FINAL PROJECT: INTERFACING AND MOTOR CONTROL

In this sequence of labs you will learn how to interface with additional hardware and implement a motor speed control system.

WEEK 1

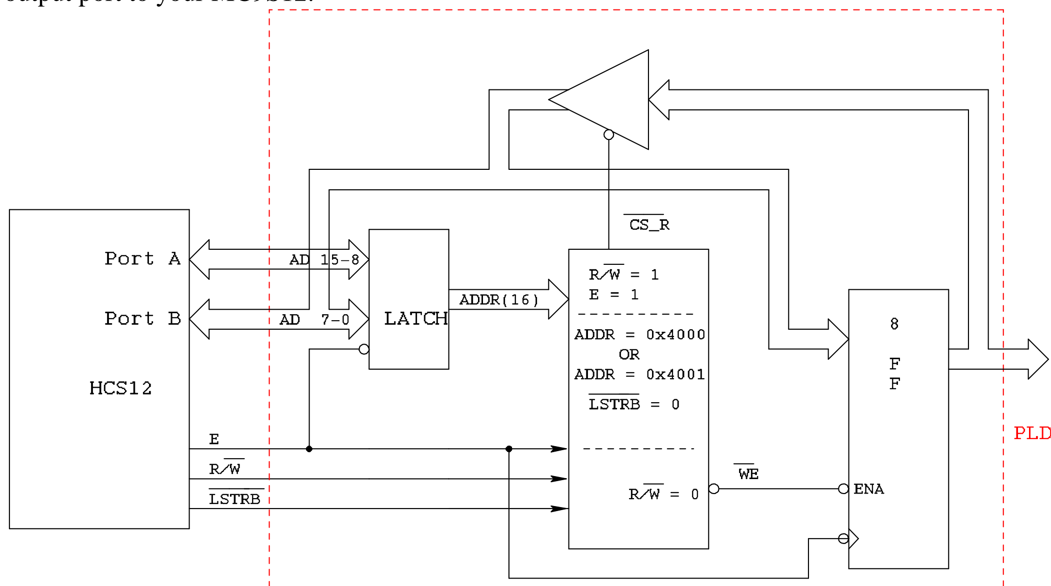
PORT EXPANSION FOR THE MC9S12

Pre-Lab

1. Write a Verilog program to implement an eight-bit output port at address 0x4001 and an eight-bit input port at address 0x4000.

Introduction and Objectives

It is sometimes necessary to add additional memory and/or hardware to a microprocessor or microcontroller. While interfaces such as the SPI allow you to add some hardware, it is often necessary to interface directly to the address/data bus. For a microprocessor, which does not have built-in peripherals, the address/data bus is the only way to add additional memory or hardware. In this lab you will add an extra output port to your MC9S12.




```
Writing to address 0x4001 (ADDR = 0x4000 or 0x4001, LSTRB low, R/W low) will bring WE low.
On the high-to-low transition of E with WE low, the data into the flip-flops

Reading from address 0x4001 (ADDR = 0x4000 or 0x4001, LSTRB low, R/W high, E high) will bring CS_R lc
This will drive the data from the flip-flops onto the data bus
The HC12 will read the data on the flip-flops on the high-to-low transition of the E-clock
```

Figure 1. Block diagram of HCS12 output port at address 0x4001

Figure 1 shows a block diagram for adding an external output port at address 0x4001 to the microcontroller. You will implement the port in your Altera FPGA board from EE 231. Note that you will have to connect the 16-bit multiplexed address/data bus and three control lines from your MC9S12 to your

Altera board. You will have the eight bits of your output port at 0x4001 control the LEDs on your Altera board to verify that the output port is working. Your Verilog program should also implement an eight-bit input port at address 0x4000. Connect the four switches on the Altera board to the input port at 0x4000 so you can see that the input port works.

1. Write an Verilog program to implement the expansion ports.

2. Assign the pins for the Altera chip so you can connect the sixteen address/data lines and three control lines to your MC9S12. Also, assign the pins so that the output port controls the LEDs on the Altera board, and the four least significant bits of the input port are connected to the switches of the Altera board. Note that there will be a lot of wires to run, so it is essential that you are neat in your wiring. Be sure to assign the E input to pin J-13 (CLK3 of the EE 231 board) or K-13 (CLK2). These are **global clock** inputs, which means the E clock will be routed directly to the clock input of the flip-flops and latches without any delay of going through other logic.

