I²C Address Translator

This short document explains the address-translator design in plain language — architecture, how the finite-state logic works, how address remapping is implemented, and the design challenges we faced. Use this in your repo README or submission documentation.

1. Architecture overview

Think of the translator as a little bridge sitting on the I²C bus between the master and two identical target devices. It has three main pieces:

- **Upstream interface (toward the Master)**
  - Presents itself as a standard I²C slave on the master's bus.
  - Uses open-drain lines (SDA pulled low to drive, released to float) and watches SCL and SDA to detect START/STOP and incoming bits.
- **Translator core (the brain / FSM)**
  - Samples bits from the master, detects the address byte, decides whether the addressed device should be remapped, and then controls downstream activity.
  - Implements timing- and bus-friendly logic: shift registers for bits, counters for bit positions, and an FSM that sequences events.
- **Downstream interface (toward the targets)**
  - Acts as an I²C master for whichever physical device is being proxied so the actual target devices can remain unaware.
  - Generates SCL (bit-banged using a clock divider inside the FPGA) and drives SDA as an open-drain master.

A few small helpers complete the architecture: pull-ups in the testbench, a pair of slave models for validation, and debug prints during simulation.

2. FSM / Logic explanation

The translator's behavior is driven by a compact finite-state machine (FSM). Here's the user-level flow in simple terms:

1. **IDLE** — The translator watches the upstream bus. When the master issues a START, the translator starts listening.

2. **ADDRESS RECEIVE** — On each rising edge of upstream SCL the translator samples SDA and builds an 8-bit byte in a shift register. After 8 bits arrive it treats that as the address+R/W byte.

3. **ADDRESS CHECK & ACK** — The translator checks if the received 7-bit address matches one of the virtual addresses it should translate (example: 0x48 or 0x49). If it matches, the translator asserts ACK back to the master (by pulling SDA low during the ACK clock window). If it does not match, the translator NACKs and returns to IDLE.

4. **DOWNSTREAM START & FORWARD** — If the address matched, the translator becomes an I²C master on the downstream bus and issues a START followed by the *physical* target address (for example a fixed 0x48 physical address) — possibly with a changed R/W bit if needed.

5. **DATA FORWARDING or READ**
    - **Write (Master → Target):** The translator receives the data bytes from the upstream master and forwards each byte to the selected downstream target, waiting for ACK/NACK at each step and relaying behavior upstream.
    - **Read (Master ← Target):** The translator requests bytes from the downstream target and then presents those bits to the upstream master (driving/releasing SDA in sync with the master's SCL).
6. **STOP / Cleanup** — When the upstream master sends a STOP, translator releases all lines and returns to IDLE.

Key implementation details in the FSM logic: - Sampling is done on SCL **rising** edge (I²C standard) and we shift MSB-first. - ACK windows are carefully timed: the translator pulls SDA low while SCL is high for the ACK bit and releases on SCL falling edge. - A single-shot 'ready' flag prevents repeating the same ACK/NACK decision every clock cycle — only one ACK per received byte.

3. How address translation is implemented

The translation approach is simple and deterministic:

1. **Virtual addresses on the upstream bus** (what the master uses) are recognized by the translator (for example 0x49).
2. **Translator maps the virtual address to a physical address** that the actual downstream device understands (for example translator remaps upstream 0x49 → downstream physical 0x48). The mapping can be implemented as a small lookup table or a pair of comparators (the initial version uses hard-coded constants for simplicity).
3. **The translator responds to the master** (ACK/NACK) as if it were the target device, so the master is unaware anything changed.
4. **The translator then acts as an I²C master to the physical target**: it issues a START and the mapped physical address, then forwards data bytes (or fetches data for reads). Throughout this, it keeps track of ACKs from downstream so it can mirror the proper behavior upstream.

This keeps the actual device unchanged — it still uses its factory address — while the translator makes it appear to the bus master as if the device had a different address.

4. Design challenges faced

While implementing and simulating this small device I hit a few practical problems; here's what they were and how they were handled:

- **Edge timing & sampling correctness.** I²C requires sampling SDA on SCL **rising** edge and driving SDA appropriately during ACK windows. Early bugs came from sampling on the wrong edge or shifting bits in the wrong order (MSB vs LSB). Fix: explicitly sample SDA on SCL rising and shift MSB-first.

- **Open-drain modeling in simulation.** Real I²C uses tri-state/open-drain lines with pull-ups. Simulating this in Verilog needs `inout` signals, driver registers that pull low or release to z, and `pullup()` statements in the testbench. Getting this wrong produced stray z readings or missing ACKs.

- **Timing between upstream and downstream buses.** The translator lives between two buses with different roles (it's a slave upstream, a master downstream). Generating downstream SCL at a correct

frequency and sequencing signals so that downstream ACKs arrive before the translator must respond upstream is subtle. Fix: use a clock divider and small FSM pauses to ensure downstream bytes and ACKs are sampled correctly before responding upstream.

- **Avoiding repeated handling/printing.** Early versions repeatedly handled the same event each system clock cycle (because flags were left set), producing huge console floods. Fix: use single-shot flags (clear them immediately after handling) so events are processed only once.

- **Multi-byte transactions and read-mode complexity.** Forwarding multi-byte writes is straightforward, but reads are trickier: the translator must request data from downstream and *then* present those bits to the upstream master on the master's SCL. This required careful design of read buffers and bit counters to present bits at exactly the right times.

- **Extensibility and runtime config.** The simplest implementation hard-codes the translation mapping. For a production-ready solution it's best to add a small configuration register (e.g., accessible over a side channel or via special I²C commands) to change mappings at runtime — this was left as a bonus feature.