

## 1 Introduction

This tutorial presents some basic concepts that can be helpful in debugging of application programs written in the Nios® II assembly language, which run on the Intel® DE-series boards. A simple illustrative example involves the Intel FPGA Monitor Program software used to compile, load and run the example assembly language code on the Nios II processor with a DE-series board.

The reader is expected to be familiar with the Nios II processor, as well as the Intel FPGA Monitor Program. An introduction to this software is provided in tutorials: *Introduction to the Intel Nios II Soft Processor* and *Intel FPGA Monitor Program*, which can be found in the University Program section of the web site.

### Contents:

- Example Circuit
- Debugging Concepts
- Corrective Action
- Finding Errors in an Application Program
- Concluding Remarks

## 2 Background

Designers of digital systems are inevitably faced with the task of debugging their imperfect initial designs. This task becomes more difficult as the system grows in complexity. The debugging process requires determination of possible flaws in the designed circuits as well as the flaws in application programs that are supposed to run on this hardware. This tutorial addresses the second of these aspects of debugging. A discussion of hardware debugging is available in the tutorial *Debugging of Hardware Designs on DE-Series Boards*.

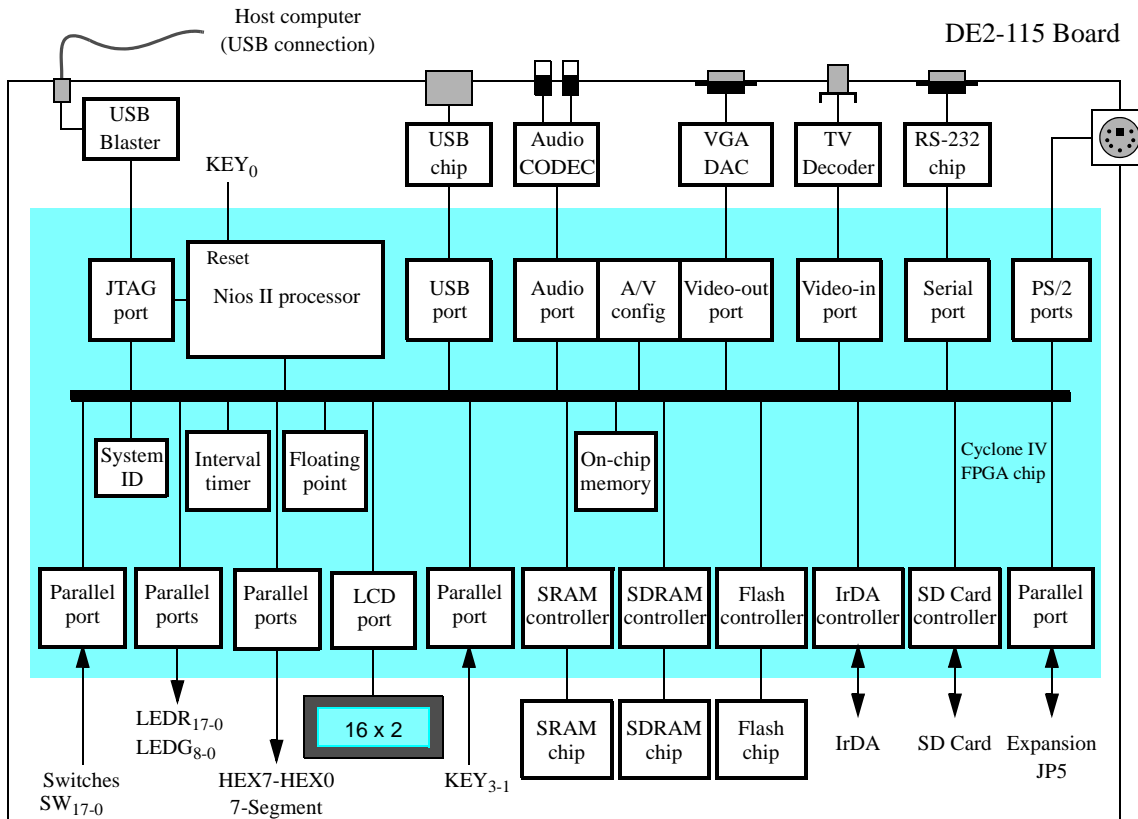


Figure 1. Block diagram of the DE2-115 Computer System.

