|  |
| --- |
| **I2C Master IP Block**  **User Guide** |

|  |
| --- |
|  |



Table of Contents

[Overview 3](#_Toc484000012)

[Functional Description 4](#_Toc484000013)

[Register Definitions 7](#_Toc484000014)

[Programming 8](#_Toc484000015)

# Overview

This I2C-bus master IP block was developed for integration into FPGAs. It provides an interface between a host microprocessor bus (Avalon) and the I2C-bus.

**Features**

* I2C bus speeds – Standard-node (100kbps), Fast-mode (400kbps), Fast-mode Plus (1Mbps),   
  High-Speed mode (3.4Mbps)
* Multi-master bus master (no slave capability) including collision detection and arbitration, clock synchronization, and clock stretching
* I2C 7-bit device addressing
* Up to 256-byte I2C transfer
* Avalon slave interface (8-bit) with 8-byte transmit FIFO and 8-byte receive FIFO
  + Transfer complete indication by interrupt or polling

**References**

* NXP Semiconductors, UM10204 I2C-bus specification and user manual, Rev. 6, 4 April 2014
* Lattice Semiconductor, RD1005 I2C Master Controller, April 2011

# Functional Description

The I2C Master Controller accepts commands from a microprocessor. These commands are decoded into I2C slave device read/write cycle transactions. I2C bus transactions can be configured to be 1 to 256 bytes in length. The I2C Master Controller can operate in interrupt or polling mode (i.e. the microprocessor programmer can choose to poll the I2C Master for a change in status periodically or wait to be interrupted by the I2C Master Controller when data needs to be read or written).

Avalon Master

I2C  
Master

s\_address

s\_chipselect

s\_read

s\_readdata

s\_write

s\_writedata

s\_waitrequest

s\_irq

clk

reset

SCL

SDA

I2C  
Slave

I2C  
Slave

. . .

Figure 1: I2C Master Interfaces

The IP block signals shown in **Error! Reference source not found.**are described in Table 1. They are consistent with the signals required by an Avalon Slave and standard I2C slave devices.

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal Name** | **Direction** | **Active State** | **Description** |
| ***Microprocessor Interface*** | | | |
| s\_address[2:0] | input | - | Register address bits |
| s\_chipselect | input | Active high | Device select |
| s\_read | input | Active high | Read enable |
| s\_readdata[7:0] | output | - | Read data |
| s\_write | input | Active high | Write enable |
| s\_writedata[7:0] | input | - | Write data |
| s\_waitrequest | output | Active high | Wait request to stretch the microprocessor bus cycle |
| s\_irq | output | Active high | Interrupt request |
| clk | input | - | System clock |
| reset | input | Active high | System reset |
| ***I2C Interface*** | | | |
| SCL | inout | - | I2C clock signal (open-drain) |
| SDA | inout | - | I2C data signal (open drain) |

Table 1: I2C Master Interface Signal Definitions

All signals with the prefix *s\_* are synchronous to *clk*. Therefore, *s\_* inputs must meet setup and hold requirements referenced to *clk*, and *s\_* outputs are delayed from *clk*. Avalon bus cycles can be as short as one *clk* cycle. The I2C master inserts wait states by asserting *s\_waitrequest* on the same cycle that *s\_read* or *s\_write* is asserted by the Avalon master. See the write and read Avalon bus cycle timing in Figure 2.

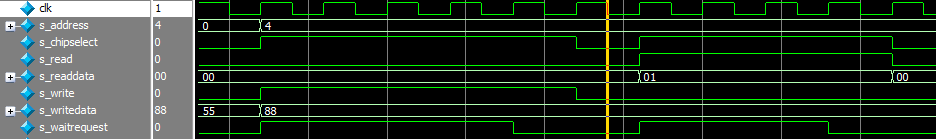


Figure 2: Avalon Interface write cycle followed by read cycle

The SCL and SDA signals are open drain. Their state is either Hi-Z or ‘0’. When the I2C function is integrated into a system, pull-up resistors provide the drive for the high state. In simulation, this is accomplished by defining the wire in the testbench as a “tri1”.

tri1 SCL; // SCL is pulled high when SCL is Hi-Z  
 tri1 SDA; // SDA is pulled high when SDA is Hi-Z

Figure 3 shows an I2C transaction with the pull-up low-to-high transition and high state as a dotted line.

