Engineering 304

Lab #6: NIOS II Assembly Language – Interrupts

Purpose
The purpose of this lab is to learn about how exceptions (or interrupts) are handled on the NIOS processor.

Overview
In this lab you will work with the NIOS processor system that includes the memory assignments shown in the table below. The interrupt assignments are included in the table shown below. Also, note that the RAM has moved to a new range of addresses, so your <u>stack pointer initial value will need to be changed accordingly</u>.

Module Name	**Base Address**	**DE2 Item**	**Port Type**
On-chip Memory	0x0 (through 0x7fff)	Internal memory (32k)	Memory
SW	0x8000	18 slide switches	PIO port (input)
LEDR	0x8040	18 Red LEDs	PIO Port (output)
SevenSeg3to0	0x8080	HEX3,2,1, and 0	PIO Port (output)
SevenSeg7to4	0x80c0	HEX7,6,5, and 4	PIO Port (output)
KEY	0x8100	Four Push buttons	PIO Port (input) w/ **IRQ 1**
Timer	0x8140	Internal timer	Programmable Timer w/ **IRQ 0**

Part I – Timer Interrupt

1. Begin by copying the provided files from the shared drive to your H: drive in a Lab_SW06 directory.
2. Open and examine nios_iodefs.s and nios_irqdefs.s, notice that they provide .equ constants for your code to use as a convenience
3. This lab builds upon Lab SW05 but replaces IO device polling with IO device interrupts. Open and review your solution from Lab SW05 as a reference for starting Lab SW06.
4. Recall the following about Lab SW05:
    - The DE2 boards were loaded with an image of the NIOS processor executing your assembly code
    - The LED devices were used to show an incrementing binary number.
    - The rate of incrementing for that number was controlled using the Timer device, which timed out after a certain period.
    - The Timer device's period could be set by the user using the Switch device and Key device.
    - When Key[3] was pressed, the current state of the Switch device was interpreted as a binary number and used to set the Timer device's most-significant-sixteen-bits of period.
    - Psuedo code for this algorithm is:

```
main() {
    // initialize IO devices
    // counter = 0
    while ( true ) {
        if ( timer has timed out ) {
            // clear the timeout flag
            // stwio: counter->LED device
            // counter++
            // if counter>=MAX then counter=0
        }

        if ( key3 was pressed ) {
            // clear the key3 edge flag
            // ldwio & stwio: Switch device->Timer device PERIODH
            // restart the timer
        } } }
```

5. For Lab SW06 Part I, you will replace the polling check of the timeout flag with an interrupt handler that services timer timeout events. Consider the Psuedo code for the **new** algorithm:

```
main() {
    // initialize interrupts
    // initialize IO devices
    // counter = 0
    while ( true ) {
        if ( key3 was pressed ) {
            // clear the key3 edge flag
            // ldwio & stwio: Switch device->Timer device PERIODH
            // restart the timer
        } } }

ISR() {
    if ( timer has timed out ) {
        // clear the timeout interrupt condition
        // stwio: counter->LED device
        // counter++
        // if counter>=MAX then counter=0
    } }
```

6. Notice that the main routine never “calls” the interrupt service routine. The ISR can be triggered at any point during the main routine; it interrupts the main routine, executes, and then the main routine resumes where it had left off.
7. Using the provided ISR template assembly file, copy/paste relevant parts of your solution from Lab SW05 into your assembly code for Lab SW06. Then, update the Lab SW06 code to use the interrupt-based timer servicing as described in the pseudo code above. Consider the following:
   - Use Interrupt #0 to indicate the condition that was previously indicated with the timeout flag
   - The timeout flag no longer needs to be checked & the timer device needs to be initialized with its timeout interrupt enabled
   - The ISR will be triggered by ALL interrupts, but only Interrupt #0 indicates a timeout event, the IPENDING register is used to distinguish which interrupt caused the ISR to be triggered
   - The ISR should not assume that registers initialized by main() have certain values, it should use its own registers to hold IO device base addresses
   - Since the “et” register is reserved for use in the ISR, you should maximize the use of “et” and minimize use of other (shared with main) registers
   - No register values in the ISR should be considered “persistent” between ISR calls, any data that must persist should be stored in memory. This means that the counter value can no longer be a dedicated register, instead it should be a global variable.
   - If any CPU-registers aer used within the ISR, they must be pushed to the stack at the start of the ISR and popped from the stack at the end of the ISR. The ISR is essentially a function in this regard.
8. Verify that the system behaves “identically” to how it did for Lab SW05, even though it is implemented using the interrupt-based timeout handling instead of polling-based timeout handling.
9. Save a copy of your code and append “_PartI” to the filename of the copy.

