

1 Introduction

This document describes a computer system that can be implemented on the Intel® DE2-115 development and education board. This system, called the *DE2-115 Computer*, is intended for use in experiments on computer organization and embedded systems.

To support such experiments, the system contains embedded processors, memory, audio and video devices, 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 DE2-115 Computer Contents

A block diagram of the DE2-115 Computer is shown in Figure 1. Its main components include the Intel Nios® II processor, memory for program and data storage, an audio-in/out port, a video-out port with both pixel and character buffers, a video-in port, a PS/2 serial port, a 16 × 2 character display, parallel ports connected to switches and lights, a timer module, an SD Card controller, an IrDA* sender and receiver, and an RS 232 serial port. As shown in the figure, the processor and its interfaces to I/O devices are implemented inside the Cyclone® IV FPGA chip on the DE2-115 board. A number of the components shown in Figure 1 are described in the remainder of this section, and the others are presented in section 4.

2.1 Nios® II Processor

The Intel Nios II processor is a 32-bit CPU that can be implemented in an Intel FPGA device. Two versions of the Nios II processor are available, designated economy (*/e*) and fast (*/f*). The DE2-115 Computer includes two instances of the Nios II/*f* version, configured with floating-point hardware support.

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's web site. An easy way to begin working with the DE2-115 Computer and the Nios II processor is to make use of a utility called the *Intel® FPGA Monitor Program*. It provides an easy way to assemble/compile Nios II programs written in either assembly language or the C 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 DE2-115 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 DE2-115 board, and similar operations. The Monitor Program includes the DE2-115 Computer as a predesigned system that can be downloaded onto the DE2-115 board, as well as several sample programs in assembly language and C that show how to use the DE2-115 Computer's peripherals. Some images that show how the DE2-115 Computer is integrated with the

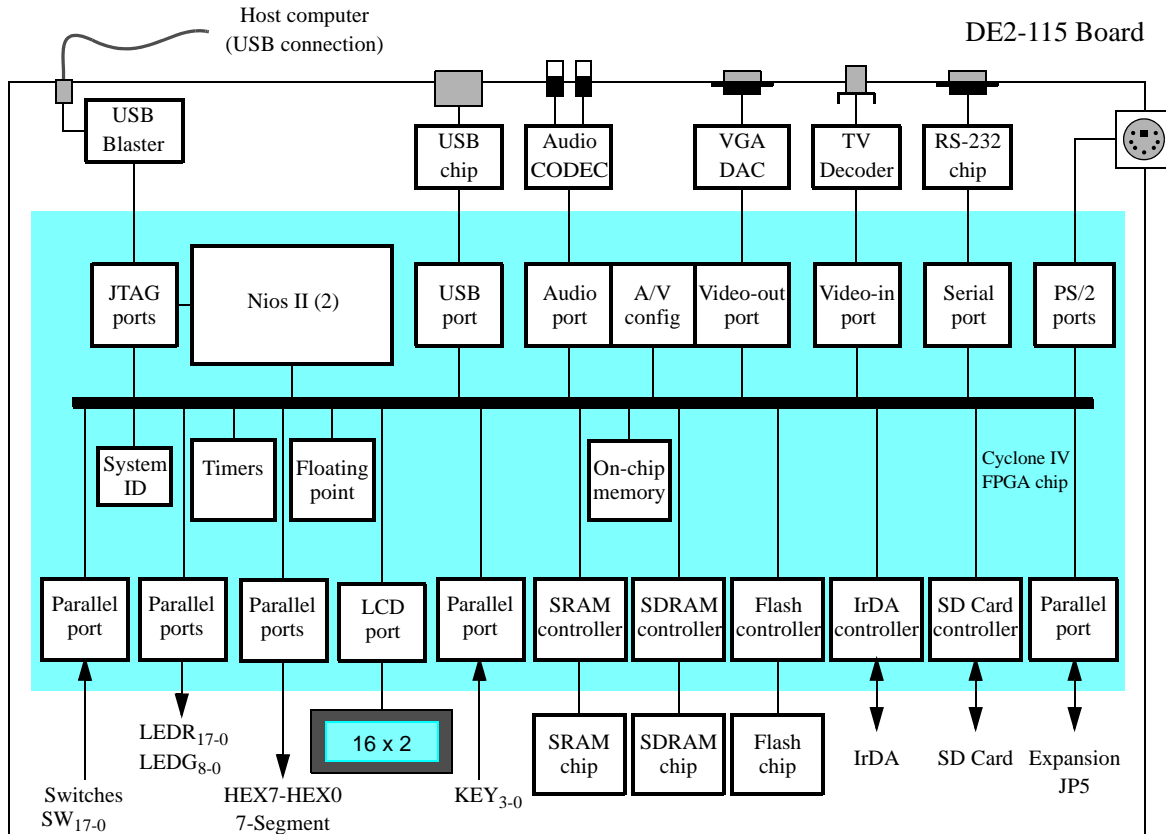


Figure 1. Block diagram of the DE2-115 Computer.

Monitor Program are described in Section 8. An overview of the Monitor Program is available in the document *Intel® FPGA Monitor Program Tutorial*, which is provided in the University Program web site.

All of the I/O peripherals in the DE2-115 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 DE2-115 Computer has three types of memory components: SDRAM, SRAM, and on-chip memory inside the FPGA chip. Each type of memory is described below.

2.2.1 SDRAM

An SDRAM Controller provides an interface to the 128 MB synchronous dynamic RAM (SDRAM) on the DE2-115 board, which is organized as 32M x 32 bits. It is accessible by the Nios II processor using word (32-bit), halfword (16-bit), or byte operations, and is mapped to the address space 0x00000000 to 0x07FFFFFFF.

2.2.2 SRAM

An SRAM Controller provides a 32-bit interface to the static RAM (SRAM) chip on the DE2-115 board. This SRAM chip is organized as 1M x 16 bits, but is accessible by the Nios II processor using word (32-bit), halfword (16-bit), or byte operations. The SRAM memory is mapped to the address space 0x08000000 to 0x081FFFFFF.

2.2.3 On-Chip Memory

The DE2-115 Computer includes a 8-Kbyte memory that is implemented in the Cyclone IV FPGA chip. This memory is organized as 8K x 8 bits, and spans addresses in the range 0x09000000 to 0x09001FFF. This memory is used as a character buffer for the video-out port, which is described in section 4.2.

2.3 SD Card

The DE2-115 Computer includes a controller for reading and writing to an SD Card. It supports SD Cards of sizes up to 4-Gbytes. The SD Card is accessible in 512-byte blocks. The currently selected block is mapped to a 512-byte buffer at address space 0x0B000000 to 0x0B0001FF.

Accessing a 512-byte block for reading is done by writing the address of the block on the SD Card to 0x0B00022C, followed by the *READ_BLOCK* command (a value of 0x11) to 0x0B000230. The data will then be available for reading. To write data to the SD Card, load the desired data into the the buffer, write the address of the block on the SD Card to 0x0B00022C, and write the *WRITE_BLOCK* command (a value of 0x18) to 0x0B000230.

A HAL interface is also available for accessing a filesystem stored on an SD Card. SD Cards used this way must be formatted using the FAT-16 standard.

2.4 Flash

The DE2-115 Computer can access to a 8-Mbyte Flash memory, which is word, halfword, and byte addressable. This memory is not intended to be used for operations requiring frequent writes, and should instead be used for data storage. It is mapped to the address space 0x0C000000 to 0x0C7FFFFFFF. The Flash memory also includes a 32-bit erase-control register, which is located at 0x0BFF0000 to 0x0BFF0003.

2.5 Parallel Ports

There are several parallel ports implemented in the FPGA 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.

Some of the parallel ports in the DE2-115 Computer have registers at addresses *Base* + 8 and *Base* + C, as indicated in Figure 2. These registers are discussed in Section 3.

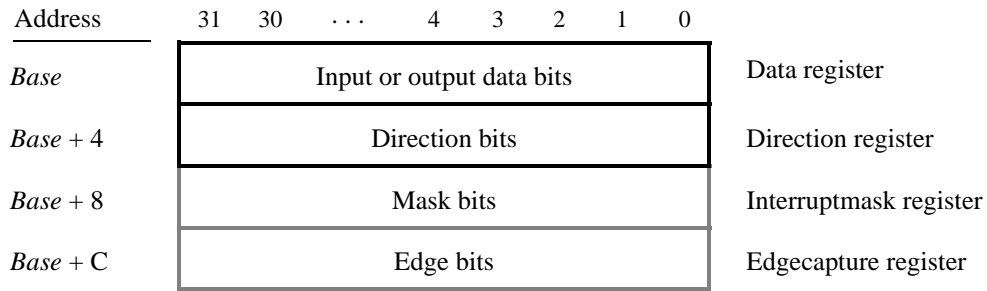


Figure 2. Parallel port registers in the DE2-115 Computer.

2.5.1 Red and Green LED Parallel Ports

The red lights $LEDR_{17-0}$ and green lights $LEDG_{8-0}$ on the DE2-115 board are each driven by an output parallel port, as illustrated in Figure 3. The port connected to $LEDR$ contains an 18-bit write-only *Data* register, which has the address $0xFF200000$. The port for $LEDG$ has a nine-bit *Data* register that is mapped to address $0xFF200010$. These two registers can be written using word accesses, and the upper bits not used in the registers are ignored.

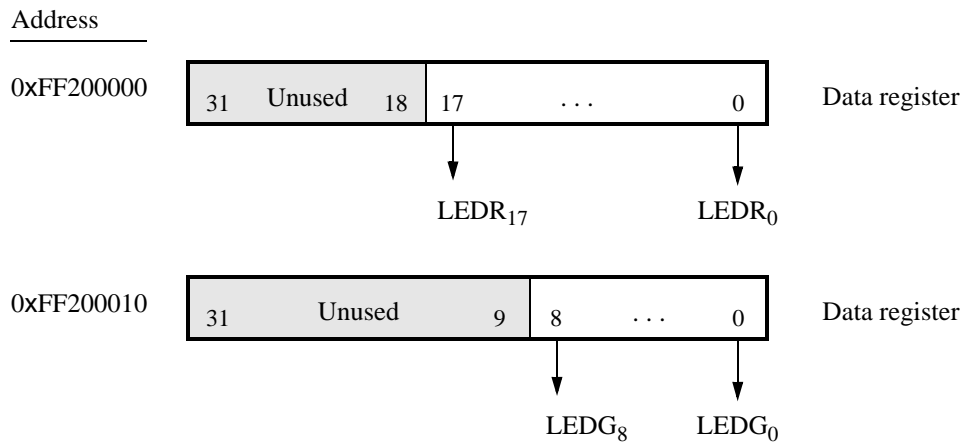


Figure 3. Output parallel ports for $LEDR$ and $LEDG$.

2.5.2 7-Segment Displays Parallel Port

There are two parallel ports connected to the 7-segment displays on the DE2-115 board, each of which comprises a 32-bit write-only *Data* register. As indicated in Figure 4, the register at address $0xFF200020$ drives digits $HEX3$ to $HEX0$, and the register at address $0xFF200030$ drives digits $HEX4$ to $HEX7$. Data can be written into these two registers by using word operations. This data directly controls the segments of each display, according to the bit locations given in Figure 4. The locations of segments 6 to 0 in each seven-segment display on the DE2-115 board is illustrated on the right side of the figure.

