Implementing an I^{2}C Master Bus Controller in a FPGA

The physical size of ICs has reduced dramatically over the years. The main reason, of course, is attributed to the fact that more and more transistors can be cramped into a smaller space. A less mentioned reason is because the pins for interconnections between ICs have also decreased both in size and in number. As you probably know, the actual circuitry of the IC is much smaller than the packaging of the IC. The reason why the packaging has to be larger is because of the larger pins needed for the connections on the PCB. One solution for reducing the packaging size is therefore to have fewer pins for the connections. For example, back in the 80's a popular real-time clock (RTC) chip is the National Semiconductor's MM58167B. It is a dual-in-line package with 24 pins. Of these 24 pins, four of them (chip select, read, write, and ready) are for interfacing control, five of them (A0 to A4) are for addressing, and eight of them (D0 to D7) are for data. So 17 out of the 24 pins are used just for sending and receiving data/command between the RTC and the microcontroller. A newer RTC chip, the Epson's RTC-58321, still uses 10 out of 16 total pins dedicated for communication between the chip and the microcontroller. In order to reduce the connection pin counts even further, the I^{2}C protocol was developed by Philips, which requires only two lines for communication between two or more chips.

In this article, I will show you how to implement an I^{2}C master bus controller using a field-programmable gate array (FPGA). A FPGA is a chip that can be used to implement any digital logic circuit. I will demonstrate the master controller's operation by having it communicate with a real-time clock chip, the Maxim DS3232, connected on the I^{2}C bus as a slave. Using the I^{2}C bus as the communication channel, the master controller will be able to send and receive data to and from the slave. In order to show the design of the I^{2}C master bus controller as simply as possible for ease of understanding, some of the subtle details of the I^{2}C specifications are not fully implemented. Hence, there might be situations where our master controller might report a false error.

The I^{2}C bus

The I^{2}C (Inter IC) bus is a simple bi-directional serial bus that supports multiple masters and slaves. It consists of only two lines; a serial bi-directional data line (SDA) and a serial bi-directional clock line (SCL). Within the I^{2}C bus specifications, a standard mode with a maximum clock rate of 100k Hz and a fast mode with a maximum clock rate of 400k Hz are defined.

Each device connected to the I^{2}C bus is software addressable by a unique address, and a simple master/slave relationship exists at all times among the devices. The device that controls the sending and receiving of messages by controlling the bus access is referred to as the *master*. Devices that are controlled by the master are the *slaves*. Both the master and the slave can send and receive messages. A device that sends data onto the bus is referred to as the *transmitter* and a device receiving data is referred to as the *receiver*.

More than one master and more than one slave can all co-exist on the same I^{2}C bus. However, the bus is always controlled by a single master who is responsible for generating the serial clock (SCL), and controlling the bus access by initiating and terminating a message transfer.

Connecting IC devices to the I^2C bus

As mentioned earlier, the I^2C bus consists of only two lines, SDA and SCL. Both of these lines are open-drained, and must be pulled up to VCC with a 5.6KΩ resistor. Regardless of how many devices are connected to the bus, only one pull-up resister is needed per line. Furthermore, the SCL line needs to be pulled up with a resistor only if there will be two or more masters in the system, or when the slave will do clock stretching as a flow-control measure. In the first case, the pull-up resistor on the SCL line is required for arbitration purposes when two or more masters try to initiate a data transfer at the same time. An arbitration procedure has been defined to avoid the chaos that might ensue from such an event. In the second case, a receiver may not be ready to receive data from the transmitter, and so it will hold the SCL line low to “stretch” the clock to delay the transmitter.

As shown in Figure 1, our system consists of only one master and one slave. Furthermore, to make our master controller as simple as possible for ease of understanding of how a controller is designed, we will assume that the slave will not do any clock stretching. (If the slave does do any clock stretching, our master controller will simply see it as an error in the data transmission, and will halt in the error state.) Hence, we will not use a pull-up resister on the SCL line. Note that this implementation of the master controller does not follow the full specifications of the I^2C. Nevertheless, after understanding how the controller is design, you can very easily modify it to follow the full specifications. Our I^2C master controller is implemented in an Altera Cyclone II FPGA, and the slave is the Maxim DS3232 real-time clock chip. At all times, the master is in control of the SCL line.

