FPGA Address Translator

GAURAV KUMAWAT

Table of Contents

1. Introduction

In many practical electronics projects, we often use several I2C-based sensors or memory chips on the same board. The I2C bus is designed in such a way that all devices share the same two wires: **SCL (clock)** and **SDA (data)**. To avoid confusion, each device must have a **unique 7-bit address**. But in the real world, we frequently face a problem:
Some devices come from the factory with the exact same I2C address, and the user cannot change it.

This creates a big issue because:

- If two devices share the same physical address,
- And the master sends that address on the bus,
- **Both devices respond at the same time**, which causes bus conflict and communication failure.

To solve this problem without adding extra hardware ICs, we designed an **FPGA-based I2C Address Translator**.
This FPGA acts like a **smart middle component** between the master and the actual slave devices.
It allows the master to use **virtual addresses**, and the FPGA internally converts them into the real physical addresses.

In simple words:

➡ **Master sends a virtual address**
➡ **FPGA modifies it in real-time using XOR mask**
➡ **FPGA forwards the modified address to the correct slave**
➡ **Only the intended slave responds**

This entire process happens automatically, and both slaves can be identical devices with the same factory-loaded address.

1.1 Key Features

- The system supports **virtual addressing**, meaning the master can treat two identical devices as if they have different addresses.
- The FPGA translates addresses **instantly using XOR masking**.
- The design follows **standard I2C protocol rules**, so no violation of timing or start/stop conditions.
- The SDA line is **bidirectional**, and the FPGA handles the direction without causing conflicts.
- The master does not need any code change. All magic happens inside the FPGA.
- Supports both **read and write operations** normally.

2. Architecture Overview

2.1 System Architecture with Address Translation

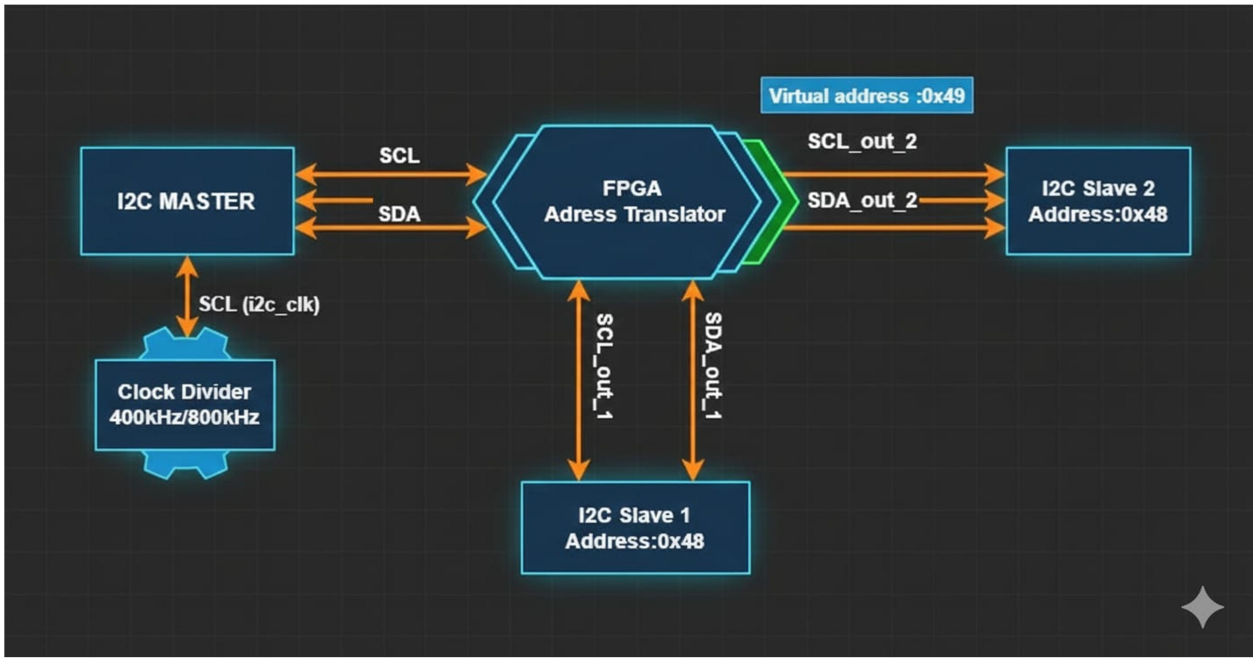
The complete system has the following important blocks:

1. **Master (Microcontroller or FPGA Master)**
   Sends data, clock, start/stop conditions, and addresses.
2. **FPGA Address Translator**
   The centrepiece that catches the virtual address and changes it into a physical address.
3. **Two Slave Devices**
   Both devices have the **same physical I2C address**, but FPGA differentiates them.
4. **Clock Divider**
   Generates the slower I2C clock (like 400 kHz) from a fast system clock (100 MHz).

The master always sends commands normally. But before the signals reach the actual slave chips, the FPGA captures the address and checks:

- Is this intended for Slave 1?
- Or is this intended for Slave 2?

Based on the virtual address, the FPGA applies a mask and forwards the correct physical address to the appropriate slave.
This ensures that although the slaves have identical hardware addresses, only one slave responds.



2.2 Problem Statement

In standard I2C communication:

- There is only **one SDA line** shared by all devices.
- Every slave listens for its 7-bit address.
- Only the matching slave is allowed to pull SDA low (ACK).

But if two identical sensors (like two temperature sensors) both have the address **0x48**, then:

- Both will see their address on the bus,
- Both try to send an ACK,
- Both try to send data,
- And the bus becomes unstable.

This normally forces designers to:

- Replace sensors,
- Use complex multiplexers,
- Or redesign the whole board.

Our solution removes the need for any extra IC or redesign.

2.3 Our Solution

We use a very simple idea:

- FPGA listens to the master.
- When master sends a virtual address (like 0x49 or 0x48),
- FPGA decides:
    - "Oh! This should go to Slave 1"
    - Or "This should go to Slave 2"
- It then modifies the last bit of the address using XOR,
- And sends the address to the correct slave.

So even though both slaves are physically 0x48,
the master behaves as if:

- Slave 1 → 0x48
- Slave 2 → 0x49

This provides complete flexibility.

2.4 Virtual Addressing Flow

Here is the complete communication flow:

1. Master creates a **START** condition.
2. Master sends **7-bit virtual address + R/W bit**.
3. FPGA captures the address before it reaches the slaves.
4. FPGA applies mask:
    - If the master sent 0x49 → Flip LSB → becomes 0x48 (slave 2)
    - If master sent 0x48 → No change → becomes 0x48 (slave 1)
5. FPGA forwards the modified address only to the selected slave.
6. Only that slave responds with **ACK**.
7. Data transfer happens normally.
8. Master sends **STOP** and communication ends

3. Module Descriptions

3.1 Clock Divider Module (clk_divider.v)

Purpose:
The job of the clock divider is to take the fast 100 MHz system clock and slow it down so that it can be used for I2C communication. Since I2C works at a much lower frequency (like 400 kHz), the module creates precise timing pulses called *ticks* that help in generating an accurate SCL clock signal.

3.1.1 How It Works

The working of the clock divider is actually simple:

- It receives the high-speed system clock (100 MHz).
- Inside, a counter keeps increasing with every clock pulse.
- When this counter reaches a specific value called **Max_count**,
  it produces a **tick** — this tick is used to time the I2C operations.
- After generating the tick, the counter resets to zero and starts counting again.

This repeating process produces a stable and slower rhythm for I2C.

Important Parameter:

- **Max_count = 125**
  This value is chosen so that:

$$100\text{MHz} \div (2 \times 400\text{kHz}) = 125$$

This means:

- Count 125 system-clock cycles → produce a tick
- 2 ticks = 1 full I2C clock period
- Resulting SCL frequency ≈ **400 kHz**

3.2 I2C Master Module (i2c_master.v)

Purpose:
The I2C Master module is basically the "brain" of the whole I2C system.
It decides *when* to send data, *which* device to talk to, and *whether* it wants to read or write.
It also ensures that all I2C rules are followed properly.

3.2.1 Key Features

- Sends the **7-bit address** plus the **R/W bit**, completing the 8-bit address frame.
- Supports **8-bit data transfer** in both read and write mode.
- Generates **START** and **STOP** conditions exactly as per I2C protocol.
- Controls the two important lines of I2C:
  - **SCL** → Clock
  - **SDA** → Data
- Handles **bidirectional communication**, meaning sometimes it drives the line and sometimes it releases it.

