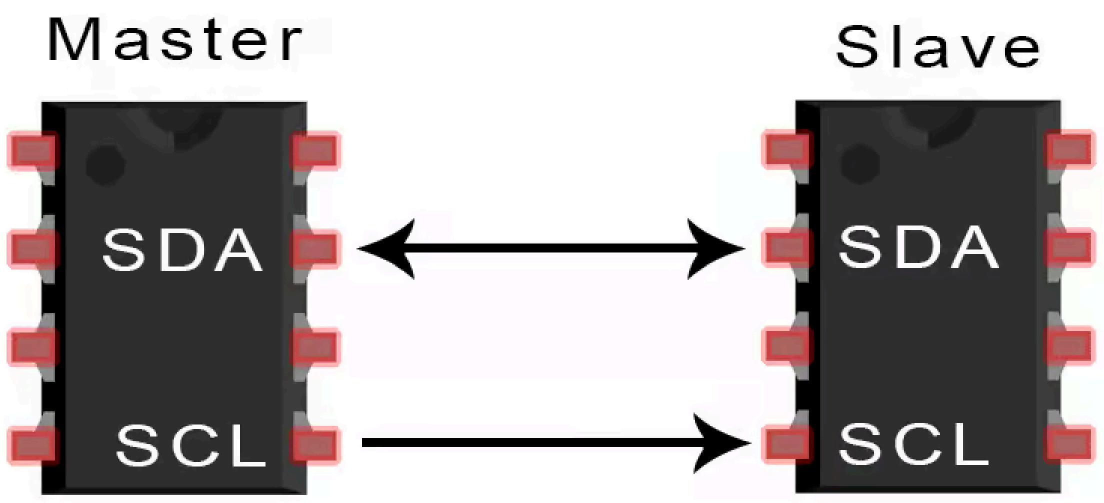
I2C implementation in VHDL

Dimitrios Rigos
Leonidas Georgiou

Brief introduction

In this project, I have implemented the I²C protocol for the slave using VHDL.
I²C (Inter-Integrated Circuit) is a widely used synchronous, multi-master, multi-slave, packet-switched, single-ended, serial communication protocol.
It is commonly used for communication between microcontrollers and peripherals such as sensors, EEPROMs, and displays.
The goal of this project was to design and simulate a basic I²C interface entirely in VHDL, focusing on understanding the protocol's timing and control mechanisms.
This kind of implementation is useful when working on FPGA-based systems where built-in I²C controllers might not be available or need to be customized.
In the following slides, I will present the architecture, the state machine design, and the simulation results of the I²C controller



But how I2C works?

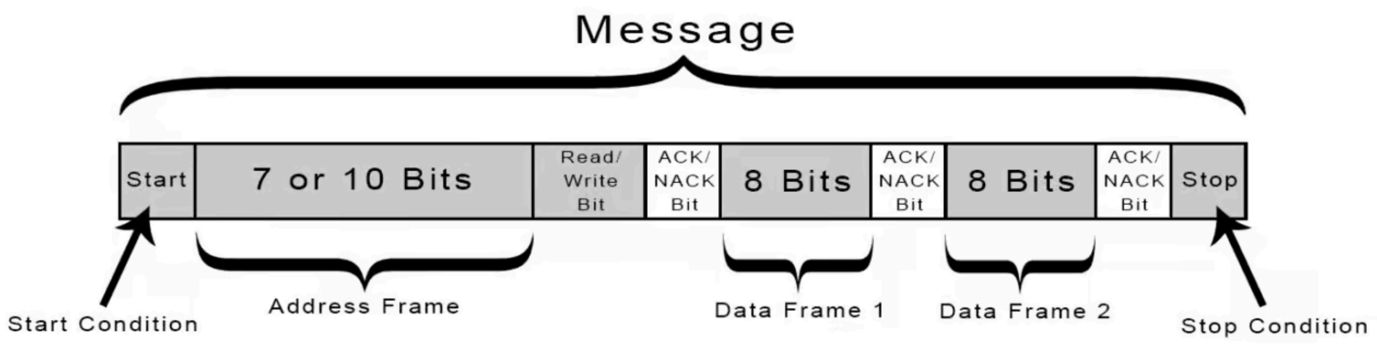
The I²C protocol operates using two bidirectional lines: **SDA (Serial Data)** and **SCL (Serial Clock)**. Data is transmitted synchronously with the clock provided by the master device. Each communication session begins with a **START condition**, followed by a 7-bit address and a **read/write** bit.

The addressed slave sends an **ACK** (acknowledge) bit to confirm reception. Data is then transferred in **8-bit bytes**, each followed by an ACK from the receiver. The session ends with a **STOP condition**.

The key features of the protocol include:

- Only two lines are required regardless of the number of devices.
- Each slave has a unique address.
- Multiple masters can exist, but only one can control the bus at a time (bus arbitration).
- The protocol supports clock stretching and acknowledges, allowing for robust communication.

