Educational Objective

The educational objectives of this exercise are to investigate the use of a bus bridge to interface the NIOS II with an external component, to use the Altera's MegaWizard Plug-in Manager to create a RAM module and to create an arbitration scheme that manages two components writing to the same RAM location.

Technical Objective

The technical objective of this demonstration is to learn how to communicate using a bus. In the designs generated by Altera's QSYS system builder, the Nios II processor connects to peripheral devices by means of the Avalon Switch Fabric. To connect to the switch fabric, previous demonstrations and labs required the creation of a QSYS custom component. To make it possible to investigate the bus communication, without requiring the creation of a custom component, this exercise will use the Avalon to External Bus Bridge component. The bridge allows the designer to create a peripheral device and connects it to the Nios II system in the Quartus software. The Avalon to External Bus Bridge component creates a bus-like interface to which one or more "slave" peripherals can be connected.

Overview

This demonstration 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 external bus signals and timing information of the Avalon to External Bus Bridge is shown in Figure 1 below. The required signals are:

Signal Name	Size	Description
Address	k-bit (up to 32)	This is the address where the data is to be transferred to or from. The address is aligned to the data size. For 32-bit data, the address bits Address[1:0] are equal to 0. The byte-enable signals can be used to transfer less than 4 bytes.
BusEnable	1-bit	Indicates that all other signals are valid, and a data transfer should occur.
RW	1-bit	Indicates whether the data transfer is a Read (1) or a Write (0) operation.
ByteEnable	16, 8, 4, 2 or 1 bits	Each bit indicates whether or not the corresponding byte should be read or written. These signals are active high.
WriteData	128, 64, 32, 16 or 8 bits	The data to be written to the peripheral device during a Write transfer.
Acknowledge	1-bit	Used by the peripheral device to indicate that it has completed the data transfer.
ReadData	128, 64, 32, 16 or 8 bits	The data that is read from the peripheral device during a Read transfer.
IRQ	1 bit	Used by the peripheral device to interrupt the Nios II processor. This is an optional signal, which is not shown in the figure.

The bus is synchronous – all bus signals to the peripheral device must be read on the rising edge of the clock. To initiate a transfer, the Address, RW, ByteEnable and possibly WriteData signals are set to the appropriate values. Then, the BusEnable signal is set to 1.

If the RW signal is 1, then the transfer is a Read operation and the peripheral device must set the ReadData signals to the appropriate values and set the Acknowledge signal to 1. The Acknowledge signal must remain at 1 for only one clock cycle. The ReadData signals must be constant while the Acknowledge signal is being asserted. Note that the reason why the Acknowledge signal must be high for

exactly one clock cycle is that if this signal spans two or more cycles it may be interpreted by the Avalon Switch Fabric as corresponding to another transaction.

If the RW signal is 0, then the transfer is a Write operation and the peripheral device should write the value on the WriteData lines to the appropriate location. Once the peripheral device has completed the Write transfer, it must assert the Acknowledge signal for one clock cycle.