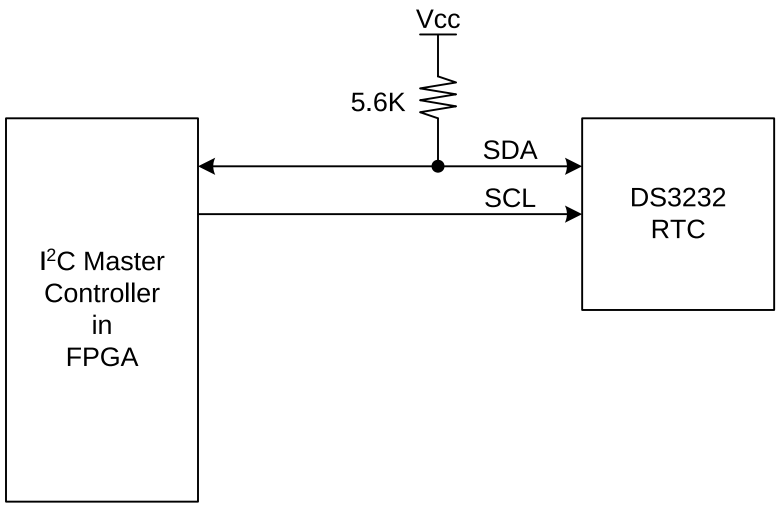

Figure 1: I^2C bus system with the I^2C master controller implemented in a FPGA and the real-time clock device acting as the slave. Since there is only one master, only the SDA line needs to be pulled up with a 5.6KΩ resistor. The SCL line is controlled by the master controller.

The I^2C protocol

The I^2C bus is idle when both SCL and SDA are at a logic 1 level. The master initiates a data transfer by issuing a START condition, which is a high to low transition on the SDA line while the SCL line is high as shown in Figure 2 (a). The bus is considered to be busy after the START condition. After the START condition, a slave address is sent out on the bus by the master. This address is 7 bits long followed by an eighth bit which is a data direction bit ($R/\overline{W}$) where a 0 indicates a write from the master to the slave and a 1 indicates a read from the slave to the master. The master, who is controlling the SCL line, will send out the bits on the SDA line, one bit per clock cycle of the SCL line, with the most significant bit sent out first. The value on the SDA line can be changed only when the SCL line is at a low.

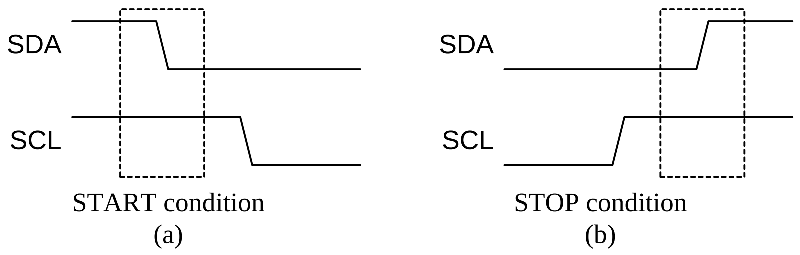

Figure 2: The START (a) and STOP (b) conditions are both initiated by the master. The START condition happens when the SDA line changes from a high to a low while the SCL line is at a high. The STOP condition happens when the SDA line changes from a low to a high while the SCL line is at a high. These are the only two situations where the SDA line can change when SCL is at a high.

The slave device whose address matches the address that is being sent out by the master will respond with an acknowledgment bit on the SDA line by pulling the SDA line low during the ninth clock cycle of the SCL line as shown in Figure 3. The direction bit ($R/\overline{W}$) determines whether the master or the slave will be the transmitter in the subsequent data transmission after the sending of the slave address.

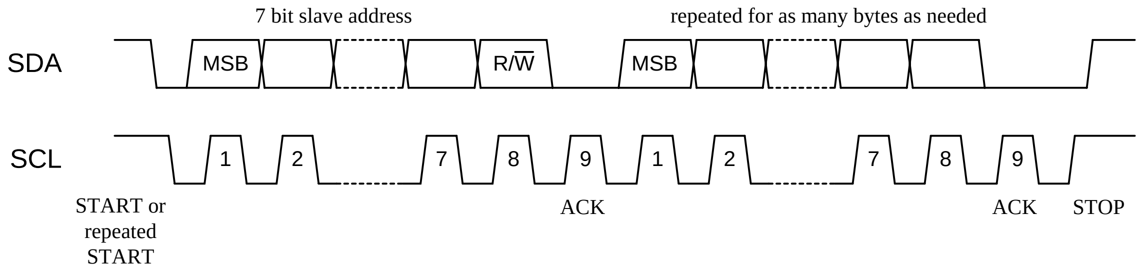

Figure 3: Data transmission is initiated by the master sending the START condition. For every eight bit of data transmitted, the receiver sends an acknowledgment during the ninth clock cycle by pulling SDA low. The only exception is when the master-receiver wants to end the data transmission in which case the master (who is the receiver) will not acknowledge by keeping SDA high. Data transmission is terminated by the master sending the STOP condition.

