Lab 17: Microcomputer I/O Projects With the SCSI Port

U.C. Davis Physics 116B
Rev 6/4/00

INTRODUCTION

How do computers communicate with the outside world? Some obvious examples are video screens, keyboards, mice, and printers. Less obvious are the *ports,* or connectors on the back of the computer. The M68000 microcomputer communicates with all these devices in a similar way: it reads from or writes to a particular address as if it were a normal memory location. This technique is called *memory-mapped I/O*. Each external device is assigned a particular address or set of addresses. When the microcomputer accesses these specific addresses, it is not accessing a memory chip but the outside world via the external device.

In this lab, you will study and modify assembly language programs which use the Small Computer System Interface (SCSI, meant to be pronounced "sexy" but "scuzzy" caught on) to connect to the outside world. Using our custom UC Davis interface socket, you will connect the computer to LEDs, switches, D flip-flops, a DAC and a comparator on your breadboard. After some preliminaries, you can make the computer output waveforms and measure external voltages. If desired, you can go on to play back or sample waveforms. A commercial device to sample, modify and play back sounds will also be available for investigation.

Caution: **The SCSI interface connects directly to the insides of the Macintosh. It can only handle voltages between 0V and +5V.**

To avoid damage to the computer, please do the following:

1. Make sure the breadboard's power supply is set to the "TTL" voltage level.
2. "Common ground" the Macintosh to the breadboard. To do this, connect any of the Mac interface GND contacts to the GND of the breadboard.
3. *Don't connect the +15V or -15V power supplies to any of the Mac cable contacts.*
4. Less critical: don't connect any of the Mac cable contacts directly to +5V. Use a resistor of about 1kΩ to do this. The common ground bus or TTL logic outputs from the breadboard are OK.

1. SCSI INTERFACE BASICS

The SCSI bus is usually used according to a specific protocol to access devices such as disk drives. It connects to the Mac M68000 I/O lines via a commercial interface chip. Here, we will make use of the bus in a highly non-standard way which would interfere with any other devices connected to it. Fortunately, the Mac SE can run its operating system and the MAS program from floppy disks which do not use the SCSI bus. Our programs would cause havoc on a normal system with a SCSI hard drive.

The interface provides us with 8 parallel bi-directional data lines. There are also control input and output lines which can also be used for negotiating valid transfers of data between the computer and the external devices (referred to as "handshaking"). The 8 data lines are designated, /DB0, /DB1,...,/DB7 on the connector. The "/" indicates that these lines are "active low" (a program writing a "1" will produce a "low" state on the corresponding line). This is done since a normal keyboard can't make a bar over the top of the signal name. Other lines which we will use are two input lines, /ATN and /ACK, and two output lines, /BSY and /SEL. I refer to the data register as register 0, the register containing the control lines as register 1 and a third register for initializing the SCSI bus as register 2. Separate addresses are used to read and write the registers. There is some inconsistency in numbering of the bits in the control register (register 1) between writing and reading, however, so perhaps it is an oversimplification to call this a single register.

We will use 5 addresses to access these lines. The data at each address is one byte wide, but except for the data lines, we will only use certain of the available bits, being careful not to modify the others. Here are the addresses we will use (recall that the dollar sign in front of the number indicates it is hexadecimal):

Function	Address
Read data lines:	$580000
Write data lines:	$580001
Read control line bits:	$580050
Write control line bits:	$580011
Write bit to initialize SCSI bus:	$580021

The functions of some of the bits in these registers (the ones we will use) are:

Register 0 (read or write):	8 bits of data input or output (I/O), namely DB7–DB0
Register 1 (write):	Bit 0 sets direction of data I/O, 0=input, 1=output
	Bit 2 asserts SEL (makes /SEL go low)
	Bit 3 asserts BSY (makes /BSY go low)
Register 1 (read):	Bit 0 reads ACK (1 when /ACK is low)
	Bit 1 reads ATN (1 when /ATN is low)
Register 2 (write):	Write $40 to initialize the SCSI port.

