ELEC 4601 Lab Report Section A4E, Mon 8:35 AM – 11:25 AM

LAB 2 Yuzhou Liu, 100853392 (no partner) Oct 4, 2015

Part A:

Please see commented code for lab2a.pas in Appendix A.

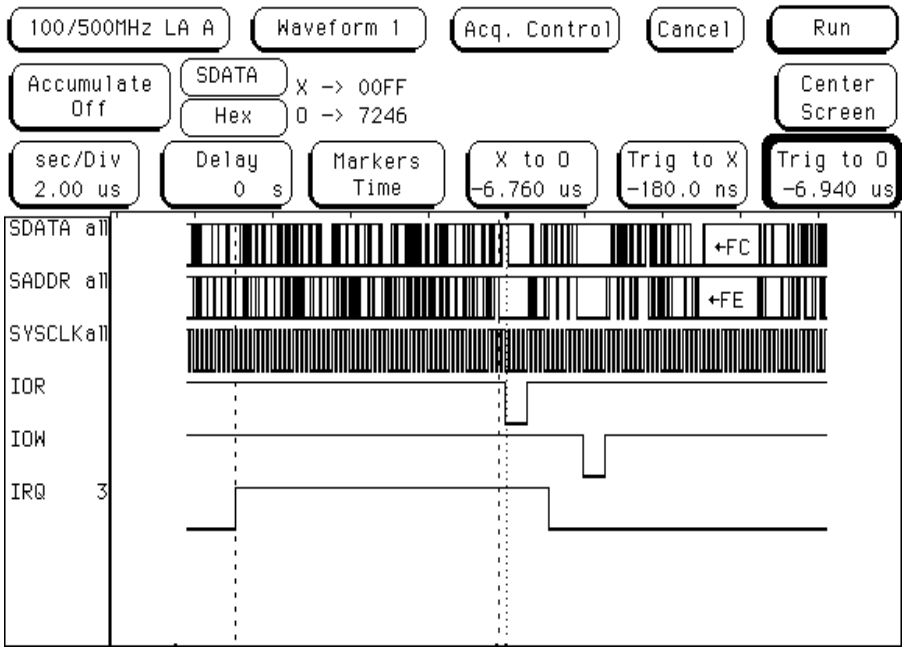

Figure 1: Time from IRQ7 rising edge to execution of marker

Shown in Figure 1, the "**interrupt response time**" or **time from IRQ7 rising edge to execution of marker** was 6.76 µs.

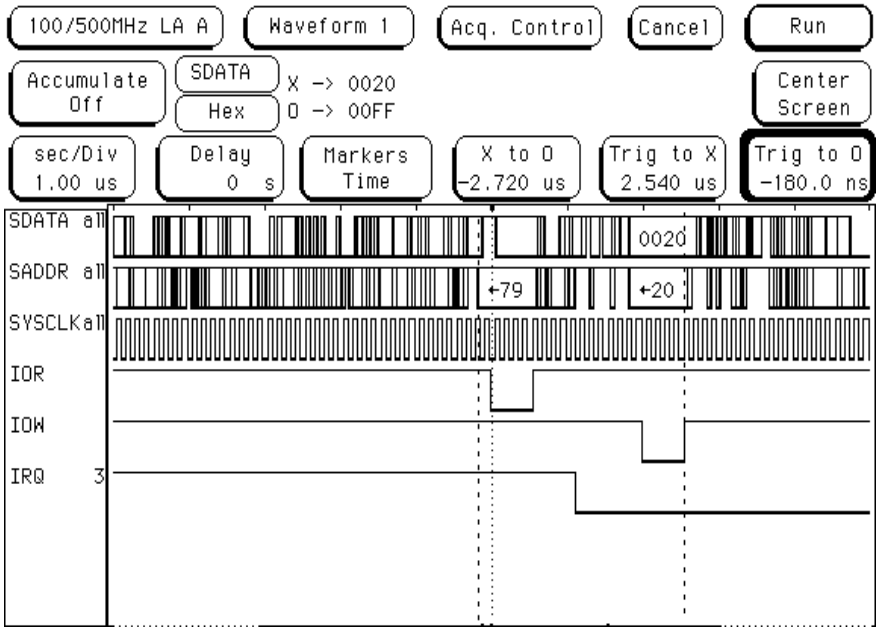

Figure 2: Time it takes to service the interrupt

In figure 2, the time it takes to **service the interrupt** from the start of the marker to when outputting EOI to PIC finishes is: 2.72 µs.

