# CSE 331 — Computer Organization

## Project 4: Sequential Multiplier (Verilog + Quartus)

Due: CSE 331 Final Exam Date
Tools: Quartus Prime, ModelSim/Questa (or equivalent), Verilog-2001

---

## 1) Overview

Design a parameterized, sequential (multi-cycle) multiplier using a clean separation of Control Unit and Datapath. The Control Unit must be written using exactly three always blocks (classic FSMD style). Favor structural/dataflow coding and continuous assignments (`assign`) for combinational logic. Use behavioral code only for sequential state (registers, counters, memories if used). Provide your own self-checking testbench.

Target functionality: unsigned multiplication of two N-bit operands producing a 2N-bit product using an iterative shift-and-add (or equivalent) algorithm.

---

## 2) Functional Requirements

- Width: parameterizable `N` (default 32)
- Operation: unsigned multiply of `multiplicand[N-1:0]` and `multiplier[N-1:0]`
- Result: `product[2*N-1:0]`
- Latency: bounded by N iterations (plus optional setup/finish cycles)
- Interface & Handshake:
  - Inputs: `start` (1-cycle pulse), `multiplicand`, `multiplier`
  - Outputs: `busy` (high while computing), `done` (1-cycle pulse when product valid)
  - New `start` accepted only when `busy==0`
- Reset: synchronous (document polarity)
- Clock: single clock domain
- Optional extensions (not required):
  - Signed two's-complement mode with control bit
  - Early termination (e.g., zero detection)
  - Booth or other optimized algorithm

Keep the base scope focused on a straightforward sequential algorithm.

---

## 3) Design Constraints & Guidance

- Must separate Control Unit and Datapath into distinct modules; additional hierarchy is welcome.
- Control Unit must use exactly 3 always blocks:
  1. State register (sequential, posedge clock)
  2. Next-state logic (combinational)
  3. Output/control logic (combinational)

- Use `assign` for combinational paths, muxes, decoders, compares, and simple adders where practical.
- Behavioral state elements (registers, counters) are expected in the datapath.
- Clearly document the FSM states (e.g., IDLE, LOAD, RUN, DONE).
- Recommend a bit counter driving loop termination.
- Word-align shifting and add-accumulate in the datapath; do not place algorithmic behavior in the control beyond signal sequencing.

---

## 4) High-Level Architecture

- Datapath (examples; adapt as needed)
  - Accumulator/Product register (2N bits)
  - Multiplicand register (N bits)
  - Multiplier register (N bits)
  - 2N-bit adder
  - Shifters (left/right as per chosen algorithm)
  - N-bit down-counter
  - Comparators/zero-detect
  - Muxes for load/hold/shift/select paths
- Control Unit (FSM)
  - Interprets `start`, `busy`, `done`
  - Sequences load/clear/add/shift/dec signals
  - Asserts `busy` during RUN, pulses `done` at completion
  - Guards against re-start during active computation

Document your chosen data movement (e.g., shift multiplier right and conditionally add multiplicand into upper product half).

