Laboratory 6 – Embedded Memory

Objective: In this laboratory, you will gain experience in using the memory resources available on your FPGA board.

The MAX 10 FPGA on the DE10-Lite contains dedicated memory resources called M9K blocks. Each M9K block contains 9216 memory bits. M9K blocks can be configured to implement memories of various sizes. A common term used to specify the size of a memory is its aspect ratio, which gives the depth in words and the width in bits (depth x width). In this laboratory, we will use an aspect ratio that is four bits wide, and we will use only the first 32 words in the memory.

A diagram of the random access memory (RAM) module that you will implement is shown in Figure 1a. It contains 32 four-bit words (rows), which are accessed using a five-bit address port, a four-bit data port, and a write control. There are two important features of the M9K blocks that have to be mentioned. First, they include registers that can be used to synchronize all of the input and output signals to a clock input. The registers on the input ports must always be used, and the registers on the output ports are optional. Second, the blocks have separate ports for data being written to the memory and data being read from the memory (Figure 1).

Given these requirements, we will implement the 32 x 4 RAM module shown in Figure 1. It includes registers for the **address**, **data input**, and **write ports**, and uses a separate unregistered **data output** port. The example that we will first see uses a shared address bus for read and write operations. This means that only sequential read and write operations are supported. When the write signal is asserted (i.e. active), the value to be written is placed on the DataIn bus and the address of the memory location where the value will be written is placed on the address port. When the write signal is not asserted (i.e. inactive), we have a read operation of the memory location whose address is provided on the address port, with the read value provided on the DataOut bus.

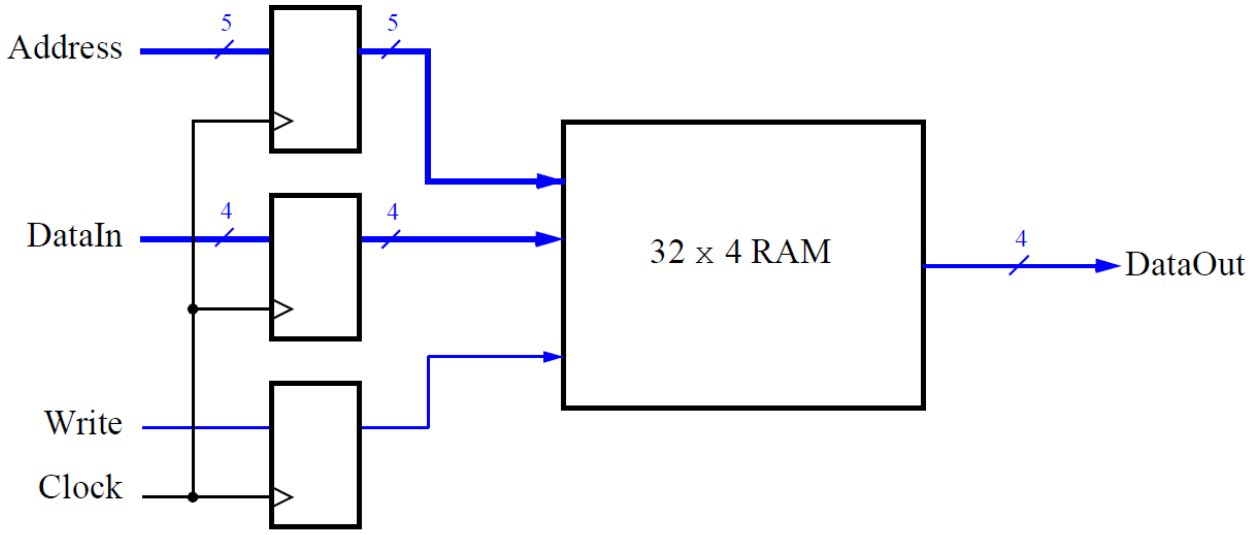

Figure 1: A 32 x 4 RAM module.

Part 1

Commonly used logic structures, such as adders, registers, counters and memories, can be implemented in an FPGA chip by using prebuilt modules that are provided in libraries. In part 1, you will use such a module to implement the memory shown in Figure 1.

1. Create a new Quartus project named *Part1* to implement the memory module. Make sure you specify your target device **10M50DAF484C7G**. Open the IP Catalog in the Quartus software click on **Tools > IP Catalog**. In the **IP Catalog** window choose the **RAM: 1-PORT** module, which is found under the **Basic Functions > On Chip Memory** category.

