

1 Introduction

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

To support such experiments, the system contains embedded processors, memory, and various I/O peripherals. The FPGA programming file that implements this system, as well as its design source files, can be obtained from the University Program section of Intel's web site.

2 DE10-Nano Computer Contents

A block diagram of the DE10-Nano 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, analog-to-digital, parallel ports connected to switches and lights, expansion ports, and HDMI*.

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-Nano 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-Nano 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-Nano 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-Nano board, and similar operations. The Monitor Program includes the DE10-Nano Computer as a pre-designed system that can be downloaded onto the DE10-Nano board, as well as several sample programs in assembly language and C that show how to use the DE10-Nano Computer's peripherals. Section 8 describes how the DE10-Nano 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.

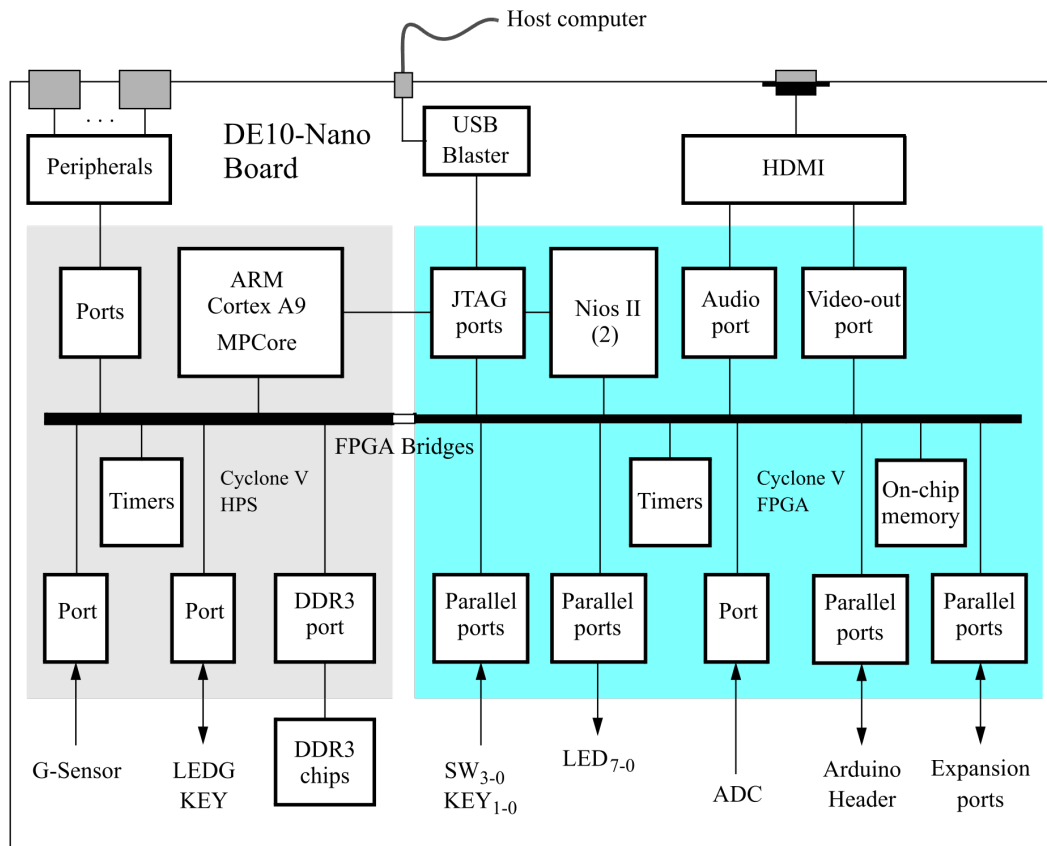


Figure 1. Block diagram of the DE10-Nano Computer.

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-Nano 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.

Address	31	...	25	24	23	...	0	
0xFF709000	Unused				Unused			Data register
0xFF709004								Data direction register
0xFF709030								Interrupt enable register
... not shown								
0xFF709050								External port register
... not shown								
0xFF709060								Level sync register

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 0xFFFFEC600, 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.

Address	31	...	16	15	...	8	7	3	2	1	0	Register name
0xFFFE600	Load value											Load
0xFFFE604	Current value											Counter
0xFFFE608	Unused					Prescaler		Unused	I	A	E	Control
0xFFFE60C	Unused										F	Interrupt status

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*.

