Laboratory Exercise 1

Using an Altera Nios II System

This is an introductory exercise that involves Altera's Nios II processor. It uses a predefined computer system, called the *DE2 Media Computer*, which includes the Nios II processor. The system is implemented as a circuit that is downloaded into the FPGA device on the Altera DE2 board. This exercise illustrates how programs written in the Nios II assembly language or in the C language can be executed on the DE2 board. We will use the *Altera Monitor Program* software to compile, load and run the application programs.

For this exercise you have to know the Nios II processor architecture and its assembly language, as well as the C language. Read the tutorial *Introduction to the Altera Nios II Soft Processor*. You also have to become familiar with the monitor program; read the tutorial *Altera Monitor Program*. Both tutorials are available on Altera's University Program web site. The monitor tutorial is also included in the Altera Monitor Program package – it can be accessed by selecting Help > Tutorial in the monitor window.

Part I

In this part you will use the Altera Monitor Program to download the DE2 Media Computer circuit into the FPGA device and execute a sample program. Perform the following:

1. Turn on the power to the Altera DE2 board.
2. Open the Altera Monitor Program, which leads to the window in Figure 1.

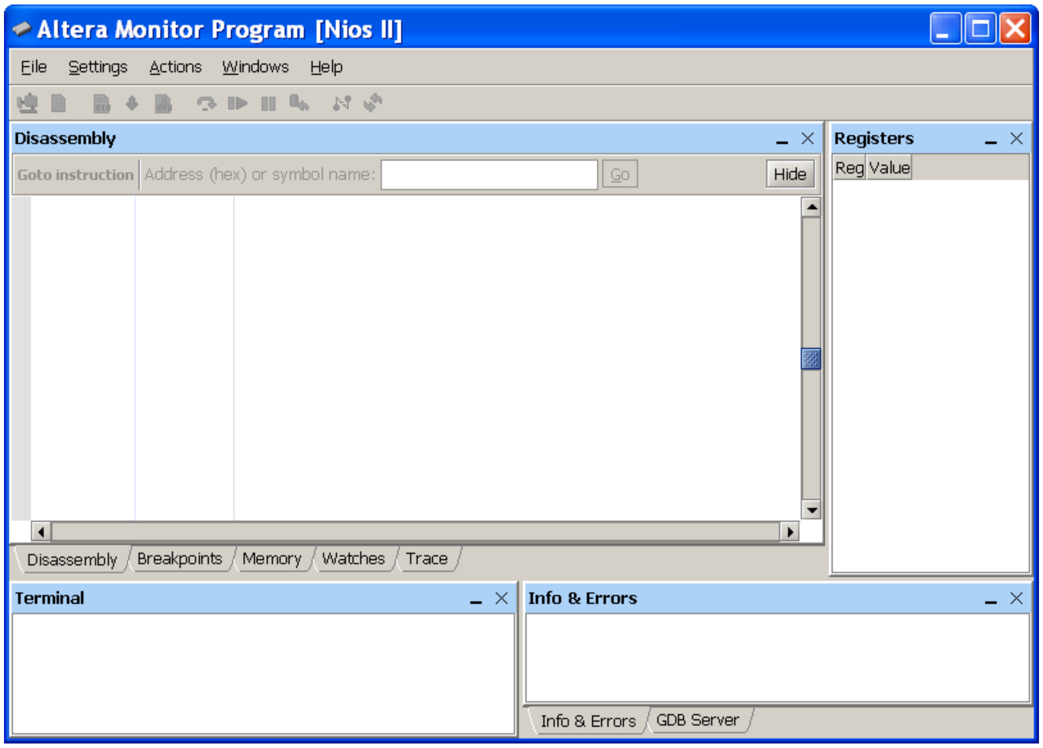


Figure 1. The Altera Monitor Program window.

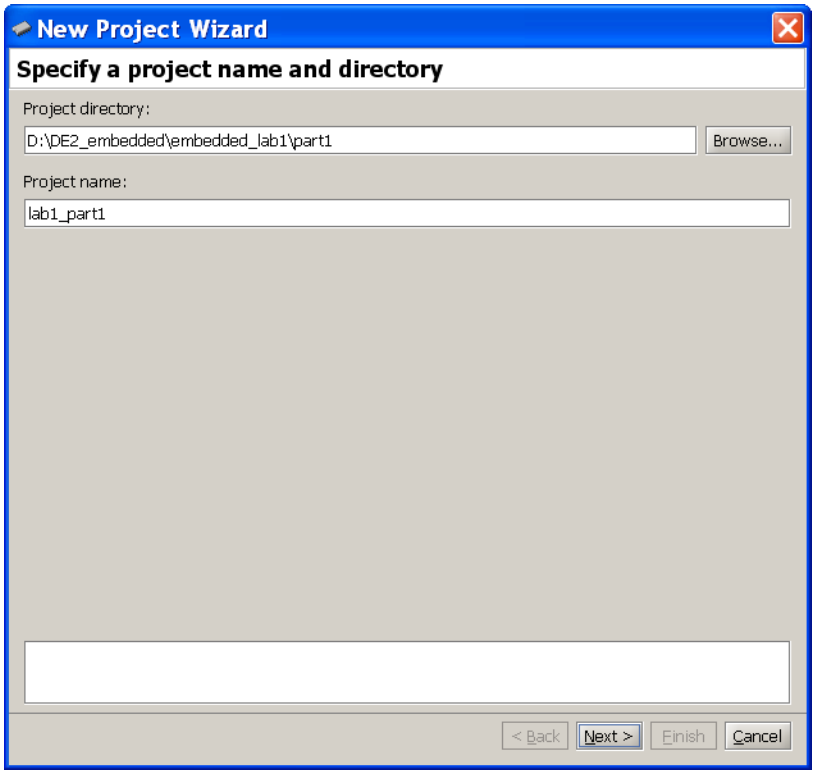
To run an application program it is necessary to create a new project. Select File > New Project to reach the window in Figure 2. Give the project a name and indicate the directory for the project; we have chosen the project name *lab1_part1* in the directory *embedded_lab1_part1*, as indicated in the figure. Click Next, to get the window in Figure 3.

3. Now, you can select your own custom system (if you have one) or a predesigned (by Altera) system. Choose the DE2 Media Computer and click Next. The display in the window will now show where files that implement the chosen system are located. This is for information purpose only; if you wanted to use a system that you designed by using Altera's Quartus II software, you would have to provide such files. Click Next.

4. In the window in Figure 4 you can specify the type of application programs that you wish to run. They can be written in either the Nios II assembly language or the C programming language. Specify that an assembly language program will be used. The Altera Monitor Program package contains several sample programs. Select the box Include a sample program with the project. Then, choose the Test Media Computer program, as indicated in the figure, and click Next.

5. The window in Figure 5 is used to specify the source file(s) that contain the application program(s). Since we have selected the *Test Media Computer* program, the window indicates the files that are used by this program. This window also allows the user to specify the starting point in the selected application program. The default symbol is *_start*, which is used in the selected sample program. Click Next.

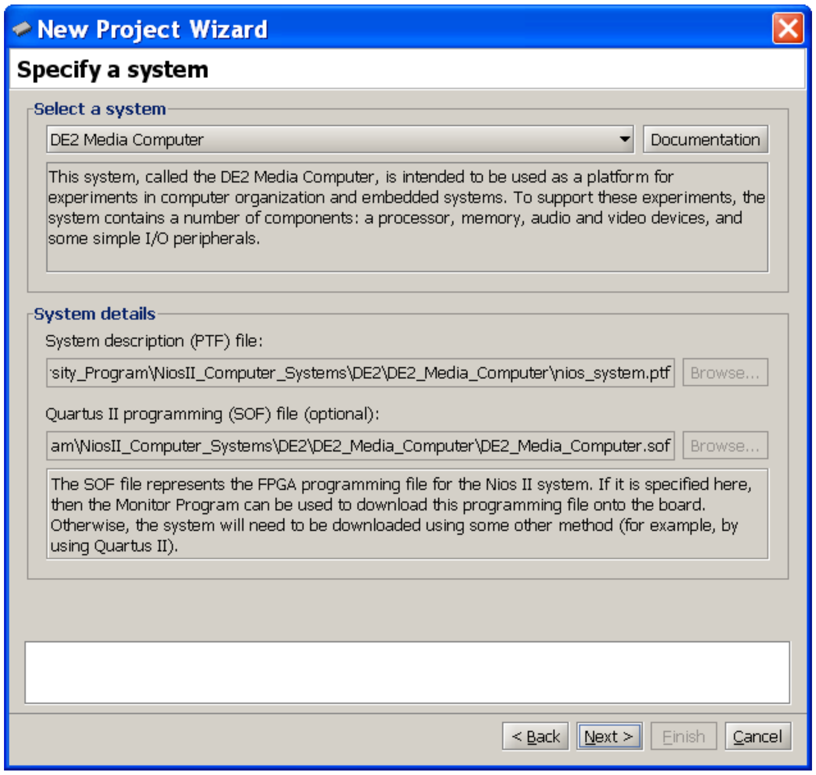
6. The window in Figure 6 indicates some system parameters. Note that the *USB-Blaster* cable is selected to provide the connection between the DE2 board and the host computer. Click Next.



Project directory:
D:\DE2_embedded\embedded_lab1\part1

Project name:
lab1_part1

Figure 2. Specify the directory and the name of the project.



DE2 Media Computer

This system, called the DE2 Media Computer, is intended to be used as a platform for experiments in computer organization and embedded systems. To support these experiments, the system contains a number of components: a processor, memory, audio and video devices, and some simple I/O peripherals.

System description (PTF) file:

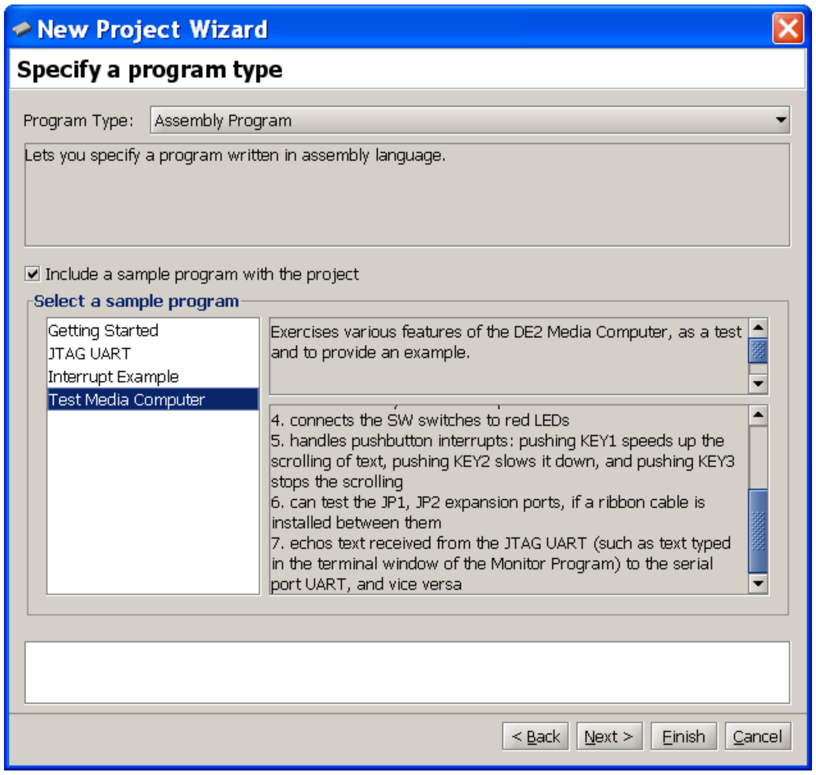
'sity_Program\NiosII_Computer_Systems\DE2\DE2_Media_Computer\nios_system.ptf

Quartus II programming (SOF) file (optional):

am\NiosII_Computer_Systems\DE2\DE2_Media_Computer\DE2_Media_Computer.sof

The SOF file represents the FPGA programming file for the Nios II system. If it is specified here, then the Monitor Program can be used to download this programming file onto the board. Otherwise, the system will need to be downloaded using some other method (for example, by using Quartus II).

Figure 3. Specification of the system.



Program Type: Assembly Program

Lets you specify a program written in assembly language.

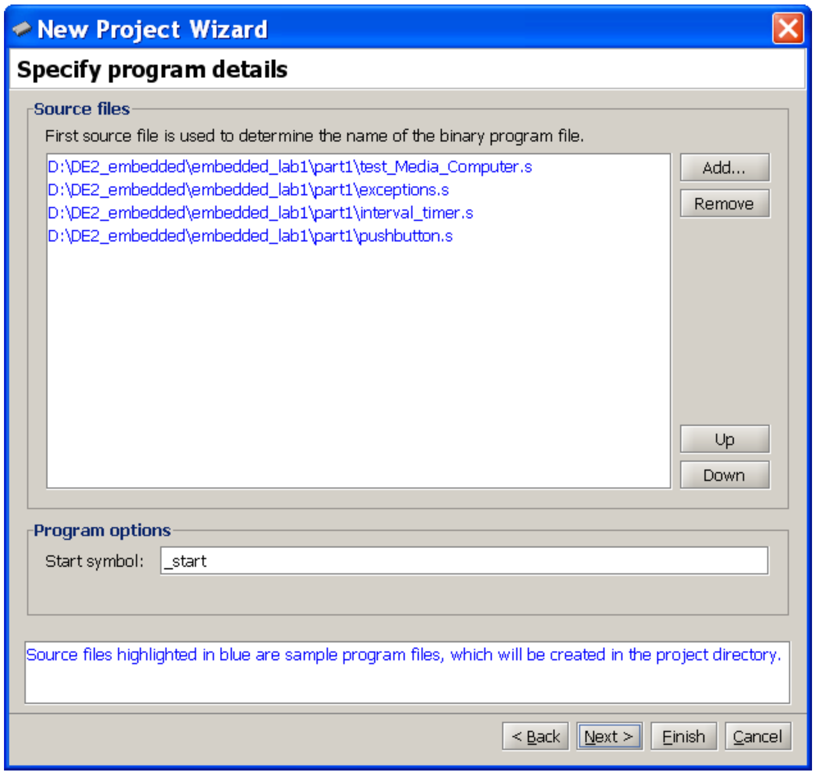
☑ Include a sample program with the project

Getting Started
JTAG UART
Interrupt Example
Test Media Computer

Exercises various features of the DE2 Media Computer, as a test and to provide an example.

4. connects the SW switches to red LEDs
5. handles pushbutton interrupts: pushing KEY1 speeds up the scrolling of text, pushing KEY2 slows it down, and pushing KEY3 stops the scrolling
6. can test the JP1, JP2 expansion ports, if a ribbon cable is installed between them
7. echos text received from the JTAG UART (such as text typed in the terminal window of the Monitor Program) to the serial port UART, and vice versa

Figure 4. Selection of an application program.



Specify program details

Source files

First source file is used to determine the name of the binary program file.

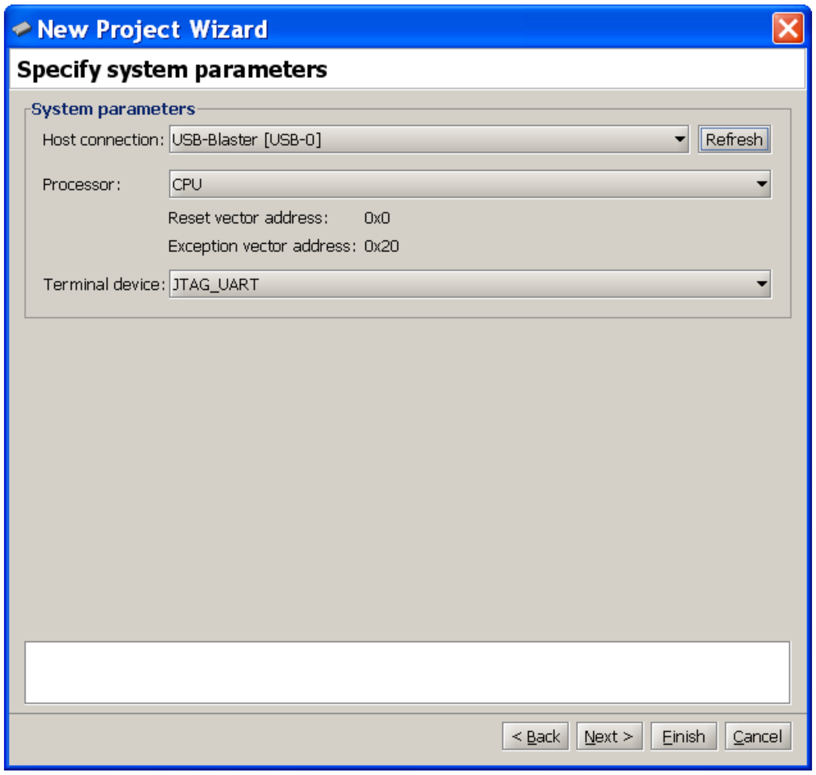
D:\DE2_embedded\embedded_lab1\part1\test_Media_Computer.s
D:\DE2_embedded\embedded_lab1\part1\exceptions.s
D:\DE2_embedded\embedded_lab1\part1\interval_timer.s
D:\DE2_embedded\embedded_lab1\part1\pushbutton.s

Program options

Start symbol: _start

Source files highlighted in blue are sample program files, which will be created in the project directory.

Figure 5. Source files used by the application program.



Specify system parameters

System parameters

Host connection: USB-Blaster [USB-0]

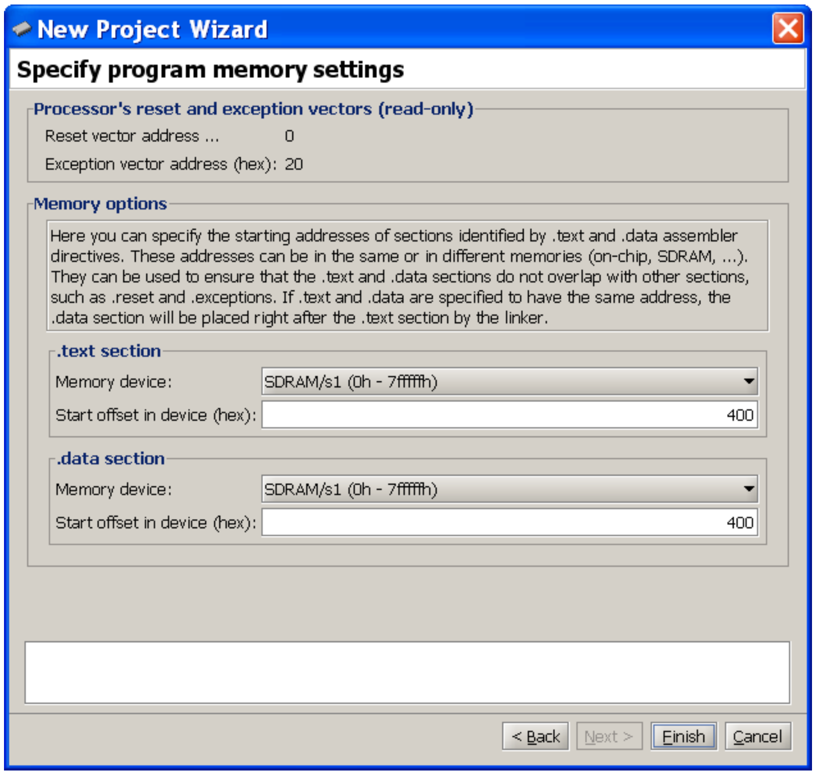
Processor: CPU

Reset vector address: 0x0

Exception vector address: 0x20

Terminal device: JTAG_UART

Figure 6. Specify the system parameters.


New Project Wizard

Specify program memory settings

Processor's reset and exception vectors (read-only)

Reset vector address ... 0

Exception vector address (hex): 20

Memory options

Here you can specify the starting addresses of sections identified by .text and .data assembler directives. These addresses can be in the same or in different memories (on-chip, SDRAM, ...). They can be used to ensure that the .text and .data sections do not overlap with other sections, such as .reset and .exceptions. If .text and .data are specified to have the same address, the .data section will be placed right after the .text section by the linker.

.text section

Memory device: SDRAM/s1 (0h - 7ffffffh)

Start offset in device (hex): 400

.data section

Memory device: SDRAM/s1 (0h - 7ffffffh)

Start offset in device (hex): 400

Figure 7. Specify the program memory settings.

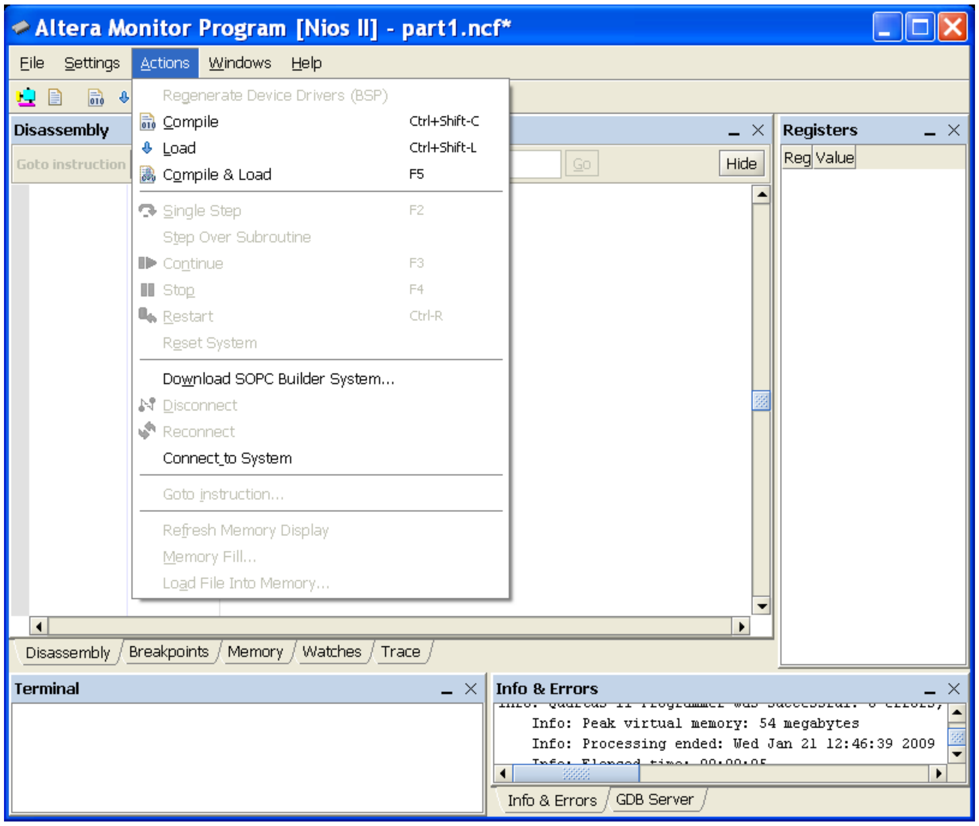
7. The window in Figure 7 displays the preselected components of the DE2 Media Computer. Observe that the SDRAM is selected as the memory device to be used. The start offset is 0x400 (hex 400), which means that the application program will be loaded in the memory locations that begin at address 400. Since this choice was made by the designer of the sample program, you cannot change the selection in Figure 7. Click Finish to complete the specification of the new project.

8. Since you specified a new project, a pop-up box will appear asking you if you want to download the system associated with this project onto the DE2 board. Make sure that the power to the DE2 board is turned on and click Yes. Watch the change in state of the blue LEDs on the DE2 board that correspond to LOAD and GOOD, which will blink as the circuit is being downloaded. A pop-up box will appear informing you that the circuit has been successfully downloaded - click OK. If the circuit is not successfully downloaded, make sure that the USB connection, through which the USB-Blaster communicates, is established and recognized by the host computer. (If there is a problem, a possible remedy may be to unplug the USB cable and then plug it back in.)

9. Having downloaded the DE2 Media Computer into the FPGA chip on the DE2 board, we can now load and run programs on this computer. In the main monitor window, shown in Figure 8, select Actions > Compile & Load to load the selected sample program into the FPGA chip. Figure 9 shows the monitor window after the sample program has been loaded.