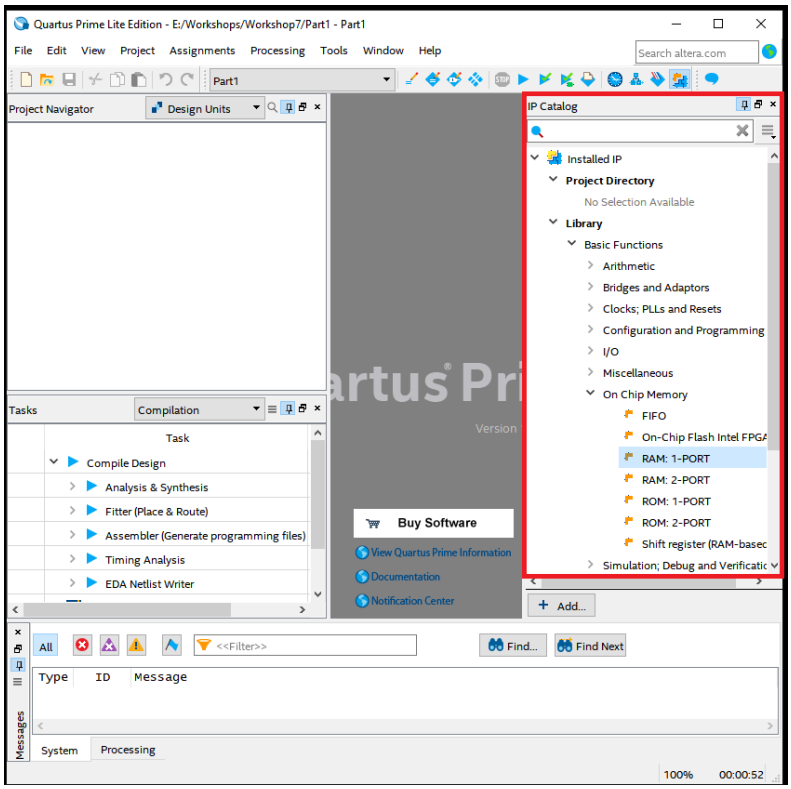

Figure 2: IP Catalog Window.

2. In the **Save IP Variation** window, select **VHDL** as the type of output file to create, give the file the name **ram32x4.vhd**, and click **OK**.

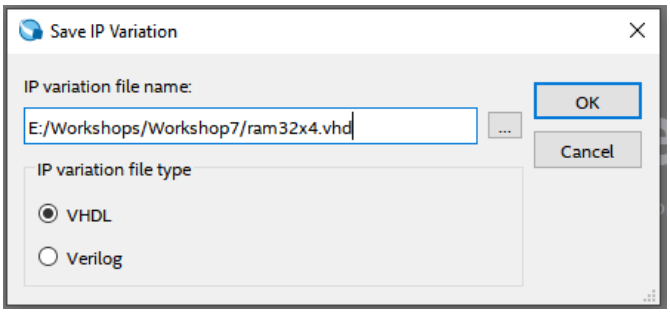

Figure 3: IP Variation Window.

3. As shown in Figure 4 specify a memory size of 32 four-bit words and select M9K. Also on this screen accept the default setting to use a single clock for the memory's registers.

