CSEE W4823x
Prof. Steven Nowick

Handout 27
October 28, 2016

Project #1: Designing a Master Controller for the Philips/NXP I2C Bus Protocol

This homework is due on **Friday, November 18, at 4pm** (submission details to be announced soon). *Note the revised Friday deadline.*

Introduction. This homework is an introduction to modelling and simulating of a real-world controller: a "master" unit for the Philips (now NXP) commercial I2C serial bus protocol. You will need to understand some subtleties of this protocol, and its operation. Then, you will design *two Moore state diagram specifications* for a master controller, model the FSM specification in VHDL, and simulate it using the Altera Quartus CAD package.

Your first FSM specification (Version #1) will support basic operation. Your second FSM specification (Version #2) will provide basic *fault tolerance* in addition to supporting basic operation. In particular, the latter will include error detection: every data byte will be transmitted as a parity code. If any errors are detected by the receiver, the sender will be notified and appropriate action can be taken.

Grading. This first project will be worth approximately *15%* of your final grade.

Working Solo or in Groups. You are allowed to do this project either solo or in a group-of-two. If you have a group-of-two, you both get the same grade.

Required Reading: Many important details are included in this current handout (Handout #27), which you should read very carefully. In addition, links to several useful documents are provided on the class web page, on the I2C bus protocol, FAQ, and other resources:

- *Handout #27a. I2C Configuration and Protocol.* A simple overview of the basic protocol.
- *Handout #27b. I2C Bus Technical Overview (Embedded Systems Academy).* A web site, from the Esacademy, containing useful summaries of the I2C bus protocol, its history, and detailed presentation of I2C bus events (http://www.esacademy.com/en/library/technical-articles-and-documents/miscellaneous/i2c-bus.html). Various links and sub-links provide useful technical information, so you should explore the various resources available at this site.
- *Handout #27c. I2C: Getting Acknowledge from a Slave (as Receiver.)* Some useful details of one transmission scenario. (Many more are included in Handout #27b link above.)
- *Handout #27d. I2C-Bus Specification and User Manual (Rev. 5, Oct. 2012) [NXP Semiconductors].* A complete reference manual from NXP Semiconductors on the protocol. Some of the relevant details for this assignment are already covered in Handouts #27 and #27a-c above. However, this manual includes particular details on some cases not covered above. It also includes further details on the basic protocol.

  Important Note: the manual includes many items you will not need. These include advanced modes (fast-mode, fast-mode plus, ultra-fast-mode), clock synchronization, bus arbitration, clock stretching, 10-bit extended addressing, broadcast, missing responses, and electrical issues. Much of this material is interesting to read, but not relevant for the assignment. You should *only* support "standard-mode" for this assignment, and assume the valid basic I2C protocol is observed.
- *Handout #27e. Frequently-Asked Questions (FAQ) + Online Discussion: Piazza.* Handout #27(e) is a detailed initial FAQ, answering some basic questions and providing important details of requirements and hardware specifications (*released next week*). Further ongoing discussion, updates and clarifications will be covered on the class "piazza" page (released next week). **Read FAQ and piazza postings carefully.**

These handouts include detailed explanations of relevant parts of the bus protocol, so be sure to read them carefully.

Optional Supplemental References: There are several several good websites which describe the I2C bus in great detail, and give interesting information on the history of the protocol and its use in hundreds of commercial products. So, optionally, we provide a list of references to some of these sites, but *you are not required to do any additional search on this topic (unless you want!).* Note that these documents contain not only useful pointers on the I2C protocol, but also a huge amount of technical material that is irrelevant to this project (bus arbitration, circuit-level issues, extended modes, etc.). These include:

(a) *I2C Protocol Wiki pages* (http://en.wikipedia.org/wiki/I2C and others);

(b) *I2C Bus Application Note [NXP Semiconductors]* (http://www.nxp.com/documents/application_note/AN10216.pdf). This document gives additional technical details. It also opens with a nice overview of the practical industrial applications of the I2C bus.