Estimate time for automatic steps ENTERING interrupt

Entering the interrupt, the process automatically:

1. Saves the PC on the stack
2. Saves the state of the CPU (status flags) on the stack
3. Identifies the source of the interruption and finds the ISR address of the interrupt
4. Executes the ISR

To calculate the time it takes for the automatic steps, subtract the time to save registers by using push commands from the time obtained in figure 1. Using the 80386 Programmer's Reference (http://intel80386.com/386htm/PUSH.htm) it lists the PUSH operand as taking 2 clock cycles to complete. From lab 1, the period of the reference SYSCLK was measured to be 120ns. So 2 clock cycles takes 240ns. In the ISR routine that I implemented, there are 3 PUSH commands, so 240ns x 3 = 720ns.

Knowing this and using figure 1 which shows time from IRQ7 rising edge to execution of marker, the time it takes for automatic steps for entering the interrupt can be estimated:

$$Time, automatic\ steps\ entering\ ISR = \ 6.76\mu s - 720ns = 6.04\mu s$$

Estimate time for automatic steps EXITING interrupt

Exiting the interrupt, the processor automatically:

1. Restores PC
2. Restores the state of the CPU (status flags)

Using the 80386 Programmer's Reference (http://intel80386.com/386htm/IRET.htm) it lists the IRET instruction (which does interrupt return restoring PC and pop flags) to take 22 clock cycles. Again knowing the period of SYSCLK as 120ns, then 22 clock cycles would take:

$$Time, automatic\ steps\ exiting\ ISR = 120ns\ x\ 22 = 2.64\ \mu s$$

Observations when input signal frequency is increased

Increasing the frequency sped up the counting of the LED lights. However as the frequency was sped up to above 120 kHz made the ISR fail completely. The LEDs would no longer be counting as they seem to be stuck. Also a keyboard press on the PC/104 to exit the program is nonresponsive.

Max Bandwidth of system

From figure 1, the time between rising edge on IRQ7 to the final execution of the code was measured as the sum of values from figure 1 and 2: 6.76µs + 2.72µs = 9.48 µs. So then the maximum frequency that interrupts can be generated before things start to fail is:

$$Freq_{max} = \frac{1}{9.48\ \mu s} = 105\ kHz$$

How does system fail above max bandwidth?

Inside our custom interrupt service routine, we do several operations such as storing registers, setting DS register, incrementing counter and sending EOI to PIC. These operations take a nonzero amount of time, as shown in figure 1.

Then this means that there is a frequency for the input clock which generates interrupts at a frequency higher than the frequency at which interrupts can be serviced. For example consider casting ballots at an election booth. If there are 4 people counting ballots at a speed of 20 ballots per minute, but there are 40 ballots being submitted every minute, then the ballots could not be accurately counted in real time as they are coming in.

Even if the interrupt service routine is made as small as possible where the only code left is to increment the counter for every interrupt, the operation still takes some nonzero amount of time to complete. So the input interrupt frequency could still be set to a level where interrupts are coming in faster than the time it takes for the increment operation.

How does the system fail if interrupts enabled in ISR?

In our lab, interrupts were not enabled within the ISR itself, so interrupts were serviced sequentially. If interrupts were enabled within the ISR itself, and if the input clock frequency was faster than the frequency at which interrupts were being serviced, then the program would be stuck generating infinite nested interrupts.

To explain further: if halfway through the ISR of interrupt #1 and before the counter is incremented, another interrupt occurred (interrupt #2), then the processor would switch to servicing the new interrupt. Due to the constant frequency of the input clock, another interrupt (interrupt #3) would be generated again before the program had a chance to fully service interrupt #2. Then while attempting to perform the ISR on interrupt #3, interrupt #4 occurs, and so on…

The program would have an ever growing list of nested interrupts to service. This occurs indefinitely unless the input clock frequency is suddenly decreased so the nested ISRs can be successfully completed one by one.

