



Project Navigator

 XILINX®

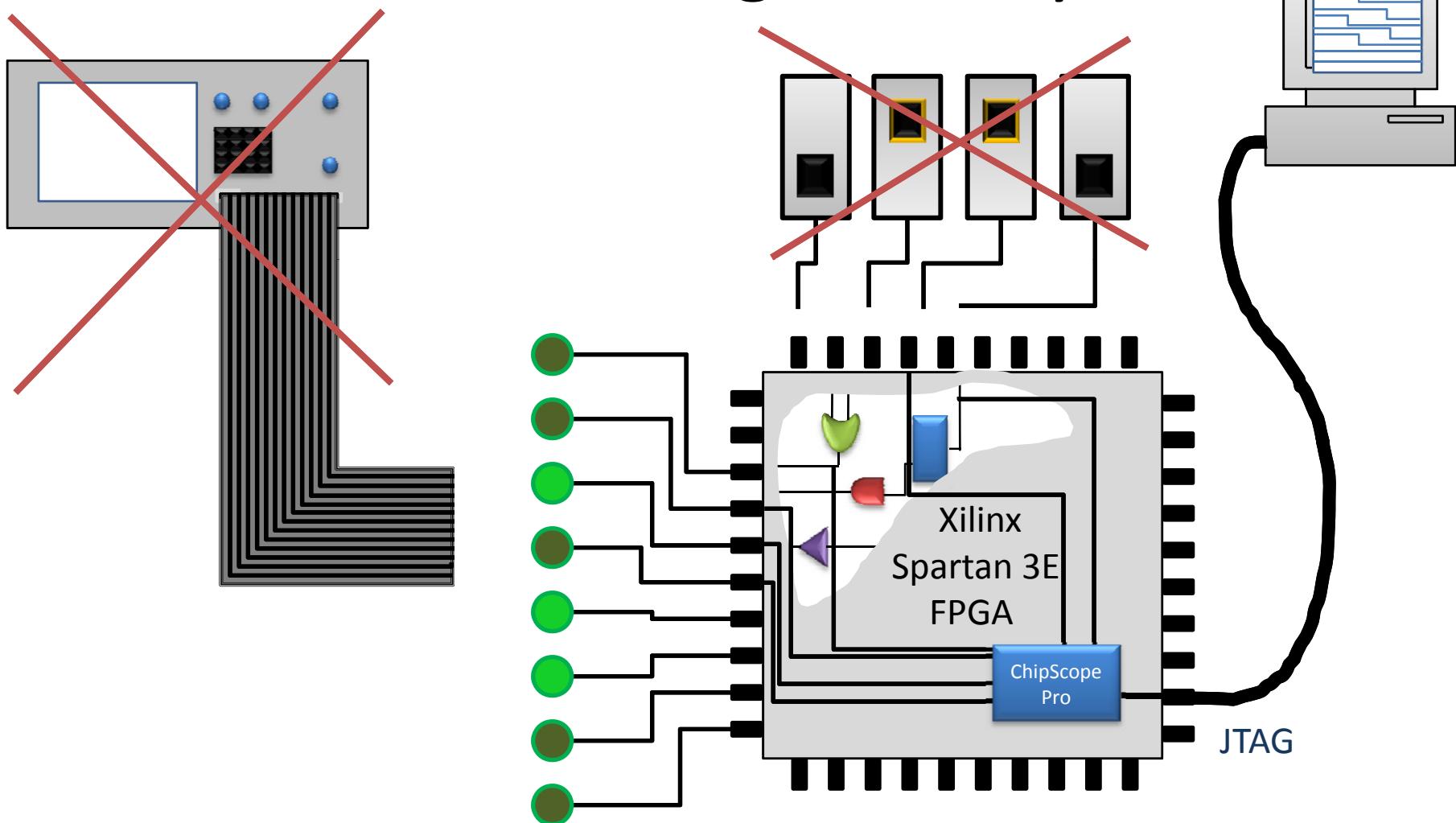
COE758 - Xilinx ISE 9.2 Tutorial 3

Integrating Virtual I/O into a project

Generating and using BlockRAM in the project



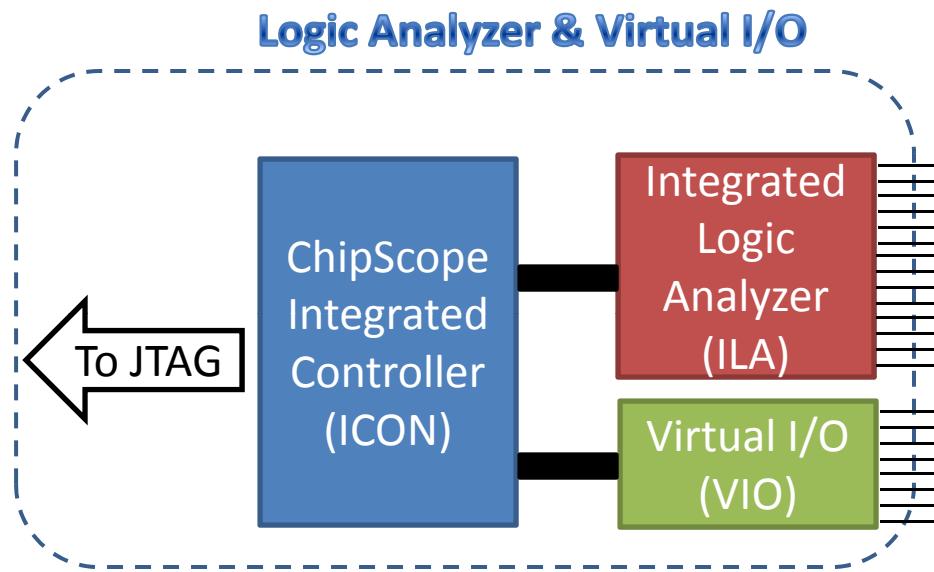
ChipScope Pro Signal Sampling and Virtual Signal Entry



ChipScope VIO Overview

- Similar to ILA it is generated by the ChipScope Pro Core generator
- Can simulate inputs to the FPGA circuits instead of connecting switches/input busses/etc.
- Can be connected to internal signals same as ILA
- Can be configured to be push/toggle buttons in GUI

ChipScope Configuration In this Project



1. ICON Module used with 2 control busses each going to ILA and VIO.
2. ILA used with 32 bit Data Line bus and 8 Bit trigger Bus as in previous tutorial.
3. VIO used with 18 bits of output to the FPGA circuit.

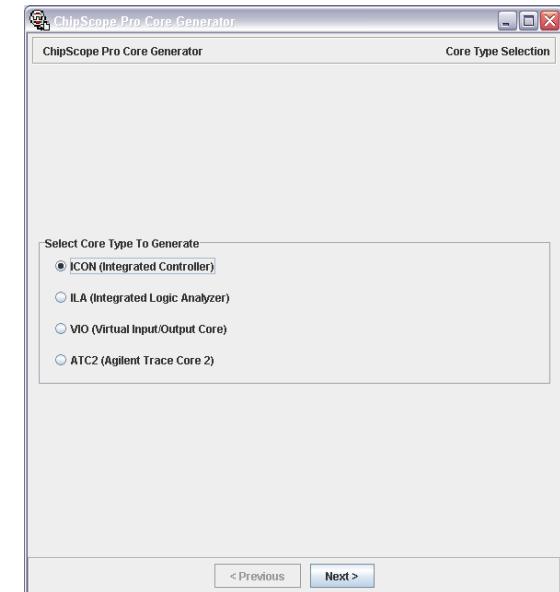
Overview of Tutorial 3 Project

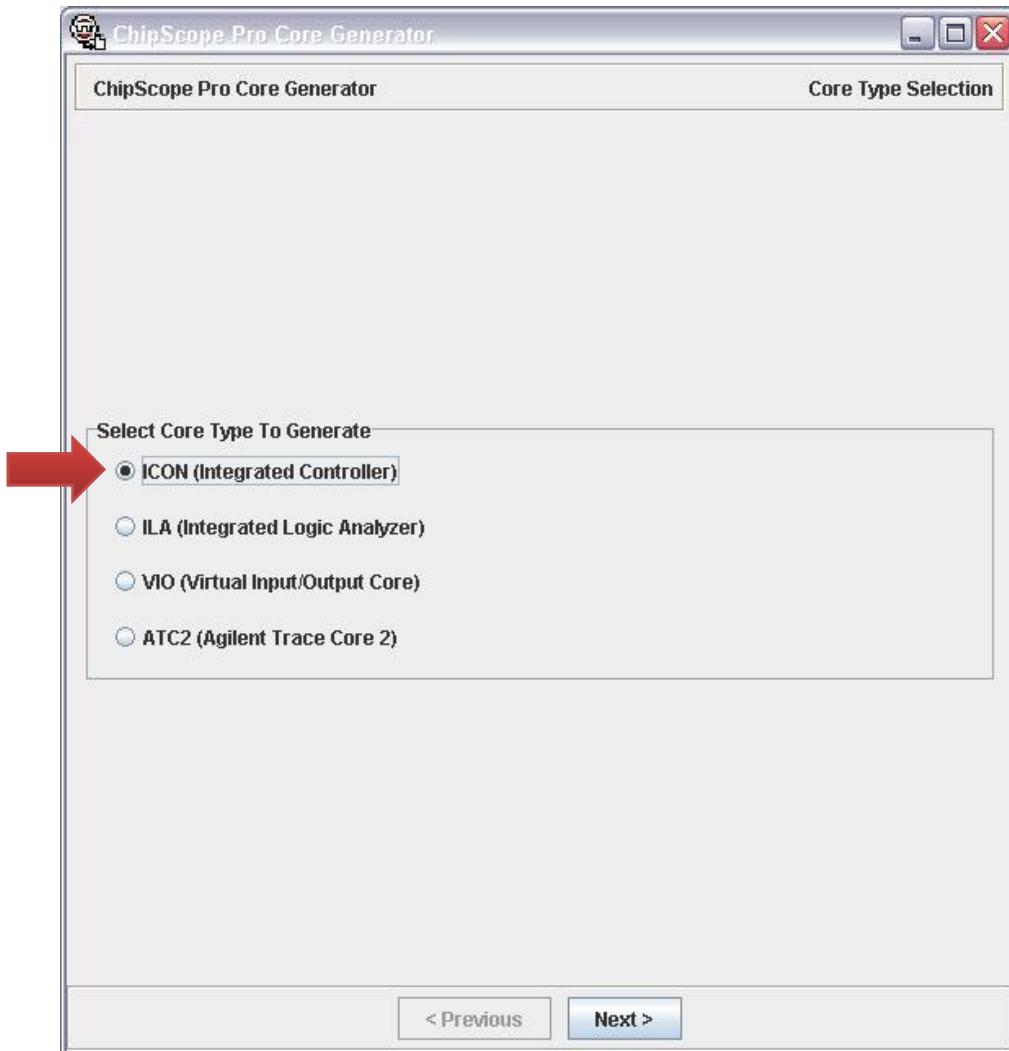
- Use tutorial 2 as a basis for this project where
 - counter gets incremented based on the position of the switch
 - ChipScope Pro is used for the signal readout
 - LEDs used for outputs of results
- Use of BlockRAM for storage and retrieval of large amounts of data
- Use of ChipScope Pro Virtual I/O for providing input to the BlockRAM address/data/write enable busses.
- Integrating all of the listed components together to be able to write data to BlockRAM using provided data from VIO and display output on ILA, as well as, LEDs.

