Building a Dual-Port Memory

2021.1

Abstract

This lab illustrates how to design a dual-port memory in VHDL. It should take approximately 60 minutes.

Objectives

After completing this lab, you will be able to:

- Demonstrate how memory can be inferred
- Guide the inference to specific types of implementations

Introduction

The wave_gen design, used in this lab, uses a dual-port block RAM memory to store incoming patterns from one side and play these patterns back out on the other side. The wave_gen design requires that the "A" side of the dual-port memory have read and write capabilities, while the "B" side only needs to read from the device.

The wave_gen design contains a dual-port memory that was created using the Block Memory Generator IP. While it is a capable tool, using the output of this tool locks a designer into a single family and the core must be regenerated in order to port the design to other devices. Inferring dual-port memory removes this hindrance.

Conveniently, both distributed and block RAM implementations of dual-port memory inference support a simple dual-port configuration (read and write on one side, read only on the other).

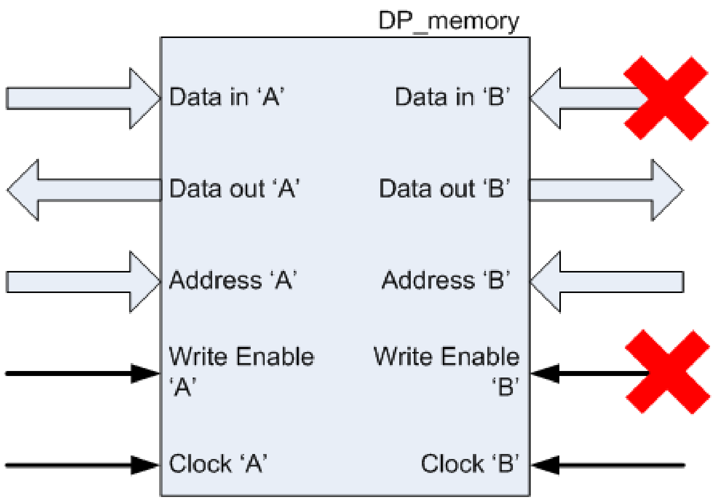


Figure 4-1: Block Memory Block Diagram

The source code that you will write will support the full reading and writing of the DP_memory and will be tested using a pre-written testbench.

You will then comment out the items marked with the 'X' in the figure above and synthesize, using block RAM resources.

Understanding the Lab Environment

The labs and demos provided in this course are designed to run on a Linux platform. Many of the labs and demos can be successfully executed in the Windows environment or a native Linux environment as well.

The instructions found in this lab are expressed using the Linux notation. This includes the forward slash ('/') as the hierarchy separator instead of the Windows backslash ('\'). Students who want to run the labs directly under Windows must use the correct hierarchy separator.

Customizable environment variables enable you to tailor your environment for specific machine configurations. The only environment variable (shown below) used in the customer training environment (CustEd_VM) points to the training directory where all the lab files are located. This reduces the amount of typing you need to do when entering directory paths.

This environment variable can be customized according to your specific location and can be set for Linux systems in the `/etc/profile` file and for Windows systems by entering "env" from the search bar.

The following is the environment variable used in the customer training VM:

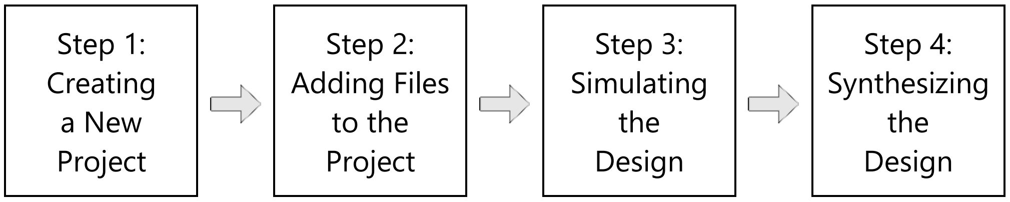
Environment Variable Name	**Description**
`$TRAINING_PATH`	Points to the space allocated for students to work through the labs. This directory includes prebuilt images and starting points for the labs and demos. The customer training VM sets `$TRAINING_PATH` to the `/home/xilinx/training` directory. Typically, Windows users will install the training directory under C: to keep the path names as short as possible.

Note: Environment variables are not supported from the Vitis IDE GUI. When using this tool, you must manually replace **`$TRAINING_PATH`** with the value of the variable, which in the customer training virtual machine, is **`/home/xilinx/training`**. Other tools, such as the Vivado Design Suite, will properly expand the environment variable.

Additional note about environments: Both the Vivado Design Suite and Vitis platform offer a Tcl environment. The contents of this environment are NOT preserved with the project. When the tools launch, they start with a pristine Tcl environment with none of the procs or variables remaining from a previous launch of the tools.

This means that if you sourced a Tcl script or manually set any Tcl variables and you closed the tool, when you reopen the tool (and perhaps even reopen the last project), you will need to source the Tcl script again and set any variables that the lab requires.