Why would you enable interrupts inside your ISR?

For more complex systems there may be higher priority interrupts which may occur during the ISR of the current interrupt. For example: a power failure which requires storage of all work to non-volatile memory, or debug functionality to pause the system at the current execution code.

If interrupts inside ISRs were disabled, high priority interrupts would have to wait for the current ISR to finish before they can run. This may not be suitable for critical interrupts, so interrupts must be enabled inside the counter ISR to allow the higher priority interrupt to be serviced first.

Decrease Latency

To decrease latency between when interrupt occurs and the execution of ISR, the processor's clock speed can be increased. Another option to keep latency to a minimum is to limit the amount of code in ISR to small as possible. In this case, we had a marker in there which was 2 lines, if we weren't using the logic analyzer then these 2 marker lines can be taken out. Taking out the markers also mean that the push dx and pop dx commands can be taken out. With the elimination of 4 commands inside the ISR, latency would decrease which increases system bandwidth.

PART B:

Please see commented code for lab2b.pas in Appendix B.

Parameters to guarantee proper communication

The following were set in both Tera Term and the PC/104

- Data: The number of data bits in each character transmitted. Here we set it to 8
- Parity: An extra bit for error checking in transmission. The bit is set so that the number of 1 bits in the character is either always odd or always even. If the receiver gets the wrong number of 1s then it knows the data must have been corrupted.
- Stop: Bits sent at the end of every character which allows the receiver to detect the end of a character and resynchronize with the character stream.
- Flow Control: Controls the transmission rate of the transmitter to prevent accidental transmission of data.

Sending Serial Data too Fast

The data is sampled at the baud rate. If the baud rate is set at the incorrect value, then the sampling of data will not be performed at the correct rate which results in a loss of data. The received data would not match the data sent. Some characters may be missing or the received characters would not match what was sent.

Flow Control

The flow control registers can be found in the Modem Control Register.

Hardware flow control works by using two wires in the serial cable: “Request to Send” and “Clear to Send” (Data terminal ready). When the computer wants to send data, it activates the Request to Send line. If the receiver has room for the data, it will reply by activating the Clear to Send (Data terminal ready) line and the computer starts sending data. If the receiver does not have room, then it will not activate the Clear to Send signal.

Error Bits

The error bits can be found in the “Line Status Register” (COM1+5).

To check if baud rate is wrong, check bit 1 and bit 3 of the Line Status Register.

Bit 1 is “Overrun Error” which occurs when the program can’t read a byte from the port fast enough before another byte comes in, the last byte will be lost and an overrun error will occur.

Bit 2 is parity error which occurs when the data transmitted has been corrupted because the expected number of 1 bits does not match what was received.

Bit 3 is framing error which occurs when the last bit is not a stop bit. This occurs when the speed at which the data being sent is different to that of the speed set for the receiver.

Explanation of odd-bit of code

```
mov dx,COM1+4
mov al,$0F
out dx,al
```

COM1+4 is the Modem Control Register. The code sets the 8-bit MCR to 00001111. The lowest bits of the MCR are described as follows:

Bit 5, 6, 7: reserved
Bit 4: Activates loopback modem, meaning any data sent out from the transmitter is looped back to the receiver on the same chip. This is useful for testing. Here loopback mode is set to false (0).
Bit 3: Aux output 2 – allows Aux output 2 to control the UART-CPU interrupt process
Bit 2: Aux output 1 – used to switch between 1.8432 MHz crystal to a 4MHz crystal
Bits 1 and 0: forces request to send line and data terminal ready line to be active

I think setting MCR to 0x0F is necessary to disable loopback, to allow aux output 2 to control the UART-CPU interrupt process and to switch between 1.8432 MHz crystal to a 4MHz crystal.

