Implementation of an I^{2}C Writing Master

Marco Imbelli Cai

Dipartimento di Ingegneria dell'Informazione

Università di Pisa

Master degree in Computer Engineering, course of Electronics and Communication Systems

Table of contents

The I^{2}C protocol

An overview

The I^{2}C (Inter Integrated Circuit) protocol is a synchronous, multi-master/multi-slave, packet-switched, single-ended, serial communication bus. The bus was developed by Philips in 1982, in its first version, and it is today (after improvements) widely used for attaching lower-speed peripherals ICs to processors and microcontrollers in short-distance, intra-board communications.

The bus is basically composed of two main wires: SDA and SCL. The former is the data line, while the latter is the clock line. To address different slaves that could be connected to such shared bus, the protocol uses a 7-bit addressing scheme (that allows to address up to 128 different slaves) to transmit 8-bit packets.

Anatomy of an I^{2}C transaction

As previously mentioned, the communication happens through two lines: a clock line, SCL (Serial CLock), and a data line, SDA (Serial DAta), and it works as here described:

1. The master receives a slave address and a data byte from the suer logic: the first identifies the slave to which the second byte is going to be serially delivered. Whenever the logic will signal to do so, the transmission will start.
2. The master starts the transmission by sending a START bit, which is identified by a high-to-low transition of the SDA line with the SCL line high.
3. The master starts transmitting serially the t bits of the slave's address. The address is sent with the most significant bit first.
4. The master sends a final bit representing the type of operation that it is about to be executed: 0 to signal a WRITE operation (data will be *sent* to the slave) or 1 to signal a READS one (data will be *received* from the slave).
5. The slave answers to the received byte (the address and the operation bit form an 8-bit packet) with an ACK (active low) or a NACK (active high) on the SDA line. The former symbolizes that the slave exists on the bus and that it is and is ready to receive the data byte or to transmit one to the master, while the former can be generated for multiple reasons. A more detailed description of ACKs/NACKs usage and generation reasons is provided in the next paragraph.
6. The master, after receiving an ACK, continues either in transmit or receive mode, by transmitting the data byte to the slave or by receiving a byte from the slave. In either case, after the transmission/reception, an ACK/NACK stage happens to symbolize the

success/failure of the operation. The data byte is sent/received by starting from its significant bit.

7. The master can:
    a. Stop the communication by generating a STOP condition, which is identified by a low-to-high transition of the SDA line with the SCL line high. If this is the case, the master stops transmitting/receiving after the ACK/NACK reception, waits for new commands and eventually goes back to point 1.
    b. Continue to interface with the chosen slave in what is called a *Repeated Start* procedure, in which after the ACK/NACK reception stage, another START bit is sent on the SDA line with the same structure as in point 2. If this is the case, the master retains the control of the bus and after receiving commands from the user logic, it eventually goes back to point 1.

Except from the generation of the START/STOP conditions, all the other transitions of the SDA line take place with the SCL one low.

Figure 1.1 shows an example of a full I^{2}C transmission, in which the master writes a byte containing 0x09 to the slave at the address 0x66:

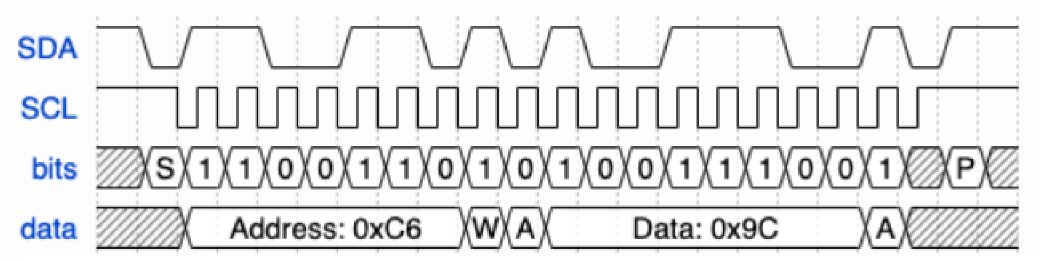


Figure 1.1: writing 0x09 to the slave at 0x06

In the above figure, the START and STOP bits are indicated respectively by "S" and "P", while the ACKs are indicated by "A".