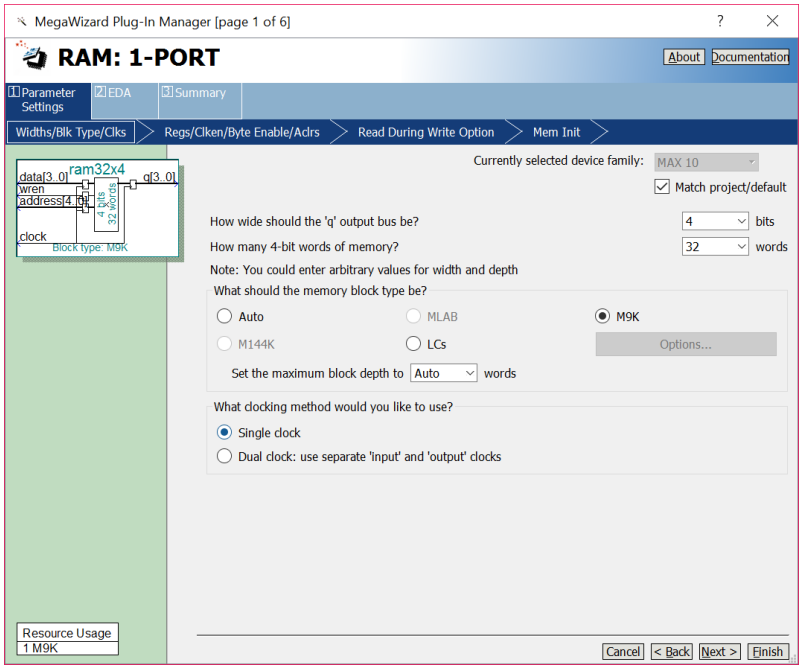

Figure 4: Configuring the size of the memory module.

4. Advance to the page shown in Figure 5 , and deselect the setting called **'q' output** port under the category **Which ports should be registered?** This setting creates a RAM module that matches the structure in Figure 1, with registered input ports and unregistered output ports.

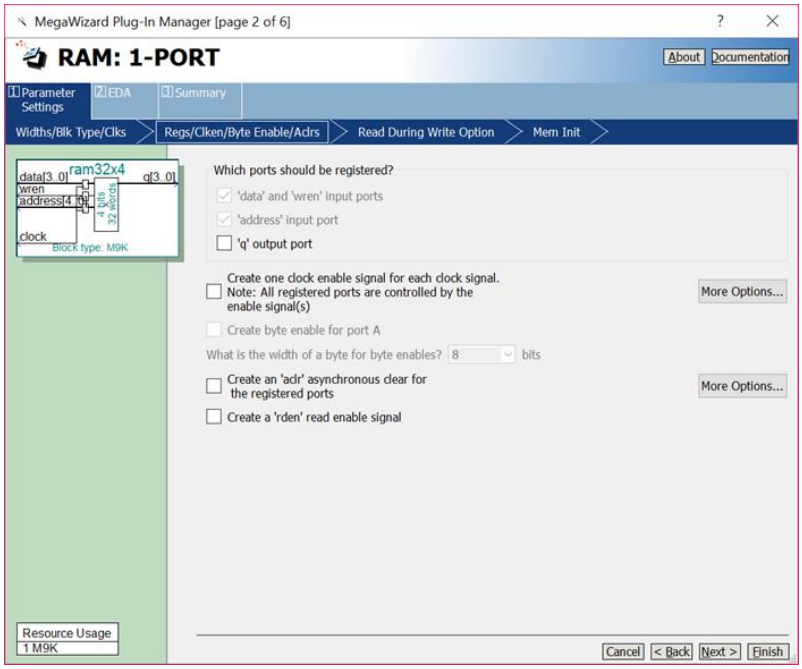

Figure 5: Configuring input and output ports.

5. **Accept** defaults for the rest of the settings in the Wizard, and click the **Finish button** to exit from this tool and click Yes in the following window:

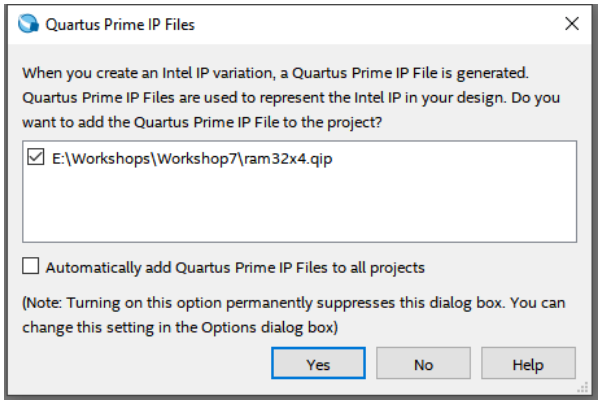

Figure 6: Quartus Prime IP Files Window.

6. Examine the ram32x4.vhd VHDL file by going to the project navigator (Figure 7) and clicking on ram32x4.vhd. You will see the entity shown in Figure 8.
7.