3.2.2 Interface Ports

Inputs:

- **clk** → The high-frequency 100 MHz system clock.
- **rst** → Reset signal to restart the module.
- **address[6:0]** → The 7-bit address of the slave that master wants to talk to.
- **data_in[7:0]** → The 8-bit data that the master wants to send.
- **read_write** → Operation mode (1 = read from slave, 0 = write to slave).
- **en** → When this signal becomes high, the master starts a new I2C transaction.

Outputs:

- **data_out[7:0]** → If the master performs a read, the received data comes here.
- **SCL** → The clock signal for the I2C bus (controlled by master).
- **SDA** → The data signal for I2C (shared with slaves, so bidirectional).

FSM Operation

The master works using a Finite State Machine (FSM).
Its main states are:

1. Idle
2. Start condition
3. Sending address
4. Waiting for ACK
5. Sending or receiving data
6. Sending final ACK/NACK
7. Stop condition

This well-defined sequence ensures the master always talks to the slave in the correct I2C order.

3.3 I2C Slave Module (i2c_slave.v)

Purpose:
The slave module represents an actual I2C device connected to the bus.
Its main responsibility is to listen for its own address and respond only when the address matches.

3.3.1 Key Features

- Has a **configurable 7-bit address** so that it knows when the master is talking to it.
- Includes an internal **8-bit memory register** to store received data or hold data for sending.
- Automatically detects **START** and **STOP** conditions.
- Automatically sends **ACK** when its address matches.
- Can either send data back to the master or receive data from it, depending on the R/W bit.

Important Parameters:

- **address = 7'd48**
  Default hardware address of the slave (can be changed).
- **address_width = 3'd7**
  Indicates that the slave uses a 7-bit address system.

How the Slave Behaves

The slave keeps watching the bus continuously. When it detects:

- START condition
- followed by its own address

it responds with an ACK and becomes active in the communication.
If it’s a write operation, it stores the data into its internal register.
If it’s a read operation, it sends data stored in that register back to the master.

4. FPGA Address Translator

4.1 Purpose and Overview

The FPGA Address Translator is the *heart* of the entire project.
This is the block that makes it possible to connect multiple I2C slave devices that have the **same hardware address**, something that is normally impossible in a standard I2C system.

In simple words, the translator works like a **smart middleman** between the I2C master and the slave devices. It carefully monitors everything happening on the I2C bus and performs the following tasks:

- **Address Translation:**
  It changes (modifies) the incoming address bits "on the fly" using a simple XOR mask. This helps the FPGA convert a *virtual* address (sent by master) into the *real* physical address of the slave.
- **Signal Routing:**
  It ensures that only the correct slave device receives the address and data meant for it.
- **Bus Multiplexing:**
  Since the SDA line is bidirectional, the FPGA decides who should drive the line – the master or the selected slave.
- **Protocol Monitoring:**
  It watches for I2C START and STOP conditions and ensures the communication does not break I2C rules.

This entire system works seamlessly, making the master believe it is talking to slaves with different addresses, even though both slaves actually share the exact same physical address.

4.2 Why Address Translation Is Needed

The motivation behind this design comes from a real and common I2C problem.

The Main Problem

If you connect two identical I2C devices (e.g., two sensors that both use address 0x48), both will respond whenever the master sends this address.
This causes:

- Data collision
- Bus contention
- Communication errors

- Unpredictable behavior

In standard I2C, you simply **cannot** use two devices with the same address on the same bus.

Traditional Solutions

There are existing methods, but they are not always practical:

- **Address pins:**
  Many chips don't offer configurable address pins, or they offer only 1 bit, meaning only 2 possible addresses.
- **I2C multiplexers:**
  Using mux chips increases the component cost, board complexity, and requires extra control signals.