Acknowledgment (ACK) and Not ACKnowledgement bits

The acknowledgement phase takes place after the transmission/reception of every byte, and allows the receiver to signal the transmitter that the byte was successfully received and another one may be sent afterwards. Since ACKs/NACKs are generated after a byte transmission, they take place during the ninth SCL cycle (which is used as a clock in the I2C protocol) if we start counting cycles from the first bit transmission one.

The ACK signal is delivered by pulling low the SDA line during this ninth SCL cycle.

When the SDA line remains instead high during the ninth SCL cycle, a NACK condition is being experienced. If this is the case, the controller can either generate a STOP condition to abort the transmission or a repeated START condition to start a new one, for example to re-transmit the last byte. The protocol defines five conditions that lead to the generation of a NACK signal:

1. No receiver is present on the bus with the transmitted address, so no device can answer with an ACK bit.
2. The receiver cannot receive/transmit at the moment, as it is busy with some other real-time function.
3. The receiver gets data that it doesn't understand.
4. The receiver cannot get any more data bytes.
5. The receiver wants to stop receiving bytes for some other reason.

Protocol applications

As previously mentioned, the I^2C protocol is a synchronous, multi-master/multi-slave, packet-switched, single-ended, serial communication bus. It is used mainly to communicate with devices in which simplicity and low costs are primary factors, compared to transmission speed.

Some common applications of the protocol include:

- Access to Flash Memories and EEPROMS
- Access to ADCs and DACs with low speed
- Computer monitors settings update
- Volume tweaking in speakers
- Display control in cellular phones
- Reading of sensor values
- Activation and deactivation of the power supply devices of electronic systems

Moreover, the protocol is directly implementable using two generic I/O pins and a rather simple software/firmware (bit-banging technique).

Figure 1.2 shows an example interconnection of I^2C bus applications.

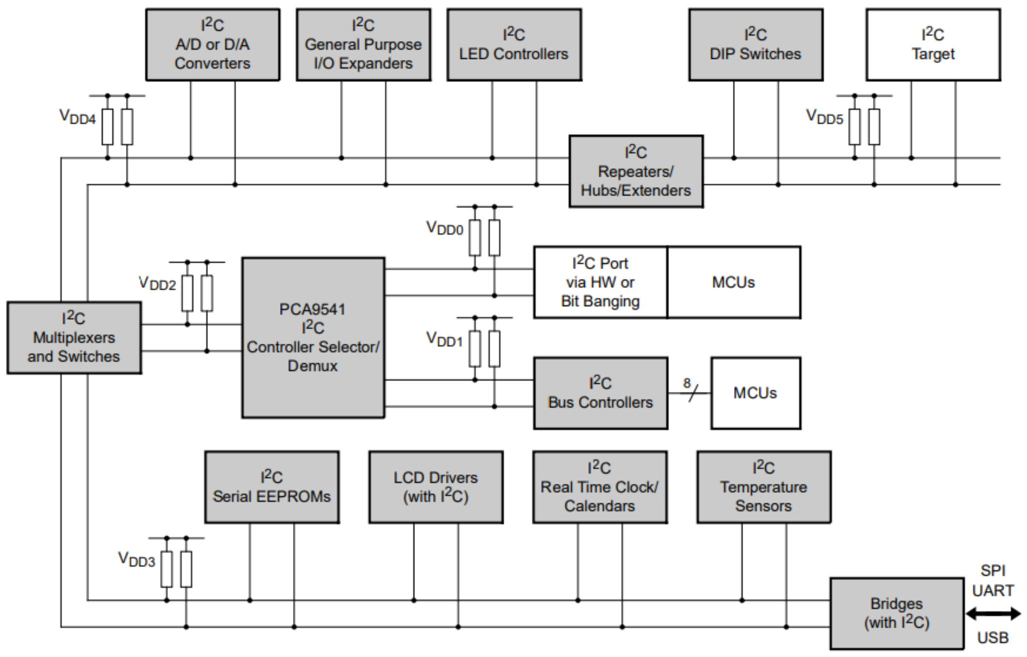

I²C A/D or D/A Converters
I²C General Purpose I/O Expanders
I²C LED Controllers
I²C DIP Switches
I²C Target
VDD4
VDD5
I²C Repeaters/ Hubs/Extenders
VDD0
VDD2
I²C Port via HW or Bit Banging
MCUs
I²C Multiplexers and Switches
PCA9541 I²C Controller Selector/ Demux
VDD1
I²C Bus Controllers
8
MCUs
I²C Serial EEPROMs
LCD Drivers (with I²C)
I²C Real Time Clock/ Calendars
I²C Temperature Sensors
VDD3
Bridges (with I²C)
SPI
UART
USB

