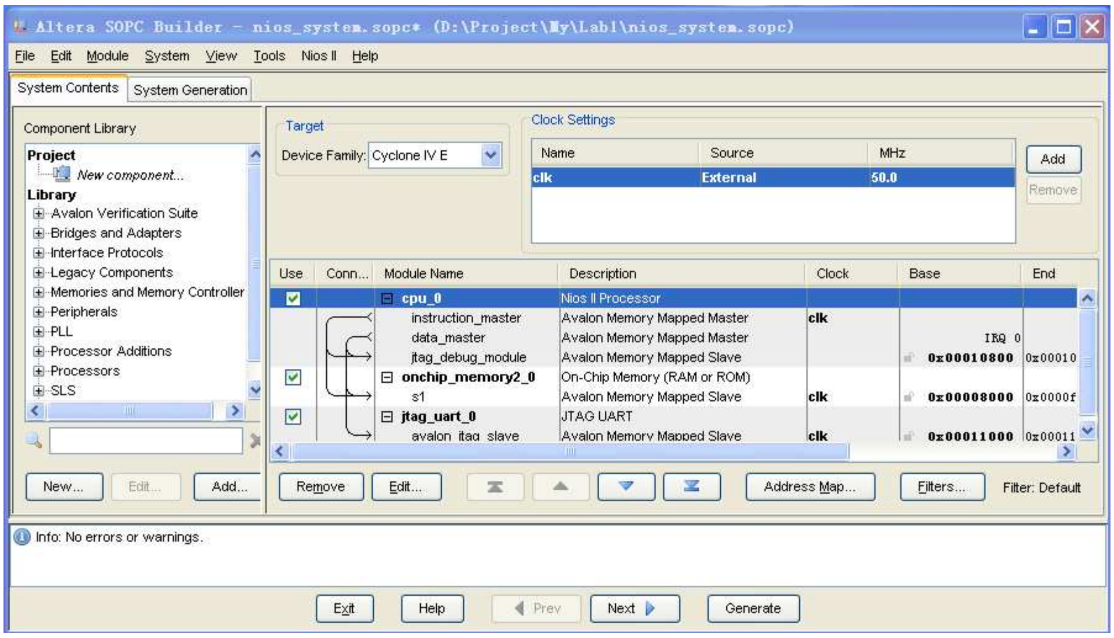
Laboratory Exercise 1
A Simple Computer System

The purpose of this exercise is to learn how to create and use a simple computer system. The system will consist of an Altera Nios II processor and an application program. We will use the Quartus II and SOPC Builder software to generate the hardware portion of the system. We will use the *Altera Monitor Program* software to compile, load and run the application program.

Part I

In this exercise, you will use the SOPC Builder to create the system in Figure 1, which consists of a Nios II/e processor and a memory block. The Nios II/e processor processes data. The memory block stores instructions and data.



Device Family: Cyclone IV E

Name	Source	MHz
clk	External	50.0

Use	Conn...	Module Name	Description	Clock	Base	End
✓		cpu_0	Nios II Processor			
		instruction_master	Avalon Memory Mapped Master	clk		
		data_master	Avalon Memory Mapped Master		IRQ 0	
		jtag_debug_module	Avalon Memory Mapped Slave		0x00010800	0x00010
✓		onchip_memory2_0	On-Chip Memory (RAM or ROM)			
		s1	Avalon Memory Mapped Slave	clk	0x00008000	0x0000f
✓		jtag_uart_0	JTAG UART			
		avalon_jtag_slave	Avalon Memory Mapped Slave	clk	0x00011000	0x00011

Info: No errors or warnings.

Figure 1. The Nios II system in SOPC Builder.

Implement the system in Figure 1 as follows:

1. Create a new Quartus II project. Select Cyclone IV EP4CE115F29C7 as the target chip, which is the FPGA chip on the Altera DE2_115 board.

2. Use the SOPC Builder to create a system named nios_system, which includes the following components:
   • Nios II/e processor with JTAG Debug Module Level 1
   • On-chip memory - RAM mode and 32 Kbytes in size with width of 32 bits

3. From the **System** menu, select **Auto-Assign Base Addresses**. You should now have the system shown in Figure 1.

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

5. Instantiate the generated Nios II system within a Verilog/VHDL module.

6. Assign the pin connections:

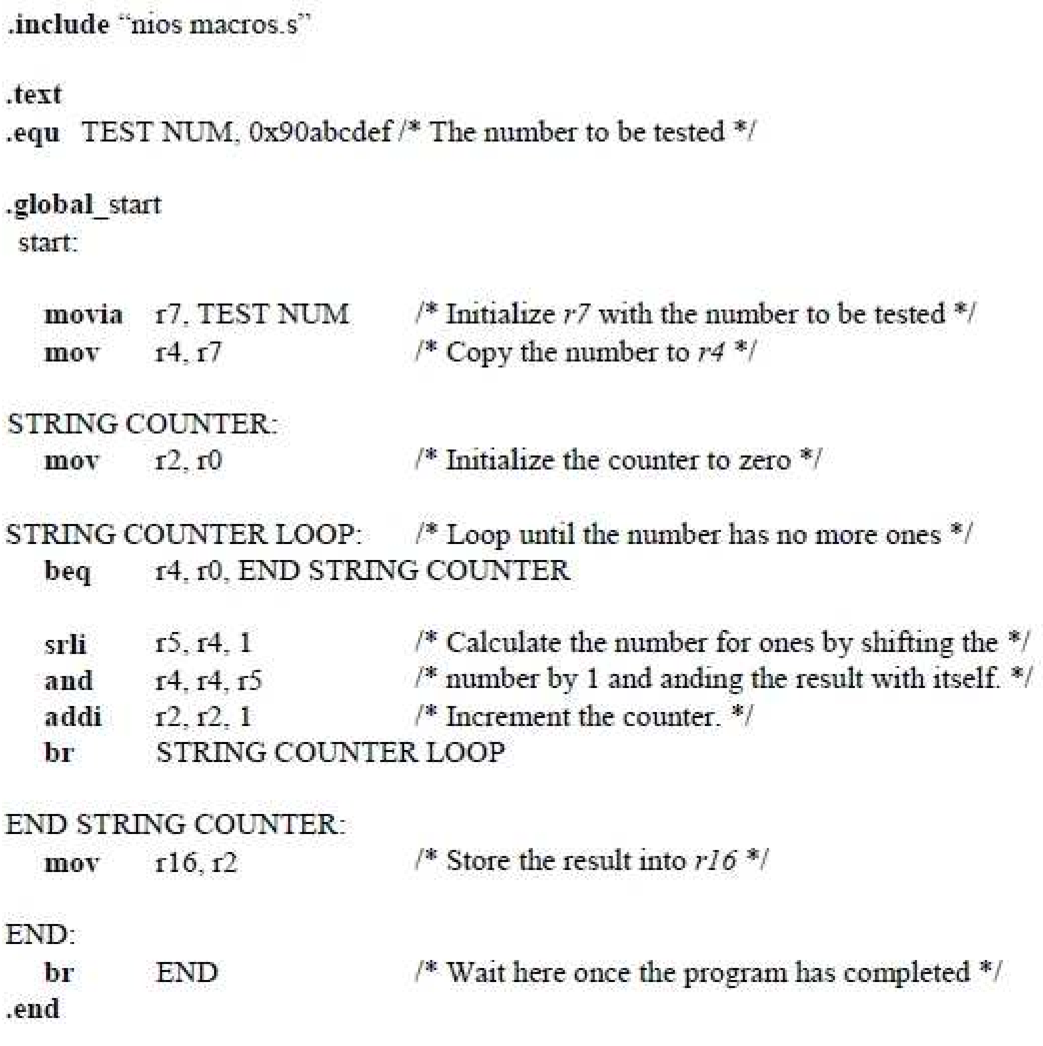
- clk **- PIN_Y2** (This is a 50 MHz clock)
- reset n **- PIN M23** (This is the pushbutton switch KEY0)

7. Compile the Quartus II project.

8. Program and configure the Cyclone IV FPGA on the DE2_115 board to implement the generated system. To use the system, we have to give the processor a program to execute, which we will do in Part II of this exercise.

Part II

In a digital computer all data is represented as strings of 1s and 0s. The Nios II assembly-language program in Figure 2 examines a word of data and determines the maximum number of consecutive 1s in that word. For example, the word 0x937a (1001001101111010) has a maximum of 4 consecutive 1s. The code in the figure calculates the number of consecutive 1s for the data 0x90abcedf.



```
.include "nios macros.s"

.text
.equ   TEST NUM, 0x90abcdef /* The number to be tested */

.global_start
 start:

    movia   r7, TEST NUM        /* Initialize r7 with the number to be tested */
    mov     r4, r7              /* Copy the number to r4 */

STRING COUNTER:
    mov     r2, r0              /* Initialize the counter to zero */

STRING COUNTER LOOP:            /* Loop until the number has no more ones */
    beq     r4, r0, END STRING COUNTER

    srli    r5, r4, 1           /* Calculate the number for ones by shifting the */
    and     r4, r4, r5          /* number by 1 and anding the result with itself. */
    addi    r2, r2, 1           /* Increment the counter. */
    br      STRING COUNTER LOOP

END STRING COUNTER:
    mov     r16, r2             /* Store the result into r16 */

END:
    br      END                 /* Wait here once the program has completed */
.end
```

Figure 2. Assembly-language code that counts consecutive ones.

Assemble and execute the program in Figure 2 as follows:

1. Open the Altera Monitor Program and configure it to use the system created in Part I and the application program in Figure 2.

2. Compile and load the program.

3. Single step through the program. Watch how the instructions change the data in the processor's registers. Notice that when the end of the program has been reached, a result of 4 is in the register r16.

4. Set the Program Counter to 0x00000008. This will allow us to execute the program again (in step 6 below), while skipping the first two instructions.