Every byte put on the SDA line for transmission must be 8-bits long with the most significant bit first. Except for the START and STOP conditions, the SDA line must not changed when the SCL line is high. The number of bytes that can be transmitted is unrestricted. Each byte has to be followed by an acknowledge bit. The acknowledge-related clock pulse is generated by the master. The transmitter releases the SDA line (sets it high) during the acknowledge clock pulse, and the receiver must pull down the SDA line during the acknowledge clock pulse to acknowledge the receipt of the byte. The one exception is when a master-receiver is involved in a transfer. In this case the master-receiver must signal the end of data to the slave-transmitter by not generating an acknowledge on the last byte that was clocked out of the slave.

To signal the end of data transfer, the master will send a STOP condition by pulling the SDA line from low to high while the SCL line is at a high as shown in Figure 2 (b). Instead of sending a STOP condition, the master can send a repeated START condition so that the master can change the direction of the data transmission without having to release the bus.

Figure 4 (a) shows the scenario where the master writes one byte of data to the slave, i.e., the master is the transmitter and the slave is the receiver. The master initiates the data transfer by first issuing the START condition followed by the 7-bit slave address plus the write (0) bit. After receiving an acknowledgment from the slave, the master sends the register number to let the slave know which register the following data is to be written into. Again the slave responds with an acknowledgment. The master then sends the data byte to the slave. After the slave acknowledges the receipt of the data byte, the master sends the STOP condition.

Figure 4 (b) shows the scenario where the master reads one byte of data from the slave, i.e., the master is the receiver and the slave is the transmitter. The master initiates the data transfer by first issuing the START condition followed by the 7-bit slave address plus the write (0) bit. Although the master wants to receive a byte, we need to send the register address byte first to let the slave know which register the master wants to read from. After receiving an acknowledgment from the slave, the master sends the register number to let the slave know which register to read from. Again the slave responds with an acknowledgment. This time the master has to do a repeated START condition because it needs to change the data direction from a write to a read. The repeated START is followed by the slave address again but this time with the read (1) bit instead. The slave acknowledges and then sends the data byte from the addressed register to the master. This time, since the master is the receiver, the master has to acknowledge the receipt of the data byte. If the master wants to receive more data bytes from the slave, it will send a 0 to acknowledge it. If the master does not want to receive any more bytes, then it will not acknowledge by keeping SDA at a high. Finally, the master sends the STOP condition.

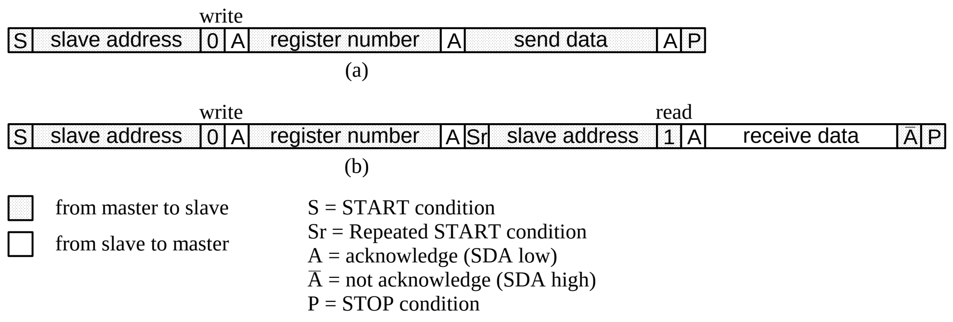

Figure 4: (a) The master transmits one byte of data to the slave. (b) The master receives one byte of data from the slave. In both cases, the master first sends the 7-bit slave address and the write bit, followed by the register number to access. In the master-transmitter scenario (a), the master can immediately send out the data byte since the data direction is still a write. For the master-receiver scenario (b), the master has to do a repeated START and resend the slave address with the read bit in order to change the direction of the data transmission from a write to a read. After this, the master can then receive a byte of data from the slave from the given register number.

Designing the I^2C master controller

The trick for implementing the I^2C master controller, or for any controller for that matter, is to use a finite-state machine (FSM). Manually designing a FSM requires some knowledge in digital logic design, and this is beyond the scope of this article. See reference 2 for a good source in learning about digital logic design. However, even without any knowledge in digital logic design, you can still very easily implement a FSM by writing VHDL or Verilog code. VHDL and Verilog are two popular hardware description languages (HDL) for designing digital circuits. If you know how to write C code, you will very quickly feel comfortable with writing VHDL or Verilog code because many software constructs are identical. It is just a matter of learning the new syntax. Reference 3 is a good source for learning VHDL. I will now discuss how the I^2C master controller is designed using VHDL.