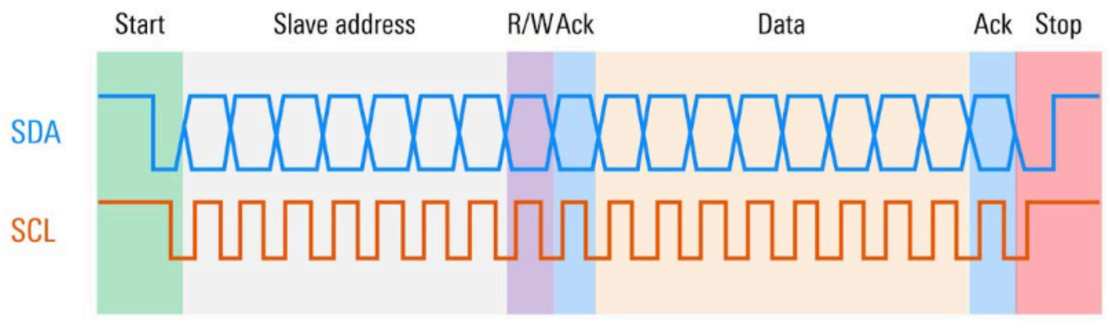
This diagram clearly illustrates the flow of messages and the structure of data packets in the I²C communication protocol, making it easier to understand how data is transmitted between the master and slave devices:



As we can see, timing is crucial in I²C communication, and all signals must strictly comply with the protocol's setup and hold time requirements. Therefore, careful consideration of timing constraints is essential when implementing I²C in a VHDL environment.

Each bit of the I²C message is transmitted on every clock cycle generated by the master device, which is the sole controller of the clock line (SCL). However, the slave device has its own internal clock, which must run at a higher frequency than the master's clock. This is necessary to properly sample and process the incoming data. Managing the transition between these different clock domains falls under the category of **clock domain crossing**, a topic that we will discuss later in this presentation.

To better understand what happens and when, I am providing this diagram showing the SCL pulses from the master and how each part of the message is transmitted:



For reference, the slave address being transmitted consists of 7 bits, so there must be 7 corresponding SCL clock cycles.

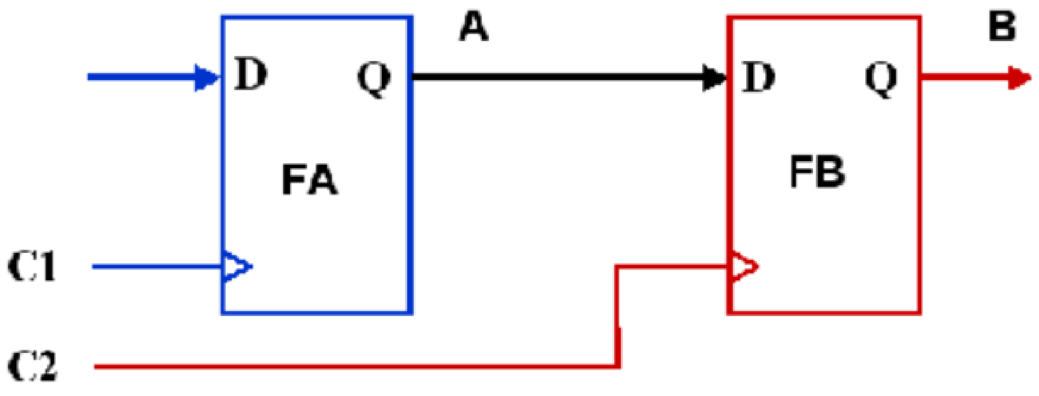
My Implementation Approach

In my implementation, I adhered to the specifications outlined in the TCA6424A I/O expander datasheet, which operates using the I²C communication protocol. To ensure structured and efficient handling of communication, I divided the program into seven distinct states: Idle, Address Frame, Read/Write Check, ACK, NACK, Command Byte, Read, and Write. Each state plays a specific role in managing the I²C transaction process. Below, I will examine the purpose and verification performed in each state to achieve the overall functionality of the system.