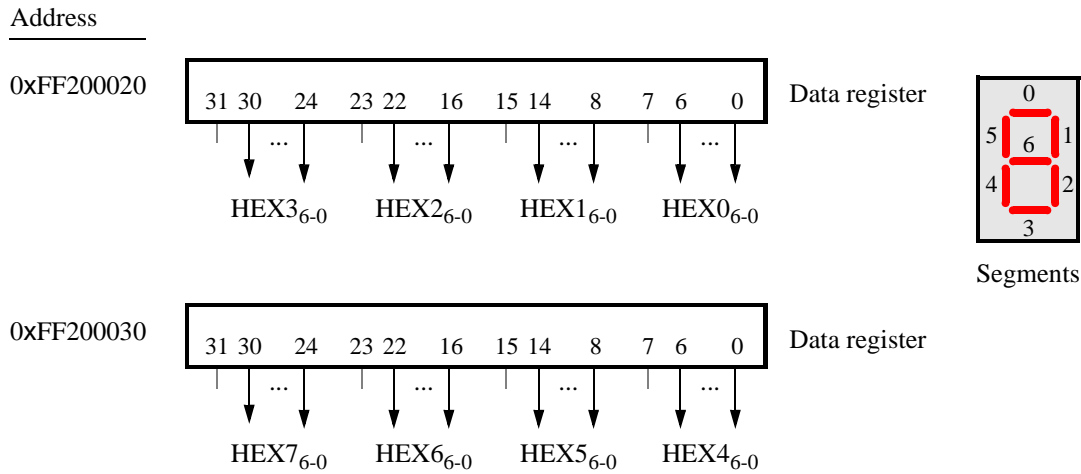


Figure 4. Bit locations for the 7-segment displays parallel ports.

2.5.3 Slider Switch Parallel Port

The SW₁₇₋₀ slider switches on the DE2-115 board are connected to an input parallel port. As illustrated in Figure 5, this port comprises an 18-bit read-only *Data* register, which is mapped to address 0xFF200040.

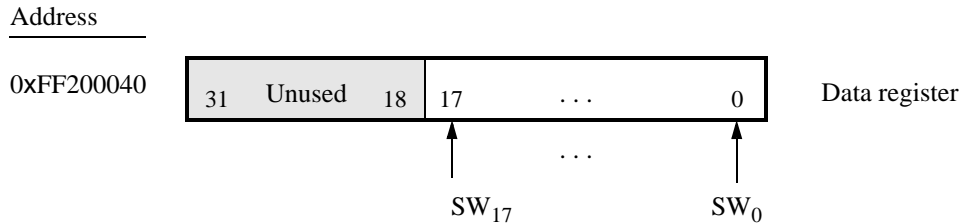


Figure 5. *Data* register in the slider switch parallel port.

2.5.4 Pushbutton Key Parallel Port

The parallel port connected to the KEY₃₋₀ pushbutton switches on the DE2-115 board comprises three 4-bit registers, as shown in Figure 6. These registers have the base address 0xFF200050 and can be accessed using word operations. The read-only *Data* register provides the values of the switches KEY₃₋₀. The other two registers shown in Figure 6, at addresses 0xFF200058 and 0xFF20005C, are discussed in Section 3.

2.5.5 Expansion Parallel Port

The DE2-115 Computer includes one bidirectional parallel port that is connected to the JP5 expansion header on the DE2-115 board. This parallel port includes the four 32-bit registers that were described previously for Figure 2. The base address of this port is 0xFF200060. Figure 7 gives a diagram of the JP1 expansion connector on the

Address	31	30	...	4	3	2	1	0	
0xFF200050	Unused				KEY ₃₋₀				Data register
Unused	Unused								
0xFF200058	Unused				Mask bits				Interruptmask register
0xFF20005C	Unused				Edge bits				Edgecapture register

Figure 6. Registers used in the pushbutton parallel port.

DE2-115 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 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 *JP5* are not usable as input/output 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.

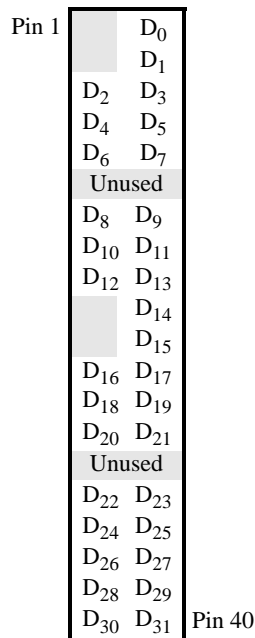


Figure 7. Assignment of parallel port bits to pins.

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

The DE2-115 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 the Appendix in Listings 1 and 2. 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 the values of the SW switches on the LED lights. A rotating pattern is displayed on the LEDs. This pattern is rotated to the left by using a Nios II *rotate* instruction, and a delay loop is used to make the shifting slow enough to observe. The pattern can be changed to the values of the SW switches by pressing a pushbutton KEY. When a pushbutton key is pressed, the program waits in a loop until the key is released.

The source code files shown in Listings 1 and 2 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*.

2.6 JTAG* Port

The JTAG* port implements a communication link between the DE2-115 board and its host computer. This link can be used by the Intel Quartus® Prime software to transfer FPGA programming files into the DE2-115 board, and by the Intel® FPGA Monitor Program, discussed in Section 8. 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 Intel® FPGA Monitor Program is used on the host computer, then this character data is sent and received through its *Terminal Window*. The programming interface of the JTAG UART consists of two 32-bit registers, as shown in Figure 8. The register mapped to address 0xFF201000 is called the *Data* register and the register mapped to address 0xFF201004 is called the *Control* register.

Address	31	...	16	15	14	...	11	10	9	8	7	...	1	0	
0xFF201000	RAVAIL			RVALID	Unused				DATA				Data register		
0xFF201004	WSPACE			Unused				AC	WI	RI		WE	RE	Control register	

Figure 8. 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 8. Note that writing into this register has no effect on received data. The amount of space, *WSPACE*, currently available in the transmit FIFO is provided in bits 31–16 of the *Control* register. If the transmit FIFO is full, then any characters written to the *Data* register will be lost.

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.

2.6.1 Using the JTAG* UART with Assembly Language Code and C Code

Listings 3 and 4 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 Listings 3 and 4 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*.

2.7 Interval Timers

The DE2-115 Computer includes a timer module implemented in the FPGA that can be used by the Nios II processor. This timer can be loaded with a preset value, and then counts down to zero using a 100-MHz clock. The programming interface for the timer includes six 16-bit registers, as illustrated in Figure 9. The 16-bit register at address 0xFF202000 provides status information about the timer, and the register at address 0xFF202004 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 interrupts, which are discussed in section 3.

Address	31	...	17	16	15	...	3	2	1	0	
0xFF202000	Unused								RUN	TO	Status register
0xFF202004	Unused				STOP		START		CONT	ITO	Control register
0xFF202008	Not present (interval timer has 16-bit registers)				Counter start value (low)						
0xFF20200C					Counter start value (high)						
0xFF202010					Counter snapshot (low)						
0xFF202014					Counter snapshot (high)						

Figure 9. 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*) is used to commence/suspend the operation of the timer by writing a 1 into the respective bit.

The two 16-bit registers at addresses 0xFF202008 and 0xFF20200C allow the period of the timer to be changed by setting the starting count value. The default setting provided in the DE2-115 Computer gives a timer period of 125 msec. To achieve this period, the starting value of the count is $100 \text{ MHz} \times 125 \text{ msec} = 12.5 \times 10^6$. It is possible to capture a snapshot of the counter value at any time by performing a write to address 0xFF202010. This write operation causes the current 32-bit counter value to be stored into the two 16-bit timer registers at addresses 0xFF202010 and 0xFF202014. These registers can then be read to obtain the count value.

A second interval timer, which has an identical interface to the one described above, is also available in the FPGA, starting at the base address 0xFF202020.

Each Nios II processor has exclusive access to two interval timers.

2.8 Floating-point Hardware

The Nios II processor in the DE2-115 Computer includes hardware support for floating-point addition, subtraction, multiplication, and division. To use this support in a C program, variables must be declared with the type *float*. A simple example of such code is given in Listing 17. When this code is compiled, it is necessary to pass the special argument `-mcustom-fpu-cfg=60-2` to the C compiler, to instruct it to use the floating-point hardware support.

2.9 System ID

The system ID module provides a unique value that identifies the DE2-115 Computer system. The host computer connected to the DE2-115 board can query the system ID module by performing a read operation through the JTAG port. The host computer can then check the value of the returned identifier to confirm that the DE2-115 Computer has been properly downloaded onto the DE2-115 board. This process allows debugging tools on the host computer, such as the Intel® FPGA Monitor Program, to verify that the DE2-115 board contains the required computer system before attempting to execute code that has been compiled for this system.

3 Exceptions and Interrupts

The reset address of the Nios II processor in the DE2-115 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 DE2-115 Computer. The rest of this section describes the interrupt behavior associated with the interval timer, parallel ports, and serial ports in the DE2-115 Computer.

I/O Peripheral	IRQ #
Interval timer	0
Pushbutton switch parallel port	1
Second Interval timer	2
Audio port	6
PS/2 port	7
JTAG port	8
IrDA port	9
Serial port	10
JP5 Expansion parallel port	11
USB Port	17
PS/2 port dual	23

Table 1. Hardware IRQ interrupt assignment for the DE2-115 Computer.

3.1 Interrupts from Parallel Ports

Parallel ports implemented in the FPGA in the DE2-115 Computer were illustrated in Figure 2, which is reproduced as Figure 10. 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 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 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 10. Registers used for interrupts from the parallel ports.

3.1.1 Interrupts from the Pushbutton Keys

Figure 6, reproduced as Figure 11, shows the registers associated with the pushbutton parallel port. The *Interruptmask* register allows 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. An interrupt service routine can read this register to determine which key has been pressed. Writing any value to the *Edgecapture* register deasserts the interrupt signal being sent to the processor and sets all bits of the *Edgecapture* register to zero.

Address	31	30	...	4	3	2	1	0	
0xFF200050	Unused				KEY ₃₋₀				Data register
Unused	Unused								
0xFF200058	Unused				Mask bits				Interruptmask register
0xFF20005C	Unused				Edge bits				Edgecapture register

Figure 11. Registers used for interrupts from the pushbutton parallel port.

3.2 Interrupts from the JTAG* UART

Figure 8, reproduced as Figure 12, shows the data and *Control* registers of the JTAG UART. As we said in Section 2.6, *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 12 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	
0xFF201000	RAVAIL		RVALID		Unused				DATA						Data register
0xFF201004	WSPACE		Unused				AC		WI	RI			WE	RE	Control register

Figure 12. Interrupt bits in the JTAG UART registers.

3.3 Interrupts from the serial port UART

We introduced the data and *Control* registers associated with the serial port UART in Figure 20, in section 4.7. The *RE* and *WE* bits in the *Control* register in Figure 20 are used to enable processor interrupts associated with the receive and transmit FIFOs. When enabled, interrupts are generated when *RAVAIL* for the receive FIFO, or *WSPACE* for the transmit FIFO, exceeds 31. Pending interrupts are indicated in the *Control* register's *RI* and *WI* bits, and can be cleared by writing or reading data to/from the UART.

3.4 Interrupts from the IrDA* UART

In section 4.8 we discussed the *Data* and *Control* registers associated with the IrDA UART, which are shown in Figure 21. The *RE* and bit in the *Control* register is used to enable or disable processor interrupts associated with the receive FIFO. When enabled, interrupts are generated when the receive FIFO is more than 75% full (*RAVAIL* is greater than 95). Pending interrupts are indicated in the *Control* register's *RI* bit, and can be cleared by reading data

from the UART. As the DE2-115 board does not include an IrDA transmitter, the controls for transmit interrupts (*WE* and *WI*) should be ignored.

3.5 Interrupts from the FPGA Interval Timer

Figure 9, in Section 2.7, shows six registers that are associated with the interval timer. As we said in Section 2.7, the *TO* bit in the *Status* register 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 *ITO* bit in the *Control* register. Setting the *ITO* bit to 1 causes an interrupt request to be sent to the processor whenever *TO* becomes 1. After an interrupt occurs, it can be cleared by writing any value into the *Status* register.

3.6 Using Interrupts with Assembly Language Code

An example of assembly language code for the DE2-115 Computer that uses interrupts is shown in Listing 5. When this code is executed on the DE2-115 board it displays a rotating pattern on the LEDs. The pattern's rotation can be toggled through pressing the pushbutton KEYS. Two types of interrupts are used in the code. The LEDs are controlled by an interrupt service routine for the interval timer, and another interrupt service routine is used to handle the pushbutton keys. The speed of the rotation is set in the main program, by using a counter value in the interval timer that causes an interrupt to occur every 50 msec.

The reset and exception handlers for the main program in Listing 5 are given in Listing 6. 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 Listings 7 and 8, respectively.

