EXPERIMENT NUMBER 9
Interfacing the 8051 with an External Sensor II

INTRODUCTION:

Almost all embedded systems require some kind of an interface to an external sensor. These sensors connect the system to the outside world, be it a temperature sensor, user-pushable button, or even an electronic gyroscope. Unfortunately for digital systems designers, there are nearly as many different digital interfaces as there are sensor types. They are differentiated by their speed, the number of signal wires, the timing specifications, and a host of other parameters. In this lab we will design an interface to a sensor using the Dallas 1-Wire protocol, where a single wire is used to send and receive all data.

The Dallas 1-Wire protocol allows several sensors to communicate over the same wire. To differentiate between sensors, each sensor is given a unique identifying number. For this experiment you will create hardware and software to interface the 8051 microcontroller with a Dallas 1-Wire device so that each parts’ unique identifier can be found. You will incorporate a 1-Wire device into your XS40 schematic, write the code to drive the device, simulate your hardware and software together, and then verify your system in hardware.

OBJECTIVES:

1. Illustrate the importance of external sensors and digital interfaces.
2. Develop a C program in uVision to implement the Dallas 1-Wire protocol, read a unique identifier, and display it on a seven segment display.
3. Reinforce concepts of cosimulation and coverification.

REFERENCES:

XS40 Schematic: http://www.ece.umr.edu/courses/cpe214/schematics.pdf
Pseudo-Random Number Generator datasheet:
http://www.ece.umr.edu/courses/cpe214/onewire-datasheet.pdf

MATERIALS REQUIRED:

1. Keil μVision/51
2. Keil dScope Debugger
3. Windows-based computer with an unused parallel port
4. Unix-based computer
5. Mentor Graphics software
6. 1-Wire Pseudo-Random Number generator:
   http://www.ece.umr.eedu/courses/cpe214/onewire.tar
7. XS40 simulation board
8. Ftp program
**BACKGROUND:**

Most digital systems rely on one or more external sensors to gather data from the outside world. In fact, with no way to interact with the environment, most embedded systems would be completely useless. From a simple keypad of buttons to gyroscopic guidance systems in an airplane (or scooter), the external sensors connected to a digital system in a way define it—or at least what it is capable of. The ability to design an interface between a digital system and a wide range of different sensors is key to creating a successful product.

Digital interfaces come in a wide variety of shapes and sizes, but can be categorized to an extent. For example, most interfaces are considered to be either parallel or serial. A parallel interface has one signal wire for every bit it needs to transfer and sends all bits at the same time. A serial interface has fewer wires and sends only one bit at a time. Another means of categorizing interfaces is whether or not they use an external clock signal. An interface that requires a common clock signal between devices is said to be synchronous. An asynchronous interface does not use a common clock. External peripherals may implement a mix of these different interface types.

Simulation of a hardware-software design can reveal many unexpected problems. In order to simulate your digital design with external hardware, however, a good simulation model of the external hardware must be available. Sometimes the manufacturer of a product will provide you with a model. Sometimes the model is created “in house” during the specification of a new chip. Sometimes, you must create the model yourself.

A problem with simulating hardware is that a significant amount of computer time is sometimes required to simulate a system. Complex hardware models may require simulation of hundreds of thousands of events that would normally occur over a matter of milliseconds in the real world. Simulation of these events can take quite some time, especially if we are simulating several seconds of "real world" time. One solution to this problem is to decrease the complexity of the model, possibly by making it less accurate. Another solution is to use specially designed hardware accelerators for simulation. Another is to carefully select the portions of the design to simulate and skip unneeded portions entirely. In industry a combination of these approaches is frequently used.

In this lab you will create hardware and software to allow the 8051 to read data from a Dallas 1-Wire random number generator. Dallas Semiconductor (now Maxim Integrated Products) has created a line of sensors, EEPROMs, real time clocks, and other peripherals that communicate to a microprocessor using only one signal wire and a ground. Furthermore, these 1-Wire devices can be powered over the signal line, obviating the need for any other wires.

