
I2C MODULE

This lab has you implement an I2C module (single-master bus only). This document serves as the lab write-up for both Labs 9 and Labs 10.

BACKGROUND

Review the notes on I2C transmission.

TASK DESCRIPTION

You are to implement an I2C module that is a subset of the full I2C specification and has the following features.

- Supports only one I2C clock frequency (400 KHz)
- Supports only one bus master (i.e., this module is the bus master)
- Supports I2C start condition, I2C byte write + ack read, I2C byte read + ack write, I2C stop condition.

The deliverables for Lab 9 are support of the I2C start condition, I2C byte write + ack read, and software reset capability.

The deliverables for Lab 10 are support of the I2C stop condition, I2C byte read + ack write, and software reset capability.

The design must pass both behavioral and post-route simulations.

The module interface is given below:

```
module i2c(clk, reset, din, dout, wren, rden, addr, sclk, sdout, sdin,
dir);
input clk, reset, wren, rden;
input [7:0] din;
output [7:0] dout;
input [2:0] addr;
output sclk;           //serial clock out
output sdout;          //serial data out
input sdin;            //serial data int
output dir;            //direction control
```

The input/output definition is:

- *clk* – clock input
- *reset* – high true reset
- *rden* – read enable for dout bus
- *din* – data input bus

- *dout* – data output bus, reflects the contents addressed by the *addr* input when *rden* is high, else *dout* is 0.
- *addr* – address bus for internal registers
- *wren* – write line for a register. The register selected by *addr* is written on the rising clock edge when *wren* is high.
- *sclk* - serial clock out, clock speed controlled by the Period Register
- *sdout* – serial data out – output data should appear on this signal
- *sdin*– serial data in – when reading data, read the value on this input
- *dir* – direction signal – ‘1’ when reading data from *sdout*, ‘0’ when writing data to *sdin*. This output must be GLITCH free, which means that it must be driven by a DFF.

Register addressing and reset behavior:

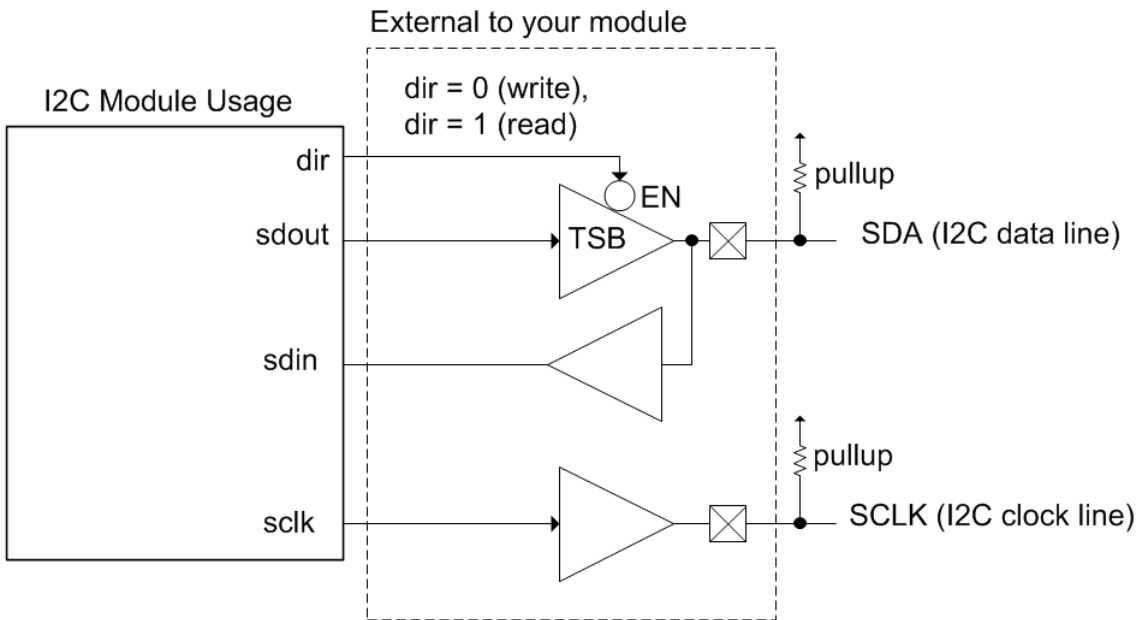
Register	Address	Comments	Value at reset
Period Register	0b00	Period register your timer module that generates the I2C clock. This register can be read/written.	cleared
TX Reg	0b01	Write port for 8-bit data to be sent to <i>sdout</i> . This register can be read/written.	cleared
RX Reg	0b10	Read port for 8-bit register that contains the byte read during a read operation. This register is read only.	cleared
Status/Control	0b11	This is an 8-bit register, can be read/written.	cleared

Status/Control register definition:

- **Bit 0: START** – Start condition control bit. Writing a 1 to this bit causes an I2C start condition, auto cleared after the I2C start condition is finished.
- **Bit 1: STOP** – Stop condition control bit. Writing a 1 to this bit causes an I2C stop condition, auto cleared after the I2C stop condition is finished.
- **Bit 2: WRITE_EN** – Write enable control bit. Writing a 1 to this causes an 8-bit serial transmission from the TX register plus a read of the ACK bit that is sent back (9-bit times). The bit is auto-cleared after the transmission is done and the ACK bit is read.
- **Bit 3: WRITE_ACK** – Write ack bit. This is the value of the ACK bit read after a write has been done, value is either 0 or 1. This is a read-only bit.
- **Bit 4: READ_EN** – Read enable control bit. Writing a 1 to this causes an 8-bit serial read with the value read loaded into the TX register, plus a write of an ACK bit(9-bit times). The bit is auto-cleared after the read is done and the ACK bit has been sent.
- **Bit 5: READ_ACK** – Read ack bit. This is the value of the ACK bit to send after a read has been done, value is either 0 or 1.

- **Bits 6: RESET** – (Software reset) Writing a 1 to this bit halts the current operation and resets the I2C control back to its initial state. Once reset is accomplished, the bit is auto-cleared.
- **BIT 7: Unimplemented**, read as 0.

I2C Module Usage



The above diagram illustrates how the *dir/sdout/sdin/sclk* signals would be interfaced to external SDA/SCLK lines on an I2C bus. The *dir* (direction) signal connects to the enable signal of a tristate buffer and controls direction of data flow on the SDA line (*dir*=0 is write, *sdout* is driving the SDA line; *dir*=1 is read, an external I2C device is driving the SDA line). Data flow on SDA is bidirectional, as during a write operation the I2C module writes 8-bits and then reads back an ACK bit. During a read operation, the I2C module reads 8-bits from an external device, and writes back an ACK bit.

We are implementing a single-master bus I2C module, so our I2C module is always providing the SCLK signal. This means the *sclk* output is always driving the SCLK line on the I2C bus (for a multi-master implementation, this would have to be a tristate buffer as well as another bus master could provide SCLK).

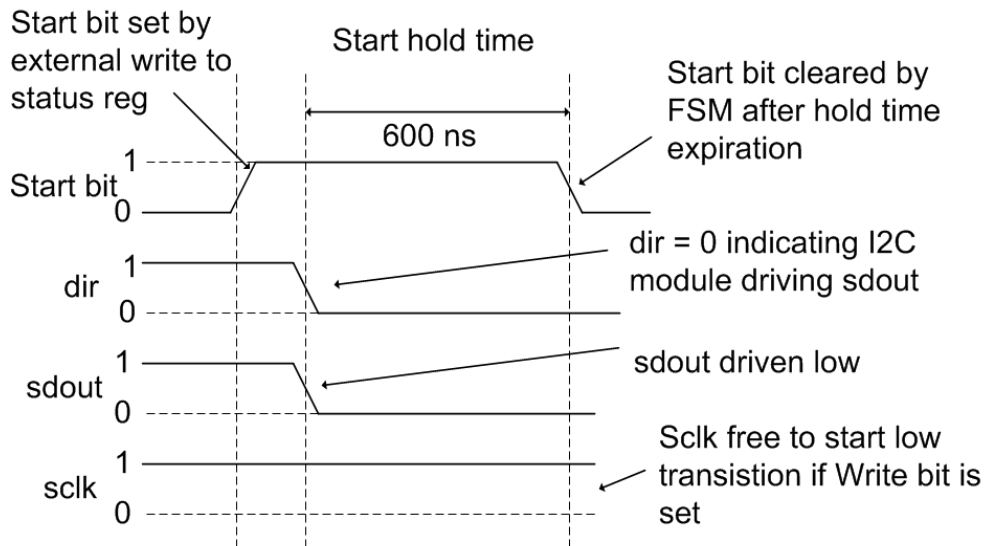
Timing Specifications

This section gives timing specifications relating to Start, Write, Read, Stop actions. All of this timing is for the 400 KHz bus SCLK. These timings would change for other I2C bus standard frequencies (100 kHz, 1 MHz), but to simplify the design, I am asking you only to support these timings. The testbench loosely checks that these timings are correct (it provides a wide margin of tolerance in these timings).

A caveat: our design does not have to prevent illegal I2C bus actions in terms of when a Write occurs versus a Start condition etc. For a legal I2C transaction, a Start condition has to occur,

followed by Write byte, followed by either multiple Write bytes or multiple read bytes, followed by a stop condition. Our implementation would allow a Write before a Start condition, or multiple Write/Read actions in a single transaction, etc.

