Introduction to FPGA Programming

2015-07-11

Contents

1 Introduction and Getting Started 1

2 4-bit Full-Adder and Simple ALU 3

3 Using Counters and 7-segment Display 4

4 Automatically Glowing and Fading LED 6

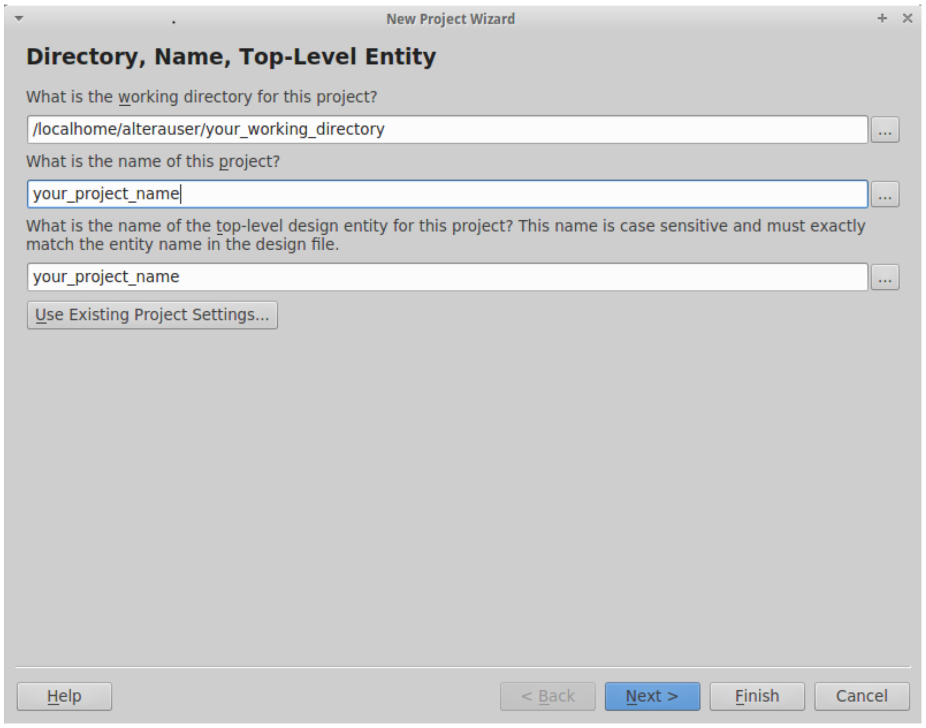
5 Working with Embedded Memory Blocks 7

Appendices 11

A Creating Memory Files 11

1 Introduction and Getting Started

Before starting to write programs, this section tells you how you can start a new project and set things up correctly. First, run quartus II on your system.



Directory, Name, Top-Level Entity

What is the working directory for this project?

/localhome/alterauser/your_working_directory

What is the name of this project?

your_project_name

What is the name of the top-level design entity for this project? This name is case sensitive and must exactly match the entity name in the design file.

your_project_name

Figure 1: Create a new project

1. In the Quartus II software, select **File** > **New Project Wizard.** The new project wizard opens, here you have to mention the project directory, name of the project and your top level design entity. See figure 1

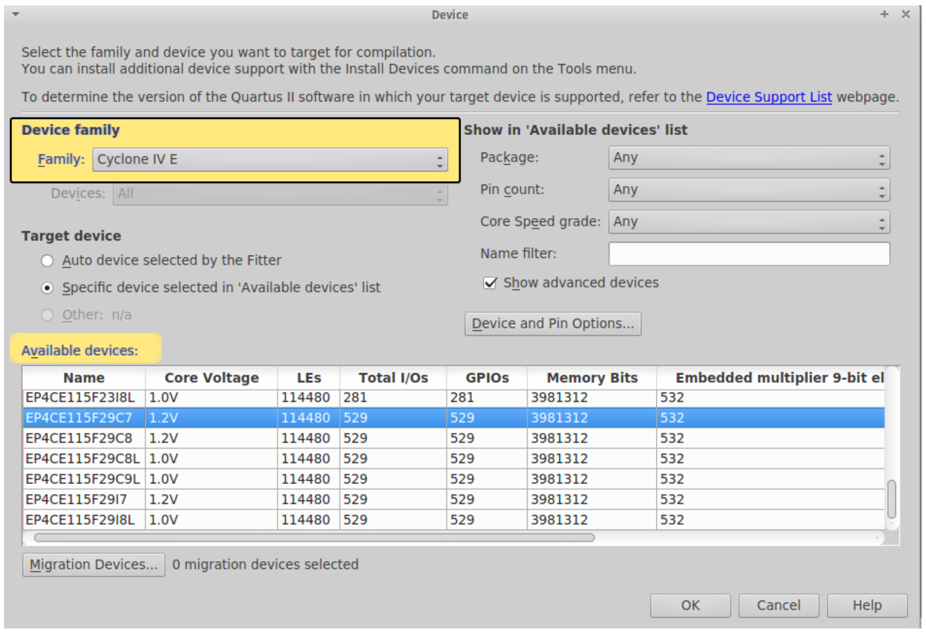
   (a) **Enter the working directory for your project in which you will store Quartus II project files.**
      - **Note:** File names, project names, and directories in the Quartus II software can not contain spaces.

