**Basic Computer System
for the Intel DE0-Nano Board**

1 Introduction

This document describes a simple computer system that can be implemented on the Intel DE0-Nano development and education board. This system, called the *DE0-Nano 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 Intel's web site.

2 DE0-Nano Basic Computer Contents

A block diagram of the DE0-Nano Basic Computer is shown in Figure 1. Its main components include the Intel Nios II processor, memory for program and data storage, parallel ports connected to switches and lights, and a timer module. As shown in the figure, the processor and its interfaces to I/O devices are implemented inside the Cyclone®IV FPGA chip on the DE0-Nano board. Each of the components shown in Figure 1 is described below.

2.1 Nios II Processor

The Intel Nios®II processor is a 32-bit CPU that can be instantiated in an Intel FPGA chip. Three versions of the Nios II processor are available, designated economy (/e), standard (/s), and fast (/f). The DE0-Nano Basic Computer includes the Nios II/e 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 Intel Nios II Processor*, which is provided in the University Program section of Intel's web site. An easy way to begin working with the DE0-Nano Basic Computer and the Nios II processor is to make use of a utility called the *Intel FPGA Monitor Program*. This utility provides an easy way to assemble and compile Nios II programs on the DE0-Nano Basic Computer that are written in either assembly language or the C programming language. The Monitor Program, which can be downloaded from Intel's web site, is an application program that runs on the host computer connected to the DE0-Nano 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/edit the contents of memory on the DE0-Nano board, and similar operations. The Monitor Program includes the DE0-Nano Basic Computer as a predesigned system that can be downloaded onto the DE0-Nano board, as well as several sample programs in assembly language and C that show how to use various peripheral devices in the DE0-Nano Basic Computer. Some images that show how the DE0-Nano Basic Computer is integrated with the Monitor Program are described in section 7. An overview of the Monitor Program is available in the document *Intel FPGA Monitor Program Tutorial*, which is provided in Intel's FPGA University Program web site.
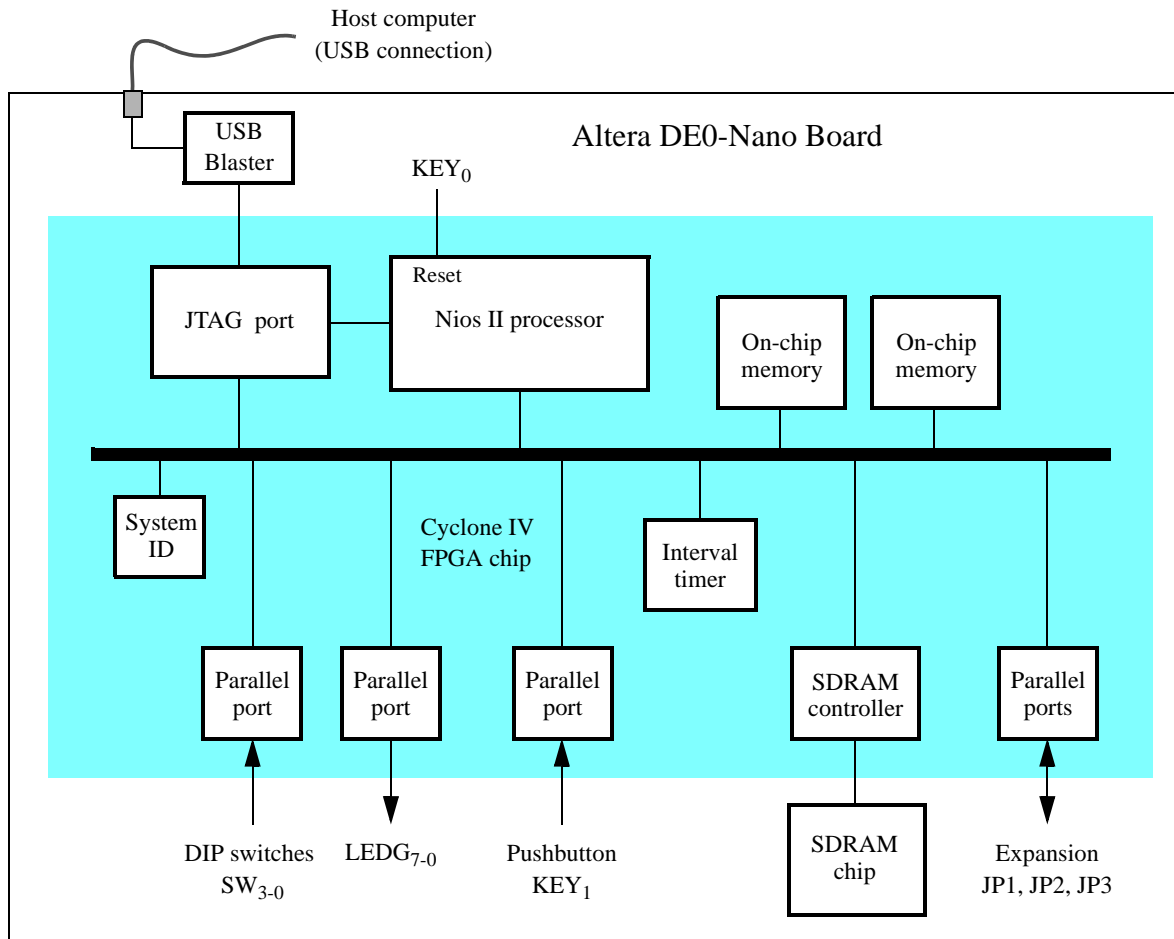
Figure 1. Block diagram of the DE0-Nano Basic Computer.

As indicated in Figure 1, the Nios II processor can be reset by pressing *KEY*$_0$ on the DE0-Nano board. The reset mechanism is discussed further in section 3. All of the I/O peripherals in the DE0-Nano Basic Computer are accessible by the processor as memory mapped devices, using the address ranges that are given in the following subsections.

2.2 Memory Components

The DE0-Nano Basic Computer has two types of memory components: SDRAM and on-chip memory inside the FPGA chip. Each type of memory is described below.

2.2.1 SDRAM

An SDRAM Controller provides a 32-bit interface to the synchronous dynamic RAM (SDRAM) chip on the DE0-Nano board. This SDRAM chip is organized as 16M x 16 bits, but is accessible by the Nios II processor using word (32-bit), halfword (16-bit), or byte operations. The SDRAM memory is mapped to the address space 0x00000000

to 0x01FFFFFF.

2.2.2 On-Chip Memory

The DE0-Nano Basic Computer includes an 8-Kbyte memory that is implemented in the Cyclone IV FPGA chip. This memory is organized as 2K x 32 bits, and can be accessed using either word, halfword, or byte operations. The memory spans addresses in the range 0x08000000 to 0x08001FFF.

2.2.3 On-Chip Memory

The DE0-Nano Basic Computer includes a second 8-Kbyte memory implemented in the Cyclone IV FPGA chip. This memory is also organized as 2K x 32 bits, and can be accessed using either word, halfword, or byte operations. It spans addresses in the range 0x09000000 to 0x09001FFF.

