Project 24 : Pipelined Adder

A Comprehensive Study of Advanced Digital Circuits

By: Gati Goyal, Nikunj Agrawal, Ayush Jain, Abhishek Sharma

Contents

1 Project Overview

A pipelined adder is an optimized version of an adder that uses pipelining to improve the speed and efficiency of arithmetic operations, particularly in digital circuits and processors. Pipelining is a technique used in computing where a task is broken down into smaller stages, and multiple stages are processed simultaneously in a staggered fashion. This increases throughput by allowing multiple additions to be processed concurrently, rather than waiting for each operation to finish before starting the next one.

2 Pipelined Adder

2.1 Description

Pipelining is a technique used in computer architecture and digital systems to increase the efficiency and throughput of processes by dividing a task into smaller stages, where each stage can operate concurrently on different parts of the input data. This method is inspired by assembly lines in manufacturing, where each worker (or stage) handles a specific task in parallel with others, allowing multiple items to be processed simultaneously.

2.2 Key Concepts of Pipelining:

- **Stages:**

The process is divided into multiple sequential stages, each performing a portion of the task. Each stage is responsible for a distinct part of the operation. In the case of a pipelined adder, for example, stages might include partial sum generation, carry propagation, and final summation.

- **Parallelism:**

While one stage is processing a specific part of the current task, other stages can begin processing the next tasks. This allows different stages to work in parallel, thus improving the overall throughput.

- **Pipeline Registers:**

Between each stage, registers are used to hold intermediate data. These registers ensure that the stages operate independently and store the results from one stage before passing them to the next.

- **Clock Cycles:**

A clock signal synchronizes the operation of the pipeline stages. During each clock cycle, data moves from one stage to the next, allowing multiple tasks to be processed concurrently.

- **Throughput vs. Latency:**

Throughput: The number of tasks (e.g., arithmetic operations) that can be completed in a given time period increases because multiple tasks are processed in parallel. Latency: The total time required to complete a single task might increase because each task needs to pass through multiple stages. However, after the initial setup, new results are produced continuously with each clock cycle.

2.3 RTL Code

Listing 1: Pipelined Adder

```
1
2 module Pipelined_Adder #(parameter WIDTH = 8, STAGES = 4) (
3     input  logic clk,
4     input  logic reset,
5     input  logic [WIDTH-1:0] a,
6     input  logic [WIDTH-1:0] b,
```

```
    output logic [WIDTH-1:0] sum
);

    logic [WIDTH-1:0] pipeline_a [STAGES:0];
    logic [WIDTH-1:0] pipeline_b [STAGES:0];
    logic [WIDTH-1:0] pipeline_sum [STAGES:0];

    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            pipeline_a[0] <= 0;
            pipeline_b[0] <= 0;
            pipeline_sum[0] <= 0;
        end else begin
            pipeline_a[0] <= a;
            pipeline_b[0] <= b;
            pipeline_sum[0] <= a + b;
        end
    end

    genvar i;
    generate
        for (i = 1; i <= STAGES; i++) begin : pipeline_stage
            always_ff @(posedge clk or posedge reset) begin
                if (reset) begin
                    pipeline_a[i] <= 0;
                    pipeline_b[i] <= 0;
                    pipeline_sum[i] <= 0;
                end else begin
                    pipeline_a[i] <= pipeline_a[i-1];
                    pipeline_b[i] <= pipeline_b[i-1];
                    pipeline_sum[i] <= pipeline_sum[i-1];
                end
            end
        end
    endgenerate

    assign sum = pipeline_sum[STAGES];

endmodule
```

