**FPGA-based I²C**

**Detailed step-by-step plan to implement the FPGA I²C address-translator (complete guide)**

Great — below is a thorough, actionable step-by-step plan you can follow to implement the translator, simulate it (EDAPlayground), synthesize it, and produce the deliverables. I’ll cover architecture, FSMs, timing math, testbench topology and test cases, EDA Playground setup, what to include in the repo/docs, and a 15-day schedule/checklist. I avoid dumping a complete submission-ready file so you can adapt the design into your own coding voice (important given the assignment’s AI restriction). Where code is helpful I provide **small skeletons and precise pseudocode** you can implement.

**1 — High level summary / topology**

* Upstream bus (master side): I²C Master (MCU) + Device1 (0x48) + FPGA-upstream (acts as **slave**) all on same SCL\_up/SDA\_up lines.
* Downstream bus (device side): FPGA-downstream (acts as **master**) talking to Device2 (0x48).
* FPGA translates a **virtual address** (e.g. 0x49) used by the master to the real downstream device address (0x48).
* The FPGA must proxy all bytes and ACK/NACKs so reads/writes are seamless.

**2 — Required modules (software design blocks)**

1. **Top module** (i2c\_translator) — wires up upstream and downstream and contains config registers.
2. **Upstream I2C slave controller** — detects START/STOP, receives address & data from master, sends ACK, and hands data/requests to proxy logic.
3. **Downstream I2C master controller** — generates SCL/SDA for device(s), sends address and data, reads data back, and returns ACK info.
4. **Proxy logic / bridge FSM** — coordinates upstream & downstream FSMs (forwards bytes, handles ACKs, repeated START, multi-byte).
5. **Clock divider / SCL generator** — to support 100 kHz and 400 kHz (parameterizable).
6. **Testbench models** — upstream master stimulus, Device1 (direct on upstream bus), Device2 (downstream), waveform/print checks.

**3 — Timing: choosing system clock and divider math**

Pick a convenient system clock (typical FPGA dev boards: 50 MHz or 100 MHz). Example: **SYSCLK = 50 MHz**.

I²C timing: rough half-period values are sufficient for synthesis and simulation.

* 100 kHz: period = 10 µs → half period = 5 µs  
  50 MHz → cycles per half period = 50e6 \* 5e-6 = 250 cycles
* 400 kHz: period = 2.5 µs → half period = 1.25 µs  
  50 MHz → cycles per half period = 50e6 \* 1.25e-6 = 62.5 → use 62 or 63 cycles.

So implement a parameterizable clock divider:

localparam integer HALF\_CYCLES\_100K = 250; // for 50MHz

localparam integer HALF\_CYCLES\_400K = 63; // approximate

Use these to toggle SCL\_down in the downstream master FSM. For upstream (master drives SCL\_up), sample SCL\_up directly (we only need SCL edge detection to sample SDA for slave behavior).

**4 — Physical I/O & open-drain behavior**

* SDA lines are open-drain: devices pull low or release (hi-Z) and external pull-ups pull high. On FPGA use tri-state outputs or IOBUF with drive enable.
* Implementation pattern:

// tristate output

reg sda\_oe; // 1 -> drive value, 0 -> high-Z (release)

reg sda\_out; // value to drive (0 to pull low)

assign SDA = sda\_oe ? 1'b0 /\* drive low only \*/ : 1'bz;

You only ever drive '0'. To release line, set sda\_oe = 0.

Upstream side: release SDA unless you ACK (drive low during ACK bit).  
Downstream side: when acting as master to send bits, drive low for 0 or release for 1 (since pull-ups create 1).

**5 — Upstream (slave) FSM — detailed states & behavior**

**Goal:** appear as a slave for certain virtual addresses; capture bytes from master or deliver requested read data.

States:

* U\_IDLE: wait for START (detect SDA falling while SCL high).
* U\_ADDR\_RX: shift 8 bits (7-bit address + R/W) sampled on SCL rising edges.
* U\_ADDR\_ACK: if address == virtual address (cfg) → assert ACK (drive SDA low during ACK bit window), else release.
* U\_DATA\_RX: if master writes (R/W=0): receive bytes (shift on SCL rising) and after each byte, drive ACK or NACK depending on downstream ACK result.
* U\_DATA\_TX: if master reads (R/W=1): present bytes (drive SDA on SCL low), after each byte release to let master ACK/NACK (sample ACK on SCL rising).
* U\_REP\_START: handle repeated START (SDA fall while SCL high) without going to STOP.