2.3 Parallel Ports

The DE0-Nano 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.

Address	31 30 $\cdots$ 4 3 2 1 0	
Base	Input or output data bits	Data register
Base + 4	Direction bits	Direction register
Base + 8	Mask bits	Interruptmask register
Base + C	Edge bits	Edgecapture register

Figure 2. Parallel port registers in the DE0-Nano Basic Computer.

Some of the parallel ports in the DE0-Nano Basic Computer have registers at addresses *Base* + 8 and *Base* + C, as indicated in Figure 2. These registers are discussed in section 3.

2.3.1 Green LED Parallel Ports

The green lights $LEDG_{7-0}$ on the DE0-Nano board are driven by an output parallel port, as illustrated in Figure 3. This port has an eight-bit *Data* register that is mapped to address 0x10000010. The register can be written using word accesses, with the upper bits that are not used in the registers being ignored.

Address

0x10000010

| 31 | Unused | 8 | 7 | $\cdots$ | 0 | Data register

LEDG$_7$ LEDG$_0$
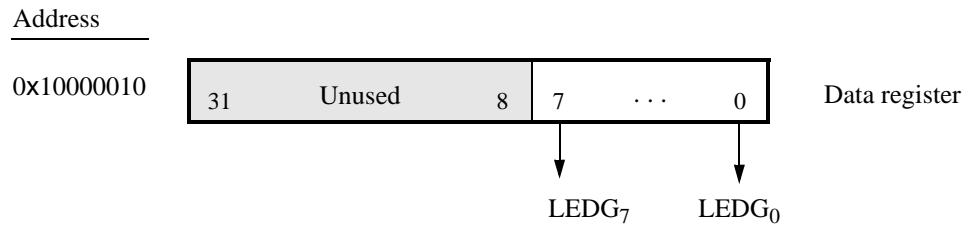
Figure 3. Output parallel ports for *LEDR* and *LEDG*.

2.3.2 DIP Switch Parallel Port

The SW_{3-0} DIP switches on the DE0-Nano board are connected to an input parallel port. As illustrated in Figure 4, this port comprises an four-bit read-only *Data* register, which is mapped to address 0x10000040.

Address

0x10000040

| 31 | Unused | 4 | 3 | $\cdots$ | 0 | Data register

$\cdots$

SW$_3$ SW$_0$
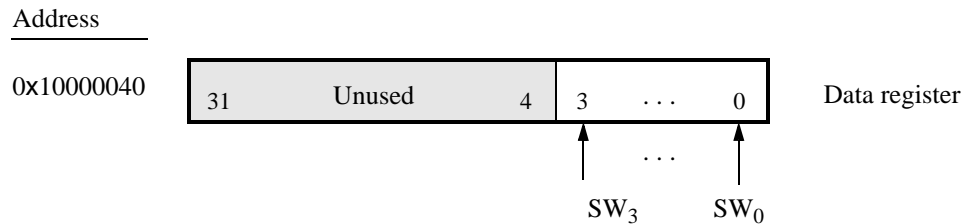
Figure 4. *Data* register in the DIP switch parallel port.

2.3.3 Pushbutton Parallel Port

The parallel port connected to the KEY_1 pushbutton switch on the DE0-Nano board comprises three 1-bit registers, as shown in Figure 5. These registers have base addresses 0x10000050 to 0x1000005C and can be accessed using word operations. The read-only *Data* register provides the value of the KEY_1 switch. Bit 0 of the data register is not used, because, as discussed in section 2.1, the corresponding switch KEY_0 is reserved for use as a reset mechanism for the DE0-Nano Basic Computer. The other two registers shown in Figure 5, at addresses 0x10000058 and 0x1000005C, are discussed in section 3.

2.3.4 Expansion Parallel Ports JP1 and JP2

The DE0-Nano Basic Computer includes two bidirectional parallel ports that are connected to the *JP1* and *JP2* expansion headers on the DE0-Nano 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 6 gives a diagram of the *JP1* and *JP2* expansion connectors on the DE0-Nano 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 right corner of the connector, bit D_1 is assigned below this, and so on. Note that some of the pins on *JP1* and *JP2* are not usable as input/output

Address	31	30	· · ·	2	1	0	
0x10000050			Unused		KEY_1		Data register
Unused				Unused			
0x10000058			Unused		Mask bits		Interruptmask register
0x1000005C			Unused		Edge bits		Edgecapture register

Figure 5. Registers used in the pushbutton parallel port.

connections, 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.
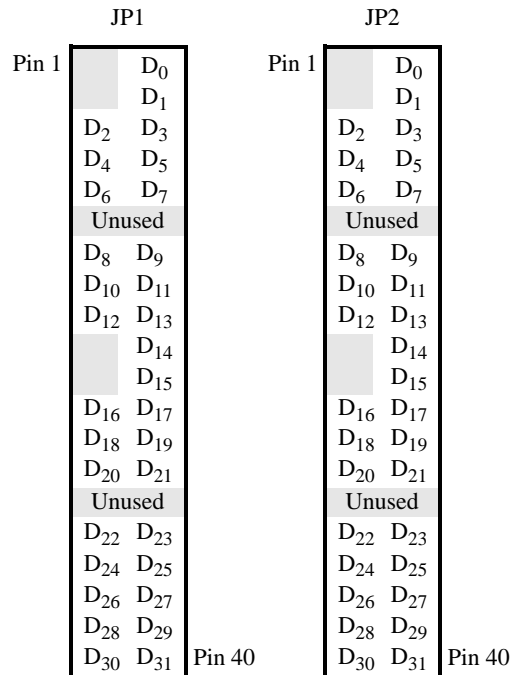
Figure 6. Assignment of parallel port bits to pins on *JP1* and *JP2*.

2.3.5 Expansion Parallel Ports JP3

The DE0-Nano Basic Computer includes two parallel ports that are connected to the *JP3* expansion header on the DE0-Nano board. One of these parallel ports provides 13 bidirectional connections, and the other port provides three input-only connections. Each of the parallel ports includes the four 32-bit registers that were described previously for Figure 2. The base address of the 13-bit bidirectional port connected to *JP3* is 0x10000080 and the base address of the input-only port is 0x10000090. Figure 7 gives a diagram of the *JP3* expansion connector on the DE0-Nano

board, and shows how both the bidirectional and input-only parallel ports are connected. The figure shows that bits D_0 to D_{12} of the bidirectional parallel port are assigned to pins 5 to 17 of the *JP3* connector. Bits IN_0 to IN_2 of the input-only parallel port are assigned to pins 2 to 4 of the connector.

Figure 7. Assignment of parallel port bits to pins on *JP3*.

2.3.6   Using the Parallel Ports with Assembly Language Code and C Code

The DE0-Nano Basic Computer provides a convenient platform for experimenting with Nios II assembly language code, or C code. A simple example of such code is provided in Figures 8 and 9. Both programs perform the same operations, and illustrate the use of parallel ports by using either assembly language or C code.