3.7 Using Interrupts with C Language Code

An example of C language code for the DE2-115 Computer that uses interrupts is shown in Listing 9. This code performs exactly the same operations as the code described in Listing 5.

To enable interrupts the code in Listing 9 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 Listing 10.

The reset and exception handlers for the main program in Listing 9 are given in Listing 11. 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 Listing 11, the *interrupt_handler* code first determines which exception has occurred, by using a macro from Listing 10 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 12, and the interrupt service routine for the pushbutton switches appears in Listing 13.

The source code files shown in Listing 5 to Listing 13 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*.

4 Media Components

This section describes the audio in/out, video-out, video-in, 16 × 2 character display, PS/2, serial, IrDA, and USB ports.

4.1 Audio In/Out Port

The DE2-115 Computer includes an audio port that is connected to the audio CODEC (COder/DECoder) chip on the DE2-115 board. The default setting for the sample rate provided by the audio CODEC is 8K samples/sec. The audio port provides audio-input capability via the microphone jack on the DE2-115 board, as well as audio output functionality via the line-out jack. The audio port includes four FIFOs that are used to hold incoming and outgoing data. Incoming data is stored in the left- and right-channel *Read* FIFOs, and outgoing data is held in the left- and right-channel *Write* FIFOs. All FIFOs have a maximum depth of 128 32-bit words.

The audio port's programming interface consists of four 32-bit registers, as illustrated in Figure 13. The *Control* register, which has the address 0xFF203040, is readable to provide status information and writable to make control settings. Bit *RE* of this register provides an interrupt enable capability for incoming data. Setting this bit to 1 allows the audio core to generate a Nios II interrupt when either of the *Read* FIFOs are filled 75% or more. The bit *RI* will then be set to 1 to indicate that the interrupt is pending. The interrupt can be cleared by removing data from the *Read* FIFOs until both are less than 75% full. Bit *WE* gives an interrupt enable capability for outgoing data. Setting this bit to 1 allows the audio core to generate an interrupt when either of the *Write* FIFOs are less than 25% full. The bit *WI* will be set to 1 to indicate that the interrupt is pending, and it can be cleared by filling the *Write* FIFOs until both are more than 25% full. The bits *CR* and *CW* in Figure 13 can be set to 1 to clear the *Read* and *Write* FIFOs, respectively. The clear function remains active until the corresponding bit is set back to 0.

Address	31 ... 24	23 ... 16	15 ... 10	9	8	7 ... 4	3	2	1	0	
0xFF203040	Unused			WI	RI		CW	CR	WE	RE	Control
0xFF203044	WSLC	WSRC	RALC		RARC				Fifospace		
0xFF203048	Left data										Leftdata
0xFF20304C	Right data										Rightdata

Figure 13. Audio port registers.

The read-only *Fifospace* register in Figure 13 contains four 8-bit fields. The fields *RARC* and *RALC* give the number of words currently stored in the right and left audio-input FIFOs, respectively. The fields *WSRC* and *WSLC* give the number of words currently available (that is, *unused*) for storing data in the right and left audio-out FIFOs. When all FIFOs in the audio port are cleared, the values provided in the *Fifospace* register are $RARC = RALC = 0$ and $WSRC = WSLC = 128$.

The *Leftdata* and *Rightdata* registers are readable for audio in, and writable for audio out. When data is read from these registers, it is provided from the head of the *Read* FIFOs, and when data is written into these registers it is loaded into the *Write* FIFOs.

A fragment of C code that uses the audio port is shown in Listing 14. The code checks to see when the depth of either the left or right *Read* FIFO has exceeded 75% full, and then moves the data from these FIFOs into a memory buffer. This code is part of a program that is distributed as part of the Intel® FPGA Monitor Program. The source code can be found under the heading *sample programs*, and is identified by the name *Audio*.

4.2 Video-out Port

The DE2-115 Computer includes a video-out port connected to the on-board VGA controller that can be connected to a standard VGA monitor. The video-out port support a screen resolution of 640×480 . The image that is displayed by the video-out port is derived from two sources: a *pixel* buffer, and a *character* buffer.

4.2.1 Pixel Buffer

The pixel buffer for the video-out port holds the data (color) for each pixel that will be displayed. As illustrated in Figure 14, the pixel buffer provides an image resolution of 320×240 pixels, with the coordinate 0,0 being at the top-left corner of the image. Since the video-out port supports the screen resolution of 640×480 , each of the pixel values in the pixel buffer is replicated in both the *x* and *y* dimensions when it is being displayed on the screen.

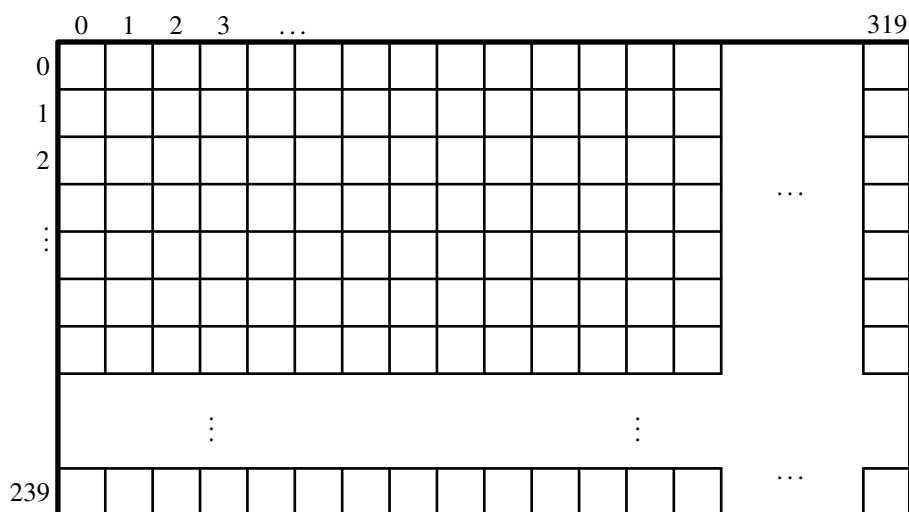
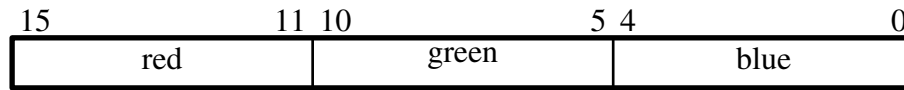


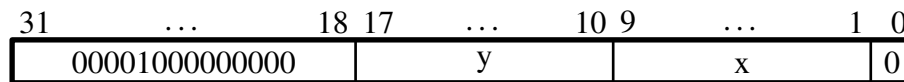
Figure 14. Pixel buffer coordinates.

Figure 15a shows that each pixel color is represented as a 16-bit halfword, with five bits for the blue and red components, and six bits for green. As depicted in part *b* of Figure 15, pixels are addressed in the pixel buffer by using the combination of a *base* address and an *x,y* offset. In the DE2-115 Computer the default address of the pixel buffer is $0x08000000$, which corresponds to the starting address of the FPGA on-chip memory. Using this scheme, the pixel at location 0,0 has the address $0x08000000$, the pixel 1,0 has the address $base + (00000000\ 00000001\ 0)_2$

= $0x08000002$, the pixel 0,1 has the address $base + (00000001\ 00000000\ 0)_2 = 0x08000400$, and the pixel at location 319,239 has the address $base + (11101111\ 100111111\ 0)_2 = 0x0803BE7E$.



(a) Pixel values



(b) Pixel address

Figure 15. Pixel values and addresses.

You can create an image by writing color values into the pixel addresses as described above. A dedicated *pixel buffer controller* continuously reads this pixel data from sequential addresses in the corresponding memory for display on the screen. You can modify the pixel data at any time, simply by writing to the pixel addresses. Thus, an image can be changed even when it is in the process of being displayed. However, it is also possible to avoid making changes to the pixel buffer while it is being displayed, by using the concept of *double-buffering*. In this scheme, two pixel buffers are involved, called the *front* and *back* buffers, described below.

4.2.2 RGB Resampling

The DE2-115 Computer contains an RGB Resampler for converting video streams between RGB color formats. Reading from the 32-bit *Status* register at address $0xFF203010$ provides information about alpha/no alpha, color/grayscale, and mode for the incoming and outgoing formats. The incoming format for the DE2-115 Computer video stream is $0x14$, which corresponds to no alpha, color, 16-bit RGB (5-bit Red, 6-bit Green, 5-bit Blue). For more information, the reader should refer to the video module's online documentation, *Video IP Cores for Intel DE-Series Boards*, which is available from Intel's FPGA University Program web site.

4.2.3 Double Buffering

As mentioned above, a pixel buffer controller reads data out of the pixel buffer so that it can be displayed on the screen. This pixel buffer controller includes a programming interface in the form of a set of registers, as illustrated in Table 2. The register at address $0xFF203020$ is called the *Buffer* register, and the register at address $0xFF203024$ is the *Backbuffer* register. Each of these registers stores the starting address of a pixel buffer. The Buffer register holds the address of the pixel buffer that is displayed on the screen. As mentioned above, in the default configuration of the DE2-115 Computer this Buffer register is set to the address $0x08000000$, which points to the start of the FPGA on-chip memory. The default value of the Backbuffer register is also $0x08000000$, which means that there is only one pixel buffer. But software can modify the address stored in the Backbuffer register, thereby creating a second pixel buffer. The pixel buffer can be located in the SDRAM memory in the DE2-115 Computer, which has the base address $0x00000000$. Note that the pixel buffer cannot be located in the DDR3 memory in the DE2-115 Computer, because the pixel buffer controller is not connected to the DDR3 memory. An image can be drawn into

the second buffer by writing to its pixel addresses. This image is not displayed on the screen until a pixel buffer *swap* is performed, as explained below.

A pixel buffer swap is caused by writing the value 1 to the Buffer register. This write operation does not directly modify the content of the Buffer register, but instead causes the contents of the Buffer and Backbuffer registers to be swapped. The swap operation does not happen right away; it occurs at the end of a screen-drawing cycle, after the last pixel in the bottom-right corner has been displayed. This time instance is referred to as the *vertical synchronization* time, and occurs every 1/60 seconds. Software can poll the value of the *S* bit in the *Status* register, at address 0xFF20302C, to see when the vertical synchronization has happened. Writing the value 1 into the Buffer register causes *S* to be set to 1. Then, when the swap of the Buffer and Backbuffer registers has been completed *S* is reset back to 0.

Address	Register Name	R/W	Bit Description								
			31...24	23...16	15...12	11...8	7...6	5...3	2	1	0
0xFF203020	Buffer	R	Buffer's start address								
0xFF203024	BackBuffer	R/W	Back buffer's start address								
0xFF203028	Resolution	R	Y				X				
0xFF20302C	Status	R	m	n	(1)	BS	SB	(1)	EN	A	S
	Control	W	(1)						EN	(1)	

Notes:

(1) Reserved. Read values are undefined. Write zero.

Table 2. Pixel Buffer Controller

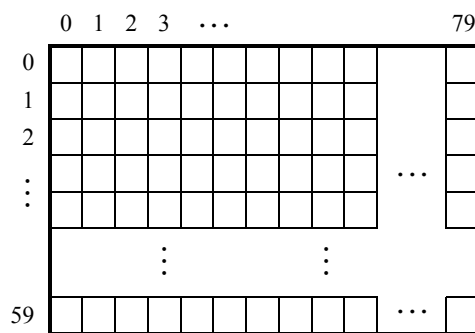
In a typical application the pixel buffer controller is used as follows. While the image contained in the pixel buffer that is pointed to by the Buffer register is being displayed, a new image is drawn into the pixel buffer pointed to by the Backbuffer register. When this new image is ready to be displayed, a pixel buffer swap is performed. Then, the pixel buffer that is now pointed to by the Backbuffer register, which was already displayed, is cleared and the next new image is drawn. In this way, the next image to be displayed is always drawn in the “back” pixel buffer, and the two pixel buffer pointers are swapped when the new image is ready to be displayed. Each time a swap is performed software has to synchronize with the video-out port by waiting until the *S* bit in the Status register becomes 0.

