Project #1: Designing a Slave Controller for the Philips I2C Bus Protocol

This homework is due on **Friday, November 20, 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 "slave" control 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 slave controller, model the FSM specifications in VHDL, and simulate them using the Altera 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 #29), 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 #29a. I2C Configuration and Protocol.* A simple overview of the basic protocol.
- *Handout #29b. 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 #29c. I2C: Getting Acknowledge from a Slave (as Receiver.)* Some useful details of one transmission scenario. (Many more are included in Handout #29b link above.)
- *Handout #29d. I2C-Bus Specification and User Manual (Rev. 5, Oct. 2012) [NXP Semiconductors]).* A complete reference manual from NXP Semiconductors on the protocol. Many of the relevant details for this assignment are already covered in Handouts #29 and #29a-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, arbitration, clock stretching, 10-bit extended addressing, 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.
- *Handout #29e. Frequently-Asked Questions (FAQ) + Online Discussion: Piazza.* Handout #29(e) is a detailed initial FAQ, answering some basic questions and providing important details of requirements and hardware specifications. 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 and web resources 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 #29(d), sec. 4.3), intelligent platform management (IPMI, Handout #29(d), sec. 4.4), and thermal management between boards (ATCA, Handout #29(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 (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: Basic Operation. 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. (In many actual applications, though, only one device is designated as the master.) 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 a single FSM for a master control unit, handling both common modes: *slave as sender* and *slave as receiver*. That is, you will be handling both case (ii)(a) **read mode** (i.e. slave as transmitter/sender) and case (ii)(b) **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 as discussed above and shown in the above handouts and web pages), until a final STOP signal is received from the master. The FSM will also handle case (i) above, where there is an address mismatch.

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. Details are given in the assigned readings.

For details on the use of ACK and NACK, see the NXP manual (Handout #29d).

Note: Handout #29a, p. 5, item 8(b) suggests that during normal transmission (i.e. no errors), when 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 #29d), which indicates that *every data byte* must be followed by some form of acknowledgment (ACK or NACK).

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

For your version #1 design, which only handles error-free communication, you will not use the repeated-START (Sr) symbol.

For your version #2 design, which handles both error-free and erroneous communication, repeated-START

(Sr) 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]).

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

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 Slave Controller Designs. For this project, you will produce 2 versions of your slave 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.

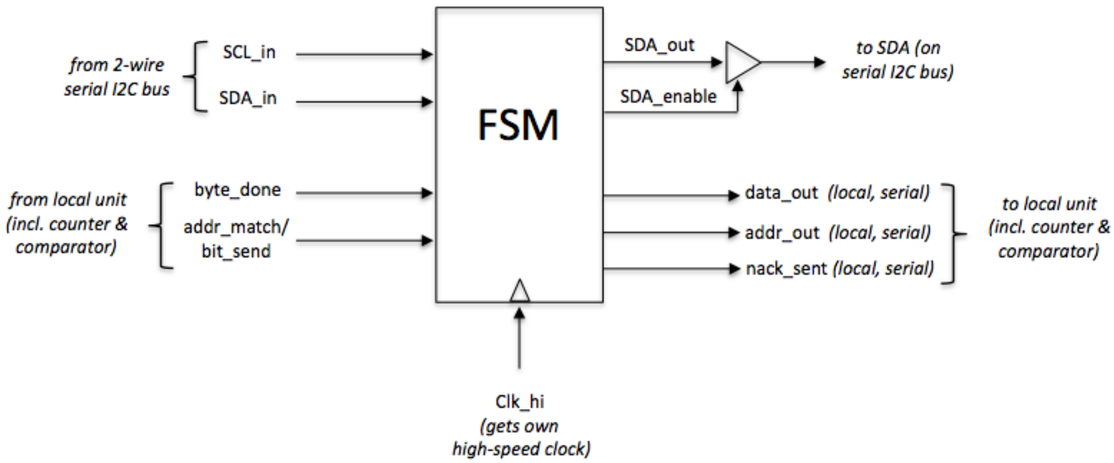
In "read mode", the master will check each byte for its parity. In "write mode", the slave will check each byte for its parity. In each case, if there is no error, a normal acknowledgment will be transmitted exactly as in version #1. However, if there is an error, a NACK symbol will be always transmitted. 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 #29d) gives information on how a retransmission is initiated, through the use of an Sr symbol.

For this assignment, we assume different handling of parity generation and detection. For "read mode", with slave 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 slave FSM on the "*bit_send*" input (see next section), so the slave FSM does not need to compute the parity bit. However, for "write mode", with slave as receiver, *parity must be checked directly by the slave FSM,* which will determine if a correct or incorrect parity bit is received. That is, in write mode, your FSM 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 slave unit is analyzing the parity bit and providing an outcome to the FSM.

Target Control Microarchitecture. A block diagram of the FSM that you are designing in the above figure. This FSM has 4 inputs (SDA_in, SCL_in, addr_match/bit_send, byte_done) and 5 outputs (SDA_out, SDA_enable addr_out, data_out, nack_sent). It also has as another input its local high-speed clock, Clk_hi. Each input and output is described below.

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 slave controller on these 2 input wires at all times.

For simplicity, you should assume there exists a separate local comparator and counter unit, as part of the rest of the slave unit, next to the FSM to generate the addr_match and byte_done signals, respectively. Do not design this local unit, assume it already exists and functions correctly. (Further details will be included in the FAQ, Handout #29(e).)