5. This time add a breakpoint at address 0x28, so that the program will automatically stop executing at the end of the program.

6. Set register r7 to 0xabcdef90. How many consecutive 1s are there in this number? Rerun the program by pressing **F3** (Continue) to see if you are correct.

Part III

Instructions are also represented as strings of 1s and 0s, similar to data. In this part, we will examine how instructions are formed.
Perform the following:

1. Reload your program (by selecting Actions > Load) to remove the memory edits done in Part II. Then, execute the program once, stopping at the end.

2. Use the Nios II Processor Reference Handbook, which is available on Altera's website, to determine the machine instruction representation of the following assembly language instructions: **and r3, r7, r16** and **sra r7, r7, r3.**

3. Use the Altera Monitor Program's memory-fill functionality to place these two instructions at memory locations 0 and 4. We should note that you will not see these updated values in the disassembly view of the Altera Monitor Program.

4. Set the Program Counter to 0x00000000. What will happen this time? To verify your answer, single step the instructions you placed at addresses 0 and 4 (to see their effect) and then execute the rest of the program.

5. Using the memory-fill feature change the last branch instruction to point to the start of the program instead of to itself. This will eliminate the need to manually edit the Program Counter.

6. Rerun the program until the number of 1s and the data being tested remain constant.

7. Now repeat steps 1 to 6, but use the instruction **srl r7, r7, r3** instead of **sra r7, r7, r3**. What is the difference?

Part IV

In most application programs there are portions of the code that will be executed multiple times from various locations inside a program. Such portions of code can be realized in the form of subroutines. A subroutine can be run from anywhere in the program by using a call instruction. The program execution returns to the call location after the subroutine has finished executing, if the subroutine ends with a ret instruction. We will now create a subroutine to calculate the number of consecutive 1s, and use it to calculate the number of consecutive 1s and the number of consecutive 0s in a given data word.

Start with the program for Part II and edit it as follows:

1. Take the code which calculates the number of consecutive 1s and make it into a subroutine. Have the subroutine use register r4 to receive the input data and register r2 for outputting the result.

2. Call the newly created subroutine twice, once to calculate the number of consecutive 1s and once to calculate the number of consecutive 0s. To calculate the number of consecutive 0s the input data must be inverted before running the subroutine.

3. Write the number of consecutive 1s into register r16 and the number of consecutive 0s into register r17.

Part V

One might be interested in the longest string of alternating 1s and 0s. For example, the binary number 101101010001 has a string of 6 alternating 1s and 0s, as highlighted here: 101**101010**001. Use the subroutine created in Part IV to count the number of consecutive bits of alternating 1s and 0s. Write the result to register r18. Assume that the two end bits can be part of the longest string. For example, 1010 has 4 consecutive bits of alternating 1s and 0s.
(Hint: What happens when the number is shifted to the right or left by 1 and XORed with the original number.)

Part VI

Perform the previous parts of this exercise using the C programming language. Create a function called count ones, which counts the number of consecutive 1s. Search through the disassembled code to find both the main and count ones subroutines. Did you write your assembly code in a similar way? What registers did the compiler use and why?

Laboratory Exercise 2

Program-Controlled Input/Output

The purpose of this exercise is to investigate the use of devices that provide input and output capabilities for a processor, and are controlled by software. We will examine these program-controlled I/O operations from both the hardware and software points of view. We will make use of parallel interfaces, *PIOs*, in a Nios II system implemented on an Altera DE2_115 board. The background knowledge needed to do this exercise can be acquired from the tutorials: *Introduction to the Altera Nios II Soft Processor* and *Introduction to the Altera SOPC Builder*, which can be found in the University Program section of the Altera web site.

The PIO interface used in this exercise, which is a component that can be generated by using the SOPC Builder, provides for data transfer in either input or output (or both) directions. The transfer is done in parallel and it may involve from 1 to 32 bits. The number of bits, n, and the direction of transfer are specified by the user through Altera's SOPC Builder. The PIO interface can contain the four registers shown in Figure 1.

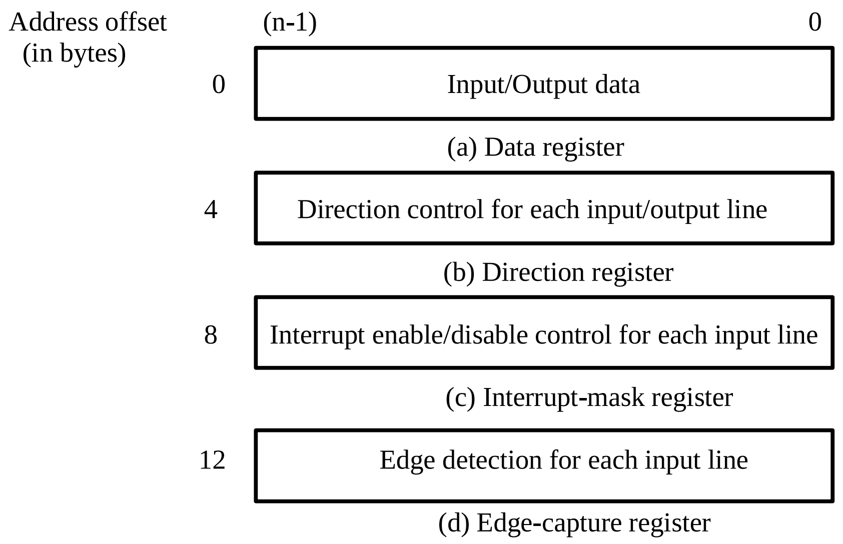

Address offset
(in bytes)
(n-1)
0
0
Input/Output data
(a) Data register
4
Direction control for each input/output line
(b) Direction register
8
Interrupt enable/disable control for each input line
(c) Interrupt-mask register
12
Edge detection for each input line
(d) Edge-capture register

Figure 1. The registers in the PIO interface.

Each register is n bits long. The registers have the following purpose:

- *Data* register holds the n bits of data that are transferred between the PIO interface and the Nios II processor. It can be implemented as an input, output, or a bidirectional register by the SOPC Builder.
- *Direction* register defines the direction of the transfer for each of the n data bits when a bidirectional interface is generated.
- *Interrupt-mask* register is used to enable interrupts from the input lines connected to the PIO.
- *Edge-capture* register indicates when a change of logic value is detected in the signals on the input lines connected to the PIO. Not all of these registers are generated in a given PIO interface. For example, the *Direction* register is included only when a bidirectional interface is specified.

Not all of these registers are generated in a given PIO interface. For example, the Direction register is included only when a bidirectional interface is specified.

The PIO registers are accessible as if they were memory locations. Any base address that has the four least significant bits equal to 0 can be assigned to a PIO (this may be done automatically by the SOPC Builder). This becomes the address of the *Data* register. The addresses of the other three registers have offsets of 4, 8, or 12 bytes (1, 2, or 3 words). A full description of the PIO module can be found in the document *PIO Core with Avalon Interface*, which is available in the literature section of Altera's web site.

The application task in this exercise consists of adding together a set of signed 8-bit numbers that are entered via the toggle switches on Altera's DE2_115 board. The resulting sum is displayed on the LEDs and 7-segment displays.

Part I

Use 8 toggle switches, $SW7-0$, as inputs for entering numbers. Use the green lights, $LEDG7-0$, to display the number defined by the toggle switches. Use the 16 red lights, $LEDR15-0$, to display the accumulated sum. A Nios II system, which includes three PIO interfaces, is the hardware needed for our task. One PIO circuit, connected to the toggle switches, will provide the input data that can be read by the processor. Two other PIO circuits, connected to the green and red lights, will serve as the output interfaces to allow displaying the number selected by the switches and the accumulated sum, respectively.

Realize the required hardware by implementing a Nios II system on the DE2_115 board, as follows:

1. Create a new Quartus II project. Select Cyclone IV EP4CE115F29C7 as the target chip, which is the FPGA chip on the Altera DE2_115 board.

2. Use the SOPC Builder to generate the desired circuit, called *nios_system*, which comprises:

   - Nios II/s processor with JTAG Debug Module Level 1; select the following options:

     – Embedded Multipliers for Hardware Multiply

     – Hardware Divide

   - On-chip memory - RAM mode and 32 Kbytes in size

   - An 8-bit PIO input circuit

   - An 8-bit PIO output circuit

   - A 16-bit PIO output circuit

   The SOPC Builder will automatically assign the names such as *pio_0*, *pio_1* and *pio_2* to the three PIO components. You can change these names to something that is more meaningful in the context of a specific design. For example, we can choose the names *new_number*, *green_LEDs* and *red_LEDs*.

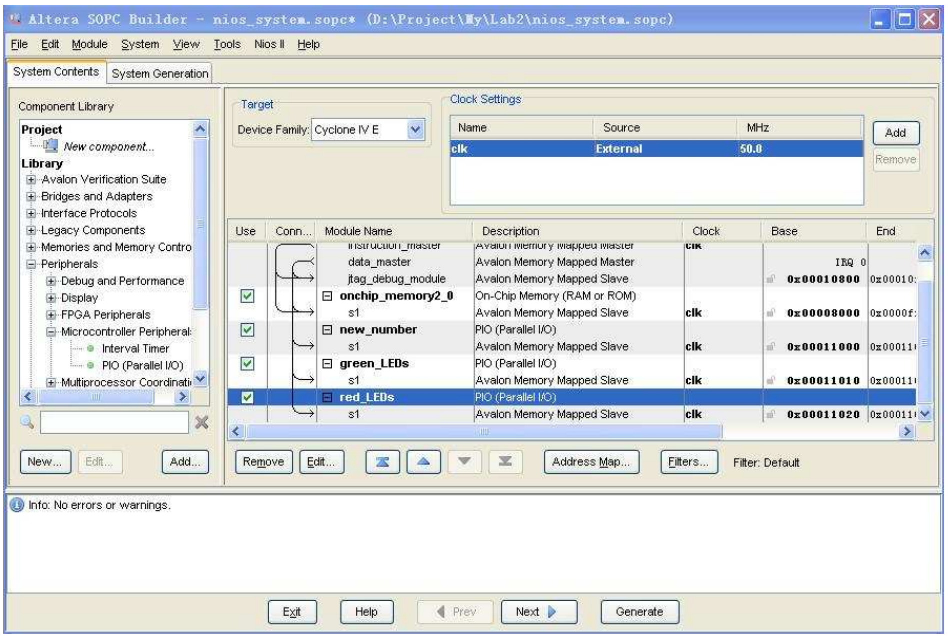
3. From the **System** menu, select **Auto-Assign Base Addresses**. This will assign addresses to all components in the designed system. The result will be a system such as the one shown in Figure 2. Observe the assigned addresses.

