Kogge-Stone Adder: Working and Algorithm

Introduction

The Kogge-Stone adder is a parallel prefix adder that performs binary addition with logarithmic depth, making it one of the fastest adder architectures available. It trades area for speed by using more hardware to achieve minimal delay, making it ideal for high-performance computing applications.

Working Principle

The Kogge-Stone adder is based on the parallel prefix computation of carry signals. Instead of waiting for carries to ripple through sequentially like in a ripple carry adder, it computes all carries in parallel using a tree-like structure.

Key Concepts

Generate (G): $G_i = A_i \cdot B_i$

- Position i generates a carry regardless of the input carry
- This occurs when both input bits are 1

Propagate (P): $P_i = A_i \oplus B_i$

- Position i propagates a carry from the previous position
- This occurs when exactly one input bit is 1

Carry: $C_i = G_i + P_i \cdot C_{i-1}$

- The carry output depends on either generating a carry or propagating the previous carry

The adder computes carry signals by combining generate and propagate signals across different spans using the parallel prefix operation.

Algorithm

The Kogge-Stone algorithm operates in three main phases:

Phase 1: Initialization

For each bit position i (from 0 to n-1):

- Calculate initial Generate: G[i,i] = Ai · Bi
- Calculate initial Propagate: P[i,i] = Ai ⊕ Bi

This creates the base level of the prefix tree with individual bit positions.

Phase 2: Parallel Prefix Computation

For each level k from 1 to $\lceil \log_2(n) \rceil$: For each position i where the operation is applicable: - G[i,j] = G[i,k] + P[i,k] · G[k-1,j] - P[i,j] = P[i,k] · P[k-1,j]

This creates a tree structure where:

- Each level doubles the span of the prefix computation
- Level k can look back 2^k positions
- The tree has logarithmic depth

Phase 3: Sum Generation

For each bit position i:

- Sum Si = Pi ⊕ Ci-1
- Where Ci-1 is derived from the computed G and P values
- C-1 = 0 (no initial carry input)

Detailed Algorithm Steps for n-bit Addition

Step 1: Input Processing

- Take two n-bit numbers A = (An-1, An-2, ..., A1, A0) and B = (Bn-1, Bn-2, ..., B1, B0)
- Initialize carry input C-1 = 0

Step 2: Initial G and P Computation

```
For i = 0 to n-1:
   G[0][i] = Ai AND Bi
   P[0][i] = Ai XOR Bi
```

Step 3: Tree Construction

```
levels = ceil(log2(n))
For level = 1 to levels:
   step = 2^level
   For i = step-1 to n-1:
```

```
    if (i-step/2) >= 0:
      G[level][i] = G[level-1][i] OR (P[level-1][i] AND G[level-1][i-step/2])
      P[level][i] = P[level-1][i] AND P[level-1][i-step/2]
```

Step 4: Carry Extraction

```
For i = 0 to n-1:
  if i == 0:
    Ci-1 = 0
  else:
    Ci-1 = G[final_level][i-1]
```

Step 5: Sum Calculation

```
For i = 0 to n-1:
  Si = P[0][i] XOR Ci-1
```

Example: 4-bit Kogge-Stone Adder

Let's trace through a 4-bit addition: A = 1011, B = 0110

Step 1: Initialize

- A = [1,0,1,1], B = [0,1,1,0]

Step 2: Compute Initial G and P

- G[0] = [0,0,1,0] (A AND B for each position)
- P[0] = [1,1,0,1] (A XOR B for each position)

Step 3: Build Tree (2 levels for 4-bit)

Level 1:

- G[1][1] = G[0][1] OR (P[0][1] AND G[0][0]) = 0 OR (1 AND 0) = 0
- P[1][1] = P[0][1] AND P[0][0] = 1 AND 1 = 1
- G[1][2] = G[0][2] OR (P[0][2] AND G[0][1]) = 1 OR (0 AND 0) = 1
- P[1][2] = P[0][2] AND P[0][1] = 0 AND 1 = 0