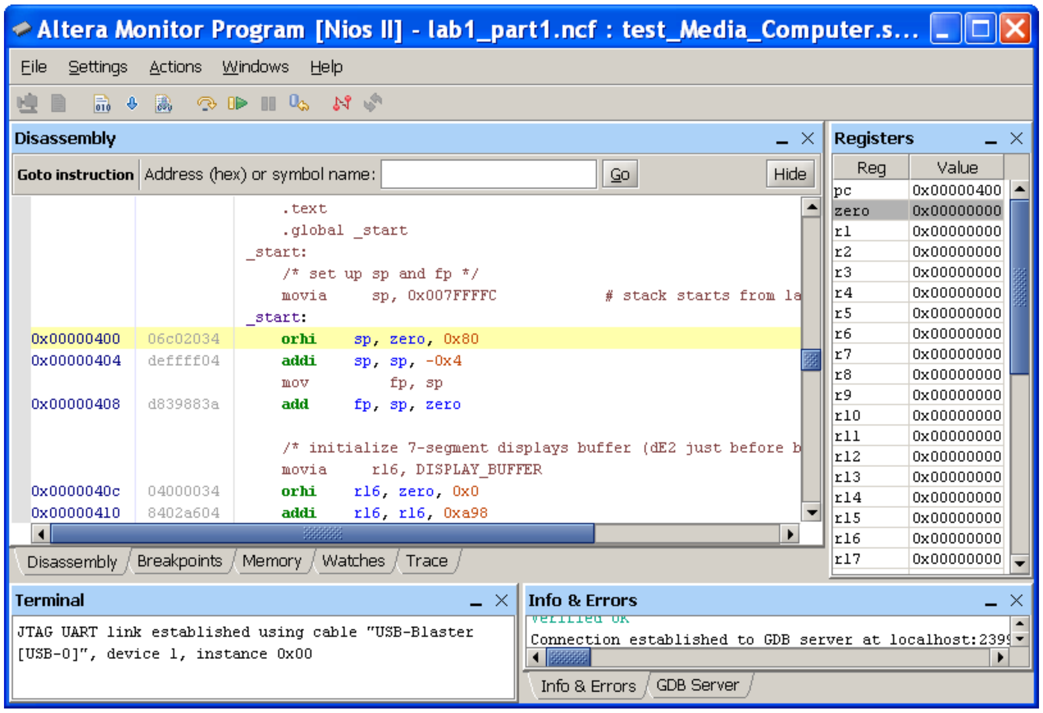
10. Run the program by selecting Actions > Continue or by clicking on the toolbar icon, and observe the test displayed on the LEDs and 7-segment displays. This test provides an indication that the DE2 board is functioning properly.

11. Stop the execution of the sample program by clicking on the icon, and disconnect from this session by clicking on the icon.



```
Info: Quartus II Programmer was successful. 0 errors,
   Info: Peak virtual memory: 54 megabytes
   Info: Processing ended: Wed Jan 21 12:46:39 2009
   Info: Elapsed time: 00:00:05
```

Figure 8. Specify an action in the monitor window.


Altera Monitor Program [Nios II] - lab1_part1.ncf : test_Media_Computer.s...
File Settings Actions Windows Help
Disassembly
Goto instruction
Address (hex) or symbol name:
Go
Hide
Registers
Disassembly Breakpoints Memory Watches Trace
Terminal
Info & Errors
Info & Errors GDB Server

```
                                .text
                                .global _start
                            _start:
                                /* set up sp and fp */
                                movia     sp, 0x007FFFFC          # stack starts from la
                            _start:
0x00000400     06c02034         orhi     sp, zero, 0x80
0x00000404     deffff04         addi     sp, sp, -0x4
                                mov       fp, sp
0x00000408     d839883a         add      fp, sp, zero

                                /* initialize 7-segment displays buffer (dE2 just before b
                                movia     r16, DISPLAY_BUFFER
0x0000040c     04000034         orhi     r16, zero, 0x0
0x00000410     8402a604         addi     r16, r16, 0xa98
```

Reg	Value
pc	0x00000400
zero	0x00000000
r1	0x00000000
r2	0x00000000
r3	0x00000000
r4	0x00000000
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00000000
r13	0x00000000
r14	0x00000000
r15	0x00000000
r16	0x00000000
r17	0x00000000

```
JTAG UART link established using cable "USB-Blaster
[USB-0]", device 1, instance 0x00
```

```
Verified OK
Connection established to GDB server at localhost:239
```

Figure 9. The monitor window showing the loaded sample program.

Part II

Now, we will explore some features of the Altera Monitor Program by using a simple application program written in the Nios II assembly language. Consider the program in Figure 10, which finds the largest number in a list of 32-bit integers that is stored in the memory. This program is available in the file *embedded_lab1_part2.s*.

```
/* Program that finds the largest number in a list of integers */
.equ  LIST, 0x800                 /* Starting address of the list */

.global  _start
_start:
    movia   r4, LIST              * r4 points to the start of the list */
    ldw     r5, 4(r4)             * r5 is a counter, initialize it with n */
    addi    r6, r4, 8             /* r6 points to the first number */
    ldw     r7, (r6)              /* r7 holds the largest number found so far */
LOOP:
    subi    r5, r5, 1             /* Decrement the counter */
    beq     r5, r0, DONE          /* Finished if r5 is equal to 0 */
    addi    r6, r6, 4             /* Increment the list pointer */
    ldw     r8, (r6)              /* Get the next number */
    bge     r7, r8, LOOP          /* Check if larger number found */
    add     r7, r8, r0            /* Update the largest number found */
    br      LOOP
DONE:
    stw     r7, (r4)              /* Store the largest number into RESULT */
STOP:
    br      STOP                  /* Remain here if done */

.org    0x800
RESULT:
.skip   4                         /* Space for the largest number found */
N:
.word  7                          /* Number of entries in the list */
NUMBERS:
.word  4, 5, 3, 6, 1, 8, 2        /* Numbers in the list */
.end
```

Figure 10. Assembly-language program that finds the largest number.

Note that some sample data is included in this program. The list starts at hex address 800, as specified by the **.org** assembler directive. The first word (4 bytes) is reserved for storing the result, which will be the largest number found. The next word specifies the number of entries in the list. The words that follow give the actual numbers in the list.

Make sure that you understand the program in Figure 10 and the meaning of each instruction in it. Note the extensive use of comments in the program. You should always include meaningful comments in programs that you will write!

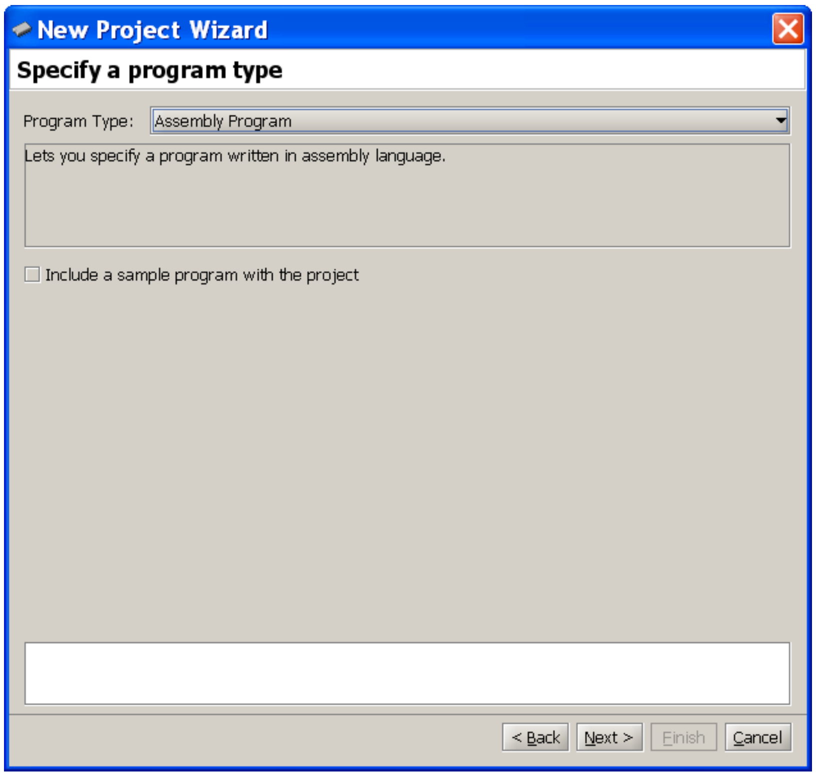
Perform the following:

1. Create a new directory; we have chosen the directory name *embedded_lab1_part2*. Copy the file *embedded_lab1_part2.s* into this directory.

2. Use the Altera Monitor Program to create a new project in this directory; we have chosen the project name *part2*. When you reach the window in Figure 4 choose Assembly Program but do not select a sample program, as shown in Figure 11. Click Next.

3. Upon reaching the window in Figure 5, you have to specify your program. Click Add and in the pop-up box that appears indicate the desired file name, *embedded_lab1_part2.s*, and its location. This should lead to the image in Figure 12. Click Next to get the window in Figure 6. Again click Next to get to the window in Figure 7. Make sure that the SDRAM is selected as the memory device. Note that the *Start offset in device* will be 0, because the program in Figure 10 does not indicate that it should be loaded at a location that is different from the default location 0. Click Finish.

4. Compile and load the program.


New Project Wizard

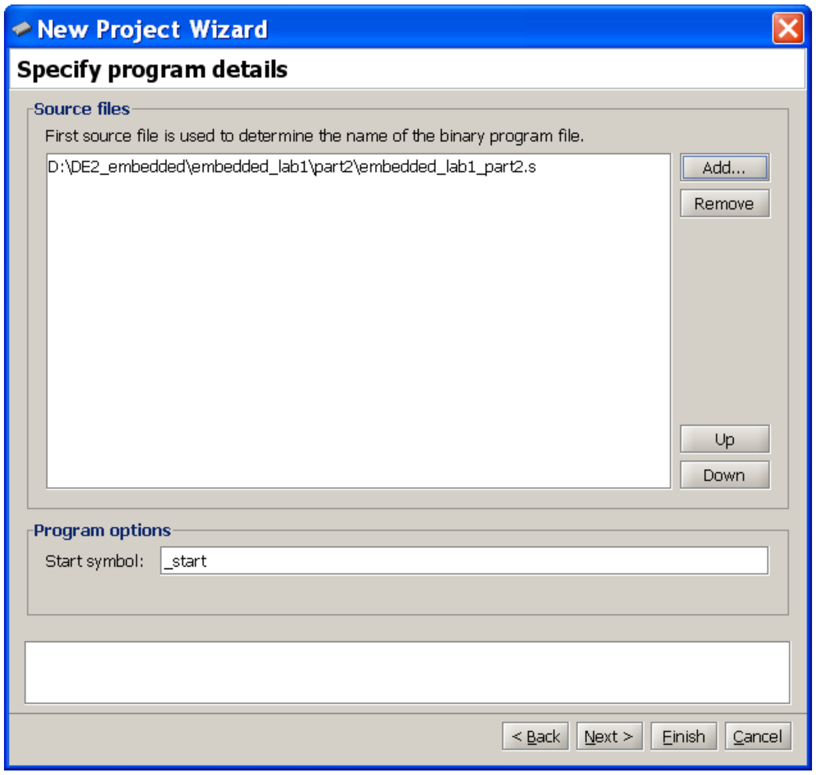
Specify a program type

Program Type: Assembly Program

Lets you specify a program written in assembly language.

☐ Include a sample program with the project

Figure 11. Select the assembly language program.



First source file is used to determine the name of the binary program file.

D:\DE2_embedded\embedded_lab1\part2\embedded_lab1_part2.s

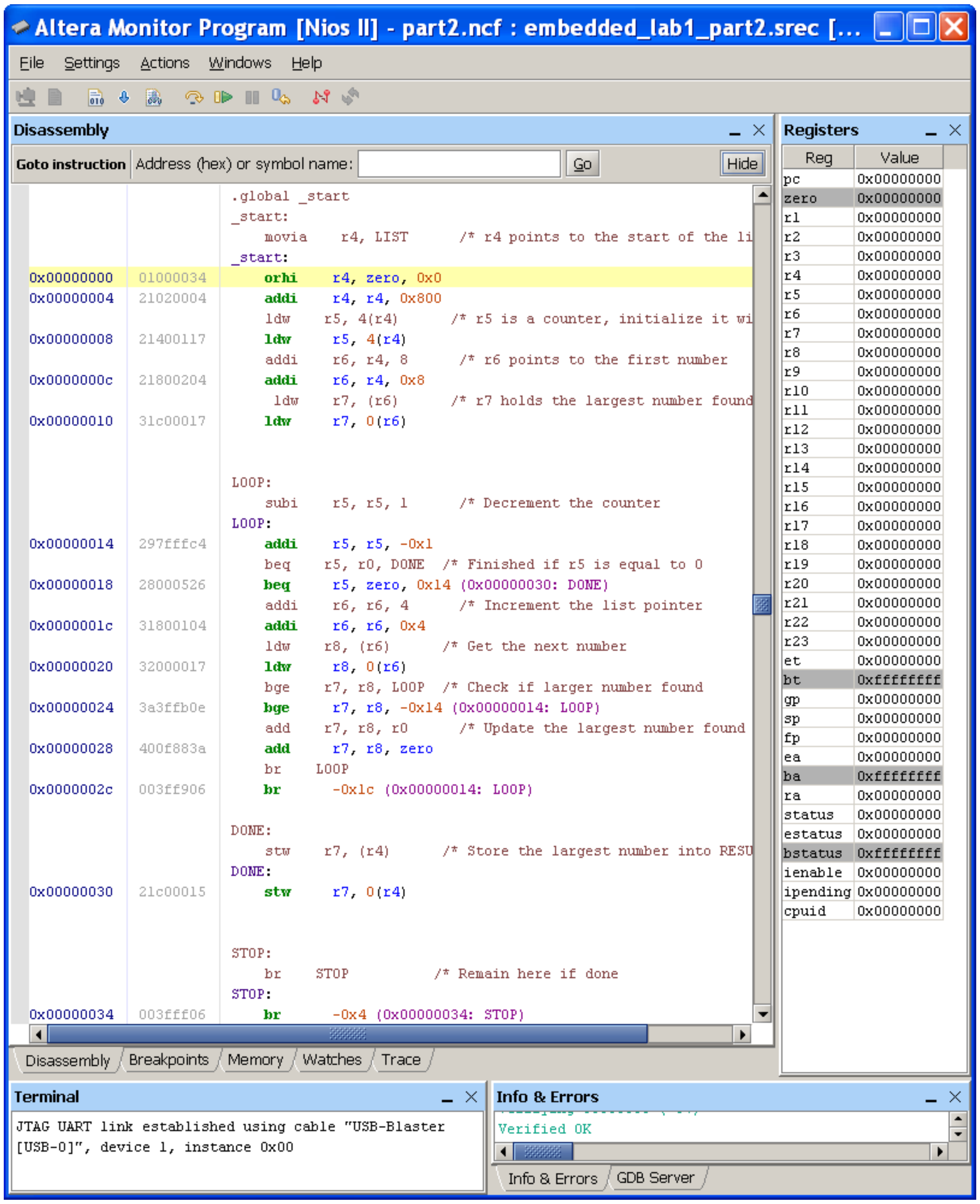
Start symbol: _start

Figure 12. Select your source program.

5. The Monitor Program will display the disassembled view of the code loaded in the memory, as indicated in Figure 13. Note that the pseudoinstruction **movia** in the original program has been replaced with two machine instructions, **orhi** and **addi**, which load the 32-bit address LIST into register *r4* in two 16-bit parts (because an immediate operand value is restricted to 16 bits). Examine the disassembled code to see the difference in comparison with the original source program. Make sure that you understand the meaning of each instruction. Observe also that your program was loaded into memory locations with the starting address 0. These addresses correspond to the SDRAM memory, which was selected when specifying the system parameters. The DE2 Media Computer has two more memories which are the *on-chip* memory (i.e. the memory on the FPGA chip) and the SRAM chip on the DE2 board. See the document *Media Computer System for Altera DE2 Board* for full information. This document can be accessed by clicking on the Documentation button in Figure 3.

6. Run the program. When the program is running, you will not be able to see any changes (such as the contents of registers or memory locations) in the monitor windows, because the monitor program cannot communicate with the processor system on the DE2 board. But, if you stop the program, the present state of these components will be displayed. Do so and observe that the program has stopped executing at the last Branch instruction which is loaded in the memory location 0x34. Note that the largest number found in the sample list is 8 as indicated by the contents of register *r7*. This value is also stored in the memory location 0x800, which can be seen by opening the Memory tab of the monitor window (in Figure 1.13).

7. Return to the beginning of the program by clicking on the icon . Now, single step through the program by clicking on the icon . Watch how the instructions change the data in the processor's registers.


Altera Monitor Program [Nios II] - part2.ncf : embedded_lab1_part2.srec [...
File Settings Actions Windows Help
Disassembly
Goto instruction Address (hex) or symbol name: Go Hide
Registers
Disassembly Breakpoints Memory Watches Trace
Terminal
Info & Errors
Info & Errors GDB Server

```
                        .global _start
                        _start:
                            movia   r4, LIST      /* r4 points to the start of the li
                        _start:
0x00000000  01000034        orhi    r4, zero, 0x0
0x00000004  21020004        addi    r4, r4, 0x800
                            ldw   r5, 4(r4)      /* r5 is a counter, initialize it wi
0x00000008  21400117        ldw     r5, 4(r4)
                            addi    r6, r4, 8     /* r6 points to the first number
0x0000000c  21800204        addi    r6, r4, 0x8
                             ldw    r7, (r6)     /* r7 holds the largest number found
0x00000010  31c00017        ldw     r7, 0(r6)


                        LOOP:
                            subi    r5, r5, 1     /* Decrement the counter
                        LOOP:
0x00000014  297fffc4        addi    r5, r5, -0x1
                            beq   r5, r0, DONE  /* Finished if r5 is equal to 0
0x00000018  28000526        beq     r5, zero, 0x14 (0x00000030: DONE)
                            addi    r6, r6, 4     /* Increment the list pointer
0x0000001c  31800104        addi    r6, r6, 0x4
                            ldw   r8, (r6)      /* Get the next number
0x00000020  32000017        ldw     r8, 0(r6)
                            bge   r7, r8, LOOP  /* Check if larger number found
0x00000024  3a3ffb0e        bge     r7, r8, -0x14 (0x00000014: LOOP)
                            add   r7, r8, r0    /* Update the largest number found
0x00000028  400f883a        add     r7, r8, zero
                            br    LOOP
0x0000002c  003ff906        br      -0x1c (0x00000014: LOOP)

                        DONE:
                            stw   r7, (r4)      /* Store the largest number into RESU
                        DONE:
0x00000030  21c00015        stw     r7, 0(r4)


                        STOP:
                            br    STOP          /* Remain here if done
                        STOP:
0x00000034  003fff06        br      -0x4 (0x00000034: STOP)
```

Reg	Value
pc	0x00000000
zero	0x00000000
r1	0x00000000
r2	0x00000000
r3	0x00000000
r4	0x00000000
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00000000
r13	0x00000000
r14	0x00000000
r15	0x00000000
r16	0x00000000
r17	0x00000000
r18	0x00000000
r19	0x00000000
r20	0x00000000
r21	0x00000000
r22	0x00000000
r23	0x00000000
et	0x00000000
bt	0xffffffff
gp	0x00000000
sp	0x00000000
fp	0x00000000
ea	0x00000000
ba	0xffffffff
ra	0x00000000
status	0x00000000
estatus	0x00000000
bstatus	0xffffffff
ienable	0x00000000
ipending	0x00000000
cpuid	0x00000000

```
JTAG UART link established using cable "USB-Blaster
[USB-0]", device 1, instance 0x00
```

```
Verified OK
```

Figure 13. The disassembled view of the program in Figure 10.

8. Set the Program Counter to 0. Note that this action has the same effect as clicking on the restart icon.

9. This time add a breakpoint at address 0x2C (by clicking on the gray bar to the left of this address), so that the program will automatically stop executing whenever the Branch instruction at this location is about to be executed. Run the program and observe the contents of register *r7* each time this breakpoint is reached.

10. Remove the breakpoint (by clicking on it). Then, set the Program Counter to 0x8, which will bypass the first two instructions which load the address LIST into register *r4*. Also, set the value in register *r4* to 0x804. Run the program by clicking on the icon. What will be the result of this execution?

Part III

Implement the task in Part II by rewriting the program in Figure 10 in the form of a subroutine. The subroutine, LARGE, has to find the largest number in a list. The calling program should pass the number of entries and the address of the first number in the list as parameters to the subroutine via registers *r5* and *r6*. The subroutine should pass the value of the largest number to the calling program via register *r7*.

Create a new directory and a new Monitor Program project to compile and download your program. The Monitor Program can have only one project per directory! Run your program to verify its correctness.

Part IV

Modify your program from Part III so that the parameters and the result are passed between the calling program and the subroutine via the processor stack. Initialize the stack pointer, *sp*, to an appropriate address value.

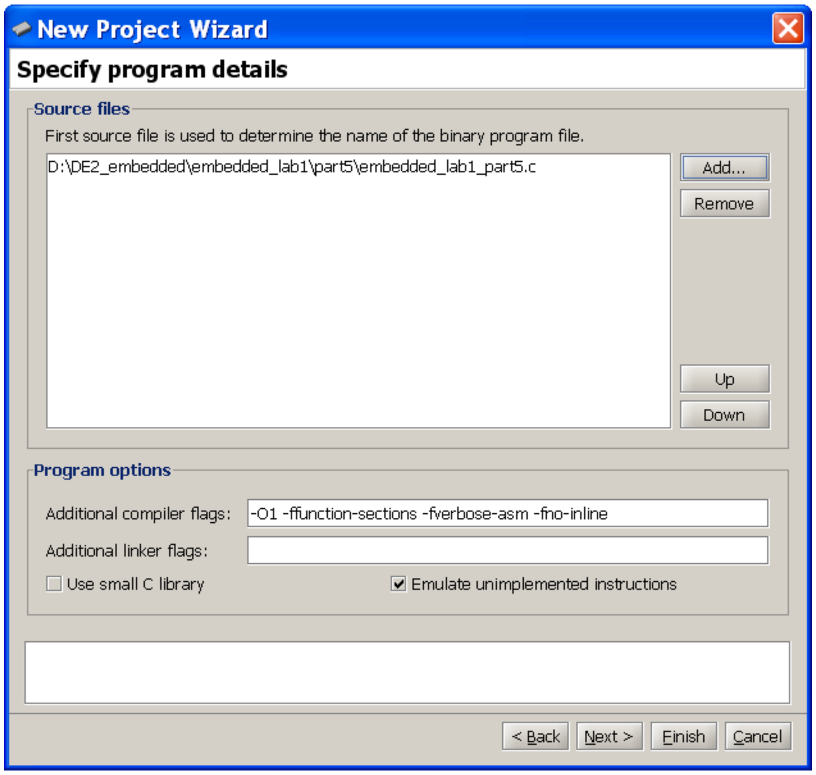
Run your program to verify its correctness.

Part V

Write a C-language program to implement the task in Part II, in a file called *embedded_lab1_part5.c*. Use the **printf** statement to display the result in the Monitor Program's terminal window.

Create a new directory and copy your file into it. Then, create a new Monitor Program project. Choose C Program in the window in Figure 11. In the window in Figure 12, include your file in the project. Use the default Program Options as indicated in Figure 14.

Compile, download and run your program. Examine the disassembled code and compare it to the code produced in Part II.



First source file is used to determine the name of the binary program file.

D:\DE2_embedded\embedded_lab1\part5\embedded_lab1_part5.c

Additional compiler flags: -O1 -ffunction-sections -fverbose-asm -fno-inline

Additional linker flags:

☐ Use small C library

☑ Emulate unimplemented instructions

Figure 14. Select the C program.

Part VI

Using the **printf** statement results in a fairly large number of assembly-language instructions, because a standard library routine is used. Augment your program to display the result in a specific memory location, e.g. 0x6000, instead of using the **printf** statement. Compile and run this program, and observe the difference.

Preparation

Your preparation should include the following:

1. Read the tutorials *Introduction to the Altera Nios II Soft Processor* and *Altera Monitor Program*.
2. Write the assembly-language programs for Parts III and IV.
3. Write the C-language programs for Parts V and VI.

Laboratory Exercise 2

Use of Logic Instructions

Logic instructions are needed in many embedded applications. They are useful for manipulation of bit strings and for dealing with data at the bit level where only a few bits may be of special interest. They are essential in dealing with input/output tasks. In this exercise we will consider some typical uses. We will use the DE2 Media Computer with your application programs being loaded in the SDRAM memory.

Since the tasks used in the exercise are of general interest, the students may be able to find various solutions on the Internet. However, in order to maximize the knowledge attained by performing the exercise, we strongly recommend that students develop and test their own solutions.

Part I

Write a Nios II assembly-language program to determine the number of bits that have the value 1 in a 32-bit word. Let the word to be tested be stored in memory location TEST_DATA. Save the result in memory location RESULT.

Create a new project and run your program to demonstrate its correctness.

Part II

Write a Nios II program to reverse the order of bits in a 32-bit word. Let the test word be in memory location TEST_DATA, and place the result in memory location RESULT. Write your program to perform the required task in the minimum amount of time, as measured in terms of the number of clock cycles needed to execute the program.

Create a new project and run your program to demonstrate its correctness.

Part III

In embedded applications it is often desirable to minimize the amount of memory space needed by an application program. Repeat the task in Part II by writing a program that requires as little memory space as possible.