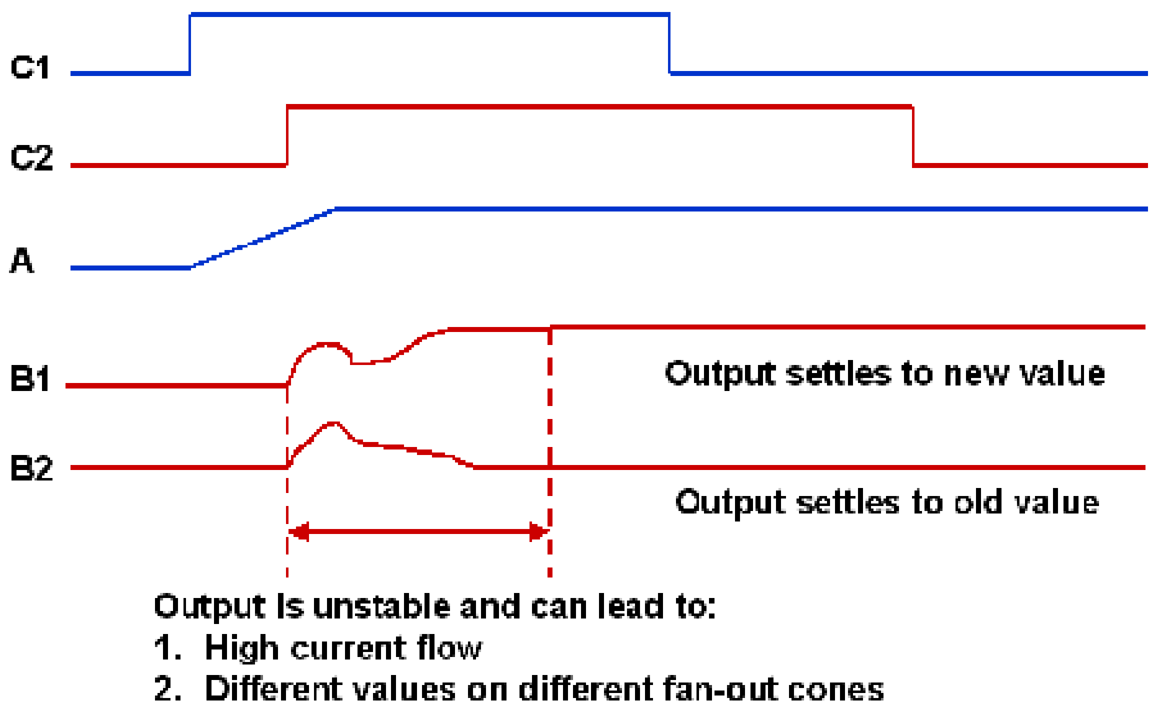
Crossing Clock Domains

Before examining the states and the code implementation, it is essential to address the challenges related to metastability and timing errors.
Such problems can arise when signals cross from one clock domain (Master) to another (Slave). Although clock domain crossings can occur in various scenarios, in this specific case, we will focus on the simplest: a transition from a slower clock domain to a faster one, with the latter operating at approximately 16 times the frequency of the former.

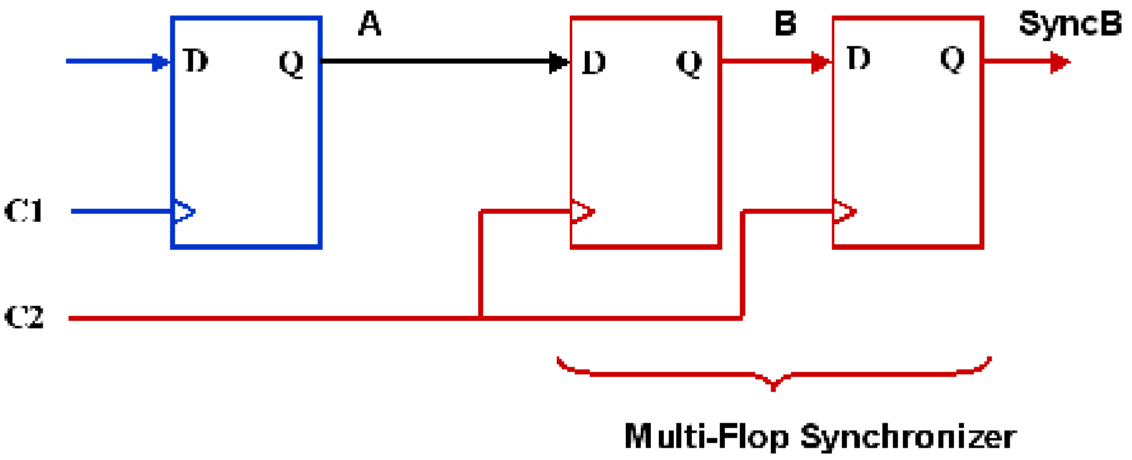
A clock domain crossing happens whenever data is transferred from a flip-flop driven by one clock to a flip-flop driven by a different clock, potentially causing metastability issues if not handled properly.



If the transition on signal **A** occurs very close to the active edge of clock **C2**, it may cause setup or hold time violations at the destination flip-flop, labeled **FB**. As a consequence, the output signal **B** can oscillate unpredictably for an indefinite duration. This results in an unstable output that may or may not stabilize before the next clock edge of **C2** arrives. This behavior is known as **metastability**, and the flip-flop **FB** is considered to have entered a metastable state. The figure below illustrates this phenomenon in detail.


C1
C2
A
B1
B2
Output settles to new value
Output settles to old value
Output Is unstable and can lead to:
1. High current flow
2. Different values on different fan-out cones

To mitigate this issue, metastability problems can be effectively avoided by incorporating special hardware structures known as **synchronizers** in the destination clock domain. Synchronizers provide sufficient time for any metastable oscillations to settle, ensuring that a stable and reliable output is produced before the data is further processed. A commonly employed solution is the **multi-flop synchronizer**, which consists of a series of flip-flops connected in sequence, as illustrated below.



My code implementation is as follows:

```vhdl
signal r1_SDA : std_logic := '0';
signal r2_SDA : std_logic := '0';
signal r3_SDA : std_logic := '0';

signal r1_SCL : std_logic := '0';
signal r2_SCL : std_logic := '0';
signal r3_SCL : std_logic := '0';
```

