(revised 3/28/2011)

A NOTE TO 259 STUDENTS:

Interrupts involve a lot of details.

The details presented after this page provide further background on exactly what happens at the CPU logic and assembly code levels. This may better help you understand the previous pages, as they define exactly how interrupts work. It may also help you understand how use interrupts in assembly code for your project. However, I will not test you on this material specifically – it is too detailed, and it is best left as a reference manual that you consult during a project (not during an examination).

I expect you to understand everything discussed prior to this page. In particular:

From 259library.c:

initInterrupts(); // clears history of all registered ISRs with IRQs, disables each specific device interrupt, enables CPU to receive interrupts

enableInterrupts(); // interrupts can be received by CPU from any specific device interrupt that is enabled
disableInterrupts(); // CPU ignores all interrupts

enableInterrupt(IRQ_NUM); // enables specific device interrupt to be received by CPU
disableInterrupt(IRQ_NUM); // CPU ignores specific device interrupt

registerISR(IRQ_NUM, ISR_name); // registers ISR_name with IRQ_NUM; interrupts received from IRQ_NUM will cause ISR_name() to be called for service

You need to know the purpose of these functions and how to use them.

You do not need to know the source code for these functions (although it is provided in 259library.c, and reading it may help you understand things better).

From example code (irq-example.c, irq-example2.c):

enableCounterIRQ(delay_amount, counterISR);
enableKeyIRQ(keymask, keyISR);

You need to know the purpose of these functions and how to use them.

You need to be familiar with the internal details of these functions, but you do not need to memorize the internal details. If needed, I will provide (most of the) internal details on a test, but may leave blank sections. You may be asked to explain, alter, or fill in the internal details.

I cannot guarantee that quiz/exam question(s) on interrupts will be exactly like the Lecture Quiz.

Nios II CPU Interrupt Details

L35-1

The Nios II CPU actually has 32 interrupt pins. These can be wired up in many ways in a computer system; the precise wiring assignment used for the UBC DE1 Media computer was already given earlier. These *hardware interrupt request pins* are inputs to the CPU, and appear internally in the CPU as irq0, irq1, …, irq31 as shown in Figure 4.

A 32-bit CPU register called *ienable* is used to mask interrupts. This value is ANDed with the set of 32 interrupt input pins. Therefore, individual interrupts with the corresponding *ienable* bit set to 0 are ignored and not be seen by the CPU. Setting *ienable* bits to 1 allows the CPU to receive interrupts from the corresponding device.

The current status of all interrupts is stored in another 32-bit CPU register called *ipending*. Reading this register allows a program to determine which specific interrupt has occurred. For bit N in *ipending* to be set, the corresponding Nth bit in *ienable* must be 1, and the corresponding irqN pin must also be set to 1.

After *ipending*, a wide OR gate combines all of the 32 *ipending* bits. This generates a single (combined) interrupt-request which is sent to the CPU. If any bit in *ipending* is a 1, then an interrupt-request will be sent to the CPU.

Finally, the CPU can choose to ignore the interrupt-request bit using single AND gate. This bit, called the PIE bit, is located in bit position 0 of another 32-bit CPU register called *status*. All interrupts (and exceptions) can be masked or disabled if this bit is a 0. If the PIE bit is a 1, *and* the interrupt-request signal is a 1, then a CPU interrupt is produced.

Individual bits in the ienable register are modified by:
disableInterrupt(irq_num);
enableInterrupt(irq_num);

All bits in the ienable reg. are cleared by:
initInterrupts();

The PIE bit can be modified by:
disableInterrupts();
enableInterrupts();
initInterrupts();