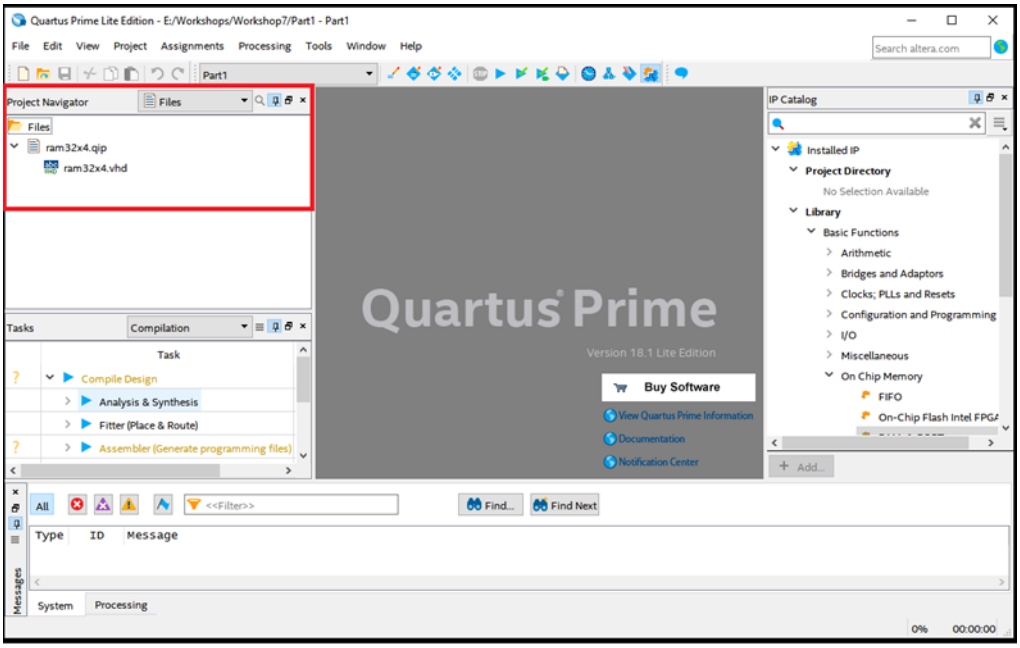

Figure 7: Project Navigator Window.

```
ENTITY ram32x4 IS
   PORT
   (
      address      : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
      clock     : IN STD_LOGIC  := '1';
      data      : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
      wren      : IN STD_LOGIC ;
      q      : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
   );
END ram32x4;
```

Figure 8: Generated entity for 1-PORT RAM

8. We will now instantiate this component in a top-level VHDL file named *Part1.vhd* that includes appropriate input and output signals for the memory ports given in Figure 1. This has been readily done for you. Download the file from the unit webpage and understand the code.

9. Compile the circuit. Observe in the Compilation Report that the Quartus Compiler uses 128 bits in one of the FPGA memory blocks to implement the RAM circuit.

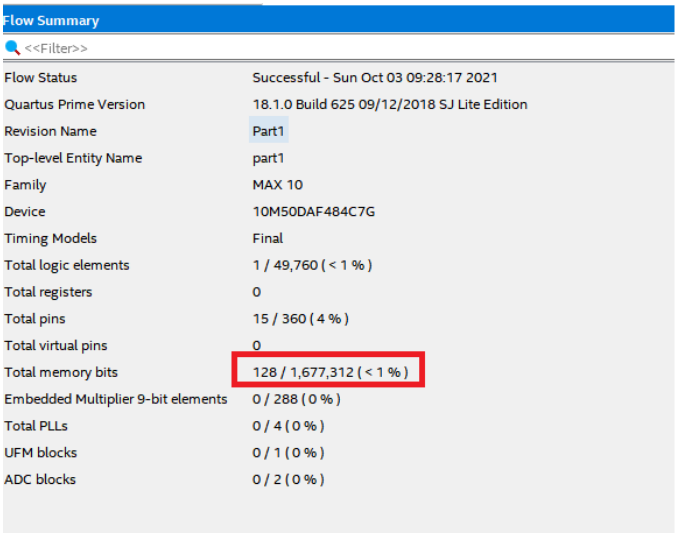

Figure 9: Compilation report

10. Using Modelsim, simulate the behavior of your circuit and ensure that you can read and write data in the memory. An example of simulation output is given in Figure 10. Note that both read and write operations are triggered at the rising clock edges for the M9K embedded memories.