Figure 2: I²C applications

Advantages and limitations

The I²C bus has many advantages:

- It is cheaper to implement than comparable low-power buses, uses fewer pins than most of the other serial communication protocols and employs a simpler physical layer.
- It supports up to 127 different devices.
- It can reach transfer rate peaks of 400Kbps, which is fast enough for most of human interfaces as well as fan control, temperature sensing and other low-speed control logic.
- It has been (historically) less onerous for manufacturers to include such protocol in their design rather than spend time and resources on developing their own.

These advantages are, anyways, balanced by some limitations of this protocol, that makes it not suitable for some classes of devices:

- On low-power systems, the pull-up resistors used in the design can consume more power than the entire rest of the design combined, and often limit the speed of the bus itself. This is the reason for which some designers are switching to other serial bus technologies (e.g., I³C or SPI, which do not need pull-up resistors at all).
- The seven bits addressing scheme can be a limitation when more than 128 devices need to be connected to the same I²C bus. An additional 3-bit extension to the addresses is available (10-bit addressing scheme), but is rarely adopted and many operating systems don't even support it.

- Devices using the I^2C protocol are allowed to stretch the SCL line speed to meet their particular needs. This can use up bandwidth that may be needed by faster devices and result in an increase in latency.
- Because I^2C is a *shared* bus, there is the possibility for a connected device to experience a failure and compromise the usage of the entire bus to the other connected devices, until a reset condition is asserted.

Because of these limitations (address management, potential faults, speed), very few I2CX buses arrive to have even a dozen devices connected. Manufacturers usually try to mitigate such limits by employing multiple I^2C bus segments, in which devices with different speed ranges are grouped.

Design of an I^{2}C Writing Master

Adopted software

The design of the device has been carried out by using the VHDL general-purpose programming language on the Visual Studio Code IDE.

The testing campaign has been performed by employing the ModelSim – Intel FPGA Starter Edition 2020.1 software.

The implementation phase has been carried out by using the Xilinx Vivado software on version 2020.2.

Device structure

The provided interface of the device to be implemented is shown in *Figure 2.1*, while *Figure 2.2* shows the actual interface that was chosen for the implementation of such device:

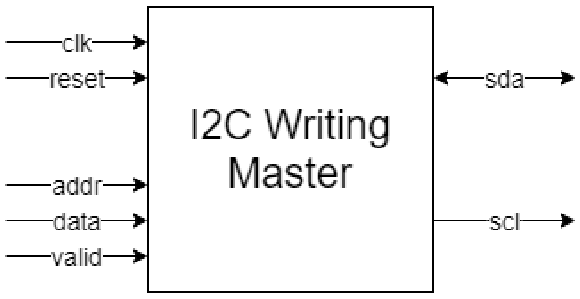


Figure 2.1: provided interface

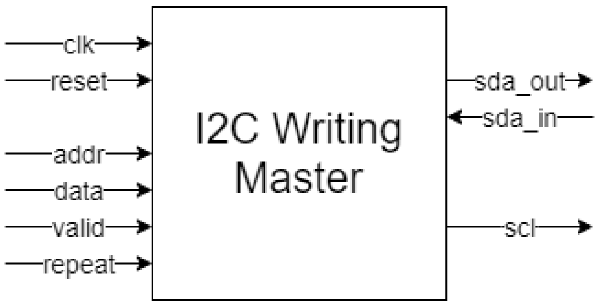


Figure 2.2: used interface

Ports' characteristics and usages are reported in *Table 2.1*:

Port	Mode	Width	Data type	Description
clk	in	1	std_logic	The system's clock
reset	in	1	std_logic	Asynchronous active high reset
addr	in	7	std_logic_vector	Address of the target slave
data	in	8	std_logic_vector	Data to be transmitted
valid	in	1	std_logic	Enables the writing process, once both addr and data have been provided by the user logic: • 0: no transaction is initiated • 1: start of the transaction
repeat	in	1	std_logic	Enables the repeated start process after a data ACK has been received:

				• 0: no repeated start is performed • 1: repeated start procedure is performed
sda_in	in	1	std_logic	Serial DAta input line of the I^2C bus, from the slave at address addr to the device
sda_out	out	1	std_logic	Serial DAta output line of the I^2C bus, from the device to the slave at address addr
scl	out	1	std_logic	Serial CLock line of the I^2C bus

Table 2.1: ports' characteristics and their usage

The additional port named "*repeat*" was not present in the original provided and proposed interface. Anyway, this port is necessary to give the user logic the possibility to decide whether to continua after a transaction with the so called *Repeated Start* procedure or not.