General Flow


Step 1: Creating a New Project
Step 2: Adding Files to the Project
Step 3: Simulating the Design
Step 4: Synthesizing the Design

Creating a New Project Step 1

1-1. Launch the Vivado Design Suite.

If you do not recall how to perform this task, refer to the "Launching the Vivado Design Suite" section under Vivado Design Suite Operations in the *Lab Reference Guide*.

1-2. Create a new Vivado Design Suite project named *buildingMemory* and locate it in the following directory.

Browse to the `$TRAINING_PATH/buildingMemory/lab/KCU105` directory.

Target the project for the Kintex UltraScale KCU105 Evaluation Board evaluation board.

If you do not recall how to perform this task, refer to the "Creating a Blank Vivado Design Suite Project" section in the *Lab Reference Guide*.

From the settings in the Flow Navigator, change the target language to VHDL.

Adding Files to the Project Step 2

You will be coding the bulk of this design based on the I/O specifications. The only file that needs to be imported is the testbench and Vivado® simulator wave configuration file.

2-1. Load the DPmemory_tb.vhd testbench file into the project. It has been preloaded into the working directory.

2-1-1. Click **Add Sources** in the Flow Navigator under Project Manager.

The Add Sources Wizard opens.

2-1-2. Select **Add or create simulation sources**.

2-1-3. Click **Next**.

2-1-4. Click the **Plus** (+) icon and select **Add Files** from the location `$TRAINING_PATH/buildingMemory/support`.

2-1-5. Select **DPmemory_tb.vhd** and click **OK**.

2-1-6. Click **Finish** to confirm adding the selected file to the project.

2-2. Create a new HDL source file called *DP_memory*.

2-2-1. Select **Add Sources** in the Flow Navigator under Project Manager.

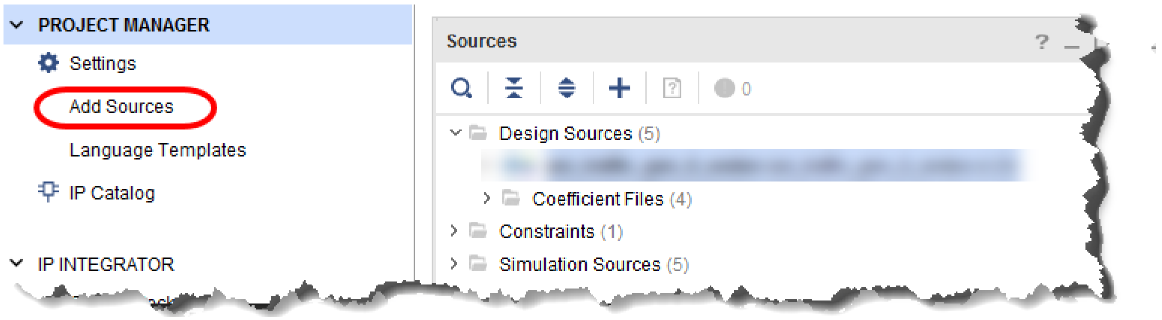


Figure 4-2: Selecting Add Sources

2-2-2. Select **Add or create design sources** (1).

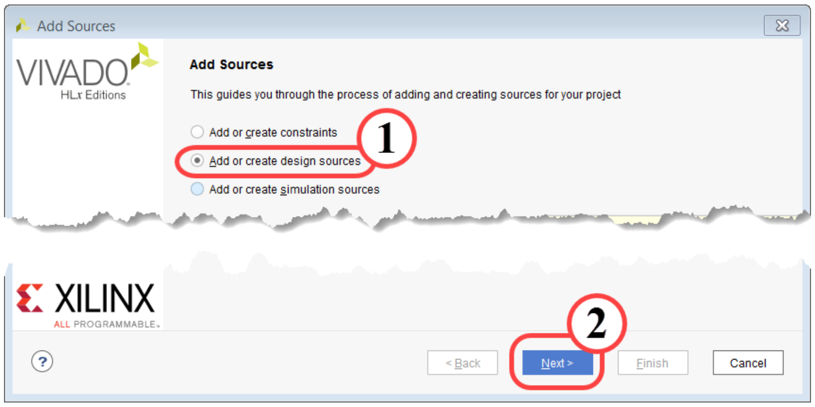

Add Sources
VIVADO
HLx Editions
Add Sources
This guides you through the process of adding and creating sources for your project
Add or create constraints
Add or create design sources
Add or create simulation sources
1
XILINX
ALL PROGRAMMABLE.
2
< Back
Next >
Finish
Cancel

Figure 4-3: Selecting Add or Create Design Sources

2-2-3. Click **Next** (2).

The Add or Create Design Sources dialog box opens.

2-2-4. Click the **Plus** (+) icon and select **Create File**.