Key details:

* Sample SDA on **SCL rising** (I²C standard).
* Drive ACK by pulling SDA low **during the 9th bit** while SCL is high (but practically you set sda\_oe before the ACK bit time).
* When a byte is received from master and must be forwarded to downstream, hand the byte to the proxy bridge and wait for downstream result before ACKing upstream.

Pseudocode snippet (address capture):

on START:

state = U\_ADDR\_RX

bit\_cnt = 7

shift = 0

on SCL rising while state == U\_ADDR\_RX:

shift[bit\_cnt] = SDA

if bit\_cnt==0:

addr7 = shift[7:1]

rw = shift[0]

if addr7 == cfg\_virtual\_addr:

state = U\_ADDR\_ACK

else

// not ours -> release line and goto IDLE (do not interfere)

**6 — Downstream (master) FSM — detailed states & behavior**

**Goal:** when FPGA must communicate with device2, it becomes master on downstream bus, issues START, sends translated address + R/W, forwards/sends data, reads data back.

States:

* D\_IDLE
* D\_START — generate a START: SDA transition high→low while SCL high (we control both).
* D\_SEND\_ADDR — send 8 bits of address (7-bit + R/W).
* D\_SEND\_BYTE — send data bytes, sample ACK after each byte.
* D\_RECV\_BYTE — release SDA and sample incoming data from device (sample on SCL rising).
* D\_STOP — drive STOP: SDA low→high while SCL high.

Bit timing for master send:

1. While SCL low, set SDA to next bit value (drive low for 0, release for 1).
2. Wait half period.
3. Raise SCL (SCL high) — hold half period (SDA must remain stable during high).
4. Lower SCL.
5. Repeat.

ACK handling:

* After sending 8 bits, release SDA (hi-Z) before the 9th clock. On the 9th clock’s rising edge, sample SDA for ACK (0 is ACK).

Pseudocode for sending a byte:

task send\_byte(byte b):

for i from 7 downto 0:

drive SDA = b[i] while SCL low

delay half\_period

SCL = 1

delay half\_period

SCL = 0

// ACK bit

release SDA (hi-Z)

delay half\_period

SCL = 1

sample ack = SDA // 0 => ACK

delay half\_period

SCL = 0

return (ack == 0)

Make SCL toggling driven by a counter that uses the half\_period value calculated from system clock and chosen I²C speed.

**7 — Proxy logic: bridging upstream ↔ downstream**

* On upstream address match (virtual addr):
  + If master writes (R/W=0): receive bytes from master (U\_DATA\_RX), for each byte:
    1. Store received byte in a FIFO / small buffer.
    2. Trigger downstream operation: START (if first byte), send real device address with write, send data bytes to downstream.
    3. Read ACK from downstream; pass upstream ACK back to master (drive ACK low if downstream ACKed).
  + If master reads (R/W=1): upon address reception:
    1. FPGA acts as downstream master, issues START + real address with READ bit.
    2. Read bytes from device into a FIFO.
    3. Present bytes to upstream master (U\_DATA\_TX) — master will ACK/NACK each byte; when master NACKs (end of read) or STOP occurs, issue STOP to downstream.
* Handle repeated START: if upstream issues a repeated START to change between writing register address followed immediately by a read, the translator should keep session state and do corresponding downstream repeated START.

Implementation notes:

* Use small FIFOs or single-byte buffers and handshakes for each byte (simpler for simulation).
* Always propagate downstream ACK/NACK upstream as appropriate.
* If downstream NACKs, the translator should NACK upstream and optionally generate STOP.

**8 — Address translation policy & configuration**

Options:

1. **Static config pins / generics** — provide cfg\_orig\_addr and cfg\_new\_addr as parameters/inputs (easy for simulation).
2. **Runtime via special I²C config address** — reserve a special address (e.g. 0x7E) that the translator listens to; when the master writes to that address, take configuration bytes to set mapping registers (allow runtime change). This is a useful bonus.
3. **UART or DIP switches** — other ways to set translation.

I recommend implementing runtime configuration via a reserved I²C config address (requires the translator to respond to that address even when virtual mapping not active).

