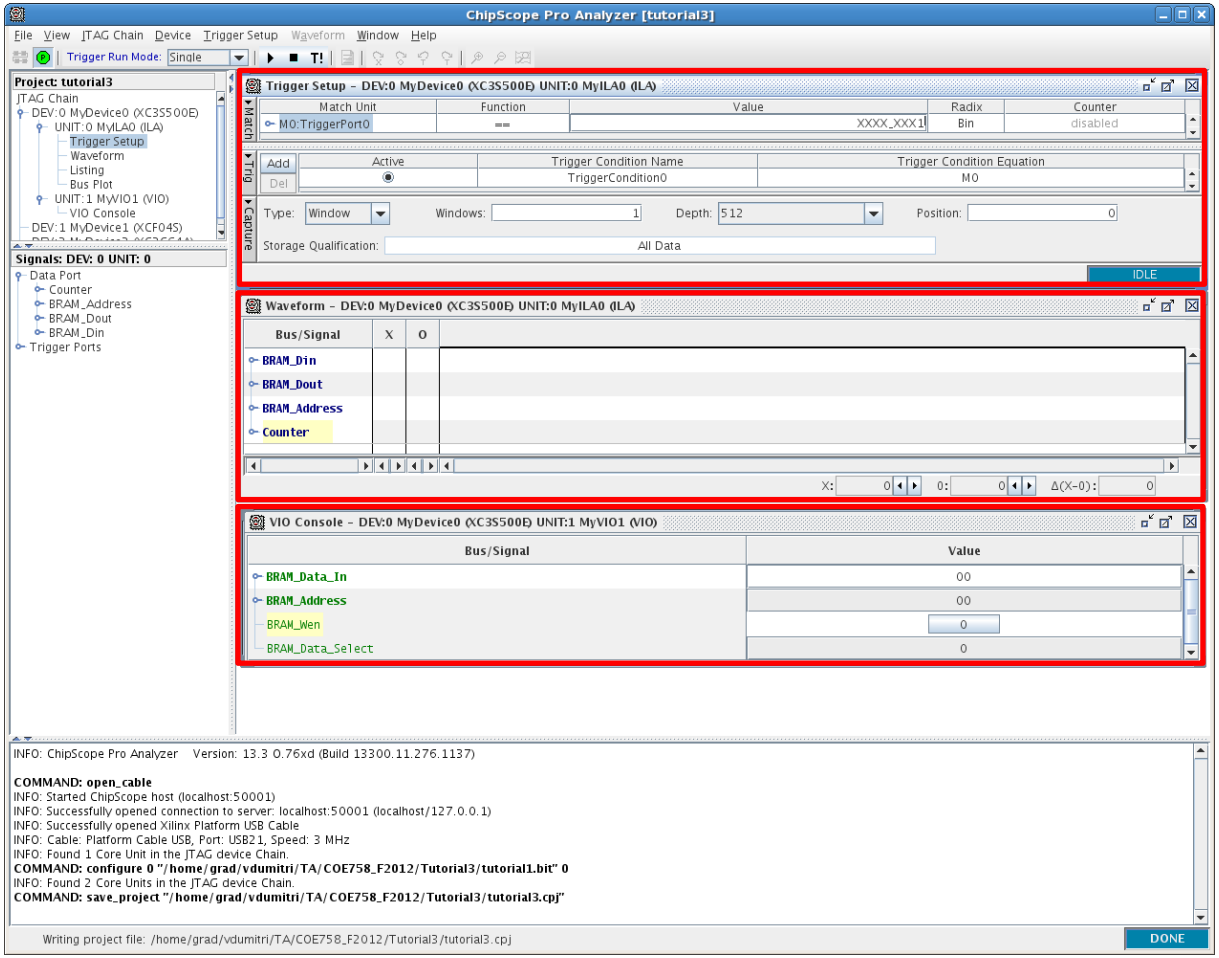
```
system_vio : vio
	port map (
		CONTROL => control1,
		ASYNC_OUT => vio_out
	);

	------------------------------------------------------------
	-- Block RAM
	------------------------------------------------------------
	system_bram : bram
	PORT MAP (
		clka => clk,
		wea => bram_wen,
		addra => bram_addr,
		dina => bram_din,
		douta => bram_dout
	);

	------------------------------------------------------------
	-- System components.
