Contents

[Educational Objective 2](#_Toc474269187)

[Technical Objective 2](#_Toc474269188)

[Part 1 – Implement Custom Component That Ignores Byte Selects 2](#_Toc474269189)

[Create Quartus Project and Nios System 2](#_Toc474269190)

[Test RAM IP Using Byte and Half Word Access in the Debugger 5](#_Toc474269191)

[Test RAM IP Component Using 32 Bit Access 5](#_Toc474269192)

[Part 2 Implement the Byte Selects and Verify 6](#_Toc474269193)

[Test RAM IP Using Byte and Half Word Access Using C Code 6](#_Toc474269194)

[Create and Verify New RAM IP Peripheral Using Byte Enables 6](#_Toc474269195)

# Educational Objective

After completing this lab, you will

1. Learn how to use the component editor to create a custom IP module block that can be instantiated in QSYS.
2. Understand the difference between 8/16/32-bit memory access.
3. Gain experience using interrupts and c functions.

# Technical Objective

The technical objective of this laboratory is to use the Component Editor in QSYS, to add custom IP (Intellectual Property) to a Nios II system. The custom IP is essentially a device implemented in HDL, and which integrates into your QSYS system through the Avalon bus. The custom IP in this case is a RAM module, that will utilize the Cyclone V FPGA block RAM. A RAM confidence test will be written in C, and used to validate your VHDL RAM module. You can add your custom component (which you will write in VHDL), to your QSYS system in basically the same way you added the GPIO, JTAG serial port, RAM, etc.

In the first part of the lab we will implement the RAM module in VHDL not using the Byte Enable signals from the Avalon bus. When we test this module, it will pass for aligned full word access, but will fail for byte or half word access. We will also observe the behavior in the memory editor in the Eclipse debugger, and see how it does not work properly.

In the second part of the lab, we will “fix” the VHDL RAM module so that it properly utilizes the Byte Enable signals from the Avalon bus. We will then see that our confidence test passes for byte and half word access.

Your C code will run a memory test (confidence test) when KEY1 is pressed. You will use the synchronize and de-bounce functions, along with an interrupt enabled on the PIO port, just like you did in lab 2. If the memory passes the test, then 8 of the 10 LEDs will light up. As soon as any memory location fails, it will exit without lighting the LEDs.

# Part 1 – Implement Custom Component That Ignores Byte Selects

## Create Quartus Project and Nios System

Realize the required hardware by implementing a Nios II system on the DE2 board, as follows:

1. Create a new Quartus project using the same method presented in Lab 2. The Quartus project should be in directory cpet561/lab3/part1/quartus. As in lab 2, please also create a directory for your source code cpet561/lab3/part1/src. Make sure you use the synchronization module and debounce module as you did in Lab 2. You will read the KEY1 button using an interrupt, like you did in part 4 of Lab 2.
2. Create a QSYS system in the same manner you did in lab 1. The only difference at this point is the PIO peripherals. You will need one 8 bit PIO port to drive the LEDs. The LEDs will light up when the RAM confidence test you will write indicates memory is functioning properly. You will also need a one bit PIO input for KEY1, which will start the confidence test. In order to make the KEY1 input generate an interrupt, you will need to configure the PIO port appropriately. Check the box **Synchronously Capture** and select **RISING** from the drop-down. Check the box **Generate IRQ**, and select **EDGE** from the drop-down.
3. Create directory nios\_system/ip/ram in your Quartus project directory (note the nios\_system directory is probably already there), in which you will store the VHDL for the custom RAM component. The VHDL code to implement this ip can be found in the file ram\_ip.vhd on MyCourses, place this code in the nios\_system/ip/ram directory.
4. Open the component editor by double clicking on create new component in the **Component Library** window in QSYS.
5. Enter a **Name** and **Display Name** in the **Component Type** tab.
6. Click on the **HDL Files** tab, and add your ram\_ip.vhd file under **Synthesis Files**. You can click on **Analyze Synthesis File** to check for errors.
7. Next click on the **Signals & Interfaces** tab. You should see all of your port signals listed. The signal types need to be changed to match the Figure 1 below. Clicking on the signal type will bring up a pull-down menu where you will find the appropriate type.

