Design and Implementation of INSERTION SORT vs. BUBBLE SORT

November 25, 2024

Charvi Pahuja (2023CSB1114) ,
Tamanna (2023CSB1169) ,
Krishna Agarwal (2023CSB1131) ,
Atul Kharat (2023CSB1105) ,
Nachiket Patil (2023CSB1136) ,
Aadit Mahajan (2023CSB1091)

Instructor:
Dr. Geeta

Summary: Sorting algorithms are fundamental to computer science and find applications in diverse domains, including database management, data analysis, and embedded systems. In this project, Bubble Sort and Insertion Sort algorithms were designed and implemented in hardware using Verilog. By leveraging a bottom-up approach, the algorithms were built from basic modules, tested, and analyzed for performance metrics such as gate count and delay.
The comparative study aims to provide insights into the computational efficiency and resource utilization of these algorithms, offering recommendations for hardware-based sorting in different scenarios.
This report presents the design and implementation of Bubble Sort and Insertion Sort algorithms in Verilog. Using a bottom-up design approach, the project emphasizes efficient hardware implementation. The performance of the algorithms is evaluated based on gate count and delay. A comparative analysis is provided, offering insights into the suitability of each algorithm for different applications.

---

1. Introduction

Sorting is a fundamental operation in computer science, with applications in data processing, searching, and optimization. Bubble Sort and Insertion Sort are two classic sorting algorithms, characterized by simplicity and distinct computational characteristics.
The goal of this project is to implement these algorithms as hardware modules in Verilog, analyze their efficiency in terms of hardware resources and timing, and provide a detailed comparison. A structured bottom-up design approach ensures modularity and reusability, starting with a basic comparator module.

1.1. Objectives

- Design a comparator module for sorting elements.
- Implement Bubble Sort and Insertion Sort in Verilog.
- Evaluate performance metrics: gate count and delay.
- Perform a comparative analysis of the two sorting algorithms.

1.2. Methodology

The project adopts a bottom-up design approach:

1. **Comparator Module:** The foundational building block.
2. **Sorting Modules:** Using the comparator, Bubble Sort and Insertion Sort modules are implemented.
3. **Testbench:** Simulation and verification of functionality using test arrays.
4. **Performance Analysis:** Evaluate gate count and delay for both algorithms.

2. Design and Implementation

2.1. Comparator Module

2.1.1 1-bit Comparator

A comparator is a digital circuit that compares two binary inputs A and B. It takes the following inputs and produces the corresponding outputs:

- **Inputs:**
  - A: A 1-bit binary number.
  - B: A 1-bit binary number.
- **Outputs:**
  - $A > B$: g = High (1) if A is greater than B; otherwise, Low (0).
  - $A = B$: e = High (1) if A equals B; otherwise, Low (0).
  - $A < B$: l = High (1) if A is less than B; otherwise, Low (0).

2.1.2 Cascading n-bit Comparator

The n-bit comparator uses a cascading approach, where a chain of 1-bit comparators evaluates two n-bit numbers (A and B). The comparison starts from the LSB, propagating the results (g, e, l) to MSB. The 1-bit comparators are connected sequentially:

• MSB priority: If MSB indicates A > B or A < B, the decision propagates, ignoring lower bits.

• Intermediate comparison results (g, e, l) are passed stage by stage until the LSB.

Equation and Diagram for Comparator Logic:

