1 Introduction

This document describes a basic computer system that can be implemented on the Altera DE2 development and education board. This system, called the DE2 Basic Computer, is intended to be used as a platform for introductory experiments in computer organization and embedded systems. To support these beginning experiments, the system contains only a few components: a processor, memory, and some simple I/O peripherals. The FPGA programming file that implements this system, as well as its design source files, can be obtained from the University Program section of Altera's website.

2 DE2 Basic Computer Contents

A block diagram of the DE2 Basic Computer is shown in Figure 1. Its main components include the Altera Nios II processor, memory for program and data storage, parallel ports connected to switches and lights, a timer module, and a serial port. As shown in the figure, the processor and its interfaces to I/O devices are implemented inside the Cyclone® II FPGA chip on the DE2 board. Each of the components shown in Figure 1 is described below.

2.1 Nios II Processor

The Altera Nios® II processor is a 32-bit CPU that can be instantiated in an Altera FPGA chip. Three versions of the Nios II processor are available, designated economy (/e), standard (/s), and fast (/f). The DE2 Basic Computer includes the Nios II/s version, which has an appropriate feature set for use in introductory experiments.

An overview of the Nios II processor can be found in the document Introduction to the Altera Nios II Processor, which is provided in the University Program section of Altera's website. An easy way to implement programs for the Nios II processor is to make use of a utility called the Altera Monitor Program. This utility provides an easy way to assemble and compile programs for Nios II that are written in either assembly language or the C programming language. The Monitor Program can be downloaded from Altera's website; it is an application program that runs on the host computer connected to the DE2 board. The Monitor Program can be used to control the execution of code on Nios II, list (and edit) the contents of processor registers, display the contents of memory on the DE2 board, and similar operations. An overview of the Monitor Program is available in the document Altera Monitor Program Tutorial, which is provided in Altera's University Program website. This website also provides tutorials that describe assembly language and C programming for the Nios II processor on the DE2 board.
As indicated in Figure 1, the Nios II processor can be reset by pressing KEY$_0$ on the DE2 board. The address of the reset vector is given in section 3. All of the I/O peripherals in the DE2 Basic Computer are accessible by the processor as memory mapped devices, using the address ranges that are given in the following subsections.

### 2.2 SRAM Controller

The SRAM Controller provides a 32-bit interface to the static RAM (SRAM) chip on the DE2 Board. This SRAM chip is organized as 256K x 16 bits, but is accessible by the Nios II processor using word (32-bit), halfword (16-bit), or byte operations. The SRAM memory is mapped to the address space 0x08000000 to 0x0807FFFF.

### 2.3 On-Chip Memory

The DE2 Basic Computer includes a 32-Kbyte memory that is implemented in the Cyclone II FPGA chip. This memory is organized as 8K x 32 bits, and can be accessed using either word, halfword, or byte operations. The memory spans addresses in the range 0x09000000 to 0x09007FFF.
2.4 Parallel Ports

The DE2 Basic Computer includes several parallel ports that support input, output, and bidirectional transfers of data between the Nios II processor and I/O peripherals. As illustrated in Figure 2, each parallel port is assigned a Base address and contains up to four 32-bit registers. Ports that have output capability include a writable Data register, and ports with input capability have a readable Data register. Bidirectional parallel ports also include a Direction register that has the same bit-width as the Data register. Each bit in the Data register can be configured as an input by setting the corresponding bit in the Direction register to 0, or as an output by setting this bit position to 1. The Direction register is assigned the address Base + 4.

<table>
<thead>
<tr>
<th>Address</th>
<th>31</th>
<th>30</th>
<th>...</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>Data register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Input or output data bits</td>
</tr>
<tr>
<td>Base + 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Direction bits</td>
</tr>
<tr>
<td>Base + 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Mask bits</td>
</tr>
<tr>
<td>Base + C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Edge capture</td>
</tr>
</tbody>
</table>

Figure 2. Parallel port registers in the DE2 Basic Computer.

The parallel ports in the DE2 basic computer with interrupt capability include an Interruptmask register. This register has the address Base + 8 and specifies whether or not an interrupt signal should be sent to the Nios II processor when the data present at an input port changes value. Setting a bit location in this register to 1 allows interrupts to be generated, while setting the bit to 0 prevents interrupts. Finally, the parallel port may contain an Edgecapture register at address Base + C. Each bit in this register has the value 1 if the corresponding bit location in the parallel port has changed its value from 0 to 1 since it was last read. Performing a write operation to the Edgecapture register sets all bits in the register to 0, and clears any associated Nios II interrupts. Interrupts are discussed further in section 3.

2.4.1 Red and Green LED Parallel Ports

The red lights LEDR\textsubscript{17−0} and green lights LEDG\textsubscript{8−0} on the DE2 board are each driven by an output parallel port, as illustrated in Figure 3. The port connected to LEDR contains an 18-bit write-only Data register, which has the address 0x10000000. The port associated with LEDG has a nine-bit Data register that is mapped to address 0x10000010. These two registers can be written using word accesses, with the upper bits not used in the registers being ignored.
### 2.4.2 7-Segment Displays Parallel Port

There are two parallel ports connected to the 7-segment displays on the DE2 board, each of which comprises a 32-bit write-only Data register. As indicated in Figure 4, the register at address 0x10000020 drives digits HEX3 to HEX0, and the register at address 0x10000030 drives digits HEX7 to HEX4. Data can be written into these two registers by using word operations. This data directly controls the active-low segments of each display, according to the bit locations given in Figure 4.