The following code illustrates the use of the data lines to control LEDs. Since Mac coding practice does not permit direct addressing, the I/O addresses are loaded into address registers to allow access by indirect addressing.

```
; Simple test of SCSI output to control LEDs
; Note use of indirect addressing to access the I/O locations
; Do not try this with actual SCSI devices attached
; Single-step with debugger and watch LED's

            xref        stop

begin:      move.l      #$580000,A0       ;Data I/O read address
            move.l      #$580001,A3       ;Data I/O write address
            move.l      #$580011,A1       ;Control write address
; Program illustrating non-standard use of SCSI bus to write
; data to external devices (LED's in this case).
            move.l      #$580021,A2       ;Init reg write address
            move.b      #$40,(A2)         ;Initialize SCSI port
            move.b      #1,(A1)           ;Set I/O direction to OUT
            move.b      #5,(A3)           ;Write 5 to the LEDs
            move.b      #6,(A3)           ;Write 6 to the LEDs
            jsr         stop
            end
```

Initial exercise:

Common ground the Mac to the breadboard as described in the "caution" section above. Connect the lower 4 data I/O lines (DB3 - DB0) to the lamp monitors on the breadboard. Type in the program above. Using the debugger, step through this program until you reach the step that writes a "5" to the LEDs. Before executing that step, the LEDs may show any data, but after executing it, they should show an *inverted* binary 5. That is, binary 0101 is decimal 5 so binary 1010 is the inverted decimal 5. Step to the next step and you should see an inverted 6, that is binary 1001.

For your lab notebook, just say that you did this part and got it to work.

2. SCSI I/O Tour

Here is a more elaborate program for the next stages of our investigation of I/O:

```
; SCSI I/O Tour
; =============
; Program to test SCSI I/O, computer speed and pulse rate
; Send pulse on data lines
; Read and test /ATN
; Repeat if /ATN is false
; Stop when /ATN is true
; Set BSY at start and clear BSY at end
;  (Note that the "/" means negation. The ATN line is "active low")
;
;    D. Pellett  6/2/00
;

      xref stop

; Define literal constants for accessing
; the memory-mapped SCSI registers.
; The EQU directive literally replaces the occurrence of the
; defined string with the constant in the input file to the
; assembler.

D_Read      equ   $580000     ; Address to read SCSI data (reg. 0)
D_Write     equ   $580001     ; Address to write SCSI data (reg. 0)
C_Read      equ   $580050     ; Address for reading controls (reg. 1)
C_Write     equ   $580011     ; Address for writing controls (reg. 1)
D_Out       equ   1     ; bit value to assert SCSI data bus for reg 1
D_In        equ   0     ; bit value to read SCSI data bus for reg 1
Set_BSY     equ   8           ; bit value to assert BSY for reg 1
Set_SEL     equ   4           ; bit value to assert SEL for reg 1
S_Init      equ   $580021     ; Address for initializing SCSI (reg. 2)

; Initialize SCSI bus and set up I/O registers properly

      move.l      #S_Init,a0  ; This translates to move.l #$580021,a0
      move.b      #$40,(a0)   ; Initialize SCSI port
      move.l      #D_Write,a0 ; Data output register address in a0
      move.l      #C_Write,a1 ; Put control output address in a1
      move.b      #D_Out+Set_BSY,(a1) ; Assert data bus and BSY in reg 1
      move.l      #C_Read,a2  ; Put control input address in a2

; Send data until /ATN is pulsed

      move.b      #$FF,d0
      move.b      d0,(a0)     ; Write $FF
loop: clr.b       d0
      move.b      d0,(a0)     ; Write 0
      not.b d0
      move.b      d0,(a0)     ; Write $FF
      move.b      (a2),d1     ; Read control reg. into d1
      btst        #1,d1       ; Test bit 1, corresponding to ATN
      bne         loop        ; Condition code Z=1 if /ATN=0
      move.b      #D_Out,(a1) ; Clear BSY
      jsr         stop        ; Stop
      end
```

You can load this program in from the disk. It should be on MAS1 as scsi_io_tour.a. Examine the program and its comments to find out what it does. Do not run it until you have made the following connections. Otherwise, you will get stuck in an infinite loop and have to reboot the Mac.