Appendix A

lab2a.pas, code for Part A

```
const
  EOI   = $20 ;
  PIC   = $20 ; { the 8259 interrupt controller }
  IRQ7  = $F ;
  LPT1  = $378 ;
  BUSY  : string[4] = '-\|/' ;

Procedure LptIsr ; far ; assembler ;
asm
  { saving registers that will be changed }
  push ds
  push ax
  push dx

  { marker for triggering logic analyzer }
  mov dx, $379
  in al, dx

  mov ax, seg counter  {Temporarily store into ax because ds cannot be accessed directly}
  mov ds, ax           {Setting the DS register to point to data segment of variables}

  { Increment the interrupt counter }
  inc counter

  { send the EOI command to the PIC }
  mov al, EOI
  out PIC, al

  { restore the registers that were changed }
  pop dx
  pop ax
  pop ds

  { make interrupt return }
  iret
{<_____________________________________________}
end ;
{
;
; The main program
;}
begin
  asm
{;
; Put zero in the 'count' variable
}
       MOV counter, 0
{
; send the EOI command to the PIC to ready it for interrupts
;}
       mov al, EOI
       out PIC, al
{;
; Disable IRQ7
; Done so that we don't get an interrupt before we are ready to
; actually process one.
;}
       in al, PIC+1
       or al, $80            {Set the most significant bit in IMR to a 1 to disable IRQ7}
       out PIC+1, al         {write the resulting bits to PIC+1, Interrupt Mask Register }
{;
; - Save the current IRQ7 vector for restoring later. For this processor,
;   this is just always done.
; - This is a system function call in any OS; We are running this in DOS
;   we will use the DOS INT 21 call to retrieve the interrupt vector.
;   reference http://en.wikipedia.org/wiki/MS-DOS_API
;       To set up the call:
;   AH = 0x35  this is the function number that fetches a vector
;   AL = the software interrupt vector that we want to read 0..255
;   INT 0x21 is the standard way of getting at DOS services. Any OS running
;            on this processor will use some variation of this idea.
; When the function call returns, you will find the interrupt vector
; in the es:bx registers; the es holds the segment, the bx holds the offset.
```

```
;}
       mov ah, $35
       mov al, IRQ7
       int $21
{;
; Put the values of ES and BX into the 32-bit 'saveint' variable
; so that we can restore the interrupt pointer at the end of the program.
;}
       {Put the value of BX into lower 16-bits of saveint}
       mov word ptr saveint, BX
       {Put the value of ES into higher 16-bits of saveint}
       mov word ptr saveint+2, ES
{
; Move the address of our ISR into the IRQ7 slot in the table
; Just like above, there is a DOS system call to do this. You could do it
; yourself, but it is always good policy to use a system call if there is
; one.
; Again, the call is made through INT 0x21 with some registers
; defined as follows in order to set up the call:
;   AH = 0x25  this is the function number that sets a vector
;   AL = the software interrupt vector that we want to set 0..255
;   DS:DX = the 32-bit address of our ISR
}
       push ds        {save ds since it will be modified}
       move ah, $25
       mov al, IRQ7

       mov bx, seg LptIsr   {Temporarily store into bx because ds cannot be accessed directly}
       mov ds, bx            {put segment of LptIsr into ds}
       mov dx, offset LptIsr  {put offset of LptIsr into dx}

       int $21
       pop ds         {restore ds}
{;
; Enable interrupts at the LPT1 device itself so that signals coming
; in on pin 10 on the LPT1 connector will cause an interrupt. }
       mov dx, LPT1+2
       in al, dx             {move the current value of LPT1+2 into al}
       or al, $10            {Set the bit to "enable IRQ through pin 10"}
       out dx, al            {Write the new values back to LPT1+2, control port }
{;
; Set up the PIC to allow interrupts on IRQ7. }
       mov dx, PIC+1
       in al, DX             {move the current value of PIC+1 into al}
       and al, $7F           {Set the bit for IRQ7 to be 0 to enable interrupts for IRQ7 }
       out dx, al            {Write the new values back to PIC+1, Interrupt Mask Register }
{
; We now go into a continuous loop
;}
@loop:
{;
; Check for a keypress to exit out, if no keypress: send the low 8-bits of the
; counter variable to the LED display on LPT1 so that you can see the
; results of counting the interrupts;
;}
    mov ah,1
    int $16
    jnz @alldone

       mov ax, counter
       mov dx, LPT1
       out dx, ax     { write the value of the counter variable to the LPT1 port to light the LEDs}
       jmp @loop      {; jump back to loop and check for key to quit}

@alldone:
{;
; Now undo all the steps that we went through to try and restore the machine to its original state
;}
{; Undo Step 6 by disabling the IRQ7 interrupt on the PIC
;}
       mov dx, PIC+1
       in al, dx
       or al, $80     {Disable IRQ7 by writing a 1 to the most significant bit}
       out dx, al     {Write the resulting bits to PIC+1, Interrupt Mask Register}
{;
; Undo Step 5 by disabling the LPT1's ability to interrupt
;}
```

```
        mov dx, LPT1+2
        in al, dx
        and al, $EF     {Set bit 4 to 0 to disable IRQ through pin 10}
        out dx, AL      {Write the resulting bits to LPT1+2, control port}
{;
; Undo Step 4 by replacing the interrupt service routine address with the original
; one that we saved in 'saveint' when we started up.
;}
        push ds         {Save ds since we will be modifying it}

        mov ah, $25
        mov al, IRQ7

        mov dx, word ptr saveint      {Move the 16-bit offset into dx}
        mov bx, word ptr saveint+2    {move the 16-bit segment into temporary register bx}
        mov ds, bx                    {move segment into ds using bx. ds cannot be accessed directly}

        int $21

        pop ds          {restore ds}
  end ;
end .
```

