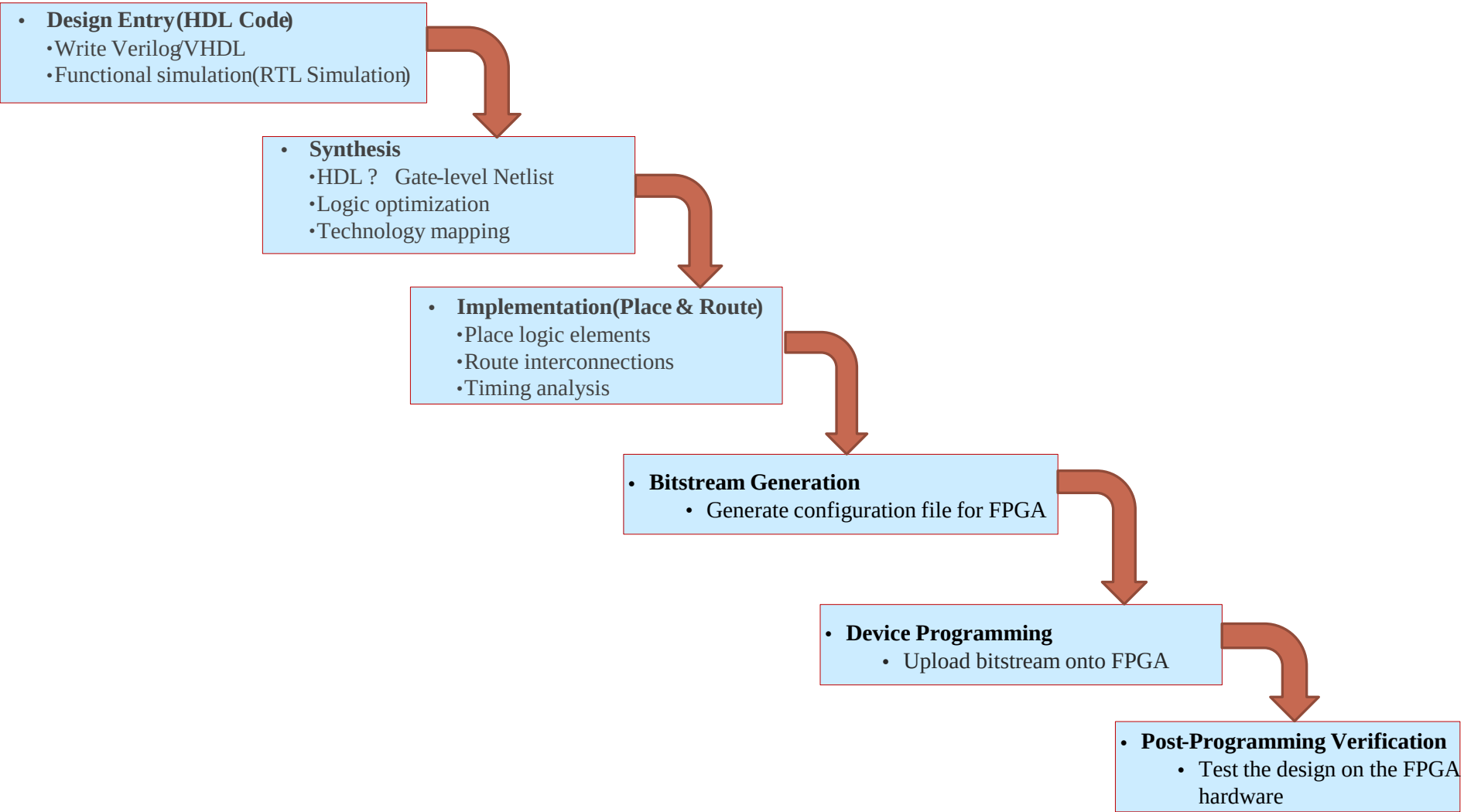
MODULE 5
SYNTHESIS AND TIMING ANALYSIS

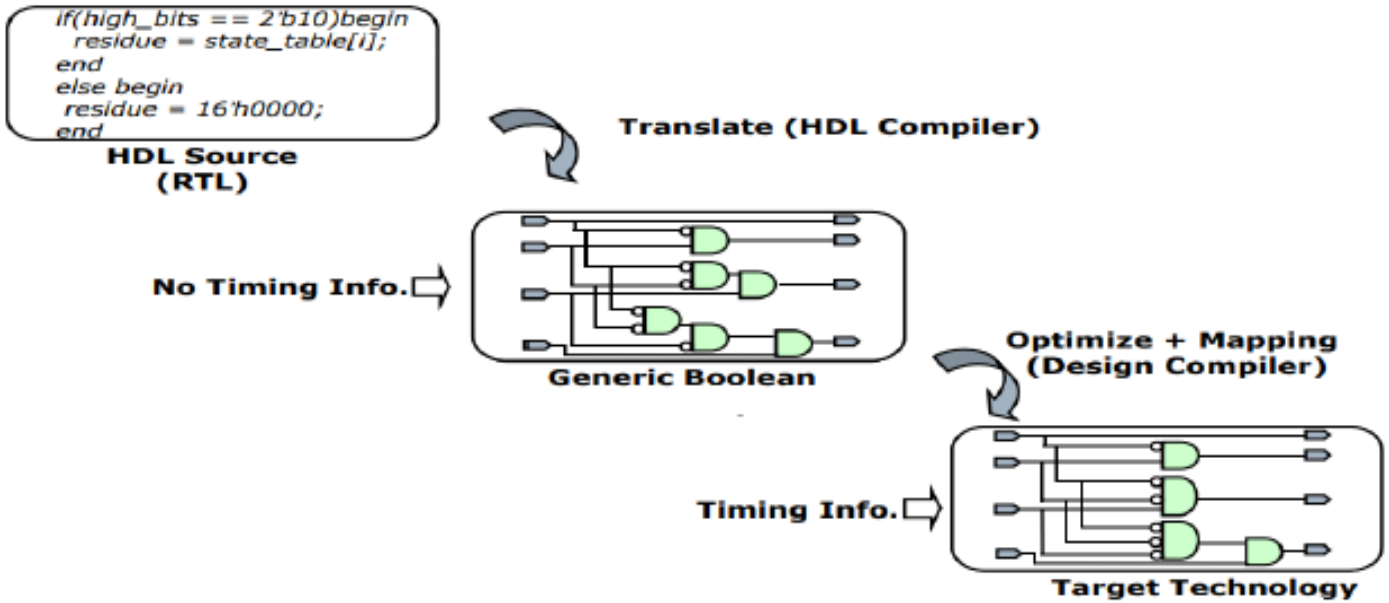
CONTENTS

- Synthesis
- Optimization of Speed:
  - Introduction
  - Strategies for Timing Improvement
- Optimization of Area
- Optimization of power


• Design Entry (HDL Code)
•Write Verilog/VHDL
•Functional simulation(RTL Simulation)
• Synthesis
•HDL ? Gate-level Netlist
•Logic optimization
•Technology mapping
• Implementation(Place & Route)
•Place logic elements
•Route interconnections
•Timing analysis
• Bitstream Generation
• Generate configuration file for FPGA
• Device Programming
• Upload bitstream onto FPGA
• Post-Programming Verification
• Test the design on the FPGA hardware

SYNTHESIS

- The synthesis flow for FPGA involves multiple steps that convert a high-level hardware description (e.g., written in Verilog or VHDL) into a bitstream file that can be loaded onto an FPGA.
- Synthesis = Translation + Optimization + Mapping



INTRODUCTION TO SPEED OPTIMIZATION IN DIGITAL DESIGN

- **Speed optimization** in digital design refers to the techniques and methodologies used to improve the performance of a circuit in terms of **execution speed** or **throughput**.
- The primary goal is to minimize the **critical path delay**, allowing the circuit to operate at higher clock frequencies and execute tasks faster.
- This is especially crucial for applications where high performance and fast data processing are required, such as in **processors**, **DSP systems**, and **high-speed communication circuits**.
- In digital systems, speed is often dictated by the **clock period**, which is determined by the longest signal propagation delay through the circuit (also known as the **critical path**).
- The clock frequency is inversely proportional to this clock period, meaning that reducing the critical path delay increases the clock frequency, and thereby, the system's speed.

