Using the ARM* Generic Interrupt Controller

For Quartus® Prime 21.1

1 Introduction

This document introduces the ARM* Generic Interrupt Controller (GIC), which is included as part of the ARM Cortex-A9* MPCORE* processor in the Intel® Cyclone® V SoC family. We do not discuss some of the advanced features of the GIC in this document; complete information is available in the publication entitled *ARM Generic Interrupt Controller Architectural Specification*, which is available from ARM Holdings.

Contents:

2 ARM* Generic Interrupt Controller

As illustrated in Figure 1, the ARM *generic interrupt controller* (GIC) is a part of the ARM A9 MPCORE processor. The GIC is connected to the IRQ interrupt signals of all I/O peripheral devices that are capable of generating interrupts. Most of these devices are normally external to the A9 MPCORE, and some are internal peripherals (such as timers). The GIC included with the A9 MPCORE processor in the Intel Cyclone V SoC family handles up to 255 sources of interrupts. When a peripheral device sends its IRQ signal to the GIC, then the GIC can forward a corresponding IRQ signal to one or both of the A9 cores. Software code that is running on the A9 core can then query the GIC to determine which peripheral device caused the interrupt, and take appropriate action. The procedure for working with interrupts for the ARM Cortex-A9 and the GIC are described in the following sections.

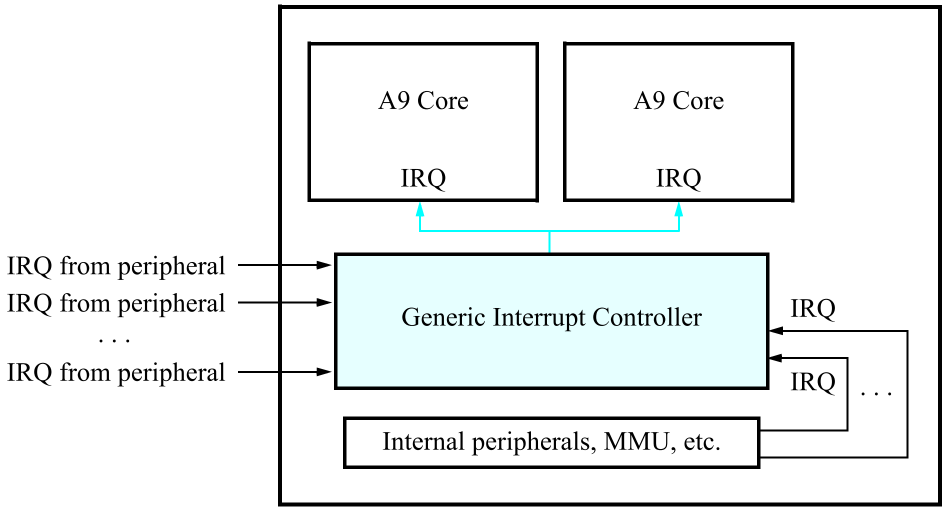


Figure 1. The ARM A9 MPCORE processor.

3 Interrupts in the ARM Cortex-A9*

An introduction to ARM processors can be found in the tutorial *Introduction to the ARM Processor Using Intel/ARM Toolchain*, which is available on Intel's FPGA University Program website. As described in that tutorial, the ARM Cortex-A9 has several main modes of operation, listed below:

- *User* mode – is the basic mode in which application programs run. This is an unprivileged mode, which has restricted access to system resources.
- *System* mode – provides full access to system resources. It can be entered only from one of the exception modes listed below.
- *Supervisor* mode – is entered when the processor executes a *supervisor call* instruction, SVC. It is also entered on reset or power-up.

- *Abort* mode – is entered if the processor attempts to access a non-legitimate memory location. This can happen, for example, when performing a word access for an address that is not word-aligned.
- *Undefined* mode – is entered if the processor attempts to execute an unimplemented instruction.
- *IRQ* mode – is entered in response to an interrupt request.
- *FIQ* mode – is entered in response to a *fast interrupt* request. We do not discuss fast interrupts in this document; they are used in some Cortex-A9 systems to provide faster service for more urgent requests. This document focuses only on IRQ interrupts.

When the processor is first powered on, or reset, it is in the *Supervisor* mode. This mode is *privileged*, which means that it allows the use of all processor instructions and operations. From supervisor mode it is possible to change into *User* mode, which is the only non-privileged mode. In User mode certain types of processor operations and instructions are prohibited. In practice, the Supervisor mode is normally used when the processor is executing software such as an operating system, whereas other software code may run in the User mode, thereby providing a level of protection for critical resources.

The operating mode of the processor is indicated in the current processor status register CPSR, as depicted in Figure 2. The mode bits are defined in Table 1.

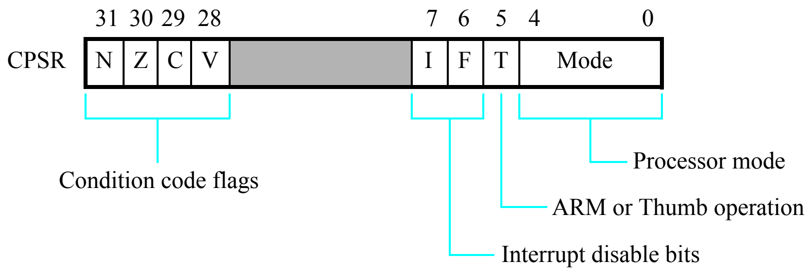


Figure 2. The current processor status register (CPSR).

TABLE 1. Mode Bits

CPSR$_{4-0}$	**Operating Mode**
10000	User
10001	FIQ
10010	IRQ
10011	Supervisor
10111	Abort
11011	Undefined
11111	System

To manipulate the contents of the CPSR, the processor must be in one of the privileged modes. Figure 3 shows the general-purpose registers in a Cortex-A9 processor, and illustrates how the registers are related to the processor mode. In User mode, there are 16 registers, $R0-R15$, plus the CPSR. These registers are also available in the System mode, which is not shown in the figure. As indicated in Figure 3, $R0-R12$, as well as the program counter $R15$, are common to all modes except FIQ. But the stack pointer register $R13$ and the link register $R14$ are not common—*banked* versions of these registers exist for each mode. Thus, the Supervisor mode has a stack pointer and link register that are used only when the processor is in this mode. Similarly, the other modes, such as IRQ mode, have their own stack pointers and link registers. The CPSR register is common for all modes, but when the processor is switched from one mode into another, the current content of the CPSR is copied into the new mode's saved processor status register (SPSR). Note that the FIQ mode, which we do not discuss in this document, has the additional banked registers $R8-R12$, as shown in the figure.

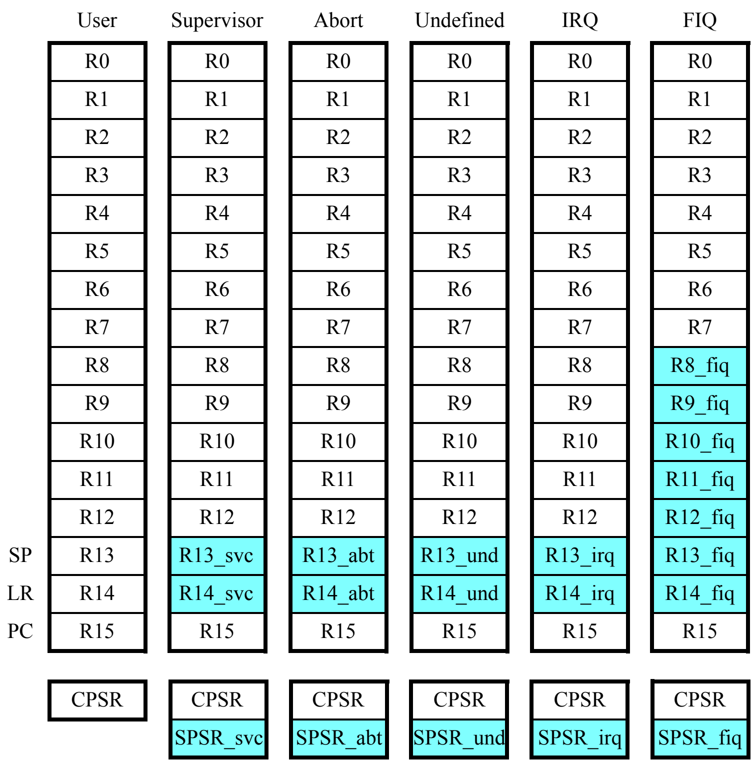


Figure 3. Banked registers in ARM processors.