   (b) **Enter the name of the project.**

   (c) **Enter the name of the top-level design entity.**

2. Click **Next.** If prompted that the working directory doesn't exist and whether you would like to create one, click **Yes** and proceed.

3. In the next page, select **Empty Project** and click **Next.**
4. Next, if you are prompted to **Add Files** page, **ignore** and click **Next.**
5. **Family and Device Settings** is the final and crucial step in creating your first Quartus II project.
   - Select **Cyclone IV E** as your device family. See figure 2.
   - Select **EP4CE115F29C7** as your device, see figure 2. and click **OK**



Select the family and device you want to target for compilation.
You can install additional device support with the Install Devices command on the Tools menu.

To determine the version of the Quartus II software in which your target device is supported, refer to the Device Support List webpage.

Device family

Family: Cyclone IV E

Devices: All

Show in 'Available devices' list

Package: Any

Pin count: Any

Core Speed grade: Any

Name filter:

✓ Show advanced devices

Target device

( ) Auto device selected by the Fitter

(●) Specific device selected in 'Available devices' list

( ) Other: n/a

Available devices:

Name	Core Voltage	LEs	Total I/Os	GPIOs	Memory Bits	Embedded multiplier 9-bit el
EP4CE115F23I8L	1.0V	114480	281	281	3981312	532
EP4CE115F29C7	1.2V	114480	529	529	3981312	532
EP4CE115F29C8	1.2V	114480	529	529	3981312	532
EP4CE115F29C8L	1.0V	114480	529	529	3981312	532
EP4CE115F29C9L	1.0V	114480	529	529	3981312	532
EP4CE115F29I7	1.2V	114480	529	529	3981312	532
EP4CE115F29I8L	1.0V	114480	529	529	3981312	532

0 migration devices selected

Figure 2: Selecting the correct device platform for your project

6. Now, you can add Verilog HDL file from **File** > **New** > **Verilog HDL File** to the project and begin writing your program.