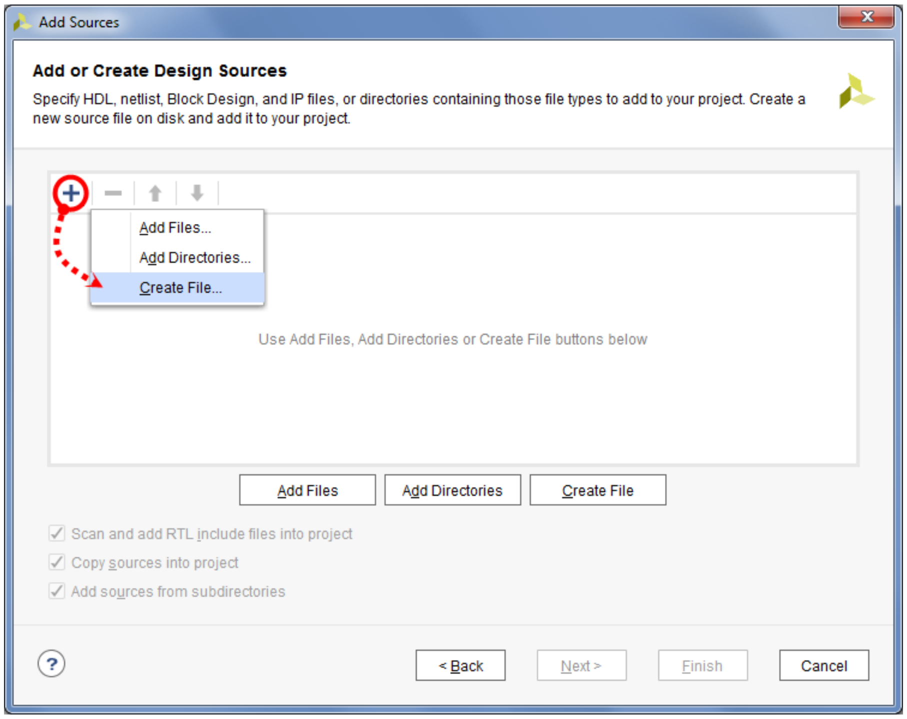
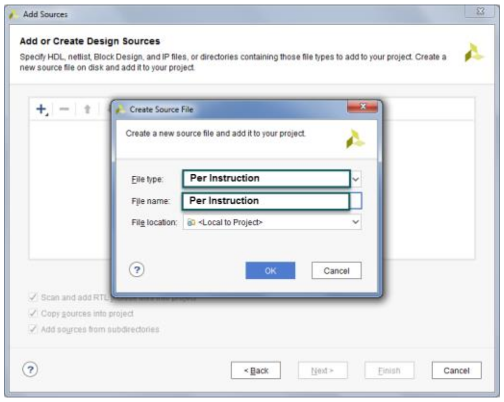

Add Sources
Add or Create Design Sources
Specify HDL, netlist, Block Design, and IP files, or directories containing those file types to add to your project. Create a new source file on disk and add it to your project.
Add Files...
Add Directories...
Create File...
Use Add Files, Add Directories or Create File buttons below
Add Files
Add Directories
Create File
Scan and add RTL include files into project
Copy sources into project
Add sources from subdirectories
< Back
Next >
Finish
Cancel

Figure 4-4: Selecting Create File

The Create Source File dialog box opens.

2-2-5. Select **VHDL** from the File type drop-down list.

2-2-6. Enter **DP_memory** as the file name.



Add Sources

Add or Create Design Sources

Specify HDL, netlist, Block Design, and IP files, or directories containing those file types to add to your project. Create a new source file on disk and add it to your project.

Create Source File

Create a new source file and add it to your project.

File type: Per Instruction

File name: Per Instruction

File location: <Local to Project>

Scan and add RTL

Copy sources into project

Add sources from subdirectories

Figure 4-5: Entering File Name and Type

2-2-7. Click **OK** in the Create Source File dialog box.

2-2-8. Click **Finish** to add the new source file(s).