Integration ChipScope ICON, ILA, and VIO into Project from Tutorial 2

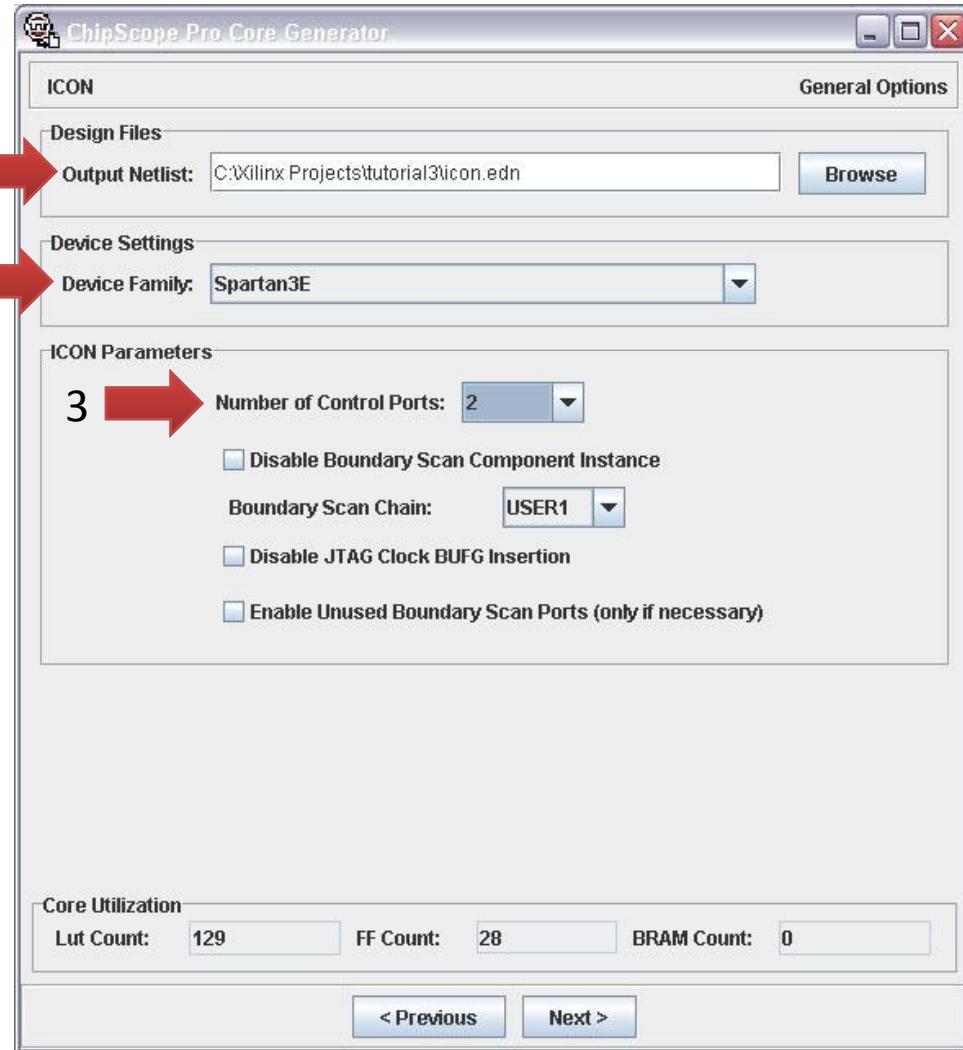
- Before inserting components we need to:
 - Copy everything related to ILA to new Tutorial 3 since it has not changed
 - Regenerate ICON for the project since it now requires to operate ILA and VIO modules
 - Generate VIO with 18 Asynchronous outputs

Open ChipScope Pro Core Generator

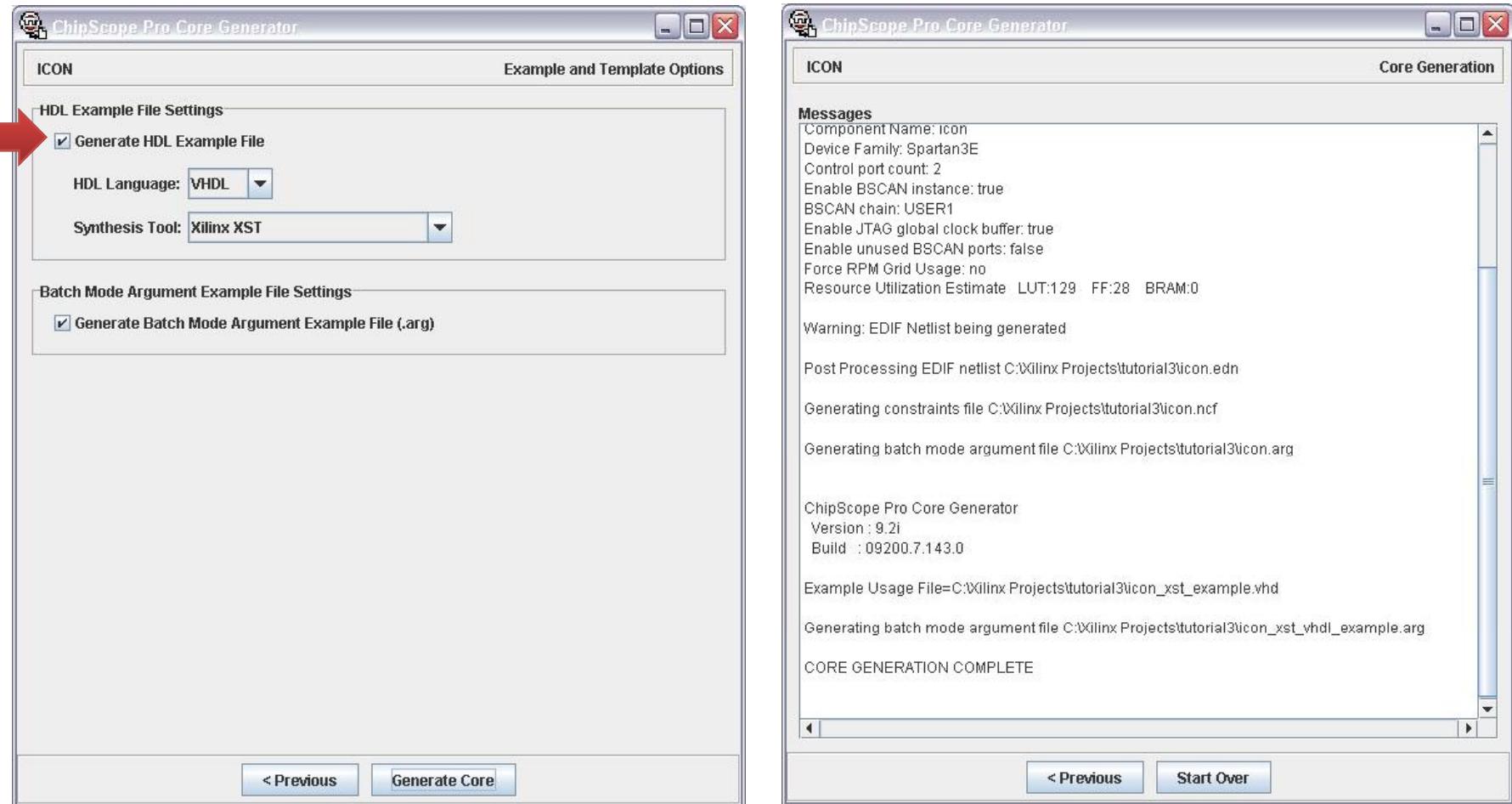




Select **ICON** (integrated Controller).
Press **Next>**



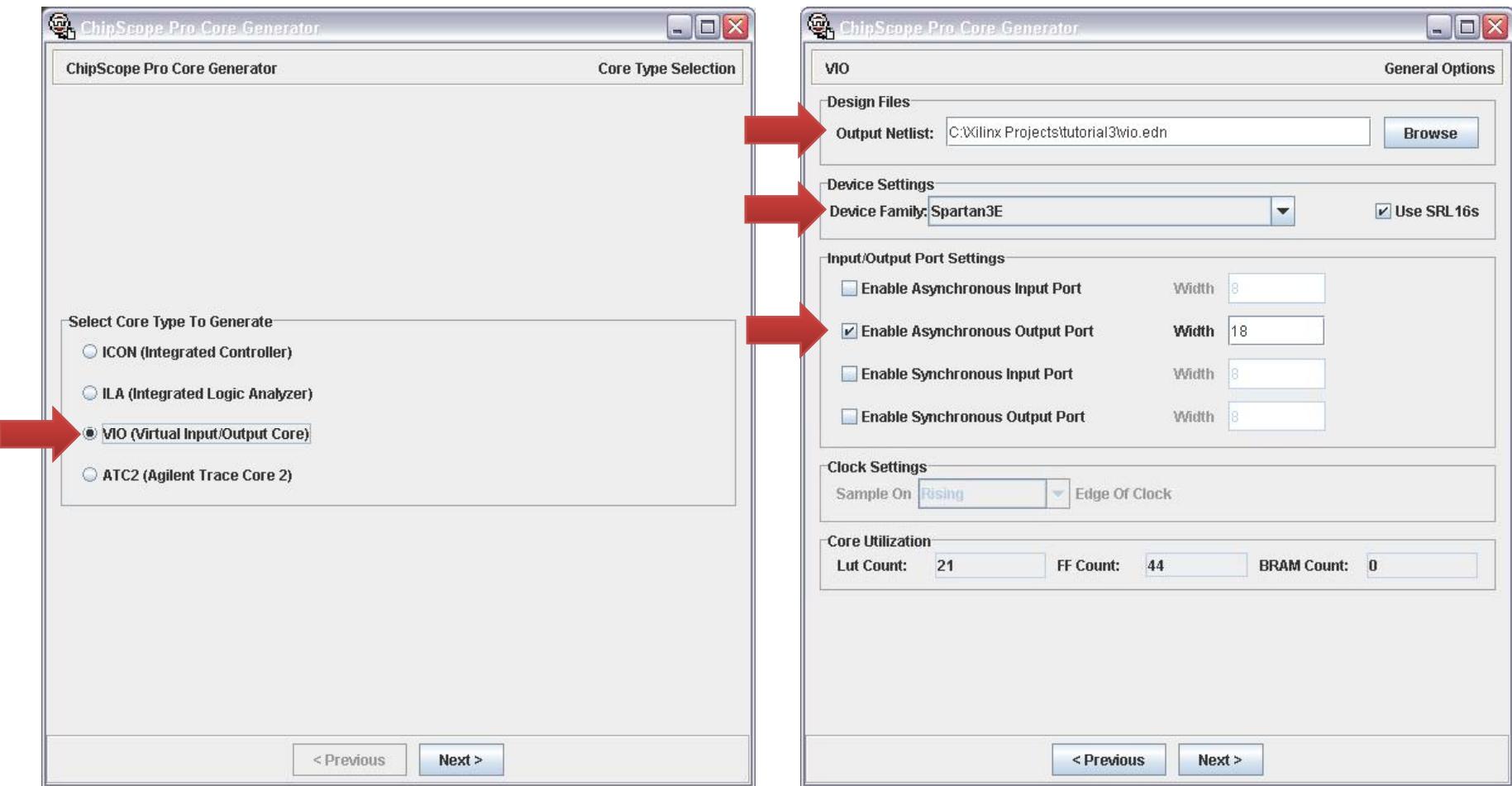
1. Specify directory where the project is located
2. Select appropriate FPGA device (Spartan3E)
3. Select number of ***Control Ports***. In this case it is 2 since we have ILA and VIO
Press ***Next>***



By selecting **Generate HDL Example File** ChipScope Pro Core generator will also generate the template which you will be able to copy directly to your design, thus minimizing possible errors.

