

## HW#5: Designing a Master Controller for the Philips I2C Bus Protocol

This homework is due on **Friday, November 18, at 4pm** (submission details to be announced soon). *Note the new 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 a single *Moore state diagram specification* for a master controller, model the FSM specification in VHDL, and simulate it using the Altera CAD package.

**Grading.** Since this homework is larger than some of the other homeworks, it will be worth 8% of your final grade.

**Working in Groups.** You are allowed to do this homework in a group-of-two. You both get the same grade. However, solo homeworks are also allowed.

**Required Reading:** Background information and a description of the bus protocol are outlined below. Supplementary handouts, Handouts #29a and #29b, on the bus protocol are also provided, as well as a supplementary URL listed as Handout #29c from the *Esacademy*. Handout #29 gives the overview of the assignment, with some important details on the bus protocol and implementation. Handout #29a is a short slide presentation giving a nice overview of the I2C bus protocol. Handout #29b focuses in detail on the protocol for getting an acknowledge from a slave (slave as a receiver). Handout #29c, from the *Esacademy*, has links to a number of useful pages, including overview and history of the I2C protocol, various parts of the protocol, clock stretching, and other techniques. One of these pages under #29c, covers how the master gives an acknowledgment to a slave unit (slave as sender), which is the opposite transmission direction of that in Handout #29b:  
<http://www.esacademy.com/en/library/technical-articles-and-documents/miscellaneous/i2c-bus/i2c-bus-events/giving-acknowledge-to-a-slave-device.html>.

*These handouts include detailed explanations of relevant parts of the bus protocol, so be sure to read them carefully. You should be able to do the entire assignment reading only Handouts #29, #29a, #29b and #29c (as well as the above web page URL on how the master ack's a slave).*

**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 reading in these manuals (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.)!

- (a) *NXP I2C Specification and User Manual* (from 2007) gives fairly complete coverage of many details (but see note above, many of the advanced modes and features that it covers will be ignored for this assignment), and is a good background reference manual for clarifications:  
[http://www.nxp.com/documents/user\\_manual/UM10204.pdf](http://www.nxp.com/documents/user_manual/UM10204.pdf);
- (b) *Philips Application Note, I2C Manual* (from 2003, now from NXP), which gives additional technical details. It also opens with a nice overview of the practical industrial applications of the I2C bus:  
[http://www.nxp.com/documents/application\\_note/AN10216.pdf](http://www.nxp.com/documents/application_note/AN10216.pdf)

**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 several embedded applications. 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.

**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. However, the basic idea is that the connected devices can force a low value on the serial wires if desired. 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.

In several of the above 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, forcing a delay in the SCL rising transition sufficiently long for a slow slave device to process at its own rate. This nice feature allows the assembly of a system with units operating at different rates, and still having them function correctly together. This is one feature you will explore and support in 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, any connected device has the ability to be the master, however *only* one device can be the master at a particular time. Hence an "arbitration phase" occurs, before a transmission, where any competing masters must contend for one to win control of the bus. You will *not* deal with the arbitration phase 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*

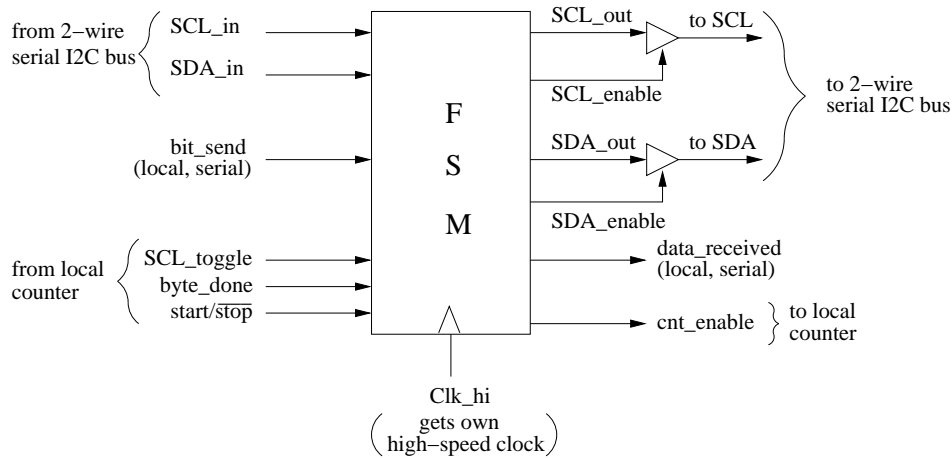
as sender and slave as receiver. That is, you will be handling both **read mode** (i.e. slave as transmitter/sender) and **write mode** (i.e. slave as receiver). In both cases, the designated slave communicates with the master, either sending or receiving data bytes bit-serially (following the I2C protocol as discussed above and shown in Handout #29a and the designated web page), 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 4 events that can occur on the bus. These events are start (S), stop (P), acknowledge (ACK), and data transfer (either 0 or 1 symbols). A novel encoding scheme is used. Details are given in the assigned readings. For details on ACK, see “Required Reading” above (for the two cases of slave as receiver and transmitter).