## 3 Example System

Our example system is the DE2-115 Computer System that can be used to test the reaction time of a person to a visual stimulus. The user initiates the test by pressing a pushbutton key,  $KEY_1$ . After some delay (of at least one second) the circuit turns on a light. In response, the user presses any one of  $KEY_3$  to  $KEY_1$ , as quickly as possible, which results in turning the light off and displaying the elapsed time in hundredths of a second on the 7-segment displays on the DE-series board. A block diagram and Platform Designer implementation of the system for the DE2-115 board is given in Figure 1 and Figure 2, respectively.

The computer system for the other DE-series boards varies only where a given peripheral does not exist or is of a different size. See the computer system documentation for your specific board for more information.

			exported			
<input checked="" type="checkbox"/>	System_PLL	System and SDRAM Clocks for DE-seri...		exported		
<input checked="" type="checkbox"/>	Video_PLL	Video Clocks for DE-series Boards		exported		
<input checked="" type="checkbox"/>	Nios2	Nios II Processor		System_PL...	0x0a00_0000	0x0a00_07ff
<input checked="" type="checkbox"/>	Nios2_Floating_Point	Floating Point Hardware			Opcode 252	Opcode 255
<input checked="" type="checkbox"/>	Nios2_2nd_Core	Nios II Processor		System_PL...	0x0a00_0000	0x0a00_07ff
<input checked="" type="checkbox"/>	Nios2_2nd_Core_Flo...	Floating Point Hardware			Opcode 252	Opcode 255
<input checked="" type="checkbox"/>	JTAG_to_FPGA_B...	JTAG to Avalon Master Bridge		System_PL...		
<input checked="" type="checkbox"/>	SDRAM	SDRAM Controller Intel FPGA IP		System_PL...	0x0000_0000	0x07ff_ffff
<input checked="" type="checkbox"/>	SRAM	SRAM Controller		System_PL...	0x0800_0000	0x081f_ffff
<input checked="" type="checkbox"/>	SD_Card	SD Card Interface		System_PL...	0x0b00_0000	0x0b00_03ff
<input checked="" type="checkbox"/>	Flash	Intel FPGA University Program Flash M...		System_PL...	multiple	multiple
<input checked="" type="checkbox"/>	Red_LEDs	PIO (Parallel I/O) Intel FPGA IP		System_PL...	0xff20_0000	0xff20_000f
<input checked="" type="checkbox"/>	Green_LEDs	PIO (Parallel I/O) Intel FPGA IP		System_PL...	0xff20_0010	0xff20_001f
<input checked="" type="checkbox"/>	HEX3_HEX0	PIO (Parallel I/O) Intel FPGA IP		System_PL...	0xff20_0020	0xff20_002f
<input checked="" type="checkbox"/>	HEX7_HEX4	PIO (Parallel I/O) Intel FPGA IP		System_PL...	0xff20_0030	0xff20_003f
<input checked="" type="checkbox"/>	Slider_Switches	PIO (Parallel I/O) Intel FPGA IP		System_PL...	0xff20_0040	0xff20_004f
<input checked="" type="checkbox"/>	Pushbuttons	PIO (Parallel I/O) Intel FPGA IP		System_PL...	0xff20_0050	0xff20_005f
<input checked="" type="checkbox"/>	Expansion_JP5	PIO (Parallel I/O) Intel FPGA IP		System_PL...	0xff20_0060	0xff20_006f
<input checked="" type="checkbox"/>	PS2_Port	PS/2 Controller		System_PL...	0xff20_0100	0xff20_0107
<input checked="" type="checkbox"/>	PS2_Port_Dual	PS/2 Controller		System_PL...	0xff20_0108	0xff20_010f
<input checked="" type="checkbox"/>	USB	USB Controller		System_PL...	0xff20_0110	0xff20_011f
<input checked="" type="checkbox"/>	JTAG_UART	JTAG UART Intel FPGA IP		System_PL...	0xff20_1000	0xff20_1007
<input checked="" type="checkbox"/>	JTAG_UART_2nd_Co...	JTAG UART Intel FPGA IP		System_PL...	0xff20_1000	0xff20_1007
<input checked="" type="checkbox"/>	Serial_Port	RS232 UART		System_PL...	0xff20_1010	0xff20_1017
<input checked="" type="checkbox"/>	IrDA	IrDA UART		System_PL...	0xff20_1020	0xff20_1027
<input checked="" type="checkbox"/>	Interval_Timer	Interval Timer Intel FPGA IP		System_PL...	0xff20_2000	0xff20_201f
<input checked="" type="checkbox"/>	Interval_Timer_2	Interval Timer Intel FPGA IP		System_PL...	0xff20_2020	0xff20_203f
<input checked="" type="checkbox"/>	Interval_Timer_2nd...	Interval Timer Intel FPGA IP		System_PL...	0xff20_2000	0xff20_201f
<input checked="" type="checkbox"/>	Interval_Timer_2nd...	Interval Timer Intel FPGA IP		System_PL...	0xff20_2020	0xff20_203f
<input checked="" type="checkbox"/>	SysID	System ID Peripheral Intel FPGA IP		System_PL...	0xff20_2040	0xff20_2047
<input checked="" type="checkbox"/>	AV_Config	Audio and Video Config		System_PL...	0xff20_3000	0xff20_300f
<input checked="" type="checkbox"/>	VGA_Subsystem	VGA_Subsystem		multiple	multiple	multiple
<input checked="" type="checkbox"/>	Audio_Subsystem	Audio_Subsystem		multiple	multiple	multiple
<input checked="" type="checkbox"/>	Char_LCD_16x2	16x2 Character Display		System_PL...	0xff20_3050	0xff20_3051
<input checked="" type="checkbox"/>	Video_In_Subsystem	Video_In_Subsystem		System_PL...	multiple	multiple