Part IV

Assume that an input device provides an 18-bit signed number, and that this number is stored in memory word INDATA. Assume also that the most-significant 14 bits of this word are set to 0 when the number is read from the input device.

If the stored number is to be used in arithmetic operations, it has to be sign-extended. Write a Nios II program to perform this task. Place the result in memory location RESULT.

Create a new project and run your program to demonstrate its correctness.

Part V

Write a C-language program to determine the number of bits that have the value 1 in a 32-bit word. Use the **scanf** statement to input the pattern to be tested via the terminal window in the Altera Monitor Program. Note that you have to click on the terminal window to activate it for input purposes. Right-click on the terminal window and select Echo input in the drop-down menu, which will display the input characters as you type them on the keyboard. Use the **printf** statement to display the computed result.

Try a number of different test patterns to verify the correctness of your program.

Part VI

Write a C program to reverse the order of bits in a 32-bit word. Use the same input/output approach as in Part V.

Part VII

Write a C program to perform the task in Part IV. Use the same input/output approach as in Part V.

Preparation

As a part of your preparation you should do the following:

1. Write the assembly-language programs for Parts I to IV.
2. Write the C-language programs for Parts V to VII.

Laboratory Exercise 3

Input/Output in a Nios II System

The purpose of this exercise is to investigate the use of devices that provide input and output capabilities for a processor. There are two basic techniques for dealing with I/O devices: program-controlled polling and interrupt-driven approaches. In this exercise, we will use the polling approach. In Laboratory Exercise 4, we will use the interrupt-driven approach. We will use both the Nios II assembly language and the C programming language to implement the necessary software.

As an example of I/O hardware, we will make use of parallel port interfaces in the DE2 Media Computer system implemented on an Altera DE2 board. The background knowledge needed to do this exercise can be acquired from the tutorials: *Introduction to the Altera Nios II Soft Processor* and *Media Computer System for Altera DE2 Board*.

The parallel port interfaces in the DE2 Media Computer were generated by using Altera's SOPC Builder software (which we will use in Laboratory Exercise 5). A parallel port provides for data transfer in either input or output direction. In the SOPC Builder, a parallel port is implemented in the form of a *PIO (Parallel Input/Output)* component. The transfer of data is done in parallel and it may involve from 1 to 32 bits. The number of bits, n, and the type of transfer are specified by the user through the SOPC Builder (at the time a Nios II based system is being designed). The PIO interface can contain the four registers shown in Figure 1.

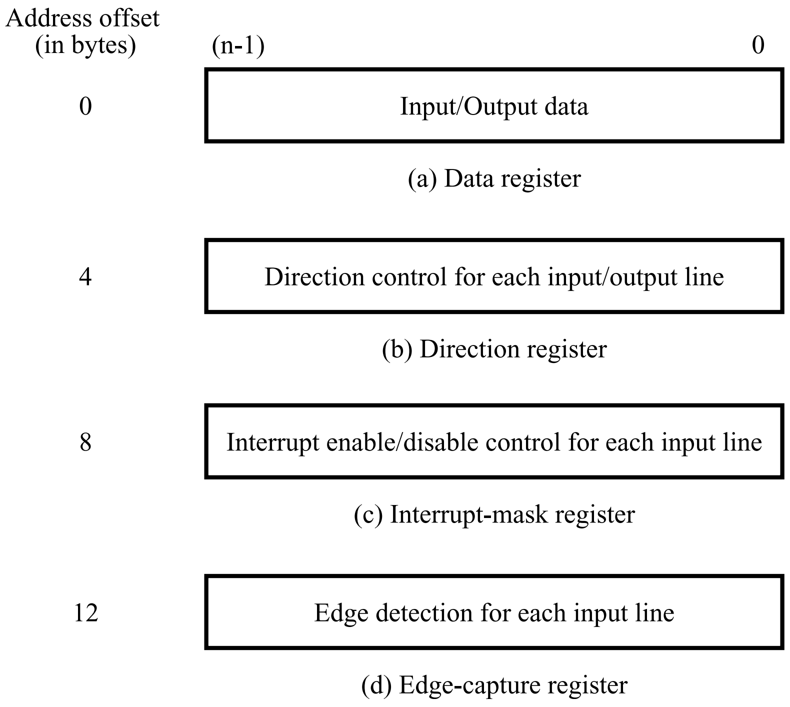

Address offset
(in bytes)
(n-1)
0
0
Input/Output data
(a) Data register
4
Direction control for each input/output line
(b) Direction register
8
Interrupt enable/disable control for each input line
(c) Interrupt-mask register
12
Edge detection for each input line
(d) Edge-capture register

Figure 1. 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 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. The *Interrupt-mask* and *Edge-capture* registers must be included if interrupt-driven input/output is used.

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 (at the time it is implemented 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) from this base address.

The DE2 Media Computer includes several PIOs that are configured for various uses. The details of these PIOs are described in the *Media Computer System for Altera DE2 Board* tutorial.

The application task in this exercise consists of adding together a set of unsigned 8-bit numbers that are entered via the toggle switches on the DE2 board. The resulting sum is displayed on the LEDs and 7-segment displays.

Part I

Use 8 toggle switches, SW_{7-0}, as inputs for entering numbers. Use the green lights, $LEDG_{7-0}$, to display the number defined by the toggle switches. Use the 16 red lights, $LEDR_{15-0}$, to display the accumulated sum. All of these components are connected via parallel ports in the DE2 Media Computer.

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. Create a new directory, *lab3_part1*. Put your program, *lab3_part1.s*, into this directory.
3. Use the *Altera Monitor Program* to create a new project, *part1*, in this directory. Select your program and download the DE2 Media Computer into the FPGA device on the DE2 board. Choose SDRAM as the memory that your program will use. Assemble and download your program.
4. Single-step through the program and verify its correctness by inputting several numbers. Note that single-stepping through the program will allow you to change the input numbers without reading the same number multiple times.

Part II

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, KEY_1, 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 parallel port in the DE2 Media Computer. To provide a status flag, we will use the Pushbutton Parallel Port, in which bit position b_1 is connected to KEY_1.

Perform the following steps:

1. Create a new project in a new directory for this part.
2. Modify your application program from Part I 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.
3. 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 KEY_1 is pressed.

Part III

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. Note that the 7-segment displays are also connected to a parallel port in the DE2 Media Computer.

Part IV

Repeat Part I by writing the necessary program in the C language. After downloading the program into the FPGA device on the DE2 board, place a breakpoint at the start of the loop that reads a new number and adds it to the sum. Then, run the program repeatedly through this loop to verify that it works properly.

Part V

Repeat Part II by writing the necessary program in the C language.

Part VI

Repeat Part III by writing the necessary program in the C language.

Preparation

Your preparation should include the programs for Parts I to VI.

Laboratory Exercise 4

Interrupts

Laboratory Exercise 3 illustrated how input/output transfers can be performed using the program-controlled polling approach. Now, we will consider the interrupt-driven approach to perform the same tasks. We will use both the Nios II assembly language and the C programming language to implement the necessary software. As an example of I/O hardware, we will again make use of parallel-port interfaces in the DE2 Media Computer system implemented on an Altera DE2 board.

The application task in this exercise consists of adding together a set of unsigned 8-bit numbers that are entered via the slider switches on the DE2 board. The resulting sum is displayed on the LEDs and 7-segment displays. Use 8 slider switches, SW_{7-0}, as inputs for entering numbers. Use the green lights, $LEDG_{7-0}$, to display the number defined by the slider switches. Use the 16 red lights, $LEDR_{15-0}$, to display the accumulated sum as a binary number. Display this sum also on the 7-segment displays *HEX3-HEX0*. All of these components are connected via parallel ports in the DE2 Media Computer. A new number is to be added to the current sum whenever an interrupt request is raised by pressing the pushbutton switch KEY_1.

Parallel Ports

The parallel port interfaces in the DE2 Media Computer were generated by using Altera's SOPC Builder software (which we will use in Laboratory Exercise 5). A parallel port provides for data transfer in either input or output direction. In the SOPC Builder, a parallel port is implemented in the form of a *PIO (Parallel Input/Output)* component. The transfer of data is done in parallel and it may involve from 1 to 32 bits. The number of bits, n, and the type of transfer are specified by the user through the SOPC Builder (at the time a Nios II based system is being designed). The PIO interface can contain the four registers shown in Figure 1.

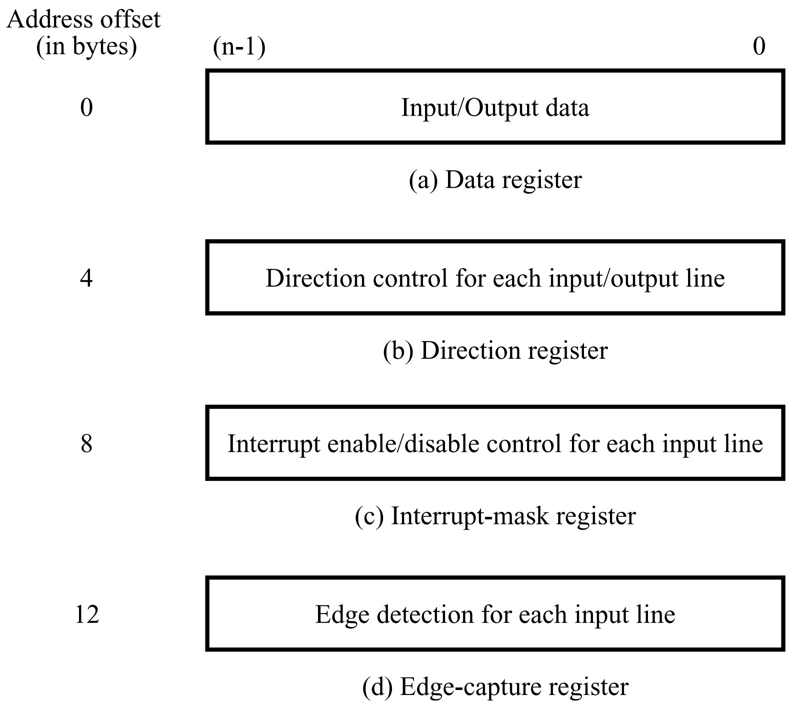

Address offset (in bytes)
(n-1)
0
0
Input/Output data
(a) Data register
4
Direction control for each input/output line
(b) Direction register
8
Interrupt enable/disable control for each input line
(c) Interrupt-mask register
12
Edge detection for each input line
(d) Edge-capture register

Figure 1. 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 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. The *Interrupt-mask* and *Edge-capture* registers must be included if interrupt-driven input/output is used.

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 (at the time it is implemented 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) from this base address.

The DE2 Media Computer includes several PIOs that are configured for various uses. The details of these PIOs are described in the *Media Computer System for Altera DE2 Board* tutorial.

Interrupts in the DE2 Media Computer

In the DE2 Media Computer, memory address 0x20 is the interrupt location. This means that when an external interrupt request is received, the Nios II processor will automatically execute the instruction stored at memory address 0x20. The processor performs this action also if an internal exception occurs, such as dividing by zero or encountering a software **trap** instruction.

One of the I/O peripherals that can raise interrupts is the *Pushbutton-switch Parallel Port*. It raises an interrupt request when a pushbutton key is pressed, if the corresponding bit in its *Interrupt-mask* register is set to 1. It also records the occurrence of a change in the signal generated by the pushbutton by setting to 1 the corresponding bit in the *Edge-capture* register.

Note that it is necessary to enable interrupts in three diferrent places: the I/O device interface (*Interrupt-mask* register in a PIO), the control register *ctl3* (*ienable*), and the control register *ctl0* (*status*).

Each I/O peripheral is assigned an IRQ (Interrupt Request) number, which is used to identify the source of an interrupt request. For the Pushbutton Port, the IRQ number is 1, which means that bit b_1 in the Nios II control register 4 will be set to 1 whenever some pushbutton key is pressed and the corresponding interrupts are enabled. The control register 4 can be accessed in an assembly language program as either *ctl4* or *ipending*.

A detailed explanation of Nios II interrupts is given in the tutorials: *Introduction to the Altera Nios II Soft Processor* and *Media Computer System for Altera DE2 Board*.

Reset in the DE2 Media Computer

Pushbutton switch KEY_0 provides the reset capability in the DE2 Media Computer. When this key is pressed, the Nios II processor is forced to execute the instruction stored at memory address 0.

Part I

In this part, you have to write a program that uses interrupts to read the contents of the slider switches, display the corresponding value on the green LEDs, and add this number to a sum that is being accumulated. Display the sum as a binary number on the red LEDs and as a hexadecimal number on the 7-segment displays *HEX3-HEX0*. A new number must be added each time the key KEY_1 is pressed, which has to cause an interrupt request.

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

1. Write a Nios II assembly-language program that performs the desired task.
2. Create a new directory, *lab4_part1*. Put your program, *lab4_part1.s*, into this directory.
3. Use the *Altera Monitor Program* to create a new project, *part1*, in this directory. Select your program and download the DE2 Media Computer into the FPGA device on the DE2 board. Choose SDRAM as the memory that your program will use. Assemble and download your program.
4. Run your program to demonstrate that it works properly.

Part II

Augment your program to display the sum as a decimal number on the 7-segment displays *HEX3-HEX0*, and verify its correctness.

Interrupts in a C-language Program

When using interrupts in a C-language program, it is necessary to deal with the specific requirements of the Nios II processor and DE2 Media Computer interrupt mechanism, which necessitates using some assembly-language instructions in the C code. Following is a brief explanation of what needs to be done when the Altera Monitor Program is used to run a C-language program.

Accessing the Nios II Control Registers

Nios II processor control registers can be accessed by using the **asm** type statements. For example, the contents of control register 4 can be copied into the general-purpose register **r8** by writing either

```
asm ("rdctl  r8, ctl4");
```

or

```
asm ("rdctl  r8, ipending");
```

To make it easier to incorporate such instructions into a program, there exists a set of macros in a file *nios2_ctrl_reg_macros.h* which is one of the design files provided with this laboratory exercise. Then, the above action can be specified with the statement

```
NIOS2_READ_IPENDING(8);
```

Using Attributes to Specify the Required Locations for Compiled C-Code

Pressing the reset key, *KEY*$_0$, forces the instruction at address 0 to be executed. This has to cause a branch to the beggining of the actual program, which can be achieved with the code:

```
void  the_reset(void)__attribute__((section(".reset")));
void  the_reset(void)
{
   asm  ("movia  r2, _start");
   asm  ("jmp  r2");
}
```

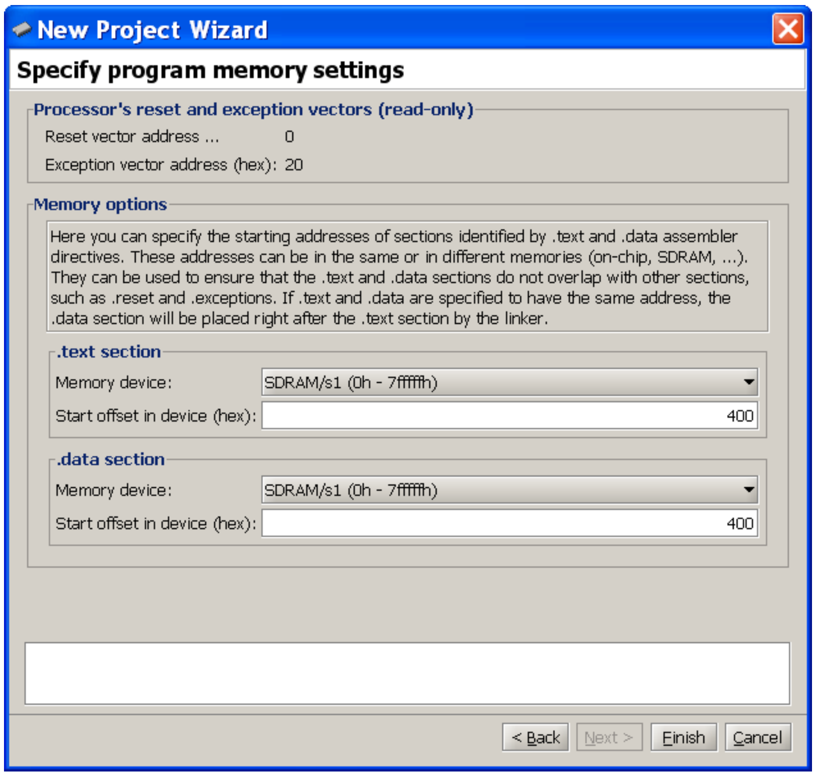
An external interrupt request, or an internal exception, forces the instruction at address 0x20 to be executed. The instructions that deal with interrupts and internal exceptions must start at this address, which can be accomplished by preceding the relevant C-code with the statements:

```
void the_exception(void)__attribute__((section(".exceptions")));
void the_exception(void)
```

The compiler associates the section attributes *.reset* and *.exceptions* with memory addresses 0x0 and 0x20, respectively, by examining the design parameters of the DE2 Media Computer.

Specifying the Address of the Main Program

The default address into which the main program is loaded is zero. But, the reset and interrupt-handler code has to be loaded into the memory starting at addresses 0x0 and 0x20, respectively. Thus, it is necessary to place the main-program code into higher-address locations to ensure that this code will not overlap with the interrupt-handler code. As shown in Figure 2, we have used the Altera Monitor Program to specify that the main-program code should start at address 0x400.



Specify program memory settings

Processor's reset and exception vectors (read-only)

Reset vector address ... 0

Exception vector address (hex): 20

Memory options

Here you can specify the starting addresses of sections identified by .text and .data assembler directives. These addresses can be in the same or in different memories (on-chip, SDRAM, ...). They can be used to ensure that the .text and .data sections do not overlap with other sections, such as .reset and .exceptions. If .text and .data are specified to have the same address, the .data section will be placed right after the .text section by the linker.

.text section

Memory device: SDRAM/s1 (0h - 7ffffffh)

Start offset in device (hex): 400

.data section

Memory device: SDRAM/s1 (0h - 7ffffffh)

Start offset in device (hex): 400

Figure 2. Specifying the memory location of the main-program.

Part III

Repeat Part I by writing the necessary program in the C language. Figure 3 gives a skeleton program that may be used for this purpose. You have to fill in the necessary details.

```
#include "nios2_ctrl_reg_macros.h"

   /* Define here the addresses of switches, lights and 7-segment displays. */

/* Code needed to implement the reset funcionality */
void  the_reset(void)__attribute__((section(".reset")));
void  the_reset(void)
{
   asm  ("movia  r2, _start");
   asm  ("jmp  r2");
}

/* Code needed to deal with exceptions */
void  the_exception(void)__attribute__((section(".exceptions")));
void  the_exception(void)
{
   asm  ("subi  sp, sp, 4");              /* Save the contents of  */
   asm  ("stw  et, (sp)");                /*   the et register.     */
   asm  ("rdctl  et, ipending");          /* If external interrupt, */
   asm  ("beq  et, r0, SKIP_EA_DEC");     /*   then decrement et   */
   asm  ("subi  ea, ea, 4");              /*   by 4.          */
   asm  ("SKIP_EA_DEC:");

   /* Insert here the code needed to save all registers except r0, et and sp. */

   asm  ("call  INTERRUPT_HANDLER");

   /* Insert here the code to restore all registers except r0 and sp. */

   asm  ("eret");                         /* Return from exception. */
}

void  INTERRUPT_HANDLER()
{
   /* Insert here the INTERRUPT_HANDLER code, which has to check if an interrupt */
   /*   from KEY1 has occurred and in response update the displayed sum.    */
}

main()
{
   /* Insert here the code for the main program, which has to enable   */
   /*   the desired interrupts. It also has to display the current sum.     */
}
```

Figure 3. Skeleton program for Part III.

Part IV

Repeat Part II by writing the necessary program in the C language.

Preparation

Your preparation should include the programs for Parts I to IV.

Laboratory Exercise 5

Implementation of an Embedded System

The purpose of this exercise is to learn how to create an embedded system and implement it in an FPGA device. The system will consist of an Altera Nios II processor and input/output interfaces that connect to switches and displays on an Altera DE-series board (intended for the boards listed in Table 1). 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 application programs. 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.

In this exercise, we will build a system that has some of the I/O capability of the DE-series Media Computer, sufficient to perform the I/O tasks performed in Laboratory Exercises 3 and 4. The desired system will include the parallel input/output interfaces (PIOs). Recall from Laboratory Exercises 3 and 4 that the PIO interface is a component that can be generated by using the SOPC Builder. It 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 (at the time a Nios II based system is being designed). The PIO interface can contain the four registers shown in Figure 1.

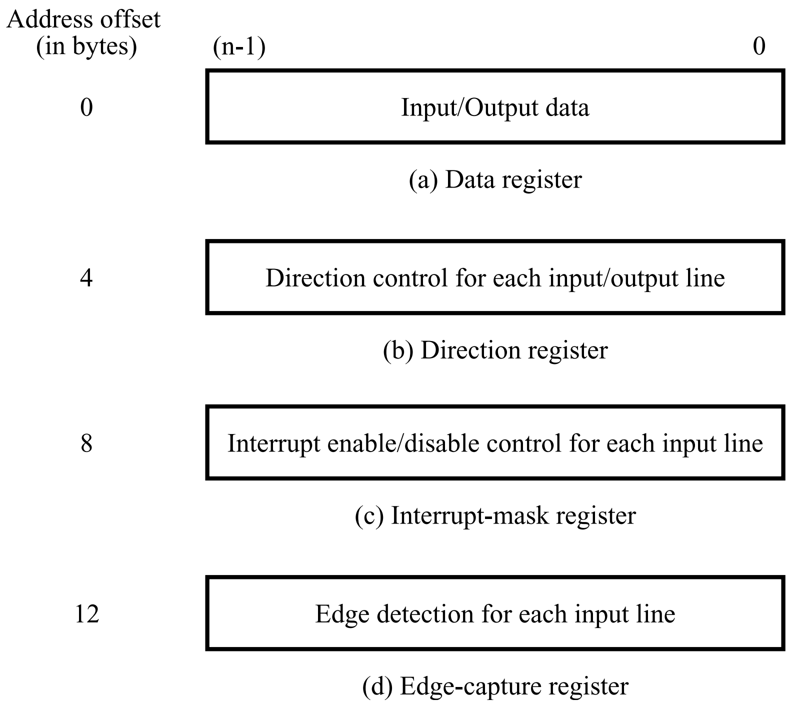

Address offset (in bytes)
(n-1)
0
0
Input/Output data
(a) Data register
4
Direction control for each input/output line
(b) Direction register
8
Interrupt enable/disable control for each input line
(c) Interrupt-mask register
12
Edge detection for each input line
(d) Edge-capture register

Figure 1. 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 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. The *Interrupt-mask* and *Edge-capture* registers are included if interrupt-driven input/output is used.

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 (at the time it is implemented 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) from this base address.

As in Labs 3 and 4, the application task in this exercise consists of adding together a set of signed 8-bit numbers that are entered via the slider switches on the DE-series board. The resulting sum is displayed on the LEDs and 7-segment displays. The exercise makes use of both polling and interrupt I/O schemes.

Part I

In this part we will use the SOPC Builder to design a Nios II based system that can be implemented in the FPGA on the DE-series board. We will use switches and LEDs on the board as input and output devices. Use 8 slider switches, SW_{7-0}, as inputs for entering numbers. Use the green lights, *LEDG*, to display the number selected by the switches. Use the red lights, *LEDR*, to display the accumulated sum as a binary number. Use the 7-segment displays, *HEX3-HEX0*, to display the sum as a hexadecimal number. A Nios II system which includes five PIO interfaces is the hardware needed for our task. One PIO circuit, connected to the slider switches, will provide the input data that can be read by the processor. Three PIO circuits, connected to the LEDs and HEX displays, will serve as the output interfaces to display the input number and the accumulated sum.

To provide a control signal for use in both polling and interrupt schemes, we will include a one-bit PIO circuit that will provide the functionality of a status flag and an ability to raise interrupt requests.