2 4-bit Full-Adder and Simple ALU

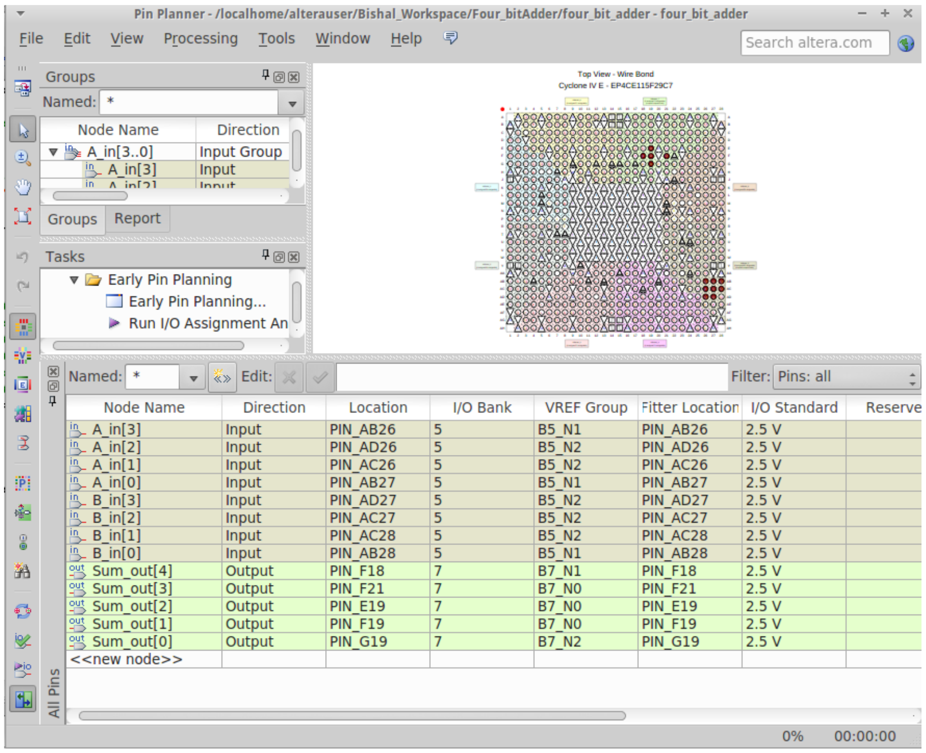
First step: create a 4-bit adder and show the output result using the LEDs. In your program, you should have two 4-bit input sets that will be added together. Think about the overflow, your output should be 5-bits long. Design your code such that whenever an input is changed the task is performed, e.g.

```
// define module entity
module four_bit_adder( A_in, B_in, Sum_out );

//define inputs and outputs
input [3:0] A_in;
input [3:0] B_in;

output [4:0] Sum_out;
reg [4:0] Sum_out;

// do the addition whenever A_in or B_in change states
always@( A_in or B_in ) begin
....write code to add A_in and B_in and assign it to output
end

endmodule
```

Hint: Compile your project once through menu (Processing > Start Compilation) or just press small play button on the toolbar. Next, open pin planner through menu **Assignments** > **Pin Planner**. **Note:** If you don't compile your program before assigning pins, you will not be able to see any listings on the **Node Name** and **Direction** columns in Pin Planner window, see figure 3.

The node name refers to your input and output signal names from the program. Now assign, input signals to switches and output signals to LEDs. On **page 35** of *Terasic DE2-115 User Manual*, you can find pin names and locations. Assign corresponding pin locations to your signals and **compile the program again**. Then you can use the programmer to run your code onto the FPGA module.

Next step: In this step, you will create a simple ALU as an extension of your earlier exercise. Now, you add 2-bit control sequence as input in addition to your earlier inputs. Instead of just performing addition, now the control sequence decides which operation is to be performed, refer to table 1. Again, show your output using LEDs.



Node Name	Direction
A_in[3..0]	Input Group
A_in[3]	Input
A_in[2]	Input

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserve
A_in[3]	Input	PIN_AB26	5	B5_N1	PIN_AB26	2.5 V	
A_in[2]	Input	PIN_AD26	5	B5_N2	PIN_AD26	2.5 V	
A_in[1]	Input	PIN_AC26	5	B5_N2	PIN_AC26	2.5 V	
A_in[0]	Input	PIN_AB27	5	B5_N1	PIN_AB27	2.5 V	
B_in[3]	Input	PIN_AD27	5	B5_N2	PIN_AD27	2.5 V	
B_in[2]	Input	PIN_AC27	5	B5_N2	PIN_AC27	2.5 V	
B_in[1]	Input	PIN_AC28	5	B5_N2	PIN_AC28	2.5 V	
B_in[0]	Input	PIN_AB28	5	B5_N1	PIN_AB28	2.5 V	
Sum_out[4]	Output	PIN_F18	7	B7_N1	PIN_F18	2.5 V	
Sum_out[3]	Output	PIN_F21	7	B7_N0	PIN_F21	2.5 V	
Sum_out[2]	Output	PIN_E19	7	B7_N0	PIN_E19	2.5 V	
Sum_out[1]	Output	PIN_F19	7	B7_N0	PIN_F19	2.5 V	
Sum_out[0]	Output	PIN_G19	7	B7_N2	PIN_G19	2.5 V	
<<new node>>							

Figure 3: Pin planner : quartus II

Table 1: Example control sequence and operation

Control Sequence	Operation
00	Adder
01	OR
10	AND
11	XOR

3 Using Counters and 7-segment Display

First step: You should use the internal clock of the FPGA (50 MHz) to count seconds and blink one of the LEDs each seconds. Create a state register that holds high or low value and assign to LED output variable. You should think about the time the state register is either high or low, so that the blinking is observable. Start off with 1 second of period between each blinks, which means the LED will be high for 0.5 seconds and low for 0.5 seconds as well.