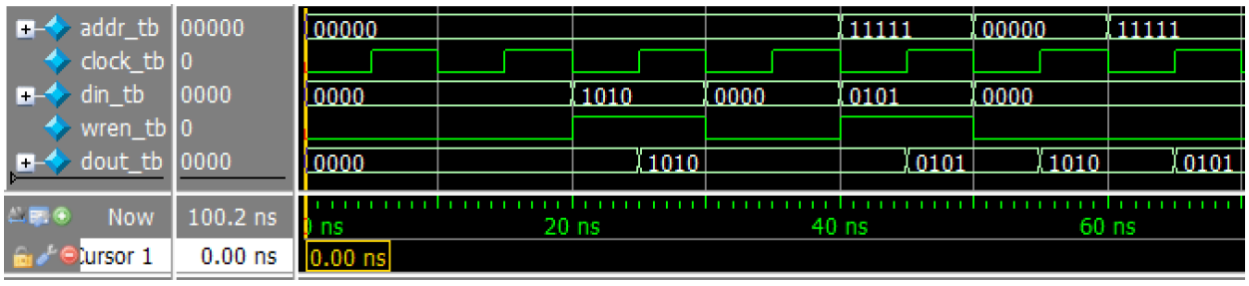

Figure 10: An example of simulation output.

Part 2

Now, we want to realize the memory circuit in the FPGA board, and use slide switches to load some data into the created memory. We also want to display the contents of the RAM on the 7-segment displays.

1. Make a new Quartus project which will be used to implement the desired circuit on your FPGA board.

2. Create another VHDL file Part 2 that instantiates the ram32x4 module and that includes the required input and output pins on your FPGA board. Use slide switches SW3−0 to provide input data for the RAM, and use switches SW8−4 to specify the address. Use SW9 as the **Write** signal and use **KEY0** as the **Clock** input. Show the address value on the 7-segment displays **HEX5-HEX4**, show the data being input to the memory on **HEX2**, and show the data read out of the memory on **HEX0**.

3. Test your circuit and make sure that data can be stored into the memory at various locations.

Part 3

Instead of creating a memory module using the IP Catalog, we can implement the required memory by writing behavioral VHDL that will model the memory. Although the VHDL compiler can infer the memory, one has to carefully construct VHDL code for the compiler to do this. It is thus necessary to refer to the relevant Quartus documentation to have examples on how memory should be specified. Fortunately, templates of inferred memories can be accessed by doing the following:

1. Click **Insert Template** on the **Edit menu** with a file open in the **Quartus II Text Editor**. In the **Insert Template dialog box**, click the **+** icon to expand the navigation tree. You can see that you have the VHDL templates of a range of memories.

In a VHDL-specified design, it is possible to define the memory as a multidimensional array. A 32 × 4 array, which has 32 words with 4 bits per word, can be declared by the statement.

```
TYPE mem IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL memory_array : mem;
```

In an FPGA, such an array can be implemented either by using the flip-flops that each logic element contains or, more efficiently, by using the built-in memory blocks.

For part 3, you will not have to write your own code but only to perform the following steps:

2. Create a new project Part 3, which will be used to implement the desired circuit on your board.

3. Download the VHDL file Part3.vhd that provides the same functionality than in Part2 but for a behavioral modelling of the memory. Take the time to examine the code given by Quartus.

4. Compile the circuit and download it into the FPGA chip.

5. Test the functionality of your design by applying some inputs and observing the output.

Part 4

The SRAM block in Figure 1 has a single port that provides the address for both read and write operations. This means that the SRAM only supports sequential read and write operations. For this part, you will create a different type of memory module, in which there is one port for supplying the address for a read operation, and a separate port that gives the address for a write operation. This dual port RAM supports simultaneous read and write operations.

Perform the following steps:

1. Create a new Quartus project for your circuit. To generate the desired memory module open the IP Catalog and select the RAM: 2-PORT module in the **Basic Functions > On Chip Memory** category. As shown in Figure 11, choose **With one read port and one write port** in the category called **How will you be using the dual port ram?**