I2C Bus Background. The I2C bus was designed to coordinate the communication of peripheral devices with different interfaces. This bus was primarily used in applications for televisions, VCRs, and other audio-visual equipment. However, today, the I2C bus is used in many embedded applications. In particular, it has become a recent standard for dynamic system power management (PMBus, Handout #27(d), sec. 4.3), intelligent platform management (IPMI, Handout #27(d), sec. 4.4), and thermal management between boards (ATCA, Handout #27(d), sec. 4.5) as well as within 3D chips.

Prior to the development of the I2C bus, a large amount of hardware, glue-logic, and wiring was needed to allow peripheral devices to coordinate and communicate. Adding additional devices would cause a substantial increase in hardware. Using the I2C bus reduces the hardware and logic complexity with the addition of more devices and also reduces the amount of noise within the system. The I2C protocol is elegant, simple, and highly scalable. It is designed to accommodate different components operating at very different rates (however, we will not focus on this aspect in this problem).

I2C Bus Configuration. The I2C bus is a 2-wire serial bus consisting of 2 bi-directional wires, Serial Data (SDA) and Serial Clock Line (SCL). All data transfers are synchronized over these two wires. Both the SDA and SCL are "open drain" drivers which allows any connected device to drive the output low. For this problem, you will not need to understand details of the electrical issues. The basic idea is that the connected devices can force a low value on the serial wires if desired: *any unit asserting a low value on a bus wire will force it low (i.e. 0).* Units can also assert a high value (i.e. 1) on the bus (if there is no contention). By default, if no unit is driving the bus (i.e. all connections tri-stated), then the value will by default go high (i.e. 1). Each peripheral device is connected to both SDA and SCL. Each such device connected to the bus has a unique serial address, which serves as an identifier.

Note: In the above required handouts and optional links, you can read discussion how the SCL clock can be "stretched" by slow units on the bus, when they are not ready for the next data item; this stretching happens easily using wired-AND drivers, but you do not need to understand this or support this feature for this assignment!

I2C Bus Protocol: The I2C bus protocol is a master-slave protocol. In general, the role of the master is to initiate communication on the bus by issuing a start condition, request a slave device to communicate with it, and eventually terminate communication through a stop condition (P). The role of the slave is to respond to the master's request by first sending an acknowledgment (ACK), and then to perform the desired communication with the master until the stop condition is issued.

For the I2C bus protocol, any connected device has the ability to be the master, however *only* one device can be

the master at a particular time.[1] Hence "clock synchronization" and "arbitration" occur, before a transmission, where any competing masters must contend for one to win control of the bus. You will *not* deal with clock synchronization and arbitration in this assignment.

In addition to the duties of the master outlined above, the master is always the owner of SCL and is responsible for determining whether it wants transmit data or receive data.

The master first initiates communicating by broadcasting a START symbol onto the I2C bus. This symbol is then followed in sequence, bit-serially, by the target device address, followed by an R/W symbol (indicating whether the master wants to be a receiver [R] or sender [W] of data, during the communication). All other devices on the bus are considered as "slaves": they all monitor this bus communication, and determine if the master is trying communicate with them, i.e. if their unique address matches the one broadcast by the master.

There are two possible outcomes after the master broadcasts the desired address: (i) a given slave unit finds that the broadcast address is different from its own unique address, or (ii) the address matches its own address. In case (i), such a slave basically does no more processing, except to wait for a final STOP signal.

In case (ii), the slave has determined that the master wants to communicate with it. There are two subcases, depending on whether the master issues (ii)(a) a "read" (i.e. receive) request, or (ii)(b) a "write" (i.e. transmit/send) request. In cases (ii)(a) and (ii)(b), the sender sends data bytes to the receiver. Each data byte is transmitted bit-serially, i.e. one bit at a time, in a designated order. An ACK symbol is then usually transmitted, defining the end of the byte. In general, a number of bytes may be sent during the given transaction between the master and slave. Finally, after the final data byte is sent, a STOP symbol is generated.