Finite-state machine

Before looking at the VHDL code for the I^2C controller, I need to tell you a little about what a finite-state machine is. A finite-state machine is a sequential circuit that uses (as the name suggests) a finite number of states to keep track of its history of operations, and based on this history and its current inputs, determine what to do next. A sequential circuit is one where its outputs are dependent on its history of operation and its current inputs. The current state that the FSM is in is stored in a memory. The operation of the FSM is simply to traverse from one state to another by changing the content of the memory, and to output the necessary control signals from the controller in the appropriate state. The FSM determines what the next state to go to by considering the current state that it is in and the input signals to the controller. Figure 5 shows the block diagram of a FSM. As seen in the figure, a FSM consists of three main parts: the state memory for storing the current state that the FSM is in; the next-state logic for determining the next state for the FSM to go to, which depends on the current state that it is in and the input signals. The input signals can either be external signals to the controller or conditional signals generated within the controller circuit itself; and finally the output logic for generating the appropriate output signals from the controller for controlling whatever devices you want. The generation of the output signals is dependent on the current state that the FSM is in. There are some FSMs where the output signals are dependent not only on the current state, but also on the input signals. The timing for the FSM is controlled by a clock signal. At every clock cycle, the FSM will go to a new state, so at every clock cycle, the FSM will generate a new set of output signals.

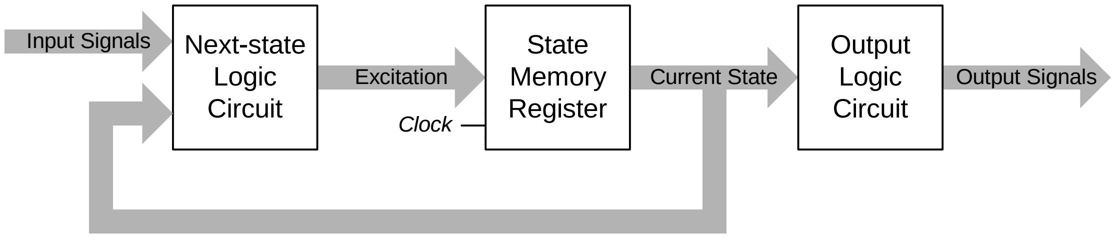

Figure 5: Block diagram of a finite-state machine. There are three main components: the state memory to store the current state that the FSM is in; the next-state logic to determine the next state to go to depending on the current state that it is in and the input signals; and the output logic to generate the appropriate output signals from the controller.

VHDL code

Figure 6 shows a VHDL code template for describing a FSM. The language syntax is not case sensitive, however, in the example, all keywords will be in upper case, and all user identifiers will be in lower case or a mixture of upper and lower case. The first two lines with the two dashes are comments. The next three lines are standard library files to include. Among other things, these library files define the STD_LOGIC and STD_LOGIC_VECTOR type that are used later on. The ENTITY section, which begins with the ENTITY keyword and ends with the END I2C_controller line, defines the interface between this module (black box) and the outside. It includes all the necessary input and output signals to the module. In the example, there are the clock and the reset input signals, and the two output signals scl and sda. The ARCHITECTURE section defines the operations of the module, which in our case will be the FSM, and therefore, must contain the three parts of a FSM as modeled in Figure 5.

The state variable declared using the SIGNAL keyword is the state memory. It is of type STD_LOGIC_VECTOR, which is an 8-bit bit string. The PROCESS block specifies that whenever there is a change in either of the two signals, Clock and ResetN, the statements inside the block will be executed in sequential order starting with the first line. We have an active low reset signal as specified in the IF statement that tests for the signal being a 0. When ResetN is asserted, i.e., when ResetN is equal to 0, the module goes into the reset mode and outputs a logic 1 value for both the scl and sda output signals. Furthermore, it assigns state x"00" as the initial state for when the FSM starts. x"00" is the syntax for the two hexadecimal digits 00. When ResetN is de-asserted, the ELSIF statement is executed. The condition, Clock'EVENT AND Clock = '1', specified inside the ELSIF statement checks for a rising clock edge. So at every rising clock edge, the FSM will go to a new state and a new set of output signals will be generated. The state that the FSM is currently in is implemented using the CASE statement that tests for the value in the state variable. The CASE statement is the same as the SWITCH statement in C, which is like a nested IF statement. The different states that the FSM can go to are defined in the different cases using the WHEN keyword. So on startup, the FSM will go to state x"00" because the case for WHEN x"00" will be true. In this state, the FSM will generate a logic 1 signal for the two output signals scl and sda. In this state, it also assigns x"01" to the state variable, which means that the next state the FSM will go to will be state x"01". At the next rising clock edge, when it executes the CASE statement, the state variable now has the value x"01" so it will go to the WHEN x"01" case. In state x"01", the FSM assigns a logic 1 to scl, a logic 0 to sda, and x"02" as the next state to go to.

```
-- FSM template
-- Copyright Enoch Hwang 2008

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY I2C_controller IS
PORT (
   Clock, ResetN: IN STD_LOGIC;
   --I2C control output signals
   scl: OUT STD_LOGIC;
   sda: OUT STD_LOGIC;
```

```
END I2C_controller;

ARCHITECTURE fsmd OF I2C_controller IS
   SIGNAL state: STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN

   PROCESS(Clock, ResetN)
   BEGIN
      IF(ResetN = '0') THEN
         scl <= '1';
         sda <= '1';
         state <= x"00";

      ELSIF(Clock'EVENT AND Clock = '1') THEN
         CASE state IS
         WHEN x"00" =>  -- Idle
            scl <= '1'; -- SCL = 1
            sda <= '1'; -- SDA = 1
            state <= x"01";

         WHEN x"01" =>  -- Start
            scl <= '1'; -- SCL stays at 1 while
            sda <= '0'; -- SDA changes from 1 to 0
            state <= x"02";

         -- remaining states here

         WHEN OTHERS =>
            scl <= '1';
            sda <= '1';
         END CASE;
      END IF;
   END PROCESS;

END fsmd;
```