Press **Generate Core** > this will generate core and all required files

Press **Start Over** to continue in generating ILA component

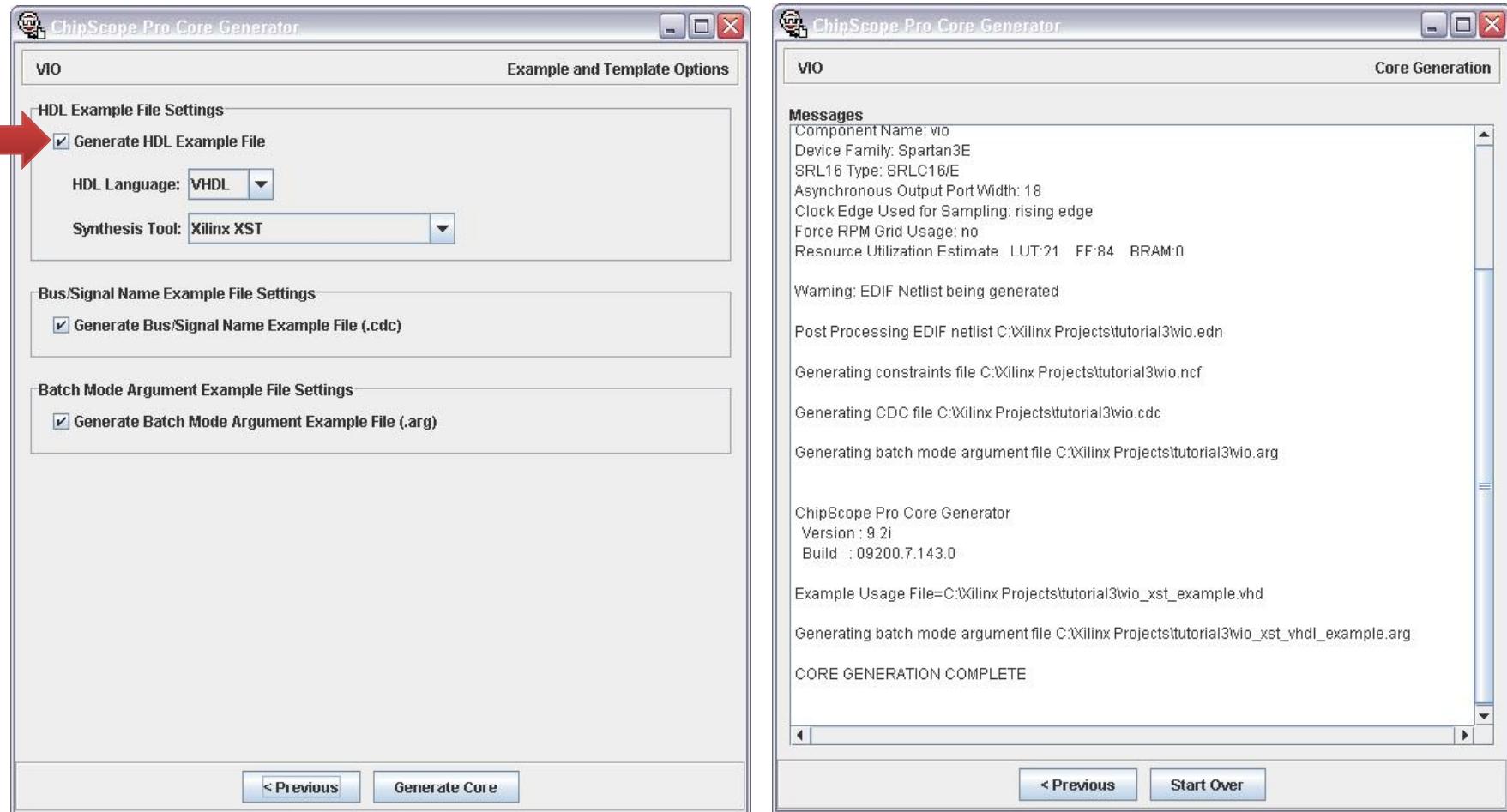


This time select ***VIO (Virtual Input/Output)*** and Press ***Next >***

As before, make sure that ***Output Netlist:*** is placed in the project directory and ***Spartan3E*** is selected as ***Device Family.***

Select ***Enable Asynchronous Outputs*** and set width of ***18***.

Press ***Next >***



Same as for the **ICON** select **Generate HDL Example File** and Press **Generate Core**

After core is generated you can close ChipScope Pro Core Generator and go back to the project to start integrating ChipScope Pro components into it.

```
27 --
28 -----
29 component vio
30   port
31   (
32     control      : in    std_logic_vector(35 downto 0);
33     async_out    : out   std_logic_vector(17 downto 0)
34   );
35 end component;
36
37
38 -----
39 -- VIO core signal declarations
40 --
41
42
43 signal control      : std_logic_vector(35 downto 0);
44 signal async_out    : std_logic_vector(17 downto 0);
45 begin
46
47   --
48   -- VIO core instance
49   --
50
51   i_vio : vio
52     port map
53   (
54     control      => control,
55     async_out    => async_out
56   );
57
```

Started : "Launching ISE Text Editor to edit tutorial.vhd".
Started : "Launching Design Summary".

Same as in the tutorial 2 open ***vio_xst_example.vhd*** and ***icon_xst_example.vhd*** and copy the declaration and instances to the main project file.

```

Xilinx - ISE - C:\Xilinx Projects\tutorial3\tutorial3.ise - [tutorial.vhd]
File Edit View Project Source Process Window Help
Sources x
Sources for: Synthesis/Implementation
tutorial3
xc3s500e-5fg320
tutorial - Behavioral (tutorial)
i_icon - icon
i_ilia - ila
tutorial.ucf (tutorial.ucf)
Processes x
Processes for: tutorial - Behavioral
Add Existing Source
Create New Source
View Design Summary
Design Utilities
User Constraints
Synthesize - XST
Implement Design
Generate Programming File
Analyze Design Using ChipScope
Processes
tutorial.vhd* Design Summary vio_xst_example.vhd icon_xst_example.vhd
Started : "Launching ISE Text Editor to edit tutorial.vhd".
Started : "Launching Design Summary".
Transcript
Console Errors Warnings Tcl Shell Find in Files
Ln 108 Col 1 CAPS NUM SCRL YHDL ...

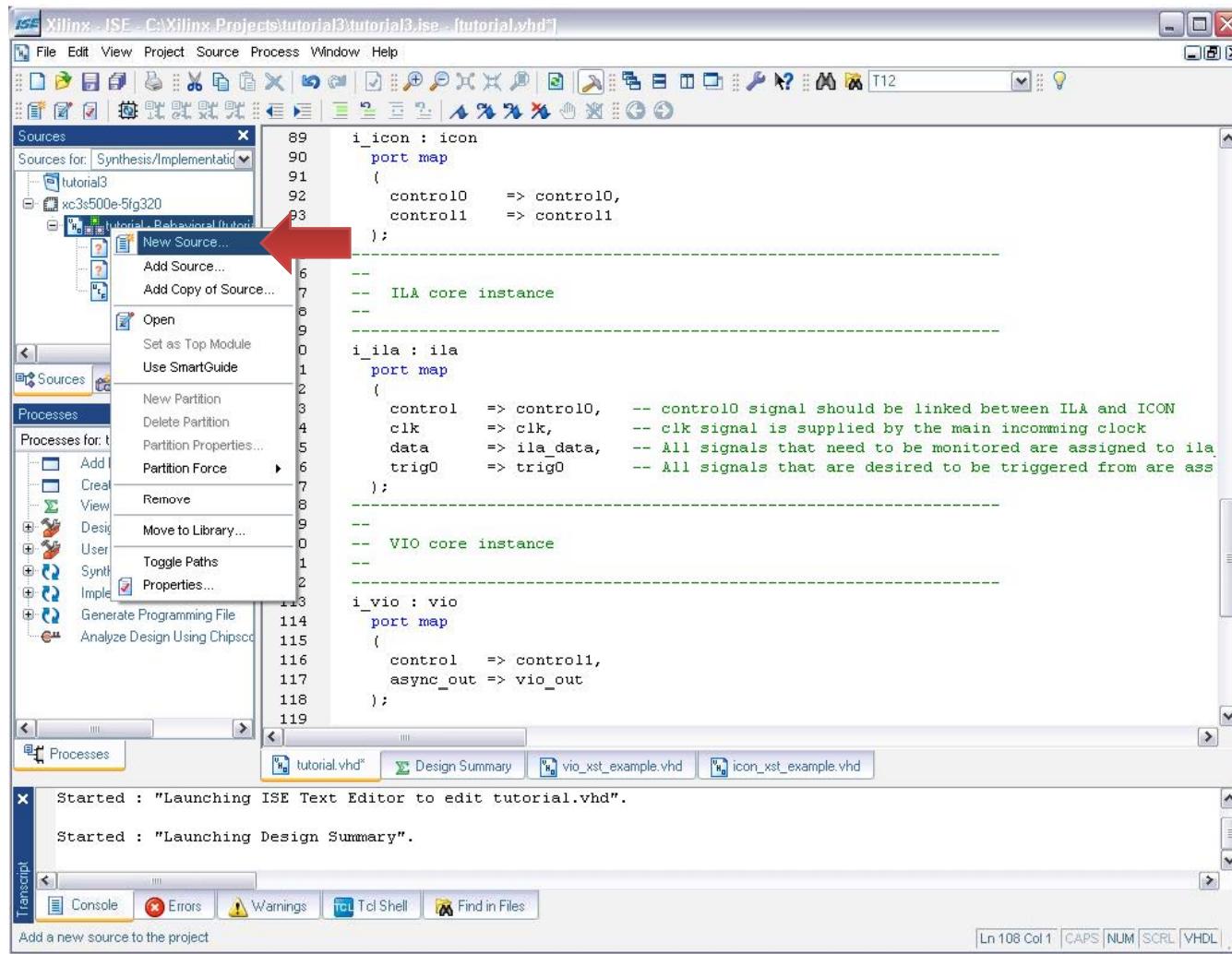
```

```

89  i_icon : icon
90    port map
91    (
92      control0  => control0, ←
93      control1  => control1 ←
94    );
95
96  --
97  -- ILA core instance
98  --
99
100 i_ilia : ila
101   port map
102   (
103     control  => control0, ← ro0 signal should be linked between ILA and ICON
104     clk      => clk,      -- clk signal is supplied by the main incoming clock
105     data     => ila_data,  -- All signals that need to be monitored are assigned to ila
106     trig0   => trig0      -- All signals that are desired to be triggered from are ass
107   );
108
109  --
110  -- VIO core instance
111  --
112 i_vio : vio
113   port map
114   (
115     control  => control1, ←
116     async_out => vio_out
117   );
118
119

```

In main project **ICON** has two control outputs, where **control0** is connected to **ILA** and **control1** connected to **VIO**. Output of **VIO** is set to **vio_out** bus of 18bit width. The overall structure of instances is shown in the figure above.

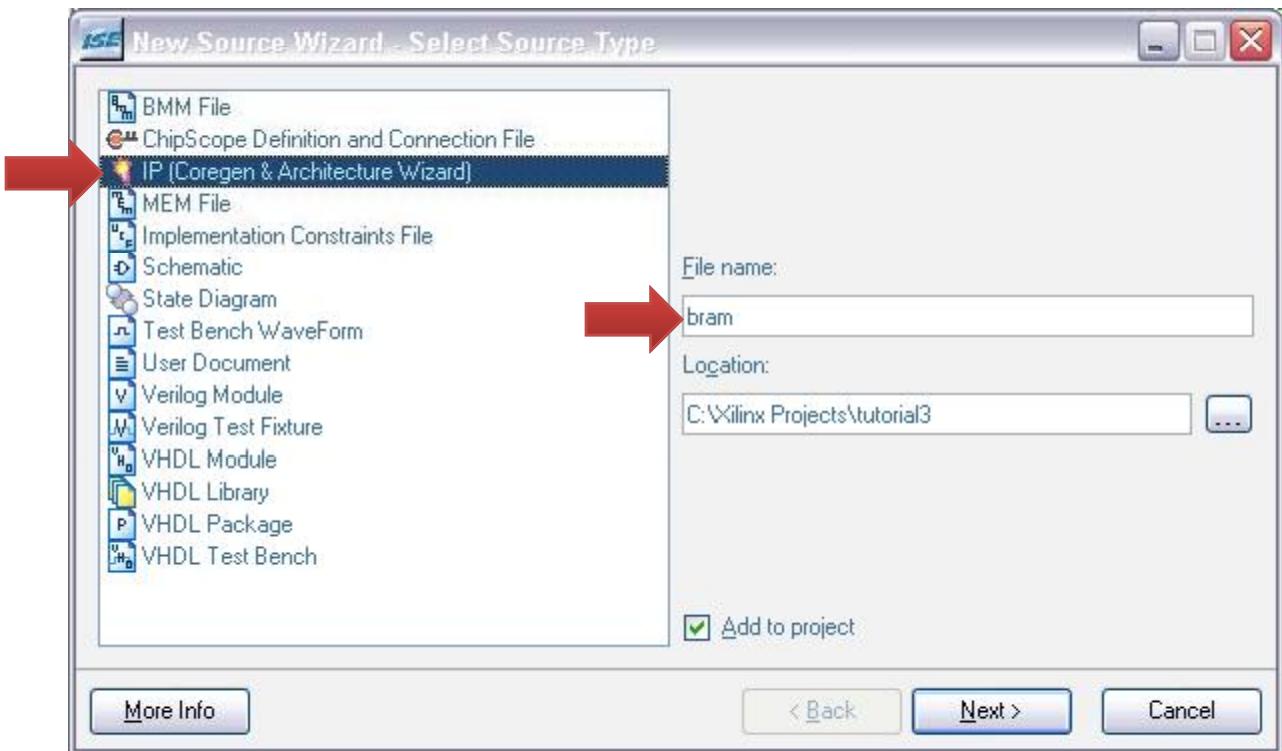


The screenshot shows the Xilinx ISE 14.7 interface. The main window displays a VHDL code editor with the file `tutorial.vhd` open. The code defines several components: `i_icon`, `i_ilia`, and `i_vio`. A red arrow points to the `New Source...` option in the context menu that appears when right-clicking on the project tree under `Sources`.

