SE 2DA4 Lab 5:

SDRAM Controller Interface

First lab Week of: Nov. 19, 2018
Prep Due week of: (8:40/14:40), Nov. 26, 2018
Demo Due Week of: (11:20/17:20), Nov. 26, 2018
Assignment due in class: N/A

Announcements:

Note 1: All lab due dates/times are for your scheduled lab in the indicated week.

Note 2: Make sure that you use Quartus Prime Standard version 17.1 for this lab. The instructions in this lab manual are for this version.

Note 3: As stated in the course outline, you can not MSAF an entire lab, only a lab session. Yor accomodation would be the opportunity to make up the missed lab session. This can be done by attending another lab section (space permitting) in the following week. For the final week of lab 5, you will either have to attend another lab section that week, or the make up lab section on Tuesday, Dec 4, 2018, 2:30PM - 5:20PM.

For this lab: after you have demonstrated your lab to a TA , you must e-mail (as per instructions in lab 1) the final versions of your Verilog files (the ones demoed to the TA) as attachments to the following e-mail address: `rlta1@cas.mcmaster.ca`

Assignment:

For this lab, there is **NO** attached assignment.

Part 1: Preparation

As your preparation, hand in the Verilog code, and the simulations results for the synchronous dynamic random access memory (SDRAM) controller only (don't simulate the processor). Also, hand in the preparation questions from Part 2. This lab is not difficult, but it is time consuming. You need to work through the pre-lab tutorial and read the corresponding reference materials well before the first lab period so that you will be able to start preparing for the actual lab.

Background

Start by reading Section B.9 in the textbook on static RAM (SRAM) as well as the corresponding lecture slides on read only memory (ROM), RAM, and dynamic RAM (DRAM) (slide set 8).

A good reference for the C-programming language (although likely a bit out of date) is: `C Programming Language (2nd Ed),` by Brian W. Kernighan and Dennis Ritchie, Prentice Hall, 1988. There are also

many resources online.

The following documents will be used during the lab. You can find most of them in the `ref` subdirectory once you uncompress the `lab5.zip` file you downloaded from the course website.

- Prelab5.pdf
- Introduction_to_Qsys.pdf
- TutMatfNIOSIISBTfEclipse.pdf
- SignalTap.pdf
- Using_the_SDRAM_DE1-SoC.pdf

Platform Designer (Qsys) Tutorial

In the main part of this lab (the Creating SOPC System part that follows), we will be using the Intel Platform Designer (formerly Qsys) integration tool to create a NIOS II hardware system on the FPGA on the DE1-SoC board. This system will connect a NIOS II soft processor to the SDRAM chip on the DE1-SoC board through an Avalon Memory Mapped Interface, an SDRAM controller component and other components. In the lab, you will complete a Verilog module for the SDRAM controller, then you will create a custom Platform Designer component for SDRAM controller based on the Verilog code you have completed. You will then use the custom component to create the hardware system on FPGA. You will write a C program to test the system and your SDRAM controller. You will also use Alteras Signal Tap Logic Analyzer to view what your chip is doing as it executes. Signal Tap Logic Analyzer is essentially a logic analyzer (like an oscilloscope but for many digital signals at once, plus capable of triggering on signal events and showing what happened at that point).

To prepare you for this, you will first work through the tutorial in file `Prelab5.pdf` from the `lab5.zip` file. This tutorial will take you through adding a simple peripheral to the NIOS II processor, loading the resulting SOPC onto the DE1-SoC boards, compiling a simple C-language program to test the system, and then running the result on the DE1-SoC board. You will then use the Signal Tap tool to examine your peripheral as it operates.

NOTE: You must work through this tutorial well before the first lab period as it will give you the necessary background to do the actual lab! You must do everything in the tutorial that doesn't require the actual boards, an d make sure you read through the entire tutorial, ven parts that require the board, before the lab starts.

Demonstrate the working switch/LED system from the above tutorial to a TA as well as the Signal Tap analysis of the blinking LEDs.

Part 2: System Description

In this lab, you will design a controller circuit for the SDRAM memory chip on the DE1-SoC board. This circuit connects the SDRAM chip to the Avalon interconnect fabric and performs some functional control for SDRAM operation. Because of the complexity of how SDRAM works, writing a Verilog SDRAM

controller from scratch is not feasible within our lab time allocation, so we will use an SDRAM controller module from Altera. The module is `DE1_SoC_QSYS_sdram.v`, which provides all the essential and basic functions to interface with an SDRAM chip. To make the Verilog ports and wire names more meaningful, a wrapper module is required. The wrapper module is `SDRAM_Controller.v`. You need to complete this module by yourself. You should spend some time to read the code of `DE1_SoC_QSYS_sdram.v` to understand the SDRAM controller. Both files can be found in the archived project file `lab5.qar` located in the `data` subdirectory of the `lab5.zip` file. Just double-click on the `lab5.qar` file to open it in Quartus and to extract the files.

You will use `SDRAM_Controller.v` to create a custom Platform Designer component, and use this custom component for your hardware system design. After you download your designed system to your DE1-SoC board, you will need to complete a piece of C code (see: `data/mem.test.c`) and use it to test the functionality of the controller. Finally, you will observe the operation of the Avalon interface to the SDRAM controller using the Signal Tap Logic Analyzer.

Please note that the purpose of the lab is not to test your C coding skills. If you are having trouble with your C code, feel free to ask for help from other students and lab TAs. The included mem.test.c should give you a good starting point.

The system you are building is shown in Figure 1. The components are described in the following sections.

