Mechtron 3TB4: Lab4

SDRAM Controller Interface

Reports Due:
Pre-lab Part : At the start of the lab session
Lab Report Part : At the start of Lab 5 (tutorial week)

Goals:

- To learn how to generate an SOPC system and how to add custom components to it using Platform Designer
- To learn how to use an SDRAM chip
- To learn how to use Signal Tap Logic Analyzer

Note: This lab consists of three (3) activities. Marks allocated to each are indicated in square brackets. **The following documents may help you with your lab. Please go over them at your convenience, in addition to the class notes:**

- DE1-SoC_Computer_Nios.pdf
- Introduction_to_the_Qsys_Tool.pdf
- Making_Qsys_Components.pdf
- SignalTap.pdf
- Tutorial_material_for_NIOSII_SBT_for_Eclipse.pdf
- Nios2_introduction.pdf
- Using_the_SDRAM.pdf
- MNL Avalon Specification
- Avalon Switch Fabric

The above documents can be found in the `ref` subdirectory once you download and umcompress the `lab4.zip` file from the course website.

Lab Equipment and Software

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. We will use an SDRAM controller module, `DE1_SoC_QSYS_sdram.v`, from Altera, that 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 module `SDRAM_Controller.v` by yourself. You should spend some time to read the code of `DE1_SoC_QSYS_sdram.v` to understand the SDRAM controller.

You will use your `SDRAM_Controller.v` to create a custom Platform Designer component, and then add this component to a Nios II based SOPC system. You will compile the resulting system to obtain the corresponding hardware configuration for the FPGA. You will write a piece of C code to test the functionality of the controller. Finally, you will observe the operation of the Avalon interface to the SDRAM controller using Signal Tap Logic Analyzer.

SDRAM Controller and the SDRAM Used in Lab

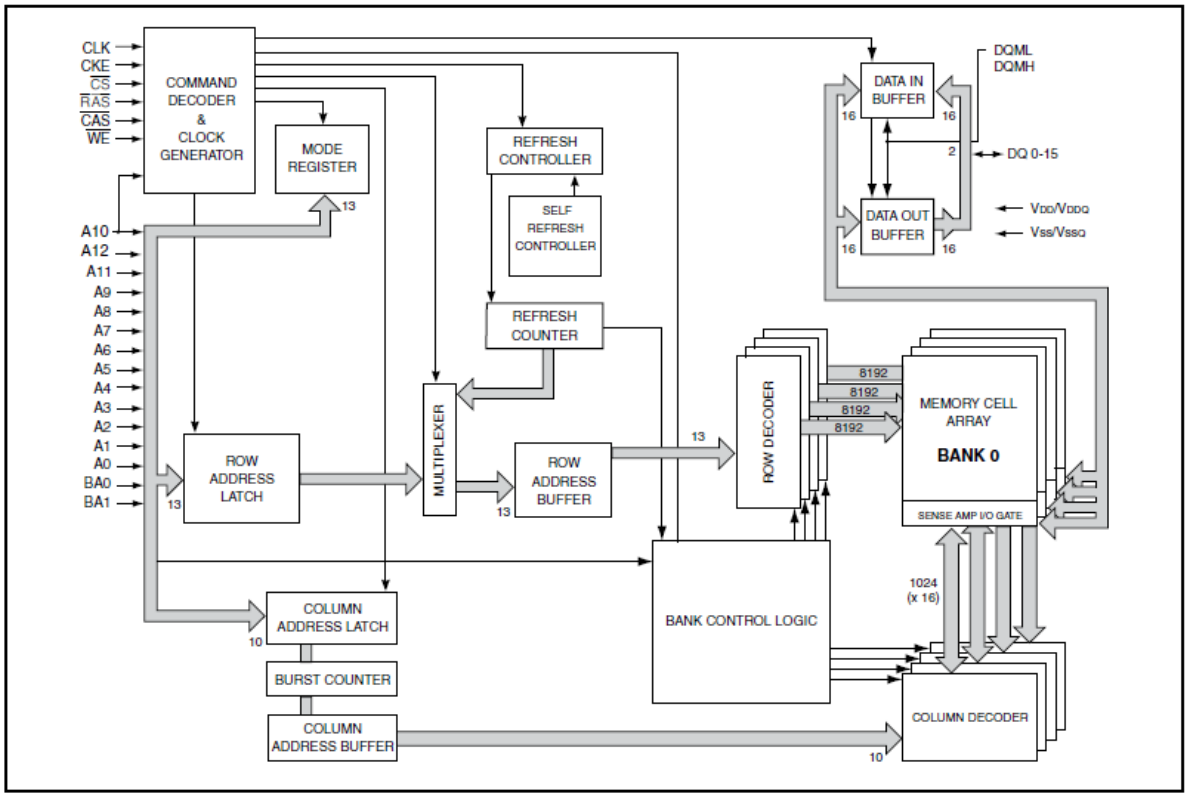

Figure 1: SDRAM Function Block Diagram (from datasheet for IS42S16320D)

The SDRAM controller is an Avalon slave device that provides an interface between the Avalon interconnect and the SDRAM chip. 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 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 1.