Realize the required hardware as follows:

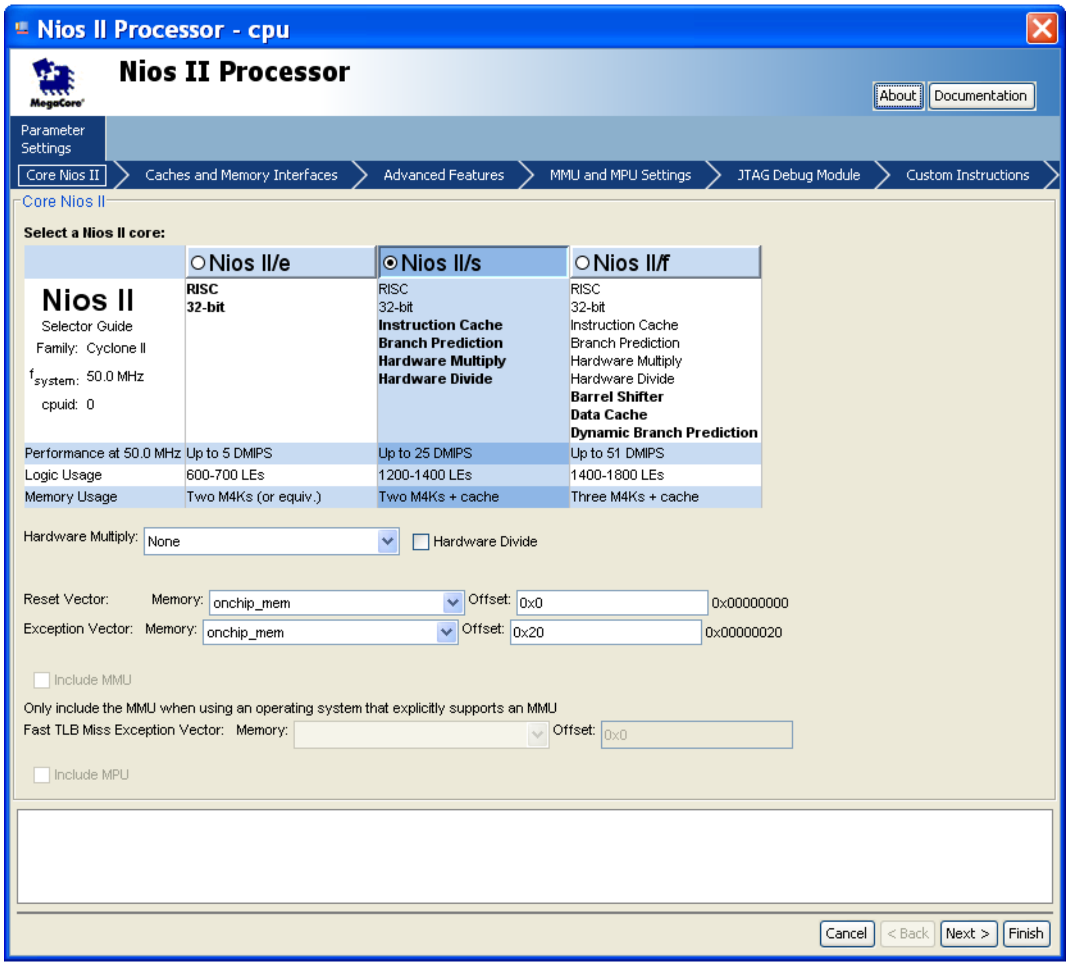
1. Create a Quartus II project. Select the FPGA Device for your DE-series board. Refer to Table 1 for a list of devices on the DE-series boards that can be used in this exercise.

Board	Device Name
DE1	Cyclone II EP2C20F484C7
DE2	Cyclone II EP2C35F672C6
DE2-70	Cyclone II EP2C70F896C6
DE2-115	Cyclone IVE EP4CE115F29C7

Table 1: DE-series FPGA device names

2. Use the SOPC Builder to generate the desired circuit, called *nios_system*, which comprises:
    - On-chip memory - RAM mode and 32 Kbytes in size (leave all other options at their default settings)
    - Nios II/s processor with JTAG Debug Module Level 1
        - Do **not** choose the Hardware Multiply and Hardware Divide options

- Choose on-chip memory as the location for Reset and Exception vectors, as indicated in Figure 2
- Leave all other options for the processor at their default settings



Core Nios II

Select a Nios II core:

	○ Nios II/e	◉ Nios II/s	○ Nios II/f
Nios II Selector Guide Family: Cyclone II f_{system}: 50.0 MHz cpuid: 0	**RISC** **32-bit**	RISC 32-bit **Instruction Cache** **Branch Prediction** **Hardware Multiply** **Hardware Divide**	RISC 32-bit Instruction Cache Branch Prediction Hardware Multiply Hardware Divide **Barrel Shifter** **Data Cache** **Dynamic Branch Prediction**
Performance at 50.0 MHz	Up to 5 DMIPS	Up to 25 DMIPS	Up to 51 DMIPS
Logic Usage	600-700 LEs	1200-1400 LEs	1400-1800 LEs
Memory Usage	Two M4Ks (or equiv.)	Two M4Ks + cache	Three M4Ks + cache

Hardware Multiply: None ☐ Hardware Divide

Reset Vector: Memory: onchip_mem Offset: 0x0 0x00000000

Exception Vector: Memory: onchip_mem Offset: 0x20 0x00000020

☐ Include MMU

Only include the MMU when using an operating system that explicitly supports an MMU

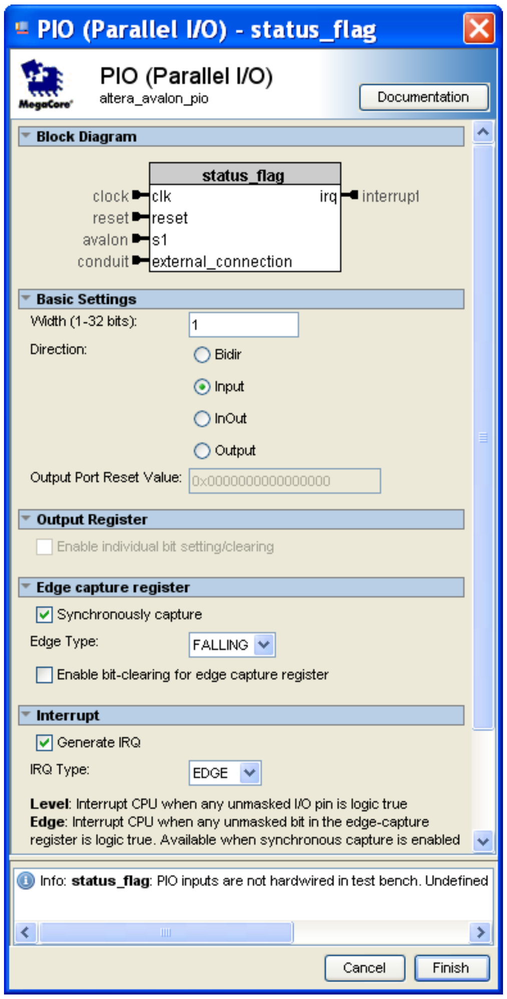
Fast TLB Miss Exception Vector: Memory: Offset: 0x0

☐ Include MPU

Figure 2. The SOPC Builder specification of the Nios II processor.

- An 8-bit PIO input circuit, which will be connected to slider switches (The PIO components are found by selecting Peripherals > Microcontroller Peripherals > PIO.)
- An 8-bit PIO output circuit, which will be connected to green LEDs
- A 16-bit PIO output circuit, which will be connected to red LEDs
- A 32-bit PIO output circuit, which will be connected to HEX displays
- A one-bit PIO circuit that will serve as a status flag, which will be connected to the pushbutton key KEY_1. Configure it to be an input port that is one-bit wide. Also, select the following:
  - Synchronously capture feature activated by the Falling edge for the *Edge capture* register.
  - Generate IRQ interrupt on Edge

  as indicated in Figure 3.



Basic Settings

Width (1-32 bits): 1

Direction: Bidir [✓ Input] InOut Output

Output Port Reset Value: 0x0000000000000000

Output Register

☐ Enable individual bit setting/clearing

Edge capture register

☑ Synchronously capture

Edge Type: FALLING

☐ Enable bit-clearing for edge capture register

Interrupt

☑ Generate IRQ

IRQ Type: EDGE

Level: Interrupt CPU when any unmasked I/O pin is logic true
Edge: Interrupt CPU when any unmasked bit in the edge-capture register is logic true. Available when synchronous capture is enabled

Info: **status_flag**: PIO inputs are not hardwired in test bench. Undefined

Cancel Finish

Figure 3. Specification for the status-flag PIO.

3. The SOPC Builder will automatically assign the names such as *pio*, *pio_1*, *pio_2*, ... to these PIO components. 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*, *red_LEDs*, *hex_displays* and *status_flag*.

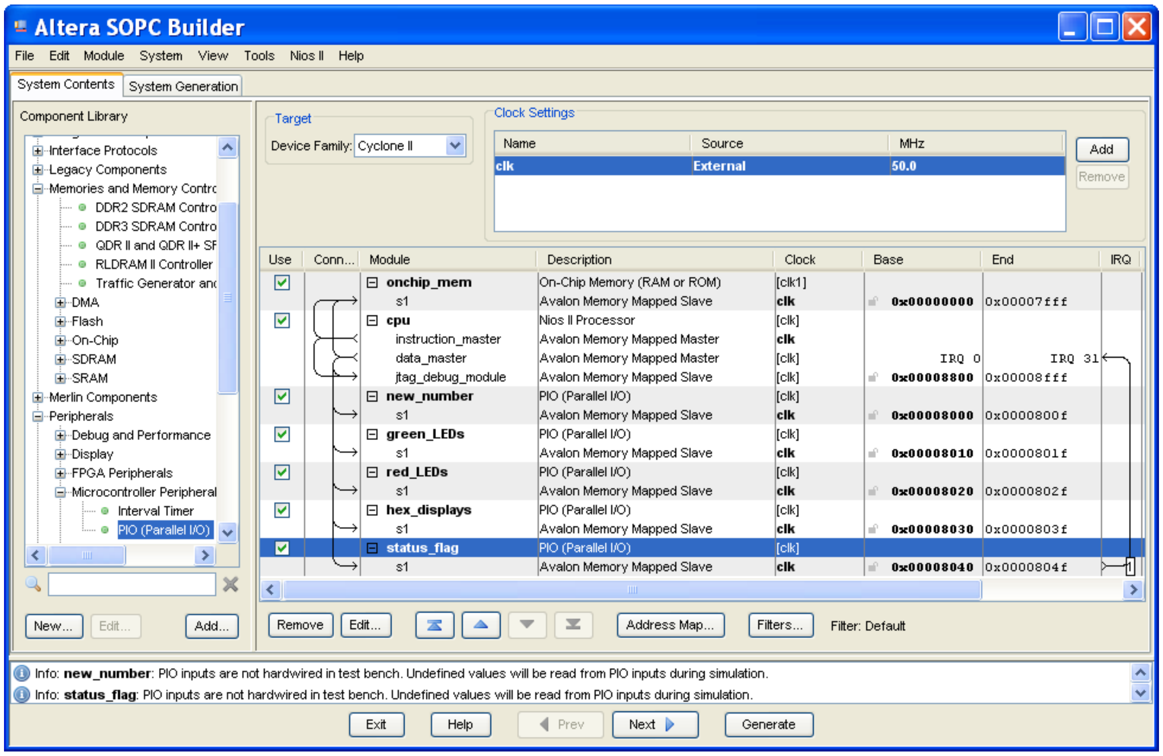
4. Having specified that the *status_flag* PIO can raise an interrupt request, it is necessary to specify the level (IRQ #) for this interrupt. This is done in the main SOPC window as illustrated in Figure 4. In the rightmost column, which is labeled IRQ, specify 1 as the desired level. Of course, the choice of level 1 is arbitrary. This choice will cause the bit position b_1 in control registers *ctl3* (*ienable*) and *ctl4* (*ipending*) to be associated with the *status_flag* PIO.
Figure 4 shows the resulting system specification.

5. Observe (and record for future reference) the assigned addresses and click on the Generate button to generate the specified system.

6. Write, in Verilog or VHDL, a file that instantiates the generated *nios_system* circuit and also defines the required connections to the switches and LEDs on the DE-series board. We will call this file *simple_computer.v/vhd*. Connect the *reset* input to the Nios II system in the file *nios_system.v/vhd* to the pushbutton switch KEY_0. Use the pushbutton KEY_1 as the input to the status-flag PIO. Keep in mind that the pushbutton switches are active low.

7. Assign the pins needed to make the necessary connections, by importing the *qsf* pin-assignment file for your board.

8. Compile the Quartus II project.



Name	Source	MHz
clk	External	50.0

Use	Conn...	Module	Description	Clock	Base	End	IRQ
✓		onchip_mem	On-Chip Memory (RAM or ROM)	[clk1]			
		s1	Avalon Memory Mapped Slave	clk	0x00000000	0x00007fff	
✓		cpu	Nios II Processor	[clk]			
		instruction_master	Avalon Memory Mapped Master	clk			
		data_master	Avalon Memory Mapped Master	[clk]	IRQ 0	IRQ 31	
		jtag_debug_module	Avalon Memory Mapped Slave	[clk]	0x00008800	0x00008fff	
✓		new_number	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	clk	0x00008000	0x0000800f	
✓		green_LEDs	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	clk	0x00008010	0x0000801f	
✓		red_LEDs	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	clk	0x00008020	0x0000802f	
✓		hex_displays	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	clk	0x00008030	0x0000803f	
✓		status_flag	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	clk	0x00008040	0x0000804f	1

```
Info: new_number: PIO inputs are not hardwired in test bench. Undefined values will be read from PIO inputs during simulation.
Info: status_flag: PIO inputs are not hardwired in test bench. Undefined values will be read from PIO inputs during simulation.
```

Figure 4. The Nios II system implemented by the SOPC Builder.

In the next four parts you will use the designed computer to investigate different aspects of performing I/O tasks, using both polling and interrupt approaches. In Parts II and III, the necessary programs are to be written in the Nios II assembly language. In Parts IV and V, the programs are to be written in the C language. You should use the Altera Monitor Program to handle your programs. For each part you should create a new project in the monitor program.

Part II

In this part, we will use the polling approach to read the numbers entered via the slider switches. The desired operation is that the user provides the next number by setting the slider switches accordingly and then pressing the pushbutton KEY_1 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, known as *polling*, 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 should be made.

In our case, the I/O device is the user who manually sets the toggle switches and presses the pushbutton key. The I/O interface that provides the desired control is the one-bit status-flag PIO circuit generated in Part I, which includes the edge-capture capability and conforms to the register map in Figure 1.

Perform the following:

1. Write an assembly-language program that reads the contents of the switches when the pushbutton KEY_1 is pressed, and displays this number on the green LEDs. Then, it adds the number to a sum that is being accumulated, and displays the sum on the red LEDs and HEX displays. Save the program in a file *lab5_part2.s*.

   When KEY_1 is pressed, the circuit designed in Part I will set the status-flag bit in the *edge-capture* register to 1. After reading the new number, your program has to clear the flag by writing a 0 into the *edge-capture* register.

2. Open the Altera Monitor Program and create a new project, as illustrated in Figure 5.

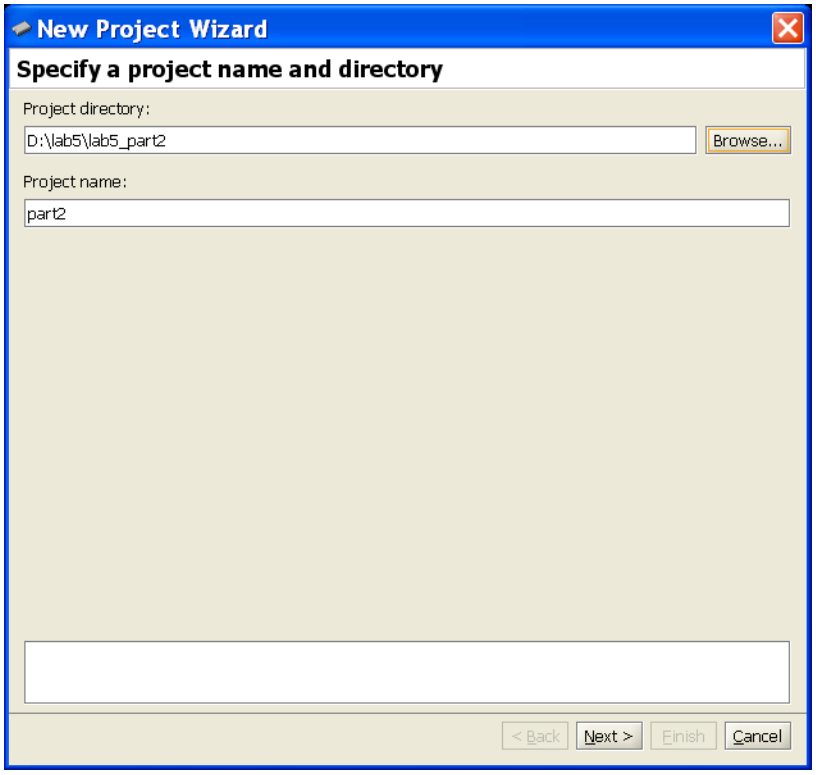
3. Specify that you wish to use the hardware that you designed, by choosing Custom System as shown in Figure 6. Find the file *nios_system.ptf*, which represents the designed Nios II system. Also, select the file *simple_computer.sof* which provides the information needed to download the designed system into the FPGA chip on the DE-series board.

4. Specify that an assembly-language program is to be used and that the program is given in the file *lab5_part2.s*, as shown in Figures 7 and 8, respectively.

5. Make sure that the USB-Blaster is used to provide the connection between the DE-series board and the host computer, as indicated in Figure 9.

6. Specify that your program has to be loaded in the on-chip memory, as illustrated in Figure 10. Since your system does not include any other memory, this choice will be made by default.

7. Click Finish in the window in Figure 10 and when a pop-up box asks you if you want to have your system downloaded onto the DE-series board click Yes.


New Project Wizard

Specify a project name and directory

Project directory: D:\lab5\lab5_part2

Project name: part2

Figure 5. Create a new project in the monitor program.

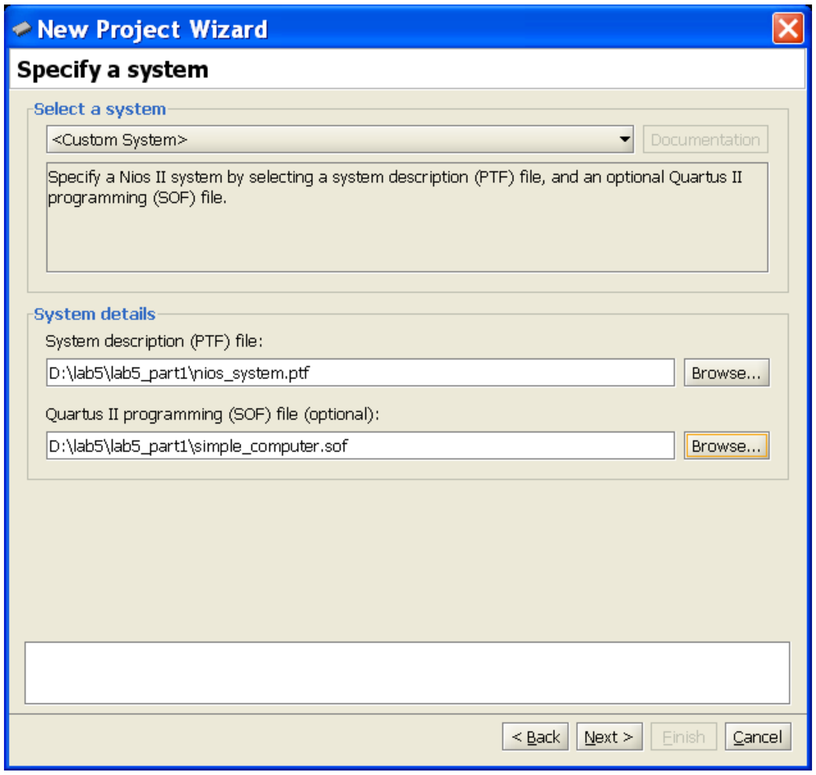

New Project Wizard
Specify a system
Select a system
<Custom System>
Documentation
Specify a Nios II system by selecting a system description (PTF) file, and an optional Quartus II programming (SOF) file.
System details
System description (PTF) file:
D:\lab5\lab5_part1\nios_system.ptf
Browse...
Quartus II programming (SOF) file (optional):
D:\lab5\lab5_part1\simple_computer.sof
Browse...
< Back
Next >
Finish
Cancel

Figure 6. Select the custom Nios II system that you designed.

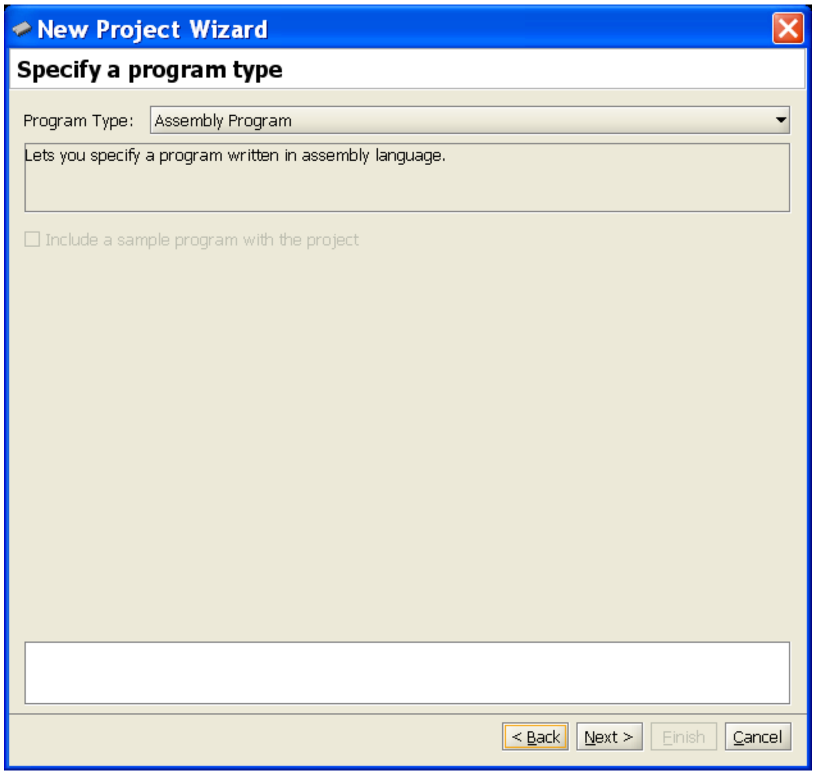
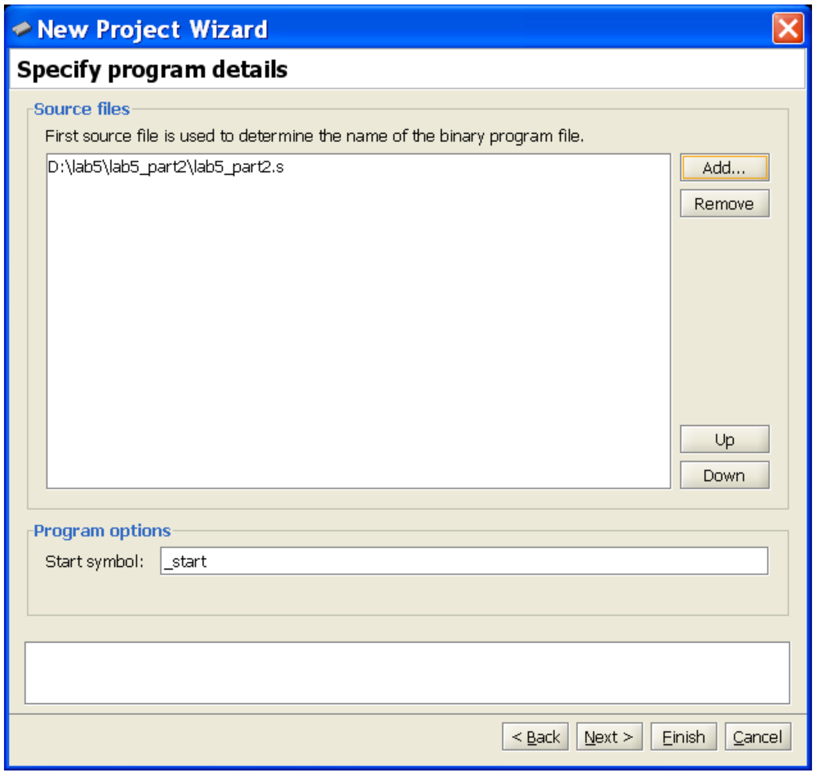

New Project Wizard
Specify a program type
Program Type: Assembly Program
Lets you specify a program written in assembly language.
Include a sample program with the project
< Back
Next >
Finish
Cancel

Figure 7. Specify that an assembly-language program is used.