Level 2:

- G[2][3] = G[1][3] OR (P[1][3] AND G[1][1]) = 0 OR (1 AND 0) = 0

- P[2][3] = P[1][3] AND P[1][1] = 1 AND 1 = 1

Step 4: Generate Sum

- S0 = P[0][0] XOR 0 = 1 XOR 0 = 1
- S1 = P[0][1] XOR G[0][0] = 1 XOR 0 = 1
- S2 = P[0][2] XOR G[1][1] = 0 XOR 0 = 0
- S3 = P[0][3] XOR G[2][2] = 1 XOR 1 = 0

Result: Sum = 0011, Carry_out = G[2][3] = 1

Characteristics and Complexity

Advantages

- **Logarithmic Delay**: O(log n) propagation delay
- **Regular Structure**: Suitable for VLSI implementation
- **Predictable Timing**: No variable-length carry chains
- **High Speed**: Fastest among practical adder architectures
- **Scalable**: Performance scales well with bit width

Disadvantages

- **High Area Cost**: O(n log n) gates required
- **Power Consumption**: Significant due to large number of gates
- **Complex Routing**: Irregular connections increase layout complexity
- **Cost**: More expensive than simpler adder architectures

Complexity Analysis

- **Time Complexity**: O(log n)
- **Space Complexity**: O(n log n)
- **Gate Count**: Approximately $2n \log_2 n$ gates
- **Levels**: $\lceil \log_2 n \rceil$ levels in the prefix tree

Applications

The Kogge-Stone adder is particularly valuable in:

- **High-Performance Processors**: Where addition speed is critical
- **Arithmetic Logic Units (ALUs)**: For fast integer operations
- **Floating-Point Units**: For mantissa addition in FP operations
- **Digital Signal Processing**: Where throughput is important

- **Cryptographic Hardware**: For modular arithmetic operations
- **Graphics Processing Units**: For parallel arithmetic operations

Implementation Considerations

Hardware Implementation

- Use of regular cell structures for easier layout
- Careful consideration of wire delays in deep submicron technologies
- Power optimization through clock gating and voltage scaling
- Pipeline register insertion for higher frequency operation

Design Trade-offs

- Area vs. Speed: Kogge-Stone maximizes speed at the cost of area
- Power vs. Performance: Higher transistor count leads to increased power
- Regularity vs. Optimality: Some optimization possible but reduces regularity

Conclusion

The Kogge-Stone adder represents an optimal solution for applications requiring the fastest possible binary addition. While it consumes more area and power compared to simpler adders, its logarithmic delay makes it indispensable in high-performance computing systems where addition speed is the primary constraint. The regular tree structure and predictable timing characteristics make it a popular choice in modern processor designs despite the increased hardware complexity.