2.4 Testbench

Listing 2: Pipelined Adder

```


module tb_Pipelined_Adder;

    parameter WIDTH = 8;
    parameter STAGES = 4;

    logic clk;
    logic reset;
    logic [WIDTH-1:0] a, b;
    logic [WIDTH-1:0] sum;

    // Instantiate the Pipelined Adder
```

```
    Pipelined_Adder #(WIDTH, STAGES) uut (
        .clk(clk),
        .reset(reset),
        .a(a),
        .b(b),
        .sum(sum)
    );

    // Clock generation
    always #5 clk = ~clk;

    // Test Procedure
    initial begin
        $display("Pipelined Adder Test Start");

        clk = 0;
        reset = 1;
        a = 0;
        b = 0;

        // Apply reset
        #10;
        reset = 0;

        // Test case 1
        a = 8'hA; // 10 in decimal
        b = 8'h5; // 5 in decimal
        #50;

        $display("Sum = %d, Expected Sum = %d", sum, a+b);

        // Test case 2
        a = 8'hF; // 15 in decimal
        b = 8'h1; // 1 in decimal
        #50;

        $display("Sum = %d, Expected Sum = %d", sum, a+b);

        // Test case 3
        a = 8'h7; // 7 in decimal
        b = 8'h9; // 9 in decimal
        #50;

        $display("Sum = %d, Expected Sum = %d", sum, a+b);

        $display("Pipelined Adder Test End");
        $finish;
    end

endmodule
```

3 How it works ?

- **Stage Division:** The addition is divided into smaller stages, each handling a part of the operation.
- **Pipeline Registers:** Intermediate results are stored in pipeline registers between stages.

- **Parallel Processing:** Multiple additions are processed simultaneously, with each addition in a different stage of the pipeline.
- **Carry Propagation:** Carry bits are managed and propagated efficiently across stages.
- **Final Result:** After passing through all stages, the final sum is generated, and a new result is produced at each clock cycle once the pipeline is full.