The 1-Wire protocol allows several devices to communicate over the same signal wire. This one-wire bus is accomplished by using a "wired AND" configuration as shown in Figure 1. The bus uses a weak pullup resistor. This resistor allows any device on the bus to easily pull the line low and allows power to be delivered to the 1-Wire devices whenever they are not actively communicating with one another. Each device connected to the bus has the ability to read the current state and to pull the bus low with an internal mosfet. Thus the line will only be at a high voltage if none of the devices are pulling it low. One device on the bus is designated the master while all others are designated as slaves. Usually the master is a microcontroller or microprocessor.
Reading and writing data to a 1-Wire device is always initiated by the master and is conducted one bit at a time. Commands, always eight bits long, are sent from the microprocessor to the 1-Wire device. Data packets sent to or from the 1-Wire device are 64-bits long. A communication sequence is initiated with a special reset pulse as shown in Figure 2. The 1-Wire devices then respond with a corresponding presence pulse. After the reset/presence pulse are complete, the two devices may then send commands or data as a sequence of 1s and 0s (bits). The reset pulse, presence pulse, reading/writing bits, and higher level commands are described in more detail below.

A reset pulse occurs anytime the master holds the bus in a low, or '0', state for more than 480 us. The master must do this before sending any commands to the one wire devices. When the slaves receive a reset pulse, they wait a short time, then respond with a presence pulse. During the presence pulse the slaves hold the bus low for approximately 120us. The master watches for this pulse to determine if at least one slave is connected to the bus.

To write a '0' bit on the bus, the master holds the bus low for 60 to 120 us. To write a '1', the master holds the bus low for less than 15us then lets it go high for at least 45us. To read a bit, the master holds the bus low for approximately 1us, then samples it within 14us. If the line is high, a '1' is read; if the line is low, a '0' is read.

A command is written immediately after the presence pulse. The commands are 8-bits long and are sent least significant bit first. A few of the possible commands and their codes are listed below:

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ ROM</td>
<td>0x33</td>
<td>Read the slave’s 64-bit unique identifier</td>
</tr>
<tr>
<td>MATCH ROM</td>
<td>0x55</td>
<td>Select a device for future commands using its identifier</td>
</tr>
<tr>
<td>READ RAND</td>
<td>0x43</td>
<td>Read a 64-bit random number</td>
</tr>
</tbody>
</table>
In this lab you will develop hardware and software that allows the 8051 to read the unique identifier from a Dallas 1-Wire part. To do this you will place the 1-Wire device into your XS40 schematic and will develop a C program to communicate with it. The 1-Wire device will be implemented with a special 1-Wire model within the FPGA on the XS40 board. When completed, the system will read the identifier from the 1-Wire device and display it on the seven segment display.

The 8051 is especially well suited to being a 1-Wire bus master because it requires little extra logic. Since Port 1 is an open-collector output with internal pull-up resistor, any of it's pins can be directly used to drive a 1-Wire device. When a bit in Port 1 is set in the 8051 the pull down transistor is off and the pin is pulled high by the pull-up resistor. Any device on the bus can easily pull the pin low again. If a bit in Port 1 is cleared, then that pin will drive the bus low.

In this laboratory, the 1-Wire device is implemented in the XC4005XL FPGA on the XS40 board. The 1-Wire device model placed within the FPGA should be connected carefully to the FPGA I/O pins to work properly. Two basic changes must be made, as outlined below and in the 1-Wire datasheet.

The first is required because the Xilinx XC4005 FPGA has no internal tri-state lines. Remember that 1-Wire devices either pull the common bus low or remain in a high impedance state. By default, FPGA output pins drive a strong high or low on a connected circuit. We would like for the FPGA to drive a strong low or a high impedance, but never a strong high. This can be accomplished by replacing the standard ipad/ibuf with an iopad/ibuf/obufe combination. In this configuration the FPGA output pin can be driven high, low, or into a high-impedance state. To guarantee that a strong high will never be output the data input of the obufe should be tied to ground and the enable pin used to transmit data. See Figure 3 for a graphical view of the required output configuration.