Moreover, the bi-directional port "*sda*" has been split into two uni-directional ports ("*sda_out*" and "*sda_in*") in order to facilitate the implementation and usage of such serial data line.

Design assumptions

The design of this I2C Writing Master unit has been caried out by considering some assumptions, that are reported below:

- The following implementation only takes care of the *writing* part of the protocol.
- The SCL line is 32 times slower than the system's clock
- The SDA line is implemented via two separate ports (sda_out, OUT port, and sda_in, IN port)
- The contacted slave is going to behave correctly and as expected by the master device

Finite state machine

Figure 2.3 shows the finite state machine that describes the behavior of the designed component. The edges that don't specify any transition condition represent transitions that are triggered only by means of clocks rising/falling edges' detection.

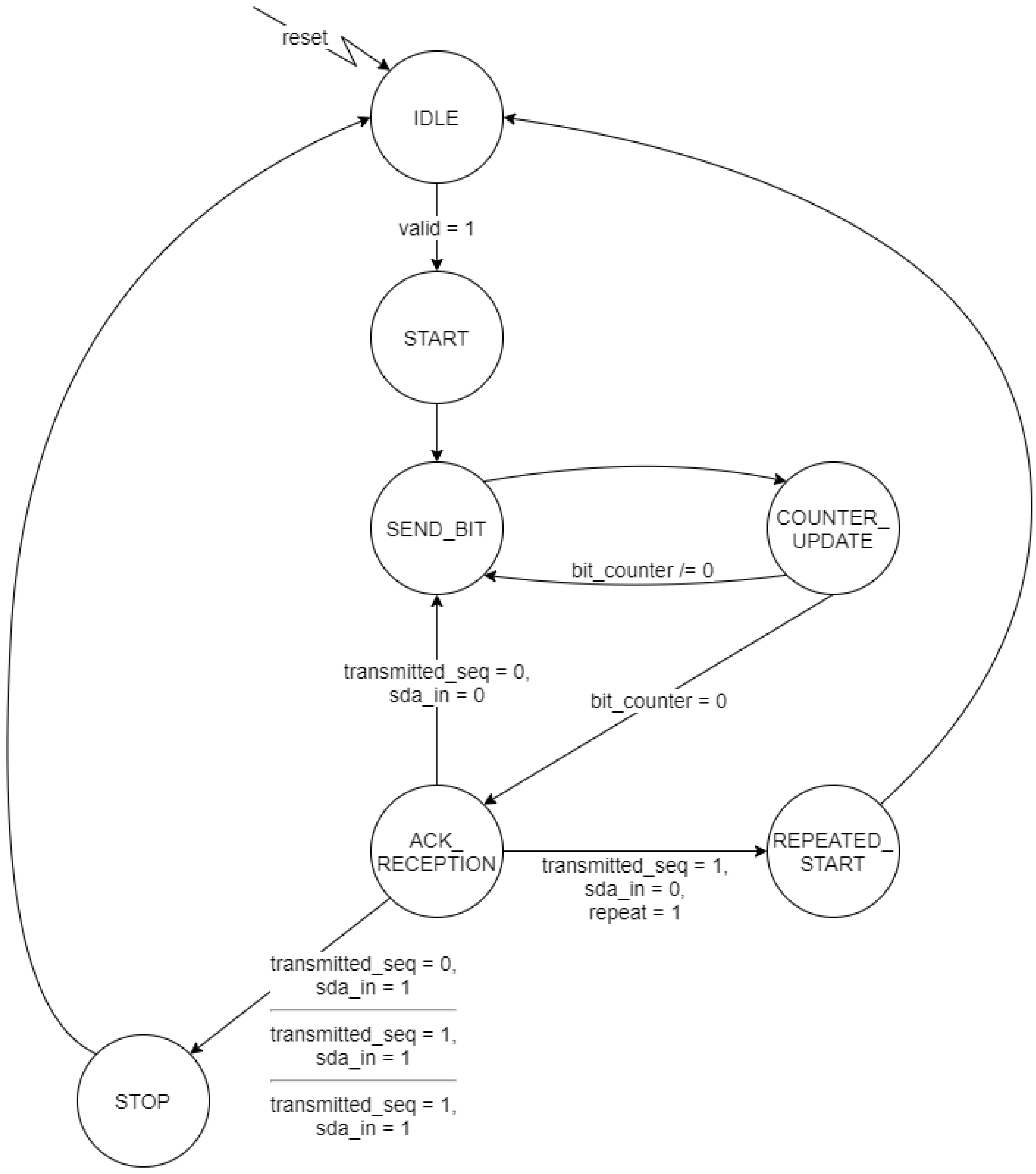

reset
IDLE
valid = 1
START
SEND_BIT
COUNTER_ UPDATE
bit_counter /= 0
bit_counter = 0
transmitted_seq = 0, sda_in = 0
ACK_ RECEPTION
transmitted_seq = 1, sda_in = 0, repeat = 1
REPEATED_ START
transmitted_seq = 0, sda_in = 1
transmitted_seq = 1, sda_in = 1
transmitted_seq = 1, sda_in = 1
STOP

Figure 2.3: FSM of the designed I^2C writing master

In *Table 2.2* a more detailed description of the states and the transitions among them is provided.

State	Description	Transition to	If
IDLE	The device waits for the address and data. If valid is asserted, transmission can start	START	Valid is asserted