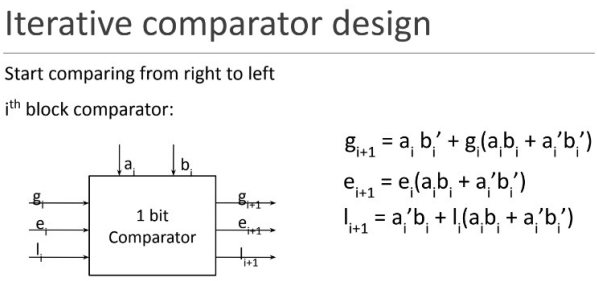


Figure 1: **Reference:** Dr. Geeta's slides

```verilog
module comparatorbit1 (
    input a_bit, b_bit, g_in, e_in, l_in,  // Inputs
    output g_out, e_out, l_out           // Outputs
);
    wire not_a, not_b, and1, and2, and3, or1, and4, and5, and6, and7, or2;

    // Invert inputs
    not (not_a, a_bit);
    not (not_b, b_bit);

    // Greater-than logic
    and (and1, a_bit, not_b);             // a_i b'_i
    and (and2, a_bit, b_bit);             // a_i b_i
    and (and3, not_a, not_b);             // a'_i b'_i
    or  (or1, and2, and3);                // (a_i b_i + a'_i b'_i)
    and (and4, g_in, or1);                // g_i (a_i b_i + a'_i b'_i)
    or  (g_out, and1, and4);              // g_{i+1} = a_i b'_i + g_i (a_i b_i + a'_i b'_i)

    // Equal-to logic
    and (and5, e_in, or1);                // e_i (a_i b_i + a'_i b'_i)
    assign e_out = and5;                  // e_{i+1}

    // Less-than logic
    and (and6, not_a, b_bit);             // a'_i b_i
    and (and7, l_in, or1);                // l_i (a_i b_i + a'_i b'_i)
    or  (l_out, and6, and7);              // l_{i+1} = a'_i b_i + l_i (a_i b_i + a'_i b'_i)
endmodule
```

Figure 2: Verilog code for 1-bit comparator

```verilog
module comparatorbitN #(parameter N = 4) (
    input [N-1:0] A, B,       // n-bit inputs
    output G, E, L            // Outputs: G (A > B), E (A == B), L (A < B)
);

    wire [N:0] g_bus, e_bus, l_bus; // Propagation buses

    // Initial conditions
    assign g_bus[0] = 0;   // g_0 = 0
    assign e_bus[0] = 1;   // e_0 = 1
    assign l_bus[0] = 0;   // l_0 = 0

    genvar i;
    generate
        for (i = 0; i < N; i = i + 1) begin : COMPARE_STAGE
            comparatorbit1 stage (  // Change here from comparator_1bit to comparatorbit1
                .a_bit(A[i]),
                .b_bit(B[i]),
                .g_in(g_bus[i]),
                .e_in(e_bus[i]),
                .l_in(l_bus[i]),
                .g_out(g_bus[i+1]),
                .e_out(e_bus[i+1]),
                .l_out(l_bus[i+1])
            );
        end
    endgenerate

    // Final outputs
    assign G = g_bus[N];
    assign E = e_bus[N];
    assign L = l_bus[N];

endmodule
```

Figure 3: Verilog code for n-bit comparator

A	B	$Outputs$		
		$G(A > B)$	$E(A = B)$	$L(A < B)$
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