Hint: clock is an input to your program but it is triggered internally through on board oscillator. You just have to declare the clock as input signal variable to your module and assign it to corresponding location through pin planner, e.g **PIN_Y2** is a 50 MHz clock source input. You can find clocks and pin locations in DE2-115 user manual. E.g. counter,

```
always@(posedge clock) begin
counter = counter  + 1;
end
```

In this case in one seconds counter will be incremented 50000000 times.

Second step: In this step, you will create a counter that goes from starting value 0 to 9 and repeats the pattern. Look at the DE2-115 manual on how to use 7-segment display in your program. Remember, each segment is turned on when it has low value.

Hint: In your program, you can use as many *always@* modules as you want. Try to seperate functioning like this for clearness. A simple 4-bit counter is sufficient to count upto 9. The counter updates in each second like, create a seconds counter and update your 0-9 counter in such a way

```
BCD <= (BCD==4'h9 ? 4'h0 : BCD+4'h1)
```

here BCD is a 4-bit register that holds value from 0 to 9. You can use the value BCD holds from a seperate *always@* block and assign it to output 7-segments.

Third step: If you have successfully completed previous task, you are now ready to create a full-fledged digital clock. It is just an extension of the previous task except that you have now a seconds, minutes and hours counter. Update each of these counters each seconds from your main *always@* block. Use another *always@(*)* block with bunch of case statements which changes the state of 7-segment display units based on current values of each timer counters. After having the digital clock ready, add more feature to it, for example incrementing and decrementing minutes and hours counters through pushbuttons on FPGA.

4 Automatically Glowing and Fading LED

In this exercise, the task is to make one of the LEDs glow starting from least possible intensity ("low always") to maximum possible intensity("high always") by appropriately driving LED output voltage.

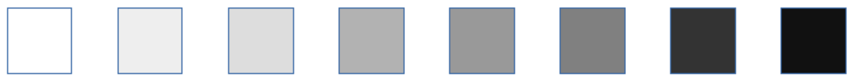


Figure 4: Sample intensity changes

Look at figure 4 to see how your led intensity must change from low (all white in the picture) to high (all dark in the picture). The square boxes represent the state of a led in time in terms of its intensity. The first box, corresponding to all white means that green or red LED in your FPGA board is turned off. Subsequently, the boxes increasing in darkness towards the right in figure 4 speak for the increase in intensity, i.e. its color whether its green or red. Again, the last box at the far right filled with all black corresponds to a led at its highest possible intensity, i.e. it is lit up all the time (it is always high!).

If you are confused with the idea of a LED having different intensity from low to high, start out by assigning a value 1 (high) to a LED all the time, and then toggle the value in each clock cycle, i.e. for 1 clock cycle it will have a high value and the next cycle a low value, see an example below.

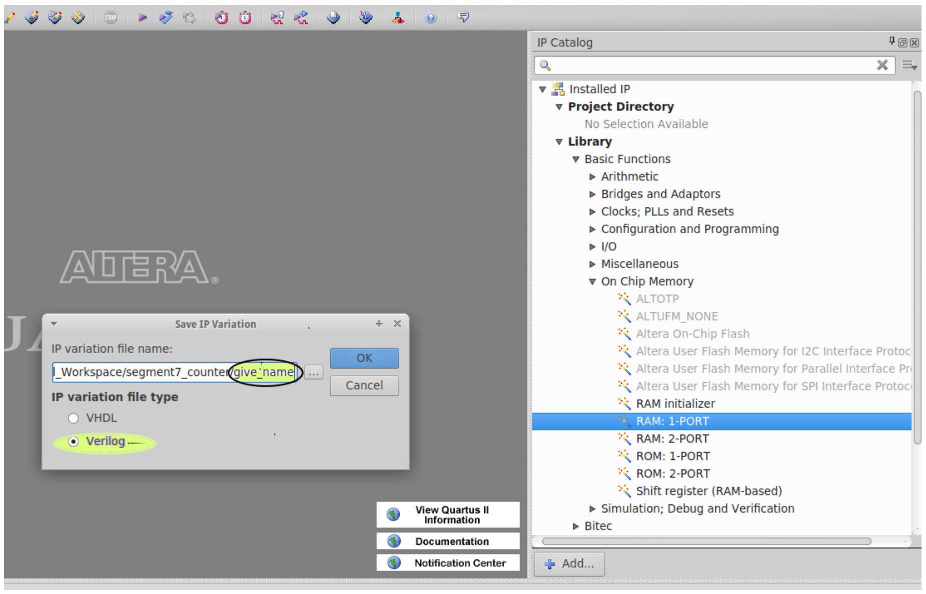
```
@always( posedge clk ) begin
LED_state = ~LED_state;
end
assign LED_out = LED_state;
```