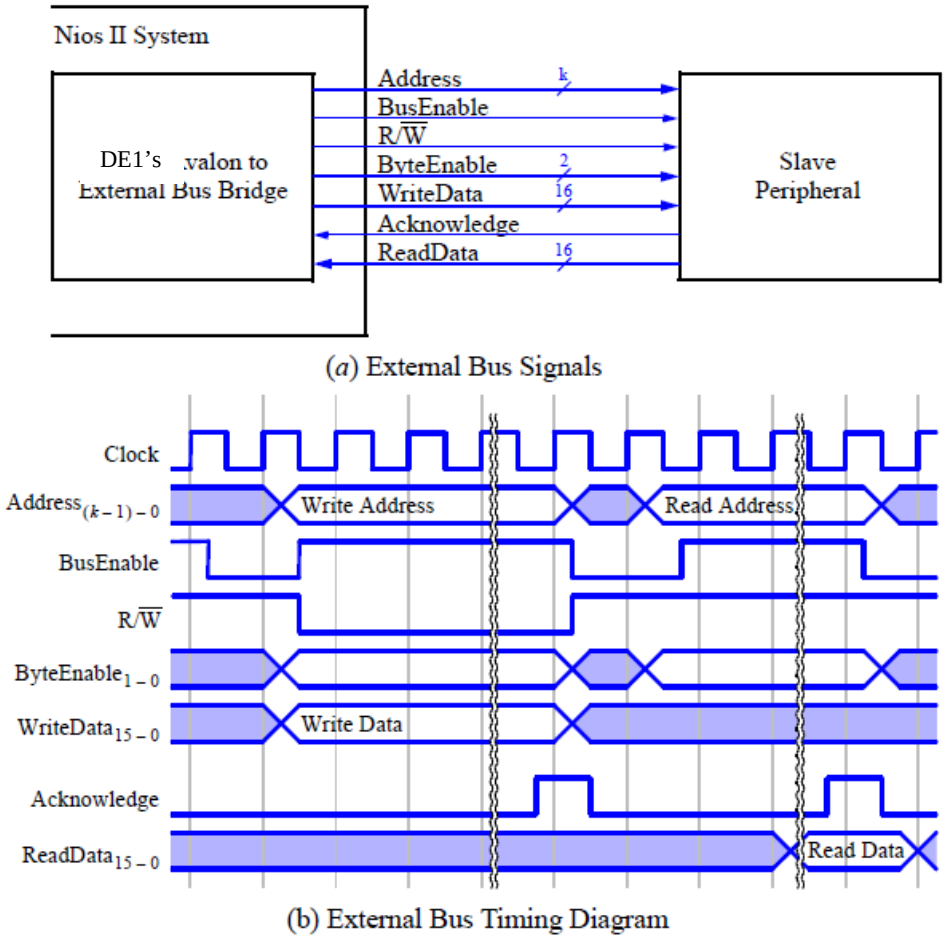

Figure 1: Avalon to External Bus Bridge signals

Part 2: Sharing a bus

In Part 1, a system was designed where the Nios II processor was the bus master that communicated with a slave connected to the Avalon Bus via a bus bridge. In this part of the exercise, we will create a system that has two bus masters (two Nios II processors) connected to one slave via two bridges. Since both bus masters may request access to the slave at the same time, it is necessary to provide an arbitration circuit which will grant access to the slave peripheral to only one master at a time.

In an embedded system where there is more than one processor, sometimes the two processors are required to communicate with each other. One way to provide this communication capability is to use a memory module that is accessible by both processors to implement what is often referred to as a mailbox structure.

Each processor can write a message into its outgoing mailbox located in the memory module. The other processor can read this message by reading from its incoming mailbox (aka the previous processor's outgoing mailbox). To help control when a valid message is present in the mailbox, a "flag" can be implemented. To implement the flag, each mailbox will have mailbox status associated with it. For example, assume we have two processors A and B where processor A wishes to send a message to processor B. Processor A will write its message into its outgoing mailbox and then assert the mailbox status flag. Processor B is monitoring processor A's mailbox status flag and when it sees it asserted, it knows that a valid message from processor A is in the mailbox. At that time processor B will read the message and then de-assert processor A's mailbox flag whereby processor A now knows processor B has read the message. The generic form of this interface is shown in the figure below

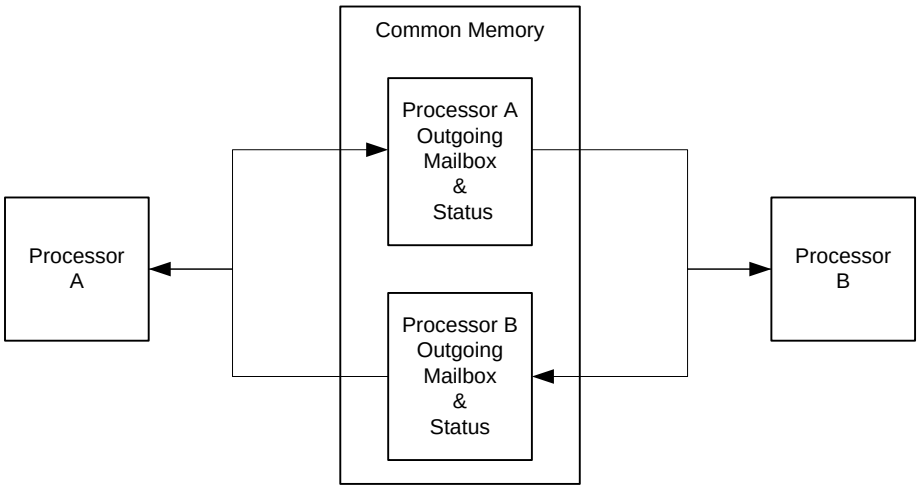

Figure 4: Block Diagram of Generic Mailbox Interface

For this lab, we will build the system shown in figure 4 using the external RAM created in Part 1. Each processor can access the RAM by means of the Avalon to External Bus Bridge, as was done in Part 1. Since there are two processors, there will be two instances of the Avalon switch fabric and two Avalon to External Bus Bridges as shown in 5 below. The Arbiter circuit must handle the read and write requests from both processors, allowing only one request to be serviced at any time.

