



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 Altera's DE-series boards. A simple illustrative example involves Altera's 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 Altera Monitor Program. An introduction to this software is provided in tutorials: *Introduction to the Altera Nios II Soft Processor* and *Altera 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

different size. See the Media Computer documentation for your specific board for more information.

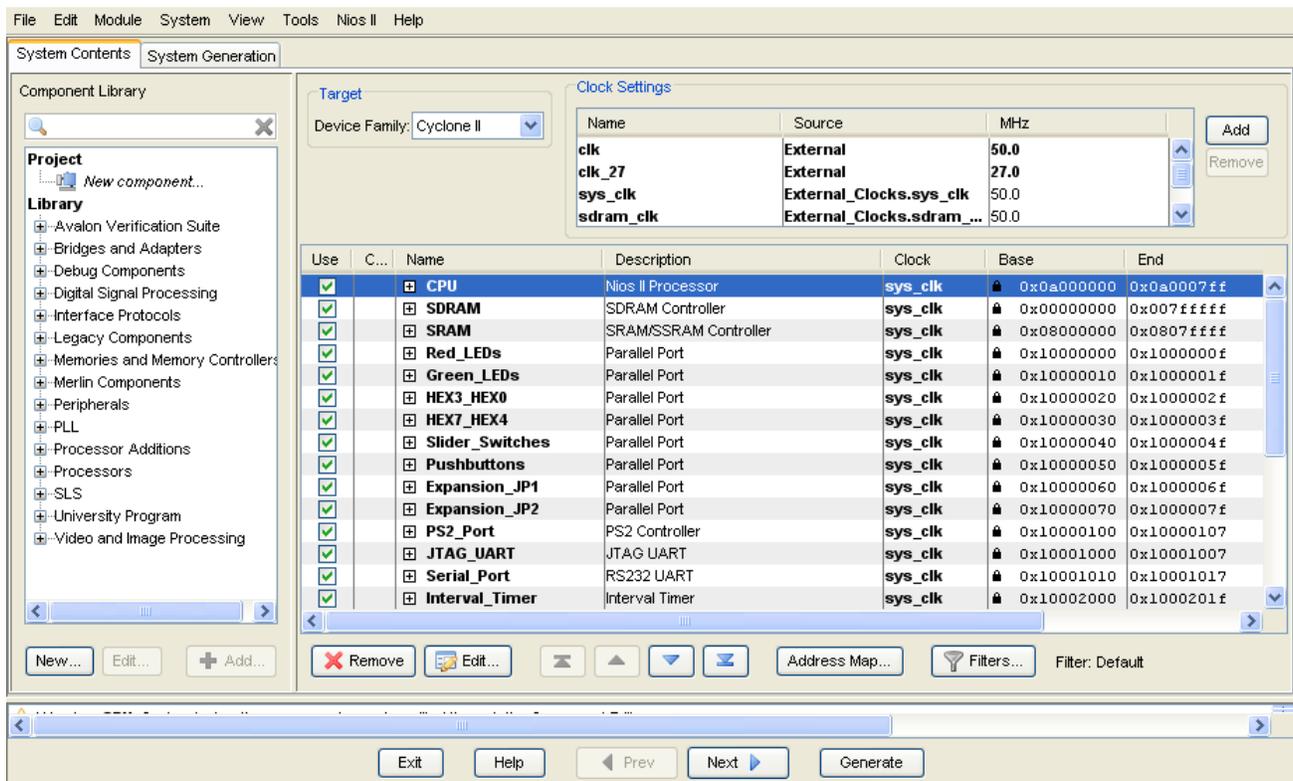


Figure 2. The system defined in the SOPC Builder.

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*; *KEY₁* (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, 0x10000010          /* Green_LEDs PIO */
3  .equ  HEX3_HEX0_BASE, 0x10000020        /* Seven Segment PIO */
4  .equ  PUSHBUTTON_BASE, 0x10000050       /* Pushbutton PIO */
5  .equ  INTERVAL_TIMER_BASE, 0x10002000   /* Interval Timer */
6  .equ  stack_end, 0x007FFFFC            /* Initial stack address */
   .text
7          br _start                        /* Upon reset go to start */
8  .org    0x20
   /* Interrupt service routine */
9          subi sp, sp, 4                    /* Save registers on */
10         stw r2, 0(sp)                     /* the stack */
11         subi sp, sp, 4
12         stw r3, 0(sp)
13         subi sp, sp, 4
14         stw r4, 0(sp)
15         subi sp, sp, 4
16         stw ra, 0(sp)
17         rdctl et, ctl4                    /* Check if external interrupt */
18         beq et, r0, END_ISR               /* has occurred */
19         subi ea, ea, 4                    /* If yes, decrement ea */
20         andi r4, et, 1                    /* Check if timer irq0 has been asserted */
21         beq r4, r0, BUTTONIRQ            /* If not, do not call timer ISR */
22         call TIMERISR                     /* Call timer ISR */
23  BUTTONIRQ: andi r4, et, 2                 /* Check if pushbuttons irq1 has been asserted */
24         beq r4, r0, END_ISR               /* If not, end exception */
25         movia r4, PUSHBUTTON_BASE        /* Clear button edge capture */
26         addi r3, r0, 0
27         stwio r3, 12(r4)
28         movia r11, 0x1                    /* Set start flag to 1 */
29  END_ISR: ldw ra, 0(sp)                   /* Restore registers */
30         addi sp, sp, 4                    /* from the stack */
31         ldw r4, 0(sp)
32         addi sp, sp, 4
33         ldw r3, 0(sp)
34         addi sp, sp, 4

```

... continued in Part *b*

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