In the first case, with high value for *LED_ state* all the time, i.e. without using clock triggered *@always* block shown above, you should observe the LED is at its highest possible intensity, thus should be bright RED or bright GREEN color. On the other hand, with the second approach of toggling the state you will observe intensity reduced by half! The fact is, any frequency of change above 100Hz is rather too quick for human eye to observe and it averages the intensity. So, considering the 50 MHz clock frequency of Altera DE2-115 FPGAs used in the lab, the LED changes its intensity from high to low with frequency 25MHz, too fast for human eye to observe and it averages the highest (max. brightness) and lowest (dark off state), so you should observe 50% of the intensity compared to the first case. This observation should be the basis for automatic glowing and fading of LEDs in your current exercise.

Hint: By carefully taking the overflow of an addition of 3-bit registers you can have 8 intensity levels, with 4-bit registers 16 levels and so on. Remember, the intensity must change in time from low to high and back to low automatically and this depends on how fast the state of your registers change. Keep clock frequency in mind when designing such a system to control how fast it glows and fades back.

5 Working with Embedded Memory Blocks

In this experiment, it is required that a simple memory block be created through quartus IP Catalog. You are going to create **RAM: 1-Port IP Core** for use in your project. The purpose of this RAM module is to store pattern of bit sequences that can be used to drive certain LEDs on the FPGA in a pre-defined manner. For this exercise, you need to create a memory file with extension .HEX or .MIF that has pre-defined pattern of bit sequences. Refer to appendix on how to create one such file using Quartus II.


Save IP Variation

IP variation file name: _Workspace/segment7_counter/give_name

IP variation file type: VHDL [✓ Verilog]

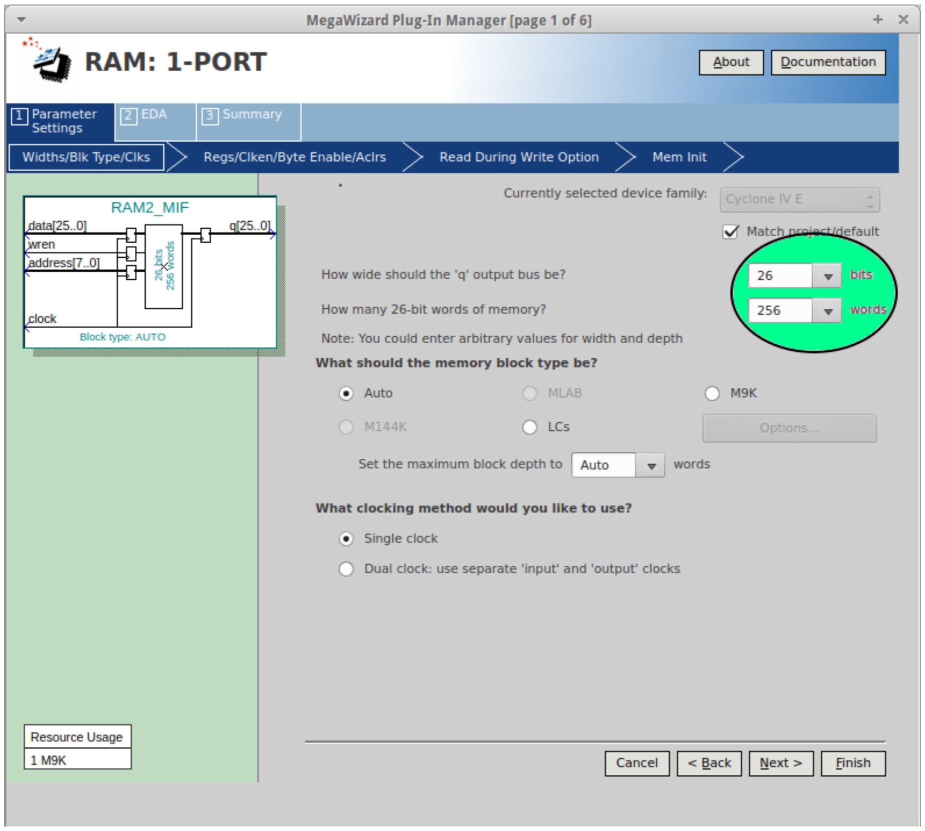
Figure 5: Accessing IP catalog on quartus II