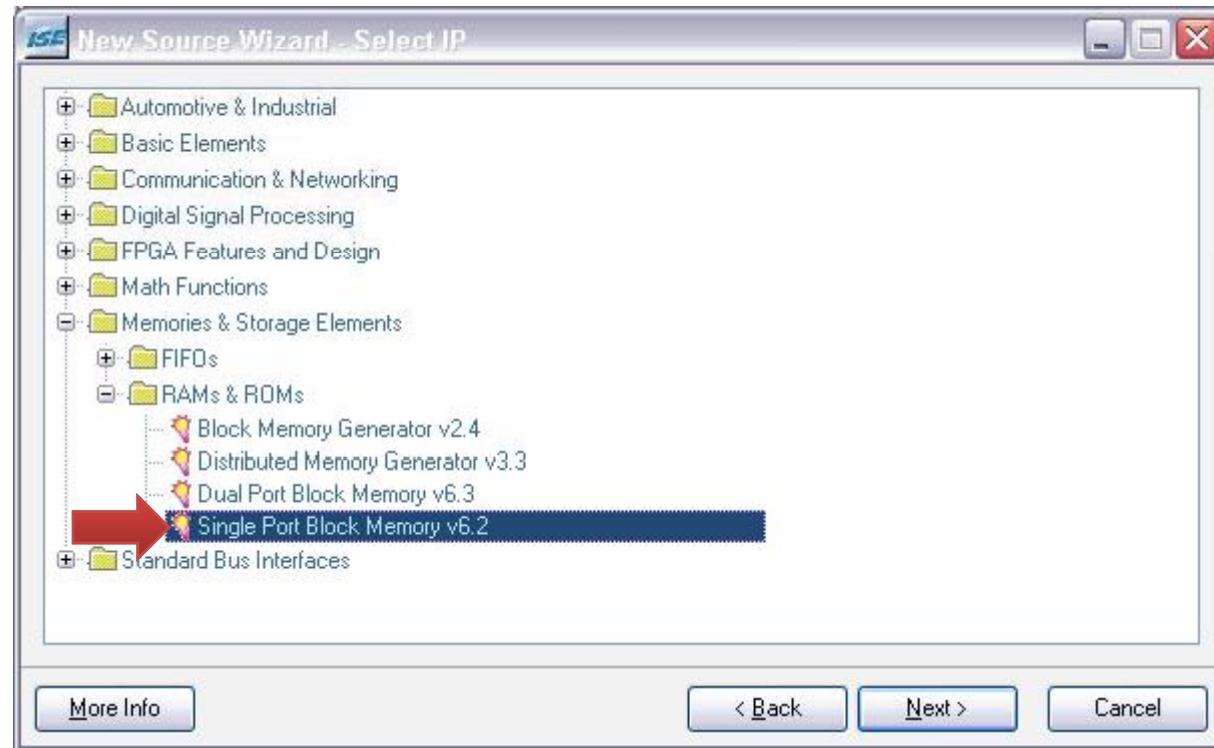
```
89 i_icon : icon
90 port map
91 (
92     control0  => control0,
93     control1  => control1
94 );
95
96 -- ILA core instance
97
98 i_ilia : ila
99 port map
100 (
101     control   => control0,    -- control0 signal should be linked between ILA and ICON
102     clk       => clk,        -- clk signal is supplied by the main incoming clock
103     data      => ila_data,   -- All signals that need to be monitored are assigned to ila
104     trig0    => trig0       -- All signals that are desired to be triggered from are ass
105 );
106
107 -- VIO core instance
108
109 i_vio : vio
110 port map
111 (
112     control   => control1,
113     async_out => vio_out
114 );
115
116
117
118
119
```

Now we add the BlockRAM module, and that is done by running a IP core generator .

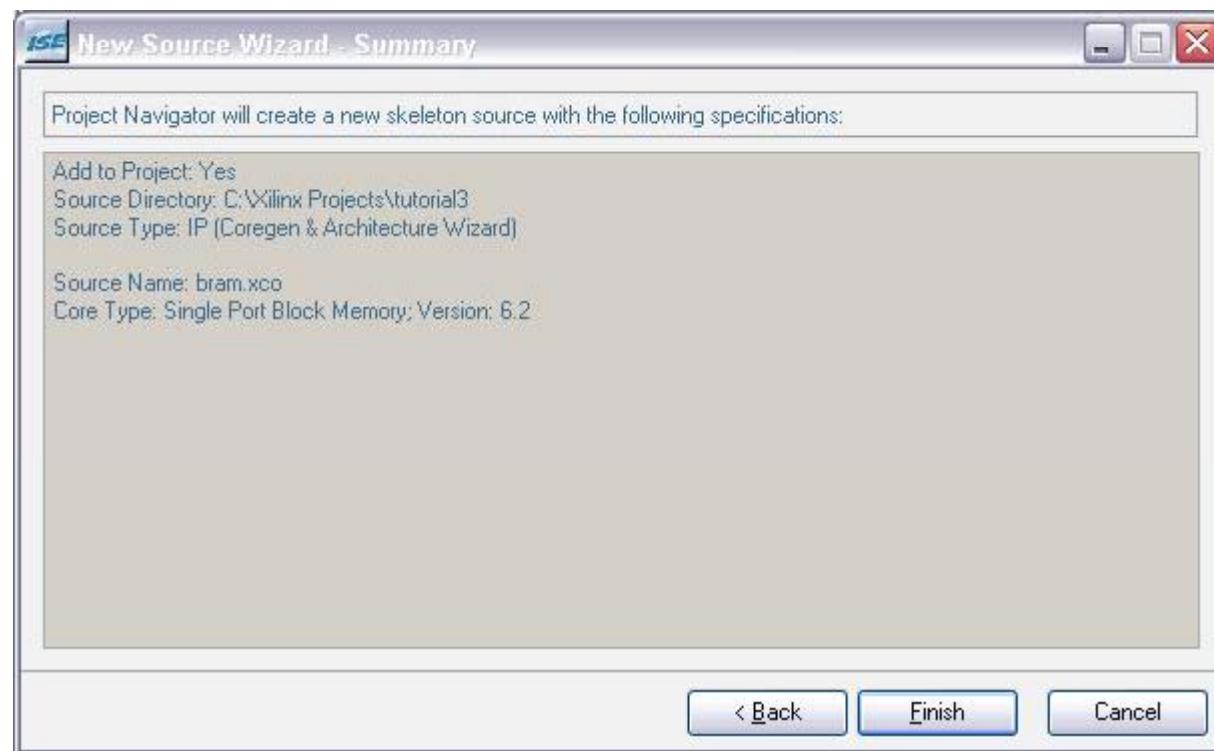
Start from adding **New Source** to the project.



Indicate **bram** as a file name and select **IP (Coregen & Architecture Wizard)**
Select **Add to project**
Press **Next>**

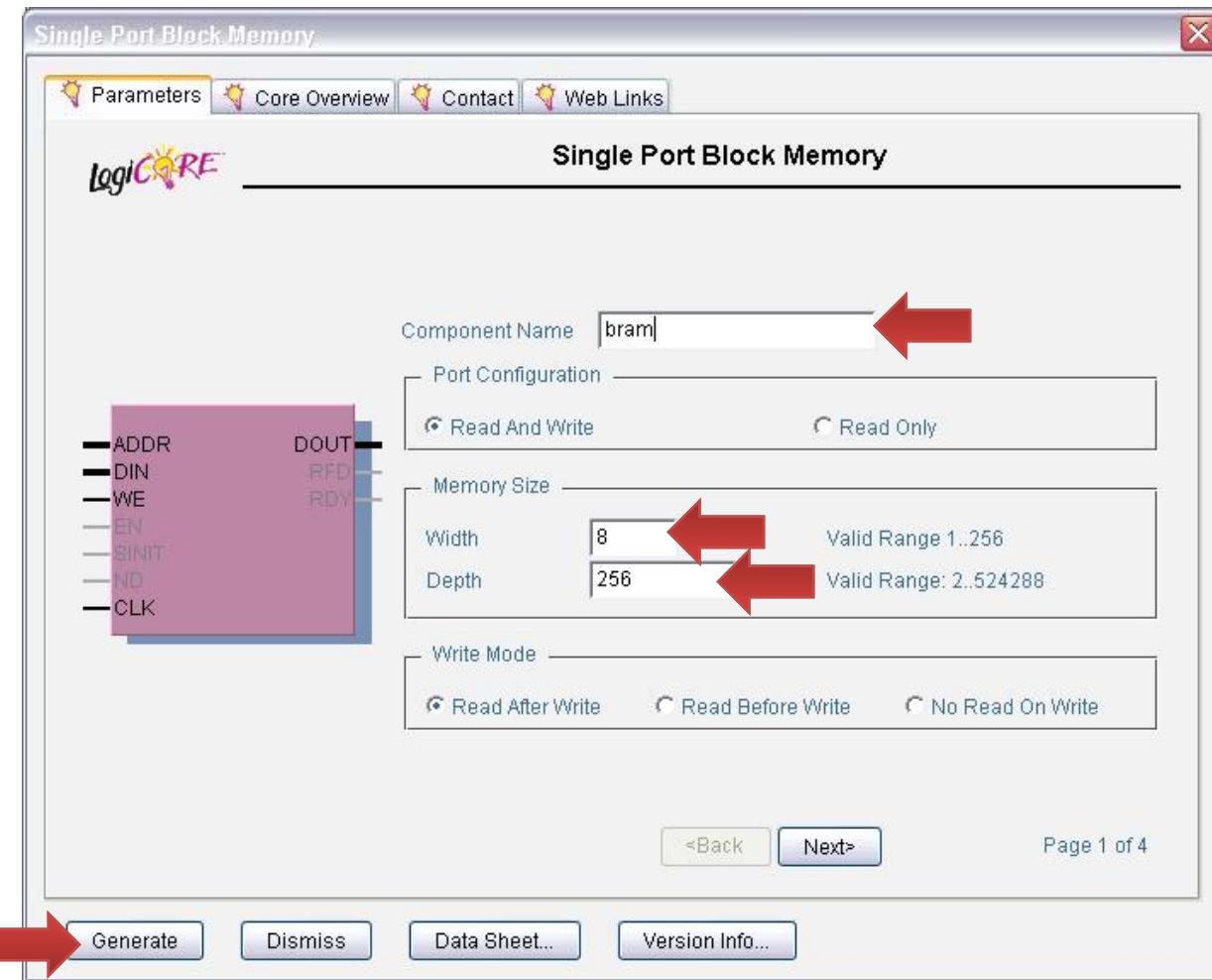


Select the **Single Port Memory** from the **Memories & Storage Elements**
Press **Next>**



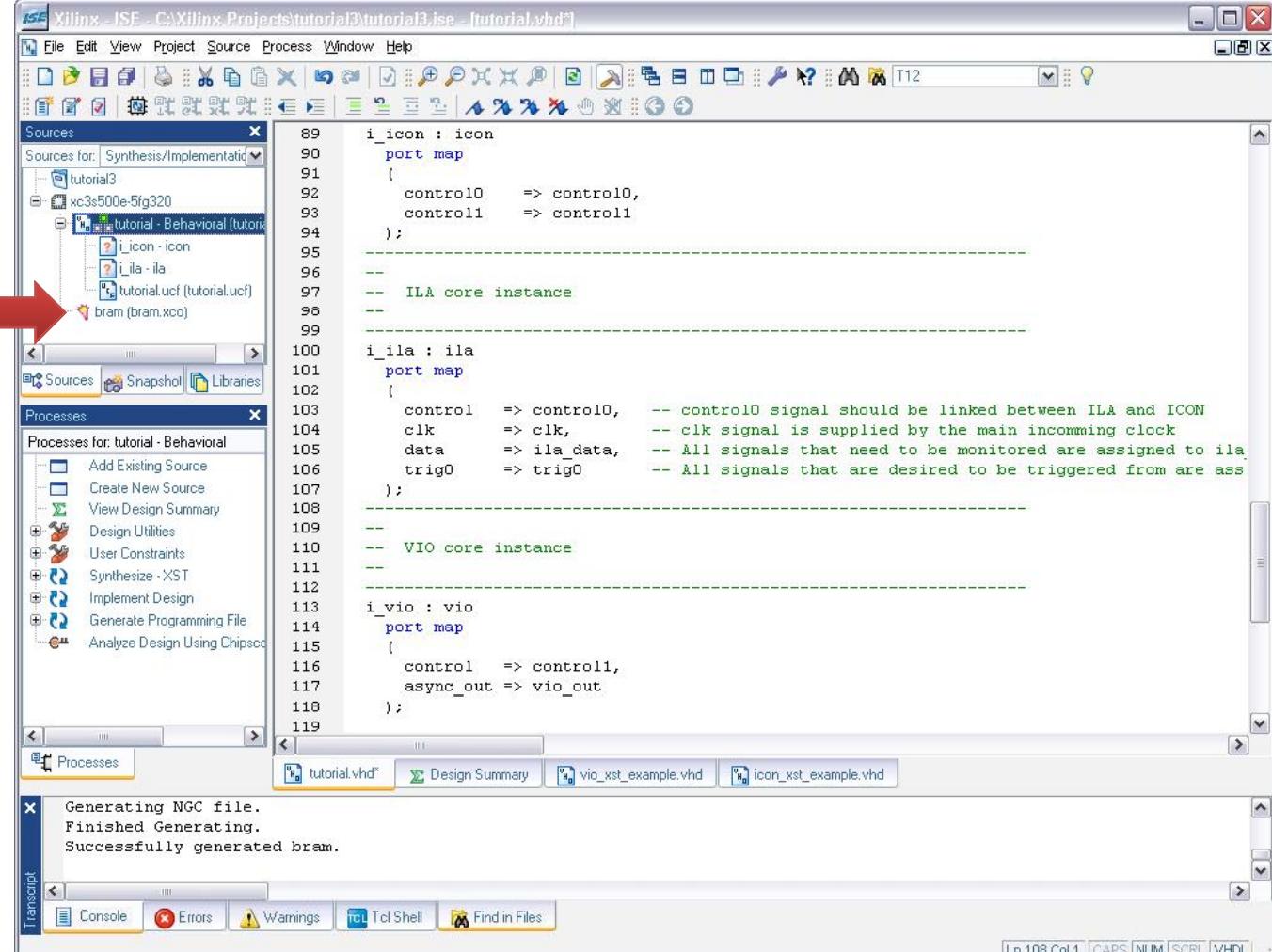
Last window shows the summary of the generated core and location where it is deposited

