Laboratory Exercise 7

DMA Data Transfer

The purpose of this exercise is to investigate the concept of DMA data transfer. *Direct Memory Access (DMA)* is a mechanism that allows two peripheral devices to transfer data directly between each other, rather than use the processor to carry out the data transfer. Therefore, it can speed up the rate of transfer and reduce the processor's load. From the implementation point of view, a DMA transfer essentially copies a block of data from one device to another. This process is usually controlled by a circuit called the *DMA Controller*. In this exercise we will implement DMA transfer between two memories on Altera's DE1 board. One memory is the SRAM chip, while the other memory is implemented on the FPGA chip in the form of an LPM module.

This exercise is more challenging than the Laboratory Exercises 1 through 6. It requires a good understanding of the concepts illustrated in the previous exercises. It is suitable for a 2-week laboratory session, where Parts I through III can be done during the first week and Part IV can be done during the second week. The exercise is based on the system shown in Figure 1.

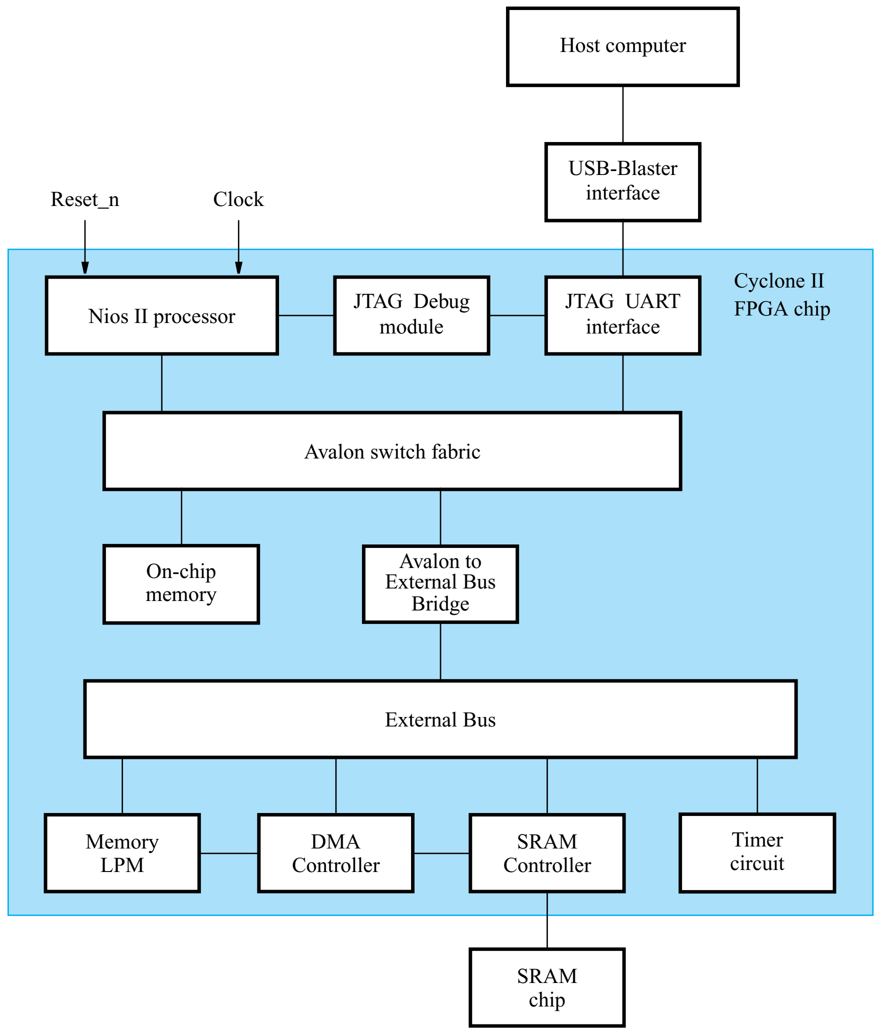


Figure 1: Block diagram of the desired DMA system.

The Nios II processor can access both the SRAM and LPM memories via the *Avalon to External Bus Bridge*. It can also access the DMA Controller to set up a DMA transfer.