Figure 2. The system defined in the Platform Designer tool.

The processor is Nios II/s and the application is loaded from SDRAM. The following key components and PIO interfaces are implemented:

- *Pushbuttons* is a multi-bit input port that reacts to the falling edge of the input signal by setting to 1 a corresponding bit in its *edge-capture register*; *KEY<sub>1</sub>* (bit1) is used as the button to start the test and is configured in software to raise an interrupt when pressed.
- *Green\_LEDs* is a multi-bit output port that drives the green LED that tells the user to react.
- *Interval\_Timer* is a 32-bit interval timer with a writable period and will be used to measure the reaction time.

The addresses assigned to the PIO interfaces and interval timer are as indicated in Figure 2. A possible application program, written in the Nios II assembly language, is presented in Figure 3. Note that we have numbered the statements to make it easy to refer to them in the discussion below. The main program sets up the stack and enables interrupts to be raised by the interval timer and the *Pushbuttons* PIO. It also sets up the processor to allow interrupts. Finally, it enters a loop continuously checking the status of the start flag before it begins the test.

```

1  .include      "nios_macros.s"
2  .equ         GREEN_LED_BASE, 0xFF200010      /* Green_LEDs PIO */
3  .equ         HEX3_HEX0_BASE, 0xFF200020     /* Seven Segment PIO */
4  .equ         PUSHBUTTON_BASE, 0xFF200050    /* Pushbutton PIO */
5  .equ         INTERVAL_TIMER_BASE, 0xFF202000 /* Interval Timer */
6  .equ         stack_end, 0x007FFFFFFC       /* Initial stack address */
7  .section     .reset, "ax"
8              br         _start                /* Upon reset go to start */
9  .section     .exceptions, "ax"
10 /* Interrupt service routine */
11      subi     sp, sp, 4                       /* Save registers on */
12      stw     r2, 0(sp)                       /* the stack */
13      subi     sp, sp, 4
14      stw     r3, 0(sp)
15      subi     sp, sp, 4
16      stw     r4, 0(sp)
17      subi     sp, sp, 4
18      stw     ra, 0(sp)
19      rdctl   et, ctl4                       /* Check if external */
20      beq     et, r0, END_ISR                 /* interrupt has occurred */
21      subi     ea, ea, 4                       /* If yes, decrement ea */
22      andi    r4, et, 1                       /* Check if timer irq0 has */
23                                             /* been asserted */
24      beq     r4, r0, BUTTONIRQ              /* If not, do not call timer ISR */
25      call    TIMERISR                       /* Call timer ISR */
26 BUTTONIRQ:  andi    r4, et, 2               /* Check if pushbuttons irq1 */
27                                             /* has been asserted */
28      beq     r4, r0, END_ISR                 /* If not, end exception */
29      movia   r4, PUSHBUTTON_BASE           /* Clear button edge capture */
30      addi    r3, r0, 0
31      stwio   r3, 12(r4)
32      movia   r11, 0x1                       /* Set start flag to 1 */
33 END_ISR:   ldw     ra, 0(sp)                 /* Restore registers */
34      addi    sp, sp, 4                       /* from the stack */
35      ldw     r4, 0(sp)
36      addi    sp, sp, 4
37      ldw     r3, 0(sp)
38      addi    sp, sp, 4
39      ldw     r2, 0(sp)
40      addi    sp, sp, 4
41      eret                                     /* Return from Exception */