4. Instantiate the generated *nios_system* within a Verilog/VHDL file, which also defines the required connections to the switches and LEDs on the DE2_115 board.

5. Assign the pins needed to make the necessary connections, by importing the pin-assignment file *DE2_115 pin assignments.csv*.

6. Compile the Quartus II project.

7. Program and configure the Cyclone IV FPGA on the DE2_115 board to implement the generated system.



Altera SOPC Builder - nios_system.sopc* (D:\Project\My\Lab2\nios_system.sopc)

Device Family: Cyclone IV E

Name	Source	MHz
clk	External	50.0

Use	Conn...	Module Name	Description	Clock	Base	End
		instruction_master	Avalon Memory Mapped Master	clk		
		data_master	Avalon Memory Mapped Master		IRQ 0	
		jtag_debug_module	Avalon Memory Mapped Slave		0x00010800	0x00010:
✓		onchip_memory2_0	On-Chip Memory (RAM or ROM)			
		s1	Avalon Memory Mapped Slave	clk	0x00008000	0x0000f:
✓		new_number	PIO (Parallel I/O)			
		s1	Avalon Memory Mapped Slave	clk	0x00011000	0x00011
✓		green_LEDs	PIO (Parallel I/O)			
		s1	Avalon Memory Mapped Slave	clk	0x00011010	0x00011
✓		red_LEDs	PIO (Parallel I/O)			
		s1	Avalon Memory Mapped Slave	clk	0x00011020	0x00011

Filter: Default

Info: No errors or warnings.

Figure . The Nios II system implemented by the SOPC Builder.

Part II

Implement the desired task using the Nios II assembly language, as follows:

1. Write a program that reads the contents of the switches, displays the corresponding value on the green LEDs, adds this number to a sum that is being accumulated, and displays the sum on the red LEDs.

2. Use the *Altera Monitor Program* software to assemble and download your program.

3. Single-step through the program and verify its correctness by inputting several numbers. Note that single stepping through the program will allow you change the input numbers without reading the same number multiple times.

Part III

In this part, we want to add the ability to run the application program continuously and control the reading of new numbers by including a pushbutton switch which is activated by the user when a new number is ready to be read. The desired operation is that the user provides the next number by setting the toggle switches accordingly and then pressing a pushbutton switch to indicate that the number is ready for reading.

To accomplish this task it is necessary to implement a mechanism that monitors the status of the circuit used to input the numbers. A commonly-used I/O scheme is to use a *status flag* which is originally cleared to 0. This flag is then set to 1 as soon as the I/O device interface is ready for the next data transfer. Upon transferring the data, the flag is again cleared to 0. Thus, the processor can *poll* the status flag to determine when an I/O data transfer can be made.

In our case, the I/O device is the user who manually sets the toggle switches. The I/O interface is a PIO circuit generated by the SOPC Builder. To provide a status flag, we will generate a special one-bit PIO circuit and use its edge-capture capability. This PIO is very similar to the regular PIO, and it conforms to the register map in Figure 1. It is defined in the directory *altera_up_avalon_DE2_pio*, which has to be included in your project. The directory can be obtained from the Altera University Program site:

ftp://ftp.altera.com/up/pub/University_Program_IP_Cores/DE2_pio.zip

Perform the following steps:

1. Create a new Quartus II project and implement the same system as done in Part I.

2. Copy the *altera_up_avalon_DE2_pio* directory into your project directory. Open the SOPC Builder, which shows the existing Nios II subsystem. To make the *altera_up_avalon_DE2_pio* directory visible to the SOPC Builder, click in the File menu of the SOPC Builder on the item Refresh Component List. The DE2_PIO component will be listed under Avalon Components > University Program DE2_board.

3. Generate a status_flag PIO using the *DE2_PIO* component. Configure it to be an input port that is one bit wide. Also, in the Input Options tab select the Synchronously capture feature activated by the Falling edge.

4. Generate the new Nios II subsystem.

5. Modify your Verilog/VHDL file that specifies the complete system. Use the pushbutton switch *KEY*0 as the input to the status_flag PIO (the pushbutton switches are active low).

6. Do the pin assignment and compile the project.

7. Modify your application program to accept a new number when the pushbutton switch is pressed. This action will set the "status_flag" bit in the *edge-capture* register to 1. After adding the number to the accumulated sum, your program has to clear the flag by writing a 0 into the *edge-capture* register.

8. Download and run your program to demonstrate that it works properly. The program should run continuously and a new number should be added each time the pushbutton switch is pressed.

Part IV

In the previous parts the accumulated sum was displayed on the red LEDs. Now, augment your design to display this sum as a hexadecimal number on the 7-segment displays HEX3-HEX0, in addition to the red LEDs.

Part V

Augment your design and the application program to display the accumulated sum on the 7-segment displays as a decimal (rather than hexadecimal) number. The application program should do the necessary number conversion.

Note: You can use the div instruction only if you specified the Hardware Divide option in Part I.

Laboratory Exercise 3

Subroutines and Stacks

The purpose of this exercise is to learn about subroutines and subroutine linkage in the Nios II environment. This includes the concepts of parameter passing and stacks.

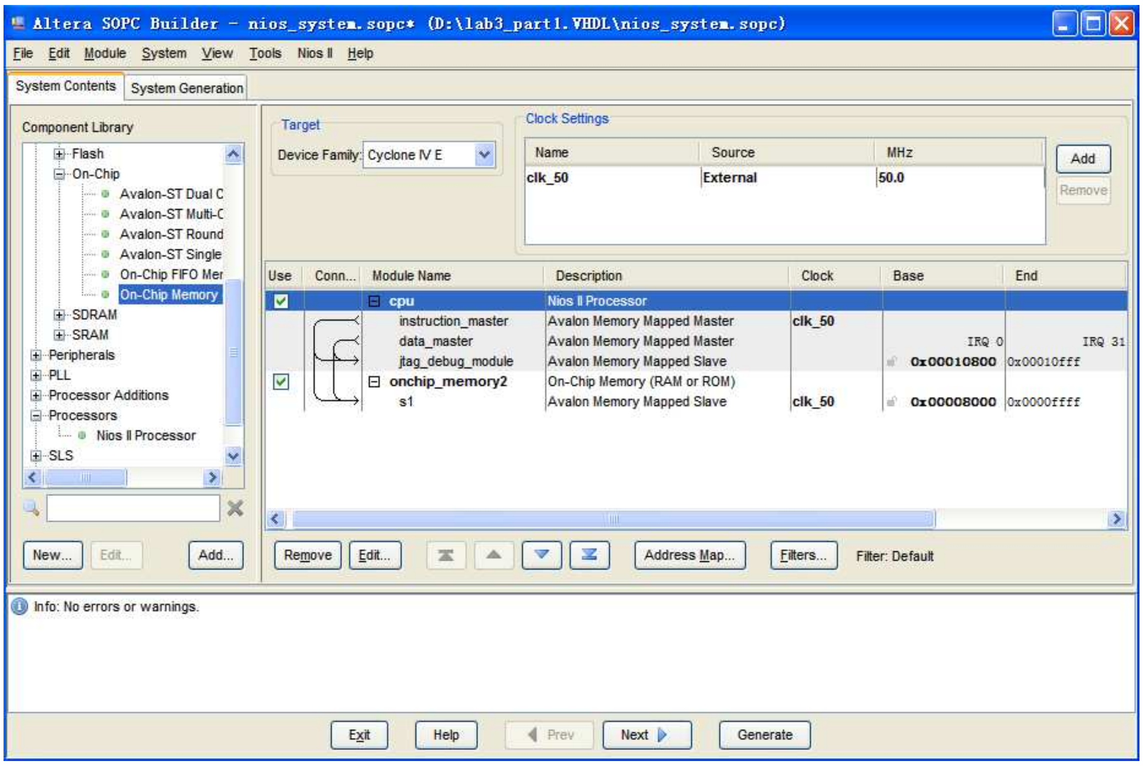
Part I

We will use a Nios II system that includes an on-chip memory block and a JTAG UART module for communication with the host computer. Implement the system as follows:

1. Create a new Quartus II project. Select Cyclone IV E EP4CE115F29C7 as the target chip, which is the FPGA chip on the Altera DE2_115 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 - RAM mode and 32 Kbytes in size

3. From the **System** menu, select **Auto-Assign Base Addresses**. You should now have the system shown in Figure 1.


Altera SOPC Builder - nios_system.sopc* (D:\lab3_part1.VHDL\nios_system.sopc)
File Edit Module System View Tools Nios II Help
System Contents System Generation
Component Library
Flash
On-Chip
Avalon-ST Dual C
Avalon-ST Multi-C
Avalon-ST Round
Avalon-ST Single
On-Chip FIFO Mer
On-Chip Memory
SDRAM
SRAM
Peripherals
PLL
Processor Additions
Processors
Nios II Processor
SLS
New... Edit... Add...
Remove Edit... Address Map... Filters... Filter: Default
Exit Help Prev Next Generate

Target — Device Family: Cyclone IV E

Clock Settings

Name	Source	MHz
clk_50	External	50.0

Use	Module Name	Description	Clock	Base	End
✓	cpu	Nios II Processor			
	instruction_master	Avalon Memory Mapped Master	clk_50		
	data_master	Avalon Memory Mapped Master		IRQ 0	IRQ 31
	jtag_debug_module	Avalon Memory Mapped Slave		0x00010800	0x00010fff
✓	onchip_memory2	On-Chip Memory (RAM or ROM)			
	s1	Avalon Memory Mapped Slave	clk_50	0x00008000	0x0000ffff

Info: No errors or warnings.