START	The START BIT is sent to signal the start of the transmission	SEND_BIT	SCL gets enabled
SEND_BIT	The device sends the address or data bit depending on the counter	COUNTER_UPDATE	Falling edge of SCL
COUNTER_UPDATE	The counter to keep track of which bit needs to be sent is updated until it reaches zero	SEND_BIT	Falling edge of SCL, still bits to transmit
		ACK_RECEPTION	Falling edge of SCL, bits have been transmitted
ACK_RECEPTION	The device reads the ACK or NACK signal from the connected slave. Depending on whether it transmitted an address or a data byte, and on repeat, different states can be reached	SEND_BIT	Address was transmitted, ACK was received
		REPEATED_START	Data was transmitted, ACK was received, repeat is asserted
		STOP	• Address was transmitted, NACK was received • Data was transmitted, NACK was received • Data was transmitted, ACK was received, repeat is not asserted
REPEATED_START	The repeated start condition is provided on SDA by the device	IDLE	Falling edge of SCL
STOP	The STOP BIT is sent by the device to signal the end of the transmission	IDLE	Counter that generates the STOP condition elapses

Each of the above states	-	IDLE	Reset is asserted

More explanations about the device’s behavior can be found in the VHDL code file.

Testing the component

Testbench design

The employed testbench interface is showed in *Figure 3.1*. As previously mentioned, the testing campaign has been performed by employing the ModelSim – Intel FPGA Starter Edition 2020.1 software.