Once you have either .MIF or .HEX memory file ready. Follow the following steps to create a RAM IP Core module and integrate it into your project. This project simulates the behaviour of a RAM hardware.

The RAM will hold *X words* (no of different on-off patterns) of *Y bits* to use a certain number of LEDs on the bottom edge of the FPGA board. The number of words will define how many different pattern of bit streams can be stored in the RAM. Find ALTERA Embedded Memory user guide for detailed specifications before following next steps.

1. Invoke IP Catalog (**Tools** > **IP Catalog**) after creating a new project. Then, expand **On Chip Memory** menu list, and double click on **RAM: 1-Port**, see figure 5. Enter IP variation file name, and select **Verilog** as the file type. Click **OK**

2. You should have a MegaWizard window open now, like the one shown in figure 6. Select the appropriate number of bits and words corresponding to the memory file created. The one given to you has 26 bit output (uses all the LEDs on the bottom of the FPGA) and 256 words (256 different pattern of 26 bits). Leave everything else as it is, Click **Next.**



Currently selected device family: Cyclone IV E

[✓] Match project/default

How wide should the 'q' output bus be? 26 bits

How many 26-bit words of memory? 256 words

Note: You could enter arbitrary values for width and depth

What should the memory block type be?

[✓ Auto] MLAB M9K M144K LCs

Set the maximum block depth to Auto words

What clocking method would you like to use?

[✓ Single clock]

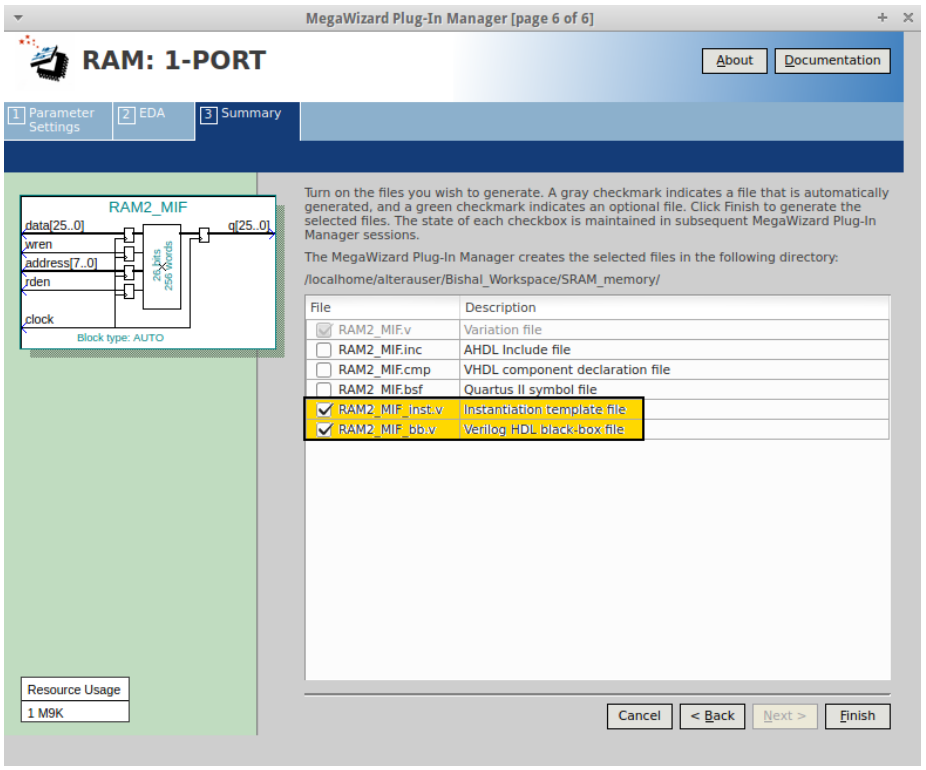
Dual clock: use separate 'input' and 'output' clocks

