Tiffany Q. Liu
April 11, 2011
CSC 270
Lab #10

Lab #10: Building Output Ports with the 6811

Introduction

The purpose of this lab was to build a 1-bit as well as a 2-bit output port with the 6811 training kit.

Materials

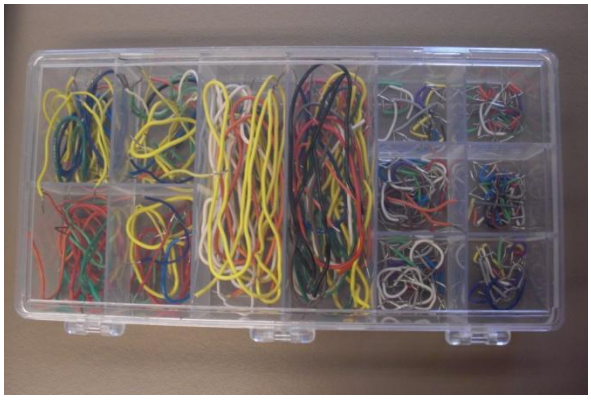

Figure 1. Wiring Kit.

Figure 2. 6811 Microprocessor Kit (Taken from D.Thiebaut).

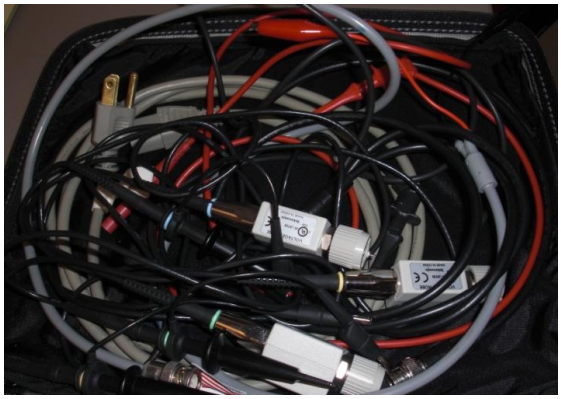

Figure 3. Oscilloscope Cables.

Figure 4. Tektronix Oscilloscope

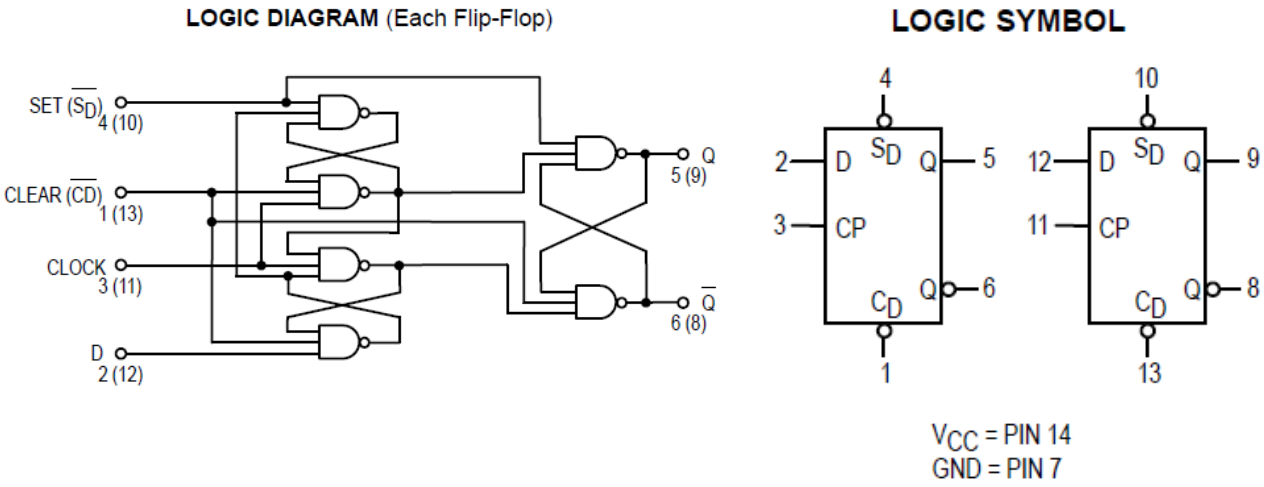

Figure 5. Dual D-Type Positive Edge-Triggered Flip-Flop 74LS74.

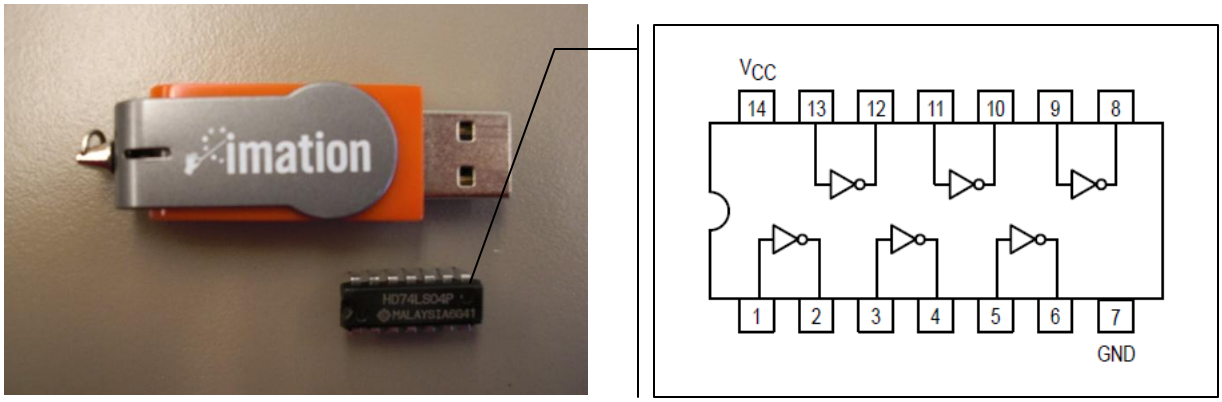

Figure 6. Hex Inverter 74LS04 Compared to a USB Flash Drive.

