DE10-Standard Computer System with ARM* Cortex* A9

For Quartus® Prime 19.1

1 Introduction

This document describes a computer system that can be implemented on the Intel® DE10-Standard development and education board. This system, called the *DE10-Standard Computer*, is intended for use in experiments on computer organization and embedded systems.

2 DE10-Standard Computer Contents

A block diagram of the DE10-Standard Computer system is shown in Figure 1. As indicated in the figure, the components in this system are implemented utilizing the *Hard Processor System* (HPS) and FPGA inside the Cyclone® V SoC chip. The HPS comprises an ARM* Cortex* A9 dual-core processor, a DDR3 memory port, and a set of peripheral devices. The FPGA implements two Intel Nios® II 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. Instructions for using the Nios II processors are provided in a separate document, called *DE10-Standard Computer System with Nios II*.

2.1 Hard Processor System

The hard processor system (HPS), as shown in Figure 1, includes an ARM Cortex A9 dual-core processor. The A9 dual-core processor features two 32-bit CPUs and associated subsystems that are implemented as hardware circuits in the Intel Cyclone V SoC chip. An overview of the ARM A9 processor can be found in the document *Introduction to the ARM Processor*, which is provided in Intel's FPGA University Program web site. All of the I/O peripherals in the DE10-Standard Computer are accessible by the processor as memory mapped devices, using the address ranges that are given in this document. A summary of the address map can be found in Section 7.

A good way to begin working with the DE10-Standard Computer and the ARM A9 processor is to make use of a utility called the *Intel FPGA Monitor Program*. It provides an easy way to assemble/compile ARM A9 programs written in either assembly language or the C language. The Monitor Program, which can be downloaded from Intel's web site, is an application program that runs on the host computer connected to the DE10-Standard board. The Monitor Program can be used to control the execution of code on the ARM A9, list (and edit) the contents of processor registers, display/edit the contents of memory on the DE10-Standard board, and similar operations. The Monitor Program includes the DE10-Standard Computer as a pre-designed system that can be downloaded onto the DE10-Standard board, as well as several sample programs in assembly language and C that show how to use the DE10-Standard Computer's peripherals. Section 8 describes how the DE10-Standard Computer is integrated with the Monitor Program. An overview of the Monitor Program is available in the document *Intel FPGA Monitor Program Tutorial*, which is provided in the University Program web site.

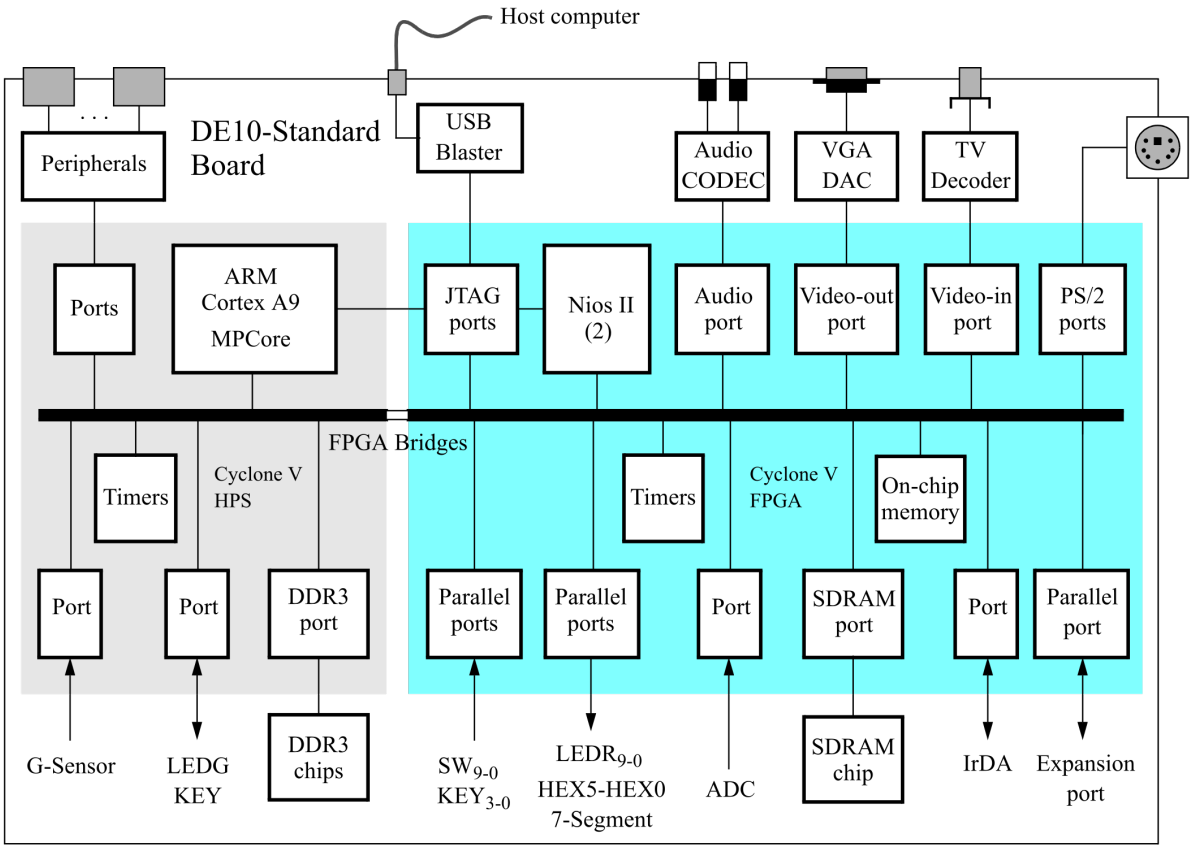


Figure 1. Block diagram of the DE10-Standard Computer.

2.2 Memory

The HPS includes a memory port that connects the ARM MPCORE* to a 1 GB DDR3 memory. This memory is normally used as the storage location of programs and data used by the ARM processors. The memory is organized as 256M x 32-bits, and is accessible using word accesses (32 bits), halfwords, and bytes. The DDR3 memory is mapped to the address space 0x00000000 to 0x3FFFFFFF. There is also a 64 KB on-chip memory available inside each ARM A9 processor. This small memory is organized as 16K x 32-bits, and is mapped to the address space 0xFFFF0000 to 0xFFFFFFFF.

2.3 Pushbutton KEY and LED Port

The HPS includes a general purpose I/O port, called *GPIO1*, that is accessible by the ARM A9 processor. As illustrated in Figure 2, this parallel port is assigned the *Base* address 0xFF709000, and includes several 32-bit registers. These registers can be read or written using word accesses. Only two bit locations in GPIO1 are used for the DE10-Standard Computer. Bit 24 of the data register (DR) is connected to a green light, LEDG, and bit 25 is connected to a pushbutton switch, KEY. To use these devices, the *data direction register* (DDR) shown in the figure has to be configured such that bit 24 is an output and bit 25 is an input. Writing a 1 into a corresponding bit position

in the DDR sets this bit as an output, while writing a 0 sets the bit as an input. After the direction bits have been set, the green light LEDG can be turned on/off by writing to bit 24 in the data register. The value of the pushbutton switch KEY can be obtained by reading the external port register and checking the value of bit 25. An example program for the ARM A9 processor that uses GPIO1 is given in Section 2.4.

As indicated in Figure 2, the GPIO1 port includes several other registers in addition to the DR and DDR registers. These other registers are mostly used for setting characteristics of input pins, which affects only the KEY input in our system. Detailed information about these registers can be found in the *Intel Cyclone V Hard Processor System* documentation, which is available on Intel's website.

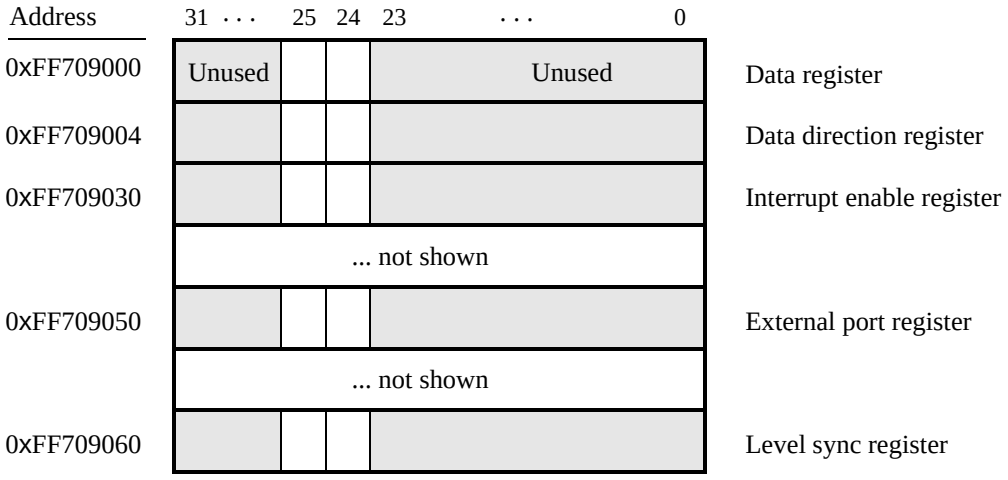


Figure 2. Parallel port GPIO1.

2.4 Timer Modules

The HPS includes several hardware timer modules that can be used to keep track of time intervals. The ARM A9 MPCore includes one *private* timer module for each A9 core, and the HPS provides four other timer modules that can be used by either A9 core. The timers are described in more detail below.

2.4.1 ARM* A9* MPCore* Timers

Figure 3 shows the registers in the programmer's interface for each A9 core private timer. These registers have the base address 0xFFFEC600, as shown in the figure, and can be read or written using word accesses. To use the timer, it is necessary to first write an initial count value into the *Load* register. The timer can then be started by setting the enable bit *E* in the *Control* register to 1, and it can be stopped by setting *E* back to 0. Once enabled the timer decrements its count value until reaching 0. When it reaches 0, the timer sets the *F* bit in the *Interrupt status* register. The *F* bit can be checked by software using polled-I/O to determine when the timer period has expired. The *F* bit can be reset to 0 by writing a 1 into it. Also, if bit *I* in the *Control* register is set to 1, then a processor interrupt can be generated when the timer reaches 0. Using interrupts with the timer is discussed in Section 3.

When it reaches 0, the timer will stop if the auto bit (*A*) in the control register is set to 0. But if bit *A* is set to 1, then the timer will automatically reload the value in the *Load* register and continue decrementing. The current value of the timer is available to software in the *Counter* register shown in Figure 3. The timer uses a clock frequency of 200 MHz. The *Prescaler* field in the *Control* register can be used to slow down the counting rate, as follows. The timer decrements each *Prescaler* +1 clock cycle. Therefore, if *Prescaler* = 0, then the timer decrements every clock cycle, if *Prescaler* = 1, the timer decrements every second clock cycle, and so on.

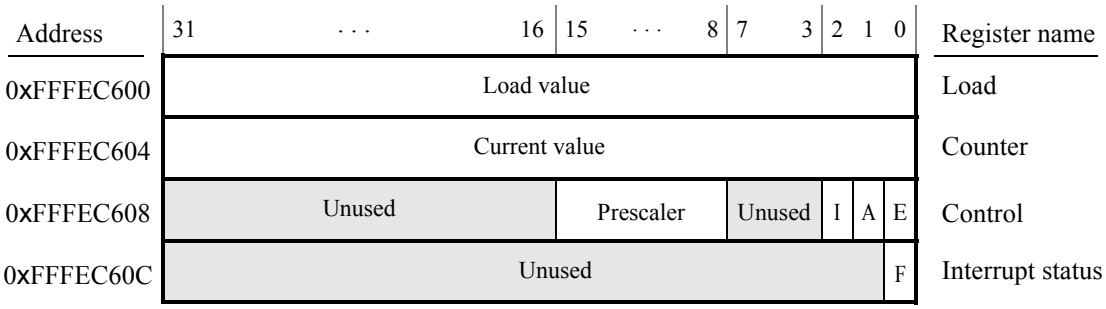


Figure 3. ARM A9 private timer port.

2.4.2 HPS Timers

Figure 4 shows the registers in the programmer's interface for one of the HPS timers. These registers have the base address 0xFFC08000, as shown in the figure, and can be read or written using word accesses. To configure the timer, it is necessary to ensure that it is stopped by setting the enable bit *E* in the *Control* register to 0. A starting count value for the timer can then be written into the *Load* register. To instruct the timer to use the specified starting count value, the *M* in the *Control* register should be set to 1, and the timer can be started by setting *E* = 1. The timer counts down to 0, and then sets both bit *F* in the *End-of-interrupt* register and bit *S* in the *Interrupt status* register to 1. Software can poll the value of *S* to determine when the timer period has expired. The *S* bit, and the *F* bit can be reset to 0 by reading the contents of the *End-of-Interrupt* register. Also, if bit *I*, the interrupt mask bit, in the *Control* register is set to 0, then an interrupt can be generated when the timer reaches 0 (note that bit *I* in the ARM A9 private timer shown in Figure 3 has the opposite polarity). The use of interrupts with the timer is discussed in Section 3.

The current value of the timer is available to software in the *Counter* register shown in Figure 4. The timer uses a clock frequency of 100 MHz.

There are three other identical timers in the HPS, with the following base addresses: 0xFFC09000, 0xFFD00000, and 0xFFD01000. The first of these timers uses a 100 MHz clock, and the last two timers use a 25 MHz clock.

We should mention that other timer modules also exist in the HPS. The ARM A9 MPCore has a *global* timer that is shared by both A9 cores, as well as a *watchdog* timer for each processor. Also, the HPS has two additional watchdog timers. Documentation about the global timer and watchdog timers is available in the *ARM Cortex A9 MPCore Technical Reference Manual*, and in the *Intel Cyclone V Hard Processor System Technical Reference Manual*.