Address	31	...	16	15	...	2	1	0	Register name	
0xFFC08000	Load value								Load	
0xFFC08004	Current value								Counter	
0xFFC08008	Unused						I	M	E	Control
0xFFC0800C	Unused							F	End-of-Interrupt	
0xFFC08010	Unused								S	Interrupt status

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-Nano 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-Nano 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-Nano Board User Manual*.

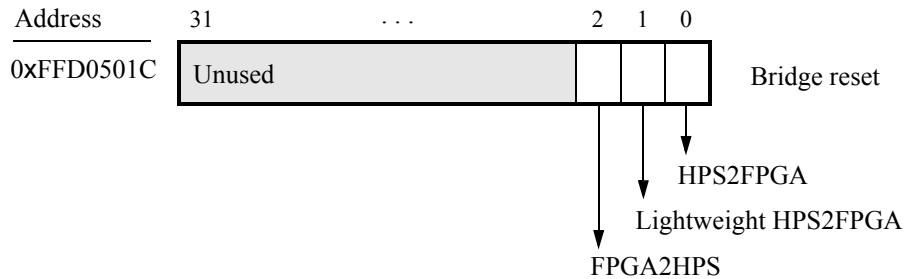


Figure 5. FPGA bridge reset register.

2.6 FPGA Components

As shown in Figure 1 a number of components in the DE10-Nano 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.

2.7 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-Nano 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-Nano Computer are provided in a separate document, called *DE10-Nano Computer with Nios II*.

2.8 On-Chip Memory

The DE10-Nano Computer includes a 256-Kbyte 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. This memory is accessible by the ARM processor as well as both Nios II processors.

2.9 On-Chip Memories

The DE10-Nano Computer includes two additional 32-Kbyte memories that are implemented in the Cyclone V FPGA chip. Each of these memories is organized as 8K x 32 bits, and can be accessed using either word, halfword, or byte operations. Each Nios II processor is connected to only one of these memories and can access the memory in the address range 0x00000000 to 0x00007FFF. Each Nios II processor uses this memory to access its reset and exception code, as well as other code or data.

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 $Base + 4$.

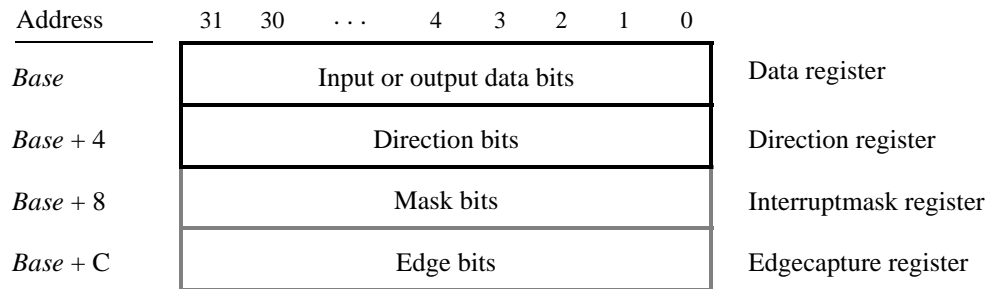


Figure 6. Parallel port registers in the DE10-Nano Computer.

Some of the parallel ports in the DE10-Nano 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 Green LED Parallel Port

The green lights LED_{7-0} on the DE10-Nano board are driven by an output parallel port, as illustrated in Figure 7. The port contains an 8-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.

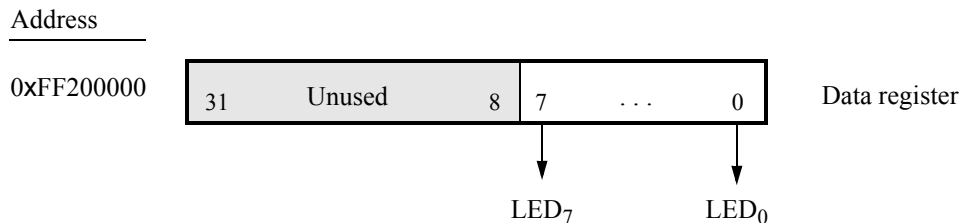


Figure 7. Output parallel port for *LED*.

