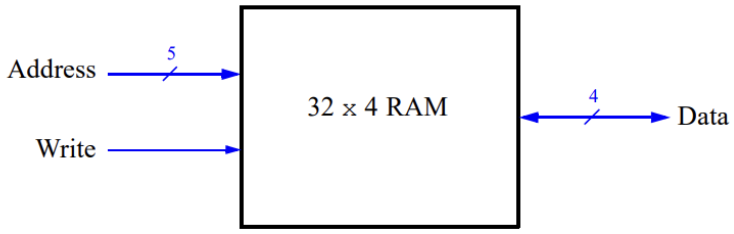
Laboratory Exercise 8

Memory Blocks

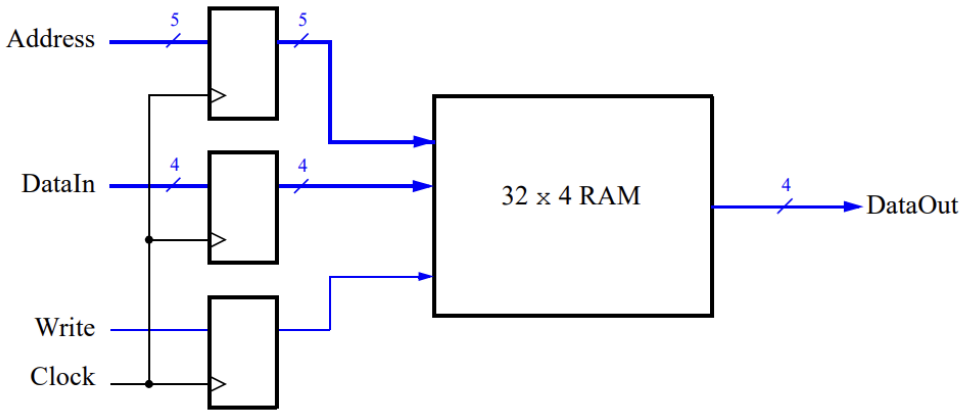
In computer systems it is necessary to provide a substantial amount of memory. If a system is implemented using FPGA technology it is possible to provide some amount of memory by using the memory resources that exist in the FPGA device. In this exercise we will examine the general issues involved in implementing such memory.

A diagram of the random access memory (RAM) module that we 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 input.

The FPGAs that are included on the Intel FPGA DE10-Lite, DE0-CV, DE1-SoC, and DE2-115 boards provide dedicated memory resources. The MAX 10 FPGA on the DE10-Lite, and Cyclone IV FPGA on the DE2-115 contain dedicated memory resources called M9K blocks. The Cyclone V FPGA on the DE0-CV and DE1-SoC boards have M10K blocks. Each M9K block contains 9216 memory bits, while each M10K block contains 10240 memory bits. Both M9K and M10k 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 exercise we will use an aspect ratio that is four bits wide, and we will use only the first 32 words in the memory. Although the M9K and M10K blocks support many other modes of operation, we will not discuss them here.



(*a*) RAM organization



(b) RAM implementation

Figure 1: A 32 x 4 RAM module.

There are two important features of the M9K and M10K blocks that have to be mentioned. First, they includes 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. Given these requirements, we will implement the modified 32 x 4 RAM module shown in Figure 1b. It includes registers for the address, data input, and write ports, and uses a separate unregistered data output port.

Part I

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 this exercise we will use such a module to implement the memory shown in Figure 1b.

