For Quartus® Prime 24.1

1 Introduction

This document describes a computer system that can be implemented on the DE1-SoC development and education board, which is described in the `Teaching and Projects Boards` section of the FPGAcademy.org website. This system, called the *DE1-SoC Computer with Nios V*, is intended for use in experiments on computer organization and embedded systems.

To support such experiments, the computer system contains embedded processors, memory, basic I/O devices like switches and lights, audio and video devices, and various other I/O peripherals. The FPGA programming file that implements this system, as well as its design source files, can be obtained from its GitHub repository.

2 DE1-SoC Computer with Nios V Contents

A block diagram of the *DE1-SoC Computer with Nios V* is shown in Figure 1. As indicated in the figure, the components in this system are implemented utilizing both the FPGA and the *Hard Processor System* (HPS) inside Altera's Cyclone® V SoC chip. The FPGA implements two Nios V® processors and several peripheral ports: memory, timer modules, audio-in/out, video-in/out, PS/2, analog-to-digital, infrared receive/transmit, and parallel ports connected to switches and lights. The HPS comprises an ARM* Cortex* A9 dual-core processor and a set of peripheral devices. Some of these HPS peripheral devices can be accessed by Nios V. Instructions for using the HPS with the ARM processor can be found in the document entitled *DE1-SoC Computer System with ARM* Cortex* A9*, which is available on the FPGAcademy.org website.

2.1 Getting Started with the DE1-SoC Computer with Nios V

To make use of the *DE1-SoC Computer with Nios V* you need to be able to assemble software programs for the Nios V processor and then execute these programs in the computer system. There are two main approaches for getting started: using a simulation of the computer system, or using an FPGA board that implements the computer system in hardware.

2.1.1 Using the CPUlator Simulator

The *CPUlator* is a powerful and easy-to-use functional simulator that runs inside a web browser. It simulates the behavior of a whole computer system, including the processor, memory, and many types of I/O devices. The CPUlator simulator supports a variety of different computer systems, including the *DE1-SoC Computer with Nios V*.

The CPUlator user interface displays all of the information that a programmer needs to develop and debug software code running on the *DE1-SoC Computer with Nios V*. It shows (and allows you to edit) the values in the processor general-purpose and control registers, as well as the contents of memories in the computer system and the values of

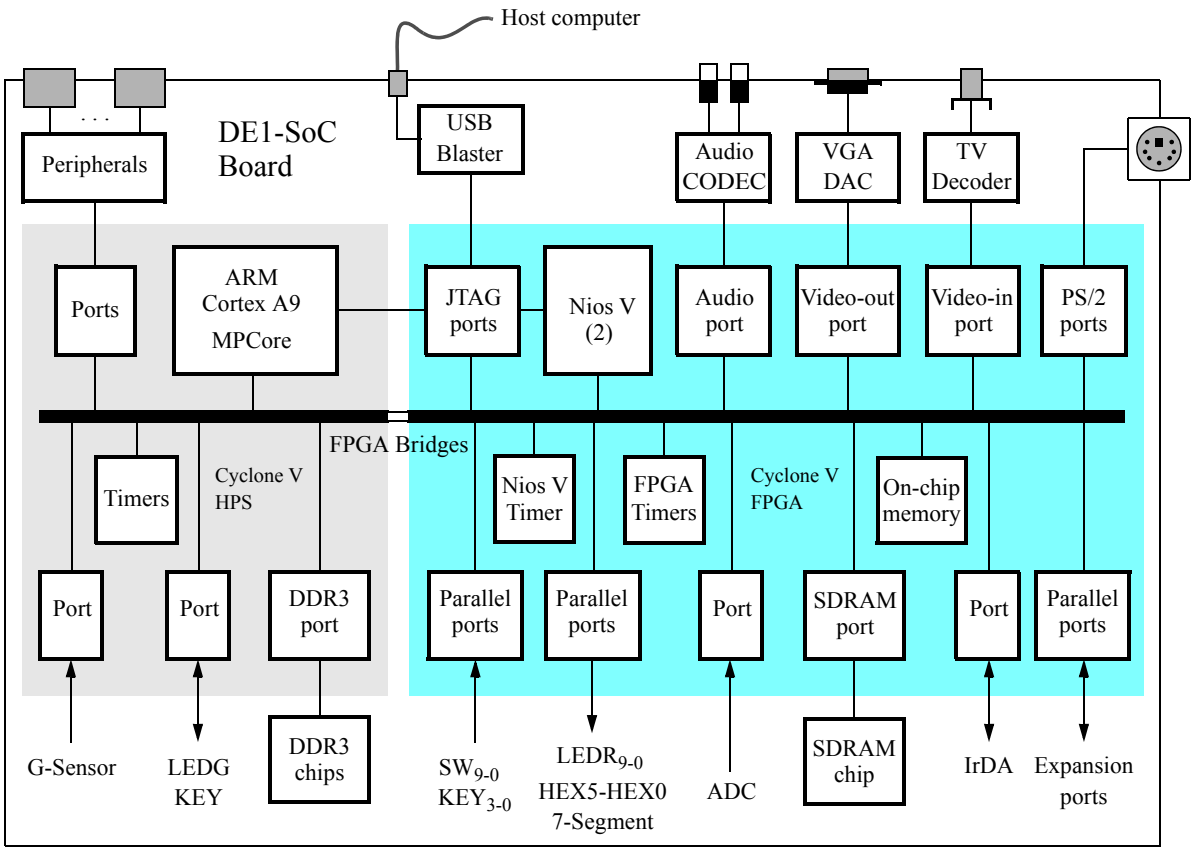


Figure 1. Block diagram of the *DE1-SoC Computer with Nios V*.

memory-mapped I/O device registers. The CPUlator allows software code, written either in assembly language or the C language, to be entered into the simulator, assembled to produce machine code, loaded into memory, and then executed. The user can set breakpoints in the machine code, single-step instructions, and perform any of the usual operations that are supported in typical debugging environments. A screen capture of the CPUlator user interface is shown in Figure 2. It displays the processor registers on the left-hand side (by default) of the screen, the program code in the middle, and graphical representations of I/O devices on the right-hand side.

2.1.2 Using the Monitor Program with an FPGA Hardware Board

The *DE1-SoC Computer with Nios V* can be implemented using a DE1-SoC hardware board. An easy way to begin working with this computer system is to make use of the utility called the *Monitor Program*. It provides an easy way to assemble/compile Nios V programs written in either assembly language or the C language. The Monitor Program, which can be downloaded from the `Software Tools` section of the FPGAcademy.org website, is an application program that runs on the host computer connected to the DE1-SoC board. The Monitor Program can be used to control the execution of code on Nios V, list (and edit) the contents of processor registers, display/edit the contents of memory on the DE1-SoC board, and similar operations. The Monitor Program includes the DE1-SoC Computer as a pre-designed system that can be downloaded onto the DE1-SoC board, as well as several sample programs in assembly language and C that show how to use the *DE1-SoC Computer with Nios V* peripheral devices.

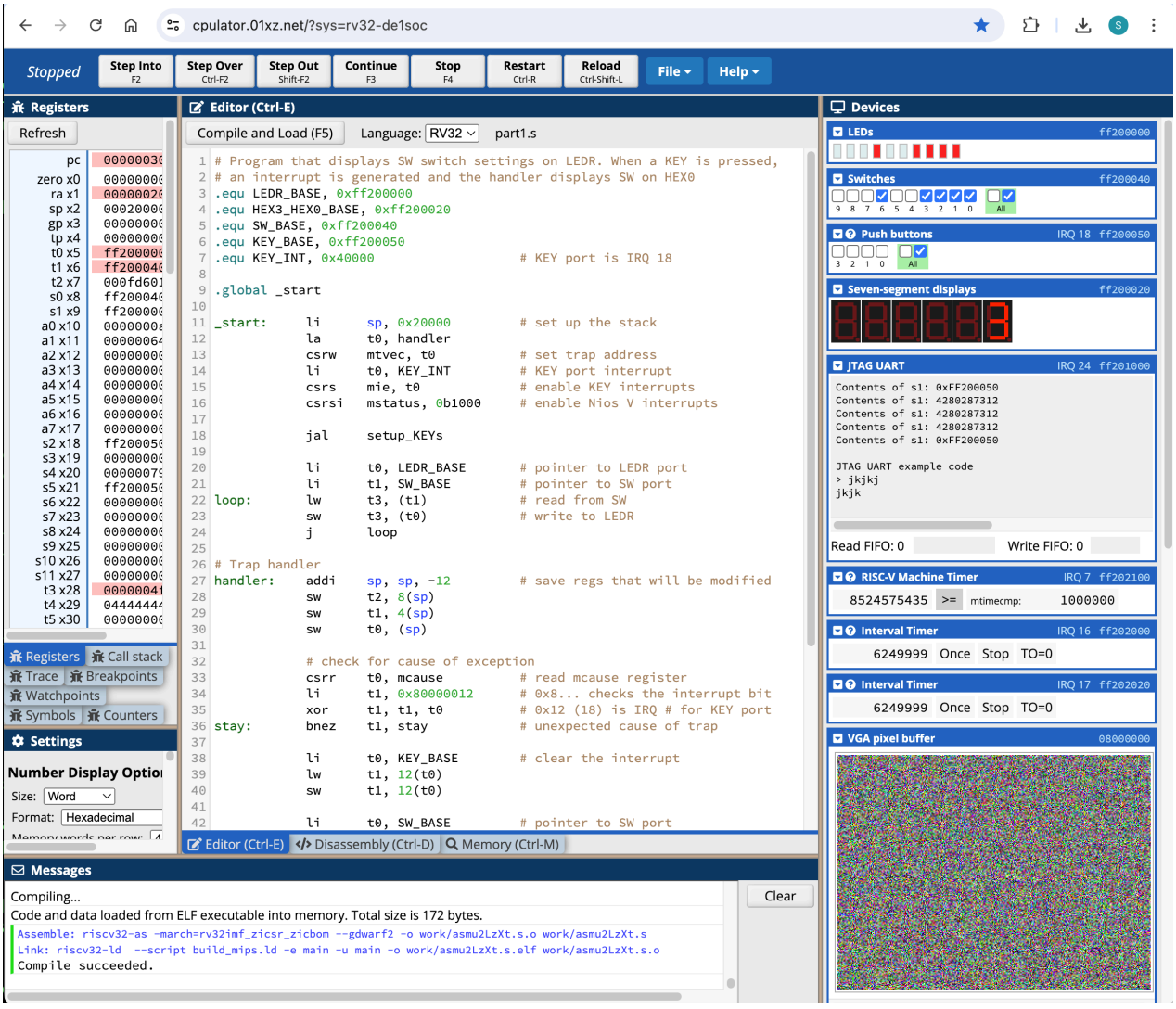


Figure 2. The CPUlator.