3.1 IRQ Mode

A Cortex-A9 processor enters IRQ mode in response to receiving an IRQ signal from the GIC. Before such interrupts can be used, software code has to perform a number of steps:

1. Ensure that IRQ interrupts are disabled in the A9 processor, by setting the IRQ disable bit in the CPSR to 1.
2. Configure the GIC. Interrupts for each I/O peripheral device that is connected to the GIC are identified by a unique *interrupt ID*.

3. Configure each I/O peripheral device so that it can send IRQ interrupt requests to the GIC.
4. Enable IRQ interrupts in the A9 processor, by setting the IRQ disable bit in the CPSR to 0.

Examples of software code that perform these steps are given in Sections 5 and 6. Complete examples of interrupt-driven code are included in the appendices.

4 Programmer's Interface to the GIC

The GIC includes a number of memory-mapped registers that provide an *application programmer's interface* (API). As illustrated in Figure 4, the GIC architecture is divided into two main parts, called the *CPU Interface* and the *Distributor*. The CPU Interface is responsible for sending IRQ requests received by the Distributor to one or both of the A9 processors in the MPCORE. The Distributor receives IRQ interrupt signals from I/O peripherals.

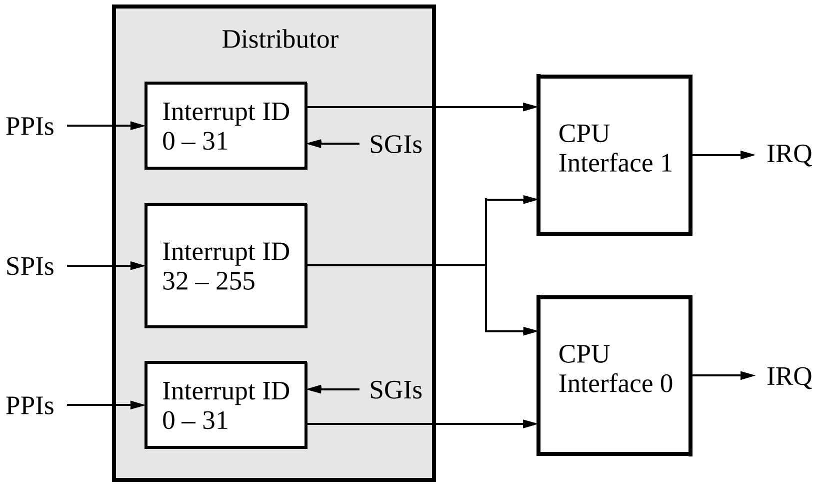


Figure 4. The GIC Architecture.

4.1 GIC CPU Interface

The CPU Interface in the GIC is used to send IRQ signals to the A9 cores. There is one CPU Interface for each A9 core in the MPCORE. API registers in each CPU Interface are depicted in Figure 5. To make the example more concrete, we have assigned addresses to these registers, as shown. These addresses correspond to those used in the document *DE1-SoC Computer System with ARM Cortex-A9*, which is available from Intel's FPGA University Program. The DE1-SoC Computer System is an ARM Cortex-A9 embedded system that can be implemented on Intel's DE1-SoC development and education board.

The *CPU Interface Control Register* (ICCICR) is used to enable forwarding of interrupts from the CPU Interface to the corresponding A9 core. Setting bit $E = 1$ in this register enables the sending of interrupts to the A9 core, and setting $E = 0$ disables these interrupts.

The *Interrupt Priority Mask Register* (ICCPMR) is used to set a threshold for the priority-level of interrupts that will be forwarded by a CPU Interface to an A9 core. Only interrupts that have a priority level greater than the *Priority* field in ICCPMR will be sent to an A9 processor by its CPU Interface. Lower priority values represent higher priority, meaning that level 0 is the highest priority and level 255 is the lowest. Setting the *Priority* field in ICCPMR to the value 0 will prevent any interrupts from being generated by the CPU Interface. The procedure for setting the priority level of individual interrupts (based on their Interrupt ID) is described in Section 4.2.

The *Interrupt Acknowledge Register* (ICCIAR) contains the Interrupt ID of the I/O peripheral that has caused an interrupt. When an A9 processor receives an IRQ signal from the GIC, software code (i.e., the *interrupt handler*) running on the processor must read the ICCIAR to determine which I/O peripheral has caused the interrupt.

After the A9 processor has completed the handling of an IRQ interrupt generated by the GIC, the processor must then clear this interrupt from the CPU Interface. This action is accomplished by writing the appropriate Interrupt ID into the *Interrupt ID* field in the *End of Interrupt Register* (ICCEOIR), depicted in Figure 5. After writing into the ICCEOIR, the interrupt handler software can then return control to the previously-interrupted main program.

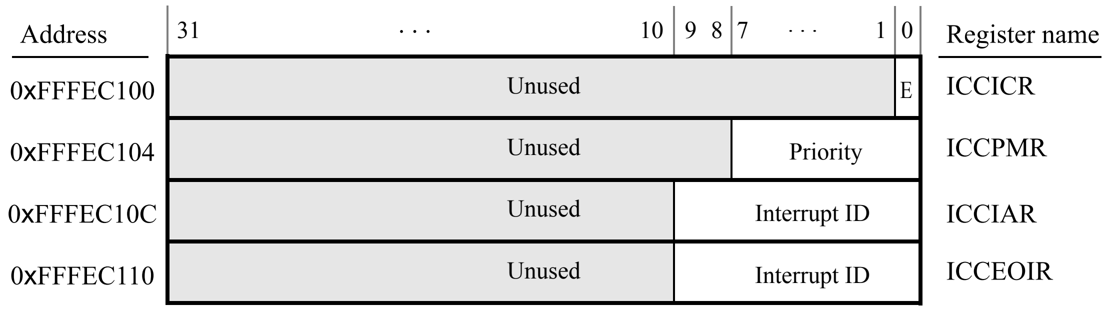


Figure 5. CPU Interface registers.

4.2 GIC Distributor

The Distributor in the GIC can handle 255 sources of interrupts. As indicated in Figure 4, Interrupt IDs in the range from 32 – 255 correspond to *shared peripheral interrupts* (SPIs). These interrupts are connected to the IRQ signals of up to 224 I/O peripherals, and these sources of interrupts are common to (shared by) both CPU Interfaces. The Distributor also handles *private peripherals interrupts* (PPIs) for each of the A9 processors, with these interrupts using IDs in the range from 0 – 31. The *software generated interrupts* (SGIs) are a special type of private interrupt that are generated by writing to a specific register in the GIC; Interrupt IDs from 0 – 15 are used for SGIs. We do not discuss SGIs further in this document.