Verilog Code

```
module kogge_stone_4bit (
    input  [3:0] a,       // 4-bit input A
    input  [3:0] b,       // 4-bit input B
    input        cin,     // Carry input
    output [3:0] sum,     // 4-bit sum output
    output       cout     // Carry output
);

// Internal signals for Generate and Propagate
wire [3:0] g0, p0;      // Level 0 (initial G and P)
wire [3:0] g1, p1;      // Level 1
wire [3:0] g2, p2;      // Level 2
wire [3:0] c;           // Carry signals

// Level 0: Initial Generate and Propagate computation
assign g0[0] = a[0] & b[0];
assign g0[1] = a[1] & b[1];
assign g0[2] = a[2] & b[2];
assign g0[3] = a[3] & b[3];

assign p0[0] = a[0] ^ b[0];
assign p0[1] = a[1] ^ b[1];
assign p0[2] = a[2] ^ b[2];
assign p0[3] = a[3] ^ b[3];

// Level 1: Span of 2
// Position 0: No change (base case)
assign g1[0] = g0[0];
assign p1[0] = p0[0];

// Position 1: Look back 1 position
assign g1[1] = g0[1] | (p0[1] & g0[0]);
assign p1[1] = p0[1] & p0[0];

// Position 2: Look back 1 position
assign g1[2] = g0[2] | (p0[2] & g0[1]);
assign p1[2] = p0[2] & p0[1];

// Position 3: Look back 1 position
assign g1[3] = g0[3] | (p0[3] & g0[2]);
assign p1[3] = p0[3] & p0[2];

// Level 2: Span of 4
// Position 0,1: No change
assign g2[0] = g1[0];
assign g2[1] = g1[1];
```

```
assign p2[0] = p1[0];
assign p2[1] = p1[1];

// Position 2: Look back 2 positions
assign g2[2] = g1[2] | (p1[2] & g1[0]);
assign p2[2] = p1[2] & p1[0];

// Position 3: Look back 2 positions
assign g2[3] = g1[3] | (p1[3] & g1[1]);
assign p2[3] = p1[3] & p1[1];

// Carry computation
assign c[0] = cin;
assign c[1] = g2[0] | (p2[0] & cin);
assign c[2] = g2[1] | (p2[1] & cin);
assign c[3] = g2[2] | (p2[2] & cin);
assign cout = g2[3] | (p2[3] & cin);

// Sum computation
assign sum[0] = p0[0] ^ c[0];
assign sum[1] = p0[1] ^ c[1];
assign sum[2] = p0[2] ^ c[2];
assign sum[3] = p0[3] ^ c[3];

endmodule
```

Testbench

```verilog
`timescale 1ns / 1ps

module kogge_stone_adder_tb;

    reg [3:0] A, B;
    reg cin;
    wire [3:0] sum;
    wire cout;

    // Instantiate the Unit Under Test (UUT)
    kogge_stone_adder_4bit uut (
        .A(A),
        .B(B),
        .cin(cin),
        .sum(sum),
        .cout(cout)
    );

    initial begin
        $monitor("A=%b, B=%b, cin=%b => sum=%b, cout=%b", A, B, cin, sum, cout);

        A = 4'b0000; B = 4'b0000; cin = 0; #10;
        A = 4'b0011; B = 4'b0101; cin = 0; #10;
        A = 4'b1010; B = 4'b1100; cin = 1; #10;
        A = 4'b1111; B = 4'b1111; cin = 0; #10;
        A = 4'b1001; B = 4'b0110; cin = 1; #10;

        #10 $finish;
    end

endmodule
```

```verilog
module tb_kogge_stone_adder;

    reg clk;
    reg rst;
    reg [3:0] a_in;
    reg [3:0] b_in;
    reg cin_in;
    wire [3:0] sum_out;
    wire cout_out;

    // Instantiate the top module