FPGA-Based Solution

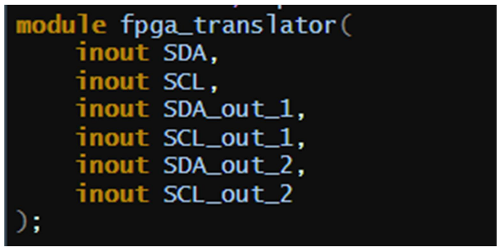
Our FPGA approach solves all of these limitations:

1. **Virtual Addressing:**
   The master uses different virtual addresses (e.g., 48 and 49) to differentiate between slaves.
2. **Real-time Address Translation:**
   The FPGA captures the address and flips bits using an XOR mask before forwarding it to the slaves.
3. **Selective Routing:**
   Only the intended slave receives the physical address, while the other slave ignores the transaction.

This technique makes it possible to use multiple identical devices without any extra hardware components.

4.3 Module Interface

The FPGA Translator module includes several bidirectional I/O ports because I2C uses open-drain lines.



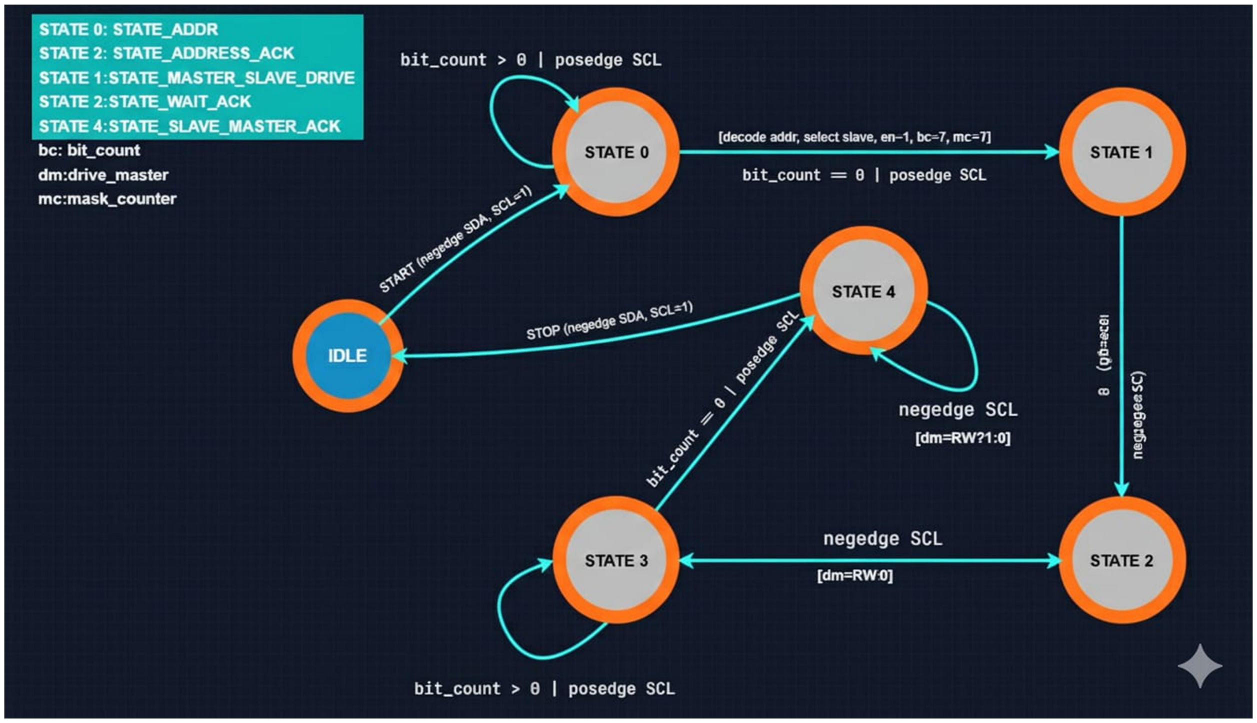
```
module fpga_translator(
    inout SDA,
    inout SCL,
    inout SDA_out_1,
    inout SCL_out_1,
    inout SDA_out_2,
    inout SCL_out_2
);
```

All these signals are **inout** because:

- No one device “owns” the bus completely.
- Devices must release the line (High-Z) when they are not driving it.
- FPGA needs to redirect SDA/SCL to the appropriate slave during translation.

This keeps the I2C communication electrically correct and prevents short circuits on the SDA line.