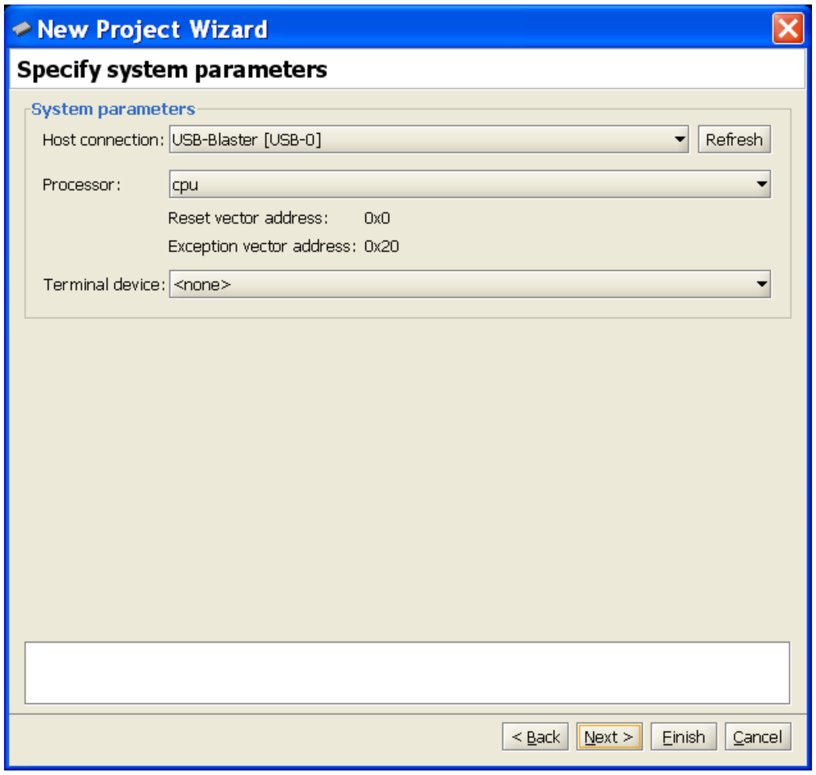


First source file is used to determine the name of the binary program file.

D:\lab5\lab5_part2\lab5_part2.s

Start symbol: _start

Figure 8. Specify the file that contains the application program.



Host connection: USB-Blaster [USB-0]

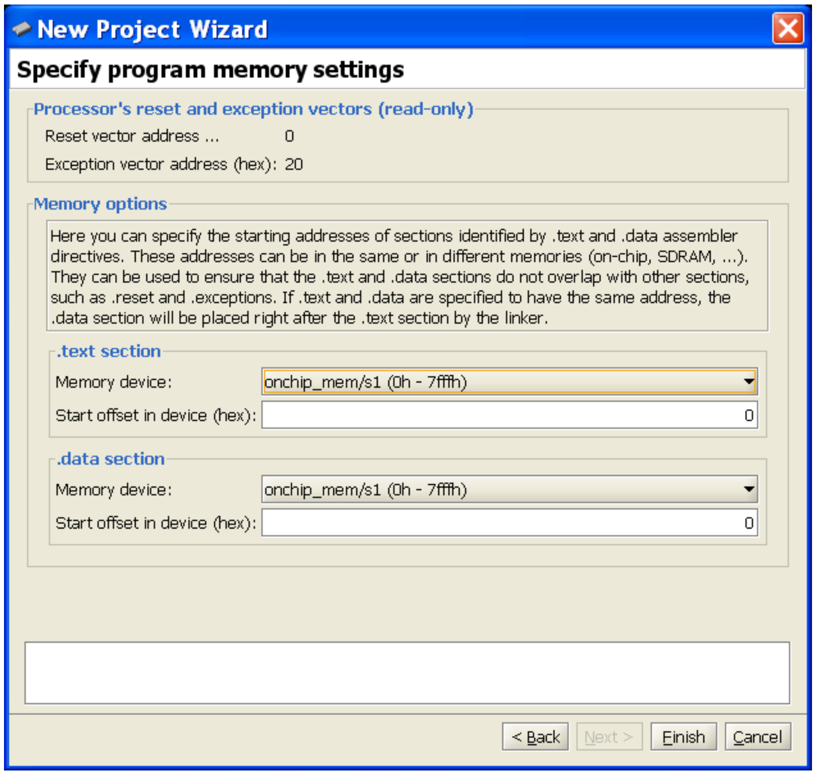
Processor: cpu

Reset vector address: 0x0

Exception vector address: 0x20

Terminal device: <none>

Figure 9. Specify the system parameters.


New Project Wizard

Specify program memory settings

Processor's reset and exception vectors (read-only)

Reset vector address ... 0

Exception vector address (hex): 20

Memory options

Here you can specify the starting addresses of sections identified by .text and .data assembler directives. These addresses can be in the same or in different memories (on-chip, SDRAM, ...). They can be used to ensure that the .text and .data sections do not overlap with other sections, such as .reset and .exceptions. If .text and .data are specified to have the same address, the .data section will be placed right after the .text section by the linker.

.text section

Memory device: onchip_mem/s1 (0h - 7fffh)

Start offset in device (hex): 0

.data section

Memory device: onchip_mem/s1 (0h - 7fffh)

Start offset in device (hex): 0

Figure 10. Specify where the program will be loaded in the memory.

8. Now, in the monitor window select Actions > Compile & Load to assemble and download your program.
9. Run the program and verify its correctness by inputting several numbers. The program should run continuously and a new number should be added each time the pushbutton KEY_1 is pressed.

Part III

Instead of using the polling approach to read new numbers from the slider switches, we now want to use interrupts for the same purpose. To accomplish this, we will use the ability of the status-flag PIO to raise an interrupt request when the pushbutton KEY_1 is pressed.

Modify your assembly-language program to realize the desired task by using the interrupt approach.

Part IV

Write a C-language program to perform the task in Part II.

Part V

Write a C-language program to perform the task in Part III.

Final Notes: Consult the description of Laboratory Exercise 4 for details on how interrupts should be handled in both the assembly-language and the C-language programs.

The system that you designed in Part I has many similarities with the DE-series Media Computer, but it is not identical. This means that the programs that you wrote for Labs 3 and 4 may need some modifications to run successfully on your system.

Preparation

Your preparation should include the following:

1. System design for Part I
2. Programs for Parts II to V

Laboratory Exercise 6

Timer/Counter Circuits

Timer/counter circuits are used in many embedded systems. This exercise introduces the *Interval Timer* module that is included in Altera's DE2 Media Computer. It shows how the timer can be used to trigger an action at specified time intervals. It also illustrates how the module's counter can be used to estimate the performance of an arbitrary application program, in terms of the total number of clock cycles needed to execute the program.

The Interval Timer has an internal counter which is set to a specified starting 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 starting value. The Interval Timer has a set of 16-bit registers that can be accessed as memory locations. These registers are shown in Figure 1. The address of the *Status* register is 0x10002000, which is the base address assigned to the Interval Timer. The *Control* register is at address 0x10002004. The starting value for the counter is specified in registers at addresses 0x10002008 (low-order 16 bits of the value) and 0x1000200C (high-order 16 bits of the value). As the counter value is decremented in each clock cycle, it is possible to capture a snapshot of the value at any time by performing a write to the address 0x10002010. This write operation causes the current 32-bit counter value to be stored into the two 16-bit registers at addresses 0x10002010 and 0x10002014. These registers can then be read to obtain the count value.

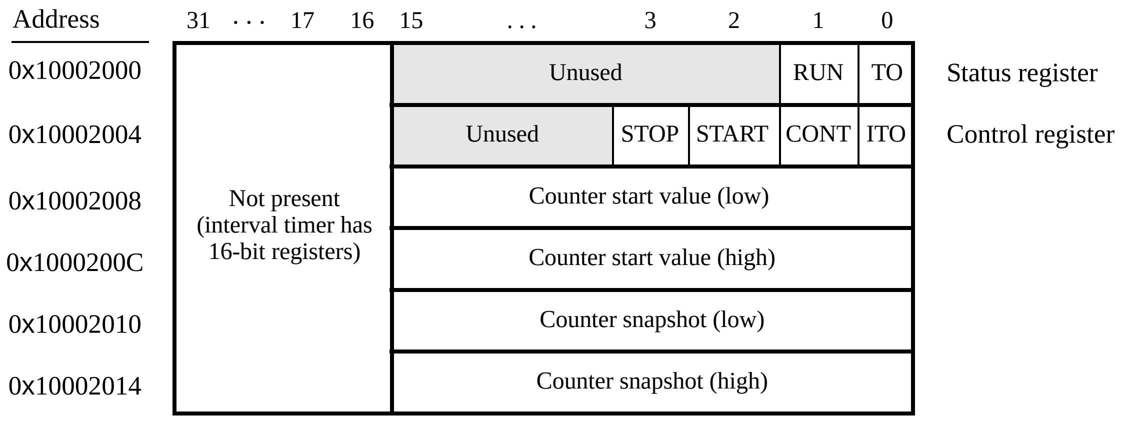


Figure 1. Registers in the Interval Timer

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

- b_0 (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, which must be done to clear an existing interrupt request.
- b_1 (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:

- b_0 (ITO) enables the Interval Timer interrupts when set to 1.
- b_1 (CONT) determines how the internal counter behaves when it reaches 0. If CONT = 1, the counter runs continuously by reloading the specified starting count value; otherwise, it stops when it reaches 0.

- b_2 (START) causes the internal counter to start running when set to 1 by a write operation.
- b_3 (STOP) stops the internal counter when set to 1 by a write operation.

To enable interrupts from the Interval Timer, the bit b_0 of the Nios II control register *ctl3* must be set to 1. The control register *ctl4*, also referred to as *ipending*, is used to determine which interrupt has occurred (if multiple I/O devices are enabled to raise interrupt requests). If an interrupt is disabled using the control register *ctl3*, an interrupt request from the corresponding device will be ignored and it will not show as having occurred in the control register *ctl4*.

Part I

To illustrate the convenience of using a timer, we wish to flash a light on a DE-series board in one-second intervals. Write an assembly-language program that continuously turns the green light LED_0 on for half a second and off for half a second. The program is to run on a DE-series Media Computer that you will download into the FPGA device on the board. Use the Altera Monitor Program to assemble, download and run your program to demonstrate that it works correctly.

Part II

In real-time embedded systems it may be necessary to know how much time is spent on execution of an application program. This time can be determined if we know how many clock cycles are needed to execute the program. The Interval Timer module in the Media Computer has a counter that can be used for this purpose. The counter can be set to some (large enough) value before the execution of the application program is started. Then, the count can be decremented in each clock cycle until the application program is finished.

Write an assembly-language program that determines the number of cycles needed to execute an application program and displays this number (in decimal form) on the seven-segment displays on the DE-series board. Run your program to demonstrate its validity.

A very simple application program, which computes the largest number in a list of integers, is provided as a test design file with this exercise. Use this application to test your program. Then, try your program on some larger application program that you have written.

Part III

Suppose that we wish to determine how long it takes to execute a specific portion, such as a particular loop. This can be done by inserting a **trap** instruction before the loop is entered, and another one at the exit from the loop. The software trap instruction causes the Nios II processor to raise an exception when it encounters this instruction.

Write an assembly-language program that determines the number of cycles needed to execute a loop in an application program and displays this number on the seven-segment displays on the DE-series board. Test your program on the loop in the example program mentioned in Part II. Then, use it on one of your own application programs.

Part IV

Implement the task in Part I using a C-language program.

Part V

Implement the task in Part II using a C-language program.

Part VI

Implement the task in Part III using a C-language program.

Preparation

As a part of your preparation you should do the following:

1. Write the assembly-language programs for Parts I to III.
2. Write the C-language programs for Parts IV to VI.

Laboratory Exercise 7

Introduction to Graphics and Animation

The purpose of this exercise is to learn how to display images and perform animation. We will use the DE-series Media Computer and the Video Graphics Array (VGA) Digital-to-Analog Converter (DAC) on an Altera DE-series Board. This lab document is intended for DE1, DE2, DE2-70 and DE2-115 boards only.

Background

The DE-series Media Computer uses a number of circuits, called *cores*, to control the VGA DAC and display images on a screen. These include a VGA Pixel Buffer and a VGA controller circuit, which are used together with the SRAM memory and the SRAM controller to allow programs executed by the Nios II processor to generate images for display on the screen. The relevant portion of the DE-series Media Computer is shown in Figure 1.

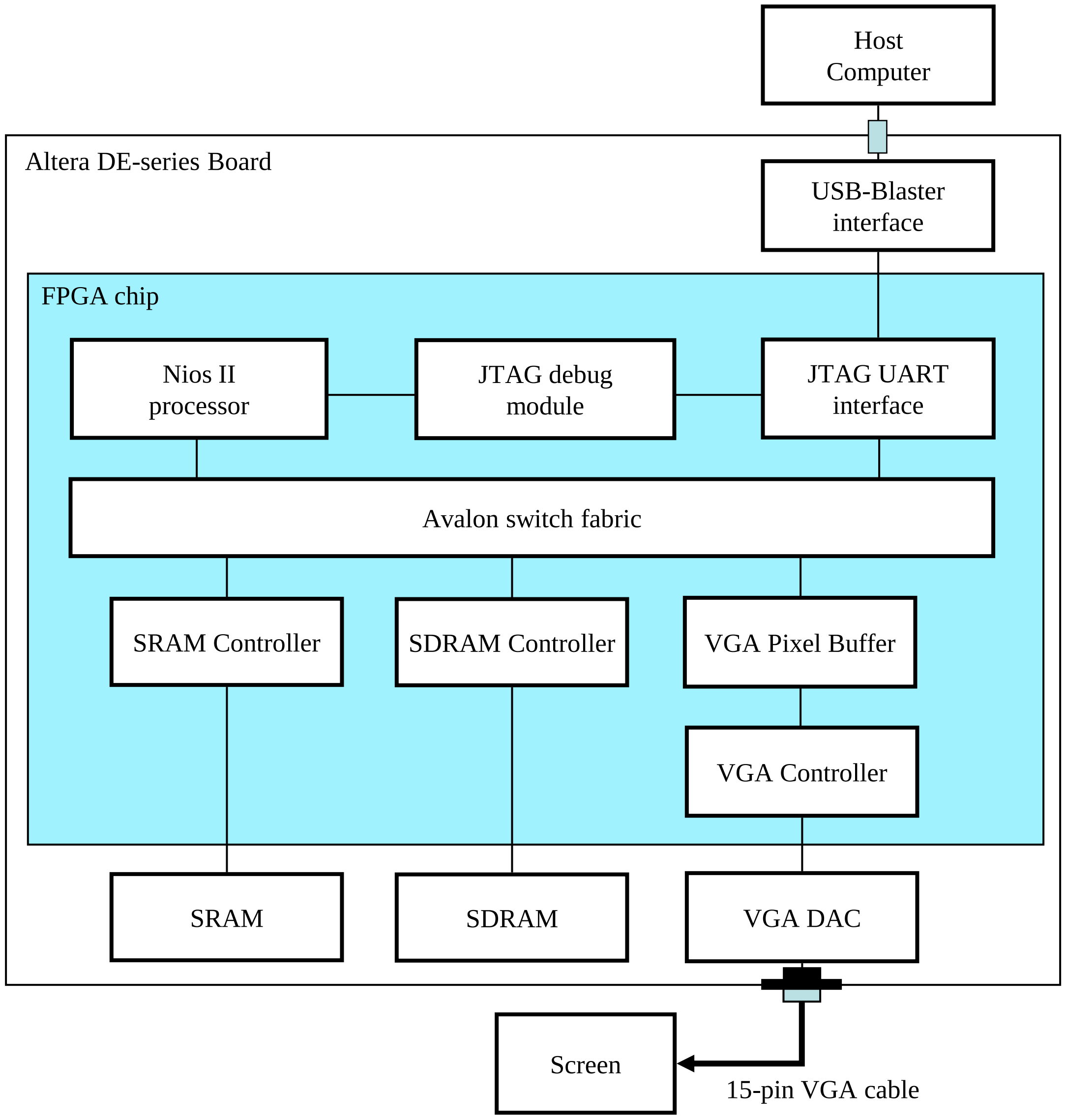


Figure 1. Portion of the DE-series Media Computer used in this exercise

The VGA pixel buffer serves as an interface between programs executed by the Nios II processor and the VGA controller. It gives the size of the screen and the location in the SRAM memory where an image to be displayed is stored. To display the image on the screen, the VGA pixel buffer retrieves it from the SRAM memory and sends it to the VGA controller. The VGA controller then uses the VGA Digital-to-Analog Converter to send the image data across the VGA cable to the screen.

An image consists of a rectangular array of picture elements, called *pixels*. Each pixel appears as a dot on the screen, and the entire screen consists of 320 columns by 240 rows of pixels, as illustrated in Figure 2. Pixels are arranged in a rectangular grid, with the coordinate $(0, 0)$ at the top-left corner of the screen.

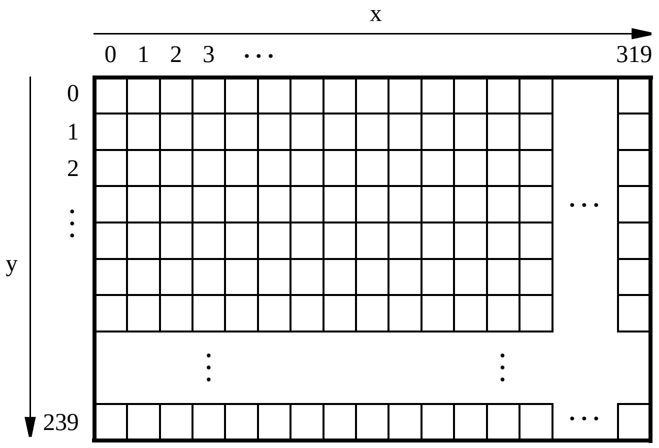
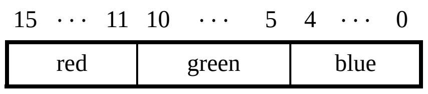
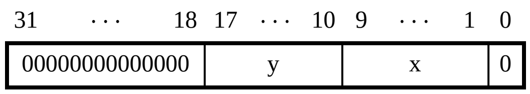


Figure 2. Pixel array.

The color of a pixel is a combination of three primary colors: red, green and blue. Any color can be created by varying the intensity of each primary color. We use a 16-bit halfword to represent the color of a pixel. The five most-significant and least-significant bits represent the intensity of the red and blue components, respectively, while the remaining six bits represent the intensity of the green color component, as shown in Figure 3a. For example, a red color would be represented by the value $(F800)_{16}$, a purple color by $(F81F)_{16}$, white by $(FFFF)_{16}$, and gray by $(8410)_{16}$.



(a) Pixel color



(b) Pixel (x,y) offset

Figure 3. Pixel color and offset.

The color of each pixel in an image is stored at a corresponding address in a buffer in the SRAM memory. The address of a pixel is a combination of a *base* address and an (x, y) offset. In the DE-series Media Computer, the buffer is located at address $(08000000)_{16}$, which is the starting address of the SRAM memory. The (x, y) offset is computed by concatenating the 9-bit x coordinate starting at bit b_1 and the 8-bit y coordinate starting at bit b_{10}, as shown in Figure 3b. This computation is accomplished in C programming language by using the left-shift operator:

$$\text{offset} = (x << 1) + (y << 10)$$

To determine the location of each pixel in memory, we add the (x, y) offset to the base address. Thus, the pixel at location $(0, 0)$ has the address $(08000000)_{16}$, the pixel at $(1, 0)$ has the address *base* + $(00000002)_{16}$ = $(08000002)_{16}$, the pixel at $(0, 1)$ has the address *base* + $(00000400)_{16}$ = $(08000400)_{16}$, and the pixel at location $(319, 239)$ has the address *base* + $(0003BE7E)_{16}$ = $(0803BE7E)_{16}$.

To display images from a program running on the DE-series Media Computer, the VGA pixel buffer module contains memory-mapped registers that are used to access the VGA pixel buffer information and control its operation. These registers, located at starting address $(10003020)_{16}$, are listed in Figure 4.

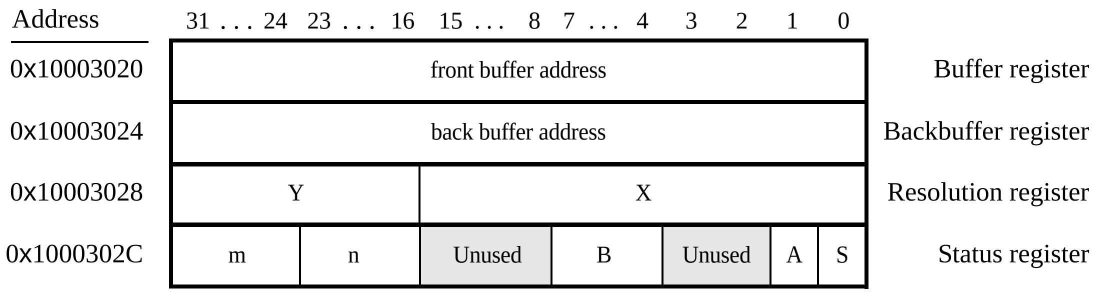


Figure 4. VGA pixel buffer memory-mapped registers.

The *Buffer* and *Backbuffer* registers store the location in the memory where two image buffers are located. The first buffer, called the *front buffer*, is the memory where the image currently visible on the screen is stored. The second buffer, called the *back buffer*, is used to draw the next image to be displayed. Initially, both registers store the value $(08000000)_{16}$. We will discuss how to use these buffers in Part III of the exercise.

The *Resolution* register holds the width and height of the screen in terms of pixels. The 16 most-significant bits give the vertical resolution, while the 16 least-significant bits give the horizontal resolution of the screen. The *Status* register holds information about the VGA pixel buffer. We will discuss the use of these registers as they are needed in the exercise.

Part I

In this part you will learn how to implement a simple line-drawing algorithm.

Drawing a line on a screen requires coloring pixels between two points, (x_1, y_1) and (x_2, y_2), such that they resemble a line as closely as possible. Consider the example in Figure 5.

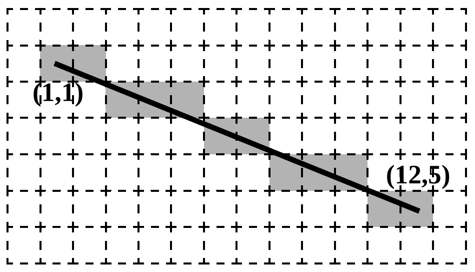


Figure 5. Drawing a line between points $(1, 1)$ and $(12, 5)$.

We want to draw a line between points $(1, 1)$ and $(12, 5)$. The squares represent pixels that can be colored. To draw a line using pixels, we have to follow the line and for each column color the pixel closest to the line. To form a line between points $(1, 1)$ and $(12, 5)$ we color the shaded pixels in the figure.

We can use algebra to determine which pixels to color. This is done using the end points and the slope of the line. The slope of the line is $slope = (y_2 - y_1)/(x_2 - x_1) = 4/11$. Starting at point $(1, 1)$ we move along the x axis and compute the y coordinate for the line as follows:

$$y = slope \times (x - x_1) + y_1$$

Thus, for column $x = 2$, the y location of the pixel is $\frac{4}{11} + 1 = 1\frac{4}{11}$. Because pixel locations are defined by integer values we round the y coordinate to the nearest integer, and determine that in column $x = 2$ we should color the pixel at $y = 1$. We perform this computation for each column between x_1 and x_2.

The approach of moving along the x axis has a drawback when a line is steep. A steep line spans more rows than columns, so if the line-drawing algorithm moves along the x axis to compute the y coordinate for each column there will be gaps in the line. For example, a vertical line has all points in a single column, so the algorithm would fail to draw it properly. To remedy this problem, we can alter the algorithm to move along the y axis when a line is steep. With this change, we can implement a line-drawing algorithm known as Bresenham's algorithm. The pseudo-code for the algorithm is shown in Figure 6.

```
draw_line(x0, x1, y0, y1)

    boolean is_steep = abs(y1 - y0) > abs(x1 - x0)
    if is_steep then
        swap(x0, y0)
        swap(x1, y1)
    if x0 > x1 then
        swap(x0, x1)
        swap(y0, y1)

    int deltax = x1 - x0
    int deltay = abs(y1 - y0)
    int error = -(deltax / 2)
    int y = y0
    if y0 < y1 then y_step = 1 else y_step = -1

    for x from x0 to x1
        if is_steep then draw_pixel(y,x) else draw_pixel(x,y)
        error = error + deltay
        if error >= 0 then
            y = y + y_step
            error = error - deltax
```

Figure 6. Pseudo-code for a line-drawing algorithm.

The original algorithm uses floating-point operations to compute the location of each pixel in the line. Since floating-point operations are usually much slower to perform, most implementations of this algorithm are altered to use integer operations only. For example, the code shown in Figure 6 is optimized to use only integer operations.

Write a C-language program that draws a few lines on the screen using this algorithm. Do the following:

1. Write a C-language program that implements the line algorithm.
2. Create a new project for the DE-series Media Computer using the Altera Monitor Program.
3. Download the DE-series Media Computer onto the DE-series board.
4. Connect a 15-pin VGA cable to the VGA connector on the DE-series board and the monitor.

   The VGA cable can be connected to the screen in two ways. If the screen has multiple VGA input ports, you can connect the VGA cable to an unused port. Then, using the buttons on the screen change the video source to the corresponding port. If the screen has only a single VGA port, a KVM (Keyboard-Video-Mouse) switch is needed. Using the KVM switch, you can connect multiple video sources to a single screen. This device will allow you to choose which video source will be displayed.
5. Compile and run your program.