	------------------------------------------------------------

	-- Selectable up/down counter.
	process(clk)
	begin
		if(clk'Event and clk='1') then
			if(switches(0) = '1') then
				counter <= counter + '1';
			else
				counter <= counter - '1';
			end if;
		end if;
	end process;

	-- Registered multiplexer used to select input sources
	-- going to the BRAM data input port.
	process(clk, counter, vio_out)
	begin
		if(clk'Event and clk = '1')then
			if(vio_out(17) = '1')then
				bram_din <= counter(7 downto 0);
			else
				bram_din <= vio_out(7 downto 0);
			end if;
		end if;
	end process;

	-- BRAM data out connection to led outputs.
	led(7 downto 0)<= bram_dout;

		-- ILA data signal connections.
		ila_data(7 downto 0) <= bram_din;
		ila_data(15 downto 8) <= bram_dout;
		ila_data(23 downto 16) <= bram_addr;
		ila_data(31 downto 24) <= counter(7 downto 0);

		-- VIO signal connections.
		bram_addr <= vio_out(15 downto 8);
		bram_wen(0) <= vio_out(16);

		-- ILA trigger signal connections.
		trig0(0) <= bram_wen(0);
		trig0(7 downto 1) <= (others => '0');

end Behavioral;
```



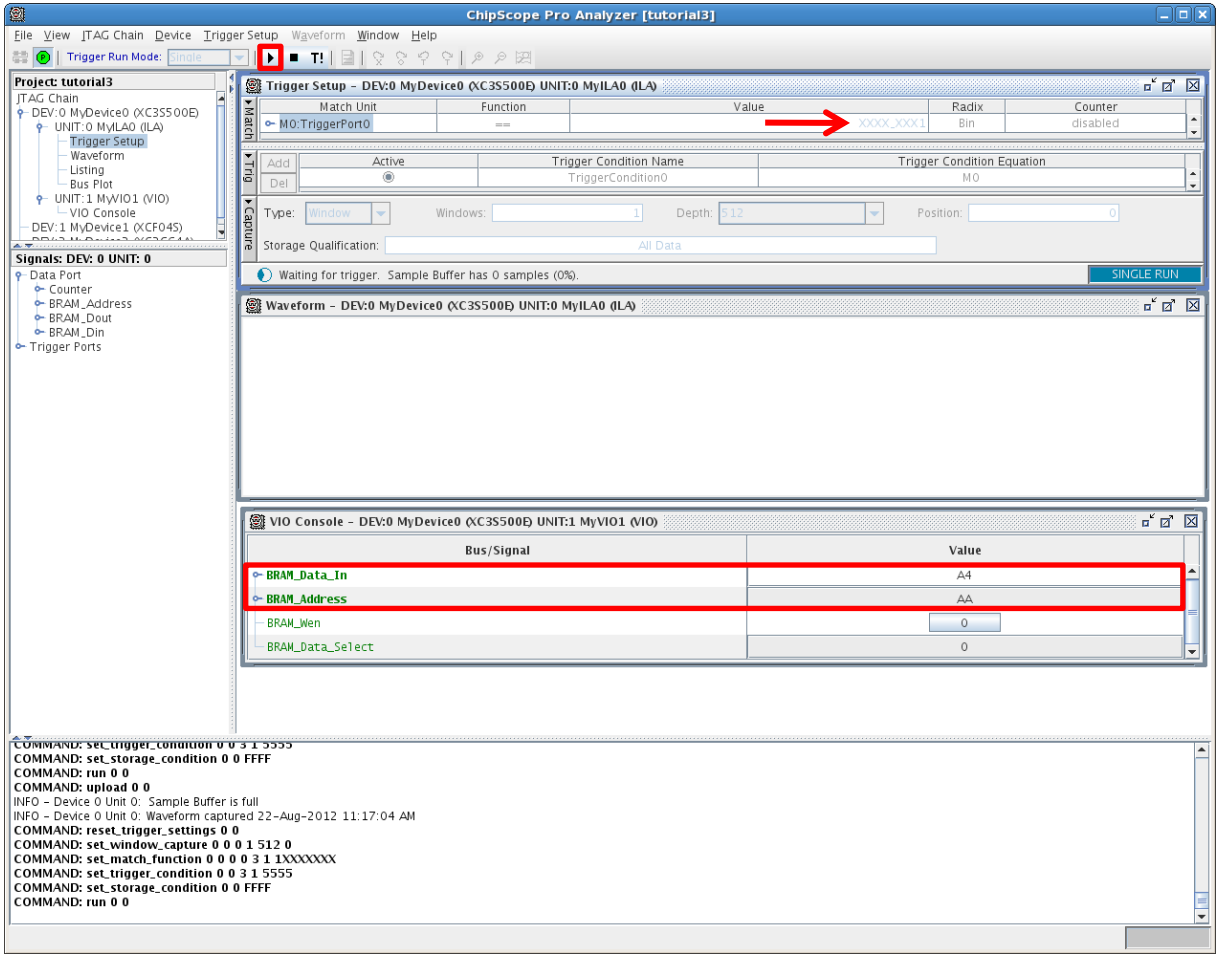
Once all code has been written, generate the configuration file, and launch the ChipScope Analyzer. Configure the FPGA with the bit-stream file generated for this tutorial (as was shown in Tutorial 2). Finally, in the **Waveform** window group the individual signals of the data port into busses that reflect the