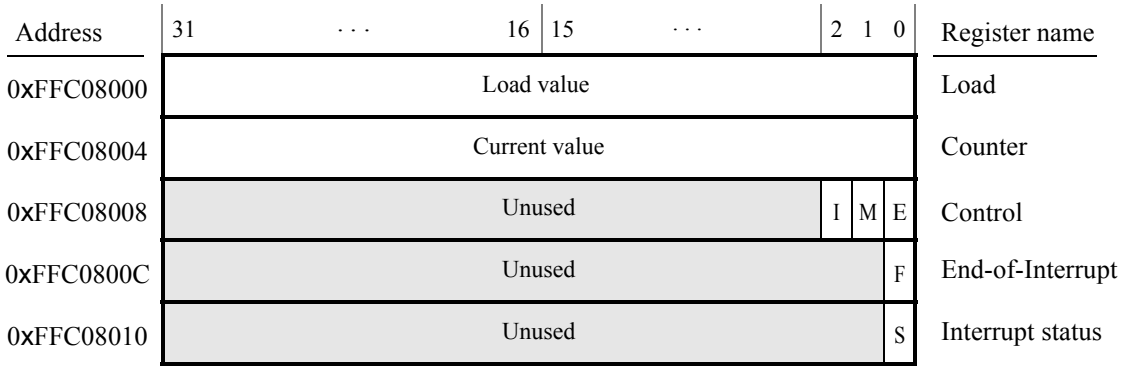


Figure 4. HPS timer port.

2.4.3 Using a Timer with Assembly Language Code

An example of ARM A9 assembly language code is included in the Appendix in Listing 1. The code configures the private timer for the A9 core so that it produces one-second timeouts. An infinite loop is used to flash the green light connected to GPIO1, discussed in Section 2.3. The light is turned on for one second, then off, and so on.

An example of C code is also included in Listing 2. This code performs the same actions as the assembly language program in Listing 1—it flashes on/off the green light connected to GPIO1 at one-second intervals.

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

2.5 FPGA Bridges

The *FPGA bridges* depicted in Figure 1 provide connections between the HPS and FPGA in the Cyclone V SoC device. The bridges are enabled, or disabled, by using the *Bridge reset* register, which is illustrated in Figure 5 and has the address 0xFFD0501C. Three distinct bridges exist, called *HPS-to-FPGA*, *lightweight HPS-to-FPGA*, and *FPGA-to-HPS*. In the DE10-Standard Computer the first two of these bridges are used to connect the ARM A9 processor to the FPGA. As indicated in Figure 5 the bridges are enabled/disabled by bits $0-2$ of the *Bridge reset* register. To use the memory-mapped peripherals in the FPGA, software running on the ARM A9 must enable the HPS-to-FPGA and lightweight HPS-to-FPGA bridges by setting bits #0 and #1 of the *Bridge reset* register to 0. We should note that if a user program is downloaded and run on the ARM A9 by using the Intel FPGA Monitor Program, described in Section 8, then these bridges are automatically enabled before the user program is started.

In addition to the components described above, the HPS also provides a number of other peripheral devices, such as USB, Ethernet, and a 3-D accelerometer (G-sensor). The G-sensor is described in the tutorial *Using the DE10-Standard Accelerometer with ARM*, available from Intel's FPGA University Program website. Documentation about the other devices connected to the HPS can be found in the *Intel Cyclone V Hard Processor System Technical Reference Manual*, as well as in the *DE10-Standard Board User Manual*.

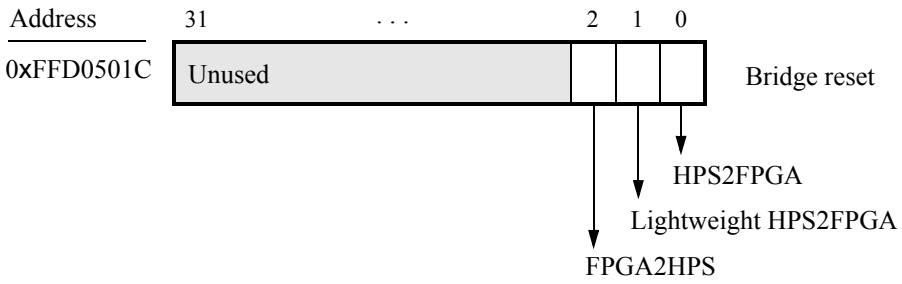


Figure 5. FPGA bridge reset register.

2.6 G-Sensor

The DE10-Standard Computer includes a 3D accelerometer (G-sensor) that is connected to the HPS. The accelerometer can be configured to produce two interrupt signals. One of these signals is used in our system and is connected to an I/O port on the HPS called GPIO2. This I/O port has the same structure as GPIO1 which was described in Section 2.3. The base address of GPIO2 is 0xFF70A000. The accelerometer interrupt signal is connected to bit 5 in the data register of GPIO2. The accelerometer generates an interrupt by driving the interrupt signal high. The "interrupt" signal is not connected to the generic interrupt controller (GIC) of the HPS, so to check for interrupts the bit must be polled.

2.7 FPGA Components

As shown in Figure 1 a number of components in the DE10-Standard Computer are implemented inside the FPGA in the Cyclone® V SoC chip. Several of these components are described in this section, and the others are presented in Section 4.

2.8 Nios® II Processor

The Intel Nios II processor is a 32-bit CPU that can be implemented in an Intel FPGA device. Two versions of the Nios II processor are available, designated economy (/e) and fast (/f). The DE10-Standard Computer includes two instances of the Nios II/f version, configured with floating-point hardware support. Instructions for using the Nios II processors in the DE10-Standard Computer are provided in a separate document, called *DE10-Standard Computer with Nios II*.

2.9 Memory Components

The DE10-Standard Computer has an SDRAM port, as well as two memory modules implemented using the on-chip memory inside the FPGA. These memories are described below.

2.9.1 SDRAM

An SDRAM Controller in the FPGA provides an interface to the 64 MB synchronous dynamic RAM (SDRAM) on the DE10-Standard board, which is organized as 32M x 16 bits. It is accessible by the ARM A9 processor using word (32-bit), halfword (16-bit), or byte operations, and is mapped to the address space 0xC0000000 to 0xC3FFFFFF.

2.9.2 On-Chip Memory

The DE10-Standard Computer includes a 256 KB memory that is implemented inside the FPGA. This memory is organized as 64K x 32 bits, and spans addresses in the range 0xC8000000 to 0xC803FFFF. The memory is used as a pixel buffer for the video-out and video-in ports.

2.9.3 On-Chip Memory Character Buffer

The DE10-Standard Computer includes an 8 KB memory implemented inside the FPGA that is used 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 address range 0xC9000000 to 0xC9001FFF.

2.10 Parallel Ports

There are several parallel ports implemented in the FPGA that support input, output, and bidirectional transfers of data between the ARM A9 processor and I/O peripherals. As illustrated in Figure 6, 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.

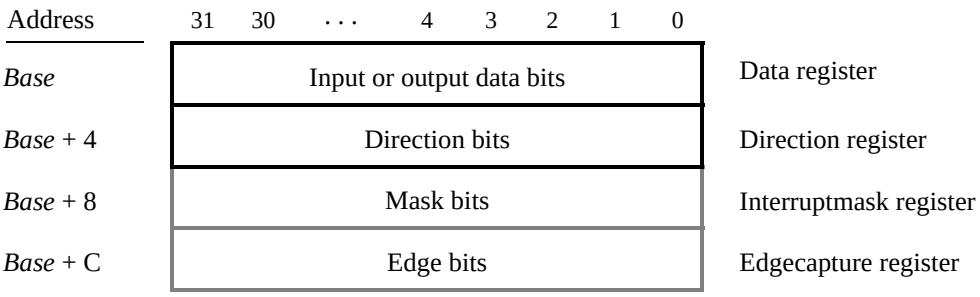


Figure 6. Parallel port registers in the DE10-Standard Computer.

Some of the parallel ports in the DE10-Standard Computer have registers at addresses *Base* + 8 and *Base* + C, as indicated in Figure 6. These registers are discussed in Section 3.

2.10.1 Red LED Parallel Port

The red lights $LEDR_{9-0}$ on the DE10-Standard board are driven by an output parallel port, as illustrated in Figure 7. 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.

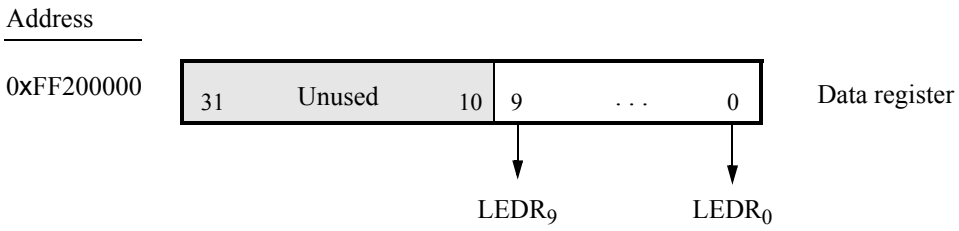


Figure 7. Output parallel port for *LEDR*.

2.10.2 7-Segment Displays Parallel Port

There are two parallel ports connected to the 7-segment displays on the DE10-Standard board, each of which comprises a 32-bit write-only *Data* register. As indicated in Figure 8, 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 8. The locations of segments 6 to 0 in each seven-segment display on the DE10-Standard board is illustrated on the right side of the figure.

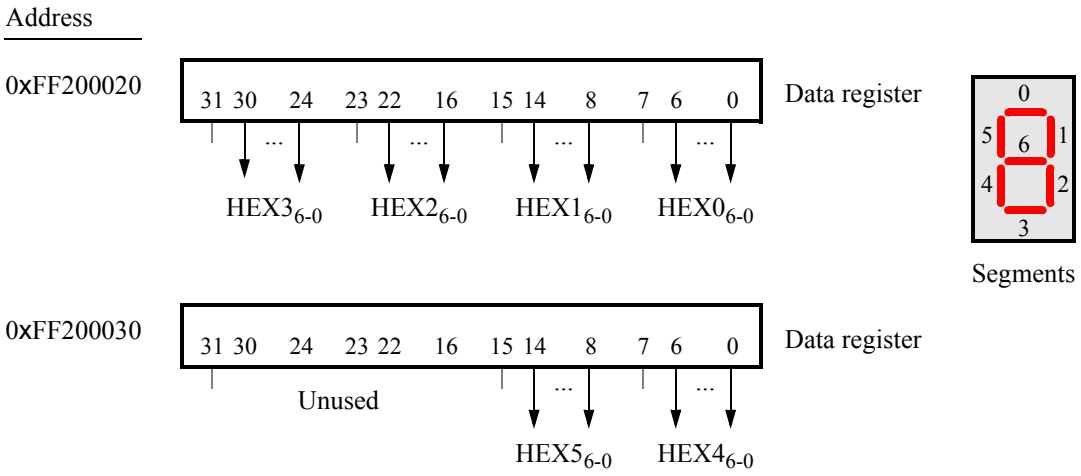


Figure 8. Bit locations for the 7-segment displays parallel ports.

2.10.3 Slider Switch Parallel Port

The SW_{9-0} slider switches on the DE10-Standard board are connected to an input parallel port. As illustrated in Figure 9, this port comprises a 10-bit read-only *Data* register, which is mapped to address 0xFF200040.

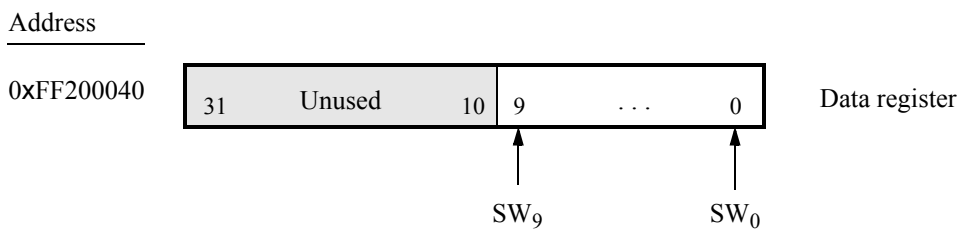


Figure 9. *Data* register in the slider switch parallel port.

2.10.4 Pushbutton Key Parallel Port

The parallel port connected to the KEY_{3-0} pushbutton switches on the DE10-Standard board comprises three 4-bit registers, as shown in Figure 10. 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 10, at addresses 0xFF200058 and 0xFF20005C, are discussed in Section 3.

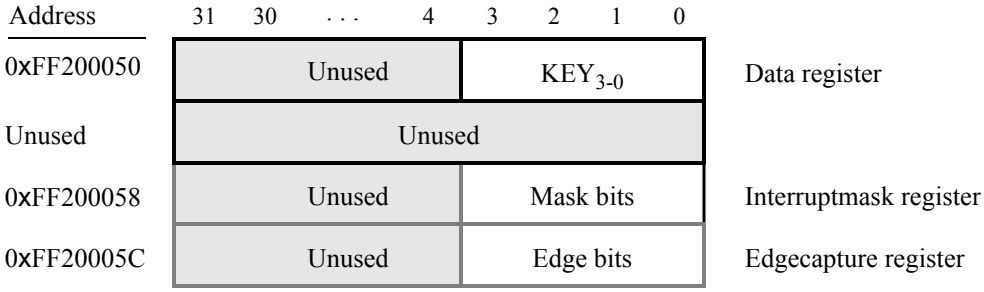


Figure 10. Registers used in the pushbutton parallel port.

2.10.5 Expansion Parallel Port

The DE10-Standard Computer includes one bidirectional parallel port that is connected to the *JP1* expansion header on the DE10-Standard board. This parallel port includes the four 32-bit registers that were described previously for Figure 6. The base address of this port is 0xFF200060. Figure 11 gives a diagram of the *JP1* expansion connector on the DE10-Standard 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 *JP1* 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.

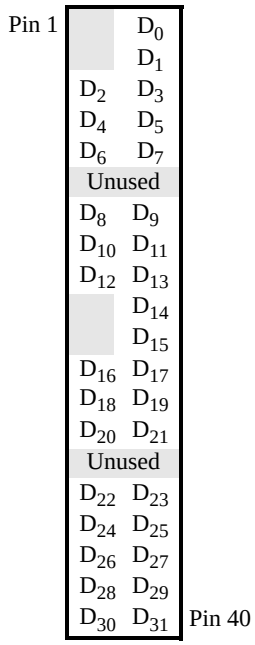


Figure 11. Assignment of parallel port bits to pins.