1. Create a new Quartus project to implement the memory module.

2. To 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. Select Verilog HDL as the type of output file to create, give the file the name ram32x4.v, and click OK. As shown in Figure 2 specify a memory size of 32 four-bit words. Select M9K if your DE-series board has a MAX 10 or Cyclone IV FPGA, otherwise select M10K. Also on this screen accept the default setting to use a single clock for the memory's registers, and then advance to the page shown in Figure 3. On this page 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 1b, with registered input ports and unregistered output ports. Accept defaults for the rest of the settings in the Wizard, and click the Finish button to exit from this tool. Examine the ram32x4.v Verilog file which defines the following subcircuit:

module ram32x4 (input [4:0] address, input clock, input [3:0] data, input wren, output [3:0] q);

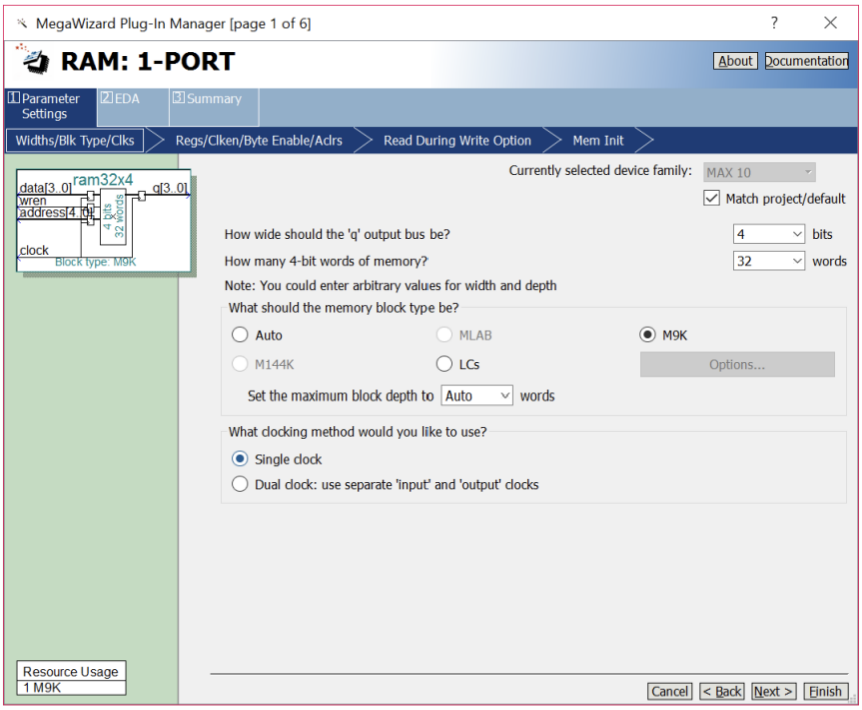


Figure 2: Configuring the size of the memory module.