In this assignment, you will design two different FSM's for a master control unit. Each FSM will handle both common modes: *slave as sender* and *slave as receiver*. That is, you will be handling both **read mode** (i.e. slave as transmitter/sender) and **write mode** (i.e. slave as receiver). In both cases, the designated slave communicates with the master, either sending or receiving data bytes bit-serially, following the I2C protocol, until a final STOP signal is received from the master.

I2C Bus Symbols. The master and slave communicate with each other through encoded bus values of SDA and SCL. In total, there are 6 key events that can occur on the bus. These events are start (S), stop (P), acknowledge (ACK or A), not acknowledge (NACK or $\overline{A}$), data transfer (either 0 or 1 symbols), and repeated-START (Sr). A novel encoding scheme is used. For details on the use of ACK and NACK, and other symbols, see the NXP manual (Handout #27d) and other readings.

In this project, you will produce two designs of the I2C master controller.

For your version #1 design, which only handles error-free communication, you will not use the repeated-START (Sr) or not acknowledge (NACK or $\overline{A}$) symbols.

For your version #2 design, which handles both error-free and erroneous communication, repeated-START (Sr) and not acknowledge (NACK or $\overline{A}$) symbols can be used. In particular, whenever a parity error is detected by the receiver in any data byte (detected by master [in read mode] or slave [in write mode]), the receiver sends a NACK to the transmitter after this data byte. In this case, NACK is a general error or failure flag. The master may then *either* (i) terminate the transaction (using STOP), or (ii) re-try the transaction (using a repeated-START [Sr]). In particular, the master will first initiate a re-transmission on the same byte; if the result again is a NACK, the master will immediately terminate the transaction.

NACK Symbol: Further Details. When a receiver sends a NACK symbol, assumes that it must *drive* the SDA bus to the appropriate value.

Note: Handout #27a, p. 5, item 8(b) suggests that during normal transmission (i.e. no errors), when master

[1] However, in many real-world systems, only one fixed device is allowed to be the master.

is receiver, after receiving the last data byte, the master omits any final ACK and immediately sends a STOP. *You should ignore this comment!* Instead, for this assignment, follow the NXP manual (Handout #27d), which indicates that *every data byte* must be followed by some form of acknowledgment (ACK or NACK).

Note: There are alternative scenarios, such as 'no ack' (which is different from NACK), described in some of the handouts, where the SDA line floats. This is a form of lack of response which also requires error handling. However, you should not consider this scenario. Instead, for this assignment, the receiver should *always* drive the SDA bus for both ACK and NACK symbols.

Overview of Assignment: 2 Master Controller Designs. For this project, you will produce 2 versions of the master FSM.

Version #1 will support the basic communication protocols outlined in this handout, but **without error handling.** Each data byte transmitted will include 8 data bits. This is a baseline design, which assumes correct data.

Version #2 includes all the functionality of Version #1, but also **with error handling.** You will assume that each data byte sent or received uses an *even parity code*. That is, the 8-bits are divided into 7 data bits and 1 final even-parity bit. In "read mode", the master will check each byte for its parity. In "write mode", the slave will check each byte for its parity. In each case, if there is no error, a normal acknowledgment will be transmitted exactly as in version #1. However, in both read and write modes, a NACK must always be sent by the receiver after an error is detected in any data byte.

Immediately after an error is detected, the master has two options: (i) immediately *terminate* the communication (using a STOP symbol, following the given protocol); or (ii) initiate a *retransmission* between the master and slave. For (ii), the NXP manual (Handout #27d) gives information on how a retransmission is initiated (using an Sr symbol). Once the NACK has been produced, the master will either terminate the transaction (using STOP) or initiate a re-transmission (using Sr). *For this assignment, you will enforce a limited re-transmission policy:* after a NACK is generated, the master will first initiate a re-transmission on the same byte; if the result again is a NACK, the master will then immediately terminate the transaction.