    kogge_stone_top uut (
        .clk(clk),
        .rst(rst),
        .a_in(a_in),
        .b_in(b_in),
        .cin_in(cin_in),
        .sum_out(sum_out),
        .cout_out(cout_out)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;   // 10ns period
    end

    // Test stimulus
    initial begin
        $dumpfile("kogge_stone_adder.vcd");
        $dumpvars(0, tb_kogge_stone_adder);
```

```
        // Initialize
        rst = 1;
        a_in = 4'b0000;
        b_in = 4'b0000;
        cin_in = 1'b0;

        // Reset
        #10;
        rst = 0;
        #10;

        // Test cases
        $display("Starting Kogge-Stone Adder Tests");
        $display("Time\tA\tB\tCin\tSum\tCout\tExpected");

        // Test case 1: 0 + 0 + 0 = 0
        a_in = 4'b0000; b_in = 4'b0000; cin_in = 1'b0;
        #20;
        $display("%0t\t%b\t%b\t%b\t%b\t%b\t%d", $time, a_in, b_in, cin_in, sum_out, cout_out, a_in + b_in + cin_in);

        // Test case 2: 1 + 1 + 0 = 2
        a_in = 4'b0001; b_in = 4'b0001; cin_in = 1'b0;
        #20;
        $display("%0t\t%b\t%b\t%b\t%b\t%b\t%d", $time, a_in, b_in, cin_in, sum_out, cout_out, a_in + b_in + cin_in);

        // Test case 3: 5 + 3 + 1 = 9
        a_in = 4'b0101; b_in = 4'b0011; cin_in = 1'b1;
        #20;
        $display("%0t\t%b\t%b\t%b\t%b\t%b\t%d", $time, a_in, b_in, cin_in, sum_out, cout_out, a_in + b_in + cin_in);

        a_in = 4'b0111; b_in = 4'b1000; cin_in = 1'b0;
        #20;
        $display("%0t\t%b\t%b\t%b\t%b\t%b\t%d", $time, a_in, b_in, cin_in, sum_out, cout_out, a_in + b_in + cin_in);
        // Test case 5: 15 + 15 + 1 = 31 (overflow case)
        a_in = 4'b1111; b_in = 4'b1111; cin_in = 1'b1;
        #20;
        $display("%0t\t%b\t%b\t%b\t%b\t%b\t%d", $time, a_in, b_in, cin_in, sum_out, cout_out, a_in + b_in + cin_in);
        // Test case 6: 9 + 6 + 0 = 15
        a_in = 4'b1001; b_in = 4'b0110; cin_in = 1'b0;
        #20;
        $display("%0t\t%b\t%b\t%b\t%b\t%b\t%d", $time, a_in, b_in, cin_in, sum_out, cout_out, a_in + b_in + cin_in);
        // Test case 7: 12 + 4 + 1 = 17 (overflow)
        a_in = 4'b1100; b_in = 4'b0100; cin_in = 1'b1;
        #20;
        $display("%0t\t%b\t%b\t%b\t%b\t%b\t%d", $time, a_in, b_in, cin_in, sum_out, cout_out, a_in + b_in + cin_in);
        // Test case 8: 0 + 15 + 1 = 16 (overflow)
        a_in = 4'b0000; b_in = 4'b1111; cin_in = 1'b1;
        #20;
        $display("%0t\t%b\t%b\t%b\t%b\t%b\t%d", $time, a_in, b_in, cin_in, sum_out, cout_out, a_in + b_in + cin_in);

        $display("Test completed");
        #20;
        $finish;
    end
      // Monitor changes
    initial begin
        $monitor("At time %t: A=%b B=%b Cin=%b -> Sum=%b Cout=%b",
                 $time, a_in, b_in, cin_in, sum_out, cout_out);
    end

endmodule
```