Some images that show how the *DE1-SoC Computer with Nios V* is integrated with the Monitor Program are given in Section **??**. An overview of the Monitor Program is available in the document *Monitor Program Tutorial for the Nios V Processor*, which is provided as part of the `Computer Organization System Design` tutorials on FPGAcademy.org.

2.1.3 Using GDB with an FPGA Hardware Board

The *Monitor Program* controls the FPGA hardware and the Nios V processor by using the industry-standard GNU Project Debugger (*GDB*). Instead of using the *Monitor Program*, you can debug code with the *GDB* tool directly.

2.2 Nios® V Processor

The Altera Nios® V processor is an implementation of the 32-bit RISC-V processor architecture. Three versions of Nios V exist, each with different features and capabilities. Documentation for these three versions, designated as *compact* (Nios V/c), *microcontroller* (Nios V/m), and *general purpose* (Nios V/g), can be found by searching on the Internet for keywords such as `Nios V versions`. The *DE1-SoC Computer with Nios V* includes two instances of the Nios V/m version. An overview of the Nios V processor can be found in the document *Introduction to Nios V*, which is available as part of the `Computer Organization and System Design` tutorials in the FPGAcademy.org website.

2.2.1 Nios V Machine Timer and Software Interrupt Registers

Nios V includes a 64-bit internal timer that is available to application programmers. The timer is reset to 0 when the DE1-SoC board is powered on, and then monotonically increases at the system clock rate, which is 100 MHz. The timer is accessible via two memory-mapped registers, called *mtime* (machine time) and *mtimecmp* (machine time compare). The *mtime* register provides the current timer value, and the *mtimecmp* register can be used to cause a timer interrupt. A Nios V timer interrupt will be pending whenever the value of *mtime* reaches or exceeds the value of *mtimecmp*. Interrupts are discussed in Section 3.

Since they are 64-bits wide, both *mtime* and *mtimecmp* comprise two 32-bit memory-mapped registers, one for the *low* word and the other for the *high* word. Nios V also contains a memory-mapped register called *msip* (machine software interrupt pending), which can be used by an application programmer to cause a *software interrupt*.

The *mtime*, *mtimecmp* and *msip* memory-mapped registers are are illustrated in Figure 3, which gives the assigned address of each register in the *DE1-SoC Computer with Nios V*.

Figure 3. Nios V memory-mapped registers.

2.3 Memory Components

The *DE1-SoC Computer with Nios V* has SDRAM and DDR3 memory ports, as well as two memory modules implemented using the on-chip memory inside the FPGA. These memories are described below.

2.3.1 SDRAM

An SDRAM Controller in the FPGA provides an interface to the 64 MB synchronous dynamic RAM (SDRAM) on the DE1-SoC board, which is organized as 32M x 16 bits. It is accessible by the Nios V processor using word (32-bit), halfword (16-bit), or byte operations, and is mapped to the address space 0x00000000 to 0x03FFFFFF.

2.3.2 DDR3 Memory

A 1 GB DDR3 memory is connected to the HPS part of the Cyclone® V SoC chip. The memory is organized as 256M x 32-bits, and is accessible using word accesses (32 bits), halfwords, and bytes. The Nios V processor can access the DDR3 memory using the addresses space 0x40000000 to 0x7FFFFFFF.

2.3.3 On-Chip Memory

A 256 KB memory is implemented inside the FPGA, organized as 64K x 32 bits. The Nios V processor can access this memory using addresses in the range 0x08000000 to 0x0803FFFF. This memory is used as a pixel buffer for the video-out and video-in ports.

2.3.4 On-Chip Memory Character Buffer

An 8 KB memory is implemented inside the FPGA for use as a character buffer for the video-out port, which is described in Section 4.2. The character buffer memory is organized as 8K x 8 bits, and spans the Nios V address range 0x09000000 to 0x09001FFF.

2.4 Parallel Ports

There are several parallel ports implemented in the FPGA that support input, output, and bidirectional transfers of data between the Nios V processor and I/O peripherals. As illustrated in Figure 4, 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.

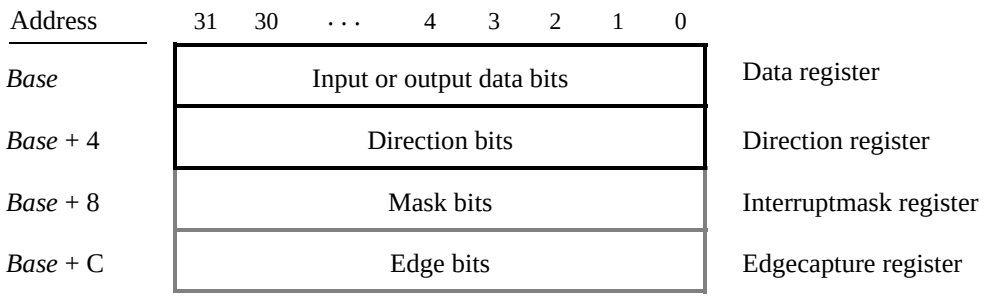


Figure 4. Parallel port registers in the *DE1-SoC Computer with Nios V*.

Some of the parallel ports have registers at addresses *Base* + 8 and *Base* + C, as indicated in Figure 4. These registers are discussed in Section 3.

2.4.1 Red LED Parallel Port

The red lights $LEDR_{9-0}$ on the DE1-SoC board are driven by an output parallel port, as illustrated in Figure 5. The port contains a 10-bit *Data* register, which has the address 0xFF200000. This register can be written or read by the processor using word accesses, and the upper bits not used in the registers are ignored.

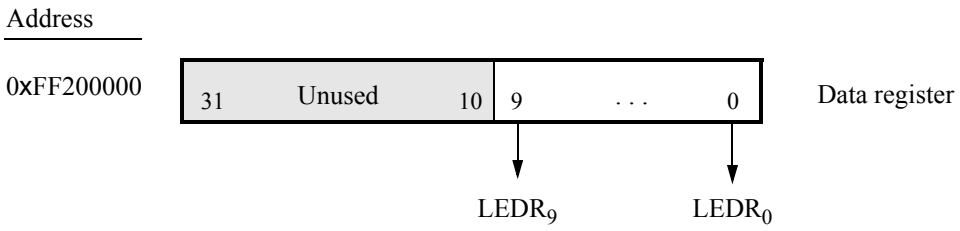


Figure 5. Output parallel port for *LEDR*.

2.4.2 7-Segment Displays Parallel Port

There are two parallel ports connected to the 7-segment displays on the DE1-SoC board, each of which comprises a 32-bit write-only *Data* register. As indicated in Figure 6, the register at address 0xFF200020 drives digits *HEX3* to *HEX0*, and the register at address 0xFF200030 drives digits *HEX5* and *HEX4*. Data can be written into these two registers, and read back, by using word operations. This data directly controls the segments of each display, according to the bit locations given in Figure 6. The locations of segments 6 to 0 in each seven-segment display on the DE1-SoC board is illustrated on the right side of the figure.

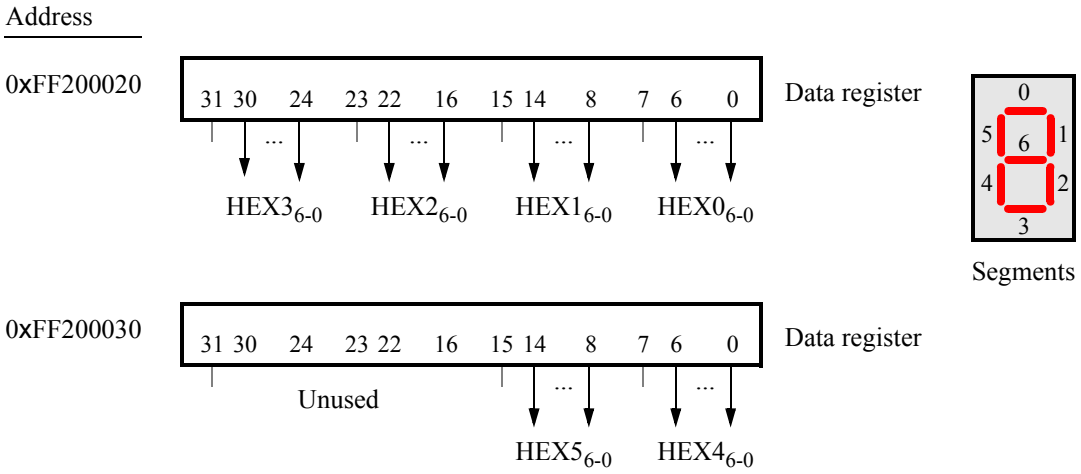


Figure 6. Bit locations for the 7-segment displays parallel ports.

2.4.3 Slider Switch Parallel Port

The SW_{9-0} slider switches on the DE1-SoC board are connected to an input parallel port. As illustrated in Figure 7, this port comprises a 10-bit read-only *Data* register, which is mapped to address 0xFF200040.

