UNIVERSITY OF NEW BRUNSWICK - FREDERICTON
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE3221 Computer Organization

Lab #4

2025-01-06

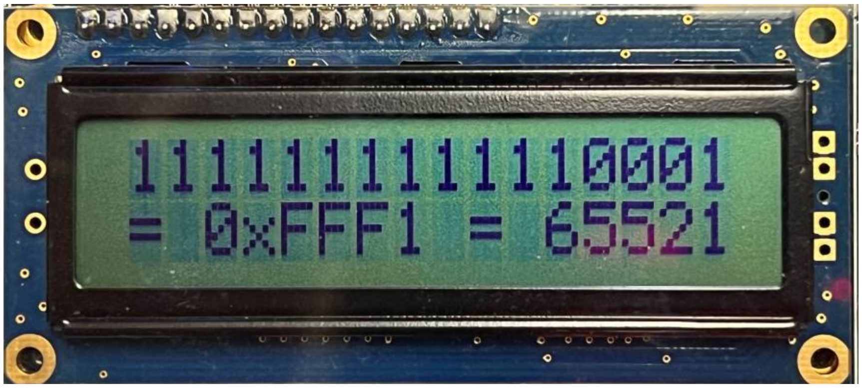
Display Updating using Interrupts

Introduction

In this lab you will use the NIOS-II processor to explore the use of interrupts and you will use the LCD display to display ASCII characters and strings. Subroutines developed in the previous lab may be reused as required and new ones will need to be developed.

1. You will set up the decade timer to generate 100 interrupts per second and write an interrupt service routine (ISR) to update a counter on the hex display. The use of interrupts ensures that this counting can continue even while a main program performs unrelated calculations that do not involve the hex display.

2. You will write a program that shows the 16-bit value of the switches on the LCD display in three different ways showing binary, hexadecimal and decimal every time the switches change.



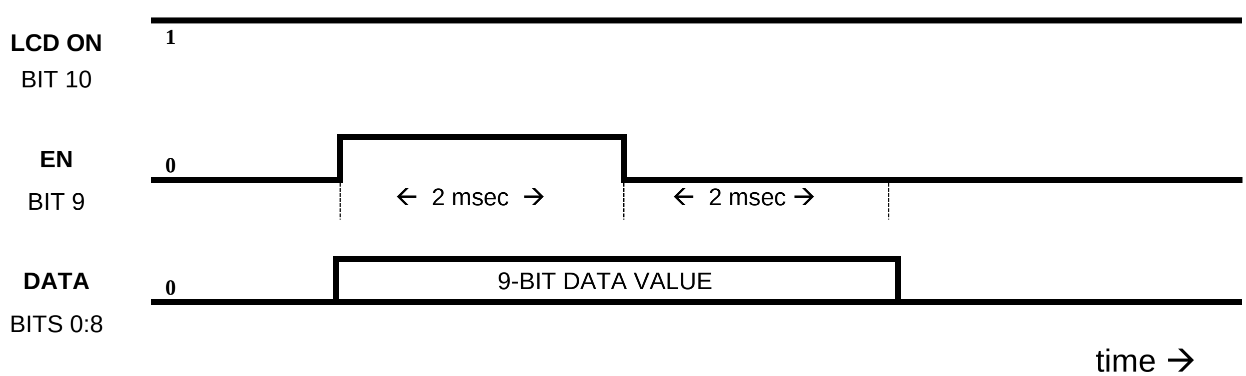
You are expected to have completed all previous ECE3221 *labs before embarking on this lab.*

Liquid Crystal Display (LCD)

The Liquid Crystal Display (LCD) can hold two lines of sixteen characters. The display is connected to a 16-bit output port as found in the *Hardware Reference Guide*. Characters are sent to the display one after the other as 8-bit ASCII codes and appear one after the other on the display until the end of a line is reached. Various 9-bit control codes can be sent in the same way; in particular, the code `0x101` clears the display, `0x180` moves the display cursor to the first position on the top line and `0x1C0` moves the cursor to the first position on the bottom line.

```
0123456789ABCDEF
0123456789ABCDEF
```

Data for the LCD is latched on every falling edge of the EN line while Bit 10 (LCD ON) must be held high at all times. **To send a control/character code to the display, send the data value in bits [8..0] with bit9=1 (EN) for 2.0 msec, then with bit9=0 for 2.0 msec while keeping the other bits unchanged.** For example, to display ‘A’=0x041, send 0x641 and wait 2.0 msec, then 0x441 and wait another 2.0 msec. How many characters could you send in 1/10 second?