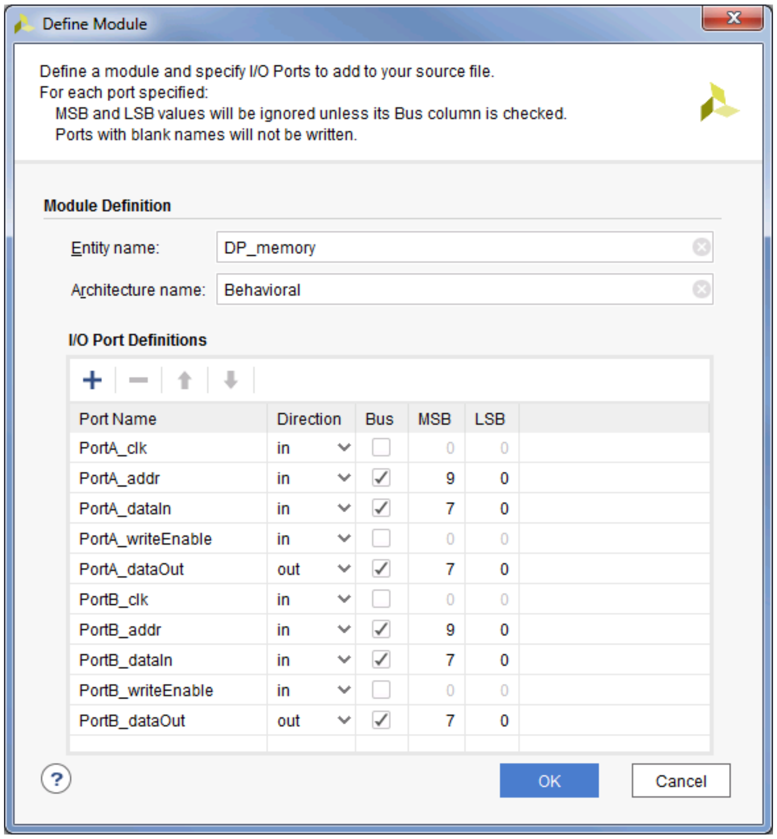
The Define Module dialog box opens.

2-3. Specify the following input and output ports for the DP_Memory module in the Define Module dialog box.

Signal Name	Direction	Type
PortA_clk	In	std_logic
PortA_addr	In	std_logic_vector(9 downto 0)
PortA_dataIn	In	std_logic_vector(7 downto 0)
PortA_writeEnable	In	std_logic
PortA_dataOut	Out	std_logic_vector(7 downto 0)
PortB_clk	In	std_logic
PortB_addr	In	std_logic_vector(9 downto 0)
PortB_dataIn	In	std_logic_vector(7 downto 0)
PortB_writeEnable	In	std_logic
PortB_dataOut	Out	std_logic_vector(7 downto 0)

DP_memory Inputs and Outputs

2-3-1. Add the ports according to the DP_memory Inputs and Outputs table.



Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.

Module Definition

Entity name: DP_memory

Architecture name: Behavioral

I/O Port Definitions

Port Name	Direction	Bus	MSB	LSB
PortA_clk	in	☐	0	0
PortA_addr	in	✓	9	0
PortA_dataIn	in	✓	7	0
PortA_writeEnable	in	☐	0	0
PortA_dataOut	out	✓	7	0
PortB_clk	in	☐	0	0
PortB_addr	in	✓	9	0
PortB_dataIn	in	✓	7	0
PortB_writeEnable	in	☐	0	0
PortB_dataOut	out	✓	7	0

Figure 4-6: Adding Ports to the New Module

2-3-2. Click **OK**.

2-4. Uncomment the IEEE.NUMERIC_STD package in the code.

2-4-1. Double-click the **DP_memory.vhd** file to make the source code visible in the main workspace window.

2-4-2. Remove the **- -** comment symbols before the **use IEEE.NUMERIC_STD.ALL;** on line 27 to include the NUMERIC_STD package.

2-5. Create a memory structure that is 1024 x 8 bits in size and an appropriate shared variable to represent the memory itself.

There are a number of significant side effects that can be caused by the misuse of shared variables; however, as in this case, it is the only way to properly model a dual-port memory. Suffice it to say that shared variables behave exactly like regular variables, but their scope is limited to the module that they reside in rather than just the process.

Although a suggested code solution can be found in the Answers section, you should make every effort to code this yourself.

2-5-1. Add the code (type statement) for a memory structure that is 1024 x 8 bits in size prior to the architecture's "begin" statement and name it memoryByteArray.

2-5-2. Add an appropriate shared variable named memoryBlock for the memory itself immediately after the type statement.

Shared variables are declared with the key words "shared variable" instead of variable or signal and are used to share data between processes. There are a number of considerations to be made when using shared variables.

2-6. Create two processes – one for managing the 'A' side of the memory, the other for the 'B' side of memory. The processes will be identical except for the signals that they will be working on.

The process should be sensitive to the rising edge of the appropriate clock. Assign data_in to the memory structure when the write enable is asserted, and always present the data referenced by the address to the data output.

Although suggested code solutions can be found in the Answers section, you should make every effort to code these yourself.

2-6-1. Construct a process that is sensitive to the rising edge of the PortA_clk immediately following the "begin" statement of the architecture.

Because VHDL uses integers as the index into arrays, the address must be converted from std_logic_vector to an integer.

2-6-2. Create a temporary variable named "address" as an integer of the proper size prior to the process's "begin" statement to act as an index in the 1024x8 memory.

2-6-3. Perform the conversion on the std_logic_vector version of the address into the variable address within the clocked process.

2-6-4. Add the appropriate code within the synchronous portion of the process so that the memory must always present the contents pointed to by the address on the data_out port.

The last step is to provide the write capability to the process.

2-6-5. Add code that tests the enable to determine if the write should occur, and if it should, perform the write operation.

2-6-6. Repeat this block of code, but for the 'B' side of the memory in another process.