The code in the figures displays a rotating pattern on the green LEDG lights. This pattern is shifted to the right by using a Nios II *rotate* instruction, and a delay loop is used to make the shifting slow enough to observe. The pattern on the LEDG lights can be changed using the settings on the SW DIP switches on the DE0-Nano board by pressing pushbutton KEY_1 (recall from section 2.1 that KEY_0 causes a reset of the DE0-Nano Basic Computer). When a pushbutton key is pressed, the program waits in a loop until the key is released.

The source code files shown in Figures 8 and 9 are distributed as part of the Intel FPGA Monitor Program. The files can be found under the heading *sample programs*, and are identified by the name *Getting Started*.

```
/***************************************************************************
 * This program demonstrates use of parallel ports in the DE0-Nano Basic Computer
 * It performs the following:
 *       1. displays a rotating pattern on the green LEDG
 *       2. if KEY[1] is pressed, uses the SW switches as the pattern
 ***************************************************************************/
        .text                           /* executable code follows */
        .global _start
_start:
        /* initialize base addresses of parallel ports */
        movia     r16, 0x10000010       /* green LED base address */
        movia     r15, 0x10000040       /* SW slider switch (DIP switches) base address */
        movia     r17, 0x10000050       /* pushbutton KEY base address */
        movia     r19, LEDG_bits
        ldw       r6, 0(r19)            /* load pattern for LEDG lights */
DO_DISPLAY:
        ldwio     r4, 0(r15)            /* load slider (DIP) switches */
        ldwio     r5, 0(r17)            /* load pushbuttons */
        beq       r5, r0, NO_BUTTON
        mov       r6, r4               /* use SW (DIP switch) values on LEDG */
        roli      r4, r4, 8
        or        r6, r6, r4
        roli      r4, r4, 8
        or        r6, r6, r4
        roli      r4, r4, 8
        or        r6, r6, r4
WAIT:
        ldwio     r5, 0(r17)            /* load pushbuttons */
        bne       r5, r0, WAIT         /* wait for button release */
NO_BUTTON:
        stwio     r6, 0(r16)            /* store to LEDG */
        roli      r6, r6, 1            /* rotate the displayed pattern */
        movia     r7, 150000           /* delay counter */
DELAY:
        subi      r7, r7, 1
        bne       r7, r0, DELAY
        br        DO_DISPLAY

        .data                           /* data follows */
LEDG_bits:
        .word     0x0F0F0F0F
        .end
```

Figure 8. An example of Nios II assembly language code that uses parallel ports.

```
/****************************************************************************
 * This program demonstrates use of parallel ports in the DE0-Nano Basic Computer
 * It performs the following:
 *      1. displays a rotating pattern on the green LEDG
 *      2. if KEY[1] is pressed, uses the SW switches as the pattern
 ****************************************************************************/
int main(void)
{
    /* Declare volatile pointers to I/O registers (volatile means that IO load and store
       instructions (e.g., ldwio, stwio) will be used to access these pointer locations) */
    volatile int * green_LED_ptr    = (int *) 0x10000010;     // green LED address
    volatile int * SW_switch_ptr    = (int *) 0x10000040;     // SW DIP switch address
    volatile int * KEY_ptr          = (int *) 0x10000050;     // pushbutton KEY address

    int LEDG_bits = 0x0F0F0F0F;                      // pattern for LEDG lights
    int SW_value, KEY_value;
    volatile int delay_count;                        // volatile so C compile does not remove loop

    while(1)
    {
        SW_value = *(SW_switch_ptr);                 // read the SW DIP switch values

        KEY_value = *(KEY_ptr);                      // read the pushbutton KEY values
        if (KEY_value != 0)                          // check if any KEY was pressed
        {
            /* set pattern using SW values */
            LEDG_bits = SW_value | (SW_value << 8) | (SW_value << 16) | (SW_value << 24);
            while (*KEY_ptr);                        // wait for pushbutton KEY release
        }
        *(green_LED_ptr) = LEDG_bits;                // light up the green LEDs

        /* rotate the pattern shown on the LEDs */
        if (LEDG_bits & 0x80000000)
            LEDG_bits = (LEDG_bits << 1) | 1;
        else
            LEDG_bits = LEDG_bits << 1;

        for (delay_count = 150000; delay_count != 0; --delay_count); // delay loop
    }
}
```

Figure 9. An example of C code that uses parallel ports.

2.4   JTAG Port

The JTAG port implements a communication link between the DE0-Nano board and its host computer. This link is automatically used by the Quartus Prime software to transfer FPGA programming files into the DE0-Nano board, and by the Intel FPGA 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 10. The register mapped to address 0x10001000 is called the *Data* register and the register mapped to address 0x10001004 is called the *Control* register.

Address	31 ··· 16	15	14···11 10 9 8	7 ··· 1 0	
0x10001000	RAVAIL	RVALID	Unused	DATA	Data register
0x10001004	WSPACE	Unused	AC WI RI	WE RE	Control register

Figure 10. JTAG UART registers.

When character data from the host computer is received by the JTAG UART 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 10. 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.

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.

The *Control* register bits *RE*, *WE*, *RI*, and *WI* are described in section 3.

2.4.1   Using the JTAG UART with Assembly Language Code and C Code

Figures 11 and 12 give simple examples of assembly language and C code, respectively, that use the JTAG UART. Both versions of the code perform the same function, which is to first send an ASCII string to the JTAG UART, and then enter an endless loop. In the loop, the code reads character data that has been received by the JTAG UART, and echoes this data back to the UART for transmission. If the program is executed by using the Intel FPGA Monitor Program, then any keyboard character that is typed into the *Terminal Window* of the Monitor Program will be echoed

back, causing the character to appear in the *Terminal Window*.

The source code files shown in Figures 11 and 12 are made available as part of the Intel FPGA Monitor Program. The files can be found under the heading *sample programs*, and are identified by the name *JTAG UART*.

```
/*****************************************************************************
 * This program demonstrates use of the JTAG UART port in the DE0-Nano Basic Computer
 *
 * It performs the following:
 *     1. sends a text string to the JTAG UART
 *     2. reads character data from the JTAG UART
 *     3. echos the character data back to the JTAG UART
 *****************************************************************************/
        .text                           /* executable code follows */
        .global     _start
_start:
    /* set up stack pointer */
        movia       sp, 0x01FFFFFC      /* stack starts from highest memory address in SDRAM */

        movia       r6, 0x10001000      /* JTAG UART base address */

    /* print a text string */
        movia       r8, TEXT_STRING
LOOP:
        ldb         r5, 0(r8)
        beq         r5, zero, GET_JTAG  /* string is null-terminated */
        call        PUT_JTAG
        addi        r8, r8, 1
        br          LOOP

    /* read and echo characters */
GET_JTAG:
        ldwio       r4, 0(r6)           /* read the JTAG UART data register */
        andi        r8, r4, 0x8000      /* check if there is new data */
        beq         r8, r0, GET_JTAG    /* if no data, wait */
        andi        r5, r4, 0x00ff      /* the data is in the least significant byte */

        call        PUT_JTAG            /* echo character */
        br          GET_JTAG
        .end
```