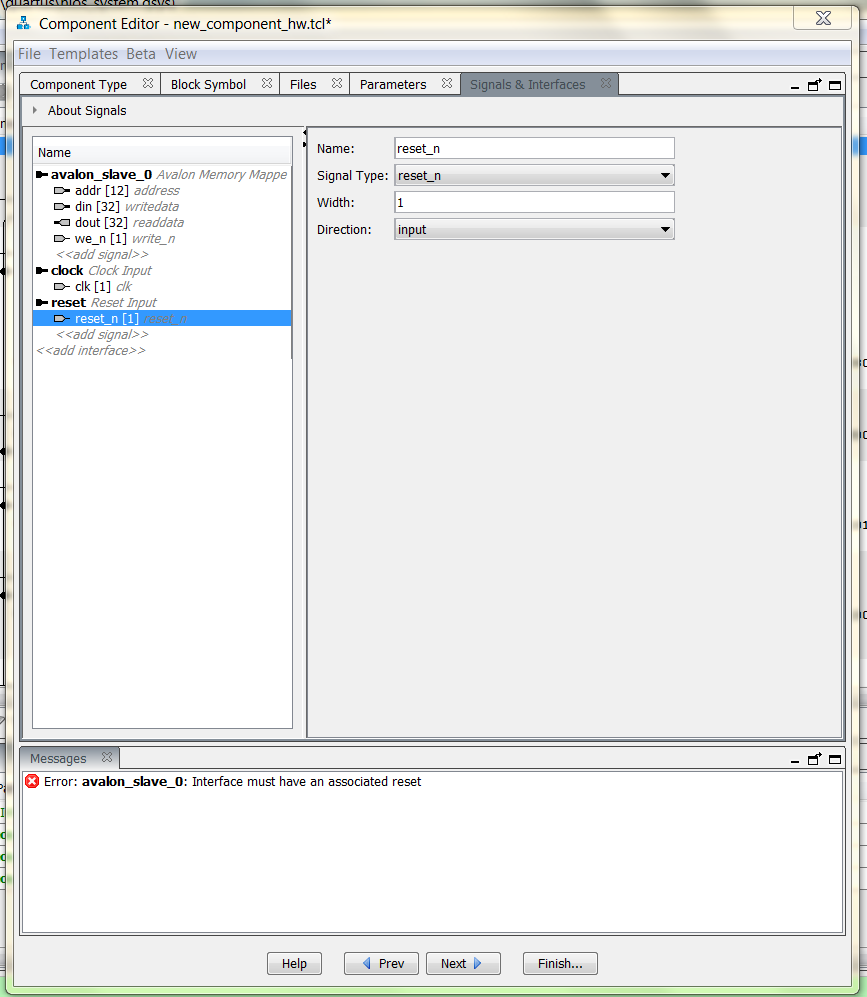


Figure 1: Required settings in the Signals and Interfaces tab.

1. Also in the **Signals and Interfaces** tab, click on **avalon\_slave\_0**, and for the **Associated Reset** pull-down, select reset. The reset signal is required to satisfy the Avalon interface but since the component is memory, the reset performs no action on the memory.
2. In the Component Wizard tab you can accept the defaults and then click on Finish.
3. The new component will appear in the system contents window. Add it to your Nios system and rename it to **inferred\_ram**.
4. Connect the **clock**, **reset**, and **avalon\_slave\_0** interfaces of the newly added custom peripheral.
5. Set the address of the inferred\_ram to 0x0000\_0000. After you set the component address, lock the address.
6. Select System > Auto‐Assign Base Addresses and note the resultant memory map.
7. Generate the system.
8. Instantiate the generated Nios II system within your VHDL top module along with any other required components. You need to use the **synchronizer** and **debounce** circuits from lab 2 in order to maintain reliable operation.
9. Add the necessary files to the project. Remember that the raminf 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 II project.
10. Assign the pins needed to make the necessary connections, by importing the lights\_import.qsf.
11. Compile the Quartus II project.
12. Program and configure the Cyclone II FPGA on the DE2 Board to implement the generated system.

## Test RAM IP Using Byte and Half Word Access in the Debugger

Although the RAM should work for 32 bit word access, in this section we will see that it does not work for byte or half word access. We will later (in part 2) fix this by implementing the Byte Enable signals, and then verify the RAM IP is fully functional.