Press **Finish**



A wizard for the memory will show the options which can be set to generate desired memory. For this example we will select width -> **8 bits** depth **256 words**. There are other options that can be selected but for now we will avoid them to keep everything simple. For more detailed explanation click on datasheet button.

Press **Generate** to generate the memory with parameters selected.



A screenshot of the Xilinx ISE Project Manager interface. The main window shows a VHDL code editor with a portion of a VHDL file displayed. The code defines several components: *i_icon*, *i_ilia*, and *i_vio*. A red arrow points to the *bram* component in the *Sources* list on the left. The *Processes* list on the right shows various design utilities like Synthesize-XST and Implement Design. The bottom status bar indicates the current line is 108, column 1, and shows keyboard shortcut keys for CAPS, NUM, SCRL, and YHDL.

```
89 i_icon : icon
90 port map
91 (
92     control0    => control0,
93     control1    => control1
94 );
95
96
97 -- ILA core instance
98
99
100 i_ilia : ilia
101 port map
102 (
103     control    => control0, -- control0 signal should be linked between ILA and ICON
104     clk        => clk,    -- clk signal is supplied by the main incoming clock
105     data       => ila_data, -- All signals that need to be monitored are assigned to ilia
106     trig0     => trig0    -- All signals that are desired to be triggered from are ass
107 );
108
109
110 -- VIO core instance
111
112
113 i_vio : vio
114 port map
115 (
116     control    => control1,
117     async_out  => vio_out
118 );
119
```

In the main project window a ***bram*** IP core will appear. However, it has to be initialized similarly as were Chipscope Pro's ***ICON***, ***ILA***, and ***VIO*** modules. Following slides includes code for the project, which can be downloaded from the website, as well in a zip file.

```

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

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

architecture Behavioral of tutorial is

signal counter: std_logic_vector(29 downto 0);

-- ICON core component declaration
-- 

component icon
    port
    (
        control0 : out std_logic_vector(35 downto 0);
        control1 : out std_logic_vector(35 downto 0)
    );
end component;

-- 
-- ILA core component declaration
-- 

component ila
    port
    (
        control : in 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;

```

```

-----
-- VIO core component declaration
-- 

component vio
    port
    (
        control  : in std_logic_vector(35 downto 0);
        async_out : out std_logic_vector(17 downto 0)
    );
end component;

-----
-- BRAM component declaration
-- 

component bram
    port
    (
        addr : IN std_logic_VECTOR(7 downto 0);
        clk  : IN std_logic;
        din  : IN std_logic_VECTOR(7 downto 0);
        dout : OUT std_logic_VECTOR(7 downto 0);
        we   : IN std_logic);
    end component;

signal control0      : std_logic_vector(35 downto 0);
-- control bus interconnected between ICON and ILA
signal control1      : std_logic_vector(35 downto 0);
-- control bus interconnected between ICON and VIO
signal ila_data      : std_logic_vector(31 downto 0);
-- bus to which all of the signals to be monitored by ILA are connected
signal vio_out       : std_logic_vector(17 downto 0);
-- bus which from which VIO outputs all the async signals
signal trig0         : std_logic_vector(7 downto 0);

-- BRAM Signals
signal bram_addr, bram_din, bram_dout: std_logic_vector(7 downto 0);
signal bram_we: std_logic;

```

```

Begin
-----
-- BRAM component instance

bram_0 : bram
port map (
    addr => bram_addr,
    clk => clk,
    din => bram_din,
    dout => bram_dout,
    we => bram_we);
-----
-- ICON core instance

i_icon : icon
port map
(
    control0 => control0,
    control1 => control1
);
-----
-- ILA core instance

i_ilal : ila
port map
(
    control => control0,
-- control0 signal should be linked between ILA and ICON
    clk => clk,
-- clk signal is supplied by the main incomming clock
    data => ila_data,
-- All signals that need to be monitored are assigned to ila_data Bus
    trig0 => trig0
-- All signals that are desired to be triggered from are assigned to trig0
);
-----
-- VIO core instance

i_vio : vio
port map
(
    control => control1,
    async_out => vio_out
);
-----
```

-- As in the previous tutorial this process counts counter up or down
--depending on the position of the switch

```

process(clk)
Begin
    if(clk'Event and clk='1') then
        if(switch(0)='1') then
            counter<=counter+'1';
        else
            counter<=counter-'1';
        end if;
    end if;
end process;
```

-- In this process bram assigned data from the VIO (7 downto 0)
-- output or a counter value based on the VIO (17) output signal.

```

process(clk)
begin
    if(clk'Event and clk='1') then
        if(vio_out(17)='1') then
            bram_din(7 downto 0)<=counter(7 downto 0);
        else
            bram_din(7 downto 0)<=vio_out(7 downto 0);
        end if;
    end if;
end process;
```

-- LEDs output the status of the

```

led(7 downto 0)<=bram_dout(7 downto 0);
-- ILA data bus 7->0 shows what goes into the BlockRAM
ila_data(7 downto 0)<=bram_din(7 downto 0);
-- ILA data bus 15->8 shows what BlockRAM outputs
ila_data(15 downto 8)<=bram_dout(7 downto 0);
-- ILA data bus 23->16 shows what address is set on BlockRAM
ila_data(23 downto 16)<=bram_addr(7 downto 0);
-- ILA data bus 31->24 outputs lowest 8 bits of the counter
ila_data(31 downto 24)<=counter(7 downto 0);
-- assignment of VIO output bus to the BlockRAM address
bram_addr(7 downto 0)<=vio_out(15 downto 8);
-- assignment of VIO output signal (16) to BRAM write enable signal
bram_we<=vio_out(16);
trig0(0)<=bram_we; -- triggering from the bram write enable signal
```