3. Check the functioning of your port using D-Bug12. When you start your MC9S12 using D-Bug12, the microcontroller is in single chip mode. In this mode you can manipulate AD-15-0, E, $R/\overline{W}$ and $\overline{LSTRB}$ as general purpose I/O lines. You can use the MM command of D-Bug12 to write data to the output port by changing AD15-0 (PORTA and PORTB), E, $R/\overline{W}$ and $\overline{LSTRB}$ in the same sequence that the MC9S12 would if it were in expanded wide mode.

a) Use the DDRE register to make E, $R/\overline{W}$ and $\overline{LSTRB}$ output pins. (Note: E is bit 4 of PORTE, R/W is bit 2 of PORTE, and LSTRB is bit 3 of PORTE.
b) Bring E low by writing to PORTE.
c) Put 0x4001 on PORTA and PORTB.
d) Bring $R/\overline{W}$ and $\overline{LSTRB}$ low.
e) Bring E high.
f) Put the data you want to write to the port on PORTB.
g) Bring E low.

4. The program below can be put into EEPROM so you can run your board in wide expanded mode. To get into wide expanded mode, you will have to put this program into EEPROM starting at address 0x0400, and then set DIP Switch SW7 so you run your EEPROM program rather than DBug12. (You cannot run in expanded wide mode using DBug12, since DBug12 uses the Flash EEPROM in the region 0x4000-0x7fff.)

```c
#include "hcs12.h"
#include "DBug12.h"
#include "vectors12.h"

#define IN_PORT  (*(volatile char *) 0x4000)
#define OUT_PORT (*(volatile char *) 0x4001)

void INTERRUPT RTI_isr(void);
volatile int done;


main()
{
/* Set bus clock to 24 MHz */
    asm(" sei;");
    CLKSEL &= ~0x80;
    PLLCTL |= 0x40;
    SYNR = 0x05;
    REFDV = 0x01;
    while ((CRGFLG & 0x08) == 0) ;
    CLKSEL |= 0x80;

/* Put MC9S12 into wide expanded mode */
    MODE = 0xe8;                /* Expanded wide mode, IV on */
    PEAR = 0x0c;                /* Turn on R/W, LSTRB, E */
    EBICTL = 0x01;              /* Use E-clock to control external bus */
    MISC = 0x03;                /* No E-clock stretch, disable ROM from
                                   4000-7FFF */

    DDRP = DDRP | 0x0F;       /* Make 4 LSB of Port P outputs */
    PTP = PTP | 0x0F;         /* Turn off seven-seg LEDs      */

/* Set up SCI for using DB12FNP->printf() */
    SCI0BDH = 0x00;            /* 9600 Baud */
    SCI0BDL = 0x9C;
    SCI0CR1 = 0x00;
    SCI0CR2 = 0x0C;            /* Enable transmit, receive */

/* Set up RTI to increment 0x4001, and display 0x4000 on the computer
 * terminal */
    UserRTI = (unsigned short) &RTI_isr;
    RTICTL = 0x13;             /* 131 ms rate */
    CRGINT |= BIT7;            /* Enable RTI interrupt */

    asm(" cli");

    DB12FNP->printf("hello, world\r\n");

    for (;;) {
        while (!done) ;
        DB12FNP->printf("switches = %x, LEDs = %x\r\n",
                IN_PORT & 0x0f, OUT_PORT & 0xff);
        done = 0;
    }
}

void INTERRUPT RTI_isr(void)
{

    OUT_PORT = OUT_PORT + 1;
    done = 1;
    CRGFLG = BIT7;
}
```

WEEKS 2 and 3

MOTOR SPEED CONTROL

Introduction and Objectives

In this lab you will control the speed of a motor. Figure 1 shows the hardware setup, which is the same as for Week 1 of Lab 4. You will use the potentiometer on your evaluation board to set the desired speed of the motor, and you will control the speed through the PWM output of the HCS12. You will measure the speed of the motor using an input capture pin, and display the desired and actual speeds on the terminal.