---

## 5) Module Skeletons (ports only; fill internals yourself)

Favor `assign` for combinational; use behavioral only for sequential state.

```
// Top-level multiplier with simple handshake
module seq_multiplier #(
    parameter N = 32
)(
    input  wire             clk,
    input  wire             rst,         // synchronous, document polarity
    input  wire             start,       // 1-cycle pulse to start
    input  wire [N-1:0]     multiplicand,
    input  wire [N-1:0]     multiplier,
    output wire [2*N-1:0]   product,
    output wire             busy,
    output wire             done         // 1-cycle pulse when product valid
);
    // TODO: instantiate datapath and control; wire control signals
endmodule
```

```
// Datapath: all arithmetic/shift state; no FSM here
module mul_datapath #(
    parameter N = 32
)(
    input  wire             clk,
    input  wire             rst,

    // Control signals (from CU)
    input  wire             ld_operands,   // load multiplicand/multiplier
    input  wire             clr_product,   // clear accumulator/product
    input  wire             add_enable,    // conditionally add multiplicand
    input  wire             shift_enable,  // shift registers
    input  wire             cnt_load,      // initialize iteration counter
    input  wire             cnt_dec,       // decrement counter
    input  wire             sel_add_src,   // example mux select (if needed)

    // Status back to CU
    output wire             cnt_zero,      // loop termination
    output wire             lsb_is_one,    // from multiplier for add decision

    // External interface
    input  wire [N-1:0]     multiplicand_in,
    input  wire [N-1:0]     multiplier_in,
    output wire [2*N-1:0]   product_out
);
    // TODO: registers, adders, shifters, counter; minimal behavioral for state
endmodule
```

```
// Control Unit: exactly 3 always blocks (state, next-state, outputs)
module mul_control (
    input  wire clk,
    input  wire rst,
    input  wire start,
    input  wire cnt_zero,
    input  wire lsb_is_one,
    output reg  ld_operands,
    output reg  clr_product,
    output reg  add_enable,
    output reg  shift_enable,
    output reg  cnt_load,
    output reg  cnt_dec,
    output reg  sel_add_src,   // if used
    output reg  busy,
    output reg  done
);
    // TODO:
    // 1) always @(posedge clk) for state register
    // 2) always @* for next-state logic
    // 3) always @* for output logic
endmodule
```

```
// Example: parameterized counter (behavioral state, comb outputs)
module down_counter #(
    parameter W = 6
)(
    input  wire        clk,
    input  wire        rst,
    input  wire        load,
    input  wire        dec,
    input  wire [W-1:0] init_val,
    output wire        is_zero,
    output wire [W-1:0] value
);
    // TODO
endmodule
```

```
// Example: simple 2N adder (combinational)
module add2N #(
    parameter N = 32
)(
    input  wire [2*N-1:0] a,
    input  wire [2*N-1:0] b,
    output wire [2*N-1:0] y
);
    // TODO: assign y = a + b;
endmodule
```

---

## 6) Handshake & Timing

- IDLE: `busy=0`, accept `start`
- LOAD: latch inputs, clear product, load counter (= N)
- RUN: iterate N times
    - Use `lsb_is_one` to gate `add_enable`
    - Shift after add as per your algorithm
    - Decrement counter each iteration
- DONE: pulse `done` for one cycle, deassert `busy`, return to IDLE

Ensure `start` while `busy==1` is ignored. Define and document any edge cases.

---

## 7) Testbench (self-checking)

- Generate clock and synchronous reset
- Drive a series of test vectors, including:
    - Directed: 0, 1, max values, powers of two, random
    - Mixed small/large operands

- For each test:
  - Apply inputs, pulse `start`
  - Wait for `done`
  - Compare `product` against a reference model (`$unsigned(a)*$unsigned(b)`)
  - Use `$error/$fatal` on mismatches; print PASS/FAIL summary
- Add timing guards (timeout) to catch stuck FSMs

Skeleton:

```
`timescale 1ns/1ps
module seq_multiplier_tb;
    localparam N = 32;
    reg  clk = 0;
    reg  rst = 1;
    reg  start = 0;
    reg  [N-1:0] multiplicand, multiplier;
    wire [2*N-1:0] product;
    wire busy, done;

    always #5 clk = ~clk;

    seq_multiplier #(.N(N)) dut (
        .clk(clk), .rst(rst), .start(start),
        .multiplicand(multiplicand), .multiplier(multiplier),
        .product(product), .busy(busy), .done(done)
    );

    initial begin
        repeat (2) @(posedge clk);
        rst = 0;

        // TODO: drive tests, pulse start, wait for done, check product
        // Use tasks for reuse; add timeout protection
        $display("TB FINISHED");
        $finish;
    end
endmodule
```

---

## 8) Build & Run

- Simulation
  - Compile all Verilog files
  - Run `seq_multiplier_tb`; ensure PASS/FAIL reporting
- Synthesis (Quartus)
  - Create project; top module: `seq_multiplier`
  - Add pin constraints only if mapping to I/O
  - Confirm the design meets timing for target clock