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

The DE10-Standard Computer provides a convenient platform for experimenting with ARM A9 assembly language code, or C code. A simple example of such code is provided in the Appendix in Listings 3 and 4. Both programs perform the same operations, and illustrate the use of parallel ports by using either assembly language or C code.

The code in the figures displays the values of the SW switches on the LED lights. A rotating pattern is displayed on the LEDs. This pattern is rotated to the left by using an ARM A9 *rotate* instruction, and a delay loop is used to make the shifting slow enough to observe. The pattern can be changed to the values of the SW switches by pressing a pushbutton KEY. When a pushbutton key is pressed, the program waits in a loop until the key is released.

The source code files shown in Listings 3 and 4 are distributed as part of the Intel FPGA Monitor Program. The files can be found under the heading *sample programs*, and are identified by the name *Getting Started*.

2.11 JTAG* Port

The JTAG* port implements a communication link between the DE10-Standard board and its host computer. This link can be used by the Intel Quartus® Prime software to transfer FPGA programming files into the DE10-Standard board, and by the Intel FPGA Monitor Program, discussed in Section 8. The JTAG port also includes a UART, which can be used to transfer character data between the host computer and programs that are executing on the ARM A9 processor. If the Intel FPGA Monitor Program is used on the host computer, then this character data is sent and received through its *Terminal Window*. The programming interface of the JTAG UART consists of two 32-bit registers, as

shown in Figure 12. The register mapped to address 0xFF201000 is called the *Data* register and the register mapped to address 0xFF201004 is called the *Control* register.

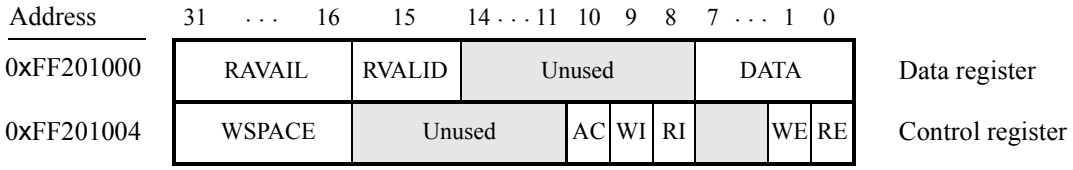


Figure 12. JTAG UART registers.

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 12. 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.

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

Listings 5 and 6 give simple examples of assembly language and C code, respectively, that use the JTAG UART. Both versions of the code perform the same function, which is to first send an ASCII string to the JTAG UART, and then enter an endless loop. In the loop, the code reads character data that has been received by the JTAG UART, and echoes this data back to the UART for transmission. If the program is executed by using the Intel FPGA Monitor Program, then any keyboard character that is typed into the *Terminal Window* of the Monitor Program will be echoed back, causing the character to appear in the *Terminal Window*.

The source code files shown in Listings 5 and 6 are made available as part of the Intel FPGA Monitor Program. The files can be found under the heading *sample programs*, and are identified by the name *JTAG UART*.

2.11.2 Second JTAG* UART

The DE10-Standard Computer includes a second JTAG UART that is accessible by the ARM A9 MPCORE. This second UART is mapped to the base address 0xFF201008, and operates as described above. The reason that two JTAG UARTs are provided is to allow each processor in the ARM A9 MPCORE to have access to a separate UART.

2.12 Interval Timers

The DE10-Standard Computer includes a timer module implemented in the FPGA that can be used by the ARM A9 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 13. 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.

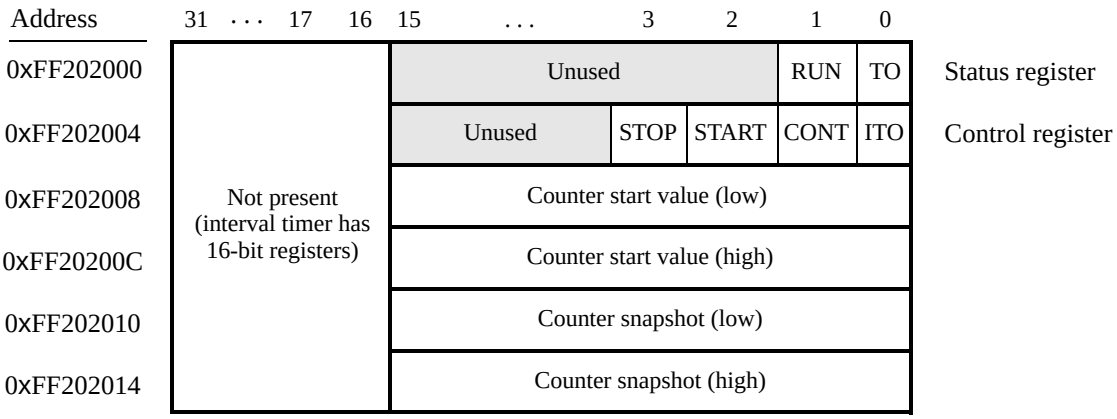


Figure 13. Interval timer registers.

- *CONT* affects the continuous operation of the timer. When the timer reaches a count value of zero it automatically reloads the specified starting count value. If *CONT* is set to 1, then the timer will continue counting down automatically. But if *CONT* = 0, then the timer will stop after it has reached a count value of 0.
- (*START*/*STOP*) is used to commence/suspend the operation of the timer by writing a 1 into the respective bit.

The two 16-bit registers at addresses 0xFF202008 and 0xFF20200C allow the period of the timer to be changed by setting the starting count value. The default setting provided in the DE10-Standard Computer 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.

2.13 System ID

The system ID module provides a unique value that identifies the DE10-Standard Computer system. The host computer connected to the DE10-Standard board can query the system ID module by performing a read operation through the JTAG port. The host computer can then check the value of the returned identifier to confirm that the DE10-Standard Computer has been properly downloaded onto the DE10-Standard board. This process allows debugging tools on the host computer, such as the Intel FPGA Monitor Program, to verify that the DE10-Standard board contains the required computer system before attempting to execute code that has been compiled for this system.

3 Exceptions and Interrupts

The A9 processor supports eight types of exceptions, including the *reset* exception and the *interrupt request* (IRQ) exception, as well a number of exceptions related to error conditions. All of the exception types are described in the document *Introduction to the ARM Processor*, which is provided in Intel's FPGA University Program web site. Exception processing uses a table in memory, called the *vector table*. This table comprises eight words in memory and has one entry for each type of exception. The contents of the vector table have to be set up by software, which typically places a branch instruction in each word of the table, where the branch target is the desired exception service routine. When an exception occurs, the A9 processor stops the execution of the program that is currently running, and then fetches the instruction stored at the corresponding vector table entry. The vector table usually starts at the address 0x00000000 in memory. The first entry in the table corresponds to the reset vector, and the IRQ vector uses the seventh entry in the table, at the address 0x00000018.

The IRQ exception allows I/O peripherals to generate interrupts for the A9 processor. All interrupt signals from the peripherals are connected to a module in the processor called the *generic interrupt controller* (GIC). The GIC allows individual interrupts for each peripheral to be either enabled or disabled. When an enabled interrupt happens, the GIC causes an IRQ exception in the A9 processor. Since the same vector table entry is used for all interrupts, the software for the interrupt service routine must determine the source of the interrupt by querying the GIC. Each peripheral is identified in the GIC by an interrupt identification (ID) number. Table 1 gives the assignment of interrupt IDs for each of the I/O peripherals in the DE10-Standard Computer. The rest of this section describes the interrupt behavior associated with the timers and parallel ports.

3.1 Interrupts from the ARM* A9* Private Timer

Figure 3, in Section 2.4.1, shows four registers that are associated with the A9 private timer. As we said in Section 2.4.1, bit *F* in the *Interrupt status* register is set to 1 when the timer reaches a count value of 0. It is possible to generate an A9 interrupt when this occurs, by using bit *I* of the *Control* register. Setting bit *I* to 1 causes the timer to

I/O Peripheral	Interrupt ID #
A9 Private Timer	29
HPS GPIO1	197
HPS Timer 0	199
HPS Timer 1	200
HPS Timer 2	201
HPS Timer 3	202
FPGA Interval Timer	72
FPGA Pushbutton KEYs	73
FPGA Second Interval Timer	74
FPGA Audio	78
FPGA PS/2	79
FPGA JTAG	80
FPGA Infrared (IrDA)	81
FPGA JP1 Expansion	83
FPGA PS/2 Dual	89

Table 1. Interrupt IDs in the DE10-Standard Computer.

send an interrupt signal to the GIC whenever the timer reaches a count value of 0. The *F* bit can be cleared to 0 by writing writing a 1 into the *Interrupt status* register.

3.2 Interrupts from the HPS Timers

Figure 4, in Section 2.4.2, shows five registers that are associated with each HPS timer. As we said in Section 2.4.2, when the timer reaches a count value of zero, bit *F* in the *End-of-Interrupt* register is set to 1. The value of the *F* bit is also reflected in the *S* bit in the *Interrupt status* register. It is possible to generate an A9 interrupt when the *F* bit becomes 1, by using the *I* bit of the *Control* register. Setting bit *I* to 0 *unmasks* the interrupt signal, and causes the timer to send an interrupt signal to the GIC whenever the *F* bit is 1. After an interrupt occurs, it can be cleared by reading the *End-of-Interrupt* register.

3.3 Interrupts from the FPGA Interval Timer

Figure 13, in Section 2.12, shows six registers that are associated with the interval timer. As we said in Section 2.12, 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 GIC whenever *TO* becomes 1. After an interrupt occurs, it can be cleared by writing any value into the *Status* register.

3.4 Interrupts from Parallel Ports

Parallel ports implemented in the FPGA in the DE10-Standard Computer were illustrated in Figure 6, which is reproduced as Figure 14. As the figure shows, parallel ports that support interrupts include two related registers at

the addresses *Base* + 8 and *Base* + C. The *Interruptmask* register, which has the address *Base* + 8, specifies whether or not an interrupt signal should be sent to the GIC 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.

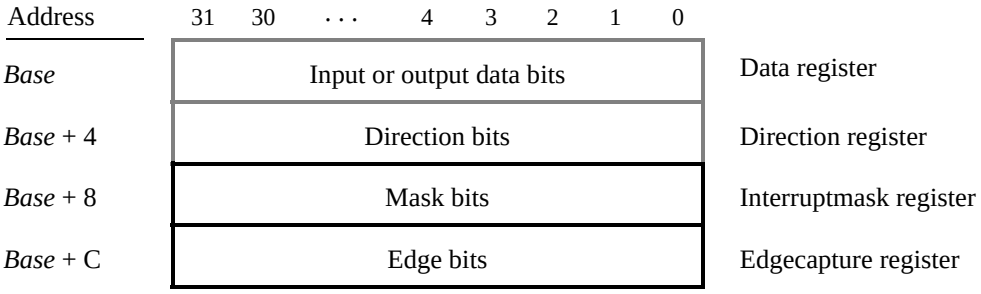


Figure 14. Registers used for interrupts from the parallel ports.

3.4.1 Interrupts from the Pushbutton Keys

Figure 10, reproduced as Figure 15, shows the registers associated with the pushbutton parallel 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 GIC.

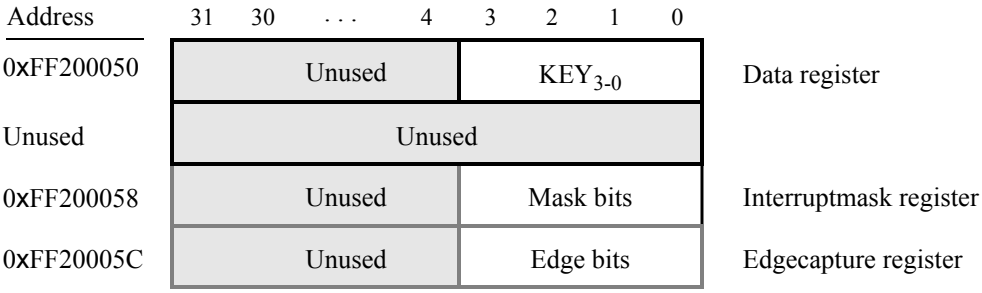


Figure 15. Registers used for interrupts from the pushbutton parallel port.

3.5 Interrupts from the JTAG* UART

Figure 12, reproduced as Figure 16, shows the data and *Control* registers of the JTAG UART. As we said in Section 2.11, *RAVAIL* in the *Data* register gives the number of characters that are stored in the receive FIFO, and *WSPACE* gives the amount of unused space that is available in the transmit FIFO. The *RE* and *WE* bits in Figure 16 are used to

enable processor interrupts associated with the receive and transmit FIFOs. When enabled, interrupts are generated when *RAVAIL* for the receive FIFO, or *WSPACE* for the transmit FIFO, exceeds 7. Pending interrupts are indicated in the Control register's *RI* and *WI* bits, and can be cleared by writing or reading data to/from the JTAG UART.

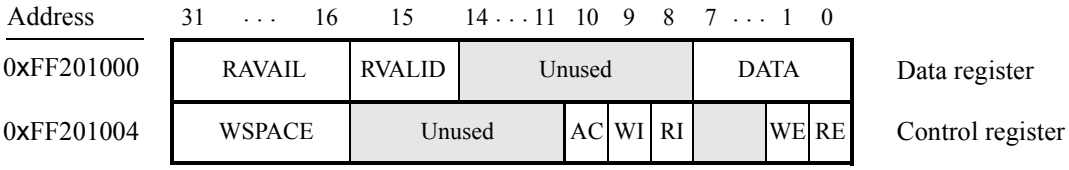


Figure 16. Interrupt bits in the JTAG UART registers.

3.6 Using Interrupts with Assembly Language Code

An example of assembly language code for the DE10-Standard Computer that uses interrupts is shown in Listing 7, which has three main parts. The beginning part of the code sets up the exception vector table. This code must be in a special assembler section called .**section**, as shown. The entries in the table provide branches to the various exception service routines; they are discussed later in this section.

When this code is executed on the DE10-Standard board it displays a rotating pattern on the LEDs. The pattern's rotation can be toggled through pressing the pushbutton KEYs. Different types of interrupts are used in the code. The LEDs are controlled by interrupts from the FPGA interval timer, and the KEYs are also handled through interrupts.