Follow the following procedure to examine half word and byte access using the debugger:

1. Using the memory window to display the memory contents for the RAM, change the data view to display the data as word (4 bytes) by right-clicking in the window and selecting Format. Set the column size to 4.
2. Select an address in the RAM and change the value to 0xABCDEF98. Notice how that memory location’s content changes to 0xABABABAB. This is because the memory window is configured to be addressable as 8-bits so only the first two bytes were taken as the data.
3. Now change the data view to display the data as half-word (column size to 2). Select a RAM address location and change the contents to 0xABCD. Notice that the RAM location changes to 0xABCD but so does the adjacent memory location.
4. Now change the data view to display the data as byte (column size to 1). Select a RAM address location and change the contents to 0xA5. Notice that the 3 adjacent RAM location changes to 0xA5 with the location we are trying to change.
5. The display is because the custom RAM implementation is only accessible by word. Any attempt to access the RAM on half-word or byte boundaries causes the data to be replicated on the other byte lanes.

## Test RAM IP Component Using 32 Bit Access

In this part you will write the function:

bool ramConfidenceTest(unsigned long \*ramLocation\_ptr, numBytesToCheck).

The input parameters to this function are the location and size in bytes of the RAM you want to check. The return parameter is a boolean which will indicate true if the memory passes the test, and false if any memory location fails.

The function will check the RAM by writing a pattern to the entire RAM, and then reading the entire RAM back to verify that it matches the pattern written. As soon as any location fails to return the correct value, the function exits and returns false, and the LEDs remain off. If the entire RAM checks out then it returns a true and the LEDs light up.

The pattern written may be a ramp pattern, which means that each byte of the word is incremented by one on each write. For example the pattern:

* 0x1248a3b7
* 0x1349a4b8
* 0x144aa5b9

For 15 points extra credit, you can use a pseudo-random pattern instead of the ramp pattern.

The KEY1 button will clear the LEDs and start the test. If the test passes the LEDs will turn back on.

Write this function, and verify that indeed the memory functions as intended.

# Part 2 Implement the Byte Selects and Verify

Before you begin this part:

* Create a backup file (qar file).
* Backup your entire project directory.
* Submit the qar file, along with all you source code in the Lab 3 Part 1 dropbox.
* Make sure you have obtained sign-off on Part 1.

## Test RAM IP Using Byte and Half Word Access Using C Code

Create two new c functions, one for testing half word access, and one for testing byte access.

* bool ramTestByte(char \*ramLocation\_ptr, numBytesToCheck)
* bool ramTestHalfWork(unsigned short \*ramLocation\_ptr, numBytesToCheck)

Run each of these functions in a manner similar to the previous section, and verify that they indicate failure.

## Create and Verify New RAM IP Peripheral Using Byte Enables

Create a new project, which will be the same as created in the previous sections, but this time the RAM IP Peripheral will be raminfr\_be.vhd, and will use the byte enable signals to allow byte and half word access. You will hand in the archive of this project for part two of the lab in the Lab 3 Part 2 drop-box.

1. This new RAM component will have a new port call be\_n[3:0] added to it. These signals will be the port lanes of the memory when you write to the RAM. This means that both the we\_n signal and be[\*] signals needs to asserted to write into the RAM. For example, to write 32-bit data into the RAM you need we\_n, be\_n[3:0] to all be low. To write a byte in the lowest byte line you will need we\_n and be\_n[0] both to be low. (VHDL Design Hint: You want to create 4 independent 4Kx8 RAMs instead of one 4Kx32 RAM).
2. After you have created this new project with raminfr\_be.vhd, re-run your functions
   1. bool ramTestByte(char \*ramLocation\_ptr, numBytesToCheck)
   2. bool ramTestHalfWork(unsigned short \*ramLocation\_ptr, numBytesToCheck)

Verify that the functions now indicate success when you test the RAM.

1. Once again go back to the memory window in the debugger, and verify you can modify bytes individually.
2. Obtain sign-off on part 2 of the lab.
3. Submit .qar file and source code in the Lab 3 Part 2 drop-box.