Figure 6: Megawizard RAM: 1-PORT Configuration

3. In the current MegaWizard page 2 of 6, **Check** read enable signal, this signal indicates to the RAM module if read operation is to be performed. There is already write enable signal ('wren') created by default, which signals if the write operation is to be performed on the memory module. However, we do not care about this for now since we have already created a memory file and we just need to perform read operations. Click **Next**. Select **Old Data** from the drop-down menu on the next page about Single Port Read-During-Write Option. Click **Next.**

4. In this step, MegaWizard page 4 of 6, select the option to set initial contents of the memory using the file created, see appendix. Specify the path to the file. Click **Next.** On MegaWizard page 5 of 6, click **Next.**

5. The last page, i.e. 6 of 6 on MegaWizard, make sure the last two options are checked as shown in figure 7. Click **Finish.** After successfully creating the memory block you will be asked if you would like to add it to your project, click **YES** and you are ready. Now you just need to instantiate the module in your main Verilog file with valid signals.



Turn on the files you wish to generate. A gray checkmark indicates a file that is automatically generated, and a green checkmark indicates an optional file. Click Finish to generate the selected files. The state of each checkbox is maintained in subsequent MegaWizard Plug-In Manager sessions.

The MegaWizard Plug-In Manager creates the selected files in the following directory:

/localhome/alterauser/Bishal_Workspace/SRAM_memory/

File	Description
✓ RAM2_MIF.v	Variation file
☐ RAM2_MIF.inc	AHDL Include file
☐ RAM2_MIF.cmp	VHDL component declaration file
☐ RAM2_MIF.bsf	Quartus II symbol file
✓ RAM2_MIF_inst.v	Instantiation template file
✓ RAM2_MIF_bb.v	Verilog HDL black-box file

Figure 7: Megawizard RAM: 1-PORT Configuration Final Step

6. Now, add a new Verilog file to your project. Define input and output signals. Output signal will be the length of the word connected to LEDs. Input will be the Clock signal. Moreover, you need to connect appropriate signals to the RAM module that has been created. **Remember**, the RAM module created is a hardware block, it has following signals you need to think about:

   - **address:** This input signal to RAM module holds the value of the address for the data you are about to read or write from. If you have 256 words in total, it would be an 8-bit long sequence going from 0-255.
   - **clock:** This is the input clock of the RAM module. You can connect the internal clock of the FPGA to the clock signal of RAM module. An example instantiation of the RAM module with all the signals is given below.
   - **data:** This is an *input signal* to the RAM module. If you were to write data to RAM block, you change this signal and corresponding address.
   - **wren:** This is also an *input signal* to the RAM module. This signals if the data is to be written to memory, thus called *write enable*. It is one-bit long, true (1) or false (0).
   - **rden:** Similar to *write enable*, this signal indicates *read enable*. It must be true when reading data from certain RAM address.
   - **q:** This is an *output signal* of the RAM block and which holds output data. Based on *address signal* and *read enable signal* it holds data corresponding to certain address from the memory. This signal's data should be used to drive the LEDs output.

Example instantiation of the memory block in your main Verilog file,

```
RAM1 my_ram(
.address (addr),
.clock (CLOCK_50),
.data (led_bit_streams),
.wren (write_enable),
.rden (read_enable)
.q (data_temp)
);
```

In the above example, RAM1 is the variation name of the memory block and my_ram is an instance of this block. The signals of the main module: *addr, CLOCK_ 50, led_ bit_ streams, write_ enable, read_ enable, data_ temp* are connected to the signals of the memory block: *address, clock, data, wren, rden, and q.* Try varying the speed of read operation, i.e. the frequency of change of signal *a*ddr from the main Verilog file, corresponding to above example instantiation.

Appendices

A Creating Memory Files

It is possible to create a .HEX or .MIF file using Quartus II. Follow the link below to start out and refer to the figures below.

`http://quartushelp.altera.com/14.0/mergedProjects/design/med/med_pro_med_files.htm`