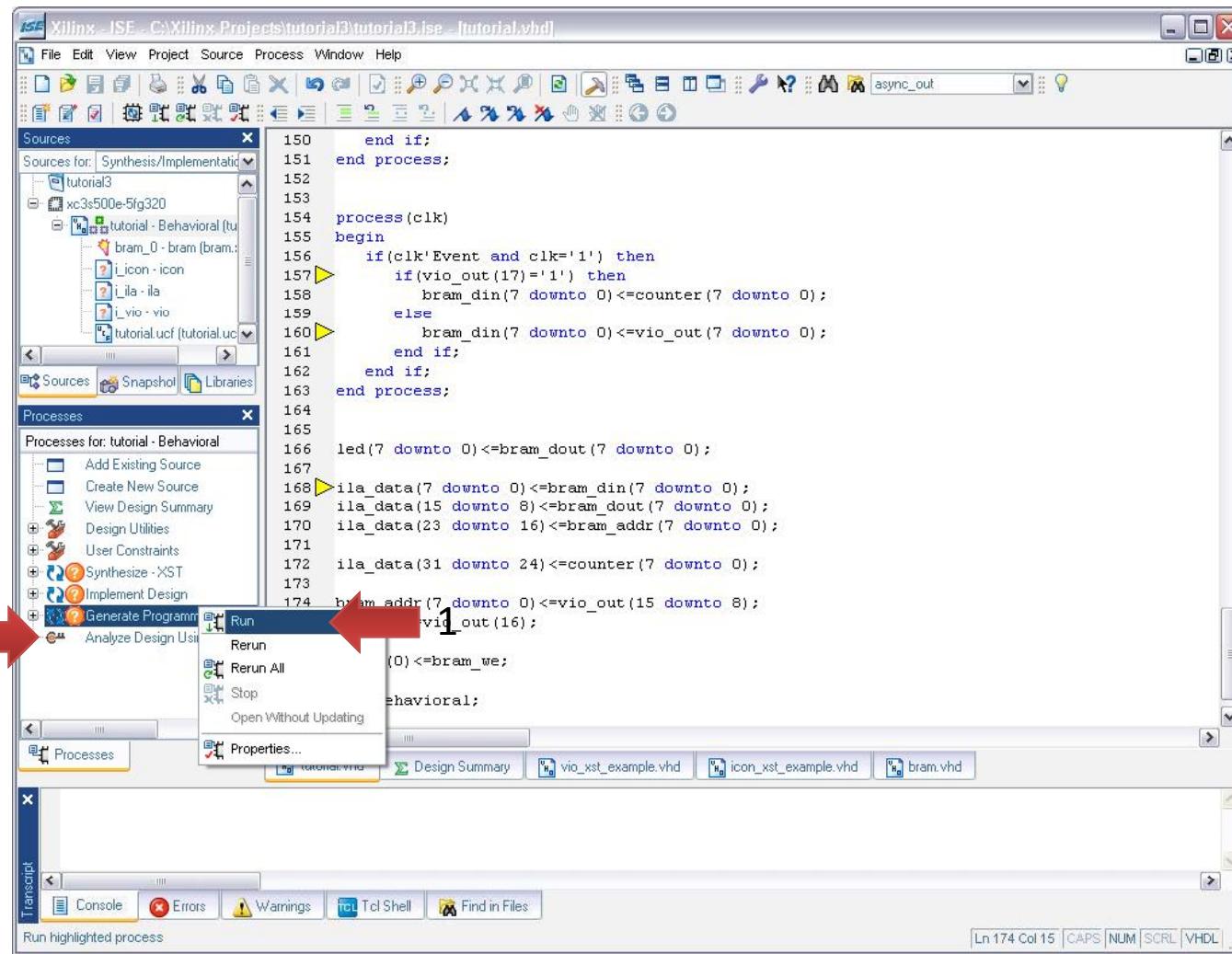
```

end Behavioral;
```

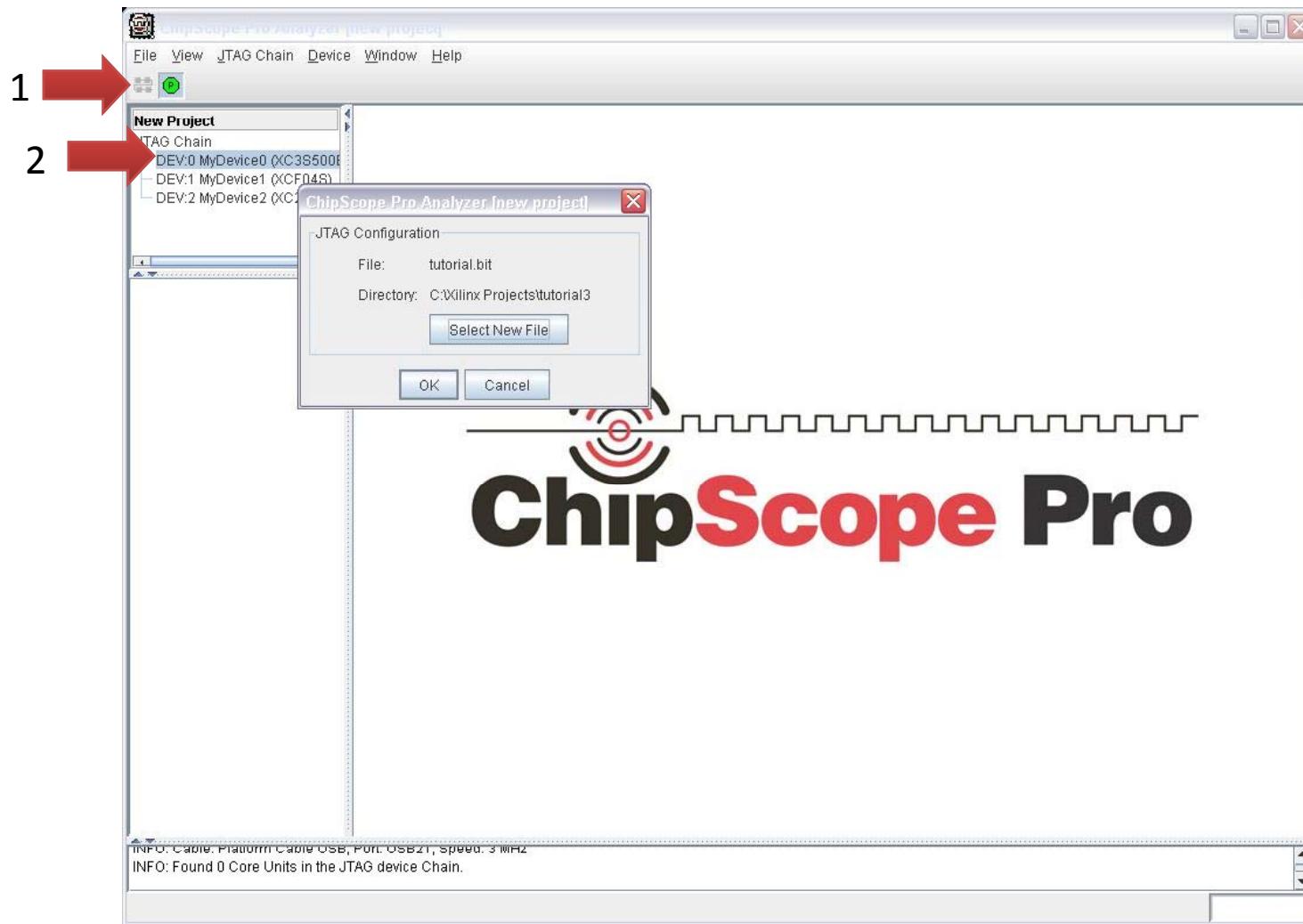
The screenshot shows the Xilinx ISE 14.7 interface. The top menu bar includes File, Edit, View, Project, Source, Process, Window, and Help. The main window features a toolbar with various icons. On the left, there are two panes: 'Sources' and 'Processes'. The 'Sources' pane shows a project structure for 'tutorial3' under 'xc3s500e-5fg320', with a red arrow pointing to the 'bram' IP core. The 'Processes' pane lists design utilities like Synthesize-XST, Implement Design, and Analyze Design Using ChipScope. The central area contains a code editor with VHDL code for a BRAM component. The bottom navigation bar includes tabs for 'tutorial.vhd', 'Design Summary', 'vio_xst_example.vhd', 'icon_xst_example.vhd', and 'bram.vhd'. A transcript window at the bottom displays messages about launching the ISE Text Editor and recustomizing IP. The status bar at the bottom right shows 'Ln 92 Col 29 CAPS NUM SCRL YHDL'.

```
74 component bram
75 port (
76     addr: IN std_logic_VECTOR(7 downto 0);
77     clk: IN std_logic;
78     din: IN std_logic_VECTOR(35 downto 0);
79     dout: OUT std_logic_VECTOR(35 downto 0);
80     we: IN std_logic);
81 end component;
82
83
84 signal control0      : std_logic_vector(35 downto 0);
85 signal control1      : std_logic_vector(35 downto 0);
86 signal ila_data       : std_logic_vector(31 downto 0);
87 signal vio_out        : std_logic_vector(17 downto 0);
88 signal trig0          : std_logic_vector(7 downto 0);
89 -- BRAM Signals
90 signal bram_addr, bram_din, bram_dout: std_logic_vector(7 downto 0);
91 signal bram_we: std_logic;
92
93 begin
94     bram_0 : bram
95         port map (
96             addr => bram_addr,
97             clk => clk,
98             din => bram_din,
99             dout => bram_dout,
100            we => bram_we);
101
102
103
104
```

After entering the code and initializing the BlockRAM in the project and saving the file **bram** IP core will appear inside the project tree.

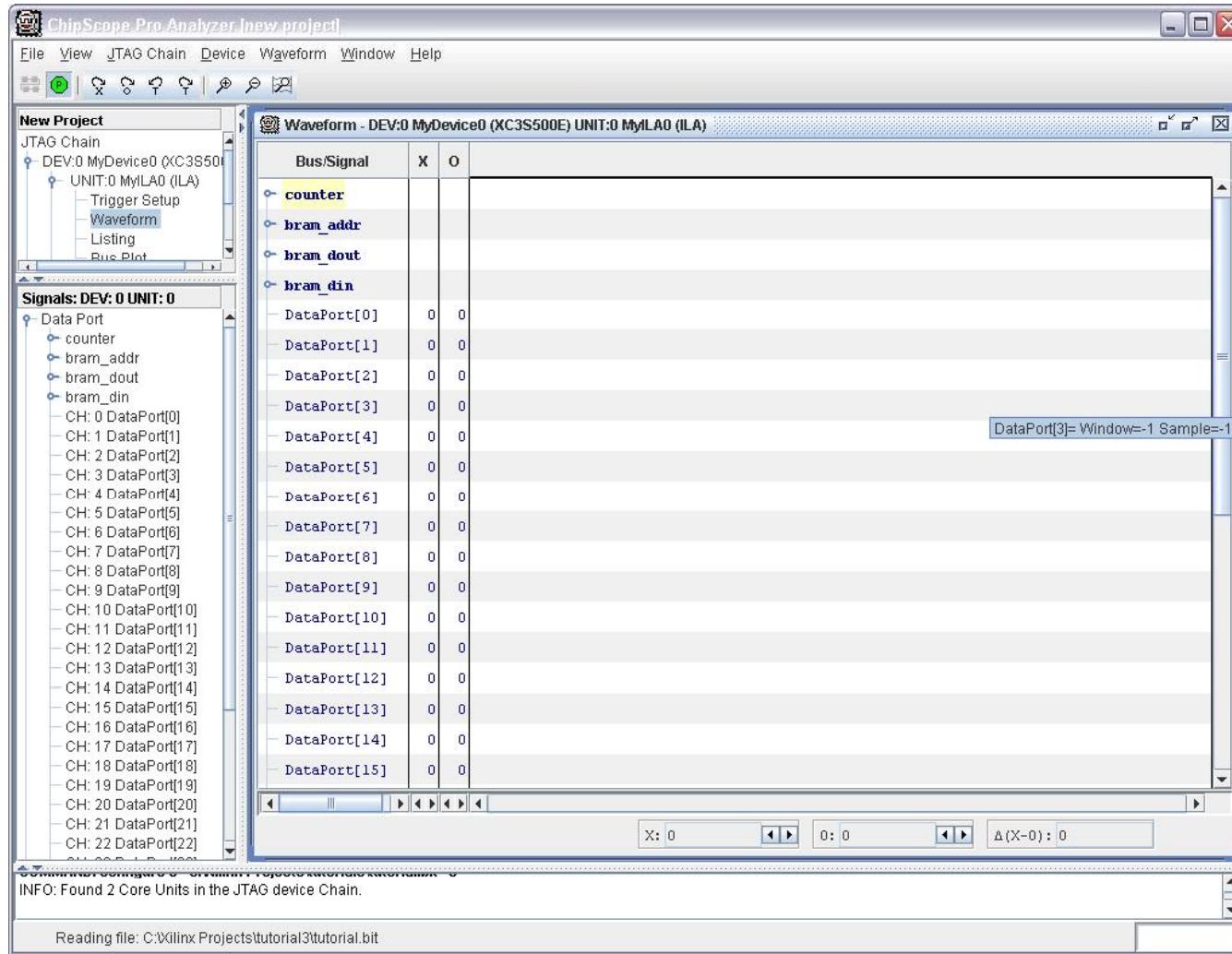


1. Right click on ***Generate Programming File*** and select ***Run***. If any of the errors occur double click on errors and correct them according to error messages.
2. When generation of configuration file completes successfully double click on ***Analyze Design Using ChipScope Pro***.

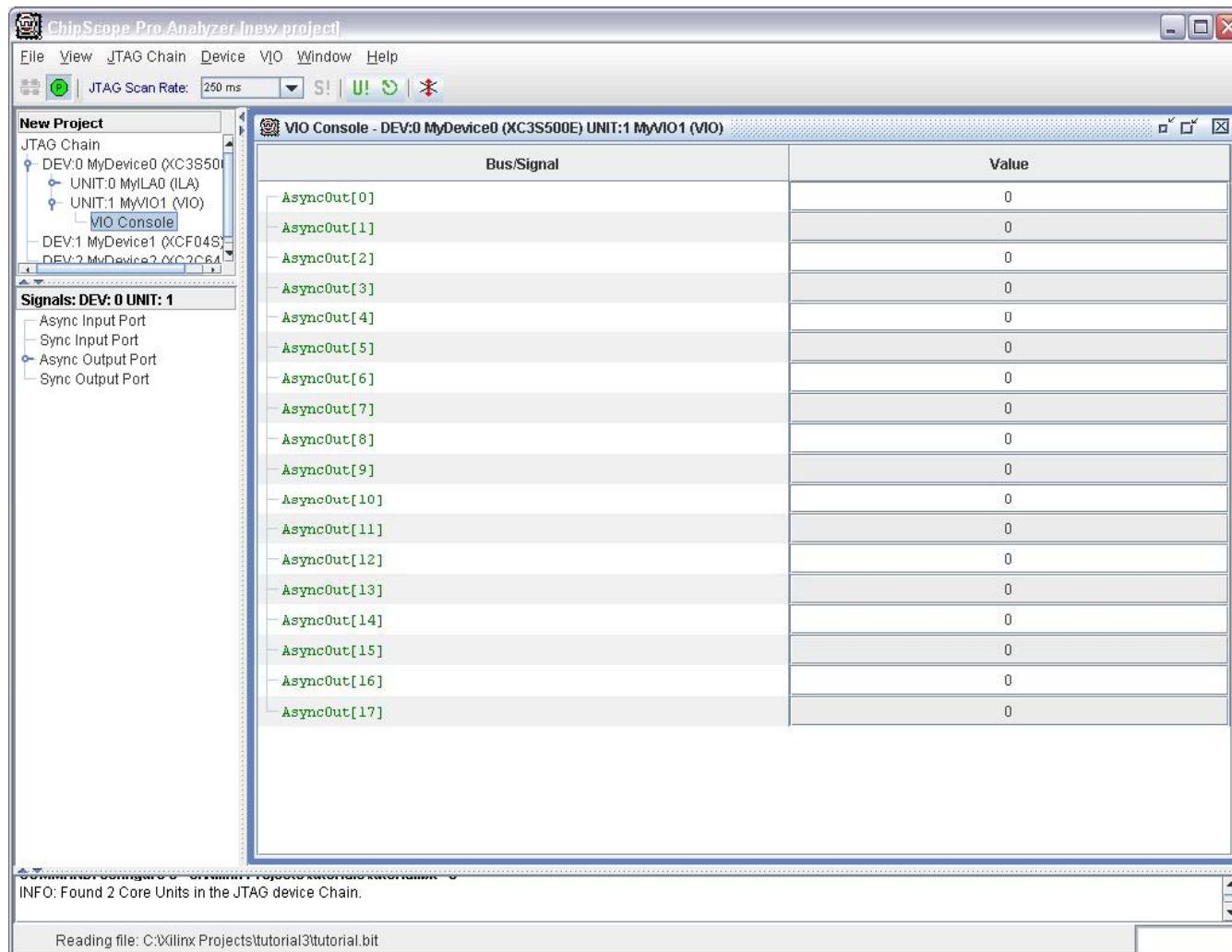


Click on the daisy chain button at the top left corner as was done in the previous tutorial.