2-6-7. Select **File** > **Text Editor** > **Save File**.

The final code should appear as in the Answers section.

2-7. Synthesize the design. Correct any errors.

2-7-1. Click **Run Synthesis** in the Flow Navigator and click **OK**.

2-7-2. Correct any errors that have been found and resynthesize.

2-7-3. Click **Cancel** when the Synthesis Completed dialog box appears.

Simulating the Design Step 3

This portion of this lab requires you to load a predefined configuration for the simulation environment, run the simulation, and view the waveform.

3-1. Launch the provided testbench and note the order and appearance of the signals.

3-1-1. Select **Simulation** > **Run Simulation** in the Flow Navigator.

3-1-2. Select **Run Behavioral Simulation**.

This will launch the Vivado simulator and run for the default 1 µs. Note the order and radices of the signals.

3-2. Load the provided configuration file DPmemory.wcfg in the Vivado simulator.

3-2-1. Select **File** > **Simulation Waveform** > **Open Configuration**.

3-2-2. Select **DPmemory.wcfg** in the `$TRAINING_PATH/buildingMemory/support` directory and click **OK**.

Question 1

What changed when the configuration file was loaded?

The time scale and port arrangement changed.

3-3. Examine the waveform, review the testbench, and answer the questions below.

Question 2

What occurs during the first 50 ns of the simulation?

A side write enable goes low, so datain and dataout remains the same. B side enable continues for

another 5ns. The address and datain stays the same look at this question again. I misread it

Question 3

What occurs at approximately 555 ns and why is this done?

Write operation occurs on Bside due to the write enable.

Question 4

What occurs after 570 ns and what does this show?

After 570ns, the address starts over from 004 to 000 and increments by 1 afterwards

3-4. Exit the Vivado simulator.

3-4-1. Select **File** > **Close Simulation**.

3-4-2. Click **OK** to close the simulation.

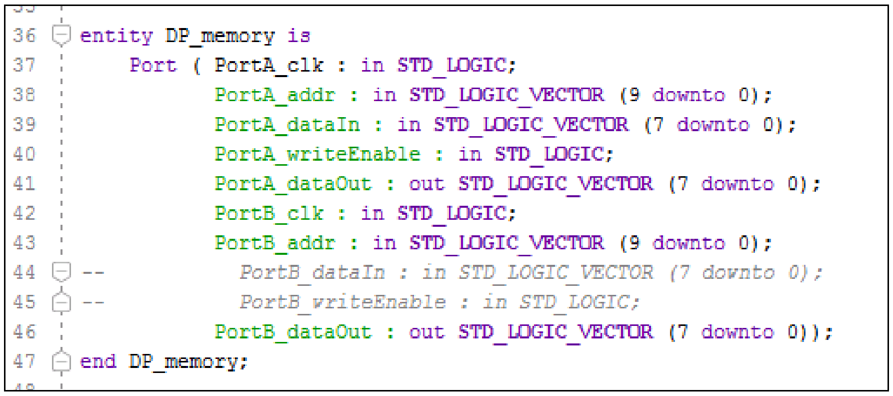
Synthesizing the Design Step 4

Now that you know that the code is operating as expected, you will remove the write capability from the 'B' side so that the functionality for the wave_gen design is met.

4-1. Return to the source file and edit the DP_memory.vhd file so that all references to writing to the Port B side of block RAM is removed. This will implement as a simple dual-port memory rather than a true dual-port memory.

Use comments rather than deleting the lines.

4-1-1. Comment out PortB_writeEnable and PortB_dataIn from the port declaration in the entity statement.

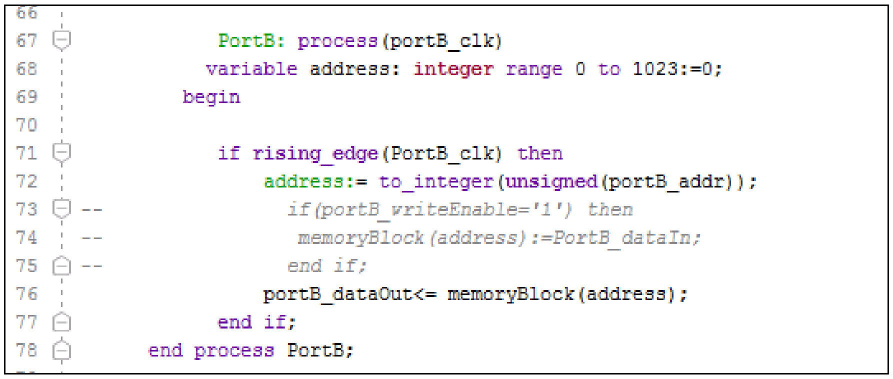


```
entity DP_memory is
    Port ( PortA_clk : in STD_LOGIC;
           PortA_addr : in STD_LOGIC_VECTOR (9 downto 0);
           PortA_dataIn : in STD_LOGIC_VECTOR (7 downto 0);
           PortA_writeEnable : in STD_LOGIC;
           PortA_dataOut : out STD_LOGIC_VECTOR (7 downto 0);
           PortB_clk : in STD_LOGIC;
           PortB_addr : in STD_LOGIC_VECTOR (9 downto 0);
--           PortB_dataIn : in STD_LOGIC_VECTOR (7 downto 0);
--           PortB_writeEnable : in STD_LOGIC;
           PortB_dataOut : out STD_LOGIC_VECTOR (7 downto 0));
end DP_memory;
```

Figure 4-7: Port B Write Capability Commented Out of the Entity

4-1-2. Remove the references for PortB_writeEnable and PortB_dataIn from the PortB process.