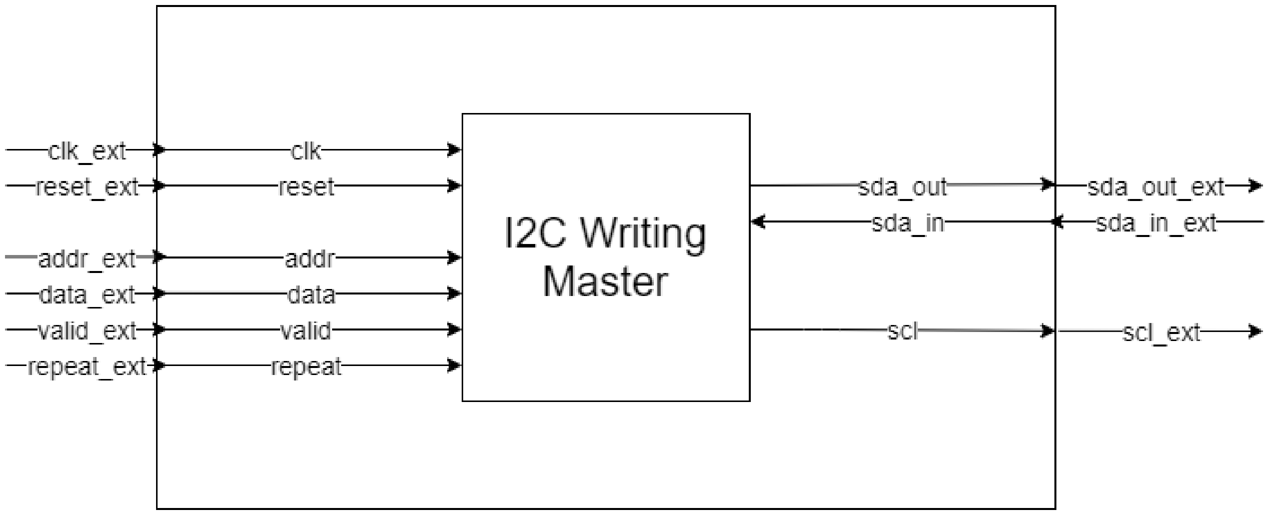


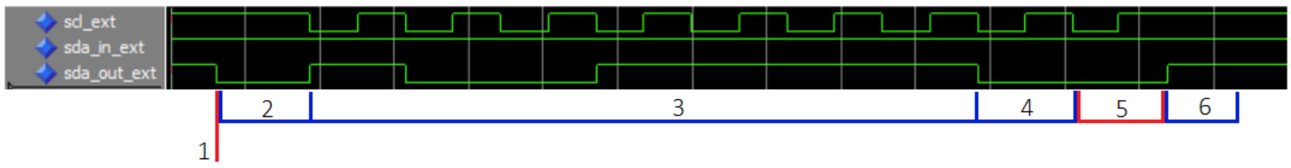
Figure 3.1: testbench interface

By compiling the VHDL file, the compilation results successful without any warning message.

Testing

In this section, four different use cases of functioning of the designed I^{2}C writing master are descripted.

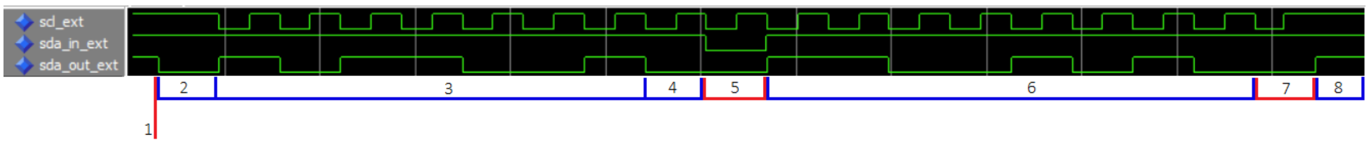
Use case 1: wrong address, NACK provided



1. Valid is asserted: the transmission can start
2. The high-to-low transition of sda_out while scl is high indicates the START condition
3. The address is sent on sda_out by the device: it can be seen that in this transmission the address that is being sent is the 7-bit sequence ‘1001111’
4. The command bit is sent on sda_out
5. The device checks the status of sda_in to find it high: this means that a NACK has been raised. The reasons for a NACK generation can be found in section 1.

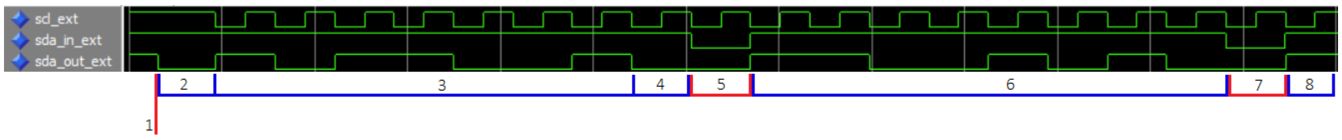
6. The low-to-high transition of sda_out while scl is high indicates the STOP condition.

Use case 2: address ACK provided, data NACK provided, no repeated start



1. Valid is asserted: the transmission can start
2. The high to low transition of sda_out while SCL is high indicates the START condition
3. The address is sent on sda_out by the device: it can be seen that in this transmission the address that is being sent is the 7-bit sequence ‘1011001’
4. The command bit is sent on sda_out
5. The device checks the status of sda_in to find it low: this means that an ACK has been generated by the contacted slave for the transmitted byte. This signals the master that the data byte can be sent.
6. The data byte is sent on sda_out by the device: it can be seen that in this transmission the byte that is being transmitted is the 7-bit sequence ‘11001010’.
7. The device checks the status of sda_in to find it high: this means that a NACK has been raised by the slave for the received byte. The reasons for a NACK generation can be found in section 1.
8. The low-to-high transition of sda_out while scl is high indicates the STOP condition.