Connections: Connect the +5V supply (be sure it is set to TTL), ground and +/- 15 V supplies to the usual breadboard busses. Connect the breadboard ground to one of the GND connections on the SCSI interface socket. Connect an LED indicator to /DB0, /BSY and /SEL on the SCSI interface socket. Connect a wire from /ATN to an unused breadboard contact strip near one of the pushbutton pulsers and connect a 1kΩ resistor from there to the positive-going pulse output.

Initial test: Go to the debugger. Single-step through the program to observe the operation of the registers and the LEDs. Is the behavior of /BSY and /D0 what you expect? Check how the program logic is done, particularly with respect to branches and the fact that the control lines are active low. Try to understand the BTST command and the way the program looks for the 0->1 transition on /ATN by comparing with the previously read value in d2. We will use this later to synchronize writing data with an external clock pulse. Optional: make a flow chart of the program.

After the loop has been traversed one or two times, hold down the pushbutton connected to /ATN as you step through the loop. This should result in exiting the loop and stopping. If not, check your connections. Once this works satisfactorily, exit the debugger and run the program normally. It should be possible to stop it by pushing the button.

Observation of pulses and timing: Now start the program and let it run while you observe the output of any data bit (e.g., /DB0) on the oscilloscope. Note that the pulse has an exponential rise and rapid drop. The pulse probably does not have enough time to get much above 2V before being set back to 0V. The slow rise is due to protective series resistors in the data output path. Measure the width of this pulse and the time between successive pulses. From the pulse width, estimate the order of magnitude of the time required for a single instruction (different instructions can take quite different times depending on how many times memory must be accessed, etc.). Are the pulse width and the time between successive pulses stable? Can you think of reasons why these times may not be constant? For example, is the computer involved in other tasks besides running this program which may be “stealing cycles?”

3. Digital Waveform Synthesizer

Note: this section describes use of an unbuffered 1408 DAC. Modify the circuit to use the buffered DAC. This will avoid the need for the separate octal D flip-flop chip.

The next project is to make a digital waveform synthesizer similar to what you did in Lab 15. In principle, we can now produce elaborate waveforms using large memory arrays and the computer’s arithmetic processing capabilities. But we will have to synchronize with an external clock signal to overcome the timing fluctuations observed in the previous section.

Turn off the breadboard power and wire the circuit as shown in the diagrams at the end of this writeup. The chips on the breadboard are a hex inverter (e.g., 74LS04), an octal D flip-flop buffer (74LS377) to latch the computer data output in response to an external clock pulse and the 1408 DAC. Leave plenty of room on the right side of the breadboard for later additions of an LM311 comparator and a few other parts. Wire the power connections first, then the logic connections on the board, and finally the connections to the SCSI socket. The /ATN line is used to stop the program as before and the /ACK line is used to request the next byte of data from the computer.

Before connecting the signal generator, set it to produce a 10 KHz TTL-compatible square wave. Adjust the amplitude and variable output offset so the signal has its low level at 0 V and its high level at approx. 4 V (less than 5 V). Be sure the oscilloscope input is DC coupled so you won’t be fooled about the low voltage.
Load the program, scsi_ramp.a, shown below. Once again, single step through it with the debugger to follow its logic and be sure it responds to the /ATN push button by stopping. Do you see how the ramp is generated? Why are the bits sent to the octal latch on /DB0–/DB7 not inverted? Check that the DAC output is in the range 0 to -2 V. Run the program and observe