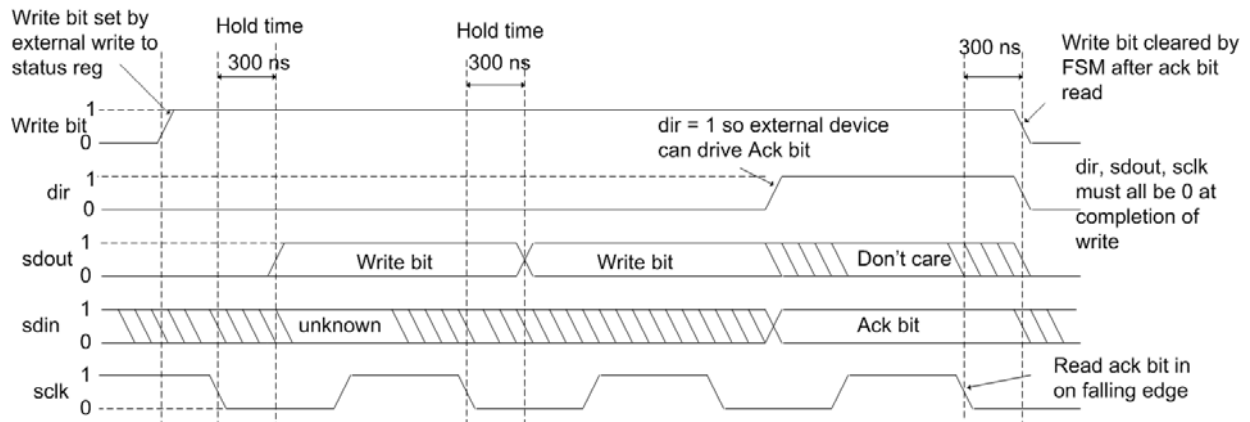
Start Condition Timing



Prior to the start condition, the I2C bus is idle, which means the *sdout*, *sclk*, *dir* signals are all logic 1 (*dir*=1 means the external SDA line is released, i.e., the TSB is not driving it).

The write of a 1 to the start bit in the status register kicks off the start condition. The *dir* signal transitions to 0 so the TSB drives the SDA line, and the *sdout* signal transitions to 0. The condition of *sdout*=0, *sclk*=1 has to persist for at least 600 ns to signal a start condition. At the end of this time, the FSM should clear the Start bit indicating that the Start condition has completed. At this point, the FSM is free to start an I2C write (causing the *sclk* signal to transition low) if the Write status bit is set.

Write data/Ack read Timing

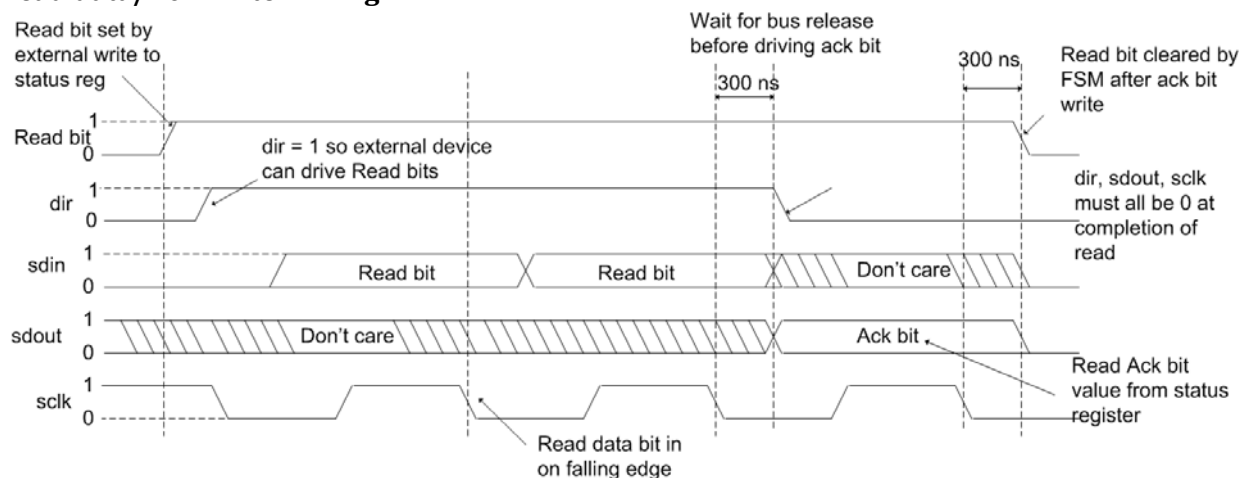


The diagram above shows timing for writing data and then reading the ack. This diagram only shows two data bits; in the actual design there will be eight data bits written before the ack bit is read back from the external device and stored in the Write Ack status bit.

The important details are:

- The new data bit cannot be changed until 300 ns after the falling edge of SCLK. This is a hold time for this data bit after this edge. Changing the data bit 300 ns after the falling edge of SCLK also satisfies setup time to the rising edge of SCLK (this timing not shown).
- When writing data bits, the *dir* signal must be 0 since the I2C module is driving the bus.
- When reading the ack bit, the *dir* needs to transition to a 1 value as shown to allow the external device to drive the data line, and the ackbit is read from the *sdin* signal.
- Once the ack bit is read, the *dir* signal needs to be set back to 0 (our I2C module is driving the SDA line) and the *sdin* signal to 0, and *sclk* to 0 (timer should be stopped at this point since write is completed). These conditions allow a Stop condition to be signaled if requested.