3.1 Simulation

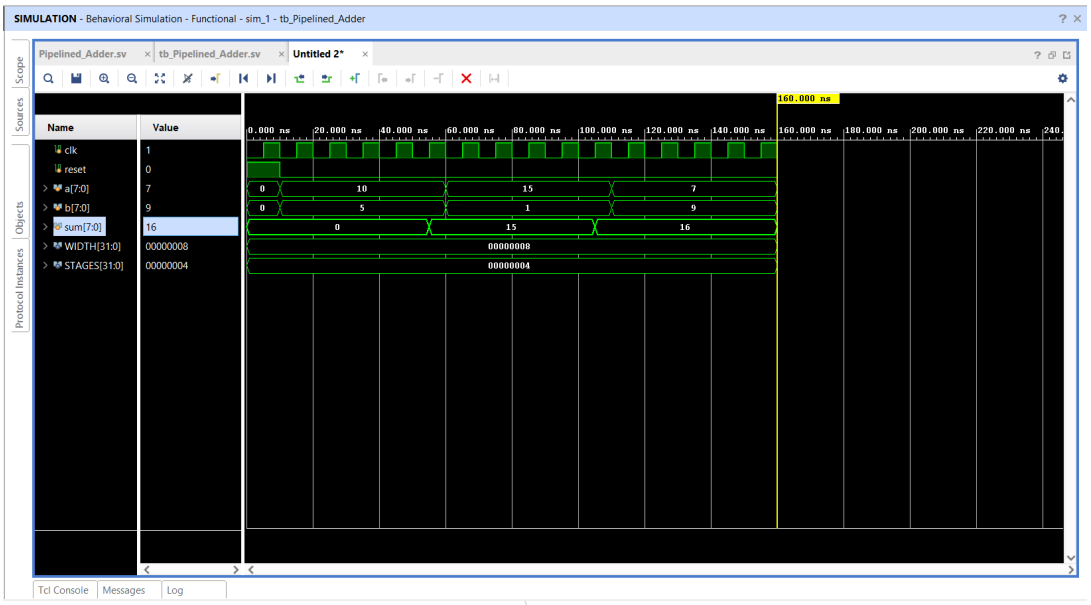

Figure 1: Simulation of Pipelined Adder

3.2 Schematic

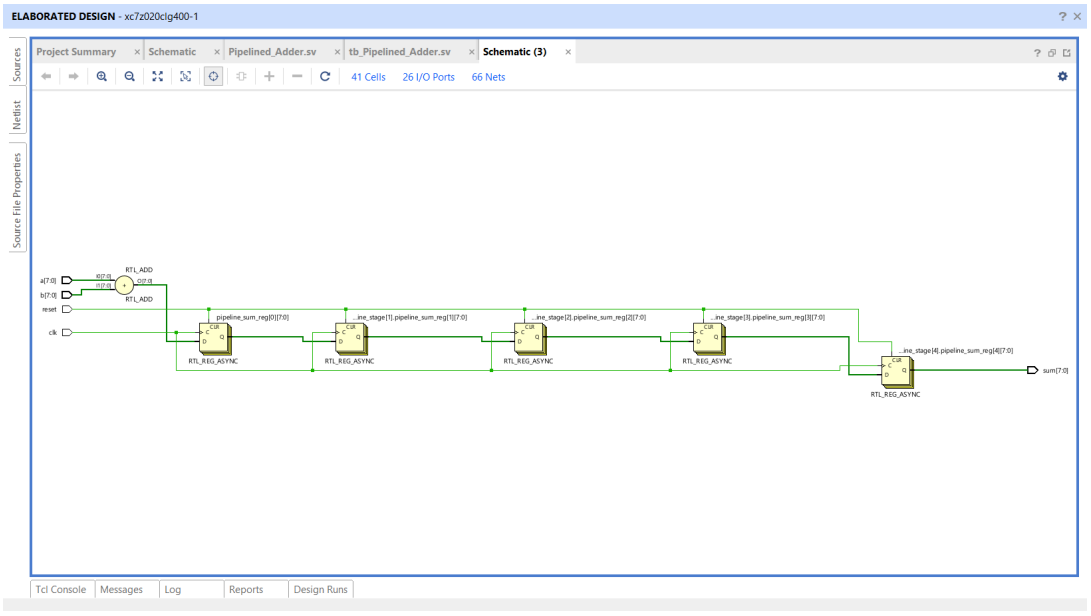

Figure 2: Schematic of Pipelined Adder

Explanation

A pipelined adder is a type of adder that enhances the performance of arithmetic operations by dividing the addition process into multiple stages. Each stage performs a portion of the addition, and different stages can operate concurrently on different input data. This allows for multiple addition operations to be processed in parallel, significantly increasing the throughput of the system.

3.3 Comparison with Other Adders:

1. Pipelined Adder vs. Ripple Carry Adder (RCA):

Pipelined Adder: Divides addition into multiple stages for concurrent processing. High throughput but may have higher latency for individual operations. More complex design due to pipeline management. Ideal for high-performance, repetitive addition tasks.

Ripple Carry Adder (RCA): Sequential carry propagation from one bit to the next. Slow for large bit-width operations. Simple design, but not efficient for high-speed applications. Suitable for smaller bit-widths or low-speed operations.

2. Pipelined Adder vs. Carry-Lookahead Adder (CLA):

Pipelined Adder:

Multiple additions processed in stages for higher throughput. Higher latency for a single addition, but increased overall performance. Used in high-throughput applications like CPUs and DSPs.

Carry-Lookahead Adder (CLA):

Uses parallel logic to compute carry bits quickly. Faster for single, large-bit-width additions. Efficient for single operations with minimal latency. Used in high-speed arithmetic operations within ALUs.

3. Pipelined Adder vs. Carry-Save Adder (CSA):

Pipelined Adder:

Breaks addition into stages, processing multiple additions concurrently. High throughput but requires synchronization of stages. Ideal for tasks requiring high-frequency arithmetic operations.

Carry-Save Adder (CSA):

Adds multiple numbers without immediate carry propagation. Efficient for adding multiple numbers at once but needs extra logic for final carry resolution. Typically used in multipliers and large integer operations.