![Figure 3. Output parallel port for LEDR and LEDG.](image)

<table>
<thead>
<tr>
<th>Address</th>
<th>Data register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10000000</td>
<td></td>
</tr>
<tr>
<td>31 18 17 ... 0</td>
<td>LEDR&lt;sub&gt;17&lt;/sub&gt;</td>
</tr>
<tr>
<td>0x10000010</td>
<td></td>
</tr>
<tr>
<td>31 9 8 ... 0</td>
<td>LEDG&lt;sub&gt;8&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

![Figure 4. Bit locations for the 7-segment displays parallel ports.](image)

<table>
<thead>
<tr>
<th>Address</th>
<th>Data register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10000020</td>
<td></td>
</tr>
<tr>
<td>31 30 24 23 22 16 15 14 8 7 6 0</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>HEX&lt;sub&gt;3&lt;/sub&gt;&lt;sub&gt;6-0&lt;/sub&gt;</td>
</tr>
<tr>
<td>0x10000030</td>
<td></td>
</tr>
<tr>
<td>31 30 24 23 22 16 15 14 8 7 6 0</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>HEX&lt;sub&gt;7&lt;/sub&gt;&lt;sub&gt;6-0&lt;/sub&gt;</td>
</tr>
</tbody>
</table>
2.4.3 Slider Switch Parallel Port

The SW\textsubscript{17−0} slider switches on the DE2 board are connected to an input parallel port. As illustrated in Figure 5, this port comprises an 18-bit read-only \textit{Data} register, which is mapped to address 0x10000040.

![Figure 5. Data register in the slider switch parallel port.](image)

2.4.4 Pushbutton Parallel Port

The parallel port connected to the KEY\textsubscript{3−0} pushbutton switches on the DE2 board comprises three 4-bit registers, as shown in Figure 6. These registers have addresses 0x10000050 to 0x1000005F and can be accessed using word operations. The read-only \textit{Data} register provides the values of the switches KEY\textsubscript{3}, KEY\textsubscript{2} and KEY\textsubscript{1}. Bit 0 of the data register is not used, because, as discussed in section 2.1, the corresponding switch KEY\textsubscript{0} is reserved for use as a reset mechanism for the Nios II processor. The \textit{Interruptmask} register allows processor interrupts to be generated when a key is pressed. Each bit in the \textit{Edgecapture} register is set to 1 by the parallel port when the corresponding key is pressed. The Nios II processor can read this register to determine which key has been pressed, in addition to receiving an interrupt request if the corresponding bit in the interrupt mask register is set to 1. Writing any value to the \textit{Edgecapture} register deasserts the Nios II interrupt request and sets all bits of the \textit{Edgecapture} register to zero. Interrupts are discussed further in section 3.

![Figure 6. Registers used in the pushbutton parallel port.](image)
2.4.5 Expansion Parallel Ports

The DE2 Basic Computer includes two bidirectional parallel ports that are connected to the JP1 and JP2 expansion headers on the DE2 board. Each of these parallel ports includes the four 32-bit registers that were described previously for Figure 2. The base addresses of the ports connected to JP1 and JP2 are 0x10000060 and 0x10000070, respectively. Figure 7 gives a diagram of the JP1 and JP2 expansion connectors on the DE2 board, and shows how the respective parallel port Data register bits, \( D_{31-0} \), are assigned to the pins on the connector. The figure shows that bit \( D_0 \) of the parallel port for JP1 is assigned to the pin at the top left corner of the connector, bit \( D_1 \) is assigned to the top right, and so on. Note that some pins on JP1 and JP2 provide only power and ground voltage levels, and are therefore not used by the parallel ports. Also, only 32 of the 36 data pins that appear on each connector can be used.

![Figure 7. Assignment of parallel port bits to pins on JP1 and JP2.](image)

2.5 JTAG Port

The JTAG port implements a communication link between the DE2 board and its host computer. This link is automatically used by the Quartus II software to transfer FPGA programming files into the DE2 board, and by the Altera Monitor Program. The JTAG port also includes a UART, which can be used to transfer character data between the host computer and programs that are executing on the Nios II processor. If the Altera Monitor Program is used on the host computer, then this character data is sent and received through its Terminal Window. The Nios II programming interface of the JTAG UART consists of two 32-bit registers, as shown in Figure 8. The register mapped to address 0x10001000 is called the Data register and the register mapped to address 0x10001004 is called the Control register.
When character data is received by the JTAG UART from the host computer it is stored in a 64-character FIFO. The number of characters currently stored in this FIFO is indicated in the field RAVAIL, which are bits 31–16 of the Data register. If the receive FIFO overflows, then additional data is lost. When data is present in the receive FIFO, then the value of RAVAIL will be greater than 0 and the value of bit 15, RVALID, will be 1. Reading the character at the head of the FIFO, which is provided in bits 7–0, decrements the value of RAVAIL by one and returns this decremented value as part of the read operation. If no data is present in the receive FIFO, then RVALID will be set to 0 and the data in bits 7–0 is undefined.

The JTAG UART also includes a 64-character FIFO that stores data waiting to be transmitted to the host computer. Character data is loaded into this FIFO by performing a write to bits 7–0 of the Data register in Figure 8. Note that writing into this register has no effect on received data. The amount of space, WSPACE, currently available in the transmit FIFO is provided in bits 31–16 of the Control register. If the transmit FIFO is full, then any characters written to the Data register will be lost.