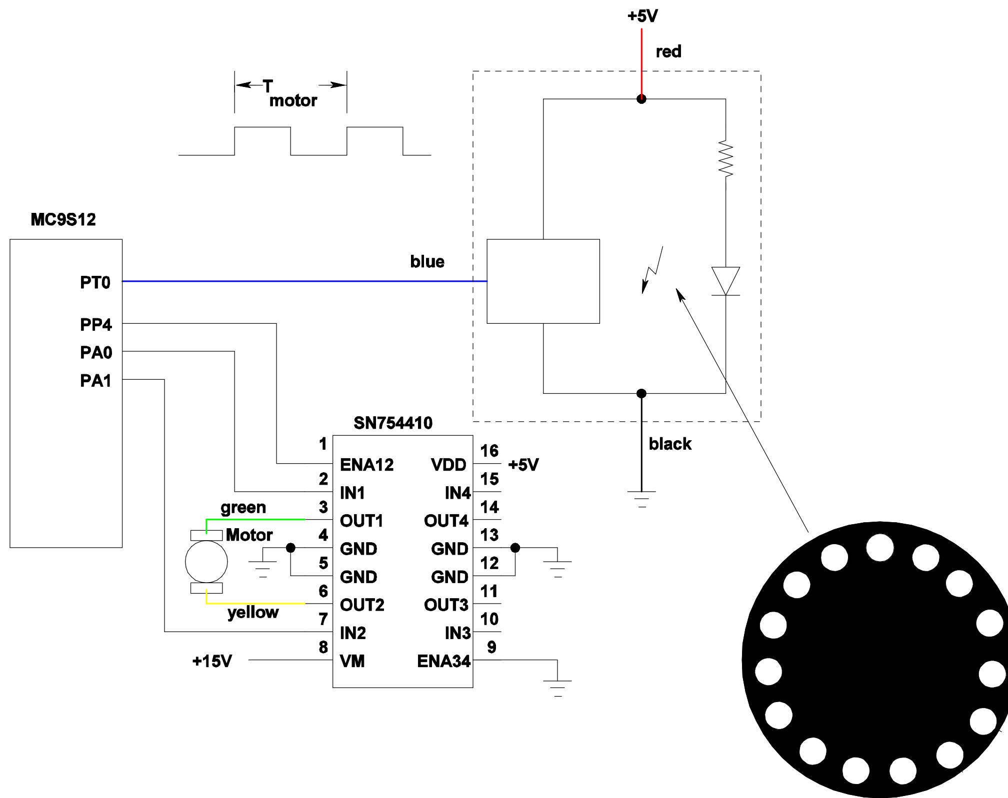


Figure 1. Block diagram of HCS12 output port at address 0x4055

1. Build the circuit shown in Figure 1.

2. Set up the RTI to generate an interrupt once every 8 ms. In the interrupt service routine, increment LEDs connected to Port A. Verify from the rate at which the LEDs are incrementing that you are getting interrupts at a rate of 8 ms.

3. Program the A/D converter to read the value from the pot. Use 8-bit A/D mode. In your RTI ISR, read the A/D converter, and write the eight most significant bits to Port A. In the main program loop, display the

value read on the LCD diplay. (Do not print inside the ISR – this will take more than 8 ms, and you will miss interrupts.) Verify that the A/D values change as expected as you use the pot to change the voltage.

4. Set up the PWM to generate a 50 kHz PWM signal on one of the four PWM channels. Set it up for high polarity. It will be easiest to set PWPERx to 255. Verify that the PWM works. In the RTI ISR, write the eight most significant bits to the A/D value you read to PWDTYx. The motor speed should change as you sue the pot to vary the voltage on the A/D.

5. Measure the speed of the motor. Set up an Input Capture interrupt to determine the time between two falling edges of the optical encoder. In your main program write display this time on the LCD display. You can use floating point arithmetic to convert this time into RPM. Display the RPM value on the LCD display.

6. Measure the speed for several different duty cycles by varying the voltage with the pot. Plot speed vs. duty cycle.

7. Implement closed-loop speed control. The desired speed Sd should be

$$Sd = (0.2 + 0.8\frac{AD}{AD\max})S\max$$

where Smax is the motor speed at 100% duty cycle, AD is the A/D converter reading, and ADmax is the maximum A/D converter reading. In this way you will be able to vary the speed between 20% and 100% of Smax.

To set the motor at the desired speed you can use a simple equation (proportional control) such as:

$$DCnew = DCold + k(Sd - Sm)$$

where Sm is the measured speed. Do this calculation inside the RTI ISR, and write the new value to PWDTYx. Try different values of k to see how the motor responds. If k is too small, it will take a long time for the motor to get to its steady-state speed. If k is too large, the motor will be jerky as it tries to settle down to its steady-state speed.

It will be much easier to do these calculations using floating point numbers rather than using integers. You can use floating point numbers with the GNU compiler. In the EmbeddedGNU IDE, select the Options menu, Project Options submenu. Near the bottom of the pop-up window, add the following to the Compiler options:

fshortdouble

By doing this, you will be able to do basic operations with floating point numbers (add, subtract, multiply, divide). Do not try to use functions which require the math library (such as sqrt()); the code generated by the Gnu compiler will be too large to fit into the HC9S12. To print out a floating point number you must first convert it to an integer. For example,

```
float x;
x = 10.2;
DB12FNP->printf("x = %d\r\n",(short) x);
```

If you use this method to print **x** when **x** is, say, 0.023, the value printed out will be zero. You could

use the following to print a usable value for **x**:

DB12FNP->printf("x = %d/1000\r\n",(short) (x*1000.0));

The output from this when **x** is 0.023 will be **23/1000**.

8. Measure the motor speed for various values of input voltage. Take about 10 equally-spaced measurements for input voltage between 0 and 5 V.

9. With the pot set at about mid-range, vary the voltage of the voltage powering the motor (say between 8 V and 14 V). With closed-loop control the speed of the motor should stay the same. Verify that this is the case.

10. Using the data from Part 8, plot the speed in RPM vs. the input voltage from the port – i.e., convert the speed measurement in time difference between two falling edges to speed in RPM.