The main program initializes the A9 banked stack pointer (sp) registers for interrupt (IRQ) mode and supervisor (SVC) mode, because these are the processor modes that are used in the program. The code then calls subroutines to initialize the HPS timer, FPGA interval timer, and FPGA pushbutton KEYs. Finally, the code initializes the HPS GPIO1 port, enables IRQ interrupts in the A9 processor, and then enters an infinite loop. The loop code turns on and off a green light whenever the global variable named *tick* is set to 1. This variable is set to 1 by the exception service routine for the HPS timer, which is described later in this section.

Following are the subroutines used to initialize the timers and pushbutton KEYs. The CONFIG_HPS_TIMER routine sets up the HPS timer 0 so that it will produce an interrupt every one second. Since this timer uses a 100 MHz clock, the timer *load* register is initialized to the value 100×10^6. The CONFIG_INTERVAL_TIMER routine configures the FPGA interval timer to produce interrupts every 50 msec. Since this timer uses a 100 MHz clock, the required starting count value is 5×10^6. The CONFIG_KEYS routine sets up the FPGA KEYs parallel port to produce an interrupt when any KEY is pressed.

The last portion of the code shows the global data used by the program. It includes the *tick* variable that was discussed for the code earlier, and other variables. The *pattern* variable holds the bit-pattern that is written, the *key_pressed* variable indicates which FPGA KEY has been recently pressed, and the *shift_dir* variable specifies the direction of shifting for the HEX displays.

Also included in part *c* of Listing 7 is the subroutine that initializes the GIC. This code performs the minimum-required steps needed to configure the three interrupts used in the program, by writing to the *processor targets* (ICDIPTRn) registers in the GIC, and the *set enable* (ICDISERn) registers. For the HPS timer, the registers used have addresses 0xFFFED8C4 and 0xFFFED118, as shown in the listing. For the FPGA interval timer and KEYs,

the register addresses are 0xFFFED848 and 0xFFFED108. Instructions for calculating these addresses, and determining the bit patterns to write into them can be found in the tutorial *Using the Generic Interrupt Controller*, available in Intel's FPGA University Program website. The last part of the code in this section enables the CPU Interface and Distributor in the GIC.

The exception service routines for the main program in Listing 7 are given in Listing 8. The first part of the listing gives the IRQ exception handler. This routine first reads from the *interrupt acknowledge* register in the GIC to determine the interrupt ID of the peripheral that caused the interrupt. The code then checks which of the three possible sources of interrupt has occurred, and calls the corresponding interrupt service routine for the HPS timer, FPGA interval timer, or FPGA KEY parallel port. These interrupt service routine are shown in Listings 9 to 10.

Finally, the exception handler in Listing 8 writes to the *end-of-interrupt* register in the GIC to clear the interrupt, and then returns to the main program by using the instruction "SUBS PC, LR, #4".

The latter part of Listing 8 shows handlers for exceptions that correspond to the reset exception, various types of error conditions, and the FIQ interrupt. The reset handler shows a branch to the start of the main program in Listing 7. This handler is just an indicator of the result of performing a reset of the A9 processor—the actual reset process involves executing code from a special boot ROM on the processor, and then executing a program called the *preloader* before actually starting the main program. The other handlers in the latter part of Listing 8, which are just loops that branch to themselves, are intended to serve as placeholders for code that would handle the corresponding exceptions. More information about each of these types of exceptions can be found in the document *Introduction to the ARM Processor*, also available in Intel's FPGA University Program web site.

3.7 Using Interrupts with C Code

An example of C code for the DE10-Standard Computer that uses interrupts is shown in Figure 12. This code performs exactly the same operations as the code described in Listing 7.

Before it call subroutines to configure the generic interrupt controller (GIC), timers, and pushbutton KEY port, the main program first initializes the IRQ mode stack pointer by calling the routine *set_A9_IRQ_stack()*. The code for this routine uses in-line assembly language instructions, as shown in Part *b* of the listing. This step is necessary because the C compiler generates code to set only the supervisor mode stack, which is used for running the main program, but the compiler does not produce code for setting the IRQ mode stack. To enable IRQ interrupts in the A9 processor the main program uses the in-line assembly code shown in the subroutine called *enable_A9_interrupts()*.

The exception handlers for the main program in Listing 12 are given in Listing 13. These routines have unique names that are meaningful to the C compiler and linker tools, and they are declared with the special type of **__attribute__** called **interrupt**. These mechanisms cause the C compiler and linker to use the addresses of these routines as the contents of the exception vector table.

The function with the name *__cs3_isr_irq* is the IRQ exception handler. As discussed for the assembly language code in Listing 8 this routine first reads from the *interrupt acknowledge* register in the GIC to determine the interrupt ID of the peripheral that caused the interrupt, and then calls the corresponding interrupt service routine for either the HPS timer, FPGA interval timer, or FPGA KEY parallel port. These interrupt service routines are shown in Listings 14 to 15.

Listing 13 also shows handlers for exceptions that correspond to the various types of error conditions and the FIQ interrupt. These handlers are just loops that are meant to serve as place-holders for code that would handle the corresponding exceptions.

The source code files shown in Listing 7 to Listing 16 are distributed as part of the Intel FPGA Monitor Program. The files can be found under the heading *sample programs*, and are identified by the name *Interrupt Example*.

4 Media Components

This section describes the audio in/out, video-out, video-in, LCD, PS/2, IrDA*, and ADC ports.

4.1 Audio In/Out Port

The DE10-Standard Computer includes an audio port that is connected to the audio CODEC (COder/DECoder) chip on the DE10-Standard 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 DE10-Standard 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 17. 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 ARM A9 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 17 can be set to 1 to clear the *Read* and *Write* FIFOs, respectively. The clear function remains active until the corresponding bit is set back to 0.

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

Figure 17. Audio port registers.

The read-only *Fifospace* register in Figure 17 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 17. The code checks to see when the depth of either the left or right *Read* FIFO has exceeded 75% full, and then moves the data from these FIFOs into a memory buffer. This code is part of a program that is distributed as part of the Intel FPGA Monitor Program. The source code can be found under the heading *sample programs*, and is identified by the name *Audio*.

4.2 Video-out Port

The DE10-Standard Computer includes a video-out port connected to the on-board VGA controller that can be connected to a standard VGA monitor. The video-out port support a screen resolution of 640 × 480. The image that is displayed by the video-out port is derived from two sources: a *pixel* buffer, and a *character* buffer.

4.2.1 Pixel Buffer

The pixel buffer for the video-out port holds the data (color) for each pixel that will be displayed. As illustrated in Figure 18, 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.

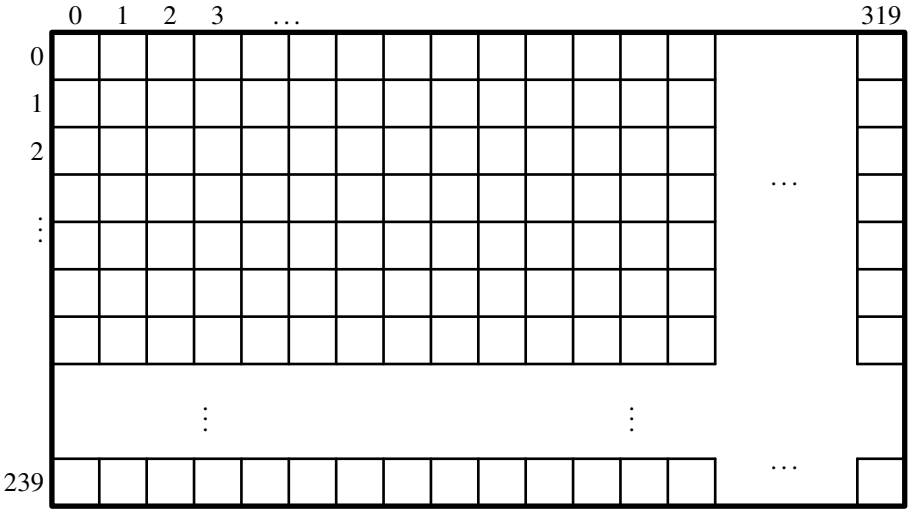
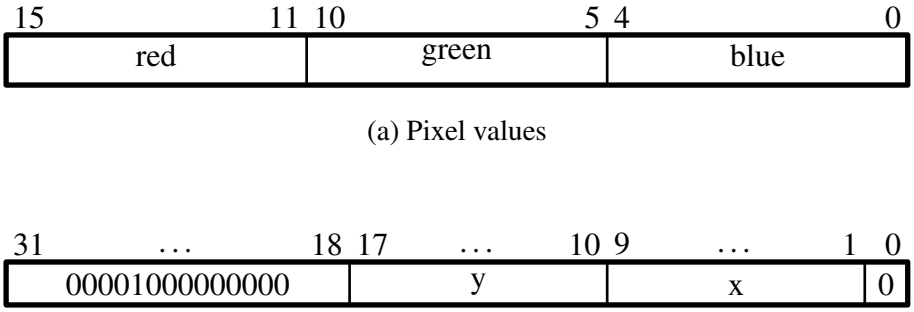


Figure 18. Pixel buffer coordinates.

Figure 19*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 19, pixels are addressed in the pixel buffer by using the combination of a *base* address and an *x,y* offset. In the DE10-Standard Computer the default address of the pixel

buffer is 0xC8000000, which corresponds to the starting address of the FPGA on-chip memory. Using this scheme, the pixel at location 0,0 has the address 0xC8000000, the pixel 1,0 has the address *base* + (00000000 000000001 0)$_2$ = 0xC8000002, the pixel 0,1 has the address *base* + (00000001 000000000 0)$_2$ = 0xC8000400, and the pixel at location 319,239 has the address *base* + (11101111 100111111 0)$_2$ = 0xC803BE7E.



(a) Pixel values

(b) Pixel address

Figure 19. Pixel values and addresses.

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

4.2.2 RGB Resampling

The DE10-Standard 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 DE10-Standard Computer video stream is 0x14, which corresponds to no alpha, color, 16-bit RGB (5-bit Red, 6-bit Green, 5-bit Blue). For more information, the reader should refer to the video module's online documentation, *Video IP Cores for Intel DE-Series Boards*, which is available from Intel's FPGA University Program web site.

4.2.3 Double Buffering

As mentioned above, a pixel buffer controller reads data out of the pixel buffer so that it can be displayed on the screen. This pixel buffer controller includes a programming interface in the form of a set of registers, as illustrated in Table 2. The register at address 0xFF203020 is called the *Buffer* register, and the register at address 0xFF203024 is the *Backbuffer* register. Each of these registers stores the starting address of a pixel buffer. The Buffer register holds the address of the pixel buffer that is displayed on the screen. As mentioned above, in the default configuration of the DE10-Standard Computer this Buffer register is set to the address 0xC8000000, which points to the start of the FPGA on-chip memory. The default value of the Backbuffer register is also 0xC8000000, 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 DE10-Standard Computer,

which has the base address 0xC0000000. Note that the pixel buffer cannot be located in the DDR3 memory in the DE10-Standard 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.

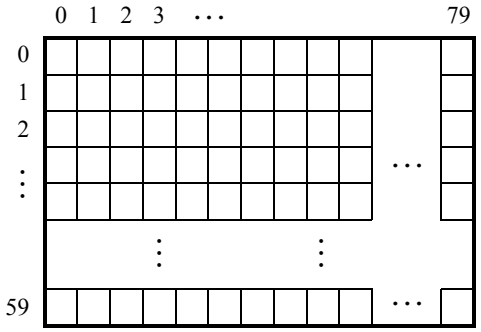
As shown in Table 2 the *Status* register contains additional information other than the *S* bit. The fields n and m give the number of address bits used for the X and Y pixel coordinates, respectively. The BS field specifies the number of data bits per symbol minus one. The SB field specifies the number 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 DE10-Standard 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.

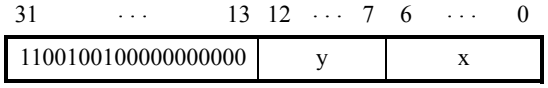
The programming interface also includes a *Resolution* register, shown in Table 2, that contains the X and Y resolution of the pixel buffer(s).

4.2.4 Character Buffer

The character buffer for the video-out port is stored in on-chip memory in the FPGA on the DE10-Standard board. As illustrated in Figure 20*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 20*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 20 shows that characters are addressed in the memory by using the combination of a *base* address, which has the value 0xC9000000, and an *x,y* offset. Using this scheme, the character at location 0,0 has the address 0xC9000000, the character 1,0 has the address *base* + $(000000\ 0000001)_2$ = 0xC9000001, the character 0,1 has the address *base* + $(000001\ 0000000)_2$ = 0xC9000080, and the character at location 79,59 has the address *base* + $(111011\ 1001111)_2$ = 0xC9001DCF.



(a) Character buffer coordinates



(b) Character buffer addresses

Figure 20. 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 18. 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 18 is included in the sample program called *Video* that is distributed with the Intel FPGA Monitor Program.

4.3 Video-in Port

The DE10-Standard Computer includes a video-in port for use with the composite video-in connector on the DE10-Standard 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 21. 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 puctuated by lighter lines along the edges that have been detected. Setting $E = 0$ causes a normal image to be stored into the pixel buffer.

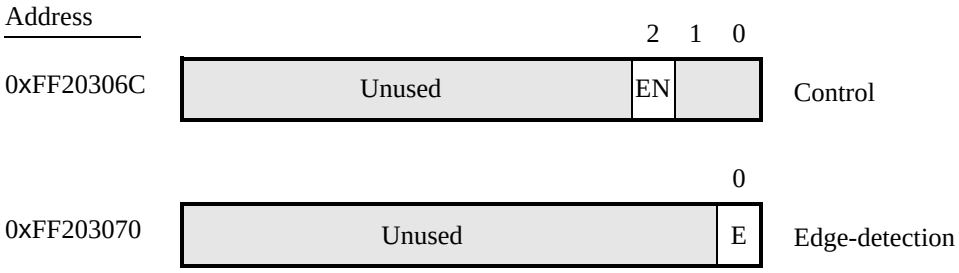


Figure 21. The video-in port programming interface.