The RE and WE bits in the control register are used to enable processor interrupts associated with the receive and transmit FIFOs. When enabled, interrupts are generated when RAVAIL for the receive FIFO, or WSPACE for the transmit FIFO, exceeds 7. Pending interrupts are indicated in the control register’s RI and WI bits, and can be cleared by writing or reading data to/from the JTAG UART. Interrupts are discussed further in section 3.

Bit 10 in the control register, called AC, has the value 1 if the JTAG UART has been accessed by the host computer. This bit can be used to check if a working connection to the host computer has been established. The AC bit can be cleared to 0 by writing a 1 into it.

2.6 Serial Port

The serial port in the DE2 Basic Computer implements a UART that is connected to an RS 232 chip on the DE2 Board. This UART is configured for 8-bit data, one stop bit, odd parity, and operates at a baud rate of 115,200. The serial port’s programming interface consists of two 32-bit registers, as illustrated in Figure 9. The register at address 0x10001010 is referred to as the Data register, and the register at address 0x10001014 is called the control register.

When character data is received from the RS 232 chip it is stored in a 256-character FIFO in the UART. As illustrated in Figure 9, the number of characters RAVAIL currently stored in this FIFO is provided in bits 31–16 of the Data register. If the receive FIFO overflows, then additional data is lost. The character at the head of the FIFO can be accessed by reading bits 7–0 of the Data register. Bit 9 indicates whether or not the data has a parity error. Performing a halfword read operation on
Figure 9. Serial port registers.

bits 31–16 of the Data register does not affect the value of RAVAIL, but if RAVAIL is greater than 1, then reading bits 15–0 decrements RAVAIL by one.

The UART also includes a 256-character FIFO that stores data waiting to be sent to the RS 232 chip. Character data is loaded into this register by performing a write to bits 7–0 of the Data register in Figure 9. Note that writing into this register has no effect on received data. The amount of space WSPACE currently available in the transmit FIFO is provided in bits 31–16 of the Control register, as indicated in Figure 9. If the transmit FIFO is full, then any additional characters written to the Data register will be lost. The RE and WE bits in the control register are used to enable processor interrupts associated with the receive and transmit FIFOs. When enabled, interrupts are generated when RAVAIL for the receive FIFO, or WSPACE for the transmit FIFO, exceeds 31. Pending interrupts are indicated in the control register’s RI and WI bits, and can be cleared by writing or reading data to/from the UART. Interrupts are discussed further in section 3.

2.7 Interval Timer

The DE2 Basic Computer includes a timer that can be used to measure various time intervals. The interval timer is loaded with a preset value, and then counts down to zero using the 50-MHz clock signal provided on the DE2 board. The programming interface for the timer includes six 16-bit registers, as illustrated in Figure 10. The 16-bit register at address 0x10002000 provides status information about the timer, and the register at address 0x10002004 allows control settings to be made. The bit fields in these registers are described below:

- **b0 (TO)** provides a timeout signal which is set to 1 by the timer when it has reached a count value of zero. The TO bit can be reset by writing a 0 into it.

- **b1 (RUN)** is set to 1 by the timer whenever it is currently counting. Write operations to the status halfword do not affect the value of the RUN bit.

- **b16 (ITO)** enables processor interrupts. Writing a 1 to ITO allows an interrupt request to be generated whenever TO becomes 1. Interrupts are discussed in more detail in section 3.

- **b17 (CONT)** affects the continuous operation of the timer. When the timer reaches a count value of zero it automatically reloads the specified starting count value. If CONT is set to 1, then the timer will continue counting down automatically. But if CONT = 0, then the timer will stop after it has reached a count value of 0.

- **b18/b19 (START/STOP)** can be used to commence/suspend the operation of the timer by writing a 1 into the respective bit.
The two 16-bit registers at addresses 0x10002008 and 0x1000200C allow the period of the timer to be changed by setting the starting count value. The default setting provided in the DE2 Basic Computer gives a timer period of 125 msec. To achieve this period, the starting value of the count is 50 MHz \times 125 \text{ msec} = 6.25 \times 10^6$. It is possible to capture a snapshot of the counter value at any time by performing a write to address 0x10002010. This write operation causes the current 32-bit counter value to be stored into the two 16-bit timer registers at addresses 0x10002010 and 0x10002014. These registers can then be read to obtain the count value.

### 2.8 System ID

The system ID module provides a unique value that identifies the basic computer system. The host computer connected to the DE2 board can query the system ID module by performing a read operation through the JTAG port. The host computer can then check the value of the returned identifier to confirm that the DE2 Basic Computer has been properly downloaded into the DE2 board. This process allows debugging tools on the host computer, such as the Altera Monitor Program, to verify that the DE2 board contains the required computer system before attempting to execute code that has been compiled for this system.

### 3 Exceptions and Interrupts

The reset address of the Nios II processor in the DE2 Basic Computer is set to 0x09000000. The address used for all other general exceptions, such as divide by zero, and hardware IRQ interrupts is 0x900020. Since the Nios II processor uses the same address for general exceptions and hardware IRQ interrupts, the Exception Handler software must determine the source of the exception by examining the appropriate processor status register. Table 1 gives the assignment of IRQ numbers to each of the I/O peripherals in the DE2 Basic Computer.
4 Modifying the DE2 Basic Computer

It is possible to modify the DE2 Basic Computer by using Altera’s Quartus II software and SOPC Builder tool. Tutorials that introduce this software are provided in the University Program section of Altera’s website. To modify the system it is first necessary to obtain all of the relevant design source code files. The DE2 Basic Computer is available in two versions that specify the system using either Verilog HDL or VHDL. After these files have been obtained it is also necessary to install the source code for the I/O peripherals in the system. These peripherals are included in a package called the Altera University Program IP Cores, which is also available from Altera’s University Program.