In your design, there wil be different handling of parity generation and detection, depending on the mode:

For "write mode", with master as transmitter, assume the parity bit is generated externally, as part of the data stream, by the local unit. That is, the seven data bits and one parity bit are passed *directly* to the master FSM on the "*bit_send*" input (see next section), so the master FSM does not need to compute the parity bit.

However, for "read mode", with master as receiver, *parity must be checked directly by the master FSM,* which will determine if a correct or incorrect parity bit is received. That is, in write mode, your FSM specification itself must explicitly identify correct or incorrect parity, for each received byte, and take appropriate action. In this case, you *cannot* assume that external local hardware in the master unit is analyzing the parity bit and providing an outcome to the FSM.

Note: As a design strategy, first focus on completing a working and correct Version #1 specification. Only when this is completed and debugged, should you start on Version #2. The latter builds *directly* on the former, with only small but subtle modifications.

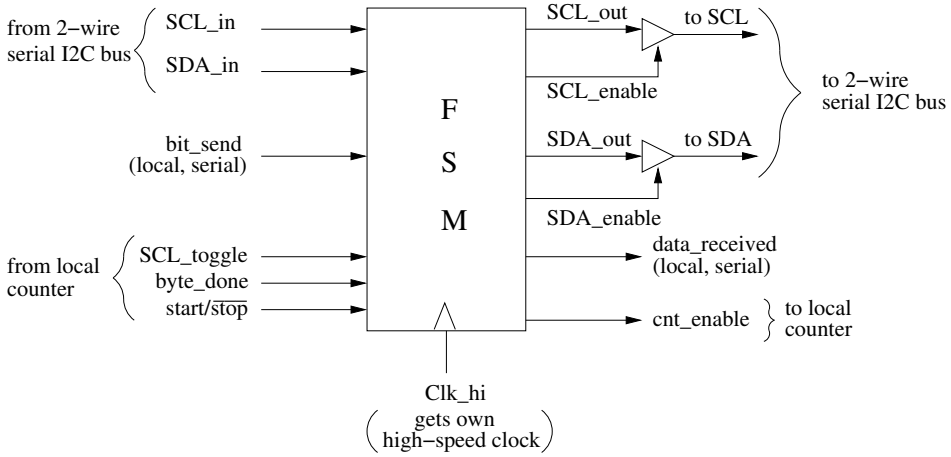
Grading: If you provide a complete solution to Version #1 only, you will receive up to 75% of full credit. The final 25% of credit is allocated for completing Version #2 with its support for fault tolerance, integrated into an FSM which also supports all the basic Version #1 scenarios.

Target Control Microarchitecture. A block diagram of the master's main controller that you are designing is shown in the figure below. The master also has other external hardware, including a local counter (discussed in detail below) as well as components which supply address or data bits to transmit and process data bits that are

received. *You will not design these other components, just the master's main controller.*

The master's FSM has *3 input channels,* containing a total of *6 input signals.* In addition, the controller operates using its own local high-speed clock, *clk_hi*. One input channel monitors activity on the I2C bus (*SCL_in, SDA_in*). The second input channel receives data or address bits from the rest of the master unit to be sent out, bit-serially, on the I2C bus (*bit_send*). The third input channel receives control signals from a local counter unit (*SCL_toggle, byte_done, start/*$\overline{stop}$), which provide the master control with useful information, such as when to generate a transition on the SCL clock, when a byte is done, and when the master unit should be enabled or disabled.

The master FSM also has *3 output channels,* containing a total of *6 output signals.* One output channel generates outputs to the I2C bus (*SCL_out, SDA_out*), as well as tristate controls for these signals (*SCL_enable* and *SDA_enable*, respectively). The second output channel outputs data bits it has received from the I2C bus, and sends them, bit-serially, on an output wire (*data_received*) to the rest of the master unit for processing (not shown, you do not need to design this other hardware). The third output channel sends an enable signal to the local counter unit, to update its count (*cnt_enable*).



Inputs *SCL_in* and *SDA_in* are the bi-directional I2C bus wires; they come directly from the global I2C bus. Any changes on the I2C bus are always observable by the master controller on these 2 input wires at all times.