Table 1: Truth table

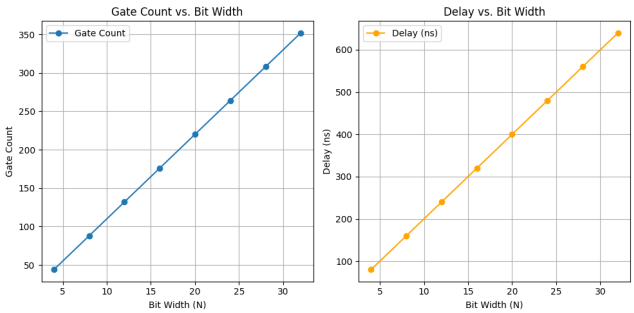
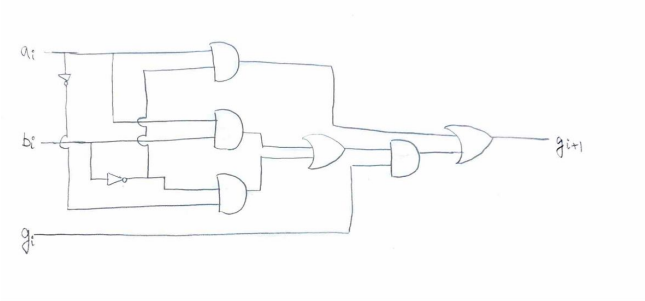
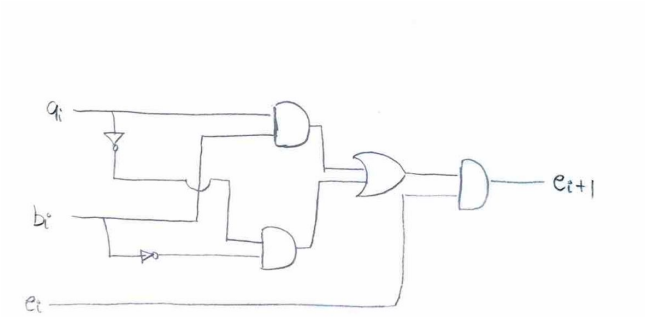
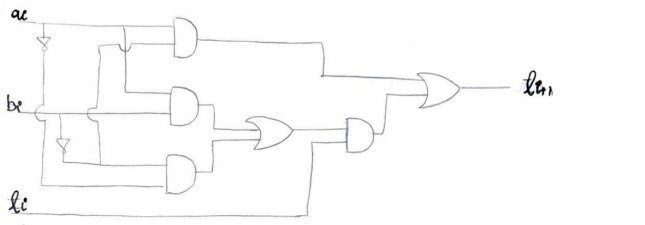


Figure 4: Relation between Number of gates and number of bits in number along with Time delay vs number of bits in number







2.2. Bubble Sort Implementation

Bubble Sort repeatedly compares adjacent elements, swapping them if necessary. Hardware-based Bubble Sort algorithm using a hierarchical design approach in Verilog. The design utilizes a custom bitwise comparator to compare and swap elements, achieving the sorting operation for an array of fixed-size integers.

Steps involved:

1. Loop through the array.
2. For each element, compare it with the next using the comparator.
3. Swap elements if they are out of order.

Pseudocode: The implemented Bubble Sort algorithm operates bottom-up by iteratively comparing and swapping adjacent elements. The pseudo-code is as follows:

```
Input: array_in[N] // Input array of SIZE elements, each N-bits wide
Output: array_out[N] // Sorted array

1. Convert the input into a 2D array representation
2. Repeat SIZE-1 times:
    For j = SIZE-1 to i:
        Load adjacent elements A[j] and A[j-1]
        If A[j] < A[j-1]: Swap A[j] and A[j-1]
3. Flatten the sorted array into output format
```