Table 2 lists the names of the SOPC Builder IP cores that are used in this system. When the DE2 Basic Computer design files are opened in the Quartus II software, these cores can be examined using the SOPC Builder tool. Each core has a number of settings that are selectable in the SOPC Builder tool, and includes a datasheet that provides detailed documentation.

<table>
<thead>
<tr>
<th>I/O Peripheral</th>
<th>SOPC Builder Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>SRAM</td>
</tr>
<tr>
<td>On-chip Memory</td>
<td>On-Chip Memory (RAM or ROM)</td>
</tr>
<tr>
<td>Red LED parallel port</td>
<td>Parallel Port</td>
</tr>
<tr>
<td>Green LED parallel port</td>
<td>Parallel Port</td>
</tr>
<tr>
<td>7-segment displays parallel port</td>
<td>Parallel Port</td>
</tr>
<tr>
<td>Expansion parallel ports</td>
<td>Parallel Port</td>
</tr>
<tr>
<td>Slider switch parallel port</td>
<td>Parallel Port</td>
</tr>
<tr>
<td>Pushbutton parallel port</td>
<td>Parallel Port</td>
</tr>
<tr>
<td>JTAG port</td>
<td>JTAG UART</td>
</tr>
<tr>
<td>Serial port</td>
<td>RS232 UART</td>
</tr>
<tr>
<td>Interval timer</td>
<td>Interval timer</td>
</tr>
<tr>
<td>System ID</td>
<td>System ID Peripheral</td>
</tr>
</tbody>
</table>

Table 2. SOPC Builder core used in the DE2 Basic Computer.

The steps needed to modify the system are:
1. Copy the design source files for the DE2 Basic Computer from Altera's University Program website. These files can be found in the *Laboratory and Tutorials* section of the website.

2. Install the *University Program IP Cores*.

3. Open the *DE2_Basic_Computer.qpf* project in the Quartus II software.

4. Open the SOPC Builder tool, and modify the system as desired.

5. Generate the modified system by using the SOPC Builder tool.

6. It may be necessary to modify the Verilog or VHDL code in the top-level module, *DE2_Basic_System.v/vhd*, if any I/O peripherals have been added or removed from the system.

7. Compile the project in the Quartus II software.

8. Download the modified system into the DE2 board.

5 Making the System the Default Configuration

The DE2 Basic Computer can be loaded into the nonvolatile FPGA configuration memory on the DE2 board, so that it becomes the default system whenever the board is powered on. Instructions for configuring the DE2 board in this manner can be found in the tutorial *Introduction to the Quartus II Software*, which is available from Altera’s University Program.

Appendix A Writing Assembly Language Programs

The DE2 Basic Computer provides a convenient platform for experimenting with Nios II assembly language code. A simple example of such code is provided in Figures 11 to 13. This code is meant to serve as an example that illustrates how I/O peripherals, memory, and interrupts can be used in the DE2 Basic Computer.

Figure 11 represents the main program for our example. The code in parts \(a\) and \(b\) of the figure performs some initializations, enables interrupts, and then enters an infinite loop which reads the values on the slider switch parallel port and shows them on the red LEDs parallel port. The switch values \(SW_{15-0}\) are also decoded by the subroutine in parts \(b\) and \(c\) of the figure and displayed on four 7-segment displays as digits 0 – 9. Depending on which pushbutton keys have been pressed, the decoded digits may appear on either \(HEX_3-HEX_0\) or \(HEX_7-HEX_4\), as described below. Figure 11\(d\) declares the memory variables that are used in the program.

Figure 12 gives the code that is executed when the Nios II processor is reset, or an exception occurs. For a reset operation the code simply branches to the start of the main program, in Figure 11. In the case of an exception, the code first checks for the type of exception that has occurred. If the exception is caused by an interrupt from the pushbutton switches in the DE2 Basic Computer, then the interrupt service routine shown in Figure 13 is invoked. Interrupts generated by the pushbutton switches have the following effects: \(KEY_1\) causes the slider switch values to be displayed on \(HEX_3-0\), \(KEY_2\) selects \(HEX_7-4\), and \(KEY_3\) clears the 7-segment displays.
/* This program exercises a few features of the DE2 basic computer. It
* 1. reads the parallel port attached to the SW switches and
* 2. assigns the SW values to the red LEDR
* 3. responds to interrupts from pushbuttons KEY1-3
*    KEY1: display digits (0-9) selected by SW values on HEX3-0
*    KEY2: display digits (0-9) selected by SW values on HEX7-4
*    KEY3: clear HEX displays
*/

# initialize HEX3-0 and HEX7-4 displays buffer
movia r16, DISPLAY_BUFFER
movia r17, 0x76543210   # initial display values
stw r17, 0(r16)

# enable pushbutton interrupts
movia r16, PUSHBUTTON_BASE
movi r15, 0b01110       # set the interrupt mask bits to 1 (bit 0 is the Nios II reset)
stwio r15, 8(r16)

# enable processor interrupts
movi r15, 0b010         # enable interrupts for pushbuttons
wrctl ienable, r15
movi r15, 1
wrctl status, r15

Figure 11. Main program for the example assembly language code (Part a).
DO_DISPLAY:
    # load slider switch value to display
    movia r15, SLIDER SWITCH_BASE
    ldwio r16, 0(r15)