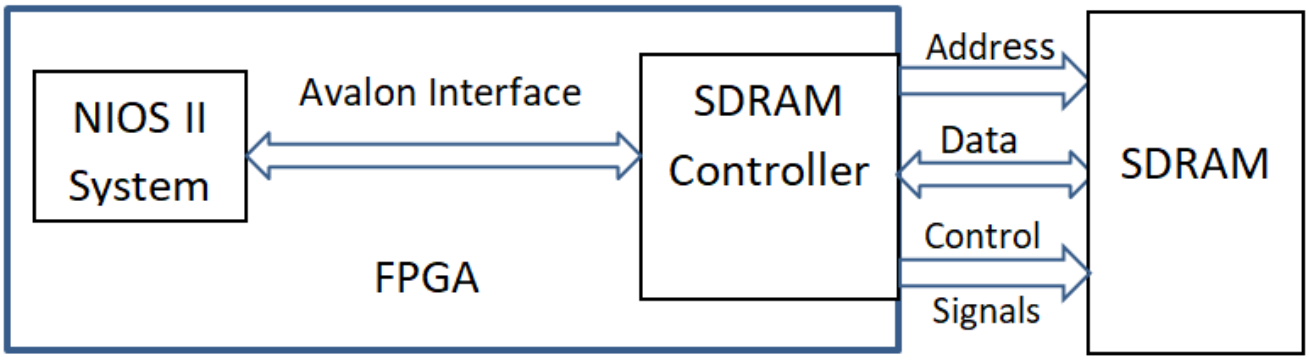

Figure 1: System Block Diagram

The SDRAM used in this lab

The SDRAM chip on the DE1-SoC board is a 16 bit synchronous DRAM containing four banks of 8M 16-bit words, providing a total size of 64M bytes of memory. In each bank, the 16-bit memory words are arranged in an array of 8K rows and 1K columns. The function block diagram of the SDRAM chip is shown in Figure 2.

The signals between blocks

The signals between blocks of the system are shown in Figure 3.

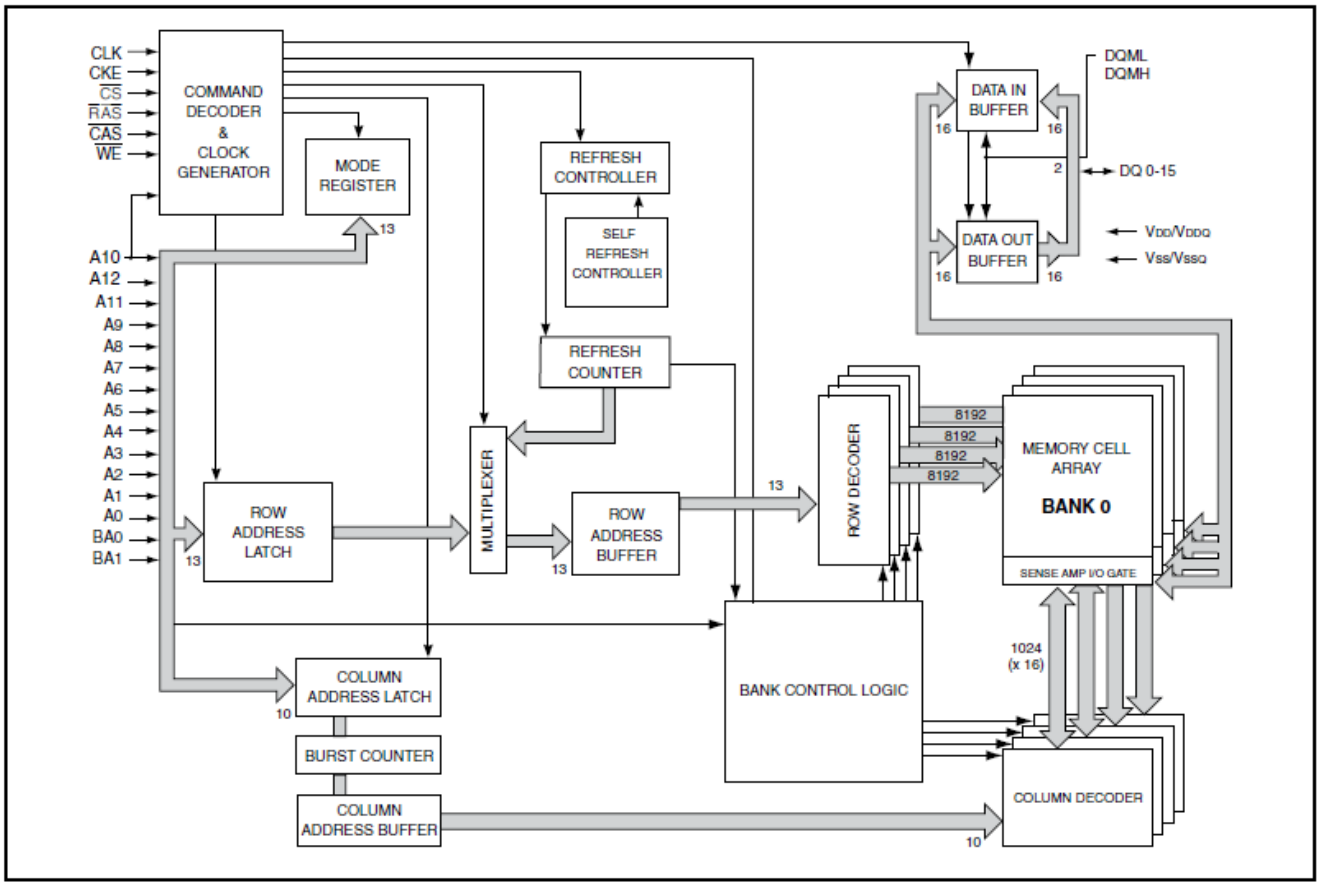

Figure 2: SDRAM Function Block Diagram (from datasheet for IS42S16320D)

On the Avalon interface, the `address` signal is 25 bits, which is capable of addressing 32M bytes of the 16-bit words. Combined with the `byte_enable_n` signal, the entire 64M bytes of SDRAM can be accessed. For the SDRAM, when working with the signal `DRAM_RAS_N`, the the 13-bit `DRAM_ADDR` signal is enough for addressing the 8K rows of memory words (16-bit) in one bank. When working with `DRAM_CAS_N`, the columns of the SDRAM words array in one bank can be referenced by 10 bits of the `DRAM_ADDR`. The `DRAM_BA` signal is used to specify which bank will operate. `DRAM_LDQM` and `DRAM_UDQM` assist in making byte reading and writing possible.