Part II

Animation is an exciting part of computer graphics. Moving a displayed object is an illusion created by showing the same object at different locations on the screen. To move an object on the screen we must display it at one position first, and then at another later on. A simple way to achieve this is to draw the object at one position, and then erase it and draw it at another position.

The key to animation is timing, because to realize animation it is necessary to move objects at regular time intervals. The time intervals depend on the graphics controller. This is because the controller draws images onto a screen at regular time intervals. The VGA controller in the DE-series Media Computer redraws the screen every $1/60^{th}$ of a second. Since the image on the screen cannot change more often than that, this will be the unit of time.

To ensure that we change the image only once every $1/60^{th}$ of a second, we use the VGA pixel buffer to synchronize a program executing on the DE-series Media Computer with the redraw cycle of the VGA controller. This is accomplished by writing the value 1 into the *Buffer* register and waiting until bit b_0 of the *Status* register in the VGA pixel buffer becomes equal to 0. This signifies that a $1/60^{th}$ of a second has passed since the last time an image was drawn on the screen.

Write a C-language program that moves a horizontal line vertically across the screen and bounces it off the top and bottom edges of the screen. Your program should first clear the screen, by setting all pixels to black color, and then repeatedly draw and erase (draw the same line using the black color) the line during every redraw cycle. When the line reaches the top, or the bottom, of the screen it should start moving in the opposite direction.

Part III

Having gained the basic knowledge about displaying images and performing animation, you can now create a more interesting animation.

Write a program that animates eight small filled rectangles on the screen. These rectangles are moving continuously and bouncing off the edges of the screen. They are connected together with lines to form a chain. An illustration of the animation is shown in Figure 7. Figure $7a$ shows an instant where the rectangles are moving in the direction of red arrows, and Figure $7b$ shows the instant after the rectangles have moved.

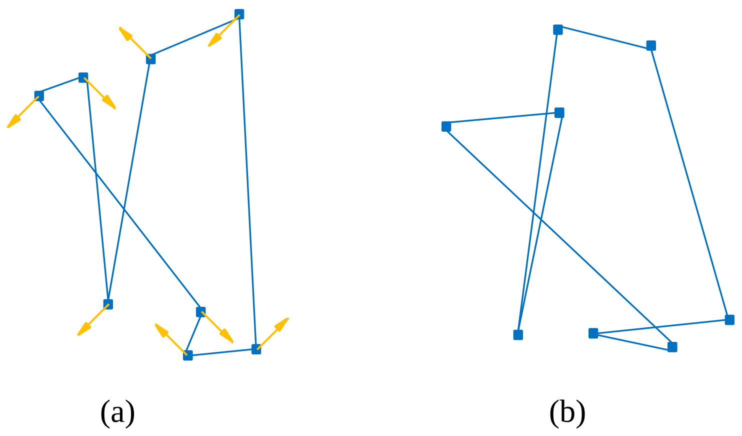


Figure 7. Two instants of the animaion.

In order to make the anmiation look different each time you run it, use *rand()* function for the initial positions of the rectangles and the directions of their movement.

You can enhance your program by using switches as an input to make your animation controllable. By toggling the switches, you can control the speed of the movement of rectangles, and stop/start the movement of some rectangles on the screen.

When you run your program, you may notice that the animation appears to flicker. This happens when the computer takes too much time to render a new image on the screen. This looks bad and is undesirable in computer animation.

A commonly used technique called double buffering can remedy this problem. Double buffering uses two buffers, rather than just one, to render an image on the screen. One of the buffers is visible on the screen and the other is hidden. The visible buffer is called the front buffer and the hidden buffer is called the back buffer. To create an animation using two buffers, we draw an image in the back buffer. When the image is ready and the VGA controller is about to draw a new image on the screen, we swap the front and the back buffers. This operation makes a seamless transition from one image to another. Once the buffers are swapped, the back buffer becomes the front buffer and vice versa.

The VGA pixel buffer in the DE-series Media Computer supports double buffering. The location of the buffers in memory is stored in registers *Buffer* and *Backbuffer*, shown in Figure 4. Initially, the location of both buffers is the same, thus only one buffer is used. To enable double buffering, we need to separate the front and the back buffers by setting the *Backbuffer* register to address $(08040000)_{16}$. This will cause half of the SRAM memory in the DE-series Media Computer to be used for the front buffer, and the other half for the back buffer.

Each time you want to display the image in the back buffer, you need to write 1 to the *Buffer* register. By doing this, the VGA pixel buffer will swap the address stored in *Buffer* register and *Backbuffer* register. The swapping can take up to $1/60^{th}$ second because it must wait until the front buffer has been completely drawn. You should continuously poll bit b_0 of the *Status* register for a value 0 before you can draw a new image in the new back buffer.

When using double buffering, your program should only draw images in the back buffer and erase the back buffer after every buffer swap. Also, the program should use two pointers pointing to the front buffer and the back buffer. When you ask the VGA pixel buffer to swap its buffers, you should swap the pointers as well.

Preparation

The recommended preparation for this laboratory exercise includes C code for Parts I to III.

Laboratory Exercise 8

Implementing Device Drivers

The purpose of this exercise is to learn how device drivers can be implemented in embedded systems. You will create a PS/2 mouse driver that works with the VGA display available in an Altera DE-series Media Computer. A block diagram of the DE2 Media Computer is shown in Figure 1. The operational details of the PS/2 port and the VGA display are described in the *DE2 Media Computer for the Altera DE2 Board* manual.

Prerequisite: It is necessary to do Laboratory Exercise 7 to gain the needed background knowledge.

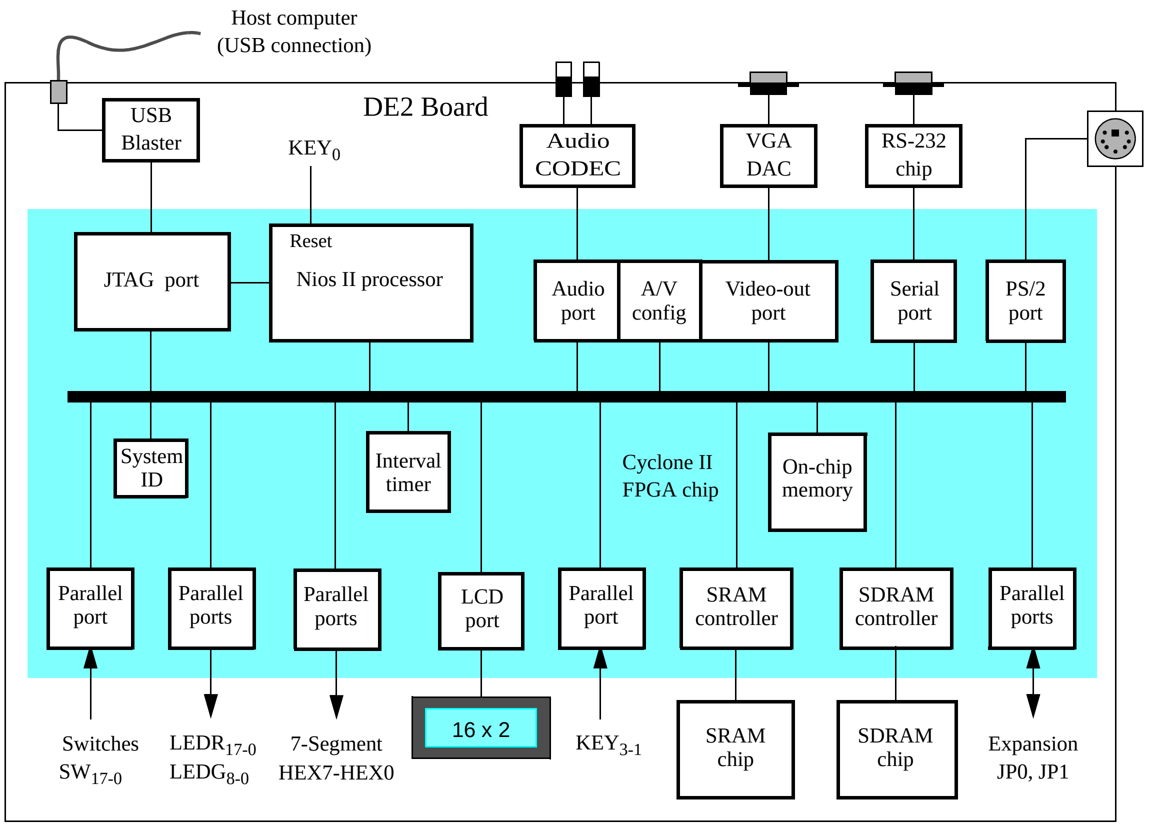


Figure 1: System diagram of the DE2 Media Computer.

Background

A device driver comprises a set of software subroutines that allow computer programs to access a peripheral device, without the need for the programmer to be familiar with the low-level details of how a peripheral device functions. A simple example of a peripheral device we use in computers is a keyboard, which can be connected to a computer via a PS/2 or a USB interface. Programmers who write a C-language program to read a string of characters from the keyboard on a Personal Computer need not know how keystrokes on the keyboard are received by their program. For example, to read an integer from the keyboard, a C programmer uses the statement:

```
scanf("%d", &some_variable);
```

Providing a high-level of abstraction for peripheral devices is the purpose of device drivers. Implementing device drivers often requires a great deal of knowledge on the part of a programmer, not only with respect to how a peripheral device operates, but also the context in which it is used.

Computer mouse is another example of a simple device that can be connected via a PS/2 or a USB interface. Today, a mouse is likely to be connected using the USB connection. Previously, it was common to use the PS/2 interface. Since the PS/2 interface is much simpler, we will use it in this laboratory exercise.

Part I

In this part, you will take the first steps to communicate with the mouse device. To do this, you have to be able to send and receive information from the mouse through the PS/2 port in the DE-series Media Computer.

The PS/2 port in the Media Computer is implemented by a PS/2 Port IP core. This core includes a 256-byte First-In First-Out (FIFO) buffer that stores data received from a PS/2 device.

The interface to the PS/2 port in the Media Computer has two memory-mapped registers, shown in Figure 2.

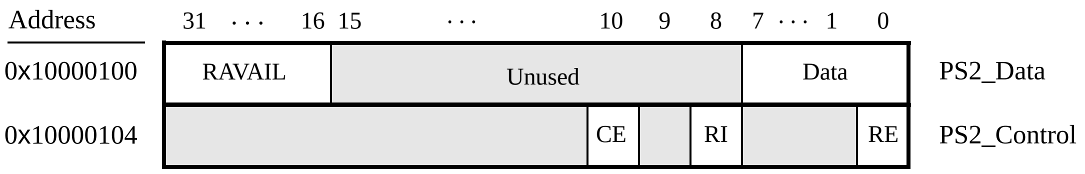


Figure 2: PS/2 Port Memory-Mapped Registers.

The PS2_Data register consists of *RAVAIL* and *Data* fields. The *RAVAIL* field indicates the number of bytes of data received through the PS/2 port that are currently stored in the FIFO buffer. The first byte of the buffer can be read through the *Data* field. Note that reading any part of the PS2_Data register causes the FIFO buffer to eject the first byte from the FIFO. The PS2_Control register is used to enable interrupts for the PS/2 port. Interrupts can be enabled by setting the *RE* field. When *RE = 1*, the PS/2 port generates an interrupt request when *RAVAIL* is greater than 0. While the interrupt is pending, the *RI* field is set to 1, and can be cleared by emptying the PS/2 FIFO. The *CE* field is set by the PS/2 port if an error occurs when a command is sent to a PS/2 device. We will use this register in the later parts of the exercise.

In your first attempt to communicate with the mouse, you will use polling to receive data from the PS/2 port. Do the following:

1. Write a C-language program that polls the PS_Data register and displays the last three bytes received on the 7-segment displays. Your program should first enable communication with the mouse by sending the byte 0xF4 to the PS_Data register. Your program should then continuously read the contents of the PS2_Data register. Each time, check the *RAVAIL* field to determine if the byte in *Data* field is valid data. Only when the number in *RAVAIL* field is larger than 0, should you save the data and display it on the 7-segment displays.
2. Create a new project in the Altera Monitor Program for the Altera DE-series Media Computer and include your source code.
3. Compile and load your program onto the DE-series board.
4. Connect a PS/2 Mouse to the PS/2 port on the board.
5. Run the program and observe the messages received from the mouse device as you move the mouse around and press the mouse buttons.

Part II

In this part, you will enhance your program to properly initialize the mouse device. To do this, you will need to understand some of the commands that can be sent to the mouse (such as the 0xF4 command we used in Part I), as well as the messages that the mouse can send in response to these commands. The necessary commands and mouse responses to the commands are shown in Table 1.

To ensure the proper operation of the mouse device, it is important to reset the mouse and enable it before attempting to process any messages. Thus, your program should first send the reset command to the mouse and

Table 1: Some of the mouse commands along with the expected responses.

Command Description	**Command Byte**	**Response**
Reset the mouse to default mode	0xFF	Responds with a **0xFA** message, followed by a 2-byte message **0xAA00** if successful. A byte **0xFC** will be sent otherwise to indicate an error.
Enable Mouse to send position and button status messages	0xF4	Responds with a single **0xFA** byte if successful.
Disable Mouse	0xF5	Responds with a single **0xFA** byte if successful. Send the Enable mouse command to resume receiving messages about the user's interactions with the mouse.

await a response. If the reset is successful, the mouse will send a 3-byte sequence equal to **0xFAAA00**. Upon receiving this response from the mouse, send the **Enable Mouse** command and wait for the acknowledgement **0xFA**.

Resetting the mouse can fail because of a malfunction, a communication error or because the mouse is not connected to the system. If a mouse malfunctions and does not respond to commands, then it can be treated as not connected. If it responds with the **0xFC** byte, then resend the reset command to ensure that the mouse is reset correctly before proceeding further.

A disconnected mouse may be reconnected at any time and should be initialized properly. Fortunately, when a mouse is plugged into a PS/2 port it will reset itself and send a 2-byte message **0xAA00** as though a reset command was sent to it. Your program should watch for the **0xAA00** message from the mouse. If this sequence is detected, your program should send the reset command to the mouse followed by the mouse enable command.

Write a C-language program that communicates with a mouse through the PS/2 port on the DE-series Media Computer. Your program should correctly reset and enable the mouse device, as well as handle the case of the mouse being reconnected to the system while the program is running.

Part III

In this part, you will learn how to process messages that the mouse device sends to specify the change in its position and the state of its buttons. The change in position is indicated by two 9-bit signed numbers, one for the change in the horizontal position, and one for the change in the vertical position. Moving the mouse to the right causes the horizontal change in mouse's position to be positive, and moving the mouse to the left is indicated by a negative value. Similarly, when you move the mouse forward, the change in vertical position is positive, and when you pull it towards you it is negative. The state of mouse buttons is specified using three bits, one per button in a basic 3-button mouse. When a button is pressed the corresponding bit is set to 1, otherwise the bit is set to 0.

By default, a mouse that is enabled will send 3-byte packets through the PS/2 port to communicate its state. The format of the packet is shown in Table 2.

Table 2: Three-byte mouse data packet.

	bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
byte 1	y overflow	x overflow	y sign	x sign	1	Middle btn	Right btn	Left btn
byte 2	Mouse x movement							
byte 3	Mouse y movement							

The first byte has bit 3 set to 1 to indicate that the two bytes to follow are part of a 3-byte data packet. In addition, it contains the state of three buttons in bits 2 through 0, as well as the sign and overflow bits for the movement along the x and y axes. The remaining two bytes contain 8-bit parts of x and y movement of the mouse.

Write a C-language program that initializes the mouse and processes the 3-byte movement packets received from the mouse. Your program should decode the packets and display the state of the buttons on **LEDR[2..0]**, the change in x position on **HEX[7..4]** and the change in y position on **HEX[3..0]**.

Part IV

In this part, you will handle the mouse messages using an Interrupt Service Routine. The interrupt service routine (ISR) is a function that is executed when an interrupt occurs. An interrupt will occur when the *RE* bit in the PS/2 control register is set to 1 and the PS/2 port has a message ready to be read.

Modify your C-language program from Part III to do this by implementing four functions:

1. initialize_driver - enable the driver to process messages from the mouse. This function should reset the mouse and enable interrupts for the PS/2 port in the DE-series Media Computer.
2. disable_driver - disable the processing of messages from the mouse. Also, disable interrupts for the PS/2 port.
3. get_mouse_change - get the most recent change in position and mouse buttons as reported by the mouse.
4. PS2_ISR - the interrupt service routine that handles the mouse events.

Note that the procedure for dealing with interrupts in a DE-series Media Computer is the subject of Laboratory Exercise 4.

Part V

In this part, you will enhance your device driver to keep track of the location of the mouse and limit its movements to specified boundaries. To do this, you will define five variables: two to keep track of the current horizontal and vertical position of the mouse, one to keep the state of mouse buttons, and two to specify the maximum horizontal and vertical coordinates at which the mouse can be located.

You are to create a mouse device driver comprising the following functions:

1. initialize_driver - as in Part IV.
2. disable_driver - as in Part IV.
3. set_mouse_bounds(max_x, max_y) - set the maximum x and y coordinates a mouse can have.
4. get_mouse_state - read the current position of the mouse on the screen and the state of its three buttons.
5. get_mouse_change - as in Part IV.
6. PS2_ISR - as in Part IV.

Write a program that shows the position of the mouse using **HEX[7..4]** to display the x position, and **HEX[3..0]** to display the y position. Set the maximum x and y coordinates of the mouse to be 319 and 239, respectively.

Part VI

In this part, you are to implement routines to draw the mouse pointer on the screen. You will use the VGA cores in the DE-series Media Computer to draw the mouse pointer on the screen. These cores implement the VGA interface that is able to produce an image on a computer screen at a resolution of 320 by 240 pixels. The image to be drawn on the screen is stored in memory and can be easily changed. Refer to Laboratory Exercise 7 for details that you need to perform the following parts.

To draw a mouse pointer on the screen, it is necessary to define the shape of the pointer as well as a secondary buffer that contains the contents of the screen beneath the mouse pointer. The shape of the pointer is drawn on the screen just like any other image. You should use the secondary buffer to store the part of the screen on which the mouse is being drawn. Then, when the mouse is moved in the future, you will be able to restore the contents of the screen easily and move the mouse to a new location.

To define the shape of the mouse pointer use a matrix of 16 rows by 8 columns. Each entry in the matrix should contain one of three values:

1. 0 - to indicate that the correpsonding pixel of the mouse pointer is black.
2. 1 - to indicate that the correpsonding pixel of the mouse pointer is white.
3. -1 - to indicate that the correpsonding pixel of the mouse pointer is transparent. Transparent pixels are not drawn when the mouse pointer is being drawn.

The matrix and the image of the mouse pointer are given in Figure 3. Note that light blue is used to indicate the transparent pixels.

0	-1	-1	-1	-1	-1	-1	-1
0	0	-1	-1	-1	-1	-1	-1
0	1	0	-1	-1	-1	-1	-1
0	1	1	0	-1	-1	-1	-1
0	1	1	1	0	-1	-1	-1
0	1	1	1	1	0	-1	-1
0	1	1	1	1	1	0	-1
0	1	1	1	1	0	0	0
0	1	1	1	0	-1	-1	-1
0	0	0	1	0	-1	-1	-1
0	-1	0	1	0	-1	-1	-1
-1	-1	-1	0	1	0	-1	-1
-1	-1	-1	0	1	0	-1	-1
-1	-1	-1	-1	0	1	0	-1
-1	-1	-1	-1	0	1	0	-1
-1	-1	-1	-1	-1	0	-1	-1

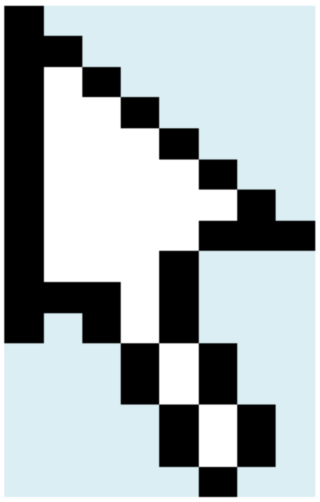


Figure 3: The matrix and the image of the mouse pointer.

The secondary buffer should also consist of 16 rows and 8 columns, where each entry stores the color of the pixel over which the mouse pointer will be drawn.

Adapt your mouse device driver to move the mouse pointer on the screen. Define two new functions for your driver:

1. draw_mouse - store the image underneath the mouse pointer and draw the mouse pointer at the current location. Use the matrix described above to draw the mouse pointer on the screen, assuming that the top-left corner of the mouse pointer image is located at the current location of the mouse. This function should only be visible to the driver.
2. erase_mouse - erase the mouse from the screen by redrawing the image beneath the mouse. This function should only be visible to the driver.

Modify your PS/2 Interrupt Service Routine to call the above two functions whenever the mouse position changes.

Run your program from Part V and connect the VGA output of the DE-series board to a computer monitor. You should be able to see the coordinates of the mouse pointer on the HEX displays and the mouse pointer on the screen, as you move the mouse.

Part VII

In this part, you are to combine the mouse pointer with animation on the screen. Write a program to implement a simple game where a filled rectangle moves and bounces off the edges of the screen. The player moves the mouse to position its pointer on the moving image of the rectangle, at which point clicking the left button on the mouse should change the color of the rectangle and stop its motion. When the button is released, the rectangle should start moving again. The game will be more interesting if you acclerate the speed or shrink the size of the rectangle after each successful catch.

To perform the animation, every 30th of a second you should hide the mouse pointer, erase the rectangle, draw the rectangle at a new location, and then show the mouse pointer again. You should define two new functions:

1. hide_mouse - causes the driver to erase the mouse from the screen. The driver should still process messages from the mouse, and update its position and the state of the buttons, but the mouse pointer should not be drawn on the screen.
2. show_mouse - enables the driver to draw the mouse pointer on the screen. If the pointer was previously hidden, the driver should draw the pointer when this function is called.

To wait a 30th of a second, we can take advantage of the fact the VGA controller continuously redraws the image from the memory onto the screen and one redraw cycle takes a 60th of a second. We can ask the VGA pixel buffer to let us know when it has finished drawing the most recent image, as explained in Laboratory Exercise 7. Waiting for two such cycles, gives us the 30th of a second delay we seek.

When you run your program you may notice that as you move the mouse pointer, parts of it are left on the screen, despite the fact that you were careful to erase and redraw it in both your ISR and in your main program. The reason is that an interrupt can occur while your main program is hiding/redrawing the mouse. You will need to adjust your driver to prevent this.

Preparation

The recommended preparation for this laboratory exercise includes C code for Parts I through VII.

Laboratory Exercise 9

Interface Protocols

Many embedded systems are implemented using a combination of a processor, memory, and off-the-shelf peripheral components. To communicate with a processor, peripheral components support a predefined set of communication interfaces. Serial interfaces are usually used to reduce the number of pins needed to exchange data. One of the popular communication standards is the Serial Peripheral Interface (SPI).

The SPI comprises four wires, as shown in Figure 1. They are: clock (CLK), Master-Out Slave-In (MOSI), Master-In Slave-Out (MISO) and chip select (CS). The clock signal is generated by the master to synchronize the exchange of data. The MOSI line is used by the master to send commands and data to the slave, while the MISO line is used by the slave to respond to commands and send data back to the master. The fourth line, called chip select, enables or disables the slave device. When multiple slave devices are present, a separate chip select line is used for each slave.

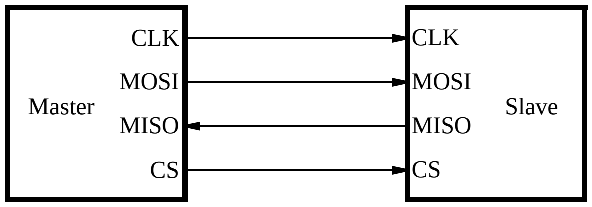


Figure 1: Diagram of SPI connecting a master and a slave.

An example of a device that supports SPI is a Secure Data (SD) card. SD cards are portable non-volatile memory devices with a capacity to store a large amount of data and provide encryption for security purposes. They are widely used to store pictures in digital cameras.

This exercise focuses on designing an embedded system on an Altera DE-series board that can access an SD card using an SPI interface.

Background

A block diagram of the SD card is shown in Figure 2. It consists of a 9-pin interface, a card controller, a memory interface and a memory core. The 9-pin interface allows the exchange of data between a connected system and the card controller. The controller can read/write data from/to the memory core using the memory core interface. In addition, several internal registers store the state of the card.

The controller responds to two types of user requests: control and data. Control requests set up the operation of the controller and allow access to the SD card registers. Data requests are used to either read data from or write data to the memory core.