To save time, highlight the block of code and use the block comment capability by clicking the // icon.



```
         PortB: process(portB_clk)
        variable address: integer range 0 to 1023:=0;
      begin

         if rising_edge(PortB_clk) then
             address:= to_integer(unsigned(portB_addr));
--             if(portB_writeEnable='1') then
--              memoryBlock(address):=PortB_dataIn;
--             end if;
             portB_dataOut<= memoryBlock(address);
         end if;
      end process PortB;
```

Figure 4-8: Port B Write Capability Commented Out of the Process

4-1-3. Select **File** > **Text Editor** > **Save File**.

4-2. Run synthesis.

4-2-1. Click **Run Synthesis** in the Flow Navigator under Synthesis.

Alternatively, you can also select **Flow** > **Run Synthesis** or press <**F11**>.

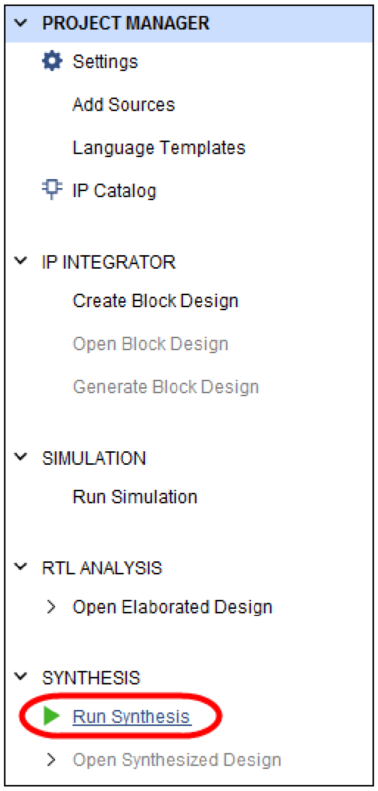
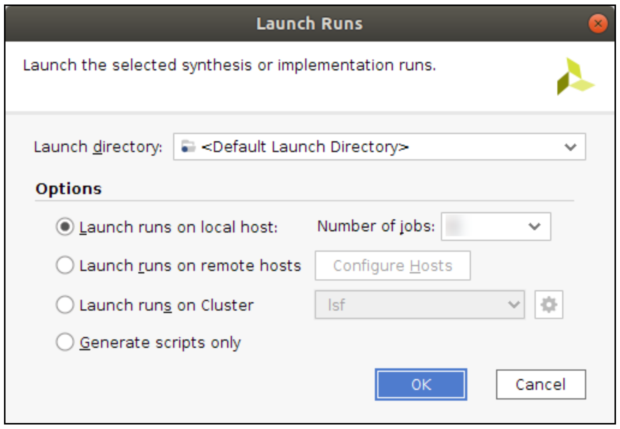


Figure 4-9: Selecting Run Synthesis

The Launch Runs dialog box opens.



Launch Runs

Launch the selected synthesis or implementation runs.

Launch directory: <Default Launch Directory>

Options

(●) Launch runs on local host: Number of jobs:

( ) Launch runs on remote hosts Configure Hosts

( ) Launch runs on Cluster lsf

( ) Generate scripts only

Figure 4-10: Setting the Launch Run Configuration

Hint: When launching the runs on a local host, it is common to set the number of jobs to the maximum value as this recruits the largest number of processors for the task and typically results in the least amount of time spent in synthesis.

4-2-2. Click **OK** to launch the runs.

4-2-3. Click **Save** if you are asked to save your files.

Once synthesis completes, you are asked what task you want to perform next: run implementation, open the synthesized design, view reports, or none of the above.

4-2-4. Select **Open Synthesis Design** (1).

4-2-5. [Optional] If you are familiar with accessing these various capabilities, you can disable this dialog box from appearing again by selecting **Don't show this dialog again** (2).

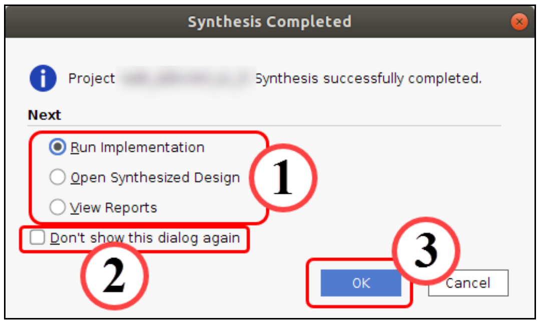


Figure 4-11: Selecting After Synthesis Options

4-2-6. Click **OK** to take the action you just selected (3) or click **Cancel** to simply close the dialog box.