A typical DMA transaction proceeds as follows:

1. The processor prepares the DMA Controller for a transfer by writing to its registers information such as an address of the data and the number of bytes to transfer.
2. The processor informs the DMA Controller to start the data transfer. Then the DMA Controller proceeds to transfer the data without further intervention from the processor. It controls the transfer.
3. When the transfer is completed, the DMA Controller may generate an interrupt request to the processor, if it is configured to do so.

During or after the transfer, the processor can observe the DMA Controller's state by examing its Status Register.

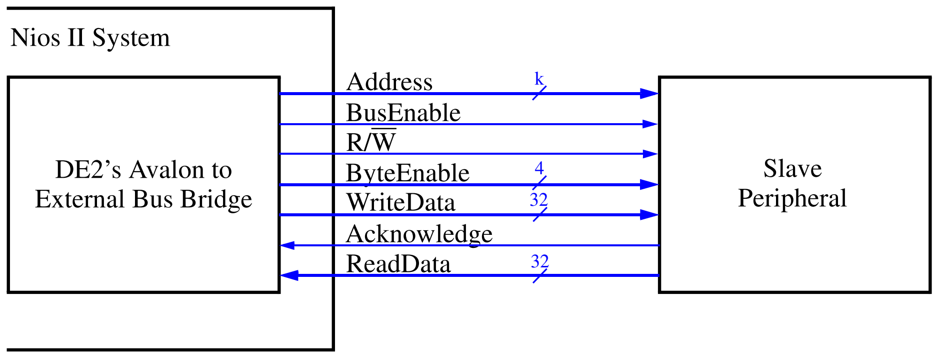
Altera's SOPC Builder can use the *Avalon to External Bus Bridge* component to implement a bus-like structure to which the user may connect a variety of peripheral devices. The bridge allows the designer to connect a peripheral device to a Nios II system in the Quartus II software. It provides a bus interface to which one or more "slave" peripherals can be connected. Figure 1 shows the bus signals and timing information for the external bus. The required signals are:

- *Address* – k bits (up to 32). The address of the data to be transferred. 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.

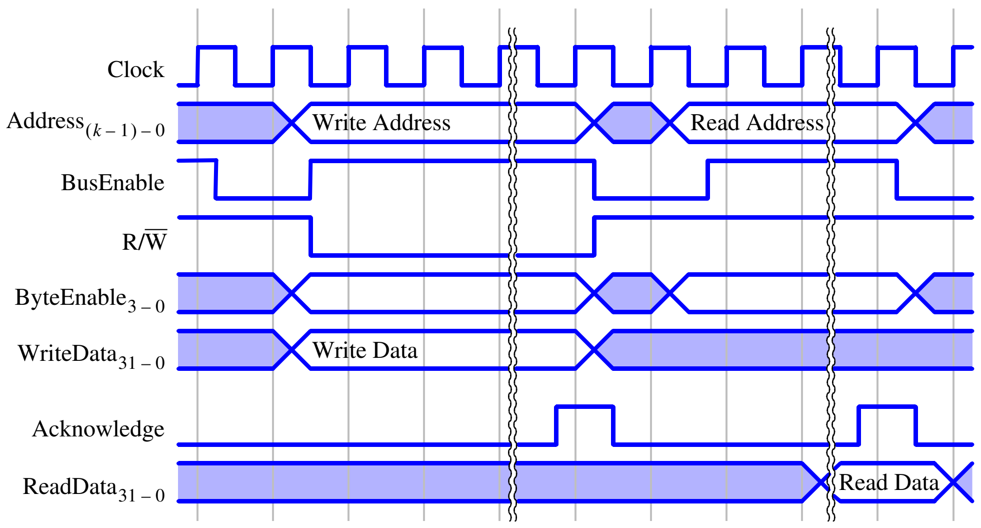
The bus is synchronous – all bus signals to the peripheral device must be read on the rising edge of the clock. To initiate a transfer the *Address*, *RW*, *ByteEnable* and possibly *WriteData* signals are set to the appropriate values. Then, the *BusEnable* signal is set to 1.

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.



(a) External Bus Signals



(b) External Bus Timing Diagram

Figure 2. The Avalon to External Bus Bridge signals.

Part I

In this part, we will create the LPM memory by instantiating the *altsyncram* module from the Quartus II Library of Parameterized Modules. The LPM memory will be connected to the External Bus. Implement the required system as follows:

1. Create a new Quartus II project called *Lab7_Part1*. Select Cyclone II EP2C20F484C7 as the target chip,

which is the FPGA chip on the Altera DE1 board.