Outputs *SCL_out* and *SDA_out* are also connected directly to the global I2C bus, but through tristate buffers, as shown in the figure). These tristate buffers are enabled by the master's outputs *SCL_enable* and *SDA_enable*, respectively. Before the master wins arbitration, it is inactive, and disconnected from the I2C bus, i.e. both tristate buffers are disabled. Once the master becomes active, i.e. wins arbitration, its controller is responsible for generating and driving the SCL clock on the I2C bus, on output SCL_out. Hence its tristate enable must be asserted high for the entire transaction. Once the master has completed the entire transaction (i.e. sending a STOP signal), it disables itself (SCL_enable and SDA_enable deasserted low), and thereby no longer generates SCL clock pulses on the bus. Whenever the master needs to drive the value of the SDA bus, it does so by asserting the SDA_enable high and transmitting a value on its output SDA_out. However, whenever the slave is driving the SDA bus, the master must disable its connection to SDA (i.e. SDA_enable deasserted low).

The input *bit_send* receives inputs from the rest of the local master unit's hardware. In the I2C bus protocol, when the master needs to transmit an address on the bus, the binary address bits are supplied, bit-serially, on this input signal. Likewise, when the master is in "write" mode (i.e. slave as receiver) and needs to send a data byte, the binary data bits are supplied, bit-serially, on this input signal.

The output *data_received* is used when the master is in "read" mode (i.e. slave as sender). In this mode, each data bit received on the I2C bus, bit-serially, is translated to a normal bit-serial binary output (0 or 1) on output wire data_received, and sent to the rest of the master unit for processing. (Again, you are not responsible for

designing the rest of the master unit.)

Finally, a separate local counter is assumed, which interacts with the master controller (discussed later). Do not design this counter, but you need to carefully understand how it operates, because it receives an output of the master controller, and provides three inputs to the master controller. In particular, the master FSM has an output *cnt_enable*, going to the local counter, which approximately follows the waveform of the SCL_out clock. The master FSM also has three inputs from the local counter: *SCL_toggle, byte_done* and $start/\overline{stop}$. The *SCL_toggle* input is a request to the master control to toggle the SCL system clock (i.e. SCL_out). The *byte_done* input indicates whenever a complete address byte (along with R/W) is received; it also indicates whenever a complete data byte is sent as output (master in write mode) or received as input (master in read mode). This signal will help you to simplify your FSM specification, since you will not have to keep track of the number of bits sent or received; the signal will tell you when a byte (transmitted or received) is complete. Finally, the $start/\overline{stop}$ signal indicates to the master controller when it is activated to start an entire transaction (asserted high), and when the transaction is complete (deasserted low).

SCL Input: A novelty of the I2C protocol is that SCL is itself a clock signal, but each receiver FSM treats it as a *data input.* Basically, the combination of SCL and SDA inputs determines what symbol is on the I2C bus.

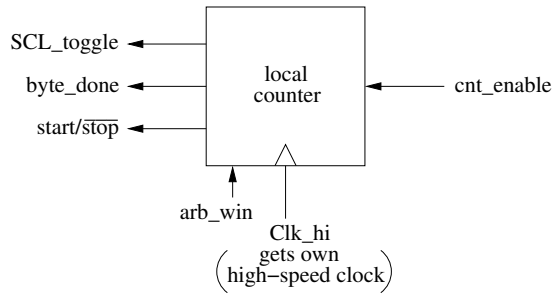
Note that the SCL input is identical to the SCL output (to right of tristate buffer), i.e. this input and output are directly attached to the global SCL bus. Similarly, the SDA input is identical to the SDA output (to right of tristate buffer), i.e. this input and output are directly attached to the global SDA bus. Likewise, all slave units are attached to both SDA and SCL buses.