API registers in the Distributor are depicted in Figure 6. As described in the previous section, addresses are shown for each register and these addresses correspond to those used in the DE1-SoC Computer. The Distributor Control Register (ICDDCR) is used to enable the Distributor. Setting $E = 0$ in this register disables the Distributor, while setting $E = 1$ enables it.

The *Interrupt Set Enable Registers* (ICDISERn) are used to enable the forwarding of each supported interrupt from the Distributor to the CPU Interface. The *n* postfix in the name ICDISERn means that multiple registers exist. Refer-

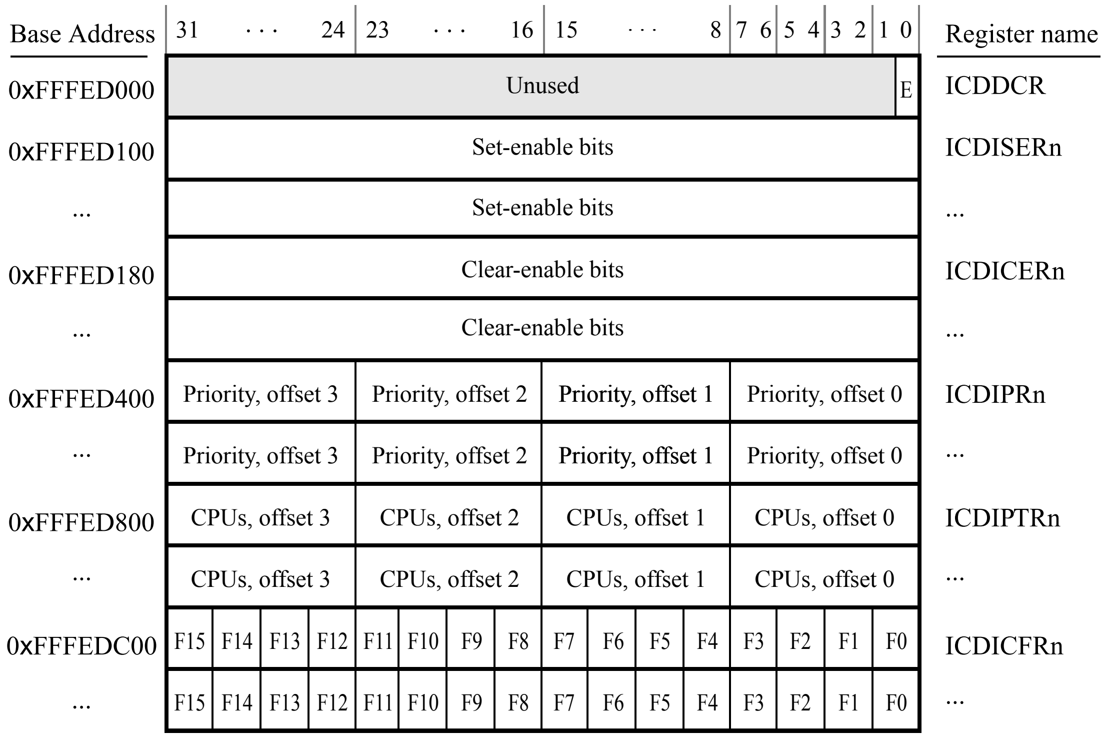


Figure 6. Distributor registers.

ring to Figure 6, the set-enable bits for the first 32 Interrupt IDs are provided in the register at address 0xFFFED100, the next 32 are provided in the register at the following word address, which is 0xFFFED104, and so on. Given a specific Interrupt ID, N, the address of the register that contains its set-enable bit is given by the integer calculation *address* = 0xFFFED100 + $(N \div 32) \times 4$, and the index of the bit inside this register is given by *index* = N mod 32. Writing the value 1 into a set-enable bit enables the forwarding of the corresponding IRQ to the CPU Interface.

In the same way that each supported interrupt can be enabled by using ICDISERn, each interrupt can be disabled by using the *Interrupt Clear Enable Registers* (ICDICERn). The method for calculating the address and index for ICDICERn is the same as that for ICDISERn, except that the base address is 0xFFFED180, as shown in Figure 6. Writing a 1 into a clear-enable bit disables the forwarding of the corresponding interrupt to the CPU Interface.

The *Interrupt Priority Registers* (ICDIPRn) are used to associate a priority level with each individual interrupt. On reset, these registers are set to 0x00000000, which represents the highest priority. In Figure 6 the base address of ICDIPRn is 0xFFFED400. Each Interrupt ID's priority field is one byte in size, which means that the register at the base address holds the priority levels for Interrupt IDs from 0 to 3. The priority levels for the next four Interrupt IDs use the register at address 0xFFFED404, and so on. Given a specific Interrupt ID, N, the address of the register that contains its priority field is given by the integer calculation *address* = 0xFFFED400 + $(N \div 4) \times 4$, and the index of the byte inside this register is given by *index* = N mod 4. Setting the priority field for an Interrupt ID to a larger number results in lower priority for the corresponding interrupt.

The *Interrupt Processor Targets Registers* (ICDIPTRn) are used to specify the CPU interfaces to which each interrupt should be forwarded. As indicated in Figure 6, the *CPUs* field for each Interrupt ID is one byte in size. This size is used because some versions of the ARM A9 MPCORE have up to eight A9 cores. A target CPU is selected by setting its corresponding bit field to 1. Thus, setting the byte at address 0xFFFED800 to the value 0x01 would target Interrupt ID 0 to CPU 0, setting this same byte to 0x02 would target CPU 1, and setting the byte to the value 0x03 would target both CPU 0 and CPU 1. The scheme for calculating the address of the ICDIPTRn register for a specific Interrupt ID, and also its byte index, is the same as the one shown above for ICDIPRn.

The *Interrupt Configuration Registers* (ICDICFRn) are used to specify whether each supported interrupt should be handled as level- or edge-sensitive by the GIC. As indicated in Figure 6, there is a two-bit field associated with each Interrupt ID. The least-significant bit in this field is not used. Setting the most-significant bit of this field to 1 makes the corresponding interrupt signal edge-sensitive, and setting this field to 0 makes it level-sensitive. When a level-sensitive IRQ signal is asserted by an I/O peripheral it is possible to de-assert this signal if the interrupt has not yet been forwarded from the Distributor to a CPU Interface. However, an edge-triggered IRQ signal cannot be de-asserted once it has been sampled in the Distributor. Referring to Figure 6, the first 16 Interrupt IDs use the ICDICFRn register at address 0xFFFEDC00, the next 16 at address 0xFFFEDC04, and so on. Given a specific Interrupt ID, N, the address of the ICDICFRn register is given by the integer calculation *address* = 0xFFFEDC00 + $(N \div 16) \times 4$, and the index of the bit inside this register is given by *index* = $(N \bmod 16) + 1$.