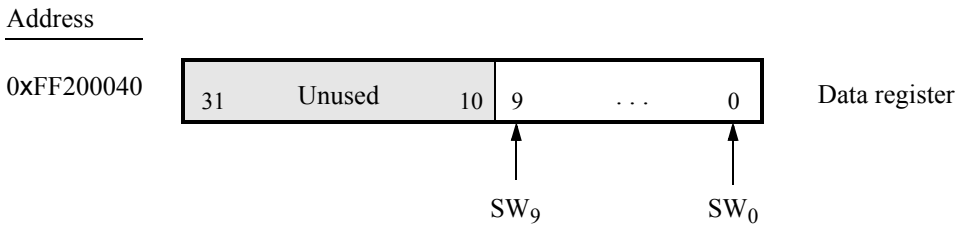


Figure 7. *Data* register in the slider switch parallel port.

2.4.4 Pushbutton Key Parallel Port

The parallel port connected to the KEY_{3-0} pushbutton switches on the DE1-SoC board comprises three 4-bit registers, as shown in Figure 8. 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 KEY_{3-0}. The other two registers shown in Figure 8, at addresses 0xFF200058 and 0xFF20005C, are discussed in Section 3.

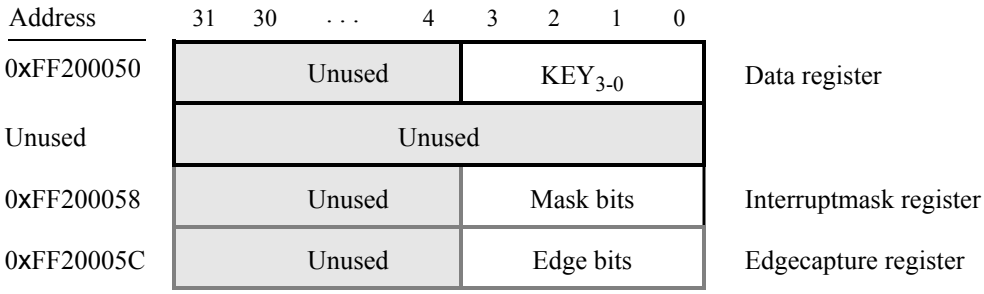


Figure 8. Registers used in the pushbutton parallel port.

2.4.5 Expansion Parallel Port

The *DE1-SoC Computer with Nios V* includes two bidirectional parallel ports that are connected to the *JP1* and *JP2* 40-pin headers on the DE1-SoC board. These parallel ports include the four 32-bit registers that were described previously for Figure 4. The base address of the port for JP1 is 0xFF200060, and for JP2 is 0xFF200070. Figure 9 gives a diagram of the 40-pin connectors on the DE1-SoC 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 the 40-pin header 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.

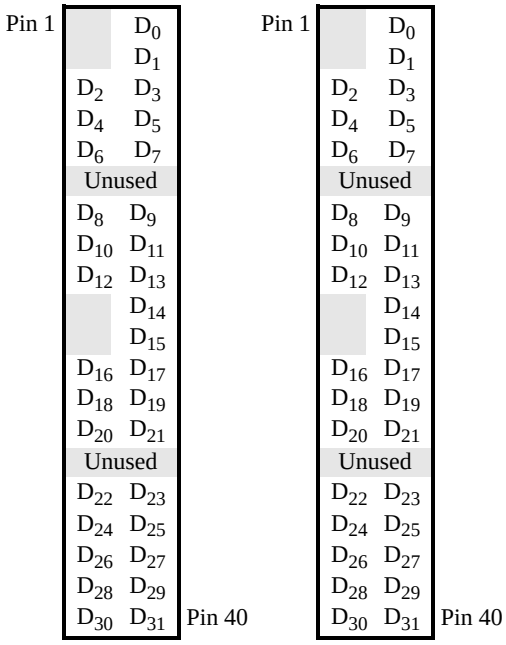


Figure 9. Assignment of parallel port bits to pins on *JP1* and *JP2*.

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

The *DE1-SoC Computer with Nios V* provides a convenient platform for experimenting with Nios V assembly language code, or C code. A simple example of such code is provided in the Appendix in Listings 1 and 2. Each of these listing *includes* a file that specifies the memory-mapped addresses of all peripheral devices in the *DE1-SoC Computer with Nios V*. These include files, called *address_map_niosv.s* and *address_map_niosv.h*, are provided in Listings 11 and 12. These include files are also used in other code samples described in this document.

The code in Listing 1 and 2 displays the values of the SW switches on the LED lights, and also shows a rotating pattern on the LEDs. This pattern is shifted in a loop, using a software delay 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 KEY is pressed, the program waits in a loop until it is released and then continues to display the pattern.

The source code files shown in Listings 1 and 2 are distributed as part of the Monitor Program. The files can be found under the heading *sample programs*, and are identified by the name *Getting Started*.

2.5 JTAG* Port

The JTAG* port implements a communication link between the DE1-SoC board and its host computer. This link can be used by the Altera Quartus® Prime software to transfer FPGA programming files into the DE1-SoC board, and by the Monitor Program, discussed in Section **??**. 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 V processor. The programming interface of the JTAG UART consists of two 32-bit registers, as shown in Figure 10. The register

mapped to address 0xFF201000 is called the *Data* register and the register mapped to address 0xFF201004 is called the *Control* register.

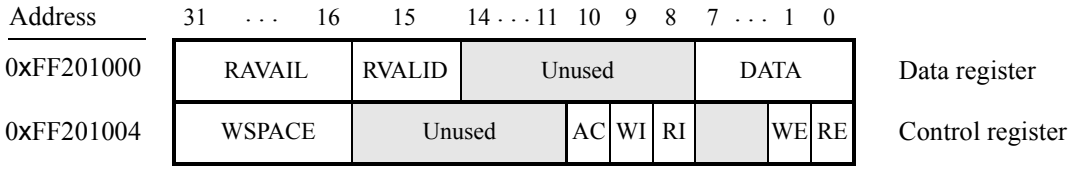


Figure 10. JTAG UART registers.

When character data from the host computer is received by the JTAG UART it is stored in a 64-character FIFO. The number of characters currently stored in this FIFO is indicated in the field *RAVAIL*, which are bits 31−16 of the *Data* register. If the receive FIFO overflows, then additional data is lost. When data is present in the receive FIFO, then the value of *RAVAIL* will be greater than 0 and the value of bit 15, *RVALID*, will be 1. Reading the character at the head of the FIFO, which is provided in bits 7 − 0, decrements the value of *RAVAIL* by one and returns this decremented value as part of the read operation. If no data is present in the receive FIFO, then *RVALID* will be set to 0 and the data in bits 7 − 0 is undefined.

The JTAG UART also includes a 64-character FIFO that stores data waiting to be transmitted to the host computer. Character data is loaded into this FIFO by performing a write to bits 7−0 of the *Data* register in Figure 10. Note that writing into this register has no effect on received data. The amount of space, *WSPACE*, currently available in the transmit FIFO is provided in bits 31−16 of the *Control* register. If the transmit FIFO is full, then any characters written to the *Data* register will be lost.

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

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

2.5.1 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. In the *CPUlator* simulator, there is a JTAG window that allows text to be typed and echoed. If the program is executed by using the 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 Monitor Program. The files can be found under the heading *sample programs*, and are identified by the name *JTAG UART*.

2.6 Interval Timers

The *DE1-SoC Computer with Nios V* includes a timer module implemented in the FPGA that can be used by the Nios V 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 11. 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.

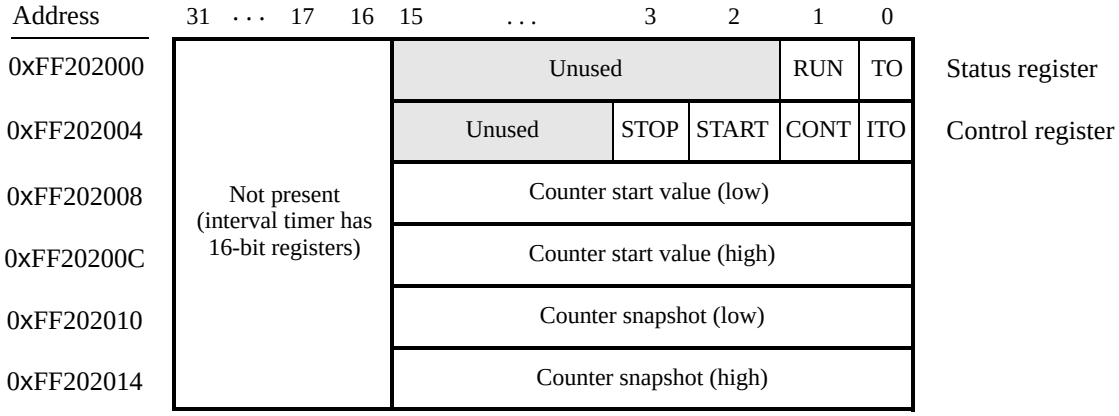


Figure 11. Interval timer registers.

- *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 gives a timer period of 125 msec. To achieve this period, the starting value of the count is 100 MHz × 125 msec = 12.5×10^6. 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 V processor has exclusive access to two interval timers, at the addresses given above.

2.7 G-Sensor

The *DE1-SoC Computer with Nios V* includes a 3D accelerometer (G-sensor) that is connected to the HPS. The Nios V processor can access this device via an I2C serial interface at the base address 0xFFC04000. More details can be found in the tutorial *Using the Accelerometer on DE-series Boards*.

3 Exceptions and Interrupts

The reset address of the Nios V processor in the *DE1-SoC Computer with Nios V* is set to 0x00000000. The address used for the trap handler for all other exceptions and interrupts can be set by the programmer (by writing to the *mtvec* control register). Table 1 gives the assignment of IRQ numbers to each of the I/O peripherals in the system. The rest of this section describes the interrupt behavior associated with the Nios V machine timer, the FPGA interval timer, parallel ports, and serial ports.

Device Name	IRQ #
Nios V software interrupt	3
Nios V machine timer	7
Interval timer	16
Second Interval timer	17
Pushbutton KEY port	18
Audio port	21
PS/2 port	22
PS/2 port dual	23
JTAG port	24
IrDA port	25
Serial port	26
JP1 Expansion port	27
JP2 Expansion port	28

Table 1. Hardware IRQ interrupt assignment for the *DE1-SoC Computer with Nios V*.