Figure 6: VHDL code template for a FSM. The language syntax is not case sensitive, however, in the example, all keywords are in upper case, and all user identifiers are in lower case or a mixture of upper and lower case.

The I^{2}C master controller

We are now ready to fill in the details into our VHDL FSM code template to implement our I^{2}C master controller. The output signals that we need to generate and the input signals to test for of course must follow the I^{2}C protocol as described previously. Furthermore, the clock speed for the FSM must also satisfy the speed defined in the I^{2}C protocol. The complete code can be downloaded from the Circuit Cellar FTP download site. Fragments of the code are shown in Figure 7, Figure 8 and Figure 9.

```
   ...
   sda: INOUT STD_LOGIC;
   ...
   SIGNAL sda01: STD_LOGIC;   -- internal SDA having values of 0 and 1
   ...
   sda <= 'Z' WHEN sda01 = '1' ELSE '0';  -- convert SDA 0/1 to 0/Z
   ...
   sda01 <= '1';  -- SDA = Z
   ...
   sda01 <= '0';  -- SDA = 0
   ...
```

Figure 7: Fragments of the VHDL code for our I^{2}C master controller showing the implementation of the SDA signal. The first line declares the interface signal sda as INOUT for bi-directional data transfer. The second line declares an internal signal named sda01, which will be assigned the logical values of 1 and 0. Next, a conditional assignment statement is used to set the sda line to either the high impedance value 'Z' or the logic value 0 depending on the value of sda01. sda gets a Z value if sda01 is a 1, otherwise sda gets a 0 value. The last two lines show the actual logical value of 1 and 0 being assigned to the internal signal sda01 instead of directly to the interface signal sda.

The implementation of the sda signal requires some special attention as shown in Figure 7. Recall that the SDA line in the I^{2}C bus is bi-directional, so in the ENTITY declaration, the sda signal has to be declared as INOUT so that it is capable for doing both input and output. Furthermore, the SDA line in the I^{2}C bus is open-drained and is pulled up by a 5.6K resistor. So to output a logic 1 on this line, we need to actually set this line to a high impedance. To get a high impedance, we need to use a tri-state output and assign to it a 'Z' value. The conditional signal assignment statement

```
   sda <= 'Z' WHEN sda01 = '1' ELSE '0';
```

assigns a high impedance value 'Z' to the sda line when the internal signal sda01 has a logic 1 value, otherwise, the sda line gets the logic value 0. Hence, to assign the logic value 0 or 1 to the sda line, we would instead assign the value 0 or 1 to the internal signal sda01, which in turn will set the sda line to 0 or Z respectively.

Figure 8 shows the relevant VHDL code for the generation of the scl clock signal. The scl signal is generated in the FSM process by going back and forth between two states: the first state will set scl to a 0 and the second state will set scl to a 1. Therefore, for every scl cycle, the FSM must go through two states or two cycles. Hence to get a 100k Hz speed for the scl signal, which is the I^{2}C standard mode maximum clock speed, the FSM clock speed must run two times faster or at 200k Hz. Our primary input clock speed is 50M Hz. We will need a clock divider to slow this clock down to 200k Hz. The clock divider process will count the 50M Hz clock ticks from 0 to 50000000/200000 = 250. To get a 50% duty cycle for the 200k Hz clock, the clock signal CLK_200k_Hz will toggle after every 250/2=125 counts.