Local Controller Clock: Note that, locally, your FSM has its own *distinct high-speed clock, Clk_hi*. This local clock has nothing to do with the bus clock, SCL, and you should assume it operates at a much higher rate than SCL. The SCL clock is produced by the master at a slower rate: several Clk_hi clock cycles for the high period of SCL, and several Clk_hi clock cycles for the low period of SCL. Basically, each FSM "samples" the input SCL at its own higher local clock rate. Therefore, you can consider SCL just as a normal data input that is being monitored by your FSM (with your FSM monitoring it under a high-speed local clock, Clk_hi).

Your Moore FSM is controlled only by its *Clk_hi* clock, as shown in the above figure.

Interaction with the Local Counter: An important component of the master is a small local counter, as shown in the figure below. While you will not be designing the counter, it is important that you understand its role and timing, since the master control interacts closely with it. You should assume this counter is a Moore machine.

The local counter generates three critical control signals to the master controller: (i) to activate a transition on the system clock SCL (*SCL_toggle*); (ii) to indicate when an address or data byte is complete (*byte_done*); and (iii) to initiate the start and stop of the master's entire transaction ($start/\overline{stop}$). Each of these signals will simplify the master control, by keeping track of important events. The *SCL_toggle* signal also is used by the master controller for SCL clock generation.



Generating the System Clock, SCL_out: Basic Operation. The interaction of two signals, *cnt_enable* and *SCL_toggle* are used by the master control to generate the pulses of the system clock, SCL. Basically, the

counter counts a number of cycles of the local high-speed clock, *Clk_hi*, which determines how long SCL is low and how long SCL is high. In particular, the master control's output *cnt_enable* is an input to the local counter. This signal is similar to the *SCL_out* output of the master control: when SCL_out is asserted high, cnt_enable is asserted high; and when SCL_out is deasserted low, cnt_enable is deasserted low. However, there are some important timing differences between them.

To understand how the local counter is used to generate the slower system clock, SCL, first assume that initially the counter's *cnt_enable* input and *SCL_toggle* output are low. When the master controller asserts *SCL_out* high, i.e. generates the rising edge of the system clock, the separate *cnt_enable* output is also asserted high and sent as input to the local counter. *This rising transition on cnt_enable resets and re-activates the counter*. The counter then counts a fixed number N of clock cycles of its local Clk_hi clock; this count determines the number of local clock cycles (of Clk_hi) for the high period of the system clock SCL.

When the count is complete, the counter asserts its *SCL_toggle* output high, which is an input to the master controller. The master controller (on the next clock cycle) then deasserts *SCL_out* low, i.e. it generates the falling edge of the system clock. It also deasserts the separate *cnt_enable* output low and sends it as input to the local counter. *This falling transition on cnt_enable also resets and re-activates the counter*. The counter then again counts a fixed number N of clock cycles of its local *Clk_hi* clock; this count determines the number of local clock cycles (of *Clk_hi*) for the low period of the system clock SCL. When the count is complete, the counter deasserts its *SCL_toggle* output low which is sent to the master controller. The master controller (on the next clock cycle) then asserts *SCL_out* high, i.e. it generates the next rising edge of the system clock. It also asserts the separate *cnt_enable* output high and sends it as input to the local counter, and so the process continues.

In sum, the above scenario indicates how the cycle of "transition on *cnt_enable*" followed by "transition on *SCL_toggle*" forms a loop which generates the slower system clock SCL pulses. The counter is used to time the number of local clock cycles (on *Clk_hi*) for the low period and high period of SCL. Hence, the SCL clock period is an integer multiple of the local higher speed clock.

Detailed Timing: outputs "cnt_enable" vs. "SCL_out". When *SCL_out* makes a falling transition, *cnt_enable* should make a *falling transition in the same clock cycle*, i.e. concurrently. However, when *SCL_out* makes a rising transition, even *without* clock stretching, then *cnt_enable* should make a *rising transition in the next clock cycle or later*. See "Clock Stretching" below for more details.