TIMING OPTIMIZATION

- Timing optimization ensures that the circuit meets its performance specifications, such as clock speed, by reducing the critical path delay.
  - **Pipelining**: Break long combinational paths into smaller stages by adding registers. This helps reduce the critical path and increase clock frequency.
  - **Retiming**: Move registers around in a circuit to balance the delays and improve timing.
  - **Logic Duplication**: Duplicate critical components to reduce fan-out and improve timing.
  - **Gate Sizing for Timing**: Increase the size of gates on critical paths to drive more load and reduce delays.
  - **Clock Tree Optimization**: Use clock gating, clock tree balancing, and optimized clock distribution to reduce clock skew and improve timing.
  - **Delay Buffers**: Add delay buffers to align timing in non-critical paths, balancing delays in the circuit.
  - **Critical Path Optimization**: Focus on reducing the delay of critical paths, either by restructuring the logic or reducing the load on critical nets.

PIPELINING: BREAK LONG COMBINATIONAL PATHS INTO SMALLER STAGES BY ADDING REGISTERS. THIS HELPS REDUCE THE CRITICAL PATH AND INCREASE CLOCK FREQUENCY

Before Optimization:
Without pipelining, the data flows through a long combinational path, limiting the clock speed.

```
module no_pipeline(input clk, rst,
input [7:0] A, B, C, D, output reg [7:0] Y);
   always @(posedge clk or posedge rst)
begin
      if (rst)
         Y <= 8'b0;
      else
         Y <= (A + B) * (C + D);
                  // Long combinational path
   end
endmodule
```

After Optimization (With Pipelining):
In pipelining, intermediate results are stored in registers to break the long combinational path into smaller parts.

```
module pipeline(input clk, rst, input
[7:0] A, B, C, D, output reg [7:0] Y);
   reg [7:0] sum1, sum2;

   // Stage 1: Calculate the sums
   always @(posedge clk or posedge
rst) begin
      if (rst) begin
         sum1 <= 8'b0;
         sum2 <= 8'b0;
      end else begin
         sum1 <= A + B;
         sum2 <= C + D;
      end
   end
```

```
 // Stage 2: Calculate the
product
   always @(posedge clk
or posedge rst) begin
      if (rst)
         Y <= 8'b0;
      else
         Y <= sum1 *
sum2;
   end
endmodule
```

RETIMING: MOVE REGISTERS AROUND IN A CIRCUIT TO BALANCE THE DELAYS AND IMPROVE TIMING.

Without retiming, the logic may have imbalanced delays, resulting in timing violations.

```
module no_retiming(input clk, rst,
input [7:0] A, B, C, output reg [7:0] Y);
   reg [7:0] mid;

   always @(posedge clk or posedge rst)
begin
      if (rst) begin
         mid <= 8'b0;
         Y <= 8'b0;
      end else begin
         mid <= (A * B);  // Critical path is
here
         Y <= mid + C;
      end
   end    endmodule
```

With Retiming, registers are moved within the combinational logic to balance the delays and improve timing.

```
module retiming(input clk, rst, input [7:0] A, B, C,
output reg [7:0] Y);
   reg [7:0] mid, mid_reg;

   always @(posedge clk or posedge rst) begin
      if (rst) begin
         mid_reg <= 8'b0;
         Y <= 8'b0;
      end else begin
         mid <= A * B;
         mid_reg <= mid;
                  // Move register to balance the critical
path
         Y <= mid_reg + C;
      end    end   endmodule
```

Delay Buffers: Add delay buffers to align timing in non-critical paths, balancing delays in the circuit.