Figure 1. The Nios II system implemented by the SOPC Builder.

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

5. Instantiate the generated Nios II system within a Verilog/VHDL module.

6. Assign the pin connections:
   • clk **- PIN Y2** (This is a 50 MHz clock)
   • reset_n **- PIN M23** (This is the pushbutton switch *KEY*0)

7. Compile the Quartus II project.

8. Program and configure the Cyclone IV E FPGA on the DE2_115 board to implement the generated system.

Part II

We wish to sort a list of 32-bit positive numbers in descending order. The list is provided in the form of a file in which the first 32-bit entry gives the size (number of items) of the list and the rest of the entries are the numbers to be sorted. Implement the desired task, using the Nios II assembly language, as follows:

1. Write a program that can sort a list contained in a file that is loaded in the memory starting at location LIST_FILE. Assume that the list is large enough so that it cannot be replicated in the available memory. Therefore, the sorting process must be done “in place”, so that both the sorted list and the original list occupy the same memory locations.

2. Compile and download your program using the Altera Monitor Program.

3. Create a sample list, load it into the memory, and run your program.

Note: The file that contains the list can be loaded into the memory by using the Debug Client.

Part III

In this part, we will use a subroutine to realize the sorting task. To make the subroutine general, the contents of registers used by the subroutine have to be saved on the stack upon entering and restored before leaving the subroutine. Note that the stack has to be created by initializing the stack pointer *sp*, register *r27*. It is a common practice to start the stack at a high-address memory location and make it grow in the direction of lower addresses. The stack pointer has to be adjusted explicitly when an entry is placed on or removed from the stack. To make the stack grow from high to low addresses, the stack pointer has to be decremented by 4 before a new entry is placed on the stack and it has to be incremented by 4 after an entry is removed from the stack. Implement the previous task by modifying your program from Part II as follows:

1. Write a subroutine, called SORT, which can sort a list of any size placed at an arbitrary location in the memory. Assume that the size and the location of the list are parameters that are passed to the subroutine

via registers, such that

- The parameter *size* is given by the contents of Nios II register *r2*.
- The address of the first entry in the list is given by the contents of register *r3*.

2. Write the main program which initializes the stack pointer, places the required parameters into registers *r2* and *r3*, and then calls the subroutine SORT. The list is loaded into the memory at location LIST_FILE.

3. Compile and download your program.

4. Create a sample list, load it into the memory, and run your program.

Part IV

Modify your program from Part III so that the parameters are passed from the main program to the subroutine via the stack, rather than through registers. Compile, download, and run your program.

Part V

A Nios II processor uses the *ra* register (*r31*) to hold the return address when a subroutine is called. In the case of nested subroutines, where one subroutine calls another, it is necessary to ensure that the original return address is not lost when a new return address is placed into the *ra* register. This can be done by storing the original return address on the stack and then reloading it into the *ra* register upon return from the second subroutine.

To demonstrate the concept of nested subroutines, we will use the computation of the factorial of a given integer n. The factorial of n is determined as

$$n! = n(n-1)(n-2) \cdot \cdot \cdot \times 2 \times 1$$

It can also be computed recursively as

$$n! = n(n-1)!$$

Note that 0! = 1.

Write a program that uses recursion to compute the factorial of n. The program has to include a subroutine, called FACTOR, which calls itself repeatedly until the desired factorial has been determined. The main program should pass n as a parameter to the subroutine by placing it on the stack.

Note: Since your subroutine will make use of the multiply instruction, mul, it is essential that in Part I you generated the Nios II/s version of the processor. The economy version, Nios II/e, does not implement the mul instruction.

Compile, download, and run your program. Verify its correctness by trying different values of n.

Laboratory Exercise 4

Polling and Interrupts

The purpose of this exercise is to learn how to send and receive data to/from I/O devices. There are two methods used to indicate whether or not data can be sent or is ready to be received to/from I/O devices. The first method, polling, is where the processor queries devices to see if they can receive data or have data available. The second method, interrupts, is when devices indicate to the processor that they can receive data or that they have data available, without the processor explicitly requesting.

A simple and commonly used scheme for transferring data between a processor and an I/O device is known as the Universal Asynchronous Receiver Transmitter (UART). A UART interface (circuit) is placed between the processor and the I/O device. It handles data one 8-bit character at a time. The transfer of data between the UART and the processor is done in parallel fashion, where all bits of a character are transferred at the same time using separate wires. However, the transfer of data between the UART and the I/O device is done in bit-serial fashion, transferring the bits one at a time.

Altera's SOPC Builder can implement an interface of the UART type for use in Nios II systems, which is called the JTAG UART. This circuit can be used to provide a connection between a Nios II processor and the host computer connected to Altera's DE2_115 board. Figure 1 shows a block diagram of the JTAG UART circuit. On one side the JTAG UART connects to the Avalon switch fabric, while on the other side it connects the host computer via the USB-Blaster interface. The JTAG UART core contains two registers: Data and Control, which are accessed by the processor as memory locations. The address of the Control register is 4 bytes higher than the address assigned to the Data register. The core also contains two FIFOs that serve as storage buffers, one for queuing up the data to be transmitted to the host and the other for queuing up the data received from the host.

Figure 2 gives the format of the registers.

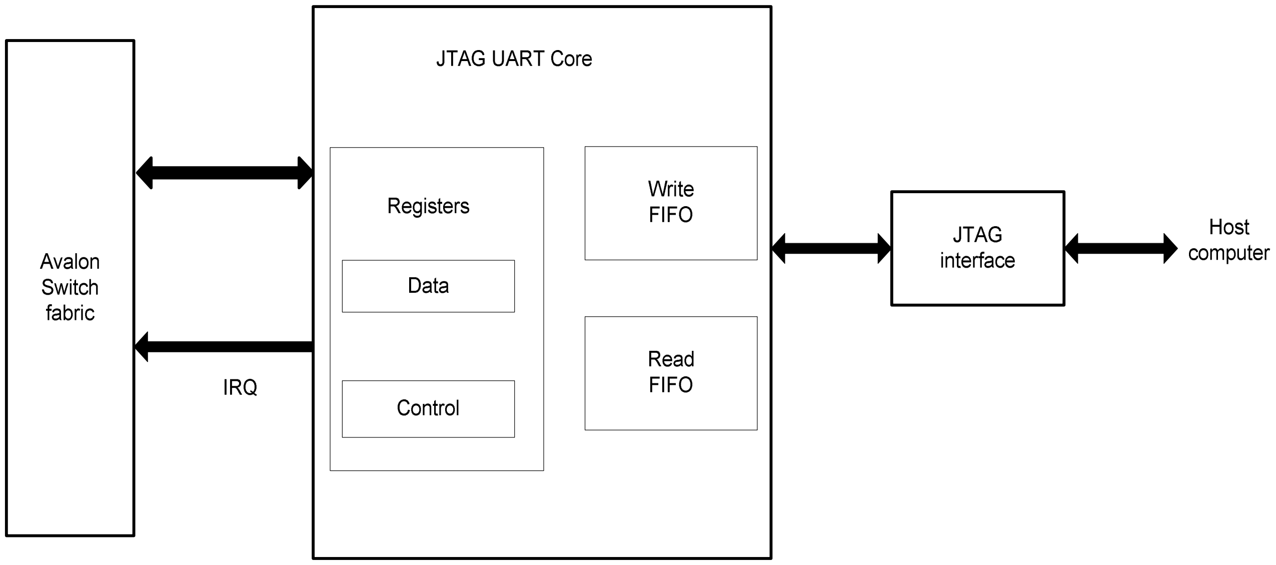
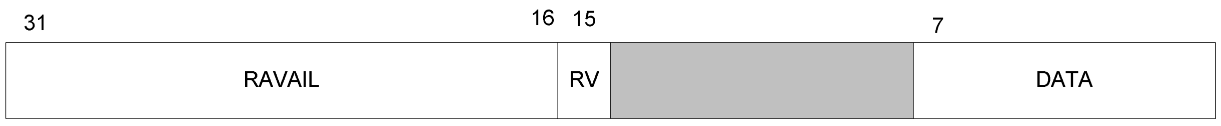
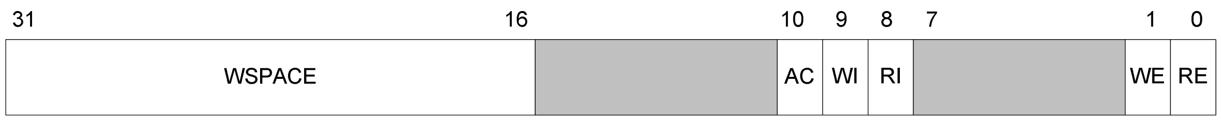


Figure 1. A block diagram of the JTAG UART circuit



(a) Data register



(b) Control register

Figure 2. Registers in the JTAG UART

The fields in the *Data* register are used as follows:

- b7−0 (DATA) is an 8-bit character to be placed into the Write FIFO when a Store operation is performed by the processor, or it is a character read from the Read FIFO when a Load operation is performed.
- b15 (RVALID) indicates whether the DATA field contains a valid character that may be read by the processor. This bit is set to 1 if the DATA field is valid; otherwise it is cleared to 0.
- b31−16 (RAVAIL) indicates the number of characters remaining in the Read FIFO (after this read).