busses in the design. An example of this grouping is shown in the figure above.



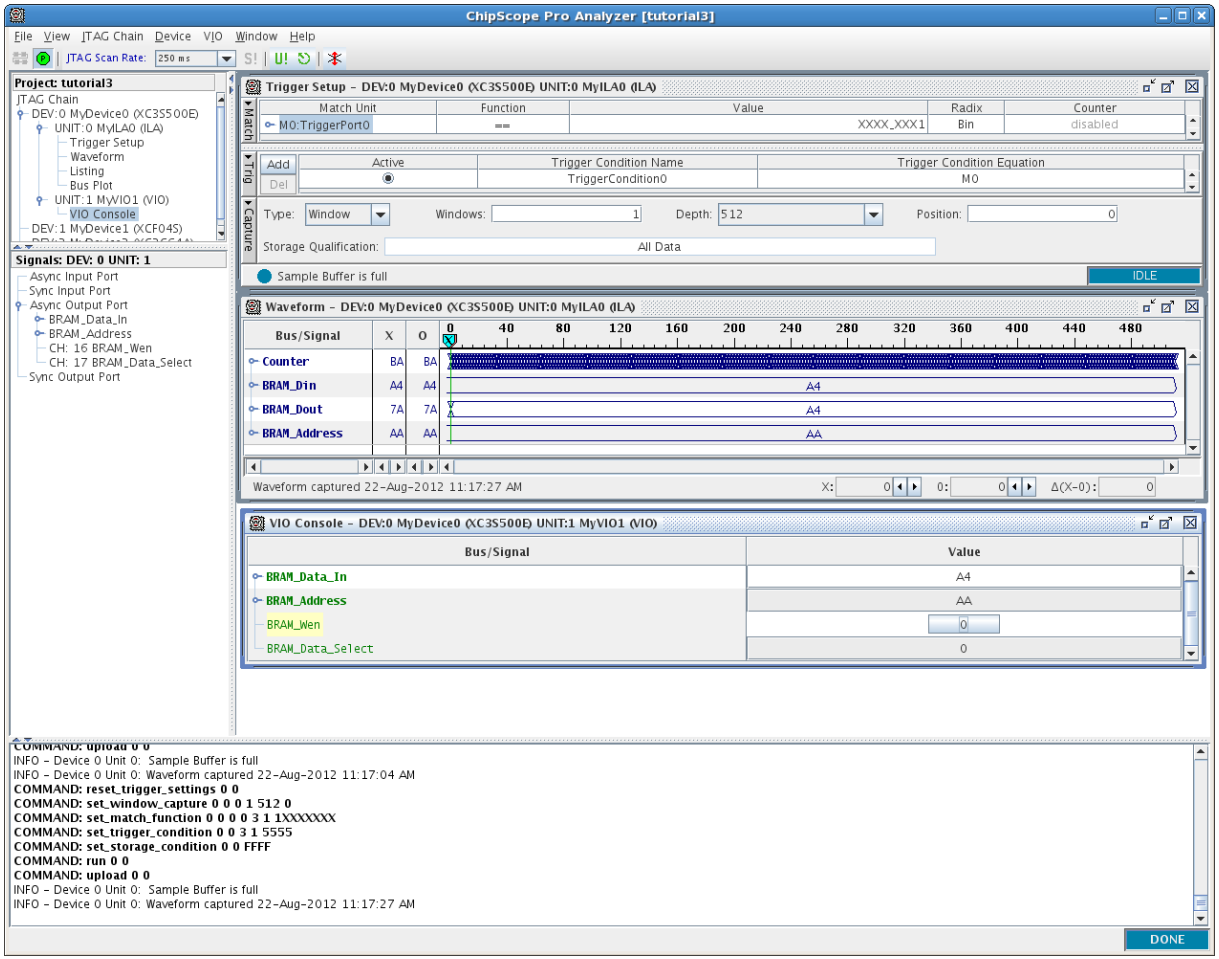
Because a VIO core was included into the design, a VIO console is also available. Expand the **Unit 1: MyVIO 1 (VIO)** entry in the top-left pane, and double-click on the **VIO Console**. The VIO port signals can also be grouped into busses, and renamed to reflect your design. Finally, VIO output signal 16 is used as a write-enable for the memory. To