Other signals in the system are (NOTE: the "**_N**" or "**_n**" stands for active low for signals) shown below.

The signals for the SDRAM:

DRAM_DQ[15:0]: Data I/O pins, which are used for both reading and writing. In the SDRAM controller, `DRAM_DQ` may get values from the signal `write_data` or may drive wires for `read_data`, depending on other input signals, for example `write_n` or `read_n`.

DRAM_ADDR[12:0]: Address to be read/written. The address refers to the full 16-bit word, not individual bytes. Combined with signals including `DRAM_BA, DRAM_RAS_N`, and `DRAM_CAS_N`, the SDRAM controller will decide in which bank, on which row and in which column the word will be accessed.

DRAM_BA[1:0]: Bank select address. It defines which of the four banks the SDRAM command applies to.

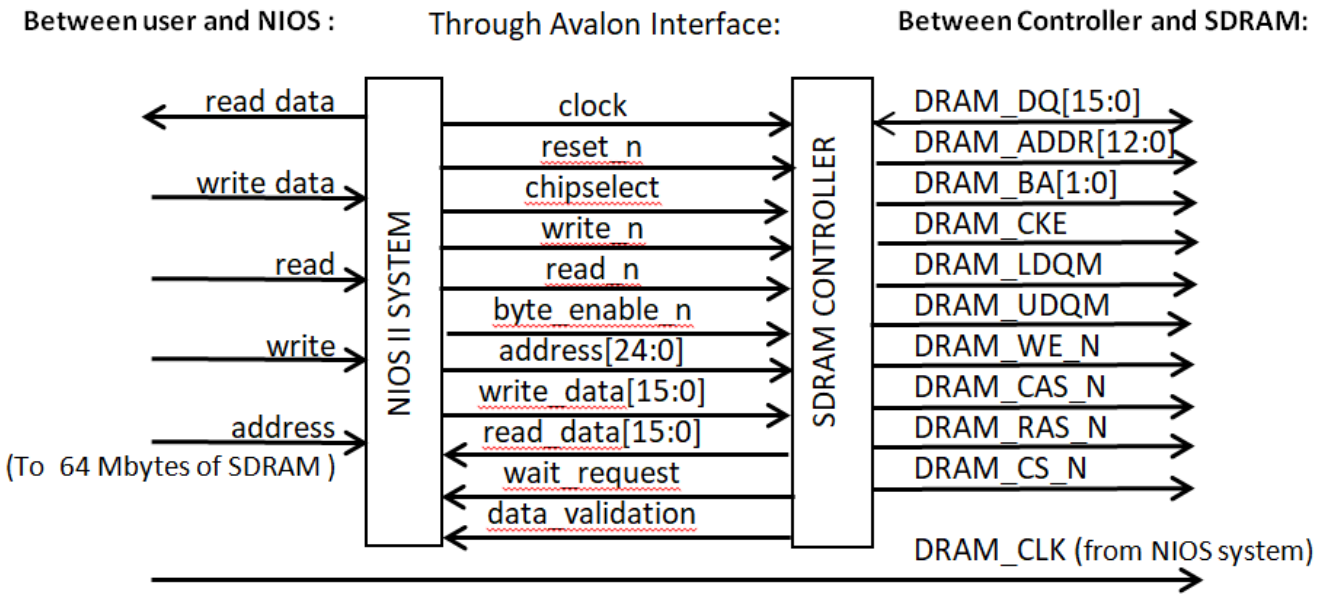

Figure 3: Signals between blocks

DRAM_CLK: The master clock input for the SDRAM. Please note, due to the clock skew on the DE1-SoC board, this signal needs to lead the NIOS II system clock by three nanoseconds. This signal will be provided by a PLL circuit from the NIOS II system.

Except for the signal `DRAM_CKE`, all inputs to the SDRAM are acquired in synchronization with the rising edge of the signal on this pin.

DRAM_CKE: The pin to determine whether the `DRAM_CLK` input is enabled.

DRAM_LDQM: SDRAM lower byte data mask. It controls whether the lower byte of the input or output buffer is enabled. This signal is active low. When `DRAM_LDQM` is high, the lower byte data buffer is disabled.

DRAM_UDQM: SDRAM upper byte data mask. It controls whether the upper byte of the input or output buffer is enabled. This signal is active low. When `DRAM_UDQM` is high, the upper byte data buffer is disabled.

DRAM_WE_N: SDRAM write enable pin. In conjunction with other signals, it starts the write operation.

DRAM_RAS_N: SDRAM row address strobe. Combined with other signals, it can be used to operate on a row of memory words in one bank.

DRAM_CAS_N: SDRAM column address strobe. Combined with other signals, it is used to operate on a column of memory words in the I/O buffer.

DRAM_CS_N: DRAM chip select. This pin controls whether or not the SDRAM chip is enabled.

For more details of functions of the SDRAM signals, please refer to the truth tables in the SDRAM datasheet for the DE1-SoC boards (ref/SDRAMDataSheetDE1-SoC.pdf).

Some signals on the Avalon MM interface:

chipselect: Other signals on the Avalon interface are only valid when this signal is active.

write_n: Indicates that the operation to be performed is write.

read_n: Indicates that the operation to be performed is read.

Byte_enable_n: Enables a specific byte of data. For 16-bit data words, the value 00 enables both the upper byte and lower byte. 01 enables the upper byte, and 10 enables the lower byte.

address[24:0]: Used for word addressing for the entire memory. The value of the address needs to align to the data width. The `byte_enable_n` signal is needed to specify which byte within a data word will be written to or read from.

write_data[15:0]: Specifies the data to be written to memory when a write operation is requested.

read_data[15:0]: This is the signal that the slave device should use to provide data when a read operation is requested.

wait_request: When further read/write requests can not be processed, the slave device requests the master device to wait.

data_validation: Asserted by the slave device that the data pins contain valid data.

For more details of signals on the Avalon interface, please refer to the Avalon interface document (ref/mnl_avalon_spec.pdf).