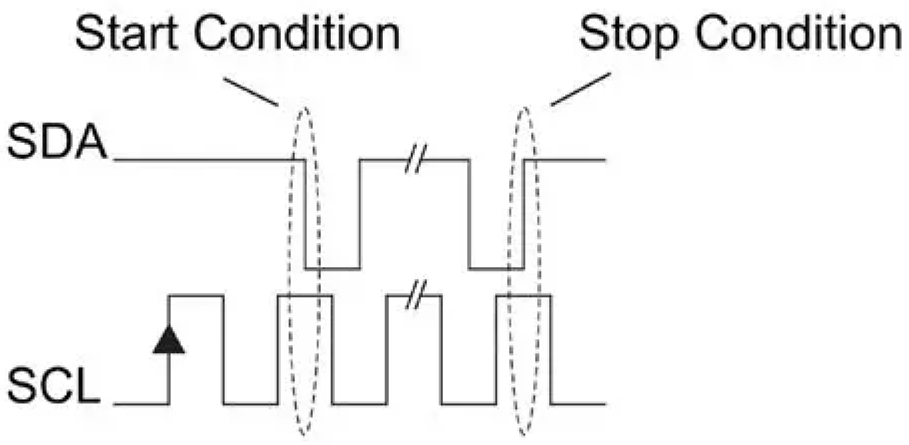
```vhdl
begin
    if rising_edge(Sys_Clock) then

        r1_SDA <= SDA;
        r2_SDA <= r1_SDA;
        r3_SDA <= r2_SDA;


        r1_SCL <= SCL;
        r2_SCL <= r1_SCL;
        r3_SCL <= r2_SCL;
```

As observed in the code, I employed three different flip-flops to minimize errors and improve signal stability during clock domain crossing.
Having addressed metastability and timing concerns, we can now proceed to examine the individual states of the implementation.

STATES

1. IDLE

In this state, as the name suggests, the slave waits for a **START condition**, which signals that the communication is about to begin. The START condition occurs when the **SDA line transitions from a high voltage level to a low voltage level while the SCL line remains high**. Once the START condition is detected, the slave moves to the next state, **Address Frame**, where it checks if the received address matches its own. We can see this more clearly here.:



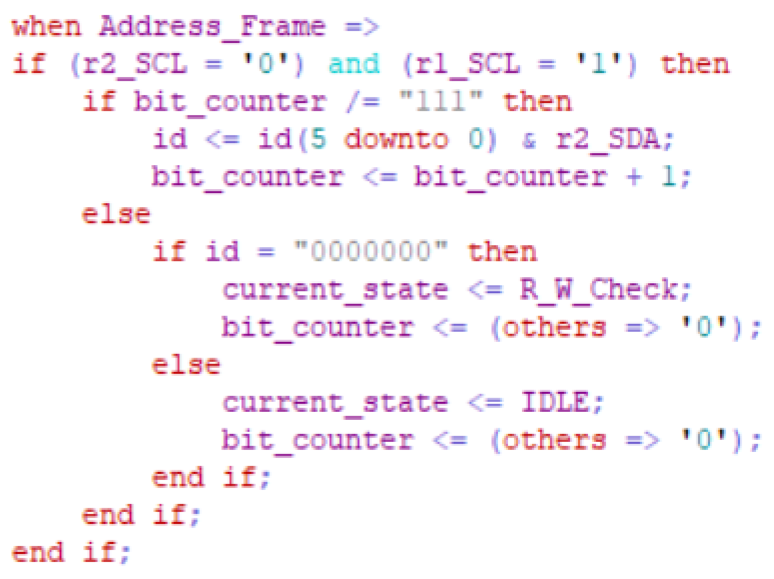
And this is the VHDL implementation:

```
case current_state is

    when IDLE =>
        if (r3_SDA = '1') and (r2_SDA = '0') and (r2_SCL = '1') and (r1_SCL = '1') then
            current_state <= Address_Frame;
            bit_counter <= (others => '0');
        end if;
```

Basically, as previously mentioned, we are interested in detecting a falling edge on the **SDA** line while the **SCL** line is high. A falling edge means that the current state of **SDA** is high, and in the following clock pulse, we expect it to transition to low (0).

2. ADDRESS FRAME

In this state, the system captures one bit from the master on each **SCL** pulse. After receiving a total of 7 bits, the address is complete, and the system checks which slave the master is attempting to communicate with. Since this implementation involves only one slave device, I assigned a fixed ID to it and perform a simple comparison to verify if the received address matches the expected value.

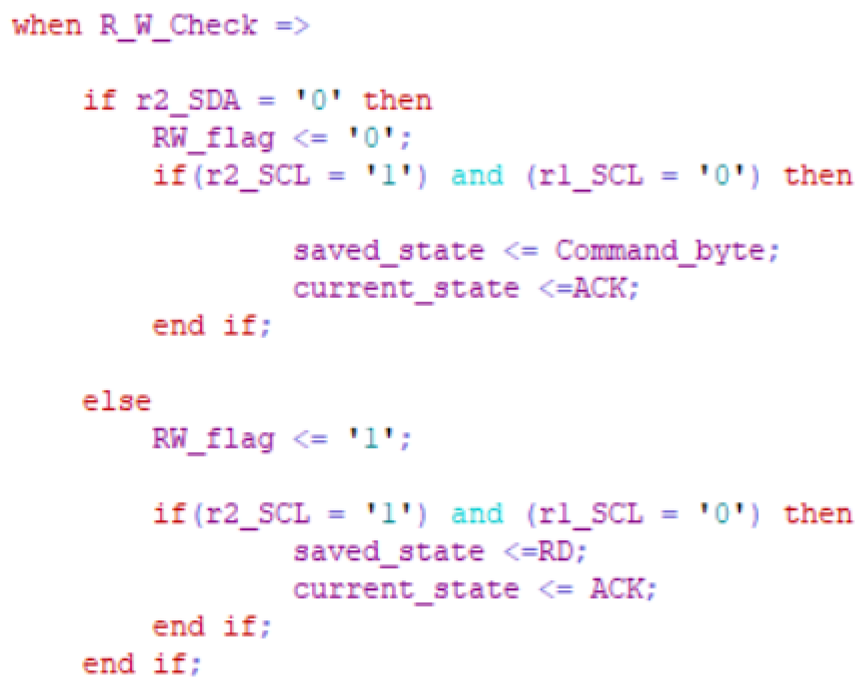


```
when Address_Frame =>
if (r2_SCL = '0') and (r1_SCL = '1') then
    if bit_counter /= "111" then
        id <= id(5 downto 0) & r2_SDA;
        bit_counter <= bit_counter + 1;
    else
        if id = "0000000" then
            current_state <= R_W_Check;
            bit_counter <= (others => '0');
        else
            current_state <= IDLE;
            bit_counter <= (others => '0');
        end if;
    end if;
end if;
```