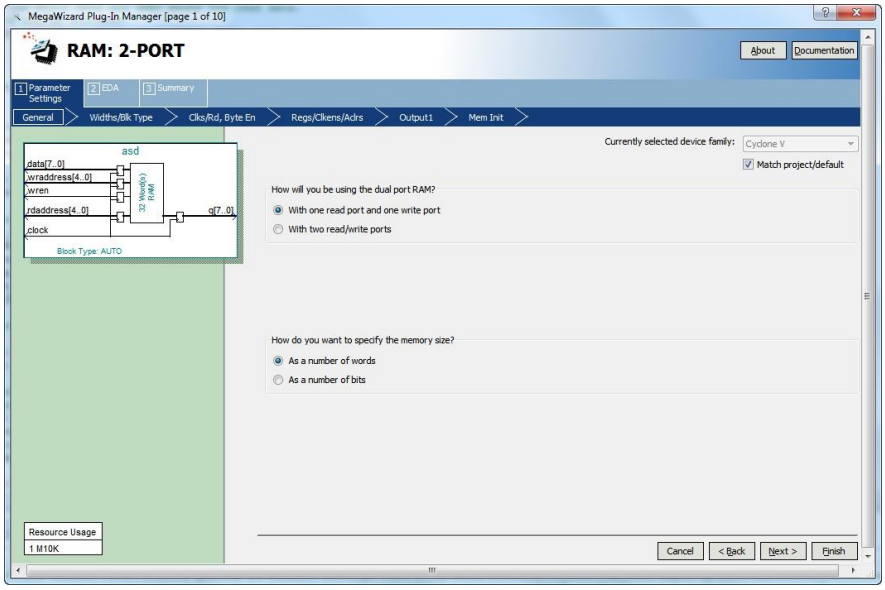

Figure 11: Configuring the two input ports of the RAM.

2. Configure the memory size, clocking method, and registered ports the same way as Part 2. As shown in Figure 12, select I do not care (The outputs will be undefined) for Mixed Port Read-During-Write for Single Input Clock RAM. This setting specifies that it does not matter whether the memory outputs the new data being written, or the old data previously stored, in the case that the write and read addresses are the same during a write operation.

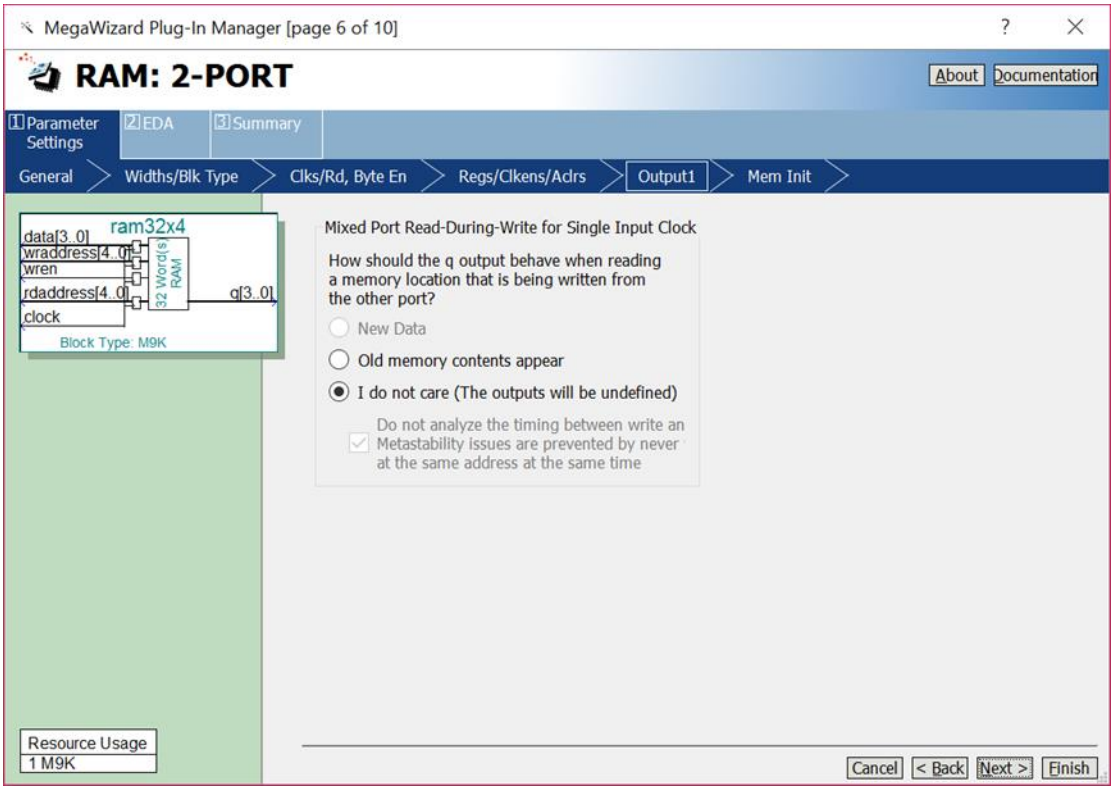

Figure 12: Configuring the output of the RAM when reading and writing to the same address.

