Educational Objective

The educational objective of this lab is to further investigate the use of the component editor to create a custom IP module block that can be instantiated in QSYS, to understand the difference between 8/16/32-bit memory access, and to reinforce concepts of using functions, polling and interrupts.

Technical Objective

The technical objective of this laboratory is to use the Component Editor in the QSYS to add custom IP to a Nios II system. The custom IP being added is a RAM module that will utilize the Cyclone V FPGA block RAM. A RAM confidence test, which can be used for board debug, will be developed. The RAM test will run continuously until the user presses KEY1 on the DE1-SoC board, thus generating an interrupt that will terminate the test.

Documentation

This lab sheet is provided to you as a reference and may not contain every step or configuration will be provided. It is your responsibility as a Design Engineer to collect and understand all the requirements needed to complete the lab. This includes obtaining any additional information from previous labs or from any course lecture material.

It is also your responsibility to fully document the lab requirements and results in your laboratory portfolio so that you have enough information to reference later.

Deliverables

This is a two week lab. A video demo for Parts 1-2 due week1 (lab 6). Parts 3-5 due week 2 (lab 7).

Lab Procedure

In this lab, you will use the QSYS System Builder to create the system in Figure 1 below. The component you will be creating is the **Inferred Memory** IP.

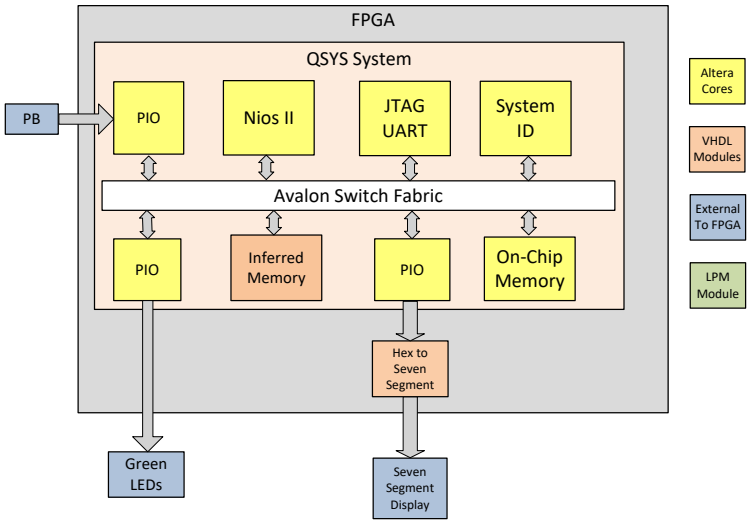

Figure 1: High-level Block Diagram of Design

To create the desired system, start the Quartus software and perform the following steps below.

Part 1

This lab is broken into multiple parts. Each part builds on the previous part so you must do all the steps in order. Read the instructions carefully and do not skip any steps.

The component editor in the QSYS system builder can be used to create custom IP cores that will interface with the processor via the Avalon Switch Fabric. The system designer must develop the logic for the IP (using VHDL in this case) and QSYS adds it to the embedded system memory map in the same manner as cores provided by Altera. In this lab you will create a custom memory component.

Realize the required hardware by implementing a Nios II system on the DE1-SoC board, as follows:

1. Create a new Quartus project that is stored in the directory **Lab6/Quartus/part1**.

Use QSYS to generate the desired circuit, called **nios_system**, which comprises:

- Nios II/e processor
- On-chip memory - RAM mode and 16Kbytes in size
- An 8-bit PIO output circuit for the red LEDs – don't forget to export
- A 1-bit PIO input circuit –configured for polling KEY1 – don't forget to export
- A JTAG UART - use the default settings
- System ID Peripheral with unique system ID

By default, Quartus II is setup to look for custom components in the directory named **ip** so we will create a new VHDL file named **raminfr.vhd** in the directory **ip/ram**.