```

Figure 3. Application program for the example system (Part a)

```

42 TIMERISR:  movia    r2, INTERVAL_TIMER_BASE
43             ldwio    r4, 0(r2)           /* Load status register */
44             andi    r4, r4, 1
45             stwio   r4, 0(r2)           /* Clear Timeout status */
46             addi   r10, r10, 1          /* Add one to counter value */
47             ret
48 .text
49 .global    __start
50 __start:   movia    sp, stack_end       /* Initialize the stack pointer */
51             addi   r2, r0, 1           /* Enable the processor to */
52             wrctl  ct10, r2           /* accept external interrupts */
53             addi   r2, r0, 3           /* Enable irq1 and irq0 */
54             wrctl  ct13, r2
55             movia  r2, PUSHBUTTON_BASE /* Setup interrupt */
56             addi   r3, r0, 2           /* mask for push button PIO */
57             stbio  r3, 8(r2)
58             movia  r2, INTERVAL_TIMER_BASE /* Setup Interval Timer */
59             addi   r3, r0, 11          /* Continous operation */
60             stbio  r3, 4(r2)           /* Timer interrupts are enabled */
61             movia  r3, 0xa120          /* Write Timer Period */
62             stwio  r3, 8(r2)
63             addi   r3, r0, 0x7
64             stwio  r3, 12(r2)
65             add    r10, r0, r0         /* Initialize counter to 0 */
66 IDLE:      andi    r11, r11, 0x1       /* Check first bit of start flag */
67                                           /* only */
68             beq    r11, r0, IDLE       /* Loop if start flag has not */
69                                           /* been set */
70 NEW_TEST:  andi    r11, r11, 0         /* Reset start flag */
71             addi   r2, r0, 1           /* Disable IRQ1 interrupt */
72             wrctl  ct13, r2
73             movia  r2, GREEN_LED_BASE /* Turn the green */
74             stbio  r0, 0(r2)         /* light off */
75             addi   r3, r0, 1           /* Set the count of the */
76             slli   r3, r3, 25         /* delay loop */
77 DELAY:     subi   r3, r3, 1           /* The delay loop */
78             bgt    r3, r0, DELAY
79             addi   r3, r0, 1           /* Turn the green */
80             stbio  r3, 0(r2)         /* light on */
81             movia  r2, INTERVAL_TIMER_BASE
82             addi   r10, r0, 0         /* Clear timer count value */
83             movia  r3, 0xa120          /* Reset timer */
84             stwio  r3, 8(r2)
85             addi   r3, r0, 0x7
86             stwio  r3, 12(r2)
87             addi   r3, r0, 0x7         /* Start counter */
88             stwio  r3, 4(r2)
89             movia  r2, PUSHBUTTON_BASE