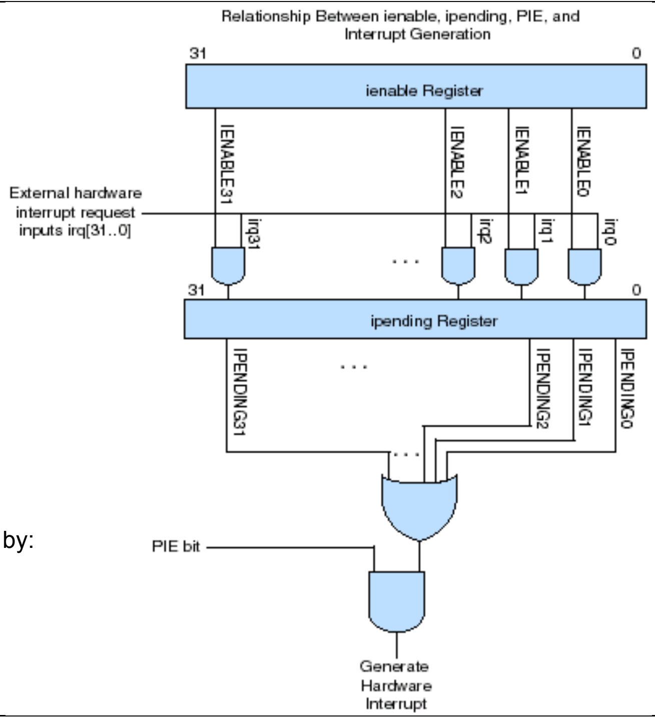
Relationship Between ienable, ipending, PIE, and Interrupt Generation
31
0
ienable Register
IENABLE31
IENABLE2
IENABLE1
IENABLE0
External hardware interrupt request inputs irq[31..0]
irq31
irq2
irq1
irq0
31
0
ipending Register
IPENDING31
IPENDING2
IPENDING1
IPENDING0
PIE bit
Generate Hardware Interrupt

Figure 4. Hardware interrupt structure inside the Nios II CPU (from Fig 3-2 in [1])

Special Nios II Registers

You already know about the 32 **general-purpose registers**, r0 to r31 listed in Figure 5. Notice the special names given to some registers, in particular:

- *r24* or *et*, exception temporary – **do not use this register in your main program**!
- *r29* or *ea*, exception return address