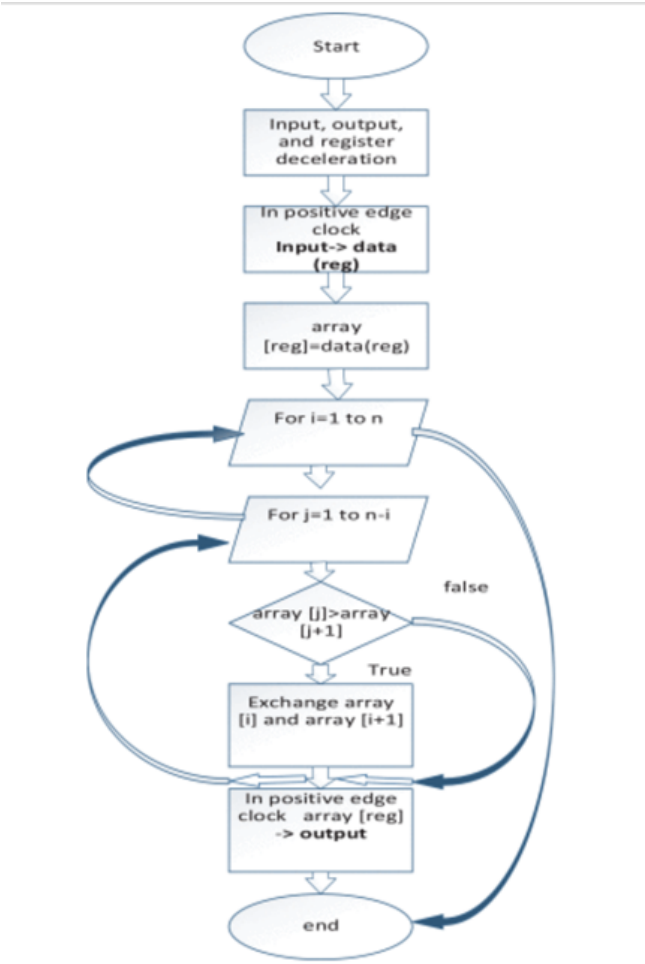


Figure 5: Flowchart for Bubble Sort Implementation

```
module BubbleSort_BottomUp #(parameter N = 16, parameter SIZE = 5) (
    input [N*SIZE-1:0] array_in,
    output reg [N*SIZE-1:0] array_out
);
    integer i, j;
    reg [N-1:0] array [0:SIZE-1];  // Array to hold individual elements
    reg [N-1:0] temp;              // Temporary variable for swapping
    wire greater, equal, less;     // Comparator outputs
    reg [N-1:0] a_reg, b_reg;      // Registers to hold elements being compared

    comparatorbitN #(N) comparator (
        .A(a_reg),           // A port connected to a_reg
        .B(b_reg),           // B port connected to b_reg
        .G(greater),         // Greater-than output
        .E(equal),           // Equal output
        .L(less)             // Less-than output
    );

    always @(array_in) begin
        // Convert the input array to a 2D register array for easier processing
        for (i = 0; i < SIZE; i = i + 1) begin
            array[i] = array_in[N*i +: N];
        end

        // Bubble Sort algorithm: Bottom-Up approach
        for (i = 0; i < SIZE-1; i = i + 1) begin
            for (j = SIZE-1; j > i; j = j - 1) begin
                a_reg = array[j];    // Load current element to compare
                b_reg = array[j-1];  // Load the previous element
                #1;                  // Allow comparator to evaluate

                // If 'less' signal is high, swap elements
                if (less) begin
                    // Swap the elements
                    temp = array[j];
                    array[j] = array[j-1];
                    array[j-1] = temp;
                end
                #1; // Ensure that the swap logic is applied after comparator evaluation
            end
        end

        // Flatten the sorted array back to the output format
        for (i = 0; i < SIZE; i = i + 1) begin
            array_out[N*i +: N] = array[i];
        end
    end
endmodule
```

Figure 6: Verilog code for Bubble Sort

2.3. Insertion Sort Implementation

Insertion Sort iteratively places each element in its correct position in the sorted portion of the array.The Insertion Sort algorithm processes the input array by maintaining a sorted subarray and inserting each subsequent element into its correct position within the subarray.

Steps involved:

1. Iterate through the array.
2. For each element, find its correct position in the sorted part.
3. Insert the element and shift the remaining elements.

Pseudocode: The implemented Insertion Sort algorithm operates bottom-up by iterating through the array and placing element into its sorted place and shifting rest of the elements. The pseudo-code is as follows:

```
InsertionSort(array):
    for i = 1 to SIZE-1 do:
        key = array[i]
        j = i - 1
        while j >= 0 and array[j] > key do:
            array[j + 1] = array[j]
            j = j - 1
        array[j + 1] = key
```