The code below implements memory as a two-dimensional array and the Quartus tool will infer the RAM in the Cyclone V block RAM. Please note that there are several ways to use the Cyclone V Block RAM including the Mega- Wizard Plug-In Manager. However, for the purposes of this lab, we will use the VHDL provided and the component editor. The VHDL to infer the RAM follows:

```
-------------------------------------------------------------------
--- your header goes here
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY raminfr IS
  PORT(
    clk          : IN  std_logic;
    reset_n      : IN  std_logic;
    write_n      : IN  std_logic;
    address      : IN  std_logic_vector(11 DOWNTO 0);
    writedata    : IN  std_logic_vector(31 DOWNTO 0);
    --
    readdata     : OUT std_logic_vector(31 DOWNTO 0)
    );
END ENTITY raminfr;

ARCHITECTURE rtl OF raminfr IS

  TYPE ram_type IS ARRAY (4095 DOWNTO 0) OF std_logic_vector (31 DOWNTO 0);
  SIGNAL RAM         : ram_type;
  SIGNAL read_addr : std_logic_vector(11 DOWNTO 0);

BEGIN

  RamBlock : PROCESS(clk)
  BEGIN
    IF (clk'event AND clk = '1') THEN
      IF (reset_n = '0') THEN
        read_addr <= (OTHERS => '0');
      ELSIF (write_n = '0') THEN
        RAM(conv_integer(address)) <= writedata;
      END IF;
      read_addr <= address;
    END IF;
  END PROCESS RamBlock;

  readdata <= RAM(conv_integer(read_addr));

END ARCHITECTURE rtl;
```

Figure 2: VHDL for Inferred RAM

Open the component editor by clicking on **create new component** in the Component Library window in QSYS. After giving your component a name, click on the **files** tab. Add your raminfr.vhd file and choose **Analyze Synthesis Files.** There will be an error that will be fixed in the next step.

Next click on the **Signals & Interfaces** tab. You should see all of your port signals listed. Because of the way the ports were named, each should have the appropriate type as seen below.

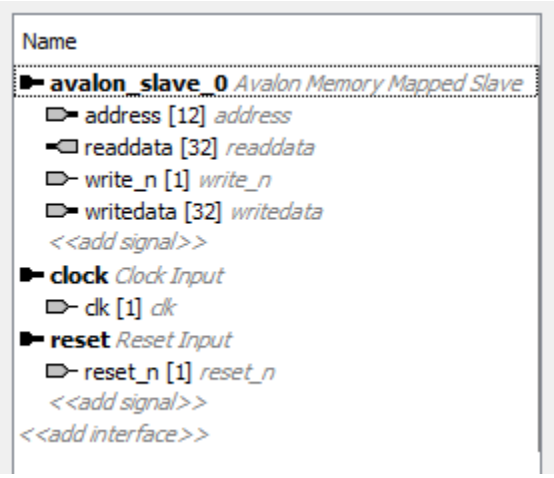

Figure 3: Signal Definition in Component Editor

As an Avalon Memory Mapped Slave, the Nios II processor will control all of the reads and writes to the RAM. Click on **avalon_slave_0** and set the reset of Avalon interface to the reset signal as shown in Figure 4 below. The reset signal is required to satisfy the Avalon interface but since the component is memory, the reset performs no action on the memory.