facilitate this mode of operation, this button should be changed to a push-button by right-clicking it and selecting **Type ➔ Push Button ➔ High**.



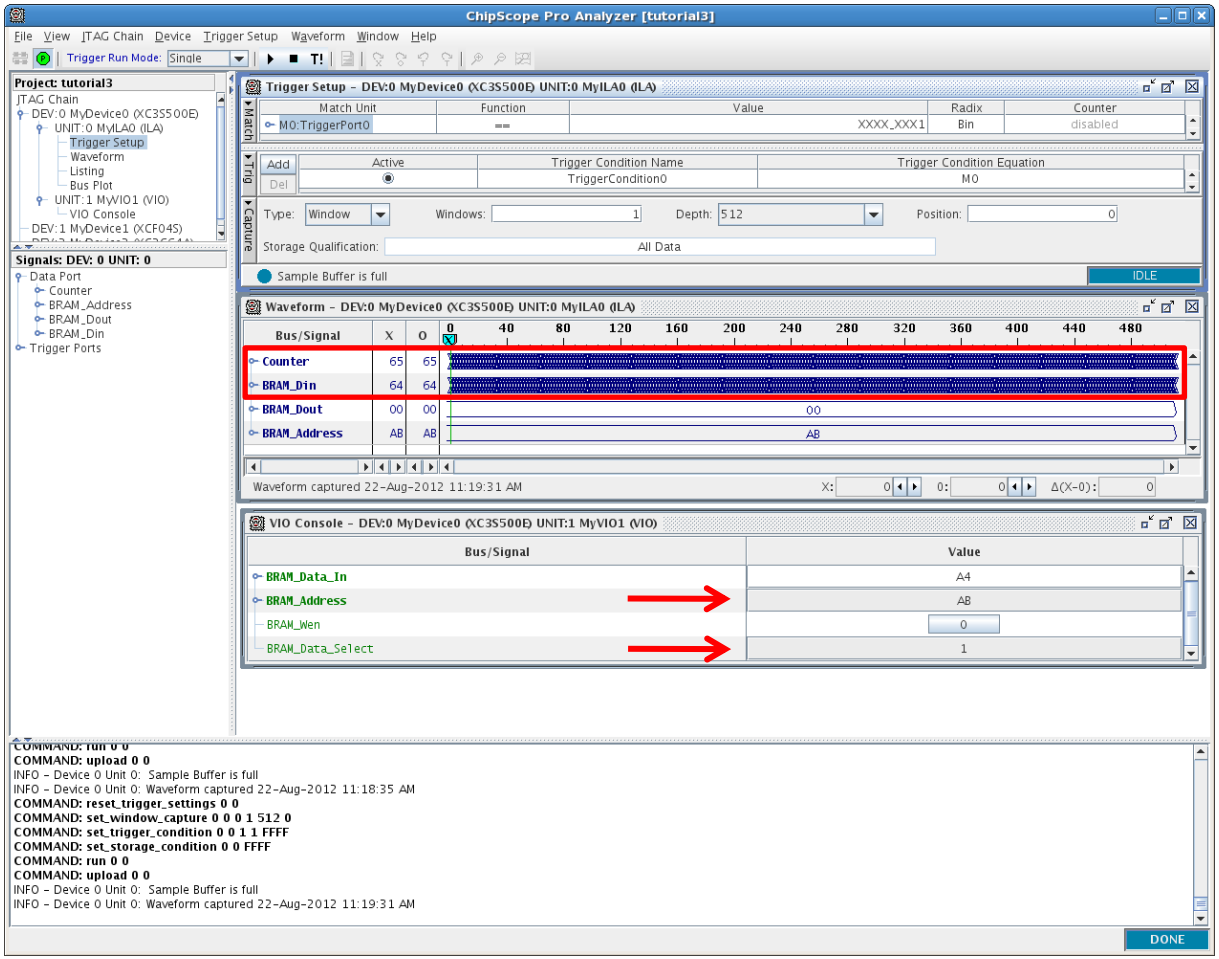
The final setup for this project should look similar to what is shown above.



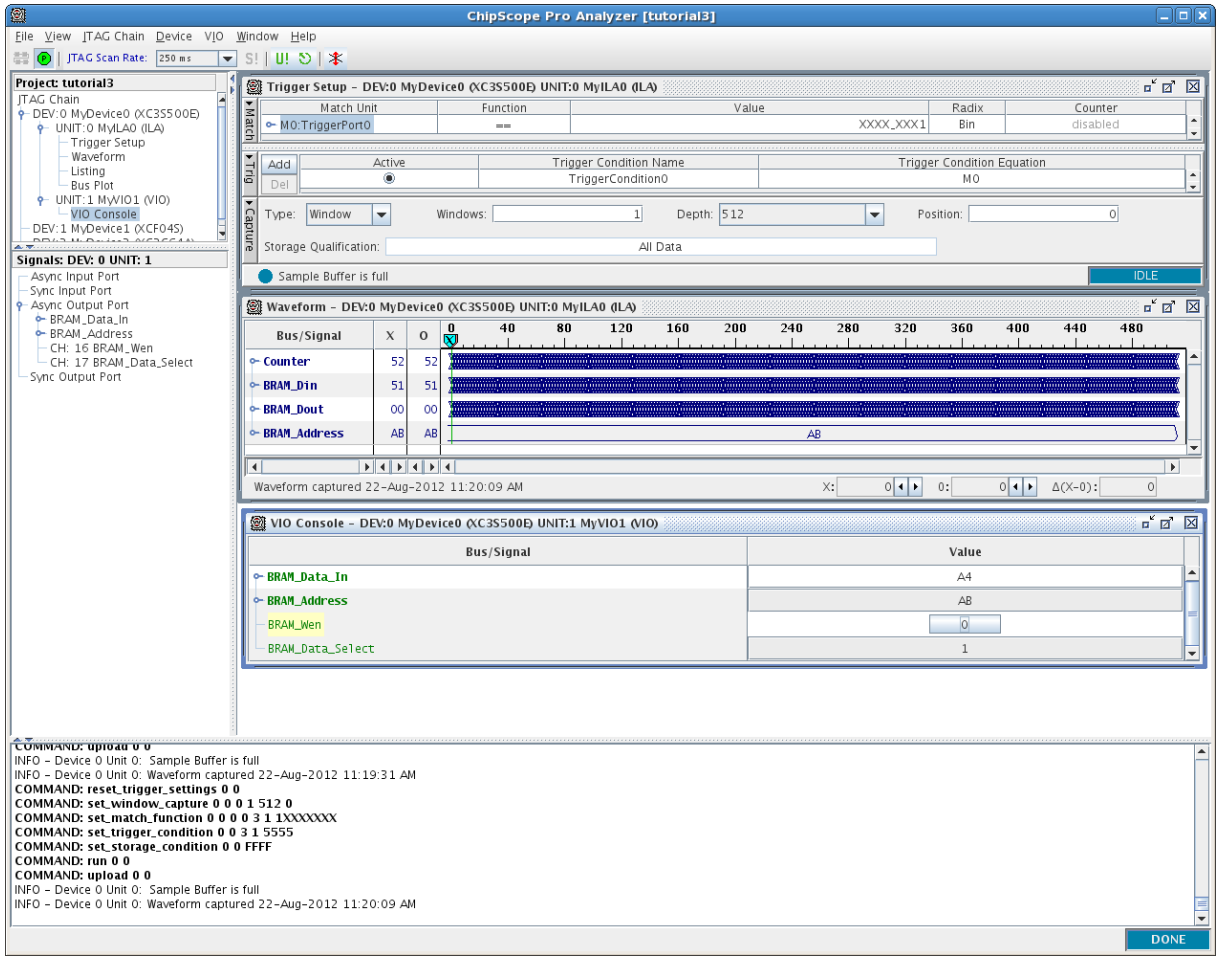
The memory can now be tested, by writing a specific value to one of the memory locations. Set the BRAM Data In to a specific value, and choose a specific address (above, A4 