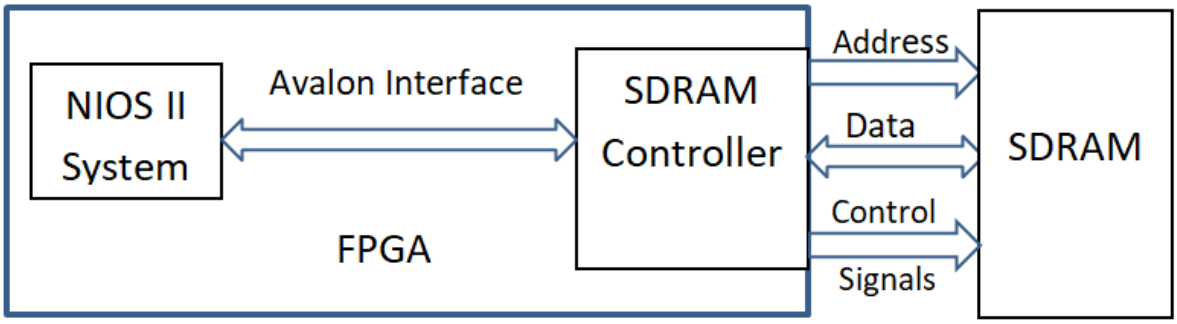

Figure 2: System Block Diagram

System Description

The system you are building is shown in Figure 2. The system components are described in the following sections.

The Signals Between Blocks

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

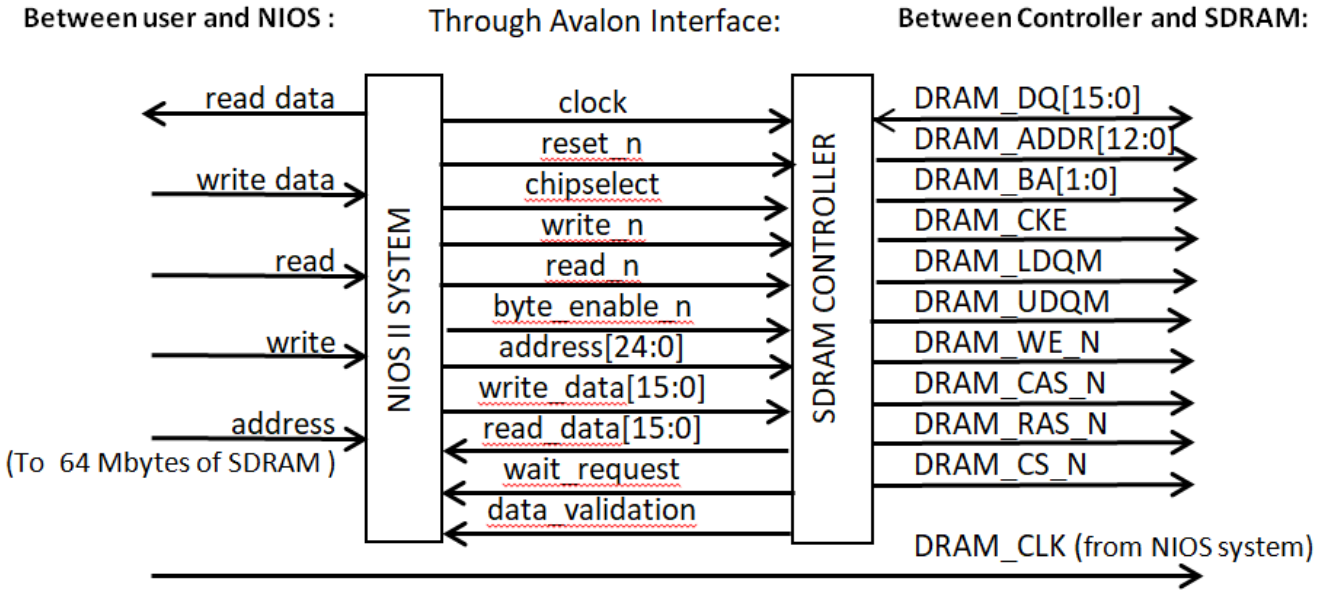

Figure 3: The Signals Between Blocks

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` signal, the entire 64M bytes of SDRAM can be accessed. For the SDRAM, when working with the signal `DRAM_RAS_N`, 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 word 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. The `DRAM_LDQM` and `DRAM_UDQM` assist in making byte reading and

writing possible.

The other signals in the system are: (NOTE: the "_N" or "_n" stands for active low for signals)

1. ***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.

