Section 2 – Sequential Logic

Objectives

- Implement following sequential logic modules:
  1. **Register (32-bit)**
  2. **Counter (up/down counter)**
  3. **Shift Register (left, and bidirectional)**
  4. **Array Multiplier**
  5. **Adder Tree**
- Develop an **assertion-based and layered testbench** to verify correct sequential behavior.

Tasks

1. Implement a 32-bit Register

Create a synchronous register module (`reg32.sv`) with:

- **Inputs**:
  - `clk`, `rst_n` (active low),
  - `load` (enable signal),
  - `d[31:0]`
- **Output**:
  - `q[31:0]`

Functional Requirements:

- On reset → `q = 0`.
- On rising edge of `clk`:
  - If `load = 1`, then `q <= d`.
  - Otherwise, `q` holds its previous value.

---

2. Implement a Counter

Design a configurable counter (`counter.sv`).

- **Up/down counter**

Inputs:

- `clk`, `rst_n`
- `en` (enable)
- `up_dn`

Outputs:

- `count[N:0]` (default 8-bit or 16-bit as assigned)

Functional Requirements:

- Counter resets to 0 on reset.
- On each rising edge:
  - If `en = 1`, increment or decrement depending on `up_dn`.
  - If `en = 0`, hold value.

---

3. Implement a Shift Register

Create a serial shift register (`shift_reg.sv`).

- **Left shift or Right shift**
- **Bidirectional shift**

Inputs:

- `clk`, `rst_n`
- `shift_en`
- `dir` (0 = left, 1 = right; only if bidirectional)
- `d_in` (serial input)

Outputs:

- `q_out[N:0]` (default 8-bit or 16-bit)

Functional Requirements:

- Reset clears register.
- On rising edge:
  - If `shift_en = 1`, shift left or right and insert `d_in`.
  - If disabled, hold previous value.

---

4. Implement an array multiplier

A four-bit circuit that implements P=A×B is illustrated in Figure 1. Because of its regular structure, this type of multiplier circuit is called an array multiplier. In each row of the multiplier AND gates are used to produce the summands, and full adder modules are used to generate the required sums.

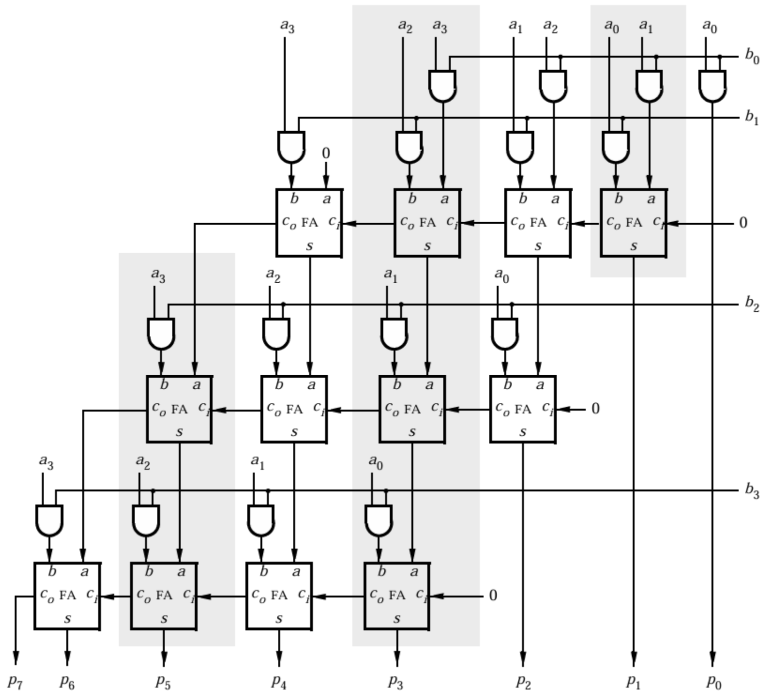

Figure 1 Array multiplier circuit

At a higher level, a row of full adders functions as an n-bit adder and the array multiplier circuit can be represented as shown in Figure 2.