As shown in Table 2 the *Status* register contains additional information other than the *S* bit. The fields *n* and *m* give the number of address bits used for the *X* and *Y* pixel coordinates, respectively. The *BS* field specifies the number of data bits per symbol minus one. The *SB* field specifies the number of symbols per beat minus one. The *A* field allows the selection of two different ways of forming pixel addresses. If configured with *A* = 0, then the pixel controller expects addresses to contain *X* and *Y* fields, as we have used in this section. But if *A* = 1, then the controller expects addresses to be consecutive values starting from 0 and ending at the total number of pixels–1. The *EN* field is used to enable or disable the DMA controller. If this bit is set to 0, the DMA controller will be turned off.

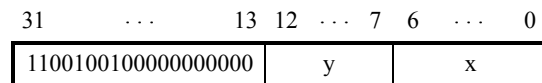
In Table 2 the default values of the status register fields in the DE2-115 Computer are used when forming pixel addresses. The defaults are *n* = 9, *m* = 8, and *A* = 0. If the pixel buffer controller is changed to provide different values of these fields, then the way in which pixel addresses are formed has to be modified accordingly. The programming interface also includes a *Resolution* register, shown in Table 2, that contains the *X* and *Y* resolution of the pixel buffer(s).

4.2.4 Character Buffer

The character buffer for the video-out port is stored in on-chip memory in the FPGA on the DE2-115 board. As illustrated in Figure 16a, the buffer provides a resolution of 80×60 characters, where each character occupies an 8×8 block of pixels on the screen. Characters are stored in each of the locations shown in Figure 16a using their ASCII codes; when these character codes are displayed on the monitor, the character buffer automatically generates the corresponding pattern of pixels for each character using a built-in font. Part b of Figure 16 shows that characters are addressed in the memory by using the combination of a *base* address, which has the value $0x09000000$, and an x, y offset. Using this scheme, the character at location 0,0 has the address $0x09000000$, the character 1,0 has the address $base + (000000\ 0000001)_2 = 0x09000001$, the character 0,1 has the address $base + (000001\ 0000000)_2 = 0x09000080$, and the character at location 79,59 has the address $base + (111011\ 1001111)_2 = 0x09001DCF$.



(a) Character buffer coordinates



(b) Character buffer addresses

Figure 16. Character buffer coordinates and addresses.

4.2.5 Using the Video-out Port with C code

A fragment of C code that uses the pixel and character buffers is shown in Listing 15. The first **for** loop in the figure draws a rectangle in the pixel buffer using the color *pixel_color*. The rectangle is drawn using the coordinates x_1, y_1 and x_2, y_2 . The second **while** loop in the figure writes a null-terminated character string pointed to by the variable *text_ptr* into the character buffer at the coordinates x, y . The code in Listing 15 is included in the sample program called *Video* that is distributed with the Intel® FPGA Monitor Program.

4.3 Video-in Port

The DE2-115 Computer includes a video-in port for use with the composite video-in connector on the DE2-115 board. The video analog-to-digital converter (ADC) connected to this port is configured to support an NTSC

video source. The video-in port provides frames of video at a resolution of 320 x 240 pixels. These video frames can be displayed on a monitor by using the video-out port described in Section 4.2. The video-in port writes each frame of the video-in data into the pixel buffer described in Section 4.2.1. The video-in port can be configured to provide two types of images: either the “raw” image provided by the video ADC, or a version of this image in which only “edges” that are detected in the image are drawn.

The video-in port has a programming interface that consists of two registers, as illustrated in Figure 17. The *Control* register at the address 0xFF20306C is used to enable or disable the video input. If the *EN* bit in this register is set to 0, then the video-in core does not store any data into the pixel buffer. Setting *EN* to 1 and then changing *EN* to 0 can be used to capture a still picture from the video-in port.

The register at address 0xFF203070 is used to enable or disable edge detection. Setting the *E* bit in this register to 1 causes the input video to be passed through hardware circuits that detect edges in the images. The image stored in the pixel buffer will then consist of dark areas that are punctuated by lighter lines along the edges that have been detected. Setting *E* = 0 causes a normal image to be stored into the pixel buffer.

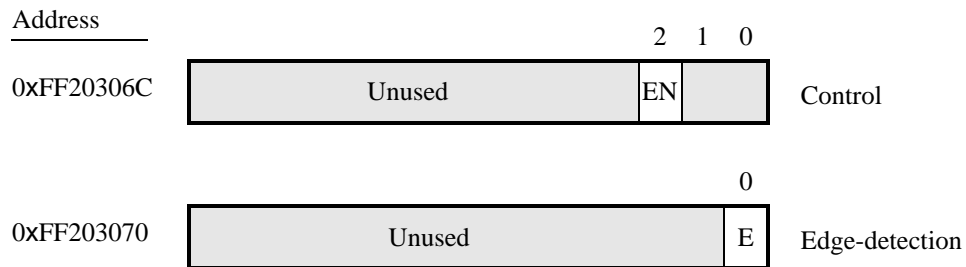


Figure 17. The video-in port programming interface.

4.3.1 DMA Controller for Video

The data provided by the Video-In core is stored into memory using a DMA Controller for Video. When operating in *Stream to Memory* mode, the DMA stores the incoming frames to memory. Table 3 describes the registers used in the DMA Controller.

Address	Register Name	R/W	Bit Description								
			31...24	23...16	15...12	11...8	7...6	5...3	2	1	0
0xFF203060	Buffer	R	Buffer's start address								
0xFF203064	BackBuffer	R/W	Back buffer's start address								
0xFF203068	Resolution	R	Y			X					
0xFF20306C	Status	R	m	n	(1)	BS	SB	(1)	EN	A	S
	Control	W	(1)						EN	(1)	

Notes:

(1) Reserved. Read values are undefined. Write zero.

Table 3. Video DMA Controller

The incoming video is stored to memory, starting at the address specified in the *Buffer* register. The *BackBuffer* register is used to store an alternate memory location. To change where the video is stored, the new location should first be written into the *BackBuffer*. Then the value in the *BackBuffer* and *Buffer* registers can be switched by performing a write to the *Buffer* register.

Bit 2 of the *Status/Control* register, *EN*, is used to enable or disable the Video DMA controller. In the DE2-115 Computer, the DMA controller is disabled by default. To enable the DMA controller, write a 1 into this location. The Video DMA Controller will then begin storing the video into the location specified in the *Buffer* register.

The default value stored in the *Buffer* register is 0x08000000. This address is also used as the source for the Video-Out port, as described in Section 4.2, allowing the Video In stream to be displayed on the VGA. If the Video-Out is intended to display a different signal, than the address stored in the Video DMA Controller's *Buffer* register should be changed.

4.4 Audio/Video Configuration Module

The audio/video configuration module controls settings that affect the operation of both the audio port and the video-out port. The audio/video configuration module automatically configures and initializes both of these ports whenever the DE2-115 Computer is reset. For typical use of the DE2-115 Computer it is not necessary to modify any of these default settings. In the case that changes to these settings are needed, the reader should refer to the audio/video configuration module's online documentation, which is available from Intel's FPGA University Program web site.

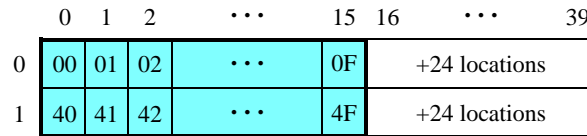
4.5 LCD Display Port

The DE2-115 Computer includes a liquid crystal display (LCD) port that is connected to the 16×2 character display on the DE2-115 board. The display includes a memory for storing character data. As indicated in Figure 18a, the memory has a total capacity of 40×2 characters. The first 16 characters stored in each row are visible on the display, and the remaining 24 characters are not visible at any given time. Each location in the memory can be accessed by combining the x, y coordinates into a 6-bit address as depicted in Figure 18b. Using this scheme, the top and bottom rows of the display start at addresses $(00)_{16}$ and $(40)_{16}$, respectively, as we show in part a of the figure.

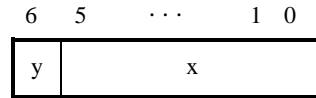
The LCD display port automatically initializes and configures the 16×2 character display when the DE2-115 Computer is reset. The programming interface for the LCD display port is illustrated in part c of Figure 18. It includes an *Instruction* register that is used to control the 16×2 character display, and a *Data* register that is used to send character data to the display. Data can be sent to the display as ASCII character codes, which are automatically converted by the 16×2 character display into bit patterns using a built-in font.

Some of the instructions supported by the 16×2 character display are listed in Table 4. The first instruction, which is identified by the setting $b_7 = 1$, is used to set the location of the cursor. The 6-bit *Address* field should be set using the values shown in Figure 18. After the location of the cursor has been set, a character can be loaded into this location by writing its ASCII value into the *Data* register.

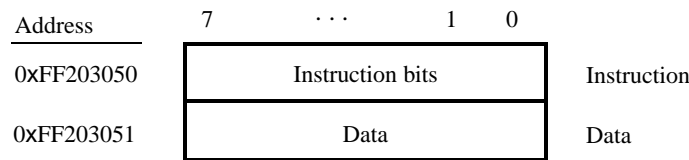
When data is written into the cursor location, the 16×2 character display automatically advances the cursor one position to the right. Multiple characters can be loaded into the display by writing each character in succession into the *Data* register. As we showed in Figure 18, the 16×2 character display includes 40 locations in each row. When



(a) 16 x 2 character display



(b) 16 x 2 character display addresses



(c) LCD display port registers

Figure 18. LCD addresses and registers.

the cursor is advanced past address $(0F)_{16}$ in the top row, the next 24 characters are stored in locations that are not visible on the display. After 40 characters have been written into the top row, the cursor advances to the bottom row at address $(40)_{16}$. At the end of the bottom row, the cursor advances back to address $(00)_{16}$.

The 16×2 character display has the capability to shift its entire contents one position to the left or right. As shown in Table 4, the instruction for shifting left is $(18)_{16}$ and the instruction for shifting right is $(1C)_{16}$. These instructions cause both rows in the display to be shifted in parallel; when a character is shifted out of one end of a row, it is rotated back into the other end of that same row. It is possible to turn off the blinking cursor in the display by using the instruction $(0C)_{16}$, and to turn it back on using $(0F)_{16}$. The display can be erased, and the cursor location set to $(00)_{16}$, by using the instruction $(01)_{16}$.

4.6 PS/2 Port

The DE2-115 Computer includes two PS/2 ports that can be connected to a standard PS/2 keyboard or mouse. The port includes a 256-byte FIFO that stores data received from a PS/2 device. The programming interface for the PS/2 port consists of two registers, as illustrated in Figure 19. The *PS2_Data* register is both readable and writable. When bit 15, *RVALID*, is 1, reading from this register provides the data at the head of the FIFO in the *Data* field, and the number of entries in the FIFO (including this read) in the *RAVAIL* field. When *RVALID* is 1, reading from the *PS2_Data* register decrements this field by 1. Writing to the *PS2_Data* register can be used to send a command in the *Data* field to the PS/2 device.

Instruction	b_7	b_6-0
Set cursor location	1	Address
Shift display left	0	0011000
Shift display right	0	0011100
Cursor off	0	0001100
Cursor blink on	0	0001111
Clear display	0	0000001

Table 4. LCD display instructions.

The *PS2_Control* register can be used to enable interrupts from the PS/2 port by setting the *RE* field to the value 1. When this field is set, then the PS/2 port generates an interrupt when *RAVAIL* > 0. While the interrupt is pending the field *RI* will be set to 1, and it can be cleared by emptying the PS/2 port FIFO. The *CE* field in the *PS2_Control* register is used to indicate that an error occurred when sending a command to a PS/2 device.

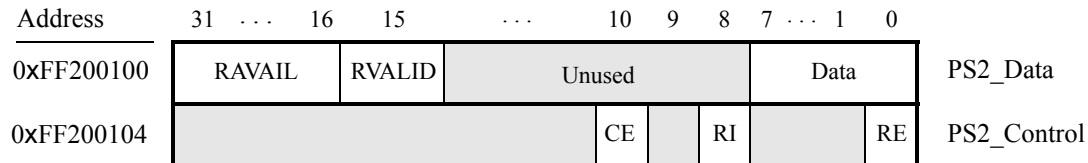


Figure 19. PS/2 port registers.