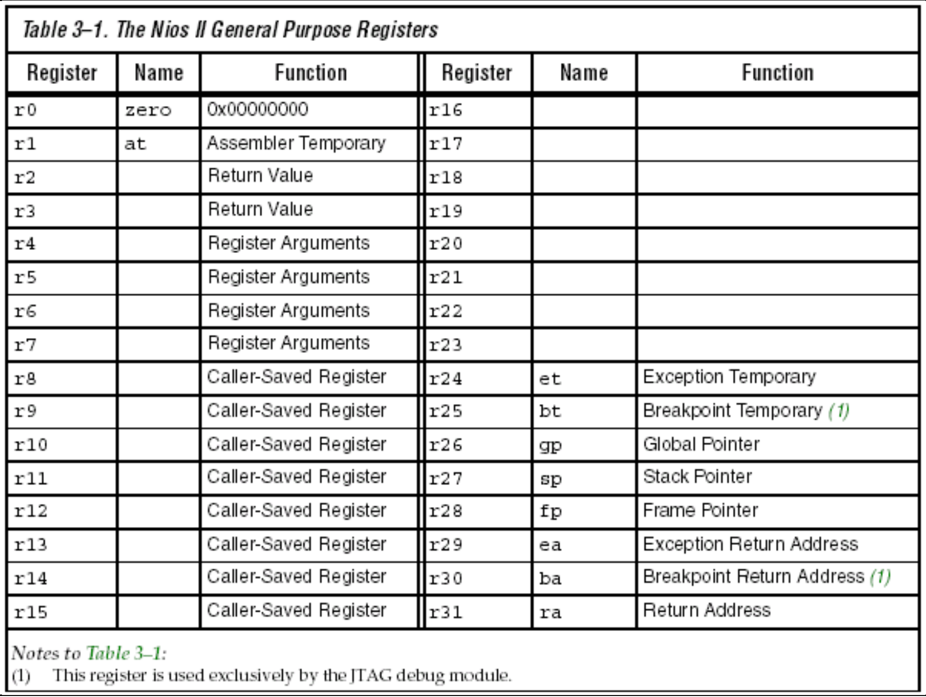

Table 3–1. The Nios II General Purpose Registers

Register	Name	Function	Register	Name	Function
r0	zero	0x00000000	r16		
r1	at	Assembler Temporary	r17		
r2		Return Value	r18		
r3		Return Value	r19		
r4		Register Arguments	r20		
r5		Register Arguments	r21		
r6		Register Arguments	r22		
r7		Register Arguments	r23		
r8		Caller-Saved Register	r24	et	Exception Temporary
r9		Caller-Saved Register	r25	bt	Breakpoint Temporary (1)
r10		Caller-Saved Register	r26	gp	Global Pointer
r11		Caller-Saved Register	r27	sp	Stack Pointer
r12		Caller-Saved Register	r28	fp	Frame Pointer
r13		Caller-Saved Register	r29	ea	Exception Return Address
r14		Caller-Saved Register	r30	ba	Breakpoint Return Address (1)
r15		Caller-Saved Register	r31	ra	Return Address

Notes to Table 3–1:
(1) This register is used exclusively by the JTAG debug module.

Figure 5. General-purpose Nios II registers (from Table 3-5 in [1]).

In addition, there are up to 32 **special control registers**, *ctl0* to *ctl31*, listed in Figure 6. These registers have names corresponding to their specialized purpose. We will use:

ctl0 or *status* *ctl3* or *ienable*
ctl1 or *estatus* *ctl4* or *ipending*

These are accessed using *only* the specialized instructions **rdctl** and **wrctl**. For example:

- rdctl r24, ctl1 or rdctl et, estatus copy *estatus* value to r24
- rdctl r8, ctl4 or rdctl r8, ipendingcopy *ipending* value to r8
- wrctl ctl0, r0 or wrctl status, r0 copy r0 to *status* (disables interrupts)
- wrctl ctl3, r13 or wrctl ienable, r13 copy r13 to *ienable*

Register	Name	Description
ctl0	*status*	Bit 0 is the PIE (processor interrupt enable) bit
ctl1	*estatus*	Holds a copy of *status* after exception handler is called
ctl2	*bstatus*	Used by debugger
ctl3	*ienable*	Interrupt enable bits
ctl4	*ipending*	Interrupt pending bits
ctl5	*CPUid*	Unique processor identifier
ctl6 – ctl31	…etc…	Some of *ctl7* to *ctl15* are defined, rest are reserved

Figure 6. Special-purpose Nios II control registers (from Table 3-6 in [1]).

Interrupts and Exceptions

Interrupts are generated by hardware devices. However, software can also generate an interrupt-like event called an exception. Exceptions are a generalization of interrupts:

- *Hardware interrupts* are generated by devices
- *Software exceptions* are generated by software instructions

Exceptions refer to both *hardware interrupts* and *software-generated exceptions*.

Just like an interrupt, an *exception* event can occur at *any time*. When it does occur, the CPU will momentarily stop running your program, go and run another short program called an **exception handler** to completion, and then return to your program. You can think of an **exception handler** as a special type of subroutine that is not called directly by your program – the CPU decides when to call it by examining specific *hardware events* that occur in a computer system (interrupts) or in the CPU (software exceptions). **It is the exception handler that decides which hardware interrupt service routine to call.**

Exceptions can also be triggered when the processor tries to execute specific instructions.

In the Nios II CPU, some complicated instructions like integer multiply or integer divide can be left out of the hardware when the system is generated. If the CPU encounters one of these unimplemented instructions, it triggers an *unimplemented instruction exception*. This allows software to emulate the missing instruction by replacing it with a short software routine that achieves the same result (eg, it executes a multiply instruction by emulating it with a software exception because the hardware multiplier was omitted from the CPU).

Nios II can also trigger exceptions on data-dependant events with certain instructions. Common data-dependant exceptions are when an instruction attempts to *divide by 0*, or when load or store *word* or *halfword* is given an *unaligned address* which is not a multiple of 4 or 2, respectively, or when attempting to load or store to a memory address that does not exist in the computer system. In most of these situations, the Nios II CPU can be configured to behave in a variety of ways: it can generate exceptions, or it can execute the bad instruction but produce an undefined result. An undefined result means that the CPU result or operation is not reliable or predictable.