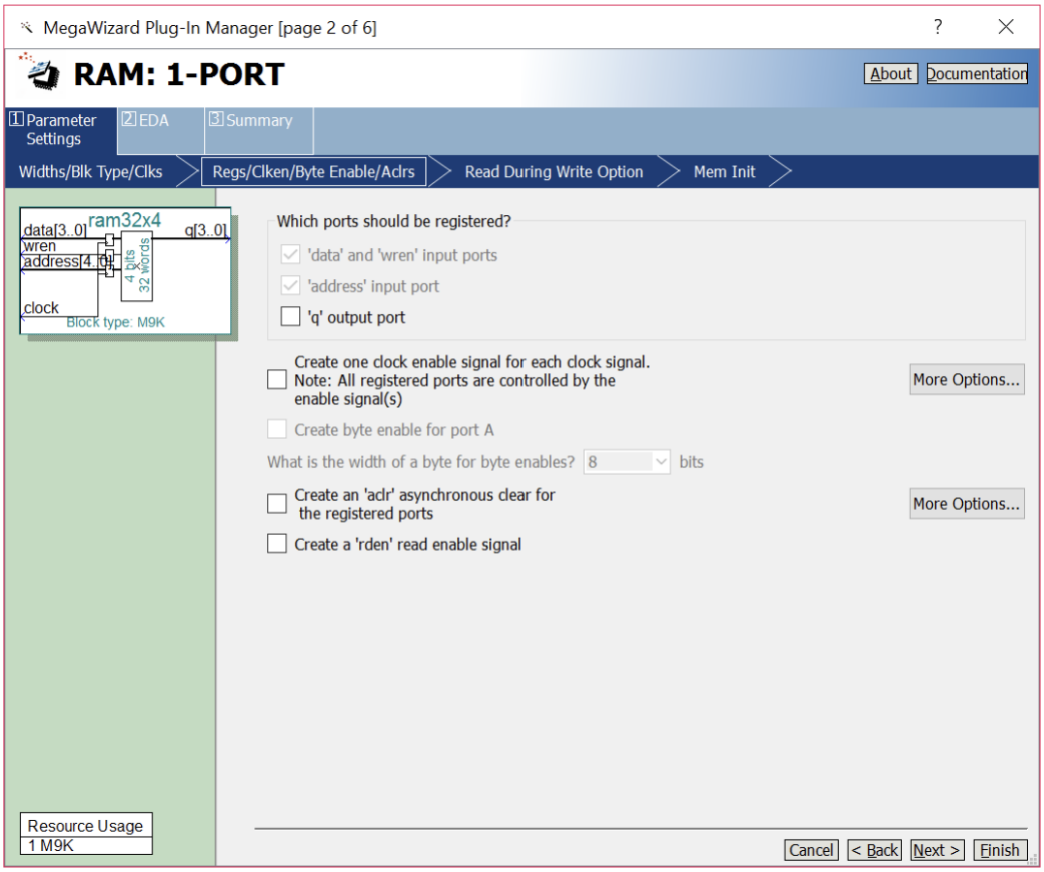


Figure 3: Configuring input and output ports.

3. Instantiate this subcircuit in a top-level Verilog file that includes appropriate input and output signals for the memory ports given in Figure 1b. 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.

4. Simulate the behavior of your circuit and ensure that you can read and write data in the memory. An example simulation output is given in Figure 4.

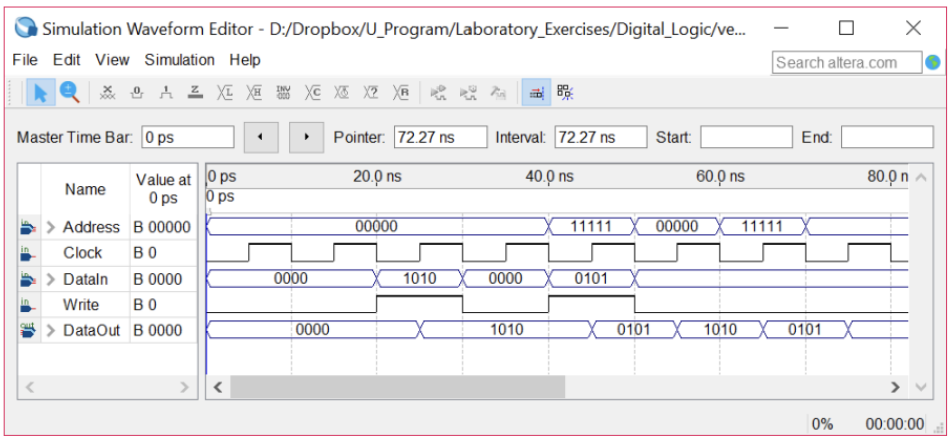


Figure 4: An example of simulation output.

CODE:

```
karthik
module part1(address,clock,data,wren,q);
input [4:0] address;
input clock;
input [3:0] data;
input wren;
output [3:0] q;
ram32x4 i1(address,clock,data,wren,q);
endmodule
```

Part II

Now, we want to realize the memory circuit in the FPGA on your DE-series 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 DE-series board.

2. Create another Verilog file that instantiates the ram32x4 module and that includes the required input and output pins on your DE-series board. Use slide switches SW3−0 to provide input data for the RAM, and use switches 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 − 4, 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.