3. R/W CHECK

In this state, we examine the 8th bit transmitted by the master, which indicates whether the operation is a write or a read from a specific register of the slave device. A value of `0` signifies a write operation, while a value of `1` indicates a read operation.

Based on this bit, a R/W flag is set accordingly for later use in the protocol flow. Once the operation type is determined, the slave responds with the first ACK signal, confirming receipt of the address and operation type.



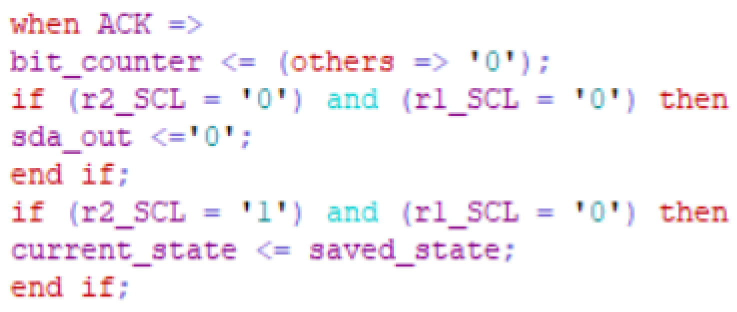
```
when R_W_Check =>

    if r2_SDA = '0' then
        RW_flag <= '0';
        if(r2_SCL = '1') and (r1_SCL = '0') then

                saved_state <= Command_byte;
                current_state <=ACK;
        end if;

    else
        RW_flag <= '1';

        if(r2_SCL = '1') and (r1_SCL = '0') then
                saved_state <=RD;
                current_state <= ACK;
        end if;
    end if;
```

4. **ACK/NACK**

In the ACK state, the master releases the SDA line (placing it in a high-impedance state), allowing the slave to drive the line.
If the addressed slave acknowledges the transmission, it pulls the SDA line low (logic 0) during the next clock pulse. This indicates successful reception of the address and read/write bit.

To ensure correct timing, the slave must place the `0` on the SDA line while the SCL clock is low, and the state transition occurs after the rising edge of SCL, when the master samples the line.

Since the ACK state can be reached from multiple preceding states, it is necessary to implement a saved_state signal. This signal stores the identity of the state from which the ACK was triggered, allowing the state machine to correctly resume execution from the appropriate point after the acknowledgment phase is complete.

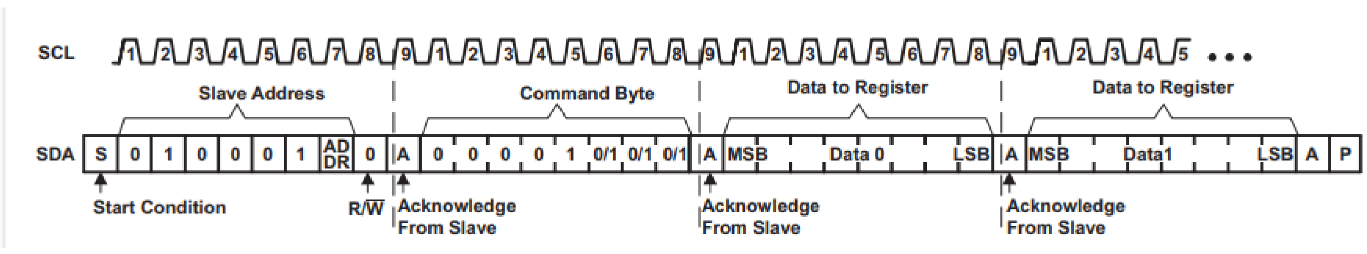


```
when ACK =>
bit_counter <= (others => '0');
if (r2_SCL = '0') and (r1_SCL = '0') then
sda_out <='0';
end if;
if (r2_SCL = '1') and (r1_SCL = '0') then
current_state <= saved_state;
end if;
```

Before proceeding to the **Command Byte**, **Read**, and **Write** states, it is essential to understand how the **TCA6424A I/O expander** handles data transfer operations, as the behavior differs between **write** and **read** transactions.