Not all exceptions are bad. In complex CPUs, the *virtual memory system* will often trigger an exception when you try to access a memory region that was swapped to disk. The exception forces the *operating system* to run momentarily, which fetches the data from disk and places it in memory, and then resumes your program. This activity is known as demand paging, and it is quite common in full systems (like Windows).

Examples of Nios II Exceptions and Interrupts

The Nios II has many different exception types. Here are a few of the common ones:

- Interrupt exception (highest priority)
- Trap exception
- Illegal instruction exception
- Unimplemented instruction exception
- Break exception
- Misaligned data address exception
- Division error exception (lowest priority)

Interrupts are often treated as a *hardware exception*, while the others are considered *software exceptions* because they are usually triggered by specific instructions.

The *trap exception* is generated whenever the special CPU instruction named *trap* is executed. This allows the user program to "call" the operating system. In our case, there is no operating system, so we do not use this instruction.

The *illegal instruction exception* is generated whenever the CPU encounters an instruction (a 32-bit pattern) that is unknown.

The *unimplemented instruction exception* is generated whenever a legal instruction, such as integer multiply or divide, is encountered that is not implemented in CPU hardware. These instructions must be emulated using software.

The *break exception* is used exclusively by the debugger. There is even a specialized CPU instruction named *break*. We do not use this exception or this instruction in our programs, as it will interfere with debugger operation.

The *misaligned data address exception* is generated when you use ldw, stw, ldwio, or stwio with an address that is not a multiple of 4. It is also generated when you use halfword load and store instructions with an odd address.

The *division error exception* is generated when a program attempts to divide by zero, or it attempts to signed division between the largest negative number (-2147483648) and -1, producing a result that is out of range.

The Nios II is a flexible CPU. The computer system designer has the choice of omitting many of these exceptions from the CPU. In the UBC DE1 Media computer, only the underlined events listed above will generate an exception.

Detailed Nios II Exception Process

When an exception is triggered, the CPU does the following steps *automatically*:

1. Copy the contents of *status* to *estatus* to save pre-exception state
2. Clear (0) PIE bit of *status* to ensure further exceptions are now disabled
3. Modify *r29* aka *ea* to hold the return address of the instruction *immediately after the one being interrupted*
4. Start running the *exception handler* program at the predefined address (0x00000020)

When the exception ends, the exception handler must use the special *eret* instruction to automatically and properly end the exception process:

5. Copy *estatus* back to *status* to restore the pre-exception state
6. Return to running the regular program at the address stored in *ea*

Your Program

To use interrupts and exceptions, your program must include the following:

A. An *exception handler*
B. An *interrupt service routine* for each interrupt source you enable
C. A *setup routine* to initialize the entire interrupts subsystem

A) Your *exception handler* must:

1. Save registers on the stack
2. Determine the cause of the exception according to the priority order
3. For hardware interrupts, adjust the return address in *ea* by subtracting 4
4. Call the appropriate *interrupt service routine* or *exception service routine*
    - Loop to call ISR associated for each hardware IRQ in *ipending*
5. Restore registers from the stack
6. Return to the main program using the instruction *eret*

B) Your *interrupt service routine* must:

1. Clear the cause of the exception or interrupt so it will not occur again (eg, tell the device to stop sending the interrupt)
2. Do the appropriate action for the interrupt (eg, read the character received from the serial port)
3. Change the state of the system (ie, modify memory to alter behaviour of system)
4. Return to the exception handler using *ret*

C) Your *main program* or *setup routine* must:

1. Place the *exception handler* in memory at address 0x00000020.
2. Enable the use of the stack
3. Specifically enable device to send interrupts (eg: ps2, timer)
4. Specifically enable CPU to receive interrupts from the device (*ienable*)
5. Enable CPU interrupts by setting PIE bit to 1 (i.e. set bit 0 in *status* to a 1)

Countdown Timer Device

The COUNTER device you have used to measure time so far always counts up by 1 every clock cycle, and it uses *irq3*. There is another device, the TIMER, which allows you to count down. It is also more complex than the COUNTER device, allowing you START and STOP the timer – a useful feature.

When the countdown TIMER reaches 0, it can generate an interrupt on *irq0* and on the next cycle reload itself with a new starting value. You can set the starting value to any number you choose. This results in very predictable, periodic interrupts. This is great for controlling time-sensitive devices without polling!