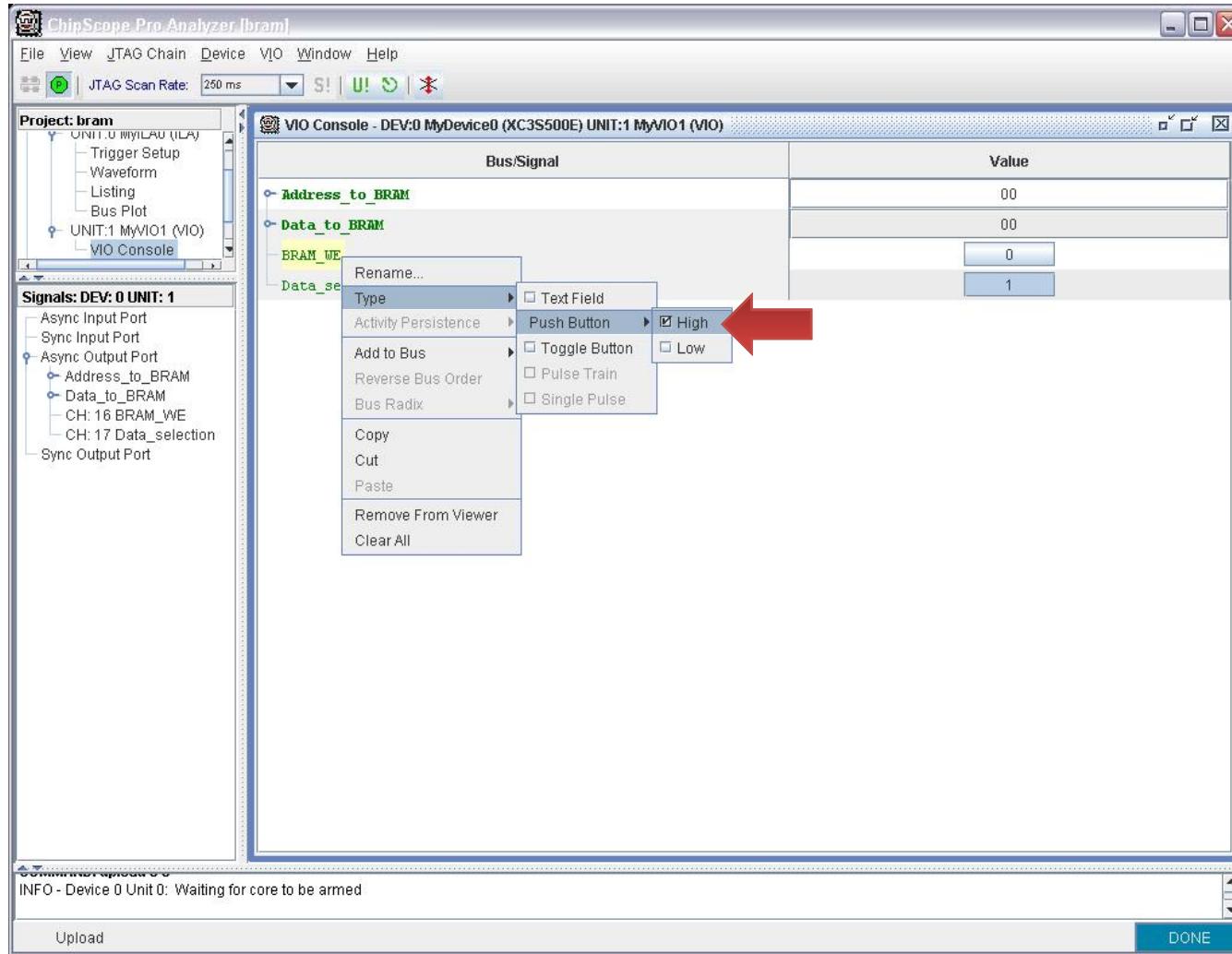
1. Select **XC3S500** device and right click to select **Configure**.
2. Select the configuration file and press OK to configure the FPGA with it.



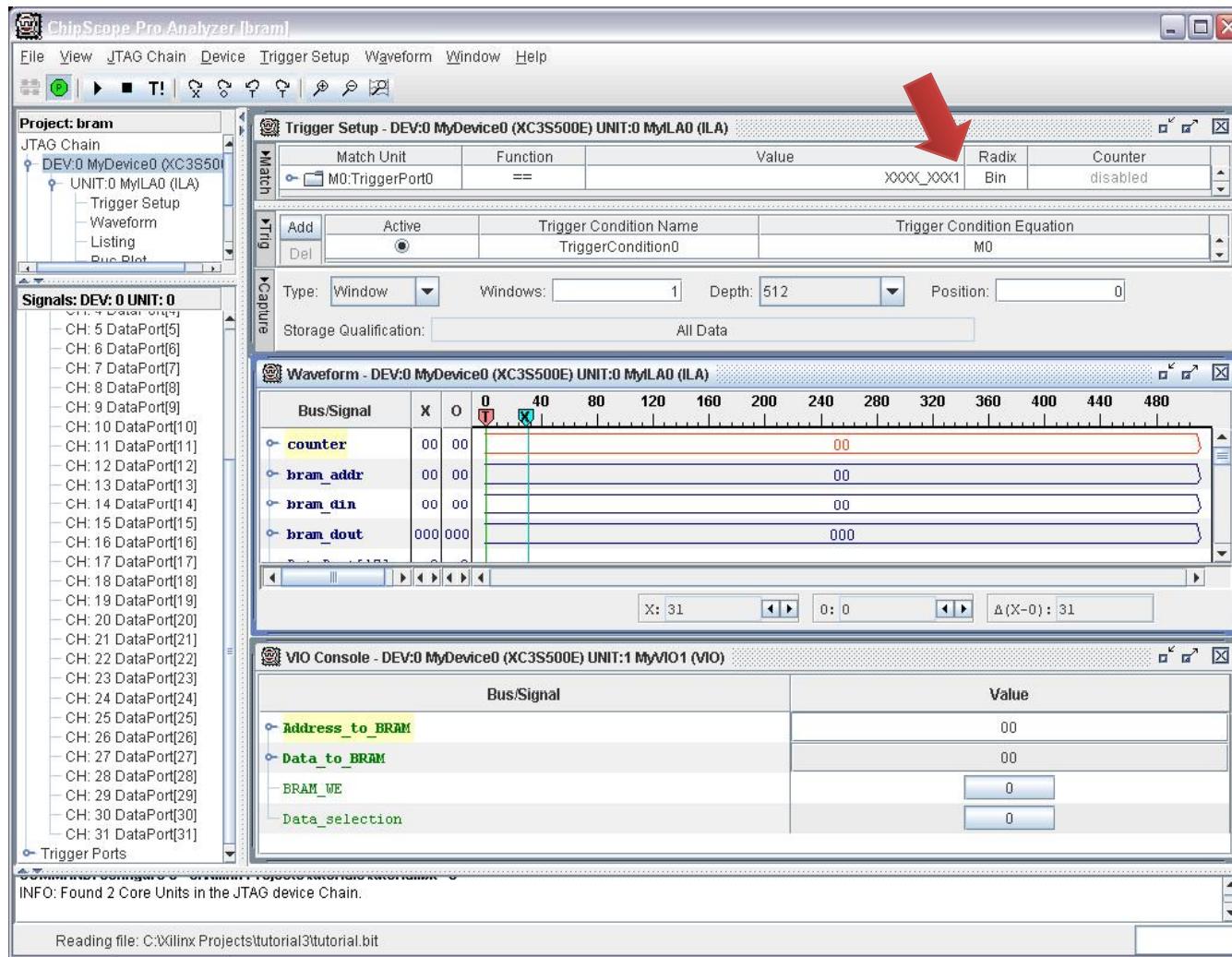
As in the tutorial 2 double click on the **Waveform** and group signals into the busses according to the inputs of the **ILA**.



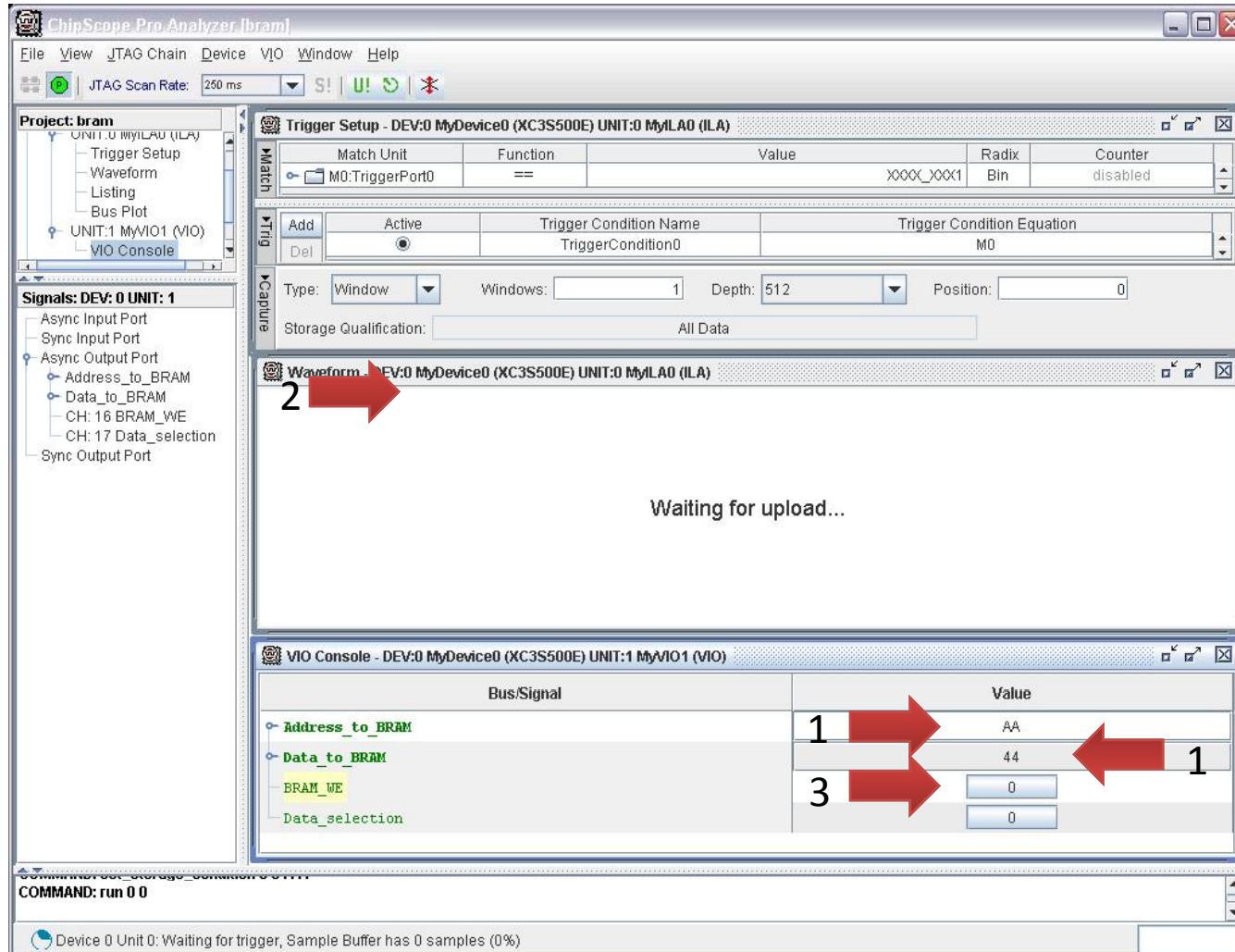
Similarly, double click on the **VIO** Console and group signals into the busses according to the outputs of the **VIO**. Signals can also be renamed as well as inputs of the **ILA**.



In **VIO** signals can also be represented as bus/ push button/ toggle switch. For **BRAM_WE** signal we can change it to a push button by right click on the **BRAM_WE->Type->Push Button->High**. **High** basically means that it would output '1' when pressed.