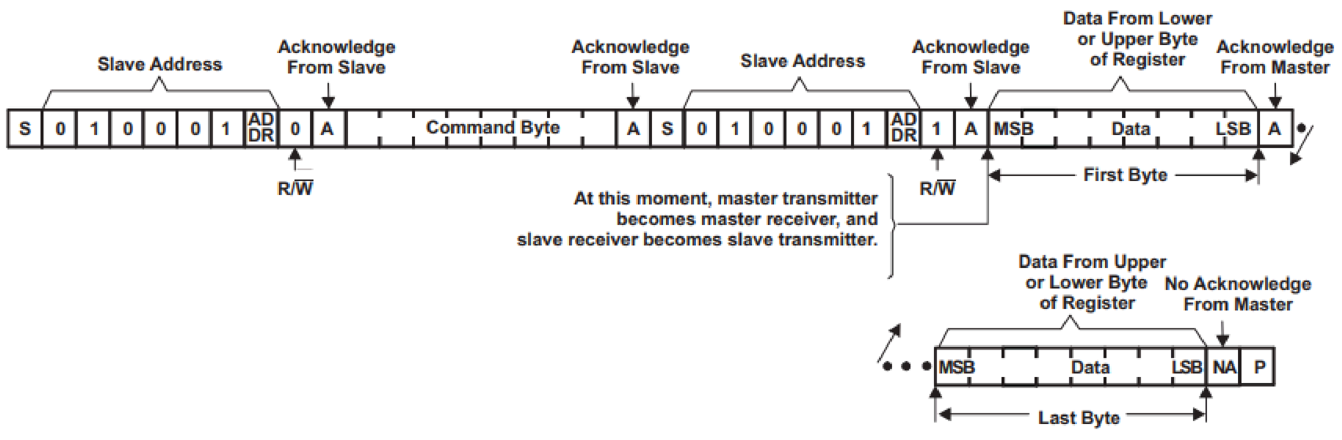
Write Operation

In a write sequence, data is transmitted to the TCA6424A by first sending the device address with the **least significant bit (LSB)** set to logic **0**. This indicates a write operation.
Immediately following the address, the **command byte** is sent. The command byte specifies **which internal register** of the device is to receive the data that follows.



Read Operation

A read operation involves a slightly more complex sequence. It begins similarly, with the master sending the device address with the **LSB = 0**, followed by the **command byte** that selects the desired register to read from.
However, to actually read the data, the master then issues a **repeated start condition** and re-sends the device address — this time with the **LSB set to 1**, indicating a read operation. The TCA6424A then transmits the data stored in the previously selected register.It is important to note that after the repeated start, the internal register address remains as defined by the earlier command byte, ensuring the correct data is retrieved.


Slave Address
Acknowledge From Slave
Acknowledge From Slave
Slave Address
Acknowledge From Slave
Data From Lower or Upper Byte of Register
Acknowledge From Master
S 0 1 0 0 0 1 ADDR 0 A
Command Byte
A S 0 1 0 0 0 1 ADDR 1 A MSB Data LSB A
R/W
At this moment, master transmitter becomes master receiver, and slave receiver becomes slave transmitter.
R/W
First Byte
Data From Upper or Lower Byte of Register
No Acknowledge From Master
MSB Data LSB NA P
Last Byte

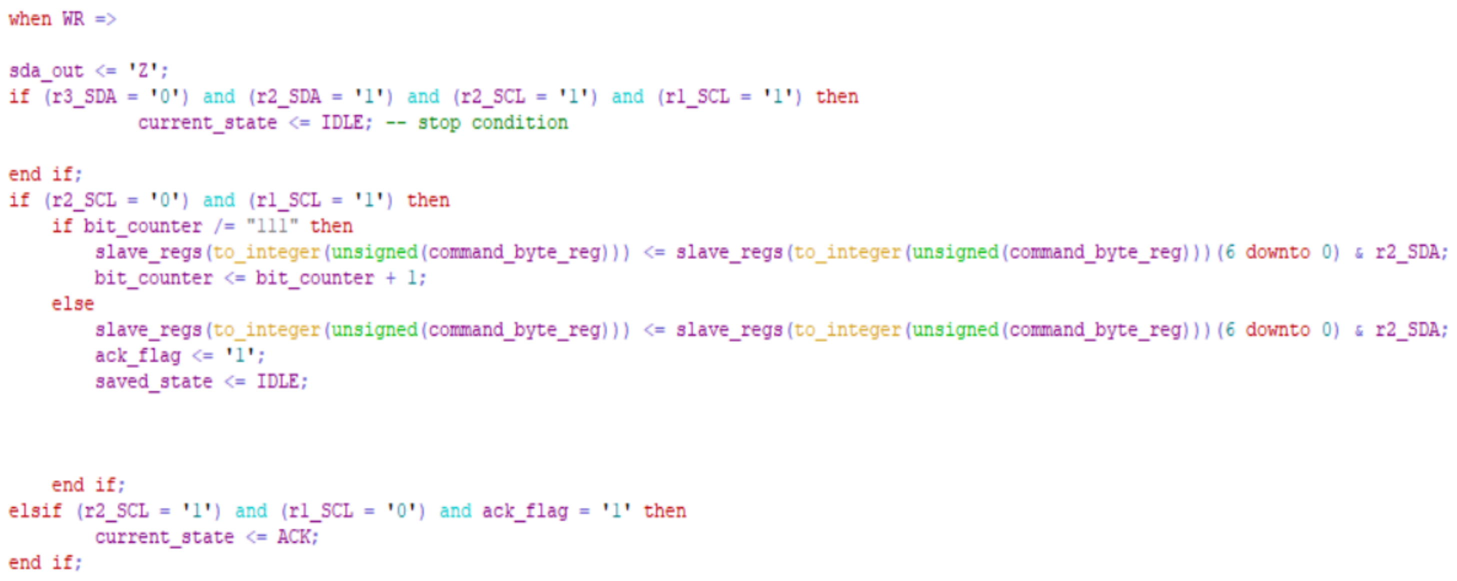
5. **Command Byte**

In the Command Byte state, the system captures and stores the value of the command byte, which indicates the target register address within the TCA6424A device. This value is saved into a dedicated internal register for later use during data transfer.