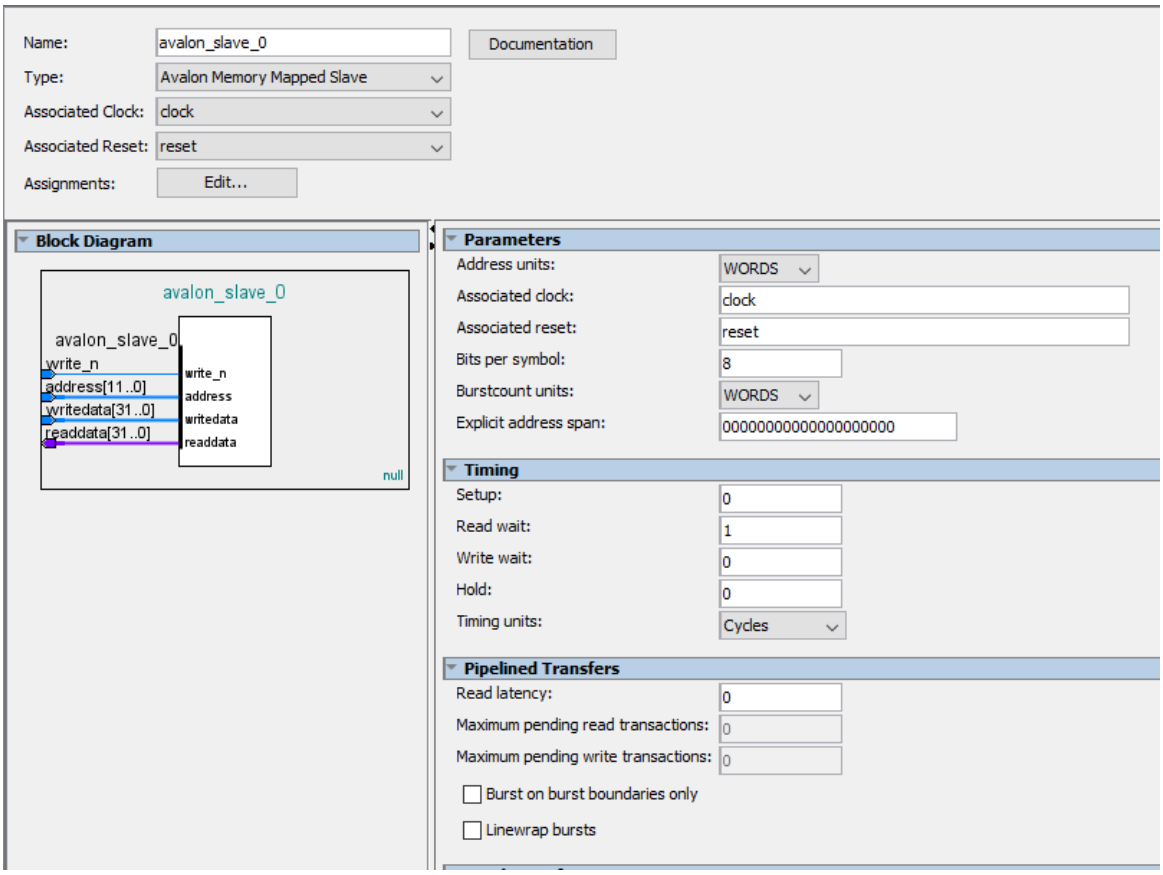

Figure 4: Defining Clock & Reset in Component Editor

There should not be any errors or warnings in the Component Wizard tab at this point. Click on **Finish**.

The new component will appear in the system contents window where you can select it for inclusion in your system.

Select the new component and name it **inferred_ram**.

Set the address of the inferred_ram to 0x0000_0000. After you set the component address, lock the address. Since the raminfr component is 16,384 bytes or 0x4000 bytes, the ending address should be located at memory address 0x3FFF.

Set the address of the on-chip RAM to 0x0000_4000. After you set the component address, lock the address. This will place the on-chip RAM immediately after the inferred_ram component in the memory map.

Once the raminfr has been added, select **System > Auto-Assign Base Addresses** and note the resultant memory map. Get in the habit of looking at this and recording the memory map.

Generate the system.

Instantiate the generated Nios II system within a VHDL module following good VHDL coding style. Also add the required files to your Quartus II project.

Add the necessary files to the project. Remember that the raminfr component is part of the nios_system so it does not have to be instantiated in the top_level VHDL nor does the file need to be added to the Quartus project.

Assign the pins needed to make the necessary connections, by importing the pin-assignment file.

Compile the Quartus project.

Program and configure the Cyclone V FPGA on the DE1-SoC Board to implement the generated system.