Before optimization

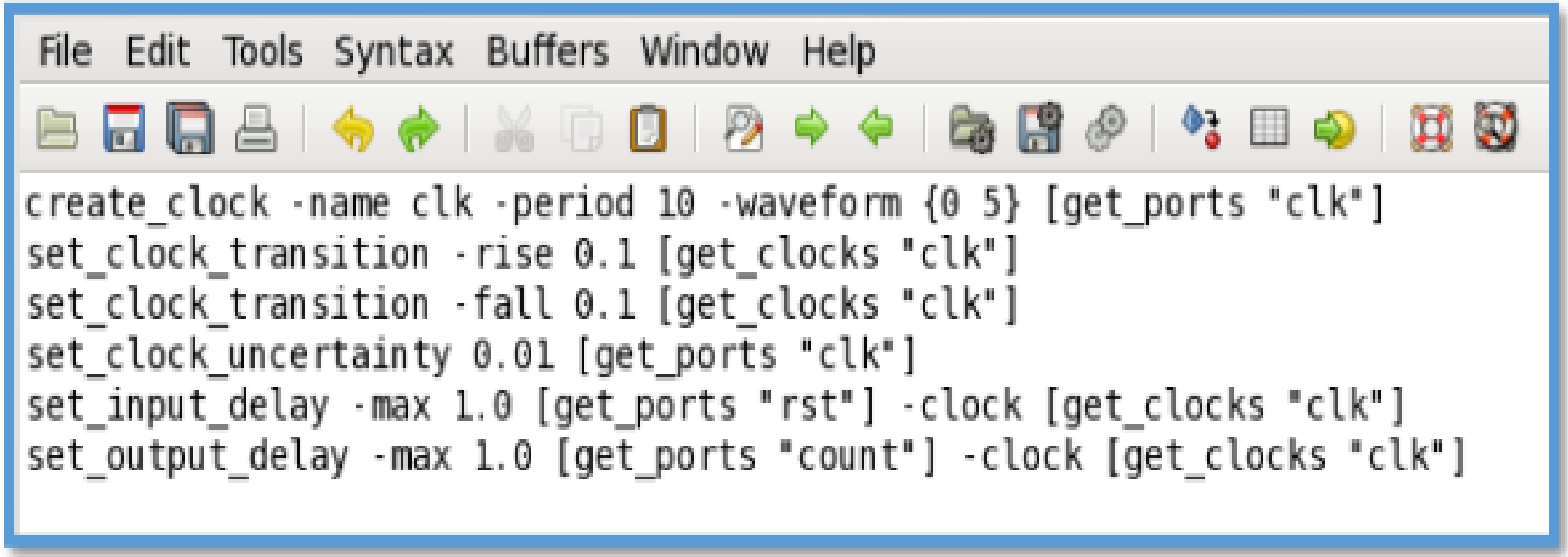
```
module no_buffer_insertion(
    input clk,
    input A,
    output reg Y
);
    always @(posedge clk) begin
        Y <= A;
            // High load on A, causing signal
delay
    end
endmodule
```

After optimization

```
module buffer_insertion(
    input clk,
    input A,
    output reg Y
);
    reg A_buffer;  // Insert buffer to improve signal
propagation

    always @(posedge clk) begin
        A_buffer <= A;
                    // Buffer to reduce delay caused by
high load
        Y <= A_buffer;
    end
endmodule
```

TIMING CONSTRAINTS (SDC FILE)



TIMING CONSTRAINTS (SDC FILE)

- **create_clock –name –period 10 –waveform {0 5} {get_port “clk”}**

  This command will define clock with period 10ns and 50% duty cycle and signal is high in the first half.

- **set_clock_transition –rise/fall**

  This command defines the transition delay for clock.

- **set_clock_uncertainty**

  This command will set the uncertainty due to (clock skew and jitter).

- **set_input/output_delay**

  This command will specify the input and output delay used for timing slack calculations.

AREA OPTIMIZATION

- **The goal here is to reduce the chip size or the number of transistors, which directly impacts the cost and yield of integrated circuits.**
- **Logic Minimization:** Simplify logic using techniques like Boolean algebra or Karnaugh maps (K-maps) to reduce the number of logic gates.
- **Resource Sharing:** Reuse functional units like adders or multipliers across different parts of the design.
- **Gate Sizing:** Use smaller gates wherever possible, or adjust gate sizes based on the criticality of the timing paths.
- **Multi-bit Cell Usage:** Combine multiple instances of single-bit cells into multi-bit cells to reduce overall area and interconnect complexity.
- **Cell Merging:** Combine smaller logic gates into more complex gates if they serve a similar function.
- **ROM/RAM Optimization:** Use memory optimizations to reduce the footprint of data storage.
- **Use of Finite State Machines (FSM) Encoding:** Optimize FSM using gray code encoding or one-hot encoding depending on the size and requirements.