    # write to red LEDs
    movia r15, RED_LED_BASE
    stwio r16, 0(r15)

    # store in memory
    movia r15, SW_VALUES
    stwio r16, 0(r15)
    call UPDATE_HEX_DISPLAY  # decode characters and write to displays

    br DO_DISPLAY

//.................................................................................................................................
* Updates the value displayed on the hex display. The value is taken from the
* buffer.
*/
.global UPDATE_HEX_DISPLAY
UPDATE_HEX_DISPLAY:
    subi sp, sp, 36  # reserve space on the stack
    # save registers
    stw ra, 0(sp)
    stw fp, 4(sp)
    stw r15, 8(sp)
    stw r16, 12(sp)
    stw r17, 16(sp)
    stw r18, 20(sp)
    stw r19, 24(sp)
    stw r20, 28(sp)
    stw r21, 32(sp)
    addi fp, sp, 36

    # load hex value to display
    movia r15, DISPLAY BUFFER
    ldw r16, 0(r15)

Figure 11. Main program for the example assembly language code (Part b).
/* Loop to fill the two-word buffer that drives the parallel port on the DE2 basic
* computer connected to the HEX7 to HEX0 displays. The loop produces for each 4-bit
* character in r16 a corresponding 8-bit code for the segments of the displays
*/

movia r17, 7
movia r15, HEX_SEGMENTS
movia r19, SEVEN_SEG_DECODE_TABLE

SEVEN_SEG_DECODER:
mov r18, r16
andi r18, r18, 0x000F
add r20, r19, r18  # index into decode table based on character
add r21, zero, zero
ldb r21, 0(r20)  # r21 <- 7-seg character code
stb r21, 0(r15)  # store 7-seg code into buffer
srli r16, r16, 4
addi r15, r15, 1
subi r17, r17, 1
bge r17, zero, SEVEN_SEG_DECODER

# write parallel port buffer words
movia r15, HEX_SEGMENTS
ldw r16, 0(r15)
movia r17, HEX3_HEX0_BASE
stwio r16, 0(r17)
ldw r16, 4(r15)
movia r17, HEX7_HEX4_BASE
stwio r16, 0(r17)

# restore registers
ldw ra, 0(sp)
ldw fp, 4(sp)
ldw r15, 8(sp)
ldw r16, 12(sp)
ldw r17, 16(sp)
ldw r18, 20(sp)
ldw r19, 24(sp)
ldw r20, 28(sp)
ldw r21, 32(sp)
addi sp, sp, 36  # release the reserved stack space
ret

Figure 11. Main program for the example assembly language code (Part c).
/* DATA SECTION */

.data

.global DISPLAY_BUFFER
DISPLAY_BUFFER:
  # 32-bit buffer to hold 8 characters for the hex displays
  .word 0

.global SW_VALUES
SW_VALUES:
  # 18-bit value read from SW switches
  .word 0

/* SEVEN_SEGMENT_DECODE_TABLE gives the on/off settings for all segments in
* a single 7-seg display in the DE2 basic computer, for the characters
* 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. values above 9 display as 'blank'. The decode table
* uses the segment indexes on the DE2 board 7-seg displays, and the assignment of
* these signals to parallel port pins in the DE2 basic computer.
*/
SEVEN_SEG_DECODE_TABLE:
  .byte 0b00111111, 0b00000110, 0b01011011, 0b01001111
  .byte 0b01100110, 0b01101101, 0b01111101, 0b00000111
  .byte 0b01111111, 0b01100111, 0b00000000, 0b00000000
  .byte 0b00000000, 0b00000000, 0b00000000, 0b00000000

/* HEX_SEGMENTS is used to hold eight 7-segment codes that are displayed on the
* the 8 digits of 7-seg displays in the DE2 basic computer.
*/
HEX_SEGMENTS:
  .fill 2, 4, 0 # reserve two 4-byte quantities (two words)

.end

Figure 11. Main program for the example assembly language code (Part d).
.include "nios Macros.s"

# global variables
.extern DISPLAY_BUFFER
.extern SW VALUES

/******************************************************************************
 * RESET SECTION
 * Note: Nios II assembler/linker places the following "reset" section at the
 * reset address specified in the CPU settings in SOPC Builder.
 ***************************************************************************/
.section .reset, "ax"
    br _start # branch to start function

/******************************************************************************
 * EXCEPTIONS SECTION
 * Note: Nios II assembler/linker automatically places the following "exceptions"
 * section at the exceptions address specified in the CPU settings in SOPC Builder.
 ***************************************************************************/
.global EXCEPTION_HANDLER
EXCEPTION_HANDLER:
    subi sp, sp, 28 # reserve space on the stack
    stw et, 0(sp)

    rdctl et, ctl4
    beq et, r0, SKIP_EA_DEC # Interrupt is not external
    subi ea, ea, 4 /* Must decrement ea by one instruction
                      * for external interrupts, so that the
                      * interrupted instruction will be run */

    SKIP_EA_DEC:
        stw ea, 4(sp) # Save all used registers on the Stack
        stw ra, 8(sp) # needed if call inst is used
        stw fp, 12(sp)
        stw r2, 16(sp)
        stw r3, 20(sp)
        stw r22, 24(sp)
        addi fp, sp, 28