2.10.2 Slider Switch Parallel Port

The SW_{3-0} slider switches on the DE10-Nano board are connected to an input parallel port. As illustrated in Figure 8, this port comprises a 4-bit read-only *Data* register, which is mapped to address $0xFF200040$.

2.10.3 Pushbutton Key Parallel Port

The parallel port connected to the KEY_{1-0} pushbutton switches on the DE10-Nano board comprises three 2-bit registers, as shown in Figure 9. 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_{1-0} . The other two registers shown

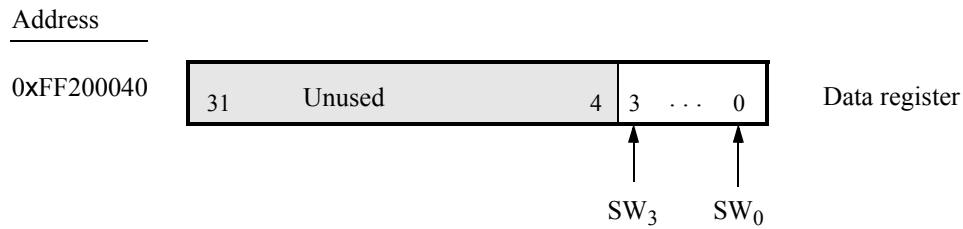


Figure 8. Data register in the slider switch parallel port.

in Figure 9, at addresses 0xFF200058 and 0xFF20005C, are discussed in Section 3.

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

Figure 9. Registers used in the pushbutton parallel port.

2.10.4 Expansion Parallel Port

The DE10-Nano Computer includes two bidirectional parallel ports that are connected to the JP1 and JP7 40-pin headers on the DE10-Nano board. These parallel ports include the four 32-bit registers that were described previously for Figure 6. The base address of the port for JP1 is 0xFF200060, and for JP7 is 0xFF200070. Figure 10 gives a diagram of the 40-pin connectors on the DE10-Nano board, and shows how the respective parallel port Data register bits, D₃₁₋₀, are assigned to the pins on the connector. The figure shows that bit D₀ of the parallel port is assigned to the pin at the top right corner of the connector, bit D₁ is assigned below this, and so on. Note that some of the pins on the 40-pin header are not usable as input/output connections, and are therefore not used by the parallel ports. Also, only 32 of the 36 data pins that appear on each connector can be used.

2.10.5 Arduino* Expansion Parallel Port

The DE10-Nano Computer includes a bidirectional parallel port that is connected to the Arduino* Uno R3 expansion header on the DE10-Nano board. This parallel port includes the four 32-bit registers that were described previously for Figure 6. The base address of the port is 0xFF200100. The Data register bits in this port are connected to the Arduino expansion header User I/O. Thus, bit 0 in the Data register connects to the signal Arduino_IO0, bit 1 to Arduino_IO1, and so on.

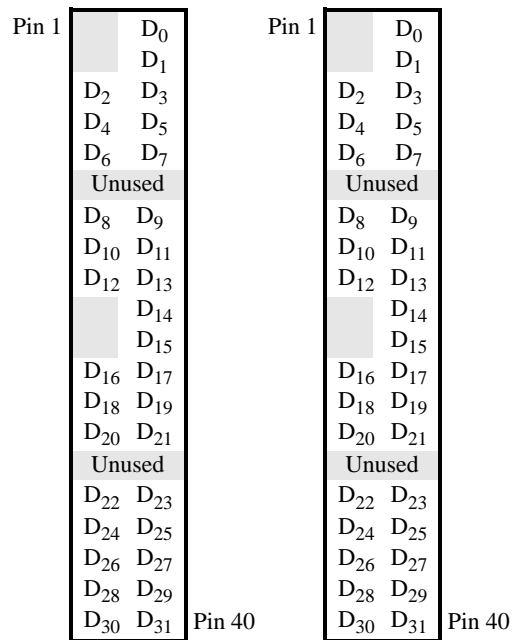


Figure 10. Assignment of parallel port bits to pins on *JP1* and *JP7*.

The DE10-Nano Computer also includes a one-bit output port that is connected to the Arduino Uno R3 expansion header on the DE10-Nano board. This one-bit port has a data register that is connected to the *Arduino_Reset_N* signal on the DE10-Nano board. The address of this port is 0xFF200110.