The complete specification sheet for the LCD display can be found on D2L. The LCD display in the lab has no backlight.

> The *Clear Display* control code (0x101) requires some extra time to erase all the characters and send the cursor to the top of the first row. After sending 0x101 to the display it is necessary to wait an additional time (e.g. 20 msec) for the operation to complete.

ASCII Characters and Strings

If a digit such as 5 is to be displayed, the ASCII character ‘5’ must be sent. It happens that the ASCII codes for the characters ‘0’ to ‘9’ are 0x30 to 0x39, so the conversion can be accomplished by adding 0x30 or '0' to the corresponding binary digit 0 to 9. For example, the instruction `addi r3,r3,'0'` replaces a single digit in `r3` with its equivalent ASCII character. Multi-digit numbers must be processed in this way digit-by-digit to be displayed on the terminal.

A *string* is a sequence of characters stored in consecutive byte locations. In Nios II assembly language, an ASCII string is defined using double quotes and describes an array of bytes as:

```
hex: .ascii "0123456789ABCDEF"
```

where `hex` is the address of the string and of its first character `'0'`. The byte address of the character `'9'` is address hex+9 and this array can serve as a lookup table to find the hexadecimal ASCII character equivalent to any single 4-bit binary value.

Finally, a *null-terminated* ASCII string is a sequence of characters followed by an implicit null (zero byte). This array of characters at address `welcome` occupies 14 bytes including the hidden null terminator:

```
welcome: .asciz "Welcome to Lab 4"
```

Printing a Value in Binary

0	1
0x30	0x31

A single bit can be printed by adding 0x30 to get the corresponding ASCII code. If a subroutine `outbit` is created that prints a single bit, it can be called to print larger binary values one bit at a time.

Printing a Hex Character

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37	0x38	0x39	0x41	0x42	0x43	0x44	0x45	0x46

Printing 4-bit binary values in hexadecimal requires converting the value to and ASCII character as shown above. A lookup table `hex` (as shown above) could be used to accomplish this conversion. If a subroutine `outhex` is created that print a single hex digit, it can be called to print larger hexadecimal values one digit at a time.

Integer Division

The Nios-II instructions `mul` and `div` perform integer multiplication and division on 32-bit registers. For example, the division 13/5 gives 2 with remainder 3; the integer division instruction `div` returns the quotient 13/5=2 and the remainder is found as 13 – (5 × 2) = 3.

Let the numerator `r3=13` and denominator `r4=5`, then the above (unsigned) division is:

```
ori   r3,r0,13    # r3 = 13 = numerator   (n)
ori   r4,r0,5     # r4 =  5 = denominator (d)

divu  r5,r3,r4    # r5 = n / d  = 2 = quotient (q)
mul   r6,r5,r4    # r6 = q * d  = 2 * 5 = 10
sub   r7,r3,r6    # r7 = n - 10 = 3 = remainder(r)
```

Decimal Digits

Given a 16-bit value such as x = `0x5BA0` = 23456_{10}, individual decimal digits (2,3,4,5,6) can be extracted successively as powers of ten found in the quotient term (q) in each of the following divisions:

```
23456 / 10000 gives q=2, r=3456
 3456 / 1000  gives q=3, r=456
  456 / 100   gives q=4, r=56
   56 / 10    gives q=5, r=6     (final digits)
    6 / 1     gives q=6
```

As each quotient term is found (a single digit) it can be sent as an ASCII character to the hex display. As 16-bit integers go from 00000 to 65535 they can all be printed using 5 digits.