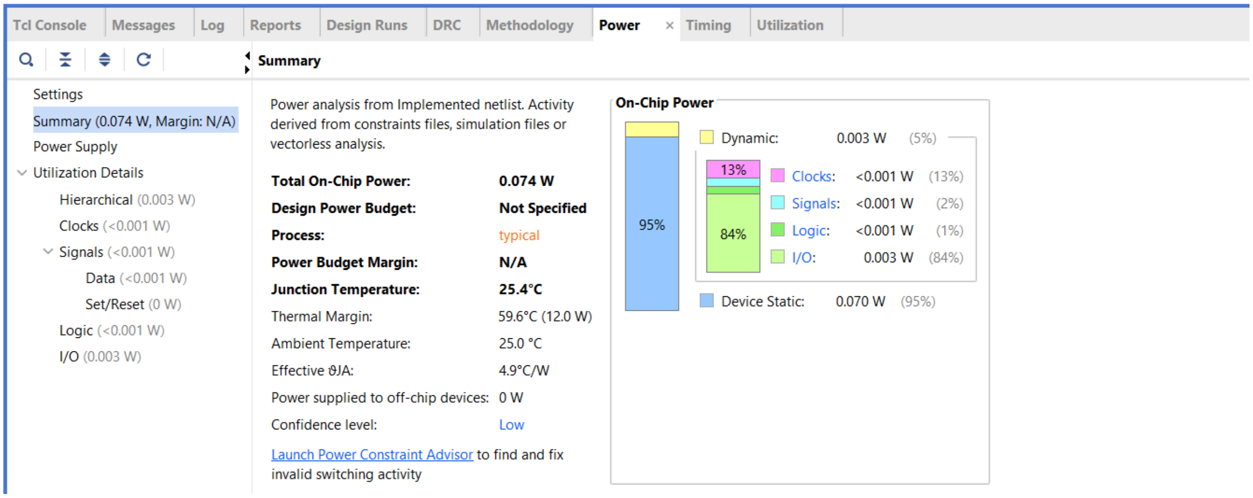
Top Module

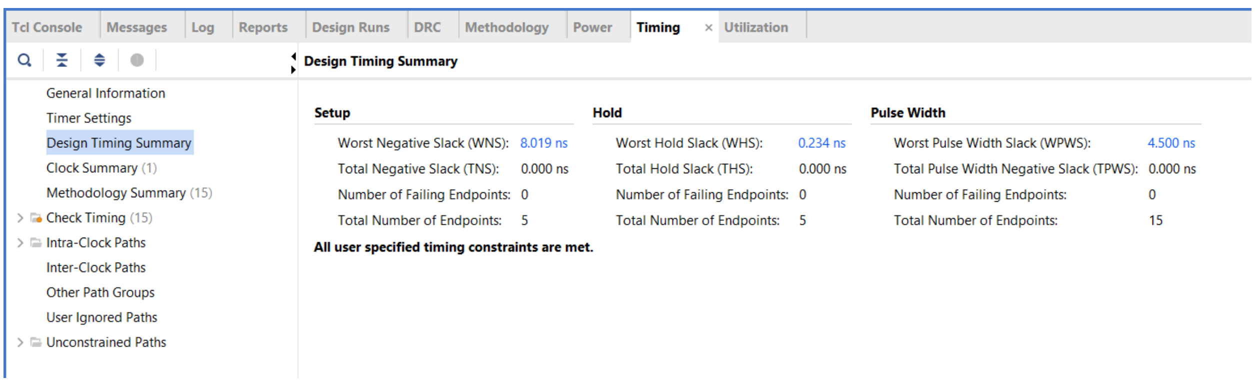
```
module kogge_stone_top (
    input wire clk,
    input wire rst,
    input wire [3:0] a_in,
    input wire [3:0] b_in,
    input wire cin_in,
    output wire [3:0] sum_out,
    output wire cout_out
);

    // Input registers
    wire [3:0] a_reg, b_reg;
    wire cin_reg;

    // Output registers
    reg [3:0] sum_reg;
    reg cout_reg;

    // Input D Flip-Flops
    dff dff_a0 (.clk(clk), .rst(rst), .d(a_in[0]), .q(a_reg[0]));
    dff dff_a1 (.clk(clk), .rst(rst), .d(a_in[1]), .q(a_reg[1]));
    dff dff_a2 (.clk(clk), .rst(rst), .d(a_in[2]), .q(a_reg[2]));
    dff dff_a3 (.clk(clk), .rst(rst), .d(a_in[3]), .q(a_reg[3]));

    dff dff_b0 (.clk(clk), .rst(rst), .d(b_in[0]), .q(b_reg[0]));
    dff dff_b1 (.clk(clk), .rst(rst), .d(b_in[1]), .q(b_reg[1]));
    dff dff_b2 (.clk(clk), .rst(rst), .d(b_in[2]), .q(b_reg[2]));
    dff dff_b3 (.clk(clk), .rst(rst), .d(b_in[3]), .q(b_reg[3]));

    dff dff_cin (.clk(clk), .rst(rst), .d(cin_in), .q(cin_reg));



    // Adder core
    wire [3:0] sum_wire;
    wire cout_wire;

    kogge_stone_4bit adder_core (
        .a(a_reg),
        .b(b_reg),
        .cin(cin_reg),
        .sum(sum_wire),
        .cout(cout_wire)
    );

    // Output D Flip-Flops
    dff dff_sum0 (.clk(clk), .rst(rst), .d(sum_wire[0]), .q(sum_out[0]));
    dff dff_sum1 (.clk(clk), .rst(rst), .d(sum_wire[1]), .q(sum_out[1]));
    dff dff_sum2 (.clk(clk), .rst(rst), .d(sum_wire[2]), .q(sum_out[2]));
    dff dff_sum3 (.clk(clk), .rst(rst), .d(sum_wire[3]), .q(sum_out[3]));

    dff dff_cout (.clk(clk), .rst(rst), .d(cout_wire), .q(cout_out));

endmodule
```