More details about the Arduino Uno R3 expansion header can be found in the DE10-Nano Board User Manual.

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

The DE10-Nano 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-Nano board and its host computer. This link can be used by the Intel Quartus® Prime software to transfer FPGA programming files into the DE10-Nano 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 11. The register mapped to address 0xFF201000 is called the *Data* register and the register mapped to address 0xFF201004 is called the *Control* register.

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

Figure 11. 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 11. 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-Nano 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-Nano 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 12. The 16-bit register at address 0xFF202000 provides status information about the timer, and the register at address 0xFF202004 allows control settings to be made. The bit fields in these registers are described below:

- *TO* provides a timeout signal which is set to 1 by the timer when it has reached a count value of zero. The *TO* bit can be reset by writing a 0 into it.
- *RUN* is set to 1 by the timer whenever it is currently counting. Write operations to the status halfword do not affect the value of the *RUN* bit.
- *ITO* is used for generating interrupts, which are discussed in section 3.

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

Figure 12. 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-Nano Computer gives a timer period of 125 msec. To achieve this period, the starting value of the count is $100 \text{ MHz} \times 125 \text{ msec} = 12.5 \times 10^6$. It is possible to capture a snapshot of the counter value at any time by performing a write to address 0xFF202010. This write operation causes the current 32-bit counter value to be stored into the two 16-bit timer registers at addresses 0xFF202010 and 0xFF202014. These registers can then be read to obtain the count value.

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

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-Nano 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 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 a 1 into the *Interrupt status* register.

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 JTAG	80
FPGA JP1 Expansion	83
FPGA JP7 Expansion	84
FPGA Arduino Expansion	85

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

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 12, 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-Nano Computer were illustrated in Figure 6, which is reproduced as Figure 13. 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 since it was last read. Performing a write operation to the *Edgecapture* register sets all bits in the register to 0, and clears any associated interrupts.

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

Figure 13. Registers used for interrupts from the parallel ports.

3.4.1 Interrupts from the Pushbutton Keys

Figure 9, reproduced as Figure 14, shows the registers associated with the pushbutton parallel port. The *Interrupt-mask* register allows processor interrupts to be generated when a key is pressed. Each bit in the *Edgecapture* register is set to 1 by the parallel port when the corresponding key is pressed. The Nios II processor can read this register to determine which key has been pressed, in addition to receiving an interrupt request if the corresponding bit in the interrupt mask register is set to 1. Writing any value to the *Edgecapture* register deasserts the Nios II interrupt request and sets all bits of the *Edgecapture* register to zero.

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

Figure 14. Registers used for interrupts from the pushbutton parallel port.

3.5 Interrupts from the JTAG* UART

Figure 11, reproduced as Figure 15, 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 15 are used to enable processor interrupts associated with the receive and transmit FIFOs. When enabled, interrupts are generated when *RAVAIL* for the receive FIFO, or *WSPACE* for the transmit FIFO, exceeds 7. Pending interrupts are indicated in the Control register's *RI* and *WI* bits, and can be cleared by writing or reading data to/from the JTAG UART.

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

Figure 15. Interrupt bits in the JTAG UART registers.

3.6 Using Interrupts with Assembly Language Code

An example of assembly language code for the DE10-Nano 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-Nano 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 0xFFFFED8C4 and 0xFFFFED118, as shown in the listing. For the FPGA interval timer and KEYS, the register addresses are 0xFFFFED848 and 0xFFFFED108. 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 *pre-loader* before actually starting the main program. More information about the reset process for the A9 processor can be found in the document "Using the Pre-loader Software for the A9 Processor," which is available from Intel's FPGA University Program website. 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-Nano 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-out, video-out, and ADC ports.