5 Example of Assembly Language Code

Figure 7 provides an example of an assembly language subroutine that configures the GIC. This code configures Interrupt ID 73, as an example, which corresponds to a parallel port connected to pushbutton KEYs in the DE1-SoC Computer. The code configures only some of the registers in the GIC and uses acceptable default values for other registers. A complete example of code that uses this subroutine is provided in the Appendix A.

```
/*
 * Configure the Generic Interrupt Controller (GIC)
 */
        .global CONFIG_GIC
CONFIG_GIC:
        PUSH     {LR}
        /* To configure the FPGA KEYS interrupt (ID 73):
         * 1. set the target to cpu0 in the ICDIPTRn register
         * 2. enable the interrupt in the ICDISERn register */
        /* CONFIG_INTERRUPT (int_ID (R0), CPU_target (R1)); */
        MOV      R0, #73             // KEY port (Interrupt ID = 73)
        MOV      R1, #1              // this field is a bit-mask; bit 0 targets cpu0
        BL       CONFIG_INTERRUPT

        /* configure the GIC CPU Interface */
        LDR      R0, =0xFFFEC100     // base address of CPU Interface
        /* Set Interrupt Priority Mask Register (ICCPMR) */
        LDR      R1, =0xFFFF         // enable interrupts of all priorities levels
        STR      R1, [R0, #0x04]
        /* Set the enable bit in the CPU Interface Control Register (ICCICR).
         * This allows interrupts to be forwarded to the CPU(s) */
        MOV      R1, #1
        STR      R1, [R0]

        /* Set the enable bit in the Distributor Control Register (ICDDCR).
         * This enables forwarding of interrupts to the CPU Interface(s) */
        LDR      R0, =0xFFFED000
        STR      R1, [R0]

        POP      {PC}
```

Figure 7. An example of assembly language code that configures the GIC (Part *a*).

```
/*
 * Configure registers in the GIC for an individual Interrupt ID
 * We configure only the Interrupt Set Enable Registers (ICDISERn) and
 * Interrupt Processor Target Registers (ICDIPTRn). The default (reset)
 * values are used for other registers in the GIC
 * Arguments: R0 = Interrupt ID (N), R1 = CPU target
 */
CONFIG_INTERRUPT:
        PUSH     {R4-R5, LR}

        /* Configure Interrupt Set-Enable Registers (ICDISERn).
         * reg_offset = (integer_div(N / 32) * 4
         * value = 1 << (N mod 32) */
        LSR      R4, R0, #3          // calculate reg_offset
        BIC      R4, R4, #3          // R4 = reg_offset
        LDR      R2, =0xFFFED100
        ADD      R4, R2, R4          // R4 = address of ICDISER

        AND      R2, R0, #0x1F       // N mod 32
        MOV      R5, #1              // enable
        LSL      R2, R5, R2          // R2 = value

        /* Using the register address in R4 and the value in R2 set the
         * correct bit in the GIC register */
        LDR      R3, [R4]            // read current register value
        ORR      R3, R3, R2          // set the enable bit
        STR      R3, [R4]            // store the new register value

        /* Configure Interrupt Processor Targets Register (ICDIPTRn)
         * reg_offset = integer_div(N / 4) * 4
         * index = N mod 4 */
        BIC      R4, R0, #3          // R4 = reg_offset
        LDR      R2, =0xFFFED800
        ADD      R4, R2, R4          // R4 = word address of ICDIPTR
        AND      R2, R0, #0x3        // N mod 4
        ADD      R4, R2, R4          // R4 = byte address in ICDIPTR

        /* Using register address in R4 and the value in R2 write to
         * (only) the appropriate byte */
        STRB     R1, [R4]

        POP      {R4-R5, PC}
```

Figure 7. An example of assembly language code that configures the GIC (Part *b*).

6 Example of C Code

Figure 8 provides an example of a subroutine written in C code that configures the GIC. This code performs the same operations as the assembly language code shown in Figure 7. A complete program that uses this subroutine is provided in the Appendix B.

```
/*
 * Configure the Generic Interrupt Controller (GIC)
*/
void config_GIC(void) {
    config_interrupt (73, 1); // configure the FPGA KEYs interrupt (73)

    // Set Interrupt Priority Mask Register (ICCPMR). Enable interrupts of all
    // priorities
    *((int *) 0xFFFEC104) = 0xFFFF;

    // Set CPU Interface Control Register (ICCICR). Enable signaling of
    // interrupts
    *((int *) 0xFFFEC100) = 1;

    // Configure the Distributor Control Register (ICDDCR) to send pending
    // interrupts to CPUs
    *((int *) 0xFFFED000) = 1;
}

/*
 * Configure Set Enable Registers (ICDISERn) and Interrupt Processor Target
 * Registers (ICDIPTRn). The default (reset) values are used for other registers
 * in the GIC.
*/
void config_interrupt(int N, int CPU_target) {
    int reg_offset, index, value, address;

    /* Configure the Interrupt Set-Enable Registers (ICDISERn).
     * reg_offset = (integer_div(N / 32) * 4
     * value = 1 << (N mod 32) */
    reg_offset = (N >> 3) & 0xFFFFFFFC;
    index      = N & 0x1F;
    value      = 0x1 << index;
    address    = 0xFFFED100 + reg_offset;
    /* Now that we know the register address and value, set the appropriate bit */
    *(int *)address |= value;
```

Figure 8. An example of C language code that configures the GIC (Part *a*).

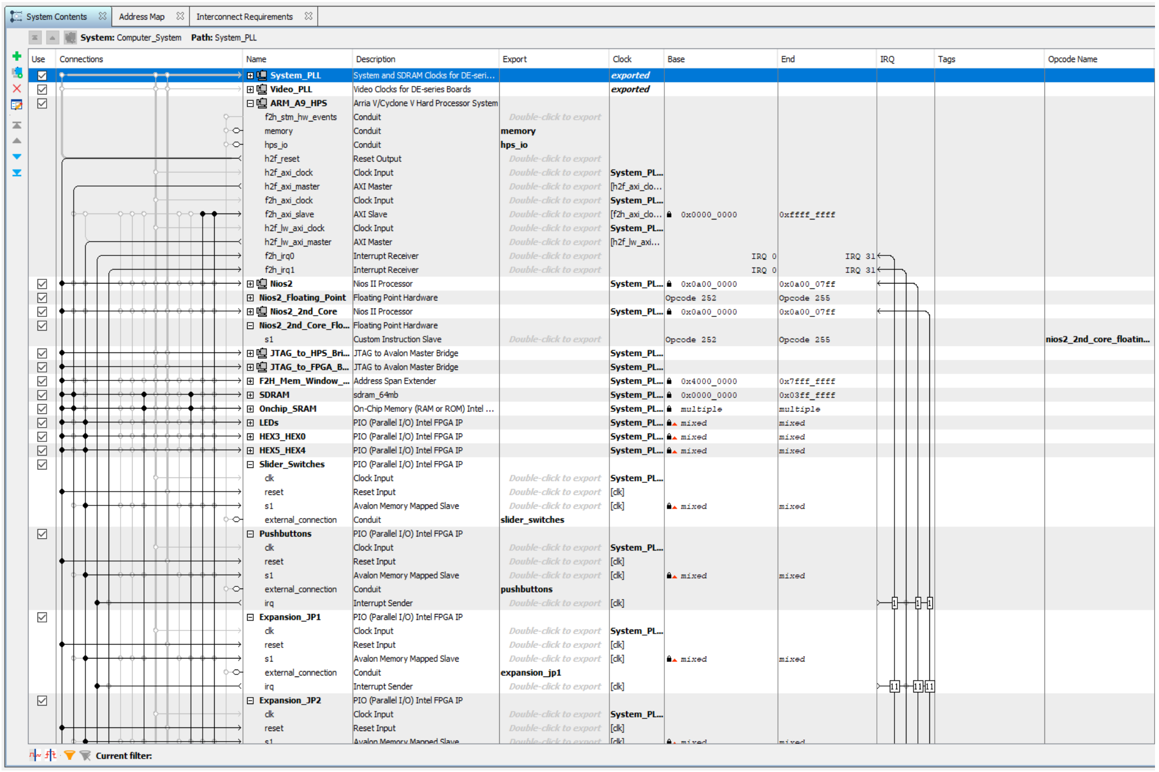
```
    /* Configure the Interrupt Processor Targets Register (ICDIPTRn)
     * reg_offset = integer_div(N / 4) * 4
     * index = N mod 4 */
    reg_offset = (N & 0xFFFFFFFC);
    index      = N & 0x3;
    address    = 0xFFFED800 + reg_offset + index;
    /* Now that we know the register address and value, write to (only) the
     * appropriate byte */
    *(char *)address = (char)CPU_target;
}
```

Figure 8. An example of C code that configures the GIC (Part *b*).