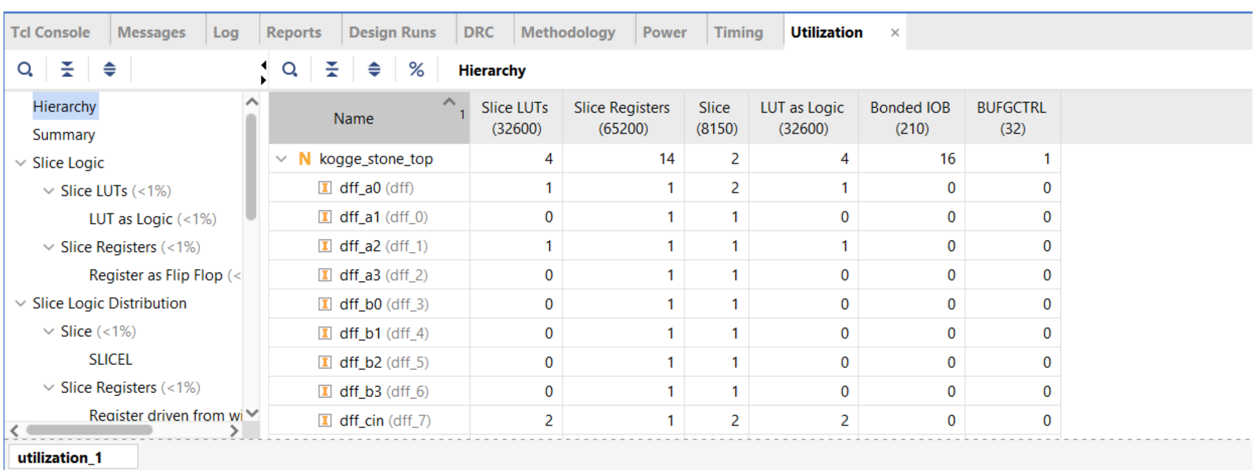
4-bit D FlipFlop

```
`timescale 1ns / 1ps

module kogge_stone_pipelined_4bit (
    input clk,
    input [3:0] A, B,
    input cin,
    output reg [3:0] sum,
    output reg cout
);

reg [3:0] G, P;
reg [3:0] C;

always @(posedge clk) begin
    G <= A & B;
    P <= A ^ B;

    C[0] <= cin;
    C[1] <= G[0] | (P[0] & C[0]);
    C[2] <= G[1] | (P[1] & G[0]) | (P[1] & P[0] & C[0]);
    C[3] <= G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) | (P[2] & P[1] & P[0] & C[0]);

    sum <= P ^ C;
    cout <= G[3] | (P[3] & G[2]) | (P[3] & P[2] & G[1]) | (P[3] & P[2] & P[1] & G[0]) | (P[3] & P[2] & P[1] & P[0] & C[0]);
end

endmodule
```