3.1 Interrupts from the Nios V Software Interrupts and Machine Timer

The IRQ numbers for the Nios V software interrupts register and machine timer are not system dependent and are part of the processor specification. The procedure that can be used to set up and handle these interrupts is described in the document *Introduction to Nios V*, which is available as part of the `Computer Organization and System Design` tutorials in the FPGAcademy.org website.

3.2 Interrupts from the FPGA Interval Timer

Figure 11, in Section 2.6, shows six registers that are associated with the interval timer. As we said in Section 2.6, 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.

3.3 Interrupts from Parallel Ports

Parallel ports were illustrated in Figure 4, which is reproduced as Figure 12. 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. A bit in the *Edgecapture* register can be cleared to 0 by writing a 1 into the corresponding bit position, which clears any associated interrupt.

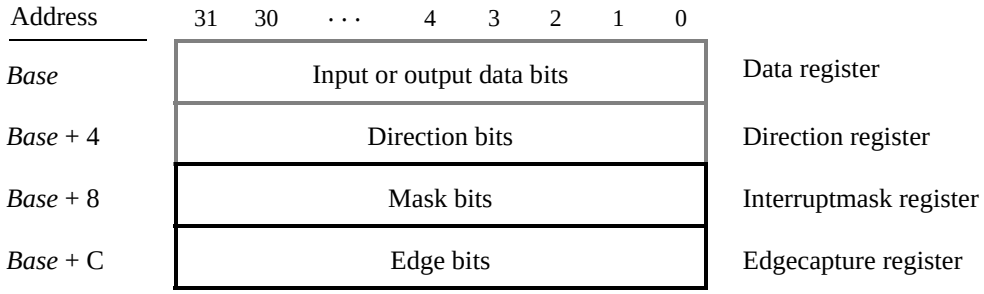


Figure 12. Registers used for interrupts from the parallel ports.

3.3.1 Interrupts from the Pushbutton KEY Port

Figure 8, reproduced as Figure 13, shows the registers associated with the pushbutton KEY port. The *Interruptmask* register allows interrupts to be generated when a key is pressed. Interrupts can be enabled individually for each key by setting its *Interruptmask* bit to 1. When a key is pressed, the corresponding bit in the *Edgecapture* register is set to 1 by the parallel port. This bit remains 1 until cleared to 0 by software. An interrupt service routine can read the *Edgecapture* register to determine which key/s has/have been pressed. An *Edgecapture* register bit can be cleared

by writing a logic value 1 into the bit position. Clearing the bit resets the corresponding interrupt signal being sent to the processor.

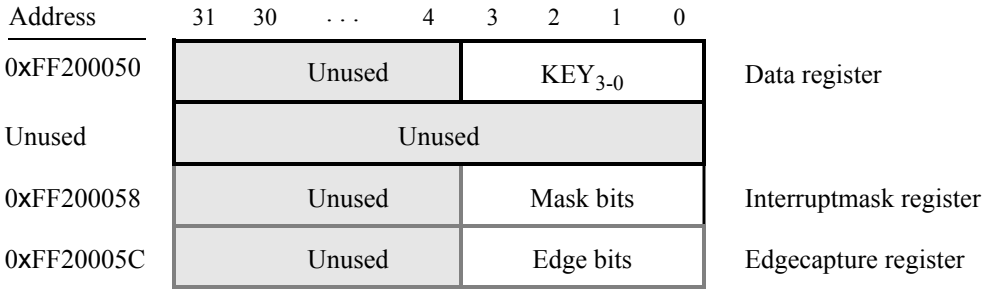


Figure 13. Registers used for interrupts from the pushbutton KEY port.

3.4 Interrupts from the JTAG* UART

Figure 10, reproduced as Figure 14, shows the *Data* and *Control* registers of the JTAG UART. As we said in Section 2.5, *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 14 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.

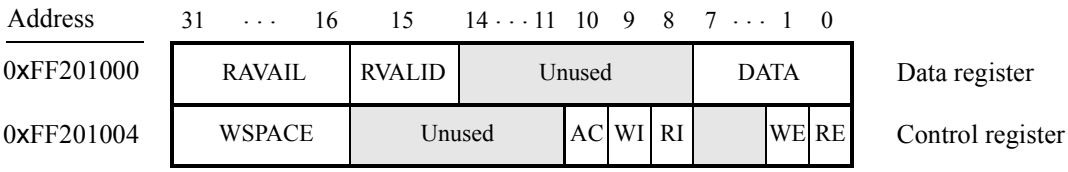


Figure 14. Interrupt bits in the JTAG UART registers.

3.5 Using Interrupts with Assembly Language Code

An example of assembly language code for the *DE1-SoC Computer with Nios V* that uses interrupts is shown in Listing 5. When this code is executed on the DE1-SoC board it first sets up interrupts from three devices: the Nios V machine timer, an FPGA interval timer, and the pushbutton KEY port. The code to initialize these devices is given in Lines 141 to 175 in Part (*d*) of Listing 5. Line 24 in Listing 5(*a*) initializes the stack pointer to the bottom of the 64 MB SDRAM on the DE1-SoC, and Lines 25 to 27 initialize the three interrupting devices. Interrupts are enabled in Lines 30 to 37. First, the address of the trap handler routine is written into the *mtvec* register, and then software interrupts, machine timer, interval timer, and KEY port interrupts are enabled by setting bits b_3, b_7, b_{16} and b_{18}, respectively, of the machine interrupt enable (*mie*) register. Finally, interrupts are enabled in Nios V by setting bit b_3 of the *mstatus* register.

Next, in Lines 40 to 42 the program makes a software interrupt occur, to illustrate how this is done. Finally, the main program loops in between Lines 51 and 57 while responding to interrupts from the timers and the KEY pushbutton port.

The trap handler is given in Lines 59 to 89. After first saving registers that will be modified, it reads the value of the *mcause* register. Based on this value, the trap handler calls the appropriate interrupt service routine.

The interrupt service routine for the software interrupt, in Lines 91 to 97, turns on most of the red lights in the LEDR port, to provide a visual indication of its execution.

The interrupt service routine for the Nios V machine timer, in Lines 99 to 114, adjusts the *mtimecmp* value for the next interrupt, and increments a counter variable. The main program displays this counter as a binary number on the red lights LEDR, which will increment for every timer interrupt.

The interrupt service routine for the FPGA interval timer, in Lines 116 to 129, increments a one-digit decimal counter. The main program displays this counter on the 7-segment display HEX0. The counter either increments or decrements, in the range 0 to 9. When a KEY is pressed, its corresponding interrupt service routine, in Lines 131 to 139, reverses the direction of counting on HEX0.

The remaining lines of code, in Listing 5(*e*), provide a subroutine for converting decimal digits to 7-segment display codes, and define the global variables that are used in the program.

3.6 Using Interrupts with C Code

An example of C code for the *DE1-SoC Computer with Nios V* that uses interrupts is shown in Listing 6. This code performs the same operations as the code in Listing 5. Lines 1 to 22 in the code declare some symbols, function prototypes, and global variables that are needed in the program. The function prototype for the *handler* subroutine, which is the trap handler in this program, is assigned the attribute `interrupt ("machine")`. This attribute instructs the C compiler to generate the appropriate assembly-language code for an interrupt handler: it saves and restores all registers that could be modified while the interrupt is being handled, and it returns to the interrupted program by using the `mret` instruction.

The main program declares pointers for accessing I/O devices in Lines 45 to 47. These pointers are given the `volatile` keyword, which tells the compiler that the value of the variables may change at any time, even if not modified in the code where they are declared (in this case the values may be modified by the interrupt service routines). Lines 49 to 51 in the code call subroutines that enable interrupts in the Nios V machine timer, the FPGA interval timer, and the KEY port.

Interrupts are enabled in the C code in lines 53 to 66 by inserting assembly-language code using the GNU C-compiler's `__asm__` inline assembly feature. The steps performed by these lines of code are the same as those in Lines 29 to 37 of Listing 5.

Inline assembly-language code is also used in the *handler* routine, in Line 84 in Part (*b*) of Listing 6, to read the Nios V *mcause* register. The handler then calls the appropriate interrupt service routine. As mentioned above, the *handler* saves and restores all temporary registers, and returns to the main program using the `mret` instruction, because the handler is declared with the `interrupt ("machine")` attribute.

4 Media Components

This section describes the audio in/out, video-out, video-in, PS/2, IrDA*, and ADC ports, as well as floating point support.

4.1 Audio In/Out Port

The *DE1-SoC Computer with Nios V* includes an audio port that is connected to the audio CODEC (COder/DECoder) chip on the DE1-SoC 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 DE1-SoC 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 15. 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 V 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 that 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 15 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.

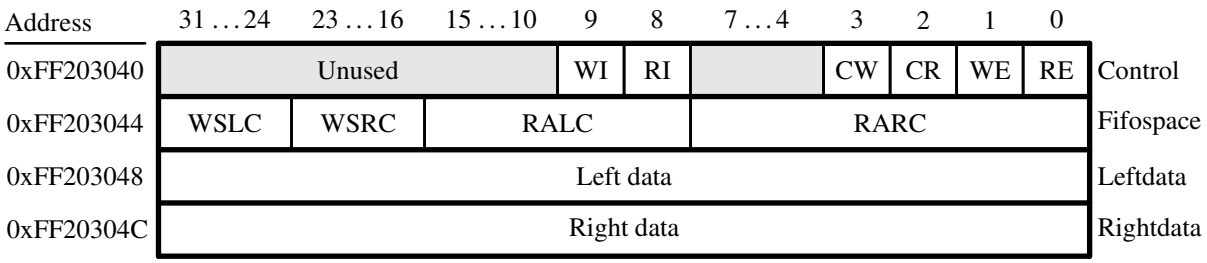


Figure 15. Audio port registers.

The read-only *Fifospace* register in Figure 15 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 7. 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 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 *DE1-SoC Computer with Nios V* 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 16, 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.