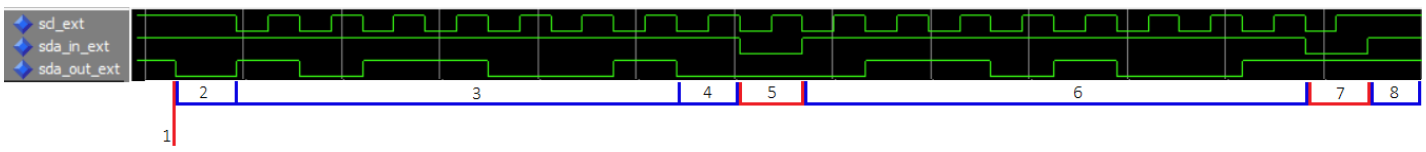
Use case 3: address ACK provided, data ACK provided, repeated start



1. Valid is asserted: the transmission can start
2. The high to low transition of sda_out while SCL is high indicates the START condition
3. The address is sent on sda_out by the device: it can be seen that in this transmission the address that is being sent is the 7-bit sequence ‘1011001’
4. The command bit is sent on sda_out
5. The device checks the status of sda_in to find it low: this means that an ACK has been generated by the contacted slave for the transmitted byte. This signals the master that the data byte can be sent.
6. The data byte is sent on sda_out by the device: it can be seen that in this transmission the byte that is being transmitted is the 7-bit sequence ‘11001010’.
7. The device checks the status of sda_in to find it low: this means that an ACK has been generated by the contacted slave for the transmitted byte.

8. The master pulls sda_out from low to high to signal the slave the start of a repeated start procedure. After this, the master will remain idle until the valid signal is asserted again for a new transmission. No STOP condition is generated.

Use case 4: address ACK provided, data ACK provided, no repeated start



1. Valid is asserted: the transmission can start
2. The high to low transition of sda_out while SCL is high indicates the START condition
3. The address is sent on sda_out by the device: it can be seen that in this transmission the address that is being sent is the 7-bit sequence ‘1011001’
4. The command bit is sent on sda_out
5. The device checks the status of sda_in to find it low: this means that an ACK has been generated by the contacted slave for the transmitted byte. This signals the master that the data byte can be sent.
6. The data byte is sent on sda_out by the device: it can be seen that in this transmission the byte that is being transmitted is the 7-bit sequence ‘01101001’.
7. The device checks the status of sda_in to find it low: this means that an ACK has been generated by the contacted slave for the transmitted byte.
8. The low-to-high transition of sda_out while scl is high indicates the STOP condition.

Of course, it has to be reminded that the functioning of the device is going to be correct if the contacted slave will be correctly driven.

Synthesis and Implementation in Xilinx Vivado

In this section, the followed steps for synthesizing and implementing the designed component on an FPGA (Field Programmable Gate Array) device are explained, and the obtained results will be reported.

Details

The FPGA device that has been employed in this phase is embedded on a ZyBO (Zynq BOard), and is identified by the code ‘xc7z010clg400-1’.

Note that the workflow is carried out by employing an “*out of context*” synthesis mode and the “Vivado synthesis default reports” strategy.

To make the Vivado tool evaluate the timing constraints correctly, a barrier of registers has been placed at every input/output port of the component (except from the ‘clock’ and ‘reset’ ones).

RTL elaboration

After visualizing the RTL elaborated design, the component is shown without any warning or error messages. Reporting an image of the schematic at this stage would be meaningless because of the out-of-context synthesis mode.

Synthesis

After running the synthesis and opening the synthesized design, some warning messages are reported. The Vivado tool reports that some signals are getting read inside a process but they weren’t placed in the sensitivity list of such process. This won’t anyway cause any problem, since the expected behavior of the device is exactly the one shown by the simulations.

Moreover, Vivado signals the risk of inferring latches on three of the employed signals. This as well is not a problem, since if-then-else flows and conditions cover all the possible states that the device could be in.

Table 4.1 shows the chosen clock frequency and the resulting Worst Negative Slack (WNS).