Part II – KEY Interrupt

10. For Lab SW06 Part II, you will replace the polling check of the key[3] button press with an interrupt handler that services the button press events. Consider the Psuedo code for the **new** algorithm:

```
main() {
    // initialize interrupts
    // initialize IO devices
    // counter = 0
    while ( true ) {
        // Do nothing forever
    } }

ISR() {
    if ( timer has timed out ) {
        // clear the timeout interrupt condition
        // stwio: counter->LED device
        // counter++
        // if counter>=MAX then counter=0
    }

    if ( key3 was pressed ) {
            // clear the key3 interrupt condition
            // ldwio & stwio: Switch device->Timer device PERIODH
            // restart the timer
        } }
```

11. Update your assembly code to use interrupt-based key handling instead of polling-based key handling just like you did for the timeout events in Part I.
    - **Note for Spring 2019: See expanded pseudo code below**
12. Verify that the system behaves "identically" to how it did for Lab SW05, even though it is implemented using the interrupt-based key handling instead of polling-based key handling.
13. Save a copy of your code and append "_PartII" to the filename of the copy.

For Spring 2019: Snow days caused one fewer lab periods than normal, so two labs are being combined into one. This means that the labs need to be shortened, and part of that shortening involves providing more explicit pseudo code for Part II.

```
... within the ISR ...
/* CHECKFOR_INTR1 */
// et = IPENDING
// et &= 0b10 /* and-mask with bit #1 */
// if ( et == 0 ) goto CHECK_FOR_INT2
/* RESPONDTO_INTR1 */
// stwio: r0 -> EDGE_OFFSET of KEY_BASE /* clear edge flag */
// ldwio: rX <- DATA_OFFSET of SW_BASE  /* get switch value */
// stwio: rX -> PERIODH_OFFSET of TIMER_BASE /* set new period */
// stwio: 0b111 -> CONTROL_OFFSET of TIMER_BASE /* restart timer */

... when initializing the IO devices ...
// stwio: r0 -> EDGE_OFFSET of KEY_BASE /* clear the edge flag */
// stwio: 0b1000 -> MASK_OFFSET of KEY_BASE /* interrupt for key3 */

... when setting IENABLE ...
ori et, et, KEY_MASK /* just like the "ori et, et, TIMER_MASK" line */
```

Part III – Doing Something in Main()

14. For Lab SW06 Part III, you will add assembly code that represents long-running calculations happening in the main routine. Recall that the ISR may interrupt the long-running calculations at any time, and then they will resume once the ISR has returned. Additionally, you will utilize the 7SEG devices to indicate that progress is being made on the long-running calculations (without a progress indicator, it would be difficult to tell if the calculations were progressing or frozen/stalled). Consider the Pseudo code for the **new** algorithm:

```
main() {
    // initialize interrupts
    // initialize IO devices
    // counter = 0
    for ( i = 0 to 69 ) {
        doSomeWork()
        // "spin" the 7SEG segments
    }
    // turn on all 7SEG segments to indicate completion
    while ( true ) { do nothing forever }
}

ISR() {
    if ( timer has timed out ) {
        // clear the timeout interrupt condition
        // stwio: counter->LED device
        // counter++
        // if counter>=MAX then counter=0
    }

    if ( key3 was pressed ) {
            // clear the key3 interrupt condition
            // ldwio & stwio: Switch device->Timer device PERIODH
            // restart the timer
        } }
```

15. Note the following:
    - doSomeWork() is an assembly function which has been provided in doSomeWork.s
    - The 7SEG segments should "spin" in circles as progress is made, similar to the Windows loading cursor animation wheel that you may be used to seeing. If the "spinning" stops, the program's main routine is not executing or is executing very slowly.
    - Because the LED counter and its rate control via the switches and keys is all being done via interrupt handling, the program should continue to behave the same as PartII in that regard.
16. Update your assembly code to "doSomeWork" and indicate progress being made on the work by consulting the pseudo code above.
    - **Note for Spring 2019: See additional provided code below**
17. Verify that the system behaves "identically" to how it did for Lab SW05, with the addition of the "spinning" progress bar.
18. Save a copy of your code and append "_PartIII" to the filename of the copy.

For Spring 2019: Snow days caused one fewer lab periods than normal, so two labs are being combined into one. This means that the labs need to be shortened, and part of that shortening involves providing assembly code for PartIII.