whether the waveform period is more stable than what you observed in the previous section (even now it may not be perfect). How high can the signal generator frequency go and still produce reliable operation?

```
; scsi_ramp.a: Program for externally synchronized SCSI I/O.
; Send waveform (ramp) on data lines synchronized with 0->1 transition
; on /ACK line for storage in external 8-bit register clocked
; by the transition (stores previously output value and
; prepares next one).
; Stop when /ATN goes true
;    D. Pellett  6/2/00
      xref stop

; Define literal constants for accessing
; the memory-mapped SCSI registers.
D_Read      equ   $580000     ; Address to read SCSI data (reg. 0)
D_Write     equ   $580001     ; Address to write SCSI data (reg. 0)
C_Read      equ   $580050     ; Address for reading controls (reg. 1)
C_Write     equ   $580011     ; Address for writing controls (reg. 1)
D_Out equ   1          ; bit value to assert SCSI data bus for reg 1
D_In  equ   0          ; bit value to read SCSI data bus for reg 1
Set_BSY equ 8          ; bit value to assert BSY for reg 1
Set_SEL equ 4          ; bit value to assert SEL for reg 1
S_Init      equ   $580021     ; Address for initializing SCSI (reg. 2)

; Initialize SCSI bus and set up I/O registers properly

      move.l      #S_Init,a0  ; This translates to move.l #$580021,a0
      move.b      #$40,(a0)   ; Initialize SCSI port
      move.l      #D_Write,a0 ; Data output register address in a0
      move.l      #C_Write,a1 ; Put control output address in a1
      move.b      #D_Out,(a1) ; Assert data bus in reg. 1
      move.l      #C_Read,a2  ; Put control input address in a2

; Send data (ramp) until /ATN is pulsed
; Synchronize with /ACK (write data when /ACK goes from 1 to 0)
; Save previous value of /ACK in d2
; Use d0 for data to send
; Use d1 to read control signals

      move.b      #$FF,d0
      move.b      d0,(a0)     ; Write $FF to zero the output lines
      sub.b       d3,d3       ; Set d3 to 0
      move.b      #1,d1       ; Prepare to initialize d2
loop: addq.b      #1,d3       ; Increment d3 to send next time
      move.b      d3,d0       ; Put in d0
      not.b       d0          ; Complement bits for positive logic out
loop1:move.b      d1,d2       ; Save previous d1 bit 0 in d2
      move.b      (a2),d1     ; Read control reg. into d1
      btst        #1,d1       ; Test /ATN
      beq         fin         ; Done if /ATN=0
      and.b       #1,d1       ; Mask off all but bit 0
      cmp.b       d1,d2       ; Look for /ACK low to high
      ble         loop1       ; Not low to high, try again
      move.b      d0,(a0)     ; Write data
      jmp   loop              ; Do next byte
fin:  jsr   stop              ; stop
      end
```

Programming exercise: Modify the code to generate a repetitive waveform using an array of data values stored in memory. Continue to use /ATN to stop the program.

4. Successive Approximation ADC

Now we add a comparator to our circuit and change the connections somewhat to make a successive approximation analog to digital converter. We do not really need the octal D flip-flops this time but we will leave them in to avoid rewiring the data connections, /DB7–/DB0. Turn off the power and wire the comparator as shown in the diagram. Also provide a voltage divider to produce a test voltage in the range 0 to –2 V. Disconnect the signal generator input to the inverter on the left of your circuit board. Now connect the /SEL line to the inverter input (which drives the octal D flip-flop clocks) so the computer can latch its own data. Disconnect the /ACK line from the inverter output and hook it to one of the pulser pushbuttons (positive going pulse) via a 1 kΩ resistor. Connect the /ATN line to the comparator output. The /BSY line can be hooked to a LED.

The conversion will be initiated by a 0->1 transition on the /ATN line. The /BSY line will be used to indicate that the digitization is in progress. Use of these two lines would allow an external device to look to see that the ADC was available before initiating a measurement if successive measurements are being made.

As indicated above, the DAC output is connected to the comparator "–" input and the unknown voltage, V_x, to the "+" input. The comparator output is connected to the /ATN input to the SCSI control register. The computer conducts a binary search for the voltage closest to V_x and in so doing, develops the binary code representing V_x. The computer code to do this is provided in the file, SA_ADC.a, printed on the next page. The BSET and BTST commands are used in conjunction with d0 (to send the data to the DAC) and d3 (to keep track of the bit being tested). The bits are numbered from 7 to 0 (lsb). First, the computer sends 10000000, (bit 7 equals 1) causing the DAC to supply a voltage half of the full scale value. The comparator output is tested to see if this is too large (meaning too negative in this case). If so, the bit is cleared (not cleared otherwise) and the next bit is tested. This proceeds until all bits are tested and the digitized value of V_x determined.

After you have finished wiring your circuit, test the program with the debugger. Follow the logic of the program and watch the approximation proceed bit by bit. Also observe the DAC output voltage as the digitization progresses. Once you are convinced it is working correctly, run the program and see if the printed output makes sense for the input voltage used.