Figure 11. An example of assembly language code that uses the JTAG UART (Part *a*).

```
/*********************************************************************************
 * Subroutine to send a character to the JTAG UART
 *      r5   = character to send
 *      r6   = JTAG UART base address
 ********************************************************************************/
    .global    PUT_JTAG
PUT_JTAG:
    /* save any modified registers */
    subi       sp, sp, 4               /* reserve space on the stack */
    stw        r4, 0(sp)               /* save register */

    ldwio      r4, 4(r6)               /* read the JTAG UART Control register */
    andhi      r4, r4, 0xffff          /* check for write space */
    beq        r4, r0, END_PUT         /* if no space, ignore the character */
    stwio      r5, 0(r6)               /* send the character */

END_PUT:
    /* restore registers */
    ldw        r4, 0(sp)
    addi       sp, sp, 4

    ret

    .data                              /* data follows */
TEXT_STRING:
    .asciz "\nJTAG UART example code\n> "

    .end
```

Figure 11. An example of assembly language code that uses the JTAG UART (Part *b*).

```c
void put_jtag(volatile int *, char);                    // function prototype

/*****************************************************************************
 * This program demonstrates use of the JTAG UART port in the DE0-Nano Basic Computer
 *
 * It performs the following:
 *      1. sends a text string to the JTAG UART
 *      2. reads character data from the JTAG UART
 *      3. echos the character data back to the JTAG UART
 *****************************************************************************/
int main(void)
{
    /* Declare volatile pointers to I/O registers (volatile means that IO load and store
       instructions (e.g., ldwio, stwio) will be used to access these pointer locations) */
    volatile int * JTAG_UART_ptr = (int *) 0x10001000;        // JTAG UART address
    int data, i;
    char text_string[ ] = "\nJTAG UART example code\n> \0";

    for (i = 0; text_string[i] != 0; ++i)                   // print a text string
        put_jtag (JTAG_UART_ptr, text_string[i]);

    /* read and echo characters */
    while(1)
    {
        data = *(JTAG_UART_ptr);                            // read the JTAG_UART data register
        if (data & 0x00008000)                              // check RVALID to see if there is new data
        {
            data = data & 0x000000FF;                       // the data is in the least significant byte
            /* echo the character */
            put_jtag (JTAG_UART_ptr, (char) data & 0xFF );
        }
    }
}
/*****************************************************************************
 * Subroutine to send a character to the JTAG UART
 *****************************************************************************/
void put_jtag( volatile int * JTAG_UART_ptr, char c )
{
    int control;
    control = *(JTAG_UART_ptr + 1);                         // read the JTAG_UART Control register
    if (control & 0xFFFF0000)                               // if space, then echo character, else ignore
        *(JTAG_UART_ptr) = c;
}
```

Figure 12. An example of C code that uses the JTAG UART.

2.5   Interval Timer

The DE0-Nano 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 DE0-Nano board. The programming interface for the timer includes six 16-bit registers, as illustrated in Figure 13. 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:

- *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.

- *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.

- *ITO* is used for generating Nios II interrupts, which are discussed in section 3.

Address	31 · · · 17	16	15 · · ·	3	2	1	0	
0x10002000			Unused			RUN	TO	Status register
0x10002004			Unused	STOP	START	CONT	ITO	Control register
0x10002008	Not present (interval timer has 16-bit registers)		Counter start value (low)					
0x1000200C			Counter start value (high)					
0x10002010			Counter snapshot (low)					
0x10002014			Counter snapshot (high)					

Figure 13. Interval timer registers.

- *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.

- (*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 DE0-Nano Basic Computer gives a timer period of 125 msec. To achieve this period, the starting value of the count is 50 MHz × 125 msec = 6.25×10^6. 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.

2.6 System ID

The system ID module provides a unique value that identifies the DE0-Nano Basic Computer system. The host computer connected to the DE0-Nano 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 DE0-Nano Basic Computer has been properly downloaded onto the DE0-Nano board. This process allows debugging tools on the host computer, such as the Intel FPGA Monitor Program, to verify that the DE0-Nano 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 DE0-Nano Basic Computer is set to 0x00000000. The address used for all other general exceptions, such as divide by zero, and hardware IRQ interrupts is 0x00000020. 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 DE0-Nano Basic Computer.

I/O Peripheral	IRQ #
Interval timer	0
Pushbutton switch parallel port	1
JTAG port	8
Serial port	10
JP1 Expansion parallel port	11
JP2 Expansion parallel port	12
JP3 Expansion parallel port (bidirectional)	13
JP3 Expansion parallel port (input-only)	14

Table 1. Hardware IRQ interrupt assignment for the DE0-Nano Basic Computer.

3.1 Interrupts from Parallel Ports

Parallel port registers in the DE0-Nano Basic Computer were illustrated in Figure 2, which is reproduced as Figure 14. As the figure shows, parallel ports that support interrupts include two related registers at the addresses *Base* + 8 and *Base* + C. The *Interruptmask* register, which has the address *Base* + 8, 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.

Address	31	30	· · ·	4	3	2	1	0	
Base	Input or output data bits								Data register
Base + 4	Direction bits								Direction register
Base + 8	Mask bits								Interruptmask register
Base + C	Edge bits								Edgecapture register

Figure 14. Registers used for interrupts from the parallel ports.

3.1.1 Interrupts from the Pushbutton Switches

Figure 5, reproduced as Figure 15, shows the registers associated with the pushbutton parallel port. The *Interrupt-mask* register allows processor interrupts to be generated when a key is pressed. Each bit in the *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 *Edgecapture* register deasserts the Nios II interrupt request and sets all bits of the *Edgecapture* register to zero.

Address	31	30	· · ·	4	3	2	1	0	
0x10000050	Unused				KEY_{2-1}				Data register
Unused	Unused								
0x10000058	Unused				Mask bits				Interruptmask register
0x1000005C	Unused				Edge bits				Edgecapture register

Figure 15. Registers used for interrupts from the pushbutton parallel port.

3.2   Interrupts from the JTAG UART

Figure 10, reproduced as Figure 16, shows the *Data* and *Control* registers of the JTAG UART. As we said in section 2.4, *RAVAIL* in the data register gives the number of characters that are stored in the receive FIFO, and *WSPACE* gives the amount of unused space that is available in the transmit FIFO. The *RE* and *WE* bits in Figure 16 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.

Address	31 · · · 16	15	14 · · · 11 10 9 8	7 · · · 1 0	
0x10001000	RAVAIL	RVALID	Unused	DATA	Data register
0x10001004	WSPACE	Unused	AC WI RI	WE RE	Control register

Figure 16. Interrupt bits in the JTAG UART registers.