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.

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 DE10-Standard 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.

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 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 DE10-Standard Computer is reset. For typical use of the DE10-Standard Computer it is not necessary to modify any of these default settings. In the case that changes to these settings are needed, the reader should refer to the audio/video configuration module's online documentation, which is available from Intel's FPGA University Program web site.

4.5 LCD Port

The DE10-Standard Computer includes a liquid crystal display (LCD) port that is connected to the 128 × 64 pixel display on the DE10-Standard board. Data is written to the LCD in pages where each page consists of an 128 × 8 pixel section of the display for a total of 8 pages. Each column of a page is presented as a byte, where each bit corresponds to one pixel in that column. To write to the display, the page address is first written to the data register of SPIM0 followed by the column addresses. Then the corresponding 8-bits of pixel data may be written. After writing one 8-bit column, the column address is incremented automatically, so that consecutive writes to the data register will be for adjacent columns.

Before writing to the display, SPIM0 and the LCD need to be initialized. To do this, SPIM0, which has address 0xFFF00000 is taken out of reset and put into transfer only mode. Then the baud rate is set to 0x40 and the slave register is enabled. Interrupts should be turned off before SPIM0 is turned on. To initialize the LCD, the output direction register of GPIO1, which has address 0xFF709000 is set to point to the LCD. Then the LCD backlight can be turned on and taken out of reset. Finally, the LCD registers must be initialized. An example using the LCD is provided in the Appendix at Listing 19.

4.6 PS/2 Port

The DE10-Standard Computer includes two PS/2 ports that can be connected to a standard PS/2 keyboard or mouse. The port includes a 256-byte FIFO that stores data received from a PS/2 device. The programming interface for the PS/2 port consists of two registers, as illustrated in Figure 22. 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.

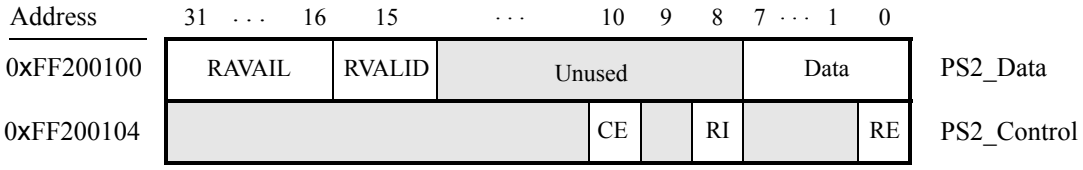


Figure 22. PS/2 port registers.

A fragment of C code that uses the PS/2 port is given in Listing 22. 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 Intel FPGA Monitor Program.

4.6.1 PS/2 Port Dual

The DE10-Standard Computer includes a second PS/2 port that allows both a keyboard and mouse to be used at the same time. To use the dual port a Y-splitter cable must be used and the keyboard and mouse must be connected to the PS/2 connector on the DE10-Standard board through this cable. The PS/2 port dual has the same registers as the PS/2 port shown in Listing 22, except that the base address of its *PS2_Data* register is 0xFF200108 and the base address of its *PS2_Control* register is 0xFF20010C.

4.7 IrDA* Port

The DE10-Standard Computer includes an IrDA UART for communicating wirelessly with peripherals over the infrared spectrum. It is configured for 8-bit data and one stop bit, and operates at a baud rate of 155,200. The default configuration does not use a parity bit. The programming interface consists of two 32-bit registers, as shown in Figure 23. The register at address 0xFF201020 is referred to as the *Data* register, and the register at address 0xFF201024 is the *Control* register.

The operation of the IrDA UART is similar to the Serial Port UART described above. Data recieved through the IrDA is stored in a 128-character FIFO in the UART. As shown in Figure 23, the number of characters, *RAVAIL*, currently

stored in this FIFO is provided in bits 23−16 of the *Data* register. If the FIFO overflows, then any additional data is lost. When a read of the *Data* register is performed, the character at the head of the FIFO is provided in bits 7−0. If the character read is valid, the the value of bit 15, *RVALID* will be one. If no data is available to be read from the receive FIFO, then *RVALID* will be set to 0 and the data in bits 7 − 0 is undefined.

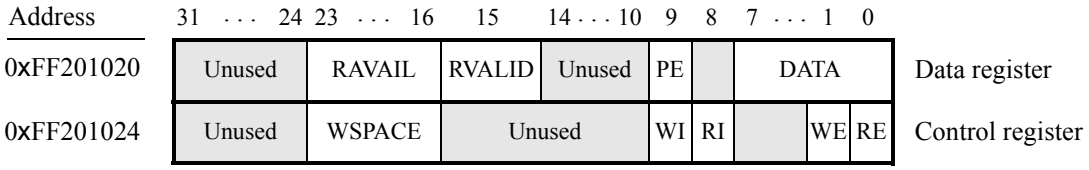


Figure 23. IrDA UART registers.

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

4.8 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 DE10-Standard board. As illustrated in Figure 24, 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 24 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.

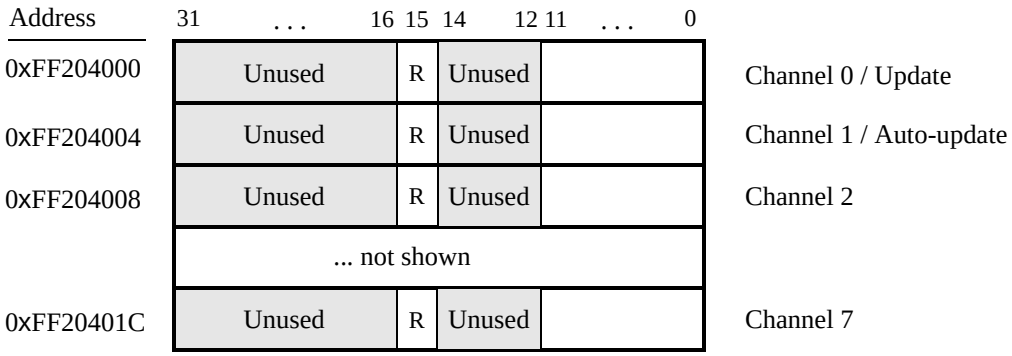


Figure 24. ADC port registers.

Figure 25 shows the connector on the DE10-Standard 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.

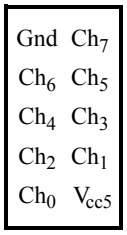


Figure 25. ADC connector.

5 Modifying the DE10-Standard Computer

It is possible to modify the DE10-Standard Computer by using Intel's Quartus® Prime software and Qsys tool. Tutorials that introduce this software are provided in the University Program section of Intel's web site. To modify the system it is first necessary to make an editable copy of the DE10-Standard Computer. The files for this system are installed as part of the Monitor Program installation. Locate these files, copy them to a working directory, and then use the Quartus Prime and Qsys software to make any desired changes.

Table 4 lists the names of the Qsys IP cores that are used in this system. When the DE10-Standard Computer design files are opened in the Quartus Prime software, these cores can be examined using the Qsys System Integration tool. Each core has a number of settings that are selectable in the Qsys System Integration tool, and includes a datasheet that provides detailed documentation.

The steps needed to modify the system are:

1. Install the *University Program IP Cores* from Intel's FPGA University Program web site
2. Copy the design source files for the DE10-Standard Computer from the University Program web site. These files can be found in the *Design Examples* section of the web site
3. Open the *DE10-Standard_Computer.qpf* project in the Quartus Prime software
4. Open the Qsys System Integration tool in the Quartus Prime software, and modify the system as desired
5. Generate the modified system by using the Qsys System Integration tool
6. It may be necessary to modify the Verilog or VHDL code in the top-level module, DE10-Standard_Computer.v/vhd, if any I/O peripherals have been added or removed from the system
7. Compile the project in the Quartus Prime software
8. Download the modified system into the DE10-Standard board

The DE10-Standard Computer includes a Nios II/f processor. When using the Quartus Prime Web Edition, compiling a design with a Nios II/s or Nios II/f processor will produce a time-limited SOF file. As a result, the board must

remain connected to the host computer, and the design cannot be set as the default configuration, as discussed in Section 6. Designs using only Nios II/e processors and designs compiled using the Quartus Prime Subscription Edition do not have this restriction.

I/O Peripheral	Qsys Core
SDRAM	SDRAM Controller
SRAM	SRAM Controller
On-chip memory character buffer	Character Buffer for VGA Display
SD Card	SD Card Interface
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
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. Qsys cores used in the DE10-Standard Computer.

6 Making the System the Default Configuration

The DE10-Standard Computer can be loaded into the nonvolatile FPGA configuration memory on the DE10-Standard board, so that it becomes the default system whenever the board is powered on. Instructions for configuring the DE10-Standard board in this manner can be found in the tutorial *Introduction to the Quartus Prime Software*, which is available from Intel's FPGA University Program.

7 Memory Layout

Table 5 summarizes the memory map used in the DE10-Standard Computer.

Base Address	End Address	I/O Peripheral
0x00000000	0x3FFFFFFF	DDR3 Memory
0xFFFF0000	0xFFFFFFFF	A9 On-chip Memory
0xC0000000	0xC3FFFFFF	SDRAM
0xC8000000	0xC803FFFF	FPGA On-chip Memory
0xC9000000	0xC9001FFF	FPGA On-chip Memory Character Buffer
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
0xFF200100	0xFF200107	PS/2
0xFF200108	0xFF20010F	PS/2 Dual
0xFF201000	0xFF201007	JTAG UART
0xFF201008	0xFF20100F	Second JTAG UART
0xFF201020	0xFF201027	Infrared (IrDA)
0xFF202000	0xFF20201F	Interval Timer
0xFF202020	0xFF20202F	Second Interval Timer
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
0xFF709000	0xFF709063	HPS GPIO1
0xFFC04000	0xFFC040FC	HPS I2C0
0xFFC08000	0xFFC08013	HPS Timer0
0xFFC09000	0xFFC09013	HPS Timer1
0xFFD00000	0xFFD00013	HPS Timer2
0xFFD01000	0xFFD01013	HPS Timer3
0xFFD0501C	0xFFD0501F	FPGA Bridge
0xFFFEC100	0xFFFEC1FC	GIC CPU Interface
0xFFFED000	0xFFFEDFFC	GIC Distributor Interface
0xFFFEC600	0xFFFEC60F	ARM A9 Private Timer

Table 5. Memory layout used in the DE10-Standard Computer.

8 Intel FPGA Monitor Program Integration

As we mentioned earlier, the DE10-Standard Computer system, and the sample programs described in this document, are made available as part of the Intel FPGA Monitor Program. Figures 26 to 29 show a series of windows that are used in the Monitor Program to create a new project. In the first screen, shown in Figure 26, the user specifies a file system folder where the project will be stored, gives the project a name, and specifies the type of processor that is being used. Pressing Next opens the window in Figure 27. Here, the user can select the DE10-Standard Computer as a pre-designed system. The Monitor Program then fills in the relevant information in the *System details* box, which includes the appropriate system info and fpga configuration files, and preloader. The first of these files specifies to the Monitor Program information about the components that are available in the DE10-Standard Computer, such as the type of processor and memory components, and the address map. The second file is an FPGA programming bitstream for the DE10-Standard Computer, which can downloaded by the Monitor Program into the DE10-Standard board. Any system which contains a Hard Processor System (HPS) component must also specify the preloader to be run immediately following the circuit being downloaded. This preloader is used to configure the components within the HPS with the setting required for the specific board.

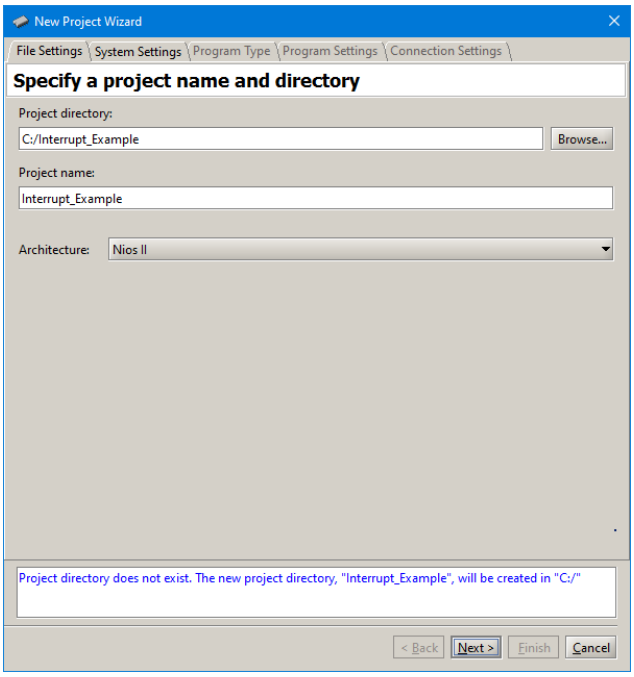


Figure 26. Specifying the project folder and project name.

Pressing Next again opens the window in Figure 28. Here the user selects the type of program that will be used, such as Assembly language, or C. Then, the check box shown in the figure can be used to display the list of sample programs for the DE10-Standard Computer that are described in this document. When a sample program is selected in this list, its source files, and other settings, can be copied into the project folder in subsequent screens of the Monitor Program.

Figure 29 gives the final screen that is used to create a new project in the Monitor Program. This screen shows the default addresses of compiler and linker sections that will be used for the assembly language or C program associated with the Monitor Program project. In the figure, the drop-down menu called *Linker Section Presets* has been set to Exceptions. With this setting the Monitor Program uses specific compiler and linker sections for the selected processor. For the Nios II processor, these sections are for reset and exceptions code, and another section for the main program, called *.text*. For the A9 processor, it has a section for the exception table, called *.vectors*, and another section for the main program, called *.text*. It also shows the initial value used to set the main stack pointer for C programs, which is the starting address of the *.stack* section.