4-3. Generate a Utilization report.

4-3-1. Click **Report Utilization** under Synthesized Design in the Flow Navigator.

Alternatively, you can select **Reports** > **Report Utilization**.

4-3-2. Click **OK** to accept the default name for the utilization report.

4-3-3. View the Utilization Report that appears at the bottom of the Vivado IDE and complete the Utilization Count column in the Resources table below.

Question 5

Complete the table below.

Resource	Utilization Count
Number of Slice Registers	8
Number of Slice LUTs	8
Number of Block RAMs	1
Number of Bonded IOBs	47

Resources Table

4-4. Close the project.

4-5. Close the Vivado Design Suite.

Some systems (particularly VMs) may be memory constrained. Removing the workspace frees a portion of the disk space, allowing other labs to be performed.

You can delete the directory containing the lab you just ran by using the graphical interface or the command line interface. You can choose either mechanism. Both processes will recursively delete all the files in the `$TRAINING_PATH/buildingMemory` directory.

4-6. [Optional] [Only for local VMs—not for CloudShare] Clean up the file system.

Using the GUI:

4-6-1. Using the graphical browser (Windows: press the <**Windows**> key + <**E**>; Linux: press <**Ctrl** + **N**>), navigate to `$TRAINING_PATH/buildingMemory`.

4-6-2. Select **`buildingMemory`**.

4-6-3. Press <**Delete**>.

-- OR --

Using the command line:

4-6-4. Open a terminal window (Windows: press the <**Windows**> key + <**R**>, then enter **cmd**; Linux: press <**Ctrl** + **Alt** + **T**>).

4-6-5. Enter the following command to delete the contents of the workspace:

[Windows users]: **`rd /s /q $TRAINING_PATH/buildingMemory`**

[Linux users]: **`rm -rf $TRAINING_PATH/buildingMemory`**

Summary

This lab showed you how to create and simulate a dual-port RAM. After the design was "proven," you removed the B-side write access for two reasons: the design did not require it, and a full dual-port memory can only be generated with a block RAM.

Answers

Answers listed represent sample solutions only. Your results may differ depending on the version of the software, service pack, or operating system that you are using.

1. What changed when the configuration file was loaded?

   The order of the signals was re-arranged so that all the Port A signals are collected together and all the Port B signals are collected together. Additionally, the radix changed on all of the std_logic_vector signals to hexadecimal for ease of reading and two dividers appeared to help delineate which signals belong to Port A and which to Port B.

2. What occurs during the first 50 ns of the simulation?

   Port A is being exercised by writing 0 to address 0, 11 hex to address 1, 22 hex to address 2, and so on, up to and including address 4.

   Simultaneously, a similar process is occurring on Port B. 44 is being written to address 4 and so on, up to address 8.

3. What occurs at approximately 555 ns and why is this done?

   Via Port B, the value of EE hex over-writes address 4. This is done to see if this value appears when the Port A side is read, thus proving that the memory is properly shared.

4. What occurs after 570 ns and what does this show?

   After 570 ns, both Ports A and B begin cycling through the lowest addresses to see if the values are properly stored and re-read.

5. Complete the table below.

Resource	Utilization Count
Number of Slice Registers	0
Number of Slice LUTs	0
Number of Block RAMs	1
Number of Bonded IOBs	47

Resources Table

Note: To get the block RAM utilization count, click the Block RAM Tile under BLOCKRAM in the left pane of utilization report and choose DP_memory. The count will be shown in the Netlist Properties window.

Coding the Memory Structure

Memory structure that is 1024 x 8 bits in size and an appropriate shared variable for the memory itself:

```
type memoryByteArray is array(0 to 1023) of std_logic_vector(7 downto 0);

shared variable memoryBlock : memoryByteArray := (others=>(others=>'0'));
```

Process that is sensitive to the rising edge of the PortA_clk:

```
PortA: process (PortA_clk)
    begin
        if rising_edge(PortA_clk) then
        end if;
    end process PortA;
```

Variable named "address" as an integer of the proper size to act as an index into the 1024x8 memory:

```
variable address : integer range 0 to 1023 := 0;
```

Performing the conversion on the address:

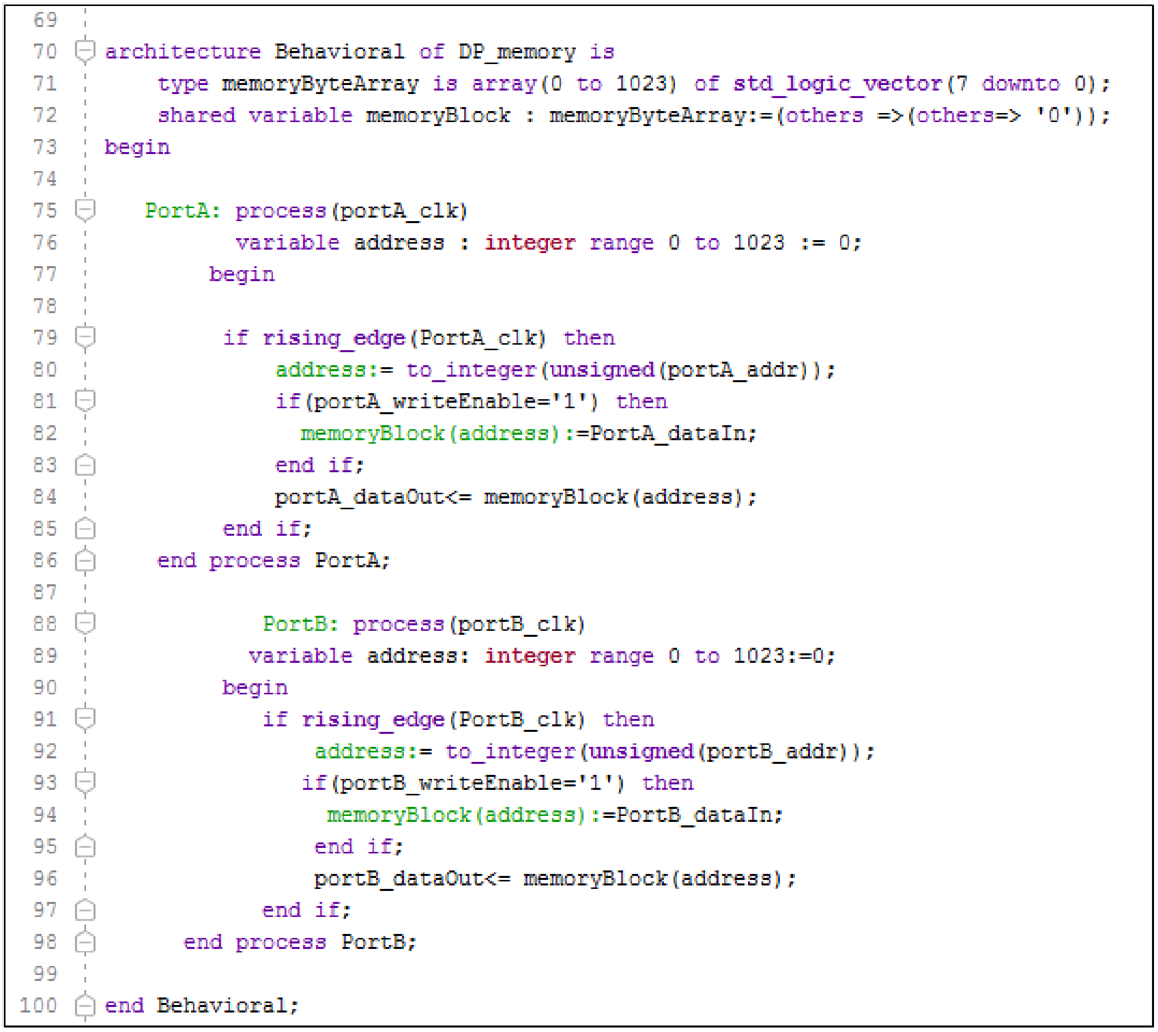
```
address := to_integer(unsigned(PortA_addr));
```

Presenting the contents pointed to by the address on the data_out port:

```
PortA_dataOut <= memoryBlock(address);
```

Testing the enable to determine if the write should occur, and if it should, performing the write operation:

```
if (PortA_writeEnable = '1') then
   memoryBlock(address) := PortA_dataIn;
end if;
```



```
architecture Behavioral of DP_memory is
    type memoryByteArray is array(0 to 1023) of std_logic_vector(7 downto 0);
    shared variable memoryBlock : memoryByteArray:=(others =>(others=> '0'));
begin

   PortA: process(portA_clk)
          variable address : integer range 0 to 1023 := 0;
        begin

         if rising_edge(PortA_clk) then
             address:= to_integer(unsigned(portA_addr));
             if(portA_writeEnable='1') then
               memoryBlock(address):=PortA_dataIn;
             end if;
             portA_dataOut<= memoryBlock(address);
         end if;
    end process PortA;

            PortB: process(portB_clk)
           variable address: integer range 0 to 1023:=0;
         begin
            if rising_edge(PortB_clk) then
                address:= to_integer(unsigned(portB_addr));
               if(portB_writeEnable='1') then
                 memoryBlock(address):=PortB_dataIn;
                end if;
                portB_dataOut<= memoryBlock(address);
            end if;
      end process PortB;

end Behavioral;
```

Figure 4-12: Architecture Part for Complete Dual-Port Memory