**9 — Testbench topology & testcases (what to simulate)**

Testbench components:

* tb\_master — generates upstream START / address / data sequences.
* tb\_device1 — slave model sitting on upstream bus address 0x48, responds to direct upstream operations (should respond when master addresses 0x48).
* tb\_translator — your design under test, upstream connected to master & device1; downstream to tb\_device2.
* tb\_device2 — slave model at address 0x48 connected to downstream bus; translator will communicate with it by translating virtual 0x49.

Testcases:

1. **Direct master→Device1 write**: Master writes to 0x48 (upstream). Ensure device1 receives data and translator does not respond (i.e., translator releases SDA and doesn't ACK).
2. **Master→Virtual (0x49) write (forward to Device2)**: Master writes to 0x49 — translator should ACK, take payload, issue downstream write to Device2 (0x48), and Device2 should receive correct bytes.
3. **Master read from virtual address**: Master sends read to 0x49. Translator should fetch data from Device2 and return it to master. Validate data integrity.
4. **Repeated START case**: Master writes a register address to 0x49 then immediately does a repeated START and reads from 0x49 — translator should forward as repeated START to device2 and return correct data.
5. **Error path**: Device2 NACKs — translator should NACK upstream.

Add message prints and waveform dumps (if using GTKWave) to inspect SDA/SCL signals and internal FIFOs.

**10 — EDA Playground setup (step-by-step)**

1. Create a new “Icarus Verilog + GTKWave” project (or ModelSim if you prefer).
2. Add files:
   * i2c\_translator.v (top + instantiates upstream & downstream FSMs)
   * i2c\_upslave.v (optional split)
   * i2c\_downmaster.v (optional split)
   * tb\_i2c\_translator.v (testbench)
   * i2c\_device\_model.v (simple slave model used for Device1 and Device2)
3. Set top module to testbench file.
4. Run simulation. On Icarus the console shows $display prints you add. Attach .vcd waveform and open GTKWave to inspect SCL/SDA timings.
5. Make playground public and copy the link into the required field.

Helpful hints:

* Use $dumpfile("dump.vcd") and $dumpvars(0,tb\_top) in your TB.
* Add $display lines at transaction boundaries: print addr bytes, data bytes, ACK results. Use assertions to check expected behavior, e.g. if (device2.received\_byte !== expected) $fatal("Mismatch!").

**11 — What to show in the simulation demo (EDAPlayground)**

* Waveform showing:
  + Upstream SCL/SDA (master → FPGA → device1): master writing to 0x48 and 0x49.
  + Downstream SCL/SDA (FPGA → device2): translator issuing START/address/data to 0x48.
* Console logs showing:
  + Bytes received by device1 and device2
  + When translator performs address remap
  + ACK/NACK status for each byte
* A read example where master reads value 0xAB and receives it via translator.

**12 — Resource report & synthesis notes**

**Tools:** Xilinx Vivado / Intel Quartus / Lattice Diamond.

Synthesis steps (Vivado example):

1. Create new Vivado project, add i2c\_translator.v and constraints (IO pins if using board).
2. Run “Synthesis” and “Implementation” (if targeting a board).
3. Generate reports:
   * utilization\_placed.rpt / utilization\_synth.rpt (LUTs, FFs, BRAM, IO)
   * timing\_summary\_routed.rpt (slack, worst paths)
   * IO constraints: ensure SDA/SCL pins configured as open-drain or use IBUF/OBUF with appropriate drive.
4. Save screenshots or PDFs of the utilization and timing summary for the deliverable.

What to include in the deliverable:

* Device family and part number used
* Resource utilization (LUT/FF/BRAM/IO)
* Clock constraints and achieved fmax
* Timing summary (worst negative slack, path details)

**13 — Documentation structure to add to GitHub**

Include README.md and REPORT.pdf with:

**README.md**

* Project overview and diagram (use your screenshot or create a neat block diagram).
* How to run simulation (EDAPlayground link + local steps).
* Files in repository and their purpose.
* How to synthesize (Vivado / Quartus steps / commands).

**REPORT.pdf or doc**

* Architecture overview (modules & data flow).
* FSM diagrams (Upstream slave FSM, Downstream master FSM, Proxy FSM). A small ASCII/state table is fine.
* Address translation explanation (virtual→real mapping & configuration method).
* Testbench details and test cases.
* Resource usage summary and timing results.
* Design challenges encountered and how you solved them (open-drain behavior, ACK timing, repeated START handling).
* Future improvements / bonus features (multi-byte transactions, multiple mappings, runtime config over UART).