To interface with an SD card, the Altera DE-series board features an SD card slot, on which pins labeled as CLK, CMD, DAT0, and CD/DAT3 are connected to the FPGA. These four pins are used to exchange data between the FPGA and the SD card using one of two modes: the SD mode or the SPI mode. By default, the SD card operates in the SD mode. However, in this exercise we will set the SD card into the SPI mode and communicate with it using the SPI protocol. Table 1 shows each pin on the SD card, what function it is used for in the SPI mode, and the pin on the FPGA to which it is connected on Altera DE-series boards.

In this exercise you will create a hardware interface to the SD card using these four pins and write software to set the SD card into SPI mode to exchange data with it using SPI.

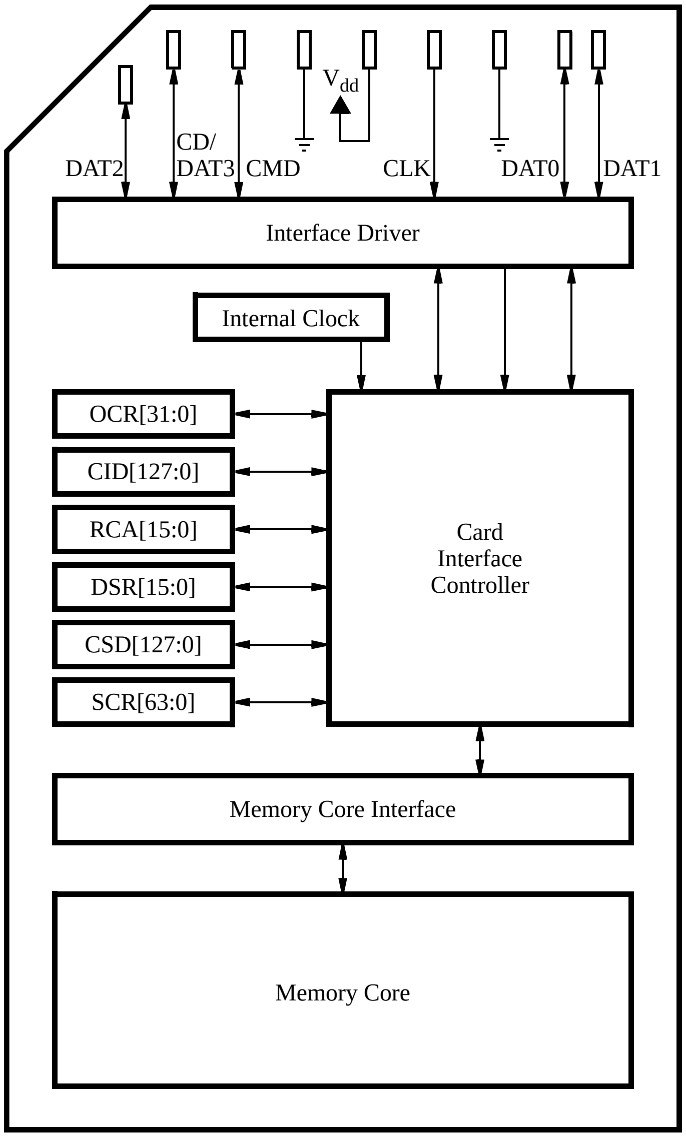


Figure 2: A block diagram of an SD card.

Table 1: SD Card Pin Function and Location Assignments for Altera DE-series boards.

SD Card Pin	**SPI Pin Function**	**FPGA Pin Location**					**Node Name**
		DE0	**DE1**	**DE2**	**DE2-70**	**DE-115**	
`CMD`	`MOSI`	`PIN_Y22`	`PIN_Y20`	`PIN_Y21`	`PIN_W28`	`PIN_AD14`	`SD_CMD`
`DAT0`	`MISO`	`PIN_AA22`	`PIN_W20`	`PIN_AD24`	`PIN_W29`	`PIN_AE14`	`SD_DAT`
`CD/DAT3`	`CS`	`PIN_W21`	`PIN_U20`	`PIN_AC23`	`PIN_Y30`	`PIN_AC14`	`SD_DAT3`
`CLK`	`CLK`	`PIN_Y21`	`PIN_V20`	`PIN_AD25`	`PIN_T26`	`PIN_AE13`	`SD_CLK`

Part I

In this part, you are to design an SPI interface to the SD card, specified in Figure 3, and implement a system with this interface component using the SOPC Builder.

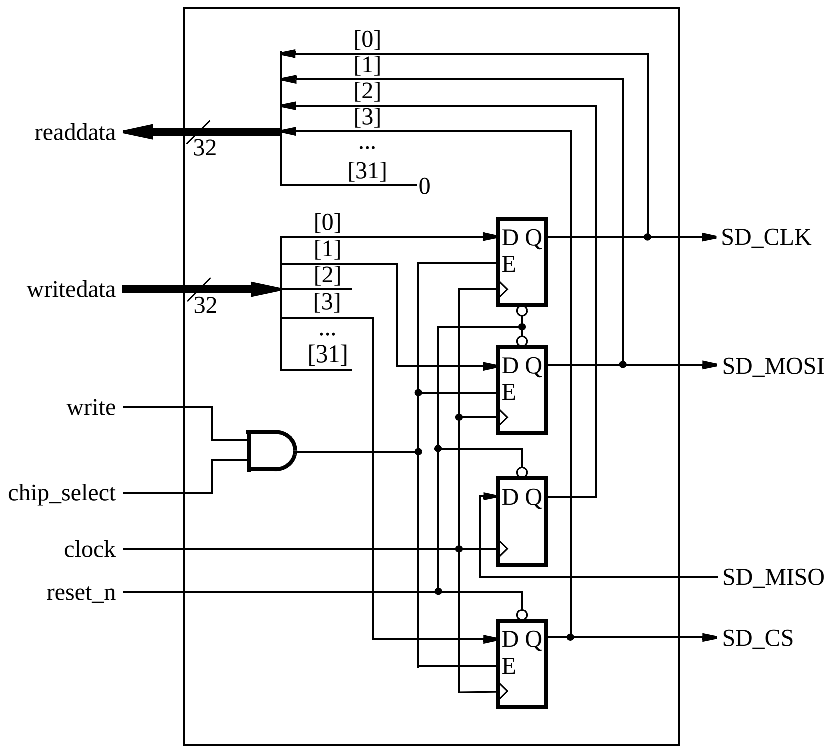


Figure 3: A circuit diagram of a simple SPI interface component.

The SPI interface component consists primarily of four flip-flops. Each flip-flop is responsible for transmitting a signal via the SD_CLK and SD_MOSI lines and receiving responses from the SD card on the SD_MISO line. The fourth SPI line, SD_CS, is used to properly initialize the SD card and set it to function in the SPI mode.

The left-hand side of Figure 3 shows several signals that serve as an Avalon memory-mapped slave interface. They comprise readdata lines to read the state of the flip-flops, writedata lines to send data through the SPI, as well as the write and chip_select lines to ensure that only data intended for this component is accepted from the Avalon Interconnect.

Implement the component shown in Figure 3 in a module called my_spi_interface. Then, include it in an SOPC Builder system comprising:

1. Nios II processor, standard version
2. SRAM Memory Controller (available as a University Program IP Core). It should have a base address of 0x00000000.
3. JTAG UART core (you can find it under Interface Protocols/Serial)
4. Your SPI interface component, with a base address set to 0x40000000. If you don't know how to create a hardware module as an SOPC Builder component, read the Tutorial called *Making SOPC Builder Components Using Verilog Designs* available on the Altera University Program website.

Connect all inputs and outputs to the circuit as needed, ensuring that the clock input is connected to the 50-MHz clock on the DE-series board. Compile and download your design onto the Altera DE-series board. Test your system by reading and writing data to your SPI interface component. Since there is no SD card connected on the right-hand side of Figure 3, test the component by simply writing data into the four flip-flops and reading it back.

Part II

In this part, you will use your SPI interface component to communicate with an SD card. To do this, you will set the SD card into the SPI mode and then send commands to it.

Communication with the SD card is performed by sending commands to it and receiving responses from it. A valid SD card command consists of 48 bits as shown in Figure 4. The leftmost two bits are the start bits which we set to (01). They are followed by a 6-bit command number and a 32-bit argument where additional information may be provided. Next, there are 7 bits containing a Cyclic Redundancy Check (CRC) code, followed by a single stop bit (1).

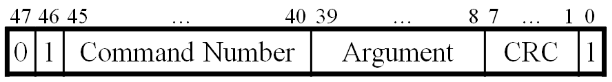


Figure 4: Format of a 48-bit command for an SD card.

The CRC code is used by the SD card to verify the integrity of a command it receives. By default, the SD card ignores the CRC bits for most commands (except CMD8) unless a user requests that CRC bits be checked upon receiving every message. In this exercise, we assume that errors during transmission of commands and data between your circuit and the SD card will not occur, and thus omit the discussion of CRC codes. Readers interested in using the CRC codes may consult the *SD Specification - Physical Layer version 2.00* that can be found on the Web.

Notation CMDX is used to represent the command with a command number X. For example, CMD8 means the command with $(001000)_2$ as its command number.

To communicate with the SD card you will write a program that runs on the Nios II processor. The program will control the values of all signals in the SPI interface component, including the SD_CLK signal, by reading and writing data to the component.

Sending a command to the SD card is performed in serial fashion. By default, the MOSI line is set to 1 to indicate that no message is being sent. The process of sending a message begins with placing its most-significant bit on the MOSI line and then toggling the SD_CLK signal from 0 to 1 and then back from 1 to 0. Then, the second bit is placed on the MOSI line and again the SD_CLK signal is toggled twice. Repeating this procedure for each bit allows the SD card to receive a complete command.

Once the SD card receives a command it will begin processing it. To respond to a command, the SD card requires the SD_CLK signal to toggle for **at least 8 cycles**. Your program will have to toggle the SD_CLK signal and maintain the MOSI line high while waiting for a response. The length of a response message varies depending on the command. The commands you will use in this exercise will get a response mostly in the form of 8-bit messages, with two exceptions where the response consists of 40 bits.

To ensure the proper operation of the SD card, the SD_CLK signal should have a frequency in the range of 100 to 400 kHz. A simple way to do this is to wait an appropriate amount of time between changing the value of the SD_CLK signal. You can experiment with various methods to generate the desired signal or use the code provided in Figure 5.

Note that the keyword **volatile** is used for the pointer to the SPI interface. This keyword is required to ensure that each processor read operation directly accesses the SPI hardware component to obtain its current value. The C compiler used in the Altera Monitor Program implements this operation by using a **ldwio** Nios II instruction. If the volatile keyword is not included, the C compiler may choose to read the SPI interface hardware only once, and then store its value in a Nios II general-purpose register. Then, subsequent accesses to this address would not provide updated values, and just use the value stored in the register.

```
volatile int *spi_interface = (int *) 0x40000000;

void create_clock_pulse(void) {
    int index;
    int contents;

    for (index = 0; index ≤ 15; index++){
        contents = *spi_interface;
    }
    *spi_interface = contents | 0x01;
    for (index = 0; index ≤ 15; index++){
        contents = *spi_interface;
    }
    *spi_interface = contents & 0x0E;
}
```

Figure 5: C-language code to generate an SD_CLK pulse.

To communicate with the SD card, your program has to place the SD card into the SPI mode. To do this, set the MOSI and CS lines to logic value 1 and toggle SD_CLK for at least 74 cycles. After the 74 cycles (or more) have occurred, your program should set the CS line to 0 and send the command CMD0:

01 000000 00000000 00000000 00000000 00000000 1001010 1

This is the reset command, which puts the SD card into the SPI mode if executed when the CS line is low. The SD card will respond to the reset command by sending a basic 8-bit response on the MISO line. The structure of this response is shown in Figure 6. The first bit is always a 0, while the other bits specify any errors that may have occured when processing the last message. If the command you sent was successfully received, then you will receive the message $(00000001)_2$.

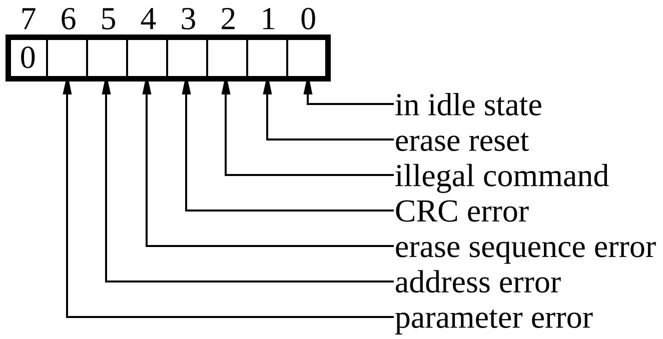


Figure 6: Format of a basic 8-bit response to every command in SPI mode.

To receive this message, your program should continuously toggle the SD_CLK signal and observe the MISO line for data, while keeping the MOSI line high and the CS line low. Your program can detect the message, because every message begins with a 0 bit, and when the SD card sends no data it keeps the MISO line high. Note that the response to each command is sent by the card a few SD_CLK cycles later. If the expected response is not received within 16 clock cycles after sending the reset command, the reset command has to be sent again.

Following a successful reset, test if your system can successfully communicate with the SD card by sending a different command. For example, send one of the following commands, while keeping the CS at value 0:

1. 01 001000 00000000 00000000 00000001 10101010 0000111 1 (CMD8) - this command is only available in the latest cards, compatible with SD card Specifications version 2.00. It is explained in more detail later in the exercise. For most older cards this command should fail and cause the SD card to respond with a message that this command is illegal.

2. 01 111010 00000000 00000000 00000000 00000000 0111010 1 (CMD58) - request the contents of the operating conditions register for the connected card.

A response to these commands consists of 40 bits, where the first 8 bits are identical to the basic 8-bit response, while the remaining 32 bits contain specific information about the SD card. Although the actual contents of the remainning 32 bits are not important for this part of the exercise, a valid response indicates that your command was transmitted and processed successfully. If successful, the first 8 bits of the response will be either 00000001 or 00000101 depending on the version of your SD card.

Write a C-language program that allows a user to input a 48-bit command on the keyboard and send it to the SD card via SPI. Your program should observe the MISO line for at least 16 clock cycles after each command is sent. If during this time the SD card starts sending a response, your program should continue to toggle the SD_CLK line until all bits in the response have been received, and then for another 8 cycles. It should then print out the response in the terminal window. Remember to access the memory-mapped SPI interface using a volatile pointer in C code to ensure that the SPI interface is read properly, as discussed for the example code in Figure 5.

Part III

In this part, you will implement a complete SD card initialization routine to allow you to exchange data with the SD card. To initialize the card correctly, you will have to send several commands in a sequence and process the responses received. Once you complete this procedure, you will be able to access the data stored on the SD card.

The SD card protocol supports many commands. In this exercise we will use only a few commands, which are listed in Table 2. Reserved bits in the argument field will be ingored by the SD card.

Table 2: Detailed Command Description.

Command Number	**Name**	**Argument**	**Width of Response**	**Command Description**
CMD0	GO_ IDLE_ STATE	[31:0]reserved bits	8	Resets the SD card.
CMD8	SEND_IF_ COND	[31:12]reserved bits [11:8]supply voltage [7:0]check pattern	40	Sends SD Card interface a condition and asks the card whether it supports the voltage specified in the argument.
CMD17	READ_ SINGLE_ BLOCK	[31:0]data address	8	Reads a block of memory selected by the data address.
CMD55	APP_CMD	[31:0]reserved bits	8	Informs the card that the next command is an application specific command rather than a standard command.
CMD58	READ_OCR	[31:0]reserved bits	40	Reads the OCR register of the card. CCS bit is assigned to OCR[30].
ACMD41	SD_SEND_ OP_COND	[31]resetved bit [30]HCS [29:0]reserved bits	8	Sends host capacity support (HCS) information and activates the card's initialization process.

The steps necessary to complete the SD card initialization are shown in the flowchart in Figure 7. The flowchart consists of boxes in which a command number is specified. Starting at the top of the flowchart, each command has to be sent to the SD card, and a response has to be received. In some cases, a response from the card needs to be processed to decide the next course of action, as indicated by the diamond-shaped decision boxes.

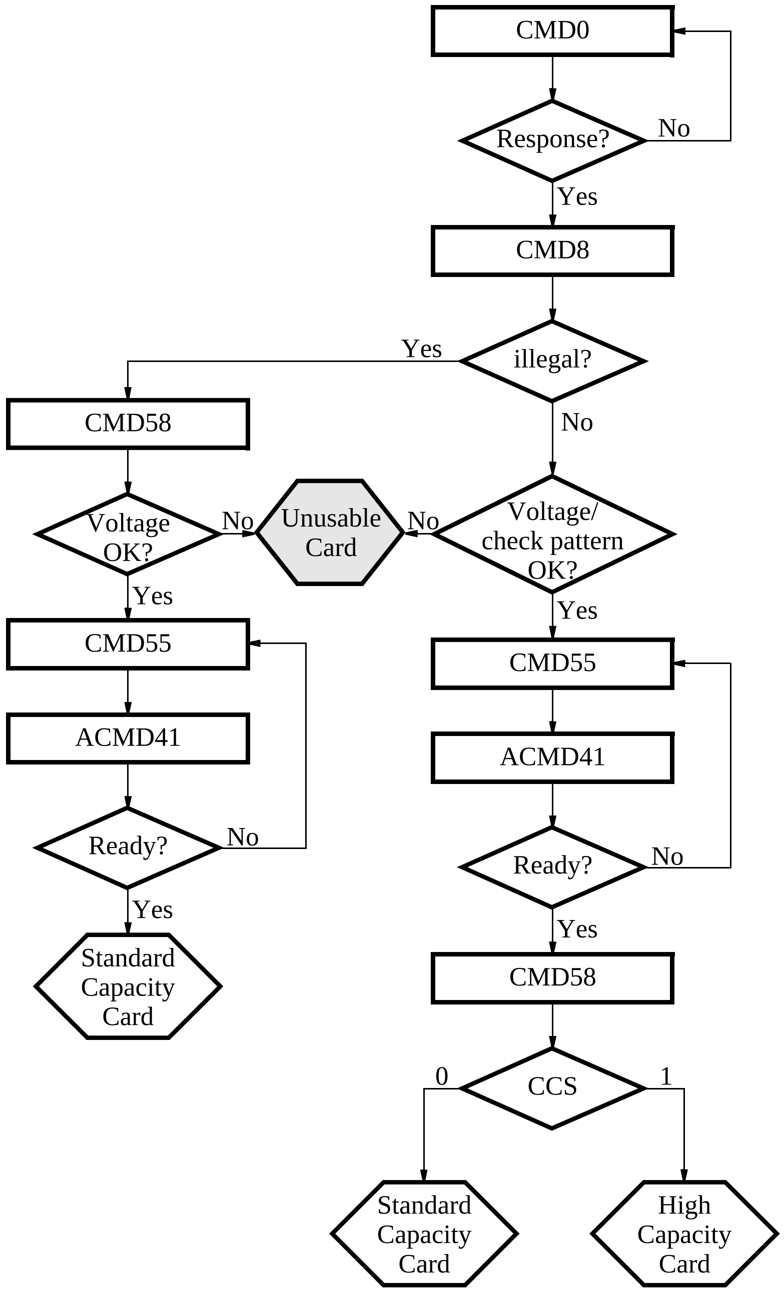


Figure 7: Flowchart representing the SD card initialization routine in SPI mode.

To begin initialization, your program should first set the SD card into the SPI mode as in Part II. That is, it should set the MOSI and CS lines to logic 1 and toggle the SD_CLK for at least 74 cycles. Next, your program should follow the initialization procedure outlined in Figure 7, while keeping the CS line low at all times.

The first command to send is the reset command, CMD0, which together with the CS set to 0 places the SD card into SPI mode. The SD card should respond to this command with the 8-bit message $(00000001)_2$ indicating no errors. However, should the SD card fail to respond, or respond with an error, another reset command should be sent until the card is reset properly.

The second command, CMD8, determines if the SD card is compliant with version 2.00 of the SD card specifications. To perform this step successfully, send the 48-bit command CMD8:

01 001000 00000000 00000000 00000001 10101010 0000111 1

Its argument field specifies the valid supply voltage range that the DE-series board supports. The last group of 8 bits of the argument are called a check pattern, and have been set to an alternating pattern of 0s and 1s. Following the command argument, the CRC code is specified. The CRC code for this command must be correct or the command will be ignored.

In reply to CMD8, the SD card will issue a 40-bit response in the format shown in Figure 8. The response indicates how to proceed with the initialization. In particular, if bit 34 is set to 1 then the SD card is an older model and should be initialized following the left branch of the flowchart in Figure 7. If the command is valid and no other errors occured, then verify if the check pattern in the response is 10101010 and that the voltage field is set to 0001. If either of these conditions fails, the SD card will not function correctly and you should stop the initialization procedure. Otherwise, proceed with the initialization steps of version 2.00 compliant SD card, following the right-hand side of the flowchart.

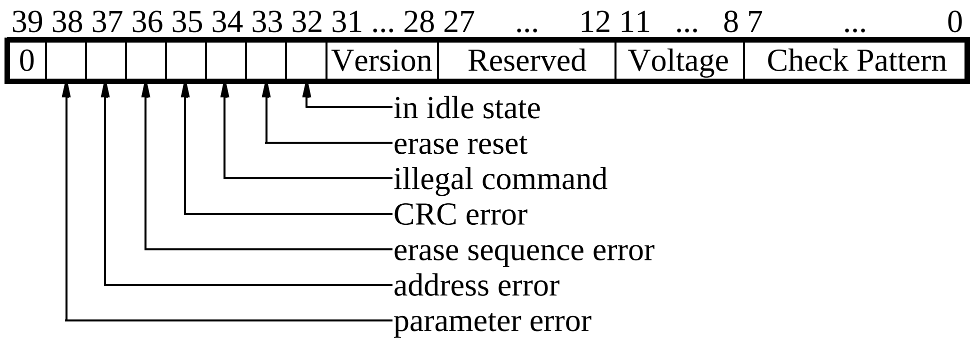


Figure 8: Format of the response to CMD8.

In the case of an older SD card, the next step of the initialization procedure is to send CMD58 to obtain the contents of the Operating Conditions Register (OCR). The OCR indicates the voltage levels the SD card can work with. To work correctly with the DE-series board, the SD card must be able to operate with supply voltage of 3.3V, which is indicated by bit 21 of the response. If this bit is not set to 1, then the card will not work properly with the DE-series board. Otherwise, wait for the card to be ready to exchange data.

To determine if the card is ready for exchange of data, your program should send two commands, CMD55 followed by ACMD41, to the SD card. The argument in each of these commands must be 0. After each command is sent, the SD card will respond. Of particular interest is the response to command ACMD41. The response is a basic 8-bit response as in Figure 6. It is expected that the bit ”in idle state” will remain as 1 for a while. However, the bit will be set to 0 when the card is ready. If it is not, send the pair of commands again and again until the response received is 00000000_2.

Write a program that initializes the SD card as described above. Your program should send the sequence of commands (all with the argument field set to 0, except for CMD8) to initialize the SD card and display the response to each command in the terminal window. When needed, your program should scan the response and proceed with the initialization only if the correct response has been received.

To simplify your task, implement only the initialization for the SD card that you use for this exercise. Consult the manufacturer’s datasheet to determine if your SD card is compliant with version 2.00 of the SD card specifications.

Part IV

In this part, you will use a read command to access data on the SD card.

CMD17 can be used to read a 512-byte segment of data from the card. A standard-capacity SD card takes as argument the byte-wise address of the data to be accessed on the card and returns a block of 512 bytes that begins at the specified address. The command is only valid if the argument is a multiple of 512. Any other address passed as an argument will result in response with the ”address error” bit set.

If a correct address is provided as an argument in CMD17 and the card is able to access the specified memory segment, a response $(00000000)_2$ giving an indication of no errors will be sent by the SD card. Following this response, the card will immediately begin sending the requested data on the MISO line. The data sent by the card should be ignored until a byte comprising $(FE)_{16}$ is received. This byte signifies the beginning of a data packet arranged as shown in Figure 9.

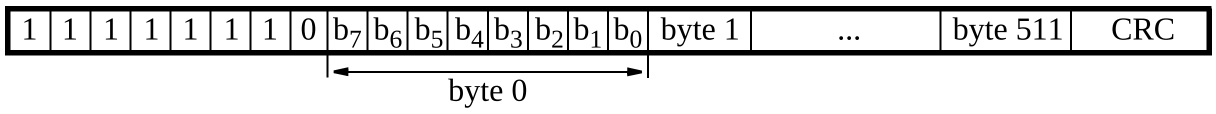


Figure 9: Format of a data packet.

The data packet consists of a leading $(FE)_{16}$ byte, 512 bytes of data, and a 16-bit CRC code. The data is arranged from the least-significant to the most-significant byte. Each byte of the data starts with the most-significant and ends with the least-significant bit.

Write a C-language program that reads any user-specified segment of the SD card and displays it in the terminal window. Your program should ask the user for the segment number (0 to (card size / 512)), send CMD17 with the appropriate address, and read the specified 512-byte segment of data from the SD card. It should then display the data in the terminal window 16 bytes per line in hexdecimal notation .