Figure 12. Exception handler for the example assembly language code (Part a).
```assembly
rctl et,ctl4
bne et,r0,CHECK_LEVEL_1  # Interrupt is an external interrupt

NOT_EI: /* Interrupt must be unimplemented instruction or
  br END_ISR * TRAP instruction. This code does not
  * handle those cases. */

CHECK_LEVEL_1: # Pushbutton is interrupt level 1
andi r22,et,0b10
beq r22,r0,END_ISR

movia r22,SW_VALUES  # pass parameters into ISR
ldw r2,0(r22)
movia r22,DISPLAY_BUFFER
ldw r3,0(r22)
call PUSHBUTTON_ISR
movia r22,DISPLAY_BUFFER
stw r3,0(r22)  # stored returned value for display

END_ISR:
ldw et,0(sp)  # Restore all used register to previous values
ldw ea,4(sp)
ldw ra,8(sp)  # needed if call inst is used
ldw fp,12(sp)
ldw r2,16(sp)
ldw r3,20(sp)
ldw r22,24(sp)
addi sp,sp,28

eret
```

Figure 12. Exception handler for the example assembly language code (Part b).
Pushbutton – Interrupt Service Routine

Input parameters: r2, r3
Output parameter: r3

If KEY 1 is pressed then lower halfword of r2 is copied into lower halfword of r3
If KEY 2 is pressed then lower halfword of r2 is copied into upper halfword of r3
If KEY 3 is pressed then r3 is set to FFFFFFFF

Figure 13. Interrupt service routine for the pushbutton switches (Part a).


FIGURE 13. Interrupt service routine for the pushbutton switches (Part b).

Appendix B Writing C Language Programs

In addition to its use for assembly language code, the DE2 Basic Computer also provides an excellent platform for experimenting with C language programs on the Nios II processor. An example C program is shown in Figures 14 to 17. This code performs exactly the same operations that we showed for the assembly language example in Figures 11 to 13. It is meant to serve as a simple example that can be extended to perform other functions.

The C code in Figure 14 gives the main program. It performs initializations, enables interrupts, and then enters an infinite loop that reads the values of the slider switch parallel port and displays them on the red LEDs and four 7-segment displays. To enable interrupts the code uses macros that provide access to the Nios II status and control registers. A collection of such macros, which can be used in any C program, are provided in Figure 15.

Figure 16 shows how the Nios II reset operation and exception processing can be handled in C code. The function called the_reset provides a simple reset mechanism by performing a branch to the main program. The function named the_exception represents a general exception handler that can be used with any C program. It includes assembly language code to check if the exception is caused by an external interrupt, and, if so, calls a C language routine named interrupt_handler. This routine can then perform whatever action is needed for the specific application. In Figure 16, the interrupt_handler code first determines which exception has occurred, by using a macro from Figure 15 that reads the content of the Nios II interrupt pending register. The interrupt service routine that is invoked for the pushbutton switches appears in Figure 17.
#include "nios2_ctrl_reg_macros.h"

#define RED_LED_BASE 0x10000000
#define HEX3_HEX0_BASE 0x10000020
#define HEX7_HEX4_BASE 0x10000030
#define SLIDER_SWITCH_BASE 0x10000040
#define PUSHBUTTON_BASE 0x10000050

/* function prototypes */
void Update_HEX_Display(int);

/* global variables */
int display_buffer = 0;
int sw_values = 0;
char seven_seg_decode_table[] = { 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07,
                                0x7F, 0x67, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
char hex_segments[] = { 0, 0, 0, 0, 0, 0, 0, 0 };

/************************************************************
* Main program: this program exercises a few features of the DE2 basic computer. *
* It performs the following: *
* 1. reads the parallel port attached to the SW switches and *
* 2. assigns the SW values to the red LEDR *
* 3. responds to interrupts from pushbuttons KEY1-3 *
* KEY1: display digits (0-9) selected by SW values on HEX3-0 *
* KEY2: display digits (0-9) selected by SW values on HEX7-4 *
* KEY3: clear HEX displays *
************************************************************/

int main(void)
{
    /* Declare volatile pointers to I/O registers (volatile means that IO load *
     * and store instructions will be used to access these pointer locations, *
     * instead of regular memory loads and stores) *
    */
    volatile int * pushbutton_ptr = (int *) PUSHBUTTON_BASE;
    volatile int * slider_switch_ptr = (int *) SLIDER_SWITCH_BASE;
    volatile int * red_LED_ptr = (int *) RED_LED_BASE;

    display_buffer = 0x76543210;   // initial digits to display
    *(pushbutton_ptr + 2) = 0xE;    // Set the interrupt mask bits to 1 (bit 0 is the Nios II reset)

    Figure 14. Main program for the example C code (Part a).
NIOS2_WRITE_IENABLE( 0x2 ); // Set Nios II interrupt mask bit for pushbuttons
NIOS2_WRITE_STATUS( 1 ); // Enable Nios II interrupts

while(1)
{
    sw_values = *(slider_switch_ptr); // Read the SW slider switch values
    *(red_LED_ptr) = sw_values; // Light up the red LEDs

    Update_HEX_Display ( display_buffer );
};

return 0;
}

/******************************************************************************************
* Updates the value displayed on the hex display. The value is taken from the            *
* buffer.                                                                        *
******************************************************************************************/

void Update_HEX_Display( int buffer )
{
    volatile int * HEX3_HEX0_ptr = (int *) HEX3_HEX0_BASE;
    volatile int * HEX7_HEX4_ptr = (int *) HEX7_HEX4_BASE;
    int shift_buffer, nibble;
    char code;
    int i;

    shift_buffer = buffer;
    for ( i = 0; i < 8; ++i )
    {
        nibble = shift_buffer & 0x0000000F; // character is in rightmost nibble
        code = seven_seg_decode_table[nibble];
        hex_segments[i] = code;
        shift_buffer = shift_buffer >> 4;
    }