3. In this part, you will make use of a feature that allows the memory module to be loaded with data when the circuit is programmed into the FPGA chip. This can be done by initializing the memory using the contents of a memory initialization file (MIF). As shown in Figure 13, choose the setting **Yes, use this file for the memory content data**, and specify the **filename** ram32x4.mif. An example of a MIF file is provided in Figure 14. The file has to be created in the folder that contains the Quartus project. More details about the format of a memory initialization file (MIF) is given in Appendix A. **In this laboratory, you only need to download the ram32x4.mif available on the unit webpage**. **Finish** the Wizard and then examine the generated memory module in the file ram32x4.vhd.

4. Write a VHDL file that instantiates your dual-port memory. To see the RAM contents, add to your design a capability to display the content of each four-bit word (in hexadecimal format) on the 7-segment display HEX0. Use a counter as a read address, and scroll through the memory locations by displaying each word for about one second. As each word is being displayed, show its address (in hex format) on the 7-segment displays HEX3−2. Use the 50 MHz clock, CLOCK_50, and use KEY0 as a reset input. For the write address and corresponding data use switches SW8−4 and SW3−0. Show the write address on HEX5 4 and show the write data on HEX1. Make sure that you properly synchronize the slide switch inputs to the 50 MHz clock signal.

5. Test your circuit and verify that the initial contents of the memory match your ram32x4.mif file.

Make sure that you can independently write data to any address by using the slide switches.

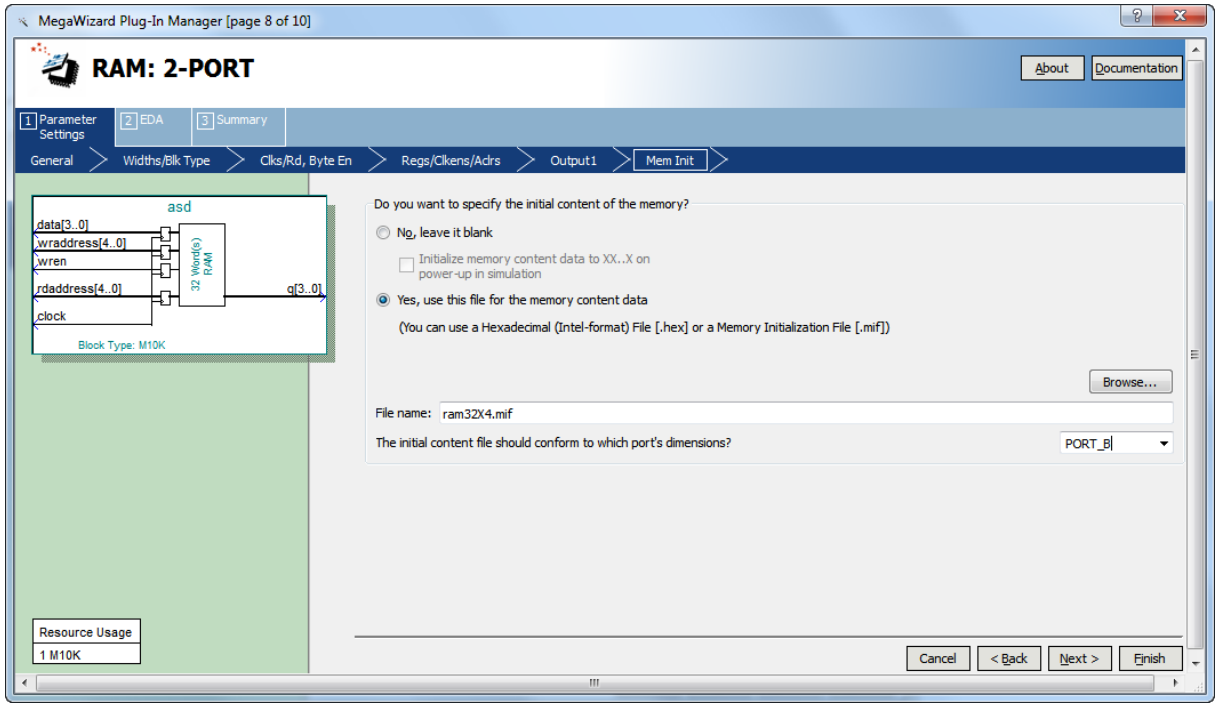

Figure 13: Specifying a memory initialization file (MIF).

```
DEPTH = 32;
WIDTH = 4;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN

  0 : 0000;
  1 : 0001;
  2 : 0010;
  3 : 0011;

. . . (some lines not shown)
 1E : 1110;
 1F : 1111;

END;
```

Figure 14: An example memory initialization file (MIF).