data is written to address AA) in the VIO Console. Change the trigger settings so that capture occurs when the write enable signal is set to 1 and arm the ILA core (press the play button).



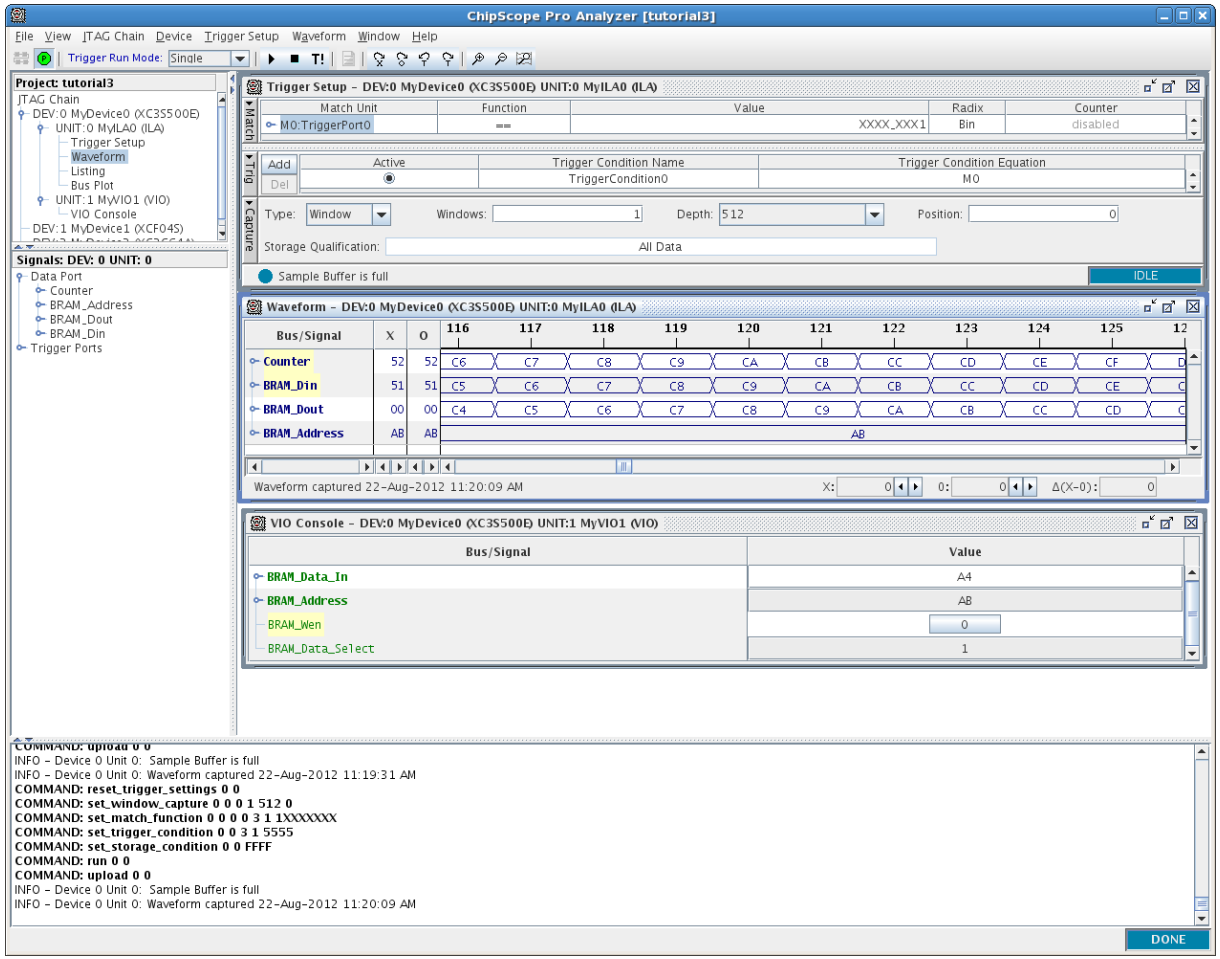
Finally, press and release the BRAM write enable button. This will enable writing in the BRAM core, at the specified address. This is reflected in the Waveform window, where the data input and data output show the same value at the specified address.