Required Subroutines

Design and debug each of the following subroutines separately and confirm that each operates properly and corresponds exactly to the design specifications given. **No registers should be affected in any routine** except as noted. *Hints are provided.*

A	NAME:	`outchr`		
	REQUIREMENTS:	Outputs to the LCD display a single ASCII character in `r3`. (allow for 9-bit input)		
	INPUTS:	`r3`	OUTPUTS:	`none`
	DEPENDENCIES:	`delayN`	REGISTERS AFFECTED:	`none`
Once this is working properly, other subroutines can simply use `outchr` whenever a character is to be displayed.				

B	NAME:	`outspc`		
	REQUIREMENTS:	Outputs a space character (0x20) to the LCD display.		
	INPUTS:	`none`	OUTPUTS:	`none`
	DEPENDENCIES:	`outchr`	REGISTERS AFFECTED:	`none`

Handy to have but not a long routine.

C NAME:	`delayN`		
REQUIREMENTS:	Does nothing but returns after a delay of N msec where N is provided in r3.		
INPUTS:	`r3`	OUTPUTS:	`none`
DEPENDENCIES:	`none`	REGISTERS AFFECTED:	`none`
This is the same delay subroutine from Lab 3.			

D NAME:	`clrscr`		
REQUIREMENTS:	Clears the LCD screen and moves the cursor to the top line by sending the command character 0x101, followed by a short delay while the clearing operation completes.		
INPUTS:	`none`	OUTPUTS:	`none`
DEPENDENCIES:	`outchr, delayN`	REGISTERS AFFECTED:	`none`
Use the delay subroutine to pause for 10 msec after the command 0x101 is sent to the display..			

E NAME:	`outhex`		
REQUIREMENTS:	Outputs to the LCD display one ASCII character being the hexadecimal representation of the 4 least significant bits in `r3`.		
INPUTS:	`r3`	OUTPUTS:	`none`
DEPENDENCIES:	`outchr`	REGISTERS AFFECTED:	`none`
A lookup table is one way to obtain the ASCII hex equivalent character to a 4-bit binary value.			

F NAME:	`out16bin`		
REQUIREMENTS:	Outputs to the LCD display 16 characters '0' or '1' being the binary representation of the 16-bit contents of `r3`.		
INPUTS:	`r3`	OUTPUTS:	`none`
DEPENDENCIES:	`outchr`	REGISTERS AFFECTED:	`none`
Loop 16 times to display 16 bits in r3 one after the other starting with bit15.			

G NAME:	`out4hex`		
REQUIREMENTS:	Outputs to the LCD display '0' and 'x' followed by 4 characters being the hexadecimal representation of the 16-bit contents of `r3`.		
INPUTS:	`r3`	OUTPUTS:	`none`
DEPENDENCIES:	`outhex, outchr`	REGISTERS AFFECTED:	`none`
Consider shifting 4 bits of interest over the LSB of r3 and calling `outhex` for each of the 4 hex digits.			

H NAME:	`out5int`		
REQUIREMENTS:	Outputs to the LCD display five ASCII characters being the 5-digit decimal representation of the 16-bit contents of `r3`.		
INPUTS:	`r3`	OUTPUTS:	`none`
DEPENDENCIES:	`outchr`	REGISTERS AFFECTED:	`none`
Decimal digits are each a power-of-ten, working down from the largest expected power in a 16-bit input.			

I NAME:	`outstr`		
REQUIREMENTS:	Outputs to the LCD display a null-terminated ASCII string at the address provided in `r3`.		
INPUTS:	`r3`	OUTPUTS:	`none`
DEPENDENCIES:	`outchr`	REGISTERS AFFECTED:	`none`

Call `outchr` repeatedly while working through a stored string, stopping when a null (0) character is encountered.

The following `init` subroutine would be used only once at the start of your program. In particular, the generation of interrupts in the decade timer and the handling of interrupts by the processor is set up and initialized. There is no need to push/pop registers in this initialization routine.

J	NAME:	`init`		
	REQUIREMENTS:	Set the stack pointer (sp), enable the hex display, enable interrupts for the 100 Hz decade timer.		
	INPUTS:	`none`	OUTPUTS:	`none`
	DEPENDENCIES:	`none`	REGISTERS AFFECTED:	`sp and any others used`

As is often the case, additional subroutines may be required as code development proceeds.