7 Example Platform Designer System

This section describes how to configure components capable of producing interrupts in a Platform Designer system. Only the parts of Platform Designer that are essential for interrupts are explained. Please refer to *Introduction to the Platform Designer* and *Making Platform Designer Components* available on the Intel FPGA University Program website for more details on using Platform Designer. The system used in this example is the *DE1-SoC Computer System*. Installing the Monitor Program allows you to easily view and modify this system in `C:\intelFPGA_lite\version\University_Program\Computer_Systems`. Please refer to *Monitor Program Tutorial for the ARM Processor* on the University Program website for more details on installing and using the Monitor Program with the ARM processor.

Peripherals that are a part of the HPS on DE-series boards are already connected to the ARM processor and do not require any additional configuration within Platform Designer to produce interrupts. Components on the FPGA side of the board must be configured within Platform Designer to produce interrupts. Figure 9 shows the DE1-SoC Computer system. The components of interest are the Cyclone V Hard Processor system and the input PIO peripherals such as the pushbuttons. Although this example is configuring the pushbuttons, it is possible to configure any IP in Platform Designer with an interrupt sender to produce interrupts with the same procedure.



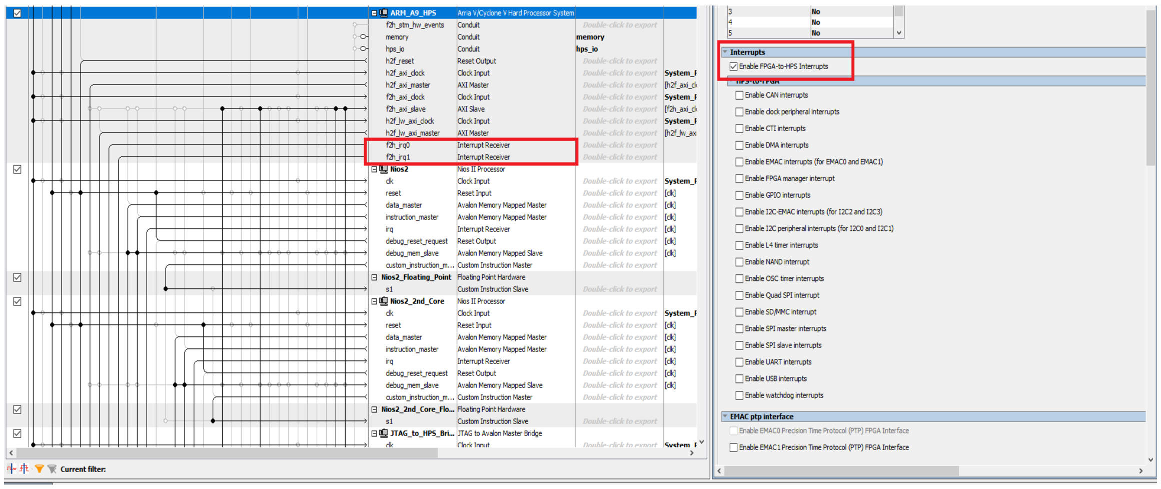
System: Computer_System **Path:** System_PLL

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
✓		⊞ System_PLL	System and SDRAM Clocks for DE-seri...		*exported*					
✓		⊞ Video_PLL	Video Clocks for DE-series Boards		*exported*					
✓		⊟ ARM_A9_HPS	Arria V/Cyclone V Hard Processor System							
		f2h_stm_hw_events	Conduit	Double-click to export						
		memory	Conduit	**memory**						
		hps_io	Conduit	**hps_io**						
		h2f_reset	Reset Output	Double-click to export						
		h2f_axi_clock	Clock Input	Double-click to export	System_PL...					
		h2f_axi_master	AXI Master	Double-click to export	[h2f_axi_clo...					
		f2h_axi_clock	Clock Input	Double-click to export	System_PL...					
		f2h_axi_slave	AXI Slave	Double-click to export	[f2h_axi_clo...	0x0000_0000	0xffff_ffff			
		h2f_lw_axi_clock	Clock Input	Double-click to export	System_PL...					
		h2f_lw_axi_master	AXI Master	Double-click to export	[h2f_lw_axi...					
		f2h_irq0	Interrupt Receiver	Double-click to export		IRQ 0	IRQ 31			
		f2h_irq1	Interrupt Receiver	Double-click to export		IRQ 0	IRQ 31			
✓		⊞ Nios2	Nios II Processor		System_PL...	0x0a00_0000	0x0a00_07ff			
✓		⊞ Nios2_Floating_Point	Floating Point Hardware			Opcode 252	Opcode 255			
✓		⊞ Nios2_2nd_Core	Nios II Processor		System_PL...	0x0a00_0000	0x0a00_07ff			
✓		⊟ Nios2_2nd_Core_Flo...	Floating Point Hardware							
		s1	Custom Instruction Slave	Double-click to export		Opcode 252	Opcode 255			nios2_2nd_core_floatin...
✓		⊞ JTAG_to_HPS_Bri...	JTAG to Avalon Master Bridge		System_PL...					
✓		⊞ JTAG_to_FPGA_B...	JTAG to Avalon Master Bridge		System_PL...					
✓		⊞ F2H_Mem_Window_...	Address Span Extender		System_PL...	0x4000_0000	0x7fff_ffff			
✓		⊞ SDRAM	sdram_64mb		System_PL...	0x0000_0000	0x03ff_ffff			
✓		⊞ Onchip_SRAM	On-Chip Memory (RAM or ROM) Intel ...		System_PL...	multiple	multiple			
✓		⊞ LEDs	PIO (Parallel I/O) Intel FPGA IP		System_PL...	mixed	mixed			
✓		⊞ HEX3_HEX0	PIO (Parallel I/O) Intel FPGA IP		System_PL...	mixed	mixed			
✓		⊞ HEX5_HEX4	PIO (Parallel I/O) Intel FPGA IP		System_PL...	mixed	mixed			
✓		⊟ Slider_Switches	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	System_PL...					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	mixed	mixed			
		external_connection	Conduit	**slider_switches**						
✓		⊟ Pushbuttons	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	System_PL...					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	mixed	mixed			
		external_connection	Conduit	**pushbuttons**						
		irq	Interrupt Sender	Double-click to export	[clk]			1 1 1		
✓		⊟ Expansion_JP1	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	System_PL...					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	mixed	mixed			
		external_connection	Conduit	**expansion_jp1**						
		irq	Interrupt Sender	Double-click to export	[clk]			11 11 11		
✓		⊟ Expansion_JP2	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	System_PL...					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	mixed	mixed			

Current filter:

Figure 9. The DE1-SoC Computer System.