Your SDRAM controller should provide any circuitry needed to provide an interface between the two groups of signals outlined above. Basically, when a read operation is underway, you should set Avalon's `read_data` signals equal to the SDRAM's `SDRAM_DQ` signals, otherwise set `read_data` to high-impedance state (Z). When a write operation is underway, then set the SDRAM's `SDRAM_DQ` signals equal to Avalon's `write_data` signals. When neither a read or write operation is underway, set the SDRAM's `SDRAM_DQ` signals to high-impedance state (Z).

To get started, use the archived project file `lab5.qar` located in the `data` subdirectory of the `lab5.zip` file. The starter project contains two uncompleted Verilog files: `lab5.v` and `SDRAM_Controller.v`. The top-level entity module is `lab5.v`. You need to complete the `SDRAM_Controller.v`, use it to create a custom component, and use this custom component to create the SOPC system, and then instantiate the SOPC system in `lab5.v`. After completing the `SDRAM_Controller.v`, ensure that Quartus can compile it without errors. To test this, temporarily make the `SDRAM_Controller.v` the top-level module in your hierarchy (select it and then go to: `Project | Set as Top-Level Entity`). Once this is done, do not forget to make `lab5.v` the top-level module again.

As part of your preparation, also hand in the answer to the following questions:

1. The SDRAM chip we are using contains 32M (M means $2^{20} = 1,048,576$) memory locations, each location corresponding to 16 bits of data. This means the chip contains 64M bytes of data. When you are writing your C program, memory locations will access byte locations (i.e. address 0x0000 will be the first byte and address 0x0001 will be the second byte). What will be the lowest byte address for accessing the SDRAM (assume the base address is 0x0000 for our purposes, the SOPC

will assign a different base address as part of the design process as discussed later in the lab). What will be the highest byte address?

2. The Avalon interface will convert from byte addresses to the word addresses used by our SDRAM. Considering that our SDRAM chip accepts word addresses, what addresses should the chip see on its input when the processor requests data from the lowest and the highest byte in the memory?

You need a C program for the NIOS II system that tests the behaviour of your memory controller. The `mem_test.c` file located in the `data` subdirectory of the `lab5.zip` file should do the trick, but you may need to adapt it a bit. You can assume that the starting address of the SDRAM memory module is defined as a constant: `SDRAM_CONTROLLER_0_BASE`. You should perform the following tests:

- Write various characters (char = 1 byte) to all locations in the memory. After all the locations have been written, read the locations to ensure that the value read is the value that was written.
- Write various short words (short = 2 bytes) to all locations in the memory. After all the locations have been written, read the locations to ensure that the value read is the value that was written.
- Write various integer words (int = 4 bytes) to all locations in the memory. After all the locations have been written, read the locations to ensure that the value read is the value that was written.

In case any of these tests fails, you should print an error message. You can use the standard printf function for this.

The simplest way this can be done is to use a simple loop and write the loop iterator into the memory. Then create another loop with the same iterator, and ensure that the number read from the memory is equal to the iterator. Keep in mind the widths of data. For instance, the iterator for char needs to be more than that for integer to cover the whole address range for the memory. Do not forget to `#include "system.h"` in your code. There are NO other special considerations that you have to worry about; the C compiler for Nios accepts standard ANSI C code.

Writing to and reading from the entire 64M bytes SDRAM on the DE1-SoC boards will take several minutes. In the lab, you may start with testing parts of the SDRAM. Once you are sure the system works fine, then start to test the entire SDRAM.

Part 3: Creating SOPC System

WARNING: Make sure to create your project in a folder that does not contain any spaces in its name! Otherwise, NIOS II IDE may not be able to compile software for your system. For example, do NOT create your project on the Desktop or My Documents, because both are mapped under the folder "Documents and Settings" whose name contains spaces.

1. Open the project you worked on for preparation.
2. Complete your `SDRAM_Controller.v` if you have not done so. Set it as Top-Level Entity and compile your project . Make sure the module is correct. (Remember to make the `lab5.v` the Top-Level Entity again once done)
3. Create a custom Platform Designer component:

- Start Platform Designer (under the Quartus `Tools` menu), and begin creating a new component by clicking `File | New Component...` .
- In the window that opens, define your own component for the system by specifying both the "Name" and the "Display Name" as "SDRAM_Controller". Click on "Next".
- Skip the Block Symbol tab by clicking "Next" to go to the Files tab.
- In the Files tab, you will need to specify the file that describe your component. Click on the "Add File..." button under "Synthesis Files" section to browse and select the module `SDRAM_Controller.v`. Then click "Open" and click on the "Analyze Synthesis Files" button. Any errors found by "Analyzing Synthesis Files" need to be fixed and the code re-analysed. Once no errors are present (should see message "`Analyzing Synthesis Files: Completed successfully`" or "`Analyzing Synthesis Files: Completed with warnings`"), the next step is to specify the types of interfaces that are used by the component.
  Please ignore any error messages that may appear in the Message panel at the bottom of the Component Editor window at this step. Those errors will be fixed as you proceed through the following steps.
- Skip the Parameters tab, click "Next" and go to the Signals & Interfaces tab.
- The Platform Designer must be told which signals in your module correspond to which signals of the interface to the Avalon interconnect. By default, the Platform Designer attempts to recognize the signals by their names, but it may not recognize all of the signals. You need to make the following changes:
  (a) Create 10 Conduit interfaces by clicking "<<add interface>>", which is at the bottom of the "Name" panel of the Component Editor window, and move the 10 signals whose names start with "DRAM_ " (if you followed the naming schemes of `SDRAM_Controller.v`) from the "Avalon_slave_0" category to the 10 conduits.(Figure 4)

  Conduits are Avalon interfaces which are normally used for signals that do not fit into the other defined standard types. Conduit signals can be exported and be used to make external connections to off-chip devices. In our case, the SDRAM is an off-chip device. Signals with the DRAM prefix will be used to connect to the SDRAM.
  (b) Click the signal `data_validation` in the left panel and change its "Signal Type" to `readdatavalid` in the right panel.
  (c) Click the signal `wait_request` and change its "Signal Type" to `waitrequest`.
  (d) Click the signal `write_data` and change its "Signal Type" to `writedata`.
  (e) Change the "Signal Type" of all the signals starting with "`DRAM_`" to `export`. You need to input `export` manually in the "Signal Type" box.
  (f) Click the interface of `clock_reset` in the right panel, change its "Type" to `Clock Input` and its "Name" to `clock`. Also click the `clock` signal under this interface and change its "Signal Type" to `clk`.
  (g) Click the interface `reset` and change its "Associated Clock" to `clock`.
  (h) Change all of the 10 conduit interfaces' "Associated Clock" to `clock` and the "Associated Reset" to `reset`.
  (i) **Check all of the signals to make sure their "Signal Type" and "Direction" are correct**. Incorrect direction settings for signals can stop the SDRAM controller from working properly.