The fields in the *Control* register are used as follows:

- b0 (RE) enables the read interrupts when set to 1.
- b1 (WE) enables the write interrupts when set to 1.
- b8 (RI) indicates that a read interrupt is pending if the value is 1. Reading the *Data* register clears the bit to 0.
- b9 (WI) indicates that a write interrupt is pending if the value is 1.
- b10 (AC) indicates that there has been JTAG activity (such as the host computer polling the JTAG UART to verify that a connection exists) since the bit was cleared. Writing a 1 to AC clears it to 0.
- b31−16 (WSPACE) indicates the number of spaces available in the Write FIFO.

More information on the JTAG UART may be found in Chapter 5 of the *Altera Embedded Peripherals*

Handbook.

In this exercise, we will use the JTAG UART to transfer ASCII-encoded characters between a Nios II processor implemented on the DE2_115 board and the host computer. We will also make use of an "interval timer" circuit to provide fixed delays.

Part I

Use the SOPC Builder to create the system in Figure 3, which consists of a Nios II/e processor, a JTAG UART, a memory block and an Interval Timer.

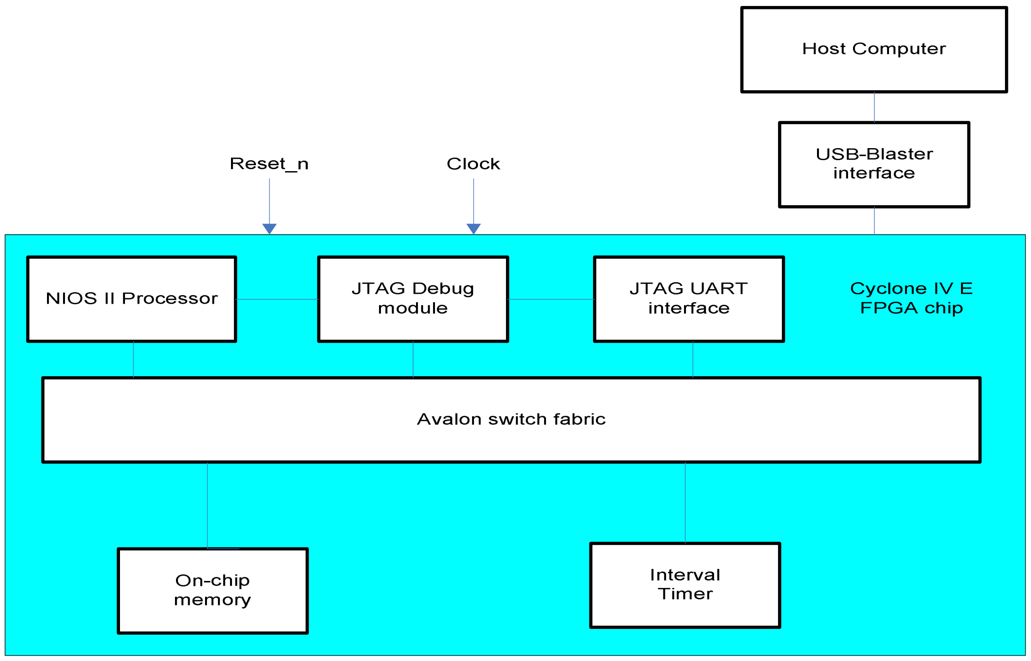


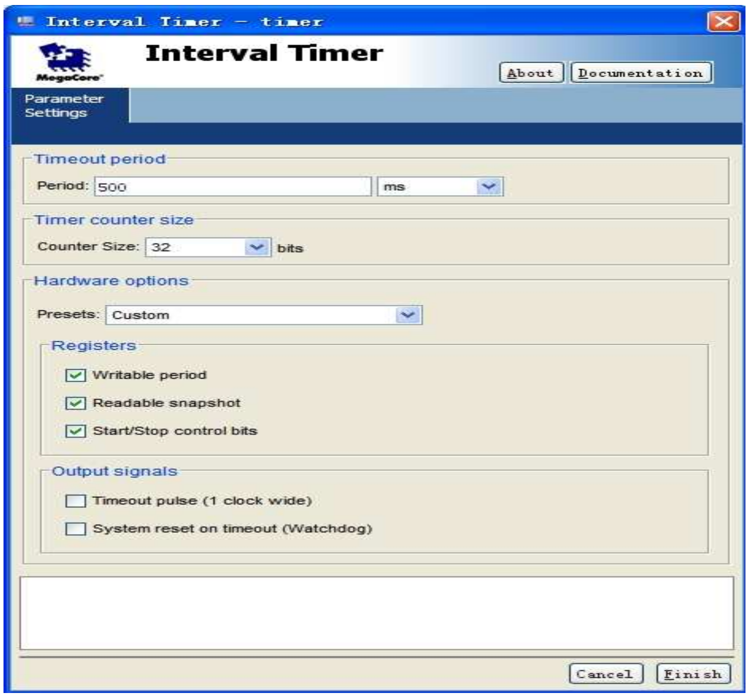
Figure 3. The desired Nios II system

Implement the system as follows:

1. Create a new Quartus II project. Select Cyclone IV E EP4CE115F29C7 as the target chip, which is the FPGA chip on the Altera DE2_115 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 - RAM mode and 32 Kbytes in size
   - JTAG UART - use the default settings
   - Interval Timer - Located in the component section named Other
     - For the Hardware Options - Preset Configurations choose Simple periodic interrupt
     - For the Timeout Period choose a Fixed Period of 500 msec

   as shown in Figure 4.


Interval Timer - timer

Interval Timer

Timeout period

Period: 500 ms

Timer counter size

Counter Size: 32 bits

Hardware options

Presets: Custom

Registers

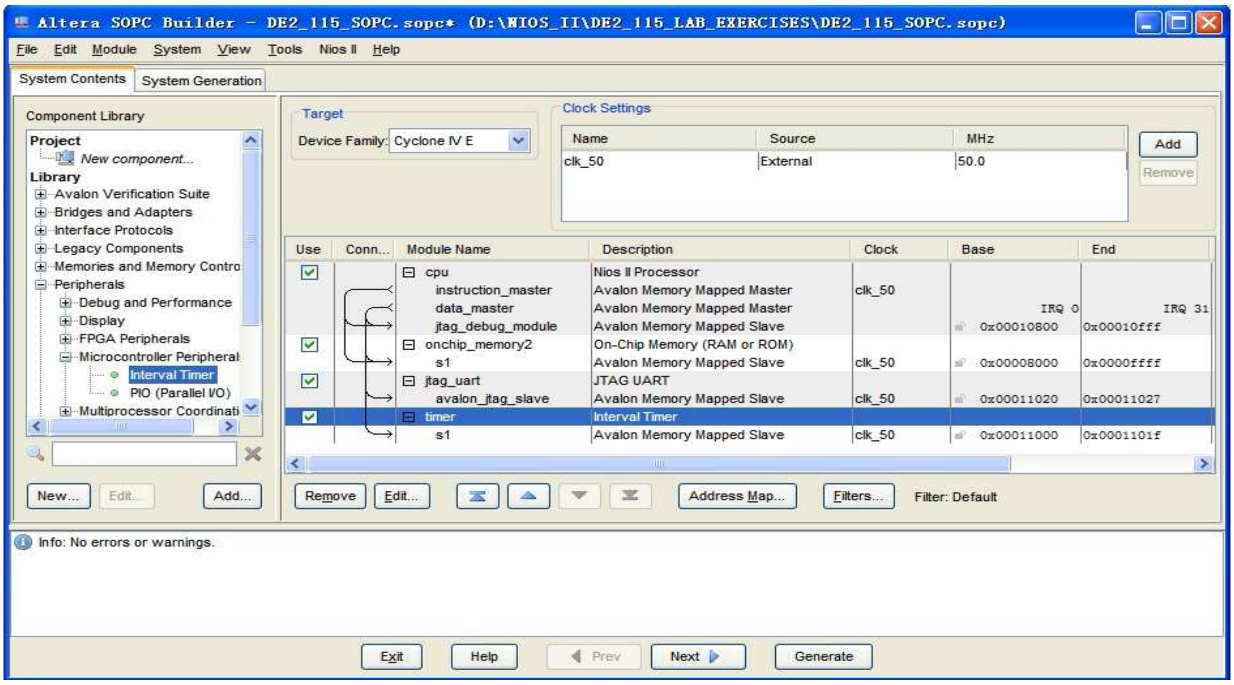
- [x] Writable period
- [x] Readable snapshot
- [x] Start/Stop control bits

Output signals

- [ ] Timeout pulse (1 clock wide)
- [ ] System reset on timeout (Watchdog)

Figure 4. Specification for the Interval Timer

3. From the **System** menu, select **Auto-Assign Base Addresses**. You should now have the system shown in Figure 5.


Altera SOPC Builder - DE2_115_SOPC.sopc* (D:\NIOS_II\DE2_115_LAB_EXERCISES\DE2_115_SOPC.sopc)

Target

Device Family: Cyclone IV E

Clock Settings

Name	Source	MHz
clk_50	External	50.0

Use	Conn...	Module Name	Description	Clock	Base	End
✓		cpu	Nios II Processor			
		instruction_master	Avalon Memory Mapped Master	clk_50		
		data_master	Avalon Memory Mapped Master		IRQ 0	IRQ 31
		jtag_debug_module	Avalon Memory Mapped Slave		0x00010800	0x00010fff
✓		onchip_memory2	On-Chip Memory (RAM or ROM)			
		s1	Avalon Memory Mapped Slave	clk_50	0x00008000	0x0000ffff
✓		jtag_uart	JTAG UART			
		avalon_jtag_slave	Avalon Memory Mapped Slave	clk_50	0x00011020	0x00011027
✓		timer	Interval Timer			
		s1	Avalon Memory Mapped Slave	clk_50	0x00011000	0x0001101f

Filter: Default

Info: No errors or warnings.

Figure 5. The Nios II system implemented by the SOPC Builder.