Read data/Ack write Timing

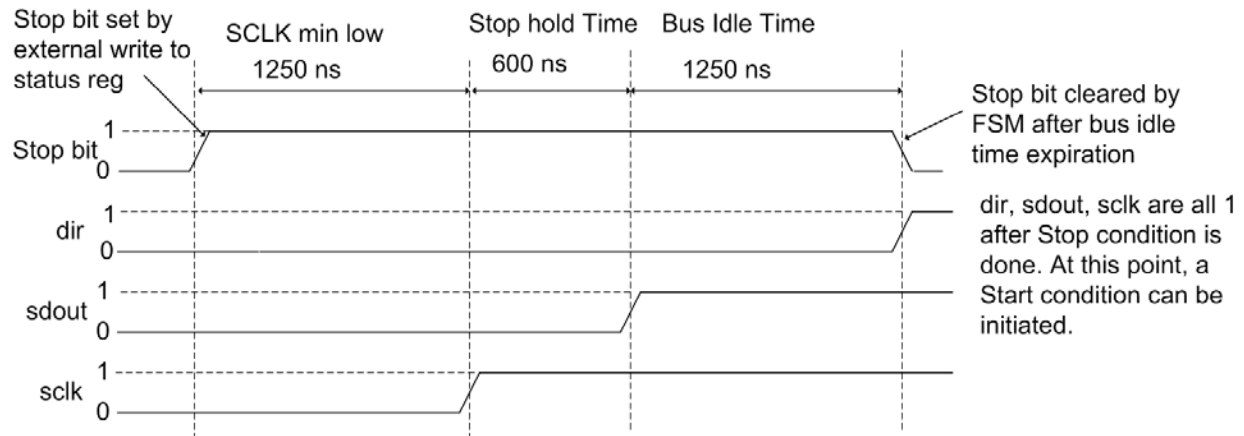


The diagram above shows timing for reading data and then writing the ack. This is simply the write data/read ack diagram with *sdin/sdout* exchanged, and *dir* complemented. This shows two read bits followed by the ack bit; in the real design it will be 8 data bits followed by the ack bit.

The important details are:

- Read data bits are read on the falling edge of the *sclk* signal.
- Do not drive the ack bit until the bus release time has expired after the falling edge of *sclk* (same as hold time, 300 ns). The value of the ack bit written is the Read Ack status bit.
- After the *ack* bit is written, the Read status bit is cleared and the *dir*, *sdout*, *sclk* values must all be 0 (timer should be stopped at this point since read is completed). These conditions allow a Stop condition to be signaled if requested.

Stop Condition Timing



The diagram above shows timing for the stop condition. A stop condition can only follow a read byte or write byte, and we are guaranteed by those timings that *dir*, *sdout*, and *sclk* are all 0 at that point.

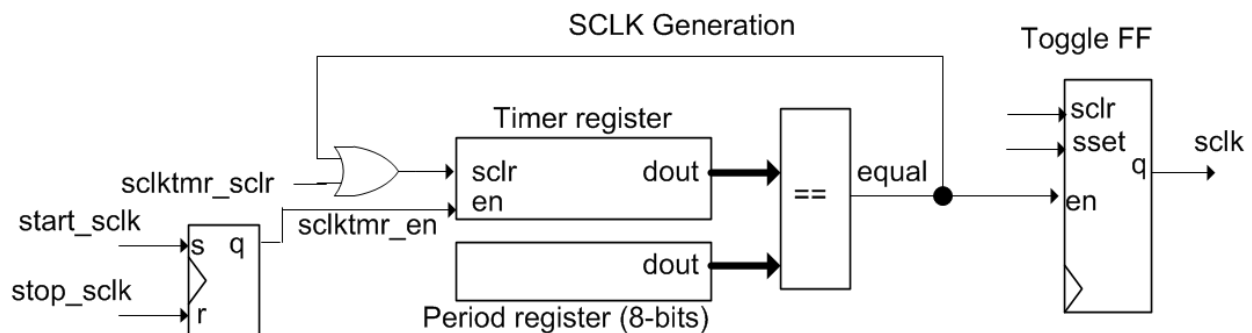
The important details are:

- A stop condition can be signaled immediately after the conclusion of a read/write (immediately after the falling edge of *sclk*). The logic must guarantee that clock remains low for $\frac{1}{2}$ of the SCLK period (2500 ns for 400 kHz), so it must remain low for ~ 1250 ns before transitioning high.
- With *sclk* high, the *sdout* line must remain a 0 for a hold time, which is 600 ns, before transitioning high.
- We must guarantee that the bus remain idle for $\frac{1}{2}$ of the SCLK period, so both *sclk* and *sdout* must remain high for 1250 ns. After this time, the STOP condition is over, so the Stop bit can be cleared in the data register and the bus released (*dir* = 1).