Initialize Interrupt Settings

The following steps are required to configure a device to generate interrupts and the processor to acknowledge and handle interrupts. Observe that the use of interrupts exceptionally involves the obscure Nios II instructions `rdctl` and `wrctl` to access internal processor control registers.

1. Set up a device to generate an interrupt request when some service is required or some event has occurred. In this case, setup the Decade Timer to generate an interrupt request (IRQ2) once every 10 msec (bit 3).

2. Set up the processor to acknowledge specific interrupts (in this case, only IRQ2);

3. More generally, setup the processor to respond to interrupts by branching to the ISR.

A final step is to create the actual interrupt service routine to provide a response to generated interrupts. In the Nios-II processor, any interrupt regardless of its source causes the processor to stop whatever it is doing and call the ISR at address 0x00000020; this is where the ISR must be located and where an action is chosen based on the interrupt number of the device requesting service (in this case IRQ2).

The stack pointer (sp) must be initialized *before* interrupts are enabled in Step 3 above.

Decade Timer Interrupt Generation

The decade timer was used in Lab 3 to create a delay of N msec. In that lab, a loop was used to wait for falling edges on a timer bit. In this lab, the timer port will be setup so that a falling edge on a selected timer bit generates an interrupt and there will be no need to wait in a loop continuously checking the bit. The setup code can be used directly in your `init` subroutine.

While the state of the decade timer outputs can be read directly at address 0x8850, the timer interface (like other DE2-115 devices) includes additional control registers related to interrupts:

ADDRESS	BIT31 – BIT8	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
0x8850	*unused*	DECADE TIMER OUTPUT BITS (read only)							
0x8858	*unused*	INTERRUPT ENABLE BITS (1 = enabled)							
0x885C	*unused*	PENDING INTERRUPT BITS (1 = waiting for service)							

Any decade timer bit can generate IRQ2 if it is first enabled by writing 1 to the corresponding control bit in the *interrupt enable* register. When an edge occurs on Bit3, the device sets Bit3 in the *pending* register and sends IRQ2 to the processor. Whenever the interrupt is finally serviced (in the ISR) the pending bit must be set back to 0 ready for the next edge on this output bit.

```
#-------------------------------------------
# SETUP INTERRUPTS IN THREE STEPS 1,2,3

      # (1) enable interrupt generation on 100 Hz edge in the decade timer
      ori   r22,r0,timer
      ori   r3,r0,0b00001000  # select 100 Hz output (bit 3)
      stbio r3,8(r22)         # bit 3 will now cause timer interrupt (INT2)

      # (2) recognize INT2 (decade timer) in the processor
      rdctl r3,ienable
      ori   r3,r3,0x00000004  # INT2 = bit2 = 1
      wrctl ienable,r3        # INT2 will now be recognized by the processor

      # (3) turn on master interrupt enable in the processor
      rdctl r3,status
      ori   r3,r3,0x01        # PIE bit = 1
      wrctl status,r3         # ISR will now be called on enabled interrupts
#-------------------------------------------
```

Interrupts may be manually disabled by simply commenting out the final line.

Interrupt Service Routine: `located at address 0x00000020`

An interrupt service routine (ISR) is essentially a subroutine that is called automatically whenever an interrupt occurs. Like a subroutine, program execution branches to the ISR (always address 0x00000020) and later returns back to where the program was when the ISR was triggered. Unlike a subroutine, an interrupt can happen at any time and at any line in the main program and when any registers may be in use; therefore, it is essential that *an ISR must not affect any registers*.

Insert the following code near the start of your program. The assembler directive `.org 0x0020` directs the assembler to place this ISR at the expected address. You must provide the `action2` subroutine to perform a specific action in response to IRQ2. In this lab, this action is to increment a counter value stored in memory and send it to the hex display.

```
# ==========================================
# interrupt service routine (ISR)

.org 0x0020   # code lies at this address

       push ra
       push r3
       push r4

       # determine source of interrupt
       # ------------------------------------

       rdctl r3,ipending          # r3 = pending interrupt bits
       andi  r4,r3,0x04           # r4 = pending int2 bit
       bne   r4,r0,int2           # service int2 if necessary
       br endint                  # or done (nothing to do)

       # ****************************
       # IRQ2 service (decade timer)
int2:
       call action2        # provide a specific response to the timer interrupt
                           # DO NOT MODIFY ANY REGISTERS

       # timer interrupt request is done
       movia r4,0x8870            # r4 = addr of decade timer
       sthio r0,12(r4)            # clear interrupt request

       br  endint                 # done
       # ****************************

       # ------------------------------------
endint:

       pop r4
       pop r3
       pop ra

       addi  ea,ea,-4             # adjust interrupt return address
       eret                       # done!
# ==========================================
```