   DRAM_CLK: The master clock input for the SDRAM. Please note that 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.

2. ***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 cannot 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 for signals on the Avalon interface, please refer to the Avalon interface document.

Activities:

Pre-lab [30 marks]

Submit answers to questions at the end of the tutorial document for this lab. [10]

The following activities must be completed and results submitted before the start of the lab. [20]

1. Considering that the SDRAM chip you will be using has the capacity of 64M bytes, and assuming that the lowest address of the SRDRAM chip is 0x00000000, what will be the highest byte address?

2. Considering that the 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? (Assume that the base address is 0x00000000 for the purposes of this question. The SOPC will assign a different base address as part of the design process as discussed later in the lab.)

3. When you write a C program to access memory, the compiler generates byte addresses (i.e. address 0x0000 will be the first byte and address 0x0001 will be the second byte). However, since the SDRAM that is used in the lab accepts a word address, the Avalon interconnect will automatically convert the byte address to a word address. What address will appear on the SDRAM address lines when the second byte is accessed?

4. Restore the archived Quartus project, `lab4.qar`, which can be found in the `data` folder in the `lab4.zip` file. The top-level module for the whole system is `lab4.v`. You need to complete module `SDRAM_Controller.v` and `lab4.v`.

5. Sketch a schematic of the circuit that is needed to implement `SDRAM_Controller.v`.

6. Complete module "`SDRAM_Controller.v`". Ensure that Quartus compiles this file without errors. To test this, temporarily make `SDRAM_Controller.v` the top-level module in your hierarchy (right click the Verilog file in Quartus' Project Navigator then select `Set as Top-Level Entity`). Once this is done, don't forget to make `lab4.v` the top-level module again.

7. Write a C program for the Nios II system to test the behaviour of your memory controller. 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 the first five 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 the first five 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 the first five 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 to do this is to use a simple loop and write the loop iterator into 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, adjust the iterator to make sure the whole address range for the memory is covered. Don't 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. (See: `memtext_stem.c` in the `data` folder in file `lab4.zip`)

In the Lab [55 marks]

This part includes four sections: SOPC Sytem Creation [20 marks], Quartus project completion [10 marks], Signal Tap [10 marks], and Software development [15 marks].

WARNING: To avoid compilation problems, make sure to create your project in a folder that does not contain any spaces in its name! For example, do NOT create your project on the Desktop or in the "My Documents" folder, because both are mapped under the folder "Documents and Settings" whose name contains spaces. Instead, create a folder without spaces in its name on a drive other than `C:` (i.e. `Z:`), and use that folder for development.

Create SOPC System[20]

1. Open the project you worked on for preparation.
2. Complete Pre-lab step 6 if you have not already done so (complete `SDRAM_Controller.v`).
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". In the "Group" field, input "MT3TB4_G-n", where "n" is your lab group number (Figure 4). Click on "Next".

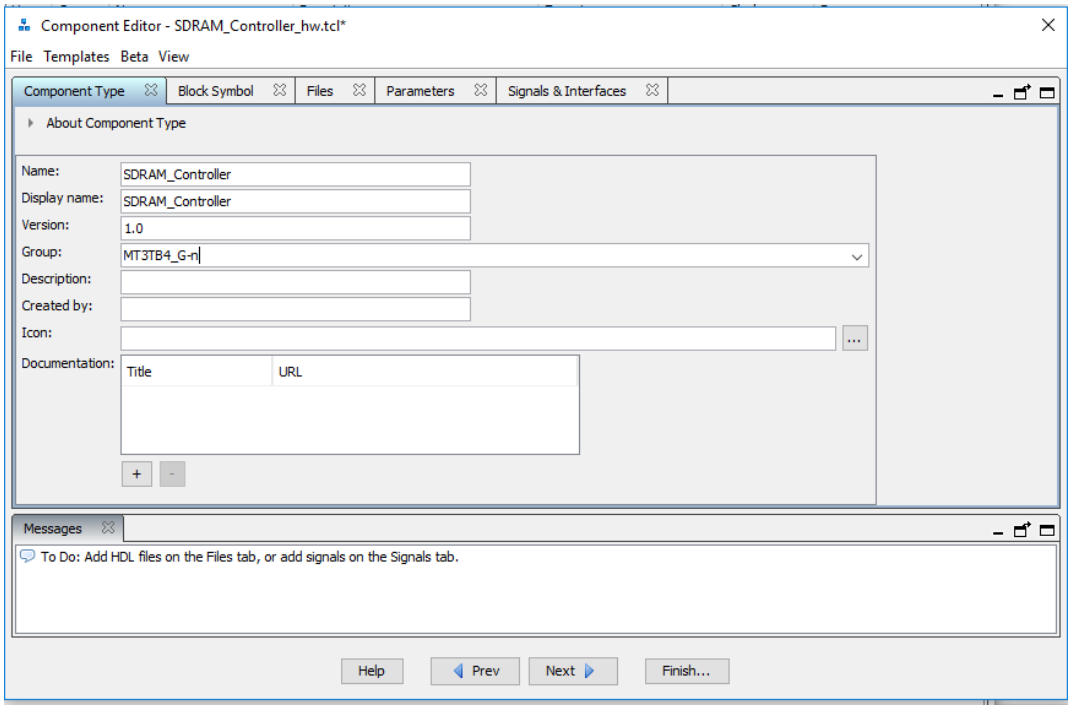

Figure 4: Specifiying Component Type

- 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 describes your component (Figure 5). Click on the "Add File..." button under the section "Synthesis Files" 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 6).