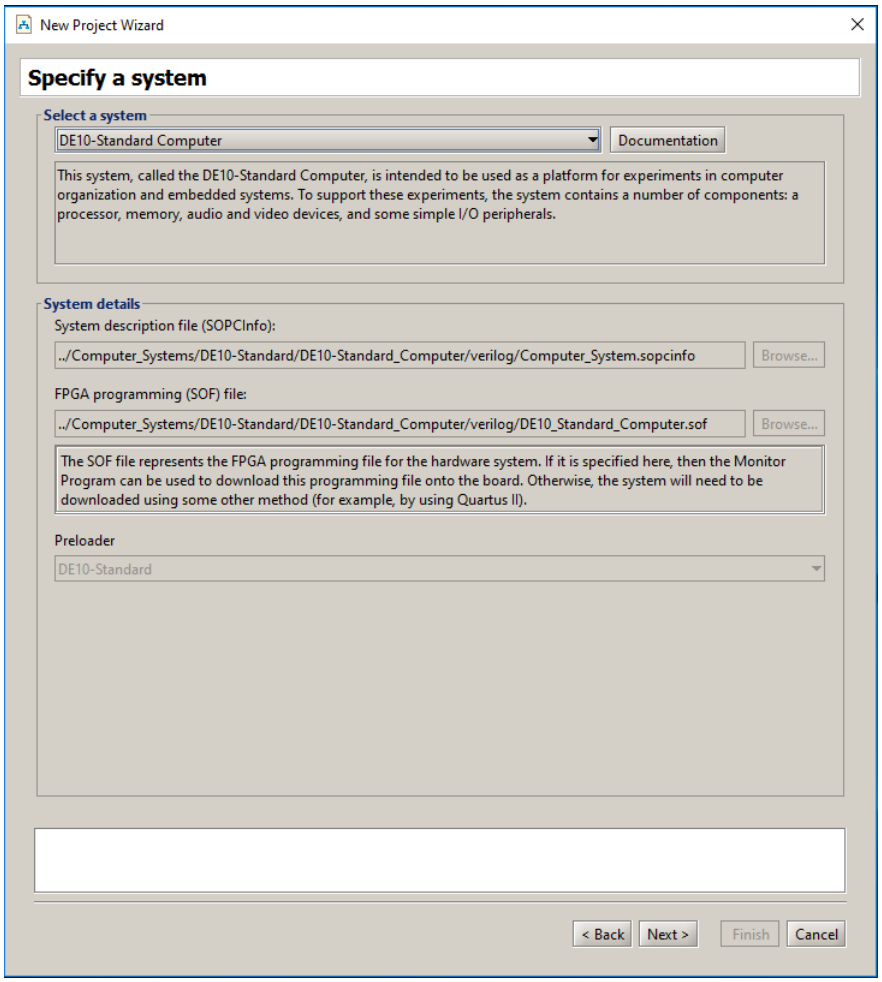


Figure 27. Specifying the DE10-Standard Computer system.

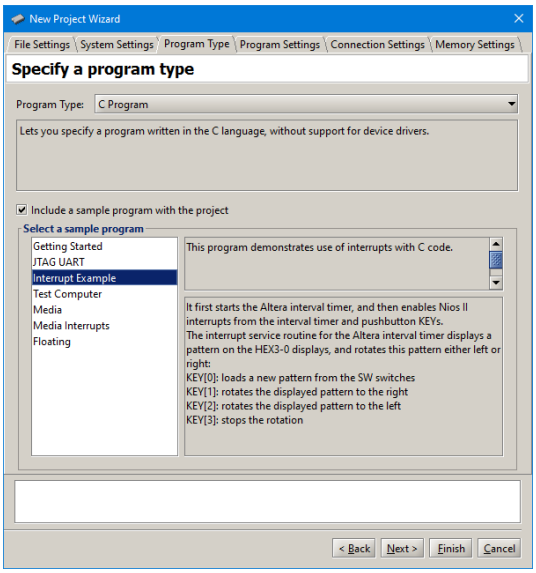


Figure 28. Selecting sample programs.

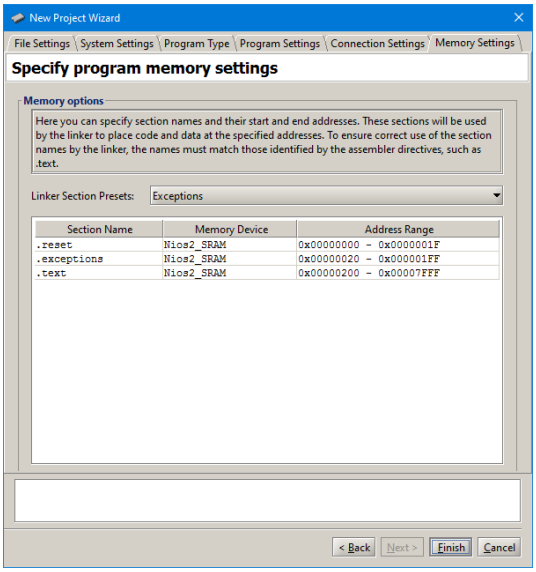


Figure 29. Setting offsets for *.text* and *.data*.

9 Appendix

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

9.1 Timers

```
.include    "address_map_arm.s"
.equ        bit_24_pattern, 0x01000000
/* This program provides a simple example of code for the ARM A9. The program
 * performs the following:
 *  1. starts the ARM A9 private timer
 *  2. loops forever, toggling the HPS green light LEDG when the timer expires
 */
.text
.global     _start
_start:
        LDR     R0, =HPS_GPIO1_BASE     // GPIO1 base address
        LDR     R1, =MPCORE_PRIV_TIMER  // MPCore private timer base address

        LDR     R2, =bit_24_pattern     // value to turn on the HPS green light LEDG
        STR     R2, [R0, #0x4]          // write to the data direction register to
                                        // set bit 24 (LEDG) of GPIO1 to be an
                                           output
        LDR     R3, =200000000          // timeout = 1/(200 MHz) x 200x10^6 = 1 sec
        STR     R3, [R1]                // write to timer load register
        MOV     R3, #0b011              // set bits: mode = 1 (auto), enable = 1
        STR     R3, [R1, #0x8]          // write to timer control register
LOOP:
        STR     R2, [R0]                // turn on/off LEDG
WAIT:
        LDR     R3, [R1, #0xC]          // read timer status
        CMP     R3, #0
        BEQ     WAIT                    // wait for timer to expire

        STR     R3, [R1, #0xC]          // reset timer flag bit
        EOR     R2, R2, #bit_24_pattern // toggle LEDG value
        B       LOOP

.end
```

Listing 1. An example of assembly language code that uses a timer.

```
#include "address_map_arm.h"

#define bit_24_pattern 0x01000000

/* This program provides a simple example of code for the ARM A9. The
 * program performs the following:
 *  1. starts the ARM A9 private timer
 *  2. loops indefinitely, toggling the green light LEDG when the timer expires
*/
int main(void)
{
    /* Declare volatile pointers to I/O registers (volatile means that the
     * locations will not be cached, even in registers) */
    volatile int * HPS_GPIO1_ptr            = (int *)HPS_GPIO1_BASE;
    volatile int * MPcore_private_timer_ptr = (int *)MPCORE_PRIV_TIMER;

    int HPS_LEDG = bit_24_pattern; // value to turn on the HPS green light LEDG
    int counter  = 200000000;      // timeout = 1/(200 MHz) x 200x10^6 = 1 sec

    *(HPS_GPIO1_ptr + 1) =
        bit_24_pattern; // write to the data direction register to set
                        // bit 24 (LEDG) of GPIO1 to be an output
    *(MPcore_private_timer_ptr)     = counter; // write to timer load register
    *(MPcore_private_timer_ptr + 2) = 0b011;   // mode = 1 (auto), enable = 1

    while (1)
    {
        *HPS_GPIO1_ptr = HPS_LEDG; // turn on/off LEDG
        while (*(MPcore_private_timer_ptr + 3) == 0)
            ;                                 // wait for timer to expire
        *(MPcore_private_timer_ptr + 3) = 1; // reset timer flag bit
        HPS_LEDG ^= bit_24_pattern;           // toggle bit that controls LEDG
    }
}
```

Listing 2. An example of C code that uses a timer.

9.2 Parallel Ports

```
.include    "address_map_arm.s"

/*********************************************************************************
 * 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 SW switches as the pattern
 ********************************************************************************/
.text        /* executable code follows */
.global     _start
_start:

        MOV     R0, #31         // used to rotate a bit pattern: 31 positions to the
                                // right is equivalent to 1 position to the left
        LDR     R1, =LED_BASE   // base address of LED lights
        LDR     R2, =SW_BASE    // base address of SW switches
        LDR     R3, =KEY_BASE   // base address of KEY pushbuttons
        LDR     R4, LED_bits

DO_DISPLAY:
        LDR     R5, [R2]        // load SW switches

        LDR     R6, [R3]        // load pushbutton keys
        CMP     R6, #0          // check if any key is presssed
        BEQ     NO_BUTTON

        MOV     R4, R5          // copy SW switch values onto LED displays
        ROR     R5, R5, #8      // the SW values are copied into the upper three
                                // bytes of the pattern register
        ORR     R4, R4, R5      // needed to make pattern consistent as all 32-bits
                                // of a register are rotated
        ROR     R5, R5, #8      // but only the lowest 8-bits are displayed on LEDs
        ORR     R4, R4, R5
        ROR     R5, R5, #8
        ORR     R4, R4, R5
WAIT:
        LDR     R6, [R3]        // load pushbuttons
        CMP     R6, #0
        BNE     WAIT            // wait for button release

NO_BUTTON:
        STR     R4, [R1]        // store pattern to the LED displays
        ROR     R4, R0          // rotate the displayed pattern to the left

        LDR     R6, =50000000   // delay counter
DELAY:
        SUBS    R6, R6, #1
        BNE     DELAY
```

```
        B       DO_DISPLAY

LED_bits:
.word       0x0F0F0F0F

.end
```

Listing 3. An example of ARM A9 assembly language code that uses parallel ports.

```
#include "address_map_arm.h"
/* This program demonstrates use of parallel ports in the Computer System
 *
 * It performs the following:
 *  1. displays a rotating pattern on the green LED
 *  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 4. An example of C code that uses parallel ports.

9.3 JTAG* UART

```
.include    "address_map_arm.s"
/*****************************************************************************
 * This program demonstrates use of the JTAG UART port in the DE1-SoC Computer
 * It performs the following:
 *   1. sends an example text string to the JTAG UART
 *   2. reads and echos character data from/to the JTAG UART
 *****************************************************************************/
.text                           /* executable code follows */
.global     _start
_start:
/* set up stack pointer */
        MOV     SP, #DDR_END - 3    // highest memory word address

/* print a text string */
        LDR     R4, =TEXT_STRING
LOOP:
        LDRB    R0, [R4]
        CMP     R0, #0
        BEQ     CONT                // string is null-terminated

        BL      PUT_JTAG            // send the character in R0 to UART
        ADD     R4, R4, #1
        B       LOOP

/* read and echo characters */
CONT:
        BL      GET_JTAG            // read from the JTAG UART
        CMP     R0, #0              // check if a character was read
        BEQ     CONT
        BL      PUT_JTAG
        B       CONT

/*****************************************************************************
 * Subroutine to send a character to the JTAG UART
 *      R0  = character to send
 *****************************************************************************/
.global     PUT_JTAG
PUT_JTAG:
        LDR     R1, =JTAG_UART_BASE // JTAG UART base address
        LDR     R2, [R1, #4]        // read the JTAG UART control register
        LDR     R3, =0xFFFF0000
        ANDS    R2, R2, R3          // check for write space
        BEQ     END_PUT             // if no space, ignore the character
        STR     R0, [R1]            // send the character
END_PUT:
        BX      LR
```

Listing 5. An example of assembly language code that uses the JTAG UART (Part a).

```
/*****************************************************************************
 * Subroutine to get a character from the JTAG UART
 *      returns the character read in R0
 ****************************************************************************/
.global     GET_JTAG
GET_JTAG:
        LDR     R1, =JTAG_UART_BASE // JTAG UART base address
        LDR     R0, [R1]            // read the JTAG UART data register
        ANDS    R2, R0, #0x8000     // check if there is new data
        BEQ     RET_NULL            // if no data, return 0
        AND     R0, R0, #0x00FF     // return the character
        B       END_GET
RET_NULL:
        MOV     R0, #0
END_GET:
        BX      LR
```

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

```
#include "JTAG_UART.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 6. An example of C code that uses the JTAG UART (Part a).

```
#include "JTAG_UART.h"
#include "address_map_arm.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 6. An example of C code that uses the JTAG UART (Part *b*).

9.4 Interrupts

```
/*********************************************************************************
 * Initialize the exception vector table
 ********************************************************************************/
.section    .vectors, "ax"

        B       _start                      // reset vector
        B       SERVICE_UND                 // undefined instruction vector
        B       SERVICE_SVC                 // software interrrupt vector
        B       SERVICE_ABT_INST            // aborted prefetch vector
        B       SERVICE_ABT_DATA            // aborted data vector
.word       0                               // unused vector
        B       SERVICE_IRQ                 // IRQ interrupt vector
        B       SERVICE_FIQ                 // FIQ interrupt vector
```

Listing 7. An example of assembly language code that uses interrupts (Part *a*).