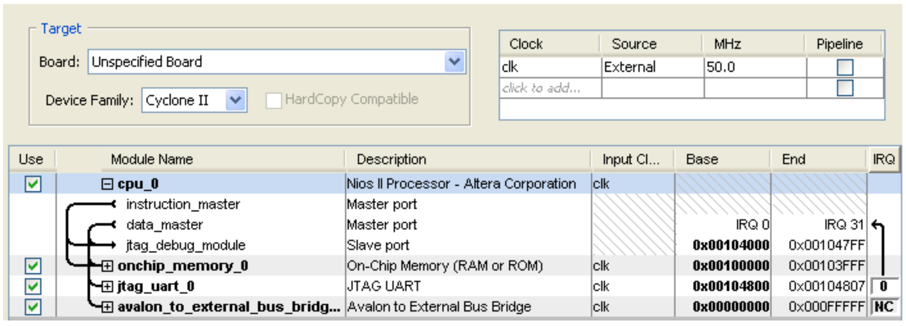
2. Use the SOPC Builder to create a system named *nios_system*, which includes the following components:

   - Nios II/s processor with JTAG Debug Module Level 1
   - On-chip memory - select the RAM mode and the size of 16 Kbytes
   - JTAG UART - use the default settings
   - Avalon to External Bus Bridge - choose the 16-bit data width and the address range of 1024 Kbytes. In the SOPC Builder window, the desired bridge is found by selecting Avalon Components > University Program DE1 Board > Avalon to External Bus Bridge.

     Note: The choice of the address range of 1024 Kbytes implies that $k = 20$ address lines will be implemented in the bus.

3. Connect the Avalon to External Bus Bridge to Nios II's data_master port and not to the instruction_master port. You can remove the connection between the bridge and the instruction master port by clicking on the connecting path in the SOPC Builder window.

4. From the System menu, select Auto-Assign Base Addresses. You should now have the system shown in Figure 3.



Target

Board: Unspecified Board

Device Family: Cyclone II

HardCopy Compatible

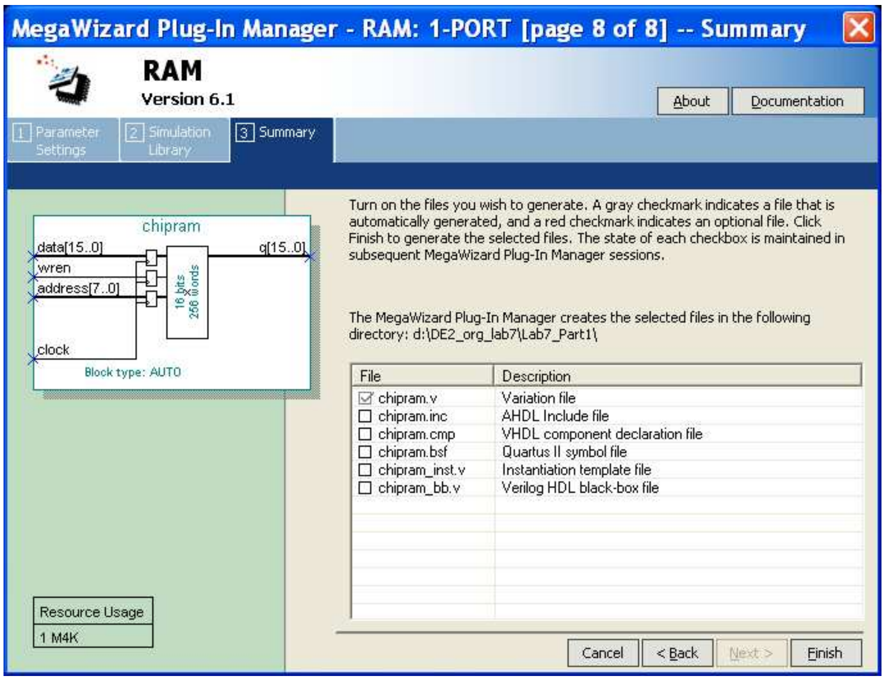
Clock	Source	MHz	Pipeline
clk	External	50.0	☐
click to add...			☐