Additional exercise if time permits: make an overall loop to perform a fixed number of successive digitizations. The loop counter limit is needed since this program does not provide for an exit signal like the previous two programs. Use the loop and the /BSY signal to determine how much time is needed for a digitization. You could also digitize a waveform and store it in successive memory locations. What would be the maximum frequency allowed for the input signal to be digitized without aliasing?

```
; SA_ADC.a: Do a single successive approximation analog to digital
; conversion initiated by 0->1 transition on the /ACK line.
; The /ATN line will be used to read back the comparator output.
; The /BSY line will be used to indicate digitization in progress.
;    D. Pellett  6/2/00

      xref hexout, stop

; Define literal constants for accessing
; the memory-mapped SCSI registers.
D_Read      equ   $580000     ; Address to read SCSI data (reg. 0)
D_Write     equ   $580001     ; Address to write SCSI data (reg. 0)
C_Read      equ   $580050     ; Address for reading controls (reg. 1)
C_Write     equ   $580011     ; Address for writing controls (reg. 1)
D_Out equ   1           ; bit value to assert SCSI data bus for reg 1
D_In  equ   0           ; bit value to read SCSI data bus for reg 1
Set_BSY equ 8           ; bit value to assert BSY for reg 1
Set_SEL equ 4           ; bit value to assert SEL for reg 1
S_Init      equ   $580021     ; Address for initializing SCSI (reg. 2)
; Initialize SCSI bus and set up I/O registers properly
      move.l      #S_Init,a0  ; This translates to move.l #$580021,a0
      move.b      #$40,(a0)   ; Initialize SCSI port
      move.l      #D_Write,a0 ; Data output register address in a0
      move.l      #C_Write,a1 ; Put control output address in a1
      move.b      #D_Out,(a1) ; Assert data bus in reg. 1
      move.l      #C_Read,a2  ; Put control input address in a2
; Synchronize with /ACK (start ADC when /ACK goes from 1 to 0)
; Save previous value of /ACK in d2
; Use d0 for data to send
; Use d1 to read control signals
; Use d3 for bit number being tested (7 to 0)
; Look for /ATN low to high to start ADC
      move.b      #1,d1       ; Prepare to initialize d2
tloop:move.b      d1,d2       ; Save previous d1 bit 0 in d2
      move.b      (a2),d1     ; Read control reg. into d1
      and.b       #1,d1       ; Mask off all but bit 0
      cmp.b       d1,d2       ; Look for /ATN low to high
      ble         tloop       ; Not low to high, try again
; Now do ADC
      move.b      #D_Out+Set_BSY,(a1)     ; Set BSY
      move.b      #7,d3       ; d3 contains the bit number being tested
loop: bset        d3,d0
      not.b       d0          ; Correct for active low output
      move.b      d0,(a0)     ; Write data
      not.b       d0
      move.b      #D_Out+Set_SEL,(a1)     ; Pulse SEL to clock FF's
      nop                     ; Allow a bit extra pulse width
      move.b      #D_Out,(a1) ;
      nop                     ; Wait for signals to stabilize
      move.b      (a2),d1     ; Read control reg into d1 again
      btst        #1,d1       ; Test bit 1, corresponding to ATN
      bne         skip        ; Skip next statement if bit is set
      bchg        d3,d0       ; Clear the data bit if -Vout too large
skip: dbra        d3,loop     ; Do next bit
      move.b      #D_Out,(a1) ; Done! Clear BSY
      jsr         hexout      ; Now output the number to the screen
      jsr stop
      end
```

BSET Test a Bit and Set BSET
(M68000 Family)

Operation: TEST (<bit number> of Destination) ♦ Z; 1 ♦ <bit number> of Destination

Assembler Syntax:
BSET Dn,<ea>
BSET #<data>,<ea>

Attributes: Size = (Byte, Long)

Description: Tests a bit in the destination operand and sets the Z condition code appropriately, then sets the specified bit in the destination operand. When a data register is the destination, any of the 32 bits can be specified by a modulo 32-bit number. When a memory location is the destination, the operation is a byte operation, and the bit number is modulo 8. In all cases, bit zero refers to the least significant bit. The bit number for this operation can be specified in either of two ways:

1. Immediate—The bit number is specified in the second word of the instruction.
2. Register—The specified data register contains the bit number.