The Hard Processor System component must be configured to accept interrupt requests from FPGA side components. Double click the component to open the settings for the HPS IP core. Scroll down to the Interrupts option and check the "Enable FPGA-to-HPS Interrupts" option. The HPS component should now have 1 or more f2h_irq lines depending on the number of cores in the HPS. The DE1-SoC ARM processor has two cores so there are 2 new lines available in the system. Figure 10 shows what the system should look like with the relevant settings selected.


```
⊟ ARM_A9_HPS                  Arria V/Cyclone V Hard Processor System
    f2h_stm_hw_events         Conduit                        Double-click to export
    memory                    Conduit                        memory
    hps_io                    Conduit                        hps_io
    h2f_reset                 Reset Output                   Double-click to export
    h2f_axi_clock             Clock Input                    Double-click to export   System_F
    h2f_axi_master            AXI Master                     Double-click to export   [h2f_axi_cl
    f2h_axi_clock             Clock Input                    Double-click to export   System_F
    f2h_axi_slave             AXI Slave                      Double-click to export   [f2h_axi_cl
    h2f_lw_axi_clock          Clock Input                    Double-click to export   System_F
    h2f_lw_axi_master         AXI Master                     Double-click to export   [h2f_lw_ax
    f2h_irq0                  Interrupt Receiver             Double-click to export
    f2h_irq1                  Interrupt Receiver             Double-click to export
⊟ Nios2                       Nios II Processor
    clk                       Clock Input                    Double-click to export   System_F
    reset                     Reset Input                    Double-click to export   [clk]
    data_master               Avalon Memory Mapped Master    Double-click to export   [clk]
    instruction_master        Avalon Memory Mapped Master    Double-click to export   [clk]
    irq                       Interrupt Receiver             Double-click to export   [clk]
    debug_reset_request       Reset Output                   Double-click to export   [clk]
    debug_mem_slave           Avalon Memory Mapped Slave     Double-click to export   [clk]
    custom_instruction_m...   Custom Instruction Master      Double-click to export
⊟ Nios2_Floating_Point        Floating Point Hardware
    s1                        Custom Instruction Slave       Double-click to export
⊟ Nios2_2nd_Core              Nios II Processor
    clk                       Clock Input                    Double-click to export   System_F
    reset                     Reset Input                    Double-click to export   [clk]
    data_master               Avalon Memory Mapped Master    Double-click to export   [clk]
    instruction_master        Avalon Memory Mapped Master    Double-click to export   [clk]
    irq                       Interrupt Receiver             Double-click to export   [clk]
    debug_reset_request       Reset Output                   Double-click to export   [clk]
    debug_mem_slave           Avalon Memory Mapped Slave     Double-click to export   [clk]
    custom_instruction_m...   Custom Instruction Master      Double-click to export
⊟ Nios2_2nd_Core_Flo...       Floating Point Hardware
    s1                        Custom Instruction Slave       Double-click to export
⊟ JTAG_to_HPS_Bri...          JTAG to Avalon Master Bridge
    clk                       Clock Input                    Double-click to export   System_F

Current filter:
```

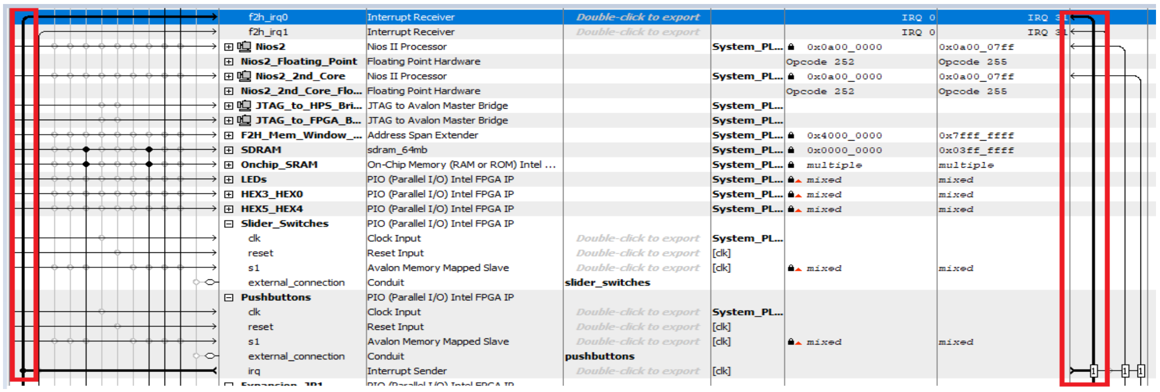
```
3    No
4    No
5    No

▾ Interrupts
  [✓] Enable FPGA-to-HPS Interrupts
  ▾ HPS-to-FPGA
    [ ] Enable CAN interrupts
    [ ] Enable clock peripheral interrupts
    [ ] Enable CTI interrupts
    [ ] Enable DMA interrupts
    [ ] Enable EMAC interrupts (for EMAC0 and EMAC1)
    [ ] Enable FPGA manager interrupt
    [ ] Enable GPIO interrupts
    [ ] Enable I2C-EMAC interrupts (for I2C2 and I2C3)
    [ ] Enable I2C peripheral interrupts (for I2C0 and I2C1)
    [ ] Enable L4 timer interrupts
    [ ] Enable NAND interrupt
    [ ] Enable OSC timer interrupts
    [ ] Enable Quad SPI interrupt
    [ ] Enable SD/MMC interrupt
    [ ] Enable SPI master interrupts
    [ ] Enable SPI slave interrupts
    [ ] Enable UART interrupts
    [ ] Enable USB interrupts
    [ ] Enable watchdog interrupts
▾ EMAC ptp interface
  [ ] Enable EMAC0 Precision Time Protocol (PTP) FPGA Interface
  [ ] Enable EMAC1 Precision Time Protocol (PTP) FPGA Interface
```

Figure 10. HPS IP core settings for enabling FPGA interrupts.

The components that will produce the interrupts must be connected to these new f2h_irq lines. For single threaded applications, connect all the interrupt senders to the f2h_irq0 line. f2h_irq0 line is connected to the first (default) core within the ARM processor. If you wish to use different cores to process different interrupts, you must connect the interrupt sender port to the appropriate f2h_irq line. Please note that the interrupt handler in software may behave differently depending on which core is executing the handler as each core has a seperate interrupt ID for each interrupt source that does not have to be the same for each core. In this example, we connect the interrupt sender ports from the pushbuttons to the interrupt reciever line f2h_irq0. Each interrupt sender must be assigned a unique Interrupt ID. This ID can be changed under the IRQ column in Platform Designer. In this example the pushbuttons are assigned an interrupt ID of 1. If your system has multiple IRQ lines, make sure that you are editing the interrupt ID coresponding to the correct processor core. Figure 11 shows the pushbutton IRQ port connected and the Interrupt ID of 1 assigned.