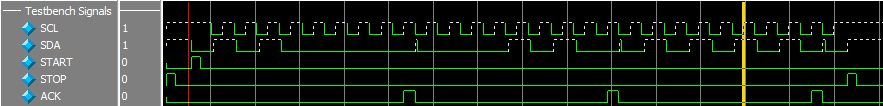


Figure 3: I2C Interface (and testbench) signals

When instantiating the I2C\_Master in a design, the system clock frequency parameter, CLK\_RATE, is set by default to 125MHz. This value is used The clock frequency unit is MHz. The value should be an integer value rounded down to an integer value (e.g. 66.67MHz 🡪 66MHz). The default value of 125MHz can be overridden as shown below.

parameter CLK\_RATE = 66; // System clock frequency in MHz

I2C\_Master #(.CLK\_RATE(CLK\_RATE)) i2c(

.SDA(SDA),

.SCL(SCL),

.clk(clk),

.reset(reset),

// Avalon Interface

.s\_address(s\_address),

.s\_chipselect(s\_chipselect),

.s\_read(s\_read),

.s\_readdata(s\_readdata),

.s\_write(s\_write),

.s\_writedata(s\_writedata),

.s\_waitrequest(s\_waitrequest),

.s\_irq(s\_irq)

);

The I2C master is comprised of a top level structural module (I2C\_Master) and three functional modules (AvalonIF, MainSM, and I2C\_BusIF).

The AvalonIF module is an Avalon Slave interface between an Avalon Master and the I2C master controller. It can be adapted to interface with other microprocessor busses. It has three main functions.

* Provides the microprocessor with access to I2C master data, command, and status information via addressable registers
* Manages data transfers between the I2C master (MainSM) and transmit/receive FIFOs in AvalonIF
* Manages interrupts to the microprocessor

The MainSM controls all I2C Bus Master functions.

Start/Stop Detection – detects start and stop events on the I2C bus including those initiated by this I2C master. Detection of start and stop events is necessary to determine whether or not the I2C bus is in use by another master on the bus when the primary master gets a GO signal from the microprocessor. Furthermore, the primary I2C Master Controller must detect its own START condition before it proceeds with the transaction.

SCL Bit Clock – The I2C Master Controller can be programmed to generate an I2C clock that will run in four modes: Standard-mode (100kbps), Fast-mode (400kbps), Fast-mode Plus (1Mbps), or High-speed mode (3.4Mbps). The clock rate depends on the value of the mode bits in the command register. Due to the nature of the I2C bus, the actual SCL clock that is seen by all devices on the bus may not be running at the same frequency that the master requests or generates. This is because the I2C SCL clock low time can be extended by a slave or another master. Thus, the SCL clock frequency can be lower than the nominal frequency of the select mode. For more information on SCL clocking see the I2C-bus Specification and User Manual, UN10204, sections 3.1.7 “Clock Synchronization” and 3.1.9 “Clock Stretching”.

Bus Arbitration – Arbitration only applies in a multi-master I2C system configuration. If two masters attempt to access the bus at the same time, the collision is detected when a master determines that SDA is being driven low after it has released the SDA line (i.e. SDA does not match its driven state) during the SCL high time. In this case, the Main State Machine will assert Lost\_Arb and Error.

Main State Machine – The main state machine orchestrates I2C bus transactions. It initiates a transaction when the GO bit in the command register is set there are no Errors that need to be cleared. Both read and write transactions are managed by the main state machine. See the I2C-bus Specification and User Manual, UN10204, section 3. “The I2C-bus protocol” for more information on transaction details.