3.3   Interrupts from the Interval Timer

Figure 13, in section 2.5, shows six registers that are associated with the interval timer. As we said in section 2.5, the bit b_0 (*TO*) is set to 1 when the timer reaches a count value of 0. It is possible to generate an interrupt when this occurs, by using the bit b_{16} (*ITO*). Setting the bit *ITO* to 1 allows an interrupt request to be generated whenever *TO* becomes 1. After an interrupt occurs, it can be cleared by writing any value to the register that contains the bit *TO*.

3.4   Using Interrupts with Assembly Language Code

An example of assembly language code for the DE0-Nano Basic Computer that uses interrupts is shown in Figure 17. When this code is executed on the DE0-Nano board it displays a rotating pattern on the LEDG green lights. The direction in which the pattern rotates is reversed whenever the pushbutton KEY_1 is pressed. The LEDG green lights are controlled by an interrupt service routine for the interval timer, and another interrupt service routine is used to handle the pushbutton key. The speed at which the lights are rotated is set in the main program, by using a counter value in the interval timer that causes an interrupt to occur every 66 msec.

```
.equ        KEY1, 0
.equ        KEY2, 1
/*******************************************************************************
* This program demonstrates use of interrupts in the DE0-Nano Basic Computer. It
* first starts the interval timer with 66 msec timeouts, and then enables
* Nios II interrupts from the interval timer and pushbutton KEYs
*
* The interrupt service routine for the interval timer displays a pattern on
* the LEDG lights, and shifts this pattern either left or right. The shifting
* direction is set in the pushbutton interrupt service routine; it is reversed
* each time KEY[1] is pressed
*******************************************************************************/
        .text                           /* executable code follows */
        .global    _start
_start:
    /* set up stack pointer */
    movia       sp, 0x01FFFFFC          /* stack starts from highest memory address in SDRAM */

    movia       r16, 0x10002000         /* internal timer base address */
    /* set the interval timer period for scrolling the LEDG lights */
    movia       r12, 0x320000           /* 1/(50 MHz) × (0x320000) = 66 msec */
    sthio       r12, 8(r16)             /* store the low halfword of counter start value */
    srli        r12, r12, 16
    sthio       r12, 0xC(r16)           /* high halfword of counter start value */

    /* start interval timer, enable its interrupts */
    movi        r15, 0b0111             /* START = 1, CONT = 1, ITO = 1 */
    sthio       r15, 4(r16)
```

Figure 17. An example of assembly language code that uses interrupts (Part *a*).

```
            /* write to the pushbutton port interrupt mask register */
    movia       r15, 0x10000050        /* pushbutton key base address */
    movi        r7, 0b010              /* set KEY[1] interrupt mask bit (KEY[0] is Nios II reset) */
    stwio       r7, 8(r15)             /* interrupt mask register is (base + 8) */


            /* enable Nios II processor interrupts */
    movi        r7, 0b011              /* set interrupt mask bits for levels 0 (interval */
    wrctl       ienable, r7            /* timer) and level 1 (pushbuttons) */
    movi        r7, 1
    wrctl       status, r7             /* turn on Nios II interrupt processing */

IDLE:
    br          IDLE                   /* main program simply idles */


    .data
/* The two global variables used by the interrupt service routines for the interval timer and the
 * pushbutton keys are declared below */


    .global     PATTERN
PATTERN:
    .word       0x0F0F0F0F             /* pattern to show on the LEDG lights */


    .global     KEY_DIR
KEY_DIR:
    .word       0


    .end
```

Figure 17. An example of assembly language code that uses interrupts (Part *b*).

The reset and exception handlers for the main program in Figure 17 are given in Figure 18. The reset handler simply jumps to the *_start* symbol in the main program. The exception handler first checks if the exception that has occurred is an external interrupt or an internal one. In the case of an internal exception, such as an illegal instruction opcode or a trap instruction, the handler simply exits, because it does not handle these cases. For external exceptions, it calls either the interval timer interrupt service routine, for a level 0 interrupt, or the pushbutton key interrupt service routine for level 1. These routines are shown in Figures 19 and 20, respectively.

```
/******************************************************************************
 * RESET SECTION
 * The Monitor Program automatically places the ".reset" section at the reset location
 * specified in the CPU settings in Qsys.
 * Note: "ax" is REQUIRED to designate the section as allocatable and executable.
 */
    .section    .reset, "ax"
    movia       r2, _start
    jmp         r2                          /* branch to main program */


/******************************************************************************
 * EXCEPTIONS SECTION
 * The Monitor Program automatically places the ".exceptions" section at the
 * exception location specified in the CPU settings in Qsys.
 * Note: "ax" is REQUIRED to designate the section as allocatable and executable.
 */
    .section    .exceptions, "ax"
    .global     EXCEPTION_HANDLER
EXCEPTION_HANDLER:
    subi        sp, sp, 16                  /* make room 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 after eret */

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         r22, 12(sp)

    rdctl       et, ctl4
    bne         et, r0, CHECK_LEVEL_0       /* exception is an external interrupt */

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

Figure 18. Reset and exception handler assembly language code (Part *a*).

```
CHECK_LEVEL_0:                      /* interval timer is interrupt level 0 */
    andi     r22, et, 0b1
    beq      r22, r0, CHECK_LEVEL_1
    call     INTERVAL_TIMER_ISR
    br       END_ISR

CHECK_LEVEL_1:                      /* pushbutton port is interrupt level 1 */
    andi     r22, et, 0b10
    beq      r22, r0, END_ISR       /* other interrupt levels are not handled in this code */
    call     PUSHBUTTON_ISR

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      r22, 12(sp)
    addi     sp, sp, 16

    eret
    .end
```

Figure 18. Reset and exception handler assembly language code (Part *b*).

```
    .extern    PATTERN               /* externally defined variables */
    .extern    KEY_DIR
/***************************************************************************
 * Interval timer interrupt service routine
 *
 * Shifts a PATTERN being displayed on the LEDG lights. The shift direction
 * is determined by the external variable KEY_DIR.
 *
 ***************************************************************************/
    .global    INTERVAL_TIMER_ISR
INTERVAL_TIMER_ISR:
    subi     sp, sp, 36            /* reserve space on the stack */
    stw      ra, 0(sp)
    stw      r4, 4(sp)
    stw      r5, 8(sp)
    stw      r6, 12(sp)
```

Figure 19. Interrupt service routine for the interval timer (Part *a*).