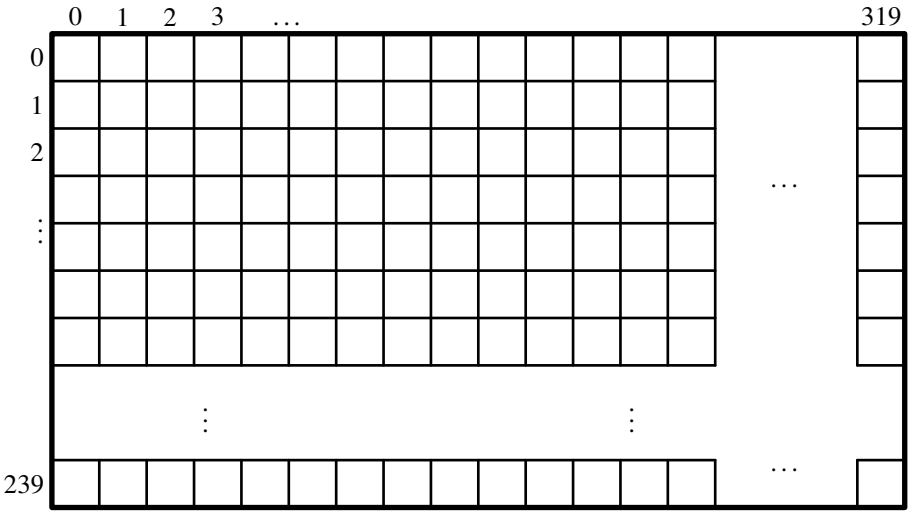
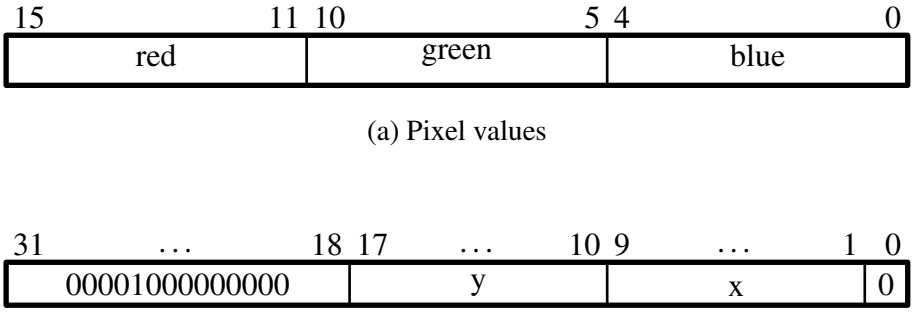


Figure 16. Pixel buffer coordinates.

Figure 17*a* 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 17, pixels are addressed in the pixel buffer by using the combination of a *base* address and an *x,y* offset. In the DE1-SoC 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* + $(00000000\ 000000001\ 0)_2$ = 0x08000002, the pixel 0,1 has the address *base* + $(00000001\ 000000000\ 0)_2$ = 0x08000400, and the pixel at location 319,239 has the address *base* + $(11101111\ 100111111\ 0)_2$ = 0x0803BE7E.

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



(a) Pixel values

(b) Pixel address

Figure 17. Pixel values and addresses.

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 DE1-SoC 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 DE1-SoC Computer video stream is 0x14, which corresponds to no alpha, color, 16-bit RGB (5-bit Red, 6-bit Green, 5-bit Blue). More information can be found in the documentation called *Video IP Cores for Altera DE-Series Computer Systems*, available as part of the `Hardware Components` tutorials on the FPGAcademy.org website.

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 DE1-SoC 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 DE1-SoC Computer, which has the base address 0x00000000. Note that the pixel buffer cannot be located in the DDR3 memory in the DE1-SoC 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

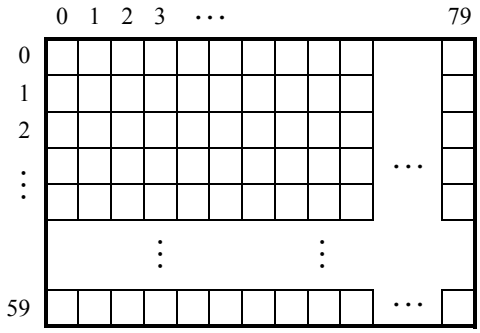
In a typical application the pixel buffer controller is used as follows. While the image contained in the pixel buffer that is pointed to by the Buffer register is being displayed, a new image is drawn into the pixel buffer pointed to by the Backbuffer register. When this new image is ready to be displayed, 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.

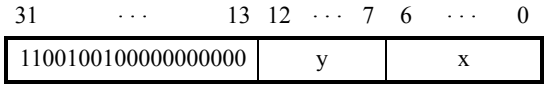
In Table 2 the default values of the status register fields in the DE1-SoC Computer are used when forming pixel addresses. The defaults are $n = 9$, $m = 8$, and $A = 0$. If the pixel buffer controller is changed to provide different values of these fields, then the way in which pixel addresses are formed 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 DE1-SoC board. As illustrated in Figure 18*a*, 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 18*a* 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 18 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 18. 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 8. 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 8 is included in the sample program called *Video* that is distributed with the Monitor Program.

4.3 Video-in Port

The *DE1-SoC Computer with Nios V* includes a video-in port for use with the composite video-in connector on the DE1-SoC 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 19. 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 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.

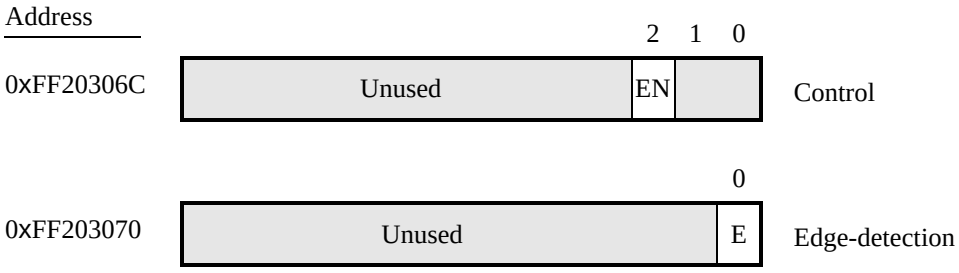


Figure 19. The video-in port programming interface.

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 DE1-SoC 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 *DE1-SoC Computer with Nios V* is reset. For typical use of the *DE1-SoC Computer with Nios V* it is not necessary to modify any of these default settings.

4.5 PS/2 Port

The *DE1-SoC Computer with Nios V* 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 20. 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.

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.

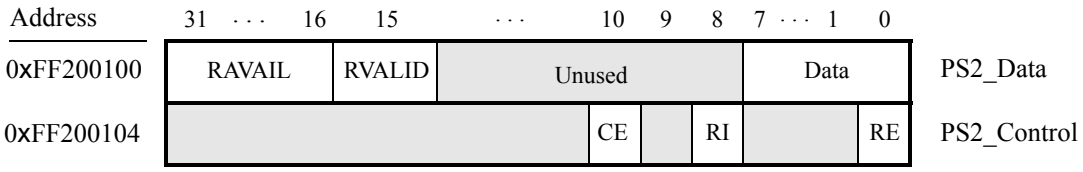


Figure 20. PS/2 port registers.

A fragment of C code that uses the PS/2 port is given in Listing 9. 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 $byte_1$, $byte_2$, and $byte_3$. This code is included as part of a sample program called *PS2* that is distributed with the Monitor Program.

4.5.1 PS/2 Port Dual

A second PS/2 port is included 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 DE1-SoC board through this cable. The PS/2 port dual has the same registers as the PS/2 port shown in Listing 9, except that the base address of its *PS2_Data* register is 0xFF200108 and the base address of its *PS2_Control* register is 0xFF20010C.

4.6 IrDA* Infrared Serial Port

The IrDA port in the *DE1-SoC Computer with Nios V* implements a UART that is connected to the infrared transmit/receive device on the DE1-SoC board. This UART is configured for 8-bit data, one stop bit, and no 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 21. The register at address 0xFF201020 is referred to as the *Data* register, and the register at address 0xFF201024 is called the *Control* register.

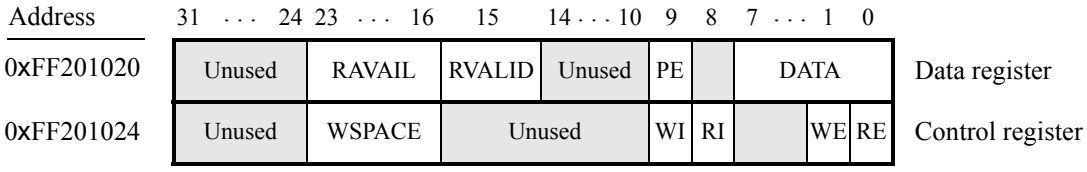


Figure 21. IrDA serial port UART registers.

When character data is received from the IrDA chip it is stored in a 256-character FIFO in the UART. As illustrated 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 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 IrDA device. Character data is loaded into this register 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 23−16 of the *Control* register, as indicated in Figure 21. If the transmit FIFO is full, then any additional characters written to the *Data* register will be lost.

The *RE* and *WE* bits in the *Control* register are used to enable Nios V 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.

4.7 Analog-to-Digital Conversion Port

The Analog-to-Digital Conversion (ADC) Port provides access to the eight-channel, 12-bit analog-to-digital converter on the DE1-SoC board. As illustrated in Figure 22, the ADC port comprises eight 12-bit registers starting at the base address 0xFF204000. The first two registers have dual purposes, acting as both data and control registers. By default, the ADC port updates the A-to-D conversion results for all ports only when instructed to do so. Writing to the control register at address 0xFF204000 causes this update to occur. Reading from the register at address 0xFF204000 provides the conversion data for channel 0. Reading from the other seven registers provides the conversion data for the corresponding channels. It is also possible to have the ADC port continually request A-to-D conversion data for all channels. This is done by writing the value 1 to the control register at address 0xFF204004. The *R* bit of each channel register in Figure 22 is used in Auto-update mode. *R* is set to 1 when its corresponding channel is refreshed and set to 0 when the channel is read.