Use	Module Name	Description	Input Cl...	Base	End	IRQ
✓	cpu_0	Nios II Processor - Altera Corporation	clk			
	instruction_master	Master port				
	data_master	Master port		IRQ 0	IRQ 31	
	jtag_debug_module	Slave port		0x00104000	0x001047FF	
✓	onchip_memory_0	On-Chip Memory (RAM or ROM)	clk	0x00100000	0x00103FFF	
✓	jtag_uart_0	JTAG UART	clk	0x00104800	0x00104807	0
✓	avalon_to_external_bus_bridg...	Avalon to External Bus Bridge	clk	0x00000000	0x000FFFFF	NC

Figure 3. The Nios II system for Part I specified in the SOPC Builder.

5. Generate the system, exit the SOPC Builder and return to the Quartus II software.

6. Configure a 4-Kbit memory by using the *altsyncram* LPM module. Let this memory be called *chipram*. It has to comprise 256 words, with each word consisting of 16 bits. Choose the LPM memory module of RAM type with a single port. Let its outputs be unregistered. Figure 4 shows the desired memory configured by using the MegaWizard Plug-In Manager. (The tutorial *Using Library Modules in Verilog/VHDL Designs* explains the use of LPMs.)



Turn on the files you wish to generate. A gray checkmark indicates a file that is automatically generated, and a red checkmark indicates an optional file. Click Finish to generate the selected files. The state of each checkbox is maintained in subsequent MegaWizard Plug-In Manager sessions.

The MegaWizard Plug-In Manager creates the selected files in the following directory: d:\DE2_org_lab7\Lab7_Part1\

File	Description
✓ chipram.v	Variation file
☐ chipram.inc	AHDL Include file
☐ chipram.cmp	VHDL component declaration file
☐ chipram.bsf	Quartus II symbol file
☐ chipram_inst.v	Instantiation template file
☐ chipram_bb.v	Verilog HDL black-box file

Resource Usage
1 M4K

Figure 4. The *chipram* module configuration.

7. Write a top-level Verilog/VHDL module that instantiates the *nios_system* and *chipram* modules. Add this file to your Quartus II project.
8. Import the pin assignments using the file *DE1_pin_assignments.csv*.
9. Compile your Quartus II project.
10. Program and configure the Cyclone II FPGA on the DE1 Board to implement the generated system.
11. Create a Nios II assembly-language program to write a few 16-bit numbers into the *chipram* memory, and then read these numbers from the memory.
12. Use the Altera Debug Client to compile, download and run your program. Verify that the *chipram* module works correctly.

Part II

In this part, we will add the SRAM memory to our system. This can done in the same manner as in Laboratory Exercises 5 and 6, by implementing an SRAM Controller that can connect the SRAM chip on the DE1 board to the bus provided by the Avalon to External Bus Bridge.

The SRAM chip uses the following signals:

- *SRAM_ADDR*$_{17-0}$ – 18-bits, input. The addresses of the SRAM's 16-bit data words.
- *SRAM_CE_N* – 1-bit, input. Indicates that all other signals are valid. (Chip Enable)
- *SRAM_WE_N* – 1-bit, input. Indicates that the bus transfer is a write operation. (Write Enable)

- *SRAM_OE_N* – 1-bit, input. Indicates that the bus transfer is a read operation. (Output/Read Enable)
- *SRAM_UB_N* – 1-bit, input. Indicates that the upper byte should be read or written. (Upper Byte Enable)
- *SRAM_LB_N* – 1-bit, input. Indicates that the lower byte should be read or written. (Lower Byte Enable)
- *SRAM_DQ*$_{15-0}$ – 16-bits, bidirectional. These lines carry the data being transferred. They are driven by the SRAM chip during read operations and by your controller during write operations.

Figure 5 demonstrates the timing for the SRAM signals. Notice that the SRAM chip completes data transfers within one cycle. Also, notice that all control signals are active low. The signal SRAM_Write_Data is an internal signal equivalent to SRAM_DQ, but it is used in write transfers only. This signal must be set to high impedance, except during a write transfer, when it is set to the data to be written.

The SRAM chip can read and write one 16-bit value within one 50-MHz clock cycle, so all signals to the SRAM chip need only be asserted for that one clock cycle for each 16-bit transfer.