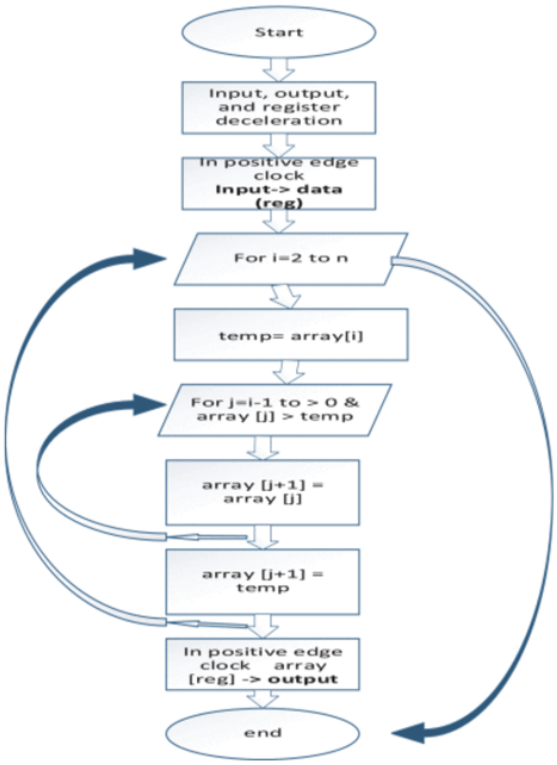


Figure 7: Flowchart for Insertion Sort Implementation

```
module InsertionSort #(parameter N = 16, parameter SIZE = 5) (
    input [N*SIZE-1:0] array_in,
    output reg [N*SIZE-1:0] array_out
);
    integer i, j;
    reg [N-1:0] array [0:SIZE-1];  // Array to hold individual elements
    reg [N-1:0] key;               // Key element for insertion
    wire greater, equal, less;     // Comparator outputs
    reg [N-1:0] a_reg, b_reg;      // Registers to hold elements being compared
    reg done;                      // Flag to exit while loop

    comparatorbitN #(N) comparator (
        .A(a_reg),           // A port connected to a_reg
        .B(b_reg),           // B port connected to b_reg
        .G(greater),         // Greater-than output
        .E(equal),           // Equal output
        .L(less)             // Less-than output
    );

    always @(array_in) begin
        // Convert the input array to a 2D register array for easier processing
        for (i = 0; i < SIZE; i = i + 1) begin
            array[i] = array_in[N*i +: N];
        end

        // Insertion Sort algorithm
        for (i = 1; i < SIZE; i = i + 1) begin
            key = array[i];          // Take the key element
            j = i - 1;
            done = 0;

            // Shift elements of array[0..i-1], that are greater than key,
            // to one position ahead of their current position
            while (j >= 0 && !done) begin
                a_reg = array[j];    // Load the current element
                b_reg = key;         // Compare with the key element
                #1;                  // Allow comparator to evaluate

                if (greater) begin
                    array[j + 1] = array[j]; // Shift element to the right
                    j = j - 1;
                end else begin
                    done = 1;        // Exit the loop
                end
                #1; // Ensure comparator evaluation is completed
            end
            array[j + 1] = key;      // Place the key element in the correct position
        end

        // Flatten the sorted array back to the output format
        for (i = 0; i < SIZE; i = i + 1) begin
            array_out[N*i +: N] = array[i];
        end
    end
endmodule
```

Figure 8: Verilog code for Insertion Sort

2.4. TestBench

The testbench is used to validate the functionality of both sorting algorithms. Test cases include:

1. Sorted array.
2. Reverse sorted array.
3. Random array.

3. Performance Analysis

3.1. Comparator Module

3.1.1 Gate Count

- **1-Bit Comparator** (`comparatorbit1`)
  The `comparatorbit1` module uses the following gates:
  - **NOT gates:** 2
  - **AND gates:** 6
  - **OR gates:** 3

  The total number of gates in a 1-bit comparator is:

$$\text{Total gates per comparator} = 2 + 6 + 3 = 11 \text{ gates.}$$

- **N-Bit Comparator** (`comparatorbitN`)
  The `comparatorbitN` module consists of a cascade of 1-bit comparators. The total number of 1-bit comparators required is equal to the bit width N.

Since each 1-bit comparator uses 11 gates, the total gate count for an N-bit comparator is:

$$\text{Total gate count for an } N\text{-bit comparator} = 11 \times N$$

where N is the number of bits.