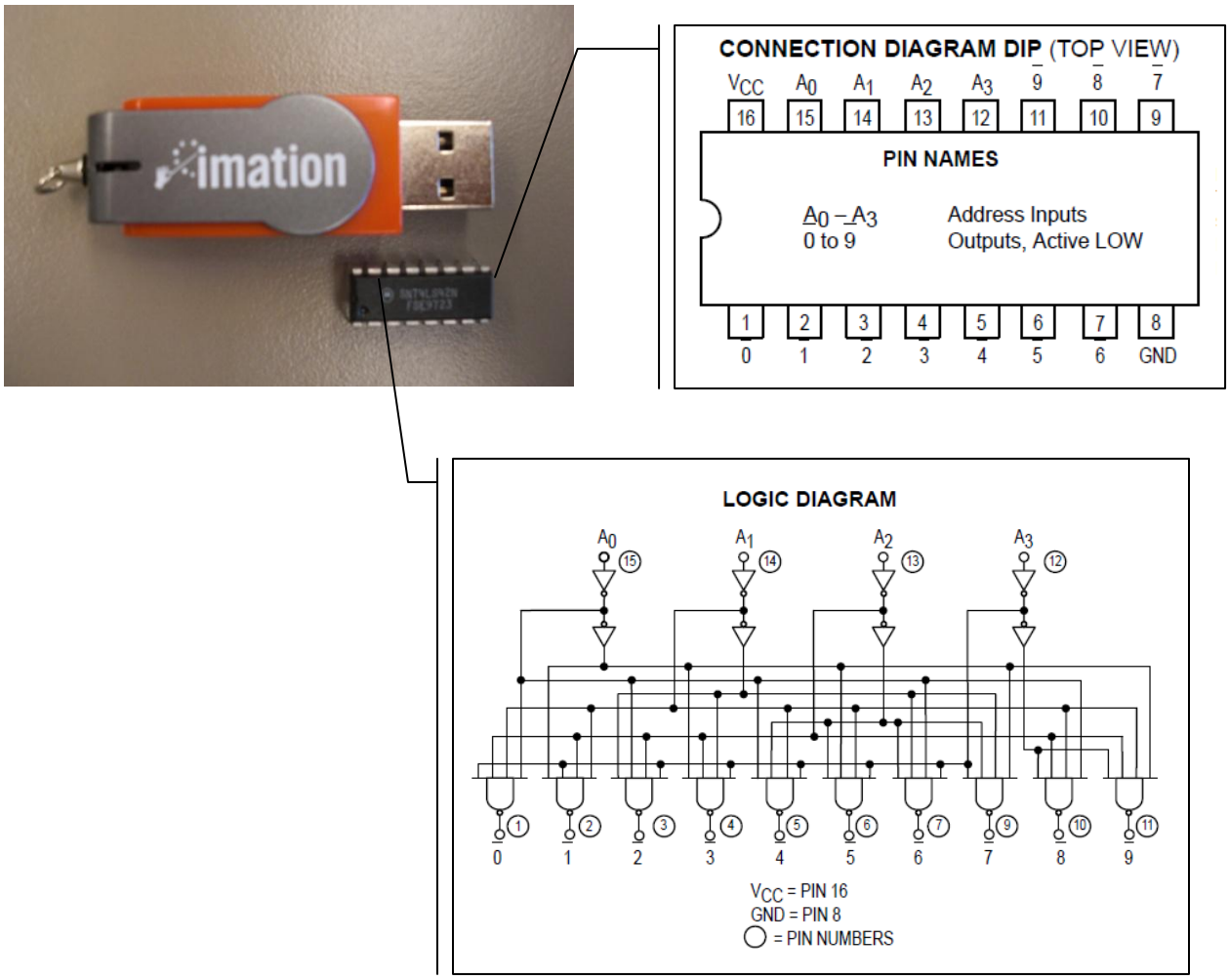

Figure 7. One-of-Ten Decoder 74LS42 Compared to a USB Flash Drive.

I/O Address

Before we began drawing our diagrams, we had to determine the range of addresses available to use for building ports (ie. determine the range of addresses that were not used for RAM, ROM, or I/O devices by the 6811). Using the Memory I/O Map chart for the kit, we determined that available addresses resided in the range named Memory Module Area (7000H to AFFFH).

Setup

When we figured out the range of addresses we could use as output ports, we then had to determine how to wire our circuit. First we connected D0 from the 6811 to the input D on the 7474 D flip-flop and output Q on the 7474 to an LED. Then we connected the negation of our clock signal E to D (A3) on the 7442 4-to-10 decoder. If we take a look at the truth table for the 7442, we can see that when D is low, the decoder will generate outputs for y0 to y7, when D is high, it will not generate outputs, which is the behavior we are looking for.