    *(HEX3_HEX0_ptr) = *(int *) hex_segments; // drive the hex displays
    *(HEX7_HEX4_ptr) = *(int *) (hex_segments+4); // drive the hex displays
    return;
}

Figure 14. Main program for the example C code (Part b).
#ifndef __NIOS2_CTRL_REG_MACROS__
#define __NIOS2_CTRL_REG_MACROS__

/***************************************************************************/
/* Macros for accessing the control registers. */
/***************************************************************************/

#define NIOS2_READ_STATUS(dest) \
  do { dest = __builtin_rdctl(0); } while (0)

#define NIOS2_WRITE_STATUS(src) \
  do { __builtin_wrctl(0, src); } while (0)

#define NIOS2_READ_ESTATUS(dest) \
  do { dest = __builtin_rdctl(1); } while (0)

#define NIOS2_READ_BSTATUS(dest) \
  do { dest = __builtin_rdctl(2); } while (0)

#define NIOS2_READ_IENABLE(dest) \
  do { dest = __builtin_rdctl(3); } while (0)

#define NIOS2_WRITE_IENABLE(src) \
  do { __builtin_wrctl(3, src); } while (0)

#define NIOS2_READ_IPENDING(dest) \
  do { dest = __builtin_rdctl(4); } while (0)

#define NIOS2_READ_CPUID(dest) \
  do { dest = __builtin_rdctl(5); } while (0)

#endif

Figure 15. Macros for accessing Nios II status and control registers.
```c
#include "nios2_ctrl_reg_macros.h"

/* function prototypes */
void main(void);
void interrupt_handler(void);
int pushbutton_isr(int, int);

extern int display_buffer; // global variable
extern int sw_values; // global variable

/* The assembly language code below handles CPU reset processing */
void the_reset (void) __attribute__((section (".reset")));
void the_exception (void)

/******************************************************************************
* Reset code. By giving the code a section attribute with the name ".reset" we *
* allow the linker program to locate this code at the proper reset vector address. *
* This code just calls the main program. *
******************************************************************************/
{
    asm (".set noat"); // Magic, for the C compiler
    asm (".set nobreak"); // Magic, for the C compiler
    asm ("br main"); // Call the C language main program
}

/* The assembly language code below handles CPU exception processing. This *
* code should not be modified; instead, the C language code in the function *
* interrupt_handler() can be modified as needed for a given application. */
void the_exception (void) __attribute__((section (".exceptions")));
void the_exception (void)

/******************************************************************************
* Exception code. By giving the code a section attribute with the name ".exceptions" *
* we allow the linker to locate this code at the proper exceptions vector address. *
* This code calls the interrupt handler and later returns from the exception. *
******************************************************************************/
{
    asm (".set noat"); // Magic, for the C compiler
    asm (".set nobreak"); // Magic, for the C compiler
    asm ("subi sp, sp, 128");
    asm ("stw et, 96(sp)");
    asm ("rdctl et, ctl4");
    asm ("beq et, r0, SKIP_EA_DEC"); // Interrupt is not external
    asm ("subi ea, ea, 4"); /* Must decrement ea by one instruction for external
* interrupts, so that the instruction will be run */
}

Figure 16. Exception handler code (Part a).
```
asm ("SKIP_EA_DEC:);
asm ("stw r1, 4(sp)");  // Save all registers
asm ("stw r2, 8(sp)");
asm ("stw r3, 12(sp)");
asm ("stw r4, 16(sp)");
asm ("stw r5, 20(sp)");
asm ("stw r6, 24(sp)");
asm ("stw r7, 28(sp)");
asm ("stw r8, 32(sp)");
asm ("stw r9, 36(sp)");
asm ("stw r10, 40(sp)");
asm ("stw r11, 44(sp)");
asm ("stw r12, 48(sp)");
asm ("stw r13, 52(sp)");
asm ("stw r14, 56(sp)");
asm ("stw r15, 60(sp)");
asm ("stw r16, 64(sp)");
asm ("stw r17, 68(sp)");
asm ("stw r18, 72(sp)");
asm ("stw r19, 76(sp)");
asm ("stw r20, 80(sp)");
asm ("stw r21, 84(sp)");
asm ("stw r22, 88(sp)");
asm ("stw r23, 92(sp)");
asm ("stw r25, 100(sp)");  // r25 = bt (skip r24 = et, because it was saved above)
asm ("stw r26, 104(sp)");  // r26 = gp
// skip r27 because it is sp, and there is no point in saving this
asm ("stw r28, 112(sp)");  // r28 = fp
asm ("stw r29, 116(sp)");  // r29 = ea
asm ("stw r30, 120(sp)");  // r30 = ba
asm ("stw r31, 124(sp)");  // r31 = ra
asm ("addi fp, sp, 128");
asm ("call interrupt_handler");  // Call the C language interrupt handler
asm ("ldw r1, 4(sp)");  // Restore all registers
asm ("ldw r2, 8(sp)");
asm ("ldw r3, 12(sp)");
asm ("ldw r4, 16(sp)");
asm ("ldw r5, 20(sp)");
asm ("ldw r6, 24(sp)");
asm ("ldw r7, 28(sp)");

