I2C Master (VHDL)

Code Download

Version 2.2: i2c_master.vhd

Corrected small SDA glitch at the end of the transaction (introduced in version 2.1)

Version 2.1: i2c_master_v2_1.vhd

Replaced gated clock with clock enable

Adjusted timing of SCL during start and stop conditions

(Thanks to Steffen Mauch for suggesting these improvements.)

Version 2.0: i2c_master_v2_0.vhd

Added ability to interface with different slaves in the same transaction

Corrected ack_error bug where ack_error went 'Z' instead of '1' on error

Corrected timing of when ack_error signal clears

Version 1.0: i2c_master_v1_0.vhd

Initial Public Release

Features

- VHDL source code of a Inter-Integrated Circuit (I2C or IIC) master component
- Meets the NXP UM10204 I2C-bus specification for single master buses
- User definable system clock
- User definable I2C serial clock frequency
- Generates Start, Stop, Repeated Start, and Acknowledge conditions
- Uses 7-bit slave addressing
- Compatible with clock-stretching by slaves
- Not recommended for multi-master buses (no arbitration or synchronization)
- Notifies user logic of slave acknowledge errors

Introduction

This details an I2C master component for single master buses, written in VHDL for use in CPLDs and FPGAs. The component reads from and writes to user logic over a parallel interface. It was designed using Quartus II, version 11.1. Resource requirements depend on the implementation. Figure 1 illustrates a typical example of the I2C master integrated into a system. A design incorporating this I2C master to create an SPI to I2C Bridge is available here.

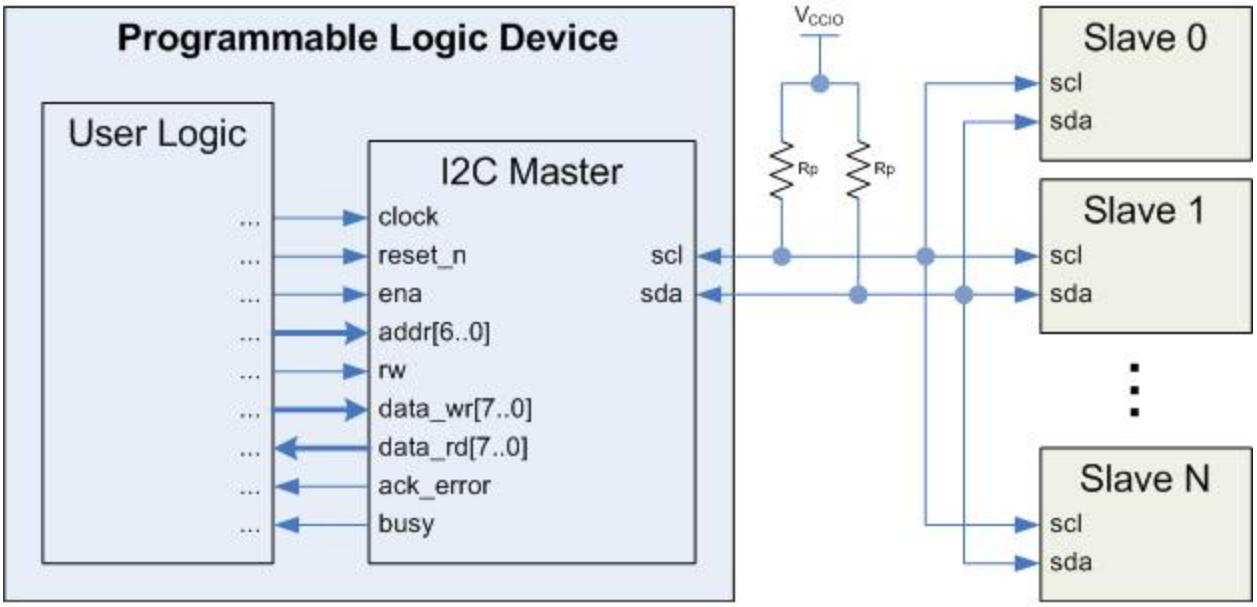

Figure 1. Example Implementation

Background

The I2C-bus is a 2-wire, half-duplex data link invented and specified by Philips (now NXP). The two lines of the I2C-bus, SDA and SCL, are bi-directional and open-drain, pulled up by resistors. SCL is a Serial Clock line, and SDA is a Serial Data line. Devices on the bus pull a line to ground to send a logical zero and release a line (leave it floating) to send a logical one.