4.1 Audio Out Port

The DE10-Nano Computer includes an audio port that is connected to the HDMI Transmitter chip on the DE10-Nano board. The default setting for the sample rate provided by the audio port is 32K samples/sec. The port provides audio output functionality. The audio port includes two FIFOs that are used to hold outgoing data. Outgoing data is held in the left- and right-channel *Write* FIFOs. Both 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 16. The *Control* register, which has the address 0xFF203040, is readable to provide status information and writable to make control settings. Bit *WE* gives an interrupt enable capability for outgoing data. Setting this bit to 1 allows the audio core to generate an interrupt when either of the *Write* FIFOs are less than 25% full. The bit *WI* will be set to 1 to indicate that the interrupt is pending, and it can be cleared by filling the *Write* FIFOs until both are more than 25% full. The bit *CW* in Figure 16 can be set to 1 to clear the *Write* FIFOs. The clear function remains active until the corresponding bit is set back to 0.

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

Figure 16. Audio port registers.

The read-only *Fifospace* register in Figure 16 contains four 8-bit fields. 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 $WSRC = WSLC = 128$.

The *Leftdata* and *Rightdata* registers are writable for audio out. When data is written into these registers it is loaded into the *Write* FIFOs.

Note that the HDMI audio requires the HDMI video-out signals *HDMI_TX_HS* (Horizontal Synchronization) and *HDMI_TX_VS* (Vertical Synchronization) to be connected.

4.2 Video-out Port

The DE10-Nano Computer includes a video-out port connected to the on-board HDMI controller that can be connected to a standard HDMI 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 17, 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.

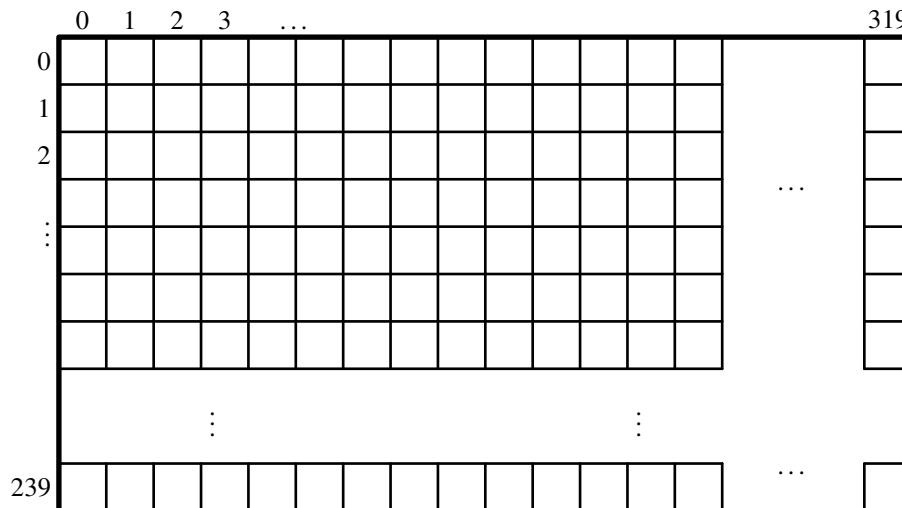


Figure 17. Pixel buffer coordinates.

Figure 18a shows that each pixel color is represented as a 8-bit symbol, with three bits for the green and red components, and two bits for blue. As depicted in part b of Figure 18, pixels are addressed in the pixel buffer by using the combination of a *base* address and an *x,y* offset. In the DE10-Nano 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 $base + (00000000\ 00000001)_2 = 0xC8000001$, the pixel 0,1 has the address $base + (00000001\ 00000000)_2 = 0xC8000200$, and the pixel at location 319,239 has the address $base + (11101111\ 10011111)_2 = 0xC801DF3F$.

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

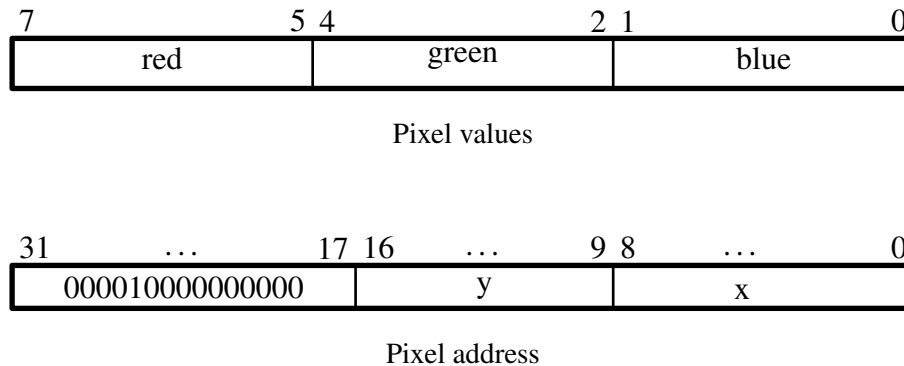


Figure 18. Pixel values and addresses.