Part 2

Perform the following steps:

1. Open the Nios II Software Build Tools for Eclipse Program and create a new NIOS II Application and BSP from Template. Name the application **Lab6_Part2_APP**. Remember to unclick the **Use default location** to keep the Nios II design under the **software** directory.
2. Generate the BSP and copy the system.h file from the BSP folder to the APP folder.

Create a new file and name it **Lab6_Part2.c**

When working with memory it is usefully to have a function that can be used to verify the RAM is working. Write a function that performs a **byte** accessible RAM test. The function should accept three arguments: start address, the number of bytes to test and the test data to write to the RAM. The function should consist of two loops. The first loop will write the data to the requested RAM locations. The second loop will read the data back to verify it was successfully written. If the contents do not equal the expected data, then all the red LEDs should be turned on to indicate the failure. Be careful of the data type you use for this program.

Using the byte accessible function, create two more test functions. One function should perform a RAM test based on **16-bit** memory access and the other function should perform a memory test based on **32-bit** accesses. Both functions should accept the same arguments as the byte accessible function. Watch the data types of your pointers and loop interaction counter.

In the main program, create a ramp test that will call the 32-bit function with a ramp pattern (0x12345678). The ramp test data should be a 32-bit word.

Compile your program. You will notice an error. Take a look at the error message and think about what it is telling you.

In an embedded system, memory is typically in short supply so you need to use it judicially. By default, the Nios II Software Build Tools includes extra support libraries that can make your program bigger than it needs to be. When the program does not fit, you have two choices, add more memory or remove anything that is not required. In this case, we can remove extra code by right clicking on the BSP project and select **Properties**. Select the Nios II BSP Properties and make selections shown in the figure below.

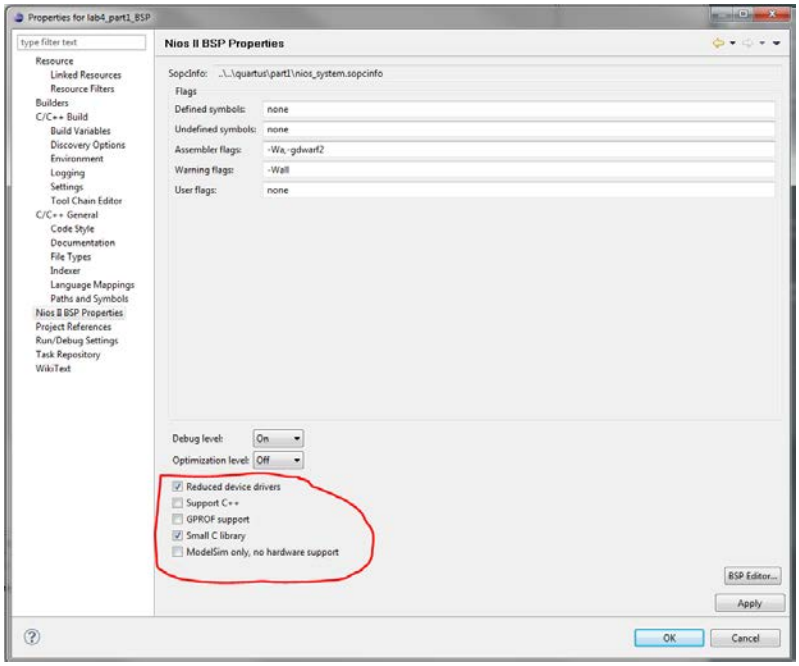

Figure 5: Nios II BSP Properties

Try to compile your program again as it should now compile without any errors.

Enter Debug mode. Click on **Memory** tab and add the address of your RAM module.

Put a breakpoint between the two loops in the function and run to the breakpoint.

Verify that 0x12345678 has been written to the correct locations. While in the memory viewer/editor, change the contents of one of the RAM locations to FFFFFFFF.

Set another breakpoint on the code that turns the LEDs on. Continue running the program and verify that the red LEDs turn on. Did the program stop on the correct RAM location that you changed?

Demo Submission:

Demonstrate your working ramp pattern, breakpoint, and failure in part2.