The fsm process executes whenever there is a change in the CLK_200k_Hz signal. The ELSIF (CLK_200k_Hz'EVENT and CLK_200k_Hz = '1') statement says that the CASE statement will be executed only at the rising edge of the CLK_200k_Hz clock. The code fragment shown in Figure 8 shows that the FSM will go back and forth between state x"02" and state x"03". In state x"02", scl is assigned a 0 and in state x"03", scl is assigned a 1. The IF statement in state x"03" tests when to exit this loop and continues to state x"12".

```
   ...
   CLK_50_MHz: IN STD_LOGIC;
   ...
   -- constants for 200kHz clock divider
   -- to get 100kHz for I2C standard; every 2 FSM cycles = 1 I2C cycle
   CONSTANT max200k: INTEGER := 50000000/(100000*2);  -- = 250
   SIGNAL clockticks200k: INTEGER RANGE 0 TO max200k;
   SIGNAL CLK_200k_Hz: STD_LOGIC;
   ...
   fsm: PROCESS(CLK_200k_Hz, ResetN)
   ...
      ELSIF(CLK_200k_Hz'EVENT and CLK_200k_Hz = '1') THEN
         CASE state IS
   ...
         WHEN x"02" =>
            SCL <= '0';
            state <= x"03";
         WHEN x"03" =>
            SCL <= '1';
            IF (bitcount - 1) >= 0 THEN
               bitcount <= bitcount - 1;
               state <= x"02";
            ELSE
               bitcount <= 7;
               state <= x"12";
            END IF;

   clockdivider200k: PROCESS
   BEGIN
      WAIT UNTIL CLK_50_MHz'EVENT and CLK_50_MHz = '1';
      IF clockticks200k < max200k THEN
         clockticks200k <= clockticks200k + 1;
      ELSE
         clockticks200k <= 0;
      END IF;
      IF clockticks200k < max200k/2 THEN
         CLK_200k_Hz <= '0';
      ELSE
         CLK_200k_Hz <= '1';
      END IF;
   END PROCESS;
```

Figure 8: Fragments of the VHDL code for our I^{2}C master controller showing the generation of the SCL signal. A second clock divider process is used to generate a 200k Hz clock. This 200k Hz clock drives the FSM process for controlling the SCL line. In state x"02" in this process, SCL is set to 0, and in state x"03", SCL is set to a 1. Since it takes two states or two cycles to generate one cycle on the SCL line, therefore, the SCL clock speed is at 100k Hz.

Figure 9 shows the portion of the VHDL code that sends the start condition signal followed by the slave address of 1101000 plus the write bit of 0 to the RTC slave. The signals to output or input on the scl and sda lines follow the I^{2}C protocol as discussed previously. The initial state, x"00", is the I^{2}C idle state when both the clock and data lines are at a logic 1 level. In the next state, x"01", we send out a start signal by assigning a 0 to sda01 while scl remains at a 1. Recall from the earlier discussion that sda gets the value 0 when sda01 is a 0. In state x"02", we assign one bit of the slave address to sda01 when scl is at a 0. The variable bitcount is used to index the bit string SlaveAddress_Write which contains the 7-bit slave address plus the write bit. In state x"03", sda remains stable while scl is at a 1. States x"02" and x"03" will repeat for eight times for sending out the eight bits in SlaveAddress_Write. After sending out the last bit, the FSM goes to state x"12" to get an acknowledgment from the slave. This is accomplished by first setting the sda line to high impedance 'Z' in state x"12", and then read the sda line in state x"13" to see if the slave has set it to a 0. In state x"13", if sda is a 0 then the slave has acknowledged the receipt of the address and the communication between the master and the slave can continue, otherwise there is an error and the FSM jumps to the error handling state x"EE".

```
ARCHITECTURE FSMD OF I2C_controller IS
   -- I2C address of the slave + write'
   CONSTANT SlaveAddress_Write: STD_LOGIC_VECTOR(7 DOWNTO 0):="1101000"&'0';
      ...
         WHEN x"00" =>  -- Idle
            -- when idle, both SDA and SCL = 1
            scl <= '1';    -- SCL = 1
            sda01 <= '1';  -- SDA = 1
            state <= x"01";
         WHEN x"01" =>  -- Start
            scl <= '1';    -- SCL stays at 1 while
            sda01 <= '0';  -- SDA changes from 1 to 0
            bitcount <= 7; -- starting bit count
            state <= x"02";
         -- send 7-bit slave address followed by R/W' bit, MSB first
         WHEN x"02" =>
            scl <= '0';
            sda01 <= SlaveAddress_Write(bitcount);
            state <= x"03";
         WHEN x"03" =>
            scl <= '1';
            IF (bitcount - 1) >= 0 THEN
               bitcount <= bitcount - 1;
               state <= x"02";
            ELSE
               bitcount <= 7;
               state <= x"12";
            END IF;
         -- get acknowledgment' from slave
         WHEN x"12" =>
            scl <= '0';
            sda01 <= '1';
            state <= x"13";
         WHEN x"13" =>
            scl <= '1';
            Ack <= sda; -- 0 = acknowledge'; error if it is a 1
            IF sda = '1' THEN
               RegisterAddressOut <= x"13";
               state <= x"EE";   -- acknowledge error
            ELSE
               state <= x"20";   -- send register address
            END IF;
```

Figure 9: A portion of the VHDL code for our I^2C master controller showing the sending of the slave address and the write bit to the RTC.