3.1.2 Time Delay

The parameterized comparator processes inputs iteratively, starting from the least significant bit (MSB) to the most significant bit (LSB). Each 1-bit comparator contributes its delay sequentially.

1-Bit Comparator Delay (`comparatorbit1`)

The delay introduced by the 1-bit comparator depends on the critical path, which includes the following:

- **NOT gates:** $2 \cdot \Delta_n$, where Δ_n is the delay of a NOT gate.
- **AND gates:** $6 \cdot \Delta_a$, where Δ_a is the delay of an AND gate.
- **OR gates:** $3 \cdot \Delta_o$, where Δ_o is the delay of an OR gate.

The total delay for a 1-bit comparator is:

$$\text{Delay for 1-bit comparator} = (2 \cdot \Delta_n) + (6 \cdot \Delta_a) + (3 \cdot \Delta_o)$$

N-Bit Comparator Delay (`comparatorbitN`)

Since the 1-bit comparators are cascaded sequentially, the total delay for the N-bit comparator is:

$$\text{Total delay for } N\text{-bit comparator} = N \cdot [(2 \cdot \Delta_n) + (6 \cdot \Delta_a) + (3 \cdot \Delta_o)]$$

where:

- N: Number of bits in the comparator.
- Δ_n: Delay of a NOT gate.
- Δ_a: Delay of an AND gate.
- Δ_o: Delay of an OR gate.

3.2. Bubble Sort

3.2.1 Gate Count

Sorting $SIZE$ elements requires comparing adjacent elements $(SIZE - 1)$ times. For each comparison:

$$\text{Comparisons per iteration} = SIZE - i - 1 \text{ (where } i \text{ is the iteration index)}$$

Hence, total comparisons:

$$\text{Total Comparisons} = \sum_{i=0}^{SIZE-1} (SIZE - i - 1) = \frac{SIZE^2}{2}$$

For each comparison, the `comparatorbitN` module is invoked once, resulting in a gate count of:

$$\text{Total gates for sorting} = 11 \times N \times \frac{SIZE^2}{2}$$

3.2.2 Time Delay

The delay through the `comparatorbitN` module is proportional to N due to cascading, and each iteration of the Bubble Sort has a delay given by:

$$\text{Delay per iteration} = (SIZE - i - 1) \cdot N \cdot (2 \cdot \Delta_n) + (6 \cdot \Delta_a) + (3 \cdot \Delta_o)$$

where Δ is the delay of a single gate.
Total delay for sorting:

$$\text{Total Delay} = (2 \cdot \Delta_n) + (6 \cdot \Delta_a) + (3 \cdot \Delta_o) \cdot N \cdot \sum_{i=0}^{SIZE-1} (SIZE - i - 1) = (2 \cdot \Delta_n) + (6 \cdot \Delta_a) + (3 \cdot \Delta_o) \cdot N \cdot \frac{SIZE^2}{2}$$

3.3. Insertion Sort

3.3.1 Gate Delay

For an `N`-bit comparator:

$$\text{Gate Count} = N \times (2 + 7 + 2) = 11N$$

For a `SIZE`-element array in the Insertion Sort algorithm, the comparator is invoked approximately:

$$\text{Total Comparisons} = \frac{\text{SIZE} \times SIZE - 1}{2}$$

Thus, the overall gate count is:

$$\text{Gate Count (Total)} = 11N \times \frac{\text{SIZE} \times SIZE - 1}{2}$$

3.3.2 Time Delay

The critical path delay is determined by the longest chain of operations:

- Comparator delay: $t_{NOT} + 3 \cdot t_{AND} + 2 \cdot t_{OR}$
- Multi-bit comparator: $N \times$ (Comparator Delay)

For Insertion Sort, assuming sequential execution of comparisons and element shifts:

$$\text{Delay (Total)} = \frac{\text{SIZE} \times SIZE - 1}{2} \times \text{Comparator Delay} = \frac{\text{SIZE} \times SIZE - 1}{2} \times (2 \cdot \Delta_n) + (6 \cdot \Delta_a) + (3 \cdot \Delta_o)$$