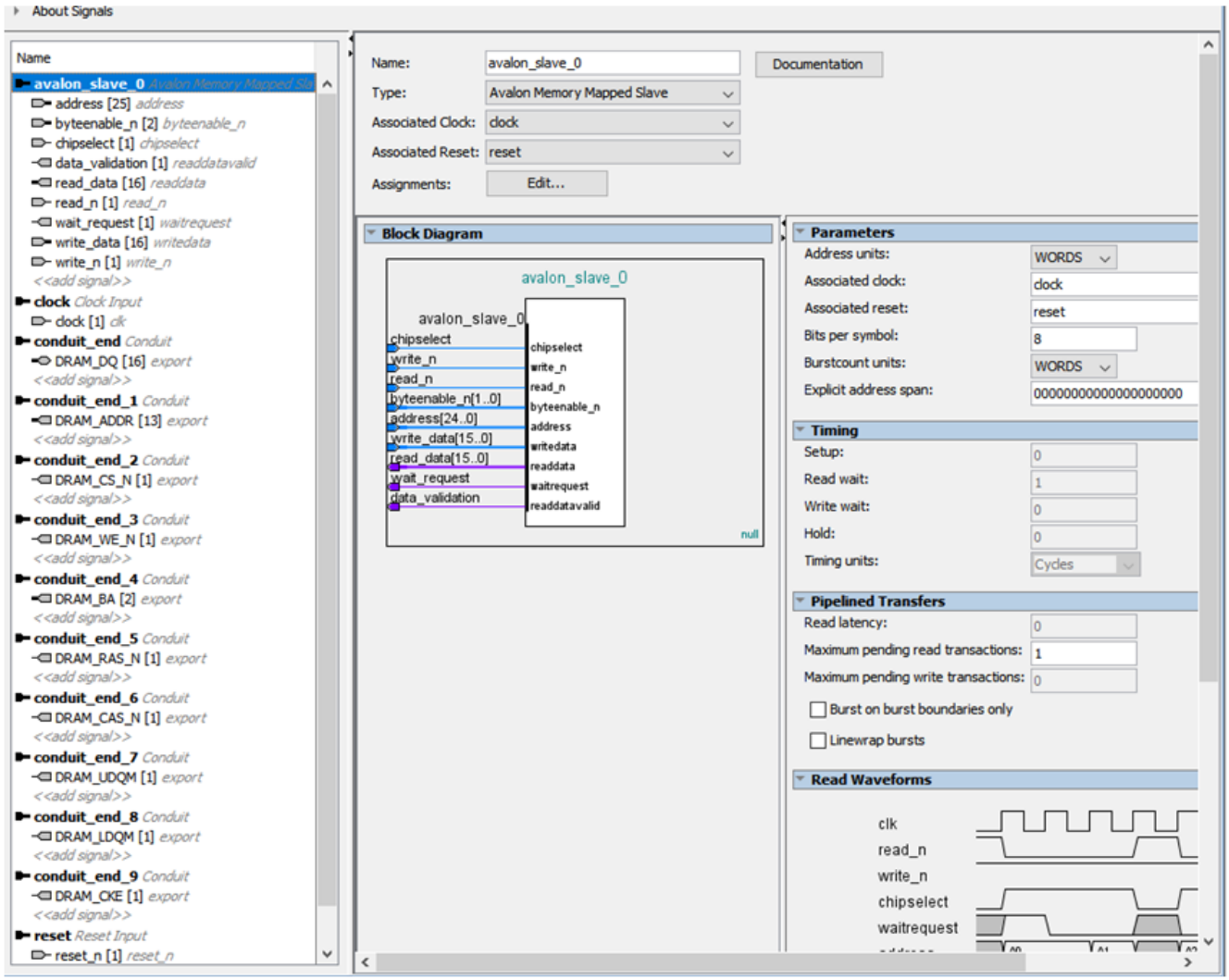

Figure 4: Creating Conduit Interfaces

(j) Click the interface Avalon_slave_0. Set its "Associated Clock" to `clock`. Set its "Associated Reset" to `reset`. Under the section "Pipelined Transfers", set the "Maximum pending read transactions" to 1.

(k) By now, all of the errors in the Component Editor should be fixed. If any errors remain, you need to fix them before proceeding to the next step. Click button "Finish" to close the Component Editor and save the new component. Now on the left side of the Platform Designer, the component list contains the `SDRAM_Controller` component.

4. Return to the Platform Designer. Check to make sure that the "Clock Frequency" of the component `clk_0` is 50MHz.

5. Add a NIIO II/f processor (under `Processor and Peripherals | Embedded Processors | NIOS II Processor`) with default settings.

   For this stage (and the following stages), there will be some error messages showing in the Message window. Do not worry about them at this point, as they will be fixed as you work through the following steps. If there are still errors before generating the NIOS system, you will need to go back and fix them.

6. Add On-chip Memory. You can find it under `Basic Functions | On-Chip Memory | On-Chip Memory(RAM or ROM)`. Set its "Total memory size" to 128K, and leave the other settings as default. DE1-SoC boards have enough on-chip memory for our lab project. Setting the total memory size to 128K bytes will allow us to use the `printf()` function without setting the compiler for NIOS II SBT for Eclipse to use the small C library.