Implementation Details:

Since the I2C specification is somewhat complex, I am going to give you specific implementation details to help you with this design instead of leaving it open. If you want to ignore my implementation details and strike out on your own, feel free to do so, except you must pass the supplied testbench (one caveat: you need to implement the SCLK generation as shown as my testbench expects this when writing the Period Register value).

SCLK Generation:



The figure above shows the preferred SCLK generation/control. An 8-bit timer/period register (PR) is used to generate the clock. The PR is writeable from the external data bus. The *equal* signal goes to a toggle FF that generates the *sclk* signal. This means the 8-bit timer/period register generates a match every half-period of the *sclk* signal. Our system clock is 50 MHz, so a value of 62 written to the period register gives a timeout value of:

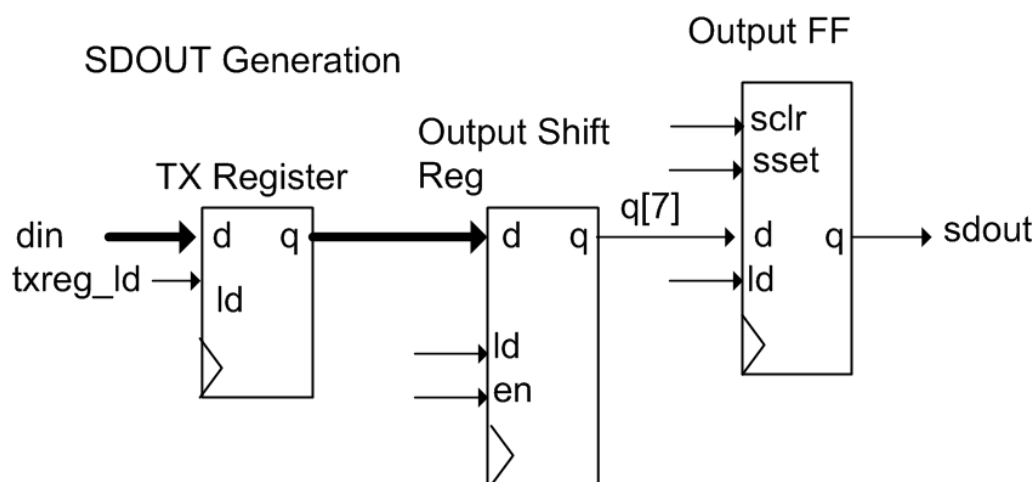
$$\text{Timeout} = (\text{PR}+1) \times \text{Clock period} = (62+1) \times 1/50 \text{ MHz} = 1.26 \text{ us.}$$

$$\text{SCLK frequency} = 1/(2 \times \text{Timeout}) = 1/(2 \times 1.26 \text{ us}) = 396.8 \text{ kHz (close enough)}$$

The timer register needs both synchronous clear and enable signals controlled from your FSM (it is not writeable from the external data bus). My testbench expects this timer implementation so you must implement the SCLK generation in this way.

The toggle FF needs synchronous clear, synchronous set signals driven from your FSM as you must be able to control the state of the SCLK output independently of the timer during generation of Start/Stop conditions. The timer should be disabled (not ticking) when you use the toggle FF *sclr/sset* signals to control the *sclk* value.

SDOUT Generation:

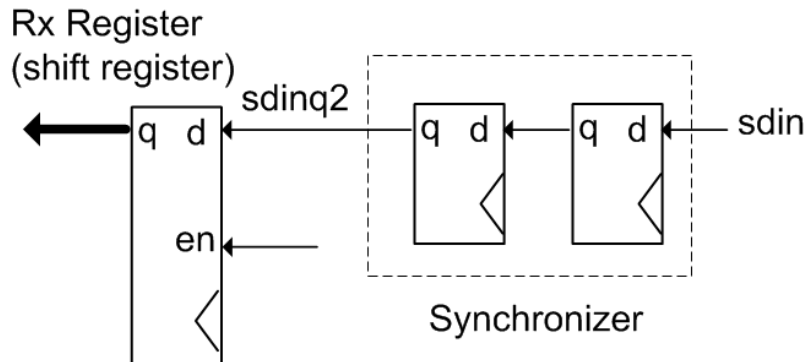


The figure above shows the preferred SDOUT generation/control. The TX register is written from the external data bus. To send 8-bits of serial data, the TX register contents should be loaded into the output shift register. The *en* input on the output shift register is used to do a left-shift of the shift register contents (data is sent MSb first, so the register has to be left-shifted). The Verilog code below implements a shift register with this capability:

```
always @posedge(clk or reset) begin
    if (reset) begin
        q <= 8'h00;
    end else begin
        if (ld) q <= d;
        if (en) q <= {q[6:0], 1'b0}; //left shift
    end
end //end always
```

The MSb of the shift register is tied to the input of the FF that drives the *sdout* signal. The *ld* signal controls loading of the *d* value into the FF. The output FF also has synchronous clear, synchronous set signals driven from your FSM as you must be able to control the state of the SDOUT output independently of shift register during generation of Start/Stop conditions.