RESOURCE SHARING

```
module no_resource_sharing
    ( input [7:0] a, b, c, d, output [15:0] result );
    wire [15:0] mult1, mult2;
    assign mult1 = a * b;
    assign mult2 = c * d;
    assign result = mult1 + mult2;
endmodule
```

• **Explanation:**

Here, two separate multipliers (mult1 and mult2) are used to compute a * b and c * d. This requires two multiplier units in hardware, which increases the area.

RESOURCE SHARING

```
module resource_sharing( input [7:0] a, b, c, d, output [15:0] result);
 reg [15:0] mult;
 always @(*) begin
 mult = a * b;
 mult = mult + (c * d); // Same multiplier resource reused
 end
 assign result = mult;
endmodule
```

• **Explanation:** The same multiplier is reused for both operations. First, it computes a * b, then it is reused for c * d. This reduces the number of multipliers needed to just one, saving hardware resources. However, this may increase computation time since the operations are now sequential, but it optimizes the area by reducing hardware.

OPERATOR STRENGTH REDUCTION

```
module no_operator_reduction(
 input [7:0] a,
 output [15:0] result
);
 assign result = a * 4; // Multiplication
endmodule
```

• **Explanation:** Multiplication by 4 requires a dedicated multiplier circuit in hardware, which consumes more logic resources (e.g., multiplier blocks on an FPGA).

OPERATOR STRENGTH REDUCTION

```
module operator_reduction(
 input [7:0] a,
 output [15:0] result
);
 assign result = a << 2; // Shift left by 2 is equivalent to multiplication by 4
endmodule
```

• **Explanation:** Multiplying by 4 is replaced by a bitwise left shift (<< 2). A shift operation is much simpler and requires fewer logic gates compared to multiplication. This reduces the logic gate count and improves efficiency without changing the functionality.

MULTI-BIT CELL USAGE

```
module multi_bit_before(
        input clk, rst, input [1:0] data_in,
        output reg Q0, Q1);
    always @(posedge clk or posedge rst)
begin
        if (rst) begin
            Q0 <= 1'b0;
            Q1 <= 1'b0;
        end else begin
            Q0 <= data_in[0];
            Q1 <= data_in[1];
        end
    end   endmodule
```

```
module multi_bit_after(
        input clk, rst,
        input [1:0] data_in,
        output reg [1:0] Q);
    always @(posedge clk or posedge rst)
 begin
        if (rst)
            Q <= 2'b00;
        else
            Q <= data_in;
    end
endmodule
//This reduces area by consolidating the flip-flops.
```

FSM ENCODING (ONE-HOT ENCODING)

```
module fsm_binary(input clk, rst, input in, output reg [1:0] state);
   parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
   always @(posedge clk or posedge rst) begin
      if (rst)
         state <= S0;
      else begin
         case(state)
            S0: state <= in ? S1 : S0;
            S1: state <= in ? S2 : S0;
            S2: state <= in ? S3 : S0;
            S3: state <= in ? S0 : S1;
            default: state <= S0;
         endcase
      end
   end
endmodule
```

Explanation:

A 2-bit binary encoding is used for the finite state machine (FSM), which involves more complex logic for state transitions since each state requires a different combination of bits.

FSM ENCODING (ONE-HOT ENCODING)

```
module fsm_binary(input clk, rst, input in, output reg [3:0] state);
   parameter S0 = 4'b0001, S1 = 4'b0010, S2 = 4'b0100, S3 = 4'b1000;
   always @(posedge clk or posedge rst) begin
      if (rst)
         state <= S0;
      else begin
         case(state)
            S0: state <= in ? S1 : S0;
            S1: state <= in ? S2 : S0;
            S2: state <= in ? S3 : S0;
            S3: state <= in ? S0 : S1;
            default: state <= S0;
         endcase
      end
   end
endmodule
```

Explanation:

One-hot encoding uses more flip-flops but can reduce the complexity of the combinational logic, depending on the size of the FSM. If the logic becomes simpler, the area can be reduced despite having more state bits.