7. Add JTAG UART. You can find it under `Interface Protocols | Serial`. Use the default settings.

8. Add component SDRAM_Controller.

9. Due to the clock skew that will happen on the DE1-SoC boards when using the SDRAM chip with CLOCK_50, reading from and writing to the SDRAM is not always correct. For proper operation, it is necessary for the signal DRAM_CLK to lead the NIOS II system clock, CLOCK_50 of the DE1-SoC, by 3 nanoseconds. This can be achieved by using a PLL circuit.

   The Intel FPGA University Program provides a clock IP core that can help users easily do this. We will use this clock IP core by adding it from: `University Program | Clock | System and SDRAM Clocks for DE-series Boards`. In the settings for this IP core, select "DE1-SoC" from the DE Board drop down list.(Figure 5.)

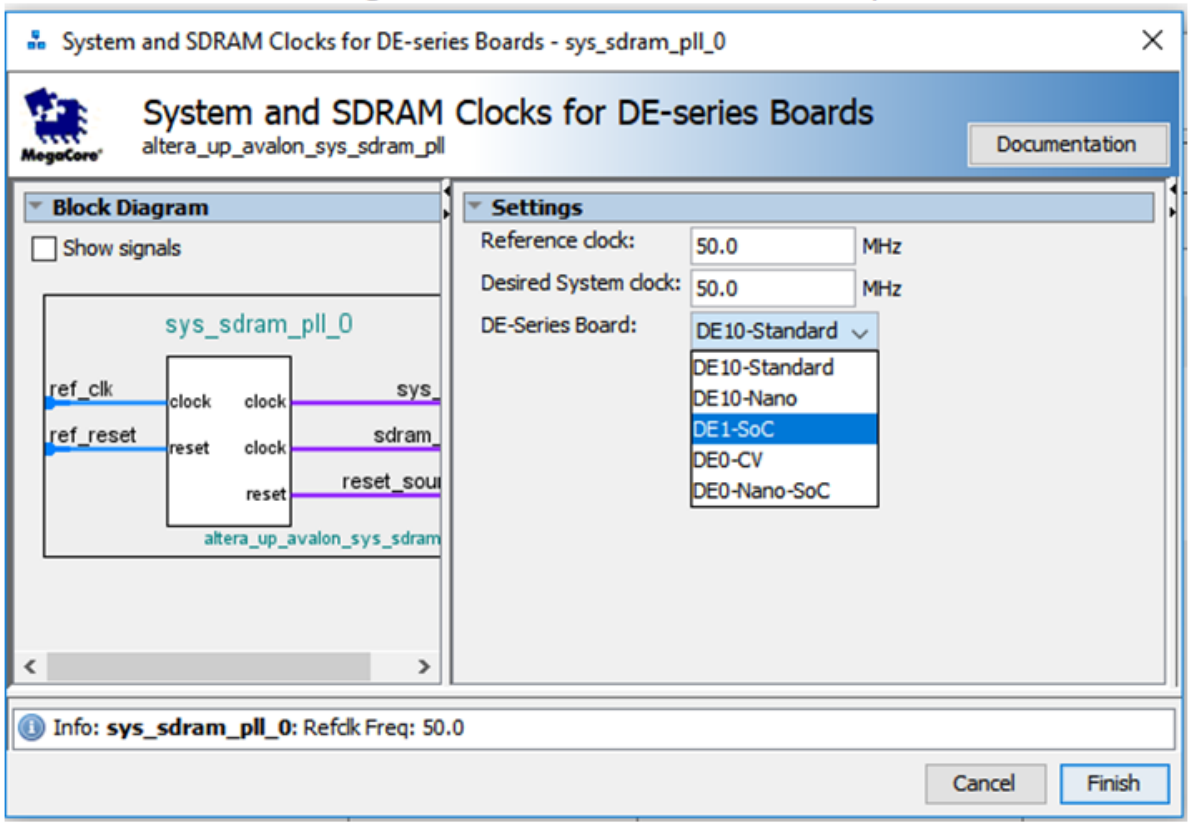

Figure 5: System and SDRAM Clocks

10. In the Connection column, make the necessary connections by clicking on the appropriate empty circles to change them to solid circles. (Figure 6.)