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-Nano 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-Nano Computer video stream is 0x11, which corresponds to no alpha, color, 8-bit RGB (3-bit Red, 3-bit Green, 2-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-Nano 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-Nano Computer, which has the base address 0xC0000000. Note that the pixel buffer cannot be located in the DDR3 memory in the DE10-Nano 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.

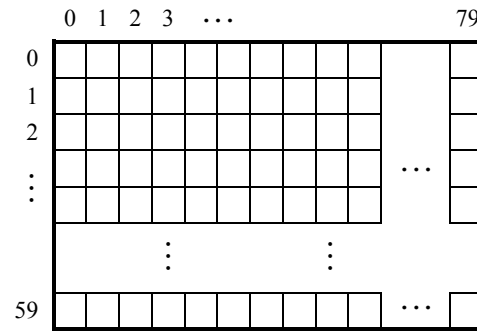
As shown in Table 2 the *Status* register contains additional information other than the *S* bit. The fields *n* and *m* give the number of address bits used for the *X* and *Y* pixel coordinates, respectively. The *BS* field specifies the number of data bits per symbol minus one. The *SB* field specifies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

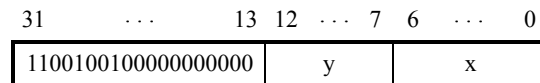
4.2.4 Character Buffer

The character buffer for the video-out port is stored in on-chip memory in the FPGA on the DE10-Nano board. As illustrated in Figure 19a, 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 19a 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 19 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 19. 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 17. 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 17 is included in the sample program called *Video* that is distributed with the Intel® FPGA Monitor Program.

4.3 Analog-to-Digital Conversion Port

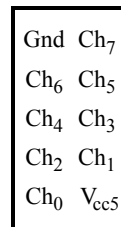
The Analog-to-Digital Conversion (ADC) Port provides access to the eight-channel, 12-bit analog-to-digital converter on the DE10-Nano board. As illustrated in Figure 20, the ADC port comprises eight 12-bit registers starting at the base address $0xFF204000$. The first two registers have dual purposes, 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 20 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.

Address	31	...	16	15	14	12	11	...	0	
0xFF204000	Unused			R	Unused					Channel 0 / Update
0xFF204004	Unused			R	Unused					Channel 1 / Auto-update
0xFF204008	Unused			R	Unused					Channel 2
... not shown										
0xFF20401C	Unused			R	Unused					Channel 7

Figure 20. ADC port registers.

Figure 21 shows the connector on the DE10-Nano 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.



JP15

Figure 21. ADC connector.

5 Modifying the DE10-Nano Computer

It is possible to modify the DE10-Nano 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-Nano 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 3 lists the names of the Qsys IP cores that are used in this system. When the DE10-Nano 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-Nano 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-Nano_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-Nano_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-Nano board

The DE10-Nano 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
On-chip memory character buffer	Character Buffer for Video Display
SD Card	SD Card Interface
Red LED parallel port	Parallel Port
Expansion parallel ports	Parallel Port
Slider switch parallel port	Parallel Port
Pushbutton parallel port	Parallel Port
JTAG port	JTAG UART
Interval timer	Interval timer
System ID	System ID Peripheral
Audio port	Audio
Video port	Pixel Buffer DMA Controller

Table 3. Qsys cores used in the DE10-Nano Computer.

6 Making the System the Default Configuration

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

7 Memory Layout

Table 4 summarizes the memory map used in the DE10-Nano Computer.