The rest of the data transfer between the master and the slave pretty much follows the same pattern as that shown in Figure 9.

Implementing the I^{2}C master controller

Now that we have the complete design of the I^{2}C master controller written in VHDL, we are ready to implement it to actually see it working. First you will need to get a hardware description language (HDL) compiler to synthesize the VHDL source code into a netlist. You can download a free web edition of Quartus II by Altera to do this. See reference 5 for the site link. After installing the program, you can start a new project and add our master controller VHDL code to the project. See reference 2 for a tutorial on how to use the Quartus software. Quartus will compile the VHDL source code to a netlist that can be downloaded onto the FPGA. For our demonstration, I used the Altera DE2 development board with a Cyclone II FPGA. The board does not have the DS3232 RTC chip so I have wire wrapped this chip to the general PIO pins on the board as shown in Figure 10. There are six wires connecting the RTC chip to the GPIO pins, but only four of them are used for the I^{2}C bus communication; they are SDL, SCL, VCC, and GND. In addition to the RTC chip connected to the GPIO is also the 5.6K pull-up resistor for the SDA line.

After you have successfully compiled the master controller VHDL code, you can use Quartus to download the resulting netlist onto the FPGA on the DE2 board. Figure 11 shows the full DE2 board with the I^{2}C master controller implemented inside the Cyclone FPGA and communicating with the RTC. The 7-segment display is showing, from left to right, the FSM state d5 that the FSM is currently in, register 02 of the slave that the FSM is reading data from, and the data 16 that is being sent from register 02 of the slave to the master. The master controller also allows you to set the address and data switches at the bottom of the board to read or write from/to specific register locations in the RTC chip.

A final note regarding the implementation of this design is that you can use a different FPGA prototype board and/or synthesizer software. The VHDL source code should be general enough so that you can use another compiler and compile it for another FPGA chip. If you do, you will need to change the hardware dependent pin mappings. Furthermore, you should also be able to use another chip instead of the RTC chip that I used as the I^{2}C slave. In any case, the I^{2}C master controller design presented here should be easy enough for you to understand and to make modifications to it as the need arises. You can also use this FSM as a template for designing other controllers.

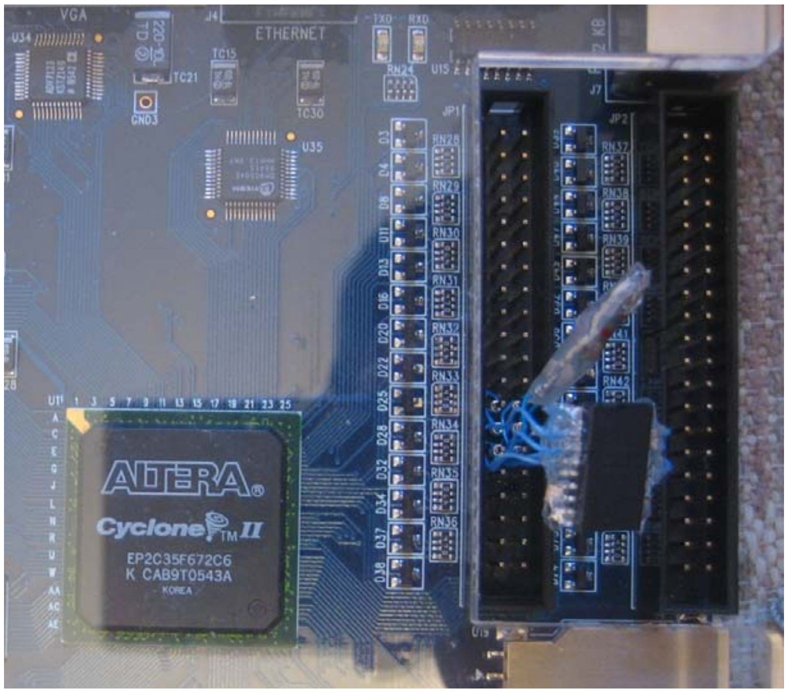

Figure 10: Connection of the DS3232 RTC chip to the I^{2}C master controller implemented inside the Cyclone FPGA via the GPIO pins. In addition to the RTC chip connected to the GPIO is also the 5.6K pull-up resistor for the SDA line.