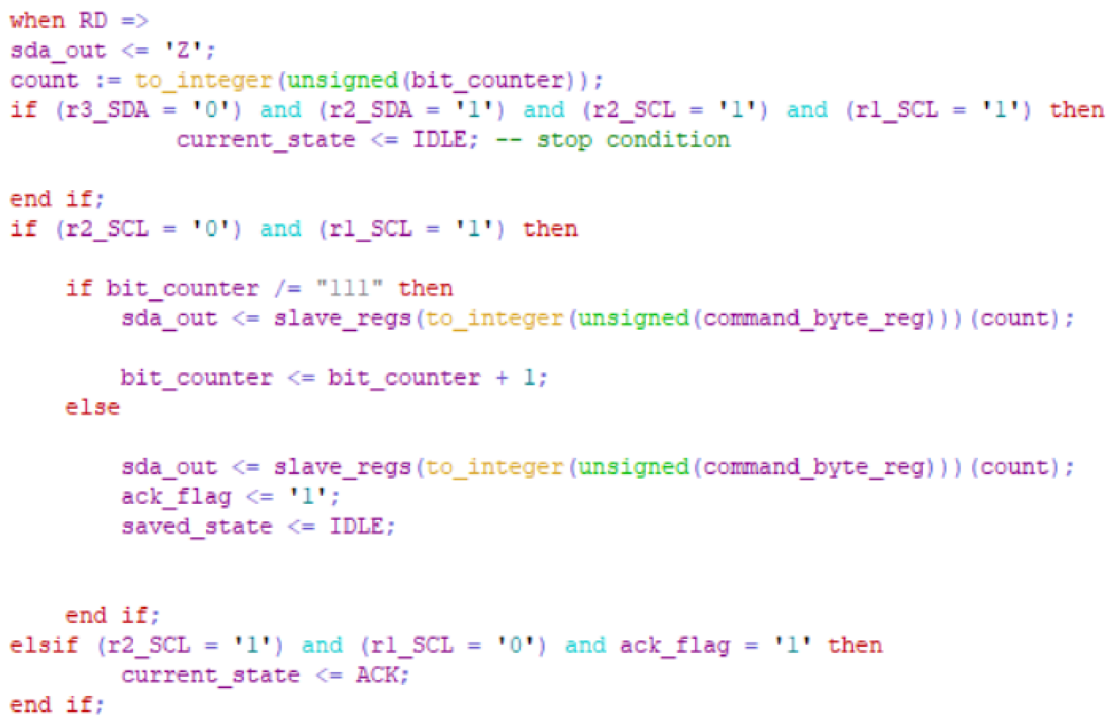
Following this, the state machine uses the previously stored R/W flag to determine the next course of action

```
when Command_Byte =>
sda_out <= 'Z';
if (r2_SCL = '0') and (r1_SCL = '1') then

    if bit_counter /= "111" then

        command_byte_reg <= command_byte_reg(6 downto 0) & r2_SDA;
        bit_counter <= bit_counter + 1;
    else
        ack_flag <= '1';
        command_byte_reg <= command_byte_reg(6 downto 0) & r2_SDA;

        if RW_flag = '1' then

                saved_state <= IDLE;

        else
                saved_state <=WR;

        end if;
     end if;

elsif (r2_SCL = '1') and (r1_SCL = '0') and ack_flag = '1' then
        current_state <= ACK;


end if;
```