Overall Program Structure: Starting Program Template

```
.global _start
_start: br Start  # begin at the main program

# ==========================================
# macro definitions (push/pop) at the top
```

```
# -------------------------------------------

# ===========================================
# interrupt service routine (ISR)
# -------------------------------------------
.org 0x0020    # ISR code lies at this address
ISR:

# ===========================================
# main program (after the ISR)
# -------------------------------------------
.org 0x0100    # code lies at this address

Start:
       call init # initialization

here:  br here    # the entire main program!!

# ===========================================


# ===========================================
# subroutines  (after main code)
# -------------------------------------------


# ===========================================
# data storage (after all code)
# -------------------------------------------
# reserve 400 bytes = 100 words for stack

.skip 400

stacktop:

counter: .word 0

# -------------------------------------------
# stored strings

welcome: .asciz "Welcome to Lab 4"

# -------------------------------------------
```

Procedure (Part One) - Interrupts

Refer to the program template as a starting program and to the provided ISR that requires an action subroutine.

1. Complete the provided interrupt service routine (ISR) template with a subroutine named `action2` to handle each occurrence of IRQ2.

   This action will be to increment a value `counter` stored in memory and to send the new value to the hex display. The counter should be set to zero in the init routine.

2. Run the main program as provided and confirm that the LEDs are changing as expected. Note that in the main program, the line `here: br here` is idling in an infinite loop yet the LEDs are being changed from within the ISR as determined by the regular decade timer interrupts.

Procedure (Part Two) – The LCD Display

In this section you will construct a program that reads the switches and prints the switch values in binary, hexadecimal and decimal. This program replaces the monotonous `here: br here` program from the supplied starting code so that the computer does something useful while the interrupts continue to update the hex display counter.

> While you are developing the code for the main program, it will be useful to turn off interrupts as it is impossible to single step through a program while an ISR is being called repeatedly. Once the program is working, turn interrupts on again to confirm that timer interrupt events are being serviced and the hex display count is updated at 100 Hz while the main program displays the value on the switches whenever they change.

As usual, it is good practice to proceed step-by-step towards the final solution so that you can confirm the operation of your other subroutines as you move through the development process.

1. Confirm the operation of your `outchr` subroutine by writing several characters to the display. This most important routine writes a single ASCII character to the display and is needed by most of the other routines.

   ```
   AAAA
   ```

   Once this routine is confirmed working, begin to develop your main program that starts by clearing the screen with your `clrscr` subroutine.

2. **Print a Welcome Message.** Use your `outstr` subroutine to print a welcome message on Line1. This only happens once the first time the program is run.

```
Welcome to Lab 4

```

3. **Read the switches.** Wait for the switches to change and record the new value in a memory location. This can be used on the next loop as you wait for a new change.

4. **Print the switch value in binary**. Clear the screen (sets the cursor to Line1) and use your `out16bit` subroutine to send 16 bits from the switches to Line1 of the display.

```
0000000000001011

```

5. **Print the switch value in hexadecimal.** Move the cursor to Line2, then separately print ‘=’ and a space before calling your `out4hex` subroutine to display the value of the switches as 4 hex digits.

```
0000000000001011
= 0x000D
```

6. **Print the switch value in decimal**. Now print space, ‘=’, and space before calling your `out5int` subroutine to display the value of the buttons as 5 decimal digits. Your program then branches back to check the switches again.

```
0000000000001101
= 0x000D = 00013
```

7. Now turn on the interrupts once more and observe as the hex display counter is updated

even as you check and display the switch values.