```
        stw        r8, 16(sp)
        stw        r10, 20(sp)
        stw        r20, 24(sp)
        stw        r21, 28(sp)
        stw        r22, 32(sp)

        movia      r10, 0x10002000        /* interval timer base address */
        sthio      r0, 0(r10)             /* clear the interrupt */

        movia      r20, 0x10000010        /* LEDG base address */
        addi       r5, r0, 1              /* set r5 to the constant value 1 */
        movia      r21, PATTERN           /* set up a pointer to the pattern for LEDG lights */
        movia      r22, KEY_DIR           /* set up a pointer to the direction variable */

        ldw        r6, 0(r21)             /* load pattern for LEDG lights */
        stwio      r6, 0(r20)             /* store to LEDG */

        ldw        r4, 0(r22)             /* check which direction is active */
        beq        r4, r0, RIGHT          /* for KEY_DIR == 0, shift right */
        rol        r6, r6, r5             /* else shift left */
        br         END_INTERVAL_TIMER_ISR
RIGHT:
        ror        r6, r6, r5             /* rotate the displayed pattern right */

END_INTERVAL_TIMER_ISR:
        stw        r6, 0(r21)             /* store LEDG light pattern */
        ldw        ra, 0(sp)              /* Restore all used register to previous */
        ldw        r4, 4(sp)
        ldw        r5, 8(sp)
        ldw        r6, 12(sp)
        ldw        r8, 16(sp)
        ldw        r10, 20(sp)
        ldw        r20, 24(sp)
        ldw        r21, 28(sp)
        ldw        r22, 32(sp)
        addi       sp, sp, 36             /* release the reserved space on the stack */
        ret
        .end
```

Figure 19. Interrupt service routine for the interval timer (Part *b*).

```
    .extern     KEY_DIR
/************************************************************************
 * Pushbutton - Interrupt Service Routine
 *
 * This routine toggles the KEY_DIR variable from 0 <-> 1
 ************************************************************************/
    .global     PUSHBUTTON_ISR
PUSHBUTTON_ISR:
    subi        sp, sp, 16              /* reserve space on the stack */
    stw         ra, 0(sp)
    stw         r10, 4(sp)
    stw         r11, 8(sp)
    stw         r12, 12(sp)

    movia       r10, 0x10000050         /* base address of pushbutton KEY parallel port */
    /* only KEY[1] could have caused the interrupt on the DE0-Nano board; just clear it */
    stwio       r0, 0xC(r10)            /* clear the interrupt */

    movia       r11, KEY_DIR            /* global variable to return the result */
    ldw         r12, 0(r11)             /* load current KEY_DIR value */
    xori        r12, r12, 1             /* toggle value */
    stw         r12, 0(r11)             /* store the toggled value */

END_PUSHBUTTON_ISR:
    ldw         ra, 0(sp)               /* Restore all used register to previous values */
    ldw         r10, 4(sp)
    ldw         r11, 8(sp)
    ldw         r12, 12(sp)
    addi        sp, sp, 16
    ret
    .end
```

Figure 20. Interrupt service routine for the pushbutton keys.

3.5   Using Interrupts with C Language Code

An example of C language code for the DE0-Nano Basic Computer that uses interrupts is shown in Figure 21. This code performs exactly the same operations as the code described in Figure 17.

To enable interrupts the code in Figure 21 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 22.

The reset and exception handlers for the main program in Figure 21 are given in Figure 23. 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 23, the *interrupt_handler* code first determines which exception has occurred, by using a macro from Figure 22 that reads the content of the Nios II interrupt pending register. The interrupt service routine that is invoked for the interval timer is shown in 24, and the interrupt service routine for the pushbutton switches appears in Figure 25.

The source code files shown in Figure 17 to Figure 25 are distributed as part of the Intel FPGA Monitor Program. The files can be found under the heading *sample programs*, and are identified by the name *Interrupt Example*.

```
#include "nios2_ctrl_reg_macros.h"

/* key_dir and pattern are written by interrupt service routines; we have to declare these as
 * volatile to avoid the compiler caching their values in registers */
volatile int key_dir = 0;
volatile int pattern = 0x0F0F0F0F;      // pattern for LEDG lights
/*****************************************************************************
 * This program demonstrates use of interrupts in the DE0-Nano Basic Computer. It first starts the
 * interval timer with 66 msec timeouts, and then enables interrupts from the interval timer
 * and pushbutton KEYs
 *
 * The interrupt service routine for the interval timer displays a pattern on the LEDG lights and
 * shifts this pattern either left or right. The shifting direction is reversed when KEY[1] is pressed
 *****************************************************************************/
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 * interval_timer_ptr = (int *) 0x10002000;    // interval timer base address
    volatile int * KEY_ptr = (int *) 0x10000050;               // pushbutton KEY address

    /* set the interval timer period for scrolling the LEDG lights */
    int counter = 0x320000;                 // 1/(50 MHz) × (0x320000) = 66 msec
    *(interval_timer_ptr + 0x2) = (counter & 0xFFFF);
    *(interval_timer_ptr + 0x3) = (counter >> 16) & 0xFFFF;

    /* start interval timer, enable its interrupts */
    *(interval_timer_ptr + 1) = 0x7;        // STOP = 0, START = 1, CONT = 1, ITO = 1

    *(KEY_ptr + 2) = 0x2;                   /* write to the pushbutton interrupt mask register, and
                                             * set KEY[1] mask bit to 1 (KEY[0] is Nios II reset) */

    NIOS2_WRITE_IENABLE( 0x3 );             /* set interrupt mask bits for levels 0 (interval timer)
                                             * and level 1 (pushbuttons) */
    NIOS2_WRITE_STATUS( 1 );                // enable Nios II interrupts

    while(1);                               // main program simply idles
}
```

Figure 21. An example of C code that uses interrupts.

```
#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 22. Macros for accessing Nios II status and control registers.

```
#include "nios2_ctrl_reg_macros.h"

/* function prototypes */
void main(void);
void interrupt_handler(void);
void interval_timer_ISR(void);
void pushbutton_ISR(void);


/* The assembly language code below handles Nios II reset processing */
void the_reset (void) __attribute__ ((section (".reset")));
void the_reset (void)
/*****************************************************************************
 * Reset code; by using the 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");                     /* Instruct the assembler NOT to use reg at (r1) as
                                                 * a temp register for performing optimizations */
    asm (".set      nobreak");                  /* Suppresses a warning message that says that some
                                                 * debuggers corrupt regs bt (r25) and ba (r30) */
    asm ("movia r2, _start");                   // call the C language main program
    asm ("jmp       r2");
}
/* The assembly language code below handles Nios II 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)
/*****************************************************************************
 * Exceptions 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 23. Reset and exception handler C 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 23. Reset and exception handler C language 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 the interrupt source and calls the appropriate subroutine
 *******************************************************************************/
void interrupt_handler(void)
{
   int ipending;
   NIOS2_READ_IPENDING(ipending);
   if ( ipending & 0x1 )                        // interval timer is interrupt level 0
        interval_timer_ISR( );
   if ( ipending & 0x2 )                        // pushbuttons are interrupt level 1
        pushbutton_ISR( );
   // else, ignore the interrupt
   return;
}
```