4.4 FSM States


STATE 0: STATE_ADDR
STATE 2: STATE_ADDRESS_ACK
STATE 1:STATE_MASTER_SLAVE_DRIVE
STATE 2:STATE_WAIT_ACK
STATE 4:STATE_SLAVE_MASTER_ACK
bc: bit_count
dm:drive_master
mc:mask_counter
bit_count > 0 | posedge SCL
STATE 0
[decode addr, select slave, en–1, bc=7, mc=7]
bit_count == 0 | posedge SCL
STATE 1
START (negedge SDA, SCL=1)
STATE 4
STOP (negedge SDA, SCL=1)
IDLE
negedge SCL
[dm=RW?1:0]
bit_count == 0 | posedge SCL
STATE 3
negedge SCL
[dm=RW:0]
STATE 2
bit_count > 0 | posedge SCL

- **STATE_ADDR:**
  Collect the address bits sent by the master.
- **STATE_ADDRESS_ACK:**
  Forward the address to the selected slave and wait for its ACK.
- **STATE_MASTER_SLAVE_DRIVE:**
  Decide whether the master or slave will drive SDA next.
- **STATE_WAIT_ACK:**
  Handle 8-bit data transfer (read/write phase).
- **STATE_SLAVE_MASTER_ACK:**
  After data transfer, final ACK/NACK is processed.

4.5 START and STOP Condition

START Detection

A START happens when:

- SDA falls **from HIGH to LOW**
- While SCL is **HIGH**

When this condition is seen:

- start flag is set
- FSM resets bit counters
- FSM enters address-capture mode

STOP Detection

A STOP happens when:

- SDA rises **from LOW to HIGH**
- While SCL is **HIGH**

When STOP is seen:

- Flags reset
- FPGA returns bus control to the master
- Translator returns to idle state

These conditions follow the official I2C protocol definition

4.6 Address Capture and Translation

As SCL rises, each bit of the address is sampled from SDA.
The FPGA stores this in `temp_address`, decreasing `bit_count` each time.

Once all 8 bits are captured:

- FPGA checks the 7 address bits (temp_address[7:1])
- If virtual address = 49 → Slave 2
- Else (virtual address = 48) → Slave 1

It then prepares to send the ACK state and applies XOR mask to Slave 2's address line.

4.7 Bus Control Handoff

At every falling edge of SCL, the FPGA decides:

- Should the **master** drive SDA?

- Or should the **selected slave** drive it?

This depends on:

- Whether ACK is expected
- Whether the operation is read or write
- Whether address phase is complete

In read:

- Slave drives SDA
  In write:
- Master drives SDA

The FPGA ensures proper high-impedance control to avoid bus conflicts.

4.8 Address Translation Example

Let's walk through a real example in simple words.

System Setup

- Both Slave 1 and Slave 2 have real physical address = **48**
- Master uses **48** (for Slave 1) and **49** (for Slave 2) as virtual addresses

Scenario: Master wants Slave 2

Master sends virtual address **49** on SDA.

Step 1: Master Sends Bits

The bit sequence (including R/W bit) is captured by FPGA.

Step 2: Translator Applies XOR Mask

Only for Slave 2, the FPGA flips bit-1 using XOR:

- Original address bit-1 = 1
- XOR mask bit-1 = 1 → produces 0

So **49 becomes 48**.

Step 3: Routing Decision

FPGA sets:

- select_slave = 0 → choose Slave 2

Step 4: Slave Response

- Slave 1 sees address 49 → No response

- Slave 2 sees 48 → Match → Sends ACK

Only Slave 2 participates in the rest of the transaction.

This allows master to talk to two identical slaves without any conflict.

5. Design Challenges

5.1 Clock Domain Issue

System_clock ≠ I2C_clock
They differ by almost 250 times.
So synchronous capture must be used; otherwise timing violations occur.

We solved this using:

- Multi-flop synchronizers
- Separate edge detection logic
- Clock enable pulses

5.2 SDA Direction Control

SDA is bidirectional.
Master drives during write,
Slave drives during read.

Translator uses:

- Controlled tri-state buffers
- `drive_master` signal
- High-Z output when needed

This avoids bus fighting.