  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

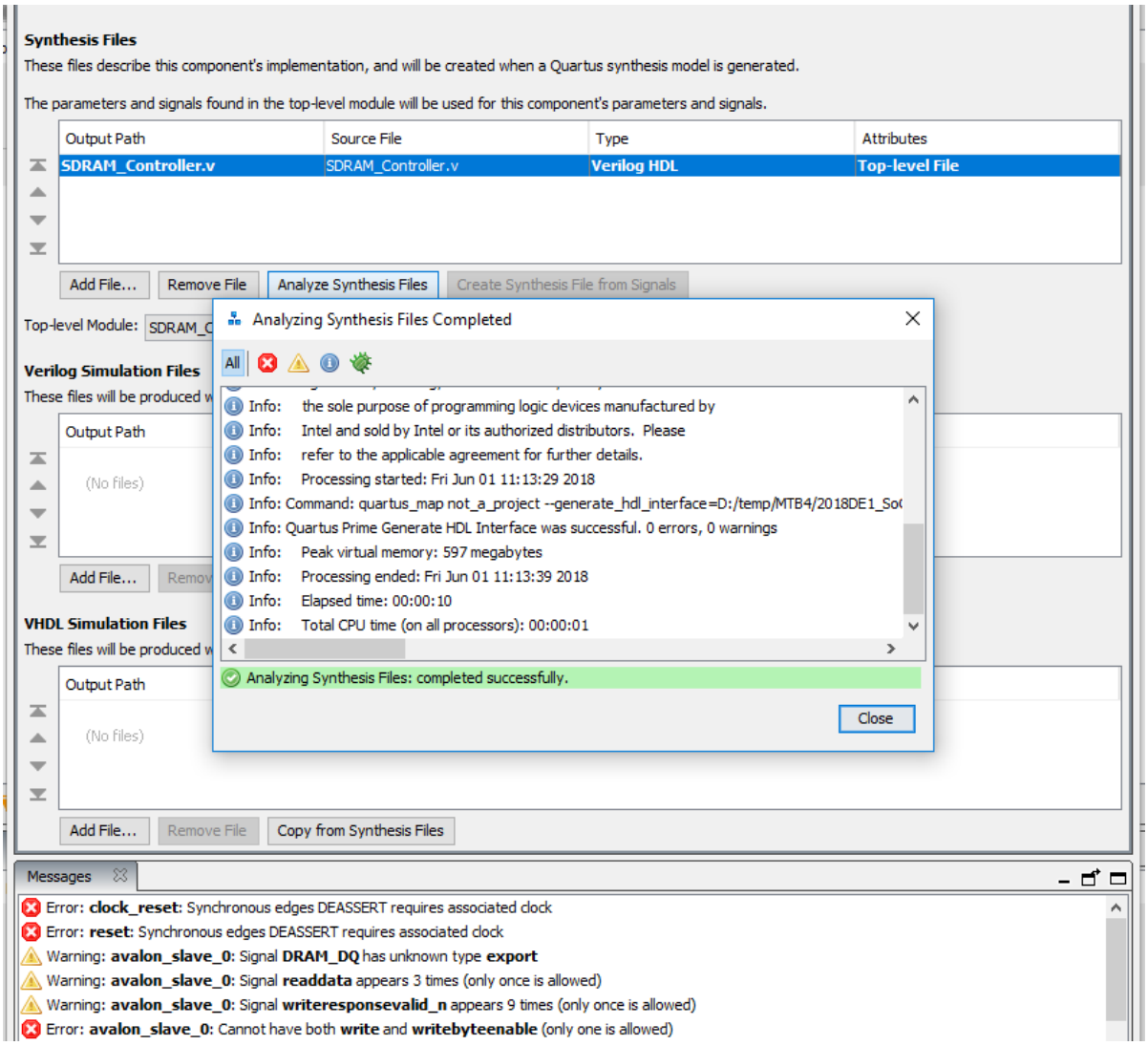

Figure 5: Specifying Files

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"**

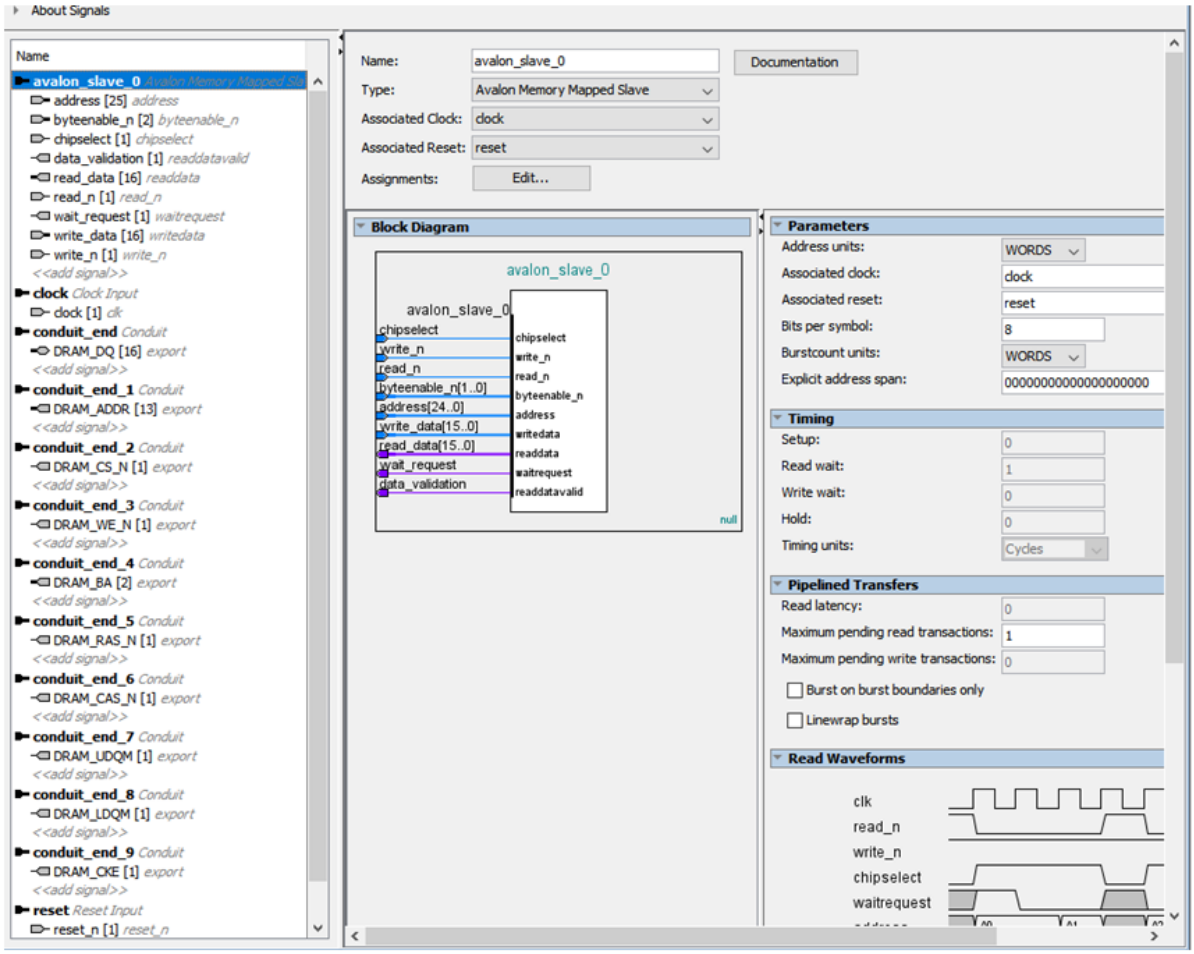

Figure 6: Creating Conduit Interfaces

are correct. Incorrect direction settings for signals can stop the SDRAM controller from working properly.

(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 under "MT3TB4_G-n".

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.