Figure 16. Exception handler code (Part b).
asm ( "ldw r8, 32(sp)" );
asm ( "ldw r9, 36(sp)" );
asm ( "ldw r10, 40(sp)" );
asm ( "ldw r11, 44(sp)" );
asm ( "ldw r12, 48(sp)" );
asm ( "ldw r13, 52(sp)" );
asm ( "ldw r14, 56(sp)" );
asm ( "ldw r15, 60(sp)" );
asm ( "ldw r16, 64(sp)" );
asm ( "ldw r17, 68(sp)" );
asm ( "ldw r18, 72(sp)" );
asm ( "ldw r19, 76(sp)" );
asm ( "ldw r20, 80(sp)" );
asm ( "ldw r21, 84(sp)" );
asm ( "ldw r22, 88(sp)" );
asm ( "ldw r23, 92(sp)" );
asm ( "ldw r24, 96(sp)" );
asm ( "ldw r25, 100(sp)" ); // r25 = bt
asm ( "ldw r26, 104(sp)" ); // r26 = gp
// skip r27 because it is sp, and we did not save this on the stack
asm ( "ldw r28, 112(sp)" ); // r28 = fp
asm ( "ldw r29, 116(sp)" ); // r29 = ea
asm ( "ldw r30, 120(sp)" ); // r30 = ba
asm ( "ldw r31, 124(sp)" ); // r31 = ra

asm ( "addi sp, sp, 128" );

asm ( "eret" );

/************************************************************
* Interrupt Service Routine: Determines what caused the interrupt and calls the
* appropriate subroutine.
* ipending – Control register 4 which has the pending external interrupts
************************************************************/

void interrupt_handler(void)
{
    int ipending;
    NIOS2_READ_IPENDING(ipending);
    if ( (ipending & 0x2) == 2 )
    {
        display_buffer = pushbutton_isr( display_buffer, sw_values );
    }
    return; // else, ignore the interrupt
}

Figure 16. Exception handler code (Part c).
/********************************************************************************************
 * Pushbutton - Interrupt Service Routine *
 * Parameters: *
 * Input: new_pattern, present_pattern *
 * Output: present_pattern (modified) *
 * *
 * If KEY 1 is pressed then lower halfword of new_pattern is copied into lower *
 * halfword of present_pattern *
 * If KEY 2 is pressed then lower halfword of new_pattern is copied into upper *
 * halfword of present_pattern *
 * If KEY 3 is pressed then present_pattern is set to FFFFFFFF *
 * *
 **********************************************************************************/

int pushbutton_isr( int present_pattern, int new_pattern )
{
    volatile int * pushbutton_ptr = (int *) PUSHBUTTON_BASE;
    int press, or_pattern, ret_pattern;

    press = *(pushbutton_ptr + 3);
    *(pushbutton_ptr + 3) = 0; // Clear the interrupt
    if (press & 0x2) // KEY1
    {
        ret_pattern = present_pattern & 0xFFFF0000;
        or_pattern = new_pattern & 0x0000FFFF;
        ret_pattern |= or_pattern;
    }
    else if (press & 0x4) // KEY2
    {
        ret_pattern = present_pattern & 0x0000FFFF;
        or_pattern = (new_pattern << 16) & 0xFFFF0000;
        ret_pattern |= or_pattern;
    }
    else // press & 8, which is KEY3
    {
        ret_pattern = 0xFFFFFFFF;
    }

    return (ret_pattern);
}

Figure 17. Pushbutton interrupt service routine.
Appendix C  Memory Layout

Table 3 summarizes the memory map used in the DE2 Basic Computer.

<table>
<thead>
<tr>
<th>Base Address</th>
<th>End Address</th>
<th>I/O Peripheral</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08000000</td>
<td>0x0807FFFF</td>
<td>SRAM</td>
</tr>
<tr>
<td>0x09000000</td>
<td>0x09007FFF</td>
<td>On-chip Memory</td>
</tr>
<tr>
<td>0x10000000</td>
<td>0x1000000F</td>
<td>Red LED parallel port</td>
</tr>
<tr>
<td>0x10000010</td>
<td>0x1000001F</td>
<td>Green LED parallel port</td>
</tr>
<tr>
<td>0x10000020</td>
<td>0x1000002F</td>
<td>7-segment HEX3–HEX0 displays parallel port</td>
</tr>
<tr>
<td>0x10000030</td>
<td>0x1000003F</td>
<td>7-segment HEX7–HEX4 displays parallel port</td>
</tr>
<tr>
<td>0x10000040</td>
<td>0x1000004F</td>
<td>Slider switch parallel port</td>
</tr>
<tr>
<td>0x10000050</td>
<td>0x1000005F</td>
<td>Pushbutton parallel port</td>
</tr>
<tr>
<td>0x10000060</td>
<td>0x1000006F</td>
<td>JP1 Expansion parallel port</td>
</tr>
<tr>
<td>0x10000070</td>
<td>0x1000007F</td>
<td>JP2 Expansion parallel port</td>
</tr>
<tr>
<td>0x10001000</td>
<td>0x10001007</td>
<td>JTAG port</td>
</tr>
<tr>
<td>0x10001010</td>
<td>0x10001017</td>
<td>Serial port</td>
</tr>
<tr>
<td>0x10002000</td>
<td>0x1000201F</td>
<td>Interval timer</td>
</tr>
<tr>
<td>0x10002020</td>
<td>0x10002027</td>
<td>System ID</td>
</tr>
</tbody>
</table>

Table 3. Memory layout used in the DE2 Basic Computer.