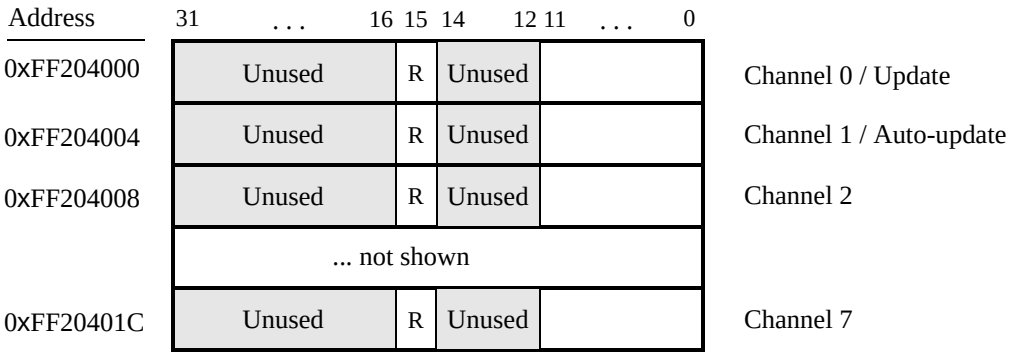


Figure 22. ADC port registers.

Figure 23 shows the connector on the DE1-SoC board that is used with the ADC port. Analog signals in the range of 0 V to the V_{CC5} power-supply voltage can be connected to the pins for channels 0 to 7.

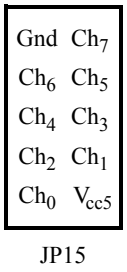


Figure 23. ADC connector.

4.8 Floating-point Hardware

The Nios V/g processor in the 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 10. When this code is compiled, it may be necessary to pass special argument to the C compiler to instruct it to use the floating-point hardware support.

5 Modifying the DE1-SoC Computer with Nios V

It is possible to modify the *DE1-SoC Computer with Nios V* by using Altera's Quartus® Prime software and Platform Designer tool. Instructions for using this software are provided as part of the `Computer Organization and System Design` tutorials on the FPGAcademy.org website. To modify the system it is first necessary to make an editable copy of the *DE1-SoC Computer with Nios V*. 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 Platform Designer software to make any desired changes.

Table 4 lists the names of the Platform Designer IP cores that are used in this system. When the *DE1-SoC Computer with Nios V* design files are opened in the Quartus Prime software, these cores can be examined using the Platform Designer System Integration tool. Each core has a number of settings that are selectable in the Platform Designer System Integration tool, and includes a datasheet that provides detailed documentation.

The steps needed to modify the system are:

1. Make of copy of the design source files for the *DE1-SoC Computer with Nios V* from the its GitHub repository.
2. Open the top-level project file (*.*qpf*) in the Quartus Prime software
3. Open the Platform Designer System Integration tool in the Quartus Prime software, and modify the system as desired
4. Generate the modified system by using the Platform Designer System Integration tool
5. It may be necessary to modify the Verilog code in the top-level module of the project, if any I/O peripherals have been added or removed from the system
6. Compile the project in the Quartus Prime software
7. Download the modified system into the DE1-SoC board

Note: to compile and use a new version of the *DE1-SoC Computer with Nios V* it may be necessary to request a license from Altera that allows you to create circuit that includes the Nios V processor.

I/O Peripheral	Qsys Core
SDRAM	SDRAM Controller
On-chip memory character buffer	Character Buffer for VGA Display
Red 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
Video In port	DMA Controller

Table 4. Platform Designer cores used in the *DE1-SoC Computer with Nios V*.

6 Making the System the Default Configuration

The *DE1-SoC Computer with Nios V* can be loaded into the nonvolatile FPGA configuration memory on the DE1-SoC board, so that it becomes the default system whenever the board is powered on. Instructions for configuring the DE1-SoC board in this manner can be found in the tutorial *Introduction to the Quartus Prime Software*, which is available as part of the `Digital Logic Hardware Design` tutorials in the FPGAcademy.org website.

7 Memory Layout

Table 5 summarizes the memory map used in the DE1-SoC Computer.

Base Address	End Address	I/O Peripheral
0x00000000	0x03FFFFFF	SDRAM
0x08000000	0x0803FFFF	FPGA On-chip Memory
0x09000000	0x09001FFF	FPGA On-chip Memory Character Buffer
0x40000000	0x7FFFFFFF	DDR3 Memory
0xFF200000	0xFF20000F	Red LEDs
0xFF200020	0xFF20002F	7-segment HEX3−HEX0 Displays
0xFF200030	0xFF20003F	7-segment HEX5−HEX4 Displays
0xFF200040	0xFF20004F	Slider Switches
0xFF200050	0xFF20005F	Pushbutton KEYs
0xFF200060	0xFF20006F	JP1 Expansion
0xFF200070	0xFF20007F	JP2 Expansion
0xFF200100	0xFF200107	PS/2
0xFF200108	0xFF20010F	PS/2 Dual
0xFF201000	0xFF201007	JTAG UART
0xFF201020	0xFF201027	Infrared (IrDA)
0xFF202000	0xFF20201F	Interval Timer
0xFF202020	0xFF20202F	Second Interval Timer
0xFF202100	0xFF202114	Nios V Machine Timer and Software Interrupts Registers
0xFF203000	0xFF20301F	Audio/video Configuration
0xFF203020	0xFF20302F	Pixel Buffer Control
0xFF203030	0xFF203037	Character Buffer Control
0xFF203040	0xFF20304F	Audio
0xFF203060	0xFF203070	Video-in
0xFF204000	0xFF20401F	ADC
0xFFC04000	0xFFC040FC	HPS I2C0

Table 5. Memory layout used in the DE1-SoC Computer.

8 Appendix

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

8.1 Parallel Ports

```
.include "address_map_niosv.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
 ********************************************************************************/

.global _start
_start:
        la      s0, SW_BASE         # SW slider switch base address
        la      s1, LED_BASE        # LED base address
        la      s2, KEY_BASE        # pushbutton KEY base address
        la      t1, LED_bits
        lw      t0, (t1)            # load pattern for LED lights
DO_DISPLAY:
        lw      t1, (s0)            # load slider switches

        lw      t2, (s2)            # load pushbuttons
        beqz    t2, NO_BUTTON
        mv      t0, t1              # use SW switch values as LED pattern
WAIT:
        lw      t3, (s2)            # load pushbuttons
        bnez    t3, WAIT            # wait for button release
NO_BUTTON:
        sw      t0, (s1)            # write to the LEDs
        srli    t1, t0, 10          # perform some operations to rotate
        slli    t0, t0, 1           # the 10-bit pattern
        or      t0, t0, t1          # completes the "rotate" operation

        li      t2, 1500000         # delay counter
DELAY:
        addi    t2, t2, -1
        bnez    t2, DELAY

        j       DO_DISPLAY

LED_bits:
.word       0x0000030F          # 10-bit pattern
```

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

```
#include "address_map_niosv.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.

8.2 JTAG* UART

```
.include    "address_map_niosVm.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
 *****************************************************************************/

.global _start
_start:
        la      s0, JTAG_UART_BASE  # JTAG UART base address

/* print a text string */
        la      s1, TEXT_STRING
LOOP:
        lb      a0, 0(s1)
        beqz    a0, GET_JTAG        # string is null-terminated
        jal     PUT_JTAG
        addi    s1, s1, 1
        j       LOOP

/* read and echo characters */
GET_JTAG:
        lw      t0, 0(s0)           # read the JTAG UART data register
        li      t1, 0x8000
        and     t1, t1, t0          # check if there is new data
        beqz    t1, GET_JTAG        # if no data, wait
        andi    a0, t0, 0x00ff      # the data is in the least significant byte

        jal     PUT_JTAG            # echo character
        j       GET_JTAG
```

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

```
/********************************************************************************
 * Subroutine to send a character to the JTAG UART
 *       a0  = character to send
 *       s0  = JTAG UART base address
 ********************************************************************************/
.global PUT_JTAG
PUT_JTAG:
/* save any modified registers */
        lw      t0, 4(s0)           # read the JTAG UART control register
        lui     t1, 0xffff0         # t1 = 0xffff0000
        and     t0, t0, t1          # check for write space
        beqz    t0, END_PUT         # if no space, ignore the character
        sw      a0, 0(s0)           # send the character

END_PUT:
        ret

/*******************************************************************************/

TEXT_STRING:
        .asciz  "\nJTAG UART example code\n> "
```

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*).

8.3 Interrupts

```
1  .include    "address_map_niosv.s"
2  /*****************************************************************************
3   * This program demonstrates use of interrupts with assembly code. It first
4   * sets up interrupts from three devices: the Nios V machine timer, an FPGA
5   * interval timer, and the pushbutton KEY port. Next, the program makes a
6   * software interrupt occur. Finally, the program loops while responding to
7   * interrupts from the timers and the pushbutton KEY port.
8   *
9   * The interrupt service routine for the software interrupt turns on most
10  * of the red lights in the LEDR port.
11  *
12  * The interrupt service routine for the Nios V machine timer causes the
13  * main program to display a binary counter on the LEDR red lights.
14  *
15  * The interrupt service routine for the interval timer causes the main
16  * program to display a decimal counter on HEX0. The counter either
17  * increases or decreases, in the range 0 to 9. When a KEY is pressed, the
18  * direction of counting on HEX0 is reversed.
19 *****************************************************************************/
20 .equ clock_rate, 100000000
21 .equ quarter_clock, clock_rate / 4
22
23 .global _start
24 _start:     li      sp, SDRAM_END-3  # bottom of memory
25             jal     set_mtimer       # initialize machine timer
26             jal     set_itimer       # initialize interval timer
27             jal     set_KEY          # initialize the KEY port
28
29             # Set handler address, enable interrupts
30             csrci   mstatus, 0x8     # disable Nios V interrupts
31             la      t0, handler
32             csrw    mtvec, t0        # set trap address
33             csrr    t0, mie          # what ints are enabled?
34             csrc    mie, t0          # disable all ints that were enabled
35             li      t0, 0x50088      # set the enable pattern
36             csrs    mie, t0          # swi, itimer, KEY, mtimer
37             csrsi   mstatus, 0x8     # enable Nios V interrupts
38
39             # Make a software interrupt happen
40             la      t0, MTIME_BASE   # base address
41             li      t1, 1            # pattern to write to msip
42             sw      t1, 16(t0)       # write to msip (sw interrupt)
43
```

Listing 5. An example of assembly language code that uses interrupts (Part *a*).