Completing a Byte Transmission. The local counter also indicates when a byte transmission is complete. This is a useful feature, which will help you to simplify your master controller specification, because you do not need to record how many bits you are sending or receiving: the signal *byte_done* will indicate when the byte is complete. In particular, when sending an address (including the 8th R/W bit), sending a data byte, or receiving a data byte, the local counter will count the appropriate number of SCL clock cycles. *It will then assert byte_done high in the same local clock cycle (i.e. of Clk_hi) in which the rising transition of SCL_toggle occurs* for the 8th bit of any data/address byte transaction. The byte_done signal will only stay asserted high for 1 Clk_hi clock cycle.

Completing an Entire Transaction. You can assume that the counter is configured or hardwired to produce a final "stop" signal for the transaction, when sufficient data bytes have been read or written. This control is handled by the $start/\overline{stop}$ signal. When the transmission begins, this signal is asserted high. The signal *remains high* for the entire transaction. Finally, along with the last byte of transmission, the $start/\overline{stop}$ signal is *deasserted low* in the same local Clk_hi clock cycle in which *byte_done* is asserted high (for the last time). See "Initialization" below for further details.

SCL and SDA: Default Values. In the I2C bus, when inactive, assume both SCL and SCA are stable at 1 values.

Detailed Initialization: Initially, assume the master unit has not won arbitration and is inactive. Hence, its tristate enables, SCL_enable and SCA_enable, are initially deasserted low. Also cnt_enable is initially low, and SCL_toggle, byte_done and $start/\overline{stop}$ are initially low. There may or may not be activity on the system bus (SCL_in, SDA_in).

Once arbitration is won, SCL_in and SDA_in are available, hence stable at default 1 values. You will not deal with how master arbitration works, but assume that the *arb_win* input to the local counter is asserted high for 1 cycle. The local counter then *asserts* $start/\overline{stop}$ *high*, thereby *activating the master controller*. The master controller, then enables its output tristate buffers and *sets its output signals SCL_out and SDA_out high.* It also *asserts output cnt_enable high 1 cycle later* (following the protocol given above). At this point, the local counter initiates a new counting sequence.

Your Task. You are to design and simulate a two symbolic Moore controller specifications for a master device on the I2C bus protocol, where the master can be either a receiver or a sender/transmitter. Version #1 will include no fault tolerance, and Version #2 will support simple fault tolerance: error detection, followed by retransmission or termination.

First, carefully read the Handouts #27, and #27a through #27e, as well as any piazza postings.

Next, create the Version #1 single Moore state diagram specification for the FSM of the master, covering the two cases specified above: (i) master as sender; and (ii) master as receiver.

Then, create the Version #2 single Moore state diagram specification for the FSM of the master, now including fault tolerance, as specified above, while handling the same two cases as above.

Your two specifications must handle the I2C protocol correctly. Carefully go over this handout, #27, to make sure you are following all assumptions and requirements correctly, as well as using #27a through #27e for further clarification.

Next, you are to model your two FSM specifications in VHDL, using one of the two Moore FSM templates in the assigned Brown/Vranesic reading. Finally, using the Altera Quartus II CAD tool, you will simulate your VHDL specification on sequences of input vectors. *Some sequences of input vectors will be provided in the next couple of weeks.*

What To Turn In? You are to turn in all documentation for your design derivation including the following. *Further details on testing, how to submit, etc., will be provided in a few days.*

(i) *Version #1 Design:* The symbolic state diagram of the Moore FSM neatly handwritten and labeled, which assumes no error detection;

(ii) *Version #1 Design:* Printout of VHDL code for the above FSM;

(iii) *Version #1 Design:* Printout of waveforms that result from your simulations for the above FSM;

(iv) *Version #2 Design:* The symbolic state diagram of the Moore FSM neatly handwritten and labeled, which includes support for error detection;

(v) *Version #2 Design:* Printout of VHDL code for the above FSM;

(vi) *Version #2 Design:* Printout of waveforms that result from your simulations for the above FSM;

(vii) Email attachments of (ii), (iii), (v) and (vi);

(viii) Summary explaining your design experiences, testing methods, and challenges that you encountered (0.5-1.0 pages).