Programming the TIMER is a little bit difficult because it uses a 16-bit interface for 32-bit values as shown in Figure 5. The base address for the timer is ADDR_TIMER. The names of individual registers such as TIMER_STATUS and TIMER_VALUE_LOW shown below are all offsets relative to IOBASE.

You can read or write halfwords or words to the individual register addresses, but only the lower 16 bits have meaning with word accesses. The individual bits of each address have different meanings and names as follows:

- TO, or bit 0 of TIMER_STATUS, provides a timeout signal which is set to 1 by the timer when it has reached a count value of 0. The TO bit can be cleared by writing a 0 to it.
- RUN, or bit 1 of TIMER_STATUS, is set to 1 by the timer whenever it is currently counting. Writing to TIMER_STATUS does not affect the value of the RUN bit.
- ITO, or bit 0 of TIMER_CONTROL, enables the device to *send* interrupts to the processor whenever TO becomes 1. These will be received by the processor on *irq0*. **Note:** to clear interrupts sent by the timer, write a 0 to the TO bit above.
- CONT, or bit 1 of TIMER_CONTROL, controls whether the timer stops after counting down to 0 (CONT=0) or continues by reloading (CONT=1).
- START and STOP, or bits 2 and 3 of TIMER_CONTROL, can be used to commence/suspend the operation of the timer by writing a 1 to the respective bit.

The values written to TIMER_START_LOW and TIMER_START_HIGH allow the period of the time to be changed. This is the value that is reloaded into the counter after it tries to count below 0 (when CONT=1). Since the registers are only 16 bits, you must break up a 32-bit value (such as 100,000 for 2 milliseconds) into the high and low parts with shift and AND instructions and write them separately.

It is possible to capture a snapshot of the counter value at any time by performing a write (store instruction) to the TIMER_VALUE_LOW address. After the snapshot, you can read out the value by reading both TIMER_VALUE_LOW and TIMER_VALUE_HIGH and building the corresponding 32-bit value with shift and OR instructions.

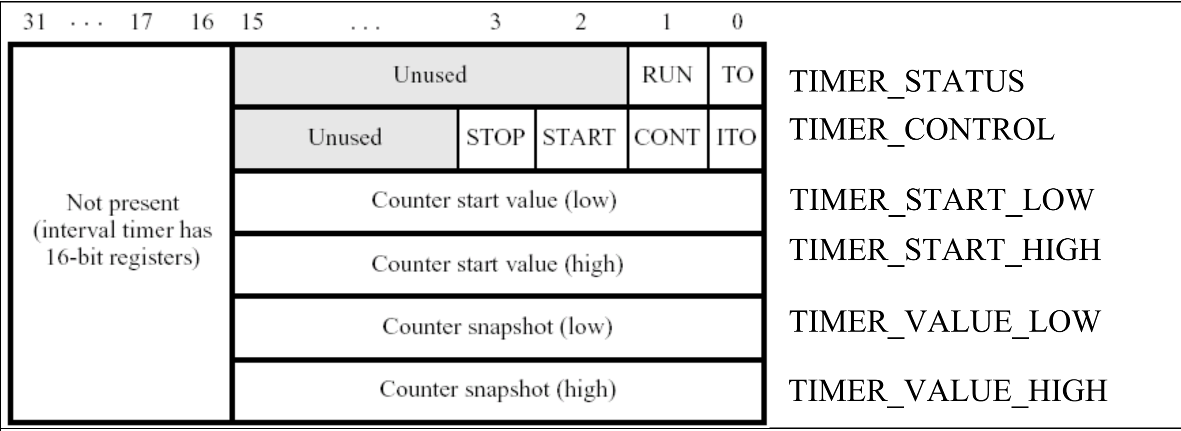

Figure 5. Countdown timer device registers.

Interrupt Example – Assembly Language using Countdown Timer

The assembly language example on the next 2 pages shows how to configure and use timer interrupts every 100ms.

Each interrupt is counted by incrementing the memory location *interrupt_counts* and displaying the count on LEDG. The main program continuously copies SWITCH to LEDR, acting as a wire. Notice the main program and interrupt service routine both communicate using r8 without any problems.