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

5. Instantiate the generated Nios II system within a Verilog/VHDL module.

6. Assign the pin connections:
   - clk **- PIN Y2** (This is a 50 MHz clock)
   - reset_n **- PIN M23** (This is the pushbutton switch *KEY*0)

7. Compile the Quartus II project.

8. Program and configure the Cyclone IV E EP4CE115F29C7 on the DE2_115 board to implement the generated system.

Part II

The JTAG UART can send ASCII characters to the Altera Monitor Program, which will display these characters in its terminal window. When the WSPACE field in the *Control* register of the JTAG UART has a non-zero value the JTAG UART can accept a new character to be written to the Altera Monitor Program. To write a character to the Debug Client, poll (continuously read) this register until space is available. Once space is available the ASCII character can be written into the *Data* register of the JTAG UART.

Write a Nios II assembly-language program to display the letter Z approximately every 500 ms in the terminal window of the Altera Monitor Program. Create and execute the program as follows:

1. Using the Nios II assembly language, write a loop which reads the *Control* register in the JTAG UART and keeps looping until there is some write space available.

2. Write the letter Z to the *Data* register.

3. Using the Altera Monitor Program, compile and load the assembly-language program.

4. Run this program using the single step feature only. If you run this program using the Continue mode, the character will be sent to the terminal window faster than the Altera Monitor Program can handle.

5. In the assembly-language code, create a delay loop so that characters are only printed approximately every half second.

6. Recompile, load and run the program.

Part III

The JTAG UART can receive ASCII characters from the terminal window, as well as write them. The RVALID bit, b15, in the *Data* register indicates whether or not a value in the DATA field is a valid received ASCII character. If more characters are still waiting to be read, the RAVAIL field will have a non-zero value.

Write a program that implements a “typewriter-like” task; that is, read each character that is received by the JTAG UART from the host computer and then display this character in the terminal window of the Debug Client. Use polling to determine if a new character is available from the JTAG UART. Note: the cursor must be in the terminal window of the Debug Client to write characters to the JTAG UART's receive port.

Part IV

Polling the JTAG UART is inefficient, due to the overhead of reading its registers to determine the UART's state. The overhead of determining if a new character is available significantly impacts the performance of the program. Instead of polling, it is possible to use the interrupt mechanism, which allows the processor to do useful work while it is waiting for an I/O transfer to take place.

Create an interrupt-service routine to read characters received by the JTAG UART from the host computer. Place the interrupt-service routine at the hex address 0x20, because this is the default location for the *exception handler* as chosen by the SOPC Builder. The exception return address in the *ea* register must be decremented by 4 for external interrupts. Figure 5 gives a skeleton of the interrupt-service routine written in the Nios II assembly language.

```
.include "nios macros.s"
.text
.org 0x20 /* Place the interrupt service routine */
/* at the appropriate address */
ISR:
rdctl et, ctl4 /* Check if an external */
beq et, r0, SKIP EA DEC /* interrupt has occurred */
subi ea, ea, 4 /* If yes, decrement ea to execute */
/* interrupted instruction */
SKIP EA DEC:
... the interrupt-service routine
END ISR:
eret /* Return from exception */
.global start
start: /* Program start location */
```

Figure 6. Assembly language code skeleton for the interrupt-service routine.

Nios II's control register *ctl3*, also referred to as *ienable*, enables interrupts on an individual basis. Note that when the system was created in Part I, the JTAG UART was placed at interrupt level 0. This means that bit *0* of the control register *ctl3* must be set to 1 to enable the JTAG UART's interrupts.
Perform the following:

1. Create an interrupt-service routine to read a character from the JTAG UART. Note that
   - The interrupt service routine must be placed at the memory address 0x20.
   - To enable interrupts, appropriate values must be written to the *Control* register of the JTAG UART,

and the Nios II's control registers *ctl0* and *ctl3*.

2. In your interrupt service routine, use the polling approach to display the characters received from the host computer in the terminal window of the Altera Monitor Program.

3. Compile, load and run your program.

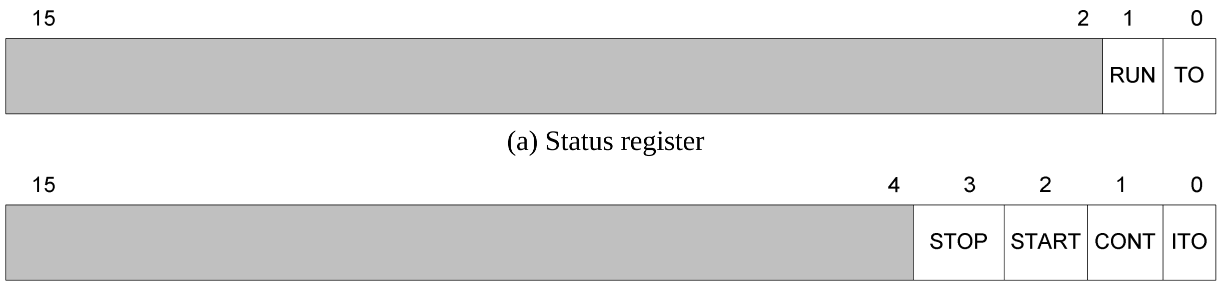
If your program does not work at a first try, you will have to debug it and fix the errors. One aid in debugging is the single-step feature of the Altera Monitor Program, which allows the user to observe the flow of execution and the contents of Nios II registers as each instruction is being executed. However, this approach cannot be used when interrupts are involved, because interrupts are automatically disabled when single stepping through a program. Therefore, use breakpoints as a debugging aid.

Note also that interrupts are automatically disabled when the execution of an interrupt-service routine begins and re-enabled upon exit from this routine. This means that if some application required nested interrupts, the interrupts would have to be re-enabled within the interrupt-service routine.

Part V

In this part we wish to write a program that uses interrupts to read characters received by the JTAG UART from the host computer and displays the last character received repeatedly every 500 milliseconds. In Part II we used a delay loop to generate an approximate time interval of this length. Now, we want to use the Interval Timer circuit for this purpose. The Interval Timer should interrupt the processor every 500 ms at which point a character should be written to the Debug Client's terminal window.

The Interval Timer has an internal counter which is set to a specified value and then decremented in each clock cycle. When the counter reaches 0, a "timeout" event is said to have occurred. At this point the Interval Timer can raise an interrupt request and the counter can be reset to the specified value. The Interval Timer has a set of 16-bit registers that can be accessed as memory locations, similar to the JTAG UART. Two of these registers, *Status* and *Control*, are shown in Figure 7. The address of the *Status* register is the base address assigned to the Interval Timer, while the address of the *Control* register is 4 bytes higher.



(a) Status register

(b) Control register

Figure 7. Registers in the Interval Timer

The bits in the *Status* register are used as follows:

- b0 (TO) is the timeout bit. It is set to 1 when the internal counter in the Interval Timer reaches 0. It remains
  set until explicitly cleared by the processor writing a 0 to it.

- b1 (RUN) is equal to 1 when the internal counter is running; otherwise, it is equal to 0. This bit is not changed by a write operation to the *Status* register.

The bits in the *Control* register are used as follows:

- b0 (ITO) enables the Interval Timer interrupts when set to 1.

- b1 (CONT) determines how the internal counter behaves when it reaches 0. If CONT = 1, the counter runs
  continuously by reloading the specified initial count value; otherwise, it stops when it reaches 0.

- b2 (START) causes the internal counter to start running when set to 1 by a write operation.

- b3 (STOP) stops the internal counter when set to 1 by a write operation.
  More information on the Interval Timer may be found in chapter 12 of the *Altera Embedded Peripherals Handbook.*

To enable both interrupts, from the Interval Timer and the JTAG UART for reading characters (from Part IV),the bits b1 and b0 of the control register *ctl3* must both be set to 1. As seen in Figure 5, the JTAG UART is on interrupt line 0 (or b0) and the Interval Timer is on interrupt line 1 (or b1). The control register *ctl4*, also referred to as *ipending*, can be used to determine which interrupt has occurred. If an interrupt is disabled using the control register *ctl3*, it will not cause the interrupt-service routine to execute, nor will it show as being triggered in the control register *ctl4*, even if the device is driving its interrupt-request line to Perform the following steps:

1. Modify the program from Part IV, so that the main program enables interrupts and then waits in an infinite loop.

2. Modify the interrupt-service routine to handle both the Interval Timer and the JTAG UART's read interrupts.

3. To enable interrupts, appropriate values must be written to the Control register of the JTAG UART, the Control register of the Interval Timer and the Nios II control registers ctl0 and ctl3.

4. Compile, load and run your program.

Laboratory Exercise 5

Bus Communication

The purpose of this exercise is to learn how to communicate using a bus. In the designs generated by using Altera’s SOPC Builder, the Nios II processor connects to peripheral devices by means of the Avalon Switch Fabric. To connect to the switch fabric, an SOPC Builder *component* is required. To make it possible to investigate the bus communication, without requiring the creation of an SOPC Builder component, this exercise will use the *Avalon to External Bus Bridge* SOPC component. The bridge allows the designer to create a peripheral device and connect it to the Nios II system in the Quartus II software. It creates a bus-like 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.