CODE:

```
// karthik
// This code instantiates a 32 x 4 memory
// inputs: KEY0 is the clock, SW3-SW0 provides data to write into
memory.
// SW8-SW4 provides the memory address, SW9 is the memory Write input.
// outputs: 7-seg displays HEX5-4 show the memory address, HEX2
// displays the data input to the memory, and HEX0 show the contents
read
// from the memory. LEDGR shows the status of the SW switches.
module part2 (KEY, SW, HEX5, HEX4, HEX2, HEX0, LEDR);
      input [0:0] KEY;
      input [9:0] SW;
      output [0:6] HEX5, HEX4, HEX2, HEX0;
      output [9:0] LEDR;

      wire Clock, Write;
      wire [4:0] Address;
      wire [3:0] DataIn, DataOut;

      assign Clock = KEY[0];
      assign Write = SW[9];
      assign DataIn = SW[3:0];
      assign Address = SW[8:4];

      // instantiate memory module
      // module ram32x4 (address, clock, data, wren, q);
```

```
	ram32x4 U1 (Address, Clock, DataIn, Write, DataOut);

	// display the data input, data output, and address on the 7-segs
	hex7seg digit0 (DataOut[3:0], HEX0);
	hex7seg digit2 (DataIn[3:0], HEX2);
	hex7seg digit5 ({3'b0, Address[4]}, HEX5);
	hex7seg digit4 (Address[3:0], HEX4);

	assign LEDR[3:0] = DataIn;
	assign LEDR[8:4] = Address;
	assign LEDR[9] = Write;
endmodule

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

	reg [0:6] display;

	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'b0000100;
			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
```

Part III

Instead of creating a memory module subcircuit by using the IP Catalog, we can implement the required memory by specifying its structure in Verilog code. In a Verilog-specified design it is possible to define the memory as a multidimensional array. A 32 x 4 array, which has 32 words with 4 bits per word, can be declared by the statement

reg [3:0] memory_array [31:0];

In the Cyclone series of FPGAs, 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. The Quartus Help provides other examples of Verilog code that show how memory can be specified (search in the Help for “Inferred memory”).

Perform the following steps:

1. Create a new project which will be used to implement the desired circuit on your DE-series board.
2. Write a Verilog file that provides the necessary functionality, including the ability to load the RAM and read its contents as was done in Part II.

3. Assign the pins on the FPGA to connect to the switches and the 7-segment displays.

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.

CODE:

```
// karthik
// This code instantiates a 32 x 4 memory
// inputs: KEY0 is the clock, SW3-SW0 provides data to write into
memory.
// SW8-SW4 provides the memory address, SW9 is the memory Write input.
// outputs: 7-seg displays HEX5-4 show the memory address, HEX2
// displays the data input to the memory, and HEX0 show the contents
read
// from the memory. LEDR shows the status of the SW switches.
module part3 (KEY, SW, HEX5, HEX4, HEX2, HEX0, LEDR);
      input [0:0] KEY;
      input [9:0] SW;
      output [0:6] HEX5, HEX4, HEX2, HEX0;
      output [9:0] LEDR;

      wire Clock, Write;
      wire [4:0] Address;
      wire [3:0] DataIn, DataOut;

      assign Clock = KEY[0];
      assign Write = SW[9];
      assign DataIn = SW[3:0];
      assign Address = SW[8:4];

      reg [3:0] memory_array [31:0];
      reg [4:0] Address_reg;

      // infer RAM module
      always @(posedge Clock)
      begin
            if (Write)
                  memory_array[Address] <= DataIn;
            Address_reg <= Address;
   end
   assign DataOut = memory_array[Address_reg];

      // display the data input, data output, and address on the 7-segs
      hex7seg digit0 (DataOut[3:0], HEX0);
      hex7seg digit1 (DataIn[3:0], HEX2);
      hex7seg digit5 ({3'b0, Address[4]}, HEX5);
      hex7seg digit4 (Address[3:0], HEX4);
```

```
	assign LEDR[3:0] = DataIn;
	assign LEDR[8:4] = Address;
	assign LEDR[9] = Write;
endmodule

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

	reg [0:6] display;

	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'b0000100;
			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
```

Part IV

The SRAM block in Figure 1 has a single port that provides the address for both 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. 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 5, choose With one read port and one write port in the category called How will you be using the dual port ram?
   Configure the memory size, clocking method, and registered ports the same way as Part II. As shown in Figure 6 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.