A_0	A_1	A_2	A_3	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{5}$	$\overline{6}$	$\overline{7}$	$\overline{8}$	$\overline{9}$
L	L	L	L	L	H	H	H	H	H	H	H	H	H
H	L	L	L	H	L	H	H	H	H	H	H	H	H
L	H	L	L	H	H	L	H	H	H	H	H	H	H
H	H	L	L	H	H	H	L	H	H	H	H	H	H
L	L	H	L	H	H	H	H	L	H	H	H	H	H
H	L	H	L	H	H	H	H	H	L	H	H	H	H
L	H	H	L	H	H	H	H	H	H	L	H	H	H
H	H	H	L	H	H	H	H	H	H	H	L	H	H
L	L	L	H	H	H	H	H	H	H	H	H	L	H
H	L	L	H	H	H	H	H	H	H	H	H	H	L
L	H	L	H	H	H	H	H	H	H	H	H	H	H
H	H	L	H	H	H	H	H	H	H	H	H	H	H
L	L	H	H	H	H	H	H	H	H	H	H	H	H
H	L	H	H	H	H	H	H	H	H	H	H	H	H
L	H	H	H	H	H	H	H	H	H	H	H	H	H
H	H	H	H	H	H	H	H	H	H	H	H	H	H

H = HIGH Voltage Level
L = LOW Voltage Level

Figure 8. Truth Table for a 74LS74.

Next we connected A13, A14, and A15 on the 6811 to pins 15, 14, and 13 on the decoder respectively. By doing so, we can figure out which yi outputs is activated for an address in the range 7000H to AFFFH. y0 is activated for the address 000XXXXXXXXXXXX, which gives the range 0000H to 1FFFH; y1 is activated for the address 001XXXXXXXXXXXX, which gives the range 2000H to 3FFFH. From this pattern, we determined that y4 is activated when the address falls in the range 8000H to 9FFFH, which lies within the address range we can use. With this information, we can connect y4 to the clock of our 7474 D flip-flop. The following is a wiring diagram of our circuit followed by the timing diagram that corresponds to this circuit:

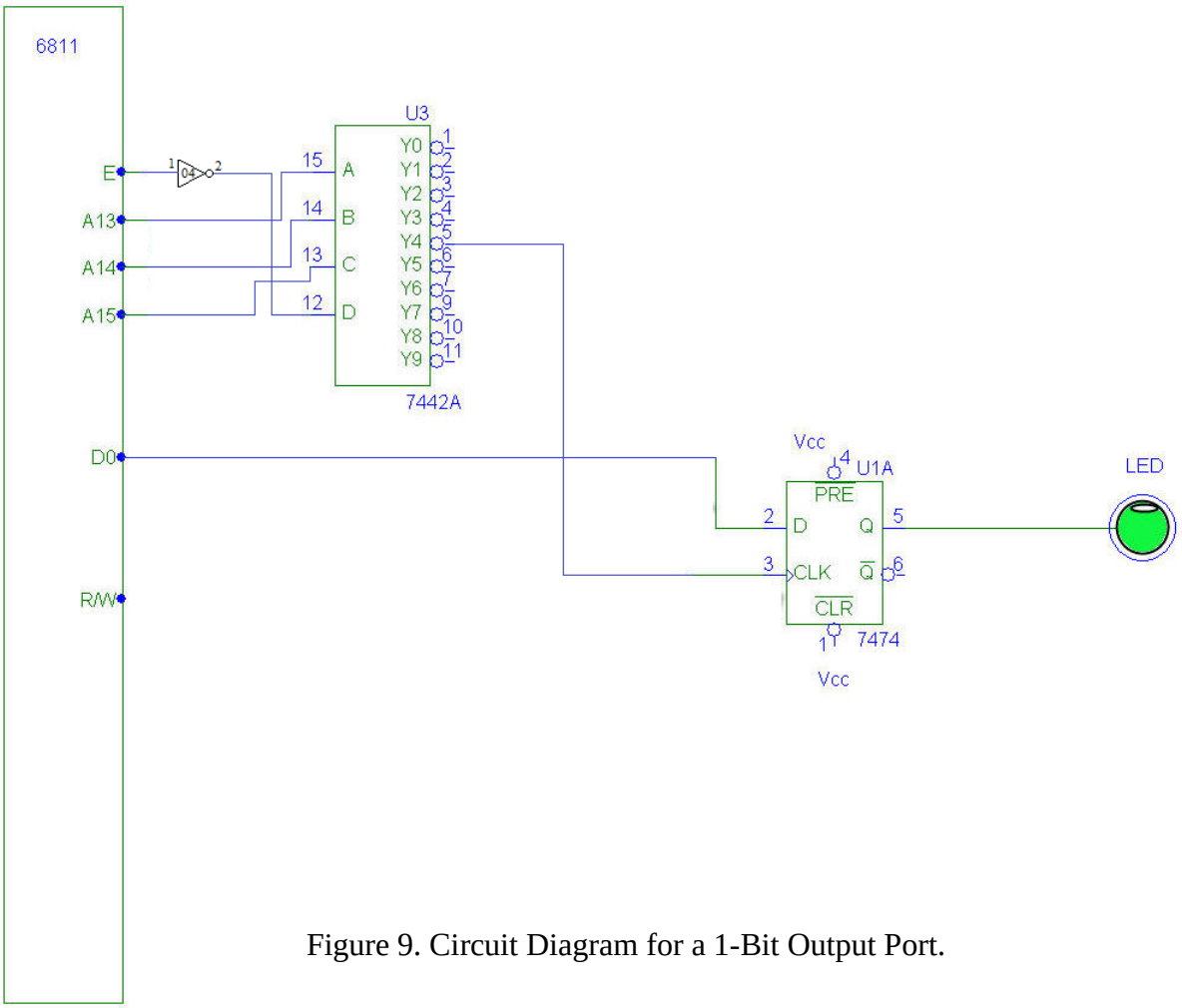

Figure 9. Circuit Diagram for a 1-Bit Output Port.

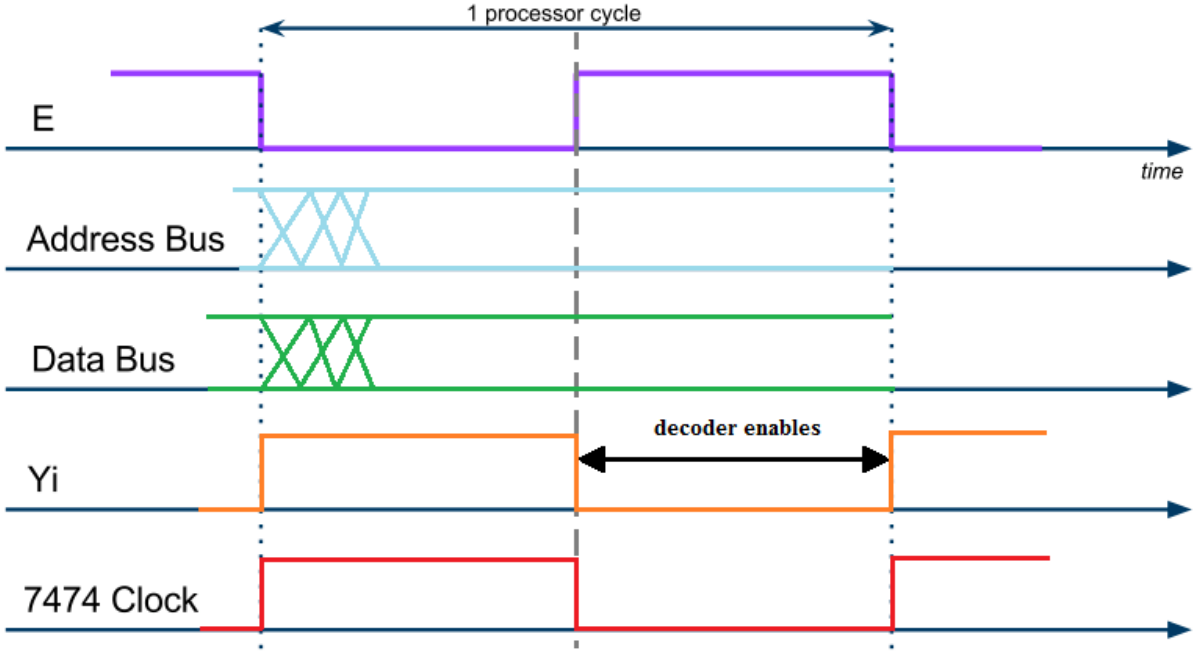

Figure 10. Timing Diagram for the Circuit in Figure 9.

Testing the Circuit

Once we had our instructor verify our schematics, we wired up our circuit according to Figure , making sure that the 7442 was placed as far left on the breadboard as possible since it we want to keep it wired to be reused in future labs. We then wrote and assembled a program that was a simple endless loop that repeatedly wrote 0, then 1, then 0, then 1, etc. in the 1-bit flip-flop:

```
                    ;--- data section ---

                    ;--- code section ---
                           ORG   0000
0000 86 00          LOOP:  LDAA  #00        ; ACCA <- 0
0002 B7 80 00              STAA  8000        ; mem[8000] <- 0
0005 86 00                 LDAA  #01        ; ACCA <- 1
0007 B7 80 00              STAA  8000        ; mem[8000] <- 1
000A 7E 00 00              JMP   LOOP        ; infinitely loop through prog.
```

This program takes a total of 15 cycles per loop: 2 for each LDAA, 4 for each STAA, and 3 for the jump. We then entered our assembled code into the 6811 kit and ran it. Since the frequency at which the 0's and 1's were being stored was so fast, we were unable to detect the LEDs blinking. To see the signals that were being generated, we used the scope to look at R/W' and y4. We obtained the following screenshot:

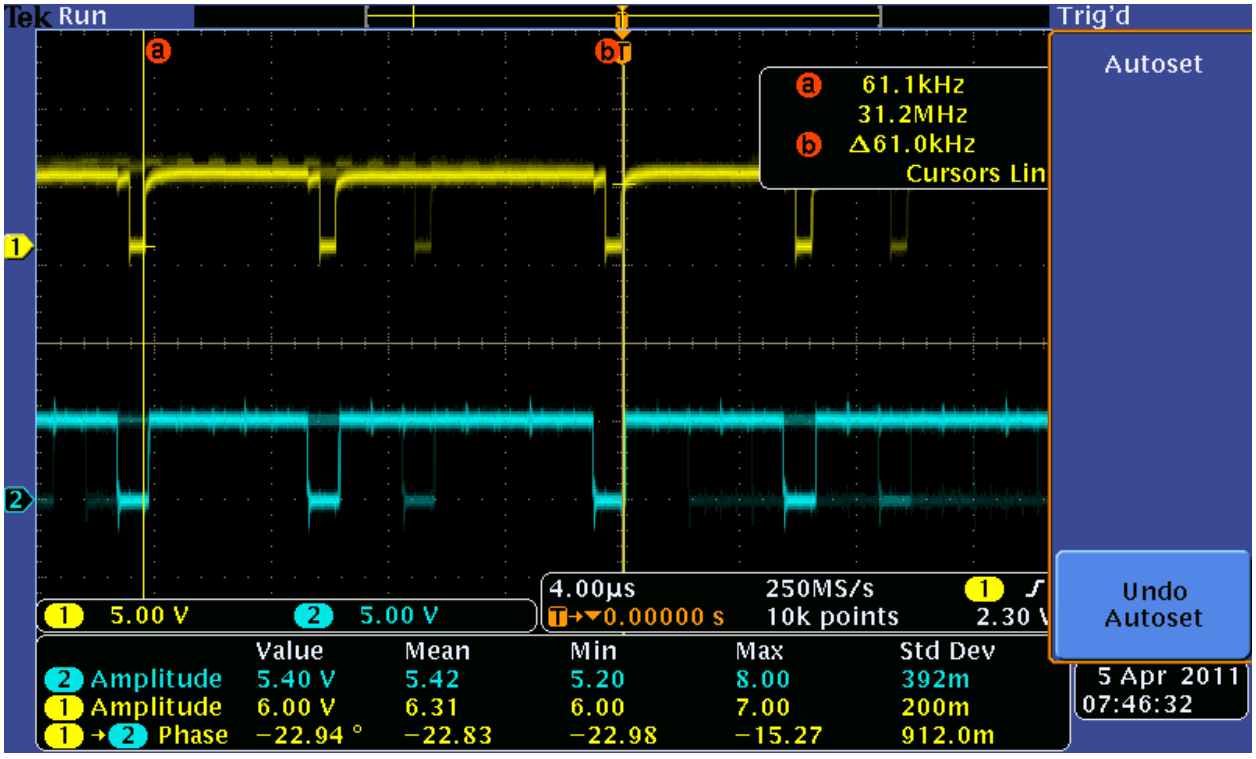

Figure 11. Screen Capture of R/W' (Blue) and y4 (Yellow) Signals.

This screenshot of the R/W' and y4 signals appeared to be correct since, the y4 output gets activated at the last part of the write, which was what was expected. The signals between the two

yellow cursors in the figure correspond to one period of the signals, which measured to 16.0µs. Knowing that 5 instructions were executed in that period, we can calculate the number of instructions that the processor performed per second: 5 instructions / 16.0µs = 312,500 instructions per second, which is equivalent to 0.313 MIPS. The signals did not appear to have a 50% duty cycle as a result of the jump instruction at the end of the loop.

We then added a scope probe to the Q output of the flip-flop (which indicated the bit we were storing in the 1-bit I/O port implemented by the flip-flop), and obtained the following screenshot:

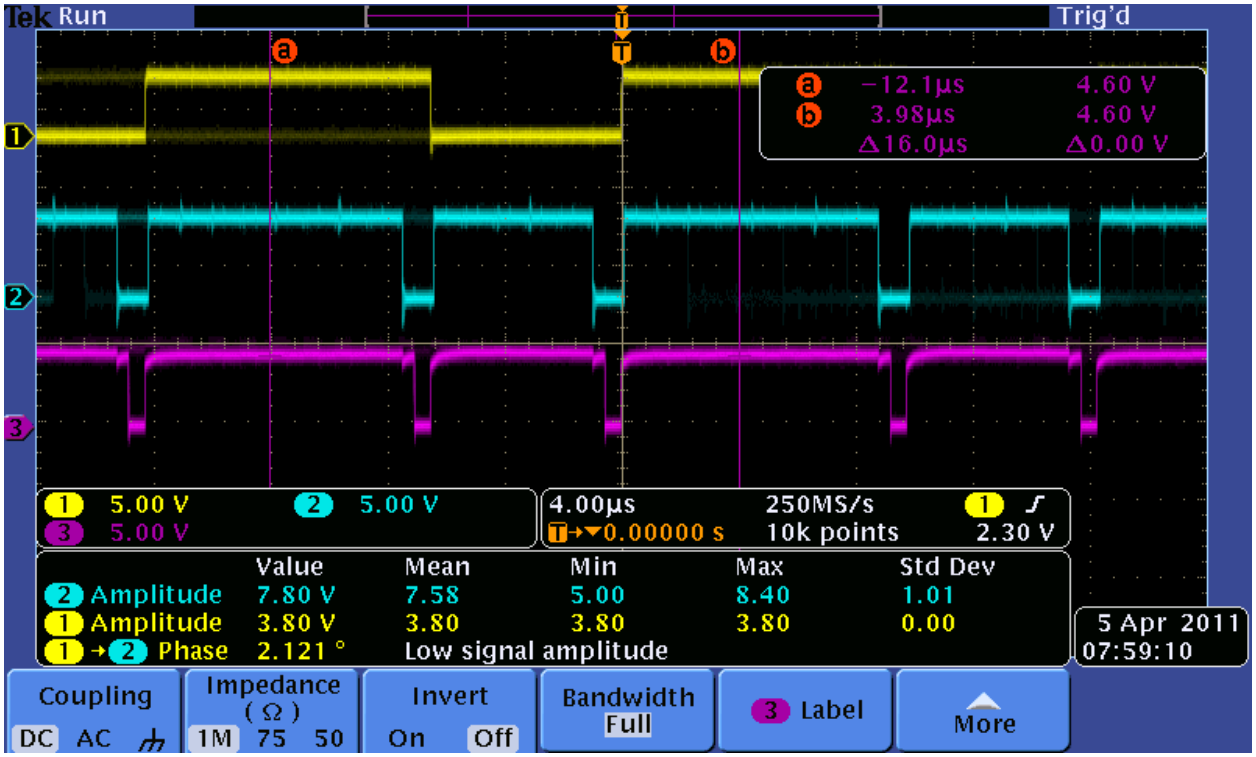

Figure 12. Screen Capture of R/W' (Blue), y4 (Purple), and Q Output (Yellow) Signals.

Again, in this screenshot, one execution of the loop can be detected by the signals within the two cursors. This timing diagram makes sense with what we are trying to do since after the R/W' signal (Blue) goes low (W), the Q output signal (Yellow) goes low (indicating that a 0 was set as the output) and after the R/W' signal goes low for a second time in the period, the Q output signal goes high (indicating that a 1 was set as the output).

A Software Driver

By creating two functions that can be called to set the LED on and turn it off, we can write a simple driver:

```
                ;--- data section ---
                        ORG   0020
0020 80 00      LED     EQU   8000

                ;--- code section ---
                        ORG   0000
                LOOP:
0000 9D 10              JSR   turnLEDOn  ; go to subroutine to turn on LED
0002 9D 16              JSR   turnLEDOff ; go to subroutine to turn off LED
0004 7E 00 00           JMP   LOOP       ; run program infinitely

                        ORG   0010
                turnLEDOn:
0010 86 01              LDAA  #01        ; ACCA <- 1
0012 B7 80 00           STAA  LED        ; LED <- 1
0015 39                 RTS              ; return to call of subroutine

                turnLEDOff:
0016 86 00              LDAA  #00        ; ACCA <- 0
0018 B7 80 00           STAA  LED        ; LED <- 0
001B 39                 RTS              ; return to call of subroutine
```

When we entered our assembled program into the kit, we got the following screenshot of the Q (Yellow), R/W’ (Blue), and y4 (Purple) signals:

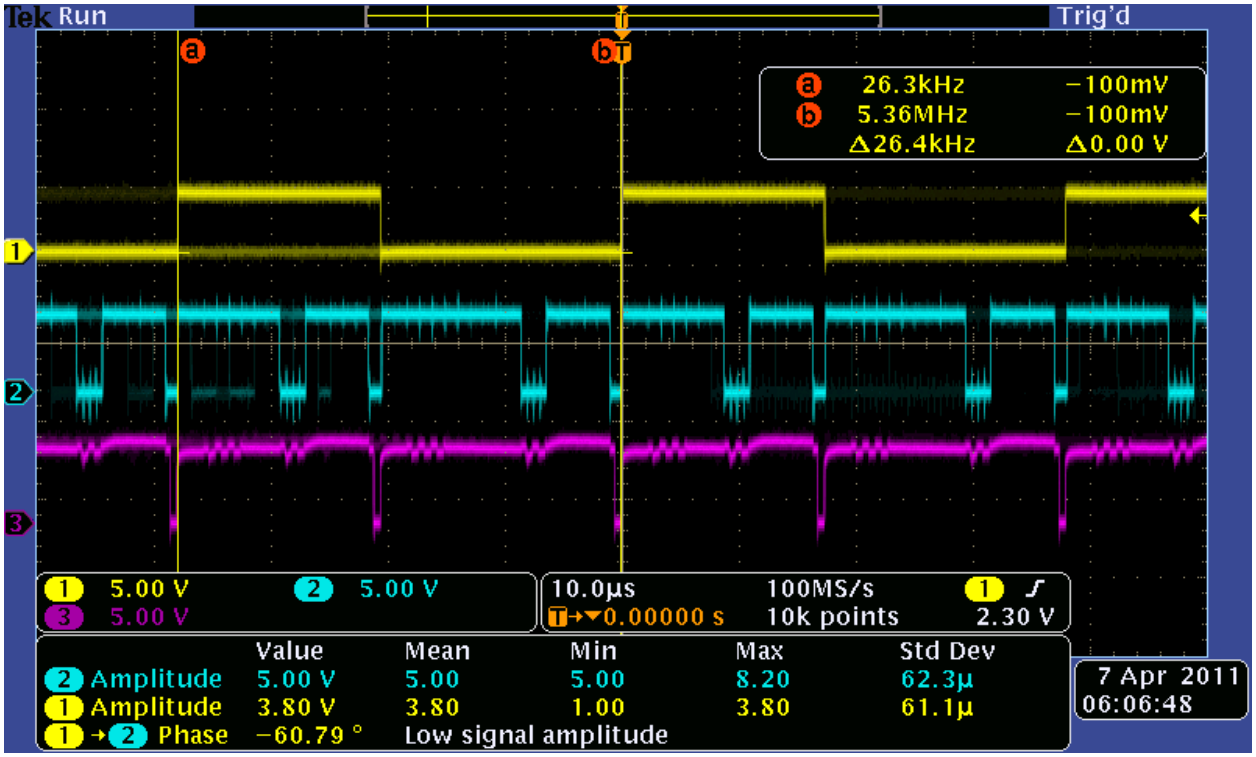

Figure 13. Screen Capture of the R/W' (Blue), y4 (Purple), and Q Output (Yellow) Signals from Driver.

As with the previous captures, the signals within the two cursors are the signals generated by one execution of the infinite loop. This figure is similar to Figure with some slight differences. The y4 signal (Purple) is still activated during the last part of the write signal (Blue going low). In addition we can see the Q output going from high to low to high to low, etc immediately after the R/W' signals goes low (writing). However, we noticed that there are two more write signals within one execution of the loop than there were in Figure . This is the result of adding the JSR and RTS instructions, which write to memory when the Program Counter gets pushed to the stack.

Using the scope, we measured the frequency of the Q output to be 26.4kHz, which indicates the frequency at which the LED blinks. By adding JSR and RTS instructions to our code, we actually increased the number of cycles a loop takes (decreasing the frequency at which our LED can blink). To increase the frequency at which we can make the LED blink, we can load 1 into ACCA, and 0 into ACCB outside of the loop and inside the loop store ACCA to 8000 followed by a store ACCB to 8000 and repeat these two instructions 100 times before looping through the block of instructions. This way we can average out the number of cycles executed within the loop: STAA takes 4 cycles and STAB takes 4 cycles = 8 cycles * 100 times + 3 cycles for the jump instruction = 803 cycles / 100, which is approximately 8 cycles to make the LED blink. From the previous section we found that it took 16.0µs to execute 15 cycles, which corresponds to about 1.07µs to execute 1 cycles → 8.53µs to execute 8 cycles, which corresponds to a frequency of 117.2kHz.

Slowing Down the Loop

Since the frequency at which the LED was blinking was too fast for our eyes to detect, we have to slow down the setting of the LED to actually see the LED turning off and on consistently. To do this, we can add delays between our calls to each of the subroutines for turning on the LED and turning off the LED. These delays can be called by accessing an entry point in the Monitor that allows the processor to wait for 1 second before executing the next instruction. This is located at address location C027 of the EEPROM. With this information we wrote and assembled the following:

```
                ;--- data section ---
                        ORG   0020
0020 80 00      LED     EQU   8000

                ;--- code section ---
                        ORG   0000
                LOOP:
0000 9D 10              JSR   turnLEDOn  ; go to subroutine to turn on LED
0002 BD C0 27           JSR   pause      ; wait 1 second
0005 9D 16              JSR   turnLEDOff ; go to subroutine to turn off LED
0007 BD C0 27           JSR   pause      ; wait 1 second
000A 7E 00 00           JMP   LOOP       ; run program infinitely

                        ORG   0010
                turnLEDOn:
0010 86 01              LDAA  #01        ; ACCA <- 1
0012 B7 80 00           STAA  LED        ; LED <- 1
0015 39                 RTS              ; return to call of subroutine

                turnLEDOff:
0016 86 00              LDAA  #00        ; ACCA <- 0
0018 B7 80 00           STAA  LED        ; LED <- 0
001B 39                 RTS              ; return to call of subroutine
```

When we entered and ran the above program, we were able to see the LED blinking.

Storing 1-Bit

In this lab, we designed a 1-bit output port that allowed us to see the least significant data bit stored in the data section of the processor via an LED (by storing the bit in the D flip-flop). However, once the bit is stored in the flip-flop, we cannot read it back from our program since we never connected anything to the Q output that would allow us to store the value of Q (only connected a LED).

Since the processor cannot discern our designed 1-bit port from a memory location that is has access to, we should theoretically be able to store a 1 or a 0 in the 1-bit output port via the keyboard. To do this we would access memory location 8000 using the keyboard and enter either 00 to set the bit to 0 or 01 to set the bit to 1. However, when we tried to do this with our kit, we were unable to store the values we entered.

Two-Bit Port

Since the 74LS74 contains two flip-flops, we can create a 2-bit parallel-output port without any addition to the hardware already on the breadboard. The following is a circuit diagram that can be used to implement a 2-bit parallel-output port:

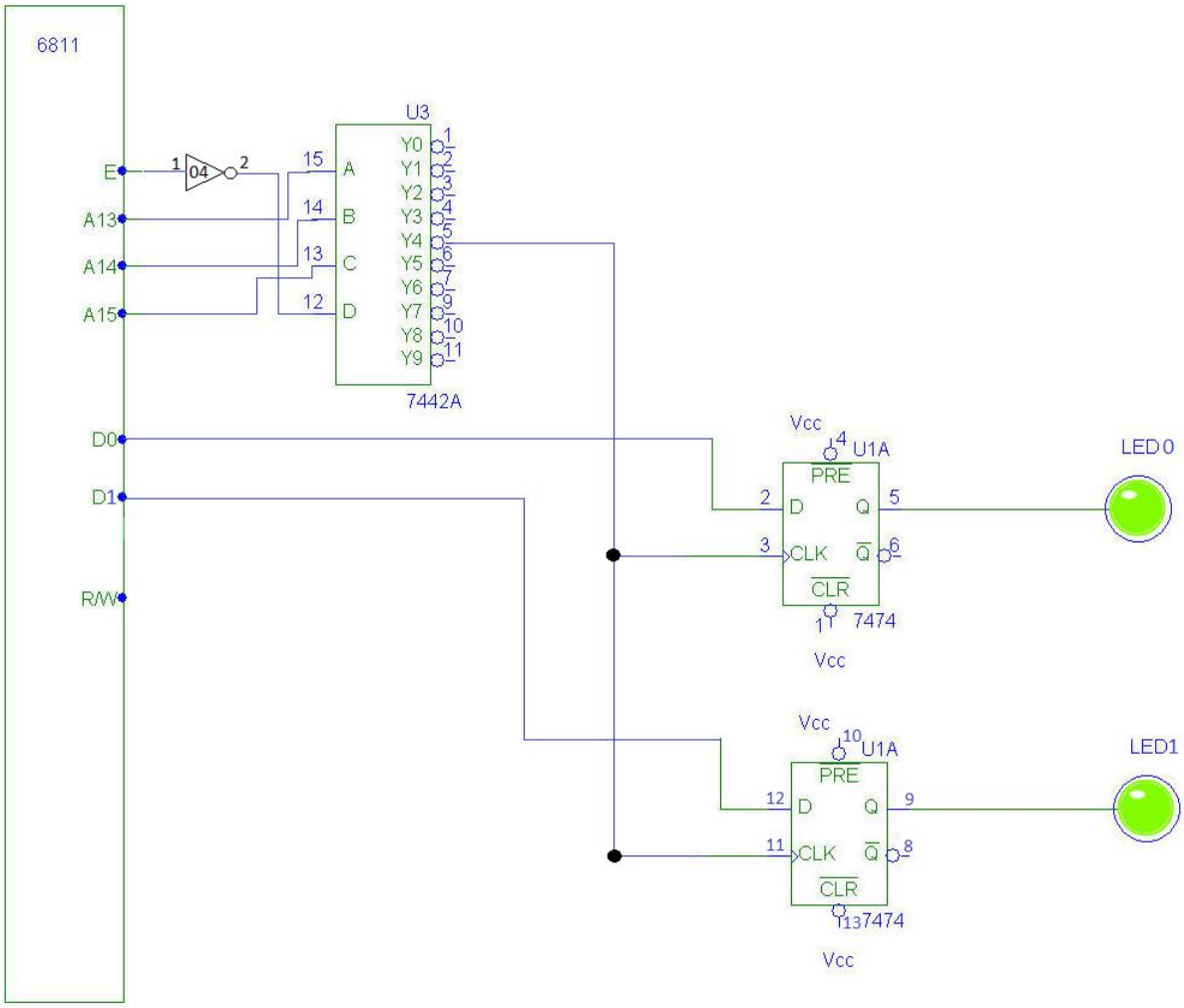

Figure 14. Circuit Diagram for a Parallel-Write 2-Bit Output Port.

To demonstrate that our 2-bit port works, we wrote and assembled the following program to make the two LEDs count in binary (00, 01, 10, 11, 00, etc.):

```
                ;--- data section ---
                       ORG   0020
0020 80 00      LED    EQU   8000

                ;--- code section ---
                       ORG   0000
                LOOP:
0000 86 00             LDAA  #00        ; ACCA <- 0
0002 B7 80 00          STAA  LED        ; LED1 <- 0, LED0 <- 0
0005 BD C0 27          JSR   pause      ; wait 1 second
0008 86 01             LDAA  #01        ; ACCA <- 1
000A B7 80 00          STAA  LED        ; LED1 <- 0, LED0 <- 1
000D BD C0 27          JSR   pause      ; wait 1 second
0010 86 02             LDAA  #02        ; ACCA <- 2
0012 B7 80 00          STAA  LED        ; LED1 <- 1, LED0 <- 0
0015 BD C0 27          JSR   pause      ; wait 1 second
0018 86 03             LDAA  #03        ; ACCA <- 3
001A B7 80 00          STAA  LED        ; LED1 <- 1, LED0 <- 1
001D BD C0 27          JSR   pause      ; wait 1 second
0020 7E 00 00          JMP   LOOP       ; run program infinitely
```

When we entered the above program into our kit, we were able to see that the LEDs blinked in the following fashion:

LED1 = off, LED0 = off
LED1 = off, LED0 = on
LED1 = on, LED0 = off
LED1 = on, LED0 = on
LED1 = off, LED0 = off
…and so on, infinitely

When we tried using a program that would increment ACCA every time after the first load instead of loading an immediate value into ACCA every time, the LEDs did not behave the way we expected them to (as they did above). We discovered that this was because the JSR instruction modifies the values in the accumulators, which would require us to either push the value in the accumulator onto a stack or store it into a temporary variable before the JSR instruction and load the value back into the accumulator with either a pop from the stack or a direct/extended load respectively.