A fragment of C code that uses the PS/2 port is given in Listing 16. This code reads the content of the *Data* register, and saves data when it is available. If the code is used continually in a loop, then it stores the last three bytes of data received from the PS/2 port in the variables *byte1*, *byte2*, and *byte3*. This code is included as part of a sample program called *PS2* that is distributed with the Intel® FPGA Monitor Program.

4.6.1 PS/2 Port Dual

The DE2-115 Computer includes a second PS/2 port that allows both a keyboard and mouse to be used at the same time. To use the dual port a Y-splitter cable must be used and the keyboard and mouse must be connected to the PS/2 connector on the DE2-115 board through this cable. The PS/2 port dual has the same registers as the PS/2 port shown in Listing 16, except that the base address of its *PS2_Data* register is 0xFF200108 and the base address of its *PS2_Control* register is 0xFF20010C.

4.7 Serial Port

The serial port in the DE2-115 Computer implements a UART that is connected to an RS232 chip on the DE2-115 board. This UART is configured for 8-bit data, one stop bit, odd parity, and operates at a baud rate of 115,200. The serial port's programming interface consists of two 32-bit registers, as illustrated in Figure 20. The register at address 0xFF201010 is referred to as the *Data* register, and the register at address 0xFF201014 is called the *Control* register.

Address	31	...	16	15	14	...	10	9	8	7	...	1	0	
0xFF201010	RAVAIL			RVALID	Unused			PE		DATA				Data register
0xFF201014	WSPACE			Unused			WI	RI			WE	RE		Control register

Figure 20. Serial port UART registers.

When character data is received from the RS 232 chip it is stored in a 256-character FIFO in the UART. As illustrated in Figure 20, the number of characters, *RAVAIL*, currently stored in this FIFO is provided in bits 31–16 of the *Data* register. If the receive FIFO overflows, then additional data is lost. When the data that is present in the receive FIFO is available for reading, then 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 available to be read from the receive FIFO, then *RVALID* will be set to 0 and the data in bits 7–0 is undefined.

The UART also includes a 256-character FIFO that stores data waiting to be sent to the RS 232 chip. Character data is loaded into this FIFO by performing a write to bits 7–0 of the *Data* register. Writing into this register has no effect on received data. The amount of space, *WSPACE*, currently available in the transmit FIFO is provided in bits 31–16 of the *Control* register, as indicated in Figure 20. If the transmit FIFO is full, then any additional characters written to the *Data* register will be lost.

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

4.8 IrDA* Port

The DE2-115 Computer includes an IrDA UART for communicating wirelessly with peripherals over the infrared spectrum. It is configured for 8-bit data and one stop bit, and operates at a baud rate of 155,200. The default configuration does not use a parity bit. The programming interface consists of two 32-bit registers, as shown in Figure 21. The register at address 0xFF201020 is referred to as the *Data* register, and the register at address 0xFF201024 is the *Control* register.

The operation of the IrDA UART is similar to the Serial Port UART described above. Data received through the IrDA is stored in a 128-character FIFO in the UART. As shown in Figure 21, the number of characters, *RAVAIL*, currently stored in this FIFO is provided in bits 23–16 of the *Data* register. If the FIFO overflows, then any additional data is lost. When a read of the *Data* register is performed, the character at the head of the FIFO is provided in bits 7–0. If the character read is valid, the value of bit 15, *RVALID* will be one. If no data is available to be read from the receive FIFO, then *RVALID* will be set to 0 and the data in bits 7–0 is undefined.

The *Control* register bits *RE* and *RI* are described in section 3. The DE2-115 does not feature an IrDA transmitter. Writing to bits 7–0 of the *Data* register will have no effect, and the values of *WSPACE*, *WE*, and *WI* should be ignored.

Address	31	...	24	23	...	16	15	14	...	10	9	8	7	...	1	0			
0xFF201020	Unused			RAVAIL			RVALID			Unused			PE				DATA		Data register
0xFF201024	Unused			WSPACE			Unused			WI		RI				WE RE		Control register	

Figure 21. IrDA UART registers.

4.9 USB Port

The DE2-115 Computer includes a USB port, which can be used either as a host for a USB peripheral device (such as a mouse or keyboard) or as a device (when connected to host, such as a PC). The USB port is controlled by the USB Controller IP Core, which provides a register-mapped interface, as well as high-level C functions through the *Hardware Abstraction Layer (HAL)*. Using the HAL is the recommended method to send and receive data through the USB port. To use the HAL, the directive `#include "altera_up_avalon_usb_high_level_driver.h"` is needed.

To use the USB port, refer to the code example in Figure 22, which can be used to read input data from a USB mouse. Included with the HAL is a basic USB mouse driver, which can be accessed using the directive `#include "altera_up_avalon_usb_mouse_driver.h"`. In Figure 22, the variable `alt_up_usb_dev *usb_dev` points to the USB device, and is initialized by the function `alt_up_usb_open_dev`. The then `alt_up_usb_setup` and `alt_up_usb_set_config` functions are called to initialize the USB chip and device. To setup the USB device as a mouse, the function `alt_up_usb_mouse_setup` is then called.

Once the USB mouse has been setup, data can be acquired using the `alt_up_usb_retrieve_mouse_packet` function. It stores the change in x-coordinate, change in y-coordinate, and button status into an `alt_up_usb_mouse_packet` data structure. The packet structure is composed of three byte-sized variables: `x_movement`, `y_movement` and `buttons`. The `x_movement` and `y_movement` variables store the change in mouse position in each axis. The left, right, and center mouse buttons are mapped to bits 0, 1 and 2 respectively of the packet's `buttons` variable, which holds the state of the three buttons.

```
#include <stdio.h>
#include "altera_up_avalon_usb_high_level_driver.h"
#include "altera_up_avalon_usb_mouse_driver.h"
int main() {
    alt_up_usb_dev * usb_dev = alt_up_usb_open_dev
        ("/dev/USB/");
    int port = -1;
    int addr = -1;
    int hid = -1;
    int x = 0;
    int y = 0;
    int l_button = 0;
    int m_button = 0;
    int r_button = 0;
    alt_up_usb_mouse_packet packet;

    hid = alt_up_usb_setup (usb_dev, &addr, &port);
    if (port != -1 && hid == 0x0209) {
        alt_up_usb_set_config (usb_dev, addr, port, 1);
        alt_up_usb_mouse_setup (usb_dev, addr, port);
        while (1) {
            alt_up_usb_retrieve_mouse_packet (usb_dev,
                &packet);
            x += packet.x_movement;
            y += packet.y_movement;
            l_button = packet.buttons && 0x01;
            r_button = packet.buttons && 0x02;
            m_button = packet.buttons && 0x04;
            // Process the data
            . . .
        }
    }
}
```

Figure 22. An example of code for a using a USB mouse.

5 Modifying the DE2-115 Computer

It is possible to modify the DE2-115 Computer by using Intel's Quartus® Prime software and Qsys 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 make an editable copy of the DE2-115 Computer. The files for this system are installed as part of the Monitor Program installation. Locate these files, copy them to a working directory, and then use the Quartus Prime and Qsys software to make any desired changes.

Table 5 lists the names of the Qsys IP cores that are used in this system. When the DE2-115 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.

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 DE2-115 Computer from the University Program web site. These files can be found in the *Design Examples* section of the web site
3. Open the *DE2-115_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, *DE2-115_Computer.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 DE2-115 board

The DE2-115 Computer includes a Nios II/f processor. When using the Quartus Prime Web Edition, compiling a design with a Nios II/s or Nios II/f processor will produce a time-limited SOF file. As a result, the board must remain connected to the host computer, and the design cannot be set as the default configuration, as discussed in Section 6. Designs using only Nios II/e processors and designs compiled using the Quartus Prime Subscription Edition do not have this restriction.

I/O Peripheral	Qsys Core
SDRAM	SDRAM Controller
SRAM	SRAM Controller
On-chip memory character buffer	Character Buffer for VGA Display
SD Card	SD Card Interface
Flash	Intel UP Flash Memory IP Core
Red LED parallel port	Parallel Port
Green LED parallel port	Parallel Port
7-segment displays parallel port	Parallel Port
Expansion parallel ports	Parallel Port
Slider switch parallel port	Parallel Port
Pushbutton parallel port	Parallel Port
PS/2 port	PS2 Controller
JTAG port	JTAG UART
Serial port	RS232 UART
IrDA port	IrDA UART
Interval timer	Interval timer
System ID	System ID Peripheral
Audio/video configuration port	Audio and Video Config
Audio port	Audio
Video port	Pixel Buffer DMA Controller
LCD display port	Character LCD 16x2
Video In port	DMA Controller

Table 5. Qsys cores used in the DE2-115 Computer.

6 Making the System the Default Configuration

The DE2-115 Computer can be loaded into the nonvolatile FPGA configuration memory on the DE2-115 board, so that it becomes the default system whenever the board is powered on. Instructions for configuring the DE2-115 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.

7 Memory Layout

Table 6 summarizes the memory map used in the DE2-115 Computer.

Base Address	End Address	I/O Peripheral
0x00000000	0x07FFFFFF	SDRAM
0x08000000	0x081FFFFFF	SRAM
0xFF203020	0xFF20302F	Pixel buffer control
0x09000000	0x09001FFF	On-chip memory character buffer
0xFF203030	0xFF203037	Character buffer control
0x0B000000	0x0B0003FF	SD Card
0x0C000000	0x0C7FFFFFF	Flash
0x0BFF0000	0x0BFF0003	Flash Erase control
0xFF200000	0xFF20000F	Red LED parallel port
0xFF200010	0xFF20001F	Green LED parallel port
0xFF200020	0xFF20002F	7-segment HEX3–HEX0 displays parallel port
0xFF200030	0xFF20003F	7-segment HEX7–HEX4 displays parallel port
0xFF200040	0xFF20004F	Slider switch parallel port
0xFF200050	0xFF20005F	Pushbutton parallel port
0xFF200060	0xFF20006F	JP5 Expansion parallel port
0xFF200100	0xFF200107	PS/2 port
0xFF200108	0xFF20010F	PS/2 port dual
0xFF201000	0xFF201007	JTAG UART port
0xFF201010	0xFF201017	Serial port
0xFF201020	0xFF201027	IrDA port
0xFF202000	0xFF20201F	Interval timer
0xFF202020	0xFF202027	System ID
0xFF203000	0xFF20301F	Audio/video configuration
0xFF203040	0xFF20304F	Audio port
0xFF203050	0xFF203051	LCD display port
0xFF203060	0xFF20306F	Video-In port

Table 6. Memory layout used in the DE2-115 Computer.

8 Intel® FPGA Monitor Program Integration

As we mentioned earlier, the DE2-115 Computer system, and the sample programs described in this document, are made available as part of the Intel® FPGA Monitor Program. Figures 23 to 26 show a series of windows that are used in the Monitor Program to create a new project. In the first screen, shown in Figure 23, the user specifies a file system folder where the project will be stored, gives the project a name, and specifies the type of processor that is being used. Pressing Next opens the window in Figure 24. Here, the user can select the DE2-115 Computer as a pre-designed system. The Monitor Program then fills in the relevant information in the *System details* box, which includes the appropriate system info and fpga configuration files, and preloader. The first of these files specifies to the Monitor Program information about the components that are available in the DE2-115 Computer, such as the type of processor and memory components, and the address map. The second file is an FPGA programming bitstream for the DE2-115 Computer, which can be downloaded by the Monitor Program into the DE2-115 board. Any system which contains a Hard Processor System (HPS) component must also specify the preloader to be run immediately following the circuit being downloaded. This preloader is used to configure the components within the HPS with the settings required for the specific board.