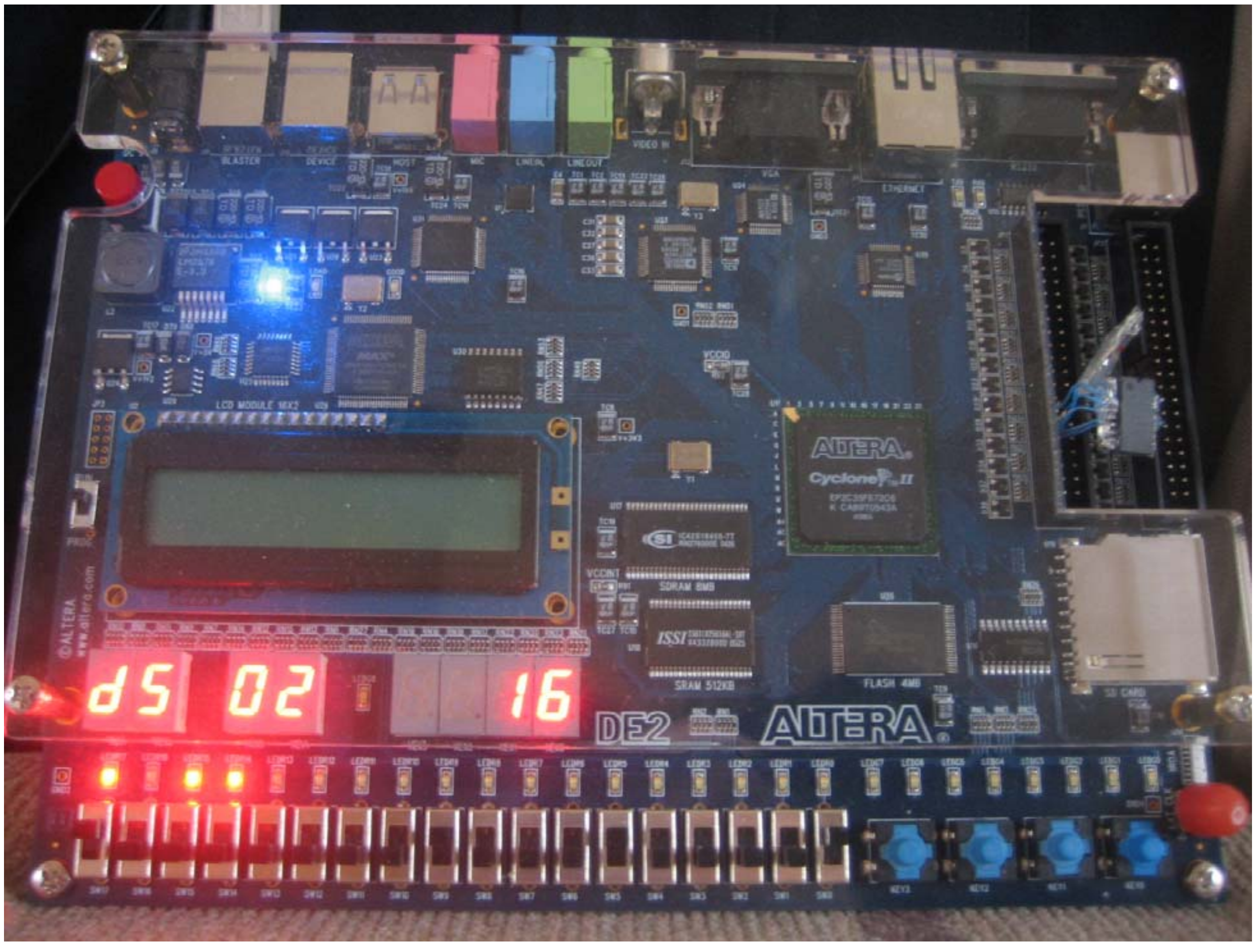

Figure 11: The Altera DE2 board with the 7-segment display showing, from left to right, the FSM state d5 that the FSM is currently in, register 02 of the slave that the FSM is reading data from, and the data 16 that is stored in register 02. The right-most 8 switches at the bottom of the board are configured for inputting an 8-bit data. The next 8 switches are for inputting an 8-bit address. And the left-most switch is the direction switch for read or write.

References

1. Philips I^{2}C specifications
   http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf
2. Digital Logic and Microprocessor Design with VHDL, E. Hwang, Thomson, 2006.
   http://faculty.lasierra.edu/~ehwang/digitaldesign
3. A VHDL Primer, J. Bhasker, Prentice Hall.
4. Data sheet for the Maxim DS3232 real-time clock,
   http://datasheets.maxim-ic.com/en/ds/DS3232.pdf
5. Altera Quartus II development software,
   http://university.altera.com/materials/software/unv-quartus2.html.
6. Altera DE2 development board,
   http://university.altera.com/materials/boards/unv-de2-board.html