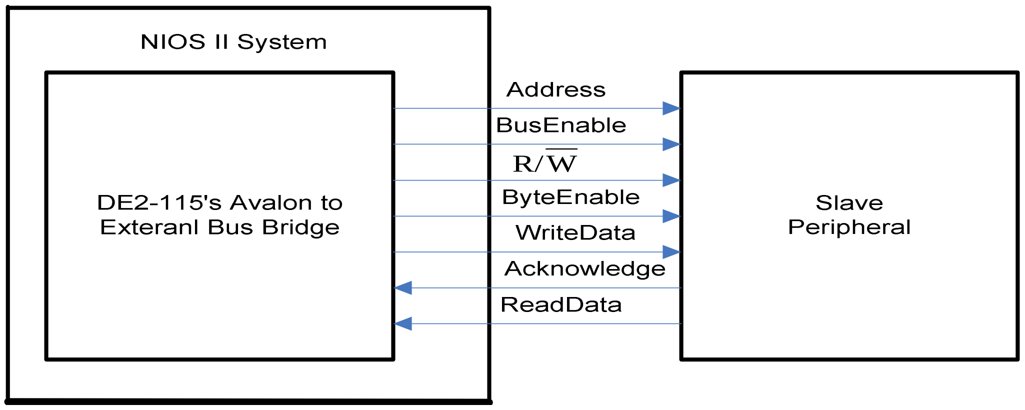
- BusEnable – 1 bit. Indicates that all other signals are valid, and a data transfer should occur.

- $R/\overline{W}$ – 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.

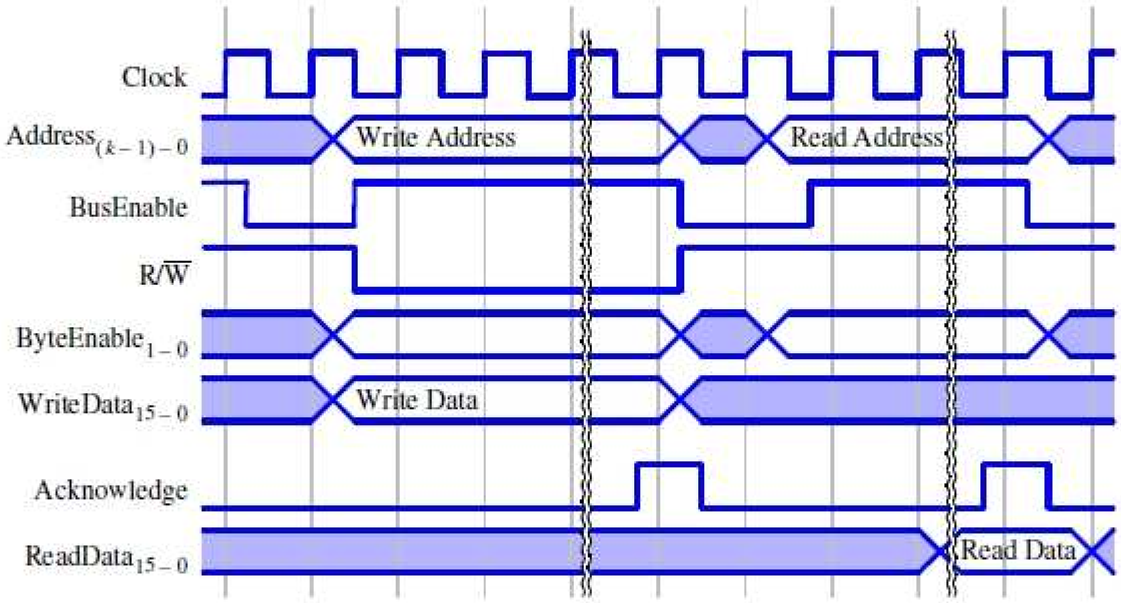
The bus is synchronous – all bus signals to the peripheral device must be read on the rising edge of the clock. To initiate a transfer the *Address*, $R/\overline{W}$, *ByteEnable* and possibly *WriteData* signals are set to the appropriate values. Then, the *BusEnable* signal is set to 1.

If the $R/\overline{W}$ *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 $R/\overline{W}$ 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 1. The Avalon to External Bus Bridge signals.

Part I

Figure 2 indicates the system that we wish to design and implement. The part of the system that consists of a Nios II/s processor, a JTAG UART, an on-chip memory block and an Avalon to External Bus Bridge can be generated using the SOPC Builder. This should produce the system given in Figure 3. The slave peripheral will be a Verilog/VHDL module which you will design. The Avalon to External Bus Bridge connects the previously discussed external bus to the Avalon Switch Fabric of the Nios II system. The switch fabric is the main interconnection network for peripherals in a system generated by the SOPC Builder.

The slave peripheral comprises just four 16-bit registers plus the circuitry needed to display the contents of these registers on the 7-segment displays on the DE2_115 board. The registers are accessible as memory locations, so that the Nios II processor can write data into them.

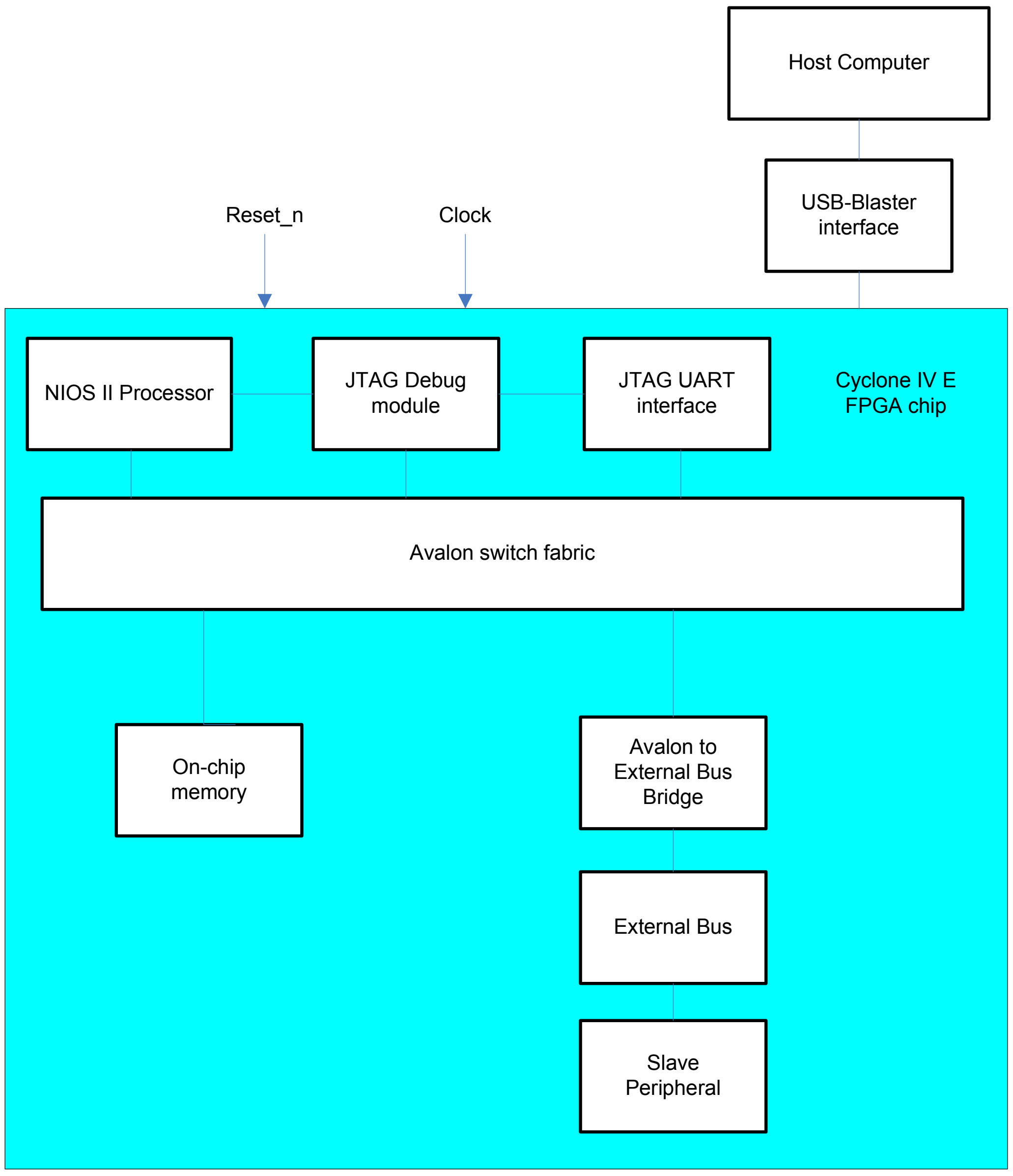


Figure 2. The desired Nios II system

To facilitate the implementation of the desired peripheral, three modules defined in Verilog/VHDL are provided. One of these modules are provided in full. The other two are given in a skeleton form and have to be completed as a part of this exercise. These modules are:

- Lab5_Part1 (which is provided in full)

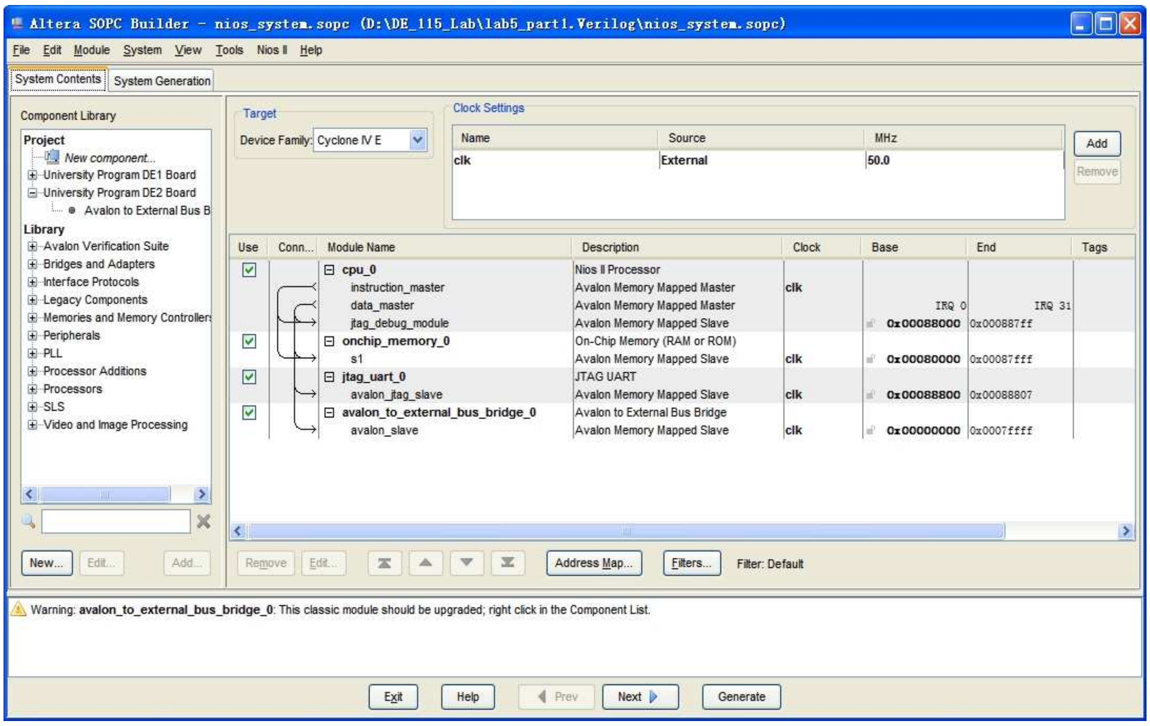
- Peripheral on External Bus (skeleton is provided)