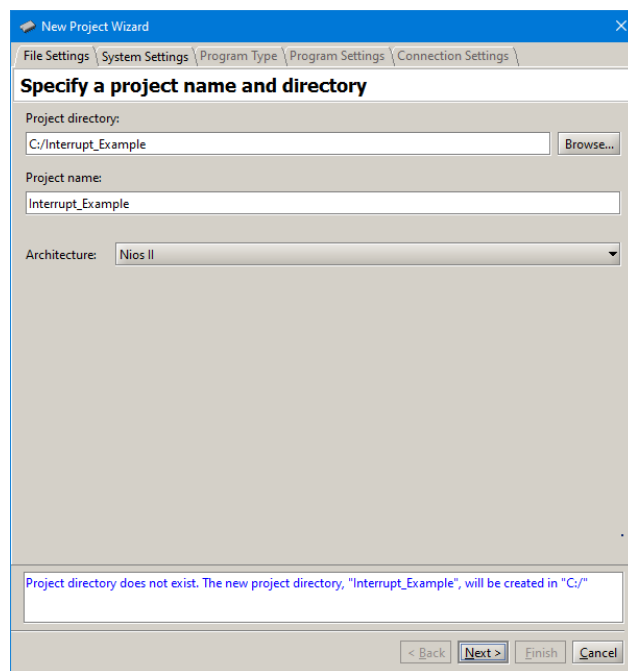


Figure 23. Specifying the project folder and project name.

Pressing **Next** again opens the window in Figure 25. 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 programs for the DE2-115 Computer that are described in this document. 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 26 gives the final screen that is used to create a new project in the Monitor Program. This screen shows the default addresses of compiler and linker sections that will be used for the assembly language or C program associated with the Monitor Program project. In the figure, the drop-down menu called *Linker Section Presets* has been set to **Exceptions**. With this setting the Monitor Program uses specific compiler and linker sections for the selected processor. For the Nios II processor, these sections are for reset and exceptions code, and another section for the main program, called *.text*. For the A9 processor, it has a section for the exception table, called *.vectors*, and another section for the main program, called *.text*. It also shows the initial value used to set the main stack pointer for C programs, which is the starting address of the *.stack* section.

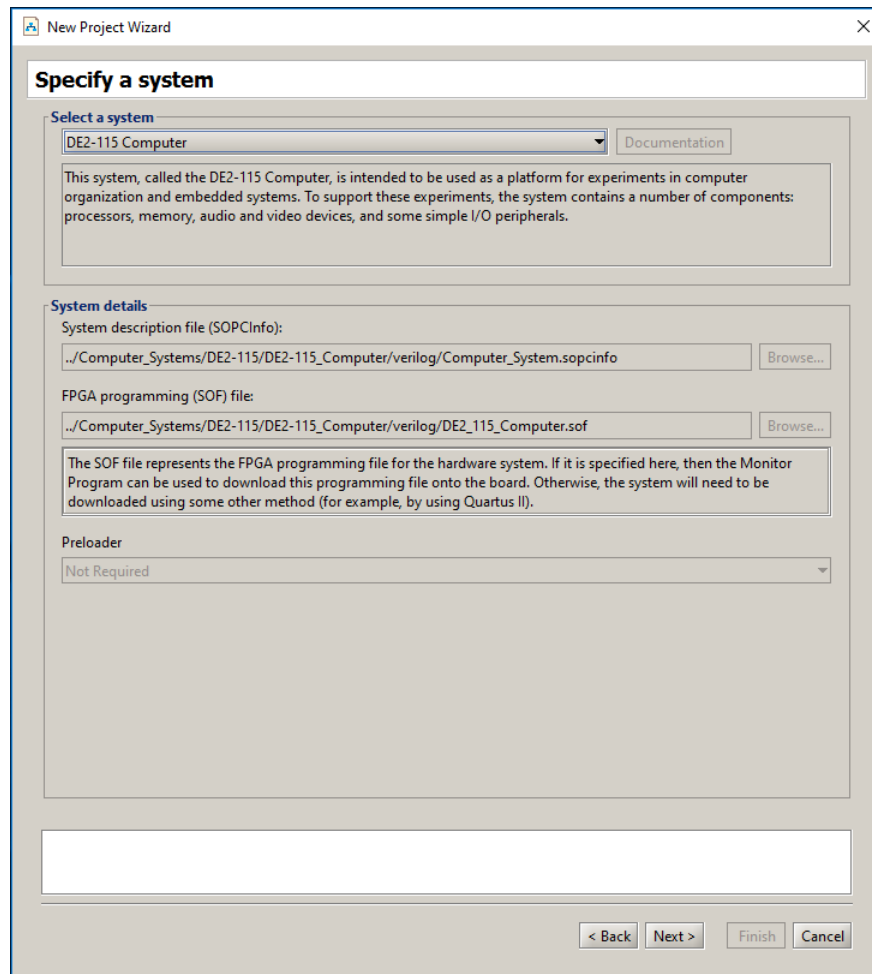


Figure 24. Specifying the DE2-115 Computer system.

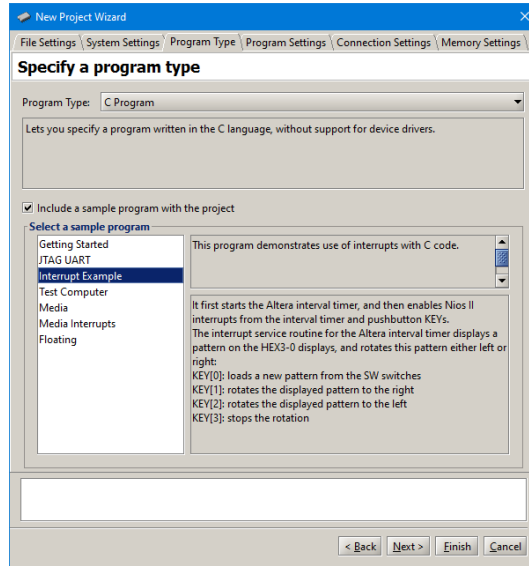


Figure 25. Selecting sample programs.

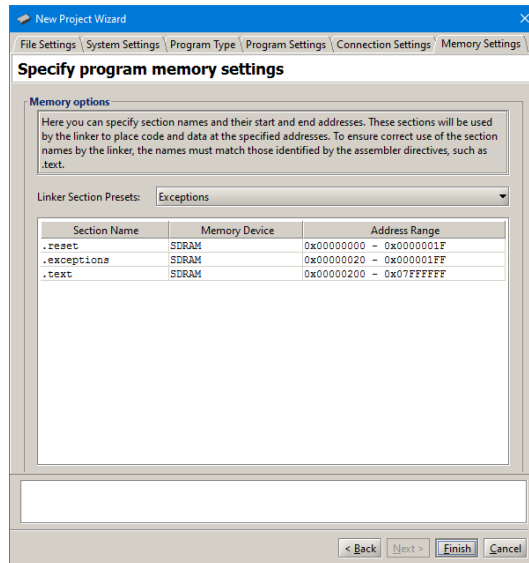


Figure 26. Setting offsets for .text and .data.

9 Appendix

This section contains all of the source code files mentioned in the document.

9.1 Parallel Ports

```
.include      "address_map_nios2.s"

/*****
 * This program demonstrates use of parallel ports
 *
 * It performs the following:
 * 1. displays a rotating pattern on the LEDs
 * 2. if any KEY is pressed, the SW switches are used as the rotating pattern
 *****/

.text                # executable code follows
.global      _start
_start:

/* initialize base addresses of parallel ports */
    movia    r15, SW_BASE      # SW slider switch base address
    movia    r16, LED_BASE     # LED base address
    movia    r17, KEY_BASE     # pushbutton KEY base address
    movia    r18, LED_bits
    ldwio    r6, 0(r18)        # load pattern for LED lights

DO_DISPLAY:
    ldwio    r4, 0(r15)        # load slider switches

    ldwio    r5, 0(r17)        # load pushbuttons
    beq      r5, r0, NO_BUTTON
    mov      r6, r4            # copy SW switch values onto LEDs
    roli     r4, r4, 8         # the SW values are copied into the upper three
                                # bytes of the pattern register
    or       r6, r6, r4        # needed to make pattern consistent as all
                                # 32-bits of a register are rotated
    roli     r4, r4, 8         # but only the lowest 8-bits are displayed on
                                # LEDs

    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 LED
    roli     r6, r6, 1         # rotate the displayed pattern
```



```
        movia    r7, 1500000          # delay counter
DELAY:
        subi    r7, r7, 1
        bne     r7, r0, DELAY

        br     DO_DISPLAY

/*****/
.data          # data follows

LED_bits:
.word         0x0F0F0F0F

.end
```

Listing 1. An example of Nios II assembly language code that uses parallel ports.

```

#include "address_map_nios2.h"
/* This program demonstrates use of parallel ports in the Computer System
 *
 * It performs the following:
 * 1. displays a rotating pattern on the LEDs
 * 2. if a KEY 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 will be used to access these pointer locations,
     * instead of regular memory loads and stores)
     */
    volatile int * LED_ptr      = (int *)LED_BASE; // LED address
    volatile int * SW_switch_ptr = (int *)SW_BASE; // SW slider switch address
    volatile int * KEY_ptr      = (int *)KEY_BASE; // pushbutton KEY address

    int LED_bits = 0x0F0F0F0F; // pattern for LED lights
    int SW_value, KEY_value;
    volatile int
        delay_count; // volatile so the C compiler doesn't remove the loop

    while (1) {
        SW_value = *(SW_switch_ptr); // read the SW slider (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 */
            LED_bits = SW_value | (SW_value << 8) | (SW_value << 16) |
                (SW_value << 24);
            while (*KEY_ptr)
                ; // wait for pushbutton KEY release
        }
        *(LED_ptr) = LED_bits; // light up the LEDs

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

        for (delay_count = 350000; delay_count != 0; --delay_count)
            ; // delay loop
    }
}

```

Listing 2. An example of C code that uses parallel ports.

9.2 JTAG* UART

```
.include      "address_map_nios2.s"

/*****
 * This program demonstrates use of the JTAG UART port
 *
 * 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, SDRAM_END - 3    # starts from largest memory address

    movia    r6, JTAG_UART_BASE   # 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

/*****
 * 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 */
```

Listing 3. An example of assembly language code that uses the JTAG UART (Part a).

```
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
```

Listing 3. An example of assembly language code that uses the JTAG UART (Part b).

```

#include "JTAG_UART.h"
#include "address_map_nios2.h"

/*****
 * 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, echo character, else ignore
        *(JTAG_UART_ptr) = c;
}

/*****
 * Subroutine to read a character from the JTAG UART
 * Returns \0 if no character, otherwise returns the character
 *****/
char get_jtag(volatile int * JTAG_UART_ptr)
{
    int data;
    data = *(JTAG_UART_ptr); // read the JTAG_UART data register
    if (data & 0x00008000) // check RVALID to see if there is new data
        return ((char)data & 0xFF);
    else
        return ('\0');
}

```

Listing 4. An example of C code that uses the JTAG UART (Part a).

```

#include "JTAG_UART.h"
#include "address_map_nios2.h"

/*****
 * This program demonstrates use of the JTAG UART port
 *
 * 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 will be used to access these pointer locations,
       instead of regular memory loads and stores) */
    volatile int * JTAG_UART_ptr = (int *)JTAG_UART_BASE; // JTAG UART address

    char text_string[] = "\nJTAG UART example code\n> \0";
    char *str, c;

    /* print a text string */
    for (str = text_string; *str != 0; ++str)
        put_jtag(JTAG_UART_ptr, *str);

    /* read and echo characters */
    while (1)
    {
        c = get_jtag(JTAG_UART_ptr);
        if (c != '\0')
            put_jtag(JTAG_UART_ptr, c);
    }
}

```

Listing 4. An example of C code that uses the JTAG UART (Part b).