Addendum

This exercise shows how to access an SD card using the SPI protocol. In most applications, the purpose of using the SD card is to provide access to data.

Usually, files on the SD card are stored with the help of a file system. This allows different systems to access the same set of files. A popular choice is a FAT (File Allocation Table) file system, because it is relatively easy to implement. Although the details of the file system are beyond the scope of this exercise, completing all of the parts of this exercise provides a sufficient basis to read the contents of an SD card. If the card contains a file system, the next step should be to implement routines to read the appropriate sections of the SD card memory to decode the locations and sizes of files stored on it.

Troubleshooting

A secure data card is a sensitive device that may easily malfunction if proper initialization and operating procedures are not followed. To ensure successful completion of this exercise, we recommend that the instructions in each part be followed very carefully. In particular, be aware that:

1. The initialization procedure works best when it follows 74 (or more) SD_CLK cycles during which both the CS and the MOSI lines are set high. Only afterwards should the CS line be set to 0. Failure to do so causes the first few instructions sent to the card to fail.

2. After sending each instruction, exactly 8 SD_CLK cycles with MOSI line set to 1 must be sent to the SD card. A second command should not follow until this condition is met. Otherwise, the SD card may malfunction. It is best to wait for a response after each instruction before sending another command.

3. It is important to wait for the $(FE)_{16}$ byte as the start of the data packet. You may not assume that there is only a leading 0 bit followed by 512 bits of data, as the MISO line is not guaranteed to be set to 1 prior to the start of the data packet.

Preparation

The recommended preparation for this laboratory exercise includes:

1. HDL code, Quartus II, and SOPC Builder project for Part I
2. C code for Parts II through III

Laboratory Exercise 10

Hardware Accelerator

The purpose of this exercise is to learn how to implement hardware acceleration. You will create an application to draw images and perform animation using a DE-series board. The application will draw lines from the center of the screen to its boundary. You will implement the line-drawing in software first, using the Bresenham's line-drawing algorithm. Then, you will accelerate your application (i.e. improve its performance) by implementing the line-drawing algorithm as a hardware circuit. Your accelerator will be controlled by the Nios II processor using an Avalon Slave Interface and will draw lines on the screen via an Avalon Master Interface.

Prerequisite: It is necessary to do Laboratory Exercise 7 to learn how to draw lines on a screen.

Background

Hardware accelerators are circuits designed to offload specific tasks from the processor. They perform functions that run faster in hardware than if executed entirely in software. An example of a system with a hardware accelerator is shown in Figure 1. It is similar to the DE-series Media Computer except that it includes a hardware accelerator designed to implement the line-drawing algorithm.

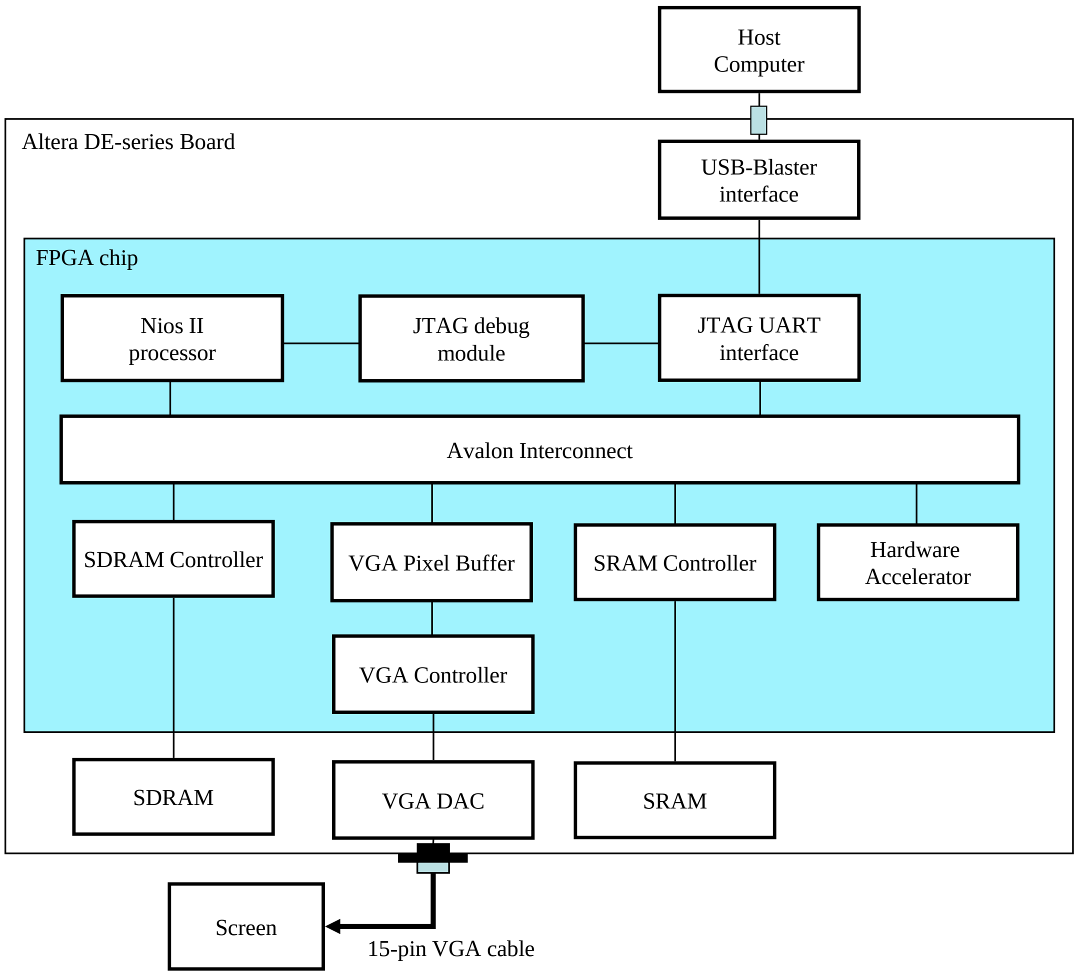


Figure 1: A system with a hardware accelerator.

When an application uses software to draw a line, the processor has to first calculate the location of each pixel on the line and then store it in the SRAM memory. On the other hand, when the hardware accelerator is used, the processor only needs to give the information about the line to the accelerator and ask it to draw. The processor can then wait for the accelerator to finish or work on other tasks.

The system in Figure 1 can be implemented using the Quartus II SOPC Builder tool. You should read the tutorial *Making SOPC Builder Components* to learn how to implement the hardware accelerator in such a system.

Part I

Write a C-language program that randomly draws lines from the center to the boundary of the screen using the Bresenham's algorithm learned in Laboratory Exercise 7. The program should first clear the screen by setting all pixels to black, and then try to fill the whole screen with the same color by drawing random lines from the center. It should include a **for** loop which will execute the line-drawing algorithm for a large number of iterations (e.g. 10,000 times). After the lines fill the screen, you should change the color and repeat the procedure.

Complete this part of the exercise as follows:

1. Write a C-language function to implement the line-drawing algorithm.
2. Write a program that performs the desired animation, using the line-drawing function.
3. Create a new project in the Altera Monitor Program using the DE-series Media Computer.
4. Change the $-O1$ to $-O0$ under $Additional\ compiler\ flags$ field. This reduces any optimizations during compilation.
5. Download the computer system onto the DE-series board.
6. Compile and run your program.

Part II

In this part of the exercise you will begin designing the hardware accelerator, called the Line-Drawing Algorithm (LDA) peripheral, which is shown in Figure 2. The accelerator includes three components: a LDA circuit, an Avalon Slave Interface, and an Avalon Master Interface. The LDA circuit implements the line-drawing algorithm in hardware. The slave interface is used by the processor to control the LDA peripheral, and the master interface is used by the LDA circuit to draw a pixel at the given location on the screen. Details about these three components will be given later.

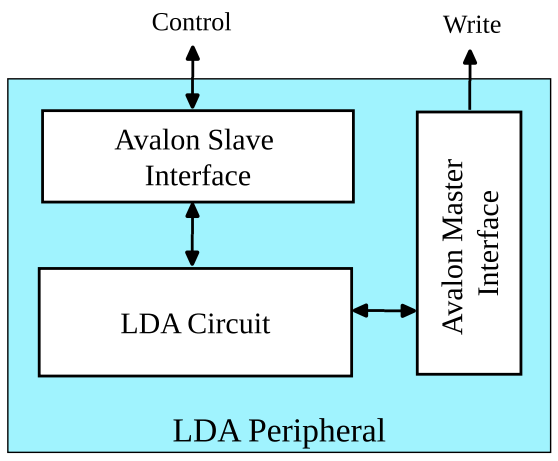


Figure 2: LDA peripheral diagram.

Design the LDA circuit which implements a line-drawing algorithm in hardware. The pseudo-code for the algorithm is shown in Figure 3

```
draw_line(x0, x1, y0, y1)

    boolean is_steep = abs(y1 - y0) > abs(x1 - x0)
    if is_steep then
        swap(x0, y0)
        swap(x1, y1)
    if x0 > x1 then
        swap(x0, x1)
        swap(y0, y1)

    int deltax = x1 - x0
    int deltay = abs(y1 - y0)
    int error = -(deltax / 2)
    int y = y0
    if y0 < y1 then y_step = 1 else y_step = -1

    for x from x0 to x1
        if is_steep then draw_pixel(y,x) else draw_pixel(x,y)
        error = error + deltay
        if error >= 0 then
            y = y + y_step
            error = error - deltax
```

Figure 3: Pseudo-code for a line-drawing algorithm.

The circuit should have the following inputs and outputs, as shown in Figure 4:

- clk – clock signal.
- $resetn$ – active-low reset signal.
- $X0$ – 9-bit input signal, specifying the x coordinate of the starting point.
- $Y0$ – 8-bit input signal, specifying the y coordinate of the starting point.
- $X1$ – 9-bit input signal, specifying the x coordinate of the end point.
- $Y1$ – 8-bit input signal, specifying the y coordinate of the end point.
- $Color$ – 16-bit inout signal, specifying the color of the line to be drawn. Its value is unchanged in the LDA circuit.
- Go – 1-bit input signal, triggers the calculations of pixel locations of the line. This signal is raised high only when all the signals above have already been asserted.
- $Write\_Finish$ – 1-bit input signal to acknowledge that the pixel has been drawn, prompting the LDA circuit to calculate the next pixel location.

- *Pixel_Address* – 32-bit output signal, indicating the pixel location (i.e. SRAM address that stores the pixel).
- *Draw* – 1-bit output signal, specifying that the *Pixel_Address* signal is valid and the pixel at that address should be drawn. After the *Draw* signal is asserted, the LDA circuit should stall until a *Write_Finish* signal is received.
- *Done* – 1-bit output signal, indicating that all pixels in the line have been drawn, and the LDA circuit is ready to draw another line.

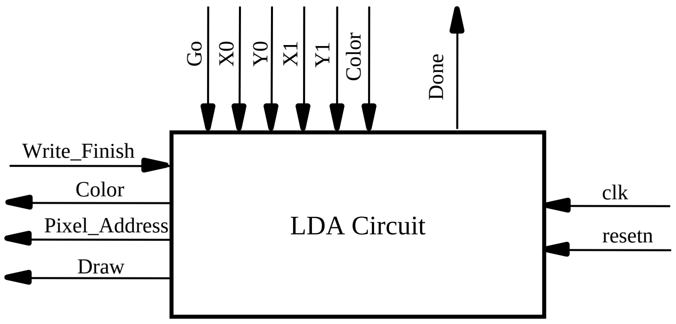
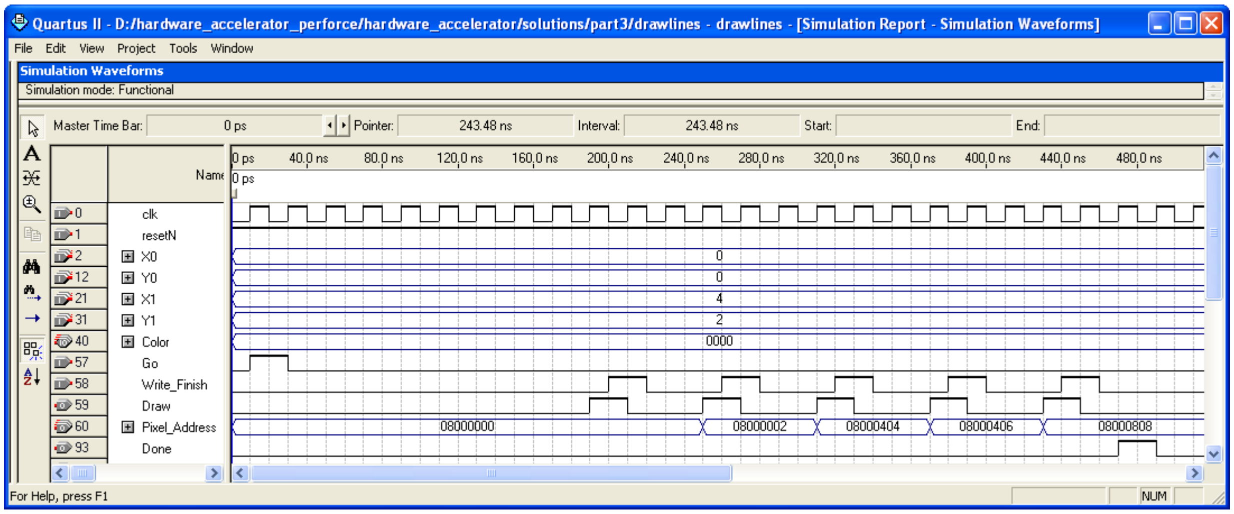


Figure 4: LDA circuit inputs and outputs.

Design and implement the LDA module by completing the following:

1. Create a new Quartus II project for this exercise.
2. Create the Verilog code for the LDA module, include it in your project, and compile the circuit.
3. Use functional simulation to verify that your circuit is correct. An example of the output produced by the functional simulation for a correctly designed circuit is given in Figure 5. The figure describes the sequence of drawing a line from (0,0) to (4,2).


Quartus II - D:/hardware_accelerator_perforce/hardware_accelerator/solutions/part3/drawlines - drawlines - [Simulation Report - Simulation Waveforms]

Simulation Waveforms
Simulation mode: Functional

Master Time Bar: 0 ps Pointer: 243.48 ns Interval: 243.48 ns Start: End:

File Edit View Project Tools Window
0 ps 40.0 ns 80.0 ns 120.0 ns 160.0 ns 200.0 ns 240.0 ns 280.0 ns 320.0 ns 360.0 ns 400.0 ns 440.0 ns 480.0 ns
Name
0 clk
1 resetN
2 X0 0
12 Y0 0
21 X1 4
31 Y1 2
40 Color 0000
57 Go
58 Write_Finish
59 Draw
60 Pixel_Address 08000000 08000002 08000404 08000406 08000808
93 Done
For Help, press F1
NUM

Figure 5: Simulation of the LDA circuit.

Part III

In this part, you will enhance your LDA circuit by adding an Avalon Slave Interface, so that the LDA peripheral can communicate with the processor. The interface has to provide the following inputs to the LDA circuit: Go signal, the starting point of the line (X_0, Y_0), the end point (X_1, Y_1), and $Color$ signal. After the LDA circuit finishes drawing a line, a $Done$ signal has to be received by the interface.

Avalon Memory-Mapped Slave Interface

The LDA peripheral should implement the Avalon Memory-Mapped Interface by including several registers, which the processor can access to instruct the peripheral to draw lines on the screen. The required registers are shown in Figure 6.

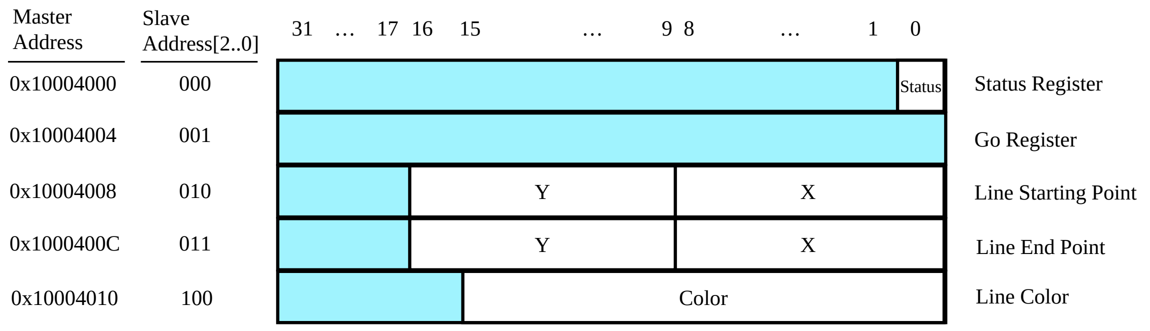


Figure 6: The LDA peripheral memory-mapped registers.

The *Line Starting Point* and the *Line End Point* registers store the *X* and *Y* coordinates for the starting point and the end point of the line, respectively. The *X* coordinate is placed in bits [8..0] and the *Y* coordinate in bits [16..9]. The color is stored in the *Line Color* register, in bits [15..0]. By writing into the *Go* register, the processor initiates the drawing algorithm implemented by the peripheral. Note that the processor must set the *Line Point* registers and the *Color* register before writing to the *Go* register.

After the processor initiates the execution of drawing a line, it should continuously poll bit b_0 of the *Status* register to check whether the LDA peripheral finished drawing the line. The bit b_0 will assume a value of 1 once the line drawing has finished. Only then can the processor instruct the peripheral to draw a new line. The peripheral will not allow (i.e. the command is ignored) the processor to initiate a new drawing operation until it finishes drawing the current line. After the processor has read the value of 1, it has to reset the status register to 0.

Pay attention to the *master address* and the *slave address* in Figure 6. The *master address* is a 32-bit address in Nios II's address space. So, if Nios II wants to write to the *Go* register, it will present the address 0x10004004 to the Avalon Interconnect. This address will then be translated by the Avalon Interconnect into the slave address, which is 001 in binary. The slave address width is only 3 bits because there are only 5 registers, hence 3 bits are enough to encode the offset of each register.

Avalon read/write transfers

A typical Avalon Memory-Mapped interface supports read and write transfers with a slave-controlled *waitrequest*. To begin a transfer, the master asserts $address$, $byteenable$, $read$ or $write$, and $writedata$ signals on the rising edge of the clock. When a slave receives these signals, it captures the data written to it or it outputs the requested data (read). If the slave is unable to respond within one cycle, it can assert $waitrequest$ before the next rising edge of the clock to hold the transfer. When the $waitrequest$ is asserted, the transfer is delayed and the address and control signals are held constant. The slave can stall the interconnect for as many cycles as required by keeping the $waitrequest$ signal high. The transfer will complete on the first rising edge of the clock after the slave deasserts $waitrequest$.

In this part of the exercise, your slave interface does not need to assert the $waitrequest$ as long as it can respond within one cycle. This means that the slave should either present valid data (read transfer) or capture the data (write transfer) before the next cycle. Figure 7 shows two examples of transfers.

1. The master asserts $address$ and $read$ on the rising edge of the clock. In the same cycle the slave decodes the signals from the master and presents valid $readdata$.
2. The master captures $readdata$ on the rising edge of the clock and deasserts the address and control signals. The transfer ends.
3. No control signals are asserted.
4. The master asserts $address$, $write$, and $writedata$ on the rising edge of the clock. The master signals are held constant and the slave decodes them.
5. The slave captures $writedata$ on the rising edge of the clock. The master deasserts the $address$, $writedata$ and control signals. The transfer ends.

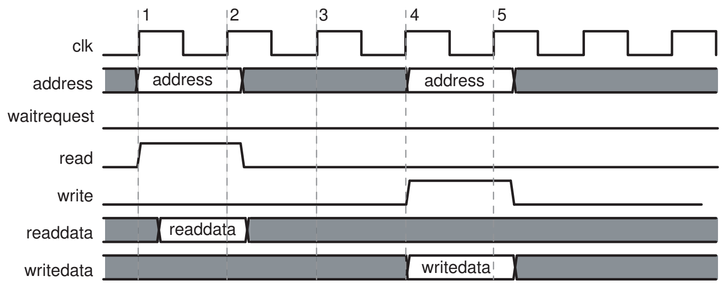


Figure 7: Avalon timing diagrams for read/write transfers (without $waitrequest$) - 1 cycle duration.

Note that the $chipselect$ signal is not shown in Figure 7. This signal is asserted by the Avalon interconnect whenever any transfer is active. Hence, the $chipselect$ signal is asserted along with a *read* or a *write* control signal.

To help you complete this part of the exercise, the system shown in Figure 1 is provided in the starter kit that comes with this exercise. Implement the Avalon slave interface using the Quarts II project provided:

1. Complete the Verilog code for the $LDA\_slave\_interface$ module and copy the LDA circuit you designed in Part II into the project.
2. Use $LEDR_{17-0}$ to display X_0 and X_1, HEX_{3-0} for $Color$, HEX_{5-4} for Y_1, and HEX_{7-6} for Y_0. Connect the Go signal to $LEDG_0$ and connect the $Done$ signal to $LEDG_1$. In order to do that, you should use the conduit signals provided in the top-level file $LDA\_peripheral$.
3. Compile the circuit.
4. Create a new project in Altera Monitor Program using <*Custom System*>. You should include the *.ptf* and *.sof* files generated by Quartus II in $System\ details$.
5. Download the computer system onto the DE-series board.
6. Test the functionality of your peripheral by executing a C-language program and observing the LEDs and 7-segment displays.

Part IV

In this part, you will complete your accelerator by adding the Avalon Master Interface to be used for transferring data between the LDA circuit and the SRAM Controller. The master interface is supposed to receive the $Pixel\_Address$ and $Color$ signals from the LDA circuit when the $Draw$ signal is asserted. It should then put the $Pixel\_Address$ on the $address$ lines and the $Color$ information on the $writedata$ lines in the Avalon interconnect. This data will get passed on to the SRAM. After the drawing is finished, the master interface will assert the $Write\_Finish$ signal to let the LDA circuit know that a new pixel can be drawn. Refer to Figure 2 for the block diagram of the LDA peripheral.

When designing the master interface, keep in mind that the $waitrequest$ signal may be asserted by the SRAM Controller. In such a case, the master interface for the LDA peripheral should maintain all of its control signals in subsequent clock cycles until the $waitrequest$ is deasserted. Notice that a $byteenable$ signal is associated with the master interface. This signal enables writing to specific bytes during transfers. Each bit in $byteenable$ corresponds to a byte in $writedata$. In our case of data transfer between the LDA peripheral and the SRAM, both bytes are written. Therefore, the $byteenable$ value should remain $(11)_2$ all the time.

Figure 8 shows examples of variable duration transfers. Below is the explanation for a write transfer that is 3 cycles in duration. The slave prolongs the transfer for 2 extra cycles by using the $waitrequest$ signal.

1. The master asserts $address$, $write$ and $writedata$ on the rising edge of the clock. In the same cycle, the slave decodes the signals from the master and asserts $waitrequest$, which stalls the transfer.

2. The master samples the asserted $waitrequest$ on the rising edge of the clock. Thus, the $address$, $write$ and $writedata$ signals remain constant.

3. The slave deasserts $waitrequest$ on the rising edge of the clock.

4. The slave captures $writedata$ on the rising edge of the clock. The master samples the deasserted $waitrequest$ and ends the transfer by deasserting all signals.

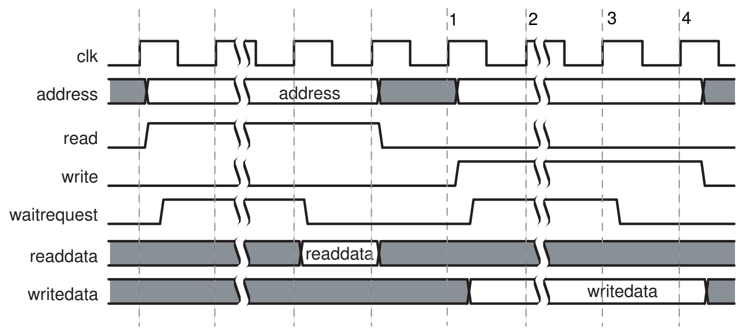


Figure 8: Avalon timing diagram for read/write transfers (with $waitrequest$) - variable duration.

Finish the LDA peripheral that has a LDA circuit, an Avalon slave interface, and an Avalon master interface. Complete the following:

1. Complete the Verilog code of $LDA\_master\_interface$ and compile the Quartus II project.

2. Modify the line-drawing function in part II so that the line is drawn using the LDA peripheral.

3. Download the computer system onto the DE-series board.

4. Compile and run your program.

5. Compare the speed of the animation obtained in Parts I and IV.

Preparation

The recommended preparation for this laboratory exercise includes:

1. C code for Part I
2. Verilog code for Parts II, III, and IV