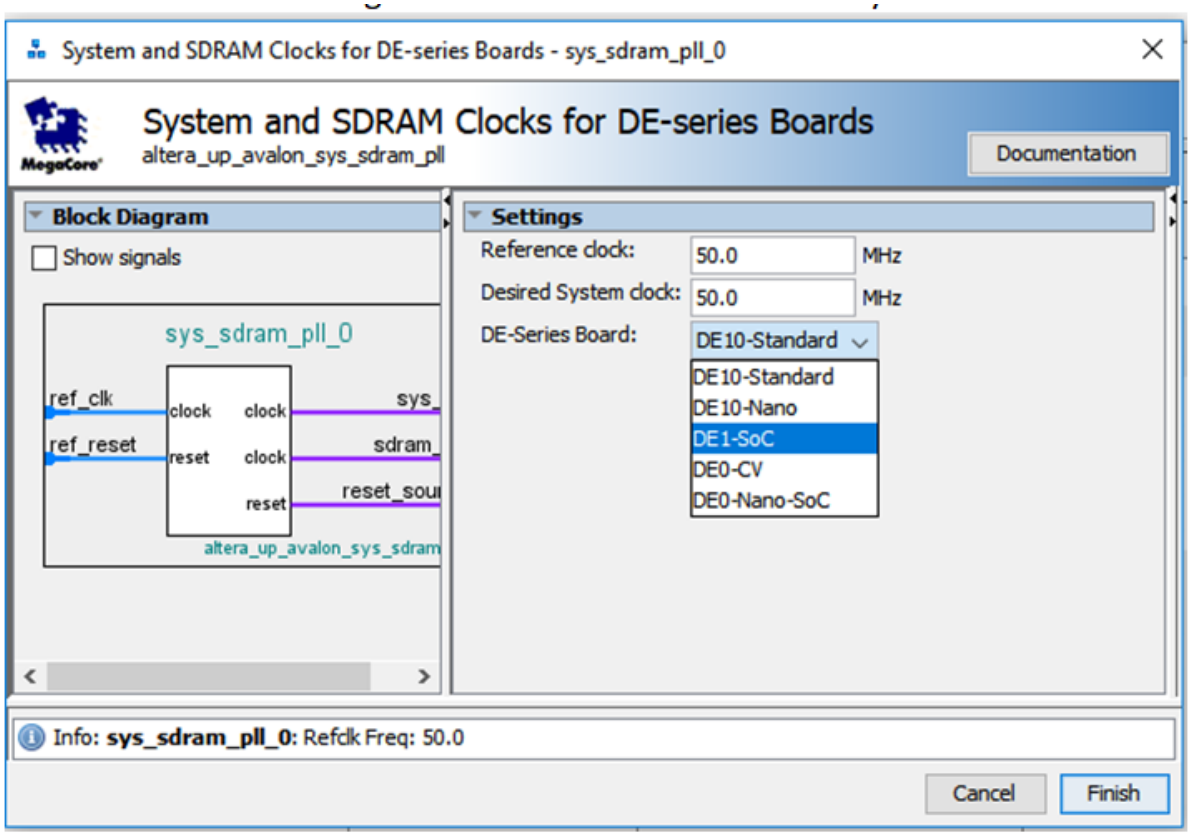

Figure 7: System and SDRAM Clocks

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 7).

10. In the Connection column, make the necessary connections by clicking on the appropriate empty circles to change them to solid circles (Figure 8).