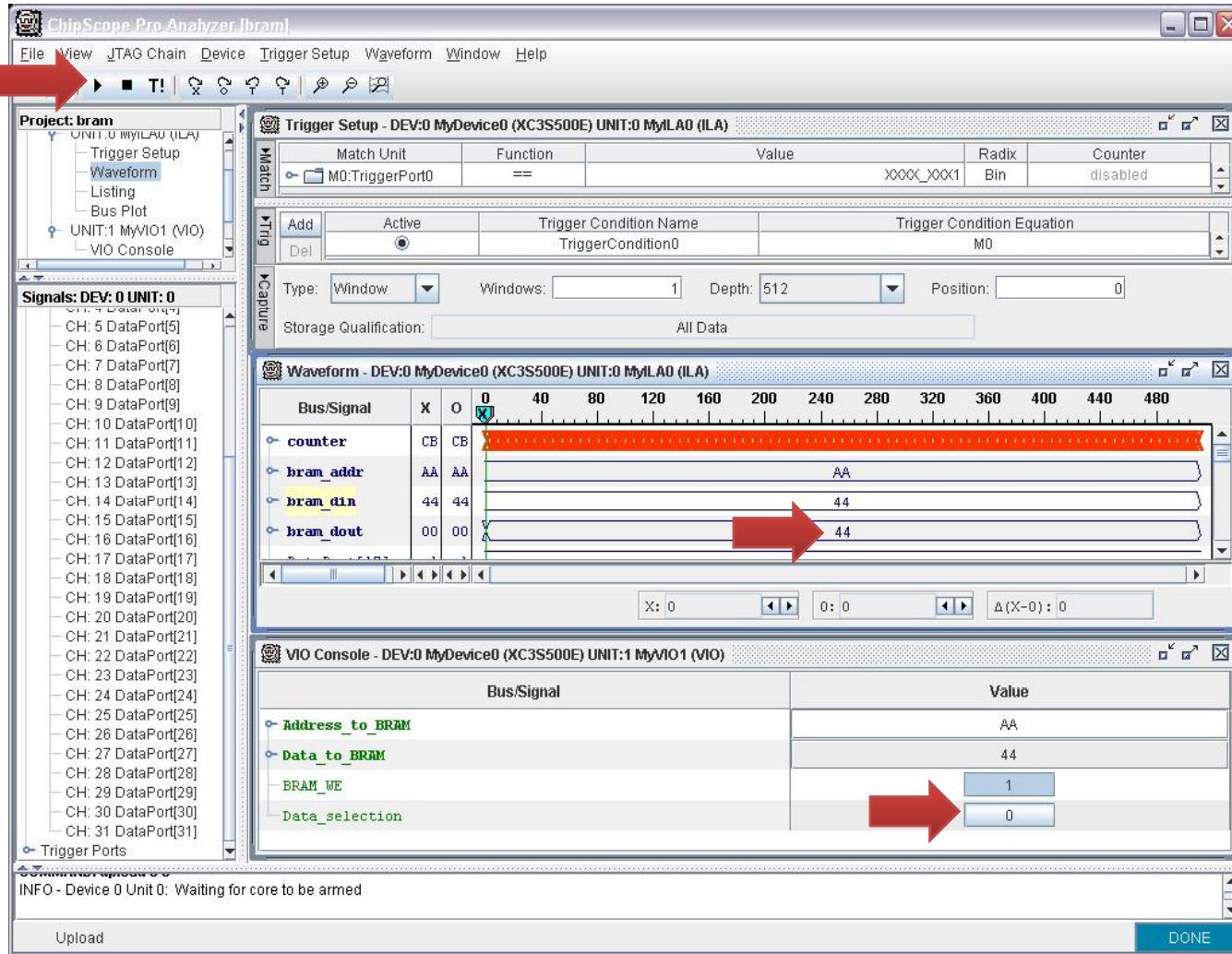


In a trigger window set first bit to '**1**' so that we can trigger the capture when the **Write_enable** is activated.
 After all of the setting arrange the windows so they look something like the figure above.

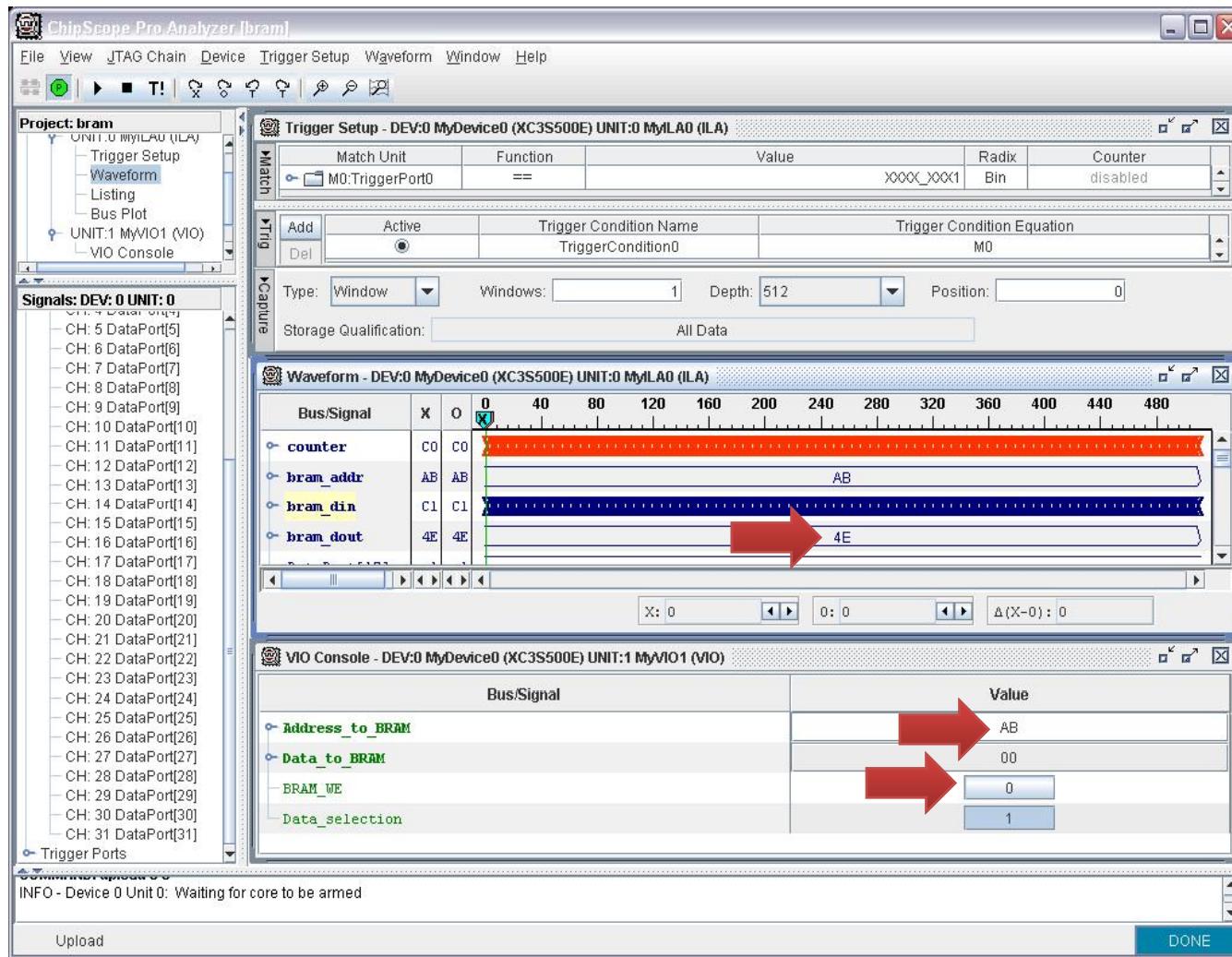


Now lets try to write and read some data to BlockRAM to verify that our BlockRAM works and ChipScope can read and operate it.

1. Set Address to **0xAA** and data to be written **0x44**.
2. Click on the **Waveform** window and press play at the top left side of the GUI.
3. To activate writing click on the **BRAM_WE** button



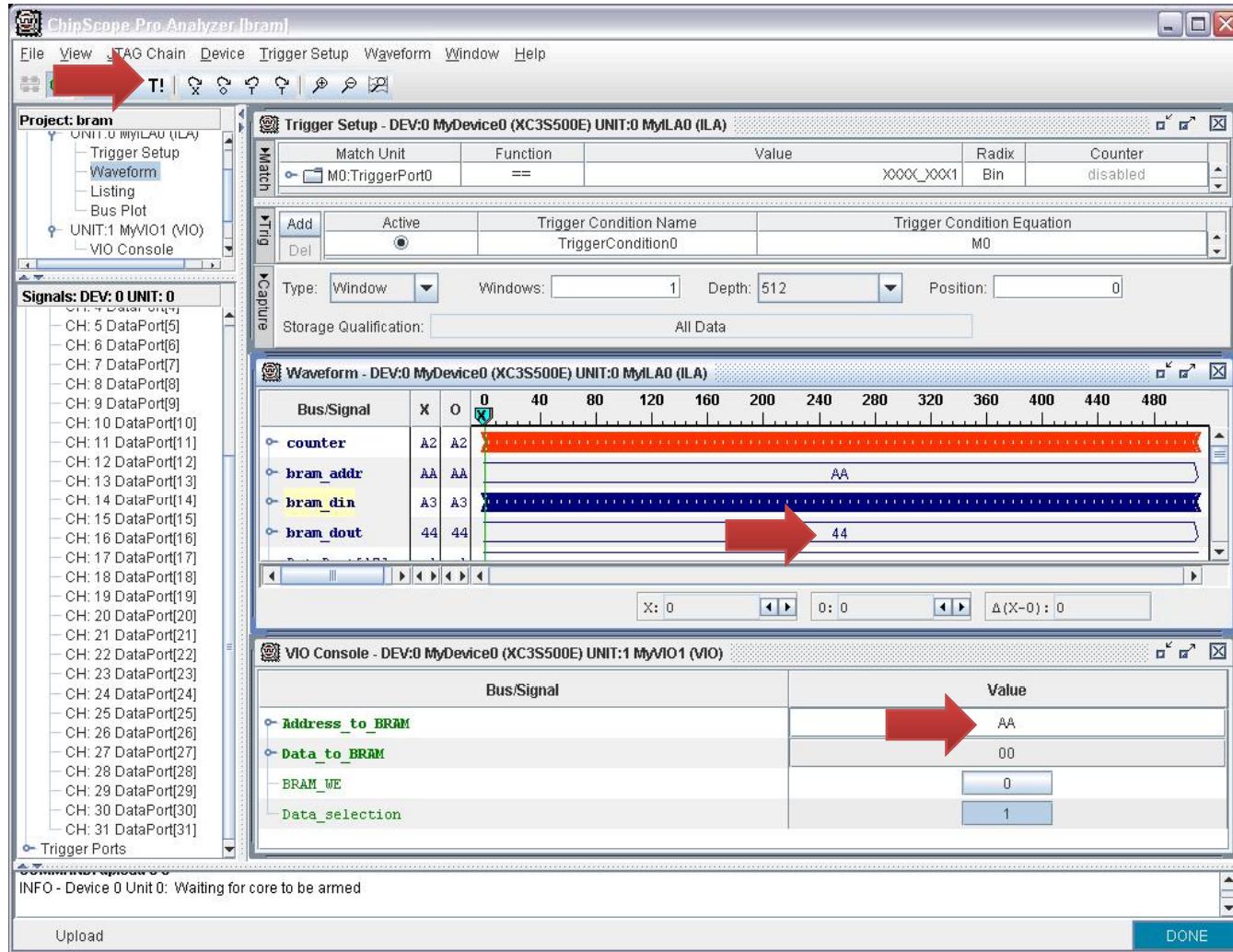
Previous step as expected triggered ChipScope Pro to start capturing, and as it can be seen the data, on BlockRAM has propagated to its output. Next step is to write data counter value to BlockRAM instead of user value. Press on ***Data_selection*** button to switch to counter.



To make sure we do not overwrite the previous data that we wrote change the address to **0xAB** (or anything else but **0xAA**).

By pressing on the **BRAM_WE** counter value that was present at a time of **BRAM_WE** rising edge is written to the **0xAB** location.

As it can be seen the value that was written in the **0xAB** location is **0x4E**



To check if location **0xAA** still contains the value that we wrote several steps ago, simply write address **0xAA** into the **Address_to_BRAM**.

Press **T!** to get immediate value of all of the busses and as you can see **0x44** shows up on the **bram_dout** bus, which proves that the data was successfully saved and not effected by other writes.

Conclusion

This completes tutorial 3 which included:

- Overview of the ChipScope Pro **VIO**.
- Generation of ChipScope Pro **ICON** and **VIO** cores.
- Insertion of ChipScope Pro cores into the project from Tutorial 2.
- Generation of BlockRAM core.
- Insertion of BlockBAM into the project and proper assignment of signals.
- Setting ChipScope Pro environment including **VIO**, **ILA**, and **Trigger** settings
- Writing and reading data to/from BlockRAM
- Capturing data_in/data_out/address signals using **ILA** module and displaying in the **Waveform** window.