Example: Count every 100ms on LEDG; no communication with main program.

```
.include "ubc-de1media-macros.s"

/****************************************************************************
 * RESET SECTION
 * The Nios II assembler/linker places this section at address 0x00000000.
 * It must be <= 8 real NiosII instructions. This is where the CPU starts
 * at "powerup" and on "reset".
 */
.section .reset, "ax"
               movia   sp, STACK_END          /* initialize stack */
               movia   ra, _start
               ret                            /* jump to _start */

/****************************************************************************
 * EXCEPTIONS SECTION
 * The Nios II assembler/linker places this section at addresss 0x00000020.
 */
.section .exceptions, "ax"

exception_handler:
               addi    sp, sp, -12                    /* save used regs on stack */
               stw     r8, 0(sp)
               stw     r9, 4(sp)
               stw     ra, 8(sp)

               /* Check if interrupts were enabled by examining the EPIE bit. */
               /* EPIE is bit0 of estatus, a copy of PIE before the exception */
               rdctl   et, estatus
               andi    et, et, 1
               beq     et, r0, check_software_exceptions
               /* interrupts are enabled, check if any are pending */
               rdctl   et, ipending
               beq     et, r0, check_software_exceptions

check_hardware_interrupts:
               /* upon return, execute the interrupted instruction */
               subi    ea, ea, 4
               /* should check interrupts one-at-a-time, from irq0 to irq31 */
               /* each time the ipending bit is set, we should call the proper ISR */
               /* since we are only expecting irq0, we will only check for it */
               andi    et, et, 0x1
               beq     et, r0, check_next_interrupt

               call    timer_isr              /* ISR uses r8, r9, and 'call' uses ra */

check_next_interrupt:
       /* no more interrupts to check */

check_software_exceptions:
       /* no software exceptions supported */
       /* they should be checked in priority order (trap, break, unimplemented) */

done_exceptions:
               ldw     ra, 8(sp)                      /* restore used regs from stack */
               ldw     r9, 4(sp)
               ldw     r8, 0(sp)
               addi    sp, sp, 12
               eret
/* Nios II exception priorities are defined as follows:
 *      1) hardware interrupt exceptions
 *             a) irq 0 (highest interrupt priority)
 *             b) irq 1, ..., irq30 (again, listed higher to lower priority)
 *             c) irq 31 (lowest interrupt priority)
 *      2) software exceptions
 *             a) trap exception
 *             b) break exception
 *             c) unimplemented instruction
 * We implement these priorities by checking the cause of the exception
 * in the same order in the exception handler above.
 */
```

```
/******************************************************************************
 * TEXT SECTION
 * The Nios II assembler/linker should put the .text section after the .exceptions.
 * You may need to configure the Altera Monitor Program to locate it at address 0x400.
 */

.text
.global _start
```

```
_start:
                movia   r23, IOBASE
                movia   sp, STACK_END                 /* make sure stack is initialized */

                movia   r4, interrupt_counts
                stw     r0, 0(r4)

                movia   r4, 100*50000  /* # of timer cycles in 100ms */
                call    setup_timer_interrupts
                call    setup_cpu_interrupts

loop:           ldwio   r8, SWITCH(r23)
                stwio   r8, LEDR(r23)
                br      loop
```

```
timer_isr:
        /* every interval, increment 'interrupt_counts' and display on LEDG */
                /* clear source of interrupt by writing 0 to TO bit */
                stwio   r0, TIMER_STATUS(r23)

                /* process the interrupt, change state of system */
                movia   r9, interrupt_counts
                ldw     r8, 0(r9)
                addi    r8, r8, 1
                stw     r8, 0(r9)
                stwio   r8, LEDG(r23)          /* show count on LEDG */

                /* return from ISR */
                ret
```

```
setup_timer_interrupts:
        /* set up timer to send interrupts */
        /* parameter r4 holds the # cycles for the timer interval */

                /* set the timer period */
                andi            r2, r4, 0xffff        /* extract low halfword */
                stwio           r2, TIMER_START_LOW(r23)
                srli            r2, r4, 16            /* extract high halfword */
                stwio           r2, TIMER_START_HIGH(r23)

                /* start timer (bit2), count continuously (bit1), enable irq (bit0) */
                movi            r2, 0b0111
                stwio           r2, TIMER_CONTROL(r23)

                ret
```

```
setup_cpu_interrupts:
        /* set up CPU to receive interrupts from timer */
                movi            r2, 0x01       /* bit0 = irq0 = countdown timer device */
                wrctl           ienable, r2
                movi            r2, 1          /* bit0 = PIE */
                wrctl           status, r2
                ret                            /* first instr. that may be interrupted */
```

```
.data
interrupt_counts:
.word 0
```

```
.end
```