```

Figure 3. Application program for the example system (Part b).

```

90 WAIT:      ldbio      r3, 0(r2)          /* Check if the Stop key */
91           ble       r3, r0, WAIT      /* has been pressed */
92           movia    r2, INTERVAL_TIMER_BASE
93           addi     r3, r0, 8          /* Stop the timer */
94           stbio    r3, 4(r2)
95           movia    r2, GREEN_LED_BASE /* Turn the green */
96           stbio    r0, 0(r2)        /* LED off */
97           call     DECIMAL           /* Prepare the 4-digit */
98           call     DISPLAY           /* decimal display */
99           addi     r2, r0, 3          /* Enable irq1 and irq0 */
100          wrctl    ct13, r2          /* (Interval Timer and */
101                                     /* Pushbutton) */
102          br       IDLE
103 DECIMAL:   add      r4, r0, r0
104           addi     r5, r0, 8
105           addi     r2, r0, 4
106           addi     r6, r0, 10
107 LOOP:     divu     r7, r10, r6        /* Divide by 10 */
108           mul      r8, r7, r6
109           sub      r8, r10, r8        /* Remainder is in r8 */
110           ldb      r9, PATTERN(r8)   /* The decoded hex display */
111           or       r4, r4, r9        /* dig is placed into proper */
112           ror      r4, r4, r5        /* position of the 4-digit */
113                                     /* display. */
114           add      r10, r7, r0
115           subi     r2, r2, 1
116           bgt     r2, r0, LOOP
117           ret
118 DISPLAY:  movia    r2, HEX3_HEX0_BASE /* Display the 4-digit */
119           stwio    r4, 0(r2)        /* reaction time */
120           ret
121
122 PATTERN:
123 /* Translation table */
124 .byte     0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F
125 .end

```

Figure 3. Application program for the example system (Part c).

The interrupt service routine is used to set the start test flag and to service the timer interrupts. The main program uses three subroutines *NEW\_TEST*, *DECIMAL*, and *DISPLAY*. *NEW\_TEST* controls the counter that measures the reaction time. *DECIMAL* converts the binary count into a decimal number by using the standard conversion method of dividing by 10 and keeping track of the remainder. The *DISPLAY* subroutine simply displays the reaction time on the seven-segment display. Note that the interrupt service routine follows the normal practice of saving the registers that it uses (*r2*, *r3* and *r4*) on the stack. Registers (*r10* and *r11*) are used to store the counter value and the start flag value.

## 4 Debugging Concepts

Debugging of very small programs is fairly simple, but the task can be difficult when larger programs are involved. The chore is made easier if one uses an organized approach with the aid of debugging tools.

The debugging task involves:

- Observing that there is a problem
- Identifying the source of the problem, i.e. the nature and the location of a bug
- Determining the corrective action that has to be taken
- Fixing the bug
- Testing the corrected program

### 4.1 Observing a Problem

Often it is easy to see that there is a problem because the observed behavior does not match the designer's expectations in an obvious way. In our reaction tester, an obvious problem appears when a bug has an effect such as not turning the green light on or off, producing a meaningless output on the 7-segment display, or not producing any output at all.

Bugs can also cause errors that are not readily apparent. For example, our tester may evaluate the reaction of a user incorrectly and display a time that is wrong by a factor of 2 or more. A difference between 20 and 40 one-hundredths of a second may not be easy to spot.

### 4.2 Identifying the Bug

Finding the source of a bug is often a challenging task. It requires a good understanding of the computation that the application program is supposed to perform. Breakpoints can be used stop the execution of a program at a particular point of interest. Then, examining the contents of processor registers may indicate a problem. Examining the contents of memory locations can also be helpful. Single stepping through a portion of the code can be used to verify that the instructions generate the expected intermediate results.

For a large program, it is prudent to use the divide-and-conquer approach. Relatively small portions of the program may be tested separately. This is especially effective if the program is written in a modular fashion. For example, in the program in Figure 3 it is possible check the code that converts a binary count into a decimal number by loading a predetermined value into register *r10* and then executing the subroutine DECIMAL.

The Intel FPGA Monitor Program provides a useful vehicle for both running and debugging of programs. It gives the user an ability to:

- Compile and assemble a program
- Load the program into the FPGA on a DE-series board

- Run the program
- Stop and restart the program
- Reset the system, which includes reloading the program
- Examine the source code and the disassembled code
- Single step through the program
- Set breakpoints in the program
- Set a watch expression
- Examine the instruction trace
- Look at the contents of various registers and memory locations
- Write data directly into processor registers and memory locations

Using the Monitor Program, the user can investigate the state of the system at various points of execution of a program that has to be debugged. The execution is stopped by placing a breakpoint, at which point one can examine the contents of processor registers and memory locations as well as observe the results of evaluated watches. New data can be written directly into both registers and memory locations. Single-stepping through critical sections of the program is useful to determine whether the flow of execution and the generated intermediate results are correct.