The RAM_Controller is required for this lab. Since the RAM_Controller will perform the same function performed in Part1, the circuit previously designed can be re-used.

A connection to the host computer is provided by incorporating a separate JTAG UART interface for each processor. These interfaces are connected through the JTAG Hub to the USB-Blaster interface as shown in 5 below.

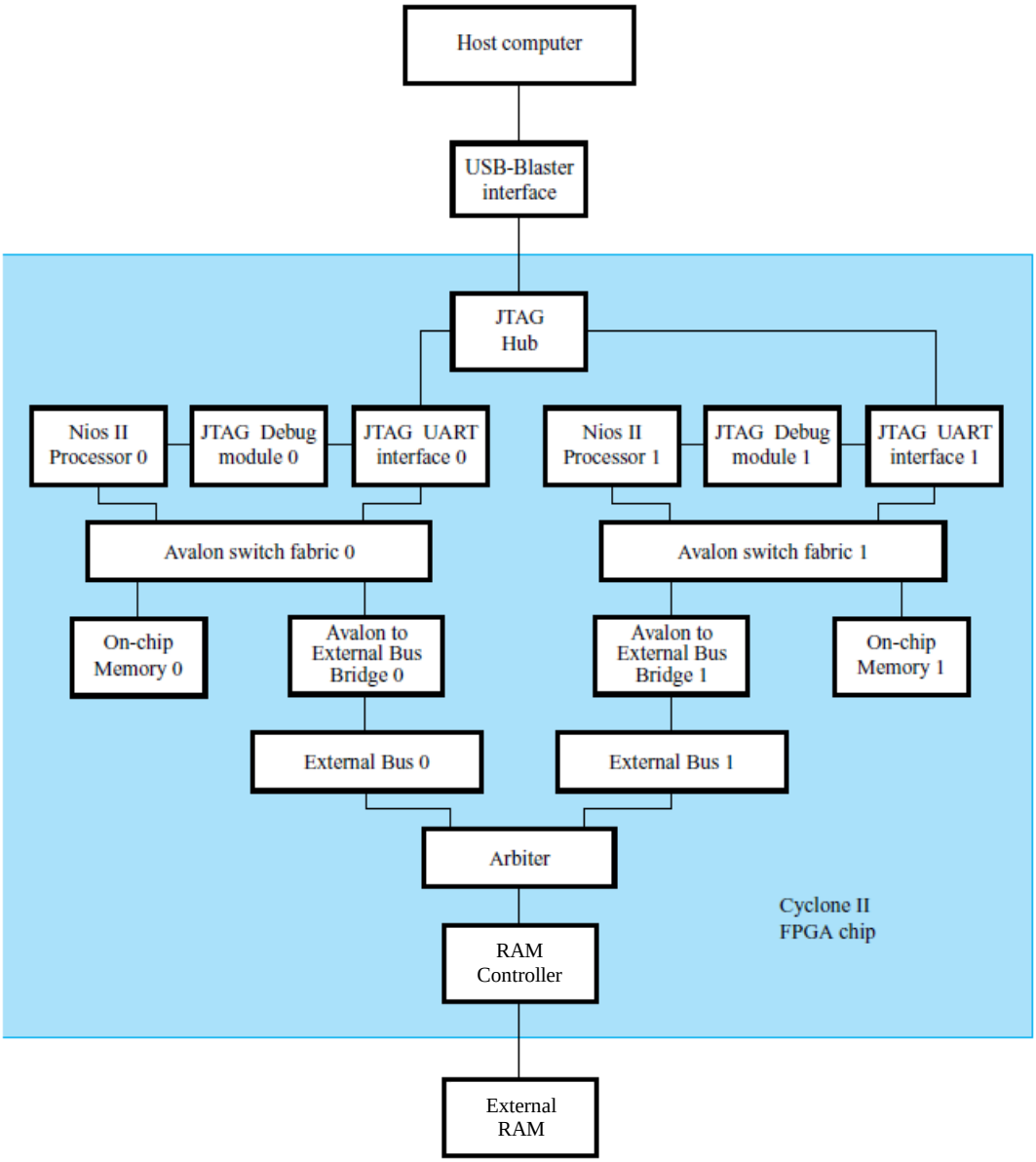

Figure 5: High-level Block Diagram of System

To illustrate the message passing between the two processors, a simple application that resembles a "chat room", will be implemented. We will implement and debug each processor's program using the Nios II Software Build Tools for Eclipse program. This means two debug sessions will be activated, one for each processor. An application program will allow the user to type a message using one Debug Client and this message will appear in the terminal window of the other Debug Client.

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 **Arbitration_Demo/Quartus/part2**.

2. Use QSYS to generate the desired circuit, called **nios_system**, which contains two Nios II processor designs where each design is comprised of:
    - Nios II/e processors – you must use the e version
    - On-chip memory – select the RAM mode and the size of 16 Kbytes
    - JTAG UART with interrupt
    - Avalon_to _external_bus_bridge – select 16 bit data width and the address range of 2 Kbytes. Connect the bridge to the Nios II's data master port.
    - System ID Peripheral with unique system ID (shared by both Nios II processors)
3. Rename the Avalon to External Bus Bridge components to **avalon_bridge0** and **avalon_bridge1**. Use the same names for the export.
4. Each Nios II processor will have its own JTAG UART, on-chip memory and Avalon Bridge. Rearrange the components so that the associated components for the Nios II processor are next to each other.
5. Connect the associated components together. Only the clock components' clock and reset go to all components. Be sure to connect to the appropriate debug_reset signal. Your final system should look similar to figure 6 below. Notice that the clock component is not shown completely and the sysid component is collapsed to show how the Nios II processors are connected.