Base Address	End Address	I/O Peripheral
0x00000000	0x3FFFFFFF	DDR3 Memory
0xFFFF0000	0xFFFFFFFF	A9 On-chip Memory
0xC0000000	0xC0007FFF	FPGA On-chip Memory
0xC8000000	0xC803FFFF	FPGA On-chip Memory
0xFF200000	0xFF20000F	Green LEDs
0xFF200040	0xFF20004F	Slider Switches
0xFF200050	0xFF20005F	Pushbutton KEYs
0xFF200060	0xFF20006F	JP1 Expansion
0xFF200070	0xFF20007F	JP7 Expansion
0xFF200100	0xFF20010F	Arduino Expansion
0xFF200110	0xFF20011F	Arduino Reset
0xFF201000	0xFF201007	JTAG UART
0xFF201008	0xFF20100F	Second JTAG UART
0xFF202000	0xFF20201F	Interval Timer
0xFF202020	0xFF20202F	Second Interval Timer
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
0xFFFE1000	0xFFFE10FC	GIC CPU Interface
0xFFFE2000	0xFFFE20FC	GIC Distributor Interface
0xFFFE6000	0xFFFE606F	ARM A9 Private Timer

Table 4. Memory layout used in the DE10-Nano Computer.

8 Intel® FPGA Monitor Program Integration

As we mentioned earlier, the DE10-Nano Computer system, and the sample programs described in this document, are made available as part of the Intel® FPGA Monitor Program. Figures 22 to 25 show a series of windows that are used in the Monitor Program to create a new project. In the first screen, shown in Figure 22, 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 23. Here, the user can select the DE10-Nano 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-Nano Computer, such as the type of processor and memory components, and the address map. The second file is an FPGA programming bit-stream for the DE10-Nano Computer, which can be downloaded by the Monitor Program into the DE10-Nano 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.

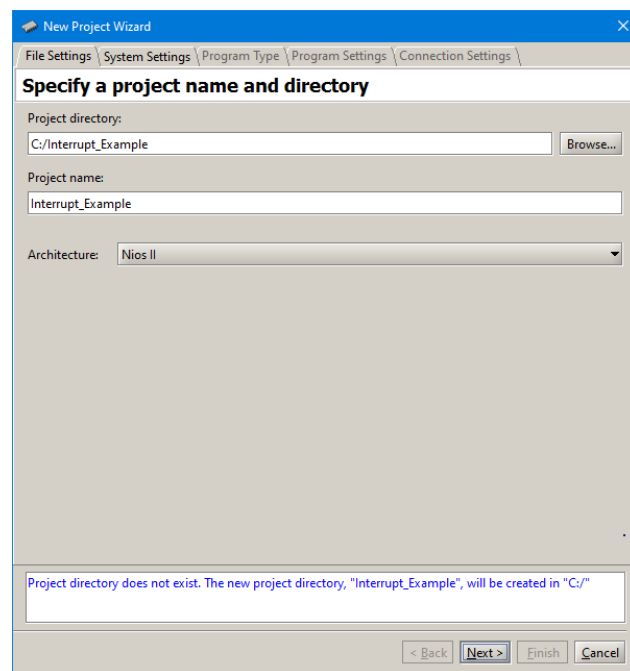


Figure 22. Specifying the project folder and project name.

Pressing **Next** again opens the window in Figure 24. 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-Nano 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 25 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.