Success in debugging tends to reflect the user's experience. An experienced user, who has debugged many different programs, will need much less time than a novice.

## 5 Corrective Action

Having identified a bug, it is necessary to determine its cause. There are many possible causes, which include:

- Typing mistakes
- Erroneous communication with input/output devices, particularly when interrupts are involved
- Improper subroutine linkage
- Errors in addressing of memory locations

Of course, a poorly written program may also contain code that simply does not implement a desired task.

Once a bug is identified, the necessary corrective action is obvious in most cases. But, sometimes the user may need to learn more about the details of the system that does not appear to be functioning correctly. For example, a bug due to an error in interrupt handling cannot be dealt with without a sufficient knowledge of the interrupt mechanism. A corrected program must be tested to ensure that it does not contain other bugs.



## 6 Examples of Errors in an Application Program

### 6.1 Syntax Errors

Syntax errors are usually easy to find and correct. The Nios II compiler and assembler will generate messages that identify the erroneous statements. For example, suppose that instead of statement 61, which loads a part of the desired period into the *periodl* register of the interval timer, we include the statement

```
addi r3,r0,0xa120
```

In this case, the Monitor Program will indicate an error and display the message

```
Error : immediate value 41248 out of range - 32768 to 32767
```

because the specified immediate operand cannot be represented with 16 bits.

### 6.2 Typing Errors

Errors due to typing mistakes may be difficult to discover. Consider an error where the statement 87 is mistyped as

```
addi r3,r0,0x8
```

Running the erroneous program will execute all instructions in the correct sequence, but the displayed output will indicate that no time was taken, which is obviously the wrong result. In this case, one can speculate that something is wrong with the counting mechanism. This can be verified by checking the value read from the counter, which is used in the conversion to a decimal number. The value of interest is in register *r10*. To discover this value, use the Monitor Program to place a breakpoint at instruction 97 and then run the application program. The execution will halt just before the **call** instruction is executed which causes a branch to the subroutine DECIMAL. Observe that the value in register *r10* will be 0, which suggests that the obtained count was wrong.


Next, we should check the counting scheme. An error may occur if there is something wrong with instructions that control the counter. Remove the previous breakpoint, place a new breakpoint at instruction 79, and run the program. The execution will stop at this instruction. Now, start single stepping through the instructions that follow. Observe that the green light will turn on when instruction 80 is executed. The next three instructions reset the reaction time count register. They are followed by instructions that enable the counter. Single step through these instructions and observe the contents of registers *r3* and *r10* in the Registers pane of the Monitor Program window. *r10* should be 0 when the counter is cleared. Observing that instruction 87 causes the contents of *r3* to be 8, which means the timer was never correctly started.

### 6.3 Errors in Interrupt Handling

We will consider two errors that an inexperienced user may make.

#### 6.3.1 Failure to Enable Interrupts

Interrupts must be handled exactly as specified by the Nios II interrupt mechanism. Any omissions in the initialization of interrupts will result in failure. For example, suppose we forget to set a required interrupt mask bit in

an I/O interface. To illustrate this, remove statements 55, 56 and 57 in our example program. Recompile and run the program. When this program is running there will be no observable response to pressing the pushbutton,  $KEY_1$ . Stop the execution of the program by clicking on the Monitor Program icon . Note that the program execution stops at instruction 68, which suggests that the interrupt service routine may not have been reached. This guess can be verified by placing a breakpoint in the interrupt service routine, perhaps at instruction 11, and running the program again. The same behavior will be observed; the breakpoint will not be reached during the execution of the program. Clearly, the interrupt service routine is not being executed. Now, check the values in the processor control registers, which will be 0x00000001 and 0x00000003 in registers *status* and *ienable*, respectively. This indicates that an interrupt from the *Pushbuttons* PIO will be recognized and serviced by the processor. However, the contents of register *ipending* are equal to zero, even though  $KEY_1$  has been pressed. This means that no interrupt request appears at the processor, because the PIO has not raised such a request. The likely reason is that this interrupt has not been enabled in the PIO circuit, which indeed is the case in this example.