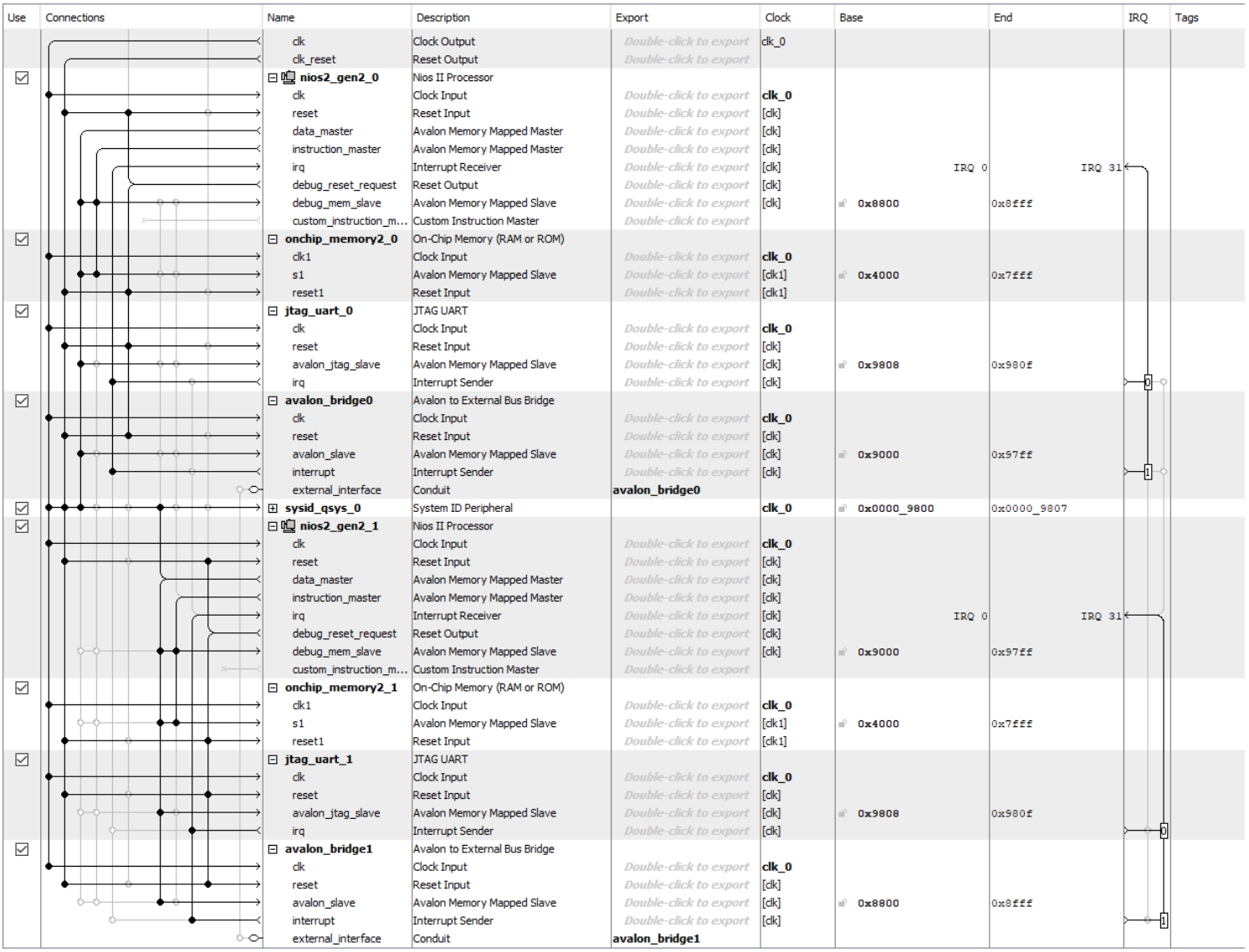

Figure 6: Nios II system specified in QSYS

6. From the system menu, select **System > Assign base Addresses**.
7. Generate the system and return to the Quartus software. Add nios_system.qip to the project
8. Add the **Arbitration_Part2.vhd**, **RAM_controller.vhd** and **arbiter.vhd** files to the project. Copy the three external_RAM support files from Part1 to the Part2 folder and add external_RAM.qip to the project.

Complete the Arbiter module using a state machine which handles transfers from both processors. Your Arbiter should provide a scheme for choosing which request to respond to first if both processors request transfers in the same clock cycle. It should also provide a rotating priority to make sure one processor does not get starved if both processors continue to make a request at the same time. A template for the bus arbiter VHDL file is provided to help you get started.

9. Import the pin assignments and compile your Quartus Project.
10. Program and configure the DE1-SoC board to implement the generated system.

The system is now ready to implement a “chat room”

1. Open two different workspaces of the Nios II Software Build Tools for Eclipse program. You could complete the following steps in one workspace, but it is less confusing to open two. Create a new Application and BSP for each Nios II processor. Name the first App **Arbitration_Nios0_APP** and name the second App **Arbitration_Nios1_APP**.

   NOTE: Remember to select the correct Nios II processor when you build each App.

2. Copy **Arbitration_nios0.c** and **Arbitration_nios1.c** to the appropriate app folders in Eclipse.