For more information, see the I2C specification attached below in the "Additional Information" section. It explains the protocol in detail, the electrical specifications, how to size the pull-up resistors, etc.

Theory of Operation

The I2C master uses the state machine depicted in Figure 2 to implement the I2C-bus protocol. Upon start-up, the component immediately enters the *ready* state. It waits in this state until the *ena* signal latches in a command. The *start* state generates the start condition on the I2C bus, and the *command* state communicates the address and rw command to the bus. The *slv_ack1* state then captures and verifies the slave's acknowledge. Depending on the rw command, the component then proceeds to either write data to the slave (*wr* state) or receive data from the slave (*rd* state). Once complete, the master captures and verifies the slave's response (*slv_ack2* state) if writing or issues its own response (*mstr_ack* state) if reading. If the *ena* signal latches in another command, the master immediately continues with another write (*wr* state) or read (*rd* state) if the command is the same as the previous command. If different than the previous command (i.e. a read following a write or a write following a read or a new slave address), then the master issues a repeated start (*start* state) as per the I2C specification. Once the master completes a read or write and the *ena* signal does not latch in a new command, the master generates the stop condition (*stop* state) and returns to the *ready* state.

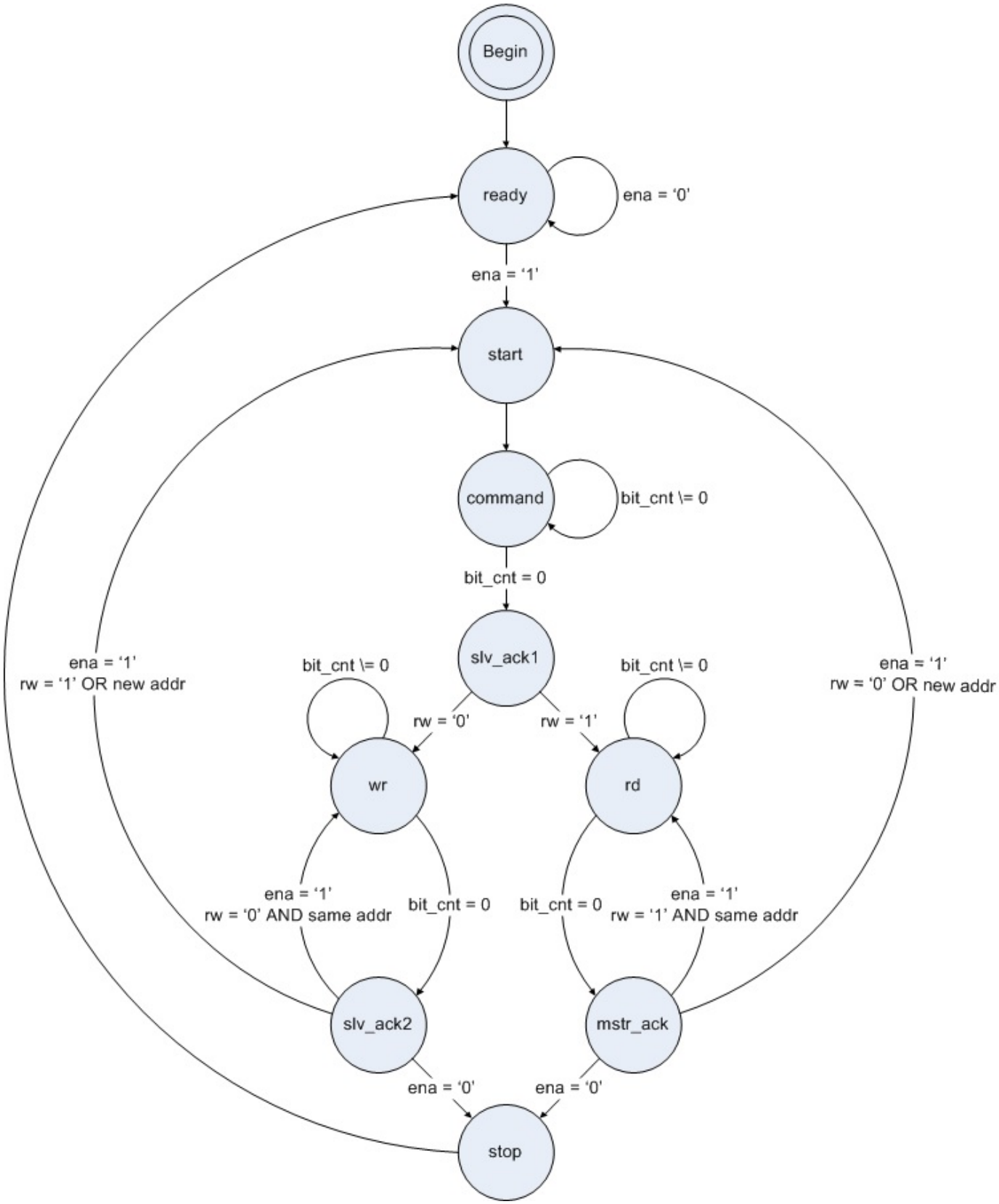

Figure 2. I2C Master State Machine

The timing for the state machine is derived from the GENERIC parameters *input_clk* and *bus_clk*, described in the "Setting the Serial Clock Speed" section below. A counter generates the data clock that runs the state machine, as well as *scl* itself.

Port Descriptions

Table 1 describes the I2C master's ports.

Port	Width	Mode	Data Type	Interface	Description
clk	1	in	standard logic	user logic	System clock.
reset_n	1	in	standard logic	user logic	Asynchronous active low reset.