The reader may be tempted to simply single step through it until the interrupt service routine is reached. This would not work because the single-stepping feature of the Intel FPGA Monitor Program disables all interrupts. Hence, single stepping will never get past the branch instruction in statement 68.

### 6.3.2 Failure to Clear Interrupt Flags

Remove statements 30 and 31 in Figure 3 and run the program. Place a breakpoint at line 29. When  $KEY_1$  is pressed, the breakpoint at the pushbutton interrupt service routine will be reached. Continue the program and observe that the breakpoint will be continuously reached without pressing  $KEY_1$ . This behavior suggests that continuous interrupts are being received by the processor. Remove the breakpoint at line 25. Observe the state of the control registers at the end of the interrupt service routine by placing a breakpoint at line 41. The *status* register is properly cleared to zero, which happens when the processor enters the interrupt service routine in response to an interrupt request. The immediately preceding state of this register is saved in the *estatus* register, which has a 1 in bit *estatus*<sub>0</sub> as expected. The *ienable* register shows that interrupts from the *Pushbuttons* PIO are enabled, which is also correct. The *ipending* register indicates that there is an outstanding interrupt request, which the processor is trying to service. This is wrong, because a single pressing of  $KEY_1$  should not result in multiple interrupts.

At this point it may be useful to look at what is happening in the PIO. Place a breakpoint in the beginning of the interrupt service routine, at statement 9. Run the program and press  $KEY_1$ . The execution will stop at the breakpoint. To examine the state of the PIO registers, click on the Memory tab in the Monitor window. The *Pushbuttons* PIO registers are mapped to memory locations starting at 0xFF200050. So, type FF200050 in the address box of the displayed window and click GO to display the contents starting at this address. Note that 32 bits are shown for each register address (because these registers can be specified to have from 1 to 32 bits), even though each of these registers are much smaller in our circuit. The unimplemented bit positions are displayed as being 0. The *data* register (at address FF200050) contains *data*<sub>0</sub> = 0, which is the state of  $KEY_1$  when it is not pressed. The *interruptmask* register (at address FF200058) shows *interruptmask*<sub>1</sub> = 1, because this bit was set by executing instruction 57. The *edgecapture* register (at address FF20005C) also shows the value 2, which means that the PIO has an active interrupt request. This bit must be cleared to 0 by the interrupt service routine. Now single step through the instructions until the end of the interrupt service routine. Observe that the state of *edgecapture* register does not change, which means that the processor will see another interrupt request as soon as it exits the interrupt service routine (at which time it restores its *status* register using the contents of register *estatus*). The conclusion is that the program lacks an instruction that clears the *edgecapture* register.

## 6.4 Subroutine Linkage Errors

When calling a subroutine by executing the `call` instruction, a Nios II processor saves the return address (which is the address of the next instruction) in register *ra* (*r31*). A return from the subroutine, performed by executing the `ret` instruction, loads the address in *ra* into the program counter. In the case of nested subroutines, it is essential to save the address in *ra* on the stack, so that it is not lost when the inner subroutine is called.

## 6.5 Errors in Addressing

If an instruction addresses a wrong location it will lead to a bad outcome. Consider an error where the address in statement 3 in Figure 4 is specified as FF200030 rather than FF200020. Running this program turns the green light on and off as expected, but the 7-segment display will not display a value which indicates that there is a problem.

Placing breakpoints into the various segments of the program and stepping through these segments will indicate that the flow of execution appears to be correct. The error can be discovered by checking the contents of processor registers while single stepping through the instructions that read the contents of the counter and convert them for display.

## 7 Concluding Remarks

In this tutorial we discussed the key issues in debugging. Bugs range from simple to very difficult to find. Successful debugging is contingent on:

- Good understanding of the program that is being debugged
- Appreciation of possible problems
- Organized approach in detecting a problem
- Debugging aids that are provided in tools such as the Intel FPGA Monitor Program

Another key factor is the experience of the user. A novice can become an expert only through experience.

Copyright © Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Avalon, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

\*Other names and brands may be claimed as the property of others.