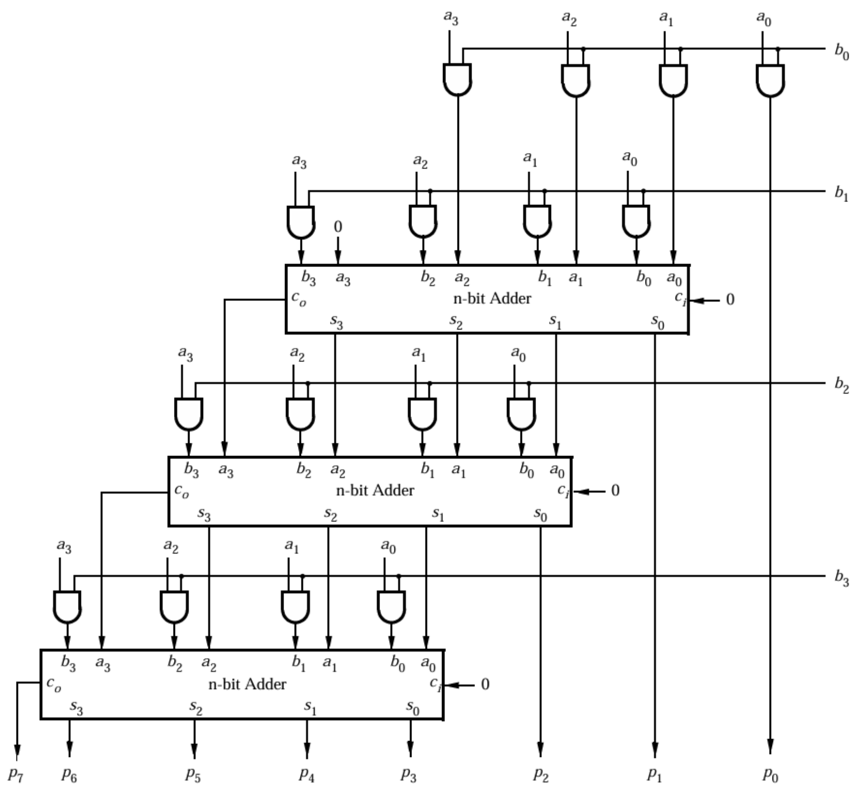
a_3
a_2
a_1
a_0
b_0
b_1
b_2
b_3
0
b_3 a_3 b_2 a_2 b_1 a_1 b_0 a_0
c_o
n-bit Adder
c_i
s_3 s_2 s_1 s_0
p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0

Figure 2

Each n-bit adder adds a shifted version of A for a given row and the partial product of the row above. Abstracting the multiplier circuit as a sequence of additions allows us to build larger multipliers. The multiplier should consist of n-bit adders arranged in a structure shown in Figure 2. Use this approach to implement an 8 x 8 multiplier circuit with registered inputs and outputs, as shown in Figure 3.

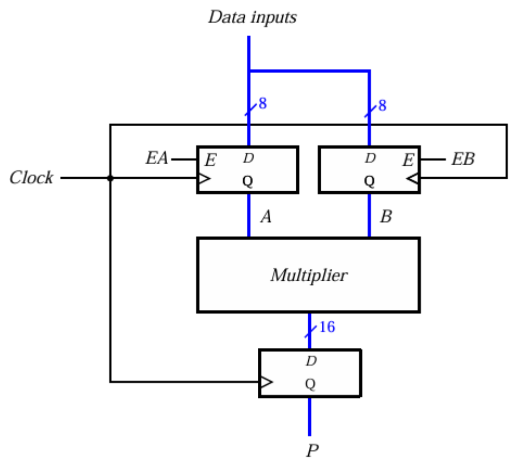

Figure 3

5. Implement an 8-bit multiplier using Adder Tree

Part 4 showed how to implement multiplication A × B as a sequence of additions, by accumulating the shifted versions of A one row at a time. Another way to implement this circuit is to perform addition using an adder tree. An adder tree is a method of adding several numbers together in a parallel fashion. This idea is illustrated in Figure 4. In the figure, numbers A, B, C, D, E, F, G, and H are added together in parallel. The addition A + B happens simultaneously with C +D, E +F and G+H. The result of these operations are then added in parallel again, until the final sum P is computed

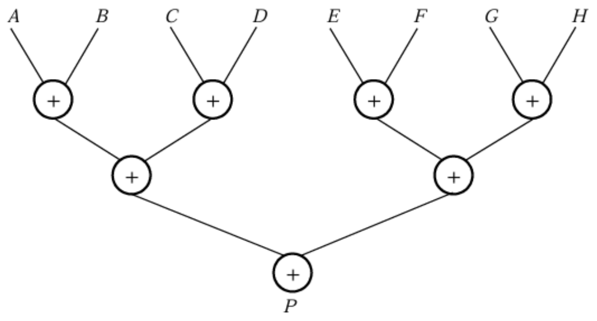

Figure 4- Adder Tree

In this part you are to implement an 8 x 8 multiplier circuit by using the adder-tree approach. Inputs A and B, as well as the output P should be registered as in Part 4.

---

6. Write an Assertion-Based Testbench

Create a testbench `reg32_tb.sv` that must include assertions that verify:

- Reset behavior:
  `assert property (reset |-> q == 0);`
- Load behavior:
  If `load` is high at `posedge clk`, then `q` equals `d`.
- Hold behavior:
  If `load = 0`, `q` remains unchanged.

---

7. Write Layered Testbench

Create testbenches for counter, shift register and multiplier.

Deliverables

- RTL files:
- Testbench files
- Simulation logs or waveform screenshots
- Optional: Assertion failure examples (if testing negative cases intentionally)