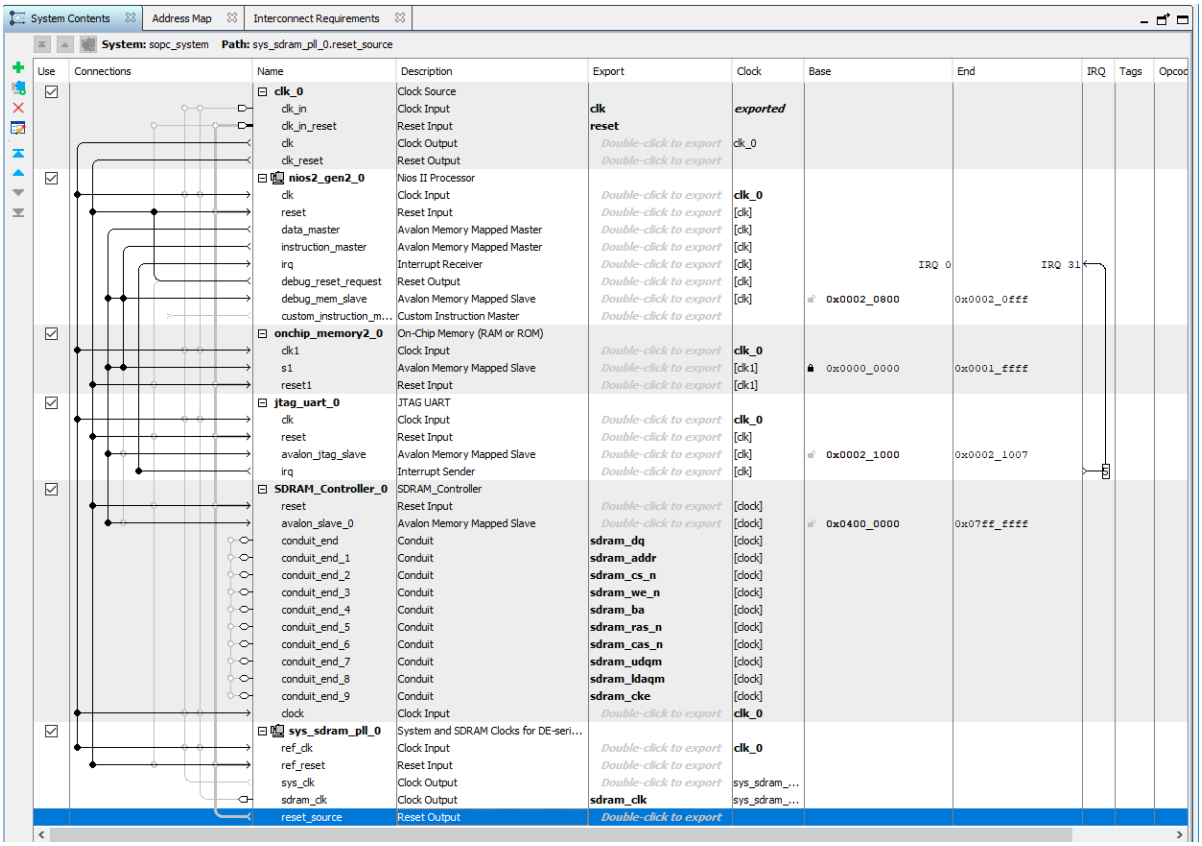

Figure 8: Component Connections, IRQ and Export

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 8).

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.

Complete Quartus Prime Project [10]

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:/lab4/sopc_system/synthesis`), find the `.qip` file (for example: `sopc_system.qip`). Add it to your project. You can also add the `.qsys` file in the project folder to your project for your convenience to edit your NIOS system.

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 `lab4.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 check box for "Show signals" to view the signals-ports correlation.)

4. Instantiate the NIOS system module in `lab4.v`, and complete `lab4.v`.

5. Set up `lab4.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.

Set up the Signal Tap Logic Analyzer [10]

The Signal Tap Logic Analyzer was introduced in the tutorial for this lab. Refer to the tutorial or the document named "`SignalTap.pdf`" referenced at the start of this document. Use this tool to monitor input/output signals to your controller.

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 a high value of the chipselect signal. (A possible setup is shown in Figure 9.)

4. Save the Signal Tap configuration and enable Signal Tap for the project. Make sure that lab4 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.

Develop Software [15]

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 and modify the printf() statement to print a welcome message from your group in the Nios console as done during the tutorial session for this lab.

4. Now run the project by right clicking memory_test, then select Run As | Nios II Hardware. If everything is correct, "Hello from MT3TB4 Group (your group number)" is printed out in the console window.

5. If there are errors, please make sure:

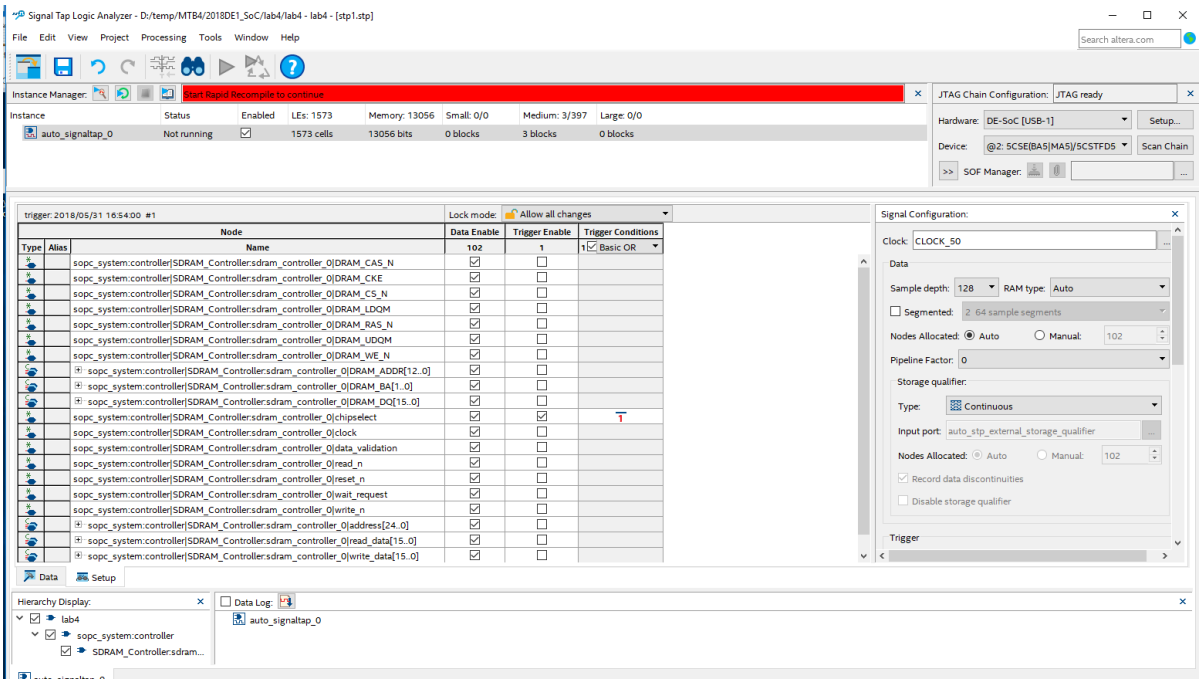

Figure 9: A possible setup for Signal Tap Logic Analyzer

- 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. 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.