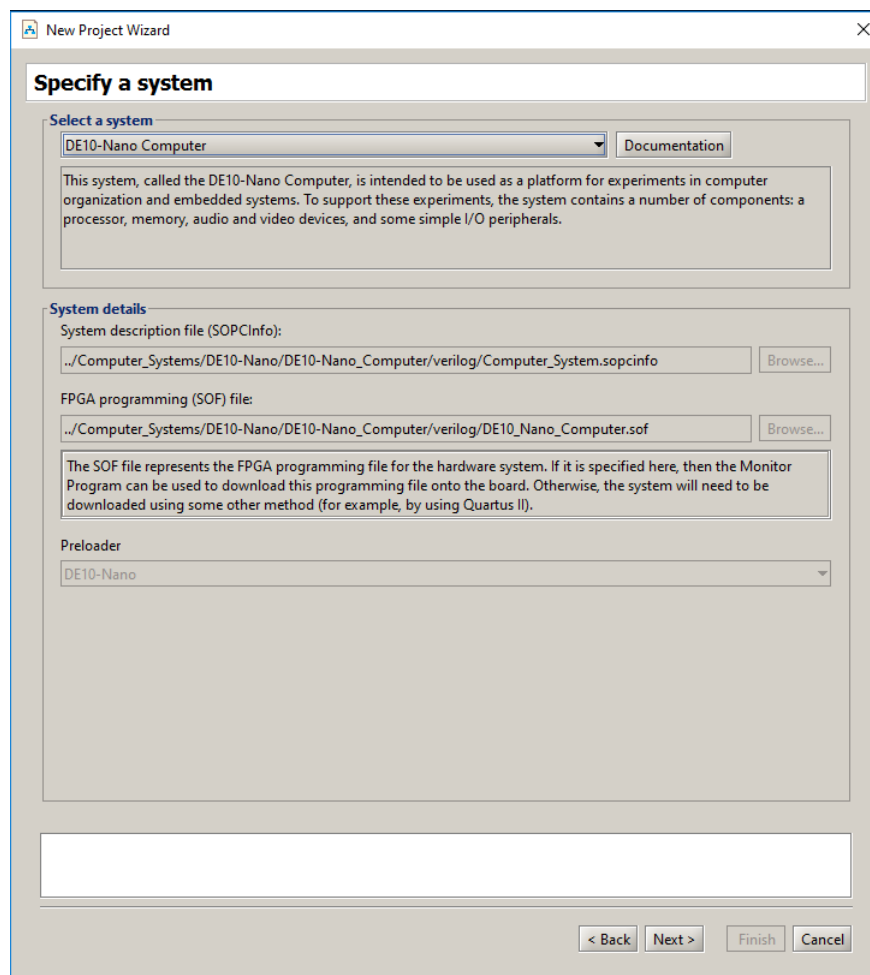


Figure 23. Specifying the DE10-Nano Computer system.

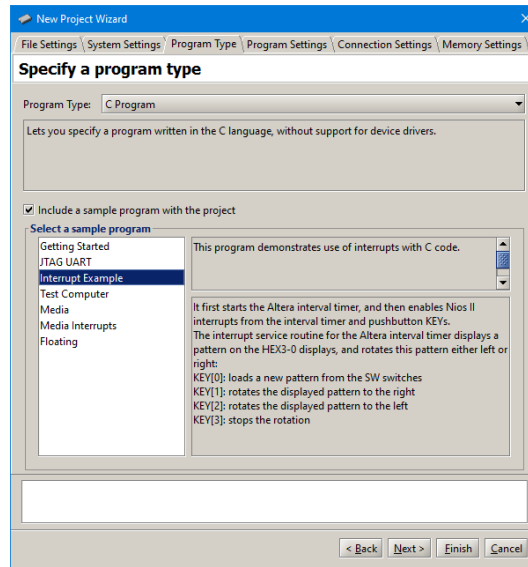


Figure 24. Selecting sample programs.

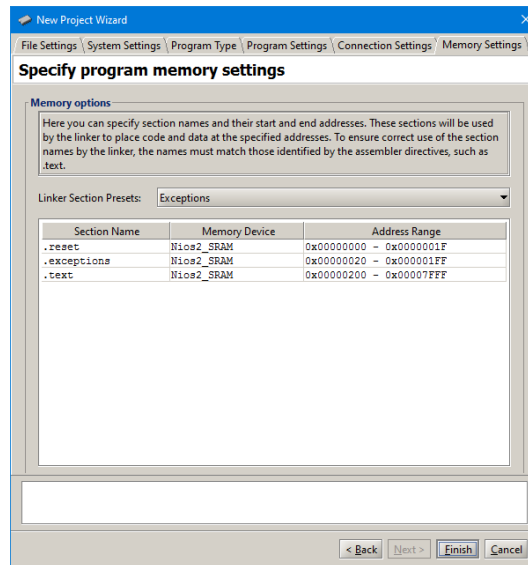


Figure 25. Setting offsets for .text and .data.

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 pressed
        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
.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 interrupt 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, =0xFFFFED8C4    // ICDIPTRn: processor targets register
    LDR    R1, =0x01000000     // set target to cpu0
    STR    R1, [R0]

    LDR    R0, =0xFFFFED118    // 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, =0xFFFFED848    // ICDIPTRn: processor targets register
    LDR    R1, =0x00000101     // set targets to cpu0
    STR    R1, [R0]

    LDR    R0, =0xFFFFED108    // 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; // interval 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 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 17. An example of code that uses the video-out port.

Copyright © Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Avalon, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.