ena	1	in	standard logic	user logic	0: no transaction is initiated. 1: latches in addr, rw, and data_wr to initiate a transaction. If ena is high at the conclusion of a transaction (i.e. when busy goes low) then a new address, read/write command, and data are latched in to continue the transaction.
addr	7	in	standard logic vector	user logic	Address of target slave.
rw	1	in	standard logic	user logic	0: write command. 1: read command.
data_wr	8	in	standard logic vector	user logic	Data to transmit if rw = 0 (write).
data_rd	8	out	standard logic vector	user logic	Data received if rw = 1 (read).
busy	1	out	standard logic	user logic	0: I2C master is idle and last read data is available on data_rd. 1: command has been latched in and transaction is in progress.
ack_error	1	buffer	standard logic	user logic	0: no acknowledge errors. 1: at least one acknowledge error occurred during the transaction. ack_error clears itself at the beginning of each transaction.
sda	1	inout	standard logic	slave device(s)	Serial data line of I2C bus.
scl	1	inout	standard logic	slave device(s)	Serial clock line of I2C bus.

Setting the Serial Clock Speed

The component derives the serial clock *scl* from two GENERIC parameters declared in the ENTITY, *input_clk* and *bus_clk*. The *input_clk* parameter must be set to the input system clock *clk* frequency in Hz. The default setting in the example code is 50 MHz (the frequency at which the component was simulated and tested). The *bus_clk* parameter must be set to the desired frequency of the serial clock *scl*. The default setting in the example code is 400 kHz, corresponding to the Fast-mode bit rate in the I2C specification.

Transactions

A low logic level on the *busy* output port indicates that the component is ready to accept a command. To initiate a transaction, the user logic places the desired slave address, rw command, and write data on the *addr*, *rw*, and *data_wr* ports, respectively, and asserts the *ena* signal. The command is not clocked in on the following system clock *clk*. Rather, the component is already generating the I2C timing internally, and clocks in the command based on that timing. Therefore, the user logic should wait for the *busy* signal to assert to identify when these inputs are latched into the I2C master.