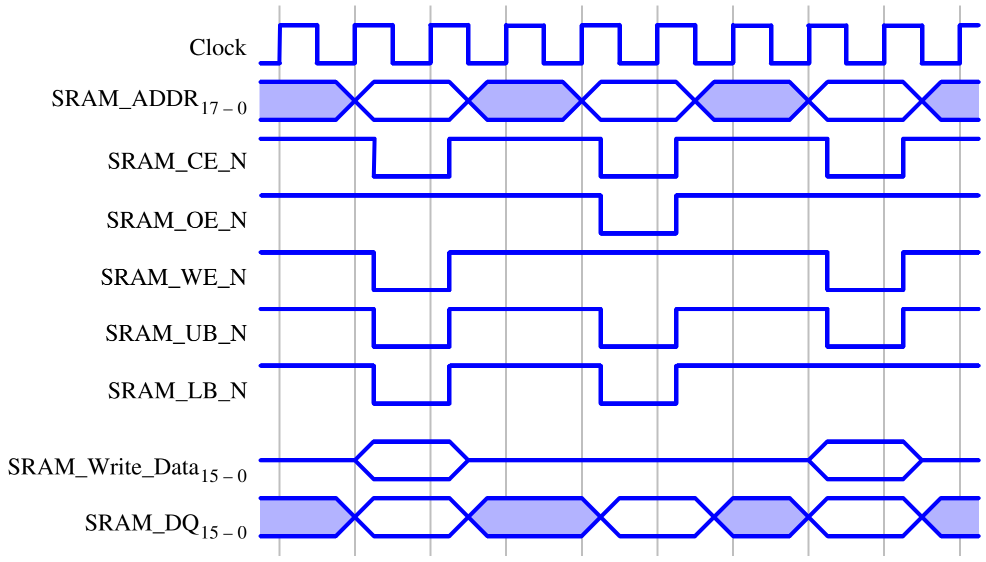


Figure 5. The SRAM signals timing diagram

You have to design the SRAM Controller. Note that both the SRAM chip and the Avalon to External Bus Bridge have 16 data lines.

Perform the following steps:

1. Create a new project called *Lab7_Part2*.
2. Use the SOPC Builder to generate the same *nios_system* as in Part I.

3. Complete the design of the SRAM Controller – start with the skeleton design in the Verilog/VHDL file *SRAM_Controller*.
4. Add the Verilog/VHDL files *Lab7_Part2*, *chipram* and *SRAM_Controller* to your Quartus II project, and specify the pin assignments by importing the file *DE1_pin_assignments.csv*.
5. Compile your Quartus II Project.
6. Program and configure the Cyclone II chip to implement the generated system.
7. Write a Nios II assembly-language program to write some data into a few locations in the SRAM chip and the *chipram* memory, and then read this data back into processor registers.
8. Use the Debug Client to compile, download and run your program. Verify that the data is written correctly into the memory locations.

Part III

Next, we will investigate the efficiency of using the processor to transfer data from the SRAM to the chipram memory. This should be done by first loading 100 numbers (e.g. from 0 to 99) into the SRAM. Then, the processor should transfer these numbers into the chipram memory by loading each number (from the SRAM) into a processor register, and then storing this number into chipram. We are interested in the time needed to transfer all 100 numbers, by measuring the number of clock cycles it takes to complete the transfer. To measure the transfer time, implement an appropriate counter that will serve as a *timer* circuit. The operation should be as follows:

1. The processor writes 100 numbers into SRAM.
2. The processor clears the timer to 0 and then issues a command to start counting.
3. The processor transfers all 100 numbers from SRAM to chipram.
4. As soon as the last number is stored in the chipram, the processor stops the timer.
5. The processor reads the contents of the timer and loads the resulting count into one of its registers.

Perform the following steps:

1. Create a new project called *Lab7_Part3*.
2. Generate the same system as in Part II.
3. Add a timer circuit to your system, by specifying it in your Verilog/VHDL file *Lab7_Part3*.
4. Compile your Quartus II Project.
5. Program and configure the Cyclone II chip to implement the generated system.
6. Write a Nios II assembly-language program to perform the data transfer as explained above.
7. Use the Debug Client to compile, download and run your program. Verify that the data is transferred correctly and record the time taken.