6. Protocol Specifications

I2C uses a fixed communication format:

```
START → Address → R/W → ACK → Data → ACK → STOP
```

Important timings:

- Clock frequency around 400 kHz
- Setup time ~1.25 µs
- Hold time ~1.25 µs
- STOP/START conditions must be cleanly detected

Our FPGA handles these carefully to remain protocol-accurate.

7. Conclusion

The FPGA-based I2C Address Translator successfully solves a real engineering problem: **allowing multiple identical I2C devices to work on the same bus without address conflict.**

By using virtual addressing and XOR-based on-the-fly translation, the system:

- Requires no extra IC
- Works transparently with the master
- Fully supports I2C timing
- Handles both read and write
- Provides safe routing between slaves

The design is reliable, lightweight, and useful in projects where hardware addresses cannot be changed.
This makes the FPGA translator a cost-effective and flexible solution for real-world embedded applications.

8. OUTPUT

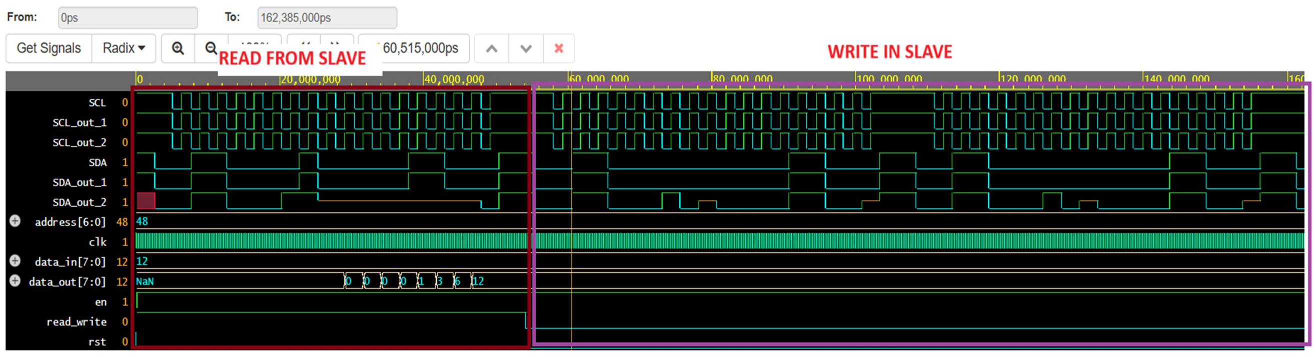


Figure 1: FPGA Translator Output Image 1(0X48)

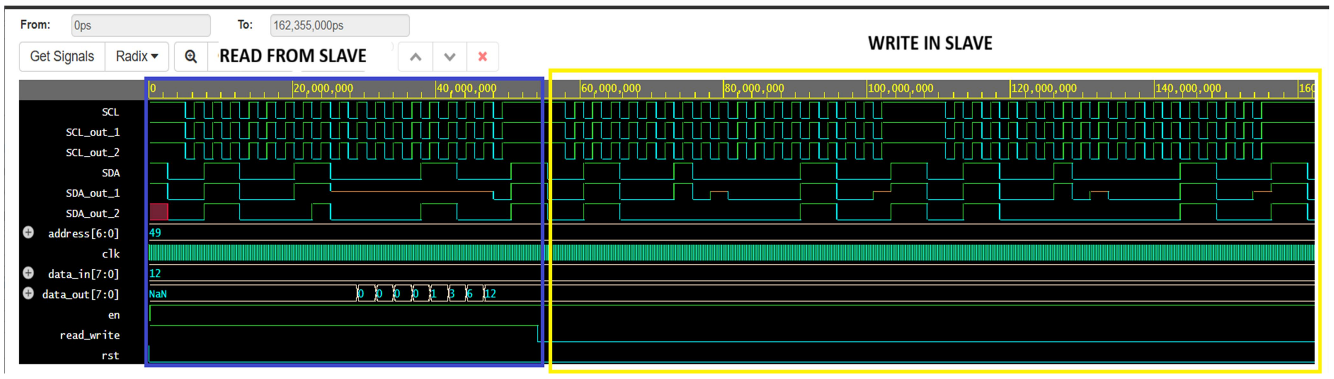


Figure 2: FPGA Translator Output Image 2(0X49)