SDIN Sampling:



The figure above shows the preferred method of reading the *sdin* signal. The *sdin* signal must go through a 2-DFF synchronizer as shown to protect against metastability since the *sdin* signal is asynchronous to your system clock.

The *sdinq2* signal becomes the input to the LSb of your RX register, this is left-shifted when you input data as data arrives MSb first. The previous code for a shift register can be used, except the *sdinq2* signal is shifted into the LSb instead of a '0' bit.

Status Register

You can figure this out mostly by yourself, but you will need individual signals for synchronous clearing the Start, Write, Stop, Read control bits. You will also need an individual signal for loading the Write ACK bit.

Software Reset Implementation (RESET bit in the status register)

The RESET bit in the status register resets your entire control allowing halting in the middle of a read/write operation. The easiest way to implement this is to use this bit as an alternate synchronous clear for all of your registers (including the status register, so it is self clearing). Also, it should force your FSM back to initial state – this is easily done in the process that implements your state registers – on the active clock edge, if this bit is a one, instead setting present state equal to next state, set it equal to your start state.

Bit Counter

When writing data bits or reading data bits, you need a counter to keep track of how many bits have been written or read. This counter can either be 3 bits or 4 bits, depending on how you check the count.

Utility Timer

You need a simple 8-bit counter for measuring the different times in the timing diagram such as start hold time, write hold time, stop hold time, etc. This can be a simple free running counter that

you synchronously clear to 0 when you need it to start, and then check its count value against a hard-coded count value that will give you the amount of time that you need.

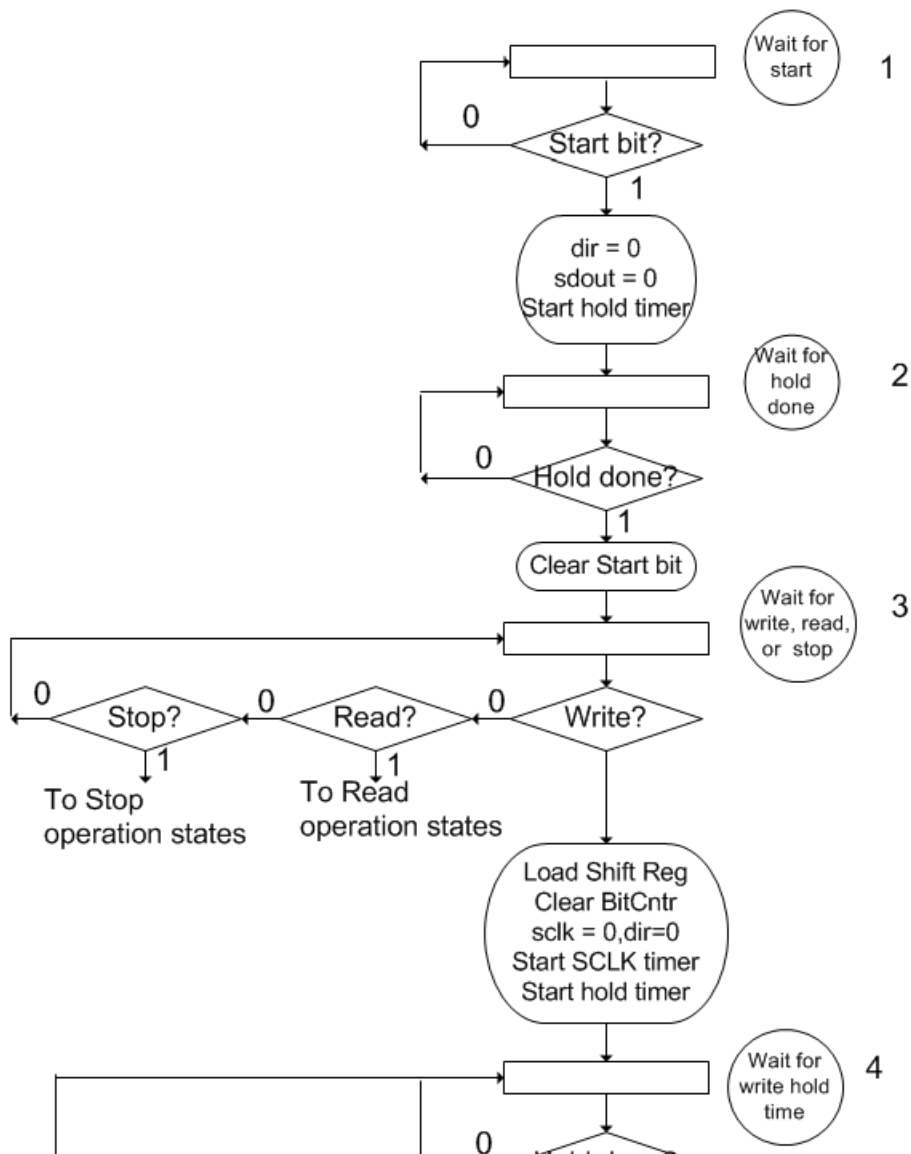
Other Stuff?