Appendix A

When creating a Memory Initialization File in the Intel® Quartus® Prime Text Editor, you must start with the **DEPTH**, **WIDTH**, **ADDRESS_RADIX** and **DATA_RADIX** keywords. You can use Tab "\t" and Space " " characters as separators, and insert multiple lines of comments with the percent "%" character, or a single comment with double dash "--" characters. Address : data pairs represent data contained inside certain memory addresses and you must place them between the **CONTENT BEGIN** and **END** keywords, as shown in the following examples:

```
% multiple-line comment
multiple-line comment %
-- single-line comment
DEPTH = 32; -- The size of memory in words
WIDTH = 8; -- The size of data in bits
ADDRESS_RADIX = HEX; -- The radix for address values
DATA_RADIX = BIN; -- The radix for data values
CONTENT -- start of (address : data pairs)
BEGIN
00 : 00000000; -- memory address : data
01 : 00000001;
02 : 00000010;
03 : 00000011;
04 : 00000100;
05 : 00000101;
06 : 00000110;
07 : 00000111;
08 : 00001000;
09 : 00001001;
0A : 00001010;
0B : 00001011;
0C : 00001100;
END;
```

Address : Data Pairs Syntax Rules	Definition	Example
`A : D`	`Addr[A]=D`	`2 : 4` Address: `01234567` Data: `00400000`
`[A0..A1] : D`	`Addr[A0]` to `[A1]` contain data `D`	`[0..7] : 6` Address: `01234567` Data: `66666666`

`[A0..A1] : D0 D1`	`Addr[A0] = D0,`
	`Addr[A0+1] = D1,`
	`Add [A0+2] = D0,`
	`Addr[A0+3] = D1,`
	`until A0+n = A1`
		`[0..7] : 5 6`
		Address: `01234567`
		Data: `56565656`

`A : D0 D1 D2`	`Addr[A] = D0,`
	`Addr[A+1] = D1,`
	`Addr[A+2] = D2`
		`2 : 4 5 6`
		Address: `01234567`
		Data: `00456000`

Appendix A

This file is shown in Figure 1. Note that you can use any base you want (BIN, DEC, HEX or OCT) in the MIF file. The DEPTH is the number of addresses (memory words) in the ROM and the WIDTH is the size of the data bus (the number of data bits per word). This example has 16 locations with zeros, 7 different data values and then $FF for the remaining locations.

```
DEPTH = 1024;  % Memory depth and width are required %
WIDTH = 8;     %   Enter decimal numbers for each    %

ADDRESS_RADIX = HEX;   % Address and value radixes are optional     %
DATA_RADIX = HEX;      % Enter BIN, DEC, HEX, or OCT; unless        %
                       % otherwise specified, radixes = HEX         %

-- Specify values for addresses, which can be single address or range

CONTENT
BEGIN

[0..F]     :    0;    % First 16 values are zero %
10         :    33;   % Single address data %
11         :    5C;   % Addr[11] = 5C %
12         :    99;
13         :    A1;   % Addr[13] = A1 %
14         :    B2;
15         :    C3;
16         :    D4;   % Addr[16] = D4 %
[17..3FF]  :    FF;   % remaining locations are FF %
END ;                 % You must have END statement! %
```

Appendix B

VHDL Modelling of memory

Memory can be implemented by:

- Instantiation
  - Creating VHDL that creates an instance of a particular (predefined) memory component
  - Altera's `altsyncram` megafuncation will be most commonly used
  - Can write structural VHDL or use Quartus Megawizard Plug-in Manager to generate structural VHDL
- Inference
  - Write behavioral VHDL that will model the memory
  - VHDL compiler can infer the memory
  - Will need to carefully construct VHDL code for the compiler to do this. Need to refer to Quartus Help