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)

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
```