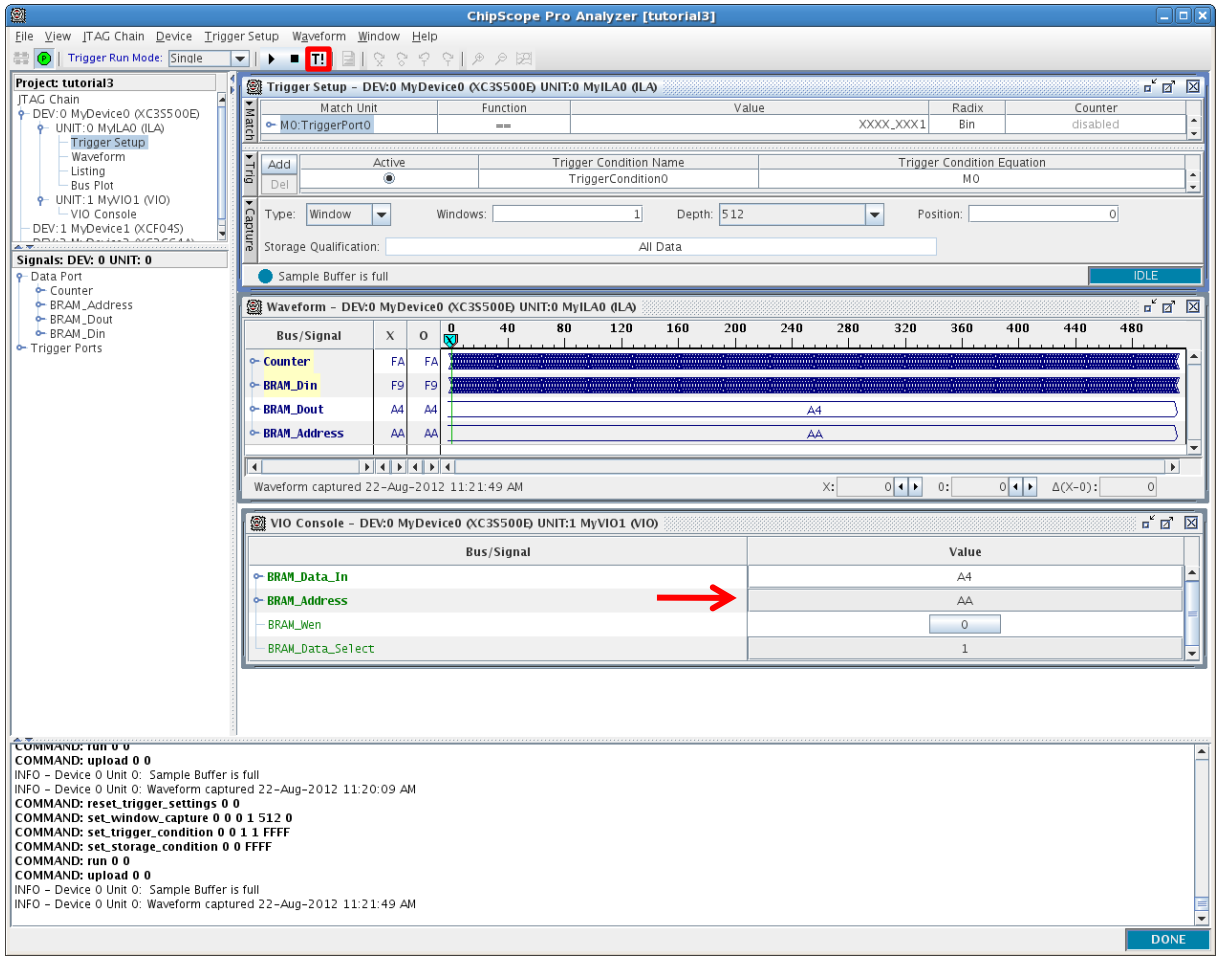
The next step is to write counter values to the BRAM. To do this, the first step is to change the multiplexer setting so that the counter values are sent to BRAM data input. If you now trigger the ILA core, you will see that counter data is now reflected in the BRAM data input. Likewise, the address should be changed, so that previous data is not overwritten.



You can now arm the core once again, and then press and release the write enable button. You will see that input data now cascades to the BRAM output.



You can zoom in on the Waveform window, to see the relationship between input data and output data. Specifically, the latency of data propagating from input to output.



You can now ensure that the data written previously is still stored correctly. Change the BRAM address to the previous value (in this case, AA), and trigger the ILA by pressing the **T!** Button. As can be seen, the original value is still present (A4).

Conclusion

- This concludes Tutorial 3. The following topics were covered:
  - Overview of the VIO core.
  - Generation and implementation of the VIO core into an ISE project.
  - Generation and implementation of a single-ported Block RAM into an ISE project.
  - Using VIO controls to interact with an on-chip design.
  - Reading and writing to the BRAM core.