Now that the interrupts have been configured in hardware, you can proceed with writing interrupt service routines in software as shown in sections 5 and 6. There is one important detail regarding interrupt IDs to note. The interrupt ID assigned in Platform Designer is not the same number that you will assign a service routine to in software. This means that although we assigned an interrupt ID of 1 to the pushbuttons in Platform Designer, they will not produce that same interrupt ID in software. The interrupt ID that you should look for in software is *First_FPGA_Interrupt_ID* + *ID_in_Platform_Designer*. An example of this can be seen in Figure 8. The pushbuttons are configured to correspond to interrupt ID 73 even though we assigned the interrupt ID of 1 in Platform Designer. This is because the first FPGA interrupt on the DE1-SoC coresponds to 72 so the interrupt ID to look for in software is 72 + 1 =



f2h_irq0	Interrupt Receiver	Double-click to export		IRQ 0	IRQ 31	
f2h_irq1	Interrupt Receiver	Double-click to export		IRQ 0	IRQ 31	
⊞ Nios2	Nios II Processor		System_PL...	🔒 0x0a00_0000	0x0a00_07ff	
⊞ Nios2_Floating_Point	Floating Point Hardware			Opcode 252	Opcode 255	
⊞ Nios2_2nd_Core	Nios II Processor		System_PL...	🔒 0x0a00_0000	0x0a00_07ff	
⊞ Nios2_2nd_Core_Flo...	Floating Point Hardware			Opcode 252	Opcode 255	
⊞ JTAG_to_HPS_Bri...	JTAG to Avalon Master Bridge		System_PL...			
⊞ JTAG_to_FPGA_B...	JTAG to Avalon Master Bridge		System_PL...			
⊞ F2H_Mem_Window_...	Address Span Extender		System_PL...	🔒 0x4000_0000	0x7fff_ffff	
⊞ SDRAM	sdram_64mb		System_PL...	🔒 0x0000_0000	0x03ff_ffff	
⊞ Onchip_SRAM	On-Chip Memory (RAM or ROM) Intel ...		System_PL...	🔒 multiple	multiple	
⊞ LEDs	PIO (Parallel I/O) Intel FPGA IP		System_PL...	🔒▲ mixed	mixed	
⊞ HEX3_HEX0	PIO (Parallel I/O) Intel FPGA IP		System_PL...	🔒▲ mixed	mixed	
⊞ HEX5_HEX4	PIO (Parallel I/O) Intel FPGA IP		System_PL...	🔒▲ mixed	mixed	
⊟ Slider_Switches	PIO (Parallel I/O) Intel FPGA IP					
clk	Clock Input	Double-click to export	System_PL...			
reset	Reset Input	Double-click to export	[clk]			
s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	🔒▲ mixed	mixed	
external_connection	Conduit	slider_switches				
⊟ Pushbuttons	PIO (Parallel I/O) Intel FPGA IP					
clk	Clock Input	Double-click to export	System_PL...			
reset	Reset Input	Double-click to export	[clk]			
s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	🔒▲ mixed	mixed	
external_connection	Conduit	pushbuttons				
irq	Interrupt Sender	Double-click to export	[clk]			1
⊟ Expansion_JP1	PIO (Parallel I/O) Intel FPGA IP					

Figure 11. Pushbutton interrupts connected and assigned an interrupt ID of 1.

73. In the DE1-SoC Computer System, the interval timer is assigned an ID of 0 in Platform Designer. Therefore, we would look for interrupt 72 + 0 = 72 in software to determine if the interrupt came from the interval timer. You should read the manual for the board you are using to determine what the first FPGA interrupt ID is.

Appendix A: Example Assembly Language Program

```
/* ********************************************************************************
 * This program demonstrates use of interrupts with assembly language code.
 * The program responds to interrupts from the pushbutton KEY port in the FPGA.
 *
 * The interrupt service routine for the pushbutton KEYs indicates which KEY has
 * been pressed on the HEX0 display.
 ********************************************************************************/

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

        .text
        .global  _start
_start:
        /* Set up stack pointers for IRQ and SVC processor modes */
        MOV      R1, #0b11010010     // interrupts masked, MODE = IRQ
        MSR      CPSR_c, R1          // change to IRQ mode
        LDR      SP, =0xFFFFFFFF - 3 // set IRQ stack to A9 onchip memory
        /* Change to SVC (supervisor) mode with interrupts disabled */
        MOV      R1, #0b11010011     // interrupts masked, MODE = SVC
        MSR      CPSR, R1            // change to supervisor mode
        LDR      SP, =0x3FFFFFFF - 3 // set SVC stack to top of DDR3 memory

        BL       CONFIG_GIC          // configure the ARM GIC
        /* Write to the pushbutton KEY interrupt mask register */
        LDR      R0, =0xFF200050     // pushbutton KEY base address
        MOV      R1, #0xF            // set interrupt mask bits
        STR      R1, [R0, #0x8]      // interrupt mask register (base + 8)
        /* Enable IRQ interrupts in the processor */
        MOV      R0, #0b01010011     // IRQ unmasked, MODE = SVC
        MSR      CPSR_c, R0
IDLE:
        B        IDLE                // main program simply idles

/* Define the exception service routines */

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

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

/*--- Aborted data reads ------------------------------------------------------*/
SERVICE_ABT_DATA:
        B        SERVICE_ABT_DATA

/*--- Aborted instruction fetch -----------------------------------------------*/
SERVICE_ABT_INST:
        B        SERVICE_ABT_INST

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

        /* Read the ICCIAR from the CPU Interface */
        LDR      R4, =0xFFFEC100
        LDR      R5, [R4, #0x0C]      // read from ICCIAR

FPGA_IRQ1_HANDLER:
        CMP      R5, #73
UNEXPECTED:
        BNE      UNEXPECTED           // if not recognized, stop here

        BL       KEY_ISR
EXIT_IRQ:
        /* Write to the End of Interrupt Register (ICCEOIR) */
        STR      R5, [R4, #0x10]      // write to ICCEOIR

        POP      {R0-R7, LR}
        SUBS     PC, LR, #4
/*--- FIQ ---------------------------------------------------------------------*/
SERVICE_FIQ:
        B        SERVICE_FIQ

        .end

/*
 * Configure the Generic Interrupt Controller (GIC)
 */
        .global CONFIG_GIC
CONFIG_GIC:
        PUSH     {LR}
        /* To configure the FPGA KEYS interrupt (ID 73):
         * 1. set the target to cpu0 in the ICDIPTRn register
         * 2. enable the interrupt in the ICDISERn register */
        /* CONFIG_INTERRUPT (int_ID (R0), CPU_target (R1)); */
        MOV      R0, #73              // KEY port (Interrupt ID = 73)
        MOV      R1, #1               // this field is a bit-mask; bit 0 targets cpu0
        BL       CONFIG_INTERRUPT

        /* configure the GIC CPU Interface */
        LDR      R0, =0xFFFEC100     // base address of CPU Interface
        /* Set Interrupt Priority Mask Register (ICCPMR) */
        LDR      R1, =0xFFFF         // enable interrupts of all priorities levels
        STR      R1, [R0, #0x04]
        /* Set the enable bit in the CPU Interface Control Register (ICCICR).
         * This allows interrupts to be forwarded to the CPU(s) */
        MOV      R1, #1
        STR      R1, [R0]

        /* Set the enable bit in the Distributor Control Register (ICDDCR).
         * This enables forwarding of interrupts to the CPU Interface(s) */
        LDR      R0, =0xFFFED000
        STR      R1, [R0]

        POP      {PC}

/*
 * Configure registers in the GIC for an individual Interrupt ID
 * We configure only the Interrupt Set Enable Registers (ICDISERn) and
 * Interrupt Processor Target Registers (ICDIPTRn). The default (reset)
 * values are used for other registers in the GIC
 * Arguments: R0 = Interrupt ID (N), R1 = CPU target
 */
CONFIG_INTERRUPT:
        PUSH     {R4-R5, LR}

        /* Configure Interrupt Set-Enable Registers (ICDISERn).
         * reg_offset = (integer_div(N / 32) * 4
         * value = 1 << (N mod 32) */
        LSR      R4, R0, #3          // calculate reg_offset
        BIC      R4, R4, #3          // R4 = reg_offset
        LDR      R2, =0xFFFED100
        ADD      R4, R2, R4          // R4 = address of ICDISER

        AND      R2, R0, #0x1F       // N mod 32
        MOV      R5, #1              // enable
        LSL      R2, R5, R2          // R2 = value

        /* Using the register address in R4 and the value in R2 set the
         * correct bit in the GIC register */
        LDR      R3, [R4]            // read current register value
        ORR      R3, R3, R2          // set the enable bit
        STR      R3, [R4]            // store the new register value

        /* Configure Interrupt Processor Targets Register (ICDIPTRn)
         * reg_offset = integer_div(N / 4) * 4
         * index = N mod 4 */
        BIC      R4, R0, #3          // R4 = reg_offset
        LDR      R2, =0xFFFED800
        ADD      R4, R2, R4          // R4 = word address of ICDIPTR
        AND      R2, R0, #0x3        // N mod 4
        ADD      R4, R2, R4          // R4 = byte address in ICDIPTR

        /* Using register address in R4 and the value in R2 write to
         * (only) the appropriate byte */
        STRB     R1, [R4]

        POP      {R4-R5, PC}

/*************************************************************************
 * Pushbutton - Interrupt Service Routine
 *
 * This routine checks which KEY has been pressed. It writes to HEX0
 ************************************************************************/

        .global  KEY_ISR
KEY_ISR:
        LDR      R0, =0xFF200050 // base address of pushbutton KEY port
        LDR      R1, [R0, #0xC]  // read edge capture register
        MOV      R2, #0xF
        STR      R2, [R0, #0xC]  // clear the interrupt

        LDR      R0, =0xFF200020 // based address of HEX display
CHECK_KEY0:
        MOV      R3, #0x1
        ANDS     R3, R3, R1      // check for KEY0
        BEQ      CHECK_KEY1
        MOV      R2, #0b00111111
        STR      R2, [R0]        // display "0"
        B        END_KEY_ISR
CHECK_KEY1:
        MOV      R3, #0x2
        ANDS     R3, R3, R1      // check for KEY1
        BEQ      CHECK_KEY2
        MOV      R2, #0b00000110
        STR      R2, [R0]        // display "1"
        B        END_KEY_ISR
CHECK_KEY2:
        MOV      R3, #0x4
        ANDS     R3, R3, R1      // check for KEY2
        BEQ      IS_KEY3
        MOV      R2, #0b01011011
        STR      R2, [R0]        // display "2"
        B        END_KEY_ISR
IS_KEY3:
        MOV      R2, #0b01001111
        STR      R2, [R0]        // display "3"
END_KEY_ISR:
        BX       LR

        .end
```