Lab Report (Part 2)[15 marks]

Describe what you did in this lab, include the code used and answer the following questions:

1. Using screen shots taken in the "Develop Software" part of the lab explain the following:

(a) Why does it appear that writing takes less clock cycles than reading?

(b) While reading or writing a char value, e.g. 5, why does the SDRAM_DQ show values such as 0x0505 instead of 0x0005?

(c) While reading or writing two consecutive char values, e.g. 5 and 6 using memory locations 0x0000 and 0x0001, why does the SDRAM_ADDR show the same 0x0000 value for both locations?

2. What steps does the Avalon interconnect take to write a 32-bit integer into the 16-bit SDRAM memory?

3. Open the compilation report in Quartus, and report the following numbers:

   (a) Total number of logic elements used by your circuit

   (b) Total number of memory bits used by your circuit

   (c) Total number of pins

   (d) The maximum number of logic elements that can fit in the FPGA you used

4. Considering just the maximum number of logic elements on the FPGA, approximately how many SOPC systems like the one you built could fit on the FPGA?

5. Considering just the maximum amount of memory available in the FPGA, approximately how many SOPC systems like the one you built could fit on the FPGA?

6. Considering just the maximum number of pins on the FPGA, how many SOPC systems like the one you built could fit on the FPGA?

Bonus [20 marks]

If you wish to complete the bonus portion of this lab, you are to design another peripheral that you will add to the system, in addition to the SDRAM controller described above. This peripheral should consist of four 16-bit registers, plus the circuitry needed to display the contents of these registers on the seven-segment displays on the DE1-SoC board. This is similar to what you have done in lab 2, except that this time the registers are of different width and the interface to them is through Avalon, so that the Nios processor can write to them. The Nios processor does NOT have to be able to read these registers. Therefore, you may be able to reuse some parts of your code from lab 2. Registers in this exercise are 16 bits wide. You should ensure that Nios can write to individual bytes of this register. You should take advantage of the byteenable signals for this purpose.

You should use the seven-segment displays HEX0 to HEX3 to display the contents of these registers. Your final system should look as outlined in Figure 10. You are responsible for fully specifying the functionality of the module with registers and integrating it into the system. You should use the process we used to integrate the SDRAM Controller module as a guideline. **You may need to use more on-chip memory**.

The Tutorial named "`Making_Qsys_Components.pdf`" referenced at the start of this document may be of great help for creating the new Platform Designer component.

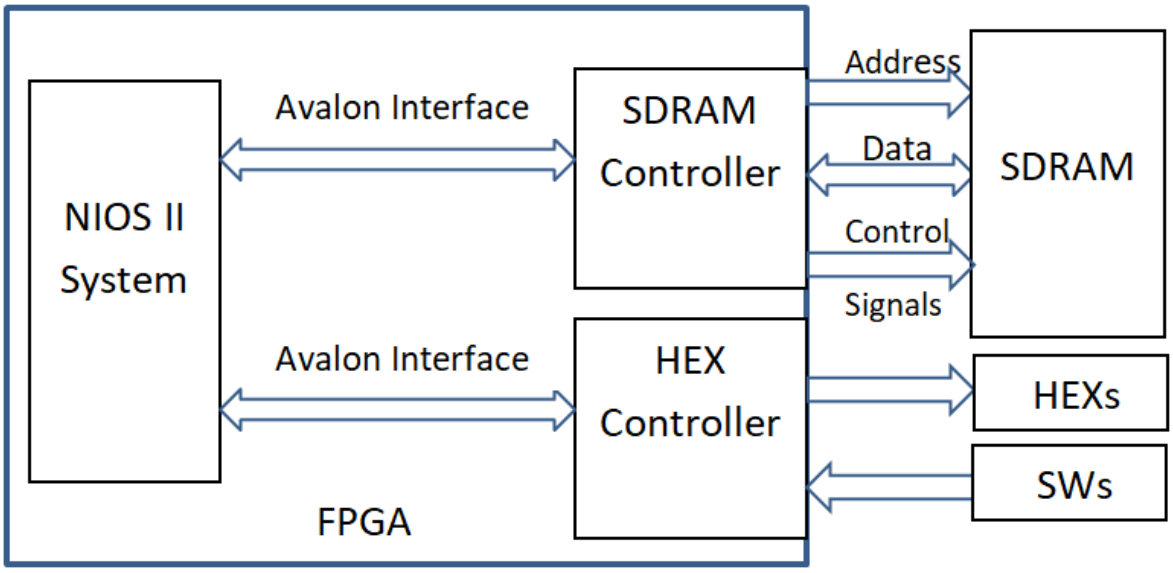

Figure 10: System Diagram Including Bonus Part