```

35         ldw r2, 0(sp)
36         addi sp, sp, 4
37         eret                                /* Return from Exception */
38  TIMERISR:  movia r2, INTERVAL_TIMER_BASE
39         ldwio r4, 0(r2)                     /* Load status register */
40         andi r4, r4, 1
41         stwio r4, 0(r2)                     /* Clear Timeout status */
42         addi r10, r10, 1                    /* Add one to counter value */
43         ret
44  .global _start
45  _start:   movia sp, stack_end              /* Initialize the stack pointer */
46         addi r2, r0, 1                      /* Enable the processor to accept */
47         wrctl ctl0, r2                      /* external interrupts. */
48         addi r2, r0, 3                      /* Enable irq1 and irq0 */
49         wrctl ctl3, r2
50         movia r2, PUSHBUTTON_BASE          /* Setup interrupt */
51         addi r3, r0, 2                      /* mask for push button PIO */
52         stbio r3, 8(r2)
53         movia r2, INTERVAL_TIMER_BASE     /* Setup Interval Timer */
54         addi r3, r0, 11                     /* Continous operation */
55         stbio r3, 4(r2)                     /* Timer interrupts are enabled */
56         movia r3, 0xa120                    /* Write Timer Period */
57         stwio r3, 8(r2)
58         addi r3, r0, 0x7
59         stwio r3, 12(r2)
60         add r10, r0, r0                     /* Initialize counter to 0 */
61  IDLE:     andi r11, r11, 0x1               /* Check first bit of start flag only */
62         beq r11, r0, IDLE                  /* Loop if start flag has not been set */
63  NEW_TEST: andi r11, r11, 0                /* Reset start flag */
64         addi r2, r0, 1                      /* Disable IRQ1 interrupt */
65         wrctl ctl3, r2
66         movia r2, GREEN_LED_BASE           /* Turn the green */
67         stbio r0, 0(r2)                     /* light off */
68         addi r3, r0, 1                      /* Set the count of the */
69         slli r3, r3, 25                     /* delay loop */
70  DELAY:   subi r3, r3, 1                   /* The delay loop */
71         bgt r3, r0, DELAY
72         addi r3, r0, 1                      /* Turn the green */
73         stbio r3, 0(r2)                     /* light on */
74         movia r2, INTERVAL_TIMER_BASE

```

... continued in Part c

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

```

75         addi r10, r0, 0                /* Clear timer count value */
76         movia r3, 0xa120             /* Reset timer */
77         stwio r3, 8(r2)
78         addi r3, r0, 0x7
79         stwio r3, 12(r2)
80         addi r3, r0, 0x7             /* Start counter */
81         stwio r3, 4(r2)
82         movia r2, PUSHBUTTON_BASE
83  WAIT:  ldbio r3, 0(r2)              /* Check if the Stop key */
84         ble r3, r0, WAIT            /* has been pressed */
85         movia r2, INTERVAL_TIMER_BASE
86         addi r3, r0, 8              /* Stop the timer */
87         stbio r3, 4(r2)
88         movia r2, GREEN_LED_BASE    /* Turn the green */
89         stbio r0, 0(r2)            /* LED off */
90         call DECIMAL                /* Prepare the 4-digit decimal display */
91         call DISPLAY
92         addi r2, r0, 3              /* Enable irq1 and irq0 */
93         wrctl ctl3, r2             /* (Interval Timer and Pushbutton)*/
94         br IDLE
95  DECIMAL: add r4, r0, r0
96         addi r5, r0, 8
97         addi r2, r0, 4
98         addi r6, r0, 10
99  LOOP:  divu r7, r10, r6            /* Divide by 10 */
100        mul r8, r7, r6
101        sub r8, r10, r8            /* Remainder is in r8 */
102        ldb r9, 0x200(r8)         /* The decoded hex display dig */
103        or r4, r4, r9            /* is placed into proper position */
104        ror r4, r4, r5            /* of the 4-digit display. */
105        add r10, r7, r0
106        subi r2, r2, 1
107        bgt r2, r0, LOOP
108        ret
109  DISPLAY: movia r2, HEX3_HEX0_BASE /* Display the 4-digit */
110        stwio r4, 0(r2)         /* reaction time */
111        ret
112  .org 0x200
113        /* Translation table */
114  .byte 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F
115  .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 Altera 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 56, 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 80 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 90 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 72, 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 73 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 80 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 50, 51 and 52 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 62, 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 9, 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 Altera Monitor Program disables all interrupts. Hence, single stepping will never get past the branch instruction in statement 62.

6.3.2 Failure to Clear Interrupt Flags

Remove statements 26 and 27 in Figure 3 and run the program. Place a breakpoint at line 25. 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 37. 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*₀ 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 0x10000050. So, type 10000050 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 10000050) contains *data*₀ = 0, which is the state of *KEY*₁ when it is not pressed. The *interruptmask*

register (at address 10000058) shows $interruptmask_1 = 1$, because this bit was set by executing instruction 52. The *edgecapture* register (at address 1000005C) 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 10000030 rather than 10000020. 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 Altera Monitor Program

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

Copyright ©1991-2011 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera 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 Altera Corporation. Altera 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.

This document is being provided on an "as-is" basis and as an accommodation and therefore all warranties, representations or guarantees of any kind (whether express, implied or statutory) including, without limitation, warranties of merchantability, non-infringement, or fitness for a particular purpose, are specifically disclaimed.