Appendix B: Example C Program

```
void disable_A9_interrupts(void);
void set_A9_IRQ_stack(void);
void config_GIC(void);
void config_KEYs(void);
void enable_A9_interrupts(void);

/* ********************************************************************************
 * This program demonstrates use of interrupts with C code.  The program
 *responds
 * to interrupts from the pushbutton KEY port in the FPGA.
 *
 * The interrupt service routine for the KEYs indicates which KEY has been
 *pressed
 * on the LED display.
 ********************************************************************************/
int main(void) {
    disable_A9_interrupts(); // disable interrupts in the A9 processor
    set_A9_IRQ_stack();      // initialize the stack pointer for IRQ mode
    config_GIC();            // configure the general interrupt controller
    config_KEYs();           // configure pushbutton KEYs to generate interrupts

    enable_A9_interrupts(); // enable interrupts in the A9 processor

    while (1) // wait for an interrupt
        ;
}

/* setup the KEY interrupts in the FPGA */
void config_KEYs() {
    volatile int * KEY_ptr = (int *) 0xFF200050; // pushbutton KEY base address

    *(KEY_ptr + 2) = 0xF; // enable interrupts for the two KEYs
}

/* This file:
 * 1. defines exception vectors for the A9 processor
 * 2. provides code that sets the IRQ mode stack, and that dis/enables
 * interrupts
 * 3. provides code that initializes the generic interrupt controller
*/
void pushbutton_ISR(void);
void config_interrupt(int, int);

// Define the IRQ exception handler
void __attribute__((interrupt)) __cs3_isr_irq(void) {
    // Read the ICCIAR from the CPU Interface in the GIC
    int interrupt_ID = *((int *)0xFFFEC10C);

    if (interrupt_ID == 73)    // 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)
    *((int *)0xFFFEC110) = interrupt_ID;
}

// 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);
}

/*
 * Turn off interrupts in the ARM processor
*/
void disable_A9_interrupts(void) {
    int status = 0b11010011;
    asm("msr cpsr, %[ps]" : : [ps] "r"(status));
}

/*
 * Initialize the banked stack pointer register for IRQ mode
*/
void set_A9_IRQ_stack(void) {
    int stack, mode;
    stack = 0xFFFFFFFF - 7; // top of A9 onchip memory, aligned to 8 bytes
    /* change processor to IRQ mode with interrupts disabled */
    mode = 0b11010010;
    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 = 0b11010011;
    asm("msr cpsr, %[ps]" : : [ps] "r"(mode));
}

/*
 * Turn on interrupts in the ARM processor
*/
void enable_A9_interrupts(void) {
    int status = 0b01010011;
    asm("msr cpsr, %[ps]" : : [ps] "r"(status));
}

/*
 * Configure the Generic Interrupt Controller (GIC)
*/
void config_GIC(void) {
    config_interrupt (73, 1); // configure the FPGA KEYs interrupt (73)

    // Set Interrupt Priority Mask Register (ICCPMR). Enable interrupts of all
    // priorities
    *((int *) 0xFFFEC104) = 0xFFFF;

    // Set CPU Interface Control Register (ICCICR). Enable signaling of
    // interrupts
    *((int *) 0xFFFEC100) = 1;

    // Configure the Distributor Control Register (ICDDCR) to send pending
    // interrupts to CPUs
    *((int *) 0xFFFED000) = 1;
}

/*
 * Configure Set Enable Registers (ICDISERn) and Interrupt Processor Target
 * Registers (ICDIPTRn). The default (reset) values are used for other registers
 * in the GIC.
*/
void config_interrupt(int N, int CPU_target) {
    int reg_offset, index, value, address;

    /* Configure the Interrupt Set-Enable Registers (ICDISERn).
     * reg_offset = (integer_div(N / 32) * 4
     * value = 1 << (N mod 32) */
    reg_offset = (N >> 3) & 0xFFFFFFFC;
    index      = N & 0x1F;
    value      = 0x1 << index;
    address    = 0xFFFED100 + reg_offset;
    /* Now that we know the register address and value, set the appropriate bit */
    *(int *)address |= value;
    /* Configure the Interrupt Processor Targets Register (ICDIPTRn)
     * reg_offset = integer_div(N / 4) * 4
     * index = N mod 4 */
    reg_offset = (N & 0xFFFFFFFC);
    index      = N & 0x3;
    address    = 0xFFFED800 + reg_offset + index;
    /* Now that we know the register address and value, write to (only) the
     * appropriate byte */
    *(char *)address = (char)CPU_target;
}

/********************************************************************
* Pushbutton - Interrupt Service Routine
*
* This routine checks which KEY has been pressed. It writes to HEX0
*******************************************************************/
void pushbutton_ISR(void) {
    /* KEY base address */
    volatile int * KEY_ptr = (int *) 0xFF200050;
    /* HEX display base address */
    volatile int * HEX3_HEX0_ptr = (int *) 0xFF200020;
    int            press, HEX_bits;

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

    if (press & 0x1)                 // KEY0
        HEX_bits = 0b00111111;
    else if (press & 0x2)            // KEY1
        HEX_bits = 0b00000110;
    else if (press & 0x4)            // KEY2
        HEX_bits = 0b01011011;
    else                             // press & 0x8, which is KEY3
        HEX_bits   = 0b01001111;

    *HEX3_HEX0_ptr = HEX_bits;
    return;
}
```