- Seven Segment Display (skeleton is provided)

The Verilog/VHDL code for these modules is provided on the Altera University Program site:

ftp://ftp.altera.com/up/pub/Laboratory_Exercises/Comp_Org/comporg_lab5_design_files.zip

Once the system in Figure 3 has been generated, modify the Verilog/VHDL module, Peripheral on External Bus to connect it to the Avalon to External Bus Bridge signals. Also, modify this module such that it contains four 16-bit registers. Each of these registers should be mapped to one quarter of the address space assigned to the Avalon to External Bus Bridge by the SOPC Builder. Use the 7-segment displays $HEX3-0$ to display the contents of these registers. Since only one register can be displayed at one time on the 7-segment displays, use two toggle switches, $SW1$ and $SW0$, to choose which register to display. These modules, in addition to the modules generated by the SOPC Builder, should implement the desired system in Figure 2.


Altera SOPC Builder - nios_system.sopc (D:\DE_115_Lab\lab5_part1.Verilog\nios_system.sopc)

Target

Device Family: Cyclone IV E

Clock Settings

Name	Source	MHz
clk	External	50.0

Use	Conn...	Module Name	Description	Clock	Base	End	Tags
✓		cpu_0	Nios II Processor				
		instruction_master	Avalon Memory Mapped Master	clk			
		data_master	Avalon Memory Mapped Master		IRQ 0	IRQ 31	
		jtag_debug_module	Avalon Memory Mapped Slave		0x00088000	0x000887ff	
✓		onchip_memory_0	On-Chip Memory (RAM or ROM)				
		s1	Avalon Memory Mapped Slave	clk	0x00080000	0x00087fff	
✓		jtag_uart_0	JTAG UART				
		avalon_jtag_slave	Avalon Memory Mapped Slave	clk	0x00088800	0x00088807	
✓		avalon_to_external_bus_bridge_0	Avalon to External Bus Bridge				
		avalon_slave	Avalon Memory Mapped Slave	clk	0x00000000	0x0007ffff	

Filter: Default

Warning: avalon_to_external_bus_bridge_0: This classic module should be upgraded; right click in the Component List.

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

To use the Bridge component, you have to copy the directory *altera up avalon to external bus bridge* into your project directory, before you start the SOPC Builder software. This directory is available on the Altera University Program site:

ftp://ftp.altera.com/up/pub/University_Program_IP_Cores/avalon_to_external_bus_bridge.zip

Implement the required system as follows:

1. Create a new Quartus II project called *Lab5_Part1*. Select Cyclone IV E EP4CE115F29C7 as the target chip, which is the FPGA chip on the Altera DE2_115 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 32 Kbytes
    • JTAG UART - use the default settings

- Avalon to External Bus Bridge - choose the 16-bit data width and the address range of 512 Kbytes. These parameters are chosen to simplify the connection to the SRAM chip in Part II of this exercise.In the SOPC Builder window, the desired bridge is found by selecting **Avalon Component**>**University Program** > **Bridges**>**Avalon to External Bus Bridge**.

Note: The choice of the address range of 512 Kbytes implies that k = 19 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.

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

6. Add the three Verilog/VHDL modules mentioned above to your Quartus II project.

7. Check that the generated Nios II system is instantiated correctly within the given Verilog/VHDL module Lab5_Part1.

8. Modify the Verilog/VHDL module Peripheral on External Bus to implement the bus protocol needed to connect the four 16-bit registers to the Avalon Switch.

9. Modify the Verilog/VHDL module Seven Segment Display to display the four registers on the 7-segment displays in hexadecimal format.

10. Import the pin assignments using the file *DE2-115 pin assignments.csv*.

11. Compile your Quartus II Project.

12. Program and configure the Cyclone IV E FPGA on the DE2_115 board to implement the generated system.

13. Create a Nios II assembly-language program to write a different 16-bit number into each of the four registers.

14. Use the Altera Monitor Program to compile, download and run your program. Verify that the registers can be selected for display by using the toggle switches $SW1-0$.

Part II

In this part of the exercise we wish to implement a different slave peripheral. The peripheral is to serve as a controller that provides access to the SRAM chip in the DE2_115 board – it has to connect the SRAM chip to the Avalon Switch Fabric.

The SRAM chip uses the following signals:

- *SRAM ADDR*17−0 – 18-bits, input. The addresses of the 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 4 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.

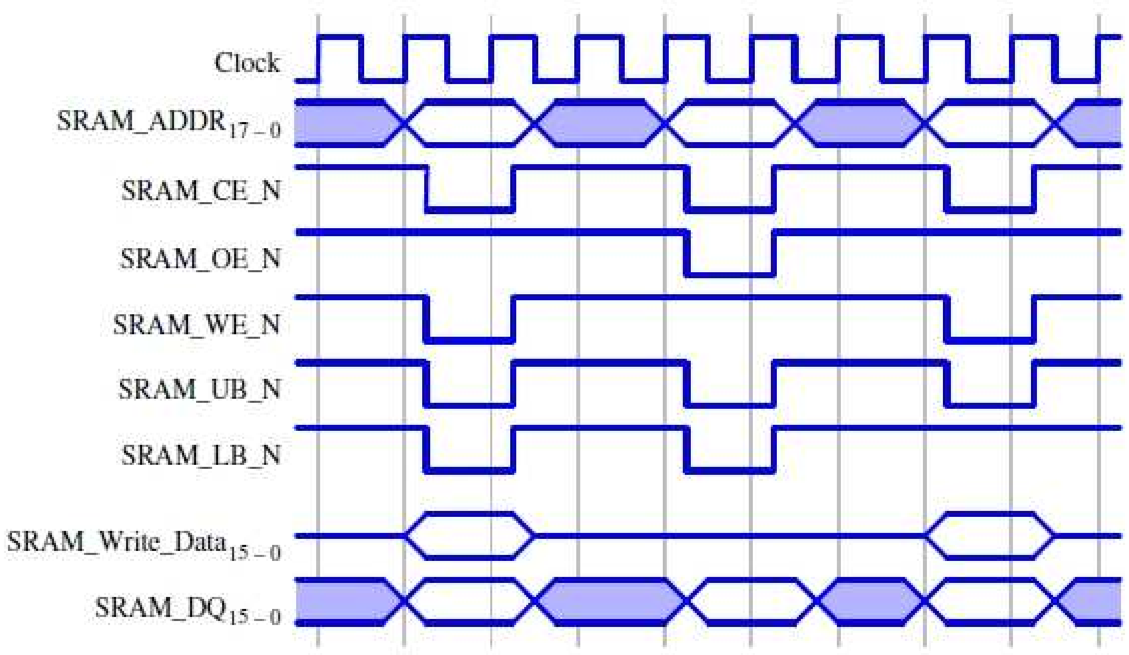


Figure 4. The SRAM signals timing diagram

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 transfer. The SRAM controller will act as the slave peripheral in Figure 1.

Perform the following steps:

1. Create a new project called *Lab5_Part2*. Use the same SOPC Builder project settings as in Part 1 of this exercise, and generate a Nios II system that corresponds to Figure 3.
2. Instantiate the generated system into your project to produce the desired controller. Two Verilog/VHDL files, *Lab5_ Part2* and *SRAM Controller* are provided (in skeleton form) to help you get started. They can be found on the the same Altera web page as the files used in Part I.
3. Compile your project and configure the Cyclone IV E chip to implement the generated system.
4. Write a Nios II assembly-language program to write some sample data to the SRAM chip and then read this data back into the processor registers.
5. Run your program to verify the correctness of your design. You can also test the SRAM controller by using the memory window in the Altera Monitor Program.

Part III

In this part we will combine the designs in Parts I and II. We want to use just one Avalon to External Bus Bridge and connect two slave peripherals to it. One peripheral will be the SRAM controller and the other peripheral will be the four registers with the associated display circuitry.

Perform the following steps:

1. Create a new project called *Lab5_Part3*.
2. Use the SOPC Builder to generate the system in Figure 3, but choose the address range of 1024 Kbytes for the Avalon to External Bus Bridge component. Note that this will change the base addresses that the SOPC Builder will auto-assign to the components of the system.
3. Prepare a Verilog/VHDL file that instantiates the Nios II system and implements the two slave peripherals such that
    - The SRAM controller uses the low-order 512 Kbytes of the 1024-Kbyte address space
    - The four registers use the remaining address space
4. Modify the other Verilog/VHDL files used in Parts I and II accordingly.
5. Compile your project and configure the Cyclone IV E chip to implement the generated system.
6. Write a Nios II assembly-language program that writes some sample data to the SRAM chip and then

reads this data into the four registers in the slave peripheral.

7. Run your program to show that the sample data is correctly displayed on the 7-segment displays.