```
/*********************************************************************************
 * This program demonstrates use of interrupts with assembly code. It first starts
 * two timers: an HPS timer, and the Altera interval timer (in the FPGA). The
 * program responds to interrupts from these timers in addition to the pushbutton
 * KEYs in the FPGA.
 *
 * The interrupt service routine for the HPS timer causes the main program to flash
 * the green light connected to the HPS GPIO1 port.
 *
 * The interrupt service routine for the interval timer displays a pattern on
 * the LED lights, and shifts this pattern either left or right. The shifting
 * direction is set in the pushbutton interrupt service routine; it is reversed
 * each time a KEY is pressed
 ********************************************************************************/
.text
.global     _start
_start:
/* Set up stack pointers for IRQ and SVC processor modes */
        MOV     R1, #INT_DISABLE | IRQ_MODE
        MSR     CPSR_c, R1                  // change to IRQ mode
        LDR     SP, =A9_ONCHIP_END - 3      // set IRQ stack to top of A9 onchip
                                            // memory

/* Change to SVC (supervisor) mode with interrupts disabled */
        MOV     R1, #INT_DISABLE | SVC_MODE
        MSR     CPSR_c, R1                  // change to supervisor mode
        LDR     SP, =DDR_END - 3            // set SVC stack to top of DDR3 memory

        BL      CONFIG_GIC                  // configure the ARM generic interrupt
                                            // controller
        BL      CONFIG_HPS_TIMER            // configure the HPS timer
        BL      CONFIG_INTERVAL_TIMER       // configure the Altera interval timer
        BL      CONFIG_KEYS                 // configure the pushbutton KEYs
```

```
/* initialize the GPIO1 port */
        LDR     R0, =HPS_GPIO1_BASE         // GPIO1 base address
        MOV     R4, #0x01000000             // value to turn on the HPS green light
                                            // LEDG
        STR     R4, [R0, #0x4]              // write to the data direction register
                                            // to set
                                            // bit 24 (LEDG) to be an output
/* enable IRQ interrupts in the processor */
        MOV     R1, #INT_ENABLE | SVC_MODE  // IRQ unmasked, MODE = SVC
        MSR     CPSR_c, R1
        LDR     R3, =TICK                   // global variable
LOOP:
        LDR     R5, [R3]                    // read tick variable
        CMP     R5, #0                      // HPS timer expired?
        BEQ     LOOP
        MOV     R5, #0
        STR     R5, [R3]                    // reset tick variable
        STR     R4, [R0]                    // turn on/off LEDG
        EOR     R4, R4, #0x01000000         // toggle bit that controls LEDG
        B       LOOP

/* Configure the HPS timer to create interrupts at one-second intervals */
CONFIG_HPS_TIMER:
/* initialize the HPS timer */
        LDR     R0, =HPS_TIMER0_BASE        // base address
        MOV     R1, #0                      // used to stop the timer
        STR     R1, [R0, #0x8]              // write to timer control register
        LDR     R1, =100000000              // period = 1/(100 MHz) x (100 x 10^6)
                                            // = 1 sec
        STR     R1, [R0]                    // write to timer load register
        MOV     R1, #0b011                  // int mask = 0, mode = 1, enable = 1
        STR     R1, [R0, #0x8]              // write to timer control register
        BX      LR

/* Configure the Altera interval timer to create interrupts at 50-msec intervals */
CONFIG_INTERVAL_TIMER:
        LDR     R0, =TIMER_BASE
/* set the interval timer period for scrolling the LED displays */
        LDR     R1, =5000000                // 1/(100 MHz) x 5x10^6 = 50 msec
        STR     R1, [R0, #0x8]              // store the low half word of counter
                                            // start value
        LSR     R1, R1, #16
        STR     R1, [R0, #0xC]              // high half word of counter start value

                                            // start the interval timer, enable its
                                               interrupts
        MOV     R1, #0x7                    // START = 1, CONT = 1, ITO = 1
        STR     R1, [R0, #0x4]
        BX      LR
```

```
/* Configure the pushbutton KEYS to generate interrupts */
CONFIG_KEYS:
                                                // write to the pushbutton port
                                                    interrupt mask register
        LDR     R0, =KEY_BASE                   // pushbutton key base address
        MOV     R1, #0x3                        // set interrupt mask bits
        STR     R1, [R0, #0x8]                  // interrupt mask register is (base + 8)
        BX      LR


/* Global variables */
.global     TICK
TICK:
.word       0x0                             // used by HPS timer
.global     PATTERN
PATTERN:
.word       0x0F0F0F0F                      // pattern to show on the LED lights
.global     KEY_DIR
KEY_DIR:
.word       0
.end
```

Listing 7. An example of assembly language code that uses interrupts (Part *b*).

```
/*
 * Configure the Generic Interrupt Controller (GIC)
 */
.global     CONFIG_GIC
CONFIG_GIC:
/* configure the HPS timer interrupt */
        LDR     R0, =0xFFFED8C4         // ICDIPTRn: processor targets register
        LDR     R1, =0x01000000         // set target to cpu0
        STR     R1, [R0]

        LDR     R0, =0xFFFED118         // ICDISERn: set enable register
        LDR     R1, =0x00000080         // set interrupt enable
        STR     R1, [R0]

/* configure the FPGA IRQ0 (interval timer) and IRQ1 (KEYs) interrupts */
        LDR     R0, =0xFFFED848         // ICDIPTRn: processor targets register
        LDR     R1, =0x00000101         // set targets to cpu0
        STR     R1, [R0]

        LDR     R0, =0xFFFED108         // ICDISERn: set enable register
        LDR     R1, =0x00000300         // set interrupt enable
        STR     R1, [R0]

/* configure the GIC CPU interface */
        LDR     R0, =MPCORE_GIC_CPUIF   // base address of CPU interface
/* Set Interrupt Priority Mask Register (ICCPMR) */
```

```
        LDR     R1, =0xFFFF             // 0xFFFF enables interrupts of all
                                        // priorities levels
        STR     R1, [R0, #ICCPMR]
/* Set the enable bit in the CPU Interface Control Register (ICCICR). This bit
 * allows interrupts to be forwarded to the CPU(s) */
        MOV     R1, #ENABLE
        STR     R1, [R0, #ICCICR]

/* Set the enable bit in the Distributor Control Register (ICDDCR). This bit
 * allows the distributor to forward interrupts to the CPU interface(s) */
        LDR     R0, =MPCORE_GIC_DIST
        STR     R1, [R0, #ICDDCR]
        BX      LR
```

Listing 7. An example of assembly language code that uses interrupts (Part *c*).

```
/*--- IRQ ---------------------------------------------------------------------*/
.global     SERVICE_IRQ
SERVICE_IRQ:
        PUSH    {R0-R7, LR}

/* Read the ICCIAR from the CPU interface */
        LDR     R4, =MPCORE_GIC_CPUIF
        LDR     R5, [R4, #ICCIAR]       // read the interrupt ID

HPS_TIMER_CHECK:
        CMP     R5, #HPS_TIMER0_IRQ     // check for HPS timer interrupt
        BNE     INTERVAL_TIMER_CHECK

        BL      HPS_TIMER_ISR
        B       EXIT_IRQ

INTERVAL_TIMER_CHECK:
        CMP     R5, #INTERVAL_TIMER_IRQ // check for FPGA timer interrupt
        BNE     KEYS_CHECK

        BL      TIMER_ISR
        B       EXIT_IRQ

KEYS_CHECK:
        CMP     R5, #KEYS_IRQ
UNEXPECTED:
        BNE     UNEXPECTED              // if not recognized, stop here

        BL      KEY_ISR
EXIT_IRQ:
/* Write to the End of Interrupt Register (ICCEOIR) */
        STR     R5, [R4, #ICCEOIR]

        POP     {R0-R7, LR}
        SUBS    PC, LR, #4


/*--- Undefined instructions --------------------------------------------------*/
.global     SERVICE_UND
SERVICE_UND:
        B       SERVICE_UND

/*--- Software interrupts -----------------------------------------------------*/
.global     SERVICE_SVC
SERVICE_SVC:
        B       SERVICE_SVC

/*--- Aborted data reads ------------------------------------------------------*/
.global     SERVICE_ABT_DATA
SERVICE_ABT_DATA:
        B       SERVICE_ABT_DATA
```

```
/*--- Aborted instruction fetch ---------------------------------------------*/
.global     SERVICE_ABT_INST
SERVICE_ABT_INST:
        B       SERVICE_ABT_INST

/*--- FIQ -------------------------------------------------------------------*/
.global     SERVICE_FIQ
SERVICE_FIQ:
        B       SERVICE_FIQ
```

Listing 8. Exception handlers assembly language code.

```
.include    "address_map_arm.s"
.extern     TICK

/*****************************************************************************
 * HPS timer0 interrupt service routine
 *
 * This code increments the TICK global variable, and clears the interrupt
 *****************************************************************************/

.global     HPS_TIMER_ISR
HPS_TIMER_ISR:

        LDR     R0, =HPS_TIMER0_BASE    // base address of timer
        LDR     R1, =TICK               // used by main program

        LDR     R2, [R1]
        ADD     R2, R2, #1
        STR     R2, [R1]                // ++tick

        LDR     R0, [R0, #0xC]          // read timer end of interrupt register to
                                        // clear the interrupt
        BX      LR
.end
```

Listing 9. Interrupt service routine for the HPS timer.

```
.include    "address_map_arm.s"
.include    "defines.s"
/* externally defined variables */
.extern     KEY_DIR
.extern     PATTERN

/*****************************************************************************
 * Interval timer interrupt service routine
 *
 * Shifts a PATTERN being displayed on the LED lights. The shift direction
 * is determined by the external variable KEY_PRESSED.
 *
 *****************************************************************************/
.global     TIMER_ISR
TIMER_ISR:
        PUSH    {R4-R7}
        LDR     R1, =TIMER_BASE // interval timer base address
        MOVS    R0, #0
        STR     R0, [R1]        // clear the interrupt

        LDR     R1, =LED_BASE   // LED base address
        LDR     R2, =PATTERN    // set up a pointer to the pattern for LED displays
        LDR     R7, =KEY_DIR    // set up a pointer to the shift direction variable

        LDR     R6, [R2]        // load pattern for LED displays
        STR     R6, [R1]        // store to LEDs


SHIFT:
        LDR     R5, [R7]        // get shift direction
        CMP     R5, #RIGHT
        BNE     SHIFT_L
        MOVS    R5, #1          // used to rotate right by 1 position
        RORS    R6, R5          // rotate right for KEY1
        B       END_TIMER_ISR
SHIFT_L:
        MOVS    R5, #31         // used to rotate left by 1 position
        RORS    R6, R5
END_TIMER_ISR:
        STR     R6, [R2]        // store LED display pattern
        POP     {R4-R7}
        BX      LR

.end
```

Listing 10. Interrupt service routine for the interval timer.

```
.include    "address_map_arm.s"
.include    "defines.s"
.extern     KEY_DIR         /* externally defined variable */
/*****************************************************************************
 * Pushbutton KEY - Interrupt Service Routine
 *
 * This routine toggles the KEY_DIR variable from 0 <-> 1
 ****************************************************************************/
.global     KEY_ISR
KEY_ISR:
        LDR     R0, =KEY_BASE   // base address of pushbutton KEY parallel port
/* KEY[1] is the only key configured for interrupts, so just clear it. */
        LDR     R1, [R0, #0xC]  // read edge capture register
        STR     R1, [R0, #0xC]  // clear the interrupt

        LDR     R1, =KEY_DIR    // set up a pointer to the shift direction variable
        LDR     R2, [R1]        // load value of shift direction variable
        EOR     R2, R2, #1      // toggle the shift direction
        STR     R2, [R1]

END_KEY_ISR:
        BX      LR
.end
```

Listing 11. Interrupt service routine for the pushbutton KEYs.

```
void set_A9_IRQ_stack(void);
void config_GIC(void);
void config_HPS_timer(void);
void config_HPS_GPIO1(void);
void config_interval_timer(void);
void config_KEYs(void);
void enable_A9_interrupts(void);
/* key_dir and pattern are written by interrupt service routines; we have to
 * declare these as volatile to avoid the compiler caching their values in
 * registers */
volatile int tick    = 0; // set to 1 every time the HPS timer expires
volatile int key_dir = 0;
volatile int pattern = 0x0F0F0F0F; // pattern for LED lights

/* ****************************************************************************
 * This program demonstrates use of interrupts with C code. It first starts
 * two timers: an HPS timer, and the FPGA interval timer. The program responds
 * to interrupts from these timers in addition to the pushbutton KEYs in the
 * FPGA.
 *
 * The interrupt service routine for the HPS timer causes the main program to
 * flash the green light connected to the HPS GPIO1 port.
 *
 * The interrupt service routine for the interval timer displays a pattern on
 * the LED lights, and shifts this pattern either left or right. The shifting
 * direction is reversed when KEY[1] is pressed
 ****************************************************************************/
int main(void)
{
    volatile int * HPS_GPIO1_ptr = (int *)HPS_GPIO1_BASE; // GPIO1 base address
    volatile int   HPS_timer_LEDG =
        0x01000000; // value to turn on the HPS green light LEDG

    set_A9_IRQ_stack();      // initialize the stack pointer for IRQ mode
    config_GIC();            // configure the general interrupt controller
    config_HPS_timer();      // configure the HPS timer
    config_HPS_GPIO1();      // configure the HPS GPIO1 interface
    config_interval_timer(); // configure Altera interval timer to generate
                             // interrupts
    config_KEYs();           // configure pushbutton KEYs to generate interrupts

    enable_A9_interrupts(); // enable interrupts

    while (1)
    {
        if (tick)
        {
            tick           = 0;
            *HPS_GPIO1_ptr = HPS_timer_LEDG; // turn on/off the green light LEDG
            HPS_timer_LEDG ^= 0x01000000; // toggle the bit that controls LEDG
        }
```

```
    }
}

/* setup HPS timer */
void config_HPS_timer()
{
    volatile int * HPS_timer_ptr = (int *)HPS_TIMER0_BASE; // timer base address

    *(HPS_timer_ptr + 0x2) = 0; // write to control register to stop timer
    /* set the timer period */
    int counter      = 100000000; // period = 1/(100 MHz) x (100 x 10^6) = 1 sec
    *(HPS_timer_ptr) = counter;   // write to timer load register

    /* write to control register to start timer, with interrupts */
    *(HPS_timer_ptr + 2) = 0b011; // int mask = 0, mode = 1, enable = 1
}

/* setup HPS GPIO1. The GPIO1 port has one green light (LEDG) and one pushbutton
 * KEY connected for the DE1-SoC Computer. The KEY is connected to GPIO1[25],
 * and is not used here. The green LED is connected to GPIO1[24]. */
void config_HPS_GPIO1()
{
    volatile int * HPS_GPIO1_ptr = (int *)HPS_GPIO1_BASE; // GPIO1 base address

    *(HPS_GPIO1_ptr + 0x1) =
        0x01000000; // write to the data direction register to set
                    // bit 24 (LEDG) to be an output
    // Other possible actions include setting up GPIO1 to use the KEY, including
    // setting the debounce option and causing the KEY to generate an interrupt.
    // We do not use the KEY in this example.
}

