**Design Documentation: I²C Address Translator**

Here’s a brief walkthrough of my I²C address translator design, covering how it works and some of the challenges I encountered while building it.

**1. My Approach to the Architecture**

At its heart, my design acts like a smart traffic cop for the I²C bus. The whole point is to solve a common problem: what to do when you have two devices that are hardwired with the same I²C address. My module sits quietly between the main controller (the master) and these devices, listening in on the conversation.

The setup is straightforward. The module has one input SDA line to listen to the master and two separate output SDA lines (SDA1 and SDA2), one for each target device. When the master starts a transaction, my module intercepts it. It checks the address the master is trying to talk to. If it's a generic address, it does nothing. But if it's one of the special "virtual" addresses I've defined (0x21 or 0x22), the module jumps into action, re-routes the entire transaction to the correct device using its *real* address, and makes sure the other device never even sees the message.

**2. The FSM: The Brains of the Operation**

To manage the whole process, I built the logic around a Finite State Machine (FSM). It’s the core of the controller, stepping through the I²C protocol to make sure everything happens in the right order.

Here’s a quick look at its journey through a transaction:

* **IDLE State**: This is the starting point, where the module is passively listening for a start signal from the master. It’s essentially waiting for something to happen.
* **ADDR State**: As soon as the master kicks off a transaction, the FSM moves here. Its only job is to carefully capture the 8-bit address byte coming down the SDA line. Once it has the full address, it makes a decision.
* **TRANSLATE State**: If the captured address matches one of our special virtual addresses, the FSM enters this state. Here, my module switches hats—it stops being a passive listener and becomes a temporary I²C master. It then initiates a *new* transaction on the correct output line (SDA1 or SDA2), sending out the device's true physical address.
* **DATA State**: With the connection now established to the correct physical device, the FSM settles into this state. It becomes a simple passthrough, relaying all the data from the master straight to the chosen device until the transaction ends.
* **STOP State**: If the address from the master didn't match any of our virtual targets, the FSM uses this state to gracefully ignore the transaction and reset itself back to IDLE, ready for the next one.

**3. The Secret Sauce: How the Translation Works**

The actual address swap is the core trick of this design, and it happens in the moment between capturing the virtual address and transmitting the physical one.

1. **Listen and Compare**: First, the module captures the full 8-bit address byte from the master. It immediately checks the 7-bit address part against the known virtual addresses, 0x21 and 0x22.
2. **Rebuild the Address**: If it gets a match, it builds a brand-new 8-bit address byte. It takes the target's real, physical address (0x48) and appends the original Read/Write bit from the master's request. This is critical because it preserves the master's original intent—if it wanted to read, we still perform a read, and the same for a write.
3. **Pick a Lane**: While rebuilding the address, it also sets a target flag. This simple flag tells the module whether to send the following communication down the SDA1 line or the SDA2 line.
4. **Send it Out**: Finally, the module sends this newly created address byte to the correct device, which thinks it's being addressed directly by a master. From that point on, all data is relayed to that device alone.

**4. Design Challenges I Faced**

Building this wasn't without its interesting puzzles. Here are a few of the main challenges I worked through:

* **Timing is Everything**: The trickiest part was managing the I²C timing. Since my module has to first receive an entire address byte before it can re-transmit a new one, it naturally introduces a delay. I had to be careful that this delay wasn't so long that the master would time out while waiting for an acknowledgment.
* **A Classic Pipelining Bug**: I ran into a subtle but classic hardware bug. My logic was checking the address register on the *exact same clock cycle* the final bit was arriving. This meant the comparison was always one cycle behind, using the value *before* the last bit was stored. It took a bit of debugging to realize this. I fixed it by changing the logic to directly check the incoming SDA bit as part of the final comparison, which solved the problem cleanly. It was a great reminder of how pipeline delays work in synchronous logic.
* **Handling Two-Way Traffic**: This design is focused on getting master-write operations working flawlessly. A more comprehensive solution would also need to handle master-reads, where data flows back from the slave. This would require adding tristate logic to the output SDA lines, allowing them to switch from "driving" the line to "listening" for data. While I kept it out of scope for this task, it was a key consideration for future expansion.