Figure 23. Reset and exception handler C code (Part *c*).

```
extern volatile int key_dir;
extern volatile int pattern;
/*******************************************************************************
 * Interval timer interrupt service routine
 *
 * Shifts a pattern being displayed on the LEDG lights. The shift direction is determined
 * by the external variable key_dir.
 *
 ******************************************************************************/
void interval_timer_ISR( )
{
    volatile int * interval_timer_ptr = (int *) 0x10002000;
    volatile int * LEDG_ptr = (int *) 0x10000010;                 // LEDG address

    *(interval_timer_ptr) = 0;                  // clear the interrupt

    *(LEDG_ptr) = pattern;                      // display pattern on LEDG

    /* rotate the pattern shown on the LEDG lights */
    if (key_dir == 0)                           // for 0 rotate left
        if (pattern & 0x80000000)
            pattern = (pattern << 1) | 1;
        else
            pattern = pattern << 1;
    else                                        // rotate right
        if (pattern & 0x00000001)
            pattern = (pattern >> 1) | 0x80000000;
        else
            pattern = (pattern >> 1) & 0x7FFFFFFF;

    return;
}
```

Figure 24. Interrupt service routine for the interval timer.

```
extern volatile int key_dir;
extern volatile int pattern;

/*****************************************************************************
 * Pushbutton - Interrupt Service Routine
 *
 * This routine toggles the key_dir variable from 0 <-> 1
 *****************************************************************************/
void pushbutton_ISR( void )
{
    volatile int * KEY_ptr = (int *) 0x10000050;

    // only KEY[1] can cause an interrupt on the DE0-Nano board; just clear it
    *(KEY_ptr + 3) = 0;                    // Clear the interrupt

    key_dir ^= 1;                          // Toggle key_dir value

    return;
}
```

Figure 25. Interrupt service routine for the pushbutton keys.

4   Modifying the DE0-Nano Basic Computer

It is possible to modify the DE0-Nano Basic Computer by using Intel's Quartus Prime software and Qsys System Integration tool. Tutorials that introduce this software are provided in the University Program section of Intel's web site. To modify the system it is first necessary to obtain all of the relevant design source code files. The DE0-Nano 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 provided in the form of Qsys IP cores and are included in a package available from Intel's FPGA University Program web site, called the *Intel FPGA University Program IP Cores*

Table 2 lists the names of the Qsys IP cores that are used in this system. When the DE0-Nano Basic Computer design files are opened in the Quartus Prime software, these cores can be examined using the Qsys System Integration tool. Each core has a number of settings that are selectable in the Qsys System Integration tool, and includes a datasheet that provides detailed documentation.

I/O Peripheral	Qsys Core
SDRAM	SDRAM Controller
On-chip Memory	On-Chip Memory (RAM or ROM)
Green LED parallel port	Parallel Port
Expansion parallel ports	Parallel Port
Slider switch parallel port	Parallel Port
Pushbutton parallel port	Parallel Port
JTAG port	JTAG UART
Interval timer	Interval timer
System ID	System ID Peripheral

Table 2. Qsys cores used in the DE0-Nano Basic Computer.

The steps needed to modify the system are:

1. Install the *University Program IP Cores* from Intel's FPGA University Program web site

2. Copy the design source files for the DE0-Nano Basic Computer from the University Program web site. These files can be found in the *Design Examples* section of the web site

3. Open the *DE0-Nano_Basic_Computer.qpf* project in the Quartus Prime software

4. Open the Qsys System Integration tool in the Quartus Prime software, and modify the system as desired

5. Generate the modified system by using the Qsys System Integration tool

6. It may be necessary to modify the Verilog or VHDL code in the top-level module, DE0-Nano_Basic_System.v/vhd, if any I/O peripherals have been added or removed from the system

7. Compile the project in the Quartus Prime software

8. Download the modified system into the DE0-Nano board

5   Making the System the Default Configuration

The DE0-Nano Basic Computer can be loaded into the nonvolatile FPGA configuration memory on the DE0-Nano board, so that it becomes the default system whenever the board is powered on. Instructions for configuring the DE0-Nano board in this manner can be found in the tutorial *Introduction to the Quartus Prime Software*, which is available from Intel's FPGA University Program.

6   Memory Layout

Table 3 summarizes the memory map used in the DE0-Nano Basic Computer.

Base Address	End Address	I/O Peripheral
0x00000000	0x01FFFFFF	SDRAM
0x08000000	0x08001FFF	On-chip Memory
0x09000000	0x09001FFF	On-chip Memory
0x10000010	0x1000001F	Green LED parallel port
0x10000040	0x1000004F	Slider switch parallel port
0x10000050	0x1000005F	Pushbutton parallel port
0x10000060	0x1000006F	JP1 Expansion parallel port
0x10000070	0x1000007F	JP2 Expansion parallel port
0x10000080	0x1000008F	JP3 Expansion parallel port (bidirectional)
0x10000090	0x1000009F	JP3 Expansion parallel port (input-only)
0x10001000	0x10001007	JTAG port
0x10002000	0x1000201F	Interval timer
0x10002020	0x10002027	System ID

Table 3. Memory layout used in the DE0-Nano Basic Computer.

7   Intel FPGA Monitor Program Integration

As we mentioned earlier, the DE0-Nano Basic Computer system, and the sample programs described in this document, are made available as part of the Intel FPGA Monitor Program. Figures 26 to 29 show a series of windows that are used in the Monitor Program to create a new project. In the first screen, shown in Figure 26, the user specifies a file system folder where the project will be stored, gives the project a name and chooses an architecture for the project. Pressing Next opens the window in Figure 27. Here, the user can select the DE0-Nano Basic Computer as a predesigned system. The Monitor Program then fills in the relevant information in the *System description file* box, which includes the files called *nios_system.sopcinfo* and the **.sof* file for the hardware system that has been selected. The first of these files specifies to the Monitor Program information about the components that are available in the hardware system, such as the type of processor and memory components, and the address map. The second file is an FPGA programming bitstream for the board, which can downloaded by the Monitor Program into the board.

Pressing Next again opens the window in Figure 28. Here the user selects the type of program that will be used, such as Assembly language, or C. Then, the check box shown in the figure can be used to display the list of sample
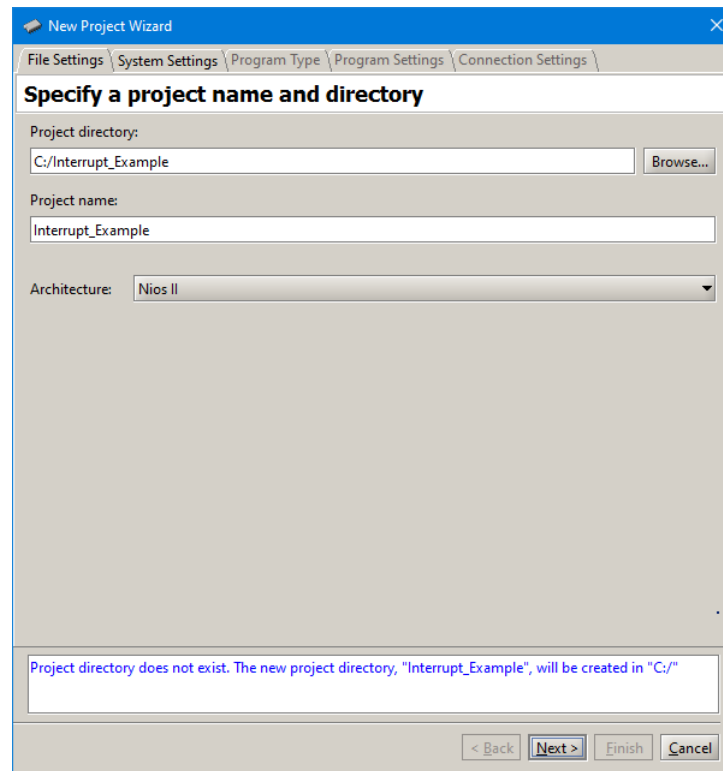
Figure 26. Specifying the project folder and project name.

programs for the selected board. When a sample program is selected in this list, its source files, and other settings, can be copied into the project folder in subsequent screens of the Monitor Program.