/* setup the interval timer interrupts in the FPGA */
void config_interval_timer()
{
    volatile int * interval_timer_ptr =
        (int *)TIMER_BASE; // interal timer base address

    /* set the interval timer period for scrolling the HEX displays */
    int counter                 = 5000000; // 1/(100 MHz) x 5x10^6 = 50 msec
    *(interval_timer_ptr + 0x2) = (counter & 0xFFFF);
    *(interval_timer_ptr + 0x3) = (counter >> 16) & 0xFFFF;

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

/* setup the KEY interrupts in the FPGA */
void config_KEYs()
{
    volatile int * KEY_ptr = (int *)KEY_BASE; // pushbutton KEY address
```

```
    *(KEY_ptr + 2) = 0x3; // enable interrupts for KEY[1]
}

/*
 * Initialize the banked stack pointer register for IRQ mode
*/
void set_A9_IRQ_stack(void)
{
    int stack, mode;
    stack = A9_ONCHIP_END - 7; // top of A9 onchip memory, aligned to 8 bytes
    /* change processor to IRQ mode with interrupts disabled */
    mode = INT_DISABLE | IRQ_MODE;
    asm("msr cpsr, %[ps]" : : [ps] "r"(mode));
    /* set banked stack pointer */
    asm("mov sp, %[ps]" : : [ps] "r"(stack));

    /* go back to SVC mode before executing subroutine return! */
    mode = INT_DISABLE | SVC_MODE;
    asm("msr cpsr, %[ps]" : : [ps] "r"(mode));
}

/*
 * Turn on interrupts in the ARM processor
*/
void enable_A9_interrupts(void)
{
    int status = SVC_MODE | INT_ENABLE;
    asm("msr cpsr, %[ps]" : : [ps] "r"(status));
}

/*
 * Configure the Generic Interrupt Controller (GIC)
*/
void config_GIC(void)
{
    int address; // used to calculate register addresses

    /* configure the HPS timer interrupt */
    *((int *)0xFFFED8C4) = 0x01000000;
    *((int *)0xFFFED118) = 0x00000080;

    /* configure the FPGA interval timer and KEYs interrupts */
    *((int *)0xFFFED848) = 0x00000101;
    *((int *)0xFFFED108) = 0x00000300;

    // Set Interrupt Priority Mask Register (ICCPMR). Enable interrupts of all
    // priorities
    address           = MPCORE_GIC_CPUIF + ICCPMR;
    *((int *)address) = 0xFFFF;
```

```
    // Set CPU Interface Control Register (ICCICR). Enable signaling of
    // interrupts
    address           = MPCORE_GIC_CPUIF + ICCICR;
    *((int *)address) = ENABLE;

    // Configure the Distributor Control Register (ICDDCR) to send pending
    // interrupts to CPUs
    address           = MPCORE_GIC_DIST + ICDDCR;
    *((int *)address) = ENABLE;
}
```

Listing 12. An example of C code that uses interrupts.

```
void HPS_timer_ISR(void);
void interval_timer_ISR(void);
void pushbutton_ISR(void);

// Define the IRQ exception handler
void __attribute__((interrupt)) __cs3_isr_irq(void)
{
    // Read the ICCIAR from the processor interface
    int address = MPCORE_GIC_CPUIF + ICCIAR;
    int int_ID  = *((int *)address);

    if (int_ID == HPS_TIMER0_IRQ) // check if interrupt is from the HPS timer
        HPS_timer_ISR();
    else if (int_ID ==
             INTERVAL_TIMER_IRQ) // check if interrupt is from the Altera timer
        interval_timer_ISR();
    else if (int_ID == KEYS_IRQ) // check if interrupt is from the KEYs
        pushbutton_ISR();
    else
        while (1)
            ; // if unexpected, then stay here

    // Write to the End of Interrupt Register (ICCEOIR)
    address           = MPCORE_GIC_CPUIF + ICCEOIR;
    *((int *)address) = int_ID;

    return;
}

// Define the remaining exception handlers
void __attribute__((interrupt)) __cs3_reset(void)
{
    while (1)
        ;
}

void __attribute__((interrupt)) __cs3_isr_undef(void)
{
    while (1)
        ;
}

void __attribute__((interrupt)) __cs3_isr_swi(void)
{
    while (1)
        ;
}

void __attribute__((interrupt)) __cs3_isr_pabort(void)
{
    while (1)
```

```
        ;
}

void __attribute__((interrupt)) __cs3_isr_dabort(void)
{
    while (1)
        ;
}

void __attribute__((interrupt)) __cs3_isr_fiq(void)
{
    while (1)
        ;
}
```

Listing 13. Exception handlers C code.

```
#include "address_map_arm.h"

extern volatile int tick;

/*****************************************************************************
 * HPS timer0 interrupt service routine
 *
 * This code increments the tick variable, and clears the interrupt
 *****************************************************************************/
void HPS_timer_ISR()
{
    volatile int * HPS_timer_ptr = (int *)HPS_TIMER0_BASE; // HPS timer address

    ++tick; // used by main program

    *(HPS_timer_ptr + 3); // Read timer end of interrupt register to
                          // clear the interrupt
    return;
}
```

Listing 14. Interrupt service routine for the HPS timer.

```
#include "address_map_arm.h"

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

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

    *(LED_ptr) = pattern; // Display pattern on LED

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

    return;
}
```

Listing 15. Interrupt service routine for the interval timer.

```
#include "address_map_arm.h"

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

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

    key_dir ^= 1; // Toggle key_dir value

    return;
}
```

Listing 16. Interrupt service routine for the pushbutton KEYs.

9.5 Audio

```
#include "address_map_arm.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 17. An example of code that uses the audio port.

9.6 Video Out

```
#include "address_map_arm.h"

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Listing 18. An example of code that uses the video-out port.

9.7 LCD

```
#include "lcd_driver.h"
#include "address_map_arm.h"

void spim_write(int data)
{
    volatile int * spim0_sr = (int *) SPIM0_SR;
    volatile int * spim0_dr = (int *) SPIM0_DR;
    while (((*spim0_sr) & 0x4) != 0x4)
        ; // check status reg for empty
    (*spim0_dr) = data;
    while (((*spim0_sr) & 0x4) != 0x4)
        ; // check fifo is empty
    while (((*spim0_sr) & 0x1) != 0x0)
        ; // check spim has completed the transfer
}

void init_spim0(void)
{
    volatile int * rstmgr_premodrst = (int *) HPS_RSTMGR_PREMODRST;
    volatile int * spim0            = (int *) SPIM0_BASE;

    // Take SPIM0 out of reset
    *rstmgr_premodrst = *rstmgr_premodrst & (~0x00040000);

    // Turn SPIM0 OFF
    *(spim0 + 2) = 0x00000000;

    // Put SPIM0 in Tx Only Mode
    *(spim0 + 0) = *(spim0 + 0) & ~0x00000300;
    *(spim0 + 0) = *(spim0 + 0) | 0x00000100;

    // Set SPIM0 BAUD RATE
    *(spim0 + 5) = 0x00000040;

    // Set SPIM0 Slave Enable Register
    *(spim0 + 4) = 0x00000001;

    // Turn off interrupts
    *(spim0 + 11) = 0x00000000;

    // Turn SPIM0 ON
    *(spim0 + 2) = 0x00000001;
}

void init_lcd(void)
{
    volatile int * gpio1 = (int *) HPS_GPIO1_BASE;
    // Set GPIO1's direction register for the outputs to the LCD
    *(gpio1 + 1) = *(gpio1 + 1) | 0x00009100;
```

```
    // Turn on the LCD Backlight and take it out of reset
    *(gpio1) = 0x00008100;

    // Initialize LCD's registers
    spim_write(0x000000C8);
    spim_write(0x0000002F);
    spim_write(0x00000040);
    spim_write(0x000000B0);
    spim_write(0x00000000);
    spim_write(0x00000010);
    spim_write(0x000000AF);
}

/*
 * Sets the mode of GPIO1.
 *
 * mode: 1 for command mode, 0 for data mode
 */
void set_mode(int mode)
{
    volatile int * gpio1 = (int *) HPS_GPIO1_BASE;

    if (mode) // Enter command mode
        *(gpio1) = (*gpio1) & (~0x00001000);
    else // Enter data mode
        *(gpio1) = (*gpio1) | (0x00001000);
}
```

Listing 19. An example of C language code that uses the LCD. (Part *a*)

```
char frame_buffer[8][128];

/*
 * Clears the entire LCD display.
 */
void clear_screen(void)
{
    int i, j;
    for (i = 0; i < FRAME_HEIGHT; i++)
    {
        for (j = 0; j < FRAME_WIDTH; j++)
        {
            frame_buffer[i][j] = 0;
        }
    }

    refresh_buffer();
}

/*
 * Writes the contents of the frame buffer to the LCD display.
 */
void refresh_buffer(void)
{
    int i, j;
    for (i = 0; i < FRAME_HEIGHT; i++)
    {
        set_mode(1);

        // Set page address
        spim_write(0x00B0 | i);
        // Set column address
        spim_write(0x0000);
        spim_write(0x0010);

        set_mode(0);
        for (j = 0; j < FRAME_WIDTH; j++)
            spim_write(frame_buffer[i][j]);
    }
}

/*
 * Draws a line starting at (x, y) with the given color to (x, y+length) if
 * vert, else to (x+length, y) to the frame buffer.
 *
 * x: x coordinate of line start.
 * y: y coordinate of line start.
 * length: length of line.
 * color: color of line (0 for white, 1 for black).
 * vert: orientation of line (0 for horizontal, 1 for vertical).
 */
```

```
void LCD_line(int x, int y, int length, int color, int vert)
{
    int  x_start, x_end, y_start, y_end;
    int  i, page;
    char mask;

    if (vert)
    {
        y_start = y;
        y_end   = y + length;

        for (i = y_start; i < y_end; i++)
        {
            page = i >> 3; // y/8
            mask = 0x01 << (i % 8);
            if (color)
                frame_buffer[page][x] |= mask;
            else
                frame_buffer[page][x] &= ~mask;
        }
    }
    else
    {
        x_start = x;
        x_end   = x + length;

        page = y >> 3; // y/8
        mask = 0x01 << (y % 8);
        for (i = x_start; i < x_end; i++)
        {
            if (color)
                frame_buffer[page][i] |= mask;
            else
                frame_buffer[page][i] &= ~mask;
        }
    }
}

/*
 * Draws a width x height rectangle with the top left corner at (x, y) to the
 * frame buffer.
 *
 * x1: x coordinate of top left corner.
 * y1: y coordinate of top left corner.
 * width: width of rectangle.
 * height: height of rectangle.
 * color: color of rectangle (0 for white, 1 for black).
 * fill: 1 if rectangle should be filled in, 0 to only draw rectangle outline.
 */
void LCD_rect(int x1, int y1, int width, int height, int color, int fill)
{
```

```
    int x2 = x1 + width;
    int y2 = y1 + width;
    int i;

    if (!fill)
    {
        LCD_line(x1, y1, width, color, 0);
        LCD_line(x1, y2, width, color, 0);
        LCD_line(x1, y1, height, color, 1);
        LCD_line(x2, y1, height, color, 1);
    }
    else
    {
        for (i = y1; i <= y2; i++)
            LCD_line(x1, i, width, color, 0);
    }
}

/*
 * Writes a string to the frame buffer starting at the given row.
 *
 * str: the string to write.
 * page: the row on the LCD to start writing at.
 */
void LCD_text(char * str, int page)
{
    int i, j, c, offset = 0, len = strlen(str);
    if (len > (8 - page) * 16)
    {
        printf("String is too long for LCD display.\n");
        return;
    }

    for (i = 0; i < len; i++)
    {
        if (offset >= FRAME_WIDTH)
        {
            page++;
            offset = 0;
        }
        c = (unsigned int)str[i];
        for (j                              = 0; j < 8; j++)
            frame_buffer[page][offset + j] = chars[c][j];
        offset += 8;
    }
}
```

Listing 20. An example of C language code that uses the LCD. (Part *b*)

```
#include "lcd_driver.h"
#include "lcd_graphic.h"

/*****************************************************************************
 * The program performs the following:
 * 1. Writes INTEL FPGA COMPUTER SYSTEMS to the top of the LCD.
 * 2. Bounces a filled in rectangle around the display and off the displayed
 * text.
 *****************************************************************************/
int main(void) {
    int          x, y, length, dir_x, dir_y;
    volatile int delay_count; // volatile so C compiler doesn't remove the loop

    /* create a message to be displayed on the VGA display */
    char text_top_lcd[17]    = "   INTEL FPGA   \0";
    char text_bottom_lcd[17] = "COMPUTER SYSTEMS\0";

    init_spim0();
    init_lcd();

    clear_screen();

    /* output the text message on the LCD display */
    LCD_text(text_top_lcd, 0);
    LCD_text(text_bottom_lcd, 1);

    /* initialize first position of box */
    x      = 0;
    y      = 16;
    length = 8;
    dir_x  = 1;
    dir_y  = 1;
    LCD_rect(x, y, length, length, 1, 1);

    refresh_buffer();

    while (1) {
        /* erase box */
        LCD_rect(x, y, length, length, 0, 1);

        /* update direction */
        if ((x + length >= SCREEN_WIDTH - 1 && dir_x == 1) ||
            (x <= 0 && dir_x == -1))
            dir_x = -dir_x;

        if ((y + length >= SCREEN_HEIGHT - 1 && dir_y == 1) ||
            (y <= 16 && dir_y == -1))
            dir_y = -dir_y;

        /* update coordinates */
        x += dir_x;
```

```
        y += dir_y;

        /* draw box */
        LCD_rect(x, y, length, length, 1, 1);
        refresh_buffer();

        for (delay_count = 100000; delay_count != 0; --delay_count)
            ; // delay loop
    }
}
```

Listing 21. An example of C language code that uses the LCD. (Part *c*)

9.8 PS/2

```
#include "address_map_arm.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 22. An example of code that uses the PS/2 port.