3. Open the BSP editor and check ‘enable_small_c_library’ and ‘enable_reduced_device_drivers’. Generate the BSP and copy the system.h file from the bsp folder to the app folder. Repeat for the second processor.
4. Build your projects for each processor.
5. Before we can debug the system we need to configure the debug session. For each of the apps, do the following: Choose Debug Configurations from the tool bar. In the configuration window, right-click on the **Nios II Hardware** and click **New**. In the **Name** field assign a description name for the configuration (ex: **Arbitration_nios0**). In the **Project name** field select one of the projects. Verify that the correct Project ELF file was also selected.
6. Next select the **Target Connection** tab. Under Connection and click on the **Refresh Connections** button. Select the Nios II processor that you want to run the application on under the **Processor** header.
7. Next select the processor’s associated jtaguart for the **Byte Stream Device**. Click **Apply** when done.
8. Repeat the setup of the debug session for the other application project and processor.
9. Select each project and debug each as Nios II Hardware. In the debug session you should see a Nios II Console.
10. Run both programs and demonstrate the chat room capability. Remember that the terminal window for each processor is activated by clicking on it. Notice that the “sending” console will echo the characters after “Enter” is pressed. This was done for debugging purposes. If you do not like this feature, you can edit the C program to take it out.

    Demonstration: Submit a video (or demo in lab) of step 10.