# Register Definitions

The register address space supports eight 8-bit registers. Only four registers are currently defined

|  |  |  |  |
| --- | --- | --- | --- |
| **Address** | **Name** | **Dir** | **Description** |
| 000 | DATA | R/W | Read/Write data from/to 8-byte deep receive and transmit FIFOs. Read – Data from the I2C slave following the slave address (R/W# = 1) Write – Data to the I2C slave following the slave address (R/W# = 0) |
| 001 | SLAVE ADDRESS | R/W | I2C slave device address byte (7-bit addressing, write only). This is the first byte sent by the I2C master after START. [7:1] Slave address  [0] I2C R/W# bit (0-write, 1-read) |
| 010 | Reserved | - |  |
| 011 | Reserved | - |  |
| 100 | STATUS | R | Status (read only) [7] - I2C\_Bus\_Busy 1 = bus busy, 0 = bus free [6] - Error 1 = error, 0 = no error (cleared by Abort) [5] - Abort\_Ack 1 = Abort completed by I2C master [4] - Lost\_Arb 1 = Lost arbitration indicator (cleared by Abort) [3] - Done 1 = I2C transaction complete, 0 = not done [2] - xFull 1 = xmit FIFO full, 0 = not full [1] - xEmpty 1 = xmit FIFO empty, 0 = not empty [0] - rEmpty 1 = rcv FIFO empty, 0 = not empty  \*\* The Done interrupt is cleared by reading the Status register |
| 100 | COMMAND | W | Command (write only) [7] - GO 1 = start I2C transaction, self clearing [6] - Abort 1 = stop I2C transaction in progress [5] - noStop 1 = no STOP after last byte of the transaction  0 = normal STOP after last byte of the transaction [4:3] - I2C mode  00 = Standard-mode (100kbps) 01 = Fast-mode (400kbps) 10 = Fast-mode Plus (1Mbps) 11 = High-speed mode (3.4Mbps)  [2:1] - reserved [0] - DoneIE, 1 = Transaction complete interrupt enabled |
| 101 | BYTE COUNT | R/W | The byte count is the number of bytes for the master to send or receive following the slave address. |
| 110 | Reserved | - |  |
| 111 | Reserved | - |  |

Table 2: I2C Master Registers

# Programming

The programming sequences to initiate an I2C bus transaction is as follows.

Write Transaction

1. Write the slave address and R/W# = 0 in the Address register 0x1.
2. Write the byte count in register 0x5. Note that the transmit FIFO is 8-bytes deep. If the transaction is more than 8-bytes, the software must manage the loading of the transmit FIFOs to avoid overflow and underflow.
3. Write the data bytes to the transmit FIFO (0x0) sequentially.
4. Send the GO command along with the other control bits set appropriately to the command register 0x4
5. Read the status register 0x4 to determine the completion state (polled or interrupt).

Read Transaction

1. Write the slave address and R/W# = 1 in the Address register 0x1.
2. Write the byte count in register 0x5. Note that the receive FIFO is 8-bytes deep. If the transaction is more than 8-bytes, the software must manage the unloading of the receive FIFO to avoid overflow and underflow.
3. Send the GO command along with the other control bits set appropriately to the command register 0x4
4. Read the status register 0x4 to determine the completion state (polled or interrupt).
5. Read the data bytes from the receive FIFO (0x0) sequentially.

Examples of some I2C device accesses are shown here. In case of reading a byte from an EEPROM, the I2C transaction involves both write and read transfers. First, the address is written. Then the data is read. This combo transaction uses the I2C repeat START feature.

Write a byte to an EEPROM (slave address = 0x50)

Read 0x04 // check ready for I2C transaction  
// if not ready, wait or fix it  
Write 0xA0 to register 0x1 // Addr = 0x50 (7 bits) and Write  
Write 0x02 to register 0x5 // 2 bytes  
Write 0x70 to register 0x0 // EEPROM address = 0x70  
Write 0x55 to register 0x0 // EEPROM data = 0x55  
Write 0x88 to register 0x4 // Send GO command, Fast-mode

Read a byte to an EEPROM (slave address = 0x50)

Read 0x04 // check ready for I2C transaction  
// if not ready, wait or fix it  
Write 0xA0 to register 0x1 // Addr = 0x50 (7 bits) and Write  
Write 0x01 to register 0x5 // 1 byte  
Write 0x70 to register 0x0 // EEPROM address = 0x70  
Write 0xA8 to register 0x4 // Send GO command, noStop=1,  
 // Fast-mode  
// wait until done setting the EEPROM address  
Write 0xA1 to register 0x1 // Addr = 0x50 (7 bits) and Read  
//Write 0x01 to register 0x5 // 1 byte already set  
Write 0x88 to register 0x4 // Send GO command, noStop=0,  
 // Fast-mode