4. Pipelined Adder vs. Parallel Prefix Adders (Brent-Kung, Kogge-Stone, etc.):

Pipelined Adder:

Processes multiple additions simultaneously in different stages. Higher throughput with some latency for individual operations. Used in high-performance, high-throughput computing environments.

Parallel Prefix Adders:

Compute carries in parallel for fast individual additions. Faster than RCA and CLA for large-bit-width operations. Complex design but optimized for low-latency additions. Used in high-speed processors and FPGAs.

5. Pipelined Adder vs. Modular Adder:

Pipelined Adder:

Focuses on high-throughput, dividing additions into multiple stages. Best suited for repetitive arithmetic tasks requiring fast, continuous operations.

Modular Adder:

Adds numbers and applies a modulus to constrain results within a specific range. Slower due to additional modulus operation.

Used in cryptography, cyclic systems, and applications requiring bounded results.

3.4 Advantages of the Pipelined Adder:

- **Increased Throughput:**

More tasks can be processed in a shorter amount of time since multiple stages are working concurrently.

- **Efficient Resource Utilization:**

Each stage of the pipeline can be optimized for specific subtasks, ensuring that the hardware resources are effectively used.

- **Scalability:**

Pipelining is scalable and can be extended to complex tasks by adding more stages to handle specific sub-operations.

- **Reduced Idle Time:**

Each part of the system is constantly working on some aspect of the process, which minimizes downtime for hardware components

3.5 Disadvantages of the Pipelined Adder:

- **Increased Latency for Individual Tasks:**

A single task may take longer to complete since it needs to pass through several stages, especially in deep pipelines with many stages.

- **Complexity in Design:**

Pipelined systems are more complex to design because the stages need to be carefully coordinated, and additional logic is required to handle synchronization and control.

- **Pipeline Hazards:**

- **Data Hazards:** When one stage depends on the output of a previous stage, which has not yet completed (e.g., in processors, this can cause stalls).

- **Control Hazards:** Branching operations in a processor can cause mispredictions that disrupt the pipeline, requiring the system to discard partially completed tasks.

- **Structural Hazards:** When hardware resources are insufficient to handle the simultaneous demands of different pipeline stages.

- **Startup Latency:**

The initial task needs to pass through all the stages before the pipeline reaches maximum throughput, causing an initial delay (known as pipeline "fill time").

3.6 Applications of the Pipelined Adder:

- **Computer Processors:**

Used in CPUs to improve instruction execution speed by breaking down instruction processing into stages like instruction fetch, decode, execute, memory access, and write-back.

- **Digital Signal Processing (DSP):**

Used in DSP systems to accelerate tasks like filtering, convolution, and Fourier transforms by dividing these operations into stages.

- **Graphics Processing Units (GPUs):**

GPUs use pipelining to handle multiple operations related to rendering, shading, and texture mapping in parallel.

- **Network Routers:**

In networking, pipelining is used to process multiple packets at different stages of routing or error checking simultaneously, enhancing the speed of data transmission.

- **Assembly Lines in Manufacturing:**

In industrial production, pipelining is used to increase the efficiency of manufacturing processes by breaking down tasks into smaller, specialized stages.