9.3 Interrupts

```
.include      "address_map_nios2.s"
.include      "globals.s"

/*****
 * This program demonstrates use of interrupts. It
 * first starts an interval timer with 50 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 LEDs, and shifts this pattern either left or right:
 *   KEY[0]: loads a new pattern from the SW switches
 *   KEY[1]: toggles the shift direction the displayed pattern
 *****/

.text        # executable code follows
.global      _start
_start:
/* set up the stack */
    movia    sp, SDRAM_END - 3    # stack starts from largest memory
                                   # address

    movia    r16, TIMER_BASE      # interval timer base address
/* set the interval timer period for scrolling the LED lights */
    movia    r12, 5000000         # 1/(100 MHz) x (5 x 10^6) = 50 msec
    sthio    r12, 8(r16)          # store the low half word of counter
                                   # start value

    srli     r12, r12, 16
    sthio    r12, 0xC(r16)        # high half word of counter start value

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

/* write to the pushbutton port interrupt mask register */
    movia    r15, KEY_BASE       # pushbutton key base address
    movi     r7, 0b11            # set interrupt mask bits
    stwio    r7, 8(r15)          # interrupt mask register is (base + 8)

/* enable Nios II processor interrupts */
    movia    r7, 0x00000001       # get interrupt mask bit for interval
                                   # timer
    movia    r8, 0x00000002       # get interrupt mask bit for pushbuttons
    or       r7, r7, r8
    wrctl    ienable, r7         # enable interrupts for the given mask
                                   # bits

    movi     r7, 1
    wrctl    status, r7          # turn on Nios II interrupt processing
```

IDLE:

```
br      IDLE      # main program simply idles

.data
/*****
 * The 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 LED lights
.global  SHIFT_DIR
SHIFT_DIR:
.word   RIGHT # pattern shifting direction

.end
```

Listing 5. An example of assembly language code that uses interrupts.


```

/*****
 * RESET SECTION
 * Note: "ax" is REQUIRED to designate the section as allocatable and executable.
 * Also, the Debug Client automatically places the ".reset" section at the reset
 * location specified in the CPU settings in SOPC Builder.
 *****/
.section    .reset, "ax"

        movia   r2, _start
        jmp     r2                # branch to main program

/*****
 * EXCEPTIONS SECTION
 * Note: "ax" is REQUIRED to designate the section as allocatable and executable.
 * Also, the Monitor Program automatically places the ".exceptions" section at
 * the exception location specified in the CPU settings in SOPC Builder.
 *****/
.section    .exceptions, "ax"
.global    EXCEPTION_HANDLER

EXCEPTION_HANDLER:
        subi    sp, sp, 16        # make room on the stack
        stw    et, 0(sp)

        rdctl  et, ct14
        beq    et, r0, SKIP_EA_DEC # interrupt is not external

        subi    ea, ea, 4         # must decrement ea by one instruction
                                     # for external interrupts, so that the
                                     # interrupted instruction will be run

SKIP_EA_DEC:
        stw    ea, 4(sp)         # save all used registers on the Stack
        stw    ra, 8(sp)         # needed if call inst is used
        stw    r22, 12(sp)

        rdctl  et, ct14
        bne    et, r0, CHECK_LEVEL_0 # interrupt is an external interrupt

NOT_EI:
                                     # exception must be unimplemented
                                     # instruction or TRAP instruction. This
                                     # code does not handle those cases

        br     END_ISR

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

.end
```

Listing 6. Reset and exception handler assembly language code.

```

.include      "address_map_nios2.s"
.include      "globals.s"
.extern      PATTERN          # externally defined variables
.extern      SHIFT_DIR

/*****
 * Interval timer - Interrupt Service Routine
 *
 * Shifts a PATTERN being displayed. The shift direction is determined by the
 * external variable SHIFT_DIR.
 *****/
.global      INTERVAL_TIMER_ISR
INTERVAL_TIMER_ISR:
    subi     sp, sp, 40        # reserve space on the stack
    stw     ra, 0(sp)
    stw     r4, 4(sp)
    stw     r5, 8(sp)
    stw     r6, 12(sp)
    stw     r8, 16(sp)
    stw     r10, 20(sp)
    stw     r20, 24(sp)
    stw     r21, 28(sp)
    stw     r22, 32(sp)
    stw     r23, 36(sp)

    movia   r10, TIMER_BASE   # interval timer base address
    sthio   r0, 0(r10)        # clear the interrupt

    movia   r20, LED_BASE     # LED base address
    movia   r21, PATTERN      # set up a pointer to the display pattern
    movia   r22, SHIFT_DIR    # set up a pointer to the shift direction variable

    ldw     r6, 0(r21)        # load the pattern
    stwio   r6, 0(r20)        # store to LEDs

CHECK_SHIFT:
    ldw     r5, 0(r22)        # get shift direction
    movi    r8, RIGHT
    bne     r5, r8, SHIFT_L

SHIFT_R:
    movi    r5, 1              # set r5 to the constant value 1
    ror     r6, r6, r5        # rotate the displayed pattern right
    br      STORE_PATTERN

SHIFT_L:
    movi    r5, 1              # set r5 to the constant value 1
    rol     r6, r6, r5        # shift left

STORE_PATTERN:
    stw     r6, 0(r21)        # store display pattern

```

```
END_INTERVAL_TIMER_ISR:
    ldw    ra, 0(sp)      # restore registers
    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)
    ldw    r23, 36(sp)
    addi   sp, sp, 40    # release the reserved space on the stack
```

```
.end
```

Listing 7. Interrupt service routine for the interval timer.

```

.include      "address_map_nios2.s"
.include      "globals.s"
.extern      PATTERN                               # externally defined variables
.extern      SHIFT_DIR

/*****
 * Pushbutton - Interrupt Service Routine
 *
 * This routine checks which KEY has been pressed and updates the global
 * variables as required.
 *****/
.global      PUSHBUTTON_ISR
PUSHBUTTON_ISR:
    subi     sp, sp, 20                          # reserve space on the stack
    stw     ra, 0(sp)
    stw     r10, 4(sp)
    stw     r11, 8(sp)
    stw     r12, 12(sp)
    stw     r13, 16(sp)

    movia   r10, KEY_BASE                        # base address of pushbutton KEY
                                                # parallel port
    ldwio   r11, 0xC(r10)                        # read edge capture register
    stwio   r11, 0xC(r10)                        # clear the interrupt

CHECK_KEY0:
    andi    r13, r11, 0b0001                    # check KEY0
    beq     r13, zero, CHECK_KEY1

    movia   r10, SW_BASE                          # base address of SW slider
                                                # switches parallel port
    ldwio   r12, 0(r10)                          # load a new pattern from the SW
                                                # switches
    movia   r10, PATTERN                          # set up a pointer to the pattern
                                                # variable
    stw     r12, 0(r10)                          # store the new pattern to the
                                                # global variable

CHECK_KEY1:
    andi    r13, r11, 0b0010                    # check KEY1
    beq     r13, zero, END_PUSHBUTTON_ISR

    movia   r10, SHIFT_DIR                        # set up a pointer to the shift
                                                # direction variable
    ldw     r12, 0(r10)                          # load the current shift direction
    xori    r12, r12, 1                          # toggle the direction
    stw     r12, 0(r10)                          # store the new shift direction

END_PUSHBUTTON_ISR:
    ldw     ra, 0(sp)                            # Restore all used register to
                                                # previous
    ldw     r10, 4(sp)

```

```
ldw    r11, 8(sp)
ldw    r12, 12(sp)
ldw    r13, 16(sp)
addi   sp, sp, 20
```

```
.end
```

Listing 8. Interrupt service routine for the pushbutton KEYS.

```

#include "address_map_nios2.h"
#include "globals.h" // defines global values
#include "nios2_ctrl_reg_macros.h"

/* the global variables are written by interrupt service routines; we have to
 * declare
 * these as volatile to avoid the compiler caching their values in registers */
volatile int pattern      = 0x0000000F; // pattern for shifting
volatile int shift_dir    = LEFT;      // direction to shift the pattern
volatile int shift_enable = ENABLE;    // enable/disable shifting of the pattern

/*****
 * This program demonstrates use of interrupts. It
 * first starts the interval timer with 50 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 LED 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 *)TIMER_BASE;           // interval timer base address
    volatile int * KEY_ptr = (int *)KEY_BASE; // pushbutton KEY address

    /* set the interval timer period for scrolling the LED lights */
    int counter = 2500000; // 1/(50 MHz) x (2500000) = 50 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) = 0x3; // enable interrupts for all pushbuttons

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

    NIOS2_WRITE_STATUS(1); // enable Nios II interrupts

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

```

Listing 9. 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

```

Listing 10. 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 CPU reset processing */
void the_reset(void) __attribute__((section(".reset")));
void the_reset(void)
/*****
 * Reset code. By giving the code a section attribute with the name ".reset" we
 * allow the linker program to locate this code at the proper reset vector
 * address. This code just calls the main program.
 *****/
{
    asm(".set      noat"); /* 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, main"); // Call the C language main program
    asm("jmp      r2");
}

/* The assembly language code below handles CPU exception processing. This
 * code should not be modified; instead, the C language code in the function
 * interrupt_handler() can be modified as needed for a given application.
 */
void the_exception(void) __attribute__((section(".exceptions")));
void the_exception(void)
/*****
 * Exceptions code. By giving the code a section attribute with the name
 * ".exceptions" we allow the linker program to locate this code at the proper
 * exceptions vector address.
 * This code calls the interrupt handler and later returns from the exception.
 *****/
{
    asm("subi    sp, sp, 128");
    asm("stw     et, 96(sp)");
    asm("rdctl   et, ct14");
    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
                                       * interrupted instruction will be run */

    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 is 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)");
asm("ldw    r8, 32(sp)");
asm("ldw    r9, 36(sp)");
asm("ldw    r10, 40(sp)");
asm("ldw    r11, 44(sp)");
asm("ldw    r12, 48(sp)");
asm("ldw    r13, 52(sp)");
asm("ldw    r14, 56(sp)");
asm("ldw    r15, 60(sp)");
asm("ldw    r16, 64(sp)");
asm("ldw    r17, 68(sp)");
asm("ldw    r18, 72(sp)");
```

```

asm("ldw    r19, 76(sp)");
asm("ldw    r20, 80(sp)");
asm("ldw    r21, 84(sp)");
asm("ldw    r22, 88(sp)");
asm("ldw    r23, 92(sp)");
asm("ldw    r24, 96(sp)");
asm("ldw    r25, 100(sp)"); // r25 = bt
asm("ldw    r26, 104(sp)"); // r26 = gp
// skip r27 because it is sp, and we did not save this on the stack
asm("ldw    r28, 112(sp)"); // r28 = fp
asm("ldw    r29, 116(sp)"); // r29 = ea
asm("ldw    r30, 120(sp)"); // r30 = ba
asm("ldw    r31, 124(sp)"); // r31 = ra

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

asm("eret");
}

/*****
 * Interrupt Service Routine
 * Determines what caused the interrupt and calls the appropriate
 * subroutine.
 *
 * ipending - Control register 4 which has the pending external interrupts
 *****/
void interrupt_handler(void) {
    int ipending;
    NIOS2_READ_IPENDING(ipending);
    if (ipending & 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;
}

```

Listing 11. Reset and exception handler C code.

```

#include "address_map_nios2.h"
#include "globals.h" // defines global values

extern volatile int pattern, shift_dir, shift_enable;
/*****
 * Interval timer interrupt service routine
 *
 * Shifts a PATTERN being displayed on the LED lights. The shift direction
 * is determined by the external variable key_dir.
 *****/
void interval_timer_ISR() {
    volatile int * interval_timer_ptr = (int *)TIMER_BASE;
    volatile int * LEDG_ptr           = (int *)LED_BASE; // LED address

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

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

    if (shift_enable == DISABLE) // check if shifting is disabled
        return;

    /* rotate the pattern shown on the LEDG lights */
    if (shift_dir == LEFT) // 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;
}

```

Listing 12. Interrupt service routine for the interval timer.

```

#include "address_map_nios2.h"
#include "globals.h" // defines global values

extern volatile int pattern, shift_dir, shift_enable;
/*****
 * Pushbutton - Interrupt Service Routine
 *
 * This routine checks which KEY has been pressed and updates the global
 * variables as required.
 *****/
void pushbutton_ISR(void) {
    volatile int * KEY_ptr          = (int *)KEY_BASE;
    volatile int * slider_switch_ptr = (int *)SW_BASE;
    int          press;

    press          = *(KEY_ptr + 3); // read the pushbutton interrupt register
    *(KEY_ptr + 3) = press;         // Clear the interrupt

    if (press & 0x1) // KEY0
        pattern = *slider_switch_ptr;

    if (press & 0x2) // KEY1
        shift_dir = shift_dir ^ 1;

    return;
}

```

Listing 13. Interrupt service routine for the pushbutton KEYS.