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

X—Not affected.
N—Not affected.
Z—Set if the bit tested is zero; cleared otherwise.
V—Not affected.
C—Not affected.

BTST Test a Bit BTST
(M68000 Family)

Operation: TEST (<bit number> of Destination) ♦ Z

Assembler Syntax:
BTST Dn,<ea>
BTST #<data>,<ea>

Attributes: Size = (Byte, Long)

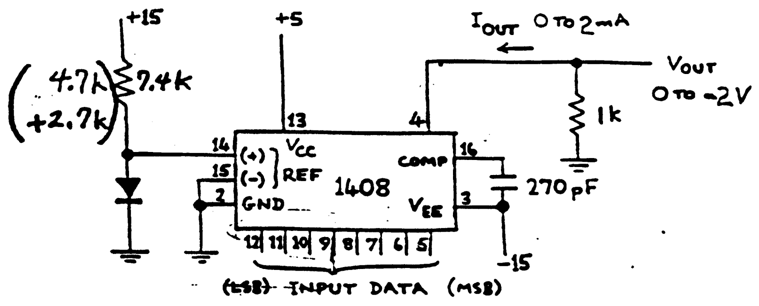
Description: Tests a bit in the destination operand and sets the Z condition code appropriately. When a data register is the destination, any of the 32 bits can be specified by a modulo 32-bit number. When a memory location is the destination, the operation is a byte operation, and the bit number is modulo 8. In all cases, bit zero refers to the least significant bit. The bit number for this operation can be specified in either of two ways:

1. Immediate—The bit number is specified in a second word of the instruction.
2. Register—The specified data register contains the bit number.

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

X—Not affected.
N—Not affected.
Z—Set if the bit tested is zero; cleared otherwise.
V—Not affected.
C—Not affected.



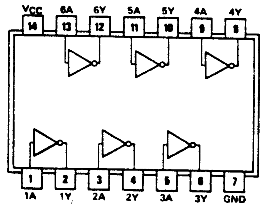
1408 8-bit monolithic current switching digital-analog converter. The current output is converted to a voltage by a 1k resistor.

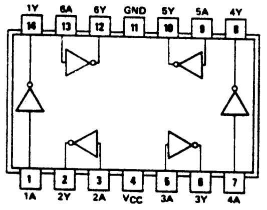
HEX INVERTERS

04

positive logic:
Y = $\overline{A}$

See page 6-2



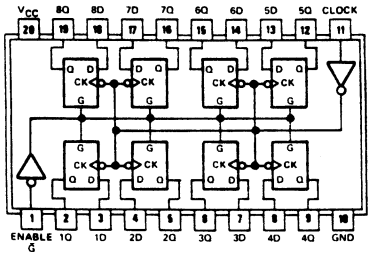


SN5404 (J)	SN7404 (J, N)	SN5404 (W)
SN54H04 (J)	SN74H04 (J, N)	SN54H04 (W)
SN54L04 (J)	SN74L04 (J, N)	SN54L04 (T)
SN54LS04 (J, W)	SN74LS04 (J, N)	
SN54S04 (J, W)	SN74S04 (J, N)	

OCTAL D-TYPE FLIP-FLOPS

377 SINGLE-RAIL OUTPUTS
COMMON ENABLE
COMMON CLOCK

See page 7-481

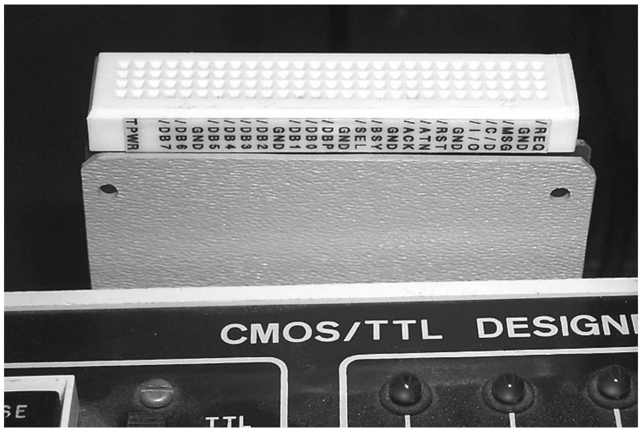


VCC 8Q 8D 7D 7Q 6Q 6D 5D 5Q CLOCK
20 19 18 17 16 15 14 13 12 11
D Q
CK
G
1 2 3 4 5 6 7 8 9 10
ENABLE $\overline{G}$ 1Q 1D 2D 2Q 3Q 3D 4D 4Q GND