4 Results

4.1 Synthesis Design

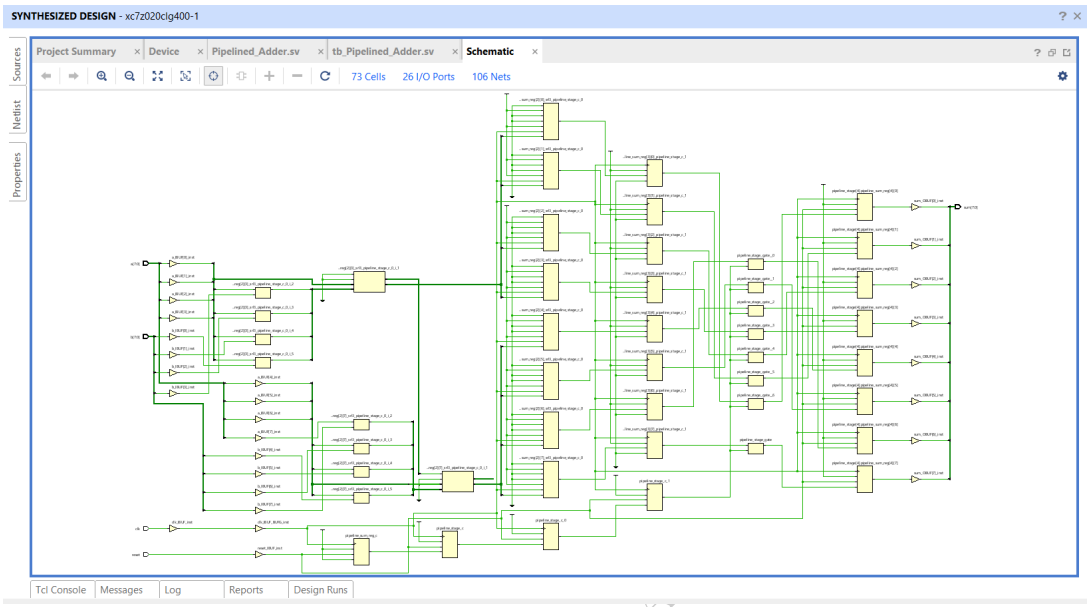

Figure 3: Synthesis Design of Pipelined Adder

5 FAQs

Q1. What does the provided code represent?

Answer: The code defines a pipelined adder using Verilog, where two input signals (a and b) are added over multiple stages. The adder is parameterized by bit-width (WIDTH) and number of pipeline stages (STAGES), which allows flexibility in its configuration.

Q2. What is the purpose of pipelining in this adder?

Answer: Pipelining in this adder breaks the addition process into multiple stages. This increases the throughput by allowing multiple additions to be in progress simultaneously. Each stage holds intermediate results in pipeline registers, ensuring smooth data flow between stages.

Q3. Why are pipeline_a, pipeline_b, and pipeline_sum arrays used in the design?

Answer: These arrays are used to store the values of a, b, and their partial sums at each pipeline stage. The pipeline registers hold intermediate values, allowing data to flow between stages without affecting the overall timing and performance.

Q4. How does this design improve performance compared to a simple adder?

Answer: In a simple adder, the entire addition operation must be completed within a single clock cycle. A pipelined adder, however, breaks the addition into smaller parts, spreading it over multiple clock cycles. This allows new inputs to be processed every clock cycle after the pipeline is filled, increasing throughput significantly.

Q5. What would happen if you removed the generate block and implemented the pipeline stages manually?

Answer: If the generate block is removed, you would need to manually replicate the pipeline stages by explicitly writing out each stage's logic. While this works, it would make the code more cumbersome, less flexible, and harder to modify for different pipeline depths (STAGES). The generate block simplifies the design and makes it scalable.

Q6. How does this pipelined adder handle latency?

Answer: The latency of this pipelined adder depends on the number of pipeline stages (STAGES). Each addition result takes STAGES clock cycles to propagate from the input to the final output sum. However, once the pipeline is filled, a new result is produced every clock cycle, improving throughput.

Q7. What are the advantages of using this pipelined adder in a high-performance system?

Answer:

Increased throughput: By processing different additions in different stages concurrently, it allows high-speed addition operations.

Scalability: The design can be easily scaled by adjusting the STAGES and WIDTH parameters. Optimized for clock speed: Pipelining allows each stage to run at higher clock speeds since each stage performs a smaller portion of the operation.

Q8. What do the parameters WIDTH and STAGES define in the module?

Answer:

WIDTH: Defines the bit-width of the input operands a and b and the result sum. This determines how many bits each input will be.

STAGES: Specifies the number of pipeline stages used in the adder. It controls how many intermediate stages are used to perform the addition, affecting both latency and throughout.