Appendix B

lab2b.pas, code for Part B

```
const
  EOI  = $20 ;
  PIC  = $20 ;  { the 8259 interrupt controller }
  IRQ4 = $C ;  { the IRQ + vector offset used by the serial port }
  COM1 = $3F8 ; { the address of the serial port, an INS8250 chip }
  LPT1 = $378 ;
{;
; variables to be used (a Pascal construct)
;}
var
  counter : word ; { 16-bit number }
  saveint : pointer ; { 32-bit pointer }
  rxchar  : byte ; { the received serial character }
  txchar  : byte ; { the transmitted character }
{;
; This is the Interrupt Service Routine (ISR)
}
Procedure ComIsr ; far ; assembler ; { this is a Pascal construct for assembly procedures, pretty much
like C }
asm
        {Save registers that will be modified}
        push ax
        push ds
        push dx

        mov ax, seg counter
        mov ds, ax              {Set DS to point to the data segment of variables}

        mov dx, COM1
        in al, dx               {Read in the value on COM1}
        out dx, al              {Write the same value read in previously back onto COM1}

        inc counter

        mov al, EOI
        out PIC, al             {Send EOI command to PIC}

        {Restore saved registers}
        pop dx
        pop ds
        pop ax

        iret
end ;
{
; The main program
;}
begin
  asm
```

```
{
; Put zero in the 'counter' variable}
       mov counter, 0
{{;
; Then need to set up the serial parameters for COM 1 to allow communication
; with a serial terminal.
; }
       mov dx, COM1+3
       in al, dx              {place the current value in COM1+3 into al}
       or al, $80             {Set the most significant bit to 1}
       out dx, al             {write the resulting bits to COM1+3}

       { Choose 9600 baud rate, which gives divisor 12}
       mov dx, COM1
       mov al, $C
       out dx, al             {Set COM1 (low byte) to be $C = decimal 12}

       mov dx, COM1+1
       mov al, $0
       out dx, al             {Set COM1+1 (high byte) to be $0}

       mov dx, COM1+3
       in al, dx
       and al, $7F            {Set the most significant bit of COM1+3 to a 0 to set baud rate}
       out dx, al             {Write the resulting bits to COM1+3}

       mov al, $3
       out dx, al             {Write $0 to COM1+3. Means no parity, one stop bit and 8-bit data}

       { send the EOI command to the PIC to ready it for interrupts}
       mov al, EOI
       out PIC, al
{
; Disable IRQ4 using the interrupt mask register IMR bit in the PIC#1 for now
; so that we don't get an interrupt before we are ready to actually
; process one.
}
       in al, PIC+1
       or al, $10
       out PIC+1, al  {Disable IRQ4 by setting bit 4 to a 1}
{
; Save the current IRQ4 vector for restoring later
;      To set up the call:
;   AH = 0x35  this is the function number that fetches a vector
;   AL = the software interrupt vector that we want to read 0..255
;   INT 0x21 is the standard way of getting at DOS services. Any OS running
;            on this processor will use some variation of this idea.
; When the function call returns, you will find the interrupt vector
; in the es:bx registers; the es holds the segment, the bx holds the offset.
;}
       mov ah, $35
       mov al, IRQ4
       int $21
{ Put the values of ES and BX into the 32-bit 'saveint' variable}
       mov word ptr [saveint], BX    {Place BX into the lower 16-bits of saveint}
       mov word ptr [saveint+2], ES  {Place ES into the higher 16-bits of saveint}
{
; Move the address of our ISR into the IRQ4 slot in the table
; The call is made through INT 0x21 with some registers