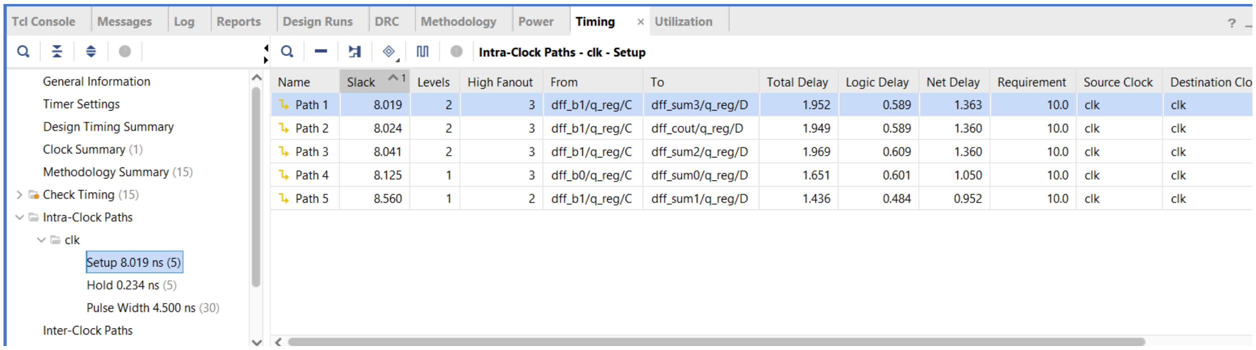
Reports



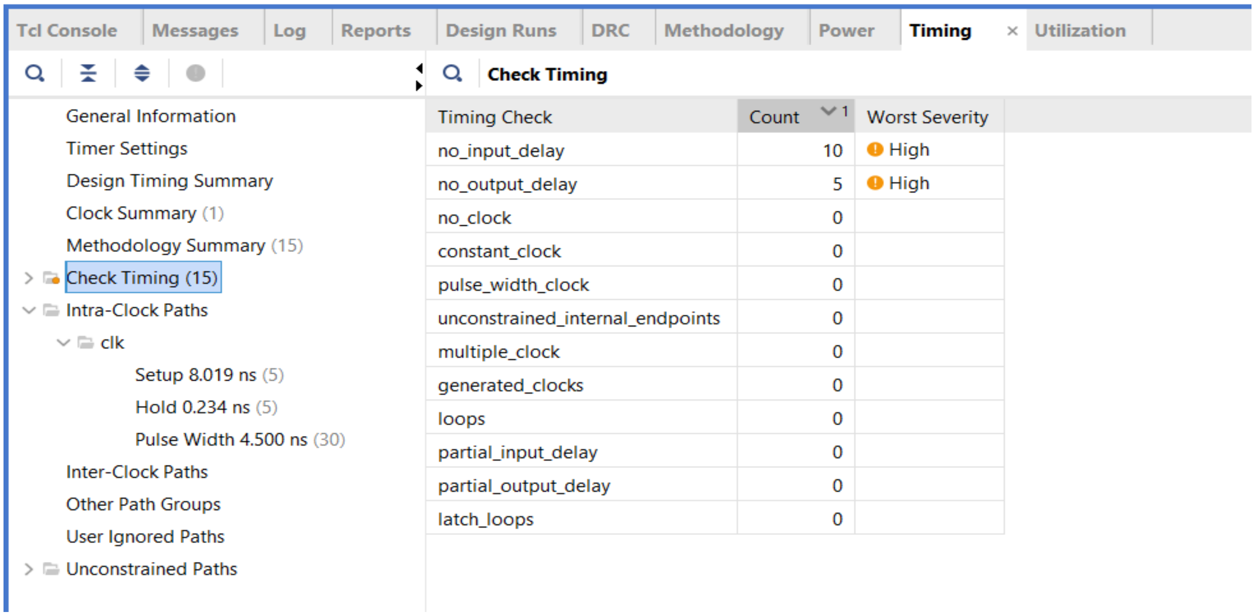




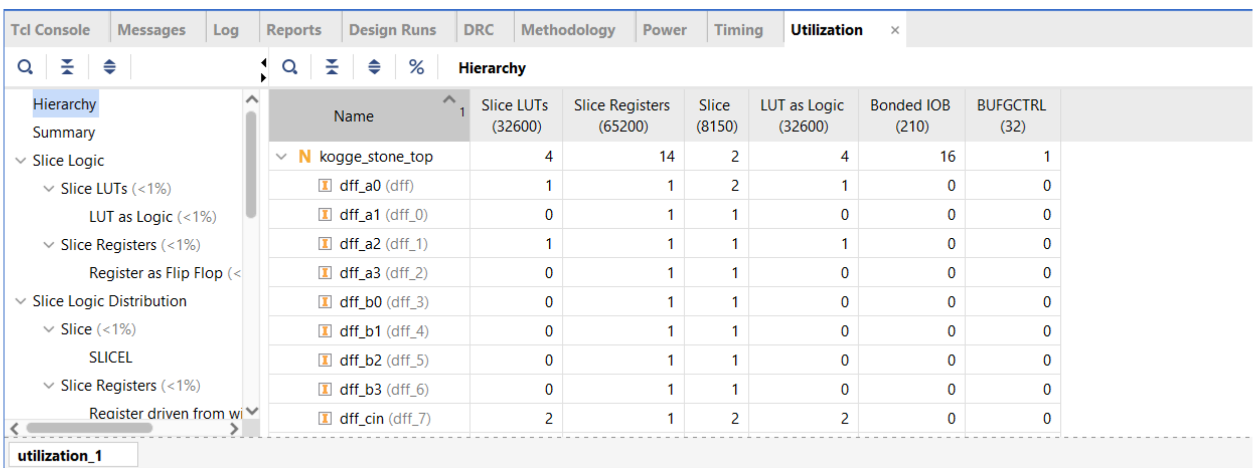
Name	Slice LUTs (32600)	Slice Registers (65200)	Slice (8150)	LUT as Logic (32600)	Bonded IOB (210)	BUFGCTRL (32)
kogge_stone_top	4	14	2	4	16	1
dff_a0 (dff)	1	1	2	1	0	0
dff_a1 (dff_0)	0	1	1	0	0	0
dff_a2 (dff_1)	1	1	1	1	0	0
dff_a3 (dff_2)	0	1	1	0	0	0
dff_b0 (dff_3)	0	1	1	0	0	0
dff_b1 (dff_4)	0	1	1	0	0	0
dff_b2 (dff_5)	0	1	1	0	0	0
dff_b3 (dff_6)	0	1	1	0	0	0
dff_cin (dff_7)	2	1	2	2	0	0



Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clo
Path 1	8.019	2	3	dff_b1/q_reg/C	dff_sum3/q_reg/D	1.952	0.589	1.363	10.0	clk	clk
Path 2	8.024	2	3	dff_b1/q_reg/C	dff_cout/q_reg/D	1.949	0.589	1.360	10.0	clk	clk
Path 3	8.041	2	3	dff_b1/q_reg/C	dff_sum2/q_reg/D	1.969	0.609	1.360	10.0	clk	clk