POWER OPTIMIZATION

- **Power consumption can be dynamic (due to switching activity) or static (due to leakage). Strategies aim to minimize both types.**
- **Clock Gating:** Disable the clock signal for idle portions of the circuit, reducing unnecessary switching.
- **Power Gating:** Shut down parts of the circuit that are not in use to save static power (leakage).
- **Voltage Scaling:** Lower supply voltage (dynamic voltage scaling) reduces dynamic power consumption, though it may also require lowering the clock frequency.
- **Multi-Voltage Design:** Use different voltage domains where critical circuits run at higher voltages and non-critical parts run at lower voltages.
- **Dynamic Frequency Scaling (DFS):** Adjust the clock frequency dynamically based on the workload to reduce power consumption when the performance requirement is lower.

POWER OPTIMIZATION

- **Low-power Libraries:** Use low-leakage cells and low-power libraries to reduce both dynamic and leakage power.
- **Sleep Transistors**: Use high-threshold voltage transistors to reduce leakage current in idle circuits.
- **Adiabatic Logic:** Recover some energy during switching to reduce overall power consumption.
- **Capacitance Minimization:** Minimize parasitic capacitances in interconnects and devices to reduce switching power.
- **Activity Reduction:** Optimize switching activity by minimizing unnecessary toggling of signals.

CLOCK GATING

- **Concept:** Disable the clock for certain parts of the circuit when they are not in use, reducing dynamic power by preventing unnecessary toggling.
- **Before Optimization:** Without clock gating, the clock is continuously running, consuming power even when the logic is idle.

```
module no_clock_gating(
    input clk, rst, input [7:0] data_in, output reg [7:0] data_out);
always @(posedge clk or posedge rst) begin
    if (rst)
        data_out <= 8'b0;
    else
        data_out <= data_in;  // Always updates on every clock cycle
end endmodule
```

CLOCK GATING

```
module clock_gating(
        input clk, rst, enable,
        input [7:0] data_in,
        output reg [7:0] data_out);
    wire gated_clk;

    // Clock gating logic
    assign gated_clk = clk & enable;

    always @(posedge gated_clk or posedge rst) begin
        if (rst)
            data_out <= 8'b0;
        else
            data_out <= data_in;  // Updates only when the enable signal is high
    end
endmodule
```

POWER GATING

- **Concept:** Shut down sections of a circuit that are not in use to save static power (leakage). This can be done by turning off the power supply to unused blocks.
- **Before Optimization:** Without power gating, the unused blocks consume leakage power even when idle.

```
module no_power_gating(input clk, rst, input [7:0] data_in, output reg [7:0] data_out);
   always @(posedge clk or posedge rst) begin
      if (rst)
         data_out <= 8'b0;
      else
         data_out <= data_in;  // Power is always consumed, even when the block is not needed
   end
endmodule
```

POWER GATING

```
module power_gating(
        input clk, rst, enable, input [7:0] data_in, output reg [7:0]
data_out);
   reg powered_on;

   always @(posedge clk or posedge rst) begin
      if (rst)
         powered_on <= 1'b0;
      else
         powered_on <= enable;  // Turn the block on only when enable
is high
   end
```

```
   always @(posedge clk or posedge rst) begin
      if (rst)
         data_out <= 8'b0;
      else if (powered_on)
         data_out <= data_in;   // Block is active only when
powered_on is high
   end
endmodule
```

OPTIMIZED FSM TRANSITIONS

```
module fsm_no_opt(
        input clk, input rst,
        output reg [1:0] state);
always @(posedge clk or posedge rst)
begin
        if (rst)
                state <= 2'b00;
        else
                state <= state + 1; // Unnecessary state transitions
end
endmodule
```

Explanation: In this design, the FSM transitions between states in a non-optimized manner, leading to higher switching activity and power consumption.

OPTIMIZED FSM TRANSITIONS

```
module fsm_optimized(
input clk, input rst, output reg [1:0] state);
 always @(posedge clk or posedge rst)
begin
 if (rst)
 state <= 2'b00;
 else if (state == 2'b11)
 state <= 2'b00; // Avoid unnecessary transitions
 else
 state <= state + 1;
 end
endmodule
```

Explanation: : In this optimized FSM, transitions are minimized by checking the current state before transitioning. Fewer state changes reduce switching activity and, consequently, power consumption.