![Figure 3: I/O pad configuration](image)

Secondly, the 1-Wire device requires a 6MHz clock while XS40 board has only a 12MHz clock. A clock divider should be created to generate the 1-Wire device's 6MHz clock. A single flip-flop in addition to an inverter will do the job. We'll talk about these steps further in the lab procedure.
PRELIMINARY:

Develop a C program to read the unique identifier from a Dallas 1-Wire device. The program should complete the following steps:

1. Send reset pulse
2. Detect presence pulse from slave(s)
3. Write 8 bit command READ ROM (0x33)
4. Read 64 bit unique identifier
5. Display identifier on seven segment display

To expedite the development of this program, functions are provided below to read and write single bits from a one-wire device, to send the reset pulse, and to detect a presence pulse. The 1-Wire bus will be connected to Port 1 pin 3.

For this lab you will use a copy of the schematic that you used in Lab 5 in which the seven segment display is mapped into memory at location 0xAA55. Use the xdata construct as you did in Lab 6 to assign the display a variable name in your C program. Then, once your program has read the 64 bit unique identifier from the 1-Wire device, it should be displayed on the seven segment display one hex digit after another starting with the most significant nibble (e.g., if the identifier is 5280H, the seven segment display will show a 5, then after a delay a 2, then after a delay an 8, and finally a 0. After some delay the number will be shown again). The result should appear similar to the program you developed in Lab 5, except with the hex digits of the 1-Wire identifier replacing your custom message. For simulation, you should make the delays between displayed digits very small. However, before you verify your design in hardware, you should make the delays long enough to be noticeable by the human eye.

Develop, compile and simulate your program in software using the Keil software development tools to ensure your program is doing what you expect it to. Come to class with a printout of your code and a hex file ready to be downloaded to the XS40 board.

Functions to read and write single bits, to send the reset pulse and to detect a presence pulse are given below. You will need to extend the functions to read and write bits to work with entire bytes or multiple bytes as needed and will need to integrate all these functions together to make your code work. They assume that IO has been defined as an sbit directed to pin 3 of Port 1 so you will need to do that at the beginning of your code. Remember that all data transmitted to and from the 1-Wire device is done with the least significant bit first.

```c
char reset(){ /* send a reset pulse and detect presence pulse */
    unsigned char i,status;
    IO=0; i=128; while(i--);   // hold reset pulse
    IO=1; i=20; while(i--);    // wait for presence

    /* read presence pulse */
    if (IO) status = 0;       // no presence pulse detected
    else status = 1;          // we received a presence pulse
    i=64; while(i--);         // wait for idle
    return status;
}

void write_bit(char b) {
```
unsigned char i;

if(b){
    /* write '1' */
    IO=0; i=1; while(i--); // start write slot
    IO=1; i=11; while(i--); // wait for end of slot
} else {
    I=0; i=11; while(i--); // start write slot and write 0
    IO=1; i=1; while(i--); // wait for end of slot
}
}

char read_bit(){
    unsigned char retval,i;

    IO = 0; i=1; while(i--); // set output to 0 for instant
    IO = 1; i=1; while(i--); // the sampling should occur within 15us

    if(IO == 1) retval=1;
    else retval=0;
    i=11; while(i--); // 60us delay
    return retval;
}

**PROCEDURE:**

**Summary:** Add the 1-Wire unit to the XS40 schematic and compile it to a bit file. Cosimulate your hardware and software together using QuickSim Pro. Eliminate any errors found. Once your hardware-software design is working, download and verify it on the XS40 board.

1. Install the 1-Wire unit in a new working directory.
   A. Create a new working directory called lab8. Download a copy of the 1-Wire model complete with XESS40 schematic at [http://www.ece.umr.edu/courses/cpe214/dist/onenewire_unit.tar](http://www.ece.umr.edu/courses/cpe214/dist/onenewire_unit.tar) untar this into the directory you just created.
   B. You will need to rebuild the XESS40 board and 1-Wire simulation at this point before you start. This only needs to be done once. Enter the command:

   `build_simulation`

   Followed by the command:

   `build_owprn`

2. Open the xc4005 sheet and place the 1-Wire symbol within the sheet. The symbol is called owprn (**One-Wire Pseudo-Random Number generator**). This symbol is located in your main design directory. Several connections must be made to the unit to make it function properly:
   A. A clock must be connected to the 1-Wire unit. As seen on the schematic symbol and in the datasheet, it requires a 6 MHz clock. The clock for the 8051 at Pin 13 is a 12 MHz clock. Create a clock divider from one flip-flop and an inverter, use this to drive the 1-Wire’s clock input.
   B. Ground the reset input of the 1-Wire device. The device can be reset through the 1-Wire protocol so this input is not needed.
   C. The 1-Wire device needs to both read and write to pin 3 of port 1. To do this, the ipad that is currently on the XESS40 schematic needs to be replaced with an ipad and the ibuf should be replaced with an appropriate tri-state driver, as shown in Figure 3.
   1. Place an ipad immediately below the ipad for P6 (Port 1 pin 3).
   2. Rename the first and second properties to match those of the original P6 ipad.
3. Place an ibuf and an obufe so that they are connected to the iopad.
4. Delete the original P6 ipad and ibuf.
D. According to the 1-Wire model datasheet, dq_out goes high when the device needs to pull the 1-Wire bus low. To accomplish this you can tie the input of the new obufe to ground, and tie dq_out to the enable of the obufe.
E. Connect the output of the new ibuf to dq_in.
F. Verify that the seven segment display is connected according to Lab 4.
G. Check and save your design.
H. After you have completed the above steps, print out a copy of your schematic and place it in your lab notebook.

3. Cosimulate your hardware with software using QuickSim Pro.
A. Download the hex file created from the preliminary via an ftp program or floppy disk to the program_files directory under your design. Rename your hex file sram.hex.
B. Load QuickSim Pro and open the XESS40 Schematic. At a minimum you will want to trace the 1-Wire bus (pin 3 of Port 1). Also trace any other signals as necessary to debug your design.
C. The 1-Wire protocol is relatively slow in terms of simulation time. Depending upon how your program is coded, you may have to simulate for more or less time than indicated here. Simulate your design first for two million nanoseconds (2 milliseconds) to verify that the reset pulse has been sent and that the device responded with a presence pulse. Estimate how long this simulation took and write it down in your notebook. You will probably want to zoom out significantly to see significant portions of the simulation waveform.
D. Continue simulation until the device ID is displayed on the seven segment display or until you notice a problem. For most designs, this will involve approximately ten or eleven million more nanoseconds of simulation (i.e. 12-13 mS of simulated time). Write down approximately how long the simulation took in your notebook.
E. Print out a trace that includes at least the 1-Wire bus signal. Label the reset pulse, presence pulse, command transfer, and data transfer. Identify the value of at least the first nibble (4-bits) of the 1-Wire identifier in the 1-Wire bus signal.

4. Verify your design in hardware using the XS40 board.
A. Create a new bit file for your design, which incorporates all the changes you have made. This bit file can be created using the xmake command, as explained in lab 1.
B. Modify your program to increase the delay length between displaying digits of the device ID. Create a hex file for your new software.
C. Download your new bit and hex file to the PC connected to the XS40 board.
D. Connect the power and parallel-port to the XS40 board. Using GXSLOAD, download both the bit and hex files to the board. Reset the microcontroller by strobing D0 in the GXSPORT program. If your design works correctly, you should see a device ID flash across the seven segment display. If you don't see the identifier, use a logic analyzer or oscilloscope to view P1^3 to see if the pulses are being sent correctly by the 8051. You might also check P0 and P2 to verify your program is executing correctly. If you cannot find your problem in hardware, you may want to go back to your simulation using the Keil software development tools.
E. Demonstrate your working design to your TA. When you are finished, delete your files from the lab computer.
**QUESTIONS:**

1. What reasons would motivate a designer to select a 1-Wire device over traditional sensors and peripherals? What disadvantages would there be to this selection?
2. In this lab you simulated reading one 64-bit quantity from the 1-Wire device. For a similar EEPROM 1-Wire device on a multi-drop bus, ten or more 64-bit quantities would need to be transferred. From your experience here, estimate how long it would take to simulate such a system? What could be done to speed up that simulation time?
3. If you were using a microcontroller that had some pins that could only work as outputs, and some that could only work as inputs, how could you drive a 1-Wire bus? Draw a sketch of the needed logic and connections.
4. Would the Dallas 1-Wire interface be characterized as a parallel or serial protocol? Synchronous or asynchronous? Why?