Path 4	8.125	1	3	dff_b0/q_reg/C	dff_sum0/q_reg/D	1.651	0.601	1.050	10.0	clk	clk
Path 5	8.560	1	2	dff_b1/q_reg/C	dff_sum1/q_reg/D	1.436	0.484	0.952	10.0	clk	clk



Timing Check	Count	Worst Severity
no_input_delay	10	High
no_output_delay	5	High
no_clock	0	
constant_clock	0	
pulse_width_clock	0	
unconstrained_internal_endpoints	0	
multiple_clock	0	
generated_clocks	0	
loops	0	
partial_input_delay	0	
partial_output_delay	0	
latch_loops	0	



Name	Slice LUTs (32600)	Slice Registers (65200)	Slice (8150)	LUT as Logic (32600)	Bonded IOB (210)	BUFGCTRL (32)
kogge_stone_top	4	14	2	4	16	1
dff_a0 (dff)	1	1	2	1	0	0
dff_a1 (dff_0)	0	1	1	0	0	0
dff_a2 (dff_1)	1	1	1	1	0	0
dff_a3 (dff_2)	0	1	1	0	0	0
dff_b0 (dff_3)	0	1	1	0	0	0
dff_b1 (dff_4)	0	1	1	0	0	0
dff_b2 (dff_5)	0	1	1	0	0	0
dff_b3 (dff_6)	0	1	1	0	0	0
dff_cin (dff_7)	2	1	2	2	0	0