9.4 Audio

```

#include "address_map_nios2.h"

/* globals */
#define BUF_SIZE 80000 // about 10 seconds of buffer (@ 8K samples/sec)
#define BUF_THRESHOLD 96 // 75% of 128 word buffer

/* function prototypes */
void check_KEYS(int *, int *, int *);

/*****
 * This program performs the following:
 * 1. records audio for 10 seconds when KEY[0] is pressed. LEDR[0] is lit
 *    while recording.
 * 2. plays the recorded audio when KEY[1] is pressed. LEDR[1] is lit while
 *    playing.
 *****/
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 * red_LED_ptr = (int *)LED_BASE;
    volatile int * audio_ptr = (int *)AUDIO_BASE;

    /* used for audio record/playback */
    int fifospace;
    int record = 0, play = 0, buffer_index = 0;
    int left_buffer[BUF_SIZE];
    int right_buffer[BUF_SIZE];

    /* read and echo audio data */
    record = 0;
    play = 0;

    while (1) {
        check_KEYS(&record, &play, &buffer_index);
        if (record) {
            *(red_LED_ptr) = 0x1; // turn on LEDR[0]
            fifospace =
                *(audio_ptr + 1); // read the audio port fifospace register
            if ((fifospace & 0x000000FF) > BUF_THRESHOLD) // check RARC
            {
                // store data until the the audio-in FIFO is empty or the buffer
                // is full
                while ((fifospace & 0x000000FF) && (buffer_index < BUF_SIZE)) {
                    left_buffer[buffer_index] = *(audio_ptr + 2);
                    right_buffer[buffer_index] = *(audio_ptr + 3);
                    ++buffer_index;
                }

                if (buffer_index == BUF_SIZE) {

```

```

        // done recording
        record      = 0;
        *(red_LED_ptr) = 0x0; // turn off LEDR
    }
    fifospace = *(audio_ptr +
                1); // read the audio port fifospace register
    }
}
} else if (play) {
    *(red_LED_ptr) = 0x2; // turn on LEDR_1
    fifospace =
        *(audio_ptr + 1); // read the audio port fifospace register
    if ((fifospace & 0x00FF0000) > BUF_THRESHOLD) // check WSRC
    {
        // output data until the buffer is empty or the audio-out FIFO
        // is full
        while ((fifospace & 0x00FF0000) && (buffer_index < BUF_SIZE)) {
            *(audio_ptr + 2) = left_buffer[buffer_index];
            *(audio_ptr + 3) = right_buffer[buffer_index];
            ++buffer_index;

            if (buffer_index == BUF_SIZE) {
                // done playback
                play      = 0;
                *(red_LED_ptr) = 0x0; // turn off LEDR
            }
            fifospace = *(audio_ptr +
                        1); // read the audio port fifospace register
        }
    }
}
}
}

/*****
 * Subroutine to read KEYS
 *****/
void check_KEYS(int * KEY0, int * KEY1, int * counter) {
    volatile int * KEY_ptr   = (int *)KEY_BASE;
    volatile int * audio_ptr = (int *)AUDIO_BASE;
    int          KEY_value;

    KEY_value = *(KEY_ptr); // read the pushbutton KEY values
    while (*KEY_ptr)
        ; // wait for pushbutton KEY release

    if (KEY_value == 0x1) // check KEY0
    {
        // reset counter to start recording
        *counter = 0;
        // clear audio-in FIFO
    }
}

```

```
    *(audio_ptr) = 0x4;
    *(audio_ptr) = 0x0;

    *KEY0 = 1;
} else if (KEY_value == 0x2) // check KEY1
{
    // reset counter to start playback
    *counter = 0;
    // clear audio-out FIFO
    *(audio_ptr) = 0x8;
    *(audio_ptr) = 0x0;

    *KEY1 = 1;
}
}
```

Listing 14. An example of code that uses the audio port.

9.5 Video Out

```

#include "address_map_nios2.h"

/* function prototypes */
void video_text(int, int, char *);
void video_box(int, int, int, int, short);
int  resample_rgb(int, int);
int  get_data_bits(int);

#define STANDARD_X 320
#define STANDARD_Y 240
#define INTEL_BLUE 0x0071C5
/* global variables */
int screen_x;
int screen_y;
int res_offset;
int col_offset;

/*****
 * This program demonstrates use of the video in the computer system.
 * Draws a blue box on the video display, and places a text string inside the
 * box
 *****/
int main(void) {
    volatile int * video_resolution = (int *) (PIXEL_BUF_CTRL_BASE + 0x8);
    screen_x      = *video_resolution & 0xFFFF;
    screen_y      = (*video_resolution >> 16) & 0xFFFF;

    volatile int * rgb_status = (int *) (RGB_RESAMPLER_BASE);
    int          db           = get_data_bits(*rgb_status & 0x3F);

    /* check if resolution is smaller than the standard 320 x 240 */
    res_offset = (screen_x == 160) ? 1 : 0;

    /* check if number of data bits is less than the standard 16-bits */
    col_offset = (db == 8) ? 1 : 0;

    /* create a message to be displayed on the video and LCD displays */
    char text_top_row[40]   = "Intel FPGA\0";
    char text_bottom_row[40] = "Computer Systems\0";

    /* update color */
    short background_color = resample_rgb(db, INTEL_BLUE);

    video_text(35, 29, text_top_row);
    video_text(32, 30, text_bottom_row);
    video_box(0, 0, STANDARD_X, STANDARD_Y, 0); // clear the screen
    video_box(31 * 4, 28 * 4, 49 * 4 - 1, 32 * 4 - 1, background_color);
}

```

```

/*****
 * Subroutine to send a string of text to the video monitor
 *****/
void video_text(int x, int y, char * text_ptr) {
    int offset;
    volatile char * character_buffer =
        (char *)FPGA_CHAR_BASE; // video character buffer

    /* assume that the text string fits on one line */
    offset = (y << 7) + x;
    while (*(text_ptr)) {
        *(character_buffer + offset) =
            *(text_ptr); // write to the character buffer
        ++text_ptr;
        ++offset;
    }
}

/*****
 * Draw a filled rectangle on the video monitor
 * Takes in points assuming 320x240 resolution and adjusts based on differences
 * in resolution and color bits.
 *****/
void video_box(int x1, int y1, int x2, int y2, short pixel_color) {
    int pixel_buf_ptr = *(int *)PIXEL_BUF_CTRL_BASE;
    int pixel_ptr, row, col;
    int x_factor = 0x1 << (res_offset + col_offset);
    int y_factor = 0x1 << (res_offset);
    x1 = x1 / x_factor;
    x2 = x2 / x_factor;
    y1 = y1 / y_factor;
    y2 = y2 / y_factor;

    /* assume that the box coordinates are valid */
    for (row = y1; row <= y2; row++)
        for (col = x1; col <= x2; ++col) {
            pixel_ptr = pixel_buf_ptr +
                (row << (10 - res_offset - col_offset)) + (col << 1);
            *(short *)pixel_ptr = pixel_color; // set pixel color
        }
}

/*****
 * Resamples 24-bit color to 16-bit or 8-bit color
 *****/
int resample_rgb(int num_bits, int color) {
    if (num_bits == 8) {
        color = (((color >> 16) & 0x000000E0) | ((color >> 11) & 0x0000001C) |
            ((color >> 6) & 0x00000003));
        color = (color << 8) | color;
    } else if (num_bits == 16) {

```

```
        color = (((color >> 8) & 0x0000F800) | ((color >> 5) & 0x000007E0) |
                ((color >> 3) & 0x0000001F));
    }
    return color;
}

/*****
 * Finds the number of data bits from the mode
 *****/
int get_data_bits(int mode) {
    switch (mode) {
        case 0x0:
            return 1;
        case 0x7:
            return 8;
        case 0x11:
            return 8;
        case 0x12:
            return 9;
        case 0x14:
            return 16;
        case 0x17:
            return 24;
        case 0x19:
            return 30;
        case 0x31:
            return 8;
        case 0x32:
            return 12;
        case 0x33:
            return 16;
        case 0x37:
            return 32;
        case 0x39:
            return 40;
    }
}
```

Listing 15. An example of code that uses the video-out port.

9.6 PS/2

```

#include "address_map_nios2.h"

/* function prototypes */
void HEX_PS2(char, char, char);

/*****
 * This program demonstrates use of the PS/2 port by displaying the last three
 * bytes of data received from the PS/2 port on the HEX displays.
 *****/
int main(void) {
    /* Declare volatile pointers to I/O registers (volatile means that IO load
       and store instructions will be used to access these pointer locations,
       instead of regular memory loads and stores) */
    volatile int * PS2_ptr = (int *)PS2_BASE;

    int PS2_data, RVALID;
    char byte1 = 0, byte2 = 0, byte3 = 0;

    // PS/2 mouse needs to be reset (must be already plugged in)
    *(PS2_ptr) = 0xFF; // reset

    while (1) {
        PS2_data = *(PS2_ptr); // read the Data register in the PS/2 port
        RVALID = PS2_data & 0x8000; // extract the RVALID field
        if (RVALID) {
            /* shift the next data byte into the display */
            byte1 = byte2;
            byte2 = byte3;
            byte3 = PS2_data & 0xFF;
            HEX_PS2(byte1, byte2, byte3);

            if ((byte2 == (char)0xAA) && (byte3 == (char)0x00))
                // mouse inserted; initialize sending of data
                *(PS2_ptr) = 0xF4;
        }
    }
}

/*****
 * Subroutine to show a string of HEX data on the HEX displays
 *****/
void HEX_PS2(char b1, char b2, char b3) {
    volatile int * HEX3_HEX0_ptr = (int *)HEX3_HEX0_BASE;
    volatile int * HEX5_HEX4_ptr = (int *)HEX5_HEX4_BASE;

    /* SEVEN_SEGMENT_DECODE_TABLE gives the on/off settings for all segments in
       * a single 7-seg display in the DE1-SoC Computer, for the hex digits 0 - F
       */
    unsigned char seven_seg_decode_table[] = {

```

```
    0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07,
    0x7F, 0x67, 0x77, 0x7C, 0x39, 0x5E, 0x79, 0x71};
unsigned char hex_segs[] = {0, 0, 0, 0, 0, 0, 0, 0};
unsigned int  shift_buffer, nibble;
unsigned char code;
int          i;

shift_buffer = (b1 << 16) | (b2 << 8) | b3;
for (i = 0; i < 6; ++i) {
    nibble = shift_buffer & 0x0000000F; // character is in rightmost nibble
    code   = seven_seg_decode_table[nibble];
    hex_segs[i] = code;
    shift_buffer = shift_buffer >> 4;
}
/* drive the hex displays */
*(HEX3_HEX0_ptr) = *(int *) (hex_segs);
*(HEX5_HEX4_ptr) = *(int *) (hex_segs + 4);
}
```

Listing 16. An example of code that uses the PS/2 port.

9.7 Floating Point

```

/*****
 * This program demonstrates use of floating-point numbers
 *
 * It performs the following:
 *   1. reads two FP numbers from the Terminal window
 *   2. performs +, -, *, and / on the numbers
 *   3. prints the results on the Terminal window
 *   Note: Please enable "Echo input" in the terminal window
 *****/
#include <stdio.h>

int flush()
{
    while (getchar() != '\n')
        ;
    return 1;
}

int main(void)
{
    float x, y, add, sub, mult, div;

    while (1)
    {
        printf("Enter FP values X: ");

        while ((scanf("%f", &x) != 1) && flush())
            ; // get valid floating point value and flush the invalid input
        printf("%f\n", x); // echo the typed data to the Terminal window

        printf("Enter FP values Y: ");

        while ((scanf("%f", &y) != 1) && flush())
            ; // get valid floating point value and flush the invalid input
        printf("%f\n", y); // echo the typed data to the Terminal window

        add = x + y;
        sub = x - y;
        mult = x * y;
        div = x / y;
        printf("X + Y = %f\n", add);
        printf("X - Y = %f\n", sub);
        printf("X * Y = %f\n", mult);
        printf("X / Y = %f\n", div);
    }
}

```

Listing 17. An example of code that uses floating-point variables.

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.