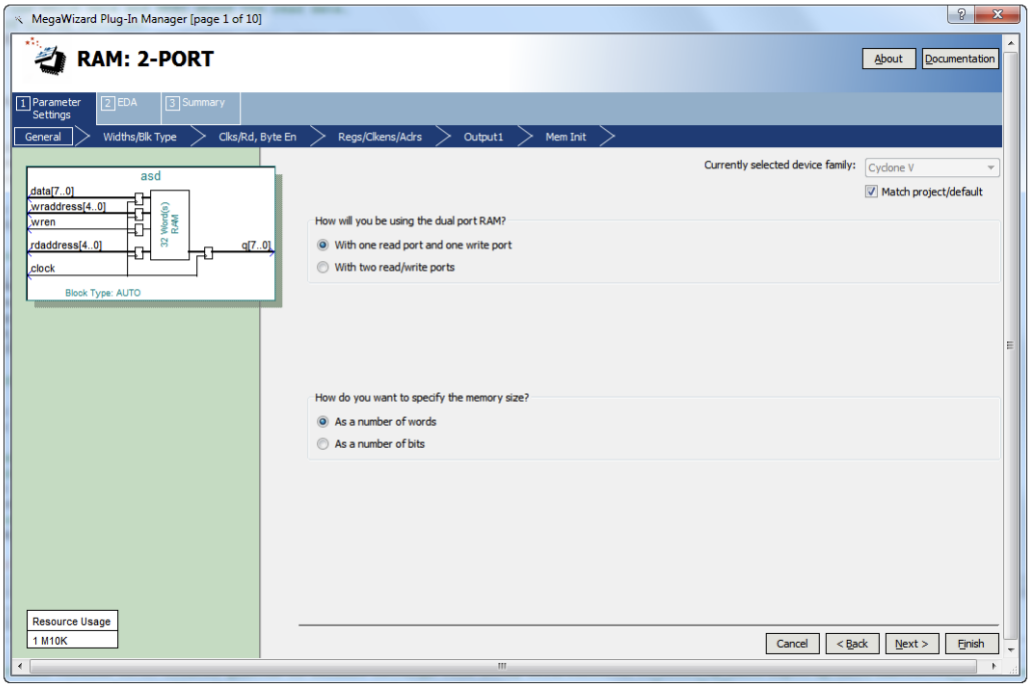


Figure 5: Configuring the two input ports of the RAM.