You may need other digital stuff, you decide.

ASM Chart

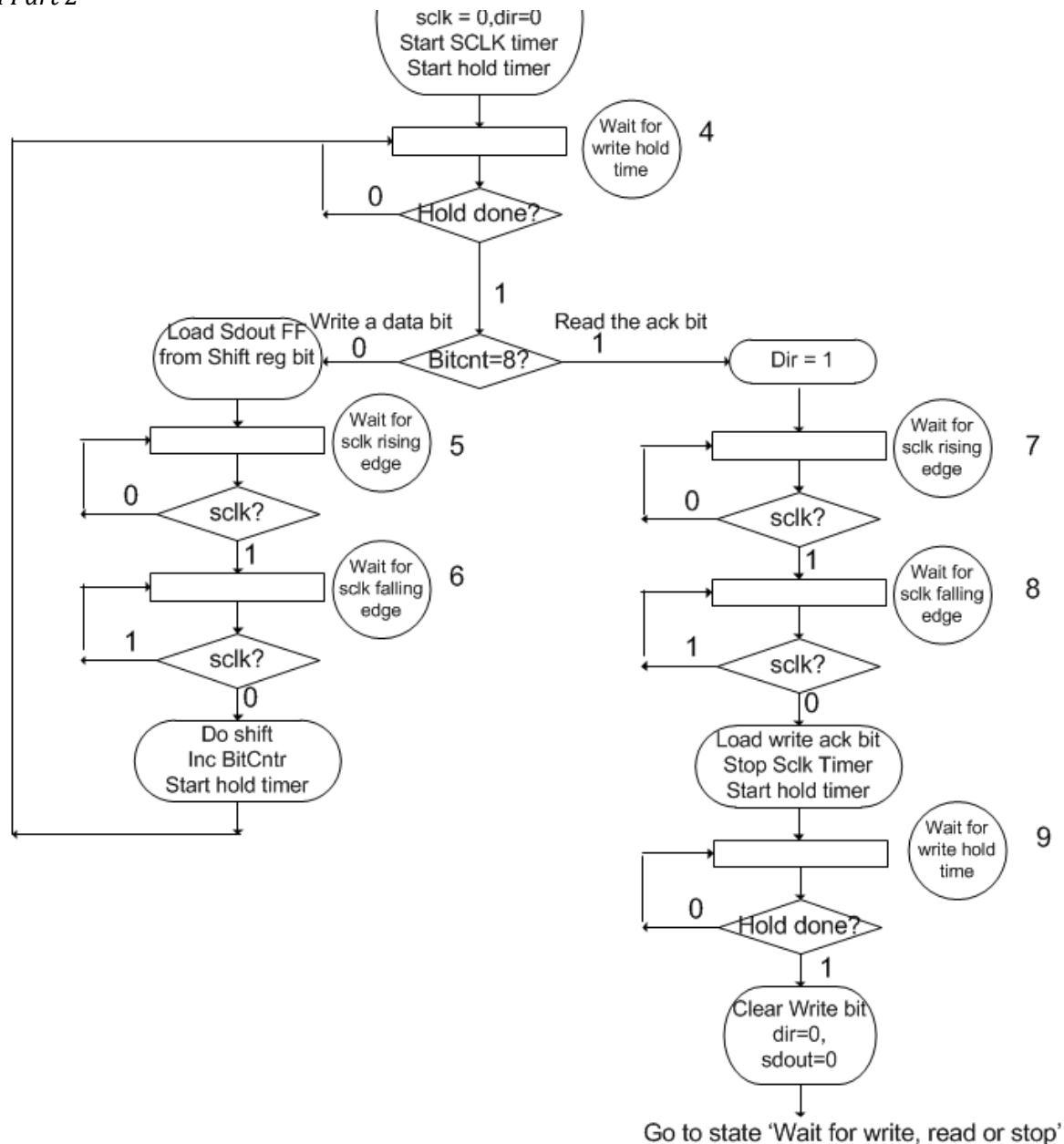
To get you started, I have given you a pseudo-ASM chart that implements the Start, Write operations (the first deliverable). This can be extended in a similar manner to implement the Stop, Read operations (the second deliverable). It is a pseudo-ASM chart in that I have not listed all of the signal conditions in the states, I have only described what goes on in each state. The individual signal assertions to accomplish those actions depend on your particular design.

ASM Part 1:



States 1, 2 implement the Start condition. State 3 waits for either the Write, Read, or Stop control bits to be set.

Asm Part 2



State 4 is the first state after the Write bit has been set, it waits for the hold time after the falling edge of SCLK. If the hold time is expired, then it either sends a data bit (States 5, 6) or reads the Ack bit (States 7, 8, 9). After the ack bit is read, the ASM goes back to State 3 to wait for Write, Read, or Stop operation.