; defined as follows in order to set up the call:
;   AH = 0x25  this is the function number that sets a vector
;   AL = the software interrupt vector that we want to set 0..255
;   DS:DX = the 32-bit address of our ISR
}
       push ds                {Save ds}

       mov ah, $25
       mov al, IRQ4

       mov dx, offset ComIsr  {move the offset of the custom Com interrupt service vector into dx}
       mov bx, seg ComIsr             {use temporary register since ds cannot be accessed directly}
       mov ds, bx                             {move the segment of the custom ComIsr into ds}
       int $21

       pop ds                 {restore ds}
{;
;
```

```
;***************** BONUS ********** BONUS ***********************
; !! This is a given because you would likely never find this step.
; BONUS, describe what this does and why this is necessary }

               mov dx,COM1+4  { ; !! odd bit only required on some systems, like this one. typical pain
}
               mov al,$0F
               out dx,al
{;
;***************** BONUS ********** BONUS ***********************
;
; Enable interrupts at the COM1 device itself so that once a serial character
; is received from the Windows terminal, you will get an interrupt}
       mov dx, COM1+1
       in al, dx              {Put the bits from Interrupt Enable Register into al}
       or al, $1              {Set least significant bit from to 1 to enable receiver interrupt}
       out dx, al             {Write resulting bits back to COM1+1, IER}
{; Set up the PIC to allow
; interrupts on IRQ4}
       in al, PIC+1
       and al, $EF    {Set bit 4 of the interrupt mask register to 0 to enable interrupts from IRQ4}
       out PIC+1, al  {write resulting bits back to PIC+1, Interrupt Mask Register}
@loop:
{;
; Check for a key press to exit out, otherwise send the low 8-bits of the
; counter variable to the LED display on LPT1 so that you can see the
; results of counting the interrupts;
;}
    mov ah,1
    int $16
    jnz @alldone

       mov ax, counter
       mov dx, LPT1
       out dx, ax     {Write OUT the value of the counter variable to the LPT1 port to light the LEDs}

       jmp @loop      {; jump back to loop and check for key to quit}

@alldone:
{;
; An interrupt was generated, the code handled it and displayed something.
; Now undo all the steps that we went through to try and restore the machine to
; it's original state
;
; Undo Step 6 by disabling the IRQ4 interrupt on the PIC
;}
       in al, PIC+1
       or al, $10             {Set bit 4 to a 1 to disable IRQ4}
       out PIC+1, al  {write resulting bits to PIC+1, Interrupt Mask Register}
{;
; Undo Step 5 by disabling the COM1 port's ability to interrupt
;}
       mov dx, COM1+1
       in al, dx
       and al, $FE            {Set least significant bit to 0 to disable receiver full interrupt}
       out dx, al             {Write resulting bits to Interrupt enable register}
{;
; Undo Step 4 by replacing the interrupt service routine address with the original
; one that we saved in 'saveint' when we started up.
;}
       push ds                {save ds}

       mov ah, $25
       mov al, IRQ4

       mov dx, word ptr saveint      {move the offset saved in low 16-bits of saveint into dx}
       mov bx, word ptr saveint+2    {use temporary register since ds cannot be accessed directly}
       mov ds, bx                    {move the segment saved in high 16-bits of saveint into ds}

       int $21

       pop ds         {restore ds}
  end ;
end .
```