The I2C master then executes the command. Once complete, it deasserts the *busy* signal to indicate that any data read is available on the *data_r* port and any errors are flagged on the *ack_error* port.

Multiple reads and writes (and any combination thereof) may be executed during a transaction. If the *ena* signal is asserted at the completion of the current command, the I2C master latches in new values of *addr*, *rw*, and *data_wr* and immediately executes the new command. The busy signal still deasserts for one *scl* cycle to indicate that any read data is available on *data_rd*, and it reasserts to indicate that the new command is latched in and being executed.

Example Transaction

Figure 3 shows the timing diagram for a typical transaction. The user logic presents the address "1010101", the rw command '0' indicating a write, and the write data "10011001". It asserts the *ena* signal to latch in these values. Once the *busy* signal asserts, the user logic issues a new command. The address remains "1010101", but the following command is a read (*rw* = '1'). *ena* remains asserted. When the I2C master finishes the first command, it deasserts the busy signal to indicate its completion. Since *ena* is asserted, the I2C master latches in the new command and reasserts the *busy* signal. Once the *busy* signal re-asserts, the user logic recognizes that the second command is being executed and deasserts the *ena* signal to end the transaction following this command. The I2C master finishes executing the read command, outputs the data read ("11001100") onto the *data_rd* port and deasserts the *busy* signal again.

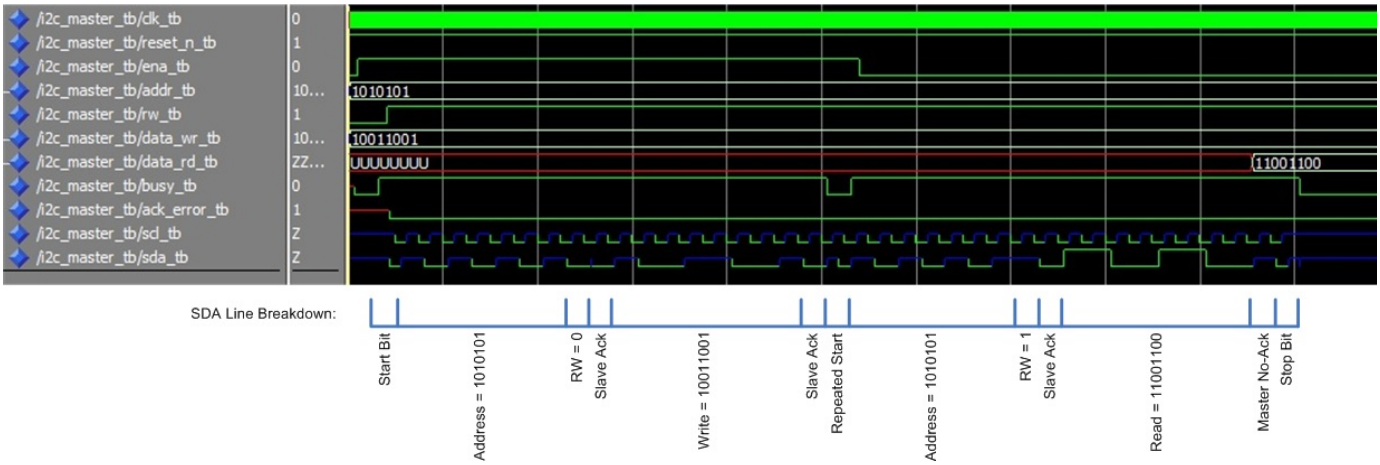

Figure 3. Typical Transaction Timing Diagram

To continue a transaction with a new command, the *ena* signal and new inputs must be present no later than the last data bit of the current command. Therefore, it is recommended to issue the new command as soon as the *busy* signal indicates that the current command is latched in. The *ena* signal can simply be left asserted until the last command of the transaction has been latched in.