Associated files

The ZIP archive associated with this lab contains the following files:

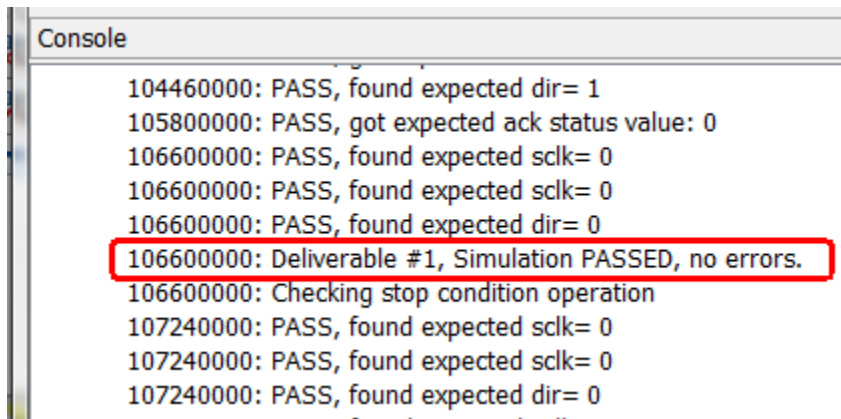
- *i2c.v* -- complete this module

- *tb_i2c.v* – test bench for the *i2c* module
- *report.doc* – report file that needs to be filled out. Fill this out for each deliverable.

Simulation

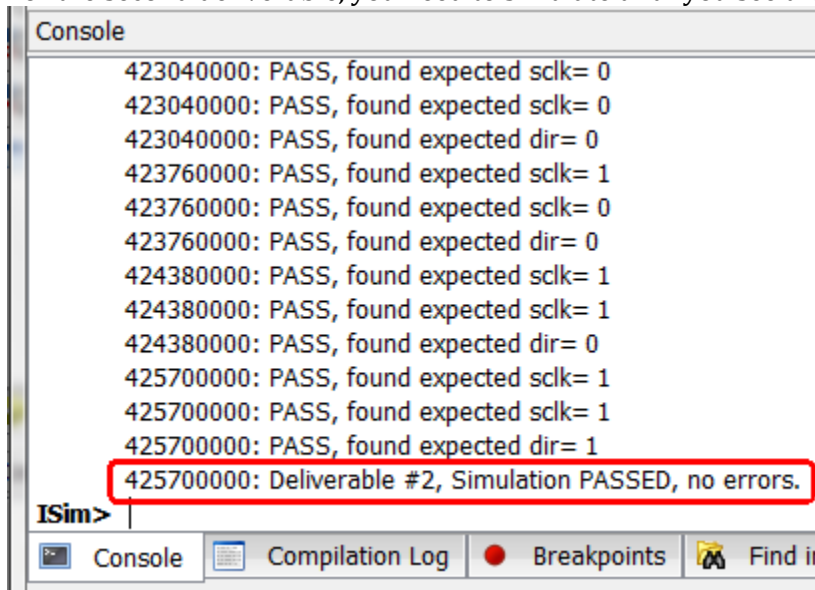
The testbench checks both deliverables. Your design must pass both behavioral and post-route simulations.

For the first deliverable, you only have to simulate until you see that the first deliverable has passed.



```
Console
104460000: PASS, found expected dir= 1
105800000: PASS, got expected ack status value: 0
106600000: PASS, found expected sclk= 0
106600000: PASS, found expected sclk= 0
106600000: PASS, found expected dir= 0
106600000: Deliverable #1, Simulation PASSED, no errors.
106600000: Checking stop condition operation
107240000: PASS, found expected sclk= 0
107240000: PASS, found expected sclk= 0
107240000: PASS, found expected dir= 0
```

For the second deliverable, you need to simulate until you see the following message:



```
Console
423040000: PASS, found expected sclk= 0
423040000: PASS, found expected sclk= 0
423040000: PASS, found expected dir= 0
423760000: PASS, found expected sclk= 1
423760000: PASS, found expected sclk= 0
423760000: PASS, found expected dir= 0
424380000: PASS, found expected sclk= 1
424380000: PASS, found expected sclk= 1
424380000: PASS, found expected dir= 0
425700000: PASS, found expected sclk= 1
425700000: PASS, found expected sclk= 1
425700000: PASS, found expected dir= 1
425700000: Deliverable #2, Simulation PASSED, no errors.
ISim>
Console Compilation Log Breakpoints Find in Files
```

Submission

Fill out the requested information in the *report.doc* file

For submission, create a directory named `lab8_netid'`, i.e., (`lab8_rbr5`) for first deliverable, and `lab9_netid'`, i.e., (`lab9_rbr5`) second deliverable.

You only need to copy your `i2c.v` and `report.doc` files to this directory.
Create a ZIP archive of this directory and submit it.