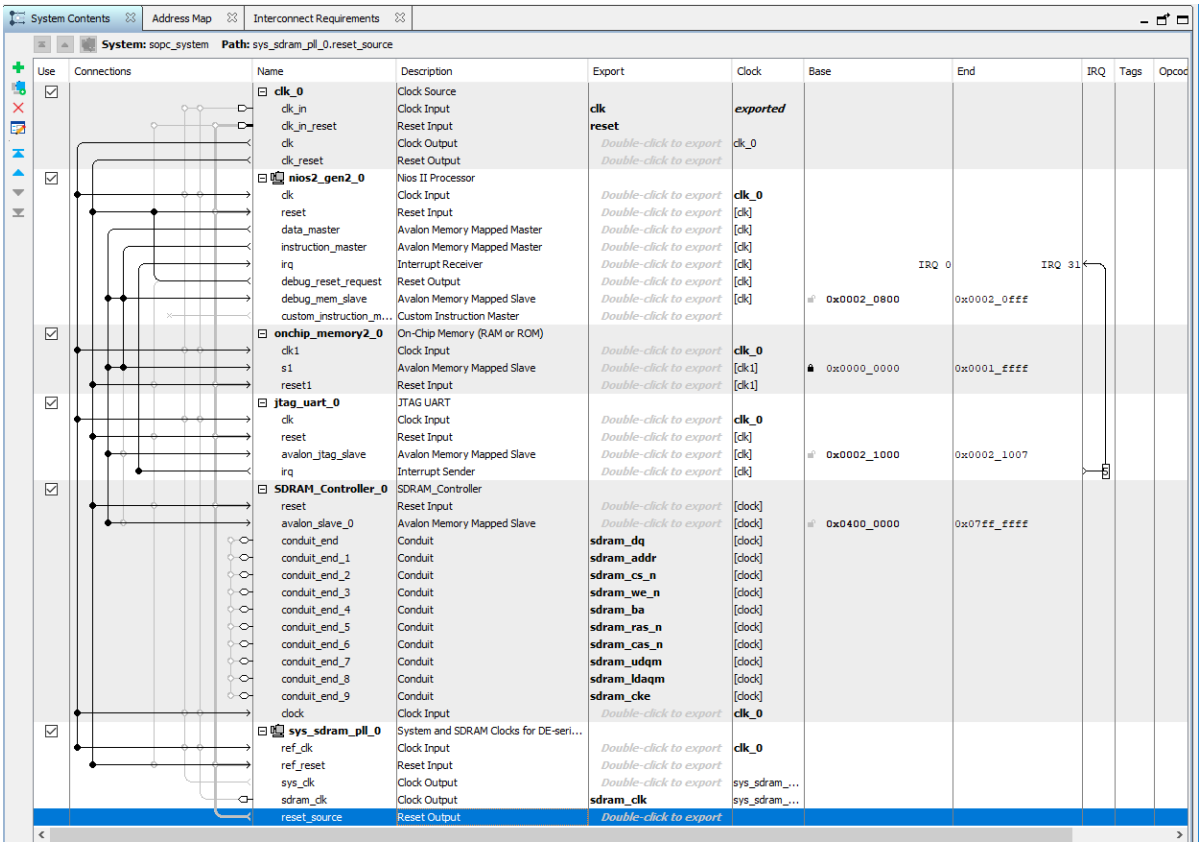

Figure 6: Components Connection, IRQ and Export

11. In the Export column, double click the corresponding items to export the 10 Conduit signals of the SDRAM_Controller components and give them meaningful names. Giving these exported conduit signals meaningful names will help to identify and connect ports or wires when instantiating the NIOS system modules created by the Platform Designer. To find out which conduit interface corresponds to which signal, you can right click the component to open the Component Editor window for it. In the Block Diagram (or Block Symbols) tab, check the box "Show Signals" to see the conduits-signals correlation.

12. In the Export column, export the "`sdram_clk`" signal of the `pll` component (Figure 6).

13. In the IRQ column, click the IRQ number and change it from 0 to 5.

14. In the Base column, click the address for the `onchip_Memory` and change its base address to 0x0000 0000, and click the lock icon beside the address to lock it.

15. From the System menu, select Assign Base Addresses (`System | Assign Base Addresses`). This command automatically assigns addresses of slaves in the system so that they do not conflict with one another.

16. Right-click on the NIOS processor and choose Edit from the pop up menu. Click the Vectors tab in the NIOS II Processor setting window. Under both "Reset Vector" and "Exception Vector" select "`onchip_memory2_0.s1`" to set these two vectors to the on-chip memory. These settings specify where the processor starts execution on reset, and where the processor jumps when an interrupt/exception

occurs, respectively.

By now there should be no error messages in the Messages window. If error messages remain, they must be corrected before proceeding to the next step.

17. Click on `File | Save` or `Save As`. Give your NIOS system a name such as `sopc_system.qsys` and save it. Your system is now ready for generation. Click on the button "Generate HDL" at the bottom of the window to generate modules for the NIOS system.

Part 4: Completing Quartus Prime Project

1. Once the system generation is finished, you can leave Platform Designer open for your convenience to check or edit, or you can exit from it. Return to Quartus Prime, and add the newly generated files to the project via : `Project | Add/Remove Files in Project.....`

2. Browse to the project_folder/NIOS_system_folder/synthesis(for example: `z:/lab5/sopc_system/synthesis`), find the `.qip` file (for example: `sopc_system.qip`). Add it to your project. You can also add the `.qsys` file to your project for your convenience to edit your NIOS system. The `.qsys` file is in the project folder.

3. In the Quartus Project Navigator window, under the Files tab, click the ">" sign beside the newly added `.qip` file to expand the list of files it contains. Find the NIOS system Verilog module (example: `sopc_system.v`) and open it to examine its input and output ports. Depending on the names you have given to the exported signals when you created the NIOS system in Platform Designer, some of the port names may or may not be meaningful. Be careful with these ports when you instantiate this module in `lab5.v`. You may need to go back to Platform Designer to see the correlation between the signals and the ports by looking at the Block Diagram.(In Platform Designer, right click a component and select the Edit command, then check the box for "Show signals" to view the signals-ports correlation.)

4. Instantiate the NIOS system module in `lab5.v`, and complete `lab5.v`.

5. Set up `lab5.v` as the Top-Level Entity. Assign pins for the project. Compile the project. Download the project to DE1-SoC board. Your hardware is prepared to test.

6. Show the compilation result indicating sucessful status to one of the TAs.

Part 5: Set up the Signal Tap Logic Analyzer

1. In Quartus Prime, select `Tools | Signal Tap Logic Analyzer`.

2. Add all inputs and outputs of the "SDRAM_Controller" to be monitored by the logic analyzer. When you open the Node Finder (by double clicking on an empty space in the analyzer) you should select Filter: `Signal Tap:pre_synthesis`. You may input filter `*SDRAM_Controller:sdram_controller_0*` in the "Named" input box to narrow down your choices to the signals you actually need . Then click on the button "List". In the "Matching Nodes" panel, expand the signal group "`sopc_system:controller`", then further expand the signal group "`SDRAM_Controller:sdram_controller_0`". If you used all the same names as recommended in this manual, you should add the following signals in Signal Tap Logic Analyzer:

```
sopc_system:controller|SDRAM_Controller:sdram_controller_0|DRAM_CAS_N
sopc_system:controller|SDRAM_Controller:sdram_controller_0|DRAM_CKE
sopc_system:controller|SDRAM_Controller:sdram_controller_0|DRAM_CS_N
sopc_system:controller|SDRAM_Controller:sdram_controller_0|DRAM_LDQM
sopc_system:controller|SDRAM_Controller:sdram_controller_0|DRAM_RAS_N
sopc_system:controller|SDRAM_Controller:sdram_controller_0|DRAM_UDQM
sopc_system:controller|SDRAM_Controller:sdram_controller_0|DRAM_WE_N
sopc_system:controller|SDRAM_Controller:sdram_controller_0|DRAM_ADDR[12..0]
sopc_system:controller|SDRAM_Controller:sdram_controller_0|DRAM_BA[1..0]
sopc_system:controller|SDRAM_Controller:sdram_controller_0|DRAM_DQ[15..0]
sopc_system:controller|SDRAM_Controller:sdram_controller_0|chipselect
sopc_system:controller|SDRAM_Controller:sdram_controller_0|clock
sopc_system:controller|SDRAM_Controller:sdram_controller_0|data_validation
sopc_system:controller|SDRAM_Controller:sdram_controller_0|read_n
sopc_system:controller|SDRAM_Controller:sdram_controller_0|address[24..0]
sopc_system:controller|SDRAM_Controller:sdram_controller_0|read_data[15..0]
sopc_system:controller|SDRAM_Controller:sdram_controller_0|write_data[15..0]
```