Figure 29 gives the final screen that is used to create a new project in the Monitor Program.This screen allows the user specify the linker sections and their addresses for the project. There are two modes that can be selected. In the Basic mode, which does not provide explicitly for the use of interrupts, the application program starts at memory address 0x00000000 as shown in the figure. A more general alternative is to use the Exceptions mode.The program in the *.text* section can start at some other address, as may be specified by the user. To change the address, double-click on the *.text* entry and change the address in the pop-up box that appears.
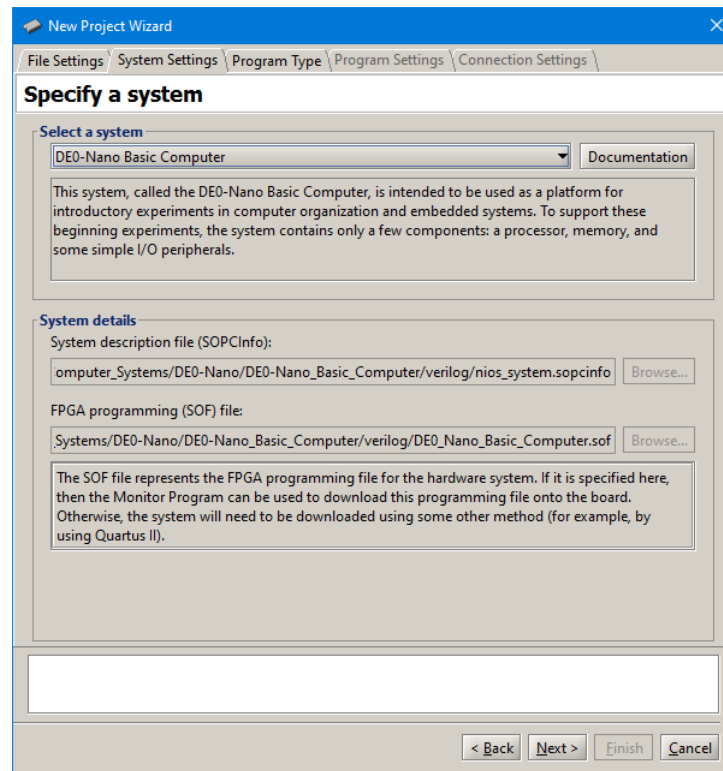
Figure 27. Specifying the Nios II system.
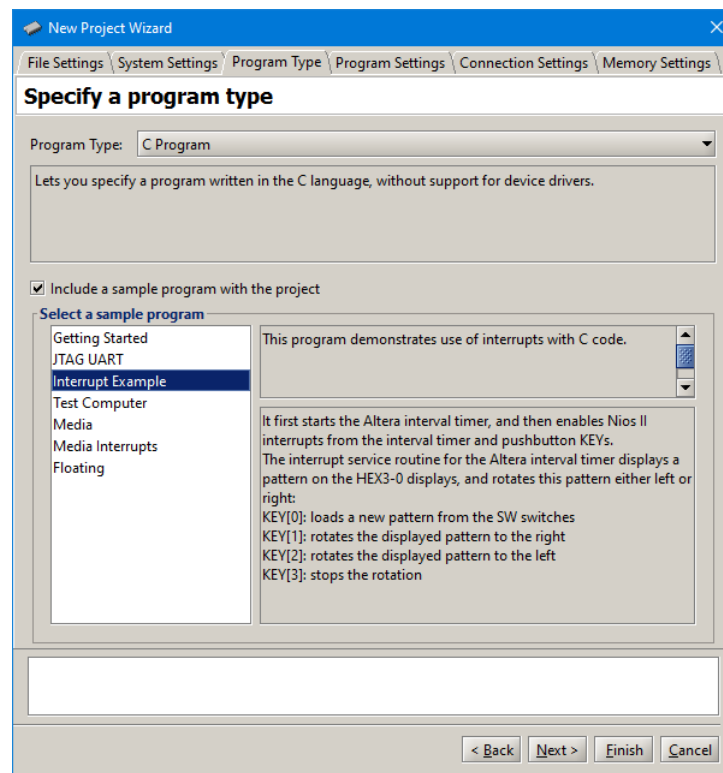
Figure 28. Selecting sample programs.
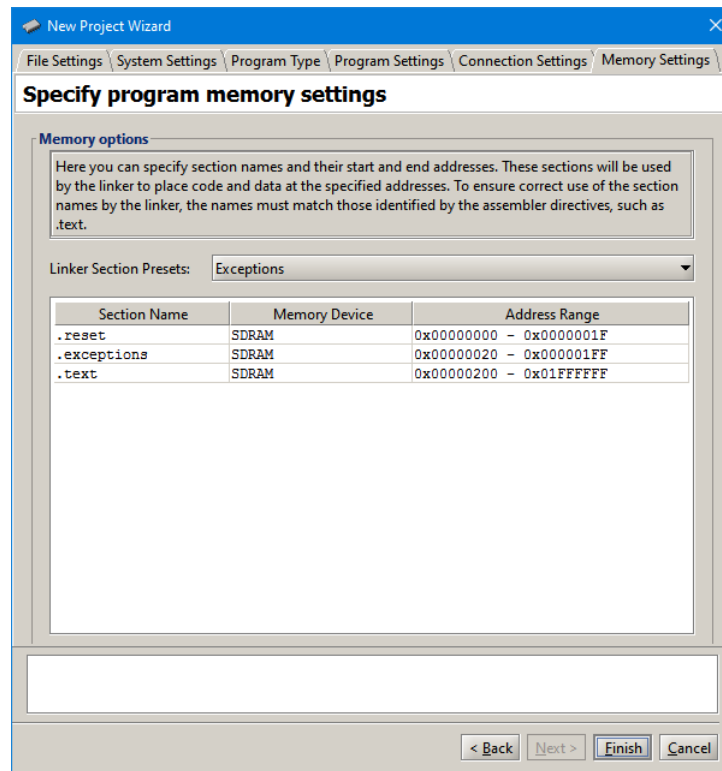
Figure 29. Setting for *.text*.