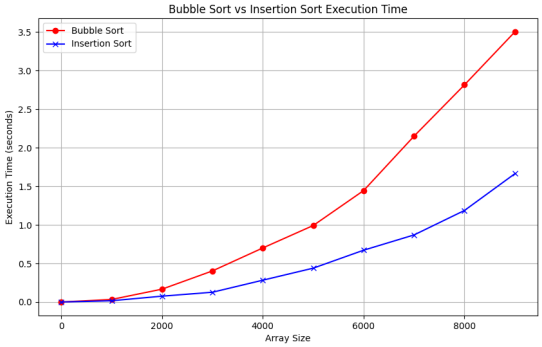


Figure 9: Time Analysis of the Sorting algorithms

3.4. Comparative Analysis

- **Gate Count:**

Module	Bubble Sort	Insertion Sort
Comparator	$11 \cdot N$	$11 \cdot N$
Control Logic	$11 \cdot N \cdot \frac{\text{SIZE}^2}{2}$	$11 \cdot N \cdot \frac{\text{SIZE} \cdot (\text{SIZE}-1)}{2}$

Table 2: Gate Count Comparison

- **Time Delay:** makecell

Module	Bubble Sort	Insertion Sort
Comparator Delay	$N \cdot \left[(2 \cdot \Delta_n) + (6 \cdot \Delta_a) + (3 \cdot \Delta_o)\right]$	$N \cdot \left[(2 \cdot \Delta_n) + (6 \cdot \Delta_a) + (3 \cdot \Delta_o)\right]$
Control Logic Delay	$\frac{\text{SIZE}^2}{2} \cdot N \cdot \left[(2 \cdot \Delta_n) + (6 \cdot \Delta_a) + (3 \cdot \Delta_o)\right]$	$\frac{\text{SIZE} \cdot (\text{SIZE}-1)}{2} \cdot \left[(2 \cdot \Delta_n) + (6 \cdot \Delta_a) + (3 \cdot \Delta_o)\right]$

Table 3: Delay Analysis of Bubble Sort and Insertion Sort

- **Efficiency:** Bubble Sort requires more comparisons and swaps, leading to higher gate count and delay.
- **Resource Utilization:** Insertion Sort is more efficient in terms of gate count.
- **Timing Performance:** Insertion Sort exhibits a shorter critical path, making it faster.

4. Conclusions

This project demonstrates the hardware implementation of Bubble Sort and Insertion Sort using a bottom-up design approach. Insertion Sort outperforms Bubble Sort in terms of gate count and delay, making it more suitable for hardware implementations where resource efficiency is critical. Future work could explore optimizations or implement advanced sorting algorithms like Merge Sort.

References

[1] R. Abirami. VHDL Implementation of Merge Sort Algorithm. *International Journal of Computer Science and Communication Engineering*, 3(2):15–18, 2014.

[2] P. Bellström and C. Thoren. Learning How to Program through Visualization: A Pilot Study on the Bubble Sort Algorithm. In *2nd International Conference Applications of Digital Information and Web Technologies*, pages 90–94, 2009.

Scenario	Preferred Algorithm	Reason
Small arrays	Bubble Sort	Simpler comparator logic.
Nearly sorted arrays	Insertion Sort	Minimal shifts needed.
Energy efficiency needed	Insertion Sort	Fewer gates for shift-based operations.
Uniform latency requirements	Bubble Sort	Fixed delay per iteration.

Table 4: Scenarios where Bubble Sort and Insertion Sort excel.

[3] R. Edjlal, A. Edjlal, and T. Moradi. A Sort Implementation Comparing with Bubble Sort and Selection Sort. In *International Conference on Computer Research and Development*, pages 380–381, 2011.

[4] W. Min. Analysis on Bubble Sort Algorithm Optimization. In *International Forum on Information Technology and Applications*, pages 208–211, 2010.
[1] [2] [3] [4]