3. Select `CLOCK_50` to be the clock for the system, as you learned in the tutorial. Set the analyzer to trigger only on high value of the chipselect signal. (A possible setup is shown in Figure 7.)

4. Save the Signal Tap configuration and enable Signal Tap for the project. Make sure that `lab5.v` is the top-level module in your system. Compile the project. Once the compilation is done, download the project to the DE1-SoC board. Now you have a hardware system containing the Nios II processor ready and waiting on the DE1-SoC board.

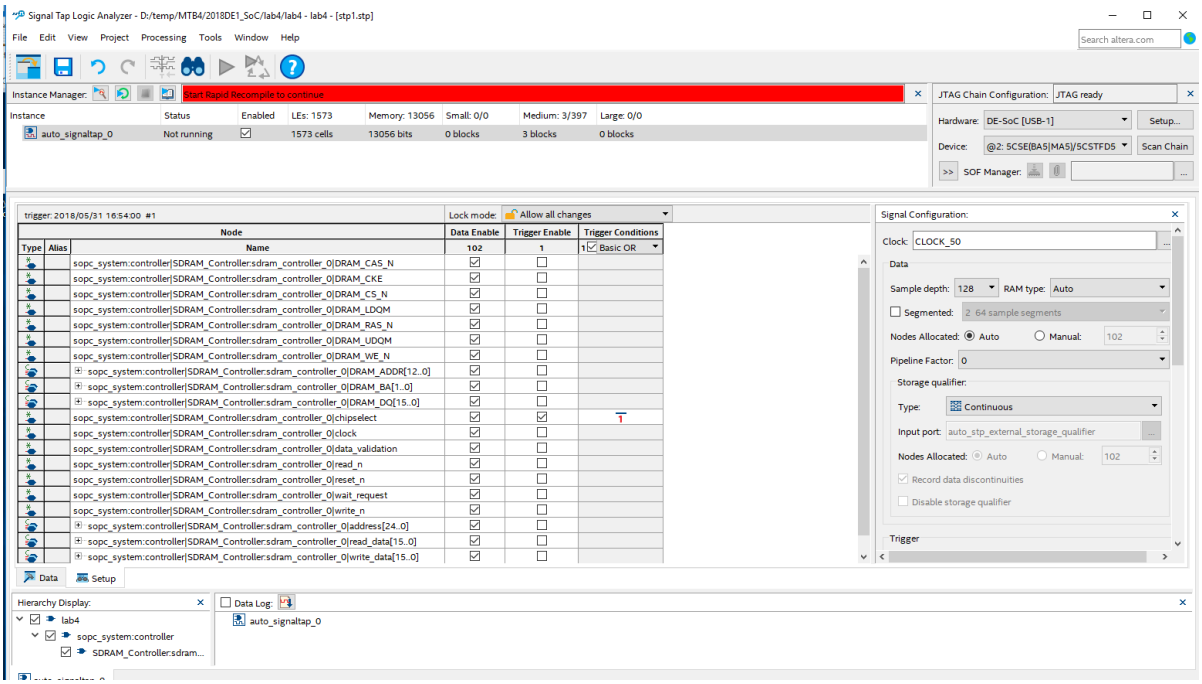

Figure 7: A possible setup for Signal Tap Logic Analyzer

Part 6: Developing Software

1. Open the Nios II Software Build Tools (SBT) for Eclipse. You can find it in Quartus: `Tools | Nios II Software Build Tools for Eclipse`

2. Follow the steps in "`Tutorial Material for Nios II SBT for Eclipse.pdf`" to create a project named "`memory_test`".

3. Choose "`hello_world.c`" as the template.

4. Now run the project by right clicking `memory_test`, then select `Run As | Nios II Hardware`. If everything is correct, "Hello from Nios II" is printed out in the console window.

5. If there are errors, please make sure:

   - The USB cable is attached to the DE1-SoC board and to the computer.
   - You compiled the Quartus project without errors.
   - Your SOPC system is correct.
   - The Quartus project with SOPC system has been downloaded to the DE1-SoC board successfully.

6. Paste the program you developed for your preparation into "`hello_world.c`". Re-compile it and verify that the SDRAM can be read and written correctly.

   Because of the large size of the SDRAM, testing all of the SDRAM may take a while. You may start with testing part of the memory when you are debugging your program.

7. Using Signal Tap Logic Analyzer, observe the state of the signals in the SDRAM Controller for different types of operations (i.e. reading and writing of chars, shorts and ints). Actually, Signal Tap Logic Analyzer can be used to help you debug your system if your system does not work or does not work properly.

8. If your system does not work, debug it using the Signal Tap Logic Analyzer. You may also want to use the RTL Viewer (`Tools | Netlist Viewers | RTL Viewer`) to verify that all the connections have been made as intended. Please note that large schematics may span several pages in the RTL Viewer. If that is the case, you can change pages in the top right corner of the RTL Viewer window.

Part 7: Demonstration

Demonstrate your working system to one of the TAs. Make sure to show them the output from your memory test software as it runs, the C code, and at least one sequence you obtained on the Signal Tap Logic Analyzer.