Communication between ISR and Regular Program

One of the most difficult things to get correct is the communication between your ISR and the regular part of your program. There are two basic methods of communicating:

1) By modifying specific registers
2) By modifying memory

Using registers is easiest, but it is really a shortcut and there are still some hidden difficulties. Using memory is the proper method, and the only way available from C language. However, it was already discussed earlier and will not be repeated here.

To use registers for communication, you must use assembly language. You must decide ahead of time which specific register will be dedicated for the communication. **The communicating register <u>must not</u> be saved/restored by the exception handler.**

In the second example assembly program (ABS brake controller), the program counts how many times the wheel spins (KEY3 goes from 0 to 1) by incrementing r22. The ISR inspects r22 to see if it should apply the brakes (frequent spins) or pulse them (infrequent spinning indicates a locked wheel). When the ISR exits, it resets r22 to 0. This communication is safe because all instructions that modify r22 behave atomically (the modifying instruction either completely executes, or is interrupted before executing).

It is also important that we dedicated r22 to the task, and not a register that sometimes has another purpose. For example, suppose we chose to use r2 – after all, it is often used by subroutines to return a value. If the main program contains subroutines, they would also use r2 to return some value. Usually, the subroutine would return the correct value. However, sometimes the ISR will interrupt the subroutine just before returning; the ISR resets r2 to 0, causing the subroutine to return the wrong value.

If you communicate using registers (not memory), it is still necessary to protect critical sections. In the example below, a shared variable is placed in register r22. The main program has a problem because it reads r22 in one place (the blt instruction) and then modifies it in another (the addi or movi instructions). Using separate instructions to read and modify the register is what causes the problem; protect this by disabling interrupts before the read, and re-enabling interrupts after the modify.

```
MainBuggy:      blt     r22, r8, resetR22
                addi    r22, r22, 1
                br      done
resetR22:       movi    r22, 0
done:           ...
```

Identifying and protecting critical sections correctly is a difficult task. You will spend a lot of time on this topic next year in your Operating Systems course (eg, EECE 314 / 315).

References

[1] Nios II Processor Reference Handbook, especially early pages in Chapter 3.
[2] Nios II Software Developer's Handbook, especially Chapter 8.
[3] Altera DE1 Media Computer manual

You can download [1] and [2] from http://www.altera.com in the *Literature* section.

Example: ABS Brake Controller – toggle LEDG0 if KEY3 is infrequent. ISR and main program communicate using register r22.