Example User Logic to Control the I2C Master

A snippet of user logic code below demonstrates a simple technique for managing transactions with multiple reads and writes. This example implementation counts the *busy* signal transitions in order to issue commands to the I2C master at the proper time. The state "get_data" does everything necessary to send a write, a read, a second write, and a second read over the I2C bus in a single transaction, such as might be done to retrieve data from multiple registers in a slave device.

```
WHEN get_data =>                                --state for conducting this transaction
  busy_prev <= i2c_busy;                        --capture the value of the previous i2c busy signal
  IF(busy_prev = '0' AND i2c_busy = '1') THEN  --i2c busy just went high
    busy_cnt := busy_cnt + 1;                     --counts the times busy has gone from low to high
during transaction
  END IF;
  CASE busy_cnt IS                              --busy_cnt keeps track of which command we are on
    WHEN 0 =>                                  --no command latched in yet
      i2c_ena <= '1';                             --initiate the transaction
      i2c_addr <= slave_addr;                     --set the address of the slave
      i2c_rw <= '0';                              --command 1 is a write
      i2c_data_wr <= data_to_write;               --data to be written
    WHEN 1 =>                                  --1st busy high: command 1 latched, okay to issue
command 2
      i2c_rw <= '1';                              --command 2 is a read (addr stays the same)
    WHEN 2 =>                                  --2nd busy high: command 2 latched, okay to issue
command 3
      i2c_rw <= '0';                              --command 3 is a write
      i2c_data_wr <= new_data_to_write;           --data to be written
      IF(i2c_busy = '0') THEN                     --indicates data read in command 2 is ready
        data(15 DOWNTO 8) <= i2c_data_rd;           --retrieve data from command 2
      END IF;
    WHEN 3 =>                                  --3rd busy high: command 3 latched, okay to issue
command 4
      i2c_rw <= '1';                              --command 4 is read (addr stays the same)
    WHEN 4 =>                                  --4th busy high: command 4 latched, ready to stop
      i2c_ena <= '0';                             --deassert enable to stop transaction after command 4
      IF(i2c_busy = '0') THEN                     --indicates data read in command 4 is ready
        data(7 DOWNTO 0) <= i2c_data_rd;            --retrieve data from command 4
        busy_cnt := 0;                              --reset busy_cnt for next transaction
        state <= home;                              --transaction complete, go to next state in design
      END IF;
    WHEN OTHERS => NULL;
  END CASE;
```

Acknowledge Errors

After each transmitted byte, the receiving end of the I2C bus must issue an acknowledge or no-acknowledge signal. If the I2C master receives an incorrect response from the slave, it notifies the user logic by flagging the error on the *ack_error* port. The I2C master does not automatically attempt to resend the information, so the user logic must decide whether or not to reissue the command and/or take any additional action. The *ack_error* port is cleared once the next transaction begins.

Clock Stretching

Section 3.1.9 of the I2C specification defines an optional feature where a slave can hold *scl* low to essentially pause the transaction. Some slaves are designed to do this if, for instance, they need more time to store received data before continuing. This I2C master component is compatible with slaves that implement this feature. It requires no action by the user logic controlling the I2C master.

Reset

The *reset_n* input port must have a logic high for the I2C master component to operate. A low logic level on this port asynchronously resets the component. During reset, the component holds the *busy* port high to indicate that the I2C master is unavailable. The *scl* and *sda* ports assume a high impedance state, and the *data_rd* and *ack_error* output ports clear. Once released from reset, the *busy* port deasserts when the I2C master is ready to communicate again.

Conclusion

This I2C master is a programmable logic component that accommodates communication with I2C slaves via a straightforward parallel interface. It adheres to the NXP I2C specification in regard to single master buses and also incorporates the optional feature of clock stretching.

Additional Information

Example using the I2C master component: SPI to I2C Bridge (VHDL)

UM10204, I2C-bus specification and user manual, NXP Semiconductors N.V.

Contact

Comments, feedback, and questions can be sent to eewiki@digikey.com.