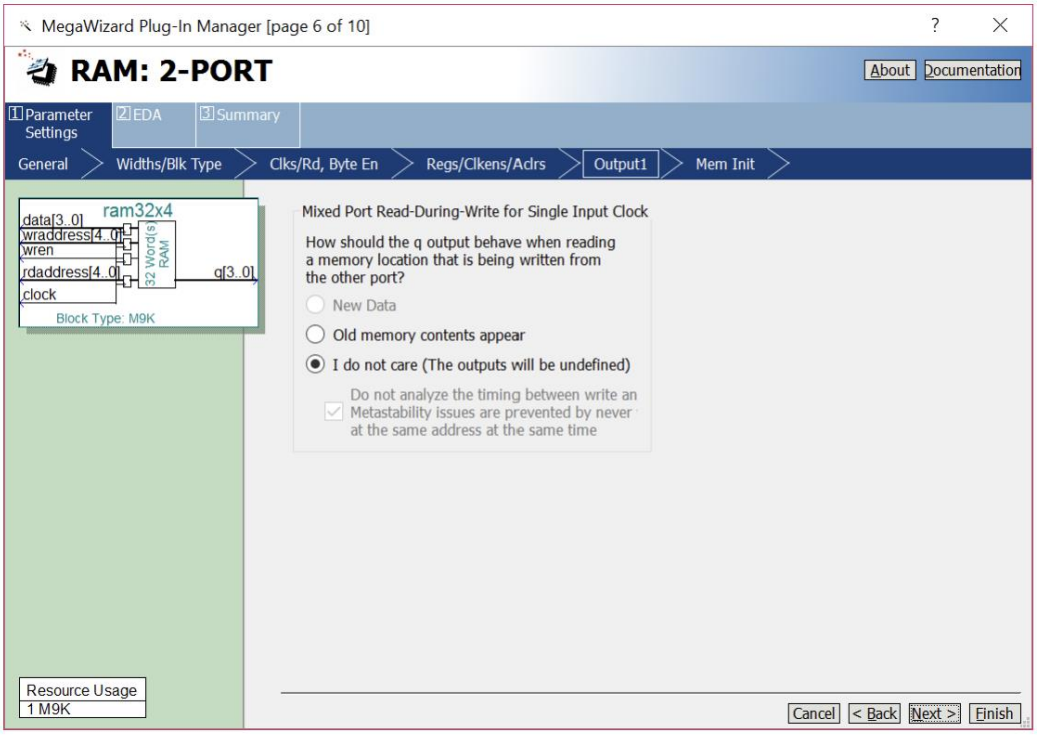


Figure 6: Configuring the output of the RAM when reading and writing to the same address.

Figure 7 shows how the memory words can be initialized to specific values. It makes use of a feature that allows the memory module to be loaded with data when the circuit is programmed into the FPGA chip. As shown in the figure, 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 8. You can also learn about the format of a memory initialization file (MIF) by using the Quartus Help. You will need

to create a MIF file like the one in Figure 8 to test your circuit. Finish the Wizard and then examine the generated memory module in the file ram32x4.v.

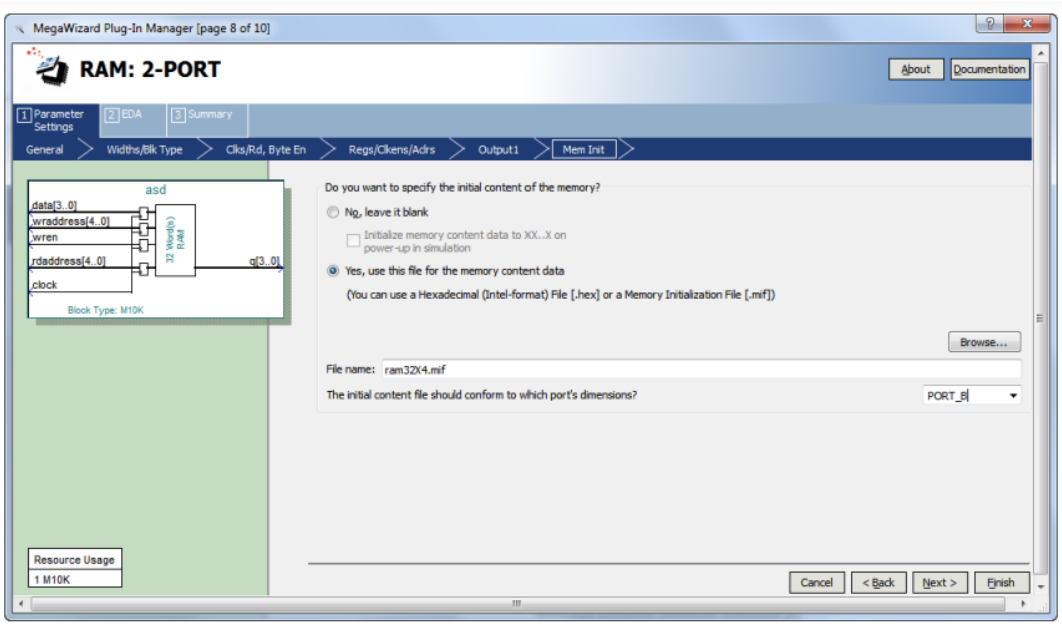


Figure 7: Specifying a memory initialization file (MIF).

```
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 8: An example memory initialization file (MIF).

2. Write a Verilog 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 6 Page 50 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.

3. 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.

CODE:

```
//karthik
// This code implements a simple dual-port memory
// inputs: CLOCK_50 is the clock, KEY0 is Resetn, SW3-SW0 provides data
to
// write into memory.
// SW8-SW4 provides the memory address for writing, SW9 is the memory
Write input.
// outputs: 7-seg display HEX5-4 displays the write address, and HEX3-2
shows the read
// address. HEX1 displays the write data and HEX0 shows the read data.
// LEDR shows the status of the SW switches.
module part4 (CLOCK_50, KEY, SW, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0,
LEDR);
	input CLOCK_50;
	input [0:0] KEY;
	input [9:0] SW;
	output [0:6] HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
	output [9:0] LEDR;

	wire Clock, Resetn, Write, Write_sync;
	wire [4:0] Write_address, Write_address_sync;
	wire [3:0] DataIn, DataIn_sync, DataOut;

	assign Resetn = KEY[0];
	assign Clock = CLOCK_50;

	// synchronize all asynchronous inputs to the clock
	regne #(.n(1)) wr_sync_reg(SW[9], Clock, Resetn, 1'b1,
Write_sync);
	regne #(.n(1)) wr_reg(Write_sync, Clock, Resetn, 1'b1, Write);
	regne #(.n(5)) addr_sync_reg(SW[8:4], Clock, Resetn, 1'b1,
Write_address_sync);
	regne #(.n(5)) addr_reg(Write_address_sync, Clock, Resetn, 1'b1,
Write_address);
	regne #(.n(4)) din_sync_reg(SW[3:0], Clock, Resetn, 1'b1,
DataIn_sync);
	regne #(.n(4)) din_reg(DataIn_sync, Clock, Resetn, 1'b1, DataIn);

	// one second cycle counter
	parameter m = 25;
	reg [m-1:0] slow_count;
	reg [4:0] Read_address; // cycles from addresses 0 to 31 at one
second per address

	// Create a 1Hz 5-bit address counter
	// A large counter to produce a 1 second (approx) enable
	always @(posedge Clock)
		slow_count <= slow_count + 1'b1;
	// the read address counter
```

```
	always @ (posedge Clock)
		if (Resetn == 0)
			Read_address <= 5'b0;
		else if (slow_count == 0)
			Read_address <= Read_address + 1'b1;

	// instantiate memory module
	// module ram32x4 (clock, data, rdaddress, wraddress, wren, q);
	ram32x4 U1 (Clock, DataIn, Read_address, Write_address, Write,
DataOut);

	// display the data input, data output, and addresses on the 7-
segs
	hex7seg digit5 ({3'b0, Write_address[4]}, HEX5);
	hex7seg digit4 (Write_address[3:0], HEX4);
	hex7seg digit3 ({3'b0, Read_address[4]}, HEX3);
	hex7seg digit2 (Read_address[3:0], HEX2);
	hex7seg digit1 (DataIn[3:0], HEX1);
	hex7seg digit0 (DataOut[3:0], HEX0);

	assign LEDR[3:0] = DataIn;
	assign LEDR[8:4] = Write_address;
	assign LEDR[9] = Write;
endmodule

module regne (R, Clock, Resetn, E, Q);
	parameter n = 7;
	input [n-1:0] R;
	input Clock, Resetn, E;
	output [n-1:0] Q;
	reg [n-1:0] Q;

	always @(posedge Clock)
		if (Resetn == 0)
			Q <= {n{1'b0}};
		else if (E)
			Q <= R;
endmodule

//-------------------------------------------------

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

	reg [0:6] display;

	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'b0000100;
                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
```