Copy the following into its own advanceProgressDisplay.s file and simply call this function from main().

```
.include "nios_iodefs.s"   /* include i/o constants */
.text
.global advanceProgressDisplay
advanceProgressDisplay:

funcInit:
    /* push to stack */
    subi sp, sp, 12
    stw r18, 8(sp)
    stw r17, 4(sp)
    stw r16, 0(sp)

funcExec:
    movia r16, SEVENSEG30_BASE
    ldwio r18, DATA_OFFSET(r16) /* get current segment */
    nor r17, r18, r18           /* use lit segment for logic */
    slli r17, r17, 1            /* move the lit segment */
    andi r17, r17, 0b00111111   /* mask to check if edge segment */
    beq r17, r0, RESET_SPIN     /* if none lit, restart */
    br SET_SPIN
RESET_SPIN:
    movi r17, 1
SET_SPIN:
    nor r18, r17, r17 /* spin unlit segment instead of lit one */
    andi r18, r18, 0b00111111   /* mask to ensure edge segment */
    stwio r18, DATA_OFFSET(r16)

funcEnd:
    /* pop from stack */
    ldw r18, 8(sp)
    ldw r17, 4(sp)
    ldw r16, 0(sp)
    addi sp, sp, 12

    ret
.end
```

For Spring 2019: Snow days caused one fewer lab periods than normal, so two labs are being combined into one. This means that the labs need to be shortened, and part of that shortening involves providing assembly code for PartIII.

Copy the following into its own displayProgressComplete.s file and simply call this function from main().

```
.include "nios_iodefs.s"   /* include i/o constants */
.text
.global displayProgressComplete
displayProgressComplete:

funcInit:
    /* push to stack */
    subi sp, sp, 8
    stw r17, 4(sp)
    stw r16, 0(sp)

funcExec:
    movia r16, SEVENSEG30_BASE
    movia r17, 0xFFFFFFFF
    stwio r17, DATA_OFFSET(r16)
    movia r16, SEVENSEG74_BASE
    stwio r17, DATA_OFFSET(r16)

funcEnd:
    /* pop from stack */
    ldw r17, 4(sp)
    ldw r16, 0(sp)
    addi sp, sp, 8

    ret
.end
```

Part IV – Analysis of Interrupt Execution

19. Note that throughout out the following steps, this terminology will be used:
    - Execution Time:
        - Using a stopwatch, measure the number of seconds between pressing "play" on Altera and "all the 7seg segments turning on".
        - This is an estimate of how long the program executed, ignoring human timing errors.
    - Using Interrupts:
        - The easiest way to toggle interrupt behavior for this part of this lab is to use the PIE bit. After PartIII, there should be a line of assembly code such as "ori et, et, PIE_MASK" during initialization. Leaving this line as-is leaves interrupts enabled.
    - Without Interrupts:
        - The easiest way to toggle interrupt behavior for this part of this lab is to use the PIE bit. After PartIII, there should be a line of assembly code such as "ori et, et, PIE_MASK" during initialization. Commenting out this line disables interrupts.
    - PERIODH and PERIODL Values:
        - For this part of this lab, changing PERIODH and PERIODL values should be done for the default (hard-coded) period – NOT changed using the switches and key3. This simplifies the timing analysis.
20. Answer the questions below using a separate, well-formatted word-document printout for your submission. Fill in as many tables of the row as you need to while performing your analysis (~6).

PeriodH Value	**PeriodL Value**	**Period (seconds)**	**Execution Time (seconds)**
No interrupts active			
0x017d	0x7840	0.5	

A. What period (in seconds) causes the execution time to be double the execution time of the "Without Interrupts" case? This is the same as asking: how often do you need to run the ISR so that half of the total time is spent handling interrupts? Note the clock is 50MHz.

B. Draw a timeline…
   a. with a min value of 0 and a max value of "6 times the answer from Q-A"
   b. legibly label which portions of this timeline are "times when the ISR is executing" and which are "times when the main routine is executing"
   c. label each time the ISR returns with its time value on the timeline (the end of the ISR portions are the same as the start of the main routine portions)
   d. Hint: the period from Q-A is the duration of time between the start of an ISR portion and the start of the next ISR portion, this is the same as the duration between the start if a main portion and the start of the next main portion.

C. Estimate the execution time of the ISR (the duration of one ISR portion on the timeline).

D. Determine the number of machine code instructions in the ISR and use the answer from Q-C to estimate the amount of time it takes to execute each instruction.

Hand In:

1) A nicely formatted printout of your well-commented assembly code from Part I
2) A nicely formatted printout of your well-commented assembly code from Part II
3) A nicely formatted printout of your well-commented assembly code from Part III
4) A nicely formatted printout of your table and questions from Part IV