The input *addr_match* receives signals from the output of the local comparator. The comparator takes in the device address sent from the master, which is processed by the FSM, and sent serially by the FSM to the local comparator on the 'addr_out' output wire. Once the complete address has been received on the I2C bus by the slave, and sent by your controller bit-serially to its comparator, the comparator compares that address with the device address of the slave which comes from *addr_out*. If the addresses match, addr_match is a 1, otherwise it is a 0. Once the address match is complete, the addr_match input is ignored for the rest of the transmission. However, there is one exception: if the slave is a transmitter, this input wire (also labelled *bit_send*) is then used to provide serial data bits to the controller to encode and place on the I2C bus (see below). Assume the serial data is provided locally by the rest of the slave unit (which you are not designing), on the *bit_send* input.

The input *byte_done* (active high) is the output of a local counter and an input to the FSM, as shown in the block diagram. This local counter simply monitors the global SCL bus line, and signals to the FSM whenever a complete byte is received: either a complete address and R/W bit at the start of the protocol, or a complete data byte throughout the remainder of the protocol. *You do not need to design this counter, nor to simulate it; just assume it is there, generating input* byte_done *to the FSM at appropriate times.* In particular, just assume that the FSM gets the input flag, byte_done, asserted high just before an acknowledgment symbol (ACK or NACK) needs to be transmitted. (In your Altera simulations, just make sure your input vector sequence has byte_done asserted at the proper times in sequence.) This byte_done signal should help you simplify your FSM design, since your FSM will not have to keep track of when you have completed the processing of an address + R/W bit, or a data byte: it will be informed by this external signal.

Output *SDA_out* is connected to the global I2C bus through a tristate buffer, as shown in the figure. The tristate buffers are enabled by the slave's output *SDA_enable*. In general, whenever the slave drives the SDA bus, it does so through its tristate buffer. This is performed using *SDA_enable* as the enable of a tri-state buffer where the value on *SDA_out* is sent to SDA. Whenever the slave is not driving the SDA bus, this buffer should be disabled and allowed to float. See the required handouts for further details.

When the slave is a receiver, the output *data_out* is used to send the data bits read from the I2C bus to the slave unit. Each 2-wire encoded 0 or 1 input symbol received from the I2C bus (bit-serially) is translated to the corresponding 1-bit binary value, which is output (bit-serially) on data_out.

When the slave is a transmitter, assume that its local processor provides the desired data in bit-serial fashion on the *bit_send* input signal. (Note: this is the same input wire as *addr_match,* discussed above, but now used for a different purpose, through external multiplexing – which you can assume is present, and do not need to provide.) You can assume that, at the appropriate point in the protocol, the *bit_send* input contains the desired input bit to encode and transmit, and your FSM specification should perform the appropriate encoding and transmission (only for this case of slave as transmitter).

Finally, the output wire *nack_sent* is a flag for error handling. For your first design, which has no error handling, this signal always remains 0 and is ignored. However, for your second design, which supports error detection, this signal is used to inform the rest of the slave unit whenever a NACK is transmitted on the I2C bus. In particular, at the end of any data byte transmission, in your second design, *regardless* of whether the slave is in read mode or write mode, your controller will assert *nack_sent* to 1 for 1 local clock cycle, between the transmission of a NACK symbol on the I2C bus and the start of next SCL clock pulse. The local slave unit can use this signal to reset itself and prepare for a retransmission of the data. (You do not need to work out details or design of the rest of the slave unit, but the above indicates why it is useful for your slave controller to assert this output signal as an error flag.)

Support for Fault-Tolerance: Error Detection and Retransmission. As indicated above, under "Overview of Assignment: 2 Slave Controller Designs", you will be completed two slave FSM specifications for this assignment, *Version #1* without fault tolerance, and *Version #2* with fault tolerance.

The Version #2 design includes support for error detection of data bytes, using even parity codes. This version will also support two options in the case of an error detected in a data byte: immediate termination of transaction, or retransmission. In both read and write modes, a NACK must always be sent by the receiver after an error is detected in any data byte. Once the NACK has been produced, the master will either terminate the transaction (using STOP) or initiate a re-transmission (using Sr).

As indicated in the previous section, for the Version #2 design, in "write mode", your slave FSM must *directly determine the parity of each received data byte*. That is, you are *not allowed* to have a separate parity unit in the rest of the slave unit to check parity. Instead, your FSM itself must determine the parity of each received data byte. However, in "read mode", assume your slave FSM is simply provided the correct parity bit from the external local hardware, as part of its data inputs (i.e. on "addr_match" input).

Note: As a design strategy, you should focus first 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.

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.

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.

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

NOTE: For this assignment, you do not need to know about or implement special extended address modes, high-speed modes (fast mode, fast mode-plus, ultra-fast mode), clock stretching techniques, synchronization, or bus arbitration. You can read on these topics, but just follow the more limited problem that we are asking you to solve, assuming the valid basic I2C protocol is observed.

Your Task. You are to design and simulate a two symbolic Moore controller specifications for a slave device on the I2C bus protocol, where the slave 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 termination or retransmission.

First, carefully read the Handouts #29, and #29a through #29e, as well as any piazza postings.

Next, create the Version #1 single Moore state diagram specification for the FSM of the slave, covering the three cases specified above: (i) no address match to slave; (ii)(a) address match, slave as sender; and (ii)(b) addresses match, slave as receiver.

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

Your two specifications must handle the I2C protocol correctly. Carefully go over this handout, #29, to make sure you are following all assumptions and requirements correctly, as well as using #29a through #29e 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).