```
44              la      s0, counter       # pointer to counter
45              la      s1, LEDR_BASE     # pointer to red lights
46              la      s2, digit         # pointer to digit
47              la      s3, HEX3_HEX0_BASE    # pointer to hex display
48              li      t0, 0x3f          # pattern for 7-segment digit 0
49              sw      t0, (s3)          # display 0 on HEX0
50
51  loop:       wfi
52              lw      t0, (s0)          # load the counter value
53              sw      t0, (s1)          # write to the lights
54              lw      a0, (s2)          # load the digit value
55              jal     seg7_code         # get 7-segment code to display
56              sw      a0, (s3)          # write code to HEX0
57              j       loop
58
59  # Trap handler
60  handler:    addi    sp, sp, -16       # save regs that will be modified
61              sw      ra, 12(sp)
62              sw      t2, 8(sp)
63              sw      t1, 4(sp)
64              sw      t0, (sp)
65
66              # check for cause of trap
67              csrr    t0, mcause        # read mcause register
68              li      t1, 0x80000003    # IRQ 3
69              bne     t0, t1, next      # software interrupt?
70              jal     SWI_ISR
71              j       trap_end
72  next:       li      t1, 0x80000007    # IRQ 7
73              bne     t0, t1, nnext     # machine timer?
74              jal     mtimer_ISR
75              j       trap_end
76  nnext:      li      t1, 0x80000010    # IRQ 16
77              bne     t0, t1, chk_KEY
78              jal     itimer_ISR
79              j       trap_end
80  chk_KEY:    li      t1, 0x80000012    # IRQ 18
81  stay:       bne     t0, t1, stay      # unexpected!
82              jal     KEY_ISR
83
84  trap_end:   lw      t0, (sp)          # restore regs
85              lw      t1, 4(sp)
86              lw      t2, 8(sp)
87              lw      ra, 12(sp)
88              addi    sp, sp, 16
89              mret
90
```

Listing 5. An example of assembly language code that uses interrupts (Part *b*).

```
91  # Handle software interrupt
92  SWI_ISR:    la      t0, counter      # pointer to counter
93              li      t1, 0b1111111100
94              sw      t1, (t0)         # write to counter
95              la      t0, MTIME_BASE   # base address
96              sw      zero, 16(t0)     # clear software interrupt in msip
97              ret
98
99  # Handle machine timer interrupt
100 mtimer_ISR: la      t0, MTIME_BASE
101             lw      t1, 8(t0)        # read mtimecmp low
102             li      t2, quarter_clock
103             add     t2, t2, t1       # add to mtimecmp
104             sw      t2, 8(t0)        # write to mtimecmp low
105             sltu    t2, t2, t1       # check for carry-out
106             lw      t1, 12(t0)       # read mtimecmp high
107             add     t1, t1, t2       # increment (t2 = carry-out)
108             sw      t1, 12(t0)       # write to mtimecmp high
109
110             la      t0, counter      # pointer to counter
111             lw      t1, (t0)         # read counter value
112             addi    t1, t1, 1        # increment the counter
113             sw      t1, (t0)         # store counter to memory
114             ret
115
116 # Handle interval timer interrupt
117 itimer_ISR: la      t0, TIMER_BASE
118             sh      zero, (t0)       # clear interrupt
119             la      t0, digit
120             lw      t1, (t0)
121
122             la      t2, KEY_dir
123             lw      t2, (t2)
124             add     t1, t1, t2
125             li      t2, 9
126             bgt     t1, t2, itimer_end
127             bltz    t1, itimer_end
128             sw      t1, (t0)         # store counter to memory
129 itimer_end: ret
130
```

Listing 5. An example of assembly language code that uses interrupts (Part *c*).

```
131 # Handle KEY port interrupt
132 KEY_ISR:    la      t0, KEY_BASE
133             lw      t1, 0xc(t0)       # read edgecapture register
134             sw      t1, 0xc(t0)       # write to edgecapture
135             la      t0, KEY_dir
136             lw      t1, (t0)          # get current direction
137             neg     t1, t1            # reverse
138             sw      t1, (t0)          # set current direction
139             ret
140
141 # Initialize Nios V machine timer
142 set_mtimer: la      t0, MTIME_BASE    # set address
143             # read the current time
144 tloop:      lw      t2, 4(t0)         # read mtime high
145             lw      t1, 0(t0)         # read mtime low
146             lw      t3, 4(t0)         # read high again
147             bne     t3, t2, tloop     # check for overflow from low to high
148             # current time is t2:t1
149             li      t3, quarter_clock
150             add     t3, t3, t1        # add to current time
151             sw      t3, 8(t0)         # write to mtimecmp low
152             sltu    t3, t3, t1        # check for carry-out
153             add     t2, t2, t3        # increment (t3 = carry-out)
154             sw      t2, 12(t0)        # write to mtimecmp high
155             ret
156
157 # Initialize FPGA interval timer
158 set_itimer: la      t0, TIMER_BASE    # set address
159             sh      zero, 4(t0)       # stop the timer
160             sh      zero, (t0)        # clear the interrupt bit
161             li      t1, clock_rate    # timeout value
162             sh      t1, 8(t0)         # write to timer low half-word
163             srli    t1, t1, 16
164             sh      t1, 0xc(t0)       # write to timer high half-word
165             li      t1, 0b0111        # START = 1, CONT = 1, ITO = 1
166             sh      t1, 4(t0)         # reset lower word of mtime
167             ret
168
169 # Enable interrupts in the KEY port
170 set_KEY:    la      t0, KEY_BASE      # set address
171             li      t1, 0xf
172             sw      t1, 0xc(t0)       # clear all EdgeCapture bits
173             li      t1, 0xf           # bit pattern for all four KEYs
174             sw      t1, 8(t0)         # write to interrupt mask register
175             ret
176
```

Listing 5. An example of assembly language code that uses interrupts (Part *d*).

```
177 # Convert digit in a0 to seven-segment code. Return code in a0
178 seg7_code:  la      t0, bit_codes    # starting address of the bit codes
179             add     t0, t0, a0       # index into the bit codes
180             lb      a0, (t0)         # read the bit code for our digit
181             ret
182
183 counter:    .word   0                # binary counter to be displayed
184 digit:      .word   0                # decimal digit to be displayed
185 KEY_dir:    .word   1                # digit counter direction
186 # 7-segment codes for digits 0, 1, ..., 9
187 bit_codes:  .byte   0x3f, 0x06, 0x5b, 0x4f, 0x66
188             .byte   0x6d, 0x7d, 0x07, 0x7f, 0x67
```

Listing 5. An example of assembly language code that uses interrupts (Part *e*).

```
1  #include    "address_map_niosv.h"
2
3  #define clock_rate 100000000
4  #define quarter_clock clock_rate / 4
5
6  static void handler(void) __attribute__ ((interrupt ("machine")));
7  void set_mtimer(void);
8  void set_itimer(void);
9  void set_KEY(void);
10 void SWI_ISR(void);
11 void mtimer_ISR(void);
12 void itimer_ISR(void);
13 void KEY_ISR(void);
14
15 /* Global variables are written by interrupt service routines; we declare
16  * as volatile to avoid the compiler caching their values in registers */
17 volatile int counter = 0;     // binary counter to be displayed
18 volatile int digit = 0;       // decimal digit to be displayed
19 volatile int KEY_dir = 1;     // digit counter direction
20 // 7-segment codes for digits 0, 1, ..., 9
21 char bit_codes[] = {0x3f, 0x06, 0x5b, 0x4f, 0x66,
22     0x6d, 0x7d, 0x07, 0x7f, 0x67};
23
24 /*****************************************************************************
25  * This program demonstrates use of interrupts with assembly code. It first
26  * sets up interrupts from three devices: the Nios V machine timer, an FPGA
27  * interval timer, and the pushbutton KEY port. Next, the program makes a
28  * software interrupt occur. Finally, the program loops while responding to
29  * interrupts from the timers and the pushbutton KEY port.
30  *
31  * The interrupt service routine for the software interrupt turns on most
32  * of the red lights in the LEDR port.
33  *
34  * The interrupt service routine for the Nios V machine timer causes the
35  * main program to display a binary counter on the LEDR red lights.
36  *
37  * The interrupt service routine for the interval timer causes the main
38  * program to display a decimal counter on HEX0. The counter either
39  * increases or decreases, in the range 0 to 9. When a KEY is pressed, the
40  * direction of counting on HEX0 is reversed.
41 *****************************************************************************/
42 int main(void) {
43     /* Declare volatile pointers to I/O registers (volatile means that the
44      * accesses will always go to the memory (I/O) address */
45     volatile int *mtime_ptr = (int *) MTIME_BASE;
46     volatile int *LEDR_ptr = (int *) LEDR_BASE;
47     volatile int *HEX3_HEX0_ptr = (int *) HEX3_HEX0_BASE;
48
```

Listing 6. An example of C code that uses interrupts (Part *a*).