Part IV

Now, we will implement the transfer by using the DMA approach. A *DMA Controller* circuit is to be included in the system as indicated in Figure 1. This circuit has to include:

- An address register (counter) that holds the address of a location in the SRAM
- An address register (counter) that holds the address of a location in the chipram memory
- A data register that will be used in transfering a data item from SRAM to chipram
- A register (counter), called *dma_counter*, which indicates how many data items are still to be transfered

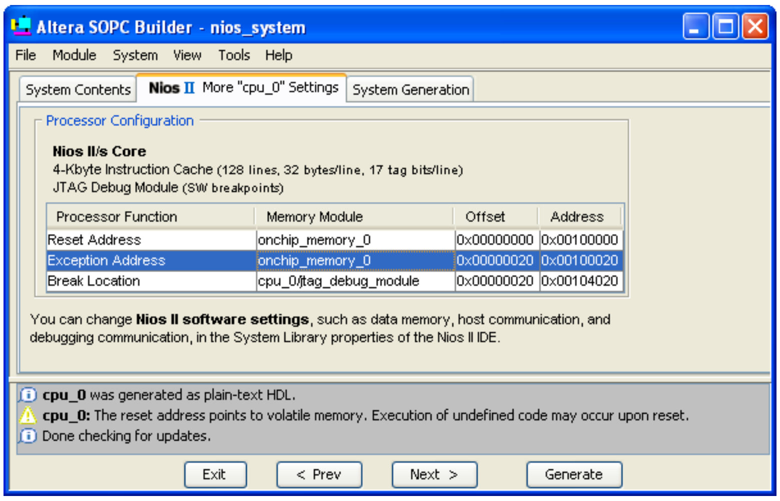
All registers (counters) should be accessible by the processor, so that it can both set and read their contents.

The required operation is:

1. Processor writes 100 numbers into SRAM.
2. Processor writes the starting addresses in SRAM and chipram into the respective registers (counters) in the DMA Controller.
3. Processor sets the initial value in the *dma_counter*.
4. Processor clears the timer to 0.
5. Processor issues a command to the DMA Controller to start the transfer. This also causes the timer to start counting the clock cycles.
6. DMA Controller copies all of the data from SRAM into chipram.
7. After transferring the last item, the DMA Controller stops the timer and raises an interrupt to signify the end of the DMA process.
8. Processor reads the count in the timer circuit.

Perform the following steps:

1. Create a new project called *Lab7_Part4*.
2. Generate the same system as in Part III. Note that if the SOPC Builder generated the addresses automatically, the on-chip memory will have addresses starting from 0x0100000 as indicated in Figure 3. Then, it is necessary to ensure that the Exception Handler (needed to use the interrupt mechanism) address is in the on-chip memory address range. This must be done when the system is generated by the SOPC Builder. Upon assigning the addresses (given in Figure 3), click on the tab Nios II More "cpu_0" Settings and set the Exception and Reset addresses to the correct range as shown in Figure 6.


Altera SOPC Builder - nios_system
File Module System View Tools Help
System Contents
Nios II More "cpu_0" Settings
System Generation
Exit
< Prev
Next >
Generate

Processor Configuration

Nios II/s Core
4-Kbyte Instruction Cache (128 lines, 32 bytes/line, 17 tag bits/line)
JTAG Debug Module (SW breakpoints)

Processor Function	Memory Module	Offset	Address
Reset Address	onchip_memory_0	0x00000000	0x00100000
Exception Address	onchip_memory_0	0x00000020	0x00100020
Break Location	cpu_0/jtag_debug_module	0x00000020	0x00104020

You can change **Nios II software settings**, such as data memory, host communication, and debugging communication, in the System Library properties of the Nios II IDE.

cpu_0 was generated as plain-text HDL.
cpu_0: The reset address points to volatile memory. Execution of undefined code may occur upon reset.
Done checking for updates.

Figure 6. Setting the Exception and Reset addresses.

3. Design and add the DMA Controller to your system.
4. Compile your Quartus II Project.
5. Program and configure the Cyclone II chip to implement the generated system.
6. Write a Nios II assembly-language program to perform the data transfer using the DMA approach, as explained above.
7. Use the Debug Client to compile, download and run your program. Verify that the data is transferred correctly and record the time taken.

Compare the times needed to perform the desired transfer using the DMA approach and the approach in Part III.