(revised 3/28/2011)

```
.include "ubc-de1media-macros.s"

/*****************************************************************************
 * RESET SECTION
 * The Nios II assembler/linker places this section at address 0x00000000.
 * It must be <= 8 real NiosII instructions. This is where the CPU starts
 * at "powerup" and on "reset".
 */
.section .reset, "ax"
               movia   sp, STACK_END          /* initialize stack */
               movia   ra, _start
               ret                            /* jump to _start */

/*****************************************************************************
 * EXCEPTIONS SECTION
 * The Nios II assembler/linker places this section at addresss 0x00000020.
 */
.section .exceptions, "ax"

exception_handler:
               addi    sp, sp, -12                    /* save used regs on stack */
               stw     r8, 0(sp)
               stw     r9, 4(sp)
               stw     ra, 8(sp)

               /* Check if interrupts were enabled by examining the EPIE bit. */
               /* EPIE is bit0 of estatus, a copy of PIE before the exception */
               rdctl   et, estatus
               andi    et, et, 1
               beq     et, r0, check_software_exceptions
               /* interrupts are enabled, check if any are pending */
               rdctl   et, ipending
               beq     et, r0, check_software_exceptions

check_hardware_interrupts:
               /* upon return, execute the interrupted instruction */
               subi    ea, ea, 4
               /* should check interrupts one-at-a-time, from irq0 to irq31 */
               /* each time the ipending bit is set, we should call the proper ISR */
               /* since we are only expecting irq0, we will only check for it */
               andi    et, et, 0x01
               beq     et, r0, check_next_interrupt

               call    timer_isr              /* ISR uses r8, r9, and 'call' uses ra */

check_next_interrupt:
       /* no more interrupts to check */

check_software_exceptions:
       /* no software exceptions supported */
       /* they should be checked in priority order (trap, break, unimplemented) */

done_exceptions:
               ldw     ra, 8(sp)                      /* restore used regs from stack */
               ldw     r9, 4(sp)
               ldw     r8, 0(sp)
               addi    sp, sp, 12
               eret


/*****************************************************************************
 * TEXT SECTION
 * The Nios II assembler/linker should put the .text section after the .exceptions.
 * You may need to configure the Altera Monitor Program to locate it at address 0x400.
 */

.text
.global _start

_start:        movia   r23, IOBASE
               movia   sp, STACK_END          /* make sure stack is initialized */
               movia  r4, brake_flag
               stw    r0, 0(r4)             /* initially, turn brake OFF */
               movi   r22, 0                /* initialize KEY3 counter = 0 */

               movia  r4, 100*50000         /* # of timer cycles in 100ms */
               call   setup_timer_interrupts
               call   setup_cpu_interrupts

loop:          stwio  r22, LEDR(r23)        /* display current KEY3 counter */
               ldwio  r16, KEY(r23)
               andi   r16, r16, 8 /* KEY3 */
               bne    r16, r0, loop         /* wait for KEY3 to become 0 */

while0:        stwio  r22, LEDR(r23)        /* display current KEY3 counter */
               ldwio  r16, KEY(r23)
               andi   r16, r16, 8 /* KEY3 */
               beq    r16, r0, while0       /* wait for KEY3 to become 1 */

               /* count the 0-to-1 transition */
               addi   r22, r22, 1
               br     loop

timer_isr:
       /* every 100ms, adjust brake_flag and display it on LEDG */
               /* clear source of interrupt by writing 0 to TO bit */
               stwio  r0, TIMER_STATUS(r23)
               /* process the interrupt */
               movia  r8, brake_flag        /* read old brake state */
               ldw    r9, 0(r8)

               movi   r8, 5
               blt    r22, r8, brakePULSE   /* if KEY3 pressed < 5 times, pulse brake */
brakeON:       movi   r9, 0                 /* turn brake off (invert turns it ON) */
brakePULSE:    xori   r9, r9, 1             /* invert state of brake to pulse it */

               /* change state of the system */
               movia  r8, brake_flag        /* remember new brake state */
               stw    r9, 0(r8)
               stwio  r9, LEDG(r23)         /* show current brake signal to LEDG[0] */
               mov    r22, r0               /* reset KEY3 counter every second */
               /* return from ISR */
               ret

setup_timer_interrupts:
       /* set up timer to send interrupts */
       /* parameter r4 holds the # cycles for the timer interval */

               /* set the timer period */
               andi          r2, r4, 0xffff        /* extract low halfword */
               stwio         r2, TIMER_START_LOW(r23)
               srli          r2, r4, 16            /* extract high halfword */
               stwio         r2, TIMER_START_HIGH(r23)

               /* start timer (bit2), count continuously (bit1), enable irq (bit0) */
               movi          r2, 0b0111
               stwio         r2, TIMER_CONTROL(r23)
               ret

setup_cpu_interrupts:
       /* set up CPU to receive interrupts from timer */
               movi          r2, 0x01      /* bit0 = irq0 = countdown timer device */
               wrctl         ienable, r2
               movi          r2, 1         /* bit0 = PIE */
               wrctl         status, r2
               ret
.data
brake_flag:
.word 0
.end
```