```
49      set_mtimer();
50      set_itimer();
51      set_KEY();
52
53      int mstatus_value, mtvec_value, mie_value;
54      mstatus_value = 0b1000;    // interrupt bit mask
55      // disable interrupts
56      __asm__ volatile ("csrc mstatus, %0" :: "r"(mstatus_value));
57      mtvec_value  = (int) &handler; // set trap address
58      __asm__ volatile ("csrw mtvec, %0" :: "r"(mtvec_value));
59      // disable all interrupts that are currently enabled
60      __asm__ volatile ("csrr %0, mie" : "=r"(mie_value));
61      __asm__ volatile ("csrc mie, %0" :: "r"(mie_value));
62      mie_value = 0x50088; // KEY, itimer, mtimer, SW interrupts
63      // set interrupt enables
64      __asm__ volatile ("csrs mie, %0" :: "r"(mie_value));
65      // enable Nios V interrupts
66      __asm__ volatile ("csrs mstatus, %0" :: "r"(mstatus_value));
67
68      *(mtime_ptr + 4) = 1;    // cause a software interrupt
69
70      *HEX3_HEX0_ptr = 0x3f;  // show 0 on HEX0
71
72      while (1) {
73          *LEDR_ptr = counter;
74          *HEX3_HEX0_ptr = bit_codes[digit]; // display in decimal
75      }
76  }
77
78  /*****************************************************************
79   * Trap handler: determine what caused the interrupt and calls the
80   *  appropriate subroutine.
81   *****************************************************************/
82  void handler (void){
83      int mcause_value;
84      __asm__ volatile ("csrr %0, mcause" : "=r"(mcause_value));
85      if (mcause_value == 0x80000003) // software interrupt
86          SWI_ISR();
87      else if (mcause_value == 0x80000007) // machine timer
88          mtimer_ISR();
89      else if (mcause_value == 0x80000010) // interval timer
90          itimer_ISR();
91      else if (mcause_value == 0x80000012) // KEY port
92          KEY_ISR();
93      // else, ignore the trap
94  }
95
```

Listing 6. An example of C code that uses interrupts (Part *b*).

```
96  // Software interrupt service routine
97  void SWI_ISR(void){
98      volatile int *mtime_ptr = (int *) MTIME_BASE;
99      counter = 0b1111111100; // set global variable
100     *(mtime_ptr + 4) = 0;   // clear interrupt
101 }
102
103 // Nios V machine timer interrupt service routine
104 typedef long long int64;
105
106 void mtimer_ISR(void){
107     volatile unsigned int *mtime_ptr = (unsigned int *) MTIME_BASE;
108     int64 mtimecmp64;
109
110     mtimecmp64 = *(mtime_ptr + 3);  // read high word of 64-bit
            register
111     mtimecmp64 = (mtimecmp64 << 32) | *(mtime_ptr + 2);      // read
            low word
112     mtimecmp64 = mtimecmp64 + (int64) quarter_clock;         // adjust
            timeout
113     *(mtime_ptr + 2)  = (unsigned int) mtimecmp64;           // store
            low word
114     *(mtime_ptr + 3)  = (unsigned int) (mtimecmp64 >> 32);   // store
            high word
115     counter = counter + 1;
116 }
117
118 // FPGA interval timer interrupt service routine
119 void itimer_ISR(void){
120     int new_digit;
121     volatile int * timer_ptr = (int *) TIMER_BASE;
122     *timer_ptr = 0; // clear the interrupt
123     new_digit = digit + KEY_dir;    // inc/dec the digit
124     if (new_digit < 10 && new_digit > -1)
125         digit = new_digit;  // decimal (0 to 9)
126 }
127
128 // KEY port interrupt service routine
129 void KEY_ISR(void){
130     int pressed;
131     volatile int *KEY_ptr = (int *) KEY_BASE;
132     pressed = *(KEY_ptr + 3);  // read EdgeCapture
133     *(KEY_ptr + 3) = pressed;  // clear EdgeCapture register
134     KEY_dir = -KEY_dir;        // reverse counting direction
135 }
136
```

Listing 6. An example of C code that uses interrupts (Part *c*).

```
137 // Configure the Nios V machine timer
138 void set_mtimer(void){
139     volatile int *mtime_ptr = (int *) MTIME_BASE;
140     unsigned int mtime_h, mtime_l, carry, mtimecmp_l;
141     do {
142         mtime_h = *(mtime_ptr + 1);          // read mtime high word
143         mtime_l = *(mtime_ptr);              // read mtime low word
144     } while (*(mtime_ptr + 1) != mtime_h);
145     mtimecmp_l = mtime_l + quarter_clock;    // add to current time
146     carry = mtimecmp_l < mtime_l ? 1 : 0;    // check for carry-out
147     *(mtime_ptr + 2) = mtimecmp_l;           // set mtimecmp low word
148     *(mtime_ptr + 3) = mtime_h + carry;      // set mtimecmp high word
149 }
150
151 // Configure the FPGA interval timer
152 void set_itimer(void){
153     volatile int *timer_ptr = (int *) TIMER_BASE;
154     // set the interval timer period
155     int load_val = clock_rate;
156     *(timer_ptr + 0x2) = (load_val & 0xFFFF);
157     *(timer_ptr + 0x3) = (load_val >> 16) & 0xFFFF;
158
159     // start interval timer, enable its interrupts
160     *(timer_ptr + 1) = 0x7; // STOP = 1, START = 1, CONT = 1, ITO = 1
161 }
162
163 // Configure the KEY port
164 void set_KEY(void){
165     volatile int *KEY_ptr = (int *) KEY_BASE;
166     *(KEY_ptr + 3) = 0xF; // clear EdgeCapture register
167     *(KEY_ptr + 2) = 0xF; // enable interrupts for all KEYs
168 }
```

Listing 6. An example of C code that uses interrupts (Part *d*).

8.4 Audio

```
#include "address_map_niosv.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 7. An example of code that uses the audio port.

8.5 Video Out

```
#include "address_map_niosv.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;

    // The following two lines are supported in hardware, but not in CPUlator
    volatile int * rgb_status = (int *)(RGB_RESAMPLER_BASE);
    int db = get_data_bits(*rgb_status & 0x3F);
    // int db = 16;  // replace above two lines with this one for CPUlator

    /* 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;
    }
    return -1;  // error
}
```

Listing 8. An example of code that uses the video-out port.

8.6 PS/2

```
#include "address_map_niosv.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 9. An example of code that uses the PS/2 port.

8.7 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 10. An example of code that uses floating-point variables.

8.8 Include Files

```
/*****************************************************************************
 * This file provides address values that exist in the DE1-SoC Computer
 *****************************************************************************/

/* Memory */
    .equ    DDR_BASE,               0x40000000
    .equ    DDR_END,                0x7FFFFFFF
    .equ    A9_ONCHIP_BASE,         0xFFFF0000
    .equ    A9_ONCHIP_END,          0xFFFFFFFF
    .equ    SDRAM_BASE,             0x00000000
    .equ    SDRAM_END,              0x03FFFFFF
    .equ    FPGA_PIXEL_BUF_BASE,    0x08000000
    .equ    FPGA_PIXEL_BUF_END,     0x0803FFFF
    .equ    FPGA_CHAR_BASE,         0x09000000
    .equ    FPGA_CHAR_END,          0x09001FFF

/* Cyclone V FPGA devices */
    .equ    LED_BASE,               0xFF200000
    .equ    LEDR_BASE,              0xFF200000
    .equ    HEX3_HEX0_BASE,         0xFF200020
    .equ    HEX5_HEX4_BASE,         0xFF200030
    .equ    SW_BASE,                0xFF200040
    .equ    KEY_BASE,               0xFF200050
    .equ    JP1_BASE,               0xFF200060
    .equ    JP2_BASE,               0xFF200070
    .equ    PS2_BASE,               0xFF200100
    .equ    PS2_DUAL_BASE,          0xFF200108
    .equ    JTAG_UART_BASE,         0xFF201000
    .equ    IrDA_BASE,              0xFF201020
    .equ    TIMER_BASE,             0xFF202000
    .equ    TIMER_2_BASE,           0xFF202020
    .equ    AV_CONFIG_BASE,         0xFF203000
    .equ    PIXEL_BUF_CTRL_BASE,    0xFF203020
    .equ    CHAR_BUF_CTRL_BASE,     0xFF203030
    .equ    AUDIO_BASE,             0xFF203040
    .equ    VIDEO_IN_BASE,          0xFF203060
    .equ    EDGE_DETECT_CTRL_BASE,  0xFF203070
    .equ    ADC_BASE,               0xFF204000

/* Nios V memory-mapped registers */
    .equ    MTIME_BASE,             0xFF202100
```

Listing 11. The *address_map_niosv.s* include file.

```
/*****************************************************************************
 * This file provides address values that exist in the DE1-SoC Computer
 *****************************************************************************/

#ifndef __SYSTEM_INFO__
#define __SYSTEM_INFO__

#define BOARD             "DE1-SoC"

/* Memory */
#define DDR_BASE                0x40000000
#define DDR_END                 0x7FFFFFFF
#define SDRAM_BASE              0x00000000
#define SDRAM_END               0x03FFFFFF
#define FPGA_PIXEL_BUF_BASE     0x08000000
#define FPGA_PIXEL_BUF_END      0x0803FFFF
#define FPGA_CHAR_BASE          0x09000000
#define FPGA_CHAR_END           0x09001FFF

/* Cyclone V FPGA devices */
#define LED_BASE                0xFF200000
#define LEDR_BASE               0xFF200000
#define HEX3_HEX0_BASE          0xFF200020
#define HEX5_HEX4_BASE          0xFF200030
#define SW_BASE                 0xFF200040
#define KEY_BASE                0xFF200050
#define JP1_BASE                0xFF200060
#define JP2_BASE                0xFF200070
#define PS2_BASE                0xFF200100
#define PS2_DUAL_BASE           0xFF200108
#define JTAG_UART_BASE          0xFF201000
#define IrDA_BASE               0xFF201020
#define TIMER_BASE              0xFF202000
#define TIMER_2_BASE            0xFF202020
#define AV_CONFIG_BASE          0xFF203000
#define RGB_RESAMPLER_BASE      0xFF203010
#define PIXEL_BUF_CTRL_BASE     0xFF203020
#define CHAR_BUF_CTRL_BASE      0xFF203030
#define AUDIO_BASE              0xFF203040
#define VIDEO_IN_BASE           0xFF203060
#define EDGE_DETECT_CTRL_BASE   0xFF203070
#define ADC_BASE                0xFF204000

/* Cyclone V HPS devices */
#define MTIME_BASE              0xFF202100

#endif
```

Listing 12. The *address_map_niosv.h* include file.