**14 — Bonus features you can add (nice for evaluation)**

* **Runtime-configurable mapping**: implement an I²C configuration address, or a UART/AXI-lite port to change mapping at run time. Mark as “bonus” in docs.
* **Support for multiple mappings** (N entries): allow remapping of multiple devices (store a small table).
* **Multi-byte buffering**: implement a FIFO for downstream data to support burst transfers.
* **Clock stretching**: implement and respect upstream master clock stretching if needed.

**15 — 15-day work schedule / checklist (practical)**

Day 1 — 2: Design & module interface; write state diagrams.  
Day 3 — 5: Implement upstream slave FSM and unit test (simulate address / ack behavior).  
Day 6 — 8: Implement downstream master FSM (byte send/receive + SCL generator).  
Day 9 — 10: Implement proxy logic + address mapping. Connect upstream & downstream; test simple write proxy.  
Day 11: Implement read proxy and repeated START behavior.  
Day 12: Build testbench including Device1 and Device2 models; run full testcases.  
Day 13: Run simulation on EDAPlayground, produce waveform and fix bugs.  
Day 14: Synthesize in Vivado/Quartus and capture resource/timing reports.  
Day 15: Prepare documentation, package repo, screen-record coding & testing, add collaborator, final checks.

Checklist (before submission):

* Verilog files in repository and each file with header comment explaining purpose.
* Testbench with multiple assertions and waveform dump.
* EDAPlayground public link saved in README.
* Resource/timing reports included.
* README and REPORT with diagrams and FSM explanations.
* Screen recording uploaded and link available.
* Add recruit-vicharak as a collaborator to private repo.

**16 — Small helpful code skeletons & pseudocode**

Top-level module skeleton:

module i2c\_translator #(

parameter SYSCLK\_HZ = 50\_000\_000

)(

input wire clk,

input wire rst\_n,

// upstream

input wire SCL\_up,

inout wire SDA\_up,

// downstream

output reg SCL\_down,

inout wire SDA\_down,

// config

input wire [6:0] cfg\_orig\_addr,

input wire [6:0] cfg\_virtual\_addr,

input wire cfg\_enable

);

// instantiate upstream slave FSM

// instantiate downstream master FSM

// instantiate bridge logic

endmodule

Upstream bit receive pseudocode:

// sample on SCL rising:

if (scl\_rising && state == U\_ADDR\_RX) begin

shift <= {shift[6:0], SDA\_up}; // MSB first depending on chosen orientation

bit\_cnt <= bit\_cnt + 1;

if (bit\_cnt == 7) // 8 bits done

// parse addr/rw and go to ACK state

end

Downstream send\_byte pseudocode:

task send\_byte(input [7:0] b, output reg ack);

for (i=7; i>=0; i--) begin

// while SCL low: set SDA (drive low for 0, release for 1)

// wait half\_period

SCL\_down <= 1'b1;

// wait half\_period

SCL\_down <= 1'b0;

end

// sample ACK: release SDA, raise SCL, read SDA, lower SCL

endtask

**17 — Testing & debugging tips**

* Start with a simple single-byte write test between master and device2 through translator, log everything.
* Use waveform to verify ACK clocks align with 9th bit.
* Use $display to log: upstream addr received, matched? translated addr sent, bytes forwarded, ack status.
* If a transaction fails, check whether SDA is being driven at the correct times (only drive low — never drive high).

**18 — Important note about AI / assignment policy**

You mentioned the employer’s rule: *“Do not use AI tool to complete the code submission...”* — **please follow that**. I provided architecture, state machines, pseudocode and skeletons to guide your work. To satisfy their rule:

* **Implement the actual Verilog files in your own editor** using the plan above; add your own comments and style.
* If you used any snippets provided here, rewrite variable names and comments in your voice and test thoroughly.
* Add a note in README explaining your testing approach and the simulation link.

**19 — If you want, next steps I can do now**

I can produce **one** of the following right now (choose one and I’ll provide it immediately):

* A fully commented **detailed FSM diagram and ASCII state tables** (ready to paste into your REPORT).
* A **complete testbench template** (no device models) you can fill in.
* A **detailed README + REPORT skeleton** in Markdown that you can drop in your repo.