**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 hardware, including a local counter (discussed in detail below) as well as other 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/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*).



Each input and output is described in detail below, grouped together by functionality.

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 (see later for more details). You are not to design this counter, but 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/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/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.

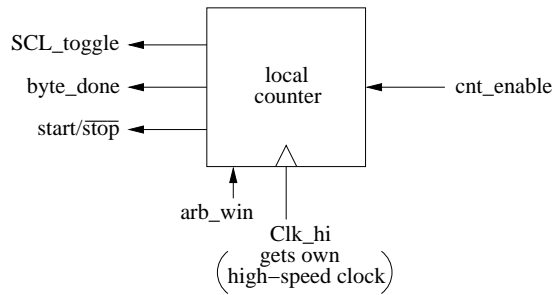
**Local Controller Clock:** Note that, locally, your FSM has its own *distinct high-speed clock*, *Clk\_hi*. This local clock has nothing to do with the bus clock, SCL, and you should assume it operates at a much higher rate than SCL. The SCL clock is produced by the master at a slower rate: several *Clk\_hi* clock cycles for the high period of SCL, and several *Clk\_hi* clock cycles for the low period of SCL (see below for more details).

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/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.

**Clock Stretching:** For this assignment, you are to implement clock stretching for the master controller. This feature enables a slow slave to put back-pressure on the master, and give it enough time to complete processing of a particular symbol.

In particular, when the system clock SCL is low, and the master attempts to assert it high (i.e. *SCL\_out* asserted high), if the slave is not ready, the slave may *hold SCL low*. In this case, *even though the master has asserted SCL\_out high, the actual SCL bus value will remain low*. The actual bus value can be checked by the master control on the *SCL\_in* input. The master will continue to assert its *SCL\_out* high. Once it detects the appropriate release by the slave of the SCL bus, i.e. *SCL\_in* appears high, the master will *finally assert cnt\_enable high*. That is, only after SCL is seen to go high, the master will then set its *cnt\_enable* output high, which as input to the counter will reset it and activate the next counting sequence. The net result is a “stretching” of the low phase of SCL, as long as needed by the slave unit. Hence, the starting of the next counting sequence is delayed until the slave is ready.

As a consequence, the master output signals *SCL\_out* and *cnt\_enable* have an asymmetric timing relation. Both *SCL\_out* and *cnt\_enable* are deasserted low in the same local Clk\_hi cycle. However, *with no clock stretching, cnt\_enable* is asserted high *1 cycle after SCL\_out* is asserted high. And if clock stretching has occurred, *cnt\_enable* may be further delayed for several more local clock cycles.

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

**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/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.

**NOTE:** For this assignment, you do not need to know about or implement special extended address modes, fast modes, broadcast modes, electrical issues, error conditions, missing responses 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, and there are no errors, failures or arbitration issues.

**Your Task:** You are to design and simulate a single Moore controller specification for a master device on the I2C bus protocol, where the master can be either a receiver or a sender/transmitter.

First, carefully read the Handouts #29, #29a, #29b and #29c, as well as the above given web link on the case of giving an ACK to the slave.

Next, create a single Moore state diagram specification for the FSM of the master, covering the two cases specified above: **case (i) “read” mode** (i.e. slave as transmitter/sender); and **case (ii) “write” mode** (i.e. slave as receiver).

Your specification must handle the I2C protocol correctly (carefully going over this handout, #29, to make sure you are following all assumptions and requirements correctly, as well as using #29a, #29b and #29c for fur-

ther clarification). Next, you are to model the FSM in VHDL, using a Moore FSM template 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 week.*

**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) The symbolic state diagram of the Moore FSM neatly handwritten and labeled;
- (ii) Printout of VHDL code for your FSM;
- (iii) Printout of waveforms that result from your simulations;
- (iv) Email attachments of (ii) and (iii);
- (v) A short paragraph explaining design experiences, testing methods, and challenges.