6. Read/Write



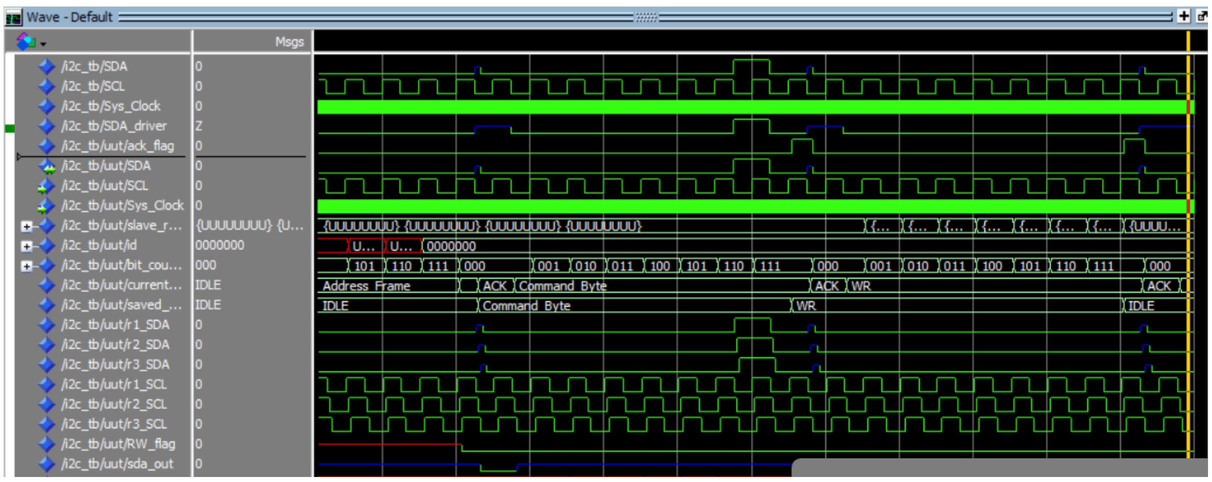
```
when WR =>

sda_out <= 'Z';
if (r3_SDA = '0') and (r2_SDA = '1') and (r2_SCL = '1') and (r1_SCL = '1') then
            current_state <= IDLE; -- stop condition

end if;
if (r2_SCL = '0') and (r1_SCL = '1') then
    if bit_counter /= "111" then
        slave_regs(to_integer(unsigned(command_byte_reg))) <= slave_regs(to_integer(unsigned(command_byte_reg)))(6 downto 0) & r2_SDA;
        bit_counter <= bit_counter + 1;
    else
        slave_regs(to_integer(unsigned(command_byte_reg))) <= slave_regs(to_integer(unsigned(command_byte_reg)))(6 downto 0) & r2_SDA;
        ack_flag <= '1';
        saved_state <= IDLE;



    end if;
elsif (r2_SCL = '1') and (r1_SCL = '0') and ack_flag = '1' then
        current_state <= ACK;
end if;
```



```
when RD =>
sda_out <= 'Z';
count := to_integer(unsigned(bit_counter));
if (r3_SDA = '0') and (r2_SDA = '1') and (r2_SCL = '1') and (r1_SCL = '1') then
            current_state <= IDLE; -- stop condition

end if;
if (r2_SCL = '0') and (r1_SCL = '1') then

    if bit_counter /= "111" then
        sda_out <= slave_regs(to_integer(unsigned(command_byte_reg)))(count);

        bit_counter <= bit_counter + 1;
    else

        sda_out <= slave_regs(to_integer(unsigned(command_byte_reg)))(count);
        ack_flag <= '1';
        saved_state <= IDLE;


    end if;
elsif (r2_SCL = '1') and (r1_SCL = '0') and ack_flag = '1' then
        current_state <= ACK;
end if;
```

Testbench Results

Below are the results obtained from the testbench, which is included in the VHDL project files. As shown in the waveform simulation, the system clock operates at a frequency 16 times higher than the I²C serial clock (SCL), enabling accurate timing control within the protocol. The SDA line is clearly visible, illustrating the data transfer process, including the transmission of addresses and data bytes. Furthermore, the acknowledgment (ACK) signals are present and correctly synchronized, confirming successful communication between the master and the implemented I²C slave module.



Wave - Default
{UUUUUUUU} {UUUUUUUU} {UUUUUUUU} {UUUUUUUU}
{... {... {... {... {... {... {... {UUUU...
U... U... 0000000
101 110 111 000 001 010 011 100 101 110 111 000 001 010 011 100 101 110 111 000
Address Frame ACK Command Byte ACK WR ACK
IDLE Command Byte WR IDLE

	Msgs
/i2c_tb/SDA	0
/i2c_tb/SCL	0
/i2c_tb/Sys_Clock	0
/i2c_tb/SDA_driver	Z
/i2c_tb/uut/ack_flag	0
/i2c_tb/uut/SDA	0
/i2c_tb/uut/SCL	0
/i2c_tb/uut/Sys_Clock	0
/i2c_tb/uut/slave_r...	{UUUUUUUU} {U...
/i2c_tb/uut/id	0000000
/i2c_tb/uut/bit_cou...	000
/i2c_tb/uut/current...	IDLE
/i2c_tb/uut/saved_...	IDLE
/i2c_tb/uut/r1_SDA	0
/i2c_tb/uut/r2_SDA	0
/i2c_tb/uut/r3_SDA	0
/i2c_tb/uut/r1_SCL	0
/i2c_tb/uut/r2_SCL	0
/i2c_tb/uut/r3_SCL	0
/i2c_tb/uut/RW_flag	0
/i2c_tb/uut/sda_out	0

Conclusion

In this project, a basic I²C slave protocol was successfully implemented in VHDL, following the specifications of the TCA6424A I/O expander. Through the design of a finite state machine and simulation in ModelSim, the critical aspects of the protocol were thoroughly understood, including timing management, START/STOP condition handling, and the correct transmission of data and ACK/NACK signals.

Special attention was given to challenges related to **clock domain crossing**, where synchronization techniques were applied to mitigate metastability issues. The simulation of waveforms provided visual verification of the design's functionality, while the consistent use of signals and variables in the VHDL code ensured reliable circuit behavior.

This experience significantly enhanced understanding of digital design and simulation tool usage, laying the groundwork for more advanced implementations and potential integration into real FPGA systems. Future extensions may include support for multiple slave devices, as well as the development of hardware testbenches for further verification.

References

[1] Basics of the I2C Communication Protocol - Scott Campbell

[2]Understanding Clock Domain Crossing (CDC) Checks and Techniques-Saurabh Verma, Ashima S. Dabare, Atrenta