SN54LS377 (J)   SN74LS377 (J, N)

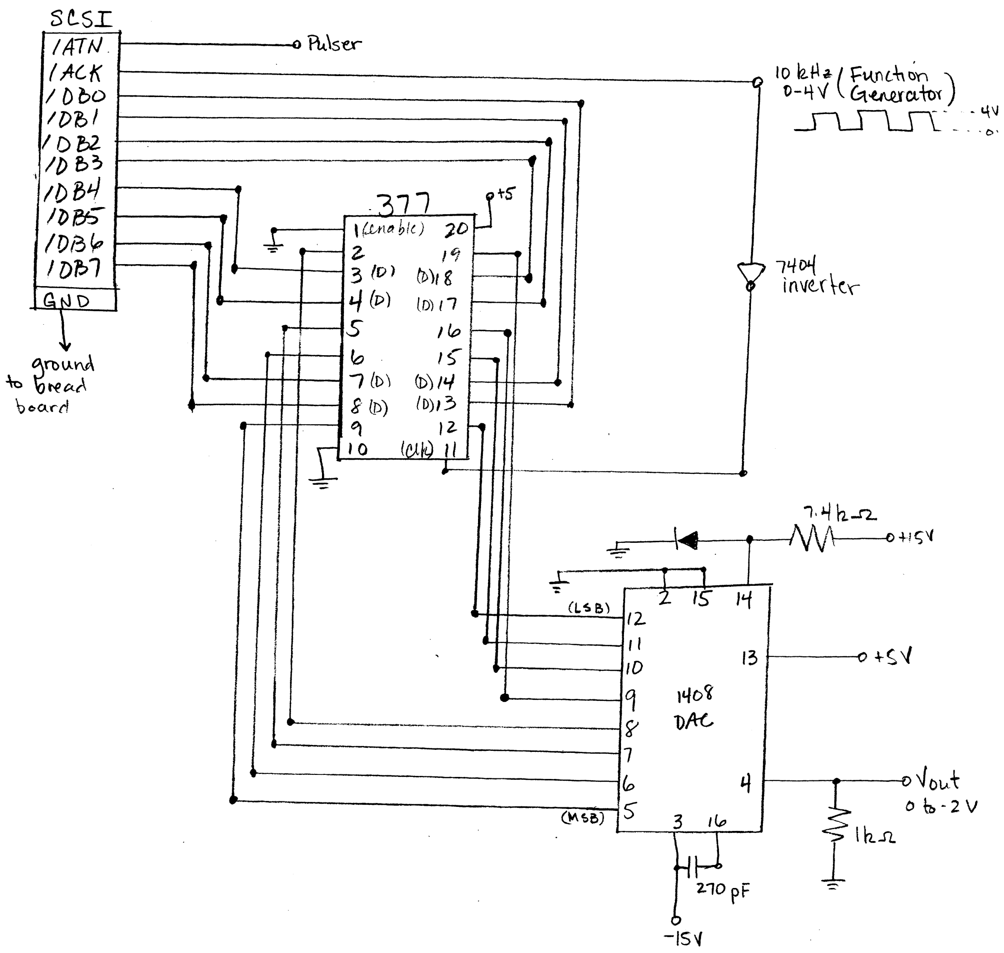
2 figs. above

Right: SCSI interface socket attached to the lab test unit. This is the connector marked "SCSI" in the diagram below. Wires can be plugged into the holes and run to the breadboard.



Below: Circuit Diagram for Digital Waveform Synthesizer

Follow instructions for part 3 on p. 39



Note:
Pin 1 on 377 is grounded to always enable the flip flops
The clk signal to the flip flops is inverted because the f-f's "toggle" on trailing edge (see pinout in handout)