Clock frequency	**Nanoseconds equivalent**	**Worst Negative Slack (WNS)**
125 Mhz	8 ns	5.649 ns

Table 4.1: timing results

Based on the data, the maximum clock frequency (theoretically) achievable by the component can be computed in the following way, knowing that $125\ MHz = 8\ ns$:

$$\frac{1}{8-WNS} \cdot 1000 = \frac{1}{2.351} \cdot 1000 \approx 425\ MHz\ .$$

The critical path reported by the Vivado tool is shown in *Figure 4.1*, while a rough power usage estimation data is reported in *Figure 4.2*.

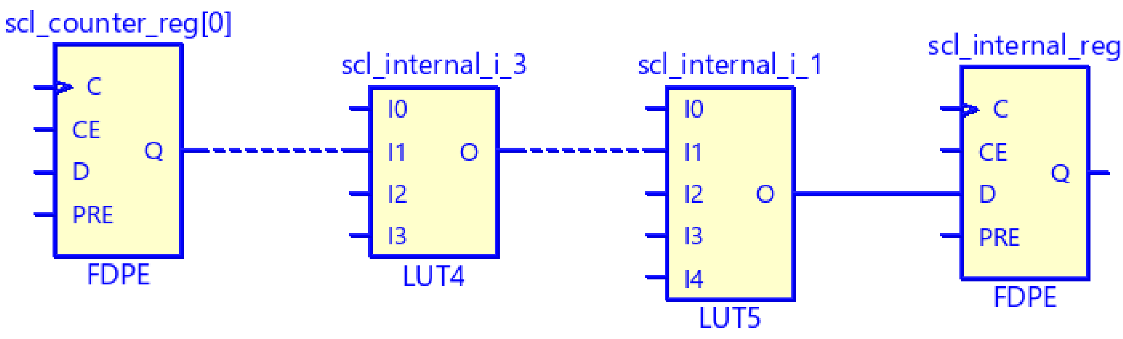
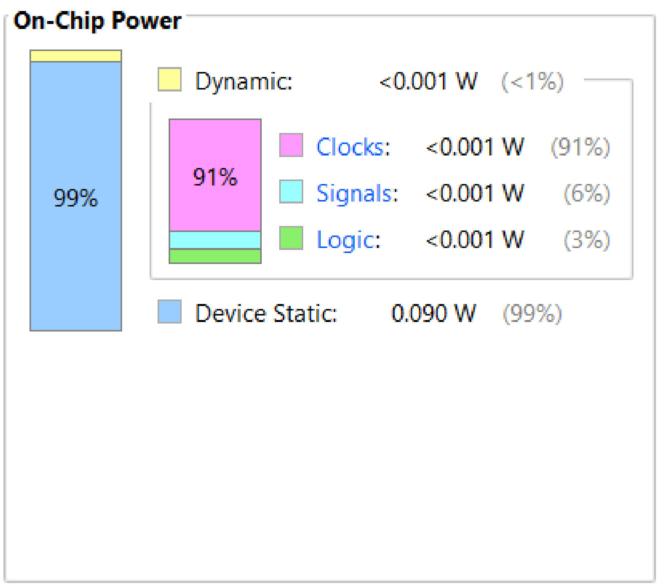


Figure 4.1: critical path



On-Chip Power

99%

🟨 Dynamic: <0.001 W (<1%)

91%

🟪	Clocks:	<0.001 W	(91%)
🟦	Signals:	<0.001 W	(6%)
🟩	Logic:	<0.001 W	(3%)

🟦 Device Static: 0.090 W (99%)

Figure 4.2: rough power estimation

Conclusions

The I^2C (Inter Integrated Circuit) protocol is a synchronous, multi-master/multi-slave, packet-switched, single-ended, serial communication bus. The bus was developed by Philips in 1982, in its first version, and it is today (after improvements) widely used for attaching lower-speed peripherals ICs to processors and microcontrollers in short-distance, intra-board communications.

The performed analysis has shown that even if I2C master devices are usually driven with a frequency that can arrive up to 1MHz (and 400kHz are usually sufficient), the design component's speed could be pushed beyond 125MHz and theoretically arrive to work with frequencies up to ≈ 425 Mhz.

However, this is never the case since slave devices that employ the I^2C bus can almost never support those kinds of clock frequencies.